Algorithm Design and Analysis

Homework 4

Haochen Huang
5140309485

November 28, 2016

1

1.1 Question

Given a non-empty integer array, find the minimum number of moves required to make all array elements equal, where a move is incrementing a selected element by 1 or decrementing a selected element by 1.

You may assume the array's length is at most 10,000.
Example:
Input: [1,2,3]
Output: 2
Explanation:
Only two moves are needed (remember each move increments or decrements one element):
$[1, 2, 3] => [2, 2, 3] => [2, 2, 2]$
Input:
int A[]: the input array.
int N: length of A.
Output:
int minMoves.

1.2 Answer

For this problem, it is easy to find that we only need to get the medium number of the array and calculate the sum of the absolute difference between every item and the medium.

The question is to find the medium number of the array. One trivial solution is to sort the array and get the number with index n/2. The time complexity is O(nlogn). Another solution is to get the medium number similar to quick sort. In this solution, the mean time complexity can be O(n).

Use the same method and the function partition of quick sort. The difference is that we use an index to check whether the n/2 index has already set. And stop here. **Time complexity O(n)**

1.3 Code

C++ :

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
using namespace std;

int partition(int *array, int left, int right)
{
        if (array == NULL)
                return -1;

        int pos = right;
        right--;
        while (left <= right)
        {
                while (left < pos && array[left] <= array[pos])
                        left++;

                while (right >= 0 && array[right] > array[pos])
                        right--;

                if (left >= right)
                        break;

                swap(array[left], array[right]);
        }
        swap(array[left], array[pos]);

        return left;
}

int getMidIndex(int *array, int size)
{
        if (array == NULL || size <= 0)
        return -1;

        int left = 0;
        int right = size - 1;
        int midPos = right >> 1;
        int index = -1;

        while (index != midPos)
        {
                index = partition(array, left, right);

                if (index < midPos)
                        left = index + 1;
                else if (index > midPos)
                        right = index - 1;
```

```
49                  else break;
50          }
51
52          //assert(index == midPos);
53          return array[index];
54  }
55
56  int minmov(int *A, int n){
57          int sum = 0;
58          int mean = getMidIndex(A, n);
59          int ans = 0;
60          for (int i=0; i<n; ++i){
61                  ans += abs(A[i]-mean);
62          }
63          return ans;
64  }
65
66  int main(){
67          int A[3] = {1,2,3};
68          int n=3;
69          cout<<minmov(A, n);
70  }
```

2

2.1 Question

Given a string that consists of only uppercase English letters, you can replace any letter in the string with another letter at most k times. Find the length of a longest substring containing all repeating letters you can get after performing the above operations.

Note:Both the string's length and k will not exceed 10^4.

Example:

Input:

s = "AABABBA", k = 1

Output:

4

Explanation:

Replace the one 'A' in the middle with 'B' and form "AABBBBA". The substring "BBBB" has the longest repeating letters, which is 4.

Input:

string s;

int k;

Output:

return the length of the longest substring.

2.2 Answer

We use slide window to solve this problem. The problem says that we can make at most k changes to the string. To convert this problem, for any substring, we can calculate a number (len(substring) - number of maximum character in this substring). As long as this answer is $\leq k$, this is a good string. And we want to maximize the length of this substring by extending the end length.

Given this, we can maintain a sliding window. First, extend the window to its limit, that is, the longest we can get to with maximum number of modifications.

Then as end increase, the whole substring disobey the k rule, so we need to move start to the right until the whole string satisfy it again. Every time we need to get the maximum length of the substring.

Time complexity O(n^2)

2.3 Code

C++ :

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <string.h>
using namespace std;

```

```
int maxchar(int character[]){
        int num = 0;
        for(int i=0;i<26;++i){
                num = max(num, character[i]);
        }
        return num;
}

int characterReplacement(string s, int k) {
        int len = s.length();
        int start = 0, end = 0;
        int character[26] = {0};
        int res = 0;

        while(end < len){
                int tmp = s[end] - 'A';
                ++character[tmp];
                ++end;
                while(end-start-maxchar(character) > k){
                        int tmp = s[start] - 'A';
                        --character[tmp];
                        ++start;
                }
                res = max(res, end-start);
        }
        return res;
}

int main(){
        string s = "AABABBA";
        int k=1;
        cout<<characterReplacement(s, k);
}
```

3

3.1 Question

You are given an array x of n positive numbers. You start at point (0,0) and moves x[0] metres to the north, then x[1] metres to the west, x[2] metres to the south, x[3] metres to the east and so on. In other words, after each move your direction changes counter-clockwise.

Write a one-pass algorithm with O(1) extra space to determine, if your path crosses itself, or not.

Example 1:
Given x = [2, 1, 1, 2]
Return true (self crossing)
Example 2:
Given x = [1, 2, 3, 4]
Return false (not self crossing)
Example 3:
Given x = [1, 1, 1, 1]
Return true (self crossing)
Input:
int x[]: the input array.
int N: length of x.
Output:
return true or false.

3.2 Answer

At first glance, this problem may be difficult and the code can be complex. However, We can consider when will there be a self cross.

There are in all three situations. The direction changes counter-clockwise. So when the direction changes for 3 4 5 times, there can be a self cross. When there are more changes of direction, they are the same to the previous ones for 360/90=4.

So consider these three situations. Take 3 as an example. We need to satisfy that $x[i] \geq x[i-2] \&\& x[i-1] \leq x[i-3]$, so that there is a cross. So do the other two situations.

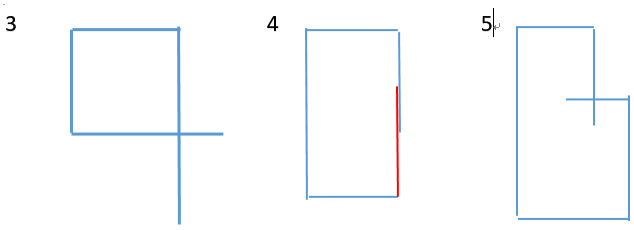

Figure 1: Three situation for self crossing

Time complexity O(n)

3.3 Code

C++ :

```cpp
#include <iostream>
#include <math.h>
using namespace std;

bool SelfCrossing(int *x, int N) {
    for(int i=3; i<N; ++i){
        if(i>=3 && x[i]>=x[i-2] && x[i-1]<=x[i-3])
            return true;
        if(i>=4 && x[i-1]==x[i-3] && x[i-2]<=x[i]+x[i-4])
            return true;
        if(i>=5 && x[i-2]>x[i-4] && x[i-1]<=x[i-3]
                && x[i-1]+x[i-5]>=x[i-3] && x[i]+x[i-4]>=x[i-2])
            return true;
    }
    return false;
}

int main(){
        int x[4] = {2, 1, 1, 2};
        int N = 4;
        cout<<SelfCrossing(x,N);
}
```