**Yelp**

Introduction

Top interview questions asked by Yelp as voted by the community.

This list will be kept up to date as frequent as possible.

Array and Strings

 Two Sum

 Longest Substring Without Repeating Characters

 Longest Common Prefix

 3Sum

 Group Anagrams

 Spiral Matrix

 Merge Sorted Array

 Pascal's Triangle

 Basic Calculator II

 Product of Array Except Self

**Find the Closest Palindrome**

 Isomorphic Strings

 Minimum Index Sum of Two Lists

Linked List

 Merge Two Sorted Lists

 Copy List with Random Pointer

Trees and Graphs

 Word Ladder

 Number of Islands

 Course Schedule

 Reconstruct Itinerary

Recursion

 Letter Combinations of a Phone Number

 Generate Parentheses

 Combination Sum

 Regular Expression Matching

Sorting and Searching

 Merge Intervals

 Meeting Rooms

 Meeting Rooms II

 Random Pick with Weight

 Sliding Window Maximum

 Top K Frequent Elements

 Top K Frequent Words

Dynamic Programming

 Longest Palindromic Substring

 Maximum Subarray

 Best Time to Buy and Sell Stock

 Word Break

 Maximum Product Subarray

 Maximal Square

**Sentence Screen Fitting**

Design

 LRU Cache

 Logger Rate Limiter

 Insert Delete GetRandom O(1)

Others

 Valid Square

 Integer to English Words

**Find the Closest Palindrome**

Given an integer n, find the closest integer (not including itself), which is a palindrome.

The 'closest' is defined as absolute difference minimized between two integers.

**Example 1:**

**Input:** "123"

**Output:** "121"

**Note:**

1. The input **n** is a positive integer represented by string, whose length will not exceed 18.
2. If there is a tie, return the smaller one as answer.

   Hide Hint #1

Will brute force work for this problem? Think of something else.

   Hide Hint #2

Take some examples like 1234, 999,1000, etc and check their closest palindromes. How many different cases are possible?

   Hide Hint #3

Do we have to consider only left half or right half of the string or both?

   Hide Hint #4

Try to find the closest palindrome of these numbers- 12932, 99800, 12120. Did you observe something?

## Solution

#### **Approach #1 Brute Force[Time Limit Exceeded]**

The simplest solution is to consider every possible number smaller than the given number n*n*, starting by decrementing 1 from the given number and go on in descending order. Similarly, we can consider every possible number greater than n*n* starting by incrementing 1 from the given number and going in ascending order. We can continue doing so in an alternate manner till we find a number which is a palindrome.

|  |
| --- |
| public class Solution {  public String nearestPalindromic(String n) {  long num = Long.parseLong(n);  for (long i = 1;; i++) {  if (isPalindrome(num - i))  return "" + (num - i);  if (isPalindrome(num + i))  return "" + (num + i);  }  }  boolean isPalindrome(long x) {  long t = x, rev = 0;  while (t > 0) {  rev = 10 \* rev + t % 10;  t /= 10;  }  return rev == x;  }  } |

**Complexity Analysis**

* Time complexity : O(\sqrt{n})*O*(*n*​). Upto 2\*\sqrt{n}2∗*n*​ numbers could be generated in the worst case.
* Space complexity : O(1)*O*(1). Constant space is used.

#### **Approach #2 Using Math[Accepted]**

**Algorithm**

To understand this method, let's start with a simple illustration. Assume that the number given to us is "abcxy". One way to convert this number into a palindrome is to replicate one half of the string to the other half. If we try replicating the second half to the first half, the new palindrome obtained will be "yxcxy" which lies at an absolute of \left|10000(a-y) + 1000(b-x)\right|∣10000(*a*−*y*)+1000(*b*−*x*)∣ from the original number. But, if we replicate the first half to the second half of the string, we obtain "abcba", which lies at an absolute difference of \left|10(x-b) + (y-a)\right|∣10(*x*−*b*)+(*y*−*a*)∣. Trying to change c*c* additionaly in either case would incur an additional value of atleast 100 in the absolute difference.

From the above illustration, we can conclude that if replication is used to generate the palindromic number, we should always replicate the first half to the second half. In this implementation, we've stored such a number in a*a* at a difference of diff1*diff*1 from n*n*.

But, there exists another case as well, where the digit at the middle index is incremented or decremented. In such cases, it could be profitable to make changes to the central digit only since such changes could lead to a palindrome formation nearer to the original digit. e.g. 10987. Using the above criteria, the palindrome obtained will be 10901 which is at a more difference from 10987 than 11011. A similar situation occurs if a 0 occurs at the middle digit. But, again as discussed previously, we need to consider only the first half digits to obtain the new palindrome. This special effect occurs with 0 or 9 at the middle digit since, only decrementing 0 and incrementing 9 at that digit place can lead to the change in the rest of the digits towards their left. In any other case, the situation boils down to the one discussed in the first paragraph.

Now, whenever we find a 0 near the middle index, in order to consider the palindromes which are lesser than n*n*, we subtract a 1 from the first half of the number to obtain a new palindromic half e.g. If the given number n*n* is 20001, we subtract a 1 from 200 creating a number of the form 199xx. To obtain the new palindrome, we replicate the first half to obtain 19991. Taking another example of 10000, (with a 1 at the MSB), we subtract a 1 from 100 creating 099xx as the new number transforming to a 9999 as the new palindrome. This number is stored in b*b* having a difference of diff2*diff*2 from n*n*

Similar treatment needs to be done with a 9 at the middle digit, except that this time we need to consider the numbers larger than the current number. For this, we add a 1 to the first half. e.g. Taking the number 10987, we add a 1 to 109 creating a number of the form 110xx(11011 is the new palindrome). This palindrome is stored in c*c* having a difference of diff3*diff*3 from n*n*.

Out of these three palindromes, we can choose the one with a minimum difference from n*n*. Further, in case of a tie, we need to return the smallest palindrome obtained. For resolving this tie's conflict, we can observe that a tie is possible only if one number is larger than n*n* and another is lesser than n*n*. Further, we know that the number b*b* is obtained by decreasing n*n*. Thus, in case of conflict between b*b* and any other number, we need to choose b*b*. Similarly, c*c* is obtained by increasing n*n*. Thus, in case of a tie between c*c* and any other number, we need to choose the number other than c*c*.

|  |
| --- |
| public class Solution {  public String mirroring(String s) {  String x = s.substring(0, (s.length()) / 2);  return x + (s.length() % 2 == 1 ? s.charAt(s.length() / 2) : "") + new StringBuilder(x).reverse().toString();  }  public String nearestPalindromic(String n) {  if (n.equals("1"))  return "0";  String a = mirroring(n);  long diff1 = Long.MAX\_VALUE;  diff1 = Math.abs(Long.parseLong(n) - Long.parseLong(a));  if (diff1 == 0)  diff1 = Long.MAX\_VALUE;  StringBuilder s = new StringBuilder(n);  int i = (s.length() - 1) / 2;  while (i >= 0 && s.charAt(i) == '0') {  s.replace(i, i + 1, "9");  i--;  }  if (i == 0 && s.charAt(i) == '1') {  s.delete(0, 1);  int mid = (s.length() - 1) / 2;  s.replace(mid, mid + 1, "9");  } else  s.replace(i, i + 1, "" + (char)(s.charAt(i) - 1));  String b = mirroring(s.toString());  long diff2 = Math.abs(Long.parseLong(n) - Long.parseLong(b));  s = new StringBuilder(n);  i = (s.length() - 1) / 2;  while (i >= 0 && s.charAt(i) == '9') {  s.replace(i, i + 1, "0");  i--;  }  if (i < 0) {  s.insert(0, "1");  } else  s.replace(i, i + 1, "" + (char)(s.charAt(i) + 1));  String c = mirroring(s.toString());  long diff3 = Math.abs(Long.parseLong(n) - Long.parseLong(c));  if (diff2 <= diff1 && diff2 <= diff3)  return b;  if (diff1 <= diff3 && diff1 <= diff2)  return a;  else  return c;  }  } |

**Complexity Analysis**

* Time complexity : O(l)*O*(*l*). Scanning, insertion, deletion,, mirroring takes O(l)*O*(*l*), where l*l* is the length of the string.
* Space complexity : O(l)*O*(*l*). Temporary variables are used to store the strings.

**Sentence Screen Fitting**

Given a rows x cols screen and a sentence represented by a list of **non-empty** words, find **how many times** the given sentence can be fitted on the screen.

**Note:**

1. A word cannot be split into two lines.
2. The order of words in the sentence must remain unchanged.
3. Two consecutive words **in a line** must be separated by a single space.
4. Total words in the sentence won't exceed 100.
5. Length of each word is greater than 0 and won't exceed 10.
6. 1 ≤ rows, cols ≤ 20,000.

**Example 1:**

**Input:**

rows = 2, cols = 8, sentence = ["hello", "world"]

**Output:**

1

**Explanation:**

hello---

world---

The character '-' signifies an empty space on the screen.

**Example 2:**

**Input:**

rows = 3, cols = 6, sentence = ["a", "bcd", "e"]

**Output:**

2

**Explanation:**

a-bcd-

e-a---

bcd-e-

The character '-' signifies an empty space on the screen.

**Example 3:**

**Input:**

rows = 4, cols = 5, sentence = ["I", "had", "apple", "pie"]

**Output:**

1

**Explanation:**

I-had

apple

pie-I

had--

The character '-' signifies an empty space on the screen.