Palindrome Partitioning II

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of *s*.

For example, given *s* = `"aab"`,
Return `1` since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

Solution 1

```
class Solution {
public:
    int minCut(string s) {
        int n = s.size();
        vector<int> cut(n+1, 0);  // number of cuts for the first k characters
        for (int i = 0; i <= n; i++) cut[i] = i-1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; i-j >= 0 && i+j < n && s[i-j]==s[i+j] ; j++) // odd length palindrome
                cut[i+j+1] = min(cut[i+j+1],1+cut[i-j]);

            for (int j = 1; i-j+1 >= 0 && i+j < n && s[i-j+1] == s[i+j]; j++) // even length palindrome
                cut[i+j+1] = min(cut[i+j+1],1+cut[i-j+1]);
        }
        return cut[n];
    }
};
```

written by tqlong original link here

Solution 2

Calculate and maintain 2 DP states:

1. pal[i][j] , which is whether s[i..j] forms a pal
2. d[i], which is the minCut for s[i..n-1]

Once we comes to a pal[i][j]==true:

- if j==n-1, the string s[i..n-1] is a Pal, minCut is 0, d[i]=0;
- else: the current cut num (first cut s[i..j] and then cut the rest s[j+1...n-1]) is 1+d[j+1], compare it to the exisiting minCut num d[i], repalce if smaller.

d[0] is the answer.

```
class Solution {
   public:
        int minCut(string s) {
            if(s.empty()) return 0;
            int n = s.size();
            vector<vector<bool>> pal(n,vector<bool>(n,false));
            vector<int> d(n);
            for(int i=n-1;i>=0;i--)
            {
                d[i]=n-i-1;
                for(int j=i;j<n;j++)
                {
                    if(s[i]==s[j] && (j-i<2 || pal[i+1][j-1]))
                    {
                       pal[i][j]=true;
                       if(j==n-1)
                           d[i]=0;
                       else if(d[j+1]+1<d[i])
                           d[i]=d[j+1]+1;
                    }
                }
            }
            return d[0];
        }
   };
```

written by heiyanbin original link here

Solution 3

One typical solution is DP based. Such solution first constructs a two-dimensional bool array isPalin to indicate whether the sub-string s[i..j] is palindrome. To get such array, we need O(N^2) time complexity. Moreover, to get the minimum cuts, we need another array minCuts to do DP and minCuts[i] saves the minimum cuts found for the sub-string s[0..i-1]. minCuts[i] is initialized to i-1, which is the maximum cuts needed (cuts the string into one-letter characters) and minCuts[0] initially sets to -1, which is needed in the case that s[0..i-1] is a palindrome. When we construct isPalin array, we update minCuts everytime we found a palindrome sub-string, i.e. if s[i..j] is a palindrome, then minCuts[j+1] will be updated to the minimum of the current minCuts[j+1] and minCut[i]+1(i.e. cut s[0..j] into s[0,i-1] and s[i,j]). At last, we return minCuts[N]. So the complexity is O(N^2). However, it can be further improved since as described above, we only update minCuts when we find a palindrome substring, while the DP algorithm spends lots of time to calculate isPalin, most of which is false (i.e. not a palindrome substring). If we can reduce such unnecessary calculation, then we can speed up the algorithm. This can be achieved with a Manancher-like solution, which is also given as following.

```
// DP solution
    class Solution {
    public:
        int minCut(string s) {
            const int N = s.size();
            if(N<=1) return 0;
            int i,j;
            bool isPalin[N][N];
            fill_n(&isPalin[0][0], N*N, false);
            int minCuts[N+1];
            for(i=0; i<=N; ++i) minCuts[i] = i-1;

            for(j=1; j<N; ++j)
            {
                for(i=j; i>=0; --i)
                {
                    if( (s[i] == s[j]) && ( ( j-i < 2 ) || isPalin[i+1][j-1] ) )
                    {
                        isPalin[i][j] = true;
                        minCuts[j+1] = min(minCuts[j+1], 1 + minCuts[i]);
                    }
                }
            }
            return minCuts[N];

        }
    };
```

The Manancher-like solution scan the array from left to right (for i loop) and only check those sub-strings centered at s[i]; once a non-palindrome string is found, it will stop and move to i+1. Same as the DP solution, minCUTS[i] is used to save the minimum cuts for s[0:i-1]. For each i, we do two for loops (for j loop) to check if the substrings s[i-j .. i+j] (odd-length substring) and s[i-j-1.. i+j] (even-length substring) are palindrome. By increasing j from 0, we can find all the palindrome sub-strings centered at i and update minCUTS accordingly. Once we meet one non-palindrome sub-string, we stop for-j loop since we know there no further palindrome substring centered at i. This helps us avoid unnecessary palindrome substring checks, as we did in the DP algorithm. Therefore, this version is faster.

```
//Manancher-like solution
class Solution {
public:
    int minCut(string s) {
        const int N = s.size();
        if(N<=1) return 0;

        int i, j, minCUTS[N+1];
        for(i=0; i<=N; ++i) minCUTS[i] = i-1;

        for(i=1;i<N;i++)
        {
            for(j=0;(i-j)>=0 && (i+j)<N && s[i-j]== s[i+j]; ++j) // odd-length su
bstrings
                minCUTS[i+j+1] = min(minCUTS[i+j+1], 1 + minCUTS[i-j]);

            for(j=0;(i-j-1)>=0 && (i+j)<N && s[i-j-1]== s[i+j]; ++j) // even-leng
th substrings
                minCUTS[i+j+1] = min(minCUTS[i+j+1], 1 + minCUTS[i-j-1]);
        }
        return minCUTS[N];
    }
};
```

written by dong.wang.1694 original link here