Top K问题

笔记本：	0_leetcode		
创建时间：	2020/8/29 12:57	**更新时间：**	2020/9/4 23:09
作者：	415669592@qq.com		
URL：	https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB		

Top K问题

- Top K问题
  - 0. 介绍
    - make_heap
    - pop_heap
    - push_heap
  - 1. 入门
    - 1.1 前K大元素
      - 思路
    - 1.2 寻找第K小元素
      - 思路
  - 2. 实战
    - 2.1 寻找前K近的到原点的点
      - 思路
    - 2.2 连接绳子
      - 思路
    - 2.3 统计top K词频的数字
      - 思路
    - 2.4 给定一个字符串，按照词频排序
      - 思路
    - 2-5 计算流中第K大的数字
      - 思路
    - 2-6 寻找距离X最近的K个值
      - 思路
    - 2.7 删除K个数字
      - 思路
    - 2.8 计算区间和
      - 思路
    - 2.9 重排字符串
      - 思路
  - 3. 挑战

- 3.1 重排字符
    - 思路
- 3.2 安排任务
    - 思路

0. 介绍

关键字：寻找前K个。常见手段是使用堆。

简单谈一下STL中的heap操作。

make_heap

根据数组进行建堆。传入数组的begin和end，以及compare对象。

pop_heap

这个操作会将元素从vector头部移动到尾部。
同时要明白，此时发生了size减小。这都是需要程序员记录的东西。

push_heap

在数组尾部添加元素后，重新调整堆到堆状态。

参考建议：cppreference

1. 入门

1.1 前K大元素

> 给定一个无序数组，寻找前K大的元素。

e.g.1

```
Input: [3, 1, 5, 12, 2, 11], K = 3
Output: [5, 12, 11]
```

e.g.2

```
Input: [5, 12, 11, -1, 12], K = 3
Output: [12, 11, 12]
```

思路

最小堆过滤一遍即可。
即遍历数组，如果值大于小根堆堆顶，则插入堆中，同时弹出堆顶。那么就可以维持K大小的一个堆。

面试时候，手撕做法。

```
using namespace std;

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

class KLargestNumbers {
public:
    static vector<int> findKLargestNumbers(const vector<int>& nums, int k) {
        vector<int> result;
        priority_queue<int, vector<int>, greater<int>> min_heap;
        // TODO: Write your code here
        for(int i = 0; i < nums.size(); i++)
        {
            if(min_heap.size() < k)
            {
                min_heap.push(nums[i]);
            }else{
                if(nums[i] > min_heap.top())
                {
                    min_heap.push(nums[i]);
                    min_heap.pop();
                }
            }
        }
        while(!min_heap.empty())
        {
            result.push_back(min_heap.top());
            min_heap.pop();
        }
        return result;
    }
};

int main(int argc, char* argv[]) {
    vector<int> result = KLargestNumbers::findKLargestNumbers(vector<int>{3, 1, 5, 12, 2, 11}, 3);
    cout << "Here are the top K numbers: ";
    for (auto num : result) {
        cout << num << " ";
    }
    cout << endl;

    result = KLargestNumbers::findKLargestNumbers(vector<int>{5, 12, 11, -1, 12}, 3);
    cout << "Here are the top K numbers: ";
    for (auto num : result) {
        cout << num << " ";
    }
    cout << endl;
}
```

但实际上，上述操作存在许多浪费性能的地方，如果是笔试的话，还是需要采用STL中的堆操作。

```
using namespace std;

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

class KLargestNumbers {
public:
    static vector<int> findKLargestNumbers(const vector<int>& nums, int k) {
        vector<int> result;
        vector<int> min_heap(nums.begin(), nums.begin() + k);
        make_heap(min_heap.begin(), min_heap.end(), greater<int>{});
        // TODO: Write your code here
        for(int i = k; i < nums.size(); i++)
        {
            if(nums[i] > min_heap.front())
            {
                pop_heap(min_heap.begin(), min_heap.end(), greater<int>{});
                min_heap.pop_back();
                min_heap.push_back(nums[i]);
                push_heap(min_heap.begin(), min_heap.end(), greater<int>{});
            }
        }
        return min_heap;
    }
};

int main(int argc, char* argv[]) {
    vector<int> result = KLargestNumbers::findKLargestNumbers(vector<int>{3, 1, 5, 12, 2, 11}, 3);
    cout << "Here are the top K numbers: ";
    for (auto num : result) {
        cout << num << " ";
    }
    cout << endl;

    result = KLargestNumbers::findKLargestNumbers(vector<int>{5, 12, 11, -1, 12}, 3);
    cout << "Here are the top K numbers: ";
    for (auto num : result) {
        cout << num << " ";
    }
    cout << endl;
}
```

1.2 寻找第K小元素

> 给定一个无序数组，寻找第K小的元素。注意这个元素是排序后第K小，并不是第K个不同元素。
> 即[1 2 5 5] 第3个元素是5，第4个也是5。

本质上就是找排名为K的元素。

思路

最大堆 or 最小堆

```
using namespace std;

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

class KthSmallestNumber {
public:
    static int findKthSmallestNumber(const vector<int> &nums, int k) {
        // TODO: Write your code here
        priority_queue<int, vector<int>, less<int>> max_heap(nums.begin(), nums.begin() + k);
        for(int i = k; i < nums.size(); i++)
        {
            if(nums[i] < max_heap.top())
            {
                max_heap.push(nums[i]);
                max_heap.pop();
            }
        }
        return max_heap.top();
    }
};

int main(int argc, char *argv[]) {
    int result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 3);
    cout << "Kth smallest number is: " << result << endl;

    // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
    result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 4);
    cout << "Kth smallest number is: " << result << endl;

    result = KthSmallestNumber::findKthSmallestNumber(vector<int>{5, 12, 11, -1, 12}, 3);
    cout << "Kth smallest number is: " << result << endl;
}
```

简单分析得到，其复杂度时O(N * logK)

也可以使用建堆，然后扔掉前K-1小的元素。复杂度是O(N + KlogN)，如下所示。

```
using namespace std;

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

class KthSmallestNumber {
public:
    static int findKthSmallestNumber(const vector<int> &nums, int k) {
        // TODO: Write your code here
        priority_queue<int, vector<int>, greater<int>> min_heap(nums.begin(), nums.end());
        k--;
        while(k--)
        {
            min_heap.pop();
        }
        return min_heap.top();
    }
};

int main(int argc, char *argv[]) {
    int result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 3);
    cout << "Kth smallest number is: " << result << endl;

    // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'
    result = KthSmallestNumber::findKthSmallestNumber(vector<int>{1, 5, 12, 2, 11, 5}, 4);
    cout << "Kth smallest number is: " << result << endl;

    result = KthSmallestNumber::findKthSmallestNumber(vector<int>{5, 12, 11, -1, 12}, 3);
    cout << "Kth smallest number is: " << result << endl;
}
```

2. 实战

2.1 寻找前K近的到原点的点

> 给定一系列二维坐标点，寻找距离原点排名为前K近的点。

e.g.1

```
Input: points = [[1,2],[1,3]], K = 1
Output: [[1,2]]
Explanation: The Euclidean distance between (1, 2) and the origin is sqrt(5).
The Euclidean distance between (1, 3) and the origin is sqrt(10).
Since sqrt(5) < sqrt(10), therefore (1, 2) is closer to the origin.
```

e.g.2

```
Input: point = [[1, 3], [3, 4], [2, -1]], K = 2
Output: [[1, 3], [2, -1]]
```

思路

好像没什么难点。就是得到排名为前K的元素。只不过需要计算下距离而已。

```
using namespace std;

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

class Point {
public:
    int x = 0;
    int y = 0;

    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }

    int distFromOrigin() const {
        // ignoring sqrt
        return (x * x) + (y * y);
    }
};

struct Compare{
    bool operator()(const Point&p1, const Point &p2)
    {
        p1.distFromOrigin() < p2.distFromOrigin();
    }
};
class KClosestPointsToOrigin {
public:
    static vector<Point> findClosestPoints(const vector<Point>& points, int k) {
        vector<Point> result(points.begin(), points.begin() + k);
        // TODO: Write your code here
        make_heap(result.begin(), result.end(), Compare{});
        for(int i = k; i < points.size(); i++)
        {
            if(points[i].distFromOrigin() < result.front().distFromOrigin())
            {
                pop_heap(result.begin(), result.end(), Compare{});
                result[result.size() - 1] = points[i];
                push_heap(result.begin(), result.end(), Compare{});
            }
        }
        return result;
    }
};

int main(int argc, char* argv[]) {
    vector<Point> maxHeap = KClosestPointsToOrigin::findClosestPoints({{1, 3}, {3, 4}, {2, -1}}, 2);
    cout << "Here are the k points closest the origin: ";
    for (auto p : maxHeap) {
        cout << "[" << p.x << " , " << p.y << "] ";
    }
}
```

2.2 连接绳子

> 给定N个不同长度的绳子，将所有的绳子连接起来。使用最小的代价。绳子的代价和绳子的长度相同。注意代价的计算方式。

e.g.1

```
Input: [1, 3, 11, 5]
Output: 33
Explanation: First connect 1+3(=4), then 4+5(=9), and then 9+11(=20). So the total cost is 33 (4+9+20)
```

e.g.2

```
Input: [3, 4, 5, 6]
Output: 36
Explanation: First connect 3+4(=7), then 5+6(=11), 7+11(=18). Total cost is 36 (7+11+18)
```

e.g.3

```
Input: [1, 3, 11, 5, 2]
Output: 42
Explanation: First connect 1+2(=3), then 3+3(=6), 6+5(=11), 11+11(=22). Total cost is 42 (3+6+11+22)
```

思路

也是比较简单。将数组做成堆。弹出前2个最小的数。然后构成一个大数再插入。结束条件可能有点需要考虑。

```
using namespace std;

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

class ConnectRopes {
public:
    static int minimumCostToConnectRopes(const vector<int> &ropeLengths) {
        int result = 0;
        priority_queue<int, vector<int>, greater<int>> min_heap(ropeLengths.begin(), ropeLengths.end());
        // TODO: Write your code here
        while(min_heap.size() > 1)
        {
            auto first = min_heap.top();
            min_heap.pop();
            auto second = min_heap.top();
            min_heap.pop();
            auto value = first + second;
            result += value;
            min_heap.push(value);
        }
        return result;
    }
};

int main(int argc, char *argv[]) {
    int result = ConnectRopes::minimumCostToConnectRopes(vector<int>{1, 3, 11, 5});
    cout << "Minimum cost to connect ropes: " << result << endl;
    result = ConnectRopes::minimumCostToConnectRopes(vector<int>{3, 4, 5, 6});
    cout << "Minimum cost to connect ropes: " << result << endl;
    result = ConnectRopes::minimumCostToConnectRopes(vector<int>{1, 3, 11, 5, 2});
    cout << "Minimum cost to connect ropes: " << result << endl;
}
```

2.3 统计top K词频的数字

> 给定一个未排序的数组，统计词频排前K的数字。

e.g.1

```
Input: [1, 3, 5, 12, 11, 12, 11], K = 2
Output: [12, 11]
Explanation: Both '11' and '12' apeared twice.
```

e.g.2

```
Input: [5, 12, 11, 3, 11], K = 2
Output: [11, 5] or [11, 12] or [11, 3]
Explanation: Only '11' appeared twice, all other numbers appeared once.
```

思路

1. 需要统计词频。
2. 用堆进行过滤

```
using namespace std;

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

struct Compare{
    bool operator()(pair<int, int> x1, pair<int, int> x2)
    {
        return x1.second > x2.second;
    }
};
class TopKFrequentNumbers {
    struct valueCompare {
        char operator()(const pair<int, int> &x, const pair<int, int> &y) {
            return x.second > y.second;
        }
    };

public:
    static vector<int> findTopKFrequentNumbers(const vector<int> &nums, int
k) {
        vector<int> topNumbers;
        topNumbers.reserve(k);
        unordered_map<int, int> frequency;
        for(int n : nums)
            frequency[n]++;
        priority_queue<pair<int, int>, vector<pair<int, int>>, Compare>
min_heap;
        for(auto elem : frequency)
        {
           if(min_heap.size() >= k)
           {
                if(elem.second > min_heap.top().second)
                {
                    min_heap.push(elem);
                    min_heap.pop();
                }
           }else{
               min_heap.push(elem);
            }
        }
        while(!min_heap.empty())
        {
            topNumbers.push_back(min_heap.top().first);
            min_heap.pop();
        }
        return topNumbers;
    }
};

int main(int argc, char *argv[]) {
    vector<int> result =
            TopKFrequentNumbers::findTopKFrequentNumbers(vector<int>{1, 3, 5,
12, 11, 12, 11}, 2);
    cout << "Here are the K frequent numbers: ";
    for (auto num : result) {
        cout << num << " ";
    }
    cout << endl;

    result = TopKFrequentNumbers::findTopKFrequentNumbers(vector<int>{5, 12,
11, 3, 11}, 2);
    cout << "Here are the K frequent numbers: ";
    for (auto num : result) {
        cout << num << " ";
    }
    cout << endl;
}
```

当然这里如果想优化的更好，需要使用STL中的堆操作。

2.4 给定一个字符串，按照词频排序

> 给定一个字符串，按照词频降序排序。

e.g.1

```
Input: "Programming"
Output: "rrggmmPiano"
Explanation: 'r', 'g', and 'm' appeared twice, so they need to appear before
any other character.
```

e.g.2

```
Input: "abcbab"
Output: "bbbaac"
Explanation: 'b' appeared three times, 'a' appeared twice, and 'c' appeared only once.
```

思路

1. 需要统计词频
2. 需要将所有字母插入到堆中

```
using namespace std;

#include <iostream>
#include <queue>
#include <string>
#include <unordered_map>

struct Compare{
   bool operator()(pair<char, int> x1, pair<char, int> x2)
   {
       return x1.second < x2.second;
   }
};
class FrequencySort {
public:
    static string sortCharacterByFrequency(const string &str) {
        string sortedString = "";
        // TODO: Write your code here
        unordered_map<char, int> frequency;
        priority_queue<pair<char, int>, vector<pair<char, int>>, Compare> max_heap;
        for(int i = 0; i < str.size(); i++)
        {
            frequency[str[i]]++;
        }
        for(auto elem : frequency)
        {
            max_heap.push(elem);
        }
        while(!max_heap.empty())
        {
            int n = max_heap.top().second;
            while(n--)
            {
                sortedString.push_back(max_heap.top().first);
            }
            max_heap.pop();
        }
        return sortedString;
    }
};

int main(int argc, char *argv[]) {
    string result = FrequencySort::sortCharacterByFrequency("Programming");
    cout << "Here is the given string after sorting characters by frequency: " << result << endl;

    result = FrequencySort::sortCharacterByFrequency("abcbab");
    cout << "Here is the given string after sorting characters by frequency: " << result << endl;
}
```

2-5 计算流中第K大的数字

> 设计一个类去有效寻找在流中第K大的数字。
> 这个类有如下来个操作。
> 构造函数，接受一个数组和一个K
> 然后给定一个add操作。

```
Input: [3, 1, 5, 12, 2, 11], K = 4
1. Calling add(6) should return '5'.
2. Calling add(13) should return '6'.
2. Calling add(4) should still return '6'.
```

思路

因为要保持第K大，因此需要维护一个K大小的最小堆。

```
using namespace std;
```

```cpp
#include <iostream>
#include <queue>
#include <vector>

class KthLargestNumberInStream {
public:
    KthLargestNumberInStream(const vector<int> &nums, int k) {
        // TODO: Write your code here
        for(int i = 0; i < nums.size(); i++)
        {
            min_heap.push(nums[i]);
            if(min_heap.size() > k)
            {
                min_heap.pop();
            }
        }
        K = k;
    }

    virtual int add(int num) {
        // TODO: Write your code here
        min_heap.push(num);
        min_heap.pop();
        return min_heap.top();
    }
    priority_queue<int, vector<int> , greater<int>> min_heap;
    int K;
};

int main(int argc, char *argv[]) {
    KthLargestNumberInStream kthLargestNumber({3, 1, 5, 12, 2, 11}, 4);
    cout << "4th largest number is: " << kthLargestNumber.add(6) << endl;
    cout << "4th largest number is: " << kthLargestNumber.add(13) << endl;
    cout << "4th largest number is: " << kthLargestNumber.add(4) << endl;
}
```

2-6 寻找距离X最近的K个值

> 给定一个数组，给定一个值K和x。
> 寻找距离x最近的K个值。

e.g.1

```
Input: [5, 6, 7, 8, 9], K = 3, X = 7
Output: [6, 7, 8]
```

e.g.2

```
Input: [2, 4, 5, 6, 9], K = 3, X = 6
Output: [4, 5, 6]
```

思路

用距离的绝对值比较。用值存储。

```cpp
using namespace std;

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

class KClosestElements {
public:
    struct numCompare {
        bool operator()(const pair<int, int> &x, const pair<int, int> &y) { return x.first < y.first; }
    };

    static vector<int> findClosestElements(const vector<int> &arr, int K, int X) {
        vector<int> result;
        priority_queue<pair<int, int>, vector<pair<int, int>>, numCompare > max_heap;
        for(int i = 0; i < arr.size(); i++)
        {
            max_heap.push(make_pair(abs(arr[i] - X), arr[i]));
            if(max_heap.size() > K)
                max_heap.pop();
        }
        while(!max_heap.empty())
        {
            result.push_back(max_heap.top().second);
            max_heap.pop();
        }
        // TODO: Write your code here
        return result;
    }
};

int main(int argc, char *argv[]) {
    vector<int> result = KClosestElements::findClosestElements(vector<int>{5,
6, 7, 8, 9}, 3, 7);
    cout << ”’K’ closest numbers to ’X’ are: ”;
    for (auto num : result) {
        cout << num << ” ”;
    }
    cout << endl;

    result = KClosestElements::findClosestElements(vector<int>{2, 4, 5, 6,
9}, 3, 6);
    cout << ”’K’ closest numbers to ’X’ are: ”;
    for (auto num : result) {
        cout << num << ” ”;
    }
    cout << endl;
}
```

2.7 删除K个数字

> 给定一个数组，和一个K值。从数组中移除K个值，在数组中留下最多的不同数字。并返回不同数字的种类。

e.g.1

```
Input: [7, 3, 5, 8, 5, 3, 3], and K=2
Output: 3
Explanation: We can remove two occurrences of 3 to be left with 3 distinct
numbers [7, 3, 8], we have
to skip 5 because it is not distinct and occurred twice.
Another solution could be to remove one instance of ’5’ and ’3’ each to be
left with three
distinct numbers [7, 5, 8], in this case, we have to skip 3 because it
occurred twice.
```

可以移除俩个3。留下7，3，5，8，5，因为5存在多个。留下[7 3 8]

e.g.2

```
Input: [3, 5, 12, 11, 12], and K=3
Output: 2
Explanation: We can remove one occurrence of 12, after which all numbers will
become distinct. Then
we can delete any two numbers which will leave us 2 distinct numbers in the
result.
```

思路

1. 需要从数组中删除K个值。
2. 在剩下的数组中选取unique的数字的个数。

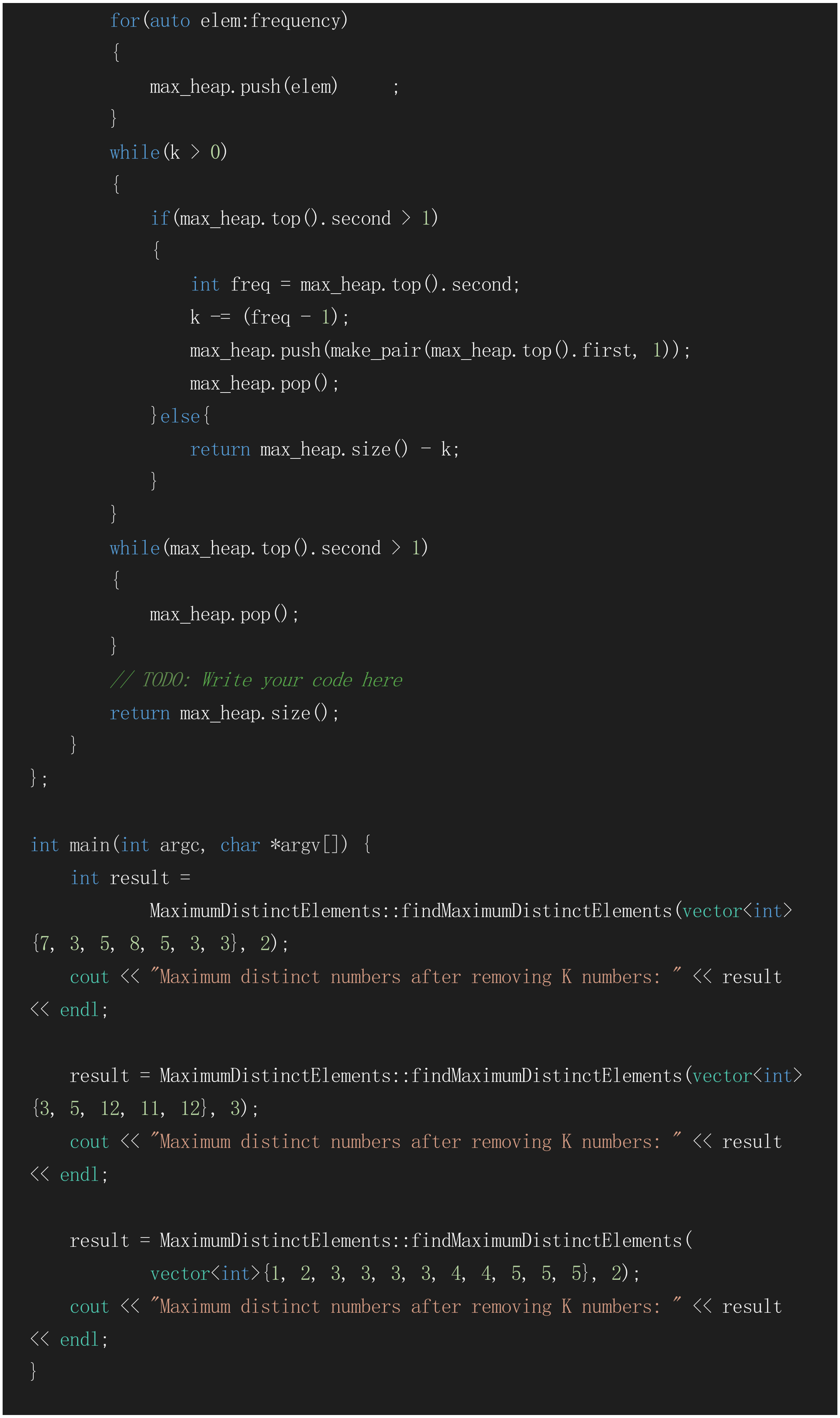
可以从词频排序做起。如果最高词频大于1，且为M。就可以删除M-1个数字。如果为1，则任意删除K个数字。

```
using namespace std;

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

struct Compare{
    bool operator()(pair<int, int> x1, pair<int, int> x2)
    {
        return x1.second < x2.second;
    }
};
class MaximumDistinctElements {
public:
    static int findMaximumDistinctElements(const vector<int> &nums, int k) {
        int distinctElementsCount = 0;
        // value,  freq
        unordered_map<int, int> frequency;
        for(int i = 0; i < nums.size(); i++)
        {
            frequency[nums[i]]++;
        }

        priority_queue<pair<int, int>, vector<pair<int, int>>, Compare>
max_heap;
```



```
        for(auto elem:frequency)
        {
            max_heap.push(elem)     ;
        }
        while(k > 0)
        {
            if(max_heap.top().second > 1)
            {
                int freq = max_heap.top().second;
                k -= (freq - 1);
                max_heap.push(make_pair(max_heap.top().first, 1));
                max_heap.pop();
            }else{
                return max_heap.size() - k;
            }
        }
        while(max_heap.top().second > 1)
        {
            max_heap.pop();
        }
        // TODO: Write your code here
        return max_heap.size();
    }
};

int main(int argc, char *argv[]) {
    int result =
            MaximumDistinctElements::findMaximumDistinctElements(vector<int>
{7, 3, 5, 8, 5, 3, 3}, 2);
    cout << "Maximum distinct numbers after removing K numbers: " << result
<< endl;

    result = MaximumDistinctElements::findMaximumDistinctElements(vector<int>
{3, 5, 12, 11, 12}, 3);
    cout << "Maximum distinct numbers after removing K numbers: " << result
<< endl;

    result = MaximumDistinctElements::findMaximumDistinctElements(
            vector<int>{1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5}, 2);
    cout << "Maximum distinct numbers after removing K numbers: " << result
<< endl;
}
```

1. 首先将所有词频大于1的，都可以减到词频=1。如果k没被剪完。就可以任意减数字了。
2. 如果k被剪完了，那么有可能存在词频大于1的。那么就将其全部删除。

2.8 计算区间和

> 给定一个数组。计算在K1th和K2th之间的元素和。

e.g.1

```
Input: [1, 3, 12, 5, 15, 11], and K1=3, K2=6
Output: 23
Explanation: The 3rd smallest number is 5 and 6th smallest number 15. The sum
of numbers coming
between 5 and 15 is 23 (11+12).
```

e.g.2

```
Input: [3, 5, 8, 7], and K1=1, K2=4
Output: 12
Explanation: The sum of the numbers between the 1st smallest number (3) and
the 4th smallest
number (8) is 12 (5+7).
```

思路

1. 求出K1和K2位置的值。
2. 然后计算元素和。

比较直接的思路是维护K2大小的堆。弹出元素直到K1。然后开始计算元素和。

```
using namespace std;

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

class SumOfElements {
  public:
    struct numCompare {
        bool operator()( const int &x, const int &y ) {
            return x > y;
        }
    };

    static int findSumOfElements( const vector<int> &nums, int k1, int k2 ) {
        priority_queue<int, vector<int>, numCompare> minHeap;

        // insert all numbers to the min heap
        for ( int i = 0; i < nums.size(); i++ ) {
            minHeap.push( nums[i] );
        }

        // remove k1 small numbers from the min heap
        for ( int i = 0; i < k1; i++ ) {
            minHeap.pop();
        }

        int elementSum = 0;
        // sum next k2-k1-1 numbers
        for ( int i = 0; i < k2 - k1 - 1; i++ ) {
            elementSum += minHeap.top();
            minHeap.pop();
        }

        return elementSum;
    }
};

int main( int argc, char *argv[] ) {
    int result = SumOfElements::findSumOfElements( vector<int> {1, 3, 12, 5, 15, 11}, 3, 6 );
    cout << "Sum of all numbers between k1 and k2 smallest numbers: " << result << endl;

    result = SumOfElements::findSumOfElements( vector<int> {3, 5, 8, 7}, 1, 4 );
    cout << "Sum of all numbers between k1 and k2 smallest numbers: " << result << endl;
}
```

另一个思路：
也可以维护前K2个值大小的堆，然后遍历数组。最后再减去K1部分。

2.9 重排字符串

> 给定一个字符串，重排字符串，使之重排之后没有相同字符在一起。

e.g.1

```
Input: "aappp"
Output: "papap"
Explanation: In "papap", none of the repeating characters come next to each other.
```

e.g.2

```

Input: "Programming"
Output: "rgmrgmPiano" or "gmringmrPoa" or "gmrPagimnor", etc.
Explanation: None of the repeating characters come next to each other.
```

思路

1. 需要计算字符的频率。
2. 利用最大堆存放。

一开始肯定要插入最大的频率的字符。然后放回去。但是这样可能会出现重复字符，所以需要下一次再放回去，即等待下一个字符插入后再放回去。

看code就很容易明白了。

```cpp
using namespace std;

#include <iostream>
#include <queue>
#include <string>
#include <unordered_map>
#include <vector>

struct Compare {
    bool operator()( pair<char, int> x1, pair<char, int> x2 ) {
        return x1.second < x2.second;
    }
};
class RearrangeString {
  public:
    static string rearrangeString( const string &str ) {
        // TODO: Write your code here
        unordered_map<char, int>frequency;
        for ( int i = 0; i < str.size(); i++ )
            frequency[str[i]]++;
        priority_queue<pair<char, int>, vector<pair<char, int>>, Compare> max_heap;
        for ( auto elem : frequency )
            max_heap.push( elem );
        pair<char, int> pre_entry{ -1, -1 };
        string result;
        while ( !max_heap.empty() ) {
            auto curr_entry = max_heap.top();
            max_heap.pop();
            if ( pre_entry.second > 0 )
                max_heap.push( pre_entry );
            curr_entry.second--;
            pre_entry = curr_entry;

            result += curr_entry.first;
        }
        return result;
    }
};

int main( int argc, char *argv[] ) {
    cout << "Rearranged string: " << RearrangeString::rearrangeString( "aappp" ) << endl;
    cout << "Rearranged string: " << RearrangeString::rearrangeString( "Programming" ) << endl;
    cout << "Rearranged string: " << RearrangeString::rearrangeString( "aapa" ) << endl;
    system( "pause" );
}
```

3. 挑战

3.1 重排字符

> 给定一个字符，重排字符，使之相同距离的字符为K个字符。

e.g.1

```
Input: "Programming", K=3
Output: "rgmPrgmiano" or "gmringmrPoa" or "gmrPagimnor" and a few more
Explanation: All same characters are 3 distance apart.
```

思路

和上题思路一样。只不过这次受K控制距离而已。我们可以将上题中的queue看做是一个。我们每次记录前K-1个元素。当插入第K-1个元素后。队列中第1个元素又可以重新插入了。所以放回到到max_heap中。

```
using namespace std;

#include <iostream>
#include <queue>
#include <string>
#include <unordered_map>
#include <vector>

struct Compare {
    bool operator()( pair<char, int> x1, pair<char, int> x2 ) {
        return x1.second < x2.second;
    }
};
class RearrangeStringKDistanceApart {
  public:
    static string reorganizeString( const string &str, int k ) {
        // TODO: Write your code here
        unordered_map<char, int>frequency;
        for ( int i = 0; i < str.size(); i++ )
            frequency[str[i]]++;
        priority_queue<pair<char, int>, vector<pair<char, int>>, Compare>
max_heap;
        for ( auto elem : frequency )
            max_heap.push( elem );
        queue<pair<char, int>> pre_entry_queue;
        string result;
        while ( !max_heap.empty() ) {
            auto curr_entry = max_heap.top();
            max_heap.pop();
            if ( pre_entry_queue.size() >= k - 1 ) {
                auto pre_entry = pre_entry_queue.front();
                pre_entry_queue.pop();
                if( pre_entry.second > 0 )
                    max_heap.push( pre_entry );
            }
            curr_entry.second--;
            pre_entry_queue.push( curr_entry );
            result += curr_entry.first;
        }
        return result.size() == str.size() ? result : "";
    }
};

int main( int argc, char *argv[] ) {
    cout << "Reorganized string: "
         << RearrangeStringKDistanceApart::reorganizeString( "mmpp", 2 ) <<
endl;
    cout << "Reorganized string: "
         << RearrangeStringKDistanceApart::reorganizeString( "Programming", 3
) << endl;
    cout << "Reorganized string: "
         << RearrangeStringKDistanceApart::reorganizeString( "aab", 2 ) <<
endl;
    cout << "Reorganized string: " <<
RearrangeStringKDistanceApart::reorganizeString( "aappa", 3 )
         << endl;
    system( "pause" );
}
```

3.2 安排任务

> 给定一系列任务，相同的任务间隔期为k。如果没有任务执行，可以执行idle任务。输出最大任务长度

e.g.1

```
Input: [a, a, a, b, c, c], K=2
```

```
Output: 7
Explanation: a -> c -> b -> a -> c -> idle -> a
```

e.g.2

```
Input: [a, b, a], K=3
Output: 5
Explanation: a -> b -> idle -> idle -> a
```

思路

这里的思路有点怪的。我自己是没想出来。

这里要明白什么时候插入idle节点。如果每次迭代能走完K+1。就不需要插入idle。如果走不完。则需要插入idle。

```
using namespace std;

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

class TaskScheduler {
  public:
    struct valueCompare {
        char operator()( const pair<int, int> &x, const pair<int, int> &y ) {
            return y.second > x.second;
        }
    };

    static int scheduleTasks( vector<char> &tasks, int k ) {
        int intervalCount = 0;
        unordered_map<char, int> taskFrequencyMap;
        for ( char chr : tasks ) {
            taskFrequencyMap[chr]++;
        }

        priority_queue<pair<char, int>, vector<pair<char, int>>,
valueCompare> maxHeap;

        // add all tasks to the max heap
        for ( auto entry : taskFrequencyMap ) {
            maxHeap.push( entry );
        }

        while ( !maxHeap.empty() ) {
            vector<pair<char, int>> waitList;
            int n = k + 1;  // try to execute as many as 'k+1' tasks from the
max-heap
            for ( ; n > 0 && !maxHeap.empty(); n-- ) {
                intervalCount++;
                auto currentEntry = maxHeap.top();
                maxHeap.pop();
                if ( currentEntry.second > 1 ) {
                    currentEntry.second--;
                    waitList.push_back( currentEntry );
                }
            }
            // put all the waiting list back on the heap
            for ( auto it = waitList.begin(); it != waitList.end(); it++ ) {
                maxHeap.push( *it );
            }
            if ( !maxHeap.empty() ) {
                intervalCount += n;  // we'll be having 'n' idle intervals
for the next iteration
            }
        }

        return intervalCount;
    }
};

int main( int argc, char *argv[] ) {
    vector<char> tasks = {'a', 'a', 'a', 'b', 'c', 'c'};
    cout << "Minimum intervals needed to execute all tasks: "
         << TaskScheduler::scheduleTasks( tasks, 2 ) << endl;

    tasks = vector<char> {'a', 'b', 'a'};
    cout << "Minimum intervals needed to execute all tasks: "
         << TaskScheduler::scheduleTasks( tasks, 3 ) << endl;
}
```