THE NEOTIA UNIVERSITY

DESIGN ANALYSIS AND ALGORITHMS

ASSIGNMENT

NAME: Aditya Narayan Dalapati

UID: TNU2022053100009

DEPT: C.S.E. (AI & ML)

SEM: VI

INDEX

S.L. No.	Assignment No.	Topic Name	Page No.	Signature
1.	A-1	Iterative binary search	3 – 5	
2.	A-2	Recursive binary search	6 – 8	
3.	A-3	Divide and Conquer	9 – 11	
4.	A-4	Quick Sort	12 – 14	
5.	A-5	Merge Sort	15 – 17	
6.	A-6	Strassen's Matrix Multiplication	18 – 23	
7.	A-7	Floyd-Warshall Algorithm	24 – 29	
8.	A-8	Kruskal Algorithm	30 - 35	
9.	A-9	Prim's Algorithm	37 - 40	
10.	A-10	Dijkstra Algorithm	41 - 45	

Assignment-1

Objective: To learn the concepts of iterative binary search.

Problem statement: Implement iterative binary search.

Underlying concepts: Binary search.

Binary search is an algorithm for locating the position of an element in a sorted list. It inspects the middle element of the sorted list: if equal to the sought value, then the position has been found; otherwise, the upper half or lower half is chosen for farther searching based on whether the sought value is greater than or less than the middle element. The method reduces the number of elements needed to be checked by a factor of two each time, and finds the sought value if it exists in the list or if not determines "not present", in logarithmic time.

Code:

```
#include <stdio.h>

void sort(int* arr, int n){
    for(int i=0; i<n; i++){
        for(int j=0; j<n-i-1; j++){
            if(arr[j]>arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
```

```
            arr[j+1] = temp;
        }
    }
  }
}

int iterative_binary_search(int arr[], int n, int key){
  int low = 0, high = n-1, mid;
  while(low <=high){
    mid = (low+high)/2;
    if(arr[mid] == key)return mid;
    else if(arr[mid]<key)low = mid+1;
    else high = mid-1;
  }
  return -1;
}

int main(){
  int n, key;
  printf("Enter the number of elements in the array: ");
  scanf("%d", &n);
  int arr[n];
  printf("Enter array elements: \n");
  for(int i=0; i<n; i++){
    scanf("%d", &arr[i]);
  }
  sort(arr, n);
```

```
    printf("\nArray after sorting: ");
    for(int i=0; i<n; i++){
        printf("%d ", arr[i]);
    }
    printf("\nEnter the element to be searched: ");
    scanf("%d", &key);
    int idx = iterative_binary_search(arr, n, key);
    if(idx == -1)printf("Element not found\n");
    else printf("Element found at index %d\n", idx);
    return 0;
}
```

Output:

```
Enter the number of elements in the array: 6
Enter array elements:
7 1 2 0 9 3

Array after sorting: 0 1 2 3 7 9
Enter the element to be searched: 7
Element found at index 4
```

Assignment-2

Objective: To learn the concepts of recursive binary search.

Problem statement: Implement recursive binary search.

Underlying concepts: Binary search, recursion.

Binary search is an algorithm for locating the position of an element in a sorted list. It inspects the middle element of the sorted list: if equal to the sought value, then the position has been found; otherwise, the upper half or lower half is chosen for further searching based on whether the sought value is greater than or less than the middle element. The method reduces the number of elements needed to be checked by a factor of two each time, and finds the sought value if it exists in the list or if not determines "not present", in logarithmic time.

Recursion is a process in which a function calls itself or calls a series of other functions that eventually calls the original function.

Code:

```
#include <stdio.h>
void sort(int* arr, int n){
   for(int i=0; i<n; i++){
      for(int j=0; j<n-i-1; j++){
         if(arr[j]>arr[j+1]){
            int temp = arr[j];
```

```
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int recursive_binary_search(int arr[], int key, int low, int high){
    if(low>high)return -1;
    while(low <= high){
        int mid = (low+high)/2;
        if(arr[mid] == key)return mid;
        else if(arr[mid]<key) return recursive_binary_search(arr, key, mid+1,
high);
        else return recursive_binary_search(arr, key, low, mid-1);
    }

}
int main(){
    int n, key;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter array elements: \n");
    for(int i=0; i<n; i++){
        scanf("%d", &arr[i]);
    }
```

```
    sort(arr, n);
    printf("\nArray after sorting: ");
    for(int i=0; i<n; i++){
        printf("%d ", arr[i]);
    }
    printf("\nEnter the element to be searched: ");
    scanf("%d", &key);
    int idx = recursive_binary_search(arr, key, 0, n-1);
    if(idx == -1)printf("Element not found\n");
    else printf("Element found at index %d\n", idx);
    return 0;
}
```

Output:

```
Enter the number of elements in the array: 5
Enter array elements:
9 1 2 0 -8

Array after sorting: -8 0 1 2 9
Enter the element to be searched: 0
Element found at index 1
```

Lab Assignment-3

Objective: To learn the concepts of divide and conquer

Problem statement: Implement divide and conquer strategy to find marimum und maximum elements.

Underlying concepts: Divide and conquer strategy, recursion

In Divide and conquer strategy we divide a large problem into smaller sub problems so that we can solve them easily. If the divided sub problems are still large we further divide them.

Recursion is a process where a function calls itself or calls a series of functions that eventually calls the first one.

Code:

```
#include <stdio.h>

#include <limits.h>


void divide_and_conquer(int arr[], int low, int high, int* min, int* max) {
```

```
    if (low == high) {
        *min = *max = arr[low];
        return;
    }

    if (high == low + 1) {
        if (arr[low] < arr[high]) {
            *min = arr[low];
            *max = arr[high];
        } else {
            *min = arr[high];
            *max = arr[low];
        }
        return;
    }

    int mid = low + (high - low) / 2;

    int left_min = INT_MAX, left_max = INT_MIN;
    int right_min = INT_MAX, right_max = INT_MIN;

    divide_and_conquer(arr, low, mid, &left_min, &left_max);
    divide_and_conquer(arr, mid + 1, high, &right_min, &right_max);

    *min = (left_min < right_min) ? left_min : right_min;
    *max = (left_max > right_max) ? left_max : right_max;
}
```

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

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

    int min = INT_MAX, max = INT_MIN;
    divide_and_conquer(arr, 0, n-1, &min, &max);

    printf("Minimum element: %d\n", min);
    printf("Maximum element: %d\n", max);

    return 0;
}
```

Output:

```
Enter the number of elements: 8
Enter the elements: 0 9 2 -4 9 1 -1 6 3
Minimum element: -4
Maximum element: 9
```

Lab Assignment-4

Objective: To learn the concepts of quick sort.

Problem statement: Implement quick sort method.

Underlying concepts: Quick sort method.

Quick sort sorts by employing a divide and conquer strategy to divide a list into two sublists.

The steps are:

1. Pick an element, called a pivot, from the list.

2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come afler it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base cases of the recursion are lists of size zero or one, which are always sorted.

Code:

```
#include <stdio.h>
void swap(int* a, int* b){
   int temp = *a;
   *a = *b;
   *b = temp;
}
int partition(int* arr, int low, int high){
   int pivot = arr[low];
   int i = low+1, j = high;
   while(i<=j){
      while(arr[i]<=pivot && i<=high){
         i++;
      }
      while(arr[j]>pivot && j>=low){
         j--;
      }
      if(i<j){
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[low], &arr[j]);
   return j;
```

```
}
void quick_sort(int* arr, int low, int high){
    if(low<high){
        int pivotIdx = partition(arr, low, high);
        quick_sort(arr, low, pivotIdx-1);
        quick_sort(arr, pivotIdx+1, high);
    }
}

int main(){
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter array elements: \n");
    for(int i=0; i<n; i++){
        scanf("%d", &arr[i]);
    }
    quick_sort(arr, 0, n-1);
    printf("\nArray after sorting: ");
    for(int i=0; i<n; i++){
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output:

```
Enter the number of elements in the array: 7
Enter array elements:
0 -1 9 8 2 4 2

Array after sorting: -1 0 2 2 4 8 9
```

Assignment-5

Objective: To learn the concepts of merge sort.

Problem statement: Implement Merge sort method

Underlying concepts: Merge sort method.

Conceptually, a merge sort works as follows

1. If the list is of length 0 or 1. then it is already sorted Otherwise:
2. Divide the unsorted list into two sub lists of about half the size.
3. Sort each sublist recursively by re-applying merge sort.
4. Merge the two sub lists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if

they're already sorted (see the merge function below for an example implementation).

Code:

```
#include <stdio.h>
void merge(int* arr, int low, int mid, int high){
   int i = low, j = mid+1;
   int temp[high-low+1], k=0;
   while(i<=mid && j<=high){
      if(arr[i]>arr[j]){
         temp[k] = arr[j];
         j++;
         k++;
      }
      else{
         temp[k] = arr[i];
         i++;
         k++;
      }
   }
   while(i<=mid){
      temp[k] = arr[i];
      i++;
      k++;
   }
   while(j<=high){
      temp[k] = arr[j];
```

```
        j++;
        k++;
    }
    for(int i=low, k=0; i<=high; i++, k++){
        arr[i] = temp[k];
    }
}
void merge_sort(int* arr, int low, int high){
    if(low>=high){
        return;
    }
    if(low<high){
        int mid = (low+high)/2;
        merge_sort(arr, low, mid);
        merge_sort(arr, mid+1, high);
        merge(arr, low, mid, high);
    }
}
int main(){
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter array elements: \n");
    for(int i=0; i<n; i++){
        scanf("%d", &arr[i]);
    }
```

```
    merge_sort(arr, 0, n-1);
    printf("\nArray after sorting: ");
    for(int i=0; i<n; i++){
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output:

```
Enter the number of elements in the array: 6
Enter array elements:
 8 0 -1 2 5 3

Array after sorting: -1 0 2 3 5 8
```

Assignment-6

Objective: To learn the concepts of Strassen's Matri Multiplication.

Problem statement: Implement Strassen's Matrix Multiplication

Underlying concepts: Merge Sort method.

- **Given:** Two N by N matrices A and B.
  Problem: Compute $C = A \times B$
- **Brute Force**

```
for i:=1 to N do
        for j:=1 to N do
                C[i,j]:=0;
                for k:=1 to N do
                        c[i,j] := C[i,j] + A[i,k] * B[k,j]
```

- $O(N^3)$ multiplications
- **Divide and Conquer**

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{21} = P_2 + P_4$$
$$C_{12} = P_3 + P_5$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

$$P_1 = (A_{11} + A_{22}) \bullet (B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22}) \bullet B_{11}$$
$$P_3 = A_{11} \bullet (B_{12} - B_{22})$$
$$P_4 = A_{22} \bullet (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12}) \bullet B_{22}$$
$$P_6 = (A_{21} - A_{11}) \bullet (B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22}) \bullet (B_{21} + B_{22})$$

From,

T(n) = {7T(n/2) + cn if n>1 and c if n=1}

$T(n) = O(n^{\log 7}) = O(n^{2.81})$

Code:

```
#include<bits/stdc++.h>
using namespace std;
const int s = 4;

void add_matrix(int *A, int *B, int *C, int n, int stride) {
   for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         C[i * n + j] = A[i * stride + j] + B[i * stride + j];
      }
   }
}

void sub_matrix(int *A, int *B, int *C, int n, int stride) {
   for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         C[i * n + j] = A[i * stride + j] - B[i * stride + j];
      }
```

```
    }
}

void matmul(int *A, int *B, int *R, int n) {
    if (n == 1) {
        *R = (*A) * (*B);
        return;
    }

    int new_n = n/2;
    int temp1[4], temp2[4];
    int M1[4], M2[4], M3[4], M4[4], M5[4], M6[4], M7[4];

    add_matrix(A, A + (n+1)*new_n, temp1, new_n, n);
    add_matrix(B, B + (n+1)*new_n, temp2, new_n, n);
    matmul(temp1, temp2, M1, new_n);

    add_matrix(A + n*new_n, A + (n+1)*new_n, temp1, new_n, n);
    for(int i = 0; i < new_n; i++)
        memcpy(temp2 + i*new_n, B + i*n, new_n * sizeof(int));
    matmul(temp1, temp2, M2, new_n);

    sub_matrix(B + new_n, B + (n+1)*new_n, temp1, new_n, n);
    for(int i = 0; i < new_n; i++)
        memcpy(temp2 + i*new_n, A + i*n, new_n * sizeof(int));
    matmul(temp2, temp1, M3, new_n);
```

```
    sub_matrix(B + n*new_n, B, temp1, new_n, n);
    for(int i = 0; i < new_n; i++)
        memcpy(temp2 + i*new_n, A + (n+1)*new_n + i*n, new_n * sizeof(int));
    matmul(temp2, temp1, M4, new_n);

    add_matrix(A, A + new_n, temp1, new_n, n);
    for(int i = 0; i < new_n; i++)
        memcpy(temp2 + i*new_n, B + (n+1)*new_n + i*n, new_n * sizeof(int));
    matmul(temp1, temp2, M5, new_n);

    sub_matrix(A + n*new_n, A, temp1, new_n, n);
    add_matrix(B, B + new_n, temp2, new_n, n);
    matmul(temp1, temp2, M6, new_n);

    sub_matrix(A + new_n, A + (n+1)*new_n, temp1, new_n, n);
    add_matrix(B + n*new_n, B + (n+1)*new_n, temp2, new_n, n);
    matmul(temp1, temp2, M7, new_n);

    for(int i = 0; i < new_n; i++) {
        for(int j = 0; j < new_n; j++) {
            R[i*n + j] = M1[i*new_n + j] + M4[i*new_n + j] - M5[i*new_n + j] +
M7[i*new_n + j];
            R[i*n + j + new_n] = M3[i*new_n + j] + M5[i*new_n + j];
            R[(i + new_n)*n + j] = M2[i*new_n + j] + M4[i*new_n + j];
            R[(i + new_n)*n + j + new_n] = M1[i*new_n + j] - M2[i*new_n + j] +
M3[i*new_n + j] + M6[i*new_n + j];
        }
```

```
    }
}

int main() {
    int a[s][s], b[s][s], c[s][s];

    for(int i = 0; i < s; i++) {
        for(int j = 0; j < s; j++) {
            a[i][j] = i + j;
        }
    }
    for(int i = 0; i < s; i++) {
        for(int j = 0; j < s; j++) {
            b[i][j] = i - j;
        }
    }
    memset(c, 0, sizeof(c));
    matmul((int*)a, (int*)b, (int*)c, s);

    cout << "Matrix A:\n";
    for(int i = 0; i < s; i++) {
        for(int j = 0; j < s; j++) {
            cout << a[i][j] << "\t";
        }
        cout << endl;
    }
```

```
    cout << "\nMatrix B:\n";
    for(int i = 0; i < s; i++) {
        for(int j = 0; j < s; j++) {
            cout << b[i][j] << "\t";
        }
        cout << endl;
    }
    cout << "\nResult Matrix C:\n";
    for(int i = 0; i < s; i++) {
        for(int j = 0; j < s; j++) {
            cout << c[i][j] << "\t";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
Matrix A:
0       1       2       3
1       2       3       4
2       3       4       5
3       4       5       6

Matrix B:
0       -1      -2      -3
1       0       -1      -2
2       1       0       -1
3       2       1       0

Result Matrix C:
14      8       2       -4
20      10      0       -10
26      12      -2      -16
32      14      -4      -22
```

Assignment-7

Objective: To learn the concepts of Floyd-Warshall algorithm using dynamic programming.

Problem statement: Implement Floyd- Warshall algorithm.

Underlying concepts: Floyd-Warshall algorithm, Dynamic programming.

The structure of a shortest path

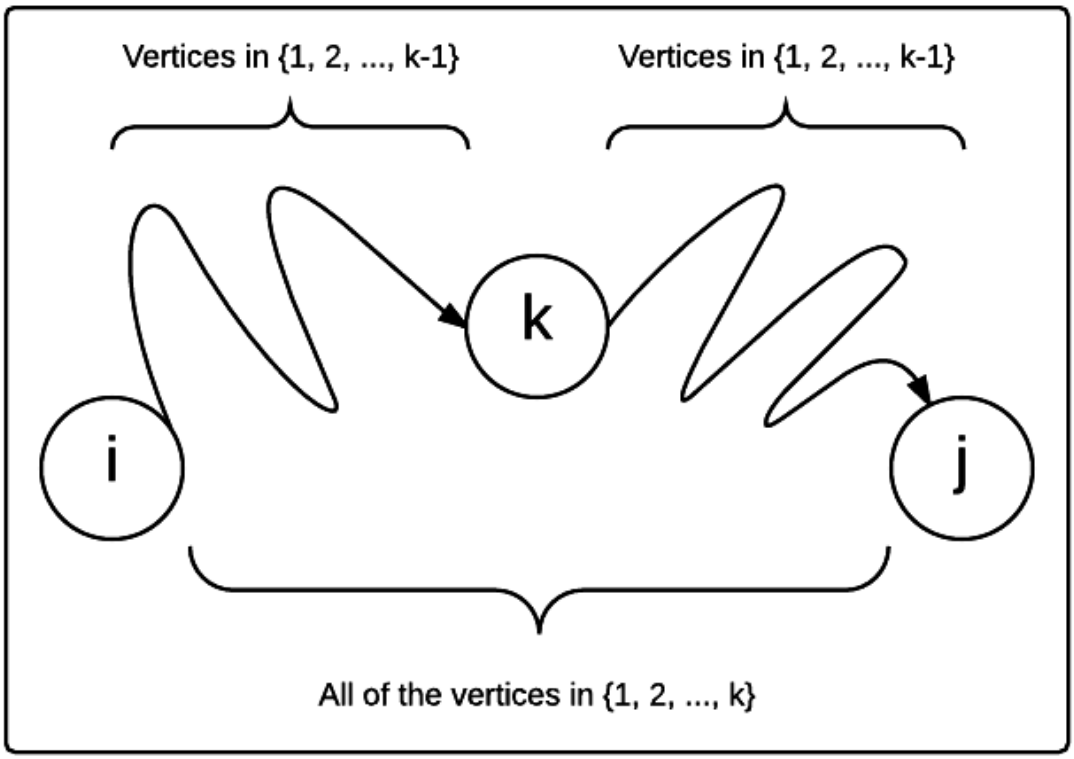
In the Floyd-Warshall algorithm, we use a different characterization of the structure of a shortest path than we used in the matrix-multiplication-based all-pairs algorithms. The algorithm considers the "intermediate" vertices of a shortest

path, where an intermediate vertex of a simple path p = (V1, V2...Vk) is any vertex of p other than V1 or Vk, that is any vertex in the set (V2, V3, Vk-1)

The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of Gare V = (1,2, n), let us consider a subset (1.2.... k) of vertices for some k. For any pair of vertices j, je V, consider all paths from to whose intermediate vertices are all drawn from [1, 2, k), and let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set (1, 2, ….., k-1). The relationship depends on whether or not k is an intermediate vertex of path p.

- If k is not an intermediate vertex of path p. then all intermediate vertices of path p are in the set {1,2,..., k-1). Thus, a shortest path from vertex ị to vertex j with all intermediate vertices in the set (1, 2, ..., k-1) is also a shortest path from i to j with all intermediate vertices in the set (1,2,....k).
- If k is an intermediate vertex of path p, then we break p down into kj shown in Figure, pl is a shortest path from i to k with all intermediate vertices in the set (1,2,k). Because vertex k is not an intermediate vertex of path pl. we see that pl is a shortest path from i to k with all intermediate vertices in the set (12-1) Similarly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set (1,2,k-1)

all intermediate vertices in (1,2,….,k-1).



A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we define a recursive formulation of shortest-path estimates that is different from the one in Section 25.1. Let d be the weight of a shortest path from vertex to vertex for which all intermediate vertices are in the set (1, 2, …, k). When k = 0, a path from vertex (to vertex) with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence d = wij. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

Because for any path, all intermediate vertices are in the set [1,2,n), the matrix D(d) gives the final answer. d = 8(i, j) for all i, j EV.

Computing the shortest-path weights bottom up

Based on above recurrence, the following bottom-up procedure can be used to compute the values d in order of increasing values of k. lts input is an an matrix W defined as

$$w_{ij} = \begin{cases} 0 & \text{if } i=j \\ \text{the weight of directed edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

The procedure returns the matrix D(n) of shortest path weights.

```
FLOYD-WARSHALL(W)
n ← rows[W]
D(0) ← W
for k ←1 to n
    do for i ←1 to n
            do for j ←1 to n
                    d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))

return D(n)
```

Code:

```
#include <bits/stdc++.h>

using namespace std;
```

```
class Solution {
public:
        void shortest_distance(vector<vector<int>>&matrix) {
                int n = matrix.size();
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < n; j++) {
                                if (matrix[i][j] == -1) {
                                        matrix[i][j] = 1e9;
                                }
                                if (i == j) matrix[i][j] = 0;
                        }
                }

                for (int k = 0; k < n; k++) {
                        for (int i = 0; i < n; i++) {
                                for (int j = 0; j < n; j++) {
                                        matrix[i][j] = min(matrix[i][j],
                                                        matrix[i][k] + matrix[k][j]);
                                }
                        }
         cout << "After " << k << " iteration" << endl;
         for (auto row : matrix) {
            for (auto cell : row) {
               if(cell == 1e9) cout << "INF\t";
               else cout << cell << "\t";
            }
            cout << endl;
```

```
        }
            }

            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (matrix[i][j] == 1e9) {
                        matrix[i][j] = -1;
                    }
                }
            }
        }
};
int main() {
        int V = 4;
        vector<vector<int>> matrix(V, vector<int>(V, -1));
   matrix[0][1] = 3;
   matrix[0][3] = 7;
   matrix[1][0] = 8;
   matrix[1][2] = 2;
   matrix[2][0] = 5;
   matrix[2][3] = 1;
   matrix[3][0] = 2;
        Solution obj;
        obj.shortest_distance(matrix);
        return 0;
}
```

Output:

```
After 0 iteration
0       3       INF     7
8       0       2       15
5       8       0       1
2       5       INF     0
After 1 iteration
0       3       5       7
8       0       2       15
5       8       0       1
2       5       7       0
After 2 iteration
0       3       5       6
7       0       2       3
5       8       0       1
2       5       7       0
After 3 iteration
0       3       5       6
5       0       2       3
3       6       0       1
2       5       7       0
```

Assignment – 8

Objective: To learn the concepts of Kruskal's algorithm

Problem statement: Implement Kruskal's algorithm for minimum spanning trees

Underlying concepts: Minimum spanning tree. Kruskal algorithm.

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning, forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

It works as follows:

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph

- while S is nonempty and F is not yet spanning
    - remove an edge with minimum weight from S

- if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
- otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

1 **function** Kruskal(G)

2 Define an elementary cluster C(v)←{v}.

3 Initialize a priority queue Q to contain all edges in G, using the weights as keys.

4 Define a forest T←O //T will ultimately contain the edges of the MST

5 // n is total number of vertices

6 **while** T has fewer than n-1 edges **do**

7 // edge u,v is the minimum weighted route from/to v

8 (u,v) ←Q.removeMin()

9 // prevent cycles in T. add u,v only if T does not already contain a path between u and v.

10 // the vertices has been added to the tree.

11 Let C(v) be the cluster containing v, and let C(a) be the cluster containing u.

12 **if** C(v) != C(u) **then**

13 Add edge (v.) to T.

14 Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u).

15 **return** tree T

Code:

```
#include <bits/stdc++.h>
using namespace std;



class DisjointSet {
```

```
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
```

```
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution
{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        // 1 - 2 wt = 5
        /// 1 - > (2, 5)
        // 2 -> (1, 5)
```

```
        // 5, 1, 2
        // 5, 2, 1
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node, adjNode}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            if (ds.findUPar(u) != ds.findUPar(v)) {
                mstWt += wt;
                ds.unionBySize(u, v);
            }
        }

        return mstWt;
    }
};
```

```
int main() {

    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
    vector<vector<int>> adj[V];
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);

        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int mstWt = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " << mstWt << endl;
    return 0;
}
```

Output:

```
The sum of all the edge weights: 5
```

Lab Assignment 9

Objective: To learn the concepts of Prim's Algorithm

Problem statement: Implement Prim's algorithm for minimum spanning trees

Underlying concepts: Minimum spanning tree. Prim's algorithm.

The minimum spanning tree of a planar graph. Each edge is labeled with its weight, which here is roughly proportional to its length.

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm

The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

- Input: A connected weighted graph with vertices V and edges E.
- Initialize: V_{new} = {x}, where x is an arbitrary node (starting point) from V, E_{new} = {}
- Repeat until V_{new} = V:
    - Choose edge (u,v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, choose arbitrarily but consistently)
    - Add v to V_{new}, add (u, v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

The minimum spanning tree of a planar graph. Each edge is labeled with its weight, which here is roughly proportional to its length.

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which

is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm

The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

- Input: A connected weighted graph with vertices V and edges E.
- Initialize: V_{new} = {x}, where x is an arbitrary node (starting point) from V, E_{new} = {}
- Repeat until V_{new} = V:
    - Choose edge (u,v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, choose arbitrarily but consistently)
    - Add v to V_{new}, add (u, v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

Code:

```
#include <bits/stdc++.h>

using namespace std;

class Solution

{

public:

    //Function to find sum of weights of edges of the Minimum Spanning Tree.

    int spanningTree(int V, vector<vector<int>> adj[])

    {

            priority_queue<pair<int, int>,
```

```
        vector<pair<int, int> >, greater<pair<int, int>>> pq;
        vector<int> vis(V, 0);
        // {wt, node}
        pq.push({0, 0});
        int sum = 0;
        while (!pq.empty()) {
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int wt = it.first;
            if (vis[node] == 1) continue;
            // add it to the mst
            vis[node] = 1;
            sum += wt;
            for (auto it : adj[node]) {
                int adjNode = it[0];
                int edW = it[1];
                if (!vis[adjNode]) {
                    pq.push({edW, adjNode});
                }
            }
        }
        return sum;
    }
};
int main() {
    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2,
2}};
    vector<vector<int>> adj[V];
```

```
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);
        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }
    Solution obj;
    int sum = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " << sum << endl;

    return 0;
}
```

Output:

```
The sum of all the edge weights: 5
```

Lab Assignment 10

Objective: To learn the concept of Dijkstra's Algorithm

Problem statement: Implement single sources shortest path algorithm.

Underlying concepts: Dijkstra's algorithm.

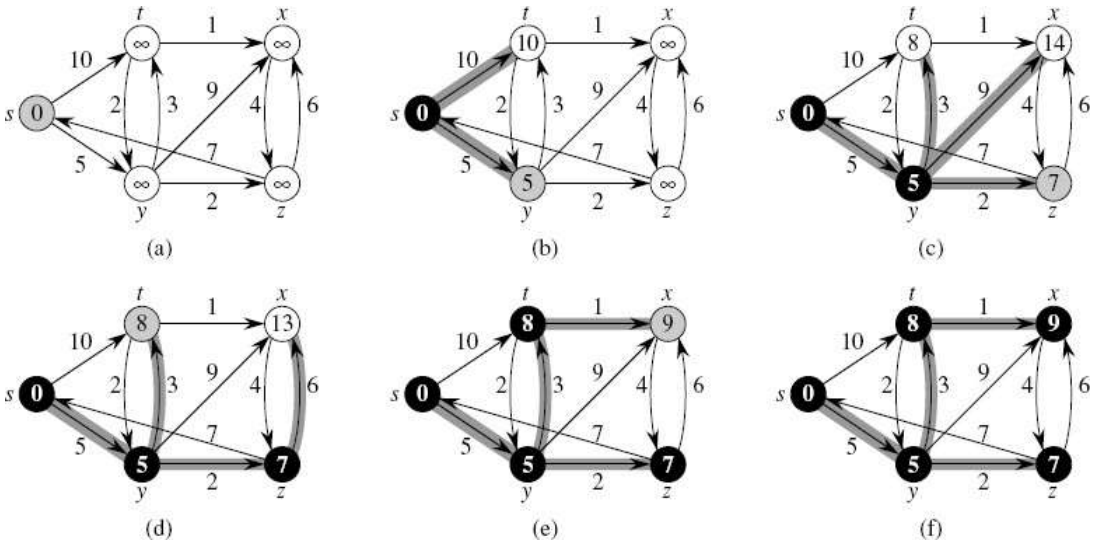
Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. Therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.
Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u
$\in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.
DIJKSTRA*(G,w, s)*

1 INITIALIZE-SINGLE-SOURCE*(G, s)*
2 $S \leftarrow \emptyset$ 3 $Q \leftarrow V[G]$

4 **while** Q != $\emptyset$ **do** $u \leftarrow$ EXTRACT-MIN*(Q)* 6
$S \leftarrow S \cup \{u\}$ 7 **for** each vertex $v \in Adj[u]$ 8
do RELAX*(u, v,w)*

Dijkstra's algorithm relaxes edges as shown



Code:

```
#include <bits/stdc++.h>
using namespace std;


class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
```

```
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
class Solution
{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
```

```
// 1 - 2 wt = 5
/// 1 - > (2, 5)
// 2 -> (1, 5)

// 5, 1, 2
// 5, 2, 1
vector<pair<int, pair<int, int>>> edges;
for (int i = 0; i < V; i++) {
    for (auto it : adj[i]) {
        int adjNode = it[0];
        int wt = it[1];
        int node = i;

        edges.push_back({wt, {node, adjNode}});
    }
}
DisjointSet ds(V);
sort(edges.begin(), edges.end());
int mstWt = 0;
for (auto it : edges) {
    int wt = it.first;
    int u = it.second.first;
    int v = it.second.second;

    if (ds.findUPar(u) != ds.findUPar(v)) {
        mstWt += wt;
        ds.unionBySize(u, v);
    }
}
```

```
        return mstWt;
    }
};

int main() {

    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
    vector<vector<int>> adj[V];
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);

        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int mstWt = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " << mstWt << endl;
    return 0;
}
```

Output:

```
The sum of all the edge weights: 5
```