RNSIT-CSE
Bengaluru

DESIGN AND ANALYSIS OF ALGORITHMS

18CS42

MODULE-1

INTRODUCTION

1.1 What is an Algorithm? **(T2:1.1)**

1.2 Algorithm Specification **(T2:1.2)**

1.3 Analysis Framework **(T1:2.1)**

1.4 **Performance Analysis**: Space complexity, Time complexity **(T2:1.3)**.

1.5 **Asymptotic Notations (T1:2.2, 2.3, and 2.4)**

- 1.5.1 Big-Oh notation (O)
- 1.5.2 Omega notation (Ω)
- 1.5.3 Theta notation (Θ)
- 1.5.4 Mathematical analysis of Non-Recursive and recursive Algorithms with Examples.

1.6 **Important Problem Types:**

- 1.6.1 Sorting, Searching
- 1.6.2 String processing
- 1.6.3 Graph Problems
- 1.6.4 Combinatorial Problems.

1.7 **Fundamental Data Structures**

- 1.7.1 Stack
- 1.7.2 Queues
- 1.7.3 Graphs
- 1.7.4 Trees
- 1.7.5 Sets and Dictionaries

1.1 What is an Algorithm?

The word algorithm comes from the name of a Persian author, Abu jafar mohammed ibn musa al khowarizmi (825A.D) who wrote a text book on mathematics.

Definition: "An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time".

(Or)

An algorithm is a finite set of instructions that if followed accomplishes a particular task.

All algorithm must satisfy the following criteria:

i. **Input** There are zero or more quantities which are externally supplied.

ii. **Output** At least one quantity is produced.

iii. **Definiteness** Each instruction must be clear and unambiguous.

iv. **Finiteness** If we trace out the instructions of the algorithm, then for all valid cases the algorithm will terminate after a finite number of steps.

v. **Effectiveness** Every instruction must be sufficiently basic that it can in principle be carried out by a person using only a pencil and paper. It is not enough that each operation be definite as in (iii), but it must be feasible

Algorithms that are definite and effective are also called computational procedures. A program is the expression of an algorithm in a programming language. The diagrammatic representation of algorithm is given as notion of an algorithm below:

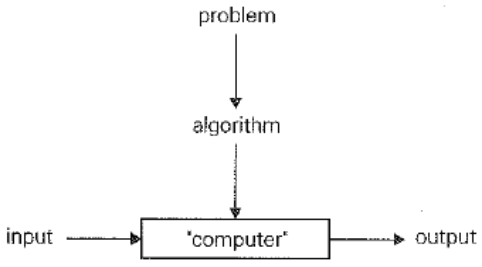


Figure 1.1: **Notion of an algorithm**

The **important properties of the algorithm** are

- The algorithm should be unambiguous.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

The **study of algorithms** includes four distinct areas.

1. **How to devise algorithms**: Mastering various design strategies helps to devise new algorithms.
2. **How to validate algorithms**: checking the algorithm, whether it gives the correct answer for all possible inputs. After validation the program can be written.
3. **How to analyze algorithms**: analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.
4. **How to test a program**: testing a program in two phases debugging and profiling (performance, measurement).
    a. Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and so correct them.
    b. Profiling is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

1.2 Algorithm specification

i) Pseudo code conventions: we can describe an algorithm in natural language like English etc. Graphics/Flowcharts is also an another method for representing.

Pseudo code conventions are:

- Comments begin with // and continue with the end of the line.
- Blocks are indicated with matching braces { and } .A compound statement can be represented as a block. An identifier begins with a letter. The data types are not explicitly declared. Compound data types can be formed with records.

    Example: node=record

    {

     data type data1

data type2 data 2

:

:

:

data type 1 data 1

node * link;

}

This is a self referential structure, data items of a record can be accessed with-> and period (.)

- Assignment of values to variables is done using the assignment statement.

<variable>:=<expression>;

- Boolean values true and false are used.
- Logical operators and, or,and not, relational operators < ,<=,>,>= are also supported.
- Conditional looping statements has the following forms

  If<condition> then <statement>

  If <condition>then<statemenmt1> else<statement2>

- We can use case statement

  Case

  {

  :<conditon1>:<statement>1>

  : <conditon1>:<statement>2>

  :

  :

  :<conditonn>:<statement>n>

  :else:<statement n+1>

  }

  Here <statement> can be simple or compound.

- Input and output are specified by read and write.
- Algorithm consists of a heading and a body

  Syntax is

  Algorithm name (<parameter lists>)

  Where name is the name of the procedure/algorithm and <parameter list> is the list of parameters.

1.3 The Analysis Framework

Analysis of algorithms means investigation of algorithm efficiency with respect to two resources:
-Running Time and Memory Space.

Analysis framework is a systematic approach that can be applied for analyzing the efficiency through:

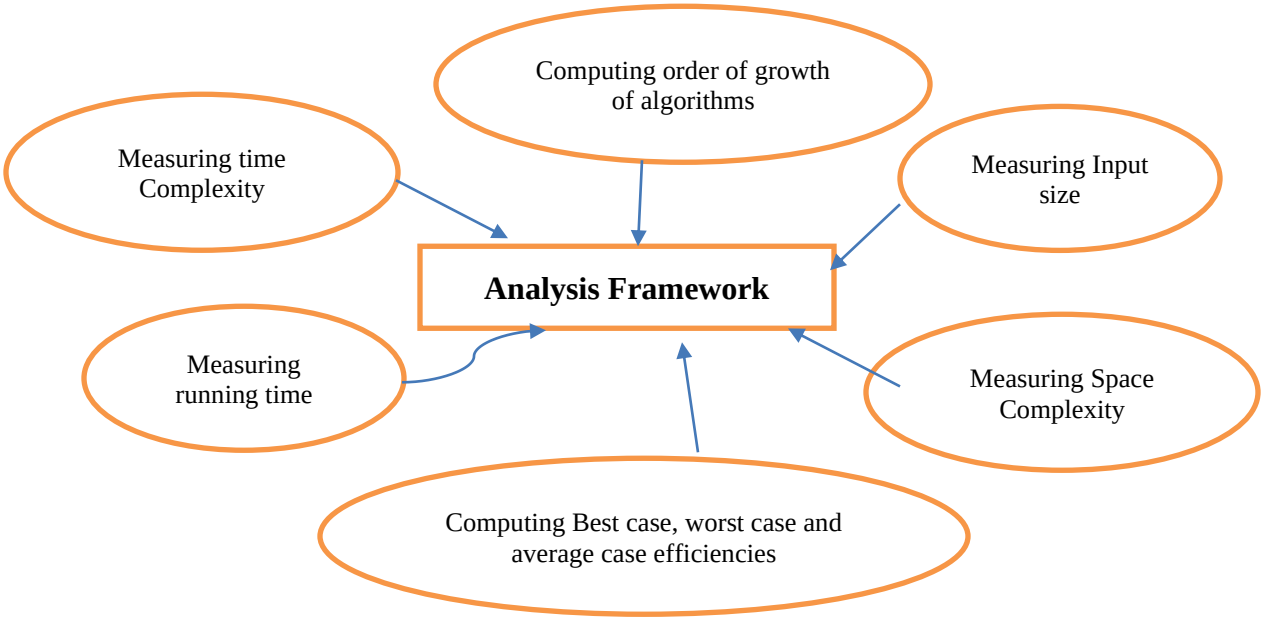
- **Time efficiency**
- **Space efficiency.**

Time efficiency also called **time complexity** indicates how fast an algorithm in question runs.

Space efficiency also called **space complexity** refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

Important factors for Analysis Framework are:

- **Measuring an Input's Size.**
- **Units for Measuring Running Time**
- **Orders of Growth**
- **Worst-Case, Best-Case, and Average-Case Efficiencies**



1. Measuring an Input's Size:

For an algorithm, the parameter 'n' specifies the input size. A common observation is that "All algorithms run longer on larger inputs". Therefore it is logical to investigate an algorithm's input efficiency as a function of parameter "n" indicating input size.

Selecting the size of input vary with respect to the type of problem. In many cases selecting 'n' is straightforward. Example: Searching, Sorting etc...

In some varieties of problems for example: Product of matrices, Diagnose disease from X-Ray, 'n' cannot be directly estimated. For such algorithms, measuring size is

$$b = [\log_2 n] + 1$$

Where b= number of bits, n= input parameters.

This metric usually gives a better idea about the efficiency of algorithms.

2. Units for Measuring Running Time

An algorithm's efficiency must be measured with a metric that does not on extraneous factors like computer used, compiler used to run the program.

One possible approach is to count the number of times each of the algorithm's operations is executed. T**he most important component used to measure the running time of the algorithm is called the *basic operation.***

Identification of the basic operation of an algorithm: **it is usually the most time-consuming operation in the algorithm's innermost loop.**

For example, most **sorting algorithms** work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison.

Let *cop* be the execution time of an algorithm's basic operation on a particular computer, and let *C(n)* be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time *T (n)* of a program implementing this algorithm on that computer by the formula

$$T(n) \approx cop * C(n)$$

Problem: How much longer will the algorithm run if we double its input size? Assume C(n) is ½ n(n+1) .

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

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

The answer is 4 times longer

3. Orders of Growth

Order of growth of an algorithm is a way of predicting the execution time of a program changes with input size.

The varying of running time with increase in input size is the order of growth of algorithm. The magnitude of the numbers in below table has significance for the analysis of algorithms. There are seven efficiency classes listed row wise in the Table2.1.

TABLE 2.1 Values (some approximate) 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}		

The function growing the slowest among these is the logarithmic function. The exponential function 2^n and the factorial function $n!$ grow so fast that their values become astronomically large even for rather small values of n. Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

4. Worst-Case, Best-Case, and Average-Case Efficiencies

Many algorithms running time depend not only on an input size but on specific input. Consider, as an example, sequential search. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted. The running time of this algorithm can be quite different for same list size n.

ALGORITHM *SequentialSearch(A*[0..$n-1$], *K)*

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element in A that matches K
// or −1 if there are no matching elements
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
    if $i < n$ **return** i
else return −1

Worst case efficiency: when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$C_{worst}(n) = n.$$

The *worst-case efficiency* of an algorithm is its efficiency for the worst-case input of size *n*, which is an input (or inputs) of size *n* for which the algorithm runs the longest among all possible inputs of that size.

The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

The *best-case efficiency* of an algorithm is its efficiency for the best-case input of size *n*, which is an input (or inputs) of size *n* for which the algorithm runs the fastest among all possible inputs of that size.

Accordingly, we can analyze the best case efficiency as follows. First, we determine the kind of inputs for which the count *C(n)* will be the smallest among all possible inputs of size *n*. For example, the best-case inputs for sequential search are lists of size *n* with their first element equal to a search key; accordingly, $C_{best}(n) = 1$ for this algorithm.

The *average-case efficiency* of an algorithm is average time taken (number of times the basic operation will be executed) to solve all the possible instances of the input.

The ***average-case efficiency:*** neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input.

1.4 Performance Analysis (TB-2-1.3)

Performance analysis is the criteria for judging algorithms. The two important criterias's are namely:

- Space Complexity.
- Time Complexity

***Space complexity:** The space complexity of an algorithm is the amount of memory it needed to run to completion.

The space needed by each of these algorithms is the sum of the following components

i) **Fixed Part**: is the aspect of an algorithm that is independent of the characteristics of the input and outputs.

Example: Number, size, Space for simple variables, fixed component variables, Space for constant etc.

ii) **A Variable Part**: consists of the space needed by the components variables whose size is dependent on the particular problem instance being solved.

Example: The space needed by referenced variables recursion stack space.

Therefore the space requirement S(P) of any algorithm P may be given as below

S(P)=C+ Sp (instance characteristics),

Where C is a constant, Sp is an instance characteristics which is the main focus in analyzing the space complexity of any algorithm. Therefore Sp is always problem specific.

Example1

Algorithm abc (Algorithm 1.5) computes $a+b+b*c+(a+b-c)/(a+b)+4.0$; Algorithm Sum (Algorithm 1.6) computes $\sum_{i=1}^{n} a[i]$ iteratively, where the $a[i]$'s are real numbers; and RSum (Algorithm 1.7) is a recursive algorithm that computes $\sum_{i=1}^{n} a[i]$.

```
1   Algorithm abc(a, b, c)
2   {
3       return  a + b + b * c + (a + b - c)/(a + b) + 4.0;
4   }
```

Here we see the values a, b, c is independent of the instance characteristics, so Sp =0.

2. **Algorithm to compute the sum of n numbers:**

Iterative version of Sum algorithm: The space needed by n is one word, since it is of type integer so,

```
1   Algorithm Sum(a, n)
2   {
3       s := 0.0;
4       for i := 1 to n do
5           s := s + a[i];
6       return s;
7   }
```

S(P)= C+ Sp

C=s, n, i =3 (1 word per variable)

Sp= a[i]= n

S(P) = 3+n

S_{sum} (n)>= (n+3)

Recursive Algorithm to compute sum of n numbers.

```
1  Algorithm RSum(a, n)
2  {
3      if (n ≤ 0) then return 0.0;
4      else return RSum(a, n − 1) + a[n];
5  }
```

In recursive algorithms the instances are characterized by the n. The recursion stack needs space which in turn space is used to store formal parameters, local variables and the return address. So each call to Rsum requires at least 3 words.

- Space for value a[n]
- Return address
- Pointer to a[ ]

Since the depth of the recursion i.e. how many times recursion is called is n+1.

S(P)= 3(n+1)

*Time complexity:

The time T (P) taken by a program P is the sum of the compile time and the run time.

T(P)=compile time +Runtime

Compile Time of a program can be ignored as the complied program will run several times without recompilationRun-time is denoted by tp.

.

Many factors affect t(p), like the characteristics of the compiler to be used, determine the number of additions, subtractions multiplications, divisions, compares, loads, stores and so on That would be made by the code for P so

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \cdots$$

where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

This formula can be used and the idle time for each operation can be considered, but the operation execution time is machine dependent. So t (p) must be deduced such that it should be machine independent.

So it is idle to calculate t(p) by machine independent feature I.e., to identify the program step and calculate the count of it .

Program Step: is a loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of their instance characteristics.

There are two ways of computing the program step.

1. Program step-count method
2. Tabular method

1. Program step count method

Example: For program to find the sum of n numbers

```
1   Algorithm Sum(a, n)
2   {
3       s := 0.0;
4       count := count + 1; // count is global; it is initially zero.
5       for i := 1 to n do
6       {
7           count := count + 1; // For for
8           s := s + a[i]; count := count + 1; // For assignment
9       }
10      count := count + 1; // For last time of for
11      count := count + 1; // For the return
12      return s;
13  }
```

Here count variable is made declared as global. Count is incremented by the step count of each statement it executes.

So, in the algorithm the for loop, the count will increase by a total of 2n.Then finally 2n+3 will be count after program termination each invocation of sum algorithm executes a total of 2n+3 steps. For recurrence sum the algorithm step count is computed as below

```
1   Algorithm RSum(a, n)
2   {
3       count := count + 1; // For the if conditional
4       if (n ≤ 0) then
5       {
6           count := count + 1; // For the return
7           return 0.0;
8       }
9       else
10      {
11          count := count + 1;  // For the addition, function
12                               // invocation and return
13          return RSum(a, n − 1) + a[n];
14      }
15  }
```

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\text{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n-1) \\ &= 2 + 2 + t_{\text{RSum}}(n-2) \\ &= 2(2) + t_{\text{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\text{RSum}}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

2. Tabular method: to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

Here 2 things we should identify.

s/e: Steps per execution of the statement

Frequency: Number of times the statement is executed.

Example:

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n+3$

Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** RSum(a, n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n-1) + a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$x = t_{\text{RSum}}(n-1)$

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	mn	mn
6 }	0	–	0
Total			$2mn+2m+1$

1.5 Asymptotic Notations and Basic Efficiency Classes

To compare and rank orders of growth, we use three notations: **O (big oh), Ω (big omega), and Θ (big theta).**

O-notation:

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

$t(n) \leq c\, g(n)$ for all $n \geq n_0$

The definition illustration is shown in the below figure

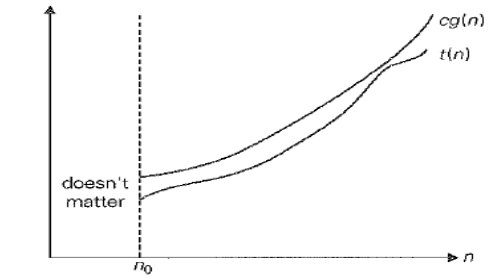


Figure : O-notation

Ex: $3n^3+2n^2 \in O(n^3)$

According to definition of O-notation,

$t(n) \leq c\, g(n)$ for all $n \geq n_0$

i.e. $3n^3+2n^2 \leq c.n^3$ Assume 2 is replaced by n

Then $3n^3+n^3$

$= 4n^3$

so c=4 and $n_0>=2$

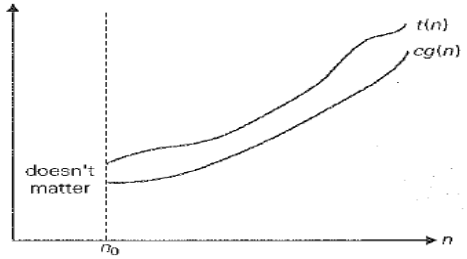
for more examples refer to the material uploaded.

Ω-notation:

A function t(n) is said to be in Ω (g(n)), denoted $t(n) \in \Omega(g(n))$, 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

$t(n) \geq c\, g(n)$ for all $n \geq n_0$

The definition illustration is shown in the below figur



Ex: $n! \in \Omega(2^n)$

According to definition of the notation

$t(n) \geq c\, g(n)$ for all $n \geq n_0$

i.e. $n! \geq c.\ 2^n$

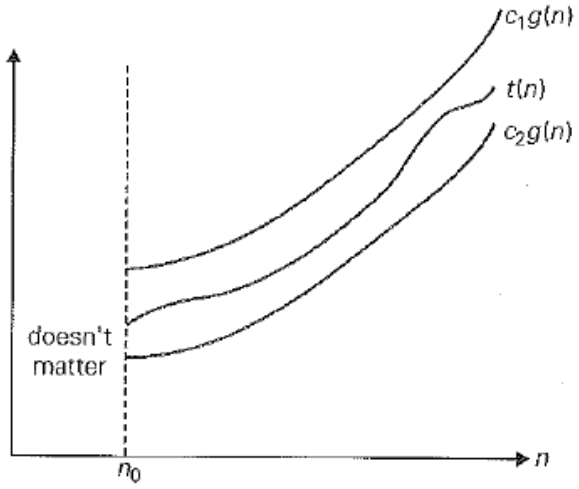
with $c=1$ & $n_0 \geq 4$ the above inequality will be satisfied.

Θ-notation:

function t(n) is said to be in **Θ**(g(n)), denoted $t(n) \in$ **Θ**(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

$c_2 g(n) \leq t(n) \leq c_1 g(n)$ for all $n \geq n_0$

The definition illustration is shown in the below figure



Ex: $\frac{1}{2}n(n-1) \in \Theta(n^2)$

According to definition of Θ notation,

$c_2 g(n) \leq t(n) \leq c_1 g(n)$ for all $n \geq n_0$

i.e. $c_2 n^2 \leq \frac{1}{2}n(n-1) \leq c_1 n^2$

Right inequality (Upper Bound)

$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$ for all $n \geq 0$

Left inequality (Lower Bound)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n = \frac{1}{4}n^2 \quad \text{for all } n \geq 2$$

with $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$ & $n_0 \geq 2$ the above inequality will be satisfied.

THEOREM:

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n)+t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

(The analogous assertions are true for the Θ and Ω notations as well.)

Proof: Let us take four arbitrary real numbers a_1, b_1, a_2 and b_2; if $a_1 \leq b_1$ and $a_2 \leq b_2$ then $a_1+a_2 \leq 2\max\{b_1, b_2\}$

Since $t_1(n) \in O(g_1(n))$,there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \text{ for all } n \geq n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2 g_2(n) \text{ for all } n \geq n_2.$$

Let $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$. Adding above two inequalities

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &= c_3[g_1(n)+g_2(n)] \\ &\leq c_3\, 2 \max\{g_1(n), g_2(n)\} \end{aligned}$$

Hence $t_1(n)+t_2(n) \in O(\max\{g_1(n),g_2(n)\})$ with the constants $c = 2c_3 = 2\max\{c_1,c_2\}$ & $n_0 = \max\{n_1,n_2\}$

Comparing Orders of Growth using limits:-

The three principal cases are:

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

The first two cases mean that t(n) ∈O(g(n)), The last two

cases mean that t(n)∈Ω(g(n)), and the second case means

that t(n)∈Θ(g(n)).

The limit -based approach is often more convenient than the approach based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as

L'Hospital's rule

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ for large values of } n.$$

Example 1: Compare the orders of growth of ½n(n − 1) and n^2.

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

Here the limit is a positive constant, i.e. the functions have the same order of growth i.e.

½n(n − 1) ∈Θ(n^2)

Example 2: Compare the orders of growth of log2n and √n.

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

Here the limit is equal to zero, i.e. log2n has a smaller order of growth than √n. i.e.

log2n∈O(√n).

Example 3: Compare the orders of growth of n! and 2^n.

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

Here the limit is equal to ∞, i.e. n! has a larger order of growth than 2^n. i.e

$n!\in\Omega(2^n)$.

Basic Efficiency classes:-

Class	Name	Example
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 4.4). 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.
nlogn	n-log-n	Many divide-and-conquer algorithms (see Chapter 5), 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 × 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.

1.6 Mathematical Analysis of Non-recursive Algorithms

General Plan for Analyzing Time Efficiency of Non-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 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.
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.

We frequently use two basic rules of sum manipulation:

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

and two summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \text{ where } l \le u \text{ are some lower and upper integer limits}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Example 1: Finding the value of the largest element in a list of n numbers.

ALGORITHM MaxElernent(A[0, … .., n -1])

//Determines the value of the largest element in a given array

//Input: An array A[O .. 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
```

Analysis:

1. The measure of input's size here is the number of elements in the array, i.e., n.
2. There are two basic operations in the algorithm: the comparison A[i] >maxval and the assignment maxval A[i]. Since the comparison is executed on each repetition of the loop and the assignment is not, so comparison to be the algorithm's basic operation.
3. The basic operation of the algorithm depends only on the size of the input, so we need to analyze only one kind of efficiency.
4. Let us denote C(n) the number of times this comparison is executed. The algorithm makes one comparison on each value of the loop's variable i within the bounds 1 and n - 1. Therefore, we get the following sum for C(n)

:

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

5. Solve the above equation by using standard formulas of Summation.

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

Example 2: Element uniqueness problem:

ALGORITHM UniqueElements(A[O …. n1])

//Determines whether all the elements in a given array are distinct or not.

//Input: An array A[O .. 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. The measure of input's size here is the number of elements in the array, i.e., n.
2. The comparison of two elements is the algorithm's basic operation.

3. The number of element comparisons will depend not only on size of input but also on whether there are equal elements in the array and, if there are, which array positions they occupy. So we need to analyze best case, worst case & average case separately.
4. Worst Case analysis:

   There are two kinds of worst-case inputs arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made 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. So, we get basic operation count as:

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

5. Solve the above equation by using standard formulas & rules of Summation.

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

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

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

Example 3: Matrix Multiplication

ALGORITHM MatrixMultiplication(A[0....n-1, 0...n- 1], B[0.... n-1, 0 .... n1])

//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

for k ←0 to n - 1 do

C[i, j] ← C[i, j] + A[i, kj * B[k, j]

return C

Analysis:

1. The measure of input's size is matrix order *n.*
2. The algorithm's innermost loop has two arithmetical operations-multiplication and addition, but as per the property of asymptotic notation we consider multiplication as the algorithm's basic operation.
3. The basic operation count of the algorithm depends only on the size of the input, so we need to analyze only one kind of efficiency.
4. Let M(n) be the total number of multiplications executed by the algorithm. There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable *k* ranging from 0 to *n* - 1. Therefore, the number of multiplications made for every pair of specific values of variables *i* and *j is*

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

The total number of multiplications *M(n)* is expressed by the following triple sum:

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

5. Solve the above equation by using standard formulas & rules of Summation.

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

Example 4: Finding the number of binary digits in the binary representation of a positive decimal integer(non-recursive algorithm).

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. The size of the input is value of n.
2. The comparison $n > 1$ that determines whether the loop's body will be executed. So comparison will be the key operation of the algorithm.
3. The basic operation count of the algorithm depends only on the size of the input, so we need to analyze only one kind of efficiency.
4. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$.
5. **So $C(n) \in \Theta(\log_2 n)$.**

1.7 Mathematical Analysis of 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.

Example 1: Compute the factorial function F(n) = n! for an arbitrary nonnegative integer n.

ALGORITHM F(n)

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
```

if n = 0 return 1

else return F(n - 1) * n

Analysis:

1. The size of the input is value of n.
2. The basic operation of the algorithm is multiplication, let M(n) denote number of executions of Multiplications.
3. The basic operation count of the algorithm depends only on the size of the input, so we need to analyze only one kind of efficiency.
4. F(n) is computed as F(n)=F(n - 1) * n, so the number of multiplications needed to compute F(n) is multiplications needed to compute F(n-1) plus 1 to multiply the result with n.

$$M(n) = \underbrace{M(n-1)}_{\text{to compute } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

To solve the above recurrence relation we need an initial condition i.e. the value with which the sequence starts. This can be obtained by looking at the condition that makes the recursive call to stop. Here

if n=0 return 1

so the recurrence relation with initial condition is

M(n)=M(n-1)+1

M(0)=0

5. Solve the above recurrence relation by using backward substitution method.

M(n) =M(n-1)+1 //Substitute M(n-1)=M(n-2)+1

= (M(n-2)+1)+1

=M(n-2)+2 //Substitute M(n-2)=M(n-3)+1

= (M(n-3)+1)+2

=M(n-3)+3

⋮

=M(n-i)+i

⋮

=M(n-n)+n

=M(0)+n

M(n) =0+n //Initial Condition

M(n) =n

The number of multiplications required is: M(n)=n

∴ The Time Complexity: T(n)∈**Θ(n).**

Example 2: Tower of Hanoi

We have n disks of different sizes & three pegs. Initially all the disks are on the first peg such that largest is on the bottom & smallest is on the top. We have to move all the disks to the third peg using second one as an auxiliary. We can move only one disk at a time and smaller one is always on the top of the larger one. This problem can be solved by recursive technique.

When n>1(number of disks), we first move recursively n-1 disks from peg1 to peg2 with peg3 as auxiliary, then move the largest disk from peg1 to peg3. Finally move the n-1 disks recursively from peg2 to peg3 with peg1 as auxiliary.

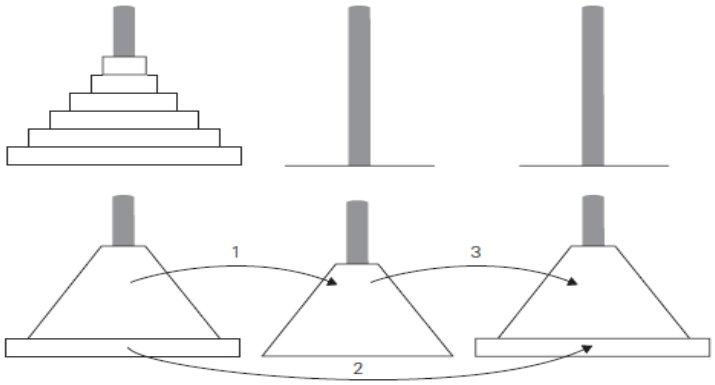


Figure **: Recursive solution to the Tower of Hanoi puzzle.**

Analysis:

1. The input parameter is the number of disks we have to move i.e. n.
2. The key operation of the algorithm is Movement of disks.
3. The number of disks movements depends only on number of disks we have to move i.e. n.
4. The recurrence relation for basic operation count is:

M(n)=M(n-1)+1+M(n-1) for n>1

M(n)=2M(n-1)+1

And the initial condition is **M(1)=1**

5. Solve the above recurrence relation by using backward substitution method.

M(n) $=2M(n-1)+1$ //Substitute M(n-1)=2M(n-2)+1

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

$=2^2M(n-2)+2+1$ //Substitute M(n-2)=2M(n-3)+1

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

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

⋮

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

⋮

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

$=2^{n-1}M(1)+2^{n-2}+2^{n-3}\ldots+ 2^2+2+1$

M(n) $=2^0+2^1+2^2+\ldots..+2^{n-3}+2^{n-2}+2^{n-1}$ //Initial Condition M(1)=1

The above series is in the G.P. So $S_n = \frac{a_1(1-r^n)}{1-r}$

In the above series $a_1=1$, r=2 & n=n.

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

The number of disk movements required is: $M(n)=2^n$

∴ The Time Complexity: $T(n)\in\mathbf{\Theta(2^n)}$.

When a recursive algorithm makes more than a single call to itself, it can be useful for analysis purposes to construct a tree of its recursive calls. In this tree, nodes correspond to recursive calls, and we can label them with the value of the parameter (or, more generally, parameters) of the calls. For the Tower of Hanoi example, the tree is given in Figure. By counting the number of nodes in the tree, we can get the total number of calls made by the Tower of Hanoi algorithm:

$C(n) = \sum_{l=0}^{n-1} 2^l$ (where l is t he level in the tre$\sum^{n-1}{}_0$e in Figure) $= 2^n - 1$.

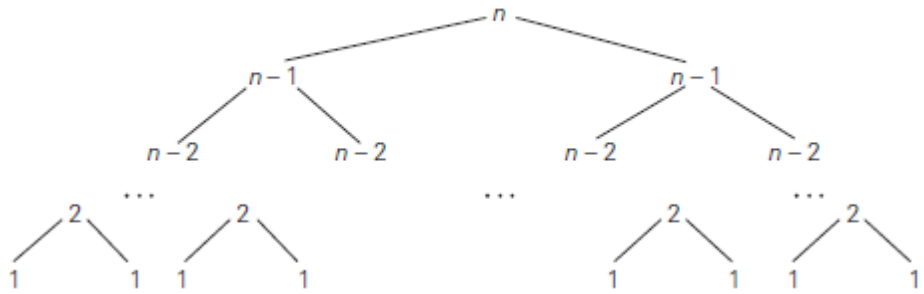


Figure: Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

Example3: Finding the number of binary digits in a 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($\lfloor n/2 \rfloor$)+1

Analysis:

1. The size of the input is value of n.
2. The basic operation of the algorithm is Addition.
3. The basic operation count depends only on the value of input parameter.
4. The recurrence relation for the basic operation count is:

   The number of additions required: **A(n)=A(** $\lfloor n/2 \rfloor$ **)+1**

   The initial condition is **A(1)=0**
5. Solve the above recurrence relation by using backward substitution method.

   Assume $n=2^k$ for simplification

$A(2^k) = A(2^{k/2})+1$

$A(2^k) = A(2^{k-1})+1$ //Substitute $A(2^{k-1})= A(2^{k-2})+1$

$= [A(2^{k-2})+1]+1$

$= A(2^{k-2})+2$ //Substitute $A(2^{k-2})= A(2^{k-3})+1$

$=[A(2^{k-3})+1]+2$

$=A(2^{k-3})+3$

$\vdots$

$= A(2^{k-k})+k$

$= A(2^0)+k$

$= A(1)+k$ //Initial condition is A(1)=0

$A(2^k) = 0+k$

But,

$n = 2^k$

$\log n = \log 2^k = k \log 2(2)$

$\log n = k$

$k = \log n$

$$\mathbf{A(2^k) = k}$$

∴ Key operation count A (n)=log n

∴ The Time Complexity: T (n) $\in \mathbf{\Theta (\log n)}$

1.8 Important Problem types: The most important problem types are

1. Sorting
2. Searching
3. String processing
4. Graph problems
5. Combinatorial problems
6. Geometric problems
7. Numerical problems

These problems are used in the subject to illustrate different algorithm design techniques and methods of algorithm analysis.

1. Sorting

The ***sorting problem*** is to rearrange the items of a given list in non decreasing order.

For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a ***key***.

Two properties of sorting algorithms are.

A sorting algorithm is called ***stable*** if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions *i* and *j* where *i* < *j*, then in the sorted list they have to be in positions *i* and *j* respectively, such that *i*< *j*.

This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA: a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically.

Better Example for stable sort

Consider the following example of student names and their respective class section

(*Dave*, *A*)
(*Alice*, *B*)
(*Ken*, *A*)
(*Eric*, *B*)
(*Carol*, *A*)

Sort the data according to names, the sorted list will not be grouped according to sections !!!!

(*Alice*, *B*)
(*Carol*, *A*)
(*Dave*, *A*)
(*Eric*, *B*)
(*Ken*, *A*)

Sort again to obtain list of students section wise too.
The dataset is now sorted according to sections, but not according to names.

(*Carol*, *A*)
(*Dave*, *A*)
(*Ken*, *A*)
(*Eric*, *B*)
(*Alice*, *B*)

In the name sorted list the tuple (Alice, B) was before (Eric, B)

A stable sorting algorithm would result in

(*Carol*, *A*)
(*Dave*, *A*)
(*Ken*, *A*)
(*Alice*, *B*)
(*Eric*, *B*)

The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be ***in-place*** if it does not require extra memory, except, possibly, for a few memory units. There are important sorting algorithms that are in-place and those that are not.

2. Searching

The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set .There are plenty of searching algorithms to choose from. Example: sequential search and binary search. These algorithms are of particular importance or real-world applications because they are indispensable for storing and retrieving information from large databases.

For searching, too, there is no single algorithm that fits all situations best. Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on. Unlike with sorting algorithms, there is no stability problem, but different issues arise.

3. String Processing

In recent decades, the applications dealing with non numerical data interest of researchers in string-handling algorithms.

A ***string*** is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A,C, G, T}.

There are many string-processing algorithms in computer science one particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it ***string matching***.

4. Graph Problems

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.

Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games. Studying different technical and social aspects of the Internet in particular is one of the active areas of current research involving computer scientists, economists, and social scientists .Basic graph algorithms include graph-traversal algorithms (how can one reach all the points in a network?), shortest-path algorithms (what is the best route between two cities?), and topological sorting for graphs with directed edges examples are the traveling salesman problem and the graph-coloring problem.

5. Combinatorial Problems

Combinatorial problem deals with, **given a finite collection of objects and a set of constraints**, finding an object of the collection that satisfies all constraints. Combinatorial problems are problems involving arrangements of elements from a finite set *and* selections from a finite set.

These problems can be divided into three basic types: (1) enumeration problems, (2) existence problems, and (3) optimization problems.

In enumeration problems the goal is *either* to find how many arrangements there are satisfying the given properties *or* to produce a list of arrangements satisfying the given properties.

In existence problems the goal is to decide whether or not an arrangement exists satisfying the given properties.

In optimization problems the goal is to find where a given function of several variables takes on an extreme value (maximum or minimum) over a given finite domain.

The traveling salesman problem and the graph coloring problem are examples of ***combinatorial problems***. These are problems that ask, explicitly or implicitly, to find a combinatorial object such as a permutation, a combination, or a subset—that satisfies certain constraints.

A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost. Combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint.

Some combinatorial problems can be solved by efficient algorithms, but they should be considered fortunate exceptions to the rule. The shortest-path problem mentioned earlier is among such exceptions.

6. Geometric Problems:

Geometric algorithms deal with geometric objects such as points, lines, and polygons. The ancient of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography. The two classic problems of computational geometry: the closest-pair problem and the convex-hull problem.

The **closest-pair problem** is self-explanatory: given n points in the plane, find the closest pair among them. The **convex-hull problem** asks to find the smallest convex polygon that would include all the points of a given set.

7. Numerical Problems

Numerical problems, another large special area of applications, are problems that involve mathematical objects of continuous nature: **solving equations and systems of equations, computing definite integrals, evaluating functions, and so on**. The majority of such mathematical problems can be solved only approximately. Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately. Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the round-off error to a point where it can drastically distort an output produced by a seemingly sound algorithm.

Many sophisticated algorithms have been developed over the years in this area, and they continue to play a critical role in many scientific and engineering applications. But in the last 30 years or so, the computing industry has shifted its focus to business applications. These new applications require primarily algorithms for information storage, retrieval, transportation through networks, and presentation to users. As a result of this revolutionary change, numerical analysis has lost its formerly dominating position in both industry and computer science programs. Still, it is important for any computer-literate person to have at least a rudimentary idea about numerical algorithms.

1.9 Important Data Structures

Since majority of algorithms operate on data, particular ways of **organizing data** play a critical role in the design and analysis of algorithms.

A ***data structure*** can be defined as a particular scheme of organizing related data items. The nature of the data items is dictated by the problem at hand; they can range from elementary data types (e.g., integers or characters) to data structures (e.g., a one-dimensional array of one-dimensional arrays is often used for implementing matrices).

There are a few data structures that have proved to be particularly important for computer algorithms.

Linear Data Structures

The two most important elementary data structures are the array and the linked list. A (one dimensional) ***array*** is a sequence of *n* items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's ***index***

Fig a***: Array of n elements***

- In the majority of cases, the index is an integer either between 0 and $n - 1$ or between 1 and *n.* Some computer languages allow an array index to range between any two integer bounds *low* and *high.*
- Each and every element of an array can be accessed in the same constant amount of time regardless of where in the array the element in question is located. Arrays are used for implementing a variety of other data structures.
- ***string***, a sequence of characters from an alphabet terminated by a special character indicating the string's end. Strings composed of zeros and ones are called ***binary strings*** or ***bit strings***.

A ***linked list:*** is a sequence of zero or more elements called ***nodes***, each containing two kinds of information: some data and one or more links called ***pointers*** to other nodes of the linked list.

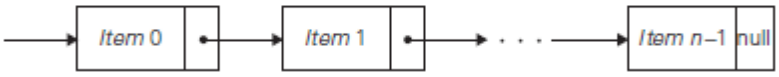


Fig b: Singly linked list of n elements.

In a ***singly linked list***(Fig b) each node except the last one contains a single pointer to the next element .To access a particular node of a linked list, one starts with the list's first node and traverses the pointer chain until the particular node is reached. Thus, the time needed to access an element of a singly linked list, unlike that of an array, depends on where in the list the element is located. On the positive side, linked lists do Item [0] Item [1] Item [n –1].

Insertions and deletions can be made quite efficiently in a linked list by reconnecting a few appropriate pointers. It is often convenient to start a linked list with a special node called the ***header***. This node may contain information about the linked list itself, such as its current length; it may also contain, in addition to a pointer to the first element, a pointer to the linked list's last element.

Doubly linked list: this is an Another extension of singly linked list in which every node, except the first and the last, contains pointers to both its successor and its predecessor

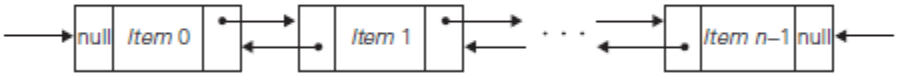


Fig c: Doubly linked list of n elements

The array and linked list are two principal choices in representing a more abstract data structure called a linear list or simply a list.

A ***list :*** is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element. Two special types of lists, stacks and queues, are particularly important.

A ***stack*** is a list in which insertions and deletions can be done only at the end. This end is called the ***top*** because a stack is usually visualized not horizontally but vertically—akin to a stack of plates whose "operations" it mimics very closely. As a result, when elements are added to (pushed onto) a stack and deleted from (popped off) it, the structure operates in a "last-in–first-out" (LIFO) fashion—exactly like a stack of plates if we can add or remove a plate only from the top.

Stacks have a multitude of applications; in particular, they are indispensable for implementing recursive algorithms.

A ***queue***, on the other hand, is a list from which elements are deleted from one end of the structure, called the ***front*** (this operation is called ***dequeue***),and new elements are added to the other end, called the ***rear*** (this operation is called ***enqueue***).

Consequently, a queue operates in a "first-in–first-out" (FIFO) fashion—akin to a queue of customers served by a single teller in a bank. Queues also have many important applications,

including several algorithms for graph problems. Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates.

A data structure that seeks to satisfy the needs of such applications is called a priority queue. A ***priority queue*** is a collection of data items from a totally ordered universe (most often integer or real numbers). The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element. .

A better implementation of a priority queue is based on an ingenious data structure called the ***heap***. We discuss heaps and an important sorting algorithm based on them in Section 6.4.

Graphs

A graph is informally thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs."

Formally, a ***graph****G* = _*V*,*E*_ is defined by a pair of two sets: a finite nonempty set *V* of items called ***vertices*** and a set *E* of pairs of these items called ***edges***. If these pairs of vertices are unordered, i.e., a pair of vertices *(u, v)* is the same as the pair *(v, u)*, we say that the vertices *u* and *v* are ***adjacent*** to each other and that they are connected by the ***undirected edge*** *(u, v)*.

We call the vertices *u* and *v* ***endpoints*** of the edge *(u, v)* and say that *u* and *v* are ***incident*** to this edge; we also say that the edge *(u, v)* is incident to its endpoints *u* and *v*. A graph *G* is called ***undirected*** if every edge in it is undirected. If a pair of vertices *(u, v)* is not the same as the pair *(v, u)*, we say that the edge *(u, v)* is ***directed*** from the vertex *u*, called the edge's ***tail***, to the vertex *v*, called the edge's ***head***. We also say that the edge *(u, v)* leaves *u* and enters *v*. A graph whose every edge is directed is called ***directed***. Directed graphs are also called ***digraphs***.

It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (**Fig d** a & Fig b). The graph depicted in Fig d (a) has six vertices and seven undirected edges:

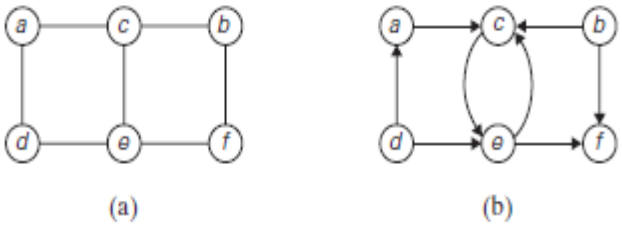


Fig d: Undirected graph ***Directed graph***

$V = \{a, b, c, d, e, f\}$, $E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}$.

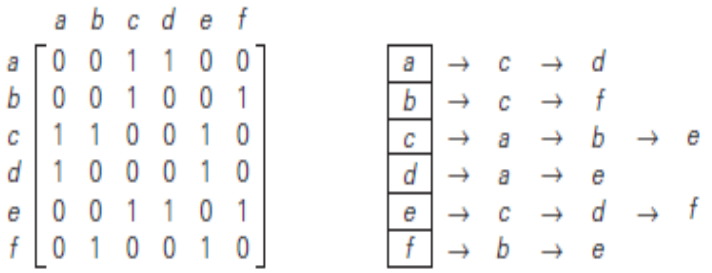
The digraph depicted in Figure has six vertices and eight directed edges:
$V = \{a, b, c, d, e, f\}$, $E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}$.

Our definition of a graph does not forbid ***loops***, or edges connecting vertices to themselves.
A graph with relatively few possible edges missing is called ***dense***. A graph with few edges relative to the number of its vertices is called ***sparse***.

Whether we are dealing with a dense or sparse graph may influence how we choose to represent the graph and, consequently, the running time of an algorithm being designed or used.

Graph Representations Graphs for computer algorithms are usually represented in one of two ways: the adjacency matrix and adjacency lists.

The ***adjacency matrix*** of a graph with n vertices is an $n \times n$ boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the j th column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge. For example, the adjacency matrix for the graph of



(a) (b)

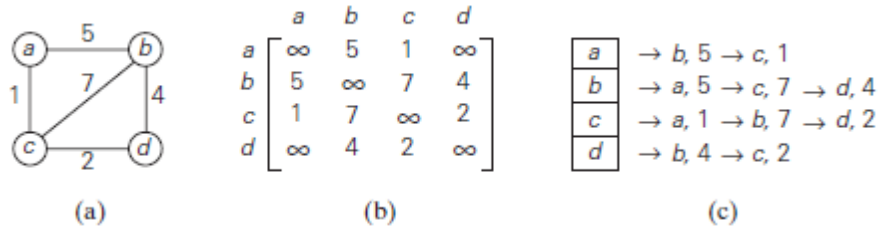
For the above **Fig d** (a) here is the adjacency matrix and (b) adjacency list respectively.

Note that the adjacency matrix of an undirected graph is always symmetric, i.e., $A[i, j] = A[j, i]$ for every $0 \le i, j \le n - 1$

The ***adjacency lists*** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge).. For example, Figure d represents the graph in Figure 1.6a via its adjacency lists. To put it another way.

Weighted Graphs A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called ***weights*** or ***costs***.

If a weighted graph is represented by its adjacency matrix, then its element $A[i, j]$ will simply contain the weight of the edge from the ith to the jth vertex if there is such an edge and a special symbol, e.g.,∞, if there is no such edge. Such a matrix is called the ***weight matrix*** or ***cost matrix***.



a) Weighted graph b) Weighted matrix c) Adjacency list

Paths and Cycles Among the many properties of graphs, two are important for a great number of applications: ***connectivity*** and ***acyclicity***. Both are based on the notion of a path.

A ***path*** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v. If all vertices of a path are distinct, the path is said to be ***simple***.

The ***length*** of a path is the total number of vertices in the vertex sequence defining the path minus 1, which is the same as the number of edges in the path. For example, *a, c, b, f* is a simple path of length 3 from *a* to *f* in the graph in Figure 1.6a, whereas *a, c, e, c, b, f* is a path (not simple) of length 5 from *a* to *f*.

A ***directed path*** is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next. For example, *a, c, e, f* is a directed path from *a* to *f* in the graph in Figure 1.6b.

A graph is said to be ***connected*** if for every pair of its vertices *u* and *v* there is a path from *u* to *v*. If we make a model of a connected graph by connecting some balls representing the graph's vertices with strings representing the edges, it will be a single piece.

If a graph is not connected, such a model will consist of several connected pieces that are called **connected components of the graph.** Formally, a ***connected component*** is a maximal (not expandable by including another vertex and an edge) connected subgraph2 of a given graph. For example, the graphs in Figures 1.6a and 1.8a are connected, whereas the graph in Figure below is not, because there is no path, for example, from *a* to *f*. The graph in the below Figure has two connected components with vertices {*a, b, c, d, e*} and {*f, g, h, i*}, respectively. Graphs with several connected components do happen in real-world applications.

. A ***cycle*** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once. For example, *f , h, i, g, f* is a cycle in the graph in Figure 1.9. A graph with no cycles is said to be ***acyclic***.

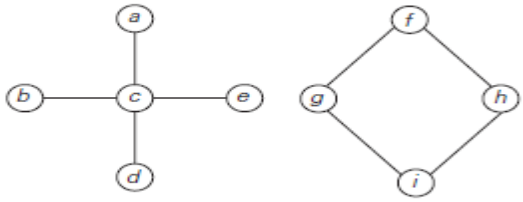
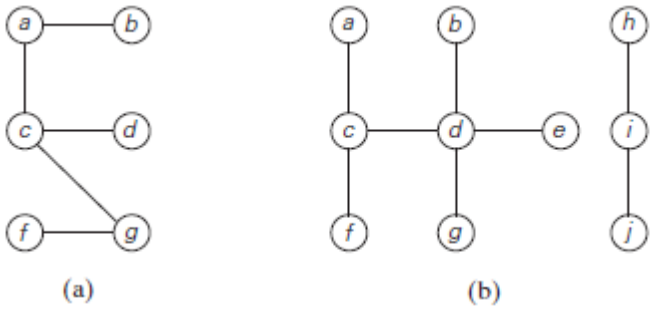


Figure : connected components of the graph

Trees

A ***tree*** (more accurately, a ***free tree***) is a connected acyclic graph (Figure below)

A graph that has no cycles but is not necessarily connected is called a ***forest***: Each of its connected components is a tree A ***subgraph*** of a given graph *G* = _*V, E*_ is a graph *G*



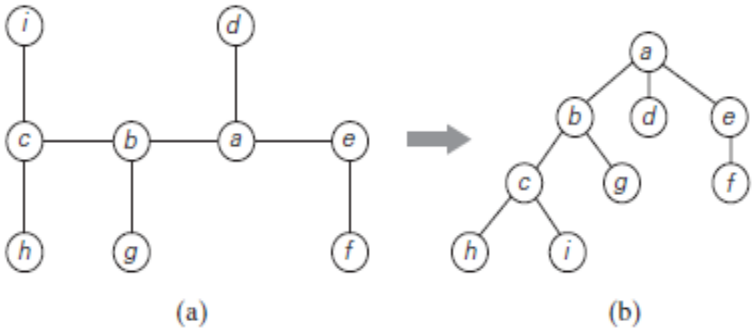
(a) Tree. (b) Forest.

Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices**:** $|E| = |V| - 1$.

Rooted Trees Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the ***root*** of the so-called ***rooted tree***.

A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on.

Figure below presents such a transformation from a free tree to a rooted tree. Rooted trees play a very important role in computer science, a much more important one than free trees do; in fact, for the sake of brevity, they are often referred to as simply "trees." An obvious application of trees is for describing hierarchies, from file directories to organizational charts of enterprises. There are many less obvious applications, such as implementing dictionaries and data encoding



(a) Free tree. (b) Its transformation into a rooted tree.

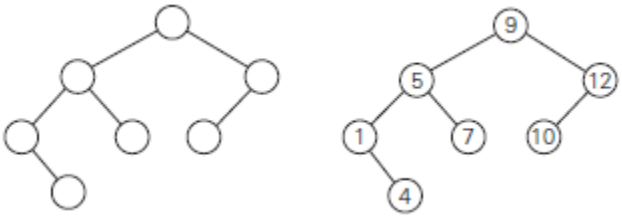
List of tree applications, we should mention the so-called ***state-space trees*** that underline two important algorithm design techniques: backtracking and branch-and-bound .

- For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called ***ancestors*** of v.
- The vertex itself is usually considered its own ancestor; the set of ancestors that excludes the vertex itself is referred to as the set of ***proper ancestors***.

- If *(u, v)* is the last edge of the simple path from the root to vertex *v* (and *u* _= *v)*, *u* is said to be the ***parent*** of *v* and *v* is called a ***child*** of *u*;
- vertices that have the same parent are said to be ***siblings***.
- A vertex with no children is called a ***leaf*** ;
- a vertex with at least one child is called ***parental***. All the vertices for which a vertex *v* is an ancestor are said to be ***descendants*** of *v*;
- the ***proper descendants*** exclude the vertex *v* itself.
- All the descendants of a vertex *v* with all the edges connecting them form the ***subtree*** of *T* rooted at that vertex.
- Thus, for the tree in Figure b, the root of the tree is *a*; vertices *d, g, f, h*, and *I* are leaves, and vertices *a, b, e*, and *c* are parental; the parent of *b* is *a*; the children of *b* are *c* and *g*; the siblings of *b* are *d* and *e*; and the vertices of the subtree rooted at *b* are {*b, c, g, h, i*}.
- The ***depth*** of a vertex *v* is the length of the simple path from the root to *v*.
- The ***height*** of a tree is the length of the longest simple path from the root to a leaf.

Ordered Trees An ***ordered tree*** is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right.

A ***binary tree*** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a ***left child*** or a ***right child*** of its parent; a binary tree may also be empty. The binary tree with its root at the left (right) child of a vertex in a binary tree is called the ***left*** (***right***) ***subtree*** of that vertex.



In Figure below, some numbers are assigned to vertices of the binary tree. Note that a number assigned to each parental vertex is larger than allthe numbers in its left subtree and smaller than all the numbers in its right subtree.Such trees are called ***binary search trees***. (This figure shows binary tree and its binary search tree representation).

Binary trees and binary search trees have a wide variety of applications in computer science; you will encounter some of them throughout the book. In particular, binary search trees can be generalized to more general types of search trees called ***multiway search trees***, which are indispensable for efficient access to very large data sets. As you will see later in the book, the efficiency of most important algorithms for binary search trees and their extensions depends on the tree's height. Therefore, the following inequalities for the height h of a binary tree with n nodes are especially important for analysis of such algorithms:$\log_2 n\_ \leq h \leq n - 1$.

A binary tree is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree. Each node contains some information associated with the vertex (its name or some value assigned to it) and two pointers to the nodes representing the left child and right child of the vertex, respectively. All the siblings of a vertex are linked via the nodes'right pointers in a singly linked list, with the first element of the list pointed to by the left pointer of their parent.

Figure 1.14a illustrates this representation for the tree in Figure 1.11b. It is not difficult to see that this representation effectively transforms an ordered tree into a binary tree said to be associated with the ordered tree. We get this representation by "rotating" the pointers about 45 degrees clockwise (see Figure 1.14b).

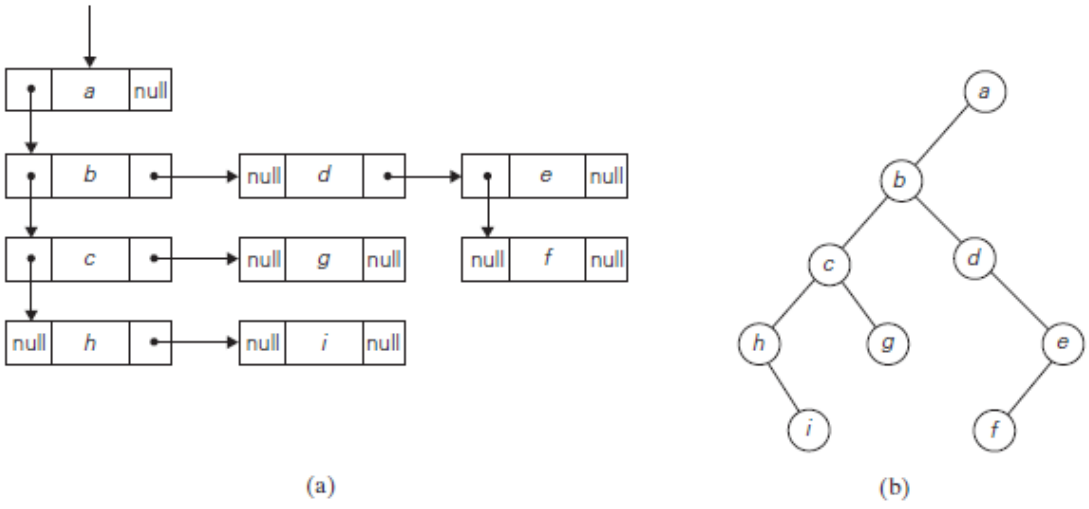


FIGURE 1.14 (a) First child–next sibling representation of the tree in Figure 1.11b. (b) Its binary tree representation.

Sets and Dictionaries

The notion of a set plays a central role in mathematics. A ***set*** can be described as an unordered collection (possibly empty) of distinct items called ***elements*** of below figure .

A specific set is defined either by an explicit listing of its elements (e.g., $S = \{2,3, 5, 7\}$) or by specifying a property that all the set's elements and only they must satisfy (e.g., $S = \{n: n$ is a prime number smaller than 10$\}$).
The most important set operations are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets.

Sets can be implemented in computer applications in two ways. The first considers only sets that are subsets of some large set *U*, called the ***universal set***. If set *U* has *n* elements, then any subset *S* of *U* can be represented by a bit string of size *n*, called a ***bit vector***, in which the *i*th element is 1 if and only if the i^{th} element of *U* is included in set *S*. Thus, to continue with our example, if $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, then $S = \{2, 3, 5, 7\}$ is represented by the bit string 011010100. This way of representing sets makes it possible to implement the standard set operations very fast, but at the expense of potentially using a large amount of storage.

The second and more common way to represent a set for computing purpose is to use the list structure to indicate the set's elements. Of course, this option, too, is feasible only for finite sets; fortunately, unlike mathematics, this is the kind of sets most computer applications need.

Note, however, there are two principal points of distinction between sets and lists. First, a set cannot contain identical elements; a list can. This requirement for uniqueness is sometimes circumvented by the introduction of a ***multiset***, or ***bag***, an unordered collection of items that are not necessarily distinct. Second, a set is an unordered collection of items; therefore, changing the order of its elements does not change the set. A list, defined as an ordered collection of items, is exactly the opposite.

This is an important theoretical distinction, but fortunately it is not important for many applications. It is also worth mentioning that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the ***dictionary***. Note the relationship between this data structure and the problem of searching mentioned in Section 1.3; obviously, we are dealing here with searching in a dynamic context. Consequently, an efficient implementation of a dictionary has to strike a compromise between the efficiency of searching and the efficiencies of the other two operations.

MODULE—2
DIVIDE AND CONQUER

2.1 General method

2.2 Binary search

2.3 Recurrence equation for divide and conquer

2.4 Finding the maximum and minimum

2.5 Merge sort (T1)

2.6 Quick sort (T1)

2.7 Stassen's matrix multiplication (T1)

2.8 Advantages and Disadvantages of divide and conquer

2.9 Decrease and Conquer Approach

2.10 Topological Sort

2.1 Divide and conquer General Method

The Divide-and-conquer strategy suggests splitting the inputs of size 'n' into 'k' distinct subsets such that $1<k \leq n$, producing k sub problems.

These sub problems must be solved & then a method to be found to combine the solutions of sub problems (sub solutions) to produce the solution to the original problem of size 'n'. If the sub problems are relatively large then divide & conquer strategy must be reapplied. For the reapplication sub problems the divide & conquer strategy will be expressed as recursive algorithm.

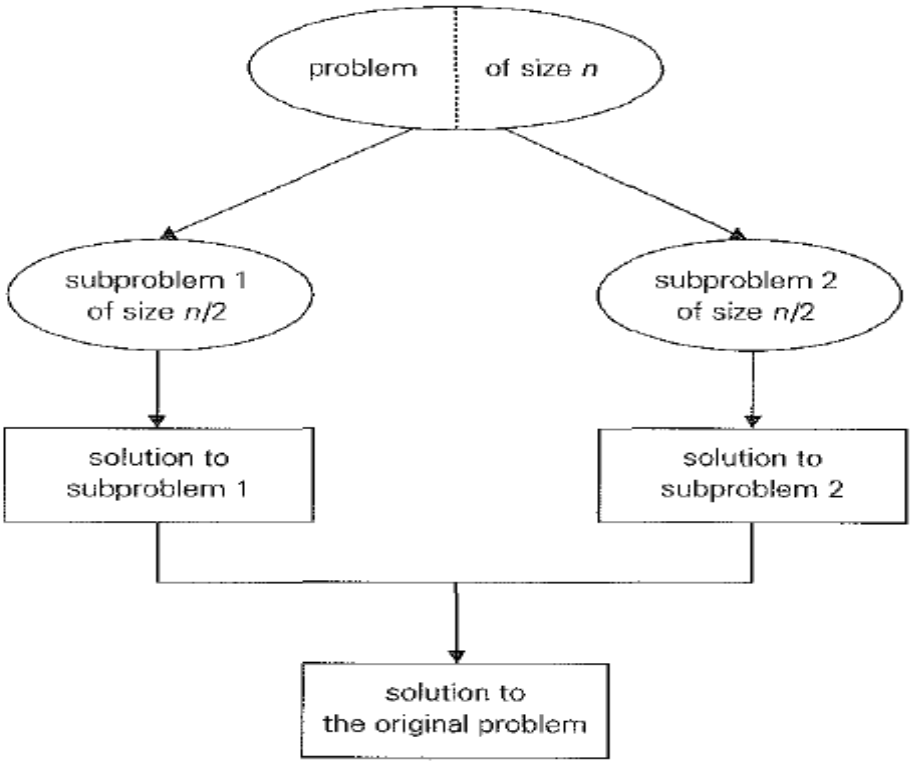


Figure 2.1: Divide and conquer

Thus, the general plan of divide and conquer strategy can be represented as follows:

1. **DIVIDE:** A problems instance is divided into several sub problem instances of the same problem, ideally of about the same size.
2. **RECUR:** solve the sub problems recursively
3. **CONQUER :** If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance

- Divide-and-Conquer (DaC) is probably the best-known general algorithm design technique.
- Given a function to compute on n input the DaC approach suggests splitting the inputs into k distinct subsets,1< k < n, yielding k subproblems

- These sub problems must be solved and then a method must be found to combine solutions into a solution of the whole.
- If the sub problems are relatively large then the divide and conquer approach can possibly be reapplied
- Often the sub problems resulting from our divide and conquer design are of the same type as the original problem.
- Further those cases the reapplication of the divide and conquer principle is naturally expressed by a recursive algorithm
- The smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced

Divide and conqueror strategy splits the input into two sub problems of the same kind as the original problem. Consider the following algorithm (DAndC) which is invoked for problem p to be solved.

Small (P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting.

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
9       }
10  }
```

If the size of p is n and the sizes of k sub problems are n_1, n_2,….., n_k then the computing time of (DAndC) is described by the recurrence relation. Divide and conquer (DAndC) is described by the recurrence relation given below which is used to know the computing time.

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

T(n) - The time for DAnd C on any input of size n

g(n) -Time to compute the answer directly for small inputs.

f(n)- Time for dividing P and combining the solutions to sub problems.

$T(n_{1...k})$ – Time taken for instances of sub problems.

Solve the Problem:

Consider the case in which a=2 and b=2. Let T(1)=2 and f(n)= n .

Solution : refer to material sent

2.2 Divide-and-conquer recurrence equation

In the most typical case of divide-and-conquer , a problem's instance of size *n* is divided into two instances of size *n*/2.

i, e $T(n) = T(n/2) + T(n/2) + f(n)$

$= 2\,T(n/2) + f(n)$

More generally, an instance of size *n* can be divided into *b* instances of size *n/b*, with *a* of them to be solved. (Here, *a* and *b* are constants; $a \geq 1$ and $b > 1$.)

Assuming that size *n* is a power of *b* to simplify analysis; we get the following recurrence for the running time *T (n)*:

$$T(n) = a\,T(n/b) + f(n) \quad \text{..... 2.1}$$

Where **f *(n)*** is a function that accounts for the time spent on dividing an instance of size *n* into instances of size *n/b* and combining their solutions.

By recursively solving equation 2.1 , we get

$$T(n) = n^{\log_b a}\,[T(1) + u(n)]$$

Where,

$$u(n) = \sum_{j=1}^{k} h(b^j)$$

$$h(n) = f(n)/\,n^{\log_b a}$$

2.3 Binary Search

Let a_i $(1 \le i \le n)$ is a list of elements stored in non decreasing order. The searching problem is to determine whether a given element is present in the list. If key x is present, we have to determine the index value j such that a[j] =x. If x is not in the list j is set to be zero.

Let P= (n, a_i, . . . a_l, x) denotes an instance of binary search problem. Divide & Conquer can be used to solve this binary search problem. Let Small (P) be true if n=1. S(P) will take the value i if $x=a_i$, otherwise it will take 0.

If P has more than one element it can be divided into new sub-problems as follows:

Take an index q within the range [i, l] & compare x with aq.

There are three possibilities.

i. If $x= a_q$ the problem is immediately solved (Successful search)

ii. If $x< a_q$, key x is to be searched only in sub list a_i, a_{i+1}, . . ., a_{q-1}.

iii. If $x > a_q$, key x is to be searched only in sub list a_{q+1}, a_{q+2},....a_l. If q is chosen such that a_q is the middle element i.e. q=(n+1)/2, then the resulting searching algorithm is known as Binary Search algorithm.

Non- Recursive Binary Search

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then  // If Small(P)
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      { // Reduce P into a smaller subproblem.
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                  return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

Recursive Binary search Algorithm:

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then  // If Small(P)
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      { // Reduce P into a smaller subproblem.
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                  return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

Analysis:-

We do the analysis with frequency count(Key operation count) and space required for the algorithm. In binary search the space is required to store n elements of the array and to store the variables low, high, mid and x i.e. n+4 locations

To find the time complexity of the algorithm, as the comparison count depends on the specifics of the input, so we need to analyze best case, average case & worst case efficiencies separately. We assume only one comparison is needed to determine which of the three possibilities of if condition in the algorithm holds. Let us take an array with 14 elements.

Index pos	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Elements	-10	-6	-3	-1	0	2	4	9	12	15	18	19	22	30
comparisons	3	4	2	4	3	4	1	4	3	4	2	4	3	4

From the above table we can conclude that, no element requires more the 4 comparisons. The average number of comparisons required is (sum of count of comparisons/number of elements) i.e 45/14=3.21 comparisons per successful search on average.

There are 15 possible ways that an unsuccessful search may terminate depending on the value of x. if x < a[1], the algorithm requires three comparisons to determine x is not present. For all the remaining possibilities the algorithm requires 4 element comparisons. Thus the average number of comparisons for an unsuccessful search is (3+14*4)/15=59/15=3.93.

To derive the generalized formula and for better understanding of algorithm is to consider the sequence of values for mid that are produced by **BinarySearch** for all possiblevalues of x. these possible values can described using binary decision tree in which the value in each node is the value of *mid*. The below figure is the decision tree for n=14.

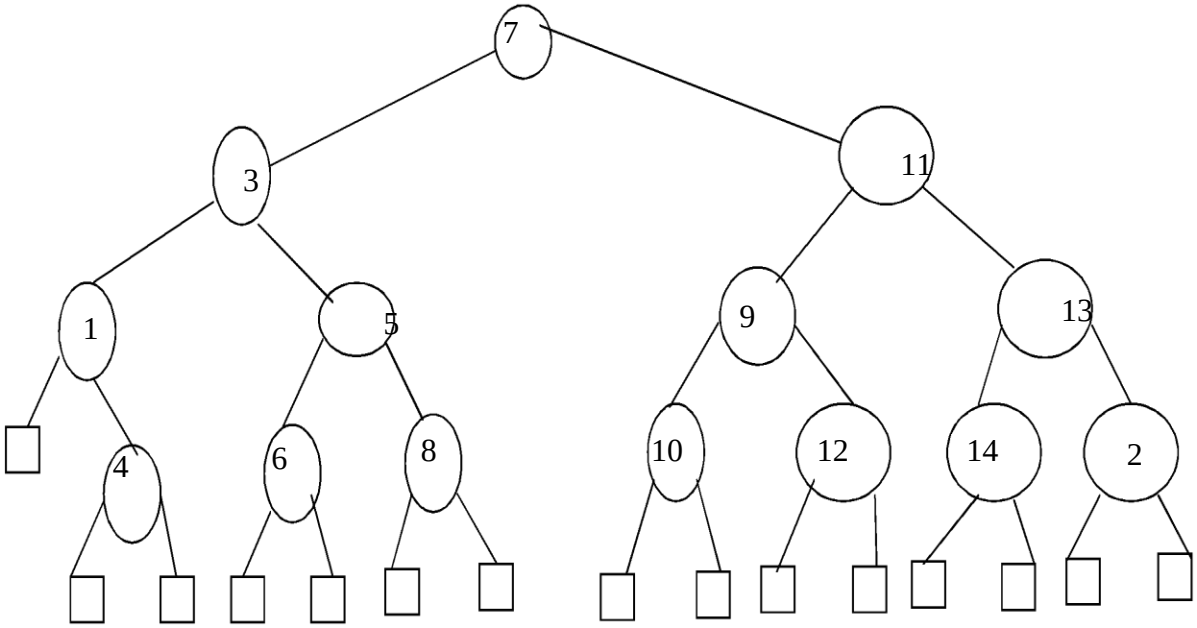


Figure 2.1: Decision tree

In the above decision tree, each path through the tree represents a sequence of comparisons in the binary search method. If x is present, then algorithm will end at one of the circular node (internal nodes) that lists the index into the array where x was found. If x is not present, the algorithm will terminate at one of the square nodes (external nodes).

Recurrence Relation

- $T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + 1 & n > 1 \end{cases}$

Solution

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

$= [T(n/4)+1]+1 = T(n/4)+2$

$= [T(n/8)+1]+2 = T(n/8)+3$

- ----------
- $= T(n/2^k)+k$ $\quad n = b^k, n = 2^k, \log n = \log 2^k, k = \log n$
- $= T(n/n)+k$
- $= 1+k$
- $= 1+\log n$
- **$T(n) = O(\log n)$**

$T(n) = \Theta(\log n)$ [in many conventions]

Theorem 1:

If n is in the range $[2^{k-1}, 2^k]$, then **Binary Search** makes at most k element comparisons for successful search and either k-1 or k comparisons for an unsuccessful search. (i.e. The time for a successful search is O (logn) and for unsuccessful search is Θ(logn)).

Proof:

Let us consider a binary decision tree, which describes the function of Binary search on n elements. Here all successful search in figure 2.1 end at a circular node, where as all unsuccessful search will end at square node. If $2^{k-1} \leq n < 2^k$, then all circular nodes are at levels 1,2,…,k, where as all square nodes are at levels k & K+1(root node is at level 1). The number of element comparisons needed to terminate at a circular node on level i is i, where as the number of element comparisons needed to terminate at a square node at level i is only i-1, hence the proof.

The above proof will tell the worst case time complexity is O(logn) for successful search & Θ(logn) for unsuccessful search.

Overall Time complexity of binary search is

For Successful search

Best:Θ(1), Average : Θ (logn) Worst: Θ (logn)

For unsuccessful case (all same)

Best, Average, Worst: Θ (logn)

Example 1: Consider an array A of elements below for binary search

-15, -6, 0, 7, 9 , 23, 54, 82 and **Key=7**

Recursive calls: BinarySearch(a,1,12,7) for the recursive algorithm defined above BinarySearch(a, i, l, x).

Solution:

Index	1	2	3	4	5	6	7	8
Array a elements	**-15**	**-6**	**0**	**7**	**9**	**23**	**54**	**82**

- Mid =(1+8)/2=4.5= rounded to 4
- Compare key x with mid element i.e. 7=a[mid]=a[4]=7
- Key matches with the middle element of the array so the algorithm returns 4 as the index position of the element found in array a. so it is successful.
- This case is the best case also .As it takes only one comparison.

Example 2:

Consider an array A of elements below for binary search

-15, -6, 0, 7, 9 , 23, 54, 82 and **Key=54**

Solution:

Index	1	2	3	4	5	6	7	8
Array a elements	**-15**	**-6**	**0**	**7**	**9**	**23**	**54**	**82**

- Compute mid =(1+8)/2=4

- Compare key x with mid element i.e 82 =a[mid] i.e a[4]=7 not equal to key 54.
- if (x<a[mid]) then no
- so (x>a[mid]) as 54>7, so call BinarySearch(a,mid+1, l, x);

Index	5	6	7	8
Array a	**9**	**23**	**54**	**82**

BinarySearch(a,5,8,7) so array is divided in to two parts and we compare only the right subarray.

Index	1	2	3	4
Array a elements	**-15**	**-6**	**0**	**7**

In right subarray again compute mid element mid=(5+8)/2=6

Since a[mid]=23 is lesser than 54 .again recursively call the Binarysearch for right subarray

BinarySearch(a,mid+1, l, x) i.e BinarySearch(a,7, 6, 8);

Index	5	6
Array a elements	**9**	**23**

Index	7	8
Array a elements	**54**	**82**

Compute mid=(7+8)/2=7

A[7]=54 matches with the key.so return sussess with index 7 saying key 54 is found at position 7 in the array.

2.4 Finding the Maximum and minimum

Problem statement: The problem is to find the maximum and minimum items in a set of n elements.

Algorithm 1: Straight method.

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

Note: This algorithm takes 2n-2 comparisons**.**

Divide and conquer

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i = j) then max := min := a[i]; // Small(P)
7       else if (i = j − 1) then  // Another case of Small(P)
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

T(n) = 2 T(n/2) + 2

= 2[2 T(n/4) +2] +2

= 4 T(n/4) +4+2

= 4 [2 T(n/8) +2] +4+2

= 8 T(n/8)+ 8+4+2

= :

= 2^k T (n/ 2^k) 2^k + 2^{k-1} +2^{k-2}+…2

As T(1) = 0 and T(2)=1 , consider k-1

= 2^{k-1} T (n/ 2^{k-1}) [2^{k-1} + 2^{k-2} +2^{k-3} +….. 2]

⟺

= 2^{k-1} T (n/ 2^{k-1}) [2^k -2]

= 2^k/2 T (n / 2^k * 2) [2^k -2]

We know that n= 2^k, substitute it

= n/2 T(2 n/n) [n-2]

= n/2 T(2) (n-2)

As T(2)=1

= n/2*(n-2)

=3n/2-2

2.5 Merge Sort

Merge is a perfect example of successful application of divide and conquer strategy.

Given a sequence of n elements a[1],a[2],...,a[n], the merge sort algorithm will split into two sets a[1], ... a[⌊n/2⌋] and a[⌊n/2⌋+1],….,a[n]. Each set is individually sorted & resulting sorted sets are merged to get a single sorted array of n elements.

Procedure:

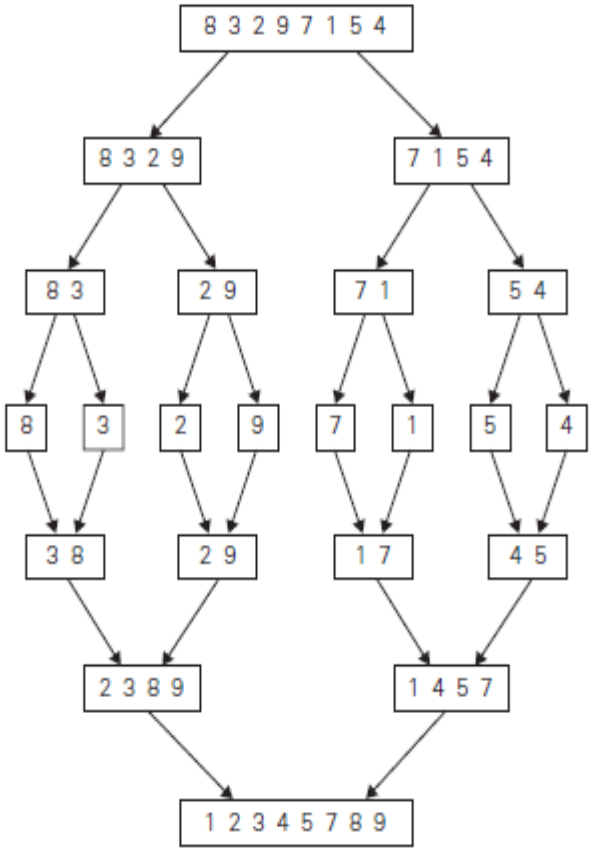
- **Divide**: Partition array in to two sub lists
- **Conque**r: Then sort two sub lists
- **Combine**: Merge sub problems

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

Example: Consider array of elements **8 3 2 9 7 1 5 4**

The merge sort operation performed on it is depicted below.



Example of a merged operation

Analysis

Assuming for simplicity that *n* is a power of 2, the recurrence relation for the number of key comparisons *C (n)* is

$$C(n) = C(n/2) + C(n/2) + C_{merge}(n) \text{ for } n > 1,\ C(1) = 0.$$

To process **1st half** **2nd half** **combine**

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n > 1,\ C(1) = 0.$$

- The recurrence relation for Merge sort is given by

$$T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + cn & n > 1, c \text{ is a constant} \end{cases}$$

- **Solution**

In the given relation a=2, b=2, f(n)=cn , n is power of b so $n=b^k$, $n=2^k$

$T(n)=2T(n/2)+cn$ substitute $T(n/2)=2T(n/4)+c(n/2)$

$= 2[2[T(n/4)+cn/2)]+cn$

$=4T(n/4)+2cn$ substitute $T(n/4)=2T(n/8)+c(n/4)$

$=4[2T(n/8)+cn/4)+2cn$

$=8T(n/8)+3cn$

The general pattern ?

$= 2^kT(n/2^k)+kcn$ $n=2^k$, $k=\log n$

$=nT(1)+\log n\, cn$

$= n+c\, n\log n$ considering only leading term and ignoring constants we get

$T(n)= \Theta(n\log n)$

2.6 Quick sort

Quick sort is the other important sorting algorithm that is based on the divide-and conquers approach. Unlike merge sort, which divides its input elements according to their position in the array, quick sort divides them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$A[0] \ldots A[s-1] \_ \_\_ \_ \text{ all are } \leq A[s]$$

$$A[s]\, A[s+1] \ldots A[n-1] \_ \_\_ \_ \text{all are } \geq A[s]$$

After a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two sub arrays to the left and to the right of $A[s]$ independently (e.g., by the same method).

Note: the difference with merge sort: there, the division of the problem into two sub problems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the sub problems.

Here is pseudo code of quick sort:

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
// l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

Quick sort execution procedure:

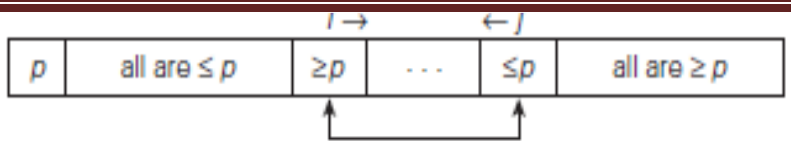
- Call to quick sort which performs sorting recursively for sub arrays.
- Partitioning or split position is identified by using partitioning function which divides the array in to two sub arrays.

Partitioning function procedure:

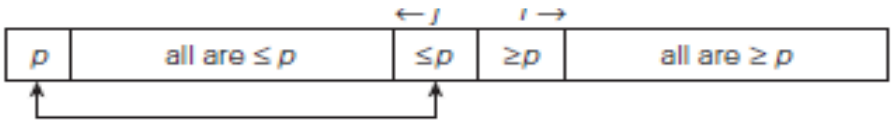
- 1. Make first element as the pivot element. **P** //many ways of selecting a pivot element exist.
- 2. set i (low index to point to 0)and j to high which is n-1.
- **Increment** i **until** $A[i] \geq p$.If condition is met stop incrementing i.
- **Decrement** j **until** $A[j] \leq p$. If condition is met stop decrementing j
- 5.Compare the positions of i and j (three cases may arise,<,>,=)

a) If i<j perform swap(A[i], A[j]) and continue incrementing i and from the same positions.

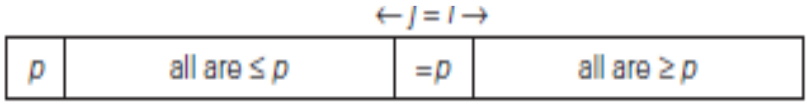
If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing I and decrementing j, respectively:



b) If i>j then swap pivot element with a[j].return j as the split position. If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:



c) Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



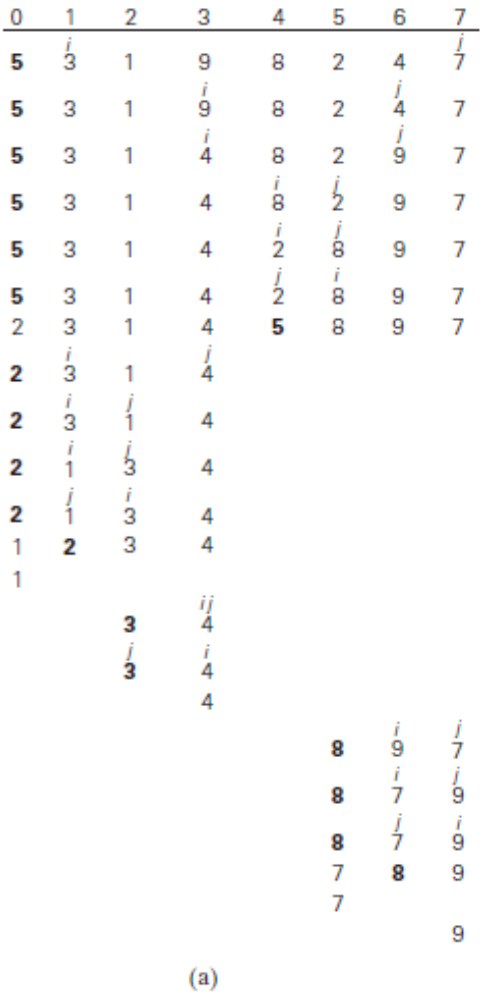
We can combine the last case with the case of crossed-over indices $(i > j)$ by exchanging the pivot with $A[j]$ whenever $i \geq j$. j is the split position. Towards left of this positions elements will be lesser in value and right side elements will be larger in value. And the same are considered as two subarrays which we take separately to sort.

ALGORITHM *Partition*$(A[l..r])$

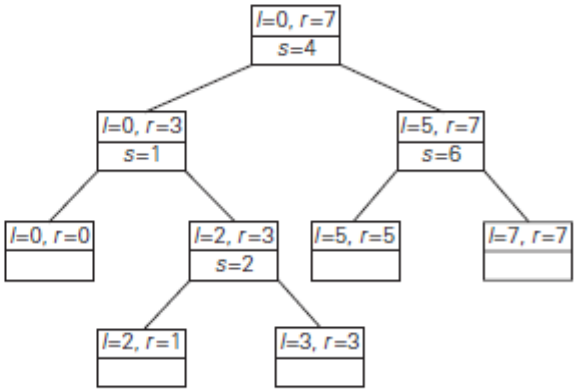
```
//Partitions a subarray by using its first element as a pivot
//Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
//       indices l and r (l < r)
//Output: A partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l;   j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Note that index *i* can go out of the subarray's bounds in this pseudocode. Rather than checking for this possibility every time index *i* is incremented, we can append to array$A[0..n-1]$a "sentinel" that would prevent index *i* from advancing beyond position *n*. Note that the more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Figure.



(a)



(b)

Analysis:

Best case: If all the splits happen in the middle of corresponding sub arrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \text{ for } n > 1, C_{best}(1) = 0.$$

From the recuurance relation,

$C(n) = 2\,C(n/2) + n$

$C(n) = 2[2\,C(n/4) + n] + n = 2^2\,C(n/2^2) + 2n$

;

$C(n) = 2^k\,C(n/2^k) + kn$

We know $n= 2^k$, so $C(n)= n.\ C(n/n) + (\log_2 n)\ n = C(1) *n + n \log n = n \log n$

$$Cbest(n) = n \log n.$$

Worst case: In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n - 1]$ to sort. The total number of key comparisons made will be equal to

$$Cworst(n) = 0 + n + T(n-1) = n(n+1)/2 = n^2$$

or

$$Cworst(n) = (n + 1) + n + \ldots + 3 = (n + 1)(n + 2)/ 2 - 3 \in \_(n^2).$$

Average case: Let $Cavg(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s $(0 \le s \le n-1)$ after $n+1$comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$Cavg(n) = 1nn-1s=0[(n + 1) + Cavg(s) + Cavg(n - 1 - s)] \text{ for } n > 1,$$

$$Cavg(0) = 0,\ Cavg(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$Cavg(n) \approx 2n \ln n \approx 1.39n \log2 n.$$

Thus, on the average, quicksort makes only 39% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it usually runs faster than mergesort on randomly ordered arrays of nontrivial sizes.

2.6 Stassen's matrix multiplication

Problem definition: Let A and B be two nxn matrices he product matrix C=AB is also an nxn matrix whose i, jth element is formed by taking the elements in the i th row of A and the j th column of B and multiplying them to get for all i and j between 1 to n.

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

To compute C(i,j) using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, is $\boldsymbol{\theta(n^{3)}}$.

The divide and conquer strategy suggests another way to compute the product of two nxn matrices. For simplicity we assume that n is a power of two, that is, that there exists a non negative integer k such that $n=2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two. If A and B are each partitioned in to four square sub matrices, each sub matrix having dimensions n/2 x n/2. Then the product AB can be computed by using the above formula for the product of two matrices. If and B are defined as below

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

To compute the AB using above formula we need to perform 8 multiplications of n/2 x n/2 matrices and four additions of n/2 x n/2 matrices. Since two n/2 x n/2 matrices can be added in C n^2 for the constant C. Therefore T(n)can be given as

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

Where b and c are constants.

$$
\begin{aligned}
P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q &= (A_{21} + A_{22})B_{11} \\
R &= A_{11}(B_{12} - B_{22}) \\
S &= A_{22}(B_{21} - B_{11}) \\
T &= (A_{11} + A_{12})B_{22} \\
U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
V &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= P + S - T + V \\
C_{12} &= R + T \\
C_{21} &= Q + S \\
C_{22} &= P + R - Q + U
\end{aligned}
$$

The resulting recurrence relation for $T(n)$ is

$$
T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}
$$

where a and b are constants. Working with this formula, we get

$$
\begin{aligned}
T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\
&\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \ c \text{ a constant} \\
&= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\
&= O(n^{\log_2 7}) \approx O(n^{2.81})
\end{aligned}
$$

Compared to regular multiplication, this method uses 7 multiplication and 18 additions or subtractions. In this method first we compute P,Q ,R,S T ,U and V (7 multiplications) and 10 matrix additions or subtractions

1. Consider the following matrices and compute the product using Strassen's Matrix multiplication:

$$A = \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix}$$

P= (2+5) (1+7)= 7* 8= 56

Q= (3+5)* 1= 8

R= 2* (3-7)= -8

S= 5*(4-1)= 15

T= (2+4) *7= 42

U= (3-2)* (1+3) = 4

V= (4-5)* (4+7)= -11

C11 = 56+ 15-42-11= 18

C12= -8+ 42= 34

C21= 8+15 = 23

C22= 56-8 -8+4 = 44

2.7 Advantages and disadvantages of divide and conquer

Divide and conquer method is a top down technique for designing an algorithm which consists of dividing the problem in to smaller sub problems hoping that the solutions of the sub problems are easier to find. The solution of all smaller problems is then combined to get a solution for the original problem.

Advantages:

- The difficult problem is broken down to sub problems and each problem is solved separately and independently. This is useful for obtaining solutions in easier way for difficult problems.
- This technique facilitates the discovers of new efficient algorithms. Example: Quick sort, Merge sort etc
- The sub problems can be executed on parallel processor.
- Hence time complexity can be reduced.

Disadvantages

- Large number of sub lists are created and need to be processed
- This algorithm makes use of recursive methods and the recursion is slow and complex.
- Difficulties in solving larger size of inputs