Algorithms Analysis

Section 3.3 of Rosen

Spring 2017

CSCE 235H Introduction to Discrete Structures (Honors)

Course web-page: cse.unl.edu/~cse235h

Questions: Piazza

Outline

- Introduction
- Input Size
- Order of Growth
- Intractability
- Worst, Best, and Average Cases
- Mathematical Analysis of Algorithms
  - 3 Examples
- Summation tools

Introduction

- How can we say that one algorithm performs better than another one?
- Quantify the resources needed to run it:
  - Time
  - Memory
  - I/O, disk access
  - Circuit, power, etc.
- We want to study algorithms independent of
  - Implementations
  - Platforms
  - Hardwar
- We need an objective point of reference
  - For that we measure time by the number of operations as a function of the size of the input to the algorithm
  - Time is not merely CPU clock cycle

Input Size

- For a given problem, we characterize the input size n appropriately
  - Sorting: The number of items to be sorted
  - Graphs: The number of vertices and/or edges
  - Matrix manipulation: The number of rows and colums
  - Numerical operations: the number of bits needed to represent a number
- The choice of an input size greatly depends on the elementary operation: the most relevant or important operation of an algorithm
  - Comparisons
  - Additions
  - Multiplications

Outline

- Introduction
- Input Size
- **Order of Growth**
- **Intractability**
- **Worst, Best, and Average Cases**
- Mathematical Analysis of Algorithms
  - 3 Examples
- Summation tools

Order of Growth

- Small input sizes can usually be computed instantaneously, thus we are most interested in how an algorithms performs as $n \to \infty$
- Indeed, for small values of n, most such functions will be very similar in running time.
- Only for sufficiently large n do differences in running time become apparent:

  As $n \to \infty$ the differences become more and more stark

Intractability

- Problems that we can solve (today) only with exponential or super-exponential time algorithms are said to be (likely) intractable. That is, though they may be solved in a reasonable amount of time for small n, for large n, there is (likely) no hope for efficient execution. It may take millions or billions of years.
- Tractable problems are problems that have efficient (read: polynomial) algorithms to solve them.
- Polynomial order of magnitude usually means that there exists a polynomial $p(n)=n^k$ for some constant k that always bounds the order of growth. More on asymptotics in the next lecture
- (Likely) Intractable problems (may) need to be solved using approximation or randomized algorithms (except for small size of input)

Worst, Best, and Average Case

- Some algorithms perform differently on various input of similar size. It is sometimes helpful to consider
  - The worst-case
  - The best-case
  - The average-case

  Performance of the algorithm.
- For example, say we want to search an array A of size n for a given value k
  - Worst-case: $k \in A$, then we must check every item. Cost = n comparisons
  - Best-case: k is the first item in the array. Cost = 1 comparison
  - Average-case: Probabilistic analysis

Average-Case: Example

- Since any worthwhile algorithm will be used quite extensively, the average running time is arguably the best measure of the performance of the algorithm (if the worst case is not frequently encountered).
- For searching an array and assuming that p is the probability of a successful search we have

  Average cost of success: $(1 + 2 + \ldots + n)/n$ operations

  Cost of failure: n operations

  $C_{average}(n)$ = Cost(success).Prob(success) + Cost(failure).Prob(failure)

  $= (1 + 2 + \ldots + i + n)\, p/n + n(1-p)$

  $= (n(n+1)/2)\, p/n + n\,(1-p) = p(n+1)/2 + n\,(1-p)$

  – If $p = 0$ (search fails), $C_{average}(n) = n$
  – If $p = 1$ (search succeeds), $C_{average}(n) = (n+1)/2 \approx n/2$

    Intuitively, the algorithm must examine on average half of all the elements in A

Average-Case: Importance

- Average-case analysis of algorithms is important in a practical sense
- Often C_{avg} and C_{worst} have the same order of magnitude and thus from a theoretical point of view, are no different from each other
- Practical implementations, however, require a real-world examination and empirical analysis

Outline

- Introduction
- Input Size
- Order of Growth
- Intractability
- Worst, Best, and Average Cases
- **Mathematical Analysis of Algorithms**
  - **3 Examples**
- Summation tools

Mathematical Analysis of Algorithms

- After developing a pseudo-code for an algorithm, we wish to analyze its performance
  - as a function of the size of the input, n,
  - in terms of how many times the elementary operation is performed.
- Here is a general strategy
  1. Decide on a parameter(s) for the input, n
  2. Identify the basic operation
  3. Evaluate if the elementary operation depends only on n
  4. Set up a summation corresponding to the number of elementary operations
  5. Simplify the equation to get as simple of a function f(n) as possible

Algorithm Analysis: Example 1 (1)

UniqueElements

Input: Integer array A of size n

Output: True if all elements a ∈ A are distinct

1. **For** i=1,…, (n-1) **Do**
2. **For** j=i+1,…,n **Do**
3. **If** $a_i=a_j$
4. **Then Return** *false*
5. **End**
6. **End**
7. **End**
8. **Return** *true*

Algorithm Analysis: Example 1 (2)

- For this algorithm, what is
  - The elementary operation? — Comparing a_i and a_j
  - Input size? — n, size of A
  - Does the elementary operation depend only on n?
- The outer for-loop runs n-1 times. More formally it contributes: $\sum_{i=1}^{n-1}$
- The inner for-loop depends on the outer for-loop and contributes: $\sum_{j=i+1}^{n}$

Algorithm Analysis: Example 1 (3)

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

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

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

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

  $= n(n-1) - \sum_{i=1}^{n-1} i$
- Computing $\sum_{i=1}^{n-1} i$
  - Check Table 2, page 157: $\sum_{k=1}^{n} k=n(n+1)/2$
  - Rewrite $\sum_{i=1}^{n-1} i = \sum_{i=1}^{n} i - n = n(n+1)/2 - n$

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

Algorithm Analysis: Example 2 (1)

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

Algorithm Analysis: ParityChecking

ParityChecking

Input: An integer *n* in binary (as an array b[])

Output: 0 if parity 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. **End**
6. LeftShift(n)
7. **End**
8. **Return** *parity*

Algorithm Analysis: Example 2 (2)

- For this algorithm, what is
  - The elementary operation?
  - Input size, *n*?
  - Does the elementary operation depend only on *n*?
- The number of bits required to represent an integer n is $\lceil \log n \rceil$
- The while-loop will be executed as many times as there are 1-bits in the binary representation.
- In the worst case we have a bit string of all 1s
- So the running time is simply log *n*

Algorithm Analysis: Example 3 (1)

MyFunction

Input: Integers n,m,p such that $n>m>p$

Output: Some function $f(n,m,p)$

1. $x \leftarrow 1$
2. **For** $i = 0,\ldots,10$ **Do**
3. **For** $j = 0,\ldots, n$ **Do**
4. **For** $k= m/2, \ldots, m$ **Do**
5. $x \leftarrow x \times p$
6. **End**
7. **End**
8. **End**
9. **Return** x

Algorithm Analysis: Example 3 (2)

- Outer for-loop: executes 11 times, but does not depend on input size
- 2nd for-loop: executes *n+1* times
- 3rd for-loop: executes *m/2+*1 times
- Thus, the cost is C(*n*,*m*,*p*)=11(*n*+1)(*m*/2+1)
- And we do NOT need to consider *p*

Outline

- Introduction
- Input Size
- Order of Growth
- Intractability
- Worst, Best, and Average Cases
- Mathematical Analysis of Algorithms
  - 3 Examples
- **Summation tools**

Summation Tools

- Table 2, Section 2.4 (page 166) has more summation rules, which will be
- You can always use Maple to evaluate and simplify complex expressions
  - But you should know how to do them by hand!
- To use Maple on cse you can use the command-line interface by typing `maple`
- Under Unix (gnome of KDE) or via xwindows interface, you can use the graphical version via `xmaple`
- Will try to demonstrate during the recitation.

Summation Tools: Maple

```
> simplify(sum(i,i=0..n));
```

$$\tfrac{1}{2}\, n^2 + \tfrac{1}{2}\, n$$

```
> Sum(Sum(j,j=i..n),i=0..n);
```

$$\sum_{i=0}^{n} \left(\sum_{j=i}^{n} j \right)$$

Summary

- Introduction
- Input Size
- Order of Growth
- Intractability
- Worst, Best, and Average Cases
- Mathematical Analysis of Algorithms
  - 3 Examples
- Summation tools