Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example,
S = `"ADOBECODEBANC"`
T = `"ABC"`

Minimum window is `"BANC"`.

Note:
If there is no such window in S that covers all characters in T, return the empty string `""`.

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

Solution 1

I will first give the solution then show you the magic template.

The code of solving this problem is below. It might be the shortest among all solutions provided in Discuss.

```
string minWindow(string s, string t) {
        vector<int> map(128,0);
        for(auto c: t) map[c]++;
        int counter=t.size(), begin=0, end=0, d=INT_MAX, head=0;
        while(end<s.size()){
            if(map[s[end++]]-->0) counter--; //in t
            while(counter==0){ //valid
                if(end-begin<d)  d=end-(head=begin);
                if(map[s[begin++]]++==0) counter++;  //make it invalid
            }
        }
        return d==INT_MAX? "":s.substr(head, d);
    }
```

Here comes the template.

For most substring problem, we are given a string and need to find a substring of it which satisfy some restrictions. A general way is to use a hashmap assisted with two pointers. The template is given below.

```
int findSubstring(string s){
        vector<int> map(128,0);
        int counter; // check whether the substring is valid
        int begin=0, end=0; //two pointers, one point to tail and one  head
        int d; //the length of substring

        for() { /* initialize the hash map here */ }

        while(end<s.size()){

            if(map[s[end++]]-- ?){  /* modify counter here */ }

            while(/* counter condition */){

                 /* update d here if finding minimum*/

                //increase begin to make it invalid/valid again

                if(map[s[begin++]]++ ?){ /*modify counter here*/ }
            }

            /* update d here if finding maximum*/
        }
        return d;
  }
```

One thing needs to be mentioned is that when asked to find maximum substring,

we should update maximum after the inner while loop to guarantee that the substring is valid. On the other hand, when asked to find minimum substring, we should update minimum inside the inner while loop.

The code of solving **Longest Substring with At Most Two Distinct Characters** is below:

```
int lengthOfLongestSubstringTwoDistinct(string s) {
        vector<int> map(128, 0);
        int counter=0, begin=0, end=0, d=0;
        while(end<s.size()){
            if(map[s[end++]]++==0) counter++;
            while(counter>2) if(map[s[begin++]]--==1) counter--;
            d=max(d, end-begin);
        }
        return d;
    }
```

The code of solving **Longest Substring Without Repeating Characters** is below:

Update 01.04.2016, thanks @weiyi3 for advise.

```
int lengthOfLongestSubstring(string s) {
        vector<int> map(128,0);
        int counter=0, begin=0, end=0, d=0;
        while(end<s.size()){
            if(map[s[end++]]++>0) counter++;
            while(counter>0) if(map[s[begin++]]-->1) counter--;
            d=max(d, end-begin); //while valid, update d
        }
        return d;
    }
```

I think this post deserves some upvotes! :)

written by zjh08177 original link here

Solution 2

```
class Solution {
public:
    string minWindow(string S, string T) {
        if (S.empty() || T.empty())
        {
            return "";
        }
        int count = T.size();
        int require[128] = {0};
        bool chSet[128] = {false};
        for (int i = 0; i < count; ++i)
        {
            require[T[i]]++;
            chSet[T[i]] = true;
        }
        int i = -1;
        int j = 0;
        int minLen = INT_MAX;
        int minIdx = 0;
        while (i < (int)S.size() && j < (int)S.size())
        {
            if (count)
            {
                i++;
                require[S[i]]--;
                if (chSet[S[i]] && require[S[i]] >= 0)
                {
                    count--;
                }
            }
            else
            {
                if (minLen > i - j + 1)
                {
                    minLen = i - j + 1;
                    minIdx = j;
                }
                require[S[j]]++;
                if (chSet[S[j]] && require[S[j]] > 0)
                {
                    count++;
                }
                j++;
            }
        }
        if (minLen == INT_MAX)
        {
            return "";
        }
        return S.substr(minIdx, minLen);
    }
};
```

Implementation of mike3's idea

running time : 56ms.

written by heleifz original link here

Solution 3

```
string minWindow(string S, string T) {
    string result;
    if(S.empty() || T.empty()){
        return result;
    }
    unordered_map<char, int> map;
    unordered_map<char, int> window;
    for(int i = 0; i < T.length(); i++){
        map[T[i]]++;
    }
    int minLength = INT_MAX;
    int letterCounter = 0;
    for(int slow = 0, fast = 0; fast < S.length(); fast++){
        char c = S[fast];
        if(map.find(c) != map.end()){
            window[c]++;
            if(window[c] <= map[c]){
                letterCounter++;
            }
        }
        if(letterCounter >= T.length()){
            while(map.find(S[slow]) == map.end() || window[S[slow]] > map[S[slow]
]){
                window[S[slow]]--;
                slow++;
            }
            if(fast - slow + 1 < minLength){
                minLength = fast - slow + 1;
                result = S.substr(slow, minLength);
            }
        }
    }
    return result;
}
```

There are three key variables in my solution:

```
unordered_map <char, int> map; unordered_map<char, int> window; int letterCounter
;
```

variable "map" is used to indicate what characters and how many characters are in T.

variable "window" is to indicate what characters and how many characters are between pointer "slow" and pointer "fast".

Now let's start.

The first For loop is used to construct variable "map".

The second For loop is used to find the minimum window.

The first thing we should do in the second For loop is to find a window which can

cover T. I use "letterCounter" to be a monitor. If "letterCounter" is equal to T.length(), then we find this window. Before that, only the first If clause can be executed. However, after we find this window, the second If clause can also be executed.

In the second If clause, we move "slow" forward in order to shrink the window size. Every time finding a smaller window, I update the result.

At the end of program, I return result, which is the minimum window.

written by zxyperfect original link here