# 第三大的数

1. 题目描述

给定一个非空数组，返回此数组中第三大的数。如果不存在，则返回数组中最大的数。要求算法时间复杂度必须是O(n)。请注意，你需要找的是数组排序后的第 3个最大的元素，而不是第 3个不同的元素。（LeetCode 147.）

1. 题目分析

要想O(n)就得记录走过的历史信息，使用三个临时变量，分别记录最大、次大、第三大

1. 代码实现
2. **class** Solution {
3. **public**:
4. //用快排的时间复杂度是O(nlogn),空间复杂度是O(1)
6. **int** thirdMax(vector<**int**>& nums) {
7. **int** n = nums.size();
8. **if**(!n) **return** -1;
9. **long** **long** first = -3e9, second = -3e9, third = -3e9;
11. **for**(**int** i = 0; i < n; i++){
12. **if**(nums[i] == first || nums[i] == second || nums[i] == third)
13. **continue**;
14. **if**(nums[i] > first){
15. third = second;
16. second = first;
17. first = nums[i];
18. }
19. **else** **if**(nums[i] > second){
20. third = second;
21. second = nums[i];
22. }
23. **else** **if**(nums[i] > third){
24. third = nums[i];
25. }
26. }
28. **if**(third == -3e9) **return** first;
29. **return** third;
30. }
31. };

# 第k大的数

1. 题目描述

在未排序的数组中找到第 **k** 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

1. 题目分析
2. 快排优化

即快排每次排序排好一个标兵，然后以此标兵为界将其左右区间分别排序。此题只需要找到第k大的数，如果是从小到大排序的话就是倒数第k个数（即n-k的位置），所以每次快排后，如果返回的标兵的位置在n-k的左面，那么只拍另一个右边的区间；相反就排左边的区间。这样选择排序区间后时间复杂度可以降到O(n)，空间复杂度是O(1)

1. 堆排序

若使用小顶堆的话，只能维护一个容量为k的小顶堆，当k+1个元素时就去掉堆顶元素，最后剩下的是TopK的元素，则堆顶的元素就是第k大的元素

若使用大顶堆，则需要将所有元素入堆后在进行删除k-1次，那么此时堆顶的元素就是第k大的元素。

大顶堆、小顶堆可以使用STL中的优先队列（自动建堆、调整堆），也可以使用自己的写的堆排序

1. 代码实现
2. 快速排序优化
3. **class** Solution {
4. **public**:
5. **int** findKthLargest(vector<**int**>& nums, **int** k) {
6. **int** n = nums.size();
7. **if**(n < k)
8. **return** -1;
10. //sort(nums.begin(), nums.end(), less<int>());
11. quickSort(nums, 0, n-1, k);
12. **return** nums[n-k];
13. }
14. **private**:
15. **void** quickSort(vector<**int**>& nums, **int** low, **int** high, **int** k){
16. **if**(low >= high)
17. **return**;
18. **int** i = low, j = high;
19. **int** flag = nums[i];
20. **while**(i < j){
21. **while**(i < j && nums[j] >= flag)
22. j--;
23. **if**(i < j){
24. nums[i] = nums[j];
25. i++;
26. }
27. **while**(i < j && nums[i] <= flag)
28. i++;
29. **if**(i < j){
30. nums[j] = nums[i];
31. j--;
32. }
33. }
34. nums[i] = flag;
35. **if**(i == (nums.size()-k))
36. **return** ;
37. **else** **if**(i > nums.size()-k)
38. quickSort(nums, low, i-1, k);
39. **else**
40. quickSort(nums, i+1, high, k);
41. }
42. };
43. 小顶堆

时间复杂度是O(nlogk)，空间复杂度是O(k)

优先队列：

1. **class** Solution {
2. **public**:
3. **int** findKthLargest(vector<**int**>& nums, **int** k) {
4. **int** n = nums.size();
5. **if**(!n) **return** -1;
7. priority\_queue<**int**, vector<**int**>, greater<**int**>> minheap;
8. **for**(**int** i = 0; i < n; i++){
9. minheap.push(nums[i]);
10. **if**(minheap.size() > k)
11. minheap.pop();
12. }
14. **return** minheap.top();
15. }
17. };

自己建堆、调整堆：

1. **class** Solution {
2. **public**:
3. **int** findKthLargest(vector<**int**>& nums, **int** k) {
4. // 堆排序
5. **int** n = nums.size();
6. **if**(n == 0) **return** 0;
7. **if**(n == 1) **return** nums[0];
8. // 建K个元素的小顶堆之后，之后需要删除然后自上而下调整
9. // 调用adjust\_heap从下往上、从右往左依次调整每个子堆
10. // 对于父节点i，左子树2i+1,右子树2i+2
11. // 对于子节点i，父节点(i-1)/2
12. vector<**int**> minheap(nums.begin(), nums.begin()+k);
13. **int** nmin = minheap.size();
14. **for**(**int** i = nmin/2-1; i >= 0; i--){
15. adjust\_heap(minheap, i, nmin);
16. }
17. **for**(**int** i = k; i < n; i++){
18. **if**(nums[i] > minheap[0]){
19. minheap[0] = nums[i];
20. adjust\_heap(minheap, 0, nmin);
21. }
22. }
24. **return** minheap[0];
25. }
27. **void** adjust\_heap(vector<**int**>& nums, **int** i, **int** n){  //对以第i个节点作为堆顶的子堆进行调整
28. **int** left = 2\*i+1, right = 2\*i+2;
29. **int** maxIndex = left;
30. **while**(left < n){
31. **if**(right < n && nums[left] > nums[right]){  //取到左右子树中较小的那个
32. maxIndex = right;
33. }
34. **if**(nums[i] > nums[maxIndex])  //建立小顶堆，如果堆顶元素大于子节点中较小的那个，则交换
35. swap(nums[i], nums[maxIndex]);
36. **else**  //否则已经是小顶堆，无需调整，退出
37. **break**;
39. i = maxIndex;   // 迭代调整
40. left = 2\*i+1, right = 2\*i+2;
41. maxIndex = left;
42. }
43. }
45. };
46. 大顶堆

时间复杂度是O(n+klogn)约等于O(nlogn)，空间复杂度是O(1)

1. **class** Solution {
2. **public**:
3. **int** findKthLargest(vector<**int**>& nums, **int** k) {
4. // 堆排序
5. **int** n = nums.size();
6. **if**(n == 0) **return** 0;
7. **if**(n == 1) **return** nums[0];
8. // 建大顶堆之后，堆顶最大，之后需要删除然后自上而下调整
9. // 调用adjust\_heap从下往上、从右往左依次调整每个子堆
10. // 对于父节点i，左子树2i+1,右子树2i+2
11. // 对于子节点i，父节点(i-1)/2
12. **for**(**int** i = n/2-1; i >= 0; i--){
13. adjust\_heap(nums, i, n);
14. }
15. **for**(**int** i = n-1; i >= n-k; i--){
16. swap(nums[0], nums[i]);
17. adjust\_heap(nums, 0, i);
18. }
20. **return** nums[n-k];
21. }
23. **void** adjust\_heap(vector<**int**>& nums, **int** i, **int** n){  //对以第i个节点作为堆顶的子堆进行调整
24. **int** left = 2\*i+1, right = 2\*i+2;
25. **int** maxIndex = left;
26. **while**(left < n){
27. **if**(right < n && nums[left] < nums[right]){  //取到左右子树中较小的那个
28. maxIndex = right;
29. }
30. **if**(nums[i] < nums[maxIndex])  //建立小顶堆，如果堆顶元素大于子节点中较小的那个，则交换
31. swap(nums[i], nums[maxIndex]);
32. **else**  //否则已经是小顶堆，无需调整，退出
33. **break**;
35. i = maxIndex;   // 迭代调整
36. left = 2\*i+1, right = 2\*i+2;
37. maxIndex = left;
38. }
39. }
41. };

# 前K个高频元素

1. 题目描述

给定一个非空的整数数组，返回其中出现频率前 **k**高的元素。

1. 题目分析
2. 小顶堆

时间复杂度O(nlogk)，空间复杂度O(n+k)

1. 减治法的快速排序

和第k个最大的数的实现方式一样，只不过里面的item不是int，而是pair对，需要用pair对的频率进行排序

时间复杂度最优是O(n)，最坏O(n^2)

1. 桶排序

首先使用Hashmap记录元素出现的频率，然后使用频率作为桶号存储元素，最后根据大的桶号（出现频率高）统计够K个元素。

时间复杂度是O(n)，空间复杂度是O(n)

1. 代码实现
2. 小顶堆
3. **class** Solution {
4. **public**:
5. //堆排序，找topk大的就用小顶堆，找topk小的就用大顶堆
6. vector<**int**> topKFrequent(vector<**int**>& nums, **int** k) {
7. **int** n = nums.size();
8. **if**(n < 2) **return** nums;
9. unordered\_map<**int**, **int**> map;
10. **for**(**int** i = 0; i < n; i++)
11. map[nums[i]]++;
13. priority\_queue<pair<**int**, **int**>, vector<pair<**int**, **int**>>, greater<pair<**int**, **int**>>> minheap;
14. **for**(auto p : map){
15. **if**(minheap.size() < k){
16. minheap.push(pair<**int**, **int**>(p.second, p.first));
17. }
18. **else** **if**(p.second > minheap.top().first){
19. minheap.pop();
20. minheap.push(pair<**int**, **int**>(p.second, p.first));
21. }
22. }
24. vector<**int**> res(k);
25. **for**(**int** i = 0; i < k; i++){
26. res[k-1-i] = minheap.top().second;
27. minheap.pop();
28. }
30. **return** res;
31. }
32. };

2. 减治法的快速排序

目前没用C++实现，如果有人实现了，欢迎PR

3. 桶排序

1. **class** Solution {
2. **public**:
3. vector<**int**> topKFrequent(vector<**int**>& nums, **int** k) {
4. //统计次数
5. unordered\_map<**int**, **int**> m;
7. **for** (**int** i = 0; i < nums.size(); ++i)
8. {
9. ++m[nums[i]];
10. }
12. //使用二维数组  统计次数作为第一维（如果统计次数有相同，则将值追加到桶中）
13. vector<vector<**int**>> buckets(nums.size() + 1);
15. **for** (auto iter = m.begin(); iter != m.end(); ++iter)
16. {
17. buckets.at(iter->second).push\_back(iter->first);
18. }
20. //将buckets中前k个高频元素放入res中
21. vector<**int**> res(k);
22. **int** count = 0;
23. **for** (**int** i = buckets.size() - 1; i >= 0; --i)
24. {
25. **for** (**int** j = 0; j < buckets.at(i).size(); ++j)
26. {
27. res.at(count) = buckets.at(i).at(j);
28. ++count;
29. **if** (count == k)
30. **return** res;
31. }
32. }
34. **return** res;
35. }
36. };