Parshvanath Charitable Trust's

A. P. SHAH INSTITUTE OF TECHNOLOGY

(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)
(Religious Jain Minority)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DATA SCIENCE

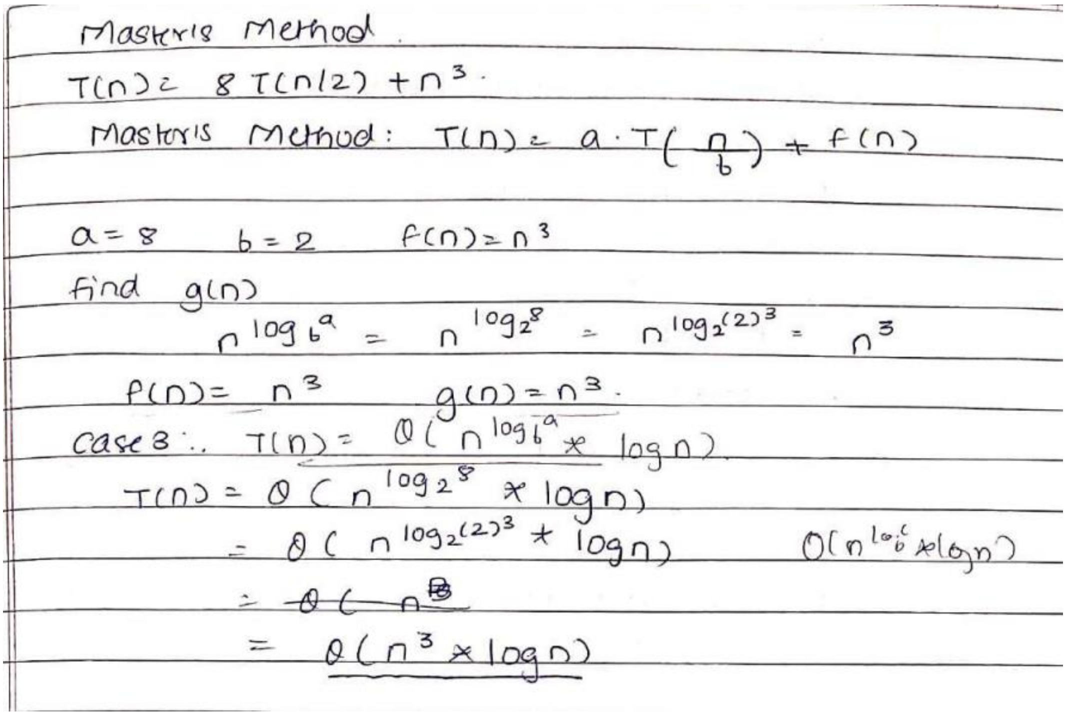
UNIT TEST-I

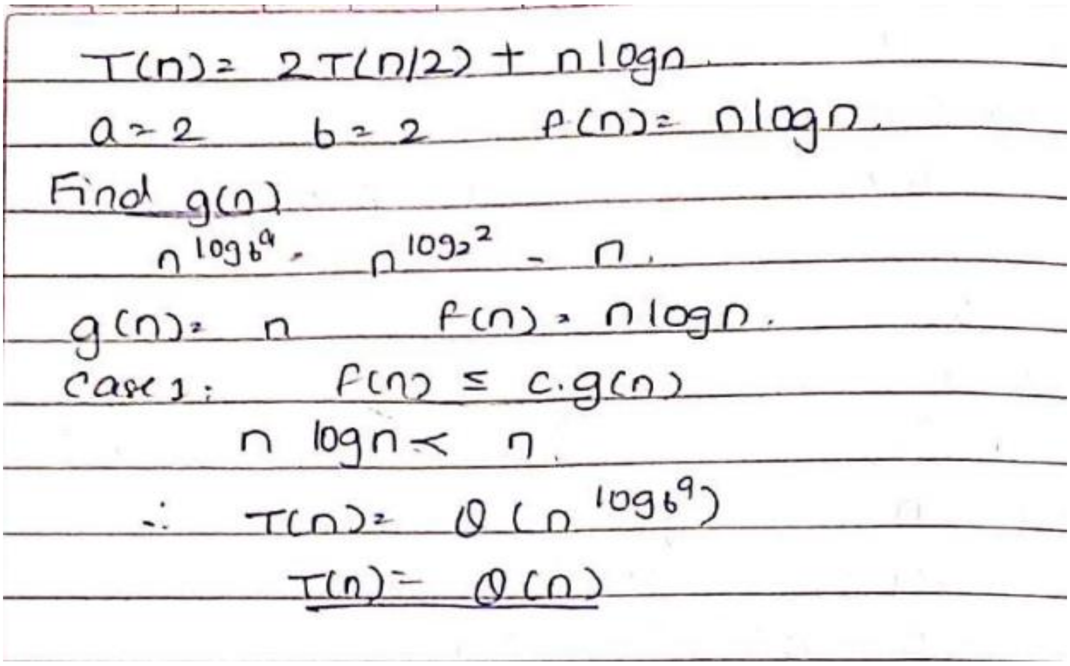
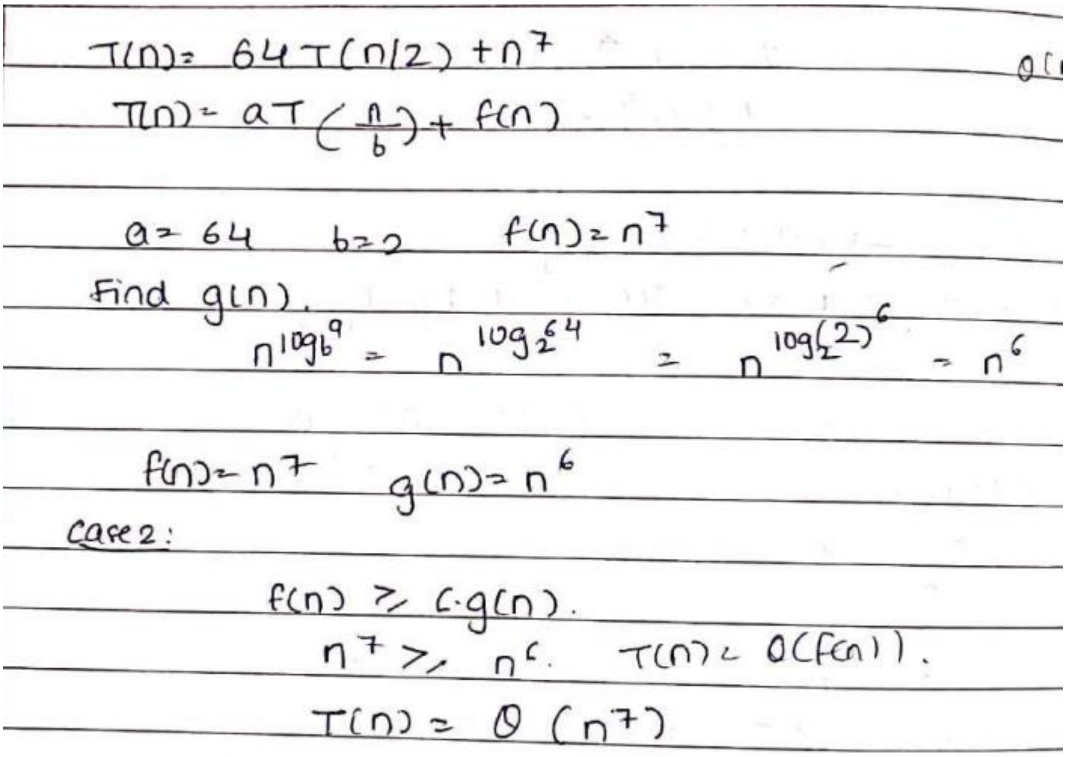
Class: SE **Semester: III** **Subject: CSC402 Analysis of Algorithm**

Date: 28-02-2023 **Time: 04:00 - 05:30** **Max marks: 40**

Note the following instructions

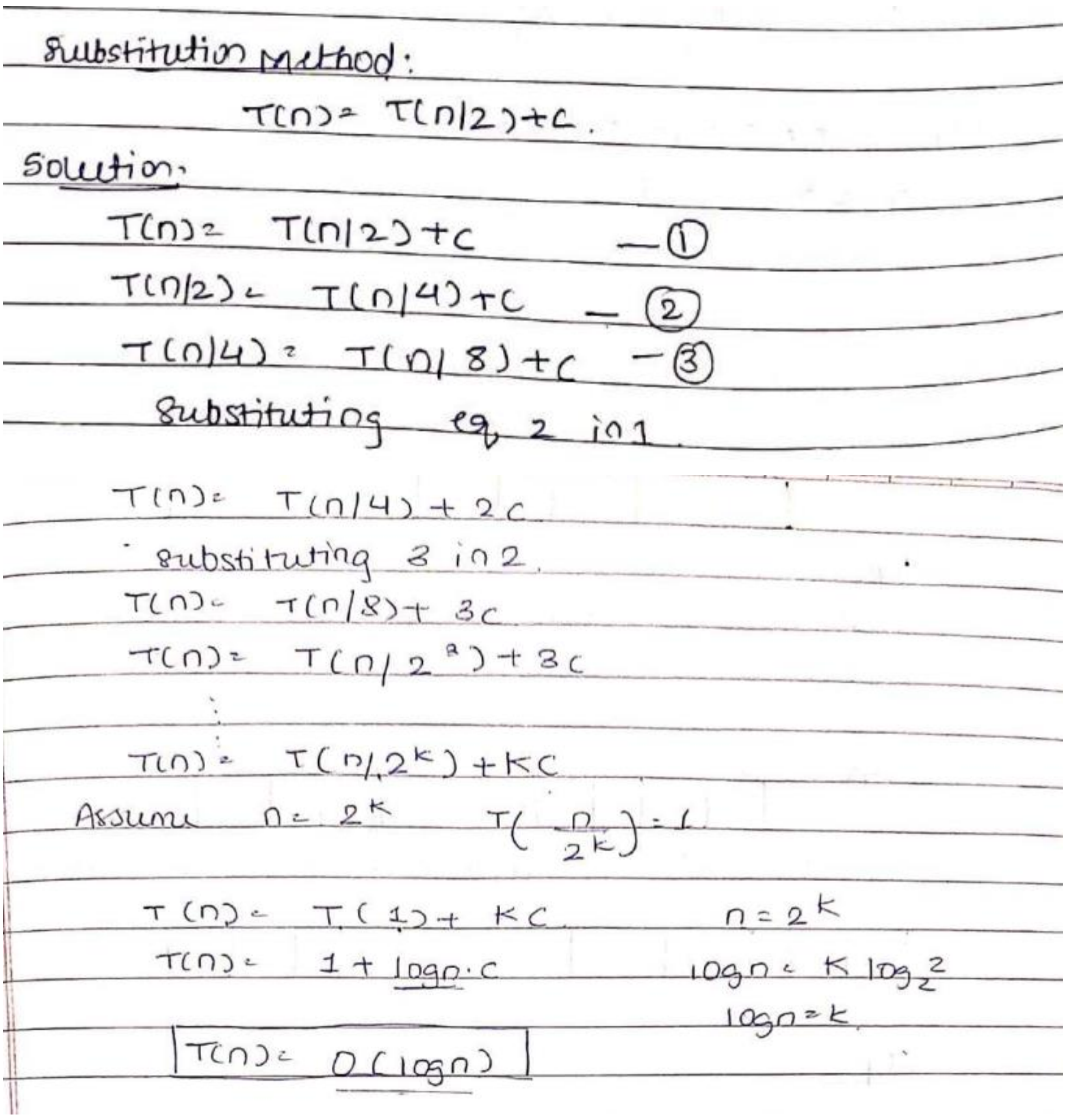
1. **Attempt all questions.**
2. **Draw neat diagrams wherever necessary.**
3. **Write everything in Black ink (no pencil) only.**
4. **Assume data, if missing, with justification.**

Q.N	**Questions**	**MARKS**
Q.1.	Attempt any two.	
i)	Solve following recurrence relation using Masters method. T(n)=8T(n/2) + n^3  Masters Method. $T(n) = 8\,T(n/2) + n^3$. Masters Method: $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ $a = 8 \quad b = 2 \quad f(n) = n^3$ find $g(n)$ $n^{\log_b a} = n^{\log_2 8} = n^{\log_2 (2)^3} = n^3$ $f(n) = n^3 \quad g(n) = n^3$. case 3 ∴ $T(n) = \theta(n^{\log_b a} * \log n)$. $T(n) = \theta(n^{\log_2 8} * \log n)$ $= \theta(n^{\log_2 (2)^3} * \log n) \quad \theta(n^{\log_b^c} * \log n)$ $=$ ~~$\theta(n^3$~~ $= \theta(n^3 * \log n)$	[5]

ii)	Solve following recurrence relation using Masters method. T(n)=2T(n/2) + nlogn  $T(n) = 2T(n/2) + n\log n$. $a=2$ $b=2$ $f(n) = n\log n$. Find $g(n)$ $n^{\log_b a} = n^{\log_2 2} = n$. $g(n) = n$ $f(n) = n\log n$. Case 1: $f(n) \leq c.g(n)$ $n \log n < n$. $\therefore$ $T(n) = \Theta(n^{\log_b a})$ $T(n) = \Theta(n)$	[5]
iii)	Solve following recurrence relation using Masters method. T(n)=64T(n/2) + n^7  $T(n) = 64T(n/2) + n^7$ $\Theta($ $T(n) = aT(\frac{n}{b}) + f(n)$ $a=64$ $b=2$ $f(n)=n^7$ Find $g(n)$. $n^{\log_b a} = n^{\log_2 64} = n^{\log_2 (2)^6} = n^6$ $f(n)=n^7$ $g(n)=n^6$ Case 2: $f(n) \geq c.g(n)$. $n^7 \geq n^6$. $T(n) = \Theta(f(n))$. $T(n) = \Theta(n^7)$	[5]

iv) Solve following recurrence relation using substitution method. [5]

T(n)=T(n/2) + C



Substitution Method:

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

Solution.

$T(n) = T(n/2) + c$ — ①

$T(n/2) = T(n/4) + c$ — ②

$T(n/4) = T(n/8) + c$ — ③

Substituting eq 2 in 1.

$T(n) = T(n/4) + 2c$

substituting 3 in 2.

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

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

$\vdots$

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

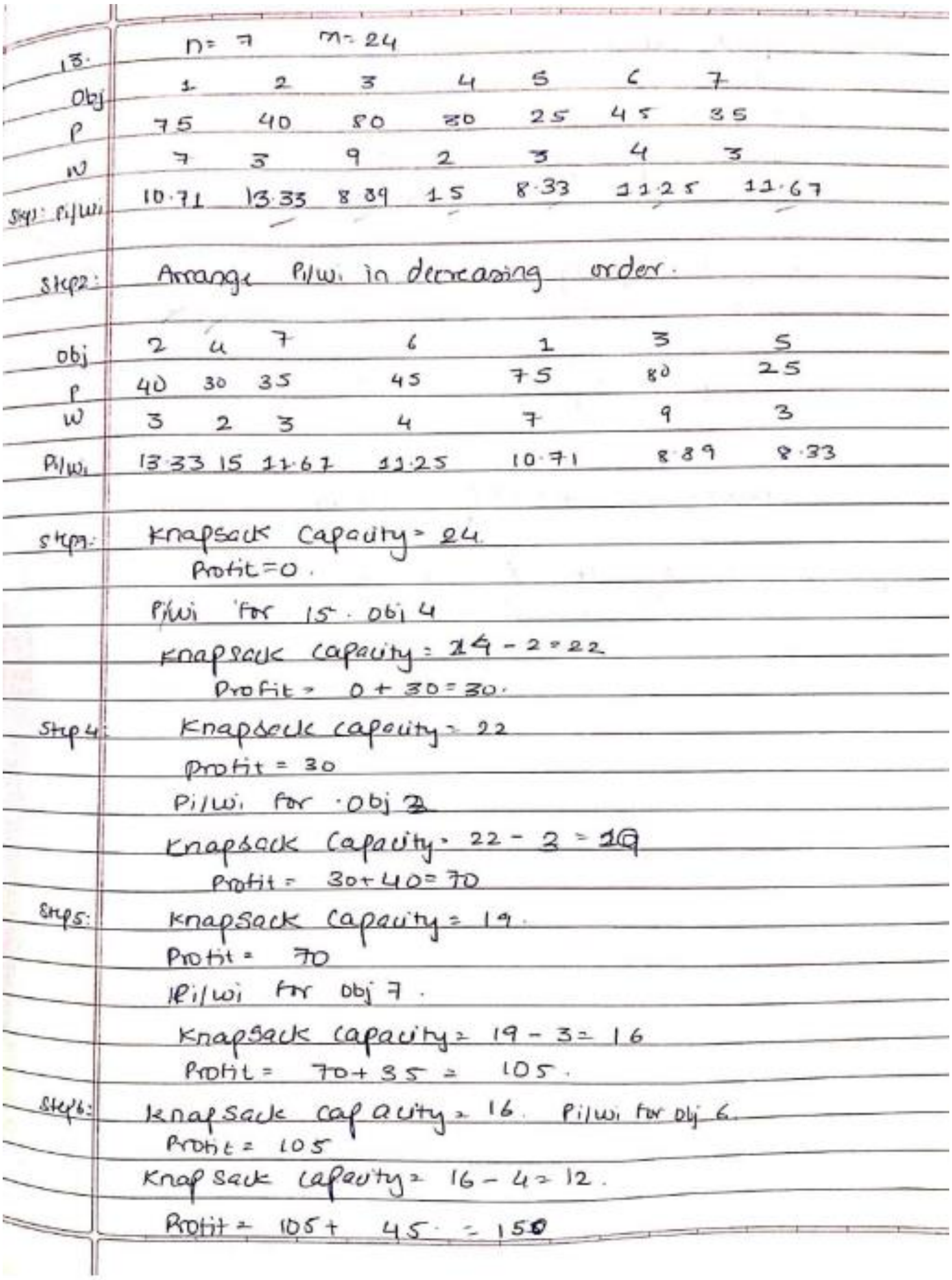
Assume $n = 2^k$ $\quad T\left(\frac{n}{2^k}\right) = 1$

$T(n) = T(1) + kc \qquad n = 2^k$

$T(n) = 1 + \underline{\log n} \cdot c \qquad \log n = k \log_2 2$

$\log n = k$

$$\boxed{T(n) = \underline{O(\log n)}}$$

Q.2.	**Attempt any two**	
i)	Write an algorithm for fractional knapsack problem and analyse its complexity. Solve following using problem using fractional knapsack algorithmic approach. N=7 M=24 (P1,P2,…,P7)=(75,40,80,30,25,45,35) (W1,W2,….,W3)=(7,3,9,2,3,4,3) 	[10]

13. n= 7 m= 24

Obj	1	2	3	4	5	6	7
P	75	40	80	30	25	45	35
W	7	3	9	2	3	4	3
Step1: Pi/wi	10.71	13.33	8.89	15	8.33	11.25	11.67

Step2: Arrange Pi/wi in decreasing order.

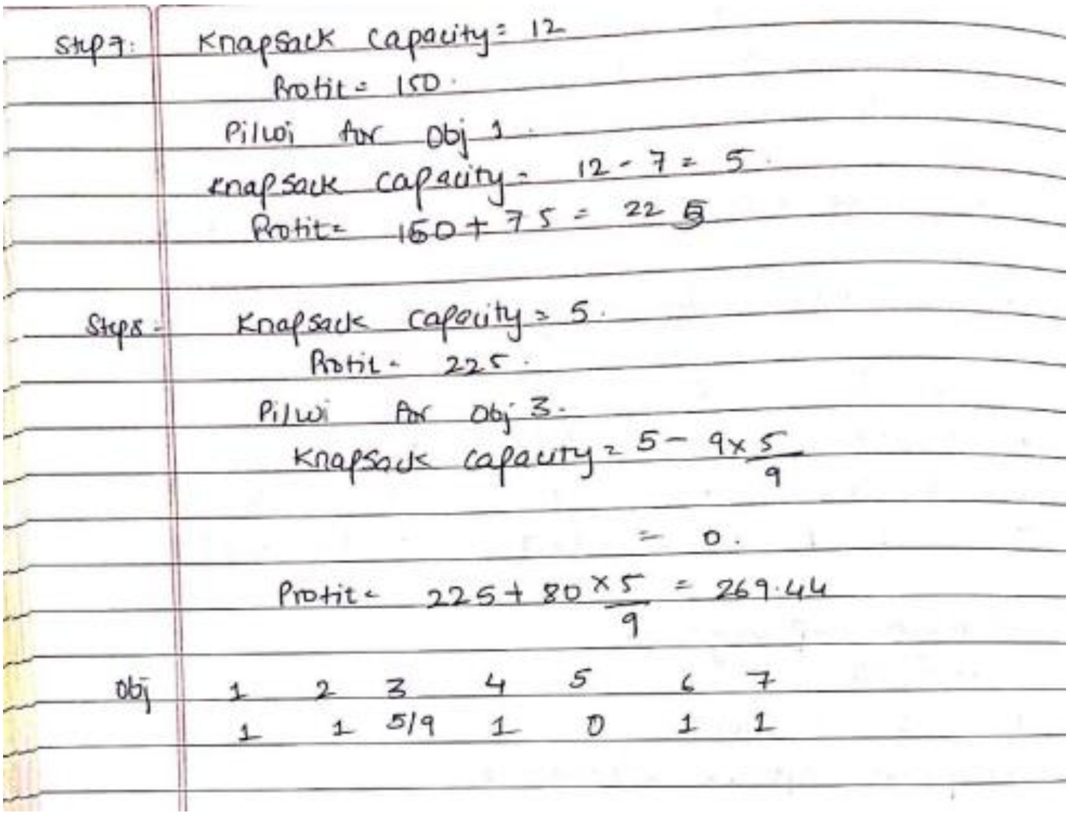
obj	2	4	7	6	1	3	5
P	40	30	35	45	75	80	25
W	3	2	3	4	7	9	3
Pi/wi	13.33	15	11.67	11.25	10.71	8.89	8.33

Step3: Knapsack Capacity = 24.
Profit = 0.
Pi/wi for 15. obj 4
Knapsack capacity = 24 - 2 = 22
Profit = 0 + 30 = 30.

Step4: Knapsack capacity = 22
profit = 30
Pi/wi for ·obj 2
Knapsack Capacity = 22 - 3 = 19
Profit = 30+40 = 70

Step5: Knapsack capacity = 19.
Profit = 70
Pi/wi for obj 7.
Knapsack capacity = 19 - 3 = 16
Profit = 70 + 35 = 105.

Step6: Knapsack capacity = 16. Pi/wi for obj 6.
Profit = 105
Knapsack capacity = 16 - 4 = 12.
Profit = 105 + 45 = 150



Step 7: Knapsack capacity = 12

Profit = 150.

Pi/wi for obj 1.

knapsack capacity = 12 - 7 = 5.

Profit = 150 + 75 = 225

Step 8: Knapsack capacity = 5.

Profit = 225.

Pi/wi for obj 3.

Knapsack capacity = $5 - 9 \times \frac{5}{9}$

$= 0.$

Profit = $225 + 80 \times \frac{5}{9} = 269.44$

obj	1	2	3	4	5	6	7
	1	1	5/9	1	0	1	1

Fractional knapsack

1) Given the weights & values of N items, in the form of {value, weight} pull these items in a knapsack of capacity W to get the maximum total value in knapsack.

2) **Algorithm**:

1) Step 1: Calculate the ratio (value/weight) for each item.

2) Step 2: Sort all the items in decreasing order of ratio.

3) Step 3: Initialize res = 0, curr _ cap = given_ cap

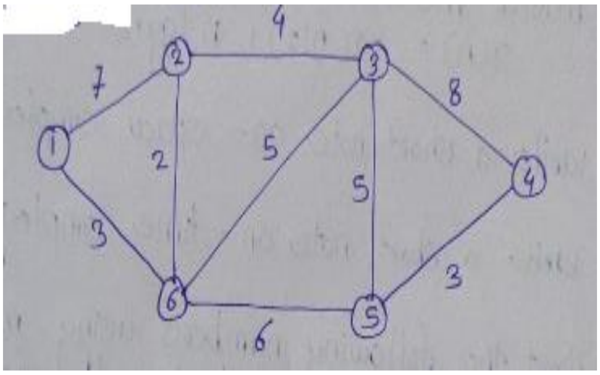
4) Step 4: Check if weight of current item is less than or equal to remaining capacity if yes then add the value of that item into the result.

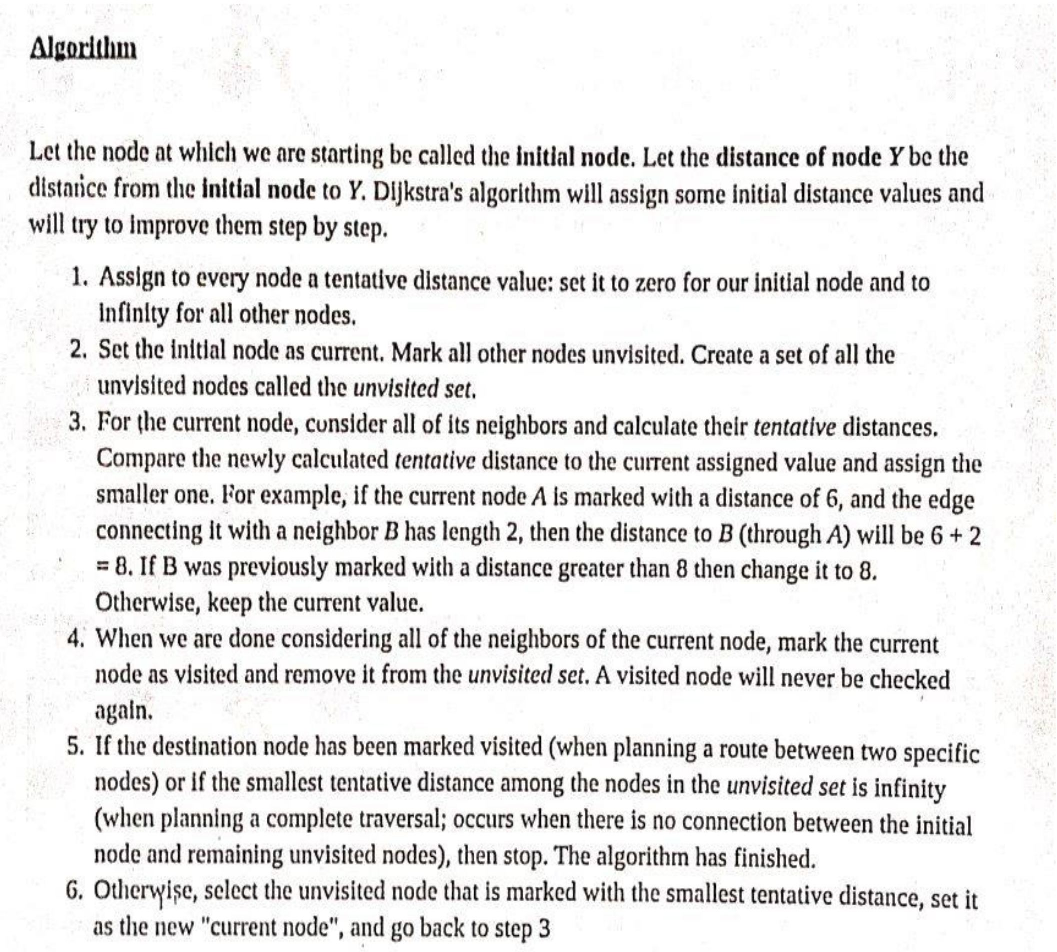
Else add the current item as much as we can and break out of loop.

5) Step 5: Return res.

Time Complexity of Fractional knapsack is **O(nlogn)**.

ii)	Write Dijkstra’s algorithm and comment on its time complexity . Find shortest path from vertex 1 to 4 for the following graph.	[10]



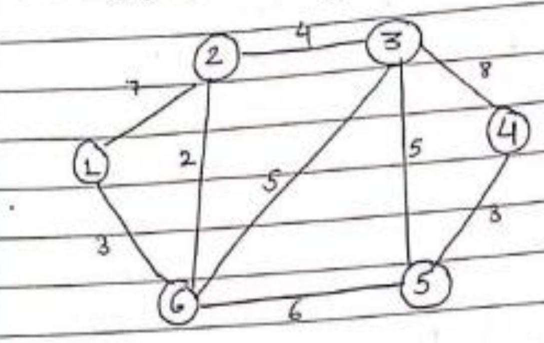


Algorithm

Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3

15 Dijkstra's Algorithm.



source vertex = 1.

step1: cost Matrix.

~~1~~		1	2	3	4	5	6
~~2~~	1	∞	7	∞	∞	∞	3
~~3~~	2	7	∞	4	∞	∞	2
~~4~~	3	∞	4	∞	8	5	5
~~5~~	4	∞	∞	8	∞	3	∞
~~6~~	5	∞	∞	5	3	∞	6
	6	3	2	5	∞	6	∞

Step2: vertex select 1.

a) mark visited[~~1~~] as 1

b) Readjust the distance of vertices which are not marked as visited.

$d_1 = 0$ Select the minimum distance

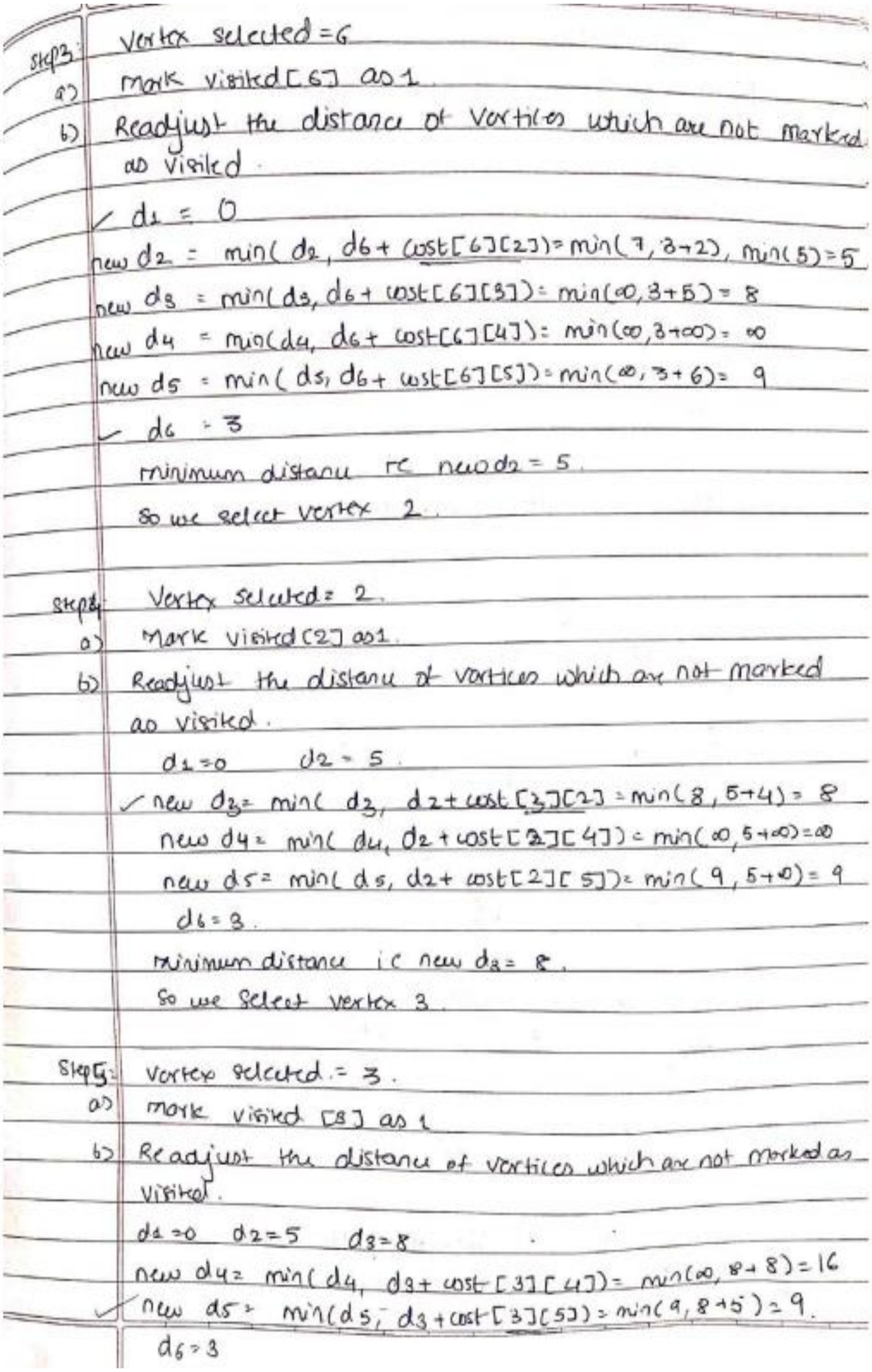
$d_2 = 7$ i.e $d_6 = 3$

$d_3 = \infty$ So vertex 6 is selected

$d_4 = \infty$

$d_5 = \infty$

$d_6 = 3$ ✓



step3: Vertex selected = 6

a) mark visited[6] as 1.

b) Readjust the distance of vertices which are not marked as visited.

✓ $d_1 = 0$

new $d_2 = \min(d_2, d_6 + cost[6][2]) = \min(7, 3+2), \min(5) = 5$

new $d_3 = \min(d_3, d_6 + cost[6][3]) = \min(\infty, 3+5) = 8$

new $d_4 = \min(d_4, d_6 + cost[6][4]) = \min(\infty, 3+\infty) = \infty$

new $d_5 = \min(d_5, d_6 + cost[6][5]) = \min(\infty, 3+6) = 9$

✓ $d_6 = 3$

minimum distance ie new $d_2 = 5$.

So we select vertex 2.

step4: Vertex selected = 2.

a) mark visited[2] as 1.

b) Readjust the distance of vertices which are not marked as visited.

$d_1 = 0$ $d_2 = 5$.

✓ new $d_3 = \min(d_3, d_2 + cost[3][2] = \min(8, 5+4) = 8$

new $d_4 = \min(d_4, d_2 + cost[2][4]) = \min(\infty, 5+\infty) = \infty$

new $d_5 = \min(d_5, d_2 + cost[2][5]) = \min(9, 5+\infty) = 9$

$d_6 = 3$.

minimum distance ie new $d_3 = 8$.

So we select vertex 3.

step5: vertex selected = 3.

a) mark visited [3] as 1

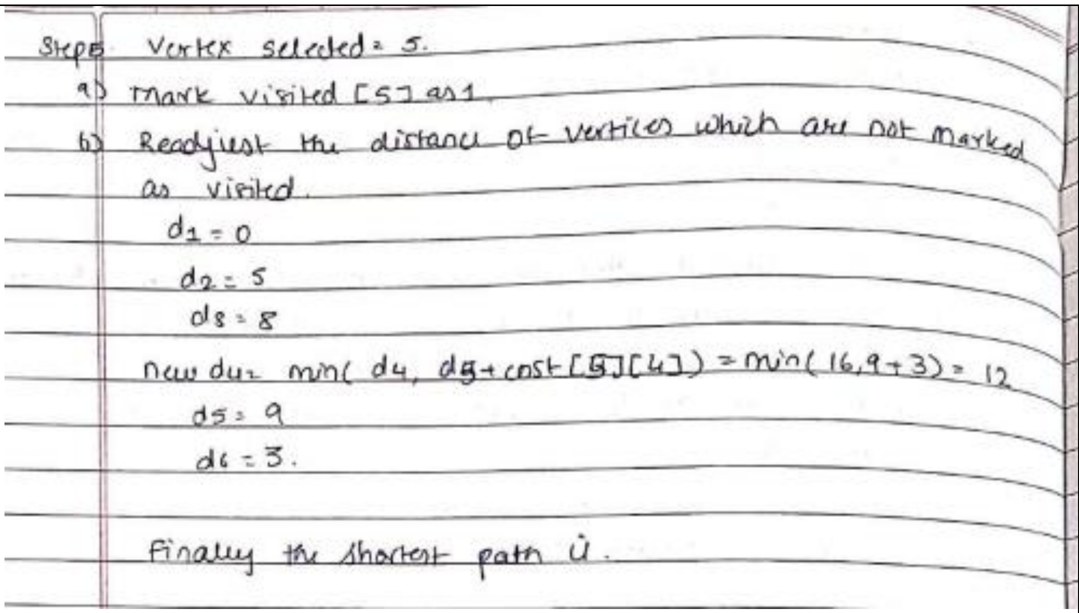
b) Readjust the distance of vertices which are not marked as visited.

$d_1 = 0$ $d_2 = 5$ $d_3 = 8$

new $d_4 = \min(d_4, d_3 + cost[3][4]) = \min(\infty, 8+8) = 16$

✓ new $d_5 = \min(d_5, d_3 + cost[3][5]) = \min(9, 8+5) = 9$.

$d_6 = 3$



Step5 Vertex selected = 5.

a) Mark visited [5] as 1.

b) Readjust the distance of vertices which are not marked as visited.

$d_1 = 0$

$d_2 = 5$

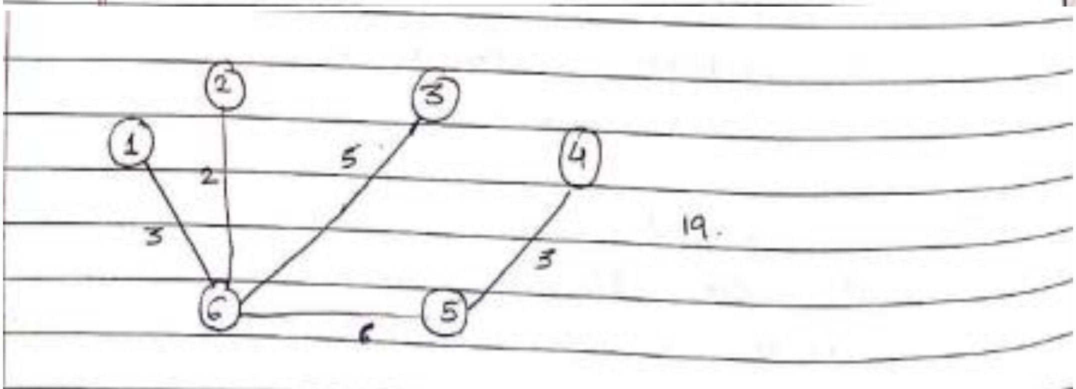
$d_3 = 8$

new d_4 = min(d_4, d_5 + cost[5][4]) = min(16, 9+3) = 12

$d_5 = 9$

$d_6 = 3$.

Finally the shortest path is.



cost of shortest path = 17.

Time complexity → O(V²) V = Vertex

Time complexity of Dijkstra's algorithm is $O(V^2)$ where **V** is the number of verices in the graph.

It can be explained as below:

1. First thing we need to do is find the unvisited vertex with the smallest path. For that we require $O(V)$ time as we need check all the vertices.

2. Now for each vertex selected as above, we need to relax its neighbours which means to update each neighbours path to the smaller value between its current path or to the newly found. The time required to relax one neighbour comes out to be of order of **O(1)** (constant time).

3. For each vertex we need to relax all of its neighbours, and a vertex can have at most V-1 neighbours, so the time required to update all neighbours of a vertex comes out to be [O(V) * O(1)] = O(V)

So now following the above conditions, we get:

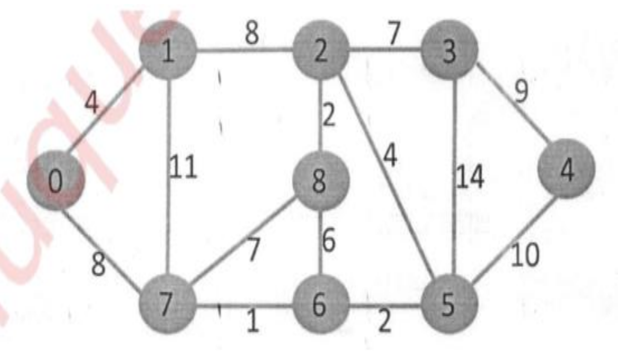
Time for visiting all vertices =O(V)

Time required for processing one vertex=O(V)

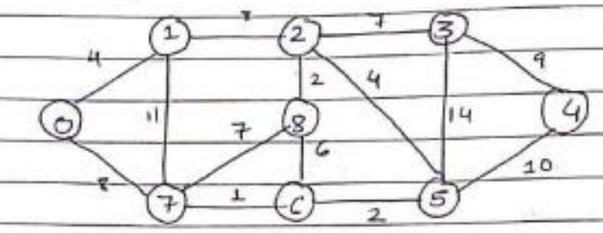
Time required for visiting and processing all the vertices = O(V)*O(V) =O(V^2)

So the time complexity of dijkstra's algorithm using adjacency matrix representation comes out to be $O(V^2)$

iii) Write Prim's algorithm and comment on its time complexity. Find minimum cost spanning tree of the following graph using Prim's algorithm. [10]



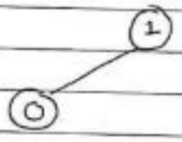
16. Prim's algorithm



Step1. Select ⓪ note as starting point.

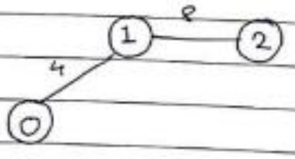
Step2. Check the outgoing edges of 0 and select the one with less cost.

0→1 = 4 ✓
0→7 = 8



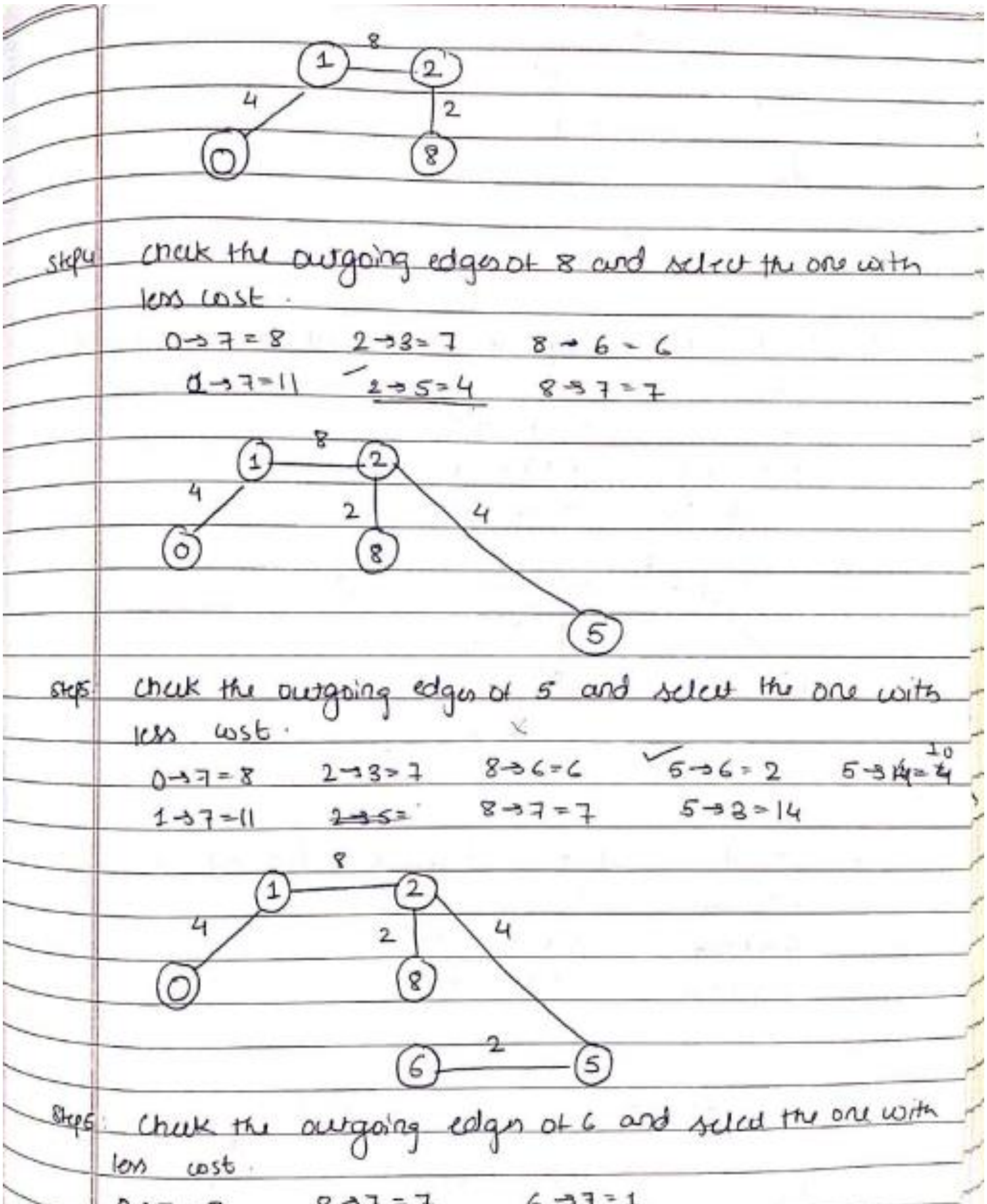
Step2: Check the outgoing edges of 1 and select the one with less cost.

0→7=8   1→2 = 8 ✓
1→7=11



Step3: Check the outgoing edges of 2 and select the one with less cost.

0→7=8   2→3=7   2→5=4
1→7=11   2→8 = 2 ✓



8
1
2
4
2
0
8

step4 check the outgoing edges of 8 and select the one with less cost.

0 → 7 = 8   2 → 3 = 7   8 → 6 = 6

~~0~~1 → 7 = 11   2 → 5 = 4   8 → 7 = 7

8
1
2
4
2
4
0
8
5

step5 check the outgoing edges of 5 and select the one with less cost.

0 → 7 = 8   2 → 3 = 7   8 → 6 = 6   5 → 6 = 2   5 → 4 = ~~4~~ 10

1 → 7 = 11   ~~2 → 5 =~~   8 → 7 = 7   5 → 3 = 14

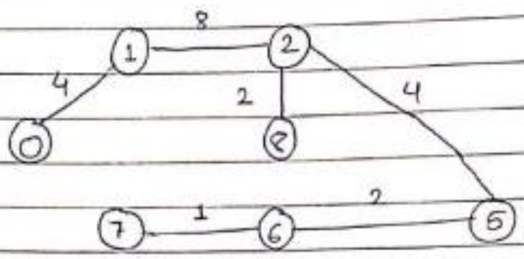
8
1
2
4
2
4
0
8
2
6
5

step6 check the outgoing edges of 6 and select the one with less cost.

0 → 7 = 8   8 → 7 = 7   6 → 7 = 1

1 → 7 = 11   5 → 3 = 14

2 → 3 = 7   5 → 4 = 10

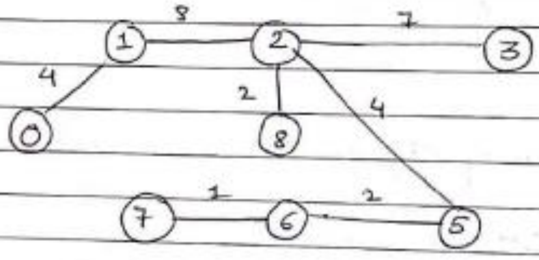


Step 7: Select outgoing edges of 7 and select the one with minimum cost.

✗ 0→7 = 8 8→7 = 7 ✗

✗ 1→7 = 11 5→3 = 14

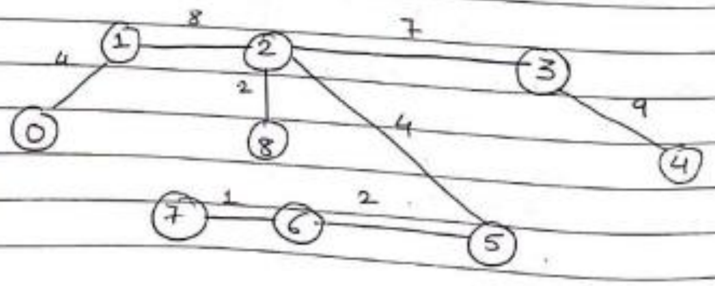
2→3 = 7 ✓ 5→4 = 10



Step 8: Select the outgoing edges of 3 and select the one with less cost.

✗ 5→3 = 14 3→4 = 9 ✓

5→4 = 10. 3→5 = 14 ✗



Minimum cost of spanning tree = 37.

Time complexity: O((V+E) log V) E: Edges V: Vertex.

Prim's algorithm is a greedy algorithm that finds the minimum spanning tree of a connected, undirected graph. Here's how it works:

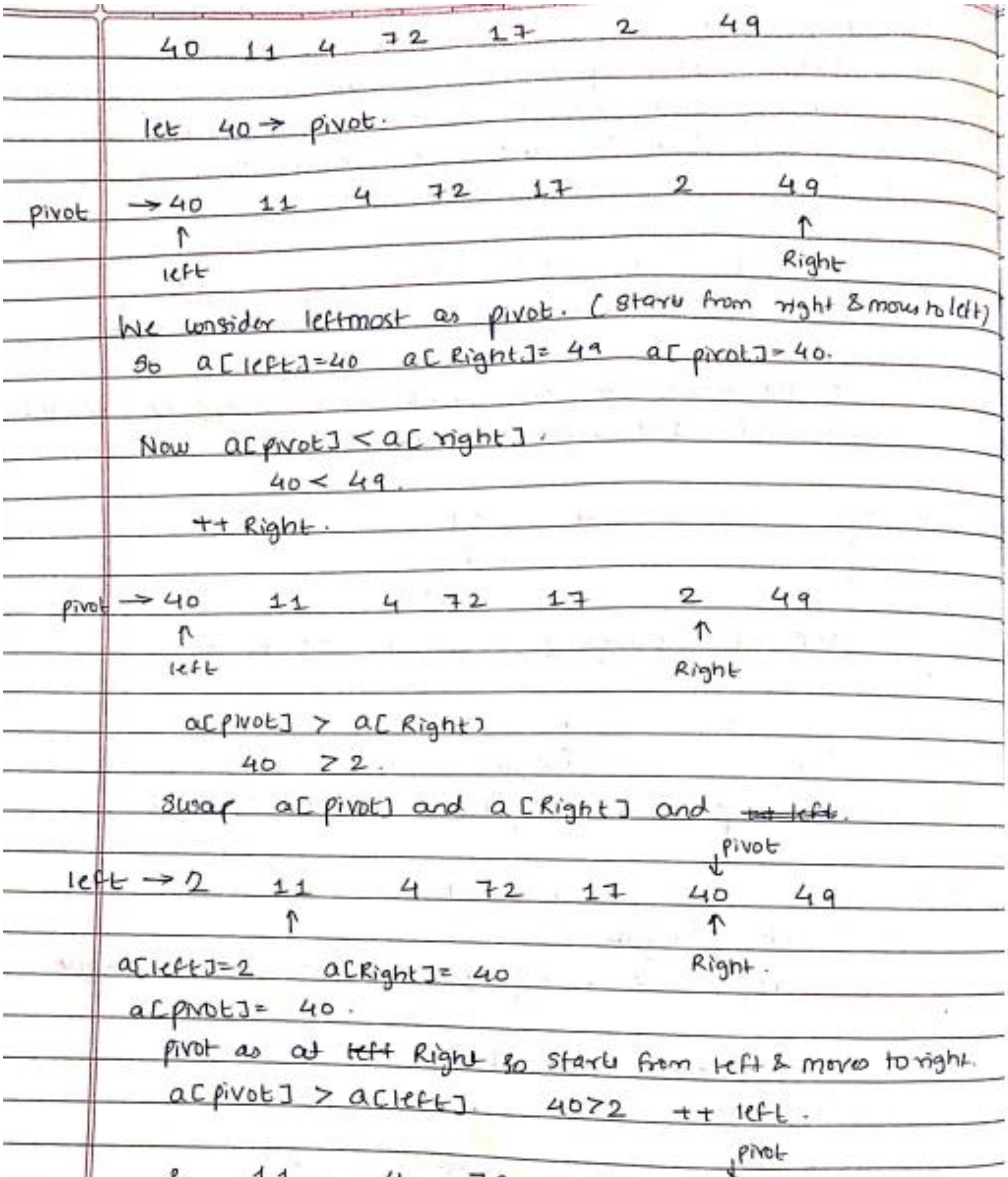
1. Start with a node, and mark it as visited.
2. For all of its neighboring nodes, add the edge with the smallest weight to a priority queue.
3. Pick the edge with the smallest weight from the priority queue. If its destination node has not been visited, add it to the minimum spanning tree, mark it as visited, and add all of its neighboring edges to the priority queue.
4. Repeat step 3 until all nodes have been visited.

Time complexity of Prim’s algorithm is **O((V+E)logV)** where **V** is the number of vertices and E is the Edges in the graph.

Q.3.	Attempt any one.	
i)	Write an algorithm for quick sort and sort the following members elements using quick sort. 40,11,4,72,17,2,49 Analyze best case time complexity of quick sort.	[10]

```
QUICKSORT (array A, start, end)
{
    if (start < end)
    {
        p = partition(A, start, end)
        QUICKSORT (A, start, p - 1)
        QUICKSORT (A, p + 1, end)
    }
}
PARTITION (array A, start, end)
{
    pivot ? A[end]
    i ? start-1
    for j ? start to end -1 {
        do if (A[j] < pivot) {
            then i ? i + 1
            swap A[i] with A[j]
    }}
    swap A[i+1] with A[end]
    return i+1
}
```

Time Complexity of Quick Sort is **O(nlogn)**.



40 11 4 72 17 2 49

let 40 → pivot.

pivot → 40 11 4 72 17 2 49
↑ left ↑ Right

We consider leftmost as pivot. (starts from right & moves to left)
So a[left]=40 a[Right]= 49 a[pivot]= 40.

Now a[pivot] < a[right].
40 < 49.
++ Right.

pivot → 40 11 4 72 17 2 49
↑ left ↑ Right

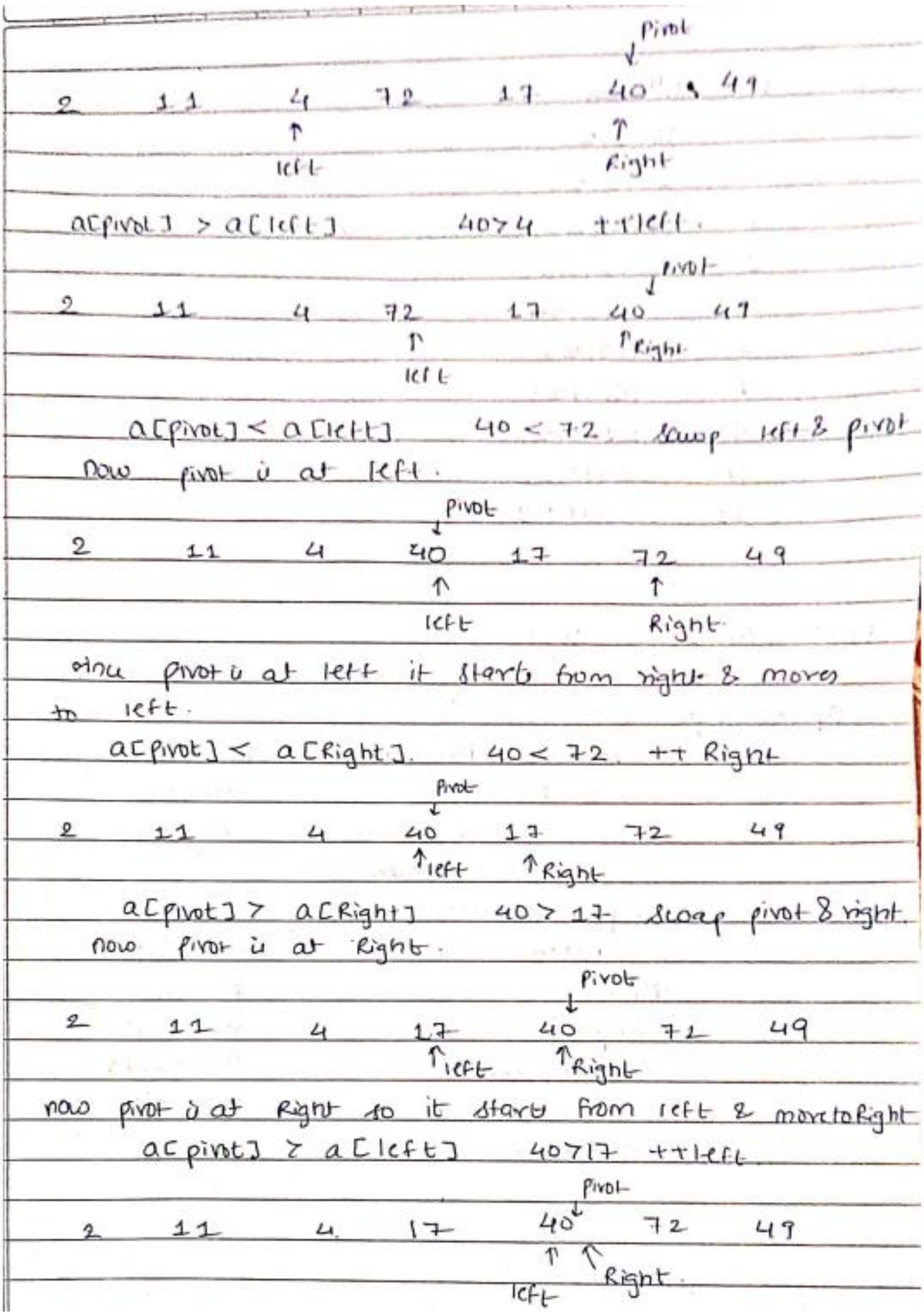
a[pivot] > a[Right]
40 > 2.
Swap a[pivot] and a[Right] and ~~tot left~~.

pivot ↓
left → 2 11 4 72 17 40 49
↑ ↑ Right.

a[left]=2 a[Right]= 40
a[pivot]= 40.
pivot as at ~~left~~ Right so starts from left & moves to right.
a[pivot] > a[left]. 40>2 ++ left.

pivot ↓
2 11 4 72 17 40 49
↑ left. ↑ Right

a[pivot] > a[left] 40> 11 ++left



```
                                   Pivot
                                     ↓
2      11      4      72      17     40      49
               ↑                     ↑
              left                 Right
```

a[pivot] > a[left]   40>4   ++left.

```
                                   pivot
                                     ↓
2      11      4      72      17     40      49
                      ↑              ↑Right
                     left
```

a[pivot] < a[left]   40 < 72   swap left & pivot
now pivot is at left.

```
                     pivot
                       ↓
2      11      4      40      17     72      49
                      ↑              ↑
                     left          Right
```

since pivot is at left it starts from right & moves to left.

a[pivot] < a[Right].   40< 72   ++ Right

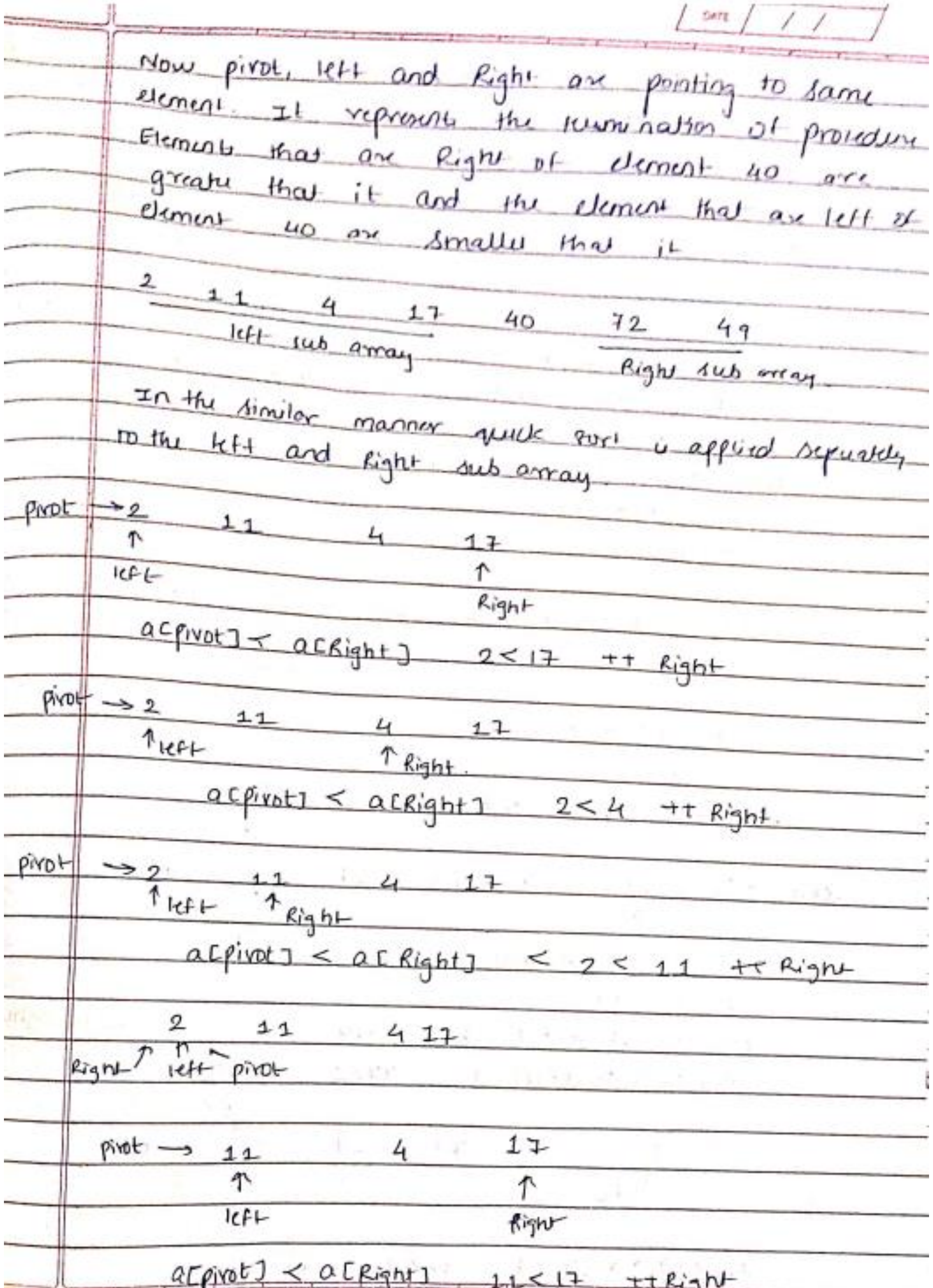
```
                     Pivot
                       ↓
2      11      4      40      17     72      49
                      ↑left   ↑Right
```

a[pivot] > a[Right]   40 > 17   swap pivot & right.
now pivot is at Right.

```
                             Pivot
                               ↓
2      11      4      17      40     72      49
                      ↑left   ↑Right
```

now pivot is at Right so it starts from left & move to Right
a[pivot] > a[left]   40>17   ++left.

```
                             Pivot
                               ↓
2      11      4      17      40     72      49
                              ↑ ↑Right
                            left
```

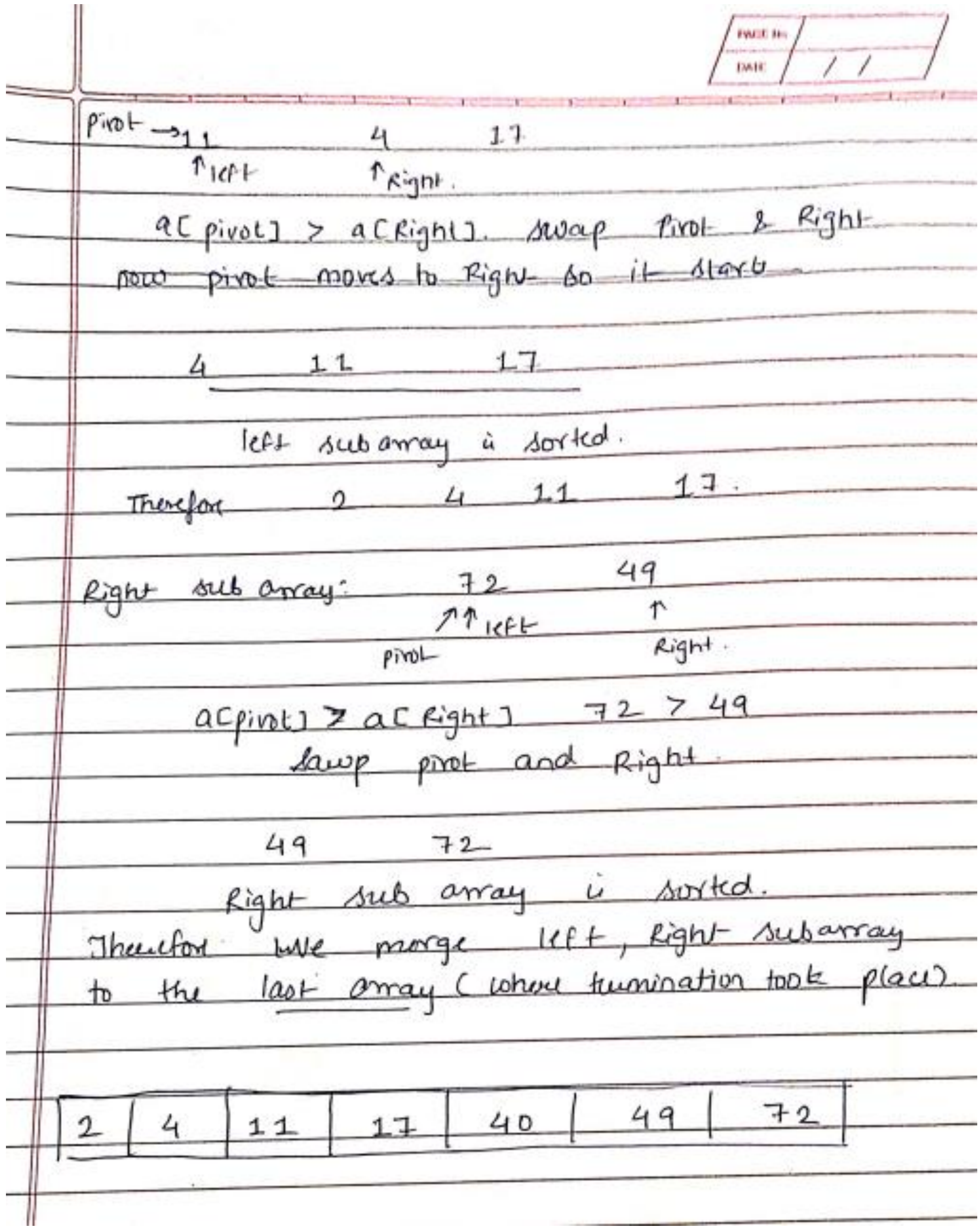


Now pivot, left and Right are pointing to same element. It represents the termination of procedure Elements that are Right of element 40 are greater that it and the element that are left of element 40 are smaller that it

```
2   11   4   17      40      72   49
  left sub array             Right sub array
```

In the similar manner quick sort is applied separately to the left and Right sub array.

```
pivot → 2     11     4     17
        ↑                  ↑
        left               Right

   a[pivot] < a[Right]     2<17   ++ Right

pivot → 2     11     4     17
        ↑ left       ↑ Right

        a[pivot] < a[Right]     2<4   ++ Right

pivot → 2     11     4     17
        ↑ left  ↑ Right

        a[pivot] < a[Right]   <  2 < 11   ++ Right

        2     11     4 17
Right ↑ ↑ left ↖ pivot

pivot → 11     4     17
        ↑            ↑
        left         Right

   a[pivot] < a[Right]    11<17   ++Right
```



PAGE No.
DATE / /

pivot → 11   4   17
↑left   ↑Right.

a[pivot] > a[Right]. swap Pivot & Right
now pivot moves to Right so it start.

4   11   17

left sub array is sorted.

Therefor   2   4   11   17.

Right sub array: 72   49
↑↑left   ↑
pivot   Right.

a[pivot] > a[Right]   72 > 49
Sawp pivot and Right.

49   72

Right sub array is sorted.
Therefor we merge left, Right subarray to the last array (where termination took place).

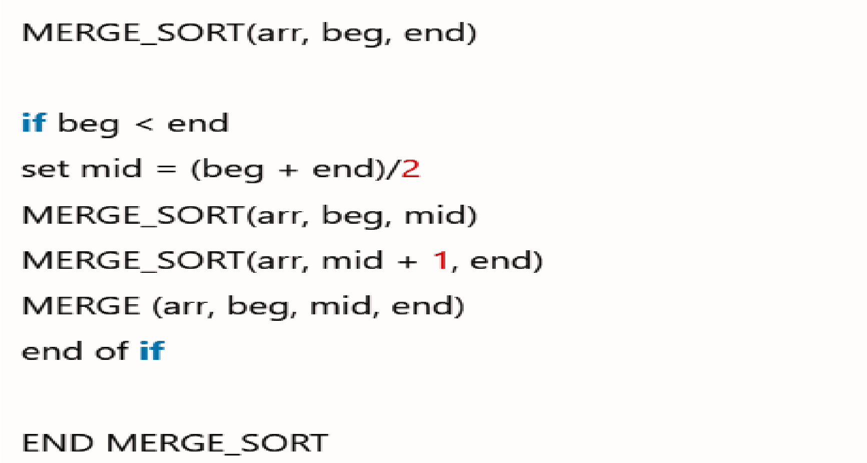
2	4	11	17	40	49	72

ii) Sort the following numbers using merge sort. Analyze the time complexity of merge sort. [10]

70,20,30,40,10,50,60

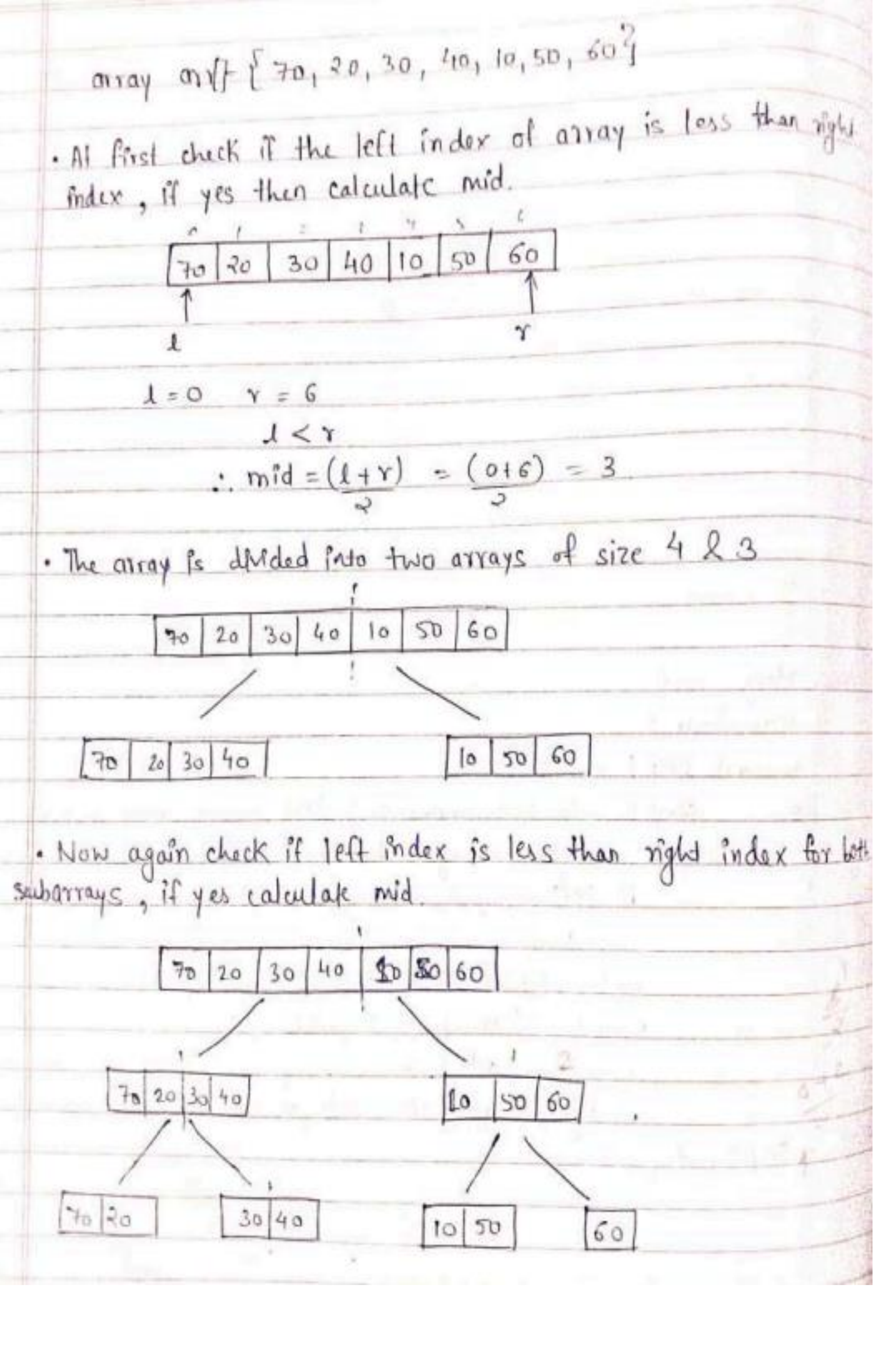
Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.



```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

Time Complexity of Merge Sort is **O(nlogn)**.



array arr[] {70, 20, 30, 40, 10, 50, 60}

- At first check if the left index of array is less than right index, if yes then calculate mid.

0	1	2	3	4	5	6
70	20	30	40	10	50	60
l						r

$l = 0 \quad r = 6$

$l < r$

$\therefore mid = \frac{(l+r)}{2} = \frac{(0+6)}{2} = 3$

- The array is divided into two arrays of size 4 & 3

70 | 20 | 30 | 40 | 10 | 50 | 60

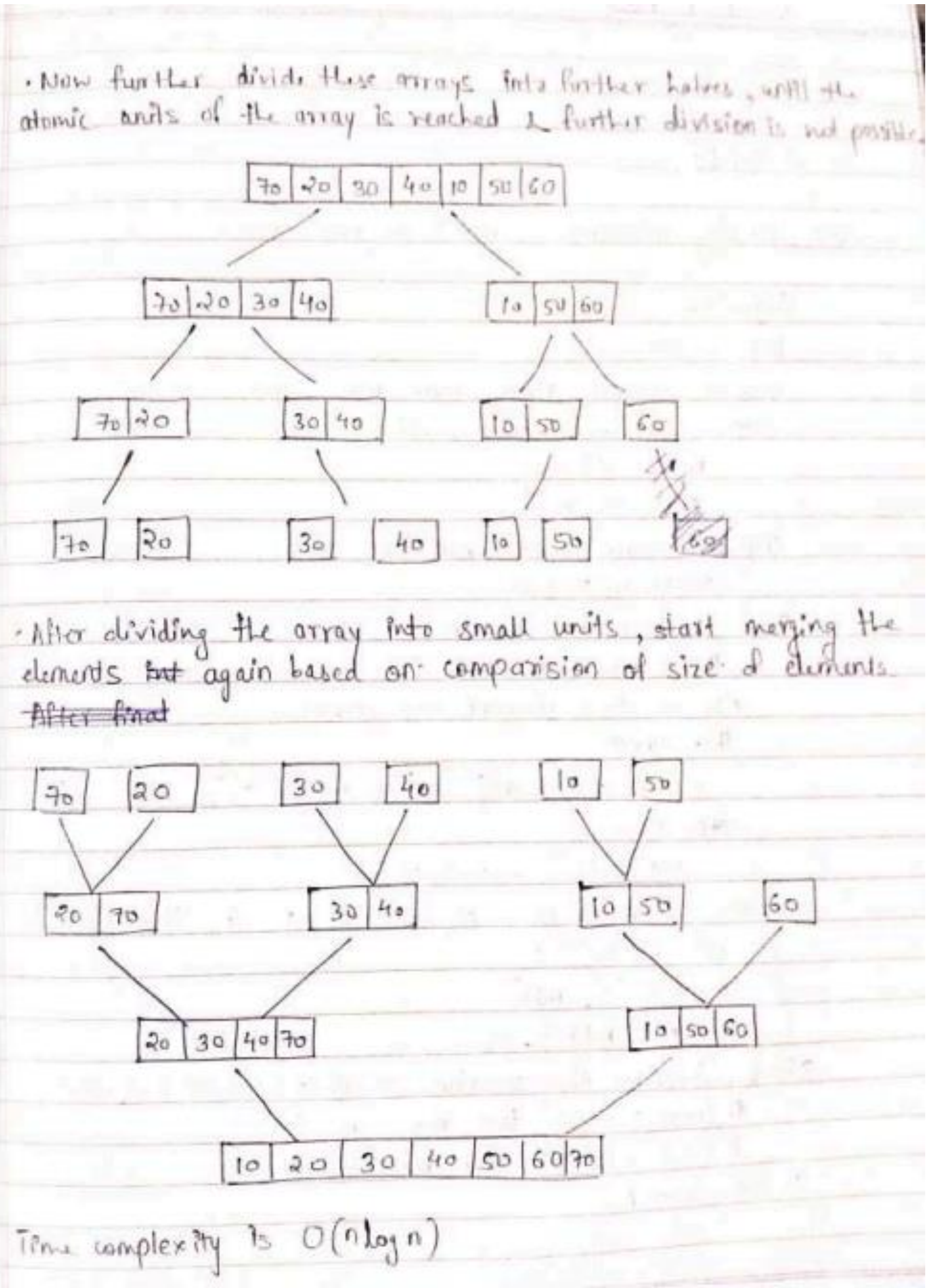
70 | 20 | 30 | 40   10 | 50 | 60

- Now again check if left index is less than right index for both subarrays, if yes calculate mid.

70 | 20 | 30 | 40 | 10 | 50 | 60

70 | 20 | 30 | 40   10 | 50 | 60

70 | 20   30 | 40   10 | 50   60



• Now further divide these arrays into further halves, until the atomic units of the array is reached & further division is not possible.

70 | 20 | 30 | 40 | 10 | 50 | 60
70 | 20 | 30 | 40
10 | 50 | 60
70 | 20
30 | 40
10 | 50
60
70
20
30
40
10
50
60

• After dividing the array into small units, start merging the elements ~~but~~ again based on comparision of size of elements
~~After final~~

70
20
30
40
10
50
20 | 70
30 | 40
10 | 50
60
20 | 30 | 40 | 70
10 | 50 | 60
10 | 20 | 30 | 40 | 50 | 60 | 70

Time complexity is $O(n \log n)$