Experiment 1: Implement binary search procedure using the divide and conquer method

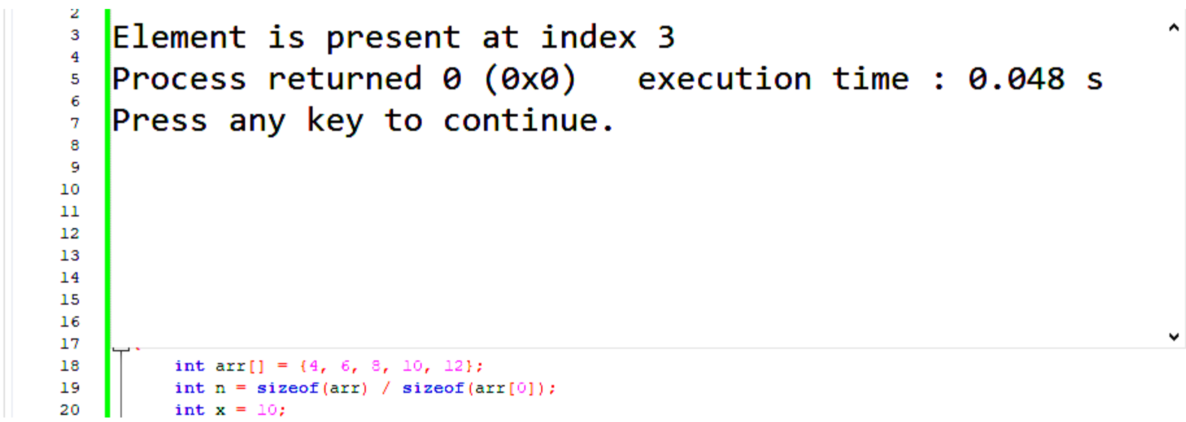
Algorithm:

```
BinSrch(a, i,l,x)
// Given an array a[i:l] of elements in nondecreasing
// order,1≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.

{

if (l = i) then // If Small(P)
{
if (x = a[i]) then return i;
   else return 0;
}
else
{
   mid:=[(i+l)/2]      // Reduce P into a smaller subproblem
   if (x = a[mid])then return mid;
   else if (x<a [mid]) then
   return BinSrch (a, i, mid-1, x);
   else return BinSrch(a, mid+1, l, x);
}

}
```

Program:

```c
#include <stdio.h>
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
            return m;
        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}

int main(void)
{
    int arr[] = {4, 6, 8, 10, 12};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present in array")
    : printf("Element is present at index %d", result);
    return 0;
}
```

Output:



Experiment 2: Implement the divide and conquer method for finding the maximum and minimum numbers.

Algorithm:

```
MaxMin(i, j, max, min)
// a[l:n] is a global array. Parameters i and j are integers,
//l≤ i ≤ j ≤ n The effect is to set max and min to the
// largest and smallest values in a[i:j], respectively.
{

if (i = j) then max:=min:=a[i]; // Small(P)
else if (i = j - 1) then // Another case of Small(P)
{
if (a[i]<a[j]) then
{
 max:=a[j];min:=a[i]
else
{
max:=a[i]; min:=a[j];
}
}
else
{
//If P is not small, divide P into subproblems.
// Find where to split the set.
mid:=[(i+j)/2];
// Solve the subproblems.
MaxMin(i,mid,max,min);
MaxMin(mid+1,j,max1,min1);
// Combine the solutions.
if (max<max1) then max:=maxl;
if (min >min1)then min:=min1;
}

}
```

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 : 5
Enter the numbers :
1
3
5
7
4
Minimum element in an array : 1
Maximum element in an array : 7

Process returned 0 (0x0)   execution time : 18.186 s
Press any key to continue.
```

Experiment 3: Write a program to measure the performance using the time function between bubble sort and quick sort.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Function to swap two elements
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void bubbleSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
        }
    }
}
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);
}
// Quicksort function
void quickSort(int arr[], int low, int high)
{
```

```
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main()
{
    int testArraySize = 10000;  // Adjust the size of the test array as per your requirement
    int testArray[testArraySize];
    int tempArray[testArraySize];
    // Generate random test array
    srand(time(0));
    for (int i = 0; i < testArraySize; i++)
    {
        testArray[i] = rand();
    }
    // Measure time for bubble sort
    clock_t start = clock();
    bubbleSort(testArray, testArraySize);
    clock_t end = clock();
    double bubbleSortTime = (double)(end - start) / CLOCKS_PER_SEC;
    // Copy the unsorted array for quicksort
    for (int i = 0; i < testArraySize; i++)
    {
        tempArray[i] = testArray[i];
    }
    // Measure time for quicksort
    start = clock();
    quickSort(tempArray, 0, testArraySize - 1);
    end = clock();
    double quickSortTime = (double)(end - start) / CLOCKS_PER_SEC;
    // Print the results
    printf("Bubble Sort Time: %f seconds\n", bubbleSortTime);
    printf("Quick Sort Time: %f seconds\n", quickSortTime);
    return 0;
}
```

Output

```
Bubble Sort Time: 0.562000 seconds
Quick Sort Time: 0.544000 seconds

Process returned 0 (0x0)   execution time : 1.512 s
Press any key to continue.
```

Experiment 4: Implement the fractional knapsack problem that will generate an optimal solution for the given set of instances

Basic Algorithm

1. Sort the instances in descending order based on their value-to-weight ratios.
2. Initialize the total value and current weight to zero.
3. Iterate through the sorted instances:
    - If the current weight plus the weight of the current instance is less than or equal to the knapsack capacity, include the whole instance in the knapsack:
        - Increase the current weight by the weight of the current instance.
        - Increase the total value by the value of the current instance.
    - Otherwise, include a fraction of the current instance in the knapsack to fill the remaining capacity:
        - Calculate the remaining weight that can be included (capacity - current weight).
        - Calculate the value of the remaining weight as the remaining weight multiplied by the value-to-weight ratio of the current instance.
        - Add the value of the remaining weight to the total value.
        - Break out of the loop.
4. Return the total value as the maximum value achievable.

Program

```
#include <stdio.h>

// Structure to represent an instance
struct Instance {
    int weight;
    int value;
    double ratio;
};

// Function to compare ratios for sorting
int compare(const void* a, const void* b) {
    struct Instance* instanceA = (struct Instance*)a;
    struct Instance* instanceB = (struct Instance*)b;
    double ratioA = instanceA->ratio;
    double ratioB = instanceB->ratio;
```

```
    if (ratioA < ratioB)
        return 1;
    else if (ratioA > ratioB)
        return -1;
    else
        return 0;
}

// Function to find the maximum value using the fractional knapsack
approach
double fractionalKnapsack(int capacity, struct Instance instances[], int
numInstances) {
    qsort(instances, numInstances, sizeof(instances[0]), compare);

    double totalValue = 0.0;
    int currentWeight = 0;

    for (int i = 0; i < numInstances; i++) {
        if (currentWeight + instances[i].weight <= capacity) {
            currentWeight += instances[i].weight;
            totalValue += instances[i].value;
        } else {
            int remainingWeight = capacity - currentWeight;
            totalValue += (double)remainingWeight * instances[i].ratio;
            break;
        }
    }

    return totalValue;
}

int main() {
    // Example instances
    struct Instance instances[] = {
        {10, 60, 0.0},
        {20, 100, 0.0},
        {30, 120, 0.0}
    };

    int capacity = 50;
    int numInstances = sizeof(instances) / sizeof(instances[0]);

    for (int i = 0; i < numInstances; i++) {
        instances[i].ratio = (double)instances[i].value /
instances[i].weight;
    }
```

```
    double maxValue = fractionalKnapsack(capacity, instances,
numInstances);
    printf("Maximum value achievable: %.2f\n", maxValue);

    return 0;
}
```

Output

```
Maximum value achievable: 240.00

Process returned 0 (0x0)   execution time : 0.054 s
Press any key to continue.
```

Experiment 5: Write a program to find the minimum cost-spanning tree using Prim's algorithm.

Algorithm

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

Program

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

#define V 5

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

    return min_index;
}

int printMST(int parent[], int n, int graph[V][V]) {
    int i;
    printf("Edge   Weight\n");
    for (i = 1; i < V; i++)
        printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

void primMST(int graph[V][V]) {
    int parent[V]; // Array to store constructed MST
    int key[V], i, v, count; // Key values used to pick minimum weight
edge in cut
    int mstSet[V]; // To represent set of vertices not yet included in
MST

    // Initialize all keys as INFINITE
    for (i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = 0;

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is picked as first
vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;

        for (v = 0; v < V; v++)
```

```
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, V, graph);
}

int main() {
    /* Let us create the following graph
     2    3
    (0)--(1)--(2)
     |   / \   |
    6| 8/   \5 |7
     | /     \ |
    (3)-------(4)
     9          */
    int graph[V][V] = { { 0, 2, 0, 6, 0 }, { 2, 0, 3, 8, 5 },
            { 0, 3, 0, 0, 7 }, { 6, 8, 0, 0, 9 }, { 0, 5, 7, 9, 0 }, };

    primMST(graph);

    return 0;
}
```

Output

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

Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```

Experiment 6: Write a program to implement a dynamic programming method for all pair's shortest path problems.

Algorithm

Algorithm AllPaths($cost, A, n$)
// $cost[1:n, 1:n]$ is the cost adjacency matrix of a graph with
// n vertices; $A[i,j]$ is the cost of a shortest path from vertex
// i to vertex j. $cost[i,i] = 0.0$, for $1 \le i \le n$.
{
    for $i := 1$ **to** n **do**
        for $j := 1$ **to** n **do**
            $A[i,j] := cost[i,j]$; // Copy $cost$ into A.
    for $k := 1$ **to** n **do**
        for $i := 1$ **to** n **do**
            for $j := 1$ **to** n **do**
                $A[i,j] := \min(A[i,j], A[i,k] + A[k,j])$;
}

Program

```
#include <stdio.h>
#define V 4
#define INF 99999
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][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    printSolution(dist);
}
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");
    }
}
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 distances between every pair of vertices
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```