Date : / /20

The first problem : Stable matching

formulating the problem :- There are 'n' applicants, 'n' companies. Each of 'n' applicants applies to each of 'n' companies. Each company wants to accept single appli. (one to one mapping).

Two sets :

1) companies ⇒ $C = \{c_1, c_2 \ldots, c_n\}$.
2) Applicants ⇒ $A = \{a_1, a_2 \ldots, a_n\}$.

* Matching :- A matching S is a set of ordered pair of the form (c, a) where $c \in C$ and $a \in A$ such that each member of C and each member of A appears in atmost one pair in S.

* Perfect matching :- A perfect matching S' is a matching with the property that each member of C and A appears in exactly one pair in S'.

We will assign preferences to both sets C and A.
Now,

1) does there exist a stable matching for every set of preference?
2) Given a set of preference list, can we efficiently construct a stable matching if there is one?

Page No.
Date : / /20

Dynamic programming :-

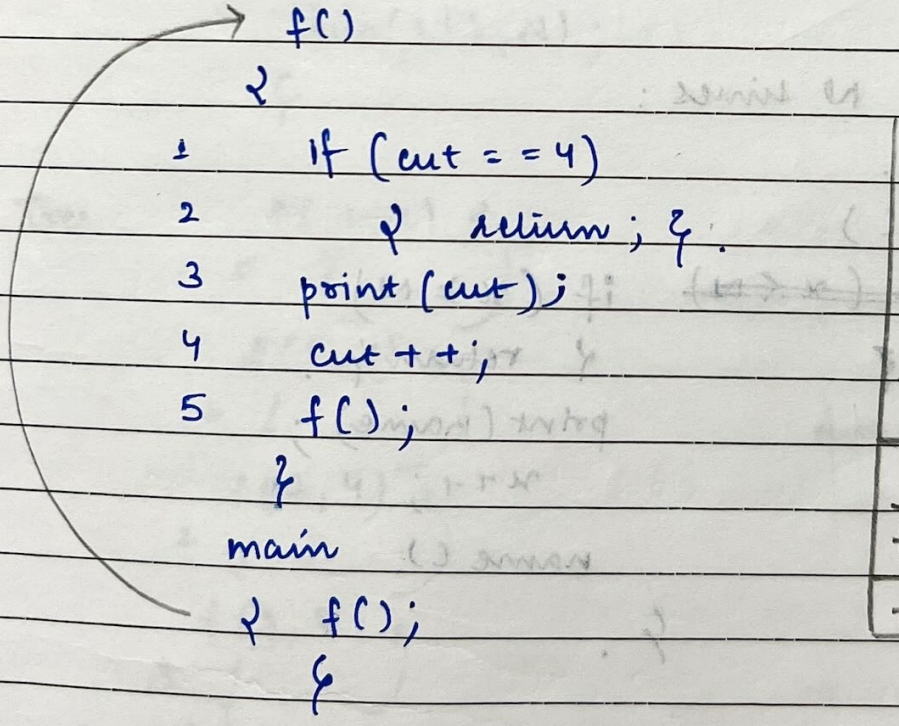
Those who can't remember the past are condemned to repeat it. ~DP.

Tabulation :- Bottom-up approach.
Memoization :- Top-down approach.
↳ store the values of sub-problems in map / table.

Recursion when a function calls itself.
(until a pre-specified condition is met).

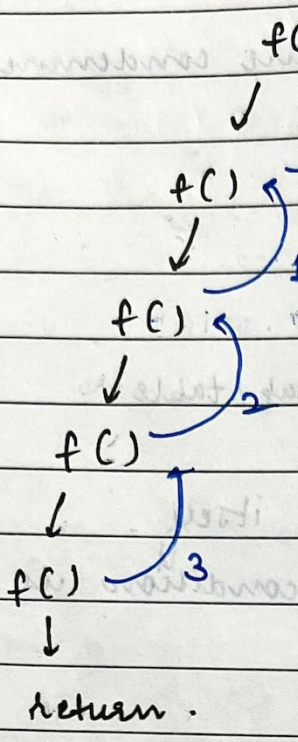
Eg:

```
cut = 0;
f()
{
1   if (cut == 4)
2       { return; }.
3   print(cut);
4   cut++;
5   f();
}
main
{ f();
}
```



output = 0 1 2 3.

Recursion Tree :



output :
0 1 2 3.

Q Print name N times :

```
int x = 0.
name ()
{   if (x < N)
    {
    }
}
```

```
if (x >= N)
{ return 0;
print (name);
x++;
name ()
}.
```

```
main()
{  name ()
{.
```

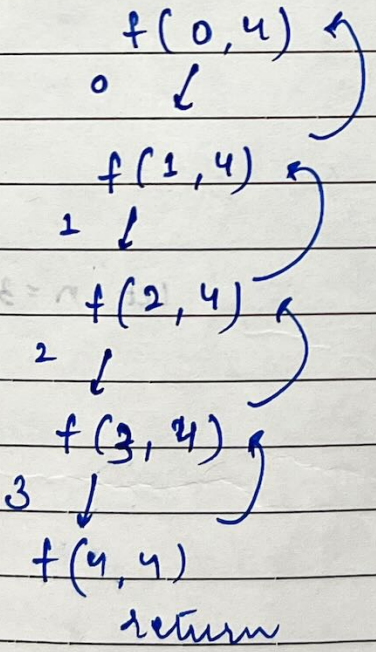
Q print 0 to N-1

```
main()
{
    int N;
    cin >> N;
    int i = 0
    fun(0, N).
}
fun(i, N)
{
    if (i >= N)
        { return; }
    print(i);
    fun(i+1, N);
}
```

let N = 4.

to print : 0 to 3.

Tree :



output : 0 1 2 3.

Page No.
Date : / /20

Fibonacci :

0 1 1 2 3 5 8 13 21 34 55

$f(n) = f(n-1) + f(n-2)$. → recurrence relation.

$f(0) = 0$

$f(1) = 1$

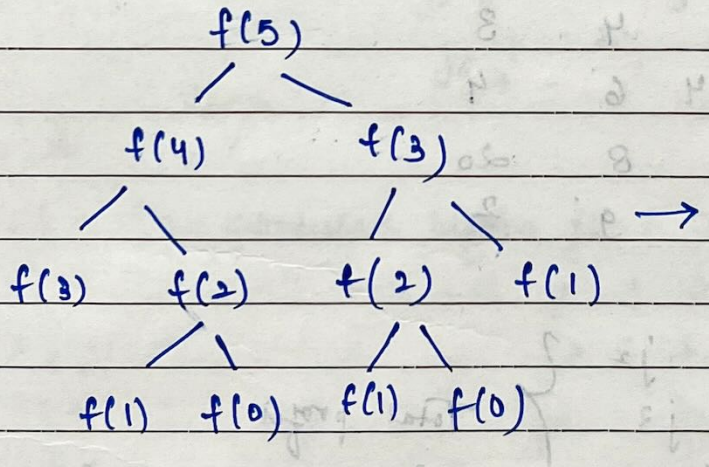
$f(2) = f(0) + f(1)$

$= 0 + 1 = 1$.

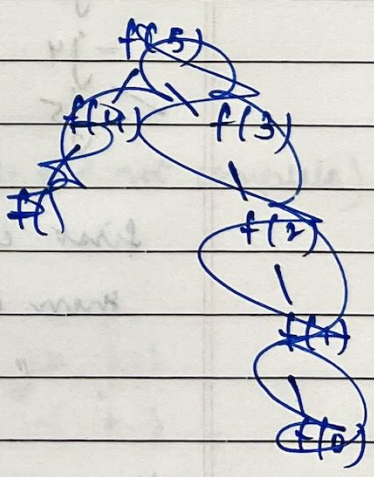
$f(3) = f(2) + f(1)$

$= 1 + 1 = 2$.

```
f(n)
{   if (n<=1)
      return n;
    return f(n-1) + f(n-2);
}
```



recursion tree.



Overlapping subproblems

condition for DP : ① larger problem can be solved using smaller problem

② there must be overlapping subproblems.

Weighted Job scheduling problem.

Problem statement :-

finding optimal schedule of compatible jobs that earns the maximum profit.

Given a set of jobs described by :

s_i : starting time of i^{th} job

f_i : finishing time of i^{th} job

p_i : profit of scheduling i^{th} job

Two jobs are said to be compatible if they do not overlap in time.

Greedy solution approach :-

choose the job that finishes first.

Job	s_i	f_i	p_i
j_1	1	5	10
✓ j_2	2	4	3
✓ j_3	4	6	4
j_4	5	8	20
✓ j_5	6	9	2

first chooses j_2
then choose j_3
" " j_5

Total profit = 3+4+2 = 9.

If we choose j_1 and then j_4 &, we obtain the profit as 10+20 = 30.

Thus, greedy is not optimal.

Recursive solution approach:

Order the jobs in increasing order of their finishing time.

Take decision about the 'n^th' job:

what choices optimal has?

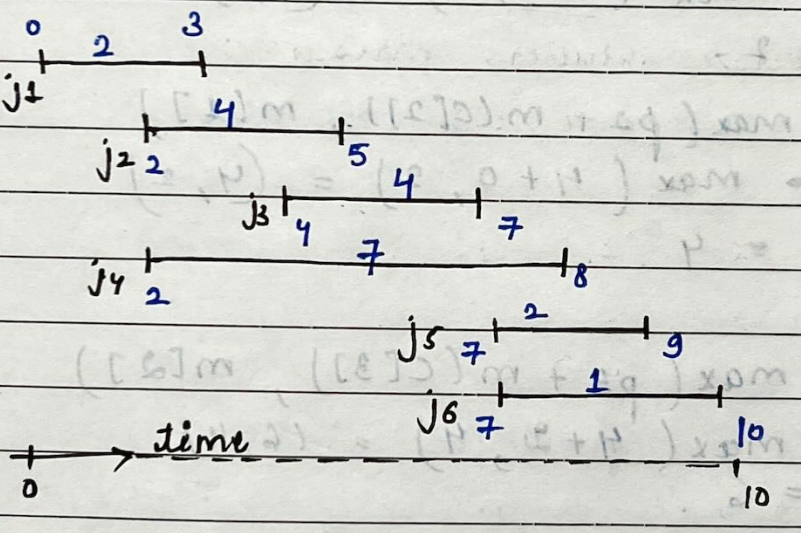
either to choose 'n' or not.

calculate profit in each case.

Let C[j] = the rightmost interval before j that is compatible with j (finishes before j).

C[j] = 0 if there is no such job.

Eg:



C[1] = jobs scheduled before j1 = 0.

C[2] = " " " j2 = 0 (j1 did not complete)

C[3] = " " " j3 = 1 (j1)

C[4] = " " " j4 = 0

C[5] = " " " j5 = 3 (j1, j2, j3)

C[6] = " " " j6 = 3 (" " ")

Profit in either case?

1) When n^th job is not scheduled? opt[n] = opt[n-1].

↓ profit of optimal sched. from first 'n' jobs.

2) when n^{th} job is scheduled :-

$$opt[n] = p_n + opt[c(n)]$$

Thus, $opt[n] = \max(p_n + opt[c(n)], opt[n-1])$

In our example taken before,

	0	1	2	3	4	5	6
p_j	0	2	4	4	7	2	1
j	0	1	2	3	4	5	6
$c[j]$	0	0	0	1	0	3	3
$m[j]$	0	2	4	6	7	8	8

$m[1] = \max(p_1 + m(c[1]), m[0])$

$= \max(2 + m[0], m[0])$

$= \max(2+0, 0) = (2, 0)$

$= 2$

$m[2] = \max(p_2 + m(c[2]), m[1])$

$= \max(4+0, 2) = (4, 2)$

$= 4$.

$m[3] = \max(p_3 + m(c[3]), m[2])$

$= \max(4+2, 4) = (6, 4)$

$= 6$.

$m[4] = \max(p_4 + m(c[4]), m[3])$

$= \max(7 + m[0], 6) = (7, 6)$

$= 7$.

$m[5] = \max(p_5 + m(c[5]), m[4])$

$= \max(2 + m[3], m[4]) = (2+6, 7) = (8, 7)$

$= 8$

$m[6] = \max(p_6 + m(c[6]), m[5])$

$= \max(1 + m[3], 8) = (1+6, 8)$

$= (7, 8)$

Date : / /20

Running time :

Number of cells in the Table = 'n'.

Time to compute each cell = k (constant)

Total time = $O(kn)$

$= O(n)$

Date : / /20

Matrix chain multiplication :-

Tabulation method.

	1	2	3	4
1	0	5785	1530	
2		0	1335	
3			0	9078
4				0

A · B · C · D

⇓ ⇓ ⇓ ⇓

13x5 5x89 89x3 3x34

A B C D

Step 1: m[1,1] m[2,2] m[3,3] m[4,4]

0 0 0 0

Step 2: m[1,2] m[2,3] m[3,4] ⇒ multiplying 2 matrices.

A·B B·C C·D

13x5, 5x89 5x89, 89x3 89x3, 3x34

= 13x5x89 = 5x89x3 = 89x3x34

= 5785 = 1335 = 9078

Step 3: m[1,3] ⇒ m[1,②,3]

A·(B·C) or (A·B)·C

13x5,(5x89,89x3) (m[1,2])·m[3,3]

m[1,1] + m[2,3] ⇒ m[3,3] + m[1,2]

+ 13x5x3 + 13x89x3

= 0 + 1335 + 195 = 0 + 5785 + 3471

= 1530 = 9256

min(1530, 9256)

= 1530

Given a chain $\langle A_1, A_2 \ldots A_n \rangle$ of 'n' matrices where for $i=1$ to n matrix A_i has dimension $(d_{i-1} \times d_i)$ resp., fully parenthesise the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ in a way that minimizes the number of scalar multiplication

Two matrices of size $m*n$ and $n*p$ when multiplied, generate a matrix of size $m*p$.
Number of multiplications performed = $m*n*p$.

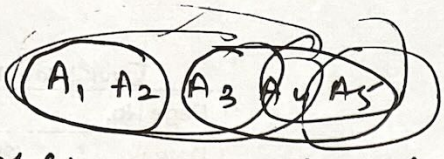
Input :- 'n' matrices : $A_1, A_2 \ldots A_n$
order $(d_1 d_2), (d_2 d_3), \ldots (d_n, d_{n+1})$ resp.
Aim :- order in which the matrices should be multiplied so as to minimise number of multiplication.
choices optimal has for the split point k :

$(A_1 A_2 \ldots A_{k-1})(A_k \ldots A_n)$.

$1 \le k \le n-1$

$k=1$: $(A_1)(A_2 \ldots A_n)$
$k=2$: $(A_1 A_2)(A_3 \ldots A_n)$
$k=n-1$: $(A_1 \ldots A_{n-1})(A_n)$

Let Opt $(1,n)$ denote the minimum number of multiplications required to multiply : $A_1 A_2 \ldots A_n$.
k be the split point, then minimum number
$= m_1 + m_2 + d_1 d_{k+1} d_{n+1}$
m_1 = min no. of mult. required to multiply $A_1 A_2 \ldots A_k$ recursively
m_2 = min no. of mult. required to multiply $A_{k+1} \ldots A_n$ recursively



opt(1,2) opt(3,5).

Evergreen
Page No.
Date : / /20

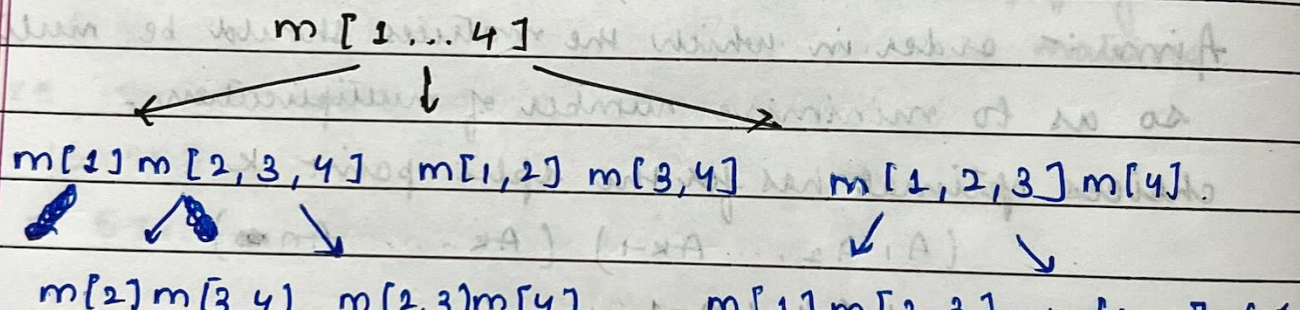
opt(1,n): $\min_k \{ m_1 + m_2 + d_1 d_{k+1} d_{n+1} \}$

$= \min_k \{ opt(1,k) + opt(k+1,n) + d_1 d_{k+1} d_{n+1} \}$

Generalising,

$opt(i,j) = \min_k \{ opt(i,k) + opt(k+1,j) + d_i d_{k+1} d_{j+1} \}$.

Eg: overlapping subproblems:



Number of subproblems $\leq n^2$

Running time = $O(n^2)$ entries in table.
$O(n)$ to compute each.
Total time = $O(n * n^2) = O(n^3)$.

Segmented least square :-

Given a set P of 'n' points in a plane denoted by (x_1, y_1) (x_2, y_2) .. (x_n, y_n), we want to determine a line that fits these points the best.

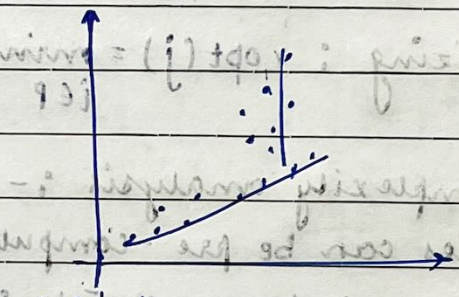
Determine a line ? $ax + b = y$ → eqn of line.

determine a and b

Line of best fit : line that minimizes Error.

$$Error(L, P) = \sum_{i \in P} (y_i - b - ax_i)^2$$

But always using a single line is not a good idea. like in the situation :



Two lines giving better coverage.

But each additional line comes with cost.

Aim :- How many line segments to use so as to minimise the sum of total cost of lines and the error of approximating the points by these line segments.

The Recurrence :

Let p_1 p_2 p_3 .... p_n be the points in increasing order of their x-coordinates.

Let opt(n) be the minimum cost of interpolating the points $p_1, p_2, \ldots, p_n$.

Page No.
Date : / /20

Guess the left end point of the last line segment in the optimal say pi.
The last segment is pi, ..., pn then the values of optimal is :

$$opt(n) = opt(i-1) + c + e_{in}.$$

c = cost of using the segment.
e_{in} = least square error of interpolating points pi, .., pn by the best line segment.

$$opt(n) = \min_{i \in P} \{ opt(i-1) + c + e_{in} \}.$$

Generalizing : $opt(j) = \min_{i \in P} \{ opt(i-1) + c + e_{ij} \}$.

Time complexity analysis :-

1) e_{ij} values can be pre-computed :
total values to be computed : $^nC_2 = \frac{n!}{(n-2)!\,2!} = \frac{n(n-1)}{2}$

$= O(n^2)$.

time to compute each entry = $O(n)$.
Total time = $O(n * n^2) = O(n^3)$.

2) Having pre-computed e_{ij} for i and j,
Number of cells / entries in table opt ()
$= O(n)$.

time for computing each cell
$= O(n)$.

Total = $O(n*n) = O(n^2)$.

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

upper bound = $O(n^3)$.

Date : / /20

Knapsack problem :

→ Fractional knapsack problem :-

Aim : pick items with maximum total value but with weight at most W. (may choose fraction of items).

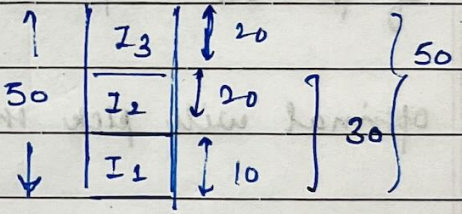
Greedy approach : Pick the items in the decreasing order of value per unit weight. i.e. highest first.

Eg :

Item	Value	weight	Value / weight
1	60	10	6
2	100	20	5
3	120	30	4

Let W = 50.

item 1 ⇒ weight = 10



item 2 ⇒ W = 20

item 3 ⇒ W = 30 but only 20 can be included.

values = $60 + 100 + \frac{20}{30} \times 120$ = $60 + 100 + 80$

= 240 Ans.

~~But not optimal.~~

~~If we took item 3 then item 1~~

~~10 + 60 ⇒ 180~~

Recursive approach :

0/1 : leave it or pick it :

Arrange the elements in any arbitrary order :

$x_1 \; x_2 \; \ldots \; x_n$

Suppose we know $w \rightarrow$ exact weight picked by optimal.

what choices optimal has for x_n object ?

Pick it or leave it.

what is the value of objective fn ?

let $\boxed{opt(n, w)}$ denote the value of optimal soln with total weight exactly equal to w.

1) If not picked : $opt(n, w) = opt(n-1, w)$
2) If picked : $opt(n, w) = opt(n-1, w - w_n) + v_n$.

optimal will pick the maximum :

$$opt(n, w) = \max \{ opt(n-1, w - w_n) + v_n \; , \; opt(n-1, w) \}.$$

↳ if $w_i \leq w$

Generalising :

$$opt(i, w) = \max \{ opt(i-1, w - w_i) + v_i \; , \; opt(i-1, w) \}.$$

final recurrance :

$opt[0, w] = 0$

$opt[i, 0] = 0$

$opt[i, w] = opt[i-1, w]$ if $w_i > w$.

Eg :

Item	weight	value
1	2	3
2	3	4
3	4	5
4	5	6

$W = 5$

$opt[1] = \max\{opt(0, 5) + 3,\ opt(0, 5)\}$

$= \max\{0+3, 0\} \Rightarrow (3, 0)$

$= 3.$

$opt[2] = \max\{opt(1, 3) + 4,\ opt(1, 3)\}$

$= \max\{3+4, 3\} = \max(7, 3)$

$= 7.$

$opt[3] = \max\{opt(2, 0) + 5,\ opt(2)\}$

$= \max(5, 7)$

$= 7.$

$w = 0.$

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7.

$opt[0, -] = 0$ $\quad opt[-, 0] = 0$

$opt[1, 1] = m(opt(0, -) + V_1,\ opt[0, -])$

$= m(0+3, 0)$

$= (3, 0) \Rightarrow 3.$

Date :

Running time :

$n \times W$ cells

constant time to compute each cell.

Total time = $O(nW)$

pseudo polynomial time.

Sequence Alignment problem

Given two strings $x = x_1 x_2 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$, transform x into y with a minimum number of edit steps.

Edit steps : insert, delete or replace a character.

x_i = position of character $= i$ in string x.

Idea : check for last two elements.

case 1 $x_1 \; x_2 \; x_3$ → x string
$y_1 \; y_2 \; y_3$ → y string.

Align. if $x_3 = y_3$, cost $= 0$.
otherwise make $x_3 = y_3$.

case 2 $x_1 \; x_2 \; x_3$ → remove x_3.
$y_1 \; y_2 \; gap$

case 3 insert y_3 in gap.
$x_1 \; x_2 \; gap$
$y_1 \; y_2 \; y_3$

$Opt(i, j)$ = minimum number of edit steps to convert $x_1 x_2 \ldots x_m$ to $y_1 y_2 \ldots y_n$.

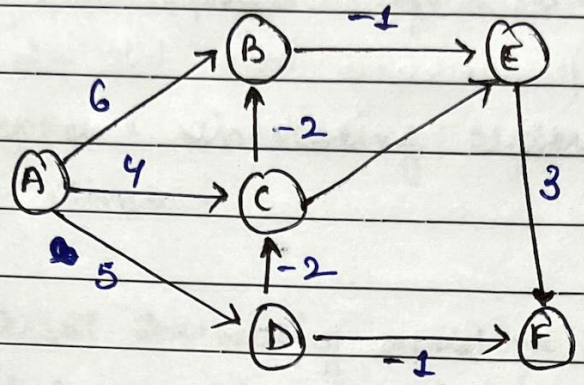
$$\min \begin{cases} \text{Align} = opt(i-1, j-1) + \delta_{ij} & \leftarrow \text{cost} \\ \text{Insert} = opt(i-1, j) \\ \text{Delete} = opt(i, j-1) \end{cases}$$

Date : / /20

Shortest Path problem with possible negative weights on edges.

: Dynamic approach.

Bellman - Ford Algorithm single source shortest path.



Go on relaxing all the edges (n-1) times.

n = no. of vertices.

if (d[u] + c(u,v) < d[v])

then d[v] = d[u] + c(u,v)

Problem statement :-

Given a directed graph G with weights, (possibly negative) on edges and vertices s and t, find shortest path from s to t. There are no negative weight cycles.

Four representative problems

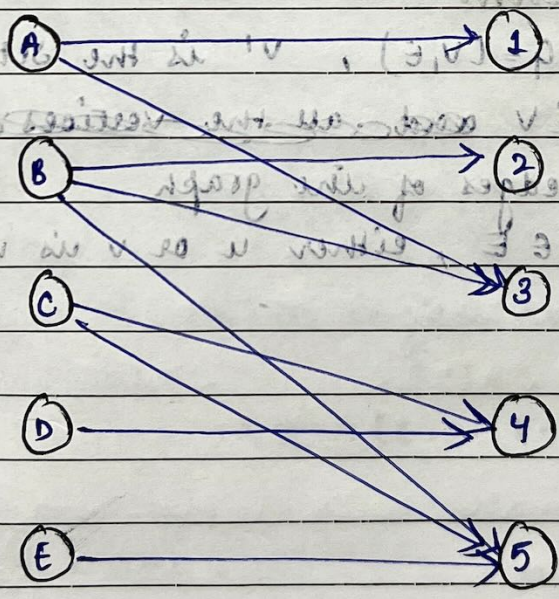
1) Interval scheduling problem : (max no. of meetings)
Two requests are said to be non-conflicting if they do not overlap : (one finishes before another starts).

Greedy approach : increasing order of their finishing time.

2) Weighted interval Scheduling problem : (max. profit)
Aim : Schedule meetings in a non-conflicting manner so as to maximise the profit ?

Greedy does not give optimal solution.

3) Maximum Bipartite matching problem :



Aim : Hire professors in such a way that we assign max subjects in such a way that one professor is assigned at most one subjects and vice versa :

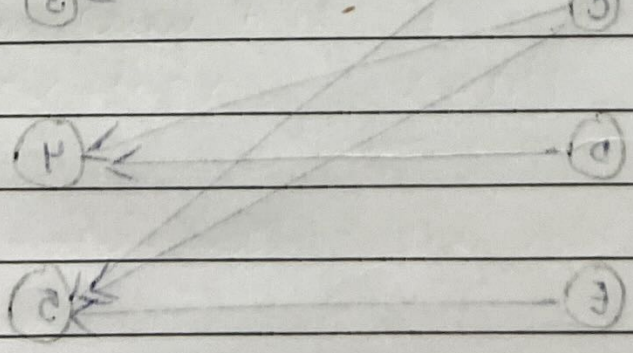
Augmentation :
Removing the infeasible edges.

Bipartite graph :
A graph G is said to be a bipartite graph if the vertex set V can be partitioned into two sets V_1 and V_2 such that no edge has both its end points in the same set.

Matching : Set of edges (matching set) M if no vertex has more than one edge from M incident on it.

Aim :- To find matching of max size.

4) Vertex cover problem
Given a graph $G = (V, E)$, V' is the set of vertices such that $V' \subseteq V$ and V' covers all the edges of the graph.
$\forall (u, v) \in E$, either u or v is in V'.



Correctness of linear search :

Input :- Array A[1...n], key

output :- index of first occurance of the key.
0 otherwise

```
i = 1
while i ≤ n do
    if A[i] == key then
        return i ;
    else i++
return 0 .
```

Step 1 : loop invariance

Property of algo that is satisfied in all iterations of the loop.

Step 2 : Prove loop inv. using Math induction

Step 3 : use loop inv. to prove correctness

Hypothesis H(r) : when control reaches while statement for r^{th} time,

$$k \notin A[1 \ldots r-1] .$$

$i = r$.

$$\forall \; 1 \le r \le n+1 .$$

Proof by induction on 'r' :

Base case : when $r = 1$, the claim holds true.

$A[1..0] \rightarrow$ empty.

Induction hypothesis :

$$H(m) \Rightarrow H(m+1) .$$

H(m) : k ∉ A[1 ... m-1] goes to while loop for m+1 only when control A[m] ≠ key.
which means key ∉ A[1 ... m].
therefore H(m+1) hold true.

Suppose that test condition in while is executed 'k' times.
loop runs for k-1 times.

(1) k ≤ n : A[k] = key
return k.
k ∉ A[1 ... k-1].

(2) k = n+1 : checks condition, condition becomes false.
returns
k ∉ A[1 ... n].

Correctness of binary search :-

Input : Give an array A[1 ... n] such that
A[1] < A[2] < ... < A[n], key x of

Output : terminates with index_mid = occurance of key in A if found.
index_mid = 0 otherwise.

Date : / /20

Loop invariance :

first ← 1 , last ← n .

while first ≤ last do

mid ← (first + last) / 2

if (A[mid] == key)

{ return mid }.

if (A[mid] < key)

{ last ← mid - 1 . }.

if (A[mid] < key)

{ first ← mid + 1 }.

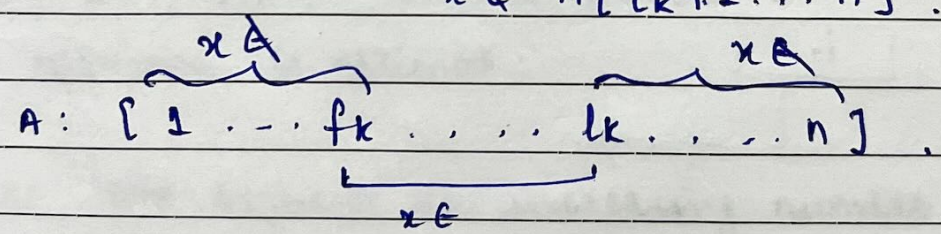
return 0

H(k) = when the control reaches "while" for the k^{th} time, let f_k and l_k be the first & last values resp.

and $m_k = \frac{f_k + l_k}{2}$

then we have : $x \notin A[1 \ldots f_k - 1]$.

$x \notin A[l_k + 1 \ldots n]$.



when k = 1 , : $f_1 = 1$ $l_1 = n$.

⇒ claim holds .

H(k) ⇒ k(k+1) .

* Greedy Algorithms :-

1) Interval scheduling problems :-

Problem statement :- Given a set of job requests $S = a_1\ a_2 \ldots$ as specified with their start and end time $a_i = \langle s_i, f_i \rangle$. We have to schedule maximum number of jobs on a resource in a compatible manner.

Greedy algorithm :- Increasing order of their finishing times.

jobs	s_i	f_i	span
a_1	2	4	2
a_2	1	5	4
a_3	4	6	2
a_4	5	8	3
a_5	6	9	3

a_1, a_3, a_5. ← max number.

This approach is optimal.

Step 1: Sort requests in increasing number of their finishing times.

$a_1 < a_2 < a_3 < a_4 < a_5$

Step 2: Consider the next request a_i from P.

for each interval, discard a_j and delete from P if it conflicts with a_i.

Step 3: Repeat until P is empty.

→ Proving optimality :

Let $i_1, i_2 \ldots i_k$ be set of jobs scheduled by my solution S in increasing finishing time.

Let $j_1, j_2 \ldots j_m$ be set of jobs scheduled by optimal solution O in increasing finishing time.

Clearly $k \leq m$.

Claim 1 : our r^{th} request finishes no later than r^{th} request of optimal

$$f_{i_r} \leq f_{j_r} \quad \forall \; r \leq k .$$

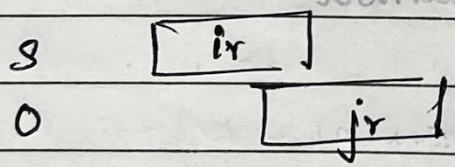
Proof by induction :-

Base case $r = 1$ follows.

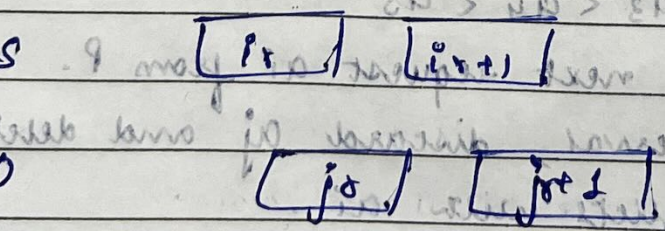
Suppose hold claims for some r'.

It will hold for $r+1$ as well

we have $f(i_r) \leq f(j_r)$



Clearly $f(i_r) \leq f(j_r) \leq s(j_{r+1})$



$$f(i_{r+1}) \leq f(j_{r+1})$$

Thus claim holds true.

Date : / /20

Claim 2 : $k = m$

Suppose (if possible) $k < m$.

we have $f(i_k) \leq f(j_k)$ (by claim 1).

Since $k < m$, there is at least one more job j_{k+1} after j_k in the optimal.

clearly j_{k+1} is compatible with i_k also as

$$f(i_k) \leq f(j_k) \leq s(j_{k+1})$$

Our algo wouldn't have stopped if there were more choice of picking a job request.

Hence by contradiction, $k < m$ is not true.

$\therefore k = m$

Time complexity :

Sort the requests in increasing order of their finishing time. $= O(n \log n)$

weaker algorithm : repeat until P is empty :

Schedule the next request a_i in P

for each interval : discard a_j

$O(1)$ for each a_j, total time for each $a_i = O(n)$.

total jobs = n.

$= O(n^2)$

$\therefore$ Time complexity = $O(n^2)$

stronger algo : include a_i in schedule.

for each interval a_j discard.

repeat until P is empty

$O(1)$ to take decision for each job. Total n jobs.

$= O(n)$.

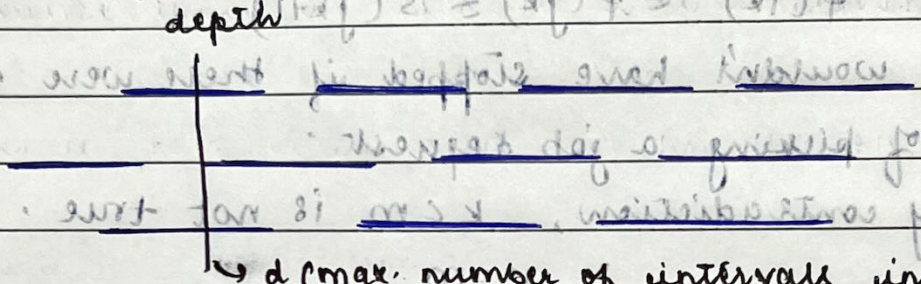
Time complexity = $O(n \log n)$.

~~1) Increasing finishing times :-~~

2) Interval partitioning :-

Problem statement :- A set of intervals (s_i, f_i) and a collection of similar resources, schedule all intervals minimising the resources.

Increasing finishing times :-



Can not schedule them in two rooms

clearly at least d resources are required to schedule.

$OPT \geq d$.

Algorithm :-

interval - partition (P, L)

P = { set of intervals }

L = { 1, 2, . . , d }.

Sort intervals in increasing order of their finishing time.

Pick next interval i in P.

Remove the colour of conflicting intervals from consideration

colour i with any available colour.

Correctness: Feasibility: Every interval gets a label.
No two intervals get the same label.

Proof: If we do not have a colour this means that all available 'd' colours have been assigned to conflicting 'd' intervals and there is still one more interval left

∴ intervals = d+1 which is a contradiction

No. of colours required ≥ d

since we are meeting the lower bound, it is an optimal solution.

Optimality = yes.

3) **Shortest path problem :**

Problem statement :- Given a directed graph $G = (V, E)$ with edge lengths 'l' and a pair s, t of vertices, we have to find a shortest path from s to t.

Djikstra's Algorithm :-
Single source shortest path
Distance table is used to store shortest distance of vertex v from source node.
works for both for directed and undirected graph.

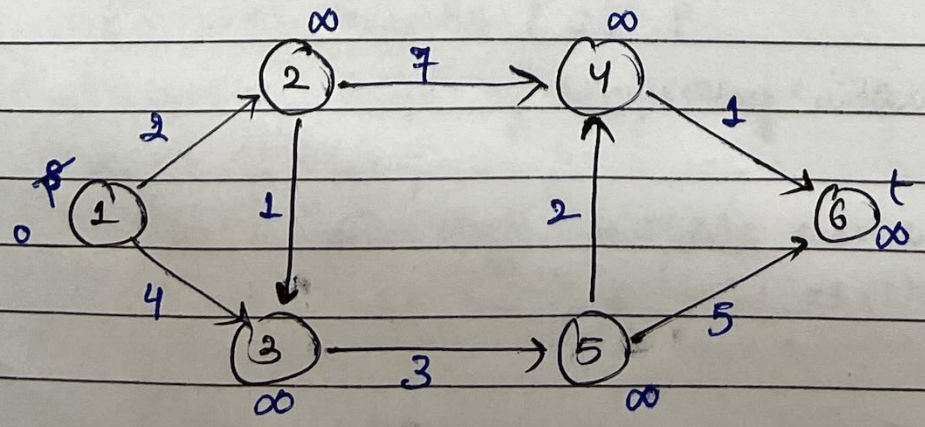
Algorithm :-

i) Initialize weights. For source node (s) = 0.
for all other nodes = ∞

ii) for current node, consider all unvisited nodes and calculate the distance.
Always assign smaller values

iii) when all neighbours are considered, current node is marked visited and never explored again.

iv) stop when all nodes are visited.



Date : / /20

Relaxation : If, (d[u] + c(u,v) < d[v])
then d[v] = d[u] + c(u,v).

nodes	1	2	3	4	5	6
distance	(0)	∞	∞	∞	∞	∞

choose 1.

d[2] = d[1] + c(1,2) = 0 + 2 = 2 ✓ min(2, ∞) = 2

d[3] = d[1] + c(1,3) = 0 + 4 = 4. min(4, ∞) = 4.

✓0 (2) 4 ∞ ∞ ∞

choose 2.

d[3] = d[2] + c(2,3) = 2 + 1 = 3. min(3,4) = 3 ✓.

d[4] = d[2] + c(2,4) = 2 + 7 = 9. min(9, ∞) = 9.

✓0 ✓2 (3) 9 ∞ ∞

choose 3.

d[5] = d[3] + c(3,5) = 3 + 3 = 6 min(6, ∞) = 6.

✓0 ✓2 ✓3 9 (6) ∞

choose 5

d[4] = d[5] + d[5,4] = 6 + 2 = 8 min(8,9) = 8.

d[6] = d[5] + d[5,6] = 6 + 5 = 11 min(11, ∞) = 11

✓0 ✓2 ✓3 (8) ✓6 11

choose 4.

d[6] = d[4] + c(4,6) = 8 + 1 = 9

min(11, 9) = 9

✓0 ✓2 ✓3 ✓8 ✓6 9

Date : / /20

visited nodes (V')	unvisited nodes (V\V')
{1}	{2, 3, 4, 5, 6}.
{1, 2}	{3, 4, 5, 6}.
{1, 2, 3}	{4, 5, 6}.
{1, 2, 3, 5}	{4, 6}.
{1, 2, 3, 5, 4}	{6}.
{1, 2, 3, 5, 4, 6}	ϕ.
	stop.

Path : 1 → 2 → 3 → 5 → 4 → 6.

Proof of invariant :-

Let Pu denote the s-u path containing Dijkstra's edges.

Clearly, d[u] = l(Pu) from the algorithm

Invariant : for each node u ∈ S, Pu is the shortest s-u path i.e. d[u] = length of shortest s → u path.

Proof by induction :-

Base case : |S| = 1 since S = {s} and d[s] = 0.

Induction hypothesis :

Let v be the next node added to S. Suppose v is added via u i.e. using the edge (u, v).

Pv = Pu followed by (u, v).

$$l(Pv) = l(Pu) + l(u,v) = d[u] + c(u,v) = \pi(v).$$

Consider any other s → v path.

$$l(P) \geq \pi(v).$$

Let e = (x, y) be the first edge in P that leaves S.

Let P' be a subpath from s to x.

Page No.
Date : / /20

Length of $P \geq \pi(v)$ as soon as it reaches y.
(because $l(P_v) \geq \pi(v)$ and we are adding some cost to reach y. $\therefore P \geq \pi(v)$.)

$\pi(y) = d[u] + l_e$

$\pi(y) \leq d[x] + l(x,y)$

$\leq l(P') + l(x,y)$ by induction hypothesis.

Since Djikstra chose v instead of y,

$\pi(v) \leq \pi(y)$.

$l(P) \geq l(P') + l_e \geq d[x] + l_e \geq \pi(y) \geq \pi(v)$.

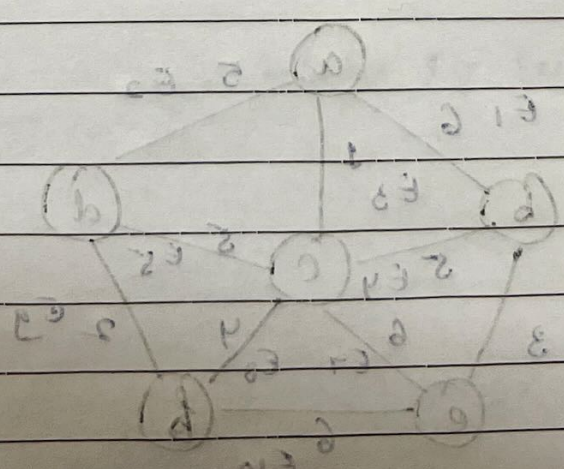
- $l(P) \geq l(P') + l_e$: non-negative lengths
- $l(P') + l_e \geq d[x] + l_e$: inductive hypothesis
- $d[x] + l_e \geq \pi(y)$: definition of $\pi(y)$
- $\pi(y) \geq \pi(v)$: djikstra chose v over y.

Time complexity :-

$O(V \log V) + O(E \log V) + O((V-1) \log V)$

$= O((V+E) \log V)$

$= O(E \log V)$ for connected graph.



Date : / /20

4) Spanning Tree :-

Problem statement :-

Given a connected undirected graph $G = (V, E)$, a spanning tree is a tree that spans all the vertices.

Minimum Spanning Tree :-

Given a connected graph (undirected) $G = (V, E)$ with weights on edges, a minimum spanning tree is a S.T with min. total weight.

Kruskal Algorithm :-

Sort the edges in increasing order of their weights.

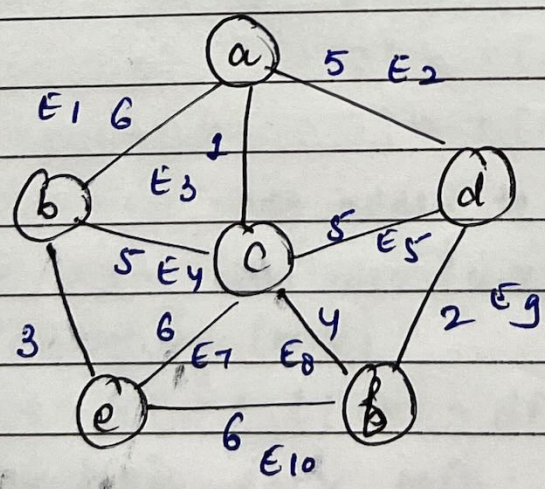
$e_1, e_2, e_3 \ldots e_m$.

while there are more edges and we have selected $< n-1$ edges. do

select the next edge if it does not form cycle

else discard it

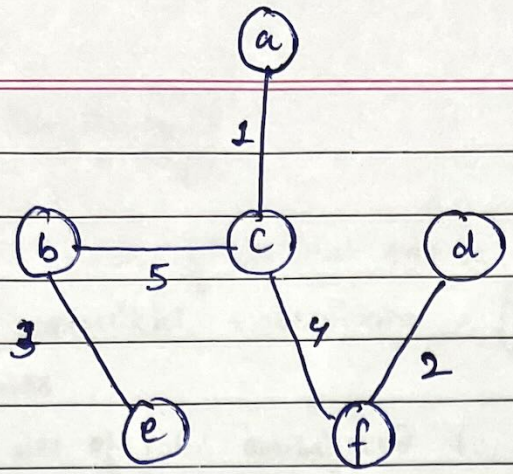
Eg :



Increasing order of weights -

$E3 = 1 < E9 = 2 < E6 = 3 < E8 = 4 < E2 = 5 \leq E4 = 5$

$< E1 = 6 \leq E10 = 6$,



← M.S.T.

choosing E_2 forms a cycle hence discard E_2.
choosing E_1 forms a cycle hence discard E_1.
choosing E_{10} " " " " " E_{10}.

weight = 1+2+3+4+5
= 15.

Correctness of Kruskal's algorithm :-

Acyclic : by algorithm

Claim : every edge selected by KA belongs to the MST.

Proof :-

edge picked by Kruskal is the cheapest edge in a cutset.

Let $e_j = (v, w)$ be an edge picked by kruskal at some point of time.

Study Algorithm from Basics :

Psuedocode :- A way of writing algorithm.

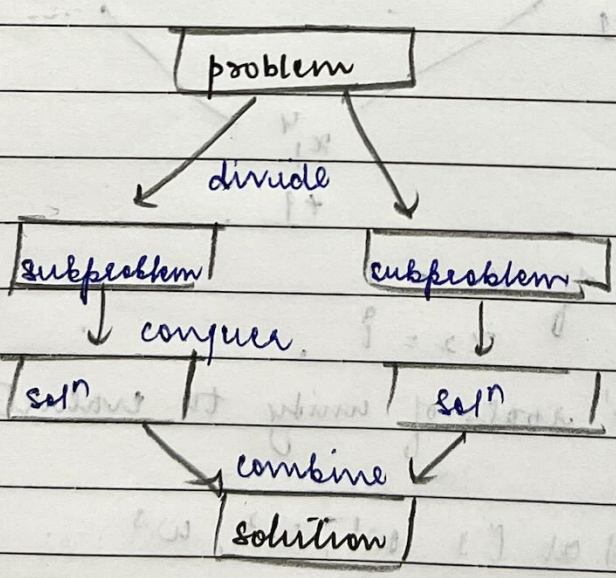
Iterative algo	Recursive Algo
The function runs until the condition is met true or false.	The function calls itself until it reaches the base condition.
Involves looping construct.	Integrates branching structure.

→ **Divide and Conquer :-**

Top-down approach.

Requires three steps :-

1. Divide the original problem into set of subproblems.
2. Solve every subproblem individually, recursively.
3. combine solution of sub problems into a solution of the whole.

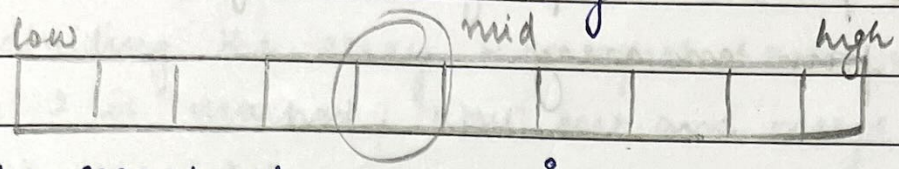


→ Relational formula :- formula we generate from the given technique.

→ Stopping condition :- condition at which we need to stop our recursion steps of D&C.

Date : / /20

1) Binary Search :- Sort the array.



let the search element = i.

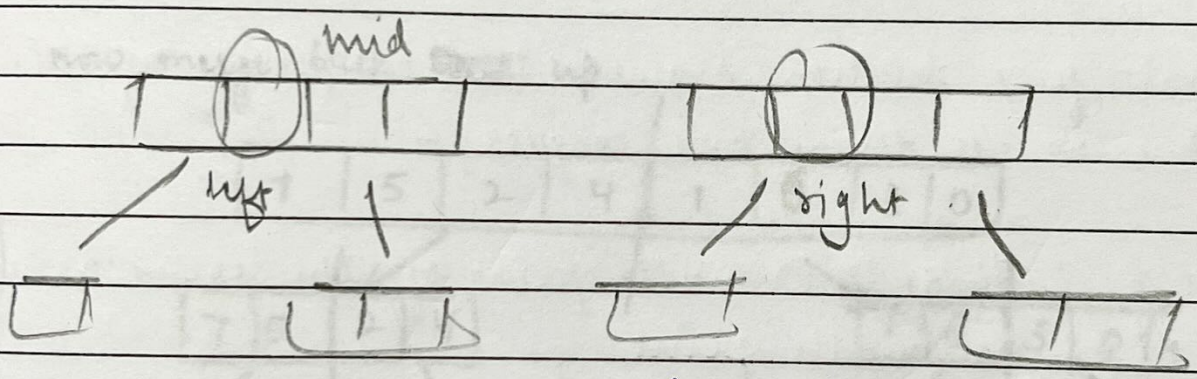
set mid = $\frac{low + high}{2}$.

compare i with mid.

if i < mid → search the left array

if i == mid → element found.

if i > mid → search the right array



keep going recursively until & i is found.

The problem is reduced to half its size in every step.

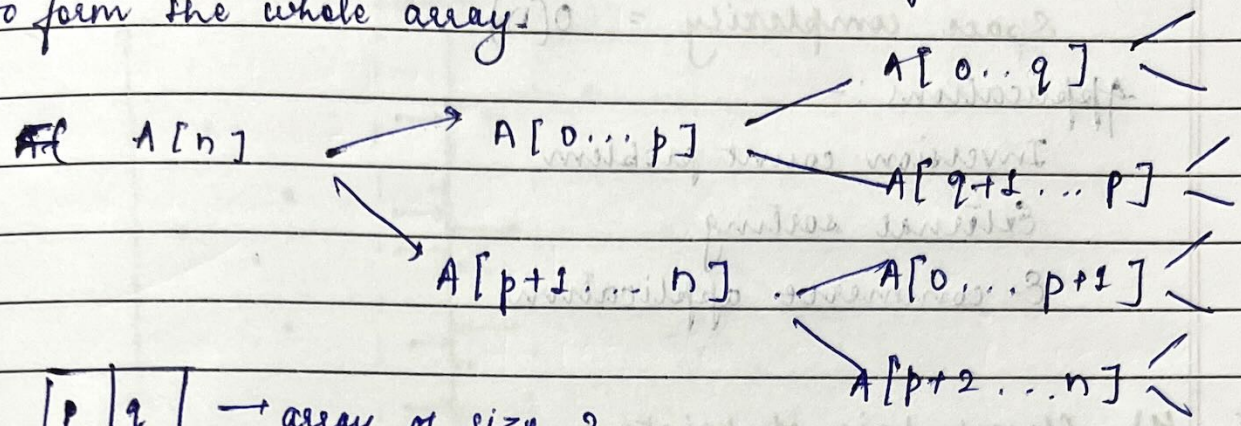
Recursive relation ⇒ $T(n) = T\left(\frac{n}{2}\right) + O(1)$.

Time complexity = $O(\log n)$.

Auxiliary space = $O(\log n)$.

3) Merge sort :-

Keep dividing the array into equal have until array of size 2 is reached. Now sort and merge back to form the whole array.

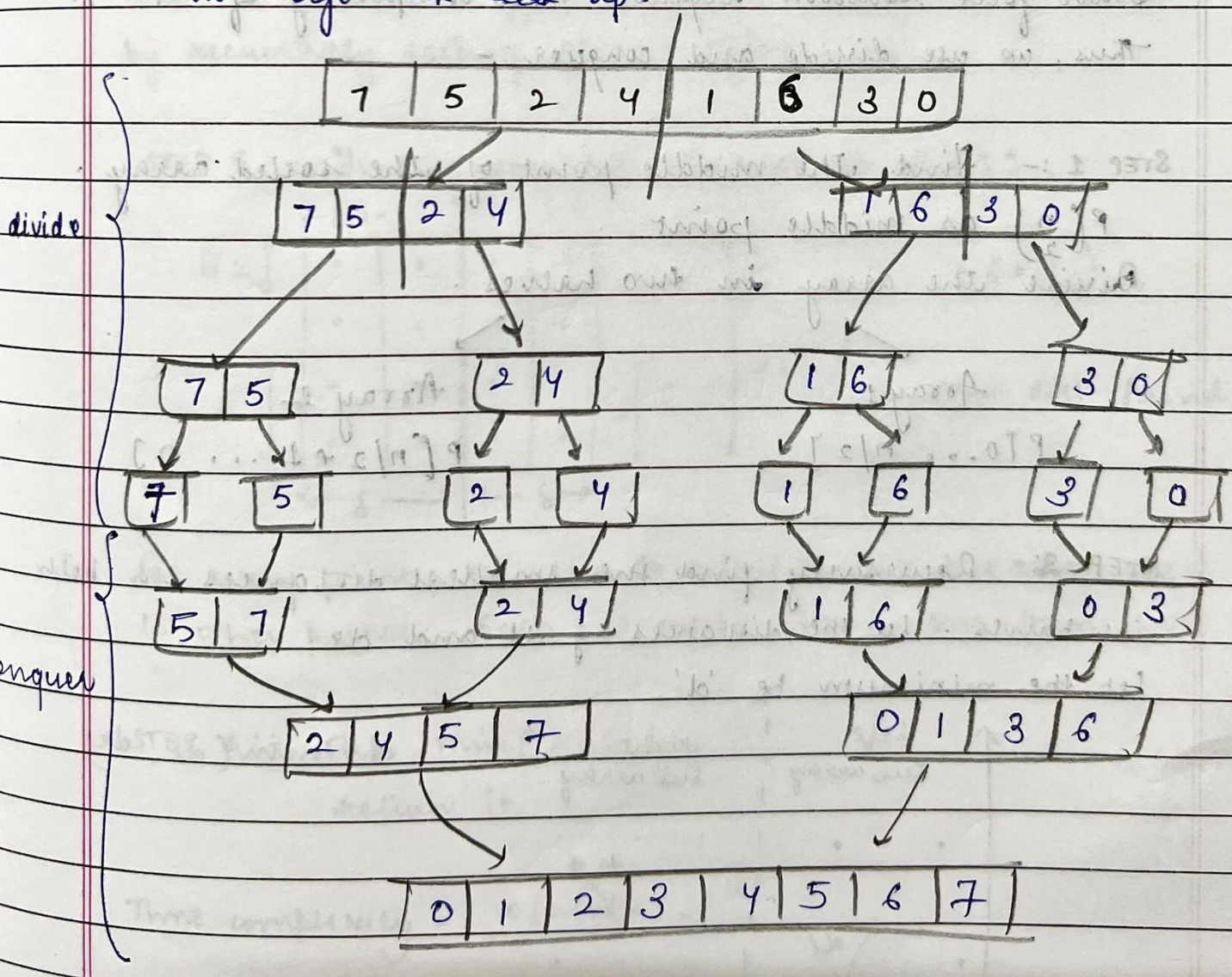


| p | q | → array of size 2.

if $p \leq q$ → it's fine

else if $p > q$, swap (p, q).

now merge back up.



Date : / /20

Recursive relation :- $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$.

Time complexity = $O(n * \log n)$

Space complexity = $O(n)$.

Applications :-

Inversion count problem

External sorting

E-commerce application

4) Closest pair of points :

Euclidean distance :- $\sqrt{(px-qx)^2 + (py-qy)^2}$

Brute force solution requires time complexity of $O(n^2)$.

Thus, we use divide and conquer :-

STEP 1 :- Find the middle point of the sorted array.

$P\left(\frac{n}{2}\right)$ as middle point.

Divide the array in two halves.

Array 1

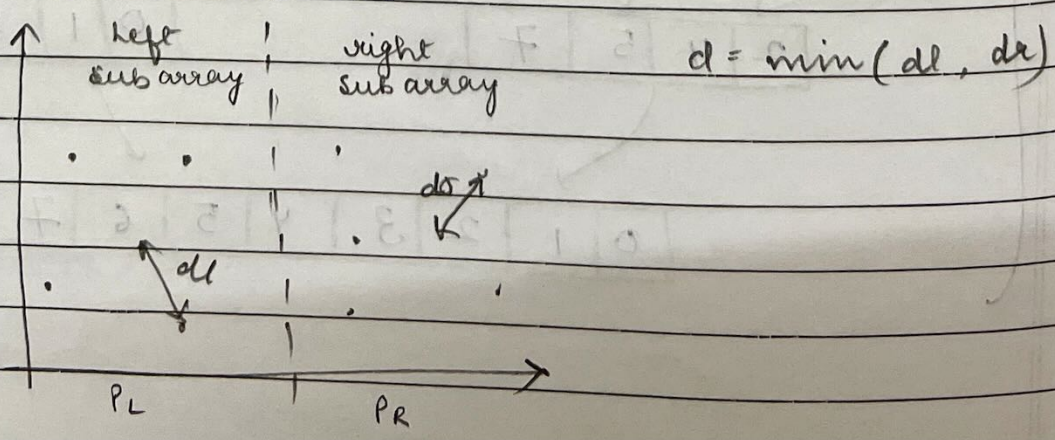
$P[0 \ldots n/2]$

Array 2

$P[n/2 + 1 \ldots n]$.

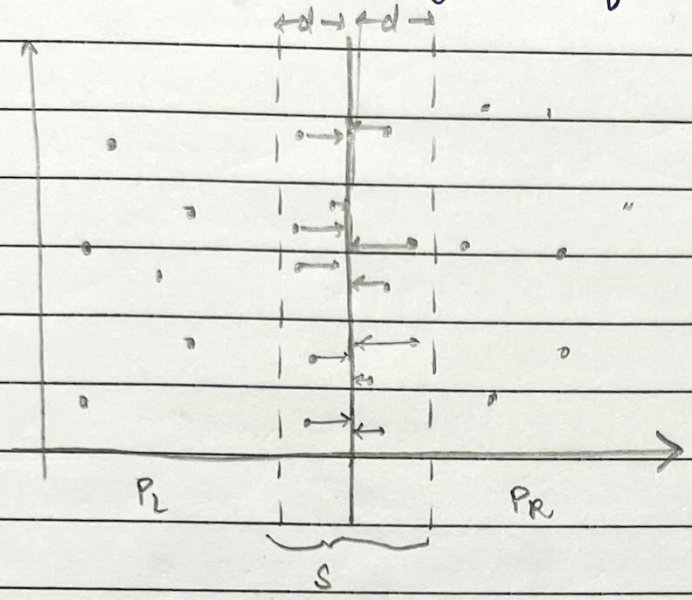
STEP 2 :- Recursively find the smallest distances in both the halves. Let the distances by dl and dr.

Let the minimum be 'd'.



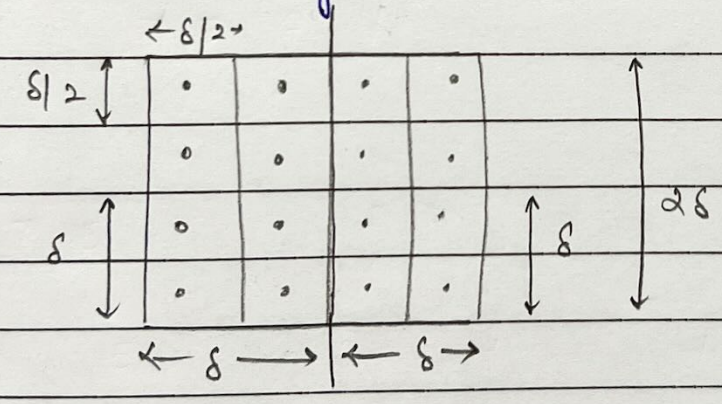
Date : / /20

STEP 3 :- The upper bound of minimum distance = d.
Now, we consider pairs such that one point is in left half and next point in right half.



STEP 4 :- Sort the strip [] according to y coordinates. This step is O(nlogn). It can be optimized to O(n) by recursively sorting and merging.

STEP 5 :- Finding smallest distance in strip [].



$\delta = d$.

let min dist found here = δ'.

For every point, we have to calculate the distance with 15 other points. Each box must contain one point only.

STEP 6 :- Find min(d, δ').
return it.

Time complexity = O(n log n)