Algorithms & Algorithm Analysis

Computer Science & Engineering 235: Discrete Mathematics

Christopher M. Bourke cbourke@cse.unl.edu

An *algorithm* is a sequence of unambiguous instructions for solving

Correct – always gives a "correct" solution.

Finite – must eventually terminate.

Algorithms

Brief Introduction

Real World Objects, Entities Activities **Computing World** Data Structures, ADTs, Classes Operations, Functions, Methods

- > Problems are descriptions of objects with an objective
- ▶ Instances are problems on a specific input
- Algorithms¹ are methods or procedures that solve instances of problems

¹"Algorithm" is a distortion of *al-Khwarizmi*, a Persian mathematician

Formal Definition II

- An algorithm is a *feasible* solution to a problem if it is also efficient
- Notion of efficiency: it executes in a "reasonable" amount of time
- Alternatively: if it uses a "reasonable" amount of memory
- ▶ In general: if it uses a "reasonable" amount of some resource
- There can be multiple algorithms acting on different data structures that solve the same problem!

General Techniques I

Formal Definition I

Definition

a problem. Algorithms must be

There are many broad categories of Algorithms:

- Randomized algorithms
- Monte-Carlo algorithms
- Approximation algorithms
- Parallel algorithms
- Distributed algorithms
- And many more!

General Techniques II

General strategies of algorithms may be classified as:

- Brute Force
- Divide & Conquer
- ► Decrease & Conquer
- Transform & Conquer
- Dynamic Programming
- Greedy Techniques

Pseudo-code

Algorithms can be specified using some form of *pseudo-code Good* pseudo-code:

- Balances clarity and detail
- Abstracts the algorithm
- Makes use of good mathematical notation
- Is easy to read

Bad pseudo-code:

- Gives too many details
- ▶ Is implementation or language specific

Designing An Algorithm

A general approach to designing algorithms is as follows.

- 1. Understand the Problem
- 2. Choose an approach (exact or approximate, probable solution)
- 3. Choose an appropriate data structure
- 4. Choose a strategy
- 5. Prove Correctness
- 6. Evaluate complexity
- 7. Test it

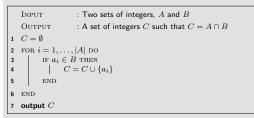
Algorithms

Example I - Algorithm

Max

Good Pseudo-code Example

INTERSECTION



Algorithms Example I

When designing an algorithm, we usually give a formal statement about the problem we wish to solve. Problem

Given a set $A = \{a_1, a_2, \ldots, a_n\}$ integers. Output the index i of the maximum integer a_i .

A straightforward idea is to simply store an initial maximum, say a_1 then compare it to every other integer, and update the stored maximum if a new maximum is ever found.

Algorithms

Example I - Understanding

This is a simple enough algorithm that you should be able to:

- Prove it correct
- Verify that it has the properties of an algorithm.
- ► Have some intuition as to its *efficiency*.

Questions to answer:

- How many "steps" would it take for this algorithm to complete?
- What constitutes a step?
- ► How do we measure its complexity?

Algorithms

Example II

In many problems, we wish to not only find *a* solution, but to find the best or *optimal* solution.

A simple technique that works for *some* optimization problems is called the *greedy technique*.

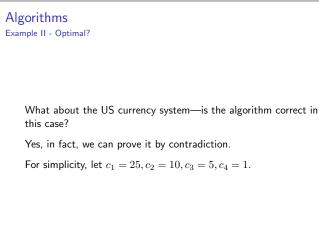
As the name suggests, we solve a problem by being greedy—that is, choosing the best, most immediate solution (i.e. a *local* solution).

However, for some problems, this technique is not guaranteed to produce the best *globally optimal* solution.

Algorithms

Example II - Algorithm

Input	: An integer n and a set of coin denominations (c_1, c_2, \ldots, c_r) with $c_1 > c_2 > \cdots > c_r$.	
Output	: A set of coins $d_1, d_2, \cdots d_k$ such that $\sum_{i=1}^k d_i = n$ and k is minimized.	
$C = \emptyset$		
2 FOR $i = 1,, r$ do		
3 WHILE $n > c_i$ DO		
1	$C = C \cup \{c_i\}$	
5	$n = n - c_i$	
,	$n = n - c_i$	
5 END		
7 END		
output C		



Algorithms Example II

Consider the change problem:

Problem

Given An integer n and a set of coin denominations (c_1, c_2, \ldots, c_r) with $c_1 > c_2 > \cdots > c_r$ **Output** A set of coins $d_1, d_2, \cdots d_k$ such that $\sum_{i=1}^k d_i = n$ and k is minimized.

- Can you describe an algorithm to solve this problem?
- How complex is it?
- Is it optimal?

Algorithms Example II - Optimal?

Will this algorithm always produce an optimal answer?

Consider a coinage system where $c_1=1,c_2=7,c_3=15,c_4=20$ and we want to give 22 "cents" in change.

What will this algorithm produce?

Is it optimal?

It is not optimal since it would give us one c_4 and two $c_1,$ for three coins, while the optimal is one c_2 and one c_3 for two coins.

Algorithms

Example II - Proof

- ▶ Let C = {d₁, d₂, ..., d_k} be the solution given by the greedy algorithm for some integer n. By way of contradiction, assume there is another solution C' = {d'₁, d'₂, ..., d'_l} with l < k.</p>
- \blacktriangleright Consider the case of quarters. Say there are q quarters in C and q' quarters in C'
- If q' > q we are done: the greedy algorithm uses fewer quarters and so fewer coins
- \blacktriangleright If q' < q: the greedy algorithm uses as many quarters as possible so:
 - ▶ n = q(25) + r where r < 25
 - ▶ since, q' < q, n = q'(25) + r' where $r' \ge 25$
- Thus, C' does not provide an optimal solution
 Finally, if q = q', then we continue this argument on dimes
- and nickels. Eventually we reach a contradiction.
- Thus, C = C' is our optimal solution.

Algorithms

Example II - Proof

Why (and where) does this proof fail in our previous counter example to the general case?

The algorithm fails because there is no *greedy choice* property: locally optimal solutions do not lead to a globally optimal solution.

Input Size I

For a given problem, we characterize the input size, $\boldsymbol{n},$ appropriately:

- Sorting The number of items to be sorted
- Graphs The number of vertices and/or edges
- Numerical The number of bits needed to represent a number

Orders of Growth

An objective analysis means that we look at the *order of growth* with respect to the input size

- Small input sizes can be computed instantaneously
- Hardware is continually improving
- Complexity should be independent of current technology

Objectively, we are more interested in how an algorithm performs as $n \to \infty$

Algorithm Analysis

How can we say that one algorithm performs better than another? Quantify the resources required to execute:

- Time
- Memory
- ► I/O
- circuits, power, etc

Time is not merely CPU clock cycles, we want to study algorithms *independent* or implementations, platforms, and hardware.

We need an objective point of reference. For that, we measure time as a function of an algorithm's *input size*.

Input Size II

The choice of an input size greatly depends on the *elementary operation*; the most relevant or important operation of an algorithm.

- Comparisons
- Additions
- Multiplications

Intractability I

Intractable problems are problems for which there are no known efficient algorithms

- May only have a brute-force exponential or super-exponential running time
- Small inputs may be solved in a reasonable amount of time
- Moderate to large inputs: no hope of efficient execution
- Even with faster technology: may take millions or billions of years
- Intractable problems are usually be solved using approximations, heuristics, randomized algorithms, etc.

Intractability II

 $\ensuremath{\textit{Tractable}}\xspace$ problems are problems that have efficient algorithms to solve them

- ► A *polynomial* order of magnitude
- \blacktriangleright The number of steps can be bounded by $p(n)=n^k$ for some constant k
- If k is large, the algorithm may still be *impractical*

Worst, Best, and Average Case

- Some algorithms perform differently on various inputs of a similar size
- Helpful to consider:Worst-Case, Best-Case, and Average-Case efficiencies of algorithms
- \blacktriangleright Motivating example: searching an array ${\cal A}$ of size n for a given value K
 - ► Worst-Case: $K \notin A$ then we must search *every* item (*n* comparisons)
 - \blacktriangleright Best-Case: \dot{K} is the first item that we check, so only one comparison

Average-Case II

Consider searching an array for an element a:

- \blacktriangleright Let p be the probability of a successful search
- Assume a uniform probability on the index
- ▶ Then number of comparisons when *a* is found at index *i*:

$$i\frac{1}{2}$$

Summing over all possible indices:

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

Amotized Cost

- C_{avg} and C_{worst} may have the same order of magnitude
- From a theoretical point of view, they are equivalent
- Practical considerations may come in to play
- May motivate another approach: Amortized efficiency
- Similar to loan amortization
- A single operation may be costly, but the overall run-time over the long-run is less expensive
- Example: rehashing a hash-based map to improve subsequent look-ups

Average-Case I

- Some inputs may lead to poor performance, but may be rare
- Some inputs may lead to great performance, but may also be rare
- Rare instances may give an unfair perspective
- A frequently used algorithm's performance may be based on how it performs *on average*

Average-Case III

- \blacktriangleright Probability of an unsuccessful search: (1-p)
- ▶ Number of comparisons in unsuccessful search: n(1-p)
- In total:

$$C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p) \approx \frac{n}{2}$$

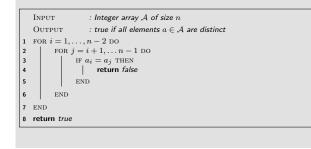
 \blacktriangleright Interpretation: on average, the search algorithm must examine half of all elements in ${\cal A}$

Mathematical Analysis of Algorithms After developing an algorithm, we must analyze; a general approach: Decide on a parameter(s) for the input, n Identify the basic operation Evaluate how the elementary operation depends on n Generate a general formula for the number of times the elementary operation is executed with respect to n Simplify the equation to get as simple of a function f(n) as possible.

Analysis Examples Example I

Consider the following code.

Algorithm (UNIQUEELLEMENTS)



Analysis Example Example I - Analysis

For this algorithm, what is

- The elementary operation?
- Input Size?
- ▶ Does the elementary operation depend only on *n*?

The outer for-loop is run n-2 times. More formally, it contributes

 $\sum_{i=1}^{n-2}$

Analysis Example

Example II

The *parity* of a bit string determines whether or not the number of 1s appearing in it is even or odd. It is used as a simple form of error correction over communication networks.

Algorithm (PARITY)

	Input	: An integer n in binary (b[])		
	Output	: 0 if the parity of n is even, 1 otherwise		
1	parity = 0			
2	While $n > 0$ do			
3	IF $b[0] = 1$ THEN			
4	$parity = parity + 1 \mod 2$			
5	rie	ht-shift(n)		
6	END			
7	END			
8	return parity			

Analysis Example Example I - Analysis

The inner for-loop *depends* on the outer for-loop, so it contributes

 $\sum_{j=i+1}^{n-1}$

We observe that the elementary operation is executed once in each iteration, thus we have

$$C_{worst}(n) = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{n(n-1)}{2}$$

Example: Selection Sort

- Pseudocode
- Input, input size
- Elementary operation
- Analysis
- Asymptotics

Example: Euclid's GCD Algorithm

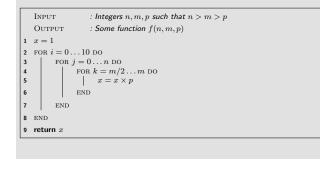
- The greatest common divisor (GCD) of two integers is the largest integer that evenly divides both of them
- ▶ Euclid (Greek, 300 BCE): any divisor must also divide the remainder of *a*/*b*, so iteratively divide until there is no remainder

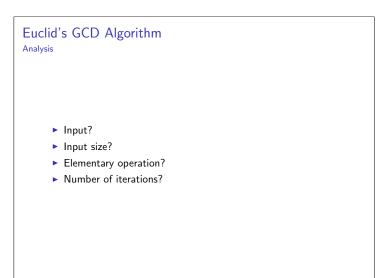
Algorithm (GCD) INPUT : Integers, a, b, a > 1, b > 1OUTPUT : g such that g = gcd(a, b)1 WHILE $b \neq 0$ DO 2 $t \leftarrow b$ 3 $b \leftarrow a \mod b$ 4 $a \leftarrow t$ 5 END 6 output a

Euclid's 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: b 2ⁿ = 1 The number of iterations, n = log b

Analysis Example Example III

Algorithm (MYFUNCTION(n, m, p))





Analysis Example Example II - Analysis

For this algorithm, what is

- The elementary operation?
- Input Size?
- ▶ Does the elementary operation depend only on *n*?

The while-loop will be executed as many times as there are 1-bits in its binary representation. In the worst case, we'll have a bit string of all ones.

The number of bits required to represent an integer n is

 $\lceil \log n \rceil$

so the running time is simply $\log n$.

Analysis Example Example III - Analysis

- ► Outer Loop: executed 11 times.
- 2nd Loop: executed n + 1 times.
- Inner Loop: executed about $\frac{m}{2}$ times.
- Thus we have

C(n, m, p) = 11(n + 1)(m/2)

- But, do we really need to consider p or m?
- ▶ If m = f(n), yes
- If n >> m, probably not