1. Write a C Program to Sort a given set of elements using Selection sort and determine the time required to sort elements.

**ALGORITHM**

**Selection Sort**

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

**PROGRAM CODE:**

**//SELECTION SORT**

#include<stdio.h>

#include<time.h>

int selection\_sort(int a[], int n);

int main()

{

int i,n,a[20],key;

clock\_tend,start;

printf("Enter the size of an array\n");

scanf("%d",&n);

printf("Enter the array elements\n");

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

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

start = clock();

selection\_sort(a,n);

end = clock();

printf("Sorted elements are\n");

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

printf("\n %d",a[i]);

printf("\nTime = %f",(double)(end - start)/CLOCKS\_PER\_SEC);

}

int selection\_sort(int a[], int n)

{

int i,j,pos,small,temp;

for(i=0;i<n-1;i++)

{

small = a[i];

pos = i;

for (j=i+1;j<n;j++)

{

if (a[j] < small)

{

small = a[j];

pos=j;

}

}

temp = a[pos];

a[pos] = a[i];

a[i] = temp;

}

}

/\*

**OUTPUT**

Enter the size of an array

7

Enter the array elements

12

67

23

0

1

45

78

Sorted elements are

0

1

12

23

45

67

78

Time = 0.000012

\*/

1. Write a C Program to Check whether a given graph is connected or not using DFS method.

**ALGORITHM**

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

**PROGRAM CODE:**

**//DFS METHOD**

#include<stdio.h>

int a[20][20],reach[20],n;

void dfs(int v)

{

int i;

reach[v]=1;

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

if(a[v][i] && !reach[i])

{

printf("\n %d --> %d",v,i);

dfs(i);

}

}

void main()

{

int i,j,count=0;

printf("Enter the number of Vertices: ");

scanf("%d",&n);

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

{

reach[i]=0;

for(j=1;j<=n;j++)

a[i][j]=0;

}

printf("Enter the Adacency matrix\n");

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

for(j=1;j<=n;j++)

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

dfs(1);

printf("\n");

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

{

if (reach[i])

count++;

}

if (count == n)

printf("\n Graph is connected : ");

else

printf("\n Graph is not connected : ");

}

/\***OUTPUT**

RUN 1:

Enter the number of Vertices: 4

Enter the Adacency matrix

0 1 1 1

0 0 0 1

0 0 0 0

0 0 1 0

1 --> 2

2 --> 4

4 --> 3

Graph is connected

RUN 2:

Enter the number of Vertices: 4

Enter the Adacency matrix

0 1 0 0

1 0 0 0

0 0 0 1

0 0 1 0

1 --> 2

Graph is not connected

\*/

1. Write a C Program to Print all the nodes reachable from a given starting node in a digraph using BFS method.

**ALGORITHM**

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

**PROGRAM CODE:**

**//BFS METHOD**

#include<stdio.h>

int i,j,n,r=-1,f,q[20],visited[20];

int a[20][20];

int bfs(int v);

void main()

{

int v;

printf("Enter the number of vertices....\n");

scanf("%d",&n);

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

{

q[i]=0;

visited[i]=0;

}

printf("Enter the adjacency matrix..\n");

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

for(j=1;j<=n;j++)

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

printf("Enter the starting vertex...\n");

scanf("%d",&v);

bfs(v);

printf("the nodes that are rachable from given node %d are..\n",v);

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

if (visited[i])

printf("%d ",i);

}

int bfs(int v)

{

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

if(a[v][i] && !visited[i])

q[++r]=i;

visited[i]=1;

if (f <= r)

{

visited[q[f]]=1;

bfs(q[f++]);

}

}

/\***OUTPUT**

Enter the number of vertices....

4

Enter the adjacency matrix..

0 1 1 1

0 0 0 1

0 0 0 0

0 0 1 0

Enter the starting vertex...

1

the nodes that are rachable from given node 1 are..

2 3 4 \*/

1. Write a C Program to sort a given set of elements using Merge sort method and determine the time required to sort the elements.

**ALGORITHM**

**Merge sort**

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

**PROGRAM CODE:**

**//Merge Sort**

#include<stdio.h>

#include<time.h>

int mergesort(int \*, int , int);

int merge(int \*, int, int, int);

void main ()

{

int i,n,a[20];

clock\_tstart,end;

start = clock();

printf("Enter the limit \n");

scanf("%d",&n);

printf("Enter the elements \n");

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

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

mergesort(a,0,n-1);

end = clock();

printf("The sorted elements are\n");

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

printf("%d\n",a[i]);

printf("\n Time = %f",(double)(end - start)/CLOCKS\_PER\_SEC);

return;

}

int mergesort (int a[], int low, int high)

{

int mid;

if (low < high)

{

mid = (low + high)/2;

mergesort(a,low,mid);

mergesort(a,mid+1,high);

merge(a,low,mid,high);

}

}

int merge(int a[], int low,int mid, int high)

{

int i,j,k,h,b[20];

h=i=low;

j=mid + 1;

while (h <= mid && j<=high)

if (a[h] < a[j])

b[i++] = a[h++];

else

b[i++]=a[j++];

if (h > mid)

for(k=j;k<=high;k++)

b[i++] = a[k];

else

for (k=h;k<=mid;k++)

b[i++] = a[k];

for(k=low;k<=high;k++)

a[k] = b[k];

}

/\*

**OUTPUT**

RUN 1:

Enter the limit

7

Enter the elements

12

67

23

89

0

34

13

The sorted elements are

0

12

13

23

34

67

89

\*/

1. Write a C Program to Sort a given set of elements using Quick sort method and determine the time required sort the elements

**ALGORITHM**

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivotStep 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

**PROGRAM CODE:**

**//QUICK SORT**

#include<stdio.h>

#include<time.h>

int partition(int a[],int low,int high);

void quicksort(int a[],int low,int high)

{

int j;

if(low < high)

{

j=partition(a,low,high);

quicksort(a,low,j-1);

quicksort(a,j+1,high);

}

}

int partition(int a[],int low,int high)

{

int i,j,temp,key;

key=a[low];

i=low+1;

j=high;

while (1)

{

while (i<high && key >= a[i])

i++;

while (key < a[j] )

j--;

if (i< j)

{

temp=a[i];

a[i]=a[j];

a[j]=temp;

}

else

{

temp = a[low];

a[low]=a[j];

a[j]=temp;

return j;

}

}

}

void main ()

{

int i,n,a[20];

float f;

clock\_tstart,end;

printf("Enter the No of Elements\n");

scanf("%d",&n);

printf("Enter the [%d] element : \n",n);

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

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

start=clock();

quicksort(a,0,n-1);

end=clock();

printf("\n The Sorted array is : \n");

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

printf("%d\t",a[i]);

printf("\n Time Taken = %f",(double)(end-start)/CLOCKS\_PER\_SEC);

}

1. Write a C Program to implement Recursive Binary search and linear search and determine the time required to search an element.

**ALGORITHM**

**Linear Search** ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i> n then go to step 7Step 3: if A[i] = x then go to step 6Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found!!

Step 8: Exit

**Binary\_search**

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound<lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

end procedure

**Linear search**

**PROGRAM CODE:**

#include<stdio.h>

#include<time.h>

#define max 20

int a[max],n,key;

void main()

{

int i,key,ch,mid,low,high,L;

clock\_t start1,end1,start2,end2;

printf("Enter the limit\n");

scanf("%d",&n);

printf("Enter the elements \n");

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

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

printf("\n LINEAR SEARCH \n");

start1=clock();

printf("Enter the key element to search\n");

scanf("%d",&key);

L=ls(1,key);

end1=clock();

if(L==-1)

printf("Element is not found \n");

else

printf("Element is found \n");

printf("Time=%f",((double)(end1-start1))/CLOCKS\_PER\_SEC);

printf("\n");

printf("\n BINARY SEARCH \n");

start2=clock();

printf("enter the key element to be searched\n");

scanf("%d",&key);

low=1;

high=n;

ch=bs(low,high,key);

end2=clock();

if(ch==-1)

printf("element is not found\n");

else

printf("element is found\n");

printf("Time=%f",((double)(end2-start2))/CLOCKS\_PER\_SEC);

}

int ls(int i, int key)

{

if(i>n)

return(-1);

if(a[i]==key)

return i;

else

ls(++i,key);

return;

}

int bs(int low,inthigh,int key)

{

int mid;

if(low>high)

return (-1);

mid=(low+high)/2;

if(a[mid]==key)

return mid;

else

{

if(key<a[mid])

bs(low,mid-1,key);

else

bs(mid+1,high,key);

}

}

/\***OUTPUT**:

enter the limit

3

enter the elements

10

20

30

LINEAR SEARCH

enter the key element to search

20

element is found

TIME =0.000078

BINARY SEAERCH

enter the key element to be searched

20

element is found

TIME = 0.000037 \*/

1. Write a C Program to Sort a given set of elements using Insertion sort and determine the time required to sort elements.

**ALGORITHM:**

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

**PROGRAM CODE:**

**//INSERTION SORT**

#include<stdio.h>

#include<time.h>

void main()

{

int a[10],v,j,n,i;

clock\_tstart,end;

printf("\n Enter the order of an array\n");

scanf("%d",&n);

start=clock();

printf("Enter the elements of an array\n");

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

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

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

{

v=a[i];

j=i-1;

while (j >= 00 && a[j] > v)

{

a[j+1]=a[j];

j=j-1;

}

a[j+1]=v;

}

end = clock();

printf("\n The Sorted array is \n");

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

printf("\n %d",a[i]);

printf("\nTime = %f",(double)(end-start)/CLOCKS\_PER\_SEC);

}

/\*

**OUTPUT**

Enter the order of an array

6

Enter the elements of an array

9 4 5 2 6 1

The Sorted array is

1

2

4

5

6

9

Time = 0.000035

\*/

1. Write a C Program to Sort a given set of elements using the Heap sort method and determine the time required to sort the elements.

**ALGORTIM:**

**Heap Sort**

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into **Min Heap.**

Step 3- Delete the root element from Min Heap using **Heapify** method.

Step 4 - Put the deleted element into the Sorted list.

Step 5 - Repeat the same until Min Heap becomes empty.

Step 6 - Display the sorted list.

**PROGRAM CODE:**

**//HEAP SORT**

#include<stdio.h>

int swap(int \*x, int \*y);

int heap(int n);

int heapsort (int n);

int a[10];

int main()

{

int i,n;

printf("Enter the limit\n");

scanf("%d",&n);

printf("Enter the elements \n");

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

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

heap(n);

heapsort(n);

printf("Sorted elements \n");

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

printf("%d\t",a[i]);

}

int heap(int n)

{

int ch, ps, temp;

for (ch=1;ch<=n;ch++)

{

temp=a[ch];

ps=ch/2;

while (ch>1 && temp > a[ps])

{

a[ch] = a[ps];

ch=ps;

ps=ch/2;

if (ps< 1)

ps=1;

}

a[ch] = temp;

}

}

int heapsort (int n)

{

while (n>1)

{

swap(&a[1],&a[n]);

{

n--;

heap(n);

}

}

}

int swap(int \*x, int \*y)

{

int temp;

temp=\*x;

\*x = \*y;

\*y = temp;

}

/\*

**OUTPUT**

Enter the limit

5

Enter the elements

2 4 3 7 6

Sorted elements

2 3 4 6 7

\*/

1. Write a C Program to Implement Horspool algorithm for String Matching.

**ALGORITHM**

Step 1- calculate the value of each letter of the substring to create the Bad Match Table, using this formula,

Value = length of substring – index of each letter in the substring – 1.

The value of the last letter and other letters that are not in the substring will be the length of the substring

Step 2- the value should be assigned to each letter in the Bad Match Table.

Step 3- compare the substring and the string. You start from the index of the end letter in the substring

A) If the letter matches, then compare with the preceding letter

B) If it doesn’t match, check its value in the Bad Match Table.

C) Then, skip the number of spaces that the table value indicates.

Step 4- Repeat this steps until all the letters match.

**PROGRAM CODE:**

**//HORSPOOL**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#define M 256

char pattern[M],text[M],table[M];

long m,n,i,j,k;

void shifttable()

{

int in;

for(i=0;i<M;i++)

table[i] = m;

for (j=0;j<m-1;j++)

{

in = (int)pattern[j] - '0';

table[in] = m-1-j;

}

}

int horsepool()

{

int index;

shifttable();

for (i=m-1;i<=n-1;)

{

k = 0;

while ((k <= m-1) && (pattern[m-1-k] == text[i-k]))

k++;

if (k == m)

return i-m+1;

else

{

index=(int)text[i] - '0';

i = i + table[index];

}

}

return -1;

}

int main ()

{

int found;

printf("Enter the text\n");

gets(text);

printf("\n Enter the Pattern ");

scanf("%s",pattern);

n = strlen(text);

m = strlen(pattern);

found = horsepool();

if (found == -1)

printf("\n Pattern not found!!!\n");

else

printf("\n Pattern found at position %d ",found+1);

}

1. Write a C Program to implement 0/1 Knapsack problem using dynamic programming.

**ALGORITHM**

Optimal Substructure:  
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.  
Therefore, the maximum value that can be obtained from n items is max of following two values.  
1) Maximum value obtained by n-1 items and W weight (excluding nth item).  
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

**PROGRAM CODE:**

**//KNAPSACK PROBLEM**

#include<stdio.h>

int n,capacity,w[50],p[50],maxprofit,i,j;

int MAX(int x, int y)

{

return (x>y)?x:y;

}

int sack (int i, int y)

{

if (i == n)

if (y < w[n])

return 0;

else

return p[n];

if (y < w[i]) return sack (i+1,y);

return MAX (sack(i+1,y),sack(i+1,y-w[i])+p[i]);

}

int main ()

{

printf("Enter the No of Objects\n");

scanf("%d",&n);

printf("Enter the weights\n");

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

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

printf("Enter the profit\n");

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

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

printf("Enter the capacity\n");

scanf("%d",&capacity);

maxprofit = sack(0,capacity);

printf("Maximum profit = %d ",maxprofit);

}

/\*

**OUTPUT**

Enter the No of Objects

4

Enter the weights

3 4 5 6

Enter the profit

10 20 30 40

Enter the capacity

10

Maximum profit = 60

\*/

1. Write a C Program to Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

**ALGORITHM**

Step 1- Sort all the edges in non-decreasing order of their weight.

Step 2- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

Step 3- Repeat step#2 until there are (V-1) edges in the spanning tree.

**PROGRAM CODE:**

**//Kruskal’s algorithm**

#include<stdio.h>

int parent[10],min,ne=1,mincost=0,cost[10][10];

int i,j,a,b,u,v,n;

int main()

{

printf("Enter the No. of Vettices of Graph\n");

scanf("%d",&n);

printf("Enter the cost Adjancey Of Matrix\n");

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

for(j=1;j<=n;j++)

{

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

if(cost[i][j] == 0)

cost[i][j]=999;

}

while (ne < n)

{

for(i=1,min=999;i<=n;i++)

for(j=1;j<=n;j++)

if(cost[i][j] < min)

{

min = cost[i][j];

a=u=i;

b=v=j;

}

while (parent[u])

u=parent[u];

while (parent[v])

v=parent[v];

if (u != v)

{

ne++;

printf("\n %d\t edge \t (%d,%d) = %d",ne,a,b,min);

mincost+=min;

parent[v] = u;

}

cost[a][b] = cost[b][a] = 999;

}

printf("\n mincost = %d \n",mincost);

}

/\*

**OUTPUT**

RUN 1:

Enter the No. of Vettices of Graph

4

Enter the cost Adjancey Of Matrix

0 5 11 4

5 0 6 13

11 6 0 7

4 13 7 0

2 edge (1,4) = 4

3 edge (1,2) = 5

4 edge (2,3) = 6

mincost = 15

RUN 2:

Enter the No. of Vettices of Graph

3

Enter the cost Adjancey Of Matrix

0 1 1

1 0 0

1 0 0

2 edge (1,2) = 1

3 edge (1,3) = 1

mincost = 2

\*/

1. Write a C Program to Find Minimum Cost Spanning Tree of a given undirected graph using Prim‟s algorithm.

**ALGORITHM**

Step 1- Create a set mstSet that keeps track of vertices already included in MST.  
Step 2- Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.  
Step 3- While mstSet doesn’t include all vertices  
 a) Pick a vertex u which is not there in mstSet and has minimum key value.

b) Include u to mstSet.  
 c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

**PROGRAM CODE**

**//PRIMS ALGORITHM**

#include<stdio.h>

int a,b,u,v,i,j,n,ne=1;

int visited [10],min,mincost=0,cost[10][10];

int main()

{

printf("Ente the No. of Vertices\n");

scanf("%d",&n);

printf("Enter the adjacent matrix\n");

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

for(j=1;j<=n;j++)

{

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

if (cost[i][j] == 0)

cost[i][j] = 999;

}

for (i=2;i<=n;i++)

visited[i] = 0;

printf("\n Edges of Spanning Tree ...\n");

visited[1]=1;

while (ne < n)

{

for(i=1,min=999;i<=n;i++)

for (j=1;j<=n;j++)

if (cost[i][j] < min)

if (visited[i] == 0)

continue;

else

{

min = cost[i][j];

a=u=i;

b=v=j;

}

if (visited[u] == 0 || visited[v] == 0)

{

ne++;

printf("\n %d Edge \t (%d, %d) = %d",ne,a,b,min);

mincost+=min;

visited[b] = 1;

}

cost[a][b] = cost [b][a] = 999;

}

printf("\n Minimum cost = %d",mincost);

}

/\* **OUTPUT**

RUN 1:

Ente the No. of Vertices

5

Enter the adjacent matrix

0 11 9 7 8

11 0 15 14 13

9 15 0 12 14

7 14 12 0 6

8 13 14 6 0

Edges of Spanning Tree ...

2 Edge (1, 4) = 7

3 Edge (4, 5) = 6

4 Edge (1, 3) = 9

5 Edge (1, 2) = 11

Minimum cost = 33

RUN 2:

Ente the No. of Vertices

3

Enter the adjacent matrix

0 1 2

1 0 3

2 3 0

Edges of Spanning Tree ...

2 Edge (1, 2) = 1

3 Edge (1, 3) = 2

Minimum cost = 3

\*/