<https://leetcode.com/problems/find-median-from-data-stream/>

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

* For example, for arr = [2,3,4], the median is 3.
* For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.

Implement the MedianFinder class:

* MedianFinder() initializes the MedianFinder object.
* void addNum(int num) adds the integer num from the data stream to the data structure.
* double findMedian() returns the median of all elements so far. Answers within 10-5 of the actual answer will be accepted.

**Example 1:**

Input

["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]

[[], [1], [2], [], [3], []]

Output

[null, null, null, 1.5, null, 2.0]

Explanation

MedianFinder medianFinder = new MedianFinder();

medianFinder.addNum(1); // arr = [1]

medianFinder.addNum(2); // arr = [1, 2]

medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)

medianFinder.addNum(3); // arr[1, 2, 3]

medianFinder.findMedian(); // return 2.0

**Constraints:**

* -105 <= num <= 105
* There will be at least one element in the data structure before calling findMedian.
* At most 5 \* 104 calls will be made to addNum and findMedian.

**Follow up:**

* If all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?
* If 99% of all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?

**Attempt 1: 2023-10-12**

**Solution 1: Two Heaps (30 min)**

**Style 1: Put on MaxPQ first**

class MedianFinder {

PriorityQueue<Integer> minPQ;

PriorityQueue<Integer> maxPQ;

public MedianFinder() {

minPQ = new PriorityQueue<>();

maxPQ = new PriorityQueue<>((a, b) -> b - a);

}

public void addNum(int num) {

if(maxPQ.isEmpty() || num <= maxPQ.peek()) {

if(maxPQ.size() > minPQ.size()) {

minPQ.offer(maxPQ.poll());

}

maxPQ.offer(num);

} else if(minPQ.isEmpty() || num >= minPQ.peek()) {

if(minPQ.size() > maxPQ.size()) {

maxPQ.offer(minPQ.poll());

}

minPQ.offer(num);

} else {

if(maxPQ.size() <= minPQ.size()) {

maxPQ.offer(num);

} else {

minPQ.offer(num);

}

}

}

public double findMedian() {

if(maxPQ.isEmpty() && minPQ.isEmpty()) {

return 0;

}

if(maxPQ.size() > minPQ.size()) {

return maxPQ.peek();

} else if(maxPQ.size() < minPQ.size()) {

return minPQ.peek();

} else {

return ((double)maxPQ.peek() + (double)minPQ.peek()) / 2;

}

}

}

/\*\*

\* Your MedianFinder object will be instantiated and called as such:

\* MedianFinder obj = new MedianFinder();

\* obj.addNum(num);

\* double param\_2 = obj.findMedian();

\*/

Time Complexity: O(logN)

**Style 2: Put on MinPQ first**

**The only difference than Style 1 is switching on logic branch of insert minPQ / maxPQ in 'addNum' method**

class MedianFinder {

PriorityQueue<Integer> minPQ;

PriorityQueue<Integer> maxPQ;

public MedianFinder() {

minPQ = new PriorityQueue<>();

maxPQ = new PriorityQueue<>((a, b) -> b - a);

}

public void addNum(int num) {

if(minPQ.isEmpty() || num >= minPQ.peek()) {

if(minPQ.size() > maxPQ.size()) {

maxPQ.offer(minPQ.poll());

}

minPQ.offer(num);

} else if(maxPQ.isEmpty() || num <= maxPQ.peek()) {

if(maxPQ.size() > minPQ.size()) {

minPQ.offer(maxPQ.poll());

}

maxPQ.offer(num);

} else {

if(minPQ.size() <= maxPQ.size()) {

minPQ.offer(num);

} else {

maxPQ.offer(num);

}

}

}

public double findMedian() {

if(maxPQ.isEmpty() && minPQ.isEmpty()) {

return 0;

}

if(maxPQ.size() > minPQ.size()) {

return maxPQ.peek();

} else if(maxPQ.size() < minPQ.size()) {

return minPQ.peek();

} else {

return ((double)maxPQ.peek() + (double)minPQ.peek()) / 2;

}

}

}

/\*\*

\* Your MedianFinder object will be instantiated and called as such:

\* MedianFinder obj = new MedianFinder();

\* obj.addNum(num);

\* double param\_2 = obj.findMedian();

\*/

Time Complexity: O(logN)

**Step by Step example**

e.g 1 2 3 4 5 6 7

Strategy two has 10 steps to finish process while Strategy one has 9 steps to finish process, the expectation is Strategy one is faster

Strategy one: first add on maxPQ

=============================================================

1

maxPQ minPQ -> both empty, put 1 on maxPQ first, maxPQ.size() > minPQ.size(), median = 1

-------------------------------------------------------------

1 2

maxPQ minPQ -> minPQ is empty, put 2 on minPQ, maxPQ.size() == minPQ.size(), median = (1 + 2) / 2

-------------------------------------------------------------

2

1 3

maxPQ minPQ -> 3 > minPQ.peek() means it will be on large half, put on minPQ, minPQ.size() > maxPQ.size(), median = 2

-------------------------------------------------------------

<- 2

1 3

maxPQ minPQ -> 4 > minPQ.peek() means it will be on large half, but before any change already have minPQ.size() > maxPQ().size(), then before put 4 on minPQ, move 2 to maxPQ first

2 3

1 4

maxPQ minPQ -> median = (2 + 3) / 2

-------------------------------------------------------------

3

2 4

1 5

maxPQ minPQ -> 5 > minPQ.peek() means it will be on large half, put on minPQ, minPQ.size() > maxPQ.size(), median = 3

-------------------------------------------------------------

<- 3

2 4

1 5

maxPQ minPQ -> 6 > minPQ.peek() means it will be on large half, but before any change already have minPQ.size() > maxPQ.size(), then before put 6 on minPQ, move 3 to maxPQ first

3 4

2 5

1 6

maxPQ minPQ -> median = (3 + 4) / 2

-------------------------------------------------------------

4

3 5

2 6

1 7

maxPQ minPQ -> 7 > minPQ.peek() means it will be on large half, put on minPQ, minPQ.size() > maxPQ.size(), median = 4

=============================================================

Strategy two: first add on minPQ

=============================================================

1

maxPQ minPQ -> both empty, put 1 on minPQ first, minPQ.size() > maxPQ.size(), median = 1

-------------------------------------------------------------

<- 1

maxPQ minPQ -> 2 > minPQ.peek() means it will be on large half, but before any change already have minPQ.size() > maxPQ.size(), then before put 2 on minPQ, move 1 to maxPQ first

1 2

maxPQ minPQ -> median = (1 + 2) / 2

-------------------------------------------------------------

2

1 3

maxPQ minPQ -> 3 > minPQ.peek() means it will be on large half, put on minPQ, minPQ.size() > maxPQ.size(), median = 2

-------------------------------------------------------------

<- 2

1 3

maxPQ minPQ -> 4 > minPQ.peek() means it will be on large half, but before any change already have minPQ.size() > maxPQ.size(), then before put 4 on minPQ, move 2 to maxPQ first

2 3

1 4

maxPQ minPQ -> median = (2 + 3) / 2

-------------------------------------------------------------

3

2 4

1 5

maxPQ minPQ -> 5 > minPQ.peek() means it will be on large half, put on minPQ, minPQ.size() > maxPQ.size(), median = 3

-------------------------------------------------------------

<- 3

2 4

1 5

maxPQ minPQ -> 6 > minPQ.peek() means it will be on large half, but before any change already have minPQ.size() > maxPQ.size(), then before put 6 on minPQ, move 3 to maxPQ first

3 4

2 5

1 6

maxPQ minPQ -> median = (3 + 4) / 2

-------------------------------------------------------------

4

3 5

2 6

1 7

maxPQ minPQ -> 7 > minPQ.peek() means it will be on large half, put on minPQ, minPQ.size() > maxPQ.size(), median = 4

=============================================================

**Refer to**

<https://segmentfault.com/a/1190000003709954>

### 最大最小堆

#### 复杂度

时间 O(NlogN) 空间 O(N)

#### 思路

维护一个最大堆，一个最小堆。最大堆存的是到目前为止较小的那一半数，最小堆存的是到目前为止较大的那一半数，这样中位数只有可能是堆顶或者堆顶两个数的均值。而维护两个堆的技巧在于判断堆顶数和新来的数的大小关系，还有两个堆的大小关系。我们将新数加入堆后，要保证两个堆的大小之差不超过1。先判断堆顶数和新数的大小关系，有如下三种情况：

(1) 最小堆堆顶小于新数时，说明新数在所有数的上半部分。

(2) 最小堆堆顶大于新数，但最大堆堆顶小于新数时，说明新数将处在最小堆堆顶或最大堆堆顶，也就是一半的位置。

(3) 最大堆堆顶大于新数时，说明新数将处在所有数的下半部分。

再判断两个堆的大小关系，如果新数不在中间，那目标堆不大于另一个堆时，将新数加入目标堆，否则将目标堆的堆顶加入另一个堆，再把新数加入目标堆。如果新数在中间，那加到大小较小的那个堆就行了（一样大的话随便，代码中是加入最大堆）。这样，每次新加进来一个数以后，如果两个堆一样大，则中位数是两个堆顶的均值，否则中位数是较大的那个堆的堆顶。

#### 注意

Java中实现最大堆是在初始化优先队列时加入一个自定义的Comparator，默认初始堆大小是11。Comparator实现compare方法时，用arg1 - arg0来表示大的值在前面

#### 代码

class MedianFinder {

PriorityQueue<Integer> maxheap;

PriorityQueue<Integer> minheap;

public MedianFinder(){

// 新建最大堆

maxheap = new PriorityQueue<Integer>(11, new Comparator<Integer>(){

public int compare(Integer i1, Integer i2){

return i2 - i1;

}

});

// 新建最小堆

minheap = new PriorityQueue<Integer>();

}

// Adds a number into the data structure.

public void addNum(int num) {

// 如果最大堆为空，或者该数小于最大堆堆顶，则加入最大堆

if(maxheap.size() == 0 || num <= maxheap.peek()){

// 如果最大堆大小超过最小堆，则要平衡一下

if(maxheap.size() > minheap.size()){

minheap.offer(maxheap.poll());

}

maxheap.offer(num);

// 数字大于最小堆堆顶，加入最小堆的情况

} else if (minheap.size() == 0 || num > minheap.peek()){

if(minheap.size() > maxheap.size()){

maxheap.offer(minheap.poll());

}

minheap.offer(num);

// 数字在两个堆顶之间的情况

} else {

if(maxheap.size() <= minheap.size()){

maxheap.offer(num);

} else {

minheap.offer(num);

}

}

}

// Returns the median of current data stream

public double findMedian() {

// 返回大小较大的那个堆堆顶，如果大小一样说明是偶数个，则返回堆顶均值

if(maxheap.size() > minheap.size()){

return maxheap.peek();

} else if (maxheap.size() < minheap.size()){

return minheap.peek();

} else if (maxheap.isEmpty() && minheap.isEmpty()){

return 0;

} else {

return (maxheap.peek() + minheap.peek()) / 2.0;

}

}

};

简洁版

class MedianFinder {

PriorityQueue<Integer> maxheap = new PriorityQueue<Integer>();

PriorityQueue<Integer> minheap = new PriorityQueue<Integer>(Collections.reverseOrder());

// Adds a number into the data structure.

public void addNum(int num) {

maxheap.offer(num);

minheap.offer(maxheap.poll());

if(maxheap.size() < minheap.size()){

maxheap.offer(minheap.poll());

}

}

// Returns the median of current data stream

public double findMedian() {

return maxheap.size() == minheap.size() ? (double)(maxheap.peek() + minheap.peek()) / 2.0 : maxheap.peek();

}

};

### 后续 Follow Up

Q：如果要求第n/10个数字该怎么做？A：改变两个堆的大小比例，当求n/2即中位数时，两个堆是一样大的。而n/10时，说明有n/10个数小于目标数，9n/10个数大于目标数。所以我们保证最小堆是最大堆的9倍大小就行了。

**Refer to**

<https://leetcode.wang/leetcode-295-Find-Median-from-Data-Stream.html>

# **解法一**

先分享 [官方](https://leetcode.com/problems/find-median-from-data-stream/solution/) 给我们提供的两个最容易的解法。

把添加的数字放到 list 中，如果需要返回中位数，把 list 排序即可。

class MedianFinder {

List<Integer> list = new ArrayList<>();

/\*\* initialize your data structure here. \*/

public MedianFinder() {

}

public void addNum(int num) {

list.add(num);

}

public double findMedian() {

Collections.sort(list);

int n = list.size();

if ((n & 1) == 1) {

return list.get(n / 2);

} else {

return ((double) list.get(n / 2) + list.get(n / 2 - 1)) / 2;

}

}

}

简单明了。但是时间复杂度有点儿高，对于 findMedian 函数，因为每次都需要排序。如果是快速排序，那时间复杂度也是 O(nlog(n))。

这里可以做一个简单的优化。我们不需要每次返回中位数都去排序。我们可以将排序融入到 addNum 中，假设之前已经有序了，然后将添加的数字插入到相应的位置即可。也就是插入排序的思想。

class MedianFinder {

List<Integer> list = new ArrayList<>();

/\*\* initialize your data structure here. \*/

public MedianFinder() {

}

public void addNum(int num) {

int i = 0;

// 寻找第一个大于 num 的数的下标

for (; i < list.size(); i++) {

if (num < list.get(i)) {

break;

}

}

// 将当前数插入

list.add(i, num);

}

public double findMedian() {

int n = list.size();

if ((n & 1) == 1) {

return list.get(n / 2);

} else {

return ((double) list.get(n / 2) + list.get(n / 2 - 1)) / 2;

}

}

}

上边的话 findMedian() 就不需要排序了，时间复杂度就是 O(1)了。对于 addNum() 函数的话时间复杂度就是 O(n) 了。

addNum() 还可以做一点优化。因为我们要在有序数组中寻找第一个大于 num 的下标，提到有序数组找某个值，可以想到二分的方法。

public void addNum(int num) {

int insert = -1;

int low = 0;

int high = list.size() - 1;

while (low <= high) {

int mid = (low + high) >>> 1;

if (num <= list.get(mid)) {

//判断 num 是否大于等于前边的数

int pre = mid > 0 ? list.get(mid - 1) : Integer.MIN\_VALUE;

if (num >= pre) {

insert = mid;

break;

} else {

high = mid - 1;

}

} else {

low = mid + 1;

}

}

if (insert == -1) {

insert = list.size();

}

// 将当前数插入

list.add(insert, num);

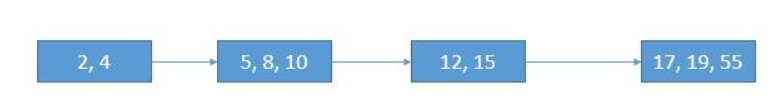
}

虽然我们使用了二分去查找要插入的位置，对应的时间复杂度是 O(log(n))，但是 list.add(insert, num) 的时间复杂度是 O(n)。所以整体上依旧是 O(n)。

参考 [这里](https://leetcode.com/problems/find-median-from-data-stream/discuss/74057/Tired-of-TWO-HEAPSET-solutions-See-this-segment-dividing-solution-(c++))，还能继续优化，这个思想也比较常见，分享一下。

我们每次添加数字的时候，都需要从所有数字中寻找要插入的位置，如果数字太多的话，速度会很慢。

我们可以将数字分成若干个子序列，类似于下图。

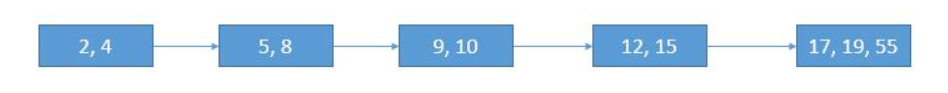


上边每一个长方形内数字都是有序的。添加数字的时候，分成两步。先找到数字应该加入的长方形，然后将数字添加到该长方形内。这样做的好处很明显，我们只需要将数字加入长方形内的有序数列中，长方形内的数字个数相对于整个序列会小很多。

我们可以设置每个长方形内最多存多少数字，如果超过了限制，就将长方形平均分成两个。

举个简单的例子，接着上图，假设每个长方形内最多存 3 个数字，现在添加数字 9。

我们首先找到 9 应该属于第 2 个长方形，然后将 9 插入。然后发现此时的数量超过了 3 个，此时我们就把该长方形平均分成两个，如下图。



至于最多存多少，我们可以根据总共有多少个数字来自己设定。但不是很好控制，太小了的话，寻找长方形的时候比较耗时间，太大的话，加入到长方形里的时候比较耗时间。事先不知道数字有多少的话，就更麻烦了。

[StefanPochmann](https://leetcode.com/stefanpochmann) 大神提出了一个建议。我们可以将大小设置成一个动态的，每个长方形最多存多少根据当前数字的总个数实时改变。假设当前数字总量是 n。长方形里数字个数是 len ，如果 len \* len > n ，那么当前长方形就分割为两个。也就是每个长方形最多存 sqrt(n) 个数字。

这里我就偷个懒了，直接分享下 [@mission4success](https://leetcode.com/mission4success) 的代码。主要就是找长方形以及找中位数那里判断的情况会多一些。

public class MedianFinder {

private LinkedList<LinkedList<Integer>> buckets; // store all ranges

private int total\_size;

MedianFinder() {

total\_size = 0;

buckets = new LinkedList<>();

buckets.add(new LinkedList<>());

}

void addNum(int num) {

List<Integer> correctRange = new LinkedList<>();

int targetIndex = 0;

// find the correct range to insert given num

for (int i = 0; i < buckets.size(); i++) {

if (buckets.size() == 1 ||

(i == 0 && num <= buckets.get(i).getLast()) ||

(i == buckets.size() - 1 && num >= buckets.get(i).getFirst()) ||

(buckets.get(i).getFirst() <= num && num <= buckets.get(i).getLast()) ||

(num > buckets.get(i).getLast() && num < buckets.get(i+1).getFirst())) {

correctRange = buckets.get(i);

targetIndex = i;

break;

}

}

// put num at back of correct range, and sort it to keep increasing sequence

total\_size++;

correctRange.add(num);

Collections.sort(correctRange);

// if currentRange's size > threshold, split it into two halves and add them back to buckets

int len = correctRange.size();

//if (len > 10) {

if (len \* len > total\_size) {

LinkedList<Integer> half1 = new LinkedList<>(correctRange.subList(0, (len) / 2));

LinkedList<Integer> half2 = new LinkedList<>(correctRange.subList((len) / 2, len));

buckets.set(targetIndex, half1); //replaces

buckets.add(targetIndex + 1, half2); //inserts

}

}

// iterate thru all ranges in buckets to find median value

double findMedian() {

if (total\_size==0)

return 0;

int mid1 = total\_size/2;

int mid2 = mid1 + 1;

int leftCount=0;

double first = 0.0, second = 0.0;

for (List<Integer> bucket : buckets) {

if (leftCount<mid1 && mid1<=leftCount+bucket.size())

first = bucket.get(mid1 - leftCount - 1);

if (leftCount<mid2 && mid2<=leftCount+bucket.size()) {

second = bucket.get(mid2 - leftCount - 1);

break;

}

leftCount += bucket.size();

}

if (total\_size % 2 != 0)

return second;

else

return (first + second)/2;

}

}

**Solution 2: BST (120 min)**

class MedianFinder {

class Node {

int val;

// 记录以当前节点为根节点的二叉树的总节点数量

int cnt;

Node left, right;

public Node(int val) {

this.val = val;

this.cnt = 1;

}

}

Node root;

public MedianFinder() {

root = null;

}

public void addNum(int num) {

Node newNode = new Node(num);

if(root == null) {

root = newNode;

return;

}

Node cur = root;

Node parent = null;

while(true) {

parent = cur;

if(num < cur.val) {

cur = cur.left;

parent.cnt++;

if(cur == null) {

parent.left = newNode;

return;

}

} else {

cur = cur.right;

parent.cnt++;

if(cur == null) {

parent.right = newNode;

return;

}

}

}

}

public double findMedian() {

int total\_cnt = root.cnt;

if(total\_cnt % 2 == 0) {

return ((double)find(total\_cnt / 2) + find(total\_cnt / 2 - 1)) / 2;

} else {

return (double)find(total\_cnt / 2);

}

}

// 可以借助二分查找树，二分查找树的性质。

// 若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

// 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；

// 任意节点的左、右子树也分别为二叉查找树；

// 没有键值相等的节点。

// 如果我们将数字存到二分查找树中，当找中位数的时候有一个明显的好处。

// 如果我们知道了左子树的数量 leftCnt ，找假设把数据排序后的第 k 个数，k 从 0 计数。

// (1) 如果 leftCnt == k ，那么根节点就是我们要找的。

// (2) 如果 leftCnt > k，我们只需要再从左子树中找第 k 个数。

// (3) 如果 leftCnt < k，我们只需要从右子树中找第 k - leftCnt - 1 个数。

public int find(int k) {

Node t = root;

while(true) {

int leftCnt = t.left != null ? t.left.cnt : 0;

if(leftCnt == k) {

return t.val;

}

if(leftCnt > k) {

t = t.left;

} else {

k = k - leftCnt - 1;

t = t.right;

}

}

}

}

/\*\*

\* Your MedianFinder object will be instantiated and called as such:

\* MedianFinder obj = new MedianFinder();

\* obj.addNum(num);

\* double param\_2 = obj.findMedian();

\*/

Time Complexity: O(logN) ~ O(N)

时间复杂度的话，最好的情况 addNum 和 findMedian 都是 O(log(n))。但如果二叉树分布不均，类似于下边这种，那么时间复杂度就都是 O(n) 了。

**Refer to**

<https://leetcode.wang/leetcode-295-Find-Median-from-Data-Stream.html>

# **解法二**

分享 [这里](https://leetcode.com/problems/find-median-from-data-stream/discuss/74166/Solution-using-Binary-Search-Tree) 的解法。

可以借助二分查找树，二分查找树的性质。

若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；

任意节点的左、右子树也分别为二叉查找树；

没有键值相等的节点。

如果我们将数字存到二分查找树中，当找中位数的时候有一个明显的好处。如果我们知道了左子树的数量 leftNum ，找假设把数据排序后的第 k 个数，k 从 0 计数。

如果 leftNum == k ，那么根节点就是我们要找的。

如果 leftNum > k，我们只需要再从左子树中找第 k 个数。

如果 leftNum < k，我们只需要从右子树中找第 k - leftNum - 1 个数。

代码的话，我们首先定义一个二分查找树。和普通的二分查找树不同的地方在于，节点多了一个成员变量，记录以当前节点为根节点的二叉树的总节点数量。

此外实现了 find 函数，来返回有序情况下第 k 个节点的值。

class BST {

class Node {

int val;

int size;

Node left, right;

Node(int v) {

val = v;

size = 1;

};

};

private Node root;

BST() {

}

public void add(int val) {

// 新增节点

Node newNode = new Node(val);

// 当前节点

Node current = root;

// 上个节点

Node parent = null;

// 如果根节点为空

if (current == null) {

root = newNode;

return;

}

while (true) {

parent = current;

//向左子树添加节点

if (val < current.val) {

current = current.left;

parent.size++;

if (current == null) {

parent.left = newNode;

return;

}

//向右子树添加节点

} else {

current = current.right;

parent.size++;

if (current == null) {

parent.right = newNode;

return;

}

}

}

}

public int find(int k) {

Node t = root;

while (true) {

int leftSize = t.left != null ? t.left.size : 0;

if (leftSize == k)

return t.val;

if (leftSize > k) {

t = t.left;

} else {

k = k - leftSize - 1;

t = t.right;

}

}

}

public int size() {

return root.size;

}

};

class MedianFinder {

BST bst;

MedianFinder() {

bst = new BST();

}

// Adds a number into the data structure.

public void addNum(int num) {

bst.add(num);

}

public double findMedian() {

int num = bst.size();

if (num % 2 == 0) {

return ((double)bst.find(num / 2) + bst.find(num / 2 - 1)) / 2;

} else {

return bst.find(num / 2);

}

}

};

**时间复杂度的话，最好的情况 addNum 和 findMedian 都是 O(log(n))。但如果二叉树分布不均，类似于下边这种，那么时间复杂度就都是 O(n) 了。**

1

\

2

\

3

\

4

# **扩展(本质上是bucket sort)**

If all integer numbers from the stream are between 0 and 100, how would you optimize it?

<https://leetcode.com/problems/find-median-from-data-stream/solutions/286238/Java-Simple-Code-Follow-Up/>

If the range of the numbers is in [0...100], we use a bucket to collect the frequency of each number. By accumulating the frequency of elements in the bucket, we can know the median numbers.

class MedianFinder {

int A[] = new int[101], n = 0;

// O(1)

public void addNum(int num) {

A[num]++;

n++;

}

// O(100) = O(1)

public double findMedian() {

// find 1st median number

int count = 0, i = 0;

while (count < n/2) count += A[i++];

// find 2nd median number

int j = i;

while (count < n/2+1) count += A[j++];

return (n%2 == 1) ? i : (i-1+j-1) / 2.0;

}

}

If 1% numbers are outside of the range [0...100], we know that when the set of numbers is large, the median numbers must be in the range of [0...100], because this range contains 99% numbers. We don't need to store values of 1% numbers, but the counts of these numbers (countLessZero & countGreater100). The findMedian method is almost the same, the difference is we start counting from countLessZero value

class MedianFinder {

int A[] = new int[101], n = 0;

int countLessZero = 0;

// int countGreater100 = 0; // not needed

// O(1)

public void addNum(int num) {

if (num < 0) countLessZero++;

// else if (num > 100) countGreater100++;

else A[num]++;

n++;

}

// O(100) = O(1)

public double findMedian() {

// find 1st median number

int count = countLessZero, i = 0;

while (count < /2) count += A[i++];

// find 2nd median number

int j = i;

while (count < n/2+1) count += A[j++];

return (n%2 == 1) ? i : (i-1+j-1) / 2.0;

}

}

Insert - O(1), Find O(1), Space Complexity O(1)

<https://leetcode.wang/leetcode-295-Find-Median-from-Data-Stream.html>

这样的话，我们可以用一个数组，num[i]记录数字 i 的数量。此外用一个变量 n 统计当前数字的总数量。这样求中位数的时候，我们只需要找到第 n/2+1个数或者 n/2,n/2+1个数即可。注意因为这里计数是从1 开始的，所以和解法一看起来找到数字不一样，解法一找的是下标。

# **再扩展**

题目说的是从数据流中找到中位数。如果这个数据流很大，很大，无法全部加载到内存呢？

分享 [这里](https://stackoverflow.com/questions/10657503/find-running-median-from-a-stream-of-integers/10693752" \l "10693752) 的想法。

如果数据整体呈某种概率分布，比如正态分布。我们可以通过 reservoir sampling 的方法。我们保存固定数量的数字，当存满的时候，就随机替代掉某个数字。伪代码如下

int n = 0; // Running count of elements observed so far

#define SIZE 10000

int reservoir[SIZE];

while(streamHasData())

{

int x = readNumberFromStream();

if (n < SIZE)

{

reservoir[n++] = x;

}

else

{

int p = random(++n); // Choose a random number 0 >= p < n

if (p < SIZE)

{

reservoir[p] = x;

}

}

}

相当于从一个大的数据集下进行了取样，然后找中位数的时候，把我们保存的数组排个序去找就可以了。