<https://leetcode.com/problems/valid-permutations-for-di-sequence/description/>

You are given a string s of length n where s[i] is either:

'D' means decreasing, or

'I' means increasing.

A permutation perm of n + 1 integers of all the integers in the range [0, n] is called a **valid** **permutation** if for all valid i:

**If s[i] == 'D', then perm[i] > perm[i + 1], and**

**If s[i] == 'I', then perm[i] < perm[i + 1].**

Return the number of **valid permutations** perm. Since the answer may be large, return it modulo 10^9 + 7.

**Example 1:**

Input: s = "DID"

Output: 5

Explanation: The 5 valid permutations of (0, 1, 2, 3) are:

(1, 0, 3, 2)

(2, 0, 3, 1)

(2, 1, 3, 0)

(3, 0, 2, 1)

(3, 1, 2, 0)

**Example 2:**

Input: s = "D"

Output: 1

**Constraints:**

n == s.length

1 <= n <= 200

s[i] is either 'I' or 'D'.

**Attempt 1: 2023-02-17**

**Solution 1: Native DFS (60 min, TLE 11/81)**

**Wrong Solution**

**Test out by: s = "DID", expected = 5, output = 31**

**Why it failed ?**

**Initially, we don't have for loop on 'helper()' method, it will failed directly on input s = "DID" and output 0, because if we only try 'lastDigit = 0', but when first char is 'D', in 'helper()' method it will go into 'if(curChar == 'D')' branch, which we won't have any choice on 'curDigit' when only have 'lastDigit = 0', then we recognize 'lastDigit' cannot be only 0, we have to loop through 0 to n. When we loop through 0 to n, which means 'curDigit' will have choices if 'lastDigit' is not 0 and 'curChar == D', but it also will cause duplicate attempts, because below code have no 'visited' recording and we will attempt same number on every digit, which as a permutation should not happen, e.g s = "DID" in wrong solution will mapping to a permutation {1,0,1,0}, 0 and 1 both duplicately used, but as definition, the permutation should only contain unique number from 0 to n, duplicate number not allowed.**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numPermsDISequence(String s) {

        int n = s.length();

        int result = 0;

        for(int lastDigit = 0; lastDigit <= n; lastDigit++) {

            result += helper(n, s, lastDigit, 0);

        }

        return result;

    }

    private int helper(int n, String s, int lastDigit, int sIndex) {

        if(sIndex == n) {

            return 1;

        }

        int count = 0;

        char curChar = s.charAt(sIndex);

        for(int curDigit = 0; curDigit <= n; curDigit++) {

            if(curChar == 'D') {

                if(curDigit < lastDigit) {

                    count = (count + helper(n, s, curDigit, sIndex + 1)) % MOD;

                }

            } else {

                if(curDigit > lastDigit) {

                    count = (count + helper(n, s, curDigit, sIndex + 1)) % MOD;

                }

            }

        }

        return count;

    }

}

**Solution 2: DFS + Backtracking (60 min, TLE 68/83)**

**But still TLE 68/83 since add Backtracking + 'visited' array time complexity still high, test out by: input s = "IDDDIIDIIIIIIIIDIDID"**

**Normally to find all permutations without duplicates we have to introduce two techniques:**

**(1) Backtracking -> find all permutations**

**(2) 'visited' array -> without duplicates**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numPermsDISequence(String s) {

        int n = s.length();

        int result = 0;

        boolean[] visited = new boolean[n + 1];

        for(int lastDigit = 0; lastDigit <= n; lastDigit++) {

            visited[lastDigit] = true;

            result = (result + helper(n, s, lastDigit, 0, visited)) % MOD;

            visited[lastDigit] = false;

        }

        return result;

    }

    private int helper(int n, String s, int lastDigit, int sIndex, boolean[] visited) {

        if(sIndex == n) {

            return 1;

        }

        long count = 0;

        char curChar = s.charAt(sIndex);

        for(int curDigit = 0; curDigit <= n; curDigit++) {

            if(visited[curDigit]) {

                continue;

            }

            visited[curDigit] = true;

            if(curChar == 'D') {

                if(curDigit < lastDigit) {

                    count = (count + helper(n, s, curDigit, sIndex + 1, visited)) % MOD;

                }

            } else {

                if(curDigit > lastDigit) {

                    count = (count + helper(n, s, curDigit, sIndex + 1, visited)) % MOD;

                }

            }

            visited[curDigit] = false;

        }

        return (int)count;

    }

}

Time Complexity: O(2^N)

The time complexity of the provided Java code is exponential, specifically O(2^n),

where n is the length of the input string s. This is because, in the worst case,

the code explores all possible permutations, and the number of recursive calls grows

exponentially with the size of the input.

The code performs a depth-first search (DFS) to explore all possible permutations.

For each position in the sequence, it tries all available options, resulting in a

branching factor of 2 (for 'I' and 'D'). As a result, the total number of recursive

calls becomes 2^n.

While the DFS approach is a valid and straightforward solution, it can become inefficient

for larger inputs due to its exponential nature. Consideration of dynamic programming or

memoization techniques could be beneficial to optimize the time complexity, especially

for cases where the same subproblems are computed multiple times.

**Solution 3: DFS + Backtracking + Memoization (10 min)**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numPermsDISequence(String s) {

        int n = s.length();

        int result = 0;

        boolean[] visited = new boolean[n + 1];

        Integer[][] memo = new Integer[n  + 1][n + 1];

        int lastDigit = 0;

        // A little promotion by shift 'lastDigit' once when first digit is 'D'

        // as when first digit is 'D', 'lastDigit' cannot start with 0 but 1

        if(s.charAt(0) == 'D') {

            lastDigit++;

        }

        for(; lastDigit <= n; lastDigit++) {

            visited[lastDigit] = true;

            result = (result + helper(n, s, lastDigit, 0, visited, memo)) % MOD;

            visited[lastDigit] = false;

        }

        return result;

    }

    private int helper(int n, String s, int lastDigit, int sIndex, boolean[] visited, Integer[][] memo) {

        if(sIndex == n) {

            return 1;

        }

        if(memo[lastDigit][sIndex] != null) {

            return memo[lastDigit][sIndex];

        }

        int count = 0;

        char curChar = s.charAt(sIndex);

        for(int curDigit = 0; curDigit <= n; curDigit++) {

            if(visited[curDigit]) {

                continue;

            }

            visited[curDigit] = true;

            if(curChar == 'D') {

                if(curDigit < lastDigit) {

                    count = (count + helper(n, s, curDigit, sIndex + 1, visited, memo)) % MOD;

                }

            } else {

                if(curDigit > lastDigit) {

                    count = (count + helper(n, s, curDigit, sIndex + 1, visited, memo)) % MOD;

                }

            }

            visited[curDigit] = false;

        }

        return memo[lastDigit][sIndex] = count;

    }

}

Time Complexity: O(N^3), N for 'lastDigit' for loop in numPermsDISequence() method,

N^2 for memoization recursive call, since worst case will touch all N^2 cells in memo

**Refer to**

<https://leetcode.com/problems/valid-permutations-for-di-sequence/solutions/1880241/c-dfs-memo/>

class Solution {

public:

int mod = 1e9+7;

if(in>=s.size()) return 1;

if(dp[in][last]!=-1) return dp[in][last];

if(s[in]=='D'){

long re = 0;

for(int i = 0; i<last; ++i){

if(vis[i]==1) continue;

vis[i] = 1;

int k = dfs(dp,vis,s,in+1,n,i);

vis[i]=0;

re = (re + k)%mod;

}

return dp[in][last] = (int)re;

}

else{

long re = 0;

for(int i = last+1; i<=s.size(); ++i){

if(vis[i]==1) continue;

vis[i] = 1;

int k = dfs(dp,vis,s,in+1,n,i);

vis[i]=0;

re = (re + k)%mod;

}

return dp[in][last] = (int)re;

}

}

int numPermsDISequence(string s) {

int n = s.size(), i = 0;

long re = 0;

vector<vector<int>> dp(n+2,vector<int>(n+2,-1));

vector<bool> vis(n+1,0);

if(s[0]=='D') i++;

for(;i<=n;i++){

vis[i] = 1;

re = (re + dfs(dp,vis,s,0,n,i))%mod;

vis[i] = 0;

}

return (int)re;

}

};

**Convert to Java by chatGPT**

import java.util.Arrays;

public class Solution {

int mod = 1000000007;

public int numPermsDISequence(String s) {

int n = s.length(), i = 0;

long result = 0;

int[][] dp = new int[n + 2][n + 2];

for (int[] row : dp) {

Arrays.fill(row, -1);

}

boolean[] vis = new boolean[n + 1];

if (s.charAt(0) == 'D') i++;

for (; i <= n; i++) {

vis[i] = true;

result = (result + dfs(dp, vis, s, 0, n, i)) % mod;

vis[i] = false;

}

return (int) result;

}

private int dfs(int[][] dp, boolean[] vis, String s, int in, int n, int last) {

if (in >= s.length()) return 1;

if (dp[in][last] != -1) return dp[in][last];

if (s.charAt(in) == 'D') {

long re = 0;

for (int i = 0; i < last; ++i) {

if (vis[i]) continue;

vis[i] = true;

int k = dfs(dp, vis, s, in + 1, n, i);

vis[i] = false;

re = (re + k) % mod;

}

return dp[in][last] = (int) re;

} else {

long re = 0;

for (int i = last + 1; i <= s.length(); ++i) {

if (vis[i]) continue;

vis[i] = true;

int k = dfs(dp, vis, s, in + 1, n, i);

vis[i] = false;

re = (re + k) % mod;

}

return dp[in][last] = (int) re;

}

}

public static void main(String[] args) {

Solution solution = new Solution();

System.out.println(solution.numPermsDISequence("DID")); // Output: 5

System.out.println(solution.numPermsDISequence("D")); // Output: 1

}

}

**Solution 3: DFS compatible for Memoization and DP (1080 min)**

**Base DFS (TLE 68/83 for input s = "IDDDIIDIIIIIIIIDIDID")**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numPermsDISequence(String s) {

        int n = s.length();

        int result = 0;

        for(int i = 0; i <= n; i++) {

            result = (result + helper(i, n - i, s, 0)) % MOD;

        }

        return result;

    }

    private int helper(int larger, int smaller, String s, int index) {

        if(index == s.length()) {

            return 1;

        }

        int count = 0;

        int c = s.charAt(index);

        if(c == 'D') {

            if(smaller > 0) {

                for(int i = 0; i < smaller; i++) {

                    count = (count + helper(larger + i, smaller - (i + 1), s, index + 1)) % MOD;

                }

            }

        } else {

            if(larger > 0) {

                for(int i = 0; i < larger; i++) {

                    count = (count + helper(larger - (i + 1), smaller + i, s, index + 1)) % MOD;

                }

            }

        }

        return count;

    }

}

**Base DFS -> DFS + Memoization**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numPermsDISequence(String s) {

        int n = s.length();

        Integer[][] memo = new Integer[n + 1][n + 1];

        int result = 0;

        for(int i = 0; i <= n; i++) {

            result = (result + helper(i, n - i, s, 0, memo)) % MOD;

        }

        return result;

    }

    private int helper(int larger, int smaller, String s, int index, Integer[][] memo) {

        if(index == s.length()) {

            return 1;

        }

        if(memo[larger][smaller] != null) {

            return memo[larger][smaller];

        }

        int count = 0;

        int c = s.charAt(index);

        if(c == 'D') {

            if(smaller > 0) {

                for(int i = 0; i < smaller; i++) {

                    count = (count + helper(larger + i, smaller - (i + 1), s, index + 1, memo)) % MOD;

                }

            }

        } else {

            if(larger > 0) {

                for(int i = 0; i < larger; i++) {

                    count = (count + helper(larger - (i + 1), smaller + i, s, index + 1, memo)) % MOD;

                }

            }

        }

        return memo[larger][smaller] = count;

    }

}

**Base DFS -> 2D DP**

**Wrong Solution**

**We cannot represent 'smaller' relation between recursive calls, only reflect 'index' and 'larger' relation between, even 'smaller' is derived by 'larger' as 'n - larger', but in recursive calls, this derivation is unstable, the 'n' will change to 'n - 1', 'n - 2'... etc. in recursion, and the evolution on 'smaller' is independant than 'larger', in other words, 'smaller' has its own flow, we cannot simply merge two variables into one, it may even require 3D DP to represent all states.**

**Test out by "DID", expect output 5, wrong output 21**

class Solution {

    int MOD = (int)1e9 + 7;

    public int numPermsDISequence(String s) {

        int n = s.length();

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

        // dp[i][j] -> i for index (bottom as n, top as 0),

        // j for 'larger'(represents 'smaller' as well) choice

        // between [0, n]

        for(int i = 0; i <= n; i++) {

            dp[n][i] = 1;

        }

        for(int index = n - 1; index >= 0; index--) {

            for(int larger = 0; larger <= n; larger++) {

// We cannot represent 'smaller' relation between recursive calls, only reflect 'index'

// and 'larger' relation between, even 'smaller' is derived by 'larger' as 'n - larger',

// but in recursive calls, this derivation is unstable, the 'n' will change to 'n - 1',

// 'n - 2'... etc. in recursion, and the evolution on 'smaller' is independant than

// 'larger', in other words, 'smaller' has its own flow, we cannot simply merge two

// variables into one, it may even require 3D DP to represent all states.

                int smaller = n - larger;

                int c = s.charAt(index);

                if(c == 'D') {

                    if(smaller > 0) {

                        for(int i = 0; i < smaller; i++) {

                            dp[index][larger] = (dp[index][larger] + dp[index + 1][larger + i]) % MOD;

                        }

                    }

                } else {

                    if(larger > 0) {

                        for(int i = 0; i < larger; i++) {

                            dp[index][larger] = (dp[index][larger] + dp[index + 1][larger - (i + 1)]) % MOD;

                        }

                    }

                }

            }

        }

        int result = 0;

        for (int i = 0; i <= n; i++) {

            result = (result + dp[0][i]) % MOD;

        }

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/valid-permutations-for-di-sequence/solutions/652054/java-dfs-with-memo/>

class Solution {

private static int MOD = 1000000007;

public int numPermsDISequence(String S) {

int res = 0;

Integer[][] dp = new Integer[S.length() + 1][S.length() + 1];

for (int i = 0; i <= S.length(); i++) {

res += dfs(i, S.length() - i, dp, S, 0);

res = res % MOD;

}

return res;

}

private int dfs(int higher, int lower, Integer[][] dp, String S, int index) {

if (index == S.length()) {

return 1;

}

int d = S.charAt(index) == 'D' ? 1 : 0;

if (dp[higher][lower] != null) {

return dp[higher][lower];

}

int count = 0;

if (d == 1) {

if (lower > 0) {

for (int i = 0; i < lower; i++) {

count += dfs(higher + i, lower - (i + 1), dp, S, index + 1);

count = count % MOD;

}

}

} else {

if (higher > 0) {

for (int i = 0; i < higher; i++) {

count += dfs(higher - (i + 1), lower + i, dp, S, index + 1);

count = count % MOD;

}

}

}

dp[higher][lower] = count;

return count;

}

}

**Step by step**

Input: s = "DID"

Output: 5

Explanation: The 5 valid permutations of (0, 1, 2, 3) are:

(1, 0, 3, 2)

(2, 0, 3, 1)

(2, 1, 3, 0)

(3, 0, 2, 1)

(3, 1, 2, 0)

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

The pros of using 'higher' and 'lower' (which indicates how many larger / smaller numbers than

previous number) is we do NOT have to care about what exactly the number is in each recursion

call, no need to record what numbers been used before, it flattens the relationship from all

historical recursive calls to two consecutive calls only, only need to know how many numbers

we have to proceed in next recursive call is fine

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

For loop in numPermsDISequence() method

----------------------------------------------------------------------------------------------

index = 0 -> start with index = 0 char as 'D' in s = "DID"

for i = 0 to i <= 3(s.length())

    i = 0 -> 1st combination higher = 0, lower = 3 - 0 = 3

 -> higher = 0 means count of larger number than previous number is 0

 -> lower = 3 means count of smaller number than previous number is 3

 -> the actual number satisfy 1st combination in {0,1,2,3} is 3 which

    has no larger number than 3 and three numbers as {0,1,2} smaller

    than 3, but we still choose to use the 'higher' and 'lower'

    combination to represent instead of the actual number 3, which

    equals assign 3 at index = 0

    Level 1 recurisve call index = 0

 -> dfs(0, 3, dp, s, 0)

    count = 0

    s.charAt(index) = s.charAt(0) = 'D'

    -> because its 'D' means 'decrease', we will look for all smaller

    numbers than previous number, it requires check on 'lower' part

    and we have 'lower = 3 > 0'

    -> loop all numbers(={0,1,2}) smaller than previous number(=3)

    -----------------------------------------------------------------------------------------

    index = 1 -> start with index = 1 char as 'I' in s = "DID"

    for i = 0 to i < 3(lower)

        i = 0 -> dfs(0 + 0, 3 - (0 + 1), dp, s, 0 + 1) = dfs(0, 2, dp, s, 1)

        after taking 3 out of {0,1,2,3} since already used for index = 0, then among remain

        three numbers as {0,1,2} we have 2 satisfy the 'higher = 0' and 'lower = 2' declearation,

        as we have no larger number than 2, and two numbers as {0,1} smaller than 2, which

        equals assign 2 at index = 1.

        Also when call next recursion as dfs(higher + i, lower - (i + 1), ...) based on 'higher'

        range and 'lower' range change when choose 'i' in original 'lower' range, 'higher'

        original range extends to 'higher + i', 'lower' original range shrinks to 'lower - (i + 1)',

        the additional '+ 1' means remove current used 'i' in original 'lower' range, as we

        concern on the remain numbers count in 'higher' and 'lower' range

        Level 2 recurisve call index = 1

     -> dfs(0, 2, dp, s, 1)

        count = 0

        s.charAt(index) = s.charAt(1) = 'I'

        -> because its 'I' means 'increase', we will look for all larger

        numbers than previous number, it requires check on 'higher' part

        but we have 'higher = 0 == 0'

        -> return 0

        -----------------------------------------------------------------------------------------

        i = 1 -> dfs(0 + 1, 3 - (1 + 1), dp, s, 0 + 1) = dfs(1, 1, dp, s, 1)

        after taking 3 out of {0,1,2,3} since already used for index = 0, then among remain

        three numbers as {0,1,2} we have 1 satisfy the 'higher = 1' and 'lower = 1' declearation,

        as we have one number as {2} larger than 1, and one number as {0} smaller than 1, which

        equals assign 1 at index = 1

        Level 2 recurisve call index = 1

     -> dfs(1, 1, dp, s, 1)

        count = 0

        s.charAt(index) = s.charAt(1) = 'I'

        -> because its 'I' means 'increase', we will look for all larger

        numbers than previous number, it requires check on 'higher' part

        and we have 'higher = 1 > 0'

        -> loop all numbers(={2}) larger than previous number(=1)

        -----------------------------------------------------------------------------------------

        index = 2 -> start with index = 2 char as 'D' in s = "DID"

        for i = 0 to i < 1(higher)

            i = 0 -> dfs(1 - (0 + 1), 1 + 0, dp, s, 1 + 1) = dfs(0, 1, dp, s, 2)

            after taking 3 out of {0,1,2,3} since already used for index = 0 and then taking 1 out of

            {0,1,2} since already used for index = 1, then among remain two numbers as {0,2} we have

            2 satisfy the 'higher = 0' and 'lower = 1' declearation, as we have no larger number than

            2, and one number as {0} smaller than 2, which equals assign 2 at index = 2

            Also when call next recursion as dfs(higher - (i + 1), lower + i, ...) based on 'higher'

            range and 'lower' range change when choose 'i' in original 'lower' range, 'higher'

            original range shrinks to 'higher - (i + 1)', 'lower' original range extends to 'lower + i',

            the additional '+ 1' means remove current used 'i' in original 'higher' range, as we

            concern on the remain numbers count in 'higher' and 'lower' range

            Level 3 recurisve call index = 2

         -> dfs(0, 1, dp, s, 2)

            count = 0

            s.charAt(index) = s.charAt(2) = 'D'

            -> because its 'D' means 'decrease', we will look for all smaller

            numbers than previous number, it requires check on 'lower' part

            and we have 'lower = 1 > 0'

            -> loop all numbers(={0}) smaller than previous number(=2)

            -----------------------------------------------------------------------------------------

            index = 3 -> start with index = 3 reach the end of recursion as 'index == S.length()'

            for i = 0 to i < 1(lower)

                i = 0 -> dfs(0 + 0, 1 - (0 + 1), dp, s, 2 + 1) = dfs(0, 0, dp, s, 3)

                after taking 3 out of {0,1,2,3} since already used for index = 0, then taking 1 out of

                {0,1,2} since already used for index = 1 and then taking 2 out of {0,2} since already

                used for index = 2, then the only remain one number as {0} we find it satisfies the

                'higher = 0' and 'lower = 0' declearation, as we have no remain larger number than 0,

                and no remain smaller number than 0, which equals assign 0 at index = 3

                Level 4 recurisve call index = 3

             -> dfs(0, 0, dp, s, 3)

                As 'index == S.length()' match, it means we find one permutation match "DID" condition

                as '3 -> 1 -> 2 -> 0', return 1

    -----------------------------------------------------------------------------------------

    i = 2 -> dfs(0 + 2, 3 - (2 + 1), dp, s, 0 + 1) = dfs(2, 0, dp, s, 1)

    after taking out 3 of {0,1,2,3} since already used for index = 0, then among remain

    three numbers as {0,1,2} we have 0 satisfy the 'higher = 2' and 'lower = 0' declearation,

    as we have two numbers as {1,2} larger than 0, and no smaller number than 0, which

    equals assign 0 at index = 1

    ... etc.

**Solution 4: 2D DP + 1D DP (??? min)**

**The above Solution 3 should be able to roll out to pure 2D DP and 1D DP solution, but require revise the definition on recursive method as currently using 'larger' and 'smaller' to represent candidates number range in next recursion level cannot directly convert into 2D DP. Below solution is a good way to "redefine" the 'larger' and 'smaller'**

**Refer to**

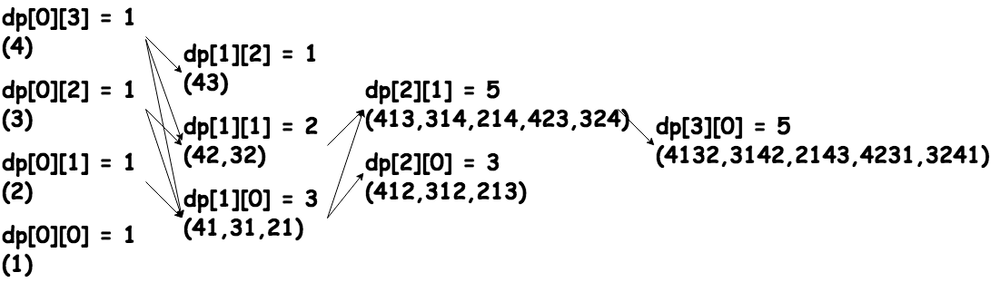
<https://leetcode.com/problems/valid-permutations-for-di-sequence/solutions/168278/c-java-python-dp-solution-o-n-2/>

**Intuition**

**dp[i][j] means the number of possible permutations of first i + 1 digits,**

**where the i + 1th digit is j + 1th smallest in the rest of unused digits.**

Ok, may not make sense ... Let's see the following diagram.



I take the example of S = "DID".

In the parenthesis, I list all possible permutations.

The permutation can start from 1, 2, 3, 4.

So dp[0][0] = dp[0][1] = dp[0][2] = dp[0][3] = 1.

We decrese from the first digit to the second,

the down arrow show the all possibile decresing pathes.

The same, because we increase from the second digit to the third,

the up arrow show the all possibile increasing pathes.

dp[2][1] = 5, mean the number of permutations

where the third digitis the second smallest of the rest.

We have 413,314,214,423,324.

Fow example 413, where 2,3 are left and 3 the second smallest of them.

**Explanation**

As shown in the diagram,

for "I", we calculate prefix sum of the array,

for "D", we calculate sufixsum of the array.

**Complexity**

Time O(N^2)

Space O(N^2)

**Java:**

public int numPermsDISequence(String S) {

int n = S.length(), mod = (int)1e9 + 7;

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

for (int j = 0; j <= n; j++) dp[0][j] = 1;

for (int i = 0; i < n; i++)

if (S.charAt(i) == 'I')

for (int j = 0, cur = 0; j < n - i; j++)

dp[i + 1][j] = cur = (cur + dp[i][j]) % mod;

else

for (int j = n - i - 1, cur = 0; j >= 0; j--)

dp[i + 1][j] = cur = (cur + dp[i][j + 1]) % mod;

return dp[n][0];

}

Now as we did for every DP, make it 1D dp.

Time O(N^2)

Space O(N)

public int numPermsDISequence(String S) {

int n = S.length(), mod = (int)1e9 + 7;

int[] dp = new int[n + 1], dp2 = new int[n];;

for (int j = 0; j <= n; j++) dp[j] = 1;

for (int i = 0; i < n; i++) {

if (S.charAt(i) == 'I')

for (int j = 0, cur = 0; j < n - i; j++)

dp2[j] = cur = (cur + dp[j]) % mod;

else

for (int j = n - i - 1, cur = 0; j >= 0; j--)

dp2[j] = cur = (cur + dp[j + 1]) % mod;

dp = Arrays.copyOf(dp2, n);

}

return dp[0];

}

**这道题本质上消耗了很大精力去寻找纯粹的DFS解法，因为纯粹的DFS根据理论是可以进化到Memoization，进化到2D DP, 1D DP的，在我们寻找到'larger'和'smaller'为代表的纯粹DFS解法(Solution 3)后直接进化出了Memoization就是证明，不过这个方法因为'larger'和'smaller'再加上'index'是需要3D DP来明确表示DP最在意的优化(Optimal)过程的，所以直接通过'larger'和'smaller'的纯粹DFS方法进化到2D DP失败了，我们需要寻找新的定义方式来优化'larger'和'smaller'这两个变量成一个变量才有可能(为何这两个变量不是稳定关系上面有详细叙述)。**

**另外，这道题最大的一个用意在于证明DP和Backtracking形态下的DFS是无法互通的，想变成DP的解法必须使用纯粹的DFS来进化**

**对于L903为何适用于Backtracking思想，是因为我们需要求出所有的满足要求的permutation的总个数，而不是唯一的一个最优解，Unlike DP, backtracking is typically not looking for one optimal solution, but is instead looking for all that satisfy some criteria. 那么是否可以在找寻单一一个满足要求的permutation的过程中使用呢？依然以L903为例，在寻找单一一个满足要求的permutation的过程中为了有效节约时间，对于已经遇到过的子状态(overlapped sub stats)不用再搜索一遍，直接用memoization处理**

**Refer to**

[DFS Backtracking and Dynamic Programming](note://WEBb2267f049a7493bf71622942b3d5a64d)

**Refer to L629.K Inverse Pairs Array**

[L629.K Inverse Pairs Array (Ref.L46)](note://WEB9bdebdf71920f84cf59d3d4434baa953)