Computer Science III

Dr. Chris Bourke
Department of Computer Science & Engineering
University of Nebraska—Lincoln
Lincoln, NE 68588, USA
http://cse.unl.edu/~cbourke
cbourke@cse.unl.edu

2016/08/05 21:48:04
Version 1.2.0

These are lecture notes used in CSCE 310 (Data Structures & Algorithms) at the University of Nebraska—Lincoln.

Contents

List of Algorithms

List of Code Samples

List of Figures

1 Introduction

These lecture notes assume that you are already familiar with the following topics:

- Mastery over at least one high-level programming language
- Working knowledge of algorithm design and analysis
- Familiarity with design and analysis of recursive algorithms
- Working knowledge of Big-O notation and proof techniques involving asymptotics
- Familiarity with basic data structures such as lists, stacks, queues, binary search trees

Nevertheless, this section serves as a high-level review of these topics.

2 Algorithm Analysis

2.1 Introduction

An *algorithm* is a procedure or description of a procedure for solving a problem. An algorithm is a step-by-step specification of operations to be performed in order to compute an output, process data, or perform a function. An algorithm must always be *correct* (it must always produce a valid output) and it must be *finite* (it must terminate after a finite number of steps).

Algorithms are not code. Programs and code in a particular language are *implementations* of algorithms. The word, "algorithm" itself is derived from the latinization of Abū 'Abdalāh Muhammad ibn Mūsā al-Khwārizmī, a Persian mathematician (c. 780 – 850). The concept of algorithms predates modern computers by several thousands of years. Euclid's algorithm for computing the greatest common denominator (see Section 2.5.3) is 2,300 years old.

Often, to be useful an algorithm must also be *feasible*: given its input, it must execute in a reasonable amount of time using a reasonable amount of resources. Depending on the application requirements our tolerance may be on the order of a few milliseconds to several days. An algorithm that takes years or centuries to execute is certainly not considered feasible.

Deterministic An algorithm is deterministic if, when given a particular input, will always go through the exact same computational process and produce the same output. Most of the algorithms you've used up to this point are deterministic.

Randomized An algorithm is randomized is an algorithm that involves some form of random input. The random source can be used to make decisions such as random selections or to generate random state in a program as candidate solutions. There are many types of randomized algorithms including Monte-Carlo algorithms (that may have some error with low probability), Las Vagas algorithms (whose results are always correct, but may fail with a certain probability to produce any results), etc.

Optimization Many algorithms seek not only to find a solution to a problem, but to find the *best*, optimal solution. Many of these type of algorithms are *heuristics*: rather than finding the actual best solution (which may be infeasible), they can approximate a solution (*Approximation* algorithms). Other algorithms simulate biological processes (Genetic algorithms, Ant Colony algorithms, etc.) to search for an optimal solution.

Parallel Most modern processors are multicore, meaning that they have more than one processor on a chip. Many servers have dozens of processors that work together. Multiple processors can be utilized by designing parallel algorithms that can split work across multiple processes or threads which can be executed in parallel to each other, improving overall performance.

Distributed Computation can also be distributed among completely separate devices that may be located half way across the globe. Massive distributed computation networks have been built for research such as simulating protein folding (Folding@Home).

An algorithm is a more abstract, generalization of what you might be used to in a typical programming language. In an actual program, you may have functions/methods, subroutines or procedures, etc. Each one of these pieces of code could be considered an algorithm in and of itself. The combination of these smaller pieces create more complex algorithms, etc. A program is essentially a concrete implementation of a more general, theoretical algorithm.

When a program executes, it expends some amount of resources. For example:

Time The most obvious resource an algorithm takes is time: how long the algorithm takes to finish its computation (measured in seconds, minutes, etc.). Alternatively, time can be measured in how many CPU cycles or floating-point operations a particular piece of hardware takes to execute the algorithm.

Memory The second major resource in a computer is memory. An algorithm requires memory to store the input, output, and possibly extra memory during its execution. How much memory an algorithm uses in its execution may be even more of an important consideration than time in certain environments or systems where memory is extremely limited such as embedded systems.

Power The amount of power a device consumes is an important consideration when you have limited capacity such as a battery in a mobile device. From a consumer's perspective, a slower phone that offered twice the batter life may be preferable. In certain applications such as wireless sensor networks or autonomous systems power may be more of a concern than either time or memory.

Bandwidth In computer networks, efficiency is measured by how much data you can transmit from one computer to another, called *throughput*. Throughput is generally limited by a network's bandwidth: how much a network connection can transmit under ideal circumstances (no data loss, no retransmission, etc.)

Circuitry When designing hardware, resources are typically measured in the number of gates or wires are required to implement the hardware. Fewer gates and wires means you can fit more chips on a silicon die which results in cheaper hardware. Fewer wires and gates also means faster processing.

Idleness Even when a computer isn't computing anything, it can still be "costing" you something. Consider purchasing hardware that runs a web server for a small user base. There is a substantial investment in the hardware which requires maintenance and eventually must be replaced. However, since the user base is small, most of the time it sits idle, consuming power. A better solution may be to use the same hardware to serve multiple virtual machines (VMs). Now several small web serves can be served with the same hardware, increasing our utilization of the hardware. In scenarios like this, the lack of work being performed is the resource.

Load Somewhat the opposite of idleness, sometimes an application or service may have occasional periods of high demand. The ability of a system to service such high *loads* may be considered a resource, even if the capacity to handle them goes unused most of the time.

These are all very important engineering and business considerations when designing systems, code, and algorithms. However, we'll want to consider the complexity of algorithms in a more abstract manner.

Suppose we have to different programs (or algorithms) A and B. Both of those algorithms are correct, but A uses fewer of the above resources than B. Clearly, algorithm A is the better, more *efficient* solution. However, how can we better quantify this efficiency?

List Operations

To give a concrete example, consider a typical list ADT. The list could be implemented as an array-based list (where the class owns a static array that is resized/copied when full) or a linked list (with nodes containing elements and linking to the next node in the list). Some operations are "cheap" on one type of list while other operations may be more "expensive."

Consider the problem of inserting a new element into the list at the beginning (index 0). For a linked list this involves creating a new node and shuffling a couple of references. The number of operations in this case is not contingent on the *size* of the the list. In contrast, for an array-based list, if the list contains n elements, each element will need to be *shifted* over one position in the array in order to make room for the element to be inserted. The number of shifts is proportional to the number of elements in the array, n. Clearly for this operation, a linked list is better (more efficient).

Now consider a different operation: given an index i, retrieve the i-th element in the list. For an array-based list we have the advantage of random access to the array. When we index an element, `arr[i]`, it only takes one memory address computation to "jump" to the memory location containing the i-th element. In contrast here, a linked list would require us to start at the head, and traverse the list until we reach the i-th node. This requires i traversal operations. In the worst case, retrieving the last element, the n-th element, would require n such operations. A summary of these operations can be found

List Type	Insert at start	Index-based Retrieve
Array-based List	n	1
Linked List	2	$i \approx n$

Table 2.1: Summary of the Complexity of List Operations

in Table 2.1.

Already we have a good understanding of the relative performance of algorithms based on the type of data structure we choose. In this example we saw *constant time* algorithms and *linear* algorithms. Constant time algorithms execute a constant number of operations regardless of the size of the input (in this case, the size of the list n). Linear algorithms perform a number of operations linearly related to the size of the input.

In the following examples, we'll begin to be a little bit more formal about this type of analysis.

2.1.1 Example: Computing a Sum

The following is a toy example, but its easy to understand and straightforward. Consider the following problem: given an integer $n \geq 0$, we want to compute the arithmetic series,

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n$$

As a naive approach, consider the algorithm in Code Sample 2.1. In this algorithm, we iterate over each possible number i in the series. For each number i, we count 1 through i and add one to a result variable.

```
int result = 0;
for(int i=1; i<=n; i++) {
  for(int j=1; j<=i; j++) {
    result = result + 1;
  }
}
```

Code Sample 2.1: Summation Algorithm 1

As an improvement, consider the algorithm in Code Sample 2.2. Instead of just adding one on each iteration of the inner loop, we omit the loop entirely and simply just add the index variable i to the result.

Can we do even better? Yes. The arithmetic series actually has a closed-form solution

```
int result = 0;
for(int i=1; i<=n; i++) {
  result = result + i;
}
```

Code Sample 2.2: Summation Algorithm 2

Algorithm	Number of Additions	Input Size					
		10	100	1,000	10,000	100,000	1,000,000
1	$\approx n^2$	0.003ms	0.088ms	1.562ms	2.097ms	102.846ms	9466.489ms
2	n	0.002ms	0.003ms	0.020ms	0.213ms	0.872ms	1.120ms
3	1	0.002ms	0.001ms	0.001ms	0.001ms	0.001ms	0.000ms

Table 2.2: Empirical Performance of the Three Summation Algorithms

(usually referred to as Gauss's Formula):

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Code Sample 2.3 uses this formula to directly compute the sum without any loops.

```
int result = n * (n + 1) / 2;
```

Code Sample 2.3: Summation Algorithm 3

All three of these algorithms were run on a laptop computer for various values of n from 10 up to 1,000,000. Table 2.2 contains the resulting run times (in milliseconds) for each of these three algorithms on the various input sizes.

With small input sizes, there is almost no difference between the three algorithms. However, that would be a naive way of analyzing them. We are more interested in how each algorithm performs as the input size, n increases. In this case, as n gets larger, the differences become very stark. The first algorithm has two nested for loops. On average, the inner loop will run about $\frac{n}{2}$ times while the outer loop runs n times. Since the loops are nested, the inner loop executes about $\frac{n}{2}$ times *for each* iteration of the outer loop. Thus, the total number of iterations, and consequently the total number of additions is about

$$n \times \frac{n}{2} \approx n^2$$

The second algorithm saves the inner for loop and thus only makes n additions. The final algorithm only performs a constant number of operations.

Observe how the running time grows as the input size grows. For Algorithm 1, increasing n from 100,000 to 1,000,000 (10 times as large) results in a running time that is about 100 times as slow. This is because it is performing n^2 operations. To see this, consider the following. Let $t(n)$ be the time that Algorithm 1 takes for an input size of n. From before we know that

$$t(n) \approx n^2$$

Observe what happens when we increase the input size from n to $10n$:

$$t(10n) \approx (10n)^2 = 100n^2$$

which is 100 times as large as $t(n)$. The running time of Algorithm 1 will grow quadratically with respect to the input size n.

Similarly, Algorithm 2 grows linearly,

$$t(n) \approx n$$

Thus, a 10 fold increase in the input,

$$t(10n) \approx 10n$$

leads to a 10 fold increase in the running time. Algorithm 3's runtime does not depend on the input size, and so its runtime does not grow as the input size grows. It essentially remains flat–constant.

Of course, the numbers in Table 2.2 don't follow this trend exactly, but they are pretty close. The actual experiment involves a lot more variables than just the algorithms: the laptop may have been performing other operations, the compiler and language may have optimizations that change the algorithms, etc. Empirical results only provide general evidence as to the runtime of an algorithm. If we moved the code to a different, faster machine or used a different language, etc. we would get different numbers. However, the general trends in the rate of growth *would* hold. Those rates of growth will be what we want to analyze.

2.1.2 Example: Computing a Mode

As another example, consider the problem of computing the *mode* of a collection of numbers. The mode is the most common element in a set of data.[1]

Consider the strategy as illustrated in Code Sample 2.4. For each element in the array, we iterate through all the other elements and count how many times it appears (its *multiplicity*). If we find a number that appears more times than the candidate mode we've found so far, we update our variables and continue. As with the previous algorithm, the nested nature of our loops leads to an algorithm that performs about n^2 operations (in this case, the comparison on line 9).

[1] In general there may be more than one mode, for example in the set $\{10, 20, 10, 20, 50\}$, 10 and 20 are both modes. The problem will simply focus on finding *a* mode, not all modes.

```java
public static int mode01(int arr[]) {

  int maxCount = 0;
  int modeIndex = 0;
  for(int i=0; i<arr.length; i++) {
    int count = 0;
    int candidate = arr[i];
    for(int j=0; j<arr.length; j++) {
      if(arr[j] == candidate) {
        count++;
      }
    }
    if(count > maxCount) {
      modeIndex = i;
      maxCount = count;
    }
  }
  return arr[modeIndex];
}
```

Code Sample 2.4: Mode Finding Algorithm 1

Now consider the following variation in Code Sample 2.5. In this algorithm, the first thing we do is *sort* the array. This means that all equal elements will be contiguous. We can exploit this to do less work. Rather than going through the list a second time for each possible mode, we can count up contiguous runs of the same element. This means that we need only examine each element exactly once, giving us n comparison operations (line 8).

We can't, however, ignore the fact that to exploit the ordering, we needed to first "invest" some work upfront by sorting the array. Using a typical sorting algorithm, we would expect that it would take about $n \log{(n)}$ comparisons. Since the sorting phase and mode finding phase were separate, the total number of comparisons is about

$$n \log{(n)} + n$$

The highest order term here is the $n \log{(n)}$ term for sorting. However, this is still lower than the n^2 algorithm. In this case, the investment to sort the array pays off! To compare with our previous analysis, what happens when we increase the input size 10 fold? For simplicity, let's only consider the highest order term:

$$t(n) = n \log{(n)}$$

Then

$$t(10n) = 10n \log{(10n)} = 10n \log{(n)} + 10 \log{(10)}$$

```java
public static int mode02(int arr[]) {
  Arrays.sort(arr);
  int i=0;
  int modeIndex = 0;
  int maxCount = 0;
  while(i < arr.length-1) {
    int count=0;
    while(i < arr.length-1 && arr[i] == arr[i+1]) {
      count++;
      i++;
    }
    if(count > maxCount) {
      modeIndex = i;
      maxCount = count;
    }
    i++;
  }
  return arr[modeIndex];
}
```

Code Sample 2.5: Mode Finding Algorithm 2

The second term is a constant additive term. The increase in running time is essentially linear! We cannot discount the additive term in general, but it is so close to linear that terms like $n \log(n)$ are sometimes referred to as *quasilinear*.

Yet another solution, presented in Code Sample 2.6, utilizes a map data structure to compute the mode. A map is a data structure that allows you to store key-value pairs. In this case, we map elements in the array to a counter that represents the element's multiplicity. The algorithm works by iterating over the array and entering/updating the elements and counters.

There is some cost associated with inserting and retrieving elements from the map, but this particular implementation offers *amortized* constant running time for these operations. That is, some particular entries/retrievals may be more expensive (say linear), but when averaged over the life of the algorithm/data structure, each operation only takes a constant amount of time.

Once built, we need only go through the elements in the map (at most n) and find the one with the largest counter. This algorithm, too, offers essentially linear runtime for all inputs. Similar experimental results can be found in Table 2.3.

The difference in performance is even more dramatic than in the previous example. For an input size of 1,000,000 elements, the n^2 algorithm took nearly *8 minutes*! This is certainly unacceptable performance for most applications. If we were to extend the experiment to

```java
public static int mode03(int arr[]) {
  Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
  for(int i=0; i<arr.length; i++) {
    Integer count = counts.get(arr[i]);
    if(count == null) {
      count = 0;
    }
    count++;
    counts.put(arr[i], count);
  }
  int maxCount = 0;
  int mode = 0;
  for(Entry<Integer, Integer> e : counts.entrySet()) {
    if(e.getValue() > maxCount) {
      maxCount = e.getValue();
      mode = e.getKey();
    }
  }
  return mode;
}
```

Code Sample 2.6: Mode Finding Algorithm 3

Algorithm	Number of Additions	Input Size					
		10	100	1,000	10,000	100,000	1,000,000
1	$\approx n^2$	0.007ms	0.155ms	11.982ms	45.619ms	3565.570ms	468086.566ms
2	n	0.143ms	0.521ms	2.304ms	19.588ms	40.038ms	735.351ms
3	n	0.040ms	0.135ms	0.703ms	10.386ms	21.593ms	121.273ms

Table 2.3: Empirical Performance of the Three Mode Finding Algorithms

$n = 10{,}000{,}000$, we would expect the running time to increase to about *13 hours*! For perspective, input sizes in the millions are *small* by today's standards. Algorithms whose runtime is quadratic are *not* considered feasible for today's applications.

2.2 Pseudocode

We will want to analyze algorithms in an abstract, general way independent of any particular hardware, framework, or programming language. In order to do this, we need a way to specify algorithms that is also independent of any particular language. For that purpose, we will use *pseudocode*.

Pseudocode ("fake" code) is similar to some programming languages that you're familiar with, but does not have any particular syntax rules. Instead, it is a higher-level description of a process. You may use familiar control structures such as loops and conditionals, but you can also utilize natural language descriptions of operations.

There are no established rules for pseudocode, but in general, good pseudocode:

- Clearly labels the algorithm
- Identifies the input and output at the top of the algorithm
- Does not involve any language or framework-specific syntax–no semicolons, declaration of variables or their types, etc.
- Makes liberal use of mathematical notation and natural language for clarity

Good pseudocode abstracts the algorithm by giving enough details necessary to understand the algorithm and subsequently implement it in an actual programming language. Let's look at some examples.

```
INPUT  : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
OUTPUT : The *mean*, $\mu$ of the values in $A$
1 $sum \leftarrow 0$
2 FOREACH $a_i \in A$ DO
3     $sum \leftarrow sum + a_i$
4 END
5 $\mu \leftarrow \frac{sum}{n}$
6 output $\mu$
```

Algorithm 1: Computing the Mean

Algorithm 1 describes a way to compute the average of a collection of numbers. Observe:

- The input does not have a specific *type* (such as `int` or `double`), it uses set notation which also indicates how large the collection is.
- There is no language-specific syntax such as semicolons, variable declarations, etc.
- The loop construct doesn't specify the details of incrementing a variable, instead using a "foreach" statement with some set notation[2]
- Code blocks are not denoted by curly brackets, but are clearly delineated by using indentation and vertical lines.
- Assignment and compound assignment operators do not use the usual syntax from C-style languages, instead using a left-oriented arrow to indicate a value is assigned

[2] To review, $a_i \in A$ is a predicate meaning the element a_i is *in* the set A.

to a variable.[3]

Consider another example of computing the mode, similar to the second approach in a previous example.

```
    INPUT   : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
    OUTPUT : A *mode* of $A$
 1  Sort the elements in $A$ in non-decreasing order
 2  $multiplicity \leftarrow -\infty$
 3  FOREACH *run of contiguous equal elements a* DO
 4      $m \leftarrow$ count up the number of times $a$ appears
 5      IF $m > multiplicity$ THEN
 6          $mode \leftarrow a$
 7          $multiplicity \leftarrow m$
 8      END
 9  END
10  output $m$
```

Algorithm 2: Computing the Mode

Some more observations about Algorithm 2:

- The use of natural language to specify that the collection should be sorted and how
- The usage of $-\infty$ as a placeholder so that any other value would be greater than it
- The use of natural language to specify that an iteration takes place over contiguous elements (line 3) or that a sub-operation such as a count/summation (line 4) is performed

In contrast, bad pseudocode would be have the opposite elements. Writing a full program or code snippet in Java for example. Bad pseudocode may be unclear or it may overly simplify the process to the point that the description is trivial. For example, suppose we wanted to specify a sorting algorithm, and we did so using the pseudocode in Algorithm 3. This trivializes the process. There are many possible sorting algorithms (insertion sort, quick sort, etc.) but this algorithm doesn't specify *any* details for how to go about sorting it.

On the other hand, in Algorithm 2, we *did* essentially do this. In that case it was perfectly fine: sorting was a side operation that could be achieved by a separate algorithm. The point of the algorithm was not to specify how to sort, but instead how sorting could be used to solve another problem, finding the mode.

[3]Not all languages use the familiar single equals sign `=` for the assignment operator. The statistical programming language R uses the left-arrow operator, `<-` and Maple uses `:=` for example.

```
    INPUT  : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
    OUTPUT : $A'$, sorted in non-decreasing order
1   $A' \leftarrow$ Sort the elements in $A$ in non-decreasing order
2   output $A'$
```

Algorithm 3: Trivial Sorting (Bad Pseudocode)

Another example would be if we need to find a minimal element in a collection. Trivial pseudocode may be like that found in Algorithm 4. No details are presented on *how* to find the element. However, if finding the minimal element were an operation used in a larger algorithm (such as selection sort), then this terseness is perfectly fine. If the primary purpose of the algorithm is to find the minimal element, then details *must* be presented as in Algorithm 5.

```
    INPUT  : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
    OUTPUT : The minimal element of $A$
1   $m \leftarrow$ minimal element of $A$
2   output $m$
```

Algorithm 4: Trivially Finding the Minimal Element

```
    INPUT  : A collection of numbers, $A = \{a_1, \ldots, a_n\}$
    OUTPUT : The minimal element of $A$
1   $m \leftarrow \infty$
2   FOREACH $a_i \in A$ DO
3       IF $a_i < m$ THEN
4           $m \leftarrow a_i$
5       END
6   END
7   output $m$
```

Algorithm 5: Finding the Minimal Element

2.3 Analysis

Given two competing algorithms, we could empirically analyze them like we did in previous examples. However, it may be infeasible to implement both just to determine which is better. Moreover, by analyzing them from a more abstract, theoretical approach, we have a better more mathematically-based *proof* of the relative complexity of two algorithm.

Given an algorithm, we can analyze it by following this step-by-step process.

1. Identify the input
2. Identify the input size, n
3. Identify the *elementary operation*
4. Analyze how many times the elementary operation is executed with respect to the input size n
5. Characterize the algorithm's complexity by providing an asymptotic (Big-O, or Theta) analysis

Identifying the Input

This step is pretty straightforward. If the algorithm is described with good pseudocode, then the input will already be identified. Common types of inputs are single numbers, collections of elements (lists, arrays, sets, etc.), data structures such as graphs, matrices, etc.

However, there may be some algorithms that have *multiple* inputs: two numbers or a collection and a key, etc. In such cases, it simplifies the process if you can, without loss of generality, restrict attention to a single input value, usually the one that has the most relevance to the elementary operation you choose.

Identifying the Input Size

Once the input has been identified, we need to identify its size. We'll eventually want to characterize the algorithm as a function $f(n)$: given an input size, how many resources does it take. Thus, it is important to identify the number corresponding to the domain of this function.

This step is also pretty straightforward, but may be dependent on the type of input or even its representation. Examples:

- For collections (sets, lists, arrays), the most natural is to use the number of elements in the collection (cardinality, size, etc.). The size of individual elements is not as important as number of elements since the size of the collection is likely to grow more than individual elements do.
- An $n \times m$ matrix input could be measured by one or both nm of its dimensions.
- For graphs, you could count either the number of vertices or the number of edges

in the graph (or both!). How the graph is represented may also affect its input size (an adjacency matrix vs. an adjacency list).

- If the input is a number x, the input size is typically the number of bits required to represent x. That is,
$$n = \lceil \log_2 (x) \rceil$$
To see why, recall that if you have n bits, the maximum number you can represent is $2^n - 1$. Inverting this expression gives us $\lceil \log_2 (x) \rceil$.

Some algorithms may have multiple inputs. For example, a collection and a number (for searching) or two integers as in Euclid's algorithm. The general approach to analyzing such algorithms to simplify things by only considering *one* input. If one of the inputs is larger, such as a collection vs. a single element, the larger one is used in the analysis. Even if it is not clear which one is larger, it may be possible to assume, without loss of generality, that one is larger than the other (and if not, the inputs may be switched). The input size can then be limited to one variable to simplify the analysis.

Identifying the Elementary Operation

We also need to identify what part of the algorithm does the actual work (where the most resources will be expended). Again, we want to keep the analysis simple, so we generally only identify one *elementary operation.* There may be several reasonable candidates for the elementary operation, but in general it should be the most common or most expensive operation performed in the algorithm. For example:

- When performing numeric computations, arithmetic operations such as additions, divisions, etc.
- When sorting or searching, comparisons are the most natural elementary operations. Swaps may also be a reasonable choice depending on how you want to analyze the algorithm.
- When traversing a data structure such as a linked list, tree, or graph a node traversal (visiting or processing a node) may be considered the elementary operation.

In general, operations that are necessary to control structures (such as loops, assignment operators, etc.) are not considered good candidates for the elementary operation. An extended discussion of this can be found in Section 2.6.2.

Analysis

Once the elementary operation has been identified, the algorithm must be analyzed to count the number of times it is executed with respect to the input size. That is, we analyze the algorithm to find a function $f(n)$ where n is the input size and $f(n)$ gives the number of times the elementary operation is executed.

The analysis may involve deriving and solving a summation. For example, if the elementary operation is performed within a for loop and the loop runs a number of times that depends on the input size n.

If there are multiple loops in which the elementary operation is performed, it may be necessary to setup multiple summations. If two loops are separate and independent (one executes *after* the other), then the *sum rule* applies. The total number of operations is the sum of the operations of each loop.

If two loops are nested, then the *product rule* applies. The inner loop will execute fully *for each* iteration of the outer loop. Thus, the number of operations are multiplied with each other.

Sometimes the analysis will not be so clear cut. For example, a while loop may execute until some condition is satisfied that does not directly depend on the input size but also on the nature of the input. In such cases, we can simplify our analysis by considering the *worst-case* scenario. In the while loop, what is the *maximum* possible number of iterations for any input?

Asymptotic Characterization

As computers get faster and faster and resources become cheaper, they can process more and more information in the same amount of time. However, the characterization of an algorithm should be *invariant* with respect to the underlying hardware. If we run an algorithm on a machine that is twice as fast, that doesn't mean that the algorithm has improved. It still takes the same *number* of operations to execute. Faster hardware simply means that the time it takes to execute those operations is half as much as it was before.

To put it in another perspective, performing Euclid's algorithm to find the GCD of two integers took the same number of steps 2,300 years ago when he performed them on paper as it does today when they are executed on a digital computer. A computer is obviously faster than Euclid would have been, but both Euclid and the computer are performing the same number of steps when executing the same algorithm.

For this reason, we characterize the number of operations performed by an algorithm using *asymptotic analysis*. Improving the hardware by a factor of two only affects the "hidden constant" sitting outside of the function produced by the analysis in the previous step. We want our characterization to be invariant of those constants.

Moreover, we are really more interested in how our algorithm performs for larger and larger input sizes. To illustrate, suppose that we have two algorithms, one that performs

$$f(n) = 100n^2 + 5n$$

operations and one that performs

$$g(n) = n^3$$

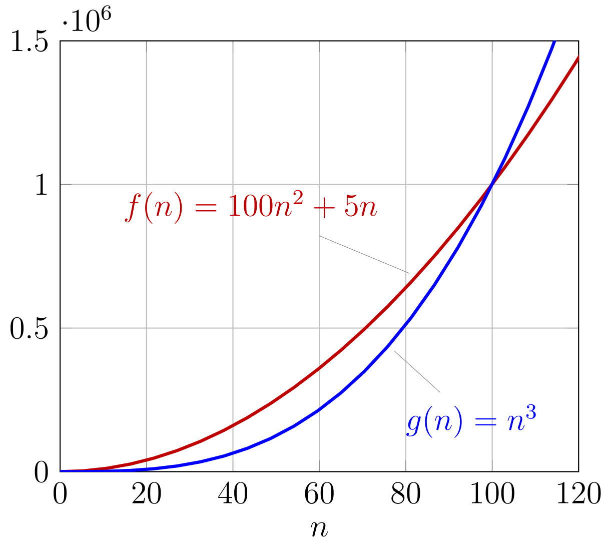


Figure 2.1: Plot of two functions.

operations. These functions are graphed in Figure 2.1. For inputs of size less than 100, the first algorithm performs *worse* than the second (the graph is higher indicating "more" resources). However, for inputs of size greater than 100, the first algorithm is better. For small inputs, the second algorithm may be better, but small inputs are not the norm for any "real" problems.[4] In any case, on modern computers, we would expect small inputs to execute fast anyway as they did in our empirical experiments in Section 2.1.1 and 2.1.2. There was essentially no discernible difference in the three algorithms for sufficiently small inputs.

We can rigorously quantify this by providing an asymptotic characterization of these functions. An asymptotic characterization essentially characterizes the *rate of growth* of a function or the *relative* rate of growth of functions. In this case, n^3 grows much faster than $100n^2 + 5n$ as n grows (tends toward infinity). We formally define these concepts in the next section.

2.4 Asymptotics

2.4.1 Big-O Analysis

We want to capture the notion that one function grows faster than (or at least as fast as) another. Categorizing functions according to their growth rate has been done for a long

[4]There are problems where we can apply a "hybrid" approach: we can check for the input size and choose one algorithm for small inputs and another for larger inputs. This is typically done in hybrid sorting algorithms such as when merge sort is performed for "large" inputs but switches over to insertion sort for smaller arrays.

time in mathematics using *big-O notation*.[5]

Definition 1. Let f and g be two functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in O(g(n))$$

read as "f is big-O of g," if there exist constants $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

First, let's make some observations about this definition.

- The "O" originally stood for "order of", Donald Knuth referred to it as the capital greek letter omicron, but since it is indistinguishable from the Latin letter "O" it makes little difference.
- Some definitions are more general about the nature of the functions f, g. However, since we're looking at these functions as characterizing the resources that an algorithm takes to execute, we've restricted the domain and codomain of the functions. The domain is restricted to non-negative integers since there is little sense in negative or factional input sizes. The codomain is restricted to nonnegative reals as it doesn't make sense that an algorithm would potentially consume a negative amount of resources.
- We've used the set notation $f(n) \in O(g(n))$ because, strictly speaking, $O(g(n))$ is a *class* of functions: the set of all functions that are asymptotically bounded by $g(n)$. Thus the set notation is the most appropriate. However, you will find many sources and papers using notation similar to

  $$f(n) = O(g(n))$$

  This is a slight abuse of notation, but common nonetheless.

The intuition behind the definition of big-O is that f is *asymptotically* less than or equal to g. That is, the rate of growth of g is *at least as fast* as the growth rate of f. Big-O provides a means to express that one function is an *asymptotic upper bound* to another function.

The definition essentially states that $f(n) \in O(g(n))$ if, after some point (for all $n \geq n_0$), the value of the function $g(n)$ will always be larger than $f(n)$. The constant c possibly serves to "stretch" or "compress" the function, but has no effect on the growth rate of the function.

[5]The original notation and definition are attributed to Paul Bachmann in 1894 [4]. Definitions and notation have been refined and introduced/reintroduced over the years. Their use in algorithm analysis was first suggested by Donald Knuth in 1976 [10].

Example

Let's revisit the example from before where $f(n) = 100n^2 + 5n$ and $g(n) = n^3$. We want to show that $f(n) \in O(g(n))$. By the definition, we need to show that there exists a c and n_0 such that

$$f(n) \leq cg(n)$$

As we observed in the graph in Figure 2.1, the functions "crossed over" somewhere around $n = 100$. Let's be more precise about that. The two functions cross over when they are equal, so we setup an equality,

$$100n^2 + 5n = n^3$$

Collecting terms and factoring out an n (that is, the functions have one crossover point at $n = 0$), we have

$$n^2 - 100n - 5 = 0$$

The values of n satisfying this inequality can be found by applying the quadratic formula, and so

$$n = \frac{100 \pm \sqrt{10000 + 20}}{2}$$

Which is $-0.049975\ldots$ and $100.0499\ldots$. The first root is negative and so irrelevant. The second is our cross over point. The next largest integer is 101. Thus, for $c = 1$ and $n_0 = 101$, the inequality is satisfied.

In this example, it was easy to find the intersection because we could employ the quadratic equation to find roots. This is much more difficult with higher degree polynomials. Throw in some logarithmic functions, exponential functions, etc. and this approach can be difficult.

Revisit the definition of big-O: the inequality doesn't have to be tight or precise. In the previous example we essentially fixed c and tried to find n_0 such that the inequality held. Alternatively, we could fix n_0 to be small and then find the c (essentially compressing the function) such that the inequality holds. Observe:

$$\begin{aligned}
100n^2 + 5n &\leq 100n^2 + 5n^2 && \text{since } n \leq n^2 \text{ for all } n \geq 0 \\
&= 105n^2 \\
&\leq 105n^3 && \text{since } n^2 \leq n^3 \text{ for all } n \geq 0 \\
&= 105g(n)
\end{aligned}$$

By adding positive values, we make the equation larger until it looks like what we want, in this case $g(n) = n^3$. By the end we've got our constants: for $c = 105$ and $n_0 = 0$, the inequality holds. There is nothing special about this c, $c = 1000000$ would work too. The point is we need only find at least one c, n_0 pair that the inequality holds (there are an infinite number of possibilities).

2.4.2 Other Notations

Big-O provides an asymptotic upper bound characterization of two functions. There are several other notations that provide similar characterizations.

Big-Omega

Definition 2. Let f and g be two functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \Omega(g(n))$$

read as "f is big-Omega of g," if there exist constants $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$f(n) \geq cg(n)$$

Big-Omega provides an asymptotic lower bound on a function. The only difference is the inequality has been reversed. Intuitively f has a growth rate that is bounded below by g.

Big-Theta

Yet another characterization can be used to show that two functions have the *same* order of growth.

Definition 3. Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that

$$f(n) \in \Theta(g(n))$$

read as "f is Big-Theta of g," if there exist constants $c_1, c_2 \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that for every integer $n \geq n_0$,

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Big-Θ essentially provides an asymptotic equivalence between two functions. The function f is bounded above *and* below by g. As such, both functions have the same rate of growth.

Soft-O Notation

Logarithmic factors contribute very little to a function's rate of growth especially compared to larger order terms. For example, we called $n \log(n)$ *quasi*linear since it was nearly linear. Soft-O notation allows us to simplify terms by removing logarithmic factors.

Definition 4. Let f, g be functions such that $f(n) \in O(g(n) \cdot \log^k(n))$. Then we say that $f(n)$ is *soft-O* of $g(n)$ and write

$$f(n) \in \tilde{O}(g(n))$$

For example,

$$n \log(n) \in \tilde{O}(n)$$

Little Asymptotics

Related to big-O and big-Ω are their corresponding "little" asymptotic notations, little-o and little-ω.

Definition 5. Let f and g be two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. We say that

$$f(n) \in o(g(n))$$

read as "f is little-o of g," if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

The little-o is sometimes defined as for every $\epsilon > 0$ there exists a constant N such that

$$|f(n)| \leq \epsilon |g(n)| \quad \forall n \geq N$$

but given the restriction that $g(n)$ is positive, the two definitions are essentially equivalent.

Little-o is a much stronger characterization of the relation of two functions. If $f(n) \in o(g(n))$ then not only is g an asymptotic upper bound on f, but they are *not* asymptotically equivalent. Intuitively, this is similar to the difference between saying that $a \leq b$ and $a < b$. The second is a stronger statement as it implies the first, but the first does not imply the second. Analogous to this example, little-o provides a "strict" asymptotic upper bound. The growth rate of g is *strictly* greater than the growth rate of f.

Similarly, a little-ω notation can be used to provide a *strict* lower bound characterization.

Definition 6. Let f and g be two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. We say that

$$f(n) \in \omega(g(n))$$

read as "f is little-omega of g," if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

2.4.3 Observations

As you might have surmised, big-O and big-Ω are duals of each other, thus we have the following.

Lemma 1. Let f, g be functions. Then

$$f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$$

Because big-Θ provides an asymptotic equivalence, both functions are big-O *and* big-Θ of each other.

Lemma 2. Let f, g be functions. Then

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

Equivalently,

$$f(n) \in \Theta(g(n)) \iff g(n) \in O(f(n)) \text{ and } g(n) \in \Omega(f(n))$$

With respect to the relationship between little-o and little-ω to big-O and big-Ω, as previously mentioned, little asymptotics provide a stronger characterization of the growth rate of functions. We have the following as a consequence.

Lemma 3. Let f, g be functions. Then

$$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$$

and

$$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$$

Of course, the converses of these statements do not hold.

Common Identities

As a direct consequence of the definition, constant coefficients in a function can be ignored.

Lemma 4. For any constant c,

$$c \cdot f(n) \in O(f(n))$$

In particular, for $c = 1$, we have that

$$f(n) \in O(f(n))$$

and so any function is an upper bound on itself.

In addition, when considering the sum of two functions, $f_1(n), f_2(n)$, it suffices to consider the one with a larger rate of growth.

Lemma 5. Let $f_1(n), f_2(n)$ be functions such that $f_1(n) \in O(f_2(n))$. Then

$$f_1(n) + f_2(n) \in O(f_2(n))$$

In particular, when analyzing algorithms with independent operations (say, loops), we only need to consider the operation with a higher complexity. For example, when we presorted an array to compute the mode, the presort phase was $O(n \log(n))$ and the mode finding phase was $O(n)$. Thus the total complexity was

$$n \log(n) + n \in O(n \log(n))$$

When dealing with a polynomial of degree k,

$$c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \cdots + c_1 n + c_0$$

The previous results can be combined to conclude the following lemma.

Lemma 6. Let $p(n)$ be a polynomial of degree k,

$$p(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \cdots + c_1 n + c_0$$

then

$$p(n) \in \Theta(n^k)$$

Logarithms

When working with logarithmic functions, it suffices to consider a single base. As Computer Scientists, we always work in base-2 (binary). Thus when we write $\log(n)$, we implicitly mean $\log_2(n)$ (base-2). It doesn't really matter though because all logarithms are the same to within a constant as a consequence of the change of base formula:

$$\log_b(n) = \frac{\log_a(n)}{\log_a(b)}$$

That means that for any valid bases a, b,

$$\log_b(n) \in \Theta(\log_a(n))$$

Another way of looking at it is that an algorithm's complexity is the same regardless of whether or not it is performed by hand in base-10 numbers or on a computer in binary.

Other logarithmic identities that you may find useful remembering include the following:

$$\log(n^k) = k \log(n)$$

$$\log(n_1 n_2) = \log(n_1) + \log(n_2)$$

Classes of Functions

Table 2.4 summarizes some of the complexity functions that are common when doing algorithm analysis. Note that these classes of functions form a hierarchy. For example, linear and quasilinear functions are also $O(n^k)$ and so are polynomial.

Class Name	Asymptotic Characterization	Algorithm Examples
Constant	$O(1)$	Evaluating a formula
Logarithmic	$O(\log(n))$	Binary Search
Polylogarithmic	$O(\log^k(n))$	
Linear	$O(n)$	Linear Search
Quasilinear	$O(n \log(n))$	Mergesort
Quadratic	$O(n^2)$	Insertion Sort
Cubic	$O(n^3)$	
Polynomial	$O(n^k)$ for any $k > 0$	
Exponential	$O(2^n)$	Computing a powerset
Super-Exponential	$O(2^{f(n)})$ for $f(n) \in \Omega(n)$ For example, $n!$	Computing permutations

Table 2.4: Common Algorithmic Efficiency Classes

2.4.4 Limit Method

The method used in previous examples directly used the definition to find constants c, n_0 that satisfied an inequality to show that one function was big-O of another. This can get quite tedious when there are many terms involved. A much more elegant proof technique borrows concepts from calculus.

Let $f(n), g(n)$ be functions. Suppose we examine the limit, as $n \to \infty$ of the ratio of these two functions.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

One of three things could happen with this limit.

The limit could converge to 0. If this happens, then by Definition 5 we have that $f(n) \in o(g(n))$ and so by Lemma 3 we know that $f(n) \in O(g(n))$. This makes sense: if the limit converges to zero that means that $g(n)$ is growing much faster than $f(n)$ and so f is big-O of g.

The limit could diverge to infinity. If this happens, then by Definition 6 we have that $f(n) \in \omega(g(n))$ and so again by Lemma 3 we have $f(n) \in \Omega(g(n))$. This also makes sense: if the limit diverges, $f(n)$ is growing much faster than $g(n)$ and so $f(n)$ is big-Ω of g.

Finally, the limit could converge to some positive constant (recall that both functions are restricted to positive codomains). This means that both functions have essentially the same order of growth. That is, $f(n) \in \Theta(g(n)$. As a consequence, we have the following Theorem.

Theorem 1 (Limit Method)**.** Let $f(n)$ and $g(n)$ be functions. Then if

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0 & \text{then } f(n) \in O(g(n)) \\ c > 0 & \text{then } f(n) \in \Theta(g(n)) \\ \infty & \text{then } f(n) \in \Omega(g(n)) \end{cases}$$

Examples

Let's reuse the example from before where $f(n) = 100n^2 + 5n$ and $g(n) = n^3$. Setting up our limit,

$$\begin{aligned} \lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\frac{100n^2+5n}{n^3} \\ &= \lim_{n\to\infty}\frac{100n+5}{n^2} \\ &= \lim_{n\to\infty}\frac{100n}{n^2} + \lim_{n\to\infty}\frac{5}{n^2} \\ &= \lim_{n\to\infty}\frac{100}{n} + 0 \\ &= 0 \end{aligned}$$

And so by Theorem 1, we conclude that

$$f(n) \in O(g(n))$$

Consider the following example: let $f(n) = \log_2 n$ and $g(n) = \log_3(n^2)$. Setting up our limit we have

$$\begin{aligned} \lim_{n\to\infty}\frac{f(n)}{g(n)} &= \frac{\log_2 n}{\log_3 n^2} \\ &= \frac{\log_2 n}{\frac{2\log_2 n}{\log_2 3}} \\ &= \frac{\log_2 3}{2} \\ &= .7924\ldots > 0 \end{aligned}$$

And so we conclude that $\log_2(n) \in \Theta(\log_3(n^2))$.

As another example, let $f(n) = \log(n)$ and $g(n) = n$. Setting up the limit gives us

$$\lim_{n\to\infty}\frac{\log(n)}{n}$$

The rate of growth might seem obvious here, but we still need to be mathematically rigorous. Both the denominator and numerator are monotone increasing functions. To solve this problem, we can apply l'Hôpital's Rule:

Theorem 2 (l'Hôpital's Rule)**.** Let f and g be functions. If the limit of the quotient $\frac{f(n)}{g(n)}$ exists, it is equal to the limit of the derivative of the denominator and the numerator. That is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Applying this to our limit, the denominator drops out, but what about the numerator? Recall that $\log(n)$ is the logarithm base-2. The derivative of the natural logarithm is well known, $\ln'(n) = \frac{1}{n}$. We can use the change of base formula to transform $\log(n) = \frac{\ln(n)}{\ln(2)}$ and then take the derivative. That is,

$$\log'(n) = \frac{1}{\ln(2)n}$$

Thus,

$$\begin{aligned}\lim_{n\to\infty} \frac{\log(n)}{n} &= \lim_{n\to\infty} \frac{\log'(n)}{n'} \\ &= \lim_{n\to\infty} \frac{1}{\ln(2)n} \\ &= 0\end{aligned}$$

Concluding that $\log(n) \in O(n)$.

Pitfalls

l'Hôpital's Rule is not always the most appropriate tool to use. Consider the following example: let $f(n) = 2^n$ and $g(n) = 3^n$. Setting up our limit and applying l'Hôpital's Rule we have

$$\begin{aligned}\lim_{n\to\infty} \frac{2^n}{3^n} &= \lim_{n\to\infty} \frac{(2^n)'}{(3^n)'} \\ &= \lim_{n\to\infty} \frac{(\ln 2)2^n}{(\ln 3)3^n}\end{aligned}$$

which doesn't get us anywhere. In general, we should look for algebraic simplifications *first.* Doing so we would have realized that

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n$$

Since $\frac{2}{3} < 1$, the limit of its exponent converges to zero and we have that $2^n \in O(3^n)$.

2.5 Examples

2.5.1 Linear Search

As a simple example, consider the problem of searching a collection for a particular element. The straightforward solution is known as Linear Search and is featured as Algorithm 6

```
INPUT  : A collection $A = \{a_1, \ldots, a_n\}$, a key $k$
OUTPUT : The first $i$ such that $a_i = k$, $\phi$ otherwise
FOR $i = 1, \ldots, n$ DO
    IF $a_i = k$ THEN
        output $i$
    END
END
output $\phi$
```

Algorithm 6: Linear Search

Let's follow the prescribed outline above to analyze Linear Search.

1. Input: this is clearly indicated in the pseudocode of the algorithm. The input is the collection A.
2. Input Size: the most natural measure of the size of a collection is its cardinality; in this case, n
3. Elementary Operation: the most common operation is the comparison in line 2 (assignments and iterations necessary for the control flow of the algorithm are not good candidates).
4. How many times is the elementary operation executed with respect to the input size, n? The situation here actually depends not only on n but also the contents of the array.
   - Suppose we get lucky and find k in the first element, a_1: we've only made one comparison.
   - Suppose we are unlucky and find it as the last element (or don't find it at all). In this case we've made n comparisons
   - We could also look at the *average* number of comparisons (see Section 2.6.3)

   In general, algorithm analysis considers at the worst case scenario unless otherwise stated. In this case, there are $C(n) = n$ comparisons.

5. This is clearly linear, which is why the algorithm is called Linear Search,

$$\Theta(n)$$

2.5.2 Set Operation: Symmetric Difference

Recall that the *symmetric difference* of two sets, $A \oplus B$ consists of all elements in A *or* B but not both. Algorithm 7 computes the symmetric difference.

```
INPUT  : Two sets, $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_m\}$
OUTPUT : The symmetric difference, $A \oplus B$
1  $C \leftarrow \emptyset$
2  FOREACH $a_i \in A$ DO
3      IF $a_i \notin B$ THEN
4          $C \leftarrow C \cup \{a_i\}$
5      END
6  END
7  FOREACH $b_j \in B$ DO
8      IF $b_j \notin A$ THEN
9          $C \leftarrow C \cup \{b_j\}$
10     END
11 END
12 output $C$
```

Algorithm 7: Symmetric Difference of Two Sets

Again, following the step-by-step process for analyzing this algorithm,

1. Input: In this case, there are two sets as part of the input, A, B.
2. Input Size: As specified, each set has cardinality n, m respectively. We could analyze the algorithm with respect to both input sizes, namely the input size could be $n + m$. For simplicity, to work with a single variable, we could also define $N = n + m$.

   Alternatively, we could make the following observation: without loss of generality, we can assume that $n \geq m$ (if not, switch the sets). If one input parameter is bounded by the other, then

   $$n + m \leq 2n \in O(n)$$

   That is, we could simplify the analysis by only considering n as the input size. There will be no difference in the final asymptotic characterization as the constants will be ignored.

3. Elementary Operation: In this algorithm, the most common operation is the set membership query ($\notin$). Strictly speaking, this operation may not be trivial depending on the type of data structure used to represent the set (it may entail a series of $O(n)$ comparisons for example). However, as our pseudocode is concerned, it is sufficient to consider it as our elementary operation.
4. How many times is the elementary operation executed with respect to the input size, n? In the first for-loop (lines 2–6) the membership query is performed n times. In the second loop (lines 7–11), it is again performed m times. Since each of these loops is independent of each other, we would *add* these operations together to get

   $$n + m$$

   total membership query operations.
5. Whether or not we consider $N = n + m$ or n to be our input size, the algorithm is clearly linear with respect to the input size. Thus it is a $\Theta(n)$-time algorithm.

2.5.3 Euclid's GCD Algorithm

The greatest common divisor (or GCD) of two integers a, b is the largest positive integer that divides both a and b. Finding a GCD has many useful applications and the problem has one of the oldest known algorithmic solutions: Euclid's Algorithm (due to the Greek mathematician Euclid c. 300 BCE).

```
INPUT  : Two integers, $a, b$
OUTPUT : The greatest common divisor, gcd($a, b$)
while $b \neq 0$ do
    $t \leftarrow b$
    $b \leftarrow a \bmod b$
    $a \leftarrow t$
end
Output $a$
```

Algorithm 8: Euclid's GCD Algorithm

The algorithm relies on the following observation: any number that divides a, b must also divide the remainder of a since we can write $b = a \cdot k + r$ where r is the remainder. This suggests the following strategy: iteratively divide a by b and retain the remainder r, then consider the GCD of b and r. Progress is made by observing that b and r are necessarily smaller than a, b. Repeating this process until we have a remainder of zero gives us the GCD because once we have that one evenly divides the other, the larger must be the GCD. Pseudocode for Euclid's Algorithm is provided in Algorithm 8.

The analysis of Euclid's algorithm is seemingly straightforward. It is easy to identify the division in line 3 as the elementary operation. But how many times is it executed with respect to the input size? What is the input size?

When considering algorithms that primarily execute numerical operations the input is usually a number (or in this case a pair of numbers). How big is the input of a number? The input size of 12,142 is not 12,142. The number 12,142 has a compact representation when we write it: it requires 5 digits to express it (in base 10). That is, the input size of a number is the number of symbols required to represent its magnitude. Considering a number's input size to be equal to the number would be like considering its representation in unary where a single symbol is used and repeated for as many times as is equal to the number (like a prisoner marking off the days of his sentence).

Computers don't "speak" in base-10, they speak in binary, base-2. Therefore, the input size of a numerical input is the number of bits required to represent the number. This is easily expressed using the base-2 logarithm function:

$$\lceil \log(n) \rceil$$

But in the end it doesn't really matter if we think of computers as speaking in base-10, base-2, or any other integer base greater than or equal to 2 because as we've observed that all logarithms are equivalent to within a constant factor using the change of base formula. In fact this again demonstrates again that algorithms are an abstraction independent of any particular platform: that the same algorithm will have the same (asymptotic) performance whether it is performed on paper in base-10 or in a computer using binary!

Back to the analysis of Euclid's Algorithm: how many times does the while loop get executed? We can first observe that each iteration reduces the value of b, but by how much? The exact number depends on the input: some inputs would only require a single division, other inputs reduce b by a different amount on each iteration. The important thing to to realize is that we want a general characterization of this algorithm: it suffices to consider the worst case. That is, at maximum, how many iterations are performed? The number of iterations is maximized when the reduction in the value of b is minimized at each iteration. We further observe that b is reduced by at least half at each iteration. Thus, the number of iterations is maximized if we reduce b by at most half on each iteration. So how many iterations i are required to reduce n down to 1 (ignoring the last iteration when it is reduced to zero for the moment)? This can be expressed by the equation:

$$n\left(\frac{1}{2}\right)^i = 1$$

Solving for i (taking the log on either side), we get that

$$i = \log(n)$$

Recall that the size of the input is $\log(n)$. Thus, Euclid's Algorithm is *linear* with respect to the input size.

2.5.4 Selection Sort

Recall that Selection Sort is a sorting algorithm that sorts a collection of elements by first finding the smallest element and placing it at the beginning of the collection. It continues by finding the smallest among the remaining $n-1$ and placing it second in the collection. It repeats until the "first" $n-1$ elements are sorted, which by definition means that the last element is where it needs to be.

The Pseudocode is presented as Algorithm 9.

```
   INPUT  : A collection $A = \{a_1, \ldots, a_n\}$
   OUTPUT : $A$ sorted in non-decreasing order
 1 FOR $i = 1, \ldots, n-1$ DO
 2     $min \leftarrow a_i$
 3     FOR $j = (i+1), \ldots, n$ DO
 4         IF $min < a_j$ THEN
 5             $min \leftarrow a_j$
 6         END
 7         swap $min, a_i$
 8     END
 9 END
10 output $A$
```

Algorithm 9: Selection Sort

Let's follow the prescribed outline above to analyze Selection Sort.

1. Input: this is clearly indicated in the pseudocode of the algorithm. The input is the collection A.
2. Input Size: the most natural measure of the size of a collection is its cardinality; in this case, n
3. Elementary Operation: the most common operation is the comparison in line 4 (assignments and iterations necessary for the control flow of the algorithm are not good candidates). Alternatively, we could have considered swaps on line 7 which would lead to a different characterization of the algorithm.
4. How many times is the elementary operation executed with respect to the input size, n?
   - Line 4 does one comparison each time it is executed
   - Line 4 itself is executed multiple times for each iteration of the for loop in line 3 (for j running from $i+1$ up to n inclusive.

- line 3 (and subsequent blocks of code) are executed multiple times for each iteration of the for loop in line 1 (for i running from 1 up to $n-1$

This gives us the following summation:

$$\underbrace{\sum_{i=1}^{n-1} \underbrace{\sum_{j=i+1}^{n} \underbrace{1}_{\text{line 4}}}_{\text{line 3}}}_{\text{line 1}}$$

Solving this summation gives us:

$$\begin{aligned}
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1} n - i \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
&= n(n-1) - \frac{n(n-1)}{2} \\
&= \frac{n(n-1)}{2}
\end{aligned}$$

Thus for a collection of size n, Selection Sort makes

$$\frac{n(n-1)}{2}$$

comparisons.

5. Provide an asymptotic characterization: the function determined is clearly $\Theta(n^2)$

2.6 Other Considerations

2.6.1 Importance of Input Size

The second step in our algorithm analysis outline is to identify the input size. Usually this is pretty straightforward. If the input is an array or collection of elements, a reasonable input size is the cardinality (the number of elements in the collection). This is usually the case when one is analyzing a sorting algorithm operating on a list of elements.

Though seemingly simple, sometimes identifying the appropriate input size depends on the nature of the input. For example, if the input is a data structure such as a graph, the input size could be either the number of vertices *or* number of edges. The most appropriate measure then depends on the algorithm and the details of its analysis. It may even depend on how the input is represented. Some graph algorithms have different efficiency measures if they are represented as adjacency lists or adjacency matrices.

Yet another subtle difficulty is when the input is a single numerical value, n. In such instances, the input size is *not* also n, but instead the number of symbols that would be needed to represent n. That is, the number of digits in n or the number of bits required to represent n. We'll illustrate this case with a few examples.

Sieve of Eratosthenes

A common beginner mistake is made when analyzing the Sieve of Eratosthenes (named for Eratosthenes of Cyrene, 276 BCE – 195 BCE). The Sieve is an ancient method for prime number factorization. A brute-force algorithm, it simply tries every integer up to a point to see if it is a factor of a given number.

```
   INPUT   : An integer $n$
   OUTPUT  : Whether $n$ is *prime* or *composite*
1  FOR $i = 2, \ldots, n$ DO
2      IF $i$ *divides* $n$ THEN
3          Output *composite*
4      END
5  END
6  Output *prime*
```

Algorithm 10: Sieve of Eratosthenes

The for-loop only needs to check integers up to $\sqrt{n}$ because any factor greater than $\sqrt{n}$ would necessarily have a corresponding factor $\sqrt{n}$. A naive approach would observe that the for-loop gets executed $\sqrt{n} - 1 \in O(\sqrt{n})$ times which would lead to the (incorrect) impression that the Sieve is a polynomial-time (in fact sub-linear!) running time algorithm. Amazing that we had a primality testing algorithm over 2,000 years ago! In fact, primality testing was a problem that was not known to have a deterministic polynomial time running algorithm until 2001 (the AKS Algorithm [3]).

The careful observer would realize that though n is the input, the actual input size is again $\log(n)$, the number of bits required to represent n. Let $N = \log(n)$ be a placeholder for our actual input size (and so $n = 2^N$). Then the running time of the Sieve is actually

$$O(\sqrt{n}) = O(\sqrt{2^N})$$

which is exponential with respect to the input size N.

This distinction is subtle but crucial: the difference between a polynomial-time algorithm and an exponential algorithm is huge even for modestly sized inputs. What may take a few milliseconds using a polynomial time algorithm may take billions and billions of years with an exponential time algorithm as we'll see with our next example.

Computing an Exponent

As a final example, consider the problem of computing a modular exponent. That is, given integers a, n, and m, we want to compute

$$a^n \bmod m$$

A naive (but common!) solution might be similar to the Java code snippet in Code Sample 2.7.

Whether one chooses to treat multiplication or integer division as the elementary operation, the for-loop executes exactly n times. For "small" values of n this may not present a problem. However, for even moderately large values of n, say $n \approx 2^{256}$, the performance of this code will be terrible.

```java
int a = 45, m = 67;
int result = 1;
for(int i=1; i<=n; i++) {
  result = (result * a % m);
}
```

Code Sample 2.7: Naive Exponentiation

To illustrate, suppose that we run this code on a 14.561 petaFLOP (14 quadrillion floating point operations per second) super computer cluster (this throughput was achieved by the Folding@Home distributed computing project in 2013). Even with this power, to make 2^{256} floating point operations would take

$$\frac{2^{256}}{14.561 \times 10^{15} \cdot 60 \cdot 60 \cdot 24 \cdot 365.25} \approx 2.5199 \times 10^{53}$$

or 252 sexdecilliion *years* to compute!

For context, this sort of operation is performed by millions of computers around the world every second of the day. A 256-bit number is not really all that "large". The problem again lies in the failure to recognize the difference between an input, n, and the input's *size*. We will explore the better solution to this problem when we examine Repeated Squaring in Section 5.3.

2.6.2 Control Structures are Not Elementary Operations

When considering which operation to select as the elementary operation, we usually do *not* count operations that are necessary to the control structure of the algorithm. For example, assignment operations, or operations that are necessary to execute a loop (incrementing an index variable, a comparison to check the termination condition).

To see why, consider method in Code Sample 2.8. This method computes a simple average of an array of `double` variables. Consider some of the minute operations that are performed in this method:

- An assignment of a value to a variable (lines 2, 3, and 4)
- An increment of the index variable `i` (line 3)
- A comparison (line 3) to determine if the loop should terminate or continue
- The addition (line 4) and division (line 6)

```
1 public static double average(double arr[]) {
2   double sum = 0.0;
3   for(int i=0; i<arr.length; i++) {
4     sum = sum + arr[i];
5   }
6   return sum / arr.length;
7 }
```

Code Sample 2.8: Computing an Average

A proper analysis would use the addition in line 4 as the elementary operation, leading to a $\Theta(n)$ algorithm. However, for the sake of argument, let's perform a detailed analysis with respect to each of these operations. Assume that there are n values in the array, thus the for loop executes n times. This gives us

- $n + 2$ total assignment operations
- n increment operations
- n comparisons
- n additions and
- 1 division

Now, suppose that each of these operations take time t_1, t_2, t_3, t_4, and t_5 milliseconds each (or whatever time scale you like). In total, we have a running time of

$$t_1(n + 2) + t_2 n + t_3 n + t_4 n + t_5 = (t_1 + t_2 + t_3 + t_4)n + (2t_1 + t_5) = cn + d$$

Which doesn't change the asymptotic complexity of the algorithm: considering the additional operations necessary for the control structures only changed the constant sitting out front (as well as some additive terms).

The amount of resources (in this case time) that are expended for the assignment, increment and comparison for the loop control structure are proportional to the true elementary operation (addition). Thus, it is sufficient to simply consider the most common or most expensive operation in an algorithm. The extra resources for the control structures end up only contributing constants which are ultimately ignored when an asymptotic analysis is performed.

2.6.3 Average Case Analysis

The behavior of some algorithms may depend on the nature of the input rather than simply the input size. From this perspective we can analyze an algorithm with respect to its best-, average-, and worst-case running time.

In general, we prefer to consider the worst-case running time when comparing algorithms.

Example: searching an array

Consider the problem of searching an array for a particular element (the array is unsorted, contains n elements). You could get lucky (best-case) and find it immediately in the first index, requiring only a single comparison. On the other hand, you could be unlucky and find the element in the last position or not at all. In either case n comparisons would be required (worst-case).

What about the average case? A naive approach would be to average the worst and best case to get an average of $\frac{n+1}{2}$ comparisons. A more rigorous approach would be to define a probability of a successful search p and the probability of an unsuccessful search, $1 - p$.

For a successful search, we further define a uniform probability distribution on finding the element in each of the n indices. That is, we will find the element at index i with probability $\frac{p}{n}$. Finding the element at index i requires i comparisons. Thus the total number of expected comparisons in a successful search is

$$\sum_{i=1}^{n} i \cdot \frac{p}{n} = \frac{p(n+1)}{2}$$

For an unsuccessful search, we would require n comparisons, thus the number of expected comparisons would be

$$n(1-p)$$

Since these are mutually exclusive events, we sum the probabilities:

$$\frac{p(n+1)}{2} + n(1-p) = \frac{p - pn + 2n}{2}$$

p	C
0	n
$\frac{1}{4}$	$\frac{7}{8}n + \frac{1}{8}$
$\frac{1}{2}$	$\frac{3}{4}n + \frac{1}{4}$
$\frac{3}{4}$	$\frac{5}{8}n + \frac{3}{8}$
1	$\frac{n+1}{2}$

Table 2.5: Expected number of comparisons C for various values of the probability of a successful search p.

We cannot remove the p terms, but we can make some observations for various values (see Table 2.6.3). When $p = 1$ for example, we have the same conclusion as the naive approach. As p decreases, the expected number of comparisons grows to n.

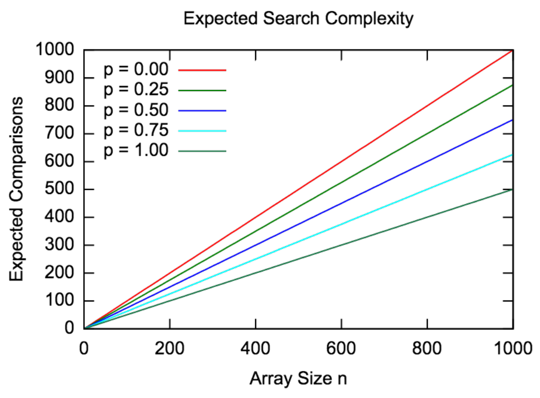


Figure 2.2: Expected number of comparisons for various success probabilities p.

2.6.4 Amortized Analysis

Sometimes it is useful to analyze an algorithm not based on its worst-case running time, but on its *expected* running time. That is, on average, how many resources does an algorithm perform? This is a more practical approach to analysis. Worst-case analysis assumes that all inputs will be difficult, but in practice, difficult inputs may be rare. The average input may require fewer resources.

Amortized algorithm analysis is similar to the average-case analysis we performed before, but focuses on how the *cost* of operations may change over the course of an algorithm. This is similar to a loan from a bank. Suppose the loan is taken out for \$1,000 at a 5% interest rate. You don't actually end up paying \$50 the first year. You pay pay slightly less than that since you are (presumably) making monthly payments, reducing the balance and thus reducing the amount of interest accrued each month.

We will not go into great detail here, but as an example, consider Heap Sort. This algorithm uses a *Heap* data structure. It essentially works by inserting elements into the heap and then removing them one by one. Due to the nature of a heap data structure, the elements will come out in the desired order.

An amortized analysis of Heap Sort may work as follows. First, a heap requires about $\log(n)$ comparisons to insert an element (as well as remove an element, though we'll focus on insertion), where n is the size of the heap (the number of elements currently in the heap. As the heap grows (and eventually shrinks), the cost of inserts will change. With an initially empty heap, there will only be 0 comparisons. When the heap has 1 element, there will be about $\log(1)$ comparisons, then $\log(2)$ comparisons and so on until we insert the last element. This gives us

$$C(n) = \sum_{i=1}^{n-1} \log(i)$$

total comparisons.

2.7 Analysis of Recursive Algorithms

Recursive algorithms can be analyzed with the same basic 5 step process. However, because they are recursive, the analysis (step 4) may involve setting up a recurrence relation in order to characterize how many times the elementary operation is executed.

Though cliche, as a simple example consider a naive recursive algorithm to compute the n-th Fibonacci number. Recall that the Fibonacci sequence is defined as

$$F_n = F_{n-1} + F_{n-2}$$

with initial conditions $F_0 = F_1 = 1$. That is, the n-th Fibonacci number is defined as the sum of the two previous numbers in the sequence. This defines the sequence

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

The typical recursive solution is presented as Algorithm 11

The elementary operation is clearly the addition in line 4. However, how do we analyze the number of additions performed by a call to Fibonacci(n)? To do this, we setup

```
  INPUT   : An integer $n$
  OUTPUT : The $n$-th Fibonacci Number, $F_n$
1 IF $n = 0$ or $n = 1$ THEN
2     output 1
3 ELSE
4     output Fibonacci($n-1$) + Fibonacci($n-2$)
5 END
```

Algorithm 11: Fibonacci(n)

a recurrence relation. Let $A(n)$ be a function that counts the number of additions performed by Fibonacci(n). If n is zero or one, the number of additions is zero (the base case of our recursion performs no additions). That is, $A(0) = A(1) = 0$. But what about $n > 1$?

When $n > 1$, line 4 executes. Line 4 contains one addition. However, it also contains two recursive calls. How many additions are performed by a call to Fibonacci($n-1$) and Fibonacci($n-2$)? We defined $A(n)$ to be the number of additions on a call to Fibonacci(n), so we can reuse this function: the number of additions is $A(n-1)$ and $A(n-2)$ respectively. Thus, the total number of additions is

$$A(n) = A(n-1) + A(n-2) + 1$$

This is a recurrence relation,[6] in particular, it is a second-order linear non-homogeneous recurrence relation. This particular relation can be solved. That is, $A(n)$ can be expressed as a non-recursive closed-form solution.

The techniques required for solving these type of recurrence relations are beyond the scope of this text. However, for many common recursive algorithms, we can use a simple tool to characterize their running time, the "Master Theorem."

2.7.1 The Master Theorem

Suppose that we have a recursive algorithm that takes an input of size n. The recursion may work by dividing the problem into a subproblems each of size $\frac{n}{b}$ (where a, b are constants). The algorithm may also perform some amount of work before or after the recursion. Suppose that we can characterize this amount of work by a polynomial function, $f(n) \in \Theta(n^d)$.

This kind of recursive algorithm is common among "divide-and-conquer" style algorithms that *divide* a problem into subproblems and *conquers* each subproblem. Depending on

[6]Also called a *difference equation*, which are sort of discrete analogs of differential equations.

the values of a, b, d we can categorize the runtime of this algorithm using the Master Theorem.

Theorem 3 (Master Theorem)**.** Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{array}{rcl} T(n) & = & aT(\frac{n}{b}) + f(n) \\ T(1) & = & c \end{array}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

"Master" is a bit of a misnomer. The Master Theorem can only be applied to recurrence relations of the particular form described. It can't, for example, be applied to the previous recurrence relation that we derived for the Fibonacci algorithm. However, it can be used for several common recursive algorithms.

Example: Binary Search

As an example, consider the Binary Search algorithm, presented as Algorithm 12. Recall that binary search takes a *sorted* array (random access is required) and searches for an element by checking the middle element m. If the element being searched for is larger than m, a recursive search on the upper half of the list is performed, otherwise a recursive search on the lower half is performed.

Let's analyze this algorithm with respect to the number of comparisons it performs. To simplify, though we technically make two comparisons in the pseudocode (lines 5 and 7), let's count it as a single comparison. In practice a comparator pattern would be used and logic would branch based on the result. However, there would still only be one invocation of the comparator function/object. Let $C(n)$ be a function that equals the number of comparisons made by Algorithm 12 while searching an array of size n *in the worst case* (that is, we never find the element or it ends up being the last element we check).

As mentioned, we make one comparison (line 5, 7) and then make one recursive call. The recursion roughly cuts the size of the array in half, resulting in an array of size $\frac{n}{2}$. Thus,

$$C(n) = C\left(\frac{n}{2}\right) + 1$$

Applying the master theorem, we find that $a = 1$, $b = 2$. In this case, $f(n) = 1$ which *is* bounded by a polynomial: a polynomial of degree $d = 0$. Since

$$1 = a = b^d = 2^0$$

by case 2 of the Master Theorem applies and we have

$$C(n) \in \Theta(\log(n))$$

```
INPUT   : A *sorted* collection of elements $A = \{a_1, \ldots, a_n\}$, bounds $1 \leq l, h \leq n$,
          and a key $e_k$
OUTPUT  : An element $a \in A$ such that $a = e_k$ according to some criteria; $\phi$ if no
          such element exists
1  IF $l > h$ THEN
2      output $\phi$
3  END
4  $m \leftarrow \lfloor \frac{h+l}{2} \rfloor$
5  IF $a_m = e_k$ THEN
6      output $a_m$
7  ELSE IF $a_m < e_k$ THEN
8      BinarySearch$(A, m+1, h, e)$
9  ELSE
10     BinarySearch$(A, l, m-1, e)$
11 END
```

Algorithm 12: Binary Search – Recursive

Example: Merge Sort

Recall that Merge Sort is an algorithm that works by recursively splitting an array of size n into two equal parts of size roughly $\frac{n}{2}$. The recursion continues until the array is trivially sorted (size 0 or 1). Following the recursion back up, each subarray half is sorted and they need to be *merged.* This basic divide and conquer approach is presented in Algorithm 13. We omit the details of the merge operation on line 4, but observe that it can be achieved with roughly $n-1$ comparisons in the worst case.

```
INPUT   : An array, sub-indices $1 \leq l, r \leq n$
OUTPUT  : An array $A'$ such that $A[l, \ldots, r]$ is sorted
1  IF $l < r$ THEN
2      MergeSort$(A, l, \lfloor \frac{r+l}{2} \rfloor)$
3      MergeSort$(A, \lceil \frac{r+l}{2} \rceil, r)$
4      Merge sorted lists $A[l, \ldots, \lfloor \frac{r+l}{2} \rfloor]$ and$A[\lceil \frac{r+l}{2} \rceil, \ldots, r]$
5  END
```

Algorithm 13: Merge Sort

Let $C(n)$ be the number of comparisons made by Merge Sort. The main algorithm makes *two* recursive calls on subarrays of size $\frac{n}{2}$. There are also $n+1$ comparisons made after the recursion. Thus we have

$$C(n) = 2C\left(\frac{n}{2}\right) + (n-1)$$

Again, applying the Master Theorem we have that $a = 2, b = 2$ and that $f(n) = (n+1) \in \Theta(n)$ and so $d = 1$. Thus,

$$2 = a = b^d = 2^1$$

and by case 2,

$$C(n) \in \Theta(n \log(n))$$

We will apply the Master Theorem to several other recursive algorithms later on.

3 Storing Things

3.1 Lists

3.2 Sets

3.3 Hash-Tables

Motivation:

- Linked lists provide $O(n), O(1)$ access/insert efficiency
- Array-based lists provide $O(1), O(n)$ access/insert efficiency
- Balanced binary search trees provide $O(\log n)$ operations
- Hash tables provide *amortized* $O(1)$ operations

Basics:

- Elements are stored in a fixed-size array
- The location of an element (its index) is computed by computing a *hash* value of the element
- We can restrict attention to integer keys (elements/objects can be mapped to an integer key based on their state)
- Computing the hash value of a key provides quick, random access to the associated bucket containing the element (value)
- Insertion and retrieval are achieved by computing the hash value and accessing the array

3.3.1 Hash Functions

- Simply put, a hash function is a function that maps a large domain to a small co-domain

- For our purposes:
$$h : \mathbb{Z} \to \mathbb{Z}_m$$
where $\mathbb{Z}_m = \{0, 1, 2, \ldots, m-1\}$
- Hash functions should be
    1. Easy to compute
    2. Distribute values as uniformly as possible among $\mathbb{Z}_m$

Example. Consider the hash function:

$$h(k) = 7k + 13 \bmod 8$$

Insertion of the following keys into a hash table with the above has function results in the following configuration.

$$10, 16, 4, 13, 86$$

Index	0	1	2	3	4	5	6	7
Value	13	4		10		16		86

Table 3.1: Resulting Hash Table

But, now suppose we attempt to insert the key 100: $h(100) = 1$. The array cell is already occupied by 4. This is known as a *collision*

3.3.2 Collisions

- By definition, hash functions map large sets to small sets; they cannot be one-to-one
- Some elements k_1, k_2 will necessarily map to the same hash value $h(k_1) = h(k_2)$
- When two elements map to the same hash value, it is called a *collision*
- Collisions can be *resolved* using various strategies; we examine two:
    1. Open addressing
    2. Chaining

Open Addressing

Open addressing involves *probing* the hash table to find an open cell in the table. Several strategies are available.

- Linear probing involves probing the table using a linear function:

$$h(k, i) = h(k) + c_1 \cdot i \bmod m$$

for $i = 1, \ldots, m-1$ and $c_1 \in \mathbb{Z}_m$

- Quadratic probing involves probing the table using a quadratic function:

$$h(k,i) = h(k) + c_1 \cdot i + c_2 \cdot i^2 \bmod m$$

for $i = 0, 1, \ldots, m$

- In general, $c_1, c_2 \in \mathbb{Z}_m$
- Probing wraps around the table (modulo m)
- Double Hashing involves using a second hash function $s(k)$ to probe:

$$h(k,i) = h(k) + i \cdot s(k) \bmod m$$

for $i = 0, 1, \ldots, m$

Retrieval and Deletion

- Retrieval now involves hashing *and* resolving collisions until either the element is found or a free cell is found (item is not in the table)
- If deletions are allowed, additional bookkeeping is needed to indicate if a cell has *ever* been occupied; sometimes referred to as a "dirty bit"

Chaining

- No probing, instead each bucket in the hash table represents a linked list
- Alternatively: another efficient collection (BST, etc.)
- Each element is added to the list associated with the hash value
- Retrieval now may involve traversing another data structure
- Retrieval may now also involve another mechanism to identify each object in the collection
- No more issues with deletion
- Rehashing may still be necessary to prevent each list/collection from getting too large

3.3.3 Efficiency Rehashing

Efficiency of a hash table depends on how full it is. Let n be the number of elements in the table, then the *load factor* of the table is

$$\alpha = \frac{n}{m}$$

For successful searches, the expected number of key comparisons is

$$S = 1 + \frac{\alpha}{2}$$

The fuller a table gets, the more its performance degrades to linear search:

- Smaller table → more collisions → slower access/insert (but less wasted space)
- Larger table → fewer collisions → faster access/insert (but more wasted space)

This represents a fundamental time/space trade-off.

- As a hash table fills up, more collisions slow down access of elements
- If deletion is allowed, more cells become dirty and slow down access of elements
- To accommodate more elements and to ensure that insert/retrieval remains fast a *rehashing* of the tables
    1. A new, larger table is created with a new hash function using a larger m
    2. Each element is reinserted into the new hash table, an $O(n)$ operation.
- Rehashing is done infrequently (when the load factor is exceeded): when averaged over the lifecycle of the data structure, it leads to *amortized* constant time.

3.3.4 Other Applications

- Pseudorandom number generators
- Data Integrity (check sums)
- Cryptography: message authentication codes, data integrity, password storage (ex: MD4, 5, SHA-1, 2, 3)
- Derandomization
- Maps, Associative Arrays, Sets, Caches

3.3.5 Java Implementations

The Java Collections library provides several data structures that are backed by hash tables that utilize an object's `hashCode()` method and rely on several assumptions. For this reason, it is best practice to always override the `hashCode()` and `equals()` methods for every user-defined object so that they depend on an object's entire state.

- `java.util.HashSet<E>`
    - Implements the `Set` interface

  - Permits a single `null` element
- `java.util.Hashtable<K,V>`
  - A chained hash table implementation
  - Documentation labels this implementation as "open" (actually refers to "open hashing")
  - Allows you to specify an initial capacity and load factor with defaults (.75)
  - Automatically grows/rehashes (`rehash()`)
  - Implements the `Map` interface
  - Synchronized for multithreaded applications
  - Does not allow `null` keys nor values
- `java.util.HashMap<K,V>`
  - Implements the `Map` interface
  - Allows at most one `null` key, any number of `null` values
  - Subclass: `LinkedHashMap` (guarantees iteration order, can be used as an LRU cache)

ol

3.4 Bloom Filters

3.5 Disjoint Sets

3.6 Exercises

Exercise 3.1. Consider the following simple hash function.

$$h(k) = (5k + 3) \bmod 11$$

and consider the following elements:

$$4, 42, 26, 5, 8, 18, 14$$

(a) Insert the above elements into a hash table, using linear probing to resolve collisions. Show what the hash table would look like after inserting the elements in the order above.

(b) Insert the above elements into a hash table using chaining to resolve collisions. Show what the hash table would look like after inserting the elements in the order above.

4 Brute Force Style Algorithms

4.1 Introduction

- Brute Force style algorithms are simple a "just do it" approach
- Usually an "obvious", straightforward solution
- Not ideal and may be completely infeasible for even small inputs
- May be necessary (ex: sequential search, max/min finding)

4.1.1 Examples

Shuffling Cards

- There are $52! \approx 8.065817 \times 10^{67}$ possible shuffles
- If 5 billion people shuffled once per second for the last 1,000 years, only:

$$1.95 \times 10^{-48} = 0.\underbrace{00\ldots00}_{\text{45 zeros}}195\%$$

  have been examined so far!
- Even at 5 billion shuffles per second, it would take

$$5.1118 \times 10^{50}$$

  years to enumerate all possibilities

Greatest Common Divisor

- Given two integers, $a, b \in \mathbb{Z}^+$, the *greatest common divisor*:

$$\gcd(a, b)$$

  is the largest integer that divides both a and b
- Naive solution: factor a, b:

$$\begin{array}{rcl} a & = & p_1^{k_1} p_2^{k_1} \cdots p_n^{k_n} \\ b & = & p_1^{\ell_1} p_2^{\ell_1} \cdots p_n^{\ell_n} \end{array}$$

- Then
$$\gcd(a,b) = p_1^{\min\{k_1,\ell_1\}} p_2^{\min\{k_2,\ell_2\}} \cdots p_n^{\min\{k_n,\ell_n\}}$$
- Factoring is not known (or believed) to be in P (deterministic polynomial time)
- Better solution: Euclid's GCD algorithm

4.1.2 Backtracking

- We iteratively (or recursively) build *partial* solutions such that solutions can be "rolled back" to a prior state
- Once a full solution is generated, we process/test it then roll it back to a (previous partial) solution
- Advantage: depending on the nature/structure of the problem, entire branches of the computation tree may be "pruned" out
- Avoiding infeasible solutions can greatly speed up computation in practice
- Heuristic speed up only: in the worst case and in general, the computation may still be exponential

4.2 Generating Combinatorial Objects

4.2.1 Generating Combinations (Subsets)

Recall that combinations are simply all possible subsets of size k. For our purposes, we will consider generating subsets of

$$\{1, 2, 3, \ldots, n\}$$

The algorithm works as follows.

- Start with $\{1, \ldots, k\}$
- Assume that we have $a_1 a_2 \cdots a_k$, we want the next combination
- Locate the last element a_i such that $a_i \neq n - k + i$
- Replace a_i with $a_i + 1$
- Replace a_j with $a_i + j - i$ for $j = i+1, i+2, \ldots, k$

```
INPUT  : A set of $n$ elements and an $k$-combination, $a_1 \cdots a_k$.
OUTPUT : The next $k$-combination.
1 $i = k$
2 WHILE $a_i = n - k + i$ DO
3     $i = i - 1$
4 END
5 $a_i = a_i + 1$
6 FOR $j = (i + 1) \ldots r$ DO
7     $a_j = a_i + j - i$
8 END
```

Algorithm 14: Next k-Combination

Example: Find the next 3-combination of the set $\{1, 2, 3, 4, 5\}$ after $\{1, 4, 5\}$

Here, $n = 5, k = 3$, $a_1 = 1, a_2 = 4, a_3 = 5$.

The last i such that $a_i \neq 5 - 3 + i$ is 1.

Thus, we set

$$\begin{aligned} a_1 &= a_1 + 1 = 2 \\ a_2 &= a_1 + 2 - 1 = 3 \\ a_3 &= a_1 + 3 - 1 = 4 \end{aligned}$$

So the next k-combination is $\{2, 3, 4\}$.

Two full examples are shown in Figure 4.1.

4.2.2 Generating Permutations

- A *permutation* of n elements, $a_1, \ldots, a_n$ is an ordered arrangement of these elements
- There are $n!$
- The Johnson-Trotter algorithm generates all permutations with the *least change* property (at most two elements are swapped in each permutation).

Lexicographically Ordered Permutations:

- Start with permutation $1234 \cdots n$
- Given a current permutation, $a_1, a_2, \ldots, a_n$
- Find the right-most pair a_i, a_{i+1} such that $a_i < a_{i+1}$
- Find the smallest element (to the right, "in the tail") larger than a_i, label it a'
- Swap a' and a_i

Iteration	Combination
1	123
2	124
3	125
4	134
5	135
6	145
7	234
8	235
9	245
10	345

(a) Sequence for $\binom{5}{3}$

Iteration	Combination
1	1234
2	1235
3	1236
4	1245
5	1246
6	1256
7	1345
8	1346
9	1356
10	1456
11	2345
12	2346
13	2356
14	2456
15	3456

(b) Sequence for $\binom{6}{4}$

Figure 4.1: Two combination sequence examples

- Order (sort) elements to the right of a'
- Example: $163542 \rightarrow 164235$
- 35 is the last pair; 4 is the minimal element (among 5, 4, 2) greater than $a_i = 3$; swap $3, 4$ giving 164532; sort 532 to 235 giving the result
- Example: $12345 \rightarrow 12354 \rightarrow 12435 \rightarrow \cdots$

Generalization:

- In general, permutations can be formed from subsets of a set
- The number ordered permutations of k elements from a set of n distinct elements is:
$$P(n, k) = \frac{n!}{(n-k)!}$$
- We can adapt the two previous algorithms to generate these permutations: use the combinations algorithm to generate all distinct subsets of size k; then run the permutation algorithm on each subset to generate ordered permutations.

4.2.3 Permutations with Repetition

- Let $A = \{a_1, \ldots, a_n\}$ be a set with n objects

```
INPUT  : A permutation $a_1a_2 \ldots a_n$ of elements $1, 2, \ldots, n$
OUTPUT : The lexicographically next permutation
$i \leftarrow n - 1$
WHILE $a_i > a_{i+1}$ DO
    $i \leftarrow (i - 1)$
END
$a' \leftarrow a_{i+1}$
FOR $j = (i + 2) \ldots n$ DO
    IF $a_j > a_i \wedge a_j < a'$ THEN
        $a' \leftarrow a_j$
    END
END
swap $a', a_i$
sort elements $a_{i+1} \cdots a_n$
```

Algorithm 15: Next Lexicographic Permutation

- We wish to form an ordered permutation of length k of elements from A but we are also allowed to take as many "copies" of each element as we wish
- Number of such permutations: n^k
- Use case: generating DNA sequences from amino acid bases, $\{A, G, C, T\}$
- Straightforward idea: count in base $b = n$ and associate each number in base b with an element in A
- Count from 0 to $n^k - 1$ and generate elements
- A direct counting algorithm is presented as Algorithm 16
- A general purpose algorithm for base-conversion is presented as Algorithm 17

```
INPUT  : A base set $A = \{a_1, \ldots, a_n\}$ and a current permutation (base-$n$
         number) $x_1 \cdots x_k$
OUTPUT : The next permutation
1  $i \leftarrow k$
2  $carry \leftarrow true$
3  WHILE $carry \wedge i \geq 1$ DO
4      $x_i \leftarrow x_i + 1$
5      IF $x_i \geq n$ THEN
6          $x_i \leftarrow 0$
7          $i \leftarrow i + 1$
8      ELSE
9          $carry \leftarrow false$
10     END
11 END
12 output $x_1 \cdots x_k$
```

Algorithm 16: Next Repeated Permutation Generator

```
INPUT  : $n \in \mathbb{N}$, an ordered symbol set, $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{b-1}\}$, a base $b$
OUTPUT : The base-$b$ representation of $n$, $x_k \ldots x_1 x_0$ using the symbol set $\Sigma$
1  $i \leftarrow 0$
2  WHILE $n > 0$ DO
3      $j \leftarrow n \mod b$
4      $n \leftarrow \lfloor \frac{n}{b} \rfloor$
5      $x_i \leftarrow \sigma_j$
6      $i \leftarrow (i + 1)$
7  END
8  optionally: pad out $x$ with leading $\sigma_0$ symbols (leading zeros) until it has the
   desired length
9  output $x_k \cdots x_1 x_0$
```

Algorithm 17: Base Conversion Algorithm

4.2.4 Set Partitions

Let $A = \{a_1, \ldots a_n\}$ be a set. A *partition* of A is a collection of disjoint subsets of A whose union is equal to A. That is, a partition splits up the set A into a collection of subsets. The number of ways to partition a set corresponds to the *Bell numbers*:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

with $B_0 = 1$.

```
INPUT  : A set $A = \{a_1, \ldots, a_n\}$
OUTPUT : A set of all set partitions of $A$
1  $partitions \leftarrow \{\{a_1\}\}$
2  FOR $i = 2, \ldots, |A|$ DO
3      $newPartitions \leftarrow \emptyset$
4      FOREACH partition $part \in partitions$ DO
5          FOREACH set $s \in part$ DO
6              $newPart \leftarrow copyofpart$
7              add $a_i$ to $s$ in $newPart$
8              add $newPart$ to $newPartitions$
9          END
10         $newPart \leftarrow copyofpart$
11         add $\{a_i\}$ to $newPart$
12     END
13     $partitions \leftarrow newPartitions$
14 END
15 output $partitions$
```

Algorithm 18: Set Partition Generator

Demonstration: Starting with $\{\{a,b\},\{\{a\},\{b\}\}\}$ (which partitions the set $A = \{a,b\}$ into two possible partitions), we can produce all partitions of the set $A' = \{a,b,c\}$ by adding c to each of the possible subsets of subsets (or in a set by itself):

$$\begin{gathered}\{\{\{a\},\{b\},\{c\}\},\\ \{\{a\},\{b,c\}\},\\ \{\{b\},\{a,c\}\},\\ \{\{c\},\{a,b\}\},\\ \{\{a,b,c\}\}\}\end{gathered}$$

4.3 Problems & Algorithms

4.3.1 Satisfiability

- Recall that a *predicate* is a boolean function on n variables:
$$P(\vec{x}) = P(x_1, \ldots, x_n)$$
- A predicate is *satisfiable* if there exists an assignment of variables that makes P evaluate to true:
$$\exists x_1, \ldots, x_n \, [P(x_1, \ldots, x_n)]$$
- Example:
$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$$
is satisfiable (there are two assignments for which the exclusive-or evaluates to true)
- Example:
$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$
is *not* satisfiable: any setting of the three variables evaluates to false
- Deciding if a given predicate is satisfiable or not is a fundamental NP-complete problem

Problem 1 (Satisfiability)**.**
Given: A boolean predicate P on n variables
Output: TRUE if P is satisfiable, false otherwise

```
INPUT  : A predicate $P(x_1, \ldots, x_n)$
OUTPUT : true if $P$ is satisfiable, false otherwise
foreach truth assignment $\vec{t}$ do
    if $P(\vec{t}) = 1$ then
        return True
    end
    return False
end
```

Algorithm 19: Brute Force Iterative Algorithm for Satisfiability

```
INPUT  : A predicate $P(x_1, \ldots, x_n)$, a partial truth assignment $\vec{t} = t_1, \ldots, t_k$
OUTPUT : true if $P$ is satisfiable, false otherwise
if $k = n$ then
    return $P(t_1, \ldots, t_n)$
else
    $t_{k+1} \leftarrow 0$
    if Sat$(P, \vec{t})$ then
        return True;
    else
        $t_{k+1} \leftarrow 1$
        return Sat$(P, \vec{t})$
    end
end
```

Algorithm 20: Brute Force Recursive Algorithm for Satisfiability

4.3.2 Hamiltonian Path/Cycle

Definition 7. Let $G = (V, E)$ be an undirected graph and let $v \in V$ be a vertex. The *neighborhood* of v, $N(v)$ is the set of vertices connected to v.

$$N(v) = \{x | (u, v) \in E\}$$

Problem 2 (Hamiltonian Path)**.**
Given: An undirected graph $G = (V, E)$
Output: True if G contains a Hamiltonian Path, false otherwise

Variations:

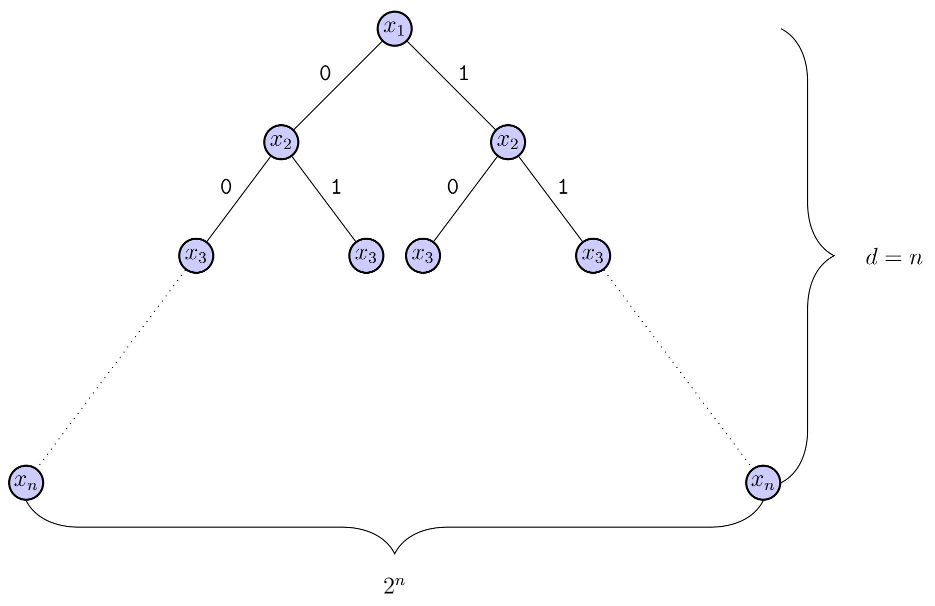


Figure 4.2: Computation Tree for Satisfiability Backtracking Algorithm

- Functional version: output a Hamiltonian path if one exists
- Hamiltonian Cycle (Circuit) problem
- Weighted Hamiltonian Path/Cycle
- Traveling Salesperson Problem

Problem 3 (Traveling Salesperson (TSP))**.**
Given: A collection of n cities, $C = \{c_1, \ldots, c_n\}$
Output: A permutation of cities $\pi(C) = (c'_1, c'_2, \ldots, c'_n)$ such that the total distance,

$$\left(\sum_{i=1}^{n-1} \delta(c'_i, c'_{i+1}) \right) + \delta(c'_n, c'_1)$$

is minimized.

Discussion: is one variation "harder" than the other?

- Contrast two brute force strategies
- Backtracking enables *pruning* of recurrence tree
- potential for huge heuristic speed up

```
INPUT  : A graph, $G = (V, E)$, a path $p = v_1, \ldots, v_k$
OUTPUT : true if $G$ contains a Hamiltonian cycle, false otherwise
foreach permutation of vertices $pi = v_1, \ldots, v_n$ do
    $isHam \leftarrow true$
    for $i = 1, \ldots, n-1$ do
        if $(v_i, v_{i+1}) \notin E$ then
            $isHam \leftarrow false$
        end
    end
    if $(v_n, v_1) \notin E$ then
        $isHam \leftarrow false$
    end
    if $isHam$ then
        return TRUE
    end
end
return FALSE
```

Algorithm 21: Brute Force Iterative Algorithm for Hamiltonian Cycle

```
INPUT  : A graph, $G = (V, E)$, a path $p = v_1, \ldots, v_k$
OUTPUT : true if $G$ contains a Hamiltonian cycle, false otherwise
if $|p| = n$ then
    return TRUE
end
foreach $x \in N(v_k)$ do
    if $x \notin p$ then
        if WALK$(G, p + x)$ then
            return TRUE
        end
    end
end
return TRUE
```

Algorithm 22: Hamiltonian DFS Cycle Walk

```
Input   : A graph, $G = (V, E)$
Output  : True if $G$ contains a Hamiltonian path, False otherwise
1 foreach $v \in V$ do
2     $path \leftarrow v$
3     if Walk$(G, p)$ then
4         output True
5     end
6     output False
7 end
8 return True
```

Algorithm 23: Hamiltonian DFS Path Walk–Main Algorithm

```
Input   : A graph, $G = (V, E)$, a path $p = v_1 \cdots v_k$
Output  : True if $G$ contains a Hamiltonian path, False otherwise
1 if $k = n$ then
2     return True
3 end
4 foreach $v \in N(v_k)$ do
5     if $v \notin p$ then
6         Walk$(G, p + v)$
7     end
8 end
9 return False
```

Algorithm 24: Walk(G, p) – Hamiltonian DFS Path Walk

- To see how the walk approach can provide a substantial speed up observe the graph if Figure 4.3
- Any permutation of vertices that involves edges that are not present need not be processed
- Backtracking computation is illustrated in Figure 4.4

4.3.3 0-1 Knapsack

Problem 4 (0-1 Knapsack).
Given: A collection of n items, $A = \{a_1, \ldots, a_n\}$, a weight function $wt : A \to \mathbb{R}+$, a value function $val : A \to \mathbb{R}^+$ and a knapsack capacity W

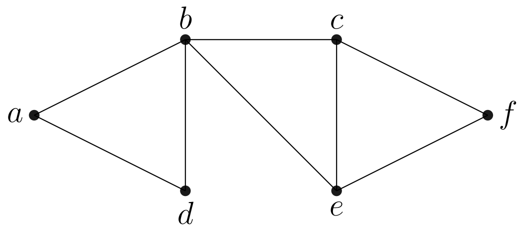


Figure 4.3: A small Hamiltonian Graph

Output: A subset $S \subseteq A$ such that

$$wt(S) = \sum_{s \in S} wt(s) \leq W$$

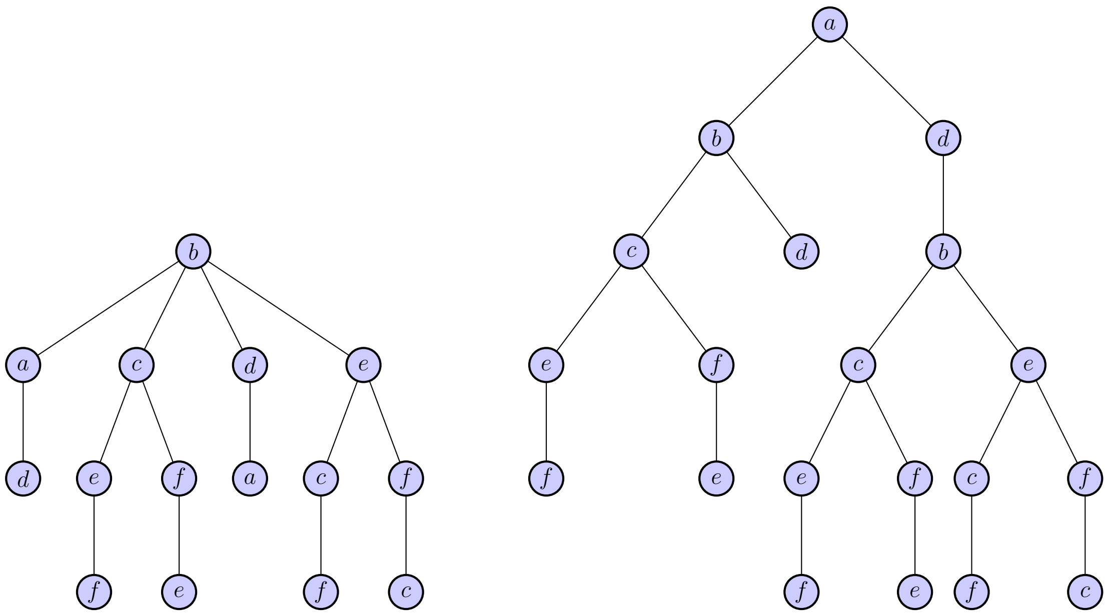
and

$$val(S) = \sum_{s \in S} val(s)$$

is maximized.

An example input can be found in Figure 4.5. The optimal solution is to take items a_1, a_3, a_4 for a value of 29. Another example can be found in Table 9.7 in a later section where we look at a dynamic programming solution for the same problem.

- We wish to maximize the value of items (in general, an "objective function") taken subject to some constraint
- 0-1 refers to the fact that we can an item or leave it
- Variation: the *dual* of the problem would be to minimize some *loss* or *cost*
- Variation: fractional knapsack
- A greedy strategy fails in general; a small example: three items with weights $1, 2, 3$ and values $6, 10, 12$ respectively; and a total weight capacity of 4. The ratios would be $6, 5, 4$ respectively meaning that a greedy strategy would have us select the first two items for a total weight of 3 and value of 16. However, the optimal solution would be to select the first and third item for a total weight of 4 and value of 18.
- A brute-force backtracking algorithm is presented as Algorithm 14; note that you would invoke this algorithm with $j = 0$ and $S = \emptyset$ initially.



(a) Hamiltonian path computation tree starting from b.

(b) Hamiltonian path computation tree starting from a.

Figure 4.4: Brute Force Backtracing Hamiltonian Path Traversal. The first traversal starts from b and finds no Hamiltonian Path. The second traversal finds several and would terminate prior to exploring the entire computation tree. The entire tree is presented for completeness.

Item	Value	Weight
a_1	15	1
a_2	10	5
a_3	9	3
a_4	5	4

Figure 4.5: Example Knapsack Input with $W = 8$

```
INPUT  : An instance of the knapsack problem $K = (A, wt, val, W)$, an index $j$,
         and a partial solution $S \subseteq A$ consisting of elements not indexed more
         than $j$
OUTPUT : A solution $S'$ that is at least as good as $S$
IF $j = n$ THEN
    return $S$
END
$S_{best} \leftarrow S$
FOR $k = j + 1, \ldots, n$ DO
    $S' \leftarrow S \cup \{a_k\}$
    IF $wt(S') \leq W$ THEN
        $T \leftarrow$ KNAPSACK$(K, k, S')$
        IF $val(T) > val(S_{best})$ THEN
            $S_{best} \leftarrow T$
        END
    END
END
return $S_{best}$
```

Algorithm 25: KNAPSACK(K, S) – Backtracking Brute Force 0-1 Knapsack

4.3.4 Closest Pair

Problem 5 (Closest Pair).
Given: A collection of n points, $P = \{p_1 = (x_1, y_2), \ldots, p_n = (x_n, y_n)\}$
Output: A pair of points (p_a, p_b) that are the closest according to Euclidean distance.

4.3.5 Convex Hull

- A set of point P in the plane is *convex* if for any pair of points $p, q \in P$ every point in the line segment $\overline{pq}$ is contained in P

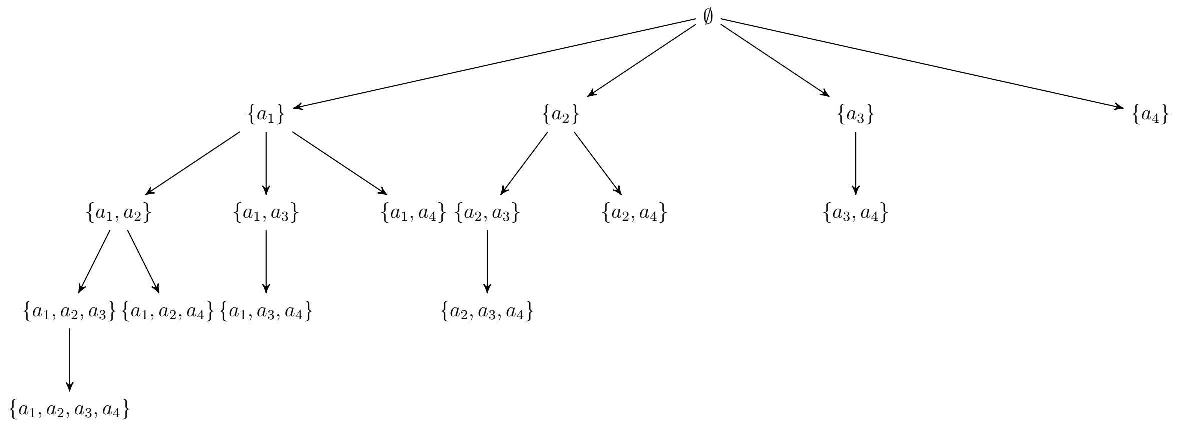


Figure 4.6: Knapsack Computation Tree for $n = 4$

- A *convex hull* of a set of points P is the smallest convex set containing P

Problem 6 (Convex Hull)**.**
Given: A set P of n points
Output: A convex hull H of P

- The idea is to choose *extreme points*: points that when connected form the border of the convex hull
- A pair of points $p_1 = (x_1, y_1), p_2 = (x_2, y_2)$ are *extreme points* if all other $p \in P \setminus \{p_1, p_2\}$ lie on the same side of the line defined by p_1, p_2

```
INPUT   : A set of points $P \subseteq \mathbb{R}^2$
OUTPUT : A convex hull $H$ of $P$
$H \leftarrow \emptyset$
FOREACH pair of points $p_i, p_j$ DO
    $m \leftarrow \frac{y_j - y_i}{x_j - x_i}$
    $b \leftarrow y_i - m \cdot x_i$ //$m, b$ define the line $\overline{p_i p_j}$
    $p_k \leftarrow$ an arbitrary point in $P \setminus \{p_i, p_j\}$
    $s \leftarrow \text{sgn}(y_k - m \cdot x_k - b)$ //$s$ indicates which side of the line $p_k$
      lies on: positive for above, negative for below
    $isExtreme \leftarrow true$
    FOREACH $p = (x, y) \in P \setminus \{p_i, p_j\}$ DO
        IF $s \neq \text{sgn}(y - m \cdot x - b)$ THEN
            $isExtreme \leftarrow false$
        END
    END
    IF $isExtreme$ THEN
        $H \leftarrow H \cup \{p_1, p_2\}$
    END
END
output $H$
```

Algorithm 26: Brute Force Convex Hull

4.3.6 Assignment Problem

Problem 7 (Assignment).
Given: A collection of n tasks $T = \{t_1, \ldots, t_n\}$ and n persons $P = \{p_1, \ldots, p_n\}$ and an $n \times n$ matrix C with entries $c_{i,j}$ representing the *cost* of assigning person p_i to task t_j.
Output: A bijection $f : P \to T$ such that

$$\sum_{p_i \in P} C[i, f(p_i)]$$

is minimized.

A brute force approach: try all bijections $f : T \to P$, equivalent to all permutations of one of these sets.

This is actually efficiently solvable via the Hungarian Method or by a Linear Program formulation.

4.3.7 Subset Sum

Problem 8 (Subset Sum)**.**
Given: A set of n integers, $A = \{a_1, \ldots, a_n\}$ and an integer k
Output: true if there exists a subset $S \subseteq A$ such that

$$\sum_{s \in S} s = k$$

- Solution: Generate all possible subsets of A and check their sum
- Backtracking solution: use the same algorithm as Knapsack; backtrack if sum exceeds k
- Variation: does there exist a partitioning into two sets whose sums are equal?

```
INPUT  : A set of n integers, $A = \{a_1, \ldots, a_n\}$, an integer $k$, a partial solution
         $S = \{a_\ell, \ldots, a_j\} \subseteq A$
OUTPUT : A subset $S' \subseteq A$ whose elements sum to $k$ if one exists; nil otherwise
1  IF $sum(S) = k$ THEN
2      return $S$
3  END
4  IF $sum(S) > k$ THEN
5      return nil
6  END
7  FOR $i = j + 1, \ldots n$ DO
8      $T \leftarrow$ SubsetSum($S \cup \{a_i\}, k$)
9      IF $T \neq nil$ THEN
10         return $T$
11     END
12 END
```

Algorithm 27: Brute Force Subset Sum

4.4 Exercises

Exercise 4.1. Implement the brute-force solution for the Hamiltonian Path problem in the high-level programming language of your choice.

Exercise 4.2. Implement the brute-force solution for the 0-1 Knapsack problem in the high-level programming language of your choice.

Exercise 4.3. Consider the following *frequent itemset problem*: we are given a set of *items* $A = \{a_1, a_2, \ldots, a_n\}$ and a set of *baskets* $B_1, B_2, \ldots, B_m$ such that each basket is a subset containing a certain number of items, $B_i \subseteq A$. Given a *support* s, *frequent itemsets* are subsets $S \subset A$ such that S is a subset of t baskets (that is, the items in S *all* appear in at least s baskets).

For example, suppose that we have the following baskets:

$$\begin{aligned}
B_1 &= \{beer, screws, hammer\} \\
B_2 &= \{saw, lugnuts\} \\
B_3 &= \{beer, screws, hammer, lugnuts\} \\
B_4 &= \{beer, screws, hammer, router\}
\end{aligned}$$

If our support $s = 2$ then we're looking for all subsets of items that appear in at least 2 of the baskets. In particular:

- $\{lugnuts\}$ as it appears in B_2, B_3
- $\{beer\}$ as it appears in B_1, B_3, B_4
- $\{hammer\}$ as it appears in B_1, B_3, B_4
- $\{screws\}$ as it appears in B_1, B_3, B_4
- $\{beer, screws\}$ as it appears in B_1, B_3, B_4
- $\{hammer, screws\}$ as it appears in B_1, B_3, B_4
- $\{hammer, beer\}$ as it appears in B_1, B_3, B_4

This problem is widely seen in commerce, language analysis, search, and financial applications to learn *association rules*. For example, a customer analysis may show that people often buy nails when they buy a hammer (the pair has high support), so run a sale on hammers and jack up the price of nails!

For this exercise, you will write a program that takes a brute force approach (with pruning) and *finds* the support for each possible subset $S \subseteq A$. If there is zero support, you will omit it from your output. Essentially you are solving the problem for $s = 1$, but we're not only interested in the actual sets, but the sets *and* the number of baskets that they appear in.

Write a solution to this problem in the high-level programming language of your choice.

5 Divide & Conquer Style Algorithms

5.1 Introduction

Divide & conquer approach already seen in other algorithms: Merge Sort, Quick Sort, Binary Search, etc.

Google's MapReduce

5.2 Problems & Algorithms

5.3 Repeated Squaring

- Wish to compute $a^n \bmod m$
- Brute force: n multiplications
- Basic idea: square and repeat to reduce total number of multiplications

Example:

$$\begin{aligned} a^{77} &= a^{64} \cdot a^{13} \\ &= a^{64} \cdot a^{8} \cdot a^{5} \\ &= a^{64} \cdot a^{8} \cdot a^{4} \cdot a^{1} \end{aligned}$$

Observe that we can compute:

$$\begin{aligned} a^2 &= a \cdot a \\ a^4 &= a^2 \cdot a^2 \\ a^8 &= a^4 \cdot a^4 \\ a^{16} &= a^8 \cdot a^8 \\ a^{32} &= a^{16} \cdot a^{16} \\ a^{64} &= a^{32} \cdot a^{32} \end{aligned}$$

Which only requires 6 multiplications, then a^{77} can be computed with an additional 3 multiplications (instead of $2^{77} = 1.511 \times 10^{23}$).

Formally, we note that

$$\begin{aligned} \alpha^n &= \alpha^{b_k 2^k + b_{k-1}2^{k-1} + \cdots + b_1 2 + b_0} \\ &= \alpha^{b_k 2^k} \times \alpha^{b_{k-1}2^{k-1}} \times \cdots \times \alpha^{2b_1} \times \alpha^{b_0} \end{aligned}$$

So we can compute α^n by evaluating each term as

$$\alpha^{b_i 2^i} = \begin{cases} \alpha^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$$

We can save computation because we can simply square previous values:

$$\alpha^{2^i} = (\alpha^{2^{i-1}})^2$$

We still evaluate each term independently however, since we will need it in the next term (though the accumulated value is only multiplied by 1).

```
INPUT  : Integers $\alpha, m$ and $n = (b_k b_{k-1} \ldots b_1 b_0)$ in binary.
OUTPUT : $\alpha^n$ mod $m$
term = $\alpha$
IF $(b_0 = 1)$ THEN
    product $\leftarrow \alpha$
END
ELSE
    product $\leftarrow 1$
END
FOR $i = 1 \ldots k$ DO
    term $\leftarrow$ term $\times$ term mod $m$
    IF $(b_i = 1)$ THEN
        product $\leftarrow$ product $\times$ term mod $m$
    END
END
output product
```

Algorithm 28: Binary Exponentiation

Example: Compute 12^{26} mod 17 using Modular Exponentiation.

1	1	0	1	0	$= (26)_2$
4	3	2	1	-	i
1	16	13	8	12	term
9	9	8	8	1	product

Thus,

$$12^{26} \bmod 17 = 9$$

5.4 Euclid's GCD Algorithm

TODO: remove this (redundancy from intro)?

Recall that we can find the gcd (and thus lcm) by finding the prime factorization of the two integers.

However, the only algorithms known for doing this are exponential (indeed, computer security *depends* on this).

We can, however, compute the gcd in polynomial time using *Euclid's Algorithm*.

Consider finding the $\gcd(184, 1768)$. Dividing the large by the smaller, we get that

$$1768 = 184 \cdot 9 + 112$$

Using algebra, we can reason that any divisor of 184 and 1768 must also be a divisor of the remainder, 112. Thus,

$$\gcd(184, 1768) = \gcd(184, 112)$$

Continuing with our division we eventually get that

$$\begin{aligned}\gcd(184, 1768) &= \gcd(184, 112)\\ &= \gcd(112, 72)\\ &= \gcd(72, 40)\\ &= \gcd(40, 32)\\ &= \gcd(32, 8) = 8\end{aligned}$$

This concept is formally stated in the following Lemma.

Lemma 7. Let $a = bq + r$, $a, b, q, r \in \mathbb{Z}$, then

$$\gcd(a, b) = \gcd(b, r)$$

```
INPUT   : Integers, $a, b \in \mathbb{Z}^+$
OUTPUT : $\gcd(a, b)$
1 WHILE $b \neq 0$ DO
2     $t \leftarrow b$
3     $b \leftarrow a$ mod $t$
4     $a \leftarrow t$
5 END
6 output $a$
```

Algorithm 29: Euclid's Simple GCD Algorithm

Analysis:

- Number of iterations is dependent on the nature of the input, not just the input size
- Generally, we're interested in the *worst case* behavior
- Number of iterations is maximized when the reduction in b (line 3) is minimized
- Reduction is minimized when b is minimal; i.e. $b = 2$
- Thus, after at most n iterations, b is reduced to 1 (0 on the next iteration), so:

$$\frac{b}{2^n} = 1$$

- The number of iterations, $n = \log b$

The algorithm we present here is actually the *Extended* Euclidean Algorithm. It keeps track of more information to find integers such that the gcd can be expressed as a *linear combination.*

Theorem 4. If a and b are positive integers, then there exist integers s, t such that

$$\gcd(a, b) = sa + tb$$

```
INPUT   : Two positive integers $a, b$.
OUTPUT : $r = \gcd(a, b)$ and $s, t$ such that $sa + tb = \gcd(a, b)$.
1  $a_0 = a$, $b_0 = b$
2  $t_0 = 0$, $t = 1$
3  $s_0 = 1$, $s = 0$
4  $q = \lfloor \frac{a_0}{b_0} \rfloor$
5  $r = a_0 - qb_0$
6  WHILE $r > 0$ DO
7      temp $= t_0 - qt$
8      $t_0 = t, t =$ temp
9      temp $= s_0 - qs$
10     $s_0 = s, s =$ temp
11     $a_0 = b_0$, $b_0 = r$
12     $q = \lfloor \frac{a_0}{b_0} \rfloor$, $r = a_0 - qb_0$
13     IF $r > 0$ THEN
14         gcd $= r$
15     END
16 END
17 output gcd, $s, t$
```

Algorithm 30: EXTENDEDEUCLIDEANALGORITHM

a_0	b_0	t_0	t	s_0	s	q	r
27	58	0	1	1	0	0	27
58	27	1	0	0	1	2	4
27	4	0	1	1	-2	6	3
4	3	1	-6	-2	13	1	1
3	1	-6	7	13	-15	3	0

Therefore,

$$\gcd(27, 58) = 1 = (-15)27 + (7)58$$

Example:
Compute $\gcd(25480, 26775)$ and find s, t such that

$$\gcd(25480, 26775) = 25480s + 26775t$$

a_0	b_0	t_0	t	s_0	s	q	r
25480	26775	0	1	1	0	0	25480
26775	25480	1	0	0	1	1	1295
25480	1295	0	1	1	-1	19	875
1295	875	1	-19	-1	20	1	420
875	420	-19	20	20	-21	2	35
420	35	20	-59	-21	62	12	0

Therefore,

$$\gcd(25480, 26775) = 35 = (62)25480 + (-59)26775$$

5.5 Peasant Multiplication

5.6 Karatsuba Multiplication

Straightforward multiplication of two n-bit integers, a, b requires $O(n^2)$ multiplications and $O(n)$ additions.

Observe:

$$47 \cdot 51 = (4 \times 10^1 + 7 \times 10^0) \cdot (5 \times 10^1 + 1 \times 10^0)$$

Requires 4 multiplications; we can rewrite using the FOIL rule:

$$47 \cdot 51 = (4 \cdot 5) \times 10^2 + (4 \cdot 1 + 7 \cdot 5) \times 10^1 + (7 \cdot 1) \times 10^0$$

But observe, the inner multiplication:

$$(4 \cdot 1 + 7 \cdot 5) = (4 + 7) \cdot (1 + 5) - (4 \cdot 5) - (7 \cdot 1)$$

In general we have that

$$(a + b)(c + d) = ac + ad + bc + bd$$

But we're interested in $ad + bc$:

$$ad + bc = (a + b)(c + d) - ac - bd$$

which only requires 3 multiplications.

In general given a, b, split them into four sub numbers:

$$a = a_1 \cdot 10^{n/2} + a_0$$

$$b = b_1 \cdot 10^{n/2} + b_0$$

That is, a_1 are the higher order bits and a_0 are the lower order bits.

$$\begin{aligned}
a \cdot b &= \big(a_1 \cdot 10^{n/2} + a_0\big) \cdot \big(b_1 \cdot 10^{n/2} + b_0\big) \\
&= a_1 \cdot b_1 \cdot 10^n + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 10^{n/2} + a_0 \cdot b_0 \\
&= a_1 \cdot b_1 \cdot 10^n + \big[(a_1 + a_0)(b_1 + b_0) - a_1 \cdot b_1 - a_0 \cdot b_0\big] \cdot 10^{n/2} + a_0 \cdot b_0
\end{aligned}$$

Observations:

- Due to Karatsuba (1960)
- Powers of 10 are a simple shift
- If a, b are not powers of two, we can "pad" them out with leading zeros
- Three recursive calls to compute the multiplication
- Overhead in recursion/book keeping only makes this better for *large* numbers (thousands of digits)

$$M(n) = 3 \cdot M\left(\frac{n}{2}\right) + 0$$

By the Master Theorem:

$$M(n) \in \Theta(n^{\log_2 3})$$

As $\log_2(3) = 1.5849625\ldots$, this is an asymptotic improvement over the naive multiplication.

Example:

$$a \cdot b = 72,893,424 \cdot 33,219,492$$

$$\begin{aligned}
a_1 &= 7,289 \\
a_0 &= 3,424 \\
b_1 &= 3,321 \\
b_0 &= 9,492
\end{aligned}$$

Then (after recursion):

$$\begin{aligned}
a_1 \cdot b_1 &= 24,206,769 \\
a_0 \cdot b_0 &= 32,500,608 \\
x = a_1 + a_0 &= 10,719 \\
y = b_1 + b_0 &= 12,813 \\
x \cdot y &= 137,342,547
\end{aligned}$$

Altogether:

$$\begin{aligned}
a \cdot b &= 24,206,769 \cdot 10^8 + (137,342,547 - 24,206,769 - 32,500,608) \cdot 10^4 + 32,500,608 \\
&= 2,421,483,284,200,608
\end{aligned}$$

Other, similar schemes:

- Toom-Cook (1963): $O(n^{1.465})$

- Schönhage-Strassen (1968, FFT): $O(n \log n \log \log n)$
- Fürer (2007): $O(n \log n 2^{\log^* n})$ (iterated log function)

5.7 Strassen's Matrix Multiplication

Matrix multiplication is well defined (we restrict our examination to square $n \times n$ matrices); $C = A \cdot B$ where

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \cdot b_{k,j}$$

This constitutes $\Theta(n^3)$ additions/multiplications.

Rather than the brute-force approach, we can use the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$\begin{aligned}
m_1 &= (a_{00} + a_{11}) \cdot (b_{00} + b_{11}) \\
m_2 &= (a_{10} + a_{11}) \cdot b_{00} \\
m_3 &= a_{00} \cdot (b_{01} - b_{11}) \\
m_4 &= a_{11} \cdot (b_{10} - b_{00}) \\
m_5 &= (a_{00} + a_{01}) \cdot b_{11} \\
m_6 &= (a_{10} - a_{00}) \cdot (b_{00} + b_{01}) \\
m_7 &= (a_{01} - a_{11}) \cdot (b_{10} + b_{11})
\end{aligned}$$

The number of multiplications and additions for a 2×2 matrix using Strassen's formulas are 7 and 18 respectively. Contrast with 8 and 4 operations for the brute force method.

We're doing a lot more in terms of additions, and only slightly less in terms of multiplications–but it turns out that this is still asymptotically better than the brute force approach.

Strassen's formulas can be generalized to matrices rather than just scalars

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

$$= \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

Where M_i are matrices defined just as the scalars were.

There is an implicit assumption in the general formulas: that a matrix can be evenly divided into four equal parts, forcing us to consider matrices of size $2^k \times 2^k$.

In fact, *any* sized matrix can be evaluated using the formulas–the trick is to "pad-out" the matrix with zero entries to get a $2^k \times 2^k$ matrix.

Say we have an $n \times n$ matrix. We have to make 7 recursive calls to Strassen's algorithm to evaluate a term in each of the $M_1, \ldots, M_7$ formulas. Each call halves the size of the input ($\frac{n^2}{4} = \frac{n}{2}$). When a matrix reaches a size of 1, no multiplications are necessary. This suggests the following recurrence:

$$M(n) = 7M\left(\frac{n}{2}\right), M(1) = 0$$

Working this recurrence relation gives us

$$M(n) = n^{\log_2 7} \approx n^{2.807}$$

For additions, the same 7 matrices of size $n/2$ must be recursively dealt with while 18 additions must be made on matrices of size $n/2$, so

$$A(n) = 7A(n/2) + 18(n/2)^2, A(1) = 0$$

By the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$.

Is this better than the brute force method?

	Multiplications	Additions
Brute-Force	$\Theta(n^3)$	$\Theta(n^3)$
Strassen's	$\Theta(n^{\log_2 7})$	$\Theta(n^{\log_2 7})$

Table 5.1: Relative complexity for two matrix multiplication algorithms

Many other algorithms have been discovered with the best being Winogard's algorithm at $\mathcal{O}(n^{2.375477})$.

More recently this was improved (analytically) to $O(n^{2.3727})$ (Williams 2012 [15])

However, the overhead and multiplicative constants for these algorithms make them unfeasible for even moderately sized matrices.

5.8 Closest Pair Revisited

Without loss of generality, we can assume that the given point set, $P = \{p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)\}$ are sorted in ascending order with respect to x coordinates, $x_1 \leq x_2 \leq \ldots \leq x_n$.

Then we have the following divide and conquer strategy:

- **Divide**: Partition the points into two subsets:

$$\begin{aligned} P_1 &= \{(x_1, y_1), (x_2, y_2), \ldots (x_{n/2}, y_{n/2})\} \\ P_2 &= \{(x_{n/2+1}, y_{n/2+1}), \ldots (x_n, y_n)\} \end{aligned}$$

- **Conquer:** Find the two closest pair of points in each set P_1, P_2 recursively
- **Combine:** We take the minimum distance between the two subproblems, d_1, d_2; *however*, we must also check points with distances that *cross* our dividing line

From the dividing line, we need only check points in two regions: $[y - d, y + d]$ where $d = \min\{d_1, d_2\}$. For each point, one needs only check, *at most* 6 points (see figure 4.6, p149).

Such a check takes linear time–there are $n/2$ points (we need only check C_1) and up to 6 distance calculations each.

This suggests the usual recurrence,

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

which by the Master Theorem $\Theta(n \log n)$.

5.9 Convex Hull Revisited

Levitin describes a “Quick Hull” algorithm. Here we present a different divide & conquer approach. Yet another approach is the Graham Scan (a “walk” approach).

Basic Idea:

1. **Divide**: partition into two sets A, B according to a lower/upper half with respect to the x coordinate
2. **Conquer**: Compute the convex hull of A and B
3. **Combine**: Combine into a larger convex hull by computing the upper/lower tangents of each and throwing out all internal points ($O(n)$ operation)

Combine procedure:

- Starting with the right-most point a of the lower hull and the left-most point of the upper hull
- Iterate through the hull points clockwise in the lower hull and counter clockwise on the upper partition, alternating between the two
- Repeat until the tangent $\overline{ab}$ has been found
- Tangent has been found when the "next" point in each hull is contained in the new hull
- Only required to look at the immediate next point (if the next point is above or below, stop)
- Repeat the process for the upper tangent

5.10 Fast Fourier Transform

- Article: `http://blog.revolutionanalytics.com/2014/01/the-fourier-transform-explained-i` `html`
- Nice Visualization: `http://bbrennan.info/fourier-work.html`

5.11 Exercises

Exercise 5.1. Implement Strassen's algorithm in the high-level programming language of your choice.

Exercise 5.2. Consider the following problem: Given an array of sorted integers and two values l, u (lower and upper bounds), find the indices i, j such that $l \leq A[k] \leq u$ for all k, $i \leq k \leq j$. Solve this problem by developing an $O(\log n)$ algorithm. Give good pseudocode and analyze your algorithm.

Exercise 5.3. Adaptive Quadrature is a numerical method for approximating the definite integral of a function $f(x)$:

$$\int_a^b f(x)\,dx$$

It works as follows: given a, b and a tolerance τ, it approximates the integral using some method. It then computes an error bound for that method and if the error is less than τ then the estimation is used. If the error is greater than τ, then the midpoint $m = \frac{a+b}{2}$ is used to recursively integrate on the intervals $[a, m]$ and $[m, b]$, updating τ to $\frac{\tau}{2}$.

For our purposes, use Simpson's rule to approximate:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

which has a known error bound of:

$$\frac{1}{90}\left(\frac{b-a}{2}\right)^5 \left|f^{(4)}(\xi)\right|,$$

Write good pseudocode for this algorithm and analyze its complexity.

Exercise 5.4. Let A be a zero-index array that is sorted. Now suppose that someone comes along and cyclicly "shifts" the contents of A so that the contents are moved some number of positions, with those at the end being shifted back to the beginning. For example, if the array contained the elements $2, 6, 10, 20, 30, 40$ and it were shifted by 3 positions, the resulting array would be $20, 30, 40, 2, 6, 10$ (in general, you will not be given how many positions the array has been shifted).

Design an *efficient* (that is an $O(\log n)$) algorithm that can compute the *median* element of A. Give good pseudocode and a short explanation. Analyze your algorithm.

6 Linear Systems

6.1 Introduction

TODO

6.2 Solving Linear Systems

A system of linear equations is a collection of n equalities in n unknowns. An example:

$$\begin{aligned}3x + 2y &= 35\\ x + 3y &= 27\end{aligned}$$

We are interested in solving such systems algorithmically. A generalized form:

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1\\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2\\ &\vdots\\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n\end{aligned}$$

In general, we want to eliminate variables until we have 1 equation in 1 unknown.

The goal of *Guassian Elimination* is to transform a system of linear equations in n unknowns to an equivalent, *upper-triangular* system.

An upper triangular system is one in which everything below and to the left of the diagonal on the matrix is zero. In general we would want something like

$$\begin{array}{rrrrcl}a'_{11}x_1+ & a'_{12}x_2+ & \ldots+ & a'_{1n}x_n & = & b'_1\\ & a'_{22}x_2+ & \ldots+ & a'_{2n}x_n & = & b'_2\\ & & & & & \vdots\\ & & & a'_{nn}x_n & = & b'_n\end{array}$$

A system of linear equations can be thought of as an $n \times n$ matrix with a variable vector $x = [x_1, x_2, \ldots x_n]$ and a solution vector $b = [b'_1, b'_2, \ldots b'_n]$. In other words, we can equivalently solve the matrix equation

$$Ax = b \to A'x = b'$$

An upper-triangular system is easier to solve because we can easily back-track (from the bottom right) and solve each unknown. This is known as *backward substitution* (not to be confused with the method for solving recurrence relations).

Now that we know how to back-solve, how do we take a given system and transform it to an upper-triangular system? We use several elementary row operations:

- We can exchange any two *rows* of a system
- We can exchange any two *columns* of a system [1]
- We can multiply an equation by a non-zero scalar
- We can add or subtract one equation with another (or a multiple thereof)

Such operations *do not* change the solution to the system.

Example

Consider the following system of linear equations:

$$\begin{aligned} 3x_1 - x_2 + 2x_3 &= 1 \\ 4x_1 + 2x_2 - x_3 &= 4 \\ x_1 + 3x_2 + 3x_3 &= 5 \end{aligned}$$

$$\begin{bmatrix} 3 & -1 & 2 & 1 \\ 4 & 2 & -1 & 4 \\ 1 & 3 & 3 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 3 & -1 & 2 & 1 \\ 4 & 2 & -1 & 4 \\ 1 & 3 & 3 & 5 \end{bmatrix} \begin{array}{c} \Rightarrow \\ \text{subtract } \frac{4}{3} \text{ times row 1} \\ \text{subtract } \frac{1}{3} \text{ times row 1} \end{array}$$

$$\begin{bmatrix} 3 & -1 & 2 & 1 \\ 0 & \frac{10}{3} & -\frac{11}{3} & \frac{8}{3} \\ 0 & \frac{10}{3} & \frac{7}{3} & \frac{14}{3} \end{bmatrix} \begin{array}{c} \Rightarrow \\ \\ \text{subtract 1 times row 2} \end{array}$$

$$\begin{bmatrix} 3 & -1 & 2 & 1 \\ 0 & \frac{10}{3} & -\frac{11}{3} & \frac{8}{3} \\ 0 & 0 & 6 & 2 \end{bmatrix}$$

[1]If we do, however, we must make note that the *variables* have also been exchanged. For instance if we exchange column i and column j then x_i and x_j have exchanged.

Back solving gives us:

$$\vec{x} = (x_1, x_2, x_3) = \left(\frac{1}{2}, \frac{7}{6}, \frac{1}{3}\right)$$

Using elementary row operations, we can proceed as follows. Using a_{11} as a pivot, we subtract a_{21}/a_{11} times the first equation from the second equation, and continue with a_{31}/a_{11} from the third, etc. In general, we subtract a_{i1}/a_{11} times row 1 from the i-th row for $2 \leq i \leq n$. We repeat for a_{22} as the next pivot, etc.

Issues:

- It may be the case that we attempt to divide by zero, which is invalid. To solve this problem, we can exchange a row!
- Though a divisor may not be zero, it could be so small that our quotient is too large. We can solve this one of two ways– either scale the current row so that it becomes 1 or we can again exchange rows. Specifically, the row with the largest absolute value in the i-th row.
- The system may be *indeterminant*: there may be one or more free-variables (an infinite number of solutions). Visually, both equations correspond to the same line (intersecting at an infinite number of points); example:

$$\begin{aligned} x + 7y &= 8 \\ 2x + 14y &= 16 \end{aligned}$$

- The system may be *inconsistent* (there are *no* solutions). Visually the equations correspond to two parallel lines that never intersect. Example:

$$\begin{aligned} x + 7y &= 8 \\ 2x + 14y &= 20 \end{aligned}$$

```
INPUT  : An $n \times n$ matrix $A$ and a $1 \times n$ vector $b$
OUTPUT : An augmented, upper triangluar matrix $A'$ preserving the solution to
         $A \cdot \vec{x} = \vec{b}$
$A' \leftarrow A|\vec{b}$ //augment
FOR $i = 1, \ldots n-1$ DO
    $pivotrow \leftarrow i$
    FOR $j = (i+1), \ldots, n$ DO
        IF $|A'[j,i]| > |A'[pivotrow, i]|$ THEN
            $pivotrow \leftarrow j$
        END
    END
    FOR $k = i, \ldots, (n+1)$ DO
        swap($A'[i,k], A'[pivotrow, k]$)
    END
    FOR $j = (i+1), \ldots, n$ DO
        $t \leftarrow \frac{A'[j,i]}{A'[i,i]}$
        FOR $k = i, \ldots, (n+1)$ DO
            $A'[j,k] \leftarrow A'[j,k] - A'[i,k] \cdot t$
        END
    END
END
output $A'$
```

Algorithm 31: (Better) Gaussian Elimination

```
INPUT  : An augmented, upper triangular $(n \times n+1)$ matrix $A$
OUTPUT : A solution vector $\vec{x} = (x_1, \ldots, x_n)$
FOR $i = n, \ldots 1$ DO
    $t \leftarrow A[i, n+1]$ //set to $b'_i$
    FOR $j = (i+1), \ldots, n$ DO
        $t \leftarrow t - x_j \cdot A[i,j]$
    END
    $x_i \leftarrow \frac{t}{A[i,i]}$
END
output $\vec{x}$
```

Algorithm 32: Back Solving

Analysis

The multiplication is our elementary operation. We thus have the following summation:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 \approx \frac{1}{3}n^3 \in \Theta(n^3)$$

Backward substitution will work on the remaining entries making about $\frac{n(n-1)}{2}$ multiplications and n divisions, so the asymptotic classification doesn't change.

6.2.1 LU Decomposition

Performing Gaussian Elimination on a matrix A can give us to additional matrices:

- L – A lower-triangular matrix with 1's on the diagonal and lower entries corresponding to the row multiples used in Gaussian elimination.
- U – The upper-triangular matrix as a result of Gaussian elimination on A.

LU-Decomposition of the previous example:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{4}{3} & 1 & 0 \\ \frac{1}{3} & 1 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 3 & -1 & 2 \\ 0 & \frac{10}{3} & -\frac{11}{3} \\ 0 & 0 & 6 \end{bmatrix}$$

It is not hard to prove that

$$A = L \cdot U$$

where $L \cdot U$ is the usual matrix multiplication. We observe then, that solving the system $Ax = b$ is equivalent to solving

$$LU\vec{x} = \vec{b}$$

We do this by performing Gaussian Elimination, then we solve the system $Ly = b$ where $y = Ux$ then we solve $Ux = y$.

Though this is essentially Gaussian elimination with a bit more work, it has the advantage that once we have the decomposition L and U, we can solve for *any* vector b—a huge advantage if we were to make changes to the $\vec{b}$ vector.

6.2.2 Matrix Inverse

The *inverse* of an $n \times n$ matrix A is another matrix A^{-1} such that

$$AA^{-1} = I$$

where I is the identity matrix (1's on the diagonal, 0's everywhere else). The inverse of a matrix is always unique. *However* not every matrix is invertible–such matrices are called *singular*.

Recall that when using Gaussian elimination, the system may be inconsistent. That is, no valid solution exists. One can prove that this is the case if and only if A is not invertible which in turn is the case if and only if one of the rows is a linear combination of another row.

Such a linear combination can be detected by running Gaussian Elimination on A. If one of the diagonal entries is 0 then A is singular. Otherwise A is consistent.

Finding an inverse is important because it allows us an implicit division operation among matrices[2]:

$$A^{-1}Ax = A^{-1}b$$
$$x = A^{-1}b$$

We can use Gaussian Elimination to find a matrix's inverse by solving

$$AX = I$$

where X are n^2 unknowns in the inverse matrix. We do this by solving n systems of linear equations with the same coefficient matrix A and a vector of unknowns x^j (the j-th column of X and a solution vector e^j (the j-th column of the identity matrix):

$$Ax^j = e^j$$

for $1 \leq j \leq n$

Alternatively, you can adapt Gaussian Elimination to compute an inverse as follows. Given A, you create a "super augmented" matrix,

$$[A|I]$$

Where I is the $n \times n$ identity matrix, resulting in an $n \times 2n$. Then you perform Gaussian Elimination to make the left part upper triangular, then perform a Gaussian Elimination for the upper part as well, then normalize each row so that each entry on the diagonal is 1 (this is actually Gauss-Jordan elimination). The resulting augmented matrix represents

$$[I|A^{-1}]$$

[2]be careful: remember, matrix multiplication is *not* commutative in general

6.2.3 Determinants

The *determinant* of an $n \times n$ square matrix is a measure of its coefficients which has several interpretations. With respect to *linear transformations*, it can quantify "area" or "volume".

The determinant has several equivalent definitions.

$$\det A = \sum_{\sigma \in S_n} \left[\operatorname{sgn}(\sigma) \prod_{i=1}^{n} A_{i,\sigma_i} \right]$$

where S_n is the symmetric group on n elements (the set of all permutations).

Alternatively, it can be recursively defined as:

$$\det A = \sum_{j=1}^{n} s_j a_{1j} \det A_j$$

where

$$s_j = \begin{cases} 1 & \text{if } j \text{ is odd} \\ -1 & \text{if } j \text{ is even} \end{cases}$$

- a_{1j} is the element in row 1 and column j
- A_j is the $(n-1) \times (n-1)$ matrix obtained by deleting row 1 and column j from A
- If $n = 1$ then $\det A = a_{11}$

Determinants are easy to compute for 2×2 matrices and even 3×3 matrices. There is also an easy to remember *visual* way to do it.

If you can easily compute a determinant you can easily tell if a matrix is invertible or not:

Theorem 5. An $n \times n$ matrix is invertible if and only if $\det A \neq 0$

If you were to use the definition to compute a determinant, you would be summing n! terms! Instead we can again use Gaussian elimination.

The determinant of an upper-triangular matrix is simply the product of the elements on its diagonal. Thus we can use Gaussian elimination to transform A into an upper-triangular matrix A' and simply compute

$$\det A' = \prod_{i=1}^{n} a'_{ii}$$

Some things to keep in mind about determinants when using Gaussian elimination:

- Realize that $\det(A + B) \neq \det A + \det B$

- However, it is true that $\det AB = \det A \det B$
- Also, $\det A = \det A^T$
- If we multiply a row by a constant c, then $\det A = \det A'/c$
- The value of $\det A$ does not change if we add a constant multiple of another row.
- If we exchange rows, the sign of $\det A$ flips
- The above hold for columns as well

Though not algorithmically sound (it is far too complex, cumbersome to program and certainly not stable[3]) Cramer's Rule does give us an explicit solution for a system of linear equations:

$$x_1 = \frac{\det A_1}{\det A}, x_2 = \frac{\det A_2}{\det A}, \ldots x_n = \frac{\det A_n}{\det A}$$

where A_j is the matrix obtained by replacing the j-th column of A by the vector b.

6.3 Linear Programming

A *linear program* is mathematical model for linear optimization problems.

- There is some *objective* function which you wish to maximize (or minimize, called the *dual*)
- There are *constraints* that cannot be violated
- Both must be *linear* functions
- In general there can be n variables
- Objective: output the value of n variables that is both *feasible* and *optimizes* the outcome

A linear program in *standard form*:

$$\begin{array}{llll} \text{maximize} & & \vec{c}^{\,T} \cdot \vec{x} & \\ \text{subject to} & Ax & \leq & \vec{b} \\ & \vec{x} & \geq & 0 \end{array}$$

Simplex Method: Example

The *simplex method* (Dantzig, 1947) is an old way of solving linear programs

- Start at an arbitrary point

[3]stable in the sense that floating point operations are prone to error and that error propagates

- Travel along the *feasible region* to the next vertex (in n-space)
- In general, the optimal solution lies on the *polytope* with $\binom{n}{m}$ extreme points (n variables, m constraints)
- May have an exponential number of such critical points
- Choice of direction: that which improves the solution the most (the direction in which the variable has the largest partial derivative)
- In general, pivot choices may lead to aberrant (exponential) behavior or could even cycle
- Very fast in practice
- Polynomial-time algorithm (Khachiyan, 1979) and better interior-point method (Karmarkar, 1984) among other improvements

$$\begin{array}{lrcl} \text{maximize} & 4x+3y & & \\ \text{subject to} & x & \geq & 0 \\ & y & \geq & 0 \\ & x & \leq & 9 \\ & 2y+x & \leq & 12 \\ & 2y-x & \leq & 8 \end{array}$$

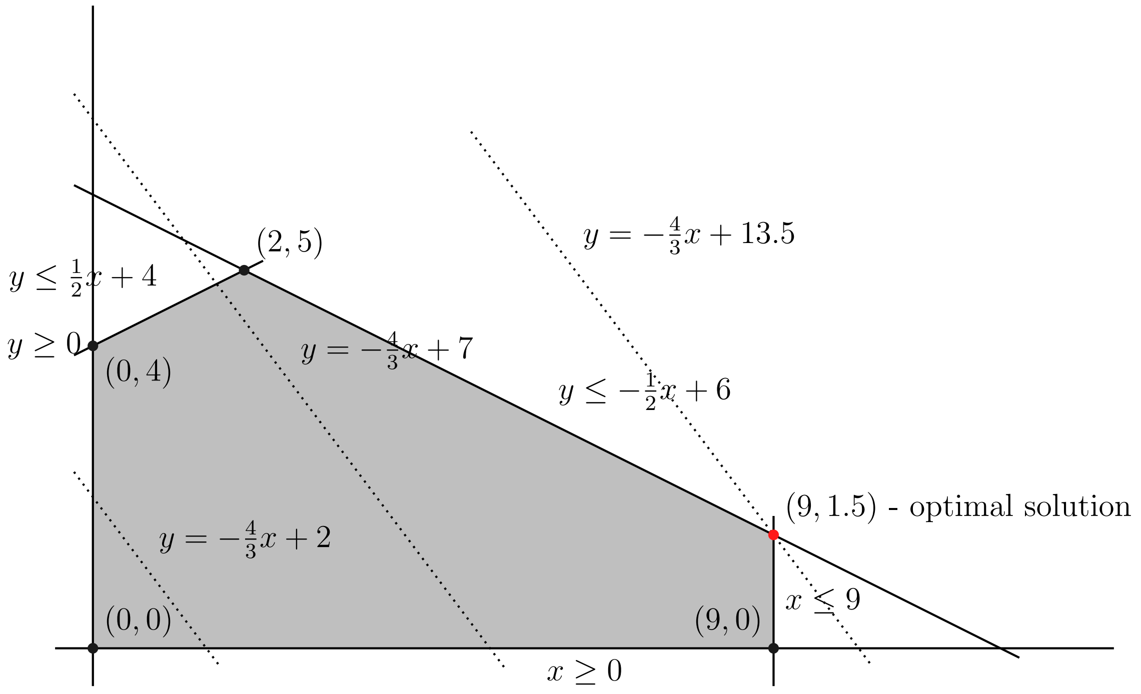


Figure 6.1: Visualization of a Linear Program with two sub-optimal solution lines and the optimal one.

Point	Value
$(0,0)$	0
$(0,4)$	12
$(2,5)$	23
$(9,1.5)$	40.5
$(9,0)$	36

Table 6.1: Function values at several critical points

6.3.1 Formulations

We can transform and conquer several previously examined problems as follows: we can reformulate the problem as an optimization problem (a linear program) and solve it with the simplex method or other algorithm for LP.

Knapsack

The Knapsack problem can be formulated as a linear program as follows.

- We want to maximize the sum of the value of all items
- Subject to the total weight constraint
- *But* we want to constrain the fact that we can take an item or not
- We establish n *indicator variables*, $x_1, \ldots, x_n$,

$$x_i \begin{cases} 1 & \text{if item } i \text{ is taken} \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{rl} \text{maximize} & \displaystyle\sum_{i=1}^{n} v_i \cdot x_i \\ \text{subject to} & \displaystyle\sum_{i=1}^{n} w_i \cdot x_i \leq W \\ & x_i \in \{0,1\} \quad 1 \leq i \leq n \end{array}$$

- This is known as an *Integer Linear Program*
- Some constraints are further required to be integers
- In general, ILP is NP-complete
- A linear program can be *relaxed*:
- Integer constraints are relaxed and allowed to be continuous

An example: the fractional knapsack problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} v_i \cdot x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i \cdot x_i \leq W \\ & 0 \leq x_i \leq 1 \quad 1 \leq i \leq n \end{aligned}$$

- This formulation doesn't solve the original 0-1 Knapsack problem, it could come up with a solution where a fraction of an item is taken

Assignment Problem & Unimodularity

Recall the assignment problem; we can formulate an ILP as follows

$$\begin{aligned} \text{minimize} \quad & \sum_{i} \sum_{j} c_{i,j} \cdot x_{i,j} \\ \text{subject to} \quad & \sum_{i} x_{i,j} = 1 \quad \forall j \\ & x_{i,j} \in \{0, 1\} \quad \forall i, j \end{aligned}$$

Where $x_{i,j}$ is an indicator variable for assigning task i to person j (with associated cost $c_{i,j}$)

This program has integer constraints (ILP) and is, in general, NP-complete. We could relax it:

$$\begin{aligned} \text{minimize} \quad & \sum_{i} \sum_{j} c_{i,j} \cdot x_{i,j} \\ \text{subject to} \quad & \sum_{i} x_{i,j} = 1 \quad \forall j \\ & 0 \leq x_{i,j} \leq 1 \quad \forall i, j \end{aligned}$$

- It doesn't make much sense to assign half a task to a quarter of a person (or maybe it does!)
- However, it turns out that certain problems are *unimodular*
- A matrix is *unimodular* if it is:

  - square
  - integer entries
  - has a determinant that is $-1, 1$

- Further it is *totally unimodular* if every square non-singular submatrix is unimodular (thus determinant is $0, -1, 1$)
- Fact: if an ILP's constraint matrix is totally unimodular, then its optimal solution is integral
- If an optimum exists, it is composed of integers: $\vec{x} = (x_1, \ldots, x_n)$, $x_i \in \mathbb{Z}$
- Fact: The assignment problem is *totally unimodular*
- Consequence: the relaxation of the assignment problem's ILP will have the *same optimal solution* as the original ILP!

6.4 Exercises

Exercise 6.1. Let T be a binary search tree and let $v_1, v_2, \ldots, v_n$ be a preorder traversal of its nodes. Design an algorithm that reconstructs the tree T given $v_1, \ldots, v_n$.

Exercise 6.2. Diagram each step of inserting the following keys into an initially empty AVL tree. Clearly indicate which type of rotations occur.

$$10, 15, 30, 25, 5, 20, 35, 40, 45, 100, 50$$

Exercise 6.3. Diagram each step of inserting the following keys into an initially empty 2-3 tree. Clearly indicate which type of promotions and node transformations occur.

$$10, 15, 30, 25, 5, 20, 35, 40, 45, 100, 50$$

Exercise 6.4. Consider the following questions regarding AVL trees.

(a) Considering adding the keys a, b, c to an empty AVL tree. How many possible orders are there to insert them and how many of those do *not* result in any rotations?

(b) Considering adding keys a, b, c, d, e, f, g into an AVL tree. How many ways could you insert these keys into an empty AVL tree such that no rotations would result?

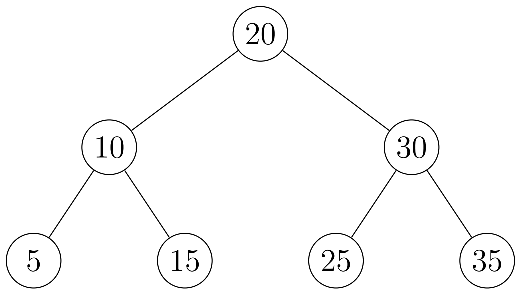
(c) Can you generalize this for keys $k_1, \ldots, k_m$ where $m = 2^n - 1$?

Exercise 6.5. Suppose we insert $1, 2, \ldots, 25$ into an initially empty AVL tree. What will be the depth of the resulting tree?

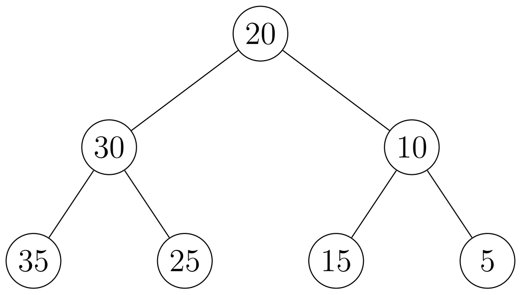
Exercise 6.6. Suppose we insert $1, 2, \ldots, 25$ into an initially empty 2-3 tree. What will be the depth of the resulting tree?

Exercise 6.7. Write an algorithm (give good pseudocode and a complete analysis) that *inverts* a binary search tree so that for each node with key k, all keys in the left-sub-tree are greater than k and all nodes in the right-sub-tree are less than k. Thus, an in-order traversal of an inverted tree will produce a key ordering in non-increasing order instead. An example of a binary search tree and its inversion is presented in Figure 6.2. Your algorithm must be efficient and should not create a new tree.

Exercise 6.8. Implement Gaussian Elimination in the high-level programming language of your choice. Use your code to solve matrix problems: finding a matrix inverse, solving a system of linear equations, etc.



(a) Binary Search Tree



(b) Inverted Binary Search Tree

Figure 6.2: Binary search tree and its inversion.

7 Trees

7.1 Introduction

Motivation: we want a data structure to store elements that offers efficient, arbitrary retrieval (search), insertion, and deletion.

- Array-based Lists
  - $O(n)$ insertion and deletion
  - Fast index-based retrieval
  - Efficient binary search if sorted
- Linked Lists
  - Efficient, $O(1)$ insert/delete for head/tail
  - Inefficient, $O(n)$ arbitrary search/insert/delete
  - Efficient binary search not possible without random access
- Stacks and queues are efficient, but are restricted access data structures
- Possible alternative: Trees
- Trees have the potential to provide $O(\log n)$ efficiency for all operations

7.2 Definitions & Terminology

- A *tree* is an acyclic graph
- For our purposes: a *tree* is a collection of *nodes* (that can hold keys, data, etc.) that are connected by *edges*
- Trees are also *oriented*: each node has a parent and children
- A node with no parents is the *root* of the tree, all child nodes are oriented downward
- Nodes not immediately connected can have an ancestor, descendant or cousin relationship

- A node with no children is a *leaf*
- A tree such that all nodes have at most two children is called a *binary tree*
- A binary tree is also oriented horizontally: each node may have a left and/or a right child
- Example: see Figure 7.1
- A *path* in a tree is a sequence nodes connected by edges
- The *length* of a path in a tree is the number of edges in the path (which equals the number of nodes in the path minus one)
- A path is *simple* if it does not traverse nodes more than once (this is the default type of path)
- The *depth* of a node u is the length of the (unique) path from the root to u
- The depth of the root is 0
- The depth of a tree is the maximal depth of any node in the tree (sometimes the term *height* is used)
- All nodes of the same depth are considered to be at the same *level*
- A binary tree is *complete* (also called *full* or *perfect*) if all nodes are present at all levels 0 up to its depth d
- A sub-tree rooted at a node u is the tree consisting of all descendants with u oriented as the root

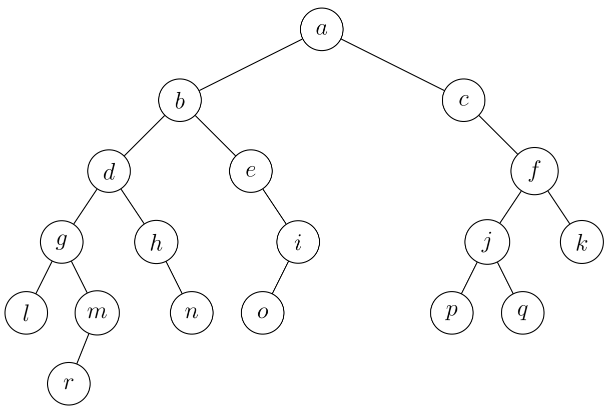


Figure 7.1: A Binary Tree

Properties:

- In a tree, all nodes are connected by exactly one unique path
- The maximum number of nodes at any level k is 2^k

- Thus, the maximum number of nodes n for any binary tree of depth d is:

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^{d-1} + 2^d = \sum_{k=0}^{d} 2^k = 2^{d+1} - 1$$

- Given a *full binary tree* with n nodes in it has depth:

$$d = \log(n+1) - 1$$

- That is, $d = O(\log n)$

Motivation: if we can create a tree-based data structure with operations proportional to its depth, then we could potentially have a data structure that allows retrieval/search, insertion, and deletion in $O(\log n)$-time.

7.3 Tree Traversal

- Given a tree, we need a way to enumerate elements in a tree
- Many algorithms exist to iterate over the elements in a tree
- We'll look at several variations on a depth-first-search

7.3.1 Preorder Traversal

- A preorder traversal strategy visits nodes in the following order: root; left-sub-tree; right-sub-tree
- An example traversal on the tree in Figure 7.1:

$$a, b, d, g, l, m, r, h, n, e, i, o, c, f, j, p, q, k$$

- Applications:
  - Building a tree, traversing a tree, copying a tree, etc.
  - Efficient stack-based implementation
  - Used in prefix notation (polish notation); used in languages such as Lisp/Scheme

Simulation of a stack-based preorder traversal of the binary tree in Figure 7.1.

`push` a

(enter loop)
`pop`, $node = a$
`push` c
`push` b
print a

(enter loop)
`pop`, $node = b$
`push` e
`push` d
print b

(enter loop)
`pop`, $node = d$
`push` h
`push` g
print d

(enter loop)
`pop`, $node = g$
`push` m
`push` l
print g

(enter loop)
`pop`, $node = l$
(no push)
(no push)
print l

(enter loop)
`pop`, $node = m$
(no push)
`push` r
print m

(enter loop)
`pop`, $node = r$
(no push)
(no push)
print r

(enter loop)
`pop`, $node = h$
`push` n
(no push)
print h

(enter loop)
`pop`, $node = n$
(no push)
(no push)
print n

(enter loop)
`pop`, $node = e$
`push` i
(no push)
print e

(enter loop)
`pop`, $node = i$
(no push)
`push` o
print i

(enter loop)
`pop`, $node = o$
(no push)
(no push)
print o

(enter loop)
`pop`, $node = c$
`push` f
(no push)
print c

(enter loop)
`pop`, $node = f$
`push` k
`push` j
print f

(enter loop)
`pop`, $node = j$
`push` q
`push` p
print j

(enter loop)
`pop`, $node = p$
(no push)
(no push)
print p

(enter loop)
`pop`, $node = q$
(no push)
(no push)
print q

(enter loop)
`pop`, $node = k$
(no push)
(no push)
print k

7.3.2 Inorder Traversal

- An inorder traversal strategy visits nodes in the following order: left-sub-tree; root; right-sub-tree
- An example traversal on the tree in Figure 7.1:

$$l, g, r, m, d, h, n, b, e, o, i, a, c, p, j, q, f, k$$

- Applications:
  - Enumerating elements in order in a binary search tree
  - Expression trees

Simulation of a stack-based inorder traversal of the binary tree in Figure 7.1.

(enter loop, $u = a$)
`push` a
update $u = b$

(enter loop, $u = b$)
`push` b
update $u = d$

(enter loop, $u = d$)
`push` d
update $u = g$

(enter loop, $u = g$)
`push` g
update $u = l$

(enter loop, $u = l$)
`push` l
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` l, update $u = l$
process l
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` g, update $u = g$
process g
update $u = m$

(enter loop, $u = m$)
`push` m
update $u = r$

(enter loop, $u = r$)
`push` r
update $u =$ `null`

(enter loop, $u =$ `null`)
`pop` r, update $u = r$
process r
update $u =$ `null`

(enter loop,

$u =$ `null`)
`pop` m, update
$u = m$
process m
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` d, update
$u = d$
process d
update $u = h$

(enter loop, $u = h$)
`push` h
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` h, update
$u = h$
process h
update $u = n$

(enter loop, $u = n$)
`push` n
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` n, update
$u = n$
process n
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` b, update
$u = b$
process b
update $u = e$

(enter loop, $u = e$)
`push` e
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` e, update
$u = e$
process e
update $u = i$

(enter loop, $u = i$)
`push` i
update $u = o$

(enter loop, $u = o$)
`push` o
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` o, update
$u = o$
process o
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` i, update
$u = i$
process i
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` i, update
$u = i$
process i
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` a, update
$u = a$
process a
update $u = c$

(enter loop, $u = c$)
`push` c
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` c, update
$u = c$
process c
update $u = f$

(enter loop, $u = f$)
`push` f
update $u = j$

(enter loop, $u = j$)
`push` j
update $u = p$

(enter loop, $u = p$)
`push` p
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` p, update
$u = p$
process p
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` j, update
$u = j$
process j
update $u = q$

(enter loop, $u = q$)
`push` q
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` q, update
$u = q$
process q
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` f, update
$u = f$
process f
update $u = k$

(enter loop, $u = k$)
`push` k
update $u =$ `null`

(enter loop,
$u =$ `null`)
`pop` k, update
$u = k$
process k
update $u =$ `null`

(done)

7.3.3 Postorder Traversal

- A postorder traversal strategy visits nodes in the following order: left-sub-tree; right-sub-tree; root
- An example traversal on the tree in Figure 7.1:

$$l, r, m, g, n, h, d, o, i, e, b, p, q, j, k, f, c, a$$

- Applications:
  - Topological sorting
  - Destroying a tree when manual memory management is necessary (roots are the last thing that get cleaned up)
  - Reverse polish notation (operand-operand-operator, unambiguous, used in old HP calculators)

– PostScript (Page Description Language)

Simulation of a stack-based postorder traversal of the binary tree in Figure 7.1:

$prev = \texttt{null}$
`push` a

(enter loop)
update $curr = (a)$
check: $prev = \texttt{null}$
`push` (b)
update $prev = a$

(enter loop)
update $curr = (b)$
check:
$prev.leftChild = curr$
$((b).leftChild = (b))$
`push` (d)
update $prev = b$

(enter loop)
update $curr = (d)$
check:
$prev.leftChild = curr$
$((d).leftChild = (d))$
`push` (g)
update $prev = d$

(enter loop)
update $curr = (g)$
check:
$prev.leftChild = curr$
$((g).leftChild = (g))$
`push` (l)
update $prev = g$

(enter loop)
update $curr = (l)$
check:
$prev.leftChild = curr$
$((l).leftChild = (l))$
(noop)
update $prev = l$

(enter loop)
update $curr = (l)$
check:
$prev.rightChild = curr$
$(null.rightChild = (l))$
process l
update $prev = l$

(enter loop)
update $curr = (g)$
check:
$prev.rightChild = curr$
$(null.rightChild = (g))$
check:
$curr.leftChild = prev$
$((l).leftChild = (l))$
`push` (m)
update $prev = g$

(enter loop)
update $curr = (m)$
check:
$prev.rightChild = curr$
$((m).rightChild = (m))$
`push` (r)
update $prev = m$

(enter loop)
update $curr = (r)$
check:
$prev.leftChild = curr$
$((r).leftChild = (r))$
(noop)
update $prev = r$

(enter loop)
update $curr = (r)$
check:
$prev.rightChild = curr$
$(null.rightChild = (r))$
process r
update $prev = r$

(enter loop)
update $curr = (m)$
check:
$prev.rightChild = curr$
$(null.rightChild = (m))$
check:
$curr.leftChild = prev$
$((r).leftChild = (r))$
update $prev = m$

(enter loop)
update $curr = (m)$
check:
$prev.rightChild = curr$
$(null.rightChild = (m))$
process m
update $prev = m$

(enter loop)
update $curr = (g)$
check:
$prev.rightChild = curr$
$(null.rightChild = (g))$
process g
update $prev = g$

(enter loop)
update $curr = (d)$
check:
$prev.rightChild = curr$
$((m).rightChild = (d))$
check:
$curr.leftChild = prev$
$((g).leftChild = (g))$
`push` (h)
update $prev = d$

(enter loop)
update $curr = (h)$
check:
$prev.rightChild = curr$
$((h).rightChild = (h))$
`push` (n)
update $prev = h$

(enter loop)
update $curr = (n)$
check:
$prev.rightChild = curr$
$((n).rightChild = (n))$
(noop)
update $prev = n$

(enter loop)
update $curr = (n)$
check:
$prev.rightChild = curr$
$(null.rightChild = (n))$
process n
update $prev = n$

(enter loop)
update $curr = (h)$
check:
$prev.rightChild = curr$
$(null.rightChild = (h))$
process h
update $prev = h$

(enter loop)
update $curr = (d)$
check:
$prev.rightChild = curr$
$((n).rightChild = (d))$
process d
update $prev = d$

(enter loop)
update $curr = (b)$
check:
$prev.rightChild = curr$
$((h).rightChild = (b))$
check:
$curr.leftChild = prev$
$((d).leftChild = (d))$
`push` (e)
update $prev = b$

(enter loop)
update $curr = (e)$
check:
$prev.rightChild = curr$
$((e).rightChild = (e))$
`push` (i)
update $prev = e$

(enter loop)
update $curr = (i)$
check:
$prev.rightChild = curr$
$((i).rightChild = (i))$
`push` (o)
update $prev = i$

(enter loop)
update $curr = (o)$
check:
$prev.leftChild = curr$
$((o).leftChild = (o))$
(noop)
update $prev = o$

(enter loop)
update $curr = (o)$
check:
$prev.rightChild = curr$
$(null.rightChild = (o))$
process o
update $prev = o$

(enter loop)
update $curr = (i)$
check:
$prev.rightChild = curr$
$(null.rightChild = (i))$
check:
$curr.leftChild = prev$
$((o).leftChild = (o))$
update $prev = i$

(enter loop)
update $curr = (i)$
check:
$prev.rightChild = curr$
$(null.rightChild = (i))$
process i
update $prev = i$

(enter loop)
update $curr = (e)$
check:
$prev.rightChild = curr$
$(null.rightChild = (e))$
process e
update $prev = e$

(enter loop)
update $curr = (b)$
check:
$prev.rightChild = curr$
$((i).rightChild = (b))$
process b
update $prev = b$

(enter loop)
update $curr = (a)$
check:
$prev.rightChild = curr$
$((e).rightChild = (a))$
check:
$curr.leftChild = prev$
$((b).leftChild = (b))$
`push` (c)
update $prev = a$

(enter loop)
update $curr = (c)$
check:
$prev.rightChild = curr$
$((c).rightChild = (c))$
`push` (f)
update $prev = c$

(enter loop)
update $curr = (f)$
check:
$prev.rightChild = curr$
$((f).rightChild = (f))$
`push` (j)
update $prev = f$

(enter loop)
update $curr = (j)$
check:
$prev.leftChild = curr$
$((j).leftChild = (j))$
`push` (p)
update $prev = j$

(enter loop)
update $curr = (p)$
check:
$prev.leftChild = curr$
$((p).leftChild = (p))$
(noop)
update $prev = p$

(enter loop)
update $curr = (p)$
check:
$prev.rightChild = curr$
$(null.rightChild = (p))$
process p
update $prev = p$

(enter loop)
update $curr = (j)$
check:
$prev.rightChild = curr$
$(null.rightChild = (j))$
check:
$curr.leftChild = prev$
$((p).leftChild = (p))$
`push` (q)
update $prev = j$

(enter loop)
update $curr = (q)$
check:
$prev.rightChild = curr$
$((q).rightChild = (q))$
(noop)
update $prev = q$

(enter loop)
update $curr = (q)$
check:
$prev.rightChild = curr$
$(null.rightChild = (q))$
process q
update $prev = q$

(enter loop)
update $curr = (j)$
check:
$prev.rightChild = curr$
$(null.rightChild = (j))$
process j
update $prev = j$

(enter loop)
update $curr = (f)$
check:
$prev.rightChild = curr$
$((q).rightChild = (f))$
check:
$curr.leftChild = prev$
$((j).leftChild = (j))$
`push` (k)
update $prev = f$

(enter loop)
update $curr = (k)$
check:
$prev.rightChild = curr$
$((k).rightChild = (k))$
(noop)
update $prev = k$

(enter loop)
update $curr = (k)$
check:
$prev.rightChild = curr$
$(null.rightChild = (k))$
process k
update $prev = k$

(enter loop)
update $curr = (f)$
check:
$prev.rightChild = curr$
$(null.rightChild = (f))$
process f
update $prev = f$

(enter loop)
update $curr = (c)$
check:
$prev.rightChild = curr$
$((k).rightChild = (c))$
process c
update $prev = c$

(enter loop)
update $curr = (a)$
check:
$prev.rightChild = curr$
$((f).rightChild = (a))$
process a
update $prev = a$

7.3.4 Breadth-First Search Traversal

- Breadth-First Search (BFS) traversal is a general graph traversal strategy that explores local or close nodes first before traversing "deeper" into the graph
- When applied to an oriented binary tree, BFS explores the tree level-by-level (top-to-bottom, left-to-right)

7.3.5 Implementations & Data Structures

- Reference based implementation: `TreeNode<T>`
  - Owns (through composition) references to: `leftChild`, `rightChild`, `parent`
  - Can use either *sentinel* nodes or `null` to indicate missing children and parent
- `BinaryTree<T>` owns a `root`
- SVN examples: `unl.cse.bst`

Preorder Implementations

```
INPUT  : A binary tree node $u$
OUTPUT : A preorder traversal of the nodes in the subtree rooted at $u$
1 print $u$
2 preOrderTraversal($u \rightarrow leftChild$)
3 preOrderTraversal($u \rightarrow rightChild$)
```

Algorithm 33: Recursive Preorder Tree Traversal

Stack-based implementation:

- Initially, we push the tree's root into the stack
- Within a loop, we pop the top of the stack and process it
- We need to push the node's children for future processing
- Since a stack is LIFO, we push the right child first.

```
INPUT   : A binary tree, $T$
OUTPUT : A preorder traversal of the nodes in $T$
1 $S \leftarrow$ empty stack
2 push $T$'s root onto $S$
3 WHILE $S$ is not empty DO
4     $node \leftarrow$ S.pop
5     push $node$'s right-child onto $S$
6     push $node$'s left-child onto $S$
7     process $node$
8 END
```

Algorithm 34: Stack-based Preorder Tree Traversal

Inorder Implementation

Stack-based implementation:

- The same basic idea: push nodes onto the stack as you visit them
- However, we want to delay processing the node until we've explored the left-sub-tree
- We need a way to tell if we are visiting the node for the first time or returning from the left-tree exploration
- To achieve this, we allow the node to be null
- If null, then we are returning from a left-tree exploration, pop the top of the stack and process (then push the right child)
- If not null, then we push it to the stack for later processing, explore the left child

```
INPUT  : A binary tree, $T$
OUTPUT : An inorder traversal of the nodes in $T$
1  $S \leftarrow$ empty stack
2  $u \leftarrow root$
3  WHILE $S$ is not empty OR $u \neq null$ DO
4      IF $u \neq null$ THEN
5          push $u$ onto $S$
6          $u \leftarrow u$.leftChild
7      ELSE
8          $u \leftarrow S$.pop
9          process $u$
10         $u \leftarrow u$.rightChild
11     END
12 END
```

Algorithm 35: Stack-based Inorder Tree Traversal

Postorder Implementation

Stack-based implementation:

- Same basic ideas, except that we need to distinguish if we're visiting the node for the first time, second time or last (so that we can process it)
- To achieve this, we keep track of *where* we came from: a parent, left, or right node
- We keep track of a previous and a current node

```
INPUT   : A binary tree, $T$
OUTPUT : A postorder traversal of the nodes in $T$
$S \leftarrow$ empty stack
$prev \leftarrow null$
push root onto $S$
WHILE $S$ is not empty DO
    $curr \leftarrow S$.peek
    IF $prev = null$ Or $prev.$leftChild $= curr$ Or $prev.$rightChild $= curr$
    THEN
        IF $curr.$leftChild $\neq null$ THEN
            push $curr.$leftChild onto $S$
        ELSE IF $curr.$rightChild $\neq null$ THEN
            push $curr.$rightChild onto $S$
        END
    ELSE IF $curr.$leftChild $= prev$ THEN
        IF $curr.$rightChild $\neq null$ THEN
            push $curr.$rightChild onto $S$
        END
    ELSE
        process $curr$
        $S$.pop
    END
    $prev \leftarrow curr$
END
```

Algorithm 36: Stack-based Postorder Tree Traversal

BFS Implementation

```
Input  : A binary tree, $T$
Output : A BFS traversal of the nodes in $T$
1 $Q \leftarrow$ empty queue
2 enqueue $T$'s root into $Q$
3 while $Q$ is not empty do
4     $node \leftarrow$ Q.dequeue
5     enqueue $node$'s left-child onto $Q$
6     enqueue $node$'s right-child onto $Q$
7     print $node$
8 end
```

Algorithm 37: Queue-based BFS Tree Traversal

Tree Walk Implementations

- Simple rules based on local information: where you are and where you came from
- No additional data structures required
- Traversal is a "walk" around the perimeter of the tree
- Can use similar rules to determine when the current node should be processed to achieve pre, in, and postorder traversals
- Need to take care with corner cases (when current node is the root or children are missing)
- Pseudocode presented Algorithm 38

7.3.6 Operations

Basic Operations:

- Search for a particular element/key
- Adding an element
  - Add at most shallow available spot
  - Add at a random leaf
  - Add internally, shift nodes down by some criteria

```
INPUT   : A binary tree, $T$
OUTPUT : A Tree Walk around $T$
1  $curr \leftarrow root$
2  $prevType \leftarrow parent$
3  WHILE $curr \neq$ null DO
4      IF $prevType = parent$ THEN
           //preorder:  process curr here
5          IF $curr.leftChild$ exists THEN
               //Go to the left child:
6              $curr \leftarrow curr.leftChild$
7              $prevType \leftarrow parent$
8          ELSE
9              $curr \leftarrow curr$
10             $prevType \leftarrow left$
11         END
12     ELSE IF $prevType = left$ THEN
           //inorder:  process curr here
13         IF $curr.rightChild$ exists THEN
               //Go to the right child:
14             $curr \leftarrow curr.rightChild$
15             $prevType \leftarrow parent$
16         ELSE
17             $curr \leftarrow curr$
18             $prevType \leftarrow right$
19         END
20     ELSE IF $prevType = right$ THEN
           //postorder:  process curr here
21         IF $curr.parent =$ null THEN
               //root has no parent, we're done traversing
22             $curr \leftarrow curr.parent$
           //are we at the parent's left or right child?
23         ELSE IF $curr = curr.parent.leftChild$ THEN
24             $curr \leftarrow curr.parent$
25             $prevType \leftarrow left$
26         ELSE
27             $curr \leftarrow curr.parent$
28             $prevType \leftarrow right$
29         END
30     END
31 END
```

Algorithm 38: Tree Walk based Tree Traversal

- Removing elements
  - Removing leaves
  - Removing elements with one child
  - Removing elements with two children

Other Operations:

- Compute the total number of nodes in a tree
- Compute the total number of leaves in a tree
- Given an item or node, compute its depth
- Compute the depth of a tree

7.4 Binary Search Trees

Regular binary search trees have little structure to their elements; search, insert, delete operations are still linear with respect to the number of tree nodes, $O(n)$. We want a data structure with operations proportional to its depth, $O(d)$. To this end, we add structure and order to tree nodes.

- Each node has an associated *key*
- Binary Search Tree Property: For every node u with key u_k in T
  1. Every node in the left-sub-tree of u has keys *less* than u_k
  2. Every node in the right-sub-tree of u has keys *greater* than u_k
- Duplicate keys can be handled, but you must be consistent and not guaranteed to be contiguous
- Alternatively: do not allow duplicate keys or define a key scheme that ensures a *total order*
- Inductive property: all sub-trees are also binary search trees
- A full example can be found in Figure 7.2

7.4.1 Basic Operations

Observation: a binary search tree has more *structure*: the key in each node provides information on where a node is *not* located. We will exploit this structure to achieve $O(\log n)$ operations.

Search/retrieve

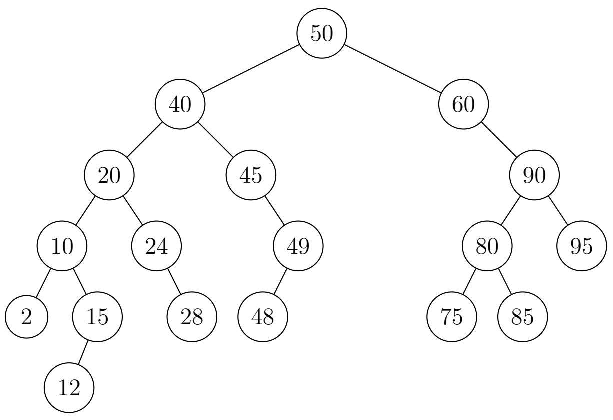


Figure 7.2: A Binary Search Tree

- Goal: find a node (and its data) that matches a given key k
- Start at the node
- At each node u, compare k to u's key, u_k:
  - If equal, element found, stop and return
  - If $k < u_k$, traverse to u's left-child
  - If $k > u_k$, traverse to u's right-child
- Traverse until the sub-tree is empty (element not found)
- Analysis: number of comparisons is bounded by the depth of the tree, $O(d)$

```
INPUT  : A binary search tree, $T$, a key $k$
OUTPUT : The tree node $u \in T$ whose key, $u_k$ matches $k$
1  $u \leftarrow T$'s root
2  WHILE $u \neq \phi$ DO
3      IF $u_k = k$ THEN
4          output $u$
5      END
6      ELSE IF $u_k > k$ THEN
7          $u \leftarrow u$'s left-child
8      ELSE IF $u_k < k$ THEN
9          $u \leftarrow u$'s left-child
10 END
11 output $\phi$
```

Algorithm 39: Search algorithm for a binary search tree

Insert

- Insert new nodes as leaves
- To determine where it should be inserted: traverse the tree as above
- Insert at the first available spot (first missing child node)
- Analysis: finding the available location is $O(d)$, inserting is just reference juggling, $O(1)$

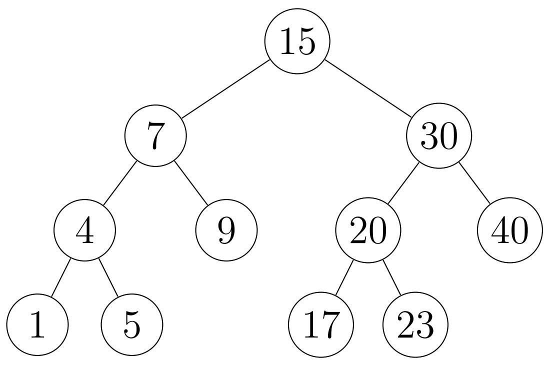
Delete

- Need to first *find* the node u to delete, traverse as above, $O(d)$
- If u is a leaf (no children): its safe to simply delete it
- If u has one child, then we can "promote" it to u's spot (u's parent will now point to u's child)
- If u has two children, we need to find a way to preserve the BST property
  - Want to minimally change the tree's structure
  - Need the operation to be efficient
  - Find the minimum element of the greater nodes (right sub-tree) or the maximal element of the lesser nodes (left sub-tree)
  - Such an element will have at most one child (which we know how to delete)
  - Delete it and store off the key/data
  - Replace u's key/data with the contents of the minimum/maximum element
- Analysis:
  - Search/Find: $O(d)$
  - Finding the min/max: $O(d)$
  - Swapping: $O(1)$
  - In total: $O(d)$
- Examples illustrated in Figure 7.3

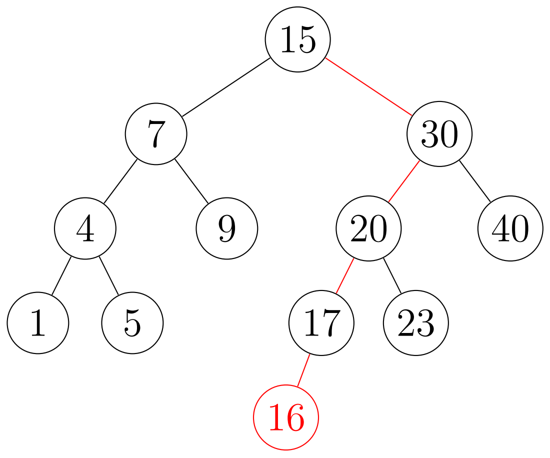
7.5 Balanced Binary Search Trees

In a Binary Search Tree (BST), insert, delete, and search operations are proportional to the *depth* of the tree.

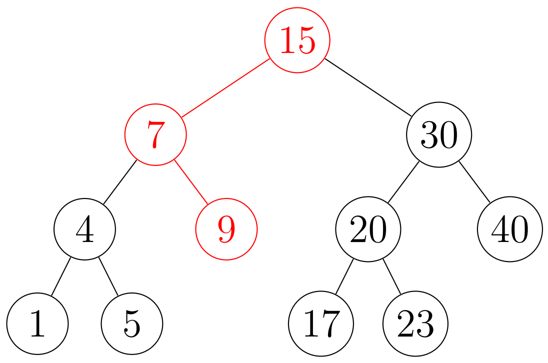
- The depth d of a tree is the length of the maximal path from the root to any leaf



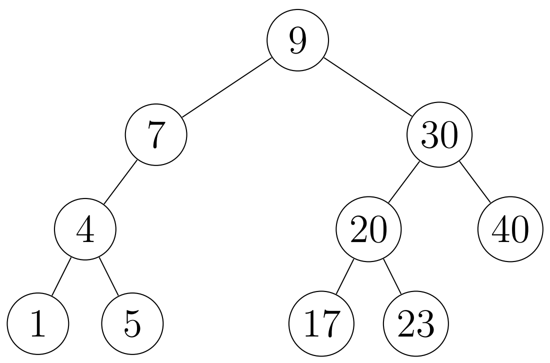
(a) A Binary Search Tree



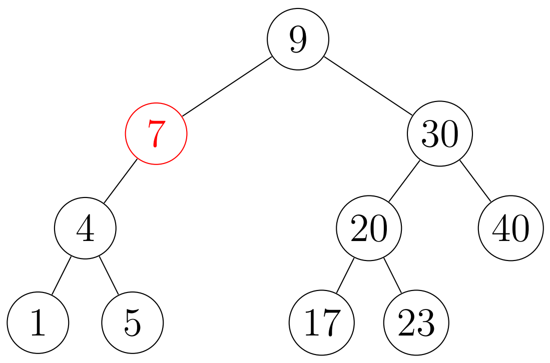
(b) Insertion of a new node (16) into a Binary Search Tree



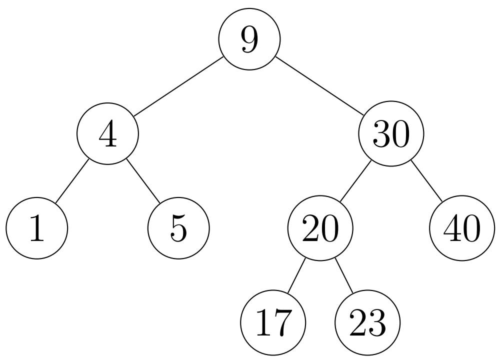
(c) Deletion of a node with two children (15). First step: find the maximum node in the left-sub-tree (lesser elements).



(d) Node 15 is replaced with the extremal node, preserving the BST property



(e) Deletion a node with only one child (7).



(f) Removal is achieved by simply promoting the single child/subtree.

Figure 7.3: Binary Search Tree Operations. Figure 7.3(b) depicts the insertion of (16) into the tree in Figure 7.3(a). Figures 7.3(c) and 7.3(d) depict the deletion of a node (15) with two children. Figures 7.3(e) and 7.3(f) depict the deletion of a node with only one child (7).

- Ideally, in a full/complete binary tree, $d \in \Theta(\log n)$
- In the worst case though, $d = n - 1 \in \Theta(n)$
- Our goal is to maintain a BST's *balance* so that the depth is *always* $O(\log n)$
- Many different types of *self-balancing* binary search trees: 2-3 trees, 2-3-4 trees, B-trees, AA trees, AVL trees, Red-Black Trees, Splay Trees, Scapegoat Trees, Treaps, T-Trees,

7.5.1 AVL Trees

Let T be a binary tree with u a node in T. The *height* of u is the length of the longest path from u to any descendant leaves in its left or right subtree. For example, in the tree in Figure 7.4, the node containing 6 has height 2 (as the longest path is from 6 to 4 which has length 2). The node containing 8 has height 3 and the root has height 4. Whereas the depth of a node is defined with respect to the root, the height of a node is defined with respect to leaves. The height of a tree (or subtree) is the height of its root node. By convention, the height of a single node tree is 0 and the height of an empty tree is -1.

The balance factor of a node u is equal to the height of u's left subtree minus the height of its right subtree.

$$\text{balance factor}(u) = h(T_L) - h(T_R)$$

A balance factor is a measure of how skewed or unbalanced a tree or subtree is. A balance factor of 0 indicates that a node is balanced between its left/right subtree. A "large" positive balance factor indicates a tree that is unbalanced to the left while a large negative balance factor indicates a tree that is unbalanced to the right. A node's balance factor only quantifies how well balanced a tree is at that particular node, not overall. That is, its a local property. Figure 7.4 depicts a BST with the balance factor indicated for each node. Observe that for the node containing 4, the left and right subtree both have a height of -1, so the balance factor is $-1 - (-1) = 0$. The balance factor of 8 is 3 since its left subtree has a height of 2 and its right subtree has a height of -1.

Definition 8. AVL Tree An *AVL Tree* is a binary search tree in which the *balance factor* of every node's left and right subtrees is 0, 1, or -1.

AVL represents the developer's names (Adelson-Velsky and Landis [8]).

Insertion into an AVL tree is done in the same way as a standard BST. However, an insertion may result in unbalanced nodes. First, we observe that since we only insert one node at a time, a node's balance factor can only ever become $+2$ or -2 (assuming it was an AVL tree to begin with). If several nodes become unbalanced, it is enough to consider the unbalanced node closest to the new node we just inserted. Correcting the imbalance at that node should correct the imbalance at other nodes.

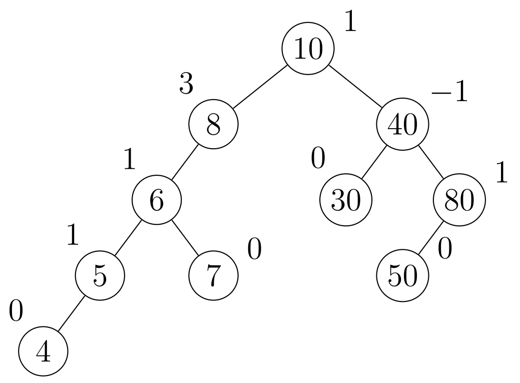


Figure 7.4: Balance Factor example on a Binary Search Tree

We need to re-balance the tree to ensure the AVL Tree properties are preserved after insertion. To do this, we use one of several *rotations* depending on the type of imbalance.

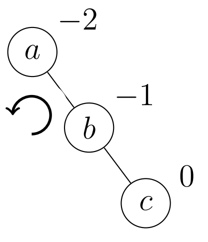
There are four basic rotations:

- *R* – A right rotation is used when three nodes are skewed all the way to the right. We rotate such that the middle one becomes the new root.
- *L* – A left rotation is used when three nodes are skewed all the way to the left. We rotate such that the middle node becomes the new root.
- *LR* – A double left-right rotation is done when the middle node is skewed to the left and its child is skewed to the right. We left-rotate at the middle node and then right rotate at the top node.
- *RL* – A double right-left rotation is done when the middle node is skewed to the right and its child is skewed to the left. We right-rotate at the middle node and then left rotate at the top node.

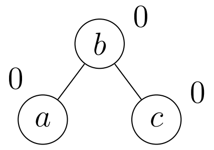
These rotations are depicted in simple trees in Figure 7.5 through 7.8 and the generalized rotations with subtrees present are presented in Figures 7.9 through 7.12.

To illustrate some of these operations, consider the example depicted Figure 7.13 where we insert the keys, $8, 4, 7, 3, 2, 5, 1, 10, 6$ into an initially empty tree.

Deleting from an AVL tree is the same as a normal BST. Leaves can be straightforwardly deleted, nodes with a single child can have the child promoted, and nodes with both children can have their key replaced with either the maximal element in the left subtree (containing the lesser elements) or the minimal element in the right subtree (containing the greater elements). However, doing so may unbalance the tree. Deleting a node may reduce the height of some subtree which will affect the balance factor of its ancestors. The same rotations may be necessary to rebalance ancestor nodes. Moreover, rebalancing may *also* reduce the height of subtrees. Thus, the rebalancing may propagate all the way back to the root node. An example of a worst case deletion is depicted in Figure 7.14.

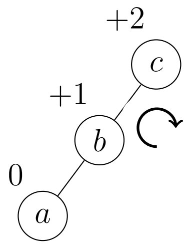


(a) Upon insertion of a new node c ($a < b < c$), the AVL tree becomes unbalanced at the root with a balance factor of -2.

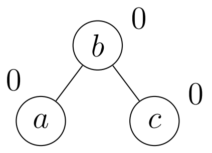


(b) AVL tree is rebalanced: b becomes the new root and the BST property is preserved.

Figure 7.5: Simple AVL L Rotation.

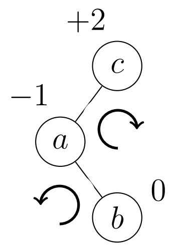


(a) Upon insertion of a new node a ($a < b < c$), the AVL tree becomes unbalanced at the root with a balance factor of $+2$.

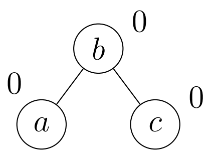


(b) AVL tree is rebalanced: b becomes the new root and the BST property is preserved.

Figure 7.6: Simple AVL R Rotation.

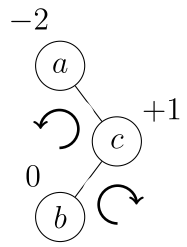


(a) Upon insertion of a new node b ($a < b < c$), the AVL tree becomes unbalanced at the root with a balance factor of $+2$, but its left-subtree is skewed to the right.

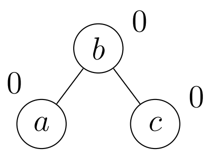


(b) AVL tree is rebalanced after a left rotation about a followed by a right rotation about c.

Figure 7.7: Simple AVL LR Rotation.

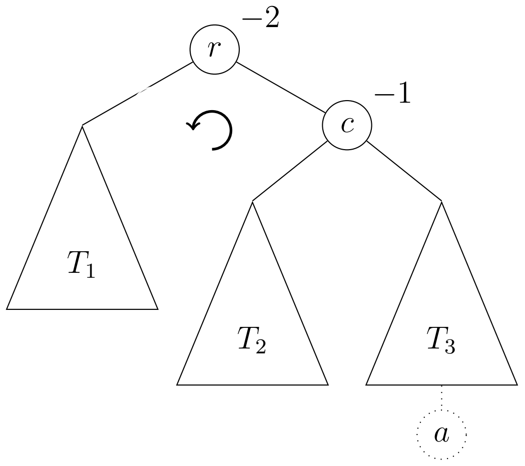


(a) Upon insertion of a new node b ($a < b < c$), the AVL tree becomes unbalanced at the root with a balance factor of -2, but its right-subtree is skewed to the left.

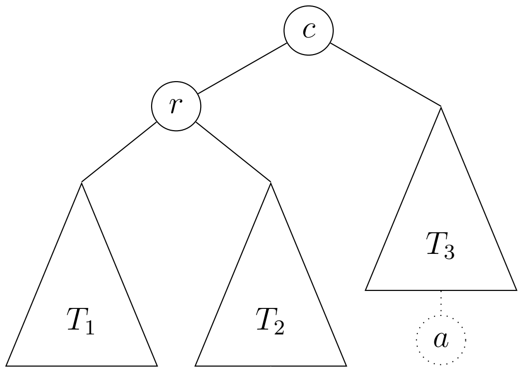


(b) AVL tree is rebalanced after a right rotation about c followed by a left rotation about a.

Figure 7.8: Simple AVL RL Rotation.

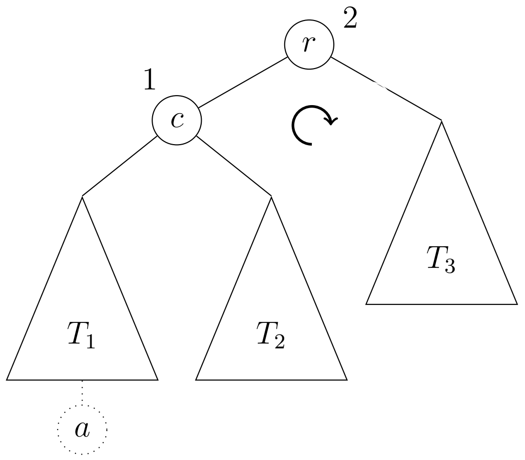


(a) Upon insertion of a new node a, the AVL Tree becomes unbalanced at r (which may be a subtree) with a balance factor of -2.

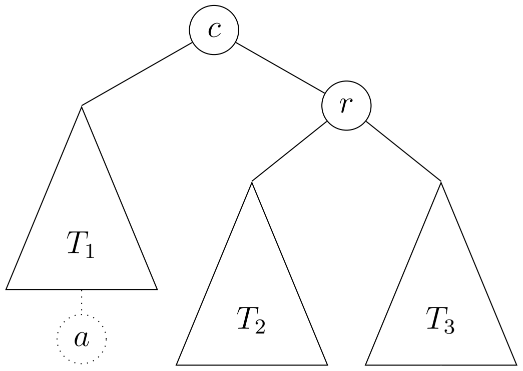


(b) AVL tree is rebalanced. The node c becomes the new root of the (sub)tree, r becomes c's left-child.

Figure 7.9: Generalized AVL L Rotation. Subtree T_2 "swings" over to become r's new right subtree.

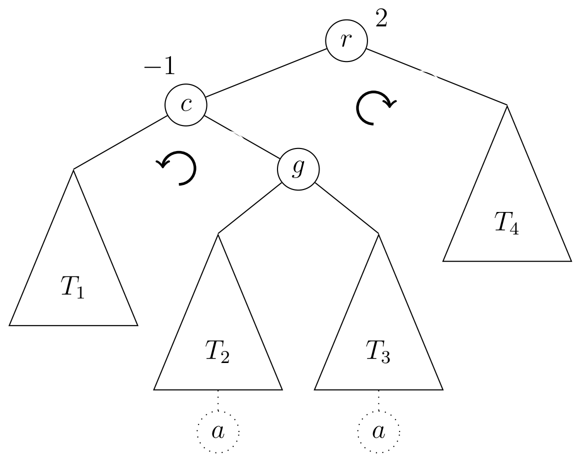


(a) Upon insertion of a new node a, the AVL Tree becomes unbalanced at r (which may be a subtree) with a balance factor of $+2$.

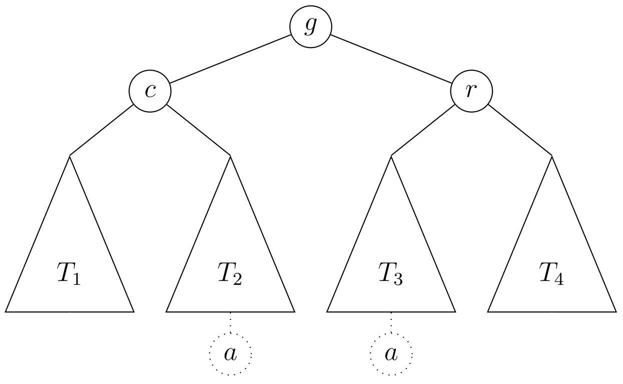


(b) AVL tree is rebalanced. The node c becomes the new root of the (sub)tree, r becomes c's right-child.

Figure 7.10: Generalized AVL R Rotation. Subtree T_2 "swings" over to become r's new left subtree.

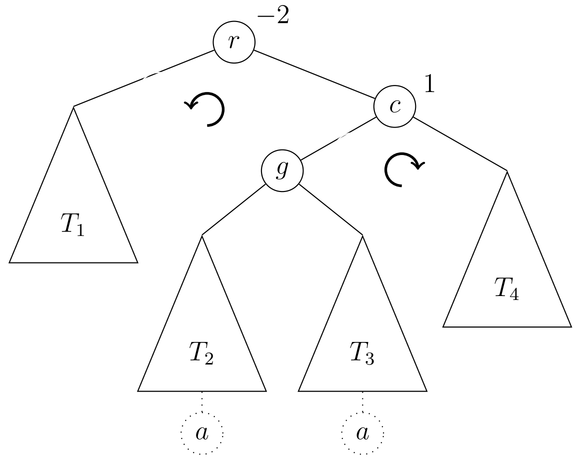


(a) Upon insertion of a new node a in either subtree, the AVL Tree becomes unbalanced at r (which may be a subtree) with a balance factor of $+2$. But the left-subtree rooted at c is skewed to the right.

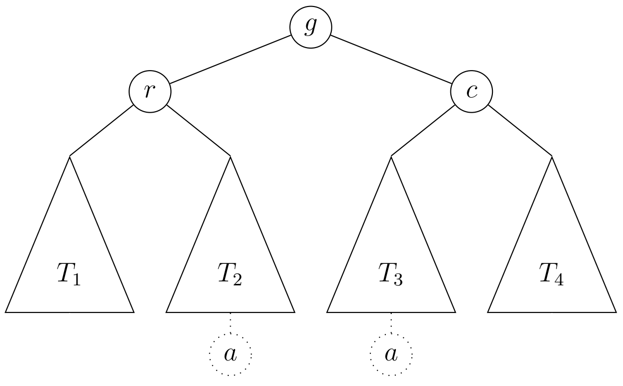


(b) AVL tree is rebalanced after a left rotation about c followed by a right rotation about r making g the new root. T_3 "swings" over to become the left-subtree of r. The node c becomes the new root of the (sub)tree, r becomes c's right-child.

Figure 7.11: Generalized AVL LR Rotation. Subtree T_3 "swings" over to become r's new left subtree.



(a) Upon insertion of a new node a in either subtree, the AVL Tree becomes unbalanced at r (which may be a subtree) with a balance factor of -2. But the right-subtree rooted at c is skewed to the left.



(b) AVL tree is rebalanced after a right rotation about c followed by a left rotation about r making g the new root. T_2 "swings" over to become the right-subtree of r. The node g becomes the new root of the (sub)tree, r becomes g's left-child.

Figure 7.12: Generalized AVL RL Rotation. Subtree T_2 "swings" over to become r's new right subtree.

Analysis

The height h of an AVL tree is bounded above and below:

$$\lfloor \log_2 n \rfloor \leq h \leq 1.4405 \log_2 (n+2) - 0.3277$$

Thus, no matter what the specific value of h is for a given tree, we can conclude that

$$h \in \Theta(\log n)$$

where n is the number of nodes in the tree. Furthermore, rotations are simply a matter of switching out pointers ($O(1)$) thus searching, insertion and deletion are all $O(h) = \Theta(\log n)$ for AVL Trees.

You can get some practice with AVL trees by trying out one of the many online simulations and animation applications. For example:

- `https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`
- `http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html`

7.5.2 B-Trees

- Every node has at most m children
- Every node (except the root) has at least $\lceil \frac{m}{2} \rceil$ children
- Root has at least 2 children unless it is a leaf
- Non-leaf nodes with k children contain $k-1$ keys
- All leaves appear in the same level and are non-empty
- Commonly used in databases
- Special case: 2-3 Trees ($m = 3$)
- 2-3 Trees credited to Hopcroft (1970) and B-Trees credited to Bayer & McCreight (1972) [5]

Rather than a single key per node, a 2-3 tree may have 1 or 2 keys per node.

- **2-node** – A node containing a single key k with two children, i.e. the usual type of node in a regular BST.
- **3-node** – A node containing two keys, k_1, k_2 such that $k_1 < k_2$. A 3-node has *three* children:
  - Left-Most-Child represents all keys less than k_1
  - Middle-Child represents all keys between k_1, k_2

  - Right-Most-Child represents all keys greater than k_2

Another requirement or property of a 2-3 Tree is that all of its leaves are on the same level. Every path from the root to any leaf will be have the same length. This ensures that the height h is uniform and balanced.

Basic Operations

- Searching is done straight forward, but slightly different than regular BSTs
- Insertion is always done *in a leaf*.
  - If the leaf is a 2-node, we insert by making it a 3-node and ordering the keys.
  - If the leaf is a 3-node, we split it up–the minimal key becomes the left-child, the maximal key becomes the right child and the middle key is promoted to the parent node

The promotion of a node to the parent can cause the parent to overflow (if the parent was a 3-node), and thus the process may continue upwards to the node's ancestors.

Note that if the promotion is at the root, a *new* node is created that becomes the new root.

Analysis

Let T be a 2-3 tree of height h. The smallest number of keys T could have is when all nodes are 2-nodes, which is essentially a regular binary search tree. As the tree is full, there would be at least

$$n \geq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

keys. Conversely, the largest number of keys would be when all nodes are 3-nodes and every node has 2 keys and three children. Thus,

$$n \leq 2 \sum_{i=0}^{h} 3^i = 2\left[3^{h+1} - 1\right]$$

Combining these two bounds and solving for h yields

$$\lceil \log_3 \left(\frac{n}{2} + 1\right) \rceil - 1 \leq h \leq \lceil \log_2 (n+1) \rceil - 1$$

Thus, $h \in \Theta(\log n)$.

As before, these bounds show that searching, insertion and deletion are all $\Theta(\log n)$ in the worst *and* average case.

Deleting: deleting from an internal node: exchange it with the maximal value in the left subtree (or minimal value in the right subtree) as is standard. Such a value will always be in a leaf.

Deletion may propagate to the root, merging siblings, etc.

7.5.3 Red-Black Trees

Definition 9. A *Red-Black Tree* is a binary search tree in which every node has a single (unique) key and a color, either red or black.

- The root is black and every leaf is black (this can be made trivial true by adding sentinels).
- Every red node has only black children.
- Every path from the root to a leaf has the same number of black nodes in it.

The black height of a node x is the number of black nodes in any path from x to a root (not counting x), denoted $bh(x)$.

Lemma 8. In a Red-Black Tree with n internal nodes satisfies the height property

$$h \leq 2 \log_2 (n+1)$$

You can prove this by induction on the height h with the base case being a leaf.

As with AVL Trees, insertion and deletion of nodes may violate the Red-Black Tree properties. Thus, we insert just like a regular BST, but we also perform rotations if necessary.

In fact, we use the *exact same* Left and Right rotations as are used in AVL Trees. However, LR and RL rotations are unnecessary.

The first step to inserting a new node is to insert it as a regular binary search tree and color it red.

There are three cases for insertion.

- Case 1 – If the inserted node z has a red parent and a red uncle (the cousin node of z's parent). In this case we recolor z's parent, uncle, and grandfather. We then recurse back up and see if any properties are violated at z's grandfather.
- Case 2 – z's parent is red but z's uncle is black. If z is the right-sub-child of its parent, then we perform a Left rotation. The new z becomes the new left-sub-child of z.
- Case 3 – z's parent is red but z's uncle is black. z is the left-sub-child of its parent, so we perform a Right rotation. The new z becomes the new left-sub-child of z.

- http://mathpost.la.asu.edu/~daniel/redblack.html
- http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html

7.6 Heaps

Definition 10. A *heap* is a binary tree that satisfies the following properties.

1. It is a *full* or *complete* binary tree: all nodes are present except possibly the last row
2. If the last row is not full, all nodes are full-to-the-left
3. It satisfies the *Heap Property*: every node has a key that is greater than both of its children (max-heap)

- As a consequence of the Heap Property, the maximal element is always at the root
- Alternatively, we can define a *min-heap*
- Variations: 2-3 heaps, fibonacci heaps, etc.
- A min-heap example can be found in Figure 7.18

Applications

- Heaps are an optimal implementation of a priority queue
- Used extensively in other algorithms (Heap Sort, Prim's, Dijkstra's, Huffman Coding, etc.) to ensure efficient operation

7.6.1 Operations

Insert

- Want to preserve the full-ness property and the Heap Property
- Preserve full-ness: add new element at the end of the heap (last row, first free spot on the left)
- Insertion at the end may violate the Heap Property
- Heapify/fix: bubble up inserted element until Heap Property is satisfied
- Analysis: insert is $O(1)$; heapify: $O(d)$

Remove from top

- Want to preserve the full-ness property and the Heap Property
- Preserve full-ness: swap root element with the last element in the heap (lowest row, right-most element)
- Heap property may be violated
- Heapify/fix: bubble new root element down until Heap Property is satisfied

- Analysis: Swap is $O(1)$; heapify: $O(d)$

Others

- Arbitrary remove
- Find
- Possible, but not ideal: Heaps are restricted-access data structures

Analysis

- All operations are $O(d)$
- Since Heaps are *full*, $d = O(\log n)$
- Thus, all operations are $O(\log n)$

7.6.2 Implementations

Array-Based

- Root is located at index 1
- If a node u is at index i, u's left-child is at $2i$, its right-child is at $2i + 1$
- If node u is at index j, its parent is at index $\lfloor \frac{j}{2} \rfloor$
- Alternatively: 0-index array left/right children/parent are at $2n + 1, 2n + 2$, and $\lfloor \frac{j-1}{2} \rfloor$
- Advantage: easy implementation, all items are contiguous in the array (in fact, a BFS ordering!)
- Disadvantage: Insert operation may force a reallocation, but this can be done in amortized-constant time (though may still have wasted space)

Tree-Based

- Reference-based tree (nodes which contain references to children/parent)
- Parent reference is now *required* for efficiency
- For efficiency, we need to keep track of the *last* element in the tree
- For deletes/inserts: we need a way to find the last element and first "open spot"
- We'll focus on finding the first available open spot as the same technique can be used to find the last element with minor modifications

Finding the first available open spot in a Tree-based Heap

Technique A: numerical technique

- WLOG: assume we keep track of the number of nodes in the heap, n and thus the depth $d = \lfloor \log n \rfloor$
- If $n = 2^{d+1} - 1$ then the tree is full, the last element is all the way to the right, the first available spot is all the way to the left
- Otherwise $n < 2^{d+1} - 1$ and the heap is not full (the first available spot is located at level d, root is at level 0)
- Starting at the root, we want to know if the last element is in the left-subtree or the right subtree
- Let $m = n - (2^d - 1)$, the number of nodes present in level d
- If $m \geq \frac{2^d}{2}$ then the left-sub tree is full at the last level and so the next open spot would be in the right-sub tree
- Otherwise if $m < \frac{2^d}{2}$ then the left-sub tree is not full at the last level and so the next open spot is in the left-sub tree
- Traverse down to the left or right respectively and repeat: the resulting sub-tree will have depth $d - 1$ with $m = m$ (if traversing left) or $m = m - \frac{2^d}{2}$ (if traversing right)
- Repeat until we've found the first available spot
- Analysis: in any case, its $O(d) = O(\log n)$ to traverse from the root to the first open spot

```
INPUT  : A tree-based heap $H$ with $n$ nodes
OUTPUT : The node whose child is the next available open spot in the heap
1  curr $\leftarrow T.head$
2  $d \leftarrow \lfloor \log n \rfloor$
3  $m \leftarrow n$
4  WHILE curr has both children DO
5      IF $m = 2^{d+1} - 1$ THEN
           //remaining tree is full, traverse all the way left
6          WHILE curr has both children DO
7              $curr \leftarrow curr.leftChild$
8          END
9      ELSE
           //remaining tree is not full, determine if the next open
             spot is in the left or right sub-tree
10         IF $m \geq \frac{2^d}{2}$ THEN
               //left sub-tree is full
11             $d \leftarrow (d - 1)$
12             $m \leftarrow (m - \frac{2^d}{2})$
13             $curr \leftarrow curr.rightChild$
14         ELSE
               //left sub-tree is not full
15             $d \leftarrow (d - 1)$
16             $m \leftarrow m$
17             $curr \leftarrow curr.leftChild$
18         END
19     END
20 END
21 output curr
```

Algorithm 40: Find Next Open Spot - Numerical Technique

Technique B: Walk Technique

- Alternatively, we can adapt the idea behind the tree walk algorithm to find the next available open spot
- We'll assume that we've kept track of the last node
- If the tree is full, we simply traverse all the way to the left and insert, $O(d)$
- If the last node is a left-child then its parent's right child is the next available spot,

finding it is $O(1)$

- Otherwise, we'll need to traverse around the perimeter of the tree until we reach the next open slot

```
INPUT  : A tree-based heap $H$ with $n$ nodes
OUTPUT : The node whose (missing) child is the next available open spot in the
         heap
$d \leftarrow \lfloor \log n \rfloor$
IF $n = 2^{d+1} - 1$ THEN
    //The tree is full, traverse all the way to the left
    $curr \leftarrow root$
    WHILE $curr.leftChild \neq null$ DO
        $curr \leftarrow curr.leftChild$
    END
ELSE IF $last$ is a left-child THEN
    //parent's right child is open
    $curr \leftarrow last.parent$
ELSE
    //The open spot lies in a subtree to the right of the last node
    //Walk the tree until we reach it
    $curr \leftarrow last.parent$
    WHILE $curr$ is a right-child DO
        $curr \leftarrow curr.parent$
    END
    //"turn" right
    $curr \leftarrow curr.parent$
    $curr \leftarrow curr.rightChild$
    //traverse all the way left
    WHILE $curr.leftChild \neq null$ DO
        $curr \leftarrow curr.leftChild$
    END
END
//current node's missing child is the open spot
output $curr$
```

Algorithm 41: Find Next Open Spot - Walk Technique

7.6.3 Java Collections Framework

Java has support for several data structures supported by underlying tree structures.

- `java.util.PriorityQueue<E>` is a binary-heap based priority queue
  - Priority (keys) based on either *natural ordering* or a provided `Comparator`
  - Guaranteed $O(\log n)$ time for insert (`offer`) and get top (`poll`)
  - Supports $O(n)$ arbitrary `remove(Object)` and search (`contains`) methods
- `java.util.TreeSet<E>`
  - Implements the `SortedSet` interface; makes use of a `Comparator`
  - Backed by `TreeMap`, a red-black tree balanced binary tree implementation
  - Guaranteed $O(\log n)$ time for add, remove, contains operations
  - Default iterator is an in-order traversal

7.6.4 Other Operations

include decrease key operation here

7.6.5 Variations

Binomial, Fibonacci, etc.

7.7 Applications

7.7.1 Heap Sort

- If min/max element is always at the top; simply insert all elements, then remove them all!
- Perfect illustration of “Smart data structures and dumb code are a lot better than the other way around”

```
Input   : A collection of elements $A = \{a_1, \ldots, a_n\}$
Output  : A collection, $A'$ of elements in $A$, sorted
1  $H \leftarrow$ empty heap
2  $A' \leftarrow$ empty collection
3  foreach $x \in A$ do
4      insert $x$ into $H$
5  end
6  while $H$ is not empty do
7      $y \leftarrow$ remove top from $H$
8      Add $y$ to the end of $A'$
9  end
10 output $A'$
```

Algorithm 42: Heap Sort

Analysis

- Amortized analysis: insert/remove operations are not constant throughout the algorithm
- On first iteration: insert is $d = O(1)$; on the i-th iteration, $d = O(\log i)$; only on the last iteration is insertion $O(\log n)$
- In total, the insert phase is:

$$\sum_{i=1}^{n} \log i = O(n \log n)$$

- A similar lower bound can be shown
- Same analysis applies to the remove phase:

$$\sum_{i=n}^{1} \log i$$

- In total, $O(n \log n)$

7.7.2 Huffman Coding

Overview

- Coding Theory is the study and theory of *codes*—schemes for transmitting data
- Coding theory involves efficiently padding out data with redundant information to increase reliability (detect or even correct errors) over a noisy channel

- Coding theory also involves *compressing* data to save space
  - MP3s (uses a form of Huffman coding, but is information lossy)
  - jpegs, mpegs, even DVDs
  - `pack` (straight Huffman coding)
  - `zip`, `gzip` (uses a Ziv-Lempel and Huffman compression algorithm)

Basics

- Let Σ be a fixed *alphabet* of size n
- A *coding* is a mapping of this alphabet to a collection of binary *codewords*,

$$\Sigma \rightarrow \{0, 1\}^*$$

- A *block encoding* is a *fixed length encoding* scheme where all codewords have the same length (example: ASCII); requires $\lceil \log_2 n \rceil$ length codes
- Not all symbols have the same frequency, alternative: *variable length encoding*
- Intuitively: assign shorter codewords to more frequent symbols, longer to less frequent symbols
- Reduction in the overall *average* codeword length
- Variable length encodings must be *unambiguous*
- Solution: *prefix free codes*: a code in which no *whole* codeword is the prefix of another (other than itself of course).
- Examples:
  - $\{0, 01, 101, 010\}$ is not a prefix free code.
  - $\{10, 010, 110, 0110\}$ is a prefix free code.
- A simple way of building a prefix free code is to associate codewords with the *leaves* of a binary tree (not necessarily full).
- Each edge corresponds to a bit, 0 if it is to the left sub-child and 1 to the right sub-child.
- Since no simple path from the root to any leaf can continue to another leaf, then we are guaranteed a prefix free coding.
- Using this idea along with a greedy encoding forms the basis of *Huffman Coding*

Steps

- Consider a precomputed relative frequency function:

$$\text{freq} : \Sigma \rightarrow [0, 1]$$

- Build a collection of *weighted* trees T_x for each symbol $x \in Sigma$ with $wt(T_x) = \text{freq}(x)$
- Combine the two least weighted trees via a new node; associate a new weight (the sum of the weights of the two subtrees)
- Keep combining until only one tree remains
- The tree constructed in Huffman's algorithm is known as a *Huffman Tree* and it defines a *Huffman Code*

```
INPUT  : An alphabet of symbols, $\Sigma$ with relative frequencies, freq($x$)
OUTPUT : A Huffman Tree
1  $H \leftarrow$ new min-heap
2  FOREACH $x \in \Sigma$ DO
3      $T_x \leftarrow$ single node tree
4      $wt(T_x) \leftarrow \text{freq}(x)$
5      insert $T_x$ into $H$
6  END
7  WHILE size of $H > 1$ DO
8      $T_r \leftarrow$ new tree root node
9      $T_a \leftarrow H.getMin$
10     $T_b \leftarrow H.getMin$
11     $T_r.leftChild \leftarrow T_a$
12     $T_r.rightChild \leftarrow T_b$
13     $wt(r) \leftarrow wt(T_a) + wt(T_b)$
14     insert $T_r$ into $H$
15 END
16 output $H.getMin$
```

Algorithm 43: Huffman Coding

Example

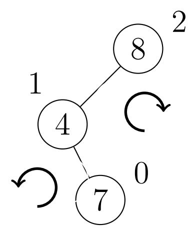
Construct the Huffman Tree and Huffman Code for a file with the following content.

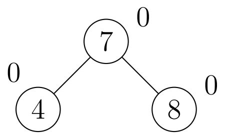
character	A	B	C	D	E	F	G
frequency	0.10	0.15	0.34	.05	.12	.21	.03

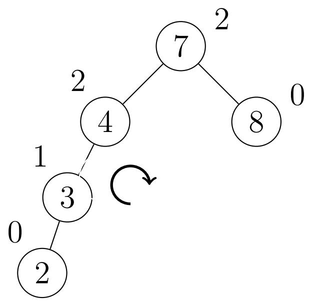
- Average codeword length:

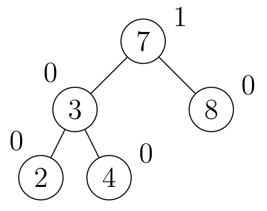
$$.10 \cdot 3 + .15 \cdot 3 + .34 \cdot 2 + .05 \cdot 4 + .12 \cdot 3 + .21 \cdot 2 + .03 \cdot 4 = 2.53$$

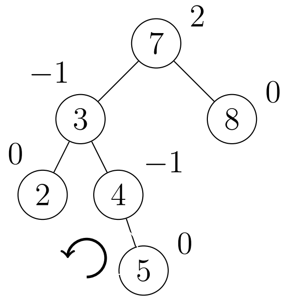
- Compression ratio:
$$\frac{(3-2.53)}{3} = 15.67\%$$
- In general, for text files, `pack` (Huffman Coding), claims an average compression ratio of 25-40%.
- Degenerative cases:
  - When the probability distribution is uniform: $p(x) = p(y)$ for all $x, y \in \Sigma$
  - When the probability distribution follows a fibonacci sequence (the sum of each of the two smallest probabilities is less than equal to the next highest probability for all probabilities)

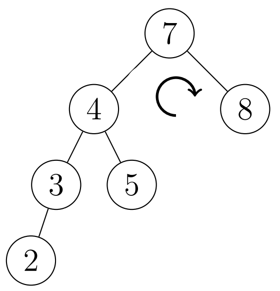

(a) Insertion of $8, 4, 7$ causes the tree to become unbalanced.

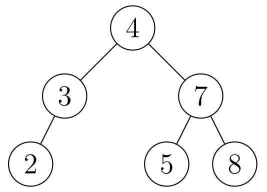

(b) AVL tree is rebalanced after an LR Rotation

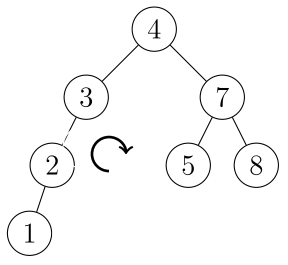

(c) Insertion of $3, 2$ unbalances the tree at node 4

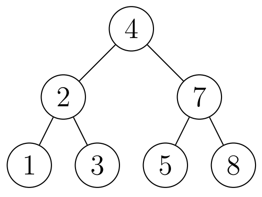

(d) AVL tree is rebalanced after an R Rotation


(e) Insertion of 5 unbalances the tree at 7, an LR rotation will be performed at node 7


(f) Step 1: A left rotation at node 3


(g) Step 2: A right rotation at node 7; node 5 swings over to become 7's left-child.


(h) Insertion of 1 unbalances the tree at 3, a right rotation about 3 is performed.


(i) The right rotation results in a complete binary tree. The balance factor of all nodes is zero.

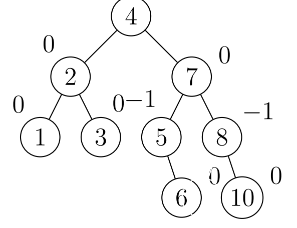
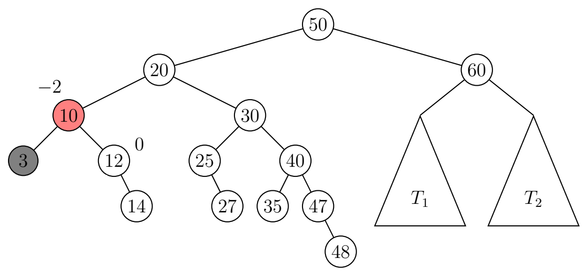
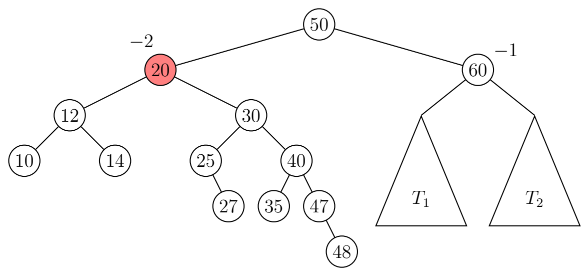

(j) Insertion of $10, 6$ does not unbalance the AVL tree.

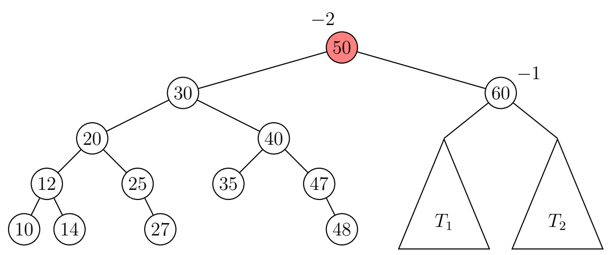
Figure 7.13: AVL Tree Insertion Sequence. Insertion of elements $8, 4, 7, 3, 2, 5, 1, 10, 6$.



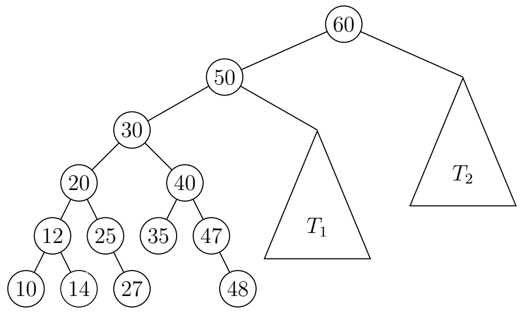
(a) Deletion of 3 causes node 10 to become unbalanced, prompting a left rotation about 10.



(b) The rotation reduces the height of the subtree at node 12 causing an unbalance at 20. A left rotation is performed about 20; the subtree rooted at 25 swings over to become the right subtree of 20.

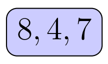


(c) The rotation reduces the height of the subtree at node 30 causing an unbalance at the root, 50. A left rotation is performed about 50; the subtree T_1 swings over to become the right subtree of 50.

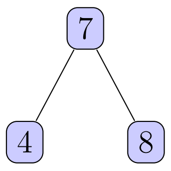


(d) The rotation at the root rebalances the tree.

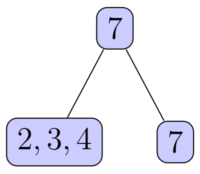
Figure 7.14: Worst-case Example of Rebalancing Following a Deletion. Deletion of 3 causes an imbalance at its parent. Each subsequent rotation correction induces another imbalance to the parent all the way up to the root. Deletion can potentially result in $\Theta(\log n)$ correction rotations.



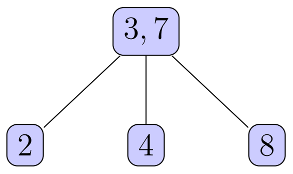
(a) Insertion of $8, 4, 7$ causes the root node to become overfull



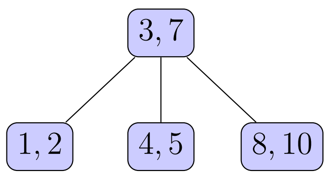
(b) The tree splits into two children and creates a new root node.



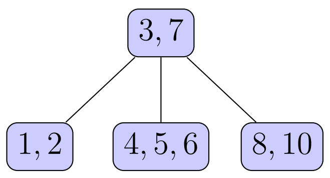
(c) Insertion of $3, 2$ causes the left node to become overfull



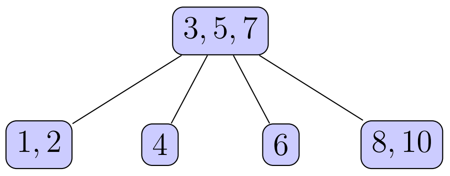
(d) The middle key 3 is promoted to the parent (root) node while $2, 4$ get split into 2 subtrees



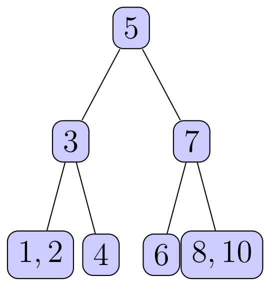
(e) Insertion of $5, 1, 10$ does not cause any disruption, but all nodes are full at this point.



(f) Insertion of 6 causes the middle node to become overfull

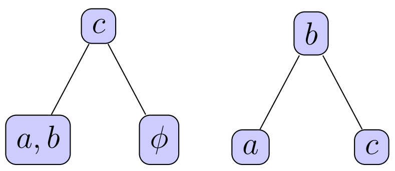


(g) 5 is promoted while 2 children are created.

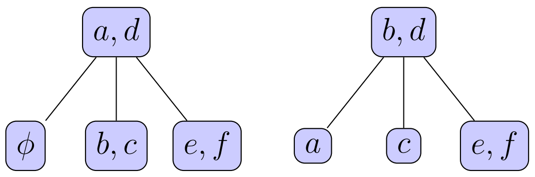


(h) However, the promotion of 5 caueses the root to become overfull, the split propagates up the tree.

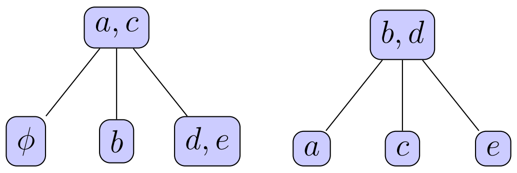
Figure 7.15: 2-3 Tree Insertion Sequence. Insertion of elements $8, 4, 7, 3, 2, 5, 1, 10, 6$.



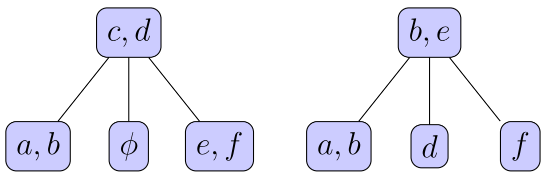
(a) The left sibling node can spare a key, b, c are rotated.



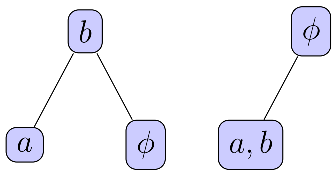
(b) The right sibling can spare a key; b, a are rotated.



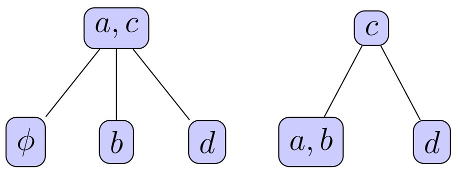
(c) The right sibling does not have a key to spare, but the far right one does, a rotation involves all three children.



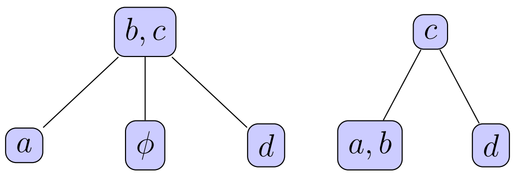
(d) The immediate right sibling has a key to spare, e is promoted, d fulfills the empty child.



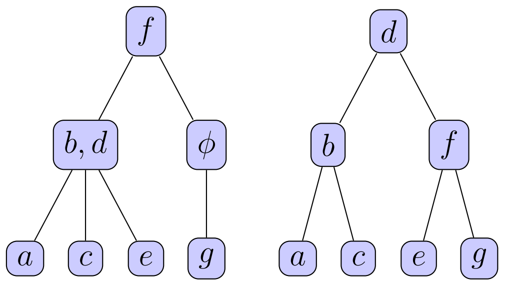
(e) The left sibling does not have a key to spare; instead the parent is *merged* down. The resulting parent is now empty and has only one child. We deal with it recursively.



(f) The right sibling does not have a key to spare; instead one of the parent keys is merged down, reducing the number of children nodes.

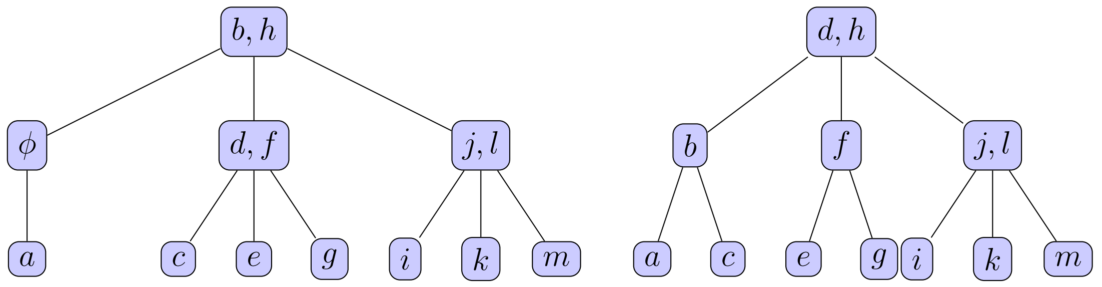


(g) Neither sibling has a key to spare, so the two left children are merged, b is demoted.

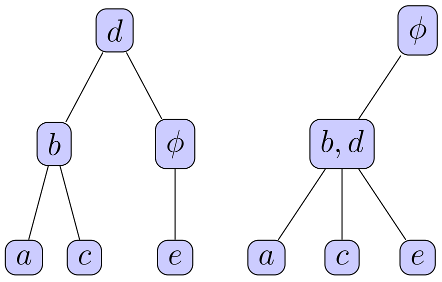


(h) A generalized rotation, b is promoted, f is demoted; e swings over to become f's new left child.

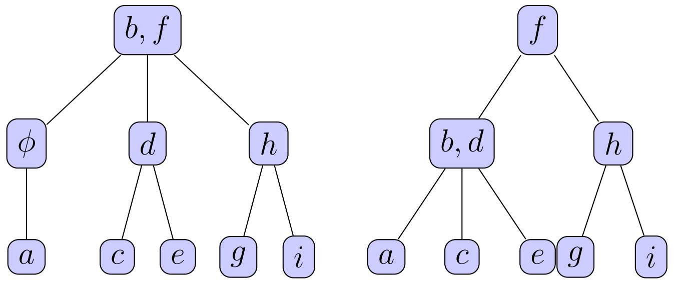
Figure 7.16: 2-3 Tree Deletion Operations Part I. Different configurations result in redistributions or merging of nodes. Symmetric operations are omitted.



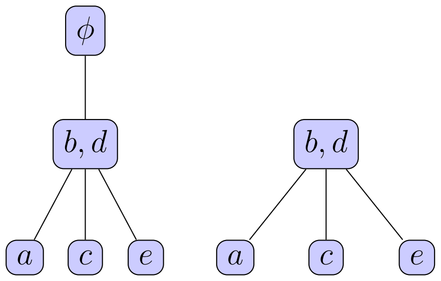
(a) Another generalized rotation. b is demoted, d promoted while c swings over to be b's right child.



(b) A generalized merge procedure: d is merged down; e swings over to become the right-most child.



(c) Another generalized merge procedure; b is merged with d; a becomes their left-most child.



(d) Deleting an empty root. The root's child becomes the new root.

Figure 7.17: 2-3 Tree Deletion Operations Part II. Different configurations result in redistributions or merging of nodes. Symmetric operations are omitted.

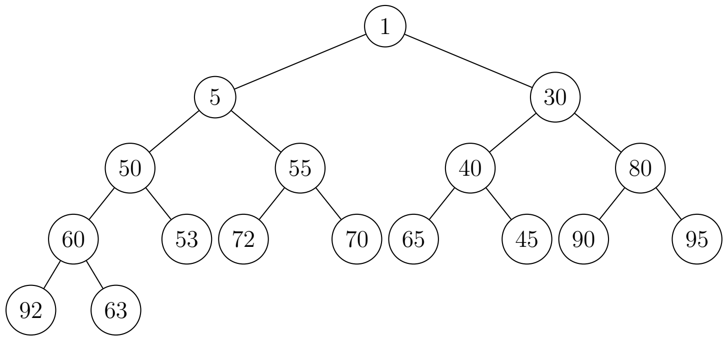


Figure 7.18: A min-heap

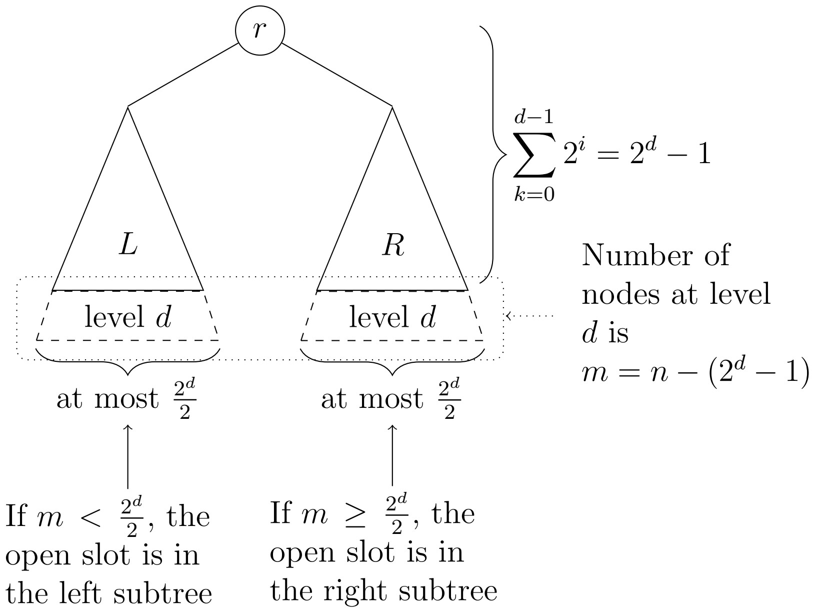

r
L
R
$\sum_{k=0}^{d-1} 2^i = 2^d - 1$
level d
level d
Number of nodes at level d is $m = n - (2^d - 1)$
at most $\frac{2^d}{2}$
at most $\frac{2^d}{2}$
If $m < \frac{2^d}{2}$, the open slot is in the left subtree
If $m \geq \frac{2^d}{2}$, the open slot is in the right subtree

Figure 7.19: Tree-based Heap Analysis. Because of the fullness property, we can determine which subtree (left or right) the "open" spot in a heap's tree is by keeping track of the number of nodes, n. This can be inductively extended to each subtree until the open spot is found.

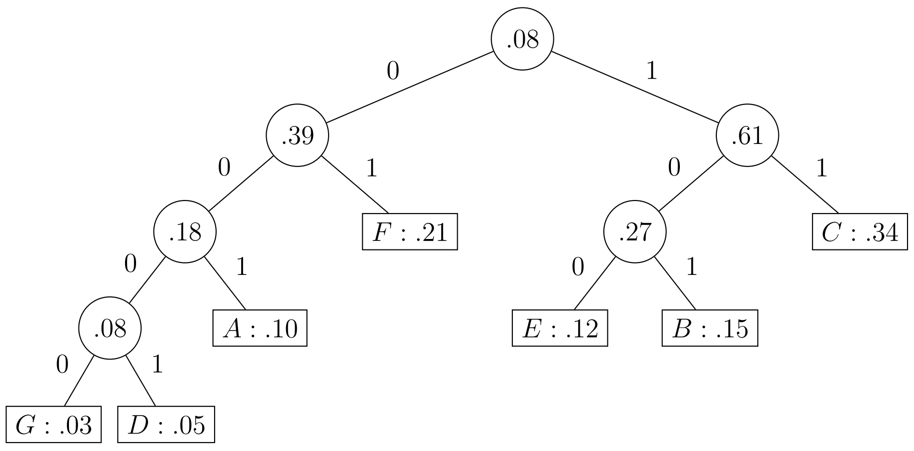


Figure 7.20: Huffman Tree

8 Graph Algorithms

8.1 Introduction

Definition 11. An undirected graph $G = (V, E)$ is a 2-tuple with

- $V = \{v_1, \ldots, v_n\}$ a set of vertices.
- $E = \{e_1, \ldots, e_m\}$ a set of edges where $e_i = (u, v)$ for $u, v \in V$

Graph variations:

- Directed: each edge is *oriented* and (u, v) is an *ordered pair*
- Weighted: there is a weight function defined on the edge set. It usually suffices to consider:
$$wt : E \to \mathbb{Z}$$

Representations:

- **Adjacency List** – An adjacency list representation of a graph $G = (V, E)$ maintains $|V|$ linked lists. For each vertex $v \in V$, the head of the list is v and subsequent entries correspond to adjacent vertices $v' \in V$.

  - Advantage: less storage
  - Disadvantage: adjacency look up is $\mathcal{O}(|V|)$, extra work to maintain vertex ordering (lexicographic)
- **Adjacency Matrix** – An adjacency matrix representation maintains an $n \times n$ sized matrix such that
$$A[i, j] = \begin{cases} 0 & \text{if } (v_i, v_j) \notin E \\ 1 & \text{if } (v_i, v_j) \in E \end{cases}$$
for $0 \leq i, j \leq (n - 1)$
  - Advantages: adjacency/Weight look up is constant
  - Disadvantage: extra storage
- In either case, weights can be stored as extra data in nodes or as entries in the matrix

Libraries:

- JGraphT (Java) – `http://jgrapht.org/`
- Boost (C++ header only library) – `http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/index.html`
- LEMON (C++ Graph library) – `http://lemon.cs.elte.hu/trac/lemon`
- Python: NetworkX (`http://networkx.github.io/`), igraph (`http://igraph.org/`)
- Ruby: `https://github.com/bruce/graphy`
- JavaScript: `https://github.com/devenbhooshan/graph.js`

8.2 Depth First Search

Depth First Search (DFS) traverses a graph by visiting "deeper" vertices *first*, before processing the vertices. That is, it explores a graph as deeply as possible before back tracking and visiting other vertices.

At each iteration of DFS we can choose to visit an unvisited vertex according to several criteria:

- Lexicographic order – visit the next vertex according to the label ordering
- Weight – in a weighted graph, we can visit the vertex whose edge is the least (or greatest) weight

In general, we can keep track of the state of nodes by two measures, a vertex's color and the discovery/processing "time".

At various points in the algorithm, a vertex is colored:

- White – indicating that the vertex is unvisited (all vertices are initially white)
- Gray – indicating that the vertex has been discovered, but not yet processed (we go deeper in the graph if possible before processing gray vertices)
- Black – indicating that the vertex has been visited and processed. By the end of the algorithm, all vertices are black

Discovery and processing time stamps can be associated with each vertex by keeping a global counter that increments each time a vertex is initially visited and again when it is processed. The sequence of time stamps indicates visit/process order.

A straightforward implementation would be a recursive algorithm (Algorithms 44 and 45). The main algorithm is necessary to accommodate:

- Directed graphs - we may need to restart the search if some vertices are unreachable from the initial vertex

- Disconnected graphs – we may need to restart at another connected component

```
INPUT   : A graph, $G = (V, E)$
1 FOREACH *vertex* $v \in V$ DO
2     color $v$ white (unvisited)
3 END
4 *count* $\leftarrow 0$
5 FOREACH $v \in V$ DO
6     IF $v$ *is white* THEN
7         DFS$(G, v)$
8     END
9 END
```

Algorithm 44: Recursive Depth First Search, Main Algorithm DFS(G)

```
INPUT   : A graph, $G = (V, E)$, a vertex $v \in V$
1  *count* $\leftarrow$ *count* $+ 1$
2  mark $v$ with count (discovery time)
3  color $v$ gray (discovered, not processed)
4  FOREACH $w \in N(v)$ DO
5      IF $w$ *is white* THEN
6          DFS$(G, w)$
7      END
8  END
9  *count* $\leftarrow$ *count* $+ 1$
10 mark $v$ with count (processing time)
11 process $v$
12 color $v$ black
```

Algorithm 45: Recursive Depth First Search, Subroutine DFS(G)

A better implementation would be to use a smarter data structure, namely a stack

(Algorithm 46).

```
INPUT   : A graph, $G = (V, E)$
FOREACH vertex $v \in V$ DO
    color $v$ white (unvisited)
END
$count \leftarrow 1$
$S \leftarrow$ empty stack
push start vertex $v$ onto $S$
mark $v$ with $count$ (discovery time)
color $v$ gray
WHILE $S$ is not empty DO
    $count \leftarrow count + 1$
    $x \leftarrow S.peek$
    $y \leftarrow$ next white vertex in $N(x)$
    //If there is a gray vertex in $N(y)$ not equal to $x$, it
      constitutes a cycle
    IF $y$ is nil THEN
        //Neighborhood of $x$ has been exhausted, start backtracking
        $S.pop$ ($x$ is popped off of $S$)
        process $x$
        color $x$ black
        mark $x$ with count (processing time)
    END
    ELSE
        $S.push \quad y$
        color $y$ gray
        mark $y$ with count (discovery time)
    END
END
```

Algorithm 46: Stack-Based Depth First Search

8.2.1 DFS Example

Consider the small graph in Figure 8.1; the discovery and processing (finishing) time stamps are presented in Table 8.1.

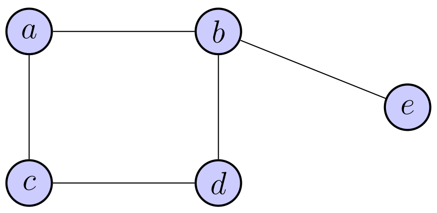


Figure 8.1: A small graph.

Vertex	discovery time	process time
a	1	10
b	2	9
c	4	5
d	3	6
e	7	8

Table 8.1: Depth First Search Time Stamps

8.2.2 DFS Artifacts

Depth-First Search can produce a *depth-first forest* (or tree if the graph is connected). Edges in a DFS forest can be classified as:

- **Tree Edges** – Edges that are part of the search path, whenever an unvisited vertex v' is discovered from the current vertex, v, (v, v') is a tree edge.
- **Back Edge** – From the current vertex, v, if an edge is found pointing to a vertex that has already been discovered, the edge (v, v') is a back edge. Back edges connect vertices to their ancestors in the DFS forest
- **Forward Edge** – If, after backtracking, the current vertex v points to a visited vertex v', (v, v') is a forward edge.
- **Cross Edge** – All other edges, they can be edges between trees in the forest or edges between vertices in the same tree with a common ancestor

Note that:

- For *undirected* graphs forward and back edges are the same (no orientation)
- For *undirected* graphs, cross edges do not exist
- For *directed* graphs, cross edges may connect between components (created by restarting the DFS subroutine) or within components

Edges can be classified as follows. From the current vertex, the color of the adjacent vertex determines the type of edge:

- White indicates a tree edge

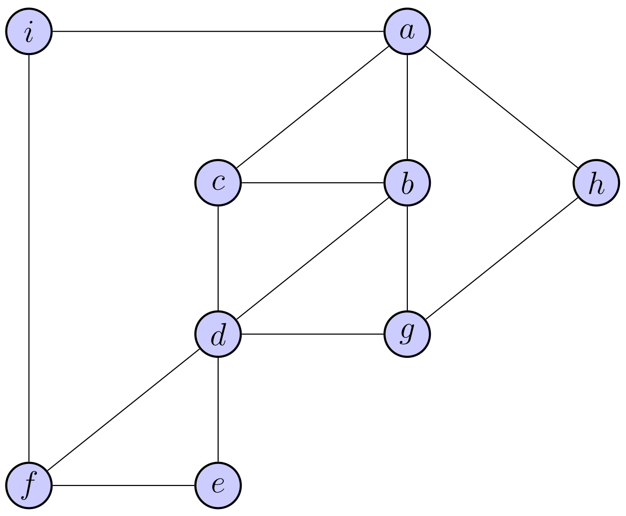


Figure 8.2: A larger graph.

- Gray indicates a back edge
- Black indicates a forward or cross edge (extra work is required to determine if the vertex is in the same component or not)

Consider the larger graph in Figure 8.2. A DFS performed starting at vertex i would lead to the following.

Visitation order (using lexicographic next choice):

$$i, a, b, c, d, e, f, g, h$$

Processing order:

$$f, e, h, g, d, c, b, a, i$$

The DFS Tree produced by this traversal can be found in Figure 8.3. Note that only forward edges are present as this is a connected, undirected graph.

8.2.3 Analysis

- Each vertex is examined once when it is first visited (pushed) and when it is finished (popped)
- Each vertex is processed exactly once
- Each edge is examined once (twice for undirected graphs): each time it appears in a neighborhood
- For adjacency matrices, this is $O(n^2)$ as each traversal will examine every entry in the matrix (to to determine the neighborhood of each vertex)

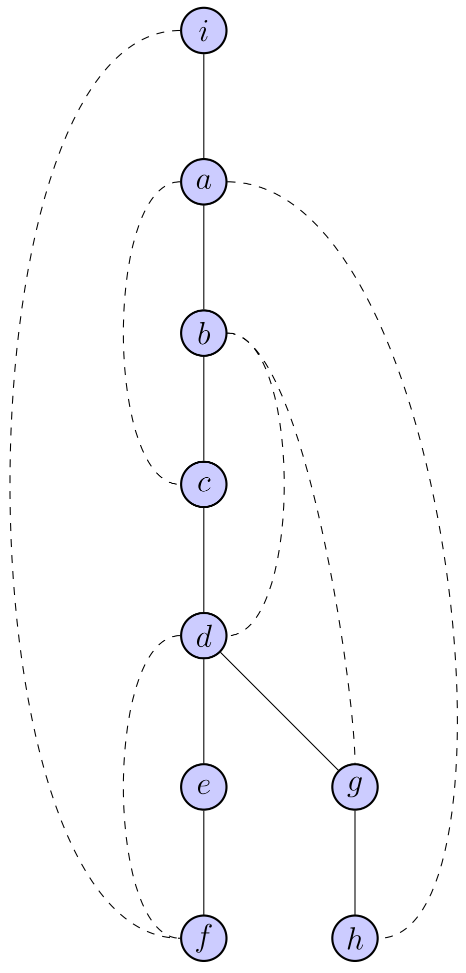


Figure 8.3: DFS Forrest with initial vertex i. Dashed edges indicate back edges.

- For adjacency matrices this is $O(n + m)$: for each vertex, its entire adjacency list will have to be examined, but need not be computed

8.3 Breadth First Search

Breadth First Search (BFS) is a "shallow search." BFS explores the nearest vertices first. At any given vertex v, the entire neighborhood, $N(v)$ is explored before progressing further in the graph.

As with DFS, a BFS may need to be restarted (disconnected graphs or unreachable vertices in a directed graph), see Algorithms 47 and 48.

Note the contrast: BFS uses a *queue* data structure.

```
Input   : A graph, $G = (V, E)$
foreach vertex $v \in V$ do
    color $v$ white (unvisited)
end
$count \leftarrow 0$
foreach $v \in V$ do
    if $v$ is white then
        bfs$(G, v)$
    end
end
```

Algorithm 47: Breadth First Search, Main Algorithm Bfs(G)

```
Input   : A graph, $G = (V, E)$, an initial vertex $v \in V$
$count \leftarrow count + 1$
$Q \leftarrow$ empty queue
mark $v$ with count (discovery time)
color $v$ gray (discovered, not processed)
$Q.enqueue(v)$
while $Q$ is not empty do
    $x \leftarrow Q.peek$
    foreach $y \in N(x)$ do
        if $y$ is white then
            $count \leftarrow count + 1$
            mark $y$ with count (discovery time)
            $Q.enqueue(y)$
        end
    end
    $z \leftarrow Q.dequeue$
    $count \leftarrow count + 1$
    mark $z$ with count (processing time)
    process $z$
    color $z$ black
end
```

Algorithm 48: Breadth First Search, Subroutine bfs(G, v)

Vertex	discovery time	process time
a	1	2
b	3	5
c	4	8
d	6	9
e	7	10

Table 8.2: Breadth First Search Time Stamps

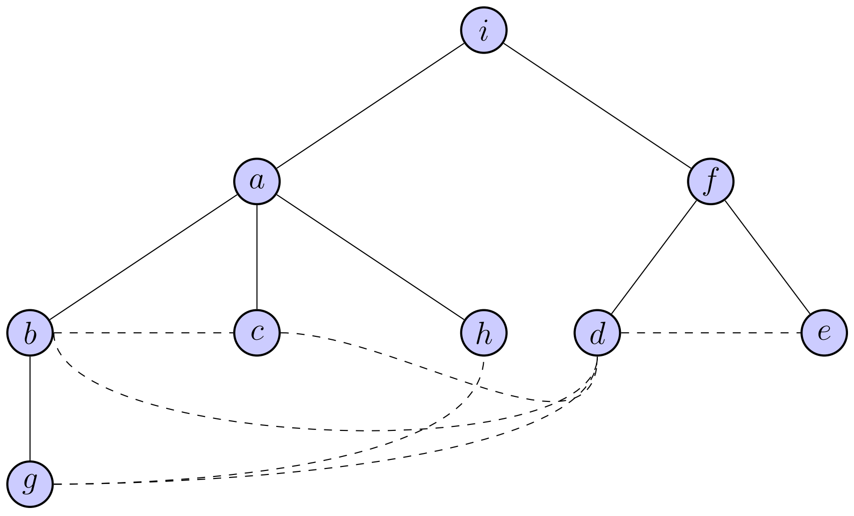


Figure 8.4: BFS Tree with initial vertex i. Dashed edges indicate cross edges.

An example BFS run on the same small example can be found in Table 8.2.

BFS can also produce a BFS Forest (or tree) with similar types of edges (Tree, Cross, Forward, Back). The BFS Tree of the graph in Figure 8.2 can be seen in Figure 8.4.

- For undirected graphs only tree and cross edges are possible (no back/forward edges: why?)
- For directed graphs, back and cross edges are possible; forward edges are not

8.4 DFS/BFS Applications

Some problems can be solved by either DFS or BFS, others are more appropriately solved by one more than the other.

8.4.1 Connectivity & Path Finding

Problem 9 (Connectivity)**.**
Given: A graph $G = (V, E)$, two vertices $s, t \in V$
Output: true if there is a path $p : s \rightsquigarrow t$

Variations:

- Directed or undirected paths
- A *functional* version: not only do we want to know if a path exists, but if one does, we want the algorithm to output one
- Does there exist a path involving a particular vertex (or edge)?

8.4.2 Topological Sorting

Recall that a partial ordering is a relation on a set that is reflexive, antisymmetric and transitive. Posets can model:

- Tasks to be performed with certain prerequisites
- Entities with constrained relations, etc.

In general, consider a directed acyclic graph (DAG). A *topological ordering* (or sorting) is an arrangement of vertices that is consistent with the connectivity defined by the underlying graph.

That is, if $(u, v) \in E$ then u precedes v in the topological ordering.

There may be many valid topological orders

DFS Solution: Run DFS on the DAG and ordering the vertices in descending order according to their finishing timestamps.

8.4.3 Shortest Path

The length of a path $p : s \rightsquigarrow t$ is equal to the number of edges in p.

Problem 10 (Shortest Path)**.**
Given: A graph $G = (V, E)$, two vertices $s, t \in V$
Output: The shortest length path $p : s \rightsquigarrow t$

Variations:

- Directed or undirected versions
- Output the shortest *weighted* path p

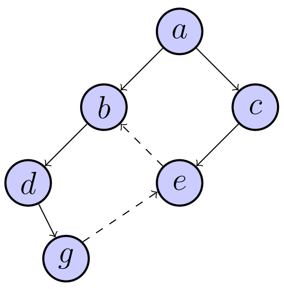


Figure 8.5: BFS Tree with cross edges (dashed) involved in a cycle.

- Output the shortest path $s \rightsquigarrow v$ for *all* vertices v
- Output the shortest path for all pairs of vertices

Observation: for directed graphs, the shortest path $s \rightsquigarrow t$ is not necessarily the shortest (or even a valid) path for $t \rightsquigarrow s$.

BFS Solution: For unweighted graphs (directed or undirected), the BFS tree provides all shortest paths $s \to v$ for all v where s is the start vertex of the BFS.

Observation: contrast this "easy" problem with the opposite problem of finding the *longest* path in a graph. Finding the longest path is in fact equivalent to the Hamiltonian Path problem, a much more difficult, NP-complete problem.

8.4.4 Cycle Detection

Problem 11 (Cycle Detection)**.**
Given: A graph $G = (V, E)$
Output: true if G contains a cycle, false otherwise

For undirected graphs:

- If DFS encounters a back edge, then a cycle exists
- If BFS encounters a cross edge, then a cycle exists

Directed graphs are more complex:

- If DFS or BFS encounters a back edge, then a cycle exists
- A single cross edge or forward edge does not necessarily imply a cycle (see Figure 8.5)
- A cycle may consist of a series or cross, tree, and forward edges
- A lot more work may be required to distinguish true cycles.

Alternatives: make two runs of DFS or BFS to answer the connectivity question:

$$\exists u, v \in V \exists \left[p_1 : u \rightsquigarrow v \wedge p_2 : v \rightsquigarrow u\right]$$

For undirected graphs, this does not necessarily work: the path could be the same. Instead we could take the following strategy: use a functional version of DFS/BFS to actually find a path p from u to v, then remove each of the vertices (and corresponding edges) in the path (except for u, v) and run DFS/BFS again to see if there is a *different* path p' which would constitute a cycle.

8.4.5 Bipartite Testing

Recall that a *bipartite graph* $G = (R, L, E)$ is a graph with two disjoint vertex sets, R and L and edges *between* vertices in R and L. That is, for $v, v' \in R$, (v, v') is never an edge (L likewise).

Problem 12 (Bipartite Testing)**.**
Given: A graph $G = (V, E)$
Output: true if G is a bipartite graph, false otherwise

Theorem 6. An undirected graph $G = (V, E)$ is bipartite if and only if G contains no odd-length cycles.

We already have an algorithm for cycle detection, how can we modify it to detect *odd-cycles*?

8.4.6 Condensation Graphs

A directed graph $G = (V, E)$ is called *strongly connected* if for any pair of vertices, $u, v \in V$ there exists a directed path $p : u \rightsquigarrow v$. That is, every vertex is *reachable* from every other vertex by some path.

More generally, a directed graph's vertices can be partitioned into maximal, disjoint subsets, $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k$ such that each induced subgraph $G_i = (\mathcal{V}_i, E_i)$ is a strongly connected graph. These are known as the *strongly connected components* of G.

A *condensation graph* $G_c = (V', E')$ of a directed graph G is defined as

- $V' = \{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$
- $E' = \{(\mathcal{V}_i, \mathcal{V}_j \mid \text{for some } u \in \mathcal{V}_i, v \in \mathcal{V}_j, (u, v) \in E\}$

That is, the vertices are the strongly connected components of G and there are edges between two components if some vertex is reachable between them in G.

A condensation graph is always a directed, acyclic graph that is *never* strongly connected (unless G was strongly connected–what would G_c look like then?). The following is the Kosaraju-Sharir Algorithm.

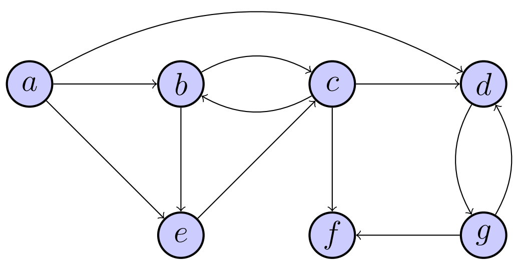


Figure 8.6: A Directed Graph.

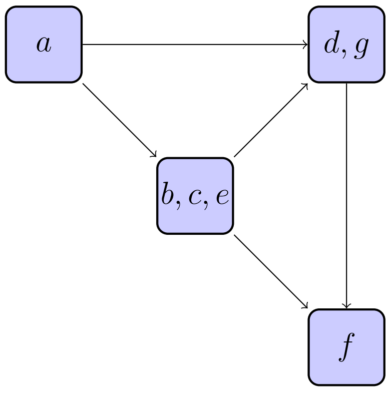


Figure 8.7: Condensation Graph

1. Do a Depth-First-Search traversal of G and note the *finish* time stamps
2. Compute the transpose graph G^T (all edges are reversed)
3. Do a Depth-First-Search on G^T in descending order of the ending time stamps in step 1.
    - Each time you start over is a new strongly connected component
    - Edges in G_c can be created on the fly–there will be at most one out edge and at most one in edge for each vertex in G_c.
4. Output G_c

8.5 Minimum Spanning Tree Algorithms

8.5.1 Greedy Algorithmic Strategy

The *greedy* algorithmic design approach involves iteratively building a solution to a problem using the next immediately available best choice.

As we iteratively build up a solution to a problem, each *greedy* choice we make has to satisfy a few properties:

- It must be *feasible* – it has to satisfy the constraints of the problem (like a total weight constraint)
- It must be *locally optimal* – it should be the best, immediately available choice we have.
- It must be *irrevocable* – once a choice has been made, we are not allowed to back track and undo it.

Greedy solutions are efficient, but make decisions based only on local information which does not necessarily guarantee a *globally optimal* solution. Often greedy algorithms can provide an *approximations.*

One such example is Kruskal's algorithm for the minimum spanning tree problem.

In a network topology there may be many redundant connections. We wish to identify the "backbone" of the network. That is, we want to find a subgraph with the minimal number of edges of minimal weight that keep the network connected.

More generally, given a weighted graph, we wish to find a subgraph that keeps the graph connected, namely a tree.

Definition 12. Let $G = (V, E)$ be a connected, weighted graph. A *spanning tree* of G is a subgraph, $T = (V, E')$ where $E' \subseteq E$ is a set of edges that *span* the vertex set V without inducing a cycle. A *minimum spanning tree* is a spanning tree of minimal weight. That is,

$$\sum_{e \in E'} wt(e)$$

is minimal.

Problem 13 (Minimum Spanning Tree)**.**
Given: A weighted graph $G = (V, E)$
Output: A minimum spanning tree T of G

In general, there may be *many* unique minimum spanning trees. In fact, the number of possible spanning trees is exponential and generating them is very difficult. Rather, solving the minimum spanning tree problem can be solved *exactly* by a greedy strategy.

8.5.2 Kruskal's Algorithm

Kruskal's algorithm takes a greedy approach

- presorts edges in nondecreasing order according to their weights
- it considers each edge in this order

Edge	Weight	Result
(a, d)	5	include
(c, e)	5	include
(d, f)	6	include
(a, b)	7	include
(b, e)	7	include
(b, c)	8	exclude
(e, f)	8	exclude
(b, d)	9	exclude
(e, g)	9	include

Table 8.3: Sequence of edges considered by Kruskal's Algorithm and results.

- if the inclusion of the edge would induce a cycle in the tree created so far, it is ignored, otherwise it takes it
- the algorithm terminates when it has taken $n - 1$ edges

```
   INPUT  : A weighted graph, $G = (V, E)$
   OUTPUT : A minimum spanning tree of $G$
 1 sort edges in nondecreasing order with respect to their weights
 2 $E_T \leftarrow \emptyset$
 3 $k \leftarrow 1$
 4 WHILE $|E_T| < (n - 1)$ DO
 5     IF $(V, E_T \cup \{e_k\})$ is acyclic THEN
 6         $E_T \leftarrow E_T \cup \{e_k\}$
 7     END
 8     $k \leftarrow (k + 1)$
 9 END
10 output $(V, E_T)$
```

Algorithm 49: Kruskal's Minimum Spanning Tree Algorithm

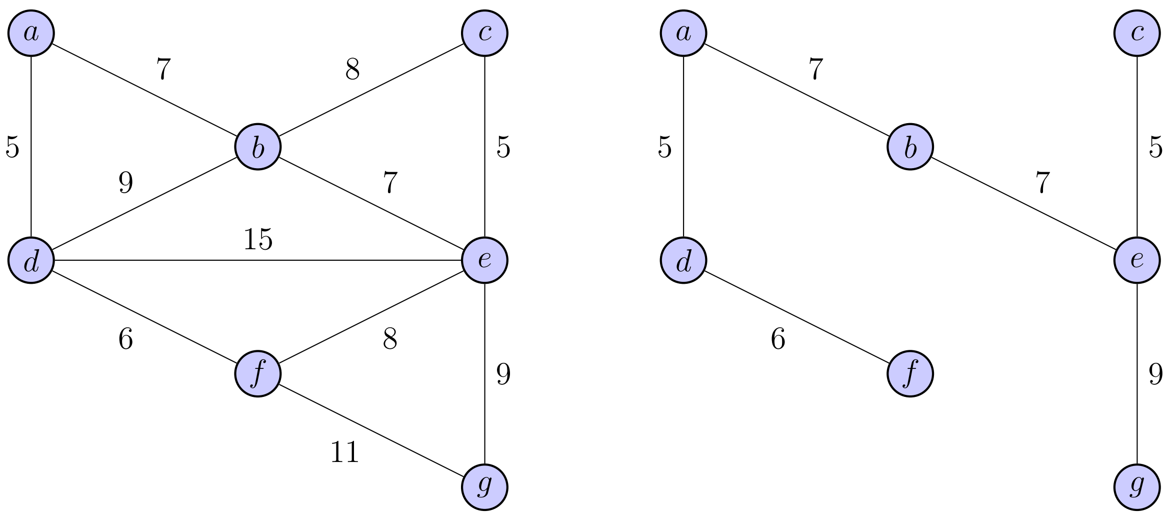
An example run of Kruskal's algorithm on the weighted graph in Figure 8.8(a) of Kruskal's can be found in Table 8.3. The resulting graph can be seen in Figure 8.8(b).

The presorting is simply $\Theta(m \log m)$ with any good sorting algorithm.

The acyclicity check can be made using either DFS or BFS. If both of the edge's endpoints are in the same connected component created thus far, reject it.

In the worst case, the while loop will run for $\Theta(m)$ iterations. Each intermediate collection of (disjoint) trees will be $\Theta(n)$ for BFS/DFS (why?) so in all,

$$\Theta(m \log m) + \Theta(m) \cdot \Theta(n) \in O(n^3)$$



(a) A weighted, undirected graph.

(b) Minimum Spanning Tree with total weight 39.

Figure 8.8: Minimum Spanning Tree example. For this particular example, Kruskal's and Prim's algorithms result in the same tree. In fact, for this example there is only one unique MST.

which can be improved to $\Theta(m \log m)$ using an appropriate data structure (that removes the need to perform a DFS/BFS).

8.5.3 Prim's Algorithm

Prim's algorithm is also greedy but works differently. It may result in a different, but equivalent minimum spanning tree.

Starting at an arbitrary vertex, it builds subtrees by adding a single vertex on each iteration. The vertex it chooses is based on a greedy choice: add the vertex whose edge connects the current subtree with the minimum weight.

```
   INPUT  : A weighted graph, $G = (V, E)$
   OUTPUT : A minimum spanning tree of $G$
 1 $E_T \leftarrow \emptyset$
 2 $V_T \leftarrow \{v_1\}$
 3 $E^* \leftarrow N(v_1)$
 4 FOR $i = 1, \ldots, n - 1$ DO
 5     $e \leftarrow$ minimum weighted edge in $E^*$
 6     $u \leftarrow$ the endpoint in $e$ that is contained in $V_T$
 7     $v \leftarrow$ the end point in $e$ not contained in $V_T$
 8     $V_T \leftarrow V_T \cup \{v\}$
 9     $E_T \leftarrow E_T \cup \{e\}$
10     $E^* \leftarrow (E^* \cup N(v)) \setminus \{e = (x, y) | x \in V_T \wedge y \in V_T\}$
11 END
12 output $(V_T, E_T)$
```

Algorithm 50: Prim's Minimum Spanning Tree Algorithm

During the execution of the algorithm, vertices can be partitioned into one of the following sets:

- Tree vertices – vertices that have already been added to the tree
- Fringe vertices – vertices in the neighborhood of the set of tree vertices
- Unseen vertices – all other vertices that would not affect the next-vertex greedy choice

On each iteration, we move the *closest* fringe vertex v to the set of tree vertices, add the relevant edge and update the fringe vertex set by adding the neighborhood of v, $N(v)$.

An example run on of Prim's algorithm on the weighted graph in Figure 8.8(a) can be found in Table 8.4. The resulting graph can be seen in Figure 8.8(b) (which is the same MST produced by Kruskal's). A visual snapshot after the second iteration can be found in Figure 8.10, detailing the tree, fringe, and unseen vertices/edges.

The performance depends on the data structures we use to maintain fringe vertices/edges. If the priority queue is implemented as a min-heap, deletion and insertion are at most $O(\log m)$. Another important observation is that Prim's does not involve any cycle detection subroutine. Instead edges can be excluded based on whether or not their end points are in the tree vertex set V_T which can be tested in amortized constant time with an appropriate data structure.

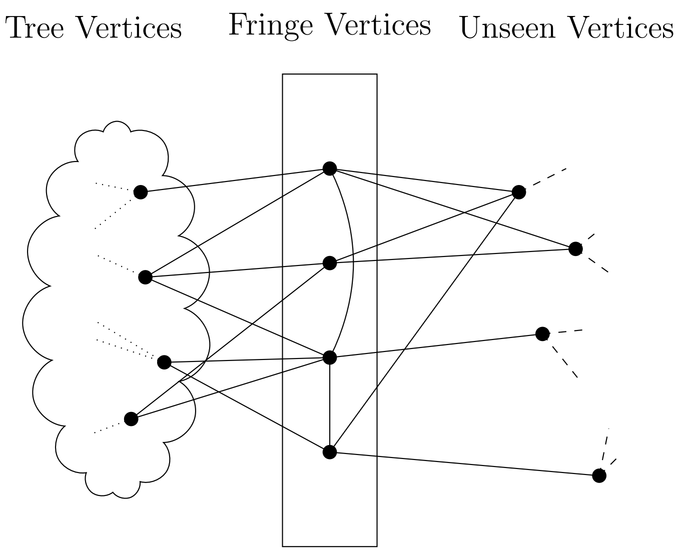


Figure 8.9: Illustration of Tree, Fringe, and Unseen vertex sets.

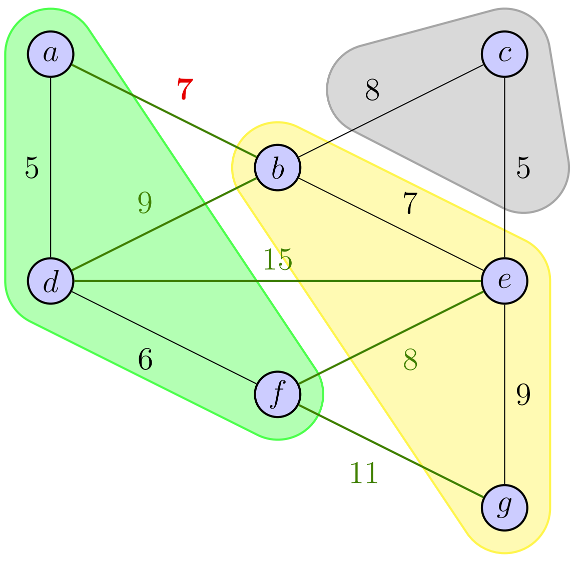


Figure 8.10: Prim's Algorithm after the second iteration. Tree vertices (green), Fringe vertices (yellow), and unseen vertices (grey) are highlighted. Fringe edges (dark green) connect tree vertices and fringe vertices. The next edge to be added, (a, b) is highlighted in red.

Iteration	Tree Vertices	Fringe Edges	Vertex added	Edge Weight
–	$\emptyset$	–	a	–
1	$\{a\}$	$(a, d, 5), (a, b, 7)$	d	5
2	$\{a, d\}$	$(d, f, 6), (a, b, 7),$ $(d, b, 9), (d, b, 15)$	f	6
3	$\{a, d, f\}$	$(a, b, 7), (f, e, 8), (d, b, 9),$ $(f, g, 11), (d, e, 15)$	b	7
4	$\{a, b, d, f\}$	$(b, e, 7), (b, c, 8), (f, e, 8),$ $(f, g, 11), (d, e, 15)$	e	7
5	$\{a, b, d, e, f\}$	$(e, c, 5), (b, c, 8),$ $(e, g, 9), (f, g, 11)$	c	5
6	$\{a, b, c, d, e, f\}$	$(e, g, 9), (f, g, 11)$	g	9

Table 8.4: Sequence of edges considered by Prim's Algorithm starting at vertex a and results.

8.6 Minimum Distance Algorithms

Recall the Shortest Path problem (Problem 10). We have seen a BFS solution that works for several special cases. We now consider two solutions that work for weighted graphs (either directed or undirected).

8.6.1 Dijkstra's Algorithm

Dijkstra's algorithm works making the following greedy choice: from a source vertex s, it chooses a path (edge) to the its nearest neighbor. Then it chooses its second nearest neighbor and so on. By the i-th iteration, it has chosen the shortest paths to $i - 1$ other vertices.

The idea is similar to Prim's in that we make our selection from a set of *fringe* vertices. Each vertex is given two labels: the tree vertex by which the current shortest path can be reached and the length of the shortest path. To choose the next vertex to be added to the tree, we simply choose the minimal length among all fringe vertices, breaking ties arbitrarily.

Once a vertex u has been moved from the fringe to part of the tree, we must update each fringe edge that is adjacent to u^*. If, via u a vertex v can be reached by a *shorter* path,

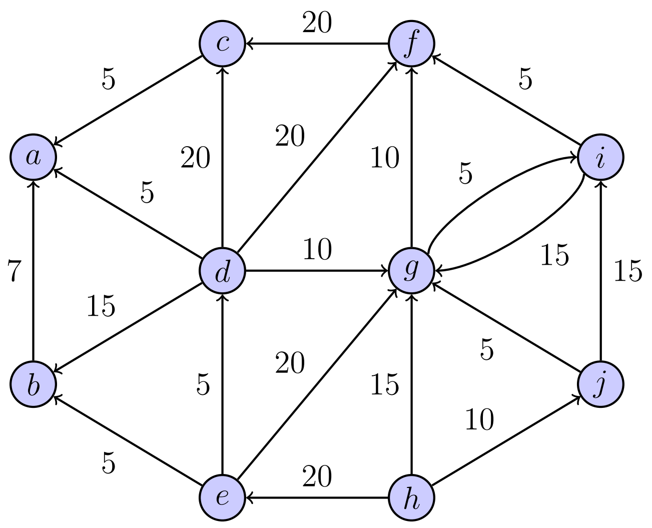


Figure 8.11: Weighted directed graph.

then we update the labels of v to the new shortest path and the vertex label u.

```
INPUT   : A weighted graph, $G = (V, E)$, a source vertex $s \in V$
OUTPUT  : For each $v \in V$, the minimal weighted path $d_v$ for $p : s \rightsquigarrow v$
1  $Q \leftarrow$ empty min-priority queue
2  FOREACH $v \in V \setminus \{s\}$ DO
3      $d_v \leftarrow \infty$
4      $p_v \leftarrow \phi$ //the predecessor to $v$ in the shortest path from $s$
5      $Q.enqueue(v, d_v)$
6  END
7  $d_s \leftarrow 0$
8  $p_v \leftarrow \phi$
9  $Q.enqueue(s, d_s)$
10 $V_T \leftarrow \emptyset$
11 FOR $i = 1, \ldots, n$ DO
12     $u \leftarrow Q.dequeue$
13     $V_T \leftarrow V_T \cup \{u\}$
14     FOREACH $v \in N(u) \setminus V_T$ DO
15         IF $d_u + wt(u, v) < d_v$ THEN
16             $d_v \leftarrow d_u + wt(u, v)$
17             $p_v \leftarrow u$
18             $Q.DecreasePriority(v, d_v)$
19         END
20     END
21 END
22 output $d_{v_1}, \ldots, d_{v_n}$, $p_{v_1}, \ldots, p_{v_n}$,
```

Algorithm 51: Dijkstra's Single-Source Minimum Distance Algorithm

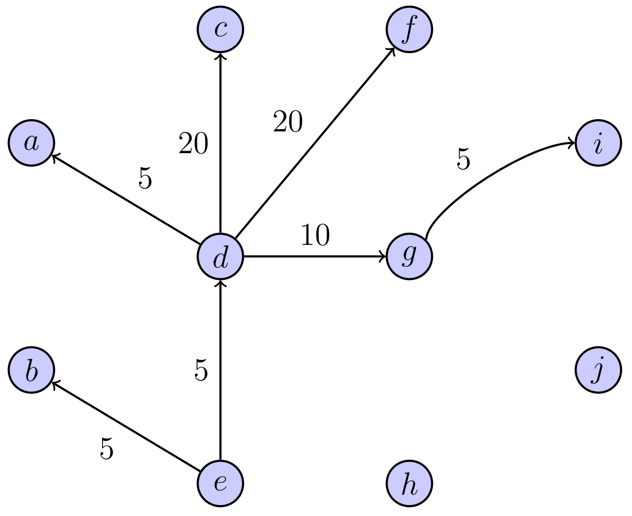


Figure 8.12: Result of Dijsktra's Algorithm with source vertex e.

An example run of Dijkstra's Algorithm on the graph in Figure 8.11 can be found in Table 8.5. The resulting shortest path tree is in Figure 8.12.

If you wanted to actually build a shortest path from the source vertex to any other vertex v, you could "back track" from v using the previous vertex values, p_v computed by Dijkstra's algorithm. Starting at v, you look up $p_v = u$, then you look up p_u and so on until you arrive at the source vertex s.

As presented, using a min-heap implementation (or even a balanced binary search tree implementation) for the priority queue will lead to a loose $O(n^2 \log n)$ analysis. However, using a more advanced heap implementation such as a Fibonacci heap can improve the running time to $O(m + n \log n)$.

8.6.2 Floyd-Warshall Algorithm

Recall that by using BFS, we could determine the shortest path to all vertices from a given source vertex. By running BFS for each vertex we can get *all pairs shortest paths.*

Floyd's algorithm works equally well on directed and undirected graphs, but is particularly interesting for *weighted* graphs of either type.

Pitfall: Floyd's algorithm does not work on a graph with a negatively weighted cycle. Why?

Floyd's starts with the usual adjacency matrix with entries that are the weights of the edges. Much like Warshall's, Floyd's algorithm computes intermediate distance matrices,

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}$$

Each intermediate distance matrix $D^{(k)}$ contains entries $d_{i,j}$ which correspond to the

(a) Prior to the first iteration.

Vertex	d_v	p_v
a	∞	ϕ
b	∞	ϕ
c	∞	ϕ
d	∞	ϕ
e	0	ϕ
f	∞	ϕ
g	∞	ϕ
h	∞	ϕ
i	∞	ϕ
j	∞	ϕ

(b) First iteration, $N(e)$ is explored.

Vertex	d_v	p_v
a	∞	ϕ
b	5	e
c	∞	ϕ
d	5	e
f	∞	ϕ
g	20	e
h	∞	ϕ
i	∞	ϕ
j	∞	ϕ

(c) Second iteration, $N(b)$ is explored

Vertex	d_v	p_v
a	12	b
c	∞	ϕ
d	5	e
f	∞	ϕ
g	20	e
h	∞	ϕ
i	∞	ϕ
j	∞	ϕ

(d) Iteration 3, $N(d)$ is explored

Vertex	d_v	p_v
a	10	d
c	25	d
f	25	d
g	15	d
h	∞	ϕ
i	∞	ϕ
j	∞	ϕ

(e) Iteration 4, $N(a)$ is explored

Vertex	d_v	p_v
c	25	d
f	25	d
g	15	d
h	∞	ϕ
i	∞	ϕ
j	∞	ϕ

(f) Iteration 5, $N(g)$ is explored

Vertex	d_v	p_v
c	25	d
f	25	d
h	∞	ϕ
i	20	g
j	∞	ϕ

(g) Iteration 6, $N(i)$ is explored

Vertex	d_v	p_v
c	25	d
f	25	d
h	∞	ϕ
j	∞	ϕ

(h) Final Shortest Paths from e and Precesessors

Vertex	d_v	p_v
a	10	d
b	5	e
c	25	d
d	5	e
e	0	ϕ
f	25	d
g	15	d
h	∞	ϕ
i	20	g
j	∞	ϕ

Table 8.5: Walkthrough of Dijkstra's Algorithm for source vertex e . Iterations 7 – 9 are omitted as they do not result in any further changes.

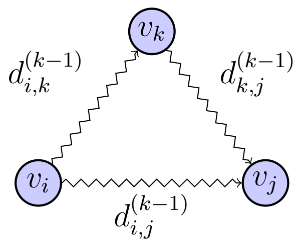


Figure 8.13: Basic idea behind Floyd-Warshal: Supposing that a path from $v_i \rightsquigarrow v_j$ has already been found with a distance of $d_{i,j}^{(k-1)}$, the consideration of v_k as a new intermediate node may have shorter distance, $d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$.

total weight of the shortest path from v_i to v_j not using any vertex numbered higher than k.

Also as before, we can compute each intermediate distance matrix by using its immediate predecessor. The same reasoning as before yields a similar recurrence:

$$d_{i,j}^{(k)} \leftarrow \min \left\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right\}, d_{i,j}^{(0)} = w_{i,j}$$

The idea behind this algorithm is illustrated in Figure 8.13.

```
    INPUT  : A weighted graph, $G = (V, E)$
    OUTPUT : A shortest distance matrix $D$ of $G$
 1  $D \leftarrow (n \times n)$ matrix
 2  FOR $1 \leq i, j \leq n$ DO
 3      $d_{i,j} \leftarrow wt(v_i, v_j)$ //$\infty$ if no such edge
        //Initialize the successor matrix $S$:  set $s_{i,j} = j$ for all edges
          $(v_i, v_j)$
 4  END
 5  FOR $k = 1, \ldots, n$ DO
 6      FOR $i = 1, \ldots, n$ DO
 7          FOR $j = 1, \ldots, n$ DO
 8              $d_{i,j} = \min\{d_{i,j}, d_{i,k} + d_{k,j}\}$
                //if $d_{i,j}$ is updated, set the successor matrix value
                  $s_{i,j} = s_{i,k}$
 9          END
10      END
11  END
12  output $D$
```

Algorithm 52: Floyd's All Pair Minimum Distance Algorithm

Observations:

- We can keep track of which k corresponds to an update to reconstruct the minimum paths
- Clearly the algorithm is $O(n^3)$.
- An example of a full run of Floyd's can be found in Figure 8.14

A mentioned, we can keep track of which value(s) of k caused the matrix entries to change. This results in a *successor* matrix S as depicted in Figure 8.14(h). We now describe how we can use this successor matrix to construct the shortest path for any pair.

Suppose that we wanted to construct the shortest path $v_i \rightsquigarrow v_j$. We start at v_i; to find the next vertex, we reference the successor matrix S by looking at the i-th row and j-th column which gives us the next vertex in the shortest path. Suppose this is v_ℓ. Then to find the next vertex, we want the entry in the ℓ-th row, j-th column. We continue until the entry is equal to v_j. The complete pseudocode is found in Algorithm 53. For the example, the shortest path $a \rightsquigarrow d$ is

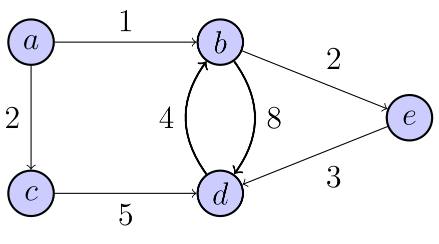
$$p : a \rightarrow b \rightarrow e \rightarrow d$$

```
INPUT  : A successor matrix $S$ produced by Floyd-Warshall for a graph
         $G = (V, E)$, two vertices $v_i, v_j \in V$
OUTPUT : The shortest path $p : v_i \rightsquigarrow v_j$ in $G$
1 $p \leftarrow v_i$
2 $x \leftarrow v_i$
3 WHILE $x \neq v_j$ DO
4     $x \leftarrow S_{x,v_j}$
5     $p \leftarrow p + x$
6 END
7 output $p$
```

Algorithm 53: Construct Shortest Path Algorithm

We present another full example. Consider the graph in Figure 8.15(a) which represents a *tournament graph* (a complete Directed Acyclic Graph (DAG)). Each iteration of Floyd-Warshall updates entries in the upper diagonal because a new, shorter path found each time. The matrices at each iteration are presented in Figure 8.15 along with the successor matrix. Consider constructing the shortest path $a \rightsquigarrow e$ using this matrix. We first look at entry $S_{a,e}$ which is b so b is the next vertex in the shortest path. Then we look at the entry $S_{b,e}$ which is c. Proceeding in this manner until $S_{d,e} = e$ we construct the path:

$$p : a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$$



(a) A weighted directed graph.

$$\begin{bmatrix} 0 & 1 & 2 & \infty & \infty \\ \infty & 0 & \infty & 8 & 2 \\ \infty & \infty & 0 & 5 & \infty \\ \infty & 4 & \infty & 0 & \infty \\ \infty & \infty & \infty & 3 & 0 \end{bmatrix}$$

(b) Initial distance matrix

$$\begin{bmatrix} 0 & 1 & 2 & \infty & \infty \\ \infty & 0 & \infty & 8 & 2 \\ \infty & \infty & 0 & 5 & \infty \\ \infty & 4 & \infty & 0 & \infty \\ \infty & \infty & \infty & 3 & 0 \end{bmatrix}$$

(c) Distance matrix after iteration $k = 1$

$$\begin{bmatrix} 0 & 1 & 2 & 9 & 3 \\ \infty & 0 & \infty & 8 & 2 \\ \infty & \infty & 0 & 5 & \infty \\ \infty & 4 & \infty & 0 & 6 \\ \infty & \infty & \infty & 3 & 0 \end{bmatrix}$$

(d) Distance matrix after iteration $k = 2$

$$\begin{bmatrix} 0 & 1 & 2 & 7 & 3 \\ \infty & 0 & \infty & 8 & 2 \\ \infty & \infty & 0 & 5 & \infty \\ \infty & 4 & \infty & 0 & 6 \\ \infty & \infty & \infty & 3 & 0 \end{bmatrix}$$

(e) Distance matrix after iteration $k = 3$

$$\begin{bmatrix} 0 & 1 & 2 & 7 & 3 \\ \infty & 0 & \infty & 8 & 2 \\ \infty & 9 & 0 & 5 & 11 \\ \infty & 4 & \infty & 0 & 6 \\ \infty & 7 & \infty & 3 & 0 \end{bmatrix}$$

(f) Distance matrix after iteration $k = 4$

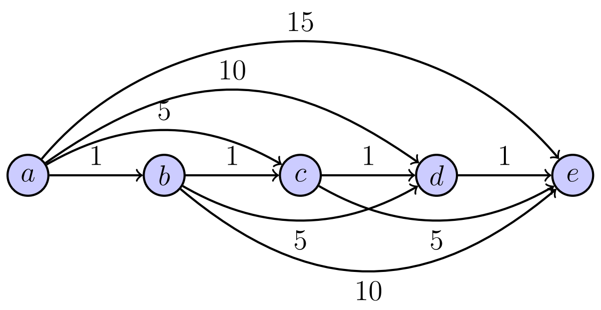
$$\begin{bmatrix} 0 & 1 & 2 & 6 & 3 \\ \infty & 0 & \infty & 5 & 2 \\ \infty & 9 & 0 & 5 & 11 \\ \infty & 4 & \infty & 0 & 6 \\ \infty & 7 & \infty & 3 & 0 \end{bmatrix}$$

(g) Distance matrix after iteration $k = 5$

$$\begin{bmatrix} - & b & c & b & b \\ - & - & - & e & e \\ - & d & - & d & d \\ - & b & - & - & b \\ - & d & - & d & - \end{bmatrix}$$

(h) Successor matrix produced by Floyd's Algorithm

Figure 8.14: Floyd's Algorithm Demonstration.



(a) A weighted directed graph.

$$\begin{bmatrix} 0 & 1 & 5 & 10 & 15 \\ \infty & 0 & 1 & 5 & 10 \\ \infty & \infty & 0 & 1 & 5 \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \quad \begin{bmatrix} a & b & c & d & e \\ - & b & c & d & e \\ - & - & c & d & e \\ - & - & - & d & e \\ - & - & - & - & e \end{bmatrix}$$

(b) Initial distance and successor matrices

$$\begin{bmatrix} 0 & 1 & 5 & 10 & 15 \\ \infty & 0 & 1 & 5 & 10 \\ \infty & \infty & 0 & 1 & 5 \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \begin{bmatrix} a & b & c & d & e \\ - & b & c & d & e \\ - & - & c & d & e \\ - & - & - & d & e \\ - & - & - & - & e \end{bmatrix}$$

(c) After iteration $k = 1$

$$\begin{bmatrix} 0 & 1 & 2 & 6 & 11 \\ \infty & 0 & 1 & 5 & 10 \\ \infty & \infty & 0 & 1 & 5 \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \begin{bmatrix} a & b & b & b & b \\ - & b & c & d & e \\ - & - & c & d & e \\ - & - & - & d & e \\ - & - & - & - & e \end{bmatrix}$$

(d) After iteration $k = 2$

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 7 \\ \infty & 0 & 1 & 2 & 6 \\ \infty & \infty & 0 & 1 & 5 \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \begin{bmatrix} a & b & b & b & b \\ - & b & c & c & c \\ - & - & c & d & e \\ - & - & - & d & e \\ - & - & - & - & e \end{bmatrix}$$

(e) After iteration $k = 3$

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ \infty & 0 & 1 & 2 & 3 \\ \infty & \infty & 0 & 1 & 2 \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \begin{bmatrix} a & b & b & b & b \\ - & b & c & c & c \\ - & - & c & d & d \\ - & - & - & d & e \\ - & - & - & - & e \end{bmatrix}$$

(f) After iteration $k = 4$

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ \infty & 0 & 1 & 2 & 3 \\ \infty & \infty & 0 & 1 & 2 \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix} \begin{bmatrix} a & b & b & b & b \\ - & b & c & c & c \\ - & - & c & d & d \\ - & - & - & d & e \\ - & - & - & - & e \end{bmatrix}$$

(g) After iteration $k = 5$

Figure 8.15: Another Demonstration of Floyd-Warshall's Algorithm.

8.6.3 Huffman Coding

See Tree Notes

8.7 Exercises

Exercise 8.1. Consider a weighted, complete graph K_n (where all vertices are connected by edges). Further, assume that we are given G as an adjacency list where the list for each vertex is ordered in increasing order with respect the edge weights. Describe an $O(|V|)$-time algorithm to compute a minimum spanning tree for such an input.

Exercise 8.2. The Food Bucket, a regional chain of restaurants wants you to develop a program that will generate mazes for its children's menu. Rather than create one maze, they want a program that will generate a random maze on an $n \times n$ grid such that there is only one solution (that is, one path from the lower left to the upper right corners). The algorithm should produce a random maze with various paths connecting grid points (representing a traversable path) with only a single path leading from grid point $(0, 0)$ to grid point $(n-1, n-1)$. There should also not be any cycles. The algorithm should be correct and efficient.

Exercise 8.3. Let $G = (V, E)$ be an undirected, unweighted graph. Let $\delta(u, v)$ be the minimum distance between vertices $u, v \in V$. The *eccentricity* of a vertex $v \in V$ is defined as the maximal minimum distance between v and any other vertex in G:

$$\epsilon(v) = \max_{u \in V} \delta(v, u)$$

The *diameter* d of G is then defined as the maximal eccentricity of any vertex in G:

$$d = \max_{v \in V} \epsilon(v)$$

The *radius* of a graph is the minimum eccentricity of any vertex in G,

$$r = \min_{v \in V} \epsilon(v)$$

(a) Design an algorithm that utilizes either DFS, BFS, or some variation/application of DFS/BFS to compute the radius of a graph G. Provide good pseudocode and give a brief analysis of your algorithm.

(b) Design an algorithm that utilizes either DFS, BFS, or some variation/application of DFS/BFS to compute the diameter of a graph G. Provide good pseudocode and give a brief analysis of your algorithm.

Exercise 8.4. Give an example of weighted (directed or undirected) in which the BFS Tree does not provide the shortest distance path.

Exercise 8.5. Reconsider the condensation graph G_c of a directed graph G.

(a) Suppose that G is strongly connected. What does G_c look like?

(b) Suppose that G is a DAG, what does G_c look like?

Exercise 8.6. Give an algorithm to solve the following problem: given a graph $G = (V, E)$ and vertices u, v, x, determine if there exists a path from u to v that involves x

Exercise 8.7. Provide a small example of a weighted undirected graph whose MST produced from Kruskal's algorithm would be *different* from that produced by Prim's algorithm.

Exercise 8.8. Suppose that we restrict the MST problem to weighted graphs whose weights are all positive integers; that is $wt : E \rightarrow \mathbb{Z}^+$. Show that each of the following variations are equivalent to this formulation by showing a transformation between them. That is, describe a transformation from the variation below to the positive weighted version and describe how a solution to the positive weighted version can be transformed back to a solution for the variation.

1. Let $wt : E \rightarrow \mathbb{Z}$ (that is we allow negative and zero weighted edges)
2. Rather than finding the minimum spanning tree, suppose that we wish to find the *maximum* spanning tree

Exercise 8.9. Let $G = (V, E)$ be an undirected graph (unweighted). Prove or disprove: the minimum spanning tree T formed by Kruskal's algorithm also provides a *minimum distance tree*: that is, for any two vertices x, y, the unique path between them in the MST T is also the shortest path in the original graph G.

Exercise 8.10. Let $G = (V, E)$ be an undirected graph (unweighted) with $x \in V$. Prove or disprove: the minimum spanning tree T formed by Prim's algorithm by starting at x also provides a *minimum distance tree*: that is the unique path from x to any other node y in T is the shortest path from x to y in G.

Exercise 8.11. Each iteration of Kruskal's algorithm considers adding an edge $e = (u, v)$ to the current minimum spanning tree T by checking whether or not its inclusion in T would induce a cycle. Gomer thinks he's found a better way: rather than checking for a cycle, just check if the end points of e are already in T: if they are, then do not include the edge as they would not add any vertex to the minimum spanning tree. If either end point (u or v) is outside the current tree then do add the edge as it would connect the tree further. Show that Gomer is wrong by providing an example of a tree where using this criteria instead would fail. Briefly explain why Gomer's criteria is wrong.

Exercise 8.12. Recall that a graph is a *tree* if it contains no cycles. Adapt either DFS or BFS to develop an algorithm that determines if a given graph $G = (V, E)$ is a tree or not. Provide good pseudocode and fully analyze your algorithm.

Exercise 8.13. Kruskal's algorithm works by checking connectivity as a sub routine. Connectivity questions are usually solved using a DFS or BFS which will require a data structure such as a stack or a queue which may need to hold up to $O(n)$ references to vertices. Assume that we have the following function (oracle, subroutine, etc.): $isConnected(G, u, v)$, that, given an undirected graph $G = (V, E)$ and two vertices $u, v \in V$ outputs true if there is a path between u, v in G and false otherwise.

Show that Kruskal's algorithm can be performed without the use of DFS/BFS or any $O(n)$ extra space (that is, show that it can be performed with only constant extra space.

Exercise 8.14. Implement DFS in the high-level programming language of your choice.

Exercise 8.15. Implement BFS in the high-level programming language of your choice.

Exercise 8.16. Implement Kruskal's in the high-level programming language of your choice.

Exercise 8.17. Implement Prim's in the high-level programming language of your choice.

Exercise 8.18. Implement Dijkstra's Algorithm in the high-level programming language of your choice.

Exercise 8.19. Implement Floyd's Algorithm in the high-level programming language of your choice.

9 Dynamic Programming

9.1 Introduction

One pitfall of recursive solutions is unnecessarily recomputing solutions. A classic example is the Fibonacci sequence.

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, F_1 = 1$$

Observing the computation tree of a recursive algorithm to compute a fibonacci number we realize that many of the computations are repeated an exponential number of times.

One method for avoiding recomputing values is *memoization*: once a value has been computed it is placed into a table. Then, instead of recomputing it we check the table; if it has been computed already, we use the value stored in the table, if not then we go through the effort of computing it.

Both of these approaches are fundamentally *top-down* solutions. That is, they start with the larger problem, then attempt to solve it by computing solutions to sub problems.

Dynamic Programming is an algorithmic technique that seeks *bottom-up* solutions. Rather than dividing the problem up into sub-problems, dynamic programming attempts to solve the sub-problems *first*, then uses them to solve the larger problem.

This approach typically works by defining a *tableau* (or several) and filling them out using previously computed values in the tableau. In general, recursion is avoided and the algorithm is achieved with just a few loops.

9.1.1 Optimal Substructure Property

In general, for a problem to have a dynamic programming solution, it must posses the *optimal substructure property.* This is a formalization of the idea that the optimal solution to a problem can be formulated by (efficiently) finding the optimal solution to each of its sub-problems. This property is also integral to greedy algorithmic solutions. Obviously, not all problems poses this property.

9.2 Binomial Coefficients

Recall the choose function:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

which is usually referred to as a binomial coefficient as it represents the k-th coefficient in the expansion of a binomial:

$$(x+y)^n = \binom{n}{0}x^n + \cdots + \binom{n}{k}x^{n-k}y^k + \cdots \binom{n}{n}y^n$$

Pascal's identity allows one to express binomial coefficients as the sum of other binomial coefficients:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

with

$$\binom{n}{0} = \binom{n}{n} = 1$$

as base cases to the recursive definition.

However, such a solution turns out to be exponential as many of the sub-solutions are recomputed many times. To illustrate, consider Code Snippet 9.1. A recent run of this program on the cse server to compute `binomial(38,12)` resulted in $1,709,984,303$ recursive calls and took about 20 seconds. In contrast, using memoization to avoid repeated computations resulted in only 573 function calls and took less than 1ms.

```
long binomial(int n, int k) {

  if(k < 0 || n < 0) {
    return -1;
  } else if(k == 0 || n == k) {
    return 1;
  } else {
    return binomial(n-1, k-1) + binomial(n-1, k);
  }
}
```

Code Sample 9.1: Recursive Binomial Computation

Memoization is still a top-down solution that utilizes recursion. Dynamic programming is a bottom-up solution that does not make function calls, but rather fills out a tableau of values.

In the case of binomial coefficients, we define a $(n+1) \times (k+1)$ sized table (see Table 9.1).

	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
$\vdots$						$\ddots$		
k	1	$\cdots$						1
$\vdots$								
$n-1$	1	$\cdots$					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n	1	$\cdots$						$\binom{n}{k}$

Table 9.1: Tableau for Binomial Coefficients. Astute readers will recognize the tableau as Pascal's Triangle.

The tableau is filled out top-to-bottom, left-to-right with the trivial base cases pre-filled.

```
INPUT  : Integers, $n, k, 0 \leq k \leq n$
OUTPUT : The binomial coefficient $\binom{n}{k}$
FOR $i = 0, \ldots, n$ DO
    FOR $j = 0, \ldots, \min i, k$ DO
        IF $j = 0 \vee j = k$ THEN
            $C_{i,j} \leftarrow 1$
        ELSE
            $C_{i,j} \leftarrow C_{i-1,j-1} + C_{i-1,j}$
        END
    END
END
output $C_{n,k}$
```

Algorithm 54: Binomial Coefficient – Dynamic Programming Solution

Analysis: clearly the elementary operation is addition and in total, the algorithm is $\Theta(nk)$. This matches the complexity of a memoization approach, but avoids any additional function calls.

9.3 Optimal Binary Search Trees

We've considered Binary Search Trees and Balanced BSTs. Consider the following variation: suppose we know in advance the keys to be searched along with a known (or estimated) probability distribution of searches on these keys.

We don't necessarily want a balanced binary search tree, rather we want a BST that will minimize the overall expected (average) number of key comparisons. Intuitively, keys with higher probabilities should be shallower in the tree while those with lower probabilities should be deeper. Specifically, average number of comparisons made would be:

$$\sum_{i=1}^{n} h(k_i) \cdot p(k_i)$$

where $h(k_i)$ is the level in which the key k_i lies and $p(k_i)$ is the probability of searching for k_i. Our goal will be to compute the *Optimal Binary Search Tree* (OBST).

As an example, consider the key distribution in Table 9.2 and the trees in Figure 9.1. There are several valid BSTs, but the expected number of comparisons given the key probabilities are different.

k	$p(k)$
a	0.6
b	0.3
c	0.1

Table 9.2: Probability distribution for keys

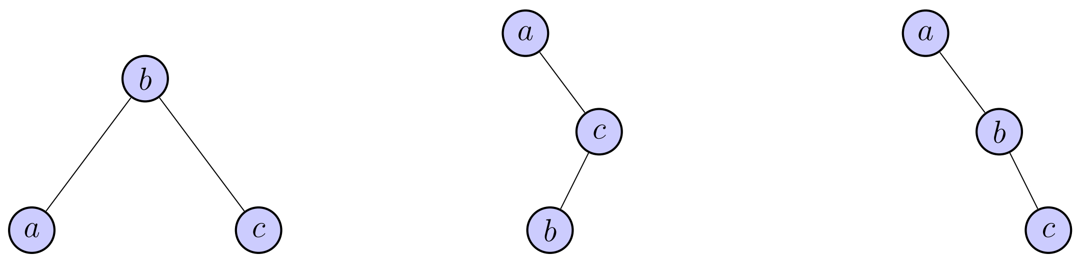
The number of possible BSTs with n keys corresponds to the Catalan numbers.

Definition 13. The *Catalan* numbers are a sequence of natural numbers defined by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

- The first few numbers in the sequence: $1, 1, 2, 5, 14, 42, 132, 429, \ldots,$
- Corresponds to the number of valid, balanced parenthesization of operations:

$$((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd))$$



(a) A balanced tree, but the expected number of comparisons is 1.7

(b) Another tree, but the expected number of comparisons is still 1.7

(c) The optimal configuration, the expected number of comparisons is 1.5

Figure 9.1: Several valid Binary Search Trees

- Corresponds to the number of "full" (every node has 0 or 2 children) binary trees with $n+1$ leaves. This is the same a binary tree with n key nodes (which will necessarily have $n+1$ *sentinel* leaves).
- *Many* other interpretations
- They have an exponential growth:

$$C_n \in O\left(\frac{4^n}{n^{1.5}}\right)$$

OBSTs are not ideal for applications that perform a lot of insertion and deletion operations. The introduction of a new key or the elimination of a key will necessarily change the probability distribution and a new OBST may have to be generated from scratch. The use cases are for more static, indexed collections that do not change frequently.

For a dynamic programming solution (due to Knuth [9]), we need a recurrence. Let $C_{i,j}$ be defined as the smallest average number of comparisons made in a successful search in a binary tree $T_{i,j}$ involving keys k_i through k_j for $1 \leq i, j \leq n$. Ultimately, we are interested in finding $C_{1,n}$—the optimal BST involving all keys.

To define a recurrence, we need to consider which key should be the root of the intermediate tree $T_{i,j}$, say k_l. Such a tree will have k_l as the root and a left subtree, $T_{i,l-1}$ containing keys $k_i, \ldots k_{l-1}$ optimally arranged and the right-sub-tree, $T_{l+1,j}$ containing keys $k_{l+1}, \ldots, j$.

$$C_{i,j} = \min_{i \leq k \leq j} \left\{ C_{i,k-1} + C_{k+1,j} \right\} + \sum_{s=i}^{j} p(k_s)$$

for $1 \leq i \leq j \leq n$. Note that the sum is invariant over all values of k: it represents the fact that building the tree necessarily adds 1 comparison to the depth of all the keys. A visualization of the split can be found in Figure 9.2.

Some obvious corner cases:

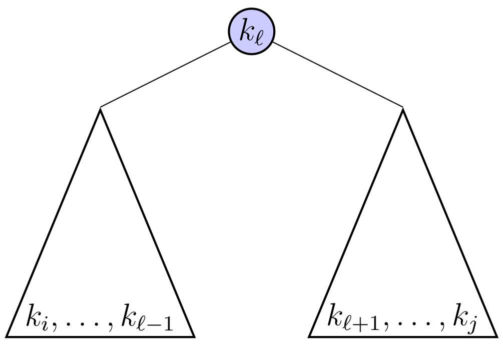


Figure 9.2: Optimal Binary Search Tree split

$i \backslash j$	0	1	2	$\cdots$	n
1	0	$p(k_1)$			$C_{1,n}$
2	0	0	$p(k_2)$		
$\vdots$				$\ddots$	
n	0	0	$\cdots$	0	$p(k_n)$
$n+1$	0	0	$\cdots$	0	0

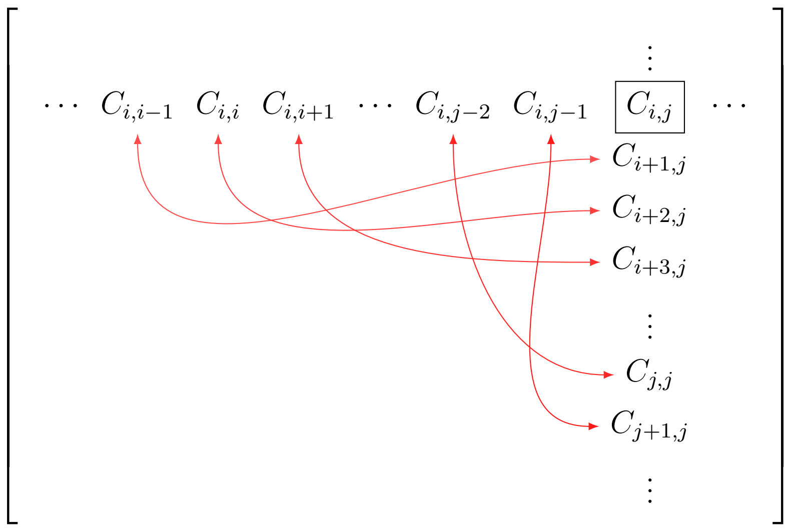
Table 9.3: Optimal Binary Search Tree Tableau

- $C_{i,i-1} = 0$ for $1 \leq i \leq n+1$ since no comparisons are made in an empty tree.
- $C_{i,i} = p(k_i)$ for $1 \leq i \leq n$ since an optimal tree of size 1 is the item itself.

With these values, we can define an $(n+1) \times (n+1)$ tableau (see Table 9.3).

Each entry in the tableau requires previous values in the same row to the left and in the same column below as depicted in Figure 9.3. Thus, the tableau is filled out along the diagonals from top-left to bottom-right. The first diagonal is initialized to zero and the second with each of the respective probabilities of each key. Then each successive table entry is calculated according to our recurrence.

The final value $C_{1,n}$ is what we seek. However, this gives us the average number of key comparisons (minimized), not the actual tree. To build the tree, we also need to maintain a *root table* that keeps track of the values of k for which we choose the minimum (thus k_k is the root and we can build top down).



$\cdots\ C_{i,i-1}\ C_{i,i}\ C_{i,i+1}\ \cdots\ C_{i,j-2}\ C_{i,j-1}\ \boxed{C_{i,j}}\ \cdots$
$C_{i+1,j}$
$C_{i+2,j}$
$C_{i+3,j}$
$C_{j,j}$
$C_{j+1,j}$

Figure 9.3: Visualization of the OBST tableau. Computing the minimal value for $C_{i,j}$ involves examining potential splits of keys, giving pairs of sub-solutions. This indicates that we need elements in the same row to the left and in the same column below. Thus, the algorithm needs to complete the tableau top-left to bottom-right diagonally.

A stack-based algorithm for constructing the OBST is presented in Algorithm 56.

```
INPUT  : A set of keys $k_1, \dots, k_n$ and a probability distribution $p$ on the keys
OUTPUT : An optimal Binary Search Tree
FOR $i = 1, \dots, n$ DO
    $C_{i,i-1} \leftarrow 0$
    $C_{i,i} \leftarrow p(k_i)$
    $R_{i,i} \leftarrow i$
END
$C_{n+1,n} \leftarrow 0$
FOR $d = 1, \dots, (n-1)$ DO
    FOR $i = 1, \dots, (n-d)$ DO
        $j \leftarrow i + d$
        $min \leftarrow \infty$
        FOR $k = i, \dots, j$ DO
            $q \leftarrow C_{i,k-1} + C_{k+1,j}$
            IF $q < min$ THEN
                $min \leftarrow q$
                $R_{i,j} \leftarrow k$
            END
        END
        $C_{i,j} \leftarrow min + \sum_{s=i}^{j} p(k_s)$
    END
END
output $C_{1,n}, R$
```

Algorithm 55: Optimal Binary Search Tree

```
INPUT   : A set of keys $k_1, \ldots, k_n$ and root Table $R$
OUTPUT  : The root node of the OBST
root ← new root node
$r.key \leftarrow R_{1,n}$
$S \leftarrow$ empty stack
$S.push(r, 1, n)$ //In general, we push a node $u$, and indices $i, j$
WHILE $S$ is not empty DO
    $(u, i, j) \leftarrow S.pop$
    $k \leftarrow R_{i,j}$ //this is the key corresponding to $u$
    IF $k < j$ THEN
        //create the right child and push it
        $v \leftarrow$ new node
        $v.key \leftarrow R_{k+1,j}$
        $u.rightChild \leftarrow v$
        $S.push(v, k+1, j)$
    END
    IF $i < k$ THEN
        //create the left child and push it
        $v \leftarrow$ new node
        $v.key \leftarrow R_{i,k-1}$
        $u.leftChild \leftarrow v$
        $S.push(v, i, k-1)$
    END
END
output root
```

Algorithm 56: OBST Tree Construction

9.3.1 Example

The full tableaus can be found in Table 9.5. As an example computation:

$$C_{1,2} = \min_{1 \leq k \leq 2} \begin{cases} k = 1 : C_{1,0} + C_{2,2} + \sum_{s=1}^{2} p(k_s) = 0 + .02 + (2.13 + .02) = .253 \\ k = 2 : C_{1,1} + C_{3,2} + \sum_{s=1}^{2} p(k_s) = 0.213 + .02 + (2.13 + .02) = .446 \end{cases}$$

These two options correspond to splitting the sub-tree involving keys A, B at A, B respectively. That is, A as the root with B as its right-child *or* B as the root with A as its left-child. The values indicate that $k = 1$ is the better option.

Key	Probability
A	.213
B	.020
C	.547
D	.100
E	.120

Table 9.4: Optimal Binary Search Tree Example Input

(a) Cost tableau for OBST

$i \backslash j$	0	1	2	3	4	5
1	0	0.213	0.253	1.033	1.233	1.573
2		0	0.020	0.587	0.787	1.127
3			0	0.547	0.747	1.087
4				0	0.100	0.320
5					0	0.120
6						0

(b) Root array indicating which keys resulted in the optimal split.

$i \backslash j$	1	2	3	4	5
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

Table 9.5: Tableaus resulting from the Optimal Binary Search Tree example

$$C_{2,5} = \min_{2 \leq k \leq 5} \begin{cases} k = 2 : C_{2,1} + C_{3,5} + \sum_{s=2}^{5} p(k_s) = 1.874 \\ k = 3 : C_{2,2} + C_{4,5} + \sum_{s=2}^{5} p(k_s) = 1.127 \\ k = 4 : C_{2,3} + C_{5,5} + \sum_{s=2}^{5} p(k_s) = 1.494 \\ k = 5 : C_{2,4} + C_{6,5} + \sum_{s=2}^{5} p(k_s) = 1.573 \end{cases}$$

The final result can be seen in Figure 9.4. The expected number of key comparisons for any search is 1.573. Though this particular example resulted in a balanced BST, in general, OBSTs need not be balanced.

9.4 Dynamic Knapsack

Recall the 0-1 Knapsack Problem (see Problem 4). This problem lends itself to an "efficient" dynamic programming solution.

Let $V_{i,j}$ be the optimal solution (a subset of items) involving the first i objects subject to an intermediate weight constraint j. Clearly, $1 \leq i \leq n$ and $1 \leq j \leq W$.

For a given $V_{i,j}$ we can divide all the possible subsets of the first i items that fit a knapsack capacity of j into two groups: those that include item i and those that do not.

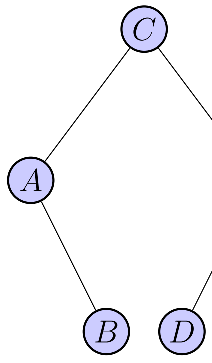


Figure 9.4: Final Optimal Binary Search Tree.

- Among subsets that *do not* include the i-th element, the value of the optimal subset is $V_{i-1,j}$.
- Among the subsets that *do* include the i-th element, the optimal subset will be made up of the i-th item and the optimal subset of the first $i-1$ items that fit into a knapsack of capacity $j - w_i$. This is exactly

  $$v_i + V_{i-1,j-w_i}$$

  .

We are interested in maximizing our total value, so we take the max of these two solutions if feasible.

$$V_{i,j} = \begin{cases} \max\left\{V_{i-1,j},\ v_i + V_{i-1,j-w_i}\right\} & \text{if } j - w_i \geq 0 \\ V_{i-1,j} & \text{if } j - w_i < 0 \end{cases}$$

In addition, we define the initial conditions

- $V_{0,j} = 0$ for $j \geq 0$ (taking no items has no value)
- $V_{i,0} = 0$ for $i \geq 0$ (no capacity means we can take no items)

The tableau that we fill out will be a $(n+1) \times (W+1)$ (rows, columns) sized table numbered $0 \ldots n$ and $0 \ldots W$ respectively.

The tableau can be filled out as follows. For each entry, we can take the maximum of:

- the entry in the previous row of the same column and
- the sum of v_i and the entry in the previous row and w_i columns to the left.

This enables us to calculate row-by-row (left to right) or column-by-column (top to bottom).

We can also build the optimal solution using the same table by working backwards. For each intermediate capacity w, item i is in the optimal subset if $V_{i,w} \neq V_{i-1,w}$. If this is the case, we can update the intermediate capacity, $w - w_i$ and look at the corresponding column to continue.

$i \backslash j$	0	$\cdots$	$j - w_i$	$\cdots$		$\cdots$	W
0	0	$\cdots$	0	$\cdots$	0	$\cdots$	0
1	0	$\cdots$		$\cdots$		$\cdots$	
$\vdots$							
$i-1$	0	$\cdots$	$V_{i-1,j-w_i}$	$\cdots$	$V_{i-1,j}$	$\cdots$	
i	0	$\cdots$		$\cdots$	$V_{i,j}$	$\cdots$	
$\vdots$							
n	0	$\cdots$		$\cdots$	$V_{i,j}$	$\cdots$	$V_{n,W}$

Table 9.6: Tableau for Dynamic Knapsack. The value of an entry $V_{i,j}$ depends on the values in the previous row and columns.

Visually, we start with entry $V_{n,W}$. We then scan upwards in this column until the table value changes. A change from row i to row $i-1$ corresponds to taking item i with weight w_i. We then jump left in the table on row $i-1$ to column $j - w_i$ (where j is the column we started out) and repeat the process until we have reached the border of the tableau.

```
INPUT  : Completed tableau $V$ of a Dynamic Programming 0-1 knapsack
         solution
OUTPUT : The optimal knapsack $S$
$S \leftarrow \emptyset$
$i \leftarrow n$
$j \leftarrow W$
WHILE $i \geq 1 \wedge j \geq 1$ DO
    WHILE $i \geq 1 \wedge V_{i,j} = V_{i-1,j}$ DO
        $i \leftarrow (i-1)$
    END
    $S \leftarrow S \cup \{a_i\}$
    $j \leftarrow (j - w_i)$
    $i \leftarrow (i-1)$
END
output $S$
```

Algorithm 57: 0-1 Knapsack Generator

Item	Weight	Value
a_1	5	10
a_2	2	5
a_3	4	8
a_4	2	7
a_5	3	7

Table 9.7: Example input for 0-1 Knapsack

i \ j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10
2	0	0	5	5	5	10	10	15
3	0	0	5	5	8	10	13	15
4	0	0	7	7	12	12	15	17
5	0	0	7	7	12	14	15	19

Table 9.8: Resulting tableau for the 0-1 Knapsack Example

9.4.1 Example

The resulting tableau can be found in figure 9.8. The corresponding optimal knapsack would be $\{a_2, a_4, a_5\}$; the backtracking is illustrated in Figure 9.5.

9.4.2 Analysis

Clearly, the running time is equivalent to the number of entries that we compute, $\Theta(nW)$. Unfortunately this fails to give a complete picture. Recall that W is part of the input, thus the input *size* is $\log W$. If $W \in O(2^n)$ for example, this is clearly not polynomial.

Algorithms with analysis like this are called *pseudopolynomial* (or more specifically, *pseudolinear* in this case). This is because another parameter of the input "hides" the true complexity of the algorithm. If we considered W to be constant or even $W \in O(n^k)$ then the algorithm runs in polynomial time. In general, such restrictions are not reasonable and the problem remains NP-complete.

9.5 Coin Change Problem

The Coin Change problem is the problem of giving the minimum number of coins adding up to a total L in change using a given set of denominations $\{c_1, \ldots, c_n\}$. For certain

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
a_1	0	0	0	0	0	10	10	10
a_2	0	0	5	5	5	10	10	15
a_3	0	0	5	~~5~~	8	10	13	15
a_4	0	0	7	7	12	~~12~~	~~15~~	17
a_5	0	0	7	7	12	14	15	19

Figure 9.5: Dymamic Knapsack Backtracking

denominations (like US currency, how fortunate!), a simple greedy strategy works.

However, for other denominations, this approach doesn't necessarily work. A dynamic programming solution, however, guarantees an optimal solution. Optimal here means that we are *minimizing* the number of coins used to make L change.

Let $C_{i,j}$ be defined as the minimum number of coins from denominations $\{c_1, \ldots, c_i\}$ that add up to j for $1 \leq i \leq n$ and $0 \leq j \leq L$. To build this array, we require two base cases.

- $C_{i,0} = 0$ for all i (no coins needed for a total of zero)
- $C_{0,j} = \infty$ for $j \neq 0$ (change is not possible with no coins)

Now, we can define a recurrence based on the intuitive subproblem split:

$$C_{i,j} = \begin{cases} C_{i-1,j} & \text{for } j < c_i \\ \min\{C_{i-1,j}, \quad C_{i,j-c_i} + 1\} & \text{for } j \geq c_i \end{cases}$$

The corresponding tableau can be found in Table 9.9.

The tableau is filled row-by-row (left-to-right) from top-to-bottom. The final solution (the optimal number of coins) is found in entry $C_{n,L}$.

9.5.1 Example

Suppose that we have a set of four coin denominations (see Table 9.10).

The resulting tableau can be found in Table 9.11.

$i \backslash j$	0	1	2	$\cdots$	C
0	∞	∞	∞	$\cdots$	∞
1	0	$\ddots$			
2	0				
$\vdots$	$\vdots$				
n	0				

Table 9.9: Tableau for the Coin Change Problem

Denomination	Value
c_1	5
c_2	2
c_3	4
c_4	1

Table 9.10: Coin Change Example Input

9.6 Matrix Chain Multiplication

Suppose we have three matrices, A, B, C of dimensions $(5 \times 10), (10 \times 15), (15 \times 10)$ respectively. If we were to perform the multiplication ABC, we could do it one of two ways; multiply AB first then by C or BC first then by A. That is, either

$$(AB)C$$

or

$$A(BC)$$

Using straightforward matrix multiplication, the first way would result in

$$(5 \cdot 10 \cdot 15) + (5 \cdot 15 \cdot 10) = 1,500$$

multiplications while the second would result in

$$(5 \cdot 10 \cdot 10) + (10 \cdot 15 \cdot 10) = 2,000$$

multiplications.

In general, suppose that we are given a *chain* of matrices $A_1, \ldots A_n$ and we wanted to multiply them all but minimize the total number of multiplications. This is the *matrix chain multiplication problem.* More generally, what parenthesization of the associative operations minimizes the overall number of multiplication made to compute the product?

Clearly, each product, $A_i A_{i+1}$ has to be a valid operation. Further, we will assume that each matrix has different dimensions $(d_1, \ldots, d_{n+1})$—if they were all square there would be no point in finding the optimal order (they would all be the same).

c_i	i \ j	0	1	2	3	4	5	6	7	8
-	0	-	∞	∞	∞	∞	∞	∞	∞	∞
5	1	0	∞	∞	∞	∞	1	∞	∞	∞
2	2	0	∞	1	∞	2	1	3	2	4
4	3	0	∞	1	∞	1	1	2	2	2
1	4	0	1	1	2	1	1	2	2	**2**

Table 9.11: Coin Change Example Results

i \ j	1	2	$\cdots$	n
1	0			
2	ϕ	0		
$\vdots$	$\vdots$		$\ddots$	
n	ϕ	ϕ	$\cdots$	0

Table 9.12: Matrix Chain Multiplication Tableau.

For this problem we define a (lower-triangular) table of size $n \times n$. Each entry represents the minimum number of multiplications it takes to evaluate the matrices from $A_i \ldots A_j$. The tableau can be filled in using the following recurrence relation:

$$C(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{C(i, k) + C(k+1, j) + d_{i-1}d_k d_j\} & \text{otherwise} \end{cases}$$

The order in which the tableau is filled in begins with the base cases, down the diagonal where $i = j$. Depending on how you orient your tableau, you will then work downward in a diagonal fashion, see Table 9.12.

In addition, at each calculation, we fill in an S-array that keeps track of which value of k was chosen for each entry. This value can be used to construct the optimal solution. The S-array ranges $2 \leq i \leq n$ and $1 \leq j \leq n - 1$; see Table 9.13.

i \ j	1	2	$\cdots$	$n-1$
2	0			
3	ϕ	0		
$\vdots$	$\vdots$		$\ddots$	
n	ϕ	ϕ	$\cdots$	0

Table 9.13: Matrix Chain Multiplication S-array.

Matrix	Dimensions
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Table 9.14: Example Matrix Chain Multiplication Input

```
INPUT  : $n$ matrices, $A_1, \ldots, A_n$ with dimensions $d_0, \ldots, d_{n+1}$
OUTPUT : The number of multiplications in the optimal multiplication order
FOR $i = 1, \ldots, n$ DO
    $C_{i,j} \leftarrow 0$
    $S_{i+1,j} \leftarrow i$
END
FOR $l = 2, \ldots, n$ DO
    FOR $i = 1, \ldots, n - l + 1$ DO
        $j \leftarrow i + l - 1$
        $C_{i,j} \leftarrow \infty$
        FOR $k = i, \ldots, (j - 1)$ DO
            $q \leftarrow C_{i,k} + C_{k+1,j} + d_{i-1}d_kd_j$
            IF $q < C_{i,j}$ THEN
                $C_{i,j} \leftarrow q$
                $S_{i,j} \leftarrow k$
            END
        END
    END
END
output $C_{1,n}, S$
```

Algorithm 58: Optimal Matrix Chain Multiplication

9.6.1 Example

Consider the following 6 matrices:

The final tableau can be found in Table 9.15.

For an example of the computation, consider the index at $C_{2,5}$:

i j	1	2	3	4	5	6
1	0	15,750	7,875	9,375	11,875	15,125
2	ϕ	0	2,625	4,375	7,125	10,500
3	ϕ	ϕ	0	750	2,500	5,375
4	ϕ	ϕ	ϕ	0	1,000	3,500
5	ϕ	ϕ	ϕ	ϕ	0	5,000
6	ϕ	ϕ	ϕ	ϕ	ϕ	0

Table 9.15: Final Tableau for the Matrix Chain Example

i j	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

Table 9.16: Final S-array for the Matrix Chain Example

$$C_{2,5} = \min_{2 \leq k \leq 4} \begin{cases} C_{2,2} + C_{3,5} + d_1 d_2 d_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13,000 \\ C_{2,3} + C_{4,5} + d_1 d_3 d_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7,125 \\ C_{2,4} + C_{5,5} + d_1 d_4 d_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11,375 \end{cases}$$

Clearly, the minimum among these is 7,125. The k value that gave this result was 3, consequently we place this value into our S-array, $S_{2,5} = 3$. The full S-array can be found in Table 9.16.

Each entry in the S-array records the value of k such that the optimal parenthesization of $A_i A_{i+1} A_{i+2} \ldots A_j$ splits the product between A_k and A_{k+1}. Thus we know that the final matrix multiplication in computing $A_{1 \ldots n}$ optimally is $A_{1 \ldots s[1,n]} A_{s[1,n]+1 \ldots n}$. Multiplications in each sub-chain can be calculated recursively; the intuitive base case is when you only have two matrices to multiply.

From the S-array above, the initial split would be at $S_{1,6} = 3$. The full parenthesization would be:

$$((A_1)(A_2 A_3))\,((A_4 A_5)(A_6))$$

9.7 Exercises

Exercise 9.1. Construct the C tableau, S-array and final parenthesization for the following matrices:

Matrix	Dimensions
A_1	5×10
A_2	10×3
A_3	3×12
A_4	12×5
A_5	5×50
A_6	50×6

Table 9.17: Matrix Chain Multiplication Exercise

Exercise 9.2. Gomer thinks he's found a better solution to constructing an Optimal Binary Search Tree. He thinks a greedy strategy would work just as well: Sort the keys and choose the one with the highest probability as the root of the tree. Then, repeat this process with the left/right keys. Show that Gomer's strategy will not work by providing an example input that will not be optimal with his strategy.

Exercise 9.3. Implement the dynamic programming solution to the Optimal Binary Search Tree problem in the high-level programming language of your choice.

Exercise 9.4. Implement the dynamic programming solution to the Matrix Chain Multiplication problem in the high-level programming language of your choice.

Exercise 9.5. Implement the dynamic programming solution to the 0-1 knapsack problem in the high-level programming language of your choice.

Exercise 9.6. Implement the dynamic programming solution to the Coin Change problem in the high-level programming language of your choice.

Exercise 9.7. Consider the following "game": n items of values $v_1, v_2, \ldots, v_n$ are placed in a line. You are allowed to take an item only at the beginning or the end of the line. Once you take an item, the second player is allowed to take an item at the beginning or the end of the line. You take turns like this until all items have been taken. The winner is the player whose item values sum to the greater amount.

Prove or disprove: a greedy-choice strategy will work; that is, always take the item with the higher value.

Exercise 9.8. Recall that a *palindrome* is a string that is the same backwards and forwards ("abba" or "deified").

(a) Write pseudocode that, given a string s, determines the longest *contiguous* palindromic subsequence in s. Analyze your algorithm.

(b) A palindromic subsequence may not necessarily be contiguous. For example,

$$BBABCBCAB$$

has palindromic subsequences “BBBBB”, “BBCBB” and “BABCBAB” among others, but are made from non-contiguous elements. Devise a dynamic programming solution to find the length of the longest palindromic subsequence of a given string s. In addition to finding the length, devise a scheme to output a palindromic subsequence of maximal length. Give pseudocode and analyze your solution. Hint: consider defining a tableau P such that $P_{i,j}$ represents the length of the maximal palindromic subsequence within the substring $s_i, \ldots, s_j$ (where $s = s_1 s_2 \ldots s_n$).

(c) Implement your dynamic programming solution in in the high-level programming language of your choice.

10 Computational Models

10.1 Introduction

At the foundation of computer science, we have questions like:

- What are the minimum amount of resources required to solve a given problem?
- Given enough time or memory, can an algorithm be designed to solve *any* problem?
- Can algorithms solve any problem?

These notes will take a breadth approach to computability and complexity touching on the following topics.

- Language Theory
- Computational Models
- Turing Machines
- Computability Theory
- Complexity Classes P & NP
- NP-Completeness
- Reductions

We will first establish a few fundamentals. As a motivation, consider the following statement:

"Algorithms solve problems"

each part of this statement is imprecise from a mathematical point of view; each has an intuitive and dictionary meaning, but we will want to give a more rigorous, quantifiable mathematical equivalent. In particular,

"Turing Machines decide languages"

10.1.1 Languages

We've looked at several different types of *problems*:

- Numerical
- Graph
- Sorting
- Logic

We need a *unified* model that captures all different types of problems; for this we use *languages.*

Definition 14.
- An *alphabet*, Σ is a finite, nonempty set of symbols. Examples: [A-Za-z], [0–9], $\{0, 1\}$.
- A *string* (or *word*) over Σ is a finite combination of symbols from Σ. Example: any integer is a string over $\Sigma = [0 - 9]$.
- The *empty string*, denoted ϵ (sometimes λ) is a string containing no symbols from Σ.
- The set of all finite strings over Σ is denoted Σ^*.
- A *language* over Σ is a set (finite or infinite), $L \subseteq \Sigma^*$.

It suffices to consider a binary alphabet $\Sigma = \{0, 1\}$

- Often for convenience we may include a delimiter in our alphabet
- Any alphabet can be converted to a binary alphabet using a reasonable encoding
- Example: a prefix-free encoding generated by a Huffman code

Language Operations

Operations can be performed on languages:

- **Union** – $L_1 \cup L_2$
- **Concatenation** – $L_1 \circ \mathsf{L}_2$ (or just $L_1 L_2$)
- **Kleene Star** – L^*
- Many others

The *concatenation* of two languages is the concatenation of all strings in each language.

$$L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

Examples: Let $L_1 = \{0, 10\}, L_2 = \{1, 01\}$. Then

$$L_1 L_2 = \{01, 001, 101, 1001\}$$

and

$$L_1 L_1 = \{00, 010, 100, 1010\}$$

The Kleene Star is a recursively defined operation. For a given language L, $L^0 = \{\lambda\}$ and for $n > 0$, we define

$$L^n = L^{(n-1)} L$$

For example, let $L = \{0, 10\}$ then,

$$\begin{array}{lcl} L^0 & = & \{\lambda\} \\ L^1 & = & \{0, 10\} \\ L^2 & = & \{00, 010, 100, 1010\} \\ L^3 & = & \{000, 0010, 0100, 01010, 1000, 10010, 10100, 101010\} \end{array}$$

The Kleene Star operation is then defined as the union of all such concatenations.

$$L^* = \bigcup_{n \geq 0} L^n$$

For the alphabet Σ itself, Σ^* is the set of all binary strings.

Sometimes it is useful to use the following notation to consider only *nonempty* strings.

$$L^+ = \bigcup_{n \geq 1} L^n = L^* - \{\lambda\}$$

Regular Languages

We say that R is a *regular expression* if

- $R = b$ for some bit $b \in \Sigma$
- $R = \lambda$
- $R = \emptyset$
- $R = (R_1 \cup R_2)$ where R_1, R_2 are regular expressions.
- $R = (R_1 \circ R_2)$ where R_1, R_2 are regular expressions.
- $R = (R_1^*)$ where R_1 is a regular expression.

Regular expressions are used in `grep`, `sed`, `vi`, Java, Perl, and most other scripting languages.

Regular languages are those that can be generated by a regular expression.

Examples:

- $0^* \cup 1^*$ is the language consisting of all strings with *either* all `1`s or all `0`s (plus the empty string).
- 0^*10^* is the language consisting of all strings with a single 1 in them.
- $(\Sigma\Sigma)^*$ the set of all even length strings
- $1\Sigma^*0$ the set of all canonical representation of even integers.

Exercise: Give a regular expression for the set of all strings where every `0` appears *before* any occurrence of a `1`.

Languages are Equivalent to Problems

An *instance* of a *decision problem* involves a given configuration of data.

An algorithm answers

- *yes* if the data conforms to or has some property, and
- *no* if it does not.

Though many natural problems (optimization, functional) are not decision problems, we can usually formulate the *decision version* of them.

An optimization problem of the form "what is the maximum (minimum) number of x such that property $\mathcal{P}$ holds?" can be reformulated as, "Does property $\mathcal{P}$ hold for all $x \geq k$?"

Languages are are equivalent to problems: given a problem, you can define a language that represents that problem.

Problem 14 (Sorting). **Given** elements $x_0, \ldots, x_{n-1}$ (properly *encoded*) and an ordering $\preceq$.

Question: is $x_i \preceq x_{i+1}$ for $0 \leq i \leq n-2$?

The language model is *robust.* Any problem $\mathcal{P}$ can be equivalently stated as a language L where

- (Encodings) x of *yes* instances are members of the language; $x \in L$.
- (Encodings) x of *no* instances are not members of the language; $x \notin L$.

The key is that we establish a proper *encoding* scheme.

A proper encoding of graphs, for example, may be a string that consists of a binary representation of n, the number of vertices.

Using some delimiter (which can also be in binary), we can specify connectivity by listing pairs of connected vertices.

$$\langle G \rangle = \texttt{11:00:01:01:10}$$

We can then define a language,

$$L = \{\langle G \rangle \mid G \text{ is a connected graph}\}$$

Graph connectivity is now a language problem;

- $\langle G \rangle \in L$ if G is a (properly encoded) graph that is connected.
- $\langle G \rangle \notin L$ if G is not connected.

Instead of asking if a given graph G is connected, we instead ask, is $\langle G \rangle \in L$?

10.2 Computational Models

There are many different computational models corresponding to many *classes* of languages.

Some are provably more powerful than others. Here, we give a brief introduction to

- Finite State Automata
- Grammars
- Turing Machines

10.2.1 Finite-State Automata

Definition 15. A *finite automaton* is a 5-tuple, $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a nonempty finite set of *states*
- Σ is our alphabet
- $\delta : Q \times \Sigma \to Q$ is the *transition function*
- $q_0 \in Q$ is an *initial state*
- $F \subseteq Q$ is the set of *accept states*

An example of an FSM can be found in Figure 10.1.

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- q_0 is our initial state
- $F = \{q_2\}$

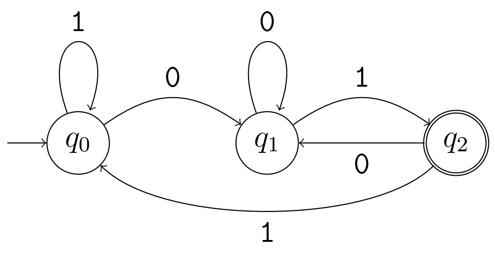


Figure 10.1: A Finite State Automaton.

The transition function is specified by the labeled arrows.

$$\begin{aligned}
\delta(q_0, 0) &= q_1 \\
\delta(q_0, 1) &= q_0 \\
\delta(q_1, 0) &= q_1 \\
\delta(q_1, 1) &= q_2 \\
\delta(q_2, 0) &= q_1 \\
\delta(q_2, 1) &= q_0
\end{aligned}$$

This FSM *accepts* is any string that ends in 01. An equivalent regular expression is simply Σ^*01.

The set of strings that a finite-state automaton A accepts is its *language*:

$$L(\mathcal{A}) = \{x \in \Sigma^* \mid A(x) \text{ accepts}\}$$

Conversely, any string that ends in a non-accept state is *rejected.* This also defines a language–the *complement* language:

$$\overline{L(\mathcal{M})} = \Sigma^* - L(\mathcal{M})$$

Finite-state automata are a simple computation model, but very restrictive. The only types of languages it can *recognize* are those that can be defined by regular expressions.

Theorem 7. *Finite-State Languages = Regular Languages = Regular Expressions*

Recognition here means that a machine, given a finite string $x \in \Sigma^*$ can tell if $x \in L(\mathcal{M})$.

Examples of regular languages:

- The language containing all strings, Σ^*
- The language consisting of all strings that are all 0s or all 1s
- The language consisting of all strings with an equal number of 0s and 1s

$$L = \{w \in \Sigma^* \mid w \text{ has an equal number of } \texttt{0}\text{s and } \texttt{1}\text{s}\}$$

- The language consisting of all even parity strings (an even number of 1s)

Not all languages are regular. Even simple languages such as

$$L = \{0^n 1^n \mid \forall n \geq 0\}$$

10.2.2 Turing Machines

A more general computational model is a *Turing Machines*.

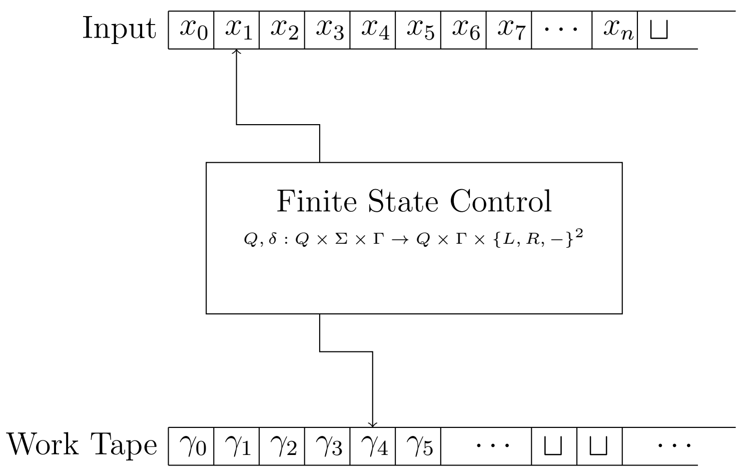
Definition 16. A *Turing Machine* is a 7-tuple, $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where

- Q is a nonempty, finite set of states
- Σ is the input alphabet
- Γ is the *work tape* alphabet
- $\delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma \times \{L, R, -\}^2$ is the transition function
- $q_0 \in Q$ is the initial state
- q_{accept} is the accept state
- q_{reject} is the reject state

A *Turing Machine* is a basic computational model which has an input tape that can be read from, an output tape that can be written on and a set of states.

- A tape head moves left and right along the input/output tape and performs reads and writes according to what symbols it encounters. A special *empty symbol*, $\sqcup$ is used to indicate the end of the input in the input tape and unwritten/uninitialized cells in the work tape.
- A definition of a given Turing Machine can be made precise by enumerating every possible transition on every possible input and output symbol for every state.
- A state diagram similar to automatons can visualize this transition. However, it is much easier to simply *describe* a Turing Machine in high level English.
- A visualization can be seen in Figure 10.2
- In our definition:
  - There are separate input and work tapes; the input tape is read-only and cannot be changed/written to
  - We allow left/right transitions for both tapes *as well as* a “no transition”, $-$

  Other definitions (such as those in [14]) may omit these conveniences, but it can be shown to be equivalent



Input
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 $\cdots$ x_n $\sqcup$
Finite State Control
$Q, \delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma \times \{L, R, -\}^2$
Work Tape
γ_0 γ_1 γ_2 γ_3 γ_4 γ_5 $\cdots$ $\sqcup$ $\sqcup$ $\cdots$

Figure 10.2: Visualization of a Turing Machine

There are many variations of Turing Machines:

- Multi-tape
- Multi-head
- Multi-work tape
- Randomized
- Random access

However, it can be shown that, from a computability point of view, all Turing machines are equivalent.

Example: consider the following language:

$$L = \{x\#x \mid x \in \Sigma^*\}$$

The following high-level description of a Turing Machine *decides* this language.

$\mathcal{M}(x)$ (read: on input x)

1. Scan the input to be sure that it contains a single #, if not *reject.*
2. Zig Zag across the tape to corresponding positions on each side of #. If symbols do not match, *reject*, otherwise cross them off (write a blank symbol, $\sqcup$) and continue.
3. After all symbols to the left of # have been crossed off, check to the right of #, if any symbols remain, *reject* otherwise, *accept.*

Extended Example

Designing Turing Machines can be a very complex process (and more arduous than programming with 1950s era punchcards). We're essentially programming using machine language.

As an example, let's design a full Turing Machine to decide the non-regular language, $\{0^n 1^n \mid \forall n \geq 1\}$. Let's start with a high-level description:

1. Start by writing a special end symbol, # to the work tape. Moreover, if we see a 1 in the input tape or it is empty, $\sqcup$ immediately halt and reject.
2. Start reading 0s in the input tape, for each 0, write a zero to the worktape (the work tape will act as a stack that we push symbols onto).
3. When the first 1 is encountered, continue reading 1s; for each 1, erase the work tape symbols.
    - If we ever encounter another 0, reject
    - If we encounter # with 1s still in the input tape, reject
    - If we encounter the end of the input but there are still 0s on the work tape, reject
    - Otherwise, accept.

The transition function is defined by in Table 10.1 and a visual depiction of the finite state control is depicted in Figure 10.3. This Turing Machine construction is actually equivalent to a push-down automaton (essentially a finite state machine with access to an infinite-capacity stack), which define *Context-Free Languages*.

10.2.3 Church-Turing Thesis

The *Church-Turing Thesis* gives a formal (though debatably not rigorous) definition of what an *algorithm* is. It states that the intuitive notion of an algorithmic process is equivalent to the computational model of Turing Machines.

This means that *any* rigorous computational model can be simulated by a Turing Machine. Moreover, *no* other computational model is *more* powerful (in terms of the *types* of languages it can accept) than a Turing Machine.

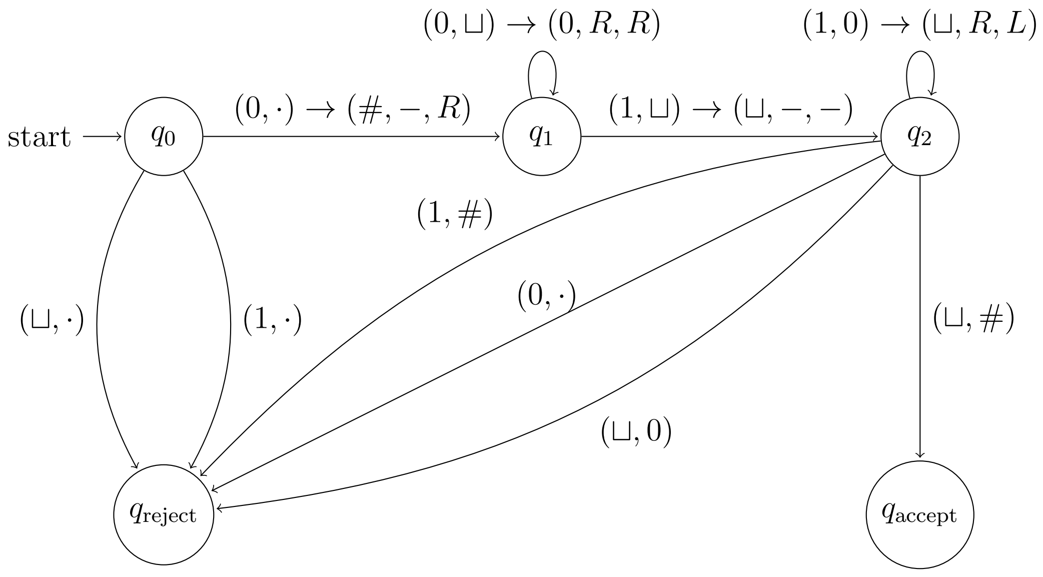
A programming language is *Turing complete* if it can do anything that a Turing machine can do.

As a consequence, any two Turing complete programming languages are equivalent.

Intuitively, anything that you can do in Java, you can do in `C++` (algorithmically, we're not talking about specific libraries), Perl, Python, PHP, etc.

Table 10.1: Turing Machine Transitions. Some states are not possible (NA).

State	**Input**	**Output**	**Transition**
q_0	0	0	NA
		1	NA
		#	NA
		$\sqcup$	$q_1, \#, -, R$
	1	0	NA
		1	NA
		#	NA
		$\sqcup$	reject
	$\sqcup$	0	NA
		1	NA
		#	NA
		$\sqcup$	reject
q_1	0	0	NA
		1	NA
		#	NA
		$\sqcup$	$q_1, 0, R, R$
	1	0	NA
		1	NA
		#	NA
		$\sqcup$	$q_2, \sqcup, -, -$
	$\sqcup$	0	NA
		1	NA
		#	NA
		$\sqcup$	reject
q_2	0	0	NA
		1	NA
		#	reject
		$\sqcup$	reject
	1	0	NA
		1	NA
		#	reject
		$\sqcup$	$q_1, \sqcup, R, L$
	$\sqcup$	0	reject
		1	NA
		#	accept
		$\sqcup$	NA



$(0, \sqcup) \rightarrow (0, R, R)$
$(1, 0) \rightarrow (\sqcup, R, L)$
start
q_0
$(0, \cdot) \rightarrow (\#, -, R)$
q_1
$(1, \sqcup) \rightarrow (\sqcup, -, -)$
q_2
$(1, \#)$
$(0, \cdot)$
$(\sqcup, \cdot)$
$(1, \cdot)$
$(\sqcup, \#)$
$(\sqcup, 0)$
q_{reject}
q_{accept}

Figure 10.3: Turing Machine Finite State Transitions. Invalid or unnecessary transitions have been omitted. The transition $(a, b) \rightarrow (x, y, z)$ indicates a is the symbol on the input tape, b is the symbol on the work tape, x is the symbol we write to the work tape, and y, z are the tape head transitions for the input tape and work tape respectively. We have used the notation $\cdot$ as in $(0, \cdot)$ to indicate that the symbol on the work tape is irrelevant. Transitions to a final halting state omit the tape write/transitions as they are final states.

The statement is a *thesis* since an algorithm is a dictionary definition, not a mathematical definition. There are other notions of computability (Lambda calculus) that do have mathematical definitions can can be *proven* to be equivalent to Turing Machines.

As with many programs, a Turing Machine may not always terminate or *halt.* On some inputs, it may get caught in an infinite loop. We require a machine to halt in order to accept (or reject) an input.

Definition 17. Let M be a Turing Machine and let L be a language. We say that M *decides* L if for all $x \in \Sigma^*$, $M(s)$ halts and:

- accepts if and only if $x \in L$
- rejects if and only if $x \notin L$

The Church-Turing thesis can then be stated as follows.

$$\text{“There exists a Turing Machine } \mathcal{M} \text{ that } \textit{decides} \text{ a language } L\text{”}$$

$$=$$

$$\text{“There exists an Algorithm } \mathcal{A} \text{ that solves a problem } \mathcal{P}\text{”}$$

10.2.4 Halting Problem & Decidability

As previously noted, a machine M on an input x may not halt. A less restrictive property is language *recognition.*

We say that a Turing machine M *recognizes* a language L if for every $x \in L$, $M(x)$ halts and accepts.

A language L is in RE if some Turing machine recognizes it.

RE is the *class of recursively enumerable languages* (also called *computably enumerable*).

For a language in $L \in \mathsf{RE}$, if $x \in L$, then some machine will *eventually* halt and accept it.

If $x \notin L$ then the machine may or may not halt.

A language L is in R if some Turing machine decides it.

R is the *class of recursive languages* (also called *computable*).

Here, if $L \in \mathsf{R}$, then there is some machine that will *halt* on all inputs and is guaranteed to accept or reject.

Its not hard to see that if a language is decidable, it is also recognizable by definition, thus

$$\mathsf{R} \subseteq \mathsf{RE}$$

There are problems (languages) that are *not* Turing Decidable: languages $L \in \mathsf{RE}, L \notin \mathsf{R}$.

We take as our first example the *halting problem*.

Problem 15 (Halting Problem)**. Given:** A Turing Machine M and an input x.
Question: does $M(x)$ halt?

This indeed would be a very useful program—once you've compiled a program, you may want to determine if you've screwed up and caused an infinite loop somewhere.

We will show that the halting problem is undecidable.

That is, *no algorithm, program or Turing Machine exists* that could ever tell if another Turing Machine halts on a given input or not.

Proof. By way of contradiction assume that there exists a Turing Machine $\mathcal{H}$ that decides the halting problem:

$$\mathcal{H}(\langle M, x\rangle) = \begin{cases} \text{halt and output 1} & \text{if } M \text{ halts on } x \\ \text{halt and output 0} & \text{if } M \text{ does not halt on } x \end{cases}$$

We now consider $\mathcal{P}$ as an input *to itself.*

In case you may think this is invalid, it happens all the time. A text editor may open itself up, allowing you to look at its binary code. The compiler for `C` was itself written in `C` and may be called on to compile itself. An *emulator* opens machine code intended for another machine and simulates that machine.

From the encoding $\langle M, M\rangle$ we construct another Turing Machine, $\mathcal{Q}$ as follows:

$$\mathcal{Q}(\langle M\rangle) = \begin{cases} \text{halts if} & \mathcal{H}(\langle M, M\rangle) = 0 \\ \text{does not halt if} & \mathcal{H}(\langle M, M\rangle) = 1 \end{cases}$$

It is easy to construct $\mathcal{Q}$: we run $\mathcal{H}(\langle M, M\rangle)$, if it outputs 0, then we halt and accept; if it outputs 1 then we go into a trivial loop state, never halting. Now that $\mathcal{Q}$ is constructed, we can run $\mathcal{Q}$ on itself:

$$\mathcal{Q}(\langle \mathcal{Q}\rangle) = \begin{cases} \text{halts if} & \mathcal{H}(\langle \mathcal{Q}, \mathcal{Q}\rangle) = 0 \\ \text{does not halt if} & \mathcal{H}(\langle \mathcal{Q}, \mathcal{Q}\rangle) = 1 \end{cases}$$

Which is a contradiction because $\mathcal{Q}(\langle \mathcal{Q}\rangle)$ will halt if and only if $\mathcal{Q}(\langle \mathcal{Q}\rangle)$ doesn't halt and vice versa.

Therefore, no such $\mathcal{H}$ can exist. □

Many other problems, some of even practical interest have been shown to be undecidable. This means that no matter how hard you try, you can *never* solve these problems with *any* algorithm.

- Hilbert's 10th problem: Given a multivariate polynomial, does it have *integral roots*?
- Post's Correspondence Problem: Given a set of "dominos" in which the top has a finite string and the bottom has another finite string, can you produce a sequence of dominos that is a *match*—where the top sequence is the same as the bottom?
- Rice's Theorem: In general, *given* a Turing Machine, M, answering any question about any non-trivial property of the language which it defines, $L(M)$ is undecidable.

To show that a problem is undecidable, you need to show a *reduction* to the halting problem (or to any other problem that is known to be undecidable). That is, given a problem $\mathcal{P}$ that you wish to show undecidable, you proceed by contradiction:

1. Assume that $\mathcal{P}$ is decidable by a Turing Machine M.
2. Construct a machine $\mathcal{R}$ that uses M to decide the Halting Problem.
3. Contradiction – such a machine M cannot exist.

Intuitive example: we can categorize all statements into two sets: lies and truths. How then can we categorize the sentence,

I am lying

The key to this seeming paradox is *self-reference*. This is where we get the terms *recursive* and *recursively enumerable*.

10.3 Complexity Classes

Now that we have a concrete model to work from: Problems as languages and Algorithms as Turing Machines, we can further delineate complexity classes *within* R (all decidable problems) by considering Turing Machines with respect to *resource bounds*.

In the computation of a Turing Machine M, the amount of memory M uses can be quantified by how many *tape cells* are required in the computation of an input x. The amount of *time* M uses can be quantified by the number of transitions M makes in the computation of x.

Of course, just as before, we are interested in how much time and memory are used as a *function* of the input size. In this case,

$$T(|x|)$$

and

$$M(|x|)$$

respectively where $x \in \Sigma^*$. Again, the restriction to decisional versions of problems is perfectly fine—we could just consider languages and Turing Machines themselves.

10.3.1 Deterministic Polynomial Time

Definition 18. The complexity class P consists of all languages that are decidable by a Turing Machine running in polynomial time with respect to the input $|x|$. Alternatively, P is the class of all decision problems that are solvable by a polynomial time running algorithm.

10.3.2 Nondeterminism

A *nondeterministic* algorithm (or Turing Machine) is an algorithm that works in two stages:

1. It *guesses* a solution to a given instance of a problem. This set of data corresponding to an instance of a decision problem is called a *certificate*.
2. It *verifies* whether or not the guessed solution is valid or not.
3. It accepts if the certificate is a valid *witness*.

As an example, recall the HAMILTONIANCYCLE problem:

- A nondeterministic algorithm would guess a solution by forming a permutation π of each of the vertices.
- It would then verify that $(v_i, v_{i+1}) \in E$ for $0 \leq i \leq n-1$.
- It *accepts* if π is a Hamiltonian Cycle, otherwise it rejects.
- An instance is in the language if *there exists* a computation path that accepts.

Therein lies the *nondeterminism* – such an algorithm does *not* determine an actual answer.

Alternatively, a nondeterministic algorithm solves a decision problem if and only if for every *yes* instance of the problem it returns *yes* on *some* execution.

This is the same as saying that *there exists* a certificate for an instance.

A certificate can be used as a *proof* that a given instance is a *yes* instance of a decision problem. In such a case, we say that the certificate is *valid*.

If a nondeterministic algorithm produces an *invalid* certificate, it does NOT necessarily mean that the given instance is a *no* instance.

We can now define the class NP.

Definition 19. NP ("Nondeterministic Polynomial Time") is the class of all languages (problems) that can be decided (solved) by a Nondeterministic Turing Machine (algorithm) running in polynomial time with respect to the size of the input $|x|$.

That is, each stage, guessing and verification, can be done in polynomial time. HamiltonianCycle
NP since a random permutation can be guessed in $O(n)$ time and the verification process can be done in $O(n^2)$ time.

It is not hard to see that

$$\mathsf{P} \subseteq \mathsf{NP}$$

since any problem that can be deterministically solved in polynomial time can certainly be solved in nondeterministic polynomial time.

The most famous unanswered question so far then is

$$\mathsf{P} \overset{?}{=} \mathsf{NP}$$

If the answer is yes (*very unlikely*), then *every* problem in NP could be solved in polynomial time. If the answer is no, then the hardest problems in NP could *never* be solved by a polynomial time algorithm. Such problems will forever remain *intractable*.

To understand this more fully, we need to explore the notion of NP-Completeness.

10.4 Reductions & NP-Completeness

Reductions between problems establish a *relative* complexity. Reductions define a *structure* and order to problems and a hierarchy of complexity.

Definition 20. A decision problem $\mathcal{P}_1$ is said to be *polynomial time reducible* to a decision problem $\mathcal{P}_2$ if there exists a function f such that

- f maps all *yes* instances of $\mathcal{P}_1$ to *yes* instances of $\mathcal{P}_2$. *no* instances likewise.
- f is computable by a polynomial time algorithm

In such a case we write

$$\mathcal{P}_1 \leq_{\mathsf{P}} \mathcal{P}_2$$

In general, there are other notions of reductions

- Adaptive reductions
- Truth-table reductions
- Projections
- Reductions that use more or less resources (time, space) than polynomial-time

However, polynomial-time reductions suffice for problems within NP. However, we want to take care that we do not "cheat" by leveraging the power of the reduction to solve the problem. Suppose we allowed *exponential* time reductions: we could solve SAT using the reduction, then make a trivial map to another, easier problem.

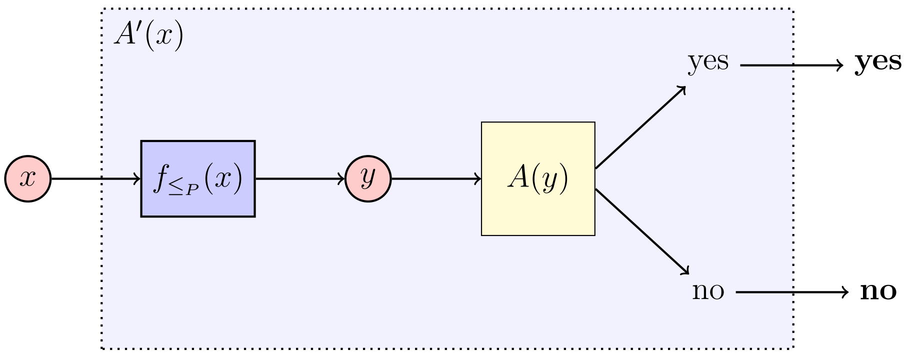


Figure 10.4: Reduction Visualization. An algorithm A' for problem X is formed by first reducing instances x into instances y of problem Y. A known algorithm A for Y is run on the reduced instance y and the output is used directly as output for A'.

An equivalent interpretation is that a language A is reducible to language B if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that

$$x \in A \iff f(x) \in B$$

A reduction establishes a relative complexity between problems as follows. Suppose that

- $P_1 \leq_{\mathsf{P}} P_2$ via a function f and
- there is an algorithm A for P_2 that runs in time $O(g(n))$
- we then have an algorithm A' for L_1:
  1. On input x, run $f(x)$ to get y
  2. Run $A(y)$ and map the answer appropriately
- This algorithm runs in $O(g(n) + n^k)$
- If $g(n) \in \Omega(n^k)$ then A' runs in $O(g(n))$
- A visualization of this interpretation is presented in Figure 10.4.

Reductions establish a *relative* complexity. If $P_1 \leq_{\mathsf{P}} P_2$ then

- P_2 is *at least* as difficult/complex as P_1.
- If we have a polynomial time solution to P_2 then we have a polynomial time solution to P_1 via its reduction

Definition 21. A problem $\mathcal{P}$ is said to be NP-*Complete* if

1. $\mathcal{P} \in \mathsf{NP}$ and
2. For every problem $\mathcal{P}' \in \mathsf{NP}$, $\mathcal{P}' \leq_{\mathsf{P}} \mathcal{P}$

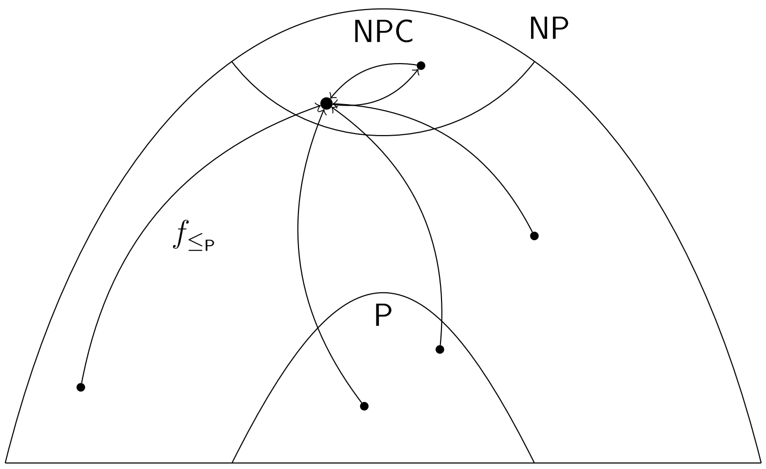


Figure 10.5: All problems in NP reduce to any NP-complete problem.

Intuitively, NP-Complete problems are the *hardest* (most difficult computationally speaking) problems in NP (of course there are *provably* harder problems in classes such as EXP). The "landscape" is depicted in Figure 10.5.

There are 5 basic steps to show that a given problem $\mathcal{P}$ is NP-Complete.

1. Prove that $\mathcal{P} \in \mathsf{NP}$ by giving an algorithm that guesses a certificate and an algorithm that verifies a solution in polynomial time.
2. Select a known NP-Complete problem $\mathcal{P}'$ that we will reduce to $\mathcal{P}$ ($\mathcal{P}' \leq_\mathsf{P} \mathcal{P}$)
3. Give an algorithm that computes $f : \mathcal{P}'_{\text{yes}} \mapsto \mathcal{P}_{\text{yes}}$ for every instance $x \in \{0,1\}^*$.
4. *Prove* that f satisfies $x \in \mathcal{P}'$ if and only if $f(x) \in \mathcal{P}$
5. Prove that the algorithm in step 3 runs in polynomial time

10.4.1 Satisfiability

To establish NP completeness we need a starting point: a "first problem" to reduce *from*. In fact, the computation of a generic NP Turing Machine is the *canonically* complete language for NP:

$$L_\mathsf{NP} = \left\{ \langle M, x \rangle \;\middle|\; \begin{array}{l} M \text{ is a non-deterministic Turing Machine that} \\ \text{accepts } x \text{ in a polynomial number of steps} \end{array} \right\}$$

However, L_NP is unnatural from a "problem" point of view. However, the key is that polynomial time reductions are *transitive*:

$$\mathcal{P}_1 \leq_{\mathsf{P}} \mathcal{P}_2 \leq_{\mathsf{P}} \mathcal{P}_3 \Rightarrow \mathcal{P}_1 \leq_{\mathsf{P}} \mathcal{P}_3$$

Thus, we need only show a reduction *from* a known NP-Complete problem to $\mathcal{P}$ to show that $\mathcal{P} \in$ NPC. In 1971, Stephen Cook [6] independently (Levin published similar concepts in 1973 [12]) defined the notions of NP and NP-Completeness showing the first NP-Complete problem ever by showing a reduction to Satisfiability, $L_{\mathsf{NP}} \leq_{\mathsf{P}} \mathsf{SAT}$

Recall the following notations:

- A *literal* is a boolean variable that can be set to 0 or 1
- $\vee$ denotes the logical *or* of 2 boolean variables
- $\wedge$ denotes the logical *and* of 2 boolean variables
- $\neg$ denotes the *negation* of a boolean variable
- A *clause* is the is the logical disjunction (*or*-ing) of a set of boolean variables. Ex: $(x_1 \vee \neg x_2 \vee x_5)$
- The *conjunction* of a collection of clauses is the logical *and* of all their values. The value is true only if *every* clause is true.

Satisfiability or simply, SAT is the following

Problem 16 (Satisfiability)**. Given:** a set of boolean variables, $\mathcal{V} = \{x_1, x_2, \ldots x_n\}$ and a set of clauses, $\mathcal{C} = \{C_1, C_2, \ldots C_m\}$.

Question: Does there exist a satisfying assignment of boolean values to each literal x_i, $1 \leq i \leq n$ such that

$$\bigwedge_{i=1}^{m} C_i = C_1 \wedge C_2 \wedge \ldots \wedge C_m = 1$$

Let $n = 4$ and consider the following conjunction:

$$\mathcal{C} = (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$$

This conjunction *is* satisfiable and is therefore a *yes* instance of SAT since if we set $x_1 = x_4 = 0$ and $x_2 = x_3 = 1$, $\mathcal{C} = 1$.

Let $n = 3$ and consider the following conjunction:

$$\begin{aligned}\mathcal{C} \quad = \quad & (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge \\ & (x_1 \vee x_3) \vee (\neg x_1 \vee \neg x_3) \wedge \\ & (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)\end{aligned}$$

This conjunction is *not* satisfiable since none of the $2^n = 8$ possible boolean assignments will ever make $\mathcal{C} = 1$.

Another way to visualize nondeterministic computation is to think of all possible computation paths being explored at once in parallel (refer back to Figure 4.2): suppose that we explored all paths in this tree and accepted if any of the leaf computations accepted. We would be able to do this in $O(n)$ nondeterministic time.

This illustrates the intuition behind the NP class. *Yes* instances are easily solved if we are lucky enough to guess a satisfying assignment. *No* instances require an exhaustive search of all possible assignments.

Theorem 8 (Cook, 1971)**.** The problem SAT is NP-Complete.

Other standard NP-complete problems:

- 3-CNF – A restricted version of SAT where each clause is a disjunction of exactly 3 literals. CNF stands for *Conjunctive Normal Form.* Note that 2-CNF $\in$ P.
- HamiltonianCycle – Determine if a given undirected graph G contains a cycle which passes through every vertex exactly once.
- TravelingSalesman – Find the least weighted cycle in a graph G that visits each vertex exactly once.
- SubsetSum – Given a collection of integers, can you form a subset $\mathcal{S}$ such that the sum of all items in $\mathcal{S}$ is exactly p.
- GraphColoring – For a given graph G, find its chromatic number $\chi(G)$ which is the smallest number of colors that are required to color the vertices of G so that no two adjacent vertices have the same color.

Some good resources on the subject can be found in:

- *Computers and Intractability – A Guide to the Theory of NP Completeness* 1979 [7]
- Annotated list of about 90 NP-Complete Problems: `http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html`

10.4.2 Satisfiability to Clique

Problem 17 (Clique)**.**
Given: An undirected graph $G = (V, E)$
Output: A subset $C \subseteq V$ of maximal cardinality such that all vertices in C are connected

A "clique" is a group that is strongly connected (a clique of friends). In terms of graphs, a clique is a *complete subgraph.*

In terms of languages we can define

$$\textsc{Clique} = \{\langle G, k\rangle \mid G \text{ is a graph with a clique of size } k\}$$

We want to prove that Clique is NP-Complete. To do this we will go by our 5 step process.

1. Clique $\in$ NP. We can nondeterministically guess k vertices from a given graph's vertex set V in $O(|V|)$ time. Further, we can check if, for each pair of vertices $v, v' \in V'$ if $(v, v') \in E$ in $O(|V|^2)$ time.
2. We select the 3-CNF-SAT problem, a known NP-Complete problem for our reduction: 3-CNF-SAT $\leq_P$ Clique.
3. We define the following function. Let $\phi = C_1 \wedge \ldots \wedge C_k$ be a 3-CNF formula. We will construct a graph G that has a clique of size k if and only if ϕ is satisfiable. For each clause $C_i = (x_1^i \vee x_2^i \vee x_3^i)$ we define vertices $v_1^i, v_2^i, v_3^i \in V$. Edges are defined such that $(v_\alpha^i, v_\beta^j) \in E$ if *both* of the following hold:
   a) If the vertices are in different clauses, i.e. $i \neq j$
   b) Their corresponding literals are *consistent*: v_α^i is not the negation of v_β^j
4. We need to show that *yes* instances of the 3-CNF function ϕ are preserved with this function. That is we want to show that ϕ is satisfiable if and only if $f(\phi) = G$ has a clique of size k (see proof below)
5. The computation of the function described in step 3 is clearly polynomial. The input size is something like $O(nk)$ so building G takes at most $O(2n^2k)$ time.

As an example, consider the following clause:

$$\mathcal{C} = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Proof. $(\Rightarrow)$: Suppose that ϕ has a satisfying assignment. This implies that each clause C_i contains at least one true literal. Remember that each literal corresponds to some vertex in G. Choosing a true literal from each clause yields a set V' of size k. To see that V' is a clique we look at our two requirements from before: v_α^i and v_β^j are consistent and both are true, thus $(v_\alpha^i, v_\beta^j) \in E$.

$(\Leftarrow)$: Suppose that G has a clique of size k. No edges in G connect vertices in the same triple corresponding to a clause so V' contains exactly one vertex per triple. Without fear of causing inconsistencies, we can assign a 1 to each literal (0 if a negation) corresponding to some vertex in each triple thus each clause is satisfied and so ϕ is satisfied.

10.4.3 Clique to Vertex Cover

Problem 18 (VertexCover).
Given: An undirected graph $G = (V, E)$
Output: A subset $V' \subseteq V$ of minimal cardinality such that each edge $e \in V$ is incident on some $v \in V'$

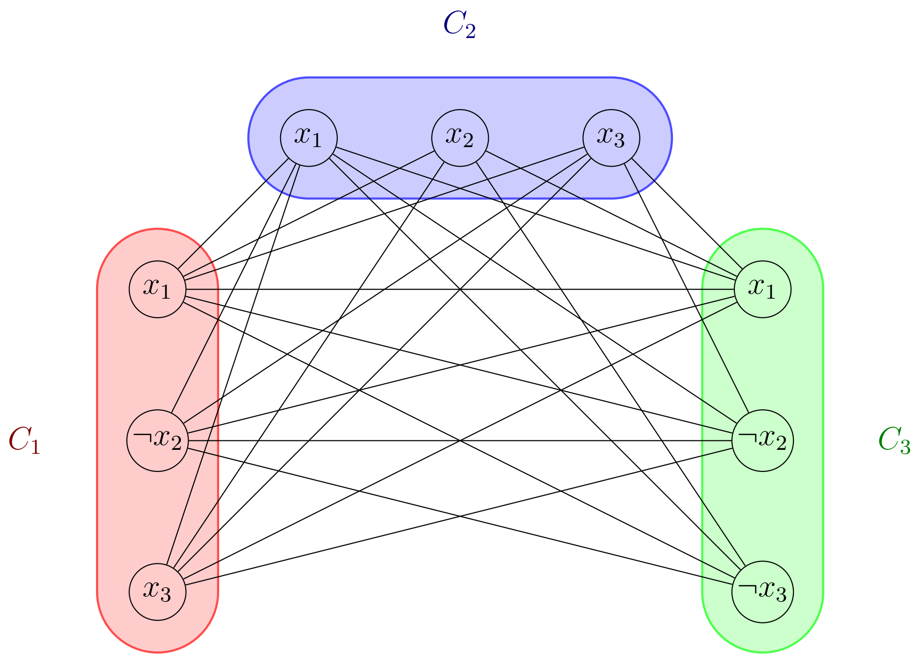


Figure 10.6: Clique Reduction Visualization

The vertices "cover" all the edges. Clearly V is a trivial vertex cover, but we are interested in vertex covers of *minimal* size.

Theorem 9. Vertex Cover is NP-complete.

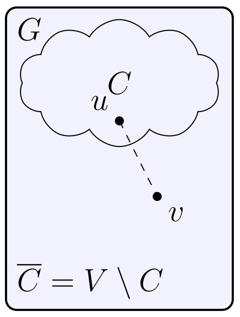
Proof. We first observe that the verification of the Vertex Cover problem can be achieved with an NP algorithm, establishing that VertexCover $\in$ NP. We first convert this problem to a decision version by adding a parameter k and ask the question, does there exist a subset $V' \subseteq V$ of cardinality k that represents a vertex cover. We then nondeterministically guess a subset $V' \subseteq V$ of size k and verify in deterministic $O(km)$ time that it constitutes a vertex cover by iterating over vertices in the cover and removing edges incident on them. If all edges have been removed, it is a vertex cover (accept) otherwise there are uncovered edges and we reject.

We complete the proof by showing a reduction from the clique problem to vertex cover. The fact that the clique problem is NP-complete was established previously. Given an encoding of the (decision version) of the clique problem, $\langle G, k\rangle$, we transform it to an instance of the vertex cover problem by taking the complement graph $\overline{G}$ and substituting $|V| - k$ for our cardinality parameter. That is:

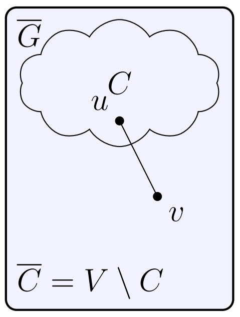
$$\langle G, k\rangle \rightarrow \langle \overline{G}, |V| - k\rangle$$

We now make the following claim:

$C \subseteq V$ is a clique of size k in G if and only if $V \setminus C$ is a vertex cover of size $|V| - k$ in $\overline{G}$. The proof is below; the idea is visualized in Figure 10.7.



(a) Suppose that a clique C of size k existed in G with u, v unconnected.



(b) In $\overline{G}$, u, v are connected and $\overline{C}$ vertex cover of size $n - k$

Figure 10.7: Reduction Idea for Clique/Vertex Cover. The complement of a clique in G acts as a vertex cover in $\overline{G}$.

($\Rightarrow$) Let C be a clique in G such that $|C| = k$. Let $e = (u, v)$ be an edge in $\overline{G}$. We need to show that this edge is covered either by u or by v. Observe:

$$\begin{aligned} e = (u,v) \in \overline{E} &\Rightarrow (u,v) \notin E \\ &\Rightarrow u \notin C \vee v \notin C \quad \text{or both} \\ &\Rightarrow u \in V \setminus C \vee v \in V \setminus C \\ &\Rightarrow (u,v) \text{ is covered by } V \setminus C \end{aligned}$$

($\Leftarrow$) Let $C \subseteq V$ be a vertex cover in $\overline{G}$ of size $n - k$. Observe that:

$$\forall u, v \in V \left[(u,v) \in \overline{E} \Rightarrow u \in C \vee v \in C\right]$$

now consider the contrapositive of this statement:

$$\forall u, v \in V \left[u \notin C \wedge v \notin C \Rightarrow (u,v) \in E\right]$$

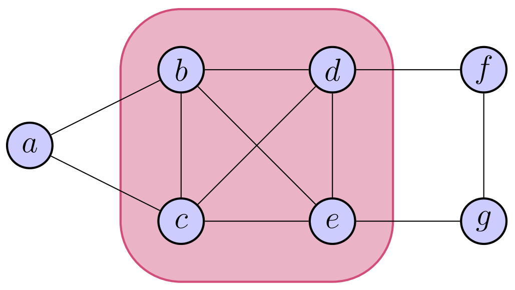
which means that $V \setminus C$ is a clique in G.

An example demonstrating this relationship can be found in Figure 10.8.

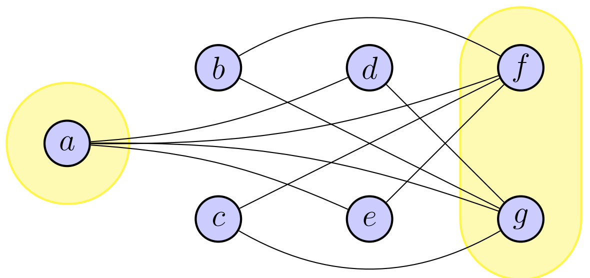
10.5 Beyond P and NP

There are *many* more complexity classes based on various computational resource bounds: memory (space); randomization, quantum, etc. A large list of complexity classes is maintained at the Complexity Zoo: `http://www.complexityzoo.com/` [2].

In summary, with respect to the complexity classes examined here,



(a) Graph G with a clique of size 4 highlighted in purple.



(b) The complement graph $\overline{G}$. The corresponding vertex cover is highlighted in yellow.

Figure 10.8: Graph and Complement: Clique and Vertex Cover. Graph G with a clique of size 4 and its complement $\overline{G}$ with a vertex cover of size 3.

- coRE is the complement class of RE, that is it consists of all decision problems for which *no* instances can be verified by a Turing Machine in a finite amount of time. *yes* instances are not guaranteed to halt.
- coNP is the complement class of NP. Rather than producing certificates, acceptance is defined as being able to produce a *disqualifier* (i.e. if some computation path produces a *no* answer, we accept. This is still a nondeterministic class. A good example: TAUTOLOGY.
- NP ∩ coNP is the intersection of NP and coNP, P is contained in this class as is
- NPI: NP intermediate, problems in NP, but that are neither in P nor NP-complete. If one were able to prove a language $L \in$ NPI, then it would show that P $\neq$ NP. Thus, no problems have been shown to be in NPI; however there are many candidates: Graph Isomorphism, Group Isomorphism, Integer Factorization, Discrete Logarithm, etc. We do have a partial result: Ladner's Theorem states that if P $\neq$ NP, then there are languages in NPI [11].

The major containments are visualized in Figure 10.9.

10.6 Misc

Definition 22. Let $G = (V, E)$ be an undirected graph. A *matching* is a set, $M \subseteq E$ of pair-wise non-adjacent edges such that $\forall v \in V$, v is the end point of at most one edge $e \in M$. A matching is *perfect* if M covers every vertex in G.

Terminology

- An *alternating path*: given a partial matching, an alternating path is a path p in G such that edges in p alternate, $e \in M, e \notin M$, etc.

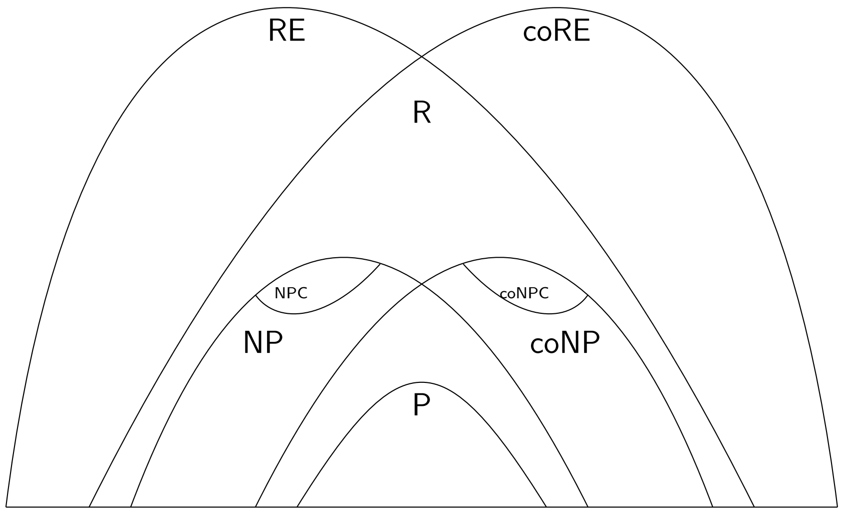


Figure 10.9: Complexity Hierarchy

TODO

Figure 10.10: Alternating and Augmenting Paths

- An *augmenting path*: An alternating path is augmenting if it begins and ends in unmatched edges ($e \notin M$)
- Examples can be found in Figure 10.10

Algorithm for matchings in bipartite graphs (Hopcroft & Karp): find an augmenting path for each $x \in L$ to R (via BFS), then switch out all the edges. Each iteration adds one more edge to the matching.

Variations & Applications:

- Maximum weighted matching for weighted graphs
- Assignment problem (bipartite matching)
- Efficiently solve (or approximate) special cases of NP-complete problems (Konig's Theorem): vertex cover in bipartite graphs

Theorem 10 (Konig's Theorem)**.** Let $G = (V, E)$ be a bipartite graph. Then any maximal matching M is also a minimum vertex cover V.

Other Algorithms:

- Edmonds (for general graphs): paths, trees and flowers (graph decomposition)
- Hungarian algorithm (for the assignment problem–equivalent to maximal bipartite matching)

Related: counting the number of perfect matchings in a bipartite graph is #P-complete (a much higher complexity class). This is equivalent to computing the *permanent* of a 0-1 matrix. Here we have a difficult counting problem whose corresponding search problem (finding a matching) is easy!

10.7 Exercises

Exercise 10.1. Design a finite-state automaton to accept the language consisting of any string in which contain no contiguous 0s.

Exercise 10.2. Can you show that polynomial time reductions, $\leq_{\mathsf{P}}$ define an equivalence relation on all languages in NP? What would the implications of such a result be?

Exercise 10.3. Let $G = (V, E)$ be an undirected graph. Given two vertices, $u, v \in V$ it is easy to determine the length of the shortest path between them using Dijkstra's algorithm. However, determining the *longest* path between u, v is NP-complete. Prove this as follows: assume that a "free" algorithm $\mathcal{A}$ exists for the longest path problem. $\mathcal{A}$ takes as input, G, u, v and outputs the length k of the longest path between u and v) and use it to design a polynomial time algorithm to solve the Hamiltonian Path problem, a known NP-complete problem.

Exercise 10.4. Show that the Hamiltonian Cycle problem is NP-complete by showing a reduction from Hamiltonian Path (hint: try adding a vertex and connecting it somehow).

Exercise 10.5. Say that we had an *oracle* (a subroutine; a "free" function) to answer yes or no whether a given graph G has a Hamiltonian Path. Design an algorithm to actually compute a Hamiltonian Path that uses this as a subroutine that runs in polynomial time (assuming that the oracle subroutine is free).

Exercise 10.6. Consider the following definition and problem:

Definition 23. An *independent set* of an undirected graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that for any two vertices $v, v' \in V'$, $(v, v') \notin E$

Problem 19 (IndependentSet)**. Given** an undirected graph $G = (V, E)$
Question: Does there exist an independent set of size k?

Prove that IndependentSet is NP-complete by showing a reduction from the clique problem.

Exercise 10.7. Show that the Clique problem is NP-complete by showing a reduction from the Independent set problem.

Exercise 10.8. Recall the *Traveling Salesperson Problem*: prove that this problem is NP-complete by showing a reduction from the Hamiltonian Cycle problem.

Exercise 10.9. Answer the following questions regarding P and NP. Here, $\leq_{\mathsf{P}}$ refers to a polynomial time reduction and P_1, P_2, P_3 are problems. Provide a brief justification for your answers.

1. Say that $P_1 \leq_{\mathsf{P}} P_2$ and that P_2 is NP-complete. Can we conclude that P_1 is NP-complete? Why or why not?
2. Say that $P_1 \leq_{\mathsf{P}} P_2$ and that P_2 is in NP. Can we conclude that P_2 is NP-complete? Why or why not?
3. Say that $P_1 \in \mathsf{P}$ and P_2 is NP-complete. Can we conclude that $P_1 \leq_{\mathsf{P}} P_2$?
4. Say that $P_1 \leq_{\mathsf{P}} P_2$. Can we conclude that P_2 is a *harder* problem than P_1?
5. Say that $P_1 \leq_{\mathsf{P}} P_2$ and that P_2 is NP-Complete. Can we conclude that $P_1 \in \mathsf{NP}$?
6. Let P_1 be a problem, is it true that $P_1 \leq_{\mathsf{P}} P_1$?

11 More Algorithms

11.1 A^* Search

11.2 Jump Point Search

http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/viewFile/5396/5212

https://en.wikipedia.org/wiki/Jump_point_search

11.3 Minimax

Glossary

algorithm a process or method that consists of a specified step-by-step set of operations.

Acronyms

BFS	Breadth First Search. 128
DAG	Directed Acyclic Graph. 147
DFS	Depth First Search. 121

Index

Bibliography

[1] An annotated list of selected NP-complete problems. http://cgi.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html, 2014. Accessed: 2014-08-12.

[2] Complexity zoo. https://complexityzoo.uwaterloo.ca/Complexity_Zoo, 2014. Accessed: 2014-08-05.

[3] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Ann. of Math*, 2:781–793, 2002.

[4] Paul Bachmann. *Die Analytische Zahlentheorie.* 1894.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[8] G. Georgy Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences (in Russian)*, number 146, pages 263–266, 1962.

[9] D.E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.

[10] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, April 1976.

[11] Richard E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, January 1975.

[12] Leonid A Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[13] Anany V. Levitin. *Introduction to the Design and Analysis of Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] Michael Sipser. *Introduction to the Theory of Computation.* International Thomson

Publishing, 1st edition, 1996.

[15] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 887–898, New York, NY, USA, 2012. ACM.