11/01/2024
Tuesday

* **Longest common subsequence**

→ is a subsequence that is common to both strings. Given two strings text1 and text2, return the length of their longest common subsequence. If their is no common subsequence, return 0.

I/P- text1 = "abcde", text2 = "ace"

O/P- 3

Explanation- Here, LCS is "ace" and its length is 3.

Approach- Base case → return 0 when there is no character in string.

If character matches -

a (b c)
a (c e)

ans = 1 + recursionKaAns

↳ adding this one because 1 char match ho gya.

If character doesn't matches -

CaseA → (a b c) / x (y z) a (b c) / (x y z) ← Case B

ans = 0 + max(caseA, caseB)

- Code using recursion

```
class Solution{
public:

int solveUsingRec (string str1, string str2, int i, int j){

    // base case
    if ( i >= str1.length()) return 0;
    if (j >= str2.length()) return 0;

    int ans = 0;
    // characters matched
    if (str1[i] == str2[j]){
      ans = 1 + solveUsingRec (str1, str2, i+1, j+1);
    }
    // characters not matched
    else {
      ans = 0 + max (solveUsingRec (str1, str2, i, j+1),
                     solveUsingRec (str1, str2, i+1, j));
    }
    return ans;
}

int LCS (string text1, string text2){
      int i = 0;
      int j = 0;
      return solveUsingRec (text1, text2, i, j);
}
};
```

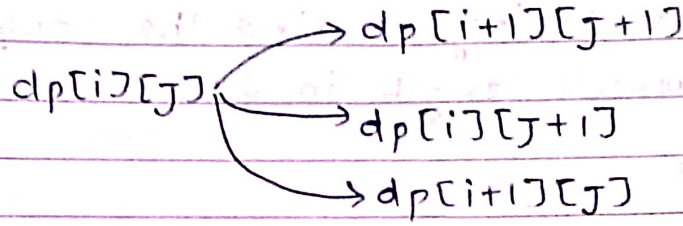
- Code using recursion + memoization

```
class Solution {
public:

int solveUsingMemo (string& str1, string& str2, int i,
                    int J, vector<vector<int>>& dp){

    //base case
    if (i >= str1.length()) return 0;
    if (J >= str2.length()) return 0;

    //if ans already exists,
    if (dp[i][J] != -1){
        return dp[i][J];
    }
    int ans = 0;
    if (str1[i] == str2[J]){
     ans = 1+ solveUsingRec (str1,str2, i+1,J+1, dp);
    }
    else{
     ans = 0+ max(solveUsingRec(str1,str2,i,J+1,dp),
                  solveUsingRec (str1,str2,i+1,J, dp));
    }
    //store & return using dp array
    dp[i][J] = ans;
    return dp[i][J];
}
int  LCS( string text1, string text2){
      int i= 0;
      int J = 0;
vector<vector<int>>dp(text1.length()+1,
                      vector<int>(text2.length()+1, -1));
return solveUsingMemo (text1, text2, i, J, dp);
}};
```

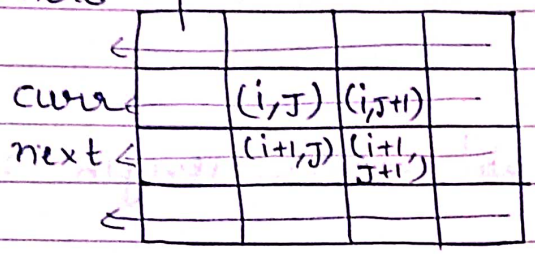
- **Code using tabulation**

```
class Solution{
public:

int solveUsingTabu(string& str1, string& str2){

vector<vector<int>>dp(str1.length()+1,
                      vector<int>(str2.length()+1, 0));

 for(int i=str1.length()-1; i>=0; i--){
    for(int j=str2.length()-1; j>=0; j--){
       int ans=0;
       if(str[i]==str[j]){
          ans=1+dp[i+1][j+1];
       }
       else{
          ans=0+max(dp[i][j+1], dp[i+1][j]);
       }
       dp[i][j]=ans;
    }
 }
  return dp[0][0];
}
int  LCS(string text1, string text2){

    return solveUsingTabu(text1, text2);
  }
};
```

from tabulation approach, we can see that



The final ans could form here ↓



So, we don't need the whole 2-D array every time. We just need is 2 1-D arrays and we can build the answer either row-wise or column-wise.

- **Optimising the code space-wise**

```
int solveUsingSO(string& str1, string& str2){
    vector<int> next(str2.length()+1, 0);
    vector<int> curr(str2.length()+1, 0);

    for(int i=str1.length()-1; i>=0; i--){
        for(int j=str2.length()-1; j>=0; j--){
            int ans = 0;
            if(str1[i]==str2[j]) ans = 1 + next[j+1];
            else ans = 0 + max(curr[j+1], next[j]);

            curr[j] = ans;
        } next = curr; // shifting
    }
    return curr[0];
}
```

* Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s.

I/P - s="cbbd"

O/P - 2

Explanation - Here, LPS is "bb" whose length is 2.

Approach - Given is string s. Let the reverse of string s be str

s, str → longest common subsequence

Find LCS of (s) and (str).

Code -

```
class Solution{
public:
// this function will find LCS of two strings
int findLCS (string& str1, string& str2){
    vector<int>next(str1.length()+1, 0);
    vector<int>curr(str1.length()+1, 0);
```

```
for(int i=str1.length()-1; i>=0; i--){
  for(int j=str2.length()-1; j>=0; j--){
      int ans=0;
      if (str1[i]==str2[j]){
          ans = 1+next[j+1];
      }
      else{
          ans = 0+max(curr[j+1], next[j]);
      }
      curr[j] = ans;
    }
    next=curr;
}
return curr[0];
}

int  LPS(string s){
        string str=s;
        reverse(str.begin(), str.end());
        return findLCS(s, str);
    }
};
```

* **Edit distance**

Given two strings word1 and word2, return min no. of operations required to convert word1 to word2.
Permitted operations - replace, insert, delete.

I/P - word1 = "horse", word2 = "course"

O/P - 2

Explanation - horse → corse (replace 'h' with 'c')
corse → course (insert 'c' b/w 'o' and 'r').

Approach → Base case

(1) word1 = h o r↑ (i)
word2 = h o r s e
↑ (J)

if (i == word1.length()). Here the only option we have is to insert the remaining words.
No. of words to be inserted is → (word2.length() - J)

(2) word1 = h o r s e
word2 = h o r↓ (J)

if (J == word2.length()). Here, the only option we have is to delete the remaining words.
No. of words to be deleted is → (word1.length() - i).

characters matched -

call recursion for (i+1, J+1)

Characters not matched -

replace → 1+ (i+1, J+1)
insert → 1+ (i, J+1)
remove → 1+ (i+1, J)
} minimum of these 3 operations

- Code using recursion

```
class Solution {
public:

int solveUsingRec (string& word1, string& word2, int i,
                   int J) {
    if (i == word1.length()) {
        return word2.length() - J;
    }
    if (J == word2.length()) {
        return word1.length() - i;
    }
    int ans = 0;
    if (word1[i] == word2[J]){
        ans = solveUsingRec (word1, word2, i+1, J+1) + 0;
    }
    else{
        int insert = 1+ solveUsingRec (word1, word2, i, J+1);
        int remove = 1+ solveUsingRec (word1, word2, i+1, J);
        int replace = 1+ solveUsingRec (word1, word2, i+1, J+1);
        ans = min (insert, min (remove, replace));
    }
    return ans; }
```

```
int minDistance(string word1, string word2){
        return solveUsingRec(word1, word2, 0, 0);
    }
};
```

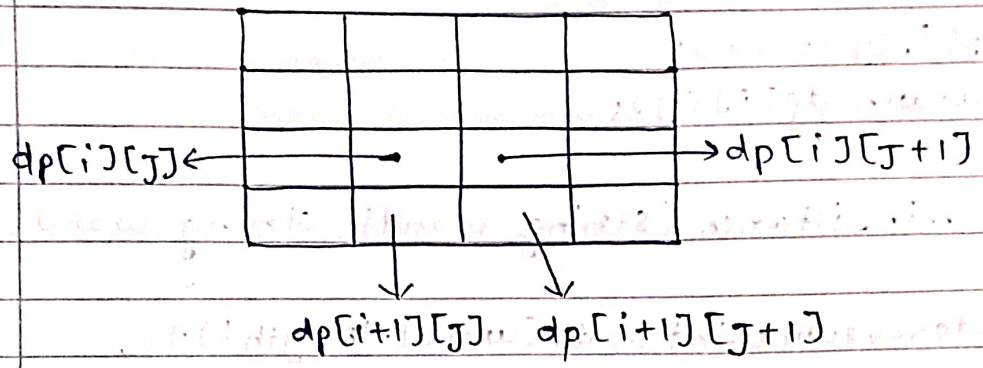
- **Code using recursion + memoization**

```
class Solution{
public:

int solveUsingMemo(string& word1, string& word2, int i,
                   int j, vector<vector<int>>& dp){
    if(i==word1.length()){
        return word2.length()-j;
    }
    if(j==word2.length()){
        return word1.length()-i;
    }
    //if ans already exists,
    if(dp[i][j] != -1){
        return dp[i][j];
    }
    int ans=0;
    if(word1[i]=word2[j]){
        ans=solveUsingMemo(word1, word2, i+1, j+1, dp);
    }
    else{
        int insert=1+solveUsingMemo(word1, word2, i, j+1, dp);
        int remove=1+solveUsingMemo(word1, word2, i+1, j, dp);
        int replace=1+solveUsingMemo(word1, word2, i+1, j+1, dp);
        ans=min(insert, min(remove, replace));
    }
```

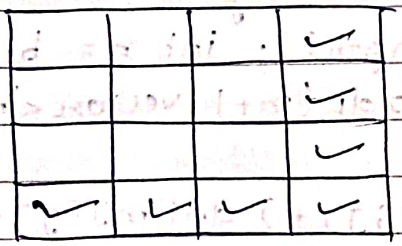
```
    dp[i][j] = ans;
    return dp[i][j];
  }
int minDistance (string word1, string word2){

vector<vector<int>> dp(word1.length()+1,
                       vector<int>(word2.length()+1, -1));
return solveUsingMemo(word1, word2, i, j, dp);
  }
};
```

- Code using Tabulation

```
int solveUsingTabu (string& a , string& b){
    int m= a.length(); int n= b.length();
vector<vector<int>>dp(m+1, vector<int>(n+1, -1));

for(int j=0; j<=n; j++) dp[m][j]= n-j;
for(int i=0; i<=m; i++) dp[i][n]= m-i;

for(int i=m-1; i>=0; i--){
   for(int j=n-1; j>=0; j--){
      int ans = 0;
      if(~~word~~ a[i] == b[j]){
          ans= 0 + dp[i+1][j+1];
      }
      else{
       int insert = 1+ dp[i][j+1];
       int remove = 1+ dp[i+1][j];
       int replace = 1+ dp[i+1][j+1];
      ans= min(insert, min(replace, remove));
      } dp[i][j] = ans;
 } } return dp[0][0]; }
```



for calculating answer of dp[i][j], we need to know the value of 3 cells i.e. dp[i][j+1], dp[i+1][j] and dp[i+1][j+1].
Overall, we need not the whole 2-D array, we need only 2 1-D arrays to calculate the answer of current position.



In tabulation approach, we are filling the last row and last column by analysing base cases to maintain initial state.

In space optimised, we are using 2 1-D arrays only: let them be next and curr.
So, we will fill the next array beforehand and everytime, when row is changed, will fill the last block of curr array.

- Space optimisation

```
int solveUsingSO (string& word1, string& word2){
    int m = word1.length();
    int n = word2.length();

    vector<int> next (n+1, -1);
    vector<int> curr (n+1, -1);

    for (int j=0; j<=n ; j++) {
        next[j] = n-j;
    }

    for(int i=m-1; i>=0; i--) {
      //har nyi row ke last block ko fill krna he
      curr[n] = m-i; //most important line
      for (int j=n-1; j>=0; j--){
        int ans = 0;
        if (word1[i]== word2[j]){
            ans = 0 + next[j+1];
        }
        else{
            int insert = 1+ curr[j+1];
            int remove = 1+ next[j];
            int replace = 1+ next[j+1];
            ans = min(insert, min(remove,replace));
        }
        curr[j]= ans;
      }
      next = curr; //shifting
    }
    return next[0];
}
```