Find the Running Median

locked

by amititkgp

Editorial by Tanzir5

This problem can be solved using two **heaps**.

Let's say we are taking input for the i^{th} number. If somehow we had the previous $(i-1)$ numbers sorted, then we can easily add the i^{th} number in $O(n)$ time to the appropriate place in our list so that the list remains sorted and find the median in $O(1)$ time. But we cannot afford to add every number with $O(n)$ complexity.

So let's say we had two sorted arrays. The first array holds the smaller half of the numbers in decreasing order. The second array contains the larger half of the numbers in increasing order. Now, after taking input for the i^{th} number, we can easily decide which half the i^{th} number belongs to and add it there in the appropriate place. If any of the arrays becomes much larger than the other array, we can remove the first element from that array and add it to the appropriate place of the other array. We have reduced time complexity by some constant factor. But obviously, that is not enough.

Now let's look at the indices of the array where we need to access at any moment for getting the median. If the total number of elements is odd, then we need the first element of the array with the higher number of elements. If the total number of elements is even, then we need the average of the first elements of both the arrays. So the only indices we need to access while getting the median and adding the elements are the first index of both of the arrays. So which data structure can help in storing data in sorted order and accessing the top element efficiently? The answer is a **heap**.

We will use a **max heap** for storing data of the smaller half of the numbers and a **min heap** for storing data of the larger half of the numbers. Now let's see how we will add the numbers and get the medians.

Adding the i^{th} number:

While adding the i^{th} number we will check the following conditions and add accordingly:

1. If the i^{th} number is greater than or equal to the max element of the max heap then it surely belongs to the larger half of the numbers i.e. the min heap. So we will add it there.
2. Otherwise, we will add it to the max heap.

Now it may happen that one of the heap becomes much larger than the other if we add the numbers in this way. So to stop this situation from taking place, we need to check the size of the heaps after adding every number. If the difference between the number of elements of the two heaps becomes more than one, then we need to pop the top element from the heap with more elements and push that element to the other heap. If we work in this way, the difference can never be more than one.

Getting the median:

To get the median after adding the i^{th} number we will check if i is odd or even. If i is odd, then surely one of the heap has one more element than the other. The median will be the

Statistics

Difficulty:	Hard
Time Complexity:	O(nlogn) Required
Knowledge:	Heaps, Priority Queue
Publish Date:	May 07 2015

top element of that heap then. If i is even, then the two heaps must have the same number of elements. So the median will be the average of the top elements of the two heaps.

Editorialist's solution:

```cpp
#include<bits/stdc++.h>
using namespace std;

priority_queue<int, vector<int>, greater <int> > min_heap;
priority_queue<int> max_heap;
void add(int a)
{
    if( max_heap.size() && a >= max_heap.top())
        min_heap.push(a);
    else
        max_heap.push(a);

    if(abs(max_heap.size() - min_heap.size()) > 1)
    {
        if(max_heap.size() > min_heap.size())
        {
            int temp = max_heap.top();
            max_heap.pop();
            min_heap.push(temp);
        }
        else
        {
            int temp = min_heap.top();
            min_heap.pop();
            max_heap.push(temp);
        }
    }
}

double get_median()
{
    int total = min_heap.size() + max_heap.size();
    double ret;
    if(total%2 == 1)
    {
        if(max_heap.size() > min_heap.size())
            ret = max_heap.top();
        else
            ret = min_heap.top();
    }
    else
    {
        ret = 0;
        if(max_heap.empty() == false)
            ret += max_heap.top();
        if(min_heap.empty() == false)
            ret += min_heap.top();
        ret/=2;
    }
    return ret;
}

int main()
{
    cout << setprecision(1) << fixed;
    int n, a;
    cin >> n;
    for(int i = 1; i<=n; i++)
    {
        cin >> a;
        add(a);
        cout << get_median() << endl;
    }
}
```

Contest Calendar | Interview Prep | Blog | Scoring | Environment | FAQ | About Us | Support | Careers | Terms Of Service | Privacy Policy | Request a Feature