Date : / /

DP on Strings

Longest Common Subsequence

What is a subsequence?

A subsequence is a sequence that you can form from a string by deleting 0 or more characters without changing the order of remaining characters.

Ex:- s = "abc"

Subsequences

"a", "b", "c", "ab", "ac", "bc", "abc", ""

Not Subsequences

"ca" ⇒ order changed (not allowed

"ba" ⇒ " "

If you pick characters in the same order as they appear in original string, it is a subsequence.

Date : / /

Longest Common Subsequence (LCS)

Return the length of longest sequence of characters that appear in both strings in the same order.

Ex:- s_1 = "abcde", s_2 = "ace"

Common subsequences are

"a", "c", "e", "ac", "ae", "ce", "ace".

Longest Common Subsequence = "ace" length = 3

(1) Express indexes

Generally f(2) ⇒ signifies string till index 2

Since we have 2 different string we'll be using 2 indexes f(ind1, ind2) (ind1 : s_1 ; ind2 : s_2)

f(2, 2) ⇒ Tells LCS of s_1(0-2) & s_2(0-2)

(2) Explore possibilities. ← match / not match

a) If characters match in both strings on indexes

Ex:- s_1 = a c d s_2 = a c e d

ind1 ↑ ind2 ↑

We can reduce the string to ac & ce & add 1 to our solution bcz we have a common subsequence with length = 1

if (s_1[ind1] == s_2[ind2]) = 1 + f(ind1 - 1, ind2 - 1)

Date : / /

b) ~~If~~ not matching we should move one of the index ~~if~~ to see if it will be matching

S_1 = ec S_2 = ce

ind1 ind2

~~If~~ we move ind2 to front & leave ind1 at the same place they may match or vicevena.

if notmatch $\Rightarrow 0 + \text{Max}(f(ind1-1, ind2), f(ind1, ind2-1))$

Base Case

~~If~~ ind goes negative

if (ind1 < 0 || ind2 < 0) return 0;

Recurrence

f(ind1, ind2) {

if (ind1 < 0 || ind2 < 0) return 0;

match → if (s1[ind1] == s2[ind2]) return 1 + f(ind1-1, ind2-1);

not match → return max(f(ind1-1, ind2), f(ind1, ind2-1));

Date : / /

Memoization

```
f(int ind1, int ind2, String s1, String s2, int[][] dp){
if(ind1<0 || ind2<0) return 0;
if(dp[ind1][ind2] != 0) return dp[ind1][ind2];
if(s1.charAt(ind1) == s2.charAt(ind2)){
return dp[ind1][ind2] = 1+f(ind1-1, ind2-1, s1, s2, dp);
}
return dp[ind1][ind2] = Math.max(
f(ind1-1, ind2, s1, s2, dp), f(ind1, ind2-1, s1, s2, dp));
}
```

main

```
int[][] dp = new int[n][m];
for(int i=0; i<n; i++){
Arrays.fill(dp[i], -1);}
return f(n-1, m-1, s, t, dp);
```

Date : / /

Tabulation

To use tabulation, there is a problem because we cannot write cases for negative in dp

dp[-1] ⇒ not possible

So we do shifting on index, currently We are reading indexes as

S_1 = a b c
ind 0 1 2
(n-1)

Now we read indexes as

S_1 = a b c
1 2 3
(n)

∴ For Base Case

i < 0 ⇒ to check if -1 ⇒ changed it to (i == 0)

dp = new int[n+1][m+1]

∴ Base Case ⇒

if (ind1 == 0 || ind2 == 0) return 0;

if (dp[i][j] != -1) return dp[i][j];

~~if (s[ind1] ==~~ if (s1[ind1 - 1] == s2[ind2 - 1]) {

return dp[ind1][ind2] = 1 + f(ind1 - 1, ind2 - 1, s1, s2, dp);

Date : / /

Tabulation

```
int [][] dp = new int[n+1][m+1];
for(int i=0; i<=m; i++) dp[0][i] = 0;
for(int j=0; j<=n; j++) dp[j][0] = 0;
for(int ind1=0; ind1<=n; ind1++) {
  for(int ind2=0; ind2<=m; ind2++) {
    if(s1[ind1-1] == s2[ind2-1])
      dp[ind1][ind2] = 1 + dp[ind1-1][ind2-1];
    else dp[ind1][ind2] = Math.max(
      dp[ind1-1][ind2], dp[ind1][ind2-1]);
  }
}
return dp[n][m];
```

Date : / /

Space Optimization

```
int[] prev = new int[m+1];
int[] curr = new int[m+1];
for(int i=0; i<=m; i++) prev[i]=0;
for(int ind1=1; ind1<=n; ind1++){
  for(int ind2=1; ind2<=m; ind2++){
  if(s1[ind1-1] == s2[ind2-1])
      curr[ind2] = 1 + prev[ind2-1];
  else
curr[ind2] = math.max(prev[ind2], curr[ind2-1]);
}
prev = curr;
}
return prev[m];
}
```

Date : / /

Printing the longest Common Subsequence

This solution works only if there is 1 longest common subsequence

To print the LCS you are supposed to trace back the path & see where is it the same string
The LCS is being created only when they match.
So we'll trace back till last & at every place where the solution is incrementing we'll add it to our solution string

```
i=n , j=m                    len = dp[n][m] for(i=0->len)
while(i>0 && j>0){                          s+="$";
  if(s1[i-1]==s2[j-1]){
```

So!

```
int len=dp[n][m];
string ans="";
for(int i=0;i<len;i++){
  ans+="$";}
int index=len-1;
int i=n, j=m;
while(i>0 && j>0){
  if(s[i-1]==t[i-1]){
    ans[index]=s[i-1];
    index--;
    i--, j--;}
  else if(dp[i-1][j]>dp[i][j-1]){
    i--;}
  else j--;
}
Sysout(ans);
```

// In else case it is check if we got the value from dp[ind1-1][ind2] or dp[ind1][ind2-1] => Since if not match case we are collecting Math.max of these two.

// If it is matching we are storing the character in string ans & reducing the index

// Since we are traversing from M-1 to 0 even in ans string we are storing at last i.e index = len-1;

Date : / /

Longest Common Substring

What is a substring?
A substring is a continuous part of a string
Characters must appear next to each other
No skipping is allowed.
Ex:- s = "substring"
Valid substrings are
"s", "sub", "ubst", "ing", "tring", "bstri"....
Invalid substrings are
"ss", "bing", "trg"....

Longest Common Substring
Return the length of longest common substring
Ex:- s1 = "abcjklp", s2 = "acjkp"
Ans :- 3 ⇒ (cjk)

In Longest Common Subsequence at the not match case i.e., Math.max(dp[ind1-1][ind2], dp[ind1][ind2-1]); we are moving to diff indexes if not match. But, in substring we only need consecutive characters so this wouldn't work.
The only thing we need to check is if it matches move to next. At the end return the max value in the dp.

Date : / /

Tabulation

```
int [][] dp = new int [n+1][m+1];
for(int i=0; i<=m; i++) dp[0][i]=0
for(int i=0; i<=n; i++) dp[i][0]=0;
int ans=0;
for(int i=1; i<=n; i++){
  for(int j=1; j<=m; j++){
    if(s1[i-1] == s2[j-1]){
      dp[i][j] = 1+dp[i-1][j-1];
      ans = max(ans, dp[i][j]);
    }
    else dp[i][j]=0;
  }
}
}
```

Date : / /

Space Optimization

```
int [] prev = new int[m+1];
int [] curr = new int[m+1];
int ans=0;
for(int i=1; i<=n; i++){
  for(int j=1; j<=m; j++){
    if(s[i-1]==t[j-1]){
      curr[j] = 1 + prev[j-1];
      ans = Math.max(ans, curr[j]);
    }
    else curr[j]=0;
  }
  prev=curr;
}
return ans;
}
```

Date : / /

Longest Palindromic Subsequence

A palindromic subsequence is a sequence that:
- comes from the string
- follows the order
- reads the same forward & backward.

Ex:- s = "agbcba", LongPalinSubsq = abcba

What happens if you reverse the string???

If a subsequence is a palindrome, then it is SAME even if you reverse it.

In s = agbcba ⇒ you pick abcba
In reverse(s) = abcbga ⇒ you can also pick abcba in order

The order is preserved even if original string is reversed because it is a palindrome.

- Every palindromic subsequence of s will automatically also be a subsequence of reverse(s)

s - agbcba reverse(s) - abcbga
longest subsequence = abcba (Palindrome)

Sol
// print longCommSubseq(s1, reverse(s1))

Date : / /

Minimum Insertions to make a string Palindrome

Turn a given string into a palindrome by inserting the minimum number of characters anywhere

Ex:- s = "abcaa"

Sol ⇒ @abcaa ⇒ Min = 2 insertion

Sol i) Keep the longest palindrome portion intact

abcaa ⇒ LPS = aaa

Which of the portion is left you can put it in string but in reverse way

Ex:- s = abcaa = aca

→ abacaba

Ex:- s = google ⇒ LPS = goog

→ reverse & add ⇒ elgoogle

s = apple ⇒ LPS = pp

→ elappale

s = banana ⇒ LPS = anana

→ bananab

Sol

return s1.size() - longestpalindromesubsequence(s1);

if they ask for Maximum operation then it would be len(s)

Ex:- s = abcaa

Sol = abcaa + rev(s)

= abcaaaacba

Sol

s.size() - LPS(s)

Ex:- banana

6 - 5 = 1

Date : / /

Minimum Operations to convert str1 → str2

Convert string1 to string2 with minimum no: of operations.

Ex :- str1 = abcd str2 = anc

Sol deletions = 2 (bd), addition = 1 (n) ⇒ Min operations = 3

Sol

1) Keep the longest common subsequence in both
   abcd , anc ⇒ LCS = ac
2) deletions = n − len(lcs)
   insertions = m − len(lc)

Total operation = n+m − 2×len(lc)

Sol return str.size() + ptr.size() − 2 * lcs(str, ptr);