Algorithm Analysis

Slides by Christopher M. Bourke
Instructor: Berthe Y. Choueiry

Fall 2007

Computer Science & Engineering 235
Section 3.3 of Rosen
cse235@cse.unl.edu

Introduction

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 by "the number of operations as a function of an algorithm's *input size*."

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
- Numerical – 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

Orders of Growth

Small input sizes can usually be computed instantaneously, thus we are most interested in how an algorithm 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 of 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 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 slide set.

Intractable problems may need to be solved using approximation or randomized algorithms.

Worst, Best, and Average Case

Some algorithms perform differently on various inputs of similar size. It is sometimes helpful to consider the Worst-Case, Best-Case, and Average-Case efficiencies of algorithms. For example, say we want to search an array $\mathcal{A}$ of size n for a given value K.

- Worst-Case: $K \notin \mathcal{A}$ then we must search *every* item (n comparisons)
- Best-Case: K is the first item that we check, so only one comparison

Average-Case

Since any worth-while algorithm will be used quite extensively, the average running-time is arguably the best measure of its performance (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:

$$\begin{aligned} C_{avg}(n) &= \frac{p + 2p + \ldots + ip + \ldots np}{n} + n(1-p) \\ &= \frac{p}{n}(1 + 2 + \ldots + i + \ldots + n) + n(1-p) \\ &= \frac{p}{n}\frac{(n(n+1)}{2} + n(1-p) \end{aligned}$$

If $p = 1$ (search succeeds), $C_{avg}(n) = \frac{n+1}{2} \approx .5n$.
If $p = 0$ (search fails), $C_{avg}(n) = n$.
A more intuitive interpretation is that the algorithm must examine, on average, half of all elements in $\mathcal{A}$.

Average-Case

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.

Mathematical Analysis of Algorithms

After developing pseudo-code for an algorithm, we wish to analyze its efficiency 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 (otherwise evaluate best, worst, and average-case separately.
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.

Analysis Examples

Example I

Consider the following code.

Algorithm (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 THEN
4              return false
5          END
6      END
7  END
8  return true
```

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 $n-1$ times. More formally, it contributes

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

Analysis Example

Example I - Analysis

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

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

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

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

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           right-shift(n)
6       END
7   END
8   return parity
```

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

Algorithm (MYFUNCTION(n, m, p))

```
    INPUT     : Integers n, m, p such that n > m > p
    OUTPUT    : Some function f(n, m, p)
1   x = 1
2   FOR i = 0, ..., 10 DO
3       FOR j = 0, ..., n DO
4           FOR k = m/2, ..., m DO
5               x = x × p
6           END
7       END
8   END
9   return x
```

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?

Summation Tools I

Section 2.4 (p157) has more summation rules. You can always use Maple to evaluate and simplify complex expressions (but know how to do them by hand!).

To invoke maple, on cse you can use the command-line interface by typing `maple`. Under unix (gnome or KDE) or via any xwindows interface, you can use the graphical version via `xmaple`.

Will be demonstrated during recitation.

Summation Tools II

Example

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

$$\frac{1}{2}n^2 + \frac{1}{2}n$$

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

$$\sum_{i=0}^{n} \left(\sum_{j=1}^{n} j \right)$$