NAME: NAGAVENI LG

SRN: PES2UG21CS315

SEC: F

Course Outcomes

At the end of the course, the student will be able to:

CO1. Identify the design technique used in an algorithm.
CO2. Analyse algorithms using quantitative evaluation.
CO3. Design and implement efficient algorithms for practical and unseen problems.
CO4. Analyse time efficiency over trading space.
CO5. Understand the limits of algorithms and the ways to cope with the limitations.

Question	Course Outcome	Topic
Q1		Boyer Moore
Q2		Horspool
Q3		Kruskal's
Q4		Prims

Ticking Session

Q1. An advertiser wants to publish his company's poster on a lane, wherever his company's competitor has published. To do so he walks down the lane and reads the posters as he goes. He maintains 2 tables - a good suffix and a bad character shift tables to see the number of steps to move ahead and read(Assume he can view only n letters forward(including the character at the position where he stands) at each position he stands, where n is the number of letters in the competitiors name). He compares the letters starting from the nth position from the place he is standing and moves towards the position where he is standing. Assume the posters as just a sequence of banners with a sentence in each. So you can treat the sequence as a single banner with text sequence. The good and bad scores are computed respectively with the value of the shift the man moves when the respective table is used. The scores are aggregated throughout the road. Return the good and bad scores at the end of the lane. The advertiser stops walking if he sees that the number of letters fall short than n when he has to shift.
Note - only one of the 2 scores increases when the advertiser shifts from each position.
Compute good score if all characters match.

Sample Input 1:
fliprflipperflflipper // Sequence of text man sees as he walks

flipper // Competitors name

Expected Output 1:
7 14

Boilerplate/Skeleton Code:
#include <stdio.h>
#include <string.h>

```
void operation
(char *text, int n, char *pattern, int m, int *gscore, int *bscore);
void bad_char_shift(char *pattern, int m, int *bad_char);
void good_suffix_shift(char *pattern, int m, int *good_suffix);
void operation
(char *text, int n, char *pattern, int m, int *gscore, int *bscore) {
    int i, j;
    int bad_char[256];
    int good_suffix[m];

    bad_char_shift(pattern, m, bad_char);
    good_suffix_shift(pattern, m, good_suffix);

    i = m - 1;
    *gscore = 0;
    *bscore = 0;
    while (i < n) {
        j = m - 1;
        while (text[i] == pattern[j]) {
            if (j == 0) {
                (*gscore)++;
                return;
            }
            (*gscore)++;
            i--;
            j--;
        }
        (*bscore)++;
        i += MAX(bad_char[(int) text[i]], good_suffix[j]);
```

```
    }
    // Pattern not found, return pattern length
    if (j == m - 1) {
        *gscore = 0;
        *bscore = m;
    }
}

void bad_char_shift(char *pattern, int m, int *bad_char) {
    int i;
    for (i = 0; i < 256; i++) {
        bad_char[i] = m;
    }
    for (i = 0; i < m - 1; i++) {
        bad_char[(int) pattern[i]] = m - i - 1;
    }
}

void good_suffix_shift(char *pattern, int m, int *good_suffix) {
    int i, j;
    int suff[m];

    suff[m - 1] = m;
    j = m - 1;
    for (i = m - 2; i >= 0; i--) {
        while (j < m - 1 && pattern[j] != pattern[i]) {
            j = suff[j + 1] - 1;
        }
        if (pattern[j] == pattern[i]) {
            j--;
        }
        suff[i] = j + 1;
    }
    for (i = 0; i < m; i++) {
        good_suffix[i] = m - suff[i];
    }
}
```

}

// **Driver's code**
int main() {

```
    char text[500],pattern[500];
    int gscore=0,bscore=0;
    gets(text);
    gets(pattern);
    operation(pattern, text,&gscore,&bscore);
    printf("%d %d",gscore,bscore);
    return 0;
}
```

Test cases:

1.
adidas
adidas

Output -
6 0

2.
powernike
power

Output -
5 0

3.
amulnandini
goverdhan

Output -
0 9

Q2. A string matching algorithm uses only the bad-character shift of the rightmost character of the text window to compute the shifts. For a given text and pattern, compute positions of two occurences of the pattern for which the shift remains at most equal to the length of the text. If no match is found return -1.

Sample Input 1:
hello hey heno // Text
he // Pattern

Expected Output 1:
1 7

Boilerplate/Skeleton Code:
#include<stdio.h>
#include<string.h>
#include<conio.h>

void operation(char text[],char pattern[])
{
// **Write your code here**

```
int text_len = strlen(text);
    int pattern_len = strlen(pattern);


    int table[MAX_CHAR];
    shift_table(pattern, pattern_len, table);


    int count = 0;
    for (int i = pattern_len - 1; i < text_len;) {
        int j = pattern_len - 1;
        while (j >= 0 && pattern[j] == text[i - pattern_len + 1 + j]) {
            j--;
        }
        if (j == -1) {
            printf("%d ", i - pattern_len + 2);
            count++;
            if (count == 2 || pattern_len == text_len) {
                return;
            }
            i++;
        } else {
            i += table[(int) text[i]];
        }
    }


    printf("-1");
```

```
}

// Driver's Code
void main()
{
        char text[500],pattern[500];
        gets(text);
        gets(pattern);
        operation(text,pattern);
}
```

Test cases:

1.
zoe joe
john

Output -
-1

2.
string matching
string matching

Output -
1

3.
this is furnishing
is

Output -
3 6

Q3. A group of kids are standing in different positions in a playground. Each of them are holding the ends of different ropes. Each rope is held by only 2 kids, each holding one end. The length of the rope held between 2 kids depend on how far apart they are standing. More than sufficient number of ropes are distributed such that everyone stay connected as a single group with the ropes. All of them have been told to stay connected with the ropes as a single group. Since the only goal is to stay connected, they tend to put down

the ropes which are heavier and does’nt get them disconnected from the group.
Print the total length of the remaining ropes between the kids.
(Hint : Weight of rope is proportional to its length)

Sample Input 1:
5 // Number of Kids
7 // Total number of ropes
1 2 1 // kid1 kid2 length_of_rope
1 3 7
2 3 5
2 4 3
2 5 4
3 4 6
4 5 2

Expected Output 1:
11

Boilerplate/Skeleton Code:
#include <stdio.h>

void operation(int n, int (*graph)[n]) {
// **Write your code here**

```
    int parent[n], rank[n];
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        rank[i] = 0;
    }


    int total_weight = 0;


    // Sort edges by weight (length of ropes)
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i - 1; j >= 0; j--) {
            if (graph[i][j] > graph[j][i]) {
                int temp = graph[i][j];
                graph[i][j] = graph[j][i];
```

```
                graph[j][i] = temp;
            }
        }
    }


    // Iterate over edges and add heaviest edge that does not create a cycle
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (graph[i][j] > 0) {
                int root_i = i, root_j = j;
                while (parent[root_i] != root_i) {
                    root_i = parent[root_i];
                }
                while (parent[root_j] != root_j) {
                    root_j = parent[root_j];
                }
                if (root_i != root_j) {
                    // Add edge to minimum spanning tree
                    total_weight += graph[i][j];
                    // Merge components using union-by-rank
                    if (rank[root_i] > rank[root_j]) {
                        parent[root_j] = root_i;
                    } else {
                        parent[root_i] = root_j;
                        if (rank[root_i] == rank[root_j]) {
                            rank[root_j]++;
                        }
                    }
                }
            }
        }
    }


    // Print total weight of remaining ropes
    printf("%d\n", total_weight);
```

}

void main() {

```
  //Driver's code
  int n, i, j,edges,e,src,dest,len;
  scanf("%d",&n);
  int graph[n][n];
  scanf("%d",&edges);
  for(i=0;i<n;i++){
  for(j=0;j<n;j++){
      graph[i][j]=0;
    }
  }
  for(e=0;e<edges;e++){
    scanf("%d %d %d",&src,&dest,&len);
    graph[src-1][dest-1]=len;
    graph[dest-1][src-1]=len;
  }
  operation(n,graph);
}
```

Test cases:

1.

```
1
0
```

Output -

```
0
```

2.

```
3
3
1 2 4
1 3 4
2 3 4
```

Output -

```
8
```

3.

```
4
4
1 2 1
1 4 3
2 3 2
3 4 4
```

Output -
6

Q4. Write a program for constructing a minimum cost spanning tree of a graph using Prim's algorithm.

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

#define MAX_NODES 100
#define INF 1e9

int graph[MAX_NODES][MAX_NODES];
int dist[MAX_NODES];
int visited[MAX_NODES];

int main() {
    int n, start, m;
    scanf("%d%d%d", &n, &start, &m);

    // Initialize graph with infinite distance between all nodes
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            graph[i][j] = INF;
        }
    }

    // Read in edges and their costs
    for (int i = 0; i < m; i++) {
```

```
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        graph[u][v] = w;
        graph[v][u] = w; // graph is undirected
    }

    // Initialize distances to all nodes as infinite except the starting node
    for (int i = 1; i <= n; i++) {
        if (i == start) {
            dist[i] = 0;
        } else {
            dist[i] = INF;
        }
    }

    // Prim's algorithm
    int total_cost = 0;
    for (int i = 1; i <= n; i++) {
        int min_dist = INF, min_node = -1;
        for (int j = 1; j <= n; j++) {
            if (!visited[j] && dist[j] < min_dist) {
                min_dist = dist[j];
                min_node = j;
            }
        }

        if (min_node == -1) {
            break;
```

```
        }

        visited[min_node] = 1;
        total_cost += dist[min_node];

        for (int j = 1; j <= n; j++) {
                    if (graph[min_node][j] != INF && !visited[j] &&
graph[min_node][j] < dist[j]) {
                dist[j] = graph[min_node][j];
            }
        }
    }

    printf("%d\n", total_cost);

    return 0;
}
```

Sample Input 1:

5 //N:number nodes

1 //start node (tower number from which the Prim's algorithm begins)
6 // number of edges associated with cost
1 2 3
1 3 4
4 2 6
5 2 2
2 3 5
3 5 7

Expected Output 1:
15

Test cases:
// give 2 test cases of your own