OCTOBER 06,2020

DESIGN AND ANALYSIS OF ALGORITHMS

(QUESTION: 6.1, LONGEST COMMON SUBSEQUENCE)

EXERCISE 6

SUBMITTED BY-

NAME - RAJ KRISHNA

ROLL NO - 1805233

BATCH - CSE-G1

GROUP - B

1. **The objective of the Experiment**

   The objective of the experiment is to select the **Longest Common Subsequence (LCS)** in the given sequences by **Dynamic Programming.**

2. **Solution Code**

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

void lcsAlgo(char *S1, char *S2, int m, int n) {
 int LCS[m + 1][n + 1];

 for (int i = 0; i <= m; i++) {
       for (int j = 0; j <= n; j++) {
       if (i == 0 || j == 0)
             LCS[i][j] = 0;
       else
             if (S1[i - 1] == S2[j - 1])
             LCS[i][j] = LCS[i - 1][j - 1] + 1;
             else
             LCS[i][j] = max(LCS[i - 1][j], LCS[i][j - 1]);
       }
 }

 int index = LCS[m][n];
 char lcsAlgo[index + 1];
 lcsAlgo[index] = '\0';
 int i = m, j = n;
 while (i > 0 && j > 0) {
       if (S1[i - 1] == S2[j - 1]) {
       lcsAlgo[index - 1] = S1[i - 1];
       i--;
       j--;
       index--;
```

```
        }
        else
        if (LCS[i - 1][j] > LCS[i][j - 1])
                i--;
        else
                j--;
  }
  cout << "[ " << S1 << " ]\n[ " << S2 << " ]\nLCS: " << lcsAlgo << "\n";
}

int main()
{
  char S1[] = "0010210122";
  char S2[] = "2202020120";
  int m = strlen(S1);
  int n = strlen(S2);

  lcsAlgo(S1, S2, m, n);
}
```

3. Summary of the program

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

The following steps are followed for finding the longest common subsequence.

- ➢ Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

- ➢ Fill each cell of the table using the following logic.
- ➢ If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
- ➢ Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
- ➢ **Step 2** is repeated until the table is filled.
- ➢ The value in the last row and the last column is the length of the longest common subsequence.
- ➢ In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.
- ➢ Thus, the longest common subsequence is 002012

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

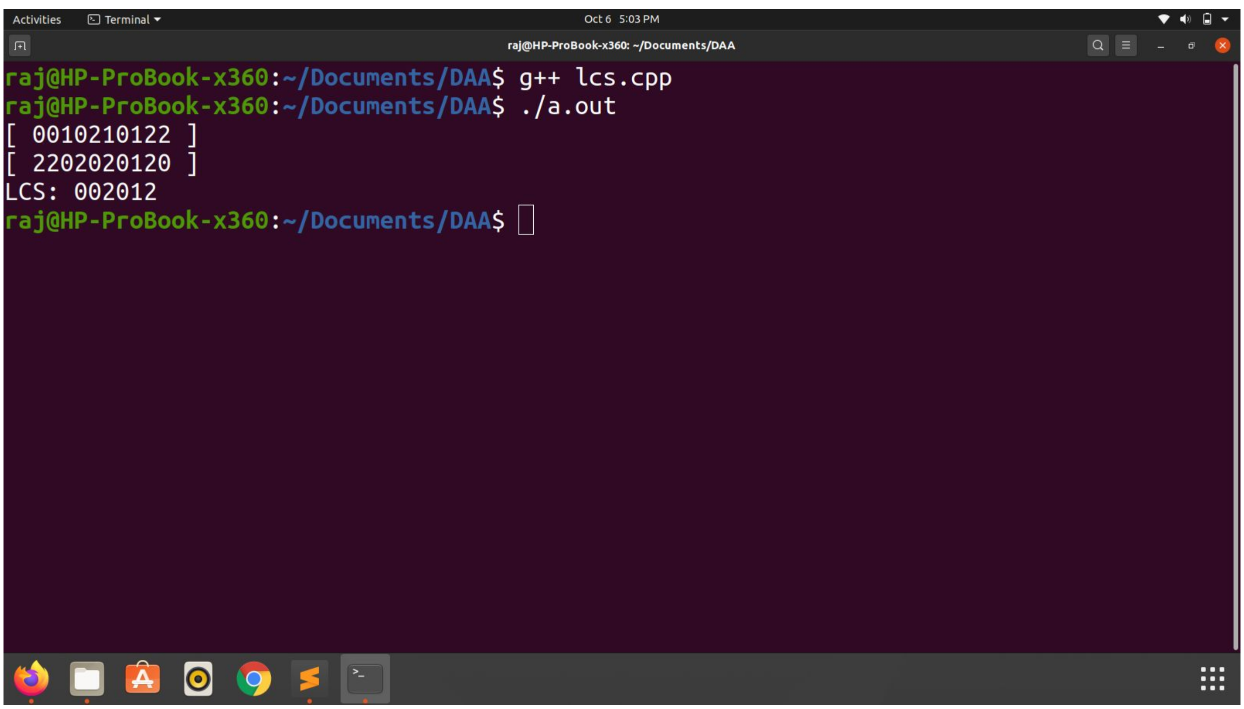
The time taken by a dynamic approach is the time taken to fill the table is O(m*n). Whereas, the recursion algorithm has the complexity of $2^{max(m, n)}$.

Time Complexity- O(m*n)

4. **Sample Output**

Input-
First Sequence: 0010210122
Second Sequence: 2202020120

Output-
[0010210122]
[2202020120]
LCS: 002012



OCTOBER 13,2020

DESIGN AND ANALYSIS OF ALGORITHMS

(QUESTION: 6.2, BINARY KNAPSACK)
EXERCISE 6

SUBMITTED BY-
NAME - RAJ KRISHNA
ROLL NO - 1805233
BATCH - CSE-G1
GROUP - B

1. **The objective of the Experiment**

The objective of the experiment is to determine the **names of each item** and **total value** from a given collection set of items each with a **name, weight and value** so that the **total weight** is less than or equal to a given limit and the total value is as large as possible by **(0/1) Binary Knapsack technique**.

2. **Solution Code**

```
#include<bits/stdc++.h>
using namespace std;
int max(int a, int b)
{
    if(a>b)
     return a;
   else
return b;
}

int knapsack(int W, int weight[], int value[], int n, string names[])
{
    int i, w,x,y;
    int K[n + 1][W + 1];
    string object[n];
int p=0;

    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (weight[i - 1] <= w)
          K[i][w] = max(value[i - 1] + K[i - 1][w - weight[i - 1]],
                    K[i - 1][w]);
```

```
            else
                K[i][w] = K[i - 1][w];
        }
    }

   int result = K[n][W];
   w = W;
    for (i = n; i > 0 && result > 0; i--)
         {

         if (result== K[i - 1][w])
           {
            continue;
           }
        else
           {

           object[p]=names[i - 1];
            p++;
            result = result - value[i - 1];
            w = w - weight[i - 1];
        }
    }
cout<<"Knapsack: ";
for(i=p;i>=0;i--)
{

  cout<<object[i]<<" ";
}
cout<<endl;
return K[n][W];
}
```

```
int main()
{
 string
names[]={"map","compass","water","sandwich","glucose","tin","banana","a
pple","cheese","beer","suntan-cream","camera","t-shirts","trousers","umbre
lla","waterproof-trousers","waterproof-overclothes","note-case","sunglasse
s","towels","socks","books"};
    int wt[] = {
9,13,153,50,15,68,27,39,23,52,11,32,24,48,73,42,43,22,7,18,4,30};
    int val[] = {
150,35,200,160,60,45,60,40,30,10,70,30,15,10,40,70,75,80,20,12,50,10 };
    int W = 400;
    int n = sizeof(val) / sizeof(val[0]);
   int result= knapsack(W, wt, val, n,names);
cout<<"value:"<<result<<endl;
    return 0;
}
```

3. Summary of the program

We have given items i_1, i_2 , ..., i_n (the item we want to put in our bag) with associated weights w_1, w_2, ... w_n and profit values V_1, V_2, ... V_n. In a **Fractional knapsack**, either you take the item completely or you don't take it.

In order to solve the **0-1 knapsack problem**, our **greedy method fails** which we used in the fractional knapsack problem. So the only method we have for this optimization problem is solved using **Dynamic Programming**, for applying Dynamic programming to this problem we have to do three things in this problem:

1. Optimal substructure
2. Writing the recursive equation for substructure
3. Whether subproblems are repeating or not

Now assume we have **'n'** items **1 2 3 ... N**. I will take an item and observe that there are two ways to consider the item either

1. it could be included in knapsack
2. we might not include it in knapsack

Likewise, every element has 2 choices. Therefore we have **2X2X2X2...** Upto **n** choices i.e **2^n choices**.

We have to consider the **2^n solution** to find out the optimal answer but now we have to find that is there any repeating substructure present in the problem so that it is exempt from examining **2^n solutions**.

The recursive equation for this problem is given below:

```
 knapsack(i,w) = { max( Vi +knapsack(i-1,W-wi) , knapsack(i-1,W) )
                   0,i=0 & W=0
                   Knapsack(i-1,W)   , wi> W
}
```

Knapsack(i-1,W) : is the case of not including the ith item. In this case we are not adding any size to knapsack.

Vi + Knapsack(i-1,W-wi) : indicates the case where we have selected the ith item. If we add i^{th} item then we need to add the value Vi to the optimal solution.

Number of unique subproblems in **0-1 knapsack problem** is **(n X W)**. We use tabular methods using Bottom-up Dynamic programming to reduce the time from **O(2^n) to O(n X W)**.

4. Sample Output

Input-

string names[]={"map","compass","water","sandwich","glucose","tin","banana","apple","cheese","beer","suntan-cream","camera","t-shirts","trousers","umbrella","waterproof-trousers","waterproof-overclothes","note-case","sunglasses","towels","socks","books"};

```
    int wt[] = {
9,13,153,50,15,68,27,39,23,52,11,32,24,48,73,42,43,22,7,18,4,30};
    int val[] = {
150,35,200,160,60,45,60,40,30,10,70,30,15,10,40,70,75,80,20,12,50,10 };
    int W = 400;
```

Output-

Knapsack: map compass water sandwich glucose banana suntan-cream waterproof-trousers waterproof-overclothes note-case
sunglasses socks
value:1030