<https://www.lintcode.com/problem/570/>

**Description**

Given a **permutation** of 1 - n integers in random order, find an integer that is missing.

n < 100

Data guarantees have only one solution.

if the list that you've found has more than one missing numbers, which could be that you didn't find the correct way to split the string.

**Example**

**Example1**

Input: n = 20 and s = 19201234567891011121314151618

Output: 17

Explanation:

19'20'1'2'3'4'5'6'7'8'9'10'11'12'13'14'15'16'18

**Example2**

Input: n = 6 and s = 56412

Output: 3

Explanation:

5'6'4'1'2

**Attempt 1: 2022-12-25**

**Solution 1: Backtracking (360 min)**

**Style 1: Similar to 0-1 knapsack, the modifying DFS end base condition fix the issue as we have to guarantee the missing element count is only 1 (by checking if visited status = false count is only 1, e.g when input = 111097654281222131272625242321320191817161514, the visited status = false count can be 2 means two missing elements is wrong solution)**

public class Solution {

/\*\*

\* @param n: An integer

\* @param s: a string with number from 1-n in random order and miss one number

\* @return: An integer

\*/

int result = -1;

public int findMissing2(int n, String s) {

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

//visited[0] = true;

helper(n, s, visited, 0);

return result;

}

// A different style of 0-1 knapsack:

// Choose 1 digit or 2 digits for current level recursion

private void helper(int n, String s, boolean[] visited, int startIndex) {

if(startIndex >= s.length()) {

// Test case:

// n = 28, "111097654281222131272625242321320191817161514"

//for(int i = 1; i <= n; i++) {

// if(!visited[i]) {

// result = i;

// return;

// }

//}

int count = 0;

int firstI = 0;

for(int i = 1; i <= n; i++) {

if(!visited[i]) {

count++;

firstI = i;

}

}

if(count == 1) {

result = firstI;

}

return;

}

if(s.charAt(startIndex) == '0') {

return;

}

// Attempt 1 digit for current level recursion

int val1 = Integer.parseInt(s.substring(startIndex, startIndex + 1));

if(!visited[val1]) {

visited[val1] = true;

helper(n, s, visited, startIndex + 1);

visited[val1] = false;

}

// Attempt 2 digits for current level recursion

if(startIndex < s.length() - 1) {

int val2 = Integer.parseInt(s.substring(startIndex, startIndex + 2));

if(val2 <= n && !visited[val2]) {

visited[val2] = true;

helper(n, s, visited, startIndex + 2);

visited[val2] = false;

}

}

}

}

**Style 2: Adding "count == n - 1" and add index out of boundary check condition "if(startIndex >= s.length()) " back**

public class Solution {

/\*\*

\* @param n: An integer

\* @param s: a string with number from 1-n in random order and miss one number

\* @return: An integer

\*/

int result = -1;

public int findMissing2(int n, String s) {

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

//visited[0] = true;

helper(n, s, visited, 0, 0);

return result;

}

// A different style of 0-1 knapsack:

// Choose 1 digit or 2 digits for current level recursion

private void helper(int n, String s, boolean[] visited, int startIndex, int count) {

// Test case:

// n = 28, "111097654281222131272625242321320191817161514"

// The additional condition "count == n - 1" is necessary

// We have two ways to cut the original input string

// "111097654281222131272625242321320191817161514" into 2 ways as output,

// digit 8 happens 2 times in original input, digit 2 happens 11 times in original

// input, the count of these 2 digits supports both cut ways as below:

// 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 0 [output 1]

// 0 1 0 3 4 5 6 7 0 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 [output 2] -> two missing elements is wrong

// So is the statement as "guarantee only one cut off way" against the fact ? Not exactly

// Because in output 2 when we cut out "28" at the same time we also MISS TWO NUMBERs

// as "2" and "8", two missing elements violate the definition in problem:

// permutation of 1 - n integers, Data guarantees have only one solution.

// if the list that you've found has more than one missing numbers

if(startIndex >= s.length() && count == n - 1) {

for(int i = 1; i <= n; i++) {

if(!visited[i]) {

result = i;

return;

}

}

return;

}

// Additional check added to avoid index out of boundray exception caused

// by "helper(n, s, visited, startIndex + 2, count + 1)" jump into

// "s.charAt(startIndex) == '0'", the 'startIndex' has chance out of boundray

// after '+2', but since we add 'count' condition in initial result output

// check as "if(startIndex >= s.length() && count == n - 1)", then we miss

// the function to individually block 'startIndex' out of boundary issue,

// so we have to make up by adding it back

if(startIndex >= s.length()) {

return;

}

if(s.charAt(startIndex) == '0') {

return;

}

// Attempt 1 digit for current level recursion

int val1 = Integer.parseInt(s.substring(startIndex, startIndex + 1));

if(!visited[val1]) {

visited[val1] = true;

helper(n, s, visited, startIndex + 1, count + 1);

visited[val1] = false;

}

// Attempt 2 digits for current level recursion

if(startIndex < s.length() - 1) {

int val2 = Integer.parseInt(s.substring(startIndex, startIndex + 2));

if(val2 <= n && !visited[val2]) {

visited[val2] = true;

helper(n, s, visited, startIndex + 2, count + 1);

visited[val2] = false;

}

}

}

}

**Note: Additional check**

We have two ways to cut the original input string "111097654281222131272625242321320191817161514"

into 2 ways as output, digit 8 happens 2 times in original input, digit 2 happens 11 times in original input, the count of these 2 digits supports both cut ways as below:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 0 **[output 1]**

0 1 0 3 4 5 6 7 0 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 **[output 2] -> two missing elements is wrong**

So is the statement as "guarantee only one cut off way" against the fact ? **Not exactly**

**Because in output 2 when we cut out "28" at the same time we also MISS TWO NUMBERs as "2" and "8", two missing elements violate the definition in problem:** permutation of 1 - n integers, Data guarantees have only one solution. if the list that you've found has **more than one missing numbers**

Refer to

<https://xizha677.gitbooks.io/codenotes/content/find-the-missing-number-ii.html>

This problem can be solved by depth first search.

* initialize a foundNums boolean array with false value for number from 1 to n.
* If the first digit is less than n, set it as found in foundNums and check the remaining part.
* If the first two digits are less than n, set this number as found in foundNums and check the remaining part.
* Stop when no more digit left, and return the only missing number in foundNums.

public class Solution {

private int result = 0;

public int findMissing2(int n, String str) {

dfs(0, n, str, new boolean[n + 1]);

return result;

}

private void dfs(int idx, int n, String str, boolean[] foundNums) {

if (idx >= str.length()) {

int count = 0;

int firstI = 0;

for (int i = 1; i <= n; i++) {

if (!foundNums[i]) {

count++;

firstI = i;

}

}

if (count == 1) {

result = firstI;

}

return;

}

//one digits

int num = (int)(str.charAt(idx) - '0');

if (num <= n && !foundNums[num]) {

foundNums[num] = true;

dfs(idx + 1, n, str, foundNums);

foundNums[num] = false;

}

//two digits

if (idx + 1 >= str.length()) {

return;

}

num = num \* 10 + (int)(str.charAt(idx + 1) - '0');

if (num <= n && !foundNums[num]) {

foundNums[num] = true;

dfs(idx + 2, n, str, foundNums);

foundNums[num] = false;

}

}

}

**Style 3: For loop**

public class Solution {

/\*\*

\* @param n: An integer

\* @param s: a string with number from 1-n in random order and miss one number

\* @return: An integer

\*/

int result = -1;

public int findMissing2(int n, String s) {

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

//visited[0] = true;

helper(n, s, visited, 0, 0);

return result;

}

// A different style of 0-1 knapsack:

// Choose 1 digit or 2 digits for current level recursion

private void helper(int n, String s, boolean[] visited, int startIndex, int count) {

if(startIndex == s.length() && count == n - 1) {

for(int i = 1; i <= n; i++) {

if(!visited[i]) {

result = i;

return;

}

}

return;

}

for(int i = startIndex + 1; i <= startIndex + 2 && i <= s.length(); i++) {

String substr = s.substring(startIndex, i);

// Make sure a number not start with 0

if(substr.charAt(0) == '0') {

continue;

}

int num = Integer.parseInt(substr);

// OR we can make sure a number not start with 0 by below way

//if(num > 0 && num <= n && !visited[num] && Integer.toString(num).length() == substr.length()) {

// number should be valid as > 0 and <= n

// number should not be visited before

if(num > 0 && num <= n && !visited[num]) {

visited[num] = true;

helper(n, s, visited, i, count + 1);

visited[num] = false;

}

}

}

}

**Refer to**

<https://www.lintcode.com/problem/570/solution/17052>

public class Solution {

private int missingNum = -1;

public int findMissing2(int n, String str) {

// write your code here

if(n < 1 || str == null) {

return -1;

}

dfs(n, str, 0, new boolean[n + 1], 0);

return missingNum;

}

private void dfs(int n,

String str,

int startIndex, // start index of each recursion

boolean[] visited, // visited items

Integer count) { // count of added numbers

if(startIndex == str.length() && count == n - 1) {

for(int i = 1; i < n + 1; i++) {

if(!visited[i]) {

missingNum = i;

return;

}

}

return;

}

for(int i = startIndex + 1; i <= startIndex + 2 && i <= str.length(); i++) {

String subString = str.substring(startIndex, i);

int num = Integer.parseInt(subString);

if(num < 1 || num > n // number should be valid

|| visited[num] // number should not be visited

|| Integer.toString(num).length() != subString.length()) { // Make sure the number doesn't start with 0

continue;

}

visited[num] = true;

dfs(n, str, i, visited, count + 1);

visited[num] = false;

}

}

}