

PMB

UNIT IV

(short notes)

- ① - Linear search
- ② - Binary search
- ③ - bubble sort
- ④ - selection sort
- ⑤ - Insertion sort
- ⑥ - Define concure
- ⑦ - ~~Greedy~~ Greedy method.
- ⑧ - Dynamic
- ⑨ - Backtracking

Example (i) First Point

Algorithm

(Linear search)

- A linear array DATA with 'N' elements & specific Item of Information are given. This Algorithm finds the Location of Item in Array Data
or sets LOC = 0

1. [Initialize] set $K=1$ & $LOC=0$

2. Repeat step 3 & 4 while $LOC=0$
& $K < N$

→ 3. IF [Item = DATA[K]] then
set $LOC = K$

→ 4. Set $K = K + 1$

5. [Successful?]

IF $LOC = 0$ then:

Write: Item is not in the
array Data

Else:

Write: LOC is the location of
Item

[End of IF structure]

6. Exit

- The complexity of search algorithm
is given by the number C of comparisons
betⁿ Item & DATA [K]

- we seek $C(n)$ for worst & average
case.

① Worst case -

- worst case occurs when

Item is the last element in the array
DATA

or is not there at all.

$$C(n) = n$$

$C(n) = n$ Worst-case complexity of Linear Search.

Average case

number of comparisons can be any numbers $1, 2, 3, \dots, n$ & each number occurs probability $P = \frac{1}{n}$ the

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

$$(1 + 2 + 3 + \dots + n) \cdot \frac{1}{n}$$

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

$$\frac{n+1}{2}$$

- Average case complexity approximately half the no. of element in the DATA list

Binary Search

A → Array

n → Input size

Item → Item to be search

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

| | | | | | | |
|----|----|----|----|----|----|----|
| 12 | 14 | 18 | 20 | 22 | 26 | 28 |
|----|----|----|----|----|----|----|

| | | | | | | | | |
|---|---|-----|---|-----|---|---|---|---|
| l | 1 | 2 | 3 | 4 | 5 | 6 | 7 | h |
| | | mid | h | mid | | | | |

$$mid = \frac{l+h}{2} = \frac{1+7}{2} = \frac{8}{2} = 4$$

tab 12

Item = 18 ?
Find

| l | h | mid | Item < A[mid] |
|---|---|-----|---------------|
| 1 | 7 | 4 | h = mid - 1 |
| 1 | 3 | 2 | h = 4 - 1 |
| 3 | 3 | 3 | <u>h = 3</u> |

$$mid = \frac{l+h}{2} = \frac{1+3}{2} = 2$$

$A[mid] == Item$

$A[mid] < Item$

$l = mid + 1$

$r = 3$

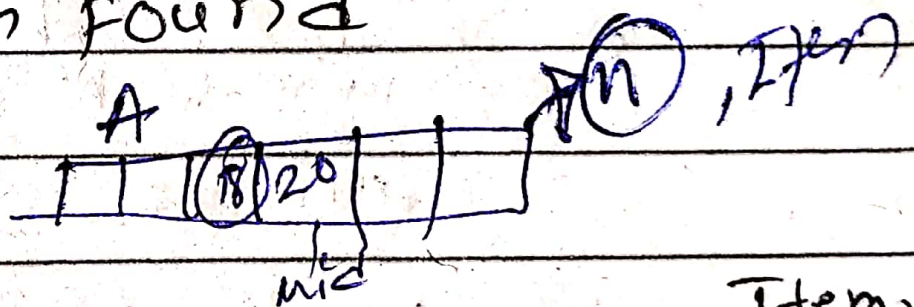
$l = 3 \quad h = 3 \quad mid = 3$

if ($A[mid] == Item$)

yes

Item found

Algorithm



```
int BinarySearch (A, n, key Item)
```

```
{
```

```
     $l = 1, h = n$ 
```

```
    while ( $l \leq h$ )
```

```
    {
```

$$\text{mid} = \left(\frac{l+h}{2} \right);$$

if (Item == A[mid])
return mid;

if (Item < A[mid])

$$h = \text{mid} - 1;$$

else

$$l = \text{mid} + 1;$$

}

return 0

}

Description

- Binary search also known as half interval search logarithmic search or binary chop.

- It divide the array in two two subarray.

- Binary search looks for particular item by comparing with middle of Array.

- IF match found Index of Item is returned.

- IF middle item is greater than item then item is searched sub-array to left (before mid)

→ IF middle item is less than item then item is searched sub-array to right

→ Repeat this process until at middle we get element

* Bubble sort

"Bubble sort is simplest sorting algorithm, that works by repeatedly swapping the adjacent elements if they are in wrong order"

Example

- Array consists following elements

$n=5$

A:

| | | | | |
|---|---|---|---|---|
| 5 | 1 | 4 | 2 | 8 |
|---|---|---|---|---|

Pass 1:

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 4 | 2 | 8 |
|---|---|---|---|---|

- compare first 2 element

$5 > 1$ swap & continue

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 5 | 2 | 8 |
|---|---|---|---|---|

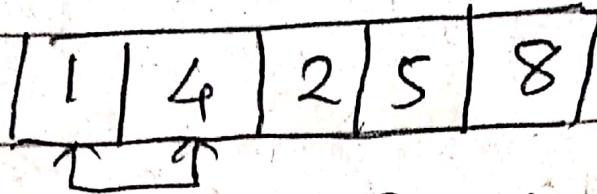
 (5 > 2) swap

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 2 | 5 | 8 |
|---|---|---|---|---|

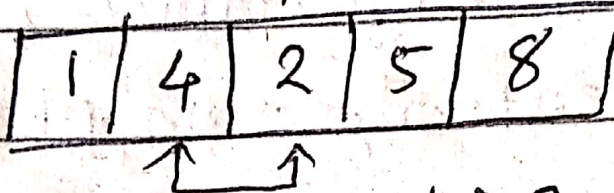
5 not greater than 8
so do not swap

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 2 | 5 | 8 |
|---|---|---|---|---|

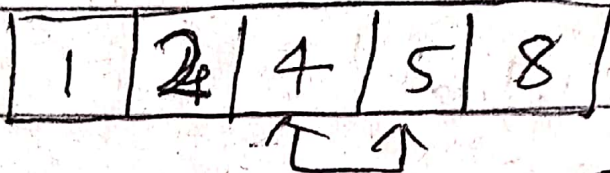
PASS 2:



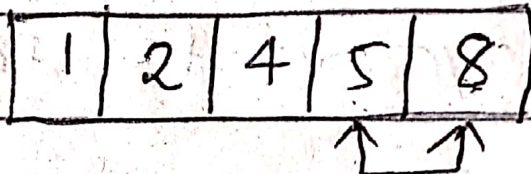
Repeat process for next iteration
4 is sorted (no change)



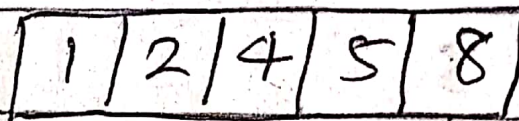
4 & 2 swap



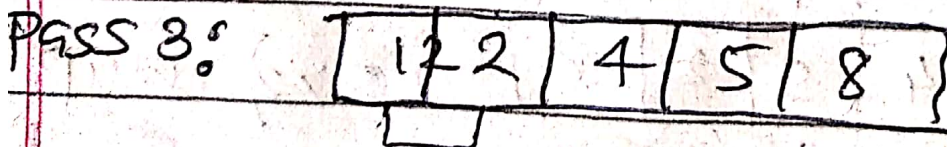
5 is sorted (no change)



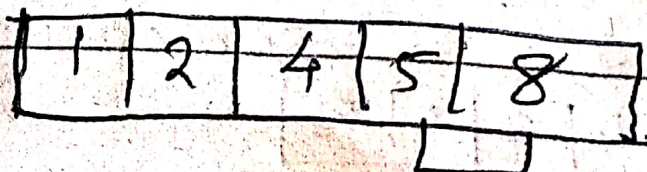
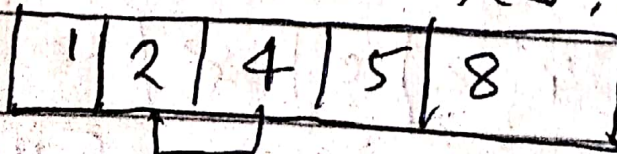
8 is sorted no change



• our Array already sorted but our algorithm does not know. It require one more swap to know



2 sorted, no change



Sorted

* Selection Sort ^{min to n} ^{swap} min, i (sorted & unsorted array, passes)

- selection sort is simple sorting algorithm
- sorting algorithm is an in-place comparison based algorithm in which list is divided into two parts.
- sorted part at left end & unsorted part at right end.
- Initially sorted part is empty & unsorted part is entire list.
- smallest element is selected from unsorted array & swapped with left most element.

that elements becomes part of sorted array.

- This process continuous moving unsorted array boundary one element to right.

Note -

- not suitable for large dataset.

[Avg & worst complexity = $O(n^2)$]

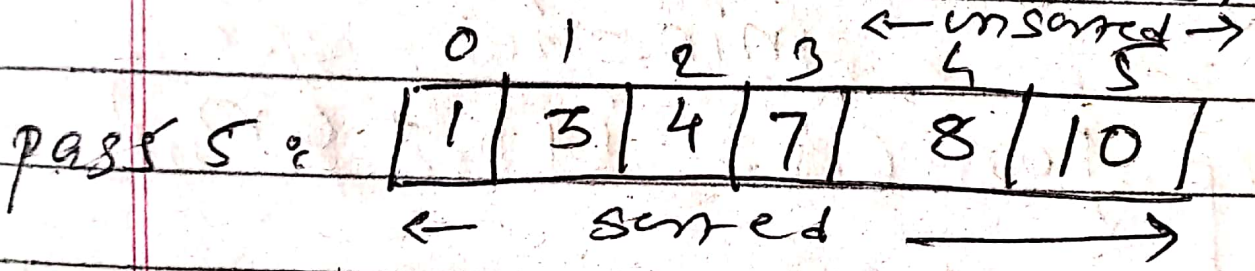
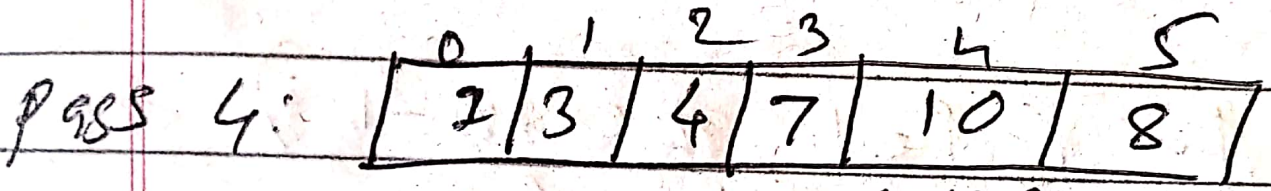
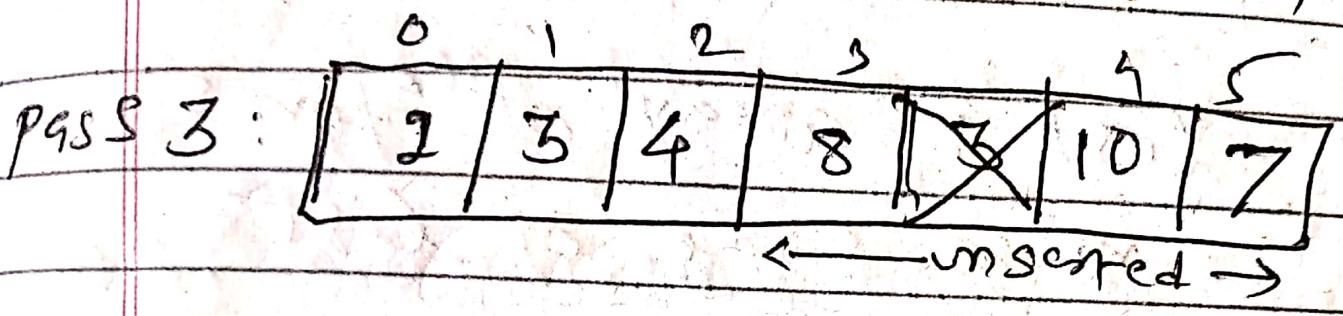
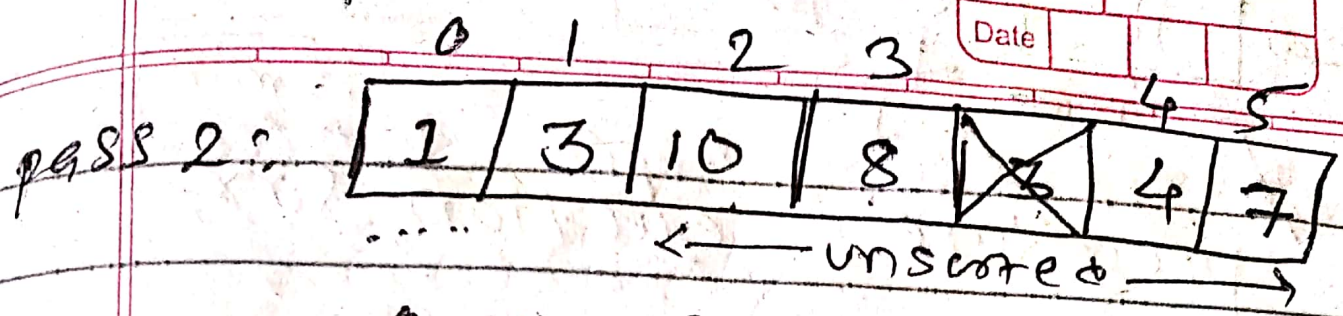
Example

| | | | | | | |
|--|---|---|----|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| | 7 | 4 | 10 | 8 | 3 | 1 |

$n = 6$

| | | | | | | |
|---------|---|---|----|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| pass 1: | 1 | 4 | 10 | 8 | 3 | 7 |

← some ~~sort~~ unsorted Find smallest element 2 & swap with 7 & shift counter to 1



⇒ sorted AT pass 5
 (n-1) steps to sort selection
 sort.

- ⇒
- Find smallest ✓
 - swap with leftmost ✓
 element of unsorted array
 - Shift counter to + 1 ✓

procedure selection sort

list : array of items

n : size of list

for ($i = 1$ to $n - 1$)

 min = i ;

 for ($j = i + 1$ to n)

 if ($list[j] < list[min]$)

 min = j ;

 end if

 end for

 if index min $\neq i$ then

 swap $list[min]$ &

$list[i]$

 end if

end for

* Insertion Sort

- Insertion sort works in the similar way as we sort cards in our hand in a card game.

- We assume, first card is already sorted.

- Then we select unsorted card or number

- unsorted number or card is greater than the card in hand or left side number It is placed on right

- otherwise (if right side number is less placed) left side.

↳ so on.

- "Insertion sort Algorithm insert unsorted element to proper place"

Example

Suppose Array having elements

A:

| | | | |
|----|----|----|---|
| 15 | 10 | 20 | 5 |
|----|----|----|---|

n = 4

Pass 1:

| | | | |
|----|----|----|---|
| 15 | 10 | 20 | 5 |
|----|----|----|---|

We assume first no 15 is sorted

Pass 2:

| | | | |
|----|----|----|---|
| 10 | 15 | 20 | 5 |
|----|----|----|---|

- 10 is less than 15 so Insert 10 before 15

Pass 3:

| | | | |
|----|----|----|---|
| 10 | 15 | 20 | 5 |
|----|----|----|---|

no change bez 20 is greater than 15

Pass 4:

| | | | |
|---|----|----|----|
| 5 | 10 | 15 | 20 |
|---|----|----|----|

∴ ~~5~~ 5 is inserted to proper position i.e before 10
5 compares with its left elements
put proper place acc. to sorting.

Divide & Conquer

↳ "Design or Approach to solve a problem"

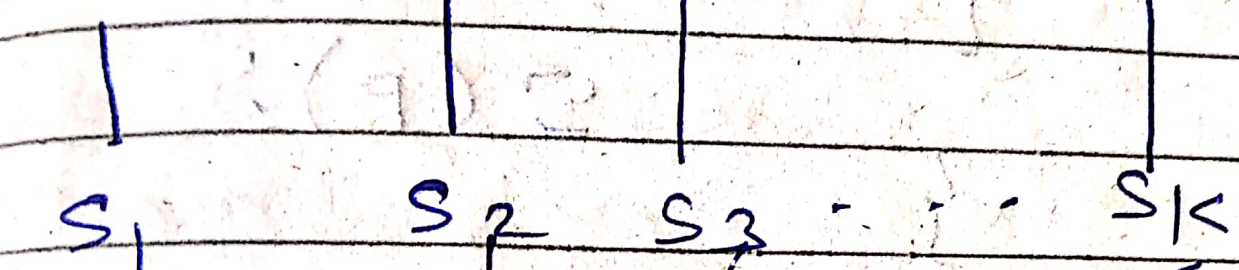
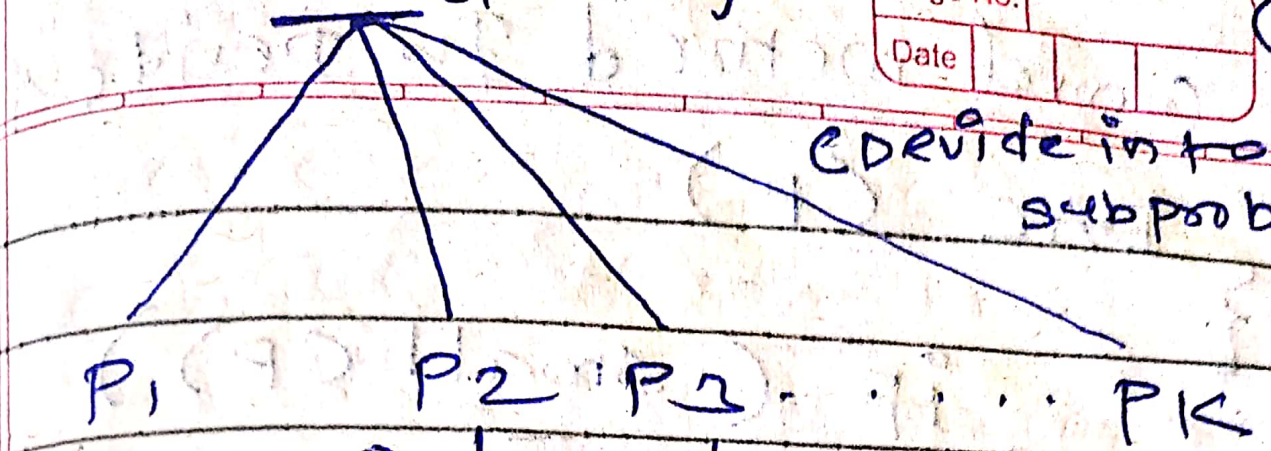
↳ solving computational problem

↳ problem is given size is n

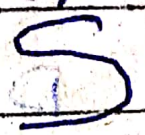
↳ problem is divided into subproblem & make solⁿ for subproblem & combine All subproblem solution to get original problem solⁿ

P (problem) size n (large)

(divide into subproblem)

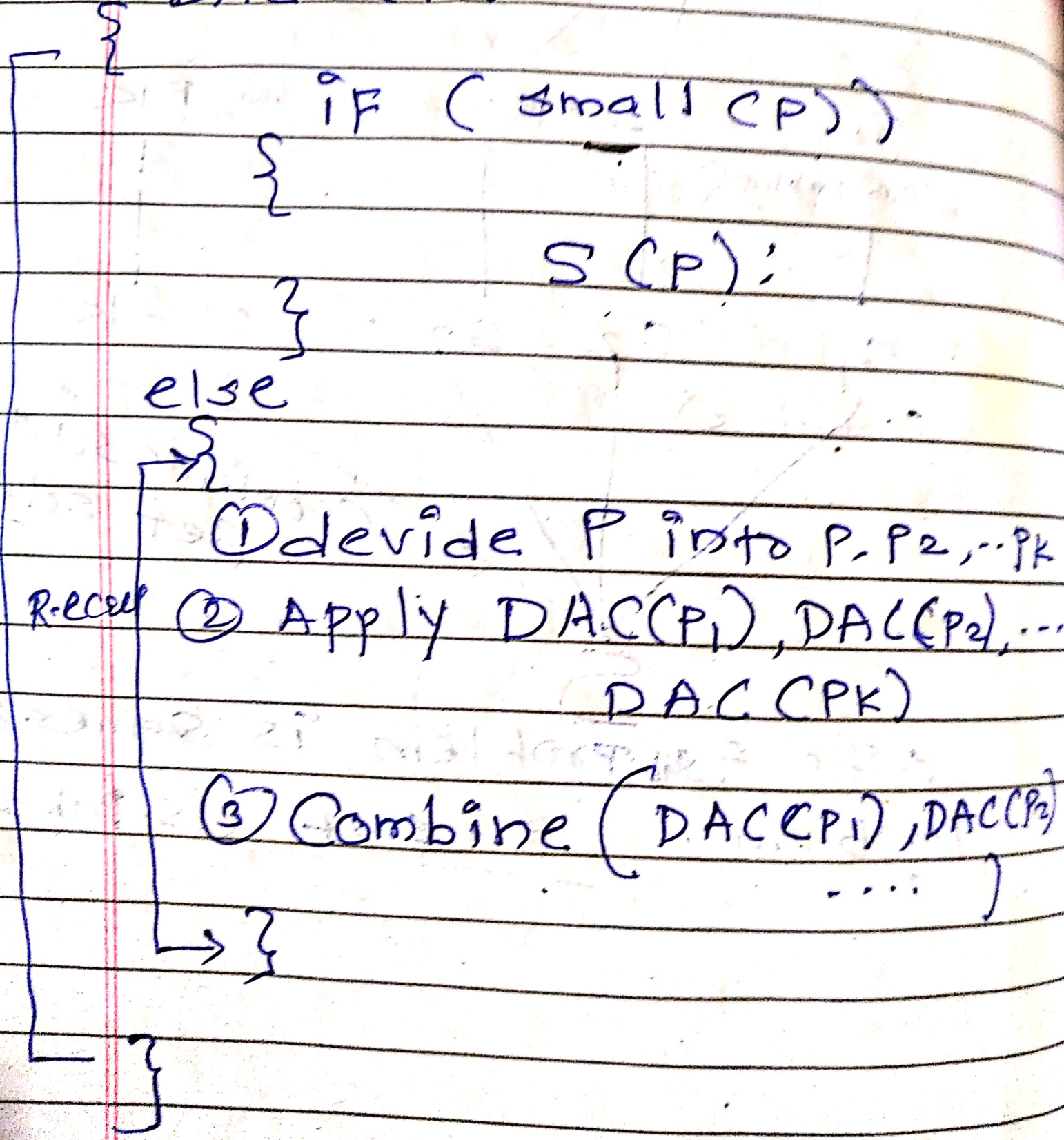


(combine solⁿ to get solution)



Note: sub problem is same as problem.
P = sort of S. Prob = sol

General method for Divide & Conquer
DAC (P)




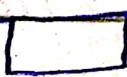
* Greedy Algorithm (Approach)

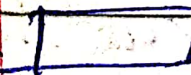
Greedy is Algorithm paradigm that builds up sol piece by piece, always choosing the next piece that offers the next most obvious & immediate benefit.

“So problems where choosing locally optimal also leads to global sol are best fit for Greedy”

e.g. Fractional Knapsack

A  wt = 10
value = 60

B  wt = 20
value = 100

C  wt = 30
value = 120

full

full

2/3

Take A, B & 2/3rd of C

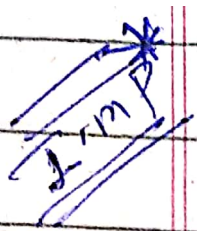
Total weight = 10 + 20 + 10

+ 2/3 = 50

Capacity = 50

Total value =

$60 + 100 + 120 \times \frac{2}{3} = 240$



Dynamic Method (Programming)

“ Dynamic programming is plain mainly optimization over plain Recursion”
- where we see Recursive solⁿ that has repeated calls for same Input. We can optimize using dynamic programming”

“ The idea is simply to store the result of subprogram subproblem so that we do not have to re-compute them when needed later”

- This simple optimization Reduces time complexities from exponential to polynomial”

e.g If we write simple Recursive solⁿ for fibonacci numbers, we get exponential time

complexity & if we optimize it by storing solⁿ of subproblems, time complexity reduces to linear!

1

```

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

↳ Recursion: Exponential



```

F[0] = 0
F[1] = 1
for (i = 2; i <= n; i++)
{
    F[i] = F[i-1] + F[i-2];
}
return F(n)
    
```

↳ Dynamic Prog: Linear

Dynamic Programming

① Dynamic prog use to obtain optimal solution

② Less efficient

③ Ex - 0/1 Knapsack

④ It guaranteed that Dynamic prog. generate optimal solution using principle of optimality

⑤ Dynamic prog. follows top down approach

⑥ we calculate optimal solution by considering previous steps e.g backtracking recursion

⑦ It does not work forward fashion (backward)

Greedy method (Algorithm)

① Greedy Algorithm Also used to get optimal solⁿ

② more efficient

③ Fractional Knapsack

④ There is no such guarantee.

⑤ Greedy method follows bottom up approach

⑥ we make best choice at moment. It lead global optimal solution

⑦ It works on forward fashion

* Backtracking

- Backtracking is Algorithmic Technique for solving problem recursively, by trying to build solution incrementally, one piece at a time, removing those solution that fail to satisfy the constraints at any point of time.

"general Algorithmic technique that consider searching every possible combination in order to solve computational problem"

There are 3 types of problem in backtracking

① Decision problem - In this we search for feasible solution.

② Optimization problem - We search for best solution.

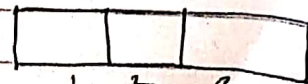
③ Enumeration problem - Find all feasible solutions

Example

- It is like conditional (constraint) probability

Suppose two Boys & one Girl they have to arrange in 3 positions or chairs

B₁ B₂ G₁
└───┘ └──┘
boys girls



$n=3$

