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)

Academic Year: 2022-2023

Name	**Prerna Sunil Jadhav**
Sap Id	**60004220127**
Class	**S. Y. B.Tech (Computer Engineering)**
Course	**Analysis of Algorithm Laboratory**
Course Code	**DJ19CEL404**
Experiment No.	**06-10** (except Exp-07)

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	06

AIM: **TO IMPLEMENT LONGEST COMMON SUBSEQUENCE PROBLEM.**

THEORY:

LCS

- 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.
- If S1 and S2 are the two given sequences then, Z is the common subsequence
- of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a strictly
- increasing sequence of the indices of both S1 and S2.
- In a strictly increasing sequence, the indices of the elements chosen from the original
- sequences must be in ascending order in Z.
- **Pseudocode:**

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
If X[i] = Y[j]
    LCS[i][j] = 1 + LCS[i-1, j-1]
    Point an arrow to LCS[i][j]
Else
    LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
    Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

CODE:

```
#include <stdio.h>
#include <string.h>
int i, j, m, n, LCS_table[20][20];
char S1[20] = "ACADB", S2[20] = "CBDA", b[20][20];
void lcsAlgo()
{
    m = strlen(S1);
```

```
    n = strlen(S2);
    // Filling 0's in the matrix
    for (i = 0; i <= m; i++)
        LCS_table[i][0] = 0;
    for (i = 0; i <= n; i++)
        LCS_table[0][i] = 0;
    // Building the mtrix in bottom-up way
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
        {
            if (S1[i - 1] == S2[j - 1])
            {
                LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
            }
            else if (LCS_table[i - 1][j] >= LCS_table[i][j - 1])
            {
                LCS_table[i][j] = LCS_table[i - 1][j];
            }
            else
            {
                LCS_table[i][j] = LCS_table[i][j - 1];
            }
        }
    int index = LCS_table[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_table[i - 1][j] > LCS_table[i][j - 1])
            i--;
        else
            j--;
    }
    printf("S1 : %s \nS2 : %s \n", S1, S2);
    printf("LCS: %s", lcsAlgo);
}
```

```
int main()
{
    lcsAlgo();
    printf("\n");
}
```

OUTPUT:

```
exe' '--interpreter=mi'
S1 : ACADB
S2 : CBDA
LCS: CB
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

- Thus, we implemented Longest Common Subsequence and found the longest common subsequence in 2 strings.

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	08

AIM: **TO IMPLEMENT N QUEEN'S PROBLEM**

THEORY:

N_QUEEN

- The N Queens problem is a classic problem in computer science and mathematics.
- The problem asks for the number of ways N queens can be placed on an N×N
- chessboard such that no two queens threaten each other.
    - In other words, the queens cannot share the same row, column, or diagonal. One approach to solving this problem is to use backtracking. The algorithm places one queen in each column, starting from the leftmost column.
    - Once a queen is placed, the algorithm checks if it is under attack by any of the previously placed queens. If it is not under attack, the algorithm moves to the next column and places another queen.
    - If a queen cannot be placed in a column without being under attack, the algorithm backtracks to the previous column and tries a different row for that column.
- **Algorithm:**

```
1. Initialize an empty chessboard of size NxN.
2. Start with the leftmost column and place a queen in the first row of that
column.
3. Move to the next column and place a queen in the first row of that column.
4. Repeat step 3 until either all N queens have been placed or it is
impossible to place a
queen in the current column without violating the rules of the problem.
5. If all N queens have been placed, print the solution.
6. If it is not possible to place a queen in the current column without
violating the rules of
the problem, backtrack to the previous column.
7. Remove the queen from the previous column and move it down one row.
8. Repeat steps 4-7 until all possible configurations have been tried.
```

CODE:

```
#define N 8
#include <stdbool.h>
#include <stdio.h>
void printSolution(int board[N][N])
{
```

```
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;
    return true;
}
bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++)
    {
        if (isSafe(board, i, col))
        {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}
bool solveNQ()
{
    int board[N][N] = {{0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0}};
    if (solveNQUtil(board, 0) == false)
```

```
    {
        printf("Solution does not exist");
        return false;
    }
    printSolution(board);
    return true;
}
int main()
{
    solveNQ();
    return 0;
}
```

OUTPUT:

```
exe' '--interpreter=mi'
 1  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0
 0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1
 0  1  0  0  0  0  0  0
 0  0  0  1  0  0  0  0
 0  0  0  0  0  1  0  0
 0  0  1  0  0  0  0  0
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

- Thus, we implemented the code to solve N Queens Problem. Here, 8 Queens Problem.

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	09

AIM: **TO IMPLEMENT SUM OF SUBSET PROBLEM**

THEORY:

SUBSET PROBLEM

- Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K.
- We are considering the set contains non-negative values.
- It is assumed that the input set is unique (no duplicates are presented).
- **Algorithm:**

```
Let, S = {S1 .... Sn} be a set of n positive integers, then we have to find a
subset whose sum is equal to given positive integer d. It is always convenient
to sort the set's elements in ascending order. That is, S1 ≤ S2 ≤.... ≤ Sn
Algorithm:
Let, S is a set of elements and m is the expected sum of subsets. Then:
1. Start with an empty set.
2. Add to the subset, the next element from the list.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set then
backtrack through the subset until we find the most suitable value.
5. If the subset is feasible then repeat step 2.
6. If we have visited all the elements without finding a suitable subset and
if no backtracking is possible then stop without solution.
```

CODE:

```
#include <stdio.h>
int m, n, arr[100], x[100] = {0};
int SumOfSubsets(int s, int k, int r)
{
    x[k] = 1;
    if (s + arr[k] == m)
    {
        for (int j = k + 1; j < n; j++)
        {
            x[j] = 0;
        }
        printf("Answer is\n");
        for (int i = 0; i < n; i++)
```

```
        {
            printf("%d ", x[i]);
        }
        printf("\n");
    }
    else if (s + arr[k] + arr[k + 1] <= m)
    {
        SumOfSubsets(s + arr[k], k + 1, r - arr[k]);
    }
    if (s + r - arr[k] >= m && s + arr[k + 1] <= m)
    {
        x[k] = 0;
        SumOfSubsets(s, k + 1, r - arr[k]);
    }
}
int main()
{
    int s = 0;
    printf("Sum of Subsets\nEnter the number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
        s += arr[i];
    }
    printf("Enter the sum needed: ");
    scanf("%d", &m);
    SumOfSubsets(0, 0, s);
    return 0;
}
```

OUTPUT:

```
exe' '--interpreter=mi'
Sum of Subsets
Enter the number of elements: 4
Enter 4 elements:
2
4
56
7
Enter the sum needed: 60
Answer is
0 1 1 0
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code> 
```

CONCLUSION:

- Thus, we implemented the code to solve Sum of subset problem.

Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	10

AIM: **TO IMPLEMENT STRING MATCHING USING RABIN KARP AND KMP ALGORITHM.**

THEORY:

RABIN KARP STRING MATCHING

- Like the Naive Algorithm, the Rabin-Karp algorithm also slides the pattern one by one.
- But unlike the Naive algorithm, the Rabin Karp algorithm matches the hash value of the pattern with the hash value of the current substring of text, and if the hash values match then only it starts matching individual characters.
- So Rabin Karp algorithm needs to calculate hash values for the following strings.
  - Pattern itself
  - All the substrings of the text of length m
- **Algorithm:**

```
Initially calculate the hash value of the pattern.
Start iterating from the starting of the string:
Calculate the hash value of the current substring having length m.
If the hash value of the current substring and the pattern are same check if
the
substring is same as the pattern.
If they are same, store the starting index as a valid answer. Otherwise,
continue
for the next substrings.
Return the starting indices as the required answer.
```

CODE:

```c
#include <stdio.h>
#include <string.h>
int d = 23;
void search(char P[], char T[], int q)
{
    int m = strlen(P);
    int n = strlen(T);
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;
    for (i = 0; i < m - 1; i++)
```

```
        h = (h * d) % q;
    for (i = 0; i < m; i++)
    {
        p = (d * p + P[i]) % q;
        t = (d * t + T[i]) % q;
    }
    for (i = 0; i <= n - m; i++)
    {
        if (p == t)
        {
            for (j = 0; j < m; j++)
            {
                if (T[i + j] != P[j])
                    break;
            }
            if (j == m)
                printf("Pattern found at index %d \n", i);
        }
        if (i < n - m)
        {
            t = (d * (t - T[i] * h) + T[i + m]) % q;
            if (t < 0)
                t = (t + q);
        }
    }
}
int main()
{
    char P[200], T[200];
    printf("Enter the text: \n");
    gets(T);
    printf("The text is %s \n", T);
    printf("Enter the pattern: \n");
    gets(P);
    printf("The pattern is %s \n", P);

    int q = 13;

    search(P, T, q);
    return 0;
}
```

OUTPUT:

```
exe' '--interpreter=mi'
Enter the text:
preftgrhhpregffpremkg
The text is preftgrhhpregffpremkg
Enter the pattern:
pregff
The pattern is pregff
Pattern found at index 9
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code> 
```

KMP STRING MATCHING

- The KMP matching algorithm uses degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n).
- The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window.
- We take advantage of this information to avoid matching the characters that we know will anyway match.
- **Algorithm:**

```
Pre-processing overview:
• KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m
(same as the size of the pattern) which is used to skip characters while
matching.
• name lps indicates the longest proper prefix which is also a suffix. A
proper prefix is a prefix with a whole string not allowed. For example,
prefixes of “ABC” are “”, “A”,
“AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string
are “”, “C”, “BC”, and “ABC”.
• We search for lps in sub-patterns. More clearly we focus on sub-strings of
patterns that are both prefix and suffix.
• For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length
of the maximum matching proper prefix which is also a suffix of the sub-
pattern pat[0..i]

Pre-processing:
• We calculate values in lps[]. To do that, we keep track of the length of the
longest prefix suffix value (we use len variable for this purpose) for the
previous index
• We initialize lps[0] and len as 0
• If pat[len] and pat[i] match, we increment len by 1 and assign the
incremented value to lps[i].
• If pat[i] and pat[len] do not match and len is not 0, we update len to
lps[len-1]
• See computeLPSArray () in the above code for details
```

```
String matching:
1. We start the comparison of pat[j] with j = 0 with characters of the current
window of text.
2. We keep matching characters txt[i] and pat[j] and keep incrementing i and j
while pat[j] and txt[i] keep matching.
3. When we see a mismatch
• We know that character's pat[0..j-1] match with txt[i-j...i-1] (Note that j
starts with 0 and increments it only when there is a match).
• We also know (from the above definition) that lps[j-1] is the count of
characters of pat[0...j-1] that are both proper prefix and suffix.
• From the above two points, we can conclude that we do not need to match
these lps[j-1] characters with txt[i-j...i-1] because we know that these
characters will anyway match. Let us consider the above example to
understand this.
```

CODE:

```
#include <bits/stdc++.h>
using namespace std;
void computeLPSArray(char *pat, int M, int *lps);
void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int lps[M];
    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while ((N - i) >= (M - j))
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }
        if (j == M)
        {
            printf("Found pattern at index %d \n", i - j);
            j = lps[j - 1];
        }

        else if (i < N && pat[j] != txt[i])
        {
            if (j != 0)
                j = lps[j - 1];
            else
```

```
                i = i + 1;
        }
    }
}
void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;
    lps[0] = 0; // lps[0] is always 0
    int i = 1;
    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
int main()
{
    char txt[25];
    char pat[5];
    cout << "Enter text:";
    cin >> txt;
    cout << "Enter pattern:";
    cin >> pat;
    KMPSearch(pat, txt);
    return 0;
}
```

OUTPUT:

```
Enter text:abdneif
Enter pattern:ne
Found pattern at index 3


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

CONCLUSION:

- Thus, we implemented the code for string matching using Rabin Karp algorithm and KMP algorithm.