***EXPERIMENT-1***

**Aim**: - *To implement Bubble Sort, Selection Sort and Insertion Sort and find the time complexity.*

**Source Code:-**

*void swap(int arr[100],int a,int b)*

*{*

*int t=arr[a];*

*arr[a]=arr[b];*

*arr[b]=t;*

*}*

*void bubble\_sort(int arr[100],int size)*

*{*

*float start=clock();*

*int t=0;*

*for(int i=1;i<=size-1;i++)*

*{*

*for(int j=1;j<=size-1;j++)*

*{*

*if(arr[j-1]>=arr[j])*

*{*

*swap(arr,j-1,j);*

*}*

*}*

*}*

*cout<<endl<<"Time Complexity is:"<<(((float)clock()-start)/CLOCKS\_PER\_SEC);*

*}*

*void selection\_sort(int arr[100],int size)*

*{*

*float start=clock();*

*int min=0;*

*for (int i=0;i<size;i++)*

*{*

*min=i;*

*for(int j=i;j<size;j++)*

*{*

*if (arr[j]<=arr[min])*

*{*

*min=j;*

*}*

*swap(arr,i,min);*

*}*

*}*

*cout<<endl<<"Time Complexity is:"<<(((float)clock()-start)/CLOCKS\_PER\_SEC);*

*}*

*void insertion\_sort(int arr[100],int size)*

*{*

*float start=clock();*

*int temp=0;*

*int j=0;*

*for(int i=1;i<size;i++)*

*{*

*temp=i;*

*j=i-1;*

*while(arr[temp]<=arr[j] && j>=0)*

*{*

*// cout<<"Temp and j "<<temp<<" "<<j<<endl;*

*// cout<<"Array Elements: "<<arr[temp]<<" "<<arr[j]<<endl;*

*swap(arr,j,temp);*

*j=j-1;*

*temp=temp-1;*

*}*

*// cout<<"Main Loop"<<endl;*

*}*

*cout<<endl<<"Time Complexity is:"<<(((float)clock()-start)/CLOCKS\_PER\_SEC);*

*}*

*int main()*

*{*

*cout<<"Implement Array Sorting.............";*

*cout<<endl<<"Enter size of array(<100):";*

*int size;*

*cin>>size;*

*cout<<endl<<"Enter Array Elements............";*

*int arr[100];*

*for(int i=0;i<size;i++)*

*{*

*cout<<endl<<"Enter Element "<<i+1<<" is:";*

*cin>>arr[i];*

*}*

*cout<<endl<<"Press 1. To implement Bubble Sort.";*

*cout<<endl<<"Press 2. To implement Insertion Sort.";*

*cout<<endl<<"Press 3. To implement Selection Sort.";*

*cout<<endl<<"Enter Choice:";*

*int ch;*

*cin>>ch;*

*switch(ch)*

*{*

*case 1:bubble\_sort(arr,size);*

*break;*

*case 2:insertion\_sort(arr,size);*

*break;*

*case 3:selection\_sort(arr,size);*

*break;*

*default:cout<<endl<<"Entered choice is wrong.......... ";*

*exit(0);*

*}*

*cout<<endl<<"Final Array is:";*

*for(int i=0;i<size;i++)*

*{*

*cout<<endl<<"Element "<<i+1<<" is:"<<arr[i];*

*}*

*return 0;*

*}*

***Experiment-2(a)***

**Aim**: - *To implement Merge Sorting and find the time complexity*

**Source Code:-**

*#include <iostream>*

*using namespace std;*

*void merge(int arr[], int l, int m, int r)*

*{*

*int i, j, k;*

*int n1 = m - l + 1;*

*int n2 = r - m;*

*int L[n1], R[n2];*

*for (i = 0; i < n1; i++)*

*L[i] = arr[l + i];*

*for (j = 0; j < n2; j++)*

*R[j] = arr[m + 1+ j];*

*i = 0;*

*j = 0;*

*k = l;*

*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 l, int r)*

*{*

*if (l < r)*

*{*

*int m = l+(r-l)/2;*

*mergeSort(arr, l, m);*

*mergeSort(arr, m+1, r);*

*merge(arr, l, m, r);*

*}*

*}*

*void printArray(int A[], int size)*

*{*

*int i;*

*for (i=0; i < size; i++)*

*printf("%d ", A[i]);*

*}*

*int main()*

*{*

*cout<<"Implement Merge Sorting.............";*

*cout<<endl<<"Enter size of array(<100):";*

*int size;*

*cin>>size;*

*cout<<endl<<"Enter Array Elements............";*

*int arr[100];*

*for(int i=0;i<size;i++)*

*{*

*cout<<endl<<"Enter Element "<<i+1<<" is:";*

*cin>>arr[i];*

*}*

*float start=clock();*

*mergeSort(arr, 0, size - 1);*

*float end=clock();*

*cout<<endl<<("\nSorted array is: \n");*

*printArray(arr, size);*

*cout<<endl<<"Time Complexity is:"<<((end-start)/CLOCKS\_PER\_SEC);*

*return 0;*

*}*

***Experiment-2(b)***

**Aim**: - *To implement Quick Sort and find the time complexity*

**Source Code:-**

*#include <iostream>*

*using namespace std;*

*void swap(int\* a, int\* b)*

*{*

*int t = \*a;*

*\*a = \*b;*

*\*b = t;*

*}*

*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);*

*}*

*}*

*void printArray(int arr[], int size)*

*{*

*int i;*

*for (i=0; i < size; i++)*

*printf("%d ", arr[i]);*

*}*

*int main()*

*{*

*cout<<"Implement Quick Sorting.............";*

*cout<<endl<<"Enter size of array(<100):";*

*int size;*

*cin>>size;*

*cout<<endl<<"Enter Array Elements............";*

*int arr[100];*

*for(int i=0;i<size;i++)*

*{*

*cout<<endl<<"Enter Element "<<i+1<<" is:";*

*cin>>arr[i];*

*}*

*float start=clock();*

*quickSort(arr, 0, size-1);*

*float end=clock();*

*cout<<endl<<("\nSorted array is: \n");*

*printArray(arr, size);*

*cout<<endl<<"Time Complexity is:"<<((end-start)/CLOCKS\_PER\_SEC);*

*return 0;*

*}*

***EXPERIMENT-3(a)***

**Aim: -** *Write a program to implement and find time complexity of Linear Search.*

**Source Code:-**

*#include <iostream>*

*#include <ctime>*

*using namespace std;*

*int linearSearch(int input[],int n,int num){*

*int i=0;*

*while(i<n){*

*if(input[i]==num) {*

*return i;*

*}*

*i++;*

*}*

*return -1;*

*}*

*int main()*

*{*

*cout<<"Implementing Linear Search.\n";*

*cout<<endl<<"Enter size of array(<100):";*

*int size;*

*cin>>size;*

*cout<<endl<<"Enter Array Elements............";*

*int arr[100];*

*for(int i=0;i<size;i++)*

*{*

*cout<<endl<<"Enter Element "<<i+1<<" is:";*

*cin>>arr[i];*

*}*

*int num\_to\_search;*

*cout<<endl<<"Enter any number to search:";*

*cin>>num\_to\_search;*

*float start=clock();*

*int result=linearSearch(arr,size,num\_to\_search);*

*float end=clock();*

*(result == -1)? printf("Element is not present in array")*

*: printf("Element is present at index %d",*

*result);*

*cout<<endl<<"Time Complexity is:"<<((end-start)/CLOCKS\_PER\_SEC);*

*return 0;*

*}*

***EXPERIMENT-3(b)***

**Aim: -** *Write a program to implement* *and find time complexity of Binary Search.*

**Source Code:-**

*#include <iostream>*

*#include <ctime>*

*using namespace std;*

*int binarySearch(int arr[], int l, int r, int x)*

*{*

*if (r >= l)*

*{*

*int mid = l + (r - l)/2;*

*if (arr[mid] == x)*

*return mid;*

*if (arr[mid] > x)*

*return binarySearch(arr, l, mid-1, x);*

*return binarySearch(arr, mid+1, r, x);*

*}*

*return -1;*

*}*

*int main()*

*{*

*cout<<"Implement Binary Search.............";*

*cout<<endl<<"Enter size of array(<100):";*

*int size;*

*cin>>size;*

*cout<<endl<<"Enter Array Elements............";*

*int arr[100];*

*for(int i=0;i<size;i++)*

*{*

*cout<<endl<<"Enter Element "<<i+1<<" is:";*

*cin>>arr[i];*

*}*

*int num\_to\_search;*

*cout<<endl<<"Enter any number to search:";*

*cin>>num\_to\_search;*

*float start=clock();*

*int result = binarySearch(arr, 0, size-1, num\_to\_search);*

*float end=clock();*

*(result == -1)? printf("Element is not present in array")*

*: printf("Element is present at index %d",*

*result);*

*cout<<endl<<"Time Complexity is:"<<((end-start)/CLOCKS\_PER\_SEC);*

*return 0;*

*}*

***EXPERIMENT-4***

**Aim: -** *Write a program to implement Matrix Chain Multiplication and analyze its time complexity*

**Source Code:-**

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

*#include <ctime>*

*using namespace std;*

*// Function for printing the optimal*

*// parenthesization of a matrix chain product*

*void printParenthesis(int i, int j, int n,*

*int \*bracket, char &name)*

*{*

*// If only one matrix left in current segment*

*if (i == j)*

*{*

*cout << name++;*

*return;*

*}*

*cout << "(";*

*// Recursively put brackets around subexpression*

*// from i to bracket[i][j].*

*// Note that "\*((bracket+i\*n)+j)" is similar to*

*// bracket[i][j]*

*printParenthesis(i, \*((bracket+i\*n)+j), n,*

*bracket, name);*

*// Recursively put brackets around subexpression*

*// from bracket[i][j] + 1 to j.*

*printParenthesis(\*((bracket+i\*n)+j) + 1, j,*

*n, bracket, name);*

*cout << ")";*

*}*

*// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n*

*// Please refer below article for details of this*

*// function*

*// https://goo.gl/k6EYKj*

*void matrixChainOrder(int p[], int n)*

*{*

*/\* For simplicity of the program, one extra*

*row and one extra column are allocated in*

*m[][]. 0th row and 0th column of m[][]*

*are not used \*/*

*int m[n][n];*

*// bracket[i][j] stores optimal break point in*

*// subexpression from i to j.*

*int bracket[n][n];*

*/\* m[i,j] = Minimum number of scalar multiplications*

*needed to compute the matrix A[i]A[i+1]...A[j] =*

*A[i..j] where dimension of A[i] is p[i-1] x p[i] \*/*

*// cost is zero when multiplying one matrix.*

*for (int i=1; i<n; i++)*

*m[i][i] = 0;*

*// L is chain length.*

*for (int L=2; L<n; L++)*

*{*

*for (int i=1; i<n-L+1; i++)*

*{*

*int j = i+L-1;*

*m[i][j] = INT\_MAX;*

*for (int k=i; k<=j-1; k++)*

*{*

*// q = cost/scalar multiplications*

*int q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];*

*if (q < m[i][j])*

*{*

*m[i][j] = q;*

*// Each entry bracket[i,j]=k shows*

*// where to split the product arr*

*// i,i+1....j for the minimum cost.*

*bracket[i][j] = k;*

*}*

*}*

*}*

*}*

*// The first matrix is printed as 'A', next as 'B',*

*// and so on*

*char name = 'A';*

*cout << "Optimal Parenthesization is : ";*

*printParenthesis(1, n-1, n, (int \*)bracket, name);*

*cout << endl<<"Optimal Cost is : " << m[1][n-1];*

*}*

*// Driver code*

*int main()*

*{*

*int arr[] = {40, 20, 30, 10, 30};*

*int n = sizeof(arr)/sizeof(arr[0]);*

*float start=clock();*

*matrixChainOrder(arr, n);*

*float end=clock();*

*cout<<endl<<"Time Complexity is:"<<((end-start)/CLOCKS\_PER\_SEC);*

*return 0;*

*}*

***EXPERIMENT-5***

**Aim: -** *Write a program to implement Longest Common Subsequence problem and analyze its complexity*

**Source Code:-**

*#include<iostream>*

*#include<cstring>*

*#include<cstdlib>*

*# include<ctime>*

*using namespace std;*

*/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/*

*void lcs( char \*X, char \*Y, int m, int n )*

*{*

*int L[m+1][n+1];*

*/\* Following steps build L[m+1][n+1] in bottom up fashion. Note*

*that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/*

*for (int i=0; i<=m; i++)*

*{*

*for (int j=0; j<=n; j++)*

*{*

*if (i == 0 || j == 0)*

*L[i][j] = 0;*

*else if (X[i-1] == Y[j-1])*

*L[i][j] = L[i-1][j-1] + 1;*

*else*

*L[i][j] = max(L[i-1][j], L[i][j-1]);*

*}*

*}*

*// Following code is used to print LCS*

*int index = L[m][n];*

*// Create a character array to store the lcs string*

*char lcs[index+1];*

*lcs[index] = ' '; // Set the terminating character*

*// Start from the right-most-bottom-most corner and*

*// one by one store characters in lcs[]*

*int i = m, j = n;*

*while (i > 0 && j > 0)*

*{*

*// If current character in X[] and Y are same, then*

*// current character is part of LCS*

*if (X[i-1] == Y[j-1])*

*{*

*lcs[index-1] = X[i-1]; // Put current character in result*

*i--; j--; index--; // reduce values of i, j and index*

*}*

*// If not same, then find the larger of two and*

*// go in the direction of larger value*

*else if (L[i-1][j] > L[i][j-1])*

*i--;*

*else*

*j--;*

*}*

*// Print the lcs*

*cout << "LCS of " << X << " and " << Y << " is " << lcs;*

*}*

*int main()*

*{*

*char X[] = "AGGTAB";*

*char Y[] = "GXTXAYB";*

*int m = strlen(X);*

*int n = strlen(Y);*

*float start=clock();*

*lcs(X, Y, m, n);*

*float end=clock();*

*cout<<endl<<"Time Complexity is:"<<((end-start)/CLOCKS\_PER\_SEC);*

*return 0;*

*}*

***EXPERIMENT-6***

**Aim: -** *Write a program to implement Optimal Binary Search Tree problem and analyze its complexity*

**Source Code:-**

*// A naive recursive implementation of optimal binary*

*// search tree problem*

*#include <stdio.h>*

*#include <limits.h>*

*# include<time.h>*

*// A utility function to get sum of array elements*

*// freq[i] to freq[j]*

*int sum(int freq[], int i, int j);*

*// A recursive function to calculate cost of optimal*

*// binary search tree*

*int optCost(int freq[], int i, int j)*

*{*

*// Base cases*

*if (j < i) // no elements in this subarray*

*return 0;*

*if (j == i) // one element in this subarray*

*return freq[i];*

*// Get sum of freq[i], freq[i+1], ... freq[j]*

*int fsum = sum(freq, i, j);*

*// Initialize minimum value*

*int min = INT\_MAX;*

*// One by one consider all elements as root and*

*// recursively find cost of the BST, compare the*

*// cost with min and update min if needed*

*for (int r = i; r <= j; ++r)*

*{*

*int cost = optCost(freq, i, r-1) +*

*optCost(freq, r+1, j);*

*if (cost < min)*

*min = cost;*

*}*

*// Return minimum value*

*return min + fsum;*

*}*

*// The main function that calculates minimum cost of*

*// a Binary Search Tree. It mainly uses optCost() to*

*// find the optimal cost.*

*int optimalSearchTree(int keys[], int freq[], int n)*

*{*

*// Here array keys[] is assumed to be sorted in*

*// increasing order. If keys[] is not sorted, then*

*// add code to sort keys, and rearrange freq[]*

*// accordingly.*

*return optCost(freq, 0, n-1);*

*}*

*// A utility function to get sum of array elements*

*// freq[i] to freq[j]*

*int sum(int freq[], int i, int j)*

*{*

*int s = 0;*

*for (int k = i; k <=j; k++)*

*s += freq[k];*

*return s;*

*}*

*// Driver program to test above functions*

*int main()*

*{*

*int keys[] = {10, 12, 20};*

*int freq[] = {34, 8, 50};*

*int n = sizeof(keys)/sizeof(keys[0]);*

*float start=clock();*

*printf("Cost of Optimal BST is %d ",*

*optimalSearchTree(keys, freq, n));*

*float end=clock();*

*printf("\n Time Complexity is:");*

*printf(((end-start)/CLOCKS\_PER\_SEC));*

*return 0;*

*}*

***EXPERIMENT-7***

**Aim: -** *Write a program to implement Huffman Coding and analyze its complexity*

**Source Code:-**

*// C program for Huffman Coding*

*#include <stdio.h>*

*#include <stdlib.h>*

*// This constant can be avoided by explicitly*

*// calculating height of Huffman Tree*

*#define MAX\_TREE\_HT 100*

*// A Huffman tree node*

*struct MinHeapNode {*

*// One of the input characters*

*char data;*

*// Frequency of the character*

*unsigned freq;*

*// Left and right child of this node*

*struct MinHeapNode \*left, \*right;*

*};*

*// A Min Heap: Collection of*

*// min heap (or Hufmman tree) nodes*

*struct MinHeap {*

*// Current size of min heap*

*unsigned size;*

*// capacity of min heap*

*unsigned capacity;*

*// Attay of minheap node pointers*

*struct MinHeapNode\*\* array;*

*};*

*// A utility function allocate a new*

*// min heap node with given character*

*// and frequency of the character*

*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;*

*}*

*// A utility function to create*

*// a min heap of given capacity*

*struct MinHeap\* createMinHeap(unsigned capacity)*

*{*

*struct MinHeap\* minHeap*

*= (struct MinHeap\*)malloc(sizeof(struct MinHeap));*

*// current size is 0*

*minHeap->size = 0;*

*minHeap->capacity = capacity;*

*minHeap->array*

*= (struct MinHeapNode\*\*)malloc(minHeap->*

*capacity \* sizeof(struct MinHeapNode\*));*

*return minHeap;*

*}*

*// A utility function to*

*// swap two min heap nodes*

*void swapMinHeapNode(struct MinHeapNode\*\* a,*

*struct MinHeapNode\*\* b)*

*{*

*struct MinHeapNode\* t = \*a;*

*\*a = \*b;*

*\*b = t;*

*}*

*// The standard minHeapify function.*

*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);*

*}*

*}*

*// A utility function to check*

*// if size of heap is 1 or not*

*int isSizeOne(struct MinHeap\* minHeap)*

*{*

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

*}*

*// A standard function to extract*

*// minimum value node from heap*

*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;*

*}*

*// A utility function to insert*

*// a new node to Min Heap*

*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;*

*}*

*// A standard funvtion to build min heap*

*void buildMinHeap(struct MinHeap\* minHeap)*

*{*

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

*int i;*

*for (i = (n - 1) / 2; i >= 0; --i)*

*minHeapify(minHeap, i);*

*}*

*// A utility function to print an array of size n*

*void printArr(int arr[], int n)*

*{*

*int i;*

*for (i = 0; i < n; ++i)*

*printf("%d", arr[i]);*

*printf("\n");*

*}*

*// Utility function to check if this node is leaf*

*int isLeaf(struct MinHeapNode\* root)*

*{*

*return !(root->left) && !(root->right);*

*}*

*// Creates a min heap of capacity*

*// equal to size and inserts all character of*

*// data[] in min heap. Initially size of*

*// min heap is equal to capacity*

*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;*

*}*

*// The main function that builds Huffman tree*

*struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size)*

*{*

*struct MinHeapNode \*left, \*right, \*top;*

*// Step 1: Create a min heap of capacity*

*// equal to size. Initially, there are*

*// modes equal to size.*

*struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);*

*// Iterate while size of heap doesn't become 1*

*while (!isSizeOne(minHeap)) {*

*// Step 2: Extract the two minimum*

*// freq items from min heap*

*left = extractMin(minHeap);*

*right = extractMin(minHeap);*

*// Step 3: Create a new internal*

*// node with frequency equal to the*

*// sum of the two nodes frequencies.*

*// Make the two extracted node as*

*// left and right children of this new node.*

*// Add this node to the min heap*

*// '$' is a special value for internal nodes, not used*

*top = newNode('$', left->freq + right->freq);*

*top->left = left;*

*top->right = right;*

*insertMinHeap(minHeap, top);*

*}*

*// Step 4: The remaining node is the*

*// root node and the tree is complete.*

*return extractMin(minHeap);*

*}*

*// Prints huffman codes from the root of Huffman Tree.*

*// It uses arr[] to store codes*

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

*{*

*// Assign 0 to left edge and recur*

*if (root->left) {*

*arr[top] = 0;*

*printCodes(root->left, arr, top + 1);*

*}*

*// Assign 1 to right edge and recur*

*if (root->right) {*

*arr[top] = 1;*

*printCodes(root->right, arr, top + 1);*

*}*

*// If this is a leaf node, then*

*// it contains one of the input*

*// characters, print the character*

*// and its code from arr[]*

*if (isLeaf(root)) {*

*printf("%c: ", root->data);*

*printArr(arr, top);*

*}*

*}*

*// The main function that builds a*

*// Huffman Tree and print codes by traversing*

*// the built Huffman Tree*

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

*{*

*// Construct Huffman Tree*

*struct MinHeapNode\* root*

*= buildHuffmanTree(data, freq, size);*

*// Print Huffman codes using*

*// the Huffman tree built above*

*int arr[MAX\_TREE\_HT], top = 0;*

*printCodes(root, arr, top);*

*}*

*// Driver program to test above functions*

*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;*

*}*

***EXPERIMENT-8***

**Aim: -** *Write a program to implement Dijkstra’s Algorithm and analyze its complexity*

**Source Code:-**

*// A C++ program for Dijkstra's single source shortest path algorithm.*

*// The program is for adjacency matrix representation of the graph*

*#include <stdio.h>*

*#include <limits.h>*

*// Number of vertices in the graph*

*#define V 9*

*// A utility function to find the vertex with minimum distance value, from*

*// the set of vertices not yet included in shortest path tree*

*int minDistance(int dist[], bool sptSet[])*

*{*

*// Initialize min value*

*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;*

*}*

*// A utility function to print the constructed distance array*

*int printSolution(int dist[], int n)*

*{*

*printf("Vertex   Distance from Source\n");*

*for (int i = 0; i < V; i++)*

*printf("%d tt %d\n", i, dist[i]);*

*}*

*// Function that implements Dijkstra's single source shortest path algorithm*

*// for a graph represented using adjacency matrix representation*

*void dijkstra(int graph[V][V], int src)*

*{*

*int dist[V];     // The output array.  dist[i] will hold the shortest*

*// distance from src to i*

*bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest*

*// path tree or shortest distance from src to i is finalized*

*// Initialize all distances as INFINITE and stpSet[] as false*

*for (int i = 0; i < V; i++)*

*dist[i] = INT\_MAX, sptSet[i] = false;*

*// Distance of source vertex from itself is always 0*

*dist[src] = 0;*

*// Find shortest path for all vertices*

*for (int count = 0; count < V-1; count++)*

*{*

*// Pick the minimum distance vertex from the set of vertices not*

*// yet processed. u is always equal to src in the first iteration.*

*int u = minDistance(dist, sptSet);*

*// Mark the picked vertex as processed*

*sptSet[u] = true;*

*// Update dist value of the adjacent vertices of the picked vertex.*

*for (int v = 0; v < V; v++)*

*// Update dist[v] only if is not in sptSet, there is an edge from*

*// u to v, and total weight of path from src to  v through u is*

*// smaller than current value of dist[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];*

*}*

*// print the constructed distance array*

*printSolution(dist, V);*

*}*

*// driver program to test above function*

*int main()*

*{*

*/\* Let us create the example graph discussed above \*/*

*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;*

*}*

***EXPERIMENT-9***

**Aim: -** *Write a program to implement Bellman Ford Algorithm and analyze its complexity*

**Source Code:-**

*// A C++ program for Bellman-Ford's single source*

*// shortest path algorithm.*

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

*// a structure to represent a weighted edge in graph*

*struct Edge*

*{*

*int src, dest, weight;*

*};*

*// a structure to represent a connected, directed and*

*// weighted graph*

*struct Graph*

*{*

*// V-> Number of vertices, E-> Number of edges*

*int V, E;*

*// graph is represented as an array of edges.*

*struct Edge\* edge;*

*};*

*// Creates a graph with V vertices and E edges*

*struct Graph\* createGraph(int V, int E)*

*{*

*struct Graph\* graph = new Graph;*

*graph->V = V;*

*graph->E = E;*

*graph->edge = new Edge[E];*

*return graph;*

*}*

*// A utility function used to print the solution*

*void printArr(int dist[], int n)*

*{*

*printf("Vertex   Distance from Source\n");*

*for (int i = 0; i < n; ++i)*

*printf("%d \t\t %d\n", i, dist[i]);*

*}*

*// The main function that finds shortest distances from src to*

*// all other vertices using Bellman-Ford algorithm.  The function*

*// also detects negative weight cycle*

*void BellmanFord(struct Graph\* graph, int src)*

*{*

*int V = graph->V;*

*int E = graph->E;*

*int dist[V];*

*// Step 1: Initialize distances from src to all other vertices*

*// as INFINITE*

*for (int i = 0; i < V; i++)*

*dist[i]   = INT\_MAX;*

*dist[src] = 0;*

*// Step 2: Relax all edges |V| - 1 times. A simple shortest*

*// path from src to any other vertex can have at-most |V| - 1*

*// edges*

*for (int i = 1; i <= V-1; i++)*

*{*

*for (int j = 0; j < E; j++)*

*{*

*int u = graph->edge[j].src;*

*int v = graph->edge[j].dest;*

*int weight = graph->edge[j].weight;*

*if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])*

*dist[v] = dist[u] + weight;*

*}*

*}*

*// Step 3: check for negative-weight cycles.  The above step*

*// guarantees shortest distances if graph doesn't contain*

*// negative weight cycle.  If we get a shorter path, then there*

*// is a cycle.*

*for (int i = 0; i < E; i++)*

*{*

*int u = graph->edge[i].src;*

*int v = graph->edge[i].dest;*

*int weight = graph->edge[i].weight;*

*if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])*

*printf("Graph contains negative weight cycle");*

*}*

*printArr(dist, V);*

*return;*

*}*

*// Driver program to test above functions*

*int main()*

*{*

*/\* Let us create the graph given in above example \*/*

*int V = 5;  // Number of vertices in graph*

*int E = 8;  // Number of edges in graph*

*struct Graph\* graph = createGraph(V, E);*

*// add edge 0-1 (or A-B in above figure)*

*graph->edge[0].src = 0;*

*graph->edge[0].dest = 1;*

*graph->edge[0].weight = -1;*

*// add edge 0-2 (or A-C in above figure)*

*graph->edge[1].src = 0;*

*graph->edge[1].dest = 2;*

*graph->edge[1].weight = 4;*

*// add edge 1-2 (or B-C in above figure)*

*graph->edge[2].src = 1;*

*graph->edge[2].dest = 2;*

*graph->edge[2].weight = 3;*

*// add edge 1-3 (or B-D in above figure)*

*graph->edge[3].src = 1;*

*graph->edge[3].dest = 3;*

*graph->edge[3].weight = 2;*

*// add edge 1-4 (or A-E in above figure)*

*graph->edge[4].src = 1;*

*graph->edge[4].dest = 4;*

*graph->edge[4].weight = 2;*

*// add edge 3-2 (or D-C in above figure)*

*graph->edge[5].src = 3;*

*graph->edge[5].dest = 2;*

*graph->edge[5].weight = 5;*

*// add edge 3-1 (or D-B in above figure)*

*graph->edge[6].src = 3;*

*graph->edge[6].dest = 1;*

*graph->edge[6].weight = 1;*

*// add edge 4-3 (or E-D in above figure)*

*graph->edge[7].src = 4;*

*graph->edge[7].dest = 3;*

*graph->edge[7].weight = -3;*

*BellmanFord(graph, 0);*

*return 0;*

*}*

***EXPERIMENT-10***

**Aim: -** *Write a program to implement Bellman Ford Algorithm and analyze its complexity*

**Source Code:-**

*#include<iostream.h>*

*#include<conio.h>*

*#include <string.h>*

*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<<"\n";*

*}*

*}*

*int main()*

*{ clrscr();*

*char txt[] = "AABAACAADAABAAABAA";*

*char pat[] = "AABAAA"; search(pat, txt);*

*getch(); return 0;*

*}*