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

$$T(n) \approx c_{op} C(n)$$

running time

execution time for basic operation

Number of times basic operation is executed

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	$3.3\cdot10^1$	10^2	10^3	10^3	$3.6\cdot10^6$
10^2	6.6	10^2	$6.6\cdot10^2$	10^4	10^6	$1.3\cdot10^{30}$	$9.3\cdot10^{157}$
10^3	10	10^3	$1.0\cdot10^4$	10^6	10^9		
10^4	13	10^4	$1.3\cdot10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7\cdot10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0\cdot10^7$	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

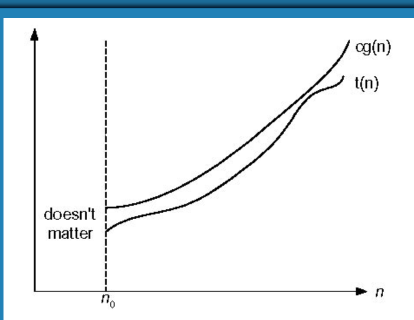

Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

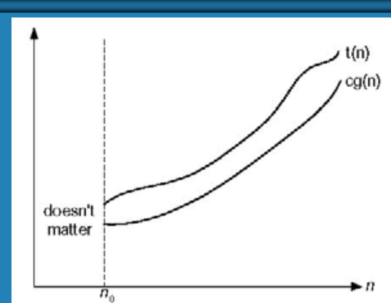

Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

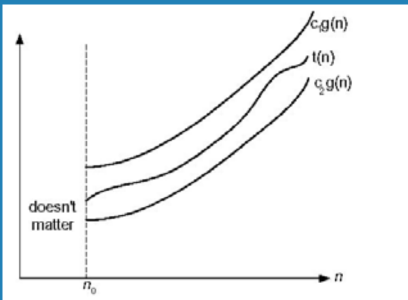

Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing rate of growth: Method 1 – using limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) \text{ \_\_\_ order of growth of } g(n) \\ c>0 & \text{order of growth of } T(n) \text{ \_\_\_ order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) \text{ \_\_\_ order of growth of } g(n) \end{cases}$$

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

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

- Example: $\log n$ vs. n

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

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

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 ← 0 to n   1 do
    for j ← 0 to n   1 do
        C[i, j] ← 0.0
        for k ← 0 to n   1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```


Selection sort

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


Insertion sort

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


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 <u>*input size*</u>
- Identify algorithm's <u>*basic operation*</u>
- Determine <u>*worst*</u>, <u>*average*</u>, and <u>*best*</u> 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.
  $T(n) = T(n-1) + c$ $\quad T(1) = d$
  Solution: $T(n) = (n-1)c + d$ <u>*linear*</u>
- A pass through input reduces problem size by one.
  $T(n) = T(n-1) + cn$ $\quad T(1) = d$
  Solution: $T(n) = [n(n+1)/2 - 1]\, c + d$ <u>*quadratic*</u>
- One (constant) operation reduces problem size by half.
  $T(n) = T(n/2) + c$ $\quad T(1) = d$
  Solution: $T(n) = c \lg n + d$ <u>*logarithmic*</u>
- A pass through input reduces problem size by half.
  $T(n) = 2T(n/2) + cn$ $\quad T(1) = d$
  Solution: $T(n) = cn \lg n + d\, n$ <u>*n log n*</u>

A general divide-and-conquer recurrence

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

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

<u>Note:</u> the same results hold with O instead of Θ.

Fibonacci numbers

- The Fibonacci sequence:
  0, 1, 1, 2, 3, 5, 8, 13, 21, …
- Fibonacci recurrence:
  $F(n) = F(n-1) + F(n-2)$
  $F(0) = 0$
  $F(1) = 1$

  2nd order linear homogeneous recurrence relation with constant coefficients
- Another example:
  $A(n) = 3A(n-1) - 2(n-2)$ $\quad A(0) = 1$ $\quad A(1) = 3$

Solving linear homogeneous recurrence relations with constant coefficients

- Easy first: 1st order LHRRCCs:
  $C(n) = a\, C(n-1)$ $\quad C(0) = t$ … Solution: $C(n) = t\, a^n$
- Extrapolate to 2nd order
  $L(n) = a\, L(n-1) + b\, L(n-2)$ … A solution?: $L(n) = r^n$
- Characteristic equation (quadratic)
- Solve to obtain roots r_1 and r_2—e.g.: $A(n) = 3A(n-1) - 2(n-2)$
- General solution to RR: linear combination of r_1^n and r_2^n
- Particular solution: use initial conditions—e.g.: $A(0) = 1$ $\quad A(1) = 3$

Computing Fibonacci numbers

1. Definition based recursive algorithm
2. Nonrecursive brute-force algorithm
3. Explicit formula algorithm
4. Logarithmic algorithm based on formula:

$$\begin{pmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

- for $n \geq 1$, assuming an efficient way of computing matrix powers.