2GI19CS175

Term Work - 01

Problem Definition

Implement merge sort algorithm to sort a given set of elements, and determine time required to sort the elements.

Theory

Divide & Conquer : The most well known design strategy

* Divide instance of a problem into 2 or more smaller instances
* Solve smaller instances recursively
* Obtain solution to original instance by combining these solns

Algorithm : Pseudocode of Mergesort :

```
Algorithm Mergesort (A, low, high)
{
    if (low < high) then
    {
        // Divide P into algorithmic sub problems
        // Find where to split to set
            mid = [(low+high)/2]
        // Solve subproblem
            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)
{
    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++;
    }
    if (h>mid) then
        for k=j to high do {
            B[i] = A[k];
            i++;
        }
    else
        for k=h to mid do {
            B[i] = A[k]; i++)
        }
}
```

Source code in Python:

```
import random
import time
max = 100000
def getdata(a):
    for i in range(max):
        ele = random.randint(0, 100000)
        a.append(ele)
```

```
def mergesort(a):
    if len(a) > 1:
        mid = len(a)
        left_sub = a[:mid]
        right_sub = a[mid:]

        mergesort(left_sub)
        Mergesort(right_sub)

        i = j = k = 0

        while i < len(left_sub) and j < len(right_sub):
            if left_sub[i] < right_sub[i]:
                array[k] = left_sub[i]
                i += 1
            else:
                array[k] = right_sub[j]
                j += 1
            k += 1

        while i < len(left_sub):
            array[k] = left_sub[i]
            i += 1
            k += 1

        while j < len(right_sub)
            array[k] = right_sub[j]
            j += 1
            k += 1
```

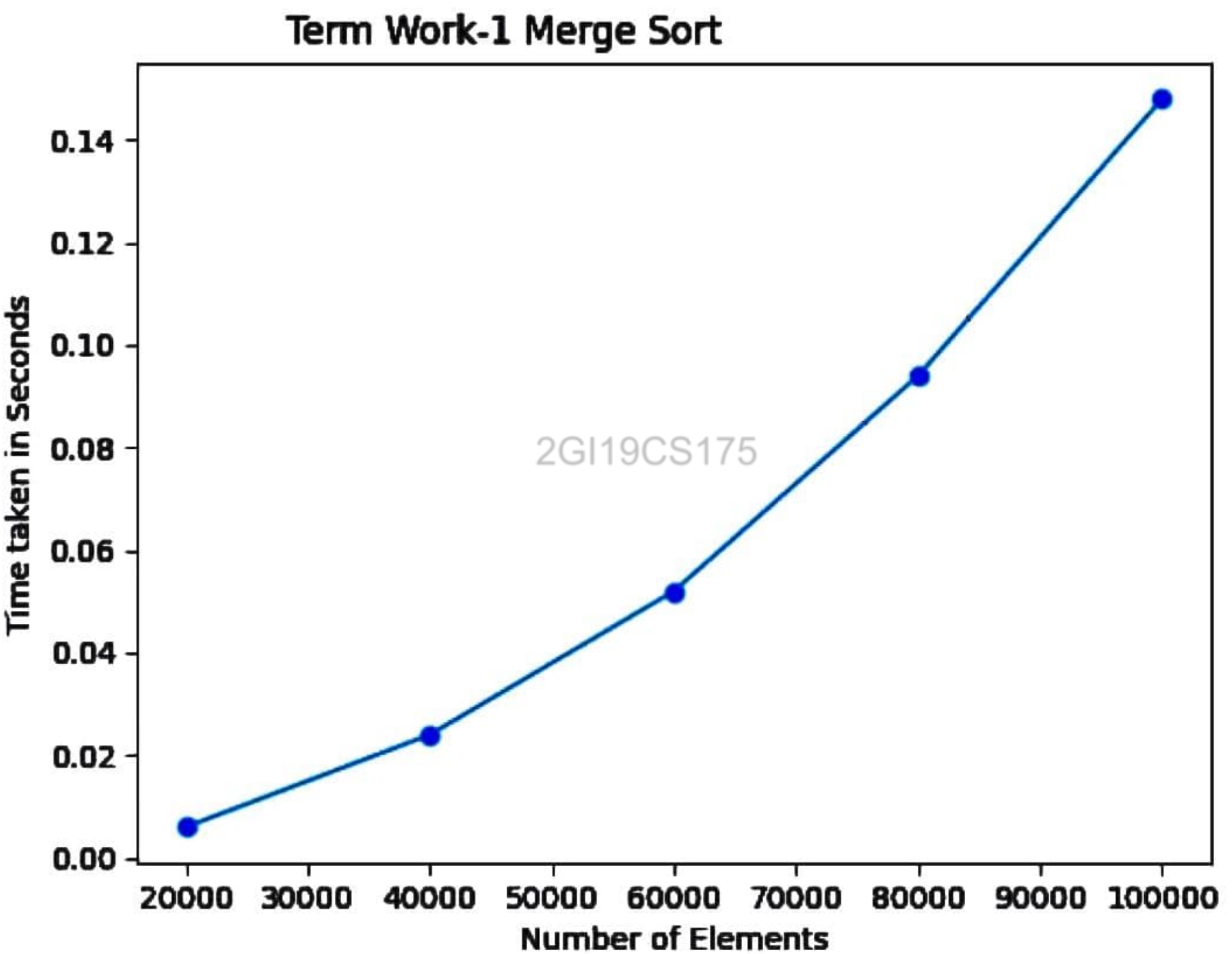
```
array = []
get_data (array)
start = time.time()
mergesort ( array )
end = time.time()
print (f" Time taken :: {end - start} seconds ")
```

References :

* K. Berman, J. Paul, Algorithms, language learning.
* Thomas H Cormen - Introduction to Algorithms 2nd edition.

Conclusion:

In this TW we learnt about divide & conquer technique & implemented merge sort.
We also learnt computing time required for recursive & iterative algorithms.


Term Work-1 Merge Sort
2GI19CS175
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

2GI19CS175

Term Work - 2

Problem Definition:

Implement Quick sort algorithm & determine the time required to sort elements.

Theory

Quicksort Algorithm.

Given an array of n elements

* If an array contains only one element, then return
* Else
  - Pick one element as pivot
  - Partition elements into 2 sub arrays
    * First array that contains elements less than equal to pivot
    * Second array that contains elements greater than pivot
  - Quick sort 2 sub arrays
  - Return result

Algorithm

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

Algorithm Partition (A[l-r])
    p <- A[i]
    i <- l; j <- r+1
repeat
    repeat i <- j+1 until s[i] ≥ p
    repeat j <- j-1 until A[j] ≤ p
    swap (A[i], A[j])
until i ≥ j
swap (A[i], A[j])
swap (A[i], A[j])
return j
```

(8)

Source Code in Python :

```
import random
import time

max = 100000

def getdata(a):
    for i in range(max):
        e = random.randint(0, 80000)
        a.append(a)

def partition (a, beg, end):
    pivot_index = beg
    pivot = a[pivot_index]
    while beg < end:
        while beg < len(a) and a[beg] <= pivot :
            beg += 1

        while a[end] > pivot :
            end -= 1

        if beg < end:
            a[beg], a[end] = a[end], a[beg]

    a[end], a[pivot_index] = a[pivot_index], a[end]

return end
```

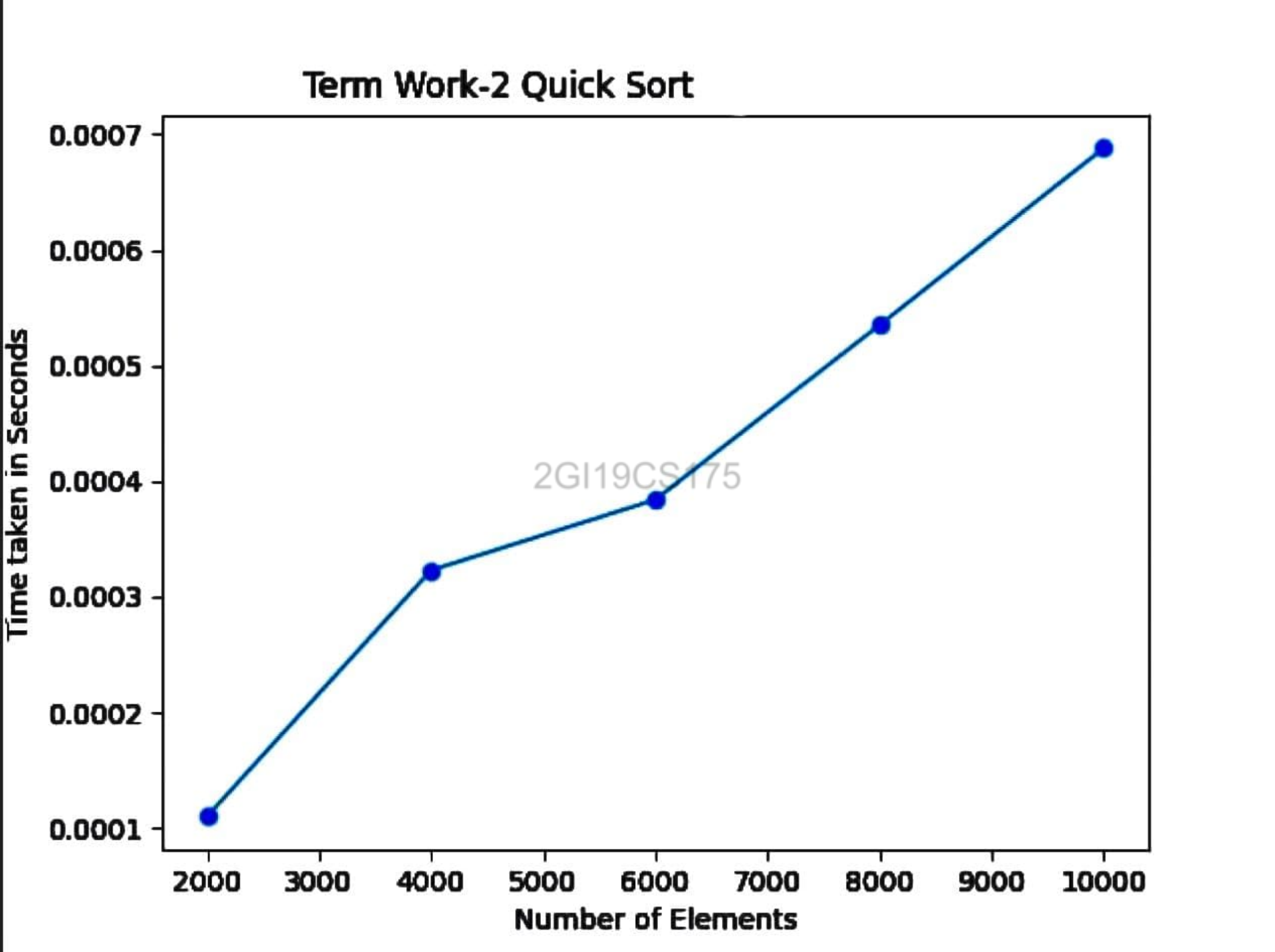
```
def quick_sort (a, beg, end):
    if beg < end:
        partition_index = partition (a, beg, end)
        quicksort (a, beg, partition_index-1)
        quicksort (a, partition_index+1, end)

array = []
get_data (array)
quick_sort (array, 0, len(array)-1)
```

References:

* K. Berman, J. Paul, Algorithms Cengage Learning
* Thomas H Cormen, Charles E, Algorithms 2nd edition.

Conclusion:

In this TW we learn about divide conquer technique & implementation of quicksort algorithm design technique we also learned computing time required for recursive & iterative algorithms.


Term Work-2 Quick Sort
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
Time taken in Seconds
2GI19CS175
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of Elements

```
----------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
```

```
----------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
```

Term Work : 03

Problem Definition

Implement insertion sort algorithm & determine the time required to sort elements

Theory

* To sort array A[0 ... n-1], sort A[0 ... n-2] recursively & then insert A[n-1] in its proper array the sorted A[0...n-2]
* Usually implemented bottom up
* Insertion sort is an example of decrease by a constant type of decrease & conquer technique
* Decrease & conquer
  - Reduce problem instance to smaller instance of same problem
  - Solve smaller instance
  - Extend solution of smaller instance to obtain solution to original instance.

Algorithm :

```
Insertionsort (A[0...n-1])
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
```

Source code in Python :


```
import random
import time
max = 100000
def getdata(a):
    for i in range(0, 80000):
        e = random.randint(0, 80000)
        a.append(e)

def insertionsort(a):
    for i in range(max):
        num = a[i]
        j = i-1
        while j >= 0 and a num < a[j]:
            a[j+1] = a[j]
            j -= 1
        a[j+1] = num
```

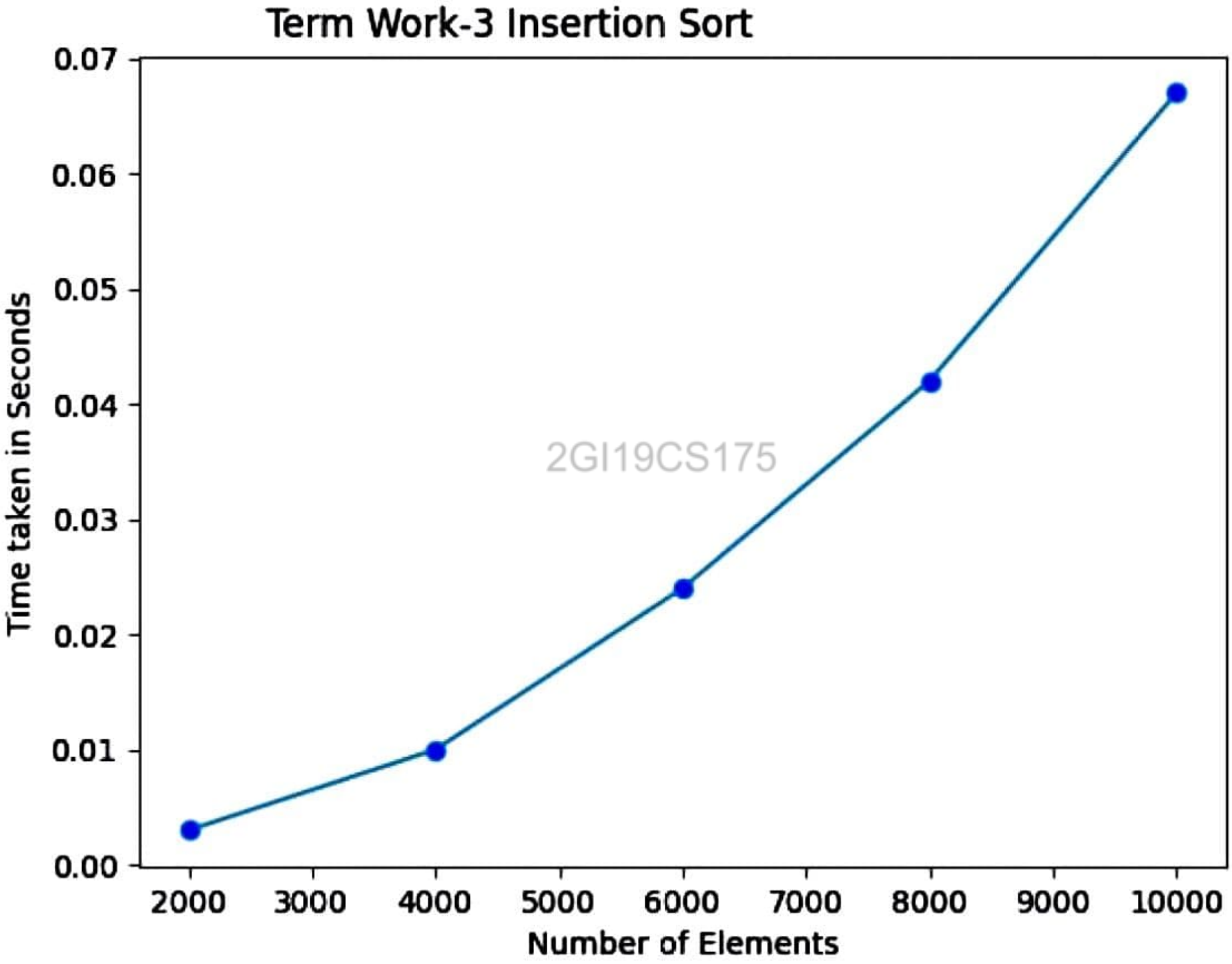
```
a=[]
get_data(a)
s = time.time()
insertion_sort(a)
e = time.time()
print(f"Time taken is : {e-s} seconds")
```

References:

- K. Berman - Algorithms Cengage learning.
- Thomas H - Algorithms 2nd edition & onwards

Conclusion:

In this homework we learnt about decrease & conquer technique & implementation of insertion sort.


Term Work-3 Insertion Sort
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Time taken in Seconds
2GI19CS175
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of Elements

(13)

2GI19CS175

Term Work - 4

Problem Definition

Implementation Heap Sort algorithm & determines the time required to sort elements
Repeat experiment for different values of N.

Theory:

Transform & Conquer: Solves a problem by transformation to:
- A simple instance of same problem
- A different representation of same instance.
- A different problem for which an algorithm is available

Heaps & Heap Sort:

* A heap is a binary tree with keys at its nodes
* It is essentially complete ie all the levels some rightmost keys may be missing.
* The key at each node is ≥ all keys in its children

Algorithms :

```
Heapsort (H[1..n])
    for i ← [n/2] downto 1 do
        k ← i ; v ← H[k]
        heap ← false
        while not heap & 2*k ≤ n do
            j ← 2*k
        if j < n
            if H[j] < H[j+1]
                j ← j+1
            if v ≥ H[j]
                heap ← true
            else
                H[k] ← H[j]
                k ← j
        H[k] ← v
```

Source Code in Python

```
import random
import time
max = 10000
def getdata(a):
    e = random.randint(0, 80000)
    a.append(e)
```

(15)

```
def heapify(a, al, i):
    parent_index = i
    left = 2*i + 1
    right = 2*i + 2

    if left < al and a[parent_index] < a[left]:
        parent_index = left

    if right < al and a[parent_index] < a[right]:
        parent_index = right

    if parent_index != i:
        a[i], a[parent_index] = a[parent_index], a[i]
        heapify(a, al, parent_index)

def heapSort(a):
    al = len(a)

    for i in range(al//2 - 1, -1, -1):
        heapify(a, al, i)

    for i in range(al-1, 0, -1):
        a[i], a[0] = a[0], a[i]
        heapify(a, i, 0)
```

```
a = []
getdata(a)
s = time.time()
heapsort(a)
e = time.time()
print(f"TIME :: {e-s} seconds")
```

References:

- K. Berman - Algorithms Cengage Learning.
- Thomas H - Algorithms 2nd edition & onwards

Conclusion:

In this homework we learnt about decrease & conquer technique & implementation of insertion sort.

```
----------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
```

TermWork : 05

Problem Definition :

From a given vertex in a weighted connected graph, find shortest path 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
- locally optimal

Dijkstra's Algorithm : With a different way of computing numerical labels : Among vertices not already in 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 from source is to v

Algorithms:

```
Dijkstras (s)...
{
    dist[s] ← 0
    for all v∈V-[s]
        do dist[v] ← ∞
    s← ϕ
    Q← v
    while Q ≠ ϕ do
        u ← distance(Q, dist)
    s ← su[v]
    for all v∈ neighbours[v] do
      if dist[v] > dist[v] + w(v,v)
      then d[v] ← d[v] + w(v,v)
  return dist
}
```


Source code in Python:

```
infinity = 99999

def shortest_dist(vertices, shortest_path, path_set):
    shortest = infinity
    for vertex in range(vertices):
        if shortest_path[vertex] < shortest & path_set[vertex] == False
            shortest = shortest_path[vertex]
            shortest_path_index = vertex
    return shortest_path_index
```

```
def Dijkstra's (v, g, s_v):
    shortest_path = [infinity] * vertices
    shortest_path[sv] = 0
    path_set = [False] * vertices
    for i in range(vertices):
        dist_index = shortest_dist(v, shortest_path, path_set)
        path_set[i] = True

        for vertex in range(v):
            shortest_path[vertex] = shortest_paths[dist_index][graph

    print("Vertex \t Distance")
    for node in range(v):
        print(node, "\t", shortest_path[node])
```



```
def main()
    v = int(input("Enter no of vertices:"))
    g = [[infinity for _ in range(v)] for _ in range(v)]
    ch = 1
    while ch == 1
        x, y, w = map(int, input("Enter vertices w/ edge:").split())
    graph[x-1][y-1] = w

    ch = int(input("Press 1 to continue or 0 to exit"))

print(graph)
Dijkstra's(vertices, graph, 0)
```

References:

* K. Berman - Algorithms Cengage Learning
* Thomas H, 2nd edition onwards Algorithms.

Conclusion:

In this TW, coc learnt about Greedy Technique & implementation of Dijkstra's algorithm.

```
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
```

2GI19CS175

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 a undirected graph G is a subgraph of G that is a tree containing all vertices of G.

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

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

Algorithm

Prim(G)
Tree of G
$V_T \leftarrow \{v_0\}$
$E_T \leftarrow \emptyset$
for $i \leftarrow 1$ to $|V|-1$ do
find a minimum weight edge $e^* = (v^*, u^*)$ among all the edges (u,v) such that v is in V_T & u in $V-V_T$.
$V_T \leftarrow V_T \cup \{u^*\}$
$E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Source Code in C

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

```
    total = prims();
    for(i=0; i<n; i++){
        for(j=0; j<n; j++)
            printf("%d\t", spanning[i][j]);
    }
    printf("\n\n Total cost of spanning tree = %d", total);
    return 0;
}

int prims(){
    int cost[MAX][MAX];
    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
        }
    }
    for(i=0; i<n; i++){
        distance[i] = cost[0][i];
        from[i]=0;
        visited[i]= 0;
    }
    min_cost = 0;
    no/off edges = n-1;
```

```
while (edges > 0){
    min_distance = infinity;
    for (i=0; i<n; i++){
        if (visited[i] == 0 && distance[i] < min_distance) {
            v = i;
            min_distance = distance[i];
        }
    a = from[v]

    spanning[0][v] = distance[v];
    spanning[0][v] = distance[v];
     edges --;
        visited[v] = 1
    for (i=0; i<n; i++){
        if (visited[i] == 0 && cost[i][v] < distance[i]) {
            distance[i] = cost[i][v];
            from[i] = v;
    min_cost = min_cost + cost
  }
  return (min_cost);
}
```

```
while (edges > 0){
    min_distance = infinity;
    for (i=0; i<n; i++){
        if (visited[i] == 0 && distance[i] < min_distance) {
            v = i;
            min_distance = distance[i];
        }
    a = from[v]

    spanning[v][v] = distance[v];
    spanning[0][v] = distance[v];
    edges --;
        visited[v] = 1
    for (i=0; i<n; i++){
        if (visited[i] == 0 && cost[i][v] < distance[i]) {
            distance[i] = cost[i][v];
            from[i] = v;
    min_cost = min_cost + cost
    }
    return (min_cost);
}
```

References

* K. Berman - Algorithms, Cengage language
* Thomas H, PHT Introduction to algorithms 2nd edition.

Conclusion

In this termwork we learnt about Prim's algorithm & technique & implementation of Prim's algorithm.

```
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
```

Term Work : 07

Problem Definition

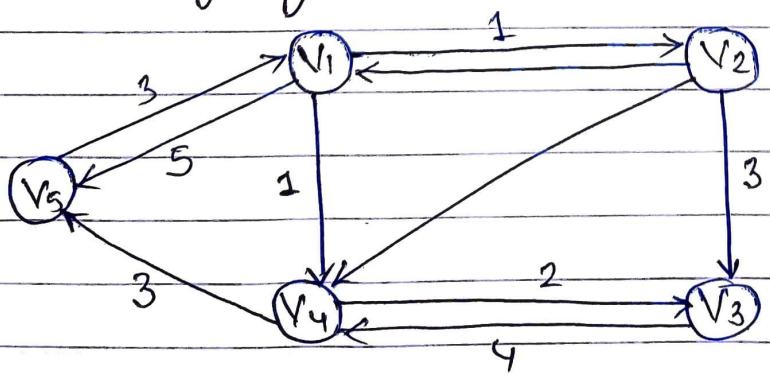
Implement all pairs shortest path problem using Floyd's algorithm.

Theory

All pairs shortest path :
A representation : a weight matrix where

$w(i,j) = 0$ if $i = j$
$w(i,j) = \infty$ if there is no edge b/w i & j
$w(i,j)$ = "weight of edge"

* The weight graph



Algorithm:

Floyd's algorithm using n+1 matrices

// Initializing array to W[]

$D^0 \leftarrow W$

$P \leftarrow 0$

for $k \leftarrow 1$ to n do

  for $i \leftarrow 1$ to n do

    for $j \leftarrow 1$ to n

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

Source Code in Python


```
infinity = 999

def solve (graph, vertices):
    distance = list (map (lambda i: list (map (lambda j: j, i)), graph))
    for m in range (vertices):
        for n in range (vertices):
            for o in range (vertices):
                distance [n][o] = min (distance [n][o], distance (n)(m]
                                       + distance [m][o] )

display (distance, vertices)
```

```
def display(distance, vertices):
    for i in range(vertices):
        for j in range(vertices):
            if distance[i][j] == infinity
                distance[i][j] = "inf"

    for rows in distance:
        print(rows)

def main():
    vertices = int(input("Enter the number of vertices :: "))
    graph = [[infinity for _ in range(vertices)] for _ in range(vertices)]
    edges = int(input("Enter the no. of edges :: "))

    for i in range(edges):
        x, y, w = map(int, input("Enter vertices of edges ").split())
        graph[x-1][y-1] = w
        print(graph)
    solve(graph, vertices)

if __name__ == "__main__":
    main()
```

References :

* K. Berman, Algorithms, Cengage Learning
* Thomas H, Introduction to algorithm 2nd edition.

Conclusion :

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

We also learned computing time required to recursive & iterative algorithm.

```
 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 . . .
```

Term Work : 08

Problem Definition

Implement 0/1 Knapsack problem using Dynamic programming

Theory:

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

Main idea:

- Set up a recurrence relating a solution to a larger instance of some smaller instances.
- Solve smaller instances once
- Record solutions in a table
- Extract solution to the initial instance from 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 & 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 in C :

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
int main (){
int i, j, m, n, val1, val2, x[MAX], w[MAX], v[MAX];

printf("\n Enter the weights & values ");
for(i=0; i<n; i++){
        scanf("%d %d", &weights[i], &values[i]);
printf("\n Enter 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<=n; 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("Maximum profit obtained is :", 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-weight[i];
    }
    i=i-1;
}
for(i=0; i<=n; i++) {
    if (x[i]==1)
        printf("%d", i);
}

return 0;

}
```

Reference:

* K. Berman - Algorithms - Cengage Learning
* Thomas - H, Introduction to Algorithms PHI, 2nd edition onwards

Conclusion:

In this homework, we learnt about dynamic programming & implementation of it.
We also learned computing knapsack problem

```
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 }
```

TermWork : 09

Problem Definition

Find a subset of a given set $S = \{s1, s2, s3, \cdots, sn\}$ of n +ve integers whose sum is equal to a given positive integer.

Theory :

Backtracking Technique : The procedure whereby after determining that a node lead to nothing but dead ends, we go back to node's parent & proceed with the search on 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 back tracking to node's parent

Algorithm:

```
void sum_of_subsets (index, i, weight, total) {
    if promising (i)
        if weight == W
            display include[i] through include[i];
        else
            include[i+1] ← "yes"
            sum of subset ← (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 <conio.h>
#define MAX TRUE 1
#define MAX FALSE 0

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", 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 i, j, n, temp, total = 0;

for (i=0; i<n; i++) {
    scanf("%d", &w[i]);
    total += w[i];
}
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;
        }
for (i=0; i<n; i++)
    printf("%d", w[i]);

if ((total< sum))
    printf("\n Subset construction not possible");
else {
    for (i=0; i<n; i++)
        inc[i] =0
    subset(-1, 0, total);
}
return 0;
```

References:

* K. Berman; Cengage Learning - Algorithms

* Thomas H, Introduction to Algorithms - 2nd edition

Conclusion:

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

```
 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 . . .
```

TermWork: 10

Problem Definition:

Implementation N Queens problem using Back Tracking.

Theary:

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

Algorithm :

```
Procedure queen (i, index, n)
        var j ← index
Begin
    if promising (i) then
            if i=n then
                write ( col[i] 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 :

```
def isAttack (i,j, board):
    if board[i][k] == 1 or board[k][j] == 1:
            return True

    for k in range (len(board)):
        for l in range (len( board)):
            if k+1 == i+j or k-1 == i-j :
                if board [k][l] == 1:
                    return True
    return False

def N_queen (n, board):
    if n == 0:
        return True
    for i in range (len(board)):
        for j in range (len(board)):
            if (not (isAttack (i,j, board))) and (board [i][j] != 1):
                board [i][j] = 1
                if N_queen (n-1, board) == True
                    return True.
                board [i][j] = 0
    return False.
```

```
def main():
    ch = 1
    while ch == 1:
        ch = int(input())
        if ch == 1:
            N = int(input())
            board = [[0]*N for _ in range(N)]

            N_queen(N, board)

            for i in board:
                print(i)
            print()

        elif ch == 2:
            print("Exiting !!")

        else:
            print("Invalid choice")

if __name__ == "__main__":
    main()
```

References :

* K. Berman - Algorithms, Cengage Learning

* Thomas H Corner, Introduction to Algorithms PHI 2nd edition

Conclusion :

* In this termwork we learnt about backtracking technique & implementation of back tracking algorithm design technique

* we also learned computing time required, design algorithm for specific applications using appropriate techniques

```
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 ...
```