

# 164. Maximum Gap

Jan. 15, 2017 | 46.8K views

Average Rating: 4.81 (62 votes)

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Return 0 if the array contains less than 2 elements.

## Example 1:

**Input:** [3,6,9,1]  
**Output:** 3  
**Explanation:** The sorted form of the array is [1,3,6,9], either (3,6) or (6,9) has the maximum difference 3.

## Example 2:

**Input:** [10]  
**Output:** 0  
**Explanation:** The array contains less than 2 elements, therefore return 0.

## Note:

- You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.
- Try to solve it in linear time/space.

## Solution

### Approach 1: Comparison Sorting

#### Intuition

Do what the question says.

#### Algorithm

Sort the entire array. Then iterate over it to find the maximum gap between two successive elements.

```
C++
1 int maximumGap(vector<int>& nums)
2 {
3     if (nums.empty() || nums.size() < 2) // check if array is empty or small sized
4         return 0;
5     sort(nums.begin(), nums.end()); // sort the array
6
7     int maxGap = 0;
8
9     for (int i = 0; i < nums.size() - 1; i++)
10         maxGap = max(nums[i + 1] - nums[i], maxGap);
11
12     return maxGap;
13 }
14
```

#### Complexity Analysis

- Time complexity:  $O(n \log n)$ .  
Time taken to sort the array is  $O(n \log n)$  (average case). Time taken for linear iteration through the array is of  $O(n)$  complexity. Hence overall time complexity is  $O(n \log n)$ .

- Space complexity: No extra space needed, other than the input array (since sorting can usually be done in-place).

### Approach 2: Radix Sort

#### Algorithm

This approach is similar to [Approach 1](#), except we use [Radix Sort](#) instead of a traditional comparison sort.

```
C++
1 int maximumGap(vector<int>& nums)
2 {
3     if (nums.empty() || nums.size() < 2)
4         return 0;
5
6     int maxVal = *max_element(nums.begin(), nums.end());
7
8     int exp = 1; // 1, 10, 100, 1000 ...
9     int radix = 10; // base 10 system
10
11     vector<int> aux(nums.size());
12
13     // LSD Radix Sort */
14     while (maxVal / exp > 0) { // Go through all digits from LSD to MSD
15         vector<int> count(radix, 0);
16
17         for (int i = 0; i < nums.size(); i++) // Counting sort
18             count[(nums[i] / exp) % 10]++;
19
20         for (int i = 1; i < count.size(); i++) // you could also use partial_sum()
21             count[i] += count[i - 1];
22
23         for (int i = nums.size() - 1; i >= 0; i--)
24             aux[--count[(nums[i] / exp) % 10]] = nums[i];
25
26         for (int i = 0; i < nums.size(); i++)
27             nums[i] = aux[i];
28     }
29 }
```

#### Complexity Analysis

- Time complexity:  $O(d \cdot (n + k)) \approx O(n)$ .

Since a linear iteration over the array (once it is sorted) is of linear (i.e.  $O(n)$ ) complexity, the performance of this approach is limited by the performance of Radix sort.

Radix sort uses [Counting sort](#) as a subroutine.

- Counting sort runs in  $O(n + k)$  time (where  $k$  is the radix or base of the digits comprising the  $n$  elements in the array). If  $k \leq O(n)$ , Counting sort would run in linear time. In our case, the radix is fixed (i.e.  $k = 10$ ). Hence our Counting sort subroutine runs in  $O(n)$  linear time.
- Radix sort works by running  $d$  passes of the Counting sort subroutine (where the elements are composed of, maximally,  $d$  digits). Hence effective runtime of Radix sort would be  $O(d \cdot (n + k))$ . However, in our case an element can, maximally, be the maximum 32-bit signed integer [2,147,483,647](#). Hence  $d \leq 10$  is a constant.

Thus Radix sort has a runtime performance complexity of about  $O(n)$  for reasonably large input.

- Space complexity:  $O(n + k) \approx O(n)$  extra space.

Counting sort requires  $O(k)$  extra space. Radix sort requires an auxiliary array of the same size as input array. However given that  $k$  is a small fixed constant, the space required by Counting sort can be ignored for reasonably large input.

### Approach 3: Buckets and The Pigeonhole Principle

#### Intuition

Sorting an entire array can be costly. At worst, it requires comparing each element with every other element. What if we didn't need to compare all pairs of elements? That would be possible if we could somehow divide the elements into representative groups, or rather, *buckets*. Then we would only need to compare these buckets.

**Digression: The Pigeonhole Principle** The [Pigeonhole Principle](#) states that if  $n$  items are put into  $m$  containers, with  $n > m$ , then at least one container must contain more than one item.

Suppose for each of the  $n$  elements in our array, there was a bucket. Then each element would occupy one bucket. Now what if we reduced, the number of buckets? Some buckets would have to accommodate more than one element.

Now let's talk about the gaps between the elements. Let's take the best case, where all elements of the array are sorted and have a uniform gap between them. This means every adjacent pair of elements differ by the same constant value. So for  $n$  elements of the array, there are  $n - 1$  gaps, each of width, say,  $t$ . It is trivial to deduce that  $t = (max - min) / (n - 1)$  (where  $max$  and  $min$  are the minimum and maximum elements of the array). This width is the maximal width/gap between two adjacent elements in the array; precisely the quantity we are looking for!

One can safely argue that this value of  $t$ , is in fact, the smallest value that  $t$  can ever accomplish of any array with the same number of elements (i.e.  $n$ ) and the same range (i.e.  $(max - min)$ ). To test this fact, you can start with a uniform width array (as described above) and try to reduce the gap between any two adjacent elements. If you reduce the gap between  $arr[i - 1]$  and  $arr[i]$  to some value  $t - p$ , then you will notice that the gap between  $arr[i]$  and  $arr[i + 1]$  would have increased to  $t + p$ . Hence the maximum attainable gap would have become  $t + p$  from  $t$ . Thus the value of the **maximum gap**  $t$  can only increase.

#### Buckets!

Coming back to our problem, we have already established by application of the Pigeonhole Principle, that if we used *buckets* instead of individual elements as our base for comparison, the number of comparisons would reduce if we could accommodate more than one element in a single bucket. That does not immediately solve the problem though. What if we had to compare elements *within* a bucket? We would end up no better.

So the current motivation remains: somehow, if we only had to compare among the buckets, and *not* the elements *within* the buckets, we would be good. It would also solve our sorting problem: we would just distribute the elements to the right buckets. Since the buckets can be already ordered, and we only compare among buckets, we wouldn't have to compare all elements to sort them!

But if we only had buckets to compare, we would have to *ensure*, that the gap between the buckets itself represent the maximal gap in the input array. How do we go about doing that?

We could do that just by setting the buckets to be smaller than  $t = (max - min) / (n - 1)$  (as described above). Since the gaps (between elements) within the same bucket would only be  $\leq t$ , we could deduce that the maximal gap would *indeed occur only between two adjacent buckets*.

Hence by setting bucket size  $b$  to be  $1 < b \leq (max - min) / (n - 1)$ , we can ensure that at least one of the gaps between adjacent buckets would serve as the **maximal gap**.

#### Clarifications

A few clarifications are in order:

- Would the buckets be of uniform size?** Yes. Each of them would be of the same size  $b$ .
- But, then wouldn't the gap between them be uniform/constant as well?** Yes it would be. The gap between them would be 1 integer unit wide. That means a two adjacent buckets of size 3 could hold integers with values, say, 3 - 6 and 7 - 9. We avoid overlapping buckets.
- Then what are you talking about when you say the gap between two adjacent buckets could be the maximal gap?** When we are talking about the size of a bucket, we are talking about its holding capacity. That is the range of values the bucket can represent (or *hold*). However the actual extent of the bucket are determined by the values of the maximum and minimum element a bucket holds. For example a bucket of size 5 could have a capacity to hold values between 6 - 10. However, if it only holds the elements 7, 8 and 9, then its actual extent is only  $(9 - 7) + 1 = 3$  which is not the same as the capacity of the bucket.
- Then how do you compare adjacent buckets?** We do that by comparing their extents. Thus we compare the minimum element of the next bucket to the maximum element of the current bucket. For example: if we have two buckets of size 5 each, holding elements [1, 2, 3] and [9, 10] respectively, then the gap between the buckets would essentially refer to the value  $9 - 3 = 6$  (which is larger than the size of either bucket).
- But then aren't we comparing elements again?!** We are, yes! But only compare about twice the elements as the number of buckets (i.e. the minimum and maximum elements of each bucket). If you followed the above, you would realize that this amount is certainly less than the actual number of elements in the array, given a suitable bucket size was chosen.

#### Algorithm

- We choose a bucket size  $b$  such that  $1 < b \leq (max - min) / (n - 1)$ . Let's just choose  $b = \lfloor (max - min) / (n - 1) \rfloor$ .
- Thus all the  $n$  elements would be divided among  $k = \lceil (max - min) / b \rceil$  buckets.
- Hence the  $i^{th}$  bucket would hold the range of values:  $\left[ min + (i - 1) * b, min + i * b \right)$  (1-based indexing).
- It is trivial to calculate the index of the bucket to which a particular element belongs. That is given by  $\lfloor (num - min) / b \rfloor$  (0-based indexing) where  $num$  is the element in question.
- Once all  $n$  elements have been distributed, we compare  $k - 1$  adjacent bucket pairs to find the maximum gap.

```
C++
1 class Bucket {
2 public:
3     bool used = false;
4     int minVal = numeric_limits<int>::max(); // same as INT_MAX
5     int maxVal = numeric_limits<int>::min(); // same as INT_MIN
6 };
7
8 int maximumGap(vector<int>& nums)
9 {
10     if (nums.empty() || nums.size() < 2)
11         return 0;
12
13     int mini = *min_element(nums.begin(), nums.end());
14     int maxi = *max_element(nums.begin(), nums.end());
15
16     int bucketSize = max(1, (maxi - mini) / ((int)nums.size() - 1)); // bucket size or capacity
17     int bucketNum = (maxi - mini) / bucketSize + 1; // number of buckets
18     vector<Bucket> buckets(bucketNum);
19
20     for (auto& num : nums) {
21         int bucketIdx = (num - mini) / bucketSize; // locating correct bucket
22         buckets[bucketIdx].used = true;
23         buckets[bucketIdx].minVal = min(num, buckets[bucketIdx].minVal);
24         buckets[bucketIdx].maxVal = max(num, buckets[bucketIdx].maxVal);
25     }
26
27     int prevBucketMax = mini, maxGap = 0;
28 }
```

#### Complexity Analysis

- Time complexity:  $O(n + b) \approx O(n)$ .

Distributing the elements of the array takes one linear pass (i.e.  $O(n)$  complexity). Finding the maximum gap among the buckets takes a linear pass over the bucket storage (i.e.  $O(b)$  complexity). Hence overall process takes linear runtime.

- Space complexity:  $O(2 \cdot b) \approx O(b)$  extra space.

Each bucket stores a maximum and a minimum element. Hence extra space linear to the number of buckets is required.

Rate this article: ★★★★★

Previous

Next

Comments: 25


Sort By ▾

- 🔁

Type comment here... (Markdown is supported)

PreviewPost
- whatsername★37🕒 September 3, 2018 5:19 AM

The explanation for using buckets is amazing and really helpful! Thx

26👍👎🔗 Share🗨 Reply
- windiang★1004🕒 November 16, 2019 12:42 PM

详细总结一下，分享一下，<https://leetcode.wang/leetcode-164-Maximum-Gap.html>

9👍👎🔗 Share🗨 Reply
- anmingyu11★425🕒 March 12, 2019 8:26 PM

This radix sort from princeton Alg4 is truly faster than this:

```
private void lsd(int[] a) {
    final int n = a.length;
    for (int i = n - 1; i >= 0; i--) {
        // ...
    }
}
```


6👍👎🔗 Share🗨 Reply
- Saurabh JITM★4🕒 October 14, 2017 6:17 PM

Solution with bucketing: factor 'b' is not linear to 'n' because (max-min) can be arbitrarily large. Scanning all such buckets is not linear to n.

4👍👎🔗 Share🗨 Reply

SHOW 1 REPLY
- 01ankith1998★2🕒 January 15, 2020 4:41 PM

if you guys have any doubt go this article nice explanation <http://www.rzshahid.com/the-maximum-gap-problem-pigeonhole-principle/>


2👍👎🔗 Share🗨 Reply
- denis.yaroshevskiy★2🕒 February 28, 2018 3:41 AM

Nice to see modern c++ in solution) Couple of nits:  
1): [http://en.cppreference.com/w/cpp/algorithm/minmax\\_element](http://en.cppreference.com/w/cpp/algorithm/minmax_element)  
2) There are different opinions, but generally speaking auto&& is for very templated code. It's for using with forwarding references etc. Your case perfectly fine with const auto&.

2👍👎🔗 Share🗨 Reply
- leetcodeaccount★2🕒 April 20, 2017 1:45 PM

Could you explain why the buckets number  $(maxi - mini) / bucketSize + 1$  and  $\lfloor (max-min) / b \rfloor$  are the same thing?

1👍👎🔗 Share🗨 Reply

SHOW 3 REPLIES
- sspectre★1🕒 August 6, 2018 11:59 PM

In Radix Sort, I believe the "% 10" should be "% radix" (lines 18, 24).

1👍👎🔗 Share🗨 Reply
- TusharBorchate★1🕒 September 19, 2017 2:20 PM

Hi LeetCodeers,

Submitting my solution in python

```
class Solution(object):
    def maximumGap(self, nums):
        # ...

```

1👍👎🔗 Share🗨 Reply
- Brooky★25🕒 August 18, 2018 6:59 AM

how about priority\_queue? The time complexity will be nlog(n)?

1👍👎🔗 Share🗨 Reply

SHOW 1 REPLY