ALGORITHMS LABORATORY- --CS3401

|  |  |
| --- | --- |
|  |  |

LIST OF EXPERIMENTS

Searching and Sorting Algorithms

1. Implement Linear Search. Determine the time required to search for an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.  
2. Implement recursive Binary Search. Determine the time required to search an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.  
3. Given a text txt [0...n-1] and a pattern pat [0...m-1], write a function search (char pat [ ], char txt [ ]) that prints all occurrences of pat [ ] in txt [ ]. You may assume that n > m.  
4. Sort a given set of elements using the Insertion sort and Heap sort methods and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

Graph Algorithms

1. Develop a program to implement graph traversal using Breadth First Search  
2. Develop a program to implement graph traversal using Depth First Search  
3. From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra’s algorithm.  
4. Find the minimum cost spanning tree of a given undirected graph using Prim’s algorithm.  
5. Implement Floyd’s algorithm for the All-Pairs- Shortest-Paths problem.  
6. Compute the transitive closure of a given directed graph using Warshall's algorithm.

Algorithm Design Techniques

1. Develop a program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.  
2. Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

State Space Search Algorithms

1. Implement N Queens problem using Backtracking.

Searching and Sorting Algorithms

# Implement Linear Search. Determine the time required to search for an element. Repeat the experiment for different values of *n*, the number of elements in the list to be searched and plot a graph of the time taken versus *n*.

Aim:

To Implement Linear Search and to determine the time required to search for an element, the number of elements in the list to be searched and plot a graph of the time taken versus *n*.

# Algorithm:

Step 1: First, read the search element (Target element) in the array.

Step 2: Set an integer i = 0 and repeat steps 3 to 4 till i reaches the end of the array.

Step 3: Match the key with arr[i].

Step 4: If the key matches, return the index. Otherwise, increment i by 1.

# Program:

#include<stdio.h> #include<conio.h> #include<time.h> #include<stdlib.h> #define max 20

int pos;

int linsearch(int,int[],int); void main()

{

int ch=1;double t;

int n,i,a[max],k,op,low,high,pos;clock\_t,begin,end; clrscr();

while (ch)

{

printf(“\n MENU \n 1. Linear Search \n 2. Exit \n”);

printf(“\n Enter Your choice\n”); scanf(“%d”,&op);

switch(op)

{

Case 1:printf(“\n Enter the Number of Elements \n”); scanf(“%d”,&n);

printf(“\n Enter the Elements of an Array\n”); for(i=0;i<n;i++)

scanf(“%d”,&a[i]);

printf(“\n Enter the Element to be Searched \n”);

scanf(“%d”,&k); begin=clock(); pos=linsearch(n,a,k); end=clock(); if(pos==-1)

printf(“\n\n Unsuccessful Search”); else

printf(“Element %d is found at position %d”,k,pos+1);

printf(“\n Time Taken is %f CPU Cycles \n”,(end-begin)/CLK\_TCK); getch();

break;

default:printf(“Invalid Choice Entered \n”); exit(0);

}

printf(“\nDo you wish to run again(1/0)\n”); scanf(“%d”,&ch);

}

getch();

}

int linsearch(int n, int a[],int k)

{

delay(1000); if(n<0) return -1; if(k==a[n-1])

return(n-1);

else

return linsearch(n-1,a,k);

}

# Output:

\*\*\*\*\*\*\*\*\*\*\*MENU\*\*\*\*\*\*\*\*\*\*\*\*

Linear Search

Exit

Enter Your Choice 1

Enter the number of elements 3

Enter the number of an array in the order 25 69 98

Enter the elements to be searched 98

Element 98 is found at position 3

Time Taken is 1.978022 CPU1 Cycles

# Implement recursive Binary Search. Determine the time required to search an element. Repeat the experiment for different values of *n*, the number of elements in the list to be searched and plot a graph of the time taken versus *n*.

Aim:

To implement recursive Binary Search and determine the time required to search an element.

# 

# Algorithm:

Step 1: Find the middle element of array. using middle = initial\_value + end\_value / 2 ;

Step 2: If middle = element, return ‘element found’ and index.

Step 3: if middle > element, call the function with end\_value = middle - 1 . Step 4: if middle < element, call the function with start\_value = middle + 1 . Step 5 : exit.

# Program:

#include <stdio.h> #include<conio.h> #include<time.h> #include<stdlib.h> #define max 20

int pos;

int binsearch(int,int[],int,int,int); void main()

{

int ch=1; double t;

int n,i,a[max],k,op,low,high,pos, clock\_t,begin,end;

clrscr(); while(ch)

{

printf("\n MENU \n l .BinarySearch \n 2.Exit \n”);

printf("\n enter your choice\n"); scanf("%d",&op);

switch(op)

{

case 1:printf("\n enter. the number of elements\n “); scanf(“%d" &n);

priritf("\n enter the elements in order \n"); for(i=0;i<n;i++)

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

printf("\n enter the element to be searched \n”); scanf("%d ",&k);

low=0;

high=n-1;

begin=clock(); pos=binsearch(n,a,k,low,high); end = clock();

if(pos==-1)

printf(“\n\n Unsuccessful Search”); else

printf("\n element %d is found at position %d” ,k, pos+1);

printf("\n Time Taken is %f CPU cycles \n”,(end-begin)/CLK\_TCK);

getch();

break;

}

printf(''\n Do you wish to run again(l/0) \n");

scanf("%d",&ch);

}

getch();

}

int binsearch(int n,int a[],int k,int low,int high)

{

int mid; delay(1000); mid=(low+high)/2; if(low>high)

return -1; if(k==a[mid]) return(mid);

else if(k<a[mid])

return binsearch(n,a,k,low,mid-1);

else

return binsearch(n,a,k,lmid+1,high);

}

# Output:

\*\*\*\*\*\*\*\*\*\*\*MENU\*\*\*\*\*\*\*\*\*\*\*\*

1.Binary Search

2.Exit

Enter Your Choice 1

Enter the number of elements

3

Enter the number of an array in the order 98

22

46

Enter the elements to be searched 22

Element 22 is found at position 2 Time Taken is 1.978022 CPU cycles.

# Given a text txt [0...n-1] and a pattern pat [0...m-1], write a function search (char pat [ ], char txt[ ]) that prints all occurrences of pat [ ] in txt [ ]. You may assume that n > m.

Aim:

To implement function search and to print all occurrences of the pattern in the given text.

# Algorithm:

NAVE\_STRING\_MATCHING (T, P)

for i←0 to n-m do

if P[1......m] == T[i+1. i+m] then

print "Match Found" end

# Program:

#include <stdio.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)

// if pat[0...M-1] = txt[i, i+1, ...i+M-1]

printf("Pattern found at index %d \n", i);

}

}

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 12

# Sort a given set of elements using the Insertion sort and Heap sort methods and determine the time required to sort the elements. Repeat the experiment for different values of *n*, the number of elements in the list to be sorted and plot a graph of the time taken versus *n*.

Aim:

To sort the elements using insertion sort and heap sort.

# Algorithm:

# Insertion Sort:

Step 1 : If the element is the first one, it is already sorted.

Step 2 : Move to next element.

Step 3 : Compare the current element with all elements in the sorted array.

Step 4 : If the

element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right

Step 5 : Insert the value at the correct position. Step 6 :Repeat until the complete list is sorted.

Program:

#include <stdio.h>

void insert(int a[], int n)

{

int i, j, temp;

for (i = 1; i < n; i++) { temp = a[i];

j = i - 1;

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

{

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

j = j-1;

}

a[j+1] = temp;

}

}

void printArr(int a[], int n)

{

int i;

for (i = 0; i < n; i++) printf("%d ", a[i]);

}

int main()

{

int a[20] ;

int n = sizeof(a) / sizeof(a[0]);

printf(“enter number of elements:”);

scanf(“%d”,&n);

printf(“enter the elements”);

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

scanf(“%d”,&a[i]);

printf("Before sorting array elements are - \n"); printArr(a, n);

insert(a, n);

printf("\nAfter sorting array elements are - \n"); printArr(a, n);

return 0;

}

# Output:

Before Sorting array elements are 12 31 25 8 32 17

After Sorting array elements are 8 12 17 25 31 32

# Heap Sort:

Algorithm:

Step 1 : Build a binary heap.

Step 2 : Start iterating from mid to the start of the binary heap array.

Step 3 : On every iteration, swap the root node with the last leaf node.

Step 4: Remove the leaf node by putting it back at the end of the new sorted array.

Step 5: Again do the heapify operation and repeat the iteration from step 2.

Step 6: Exit

#include<stdio.h>

*// function prototyping*

void heapify(int\*,int, int);

void heapsort(int\*, int);

void print\_array(int\*, int);

int main()

{

int arr[] ,CLK\_TCK,end,start;

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

printf("enter no of elements");

scanf("%d",&n);

printf("enter elements");

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

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

printf("\nArray before sorting:\n");

print\_array(arr, n);

 start=clock();

heapsort(arr, n);

end=clock();

printf("\nTime taken by Heapsort of CPU Cycle”,(end-start)/CLK\_TCK);

getch();

printf("\n\nArray after sorting:\n");

print\_array(arr, n);

return 0;

}

*/\* sorts the given array of n size \*/*

void heapsort(int\* arr, int n)

{

*// build the binary max heap*

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

{

heapify(arr, n, i);

}

*// sort the max heap*

for (int i = n - 1; i >= 0; i--)

{

*// swap the root node and the last leaf node*

int temp = arr[i];

arr[i] = arr[0];

arr[0] = temp;

*// again heapify the max heap from the root*

heapify(arr, i, 0);

}

}

*/\* heapify the subtree with root i \*/*

void heapify(int\* arr, int n, int i)

{

*// store largest as the root element*

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

*// now check whether the right and left right is larger than the root or not*

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

{

largest = left;

}

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

{

largest = right;

}

*// if the root is smaller than the children then swap it with the largest children's value*

if (largest != i)

{

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

*// again heapify that side of the heap where the root has gone*

heapify(arr, n, largest);

}

}

*/\* printf the array \*/*

void print\_array(int\* arr, int n)

{

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

{

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

}

}

# Output:

Enter the number of Elements:3 Enter the 3 Elements:

42

85

58

Sorted Elements are 42 58 85

Time taken by Heapsort 0.109890 CPU Cycle.

# 

# Graph Algorithms

* 1. Develop a program to implement graph traversal using Breadth First Search.

Aim:

To Develop a program to implement graph traversal using Breadth First Search.

# Algorithm:

1. First take the number of nodes in the graph as input and then create an adjacency matrix of size n x n where n is the number of nodes in the graph.
2. Next take the adjacency matrix as input.
3. Take the starting node as input.
4. We then call the bfs function with the starting node as the argument.
5. In the bfs function, we first mark the current node as visited and then we enqueue all the nodes which are adjacent to the current node and are not visited.
6. We then dequeue a node from the queue and call the bfs function with the dequeued node as the argument.
7. We repeat the above steps until the queue is empty.
8. Finally, we print the nodes which are reachable from the starting node.

# Program:

#include <stdio.h>

int n, i, j, visited[10], queue[10], front = -1, rear = -1;

int adj[10][10];

void bfs(int v)

{

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

if (adj[v][i] && !visited[i]) queue[++rear] = i;

if (front <= rear)

{

visited[queue[front]] = 1; bfs(queue[front++]);

}

}

void main()

{

int v;

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

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

{

queue[i] = 0;

visited[i] = 0;

}

printf("Enter graph data in matrix form:\n");

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

for (j = 1; j <= n; j++) scanf("%d", &adj[i][j]);

printf("Enter the starting vertex: "); scanf("%d", &v);

bfs(v);

printf("The node which are reachable are:\n");

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

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

else

printf("BFS is not possible. Not all nodes are reachable"); return 0;

}

# Output:

Enter the number of vertices: 4 Enter graph data in matrix form:

0 1 1 0

1 0 0 1

1 0 0 1

0 1 1 0

Enter the starting vertex: 2

The node which are reachable are:

1 2 3 4

# Develop a program to implement graph traversal using Depth First Search.

Aim:

To develop a program to implement graph traversal using Depth First Search.

# Algorithm:

Step 1: Create a stack with the total number of vertices in the graph as the size.

Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to the stack.

Step 3: Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.

Step 4: Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.

Step 5 : If there are no new vertices to visit, go back and pop one from the stack using backtracking.

Step 6 : Continue using steps 3, 4, and 5 until the stack is empty

Step 7 :When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

# Program:

#include<stdio.h>

# void DFS(int);

int G[10][10],visited[10],n;

void main()

{

int i,j;

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

printf("\nEnter adjecency matrix of the graph:"); for(i=0;i<n;i++)

for(j=0;j<n;j++) scanf("%d",&G[i][j]); for(i=0;i<n;i++) visited[i]=0;

DFS(0);

}

void DFS(int i)

{

int j;

printf("\n%d",i);

visited[i]=1; for(j=0;j<n;j++)

if(!visited[j]&&G[i][j]==1) DFS(j);

}

# Output:

Enter number of vertices :8

Enter adjancency matrix of the graph : 0 1 1 1 1 0 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 0 1 0

1 0 0 0 0 0 1 0

0 1 1 0 0 0 0 1

0 0 0 1 1 0 0 1

0 0 0 0 0 1 1 0

0

1

5

2

7

6

3

4

Process returned 8 (0x8) execution time : 64.785 s

# From a given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using Dijkstra’s algorithm.

Aim:

To find a shortest path to other vertex in a weighted connected graph using Dijkstra’s algorithm.

# Algorithm:

Step 1 : Create a set shortPath to store vertices that come in the way of the shortest path tree.

Step 2 : Initialize all distance values as INFINITE and assign distance values as 0 for source vertex so that it is picked first.

Step 3 : Loop until all vertices of the graph are in the shortPath. Program:

#include <limits.h> #include <stdbool.h> #include <stdio.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[])

{

printf("Vertex \t\t Distance from Source\n"); for (int i = 0; i < V; i++)

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

}

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 |

* 1. Find the minimum cost spanning tree of a given undirected graph using Prim’s algorithm.

# Aim:

To Find the minimum cost spanning tree of a given undirected graph using Prim’s algorithm.

# Algorithm:

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.

Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).

Step 3: Find edges connecting any tree vertex with the fringe vertices.

Step 4: Find the minimum among these edges.

Step 5: Add the chosen edge to the MST if it does not form any cycle.

Step 6: Return the MST and exit.

# Program:

#include <limits.h> #include <stdbool.h> #include <stdio.h> #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;

}

int printMST(int parent[], int graph[V][V])

{

printf("Edge \tWeight\n"); for (int i = 1; i < V; i++)

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

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 |
|  | 0-2 | 6 |
|  | 1-3 | 5 |

# Implement Floyd’s algorithm for the All-Pairs- Shortest-Paths problem.

# Aim :

To implement Floyd’s algorithm for the All-Pairs- Shortest-Paths problem.

# Algorithm:

Step 1: Initialize the shortest paths between any 2 vertices with Infinity.

Step 2: Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.

Step 3: Minimize the shortest paths between any 2 pairs in the previous operation.

# Step 4:

For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be:

min(dist[i][k]+dist[k][j],dist[i][j]).

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j.

As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

# Program:

#include <bits/stdc++.h> using namespace std; #define V 4

void printSolution(int dist[][V]); void floydWarshall(int dist[][V])

{

int i, j, k;

for (k = 0; k < V; k++)

{

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

{

for (j = 0; j < V; j++)

{

if (dist[i][j] > (dist[i][k] + dist[k][j]) && (dist[k][j] != INF && dist[i][k] != INF)) dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

printSolution(dist);

}

void printSolution(int dist[][V])

{

cout << "The following matrix shows the shortest " "distances" " between every pair of vertices \n"; for (int i = 0; i < V; i++)

{

for (int j = 0; j < V; j++)

{

if (dist[i][j] == INF)

cout << "INF" << " ";

else

cout << dist[i][j] << " ";

}

cout << endl;

}

}

int main()

{

int graph[V][V] = { { 0, 5, INF, 10 }, { INF, 0, 3, INF }, { INF, INF, 0, 1 }, { INF, INF, INF, 0 } };

floydWarshall(graph); return 0;

}

# Output:

The following matrix shows the shortest distance between every pair of vertices.

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 5 | 8 | 9 |
| INF | 0 | 3 | 4 |
| INF | INF | 0 | 1 |
| INF | INF | INF | 0 |
|  |  |  |  |

# Compute the transitive closure of a given directed graph using Warshall's algorithm.

Aim :

To Compute the transitive closure of a given directed graph using Warshall's algorithm.

# Algorithm:

Step 1 :Create a matrix A0 of dimension n\*n where n is the number of vertices. ...

Step 2:Now, create a matrix A1 using matrix A0 . ...

Step 3 :Similarly, A2 is created using A1 . ...

Step 4:Similarly, A3 and A4 is also created. ...

Step 5: A4 gives the shortest path between each pair of vertices.

# Program:

# include <stdio.h> # include <conio.h>

int n,a[10][10],p[10][10];

void path()

{

int i,j,k; for(i=0;i<n;i++) for(j=0;j<n;j++) p[i][j]=a[i][j]; for(k=0;k<n;k++) for(i=0;i<n;i++) for(j=0;j<n;j++)

if(p[i][k]==1&&p[k][j]==1) p[i][j]=1;

}

void main()

{

int i,j; clrscr();

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

printf("\nEnter the adjacency matrix:\n"); for(i=0;i<n;i++)

for(j=0;j<n;j++) scanf("%d",&a[i][j]); path();

printf("\nThe path matrix is showm below\n"); for(i=0;i<n;i++)

{

for(j=0;j<n;j++) printf("%d ",p[i][j]); printf("\n");

}

getch();

}

# Output:

Enter the number of nodes:4 Enter the adjacency matrix:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |

The path matrix is shown below

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

# 

# Algorithm Design Techniques

* + 1. Develop a program to find out the maximum and minimum numbers in a given list of *n*

# Numbers using the divide and conquer technique.

# Aim:

To develop a C program to find out the maximum and minimum numbers in a given list of *n*

Numbers using the divide and conquer technique.

# Algorithm:

1. Set min and max to arr[0]
2. Start loop at 1, using index i

i.Check if arr[i] < min; if so, set min to arr[i]

ii.Check if arr[i] > max; if so, set max to arr[i]

1. At end of loop, respond with an array containing [ min, max ]
2. If the array is 1 element long, return [ arr[0], arr[0] ]
3. If the array is 2 elements long

i.If arr[0] > arr[1], return [ arr[1], arr[0] ]

ii.Otherwise, return arr

1. Split the array roughly into a left half and right half, calling minmax on each half
2. return [ min(left[0], right[0]), max(left[1], right[1] ]

Program: #include<stdio.h> #include<stdio.h> int max, min;

int a[100];

void maxmin(int i, int j)

{

int max1, min1, mid; if(i==j)

{

max = min = a[i];

}

else

{

if(i == j-1)

{

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

{

max = a[j];

min = a[i];

}

else

{

max = a[i];

min = a[j];

}

}

else

{

mid = (i+j)/2; maxmin(i, mid);

max1 = max; min1 = min; maxmin(mid+1, j); if(max <max1)

max = max1; if(min > min1) min = min1;

}

}

}

int main ()

{

int i, num;

printf ("\nEnter the total number of numbers : "); scanf ("%d",&num);

printf ("Enter the numbers : \n");

for (i=1;i<=num;i++) scanf ("%d",&a[i]);

max = a[0];

min = a[0]; maxmin(1, num);

printf ("Minimum element in an array : %d\n", min); printf ("Maximum element in an array : %d\n", max); return 0;

}

# Output:

Enter the total number of numbers :

72

88

82

85

65

Minimum element in an array : 72 Maximum element in an array : 82

# 2. Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of *n*, the number of elements inthe list to be sorted and plot a graph of the time taken versus *n*.

Aim:

To write a C program to Implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort.

# Algorithm:

Step 1: Find the middle index of the array. Middle = 1 + (last – first)/2

Step 2: Divide the array from the middle.

Step 3: Call merge sort for the first half of the array MergeSort(array, first, middle)

Step 4: Call merge sort for the second half of the array. MergeSort(array, middle+1, last)

Step 5: Merge the two sorted halves into a single sorted array.

# Merge Sort:

step 1: start

step 2: declare array and left, right, mid variable step 3: perform merge function.

if left > right return

mid= (left+right)/2 mergesort(array, left, mid)

mergesort(array, mid+1, right) merge(array, left, mid, right)

step 4: Stop

# Program:

#include <stdio.h>

#include<time.h>

/\* Function to merge the subarrays of a[] \*/

void merge(int a[], int beg, int mid, int end)

{

    int i, j, k;

    int n1 = mid - beg + 1;

    int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays

    /\* copy data to temp arrays \*/

    for (int i = 0; i < n1; i++)

    LeftArray[i] = a[beg + i];

    for (int j = 0; j < n2; j++)

    RightArray[j] = a[mid + 1 + j];

    i = 0; /\* initial index of first sub-array \*/

    j = 0; /\* initial index of second sub-array \*/

    k = beg;  /\* initial index of merged sub-array \*/

    while (i < n1 && j < n2)

    {

        if(LeftArray[i] <= RightArray[j])

        {

            a[k] = LeftArray[i];

            i++;

        }

        else

        {

            a[k] = RightArray[j];

            j++;

        }

        k++;

    }

    while (i<n1)

    {

        a[k] = LeftArray[i];

        i++;

        k++;

    }

    while (j<n2)

    {

        a[k] = RightArray[j];

        j++;

        k++;

    }

}

void mergeSort(int a[], int beg, int end)

{

   if (beg < end)

    {

        int mid = (beg + end) / 2;

        mergeSort(a, beg, mid);

        mergeSort(a, mid + 1, end);

        merge(a, beg, mid, end);

    }

}

+

/\* Function to print the array \*/

void printArray(int a[], int n)

{

    int i;

    for (i = 0; i < n; i++)

        printf("%d ", a[i]);

    printf("\n");

}

int main()

{

    int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 },clock\_t,end,begin;

    int n = sizeof(a) / sizeof(a[0]);

    printf("Before sorting array elements are - \n");

    printArray(a, n);

//begin=clock();

    mergeSort(a, 0, n - 1);

    printf("After sorting array elements are - \n");

    printArray(a, n);

//end=clock();

//printf(“time taken of %f cycles”,(end-begin)/CLK\_TK);

    return 0;

}

# Output:

Given array is 12 11 13 5 6 7

Sorted array is 5 6 7 11 12 13

# Quick Sort:

Algortihm:

Step 1: Consider the first element of the list as pivot (i.e., Element at first position in the list). Step 2 : Define two variables i and j. Set i and j to first and last elements of the list respectively. Step 3 : Increment i until list[i] > pivot then stop.

Step 4 : Decrement j until list[j] < pivot then stop.

Step 5 : If i < j then exchange list[i] and list[j].

Step 6 : Repeat steps 3,4 & 5 until i > j.

Step 7: Exchange the pivot element with list[j] element.

Program: #include<stdio.h> #include<conio.h>

void quickSort(int [10],int,int); void main(){

int list[20],size,i;

printf("Enter size of the list: "); scanf("%d",&size);

printf("Enter %d integer values: ",size); for(i = 0; i < size; i++) scanf("%d",&list[i]); quickSort(list,0,size-1);

printf("List after sorting is: "); for(i = 0; i < size; i++) printf(" %d",list[i]);

getch();

}

void quickSort(int list[10],int first,int last){ int pivot,i,j,temp;

if(first < last){ pivot = first; i = first;

j = last; while(i < j){

while(list[i] <= list[pivot] && i < last) i++;

while(list[j] > list[pivot]) j--;

if(i <j){

temp = list[i]; list[i] = list[j]; list[j] = temp;

}

}

temp = list[pivot]; list[pivot] = list[j]; list[j] = temp; quickSort(list,first,j-1); quickSort(list,j+1,last);

}

}

# Output:

Enter size of the list:8

Enter 8 integer values : 5 3 8 1 4 6 2 7

List after sorting is: 1 2 3 4 5 6 7 8

# State Space Search Algorithms

1. Implement N Queens problem using Backtracking. Aim:

To write a C program to Implement N Queens problem using Backtracking.

# Algorithm:

Step 1 : Place the queen row-wise, starting from the left-most cell.

Step 2 : If all queens are placed then return true and print the solution matrix.

Step 3 : Else try all columns in the current row.

Condition 1 : Check if the queen can be placed safely in this column then mark the current cell [Row, Column] in the solution matrix as 1 and try to check the rest of the problem recursively by placing the queen here leads to a solution or not.

Condition 2 : If placing the queen [Row, Column] can lead to the solution return true and print the solution for each queen's position.

Condition 3 : If placing the queen cannot lead to the solution then unmark this [row, column] in the solution matrix as 0, BACKTRACK, and go back to condition 1 to try other rows.

Step 4 : If all the rows have been tried and nothing worked, return false to trigger backtracking.

# Program:

#include <bits/stdc++.h> #define N 4

using namespace std;

void printSolution(int board[N][N])

{

for (int i = 0; i < N; i++) { for (int j = 0; j < N; j++)

if(board[i][j]) cout << "Q "; else cout<<". ";

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col)

{

int i, j;

for (i = 0; i < col; i++) if (board[row][i]) return false;

for (i = row, j = col; i >= 0 && j >= 0; i--, j--) if (board[i][j])

return false;

for (i = row, j = col; j >= 0 && i < N; i++, j--) if (board[i][j])

return false; return true;

}

bool solveNQUtil(int board[N][N], int col)

{

if (col >= N) return true;

for (int i = 0; i < N; i++) { if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQUtil(board, col + 1)) return true;

board[i][col] = 0; // BACKTRACK

}

}

return false;

}

bool solveNQ()

{

int board[N][N] = { { 0, 0, 0, 0 }, { 0, 0, 0, 0 }, { 0, 0, 0, 0 },{ 0, 0, 0, 0 } };

if (solveNQUtil(board, 0) == false)

{

cout << "Solution does not exist"; return false;

}

printSolution(board); return true;

}

int main()

{

solveNQ(); return 0;

}

# Output:

. . Q .

Q . . .

. . . Q

. Q . .

# Approximation Algorithms Randomized Algorithms

1. Implement any scheme to find the optimal solution for the Traveling Salesperson problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.

# Aim:

To find the solution using Traveling Salesperson Problem and determine the error in the approximation.

# Algorithm:

Step 1 :Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.

Step 2 :Generate all (n-1)! permutations of cities.

Step 3 : Calculate the cost of every permutation and keep track of the minimum cost permutation. Return the permutation with minimum cost.

# Program:

#include <bits/stdc++.h> using namespace std; #define V 4

int travllingSalesmanProblem(int graph[][V], int s)

{

vector<int> vertex;

for (int i = 0; i < V; i++) if (i != s)

vertex.push\_back(i); int min\_path = INT\_MAX; do {

int current\_pathweight = 0; int k = s;

for (int i = 0; i < vertex.size(); i++) { current\_pathweight += graph[k][vertex[i]]; k = vertex[i];

}

current\_pathweight += graph[k][s];

min\_path = min(min\_path, current\_pathweight);

} while (

next\_permutation(vertex.begin(), vertex.end())); return min\_path;

}

int main()

{

// matrix representation of graph int graph[][V] = { { 0, 10, 15, 20 },

{ 10, 0, 35, 25 },

{ 15, 35, 0, 30 },

{ 20, 25, 30, 0 } };

int s = 0;

cout << travllingSalesmanProblem(graph, s) << endl; return 0;

}

# Output:

80

2.Implement randomized algorithms for finding the kth smallest number.

The programs can be implemented in C/C++/JAVA/ Python.

Aim:

To Implement randomized algorithms for finding the kth smallest number.

Algorithm:

Step 1 :Partition the array A[left .. right] into two subarrays A[left .. pos] and A[pos + 1 .. right] such that each element of A[left .. pos] is less than each element of A[pos + 1 .. right].

Step 2: Computes the number of elements in the subarray A[left .. pos] i.e. count = pos - left + 1 if (count == K), then A[pos] is the Kth smallest element.

Step 3: Otherwise determines in which of the two subarrays A[left .. pos-1] and A[pos + 1 .. right] the Kth smallest element lies.

Step 4: If (count > K) then the desired element lies on the left side of the partition.

Step 5: If (count < K), then the desired element lies on the right side of the partition. Since we already know i values that are smaller than the kth smallest element of A[left .. right], the desired element is the (K - count)th smallest element of A[pos + 1 .. right].

Step 6: Base case is the scenario of single element array i.e left ==right. return A[left] or A[right].

Program:

#include<stdio.h>

#include<conio.h>

int cmpfunc(const void\* a, const void\* b)

{

return (\*(int\*)a - \*(int\*)b);

}

int kthSmallest(int arr[], int N, int K)

{

qsort(arr, N, sizeof(int), cmpfunc);

return arr[K - 1];

}

int main()

{

int arr[] = { 12, 3, 5, 7, 19 };

int N = sizeof(arr) / sizeof(arr[0]), K = 2;

printf("K'th smallest element is %d", kthSmallest(arr, N, K));

return 0;

}

Output: K'th smallest element is 5