Foundation of Algorithm Analysis.

Principles of Analyzing algorithms and problem

An algorithm is a finite set of computational instruction, each instruction can be executed in finite time to perform computation or problem solving. by giving input to produce output at finite number of times. An algorithms are not dependent on a particular machine, programming language or compiler so algorithm is a mathematical object where the algorithms are assumed to be run under machine with unlimited capacity.

Properties of an Algorithm

1. Input/Output

   There must be some inputs from the standard sets of input and an algorithms must produce output.

2. Definiteness.

   Each step must be clear and unambigious.

3. Finiteness

   Algorithm must terminate after finite time.

4. Correctness

   Correct set of output values must be produced from each sets of input.

5. Effectiveness

   Each steps must be carried out in finite time.

RAM (Random Access Machine Model)

It is the base model for study of design and analysis of algorithm to have design and analysis in machine independent. In this model, each basic operation (+, −) takes 1 step. Loops and sub-routines are not considered basic operations. Each memory reference is 1 step. We measure run time of algorithm by counting the steps.

Algorithms:-
- Design of algorithm
- Analysis of algorithm
- Validation of algorithm
- Testing of algorithm.

Best, Worst and Average Case.

The least possible execution time taken by an algorithm for a particular input known as best case. Best case complexity gives lower bound on the running time of the algorithm for any instance of input. This indicates that the algorithm can never have lower running time than best case for particular class of problem.

Worst Case

The maximum possible execution time taken by an algorithm or a particular input is known as worst case. It gives upper bound on the running time of the algorithm for all instances of the input.

Average Case:-

It gives average number of steps on any instance of the input.

The quick sort algorithm can be represented as,

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

The best case complexity is $O(n \log n)$. Worst case time complexity when an array is already sorted or sorted in reverse order, one portion contain $(n-1)$ items and another contains zero items, therefore its recurrance relation is,

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

Asymptotic Notation

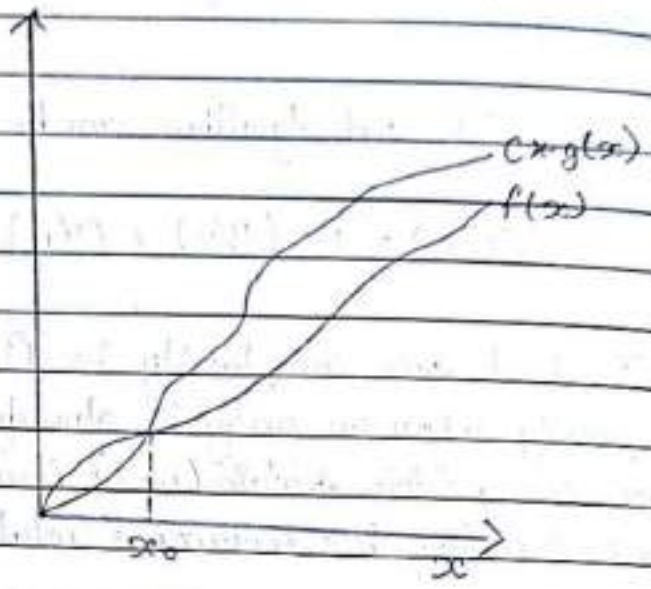
Complexity analysis of an algorithm is very harsh if we try to analyze exact. We Know that the complexity (best, worst or average) of an algorithm or the mathematical function of the size of the input.

So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier. For this purpose we need the concept of asymptotic notation. The figure below gives upper and lower bound concept.

1. Big Oh (O) notation.

When we have only asymptotic upper bound then we use O notation. If f and g are two functions than set of integers then $f(x)$ is said to be big oh of

$g(x)$ i.e $f(x) = O(g(x))$ iff there exist two positive consts 'c' and 'x_0' such that for all $x >= x_0$, $0 <= f(x) <= c * g(x)$



$f(x) = Og(x)$

For e.g:-

$f(n) = 3n^2 + 4n + 7$

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

$f(n) <= 3n^2 + 4n^2 + 7n^2$

$<= 14n^2$

$\therefore f(n) <= c * g(n^2)$

where, $c = 14$ and $n_0 = 1$.

Q. Show that $3x^2 + 2x + 1$ is $O(x^2)$.

=> Soln

Here,

$f(x) = 3x^2 + 2x + 1$

$f(x) \leq 3x^2 + 2x^2 + 1x^2$

$\leq 6x^2$

$\therefore f(x) \leq c * g(x^2)$

where $c = 6$ and $x_0 = 1$

$\therefore f(x) = O(g(x))$

Q. Show that $f(x) = 8x^5 - 17x^2 + 80x - 24$ is $O(x^5)$.

=> Soln

Here,

$f(x) = 8x^5 - 17x^2 + 80x - 24$

$f(x) \leq 8x^5 - 17x^5 + 80x^5 - 24x^5$

$\leq 47x^5$

Now,

$f(x) \leq c \times g(x^5)$ ~~where~~

where, $c = 47$ and $x_0 = 0$

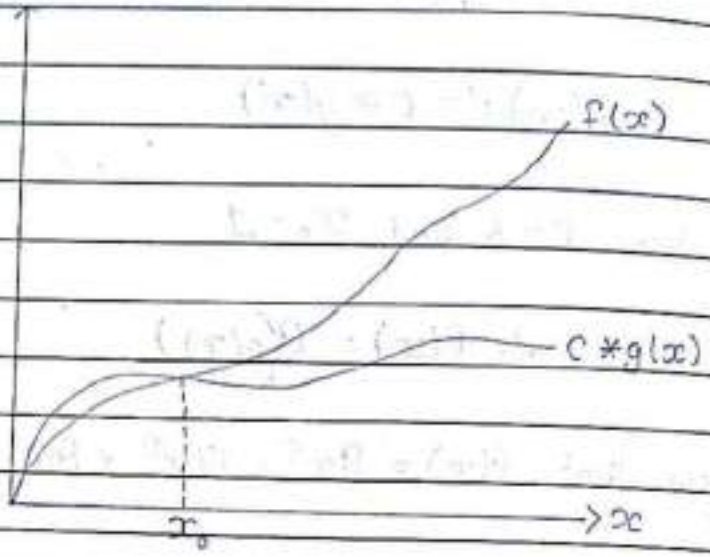
i.e $f(x) \leq c \times g(x)$ where $g(x) = x^5$

$\therefore f(x) = O(g(x))$

2. Big Omega (Ω) notation.

Big omega notation gives asymptotic lower bound. If f and g are two functions from set of integers then $f(x)$ is said to be big omega of $g(x)$ i.e $f(x) = \Omega g(x)$ iff there exist two positive constant 'c' and 'x_0' such that for all $x >= x_0$, $0 \leq c * g(x) \leq f(x)$. The above relation says that

$g(x)$ is lower bond of $f(x)$.



For e.g:-

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$ then prove that $f(n) = \Omega(g(n))$

Proof:-

Let us choose C and n_0 value as 3 and 1 respectively then we have,

$f(n) = C * g(n)$, $n \geq n_0$

or, $3n^2 + 4n + 7 >= 3 * n^2$ for all $n >= 1$.

The above inequality is trivally true. Hence $f(x) = \Omega(g(x))$

3. Big theta (Θ) notation.

When we need asymptotically tight bo then we use this notation. If 'f' and 'g' are any two functio from set of integers to set of integers then function f(x) is said to be big theta g(x) i.e $f(x) = \Theta(g(x))$ iff there exists three positive constants C_1, C_2 and x_0 such that for all

$x >= x_0$, $C_1 * g(x_0) <= f(x) <= C_2 * g(x_0)$

For e.g:-

Show that $3x^2 + 8x + 7$ is $O(x^2)$.

= Soln

Here,

$f(x) = 3x^2 + 8x + 7$

$g(x) = x^2$

When $x \geq 1$,

$3x^2 + 8x + 7 \leq 3x^2 + 8x^2 + 7x^2$

$<= 18x^2$

$\Rightarrow f(x) <= C \cdot x^2$

$\therefore$

$\Rightarrow C_2 = 18$

$\therefore f(x) <= O(x^2)$ ------①

Again,

$3x^2 + 8x + 7 >= 3 \cdot x^2$

$f(x) >= C \cdot x^2$

$\Rightarrow C_1 = 3$

$\therefore f(x) >= \Omega(x^2)$ ------②

From eqn ① and ②

$3x^2 <= 3x^2 + 8x + 7 <= 18x^2$

$\therefore f(x) = \Theta(x^2)$

$\therefore f(x) = \Theta(g(x))$

Mathematical Foundation

1. Exponents

Some of the formulas are,

- $\frac{x^a}{x^b} = x^{a-b}$
- $x^a \times x^b = x^{a+b}$
- $(x^a)^b = x^{ab}$
- $x^n + x^n = 2x^n$
- $2^n + 2^n = 2^{n+1}$

2. Logarithmic

Some formulas are,

- $\log_a b = \frac{\log_c b}{\log_c a}$
- $\log ab = \log a + \log b$
- $\log a/b = \log a - \log b$
- $\log 1 = 0$
- $\log 2 = 1$
- $a \log b^n = na \log b$

3. Series

Some formulas are,

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{n=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=0}^{n} a^i = \frac{1}{1-a}$, if $0 < a < 1$

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

Recurrence Relation

It is an equation or an inequality that is defined in terms of itself. Recurrences are useful in computer science because it makes the expression of an algorithm easy and can be solve easily. To analyze the recursive algorithm we must need to find their recurrence relation.

Recursive algorithm for finding factorial.

$T(n) = 1$ when $n = 1$

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

Recursive algorithm for finding n^{th} Fibonacci numbers.

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

$T(1) = 1$ when $n = 1$

$T(2) = 1$ when $n = 2$

Technique for solving recurrence.

There are four ways to solve recurrence. They are :-

1. Iteration Method.
2. Recursion Method.
3. Substitution Method.
4. Master Method.

1. Iterative method.

In this method, we expand the given relation until the boundary is not meet. Expand the relation so that summation independent on is obtained. For example:-

Q. Solve the following.

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

$T(n) = 1$ when $n = 1$

⇒ Solⁿ

Here,

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

$= 2\{2T(n/4) + 1\} + 1$

$= 2^2 T(n/2^2) + 2 + 2^0$

$= 2^2\{2T(n/8) + 1\} + 2^1 + 2^0$

$= 2^3 T(n/2^3) + 2^2 + 2^1 + 2^0$

- - - - - - - - - -

- - - - - - - - - -

$= 2^K T(n/2^K) + 2^{K-1} + \dots + 2^0$

Note:-

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

$T(n/2) = 2 \cdot T(n/4) + 1$

$T(n/4) = 2 \cdot T(n/8) + 1$

For simplicity assume,

$\frac{n}{2^K} = 1$

$\Rightarrow n = 2^K$

Taking log on both sides

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

$\Rightarrow K = \log n \quad (\because \log 2 = 1)$

Now,

$$T(n) = 2^K \cdot T(1) + 2^{K-1} + \dots + 2^0$$
$$= 2^K \cdot 1 + 2^{K-1} + \dots + 2^0$$
$$= 2^{K+1} - 1$$
$$= 2 \cdot 2^K - 1$$
$$= 2n - 1$$

$$\therefore T(n) = O(n)$$

8 Solve the following by iterative method.

$$T(n) = T(n-1) + O(1)$$
$$T(n) = 1 \quad \text{when } n = 1$$

⇒ Sol^n

Here,

$$T(n) = T(n-1) + 1$$
$$= T(n-2) + 1 + 1$$
$$= T(n-3) + 1 + 1 + 1$$
$$= T(n-K) + K * 1.$$

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

For simplicity assume,

$$n - K = 1$$
$$\Rightarrow K = n - 1$$

Now,

$$T(n) = T(1) + 1 \times K$$
$$= 1 + K$$
$$= 1 + n - 1 = n$$
$$\therefore T(n) = O(n)$$

Q. Solve the following by iterative method.

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

$T(n) = 1$ when $n = 1$

⇒ Solⁿ

Here,

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

$= 2\{2T(n/4) + n/2\} + n$

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

$= 2^3\{2T(n/8) + n/4\} + n + n$

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

- - - - - - - - - - - - -

- - - - - - - - - - - - -

$= 2^K T(n/2^K) + K \times n$

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

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

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

For simplicity assume,

$\frac{n}{2^K} = 1$

$\Rightarrow n = 2^K$

Taking log on both sides,

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

$\Rightarrow K = \log n$

Now,

$T(n) = n \cdot T(1) + n \log n$

$= n \cdot 1 + n \log n$

$= n + n \log n$

$\therefore T(n) = O(n \log n)$

Q. Solve the following by iterative method.

$$T(n) = T(n/3) + O(n) \quad \text{when } n > 1$$
$$T(n) = 1 \quad \text{when } n = 1$$

=> Soln

Here,

$$T(n) = T(n/3) + cn$$
$$= T(n/9) + cn/3 + cn$$
$$= T(n/27) + cn/9 + cn/3 + cn$$
$$= \cdots$$
$$= T\left(\frac{n}{3^k}\right) + \frac{cn}{3^{k-1}} + \cdots + \frac{cn}{3^0}$$

For simplicity

$$\frac{n}{3^k} = 1$$
$$\Rightarrow n = 3^k$$

Taking log on both sides

$$\log(n) = \log(3^k)$$
$$\Rightarrow k = \log_3 n$$

Now,

$$T(n) = T(1) + \frac{cn}{3^{k-1}} + \cdots + \frac{cn}{3^0}$$
$$= 1 + cn\left(\frac{1}{3^{k-1}} + \cdots + \frac{1}{3^0}\right)$$

$$= 1 + cn \times \frac{1}{1 - 1/3}$$

$$= 1 + \frac{3cn}{2}$$

$$\therefore T(n) = O(n)$$

Recurrence Tree Method.

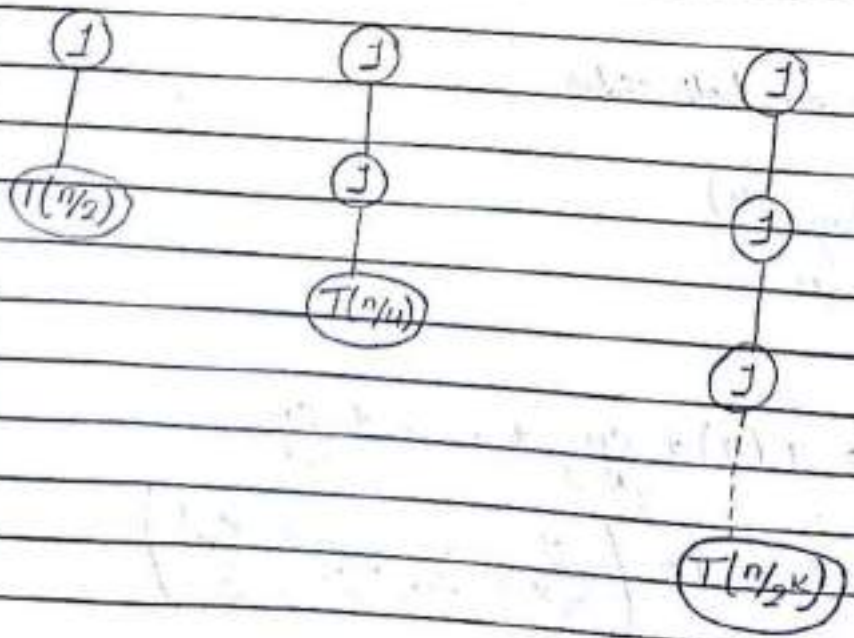
Recursion tree method is a pictorial representation of an iteration method which is of a tree where at each level nodes are expanded. In general, we consider recurrance as root. It is useful when the divide and conquer algorithm is used.

Q. Solve the following recurrence relation by using recursion tree method.

$T(1) = 1$ when $n = 1$

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

⇒ Sol:



Cost at each level = 1

For simplicity assume that,

$$\frac{n}{2^K} = 1$$

$$\Rightarrow n = 2^K$$

Taking log on both sides

$$\Rightarrow K = \log n$$

Summing the cost at each level;

Total cost, $T(n) = 1 + 1 + 1 + \dots + 1$ (K-times) $+ T(n/2^K)$

$$= K * 1 + T(1)$$

$$= K + 1$$

$$= \log n + 1$$

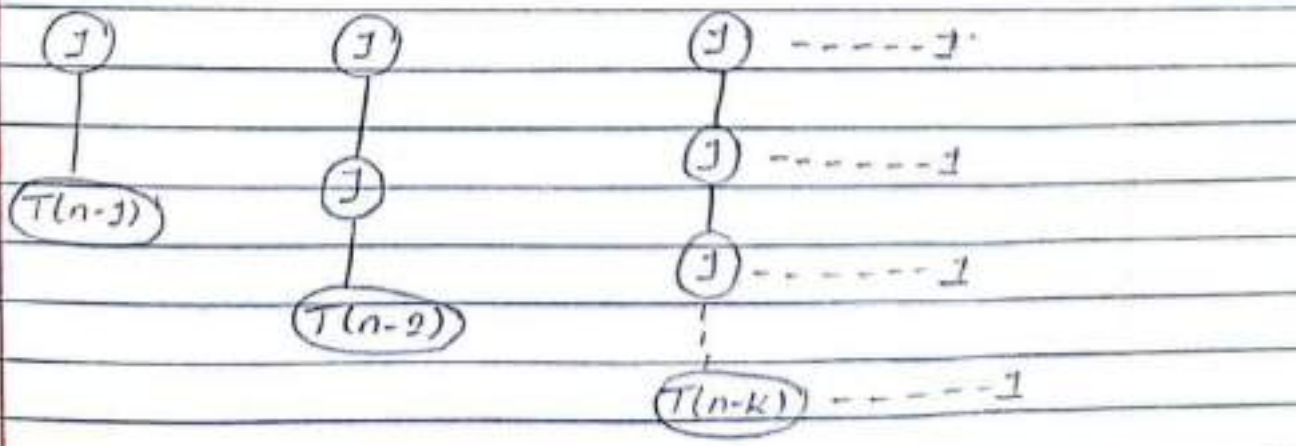
$$\Rightarrow T(n) = O(\log n)$$

Q. Solve following recurrence relation by using recursion tree method.

$T(1) = 1$ when $n = 1$

$T(n) = T(n-1) + 1$ when $n \geq 1$

$\Rightarrow$ Sol^n



Let put,

$n-K=1$

$\Rightarrow K = n+1$

Total cost, $T(n) = 1+1+\cdots+1$ (K-times) $+ T(n-K)$

$= K*1 + T(1)$

$= K+1$

$= n-1+1$

$= n+0$

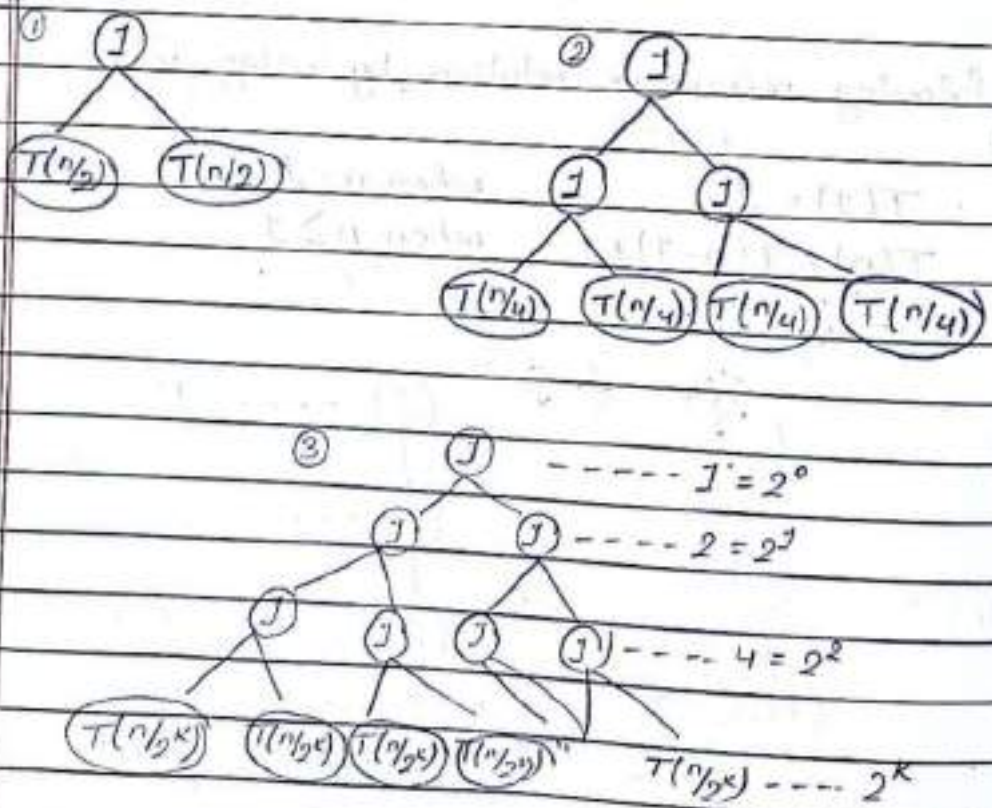
$\Rightarrow T(n) = O(n)$

Q. Solve the following recurrence relation by using recursion tree method.

$T(1) = 1$ when $n=1$

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

$\Rightarrow$ Sol^{n}



For simplicity assume that,

$$\frac{n}{2^K} = 1$$

$$\Rightarrow n = 2^K$$

Taking log on both sides,

$$\Rightarrow K = \log n$$

Now,

$$T(n) = 2^0 + 2^1 + 2^2 + \dots + 2^K$$

$$= 2^{K+1} - 1$$

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

$$= 2n - 1$$

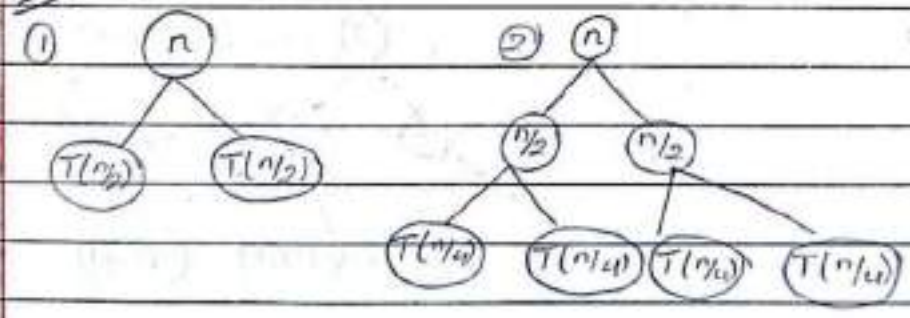
$$\Rightarrow T(n) = O(n)$$

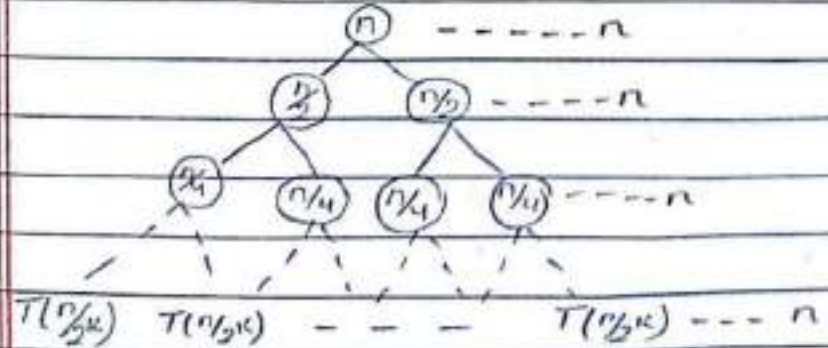
Q. Solve the following recurrence relation by using recursion tree method.

$$T(1) = 1$$

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

$\Rightarrow$ Solr





For simplicity assume that,

$$\frac{n}{2^K} = 1$$

$$\Rightarrow n = 2^K$$

Taking log on both sides

$$\Rightarrow K = \log n$$

Now,

$$T(n) = n + n + n + \dots + n(K \text{ times}) + T(n/2^K)$$

$$= n * K + T(1)$$

$$= n \log n + 1$$

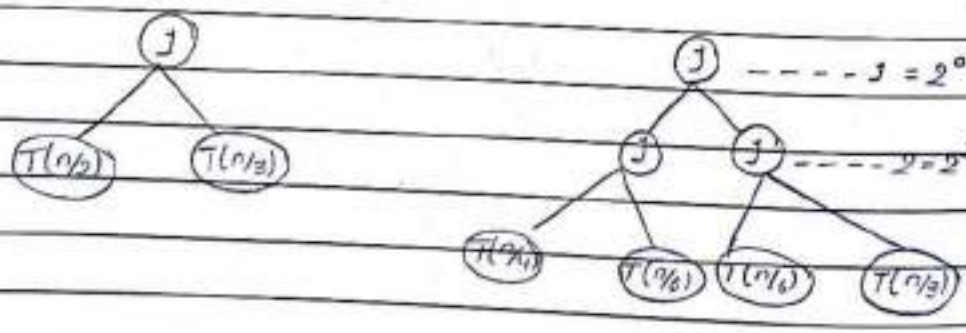
$$\Rightarrow T(n) = O(n \log n)$$

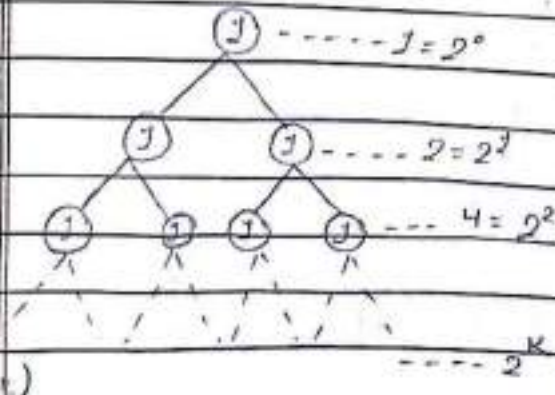
Q. Solve the following recurrence relation.

$$T(n) = T(n/2) + T(n/3) + O(1) \quad \text{when } n > 1$$

$$T(1) = 1 \quad \text{when } n = 1$$

$\Rightarrow$ Self





For simplicity,

$$\frac{n}{2^k} = 1$$

$$\Rightarrow n = 2^k$$

Now,

$$T(n) = 2^0 + 2^1 + 2^2 + \dots + 2^k + T(n/2^k)$$
$$= 2^{k+1} - 1 + T(1)$$
$$= 2^{k+1} - 1 + 1$$
$$= 2^k \cdot 2$$
$$= 2n$$
$$\Rightarrow T(n) = O(n)$$

Substitution Method.

In this method, to obtain the upperbound of the reference relation we must use following two steps:-

1. At first guess the solution.
2. Then verify the solution by using mathematical induction.

Q. Solve the following recurrance relation.

$$T(n) = 1 \quad \text{when } n = 1$$
$$T(n) = 4T(n/2) + n \quad \text{when } n > 1.$$

$\Rightarrow$ Solr

Guess, $T(n) = O(n^3)$

$$T(n) \leq cn^3 \text{ for all } n \geq n_0 \quad \text{-----} (1)$$

Now, prove this by using mathematical induction.

Basis Step:

For $n = 1$.

$T(n) = c * 1^3$

$T(1) \leq c * 1^3$

$1 \leq c \times 1^3$

which is true for $n = 1$.

Inductive step:

Let we assume that it is true for $K = n$. Then,

$T(K) \leq c \cdot K^3$ ---- (2)

It is also true for $K = n/2$

Now eqn(2) becomes,

$T(n/2) \leq c \cdot (n/2)^3$

Now,

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

$= 4 \times \frac{cn^3}{8} + n$

$= \frac{cn^3}{2} + n$

$= cn^3 - \frac{cn^3}{2} + n$

$= cn^3 - n\left(\frac{cn^2}{2} - 1\right)$

Therefore, $T(n) \leq cn^3$

Thus,

$$T(n) = O(n^3)$$

Q. Show that $O(n^2)$ is a solution of recurrance.

$$T(n) = 4T(n/2) + n \quad \text{when } n > 1$$
$$T(n) = 1 \quad \text{when } n = 1.$$

⇒ Solⁿ

Assume,

$$T(n) = O(n^2)$$
$$T(n) \leq cn^2 \text{ for all } n \geq n_0$$

Now, prove this by using mathematical induction.

Basic Step:-

For $n = 1$

$$T(n) \leq cn^2$$
$$T(1) \leq c \times 1^2$$
$$\text{or, } 1 \leq c \times 1^2 \quad \text{------(1)}$$

which is true for $n = 1$

Inductive Step:-

Let we assume that it is true for $K = n$.

Then,

$$T(K) \leq c.K^2 \quad \text{------(2)}$$

It is also true for $K = n/2$

Now, eqⁿ (2) becomes,

$T(n/2) \leq c \cdot (n/2)^2$

Now,

$$T(n) = 4T(n/2) + n$$
$$= 4 \times \frac{cn^2}{4} + n$$
$$\leq cn^2 + n$$

It is not possible to show that $cn^2 + n \leq cn^2$ thus we try to subtract lower order term as,

$$T(n) \leq cn^2 - dn \leq cn^2 \text{ ------ } (3)$$

where c and d are positive constant.

Now, prove this relation by using mathematical induction.

Basic Step :-

For $n = 1$

$$T(n) = c \times 1^2 - d \times 1$$
$$= c - d$$

which is true for positive value of c and $d \leq cn^2$

Inductive Step :-

Let us assume that it is true for $K = n$. Then,

$$T(n) \leq cK^2 - dK \text{ ----- } (4)$$

It is also true for $K = n/2$

$T(n/2) \leq c(n/2)^2 - d(n/2)$

$= \frac{cn^2}{4} - \frac{dn}{2}$

Now,

$T(n) = 4 \cdot T(n/2) + n$

$= 4\left[\frac{cn^2}{4} - \frac{dn}{2}\right] + n$

$= cn^2 - 2dn + n$

$= cn^2 - dn - dn + n$

$= (cn^2 - dn) - n(d-1) \leq cn^2 - dn$

$\leq cn^2 - dn$

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

Q. Show that $O(n^3)$ is its solution.

$T(n) = 8T(n/2) + n^2$

⇒ Sol^n

Assume,

$T(n) = O(n^3)$

$T(n) \leq cn^3$ for all $n \geq n_0$ ------①

Basic Step:

For $n = 1$,

$T(n) \leq c \times 1^3$

$1 \leq c$ which is true for all +ve values of c

Inductive step:

Lets assume that it is true for $K < n$

$$T(K) \leq cK^3 \text{ ------- } (2)$$

Also true for $K = n/2$

Now, equation (2) becomes,

$$T(n/2) \leq c(n/2)^3$$
$$= \frac{cn^3}{8}$$

Now,

$$T(n) = 8T(n/2) + n^2$$
$$\leq 8\frac{cn^3}{8} + n^2$$
$$= cn^3 + n^2$$

$$\Rightarrow T(n) = cn^3 + n^2$$

It is not possible to show that $cn^3 + n^2 \leq cn^3 \ \forall \ n > 0$. thus we try to subtract lower order term as,

$$T(n) \leq cn^3 - dn^2 \text{ ------ } (3)$$

where c and d are +ve constants.

Now proof this relation by mathematical induction.

Basic Steps:-

For $n = 1$,

$T(n) = c*1^3 - d*1^2$

$1 \leq c-d$

which is true for all +ve values of c and $d < c$

Inductive Step

Lets assume that it is true for $K < n$.

Then,

$T(K) \leq cK^3 - dK^2$ ------ (4)

It is also true for $K = n/2$

Now eqn (4) becomes,

$T(n/2) \leq c(n/2)^3 - d(n/2)^2$

$= \frac{cn^3}{8} - \frac{dn^2}{4}$

Now,

$T(n) = 8T(n/2) + n^2$

$\leq 8\left[\frac{cn^3}{8} - \frac{dn^2}{4}\right] + n^2$

$\leq cn^3 - 2dn^2 + n^2$

$\leq cn^3 - dn^2 - dn^2 + n^2$

$\leq (cn^3 - dn^2) - n^2(d-1) \leq cn^3 - dn^2$

$\Rightarrow T(n) \leq (cn^3 - dn^2) \; \forall n > 0$

Thus $T(n) = O(n^3)$

Master Method

Master method is used to solve the recurrance relation of the form,

$$T(n) = aT(n/b) + f(n)$$

where, $a \geq 1$, $b \geq 1$ are constants, $f(n)$ asymptotically positive function. If the recurrance relation is in this form there. There are 4 possible cases ocurred.

Case I:

If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. Then,

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

Case II:

If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constants $\epsilon > 0$. Then,

$$T(n) = \theta(f(n))$$

Case III:

If $f(n) = O(n^{\log_b a})$ for some constants $\epsilon > 0$. Then,

$$T(n) = \theta(f(n) . \log n)$$

In the above three cases we are comparing the values of $f(n)$ and $n^{\log_b a}$ and then find complexity of the given recurrence relation.

Q Find the complexity of following recurrance relation.

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

=> Solⁿ

Here, we have,

$$a = 3,\ b = 2 \text{ and } f(n) = n$$

Now,

$$n^{\log_b a} = n^{\log_2 3} = n^{1.584}$$

Since, $f(n) \leq n^{\log_b a - \epsilon}$ where choose $\epsilon = 0.1$

Thus it satisfy the first case of Masters method.

Thus its complexity,

$$T(n) = O(n^{\log_b a})$$
$$= O(n^{1.584})$$

Q. Solve the following recurrance relation by using Masters method.

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

=> Solⁿ

Here,

$$a = 4,\ b = 2 \text{ and } f(n) = n^2$$

Now,

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

Here,

$f(n) = O(n^{\log_b a})$. It satisfies third case

$$T(n) = \theta(f(n) \cdot \log n)$$
$$= \theta(n^2 \log n)$$

Q. Solve the following recurrance relation by using Master's method.

$$T(n) = 3T(n/4) + n \log n$$

=> Solⁿ

Here we have,

$$a = 3, \quad b = 4 \quad \text{and} \quad f(n) = n \log n$$

Now,

$$n^{\log_b a} = n^{\log_4 3} = n^{0.658}$$

Since, $f(n) \geq n^{\log_b a + \epsilon}$ where choose $\epsilon = 0.1$

Thus, it satisfy the second case of Master's method.

Thus its complexity,

$$T(n) = O(f(n))$$
$$\Rightarrow T(n) = O(n \log n)$$

Q. Solve the following recurrance relation by using Master's Method.

$$T(n) = 4T(n/2) + n^2/\log n$$

=> Solⁿ

Comparing this relation with the general form.

$$T(n) = aT(n/b) + f(n)$$

where,

$$a = 4, \quad b = 2 \quad \text{and} \quad f(n) = n^2/\log n$$

Test Case I :-

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

$$f(n) \leq n^{\log_b a - \epsilon}$$

$$\text{or, } \frac{n^2}{\log n} \leq n^2 - \epsilon \; n^{2-\epsilon}$$

$$\text{or, } \frac{n^2}{\log n} \leq \frac{n^2}{n^\epsilon}$$

$$\text{or, } \frac{n^2}{\log n} \leq \frac{n^2}{n^{0.1}} \quad \text{where } \epsilon = 0.1$$

$$\text{or, } n^2 n^{0.1} \leq n^2 \log n$$

To satisfy this relation, the value of $\log n$ must be greater than $n^{0.1}$. But $n^{0.1}$ is polynomial in 0.1 than $n^{0.1}$ must be greater than $\log n$

$$\text{i.e } n^{0.1} > \log n$$

Master method is failed in this case.

Test Case II:

$$f(n) = \Omega(n^{\log_b a + \epsilon})$$

$$\Rightarrow f(n) \geq n^{\log_b a + \epsilon}$$

$$\text{or, } \frac{n^2}{\log n} \geq n^{2+\epsilon}$$

$$\text{or, } \frac{n^2}{\log n} \geq n^{2+0.1}$$

$$\Rightarrow n^2 \geq (n^{2+0.1}) \log n$$

which is false, so master method failed in this case.

Test Case III:

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

$$\Rightarrow f(n) = n^{\log_b a}$$

$$\text{or, } \frac{n^2}{\log n} \geq n^2$$

$$\Rightarrow n^2 \geq n^2 \log n$$

which is also false so master method cannot apply in the relati

Sorting

Bubble Sort

It is the method for sorting an array element by finding the largest element from the array of element. The algorithm for finding largest element using bubble sort as,

```
Bubblesort(A,n)
{
  for(i=0; i<n; i++)
    for(j=i+1; j<n; j++)
    {
      if(A[i]<A[j])
      {
        temp = A[i];
        A[i] = A[j];
        A[j] = temp;
      }
    }
}
```

For e.g:-

Trace the following data using bubble sort.

10 2 5 20 30 0 4

Iteration 1:- 30 20 5 10 20 0 4

Iteration 2:- 30 20 10 2 5 10 0 4

Iteration 3:- 30 20 10 2 5 0 4

Iteration 4:- 30 20 10 5 2 0 4

Iteration 5:- 30 20 10 5 4 0 2

Iteration 6:- 30 20 10 5 4 2 0

Time Complexity

The total time complexity is,

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

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

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

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

Best, average and worst case complexity = $O(n^2)$.

Insertion Sort

An insertion sort is one that sorts a set of records by inserting records into an existing sorted file. Suppose that a[] is a list of 'n' elements. The insertion sort technique scans the list a[] from a[0] to a[n], inserting each element a[i] into it's proper position in the previously sorted list a[0], a[1], ---, a[i-1].

Pseudocode:-

```
Insertion (A, n)
{
```

```
for i = 1 to n
{
    temp = A[i]
    j = i - 1
    while( j >= 0 && A[j] > temp)
    {
        A[j+1] = A[j]
        j = j - 1
    }
}
A[j+1] = temp
}
```

For eg:-

7 3 4 1 8 2 6 5

⇒ Solution:-

Iteration 0:-

7	3	4	1	8	2	6	5
0	1	2	3	4	5	6	7

Pick 7 place it with array position 0.

7							
0	1	2	3	4	5	6	7

Iteration 1:-

Pick 3 place it with array position 0.

3	7						
0	1	2	3	4	5	6	7

Iteration 2:-

Pick 4 place it with array position 1.

3	4	7					
0	1	2	3	4	5	6	7

Iteration 3:-

Pick 1 place it with array position 0

1	3	4	7				
0	1	2	3	4	5	6	7

Iteration 4:-

Pick 8 place it with array position 4

1	3	4	7	8			
0	1	2	3	4	5	6	7

Iteration 5:-

Pick 2 place it with array position 1.

1	2	3	4	7	8		
0	1	2	3	4	5	6	7

Iteration 6:-

Pick 6 place it with array position 4.

1	2	3	4	6	7	8	
0	1	2	3	4	5	6	7

Iteration 7:-

Pick 5 place it with array position 4.

1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7

In insertion sort, the total number of comparison is,

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

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

$$= O(n^2)$$

For best case, if the file is initially sorted only one comparison is made on each pass so that the sort is $O(n)$.

Selection Sort

In selection sort all the elements are examined to obtain smallest (greatest) element of one pass then it is placed into its position. this process continues until the whole array is sorted.

Pseudocode

```
Selection sort( )
{
    for( i = 0; i < n; i++ )
    {
        for( j = i + 1; j < n; j++ )
            if [A[i] > A[j]
            swap ( A[i], A[j] )
    }
}
```

For. e.g:-

Sort the given data 38, 47, 24, 42, 17

=> Sol^n

Pass 1:- | 38 | 47 | 24 | 42 | 17 |
0 1 2 3 4

Exchange 38 with 17 as 17 is the smallest element.

Pass 2:- | 17 | 47 | 24 | 42 | 38 |
0 1 2 3 4

Exchange 47 with 24 as 24 is the smallest element.

Pass 3:- | 17 | 24 | 47 | 42 | 38 |
0 1 2 3 4

Exchange 47 with 38 as 38 is the smallest elem

Pass 4:- | 17 | 24 | 38 | 42 | 47 |
0 1 2 3 4

Complexity:-

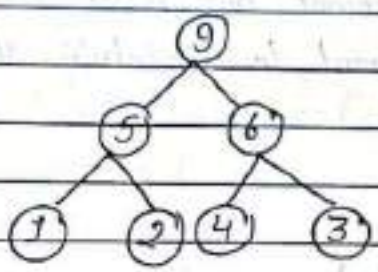
The first pass makes (n-1) comparison. The sec pass makes (n-2) and so on.

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

$$= O(n^2)$$

Heap Sort

The implementation of Heap in sorting is called heap sort. Heap are of two types i.e Max heap and min heap.

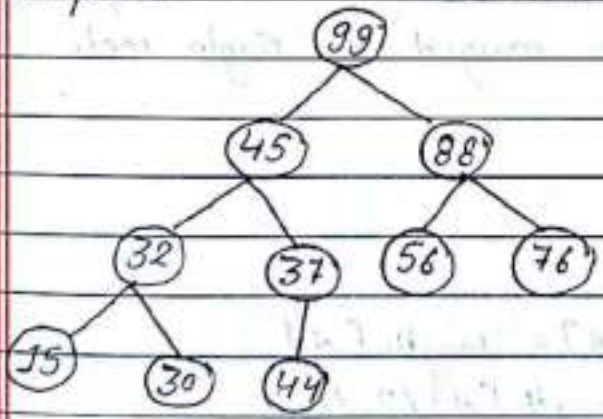


To represent above max heap as an array,

9	5	6	1	2	4	3
0	1	2	3	4	5	6

Remember if parent P is not A[i] place then His left child is at A[2i] and the right child is at A[2i+1].

Is the sequence 99, 45, 88, 32, 37, 56, 76, 15, 30, 44 a max heap?



=> It is not a max heap.

Constructing Heaps (Max)

To create a heap, insert an element into leftmost empty slot of an array. If the inserted element is larger than its parent then swap the element and recur. At every step, we swap the element with parent to maintain the heap order.

Pseudocode:-

```
Parent (i)
    return (i/2)
left (i)
    return (2i)
right (i)
    return (2i+1)
```

Building a Heap

A bottom of procedure for constructing heap is a good way but there is a process that uses merge methods to create a heap called heapify. Here we have given two heaps and new element these two heaps are merged for single root.

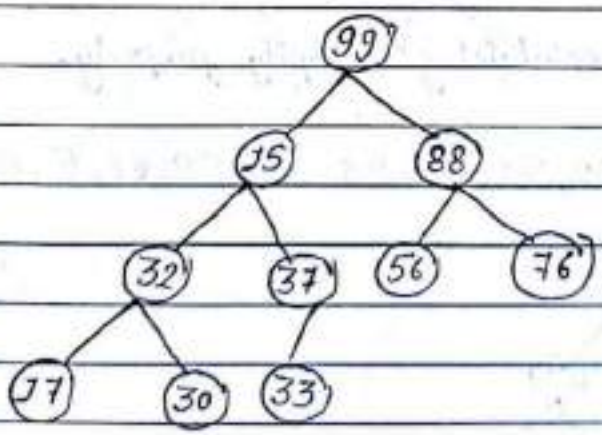
Pseudocode:-

```
Buildheap(A)
{
    heapsize[A] = length[A]
    for i = length[A]/2 to 1
        do heapify(A, i)
}
```

Heapify algorithm.

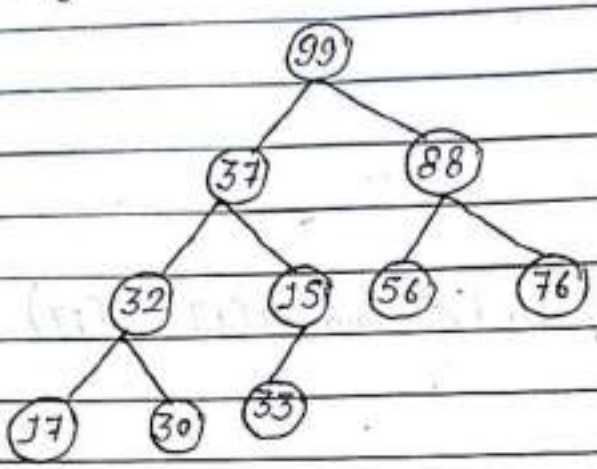
```
Heapify (A, i)
{
    l = left(i)
    r = right(i)
    if (l <= heapsize[A] and A[l] > A[i])
        max = l
    else
        max = i
    if (r <= heapsize[A] and A[r] > A[max])
        max = r
    if max != i
    {
        Swap(A[i], A[max])
        Heapify(A, max)
    }
}
```

For e.g:- Heapify for heapsize[A] = 10

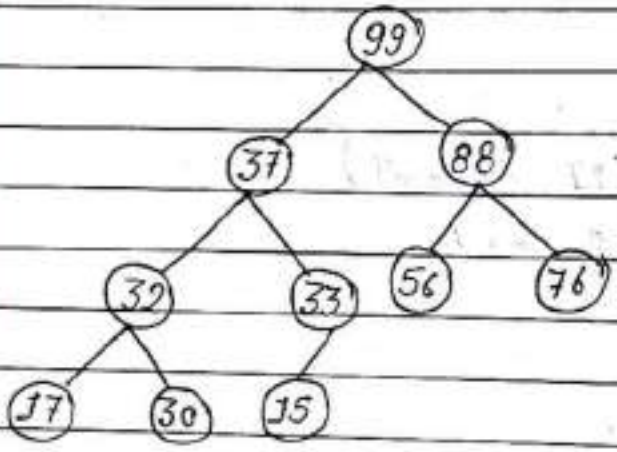


=> Solution:-

Heapify (A, 2)



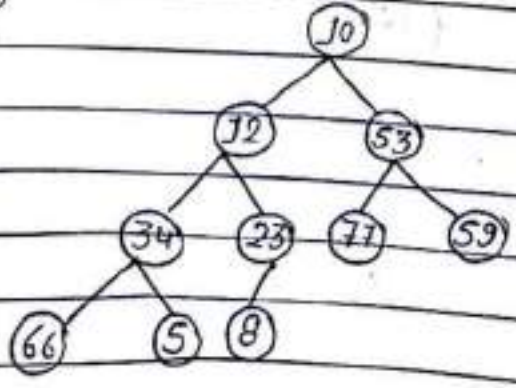
Heapify (A, 5)



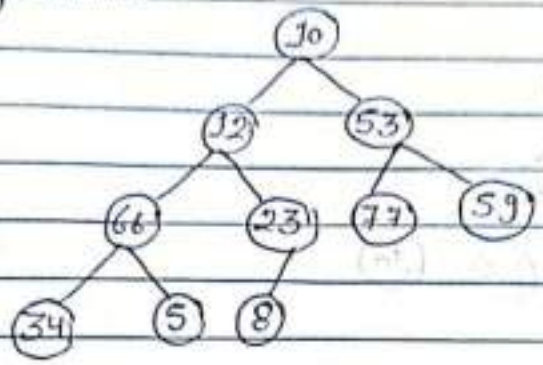
Q. Create a heap with maintaining heapify properly.

A[] = { 10, 12, 53, 34, 23, 77, 59, 66, 5, 8 }

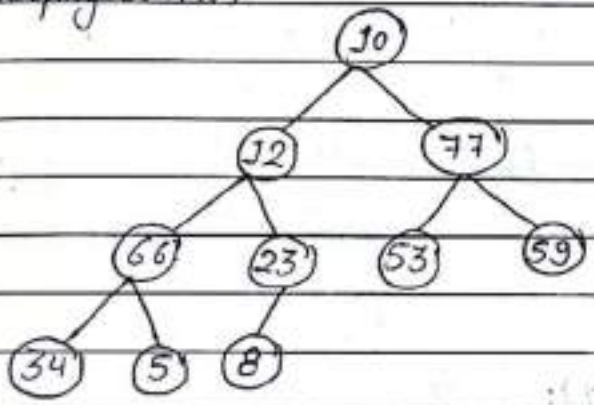
⇒ Sol^n



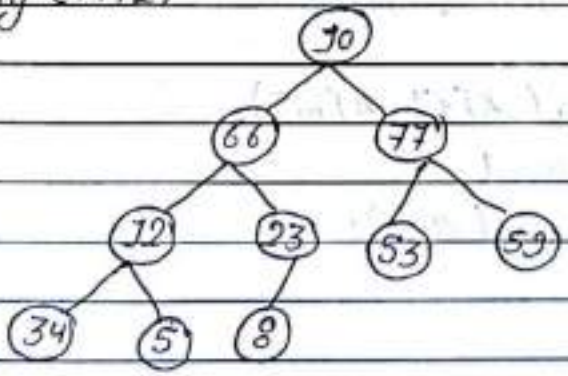
Heapify (A, 4)



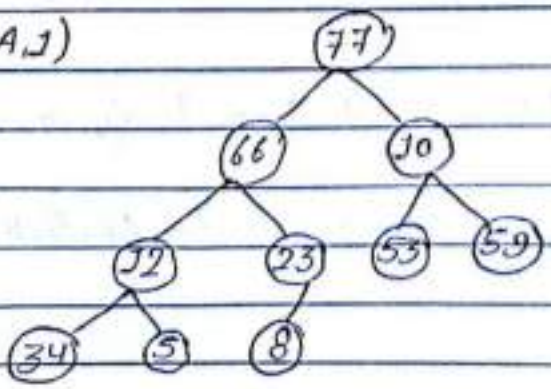
Heapify (A, 3)



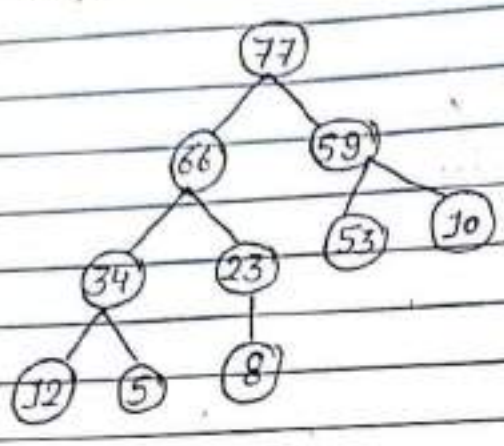
Heapify (A, 2)



Heapify (A, 1)



Final Heap



Heap Sort

Pseudocode :-

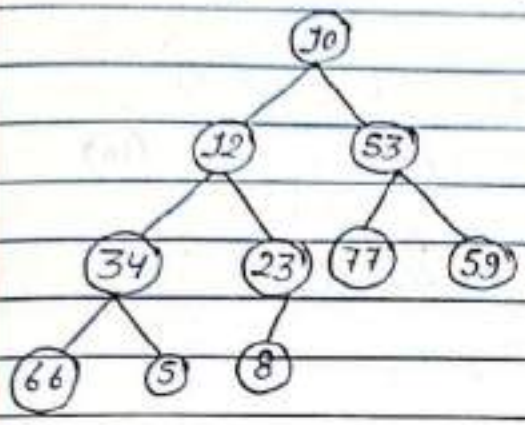
```
HeapSort (A)
{
    Buildheap (A);
    m = length [A];
    for ( i = m; i >= 2; i--)
    {
        swap ( A[1], A[m])
        m = m - 1
        Heapify (A, 1);
    }
}
```

Q. Sort the following data using heap sort algorithm.

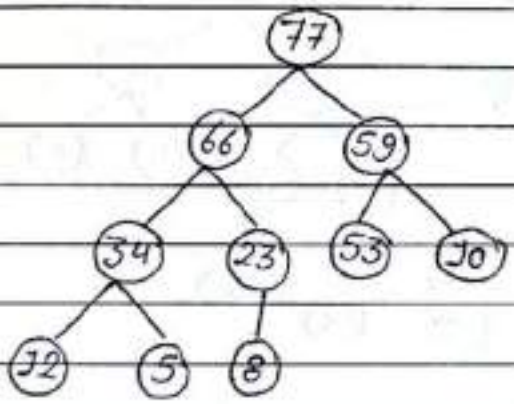
A[ ] = {10, 12, 53, 34, 23, 77, 59, 66, 5, 8}

=> Sol:

Constructing a tree using the given data.

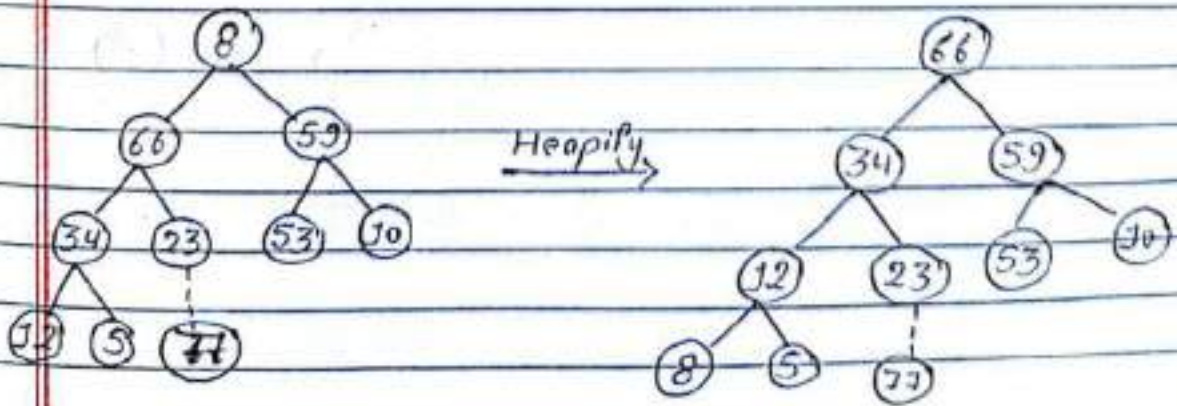


After heapified (max head) the required tree is,

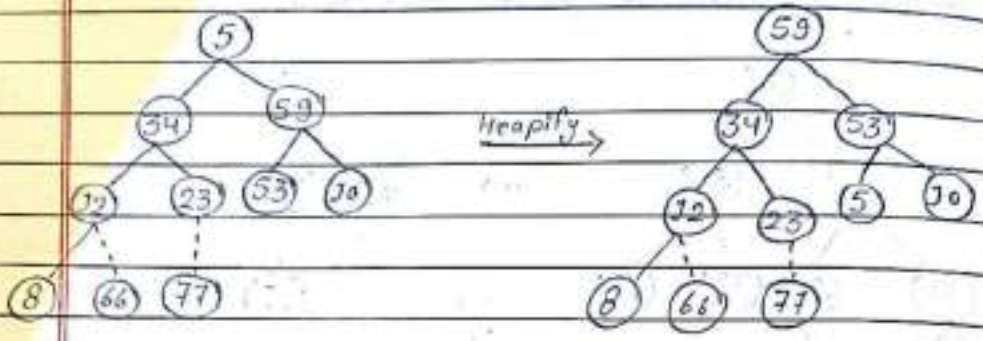


Now, Sorting the given tree.

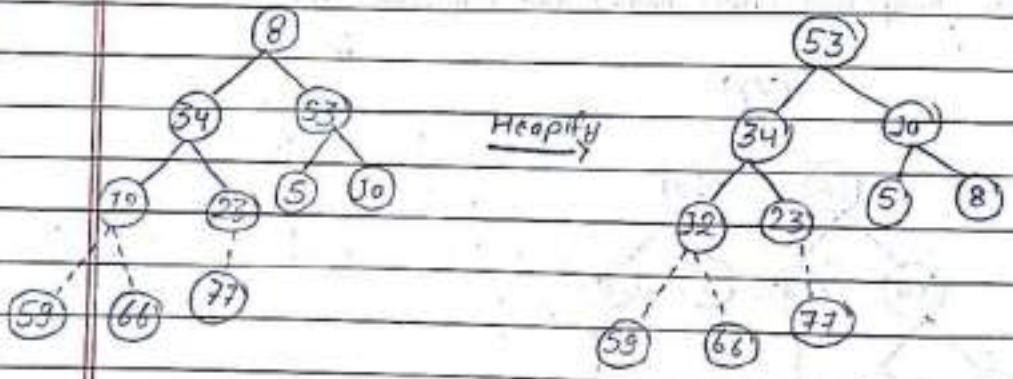
Iteration 1:-



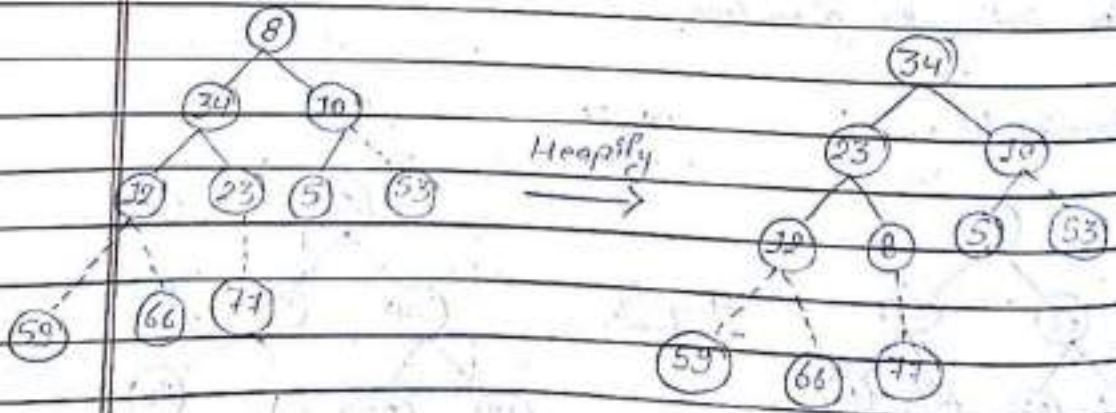
Iteration 2:-



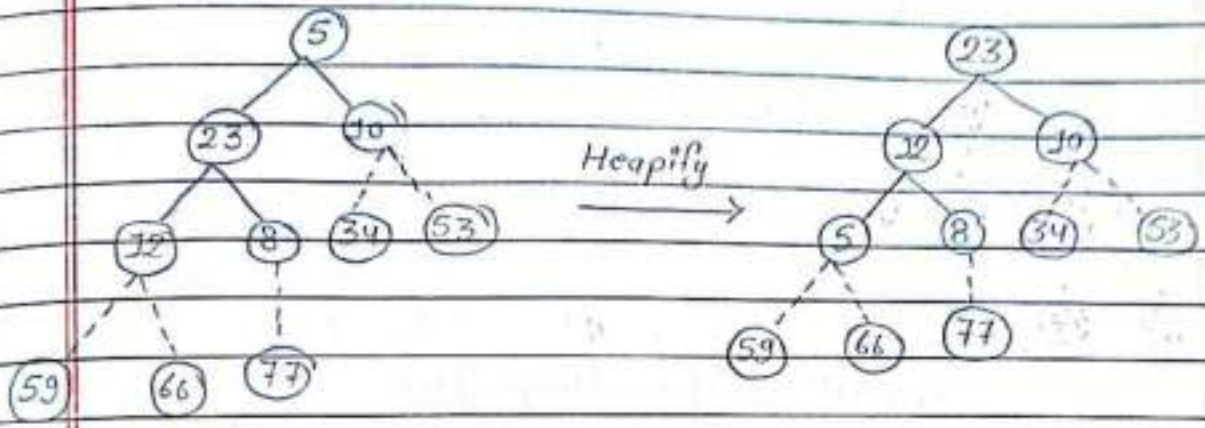
Iteration 3:-



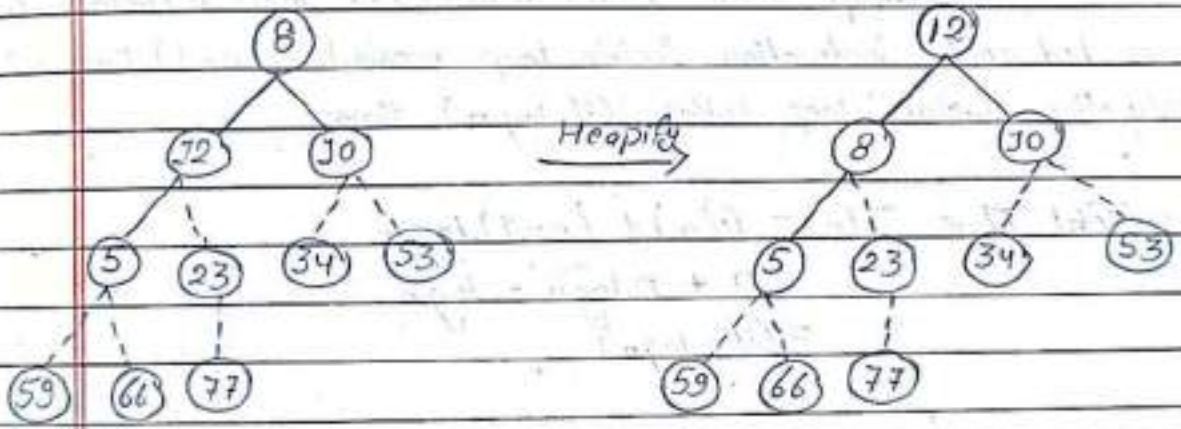
Iteration 4:-



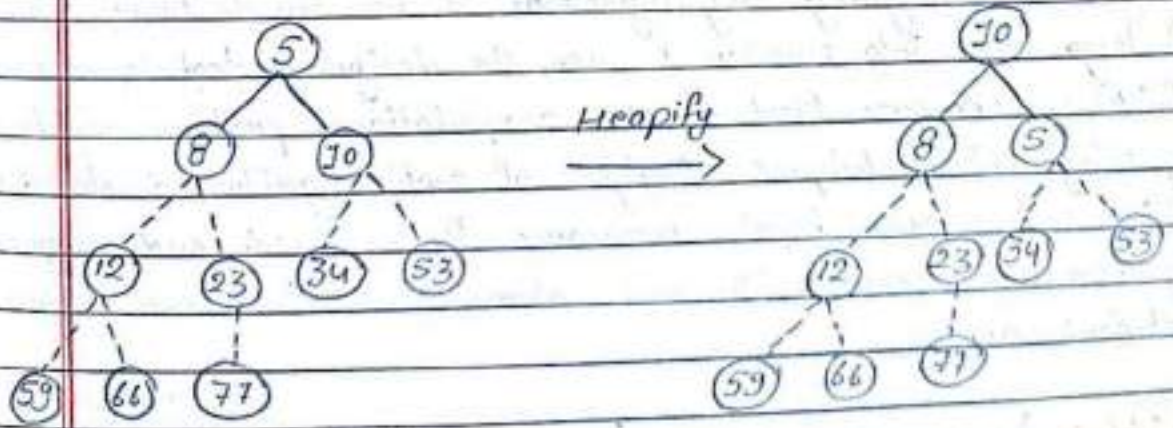
Iteration 5:-



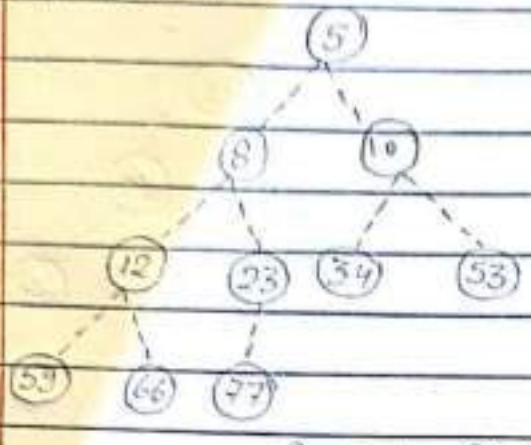
Iteration 6:-



Iteration 7:-



Iteration 8:-



Required Sorted Heap Tree

Complexity

Buildheap takes $O(n)$ times. For loop executes n times but each instruction inside loop executes $(n-1)$ time when instruction inside loop takes $O(\log n)$ times.

Total Time, $T(n) = O(n) + (n-1)\log n$

$= n + n\log n - \log n$

$= O(n \log n)$

Divide and Conquer Algorithm

While designing any algorithm if we try to break the problem of large size into smaller we use the designing techniques called divide and conquer. Most of the computational problem can be solved by using this techniques. Analysis of such algorithm developed using divide and conquer needs recurrance. Since, divide and conquer algorithm use recursion. The main elements of divide and conquer solutions are,

[a] Divide :- Divide the larger problem of smaller size.

[b] Conquer :- Solve each piece by applying recursive divide and conquer.

[c] Combine :- add up all the solved pieces to a larger solution.

Binary Search

An array of element A[] has sorted element.

Pseudocode:-

```
Binary Search( low, high, Key)
{
    if ( high == low)
    {
        if (Key == A[low] then return low+1;
        else
            return 0;
    }
    else
    {
        mid = (low + high)/2
        if(Key == A[mid] then return mid+1;
        else if (Key < A[mid] then return Binarysearch (low,
                mid-1, Key);
        else return Binary search (mid+1, high, Key);
    }
}
```

For e.g:-

Vibhu
PAGE No
DATE

Trace the binary search

0	1	2	3	4	5	6	7	8	9
75	131	203	275	318	489	524	591	647	727

Search for X = 275, 727 and 349

⇒ Sol^n

For X = 275

Iteration	Lower	Upper	Middle	Remarks
0	0	9	4	X < K[4]
1	0	3	1	X > K[1]
2	2	3	2	X > K[2]
3	3	3	3	X == K[3]

Search Successful.

For X = 727

Iteration	Lower	Upper	Middle	Remarks
0	0	9	4	X > K[4]
1	5	9	7	X > K[7]
2	8	9	8	X > K[8]
3	9	9	9	X == K[9]

Search Successful.

For X = 349

Iteration	Lower	Upper	Middle	Remarks
0	0	9	4	X > K[4]
1	5	9	7	X < K[7]
2	5	6	5	X < K[5]
3.	5	4	4	X > K[4]
4	5	4	4	X > K[4]

Search is unsucessful.

Complexity

From above algorithm we can say that the running time of algorithm is,

$$T(n) = T(n/2) + O(1)$$
$$= O(\log n)$$

Sequential Search

Pseudocode:-

```
for(i=0; i<n; i++)
{
    if(a[i]==data)
    {
        printf("Element found");
        break;
    }
}
if(i==n)
{
    printf("Element not found");
}
```

Time Complexity:

Best case = $O(1)$

Worst case = $O(n)$

$$\text{Average case} = \frac{\sum \text{all cases}}{\text{no. of cases}} = \frac{1+2+\dots+n}{n} = \frac{n(n+1)}{2n}$$

$$= O(n)$$

Max and Min Finding.

Here our problem is to find the minimum and maximum elements in a set of 'n' elements. We use two methods i.e iterative method and divide & conquer method.

1. Iterative method.

Pseudocode:

```
Iterminmax(A,n)
{
    max = min = A[0]
    for( i=1; i<n; i++ )
    {
      if(A[i] > max)
        max = A[i]
      if(A[i] < min)
        min = A[i]
    }
}
```

The above algorithm requires 2(n-1) comparison in worst case, best and average case. Since, the comparison A[i] < min is needed only A[i] > max is not true.

If we replace the content inside the loop by,

```
if(A[i] > max)
    max = A[i];
else if(A[i] < min)
    min = A[i];
```

The best case occurs when the elements are in increasing order with (n-1) comparison.

Worst case occurs when elements are in decreasing order with 2(n-1) comparison. So, the complexity is O(n).

Q. Find the maximum and minimum elements of.

0	1	2	3	4
4	8	0	2	5

⇒ Sol^{n}

max = A[0] = 4

min = A[0] = 4

Iteration 1:- A[1] > 4 A[1] < 4

8 > 4 (T) 8 < 4 (F)

max = 8 min = 4

Iteration 2:- A[2] > 8 A[2] < 4

0 > 8 (F) 0 < 4 (T)

max = 8 min = 0

Iteration 3:- A[3] > 8 A[3] < 8

2 > 8 (F) 2 < 0 (F)

max = 8 min = 0

Iteration 4:- A[4] > 8 A[4] < 0

5 > 8 (F) 5 < 0 (F)

max = 8 min = 0

2. Divide and Conquer method.

Pseudocode:-

```
Minmax ( i, j, max, min)
{
  if (i==j)
    max=min=A[i];
  else if ( i=j-1)
  {
    if (A[i] < A[j])
    {
      max = A[j];
      min = A[i];
    }
    else
    {
      max = A[i]; min = A[j];
    }
  }
  else
  {
    mid = (i+j)/2;
    minmax( i, mid, max, min);
    minmax (mid+1, j, max1, min1);
    if (max1 > max)
      max = max1;
    if ( min1 < min)
      min = min1;
  }
}
```

The above algorithm adopt the following idea if the number of element is 1 or 2 then max and min are obtained trivially otherwise split the problem into approximately equal part and solved recursively.

Analysis.

Here we analyze in terms of numbers of comparison as cost because elements may be polynomials. Now, we can use recurrance relation as,

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

Solving the recurrance relation.

$$T(n) = O(n)$$

Q. Find the min max of following data using divide and conquer.

0	1	2	3	4	5
4	5	2	1	0	8

Soln

Here,

$i = 0$, $j = 5$

Step 1:-

$mid = (0+5)/2 = 2$

Minmax (0 , 2 , max , min)

~~Minmax (3 , 5 , max , min)~~

~~Step 2:-~~

$mid = (i+j)/2 = (0+2)/2 = 1$

Minmax (0 , 1 , max , min)

$i = j - 1$ i.e $0 = 1 - 1$ (T)

if (A [i] < A [j]) i.e (4 < 5) (T)

max = 5

min = 4

Minmax (2, 2, max1, min1)

if (i == j) (T)

max1 = min1 = A[2] = 2

if (max1 > max) i.e if (2 > 5) F

max = 5

if (min1 < min) i.e if (2 < 4) T

min = 4

Step 2:- i = 3, j = 5

mid = (i+j)/2 = (3+5)/2 = 4

Minmax (3, 4, max, min)

i = j-1 i.e 3 = 4-1 (T)

if (A[i] < A[j]) i.e (1 < 0) F

min = 0

max = 1

Minmax (5, 5, max1, min1)

i == j

max1 = min1 = 8

if (max1 > max) i.e (8 > 1) T

max = 8

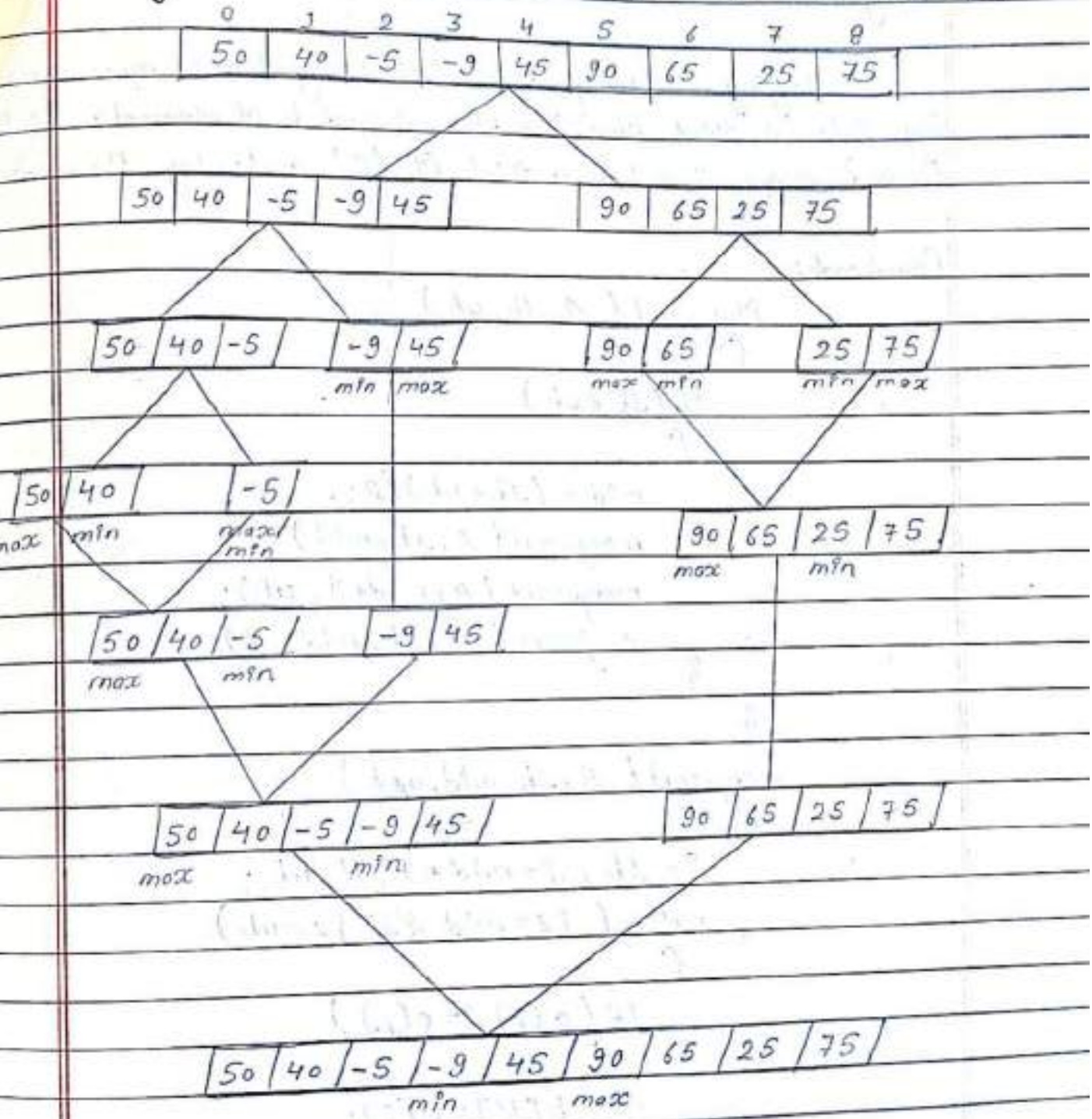
if (min1 < min) i.e (8 < 0) F

min = 0

∴ max = 8 and min = 0

Next Method :-

For e.g:- 50 40 -5 -9 45 90 65 25 75



Merge Sort.

Merging is the process of combining two or more sorted files into a third files. The files a and b of elements n_1 and n_2 and merge them into a third file 'c' containing n_3 elements

Pseudocode:-

```
Mergesort ( A, lb, ub )
{
    if ( lb < ub )
    {
        mid = (lb+ub)/2;
        mergesort ( A, lb, mid ).
        mergesort ( A, mid+1, ub);
        mergesort ( A, lb, mid, ub);
    }
}
mergesort ( A, lb, mid, upb )
{
    i = lb , j = mid+1 , K = ub ;
    while ( i <= mid && j <= ub )
    {
        if ( a[i] <= a[j] )
        {
            b[K] = a[i];
            K++ ;
        }
        else
        {
            b[K] = a[j]; j++;
            K++;
        }
```

```
}
if (i > mid)
 {
    while (j <= ub)
     {
       b[k] = a[j]; j++, k++;
     }
 }
else
 {
    while( i <=mid)
     {
       b[k] = a[i]; i++; k++;
     }
    for(k=lb; k<=ub; k++)
     {
       a[k] = b[k];
     }
 }
}
```

Quick Sort

Quick sort was developed by C.A.R Hoare is an unstable sorting. This algorithm is based on divide and conquer. The main idea behind this algorithm is partitioning of the element.

Steps of Quick sort.

Divide :- Partition the array into two non-empty subarrays.

Conquer :- Two sub arrays are sorted recursively.

Combine :- Two sub arrays are already sorted in place and so no need to combine.

Partitioning

An array A[] is said to be partitioning about A if all the elements on the left of A[t] are less than or equal all elements on right are greater than or equal to A[t].

Partition pseudocode

```
Partition (A, i, j)
{
    x = i-1; y = j+1; v = A[i];
    while (x < y)
    {
        do { x++; }
        while (A[x] <= v)
        do {
            y--;
        }
        while (A[y] >= v)
        if (x < y)
            swap(A[x], A[y]);
    }
    A[i] = A[y], A[y] = v;
    return y;
}
```

It can be written as,

1. Repeatedly increase the pointer down by one position until x[down] <= a.
2. Decrement the pointer up by one position until x[up] >= a
3. If up > down interchange x[down] with x[up].
4. The process continues until the condition fails [up <= down] x[up] is interchange with x[lb].

For e.g:-

Sort the following data using quick sort.

A[] = { 25, 57, 48, 37, 12, 92, 86, 33 }

a = x[lb] = 25

25	57	48	37	12	92	86	33
↑ up, down							↑ up

25	57	48	37	12	92	86	33
	↑ down						↑ up

25	57	48	37	12	92	86	33
	↑ down					↑ up	

25	57	48	37	12	92	86	33
	↑ down				↑ up		

25 57 48 37 12 92 86 33
↑ down ↑ up

25 12 48 37 57 92 86 33
↑ down ↑ up

25 12 48 37 57 92 86 33
↑ down ↑ up

12 25 48 37 57 92 86 33
↑ up ↑ down

Here, up < down so interchange x[up] and x[lb]. 25 is pivot point.

Again take right subarray.

48 37 57 92 86 33

a = x[lb] = 48

48 37 57 92 86 33
↑ up down ↑ up

48 37 57 92 86 33
↑ down ↑ up

48 37 57 92 86 33
↑ down ↑ up

48	37	33	92	86	57
		↑ down			↑ up
48	37	33	92	86	57
			↑ down		↑ up
48	37	33	92	86	57
			↑ down	↑ up	
48	37	33	92	86	57
			↑ down/up		
48	37	33	92	86	57
		↑ up	↑ down		
33	37	48	92	86	57
		↑ up	↑ down		

Here, up < down so interchange x[up] and x[lb]. 48 is pivot point.

Again take right subarray.

92 86 57

a = x[lb] = 92

92	86	57
↑ down		↑ up
92	86	57
	↑ down	↑ up
92	86	57
		↑ up/down

57 86 92
↑
up/down

Algorithm for QuickSort :

```
QuickSort ( A, p, q)
{
  if (p<q)
  {
    r = partition (A, p, q);
    QuickSort (A, p, r-1);
    QuickSort (A, r+1, q)
  }
}
```

In partition algorithm either left pointer or right pointer moves at a time and Goon the array at most time. We can note that at most n swaps are done ha the time complexity of portion is O(n).

Complexity Analysis :

1. Best Case :

The best case for divide and conq relation occurs when division is as balanced as possible If we can choose pivot as middle part of the array We have,

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

Solving this we get,

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

2. Worst Case :

Worst case occurs if the partition gives pivot as first element (last element) all the time. ie $k = 1$ or $k = n$. This happens when the elements are completely sorted. So we have

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

Solving these we get

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

3. Average Case :

Case when the partition is balanced but not as a best case partion. Let us suppose that partition algorithm always produce 9:1 split of the array. Here 9:1 seems unbalanced, for this situation we can write the recurrence relation as

$$T(n) = T(9n/10) + T(n/10) + O(n)$$

Solving this we get,

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

Greedy Algorithm :

It is the simplest way of algorithm design. The general class of problem solved by greedy approach is optimization problem. In this approach the input element are exposed to some constraints to get feasible solution and feasible solution that meets some objective function best among all solution is called optimal solution.

Greedy algorithm always make optimised choice ie l
local to generate globally optimal solution. However
is not guaranted that all greedy algorithm produce
optimal solution, problem solved using greedy approa
have two parts.

1. Greedy Choice Property

Globally optimal solutio
can be obtained by making locally optimal choice and
the choice at present cannot reflect possible choice of
future.

2. Optimal Sub Structure property

Optimal sub structure is defined by a problem if an
optimal solution to the problem contains optimal solutio
to the sub problem within it.

Fractional Knapsack Problem :

Statement : A thief has a bag or knapsack tha
can contain maximum weight (W) of hi
loot. There are n items and weight of ith items
is Wi and it's worth Vi. Any amount of item
can be put into the bag that is xi fraction of
item can be collected. When $0 <= x_i <= 1$. Here
objective is to collect the items that maximize
the total profit earned.

We can formally state this problem as maximize

$$\sum_{i=1}^{n} x_i w_i \leq W$$

Here in this problem, it is clear that any optimal solution must be fill the knapsack completely otherwise there will be partial left that can be filled by some part of some items.

Algorithm:-

Take as much of the item with the highest value per weight (V_i/W_i) as we can. If the item is finished then move on the next item that has highest (V_i/W_i) continue this until the Knapsack is full.

V[1---n] and W[1---n] contains the values and weights respectively of the 'n' objected sorted in an increasing ordered of V[i]/W[i]. W is the capacity of Knapsack. X[1---n] is the solution vector that includes fractional amount of items and 'n' is the number of items.

```
GreedyFracKnapsack(W,n)
{
  for(i=1; i<=n; i++)
    x[i] = 0.0;
  tempW = W;
  for(i=1; i<=n; i++)
  {
    if(W[i] > tempW) then break;
    x[i] = 1.0;
    tempW- = w[i];
  }
```

```
if ( i<=n )  x[i] = temp W/w[i];
}.
```

For eg:-

Consider 5 items along their respective weights and values.

$I = \{I_1, I_2, I_3, I_4, I_5\}$

$w = \{5, 10, 20, 30, 40\}$

$V = \{30, 20, 100, 90, 160\}$

The capacity of Knapsack $W = 60$. Find the solution to the fractional Knapsack problem.

=>

Solution:-

Initially

Items	w_i	V_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight $P_i = V_i/w_i$

Items	w_i	V_i	$P_i = V_i/w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Now, arrange the value of P_i in non-decreasing order.

Items	w_i	V_i	$P_i = V_i/w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the Knapsack according to the decreasing value of P_i. First we choose item I_1 whose weight is 5 then choose item I_3 whose weight is 20. Now total weight in Knapsack is $5 + 20 = 25$. Now, next item is I_5 and its weight is 40 but we want only 35. So we choose fractional part of it. i.e

35	I_5
20	I_3
5	I_1

60

Total value of fractional part of I_5 is,

$$\frac{160}{40} \times 35 = 140$$

Thus, the maximum value is,

$$30 + 100 + 140 = 270$$

Analysis

We can see that the above algorithm contains a single loop i.e no-nested loop. The running time for above algorithm is O(n). However our requirement is that V[1---n] and W[1-- are sorted so we can sorting method to sord it in O(nlogn) time such that the complexity of above including sorting beco O(nlogn).

Huffman Codes

Huffman codes are used to compare data by representing each alphabets by unique binary codes in an optimal way. Consid a file of 100,000 character with the following frequency distribu assuming that there are only 7 characters.

f(a) = 40,000 , f(b) = 20,000 , f(c) = 15,000 , f(d) = 12,000
f(e) = 8,000 , f(f) = 3,000 , f(g) = 2,000.

Here, fixed length code for 7 characters we need 3 bits to represent all characters like a = 000, b = 001, c = 010, d = 011, e = 100, f = 101, g = 110.

Total number of bits required due to fixed length code is 300

Now consider variable length character so that the character with highest frequency is given smaller code like.

a = 0 , b = 10 , c = 110 , d = 1110 , e = 11111 , f = 111101,
g = 111011

∴ Total number of bits required due to variable length is,

40,000x1 + 20000x2 + 15000x3 + 12000x4 + 8000x5 + 3000x6 + 2000x6 =

Algorithm

A greedy algorithm can construct Huffmann code is optimal prefix codes. A tree corresponding to optimal codes is constructed in a bottom-up manner starting from the |c| leaves and |c|-1 merging operations.

A priority queue 'Q' to keep nodes ordered by frequency. Here, the priority queue we consider is binary heap.

```
HuffmanAlgo(c)
{
    n = |c|;
    Q = c;
    for(i=1; i<n-1; i++)
    {
        z = Allocate_Node();
        x = Extract-min(Q);
        y = Extract-min(Q);
        left(z) = x;
        right(z) = y;
        f(z) = f(x) + f(y);
        insert(Q,z);
    }
    return Extract-min(Q)
}
```

For eg:-

Character	Frequency
a	40
b	20
c	15
d	12
e	8
f	3
g	2

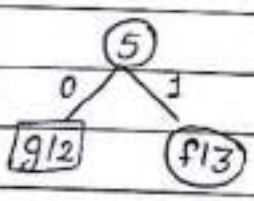
⇒ Soln

Initial priority queue is, (frequency के ascending order में अक्षरों को लेकर, यह Priority queue में)

g|2	f|3	e|8	d|12	c|15	b|20	a|40

Iteration 1:-

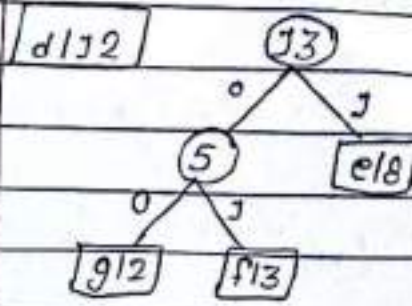
i = 1 (Queue में अक्षरों को smaller frequency लेके Sum करें)



e|8	d|12	c|15	b|20	a|40

Iteration 2:-

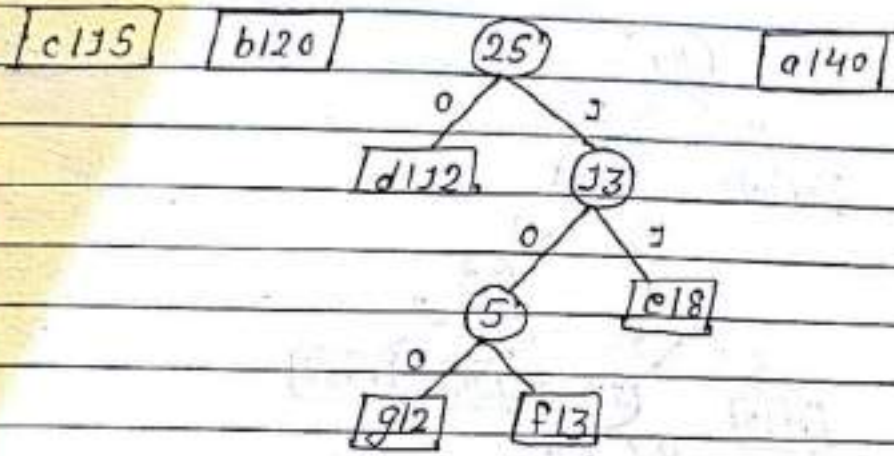
i = 2



c|15	b|20	a|40

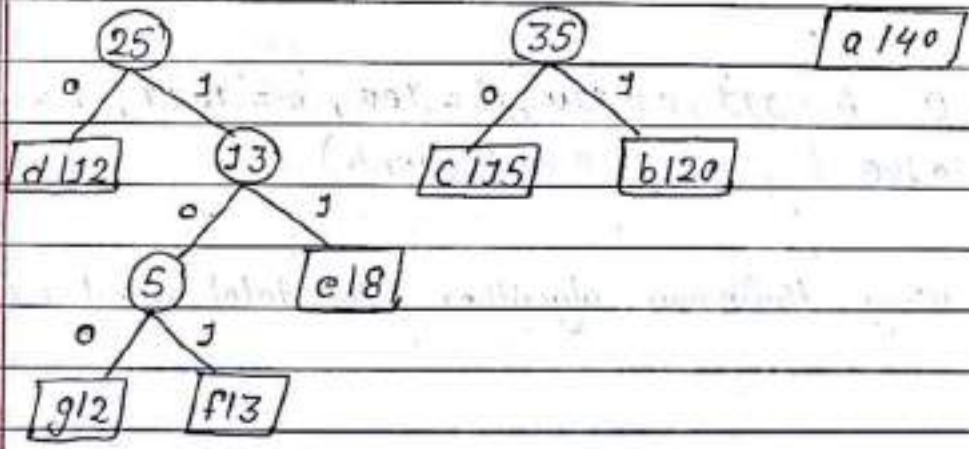
Iteration 3:-

$i = 3$



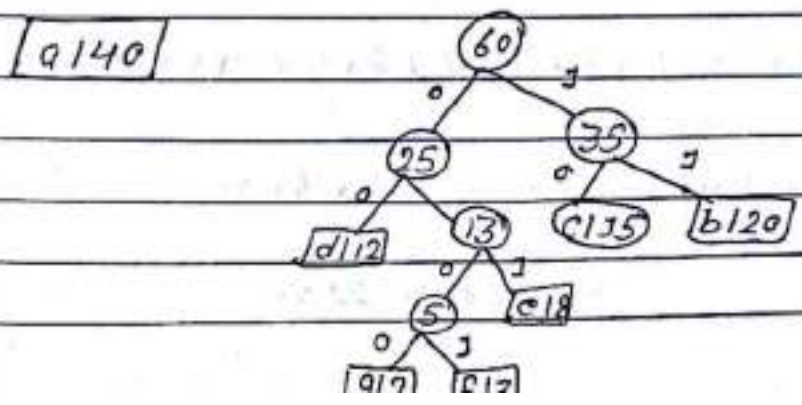
Iteration 4:-

$i = 4$



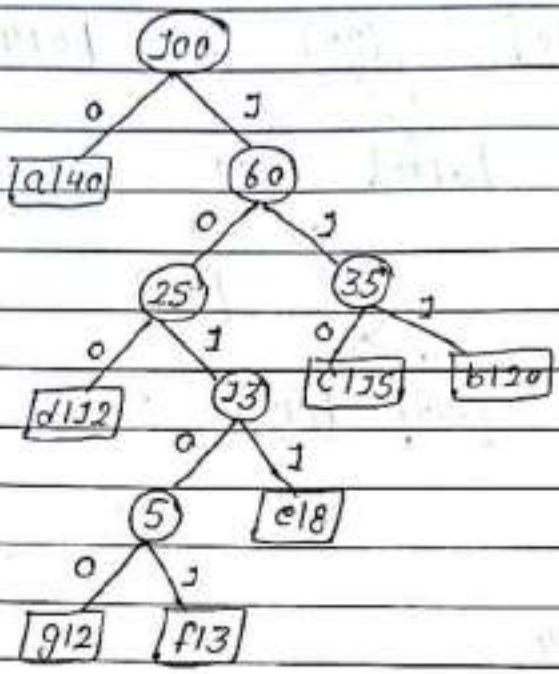
Iteration 5:-

$i = 5$



Iteration 6 :-

$i = 6$



Here,

$a = 0$, $b = 111$, $c = 110$, $d = 100$, $e = 1011$, $f = 1101$ and $g = 11100$ (path travelled in graph)

Before using Huffman algorithm, the total number of bit required is,

$40 \times 3 + 20 \times 3 + 15 \times 3 + 12 \times 3 + 8 \times 3 + 3 \times 3 + 2 \times 3 = 300$

Using Huffman algorithm the total number of bit required is,

$40 \times 1 + 20 \times 3 + 15 \times 3 + 12 \times 3 + 8 \times 4 + 3 \times 5 + 2 \times 5 = 238.$

Space saved using Huffman $= \frac{300 - 238}{300} \times 100\%$

$= 20.67 \approx 21\%$

Analysis

We can use buildheap(c) to create a priority queue that takes O(n) time. Inside the for loop the expensive operation can be done in O(logn) time. Since, operation inside for loop executes for (n-1) time total running time of Huffman algorithm is O(nlogn) time.

Dynamic Programming

It is the powerful way of designing the algorithm. Dynamic programming is widely used to solve the optimization problem. Here, a set of choices is made to come up with optimal solution. Most of the problem deals with dynamic techniques consists of sub-problems recurring more than one time while calculating the solution to the problem. Here if we store the result of solution of the sub-problem then we donot have to re-calculate the solution of subproblem later and can be used when necessary. The optimization problem that are solved using dynamic programming have two properties:-

[a] Optimal Substructure

Optimal substructure is exhibited by a problem if an optimal solution to a problem contains optimal solution to the subproblem within it.

[b] Overlapping Subproblems.

The given problem has overlapping sub-problem if an algorithm calculate the solution to the subproblem more than once in order to use its result for problem of size greater than the solved subproblems.