**Ex. No. 1** **LINEAR SEARCH**
**Date:**

**Aim**

**To Implement Linear Search and calculate the time required to search for an element.**

**Algorithm**

1. **Step 1: set pos = -1**
2. **Step 2: set i = 1**
3. **Step 3: repeat step 4 while i <= n**
4. **Step 4: if a[i] == val**
5. **set pos = i**
6. **print pos**
7. **go to step 6**
8. **[end of if]**
9. **set ii = i + 1**
10. **[end of loop]**
11. **Step 5: if pos = -1**
12. **print "value is not present in the array "**
13. **[end of if]**
14. **Step 6: exit**

**Program**

```
#include<stdio.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,ctime;
double cpu_time_used;
begin=clock();
end=clock();
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);
ctime=(end-begin)/cpu_time_used;
printf("\n Time taken is %ld CPU cycles \n",ctime);
break;

default:printf("Invalid choice entered \n");
exit(0);
}
printf("\n Do you wish to run again(1/0) \n");
scanf("%d",&ch);
}
}

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

```
if(n<0) return -1;
if(k==a[n-1])
return (n-1);
else
return linsearch(n-1,a,k);
}
```

**Sample output:**

```
.......MENU.......
 1.Linear search
 2.Exit

 enter your choice
1
enter the number of elements
5
enter the elements of an array
1 2 5 8 7
enter the element to be searched
8
element 8 is found at position 4
 Time taken is -9223372036854775808 CPU cycles

 Do you wish to run again(1/0)
```

**Result**

Thus the program to execute the linear Search was executed successfully.

**EX NO:2** **RECCURSIVE BINARY SEARCH**
**DATE:**

**AIM:**

To Implement the recursive binary search and calculate the CPU running time of the algorithm.

**ALGORITHM**

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

**PROGRAM**

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
#define max 20
int pos;
int binsearch (int,int[],int,int,int);
int linsearch (int,int[],int);
void main()
{ int ch=1; double t; int n,i,a [max],k,op,low,high,pos;
long tick1,tick2;
long elapsed=tick2-tick1;
double elapsed_time = ((double)elapsed/CLOCKS_PER_SEC);

while(ch)
{
printf("\n.......MENU.......\n 1.BinarySearch \n 2.Linear search \n 3.Exit \n");
printf("\n enter your choice\n");
scanf("%d",&op);
switch(op)
{
case 1:printf("\n enter the number of elments\n"); scanf("%d",&n);
printf("\n enter the number of an array in the order \n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\n enter the elements to be searched \n");
scanf("%d",&k); low=0;high=n-1;
tick1=clock();
pos=binsearch(n,a,k,low,high);
tick2=clock();
if(pos==-1)
printf("\n\nUnsuccessful search");
else
```

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

printf("Time taken by the CPU is %lf seconds

\n",elapsed_time);
break;
case 2: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);
tick1=clock();
pos=linsearch(n,a,k);
tick2=clock();
if(pos==-1)
printf("\n\n Unsuccessful search");
else
printf("element %d is found at position %d",k,pos+1);
printf("Time taken by the CPU is %lf seconds

\n",elapsed_time);
break;
default:printf("Invalid choice entered \n");
exit(0);
}
printf("\n Do you wish to run again(1/0) \n");
scanf("%d",&ch);
}

}
int binsearch(int n,int a[],int k,int low,int high)
{
int mid;
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,mid+1,high);
}
int linsearch(int n,int a[],int k)
{
if(n<0) return -1;
if(k==a[n-1])
return (n-1);
else
return linsearch(n-1,a,k);
}
```

**SAMPLE OUTPUT**

```
Output                                                                Clear

/tmp/47Ev97lrXl.o
.......MENU.......
1.BinarySearch
2.Linear search
3.Exit
enter your choice
1
enter the number of elments
6
enter the number of an array in the order
8 5 4 3 4 5
enter the elements to be searched
5
element 5 is found at position 6Time taken by the CPU is 0.000000 seconds

 Do you wish to run again(1/0)
```

**RESULT**

Thus the program to implement the recursive binary search was executed successfully.

**EX NO :3** **NAÏVE PATTERN SEARCH**
**DATE:**

**AIM:**

To perform 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 [ ].

**ALGORITHM**

1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift"

**PROGRAM CODE**

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

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

**SAMPLE OUTPUT**

```
/tmp/zRuIsnL2Cq.o
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

**Result**

Thus the naïve pattern matching algorithm was implemented successfully

**EX NO : 4a** **INSERTION SORT**
**DATE:**

## AIM

To Sort a given set of elements using the Insertion sort method and determine the time required to sort the elements.

## ALGORITHM

1. Iterate from arr[1] to arr[N] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

## PROGRAM

```
#include <math.h>
#include <stdio.h>
#include <time.h>
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   sleep(4);
        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++)
                printf("%d ", arr[i]);
        printf("\n");
}
int main()
{
   long tick1,tick2;
        int arr[] = { 12, 11, 13, 5, 6 };
        int n = sizeof(arr) / sizeof(arr[0]);
tick1 = clock();
```

```
	insertionSort(arr, n);
	tick2 = clock();
	long elapsed = tick2-tick1;
	double elapsed_time = ((double)elapsed/CLOCKS_PER_SEC);
	printArray(arr, n);
printf("Time taken by the CPU is %lf seconds \n",elapsed_time);
	return 0;
}
```

**SAMPLE OUTPUT**

Output

```
/tmp/tlgdhC8Cn9.o
5 6 11 12 13
Time taken by the CPU is 0.000024 seconds
```

**RESULT**

Thus the program to Sort a given set of elements using the Insertion sort method and determine the time required to sort the elements

**EX NO :4b** **HEAP SORT**
**DATE:**

**AIM**

To Sort a given set of elements using the Heap sort method and determine the time required to sort the elements.

**ALGORITHM**

1. First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.
2. Build a heap from the given input array.
3. Repeat the following steps until the heap contains only one element:
   a. Swap the root element of the heap (which is the largest element) with the last element of the heap
   b. Remove the last element of the heap (which is now in the correct position).
   c. Heapify the remaining elements of the heap.
4. The sorted array is obtained by reversing the order of the elements in the input array.

**PROGRAM**

```
#include <stdio.h>
#include<time.h>

void main()
{
  int heap[10], num, i, j, c, rootElement, tempVar;
  long tick1,tick2;
  printf("\n Enter num of elements :");
  scanf("%d", &num);
  printf("\n Enter the nums : ");
  for (i = 0; i < num; i++)
    scanf("%d", &heap[i]);
  for (i = 1; i < num; i++)
  {
     c = i;
     do
     {
        rootElement = (c - 1) / 2;
        if (heap[rootElement] < heap[c]) /* to create MAX heap array */
        {
           tempVar = heap[rootElement];
```

```
            heap[rootElement] = heap[c];
            heap[c] = tempVar;
        }
        c = rootElement;
    } while (c != 0);
 }
tick1=clock();
 printf("Heap array : ");
 for (i = 0; i < num; i++)
    printf("%d\t ", heap[i]);
 for (j = num - 1; j >= 0; j--)
 {
    tempVar = heap[0];
    heap[0] = heap[j];
    heap[j] = tempVar;
    rootElement = 0;
    do
    {
        c = 2 * rootElement + 1;
        if ((heap[c] < heap[c + 1]) && c < j-1)
            c++;
        if (heap[rootElement]<heap[c] && c<j)            {
            tempVar = heap[rootElement];
            heap[rootElement] = heap[c];
            heap[c] = tempVar;
        }
        rootElement = c;
    } while (c < j);
 }
 printf("\n The sorted array is : ");
 for (i = 0; i < num; i++)
   printf("\t %d", heap[i]);
   tick2-clock();
   long elapsed = tick2-tick1;
         double elapsed_time = ((double)elapsed/CLOCKS_PER_SEC);
printf("Time taken by the CPU is %lf seconds \n",elapsed_time);
}
```

**SAMPLE OUTPUT**

Output Clear

```
/tmp/47Ev97lrXl.o
Enter num of elements :6
Enter the nums : 8 9 7 2 6 4
Heap array : 9   8   7   2   6   4
 The sorted array is :   2   4   6   7   8   9Time taken by the CPU is -0.000933
    seconds
```

**RESULT**

Thus the program to implement heap sort was executed successfully

**EX NO :6** **DEPTH FIRST SEARCH**
**DATE:**

## AIM

To Implement the Graph traversal using depth first search.

## ALGORITHM

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

## PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
// Globally declared visited array
int vis[100];
// Graph structure to store number
// of vertices and edges and
// Adjacency matrix
struct Graph {
	int V;
	int E;
	int** Adj;
};
// Function to input data of graph
struct Graph* adjMatrix()
{
	struct Graph* G = (struct Graph*)
		malloc(sizeof(struct Graph));
	if (!G) {
		printf("Memory Error\n");
		return NULL;
	}
	G->V = 7;
	G->E = 7;

	G->Adj = (int**)malloc((G->V) * sizeof(int*));
	for (int k = 0; k < G->V; k++) {
		G->Adj[k] = (int*)malloc((G->V) * sizeof(int));
	}
```

```
        for (int u = 0; u < G->V; u++) {
                for (int v = 0; v < G->V; v++) {
                        G->Adj[u][v] = 0;
                }
        } G->Adj[0][1] = G->Adj[1][0] =
        1; G->Adj[0][2] = G->Adj[2][0] =
        3;
        G->Adj[1][3] = G->Adj[3][1] = 1;
        G->Adj[1][4] = G->Adj[4][1] = 4;
        G->Adj[1][5] = G->Adj[5][1] = 1;
        G->Adj[1][6] = G->Adj[6][1] = 6;
        G->Adj[6][2] = G->Adj[2][6] = 1;

        return G;

}// DFS function to print DFS traversal of graph
void DFS(struct Graph* G, int u)
{
        vis[u] = 1;
        printf("%d ", u);
        for (int v = 0; v < G->V; v++) {
                if (!vis[v] && G->Adj[u][v]) {
                        DFS(G, v);
                }
        }
}
// Function for DFS traversal
void DFStraversal(struct Graph* G)
{
        for (int i = 0; i < 100; i++) {
                vis[i] = 0;
        }
        for (int i = 0; i < G->V; i++) {
                if (!vis[i]) {
                        DFS(G, i);
                }}}
// Driver code
void main()
{
        struct Graph* G;
        G = adjMatrix();
        DFStraversal(G);
}
```

**SAMPLE OUTPUT**

```
Output

/tmp/zRuIsnL2Cq.o
0 1 3 4 5 6 2
```

**RESULT**

Thus the program to implement the graph traversal using depth first search was completed successfully.

**EX NO :7** **DIJIKSTRA'S ALGORITHM**
**DATE:**

## AIM

To implement a program to find the shortest paths to other vertices using Dijkstra's algorithm.

## ALGORITHM

1. Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
2. Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
3. Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
4. Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
5. If the popped vertex is visited before, just continue without using it.
6. Apply the same algorithm again until the priority queue is empty.

## PROGRAM

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.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;
}
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]);

// 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 be 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++) {
		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);
}
```

```
// driver's code
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 } };

	// Function call
	dijkstra(graph, 0);
	return 0;
}
```

**SAMPLE OUTPUT**

Output

```
/tmp/zRuIsnL2Cq.o
Vertex 		 Distance from Source
0 				 0
1 				 4
2 				 12
3 				 19
4 				 21
5 				 11
6 				 9
7 				 8
8 				 14
```

**RESULT**

Thus the program to implement the shortest paths to other vertices using Dijkstra’s algorithm was executed successfully.

**EX NO :9** **FLOYD'S ALGORITHM**
**DATE:**

**AIM**

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

**ALGORITHM:**

1. Initialize the solution matrix same as the input graph matrix as a first step.
2. Then update the solution matrix by considering all vertices as an intermediate vertex.
3. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
4. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
5. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
   - o k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
   - o k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

**PROGRAM**

```
// C Program for Floyd Warshall Algorithm
#include <stdio.h>

// Number of vertices in the graph
#define V 4
/* Define Infinite as a large enough
value. This value will be used
for vertices not connected to each other */
#define INF 99999
// A function to print the solution matrix
void printSolution(int dist[][V]);
// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{

    int i, j, k;

    /* Add all vertices one by one to
    the set of intermediate vertices.
    ---> Before start of an iteration, we
    have shortest distances between all
    pairs of vertices such that the shortest
```

```
	distances consider only the
	vertices in set {0, 1, 2, .. k-1} as
	intermediate vertices.
	----> After the end of an iteration,
	vertex no. k is added to the set of
	intermediate vertices and the set
	becomes {0, 1, 2, .. k} */
	for (k = 0; k < V; k++) {
		// Pick all vertices as source one by one
		for (i = 0; i < V; i++) {
			// Pick all vertices as destination for the
			// above picked source
			for (j = 0; j < V; j++) {
				// If vertex k is on the shortest path from
				// i to j, then update the value of
				// dist[i][j]
				if (dist[i][k] + dist[k][j] < dist[i][j])
					dist[i][j] = dist[i][k] + dist[k][j];
			}
		}
	}

	// Print the shortest distance matrix
	printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
	printf(
		"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)
				printf("%7s", "INF");
			else
				printf("%7d", dist[i][j]);
		}
		printf("\n");
	}
}
// driver's code
int main()
{

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

```
    // Function call
    floydWarshall(graph);
    return 0;
}
```

**SAMPLE OUTPUT**

```
Output

/tmp/zRuIsnL2Cq.o
The following matrix shows the shortest distances between every pair of vertices
0      5      8      9
INF      0      3      4
INF    INF      0      1
INF    INF    INF      0
```

**RESULT**

Thus the program to implement Floyd's algorithm for the All-Pairs- Shortest-Paths problem was executed successfully.

**EX NO :11** **FINDING MAXIMUM AND MINIMUM NUMBERS IN A ARRAY**
**DATE:**

## AIM

To implement a program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.

## ALGORITHM

**1.** Create two intermediate variables max and min to store the maximum and minimum element of the array.
**2.** Assume the first array element as maximum and minimum both, say max = arr[0] and min = arr[0].
**3.** Traverse the given array arr[].
**4.** If the current element is smaller than min, then update the min as the current element.
**5.** If the current element is greater than the max, then update the max as the current element.
**6.** Repeat the above two steps 4 and 5 for the element in the array.

## 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;
}
```

SAMPLE OUTPUT

Output

```
/tmp/zRuIsnL2Cq.o
Enter the total number of numbers : 6
Enter the numbers :
86 98 95 97 2 85
Minimum element in an array : 2
Maximum element in an array : 98
```

**RESULT**

Thus the program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique was executed successfully.

**EX NO :12B** **QUICK SORT**
**DATE:**

**AIM**

To Implement Quick sort methods to sort an array of elements and determine the time required to sort.

ALGORITHM

Step 1 – Pick an element from an array, call it as pivot element.

Step 2 – Divide an unsorted array element into two arrays.

Step 3 – If the value less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

**PROGRAM**

```
#include<stdio.h>
#include <time.h>
void quicksort(int number[25],int first,int last){
  int i, j, pivot, temp;
  sleep(10);
    if(first<last){
    pivot=first;
    i=first;
    j=last;
    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
      i++;
      while(number[j]>number[pivot])
      j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
    temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);
  }
}
int main(){
  int i, count, number[25];
  long tick1,tick2;
```

```
 printf("How many elements are u going to enter?: ");
  scanf("%d",&count);
  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
  scanf("%d",&number[i]);
  tick1=clock();
  quicksort(number,0,count-1);
  tick2=clock();
  long elapsed =tick2-tick1;
  double elapsed_time = ((double)elapsed/CLOCKS_PER_SEC);
  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
  printf(" %d",number[i]);
  printf("Time taken by the CPU is %lf seconds \n",elapsed_time);
  return 0;
}
```

**SAMPLE OUTPUT**

Output Clear

```
/tmp/tlgdhC8Cn9.o
How many elements are u going to enter?: 2
Enter 2 elements: 8 9
Order of Sorted elements:  8 9Time taken by the CPU is 0.000072 seconds
```

**RESULT**

Thus to Implement Quick sort methods to sort an array of elements and determine the time required to sort was executed successfully.

**EX NO :13** **N-QUEENS PROBLEM**
**DATE:**

**AIM**

To Implement N Queens problem using Backtracking

**ALGORITHM**

1. Initialize an empty chessboard of size NxN.
2. Start with the leftmost column and place a queen in the first row of that column.
3. Move to the next column and place a queen in the first row of that column.
4. Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.
5. If all N queens have been placed, print the solution.
6. If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.
7. Remove the queen from the previous column and move it down one row.
8. Repeat steps 4-7 until all possible configurations have been tried.

**PROGRAM**

```
#define N 4
#include <stdbool.h>
#include <stdio.h>

void printSolution(int board[N][N])
{
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++)
                        printf(" %d ", board[i][j]);
                printf("\n");
        }
}
bool isSafe(int board[N][N], int row, int col)
{
        int i, j;

        /* Check this row on left side */
        for (i = 0; i < col; i++)
                if (board[row][i])
                        return false;

        /* Check upper diagonal on left side */
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
                if (board[i][j])
                        return false;

        /* Check lower diagonal on left side */
```

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

	return true;
}
/* A recursive utility function to solve N
Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
	/* base case: If all queens are placed
	then return true */
	if (col >= N)
		return true;

	/* Consider this column and try placing
	this queen in all rows one by one */
	for (int i = 0; i < N; i++) {
		/* Check if the queen can be placed on
		board[i][col] */
		if (isSafe(board, i, col)) {
			/* Place this queen in board[i][col] */
			board[i][col] = 1;
			/* recur to place rest of the queens */
			if (solveNQUtil(board, col + 1))
				return true;

			/* If placing queen in board[i][col]
			doesn't lead to a solution, then
			remove queen from board[i][col] */
			board[i][col] = 0; // BACKTRACK
		}
	}
	/* If the queen cannot be placed in any row in
		this column col then return false */
	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) {
		printf("Solution does not exist");
```

```
                return false;
        }

        printSolution(board);
        return true;
}

// driver program to test above function
int main()
{
        solveNQ();
        return 0;
}
```

**SAMPLE OUTPUT**

```
Output
/tmp/tlgdhC8Cn9.o
0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0
```

**RESULT**

Thus to Implement N Queens problem using Backtracking was executed successfully.