CSCE 310J: Data Structures & Algorithms

Analysis of Algorithms

Dr. Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE310J

CSCE 310J: Data Structures & Algorithms

- **Giving credit where credit is due:**
  - **Most of the lecture notes are based on the slides from the Textbook's companion website**
    - **http://www.aw.com/cssuport/**
  - **Several slides are from William Spears of the University of Wyoming**
  - **I have modified them and added new slides**

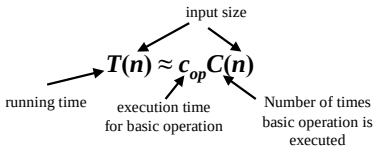
Analysis of Algorithms

- **Issues:**
  - **Correctness**
  - **Time efficiency**
  - **Space efficiency**
  - **Optimality**
- **Approaches:**
  - **Theoretical analysis**
  - **Empirical analysis**

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- ***Basic operation:*** **the operation that contributes most towards the running time of the algorithm.**



Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Search for key in list of n items	Number of items in list n	Key comparison
Multiply two matrices of floating point numbers	Dimensions of matrices	Floating point multiplication
Compute a^n	n	Floating point multiplication
Graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Empirical analysis of time efficiency

- **Select a specific (typical) sample of inputs**
- **Use physical unit of time (e.g., milliseconds)**

  OR
- **Count actual number of basic operations**
- **Analyze the empirical data**

Best-case, average-case, worst-case

For some algorithms efficiency depends on type of input:

- **Worst case: W(*n*) – maximum over inputs of size *n***
- **Best case: B(*n*) – minimum over inputs of size *n***
- **Average case: A(*n*) – "average" over inputs of size *n***
  - **Number of times the basic operation will be executed on typical input**
  - **NOT the average of worst and best case**
  - **Expected number of basic operations repetitions considered as a random variable under some assumption about the probability distribution of all possible inputs of size *n***

Example: Sequential search

- ***Problem:* Given a list of *n* elements and a search key *K*, find an element equal to *K*, if any.**
- ***Algorithm:* Scan the list and compare its successive elements with *K* until either a matching element is found (*successful search*) of the list is exhausted (*unsuccessful search*)**
- **Worst case**
- **Best case**
- **Average case**

Types of formulas for basic operation count

- **Exact formula**
  e.g., $C(n) = n(n-1)/2$
- **Formula indicating order of growth with specific multiplicative constant**
  e.g., $C(n) \approx 0.5\, n^2$
- **Formula indicating order of growth with unknown multiplicative constant**
  e.g., $C(n) \approx cn^2$

Order of growth

- **Most important: Order of growth within a constant multiple as $n \to \infty$**
- **Example:**
  - **How much faster will algorithm run on computer that is twice as fast?**
  - **How much longer does it take to solve problem of double input size?**
- **See table 2.1**

Table 2.1

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Asymptotic growth rate

- **A way of comparing functions that ignores constant factors and small input sizes**
- **$O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$**
- **$\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$**
- **$\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$**

see figures 2.1, 2.2, 2.3

Big-oh

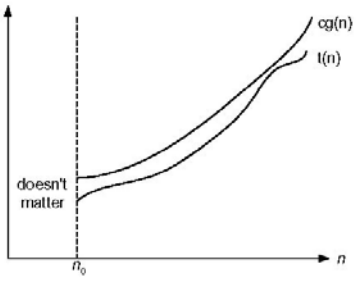


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

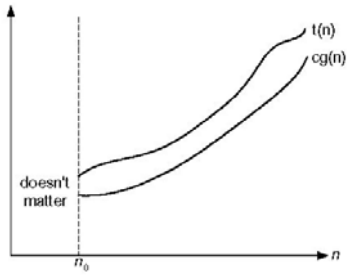


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

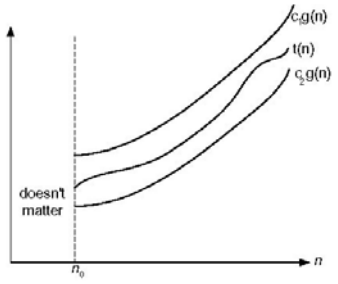


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing rate of growth: Method 1 – using limits

$\lim_{n\to\infty} T(n)/g(n) =$

- 0 — order of growth of $T(n)$ ___ order of growth of $g(n)$
- $c>0$ — order of growth of $T(n)$ ___ order of growth of $g(n)$
- ∞ — order of growth of $T(n)$ ___ order of growth of $g(n)$

Examples:

- $10n$ vs. $2n^2$
- $n(n+1)/2$ vs. n^2
- $\log_b n$ vs. $\log_c n$

L'Hôpital's rule

If

- $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$
- **The derivatives f', g' exist,**

Then

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

An Example

Let $f(N) = 25N^2 + N$ and $g(N) = N^2$.
Then $\lim_{N\to\infty} f(N)/g(N) = 25$.
So $f(N) = \Theta(N^2)$.

Another Example

Let $f(N) = N \log N$ and $g(N) = N^{1.5}$.

Then $\lim_{N\to\infty} f(N)/g(N) = N \log N / N^{1.5} =$ $\log N / N^{.5}$. Now take the derivative of the top and bottom to get : $(1/N)/.5N^{-.5} = 2/N^{.5}$

This approaches 0, so $N \log N = o(N^{1.5})$.

Establishing rate of growth: Method 2 – using definition

- **$f(n)$ is $O(g(n))$ if order of growth of $f(n) \le$ order of growth of $g(n)$ (within constant multiple)**
- **There exist positive constant c and non-negative integer n_0 such that**

$f(n) \le c\, g(n)$ for every $n \ge n_0$

Examples:

- **$10n$ is $O(2n^2)$**
- **$5n+20$ is $O(10n)$**

Basic Asymptotic Efficiency classes

1	**constant**
log *n*	**logarithmic**
n	**linear**
n* log *n	***n* log *n***
n^2	**quadratic**
n^3	**cubic**
2^n	**exponential**
n!	**factorial**

Time efficiency of nonrecursive algorithms

Steps in mathematical analysis of nonrecursive algorithms:

- **Decide on parameter *n* indicating *input size***
- **Identify algorithm's *basic operation***
- **Determine *worst*, *average*, and *best* case for input of size *n***
- **Set up summation for *C(n)* reflecting algorithm's loop structure**
- **Simplify summation using standard formulas (see Appendix A)**

Examples:

- **Matrix multiplication**
- **Selection sort**
- **Insertion sort**
- **Mystery Algorithm**

Matrix multipliacation

Algorithm *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i,j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$
return C

Selection sort

Algorithm *SelectionSort*$(A[0..n \quad 1])$
//The algorithm sorts a given array by selection sort
//Input: An array $A[0..n \quad 1]$ of orderable elements
//Output: Array $A[0..n \quad 1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n \quad 2$ **do**
    $min \leftarrow i$
    for $i \leftarrow i+1$ **to** $n \quad 1$ **do**
        if $A[j] < A[min] \quad min \leftarrow j$
    swap $A[i]$ and $A[min]$

Insertion sort

Algorithm *InsertionSort*$(A[0..n \quad 1])$
//Sorts a given array by insertion sort
//Input: An array $A[0..n \quad 1]$ of n orderable elements
//Output: Array $A[0..n \quad 1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n \quad 1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i \quad 1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j \quad 1$
    $A[j+1] \leftarrow v$

Mystery algorithm

```
for i := 1 to n - 1 do
    max := i ;
    for j := i + 1 to n do
        if |A[ j, i ]| > |A[ max, i ]| then max := j ;
    for k := i  to n + 1 do
        swap A[ i, k ]  with  A[ max, k ];
    for j := i + 1 to n do
        for k := n + 1 downto i do
            A[ j, k ] := A[ j, k ]  - A[ i, k ] * A[ j, i ] / A[ i, i ] ;
```


Programming with Recursion

- **Recursion is similar to a proof by induction:**
  - **There must be a base (trivial) case.**
  - **The recursion is assumed to hold for all *k* < *N*.**
  - **The *N*th case is built from the *k* < *N* cases.**

Aside: Recall Proof by Induction

- **Proof by (strong) induction:**
  - **Show theorem true for trivial case(s). Then, assuming theorem true up to case *N*, show true for *N+1*. Thus true for all *N*.**

Proof that T(N) >= F(N)

Base cases: $T(0) = 1 \geq F(0) = 1,$
$T(1) = 1 \geq F(1) = 1.$
$T(2) = 4 \geq F(2) = 2.$
We know that $T(N+1) > T(N) + T(N-1)$
and $F(N+1) = F(N) + F(N-1).$
Assume theorem holds for all $k, 1 \leq k \leq N$
Now prove for the $N+1$ case:
$T(N+1) > T(N) + T(N-1) \geq F(N) + F(N\text{-}1) = F(N+1)$

Proof that F(N) >= $(3/2)^N$

Base cases: $F(5) = 8 \geq (3/2)^5 = 7.6$,

$F(6) = 13 \geq (3/2)^6 = 11.4$.

Assume theorem holds for all $k, 1 \leq k \leq N$.

Now prove for the $N+1$ case:

$F(N+1) = F(N) + F(N\text{-}1) \geq (3/2)^N + (3/2)^{N-1} =$

$(3/2)^N (1+(2/3)) = (3/2)^N (5/3) >$

$(3/2)^N (3/2) = (3/2)^{N+1}$.

Back to Recursion: Example Recursive evaluation of *n*!

- **Definition:** ***n* ! = 1*2*…*(*n*-1)**n***
- **Recursive definition of *n*!:**
- **Algorithm:**

  if *n*=0 then *F*(*n*) := 1

  else *F*(*n*) := *F*(*n*-1) * *n*

  return *F*(*n*)
- **Recurrence for number of multiplications to compute *n*!:**

Time efficiency of recursive algorithms

Steps in mathematical analysis of recursive algorithms:

- **Decide on parameter *n* indicating *input size***
- **Identify algorithm's *basic operation***
- **Determine *worst*, *average*, and *best* case for input of size *n***
- **Set up a recurrence relation and initial condition(s) for *C*(*n*)-the number of times the basic operation will be executed for an input of size *n* (alternatively count recursive calls).**
- **Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution (see Appendix B)**

Important recurrence types:

- **One (constant) operation reduces problem size by one.**

  $T(n) = T(n\text{-}1) + c$ $T(1) = d$

  Solution: $T(n) = (n\text{-}1)c + d$ ***linear***
- **A pass through input reduces problem size by one.**

  $T(n) = T(n\text{-}1) + cn$ $T(1) = d$

  Solution: $T(n) = \lfloor n(n+1)/2 - 1 \rfloor c + d$ ***quadratic***
- **One (constant) operation reduces problem size by half.**

  $T(n) = T(n/2) + c$ $T(1) = d$

  Solution: $T(n) = c \lg n + d$ ***logarithmic***
- **A pass through input reduces problem size by half.**

  $T(n) = 2T(n/2) + cn$ $T(1) = d$

  Solution: $T(n) = cn \lg n + d n$ ***n* log *n***

A general divide-and-conquer recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^k)$

1. $a < b^k$ $T(n) \in \Theta(n^k)$
2. $a = b^k$ $T(n) \in \Theta(n^k \lg n)$
3. $a > b^k$ $T(n) \in \Theta(n^{\log_b a})$

Note: the same results hold with O instead of Θ.

Math Review: Exponents

$X^A X^B = X^{A+B}$ (not X^{AB} !!)

$X^A / X^B = X^{A-B}$

$(X^A)^B = X^{AB}$

$X^A + X^A = 2X^A$

$2^A + 2^A = 2^{A+1}$

Logarithms

- **Definition: $X^A = B$ if and only if $\log_X B = A$ (x is the "base" of the logarithm).**
  - **Example: $10^2 = 100$ means $\log_{10} 100 = 2$.**
- **Theorems**

  $\log_X AB = \log_X A + \log_X B$

  $\log_X A/B = \log_X A - \log_X B$

  $\log_X A^B = B \log_X A$

Logarithms...

- **Theorem: $\log_A B = \log_C B / \log_C A$**

  Proof: Let $X = \log_C B$, $Y = \log_C A$, and $Z = \log_A B$. By the definition of logarithm: $C^X = B$, $C^Y = A$, and $A^Z = B$.

  Thus $C^X = B = A^Z = C^{YZ}$, $X = YZ$, $Z = X/Y$.

Logarithms...

- ***The notation for logs can be confusing. There are two alternatives:***
  - ***$\log^2(x) = \log (\log x)$ or***
  - ***$\log^2(x) = (\log x)^2$***
- ***Generally, we use the 2nd definition.***
- ***Note: $\log^2(x)$* is not *$(\log x^2)$***
- ***Note: log* is not a multiplicative entity, it is a function.**

Series

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

- **Proof by Gauss when 9 years old (?!):**

$$S = 1 + 2 + 3 + \ldots + (N-2) + (N-1) + N$$
$$S = N + (N-1) + (N-2) + \ldots + 3 + 2 + 1$$
$$2S = N(N+1)$$

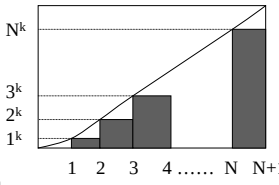
Finite Series

$$S = \sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$$

Finite Series

$$S = \sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|}$$

- **Proof:**



$$\sum_{i=1}^{N} i^k \approx \int_1^N x^k dx = \frac{x^{k+1}}{k+1}\Big|_1^N$$
$$= \frac{N^{k+1}}{k+1} - \frac{1}{k+1}$$

Finite Series

$$S = \sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1} \qquad S = \sum_{i=0}^{N} 2^i = 2^{N+1}-1$$

Proof:

$$(1+A+A^2+\ldots+A^N)(A-1) =$$
$$A+A^2+\ldots+A^{N+1}-1-A-A^2-\ldots-A^N = A^{N+1}-1$$

Finite Series

$$S = \sum_{i=1}^{N} A^i = \frac{A^{N+1}-A}{A-1} \qquad S = \sum_{i=1}^{N} 2^i = 2^{N+1}-2$$

Proof:

$$(A+A^2+\ldots+A^N)(A-1) =$$
$$A^2+\ldots+A^{N+1}-A-A^2-\ldots-A^N = A^{N+1}-A$$

General Rules for Sums

$$\sum_{i=m}^{n} c = c\sum_{i=m}^{n} 1 = c(n-m+1)$$
$$\sum_{i} (a_i + b_i) = \sum_{i} a_i + \sum_{i} b_i$$
$$\sum_{i} c a_i = c\sum_{i} a_i$$
$$\sum_{i=m}^{n} a_{i+k} = \sum_{i=m+k}^{n+k} a_i$$
$$\sum_{i} a_i x^{i+k} = x^k \sum_{i} a_i x^i$$