Defn of Algorithm: An algorithm is an ordered set of unambiguous, executable steps, defining a terminating process.

Algorithm is considered as one of the fundamental concepts in computer science.

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value or set of values are output.

What kind of problems are solved by algorithms?

1. Transportation and Logistics: Algorithms are used to
2. optimize transportation routes,
2. Internet search
3. Recommendation systems
4. Social media and news feeds
5. Financial modeling and trading
6. Medical diagnosis and treatment planning
7. Natural language processing
8. Image and video processing

A machine - compatible representation of an algorithm is called a program.

Criteria / properties:

1. Input - Zero or more quantities are externely supplied.
2. Output - At least one quantity is produced.
3. Definiteness - Each instruction is clear.
4. Finiteness - If we ~~fin~~ trace out the instructions of an algorithm, for all cases the algorithm terminates.
5. Effectiveness: Every instruction must be very basic so that it can be carried out. It must be flexible.

Areas of study: 1. How to devise algorithms.
2. How to validate algorithms.
3. How to analyze algorithms.
4. How to test a program.
5. Algorithm specification

- Pseudocode : algorithms are represented with precisely defined textural structure.
- Flowchart

Algorithm specification:

Pseudocode : Algorithm are represented with precisely defined textural structure.

→ Flowcharts.

Algorithm as technology:

→ Analysis of algorithm refers to the task of determining how much computing time and storage an algorithm requires

→ By analyzing several candidate algorithm for a problem, a most efficient one can be easily identified.

→ Total system performance depends on choosing efficient algorithms as much as on choosing on fast hardware.

Loop ~~intervals~~ invariants:

Loop invariants helps to understand why an algorithm is correct.

There are three things about a loop invariant:

Initialization: It is true prior to the first iteration for loop.

Maintenance: It is true before an iteration of the loop, it remains true before ~~an iteration of~~ the ~~loop~~ next iteration.

Termination: when the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Insertion Sort:

Example: Suppose an array A contains 8 elements as follows: 77, 33, 44, 11, 88, 22, 66, 55.

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1	$-\infty$	(77)	33	44	11	88	22	66	55
K = 2	$-\infty$	77	(33)	44	11	88	22	66	55
K = 3	$-\infty$	33	77	(44)	11	88	22	66	55
K = 4	$-\infty$	33	44	77	(11)	88	22	66	55
K = 5	$-\infty$	11	33	44	77	(88)	22	66	55
K = 6	$-\infty$	11	33	44	77	88	(22)	66	55
K = 7	$-\infty$	11	22	33	44	77	88	(66)	55
K = 8	$-\infty$	11	22	33	44	66	77	88	(55)
Sorted:	$-\infty$	11	22	33	44	55	66	77	88

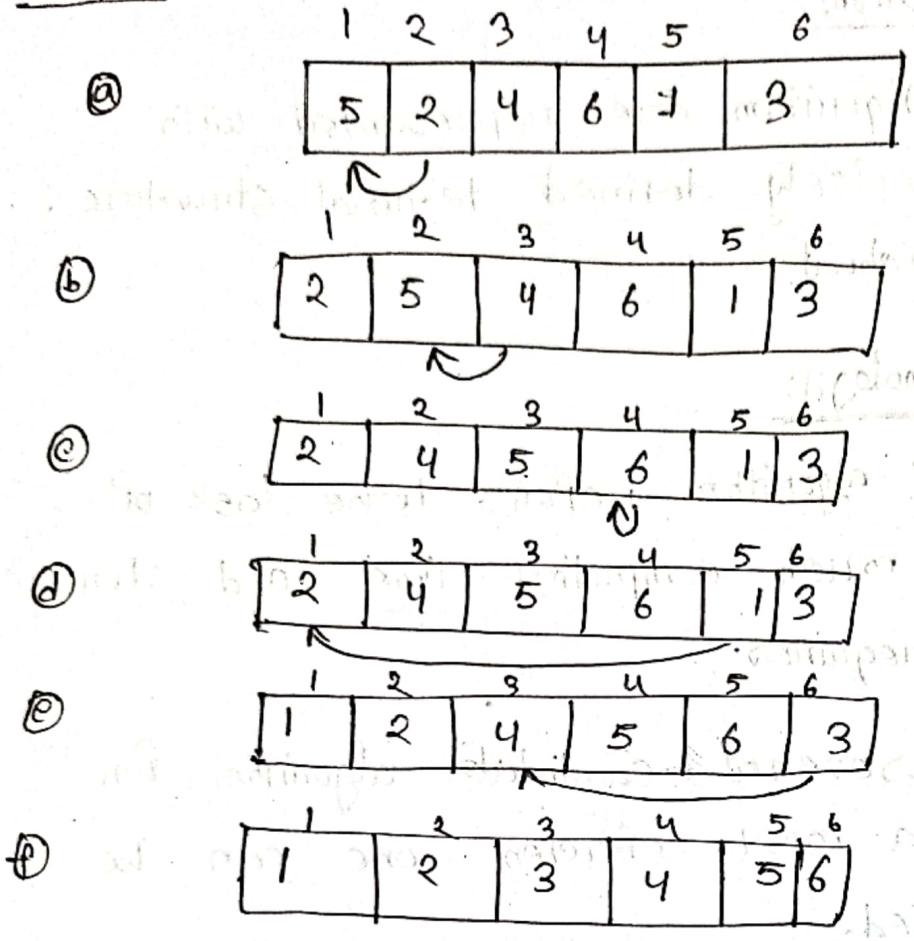
Insertion sort for n = 8 items.

Pseudocode:

① for j = 2 to A length
{
② key = A[j]
③ Insert A[j] into the sorted sequence
A[1 . . . j-1]
④ i = j-1
⑤ while i > 0 && A[i] > key
{
⑥ A[i+1] = A[i]
⑦ i = i-1
}
⑧ A[i+1] = key
}

loop start from j=2 cause the first or the only element is always sorted.

Example:



Sorted array, $A = \{5, 2, 4, 6, 1, 3\}$

Array indices appear above and values stored in the array position appear within the rectangles.

Correctness of Insertion sort:

- **Initialization:** We start by showing that the loop invariant holds before the first loop iteration, when $j=2$.

The subarray $A[1, \ldots, j-1]$, therefore consists of just the single element $A[1]$, which is in fact the original element in $A[1]$.
Moreover, this subarray is sorted, which shows that the loop invariant holds prior to the first iteration of the loop.

- **Maintenance:**

The body of the for loop works by moving $A[j-1], A[j-2] \ldots$ by one position to the right until it finds the proper position for $A[j]$ at which point it inserts the value of $A[j]$. The subarray $A[1 \ldots j-1]$ then consists of the elements originally $A[1 \ldots j]$ but in sorted order. Incrementing 'j' for the next iteration of the for loop then preserves the loop invariant.

- **Termination:**

The condition causing the for loop to terminate is $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n+1$ at that time. Substituting $n+1$ for j in the wording of loop invariant, we have that the subarray $A[1 \ldots n]$ consists of the elements originally in $A[1 \ldots n]$, but in sorted order. Observing that the subarray $A[1 \ldots n]$ is the entire array, we conclude the array is sorted.

→ Hence, the algorithm is correct.

Algorithm analysing: Analysing an algorithm has come to mean predicting the resources that the algorithm requires.

→ Computational time measure.

→ By analysing - can identify the most efficient one.

Method:

⇒ count steps taken by algorithm

⇒ use count to derive formula, based on size of problem n.

⇒ use formula to understand overall efficiency.

Example:

```
int sum (int a[], int n)
{
    int sum = 0;  —①
for (j=0; j<n; j++)
      ②   ③   ④
sum = sum + a[j]
 ⑦     ⑤     ⑥
return sum;  —⑧
```

1, 2, 8: happen once (3)

3, 4, 5, 6, 7: Happens for n times } 5n+3

Complexity function: $f(n) = 5n+3$

Time complexity depends on:

(1) Input size — Best case
worst case
Average case

(2) Number of operation:

$1 < O(\log(n)) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

Problems:

(1)
```
{ int a, b, sum;        —①
  a = 10; b = 20;
    ②     ③
  sum = a+b;            —④
  Printf ("%d\n", sum); —⑤
  return 0;             —⑥
}
```

$\Rightarrow T = O(6)$

(2)
```
{
  int i;                —①
  for (i=1; i<=n; i++)
       ②    ③   ④
  {
    printf (" ");       —⑤
  }
  return 0;             —⑥
}
```

$f(n) = 3n+3$, $T = O(n)$

[II]

```
{
int i=0, j=0;                    -①  -②
for (i=1; i<=n; i++)
        ③   ④   ⑤
{
 for (j=1; j<=n; j++)
        ⑥   ⑦   ⑧
 {
  Printf (" ");                  -⑨
 }
}
return 0;                       -⑩
```

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

$= 2n^2 + 2n + 5$

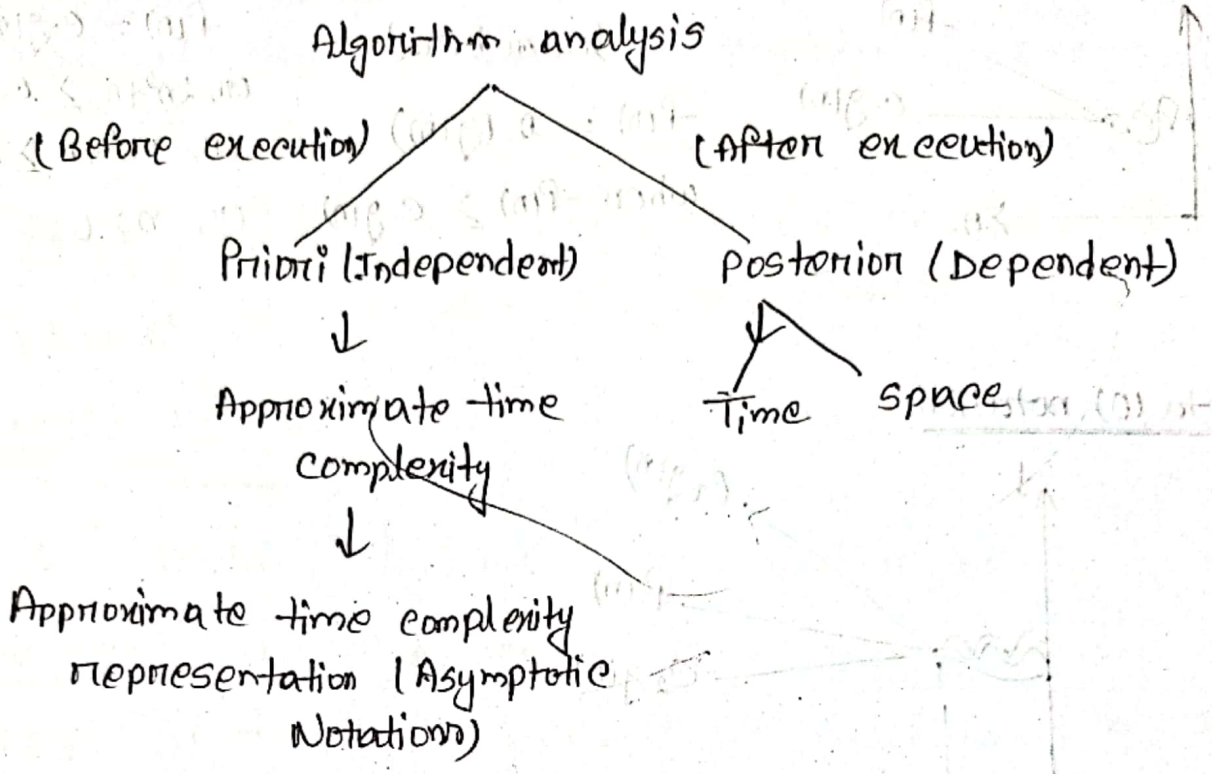
$T = O(n^2)$

[III]

```
{
for (i=1; i<=n; i*=2)
{
 Printf = (" ")
 for (j=1; j>=1; j/=2)
    {
     printf (" ");
    }
 return 0;
}
}
```

⇒

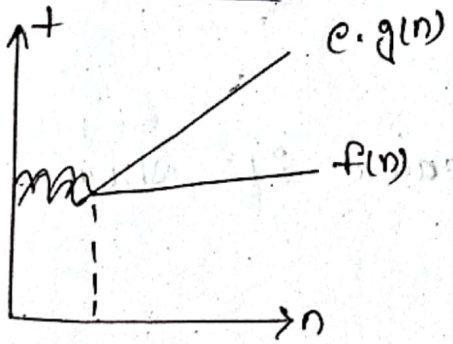
Asymptotic notations:



Asymptotic Notation: Big O, Big (Ω), Theta (θ), little (o), little (Ω)

$f(n) = g(2n+3)$
$O(n) = g(n)$

Big-O notation:



worst case
upper bound
$f(n) = O(g(n))$
when $f(n) \leq c \cdot g(n)$

Ex: $f(n) = 2n^2+n$

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

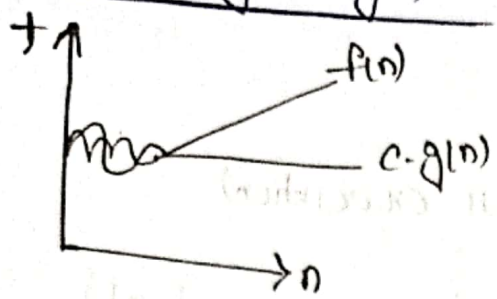
or, $2n^2+n \leq 3n^2$

or, $n \leq 3n^2 - 2n^2$

or, $n \leq n^2$ or, $1 \leq n$

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

Big-Ω (Big omega):



Best case

lower bound

$f(n) = \Omega(g(n))$

when $f(n) \geq c \cdot g(n)$

Ex: $f(n) = 2n^2 + n$

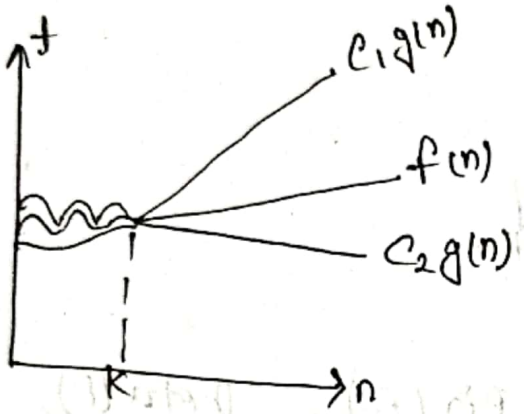
$f(n) = c \cdot g(n)$

or, $2n^2 + n \geq c \cdot n^2$

or, $2n^2 + n \geq 2n^2$

or, $n \geq 0$.

Theta (Θ) notation:



average case

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

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

Ex: $f(n) = \Theta(g(n))$

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

$2n^2 < 2n^2 + n < 3n^2$

Big O notation is widely used because it shows the worst and average case.

Divide and conquer

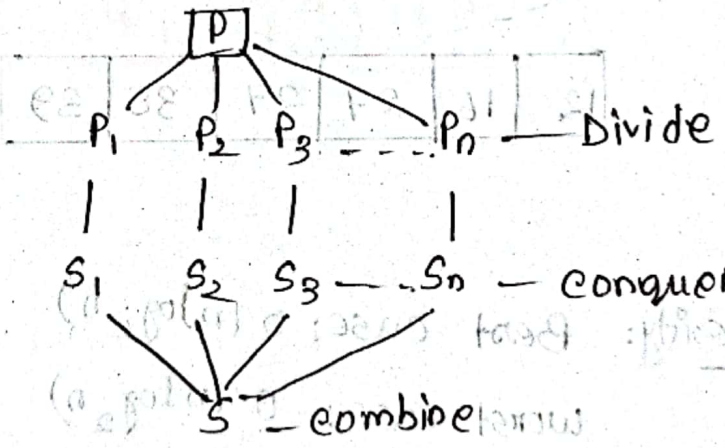
In this method, to solve a problem it's partitioned the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole.

Divide and conquer method-

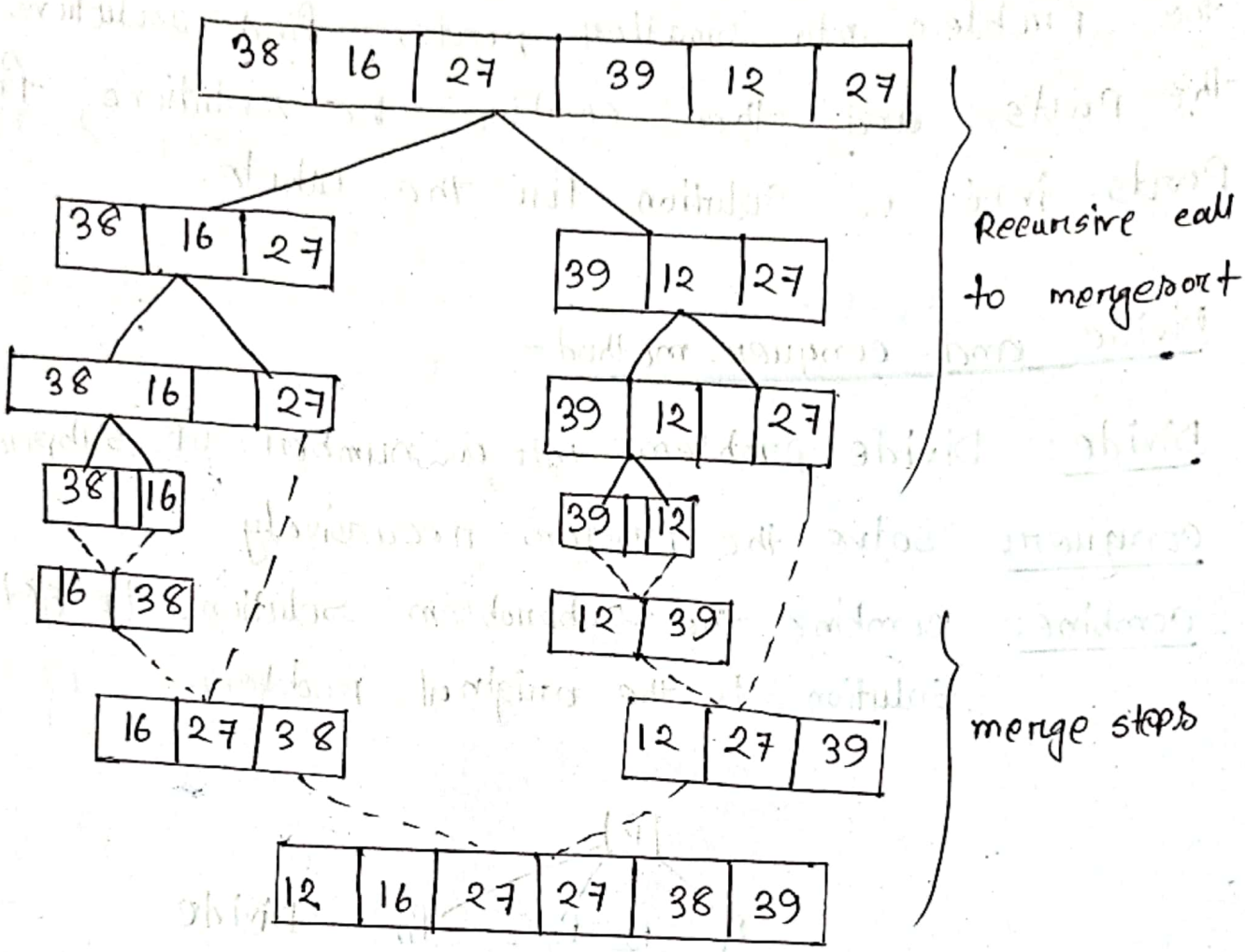
Divide: Divide problems into a number of subproblems

conquer: solve the problem recursively

combine: combine the subproblem solutions to get the solution to the original problem.



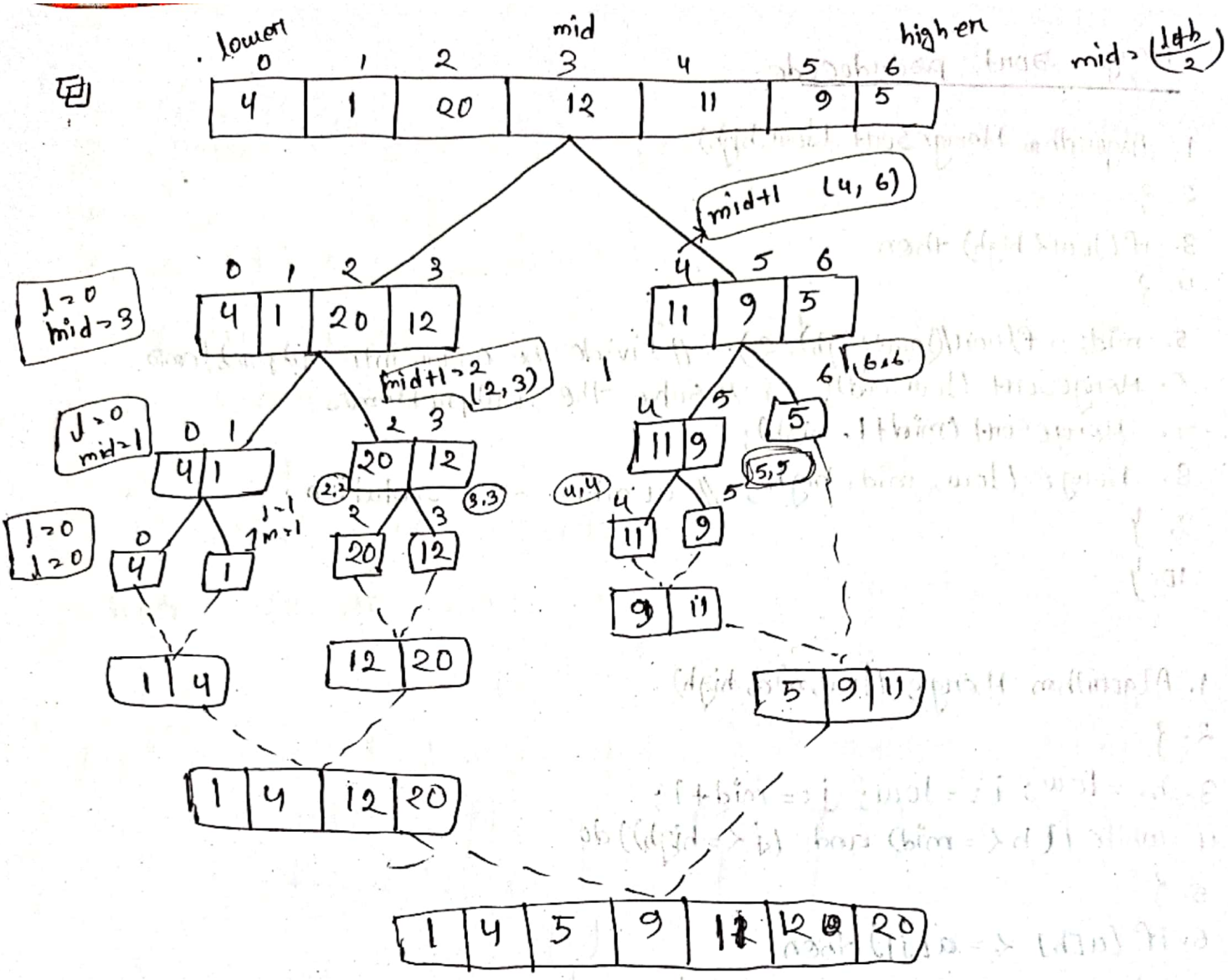
Merge Sort: (It follows divide-conquer method)



Time complexity: Best case: $O(n \log_2 n)$

Worst case: $O(n \log_2 n)$

Average : $O(n \log_2 n)$


lower
mid
higher
mid = (l+h)/2
0 1 2 3 4 5 6
4 1 20 12 11 9 5
mid+1 (4, 6)
l=0
mid=3
mid+1=2
(2,3)
6,6
5,5
4,4
2,3
3,3
l=0
l=0
l=1
m+1
1 4
12 20
5 9 11
1 4 12 20
1 4 5 9 12 20

Merge sort pseudocode:

```
1. Algorithm MergeSort (low, high)
2. {
3. if (low < high) then
4. {
5. mid := floor((low+high)/2); // Divide the array into subproblems
6. MergeSort (low, mid);       // Solve the subproblems
7. MergeSort (mid+1, high);
8. Merge (low, mid, high); // Combine the solution;
9. }
10. }
```

```
1. Algorithm Merge (low, mid, high)
2. {
3. h := low; i := low; j := mid+1;
4. while ((h <= mid) and (j <= high)) do
5. {
6. if (a[h] <= a[j]) then
7. { b[i] := a[h]; h := h+1; }
8. else
9. { b[i] := a[j]; j := j+1; }
10. i := i+1;
11. }
12. if (h > mid) then
13. for k := j to high do
14. {
15. b[i] := a[k]; i := i+1;
16. }
17. else
18. for k := h to mid do
19. {
20. b[i] := a[k]; i := i+1;
21. }
22. for k := low to high do a[k] := b[k];
23. }
```

Quick Sort:

Quick sort is based on the divide-and-conquer paradigm. Here is the three step divide-and-conquer process for sorting a typical subarray.

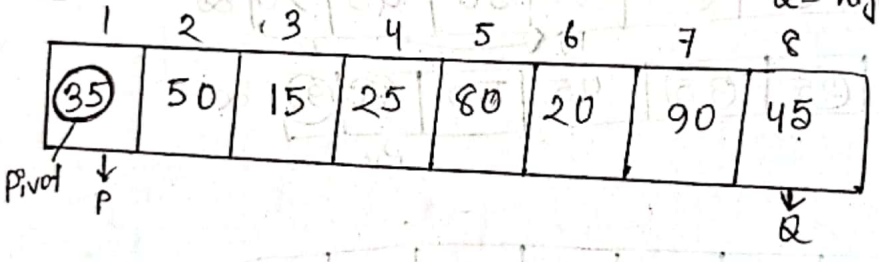
Divide: Partition the array $A[p \ldots r]$ into two non-empty sub-arrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$

Conquer: Sort the two subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ by recursive calls to quicksort

Combine: Since the subarray are sorted in place, no work is needed to combine them.

P_v = Pivot;
P = left to right
Q = right to left

1st element
Pivot



$P > P_v$ – P stop হবে, মানুষ আছে বোঝা যাবে

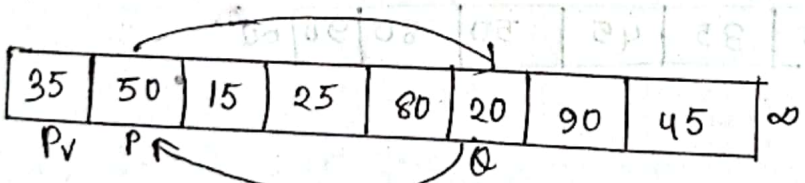
$Q \leq P_v$ – Q উইকানে বোঝাবে,

P, Q – Cross করলে P_v এর value Q তে যাবে, Q এর value P_v তে যাবে,

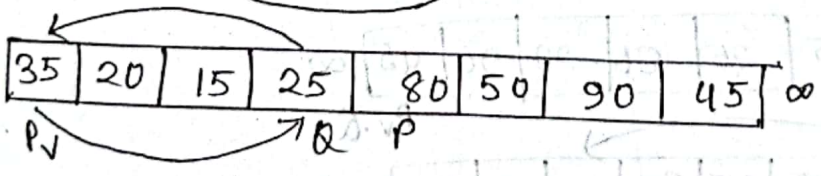
Cross না করলে P এর value Q তে Q এর value P তে যাবে,

1st partition:

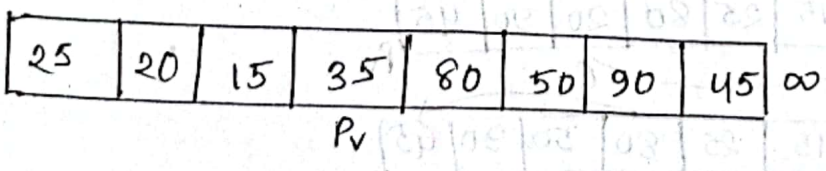
S-(i) –



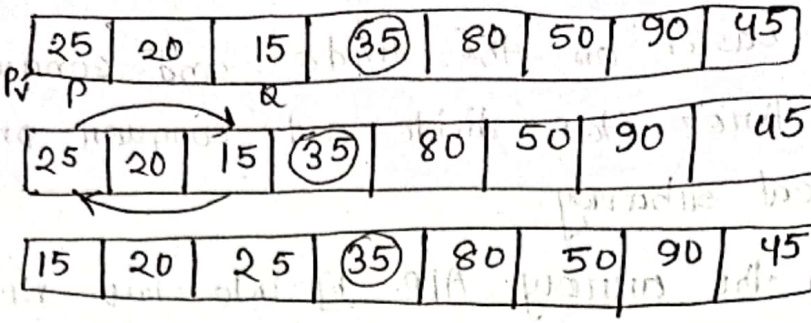
S-(ii) –



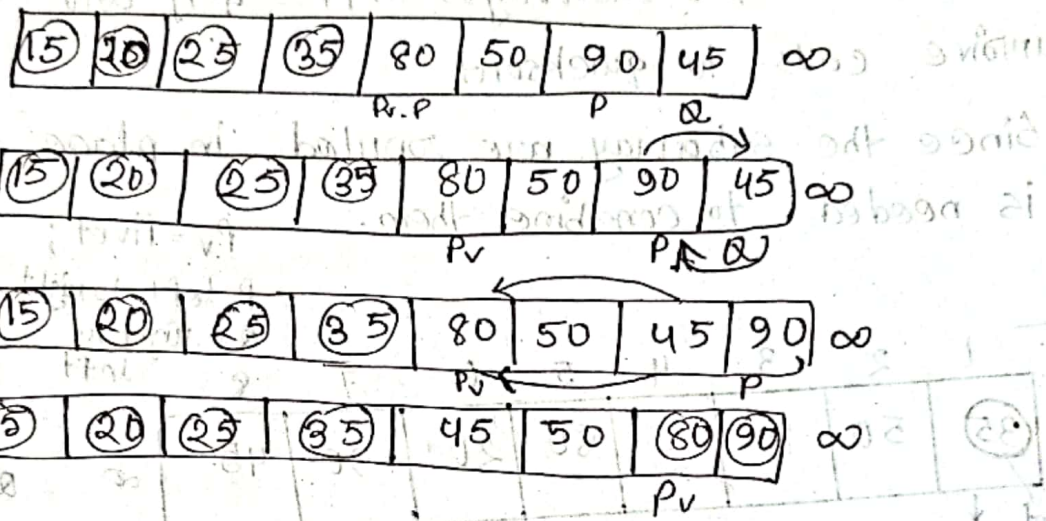
S-(iii) –



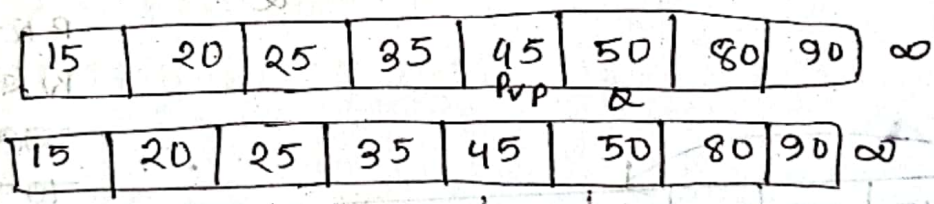
2nd partition:



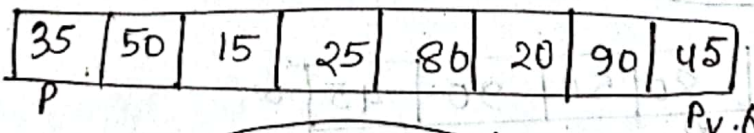
3rd partition:



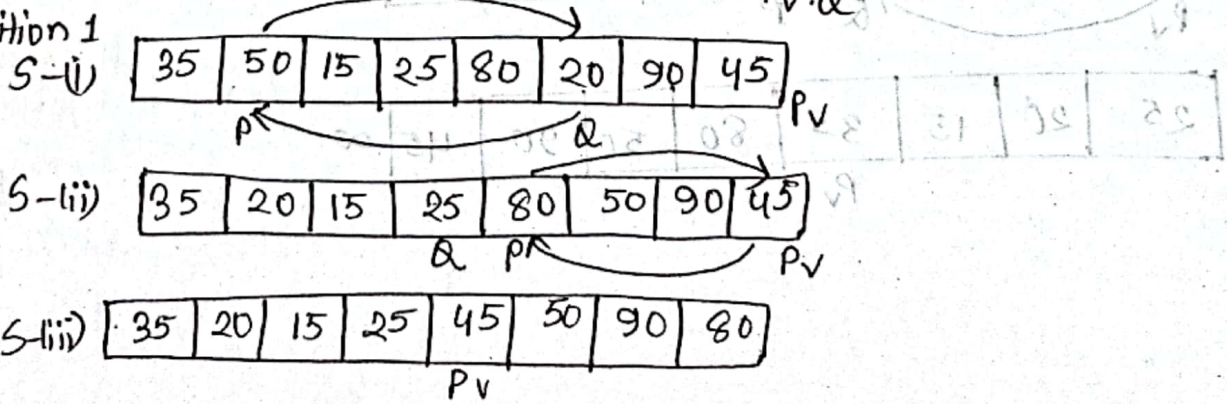
4th partition:



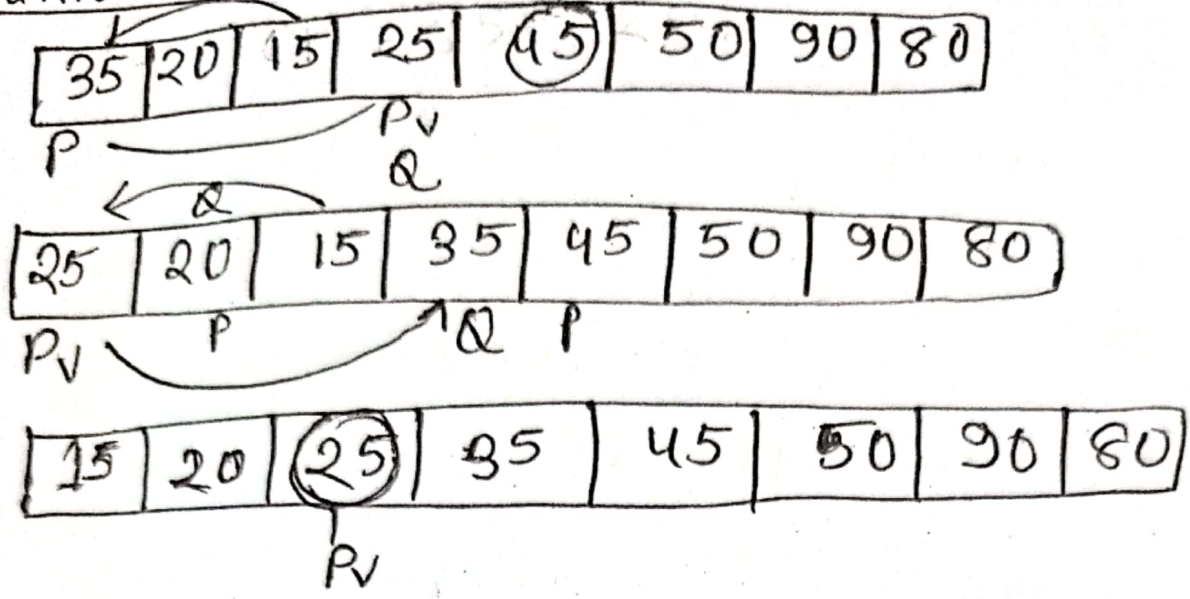
Last element Pivot



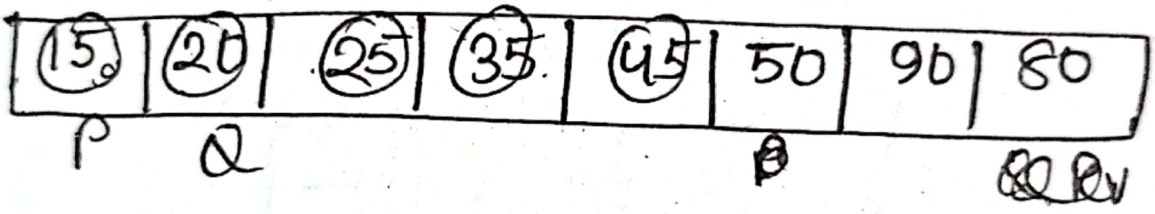
Partition 1



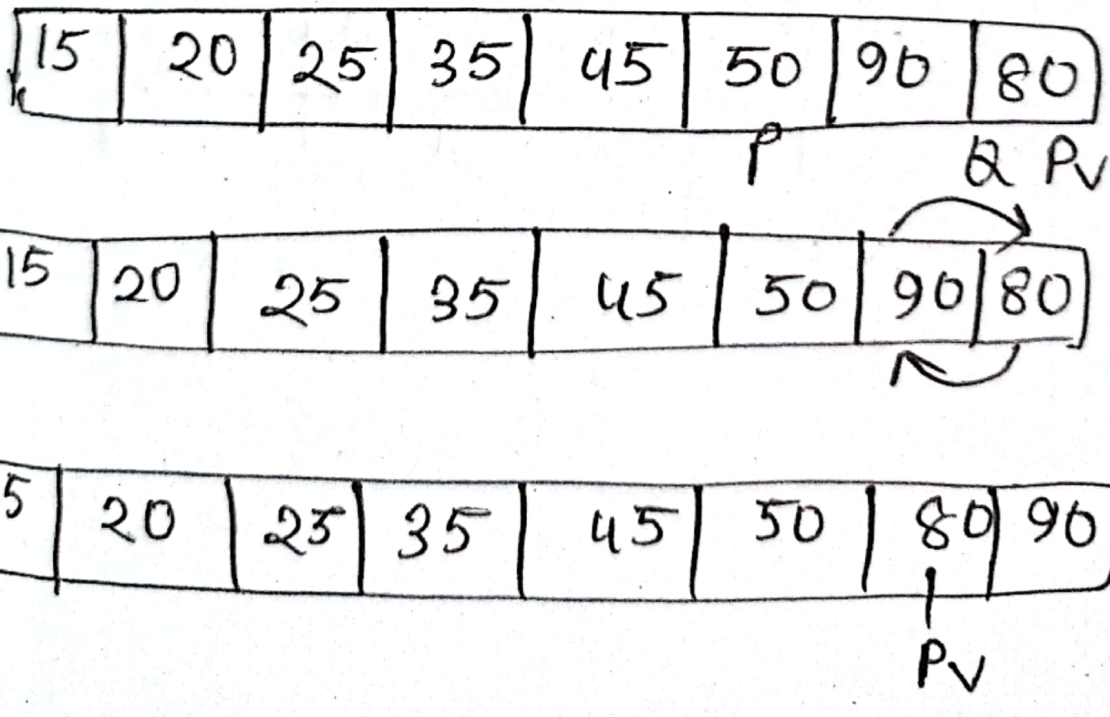
Partition-2



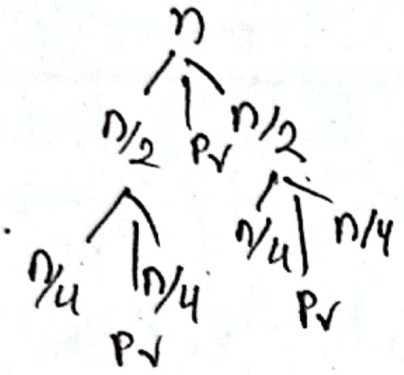
Partition-3.



Partition-4



Best case:



$\therefore \frac{n}{2^k} \geq 1$

$\Rightarrow n = 2^k$

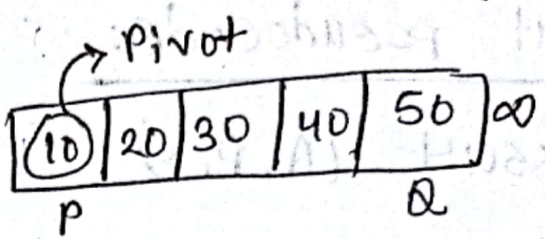
$\Rightarrow \log n = \log 2^k$

$\Rightarrow \log n = k \log 2$

$\Rightarrow k = \log n$

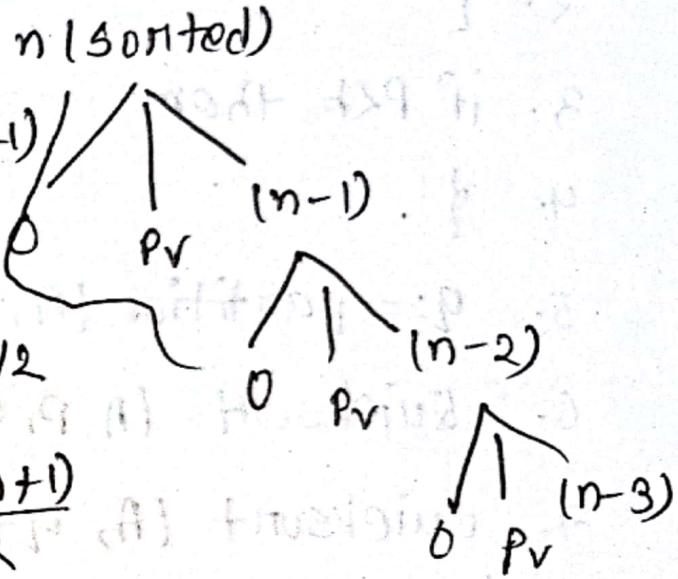
Complexity - $O(n \log n)$ / $O(n \log_2 n)$

Worst case: (Ascending order)



$P > P_v$

$Q \leq P_v$



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

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

$= (n-1)\, n/2$

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

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

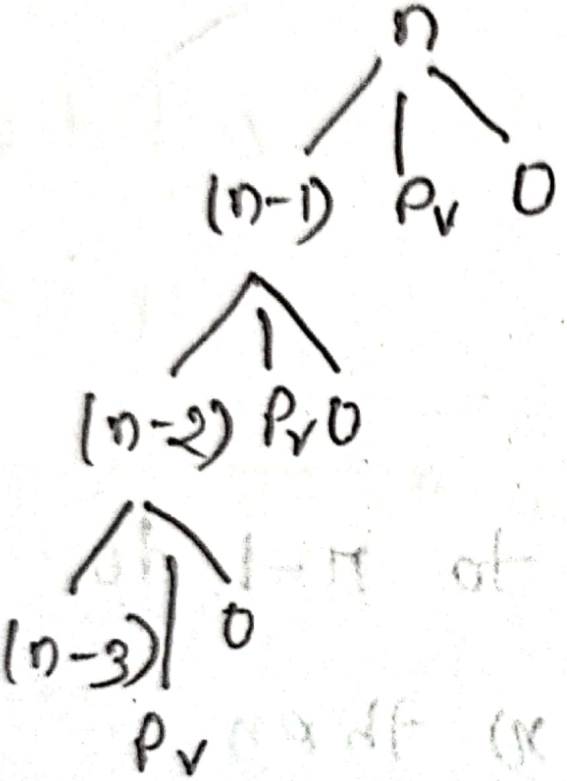
$\therefore$ complexity - $O(n^2)$

Recurrance Relation: $T(n-1)+n$

Recurrance relation : $T(n-1)+n$

Descending:

50	40	30	20	10	∞



Complexity : $O(n^2)$

Recurrence relation:

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

Quick sort pseudocode:

```
1. Quicksort (A, p, h)
2. {
3. if p<h then
4. {
5.  q:= partition (A, p, h);
6.  Quicksort (A, p, q-1);
7.  Quicksort (A, q+1, h);
8. }
9. }
```

④ To sort an entire array A, the initial call is Quicksort (A, 1, length [A]).

```
1. Partition (A, p, h)
2. {
3.  x:= A[h];
4.  i:= p-1;
5.  for j:= p to n-1 do
6.  if (A[j] <= x) then
7.  {
8.  i:= i+1;
9.  exchange A[i] <-> A[j];
10. }
11. exchange A[i+1] <-> A[h]
12. return i+1
13. }
```

Randomized version of Quick sort:

(Randomized Quick sort কখন use করি? কেন দরকার?)

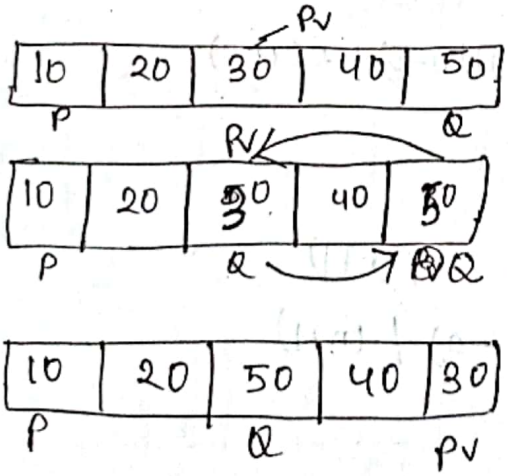
```
Randomized-Partition (A, P, h)
1. i = RANDOM (P, h)
2. exchange A[h] with A[i]
3. return PARTITION (A, P, h)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED - QUICKSORT (A, P, h)
1. if P<h
2. q = RANDOMIZED -PARTITION (A, P, h)
3. RANDOMIZED- QUICKSORT (A, P, q-1)
4. RANDOMIZED - QUICKSORT (A, q+1, h)
```

যেকোনো value কে PV হিসেবে (P, Q ছাড়া)

সাধারণত শেষে PV হিসেবা।



- Randomized quick sort is used to avoid worst case.
- It can solve the problem of all value same.
- Pivot is choosen randomly and cannot be first or last element.

In next step it will act as regular quick sort.

Time complexity $O(n \log_2 n)$.

Worst case analysis:

$C(1) = 0$

$C(2) = 1 + C(1) = 1$

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

$C(4) = 3 + C(3) = 3+2+1$

- - - - - - -

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

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

$= (n-1)n/2$

$= 1/2\, n^2 - 1/2\, n$

(sum of integers from 1 to n-1)

$= O(n^2)$

Average case analysis:

$C(n,p) = n-1 + C(p-1) + C(n-p) \quad [n \geq 2]$

$C(n) = n-1 + 2/n\,(C(0) + C(1) + \cdots + C(n-1))$

$C(n-1) = n-2 + 2/(n-1)\,(C(0) + C(1) + \cdots + C(n-2)) \quad [n > 2]$

$nC(n) - (n-1)C(n-1) = n(n-1) - (n-1)(n-2) + 2C(n-1)$

Rearrange:

$(C(n)-2)/(n+1) = (C(n-1)-2)/n + 2/(n+1)$

Let $S(n)$ be defined as $(C(n)-2)/(n+1)$

$S(n) = S(n-1) + 2/(n+1)$

Now, we can write,

$(c(n) - 2) / (n+1) = 2H_n + 1$

$(c(n) - 2) / (n+1) = 2 \ln n + O(1)$

$c(n) = (n+1) * 2 \ln n + (n+1) * O(1)$

$= 2n \ln n + O(n)$

$\therefore \ln n = (\ln 2)(\lg n)$ and $\ln 2 = 0.693$,

so that $c(n) = 1.386 n \lg n + O(n)$

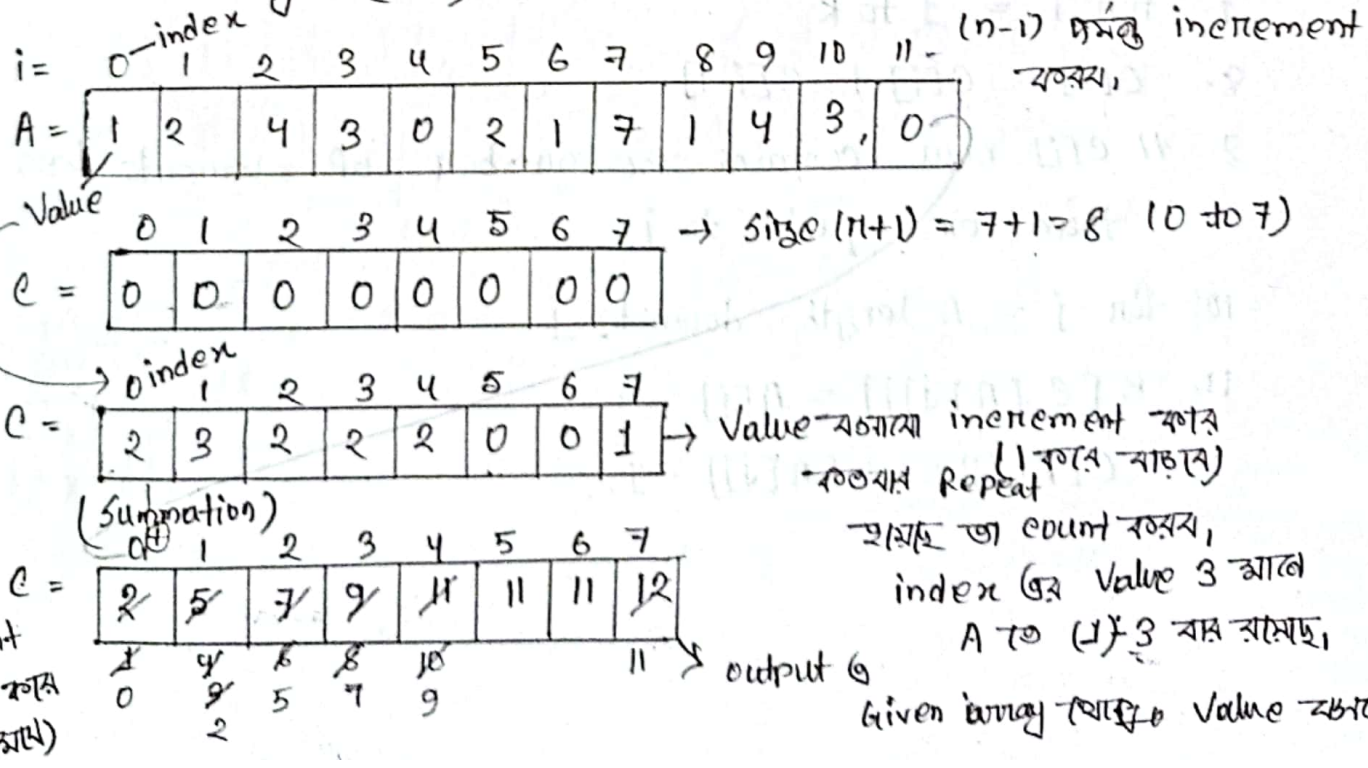
$= O(n \log n)$

Count Sort: Counting sort is one kind of linear algorithm which is not comparison based

Example: A = {1, 2, 8, 4, 3, 0, 2, 1, 7, 1, 4, 3, 0}

n = 12, R = 7 (Range) ~~(0-7)~~

Range - (0 - 7)



কতবার Repeat হয়েছে তা count করব,
index এর Value 3 আছে
A তে (1) 3 বার আসেছে,
Given array থেকে Value অনুযায়ে

output sorted array =

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

output এর আগে last এর আগে যেয়ে count করবে

Pseudocode:

```
COUNTING-SORT (A, B, k)
1. let C[0...k] be a new array
2. for i = 0 to k
3.     C[i] = 0
4. for j = 1 to A.length
5.     C[A[j]] = C[A[j]] + 1
6. // C[i] now contains the number of elements equal
       to i.
7. for i = 1 to k
8.     C[i] = C[i] + C[i-1]
9. // C[i] now contains the number of elements less
       than or equal to i.
10. for j = A.length down to 1
11.     B[C[A[j]]] = A[j]
12.     C[A[j]] = C[A[j]] - 1.
```

Advantage of counting sort:

⇒ Counting sort generally performs faster than all comparison based sorting algorithm

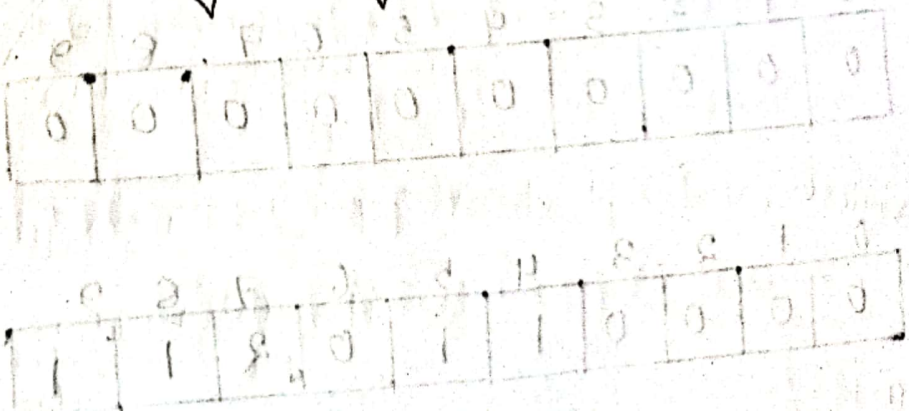
⇒ Counting sort is easy to code.

⇒ They can help to organize and analyze data in a more efficient and meaningful way.

Disadvantage of counting sort:

⇒ Counting sort doesn't work on decimal values.

⇒ Counting sort is inefficient if the range of values to be sorted is very large.



④ Consider an array for merge sort:

$$A = \{8, 4, 5, 2, 7, 3\}$$

(i) How many merge sort function call will be there when the initial call is array A? – 11

(ii) How many merge operation will be there? – 5

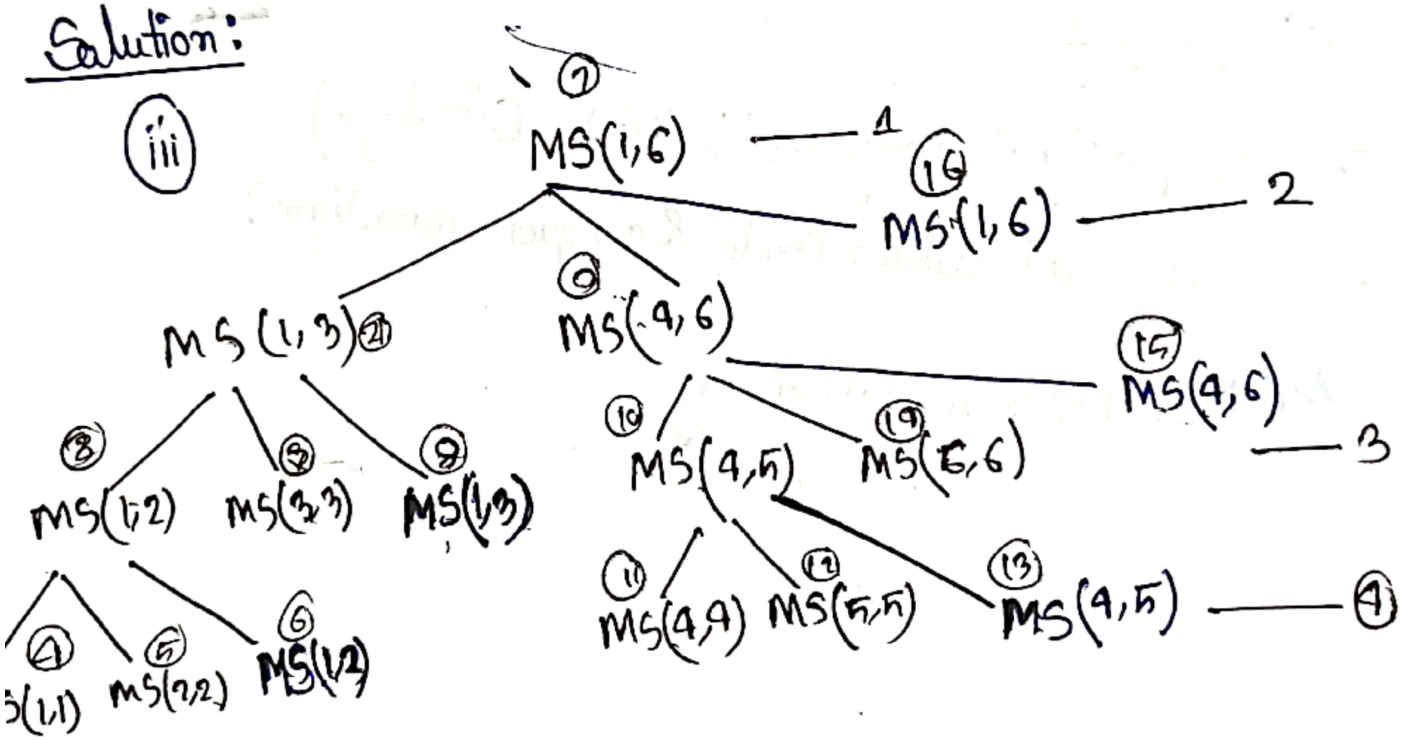
(iii) Show the function calls of the merge sort algorithm for the array using data structure's stack?

(iv) What ~~does~~ would be the depth of the stack required for this case? – 4

(v) How many recursive call will be there? – 16

Solution:

(iii)



Radix sort:

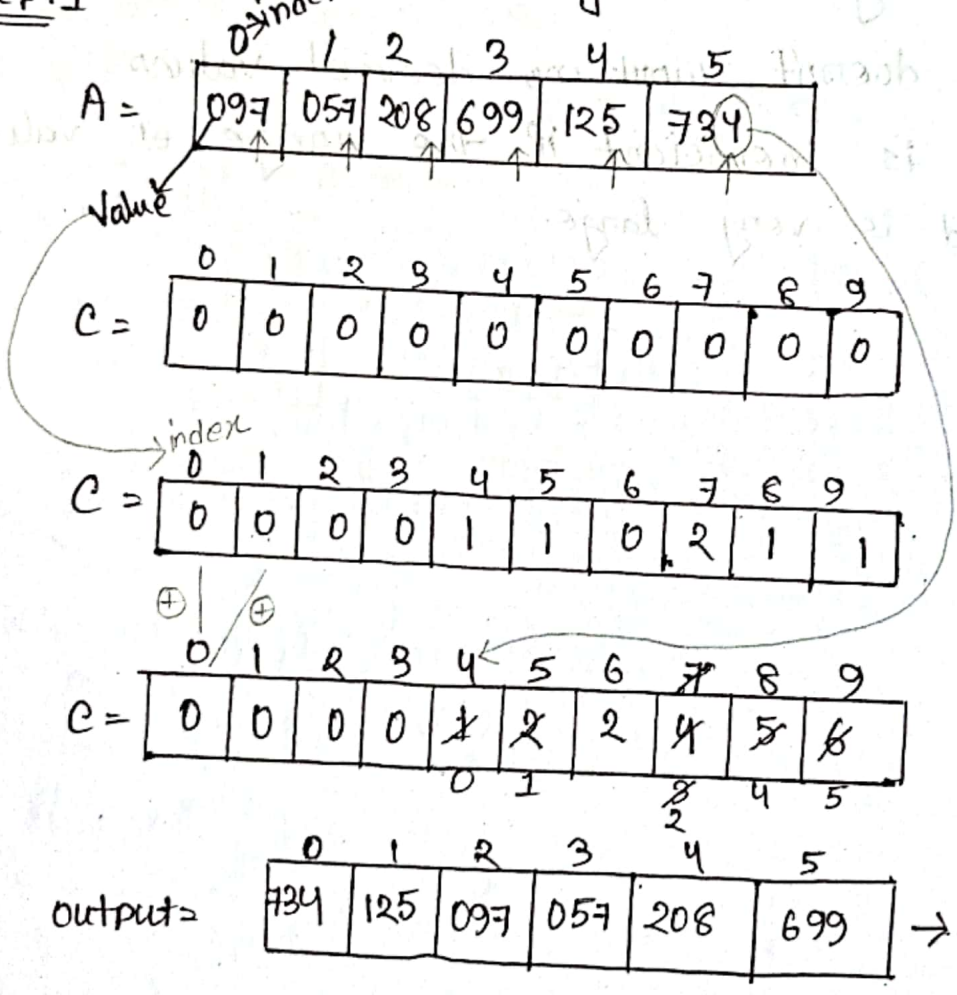
Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums.

Radix sort is a non-comparative sorting algorithm.

Example: A = { 97, 57, 208, 699, 125, 734 }
(2, 2, 3, 3, 3, 3)

Step: 1

n = 6, Range = 9



maximum digit 3 তাই 1st ও 2টার আগে 0 নিছি।

Step-1 ও বলছে করব 1st ও তার পাশের digit গুলা নিয়ে।

→ Output ও Given array থেকে Value বসাবে যা তার পাশ থেকে count করবে

Step-2 to middle digit বিসে বসাব বসবে.

Step-2

A =

0	1	2	3	4	5
734	125	097	057	208	699

C =

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

C =

0	1	2	3	4	5	6	7	8	9
1	0	1	1	0	1	0	0	0	2

C =

0	1	2	3	4	5	6	7	8	9
1	1	2	3	3	4	4	4	4	6

0 1 2 3 5 4

output =

0	1	2	3	4	5
208	125	734	057	097	699

Step-3

Step-3 – last digit বিসে বসাব বসবে

A =

0	1	2	3	4	5
208	125	734	057	097	699

C =

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

C =

0	1	2	3	4	5	6	7	8	9
2	1	1	0	0	0	1	1	0	0

C =

0	1	2	3	4	5	6	7	8	9
2	3	4	4	4	4	5	6	6	6

1 2 3 4 5
0

output =

0	1	2	3	4	5
057	097	125	208	699	734

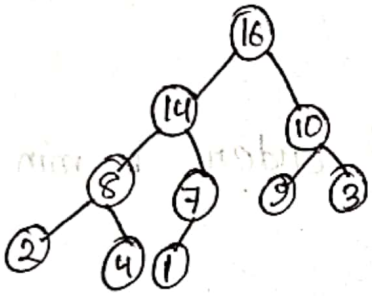
Pseudocode of Radix sort:

```
Radix_Sort (A, d)
 for j=1 to d do
 int count [10] ={0};
for i=0 to n do
  count [key of (A[i]) in pass j] ++
for k=1 to 10 do
    count [k] = count[k] + count [k-1]
  for i=n-1 down to 0 do
    result [count [key of (A[i])]] = A[j]
    count [key of (A[i])] --
    for i=0 to n do
    A[i] =result[i]
    end for (j)
  end func
```

Heapsort: A heap is a tree where in each node is of a value equal to or less than its parent.

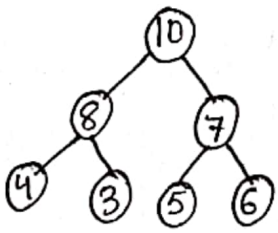
⇒ A binary tree is used which adds the constraint that no parent may have more than two nodes. This allows heapsort to run within its own array, without additional space complexity.

⇒ A heap can be seen as a complete binary tree.

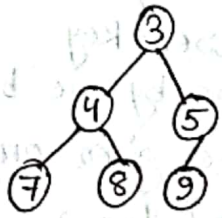


Heap tree → ① Structure property (CBT)
② Ordering property (Max heap, Min heap)

Max heap: Parent > child



Min heap: Parent < child



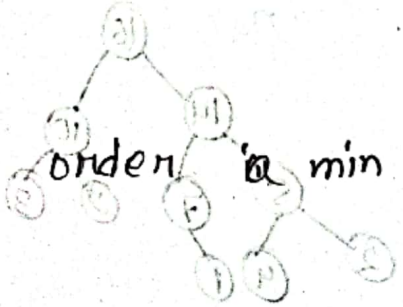
Heap height: The height of a node in the heap
= the number of edges on the longest downward path from the node to a leaf

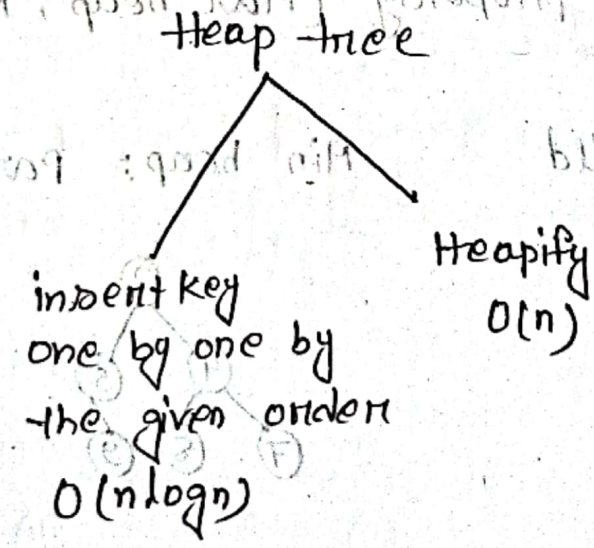
⇒ The height of a heap = the height of its root.

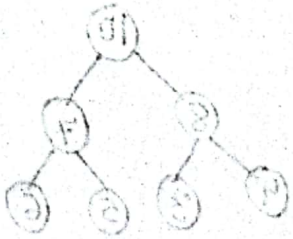
Q What is the height of an n-element heap? Why? (O(log n))

Hint ⇒ The height is defined as the number of edges in the longest simple path from the root.

Q Is an array that is in sorted order a min heap?







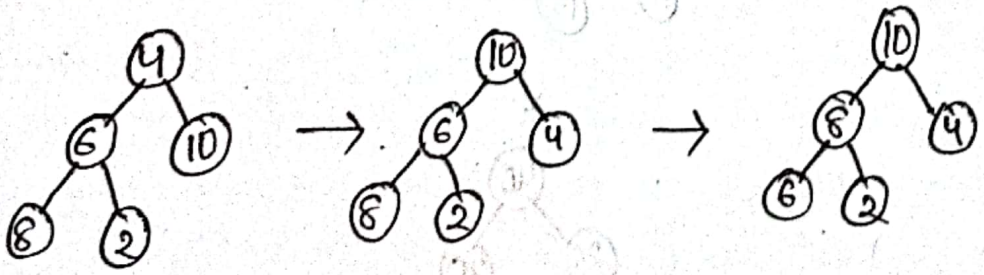
Heap tree construction:

- Insert key (one by one given order)
- Heapify O(n) — [Max heapify / min heapify]

Heap operations: Max heapify ()

```
Max_heapify (A, i)
{
  l = left (i);
  r = Right (i);
  if (l <= heap-size (A) && A[l] > A[i])
    largest = l;
  else
    largest = i;
  if (r <= heap-size (A) && A[r] > A[largest])
    largest = r;
  if (largest != i)
    exchange A[i] ↔ A[largest]
    Max_heapify (A, largest);
```

Max heapify Example:



Heap operations : Deletion

Extract max heap

Extract min heap

For extracting, either delete root or the last element. can't delete any element from middle as it will not maintain complete Binary tree.

when, insertion - Bottom to Top ; swap (min or max)

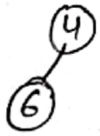
Deletion - Top to bottom ; swap (min or max)

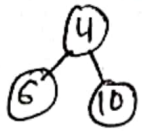
Heap sort
- Heapify (O(n))
- Deletion (O(nlogn))

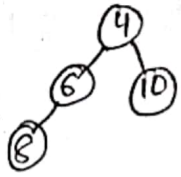
Example: A = | 4 | 6 | 10 | 8 | 2 |

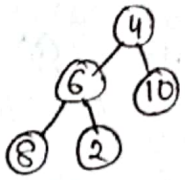
Insertion in tree:

1. (4)

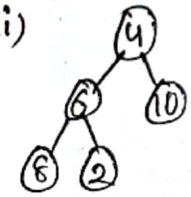
2. 

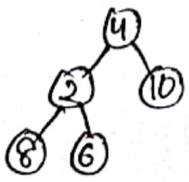
3. 

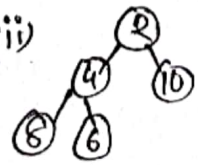
4. 

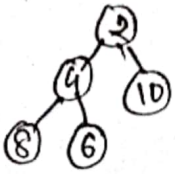
5. 

Min heapify:

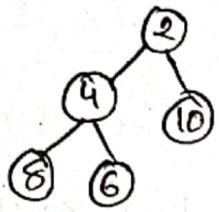
(i) 

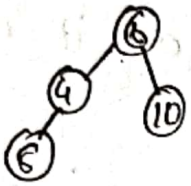
(ii) 

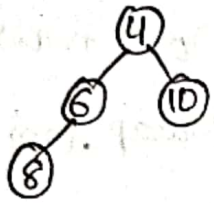
(iii) 

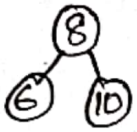
(iv) 

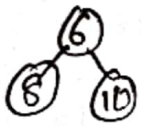
Deletion:



(i)


(ii)


(iii)


(iv)


(v)


(vi)


(vii) 10

(viii)

(i) 2

(iii) 2 4

(v) 2 4 6

(vii) 2 4 6 8

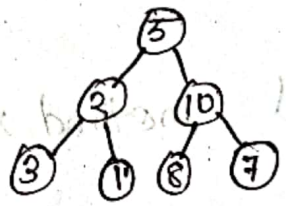
(viii) 2 4 6 8 10

⇒ It doesn't take any extra space that's why it is known as inplace algorithm.

⇒ It is not possible to maintain the index as same as given array in sorting array that's why it is an unstable algorithm

⇒ A max heap can be viewed as

① Binary tree –



② Array – 5, 2, 3, 1, 10, 8, 7

Recurrence Relation: A recurrence relation is a mathematical equation or formula that follows that follows a sequence of values based on previous term in the sequence.

In computer science, recurrence relation is used to describe the time complexity of recursive algorithm.

There are 4 general methods for solving recurrences.

1. Back substitute method
2. Recursion tree method
3. Master theorem
- Iteration (loop)

Back substitute method: Use mathematical induction to find the constants and verify solution.

Example-1: $T(n) = n + T(n-1)$; if $(n>1)$

Soln: $T(n) = n + T(n-1)$ ——— (i)

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

$= (n-1) + T(n-2)$ ——— (ii)

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

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

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

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

or, $T(n) = 3n-3 + T(n-3) + \cdots + (n-k) + T(n-k-1)$

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

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

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

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

To terminate:
Let,
$n-k-1 = 1$
or, $n-k = 2$

Ex: 2

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

Soln: $T(n) = n * T(n-1)$ —— (i)

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

$= (n-1) * T(n-2)$ —— (ii)

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

$= (n-2) * T(n-3)$ —— (iii)

Now,

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

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

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

$= n * (n-1) * (n-2) * \cdots (n-k) * T(n-k-1)$

$= n * (n-1) * (n-2) * \cdots 2 * 1$

$= n!$

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

Let,
$n-k-1 = 1$
or, $n-k = 2$

Ex: 3 $T(n) = T(n/2) + c$

Soln: $T(n) = T(n/2) + c$ —①

$T(n/2) = T(n/4) + c$ —②

$T(n/4) = T(n/8) + c$ —③

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

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

$= T(n/8) + c + c + c$

$= T(n/8) + 3c$

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

⋮

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

$= T(n/n) + kc$

$= T(1) + \log_2 n$

$\therefore T(n) = O(\log_2 n)$

Let,

$2^k = n$

or, $k = \log_2 n$

Master theorem method:

Formula: ① $T(n) = aT(n/b) + \theta(n^k \log^p n)$

$a \geq 1$, $b > 1$, $k \geq 0$ and is a real number

(i) if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

(ii) if $a = b^k$

⇒ if $p > -1$ then, $T(n) = \theta(n^{\log_b a} \cdot \log^{p+1} n)$

⇒ if $p = -1$ then $T(n) = \theta(n^{\log_b a} \cdot \log \log n)$

⇒ if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

(iii) if $a < b^k$

⇒ if $p \geq 0$ then $T(n) = \theta(n^k \log^p n)$

⇒ if $p < 0$ then $T(n) = \theta(n^k)$

Master theorem cannot be apply in every function.

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

Here,
$a = 3$
$b = 2$
$k = 2$
$b^k = 2^2 = 4 > a$

$T(n) = \theta(n^k \log^p n)$

$= \theta(n^2 \log^0 n)$

$= \theta(n^2 \cdot 1)$

$= \theta(n^2)$

Ex: 2

$T(n) = 9T(n/3) + n$

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

$= \theta(n^{\log_3 9})$

$= \theta(n^{\log_3 3^2})$

$= \theta(n^2 \log_3 3)$

$= \theta(n^2)$

Here,

$a = 9$

$b = 3$

$k = 1$

$p = 0$

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

$\Rightarrow$ Master theorem cannot be applied.

Ex: 4 $T(n) = 4T(n/16) + \sqrt{n}$

$T(n) = \theta(n^{\log_b a} \cdot \log^{p+1} n)$

$= \theta(n^{\log_{16} 4} \cdot \log n^1)$

$= \theta(n^{\log_{16} 16^{1/2}} \cdot \log n)$

$= \theta(n^{1/2 \log_{16} 16} \cdot \log n)$

$= \theta(\sqrt{n} \cdot \log n)$

$a = 4$

$b = 16$

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

$p = 0 > -1$

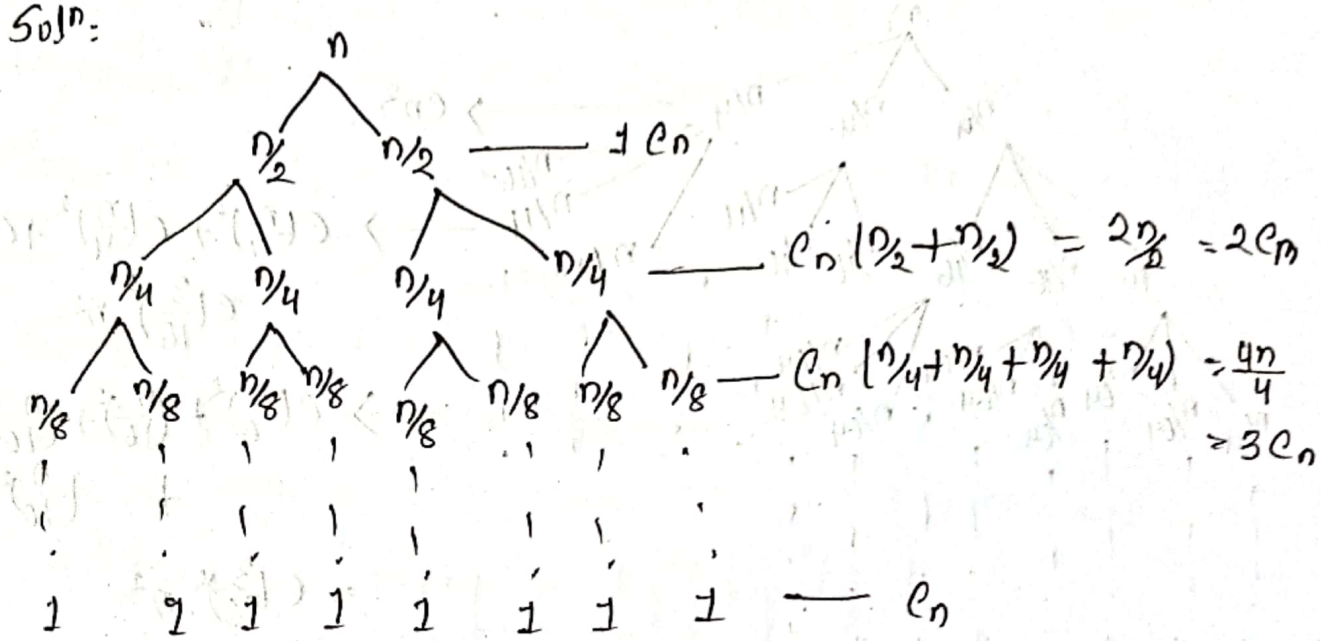
$b^k = \sqrt{16} = 4 = a$

Recursion tree method:

Ex: 01

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

Soln:

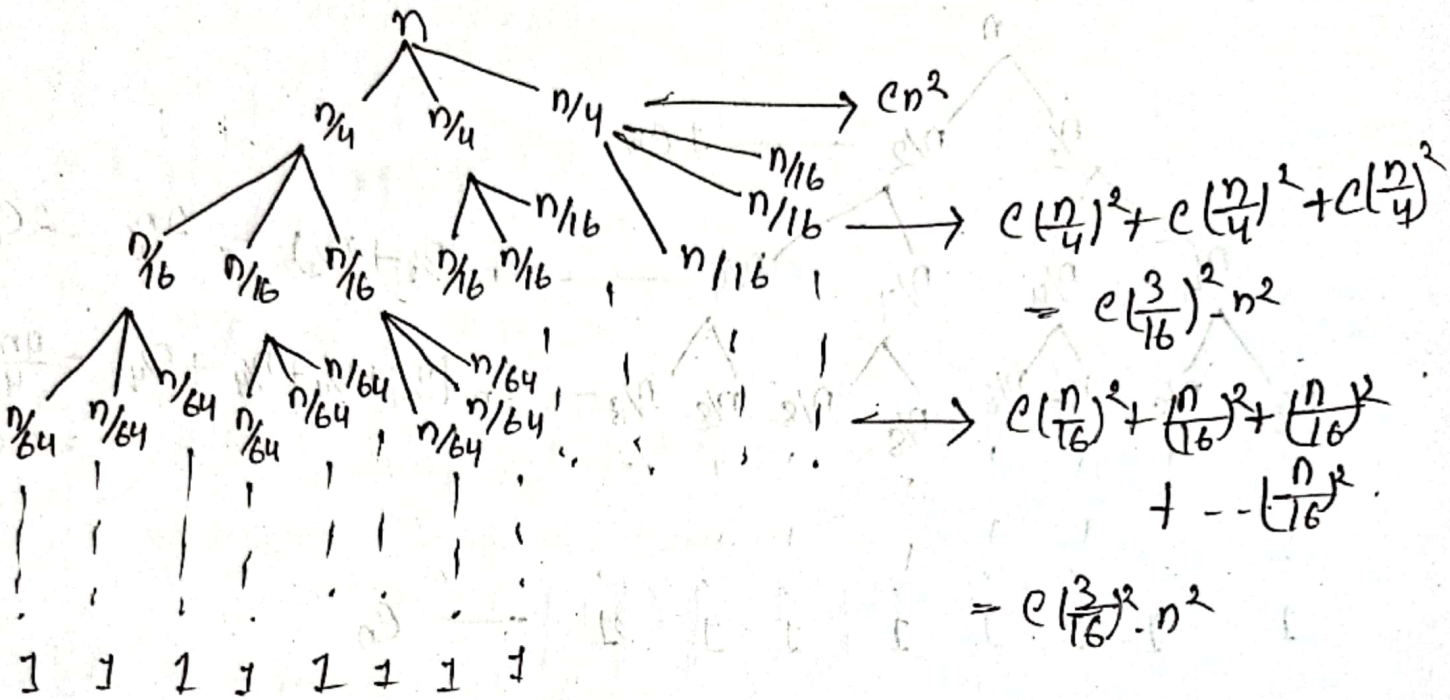


$$T(n) = c_n \log n$$

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

Ex: 02

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



$T(n) = cn^2 + c(\frac{3}{16})n^2 + c(\frac{3}{16})^2 \cdot n^2 + \cdots$

$= cn^2 (1 + \frac{3}{16} + (\frac{3}{16})^2 + \cdots)$

$= cn^2 (1 + r + r^2 + \cdots)$ $\left[\frac{1}{1-r} ; r<1\right]$

$= cn^2 \left(\text{for } \frac{1}{1-\frac{3}{16}}\right)$

$= cn^2 \cdot \frac{16}{13}$

$= O(n^2)$

Dynamic Programming

Dynamic programming is a general algorithm design technique which solves optimization problems by combining solutions to subproblems

Dynamic programming is applicable when subproblems are not independent.

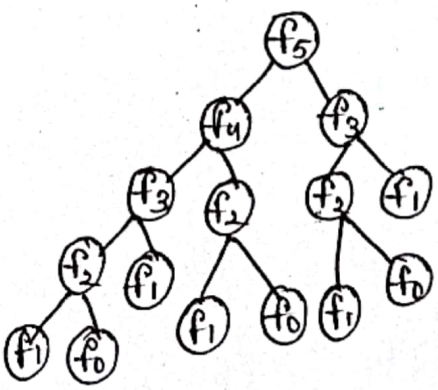
→ subproblems share subproblems

→ solve every subproblem only once and store the answer for use when it reappears.

→ Dynamic programming follows both Top down approach and Bottom up approach.

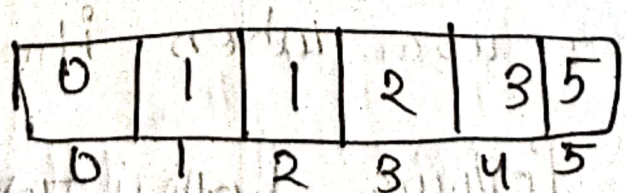
Top down approach: (উপর থেকে নিচে Value বসাতে থাকে)

$$fib(n) = 0, \text{ if } n = 0$$
$$= 1, \text{ if } n = 1$$
$$= fib(n-2) + fib(n-1), \text{ if } n > 1$$



Bottom up approach:

```
main ()
{
  int i, f[10];
  f[0] = 0;
  f[1] = 1;
  for (i=2; i<=n; i++)
  {
    f[i] = f[i-1] + f[i-2];
  }
}
```



Elements of Dynamic Programming:

Development of an dynamic programming solution to an optimization problem involves four steps.

(1) Characterize the structure of an optimal solution
- Optimal substructures, where an optimal solution consists of sub-sol that are optimal.
- Overlapping sub-problems where the space of sub-problems is small in the sense that the algorithm solves the same sub-p lems over and over rather than generating new sub-proble

(2) Recursively defined the value of an optimal solution.
- define the value of an optimal solution based on value solutions to sub-problems.

(3) Compute the value of a optimal solution in a bottom-u manner.
- compute in bottom-up fashion and save the values along the way.
- later steps use the save values of previous steps.

(4) Construct an optimal solution from the computed optima value.

Difference between dynamic programming and divide and conquer method:

Dynamic programming	Divide and conquer
① It is used to solve a range of overlapping subproblems.	① It is used to solve complicated problems.
② It is non-recursive	② It is a form of recursive.
③ It is a bottom up and top down approach.	③ It is top down approach
④ It is faster	④ It is completely slower.
⑤ It is more efficient.	⑤ It is less efficient

Difference between Top down approach and bottom up approach:

Top down approach	Bottom up approach
① The problem is broken down into smaller parts.	① The smaller problems are solved.
② It doesn't require communication between modules.	② It requires relatively more communication between modules.
③ Decomposition approach is used here	③ Composition approach is used here.
④ The implementation depends on the programming platform	④ Data encapsulation and data hiding is implemented in this approach.
⑤ It is generally used with module and debugging code.	⑤ It is generally used in testing modules.

Difference between memorization and tabulation.

	Memorization	Tabulation
State	State transition is easy to think	State transition relation is difficult to think
Code	code is easy and less complicated	code gets complicated when lot of condition are required
Speed	Slow, due to lot of recursive calls and return statement	fast, as we directly access previous states from the table.
Subproblem solving	The memorized solution has the advantage of solving only those subproblems that are definitely required	the tabulation method has the advantage of solve those problem that must be solved atleast once.
Table entries	All entries of the lookup table are not necessarily filled in memorized version.	In tabulated version starting from the first entry, all entries are filled one by one.

Matrix Chain Multiplication:

with this, it chose the less operational process.

Formula:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k j} & m[i,k] + m[k+1,j] + P_{i-1} P_k P_j ; \quad i < j \end{cases}$$

i = Row (1 to n-1)

j = col (1 to n-1)

Exp: Given four matrixes with dimensions

$A_1 (P_0 \times P_1)$; $A_2 (P_1 \times P_2)$; $A_3 (P_2 \times P_3)$; $A_4 (P_3 \times P_4)$

where, $P_0 = 5$, $P_1 = 4$, $P_2 = 6$, $P_3 = 2$, $P_4 = 7$

Solution:

Now, $m(1,2) = m(1,1) + m(2,2) + P_0 P_1 P_2$

$= 0 + 0 + 5 \times 4 \times 6$

$= 120$

$\therefore m(2,3) = m(2,2) + m(3,3) + P_1 P_2 P_3$

$= 0 + 0 + 4 \times 6 \times 2$

$= 48$

$\therefore m(3,4) = m(3,3) + m(4,4) + P_2 P_3 P_4$

$= 0 + 0 + 6 \times 2 \times 7$

$= 84$

$\therefore m(2,4) = m(2,2) + m(3,4) + P_1 P_2 P_4 \quad ; k = 2$

$= 0 + 84 + 4 \times 6 \times 7$

$= 252$

$\therefore m(2,4) = m(2,3) + m(4,4) + P_1 P_3 P_4 \quad ; k = 3$

$= 48 + 0 + 4 \times 2 \times 7$

$= 104$

we will take 104.

$\therefore m(1,3) = m(1,1) + m(2,3) + P_0 P_1 P_3 \quad ; k = 1$

$= 0 + 48 + 5 \times 4 \times 2$

$= 88$

$\therefore m(1,3) = m(1,2) + m(3,3) + P_0 P_2 P_3 \quad ; k = 2$

$= 120 + 0 + 5 \times 6 \times 2$

$= 180$

we will take 88.

$\therefore m(1,4) = m(1,1) + m(2,4) + P_0 P_1 P_4 \quad ; k = 1$

$= 0 + 104 + 5 \times 4 \times 7$

$= 244$

$\therefore m(1,4) = m(1,2) + m(3,4) + P_0 P_2 P_4 \quad ; k = 2$

$= 120 + 84 + 5 \times 6 \times 7$

$= 414$

$\therefore m(1,4) = m(1,3) + m(4,4) + P_0 P_3 P_4 \quad ; k = 3$

$= 88 + 0 + 5 \times 2 \times 7$

$= 158$

$\therefore$ we will take 158

i \ j →	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

M

S	1	2	3	4
1	D	1	1	3
2		0	2	3
3			0	3
4				0

i = 3, ((A_1 A_2 A_3) A_4)

i = 1 ; (((A_1) A_2 A_3) A_4)

i = 1 ; (((A_1) A_2 A_3) A_4)

i = 1 ; ((A_1 (A_2 A_3)) A_4)

Longest common subsequence:

Given two string ABCDBAB, BDCABA.

Solution:

X = ABCDBAB — (length 7)

Y = BDCABA — (length 6)

i-		j- 0	B 1	D 2	C 3	A 4	B 5	A 6
	0	0	0	0	0	0	0	0
A	1	0	←0↑	0↑	0↑	~~0↑~~	←1	↖1
B	2	0	↖(1)	←1	←1	1↑	↖2	←2
C	3	0	(1↑)	1↑	↖2	←2	2↑	2↑
D	4	0	1↑	↖(2)	(2↑)	(2↑)	2↑	2↑
B	5	0	↖1	2↑	2↑	2↑	↖(3)	←3
A	6	0	1↑	2↑	←2	↖3	3↑	↖(4)
B	7	0	↖1	2↑	←2	3↑	↖4	(4↑)

Solution: BDBA