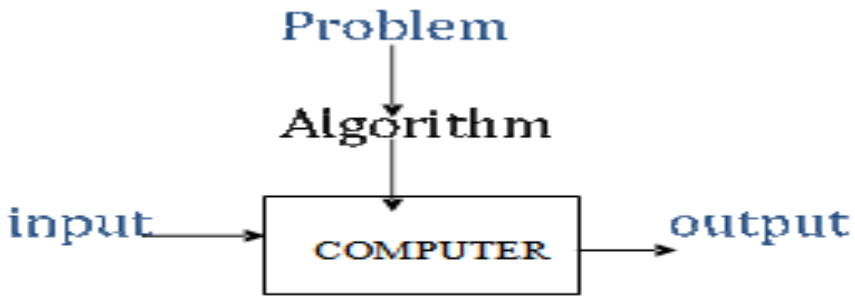
Unit 1 - Introduction

1.1 Algorithm

An algorithm is finite set of instructions that is followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and produced.
4. Finiteness. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principal, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.



An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

The fourth criterion for algorithms we assume in this book is that they terminate after a finite number of operations.

- Algorithms that are definite and effective are also called computational procedures.
- The same algorithm can be represented in several ways.
- Several algorithms to solve the same problem.
- Different ideas different speed.

Example:

Problem:GCD of Two numbers m,n

Input specifiastion :Two inputs,nonnegative,not both zero

Euclids algorithm

-gcd(m,n)=gcd(n,m mod n)

Untill m mod n =0,since gcd(m,0) =m

Another way of representation of the same algorithm

Euclids algorithm

Step1:if n=0 return val of m & stop else proceed step 2

Step 2:Divide m by n & assign the value of remainder to r

Step 3:Assign the value of n to m,r to n,Go to step1.

Another algorithm to solve the same problem

Euclids algorithm

Step1:Assign the value of min(m,n) to t

Step 2:Divide m by t.if remainder is 0,go to step3 else goto step4

Step 3: Divide n by t.if the remainder is 0,return the value of t as the answer and stop,otherwise proceed to step4

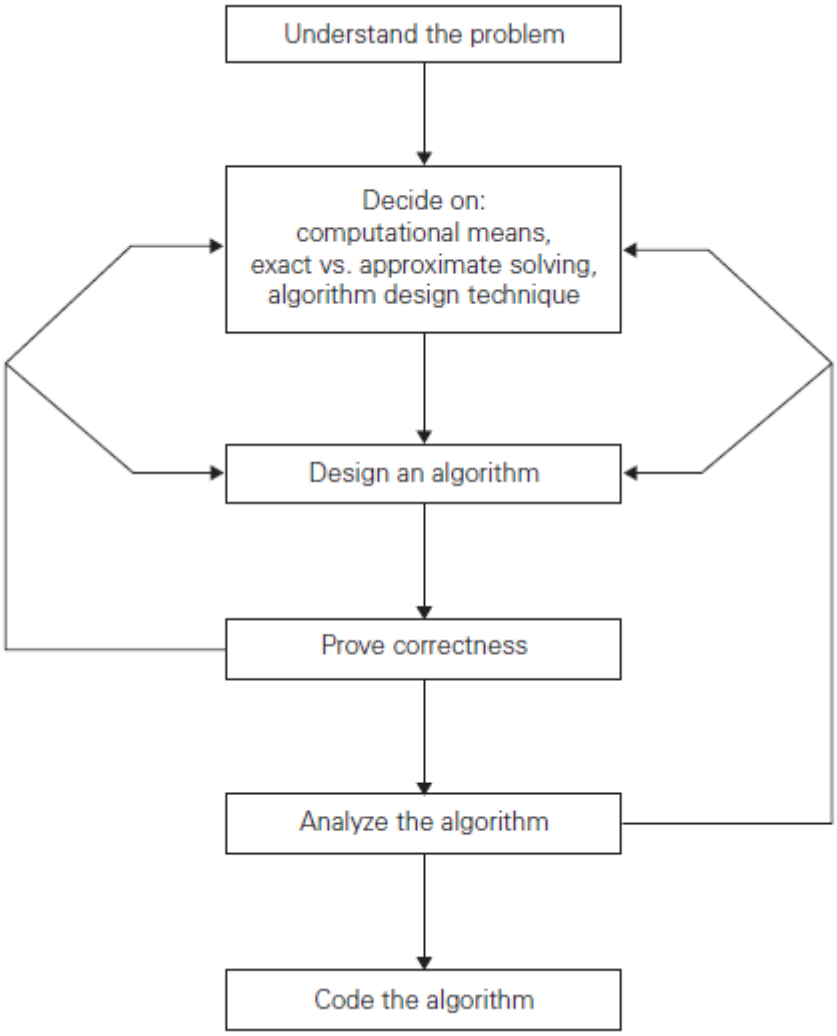
Step4 :Decrease the value of t by 1. go to step 2

1.2 Fundamentals of Algorithmic Problem Solving

Understanding the Problem: From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problems description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases? An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.

Ascertaining the Capabilities of the Computational Device: Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903–1957), in collaboration with A. Burks and H. Goldstine, in 1946. The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***). Its central

assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.



Choosing between Exact and Approximate Problem Solving: The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity. This happens, in particular, for many problems involving a very large number of choices;

Algorithm Design Techniques: Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem? This is the main question this book seeks to answer by teaching you several general design techniques. An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. Check this book's table of contents and you will see that a majority of its chapters are devoted to individual design techniques. There till a few key ideas that have proven to be useful in designing algorithms. Learning these techniques is of utmost importance for the following reasons.

First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm. Therefore—to use the language of a famous proverb—learning such techniques is akin to learning to fish as opposed to being given a fish caught by somebody else. It is not true, of course, that each of these general techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work.

Designing an Algorithm and Data Structures: While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques. Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part of the algorithm designer. With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy.

Methods of Specifying an Algorithm: Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give you an example, Euclid's algorithm is described in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms. Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

Proving an Algorithm's Correctness: Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n)* (which, in turn, needs a proof; see Problem 7 in Exercises 1.1), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0. For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.Acommon technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

Analyzing an Algorithm: We usually want our algorithms to possess several qualities. After correctness, by far the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses. A general framework and specific techniques for analyzing an algorithm's efficiency appear in Chapter 2. Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. For example, most people would agree that Euclid's algorithm is simpler than the middle-school procedure for computing gcd*(m, n)*, but it is not clear whether Euclid's algorithm is simpler than the consecutive integer checking algorithm. Still, simplicity is an important algorithm characteristic to strive for.Why? Because simpler algorithms are easier to understand and easier to program; consequently, the resulting programs usually contain fewer bugs. There is also the undeniable aesthetic appeal of simplicity. Sometimes simpler algorithms are also more efficient than more complicated alternatives. Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

1.3 Review of Asymptotic Notation: Fundamentals of the analysis of algorithm efficiency

 Analysis of algorithms means to investigate an algorithm‘s efficiency with respect to resources:

 running time (time efficiency)

 memory space (space efficiency)

Time being more critical than space, we concentrate on Time efficiency of algorithms. The theory developed, holds good for space complexity also.

Experimental Studies: requires writing a program implementing the algorithm and running the program with inputs of varying size and composition. It uses a function, like the built-in clock() function, to get an accurate measure of the actual running time, then analysis is done by plotting the results.

Limitations of Experiments

• It is necessary to implement the algorithm, which may be difficult

• Results may not be indicative of the running time on other inputs not included in the experiment.

• In order to compare two algorithms, the same hardware and software environments must be used.

Theoretical Analysis: It uses a high-level description of the algorithm instead of an implementation. Analysis characterizes running time as a function of the input size, *n*, and takes into account all possible inputs. This allows us to evaluate the speed of an algorithm independent of the hardware/software environment. Therefore, theoretical analysis can be used for analysing any algorithm.

Framework for Analysis

We use a hypothetical model with following assumptions

• Total time taken by the algorithm is given as a function on its input size

• Logical units are identified as one step

• Every step requires ONE unit of time

• Total time taken = Total Num. of steps executed.

Input’s size: Time required by an algorithm is proportional to size of the problem instance. For e.g., more time is required to sort 20 elements than what is required to sort 10 elements.

Units for Measuring Running Time: Count the number of times an algorithm‘s **basic operation** is executed. (**Basic operation**: The most important operation of the algorithm, the operation contributing the most to the total running time.) For e.g., The basic operation is usually the most time-consuming operation in the algorithm ‘s innermost loop.

Order of Growth: For order of growth, consider only the leading term of a formula and ignore the constant coefficient. The following is the table of values of several functions important for analysis of algorithms.

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}		

Worst-case, Best-case, Average case efficiencies

Algorithm efficiency depends on the **input size n.** And for some algorithms efficiency depends on **type of input**. We have best, worst & average case efficiencies.

• **Worst-case efficiency:** Efficiency (number of times the basic operation will be executed) **for the worst case input of size n.** *i.e.* The algorithm runs the longest among all possible inputs of size n.

• **Best-case efficiency:** Efficiency (number of times the basic operation will be executed) **for the best case input of size n.** ***i.e.*** The algorithm runs the fastest among all possible inputs of size n.

• **Average-case efficiency:** Average time taken (number of times the basic operation will be executed) **to solve all the possible instances (random) of the input.** NOTE: NOT the average of worst and best case.

1.4 Asymptotic Notations

Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm‘s basic operation count are:

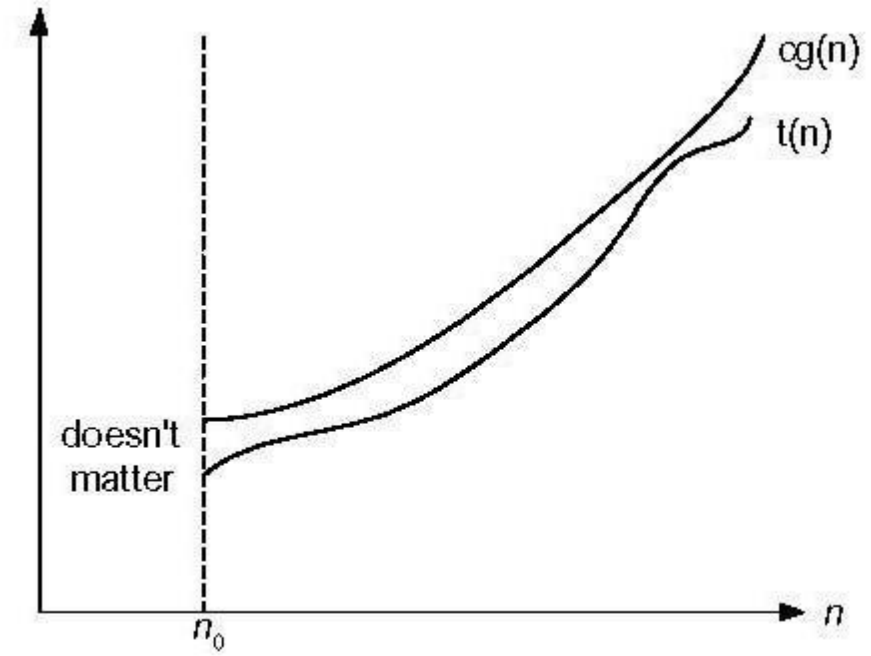
Big Oh- O notation

Definition:

A function t(n) is said to be in O(g(n)), denoted **t(n) ∈ 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 n0 such that

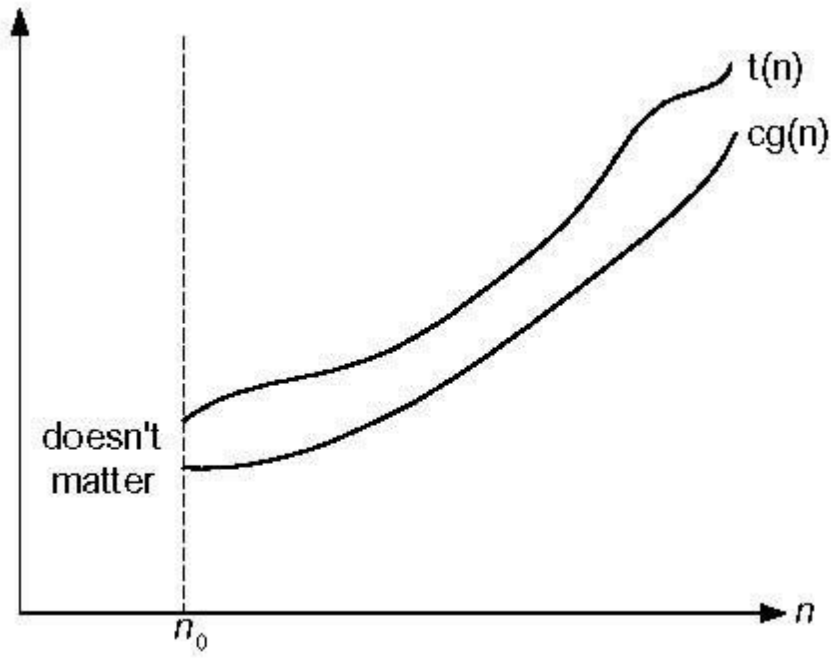
t(n) ≤ cg(n) for all n ≥ n0



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

Big Omega- Ω notation

A function t (n) is said to be in Ω (g(n)), denoted **t(n) ∈ Ω (g (n)),** if t (n) is bounded below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that **t(n) ≥ cg(n) for all n ≥ n0.**

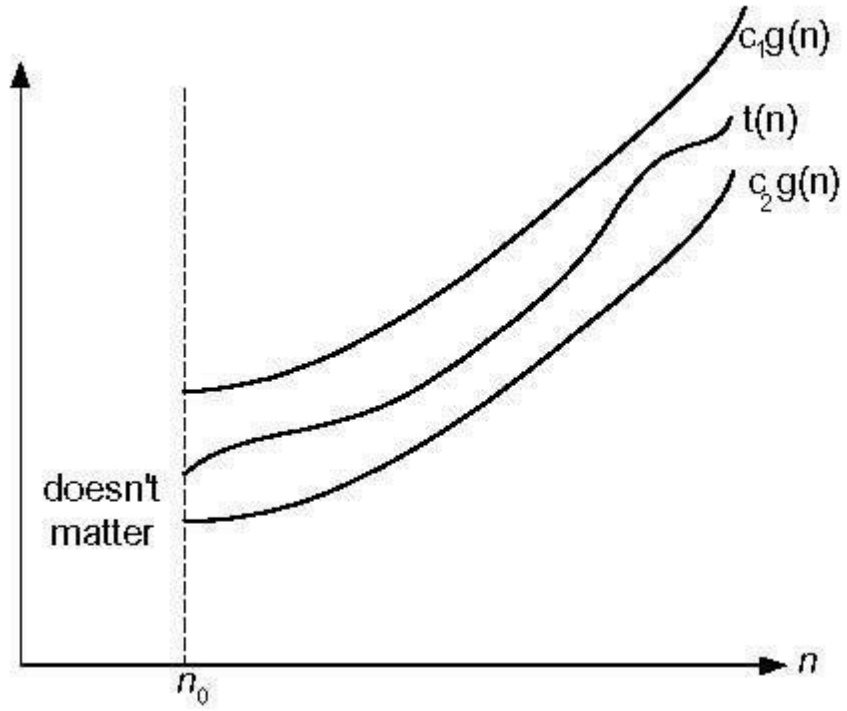


Big-omega notation: $t(n) \in \Omega(g(n))$

Big Theta- Θ notation:

A function t (n) is said to be in Θ(g (n)), denoted **t(n) ∈ Θ(g (n)),** if t (n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

c2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0



1.5 Mathematical Analysis of Non-Recursive and Recursive Algorithms:

Mathematical analysis (Time Efficiency) of Non-recursive Algorithms

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating **input size**
2. Identify algorithm ‘s **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best-case efficiency** separately.
4. Set up **summation** for C(n) reflecting the number of times the algorithm ‘s basic operation is executed.
5. Simplify summation using standard formulas.

Example: Finding the largest element in a given array

ALOGORITHM MaxElement(A[0..n-1])

//Determines the value of 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

currentMax ← A[0]

for i ← 1 to n - 1 do

if A[i] > currentMax

currentMax ← A[i]

return currentMax

Analysis:

1. Input size: number of elements = n (size of the array)
2. Basic operation:

a) Comparison

b) Assignment

3. NO best, worst, average cases.
4. Let C (n) denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop‘s variable within the bound between 1 and $n - 1$.

$$C(n) = \sum_{i=1}^{n-1} 1$$

5. **Simplify summation** using standard formulas

$$C(n) = \sum_{i=1}^{n-1} 1 \qquad \begin{matrix} 1+1+1+\ldots+1 \\ \textbf{[(n-1) number of times]} \end{matrix}$$

$$C(n) = n-1$$

$$C(n) \square \Theta(n)$$

Example: Element uniqueness problem

Algorithm UniqueElements (A[0..n-1])

//Checks 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**

Analysis

1. Input size: number of elements = n (size of the array)

2. Basic operation: Comparison

3. Best, worst, average cases EXISTS.

Worst case input is an array giving largest comparisons.

• Array with no equal elements

• Array with last two elements are the only pair of equal elements

4. Let C (n) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop‘s variable j between its limits i + 1 and n – 1; and this is repeated for each value of the outer loop i.e, for each value of the loop‘s variable i between its limits 0 and n – 2.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

5. **Simplify summation** using standard formulas

$$C(n) = \sum_{i=0}^{n-2} ((n-1) - (i+1) + 1)$$

$$C(n) = \sum_{i=0}^{n-2} (n-1-i)$$

$$C(n) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$C(n) = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$C(n) = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$C(n) = (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$C(n) = (n-1)((n-1) - \frac{(n-2)}{2})$$

$$C(n) = (n-1)\frac{(2n-2-n+2)}{2}$$

$$C(n) = (n-1)(n)/2$$
$$= (n^2 - n)/2$$
$$= (n^2)/2 - n/2$$

$C(n) \in \Theta(n^2)$

EXAMPLE 3 Given two $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:

A B C

col. j

row i $C[i, j]$

* =

where $C[i, j] = A[i, 0]B[0, j] + \ldots + A[i, k]B[k, j] + \ldots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

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 $n \times n$ 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

Analysis

1. Input size: number of elements = n (size of the array)

2. Basic operation: Multiplication

3.

$$\sum_{k=0}^{n-1} 1,$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

EXAMPLE 4 The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

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*

Analysis

1. Input size: number of elements = n

2. Basic operation: Division

3. Complexity : $\log_2 n$

1.6. Mathematical Analysis of Recursive Algorithms:

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating **input size**

2. Identify algorithm's **basic operation**

3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.

4. Set up **recurrence relation**, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.

5. **Solve** the recurrence.

Example: Factorial function

ALGORITHM *Factorial (n)*

//Computes n! recursively

//Input: A nonnegative integer n

//Output: The value of n!

if *n* = = 0

return 1

else

return Factorial (*n* – 1) * *n*

Analysis:

1. Input size: given number = n

2. Basic operation: multiplication

3. NO best, worst, average cases.

4. Let M (*n*) denotes number of multiplications.

M (*n*) = M (*n* – 1) + 1 for *n* > 0

M (0) = 0 *initial condition*

Where: M (*n* – 1) : to compute Factorial (*n* – 1)

1 : to multiply Factorial (*n* – 1) by *n*

Solve the recurrence: Solving using *"Backward substitution method":*

M (*n*) = M (*n* – 1) + 1

= [M (*n* – 2) + 1] + 1

= M (*n* – 2) + 2

= [M (*n* – 3) + 1] + 3

= M (*n* – 3) + 3

…

In the ith recursion, we have

= M (*n* – *i*) + *i*

When i = n, we have

= M (*n* – *n*) + *n* = M (0) + *n*

Since M (0) =0 = **n**

M (*n*) ∈ Θ (n).

Example: Find the number of binary digits in the binary representation of a positive decimal integer

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.

Analysis:

1. Input size: given number = n

2. Basic operation: addition

3. NO best, worst, average cases.

4. Let A (*n*) denotes number of additions.

A (*n*) = A (⌊ *n/2* ⌋) + 1 for *n* > 1

A (1) = 0 *initial condition*

Where: A (⌊ *n/2* ⌋) : to compute *BinRec* (⌊ *n/2* ⌋)

1 : to increase the returned value by 1

5. Solve the recurrence:

A (*n*) = A (⌊ *n/2* ⌋) + 1 for *n* > 1

Assume n = 2k (smoothness rule)

A (2k) = A (2k-1) + 1 for *k* > 0; A (20) = 0

Solving using "Backward substitution method":

A (2k) = A (2k-1) + 1

= [A (2k-2) + 1] + 1

= A (2k-2) + 2

= [A (2k-3) + 1] + 2

= A (2k-3) + 3.

In the ith recursion, we have

= A (2k-i) + i

When i = k, we have

= A (2k-k) + k = A (20) + k

Since A (20) = 0

A (2k) = k

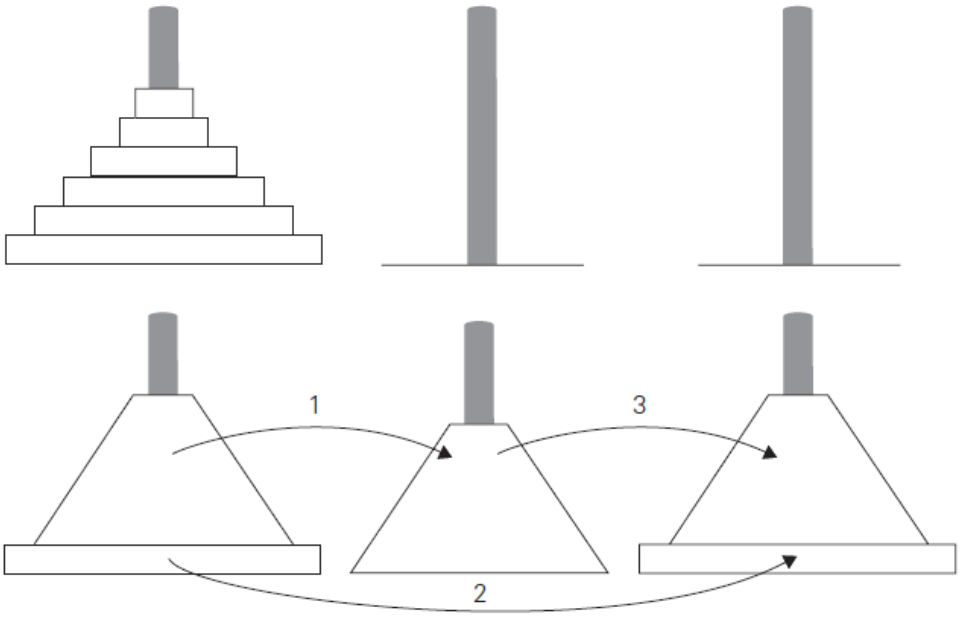
Since n = 2k, HENCE *k* = log2 *n*

A (*n*) = log2 *n*

A (*n*) ∈ Θ (log n)

EXAMPLE 3 As our next example, we consider another educational workhorse of recursive algorithms: the ***Tower of Hanoi*** puzzle. In this puzzle, we (or mythical monks, if you do not like to move disks) have n disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

The problem has an elegant recursive solution, which is illustrated in Figure. To move $n>1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n-1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n-1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if $n=1$, we simply move the single disk directly from the source peg to the destination peg.



Let us apply the general plan outlined above to the Tower of Hanoi problem.
The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:
$M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$.
With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$M(n) = 2M(n-1) + 1$ for $n > 1$, **(2.3)**

$M(1) = 1$.

We solve this recurrence by the same method of backward substitutions:

$M(n) = 2M(n-1) + 1$ sub. $M(n-1) = 2M(n-2) + 1$

$= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$

$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1$.

The pattern of the first three sums on the left suggests that the next one will be $2^4M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2 + 1 = 2^iM(n-i) + 2^i - 1$.

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1$

$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = \mathbf{2^n - 1}$.

1.7 Brute Force Approaches:

Introduction

Brute force is a straightforward approach to problem solving, usually directly based on the problem 's statement and definitions of the concepts involved. Though rarely a source of clever or

efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. Unlike some of the other strategies, **brute force** is applicable to a very wide variety of problems. For some important problems (e.g., sorting, searching, string matching),the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. A brute-force algorithm can serve an important theoretical or educational purpose.

Selection Sort

Problem:Given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order.

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]

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Selection sort's operation on the list 89, 45, 68, 90, 29, 34, 17. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

Thus, selection sort is a O*(n*2*)* algorithm on all inputs. The number of key swaps is only O*(n)* or, more precisely, *n*-1 (one for each repetition of the *i* loop). This property distinguishes selection sort positively from many other sorting algorithms.

Sequential Search/Linear Search

ALGORITHM *SequentialSearch2(A*[0..*n*], *K)*

//The algorithm implements sequential search with a search key as a sentinel

//Input: An array *A* of *n* elements and a search key *K*

//Output: The position of the first element in *A*[0..*n* - 1] whose value is

// equal to *K* or -1 if no such element is found

A[*n*]=*K*

i=0

while *A*[*i*] = *K* **do**

i=*i* + 1

if *i* < *n* **return** *i*

else return -1

Best Case : 1

Avearge case : n/2

Worst case : n.