

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

On

ANALYSIS AND DESIGN OF ALGORITHMS (23CS4PCADA)

Submitted by

DISHA D S (1BM23CS094)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Feb - June 2025

**B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019**
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



This is to certify that the Lab work entitled "**ANALYSIS AND DESIGN OF ALGORITHMS**" carried out by **DISHA D S (1BM23CS094)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of Analysis and Design of Algorithms Lab - **(23CS4PCADA)** work prescribed for the said degree.

Dr. K Panimozhi

Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda

Professor and Head
Department of CSE
BMSCE, Bengaluru

INDEX SHEET

SL NO	Date	Programs	Page no:
1	21/3/25	Merge sort Leetcode 1: Count range sum	4
2	4/4/25	Quick sort Leetcode 2: Kth largest element	13
3	4/4/25	Minimum spanning tree using Prims algorithm Minimum spanning tree using Kruskal's algorithm	21
4	16/5/25	Topological ordering of vertices of a given digraph Leetcode 3: Course Schedule	31
5	16/5/25	0/1 Knapsack using dynamic programming Leetcode 4: Pizza with 3n slices	41
6	16/5/25	All pair shortest paths problem using Floyd's algorithm Leetcode 5: Number of ways to arrive at a destination	50
7	17/5/25	Fractional Knapsack using Greedy technique Leetcode 6: Maximum units on a truck	62
8	17/5/25	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	69
9	23/5/25	Implement "N-Queens Problem" using Backtracking.	73
10	23/5/25	Implement Johnson Trotter algorithm to generate permutations.	79
11	23/5/25	Sort a given set of N integer elements using Heap Sort technique and compute its time taken.	84

COURSE OUTCOMES

CO1	Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations.
CO2	Apply various design techniques for the given problem.
CO3	Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete
CO4	Design efficient algorithms and conduct practical experiments to solve problems.

PROGRAM 1

Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

ALGORITHM:

```
void merge (int arr[], int left, int mid, int right) {  
    int n1 = mid - left + 1;  
    int n2 = right - mid ;  
    int L[n1], R[n2];  
    for (int i=0; i<n1; i++) arr[left+i] = arr[i];  
    for (int j=0; j<n2; j++) arr[mid+j+1] = arr[mid+j];  
    int i=0, j=0, k=left;  
    while (i<n1 && j<n2) {  
        if (L[i] <= R[j]) {  
            arr[k] = L[i];  
            i++;  
        } else {  
            arr[k] = R[j];  
            j++;  
        }  
        k++;  
    }  
}
```

```
while (i<n1) {  
    arr[k] = L[i];  
    i++;  
    k++;  
}  
while (j<n2) {  
    arr[k] = R[j];  
    j++;  
    k++;  
}  
}  
void mergeSort (int arr[], int left, int right) {  
    if (left < right) {  
        int mid = left + (right - left) / 2;  
        mergeSort (arr, left, mid);  
        mergeSort (arr, mid+1, right);  
        merge (arr, left, mid, right);  
    }  
}
```

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
// Function to merge two halves
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temporary arrays
    int L[n1], R[n2];

    // Copy data
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temp arrays
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }
}
```

```

// Copy remaining elements
while (i < n1) {
    arr[k++] = L[i++];
}

while (j < n2) {
    arr[k++] = R[j++];
}

// Merge Sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        mergeSort(arr, left, mid);      // Left half
        mergeSort(arr, mid + 1, right); // Right half

        merge(arr, left, mid, right);   // Merge halves
    }
}

int main() {
    int N, i;
    printf("Enter the number of elements (N): ");
    scanf("%d", &N);

    int arr[N];
    printf("Enter %d integer elements:\n", N);
    for (i = 0; i < N; i++) {

```

```

        scanf("%d", &arr[i]);

    }

clock_t start, end;
double time_taken;

start = clock();
mergeSort(arr, 0, N - 1);
end = clock();

time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

printf("Sorted array:\n");
for (i = 0; i < N; i++) {
    printf("%d ", arr[i]);
}

printf("\nTime taken to sort %d elements: %f seconds\n", N, time_taken);

return 0;
}

```

OUTPUT:

```

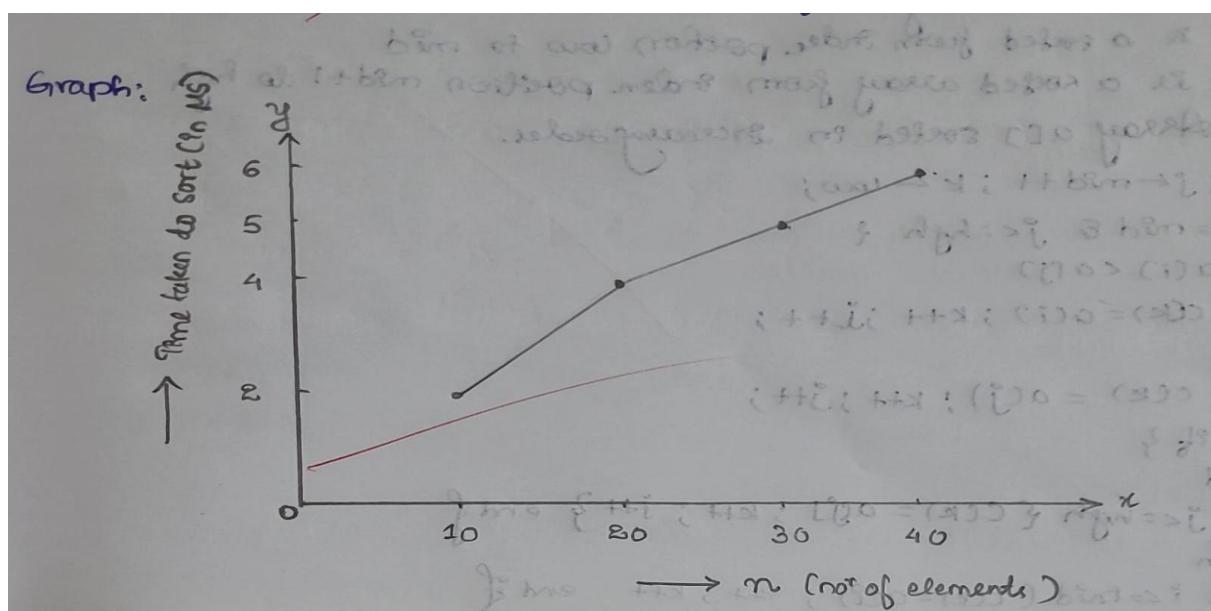
Enter the number of elements (N): 5
Enter 5 integer elements:
1 2 3 4 5
Sorted array:
1 2 3 4 5
Time taken to sort 5 elements: 0.000002 seconds

```

TRACING:

Tracing: $[5, 2, 8]$
 Input: $[5, 2, 8]$
 call: mergeSort $(arr, 0, 2)$
 left half: mergeSort $(arr, 0, 1)$
 right half: mergeSort $(arr, 1, 2)$
 mergeSort $(arr, 10, 11)$
 → mergeSort $(arr, 0, 0)$ $\rightarrow 115$
 → mergeSort $(arr, 1, 1)$ $\rightarrow 112$
 → mergeSort $(arr, 0, 0, 1)$ $\rightarrow 11$ merge $5 \& 2$
 Result: $[2, 5]$
 merge $(arr, 0, 1, 2)$
 compare:
 $2 < 8 \rightarrow arr(0) = 2$
 $5 < 8 \rightarrow arr(1) = 5$
 $arr(2) = 8$.
 Final result: $[2, 5, 8]$

GRAPH:



LEETCODE 1

COUNT RANGE SUM

ALGORITHM

Input:
+ num[] = array of integers
+ lower, upper : the exclusive range of valid subarray sums.

OP:
+ number of subarray sum(s) $\leq \text{sum}$ such that sum lies between lower and upper .

countRangeSum (num[], l, u)
n = length (num[])
prefix = array of size $n + 1$
prefix [0] = 0
for i from 0 to n - 1:
 prefix [i + 1] = prefix [i] + num [i].
return mergeAndCount (prefix, 0, n + 1, l, u)

mergeAndCount (prefix, left, right, l, u):
if avg - left <= 1:
 return 0
mid = (left + right) / 2
count = 0
count += mergeAndCount (prefix, left, mid, lower, upper)
count += mergeAndCount (prefix, mid, right, lower, upper)

j = k = t = mid
temp = empty array
s = 0
for i from left to mid - 1:
 while k < right & prefix [k] - prefix [i] <= lower:
 k += 1
 while j < right & prefix [j] - prefix [i] <= upper:
 j += 1
 count += (j - k)
 while t < right & prefix [t] < prefix [i]:
 append prefix [t] to temp
 t += 1.
 append prefix [i] to temp.
merge remaining elements
copy temp back to prefix [left to left + size of temp]
return count.

CODE:

```
#include <stdio.h>
#include <stdlib.h>

int countWhileMergeSort(long* sum, int left, int right, int lower, int upper) {
    if (right - left <= 1) return 0;

    int mid = (left + right) / 2;
    int count = countWhileMergeSort(sum, left, mid, lower, upper) +
               countWhileMergeSort(sum, mid, right, lower, upper);

    int j = mid, k = mid, t = mid, r = 0;
    long* cache = (long*)malloc((right - left) * sizeof(long));

    for (int i = left; i < mid; ++i) {
        while (k < right && sum[k] - sum[i] < lower) k++;
        while (j < right && sum[j] - sum[i] <= upper) j++;
        count += j - k;

        while (t < right && sum[t] < sum[i])
            cache[r++] = sum[t++];
        cache[r++] = sum[i];
    }

    for (int i = 0; i < t - left; ++i)
        sum[left + i] = cache[i];

    free(cache);
    return count;
}
```

```

int countRangeSum(int* nums, int numsSize, int lower, int upper) {
    long* prefix = (long*)malloc((numsSize + 1) * sizeof(long));
    prefix[0] = 0;
    for (int i = 0; i < numsSize; i++) {
        prefix[i + 1] = prefix[i] + nums[i];
    }

    int result = countWhileMergeSort(prefix, 0, numsSize + 1, lower, upper);
    free(prefix);
    return result;
}

int main() {
    int nums[] = {-2, 5, -1};
    int size = sizeof(nums) / sizeof(nums[0]);
    int lower = -2, upper = 2;

    int result = countRangeSum(nums, size, lower, upper);
    printf("Count of Range Sum in [%d, %d] is: %d\n", lower, upper, result);

    return 0;
}

```

OUTPUT:

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
nums =
[0]
```

```
lower =
0
```

```
upper =
0
```

Output

```
1
```

Expected

```
1
```

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
nums =
[-2, 5, -1]
```

```
lower =
-2
```

```
upper =
2
```

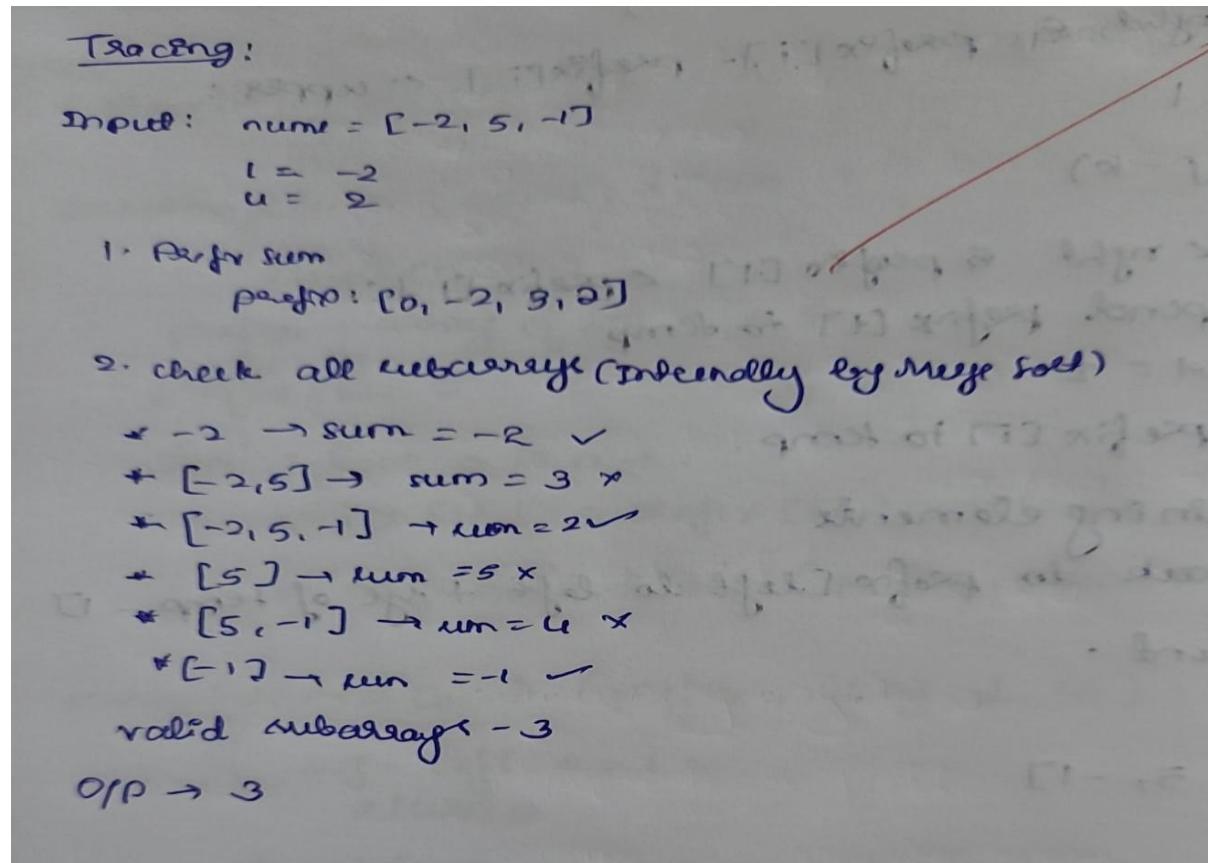
Output

```
3
```

Expected

```
3
```

TRACING:



PROGRAM 2

Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

ALGORITHM:

```
Algorithm:  
Quicksort (a[0...n-1], low, high)  
if low < high  
    // sort only if there are more than two elements in the array  
    pivot_pos = partition (a, low, high) // pivot_pos is a split position  
    Quicksort (a, low, pivot_pos - 1)  
    Quicksort (a, pivot_pos + 1, high).  
end if  
partition (a[0...n-1], low, high)  
// partition the array into parts such that elements towards the left of the pivot element are less than pivot & elements towards right of the pivot are greater than pivot element.  
// Input: A array a[0...n-1] to be sorted from index position low to high.  
// Output: A partition of a[0...n-1] with split position returned  
    // as function's value.  
pivot = a[low]  
i = low + 1  
j = high  
while (1) {  
    while (a[i] <= pivot and i <= high) { i++ }  
    while (a[j] > pivot and j >= low) { j-- }  
    if (i < j)  
        swap a[i] and a[j]  
    else {  
        a[low] = a[j];  
        a[j] = pivot;  
        return j  
    }  
}
```

```
while (a[i] > pivot and j >= low) { j-- }  
if (i < j)  
    swap a[i] and a[j]  
else {  
    a[low] = a[j];  
    a[j] = pivot;  
    return j  
}  
}
```

CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // last element as pivot
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// QuickSort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
```

```

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }

}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    clock_t start = clock();

    quickSort(arr, 0, n - 1);

    clock_t end = clock();
    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;

    printf("\nSorted array:\n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\n\nTime taken: %f seconds\n", time_taken);
    return 0;
}

```

OUTPUT:

Enter number of elements: 5

Enter 5 integers:

56 89 45 2 45

Sorted array:

2 45 45 56 89

Time taken: 0.000002 seconds

TRACING:

Tracing:

Input : Array = [8, 4, 7, 3, 9]
S1: quicksort (a, 0, 4)
pivot = 8

4 < 8 → move left
7 < 8 → move left
3 < 8 → move left
9 < 8 → X do nothing

swap pivot & with last smaller element → 3.
= [3, 4, 7, 8, 9]

S2 left side quicksort (a, 0, 2).
pivot = 3

4 < 3 - X
7 < 3 - X
No swaps

= [3, 4, 7, 8, 9]

quicksort (a, 0, -1) → X
quicksort (a, 1, 2)

S3 quicksort (a, 1, 2)

pivot

7 < 4 → X

No swaps

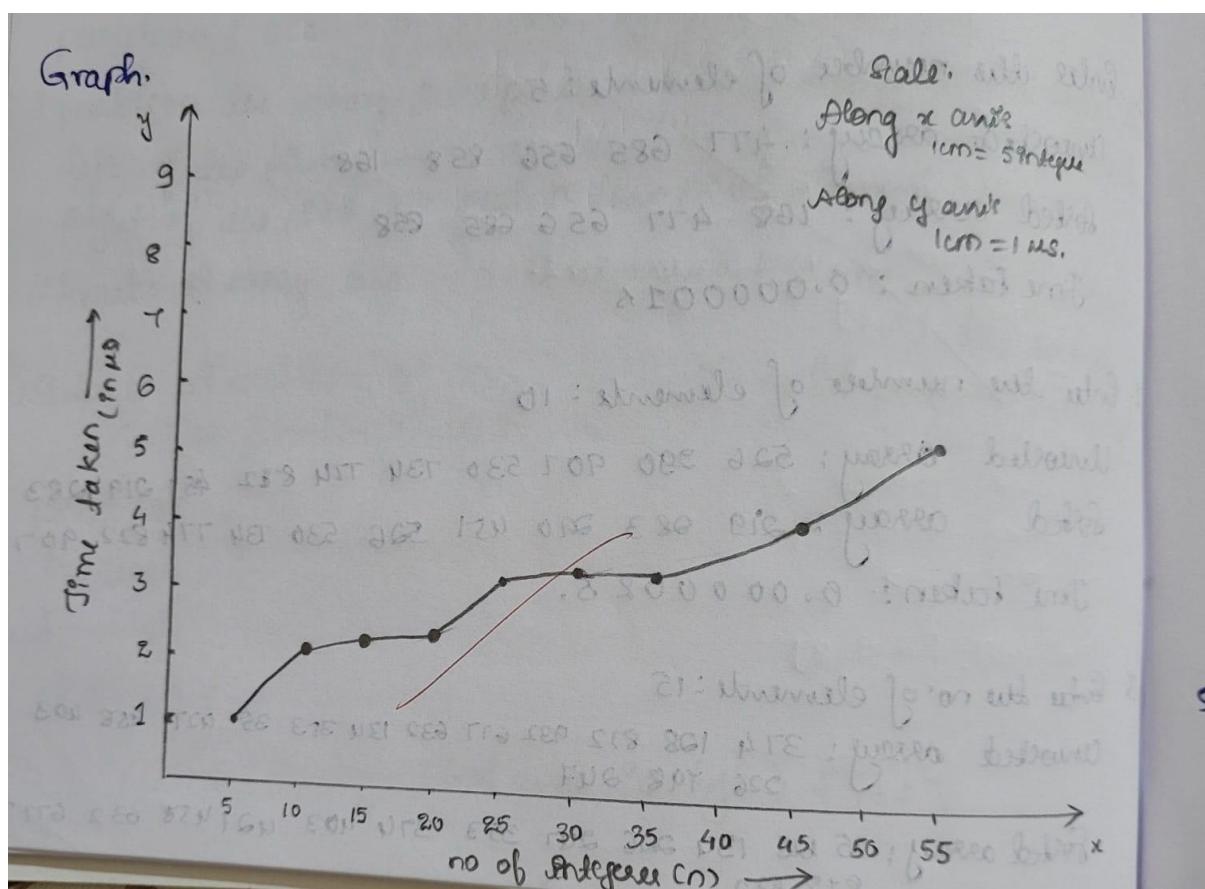
= [3, 4, 7, 8, 9]

S4 Right side quicksort (a, 4, 4)
→ only one element - already sorted.

Final sorted Array: [3, 4, 7, 8, 9].

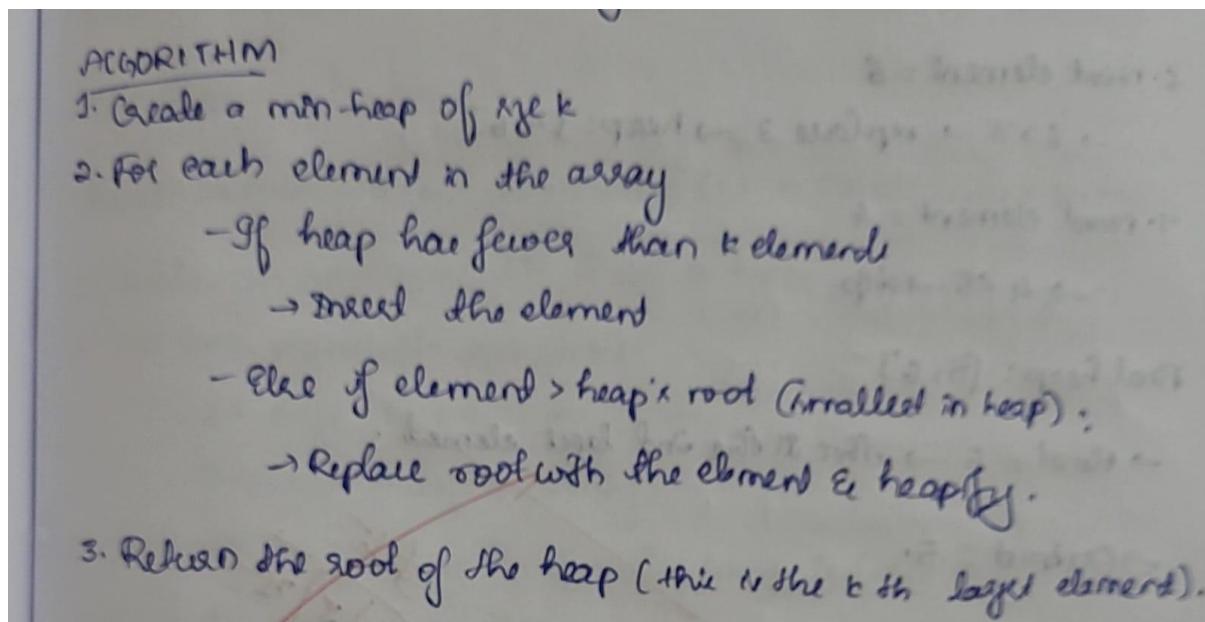
X
4/4/2K

GRAPH:



LEETCODE 2: K th LARGEST ELEMENT IN AN ARRAY

ALGORITHM:



CODE

```
#include <stdlib.h>

// Swap function
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition for QuickSort
int partition(int* nums, int low, int high) {
    int pivot = nums[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (nums[j] <= pivot) {
```

```

        i++;
        swap(&nums[i], &nums[j]);
    }
}

swap(&nums[i + 1], &nums[high]);
return i + 1;
}

// QuickSort
void quickSort(int* nums, int low, int high) {
    if (low < high) {
        int pi = partition(nums, low, high);
        quickSort(nums, low, pi - 1);
        quickSort(nums, pi + 1, high);
    }
}

int findKthLargest(int* nums, int numsSize, int k) {
    quickSort(nums, 0, numsSize - 1);
    return nums[numsSize - k]; // kth largest is (n-k)th smallest in sorted array
}

```

OUTPUT:

Accepted Runtime: 0 ms

- Case 1
- Case 2

Input

```
nums =  
[3,2,1,5,6,4]
```

```
k =  
2
```

Output

```
5
```

Expected

```
5
```

Accepted Runtime: 0 ms

- Case 1
- Case 2

Input

```
nums =  
[3,2,3,1,2,4,5,5,6]
```

```
k =  
4
```

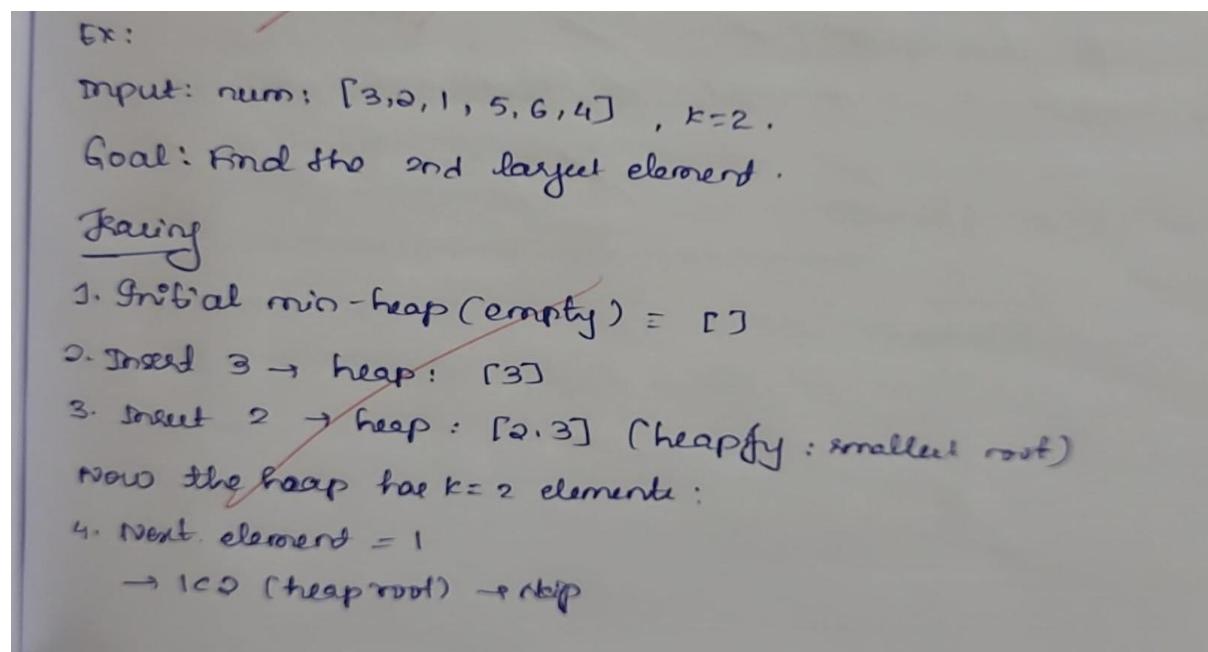
Output

```
4
```

Expected

```
4
```

TRACING:



PROGRAM 3

3.1 Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

ALGORITHM

Algorithm:

```
PRIM(graph, start-node)
1. Create a set MST[] to track nodes included in the MST
2. Create an array key[] and fill with ∞(infinity)
3. Create an array Parent[] to store MST structure
4. Set key[start-node] = 0
5. Repeat (n-1) times:
   a. Pick the vertex u not in MST[] with the smallest
      key[u]
   b. Add u to MST[]
   c. For each vertex v adjacent to u:
      i. If v is not in MST[] and weight(u,v) < key[v]:
         - set key[v] = weight(u,v)
         - set Parent[v] = u
6. Return Parent[] as the MST.
```

CODE

```
#include <stdio.h>
#include <limits.h>

#define MAX 100

int findMinKey(int key[], int mstSet[], int n) {
    int min = INT_MAX, minIndex;
```

```

for (int v = 0; v < n; v++) {
    if (mstSet[v] == 0 && key[v] < min) {
        min = key[v];
        minIndex = v;
    }
}

return minIndex;
}

void primMST(int graph[MAX][MAX], int n) {
    int parent[MAX]; // stores MST
    int key[MAX]; // used to pick minimum weight edge
    int mstSet[MAX]; // included in MST

    for (int i = 0; i < n; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }

    key[0] = 0; // Start from first vertex
    parent[0] = -1; // First node is root

    for (int count = 0; count < n - 1; count++) {
        int u = findMinKey(key, mstSet, n);
        mstSet[u] = 1;

        for (int v = 0; v < n; v++) {
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
}

```

```

        }
    }
}

int totalCost = 0;
printf("\nEdge \tWeight\n");
for (int i = 1; i < n; i++) {
    printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    totalCost += graph[i][parent[i]];
}
printf("\nTotal Cost of MST = %d\n", totalCost);
}

int main() {
    int n;
    int graph[MAX][MAX];

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter the adjacency matrix (use 0 for no edge):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    primMST(graph, n);

    return 0;
}

```

OUTPUT

Enter number of vertices: 5

Enter the adjacency matrix (use 0 for no edge):

0 2 0 6 0

2 0 3 8 5

0 3 0 0 7

6 8 0 0 9

0 5 7 9 0

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

Total Cost of MST = 16

TRACING

Tracing:

vertices: 0, 1, 2, 3, 4

starting node: 0

Initially in MST = { }

S1: Start at node 0.

Available $0-1=2$ ✓ smallest pick.
 $0-3=6$.

MST = {0, 1, 3}

edge = 0-1

S2: Form nodes in MST {0, 1, 3}

Available $1-2=3$ ✓
 $1-4=5$
 $0-3=6$

MST = {0, 1, 2}

Edge: 0-1, 1-2

S3: Form nodes in MST {0, 1, 2, 3}

$1-4=5$ ✓

$0-3=6$

$2-4=7$

MST = {0, 1, 2, 4}

Edge: 0-1, 1-2, 1-4.

S4:

From MST nodes $\{0, 1, 2, 4\}$

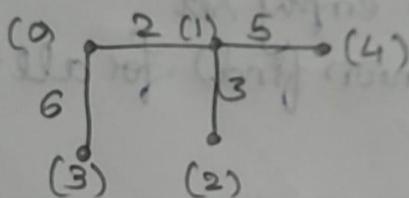
$$0-3 = 6$$

$$3-4 = 9$$

PICK $0-3$

MST: $\{0, 1, 2, 4, 3\} \rightarrow$ All done

edges: $0-1, 1-2, 1-4, 0-3$



3.2 Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

ALGORITHM

1. ~~Input: A connected undirected graph $G(V, E)$ with weight~~
"Input: A connected undirected graph $G(V, E)$ with weight
w_{ij} on each edge."
2. ~~Output: A minimum spanning tree (mst) i.e. a subset of~~
"Output: A minimum spanning tree (mst) i.e. a subset of
edges that connects all vertices with minimum total weight
& without cycles"
3. Sort all edges in non decreasing order of their weight
4. Initialize the mst as an empty set
5. Create a disjoint set (union find) for all vertices to detect cycles.
6. For each edge (u, v) in sorted list:
 - If $\text{find}(u) \neq \text{find}(v)$
 - * Include edge (u, v) in mst
 - * Union the sets of $u \& v$
7. Repeat until mst has $(V-1)$ edges
8. Return the mst

Pseudocode:

KRUSKAL (G):

```
mst = empty set
sort edges of G by increasing weight
for each vertex v in G:
    make_set(v)
for each edge  $(u, v)$  in sorted edges:
    if  $\text{find}(u) \neq \text{find}(v)$ :
        add  $(u, v)$  to mst
        union( $u, v$ )
```

union(u, v)

return mst

CODE

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph
struct Graph {
    int V, E;
    struct Edge edge[MAX];
};

// Find parent of a node (with path compression)
int find(int parent[], int i) {
    if (parent[i] != i)
        parent[i] = find(parent, parent[i]);
    return parent[i];
}

// Union of two sets
void Union(int parent[], int x, int y) {
    parent[x] = y;
}

// Comparator for sorting edges by weight
```

```

int compare(const void* a, const void* b) {
    struct Edge* e1 = (struct Edge*)a;
    struct Edge* e2 = (struct Edge*)b;
    return e1->weight - e2->weight;
}

// Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[MAX];
    int parent[MAX];
    int e = 0; // count of edges in MST
    int i = 0;

    // Initially each node is its own parent
    for (int v = 0; v < V; v++)
        parent[v] = v;

    // Sort edges by weight
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compare);

    while (e < V - 1 && i < graph->E) {
        struct Edge next = graph->edge[i++];
        int x = find(parent, next.src);
        int y = find(parent, next.dest);

        if (x != y) {
            result[e++] = next;
            Union(parent, x, y);
        }
    }
}

```

```

    }

int totalCost = 0;
printf("\nEdge \tWeight\n");
for (i = 0; i < e; ++i) {
    printf("%d - %d \t%d\n", result[i].src, result[i].dest, result[i].weight);
    totalCost += result[i].weight;
}
printf("\nTotal Cost of MST = %d\n", totalCost);

}

int main() {
    struct Graph graph;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &graph.V, &graph.E);

    printf("Enter each edge as: src dest weight\n");
    for (int i = 0; i < graph.E; i++) {
        scanf("%d %d %d", &graph.edge[i].src, &graph.edge[i].dest, &graph.edge[i].weight);
    }

    KruskalMST(&graph);

    return 0;
}

```

OUTPUT

Enter number of vertices and edges: 4 5

Enter each edge as: src dest weight

0 1 10

0 2 6

0 3 5

1 3 15

2 3 4

Edge Weight

2 - 3 4

0 - 3 5

0 - 1 10

Total Cost of MST = 19

TRACING

Tracing:

I/P edges: (0,1,10), (0,2,6), (0,3,5), (1,3,15)

S1: Sort edge by weight.

(2,3,4), (0,3,5), (0,2,6), (0,1,10)

S2: Initialize MST = {}, cost = 0

S3: Process Each Edge

1. (0,3): Not connected \rightarrow add \rightarrow MST = {(2,3)}, cost = 4

2. (0,3): Not connected \rightarrow add \rightarrow MST = {(2,3), (0,3)}, cost = 9

3. (0,2): Form cycle \rightarrow skip

4. (0,1): Not connected \rightarrow add \rightarrow MST = {(2,3), (0,3), (0,1)}, cost = 19

5. (1,3): Form cycle \rightarrow skip

Result:

MST Edges: (2,3), (0,3), (0,1)

Total Cost: 19.

PROGRAM 4

Write program to obtain the Topological ordering of vertices in a given digraph.

ALGORITHM

→ Function Topological sort (Graph) :

1. Input: Graph with v vertices & adjacency list $Adj[v]$

Output: List representing the topological order or a message if a cycle is detected.

1. Initialize indegree ($0 \dots v-1$) to 0

2. For each vertex u in Graph :

for each neighbour v in $Adj[u]$:

$indeg[v] = indeg[v] + 1$

3. Initialize an empty queue Q

4. For each vertex i from 0 to $v-1$:

If $indeg[i] = 0$:

Enqueue (Q, i)

5. Initialize count = 0

6. Initialize empty list $topOrder$

7. While Q is not empty:

$u = Dequeue(Q)$

Append u to $topOrder$.

count = count + 1

For each neighbour v in $Adj[u]$:

$indeg[v] = indeg[v] - 1$

If $indeg[v] == 0$:

Enqueue (Q, v)

8. If $count != v$:

Output "Cycle detected, Topological sort not possible"

Else:

Output Top Order.

CODE

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int adj[MAX][MAX]; // adjacency matrix
int indegree[MAX]; // array to store indegree of vertices
int queue[MAX]; // queue for BFS
int front = -1, rear = -1;

// Add edge from u -> v
void addEdge(int u, int v) {
    adj[u][v] = 1;
    indegree[v]++;
}

// Enqueue
void enqueue(int value) {
    if (rear == MAX - 1)
        return;
    if (front == -1)
        front = 0;
    queue[++rear] = value;
}

// Dequeue
int dequeue() {
    if (front == -1 || front > rear)
        return -1;
```

```

        return queue[front++];
    }

// Topological Sort using Kahn's Algorithm
void topologicalSort(int n) {
    // Enqueue vertices with 0 indegree
    for (int i = 0; i < n; i++)
        if (indegree[i] == 0)
            enqueue(i);

    int count = 0;
    int topoOrder[MAX];

    while (front <= rear) {
        int u = dequeue();
        topoOrder[count++] = u;

        for (int v = 0; v < n; v++) {
            if (adj[u][v]) {
                indegree[v]--;
                if (indegree[v] == 0)
                    enqueue(v);
            }
        }
    }

    if (count != n) {
        printf("Cycle detected! Topological sort not possible.\n");
    } else {
        printf("Topological Order: ");
    }
}

```

```

        for (int i = 0; i < count; i++)
            printf("%d ", topoOrder[i]);
        printf("\n");
    }

}

int main() {
    int n, e, u, v;

    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);

    printf("Enter edges (from to):\n");
    for (int i = 0; i < e; i++) {
        scanf("%d %d", &u, &v);
        addEdge(u, v);
    }

    topologicalSort(n);

    return 0;
}

```

OUTPUT

Enter number of vertices: 6

Enter number of edges: 6

Enter edges (from to):

5 2

5 0

4 0

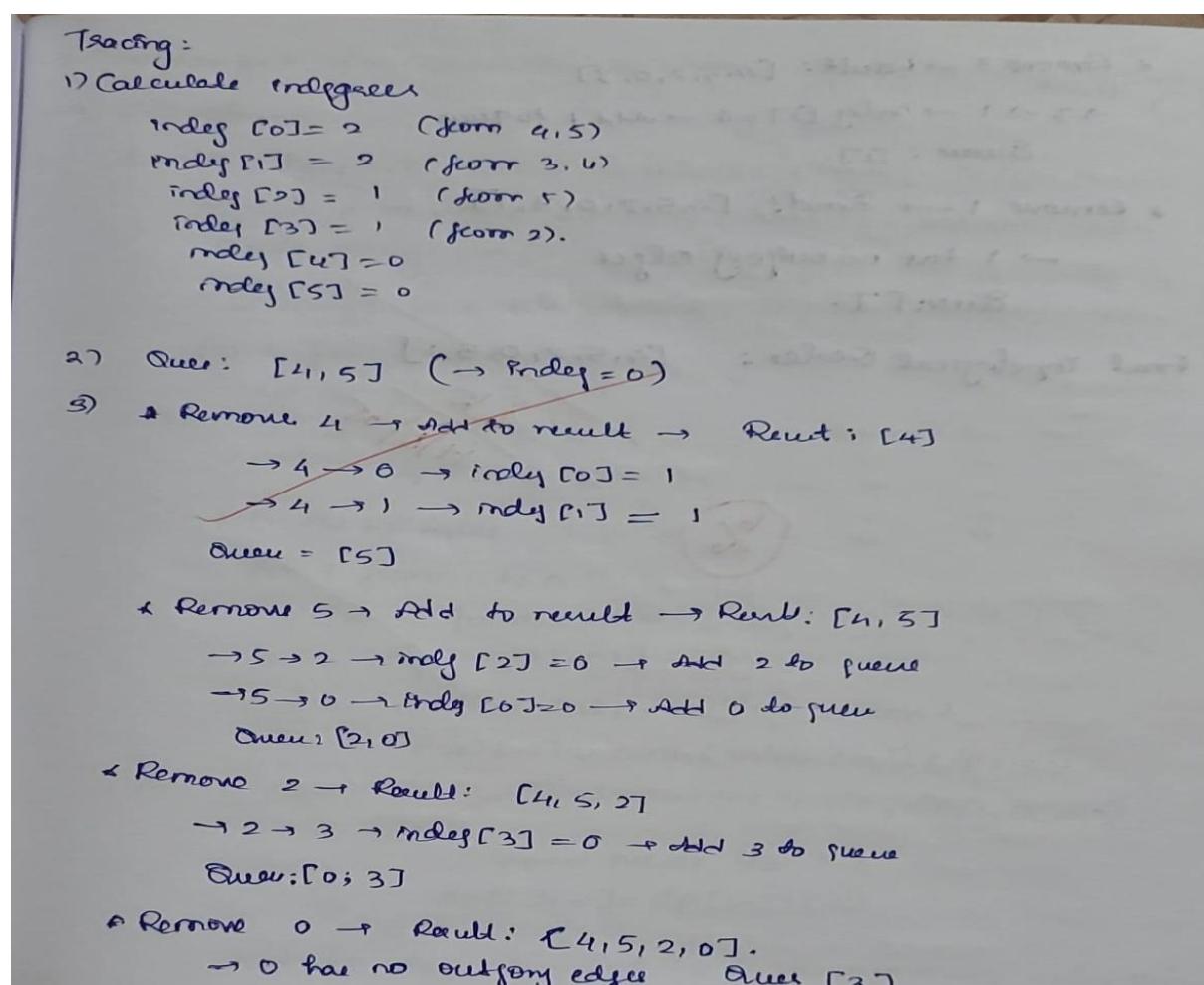
4 1

2 3

3 1

Topological Order: 4 5 0 2 3 1

TRACING



* Remove 3 → Result: [4, 5, 2, 0, 3]

→ 3 → 1 → index [1] = 0 → Add 1 to queue

Queue: [1].

* Remove 1 → Result: [4, 5, 2, 0, 3, 1]

→ 1 has no outgoing edges

Queue: [].

Final topological Order: [4, 5, 2, 0, 3, 1]

(6%)

LEETCODE 3

COURSE SCHEDULE

ALGORITHM

Course Schedule:

Algorithm

Input: numCourses (N), prerequisite (i) = (a, b) means b → a

1. Create an adjacency list & an in-degree array of size N.
2. For each prerequisite pair (a, b):
 - Add edge b → a in the graph
 - Increment in-degree [a] by 1
3. Initialize a queue with all nodes having Prerequisite 0.
4. While the queue is not empty:
 - a. Pop a node u
 - b. For each neighbor v of u:
 - Decrease indegree[v] by 1
 - If indegree[v] = 0, add v to queue
 - c. Count how many nodes were processed.
5. If count == numCourses, return True (possible)
Else return False (Cycle exists)

CODE

```
bool canFinish(int numCourses, int** prerequisites, int prerequisitesSize, int* prerequisitesColSize){  
    // Build graph using adjacency list  
    int* adj[numCourses];
```

```

int adjSize[numCourses];
int indegree[numCourses];

for (int i = 0; i < numCourses; i++) {
    adj[i] = (int*)malloc(sizeof(int) * numCourses); // worst case
    adjSize[i] = 0;
    indegree[i] = 0;
}

// Fill graph and indegree
for (int i = 0; i < prerequisitesSize; i++) {
    int to = prerequisites[i][0];
    int from = prerequisites[i][1];
    adj[from][adjSize[from]++] = to;
    indegree[to]++;
}

// Queue for BFS
int* queue = (int*)malloc(sizeof(int) * numCourses);
int front = 0, rear = 0;

// Push all courses with 0 indegree
for (int i = 0; i < numCourses; i++) {
    if (indegree[i] == 0)
        queue[rear++] = i;
}

int visited = 0;

while (front < rear) {

```

```
int curr = queue[front++];
visited++;

for (int i = 0; i < adjSize[curr]; i++) {
    int neighbor = adj[curr][i];
    indegree[neighbor]--;
    if (indegree[neighbor] == 0)
        queue[rear++] = neighbor;
}

free(queue);
for (int i = 0; i < numCourses; i++) {
    free(adj[i]);
}

return visited == numCourses;
}
```

OUTPUT:

Accepted Runtime: 0 ms

• Case 1

• Case 2

• Case 1

• Case 2

Input

numCourses =

2

prerequisites =

$[[1, 0]]$

Output

true

Expected

true

Input

numCourses =

2

prerequisites =

$[[1, 0], [0, 1]]$

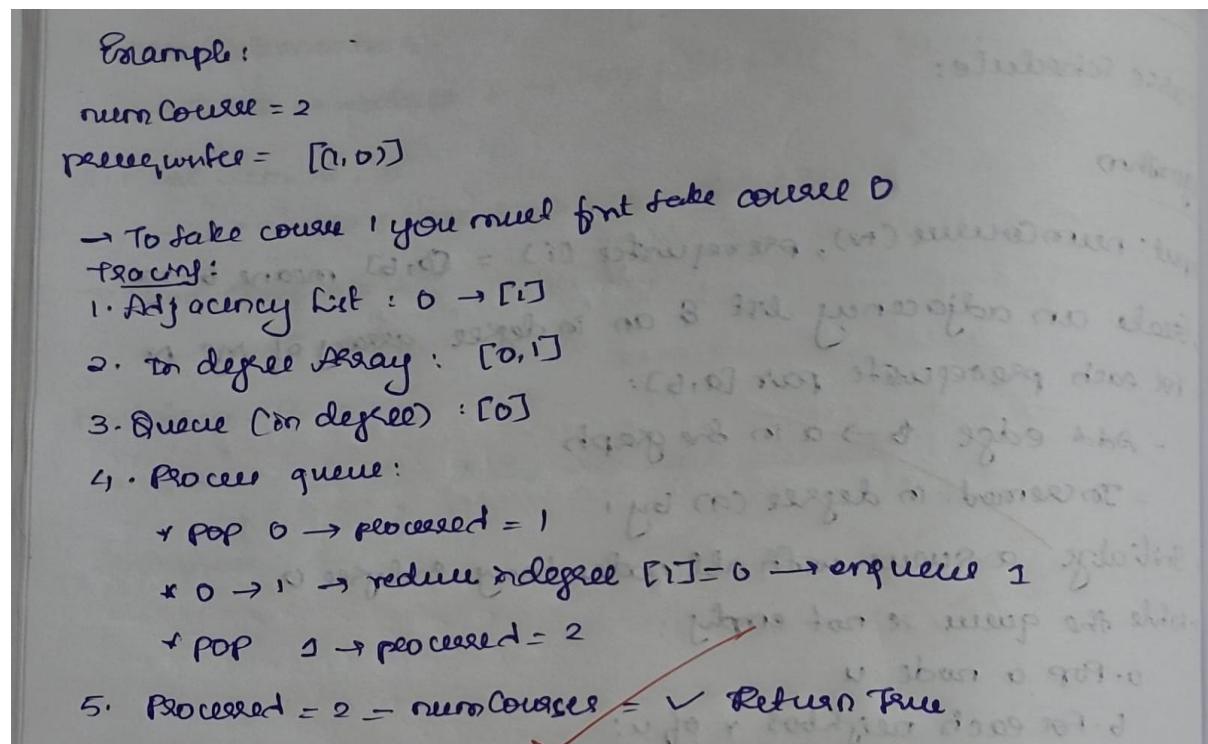
Output

false

Expected

false

TRACING



PROGRAM 5

Implement 0/1 Knapsack problem using dynamic programming.

ALGORITHM

→ Algorithm:

```
function knapsack (weight [], value [], n, w):
    // Create a 2D table dp[n+1][w+1]
    Create array dp [0 ... n] [0 ... w]
    // Initialize base case: zero items or zero capacity
    for i from 0 to n:
        for w from 0 to w:
            if i == 0 or w == 0:
                dp[i][w] = 0
    // Fill the DP table
    for i from 1 to n:
        for w from 1 to w:
            if weight[i-1] <= w:
                // Either take the item or don't
                dp[i][w] = max (value[i-1] + dp[i-1][w - weight[i-1]], dp[i-1][w])
            else:
                // Cannot include item i-1
                dp[i][w] = dp[i-1][w]
    return dp[n][w] // This is the maximum value achievable
```

CODE

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve knapsack problem using DP
int knapsack(int W, int wt[], int val[], int n) {
    int dp[n + 1][W + 1];

    // Build table dp[][] in bottom-up manner
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[n][W];
}

int main() {
    int n, W;

    printf("Enter number of items: ");
}
```

```

scanf("%d", &n);

int wt[n], val[n];

printf("Enter weights of items:\n");
for (int i = 0; i < n; i++)
    scanf("%d", &wt[i]);

printf("Enter values of items:\n");
for (int i = 0; i < n; i++)
    scanf("%d", &val[i]);

printf("Enter capacity of knapsack: ");
scanf("%d", &W);

int maxValue = knapsack(W, wt, val, n);

printf("Maximum value that can be obtained = %d\n", maxValue);

return 0;
}

```

OUTPUT

```

Enter number of items: 4
Enter weights of items:
2 3 4 5
Enter values of items:
3 4 5 6
Enter capacity of knapsack: 5
Maximum value that can be obtained = 7

```

TRACING

Tracing:

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

capacity - 5

S1: Initialize to 0, max val = 0

S2: Add $i=1$ ($w=2, v=3$)

$\text{Cap} \geq 2$, take $i=1$ & $\text{max val} = 3$

S3: Add $i=2$ ($w=3, v=4$)

* check if we can take Pitem 2 alone $\rightarrow \text{val} = 4$

* check if we can take $i_1 + i_2$ together $2+3 = 5 \leq \text{capacity}$

* value if both take $3+4 = 7 \rightarrow > 4 \times 3$.

* Max val = 7.

S4: Add $i=3$ ($w=4, v=5$)

* P3 alone $\rightarrow \text{val} = 5$

* $P_1 + i_3 \rightarrow \text{wt} = 2+4 = 6 \geq \text{Capacity} - \text{rec}$

* $P_2 + i_3 \rightarrow \text{wt} = 3+4 = 7 \geq \text{cap} \rightarrow \text{no}$

So max value stays the same 7.

S5: Add $i=4$ ($w=5, v=6$)

* $i=4$ alone $\rightarrow v = 6$.

$P_3 + P_4 \rightarrow \text{wt} = 2+5 = 7 < \text{capacity} - \text{no}$

$i_2 + P_4 \rightarrow \text{wt} = 3+5 = 8 < \text{capacity} - \text{no}$

max value remains same

So final maximum value = 7

selected items chosen 1 & 2 ($\text{wt } 2+3 = 5$).

(val $3+4 = 7$)

HW	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Backtracking

$i=4, w=5$

$\leftarrow dp[0][5] = dp[3][5] = 7$

4 not included.

$w=5 \quad i=3$

$\leftarrow dp[3][5] = 7 \quad \} \text{equal}$

$dp[2][5] = 7$

3 not included

$w=5 \quad i=2$

$\leftarrow dp[2][5] = 7 \quad \} \text{no equal}$

$dp[1][5] = 3$

2 included

$w = 5 - w(\text{left}[2-1]) = 5 - 3 = 2,$

$i=1$

$\Rightarrow \text{Total weight}$

$= 2 + 3 = 5$

Total value

$= 3 + 4 = 7.$

$\leftarrow dp[1][2] = 3 \neq dp[0][2] = 8$

Item 1 were included.

$w = 2 - w(1-1) = 2 - 2 = 0$

~~34
17/8728~~

(%)

LEETCODE 4

PIZZA WITH 3n SLICES

ALGORITHM

ALGORITHM:

Define: $dp[i][j] = \text{max sum choosing } j \text{ slices from first } i \text{ slices (no adjacent)}$

1. Initialize dp table of size $(len+1) \times (n+1)$ with 0

2. For $i=1$ to len :

 For $j=1$ to n :

$dp[i][j] = \max [$

$dp[i-1][j], \quad \# \text{don't take current}$

$dp[i-2][j-1] + \text{slices}[i-1] \quad \# \text{take current}$

$]$

$)$

3. Return $\max [dp_1[\text{len}-1][n], dp_2[\text{len}-1][n]]$

Ex: slices = [1, 2, 3, 4, 5, 6]

$n = 6/3 = 2 \text{ slices per pack}$

CODE

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int max(int a, int b) {
```

```
    return (a > b) ? a : b;
```

```
}
```

```

// Function to compute the max sum using DP (like 0/1 Knapsack)
int calculate(int* slices, int start, int end, int n) {
    int len = end - start + 1;
    int dp[len + 2][n + 1];
    memset(dp, 0, sizeof(dp));

    for (int i = len - 1; i >= 0; i--) {
        for (int j = 1; j <= n; j++) {
            int take = slices[start + i] + dp[i + 2][j - 1]; // pick slice
            int skip = dp[i + 1][j]; // skip slice
            dp[i][j] = max(take, skip);
        }
    }
    return dp[0][n];
}

int maxSizeSlices(int* slices, int slicesSize) {
    int n = slicesSize / 3;
    int max1 = calculate(slices, 0, slicesSize - 2, n); // exclude last
    int max2 = calculate(slices, 1, slicesSize - 1, n); // exclude first
    return max(max1, max2);
}

```

OUTPUT

Accepted Runtime: 0 ms

- Case 1
- Case 2

Input

```
slices =  
[1,2,3,4,5,6]
```

Output

```
10
```

Expected

```
10
```

Accepted Runtime: 0 ms

- Case 1
- Case 2

Input

```
slices =  
[8,9,8,6,1,1]
```

Output

```
16
```

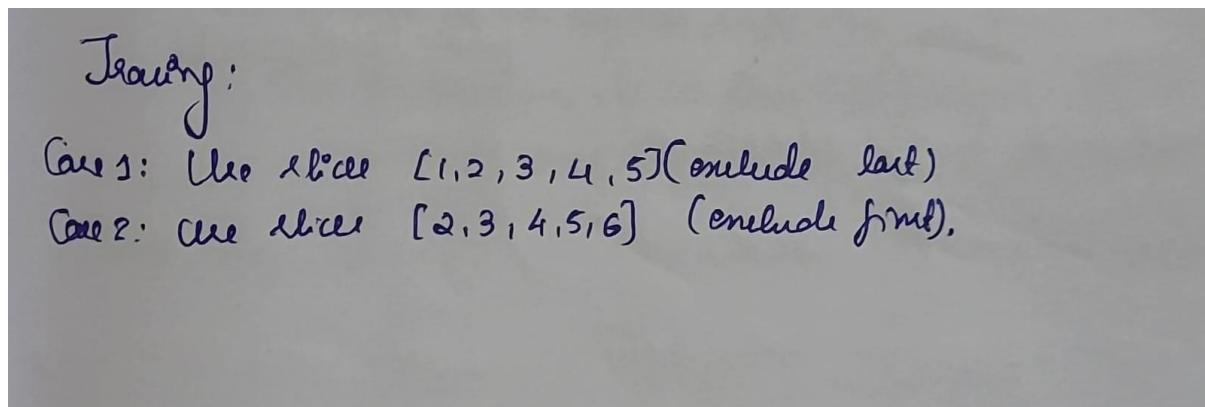
Expected

```
16
```

♥ Contribute a testcase

♥ Contribute a testcase

TRACING



Case 1: [1, 2, 3, 4, 5]

Best 2 non adjacent

$$\star \text{Pick } 1 \& 3 \rightarrow 1+3=4$$

$$\star \text{Pick } 2 \& 4 \rightarrow 2+4=6$$

$$\star \text{Pick } 1 \& 5 \rightarrow 1+5=6$$

$$\star \text{Pick } 3 \& 5 \rightarrow 3+5=8 \quad \checkmark \text{ max.}$$

Case 1 result = 8.

Case 2: [2, 3, 4, 5, 6]

Best 2 non adjacent

$$\star \text{Pick } 2 \& 4 \rightarrow 2+4=6$$

$$\star \text{Pick } 2 \& 5 \rightarrow 2+5=7$$

$$\star \text{Pick } 3 \& 5 \rightarrow 3+5=8$$

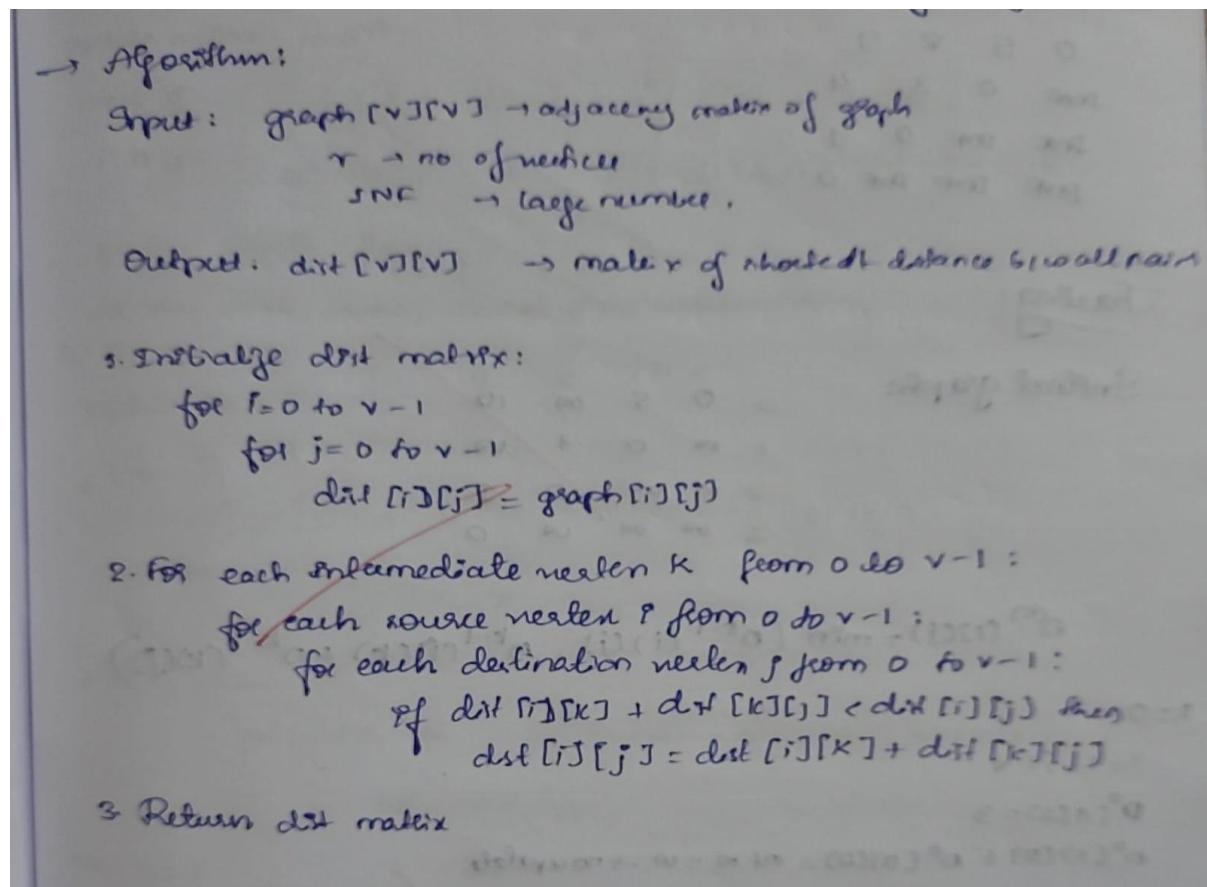
Case 2 result = 9.

Final answer = $\max(8, 9) = 9.$

PROBLEM 6

Implement All Pair Shortest paths problem using Floyd's algorithm.

ALGORITHM



CODE

```
#include <stdio.h>
```

```
#define INF 99999
```

```
#define V 4 // Number of vertices
```

```
// Function to print the shortest distance matrix
```

```
void printSolution(int dist[V][V]) {  
    printf("Shortest distances between every pair of vertices:\n");  
    for (int i = 0; i < V; i++) {  
        for (int j = 0; j < V; j++) {  
            if (dist[i][j] == INF)
```

```

        printf("%7s", "INF");
    else
        printf("%7d", dist[i][j]);
    }
    printf("\n");
}
}

// Floyd-Warshall algorithm
void floydWarshall(int graph[V][V]) {
    int dist[V][V];

    // Initialize the solution matrix with input graph
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    // Update dist[i][j] to the shortest path
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    printSolution(dist);
}

```

```

// Driver code
int main() {
    // Example graph with 4 vertices
    int graph[V][V] = {
        {0, 5, INF, 10},
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}
    };

    floydWarshall(graph);

    return 0;
}

```

OUTPUT:

Shortest distances between every pair of vertices:			
0	5	8	9
INF	0	3	4
INF	INF	0	1
INF	INF	INF	0

TRACING

Tracing:

Initial graph:

	0	1	2	3
0	0	5	∞	10
1	∞	0	3	6
2	∞	∞	0	1
3	∞	∞	∞	0

$D^{(k)}(i)(j) = \min(D^{k-1}(i)(j), D^{k-1}(i)(k) + D^{k-1}(k)(j))$

$k=0$

$i=1, j=2$

$D^0(1)(2) = 3$

$D^0(1)(0) + D^0(0)(2) = 10 + \infty = \infty \rightarrow \text{no update}$

$i=0, j=2$

$D^0(0)(2) = \infty$

$D^0(0)(0) + D^0(0)(2) = 0 + \infty = \infty \rightarrow \text{no update}$

No update since going through 0 doesn't improve any distance.

s2: $k=1$
update makes $\rightarrow D^{(1)}$:
check if $D^1(i)(j) > D^1(i)(l) + D^1(l)(j)$ - yes
 $i=0, j=2 \Rightarrow D^{(1)}(0)(2) = 10, D^1(0)(1) + D^1(1)(2) = 5 + 3 = 8$.
 $8 < \infty$, update $D^2(0)(2) = 8$.
all other pairs, similarly checked is no update here.
s3: $k=2$
update to $D^{(2)}$:
 $i=0, j=3$
 $D^2(0)(3) = 10, D^2(0)(2) + D^2(2)(3) = 8 + 1 = 9$.
 $9 < 10 \Rightarrow D^2(0)(3) = 9$.
 $i=1, j=3$:
 $D^2(1)(3) = \infty, D^2(1)(2) + D^2(2)(3) = 3 + 1 = 4$.
 $D^3(1)(3) = 4$.

s4: $k=3$.
Update to $D^{(3)}$:
when checked $D^3(i)(j) > D^3(i)(l) + D^3(l)(j)$
for $i=0, j=0$ $D(0)(0) = 0, D(0)(3) + D(3)(0) = 9 + 6 = 15 \rightarrow \text{no update}$.
 $i=0, j=3$ $D(0)(3) = 9, D(0)(0) + D(3)(0) = 9 + 0 = 9 \rightarrow \text{same, no update}$.
Similarly when checked for remaining entries, the condition not satisfied. Hence stop.

Final matrix $D^{(n)}$:

$$\begin{bmatrix} 0 & 5 & 8 & 9 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

~~6/6~~

~~8/8~~
~~17/17/25~~

LEETCODE 5

NUMBER OF WAYS TO ARRIVE AT DESTINATION

ALGORITHM

```

# Number of ways to arrive at a destination
Pseudo code:
Initialize graph as adjacency list
Initialize minTime[n] = 0 for all nodes
Initialize ways[n] = 0 for all nodes
Set minTime[0] = 0
Set ways[0] = 1
Create min-heap priority queue pq & insert (0,0) → (time, node)
while pq is not empty:
    time, u = pq.pop()
    If time > minTime[u]: skip
    Continue
    For each neighbor v of u with edge time + t:
        newTime = time + t
        If newTime < minTime[v]:
            minTime[v] = newTime
            ways[v] = ways[u] mod (10^9 + 7)
            pq.push((newTime, v))
        Else if newTime == minTime[v]:
            ways[v] = (ways[v] + ways[u]) mod (10^9 + 7)
Return ways[n - 1]

```

CODE

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MOD 1000000007

```

```

typedef struct {
    int to, weight;
} Edge;

```

```

typedef struct {
    Edge* edges;
    int size, capacity;
} AdjList;

typedef struct {
    long long time;
    int node;
} HeapNode;

typedef struct {
    HeapNode* arr;
    int size;
} MinHeap;

void swap(HeapNode* a, HeapNode* b) {
    HeapNode tmp = *a;
    *a = *b;
    *b = tmp;
}

void heapify_up(MinHeap* h, int i) {
    while (i > 0 && h->arr[i].time < h->arr[(i - 1) / 2].time) {
        swap(&h->arr[i], &h->arr[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

void heapify_down(MinHeap* h, int i) {
    int left, right, smallest;
}

```

```

while (2 * i + 1 < h->size) {
    left = 2 * i + 1;
    right = 2 * i + 2;
    smallest = i;

    if (left < h->size && h->arr[left].time < h->arr[smallest].time)
        smallest = left;
    if (right < h->size && h->arr[right].time < h->arr[smallest].time)
        smallest = right;

    if (smallest == i) break;
    swap(&h->arr[i], &h->arr[smallest]);
    i = smallest;
}

void push(MinHeap* h, long long time, int node) {
    h->arr[h->size++] = (HeapNode){time, node};
    heapify_up(h, h->size - 1);
}

HeapNode pop(MinHeap* h) {
    HeapNode top = h->arr[0];
    h->arr[0] = h->arr[--h->size];
    heapify_down(h, 0);
    return top;
}

int countPaths(int n, int** roads, int roadsSize, int* roadsColSize) {
    // Step 1: Build graph

```

```

AdjList* graph = malloc(n * sizeof(AdjList));
for (int i = 0; i < n; ++i) {
    graph[i].edges = malloc(100 * sizeof(Edge));
    graph[i].size = 0;
    graph[i].capacity = 100;
}

for (int i = 0; i < roadsSize; ++i) {
    int u = roads[i][0], v = roads[i][1], w = roads[i][2];
    graph[u].edges[graph[u].size++] = (Edge){v, w};
    graph[v].edges[graph[v].size++] = (Edge){u, w};
}

// Step 2: Initialize minTime and ways
long long* minTime = malloc(n * sizeof(long long));
int* ways = malloc(n * sizeof(int));
for (int i = 0; i < n; ++i) {
    minTime[i] = LLONG_MAX;
    ways[i] = 0;
}
minTime[0] = 0;
ways[0] = 1;

// Step 3: Min-heap priority queue
MinHeap heap;
heap.arr = malloc(100000 * sizeof(HeapNode));
heap.size = 0;
push(&heap, 0, 0); // (time, node)

while (heap.size > 0) {

```

```

HeapNode curr = pop(&heap);
long long time = curr.time;
int u = curr.node;

if (time > minTime[u]) continue;

for (int i = 0; i < graph[u].size; ++i) {
    int v = graph[u].edges[i].to;
    int t = graph[u].edges[i].weight;
    long long newTime = time + t;

    if (newTime < minTime[v]) {
        minTime[v] = newTime;
        ways[v] = ways[u];
        push(&heap, newTime, v);
    } else if (newTime == minTime[v]) {
        ways[v] = (ways[v] + ways[u]) % MOD;
    }
}

int result = ways[n - 1];

// Clean up
for (int i = 0; i < n; ++i) free(graph[i].edges);
free(graph);
free(minTime);
free(ways);
free(heap.arr);

```

```

    return result;
}

```

OUTPUT

Accepted Runtime: 0 ms

- Case 1
- Case 2

Input

```

n =
7

```

```

roads =
[[0, 6, 7], [0, 1, 2], [1, 2, 3], [1, 3, 3], [6, 3, 1], [6, 5, 1], [2, 5, 1], [0, 4, 5], [4, 6, 2]]

```

Output

```
4
```

Expected

```
4
```

Accepted Runtime: 0 ms

- Case 1
- Case 2

Input

```

n =
2

```

```

roads =
[[1, 0, 10]]

```

Output

```
1
```

Expected

```
1
```

TRACING

Tracing

On: $n=4$

nodes = [0, 1, 2, 3]

dist = [0, 0, 0, 0]

ways = [1, 0, 0, 0]

heap = [(0, 0)]

1) pop(0, 0)

→ neighbour: 1 (wt 1), 2 (wt 2)

check n1:

dist[1] = 0 > 0 + 1 → update dist[1] = 1, ways[1] = ways[0] = 1

push(1, 1)

check n2:

dist[2] = 0 > 0 + 2 → update dist[2] = 2, ways[2] = ways[0] = 1

push(2, 2)

now: dist = [0, 1, 2, 0]

ways = [1, 1, 1, 0]

heap = [(1, 1), (2, 2)]

2) Pop(1, 1)

n = 0, 1, 3

d[0]: d[0] = 0 < 1 + 1(2) → x

n2: d[2] = 2 = 1 + 1(2) → equal distance

Push(3, 3)

dist = [0, 1, 2, 3]

ways = [0, 1, 1]

heap = [(2, 2), (0, 3)]

3) Pop(2, 2)

n: 0 (wt 2), 1 (wt 1), 3 (wt 1)

Check n0

* d[0] = 0 < 2 + 2(1) → no update

Check n1

d[1] = 1 < 2 + 3(1) → no update

Check n3:

d[3] = 2 = 2 + 1(1) → equal distances → ways[3] += ways[2] = 112

Now: $dist = [0, 1, 2, 3]$

ways = $[1, 1, 0, 3]$

heap = $(3, 3)$

4) Pop $(3, 3)$

+ n = 2 (wt 1), 1 (wt 2)

n 2: $d[2] = 2 < 3 + 1 (4)$ → no update

n 3: $d[3] = 1 < 3 + 2 (5)$ → no update

Heap → empty → end.

Result:

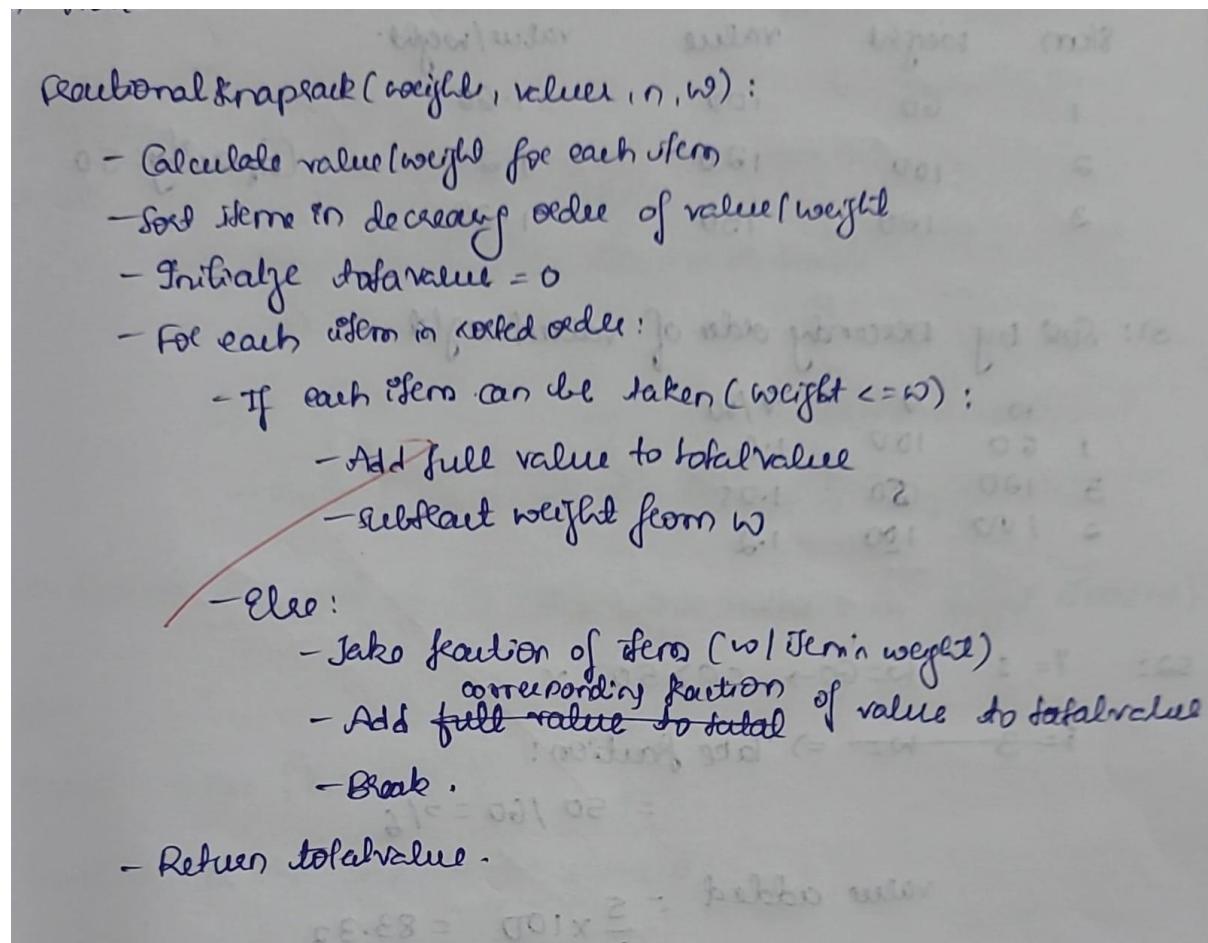
Shortest distance from 0 to 3 is 3

Number of shortest paths ways $[3] = 3$

PROGRAM 7

Implement Fractional Knapsack using Greedy technique.

ALGORITHM



CODE

```
#include <stdio.h>
#include <stdlib.h>

// Structure for an item
struct Item {
    int value, weight;
};

// Function to compare items by value/weight ratio
int compare(const void *a, const void *b) {
```

```

double r1 = (double)((struct Item *)a)->value / ((struct Item *)a)->weight;
double r2 = (double)((struct Item *)b)->value / ((struct Item *)b)->weight;
return (r1 < r2) ? 1 : -1; // Descending order
}

// Function to return maximum value that can be put in knapsack
double fractionalKnapsack(int W, struct Item arr[], int n) {
    // Sort items by value/weight ratio
    qsort(arr, n, sizeof(arr[0]), compare);

    double totalValue = 0.0;

    for (int i = 0; i < n; i++) {
        if (W == 0) break;

        if (arr[i].weight <= W) {
            // Take full item
            W -= arr[i].weight;
            totalValue += arr[i].value;
        } else {
            // Take fractional part
            totalValue += arr[i].value * ((double)W / arr[i].weight);
            break;
        }
    }

    return totalValue;
}

// Main function

```

```
int main() {
    int n, W;

    printf("Enter number of items: ");
    scanf("%d", &n);

    struct Item arr[n];
    printf("Enter value and weight of each item:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &arr[i].value, &arr[i].weight);
    }

    printf("Enter capacity of knapsack: ");
    scanf("%d", &W);

    double maxValue = fractionalKnapsack(W, arr, n);

    printf("Maximum value in knapsack = %.2f\n", maxValue);

    return 0;
}
```

OUTPUT:

```
Enter number of items: 3
Enter value and weight of each item:
100 60
120 100
150 120
Enter capacity of knapsack: 50
Maximum value in knapsack = 83.33
```

TRACING

Tracing:

Item	weight	value	value/weight	Capacity: 50
1	60	100	1.66	
2	100	120	1.2	
3	120	150	1.25	

S1: Sold by secondly order of value/weight

i	w	v/w
1	60	1.66
3	120	1.25
2	100	1.2

S2: $i = 1 \quad w = 60 \rightarrow 60 > 50 \times 1$

~~i=3~~ $w = \Rightarrow$ Take fraction:

$$= 50 / 60 = 5/6$$

$$\text{value added} = \frac{5}{6} \times 100 = 83.33$$

Remaining weight = 0

Knapsack is full.

Final:	Item	Weight Taken	Value Added
	1	50 (partial)	83.33

\Rightarrow Maximum value of knapsack = 83.33.

LEETCODE 6

MAXIMUM UNITS ON A TRUCK

ALGORITHM

Pseudocode

```
Function maxUnits (BoxTypes, truckSize) :  
    Sort BoxTypes in decreasing order by units per box  
    totalUnits = 0  
    For each box in BoxTypes:  
        numBoxes = min (box[0], truckSize)  
        totalUnits += numBoxes * box[1]  
        truckSize -= numBoxes  
        If truckSize == 0:  
            Break  
  
Return totalUnits.
```

CODE

```
int cmp(const void* a, const void* b) {  
    // Sort by units per box in descending order  
    int* boxA = *(int***)a;  
    int* boxB = *(int***)...b;  
    return boxB[1] - boxA[1];  
}  
  
int maximumUnits(int*** boxTypes, int boxTypesSize, int* boxTypesColSize, int truckSize) {  
    qsort(boxTypes, boxTypesSize, sizeof(int*), cmp);  
  
    int totalUnits = 0;  
  
    for (int i = 0; i < boxTypesSize; i++) {  
        int boxCount = boxTypes[i][0];  
        int unitsPerBox = boxTypes[i][1];  
        totalUnits += min(boxCount, truckSize) * unitsPerBox;  
        truckSize -= min(boxCount, truckSize);  
        if (truckSize == 0) break;  
    }  
}
```

```

int numBoxes = boxCount < truckSize ? boxCount : truckSize;
totalUnits += numBoxes * unitsPerBox;
truckSize -= numBoxes;

if (truckSize == 0)
    break;

}

return totalUnits;
}

```

OUTPUT

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```

boxTypes =
[[1,3],[2,2],[3,1]]

```

truckSize =

4

Output

8

Expected

8

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```

boxTypes =
[[5,10],[2,5],[4,7],[3,9]]

```

truckSize =

10

Output

91

Expected

91

TRACING

Example tracing

Input: $\text{boxtypes} = [[5,10], [3,9], [4,7], [2,5]]$
 $\text{trunksize} = 10.$

S1: Sort by width per box (descending):

Sorted boxtypes : $[[5,10], [3,9], [4,7], [2,5]]$

S2: Initialize totallength = 0, trunksize = 10

S3: Loop through sorted box types

Step	Picked Boxes	Width/Box	Used	Remaining Trunk Size	TotalLength
1	[5,10]	10	5	$10 - 5 = 5$	$0 + 50 = 50$
2	[3,9]	9	3	$5 - 3 = 2$	$50 + 27 = 77$
3	[4,7]	7	2	$2 - 2 = 0$	$77 + 14 = 91$

→ trunksize = 0 : stop .

Output:

91.
↗

PROGRAM 8

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

ALGORITHM

→ ALGORITHM:

Dijkstra (Graph, source):

1. Create a set ~~wisted~~ to store visited vertices (prologue to false for all)
2. Create a distance array 'dist[]' and initialize:
 - $\text{dist}[\text{source}] = 0$
 - $\text{dist}[v] = \infty$ for all other vertices $v \neq \text{source}$
3. Repeat $(N-1)$ times:
 - a. u = vertex in $\text{dist}[]$ with the minimum $\text{dist}[u]$ such that $\text{wisted}[u]$ is false
 - b. Mark $\text{wisted}[u] = \text{true}$
 - c. For each neighbour v of u:
 - i. If v is not wisted AND there is an edge from u to v:
 - if $\text{dist}[u] + \text{weight}(u,v) < \text{dist}[v]$:
 - $\text{dist}[v] = \text{dist}[u] + \text{weight}(u,v)$
4. End loop
5. Output the dist[] array (shortest distance from all source)

CODE

```
#include <stdio.h>
#include <limits.h>
#define V 100 // max number of vertices

// Find vertex with minimum distance not yet processed
int minDistance(int dist[], int visited[], int n) {
    int min = INT_MAX, min_index = -1;
    for (int v = 0; v < n; v++) {
        if (!visited[v] && dist[v] <= min) {
```

```

        min = dist[v], min_index = v;
    }
}

return min_index;
}

void dijkstra(int graph[V][V], int src, int n) {
    int dist[V]; // Shortest distances
    int visited[V]; // True if vertex is included in shortest path tree

    // Initialize distances and visited
    for (int i = 0; i < n; i++) {
        dist[i] = INT_MAX;
        visited[i] = 0;
    }

    dist[src] = 0; // Distance to itself is 0

    // Find shortest path for all vertices
    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(dist, visited, n);
        visited[u] = 1;

        // Update distances of adjacent vertices
        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
                dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
}

```

```

    }

// Print results
printf("Vertex \t Distance from Source\n");
for (int i = 0; i < n; i++)
    printf("%d \t\t %d\n", i, dist[i]);
}

// Driver code
int main() {
    int n;
    printf("Enter number of vertices: ");
    scanf("%d", &n);

    int graph[V][V];

    printf("Enter the adjacency matrix (use 0 if no edge):\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    int src;
    printf("Enter source vertex: ");
    scanf("%d", &src);

    dijkstra(graph, src, n);

    return 0;
}

```

OUTPUT

```

Enter number of vertices: 5
Enter the adjacency matrix (use 0 if no edge
):
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
Enter source vertex: 0
Vertex Distance from Source
0 0
1 10
2 50
3 30
4 60

```

TRACING

Tracing:
 Starting vertex from source vertex 0.
 $\text{dist}[v] = \min(\text{dist}[u] + \text{wt}(u,v), \text{dist}[v])$

Step	Current vertex	Distances	Verified
1	0	0 10 INF 30 100	0,1,2,3,4
2	1	0 10 60 30 100	0,1
3	3	0 10 50 30 90	0,1,3
4	2	0 10 50 30 60	0,1,3,2
5	4	0 10 50 30 60	all

PROGRAM 9

Implement “N-Queens Problem” using Backtracking.

ALGORITHM

PSEUDO CODE:

```
→ function solveNQueen (Board, col, N):
    if col == N:
        printSolution (Board, N)
        return true

    res = false
    for each row in 0 to N-1:
        if isSafe (Board, row, col, N):
            board [row][col] = 1
            res = solveNQueen (Board, col+1, N) or res
            board [row][col] = 0

    return res

function isSafe (board, row, col, N):
    for i in 0 to col - 1:
        if board [row][i] == 1:
            return false

    for i = row, j = col ; i >= 0 & j >= 0 ; i--, j--:
        if board [i][j] == 1:
            return false

    for i = row, j = col ; j < N & i >= 0 ; i++, j++:
        if board [i][j] == 1:
            return false

    return true
```

CODE

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 20

int board[MAX];
int N;

int isSafe(int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row))
            return 0;
    }
    return 1;
}

void printSolution() {
    printf("\nSolution:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i] == j)
                printf("Q ");
            else
                printf(".");
        }
        printf("\n");
    }
}
```

```

int solveNQueens(int row) {
    if (row == N) {
        printSolution();
        return 1;
    }

    int found = 0;
    for (int col = 0; col < N; col++) {
        if (isSafe(row, col)) {
            board[row] = col;
            found |= solveNQueens(row + 1); // Try next row
        }
    }

    return found;
}

int main() {
    printf("Enter number of queens (N): ");
    scanf("%d", &N);

    if (N < 1 || N > MAX) {
        printf("N should be between 1 and %d\n", MAX);
        return 1;
    }

    if (!solveNQueens(0))
        printf("No solution exists for N = %d\n", N);

    return 0;
}

```

OUTPUT

Enter number of queens (N): 4

Solution:

. Q . .
. . . Q
Q . . .
. . Q .

Solution:

. . Q .
Q . . .
. . . Q
. Q . .

TRACING

Tracing

Start with
4x4 empty board

S1. Place Q in col 0

Try $\text{r0, c0} \rightarrow$ safe \rightarrow place queen

Q	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Move to col 1

S2: Place Q in col 1

Try $\text{r0, c1} \rightarrow$ check safety
R0 already has queen at C0 \rightarrow not safe

Try $\text{r1, c1} \rightarrow$ Diagonal of (1,1) attacked queen at (0,0) \rightarrow not safe

* Try τ_0, c_0
 * No q in same row, no diagonal attack \rightarrow safe
 Place queen

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

Move to col 2.

S3: Place queen in col 2.

- * Try $\tau_0, c_1 \rightarrow (0,0) q$ attack \rightarrow not safe.
- * Try $\tau_1, c_1 \rightarrow$ diagonal attack from $q \tau_2, c_1 \rightarrow (1,2)$ upper left diagonal \rightarrow not safe
- * Try $\tau_2, c_1 \rightarrow \tau_2$ queen at c_1 attack \rightarrow not safe
- * Try $\tau_3, c_1 \rightarrow$ ^{No} attack \rightarrow safe

Place queen \rightarrow

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix}$$

Move to col 3

S4: Place q in col 3.

- * Try $\tau_0, c_3 \rightarrow (0,0) q$ attack \rightarrow X safe
- * Try $\tau_1, c_3 \rightarrow$ diagonal attack $(2,1) \rightarrow (1,3) \rightarrow$ X safe
- * Try $\tau_2, c_3 \rightarrow \tau_2$ queen at col 1 attack \rightarrow X safe
- * Try $\tau_3, c_3 \rightarrow \tau_3$ q at c_2 \rightarrow X safe attack

No valid pair in col 3 \rightarrow Backtrack

Backtrack S1: Remove q from $(2,1)$,

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

S3 continued: Try next row in col 2

- * No more rows in col 2 \rightarrow Backtrack again.

Backtrack S2: Remove q from $(2,1)$,

$$\rightarrow \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

S2 continued: Try next row in col 1,

- * Try $\tau_3, c_1 \rightarrow$ no attack safe \rightarrow place queen

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

S3 continued: Place queen in col 2.

- * Try $\tau_0, c_2 \rightarrow q(0,0) q$ attack \rightarrow X safe
- * Try $\tau_1, c_2 \rightarrow$ Diagonal attack from $(3,1) \rightarrow$ X safe.
- * Try $\tau_2, c_2 \rightarrow$ No attack \rightarrow safe. \rightarrow place queen

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

S4: Place Q in col 3.

* Try $(0, c_3) \rightarrow (0, 0)$ attack - ✗ Rofe

* Try $(1, c_3) \rightarrow$ no attack - ✓ Rofe

Place queen.

1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0

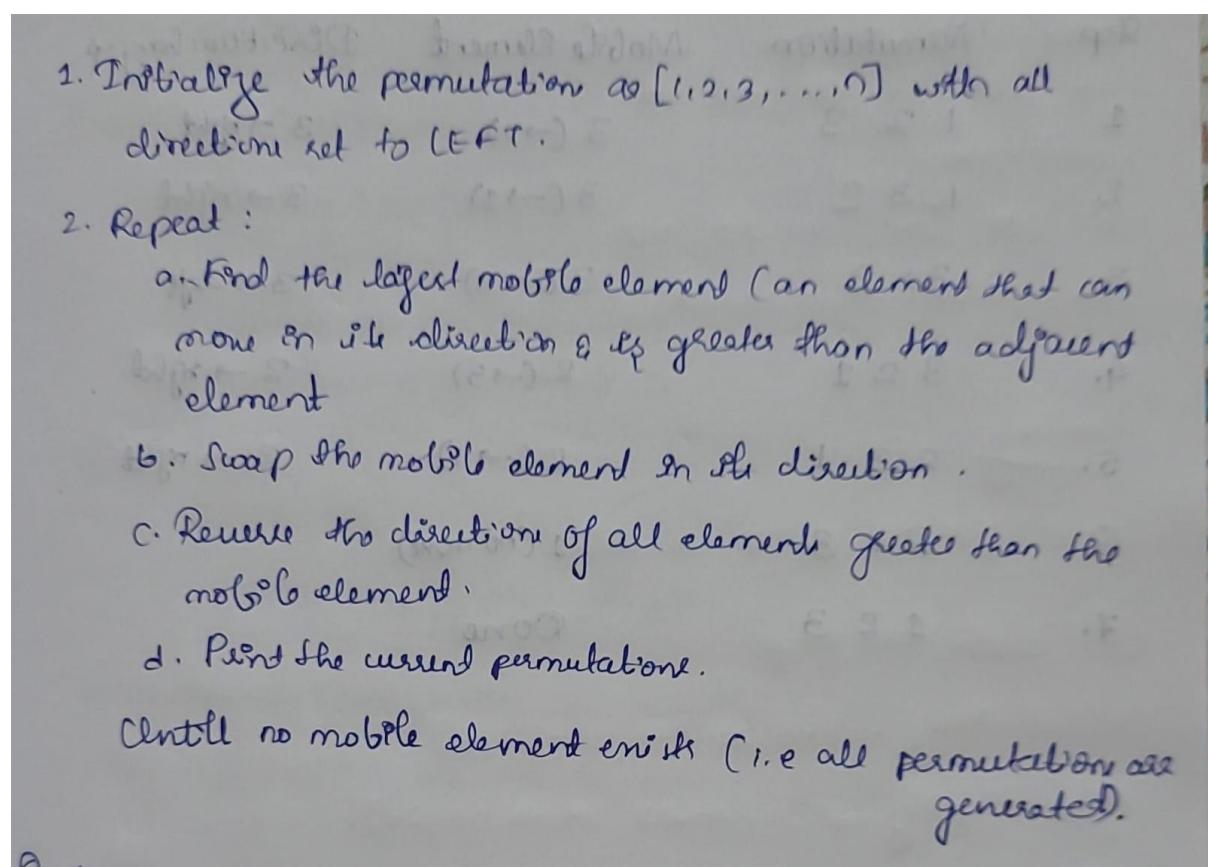
Final step: All queens placed

1	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0

PROGRAM 10

Implement Johnson Trotter algorithm to generate permutations.

ALGORITHM



CODE

```
#include <stdio.h>

#define MAX 20
#define LEFT -1
#define RIGHT 1

int n;
int perm[MAX]; // Current permutation
int dir[MAX]; // Directions: -1 for LEFT, 1 for RIGHT
```

```

// Function to print the current permutation
void printPerm() {
    for (int i = 0; i < n; i++)
        printf("%d ", perm[i]);
    printf("\n");
}

// Function to find the largest mobile integer
int getMobile() {
    int mobile = 0, mobileIndex = -1;

    for (int i = 0; i < n; i++) {
        int next = i + dir[i];
        if (next >= 0 && next < n && perm[i] > perm[next]) {
            if (perm[i] > mobile) {
                mobile = perm[i];
                mobileIndex = i;
            }
        }
    }

    return mobileIndex;
}

// Function to swap two elements and their directions
void swap(int i, int j) {
    int temp = perm[i];
    perm[i] = perm[j];
    perm[j] = temp;
}

```

```

int dtemp = dir[i];
dir[i] = dir[j];
dir[j] = dtemp;

}

void generatePermutations() {
    printPerm(); // First permutation

    while (1) {
        int mobileIndex = getMobile();
        if (mobileIndex == -1)
            break;

        int next = mobileIndex + dir[mobileIndex];
        swap(mobileIndex, next);

        // After moving, change direction of all elements > moved element
        for (int i = 0; i < n; i++) {
            if (perm[i] > perm[next])
                dir[i] *= -1;
        }

        printPerm();
    }
}

int main() {
    printf("Enter number of elements (n): ");
    scanf("%d", &n);
}

```

```
// Initialize permutation and directions
for (int i = 0; i < n; i++) {
    perm[i] = i + 1;
    dir[i] = LEFT;
}

printf("All permutations using Johnson-Trotter:\n");
generatePermutations();

return 0;
}
```

OUTPUT

```
Enter number of elements (n): 3
All permutations using Johnson-Trotter:
1 2 3
1 3 2
3 1 2
3 2 1
2 3 1
2 1 3
```

TRACING

<u>Tracing for $n=3$</u>	<u>Permutation</u>	<u>Mobile Element</u>	<u>Direction change</u>
Step 1	1 2 3	3 ($\rightarrow 2$)	3 \rightarrow right
2	1 3 2	3 ($\rightarrow 1$)	3 \rightarrow right
3	3 1 2	3 ($\rightarrow 2$)	3 \rightarrow left
4.	3 2 1	2 ($\rightarrow 3$)	2 \rightarrow right
5.	2 3 1	2 ($\rightarrow 1$)	2 \rightarrow right
6.	2 1 3	1 (no move)	
7.	1 2 3	Done	

PROGRAM 11

Sort a given set of N integer elements using Heap Sort technique and compute its time taken.

ALGORITHM

Heap-Sort (arr[], n) :

Build-max-heap (arr, n)

for i = n - 1 to 1 :

swap arr[0] with arr[i]

max-heapify (arr, 0, i)

Build-max-heap (arr, n)

for i = n/2 - 1 down to 0 :

max-heapify (arr, i, n)

max-heapify (arr, i, n) :

left = 2 * i + 1

right = 2 * i + 2

largest = i

if left < n & arr[left] > arr[largest]:

largest = left

if right < n & arr[right] > arr[largest]:

largest = right

if largest != i:

swap arr[i] with arr [largest]

max-heapify (arr, largest, n)

CODE

```
#include <stdio.h>
#include <time.h>

// Function to heapify a subtree rooted at index i
void max_heapify(int arr[], int i, int n) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int largest = i;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        max_heapify(arr, largest, n);
    }
}

// Function to build a max heap
void build_max_heap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        max_heapify(arr, i, n);
}
```

```

// Heap Sort function

void heap_sort(int arr[], int n) {
    build_max_heap(arr, n);

    for (int i = n - 1; i >= 1; i--) {
        // Swap arr[0] with arr[i]
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max_heapify on the reduced heap
        max_heapify(arr, 0, i);
    }
}

// Function to print the array

void print_array(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[100], n;
    clock_t start, end;
    double time_taken;

    printf("Enter number of elements: ");
    scanf("%d", &n);
}

```

```
printf("Enter %d elements:\n", n);
for (int i = 0; i < n; i++)
    scanf("%d", &arr[i]);

start = clock(); // Start timing

heap_sort(arr, n);

end = clock(); // End timing
time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

printf("Sorted array:\n");
print_array(arr, n);

printf("Time taken: %f seconds\n", time_taken);

return 0;
}
```

OUTPUT

```
Enter number of elements: 6
Enter 6 elements:
45 20 35 10 50 25
Sorted array:
10 20 25 35 45 50
Time taken: 0.000003 seconds
```

TRACING

Tracing

Input: [45, 20, 35, 10, 50, 25] + count = 6 + 1 = 7 (last index)

(45, 20, 35, 10, 50, 25) [Initial array]

S1: Build max Heap

Initial array : [45, 20, 35, 10, 50, 25] : (45, 20, 35, 10, 50, 25) [Initial array]

After heapify (2): [45, 20, 35, 10, 50, 25] : (45, 20, 35, 10, 50, 25) [Initial array]

After heapify (1): [45, 50, 35, 10, 20, 25] : (45, 20, 35, 10, 50, 25) [Initial array]

After heapify (0): [50, 45, 35, 10, 20, 25] : (45, 20, 35, 10, 50, 25) [Initial array]

S2: sorting phase

Swap 50 \leftrightarrow 25 : [25, 45, 35, 10, 20, 50] : (45, 20, 35, 10, 50, 25) [Initial array]

Heapify : [45, 25, 35, 10, 20, 50]

[Initial array] swap after (3) max popped

Swap 45 \leftrightarrow 20 : [20, 25, 35, 10, 45, 50]

Heapify : [35, 25, 20, 10, 45, 50]

Swap 35 \leftrightarrow 10 : [10, 25, 20, 35, 45, 50]

Heapify : [25, 10, 20, 35, 45, 50]

Swap 25 \leftrightarrow 20 : [20, 10, 25, 35, 45, 50]

Heapify : [20, 10, 25, 35, 45, 50]

Swap 20 \leftrightarrow 10 : [10, 20, 25, 35, 45, 50]

Final Sorted Array

[10, 20, 25, 35, 45, 50].

GIT HUB LINK

https://github.com/DishaDS094/DISHA_DS_1BM23CS094 ADA LAB

LAB OBSERVATION INDEX PAGE: