Name: John Nixon.
USN: 2GI19IS016

Team Work: 01

Problem Definition:

Implement Mergesort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list list to be sorted and plot a graph of the time taken verses n.

Theory:

Divide and Conquer: The most well known design strategy

* Divide instance of problem into 2 (or) more smaller instances.
* solve smaller instances recursively.
* obtain solution ~~instances recursively~~ to original (larger) instance by combining these solutions.

Mergesort:

* Split array A[0----n-1] into about equal halves and make copies of each half in arrays B and C.
* Sort arrays B and C recursively.
* Merge sorted arrays B and C into array A.

Algorithm:

Pseudocode of Mergesort:

```
Algorithm Mergesort (A, low, high)
    // A[low : high] is global array to be sorted
    // Small(p) is true if there is only one element
    // to sort. In this case the list is already sorted
{
  if (low < high) then
  {
    // Divide P into subproblems
    // Find where to split to set
    mid := [(low + high) /2];
    // Solve the subproblems
    Mergesort (A, low, mid);
    Mergesort (A, mid+1, high);
    // Combine the solutions
    Merge (A, low, mid, high);
  }
}
```

Pseudocode of Merge:

```
Algorithm Merge (A, low, mid, high)
  // A [low:high] is a global array containing
2 sorted subsets in A [low: mid]
```

```
// and in A[mid+1 : high].
// The goal is to merge these 2 sets into a single set
// residing in A[low : high]
// B[] is an auxiliary global array
{
  h := low; i := low; j := mid+1;
  while ((h <= mid) and (j <= high)) do
  {
    if (A[h] <= A[j]) then
    {
      B[i] := A[h];
      h := h+1;
    }
    else
    {
      B[i] := A[j];
      j := j+1;
    }
    i := i+1;
  }
  if (h > mid) then
    for k := j to high do
    {
      B[i] := A[k];
      i := i+1;
    }
```

```
else
    for k := h to mid do
    {
        B[i] := A[k]; i := i + 1;
    }
    for k := low to high do A[k] := B[k];
}
```

Source Code

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

#define MAX 100000
void getdata (int arr[])
{
    int i;
    // generating random numbers
    for (i=0; i< MAX; i++)
    {
        arr[i] = rand();
    }
}
void sort (int arr[], int low, int mid, int high)
{
    int i, j, k, l, b[MAX];
    l = low;

    i = low;
    j = mid + 1;
    while ((l <= mid) && (j <= high))
    {
```

```
    if (arr[l] <= arr[j])
    {
        b[i] = arr[l];
    }
    else
    {
        b[i] = arr[j];
        j++;
    }
    i++;
}
if (l > mid)
{
    for (k = j; k <= high; k++)
    {
        b[i] = arr[k];
        i++;
    }
}
else
{
    for (k = r; k <= mid; k++)
    {
        b[i] = arr[k];
        i++;
    }
}
```

```
    for (k = low; k <= high ; k++)
    {
        arr [k] = b[k];
    }
void partition (int arr[], int low, int high)
{
    int mid ;
    if ( low < high)
    {
      mid = ( low + high ) /2 ;
      partition (arr, low, mid );
      partition (arr, mid +1, high );
      sort (arr, low, mid , high);
    }
}
int main (int argc, char * argv [])
{
    int array [MAX] ;
    int n, i ;
    getdata (array);
    clock_t t ;
    t = clock ();
```

```
partition (array, 0, MAX-1);
t = clock() - t;
double time_taken = ((double)t) / CLOCKS_PER_SEC;
                                    // in seconds.
printf("fun() took %f seconds to execute \n",
                                    time_taken);
return 0;
}
```

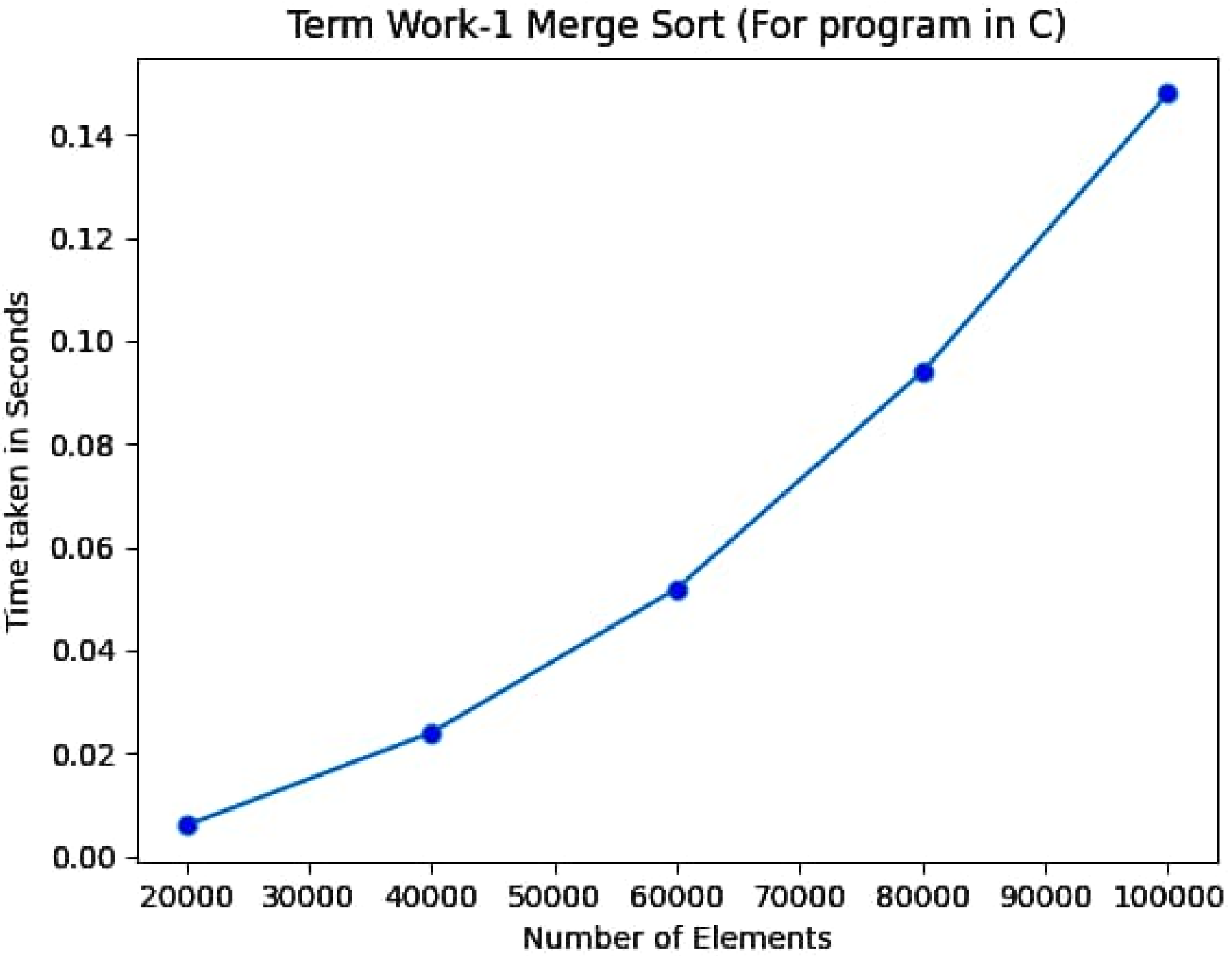
Refernces:

* Kenneth Berman, Jerome Paul, Algorithms, Cengage learning

* Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford Stein, Introduction to Algorithms PHI. 2nd edition and onwards.

Conclusion:

In the teamwork, we learnt about divide and conquer technique and implemented mergeSort algorithm design technique. We also learned computing time required for recursive and iterative algorithms.


Term Work-1 Merge Sort (For program in C)
Time taken in Seconds
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
20000
30000
40000
50000
60000
70000
80000
90000
100000
Number of Elements

```
----------Merge Sort----------
Array before Sorting ::  40647 2534 44071 21220 21115 18922 16577 38449 74546 8366 42168 15716 16034 5080 29443
Array after sorting ::  2534 5080 8366 15716 16034 16577 18922 21115 21220 29443 38449 40647 42168 44071 74546
PS D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>
```

Name : John Nixon.
USN : 2GI 19 IS 016

Term Work : 02

Problem Definition :

Implement Quick sort algorithm 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.

Theory:

Quicksort algorithm:

Given an array of n elements (eg: integers):

* If an array contains only one element, then return
* Else:
  - pick one element to use as pivot.
  - Partition elements into 2 sub-arrays:
    * First array that contains elements less than (a) equal to pivot.
    * second array that contains elements greater than pivot.
  - Quicksort 2 sub-arrays
  - Return results

Algorithms:

```
Quicksort (A[l...r])
// sorts a subarray by quicksort
// input : A subarray A[l...r] of A[0...n-1] defined by its left //
and right indices L and r
// Output : The subarray A[L...r] sorted in non decreasing
            order.

if (L < r)
    s <-- partition (A[L...r])
    Quicksort (A[L... s-1])
    Quicksort (A[s+1...r])
return

Algorithm Partition (A[L-r])
    // Partitions a subarray by using its first element and as
    a pivot
  // Input : A subarray A[L...r] of A[0...n-1], defined by
its left and right indices l and r (l < r)
  // Output : A partition of A[l...r] with the split position
returned as this functions value

        p ← A[i]
        i ← l ; j ← r+1
```

```
repeat
        repeat i ← i + 1 until  A[i] >= p
        repeat j ← j - 1 until  A[j] < p
        swap (A[i], A[j])
until i >= j
swap(A[i], A[j] // undo last swap when i >= j
swap (A[i], A[j]);
return j;
// end
```

Source Code.

```
# include <stdio.h>
# include <stdlib.h>
# include <time.h>
int MAX_ELEMENTS = 500;

void swap (int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int choose_pivot (int i, int j)
{
    return ((i+j) /2);
}

void quicksort (int list [], int m, int n)
{
    int key, i, j, k;
    if (m < n)
    {
        k = choose_pivot (m, n);
        swap (&list [m], &list [k]);
        key = list[m];
```

```
        i = m+1;
        j = n;
        while (i <= j)
        {
            while((i<=n) && (list[i]<=key))
                i++;
            while((j>=m) && (list[j]>key))
                j--;
            if (i<j)
                swap(&list[i], &list[j]);
        }
        // swap two elements
        swap(&list[m], &list[j]);
        // recursively sort the lesser list
        quicksort(list, m, j-1);
        quicksort(list, j+1, n);
    }
}
void printlist(int list[], int n)
{
    int i;
    for(i=0; i<n; i++)
        printf("%d\t", list[i]);
}
int main(int argc, char *argv[])
{
    time_t t1, t2;
    int list[MAX_ELEMENTS];
```

```
int j=0, i=0;
// generate random numbers and fill them to the list
for(i=0; i<MAX_ELEMENTS; i++){
    list[i] = rand();
}
printf("The list before sorting is:\n");
printlist(list, MAX_ELEMENTS);
t1 = time(&t1);
// sort the list using quicksort
for(i=0; i<MAX_ELEMENTS; i++)
for(j=0; j<MAX_ELEMENTS; j++)
quicksort(list, 0, MAX_ELEMENTS-1);
t2 = time(&t2);
// print the result
printf("The list after sorting using quicksort algorithm:\n");
printlist(list, MAX_ELEMENTS);
printf("time taken: %.f\n", (float)(t2-t1)/CLOCKS_PER
                                        -SEC);
// system("PAUSE");
return 0;
}
```

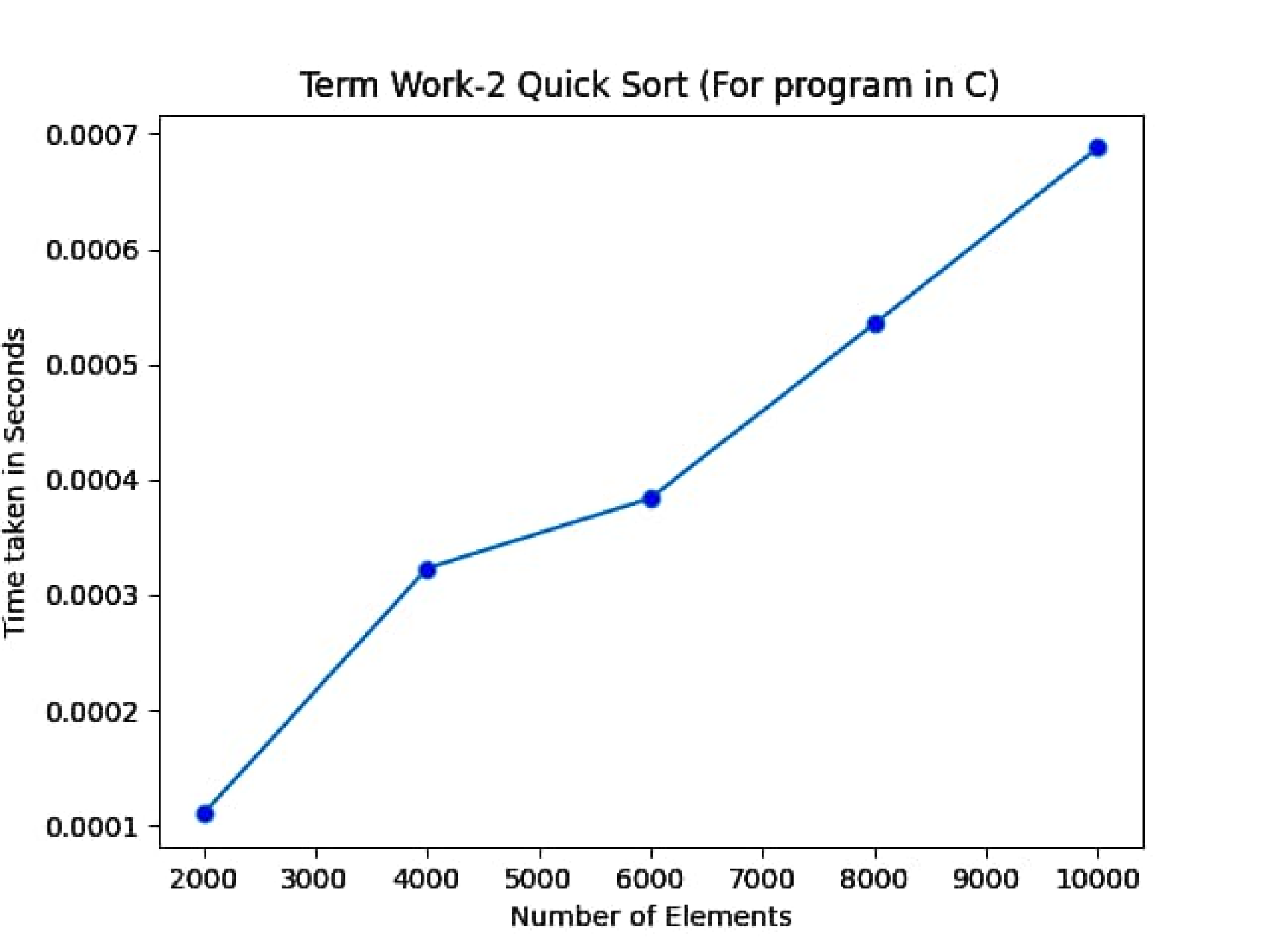
References:

* Kenneth Berman, Jerone Paul, Algorithms, Cengage learning.

* Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford Stein, introduction to Algorithm PHI, 2nd edition and onwards

Conclusion:

In this teamwork, we learnt about divide and conquer technique and implementation of quicksort algorithm design technique we also learned computing time required for recursive and interative algorithms.


Term Work-2 Quick Sort (For program in C)
Time taken in Seconds
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of Elements

```
----------Quick Sort----------
Array before sorting ::  5713 74921 69409 73302 62508 64018 28726 21140 31565 24849 51198 59030 8789 61678 76381
Array after sorting ::  5713 8789 21140 24849 28726 31565 51198 59030 61678 62508 64018 69409 73302 74921 76381
PS D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>
```

Name: John Nixon
USN : 2GI 19IS016

Term Work : 03

Problem Definition:

Implement Insertion sort algorithm and determine the time required to sort the elements.
Repeat the experiment your different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

Theory:

* To sort array A[0--n-1], sort A[0...n-2] recursively and then insert A[n-1] in its proper among the sorted A[0...n-2]
* Usually implemented bottom up (nonrecursively)
* Insertion sort is an example of decrease by a constant (usually by 1) type of decrease and conquer technique.

Decrease and Conquer:

- Reduce problem instance to smaller instance of the same problem.
- Solve smaller instance
- Extend solution of smaller instance to obtain solution to original instance.

Algorithm:

```
InsertionSort (A[0..n-1])
// sorts a given array by insertion sort
// Input : An array A[0..n-1] of n orderable elements
// output : Array A[0..n-1] sorted in nondecreasing orders.

for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v

// end
```

Source Code.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10000

void getdata(int arr[])
{
    int i;
    // generating random numbers
    for (i=0; i<MAX; i++)
    {
        arr[i] = rand();
    }
}
void insertion_sort(int *x)
{
    int temp, i, j;

    for (i=1; i<MAX; i++)
    {
        temp = x[i];
        j = i-1;
        while (temp < x[j] && j>=0)
        {
            x[j+1] = x[j];
            j = j-1;
        }
        x[j+1] = temp;
```

```
    }
}
int main (int argc, char *argv[])
{
    int a[MAX];
    getdata (a);
    clock_t t;

    t = clock ();
    insertion_sort (a);
    t = clock() - t;
    double time_taken = ((double)t) / CLOCKS_PER_SEC; // in seconds

    print ("fun() took %f seconds to execute \n", time_taken);

    return 0;
}
```

References :

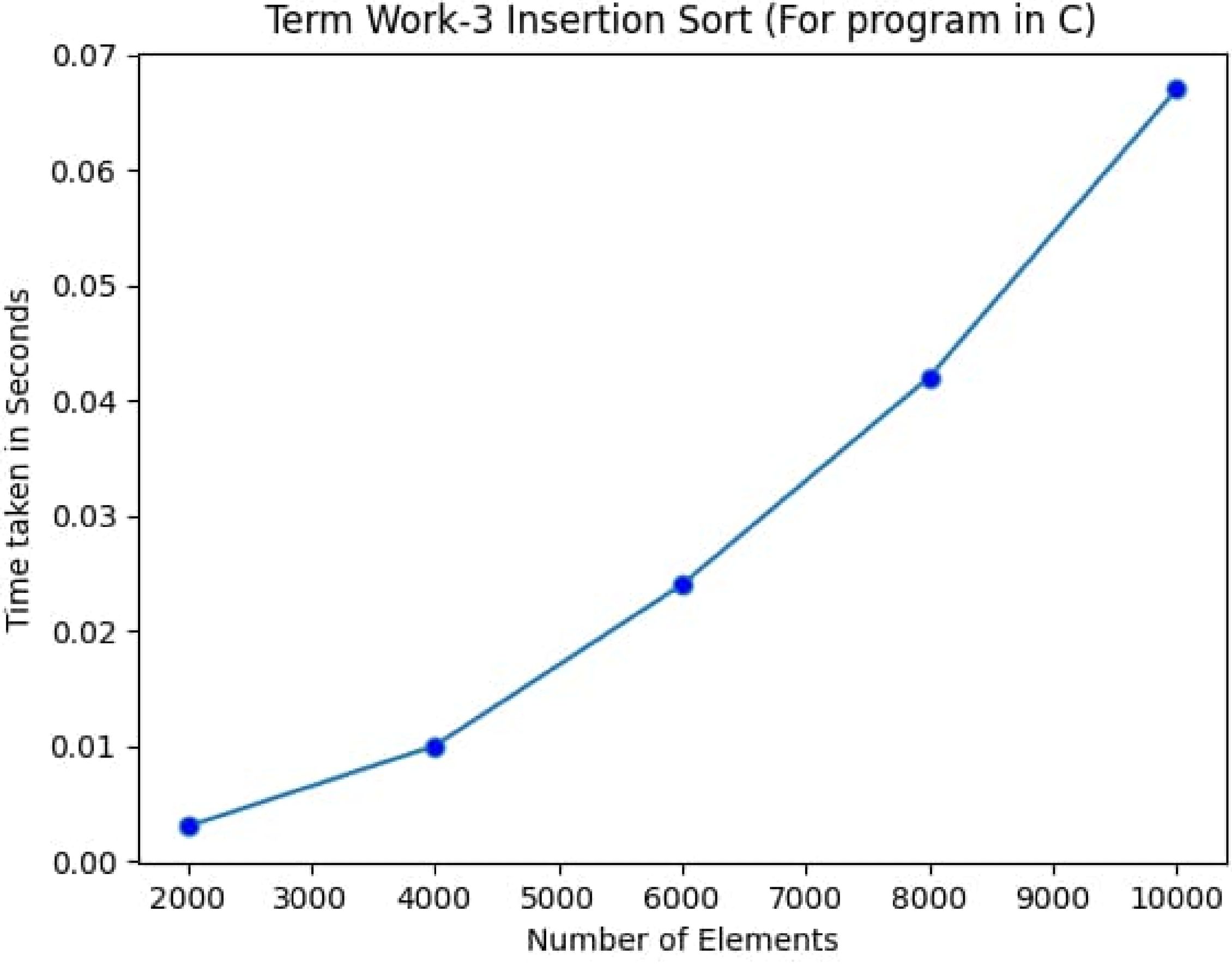
* Kenneth Berman, Jerome Paul, Algorithms, Cengage Learning.
* Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford stein. introduction to Algorithms PHI, 2nd edition and onwards.

Conclusion:

In this termwork, we learnt about decreare and conquer techniques and implementation of Insertion sort algorithm design techniques.

We also learned computing time required for recursive and iterative algorithms.

```
----------Quick Sort----------
Array before sorting ::  45244 11747 59295 3006 9775 71382 68397 57296 11127 35777 75406 43269 35340 32097 21899
Array after sorting ::  3006 9775 11127 11747 21899 32097 35340 35777 43269 45244 57296 59295 68397 71382 75406
PS D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>
```


Term Work-3 Insertion Sort (For program in C)
Time taken in Seconds
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of Elements

Name: John Nixon.
USN : 2GD 19IS 016

Term Work : 04

Problem Definition:

Implementation Heap sort algorithm 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.

Theory:

Transform and conquer : solves a problem by a transformation to :

* a simpler instance of the same problem
* a different representation of same instance.
* a different problem for which an algorithm is available.

Heaps and Heapsort :-

A heap is a binary tree with keys at its nodes such that :

* It is essentially complete, i.e, all its levels only some rightmost keys may be missing.

Name: John Nixon.
USN : 2GI 19IS 016

Term Work : 04

Problem Definition :

Implementation Heap sort algorithm 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.

Theory :

Transform and Conquer : solves a problem by a transformation to :

* a simpler instance of the same problem
* a different representation of same instance.
* a different problem for which an algorithm is available.

Heaps and Heapsort :-

A heap is a binary tree with keys at its nodes such that :

* It is essentially complete, i.e, all its levels only some rightmost keys may be missing.

* The key at each node is ≥ all keys in its children (and descendants).

Algorithms:

```
Algorithm HeapBottomUp (H[1...n])
// constructs a heap from a given array.
// by the bottom up algorithm
// Input : An array H[1...n] of orderable items.
// output : A heap H[1...n]

for  i ← [n/2] downto 1 do
     k ← i ; v ← H[k]
     heap ← false
     while not heap and 2 * k ≤ n do
          j ← 2 * k
     // there are 2 children
     if j < n
       if H[j] < H[j+1]
            j ← j + 1
       if v ≥ H[j]
            heap ← true
       else
```

$H[R] \leftarrow H[j];$

$R \leftarrow j$

$H[k] \leftarrow V$

Source Code.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 1500
void getdata (int arr[])
{
    int i;
      // generating random numbers
      for (i=0; i< MAX; i++)
      {
          arr[i]= rand();
      }
}
void heap-bottom-up (int n, int a[])
{
   int i, j, p, k, v, heap;
   p = n/2;

   for (i=p; i>0; i--)
   {
      heap =0;
      k =i;
      v = a[k];

      while (! heap && ((2*k) <= n))
      {
        j = 2*k;
        if (j<n)
        {
            if (a[j]<a[j+1])
```

```
            j = j + 1;
        }
        if (v >= a[j])
            heap = 1;
        else
        {
            a[k] = a[j];
            k = j;
        }
    }
    a[k] = v;
  }
}
void heapsort(int n, int a[])
{
  int i, temp;
  heap_bottom_up(n, a);
  for(i=n; i>1; i--)
  {
      temp = a[1];
      a[1] = a[i];
      a[i] = temp;
      heap_bottom_up(i-1, a);
  }
}
int main()
{
  int i, a[MAX], j, n;
  clock_t start, end;
  double duration;
```

```
    getdata(a);
    clock_t t;
        t = clock();
        heapsort(MAX, a);
        t = clock() - t;
        double time_taken = ((double)t) / CLOCKS_PER_SEC; // in seconds
        printf("fun() took %f seconds to execute \n", time_taken);
    return 0;
}
```

Reference:

* Kenneth Berman, Jerone Paul, Algorithms. Cengage learning.

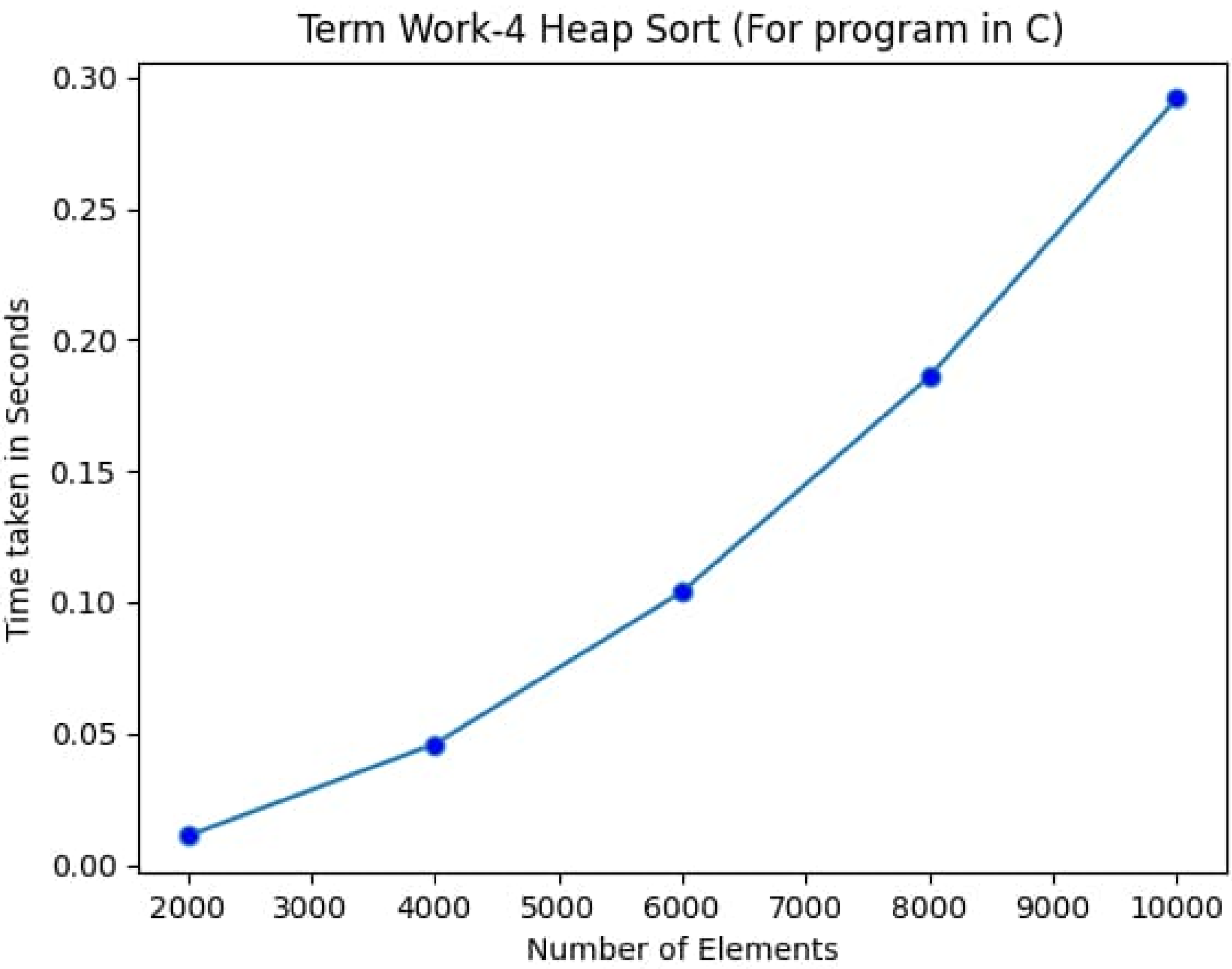
* Thomas H. Cormen. Charles. E. Leiserson, Ronal L. Rivest, Clifford Stein, Introduction to Algorithms PHI, 2nd edition and onwards.

Conclusion:

In this teamwork, we learnt about transform and conquer technique and implementation of Heaps and Heapsort algorithm.

We also learned computing time required for recursive and iterative algorithms.

```
----------Heap Sort----------
Array before sorting ::  76388 24284 46877 14828 60573 45882 77938 23725 76796 42958 21868 23992 39752 54837 51388
Array after sorting ::  14828 21868 23725 23992 24284 39752 42958 45882 46877 51388 54837 60573 76388 76796 77938
PS D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>
```


Term Work-4 Heap Sort (For program in C)
Time taken in Seconds
0.30
0.25
0.20
0.15
0.10
0.05
0.00
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of Elements

Name: John Nixon.
USN: 2GI19IS016

Term Work : 05

Problem Definition:

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's Algorithm.

Theory:

Greedy Technique: Constructs a solution to an optimization problem piece by piece through a sequence of choices that are:

* feasible, i.e, satisfying the constraints
* locally optimal (wrt some neighborhood definition)
* greedy (in terms of some measure) and irrevocable

Dijkstra's Algorithm: With a different way of computing numerical labels: Among vertices not already in the ~~terms~~ tree, it finds vertex u with the smallest sum.

$$d_v + w(v,u)$$

where, v is a vertex

d_v is length of shortest path form source is to v.

$w(v,u)$ is the length (weight) of edge from v to u.

Algorithm:

```
Dijkstra(s)...
{
  # distance to source vertex is zero
    dist[s] ← 0
    for all v ∈ V - {s}
    // set all other distances to infinity
        do dist[v] ← ∞
    // s, the set of visited vertices is initially empty
    S ← ∅
  // Q, the queue initially contains all vertices
    Q ← V
  // while queue is not empty.
      while Q ≠ ∅ do
  // select element of Q with min distance
        u ← minDistance(Q, dist)
  S ← S ∪ {v} // add u to list of visited vertices
  for all v ∈ neighbours[u] do
```

// if new shortest path found

if dist[v] > dist[u] + w(u, v)

// set new value of shortest path

then d[v] ← d[u] + w(u, v)

return dist

}

Source Code.

```
# include <stdio.h>

    # define  INFINITY  9999
    # define  TRUE   1
    # define  FALSE  0

    void  InitGraph (int  g[20][20], int n)
    {
        int i, j ;
        for (i=1 ; i<=n ; i++)
        for (j=1 ; j<=n ; j++)
            g[i][j] = INFINITY ;
    }
    void  ReadGraph(int g[20][20])
    {
        int i, j , wt ; int ch;
        do
        {
            printf ("\n Input Directed Edge <V1, V2, Wt > : ");
            scanf ("%d %d %d", &i , &j , &wt);
            g[i][j] = wt ;
            printf (" One More Edge then press 1 else type any key ");
            scanf ("%d", &ch);
    } while (ch == 1);
}
void PrintGraph(int a [20][20], int n)
```

```
{
    int i, j;
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n; j++)
    printf("%.8d", a[i][j]);
        printf("\n\n");
    }
}
int MIN(int a, int b)
{
   return (a<b) ? a : b);
}
int MinNode (int dist[], int n, int selected[])
{
    int k, min = INFINITY, index = 0;
    for (k=1; k<=n; k++)
       if ( dist[k]<min && selected[k]==FALSE)
        { min = dist[k]; index = k; }
    return (index);
}
void mspaths (int g[20][20], int n, int sv, int dist[])
{
    int selected[20], k, u, w;
    for (k=1; k<=n; k++)
    {
        selected[k] = FALSE;
        dist[k] = g[sv][k];
    }
```

```
        selected [sv] = TRUE;
        dist [sv] = 0;
        for ( k=1; k<=n-1; k++)
        {
    u = MinNode ( dist , n , selected);
    selected [u] = TRUE;
    for ( w = 1; w<=n; w++)
        if ( selected [w] == FALSE )
            dist [w] = MIN ( dist [w], dist [u] + g [u] [w] );
        }
}
void main ()
{
    int n, g [20][20], sv, dist [20];
    int k;
    printf ("\n Enter How many nodes : ");
    scanf ("%d", &n);
    InitGraph (g, n);
    ReadGraph (g);
printf ("\n Enter the starting vertex : ");
    scanf ("%d", &sv);
    shortpaths ( g, n, sv, dist);
    printf ("\n\n The given graph in Matrix Format : \n\n");
    PrintGraph (g, n);
    getchar ();
```

```
printf("\n single Vertex Shortest Paths : \n");
for(k = 1 ; k<=n; k++)
    printf("\n From Node - %d to Node - %d : %d",
                                    \v, k, dist[k]);

getchar();
}
```

Reference:

* Kenneth Berman, Jerone Paul, Algorithms, Cengage learning.

* Thoms H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford stein, introduction to Algorithms PHI, 2nd edition and Onwards.

Conclusion:

In this teamwork, we learnt about Greedy Technique and implementation of the Dijkstra's Algorithm design technique.

We also learned computing time required for recursive of iterative algorithms.

```
D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>"d:\ISE - 4th Sem\SEM-4

 Enter How many nodes : 5

 Input Directed Edge <v1, v2, Wt> : 1 2 3
 One More Edge then press 1 else type any key 1

 Input Directed Edge <v1, v2, Wt> : 2 4 2
 One More Edge then press 1 else type any key 1

 Input Directed Edge <v1, v2, Wt> : 1 4 7
 One More Edge then press 1 else type any key 1

 Input Directed Edge <v1, v2, Wt> : 2 3 4
 One More Edge then press 1 else type any key 1

 Input Directed Edge <v1, v2, Wt> : 3 4 5
 One More Edge then press 1 else type any key 1

 Input Directed Edge <v1, v2, Wt> : 3 5 6
 One More Edge then press 1 else type any key 1

 Input Directed Edge <v1, v2, Wt> : 4 5 4
 One More Edge then press 1 else type any key 2

 Enter the Starting Vertex : 1


 The Given Graph in Matrix Format :

    9999       3    9999       7    9999

    9999    9999       4       2    9999

    9999    9999    9999       5       6

    9999    9999    9999    9999       4

    9999    9999    9999    9999    9999


 Single Vertex Shortest Paths :

 From Node-1 to Node-1 : 0
 From Node-1 to Node-2 : 3
 From Node-1 to Node-3 : 7
 From Node-1 to Node-4 : 5
 From Node-1 to Node-5 : 9
```

Name : John Nixon
USN : 2GI 19IS 016

Term Work : 06

Problem Definition :

Find the minimum cost spanning Tree of a given undirected graph using Prim's Algorithm.

Theory :

Prim's Algorithm :

A spanning tree of an undirected graph G is a subgraph of G that is a tree containing all the vertices of G.

In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.

A minimum spanning Tree (MST) for a weighted undirected graph is a spanning tree with minimum weight.

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.

Algorithm :

```
Prim(G)
// Prim's algorithm for constructing a minimum spanning tree
// Input : A weighted connected graph G = (V, E)
```

// Output : E_T, the set of edges comprising a minimum spanning tree of G.

$V_T \leftarrow \{v_0\}$ // the set of tree vertices can be initialized with any vertex.

$E_T \leftarrow \emptyset$

for $i \leftarrow 1$ to $|V| - 1$ do

find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

Source code.

```
# include <stdio.h>
# include <stdlib.h>
# define infinity 9999
# define MAX 20
int G[MAX][MAX], spanning[MAX][MAX], n;
int prims();
int main()
{
    int i, j, total_cost;
    printf("Enter no. of vertices :");
    scanf("%d", &n);
    printf("\n Enter the adjacency matrix :\n");
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            scanf("%d", &G[i][j]);
    total_cost = prims();
    printf("\nspanning tree matrix :\n");
    for(i=0; i<n; i++)
    {
        printf("\n");
```

```
        for (j=0; j<n; j++)
            printf("%.d\t", spanning[i][j]);
    }
    printf("\n\nTotal cost of spanning tree = %d", total_cost);
    return 0;
}
int prims()
{
    int cost[MAX][MAX];
    int u, v, min_distance, distance[MAX], from[MAX];
    int visited[MAX], no_of_edges, i, min_cost, j;
    // create cost[][] matrix, spanning[][]
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
        {
            if(G[i][j]==0)
                cost[i][j] = infinity;
            else
                cost[i][j] = G[i][j];
            spanning[i][j] = 0;
        }
```

```
// initialise visited[], distance[] and from[]
distance[0] = 0;
visited[0] = 1;
for (i = 1; i < n; i++)
{
        distance[i] = cost[0][i];
        from[i] = 0;
        visited[i] = 0;
}
min_cost = 0;          // cost of spanning tree
no_of_edges = n - 1;   // no. of edges to be added
while (no_of_edges > 0)
{
    // find the vertex at minimum distance from the
       tree
    min_distance = infinity;
    for (i = 1; i < n; i++)
        if (visited[i] == 0 && distance[i] < min
                                _distance)
        {
            v = i;
            min_distance = distance[i];
        }
```

```
u = from[v];
// insert the edge in spanning tree
spanning[u][v] = distance[v];
spanning[v][u] = distance[v];
no_of_edges--;
visited[v] = 1;
// updated the distance[] array
for(i = 1; i<n; i++)
    if(visited[i]==0 && cost[i][v]<distance[i])
    {
        ~~v=i;~~
        distance[i] = cost[i][v];
        from[i]=v;
    }
min_cost = min_cost + cost[u][v];
}
return(min_cost);
}
```

Reference :

* Kenneth Berman, Jeroume Paul, Algorithms, Cengage language.

* Thomas H Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford stein, introduction to algorithms, PHT 3rd edition and onwards.

conclusion :

In this termwork we learnt about Prim's Algorithm technique and implementation of Prim Algorithm.

We also learned computing time required for recursive and iterative algorithms.

```
D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>"d:\ISE
Enter no. of vertices:5

Enter the adjacency matrix:
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

spanning tree matrix:

0       2       0       6       0
2       0       3       0       5
0       3       0       0       0
6       0       0       0       0
0       5       0       0       0

Total cost of spanning tree=16
```

Name : John Nixon
USN : 2GI 19 IS 016

Teamwork : 07

Problem Definition:

Implement All-Pairs shortest Paths Problem using Floyd's Algorithm.

Theory:

All pairs shortest path:

The problem: find the shortest path between every pair of vertices of a graph.

A representation: a weight matrix where

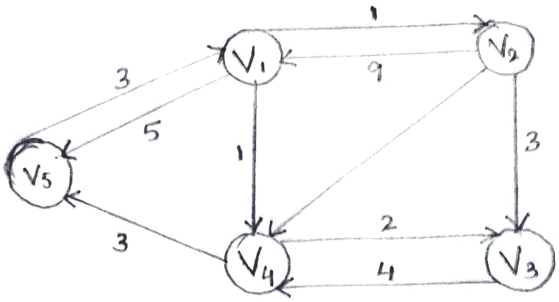
$w(i,j) = 0$ if $i = j$

$w(i,j) = \infty$ if there is no edge between i and j

$w(i,j) =$ "weight of edge".

The weight matrix and the graph

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0



Algorithm:

Floyd's Algorithm using n+1 D matrices

Floyd // computes shortest distance between
// all pairs of nodes and saves P
// to enable finding shortest paths

```
// Initializing 0 array to W[]
D^0 ← W
// Initialize P array to [0]
P ← 0
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
```

$$D^k[i,j] = \min(D^{k-1}[i,j] \text{ or}$$

$$(D^{k-1}[i,k] + D^{k-1}[k,j])$$

Source Code

```
#include <stdio.h>
#include <stdlib.h>
int min (int, int);
void floyds (int p[10][10], int n)
{
        int i, j, k;
        for (k = 1; k <= n; k++)
           for (i = 1; i <= n; i++)
              for (j = 1; j <= n; j++)
                 if (i == j)
                    p[i][j] = 0;
                 else
                    p[i][j] = min (p[i][j], p[i][k] + p[k][j]);
 }
int min ( int a, int b)
{
    if (a<b)
       return (a);
   else
       return(b);
}
int main (int argc, char *argv[])
{
    int p[10][10], w, n, e, u, v;
    int i, j;
    int start = 0, end = 0;
    float sp1, sp2, sp3;
```

```
printf("\n Enter the number of vertices :");
scanf("%d", &n);
printf("\n Enter the number of edges : \n");
scanf("%d", &e);
for( i = 1; i <= n; i++)
{
    for( j = 1; j <= n; j++)
        p[i][j] = 999;
}
for (i = 1; i <= e; i++)
{
    printf("\n Enter the end vertices of edge %d with its weight
                                        \n", i);
    scanf("%d %d %d", &u, &v, &w);
    p[u][v] = w;
}
printf("\n Matrix of input data : \n");
for( i = 1; i <= n; i++)
{
    for( j = 1; j <= n; j++)
        printf("%d \t", p[i][j]);
    printf("\n");
}
floyds(p, n);
printf("\n The shortest paths are : \n");
for( i = 1; i <= n; i++)
```

```
{
        for (j=1 ; j <=n; j ++)
        {
                if (i != j)
                    printf (" %d\t", p[i][j]);
                else
                    printf ("0\t");
        }
        printf ("\n");
}
system ("PAUSE");
return 0;
}
```

References:

* Kenneth Berman, Jerome Paul, Algorithms, Cengage learning.

* Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford Stein, introduction to Algorithms PHI, 2nd edition and onwards.

Conclusion:

In this teamwork, we learnt about all pairs shortest path design technique and weight matrix representation.

We also learned computing time required to recursive and iterative algorithms.

```
D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>"d:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB\TW_7

 Enter the number of vertices:4

 Enter the number of edges:
5

 Enter the end vertices of edge1 with its weight
2 1 2

 Enter the end vertices of edge2 with its weight
3 2 7

 Enter the end vertices of edge3 with its weight
1 3 3

 Enter the end vertices of edge4 with its weight
3 4 1

 Enter the end vertices of edge5 with its weight
4 1 6

 Matrix of input data:
999     999     3       999
2       999     999     999
999     7       999     1
6       999     999     999

 The shortest paths are:
 0       10      3       4
 2       0       5       6
 7       7       0       1
 6       16      9       0
Press any key to continue . . .
```

Name: John Nixon
USN : 2GI 19IS 016

Term Work : 08

Problem Definition:

Implement 0/1 knapsack problem using Dynamic.

Theory:

Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances.

"Programming" here means "planning".

main idea :

- set up a recurrance relating a solution to a larger instance of some smaller instances.
- solve smaller instances once
- record solutions in a table
- extract solution to the inital instance from the table

Algorithm:

```
OP   knapsack (w[1 ... n], V[1 ... n], W)
     var V[0 ... n, 0 ... W], P[1 ... n, 1 ... W]: int
     for i := 0 to W do
         V[0, j] := 0
```

```
for i:=0 to n do
    v[i,0]:=0
for i:=1 to n do
    for j:=1 to w do
        if w[i]≤j and v[i]+V[i-1, j-w[i]] > V[i-
                                        1, j] then
            v[i,j]:=v[i]+v[i-1, j-w[i];
            p[i,j]:=j-w[i]
        else
            v[i,j]:=v[i-1.j];
            p[i,j]:=j
return v[n,w].
```

Source Code.

```
# include < stdio.h>
# include < stdlib.h>
# define MAX 20
int main (int argc, char *argv[])
{
    int weights[MAX], values [MAX], V [MAX]
[MAX], m, i, j , n, val1 , val2 , x[MAX];
    printf ("Enter THE NUMBER OF ITEMS : ");
    scanf ("%d", &n);
    printf ("ENTER WEIGHTS -/n");
    for ( i = 1; i <=n; i++)
        scanf ("%d", &weights [i]);
    printf ("ENTER VALUES -\n");
    for ( i = 1 ; i <=n ; i++)
        scanf ("%d" , &values [i]);
    printf ("ENTER THE KNAPSACK CAPACITY : ");
    scanf ("%d , &m);
    for ( i = 0; i <=n ; i++)
    {
        for (j =0 ; j <= m; j++)
        {
```

```
        if (i==0 || j==0)
            v[i][j]=0;
        else
            v[i][j]=-1;
    }
}
for (i=1; i<=n; i++)
{
    for (j=1; j<=m; j++)
    {
        if (j<weights[i])
            v[i][j]= v[i-1][j];
        else
        {
          val1 = values[i] + v[i-1][j-weights[i]];

          val2 = v[i-1][j];
          if (val1 > val2)
              v[i][j] = val1;
          else
              v[i][j] = val2;
        }
    }
}
```

```
printf ("THE MATRIX IS ....\n");
for (i=0 ; i<=n; i++)
{
    for (j=0; j<=m ; j++)
    {
        printf ("%d \t", v[i][j]);
    }
  printf ("\n");
}
printf ("MAXIMUM PROFIT OBTAINED IS : %d\n", v[n][m]);
i = n;
j = m;
while (i !=0 && j != 0)
{
        if (v[i][j] != v[i-1][j])
        {
            x[i] = 1 ;
            j = j - weights [i];
        }
        i = i -1;
    }
    printf (" ITEM SELECTED ARE --\n");
```

```
printf ("{");
for (i=0; i<=n; i++)
{
    if (x[i]==1)
        printf("%d", i);
}
printf ("}\n");
// system ("PAUSE");
return 0;
}
```

Reference :

* Kenneth Berman, Jerome Paul, Algorithms, Cengage learning.

* Thomas H. Cormen, Charles E. Leiserson Ronal L. Rivest, Clifford stein, introduction to Algorithm PHI, 2nd edition & onwards.

Conclusion:

In this teamwork, we learnt about dynamic programming and implementation of it.

we also learned computing knapsack problem.

```
D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB\TW_8_knapsack(Dynamic Programming)>"d:\ISE - 4
th Sem\SEM-4-main\DAA\DAA LAB\TW_8_knapsack(Dynamic Programming)\TW8.exe"
ENTER THE NUMBER OF ITEMS : 4
ENTER WEIGHTS -
2
1
3
2
ENTER VALUES -
12
10
20
15
ENTER THE KNAPSACK CAPACITY : 5
THE MATRIX IS....
0       0       0       0       0       0
0       0       12      12      12      12
0       10      12      22      22      22
0       10      12      22      30      32
0       10      15      25      30      37
MAXIMUM PR0FIT OBTAINED IS : 37
ITEM SELECTED ARE --
{ 1  2  4 }

D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB\TW_8_knapsack(Dynamic Programming)>
```

Name: John Nixon
USN: 2GI19ISO16

Team Work : 09

Problem Definition:

Find a subset of a given set $S = \{S1, S2, \ldots Sn\}$ of n positive integers whose sum is equal to a given positive integer. For example, if $S = \{1, 2, 5, 6, 8\}$ & $d = 9$ there are two solutions $\{1, 2, 6\}$ & $\{1, 8\}$. A suitable message is to be displayed if the given problem instance doesn't have a solution.

Theory:

Backtracking Technique:

The procedure whereby after determining that a node lead to nothing but dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child.

* Promising : The node can lead to a solution, otherwise, it is called as 'non-promising'.

* Pruning : Check each node whether it is promising, if not, backtracking to the node's parent.

Algorithm:

// Problem : Given n positive weights and a +ve integer W, find all combⁿ of weight that sum to W.

// Inputs : Positive integer n, sorted array a index from 1 to n and a +ve integer W.

// Output : All combinations of the weights that sum to W.

```
void sum_of_subsets (index i, int weight, int total)
{
  if (promising (i))
     if (weight = W)
          cout << include [1] through include [i];
     else {
      include [i+1] = "yes";
      sum_of_subsets (i+1, weight + w[i+1], total - w[i+1]);
      include [i+1] = "no";
      sum_of_subsets (i+1, weight, total - w[i+1]);
    }
  bool promising (index i)
{
  return (weight + total >= W) && (weight = W ||
          weight + w [i+1] <= W);
  }
```

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50
#define TRUE 1
#define FALSE 0
int inc[MAX], w[MAX], sum, n;

int promising (int i, int wt, int total)
{
    return ((( wt + total ) >= sum) && (( wt == sum) || ( wt + w[i+1]
                                                <= sum )));
}

void sumset (int i, int wt, int total)
{
    int j;
    if (promising (i, wt, total))
    {
        if ( wt == sum)
        {
            printf ("\n{\t");
            for (j = 0; j <= i ; j++)
                if (inc[j])
                    printf ("%d\t", w[j]);
            printf ("}\n");
        }
        else
        {
```

```
            inc[i+1] = TRUE;
            sumset(i+1, wt + w[i+1], total - w[i+1]);
            inc[i+1] = FALSE;
            sumset(i+1, wt, total - w[i+1]);
        }
    }
}
int main(int argc, char *argv[])
{
    int i, j, n, temp, total = 0;
    printf("\n Enter how many numbers : \n");
    scanf("%d", &n);
    printf("\n Enter %d numbers to the set : \n", n);

    for(i=0; i<n; i++)
    {
        scanf("%d", &w[i]);
        total += w[i];
    }
    printf("\n Input the sum value to create sub set : \n");
    scanf("%d", &sum);

    for(i=0; i<=n; i++)
        for(j=0; j<n-1; j++)
            if(w[j]>w[j+1])
            {
                temp = w[j];
                w[j] = w[j+1];
                w[j+1] = temp;
            }
```

```
printf ("\n The given %d numbers in ascending order :\n", n);
for (i=0; i <n ; i++)
    printf ("%d \t", w[i]);
if ((total < sum))
    printf ("\n Subset construction is not possible");
else
{
    for (i = 0 ; i < n ; i++)
        inc[i]=0 ;
    printf ("\n The solution using backtracking is :\n");
    sumset (-1, 0, total);
}
system ("PAUSE");
return 0;
}
```

References:

* Kenneth Berman, Jerome Paul, Algorithms, Cengage learning.

* Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford stein, Introduction to Algorithms PHI, 2nd edition and onwards.

Conclusion:

In this termwork, we learnt about Backtracking technique and implementation of backtracking design technique.

We also learned computing time required for recursive and iterative algorithms.

```
D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>"d:\ISE - 4th Sem\SEM-

 Enter how many numbers:
5

 Enter 5 numbers to the set:
5 6 10 11 16

 Input the sum value to create sub set:
21

 The given 5 numbers in ascending order:
5        6        10       11       16
 The solution using backtracking is:

{        5        6        10       }

{        5        16       }

{        10       11       }
Press any key to continue . . .
```

Name: John Nixon.
USN: 2GI19IS016

Team Work : 10

Problem Definition :

Implement N Queen's problem using Back Tracking.

Theory :

* construct the state - space tree.
  - nodes : partial solutions
  - edges : choices in extending partial solution
* Explore the state space tree using depth - first search.
* N - Queens :-
  * The object is to place queens on a chess board in such a way as no queen can capture another one in a single move.
  - Recall that a queen can move horizontal, vertical (or) diagonally an infinite distance.
  - This implies that no two, queen can be on the same row, column (or) diagonal.

Algorithm:

```
Procedure queens (i: index, n);
    var j: index;
Begin
    if promising (i) then
        if i = n then
            write (col[1] through col[n])
        else
            for j := 1 to n do
                col[i+1] := j;
                queens (i+1, n);
            end
        end
    end
End
function promising (i : index) : boolean;
    var k : index;
    Begin
    k := 1;
    promising := true;
    while k < i and promising do
        if cd[i] = cd[k] or abs (col[i] - cd[k]) = i
```

```
            then
            promising := false
        end
        k := k + 1
    end
End
```

Source Code

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

// chess board
int board[10000][10000];

int isItSafe(int i, int j, int N) {

    // Check any queen is present in the
    // current column above the current cell
    // if found to be true return false
    for (int r=0; r<i; ++r)
        if (board[r][j] return 0;

    int r=i, c=j;

    // check any queen is present in the upper
    // left diagonal of the current cell if found
    // to be true return false
    while (r>=0 && c>=0) {
        if (board[r][c]) return 0;
        r--; c--;
    }

    r=i, c=j;
    // Check any queen is present in the upper
    // right diagonal of the current cell if found
    // to be true return false
```

```
while (r >= 0 && c < N) {

    if (board[r][c]) return 0;

    r--; c++;

  }

  // otherwise current cell is safe
  // return true

  return 1;

}

void display(int N) {

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

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

      printf("%d", board[i][j]);

    }

    printf("\n");

  }

  printf("\n\n");

}

int nQueens(int N, int r) {

  // If all the queens are placed in all
  // the rows return true

  if (r == N) return 1;
```

```
    // Traversing through the columns
    for (int c=0; c<N; c++){

        // check current cell in safe
        if (isItSafe (r, c, N)) {

            // Place the queen if the cell is safe
            board [r][c] = 1;
            if (nQueens (N, r+1)) return 1;
            board [r][c] = 0;
        }
    }
    return 0;
}
int main () {
  while (1) {
    int ch;
    printf ("1. Enter the value of N : \n 2. Exit the program\n Enter
                                        the choice : ");
    scanf ("%d", &ch);
    if (ch == 1) {
      int N;
      scanf ("%d", &N);
      for( int i=0; i<N; i++)
        for (int j =0; j<N; ++j)
          board[i][j]=0;
```

```
    If (nQueen(N, 0)) display(N);
}
else if (ch == 2) break;
    else printf("Invalid choice!\n")
}
return 0;
}
```

Reference:

* Kenneth Berman, Jerome Paul. Algorithms, Cengage learning.

* Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, Clifford stein, Introduction to Algorithms PHI, 2nd edition and onwards.

Conclusion:

In this teamwork we learnt about backtracking technique and implementation of backtracking algorithm design technique we also learned computing time required, design algorithms for specific applications using appropriate techniques.

```
PS D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB> & C:
Press 1 to Test the program
Press 2 to Exit the program
Enter your choice :: 1
Enter the number of queens :: 5
[1, 0, 0, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 0, 1]
[0, 1, 0, 0, 0]
[0, 0, 0, 1, 0]

Press 1 to Test the program
Press 2 to Exit the program
Enter your choice :: 1
Enter the number of queens :: 2
[0, 0]
[0, 0]

Press 1 to Test the program
Press 2 to Exit the program
Enter your choice :: 1
Enter the number of queens :: 3
[0, 0, 0]
[0, 0, 0]
[0, 0, 0]

Press 1 to Test the program
Press 2 to Exit the program
Enter your choice :: 2
Exiting ...
PS D:\ISE - 4th Sem\SEM-4-main\DAA\DAA LAB>
```