1. Asymptotic Analysis

(Q.1) Find the complexity of the program below.

O(1)

```
function(int n)
{
  if(n==1)          ——— constant.
    return;
  for(int i=1; i<=n; i++)          ——— n times
  {
    for(int j=1; j<=n ; j++)       ——— n times
    {
      printf("*");
      break;                       ——— but once due to 'break'.
    }
  }
}
```

→ Even though the inner loop is bounded by 'n', due to the break statement it is executing only once. Therefore the time complexity of above function is O(n);.

→ Space Complexity = O(1).

(Q.2) Find the time complexity of given code.

```
void fun(int n, int x) {
    for (int i=1; i<n; i=i*x)
        cout<<"*";
}
```

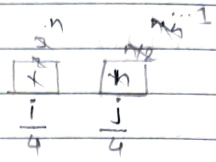
→ (i=i*x) is equal to dividing the i by x, 'n' times.

∴ Time complexity is $O(\log_x n)$.

→ Space Complexity = O(1).

Q.3) Provide the time complexity of the following code:

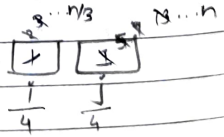
```
void fun(int n).{
    int i=1;
    while (i<n) {
        int j=n;
        while (j>0) {
            j = j/2
        }
        i= i*2;
    }
}
```



→ In each iteration, 'i' becomes twice & 'j' becomes half. Both are multiplied or divided by the power of 2.
∴ Time complexity of the code is $O(\log n * \log n) = O(\log^2 n)$.
→ Space complexity = $O(1)$.

Q.4) Find the time complexity of below function.

```
function (int n) {
    for (int i=1; i<=n/3; i++)          ——— n/3 times
        for (int j=1; j<=n; j=j+4)      ——— n/4 times
            printf("*");
}
```



→ The outer for loop will run total 'n/3' times which is O(n).
The inner for loop will run total 'n/4' times which is O(n).
∴ Time complexity = $O(n*n) = O(n^2)$.
→ Space complexity = $O(1)$.

X——X

Chinmay (142203003)

classmate

Date

Page

2. Recurrence Relations. (SM).

Q.1) Find the Time complexity using substitution method.

$T(n) = \sqrt{2} \cdot T(n/2) + \sqrt{n}$; $T(1) = 1$. $\left(\frac{n}{2}\right) \to n$

→ $T(n) = \sqrt{2} \cdot T(n/2) + \sqrt{n}$

$= \sqrt{2} \cdot [\sqrt{2} \cdot T(n/4) + \sqrt{n/2}] + \sqrt{n}$

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

$= 2[\sqrt{2} \cdot T(n/8) + \sqrt{n/4}] + \sqrt{2} \cdot \sqrt{n/2} + \sqrt{n}$.

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

$= \sqrt{2^3} \cdot T(n/8) + \sqrt{n} + \sqrt{n} + \sqrt{n}$.

$= \sqrt{2^3} \cdot T(n/2^3) + 3\sqrt{n}$.

$\vdots$

$= \sqrt{2^k} \cdot T(n/2^k) + k\sqrt{n}$

$= \sqrt{2^{\log n}} + \log n \sqrt{n}$.

$= \sqrt{n} + \log n \cdot \sqrt{n}$

$= \sqrt{n}(\log n + 1)$.

Q.2) Find the time complexity using substitution method.

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

→ $T(n) = T(n-1) + n$.

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

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

$= T(n-4) + (n-3) + (n-2) + (n-1) + n$.

$\vdots$

$= T(n-k) + (n-(k-1)) + \dots + n$.

Let $n - k = 1$.

$= T(1) + 2 + 3 + \dots + n$.

$= 1 + 2 + 3 + \dots + n$.

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

Q.3) Solve the following recurrence relation using substitution method.

→ $T(n) = T(n-1) + \log n$.

$= [T(n-2) + \log(n-1)] + \log n$.

$= [T(n-3) + \log(n-2)] + \log(n-1) + \log n$.

⋮

$= T(n-K) + \log(n-(K-1)) + \cdots + \log n$.

$= T(1) + \log 2 + \log 3 + \cdots + \log n$.

$= \log(1 \cdot 2 \cdot 3 \cdots n)$

$= \log(n!)$ — $\because n! = n^n$.

$= O(n \log n)$.

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

$\therefore T(n/2) = 2T(n/4) + n/2$.

$\therefore T(n/4) = 2T(n/8) + n/4$.

$\therefore T(n) = 2[2T(n/4) + n/2] + n$.

$\therefore T(n) = 2^2 T(n/2^2) + n + n$.

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

$\therefore T(n) = 2^2 [2T(n/8) + n/4] + 2n$.

$\therefore T(n) = 2^3 T(n/2^3) + 3n$.

⋮

$T(n) = 2^k T(n/2^k) + kn$.

$\therefore T(n) = 2^k T(1) + kn$.

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

$\therefore T(n) = n + n \log n$.

X—X

Chinmay (142203003)

classmate

Date

Page

2. Recurrence Relation (MM).

Q.1) $T(n) = 16T(n/4) + n$.

→ $T(n) = aT(n/b) + \theta(n^k \log^p n)$.

$\therefore a = 16, b = 4, k = 1, p = 1$.

$\therefore a > b^k$ — case 1.

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

$\therefore T(n) = \theta(n^{\log_4 16})$.

$\therefore T(n) = \theta(n^2)$.

Q.2) $T(n) = 2T(n/2) + n\log n$.

→ $\therefore a = 2, b = 2, k = 1, p = 1$.

$\therefore a = b^k$ & $p > -1$. — case (2a).

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

$\therefore T(n) = \theta(n^{\log_2 2} \cdot \log^2 n)$.

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

Q.3) $T(n) = 6T(n/3) + n^2 \log n$.

→ $a = 6, b = 3, k = 2, p = 1$.

$\therefore a < b^k$ & $p \geq 0$. — case 3a.

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

$\therefore T(n) = \theta(n^2 \log n)$.

Q.4) $T(n) = \sqrt{2}\,T(n/2) + \log n$.

→ $a = \sqrt{2}, b = 2, k = 0, p = 1$.

$\therefore a > b^k$ — case 1.

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

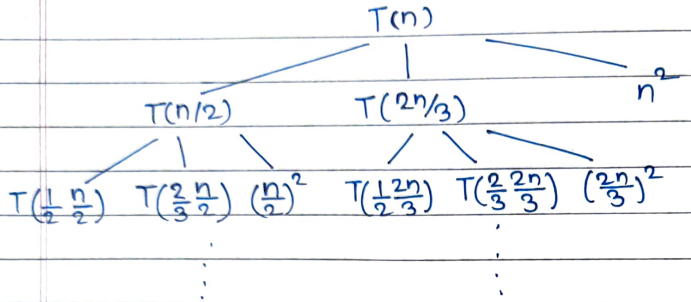
$\therefore T(n) = \theta(n^{\log_2 \sqrt{2}})$

$\therefore T(n) = \theta(\sqrt{n})$.

X—X

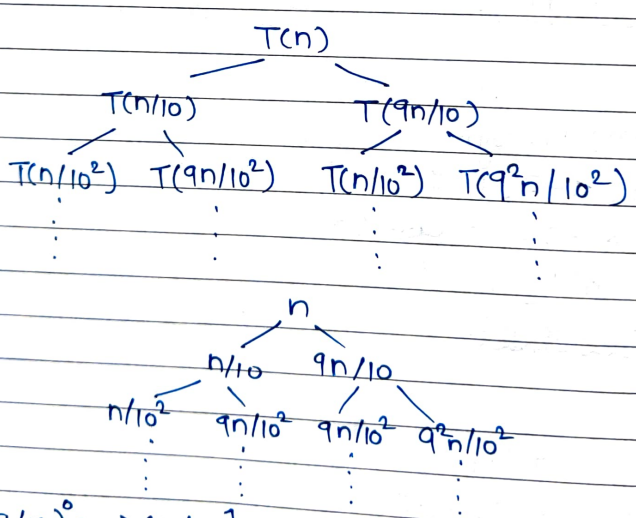
2. Recurrence Relation. (RTM).

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



$$\left(\frac{1}{2}n\right)^2 + \left(\frac{2}{3}n\right)^2 = \left(\frac{1}{4}+\frac{4}{9}\right)n^2 = \left(\frac{25}{36}\right)n^2 \approx O(n^2)$$

Q.2) $T(n) = T(n/10) + T(9n/10) + n$.



$\therefore (9/10)^0 n \to (9/10)^1 n \to (9/10)^2 n \to \ldots (9/10)^k n$.

$\therefore (9/10)^k n = 1$.

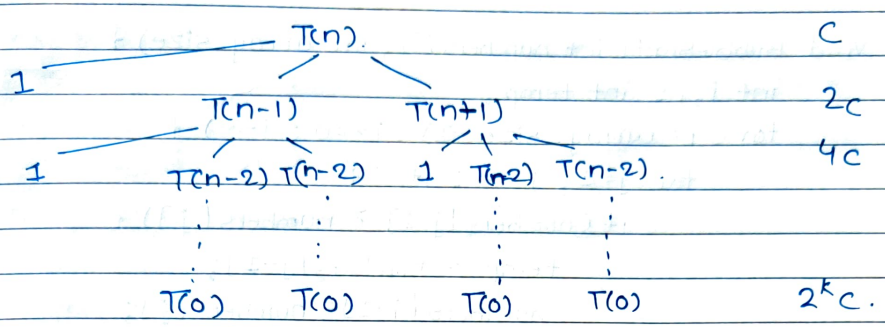
$\therefore (9/10)^k = 1/n$.

$\therefore k = \log_{10/9} n$.

Also, $T(n) = n + n + n + \ldots$ $\log n$ times.

$\therefore T(n) = O(n \log_{10/9} n)$.

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



$\therefore C + 2^1C + 2^2C + 2^3C + 2^4C + \cdots + 2^k C.$

$= 1 + 2 + 4 + \cdots + c\,c.$

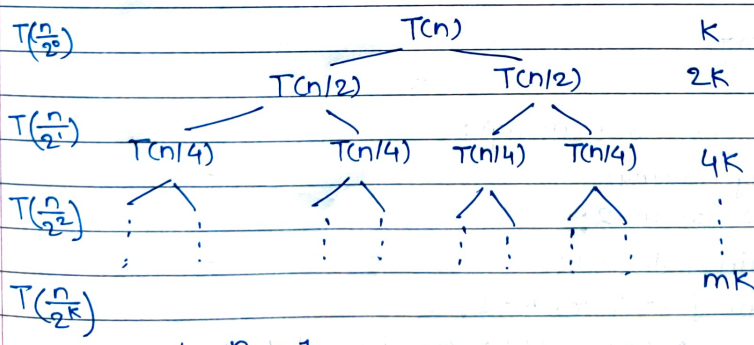
$= 2^{k+1} - 1.$ — GP.

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

$\approx \Theta(2^n).$

X——X

Q.4) $T(n) = 2T(n/2) + K.$



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

$\therefore n = 2^k$

$\therefore \log n = k \cdot \log 2.$

$\therefore k = \log n.$

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

3. Sorting Algorithms!

① Bubble Sort.

→
```
void BubbleSort( int numbers[], int array_size) {
    int i,j; int temp;
    for ( i = (array_size -1) ; i>=0 ; i--) {
        for (j=1; j<=i ; j++) {
            if (numbers[j-1] > numbers[j]) {
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
```

→ Time Complexity – Best Case :- $O(N)$.
Average Case :- $O(N^2)$.
Worst Case :- $O(N^2)$.

② Selection Sort.

→
```
function selection_sort (array):
    m= length(array)
    for i=0 to m-2:
        min_index = i;
        for j=i+1 to m-1:
            if (array [j] < array[min_index]:
                min_index = j;
        swap(array[i], array [min_index]);
```

→ Time Complexity – Best, Average, Worst Case :- $O(N^2)$.

③ Quick Sort.

→ Function quickSort (arr, l, r)
    if (l < r)
        pivotIndex = partition (arr, l, r);
        quickSort (arr, l, pivotIndex - 1)
        quickSort (arr, pivotIndex + 1, r).

function partition (arr, l, r).
    pivot = arr [r]
    i = l - 1.
    For j = l to r - 1.
        if arr[j] < pivot
        i = i + 1
        Swap (arr[i] and arr[j]).
    swap (arr[i+1], arr[r])
    return i+1.

→ Time Complexity – Best Case :- O(NLogn).
Average Case :- O(NLogn).
Worst Case :- $O(N^2)$.

X—X

4. Greedy Algorithms.

1) Knapsack Problem.

Q) →

Object	X1	X2	X3	X4	X5	X6	X7
Weight	2	3	5	7	1	4	1
Profit	10	5	15	7	6	18	3
P/W	5	1.66	3	1	6	4.5	3

→ Step 1 :- Find each object's profit by weight ratio.

Step 2 :- Arrange the profit by weight ratio in descending order & pick the objects accordingly.

X5	X1	X6	X7	X3	X2	X4
6	5	4.5	3	3	1.6	1

Step 3 :- Total capacity of Knapsack = 15.

After selecting X5, capacity = 15-1 = 14.

After selecting X1, capacity = 14-2 = 12.

After selecting X6, capacity = 12-4 = 8.

After selecting X7, capacity = 8-1 = 7.

After selecting X3, capacity = 7-5 = 2.

After selecting X2, capacity = $2-(2/3)*3 = 0$.

We are taking only $2/3^{rd}$ weight of X2.

Step 4 :- Calculate the profit.

$10 + 5 + 15 + 7 + 6 + 18 + 3 + 3.33 = 62.33$.

2) Job Sequencing with deadlines.

a)

Sr. No.	1	2	3	4	5
Jobs	J1	J2	J3	J4	J5
Deadlines	2	2	1	3	4
Profits	20	60	40	100	80

→ Step1 :- Find the maximum value of deadline, from the deadlines given.

∴ dm = 4.

Step2 :- Arrange the jobs in descending order of their profits.

J4	J5	J2	J3	J1
100	80	60	40	20

The max. deadline, dm is 4. Therefore, all the tasks must end before 4.

Choose the job with highest profit, J4. It takes up 3 parts of maximum deadlines.

Therefore, the next job must have the time period 1.

Total profit = 100.

Step3 :- Arrange & select jobs according to their deadlines.

∴ J4 & J3 are choosen.

∴ Maximum profit = 100 + 40 = 140.

X—X

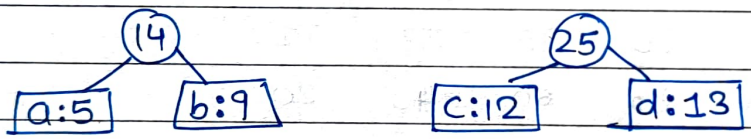
3) Optimal Merge Patterns. (Huffman Coding).

Q)

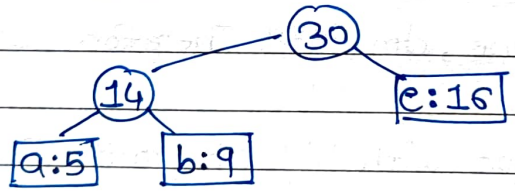
Character.	Frequency.
a	5
b	9
c	12
d	13
e	16
f	45.

→ ① a:5 b:9 c:12 d:13 e:16 f:45.

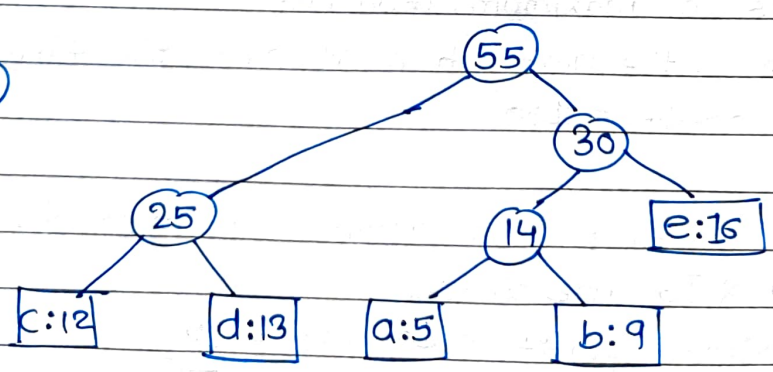
②



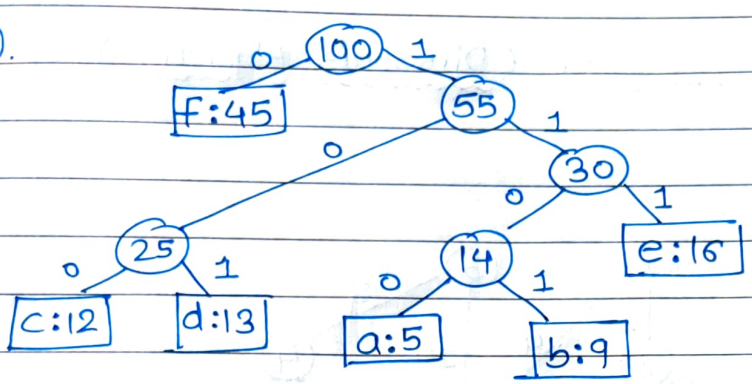
③



④



(5)

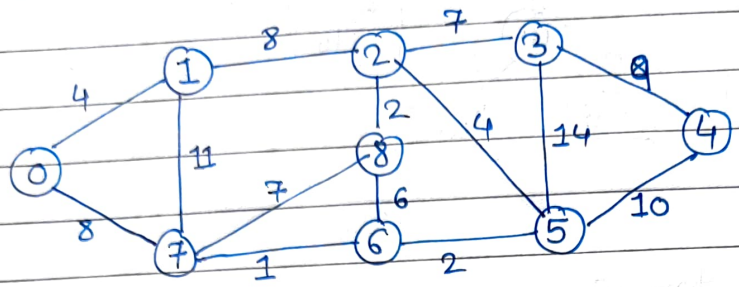


∴ Code words for given characters will be –

Character	Codeword.
f	0
c	100
d	101
a	1100
b	1101
e	111

X—X

4) Single Source Shortest Path. (Dijkstra's Algorithm).



→

Source (Path)	1	2	3	4	5	6	7	8
0	(4)	∞	∞	∞	∞	∞	8	∞
0,1	[4]	12	∞	∞	∞	∞	(8)	∞
0,1,7,	[4]	12	∞	∞	∞	(9)	[8]	15
0,1,7,6	[4]	12	∞	∞	(11)	[9]	[8]	15
0,1,7,6,5	[4]	(12)	25	21	[11]	[9]	[8]	15
0,1,7,6,5,2	[4]	[12]	19	21	[11]	[9]	[8]	(14)
0,1,7,6,5,2,8	[4]	[12]	(19)	21	[11]	[9]	[8]	[14]
0,1,7,6,5,2,8,3,4	[4]	[12]	[19]	21	[11]	[9]	[8]	[14]

∴ Path = 0 → 1 → 7 → 6 → 5 → 2 → 8 → 3 → 4

∴ Cost = 1 : 4
2 : 12
3 : 19
4 : 21
5 : 11
6 : 9
7 : 8
8 : 14.

X — X

3. Sorting Algorithms (2)

① Insertion Sort.

→
```
Void insertionSort(array) {
    for (i=1 ; i<=length(array) ; i++) {
        Key = array[i] ;
        j = i-1;
        while (j>=0 && array[i] > key) {
            array[j+1] = array[j];
            j=j-1; }
        array[j+1] = key;
    }
}
```

→ Time complexity – Best Case :- $O(N)$.
Average Case :- $O(N^2)$.
Worst Case :- $O(N^2)$.

② Heap Sort.

→
```
function HeapSort(array) {
    for (i=n/2 ; i=1 ; i--) {
        Heapify (A,n,i)
    }
    for (i=n ; i>=1 ; i--) {
        Swap(A[1], A[i]
        A.heapSize = A.heapSize - 1.
        Heapify (A,i,0).
    }
}
```

Time Complexity.
↓
Best Case :- O(nlogn).
Avg. Case :- O(nlogn).
Worst Case :- O(nlogn).

→ Heapify (A, n, i)

```
{
    max = i ;
    leftchild = 2i+1 .
    rightchild = 2i+2 .
    if (leftchild <=n) and (A[i] < A[leftchild])
        max = leftchild.
    else.
        max = i.
    if (rightchild <=n) and (A[max] > A[rightchild])
        max = rightchild.

    if (max != i)
        swap(A[i], A[max])
        Heapify(A, n, max) .
}
```

③. MergeSort .

→

```
function mergeSort(array).
    if length of array <= 1 .
        return array.
    middle = length of array / 2.
    leftarray = mergeSort ( first half of array).
    rightarray = merge Sort (Second half of array).
    return merge (leftArray, RightArray);
```

→ Time Complexity :- Best Case :- O(N log N).
Avg. Case :- O(N log N).
Worst Case :- O(N log N).

X ——X

5. Process Management Algorithms.

1) Shortest Job First (SJF).

→ Consider the following set of four processes. Calculate the average waiting time of the process using SJF Algorithm.

⇒

Process	Arrival Time (T0)	CPU Burst Time (ΔT)
P0	0	10
P1	1	6
P2	3	2
P3	5	4

(*) Gantt Chart :-

P0	P2	P3	P1	
0	10	12	16	22.

Process	T0	ΔT	Finish Time (T1)	TAT	WT.
P0	0	10	10	10	0
P1	1	6	22	21	15
P2	3	2	12	9	7
P3	5	4	16	11	7

∴ Average Waiting Time = ~~(10+21+9+11)~~ $\frac{(0+15+7+7)}{4}$ = 7.25 ms.

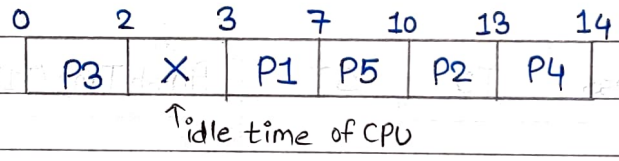
Formulae used :- Turnaround time (TAT) = Finish Time - Arrival Time.
Waiting time (WT) = TAT - CPU Burst Time.

2) First Come First Serve (FCFS).

Q) Consider the set of 5 processes, if the CPU scheduling policy is FCFS, calculate average waiting time & average turn around time.

Process	Arrival Time	Burst Time.
P0	3	4
P1	5	3
P2	0	2
P3	5	1
P4	4	3

→ Gantt Chart :-



→ Turn Around Time = Exit Time - Arrival Time.

Waiting Time = Turn around time - Burst time.

Process	Exit Time	TAT	WT
P1	7	4	0
P2	13	8	5
P3	2	2	0
P4	14	9	8
P5	10	6	3

∴ Avg. TAT = (4+8+2+9+6)/5 = 5.8ms.

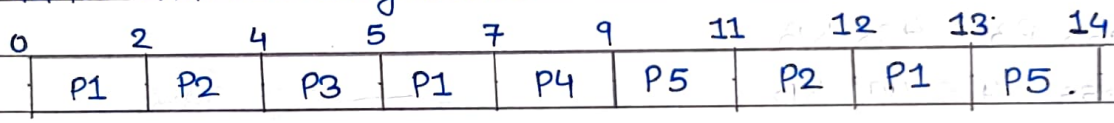
∴ Avg. WT = (0+5+0+8+3)/5 = 3.2ms.

3) Round Robin (RR).

Q) Consider the set of 5 processes whose arrival & Burst time are given. If CPU scheduling policy is Round Robin & time quantum = 2 units, calculate average waiting time & average turnaround time.

Process	Arrival Time	Exit Time.
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

→ Gantt Chart (Ready Queue) :-



→

Process	Exit Time	Turn Around Time	Waiting Time.
P1	13	13	8
P2	12	11	8
P3	5	3	2
P4	9	6	4
P5	14.	10	7

∴ Average TAT = (13+11+3+6+10)/5 = 8.6 units

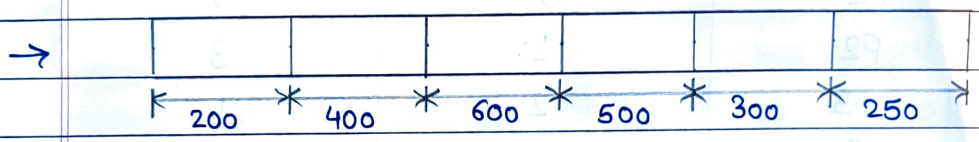
∴ Average WT = (8+8+2+4+7)/5 = 5.8 units.

X — X

6. Memory Management Algorithm.

(*) Consider six memory partitions of size 200KB, 400KB, 600KB, 500KB, 300KB & 250KB. These partitions need to be allocated to four processes of size 357 KB, 210 KB, 468 KB & 491 KB in that order.

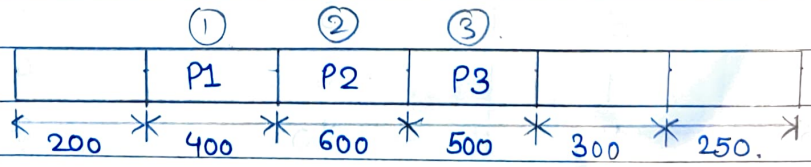
Perform the allocation using -
1) First Fit.
2) Best Fit.
3) Worst Fit.



Main Memory

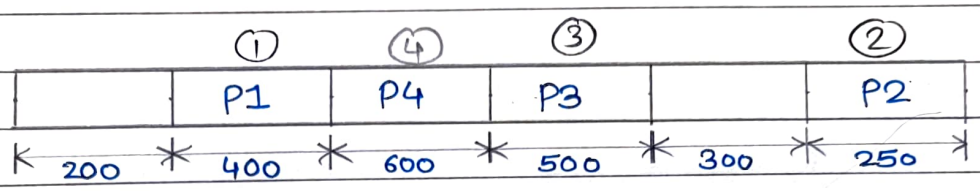
→ Process P1 = 357 KB.
Process P2 = 210 KB.
Process P3 = 468 KB.
Process P4 = 491 KB.

1) First Fit Algorithm.

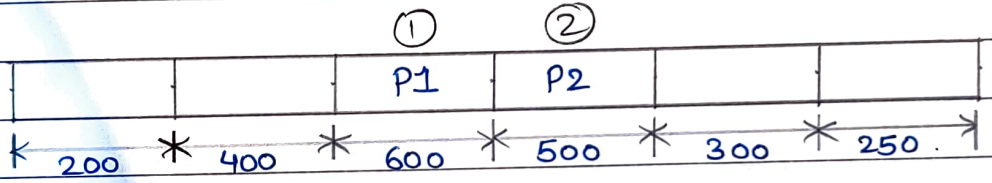


→ Process P4 can not be allocated in the memory, this is because no partition of size greater than or equal to size of process P4 is available.

2) Best Fit Algorithm.



3) Worst Fit Algorithm.



→ Process P3 & P4 cannot be allocated in the memory, because no partition of size greater than or equal to size of P3 & P4 is available.

—

① In First Fit Algorithm, the algorithm starts scanning the partitions serially. When the partition is big enough to store the process is found, it allocates that partition to the process.

② In Best Fit Algorithm, the algorithm first scans all the partitions. It then allocates the partitions of smallest size that can store the process.

③ In Worst Fit Algorithm, the algorithm first scans all the partitions, it then allocates the partition of largest size to the process.

X — X.