https://leetcode.com/problems/distinct-subsequences/

Given two strings s and t, return *the number of distinct **subsequences** of* s *which equals* t.

The test cases are generated so that the answer fits on a 32-bit signed integer.

Example 1:

Input: s = "rabbbit", t = "rabbit"

Output: 3

Explanation:

As shown below, there are 3 ways you can generate "rabbit" from s.

(The caret symbol ^ means the chosen letters)

```
rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^
```

Example 2:

Input: s = "babgbag", t = "bag"

Output: 5

Explanation:

As shown below, there are 5 ways you can generate "bag" from s.

(The caret symbol ^ means the chosen letters)

```
babgbag
^^ ^
babgbag
^^    ^
babgbag
^    ^^
babgbag
  ^  ^^
babgbag
```

```
    ^^^
```

Constraints:

- 1 <= s.length, t.length <= 1000
- s and t consist of English letters.

Attempt 1: 2023-07-04

Wrong Solution

Why Base condition 1 before Base condition 2 is wrong ?

Test out by:

Input: s = "rabbbit", t = "rabbit"

Expect Output: 3, Actual Output: 0

```java
class Solution {
    public int numDistinct(String s, String t) {
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, 0, t, 0);
    }

    private int helper(String s, int s_start, String t, int t_start) {
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(s_start == s.length()) {
            return 0;
        }
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return 1;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
```

```java
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count = helper(s, s_start + 1, t, t_start + 1) + helper(s, s_start + 1, t, t_start);
        } else {
            count = helper(s, s_start + 1, t, t_start);
        }
        return count;
    }
}
```

Because if Base condition 1 is ahead without additional limitation as "t_start < t.length()", then the Base condition 2 will never able to approach, it can be test out by print log in both Base condition 1 and 2, and we observe not able to touch Base condition 2, and only keep returning 0 after each recursion finish

The most tricky part is the correct condition for Base condition 1 below should be: if(t_start < t.length() && s_start == s.length()) {return 0;}, the additional condition as t_start < t.length() is a guarantee to make sure when scanning on s finished, at the same time, scanning on t not finish yet, otherwise if s and t both finished scanning at the same time, then both remain as "" empty string, which should return as 1

So there are two ways to avoid s and t finished scanning at same time and both remain empty string:

(1) Put Base condition 2 ahead of Base condition 1

(2) Keep Base condition 1 ahead of Base condition 2 but add additional condition as t_start < t.length()

Solution 1: Divide and Conquer (30 min, TLE 53/65)

Style 1: Base condition 1 ahead of Base condition 2 but additional condition as t_start < t.length()

```java
class Solution {
    public int numDistinct(String s, String t) {
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, 0, t, 0);
    }

    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start) {
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(t_start < t.length() && s_start == s.length()) {
            return 0;
        }
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return 1;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t
one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to
find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len -
1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1);
            count += helper(s, s_start + 1, t, t_start);
        } else {
            count = helper(s, s_start + 1, t, t_start);
        }
        return count;
    }
}
```

Style 2: Base condition 2 ahead of Base condition 1 then no additional condition required

```
class Solution {
    public int numDistinct(String s, String t) {
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, 0, t, 0);
    }

    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start) {
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
```

```java
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return 1;
        }
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(s_start == s.length()) {
            return 0;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1);
            count += helper(s, s_start + 1, t, t_start);
        } else {
            count = helper(s, s_start + 1, t, t_start);
        }
        return count;
    }
}
```

Divide and Conquer update with Memoization

Style 1: Use classic memo

```java
class Solution {
    public int numDistinct(String s, String t) {
        // +1 because index for s[0..s.length()] and t[0..t.length()] during recursion
        Integer[][] memo = new Integer[s.length() + 1][t.length() + 1];
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, 0, t, 0, memo);
    }

    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start, Integer[][] memo) {
        if(memo[s_start][t_start] != null) {
            return memo[s_start][t_start];
        }
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return memo[s_start][t_start] = 1;
        }
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(s_start == s.length()) {
            return memo[s_start][t_start] = 0;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1, memo);
            count += helper(s, s_start + 1, t, t_start, memo);
        } else {
            count = helper(s, s_start + 1, t, t_start, memo);
        }
        return memo[s_start][t_start] = count;
    }
}

==========================================================================================================
Refer to
https://leetcode.com/problems/distinct-subsequences/solutions/2738744/recursion-to-dp-optimise-easy-understanding/

class Solution {
    public int numDistinct(String s, String t) {
        int[][] memo = new int[s.length() + 1][t.length()+1];
        for (int i = 0; i < s.length()+1; i++) {
            Arrays.fill(memo[i], -1);
        }
       return topTobottom(memo,s,t,s.length(), t.length());
    }
    private int topTobottom(int[][] memo, String s, String t, int i, int j) {
        if (memo[i][j] != -1) return memo[i][j];
        if (j==0)  memo[i][j] = 1;
        else if (i == 0) memo[i][j] = 0;
        else {
            int sol1 = topTobottom(memo, s, t,i-1, j);
```

```java
                int sol2 = 0;
                if (s.charAt(i-1) == t.charAt(j-1)) sol2 = topTobottom(memo, s, t, i-1, j-1);
                memo[i][j]= sol1 + sol2;
            }
            return memo[i][j];
        }
}
```

Style 2: Use String as key in HashMap to create memo

```java
class Solution {
    public int numDistinct(String s, String t) {
        Map<String, Integer> memo = new HashMap<String, Integer>();
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, 0, t, 0, memo);
    }
    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start, Map<String, Integer> memo) {
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return 1;
        }
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(s_start == s.length()) {
            return 0;
        }
        String key = s_start + "_" + t_start;
        if(memo.containsKey(key)) {
            return memo.get(key);
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1, memo);
            count += helper(s, s_start + 1, t, t_start, memo);
        } else {
            count = helper(s, s_start + 1, t, t_start, memo);
        }
        memo.put(key, count);
        return count;
    }
}
```

Refer to
https://leetcode.wang/leetcode-115-Distinct-Subsequences.html

解法一 递归之分治

S 中的每个字母就是两种可能选他或者不选他。我们用递归的常规思路，将大问题化成小问题，也就是分治的思想。

如果我们求 S[0，S_len - 1] 中能选出多少个 T[0，T_len - 1]，个数记为 n。那么分两种情况，

- S[0] == T[0]，需要知道两种情况
    - 从 S 中选择当前的字母，此时 S 跳过这个字母, T 也跳过一个字母。去求 S[1，S_len - 1] 中能选出多少个 T[1，T_len - 1]，个数记为 n1
    - S 不选当前的字母，此时S跳过这个字母，T 不跳过字母。去求S[1，S_len - 1] 中能选出多少个 T[0，T_len - 1]，个数记为 n2
- S[0] ！ = T[0]

S 只能不选当前的字母，此时S跳过这个字母， T 不跳过字母。去求S[1，S_len - 1] 中能选出多少个 T[0，T_len - 1]，个数记为 n1
也就是说如果求 S[0，S_len - 1] 中能选出多少个 T[0，T_len - 1]，个数记为 n。转换为数学式就是

```
if(S[0] == T[0]){
    n = n1 + n2;
}else{
    n = n1;
}
```

推广到一般情况，我们可以先写出递归的部分代码。

```java
public int numDistinct(String s, String t) {
    return numDistinctHelper(s, 0, t, 0);
}
private int numDistinctHelper(String s, int s_start, String t, int t_start) {
    int count = 0;
    //当前字母相等
    if (s.charAt(s_start) == t.charAt(t_start)) {
        //从 S 选择当前的字母，此时 S 跳过这个字母, T 也跳过一个字母。
        count = numDistinctHelper(s, s_start + 1, t, t_start + 1, map)
        //S 不选当前的字母，此时 S 跳过这个字母，T 不跳过字母。
                + numDistinctHelper(s, s_start + 1, t, t_start,  map);
    //当前字母不相等
    }else{
       //S 只能不选当前的字母，此时 S 跳过这个字母， T 不跳过字母。
       count = numDistinctHelper(s, s_start + 1, t, t_start,  map);
    }
    return count;
}
```

递归出口的话，因为我们的S和T的开始下标都是增长的。
如果S[s_start, S_len - 1]中， s_start 等于了 S_len ，意味着S是空串，从空串中选字符串T，那结果肯定是0。
如果T[t_start, T_len - 1]中，t_start等于了 T_len，意味着T是空串，从S中选择空字符串T，只需要不选择 S 中的所有字母，所以选法是1。

综上，代码总体就是下边的样子

```java
public int numDistinct(String s, String t) {
    return numDistinctHelper(s, 0, t, 0);
}
private int numDistinctHelper(String s, int s_start, String t, int t_start) {
    //T 是空串，选法就是 1 种
    if (t_start == t.length()) {
        return 1;
    }
    //S 是空串，选法是 0 种
    if (s_start == s.length()) {
        return 0;
    }
    int count = 0;
    //当前字母相等
    if (s.charAt(s_start) == t.charAt(t_start)) {
        //从 S 选择当前的字母，此时 S 跳过这个字母, T 也跳过一个字母。
        count = numDistinctHelper(s, s_start + 1, t, t_start + 1)
        //S 不选当前的字母，此时 S 跳过这个字母，T 不跳过字母。
              + numDistinctHelper(s, s_start + 1, t, t_start);
    //当前字母不相等
    }else{
        //S 只能不选当前的字母，此时 S 跳过这个字母， T 不跳过字母。
        count = numDistinctHelper(s, s_start + 1, t, t_start);
    }
    return count;
}
```

遗憾的是，这个解法对于如果S太长的 case 会超时。

原因就是因为递归函数中，我们多次调用了递归函数，这会使得我们重复递归很多的过程，解决方案就很简单了，
Memoization 技术，把每次的结果利用一个map保存起来，在求之前，先看map中有没有，有的话直接拿出来就可以了。
map的key的话就标识当前的递归，s_start 和 t_start 联合表示，利用字符串 s_start + '@' + t_start。
value的话就保存这次递归返回的count。

```java
public int numDistinct(String s, String t) {
    HashMap<String, Integer> map = new HashMap<>();
    return numDistinctHelper(s, 0, t, 0, map);
}
private int numDistinctHelper(String s, int s_start, String t, int t_start,
HashMap<String, Integer> map) {
    //T 是空串，选法就是 1 种
    if (t_start == t.length()) {
        return 1;
    }
    //S 是空串，选法是 0 种
    if (s_start == s.length()) {
        return 0;
    }
    String key = s_start + "@" + t_start;
    //先判断之前有没有求过这个解
    if (map.containsKey(key)) {
        return map.get(key);
    }
    int count = 0;
    //当前字母相等
    if (s.charAt(s_start) == t.charAt(t_start)) {
        //从 S 选择当前的字母，此时 S 跳过这个字母, T 也跳过一个字母。
        count = numDistinctHelper(s, s_start + 1, t, t_start + 1, map)
        //S 不选当前的字母，此时 S 跳过这个字母，T 不跳过字母。
              + numDistinctHelper(s, s_start + 1, t, t_start, map);
    //当前字母不相等
    }else{
        //S 只能不选当前的字母，此时 S 跳过这个字母， T 不跳过字母。
        count = numDistinctHelper(s, s_start + 1, t, t_start, map);
    }
    //将当前解放到 map 中
    map.put(key, count);
    return count;
}
```

Solution 2: Recursion (30 min, TLE 53/65)

Difference between Recursion and Divide and Conquer:

1.Recursion as void return, Divide and Conquer as actual return

2.Recursion has global variable, Divide and Conquer only local variable

3.Usually in Recursion (base condition -> process on current level -> recursive into smaller problem), in Divide and Conquer (base condition -> recursive into smaller problem -> process on bottom level then return to parent level), but the difference is not significant in this L115, the base condition narrative is a bit different

```java
class Solution {
    int count = 0;
    public int numDistinct(String s, String t) {
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        helper(s, 0, t, 0);
        return count;
    }

    private void helper(String s, int s_start, String t, int t_start) {
        // Base condition 2:
        // For t_start == t_len - 1, which means all selected chars from s build up t,
        // it also means we find one solution, we can use global variable as 'count'
        // to record one solution found, as count++, then return to previous recursion
        // level to find more solutions
        if(t_start == t.length()) {
            count++;
            return;
        }
        // Base condition 1:
        // For s_start == s_len - 1, which means s reach the end, but for now
        // t_start != t_len - 1, hence no combination of selected chars from s
        // can build up t, directly return without update global variable 'count'
        if(s_start == s.length()) {
            return;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], then move index in both s and t one
step ahead
        // Case 2: If s[s_start] != t[t_start], then only move index in s one step
ahead as skip
        // current char s[s_start], no move for t_start
        if(s.charAt(s_start) == t.charAt(t_start)) {
            helper(s, s_start + 1, t, t_start + 1);
        }
        helper(s, s_start + 1, t, t_start);
    }
}
```

Recursion update with Memoization (Memoization on global variable is new skill)

```
class Solution {
    int count = 0;
    public int numDistinct(String s, String t) {
        Integer[][] memo = new Integer[s.length() + 1][t.length() + 1];
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        helper(s, 0, t, 0, memo);
        return count;
    }

    private void helper(String s, int s_start, String t, int t_start, Integer[][] memo)
 {
        // Base condition 2:
        // For t_start == t_len - 1, which means all selected chars from s build up t,
        // it also means we find one solution, we can use global variable as 'count'
        // to record one solution found, as count++, then return to previous recursion
        // level to find more solutions
        if(t_start == t.length()) {
```

```java
            count++;
            return;
        }
        // Base condition 1:
        // For s_start == s_len - 1, which means s reach the end, but for now
        // t_start != t_len - 1, hence no combination of selected chars from s
        // can build up t, directly return without update global variable 'count'
        if(s_start == s.length()) {
            return;
        }
        // Different than Divide and Conquer Memoization strategy since 'count'
        // here is global variable, we cannot store each recursion local 'count'
        // into memo, and we also cannot store global 'count' in reach recursion,
        // instead, we only store variation between each recursion
        if(memo[s_start][t_start] != null) {
            count += memo[s_start][t_start];
            return;
        }
        // Record previous global 'count' before moving on to next level recursion
        int count_pre = count;
        // Divide
        // Case 1: If s[s_start] == t[t_start], then move index in both s and t one step ahead
        // Case 2: If s[s_start] != t[t_start], then only move index in s one step ahead as skip
        // current char s[s_start], no move for t_start
        if(s.charAt(s_start) == t.charAt(t_start)) {
            helper(s, s_start + 1, t, t_start + 1, memo);
        }
        helper(s, s_start + 1, t, t_start, memo);
        // After finishing next level recursion we will get a new global 'count',
        // the memo only store variation between each recursion
        int diff = count - count_pre;
        memo[s_start][t_start] = diff;
    }
}
```

Refer to

https://leetcode.wang/leetcode-115-Distinct-Subsequences.html

解法二 递归之回溯

回溯的思想就是朝着一个方向找到一个解，然后再回到之前的状态，改变当前状态，继续尝试得到新的解。可以类比于二叉树的DFS，一路走到底，然后回到之前的节点继续递归。对于这道题，和二叉树的DFS很像了，每次有两个可选的状态，选择S串的当前字母和不选择当前字母。当S串的当前字母和T串的当前字母相等，我们就可以选择S的当前字母，进入递归。递归出来以后，继续尝试不选择S的当前字母，进入递归。
代码可以是下边这样。

```java
public int numDistinct3(String s, String t) { 
    numDistinctHelper(s, 0, t, 0);
}

private void numDistinctHelper(String s, int s_start, String t, int t_start) {
    //当前字母相等，选中当前 s 的字母，s_start 后移一个
    //选中当前 s 的字母，意味着和 t 的当前字母匹配，所以 t_start 后移一个
    if (s.charAt(s_start) == t.charAt(t_start)) {
        numDistinctHelper(s, s_start + 1, t, t_start + 1);
    }
    //出来以后，继续尝试不选择当前字母，s_start 后移一个，t_start 不后移
    numDistinctHelper(s, s_start + 1, t, t_start);
}
```

递归出口的话，就是两种了。

- 当t_start == T_len，那么就意味着当前从S中选择的字母组成了T，此时就代表一种选法。我们可以用一个全局变量count，count计数此时就加一。然后return，返回到上一层继续寻求解。
- 当s_start == S_len，此时S到达了结尾，直接 return。

```java
int count = 0;
public int numDistinct(String s, String t) { 
    numDistinctHelper(s, 0, t, 0);
    return count;
}
private void numDistinctHelper(String s, int s_start, String t, int t_start) {
    if (t_start == t.length()) {
        count++;
        return;
    }
    if (s_start == s.length()) {
        return;
    }
    //当前字母相等，s_start 后移一个，t_start 后移一个
    if (s.charAt(s_start) == t.charAt(t_start)) {
        numDistinctHelper(s, s_start + 1, t, t_start + 1);
    }
    //出来以后，继续尝试不选择当前字母，s_start 后移一个，t_start 不后移
    numDistinctHelper(s, s_start + 1, t, t_start);
}
```

好吧，这个熟悉的错误又出现了，同样是递归中调用了两次递归，会重复计算一些解。怎么办呢？Memoization 技术。map的key和之前一样，标识当前的递归，s_start 和 t_start 联合表示，利用字符串 s_start + '@' + t_start。

map的value的话？存什么呢。区别于解法一，我们每次都得到了当前条件下的count，然后存起来了。而现在我们只有一个全局变量，该怎么办呢？存全局变量count吗？

如果递归过程中

```
if (map.containsKey(key)) {
    ... ...
}
```

遇到了已经求过的解该怎么办呢？

我们每次得到一个解后增加全局变量count，所以我们map的value存两次递归后 count 的增量。这样的话，第二次遇到同样的情况的时候，就不用递归了，把当前增量加上就可以了。

```
if (map.containsKey(key)) {
    count += map.get(key);
    return;
}
```

综上，代码就出来了

```java
int count = 0;
public int numDistinct(String s, String t) {
    HashMap<String, Integer> map = new HashMap<>();
    numDistinctHelper(s, 0, t, 0, map);
    return count;
}

private void numDistinctHelper(String s, int s_start, String t, int t_start,
            HashMap<String, Integer> map) {
    if (t_start == t.length()) {
        count++;
        return;
    }
    if (s_start == s.length()) {
        return;
    }
    String key = s_start + "@" + t_start;
    if (map.containsKey(key)) {
        count += map.get(key);
        return;
    }
    int count_pre = count;
    //当前字母相等，s_start 后移一个，t_start 后移一个
    if (s.charAt(s_start) == t.charAt(t_start)) {
        numDistinctHelper(s, s_start + 1, t, t_start + 1, map);
    }
    //出来以后，继续尝试不选择当前字母，s_start 后移一个，t_start 不后移
    numDistinctHelper(s, s_start + 1, t, t_start, map);
    //将增量存起来
    int count_increment = count - count_pre;
    map.put(key, count_increment);
}
```

Solution 3: DP (-- min)

Recursion to DP evolution lecture

1.基本文献：

Refer to

https://leetcode.wang/leetcode-115-Distinct-Subsequences.html

解法三 动态规划

让我们来回想一下解法一做了什么。s_start 和 t_start 不停的增加，一直压栈，压栈，直到

```
//T 是空串，选法就是 1 种
if (t_start == t.length()) {
    return 1;
}
//S 是空串，选法是 0 种
if (s_start == s.length()) {
    return 0;
}
```

T 是空串或者 S 是空串，我们就直接可以返回结果了，接下来就是不停的出栈出栈，然后把结果通过递推关系取得。

递归的过程就是由顶到底再回到顶。

动态规划要做的就是去省略压栈的过程，直接由底向顶。

这里我们用一个二维数组 dp[m][n] 对应于从 S[m，S_len) 中能选出多少个 T[n，T_len)。

当 m == S_len，意味着S是空串，此时dp[S_len][n]，n 取 0 到 T_len - 1的值都为 0。

当 n == T_len，意味着T是空串，此时dp[m][T_len]，m 取 0 到 S_len的值都为 1。

然后状态转移的话和解法一分析的一样。如果求dp[s][t]。

- S[s] == T[t]，当前字符相等，那就对应两种情况，选择S的当前字母和不选择S的当前字母dp[s][t] = dp[s+1][t+1] + dp[s+1][t]
- S[s] != T[t]，只有一种情况，不选择S的当前字母dp[s][t] = dp[s+1][t]

代码就可以写了。

```java
public int numDistinct(String s, String t) {
    int s_len = s.length();
    int t_len = t.length();
    int[][] dp = new int[s_len + 1][t_len + 1];
    //当 T 为空串时，所有的 s 对应于 1
    for (int i = 0; i <= s_len; i++) {
        dp[i][t_len] = 1;
    }
    //倒着进行，T 每次增加一个字母
    for (int t_i = t_len - 1; t_i >= 0; t_i--) {
        dp[s_len][t_i] = 0; // 这句可以省去，因为默认值是 0
        //倒着进行，S 每次增加一个字母
        for (int s_i = s_len - 1; s_i >= 0; s_i--) {
            //如果当前字母相等
            if (t.charAt(t_i) == s.charAt(s_i)) {
                //对应于两种情况，选择当前字母和不选择当前字母
                dp[s_i][t_i] = dp[s_i + 1][t_i + 1] + dp[s_i + 1][t_i];
            //如果当前字母不相等
            } else {
                dp[s_i][t_i] = dp[s_i + 1][t_i];
            }
        }
    }
    return dp[0][0];
}
```

对比于解法一和解法二，如果Memoization 技术我们不用hash，而是用一个二维数组，会发现其实我们的递归过程，其实就是在更新下图中的二维表，只不过更新的顺序没有动态规划这么归整。这也是不用Memoization 技术会超时的原因，如果把递归的更新路线画出来，会发现很多路线重合了，意味着我们进行了很多没有必要的递归，从而造成了超时。

我们画一下动态规划的过程。
S = "babgbag", T = "bag"
T 为空串时，所有的 s 对应于 1。 S 为空串时，所有的 t 对应于 0。

此时我们从

dp[6][2] 开始求。根据公式，因为当前字母相等，所以
dp[6][2] = dp[7][3] + dp[7][2] = 1 + 0 = 1 。
接着求
dp[5][2]，当前字母不相等，
dp[5][2] = dp[6][2] = 1。
一直求下去。

求当前问号的地方的值的时候，我们只需要它的上一个值和斜对角的值。换句话讲，求当前列的时候，我们只需要上一列的信息。比如当前求第1列，第3列的值就不会用到了。

所以我们可以优化算法的空间复杂度，不需要二维数组，需要一维数组就够了。
此时需要解决一个问题，就是当求上图的dp[1][1]的时候，需要dp[2][1]和dp[2][2]的信息。但是如果我们是一维数组，dp[2][1]之前已经把dp[2][2]的信息覆盖掉了。所以我们需要一个pre变量保存之前的值。

```java
public int numDistinct(String s, String t) {
    int s_len = s.length();
    int t_len = t.length();
    int[]dp = new int[s_len + 1];
    for (int i = 0; i <= s_len; i++) {
        dp[i] = 1;
    }
  //倒着进行，T 每次增加一个字母
    for (int t_i = t_len - 1; t_i >= 0; t_i--) {
        int pre = dp[s_len];
        dp[s_len] = 0;
         //倒着进行，S 每次增加一个字母
        for (int s_i = s_len - 1; s_i >= 0; s_i--) {
            int temp = dp[s_i];
            if (t.charAt(t_i) == s.charAt(s_i)) {
                dp[s_i] = dp[s_i + 1] + pre;
            } else {
                dp[s_i] = dp[s_i + 1];
            }
```

```java
                pre = temp;
            }
        }
        return dp[0];
}
```

利用temp和pre两个变量实现了保存之前的值。

其实动态规划优化空间复杂度的思想，在 5题，10题，53题，72题 等等都已经用了，是非常经典的。**上边的动态规划是从字符串末尾倒着进行的，其实我们只要改变dp数组的含义，用dp[m][n]表示S[0,m)和T[0,n)，然后两层循环我们就可以从**
1 往末尾进行了，思想是类似的，leetcode 高票答案也都是这样的，如果理解了上边的思想，代码其实也很好写。这里只分享下代码吧。

```java
public int numDistinct(String s, String t) {
    int s_len = s.length();
    int t_len = t.length();
    int[] dp = new int[s_len + 1];
    for (int i = 0; i <= s_len; i++) {
        dp[i] = 1;
    }
    for (int t_i = 1; t_i <= t_len; t_i++) {
        int pre = dp[0];
        dp[0] = 0;
        for (int s_i = 1; s_i <= s_len; s_i++) {
            int temp = dp[s_i];
            if (t.charAt(t_i - 1) == s.charAt(s_i - 1)) {
                dp[s_i] = dp[s_i - 1] + pre;
            } else {
                dp[s_i] = dp[s_i - 1];
            }
            pre = temp;
        }
    }
    return dp[s_len];
}
```

总结：这道题太经典了，从递归实现回溯，递归实现分治，
Memoization 技术对递归的优化，从递归转为动态规划再到动态规划空间复杂度的优化，一切都是理所当然，不需要什么特殊技巧，一切都是这么优雅，太棒了。

自己一开始是想到回溯的方法，然后卡到了超时的问题上，看了这篇 和 这篇 的题解后才恍然大悟，一切才都联通了，解法一、解法二、解法三其实本质都是在填充那个二维矩阵，最终殊途同归，不知为什么脑海中有宇宙大爆炸，然后万物产生联系的画面，2333。
这里自己需要吸取下教训，自己开始在回溯卡住了以后，思考了动态规划的方法，dp数组的含义已经定义出来了，想状态转移方程的时候在脑海里一直想，又卡住了。所以对于这种稍微复杂的动态规划还是拿纸出来画一画比较好。

---

2. 递归从正反两条路线进化到DP的思路和区别：

中文文献中二维DP图从右下角开始，多出来的最后一列代表T空串时子串个数状态，多出来的最后一列代表S空串时字串个数状态，反推到左上角代表最终状态的体系，一脉相承于解法一和解法二构建的扫描S和T的时候坐标从0增长到自身长度并剩下空字符串的体系，该体系中递归解法（解法一和解法二）的终止条件（base condition）完全符合它们如何进化到解法三DP解法的预期，而在leetcode讨论中的高赞英文样例中二维DP图从左上角开始，多出来的第一列（或第一行，参见Solution 3 DP style 2）代表T空串时子串个数状态，多出来的第一行（或第一列，参见Solution 3 DP style 2）代表S空串时字串个数状态，正推到右下角代表最终状态的体系，则是因为把S和T空串情况放在最开始考虑所得。

自此我们把从二维状态表右下角反推到左上角的递归和优化后的DP解称为"逆向"，把从二维状态表左上角正推到右下角的递归和优化后的DP解称为"正向"

回顾前文中文文献中的关键定义：
T 是空串或者 S 是空串，我们就直接可以返回结果了，接下来就是不停的出栈出栈，然后把结果通过递推关系取得。
递归的过程就是由顶到底再回到顶。
动态规划要做的就是去省略压栈的过程，直接由底向顶。

此刻我们抛出最关键的问题：底是什么？顶又是什么？在本题中如何定义？

(1) 基于上述文献的逆向递归进化到DP的思路

在上述中文文献中，从递归的角度讲，"顶"就是递归最开始在主体方法中被呼叫的状态，本题中就是m == 0 和 n == 0 时，"底"在本题中就是当 递归到达m == s_len 和 n == t_len 时，也就是递归实际方法中的base condition，递归就是先从"顶"即m == 0 和 n == 0逐层到达"底"即m == s_len 和 n == t_len，然后在到达"底"后再通过返回语句逐层从"底"返回到"顶"，而DP能够省略掉递归中"从顶到底"的过程，而"直接由底向顶"，这也意味着从二维数组DP状态表的角度讲，从右下角逆推到左上角的过程，也就是m == s_len(底) --> m == 0(顶)，n == t_len(底) --> n == 0(顶)的过程

第一步：实现一个基本递归(逆向版本)：
在递归的过程就是由顶到底再回到顶

递归中由顶到底的过程：
我们的递归始于m == 0和n == 0时，m == 0(顶) --> m == s_len(底)，n == 0(顶) --> n == t_len(底)，然后在到底的时候触碰到base condition开启return返回过程

递归中再由底回到顶的过程：
在从顶到底并触碰到base condition开启return之后，逐层返回，m == s_len(底) --> m == 0(顶)，n == t_len(底) --> n == 0(顶)，此时最终状态实际上在顶，也就是m == 0和n == 0时取得，和二维DP中最终状态在左上角[0, 0]处获得形成一致

```
Style 1: Base condition 1 ahead of Base condition 2 but additional condition as t_start < t.length()

class Solution {
    public int numDistinct(String s, String t) {
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        // 从顶m == 0和n == 0开始递归
        return helper(s, 0, t, 0);
    }

    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start) {
        // 在底m == s_len和n == t_len触底开启逐层返回到顶过程
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(t_start < t.length() && s_start == s.length()) {
            return 0;
        }
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return 1;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1);
            count += helper(s, s_start + 1, t, t_start);
        } else {
            count = helper(s, s_start + 1, t, t_start);
        }
        return count;
    }
}

===============================================================================================
Style 2: Base condition 2 ahead of Base condition 1 then no additional condition required

class Solution {
    public int numDistinct(String s, String t) {
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        // 从顶m == 0和n == 0开始递归
        return helper(s, 0, t, 0);
    }

    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start) {
        // 在底m == s_len和n == t_len触底开启逐层返回到顶过程
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return 1;
        }
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(s_start == s.length()) {
            return 0;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len -
1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step
ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to
find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len -
1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1);
            count += helper(s, s_start + 1, t, t_start);
        } else {
            count = helper(s, s_start + 1, t, t_start);
        }
        return count;
    }
}
```

第二步：递归配合Memoization(逆向版本)：

```
Style 1: Use classic memo

class Solution {
    public int numDistinct(String s, String t) {
        // +1 because index for s[0..s.length()] and t[0..t.length()] during recursion
        Integer[][] memo = new Integer[s.length() + 1][t.length() + 1];
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, 0, t, 0, memo);
    }
    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start, Integer[][] memo)
{
        if(memo[s_start][t_start] != null) {
            return memo[s_start][t_start];
        }
```

```java
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return memo[s_start][t_start] = 1;
        }
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(s_start == s.length()) {
            return memo[s_start][t_start] = 0;
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1, memo);
            count += helper(s, s_start + 1, t, t_start, memo);
        } else {
            count = helper(s, s_start + 1, t, t_start, memo);
        }
        return memo[s_start][t_start] = count;
    }
}
```

```
===========================================================================================
Style 2: Use String as key in HashMap to create memo

class Solution {
    public int numDistinct(String s, String t) {
        Map<String, Integer> memo = new HashMap<String, Integer>();
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, 0, t, 0, memo);
    }
    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_start, String t, int t_start, Map<String, Integer> memo) {
        // Base condition 2:
        // For t[t_start, t_len - 1], t_start == t_len - 1, which means t is empty
        // string now, to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_start == t.length()) {
            return 1;
        }
        // Base condition 1:
        // For s[s_start, s_len - 1], s_start == s_len - 1, which means s is empty
        // string now, to pick chars from empty string, no choice, hence return 0
        if(s_start == s.length()) {
            return 0;
        }
        String key = s_start + "_" + t_start;
        if(memo.containsKey(key)) {
            return memo.get(key);
        }
        // Divide
        // Case 1: If s[s_start] == t[t_start], we have two options:
        // (1) Use current char as s[s_start] in s, then move index in both s and t one step ahead
        // e.g if s[0] == t[0], if use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[1, t_len - 1], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] == t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_start] != t[t_start], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[0] != t[0], not use s[0], in next recursion level, we are going to find how many distinct sequence can be found in s[1, s_len - 1] equal to t[0, t_len - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_start) == t.charAt(t_start)) {
            count += helper(s, s_start + 1, t, t_start + 1, memo);
            count += helper(s, s_start + 1, t, t_start, memo);
        } else {
            count = helper(s, s_start + 1, t, t_start, memo);
        }
        memo.put(key, count);
        return count;
    }
}
```

第三步：基于递归的2D DP(逆向版本)：
DP能够省略掉递归中"从顶到底"的过程，而"直接由底向顶"，这也意味着从二维数组DP状态表的角度讲，从右下角逆推到左上角的过程，也就是m == s_len(底) --> m == 0(顶)，n == t_len(底) --> n == 0(顶)的过程

这里我们用一个二维数组 dp[m][n] 对应于从 s[m，s_len) 中能选出多少个 t[n，t_len)。
当 m == s_len，意味着s是空串，此时dp[s_len][n]，n 取 0 到 t_len - 1的值都为 0。
当 n == t_len，意味着t是空串，此时dp[m][t_len]，m 取 0 到 s_len的值都为 1。
然后状态转移的话和解法一分析的一样。如果求dp[s][t]。

- S[s] == T[t]，当前字符相等，那就对应两种情况，选择S的当前字母和不选择S的当前字母dp[s][t] = dp[s+1][t+1] + dp[s+1][t]
- S[s] != T[t]，只有一种情况，不选择S的当前字母dp[s][t] = dp[s+1][t]

代码就可以写了。

```java
class Solution {
    /**
        t.charAt(i) != s.charAt(j)
        -> dp[j][i] = dp[j + 1][i];

        t.charAt(i) == s.charAt(j)
        -> dp[j][i] = dp[j + 1][i + 1] + dp[j + 1][i];

           0 1 2 3 4 5 6
         t r a b b i t '' -> i
       s
    0  r   3 3 3 3 1 1 1
    1  a   0 3 3 3 1 1 1
    2  b   0 0 3 3 1 1 1
    3  b   0 0 1 2 1 1 1
    4  b   0 0 0 1 1 1 1
    5  i   0 0 0 0 1 1 1
    6  t   0 0 0 0 0 1 1
    7 ''   0 0 0 0 0 0 1
    -> j
     */
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        int[][] dp = new int[s_len + 1][t_len + 1];
        // 当 t 为空串时，所有的 s 对应于 1
        for(int i = 0; i <= s_len; i++) {
            dp[i][t_len] = 1;
        }
        // 倒着进行，s 每次增加一个字母
        for(int j = s_len - 1; j >= 0; j--) {
            // 倒着进行，t 每次增加一个字母
            for(int i = t_len - 1; i >= 0; i--) {
                // 如果当前字母相等
                if(t.charAt(i) == s.charAt(j)) {
                    // 对应于两种情况，选择当前字母和不选择当前字母
                    dp[j][i] = dp[j + 1][i + 1] + dp[j + 1][i];
                // 如果当前字母不相等
                } else {
                    dp[j][i] = dp[j + 1][i];
                }
            }
        }
        return dp[0][0];
    }
}
```

第四步：基于2D DP的空间优化1D DP(逆向版本)：

优化为2 rows
Style 1: Pre-initialize dp along with dpPrev initialize for loop before major for loop since dp[t_len] always same value in this problem (if dp[t_len] not always same value cannot pre-initialize dp)

```
class Solution {
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        // 原2D DP数组中的定义：s是row维度, t是column维度
        //int[][] dp = new int[s_len + 1][t_len + 1];
        // -> 现在只保留了column维度，因为本质上是row的维度上"上一行只依赖于下一行"，在原2D数组中上一行是dp[i]，下一行是dp[i + 1]，现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i + 1][j]平行替换为dpPrev[j]
        int[] dp = new int[t_len + 1];
        int[] dpPrev = new int[t_len + 1];
        // 当t为空串时，所有的s对应于 1，即2D DP数组中最后一列全部为1，dp[i][t_len] = 1
        //for(int i = 0; i <= s_len; i++) {
        //    dp[i][t_len] = 1;
        //}
        // -> 去掉row维度后初始化状态进化为只需要设定剩下column维度的第一个数即dp[t_len]为1，等价于t是空串时dp[i][t_len] = 1
        // -> Pre-initialize dp[t_len] along with dpPrev since dp[t_len] always same value as 1, otherwise must only
        // assign value to dp[t_len] in each for loop iteration
        dp[t_len] = 1;
        dpPrev[t_len] = 1;
        // 倒着进行，s 每次增加一个字母
        // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维持row维度的替换，外层循环必须使用row维度
        for(int j = s_len - 1; j >= 0; j--) {
            // 倒着进行，t 每次增加一个字母
            for(int i = t_len - 1; i >= 0; i--) {
                // 如果当前字母相等
                if(t.charAt(i) == s.charAt(j)) {
                    // 对应于两种情况，选择当前字母和不选择当前字母
                    //dp[j][i] = dp[j + 1][i + 1] + dp[j + 1][i];
                    // -> 现在由于去掉了row维度，dp[j][i]平行替换为dp[i]，dp[j + 1][i + 1]和dp[j + 1][i]平行替换为dpPrev[i + 1]和dpPrev[i]
                    dp[i] = dpPrev[i + 1] + dpPrev[i];
                // 如果当前字母不相等
                } else {
                    //dp[j][i] = dp[j + 1][i];
                    dp[i] = dpPrev[i];
                }
            }
            // -> 每次循环中dp是承接新计算结果的数组，为了腾出空间承接下一次循环的新计算结果，也为了让dpPrev更新为新的当前行计算结果，在每次循环结束的时候必须把计算结果从dp转存到dpPrev中
            dpPrev = dp.clone();
        }
        return dpPrev[0];
    }
}
```

2 rows array如何替代2D DP array的具体步骤

```
2D DP array

         t r a b b i t ''
       s
       r   3 3 3 3 1 1 1   -> equal j = 0 dp array
       a   0 3 3 3 1 1 1   -> equal j = 1 dp array
       b   0 0 3 3 1 1 1   -> equal j = 2 dp array
       b   0 0 1 2 1 1 1   -> equal j = 3 dp array
       b   0 0 0 1 1 1 1   -> equal j = 4 dp array
       i   0 0 0 0 1 1 1   -> equal j = 5 dp array
```

```
      t   0 0 0 0 0 1 1   -> equal j = 6 dp array
     ''   0 0 0 0 0 0 1   -> equal initial

外层循环为row维度，逐行填充，用2 rows array取代原先2D DP array
for(int j = s_len - 1; j >= 0; j--)
Initial:
    dp = [0, 0, 0, 0, 0, 0, 1] -> Pre-initialize dp[t_len] along with dpPrev since dp[t_len] always same value as 1, otherwise must only assign value to dp[t_len] in each for loop iteration
dpPrev = [0, 0, 0, 0, 0, 0, 1]
================================
j = 6 ->
before dpPrev = dp.clone()
    dp = [0, 0, 0, 0, 0, 1, 1]
dpPrev = [0, 0, 0, 0, 0, 0, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 0, 0, 0, 0, 1, 1]
dpPrev = [0, 0, 0, 0, 0, 1, 1]
================================
j = 5 ->
before dpPrev = dp.clone()
    dp = [0, 0, 0, 0, 1, 1, 1]
dpPrev = [0, 0, 0, 0, 0, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 0, 0, 0, 1, 1, 1]
dpPrev = [0, 0, 0, 0, 1, 1, 1]
================================
j = 4 ->
before dpPrev = dp.clone()
    dp = [0, 0, 0, 1, 1, 1, 1]
dpPrev = [0, 0, 0, 0, 1, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 0, 0, 1, 1, 1, 1]
dpPrev = [0, 0, 0, 1, 1, 1, 1]
================================
j = 3 ->
before dpPrev = dp.clone()
```

```
    dp = [0, 0, 1, 2, 1, 1, 1]
dpPrev = [0, 0, 0, 1, 1, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 0, 1, 2, 1, 1, 1]
dpPrev = [0, 0, 1, 2, 1, 1, 1]
================================
j = 2 ->
before dpPrev = dp.clone()
    dp = [0, 0, 3, 3, 1, 1, 1]
dpPrev = [0, 0, 1, 2, 1, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 0, 3, 3, 1, 1, 1]
dpPrev = [0, 0, 3, 3, 1, 1, 1]
================================
j = 1 ->
before dpPrev = dp.clone()
    dp = [0, 3, 3, 3, 1, 1, 1]
dpPrev = [0, 0, 3, 3, 1, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 3, 3, 3, 1, 1, 1]
dpPrev = [0, 3, 3, 3, 1, 1, 1]
================================
j = 0 ->
before dpPrev = dp.clone()
    dp = [3, 3, 3, 3, 1, 1, 1]
dpPrev = [0, 3, 3, 3, 1, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [3, 3, 3, 3, 1, 1, 1]
dpPrev = [3, 3, 3, 3, 1, 1, 1]
================================
Finally either return dp[0] or dpPrev[0] is same
```

Style 2: dp along with dpPrev initialize in major for loop (more general way, since we cannot guarantee dp[t_len] always same value, refer to L72. Edit Distance)

First is wrong way if we miss the dp[t_len] initialize in each for loop iteration

```java
class Solution {
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        // 原2D DP数组中的定义：s是row维度，t是column维度
        //int[][] dp = new int[s_len + 1][t_len + 1];
        // -> 现在只保留了column维度，因为本质上是row的维度上"上一行只依赖于下一行"，在原2D数组中上一行是dp[i]，下一行是dp[i + 1]，现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i + 1][j]平行替换为dpPrev[j]
        int[] dp = new int[t_len + 1];
        int[] dpPrev = new int[t_len + 1];
        // 当t为空串时，所有的s对应于 1，即2D DP数组中最后一列全部为1，dp[i][t_len] = 1
        //for(int i = 0; i <= s_len; i++) {
        //    dp[i][t_len] = 1;
        //}
        // -> 去掉row维度后初始化状态进化为只需要设定剩下column维度的第一个数即dp[t_len]为1，等价于t是空串时dp[i][t_len] = 1
        //dp[t_len] = 1; --> remove from here and suppose to relocate into major for loop then initialize in each for loop iteration

        dpPrev[t_len] = 1;
        // 倒着进行，s 每次增加一个字母
        // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维持row维度的替换，外层循环必须使用row维度
        for(int j = s_len - 1; j >= 0; j--) {
            //dp[t_len] = 1; --> comment out to show what will happen if we miss the initialize for dp[t_len] in each iteration
            // 倒着进行，t 每次增加一个字母
            for(int i = t_len - 1; i >= 0; i--) {
                // 如果当前字母相等
                if(t.charAt(i) == s.charAt(j)) {
                    // 对应于两种情况，选择当前字母和不选择当前字母
                    //dp[j][i] = dp[j + 1][i + 1] + dp[j + 1][i];
                    // -> 现在由于去掉了row维度，dp[j][i]平行替换为dp[i]，dp[j + 1][i + 1]和dp[j + 1][i]平行替换为dpPrev[i + 1]和dpPrev[i]
                    dp[i] = dpPrev[i + 1] + dpPrev[i];
                // 如果当前字母不相等
                } else {
                    //dp[j][i] = dp[j + 1][i];
                    dp[i] = dpPrev[i];
                }
            }
            // -> 每次循环中dp是承接新计算结果的数组，为了腾出空间承接下一次循环的新计算结果，
也为了让dpPrev更新为新的当前行计算结果，在每次循环结束的时候必须把计算结果从dp转存到dpPrev中
            dpPrev = dp.clone();
        }
        return dpPrev[0];
    }
}

=====================================================================================
================
Wrong result below:
e.g
s = "babgbag", t = "bag"

           0 1 2 3
         t b a g '' -> i
       s
    0  b   4 2 1 0 -> equal j = 0 dp array
    1  a   2 2 1 0 -> equal j = 1 dp array
    2  b   2 1 1 0 -> equal j = 2 dp array
    3  g   1 1 1 0 -> equal j = 3 dp array
    4  b   1 1 1 0 -> equal j = 4 dp array
    5  a   0 1 1 0 -> equal j = 5 dp array
    6  g   0 0 1 0 -> equal j = 6 dp array
    7 ''   0 0 0 1 -> equal initial
    -> j

外层循环为row维度，逐行填充，用2 rows array取代原先2D DP array
for(int j = s_len - 1; j >= 0; j--)
Initial:
dp -> no initialize for dp inside dpPrev initialize for loop
dpPrev = [0, 0, 0, 1]
================================
j = 6 ->
before dpPrev = dp.clone()
    dp = [0, 0, 1, 0] -> No initialize of last element dp[t_len] = 1 in each round
dpPrev = [0, 0, 0, 1]
```

```
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 0, 1, 0]
dpPrev = [0, 0, 1, 0]
================================
j = 5 ->
before dpPrev = dp.clone()
    dp = [0, 1, 1, 0] -> No initialize of last element dp[t_len] = 1 in each round
dpPrev = [0, 0, 1, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 1, 1, 0]
dpPrev = [0, 1, 1, 0]
================================
j = 4 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 0] -> No initialize of last element dp[t_len] = 1 in each round
dpPrev = [0, 1, 1, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 0]
dpPrev = [1, 1, 1, 0]
================================
j = 3 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 0] -> No initialize of last element dp[t_len] = 1 in each round
dpPrev = [1, 1, 1, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 0]
dpPrev = [1, 1, 1, 0]
================================
j = 2 ->
before dpPrev = dp.clone()
    dp = [2, 1, 1, 0] -> No initialize of last element dp[t_len] = 1 in each round
dpPrev = [1, 1, 1, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [2, 1, 1, 0]
dpPrev = [2, 1, 1, 0]
```

```
================================
j = 1 ->
before dpPrev = dp.clone()
    dp = [2, 2, 1, 0] -> No initialize of last element dp[t_len] = 1 in each round
dpPrev = [2, 1, 1, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [2, 2, 1, 0]
dpPrev = [2, 2, 1, 0]
================================
j = 0 ->
before dpPrev = dp.clone()
    dp = [4, 2, 1, 0] -> No initialize of last element dp[t_len] = 1 in each round
dpPrev = [2, 2, 1, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [4, 2, 1, 0]
dpPrev = [4, 2, 1, 0]
================================
Finally either return dp[0] or dpPrev[0] is same
```

Then we add back the dp[t_len] initialize in each for loop iteration as correct way

```
class Solution {
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        // 原2D DP数组中的定义：s是row维度，t是column维度
        //int[][] dp = new int[s_len + 1][t_len + 1];
        // -> 现在只保留了column维度，因为本质上是row的维度上"上一行只依赖于下一行"，在原2D数组中上一行是dp[i]，下一行是dp[i + 1]，现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i + 1][j]平行替换为dpPrev[j]
        int[] dp = new int[t_len + 1];
        int[] dpPrev = new int[t_len + 1];
        // 当t为空串时，所有的s对应于 1，即2D DP数组中最后一列全部为1，dp[i][t_len] = 1
        //for(int i = 0; i <= s_len; i++) {
        //    dp[i][t_len] = 1;
        //}
        // -> 去掉row维度后初始化状态进化为只需要设定剩下column维度的第一个数即dp[t_len]为1，等价于t是空串时dp[i][t_len] = 1
        //dp[t_len] = 1; --> remove from here and suppose to relocate into major for loop then initialize in each for loop iteration

        dpPrev[t_len] = 1;
        // 倒着进行，s 每次增加一个字母
        // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维持row维度的替换，外层循环必须使用row维度
        for(int j = s_len - 1; j >= 0; j--) {
            // Correct way to initialize dp[t_len] in each iteration
            dp[t_len] = 1;
            // 倒着进行，t 每次增加一个字母
            for(int i = t_len - 1; i >= 0; i--) {
                // 如果当前字母相等
                if(t.charAt(i) == s.charAt(j)) {
                    // 对应于两种情况，选择当前字母和不选择当前字母
                    //dp[j][i] = dp[j + 1][i + 1] + dp[j + 1][i];
                    // -> 现在由于去掉了row维度，dp[j][i]平行替换为dp[i]，dp[j + 1][i + 1]和dp[j + 1][i]平行替换为dpPrev[i + 1]和dpPrev[i]
                    dp[i] = dpPrev[i + 1] + dpPrev[i];
                // 如果当前字母不相等
                } else {
                    //dp[j][i] = dp[j + 1][i];
                    dp[i] = dpPrev[i];
                }
            }
            // -> 每次循环中dp是承接新计算结果的数组，为了腾出空间承接下一次循环的新计算结果，也为了让dpPrev更新为新的当前行计算结果，在每次循环结束的时候必须把计算结果从dp转存到dpPrev中
            dpPrev = dp.clone();
        }
        return dpPrev[0];
    }
}



=======================================================================================================
Correct result below:
e.g
```

```
s = "babgbag", t = "bag"

          0 1 2 3
        t b a g '' -> i
      s
    0  b   5 3 2 1 -> equal j = 0 dp array
    1  a   2 3 2 1 -> equal j = 1 dp array
    2  b   2 1 2 1 -> equal j = 2 dp array
    3  g   1 1 2 1 -> equal j = 3 dp array
    4  b   1 1 1 1 -> equal j = 4 dp array
    5  a   0 1 1 1 -> equal j = 5 dp array
    6  g   0 0 1 1 -> equal j = 6 dp array
    7 ''   0 0 0 1 -> equal initial
    -> j

外层循环为row维度，逐行填充，用2 rows array取代原先2D DP array
for(int j = s_len - 1; j >= 0; j--)
Initial:
dp -> no initialize for dp inside dpPrev initialize for loop
OR we can initialize dp along with dpPrev initialize for loop then no need
initialize dp[t_len] = 1 in each round later in for loop
dpPrev = [0, 0, 0, 1]
================================
j = 6 ->
before dpPrev = dp.clone()
    dp = [0, 0, 1, 1] -> Initialize of last element dp[t_len] = 1 in each round
dpPrev = [0, 0, 0, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 0, 1, 1]
dpPrev = [0, 0, 1, 1]
================================
j = 5 ->
before dpPrev = dp.clone()
    dp = [0, 1, 1, 1] -> Initialize of last element dp[t_len] = 1 in each round
dpPrev = [0, 0, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [0, 1, 1, 1]
```

```
dpPrev = [0, 1, 1, 1]
================================
j = 4 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 1] -> Initialize of last element dp[t_len] = 1 in each round
dpPrev = [0, 1, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 1]
dpPrev = [1, 1, 1, 1]
================================
j = 3 ->
before dpPrev = dp.clone()
    dp = [1, 1, 2, 1] -> Initialize of last element dp[t_len] = 1 in each round
dpPrev = [1, 1, 1, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 2, 1]
dpPrev = [1, 1, 2, 1]
================================
j = 2 ->
before dpPrev = dp.clone()
    dp = [2, 1, 2, 1] -> Initialize of last element dp[t_len] = 1 in each round
dpPrev = [1, 1, 2, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [2, 1, 2, 1]
dpPrev = [2, 1, 2, 1]
================================
j = 1 ->
before dpPrev = dp.clone()
    dp = [2, 3, 2, 1] -> Initialize of last element dp[t_len] = 1 in each round
dpPrev = [2, 1, 2, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [2, 3, 2, 1]
dpPrev = [2, 3, 2, 1]
================================
j = 0 ->
before dpPrev = dp.clone()
```

```
    dp = [5, 3, 2, 1] -> Initialize of last element dp[t_len] = 1 in each round
dpPrev = [2, 3, 2, 1]
--------------------------------
after dpPrev = dp.clone()
    dp = [5, 3, 2, 1]
dpPrev = [5, 3, 2, 1]
================================
Finally either return dp[0] or dpPrev[0] is same
```

进一步优化为1 row

```java
class Solution {
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        int[] dpPrev = new int[t_len + 1];
        dpPrev[t_len] = 1;
        for(int j = s_len - 1; j >= 0; j--) {
            // 我们必须改变内循环中基于column维度的扫描方向，与2 rows array
            // DP中从右向左(减小i)不同，改为从左向右(增大i)，因为dp状态
            // 表实际上是后续状态基于已有状态的生成过程，已有状态如果在生成后
            // 被重写改变，则基于该已有状态的后续状态无法稳定，该情形只会
            // 在将2 rows array DP合并为1 row array DP的时候出现
            // 例如如果按照常规从右往左扫描，在合并后1 row array中dpPrev[i + 1]
            // 会被更新，导致基于它的dpPrev[i]不稳定，因为上一层循环中的
            // dpPrev[i + 1]发生了变化(因为是逆推，所以i + 1才是上一层),
            // 被当前层的新数值覆盖，而基于原则，当前循环结果dpPrev[i]只能
            // 基于上一层循环的稳定结果，即期待本应代表上一层循环的结果的
            // dpPrev[i + 1]不做改变，这种不做改变在2 rows array是可以轻易
            // 实现的，即用两个单独的数列来分别存储上一层和当前层循环结果，
            // 但当合并为1 row array以后，上一层的dpPrev[i + 1]和当前层的
            // dpPrev[i]变成了必须存储在同一数组中，换句话说上一层的dpPrev[i + 1]
            // 值会被当前层的dpPrev[i]更新，导致上一层dpPrev[i + 1]的原值丢失，
            // 导致基于上一层dpPrev[i + 1]的原值计算获得的dpPrev[i]不稳定,
            // 而在正确的写法中，即改为从左向右循环以后，实际上利用了前后两次循环
            // 的时间差，即后面一次循环(写作dpPrev[i]的dp[i])依然能读取到前一次
            // 循环中dpPrev[i]的值，因为dpPrev[i]还没有更新，虽然表面上看标记仍
```

```
                // 为dpPrev，但本质上在后一次循环开启的时候dpPrev实际从左到右开始承担
                // dp的作用，更新为dp的值
                for(int i = 0; i <= t_len - 1; i++) {
                    if(t.charAt(i) == s.charAt(j)) {
                        dpPrev[i] = dpPrev[i + 1] + dpPrev[i];
                    }
                }
            }
            return dpPrev[0];
        }
}
```

(2) 正向递归进化到DP的思路

第一步：实现一个基本递归(正向版本)：

在LeetCode解答中，从递归的角度讲，"顶"就是递归最开始在主体方法中被呼叫的状态，本题中就是m == s_len 和 n == t_len 时，"底"在本题中就是当 递归到达m == 0 和 n == 0 时，也就是递归实际方法中的base condition，递归就是先从"顶"即m == s_len 和 n == t_len逐层到达"底"即m == 0 和 n == 0，然后在到达"底"后再通过返回语句逐层从"底"返回到"顶"，而DP能够省略掉递归中"从顶到底"的过程，而"直接由底向顶"，这也意味着从二维数组DP状态表的角度讲，从左上角正推到右下角的过程，也就是m == 0(底) --> m == s_len(顶)，n == 0(底) --> n == t_len(顶)的过程

递归中由顶到底的过程：

我们的递归始于m == s_len和n == t_len时，m == s_len(顶) --> m == 0(底)，n == t_len(顶) --> n == 0(底)，然后在到底的时候触碰到base condition开启return返回过程

递归中再由底回到顶的过程：

在从顶到底并触碰到base condition开启return之后，逐层返回，m == 0(底) --> m == s_len(顶)，n == 0(底) --> n == t_len(顶)，此时最终状态实际上在顶，也就是m == s_len和n == t_len时取得，和二维DP中最终状态在右下角[s_len, t_len]处获得形成一致

```
Style 1: Base condition 1 ahead of Base condition 2 but additional condition as t_end >
 0

class Solution {
    public int numDistinct(String s, String t) {
        // 从顶m == S_len和n == T_len开始递归
        return helper(s, s.length(), t, t.length());
    }
    private int helper(String s, int s_end, String t, int t_end) {
        // 在底m == 0和n == 0触底开启逐层返回到顶过程
        // Base condition 1:
        // For s[0, s_end], s_end == 0, which means s is empty string now,
        // to pick chars from empty string, no choice, hence return 0
        if(t_end > 0 && s_end == 0) {
            return 0;
        }
        // Base condition 2:
        // For t[0, t_end], t_end == 0, which means t is empty string now,
        // to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_end == 0) {
            return 1;
        }
        int count = 0;
        if(s.charAt(s_end - 1) == t.charAt(t_end - 1)) {
            count += helper(s, s_end - 1, t, t_end - 1);
            count += helper(s, s_end - 1, t, t_end);
        } else {
            count += helper(s, s_end - 1, t, t_end);
        }
        return count;
    }
}

=====================================================================================
===========
Style 2: Base condition 2 ahead of Base condition 1 then no additional condition
required

class Solution {
    public int numDistinct(String s, String t) {
        // 从顶m == s_len和n == t_len开始递归
        return helper(s, s.length(), t, t.length());
    }

    private int helper(String s, int s_end, String t, int t_end) {
        // 在底m == 0和n == 0触底开启逐层返回到顶过程
        // Base condition 2:
        // For t[0, t_end], t_end == 0, which means t is empty string now,
        // to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_end == 0) {
            return 1;
        }
        // Base condition 1:
        // For s[0, s_end], s_end == 0, which means s is empty string now,
        // to pick chars from empty string, no choice, hence return 0
        if(s_end == 0) {
            return 0;
        }
        int count = 0;
        if(s.charAt(s_end - 1) == t.charAt(t_end - 1)) {
            count += helper(s, s_end - 1, t, t_end - 1);
            count += helper(s, s_end - 1, t, t_end);
        } else {
            count += helper(s, s_end - 1, t, t_end);
        }
        return count;
    }
}
```

第二步：递归配合Memoization(正向版本)：

```
Style 1: Use classic memo

class Solution {
    public int numDistinct(String s, String t) {
        Integer[][] memo = new Integer[s.length() + 1][t.length() + 1];
        return helper(s, s.length(), t, t.length(), memo);
    }

    private int helper(String s, int s_end, String t, int t_end, Integer[][] memo) {
        if(memo[s_end][t_end] != null) {
            return memo[s_end][t_end];
        }
        // Base condition 2:
        // For t[0, t_end], t_end == 0, which means t is empty string now,
        // to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_end == 0) {
            return memo[s_end][t_end] = 1;
        }
        // Base condition 1:
        // For s[0, s_end], s_end == 0, which means s is empty string now,
        // to pick chars from empty string, no choice, hence return 0
        if(s_end == 0) {
            return memo[s_end][t_end] = 0;
        }
        int count = 0;
        if(s.charAt(s_end - 1) == t.charAt(t_end - 1)) {
            count += helper(s, s_end - 1, t, t_end - 1, memo);
            count += helper(s, s_end - 1, t, t_end, memo);
        } else {
            count += helper(s, s_end - 1, t, t_end, memo);
        }
        return memo[s_end][t_end] = count;
    }
}

=========================================================================================================
Style 2: Use String as key in HashMap to create memo

class Solution {
    public int numDistinct(String s, String t) {
        Map<String, Integer> memo = new HashMap<String, Integer>();
        // Given s and t, pick chars from s to match t, find out how many
        // distinct subsequences in s could match t
        return helper(s, s.length(), t, t.length(), memo);
    }
    // Why Base condition 1 cannot switch with Base condition 2 ?
    // Test out by:
    // Input: s = "rabbbit", t = "rabbit"
    // Expect Output: 3, Actual Output: 0
    private int helper(String s, int s_end, String t, int t_end, Map<String, Integer> memo) {
        // Base condition 2:
        // For t[0, t_end], t_end == 0, which means t is empty string now,
        // to pick empty string from s to match t, there is only one choice
        // as not pick up any char from s, hence return 1
        if(t_end == 0) {
            return 1;
        }
        // Base condition 1:
        // For s[0, s_end], s_end == 0, which means s is empty string now,
        // to pick chars from empty string, no choice, hence return 0
        if(s_end == 0) {
            return 0;
        }
        String key = s_end + "_" + t_end;
        if(memo.containsKey(key)) {
            return memo.get(key);
        }
        // Divide
        // Case 1: If s[s_end - 1] == t[t_end - 1], we have two options:
        // (1) Use current char as s[s_end - 1] in s, then move index in both s and t one step backward
        // e.g if s[s_end - 1] == t[t_end - 1], if use s[s_end - 1], in next recursion level, we are going to find how many distinct sequence can be found in s[0, s_len - 2] equal to t[0, t_len - 2], in another word, in next recursion level, use a "shorter" s to find a "shorter" t
        // (2) Not use current char as s[s_end - 1] in s, then only move index in s one step backward
        // e.g if s[s_end - 1] == t[t_end - 1], not use s[s_end - 1], in next recursion level, we are going to find how many distinct sequence can be found in s[0, s_end - 2] equal to t[0, t_end - 1], in another word, in next recursion level, use a "shorter" s to find original t
        // Case 2: If s[s_end - 1] != t[t_end - 1], we have one option:
        // Not use current char as s[s_start] in s, then only move index in s one step ahead
        // e.g if s[s_end - 1] != t[t_end - 1], not use s[s_end - 1], in next recursion level, we are going to find how many distinct sequence can be found in s[0, s_end - 2] equal to t[0, t_end - 1], in another word, in next recursion level, use a "shorter" s to find original t
        int count = 0;
        if(s.charAt(s_end - 1) == t.charAt(t_end - 1)) {
            count += helper(s, s_end - 1, t, t_end - 1, memo);
```

```
                count += helper(s, s_end - 1, t, t_end, memo);
            } else {
                count = helper(s, s_end - 1, t, t_end, memo);
            }
            memo.put(key, count);
            return count;
        }
}
```

第三步：基于递归的2D DP(正向版本)：

dp[i][j] means s[0..i - 1] contains t[0..j - 1] that many times as distinct subsequences

注意：正向版本的和之前的逆向版本定义不同, 之前的逆向版本定义：dp[m][n] 对应于从 s[m，s_len) 中能选出多少个 t[n，t_len),

换成i和j的说法是 dp[i][j] means s[i, s_len) contains t[j, t_len) that many times as distinct subsequences

```
Style 1: String s present by each column, String t present by each row

class Solution {
    /**
        s.charAt(i - 1) != t.charAt(j - 1)
        -> dp[i][j] = dp[i - 1][j];

        s.charAt(i - 1) == t.charAt(j - 1)
        -> dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1];

          t '' r a b b i t
        s
        ''    1 0 0 0 0 0 0
        r     1 1 0 0 0 0 0
        a     1 1 1 0 0 0 0
        b     1 1 1 1 0 0 0
        b     1 1 1 2 1 0 0
        b     1 1 1 3 3 0 0
        i     1 1 1 3 3 3 0
        t     1 1 1 3 3 3 3
     */
```

```java
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        // dp[i][j] means s[0..i - 1] contains t[0..j - 1] that many times as distinct subsequences
        // s -> i as 1st dimension on row, t -> j as 2nd dimension on column
        // +1 for both dimensions means consider empty string for both s and t
        int[][] dp = new int[s_len + 1][t_len + 1];
        // t as "" empty string will always be a subsequence of s
        // we lock the column as j = 0, and any dp[i][j] on row i = [0, s_len] is 1
        // dp[0][0] is special one as both s and t as "" empty string, still
        // recognize t as subsequence of s
        for(int i = 0; i <= s_len; i++) {
            dp[i][0] = 1;
        }
        // Reversely, if s as "" empty string will never contain t as subsequence
        // but since dp[0][j] on column j = [1, n] is naturally 0, no need initialize
        //for(int j = 1; j <= n; j++) {
        //    dp[0][j] = 0;
        //}
        // Each time we attempt to increase one more character on subequence of s for a fixed t
        // That's why we prefer outer loop for t & inner loop for s, but evetually if we exchange
        // outer loop for s & inner loop for t will be same effect as we only have one condition
        // if(s.charAt(i - 1) != t.charAt(j - 1)) to check, it has no dependency on order
        for(int i = 1; i <= s_len; i++) {
            for(int j = 1; j <= t_len; j++) {
                // In both cases, the subsequence in String t should be ending with character t.charAt(j - 1)
                // Case 1: If subsequence in s with newly added one more character not ending with same characater as t.charAt(j - 1), which means the newly added one more character in s not contribute on number of distinct subsequences, then skip that char in s at index i, which means only need to consider how many distinct sequences s[0, i] contains t[0, j] equal to s[0, i - 1] contains t[0, j] when s[i] != t[j], so dp[i][j] = dp[i-1][j]
                // Case 2: If subsequence in s with newly added one more character has same ending with character as t.charAt(j - 1), which means the newly added one more character contribute on number of distinct subsequences, so we sum up two parts: 1. Not use it, the number we had before, same as Case 1, dp[i - 1][j], 2. Use it, the distinct number of subsequences equal to we had with one character less longer t and one character less longer s, dp[i - 1][j - 1]
                if(s.charAt(i - 1) != t.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1];
                }
            }
        }
        return dp[s_len][t_len];
    }
}

===========================================================================================

Style 2: String s present by each row, String t present by each column

class Solution {
    /**
        s.charAt(i - 1) != t.charAt(j - 1)
        -> dp[j][i] = dp[j][i - 1];

        s.charAt(i - 1) == t.charAt(j - 1)
        -> dp[j][i] = dp[j][i - 1] + dp[j - 1][i - 1];

          s '' r a b b b i t
        t
        ''   1 1 1 1 1 1 1 1
        r    0 1 1 1 1 1 1 1
        a    0 0 1 1 1 1 1 1
        b    0 0 0 1 2 3 3 3
        b    0 0 0 0 1 3 3 3
        i    0 0 0 0 0 0 3 3
        t    0 0 0 0 0 0 0 3
     */
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        // dp[i][j] means s[0..i - 1] contains t[0..j - 1] that many times as distinct subsequences
        // t -> j as 1st dimension on row, s -> i as 2nd dimension on column
        // +1 for both dimensions means consider empty string for both s and t
```

```java
        int[][] dp = new int[t_len + 1][s_len + 1];
        // t as "" empty string will always be a subsequence of s
        // we lock the row as j = 0, and any dp[i][j] on column i = [0, s_len] is 1
        // dp[0][0] is special one as both s and t as "" empty string, still
        // recognize t as subsequence of s
        for(int i = 0; i <= s_len; i++) {
            dp[0][i] = 1;
        }
        // Reversely, if s as "" empty string will never contain t as subsequence
        // but since dp[j][0] on row j = [1, n] is naturally 0, no need initialize
        //for(int j = 1; j <= t_len; j++) {
        //    dp[j][0] = 0;
        //}
        // Each time we attempt to increase one more character on subequence of s for a fixed t
        // That's why we prefer outer loop for t & inner loop for s, but evetually if we exchange
        // outer loop for s & inner loop for t will be same effect as we only have one condition
        // if(s.charAt(i - 1) != t.charAt(j - 1)) to check, it has no dependency on order
        for(int j = 1; j <= t_len; j++) {
            for(int i = 1; i <= s_len; i++) {
                // In both cases, the subsequence in String t should be ending with character t.charAt(j - 1)
                // Case 1: If subsequence in s with newly added one more character not ending with same characater as t.charAt(j - 1), which means the newly added one more character not contribute on number of distinct subsequences, so we just copy previous count(subsequence in s without newly added one more character as dp[j][i - 1]) into current one dp[j][i]
                // Case 2: If subsequence in s with newly added one more character has same ending with character as t.charAt(j - 1), which means the newly added one more character contribute on number of distinct subsequences, so we sum up two parts: 1. Not use it, the number we had before, same as Case 1, dp[i - 1][j], 2. Use it, the distinct number of subsequences equal to we had with one character less longer t and one character less longer s, dp[i - 1][j - 1]
                if(s.charAt(i - 1) != t.charAt(j - 1)) {
                    dp[j][i] = dp[j][i - 1];
                } else {
                    dp[j][i] = dp[j][i - 1] + dp[j - 1][i - 1];
                }
            }
        }
        return dp[t_len][s_len];
```

```
    }
}
```

第四步：基于2D DP的空间优化1D DP(正向版本，基于第三步Style 1)：

优化为2 rows

```
class Solution {
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        // 原2D DP数组中的定义：s是row维度, t是column维度
        //int[][] dp = new int[s_len][t_len];
        // -> 现在只保留了column维度，因为本质上是row的维度上"下一行只依赖于上一行"，在原2D数组中下一行是dp[i]，上一行是dp[i - 1]，现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i - 1][j]平行替换为dpPrev[j]
        int[] dp = new int[t_len + 1];
        int[] dpPrev = new int[t_len + 1];
        // 当t为空串时，所有的s对应于1，即2D DP数组中第一列全部为1，dp[i][0] = 1
        //for(int i = 0; i <= s_len; i++) {
        //    dp[i][0] = 1;
        //}
        // -> 去掉row维度后初始化状态进化为只需要设定剩下column维度的第一个数即dp[0]为1，等价于t是空串时dp[i][0] = 1
        dp[0] = 1;
        dpPrev[0] = 1;
        // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维持row维度的替换，外层循环必须使用row维度
        for(int i = 1; i <= s_len; i++) {
            // -> 内层循环为保留的column维度
            for(int j = 1; j <= t_len; j++) {
                // 如果当前字母相等
                if(s.charAt(i - 1) == t.charAt(j - 1)) {
                    // 对应于两种情况，选择当前字母和不选择当前字母
                    //dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
                    // -> 现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i - 1][j - 1]和dp[i - 1][j]平行替换为dp[j - 1]和dpPrev[j]
                    dp[j] = dpPrev[j - 1] + dpPrev[j];
                // 如果当前字母不相等
                } else {
                    //dp[i][j] = dp[i - 1][j];
                    dp[j] = dpPrev[j];
                }
            }
            // -> 每次循环中dp是承接新计算结果的数组，为了腾出空间承接下一次循环的新计算结果，也为了让dpPrev更新为新的当前行计算结果，在每次循环结束的时候必须把计算结果从dp转存到dpPrev中
            dpPrev = dp.clone();
        }
        return dpPrev[t_len];
    }
}

2 rows array如何替代2D DP array的具体步骤
2D DP array
          t '' r a b b i t
        s
        ''   1 0 0 0 0 0 0  -> equal initial
        r    1 1 0 0 0 0 0  -> equal i = 1 dp array
        a    1 1 1 0 0 0 0  -> equal i = 2 dp array
        b    1 1 1 1 0 0 0  -> equal i = 3 dp array
        b    1 1 1 2 1 0 0  -> equal i = 4 dp array
        b    1 1 1 3 3 0 0  -> equal i = 5 dp array
        i    1 1 1 3 3 3 0  -> equal i = 6 dp array
        t    1 1 1 3 3 3 3  -> equal i = 7 dp array

外层循环为row维度，逐行填充，用2 rows array取代原先2D DP array
for(int i = 1; i <= s_len; i++)

Initial:
    dp = [1, 0, 0, 0, 0, 0, 0]
dpPrev = [1, 0, 0, 0, 0, 0, 0]
================================
i = 1 ->
before dpPrev = dp.clone()
    dp = [1, 1, 0, 0, 0, 0, 0]
dpPrev = [1, 0, 0, 0, 0, 0, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 0, 0, 0, 0, 0]
```

```
dpPrev = [1, 1, 0, 0, 0, 0, 0]
================================
i = 2 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 0, 0, 0, 0]
dpPrev = [1, 1, 0, 0, 0, 0, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 0, 0, 0, 0]
dpPrev = [1, 1, 1, 0, 0, 0, 0]
================================
i = 3 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 1, 0, 0, 0]
dpPrev = [1, 1, 1, 0, 0, 0, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 1, 0, 0, 0]
dpPrev = [1, 1, 1, 1, 0, 0, 0]
================================
i = 4 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 2, 1, 0, 0]
dpPrev = [1, 1, 1, 1, 0, 0, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 2, 1, 0, 0]
dpPrev = [1, 1, 1, 2, 1, 0, 0]
================================
i = 5 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 3, 3, 0, 0]
dpPrev = [1, 1, 1, 2, 1, 0, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 3, 3, 0, 0]
dpPrev = [1, 1, 1, 3, 3, 0, 0]
================================
i = 6 ->
```

```
before dpPrev = dp.clone()
    dp = [1, 1, 1, 3, 3, 3, 0]
dpPrev = [1, 1, 1, 3, 3, 0, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 3, 3, 3, 0]
dpPrev = [1, 1, 1, 3, 3, 3, 0]
================================
i = 7 ->
before dpPrev = dp.clone()
    dp = [1, 1, 1, 3, 3, 3, 3]
dpPrev = [1, 1, 1, 3, 3, 3, 0]
--------------------------------
after dpPrev = dp.clone()
    dp = [1, 1, 1, 3, 3, 3, 3]
dpPrev = [1, 1, 1, 3, 3, 3, 3]
================================
Finally either return dp[t_len] or dpPrev[t_len] is same
```

进一步优化为1 row

```java
class Solution {
    public int numDistinct(String s, String t) {
        int s_len = s.length();
        int t_len = t.length();
        int[] dpPrev = new int[t_len + 1];
        dpPrev[0] = 1;
        for(int i = 1; i <= s_len; i++) {
            // 我们必须改变内循环中基于column维度的扫描方向，与2 rows array
            // DP中从左向右(增长j)不同，改为从右向左(减小j)，因为dp状态
            // 表实际上是后续状态基于已有状态的生成过程，已有状态如果在生成后
            // 被重写改变，则基于该已有状态的后续状态无法稳定，该情形只会
            // 在将2 rows array DP合并为1 row array DP的时候出现
            // 例如如果按照常规从左往右扫描，在合并后1 row array中dpPrev[j - 1]
            // 会被更新，导致基于它的dpPrev[j]不稳定，因为上一层循环中的
            // dpPrev[j - 1]发生了变化，被当前层的新数值覆盖，而基于原则，
            // 当前循环结果dpPrev[j]只能基于上一层循环的稳定结果，即期待本应
            // 代表上一层循环的结果的dpPrev[j - 1]不做改变，这种不做改变在
            // 2 rows array是可以轻易实现的，即用两个单独的数列来分别存储上
            // 一层和当前层循环结果，但当合并为1 row array以后，上一层的
            // dpPrev[j - 1]和当前层的dpPrev[j - 1]变成了必须存储在同一数组
            // 中，换句话说上一层的dpPrev[j - 1]值会被当前层的dpPrev[j - 1]更新，
            // 导致上一层dpPrev[j - 1]的原值丢失，导致基于上一层dpPrev[j - 1]
            // 的原值计算获得的dpPrev[j]不稳定
            for(int j = t_len; j >= 1; j--) {
                if(s.charAt(i - 1) == t.charAt(j - 1)) {
                    dpPrev[j] = dpPrev[j - 1] + dpPrev[j];
                }
            }
        }
        return dpPrev[t_len];
    }
}
```

Refer to

In 1D DP array 1 row solution, why scan from the right instead of left ?

https://leetcode.com/problems/partition-equal-subset-sum/solutions/90592/0-1-knapsack-detailed-explanation/comments/140416

Because dp[i] = dp[i] || dp[i-num] uses smaller index value dp[i-num].

When the current iteration begins, the values in dp[] are the result of previous iteration.

Current iteration's result should only depend on the values of previous iteration.

If you iterate from i = 0, then dp[i-num] will be overwritten before you use it, which is wrong.

You can avoid this problem by iterating from i=sum

https://leetcode.com/problems/partition-equal-subset-sum/solutions/90592/0-1-knapsack-detailed-explanation/comments/241664

```
public boolean canPartition(int[] nums) {
    int sum = 0;
    for(int num : nums) {
        sum += num;
    }
    if((sum & 1) == 1) {
        return false;
```

```java
    }
    sum /= 2;
    int n = nums.length;
    boolean[] dp = new boolean[sum+1];
    Arrays.fill(dp, false);
    dp[0] = true;
    for(int num : nums) {
        for(int i = sum; i > 0; i--) {
            if(i >= num) {
                dp[i] = dp[i] || dp[i-num];
            }
        }
    }
    return dp[sum];
}
```

Yes, the magic is observation from the induction rule/recurrence relation!

For this problem, the induction rule:

1. If not picking nums[i - 1], then dp[i][j] = dp[i-1][j]
2. if picking nums[i - 1], then dp[i][j] = dp[i - 1][j - nums[i - 1]]

You can see that if you point them out in the matrix, it will be like:

```
                          j
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . ? . . ? . . . . . .   ?(left): dp[i - 1][j - nums[i], ?(right): dp[i - 1][j]
i           . . . . . # . . . . . .   # dp[i][j]
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . . . . . . . . . . .
```

1. Optimize to O(2*n): you can see that dp[i][j]

only depends on previous row, so you can optimize the space by only using **2 rows** instead of the matrix. Let's say array1 and array2. Every time you finish updating array2, array1 have no value, you can copy array2 to array1 as the previous row of the next new row.

2. Note: For 2 rows array solution since the previous iteration result will only keep in row array1 as dp[i - 1], current iteration result will only keep in row array2 as dp[i], and based on formula dp[i][j] = dp[i-1][j] || dp[i - 1][j - nums[i - 1]], the current iteration result row array2 as dp[i] will only depend on previous iteration result row array1 as dp[i - 1] , and since we use 2 rows, the previous and current iteration result naturally decoupled into 2 separate rows, when calculate current iteration result and store into array2 as dp[i] by using previous iteration result row array1 as dp[i - 1], no overwrite happen on row array1, its safe to keep iterating forwards on inner for loop in 2 rows array solution, and after finishing update array2, value in array1 is no use anymore, we can copy array2 into array1 and clean up array2 to prepare receiving new calculated result in next iteration
3. Optimize to O(n): you can also see that, the column indices of dp[i - 1][j - nums[i] and dp[i - 1][j] are <= j. The conclusion you can get is: the elements of previous row whose column index is > j(i.e. dp[i - 1][j + 1 : n - 1]) will not affect the update of dp[i][j] since we will not touch them:

```
                          j
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . ? . . ? x x x x x x  you will not touch x for dp[i][j]
i           . . . . . # . . . . . .  # dp[i][j]
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . . . . . . . . . . .
            . . . . . . . . . . . .
```

But thus if we merge array1 and array2 to a single array, if we update array backwards, all dependencies are not touched!

```
            (n represents new value, i.e. updated)
            . . ? . . ? n n n n n n n
                      #
```

However if we update forwards, dp[j - nums[i - 1]] is updated already, we cannot use it:

```
        (n represents new value, i.e. updated)
        n n n n n ? . . . . . .  where another ? goes? Oops, it is overriden, we lost it :(
                  #
```

Conclusion:
So the rule is that observe the positions of current element and its dependencies in the matrix. Mostly if current elements depends on the elements in previous row(most frequent case)/columns, you can optimize the space.

Refer to
L72.Edit Distance (Refer L115.Distinct Subsequences)