647. Palindromic Substrings

915 52

Description (/problems/palindromic-substrings/description/) Hints (/problems/palindromic-substrings/hints/) Submissions (/problems/palinc

Quick Navigation View in Article (/articles/palindromic-substrings/) Notes

Approach #1: Expand Around Center [Accepted]

Intuition

Let `N` be the length of the string. The middle of the palindrome could be in one of `2N - 1` positions: either at letter or between two letters.

For each center, let's count all the palindromes that have this center. Notice that if `[a, b]` is a palindromic interval (meaning `S[a], S[a+1], ..., S[b]` is a palindrome), then `[a+1, b-1]` is one too.

Algorithm

For each possible palindrome center, let's expand our candidate palindrome on the interval `[left, right]` as long as we can. The condition for expanding is `left >= 0 and right < N and S[left] == S[right]`. That means we want to count a new palindrome `S[left], S[left+1], ..., S[right]`.

Java Python Copy

Complexity Analysis

- Time Complexity: $O(N^2)$ where N is the length of `S`. Each expansion might do $O(N)$ work.
- Space Complexity: $O(1)$.

Approach #2: Manacher's Algorithm [Accepted]

Intuition

Manacher's algorithm is a textbook algorithm that finds in linear time, the maximum size palindrome for any possible palindrome center. If we had such an algorithm, finding the answer is straightforward.

What follows is a discussion of why this algorithm works.

Algorithm

Our loop invariants will be that `center, right` is our knowledge of the palindrome with the largest right-most boundary with `center < i`, centered at `center` with right-boundary `right`. Also, `i > center`, and we've already computed all `Z[j]`'s for `j < i`.

When `i < right`, we reflect `i` about `center` to be at some coordinate `j = 2 * center - i`. Then, limited to the interval with radius `right - i` and center `i`, the situation for `Z[i]` is the same as for `Z[j]`.

For example, if at some time `center = 7, right = 13, i = 10`, then for a string like `A = '@#A#B#A#A#B#A#$'`, the `center` is at the `'#'` between the two middle `'A'`'s, the right boundary is at the last `'#'`, `i` is at the last `'B'`, and `j` is at the first `'B'`.

Notice that limited to the interval `[center - (right - center), right]` (the interval with center `center` and right-boundary `right`), the situation for `i` and `j` is a reflection of something we have already computed. Since we already know `Z[j] = 3`, we can quickly find `Z[i] = min(right - i, Z[j]) = 3`.

Now, why is this algorithm linear? The while loop only checks the condition more than once when `Z[i] = right - i`. In that case, for each time `Z[i] += 1`, it increments `right`, and `right` can only be incremented up to `2*N+2` times.

Finally, we sum up `(v+1) / 2` for each `v in Z`. Say the longest palindrome with some given center C has radius R. Then, the substring with center C and radius R-1, R-2, R-3, ..., 0 are also palindromes. Example: `abcdedcba` is a palindrome with center `e`, radius 4: but `e`, `ded`, `cdedc`, `bcdedcb`, and `abcdedcba` are all palindromes.

We are dividing by 2 because we were using half-lengths instead of lengths. For example we actually had the palindrome `a#b#c#d#e#d#c#b#a`, so our length is twice as big.

Java Python Copy

Complexity Analysis

- Time Complexity: $O(N)$ where N is the length of `S`. As discussed above, the complexity is linear.
- Space Complexity: $O(N)$, the size of `A` and `Z`.

Analysis written by: @awice (https://leetcode.com/awice).

Comments: 6

Sort By

Type comment here... (Markdown is supported)

Preview Post