Experiment 10 String Matching

Name: Kuldeep Suresh Choudhary
Sapid: 60003220294
Batch: I2 – 1
Rollno: I080

Aim: Implementation of String Matching Algorithms.

Theory:

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm is declared as "this is the method to find a place where one is several strings are found within the larger string."

Given a text array, T [1.....n], of n character and a pattern array, P [1.....m], of m characters. The problems are to find an integer s, called **valid shift** where $0 \leq s < n-m$ and T [s+1.....s+m] = P [1.....m]. In other words, to find even if P in T, i.e., where P is a substring of T. The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}. Given a string T [1.....n], the **substrings** are represented as T [i.....j] for some $0 \leq i \leq j \leq n-1$, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m).
The **proper substring** of string T [1.....n] is T [1.....j] for some $0 < i \leq j \leq n-1$. That is, we must have either i>0 or j < m-1.

There are different types of method is used to finding the string

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm

The Rabin-Karp-Algorithm

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

RABIN-KARP-MATCHER (T, P, d, q)

1. n ← length [T]

2. m ← length [P]
3. h ← d^{m-1} mod q
4. p ← 0
5. t_0 ← 0
6. for i ← 1 to m
7. do p ← (dp + P[i]) mod q
8. t_0 ← (dt_0+T [i]) mod q
9. for s ← 0 to n-m
10. do if p = t_s
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then t_{s+1} ← (d (t_s-T [s+1]h)+T [s+m+1])mod q

Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π:

COMPUTE- PREFIX- FUNCTION (P)

1. m ←length [P] //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0 // numbers of characters matched
5. for i ← 1 to n // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q] // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1 // next character matches 10. If q = m
   // is all of p matched?
11. then print "Pattern occurs with shift" i - m

12. q ← Π [q] // look for the next match

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

Code:

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

int main()
{
    char S[100], P[100];
    printf("Enter the string: ");
    scanf("%s", S);
    printf("Enter the pattern: ");
    scanf("%s", P);

    int pieSize = strlen(P);
    int *pie = (int *)malloc(pieSize * sizeof(int));
    if (pie == NULL)
    {
        printf("Memory allocation failed. Exiting...\n");
        return 1;
    }

    pie[0] = 0;
    for (int i = 1; i < pieSize; i++)
    {
        int l = pie[i - 1];
        while (l > 0 && P[i] != P[l])
        {
            l = pie[l - 1];
        }
        if (P[i] == P[l])
            l++;
        pie[i] = l;
    }

    printf("Prefix table: ");
    for (int i = 0; i < pieSize; i++)
        printf("%d ", pie[i]);
    printf("\n");
```

```
    int i = 0, j = 0;
    while (i < strlen(S))
    {
        if (S[i] == P[j])
        {
            i++;
            j++;
            if (j == pieSize)
            {
                printf("Given Pattern is present in the given string\n");
                free(pie);
                return 0;
            }
        }
        else
        {
            if (j != 0)
                j = pie[j - 1];
            else
                i++;
        }
    }

    printf("Given Pattern is not present in the given string\n");
    free(pie);
    return 0;
}
```

Output:

```
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> gcc .\Rabin-Karp.c
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> .\a.exe
Enter the string: Mamlayalayalam
Enter the pattern: alaya
Prefix table: 0 0 1 0 1
Given Pattern is present in the given string
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
```

Lab Assignment to Complete:

Apply string matching algorithms on the following:

String: Mamlayalayalam

Pattern: alaya

Department of Information Technology

Experiment 9

(Branch and

Bound)

Name: Kuldeep Suresh Choudhary

SapID: 60003220294 **Batch:** I2 – 1

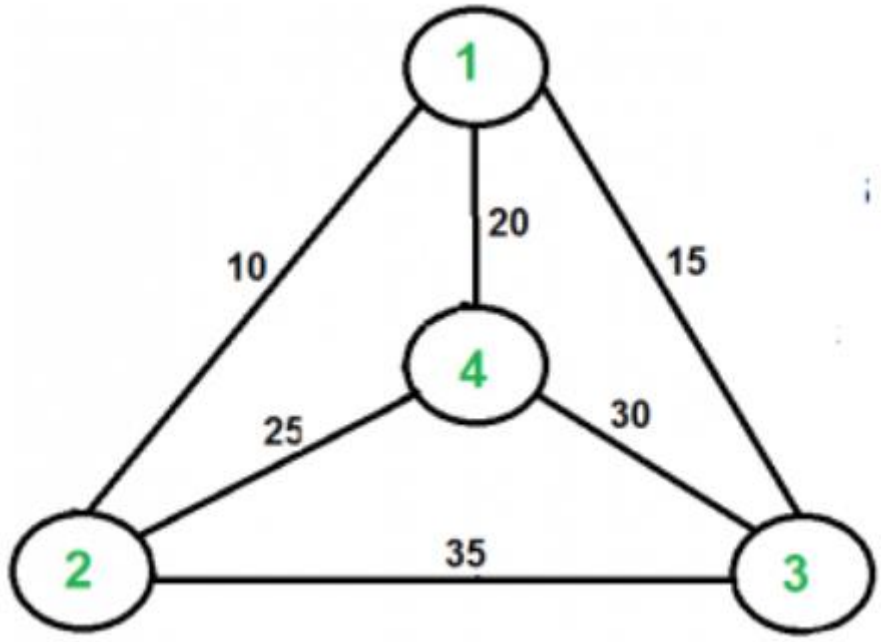
Rollno: I080

Aim: In a world of interconnected cities, a traveling salesperson is tasked with visiting a set of cities and returning to the starting city, aiming to minimize the total distance traveled. Each city is connected to some or all other cities by direct routes, and the distances between cities are known. The goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.

Theory:

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the original city.

Example:




```
Output of Given Graph:
minimum weight Hamiltonian Cycle :
10 + 25 + 30 + 15 := 80
```

Department of Information Technology

Algorithm:

1. A row (column) is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.
2. A matrix is said to be reduced iff every row and column is reduced.
3. Let R be a node in the tree and A(R) its reduced matrix.
4. Let S be the child of R. Cost of S will be computed as follows –
   1.Set all entries in row i and column j of A to ¥.
   2.Set A(j,1) to ¥ // to prevent use of edge <j, 1>
   3. Reduce all rows & columns in the resulting matrix except for rows & columns having infinity. Let the matrix be B.
5. Let r be the reduced cost or total amount subtracted in step 3, then
   c^ (S) = c^ (R) + A(i, j) + r
   The cost of the path is exactly reduced by r.

Code:

```
#include <stdio.h>
#include <conio.h>
int a[10][10], visited[10], n, cost = 0;
void get()
{
   int i, j;
   printf("Enter No. of Cities: ");
   scanf("%d", &n);
   printf("\nEnter Cost Matrix: \n");
   for (i = 0; i < n; i++)
   {
      printf("\n Enter Elements of Row : %d\n", i + 1);
      for (j = 0; j < n; j++)
         scanf("%d", &a[i][j]);
      visited[i] = 0;
   }
   printf("\n\nThe cost list is:\n\n");
   for (i = 0; i < n; i++)
   {
      printf("\n\n");
      for (j = 0; j < n; j++)
         printf("\t % d", a[i][j]);
   }
}
void mincost(int city)
{
   int i, ncity;
   visited[city] = 1;
   printf("%d –>", city + 1);
   ncity = least(city);
   if (ncity == 999)
   {
```

```
        ncity = 0;
        printf("%d", ncity + 1);
        cost += a[city][ncity];
        return;
    }
    mincost(ncity);
}
int least(int c)
{
    int i, nc = 999;
    int min = 999, kmin;
    for (i = 0; i < n; i++)
    {
        if ((a[c][i] != 0) && (visited[i] == 0))
            if (a[c][i] < min)
            {
                min = a[i][0] + a[c][i];
                kmin = a[c][i];
                nc = i;
            }
    }
    if (min != 999)
        cost += kmin;
    return nc;
}
void put()
{
    printf("\n\nMinimum cost:");
    printf("%d", cost);
}
void main()
{
    get();
    printf("\n\nThe Path is:\n\n");
    mincost(0);
    put();
}
```

Output:

```
The Path is:

1 ->6 ->3 ->4 ->5 ->2 ->1

Minimum cost:46
```

Department of Information Technology

Lab Assignment to Complete:

Consider the instance of TSP defined by cost matrix C as given below.

$$\begin{pmatrix} \infty & 8 & 5 & 10 & 4 \\ 5 & \infty & 11 & 6 & 9 \\ 3 & 7 & \infty & 12 & 15 \\ 8 & 7 & 12 & \infty & 19 \\ 1 & 3 & 14 & 17 & \infty \end{pmatrix}$$

Find an optimal tour of the graph for Travelling Salesperson by generating state space tree using LCBB.

Department of Information Technology

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

Experiment 3

(Greedy Algorithm)

Name: Kuldeep Suresh Choudhary **Roll_no:** I080

Sap-id: 60003220294 **Batch:** I2 – 1

Date: 23-02-24

Aim: Implementation of activity selection problem using greedy approach.

Theory:

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame. Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

Input Data for the Algorithm:

- act[] array containing all the activities.
- s[] array containing the starting time of all the activities.
- f[] array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- sol[] array referring to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem:

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array act[] and add it to sol[] array.

Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.

Step 5: Select the next activity in act[] array.

Step 6: Print the sol[] array.

Algorithm:

Activity-Selection(Activity, start, finish) Sort
Activity by finish times stored in finish

```
    Selected =
    {Activity[1]} n =
    Activity.length j = 1
    for i = 2 to n:
       if start[i] ≥ finish[j]:
          Selected = Selected U {Activity[i]}
          j = i
return Selected
```

Complexity:

Time complexity: O(nlogn)

Code:

```c
#include <stdio.h>

int activity[100], start[100], finish[100], solution[100];

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int n, i, j;
    printf("Enter the total number of activities: ");
    scanf("%d", &n);
    printf("Enter the activities: \n");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &activity[i]);
    }

    printf("Enter the start time: \n");
```

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

printf("Enter the finish time: \n");
for (i = 0; i < n; i++)
{
    scanf("%d", &finish[i]);
}

// Sorting activities according to their finish time
for (i = 0; i < n - 1; i++)
{
    for (j = 0; j < n - i - 1; j++)
    {
        if (finish[j] > finish[j + 1])
        {
            swap(&finish[j], &finish[j + 1]);
            swap(&start[j], &start[j + 1]);
            swap(&activity[j], &activity[j + 1]);
        }
    }
}

printf("Sorted activities according to their finish time.\n");
printf("start time(s)\tfinish time(f)\tactivity name\n");
for (i = 0; i < n; i++)
{
    printf("%d\t\t%d\t\t%d\n", start[i], finish[i], activity[i]);
}

int k = 0;
solution[k] = activity[0];
```

```
    for (i = 1; i < n; i++)
    {
        if (start[i] >= finish[k])
        {
            k++;
            solution[k] = activity[i];
        }
    }
    printf("Hence the schedule is: \n");
    for (i = 0; i < k; i++)
    {
        printf("%d --> ", solution[i]);
    }
    printf("%d\n", solution[k]);
    return 0;
}
```

OUTPUT:

```
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> gcc ActivitySelection.c
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> .\a.exe
Enter the total number of activities: 6
Enter the activities:
1
2
3
4
5
6
Enter the start time:
5
1
3
0
5
8
Enter the finish time:
9
2
4
6
7
9
```

Department of Information Technology

```
Sorted activities according to their finish time.
start time(s)   finish time(f)  activity name
1               2               2
3               4               3
0               6               4
5               7               5
5               9               1
8               9               6
Hence the schedule is:
2 --> 3 --> 5 --> 6
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
```

Department of Information Technology Lab

Assignment:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

Conlusion: In this experiment, We have implemented activity selection problem using Greedy approach. It efficiently finds the optimal solution for scheduling non – conflicting activities by selecting those with the earliest finish time.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

Experiment 9

(Backtracking)

Name: Kuldeep Suresh Choudhary **Batch:** I2 – 1

SapID: 60003220294 **Roll_no:** I080

Aim: Implementation of 8 queen problem.

Theory:

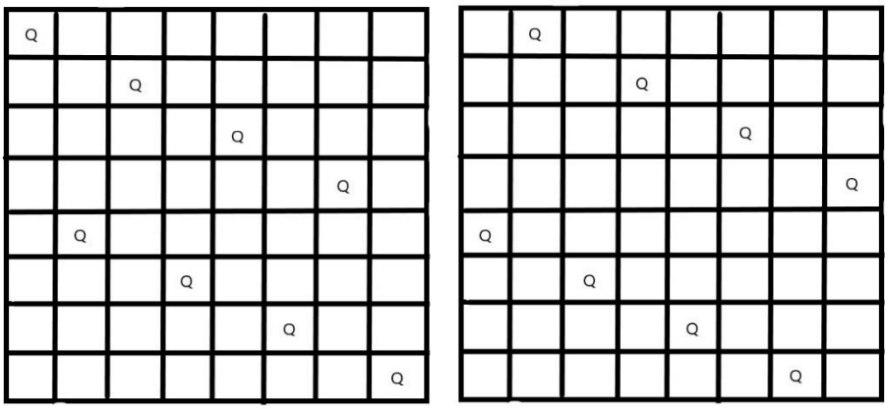
You are given an 8x8 chessboard, find a way to place 8 queens such that no queen can attack any other queen on the chessboard. A queen can only be attacked if it lies on the same row, orsame column, or the same diagonal of any other queen. Print all the possible configurations.

To solve this problem, we will make use of the **Backtracking algorithm**. The backtracking algorithm, in general checks all possible configurations and test whether the required result is obtained or not. For thr given problem, we will explore all possible positions the queens can be relatively placed at. The solution will be correct when the number of placed queens = 8.

Input Format - the number 8, which does not need to be read, but we will take an input numberfor the sake of generalization of the algorithm to an NxN chessboard.

Output Format - all matrices that constitute the possible solutions will contain the numbers 0(for empty cell) and 1(for a cell where queen is placed). Hence, the output is a set of binary matrices.

Example:



Algorithm:

1. Begin from the leftmost column
2. If all the queens are placed, return true/ print the matrix
3. Check for all rows in the current column
    a) if queen placed safely, mark row and column; and recursively check if weapproach in the current configuration, do we obtain a solution or not

b) if placing yields a solution, return true
c) if placing does not yield a solution, unmark and try other rows

4. If all rows tried and solution not obtained, return false and backtrack

Pseudocode:

Algorithm N_QUEEN (k, n)
// Description : To find the solution of n x n queen problem using backtracking
// Input :
n: Number of queen
k: Number of the queen being processed currently, initially set to 1.

// Output : n x 1 Solution tuplefor i ← 1 to n do
if PLACE(k , i) thenx[k] ← i
if k == n then print X[1…n] else
N_QUEEN(k + 1, n)
endend end

Function PLACE(k, i)
// k is the number of queen being processed
// i is the number of columns

for j ← 1 to k – 1 do
if x[j] == i OR ((abs(x[j]) - i) == abs(j - k)) thenreturn false
endend
return true

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int board[20], count;

int main()
{
    int n, i, j;
    void queen(int row, int n);

    printf("\n***** N-Queens using Backtracking *****");
    printf("\nEnter number of Queens : ");
    scanf("%d", &n);
    queen(1, n);
    return 0;
}

void print(int n)
{
    int i, j;
```

```
    printf("\n\nSolution %d : \n\n", ++count);

    for (i = 1; i <= n; ++i)
        printf("\t%d", i);

    for (i = 1; i <= n; ++i)
    {
        printf("\n\n%d", i);
        for (j = 1; j <= n; ++j)
        {
            if (board[i] == j)
                printf("\tQ");
            else
                printf("\t-");
        }
    }
}

int place(int row, int column)
{
    int i;
    for (i = 1; i <= row - 1; ++i)
    {
        if (board[i] == column)
            return 0;
        else if (abs(board[i] - column) == abs(i - row))
            return 0;
    }
    return 1;
}

void queen(int row, int n)
{
    int column;
    for (column = 1; column <= n; ++column)
    {
        if (place(row, column))
        {
            board[row] = column;
            if (row == n)
                print(n);
            else
                queen(row + 1, n);
        }
    }
}
```

Output:

```
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> gcc nQueen.c
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> .\a.exe

 ***** N-Queens using Backtracking *****
 Enter number of Queens : 4


 Solution 1 :

        1       2       3       4

 1      -       Q       -       -

 2      -       -       -       Q

 3      Q       -       -       -

 4      -       -       Q       -

 Solution 2 :

        1       2       3       4

 1      -       -       Q       -

 2      Q       -       -       -

 3      -       -       -       Q

 4      -       Q       -       -
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
```

Time Complexity: O(N!)

Lab Assignment to Complete:

Using the above code and find a possible solution for N-queen problem, where N=4, 6, 8

Department of Information Technology

NAME: Kuldeep Choudhary **SAPID: 60003220294**

BATCH: I2 – 1 **DATE: 6/04/23**

Experiment 4

(Greedy Algorithm)

Aim: Implementation of Prims's & Kruskal's method.

Prim's algorithm:

Theory:

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph?

Algorithm:

Step 1:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step 2:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step 3:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

Example:

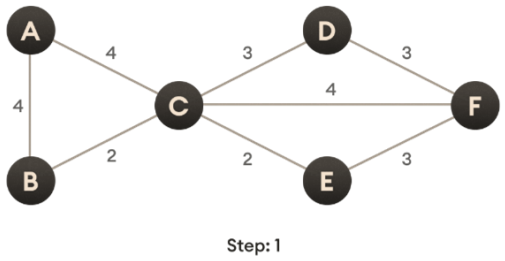
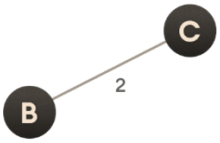


Figure 1. Start with a weighted graph

Department of Information Technology

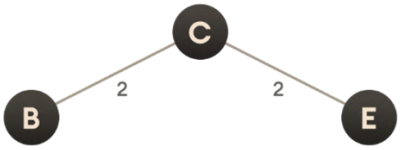
Step: 2

Figure 2.Choose a vertex



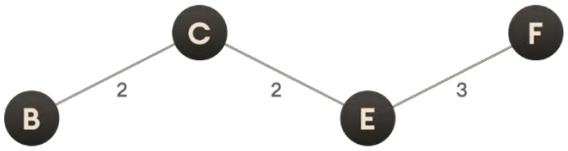
Step: 3

Figure 3.Choose the shortest edge from this vertex and add it



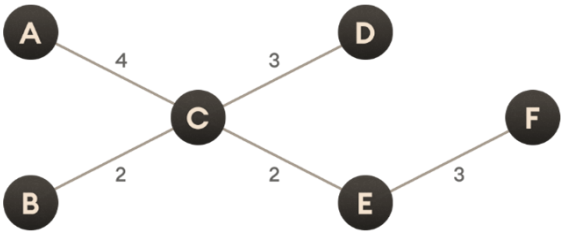
Step: 4

Figure 4.Choose the nearest vertex not yet in the solution



Step: 5

Figure 5.Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Figure 6.Repeat until you have a spanning tree

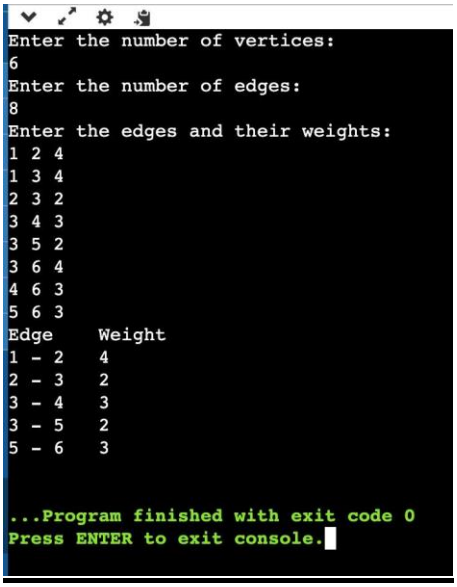
Department of Information Technology

Code:

```
#include <bits/stdc++.h>
using namespace std;

int minKey(int key[], bool mstSet[], int V)
{
   int min = INT_MAX, min_index;

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

   return min_index;
}

void printMST(int parent[], int graph[100][100], int V)
{
   cout << "Edge \tWeight\n";
   for (int i = 1; i < V; i++)
      cout << parent[i] + 1 << " - " << i + 1 << " \t" << graph[i][parent[i]] << "\n";
}

void primMST(int graph[100][100], int V)
{
   int parent[V];
   int key[V];
   bool mstSet[V];

   for (int i = 0; i < V; i++)
   {
      parent[i] = -1;
      key[i] = INT_MAX;
      mstSet[i] = false;
   }

   key[0] = 0;
   parent[0] = -1;

   for (int count = 0; count < V - 1; count++)
   {
      int u = minKey(key, mstSet, V);

      mstSet[u] = true;

      for (int v = 0; v < V; v++)
         if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
   }

   printMST(parent, graph, V);
}
```

```
int main()
{
   int V, E;
   cout << "Enter the number of vertices:\n";
   cin >> V;
   cout << "Enter the number of edges:\n";
   cin >> E;

   int graph[100][100];
   memset(graph, 0, sizeof(graph));

   cout << "Enter the edges and their weights:\n";
   for (int i = 0; i < E; i++)
   {
      int u, v, w;
      cin >> u >> v >> w;
      graph[u - 1][v - 1] = graph[v - 1][u - 1] = w;
   }

   primMST(graph, V);

   return 0;
}
```

Output:



Complexity:

The time complexity of Prim's algorithm is O(E log V).

Kruskal's algorithm:

Theory:

Department of Information Technology

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph?

Algorithm:

Step 1:

- Sort all the edges from low weight to high weight.

Step 2:

- Take the edge with the lowest weight and use it to connect the vertices of graph.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step 3:

- Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

Example:

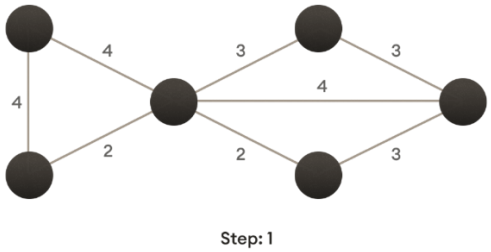


Figure 7. Start with a weighted graph

Department of Information Technology

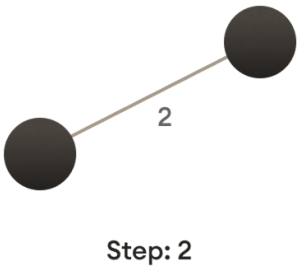


Figure 8. Choose the edge with the least weight, if there are more than 1, choose anyone

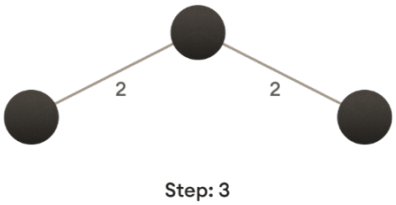


Figure 9. Choose the next shortest edge and add it

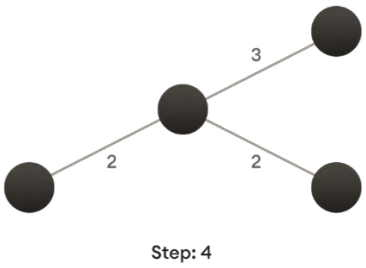


Figure 10. Choose the next shortest edge that doesn't create a cycle and add it

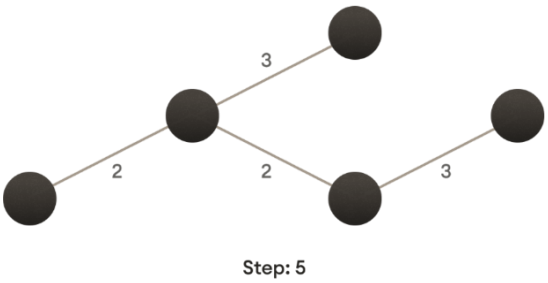
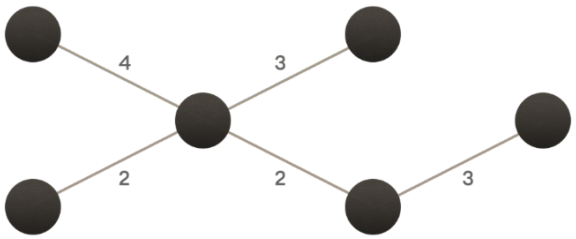


Figure 11. Choose the next shortest edge that doesn't create a cycle and add it

Department of Information Technology



Step: 6

Code:

```
#include <bits/stdc++.h>

using namespace std;

struct Edge
{
   int u, v, weight;
   bool operator<(const Edge &other) const
   {
      return weight < other.weight;
   }
};

int find(vector<int> &parent, int x)
{
   if (parent[x] == -1)
      return x;
   return find(parent, parent[x]);
}

void unite(vector<int> &parent, int x, int y)
{
   int x_root = find(parent, x);
   int y_root = find(parent, y);
   if (x_root != y_root)
      parent[x_root] = y_root;
}

vector<Edge> kruskalMST(vector<Edge> &edges, int n)
{
   vector<Edge> mst;
   vector<int> parent(n, -1);

   sort(edges.begin(), edges.end());

   for (const Edge &edge : edges)
   {
```

```
        int x_root = find(parent, edge.u);
        int y_root = find(parent, edge.v);

        if (x_root != y_root)
        {
            mst.push_back(edge);
            unite(parent, x_root, y_root);
        }
    }

    return mst;
}

int main()
{
    int n, m;
    cout << "Enter the number of vertices and edges: ";
    cin >> n >> m;

    vector<Edge> edges(m);
    cout << "Enter the edges (u, v, weight):\n";
    for (int i = 0; i < m; i++)
    {
        cin >> edges[i].u >> edges[i].v >> edges[i].weight;
    }

    vector<Edge> mst = kruskalMST(edges, n);

    cout << "Minimum Spanning Tree:\n";
    for (const Edge &edge : mst)
    {
        cout << edge.u << " -- " << edge.v << " : " << edge.weight << "\n";
    }

    return 0;
}
```

Output:

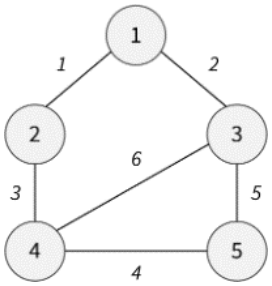
```
Enter the number of vertices and edges: 6 8
Enter the edges (u, v, weight):
1 2 4
1 3 4
2 3 2
3 4 3
3 5 2
3 6 4
4 6 3
5 6 3
Minimum Spanning Tree:
2 -- 3 : 2
3 -- 5 : 2
3 -- 4 : 3
4 -- 6 : 3
1 -- 2 : 4


...Program finished with exit code 0
Press ENTER to exit console.
```

Complexity:

The time complexity Of Kruskal's Algorithm is: O(E log E).

Lab Assignment:

Write a C program to implement Prim's and Kruskal's Algorithm using greedy algorithm for the following graph.



Department of Information Technology
Experiment 2

(Greedy Algorithm)

Name: Kuldeep Suresh Choudhary **Roll_no:** I080

Sap-id: 60003220294 **Batch:** I2 – 1

Date: 23-02-24

Aim: Implementation of fractional Knapsack using greedy algorithm.

Theory:

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications:

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are n items in the store
- Weight of i^{th} item $w_i>0$
- Profit for i^{th} item $p_i>0$ and
- Capacity of the Knapsack is W
-

Pseudocode:

```
Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)
for i = 1 to n do x[i] = 0
weight = 0 for i = 1 to n if
weight + w[i] ≤ W then
x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
```

weight = W break
return x

Complexity:

Time Complexity: O(n logn).

Example:

Problem: Consider the following instances of the fractional knapsack problem: n = 3, M = 20, V = (24, 25, 15) and W = (18, 15, 20) find the feasible solutions.

Solution:

Arrange items by decreasing order of profit density. Assume that items are labeled as X = (I1, I2, I3), have profit V = {24, 25, 15} and weight W = {18, 15, 20}.

Item (x_i)	**Value (v_i)**	**Weight (w_i)**	$p_i = v_i / w_i$
I_2	25	15	1.67
I_1	24	18	1.33
I_3	15	20	0.75

Initialize, Weight of selected items, SW = 0,

Profit of selected items, SP = 0,

Set of selected items, S = { },

Here, Knapsack capacity M = 20.

Iteration 1 : SW= (SW + w_2) = 0 + 15 = 15

SW ≤ M, so select I_2

S = { I_2 }, SW = 15, SP = 0 + 25 = 25

Iteration 2 : SW + w_1 > M, so break down item I_1.

The remaining capacity of the knapsack is 5 unit, so select only 5 units of item I_1.

frac = (M – SW) / W[i] = (20 – 15) / 18 = 5 / 18
S = { I_2, I_1 * 5/18 }

SP = SP + v_1 * frac = 25 + (24 * (5/18)) = 25 + 6.67 = 31.67

SW = SW + w_1 * frac = 15 + (18 * (5/18)) = 15 + 5 = 20

The knapsack is full. Fractional Greedy algorithm selects items {I_2, I_1 * 5/18 }, and it gives a profit of **31.67 units**.

Code:

```c
#include <stdio.h>

void swap(float *a, float *b)
{
    float temp = *a;
    *a = *b;
    *b = temp;
}

float fractionalKnapsack(int n, float k, float w[], float nu[])
{
    float ratio[n];
    int i;
    float total_nutrition = 0.0;
    for(i=0; i<n; i++)
    {
        ratio[i] = nu[i]/w[i];
    }

    for(i=0; i<n-1; i++)
    {
        int j;
        for(j=i; j<n; j++)
        {
            if(ratio[i] < ratio[j])
            {
                swap(&ratio[i], &ratio[j]);
                swap(&w[i], &w[j]);
                swap(&nu[i], &nu[j]);
            }
        }
    }
    for(i=0; i<n; i++)
    {
        if(k > w[i])
        {
            total_nutrition += (float)(ratio[i]*w[i]);
            k -= w[i];
        }
        else if(k == w[i])
```

```
        {
            total_nutrition += (float)(ratio[i]*w[i]);
            return total_nutrition;
        }
        else
        {
            float r;
            r = (k*ratio[i]);
            total_nutrition += (float)r;
            return total_nutrition;
        }
    }
}

int main()
{
    float k;
    int n, i;
    printf("Enter the knapsack value: ");
    scanf("%f", &k);
    printf("\nEnter the number of fruits: ");
    scanf("%d", &n);

    float weight[n], nutrition[n];
    printf("\nEnter the weight of all fruits: ");
    for(i=0; i<n; i++)
    {
        scanf("%f", &weight[i]);
    }
    printf("\nEnter the nutrition value of all fruits: ");
    for (i=0; i<n; i++)
    {
        scanf("%f", &nutrition[i]);
    }
    float val = fractionalKnapsack(n, k, weight, nutrition);
    printf("\n Maximum nutrition value one can obtain is: %f",
val);

    return 0;
}
```

Department of Information Technology

Output:

```
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> gcc KnapsackGreedy.c
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> .\a.exe
Enter the knapsack value: 20

Enter the number of fruits: 3

Enter the weight of all fruits: 14
18
10

Enter the nutrition value of all fruits: 24
20
16

 Maximum nutrition value one can obtain is: 33.599998
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
```

Lab Assignment to Complete:

The capacity of the knapsack W = 60 and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24

Conclusion: In this experiment, We have implemented the Fractional Knapsack problem using Greedy approach. This approach efficiently solves the optimization problem of selecting fractional parts of items to maximize the total value within a Knapsack's capacity. This algorithm showcases a greedy approach by selecting items based on their value – to – weight ratios, providing an optimal solution in many scenarios.

Department of Information Technology

Experiment6

(Dynamic Programming)

Name: Kuldeep Suresh Choudhary **Batch:** I2 – 1

SapID: 60003220294 **Roll_no:** I080

Aim: Implementation of Matrix Chain Multiplication using dynamic programming.

Theory:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

For example: A is a 10 x 30 matrix, B is a 30 x 5 matrix, and C is a 5 x 60 matrix.

(AB)C = (10x30x5) + (10x5x60) = 1500 + 3000 = 4500 operations

A(BC) = (30x5x60) + (10x30x60) = 9000 + 18000 = 27000 operations.

APPROACH: -

1) Optimal Substructure:

- A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 3 matrices. let the chain be ABC, then there are 2 ways to place first set of parenthesis outer side: (A)(BC) and (AB)(C).
- So when we place a set of parenthesis, we divide the problem into sub-problems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.
- Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements.

RECURSIVE ALGORITHM to find the minimum cost:-

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out of each subsequence.
- Add these costs together, and add in the cost of multiplying the two matrices.
- Do this for each possible position at which the sequence of matrices can be split and take the minimum over all of them.
- The time complexity of above solution is exponential.

Since same suproblems are called again, this problem has overlapping subprolems property. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array dp[][] in bottom up manner.

Department of Information Technology

Pseudocode:

MATRIX-CHAIN-ORDER (p)
n ← length[p]-1
for i ← 1 to n
do m [i, i] ← 0
4. for l ← 2 to n // l is the chain length
do for i ← 1 to n-l + 1
do j ← i+ l -1
m[i,j] ← ∞
for k ← i to j-1
do q ← m [i, k] + m [k + 1, j] + pi-1 pk pj
if q < m [i,j]
then m [i,j] ← q
s [i,j] ← k
return m and s

Code:

```
#include <stdio.h>
#include <limits.h>
#define INFY 999999999
long int m[20][20];
int s[20][20];
int p[20], i, j, n;

void print_optimal(int i, int j)
{
    if (i == j)
        printf(" A%d ", i);
    else
    {
        printf("( ");
        print_optimal(i, s[i][j]);
        print_optimal(s[i][j] + 1, j);
        printf(" )");
    }
}

void matmultiply(void)
{
    long int q;
    int k;
    for (i = n; i > 0; i--)
    {
        for (j = i; j <= n; j++)
        {
            if (i == j)
```

```
                m[i][j] = 0;
            else
            {
                for (k = i; k < j; k++)
                {
                    q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                    if (q < m[i][j])
                    {
                        m[i][j] = q;
                        s[i][j] = k;
                    }
                }
            }
        }
    }
}

int MatrixChainOrder(int p[], int i, int j)
{
    if (i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k + 1, j) +
                p[i - 1] * p[k] * p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

void main()
{
    int k;
    printf("Enter the no. of elements: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        for (j = i + 1; j <= n; j++)
        {
```

```
            m[i][i] = 0;
            m[i][j] = INFY;
            s[i][j] = 0;
        }
    printf("\nEnter the dimensions: \n");
    for (k = 0; k <= n; k++)
    {
        printf("P%d: ", k);
        scanf("%d", &p[k]);
    }
    matmultiply();
    printf("\nCost Matrix M:\n");
    for (i = 1; i <= n; i++)
        for (j = i; j <= n; j++)
            printf("m[%d][%d]: %ld\n", i, j, m[i][j]);

    i = 1, j = n;
    printf("\nMultiplication Sequence : ");
    print_optimal(i, j);
    printf("\nMinimum number of multiplications is : %d ",
            MatrixChainOrder(p, 1, n));
}
```

Output:

```
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> gcc MCM.c
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> gcc MCM.c
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> .\a.exe
Enter the no. of elements: 4

Enter the dimensions:
P0: 5
P1: 4
P2: 6
P3: 2
P4: 7

Cost Matrix M:
m[1][1]: 0
m[1][2]: 120
m[1][3]: 88
m[1][4]: 158
m[2][2]: 0
m[2][3]: 48
m[2][4]: 104
m[3][3]: 0
m[3][4]: 84
m[4][4]: 0

Multiplication Sequence : ( (  A1 (  A2  A3  ) ) A4  )
Minimum number of multiplications is : 158
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
```

Department of Information Technology

Complexity:

Best Case Time Complexity: $O(n^2)$

Lab Assignment to Complete:

Find a minimum number of multiplications required to multiply: A [1 × 5], B [5 × 4], C [4 × 3], D [3 × 2], and E [2 × 1]. Also, give optimal parenthesization.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

NAME: Kuldeep Choudhary **DATE:** 01/04/2024
ROLL NO.: I080 **SAPID:** 60003220294
CLASS: S.Y. B.Tech I2-1 **COURSE:** DAA

Experiment 5

(Dynamic Programming)

Aim: Implementation of coin change problem using dynamic programming.

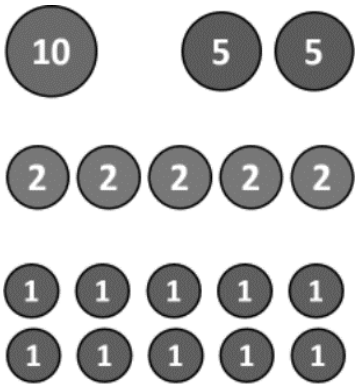
Theory:

Making Change problem is to find change for a given amount using a minimum number of coins from a set of denominations.

Explanation: If we are given a set of denominations D = {d0, d1, d2, …, dn} and if we want to change for some amount N, many combinations are possible. Suppose {d1, d2, d5, d8}, {d0, d2, d4}, {d0, d5, d7} all feasible solutions.

The aim of making a change is to find a solution with a minimum number of coins / denominations. Clearly, this is an optimization problem.

This problem can also be solved by using a greedy algorithm. However, greedy does not ensure the minimum number of denominations.



Various denominations for amount 10

General assumption is that infinite coins are available for each denomination. We can select any denomination any number of times.

Mathematical Formulation:

$$C[i,j] = \begin{cases} 1 + C[1, j - d_i], & \text{, if } i = j \\ C[i-1, j], & \text{if } j < d_i \\ \min(C[i-1, j], 1 + C[i, j - d_i]), & \text{otherwise} \end{cases}$$

Pseudocode:

Algorithm MAKE_A_CHANGE(d,N)
// d[1…n] = [d1,d2,…,dn] is array of n denominations

Department of Information Technology

```
// C[1…n, 0…N] is n x N array to hold the solution of sub problems
// N is the problem size, i.e. amount for which change is required

for i ← 1 to n do
  C[i, 0] ← 0
end

for i ← 1 to n do
  for  j ← 1 to N do
    if i = = 1 & & j < d [i] then
          C[i, j] ←  ∞
    else if i == 1 then
          C[i, j] ← 1 + C[1, j – d[1])
    else if j < d [i] then
          C[i, j] ←  C[I – 1, j]
    else
          C[i, j] ← min (C[i – 1, j] ,1 + C[i, j – d[i])
    end
end
end

return C[n, N]

Algorithm TRACE_MAKE_A_CHANGE(C)
// When table C is filled up, i = n and j = N

Solution = { }
while ( j > 0 ) do
  if (C[i, j] == C[i – 1, j] then
    i ← i – 1
  else
    j ← j – di
    Solution = Solution U {d[i] }
  end
end
```

Complexity:

Best Case Time Complexity: O(n)

Lab Assignment:

Write a C Program and consider the set of denominations, D=1,4,6. Achieve the sum of 8 and calculate the number of coins required and the actual denominations needed using dynamic programming.

Code:

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

```
#include <limits.h>

int minCoins(int coins[], int n, int sum) {
    int dp[sum + 1];
    dp[0] = 0;

    for (int i = 1; i <= sum; i++) {
        dp[i] = INT_MAX;
        for (int j = 0; j < n; j++) {
            if (coins[j] <= i && dp[i - coins[j]] != INT_MAX && dp[i - coins[j]] + 1 < dp[i]) {
                dp[i] = dp[i - coins[j]] + 1;
            }
        }
    }
    return dp[sum];
}

void printDenominations(int coins[], int n, int sum) {
    int dp[sum + 1];
    int prev[sum + 1];
    dp[0] = 0;

    for (int i = 1; i <= sum; i++) {
        dp[i] = INT_MAX;
        for (int j = 0; j < n; j++) {
            if (coins[j] <= i && dp[i - coins[j]] != INT_MAX && dp[i - coins[j]] + 1 < dp[i]) {
                dp[i] = dp[i - coins[j]] + 1;
                prev[i] = coins[j];
            }
        }
```

```
    }

    printf("Actual denominations needed: ");
    while (sum > 0) {
        printf("%d ", prev[sum]);
        sum -= prev[sum];
    }
    printf("\n");
}

int main() {
    int coins[] = {1, 4, 6};
    int sum = 8;
    int n = sizeof(coins) / sizeof(coins[0]);

    int numCoins = minCoins(coins, n, sum);

    printf("Minimum number of coins required to make %d: %d\n", sum, numCoins);
    printDenominations(coins, n, sum);

    return 0;
}
```

Output:

```
Minimum number of coins required to make 8: 2
Actual denominations needed: 4 4
```

Department of Information Technology

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Experiment 1

(Divide and Conquer)

Name: Kuldeep Suresh Choudhary **Roll_no:** I080

Sap-id: 60003220294 **Batch:** I2 – 1

Date: 20-02-24

Aim: Implementation of min-max algorithm using divide and conquer.

Theory:

Max-Min problem is to find a maximum and minimum element from the given array. We can effectively solve it using divide and conquer approach.

In the traditional approach, the maximum and minimum element can be found by comparing each element and updating Max and Min values as and when required. This approach is simple but it does (n – 1) comparisons for finding max and the same number of comparisons for finding the min. It results in a total of 2(n – 1) comparisons. Using a divide and conquer approach, we can reduce the number of comparisons.

Divide and conquer approach for Max. Min problem works in three stages.

- If a1 is the only element in the array, a1 is the maximum and minimum.
- If the array contains only two elements a1 and a2, then the single comparison between two elements can decide the minimum and maximum of them.
- If there are more than two elements, the algorithm divides the array from the middle and creates two subproblems. Both subproblems are treated as an independent problem and the same recursive process is applied to them. This division continues until subproblem size becomes one or two.

After solving two subproblems, their minimum and maximum numbers are compared to build the solution of the large problem. This process continues in a bottom-up fashion to build the solution of a parent problem.

Algorithm:

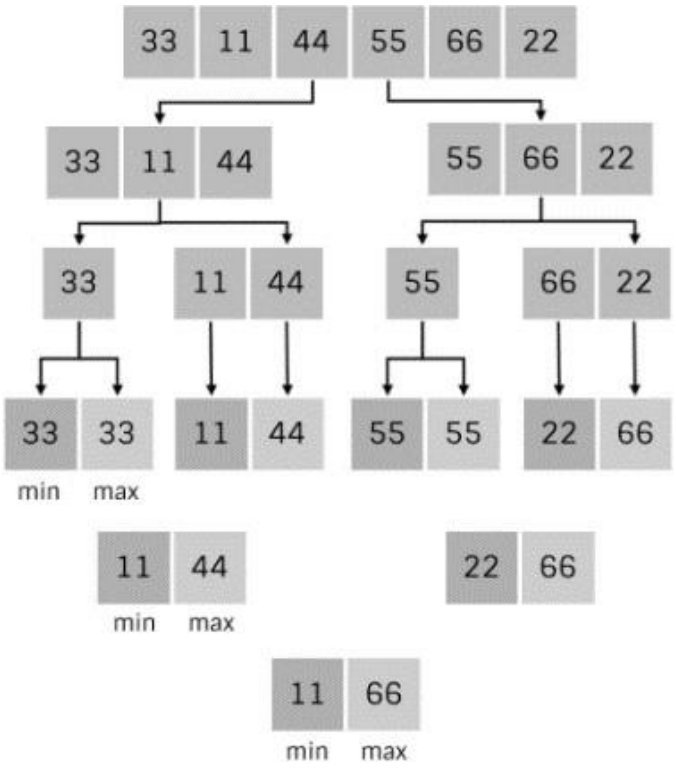
```
Algorithm DC_MAXMIN (A, low, high)
// Description : Find minimum and maximum element from array using divide and conquer approach
// Input : Array A of length n, and indices low = 0 and high = n - 1
// Output : (min, max) variables holding minimum and maximum element of array

if n == 1 then
  return (A[1], A[1])
else if n == 2 then if
  A[1] < A[2] then
```

```
return (A[1], A[2])
else return
(A[2],A[1]) else
mid ← (low + high) / 2
[LMin, LMax] = DC_MAXMIN (A, low, mid)
[RMin, RMax] = DC_MAXMIN (A, mid + 1, high)
if LMax > RMax then // Combine solution
  max ← LMax
else
  max ← RMax
end if LMin < RMin then    // Combine
solution
  min ← LMin
else
  min ← RMin
end return (min,
max)
end
```

Complexity:

Time complexity: O(n)

Example:



Code:

```
#include <stdio.h>

int max, min;
int a[100];

void minmax(int i, int j)
{
    int min1, max1;
    if(i == j)
    {
        max = min = a[i];
    }
    else
    {
        if(i == j-1)
        {
            if(a[i] < a[j])
            {
                max = a[j];
                min = a[i];
            }
            else
            {
                min = a[j];
                max = a[i];
            }
        }
        else
        {
            int mid = (i+j)/2;
            minmax(i, mid);
            max1 = max;
            min1 = min;
            minmax(mid+1, j);
            if(max1 > max)
            {
                max = max1;
            }
            if(min1 < min)
            {
                min = min1;
            }
        }
    }
}
void main()
{
    int n, i;
    printf("Enter the number of elements in the array: ");
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

```
    scanf("%d", &n);
    printf("\n Enter the values: ");
    for(i=1; i<=n; i++)
    {
        scanf("%d", &a[i]);
    }
    minmax(1, n);
    printf("\n Maximum value in the array is: %d", max);
    printf("\n Minimum value in the array is: %d", min);
}
```

Output:

```
Enter the number of elements in the array: 9

 Enter the values: 50
40
-5
-9
45
90
65
25
75

 Maximum value in the array is: 90
 Minimum value in the array is: -9
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
```

Conlusion: We have implemented the program which takes elements as an input in the array and find maximum and minimum value in the array using Divide and Conquer approach. This approach reduces the number of comparisons and offers a more efficient solution to the problem.

Department of Information Technology

Experiment 7

(Dynamic Programming)

Name: Kuldeep Suresh Choudhary **Batch:** I2 – 1

SapID: 60003220294 **Roll_no:** I080

Aim: Implementation of All pairs shortest path (Floyd Warshall) using dynamic programming.

Theory:

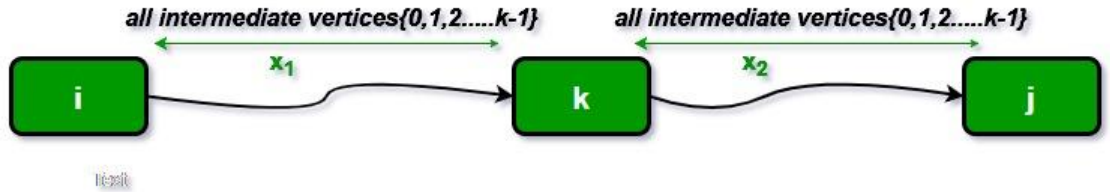
The **Floyd-Warshall algorithm**, named after its creators **Robert Floyd and Stephen Warshall**, is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both **positive** and n**egative edge weights**, making it a versatile tool for solving a wide range of network and connectivity problems.

The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Idea Behind Floyd Warshall Algortihm:
Suppose we have a graph G[][] with V vertices from 1 to N. Now we have to evaluate a shortestPathMatrix[][] where shortestPathMatrix[i][j] represents the shortest path between vertices i and j.

Obviously the shortest path between i to j will have some k number of intermediate nodes. The idea behind floyd warshall algorithm is to treat each and every vertex from 1 to N as an intermediate node one by one.

The following figure shows the above optimal substructure property in floyd warshall algorithm:



Floyd Warshall Algorithm Algorithm:

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.

- When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices.
- For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.
  - **k** is not an intermediate vertex in shortest path from **i** to **j**. We keep the value of **dist[i][j]** as it is.
  - **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]**

Pseudo-Code of Floyd Warshall Algorithm:

For k = 0 to n – 1
For i = 0 to n – 1
For j = 0 to n – 1
Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])

where i = source Node, j = Destination Node, k = Intermediate Node

Code:

```
#include <stdio.h>
int i, j, k, n, dist[10][10];
void floydWarshall()
{
    for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
}
int main()
{
    int i, j;
    printf("Enter no of vertices :");
    scanf("%d", &n);
    printf("\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            printf("dist[%d][%d]:", i, j);
            scanf("%d", &dist[i][j]);
        }
    floydWarshall();
    printf(" \n\n Shortest distances between every pair of vertices
\n");
```

```
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            printf("%d\t", dist[i][j]);
        printf("\n");
    }
    return 0;
}
```

Output:

```
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> gcc floydWarshall.c
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes> .\a.exe
Enter no of vertices :4

dist[0][0]:0
dist[0][1]:3
dist[0][2]:999
dist[0][3]:7
dist[1][0]:8
dist[1][1]:0
dist[1][2]:2
dist[1][3]:999
dist[2][0]:5
dist[2][1]:999
dist[2][2]:0
dist[2][3]:1
dist[3][0]:2
dist[3][1]:999
dist[3][2]:999
dist[3][3]:0


 Shortest distances between every pair of vertices
0       3       5       6
5       0       2       3
3       6       0       1
2       5       7       0
PS C:\Users\choud\Documents\IT SEM 4 NOTES\Design and Analysis of Algorithms (DAA)\DAA Codes>
```

Complexity:

Time Complexity: $O(n^3)$

Lab Assignment to Complete:

Apply Floyd Warshall on the following graph and get the shortest path from each node to every other node.

Department of Information Technology