Wild Card Matching - DP (VI) [s1 → pattern, s2 → text given]

(LC-44)

(s1) may have:
- ? → matches with single character
- * → matches with sequence of length 0 or more
- normal character

We have to match it with (s2) which has only lowercase characters.

---

e.g., (i) s1 = "?ay" ✓ matches
s2 "ray"

(ii) s1 = "ab*cd" * matches with de
s2 = "abdecd"
∴ s1 & s2 ✓ matches

(iii) s1 = "**abcd" ✓ matches
s2 = "abcd"

(iv) s1 = "ab?d" ✗ doesn't match
s2 = "abcc"

a b * c d

a b d e f c d

We don't know exactly how many characters we have to consider in place of '*'. So we have to try all possible ways.

↳ Recursion (string matching)

Rules to write recurrence :—

(i) Express string1 & string2 as (i, j) indexes

(ii) Explore comparisons.

(iii) out of all comparisons if any one returns TRUE, return true overall. (True & False recurrence common for all such probs).

---

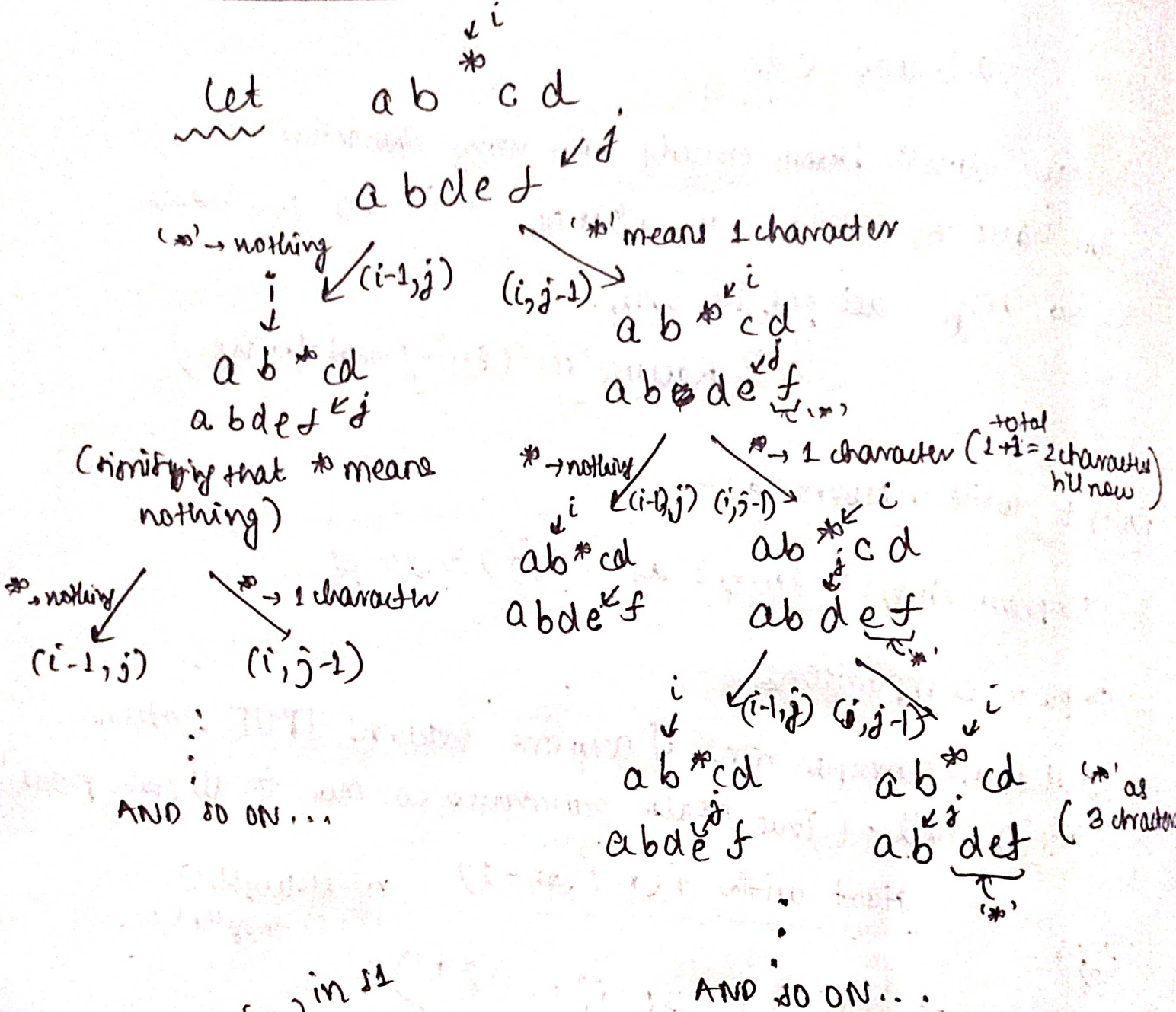
start with f(n-1, m-1)

n = s1.length()
m = s2.length()

```
f(i,j)
{   if (s1[i] == s[j] || s1[i] == '?')
        return f(i-1, j-1); // reduce indexes for both strings
```

∴ we can reduce both indexes/move forward only when both characters are same or s1[i] == '?'

Now for (*) condition:



So for a '*' in s1, either we consider it as in place of a single character in s2, or we don't consider it as anything, and continue the recursion.

Ultimately, as can be seen from the recursion tree, we will cover all the cases of '*' → 0 character, 1 character, 2 characters, ... etc.

'*' → include no character (zero) → $f(i-1, j)$

'*' → include one character → $f(i, j-1)$

Code:-

```
if (s1[i] == '*')
    return f(i-1, j) || f(i, j-1);
```

→ If s1[i] is neither matching with s2[j], nor s1[i] == '?' nor s1[i] == '*', then return FALSE;

→ BASE CASE analysis :-

(i) If index i in s1 gets exhausted, i.e., if (i < 0)
If s1 has no more characters to include, s2 must also have no more characters left.

∴ if (i<0 && j<0) return TRUE;
if (i<0 && j>=0) return FALSE;

(ii) If j index is string s2 gets exhausted, i.e., (j<0)

No more characters left in s2.

Now, in this case result can be TRUE iff all characters left in s1 are '*'.

Because '*' can be matched to empty string too.

```
if (j<0 && i>=0)
{
    while(i>=0)
    { if(s1[i] != '*')
        return FALSE;
      --i;
    }
    return TRUE;
}
```

(*) Recursive soln. → Time complexity (TC): exponential

Space complexity (SC): O(N+M)
(auxiliary stack space due to recursion)

If we memoize the solution, into a $n \times m$ matrix because we are working with indices here i & j

$i <= n-1$ and $j <= m-1$.

Memoization DP → TC: $O(N \times M)$ ← $N \times m$ different possible states

SC: $O(N \times M) + O(N+M)$

$O(N \times M)$ → dp matrix

$O(N+M)$ → stack space

FINAL MEMOIZATION CODE:- (let dp[m][n] be memoization matrix)

```
bool util( int i, int j, string &pattern, string &text)
{ if (i<0 && j<0) return true;
  if(i<0 && j>=0) return false;
  if(j<0 && i>=0)
  { while (i>=0)
    { if ( pattern[i] != '*' ) return false;
      --i;
    }
    return true;
  } if (dp[i][j] != -1) return dp[i][j];
if (pattern[i] == text[j] || pattern[i] == '?')
    return util(i-1, j-1, pattern, text);

if (pattern[i] == '*')
  return dp[i][j] = util(i-1, j, pattern, text) || util(i, j-1, pattern, text);

return dp[i][j] = false;
}
```

⭐ Tabulation DP :-

TC : O(N × M)
SC : O(N × M)

Step 1. Base case

Step 2. Write changing parameters (indexes → i & j)

Step 3. copy the recurrence relation

Step 1 : Analysing the base cases

For string matching tabulation in DP, we take the 1-based indexing in order to avoid negative index access.

```
>> vector<vector<bool>> dp(n+1, vector<int>(m+1, false));
```

(i) In memoization base case 1 :

```
if (i<0 and j<0) return true;
```

In tabulation base case 1 :

```
if (i==0 && j>=0) dp[i][j] = true;
```

OR

```
dp[0][0] = true;
```

(ii) Base case 2 :

memoization : if (i<0 && j>=0) return false.

tabulation : for i==0 and every value of j we have to set dp[0][j] = false;

(iii) Base case 3:

memoization: if (j<0 && i>=0)

check if pattern has only '*' left for any i>=0

tabulation: for(int i=1 ; i<=n; i++)

```
{ bool flag=true;
  for (int ii= i; ii>=1; i--)
    if ( pattern[ii-1] != '*') flag = false;
  dp[i][0] = flag;
}
```

Step 2 (and Step 3) Changing parameters 'i' and 'j' (and copying recurrence) code for (VERY SIMILAR TO MEMOIZATION CODE)

```
for (int i=1; i<=n; i++)
{ for (int j=1 ; j<=m ; j++)
  [
    if (pattern[i-1]==text[j-1] || pattern[i-1]=='?')
        dp[i][j] = dp[i-1][j-1];
    else if ( pattern[i]=='*')
        dp[i][j] = dp[i-1][j] + dp[i][j-1];
    else
        dp[i][j] = false;
  }
}

return dp[n][m];
```

⊛ Space optimisation in tabulation DP: —

In tabulation here, we deal only with the previous and current rows of the dp array, while writing the recurrence relation.

```
vector<bool> prev(m+1, false);
prev[0] = true; //in place of dp[0][0] = true (see Base case 1)

for (int j=1; j<=m; j++)  (see base case 2)
    prev[j] = false;
```

— x —

NOTE: For base case 3, for every row we were assigning the 0th column's value i.e., dp[i][0] = flag. according to the condition that '*' only is found in s1 or not.

Here we will do the same, but we will do this simultaneously with recurrence relations, at the starting of iteration of every row.

— x —

```
for(int i=1; i<=n; i++)
{ [redacted] bool flag=true;
  for(int ii=i; ii>=1; ii--)
    if(pattern[ii-1] != '*') {flag=false; break;}
  curr[0] = flag; //setting 0th column's value of current row
```

```
for (int j = 1; j <= m; j++)
{ if (pattern[i-1] == text[j-1] || pattern[i-1] == '?')
        curr[j] = prev[j-1];
  else if (pattern[i-1] == '*')
        curr[j] = prev[j] + curr[j-1];
  else
        curr[j] = false;
}
prev = curr;
}

return prev[m];
```

TC: O(N×M)

SC: O(N+M)