# Distinct Subsequences DP explanation



From LeetCode

> Given a string S and a string T, count the number of distinct subsequences of T in S.
>
> A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).
>
> Here is an example: S = "rabbbit", T = "rabbit"
>
> Return 3.

I see a very good DP solution, however, I have hard time to understand it, anybody can explain how this dp works?

```
int numDistinct(string S, string T) {

        vector<int> f(T.size()+1);

        //set the last size to 1.
        f[T.size()]=1;

        for(int i=S.size()-1; i>=0; --i){
            for(int j=0; j<T.size(); ++j){
                f[j]+=(S[i]==T[j])*f[j+1];
                printf("%d\t", f[j] );
            }
            cout<<"\n";
        }
        return f[0];
    }
```

algorithm dynamic-programming

edited May 13 at 4:58
Heuster
7,423 1 9 31

asked Dec 8 '13 at 21:21
J.W.
8,255 2 21 51

add a comment

## 2 Answers

First, try to solve the problem yourself to come up with a naive implementation:

Let's say that `S.length = m` and `T.length = n`. Let's write `S{i}` for the substring of `S` starting at `i`. For example, if `S = "abcde"`, `S{0} = "abcde"`, `S{4} = "e"`, and `S{5} = ""`. We use a similar definition for `T`.

Let `N[i][j]` be the distinct subsequences for `S{i}` and `T{j}`. We are interested in `N[0][0]` (because those are both full strings).

There are two easy cases: `N[i][n]` for any `i` and `N[m][j]` for `j<n`. How many subsequences are there for `""` in some string `S`? Exactly 1. How many for some `T` in `""`? Only 0.

Now, given some arbitrary `i` and `j`, we need to find a recursive formula. There are two cases.

If `S[i] != T[j]`, we know that `N[i][j] = N[i+1][j]` (I hope you can verify this for yourself, I aim to explain the cryptic algorithm above in detail, not this naive version).

If `S[i] = T[j]`, we have a choice. We can either 'match' these characters and go on with the next characters of both `S` and `T`, or we can ignore the match (as in the case that `S[i] != T[j]`). Since we have both choices, we need to add the counts there: `N[i][j] = N[i+1][j] + N[i+1][j+1]`.

In order to find `N[0][0]` using dynamic programming, we need to fill the `N` table. We first need to set the boundary of the table:

```
N[m][j] = 0, for 0 <= j < n
N[i][n] = 1, for 0 <= i <= m
```

Because of the dependencies in the recursive relation, we can fill the rest of the table looping `i` backwards and `j` forwards:

```
for (int i = m-1; i >= 0; i--) {
    for (int j = 0; j < n; j++) {
        if (S[i] == T[j]) {
            N[i][j] = N[i+1][j] + N[i+1][j+1];
        } else {
            N[i][j] = N[i+1][j];
        }
    }
}
```

We can now use the most important trick of the algorithm: we can use a 1-dimensional array `f`, with the invariant in the outer loop: `f = N[i+1];` This is possible because of the way the table is filled. If we apply this to my algorithm, this gives:

```
f[j] = 0, for 0 <= j < n
f[n] = 1

for (int i = m-1; i >= 0; i--) {
    for (int j = 0; j < n; j++) {
        if (S[i] == T[j]) {
            f[j] = f[j] + f[j+1];
        } else {
            f[j] = f[j];
        }
    }
}
```

We're almost at the algorithm you gave. First of all, we don't need to initialize `f[j] = 0`. Second, we don't need assignments of the type `f[j] = f[j]`.

Since this is `C++` code, we can rewrite the snippet

```
if (S[i] == T[j]) {
    f[j] += f[j+1];
}
```

to

```
f[j] += (S[i] == T[j]) * f[j+1];
```

and that's all. This yields the algorithm:

```
f[n] = 1

for (int i = m-1; i >= 0; i--) {
    for (int j = 0; j < n; j++) {
        f[j] += (S[i] == T[j]) * f[j+1];
    }
}
```

answered Dec 9 '13 at 7:43
Heuster
**7,423** 1 9 31

thanks for explanation, hope I can vote more times. – J.W. Dec 11 '13 at 4:06

can you explain this "N[i][n] = 1, for 0 <= i <= m"??? – S. H. Apr 28 at 0:17

@S.H. you can think of it as `for(int i = 0; i <= m; i++) { N[i][n] = 1; }` . The big difference is that that way is *operational*: I provide an 'algorithm' how to set the values, whereas the way in the post is *declarative*: I only care about the values, not about how to achieve them. That's a more mathematical way of writing it. – Heuster Apr 28 at 6:19

add a comment



I think the answer is wonderful, but something may be not correct.

I think we should iterate backwards over `i` and `j` . Then we change to array `N` to array `f` , we looping `j` forwards for not overlapping the result last got.

```
for (int i = m-1; i >= 0; i--) {
    for (int j = 0; j < n; j++) {
        if (S[i] == T[j]) {
            N[i][j] = N[i+1][j] + N[i+1][j+1];
        } else {
            N[i][j] = N[i+1][j];
        }
    }
}
```

edited Feb 8 at 2:36
jbaums
**5,249** 11 31

answered Feb 8 at 2:17
user2313762
**1**

I don't understand your final sentence. Could you please review what you have written and make sure it makes sense? – jbaums Feb 8 at 2:36

I was wrong The code written the two-dimensional array is also correct, both forwards and backwards for j is correct. but when we change two-dimensional array into one-dimensional array, we have to loop j forwards, we couldn't get the result in lastest looping(here is j+1), the result of last loop is stored in array f, and we have the "f[j] += (S[i] == T[j]) * f[j+1]", we loop j forwards so we make sure f[j+1] is not modfied when we calcute f[j]. – user2313762 Feb 8 at 3:08

add a comment

## Not the answer you're looking for? Browse other questions tagged algorithm dynamic-programming or ask your own question.