<https://leetcode.com/problems/find-the-closest-palindrome/description/>

Given a string n representing an integer, return the closest integer (not including itself), which is a palindrome. If there is a tie, return the smaller one.

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

**Example 1:**

**Input:** n = "123"

**Output:** "121"

**Example 2:**

**Input:** n = "1"

**Output:** "0"

**Explanation:** 0 and 2 are the closest palindromes but we return the smallest which is 0.

**Constraints:**

1 <= n.length <= 18

n consists of only digits.

n does not have leading zeros.

n is representing an integer in the range [1, 1018 - 1].

**Attempt 1: 2025-03-14**

**Refer to Deepseek**

To solve this problem, we need to find the closest palindrome to a given integer represented as a string. The closest palindrome is defined as the number that has the smallest absolute difference with the given number. If there is a tie, the smaller number is chosen. The solution must not return the same number as the input even if the input is a single-digit.

**Approach**

Generate Candidates: Generate potential palindrome candidates based on the following:

**Mirror of the First Half**: Create a palindrome by mirroring the first half of the number.

**Higher and Lower Candidates**: Adjust the first half by incrementing and decrementing it, then generate palindromes from these adjusted values.

**Edge Candidates**: Consider numbers like 999...999 (all 9s) and 100...001 (starting and ending with 1 with zeros in between), which are common edge cases for palindromes.

Select Closest Palindrome: Compare all generated candidates to find the one with the smallest absolute difference from the original number. In case of a tie, choose the smaller number.

**Solution Code**

class Solution {

public String nearestPalindromic(String n) {

long originalNum = Long.parseLong(n);

int len = n.length();

List<Long> candidates = new ArrayList<>();

// Generate mirror candidate

String mirrorCandidate = generateMirror(n);

long mirrorNum = Long.parseLong(mirrorCandidate);

if (mirrorNum != originalNum) {

candidates.add(mirrorNum);

}

// Generate higher candidate

String higherCandidate = generateHigher(n);

candidates.add(Long.parseLong(higherCandidate));

// Generate lower candidate

String lowerCandidate = generateLower(n);

if (!lowerCandidate.isEmpty()) {

candidates.add(Long.parseLong(lowerCandidate));

}

// Generate 999...999

if (len > 1) {

String all9 = String.join("", Collections.nCopies(len - 1, "9"));

candidates.add(Long.parseLong(all9));

}

// Generate 100...001

String upperEdge = "1" + String.join("", Collections.nCopies(len - 1, "0")) + "1";

candidates.add(Long.parseLong(upperEdge));

// Handle case for numbers like 10, 100 where lower candidate might not be generated

if (len == 1) {

candidates.add(0L);

}

// Find the closest candidate

long minDiff = Long.MAX\_VALUE;

long result = Long.MAX\_VALUE;

for (long candidate : candidates) {

long diff = Math.abs(candidate - originalNum);

if (diff < minDiff) {

minDiff = diff;

result = candidate;

} else if (diff == minDiff && candidate < result) {

result = candidate;

}

}

return String.valueOf(result);

}

private String generateMirror(String n) {

String prefix = getPrefix(n);

boolean isEven = (n.length() % 2 == 0);

return generateMirrorFromPrefix(prefix, isEven);

}

private String generateHigher(String n) {

String prefix = getPrefix(n);

long prefixNum = Long.parseLong(prefix) + 1;

String newPrefix = String.valueOf(prefixNum);

boolean isEven = (n.length() % 2 == 0);

return generateMirrorFromPrefix(newPrefix, isEven);

}

private String generateLower(String n) {

String prefix = getPrefix(n);

long prefixNum = Long.parseLong(prefix);

prefixNum -= 1;

if (prefixNum < 0) {

return "";

}

String newPrefix = String.valueOf(prefixNum);

if (newPrefix.length() == 0) {

return "";

}

boolean isEven = (n.length() % 2 == 0);

return generateMirrorFromPrefix(newPrefix, isEven);

}

private String generateMirrorFromPrefix(String prefix, boolean isEven) {

if (isEven) {

return prefix + new StringBuilder(prefix).reverse().toString();

} else {

if (prefix.length() <= 1) {

return prefix;

}

String part = prefix.substring(0, prefix.length() - 1);

return prefix + new StringBuilder(part).reverse().toString();

}

}

private String getPrefix(String n) {

int len = n.length();

if (len % 2 == 0) {

return n.substring(0, len / 2);

} else {

return n.substring(0, len / 2 + 1);

}

}

}

**Explanation**

Mirror Candidate: This candidate is generated by taking the first half of the number and mirroring it. For even lengths, the entire first half is mirrored. For odd lengths, the first half plus the middle digit is taken, and the part before the middle digit is mirrored.

Higher and Lower Candidates: These are generated by incrementing and decrementing the first half of the number and then creating a palindrome from the adjusted value.

Edge Candidates: These include numbers like all 9s (e.g., 999) and numbers like 1001, which are common edge cases for palindromes.

Comparison: All candidates are compared based on their absolute difference from the original number. The smallest difference is chosen, with ties resolved by selecting the smaller number.

This approach efficiently generates the most likely candidates and ensures the closest palindrome is found with minimal computational overhead. The correction ensures that even single-digit inputs are handled correctly by generating the closest palindromes around them.

**One question, why we don't choose to mirror the second half of the original number ? but looks like keep mirror the first half ? for example, if a number is 54321, if we choose mirror the second half and create 12321 better than mirror the first half as 54345 ?**

**Impact on Magnitude:**

**Adjusting the first half (higher-order digits) has a larger numerical impact, we should keep higher-order digits but modify lower-order digits. For example: For 54321 if we change its high-order digits by mirror the second half, then we have 12321, The difference between two numbers as 54321 - 12321 which is siginificant larger than mirror lower-order digits by mirror the first half (which we will have 54345, the difference will be 54345 - 54321).**

**Final Answer:**

**Mirroring the first half allows the algorithm to prioritize adjustments to higher-order digits, generating candidates that are numerically closer to the original number.** This approach, combined with edge cases, ensures the closest palindrome is found efficiently. For 54321, the closest palindrome is 54345, not 12321.

**Refer to**

<https://leetcode.com/problems/find-the-closest-palindrome/editorial/>

**Approach 1: Find Previous and Next Palindromes**

**Intuition**

The problem asks us to find the closest palindrome to a given integer n represented as a string. The string length is at most 18, meaning n can be as large as 999,999,999,999,999,999. The goal is to return the nearest palindrome to n that is not equal to n itself, minimizing the absolute difference.

To solve this, we can think of a palindrome as a number where the first half is mirrored to create the second half. For example, the palindrome for 12321 is formed by reversing the first half (12) and appending it to itself (12 -> 12321). This observation is key to finding the closest palindrome.

If we consider changing the second half of n to match the reverse of the first half, we might obtain a palindrome close to n. However, there are cases where this method might not give us the optimal answer, particularly for odd-length strings or when small adjustments to the first half could yield a closer palindrome.

For instance, consider n = 139. If we mirror the first half (13), we get 131, but a closer palindrome is 141. Therefore, it's important to also check palindromes formed by slightly adjusting the first half of n:

Same Half: Create a palindrome by mirroring the first half.

Decremented Half: Create a palindrome by decrementing the first half by 1 and mirroring it.

Incremented Half: Create a palindrome by incrementing the first half by 1 and mirroring it.

Note: Adding +1 or subtracting -1 to/from the first half ensures that we stay as close as possible to the original number while creating new potential palindromes. If we were to add or subtract a larger value, such as +2 or -2, the resulting palindrome would be farther away from the original number, potentially missing a closer palindrome, and it's given that we need to find the closest palindrome.

In addition to these cases, we must handle edge cases where n is close to numbers like 1000, 10000, etc., or very small numbers like 11 or 9. These can produce palindromes like 99, 999, or 101, 1001, which might be closer to n.

To summarize, we need to check the following five candidates:

Palindrome formed from the first half of n.

Palindrome formed from the first half decremented by 1.

Palindrome formed from the first half incremented by 1.

Nearest palindrome of the form 99, 999, etc.

Nearest palindrome of the form 101, 1001, etc.

After generating these candidates, we compare them to n and choose the one with the smallest absolute difference.

**Algorithm**

Main Function - nearestPalindromic(n)

Calculate the length of n and determine the midpoint.

Extract the first half of the number.

Generate possible palindromic candidates and append them to possibilities list:

Mirror the first half and append it to the string.

Mirror the first half incremented by 1 and append it to the string.

Mirror the first half decremented by 1 and append it to the string.

Add the form 999....

Add the form 100...001.

Find the nearest palindromic number by comparing absolute differences.

Return the closest palindrome.

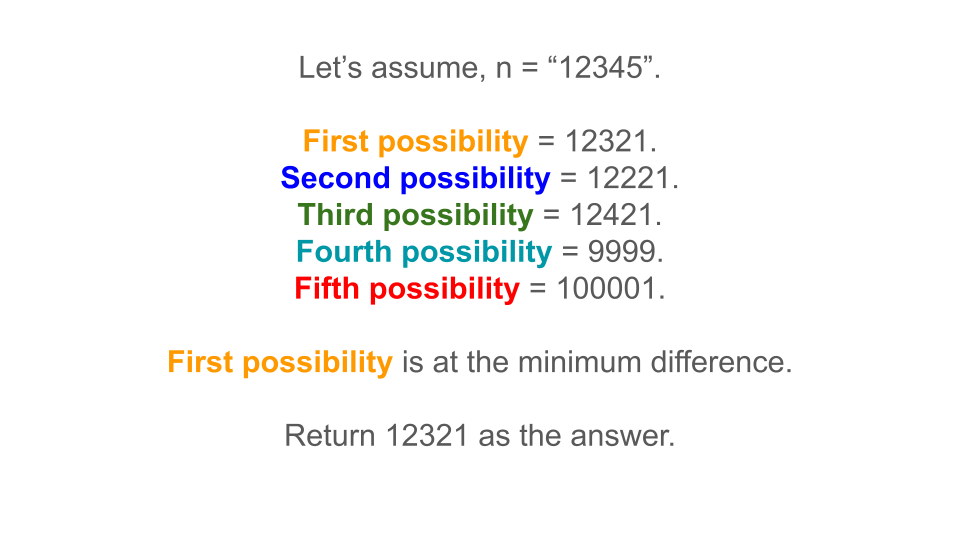
Helper Function - halfToPalindrome(left, even)

Initialize res with left.

If the length is odd, divide left by 10.

Mirror the digits of left to form a palindrome.

Return the palindrome res.



**Implementation**

class Solution {

public String nearestPalindromic(String n) {

int len = n.length();

int i = len % 2 == 0 ? len / 2 - 1 : len / 2;

long firstHalf = Long.parseLong(n.substring(0, i + 1));

/\*

Generate possible palindromic candidates:

1. Create a palindrome by mirroring the first half.

2. Create a palindrome by mirroring the first half incremented by 1.

3. Create a palindrome by mirroring the first half decremented by 1.

4. Handle edge cases by considering palindromes of the form 999...

and 100...001 (smallest and largest n-digit palindromes).

\*/

List<Long> possibilities = new ArrayList<>();

possibilities.add(halfToPalindrome(firstHalf, len % 2 == 0));

possibilities.add(halfToPalindrome(firstHalf + 1, len % 2 == 0));

possibilities.add(halfToPalindrome(firstHalf - 1, len % 2 == 0));

possibilities.add((long) Math.pow(10, len - 1) - 1);

possibilities.add((long) Math.pow(10, len) + 1);

// Find the palindrome with minimum difference, and minimum value.

long diff = Long.MAX\_VALUE, res = 0, nl = Long.parseLong(n);

for (long cand : possibilities) {

if (cand == nl) continue;

if (Math.abs(cand - nl) < diff) {

diff = Math.abs(cand - nl);

res = cand;

} else if (Math.abs(cand - nl) == diff) {

res = Math.min(res, cand);

}

}

return String.valueOf(res);

}

private long halfToPalindrome(long left, boolean even) {

// Convert the given half to palindrome.

long res = left;

if (!even) left = left / 10;

while (left > 0) {

res = res \* 10 + (left % 10);

left /= 10;

}

return res;

}

}

**Complexity Analysis**

Let *n* be the number of digits in the input number.

Time complexity: *O*(*n*)

We perform operations on exactly 5 strings. The palindrome construction for each string takes *O*(*n*) time. Therefore, total time complexity is given by *O*(*n*).

Space complexity: *O*(*n*)

We store the 5 possible candidates in the possibilities array. Apart from this, the built-in functions used to make the firstHalf can potentially lead to *O*(*n*) space complexity, as they copy the characters into a new String. Therefore, the total space complexity is *O*(*n*).

**Approach 2: Binary Search**

**Intuition**

Another way to solve the problem is by using binary search. The task is to find the smallest palindrome greater than n and the largest palindrome smaller than n, then return the one with the smallest absolute difference. Since this is a minimization/maximization, we can try to use binary search to solve this problem. But, our search space should be sorted to apply binary search. Observe that when you construct the palindromes using the first half for two integers, then the greater integer would always have it's constructed palindrome greater. Therefore, our search space is sorted in a non-decreasing order.

Given that palindromes are symmetric numbers, we can search within a specific range by leveraging binary search. The key is to first determine potential palindromes by constructing them based on the first half of n.

Finding the Next Palindrome:

Start with the left boundary as n + 1 and the right boundary as an infinitely large value.

Perform binary search within this range. For each midpoint value, construct the palindrome by mirroring its first half.

If the constructed palindrome is greater than n, shift the search to the left (smaller values). Otherwise, move to the right.

Finding the Previous Palindrome:

Start with the left boundary as 0 and the right boundary as n - 1.

Perform binary search, constructing palindromes as above.

If the constructed palindrome is smaller than n, shift the search to the right (larger values). Otherwise, move to the left.

Binary search efficiently narrows down the range of possible palindromes, finding the closest one that is greater and the closest one that is smaller. Once we have these two candidates, we simply compare their differences with n to determine the closest palindrome.

This approach is particularly useful when n is large, as it reduces the search space compared to checking all potential candidates directly.

**Algorithm**

convert(num)

Convert the number num to a string s.

Identify the midpoint indices l (left) and r (right).

Mirror the left half of the string s onto the right half to create a palindrome.

Return the palindrome as a long integer.

nextPalindrome(num)

Initialize left to 0 and right to num.

Use binary search to find the next palindrome greater than num:

Calculate mid as the midpoint between left and right.

Convert mid to a palindrome using convert(mid).

If the palindrome is less than num, update ans to the palindrome and set left to mid + 1.

Otherwise, set right to mid - 1.

Return the result ans.

previousPalindrome(num)

Initialize left to num and right to a large value (1e18).

Use binary search to find the previous palindrome smaller than num:

Calculate mid as the midpoint between left and right.

Convert mid to a palindrome using convert(mid).

If the palindrome is greater than num, update ans to the palindrome and set right to mid - 1.

Otherwise, set left to mid + 1.

Return the result ans.

Main Function - nearestPalindromic(n)

Convert the input string n to a long integer num.

Call nextPalindrome(num) to find the next palindrome greater than num.

Call previousPalindrome(num) to find the previous palindrome smaller than num.

Compare the differences between num and the two palindromes found:

If the difference with the next palindrome is less than or equal to the difference with the previous palindrome, return the next palindrome. Otherwise, return the previous palindrome as a string.

**Implementation**

class Solution {

// Convert to palindrome keeping first half constant.

private long convert(long num) {

String s = Long.toString(num);

int n = s.length();

int l = (n - 1) / 2, r = n / 2;

char[] sArray = s.toCharArray();

while (l >= 0) {

sArray[r++] = sArray[l--];

}

return Long.parseLong(new String(sArray));

}

// Find the previous palindrome, just smaller than n.

private long previousPalindrome(long num) {

long left = 0, right = num;

long ans = Long.MIN\_VALUE;

while (left <= right) {

long mid = (right - left) / 2 + left;

long palin = convert(mid);

if (palin < num) {

ans = palin;

left = mid + 1;

} else {

right = mid - 1;

}

}

return ans;

}

// Find the next palindrome, just greater than n.

private long nextPalindrome(long num) {

long left = num, right = (long) 1e18;

long ans = Long.MIN\_VALUE;

while (left <= right) {

long mid = (right - left) / 2 + left;

long palin = convert(mid);

if (palin > num) {

ans = palin;

right = mid - 1;

} else {

left = mid + 1;

}

}

return ans;

}

public String nearestPalindromic(String n) {

long num = Long.parseLong(n);

long a = previousPalindrome(num);

long b = nextPalindrome(num);

if (Math.abs(a - num) <= Math.abs(b - num)) {

return Long.toString(a);

}

return Long.toString(b);

}

}

**Complexity Analysis**

Let *m* be the input number and *n* be the number of digits in it.

Time complexity: *O*(*n*⋅*log*(*m*))

We perform two binary search operations on a search space of size m, and in each operation iterate through all the digits. Therefore, the total time complexity is given by *O*(*n*⋅*log*(*m*)).

Space complexity: *O*(*n*)

The space complexity is primarily determined by the storage needed for the string representation of the number and the intermediate list or character array used for manipulation. Since these data structures are proportional to the number of digits in *O*(*n*), the total space complexity is *O*(*n*).

For C++: to\_string(num) - Converts the number to a string, which requires space proportional to the number of digits in *O*(*n*), i.e., *O*(*n*).

For Java: Long.toString(num) - Converts the number to a string, requiring *O*(*n*) space.

For Python: ''.join(s\_list) - Creates a new string from the list, requiring *O*(*n*) space.