Notes on Leetcode

Hanlin He

August 14, 2017

1 Longest Palindromic Substring (P005)

1.1 Brute Force

First idea is just brute force.

> Traverse the string.
>
> For each character, find the longest palindromic substring.
>
> In order to do that, traverse the substring began at that character. If a certain substring is palindromic, mark the length and continue.

The biggest problem for this brute force solution is two characters in the string might be repeatedly compared in all n round. In all the effort to optimize this solution, it still exceeds time limits occasionally.

1.2 Expanding

So here comes the second intuitive idea.

> Consider a character at index i in the middle of the string.
>
> If it is the center of a palindromic substring, then all the characters at its left and right must be mirrored. Therefore, we can check the characters at $i \pm k$ recursively.
>
> If we traverse the string, perform the previous operations, we can calculate all the palindromic substrings centered at each character.

This idea hugely reduce comparisons. Since each pair of characters can only be centered at one character. And each character is visited only once. Thus each pair of characters are compared only once. However, it contains an obvious ambiguity: What if a palindromic is even in length? There is no center character for an even length string.

To generalize the solution, here comes a modification: Use a series of same characters as center, rather than an aimless character. Here is the optimized solution.

1. Traverse the string, continue if same character found, stop until finding a different character.
2. View the series of same characters just found as center, expanding at both direction, until find unmatching character. The substring between two ends is a palindromic substring. Mark length and continue at the different character found in 1.

This solution turns out to be even more efficient than the original idea since long series of same character can be stepped over.

1.3 Longest Common Substring

Although the previous solution is quite efficient, we can still view the problem from another aspect.

> If we reverse the original string, the palindromic substring must be a common string of the reversed and original string. In this way, finding the longest palindromic substring is equals to finding the longest common substring between two string.

Nevertheless, the idea is not correct in all circumstances. As an example given in the editorial:

- $S = {}''$`abacdfgdcaba`$''$
- $S_r = {}''$`abacdgfdcaba`$''$

Obviously, the common substring "abacd" is not palindromic. A quick fix to this problem is check every time we find a common substring.

To solve the longest common substring problem, we can implement a $\mathcal{O}(n^2)$ dynamic programming solution, using $\mathcal{O}(n^2)$ or $\mathcal{O}(n)$ space.

1.4 To-Do

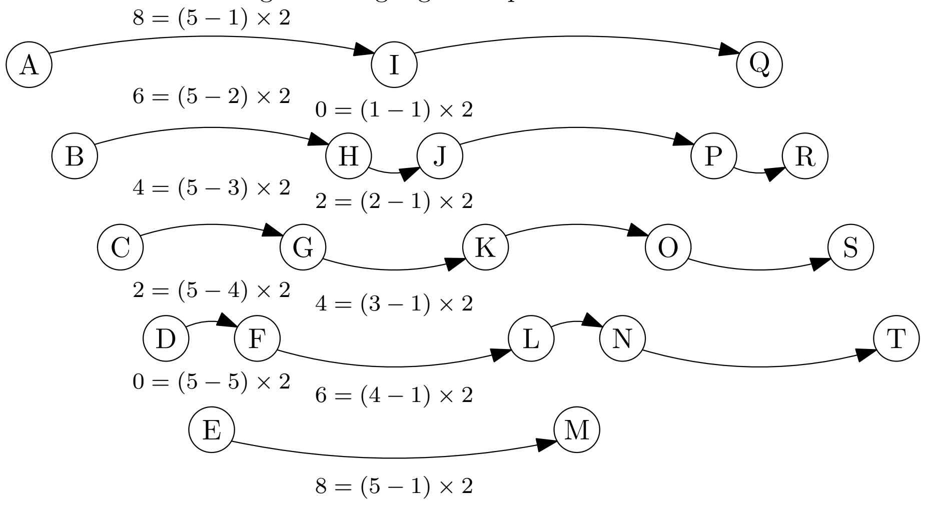
Besides the solution above, an even more efficient solution named Manacher's Algorithm is available, which is left as to do in the section.

2 ZigZag Conversion (P006)

2.1 Pattern and Mapping

First idea is find the pattern of the ZigZag conversion, as shown in fig. 1.

Figure 1: ZigZag Example and Pattern



It easy to see that for one round, there are $2 \times n - 1$ characters. At each line, we need to take two step to get to the next round (consider the first and last line take a 'step' of zero step forward). Furthermore, we can conclude the two step are as follow:

- $Step1 = (NumOfRow - CurrentRow) \times 2$.
- $Step2 = (CurrentRow - 1) \times 2$.

Thus, we can directly traverse each line of the converted string, find the corresponding characters from the original string and fill it in.

2.2 Min Priority Queue

View the problem in another aspect:

> When we traverse the original string, if we can determine the line number of each character in the ZigZag conversion, the line number can be used as a priority.
>
> By doing this, if we put all the character into a min priority queue, the sequence would naturally consistent with the ZigZag requirement, which characters with less line number would appear first.

In addition, we need to ensure for characters with same line number, they are FIFO in the min priority queue. One way to achieve this is use the tuple of line number and index in original string together as priority. For example (line, index) of (2, 4) should appear before (2, 8).

My first implementation of this method is in python. The solution turned out to be really slow.

Sad.

3 3Sum (P015)

3.1 Based on Two Sum

First idea is based on 001.Two Sum. It's rather intuitive. Since two sum can get two number with target sum, if we set the target as the negative of $\texttt{nums}[\texttt{i}]$, and get another two number $\texttt{nums}[\texttt{j}]$ and $\texttt{nums}[\texttt{k}]$ with sum of $-\texttt{nums}[\texttt{i}]$, we automatically get a triplet with sum of $\texttt{nums}[\texttt{i}] + \texttt{nums}[\texttt{j}] + \texttt{nums}[\texttt{k}] = 0$. We can repeat the process for all number in the array to get all the possible triplets. The running time is intuitively $\mathcal{O}(n^2)$, since two sum take $\mathcal{O}(n)$ and was repeated n times.

To simplify the operation of duplicating number, we may sort the array first. Since sorting takes $\mathcal{O}(n \log n)$ time, less than $\mathcal{O}(n^2)$, sorting time can be discard.

Although the solution is after all $\mathcal{O}(n^2)$, the overall running time is unacceptably slow, with my C implementation.

The main reason is to simplify the hash function, I let the index equals to the number itself plus a rather big number, so that even negative number can directly map into an array. The big number is unfortunately set to 100000. And to check the existance of an integer, the condition became **if** (h[x + IMAX + 1] != 0). Therefore, each time hash table was accessed, it must move at least 100000 to get the hash value, greatly consuming time without any alarm.

3.2 Two Pointers

Although previous solution used solution for two sum, it failed to see another chance to reduce time and space.

> *The array is sorted in the first step.*

An sorted array is each to manipulate. We can still use the idea of turning three sum into two sum, but when dealing with sorted array, two sum can easily implemented using two pointers

Let two pointers `i` `j` start at each side, where $i < j$. For a given *target*, if $\texttt{nums}[\texttt{i}] + \texttt{nums}[\texttt{j}] < target$. We can move i rightward to increase the sum of the two. Or on the other hand if $\texttt{nums}[\texttt{i}] + \texttt{nums}[\texttt{j}] > target$. We can move j leftward to decrease the sum of the two. Until otherwise, if $\texttt{nums}[\texttt{i}] + \texttt{nums}[\texttt{j}] = target$, we found one answer, or $i > j$, there is no possible answer.

The biggest advantage is this solution does not use hash table. Only traverse through the array is enough to get all possible answers.

4 First Missing Positive (P041)

Requirement asked for $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space. The intuitive solution would be Bucket sort. The only subtle point is in memory bucket sort.

Since only the first missing positive was demanded. We can can consider the array itself a bucket, and switch each number to its corresponding index. After a switch, switch again until the index was the same as the number. OR result in one of the following situations:

- Negative number in current index.
- Current number greater than the size of the array.
- The target index (to be switched to) has already match (number within is the same as the index).
- (Redundant) Index reach array size.

After all switches, traverse the array. The first encountered index with number different from the index would be the first missing positive.

Each number with range of the array size could be switched at most once. Hence switching is $\mathcal{O}(n)$.

In actual implementation of C, to accommodate all possible integer less than the array size, the number could be placed in the minus one index, i.e., 1 should be put in index 0.