# 题目

输入整数数组arr，找出其中最小的k个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

**示例 1：**

输入：arr = [3,2,1], k = 2

输出：[1,2] 或者 [2,1]

**示例 2：**

输入：arr = [0,1,2,1], k = 1

输出：[0]

**限制：**

0 <= k <= arr.length <= 10000

0 <= arr[i] <= 10000

**说明：**类似LeetCode 703

# 分析

## 方法一：排序

**思路：**

对原数组从小到大排序后取出前k个数即可。

**代码：**

class Solution {

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

vector<int> vec(k, 0);

sort(arr.begin(), arr.end()); //排序

for (int i = 0; i < k; ++i) vec[i] = arr[i]; //申请内存暂存

return vec;

}

};

**复杂度：**

时间复杂度：O(nlogn)，其中n是数组arr的长度。算法的时间复杂度即排序的时间复杂度。

空间复杂度：O(logn)，排序所需额外的空间复杂度为O(logn)。

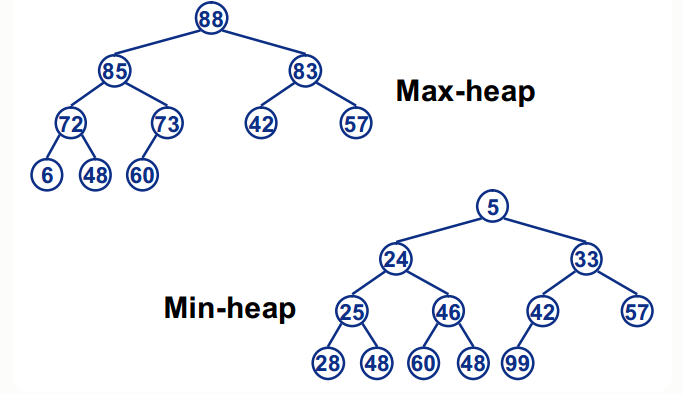
## 方法二：堆/优先队列

**思路：**

我们用一个大根堆实时维护数组的前k小值。首先将前k个数插入大根堆中，随后从第 k+1个数开始遍历，如果当前遍历到的数比大根堆的堆顶的数要小，就把堆顶的数弹出，再插入当前遍历到的数。最后将大根堆里的数存入数组返回即可。

由于C++语言中的堆（即优先队列）为大根堆，我们可以这么做。而Python语言中的堆为小根堆，因此我们要对数组中所有的数取其相反数，才能使用小根堆维护前k小值。

**C++语言中的堆（即优先队列）为大根堆**。



### **小顶堆**

**代码：**

class Solution {

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

priority\_queue<int,vector<int>,greater<int>> pri\_queue;

vector<int> vec;

for(auto c :arr)

pri\_queue.push(c); //元素全部存入，堆自动排序

for(auto i = 0; i< k;i++)

{

vec.push\_back(pri\_queue.top()); //只需要取K个值即可

pri\_queue.pop();

}

return vec;

}

};

说明：这种方法简单易懂，但是比较浪费空间，其实只需要维护k大小的堆即可，不需要全部都放到堆中。

**另一种写法：**

class Solution {

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

priority\_queue<int, vector<int>, greater<int> > pq;

vector<int> ans;

for( int v : arr ) pq.push(v);

while(k--){

ans.push\_back( pq.top() );

pq.pop();

}

}

};

**错误答案：**

class Solution {

public:

    vector<int> getLeastNumbers(vector<int>& arr, int k) {

        priority\_queue<int,vector<int>,greater<int>> pri\_que;

        vector<int> ret;

        for(int i:arr)

        {

            pri\_que.push(i);

            if(pri\_que.size()>k)

            {

                pri\_que.pop();//此时将最小值剔除了

            }

        }

        while(!pri\_que.empty())

        {

            ret.push\_back(pri\_que.top());

            pri\_que.pop();

        }

        return ret;

    }

};

**错误答案：**

class Solution {

public:

    vector<int> getLeastNumbers(vector<int>& arr, int k) {

        priority\_queue<int,vector<int>,greater<int>> pri\_que;

        vector<int> ret;

        if(0 ==k)

            return ret;

        for(int i=0;i<arr.size();i++)

        {

            if(pri\_que.size()<k)

            {

                pri\_que.push(arr.at(i));

            }

            else

            {

                if(pri\_que.top() > arr.at(i))

                {

                    pri\_que.pop();

                    pri\_que.push(arr.at(i));

                }

            }

        }

        for(int i=0;i<k;i++)

        {

            ret.push\_back(pri\_que.top());

            pri\_que.pop();

        }

        return ret;

    }

};

说明：上述方法维护一个k大小的小顶堆是错误的，这样其实已经改变了最小值。

### 大顶堆

**为了更加节约空间，可以只维护k大小的堆：**

class Solution {

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

vector<int>vec(k, 0);

if (k == 0) return vec; // 排除 0 的情况

priority\_queue<int> pri\_queue;//默认大顶堆

for (int i = 0; i < k; ++i)

pri\_queue.push(arr[i]);

for (int i = k; i < (int)arr.size(); ++i) {

if (pri\_queue.top() > arr[i]) {

pri\_queue.pop();

pri\_queue.push(arr[i]);

}

}

for (int i = 0; i < k; ++i) {

vec[i] = pri\_queue.top();

pri\_queue.pop();

}

return vec;

}

};

**另一种写法：**

class Solution {

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

vector<int>vec;

if(0 == k) //特殊情况：排除k=0的情况

return vec;

std::priority\_queue<int> pri\_queue; //使用默认的堆，大根堆即可

for(size\_t i=0;i<arr.size();i++){

if(pri\_queue.size() < k)//不要写成i<k

{

pri\_queue.push(arr.at(i)); //先存放k个数

}else{

if(pri\_queue.top() > arr.at(i)) //存入更小的值

{

pri\_queue.pop();

pri\_queue.push(arr.at(i));

}

}

}

for(size\_t i=0;i<k;++i)

{

vec.push\_back(pri\_queue.top());

pri\_queue.pop();

}

return vec;

}

};

**复杂度：**

时间复杂度：O(nlogk)，其中n是数组arr的长度。由于大根堆实时维护前k小值，所以插入删除都是O(logk) 的时间复杂度，最坏情况下数组里n个数都会插入，所以一共需要 O(nlogk)的时间复杂度。

空间复杂度：O(k)，因为大根堆里最多k个数。

## 方法三：快排思想

思路：

我们可以借鉴快速排序的思想。我们知道快排的划分函数每次执行完后都能将数组分成两个部分，小于等于分界值 pivot 的元素的都会被放到数组的左边，大于的都会被放到数组的右边，然后返回分界值的下标。与快速排序不同的是，快速排序会根据分界值的下标递归处理划分的两侧，而这里我们只处理划分的一边。

我们定义函数 randomized\_selected(arr, l, r, k)表示划分数组arr的[l,r]部分，使前k小的数在数组的左侧，在函数里我们调用快排的划分函数，假设划分函数返回的下标是pos（表示分界值pivot最终在数组中的位置），即pivot是数组中第pos - l + 1小的数，那么一共会有三种情况：

如果pos-l+1==k，表示pivot就是第kk小的数，直接返回即可；

如果pos-l+1<k，表示第k小的数在 pivot 的右侧，因此递归调用randomized\_selected(arr, pos + 1, r, k - (pos - l + 1))；

如果pos-l + 1 > k，表示第k 小的数在pivot的左侧，递归调用randomized\_selected(arr, l, pos - 1, k)。

函数递归入口为randomized\_selected(arr, 0, arr.length - 1, k)。在函数返回后，将前k个数放入答案数组返回即可。

**代码：**

class Solution {

int partition(vector<int>& nums, int l, int r) {

int pivot = nums[r];

int i = l - 1;

for (int j = l; j <= r - 1; ++j) {

if (nums[j] <= pivot) {

i = i + 1;

swap(nums[i], nums[j]);

}

}

swap(nums[i + 1], nums[r]);

return i + 1;

}

// 基于随机的划分

int randomized\_partition(vector<int>& nums, int l, int r) {

int i = rand() % (r - l + 1) + l;

swap(nums[r], nums[i]);

return partition(nums, l, r);

}

void randomized\_selected(vector<int>& arr, int l, int r, int k) {

if (l >= r) return;

int pos = randomized\_partition(arr, l, r);

int num = pos - l + 1;

if (k == num) return;

else if (k < num) randomized\_selected(arr, l, pos - 1, k);

else randomized\_selected(arr, pos + 1, r, k - num);

}

public:

vector<int> getLeastNumbers(vector<int>& arr, int k) {

srand((unsigned)time(NULL));

randomized\_selected(arr, 0, (int)arr.size() - 1, k);

vector<int>vec;

for (int i = 0; i < k; ++i) vec.push\_back(arr[i]);

return vec;

}

};

。

**复杂度：**

# 总结

对于经典TopK问题，4种通用解决方案。

解题思路：

1、用快排最高效解决TopK问题；

2、大根堆(前K小) /小根堆（前K大)，Java中有现成的PriorityQueue，实现起来最简单；

3、二叉搜索树也可以解决 TopK问题；

4、数据范围有限时直接计数排序就行了。