

Algorithm Analysis & Design Lab (IT-354)

Submitted by: Gaganish Yadav

Enrolment Number: 02016403220

Course- B. Tech CSE (6th Semester)



**University School of Information, Communication and
Technology**

Guru Gobind Singh Indraprastha University

Dwarka, Delhi

2023

Submitted to: Mr. Lalit Yadav

USICT, GGSIPU

S.NO.	NAME OF EXPERIMENT	PAGE NO.	DATE	SIGN
1	Program to implement Counting Sort			
2	Program to implement Bucket Sort			
3	Program to implement Merge Sort			
4	Program to implement Quick Sort			
5	Program to implement Heap Sort			
6	Program to implement Insertion Sort			
7	Program to implement Bubble Sort			
8	Program to implement Selection Sort			
9	Program to implement Radix Sort			
10	Program to implement Linear Search			
11	Program to implement Binary Search			
12	Program to implement Direct Search			

13	Program to implement String Matching			
14	Program to implement Matrix Chain Multiplication			
15	Program to implement Disjoint Set Representation using Linked List			
16	Program to implement Knapsack Problem			
17	Program to implement Huffman Coding			
18	Program to implement DFS			
19	Program to implement BFS			
20	Program to implement Kruskal's and Prim's Algorithm			
21	Program to implement Dijkstra's Algorithm			

Aim: Program to implement Counting Sort

Code:

```
#include <bits/stdc++.h>
#include <string.h>
using namespace std;
#define RANGE 255
void countSort(char arr[])
{
    char output[strlen(arr)];
    int count[RANGE + 1], i;
    memset(count, 0, sizeof(count));
    for (i = 0; arr[i]; ++i)
        ++count[arr[i]];
    for (i = 1; i <= RANGE; ++i)
        count[i] += count[i - 1];
    for (i = 0; arr[i]; ++i) {
        output[count[arr[i]] - 1] = arr[i];
        --count[arr[i]];
    }
    for (i = 0; arr[i]; ++i)
        arr[i] = output[i];
}
int main()
{
    char arr[] = "delhi";
    countSort(arr);

    cout << "Sorted character array is " << arr;
    return 0;
}
```

Output:

```
Sorted character array is dehil
```

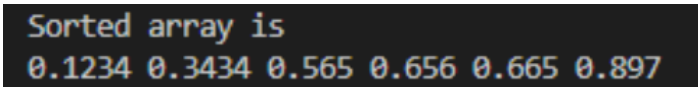
Aim: Program to implement Bucket Sort

Code:

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
void bucketSort(float arr[], int n)
{
    vector<float> b[n];
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i];
        b[bi].push_back(arr[i]);
    }
    for (int i = 0; i < n; i++)
        sort(b[i].begin(), b[i].end());
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}
int main()
{
    float arr[]
        = { 0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bucketSort(arr, n);

    cout << "Sorted array is \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Output:

A screenshot of a terminal window showing the output of the bucket sort program. The text is displayed in a monospaced font with a light blue/cyan color on a dark background. The output consists of two lines: "Sorted array is" followed by a space, and then the sorted array elements "0.1234 0.3434 0.565 0.656 0.665 0.897" on the same line.

```
Sorted array is
0.1234 0.3434 0.565 0.656 0.665 0.897
```

Aim: Program to implement Merge Sort

Code:

```
#include <iostream>
using namespace std;
void merge(int array[], int const left, int const
           mid, int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;
    auto *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne
        = 0,
        indexOfSubArrayTwo
        = 0;
    int indexOfMergedArray
        = left;
    while (indexOfSubArrayOne < subArrayOne
        && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne]
            <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    while (indexOfSubArrayOne < subArrayOne)
    { array[indexOfMergedArray]
      = leftArray[indexOfSubArrayOne];
      indexOfSubArrayOne++;
      indexOfMergedArray++;
    }
    while (indexOfSubArrayTwo < subArrayTwo)
    { array[indexOfMergedArray]
      = rightArray[indexOfSubArrayTwo];
      indexOfSubArrayTwo++;
      indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
```

```

}
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return;

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
void printArray(int A[], int size)
{
    for (auto i = 0; i < size; i++)
        cout << A[i] << " ";
}
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    auto arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}

```

Output:

```

Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13

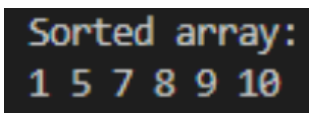
```

Aim: Program to implement Quick Sort

Code:

```
#include <bits/stdc++.h>
using namespace std;
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
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 arr[] = { 10, 7, 8, 9, 1, 5 };
    int N = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, N - 1);
    cout << "Sorted array: " << endl;
    for (int i = 0; i < N; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Output:



```
Sorted array:
1 5 7 8 9 10
```


Aim: Program to implement Heap Sort

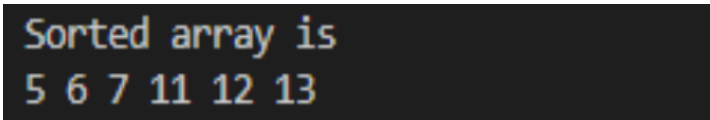
Code:

```
#include <iostream>
using namespace std;
void heapify(int arr[], int N, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < N && arr[l] > arr[largest])
        largest = l;
    if (r < N && arr[r] > arr[largest])
        largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, N, largest);
    }
}
void heapSort(int arr[], int N)
{
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    for (int i = N - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);
    heapSort(arr, N);

    cout << "Sorted array is \n";
    printArray(arr, N);
}
```

Output:



```
Sorted array is
5 6 7 11 12 13
```

Aim: Program to implement Insertion Sort

Code:

```
#include <bits/stdc++.h>
using namespace std;
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int N = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, N);
    printArray(arr, N);

    return 0;
}
```

Output:



5 6 11 12 13

Aim: Program to implement Bubble Sort

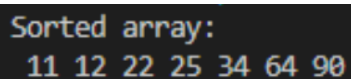
Code:

```
#include <bits/stdc++.h>
using namespace std;
void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << " " << arr[i];
}

int main()
{
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int N = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, N);
    cout << "Sorted array: \n";
    printArray(arr, N);
    return 0;
}
```

Output:



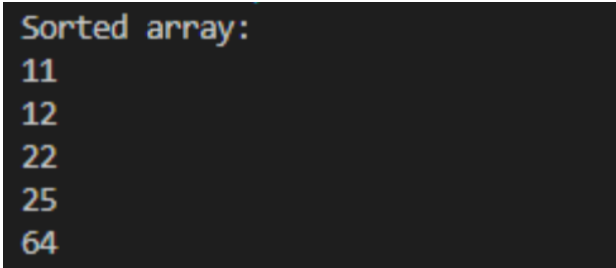
Sorted array:
11 12 22 25 34 64 90

Aim: Program to implement Selection Sort

Code:

```
#include <bits/stdc++.h>
using namespace std;
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    for (i = 0; i < n - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        if (min_idx != i)
            swap(arr[min_idx], arr[i]);
    }
}
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++) {
        cout << arr[i] << " ";
        cout << endl;
    }
}
int main()
{
    int arr[] = { 64, 25, 12, 22, 11 };
    int n = sizeof(arr) / sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Output:



```
Sorted array:
11
12
22
25
64
```

Aim: Program to implement Radix Sort

Code:

```
#include <iostream>
using namespace std;
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}
void countSort(int arr[], int n, int exp)
{
    int output[n];
    int i, count[10] = { 0 };
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
void radixsort(int arr[], int n)
{
    int m = getMax(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}
int main()
{
    int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
    int n = sizeof(arr) / sizeof(arr[0]);
    radixsort(arr, n);
    print(arr, n);
    return 0;
}
```

Output:

2 24 45 66 75 90 170 802

Aim: Program to implement Linear Search

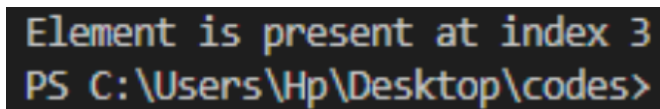
Code:

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

int search(int arr[], int N, int x)
{
    for (int i = 0; i < N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int N = sizeof(arr) / sizeof(arr[0]);
    int result = search(arr, N, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Output:

A screenshot of a terminal window with a black background and yellow text. The first line shows the output of the program: "Element is present at index 3". The second line shows the command prompt: "PS C:\Users\Hp\Desktop\codes>".

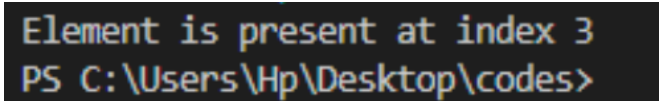
```
Element is present at index 3
PS C:\Users\Hp\Desktop\codes>
```

Aim: Program to implement Binary Search

Code:

```
#include <bits/stdc++.h>
using namespace std;
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
            return m;
        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Output:



```
Element is present at index 3
PS C:\Users\Hp\Desktop\codes>
```

Aim: Program to implement Direct Search

Code:

```
#include<iostream>
#include<map>
using namespace std;

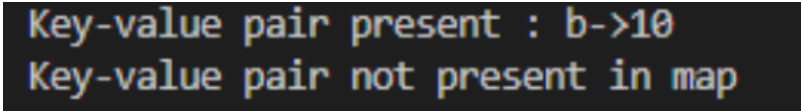
int main()
{
    map< char, int > mp;
    map<char, int>::iterator it ;
    map<char, int>::iterator it1 ;
    mp['a']=5;
    mp['b']=10;
    mp['c']=15;
    mp['d']=20;
    mp['e']=30;
    it = mp.find('b');
    if(it == mp.end())
        cout << "Key-value pair not present in map" ;
    else
        cout << "Key-value pair present : "
        << it->first << "->" << it->second ;

    cout << endl ;
    it1 = mp.find('m');

    if(it1 == mp.end())
        cout << "Key-value pair not present in map" ;
    else
        cout << "Key-value pair present : "
        << it1->first << "->" << it1->second ;

}
```

Output:

A screenshot of a terminal window showing the output of the C++ program. The text is displayed in a monospaced font with a light blue/cyan color on a dark background. The output consists of two lines: "Key-value pair present : b->10" followed by "Key-value pair not present in map".

```
Key-value pair present : b->10
Key-value pair not present in map
```


Aim: Program to implement String Matching

Code:

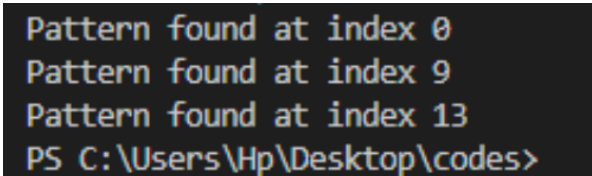
```
#include <bits/stdc++.h>
using namespace std;

void search(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        if (j == M)
            cout << "Pattern found at index " << i << endl;
    }
}

int main()
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:



```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
PS C:\Users\Hp\Desktop\codes>
```

Aim: Program to implement string matching using KMP algorithm

Code:

```
#include <bits/stdc++.h>
void computeLPSArray(char* pat, int M, int* lps);
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int lps[M];
    computeLPSArray(pat, M, lps);

    int i = 0;
    int j = 0;
    while ((N - i) >= (M - j)) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }
        else if (i < N && pat[j] != txt[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

void computeLPSArray(char* pat, int M, int* lps)
{
    int len = 0;
    lps[0] = 0;
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0) {
                len = lps[len - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

```
        }  
    }  
}  
int main()  
{  
    char txt[] = "ABABDABACDABABCABAB";  
    char pat[] = "ABABCABAB";  
    KMPSearch(pat, txt);  
    return 0;  
}
```

Output:

Found pattern at index 10

Aim: Program to implement Matrix Chain Multiplication

Code:

```
#include <bits/stdc++.h>
using namespace std;
int MatrixChainOrder(int p[], int i, int j)
{
    if (i == j)
        return 0;
    int k;
    int mini = INT_MAX;
    int count;
    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k)
                + MatrixChainOrder(p, k + 1, j)
                + p[i - 1] * p[k] * p[j];

        mini = min(count, mini);
    }
    return mini;
}
int main()
{
    int arr[] = { 1, 2, 3, 4, 3 };
    int N = sizeof(arr) / sizeof(arr[0]);
    cout << "Minimum number of multiplications is "
          << MatrixChainOrder(arr, 1, N - 1);
    return 0;
}
```

Output:

```
Minimum number of multiplications is 30
```

Aim: Program to implement Disjoint Set Representation using Linked List

Code:

```
#include <bits/stdc++.h>
using namespace std;
struct Item;
struct Node
{
    int val;
    Node *next;
    Item *itemPtr;
};
struct Item
{
    Node *hd, *tl;
};
class ListSet
{
private:
    unordered_map<int, Node *> nodeAddress;

public:
    void makeset(int a);
    Item* find(int key);
    void Union(Item *i1, Item *i2);
};
void ListSet::makeset(int a)
{
    Item *newSet = new Item;
    newSet->hd = new Node;
    newSet->tl = newSet->hd;
    nodeAddress[a] = newSet->hd;
    newSet->hd->val = a;
    newSet->hd->itemPtr = newSet;
    newSet->hd->next = NULL;
}
Item *ListSet::find(int key)
{
    Node *ptr = nodeAddress[key];
    return (ptr->itemPtr);
}
void ListSet::Union(Item *set1, Item *set2)
{
    Node *cur = set2->hd;
    while (cur != 0)
    {
        cur->itemPtr = set1;
        cur = cur->next;
    }
    (set1->tl)->next = set2->hd;
    set1->tl = set2->tl;
}
```

```

        delete set2;
    }
    int main()
    {
        ListSet a;
        a.makeset(13);
        a.makeset(25);
        a.makeset(45);
        a.makeset(65);

        cout << "find(13): " << a.find(13) << endl;
        cout << "find(25): "
             << a.find(25) << endl;
        cout << "find(65): "
             << a.find(65) << endl;
        cout << "find(45): "
             << a.find(45) << endl << endl;
        cout << "Union(find(65), find(45)) \n";

        a.Union(a.find(65), a.find(45));

        cout << "find(65]): "
             << a.find(65) << endl;
        cout << "find(45]): "
             << a.find(45) << endl;
        return 0;
    }

```

Output:

```

find(13): 0xec35a0
find(25): 0xec3ac0
find(65): 0xec3c80
find(45): 0xec3ba0

Union(find(65), find(45))
find(65]): 0xec3c80
find(45]): 0xec3c80

```

Aim: Program to implement Knapsack

Code:

```
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
    else
        return max(
            val[n - 1]
            + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << knapSack(W, weight, profit, n);
    return 0;
}
```

Output:



220

Aim: Program to implement Huffman Coding

Code:

```
#include <cstdlib>
#include <iostream>
using namespace std;
#define MAX_TREE_HT 100
struct MinHeapNode {
    char data;
    unsigned freq;
    struct MinHeapNode *left, *right;
};
struct MinHeap {
    unsigned size;
    unsigned capacity;
    struct MinHeapNode** array;
};
struct MinHeapNode* newNode(char data, unsigned
freq) {
    struct MinHeapNode* temp = (struct
        MinHeapNode*)malloc( sizeof(struct MinHeapNode));

    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;

    return temp;
}
struct MinHeap* createMinHeap(unsigned
capacity) {

    struct MinHeap* minHeap
        = (struct MinHeap*)malloc(sizeof(struct
        MinHeap)); minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array = (struct MinHeapNode**)malloc(
        minHeap->capacity * sizeof(struct MinHeapNode));
    return minHeap;
}
void swapMinHeapNode(struct MinHeapNode** a,
    struct MinHeapNode** b)

{

    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}
```



```

void minHeapify(struct MinHeap* minHeap, int

idx) {

    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size
        && minHeap->array[left]->freq
            < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size
        && minHeap->array[right]->freq
            < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest],
                        &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

int isSizeOne(struct MinHeap* minHeap)
{

    return (minHeap->size == 1);
}

struct MinHeapNode* extractMin(struct MinHeap*

minHeap) {

    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size -
1];

    --minHeap->size;
    minHeapify(minHeap, 0);

    return temp;
}

void insertMinHeap(struct MinHeap* minHeap,
                    struct MinHeapNode* minHeapNode)

{

    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i
        && minHeapNode->freq

```

```

        < minHeap->array[(i - 1) / 2]->freq) {

    minHeap->array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
}

minHeap->array[i] = minHeapNode;
}
void buildMinHeap(struct MinHeap* minHeap)

{

    int n = minHeap->size - 1;
    int i;

    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        cout << arr[i];

    cout << "\n";
}
int isLeaf(struct MinHeapNode* root)

{

    return !(root->left) && !(root->right);
}
struct MinHeap* createAndBuildMinHeap(char data[], int
                                     freq[], int size)

{

    struct MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;
    buildMinHeap(minHeap);

    return minHeap;
}
struct MinHeapNode* buildHuffmanTree(char data[], int
                                     freq[], int size)

{

    struct MinHeapNode *left, *right, *top;
    struct MinHeap* minHeap

```

```

        = createAndBuildMinHeap(data, freq, size);
while (!isSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);
    top = newNode('$', left->freq + right->freq);

    top->left = left;
    top->right = right;

    insertMinHeap(minHeap, top);
}
return extractMin(minHeap);
}

void printCodes(struct MinHeapNode* root, int
                arr[], int top)

{
    if (root->left) {

        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }
    if (root->right) {

        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
    if (isLeaf(root)) {

        cout << root->data << ": ";
        printArr(arr, top);
    }
}

void HuffmanCodes(char data[], int freq[], int size)

{
    struct MinHeapNode* root
        = buildHuffmanTree(data, freq, size);
    int arr[MAX_TREE_HT], top = 0;

    printCodes(root, arr, top);
}

int main()
{
    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f'
    }; int freq[] = { 5, 9, 12, 13, 16, 45 };

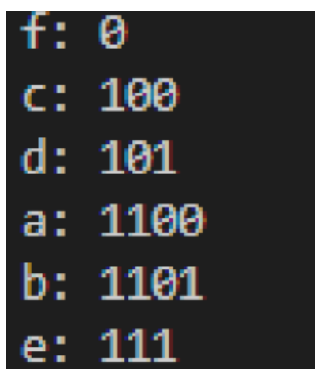
    int size = sizeof(arr) /

    sizeof(arr[0]); HuffmanCodes(arr, freq,

```

```
    size);  
  
    return 0;  
}
```

Output:



```
f: 0  
c: 100  
d: 101  
a: 1100  
b: 1101  
e: 111
```

Aim: Program to implement DFS

Code:

```
#include <bits/stdc++.h>
using namespace std;
class Graph {
public:
    map<int, bool> visited;
    map<int, list<int> > adj;
    void addEdge(int v, int w);
    void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

void Graph::DFS(int v)
{
    visited[v] = true;
    cout << v << " ";
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}

int main()
{
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
         << " (starting from vertex 2) \n";
    g.DFS(2);
    return 0;
}
```

Output:

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Aim: Program to implement BFS

Code:

```
#include <bits/stdc++.h>
using namespace std;
class Graph {
    int V;
    vector<list<int> > adj;

public:
    Graph(int V);
    void addEdge(int v, int w);
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj.resize(V);
}

void Graph::addEdge(int v, int
w) {
    adj[v].push_back(w);
}

void Graph::BFS(int s)
{
    vector<bool> visited;
    visited.resize(V, false);
    list<int> queue;
    visited[s] = true;
    queue.push_back(s);

    while (!queue.empty()) {
        s = queue.front();
        cout << s << " ";
        queue.pop_front();
        for (auto adjacent : adj[s]) {
            if (!visited[adjacent]) {
                visited[adjacent] = true;
                queue.push_back(adjacent);
            }
        }
    }
}

int main()
{
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
}
```

```
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "
      << "(starting from vertex 2) \n";
g.BFS(2);

return 0;
}
```

Output:

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Aim: Program to implement Kruskal's Algorithm

Code:

```
#include <bits/stdc++.h>
using namespace std;
class DSU {
    int* parent;
    int* rank;

public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = -1;
            rank[i] = 1;
        }
    }
    int find(int i)
    {
        if (parent[i] == -1)
            return i;

        return parent[i] = find(parent[i]);
    }
    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2) {
            if (rank[s1] < rank[s2]) {
                parent[s1] = s2;
            }
            else if (rank[s1] > rank[s2]) {
                parent[s2] = s1;
            }
            else {
                parent[s2] = s1;
                rank[s1] += 1;
            }
        }
    }
};

class Graph {
    vector<vector<int> > edgelist;
    int V;

public:
```



```

Graph(int V) { this->V = V; }
void addEdge(int x, int y, int w)
{
    edgelist.push_back({ w, x, y });
}

void kruskals_mst()
{
    sort(edgelist.begin(), edgelist.end());
    DSU s(V);
    int ans = 0;
    cout << "Following are the edges in the "
           "constructed MST"
           << endl;
    for (auto edge : edgelist) {
        int w = edge[0];
        int x = edge[1];
        int y = edge[2];
        if (s.find(x) != s.find(y)) {
            s.unite(x, y);
            ans += w;
            cout << x << " -- " << y << " == " << w
                  << endl;
        }
    }
    cout << "Minimum Cost Spanning Tree: " << ans;
}
};
int main()
{
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);
    g.kruskals_mst();

    return 0;
}

```

Output:

```

Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19

```

Aim: Program to implement Prim's Algorithm

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define V 5
int minKey(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}
void printMST(int parent[], int
graph[V][V]) {
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << "
        \t" << graph[i][parent[i]] << " \n";
}
void primMST(int graph[V][V])
{
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++)
    { int u = minKey(key, mstSet);
      mstSet[u] = true;
      for (int v = 0; v < V; v++)
          if (graph[u][v] && mstSet[v] == false
              && graph[u][v] < key[v])
              parent[v] = u, key[v] = graph[u][v];
    }
    printMST(parent, graph);
}
int main()
{
    int graph[V][V] = { { 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 } };

    primMST(graph);

    return 0;
}
```

}

Output:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

```
#include <iostream>
using namespace std;
#include <limits.h>
#define V 9
int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}
void printSolution(int dist[])
{
    cout << "Vertex \t Distance from Source" <<
    endl; for (int i = 0; i < V; i++)
        cout << i << " \t\t\t\t" << dist[i] <<
    endl; }
void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++)
        { int u = minDistance(dist, sptSet);
          sptSet[u] = true;
          for (int v = 0; v < V; v++)
              if (!sptSet[v] && graph[u][v]
                  && dist[u] != INT_MAX
                  && dist[u] + graph[u][v] < dist[v])
                  dist[v] = dist[u] + graph[u][v];
        }
    printSolution(dist);
}
int main()
{
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        }
```

```
        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };  
dijkstra(graph, 0);  
  
    return 0;  
}
```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14