__/__/____

DAA

→ Algorithm : It is a combination of finite steps to solve a particular problem.

It is basically a set of rules that must be followed when solving a specific problem. An algorithm is considered to be correct if for every input instance it generates the correct output & gets terminated.

The incorrect algorithm does not terminate on the desire output for the input instance or might terminate for some other output value other than the desired one.

1.) Take two inputs
2.) c = a+b
3.) Print c

→ Analysis of an algorithm -

Time complexity is always more important than space complexity.

Time complexity : It is the amount of time needed by a program or algorithm to run to completion is referred as time complexity.

Compiler + Running / Execution time
(depends on compiler) (depends on processor)

__/__/____

c will be more faster because it directly executes the program. ie compilation is not a necessary step in c.
Java takes more time as compilation is must.

Space complexity: The amount of memory needed for a program to run to completion is known as space complexity.
space is never an issue, time's a issue.

* We deal with different time may arise for same algorithm. This is due to the processor as different systems have different processors.

→ Best case: The minimum amount of time that an algorithm requires for an input size of n is referred as best case complexity.

→ Worst case: The maximum amount of time needed by an algorithm for input of size n is referred as worst case complexity.

→ Average case: It is the execution of algorithm having typical input data of size n.

Time complexities:-

1) Big-oh (Asymptotic Complexity)

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

$f(n) \leq c \cdot g(n)$ $\quad n \leq n^2$

__/__/____

eg - $f(n) = n$, $g(n) = n$
$n = n$
Big-Oh

eg - $f(n) = n^3$
$g(n) = n$
Not big-oh (worst case)

2) Omega : $f(n)$ and $g(n)$ are two +ve function
then $f(n) = \Omega g(n)$ if and only if
$f(n) \geq c \cdot g(n)$
↓
constant
Lower bound
Best case

3) Theta : Let $f(n)$ and $g(n)$ be two positive function,
then $f(n) \geq c \cdot g(n)$ if and only if
$f(n) = \theta \cdot g(n)$
$f(n) \geq c \cdot g(n)$
Average case if $f(n) = g(n)$.

4) Small o : Let $f(n)$ and $g(n)$ be 2 function
then $f(n) = o\, g(n)$ if and only if
$f(n) < o \cdot g(n)$

5) Small omega : $f(n) > g(n)$

__/__/____

→ Classes of Complexity –

1) Constant complexity : order of 1. O(1).

2) Logarithmic complexity : O(log(n)) if $f(n) < a \cdot g(n)$.

3) Quadratic complexity : $O(n^2)$.

4) cubic : $O(n^3)$

5) Polynomial : $O(n^c)$ c=1,2,3 ----

6) Linear : c >/ 1, 2, 3 ------

7) Exponential : order of power n $O(c^n)$
c value starting from 2. ↓ constant

→ Recursion : When a function calls itself again and again, it is termed as recursion.

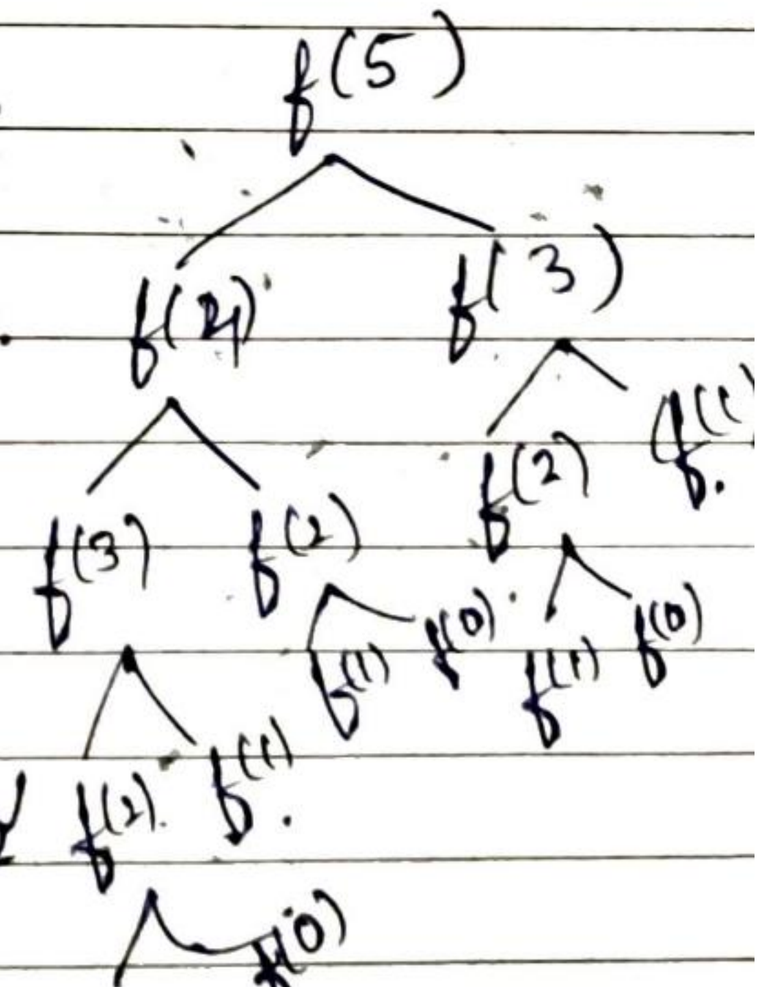
Program to find factorial of 5.

```
class Factorial {
  static int factorial (int n) {
    if (n == 0)
      return 1;
    else
      return (n * factorial (n-1));
  }
```

```
public static void main (String args[]){
    int i, fact = 1;
    int number = 5;
    fact = factorial (number);
    Syso ("Factorial of " +number + " is : +fact);
}
}
```

→ Program to calculate fibonacci series —

```
class fibonacci {
    static int n1 = 0, n2 = 1, n3 = 0;
    static void printfibonacci (int count) {
        if (count > 0) {
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;
            Syso (" " + n3);
            printfibonacci (count - 1);
        }
    }
    p s v m (String args[]) {
        int count = 10;
        Syso (n1 + " " + n2);
        printfibonacci (count - 2);
    }
}
```



$O(2^n)$

(1+2) (3)

__/__/____

H.W

Q- Divide & conquer ?
how merge sort work ?
whether array sorted in merge sort or not ?

Merge sort is a divide and conquer algorithm. It divides the input array into two halves, calls itself for the two halves and then merges the two sorted halves.

Unit - 2

Divide and conquer

__/__/____

Inplace: In sorting if no extra space.
Outplace: extra space needed.

⇒ Algorithm -

1.) It divides the input array into two halves calls itself for the two halves and then merge the sorted arrays.

2.) Merge sort (arr [], l, r) if r > l

3.) m = l + r-l/2

4.) merge sort (arr, l, m) (First half)

5.) (arr, m+1, r) (second half)

6.) (arr, l+m, r)

Analysis - (a, i, j)

if (i==j) Best case.
(single element array)

complexity : O(1).. No comparison as single element.

return a(i);

else $\left(\frac{i+j}{2}\right)$ at first, there will be $\frac{n}{2}$ elements and at the end, there will be n elements.

a (i, mid)
(a, mid+1)

Masta theaem __/__/____

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + T(n) \end{cases}$$

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

——— ***** ———

⇒ Classes of complexity

Sequence

$O(1) < O\log((\log n)) < O\log(n) < O(\sqrt{n}) < O(n)$
$< O(n \log n) < O(n^2) < O(n^3) < O(n^c) <$
$O(c^n) < O(n!) < O(n^3)$

↓ exponential ($c > 2$)

↓ polynomial ($c \geq 1$)

$\log_2 1000 = 10$ (ceiling value)
$\log_{10} 1000 = 3$

- $O(\log(n))$ | $O(\sqrt{n})$

  Taking log on both sides

  $\log(\log(n))$ | $\log n^{1/2}$
  $\log(\log(n))$ | $\frac{1}{2}\log n$

  $\therefore \log(\log(n)) < O\log(n)$

* To calculate complexity of code, focus on key operations —

Eg 1: main() { $x = y + z;$ } $O(1)$.

__/__/__

Eg 2 -

```
main() {
    x = y + z ;                    → (1)
    for (i = 1; i ≤ n; i++)
    {
        x = y + z ;                → (n)
    }
}
```

1 + n

∴ O(n)

Eg 3 -

```
main() {
    x = y + z ;                    — 1
    for (i = 1; i ≤ n; i++)        — n
    {
        a = b + c ;
    }
    for (j = 1; j ≤ n; j++)        — n
    {                                      ] n²
        for (k = 1; k ≤ n; k++)    — n
        {
            p = q + r ;
        }
    }
```

1 + n + n²

∴ O(n²)

Eg 4 -

```
for (i = 1; i ≤ n/2; i++)          — n/2
{                                          ]
    for (j = 1; j ≤ n; j++)        — n     ]
}
```

n × n/2 = n²/2

∴ O(n)

Eg 5 -

```
while (n ≥ 1)
{ n++; }
```

not an algo as this is not finite.

__/__/____

Eg 6 - main()
{
i = 1;
while (i ≤ n) — n
{
i++; }

∴ O(n)

Eg 7 - while (n ≥ 1)
{
n = n/2; }

$\frac{n}{2^0} \cdot \frac{n}{2^1} \cdot \frac{n}{2^2} \cdot \frac{n}{2^3} \cdot \cdots$

$\frac{n}{2^0} \cdot \frac{n}{2} \cdot \frac{n}{4} \cdot \frac{n}{8} \cdots$

$\frac{n}{2^k} = 1 \quad k = 0, \cdots$

$n = 2^k$

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

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

$\boxed{k = \log(n)}$

——*****——

⇒ Master theorem —

$\frac{a}{b} T(n) + f(n)$

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

∴ a, b are some constant. f(n) is some positive function.

Eg - $T(n) = \frac{8}{2} T(n) + n^2$

∴ a = 8, b = 2 ∴ $n^{\log_2 8}$

$= n^3$

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

(worst case)

Eg- $4\frac{T(n)}{2} + n^5$

$a = 4, b = 2$

$n^{\log_2 4} = n^2 + n^5$

$= O(n^5)$

Eg- $2\frac{T(n)}{2} + n$

$a = 2, b = 2$

$n^{\log_2 2}$

$n + n$

same

add $\log n$

$O(n \log n)$

Eg- $\frac{T(n)}{2} + C$

$a = 1, b = 2$

$n^{\log_2 1}$

$1 + C$

$C + C$

same

so, $O(\log n)$

__/__/____

→ **Rule for master theorem :-**

The given equation must be polynomial equation.

$$\frac{n^3}{n^2} = (n^1) \longrightarrow \text{polynomial form} \quad (\text{minimum power } 1)$$

Q- $2T\left(\frac{n}{2}\right) + n\log n$

$a = 2, b = 2$ → Master theorem not applicable

~~$n\log n + n\log n$~~

$n + n\log n$ as $\frac{n}{n\log n}$

⇒ $\underset{f(n)}{n} \mid \underset{T(n)}{n^2 \log n}$ not a polynomial equation.

↳ $\frac{n}{n^2 \log n}$ ✗ Not possible as not a polynomial eqn.

⇒ $\underset{f(n)}{n^2 \log n} \mid \underset{T(n)}{n}$

↳ $\frac{n^2 \log n}{n^2} \rightarrow n^1 \log n$ → polynomial eqn. of power 1.

Now, apply master theorem.

__/__/____

Complexity ? (worst case / upper bound)

↳ key operations..

Eg 1: main ()

{

x = y + 2; → O(1)

}

~~Eg 2~~

Tute sheet

main ()

{

x = y + z;

for (i = 1 ; i ≤ n ; i++)

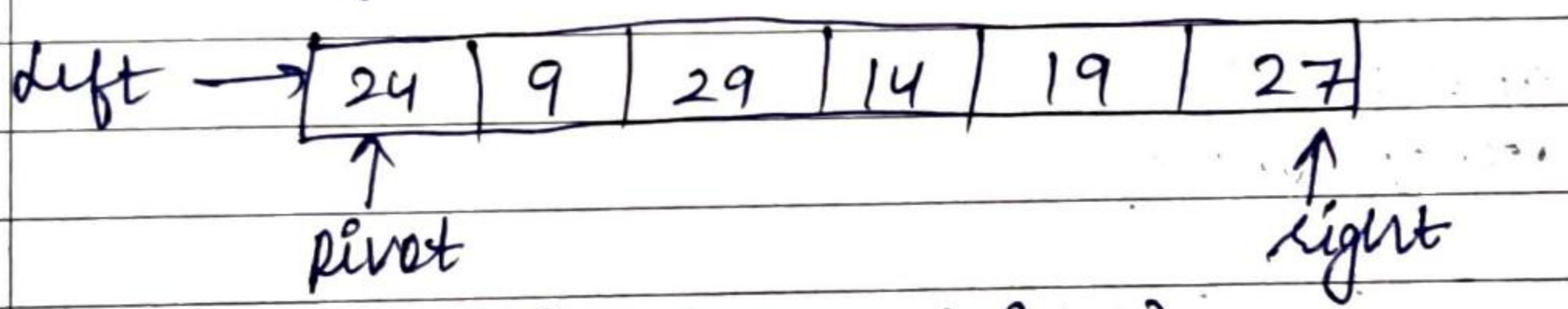
{

x = y + z;

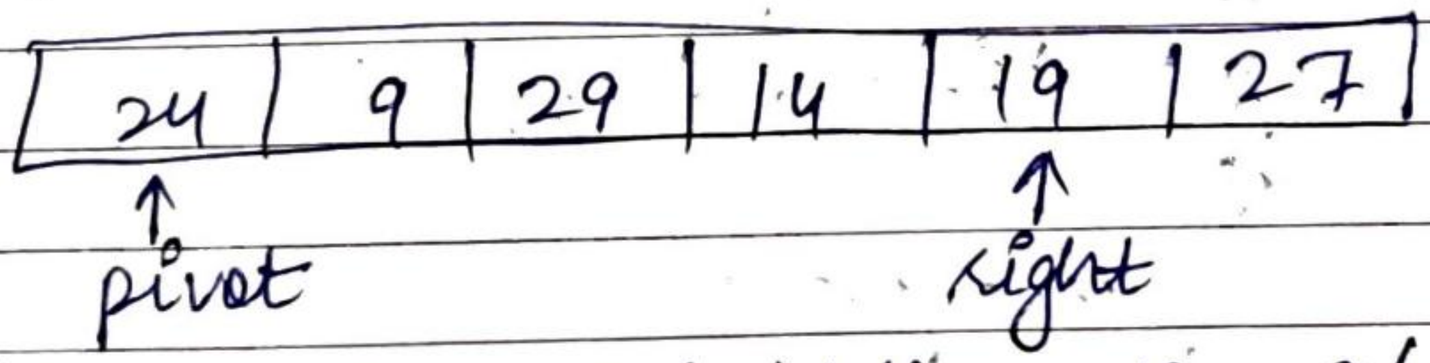
}

}

⇒ Quick Sort : Quick sort picks an element as pivot and then it partitions the given array around ~~the~~ the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (pivot) and another holds the values that are greater than the pivot.

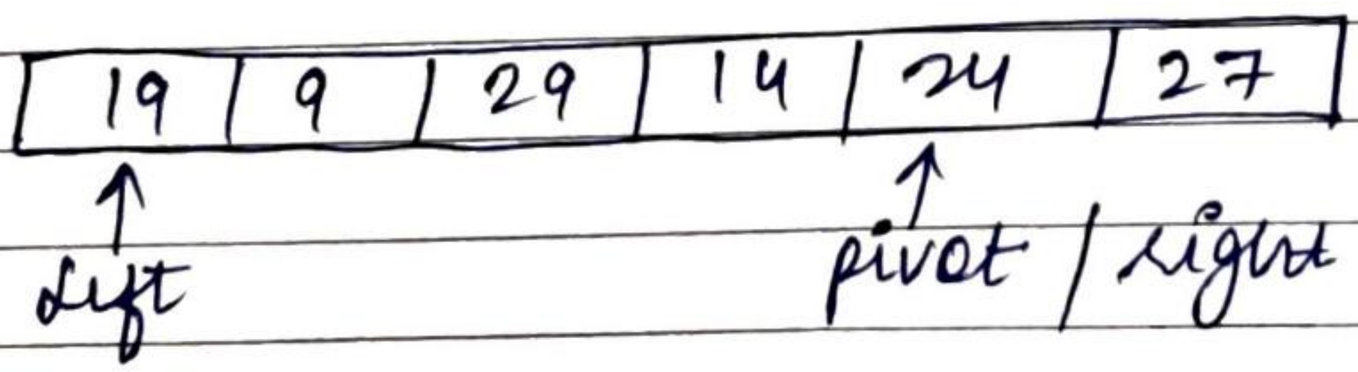


a (pivot) < a(right), so move one position towards left.

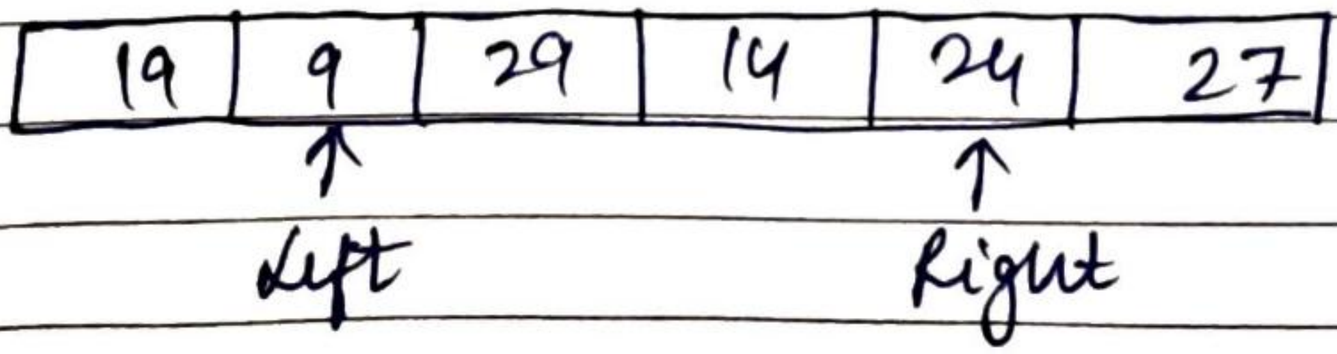


a (left) = 24, a (right) = 19, a(pivot) = 24

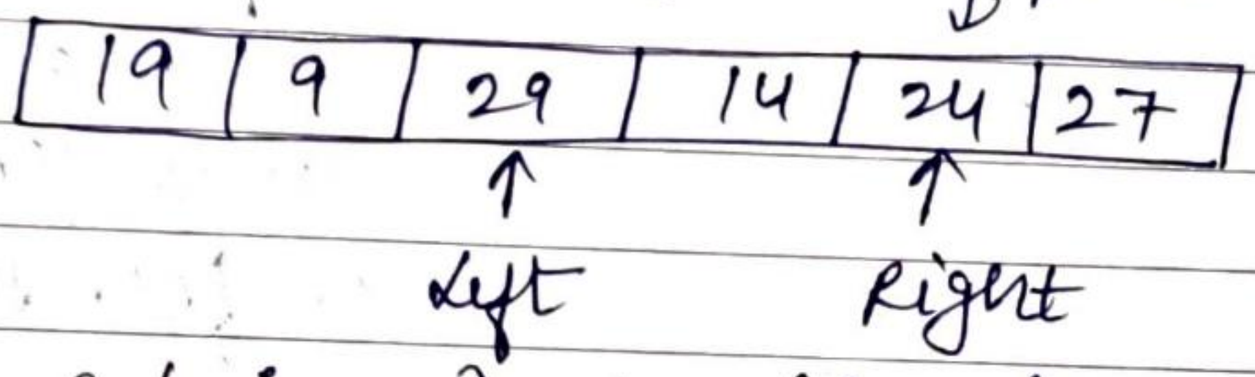
a(pivot) > a (right) ⇒ so swap.



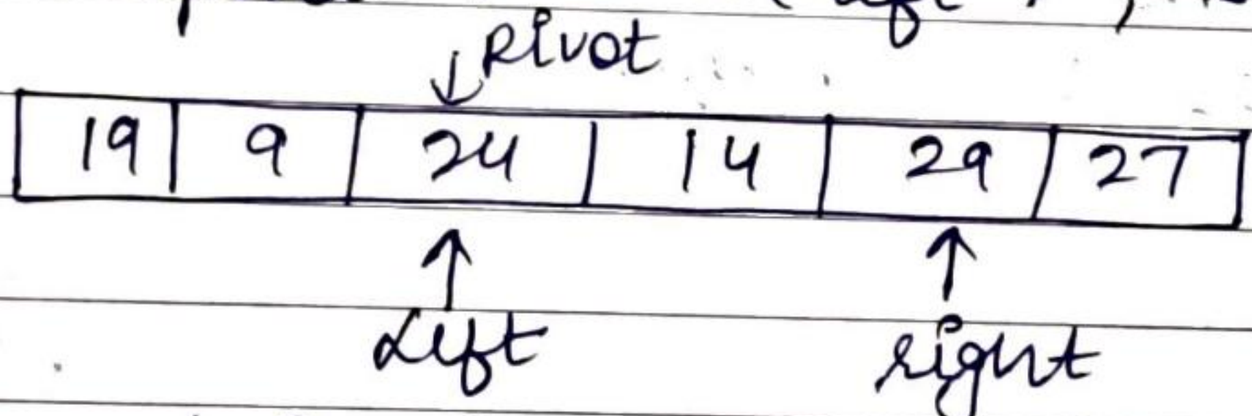
a (pivot) > a (left), so move one position of left to right.



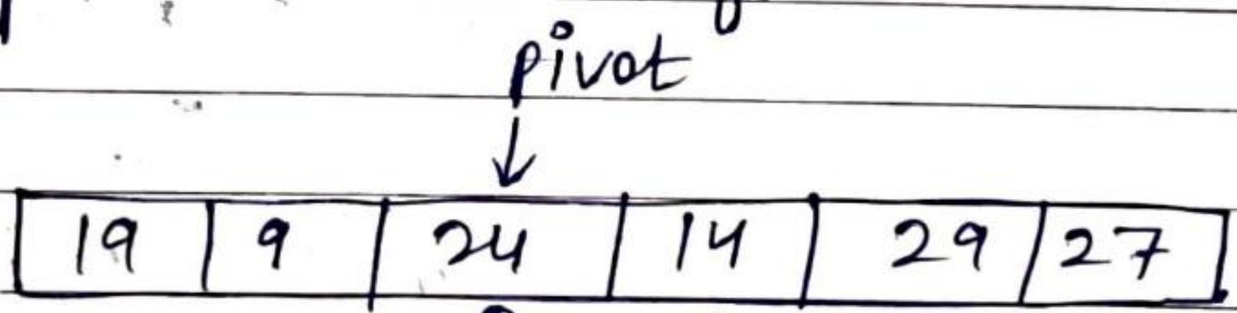
a(pivot) > a(left), so move one position to right.



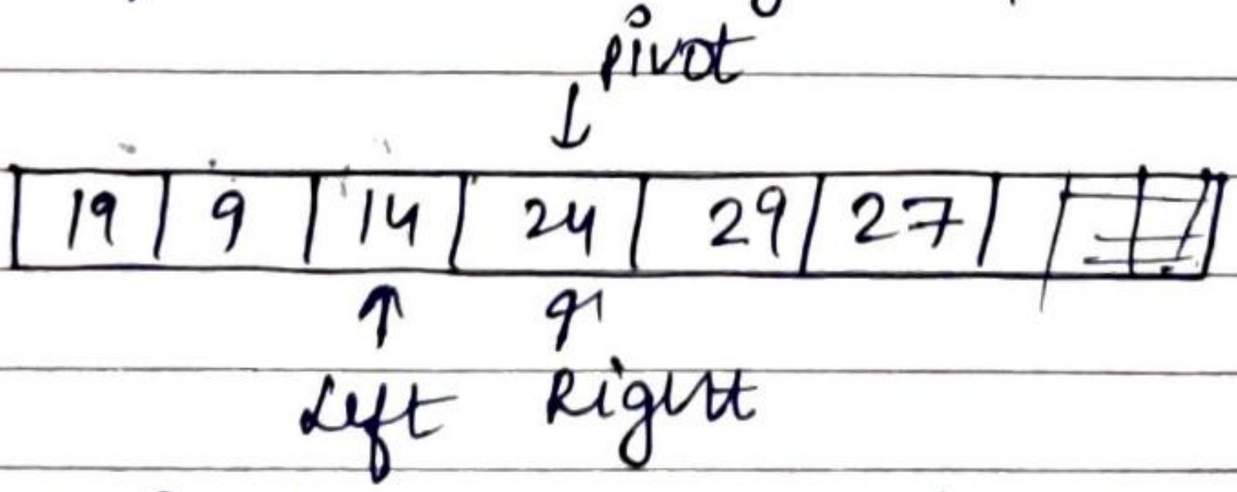
a(pivot) < a(left), so swap.



a(pivot) < a(right), so move one position to left.



a(pivot) > a(right), so swap.



Pivot is at right, so algorithm starts from

__/__/____

First element is choosen as pivot element.

a = b
b = c
a = c

Outplace concept is used as we are using temp variable c for sorting.

Worst case : $O(n^2)$
Best case : $O(n \log n)$

$T(n) + T$

$n(n) - n*(n-1)/2$

$n \times n - \frac{n*(n-1)}{2}$

$O(n^2)$

→ Partition (a, p, q)

```
{
    i = p;
  x = a[i];
  for (j = p+1 ; j ≤ q , j++)
  {
    if (a[j] ≤ x)
    {
      i = i+1;
  swap (a[i] , a[j]);
    return i;
  }
    swap (a[i], a[p]);
  return i;
}
```

→ Quick sort (a, p, q)

(a → array, p → start, q → end.)

```
{
  if (p == q)
    return a[p];
  else
    m = partition (a, p, q)

  Quick sort (a, p, m-1);
  Quick sort (a, m+1, q);
  return a;
}
```

Q- 45, 70, 90, 35, 25, 48, 80, 120, 155

p = 45 , q = 155

45 70 90 35 25 48 80 120 155

pivot, right

45 70 90 35 25 48 80 120 155

pivot, right

45 70 90 35 25 48 80 120 155

pivot, right

45 70 90 35 25 48 80 120 155

right

45 70 90 35 25 48 80 120 155

pivot > right (swap)

25 70 90 35 45 48 80 120 155

left, pivot/right

25 70 90 35 45 48 80 120 155

left, pivot

pivot < left (swap)

25 45 90 35 70 48 80 120 155

pivot/left, Right

25 45 90 35 70 48 80 120 155

$n-1$

$1-1=0$

Best Case : $p == q$

complexity : $O(1)$.

comparisons : $n-1$

$1-1 \rightarrow 0$

For swap ;

$BC \rightarrow 1$

$WC \rightarrow n-1$

Two key operations — Comparison

— Swapping

$$\Rightarrow T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(m-p) + T(q-m) + n \end{cases}$$

starting, end, index value

$T(m-p)$: left side ; $T(q-m)$: right side

$T(m-p)$

$T(5-1)$

$T(4)$

→ 4 elements on left side

$T(q-m)$

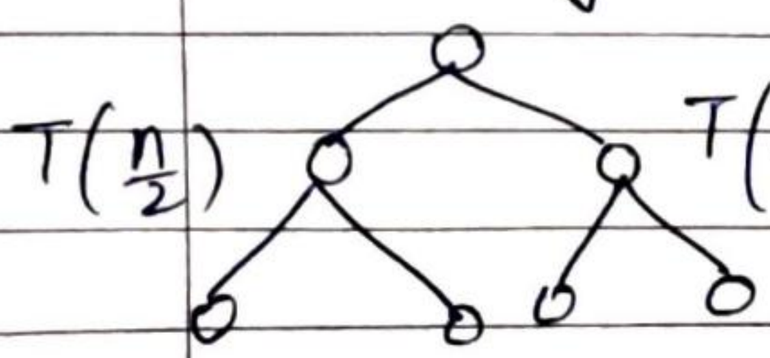
$(10-5)$

$T(5)$

→ 5 elements on right

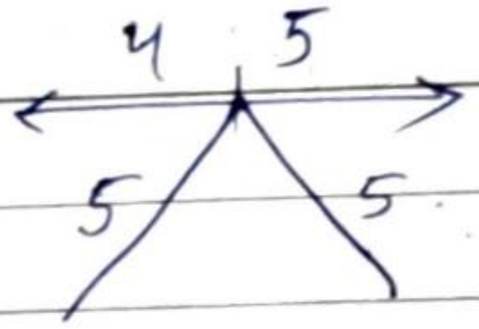
→ There is no guarantee in quick sort that equally it will divide.

→ Quick sort doesn't provide any guarantee that array is divided into equal half.



$2T(\frac{n}{2}) + n$

(merge sort of equal division)

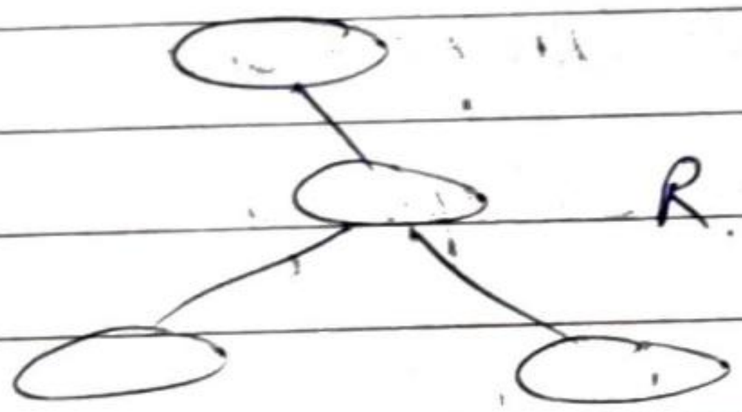


0 to 9

−1 +1

LHS RHS

This is merge sort, binary

→ 

By taking pivot as biggest element, it is possible that it is in right side.

WORST CASE OF QUICKSORT-

$T(0) + T(n-1) + n$

Master theorem is $\frac{aT(n)}{b} + f(n)$ $b \geq 1$

↓

not possible by this

It is possible that it does not have element on left side as we have ~~all~~ selected right side element as pivot.

$T(0) + T(n-1) + n$

8- Consider the following inputs —

a: 1, 2, 3, — — — — — n-1, n (ascending)
b: n, n-1, n-2 — — — — — 1 (descending)

Let c_1 and c_2 are no. of comparisions make for the input a and b. To arrange the above elements in ascending order using quick sort what will be the relation b/w c_1 and c_2.

(a) $c_1 < c_2$ (b) $c_1 == c_2$ (c) $c_1 > c_2$

Comparison —

Best case of c_1 is $(n-1)n$
Best case of c_2 is $(n-1)n$

$c_1 \rightarrow (n-1)n \rightarrow O(n^2) - n$

$c_2 \rightarrow (n-1)n \rightarrow O(n^2) - n$

↓
as we have n element as n elements are comparing with each and moving in descending order.

So complexity is n^2
$c_1 == c_2$
option (b) is correct.

In case of swap $c_2 > c_2$ as we have to swap and c_2 in descending order.

Ascending in best case of c_1 swap → $O(1)$
Descending in " " " c_2 → $O(n^2)$

__/__/____

When assay is sorted the time complexity is same, it will not decrease as we need to go to check if comparison do same.

complexity for swap :-

Worst case will be of descending order

$c_2 \longrightarrow$ ie $O(n^2)$

Max- Min Technique — (without divide & conquer)

Straight forward -

```
void straightMaxMin (Type a[], int n, maxT)
{
  max = min = a[i];
  for (int i=2 , i<=n , i++)
  {
    if (a[i] > max)  max = a[i];
    if a[i] < min  = min = a[i]
  }
}
```

It is doing 2(n-1) comparisons to find max-min

eg - 3 9 11 1 7 6

2 (n-1)
2 (6-1) = 2(5) = 10

min = 3
max = 11

when index = 1 , 3<9
3<11
3>1

3<7
3<6

When index = 2, $9 > 3$
$9 < 11$
$9 > 1$
$9 > 7$.
$9 > 6$

i=2, we will start comparing from 9 as i=2.

3 9 11 1 7 6

max = 11, min = 1

max = min = a[1] (let = 3)

$3 < 9$		$9 > 3$ –
$3 < 11$	min=1	$9 < 11$
$3 > 1$ –	max=11	$9 > 1$ –
$3 < 7$		$9 > 7$ –
$3 < 6$		$9 > 6$ –

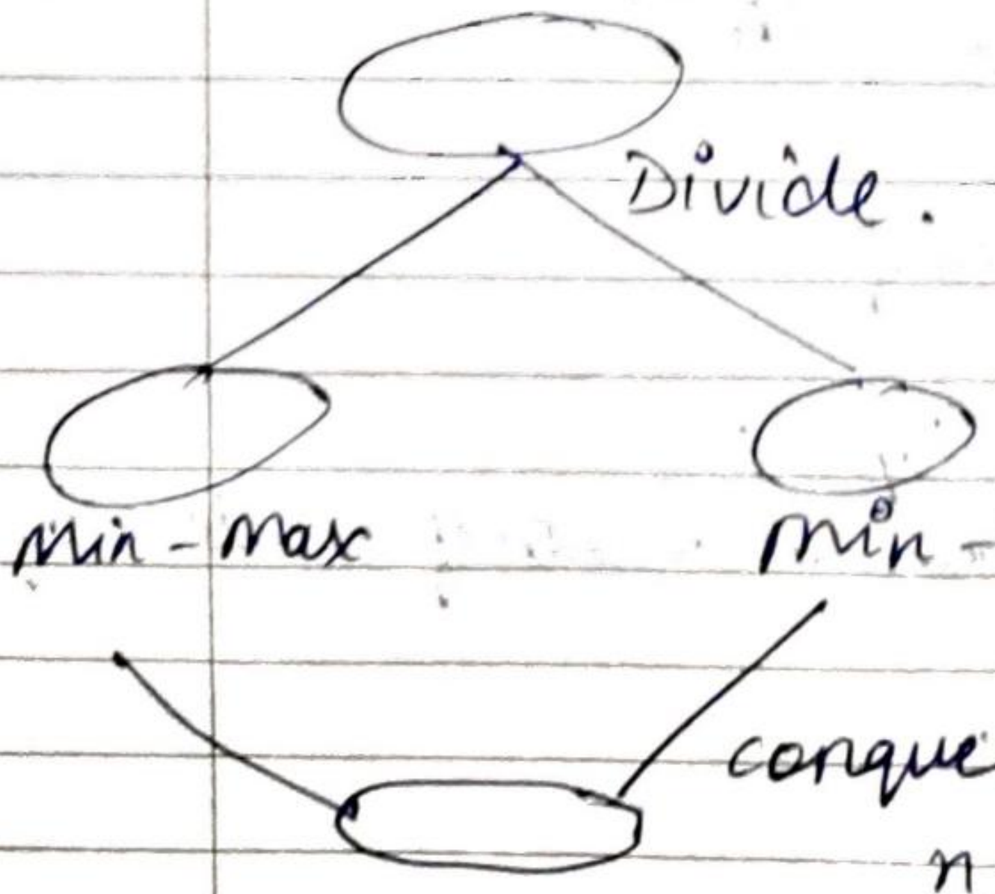
5 max, 5 min

Worst case: O(n)

Best case: O(n-1)

Avg. case $= \frac{2n}{2}$

Avg. case : O(n)



Av. Case

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

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

$O(n)$

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

Apply master theorem

first, traverse each and every element on the left side.

Imp & Explain the function calling using min-max.

Post order : traverse right
pre order : traverse left

→ $2T\left(\frac{n}{2}\right) + n$

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

$a \frac{T(n)}{b} + f(n)$

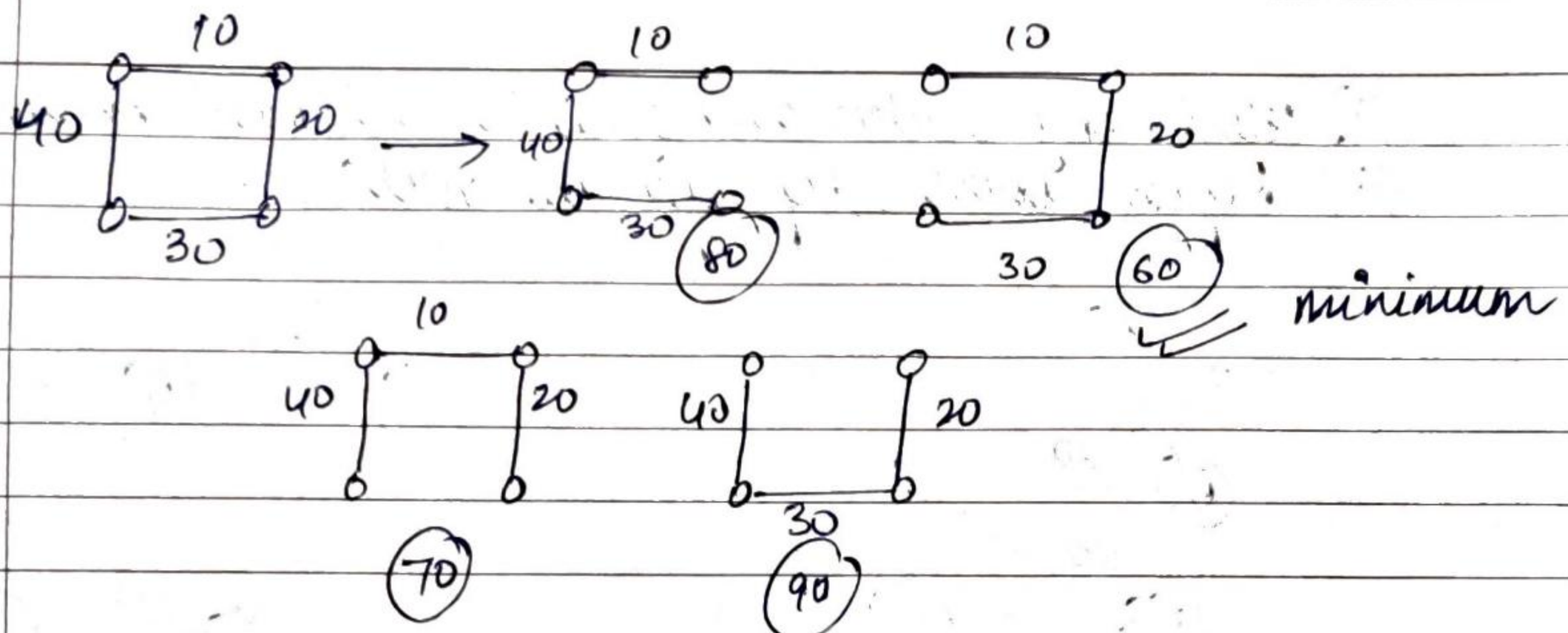
$T(n) = n^{\log_2^1}$

$T(n) \rightarrow \left(\frac{1}{2}\right) n^1$

→ Minimum Cost Spanning Tree- (MCST)

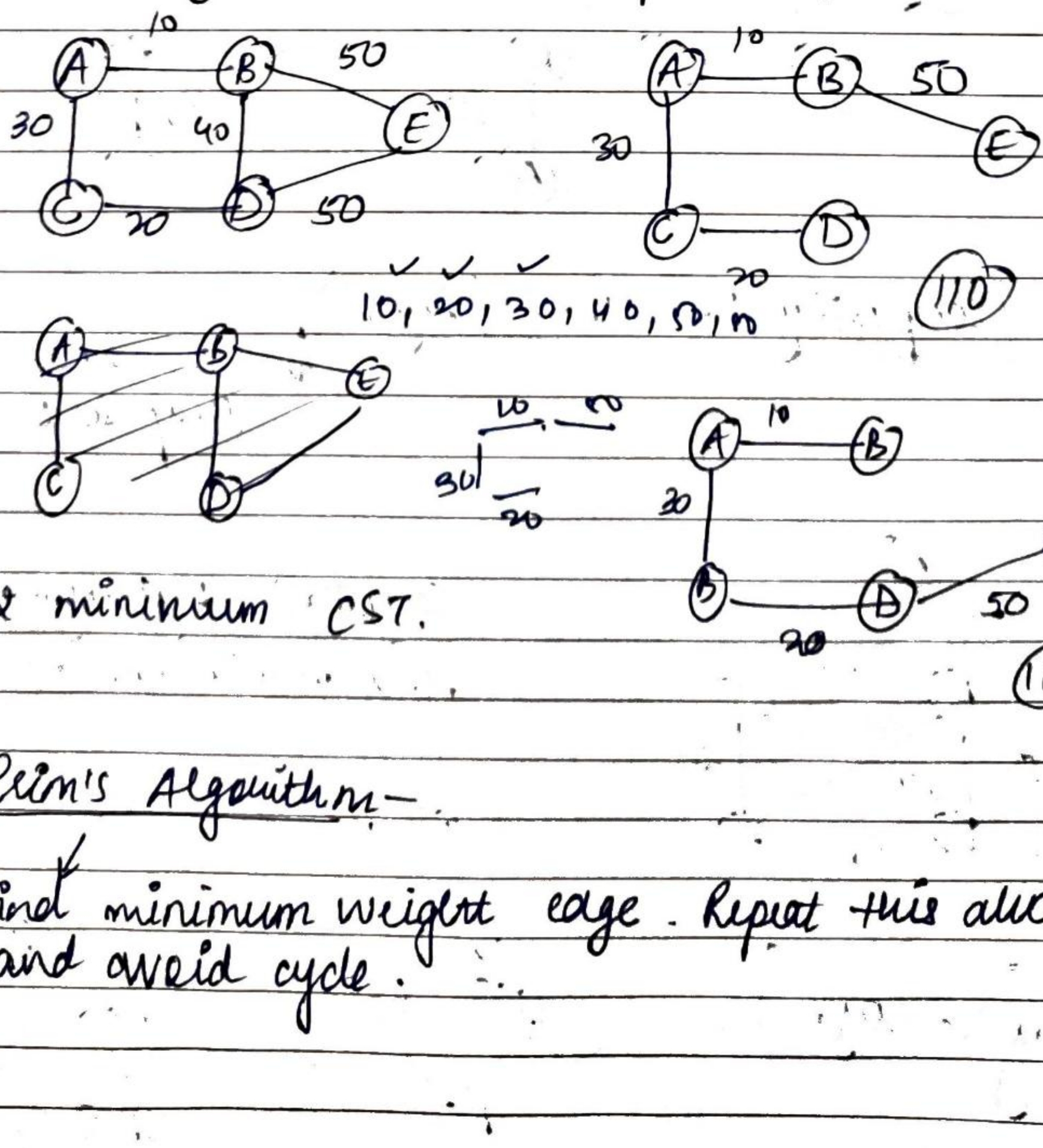
A tree is a spanning tree if -

(a) contain all vertices of graph
(b) should be $n-1$ edges ; where n is no. of vertex of graph.
(c) doesnot contain cycle.



(i) Prim's Algorithm
(ii) Kruskal's Theorem

Q- How many minimum CST possible?

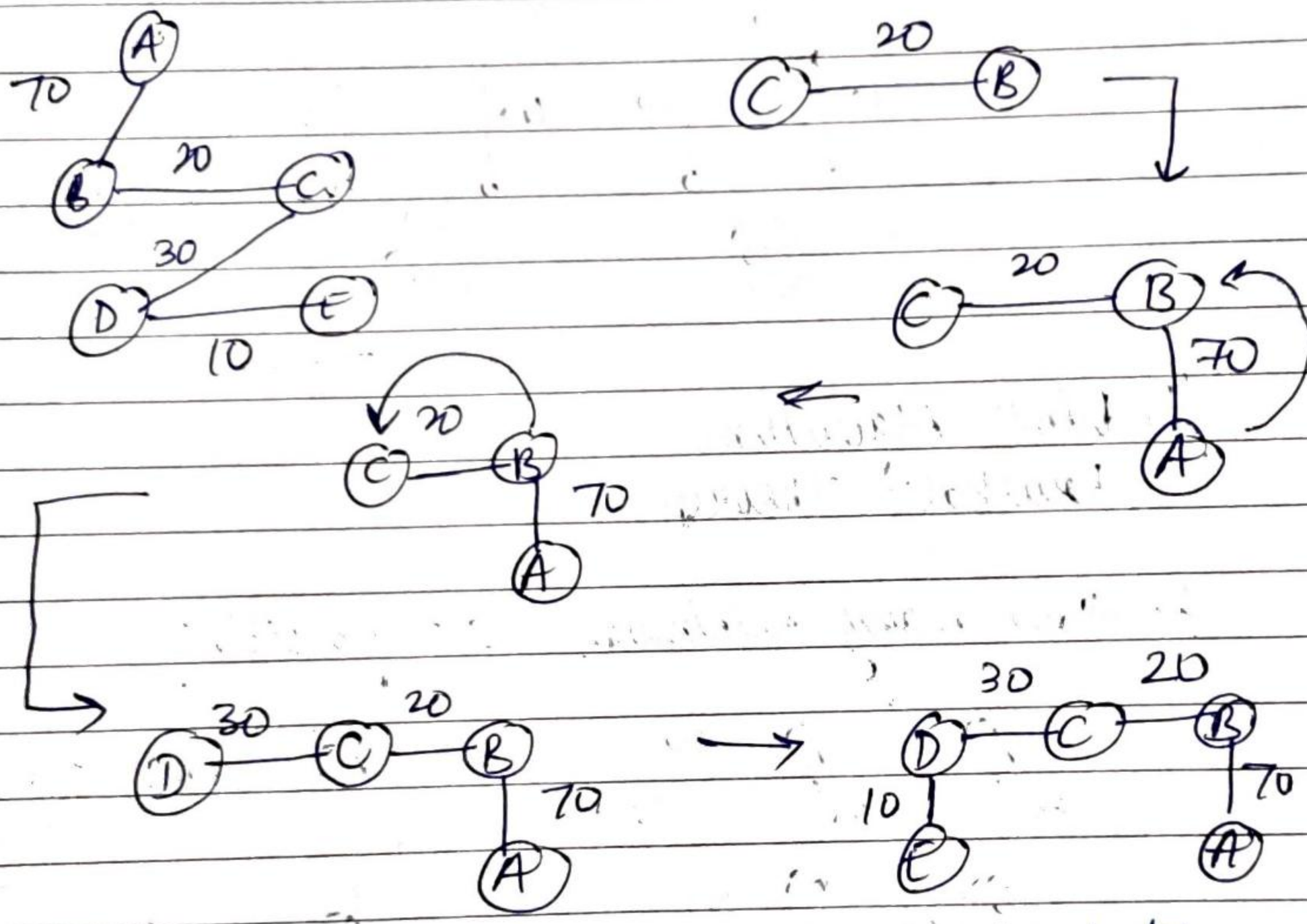


& minimum CST.

⇒ Prim's Algorithm -

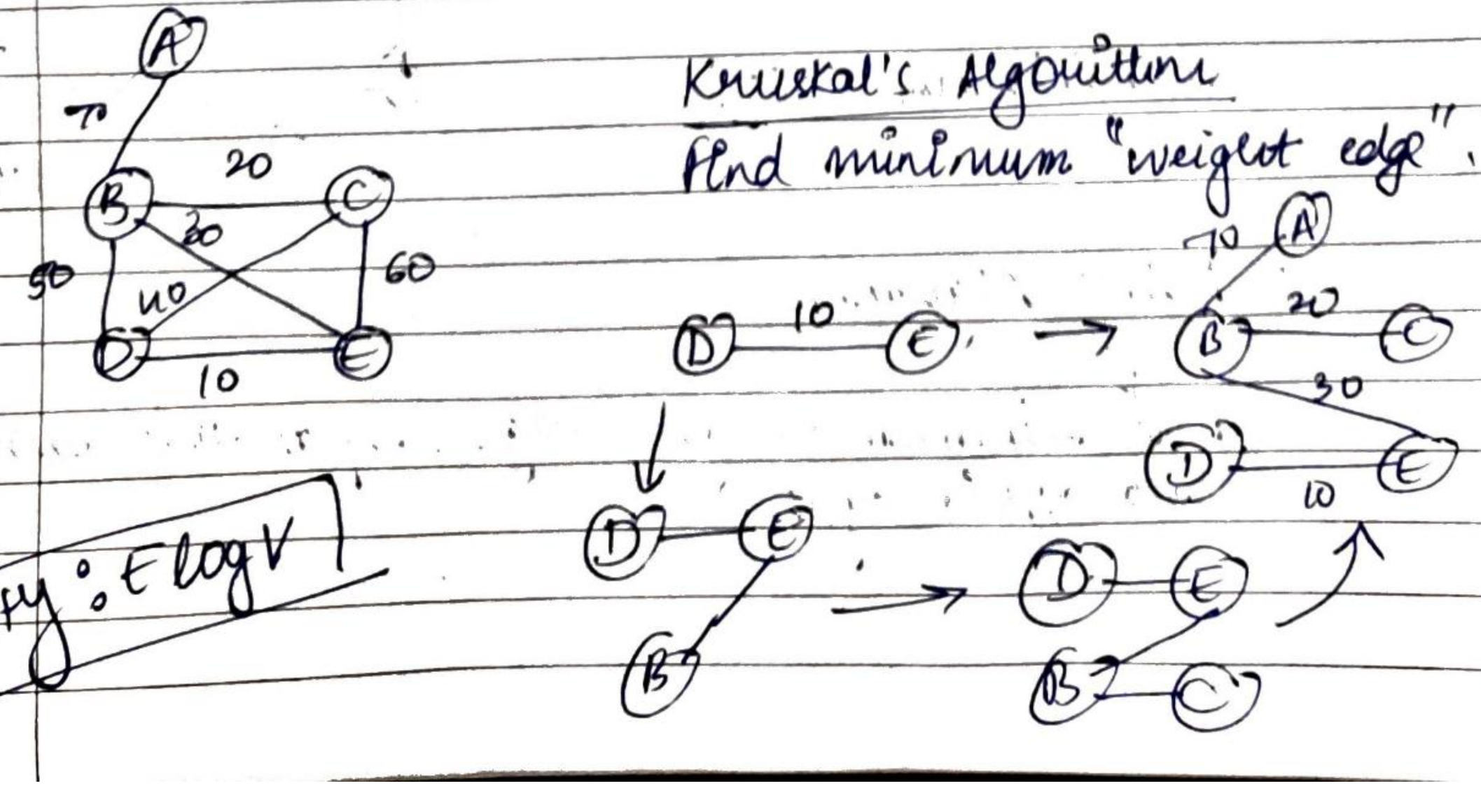
Find minimum weight edge. Repeat this always and avoid cycle.

(i) Select any vertex and find its adjacent minimum.
(ii) Repeat above step for all vertices.



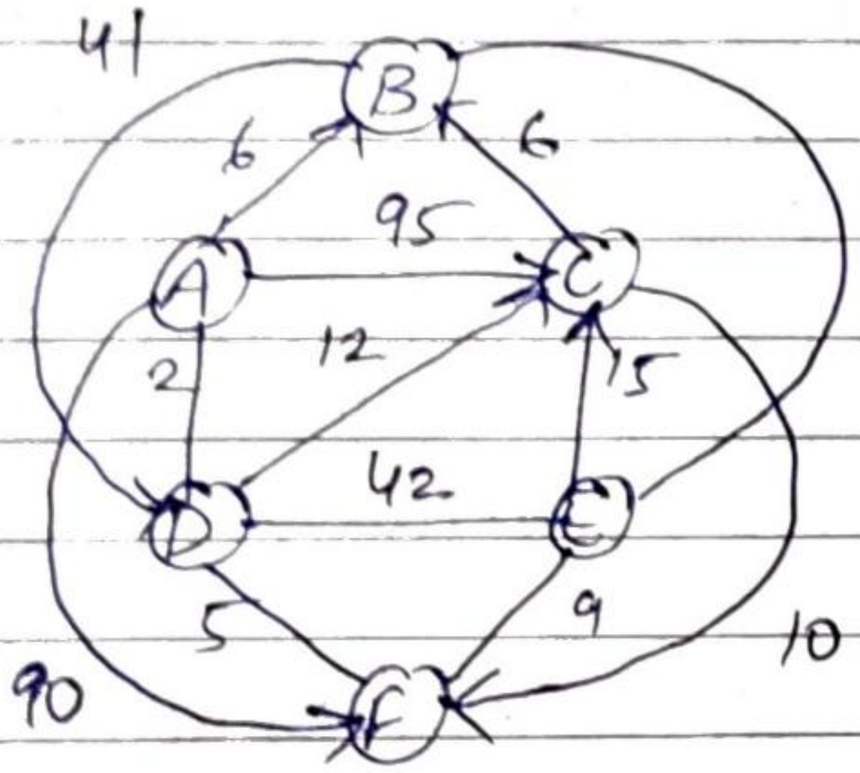
Complexity :- $V + E(\log V)$ → Red black tree
(on the basis of heap sort)

Ex-



Single source shortest path (Djkstra's theorem)

↓
It works on the edge only.



Path:

A → B → D → C → F → E

6 41 12 10 9

= 78

(i) All +ve edge height.

(ii) may gave wrong answer when -ve edge weight or -ve edge weight cycle.

Complexity: $O(V+E) \log V$.

Greedy Algorithm -

General method:

```
Greedy (A, n)
soln ← ϕ
for i ← 1 to n
do x ← select A
feasible (soln ; x) then
soln union (soln, x)
return soln.
```

The difference between greedy & dynamic algorithm is that greedy algorithm looks for locally optimal solution and assume it as best but they

__/__/____

don't always yield the optimal solution.

Note: All of the greedy problem have n inputs & require to obtain a subset that satisfy some constraints. Any subset that satisfy these constraints is called feasible solution. In order to find the feasible solution. We can have min or max-value of objective function.

A greedy algorithm selects a input with the heap of select function. 1. check for its feasibility constraints. If it is feasible then it is added to the solution. The func. union combines x with solution & updates the objective function.

→ Greedy 1st technique:

Knapsack problem :- There are n objects & a knapsack or bag object i → n, i → wi and knapsack has capacity of m. If a ~~function~~ fraction xi, $0 \leq x \leq 1$ of object is 1 and is placed into the knapsack then profit pixi is earned.

$$\text{maximise} \quad \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to} \quad \sum_{1 \leq i \leq n} w_i x_i < m$$

& $0 \leq x_i \leq 1, \quad 1 \leq i \leq n$

Example 1: no. of objects $n = 3$

Total capacity of bag $m = 20$

	obj 1	obj 2	obj 3
Profit (P_i)	25	24	15
Weight (w_i)	18	15	10
P_i / w_i	1.4	1.6	1.5

without w_i, actual profit can't be found.

2015

(i) division P_i/w_i
(ii) max P_i/w_i weight putin
(iii) ans. max 1.5

when weight of object is more than need

$$= \left(\frac{\text{need}}{\text{total}} \times \text{profit}\right)$$

$$= \frac{5}{10} \times 15 = 7.5$$

$$= 24 + 7.5 = 31.5$$

$n \log n$
↓
complexity

Example 2: no. of objects = 9
Total capacity $m = 40$

Object	1	2	3	4	5	6	7	8	9
Profit	25	77	99	66	55	11	40	80	60
Weight	2	11	~~9~~	10	8	2	3	10	8
P_i/w_i	~~12.5~~	7	~~11~~	6.6	6.8	5.5	13.3 ↑	8	7.5

Find max profit ?

40 | 31 | 35 | 26 | 16 | 8

$$= \frac{8}{10} \times 77 = 56$$

__/__/____

Unit-3
Dynamic Programming

→ Principal of optimality : It is an algorithm designed that can be used when the solution to a problem can be viewed as a result of sequences.

→ This principle states that optimal sequence of decision has a property that whatever the critical state and decision are the remaining decision must constitute and optimal with regard to the state resulting from first decision.

↓

to reach goal must follow a sequence & that sequence must give optimal result.

→ All pairs shortest path :

Floyd - Warshall algorithm →

Compute the value dij (K) in order of increasing value of K

```
n ← row [W]
D° ← W
for k ← 1 to n
do
   for i ← 1 to n
   do for j ← 1 to n
```

$d_{ij}^{k} \leftarrow \min\left(d_{ij}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}\right)$

↓ This will work only 1 time.

complexity : $O(n^3)$

⇒ Imp Ques -

1) Explain asymptotic notations.
2) Illustrate the time complexity of an algorithm.
3) Illustrate the principle of optimality.
4) Evaluate the efficiency of binary search.
5) Examine the complexity of quick sort and merge sort.
   ↓ we have to give recurrence relation for time complexity.
   (not to mention working of algorithms)
6) Compare Dijkstra algorithm & Bellman ford.
7) Compare prim's algorithm & Kruskal's algo.
   ↓ (just in terms of complexity)
8) Discuss with example of Greedy algo with knapsack method.
9) How dynamic programming is used to solve 0/1 knapsack problem?

Greedy	Dynamic.
knapsack / fractional	0/1 knapsack.

10) Explain how dynamic programming is used by Floyd Marshall in terms of complexity.

__/__/____

11) What is complexity analysis of an algorithm?
↓
(Explain ~~comp~~ classes of complexity)

⇒ Diff. b/w Greedy & Dynamic.

Greedy	Dynamic
1.) It check less possibilities.	1.) It will check all possibilities.
2.) Less time	2.) More time.
3.) Sometimes give wrong result.	3.) always correct because it checks all possibilities.

⇒ Diff. b/w Dijkstra and Bellman Ford.

Dijkstra	Bellman
1.) This work on +ve edge only.	1.) This work on +ve and -ve edge.
2.) Complexity: $O((V+E)\log V)$	2.) complexity: $O(n^2)$ [simple graph] $O(n^3)$ (complete graph)
3.) If we work -ve edge, it will give incorrect output and solution for that is Bellman ford algo.	3.)

→ **Multistage Graph**
(Dynamic Programming)

source	sink
Only outgoing	Only incoming flow

forward Approach –

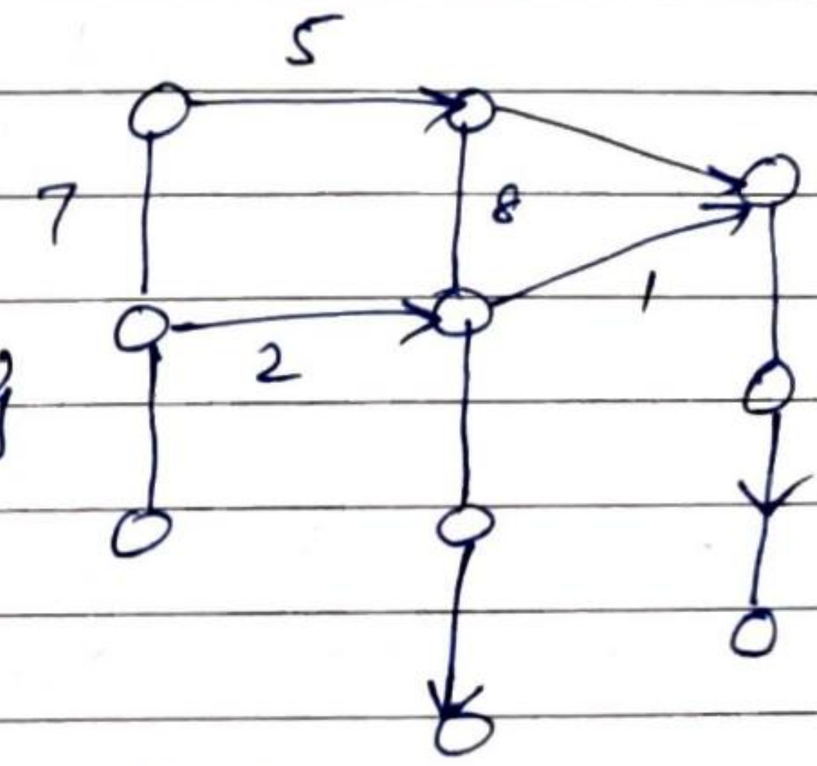
$$cost(i,j) = \min \{ c(\underbrace{j,l}_{edge}) + cost(\underbrace{i+1}_{\text{next stage}}, l) \}$$

↑ stage

backward Approach –

$$bcost(i,j) = \min \{ bcost(\underbrace{i-1}_{\text{previous stage}}, l) + c(\underbrace{l,j}_{edge}) \}$$

↑ stage



← Stage

Time complexity → $O(V+E)$ for 2
↓ ↓
vertices edges

- sink may or may not be a destination. }} X

+

→ **Bellman Ford**

Initialize single source (G, s) — O(1)

Within same for loop:
for i ← 1 to |V[G]| − 1 → V
do for each edge (U, V) ∈ E[G] → E
do Relax (U, V, W) ⇒ O(VE)
for each edge (U, V) ∈ E[G] → E

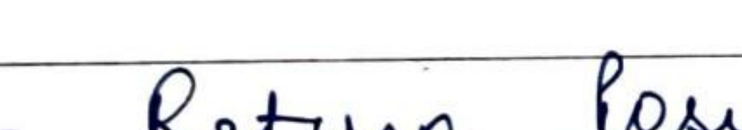
(8) 1 4 5 9 ~~9~~ 1

↑ → Print → Pos^n ___/___/___

check -ve edge

[do if d[v] > d[u] + w[u,w] → V^th
then return false
return true

~~O(E+V^2)~~



complexity : O(VE)

9

---- * * * * ----

Binary Search ⟷ divide and Conquer

I/P: A Sorted array → 'n' elements
and element x

O/P :- Return position (location) of x
if found.

array → A [10, 20, 30, 40, 50, 60, 70]
start → i j ← end

BS (A, i, j, x)

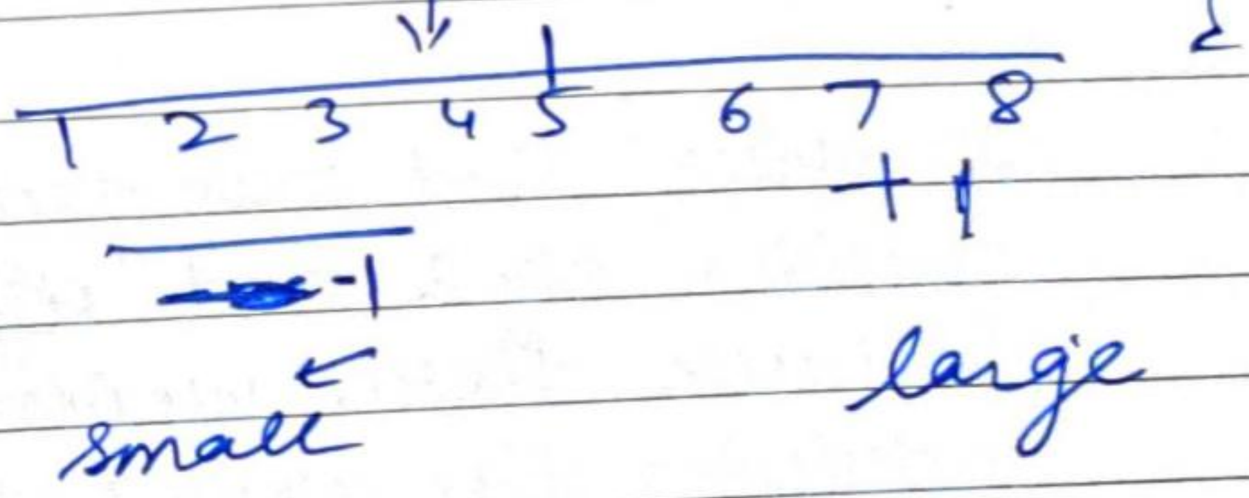
O(1) if (i == j) → means only 1 element

then check if (a[i] == x)
return x.

else

mid $\left[\frac{i+j}{2}\right]$;

```
if a[mid] == x
        return mid;
else
        if a[mid] < x
    BS (a, mid+1, j, x);
else
    BS (a, i, mid-1, -x);
```



Analysis

WC

$n, \frac{n}{2}, \frac{n}{4}$ — —

$\frac{n}{2^R} = 1 \longrightarrow$ (Coz we have 1 element)

$n = 2^R$

Take log on both sides

$\log n = k \log 2$

$\log n = \log n = k$

BC O(1) — (mid == x } i == j)

WC → O(log n)

BC → O(log n)

__/__/____

0/1 Knapsack

$n = 3$; $m = 15kg$

	obj 1	obj 2	obj 3
Profit	60	35	40
Weight	10	7	8

$m = 15$

In order find max profit;
consider obj 2 and obj 3;
(because there weight is satisfying the weight of Knapsack / bag)
Although max profit was 100; but the weight exceeds the weight of bag.

Combinations

$10+7 = 17$ X
$10+8 = 18$ X
$7+5 = 15$

Profit → obj 2 + obj 3
$= 35 + 40$

Profit $= 75$

obj 1 obj 2 obj 3
{ 0 , 1 , 1 }
not pick pick pick

Complexity → $O(nm)$ → because we check possibilities
or $O(2^n)$