EECS 2011 M: Fundamentals of Data Structures

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/2011M`
Also on Moodle

Introduction to Algorithm Analysis

Objectives

- Review how to describe algorithms
- Learn to reason about the running times of algorithms, and compare their efficiency
- To be able to compare algorithms and choose appropriate ones

Note: Some slides in this presentation have been adapted from the authors' slides.

Algorithms and Pseudo-code

[Webster] Algorithm: a procedure for solving a mathematical problem in a finite number of steps ...

Pseudo-code

- Use English rather than a real programming language
- More high-level than code
- Hides many details
- Preferred notation for describing algorithms

Pseudo-code - 2

- Control flow
  - if ... then ... [else . . .]
  - while ... do ...
  - repeat ... until ...
  - for ... do ...
    Indentation replaces braces
- Method declaration
  Algorithm method (arg [, arg ...])
  Input ...
  Output ...

Pseudo-code - 3

- Method call: method (arg [, arg. . .])
- Return value: return expression
- Expressions:
  - Assignment: $\leftarrow$ item Equality testing: $=$
  - Superscripts and other mathematical formatting allowed

Reasoning about Algorithms

- I/O specs: Needed for correctness proofs, performance analysis. E.g. for sorting:
  INPUT: $A[1..n]$ - an array of integers
  OUTPUT: a permutation B of A such that
  $B[1] \leq B[2] \leq \ldots \leq B[n]$
- Correctness: The algorithm satisfies the output specs for EVERY valid input – Later
- Analysis: Compute the performance of the algorithm, e.g., in terms of running time – next

Analysis of Algorithms

- Measures of efficiency:
  - Running time
  - Space used
  - others, e.g., number of disk accesses, network accesses,...
- Efficiency as a function of input size (NOT value!)
  - Number of data elements (numbers, points)
  - Number of bits in an input number
  - Examples: Find the factors of a number n, Determine if an integer n is prime
- Machine Model

Machine Model: Specific or Generic?

Modern computers are incredibly complex.

- Modeling the memory hierarchy and network connectivity generically is very difficult
- All modern computers are "similar" in that they provide the same basic operations.
- Most general-purpose processors today have at most eight processors or "cores". The vast majority have one or two or four. GPU's have hundreds or thousands.

Note: Need a generic model that models (approximately) all machines

A Standardized, Abstract Machine Model

Random Access Machine (RAM) assumptions:

- Instructions (each taking constant time):
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
    - Comparison
- Data types – integers, characters, and floats

Note:

Ignores memory hierarchy, network!

Asymptotic Analysis

- Instructions (each taking constant time):
  - Arithmetic (add, subtract, multiply, etc.)
  - Data movement (assign)
  - Control (branch, subroutine call, return)
  - Comparison
- Data types – integers, characters, and floats

Asymptotic Analysis

- Cannot capture exact running times on a specific machine
- Captures the nature of growth of running times, NOT actual values
- Want to make statements like, "the running time of an algorithm grows linearly with input size".
- Very useful for studying the behavior of algorithms for LARGE inputs

An Example: Find the max of n numbers

Input: $A[1..n]$ - an array of integers
Output: an element m of A such that $A[j] \leq m$, $1 \leq j \leq n$

$\textsc{FindMax}(A)$

$n \leftarrow length(A)$
$max \leftarrow A[1]$
for $j \leftarrow 2$ **to** n
do if $max < A[j]$
    then $max \leftarrow A[j]$
return max

Find the max of n numbers: Java

```
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[ ] data) {
  int n = data.length;
  double currentMax = data[0];              // assume first entry is biggest (for now)
  for (int j=1; j < n; j++)                 // consider all other entries
    if (data[j] > currentMax)               // if data[j] is biggest thus far...
      currentMax = data[j];                 // record it as the current max
  return currentMax;
}
```

Analysis of FindMax

```
FindMax(A)
1  n ← length(A)
2  max ← A[1]
3  for j ← 2 to n
4  do if max < A[j]
5        then max ← A[j]
6  return max
```

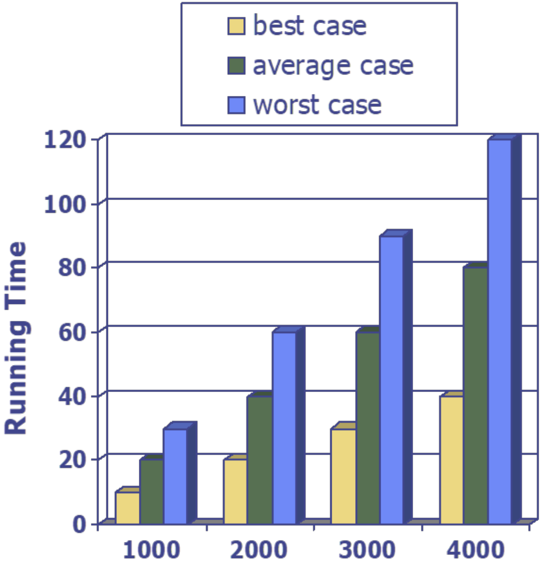
line	Cost	Times
1	c_1	1
2	c_2	1
3	c_3	n
4	c_4	$n-1$
5	c_5	$0 \leq k \leq n-1$
6	c_6	1

Best Case: $k = 0$

Worst Case: $k = n - 1$

Average Case: ?

Best/Worst/Average Case Analysis



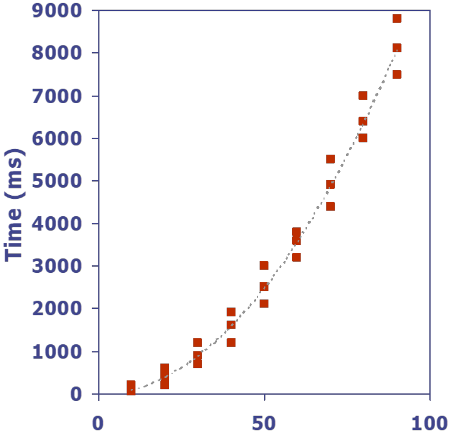
Best/Worst/Average Case Analysis - 2

The running time of an algorithm typically grows with the input size.

- Best Case: Not very informative
- Average Case: Often very useful, but hard to determine
- Worst Case: Easier to analyze. Crucial in applications like
  - Games
  - Finance
  - Robotics

Experimental Analysis of Running Time

```
long startTime = System.currentTimeMillis();     // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis();       // record the ending time
long elapsed = endTime − startTime;              // compute the elapsed time
```



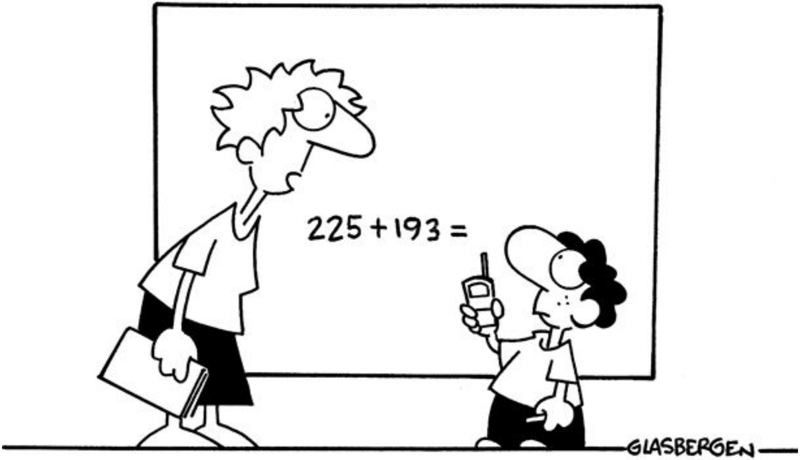
Experimental Analysis of Running Time - Issues

- Need an (efficient) implementation
- Get running time as a function of the input size n
- Takes into account all possible inputs
- Only valid on an abstract model of the hardware/software environment

Theoretical Analysis of Running Time

- Need description/pseudo-code, not implementation
- Hard to know if the inputs used are representative
- To compare two algorithms, the same hardware and software environments must be used

Some Math Review



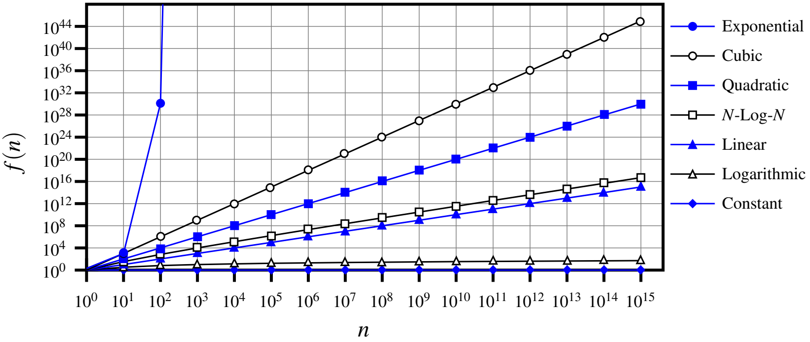
"You have to solve this problem by yourself. You can't call tech support."

Seven Important Functions

Seven functions that appear frequently in algorithm analysis:

- Constant ≈ 1
- Logarithmic $\approx \log n$
- Linear $\approx n$
- N-Log-N $\approx n \log n$
- Quadratic $\approx n^2$
- Cubic $\approx n^3$
- Exponential $\approx 2^n$

Seven Important Functions - 2



Note the log-log axes

Relevant Math Facts - Exponents

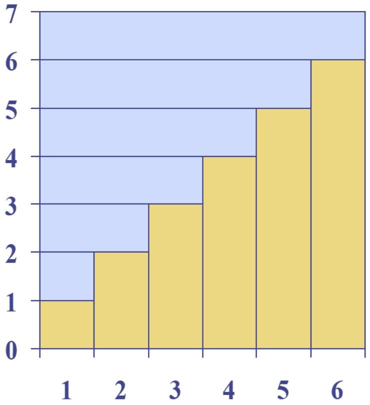
- $a^{(b+c)} = a^b a^c$
- $a^{bc} = (a^b)^c$
- $a^b / a^c = a^{b-c}$
- $b = a^{\log_a b}$
- $b^c = a^{c \log_a b}$

Relevant Math Facts - Logarithms

- $\log_b(xy) = \log_b x + \log_b y$
- $\log b(x/y) = \log_b x - \log_b y$
- $\log_b x^a = a \log_b x$
- $\log_b a = \log_x a / \log_x b$

Also, note the difference between $\log \log n$ and $(\log n)^2 = \log^2 n$.

Relevant Math Facts - Sums of Series



The sum of the first n integers is
$1 + 2 + \ldots + n = n(n+1)/2$

Analysis of FINDMAX - Continued

FINDMAX(A)

```
1   n ← length(A)
2   max ← A[1]
3   for j ← 2 to n
4   do if max < A[j]
5           then max ← A[j]
6   return max
```

line	Cost	Times
1	c_1	1
2	c_2	1
3	c_3	n
4	c_4	$n-1$
5	c_5	$0 \leq k \leq n-1$
6	c_6	1

Running time (worst-case):
$c_1 + c_2 + c_6 - c_4 - c_5 + (c_3 + c_4 + c_5)n$
Running time (best-case): $c_1 + c_2 + c_6 - c_4 + (c_3 + c_4)n$

Simplifying Running Times

Note that the worst-case time of
$c_1 + c_2 + c_6 - c_4 - c_5 + (c_3 + c_4 + c_5)n$ is

- Complex
- Not useful as the c_i's are machine dependent

A simpler expression: $C + Dn$ [still complex].

Want to say this is Linear, i.e., $\approx n$

Q: How/why can we throw away the coefficient D and the lower order term C?

Simplifying Running Times - Rationale

- Discarding lower order terms: We are interested in large n – cleaner theory, usually realistic.
- Discarding coefficients (multiplicative constants): the coefficients are machine dependent

Caveat: remember these assumptions when interpreting results! We will not get:

- Exact run times
- Comparison for small instances
- Small differences in performance

Asymptotic Analysis

Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware

- So $3n^2 - 5n + 6 \approx n^2$
- Capturing the essential information: how the running time of an algorithm increases with the size of the input in the limit
- Asymptotically more efficient algorithms are best for all but small inputs

Asymptotic Notation: Big-Oh

Suppose $f(n)$ and $g(n)$ are functions over non-negative integers

The "big-Oh" Notation $\mathcal{O}()$ is defined as

$f(n) \in \mathcal{O}(g(n))$, if there exists real number constants $c > 0$ and $n_0 > 0$, satisfying $f(n) \leq cg(n)$ for all natural numbers $n \geq n_0$

Example:

- $2n + 10 \in \mathcal{O}(n)$
- $3n^2 - 5n + 6 \in \mathcal{O}(n^2)$
- $2n + 10 \in \mathcal{O}(n^3)$

Big-Oh: Intuition

We choose $g(n)$ to be a very simple function

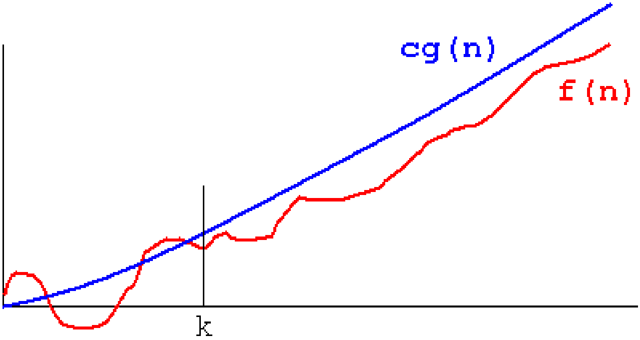


Image: https://xlinux.nist.gov/dads/Images/bigOGraph.gif

Asymptotic Notation: Big-Omega

The "big-Oh" Notation $\Omega()$ is defined as
$f(n) \in \Omega(g(n))$, if there exists real number constants $c > 0$ and $n_0 > 0$, satisfying $f(n) \geq cg(n)$ for all natural numbers $n \geq n_0$

Example:

- $2n + 10 \in \Omega(n)$
- $3n^2 - 5n + 6 \in \Omega(n^2)$
- $3n^2 - 5n + 6 \in \Omega(n)$

Asymptotic Notation: Big-Theta

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

- $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$
- there exists real number constants $c_1 > 0, c_2 > 0$ and $n_0 > 0$, satisfying $c_2 g(n) \geq f(n) \geq c_1 g(n)$ for all natural numbers $n \geq n_0$

Example:

- $2n + 10 \in \Theta(n)$
- $3n^2 - 5n + 6 \in \Theta(n^2)$
- $3n^2 - 5n + 6 \notin \Omega(n)$, $2n + 10 \notin \Theta(n^2)$

Big-Theta: Intuition

Again, we choose $g(n)$ to be a very simple function

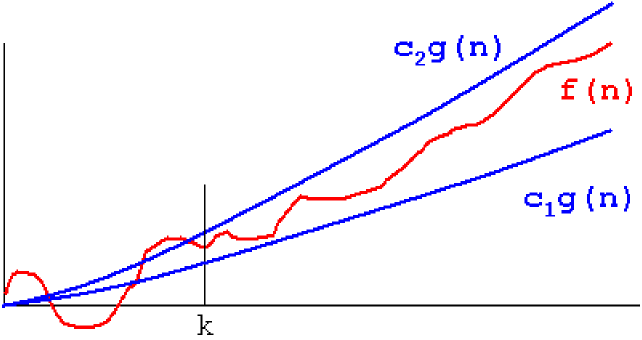


Image: https://xlinux.nist.gov/dads/Images/thetaGraph.gif

Common Abuses of Notation

Many, many abuses of asymptotic notation in EECS literature.

- $f(n) = \mathcal{O}(g(n))$ instead of $f(n) \in \mathcal{O}(g(n))$
- $\mathcal{O}(g(n))$ instead of $\Theta(g(n))$

Common "colloquial" uses:

- $\Theta(1)$ – constant.
- $n^{\Theta(1)}$ – polynomial
- $2^{\Theta(n)}$ – exponential

Common Mistakes

- $n^{\Theta(1)} \notin \Theta(n^1)$
- $2^{\Theta(n)} \notin \Theta(2^n)$

Important Facts

- Logarithmic << Polynomial: $\log^{1000} n << n^{0.001}$ For sufficiently large n
- Linear << Quadratic: $10000n << 0.0001n^2$ For sufficiently large n
- Polynomial << Exponential: $n^{1000} << 2^{0.001n}$ For sufficiently large n

Proving Asymptotic Facts

$f(n) = 3n^2 + 7n + 8 \in \Theta(g(n))$

- Choosing $g(n)$: Simple Rule – Drop lower order terms and constant factors. So $g(n) = n^2$.
- Use definitions
  e.g. there exists real number constants
  $c_1 > 0, c_2 > 0$ and $n_0 > 0$, satisfying
  $c_2 g(n) \geq f(n) \geq c_1 g(n)$ for all natural numbers
  $n \geq n_0$

Proving Asymptotic Facts - 2

- $3n^2 + 7n + 8 > 3n^2 + 7n > 3n^2 > n^2$ for all $n \geq 0$, so $f(n) \geq c1g(n)$ with $c_1 = 3$ and $n_0 = 1$
- $7n < 7n^2$ for $n > 1$. Similarly $8 < 8n^2$ for $n > 1$. So $3n^2 + 7n + 8 < 3n^2 + 7n^2 + 8n^2 = 18n^2$ for all $n > 1$, so $f(n) \leq c_2g(n)$ with $c_2 = 18$ and $n_0 = 2$

So we have shown that $f(n) \in \Theta(n^2)$ using the definition of $\Theta()$ with $c_1 = 3, c_2 = 18, n_0 = 2$

Proving Asymptotic Facts - 3

- constants c_1, c_2 MUST be POSITIVE (> 0)
- Could have chosen $c_2 = 3 + \epsilon$ for any $\epsilon > 0$, because $7n + 8 \leq \epsilon n^2$ for sufficiently large n. Usually, the smaller the ϵ you choose, the harder it is to find n_0. So choosing a larger ϵ is easier
- Order of quantifiers matters!
  $\exists c_1 c_2 \exists n_0 \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$
  vs
  $\exists n_0 \forall n \geq n_0 \exists c_1 c_2, c_1 g(n) \leq f(n) \leq c_2 g(n)$
- allows a different c_1 and c_2 for each n. Can choose $c_2 = 1/n$, and "prove" $n^3 \in \Theta(n^2)$.

Another problem

The i^{th} prefix average of an array X is the average of the first $i+1$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

We will look at 2 implementations.

A Slower Algorithm

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];                    // filled with zeros by default
  for (int j=0; j < n; j++) {
    double total = 0;                             // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1);                         // record the average
  }
  return a;
}
```

Good example for determining the running time

Analysis

- Outer loop iterates for $j = 0, \ldots, n - 1$
- Inner loop iterates for $i = 0, \ldots, j$
- The loop body takes $\Theta(1)$ steps

Analysis - 2

The easiest way to sum the running time is

$$
\begin{aligned}
T(n) &= \sum_{j=0}^{n-1} \sum_{i=0}^{j} 1 \\
&= \sum_{j=0}^{n-1} (j+1) \\
&= \sum_{j=1}^{n} j \\
&= n(n+1)/2
\end{aligned}
$$

So $T(n) \in \Theta(n^2)$

A Faster Algorithm

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];          // filled with zeros by default
  double total = 0;                     // compute prefix sum as x[0] + x[1] + ...
  for (int j=0; j < n; j++) {
    total += x[j];                      // update prefix sum to include x[j]
    a[j] = total / (j+1);               // compute average based on current sum
  }
  return a;
}
```

Analysis: Linear time $\Theta(n)$

More practice - 1

Find the running time:

```
MatMult(Y, Z, n)
// multiply n x n matrices Y, Z
for i ← 1 to n
do for j ← 1 to n
      do X[i, j] ← 0
         for k ← 1 to n
         do X[i, j] ← X[i, j] + Y[i, k] ∗ Z[k, j]
return x
```

More practice - 2

Analyze the running time of the following algorithm.

POWER(y, z)

// return y^z where $y \in R, z \in N$
$x \leftarrow 1$
while $z > 0$
do if $odd(z)$
        then $x \leftarrow x * y$
    $z \leftarrow \lfloor z/2 \rfloor$
    $y \leftarrow y^2$
return x