1 Basics of Algorithms and Mathematics

Syllabus

What is an algorithm ?, Mathematics for algorithmic sets, Functions and relations, Vectors and matrices, Linear inequalities and linear equations.

Contents

1.1 Introduction

An algorithm, named for the ninth century Persian mathematician **al-Khowarizmi**, is simply a set of rules used to perform some calculations, either by hand or more usually on a machine. In this subject we will refer algorithm as a method that can be used by the computer for solution of a problem. For instance performing the addition, multiplication, division or subtraction is an algorithm. Use of algorithm for some computations is a common practice. Even ancient Greek has used an algorithm which is popularly known as Euclid's algorithm for calculating the greatest common divisor of two numbers.

Basically algorithm is a finite set of instructions that can be used to perform certain task. In this chapter we will learn some basic concepts of algorithm. We will start our discussion by understanding the notion of algorithm. And for understanding how to write an algorithm we will discuss some examples.

1.2 What is an Algorithm ?

GTU : Dec.-10, Winter-15, Marks 3

In this section we will first understand ***"What is algorithm?"*** and ***"When it is required ?"***

> **Definition of algorithm :** The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

This definition of algorithm is represented in Fig. 1.2.1.

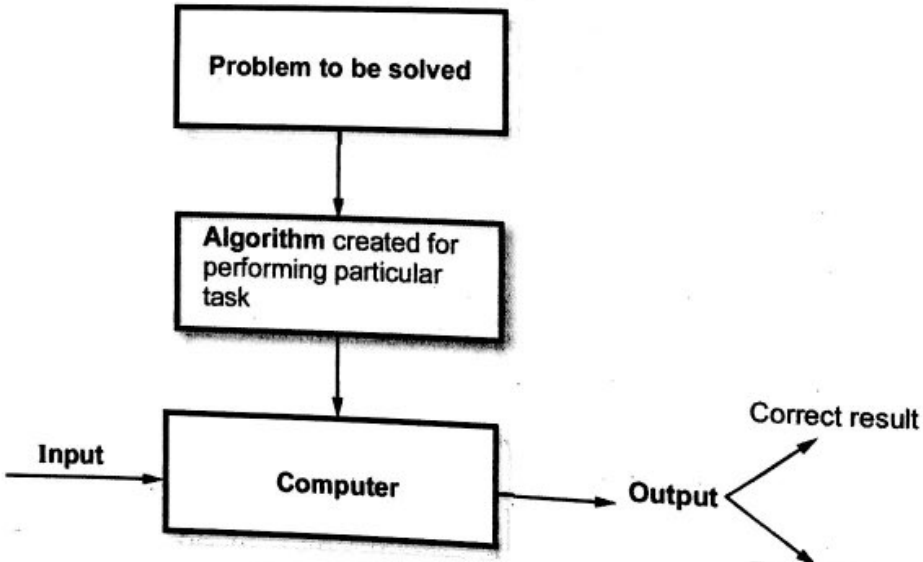


Fig. 1.2.1 Notion of algorithm

After understanding the problem statement we have to create an algorithm carefully for the given problem. The algorithm is then converted into some programming language and then given to some computing device (computer). The computer then executes this algorithm which is actually submitted in the form of source program. During the process of execution it requires certain set of input. With the help of algorithm (in the form of program) and input set, the result is produced as an output. If the given input is invalid then it should raise appropriate error message ; otherwise correct result will be produced as an output.

1.2.1 Properties of Algorithm

Simply writing the sequence of instructions as an algorithm is not sufficient to accomplish certain task. It is necessary to have following properties associated with an algorithm :

1. **Non-ambiguity :** Each step in an algorithm should be non-ambiguous. That means each instruction should be clear and precise. The instruction in an algorithm should not denote any conflicting meaning. This property also indicate the *effectiveness* of algorithm.
2. **Range of input :** The range of input should be specified. This is because normally the algorithm is input driven and if the range of the input is not been specified then algorithm can go in an infinite state.
3. **Multiplicity :** The same algorithm can be represented in several different ways. That means we can write in simple English the sequence of instructions or we can write it in the form of pseudo code. Similarly for solving the same problem we can write several different algorithms. For instance : for searching a number from the given list we can use sequential search or a binary search method. Here *"searching"* is a task and use of either a *"sequential search method"* or *"binary search method"* is an algorithm.
4. **Speed :** The algorithms are written using some specific ideas (which is popularly known as logic of algorithm). But such algorithms should be efficient and should produce the output with fast speed.
5. **Finiteness :** The algorithm should be finite. That means after performing required operations it should terminate.

s : = s + a[i] ; ← O(n)
returns ; ← O(1)

Hence the space complexity of given algorithm can be denoted in terms of big-oh notation. It is **O(n).**

Review Question

1. *Explain why analysis of algorithms is important ?* **GTU : Winter-12, Marks 7**

2.2 Average and Worst Case Analysis

GTU : Dec.-10, May-12, Marks 4

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called **best case** time complexity.

For example : While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.

If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called **worst case** time complexity.

For example : While searching an element by using linear searching method if desired element is placed at the end of the list then we get worst time complexity.

The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called **average case** time complexity.

Consider the following algorithm

```
Algorithm Seq_search(X[0 ... n – 1 ],key)
// Problem Description : This algorithm is for searching the
//key element from an array X[0...n – 1] sequentially.
//Input : An array X[0...n – 1] and search key
//Output : Returns the index of X where key value is present
for i ← 0 to n – 1 do
 if(X[i]=key)then
 return i
```

Best case time complexity

Best case time complexity is a time complexity when an algorithm runs for short time. In above searching algorithm the element **key** is searched from the list of n elements. If the **key** element is present at first location in the list(X[0...n–1]) then algorithm runs for a very short time and thereby we will get the best case time complexity. We can denote the best case time complexity as

$$C_{best} = 1$$

Worst case time complexity

Worst case time complexity is a time complexity when algorithm runs for a longest time. In above searching algorithm the element **key** is searched from the list of **n** elements. If the **key** element is present at n^{th} location then clearly the algorithm will run for longest time and thereby we will get the worst case time complexity. We can denote the worst case time complexity as

$$C_{worst} = n$$

The algorithm guarantees that for any instance of input which is of size n, the running time will not exceed $C_{worst}(n)$. Hence the worst case time complexity gives important information about the efficiency of algorithm.

Average case time complexity

This type of complexity gives information about the behaviour of an algorithm on specific or random input. Let us understand some terminologies that are required for computing average case time complexity.

Let the algorithm is for sequential search and

P be a probability of getting successful search.

n is the total number of elements in the list.

The first match of the element will occur at i^{th} location. Hence probability of occurring first match is P/n for every i^{th} element.

The probability of getting unsuccessful search is (1 – P).

Now, we can find average case time complexity $C_{avg}(n)$ as -

$C_{avg}(n)$ = Probability of successful search (for elements 1 to n in the list)
+ Probability of unsuccessful search

$$C_{avg}(n) = \left[1\cdot\frac{P}{n} + 2\cdot\frac{P}{n} + \ldots + i\cdot\frac{P}{n}\right] + n\cdot(1-P)$$

$$= \frac{P}{n}[1+2+\ldots+i\ldots n] + n(1-P)$$

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

There may be n elements at which chances of 'not getting element' are possible.

$$\boxed{C_{avg}(n) = \frac{P(n+1)}{2} + n(1-P)}$$

Thus we can obtain the general formula for computing average case time complexity.

The total time required by the algorithm to execute can be expressed as,

$t \leq at1 + bt2 + ct3$ or

$t \leq \max \{t1, t2, t3\} \times \{a + b + c\}$

Thus t is bounded by a constant multiple of time taken by elementary operations to execute.

Example

```
Algorithm SUM(n)
{
    //Problem Description: This algorithm calculates the sum of integers from 1 to n
    //Input: Integer values from 1 to n
    //Output: returns the sum value
    sum ← 0
    for (i ← 1 to n ) do
        sum ← sum+i
    return sum;
}
```

In above algorithm the elementary operation is **addition**. If a machine of 32-bit words is used to execute above algorithm, then all the additions can be executed directly provided n with no greater than 65535. Theoretically we consider that the additions costs for **n units.**

2.4 Asymptotic Notations

GTU : May-11,12, Dec.-10,11, Summer-13,14, Winter-14,15, Marks 6

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.

Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time".

Various notations such as Ω, Θ and O used are called **asymptotic notions.**

2.4.1 Big oh Notation

The **Big oh** notation is denoted by 'O'. It is a method of representing the **upper bound of** algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

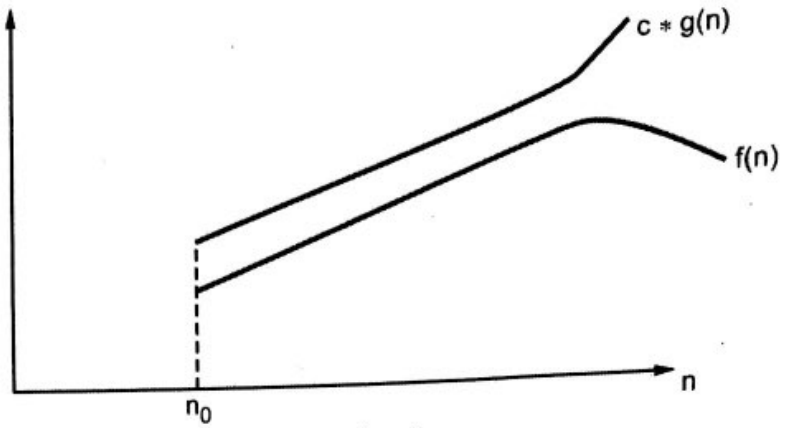
Definition

Let f(n) and g(n) be two non-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$f(n) \leq c*g(n)$$

then f(n) is big oh of g(n). It is also denoted as $f(n) \in O(g(n))$. In other words f(n) is less than g(n) if g(n) is multiple of some constant c.



$f(n) \in O(g(n))$

Fig. 2.4.1

Example : Consider function $f(n) = 2n + 2$ and $g(n) = n^2$. Then we have to find some constant c, so that $f(n) \leq c * g(n)$. As $f(n) = 2n + 2$ and $g(n) = n^2$ then we find c for

n = 1 then,

$$f(n) = 2n+ 2$$
$$= 2(1) + 2$$
$$f(n) = 4$$

and $$g(n) = n^2$$
$$= (1)^2$$
$$g(n) = 1$$

i.e. $$f(n) > g(n)$$

If n = 2 then,

$$f(n) = 2(2) + 2$$
$$= 6$$
$$g(n) = (2)^2$$
$$g(n) = 4$$

i.e. $$f(n) > g(n)$$

If n = 3 then,

$$f(n) = 2(3) + 2$$
$$= 8$$

$g(n) = (3)^2$

$g(n) = 9$

i.e. $f(n) < g(n)$ is true.

Hence we can conclude that for $n > 2$, we obtain

$f(n) < g(n)$

Thus always upper bound of existing time is obtained by big oh notation.

2.4.2 Omega Notation

Omega notation is denoted by 'Ω'. This notation is used to represent the **lower bound** of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition

A function f(n) is said to be in Ω (g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that

$f(n) \geq c * g(n)$ For all $n \geq n_0$

It is denoted as $f(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.

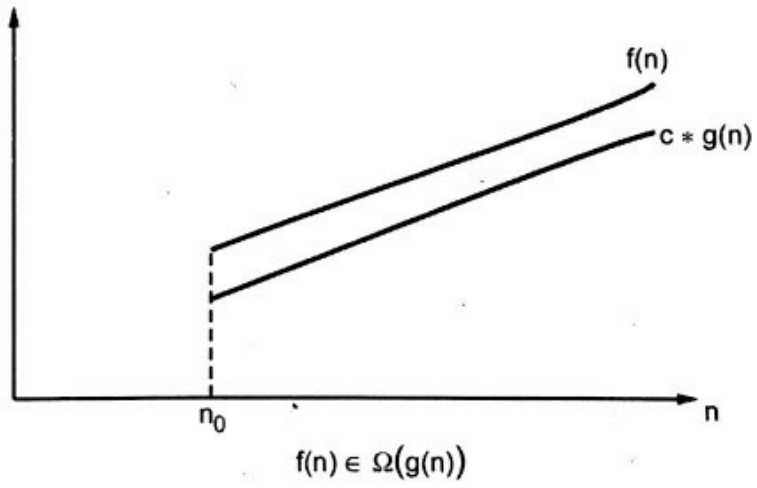


Fig. 2.4.2

Example :

Consider $f(n) = 2n^2 + 5$ and $g(n) = 7n$

Then if $n = 0$

$f(n) = 2(0)^2 + 5$

$= 5$

$g(n) = 7\ (0)$

$= 0$ i.e. $f(n) > g(n)$

But if $n = 1$

$f(n) = 2(1)^2 + 5$

$= 7$

$g(n) = 7(1)$

7 i.e. $f(n) = g(n)$

If n = 3 then,

$f(n) = 2(3)^2 + 5$

$= 18 + 5$

$= 23$

$g(n) = 7(3)$

$= 21$

i.e. $f(n) > g(n)$

Thus for $n > 3$ we get $f(n) > c * g(n)$.

It can be represented as,

$2n^2 + 5 \in \Omega\ (n)$

Similarly any

$n^3 \in \Omega\ (n)^2$

2.4.3 Θ Notation

The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

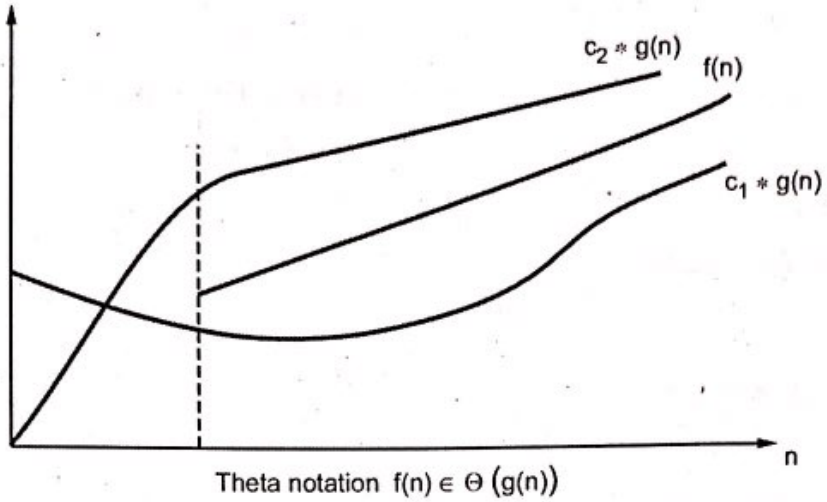
Definition

Let f(n) and g(n) be two non negative functions. There are two positive constants namely c_1 and c_2 such that,

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

Then we can say that,

$$f(n) \in \Theta(g(n))$$



Theta notation $f(n) \in \Theta(g(n))$

Fig. 2.4.3

Example :

If $f(n) = 2n + 8$ and $g(n) = 7n$.

where $n \geq 2$

Similarly $f(n) = 2n + 8$

$g(n) = 7n$

i.e. $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$.

The theta notation is more precise with both big oh and omega notation.

Some examples of asymptotic order

1) $\log_2 n$ is f(n) then

$\log_2 n \in O(n)$ $\because$ $\log_2 n \leq O(n)$, the order of growth of $\log_2 n$ is slower than n.

$\log_2 n \in O(n^2)$ $\because$ $\log_2 n \leq O(n^2)$, the order of growth of $\log_2 n$ is slower than n^2 as well.

But

$\log_2 n \notin \Omega(n)$ $\because$ $\log_2 n \leq \Omega(n)$ and if a certain function f(n) is belonging to $\Omega(n)$ it should satisfy the condition $f(n) \geq c * g(n)$.

Similarly $\log_2 n \notin \Omega(n^2)$ or $\Omega(n^3)$

2) Let $f(n) = n(n-1)/2$

Then,

$n(n-1)/2 \notin O(n)$ $\because$ $f(n) > O(n)$ we get $f(n) = n(n-1)/2 = \frac{n^2-1}{2}$

i.e. maximum order is n^2 which is $> O(n)$.

Hence $f(n) \notin O(n)$

But $n(n-1)/2 \in O(n^2)$ As $f(n) \leq O(n^2)$

and $n(n-1)/2 \in O(n^3)$

Similarly,

$n(n-1)/2 \in \Omega(n)$ $\because$ $f(n) \geq \Omega(n)$

$n(n-1)/2 \in \Omega(n^2)$ $\because$ $f(n) \geq \Omega(n^2)$

$n(n-1)/2 \notin \Omega(n^3)$ $\because$ $f(n) > \Omega(n^3)$

2.4.4 Properties of Order of Growth

1. If $f_1(n)$ is order of $g_1(n)$ and $f_2(n)$ is order of $g_2(n)$, then
   $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))$.
2. Polynomials of degree $m \in \Theta(n^m)$.

   That means maximum degree is considered from the polynomial.

   For example : $a_1n^3 + a_2n^2 + a_3n + c$ has the order of growth $\Theta(n^3)$.
3. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$.
4. Exponential functions a^n have different orders of growth for different values of a.

Key Points *i) O(g(n)) is a class of functions f(n) that grows less fast than g(n), that means f(n) possess the time complexity which is always lesser than the time complexities that g(n) have.*

ii) Θ (g(n)) is a class of functions f(n) that grows at same rate as g(n).

iii) Ω(g(n)) is a class of functions f(n) that grows faster than or atleast as fast as g(n). That means f(n) is greater than $\Omega(g(n))$.

2.4.5 Order of Growth

Measuring the performance of an algorithm in relation with the input size n is called **order of growth**. For example, the order of growth for varying input size of n is as given below.

n	log n	n log n	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Table 2.4.1 Order of growth

We will plot the graph for these values.

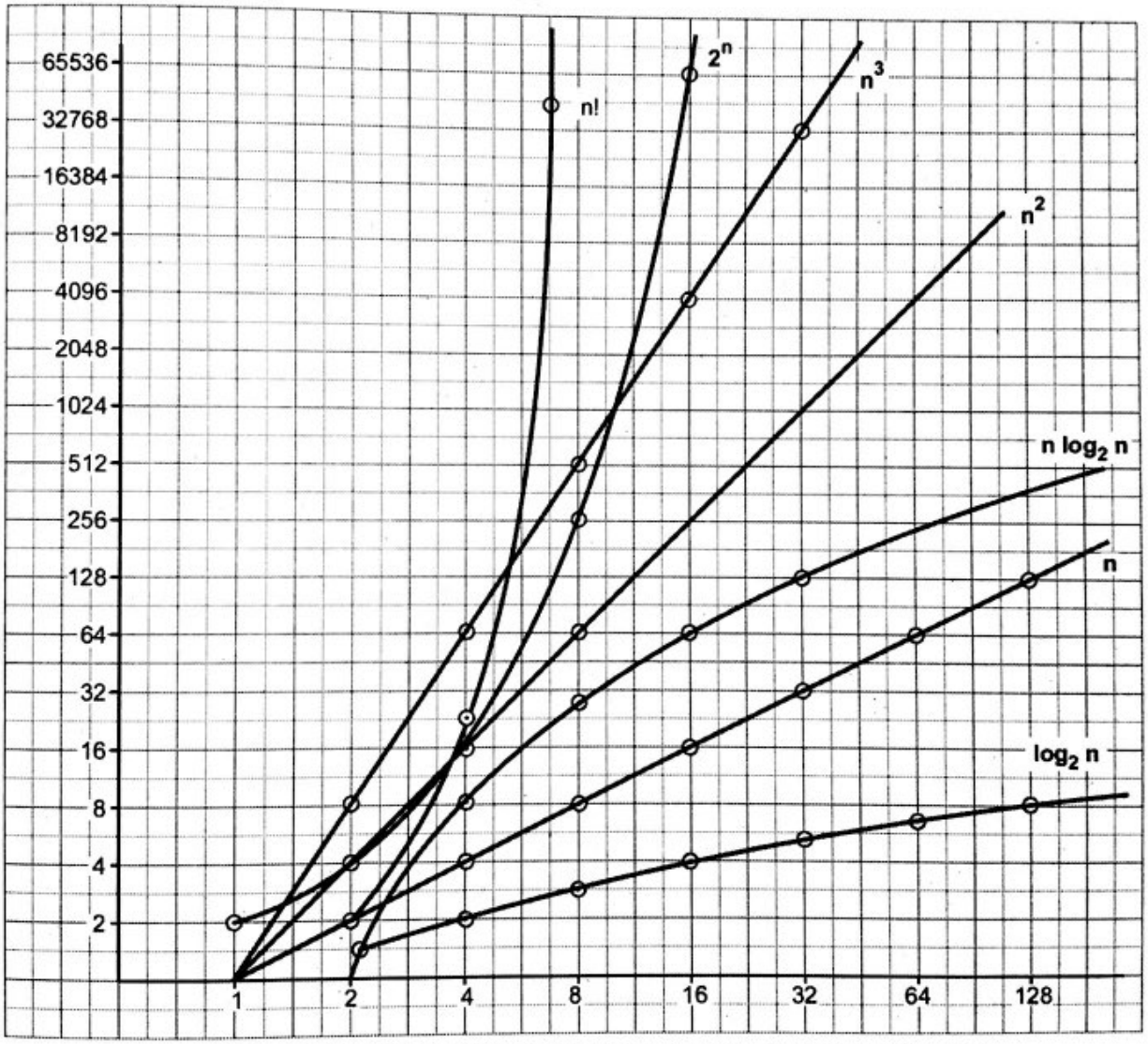


Fig. 2.4.4 Rate of growth of common computing time function

From the above drawn graph it is clear that the logarithmic function is the slowest growing function. And the exponential function 2^n is fastest and grows rapidly with varying input size n. The exponential function gives huge values even for small input n. For instance : for the value of n = 16 we get $2^{16} = 65536$.

Example 2.4.1 *Arrange following rate of growth in increasing order.* **GTU : Dec.-10, Marks 2**

2^n, $n\log n$, n^2, 1, n, $\log n$, $n!$, n^3

Solution : 1, logn, n, nlogn, n^2, n^3, 2^n, n!

Example 2.4.2 *Reorder the following complexity from smallest to largest*

i) $n\log_2(n)$, $n + n^2 + n^3$, 2^4, $sqrt(n)$

ii) n^2, 2^n, $n\log_2(n)$, $\log_2(n)$, n^3

iii) $n\log(n)$, n^8, $n^2/\log n$, $(n^2 - n + 1)$

iv) $n!$, 2^n, $(n+1)!$, 2^{2n}, n^n, $n^{\log n}$

Solution :

i) sqrt(n), $n \log_2 n$, $n + n^2 + n^3$, 2^4

ii) $\log n$, $n \log n$, n^2, n^3, 2^n

iii) $n \log n$, $\frac{n^2}{\log n}$, $(n^2 - n + 1)$, n^8

iv) $n^{\log n}$, 2^n, n!, $(n+1)!$, 2^{2n}, n^n

Example 2.4.3 *Arrange following functions n in increasing order :*

2^n, $\log_2 n$, n^3, $n^{\log 2^n}$, $2^{\log 2^n}$, $n^2 \log_2 n$, $e^{\log 2^n}$, 3^n, 2^{2^n}, $\frac{1}{n}$, $n \log_2 n$

GTU : Summer-14, Marks 4

Solution : The increasing order will be,

$$\frac{1}{n} < \log_2 n < 2^{\log 2n} < e^{\log 2n} < n \log n < n^2 \log_2 n < n^3 < n^{\log 2n} < 2^n < 3^n < 2^{2^n}$$

Example 2.4.4 *Define time complexity and space complexity. Why we are generally concerned with time complexities than space complexities ? What is a major contributor for inefficiency of a loop ? What will be theta notation for :* $4n^3 + 5n + 6$ *?*

GTU : Summer-14, Marks 7

Solution : Refer section 2.1 (Page 2-2) for definitions of time and space complexity.

Time complexity deals with efficient execution of algorithm. The major contributor for inefficiency of loop is the step count of the loop.

Let, $f(n) = 4n^3 + 5n + 6 \in \Theta(n^3)$ where $g(n^3) = n^3$.

To show that $4n^3 + 5n + 6 \in \Theta(n^3)$. We have to find c_1, c_2 and n_0 such that

$0 \le c_1, g(n) \le f(n) \le c_2 g(n)$ for all $n, n \ge n_0$

We can obtain $c_1 = 4$, $c_2 = 15$ for all $n_0 = 1$ and $n \ge 1$.

as $4n^3 \le 4n^3 + 5n + 6 \le 15n^3$

Hence $\mathbf{4n^3 + 5n + 6 \in \Theta(n^3)}$ for all $n \ge 1$

Example 2.4.5 *Explain why the statement "The running time of algorithm A is at least O (n^2)" is meaningless. Also explain what is the smallest value of n such that an algorithm whose running time is 100 n^2 runs faster than an algorithm whose running time is 2^n on the same machine ?* **GTU : Summer-14, Marks 7**

Solution : The f(n) = O (g(n) when

$$f(n) \leq c * g(n) \text{ for all } n > n_0$$

If we assume T(n) be the running time of algorithm A. The statement given in the problem say $T(n) \geq O(n^2)$.

From this we can not extract information about lower bounds and the statement also says nothing about the upper bound which could be $O(n^3)$, $O(2^n)$ etc. Hence the statement is meaningless.

We have to find smallest value of such that $100n^2 < 2^n$. Consider the values ranging 10, 11, 12, ... 20. The calculations will be,

n	$100n^2$	2^n
10	10^4	10^3 approx.
11	1.21×10^4	2.0×10^3 approx.
⋮		
14	1.96×10^4	1.64×10^4
15	2.25×10^4	3.28×10^4
⋮	⋮	⋮
20	4×10^4	10^6 approx.

At **n = 15**, 2^n exceeds $100n^2$.

2.4.6 Summation Formula and Rules used in Efficiency Analysis

1. $\sum_{i=1}^{n} 1 = 1 + 1 + 1 + \ldots + 1 = n \in \Theta(n)$

2. $\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} \in \Theta(n^2)$

3. $\sum_{i=1}^{n} i^k = 1 + 2^k + 3^k + \ldots + n^k \approx \frac{n^{k+1}}{k+1} \in \Theta(n^{k+1})$

4. $\sum_{i=1}^{n} a^i = 1 + a + \ldots + a^n = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$

5. $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$

6. $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$

7. $\sum_{i=k}^{n} 1 = n - k + 1$ where n and k are some upper and lower limits.

Example 2.4.6 *Let f(n) and g(n) be asymptotically positive functions. Prove or disprove following.*

GTU : Dec.-11, Marks 4

$f(n)+g(n) = \Theta(\min(f(n), g(n)))$

Solution : To prove $f(n)+g(n) = \Theta(\min(f(n), g(n)))$

$c_2 \min(f(n), g(n)) <= f(n) + g(n) <= c_1 \min(f(n), g(n))$

If $\min(f(n), g(n)) = f(n)$ then

$c_2 f(n) <= f(n)+g(n) <= c_1 f(n)$ where

c_2 is small and c_1 is large.

Example 2.4.7 *Prove that* $(n+b)^b = \Theta(n^b)$, $b > 0$

GTU : Dec.-11, Marks 4

Solution : To prove this we have to obtain two constants c_1 and c_2 in such a way that $(n+a)^b = \Omega(n^b)$ and $(n+a) = 0(n^b)$

Step 1 :

Let,

$$(n+b)^b \leq (n+|a|)^b, \quad \text{where } n > 0$$

$$\leq (n+n)^b \quad \text{for } n \geq |a|$$

$$= (2n)^b$$

$$= c_1 \cdot n^b \quad \text{where } c_1 = 2^b.$$

$\therefore \quad (n+a)^b = \Omega(n^b)$.

Step 2 :

$$(n+a)^b \geq (n-|a|)^b, \quad \text{where } n > 0$$

$$\geq (c_2' n)^b \quad \text{for } c_2' = 1/2 \quad \text{where } n \geq 2|a|$$

$$\text{as } n/2 \leq n - |a|, \quad \text{for } n \geq 2\,|a|$$

$$\therefore \quad (n+a)^b = 0(n^b)$$

From step 1 and step 2 we get

$c_1 = 2^b$, $c_2 = 2^{-b}$ and $n_0 \geq 2|a|$

Hence

$$(n+a)^b = \Theta(n^b) \quad b>0 \quad \text{is proved.}$$

Example 2.4.8 *Find big oh (O) notation for following :*

1) $f(n) = 6993$ 2) $f(n) = 6n^2 + 135$. **GTU : Dec.-11, Marks 3**

Solution : $f(n) = O(g(n))$ where

$$f(n) < c * g(n)$$

If $c > 1$ and $g(n) = 6993$.

Then $\mathbf{f(n) = O(n)}$

2) As $f(n) = 6n^2 + 135$

$$\mathbf{f(n) = O(n^2)}$$

where $c < 6$ and $n <= 2$.

This is because with these values

$$f(n) <= c * g(n)$$

holds true.

Example 2.4.9 *Answer the following,*

i) Find big theta (θ) and big omega (Ω) notation.

1) $f(n) = 14 * 7 + 83$ 2) $f(n) = 83n^2 + 84n$

ii) Is $2^{n+1} = O(2^n)$? Explain. **GTU : Dec.-11, Marks 7**

Solution : 1) To obtain big omega -

$f(n) = 14 * 7 + 83$ we can write it in polynomial form as

$f(n) = 2n^2 + 11n + 6$ where $n = 7$.

Now if $g(n) = 7(n)$ then

we get $f(n) > 7(n)$ Hence

$$2n^2 + 11n + 6 = \Omega(n)$$

To obtain big theta notation

We have to find out

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n).$$

If we assume $c_1 = 7$ and $c_2 = 26$ then with $n = 7$,

$$7n \leq 2n^2 + 11n + 6 \leq 26n \text{ is true}$$

$\therefore$ $f(n) = \Theta(n)$ where $n = 7$,

$c_1 = 7$ and $c_2 = 26$.

2) $f(n) = 83n^3 + 84n \in \Omega(n^2)$

If $c_1 * g_1(n) \leq f(n) \leq c_2 * g_2(n)$ then $f(n) \in \theta(g(n))$

$\therefore$ $f(n) = \theta(n^3)$ where $c_1 = 83$ and $c_2 = 167$ with $n \geq 1$.

ii) $2^{n+1} = O(2^n)$ is true because

$2^{n+1} = 2.2^n \leq c\ 2^n$, where $c \geq 2$.

Example 2.4.10 *Check equalities (True/False) :*

$5n^2 - 6n \in \Theta(n^2)$

$n! \in O(n^n)$

$2n^2 2^n + n \log n \in \Theta(n^2 2^n)$

$\sum_{i=0}^{n} i^2 \in \Theta(n^3)$

$n^2 \in \Theta(n^3)$

$2^n \in \Theta(2^{n+1})$

$n! \in \Theta((n+1)!)$

GTU : Summer-14, Marks 7

Solution : i) $5n^2 - 6n \in \Theta(n^2)$ True because

$f(n) = 5n^2 - 6n$

$g(n) = n^2$

And $f(n) \leq c * g(n)$ is true.

ii) $f(n) = n! = 1*2*3*4*\ldots*n$

$g(n) = n^n = n*n*n*\ldots*n$

$\therefore$ $f(n) \leq c * g(n)$ is true.

Hence $n! \in O(n^n)$ is false.

iii) $f(n) = 2n^2\,2^n + n \log n$

$g(n) = n^2 2^n$

If n= 16 then

$f(n) = 2\,(16)^2(2)^{16} + 16 \log 16$

$g(n) = (16)^2(2)^{16}$

This shows that,

$f(n) \geq c * g(n)$

Hence $2n^2 2^n + n \log n \in \Theta\,(n^2 2^n)$ is false.

iv) $f(n) = \sum_{i=0}^{n} i^k = 0 + 1 + 2^k + 3^k + \ldots + n^k = \frac{n^{k+1}}{k+1}$

$g(n) = \Theta(n^{k+1})$

$\therefore \quad \sum_{i=0}^{n} i^k \in \Theta(n^{k+1})$ is true.

v) $f(n) = n^2$

$g(n) = n^3$

As $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ is true.

Hence $n^2 \in \Theta\,(n^3)$ is true.

vi) $2^n \in \Theta\,(2^{n+1})$

Let, $2^n = \frac{1}{2} \cdot 2^{n+1}$

$\therefore \quad 2^n \leq 2^{n+1}$

Hence $2^n \in \Theta\,(2^{n+1})$ is true.

vii) $n\,! \in \Theta(n+1)\,!$

$n! = 1 * 2 * 3 \ldots * n$

$(n+1)\,! = (n+1) * n * \ldots\ 2 * 1$

$\therefore \quad n\,! \in \Theta(n+1)\,!$ is false.

Example 2.4.11 *Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For input of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in 64n lg n steps. For which values of n does insertion sort beat merge sort ?* **GTU : Summer - 14, Marks 5**

Solution : Finding insertion sort beat merge sort means time complexity of insertion sort is less than merge sort.

$$\therefore \quad 8n^2 < 64\, n \lg n$$

$$n < 8 \lg n$$

$$\frac{n}{8} < \lg n$$

$$2^{n/8} < n$$

$$2^{n/8} - n < 0$$

$$\therefore \quad 2 \leq n \leq 43$$

Example 2.4.12 *Is 3 log n + log log n is O (log_{10} n) ? Is 3 log n + log log n is Ω (log_{10}n) ?* **GTU : Summer-14, Marks 4**

Solution : We assume n = 1000

Consider RHS

$$3 \log_2 n = 3 \log_2 n\ 1000 = 3 * (10 \text{ approx}) \approx 30$$

$$\log \log n = \log_2 (\log_2 1000) = \log_2(10) \approx 3$$

$$\therefore 3 \log n + \log \log n \approx 30 + 3 \approx 33$$

Consider LHS

$$\log_{10} n = \log_{10} 1000 = 3$$

As $3 \log n + \log \log n > \log_{10} n$

$$3 \log n + \log \log n \in \Omega (\log_{10} n)$$

Examples for Practice

Example 2.4.13 *Interprete the following equations :*
i) $2^n + \Theta(n) = \Theta(n^2)$ *ii)* $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

Example 2.4.14 *Show that i)* $3n+2 = \theta(n)$, *ii)* $6 * 2^n + n^2 = \theta\ (2^n)$

Example 2.4.15 : *Determine the asympotic order of the following functions.*

$$i)\ f(n) = 3x^2 + 5 \quad ii)\ f(n) = \sum_{i=1}^{n} i^2 \quad iii)\ f(n) = 5$$

Example 2.4.16 : *Let $f(n) = 100\ n + 5$. Express $f(n)$ using big omega.*

Review Questions

1. *What do you mean by asymptotic notations ? Explain.* GTU : Dec.-10, Marks 2
2. *Answer the following : Explain asymptotic analysis of algorithm.* GTU : Dec.-11, Marks 3
3. *Define : Big Oh, Big Omega and Big Theta notation.* GTU : May-12, Winter-15, Marks 3
4. *Explain different asymptotic notations in brief.* GTU : June-11, Marks 6
5. *Explain all asymptotic notations used in algorithm analysis.* GTU : Summer-13, Winter-14, Marks 6
6. *Why do we use asymptotic notations in the study of algorithms ? Briefly describe the commonly used asymptotic notations.* GTU : Summer-14, Marks 6

2.5 Recurrence Equation

GTU : May-12,Winter-14,15, Marks 7

The recurrence equation is an equation that defines a sequence recursively. It is normally in following form -

$$T(n) = T(n-1) + n \quad \text{for } n > 0 \qquad \ldots (1)$$

$$T(0) = 0 \qquad \ldots (2)$$

Here equation (1) is called recurrence relation and equation (2) is called initial condition. The recurrence equation can have infinite number of sequences. The **general solution** to the recursive function specifies some formula.

For example : Consider a recurrence relation

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

$$f(1) = 1$$

Then by solving this recurrence relation we get $f(n) = 2^n - 1$. When n = 1, 2, 3 and 4.

2.5.1 Solving Recurrence Equations

The recurrence relation can be solved by following methods -

1. Substitution method
2. Master's method.

Let us discuss methods with suitable examples -

Substitution method -

The substitution method is a kind of method in which a guess for the solution made.

There are two types of substitutions -

- Forward substitution
- Backward substitution.

Forward substitution method - This method makes use of an initial condition in th initial term and value for the next term is generated. This process is continued unt some formula is guessed. Thus in this kind of substitution method, we use recurrenc equations to generate the few terms.

Example 2.5.1 *Solve a recurrence relation* $T(n) = T(n-1) + n$. *With initial condition* $T(0) = 0$ *by forward substitution method.*

Solution : Let,

$$T(n) = T(n-1) + n \quad \ldots (1$$

If n = 1 then

$$T(1) = T(0) + 1$$

$$= 0 + 1 \quad \because \text{Initial condition}$$

$$\therefore \quad \mathbf{T(1) = 1} \quad \ldots (2$$

If n = 2, then

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

$$= 1 + 2$$

$$\therefore \quad \mathbf{T(2) = 3} \quad \because \text{equation (2}$$

If n = 3 then $\ldots (3$

$$T(3) = T(2) + 3$$

$$= 3 + 3$$

$$\therefore \quad \mathbf{T(3) = 6} \quad \because \text{equation (3}$$

By observing above generated equations we can derive a formula, $\ldots (4$

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

We can also denote T(n) in terms of big oh notation as follows -

$$T(n) = O(n^2)$$

But, in practice, it is difficult to guess the pattern from forward substitution. Hence this method is not very often used.

Backward substitution

In this method backward values are substituted recursively in order to derive some formula.

Example 2.5.2 *Solve, a recurrence relation*

$T(n) = T(n-1) + n$ ... (5)

With initial condition $T(0) = 0$ by backward substitution method.

GTU : Winter-15, Marks 7

Solution : $T(n-1) = T(n-1-1) + (n-1)$ ... (6)

Putting equation (6) in equation (5) we get

$$T(n) = T(n-2) + (n-1) + n \quad \text{... (7)}$$

Let

$$T(n-2) = T(n-2-1) + (n-2) \quad \text{... (8)}$$

Putting equation (8) in equation (7) we get.

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

$$\vdots$$

$$= T(n-k) + (n-k+1) + (n-k+2) + \ldots + n$$

if $k = n$ then

$$T(n) = T(0) + 1 + 2 + \ldots n$$

$$T(n) = 0 + 1 + 2 + \ldots + n \qquad \because T(0) = 0$$

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

Again we can denote T(n) in terms of big oh notation as

$$\mathbf{T(n) \in O(n^2)}$$

Some Examples on Recurrence Relation

Example 2.5.3 *Solve the following recurrence relation $T(n) = T(n-1) + 1$ with $T(0) = 0$ as initial condition. Also find big oh notation.*

Solution : Let,

$$T(n) = T(n-1) + 1$$

By backward substitution,

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

$$\therefore \quad T(n) = T(n-1) + 1$$

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

$$\mathbf{T(n) = T(n-2) + 2}$$

Again $T(n-2) = T(n-2-1) + 1$

$$= T(n-3) + 1$$

$$\therefore \quad T(n) = T(n-2) + 2$$

$$= (T(n-3) + 1) + 2$$

$$T(n) = T(n-3) + 3$$

$$\vdots$$

$$T(n) = T(n-k) + k \quad \ldots (1)$$

If $k = n$ then equation (1) becomes

$$T(n) = T(0) + n$$

$$= 0 + n$$

$$T(n) = n$$

$\therefore$ We can denote T(n) in terms of big oh notation as

$$\mathbf{T(n) = O(n)}$$

Example 2.5.4 *Solve the following recurrence relations :*

a) $x(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$

b) $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$

c) $x(n) = x(n/2) + n$ for $n > 1$, $x(1) = 1$ (Solve for $n = 2^k$)

d) $x(n) = x(n/3) + 1$ for $n > 1$, $x(1) = 1$ (Solve for $n = 3^k$)

Solution : a) Let

$$x(n) = x(n-1) + 5$$

$$= [x(n-2) + 5] + 5$$

$$= [x(n-3) + 5] + 5 + 5 = x(n-3) + 5 * 3$$

$$\ldots$$

$= x(n-i) + 5 * i$

...

If $i = n-1$ then

$= x(n-(n-1)) + 5 * (n-1)$

$= x(1) + 5(n-1)$

$= 0 + 5(n-1)$ $\because x(1) = 0$

$\therefore$ $\boxed{x(n) = 5(n-1)}$

b) $x(n) = 3x(n-1)$

$= 3[3x(n-2)] = 3^2 \cdot x(n-2)$

$= 3 \cdot 3[3x(n-3)] = 3^3 \cdot x(n-3)$

...

$= 3^i x(n-i)$

If we put $i = n-1$ then

$= 3^{(n-1)} x(n-(n-1))$

$= 3^{(n-1)} x(1)$

$\boxed{x(n) = 3^{(n-1)} \cdot 4}$ $\because x(1) = 4$

c) Put $n = 2^k$, then

$$x(2^k) = x\left[\frac{2^k}{2}\right] + 2^k$$

$x(2^k) = x(2^{k-1}) + 2^k$

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

$= x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k$

...

$= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \ldots + 2^k$

...

$= x(2^{k-k}) + 2^1 + 2^2 + \ldots + 2^k$

$= x(2^0) + 2^1 + 2^2 + \ldots + 2^k$

$$= 1+2^1+2^2+\ldots+2^k \quad \because \quad x(1) = 1$$

$$= 2^{k+1}-1 \quad \because \quad 2^k = n$$

$$= 2\cdot 2^k - 1$$

$$\therefore \quad \boxed{x(2^k) = 2\cdot n - 1}$$

d) Let $x(n) = x(n/3) + 1$

Assume $n = 3^k$

$$x(3^k) = x(3^{k-1})+1$$

$$= [x(3^{k-2})+1] + 1 = x3^{k-2}+2$$

$$= [x(3^{k-3})+1] + 2 = x3^{k-3}+3$$

$$\ldots$$

$$= x(3^{k-i})+i$$

If we put i = k then,

$$= x(3^{k-k})+k$$

$$= x(1) + k$$

$$= 1 + k \quad \because x(1) = 1$$

$$\therefore \quad \boxed{x(3^k) = 1 + \log_3 n} \quad \because 3^k = n \text{ and } \log_3 n = k$$

Example 2.5.5 *Prove :* $3n^3+2n^2 = O(n^3)$; $3^n \neq O(2^n)$.

Solution : The big-Oh notation denotes the upper bound of algorithm's running tim Using big oh notation we can give longest amount of time taken by the algorithm complete.

Hence we can write

$$F(n) \leq c * g(n)$$

Then $F(n) \in O(g(n))$

In short we will find the values of n,c such that

$$F(n) \leq c * g(n)$$

remains true

Assume $F(n) = 3n^3 + 2n^2$

$g(n) = n^3$

Then for $n >= 2$ and $c = 4$ ($F(n) \in O(g(n))$ is true. That is when $n = 2$ and $c = 4$

$F(n) = 3n^3 + 2n^2$

$= 3(2)^3 + 2(2)^2$

$= 32 \quad \rightarrow$ L.H.S.

$g(n) = n^3$

$= 8$

$c * g(n) = 4 \times 8$

$= 32 \quad \rightarrow$ R.H.S.

L.H.S. = R.H.S. is thus proved.

But when

$F(n) = 3^n$

$g(n) = 2^n$ then let us find

$3^n \leq c * 2^n$

i.e. $\frac{3^n}{2^n} \leq c = \left(\frac{3}{2}\right)^n \leq c$

But there is no such value of c which is $\geq \left(\frac{3}{2}\right)^n$. Hence $3^n \neq 0(2^n)$. In other words $3^n < c * (2^n)$ Will never be true (However $3^n > 2^n$ always).

Example 2.5.6 *If* $T_1(n) = O(f(n))$ *and* $T_2(n) = O(g(n))$ *then show that* $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$

Solution : Let there be some constant c_1 such that

$T_1(n) \leq c_1 g_1(n) \quad$ for $n \geq n_1$ ... (1)

Similarly there will be some constant c_2

Such that

$T_2(n) \leq c_2 g_2(n) \quad$ for $n \geq n_2$ ... (2)

Let $c_3 = \max\{c_1, c_2\}$ such that $n \geq \max\{n_1, n_2\}$

Then we can write using equations (1) and (2) as :

$T_1(n) + T_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$

$$\leq (c_1 + c_2)(g_1(n) + g_2(n))$$

$$\leq c_3 (g_1(n) + g_2(n))$$

$$\leq c_3\, 2 \max (g_1(n) + g_2(n))$$

Hence

$$T_1(n) + T_2(n) \in O(\max (g_1(n) + g_2(n)) \text{ is true.}$$

Example 2.5.7 *Prove that - If $f(n) \in O(n)$ then $[f(n)]^2 \in O(n^2)$*

Solution : Let, $F(n) = a(n) + c$ be a linear function

$$[f(n)]^2 = (a(n) + c)^2$$

$$= [a(n)]^2 + 2a(n)c + c^2$$

$$[f(n)]^2 = x^2 + 2xc + c^2 \qquad \because a(n) = x$$

By considering only highest degree of polynomial we get

$$[f(n)]^2 = O(n^2)$$

Hence if $f(n) \in O(n)$ then $[f(n)]^2 \in O(n^2)$ is true.

2) Master's Method

Efficiency analysis using master theorem

Consider the following recurrence relation -

$$\mathbf{T(n) = aT(n/b) + f(n)}$$ where $n \geq d$ and d is some constant.

Then the Master theorem can be stated for efficiency analysis as -

Consider the following recurrence relation -

$$T(n) = aT(n/b) + f(n)$$ where $n \geq d$ and d is some constant.

If $F(n)$ is $\Theta(n^d)$ where $d \geq 0$ in the recurrence relation then,

1. $T(n) = \Theta(n^d)$ if $a < b^d$
2. $T(n) = \Theta(n^d \log n)$ if $a = b$
3. $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

m	k
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10

1. If $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then
   $T(n) = \Theta(n^{\log_b a})$
2. If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then
   $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then
   $T(n) = \Theta(f(n))$

GTU : May-12, Marks

Example 2.5.9 *Solve $T(n) = 2T(n/2) + n \log n$*

Solution :

Here $f(n) = n \log n$

$a = 2$, $b = 2$

$\log_2 2 = 1$

According to case 2 given in above Master theorem

$f(n) = \Theta(n^{\log_2 2} \log^1 n)$ i.e. $k = 1$

Then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$= \Theta(n^{\log_2 2} \log^2 n)$

$= \Theta(n^1 \log^2 n)$

$\therefore T(n) = \Theta(n \log^2 n)$

Example 2.5.10 *$T(n) = 8T(n/2) + n^2$*

Solution : Here $f(n) = n^2$

$a = 8$ and $b = 2$

$\therefore \log_2 8 = 3$

Then according to case 1 of above given Master theorem

$f(n) = O(n^{\log_b a - \varepsilon})$

$= O(n^{\log_2 8 - \varepsilon})$

$= O(n^{3-\varepsilon})$

Then $T(n) = \Theta(n^{\log_b a})$

$\therefore \quad T(n) = \Theta(n^{\log_2 8})$

$T(n) = \Theta(n^3)$

Example 2.5.11 *Solve $T(n) = 9T(n/3) + n^3$*

Solution : Here $a = 9$, $b = 3$ and $f(n) = n^3$

And $\log_3 9 = 2$

According to case 3 in above Master theorem

As $f(n)$ is $= \Omega(n^{\log_3 9 + \varepsilon})$

i.e. $\Omega(n^{2+\varepsilon})$ and we have $f(n) = n^3$

Then $T(n) = \Theta(f(n))$

$T(n) = \Theta(n^3)$

Example 2.5.12 *Solve $T(n) = T(n/2) + 1$*

Solution : Here $a = 1$ and $b = 2$.

and $\log_2 1 = 0$

Now we will analyse $f(n)$ which is = 1.

We assume $f(n) = 2^k$

When $k = 0$ then $f(n) = 2^0 = 1$.

That means according to case 2 of above given Master theorem.

$f(n) = \Theta(n^{\log_b a} \log^k n)$

$= \Theta(n^{\log_2 1} \log^0 n)$

$= \Theta(n^0 \cdot 1)$

$= \Theta(1)$ $\quad \because n^0 = 1$

$\therefore$ We get $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$= \Theta(n^{\log_2 1} \log^{0+1} n)$

$= \Theta(n^0 \cdot \log^1 n)$

$T(n) = \Theta(\log n)$ $\quad \because n^0 = 1$

Example 2.5.13 *Find the order of growth for solutions of the following recurrences.*

a. $T(n) = 4T(n/2) + nT(1) = 1$

b. $T(n) = 4T(n/2) + n^2T(1) = 1$

c. $T(n) = 4T(n/2) + n^3T(1) = 1$

Solution : We can solve the given recurrence using Master's theorem. It is as given below.

If $f(n) \in \Theta(n^d)$ where $d \geq 0$

then

Case 1 : $T(n) = \Theta(n^d)$ if $a < b^d$

Case 2 : $T(n) = \Theta(n^d \log n)$ if $a = b^d$

Case 3 : $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

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

Here $a = 4$, $b = 2$ and $f(n) = n^1$ $\therefore d = 1$

It is matching with case 3 i.e.

$$a > b^d$$

i.e. $4 > 2^1$

$\therefore$ $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4})$

But $\log_2 4 = 2$

$\therefore$ $\boxed{T(n) = \Theta(n^2)}$ is the time complexity.

b) $T(n) = 4T(n/2) + n^2$

Here $a = 4$, $b = 2$ and $f(n) = n^d = n^2$.

$\therefore$ $d = 2$

It is matching with case 2 i.e.

$$a = b^d$$

i.e. $4 = 2^2$

i.e. $4 = 4$

$\therefore$ $T(n) = \Theta(n^d \log n)$

$\therefore$ $\boxed{T(n) = \Theta(n^2 \log n)}$ is the time complexity.

c) $T(n) = 4T(n/2) + n^3$

Here a = 4, b = 2 and $f(n) = n^3$

$\therefore \quad f(n) = n^d$

$\therefore \quad d = 3$

It is matching with case 1.

$a < b^d$

i.e. $4 < 2^3$

i.e. $4 < 8$

$\therefore \quad T(n) = \Theta(n^d)$

$\therefore \quad \boxed{T(n) = \Theta(n^3)}$ is the time complexity.

Example 2.5.14 *Solve the following recurrence equations.*

1. $T(n) = 7T(n/3) + n^2$
2. $T(n) = 4T(n/2) + \log n$

Solution: 1. $T(n) = 7T\left(\frac{n}{3}\right) + n^2$

Here a = 7, b = 3 and $f(n) = n^2$.

$\log_b a = \log_3 7 \approx 1.77$

For f(n), it matches with case 3 of Master's Theorem, i.e.

$f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) = n^2$

$\therefore \quad T(n) = \Theta(f(n)) = \Theta(n^2)$

2. $T(n) = 4T\left(\frac{n}{2}\right) + \log n$

Here a = 4, b = 2 and f(n) = log n.

$\log_b a = \log_2 4 = 2$

We find a match with cases of Master's theorem,

$f(n) = \log n = O\left(n^{\log_b a - \varepsilon}\right) = O(n^{2-\varepsilon})$

$\therefore \quad T(n) = \Theta(n^{\log_b a})$

$\mathbf{T(n) = \Theta(n^2)}$

Example 2.5.15 *Explain master theorm and solve the following recurrence equation with master method -* **GTU [IT] : Winter -14, Marks 7**

1. $T(n) = 9T(n/3) + n$ 2. $T(n) = 3T(n/4) + n \lg n$

Solution : 1. By Master Theorem.

$$a = 9, b = 3, f(n) = n$$

$$n^{\log_b a} = n^{\log 3^9} = \Theta(n^2)$$

because $f(n) = O\left(n^{\log 3^{9-\varepsilon}}\right)$ where $\varepsilon = 1$

The case 1 is applied here.

$$T(n) = \Theta\left(n^{\log_b a}\right) \text{ when}$$

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

Hence $\mathbf{T(n) = \Theta(n^2)}$

2. By Master Theorem,

$$a = 3, b = 4, f(n) = n\lg n$$

$$n \log_b a = n \log_4 3 = \Theta\left(n^{0.793}\right)$$

$$f(n) = \Omega\left(n^{\log_4 3+\varepsilon}\right) \text{ for } \varepsilon = 0.2$$

Besides, for large n the regularity holds for $c = 3/4$

$$a \cdot f(n/b) = 3(n/a)\lg(n/4) \leq (3/4)\, n\lg n = c\, f(n)$$

Thus case 3 is applicable

$$\therefore \quad T(n) = \Theta(f(n))$$

$$\therefore \quad \mathbf{T(n) = \Theta(n\lg n)}$$

Review Question

1. *Explain the concept of recurrence relation with some illustrative example.*
2. *State and explain the Master's theorem.*

2.6 Analyzing Control Statements

GTU : May-12, Marks 4

While analyzing the given fragment of code various programming constructs appear Those are :

1. Sequencing
2. For loops
3. Recursive calls
4. While and repeat loops

$bi \leq b_{i-1} - t_i + 1$

Hence potential change is

$$\Phi(D_i) - \Phi(D_{i-1}) \leq b_{i-1} - t_i + 1 - b_i$$

Potential change = $1 - t_i$

$$\therefore \text{Amortized cost } c'_i = \text{Actual cost} + \text{Potential change}$$
$$= c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$\leq t_i + 1 - 1 - 1 - t_i$$
$$= 2$$

The total **amortized cost** of n operations is **O(n)**

The **worst case** cost is **O(n).**

Review Questions

1. *What is an amortized analysis ? Explain aggregate method of amortized analysis using suitable example.* **GTU : June-11, Winter-14, Marks 7**
2. *What is an amortized analysis ? Explain potential method of amortized analysis using suitable example .* **GTU : Winter -14, Marks 7**
3. *Define an amortized analysis. Briefly explain its different techniques. Carry out aggregate analysis for the problem of implementing a k-bit binary counter that counts upward from 0.* **GTU : Summer -15, Marks 3**

2.8 What Kind of Problems are Solved by Algorithms ?

GTU : May-12, Dec.-10, Winter-14, Marks 7

Generally any computing problem can be solved by algorithm. There is large number of computing problems and some of them can be classified as -

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

2.8.1 Sorting Algorithm

- Sorting means arranging the elements in increasing order or decreasing order (also called **ascending order or descending order** respectively). The sorting can be done on numbers, characters (alphabets), strings or employees record. For sorting any record we need to choose certain piece of information based on which sorting can be done. For instance : for keeping the employee ID. Similarly one can arrange the library books according to the title of the books. This piece of information which

is required to sort the record is called as **key**. The important property of this key is that it should be unique.

- The sorting algorithms are classified as **internal sorting** and **external sorting.** The internal sorting is used for sorting the reasonable amount of data and it can be carried out on main memory whereas the external sorting is applied to sort a huge amount of data and it can be carried out on the main memory along with the secondary memory.
- Various sorting methods are -
  1. Bubble sort
  2. Selection sort
  3. Insertion sort
  4. Quick sort
  5. Merge sort
  6. Heap sort
  7. Radix sort.
- The insertion sort, bubble sort and quick sort are comparison sort algorithms. The radix sort is counting sort algorithm in which it is assumed that input numbers are in the set {1, 2, 3....., n} then using array index the elements are arranged in ordered manner.

2.8.1.1 Bubble Sort

Bubble sort is another simplest sorting algorithm. In bubble sort, largest element is moved at highest index in the array. As the largest element moves to the highest index position it is called that element is **"bubbled up"** from its position. Hence is the name!

In the next pass, the second largest element bubbles up, then the third largest element bubbles up, and so on. After (n – 1) passes the list gets sorted. The pass i ($0 \leq i \leq n-2$) of bubble sort can be represented as

$$A[0], \ldots, A[j] \overset{?}{\longleftrightarrow} A[j+1], \ldots, A[n-i-1] \mid A[n-i] \leq \ldots < A[n-1]$$

Example

Consider the elements

70, 30, 20, 50, 60, 10, 40

We can store these elements in array as

70	30	20	50	60	10	40
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

We will use $\overset{?}{\longleftrightarrow}$ symbol to check whether previous element is greater than next. If we found it greater, then swap them.

Pass 1 :

70	$\overset{?}{\longleftrightarrow}$	30		20		50		60		10		40	
30		70	$\overset{?}{\longleftrightarrow}$	20		50		60		10		40	
30		20		70	$\overset{?}{\longleftrightarrow}$	50		60		10		40	
30		20		50		70	$\overset{?}{\longleftrightarrow}$	60		10		40	
30		20		50		60		70	$\overset{?}{\longleftrightarrow}$	10		40	
30		20		50		60		10		70	$\overset{?}{\longleftrightarrow}$	40	
30		20		50		60		10		40		70	

The list obtained at the end of first pass will be considered as input to second pass.

Pass 2 :

30	$\overset{?}{\longleftrightarrow}$	20		50		60		10		40		70	
20		30	$\overset{?}{\longleftrightarrow}$	50	$\overset{?}{\longleftrightarrow}$	60	$\overset{?}{\longleftrightarrow}$	10		40		70	
20		30		50		10		60	$\overset{?}{\longleftrightarrow}$	40		70	
20		30		50		10		40		60	$\overset{?}{\longleftrightarrow}$	70	
20		30		50		10		40		60		70	

Pass 3 :

20	$\overset{?}{\longleftrightarrow}$	30	$\overset{?}{\longleftrightarrow}$	50	$\overset{?}{\longleftrightarrow}$	10		40		60		70	
20		30		10		50	$\overset{?}{\longleftrightarrow}$	40		60		70	
20		30		10		40		50	$\overset{?}{\longleftrightarrow}$	60	$\overset{?}{\longleftrightarrow}$	70	
20		30		10		40		50		60	$\overset{?}{\longleftrightarrow}$	70	

Pass 4 :

20	$\overset{?}{\longleftrightarrow}$	30	$\overset{?}{\longleftrightarrow}$	10		40		50		60		70	

20		10		30	?	40	?	50	?	60	?	70
20		10		30		40		50	?	60	?	70
20		10		30		40		50		60		70
20		10		30		40		50				

Pass 5 :

20	?	10		30		40		50	?	60	?	70
10		20	?	30	?	40	?	50		60		70
10		20		30		40		50				

Now we get the sorted list as,

10	20	30	40	50	60	70

Example 2.8.1 *Sort the letters of word "DESIGN" in alphabetical order using bubble sort.*

GTU : Summer-14, Marks 7

Solution : Pass 1

D	?	E		S		I		G		N	
D		E	?	S		I		G		N	
D		E		S	?	I		G		N	
D		E		I		S	?	G		N	
D		E		I		G		S	?	N	
D		E		I		G		N		S	

Pass 2

D	?	E		I		G		N		S	
D		E	?	I		G		N		N	
D		E		I	?	G		N		N	
D		E		G		I	?	N		N	
D		E		G		I		N	?	S	
D		E		G		I		N		S	

Continuing in this manner we get the list sorted finally. It is as follows.

D	E	G	I	N	S

Algorithm

```
Algorithm Bubble(A[0...n-1])
//Problem Description : This algorithm is for sorting the
//elements using bubble sort method
//Input:An array of elements A[0...n-1] that is to be sorted
//Output: The sorted array A[0...n-1]
for i ← 0 to n – 2 do
{
 for j ←0 to n–2-i do
 {
          if(A[j]>A[j+1])then
          {
          temp ← A[j]
          A[j] ← A[j+1]
          A[j+1] ← temp
          }
 }//end of inner for loop
}//end of outer for loop
```

Analysis

The above algorithm can be analysed mathematically. We will use general plan for non recursive mathematical analysis.

Step 1 : The input size is total number of elements in the list.

Step 2 : In this algorithm the basic operation is key comparison.

```
if A[j] > A[j+1]
```

Step 3 : We can obtain **sum** as follows :

$$C(n) = \text{Outer for loop with variable i} \times \text{Inner for loop with variable j} \times \text{Basic operation}$$

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

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

$\sum_{i=0}^{n} 1 = (n-0+1)$ using this formula

we get,

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

$= (n-1-i)$

Step 4 : **Simplifying** sum we get,

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

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

$$= \sum_{i=0}^{n-2} (n-1) - \frac{(n-2)(n-1)}{2} \quad \text{... (Refer rule 2 in section 2.4.5 for arriving at this answer)}$$

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

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

If the elements are arranged in decreasing manner, the key swaps will be

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

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

The time complexity of bubble sort is $\Theta(n^2)$.

2.8.1.2 Selection Sort

Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element scan the entire list to find the smallest element and swap it with the second element. Then starting from the third element the entire list is scanned in order to find the next smallest element. Continuing in this fashion we can sort the entire list.

Generally, on pass i $(0 \le i \le n-2)$, the smallest element is searched among last $n-i$ elements and is swapped with A[i]

$$A[0] \le A[1] \le \ldots \le A[i-1] \mid \overbrace{A[i], \ldots, A[k], \ldots, A[n-1]}^{\text{Last n–i elements}}$$

A[k] is smallest element

so swap A[i] and A[k]

The list gets sorted after n–1 passes.

Example

Consider the elements

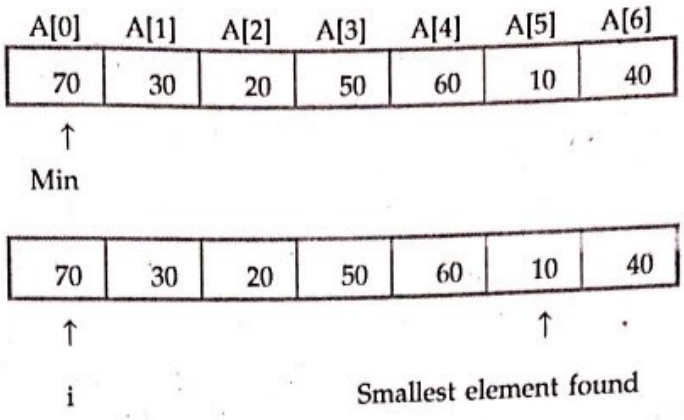
70, 30, 20, 50, 60, 10, 40

We can store these elements in array A as :

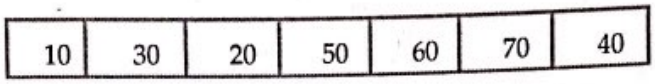
70	30	20	50	60	10	40
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
↑	↑					
Min	j					

Initially set

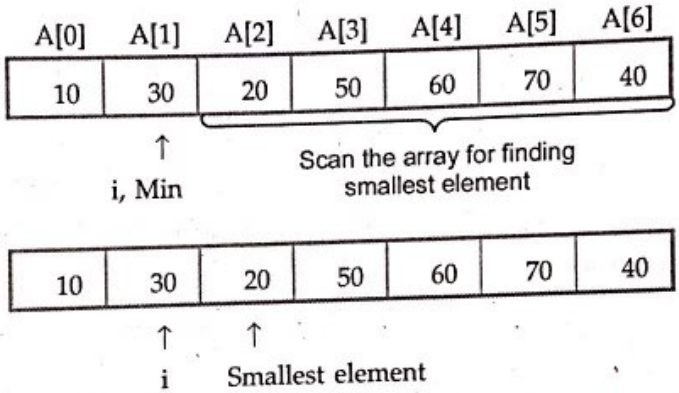
1st pass :



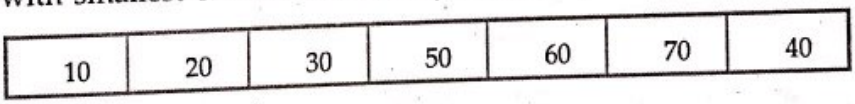
Now swap A[i] with smallest element. Then we get,



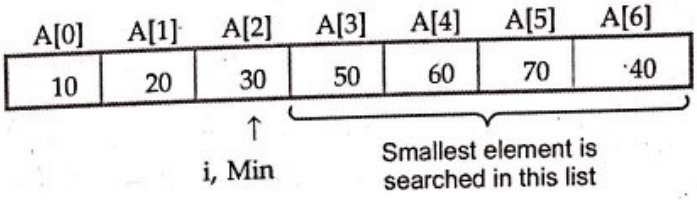
2nd pass :



Swap A[i] with smallest element. The array becomes,

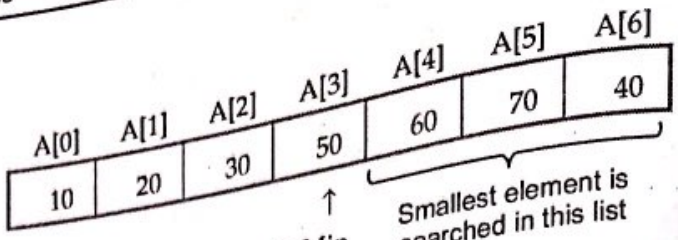


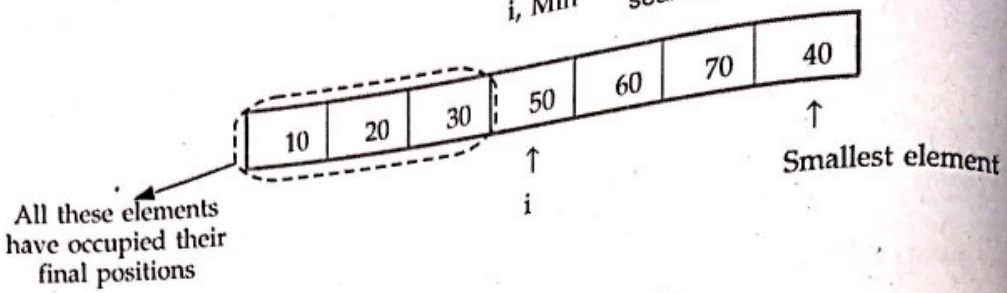
3rd pass :



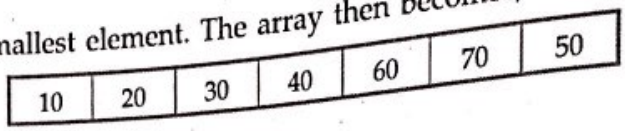
As there is no smallest element than 30 we will increment i pointer.

4th pass :

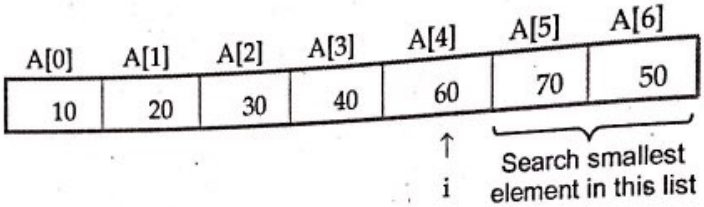


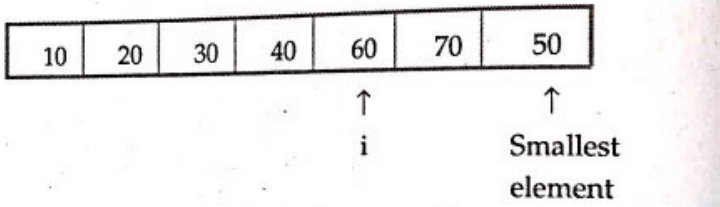


Swap A[i] with smallest element. The array then becomes,

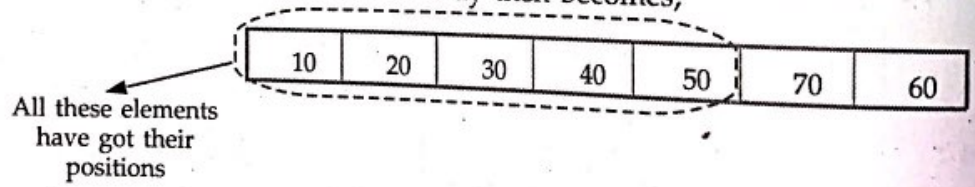


5th pass :

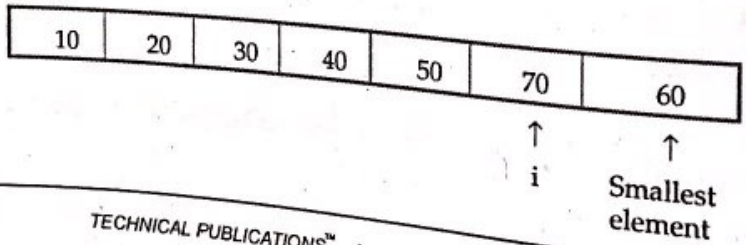




Swap A[i] with smallest element. The array then becomes,



6th pass :



Swap A[i] with smallest element. The array then becomes,

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
10	20	30	40	50	60	70

This is a sorted array.

Algorithm

The pseudo code for sorting the elements using selection sort is as given below.

```
Algorithm selection(A[0...n-1])
//Problem Description: This algorithm sorts the elements
//using selection sort
//Input:An array of elements A[0...n-1] that is to be sorted
//Output: The sorted array A[0...n-1]
for i ← 0 to n–2 do
{
 Min ← i
 for j ← i+1 to n–1 do
 {
          if(A[j]<A[min])then
      min ← j
 //swap A[i] and A[min]
 }//end of inner for loop
 temp ← A[i]
 A[i] ← A[min]
 A[min] ← temp
}//end of outer for loop
```

Analysis

The above algorithm can be analysed mathematically. We will apply a general plan for non recursive mathematical analysis.

Step 1 : The **input size** is n i.e. total number of elements in the list.

Step 2 : In the algorithm the **basic operation** is key comparison.

```
if A[j] < A[min]
```

Step 3 : This basic operation depends only on array size n. Hence we can find **sum** as

$$C(n) = \text{Outer for loop with variable i} \times \text{Inner for loop with variable j} \times \text{Basic operation}$$

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

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

$\sum_{i=0}^{n} 1 = (n - 0 + 1)$ using this formula we g

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

$= (n - 1 - i)$

Step 4 : Simplifying sum we get,

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

$$= \sum_{i=0}^{n-2} (n-1) - \left[\sum_{i=0}^{n-2} i \right]$$

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

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ using this formula

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

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

Now taking $(n - 1)$ as common factor we get,

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

As $\sum_{i=1}^{n} 1 = (n - 1 + 1)$, we g

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

$= (n - 1)$

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

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

Solving this equation we will get,

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

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

$$= \frac{n^2 - n}{2}$$

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

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

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

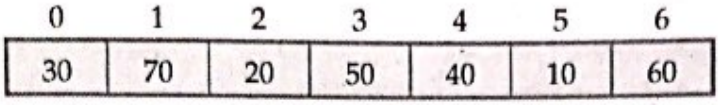
Thus time complexity of selection sort is $\Theta(n^2)$ for all input. But total number of swaps is only $\Theta(n)$.

2.8.1.3 Insertion Sort

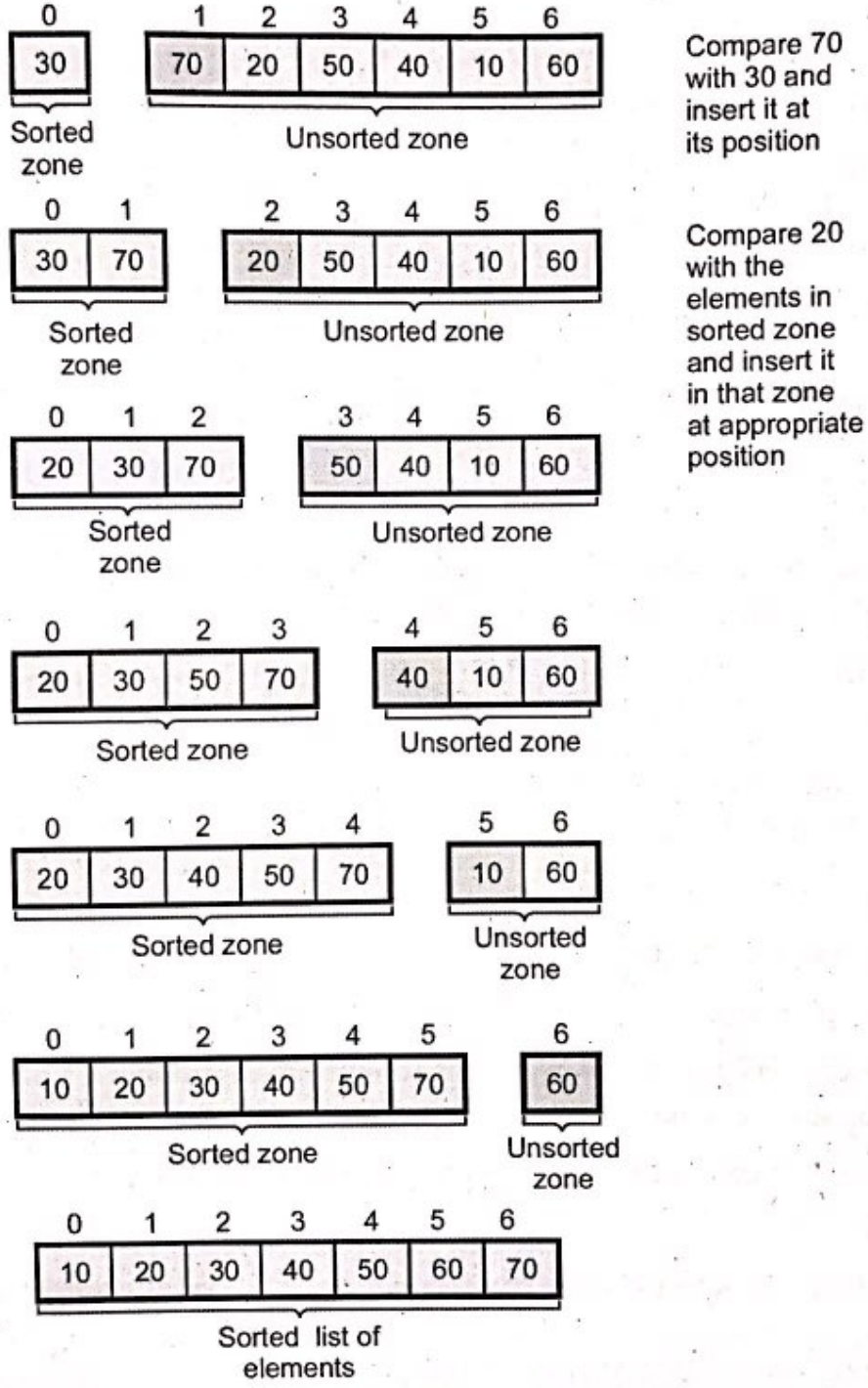
In this method the elements are inserted at their appropriate place. Hence is the name **insertion sort**. Let us understand this method with the help of some example -

For Example

Consider a list of elements as,



The process starts with first element



Algorithm

Although it is very natural to implement insertion using recursive(top down algorithm but it is very efficient to implement it using bottom up(iterative) approach.

```
Algorithm Insert sort(A[0.. n-1])
//Problem Description This algorithm is for sorting the
//elements using insertion sort
//Input An array of n elements
//Output Sorted array A[0.. n-1] in ascending order
 for i ← 1 to n-1 do
{
        temp ← A[i]//mark A[i]th element
        j ← i-1//set j at previous element of A[i]
        while(j>=0)AND(A[j]>temp)do
        {
    //comparing all the previous elements of A[i] with
    //A[i].If any greater element is found then insert
    //it at proper position
        A[j+1] ← A[j]
        j ← j-1
        }
        A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis

When an array of elements is almost sorted then it is **best case** complexity. The best case time complexity of insertion sort is **O(n).**

If an array is randomly distributed then it results in **average case** time complexity which is $O(n^2)$.

If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

Advantages of insertion sort

1. *Simple* to implement.
2. This method is *efficient* when we want to sort small number of elements. And this method has excellent performance on almost sorted list of elements.
3. More efficient than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort.
4. This is a *stable* (does not change the relative order of equal elements).

5. It is called *in-place* sorting algorithm (only requires a constant amount O(1) of extra memory space). The in-place sorting algorithm is an algorithm in which the input is overwritten by output and to execute the sorting method it does not require any more additional space.

C Function

```
void Insert_sort(int A[10],int n)
{
int i,j,temp;
for(i=1;i<=n-1;i++)
{
        temp=A[i];
        j=i-1;
        while((j>=0)&&(A[j]>temp))
        {
         A[j+1]=A[j];
         j=j-1;
        }
        A[j+1]=temp;
}
printf("\n The sorted list of elements is...\n");
for(i=0;i<n;i++)
        printf("\n%d",A[i]);
}
```

Let us now see the implementation of this method using C.

C Program

```
/*****************************************************************
     Implementation of insertion sort
*****************************************************************/
#include<stdio.h>
#include<conio.h>
void main()
{
int A[10],n,i;
void Insert_sort(int A[10],int n);
clrscr();
printf("\n\t\t Insertion Sort");
printf("\n How many elements are there?");
scanf("%d",&n);
printf("\n Enter the elements\n");
for(i=0;i<n;i++)
        scanf("%d",&A[i]);
Insert_sort(A,n);
getch();
```

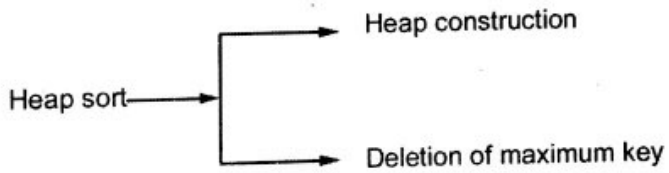
```
}
}
```

Output

```
    Program for Shell Sort
Enter total number of elements in an array 10

Enter some elements in array
10
9
8
7
6
5
4
3
2
1

Before sorting   10 9 8 7 6 5 4 3 2 1

After sorting   1 2 3 4 5 6 7 8 9 10
```

2.8.1.5 Heap Sort

Heap sort is a sorting method discovered by J. W. J. Williams. It works in two stages.



1. **Heap construction :** First construct a heap for given numbers.
2. **Deletion of maximum key :** Delete root key always for (n – 1) times to remaining heap. Hence we will get the elements in decreasing order. For an array implementation of heap, delete the element from heap and put the deleted element in the last position in array. Thus after deleting all the elements one by one, if we collect these deleted elements in an array starting from last index of array.

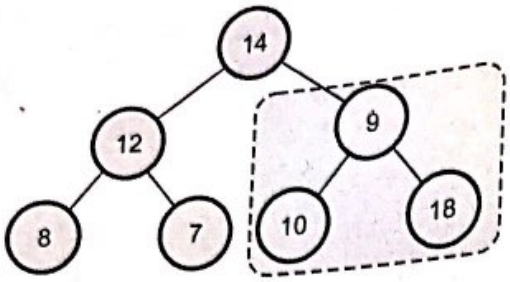
We get a list of elements in a ascending order.

Let us understand this technique with the help of some example.

Sort the following elements using heap sort

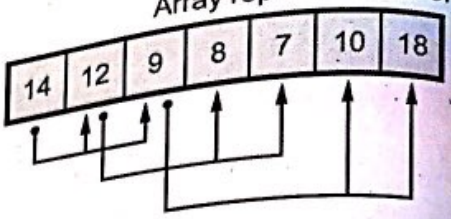
14, 12, 9, 8, 7, 10, 18.

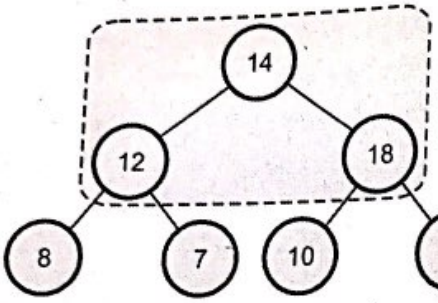
Stage I : Heap construction



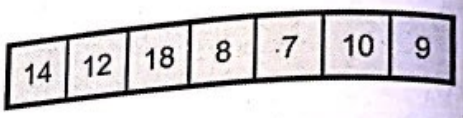
We start heapifying from bottom

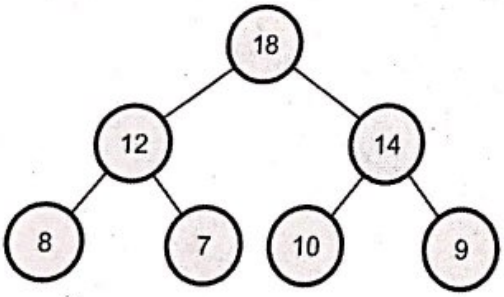
Array representation of heap

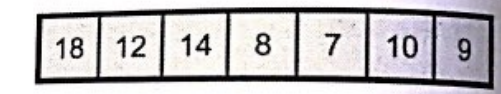




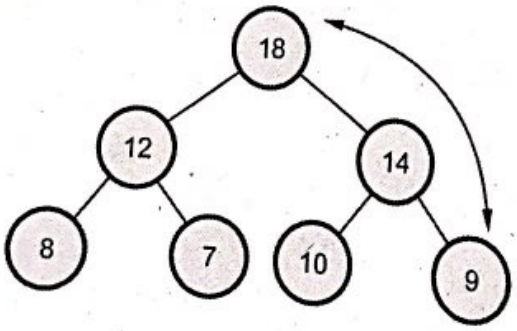
Heapifying this tree

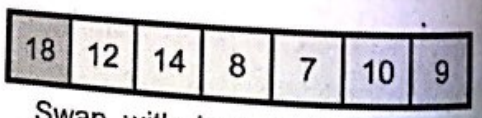




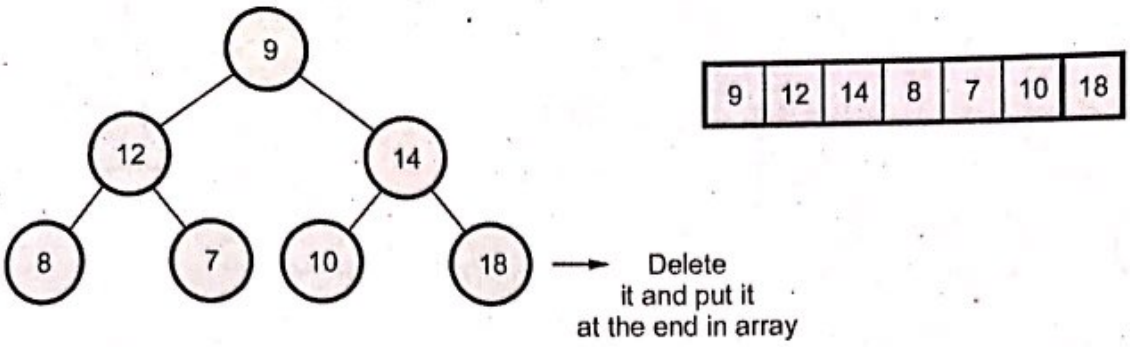


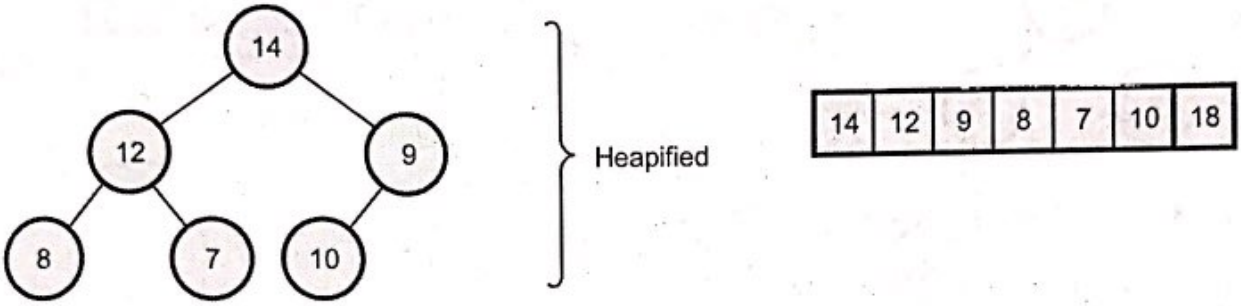
Stage II : Maximum deletion

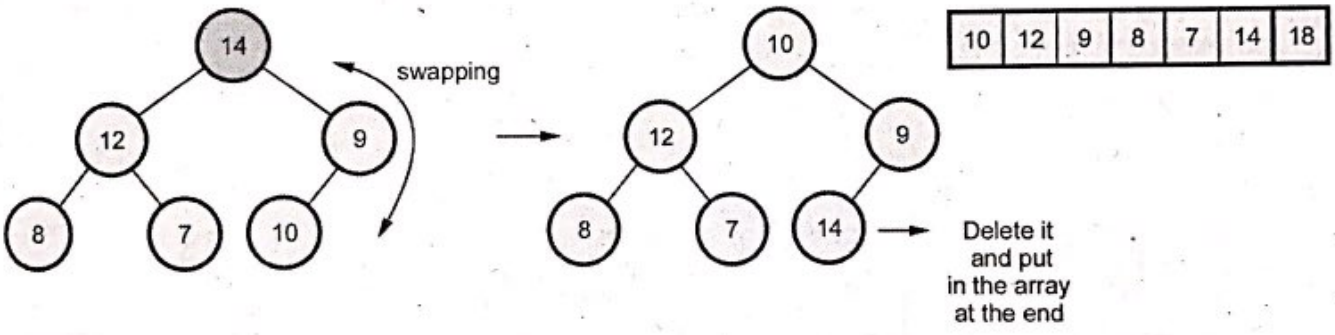


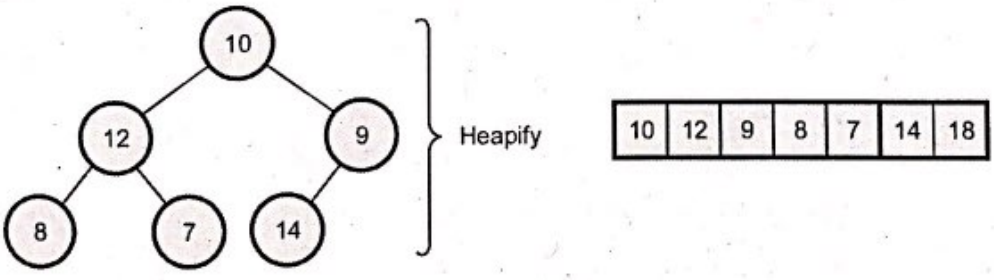


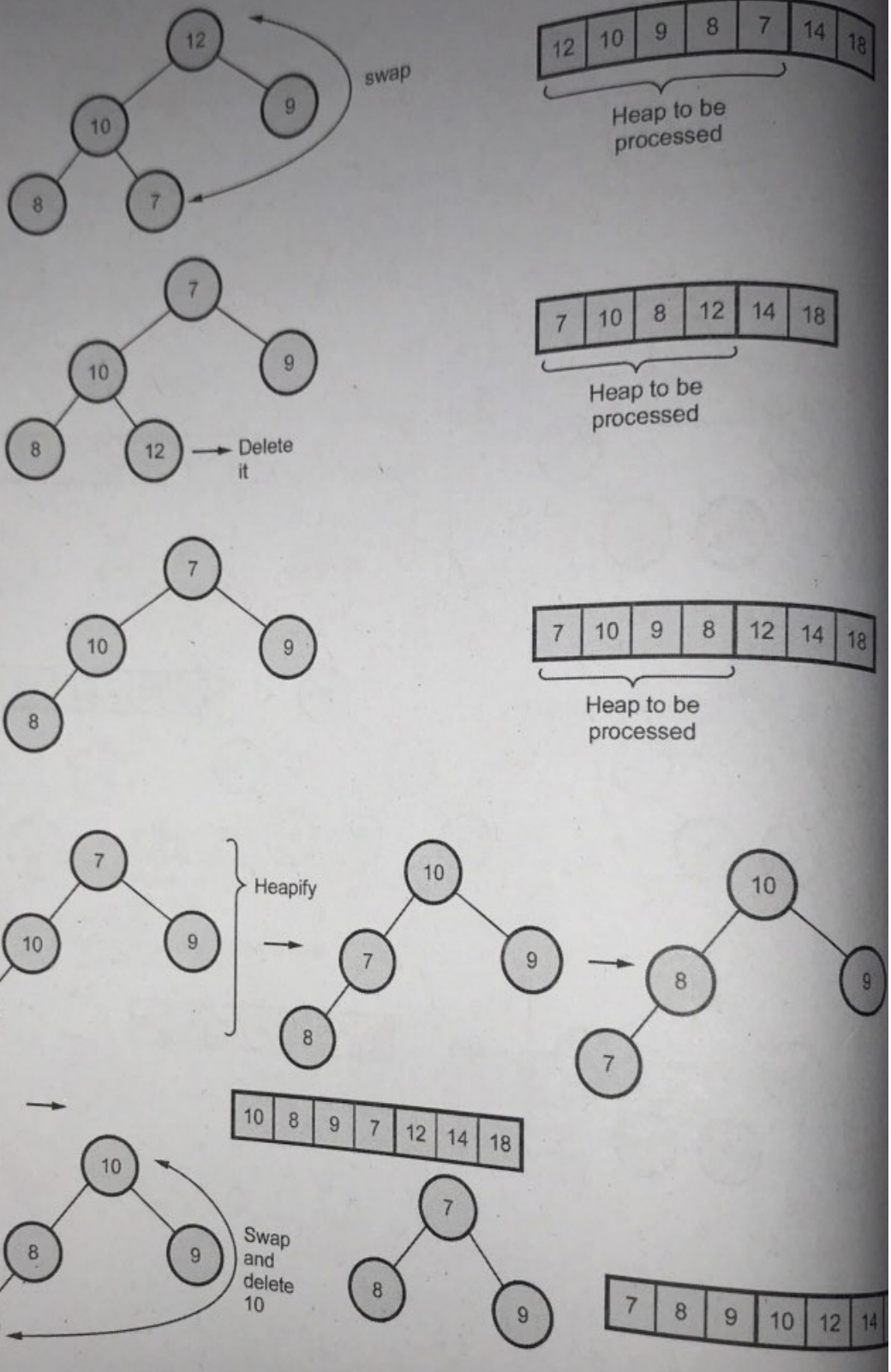
Swap with last node in heap

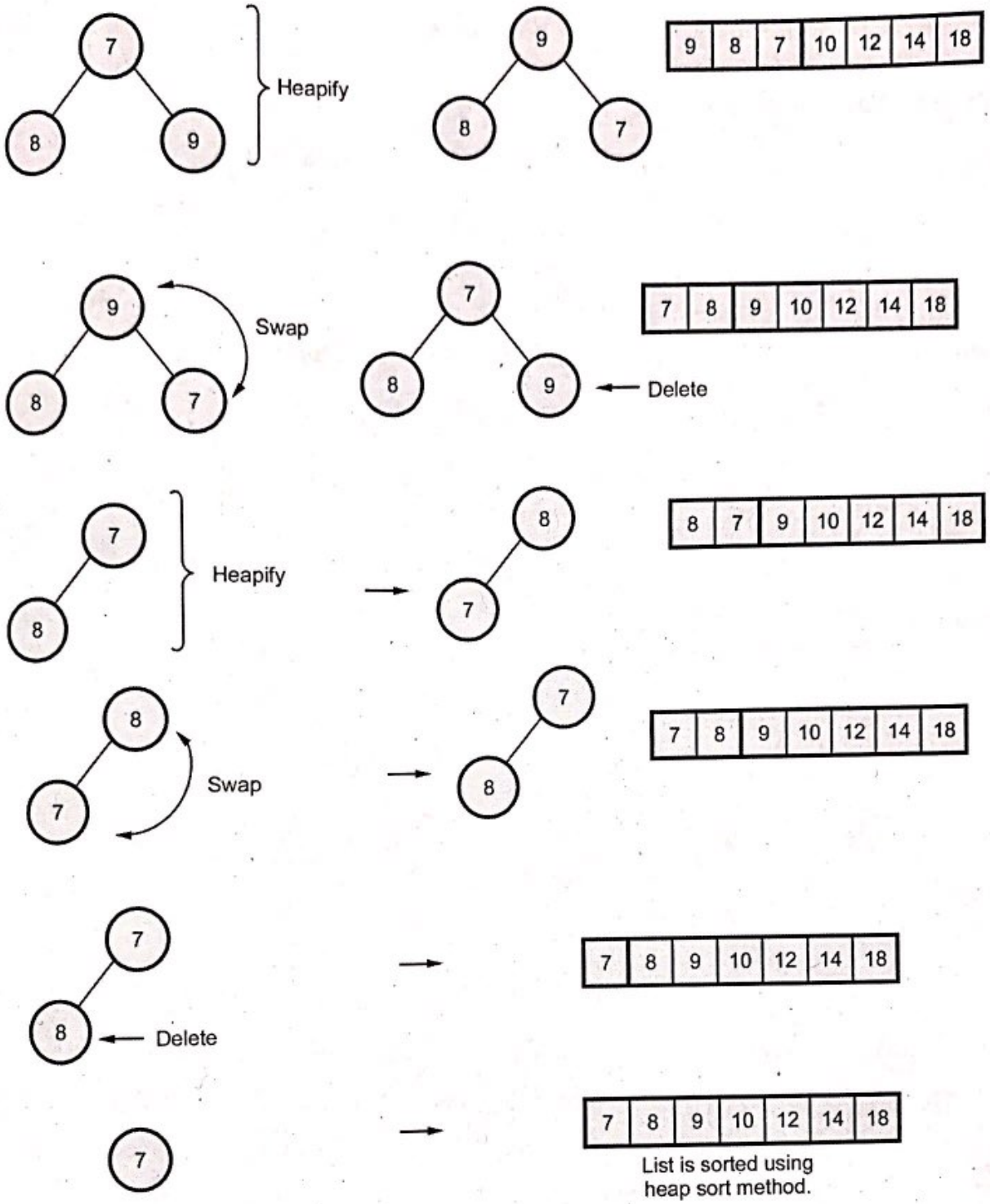












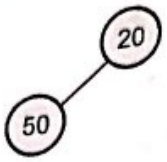
Example 2.8.3 *Sort the given elements with Heap Sort Method: 20, 50, 30, 75, 90, 60, 25, 10, 40.* **GTU : Winter-15, Marks 7**

Solution : The heap sorting is done in two stages

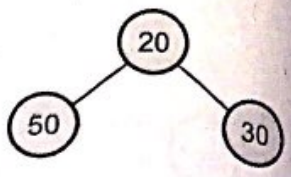
Stage I : Heap construction

Step : 1

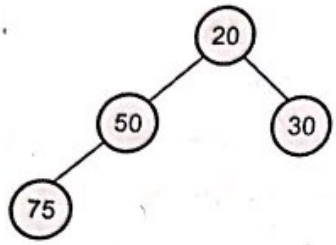
Step : 2



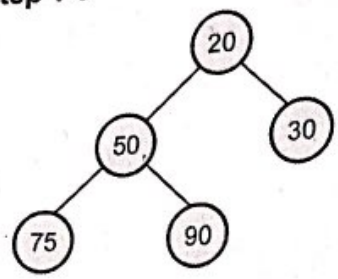
Step : 3



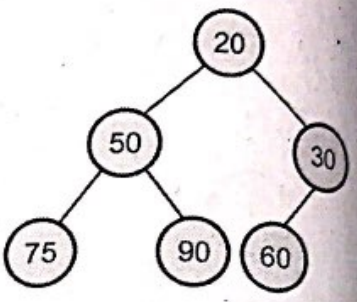
Step : 4



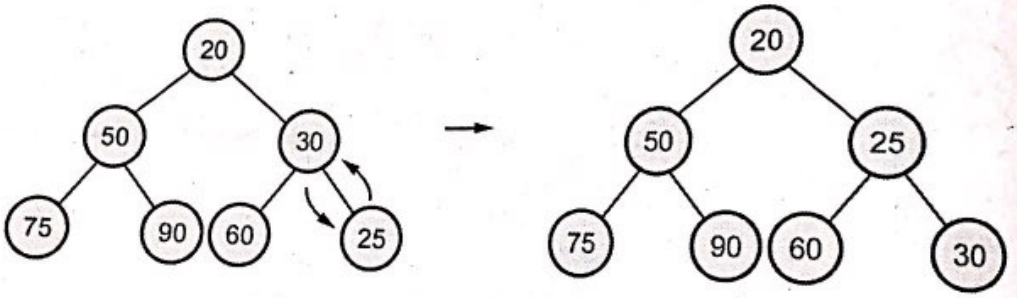
Step : 5



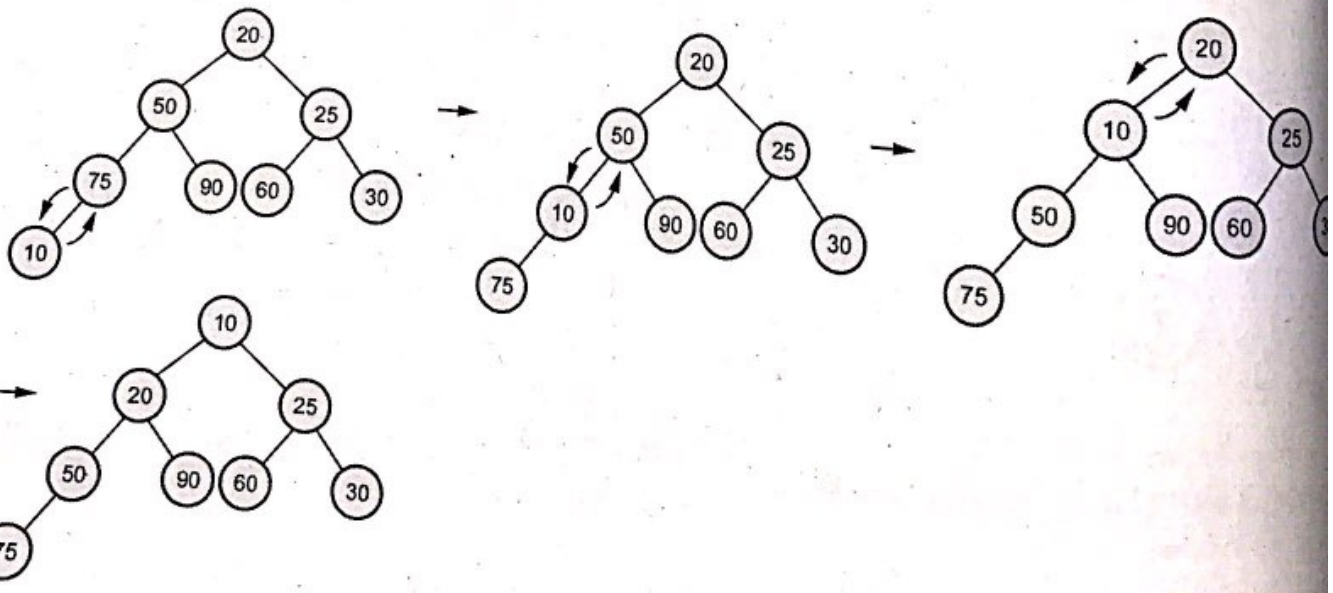
Step : 6



Step : 7



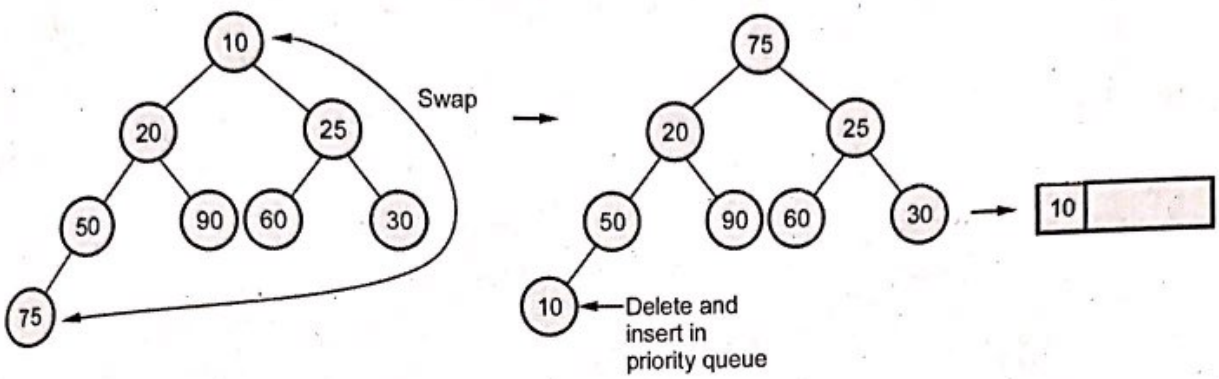
Step : 8



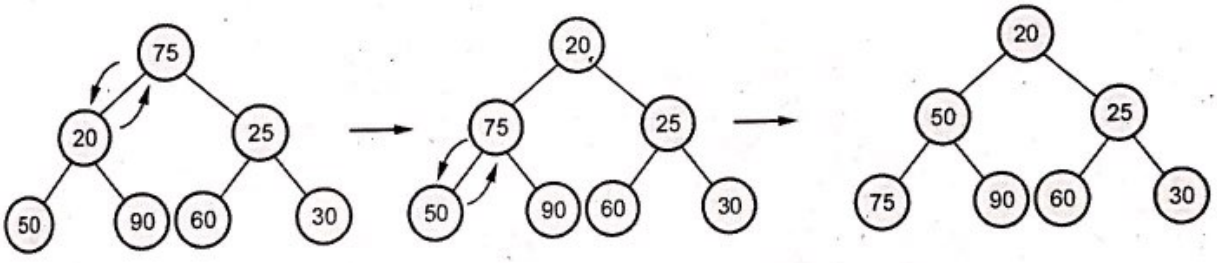
Stage II : Deletion of root

In this stage the root node is deleted and stored in priority queue. The min heap property is always maintained.

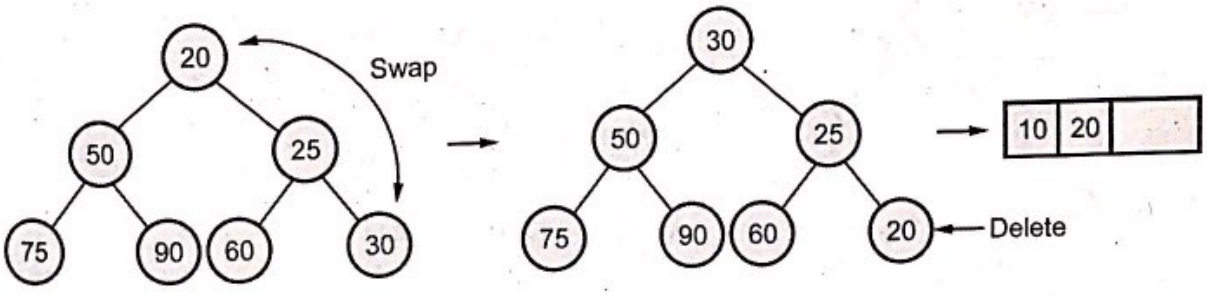
Step : 1 a : Deletion



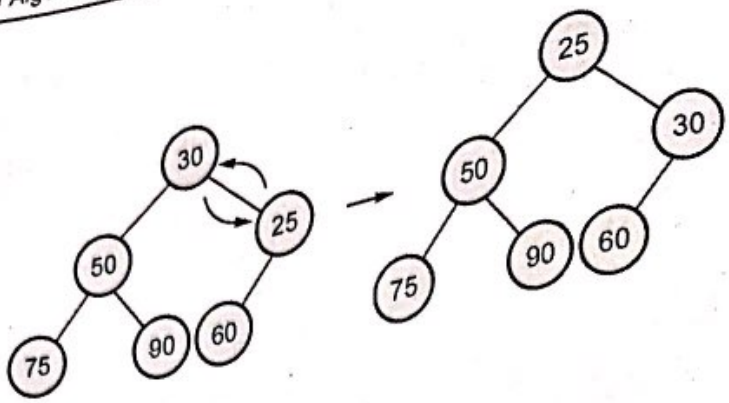
Step : 1 b : Adjusting heap property



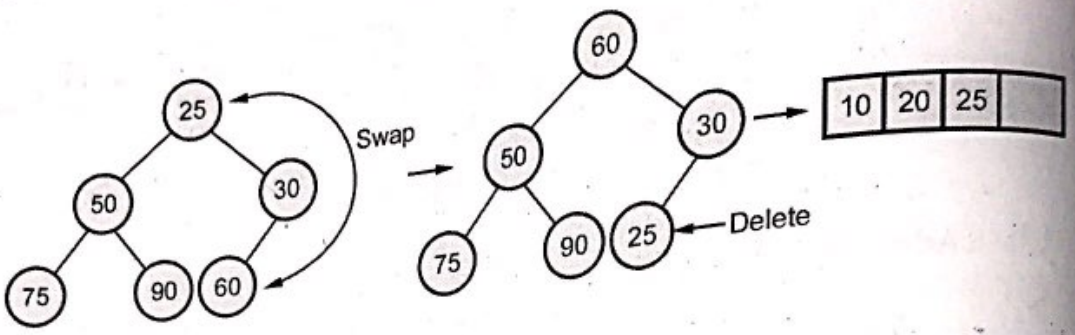
Step : 2 a :



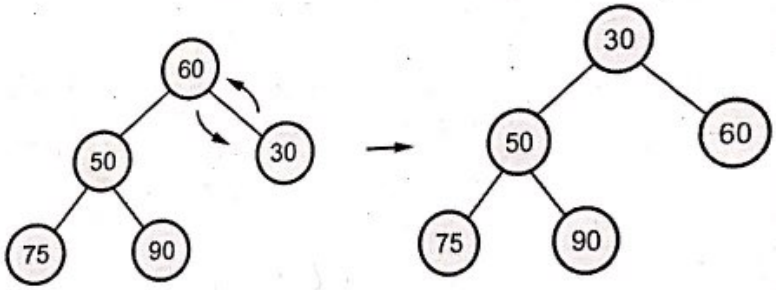
Step : 2 b :



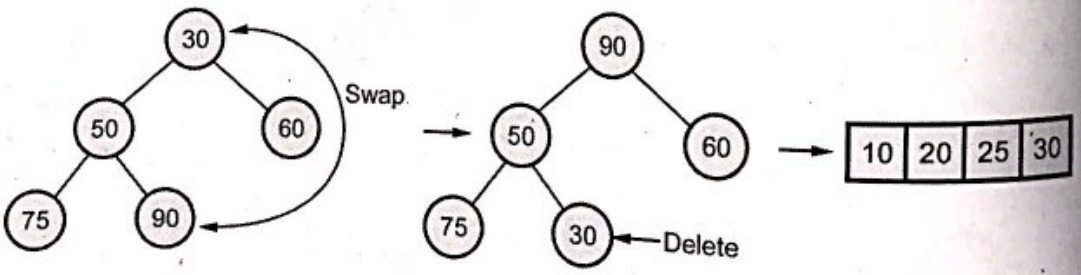
Step : 3 a :



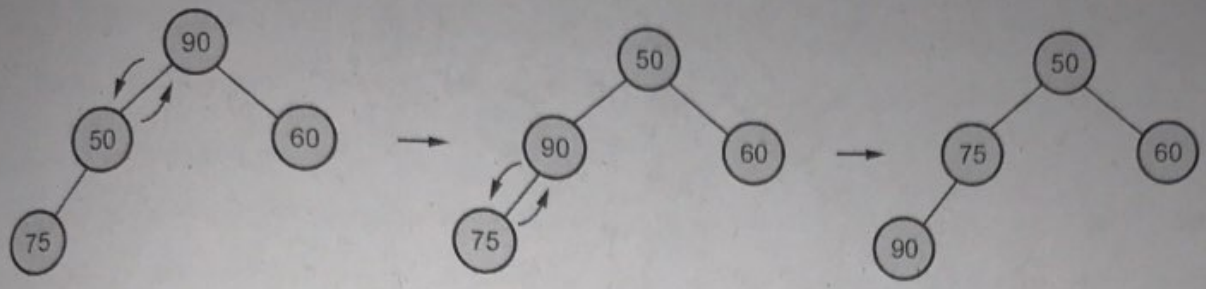
Step : 3 b :



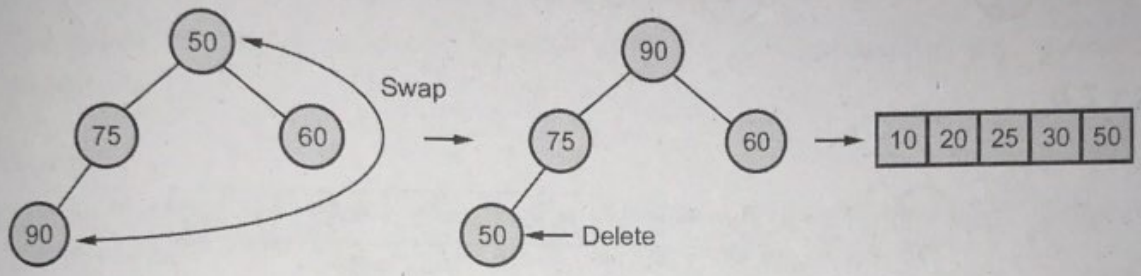
Step : 4 a :



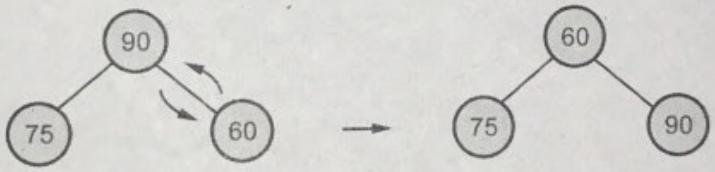
Step : 4 b :



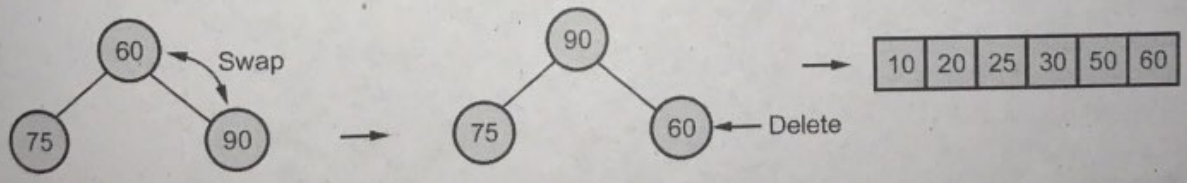
Step : 5 a :



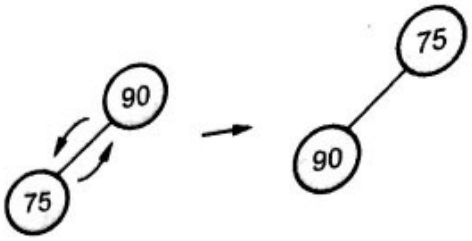
Step : 5 b :



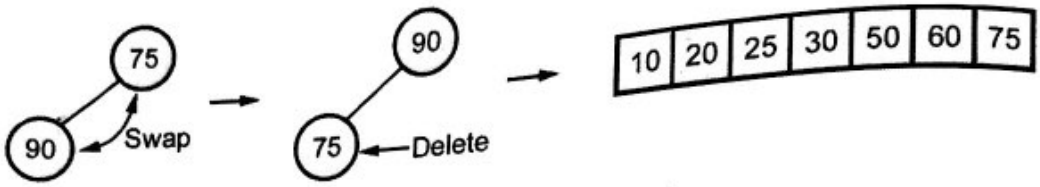
Step : 6 a :



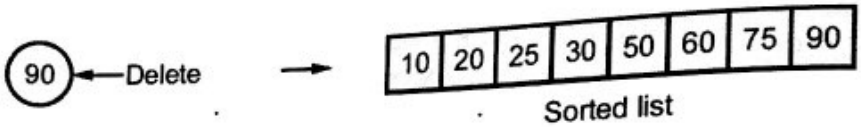
Step : 6 b :



Step : 7 a :



Step : 7 b :



Pseudo code

```
void delet(int item)
{
int item,temp;
if(size==0)
 printf("Heap is empty");
else
{
 //remove the last element and reheapify
 item=H[size--];
//item is placed at root
 temp=1;
 child=2
 while(child<=size)
 {
  if(child<size && H[child]< H[child+1]
   child++;
 if(item>=H[child])
  break;
 H[temp]=H[child]
 temp=child;
 child=child*2;
}
```

```
//place the largest item at root
H[temp]=item;
}
}
```

Analysis : Time complexity of heapsort is **O (log n)** in both worst and average case.

Features of heap sort

1. The time complexity of heap sort is Θ (n log n).
2. This is an in-place sorting algorithm. That means it does not require extra storage space while sorting the elements.
3. For random input it works slower than quick sort.
4. Heap sort is not a stable sorting method.
5. The space complexity of heap sort is O (1). As it does not require any extra storage to sort

'C' Program

```
/*****************************************************************
    This program is for implementing heap sort using heap construction
*****************************************************************/
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define MAX 10
void main()
{
    int i,n;
    int arr[MAX];
    void makeheap(int arr[MAX],int n);
    void heapsort(int arr[MAX],int n);
    void display(int arr[MAX],int n);
    clrscr();
     for(i=0;i<MAX;i++)
          arr[i]=0;
    printf("\n How many elements you want to insert?");
    scanf("%d",&n);
    printf("\n Enter the elements");
     for(i=0;i<n;i++)
          scanf("%d",&arr[i]);
    printf("\n The Elements are ...");
    display(arr,n);
    makeheap(arr,n);
    printf("\n Heapified");
    display(arr,n);
```

2.8.2 Sorting in Linear Time

2.8.2.1 Bucket Sort

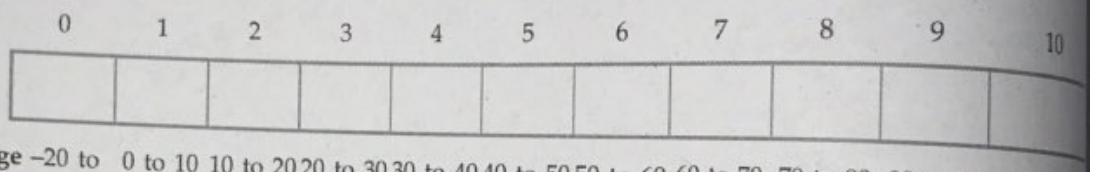
Bucket sort is a sorting technique in which array is partitioned into buckets. Each bucket is then sorted individually, using some other sorting algorithm such as insertion sort.

Algorithm

1. Set up an array of initially empty buckets.
2. Put each element in corresponding bucket.
3. Sort each non empty bucket.
4. Visit the buckets in order and put all the elements into a sequence and print them.

Example 2.8.4 *Sort the elements using bucket sort. 56, 12, 84, 56, 28, 0, -13, 47, 94, 31, 12, -2.* May-08

Solution : We will set up an array as follows



Range	–20 to –1	0 to 10	10 to 20	20 to 30	30 to 40	40 to 50	50 to 60	60 to 70	70 to 80	80 to 90	90 to 100

Now we will fill up each bucket by corresponding elements

0	1	2	3	4	5	6	7	8	9	10
–13 –2	0	12 12	28	31	47	56 56			84	94

Now sort each bucket

0	1	2	3	4	5	6	7	8	9	10
–13 –2	0	12 12	28	31	47	56 56			84	94

Print the array by visiting each bucket sequentially.

-13, -2, 0, 12, 12, 28, 31, 47, 56, 56, 84, 94.

This is the sorted list.

Example 2.8.5 *Sort the following elements in ascending order using bucket sort. Show all passes 121, 235, 55, 973, 327, 179* **Dec.-11**

Solution : We will set up an array as follows -

0 to 100	100 to 200	200 to 300	300 to 400	400 to 500	500 to 600	600 to 700	700 to 800	800 to 900	900 to 1000

Now we will fill up each bucket by corresponding element in the list

55	121, 179	235	237						973
0 to 100	100 to 200	200 to 300	300 to 400	400 to 500	500 to 600	600 to 700	700 to 800	800 to 900	900 to 1000

Now visit each bucket and sort it individually.

Finally read each element of the bucket and place in some array say b[]. The elements from array b are printed to display the sorted list

55	121	179	235	327	973

Pseudo Code

```
void bucketsort( int a[],int n,int max )
{
 int i,j=0;
//inialize each bucket 0 and thus indicates bucket to be empty
 int *buckets=calloc(max+1,sizeof(int));
//place Each element from unsorted array in each corresponding bucket
for(int i=0;i<n;i++)
  buckets[a[i]]++;
//sort each bucket individually
// Sequentially empty each bucket in some array
for(i=0;i<max;i++)
  while(buckets[i]--)
    b[j++]=i;
//display the array b as sorted list of elements
```

Drawbacks

1. For bucket sort the maximum value of the element must be known.
2. We must have to create enough buckets in the memory for every element to place in the array.

Analysis

The best case, worst case and average case time complexity of this algorithm is O(

2.8.2.2 Radix Sort

In this mehod sorting can be done digit by digit and thus all the elements can sorted.

Example for Radix sort

Consider the unsorted array of 8 elements.

45 37 05 09 06 11 18 27

Step 1 : Now sort the element according to the last digit.

Last digit	0	1	2	3	4	5	6	7	8	9
Elements		11				45, 05	06	37,27	18	09

Now sort this number

Last digit	Element
0	
1	11
2	
3	
4	
5	05,45
6	06
7	27,37
8	18
9	09

Step 2 : Now sort the above array with the help of second last digit.

Second last digit	Element
0	05,06,09
1	11,18
2	27
3	37
4	45
5	
6	
7	
8	
9	

Since the list of element is of two digit that is why, we will stop comparing. Now whatever list we have got (shown in above array) is of sorted elements. Thus finally the sorted list by radix sort method will be

05 06 09 11 18 27 37 45

Algorithm :

1. Read the total number of elements in the array.
2. Store the unsorted elements in the array.
3. Now the simple procedure is to sort the elements by digit by digit.
4. Sort the elements according to the last digit then second last digit and so on.
5. Thus the elements should be sorted for up to the most significant bit.
6. Store the sorted element in the array and print them.
7. Stop.

'C' Program

```
/*************************************************************
      Program for sorting the elements by radix sort.
*************************************************************/
/*Header Files*/
```

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
  { int a[100][100],r=0,c=0,i,sz,b[50],temp;
    clrscr();
    printf("Size of array: ");
     scanf("%d",&sz);
    printf("\n");
    for(r=0;r<100;r++)
            { for(c=0;c<100;c++)
              a[r][c]=1000;
    }
    for(i=0;i<sz;i++)
    { printf("Enter element %d: ",i+1);
       scanf("%d",&b[i]);
      r=b[i]/100;
      c=b[i]%100;
      a[r][c]=b[i];
    }
    for(r=0;r<100;r++)
    { for(c=0;c<100;c++)
      { for(i=0;i<sz;i++)
        { if(a[r][c]==b[i])
          { printf("\n\t");
            printf("%d",a[r][c]);
          }
        }
      }
    }
    getch();
  }
/*********************End of Program*****************************/
```

Output

```
Size of array: 5

Enter element 1 : 7
Enter element 2 : 5
Enter element 3 : 37
Enter element 4 : 27
Enter element 5 : 29
        5
        7
        27
        29
        37
```