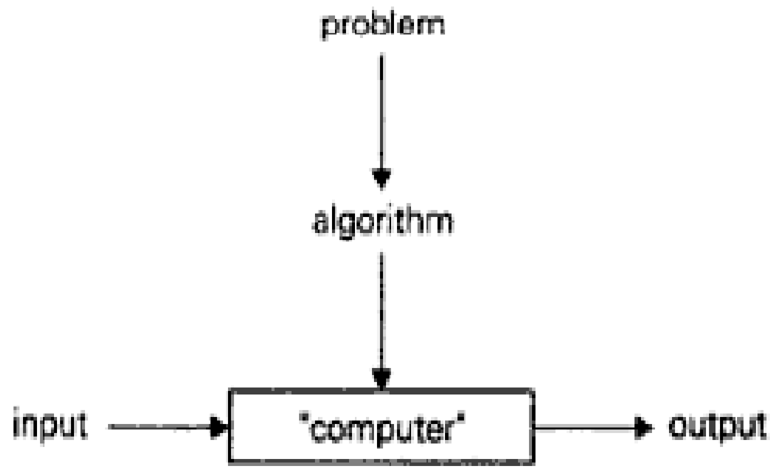
Introduction

Notion of Algorithm:



Example: Find the GCD of two integer number:

Euclid's algorithm for computing gcd(m, n)

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in a pseudocode:

Method-1:

ALGORITHM Euclid(m, n)

//Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while $n \neq 0$ **do**
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return m

Method-2:

Consecutive integer checking algorithm for computing gcd(m, n)

Step 1 Assign the value of min{m, n} to t.

Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

Method-3:

Middle-school procedure for computing $\gcd(m, n)$

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

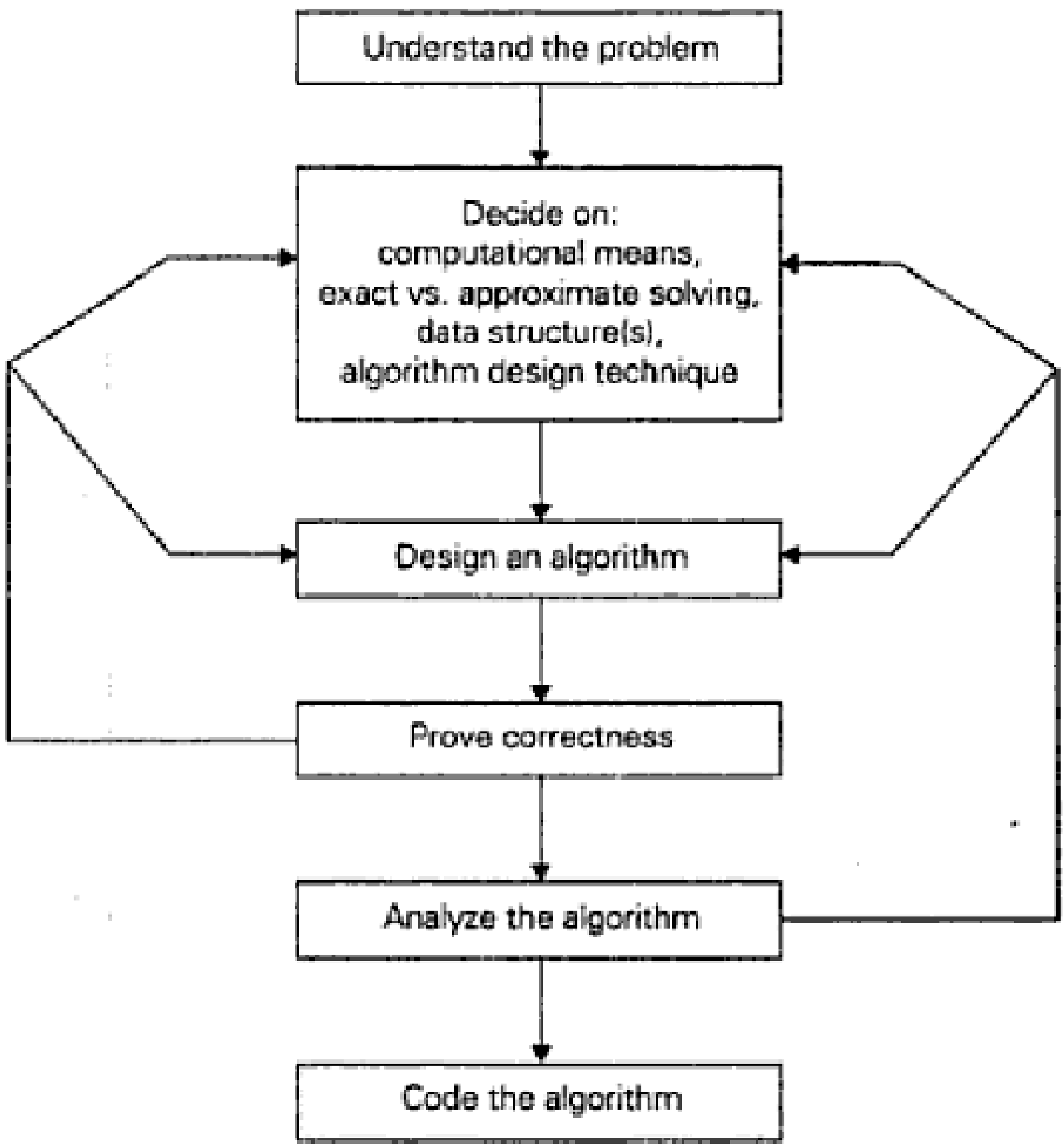
Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Finding Prime numbers less than or equal to N:

```
ALGORITHM Sieve(n)
    //Implements the sieve of Eratosthenes
    //Input: An integer n ≥ 2
    //Output: Array L of all prime numbers less than or equal to n
    for p ← 2 to n do A[p] ← p
    for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
        if A[p] ≠ 0    //p hasn't been eliminated on previous passes
            j ← p * p
            while j ≤ n do
                A[j] ← 0    //mark element as eliminated
                j ← j + p
    //copy the remaining elements of A to array L of the primes
    i ← 0
    for p ← 2 to n do
        if A[p] ≠ 0
            L[i] ← A[p]
            i ← i + 1
    return L
```

Algorithm Design Process:

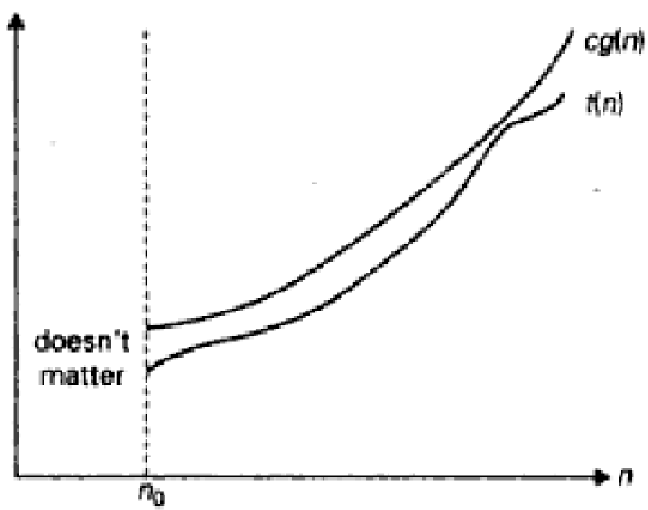


Asymptotic Notations

O-notation

DEFINITION 1 A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

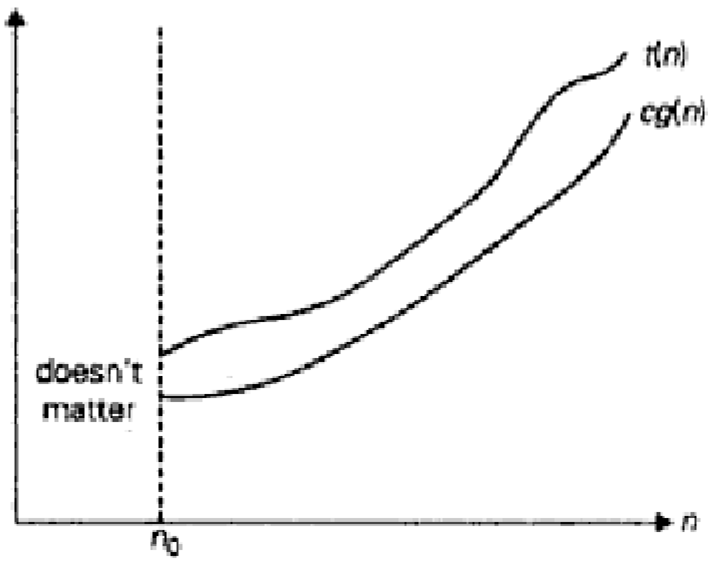
$$t(n) \le cg(n) \quad \text{for all } n \ge n_0.$$



Ω-notation

DEFINITION 2 A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

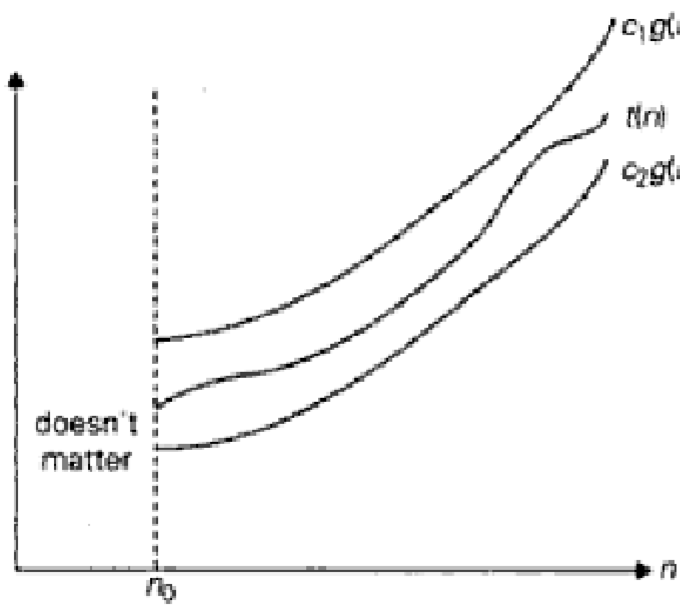
$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$



Θ-notation

DEFINITION 3 A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \le t(n) \le c_1 g(n) \quad \text{for all } n \ge n_0.$$



Using Limits for Comparing Orders of Growth

$$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & \text{implies that } t(n) \text{ has the same order of growth as } g(n) \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n}) = \frac{1}{2}.$$

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty}\frac{\sqrt{n}}{n} = 0.$$

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this issue informally in the previous section.) Taking advantage of Stirling's formula, we get

$$\lim_{n\to\infty}\frac{n!}{2^n} = \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty.$$

Efficiency Classes:

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Mathematical Analysis of Algorithms:

1.Nonrecursive Algorithms

General Plan for Analyzing Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.[4]
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

ALGORITHM MaxElement($A[0..n-1]$)

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
maxval ← $A[0]$
for i ← 1 **to** $n-1$ **do**
    if $A[i] >$ *maxval*
        maxval ← $A[i]$
return *maxval*

ALGORITHM UniqueElements($A[0..n-1]$)

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//         and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

ALGORITHM *MatrixMultiplication*($A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$)

```
//Multiplies two n-by-n matrices 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
```

ALGORITHM *Binary(n)*

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

2. Recursive Algorithms

General Plan for Analyzing Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

ALGORITHM *BinRec*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
else return BinRec(⌊n/2⌋) + 1
```

ALGORITHM *Fib*(n)

```
//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
F[0] ← 0; F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

ALGORITHM *SelectionSort*($A[0..n-1]$)

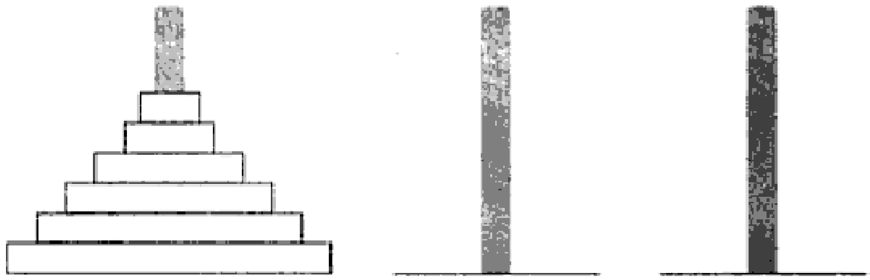
```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
        for j ← i + 1 to n − 1 do
         if A[j] < A[min]    min ← j
     swap A[i] and A[min]
```

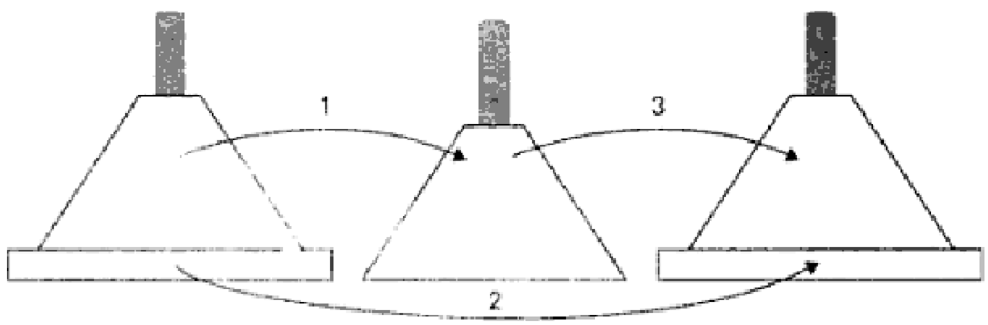
```
ALGORITHM  BubbleSort(A[0..n − 1])
    //Sorts a given array by bubble sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in ascending order
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if A[j + 1] < A[j]  swap A[j] and A[j + 1]
```

INSERTION-SORT(*A*)	*cost*	*times*
for $j \leftarrow 2$ **to** *length*[*A*]	c_1	n
do $key \leftarrow A[j]$	c_2	$n - 1$
▹ Insert $A[j]$ into the sorted sequence $A[1$ □ $j - 1]$.	0	$n - 1$
$i \leftarrow j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n - 1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}
$$

Tower of Hanoi Problem:





$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1,$$
$$M(1) = 1.$$

```
void towers(int n1,char source,char dest,char aux)
{
  if(n1==1)
  {
    printf("step%d:move%d from %c to %c\n",++count,n1,source,dest);
  }
  else
  {
    towers(n1-1,source,aux,dest);
    printf("step%d:move%d from %c to %c\n",++count,n1,source,dest);
    towers(n1-1,aux,dest,source);
  }
}
```