Palindrome Count – Analysis

Problem: Iliyan Yordanov, Analysis: Ivan Lupov

Subtask 1

The constraint $N \leq 20$ allows any solution iterating over the possible configurations of removed and kept letters to pass. One such is iterating over the integers from 1 to 2^N and using their binary representations as bitmasks – if the bitmask has 1 on some position, we keep the corresponding letter, otherwise remove it. For each configuration check that it is a palindrome with a simple for-loop. Final complexity will be $O(2^N N)$.

Subtask 2

The higher constraints and the problem dealing with counting some number of objects should suggest a dynamic programming solution. Let's establish $\text{dp}[l][r]$ as the number of palindromes considering only the substring $s[l:r]$ of the original string.

Let's first deal with the base cases of this approach: $\text{dp}[i][i] = 1$ for all $1 \leq i \leq N$. The transitions should be separated in two cases – those that simply elongate the interval and those that create new palindromes.

If $s[l] \neq s[r]$, all the palindromes we should count can be calculated from previous values of dp, namely $\text{dp}[l][r] = \text{dp}[l][r-1] + \text{dp}[l+1][r] - \text{dp}[l+1][r-1]$. Note that the last term takes care of any overcounting from palindromes that appear in both the $[l, r-1]$ and the $[l+1, r]$ intervals.

Otherwise, in the case that $s[l] = s[r]$, new palindromes will appear that append and prepend the character $s[l]$. Do not forget that there is also the palindrome made up of two repetitions of that character. Thus dp, namely $\text{dp}[l][r] = \text{dp}[l][r-1] + \text{dp}[l+1][r] + 1$ is the transition in this case. Here we don't take care of any overcounting – the "overcounted" palindromes are simply extended with the character $s[l]$ in the beginning and the end. Final time and memory complexity is $O(N^2)$.

Subtask 3

Although $O(N^2)$ time complexity is good enough for $N \leq 9000$, this is not the case for the memory consumption. Here we will look for a solution that uses $O(N)$ memory.

In the case of "range dp" solutions (such is our solution to subtask 2) it is often obvious that from the three quantities of an interval – its left endpoint, right endpoint and length – we only need two of them to clearly define the interval and the third one can be computed. If our solution has to calculate the answers for longer ranges from the answers for shorter ones, we might as well exclude one of the endpoints from the dynamic programming and include in its place the length of the range. That way we have $\text{dp}[l][len]$ – the number of palindromes in the range $[l, l + len - 1]$. How does our solution change?

For the base case, we instead have $\text{dp}[i][1] = 1$ for all $1 \leq i \leq N$.

In the case of $s[l] \neq s[r]$ (more precisely, $s[l] \neq s[l + len - 1]$) the transition will look like this: $\text{dp}[l][len] = \text{dp}[l][len - 1] + \text{dp}[l+1][len-1] - \text{dp}[l+1][len-2]$, and for $s[l] = s[r]$ we get $\text{dp}[l][len] = \text{dp}[l][len-1] + \text{dp}[l+1][len-1] + 1$.

We can notice that in this form, the value of $\text{dp}[l][len]$ only depends on the values in $\text{dp}[:][len-1]$ and $\text{dp}[:][len-2]$ – so we can actually only keep 3 lines of the dynamic programming table as we are computing it, instead of all N.