Name: Amruta Pede

Reg. no.:2020BIT070

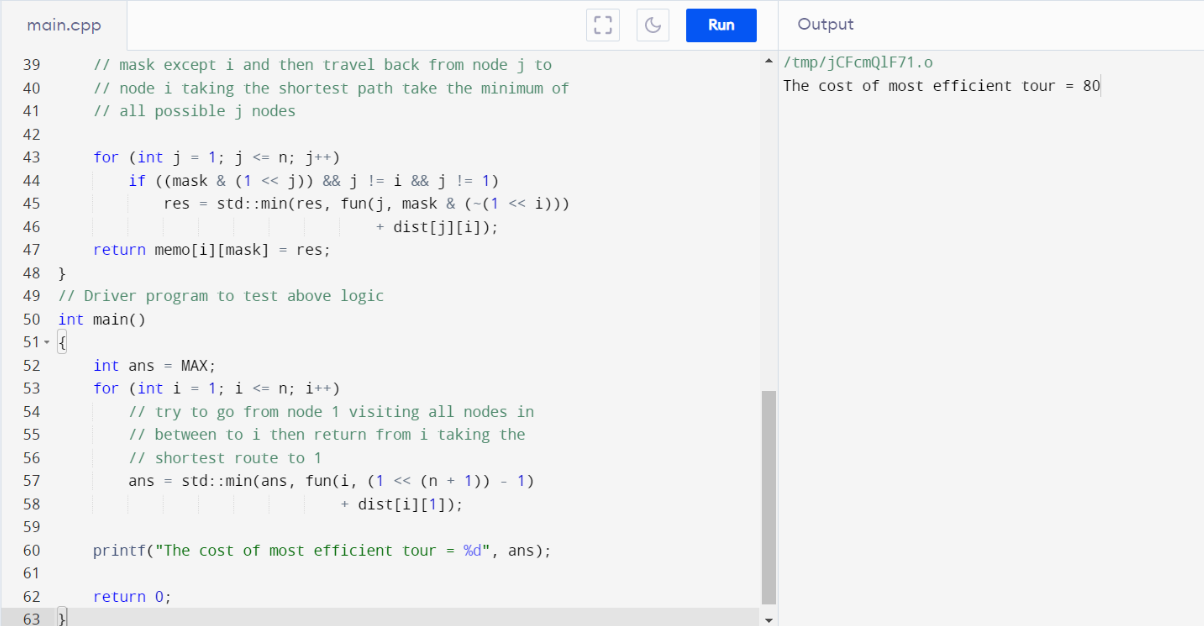
Assignment 4

write a c/c++ code for following algorithm with explanation

1) travelling salesman problem :

```
#include <iostream>

using namespace std;

// there are four nodes in example graph (graph is 1-based)
const int n = 4;
// give appropriate maximum to avoid overflow
const int MAX = 1000000;

// dist[i][j] represents shortest distance to go from i to j
// this matrix can be calculated for any given graph using
// all-pair shortest path algorithms
int dist[n + 1][n + 1] = {
        { 0, 0, 0, 0, 0 }, { 0, 0, 10, 15, 20 },
        { 0, 10, 0, 25, 25 }, { 0, 15, 25, 0, 30 },
        { 0, 20, 25, 30, 0 },
};

// memoization for top down recursion
int memo[n + 1][1 << (n + 1)];

int fun(int i, int mask)
{
        // base case
        // if only ith bit and 1st bit is set in our mask,
        // it implies we have visited all other nodes already
        if (mask == ((1 << i) | 3))
                return dist[1][i];
        // memoization
        if (memo[i][mask] != 0)
                return memo[i][mask];

        int res = MAX; // result of this sub-problem

        // we have to travel all nodes j in mask and end the
        // path at ith node so for every node j in mask,
        // recursively calculate cost of travelling all nodes in
        // mask except i and then travel back from node j to
        // node i taking the shortest path take the minimum of
        // all possible j nodes

        for (int j = 1; j <= n; j++)
                if ((mask & (1 << j)) && j != i && j != 1)
                        res = std::min(res, fun(j, mask & (~(1 << i)))
                                                                + dist[j][i]);

        return memo[i][mask] = res;
}
// Driver program to test above logic
int main()
{
        int ans = MAX;
        for (int i = 1; i <= n; i++)
                // try to go from node 1 visiting all nodes in
                // between to i then return from i taking the
                // shortest route to 1
                ans = std::min(ans, fun(i, (1 << (n + 1)) - 1)
                                                        + dist[i][1]);

        printf("The cost of most efficient tour = %d", ans);

        return 0;
}
```

Output:



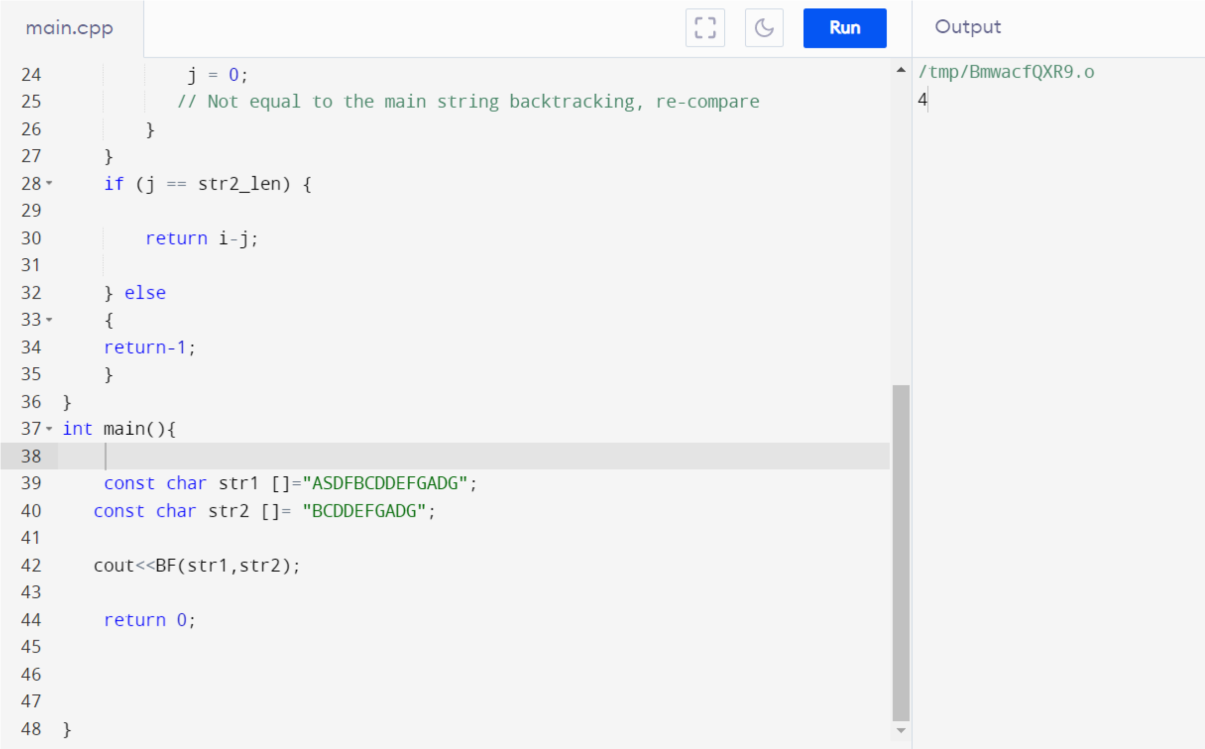
```
    // mask except i and then travel back from node j to
    // node i taking the shortest path take the minimum of
    // all possible j nodes

    for (int j = 1; j <= n; j++)
        if ((mask & (1 << j)) && j != i && j != 1)
            res = std::min(res, fun(j, mask & (~(1 << i)))
                                    + dist[j][i]);
    return memo[i][mask] = res;
}
// Driver program to test above logic
int main()
{
    int ans = MAX;
    for (int i = 1; i <= n; i++)
        // try to go from node 1 visiting all nodes in
        // between to i then return from i taking the
        // shortest route to 1
        ans = std::min(ans, fun(i, (1 << (n + 1)) - 1)
                                + dist[i][1]);

    printf("The cost of most efficient tour = %d", ans);

    return 0;
}
```

```
/tmp/jCFcmQ1F71.o
The cost of most efficient tour = 80
```

2)BF String matching Algorithm:

```
#include <istream>
using namespace std;
#include <bits/stdc++.h>
int BF (const char *str1, const char *str2)
{
   int str1_len = strlen (str1);
   int str2_len = strlen (str2);
   int i = 0;
   int j = 0;

   if (str1 == NULL || str2 == NULL) {
      return-1;
   }

   while (i < str1_len && j < str2_len) {
      if (str1[i] == str2[j]) {
         i++;
         j++;
         //Equality continues to be compared
      }
      else{
         i = i-j + 1;
         j = 0;
        // Not equal to the main string backtracking, re-compare
      }
   }
   if (j == str2_len) {

      return i-j;

   } else
   {
   return-1;
   }
}
int main(){

   const char str1 []="ASDFBCDDEFGADG";
  const char str2 []= "BCDDEFGADG";

  cout<<BF(str1,str2);

   return 0;
   }
```

Output:



```
            j = 0;
           // Not equal to the main string backtracking, re-compare
        }
    }
    if (j == str2_len) {

        return i-j;

    } else
    {
    return-1;
    }
}
int main(){

    const char str1 []="ASDFBCDDEFGADG";
   const char str2 []= "BCDDEFGADG";

   cout<<BF(str1,str2);

    return 0;



}
```

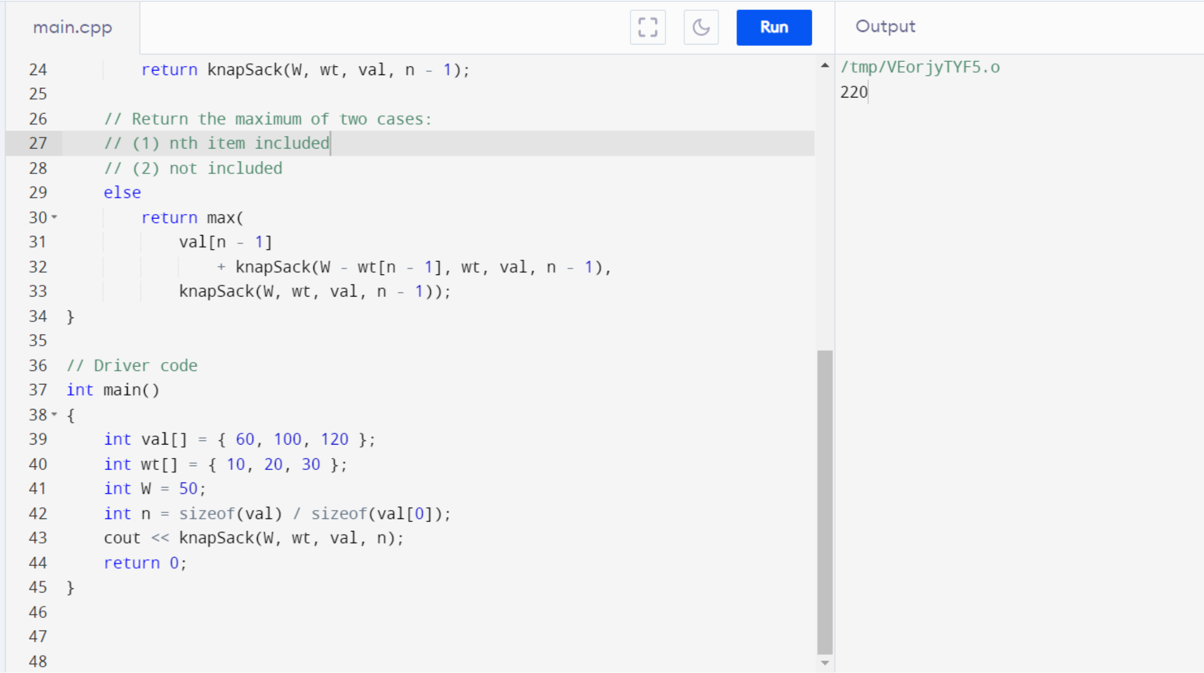
```
/tmp/BmwacfQXR9.o
4
```

3)Exhaustive Search algorithm:

```
/* A Naive recursive implementation of
0-1 Knapsack problem */
#include <bits/stdc++.h>
using namespace std;

// A utility function that returns
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{

        // Base Case
        if (n == 0 || W == 0)
                return 0;

        // If weight of the nth item is more
        // than Knapsack capacity W, then
        // this item cannot be included
        // in the optimal solution
```

```
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}

// Driver code
int main()
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    cout << knapSack(W, wt, val, n);
    return 0;
}
```

Output:



```
        return knapSack(W, wt, val, n - 1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}

// Driver code
int main()
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    cout << knapSack(W, wt, val, n);
    return 0;
}
```

```
/tmp/VEorjyTYF5.o
220
```