双堆问题

笔记本:	0_leetcode		
创建时间:	2020/8/25 12:41	**更新时间:**	2020/8/26 20:42
作者:	415669592@qq.com		
URL:	https://www.educative.io/courses/grokking-the-coding-interview/3Yj2BmpyEy4		

双堆问题

0. 介绍

双堆问题主要用来将一个list划分成为俩个堆。

前提知识
log1 + log2 + ... + logN = NlogN

1. 实战

1.1 计算中值

> 中值的计算。

思路

中位数计算是很明显的双堆问题。将一部分划分为比中值大的，另一部分划分为比中值小的。这里需要对堆有清晰的理解。

给定一个数组。依次将所有数插入到俩个堆中。小根堆存放大于中值的数。大根堆存放小于中值的数字。

[3 9 6 12 1]

这里的核心思路就是：

1. 当前我们以最大堆为主。即如果是奇数，让最大堆比最小堆多一个数字，那么就返回最大堆。否则返回俩个堆顶的均值。
2. 因此，第一个数一定插入到大根堆中。且比大根堆堆顶小的元素也插入到最大堆中。剩余插入到最小堆中。
3. 最后设置一个调整，保证最大堆size == 最小堆size + 1 或者最大堆size == 最小堆size

```
using namespace std;

#include <iostream>
#include <queue>
#include <vector>
#include <functional>

class MedianOfAStream {
  public:
    priority_queue<int> max_heap;
    priority_queue<int, vector<int>, greater<int>> min_heap;
    double getMedian( const vector<int> &ivec ) {
        // 大根堆堆顶是最大值
        for ( int i = 0; i < ivec.size(); i++ ) {
            if ( max_heap.empty() || ivec[i] < max_heap.top() )
                max_heap.push( ivec[i] );
            else
                min_heap.push( ivec[i] );

            // 调整
            if ( max_heap.size() > min_heap.size() + 1 ) {
                auto x = max_heap.top();
                max_heap.pop();
                min_heap.push( x );
```

```
            } else if ( min_heap.size() > max_heap.size() ) {
                auto x = min_heap.top();
                min_heap.pop();
                max_heap.push( x );
            }
        }
        if ( ivec.size() % 2 == 0 )
            return ( min_heap.top() + max_heap.top() ) / 2.0;
        else
            return max_heap.top();
    }
};

int main( int argc, char *argv[] ) {
    vector<int> arr = { 1, 2, 3, 4, 5, 6 };
    cout << MedianOfAStream{} .getMedian( arr ) << endl;

    system( "pause" );
}
```

分析:

1. 上面的时间复杂度是NlogN。简单将俩个堆看成一个堆。每次操作是log i。根据介绍知识我们知道这里的复杂度最差是NlogN

下面的场景更加适合使用双堆法。

```
using namespace std;

#include <iostream>
#include <queue>
#include <vector>

class MedianOfAStream {
  public:
    priority_queue<int> maxHeap;                              // containing
first half of numbers
    priority_queue<int, vector<int>, greater<int>> minHeap;  // containing
second half of numbers

    virtual void insertNum( int num ) {
        if ( maxHeap.empty() || maxHeap.top() >= num ) {
            maxHeap.push( num );
        } else {
```

```
            minHeap.push( num );
        }

        // either both the heaps will have equal number of elements or max-
heap will have one
        // more element than the min-heap
        if ( maxHeap.size() > minHeap.size() + 1 ) {
            minHeap.push( maxHeap.top() );
            maxHeap.pop();
        } else if ( maxHeap.size() < minHeap.size() ) {
            maxHeap.push( minHeap.top() );
            minHeap.pop();
        }
    }

    virtual double findMedian() {
        if ( maxHeap.size() == minHeap.size() ) {
            // we have even number of elements, take the average of middle
two elements
            return maxHeap.top() / 2.0 + minHeap.top() / 2.0;
        }
        // because max-heap will have one more element than the min-heap
        return maxHeap.top();
    }
};

int main( int argc, char *argv[] ) {
    MedianOfAStream medianOfAStream;
    medianOfAStream.insertNum( 3 );
    medianOfAStream.insertNum( 1 );
    cout << "The median is: " << medianOfAStream.findMedian() << endl;
    medianOfAStream.insertNum( 5 );
    cout << "The median is: " << medianOfAStream.findMedian() << endl;
    medianOfAStream.insertNum( 4 );
    cout << "The median is: " << medianOfAStream.findMedian() << endl;
}
```

1.2 求滑动窗口的中位数

哈哈哈，梦幻联动。滑动窗口和双堆联动。

> 给定一个数组。给定一个滑动窗口大小。计算窗口中的中为数。

思路

1. 很明显，这道题可以直接使用滑动窗口的模式和双堆。
2. 右边界扩张时候，很简单。只需要将该数字插入到双堆中就可以了。
3. 但是在左边界收缩的时候。如何将数字从双堆中拿出来呢。

实际上只要给堆提供一个remove方法就好了。
如何快速写一个堆呢？如果看过C++标准库的人，就知道STL是提供了一套针对Heap的操作的。然后参考cppreference我们可以快速写出一个带有remove方法的堆。

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
#include <functional>
#include <list>

class MyPriorityQueue {
  public:
    void set_compare( function<bool( int, int )> x ) {
        comp = x;
    }
    void push( int elem ) {
        data.push_back( elem );
        std::push_heap( data.begin(), data.end(), comp );
    }
    int size()const {
        return data.size();
    }
    int top()const {
        return data[0];
    }
    void pop() {
        std::pop_heap( data.begin(), data.end(), comp );
        data.resize( data.size() - 1 );
    }
    void remove( int x ) {
        auto iter = std::find( data.begin(), data.end(), x );
        data.erase( iter );
        std::make_heap( data.begin(), data.end(), comp );
    }
  private:
```

```
    function<bool( int, int )> comp;
    vector<int> data;
};
class SlidingWindowMedian {
  public:
    MyPriorityQueue max_heap;
    MyPriorityQueue min_heap;
    virtual vector<double> findSlidingWindowMedian( const vector<int> &nums,
int k ) {
        vector<double> result;
        max_heap.set_compare( []( int x1, int x2 ) {
            return x1 < x2;
        } );
        min_heap.set_compare( []( int x1, int x2 ) {
            return x1 > x2;
        } );
        // TODO: Write your code here
        int win_start = 0;
        for ( int win_end = 0; win_end < nums.size(); win_end++ ) {
            if ( max_heap.size() == 0 || nums[win_end] < max_heap.top() ) {
                max_heap.push( nums[win_end] );
            } else {
                min_heap.push( nums[win_end] );
            }
            // 重新进行堆的调整
            if ( max_heap.size() > min_heap.size() + 1 ) {
                auto x = max_heap.top();
                max_heap.pop();
                min_heap.push( x );
            }
            if ( min_heap.size() > max_heap.size() ) {
                auto x = min_heap.top();
                min_heap.pop();
                max_heap.push( x );
            }

            if ( win_end + 1 - win_start == k ) {
                if ( k % 2 == 0 )
                    result.push_back( ( max_heap.top() + min_heap.top() ) /
2.0 );
                else
                    result.push_back( max_heap.top() );

                int left_value = nums[win_start];
```

```
                if ( left_value <= max_heap.top() ) {
                    max_heap.remove( left_value );
                } else {
                    min_heap.remove( left_value );
                }
                win_start++;
            }
            // 重新进行堆的调整
            if ( max_heap.size() > min_heap.size() + 1 ) {
                auto x = max_heap.top();
                max_heap.pop();
                min_heap.push( x );
            }
            if ( min_heap.size() > max_heap.size() ) {
                auto x = min_heap.top();
                min_heap.pop();
                max_heap.push( x );
            }

        }
        return result;
    }
};

int main( int argc, char *argv[] ) {
    SlidingWindowMedian slidingWindowMedian;
    vector<double> result =
        slidingWindowMedian.findSlidingWindowMedian( vector<int> {1, 2, -1,
3, 5}, 2 );
    cout << "Sliding window medians are: ";
    for ( auto num : result ) {
        cout << num << " ";
    }
    cout << endl;

    slidingWindowMedian = SlidingWindowMedian();
    result = slidingWindowMedian.findSlidingWindowMedian( vector<int> {1, 2,
-1, 3, 5}, 3 );
    cout << "Sliding window medians are: ";
    for ( auto num : result ) {
        cout << num << " ";
    }
    system( "pause" );
```

```
}
```

1.3 贪心问题

> 给定项目的初始资金，给定项目的收益。给定你一个初始资金，在限制投资次数的情况下，问如何获得最大收益。

e.g.1

```
Input:  Project Capitals=[0,1,2],
Project Profits=[1,2,3],
Initial Capital=1, Number of Projects=2
Output:  6
解释：
1. 投资第二个项目。得到收益为2。此时我们的资本变为3。（加上初始资本）
2. 投资第三个醒目。得到收益为3。最大收益为6。
```

e.g.2

```
Input:  Project Capitals=[0,1,2,3],
Project Profits=[1,2,3,5], Initial Capital=0, Number of Projects=3
Output:  8
1. 投资第一个项目，总资本1
2. 投资第二个项目，总资本3
3. 投资第四各项目，总资本8
```

思路

典型的贪心问题：
每次投资，我们要在能投资的所有项目中选择收益最大的。

按照人脑思路是这样的：

1. 找到可以投资的项目中收益最大的。
2. 累积资本
3. 重复1

编程思路

1. 将所有项目放到最小堆中。
2. 将可以投的项目取出，其收益放到最大堆中。
3. 每次取出最大收益，并更新资本。

使用双堆的复杂度：

1. 将所有元素插入到最小堆中。即NlogN
2. 假设所有项目都能被插入到最大堆中。即NlogN
3. 弹出K次，KlogN。

复杂度是O(2*NlogN+KlogN)

```
using namespace std;

#include <iostream>
#include <queue>
#include <vector>
#include <functional>

class CompareGreater {
  public:
    bool operator()( std::pair<int, int> &x1, std::pair<int, int> &x2 ) {
        return x1.first > x2.first;
    }
};
class MaximizeCapital {
  public:
    static int findMaximumCapital( const vector<int> &capital, const
vector<int> &profits,
                                   int numberOfProjects, int initialCapital )
{
        priority_queue<int> max_heap;
        priority_queue<std::pair<int, int>, vector<std::pair<int, int>>,
CompareGreater> min_heap;
        // TODO: Write your code here
        // 1. 将capital加入到最小堆中
        for ( int i = 0; i < capital.size(); i++ ) {
            min_heap.push( std::make_pair( capital[i], i ) );
        }
        while ( numberOfProjects-- ) {
            while ( !min_heap.empty() && min_heap.top().first <=
initialCapital ) {
                auto x = min_heap.top();
                min_heap.pop();
```

```
                max_heap.push( profits[x.second] );
            }
            initialCapital += max_heap.top();
            max_heap.pop();
        }
        return initialCapital;
    }
};

int main( int argc, char *argv[] ) {
    int result =
        MaximizeCapital::findMaximumCapital( vector<int> {0, 1, 2},
vector<int> {1, 2, 3}, 2, 1 );
    cout << "Maximum capital: " << result << endl;
    result =
        MaximizeCapital::findMaximumCapital( vector<int> {0, 1, 2, 3},
vector<int> {1, 2, 3, 5}, 3, 0 );
    cout << "Maximum capital: " << result << endl;
    system( "pause" );
}
```

为什么不用dfs

因为没得选择——这是什么意思呢?

使用dfs，一般是我们存在同样的抉择。我们在不知道后续的情况下，无法判断当前哪个抉择更好时，我们使用dfs。即经典背包问题，对于一件物品，拿还是不拿这就是俩种抉择。

而这里的思想是贪心。就是我们肯定选择收益最大的。而且当前选择对后续选择时没有影响的。即当前的选择已经是全局最优了。

起始简单改一下这道题，就可以变成dfs问题。即只有一次投资，但是可以投多个产品。这道题是投资一个，然后回收收益后，再去投下一个。而动态规划则是，只有一次投资，求这次投资的收益最大。
虽然每个投资都有性价比，但是有的投资我们买不起。比如
初始资本为50;
有如下三个投资。

[{40, 80}, {15, 20},{35, 69}] 分别代表{投资，收益}
按照性价比来说，第一个性价比最高。应该先把第一个买完。但是2+3的组合实际收益却更好。

1.4 下一个区间

> 给定一个区间数组。寻找改区间的下一个区间。对于区间i来说的下一个区间j。i和j不重合。且start最小。

e.g.1

```
Input:  Intervals [[2,3], [3,4], [5,6]]
Output:  [1, 2, -1]
Explanation:  The next interval of [2,3] is [3,4] having index ‘1’.
Similarly, the next interval of [3,4] is [5,6] having index ‘2’. There is
no next interval for [5,6] hence we have ‘-1’.
```

e.g.2

```
Input:  Intervals [[3,4], [1,5], [4,6]]
Output:  [2, -1, -1]
Explanation:  The next interval of [3,4] is [4,6] which has index ‘2’.
There is no next interval for [1,5] and [4,6].
```

思路

按照题意描述，我们寻找的是一个start最小，且不重合的区间。直接可以得到判断条件。

[8 9] [3 4] [1 2] [5 6]
-1 3 1 0

对于8,9来说，我们要寻找其下一个区间。

1. 肉眼一看就知道没有。为什么呢？因为不存在比9更大的start。

对于3 4来说。我们则要从比其大的start区间中，寻找最小的。

目前可以得到一个初步思路。
对于区间A来说哦，遍历数组，将所有start大于它的区间放到一个最小堆中。

显然这个操作是比较费时的。原因是我们当前的数组是乱序的。对于区间[3, 4]来说，最小堆有[8, 9][5, 6]
对于区间[1, 2]来说，最小堆有[3, 4], [5,6], [8,9]，可见其中有很多重复。

我们再次建模问题。将这个问题变为
对于第一个区间，我们的目的是找比9大的最小数字。
对于第二个区间，我们的目的是找比4大的最小数字。
对于第三个区间，我们的目的是找比2大的最小数字。

。。。。
我们以end_time建立小根堆。再以start_time建立小根堆。
从end_time中选取最小的end_time。再从start_time中选取比end_time大的数字。被

pop的都是比end_time1小的。
显然，比2小的数字必定比4小。所以弹出是合理的。

其实可以将这道题换一下。假设给定的不是区间。而是数。寻找比每个数大的最小的数。
我们显然知道是要将其排序的。
那么在增加一点。给定的是俩个数组呢？寻找比A数组中大的最小的B数组中的数。
显然也是排序。俩个数组都排好序之后。用俩个游标去做。

那么这道题是不是可以等价于寻找比每个endtime大的最小的start_time呢？显然也是俩个数组。只不过排序需要我们自己维护一个游标来控制头。而使用堆的话则是数据结构本身保证这个问题。只能说二者都可以。但这里是双堆主题。

动手！

```
using namespace std;

#include <iostream>
#include <queue>
#include <vector>

class Interval {
  public:
    int start = 0;
    int end = 0;
    int index = 0;

    Interval( int start, int end ) {
        this->start = start;
        this->end = end;
    }
};

struct CompareEndLess {
    bool operator()( const Interval &x1, const Interval &x2 ) {
        return x1.end > x2.end;
    }
};
struct CompareStartLess {
    bool operator()( const Interval &x1, const Interval &x2 ) {
        return x1.start > x2.start;
    }
};
class NextInterval {
  public:
    static vector<int> findNextInterval( vector<Interval> &intervals ) {
        int n = intervals.size();
```

```
        vector<int> result( n );
        priority_queue<Interval, vector<Interval>, CompareEndLess>
min_heap_end;
        priority_queue<Interval, vector<Interval>, CompareStartLess>
min_heap_start;
        for ( int i = 0; i < intervals.size(); i++ ) {
            intervals[i].index = i;
            min_heap_end.push( intervals[i] );
            min_heap_start.push( intervals[i] );
        }

        // TODO: Write your code here
        while ( !min_heap_end.empty() ) {
            // 取出
            auto x = min_heap_end.top();
            min_heap_end.pop();
            while ( !min_heap_start.empty() && min_heap_start.top().start <
x.end ) {
                min_heap_start.pop();
            }
            if ( min_heap_start.empty() ) {
                result[x.index] = -1;
            } else {
                result[x.index] =  min_heap_start.top().index ;
            }
        }
        return result;
    }
};

int main( int argc, char *argv[] ) {
    vector<Interval> intervals = {{2, 3}, {3, 4}, {5, 6}};
    vector<int> result = NextInterval::findNextInterval( intervals );
    cout << "Next interval indices are: ";
    for ( auto index : result ) {
        cout << index << " ";
    }
    cout << endl;

    intervals = {{3, 4}, {1, 5}, {4, 6}};
    result = NextInterval::findNextInterval( intervals );
    cout << "Next interval indices are: ";
    for ( auto index : result ) {
        cout << index << " ";
```

```
    }
    cout << endl;
    intervals = { {8,9}, {3, 4}, {1, 2}, {5, 6} };
    result = NextInterval::findNextInterval( intervals );
    cout << "Next interval indices are: ";
    for ( auto index : result ) {
        cout << index << " ";
    }
    system( "pause" );
}
```

2. 总结

怎么说呢？这个感觉还不是很明显。可能是题量不够的原因。
正如第一道题。堆给我的感觉是可以动态维护数组中的最值。当然排序+游标的数据结构也可以实现。就是麻烦点。但堆的思想还是涉及在那些只和最值相关的问题上。重点是建好堆之后，只需要O(logN)的时间就可以寻找最值。这应该是最快的了。

1. 熟练掌握优先级队列的代码编写。
2. 熟练掌握对自定义类的优先级队列使用。
3. 明白怎么写是最大堆，怎么写是最小堆。
4. 了解堆背后的时间复杂度，以及make_heap，push_heap,pop_heap的STL使用。这个基本上要做到只要看cppreference就能明白其意思的地步。