Previous Next

462. Minimum Moves to Equal Array Elements II

March 16, 2017 | 5.5K views

Given a **non-empty** integer array, find the minimum number of moves required to make all array elements equal, where a move is incrementing a selected element by 1 or decrementing a selected element by 1.

You may assume the array's length is at most 10,000.

Example:

```
Input:
[1,2,3]

Output:
2

Explanation:
Only two moves are needed (remember each move increments or decrements one element):

[1,2,3]  =>  [2,2,3]  =>  [2,2,2]
```

Solution

Approach 1: Brute Force

In the brute force approach, we consider every possible number to which all the array elements should be equated so as to minimize the number of moves required. One point is obvious that the number to which all the elements are equated at the end should lie between the minimum and the maximum elements present in the array. Thus, we first find the minimum and the maximum element in the array. Suppose k is the number to which all the elements are equated. Then, we iterate k over the range between the minimum and maximum values and find the number of moves required for each k, simultaneously finding the minimum moves, which will be the end result.

Java

```java
public class Solution {
    public int minMoves2(int[] nums) {
        long ans = Long.MAX_VALUE;
        int minval = Integer.MAX_VALUE;
        int maxval = Integer.MIN_VALUE;
        for (int num : nums) {
            minval = Math.min(minval, num);
            maxval = Math.max(maxval, num);
        }
        for (int i = minval; i <= maxval; i++) {
            long sum = 0;
            for (int num : nums) {
                sum += Math.abs(num - i);
            }
            ans = Math.min(ans, sum);
        }
        return (int) ans;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \cdot \text{diff})$, where n is the length of the array and diff is the difference between maximum element and minimum element.
- Space complexity : $O(1)$. No extra space required.

Approach 2: Better Brute Force

Algorithm

In this approach, rather than choosing every possible k between the minimum and the maximum values in the array, we can simply consider k as every element of the array. To understand why we need not iterate over all the complete range but only the elements of the array, consider the following example.

Say the array is:

$mums = [x_1 x_2 x_3 x_4 x_5 x_6 x_7]$. Now, if we try to equalize all the elements to x_4, which by the way, may or may not be the final number required to be settled down to.

The total number of moves for doing this is given by: $moves_1 = (x_4 - x_1) + (x_4 - x_2) + (x_4 - x_3) + (x_5 - x_4) + (x_6 - x_4) + (x_7 - x_4)$

Suppose, now, instead of x_4, we try to equalize all the elements to a number x', which is not present in the given array, but is slightly larger than x_4 and is thus given by say $x' = x_4 + \delta x$, where δx is an integer. Thus, the total number of moves required now will be given by:

$moves_2 = (x' - x_1) + (x' - x_2) + (x' - x_3) + (x' - x_4) + (x_5 - x') + (x_6 - x') + (x_7 - x')$

$moves_2 = ((x_4 + \delta x) - x_1) + ((x_4 + \delta x) - x_2) + ((x_4 + \delta x) - x_3) + ((x_4 + \delta x) - x_4) + (x_5 - (x_4 + \delta x)) + (x_6 - (x_4 + \delta x)) + (x_7 - (x_4 + \delta x))$

$moves_2 = (x_4 - x_1) + \delta x + (x_4 - x_2) + \delta x + (x_4 - x_3) + \delta x + 0 + \delta x + (x_5 - x_4) - \delta x + (x_6 - x_4) - \delta x + (x_7 - x_4) - \delta x$

$moves_2 = (x_4 - x_1) + (x_4 - x_2) + (x_4 - x_3) + (x_5 - x_4) + (x_6 - x_4) + (x_7 - x_4) + 4\delta x - 3\delta x$

$moves_2 = moves_1 + \delta x$ ...using $moves_1$ from above

From this equation, it is clear that the number of moves required to settle to some arbitrary number present in the array x_4 is always lesser than the number of moves required to settle down to some arbitrary number $x' = x_4 + \delta x$. This completes the proof.

Java

```java
public class Solution {
    public int minMoves2(int[] nums) {
        long min = Integer.MAX_VALUE;
        for (int num : nums) {
            long sum = 0;
            for (int n : nums) {
                sum += Math.abs(n - num);
            }
            min = Math.min(min, sum);
        }
        return (int) min;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two nested loops are there.
- Space complexity : $O(1)$. No extra space required.

Approach 3: Using Sorting

Algorithm

In the previous approach, we needed to find the number of moves required for every k chosen from the array, by iterating over the whole array. We can optimize this approach to sum extent by sorting the array and observing the following fact. The number of moves required to raise the elements smaller than k to equalize them to k will be given by: $(k * countBefore_k) - (sumBefore_k)$(The meanings of the keywords are given below) . Similarly, the number of moves required to decrement the elements larger than k to equalize them to k will be: $(sumAfter_k) - (k * countAfter_k)$. The total number of moves required will, thus, be the sum of these two parts. Hence, for a particular k chosen, the total number of moves required will be given by:

$numberOfMoves_k = [(k * countBefore_k) - (sumBefore_k)] + [(sumAfter_k) - (k * countAfter_k)]$

where, k = The number to which all the elements are equalized at the end.

$countBefore_k$ = The number of elements which are lesser than k.

$sumBefore_k$ = The sum of elements which are lesser than k.

$countAfter_k$ = The number of elements which are larger than k.

$sumAfter_k$ = The sum of elements which are larger than k.

$numberOfMoves_k$ = The total number of moves required to equalize all the elements of the array to k.

Let's say that the index of the element corresponding to the element k be given by $index_k$. Instead of iterating over the array for calculating $sumBefore_k$ and $sumAfter_k$, we can keep on calculating them while traversing the array since the array is sorted. We calculate the total sum of the given array $nums$ once, given by $total$. We start by choosing $sumBefore_k = 0$ and $sumAfter_k$ as $total$. To calculate $sumBefore_k$, we just add the element $nums[index_k - 1]$ to the previous $sumBefore_k$. To calculate $sumAfter_k$, we subtract the element k from the previous $sumAfter_k$.

Java

```java
public class Solution {
    public int minMoves2(int[] nums) {
        Arrays.sort(nums);
        long min = Long.MAX_VALUE, sum = 0, total = 0;
        for (int num : nums) {
            total += num;
        }
        for (int i = 0; i < nums.length; i++) {
            long ans = ((long) nums[i] * i - sum) + ((total - sum) - (long) nums[i] * (nums.length - i));
            System.out.println(nums[i] + " " + ans);
            min = Math.min(min, ans);
            sum += nums[i];
        }
        return (int) min;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \log n)$. Sorting will take $O(n \log n)$ time.
- Space complexity : $O(1)$. No extra space required.

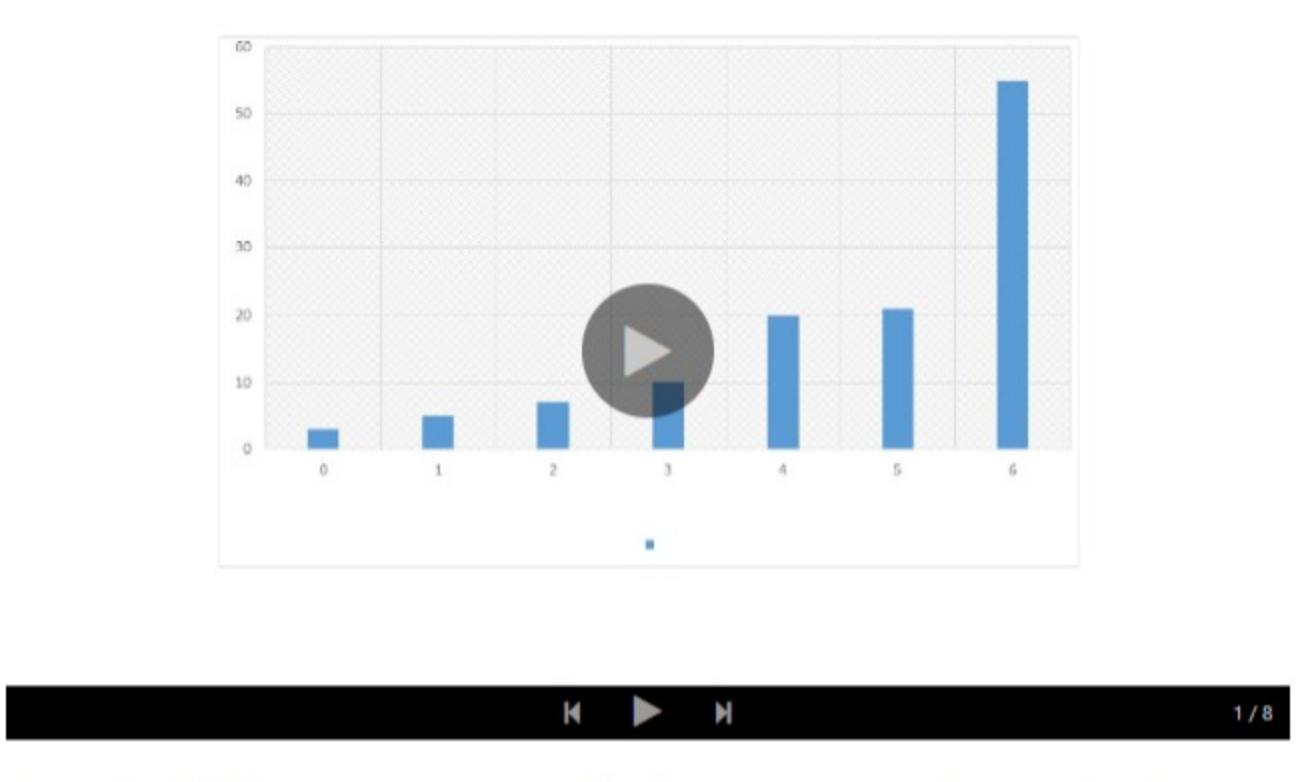
Approach 4: Using Median and Sorting

Algorithm

The problem of finding the number k to which all the other numbers eventually settle can also be viewed as: Given a set of points in 1-d. Find a point k such that the cumulative sum of distances between k and the rest of the points is minimum. This is a very common mathematical problem whose answer is known. The point k is the median of the given points. The reason behind choosing the median is given after the algorithm.

We can simply sort the given points and find the $median$ as the element in the middle of the array. Thus, the total number of moves required to equalize all the array elements is given by the sum of differences of all the elements from the $median$. In mathematical terms, the solution is given by:

$moves = \sum_{i=0}^{n-1} |median - nums[i]|$, where n is the size of the given array.



Now, we'll look at the mathematical reasoning behind choosing the median as the number k to which we'll settle. As discussed in the previous approach, the total number of moves required is given by:

$numberOfMoves_k = [(k * countBefore_k) - (sumBefore_k)] + [(sumAfter_k) - (k * countAfter_k)]$, where all the variables have the same definition.

Now, as we know, in order to maximize this term w.r.t. the changes in k, we can take the derivative of the above term w.r.t. k. Thus, we proceed as:

$\frac{d(numberOfMoves_k)}{dk} = \frac{[(k*countBefore_k)-(sumBefore_k)]+[(sumAfter_k)-(k*countAfter_k)]}{dk}$

$\frac{d(numberOfMoves_k)}{dk} = \frac{(k*countBefore_k)}{dk} - \frac{d(sumBefore_k)}{dk} + \frac{d(sumAfter_k)}{dk} - \frac{(k*countAfter_k)}{dk}$

$\frac{d(numberOfMoves_k)}{dk} = countBefore_k - countAfter_k$

Setting derivative $\frac{d(numberOfMoves_k)}{dk}$ equal to 0, we get:

$countBefore_k - countAfter_k = 0$ or $countBefore_k = countAfter_k$. This property is satisfied by the median only, which completes the proof.

Java

```java
public class Solution {
    public int minMoves2(int[] nums) {
        Arrays.sort(nums);
        int sum = 0;
        for (int num : nums) {
            sum += Math.abs(nums[nums.length / 2] - num);
        }
        return sum;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \log n)$. Sorting will take $O(n \log n)$ time.
- Space complexity : $O(1)$. Only single extra variable is used.

Approach 5: Without Finding Median

Algorithm

In the previous approach, we went for finding the median after sorting and then calculated the number of moves required. But, if we observe properly, we'll find that if the array is sorted, we can do the same task without actually finding the median or the number k to which we need to settle at the end. To proceed with this, let's look at the maximum(max) and the minimum numbers(min) in the array, which currently lie at its extreme positions. We know, at the end, both these numbers should be equalized to k. For the number max , the number of moves required to do this is given by $max - k$. Similarly, for the number min, the number of moves is given by $k - min$. Thus, the total number of moves for both max and min is given by $max - k + (k - min) = max - min$, which is independent of the number k. Thus, we can continue now, with the next maximum and the next minimum number in the array, until the complete array is exhausted.

Therefore, the equation becomes:

$moves = \sum_{i=0}^{\lceil \frac{n}{2} \rceil - 1} |nums[n-i] - nums[i]|$, where n is the number of elements in the array $nums$.

Java

```java
public class Solution {
    public int minMoves2(int[] nums) {
        int l = 0, r = nums.length - 1, sum = 0;
        Arrays.sort(nums);
        while (l < r) {
            sum += nums[r] - nums[l];
            l++;
            r--;
        }
        return sum;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \log n)$. Sorting will take $O(n \log n)$ time.
- Space complexity : $O(1)$. No extra space required.

Approach 6: Using Quick-Select

Algorithm

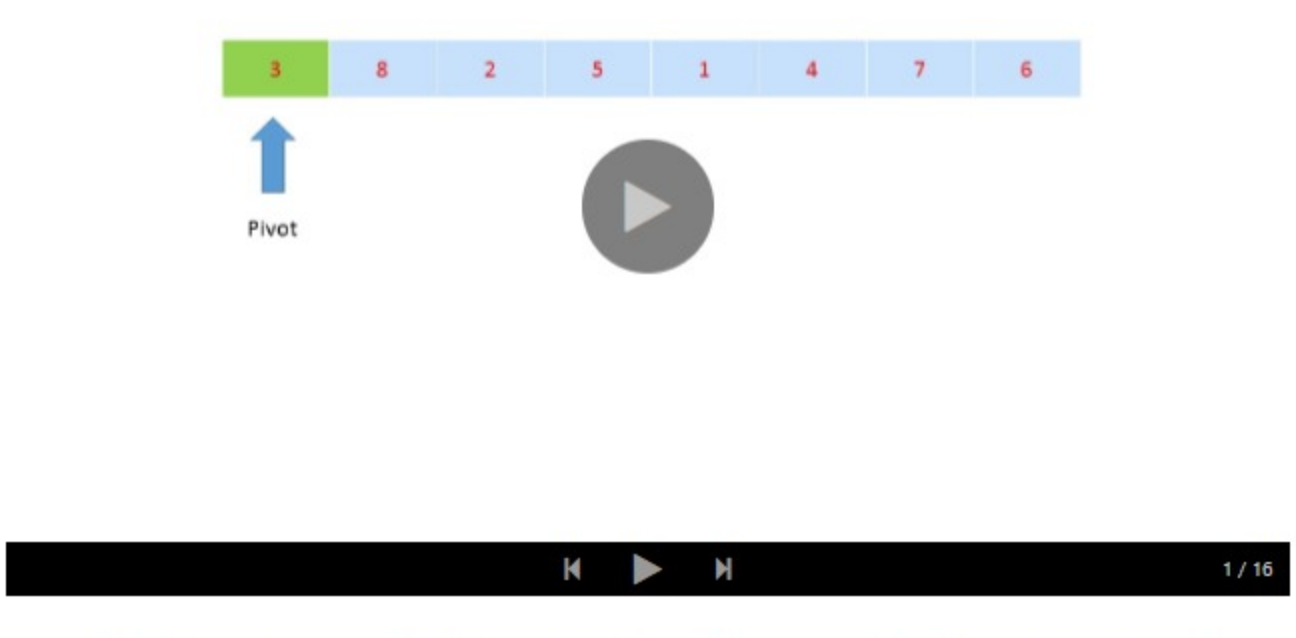
In order to find the median, we need not necessarily sort the given array. But we can find the median directly using the Quick-Select method to find the median, which doesn't use sorting.

The quick-select method is similar to the Quick-Sort method. In a single iteration, we choose a pivot and somehow bring it to its correct position in the array. If the correct position happens to be the central position(corresponding to the median), we can return the median directly from there. Now, let's look at the implementation of quick-select.

Quick-Select makes use of two functions $partition$ and $select$. $select$ function takes the leftmost and the rightmost indices of the given array and the central index as well. If the element reaching the correct position in the current function call to $select$ function happens to be the median(i.e. it reaches the central position), we return the element(since it is the median). The function $partition$ takes the leftmost and the rightmost indices of the array and returns the correct position of the current pivot(which is chosen as the rightmost element of the array). This function makes use of two pointers i and j. Both the pointers initially point to the leftmost element of the array.

At every step, we compare the element at the j^{th} index($list[j]$) with the pivot element($pivot$). If $list[j]$, we swap the elements $list[i]$ and $list[j]$ and increment i and j. Otherwise, only j is incremented. When j reaches the end of the array, we swap the $pivot$ with $list[i]$. In this way, now, all the elements lesser than $pivot$ lie to the left of the i^{th} index, and all the elements larger than $pivot$ lie to the right of the i^{th} index and thus, the $pivot$ reaches at its correct position in the array. If this position isn't the central index of the array, we again make use of the $select$ functions passing the left and the right subarrays relative to the i^{th} index.

For more clarification, look at the animation below for this example: `[3 8 2 5 1 4 7 6]`



After finding the median, we can find the sum of absolute differences of all the elements from the median to determine the number of moves required. Mathematically, we use:

$moves = \sum_{i=0}^{n-1} |median - list[i]|$, where n is the size of the given array.

Java

```java
public class Solution {
    public void swap(int[] list, int i, int pivot_index) {
        int temp = list[i];
        list[i] = list[pivot_index];
        list[pivot_index] = temp;
    }
    public int partition(int[] list, int left, int right) {
        int pivotValue = list[right];
        int i = left;
        for (int j = left; j <= right; j++) {
            if (list[j] < pivotValue) {
                swap(list, i, j);
                i++;
            }
        }
        swap(list, right, i);
        return i;
    }
    int select(int[] list, int left, int right, int k) {
        if (left == right) {
            return list[left];
        }
        int pivotIndex = partition(list, left, right);
        if (k == pivotIndex) {
            return list[k];
        } else if (k < pivotIndex) {
            return select(list, left, pivotIndex - 1, k);
```

Complexity Analysis

- Time complexity : Average Case: $O(n)$. Quick-Select average case time complexity is $O(n)$. Worst Case: $O(n^2)$. In worst case quick-select can go upto n^2
- Space complexity : $O(1)$. No extra space required.

Approach 7: Using Median of Medians

Algorithm

It isn't hard to see that, in quick-select, if we naively choose the pivot element, this algorithm has a worst case performance of $O(n^2)$. To guarantee the linear running time in order to find the median, however we need a strategy for choosing the pivot element that guarantees that we partition the list into two sublists of relatively comparable size. Obviously the median of the values in the list would be the optimal choice, but if we could find the median in linear time, we would already have a solution to our problem.

The median-of-medians algorithm chooses its pivot in the following clever way:

$kthSmallest(arr[0..n-1], k)$

1. Divide $arr[]$ into $\lceil \frac{n}{5} \rceil$ groups where size of each group is 5 elements, except possibly the last group which may have less than 5 elements.
2. Sort the above created $\lceil \frac{n}{5} \rceil$ groups and find median of all groups. Create an auxiliary array $median[]$ and store medians of all $\lceil \frac{n}{5} \rceil$ groups in this median array. Also, recursively call this method to find median of median[0...($\lceil \frac{n}{5} \rceil - 1$)]
3. $\text{medOfMed} = kthSmallest(median[0...(\lceil \frac{n}{5} \rceil - 1)], \lceil \frac{n}{10} \rceil)$
4. Partition $arr[]$ around medOfMed and obtain its position(i.e. use medOfMed as the pivot element). $pos = partition(arr, n, \text{medOfMed})$
5. If $pos == k$ return medOfMed
6. If $pos < k$ return $kthSmallest(arr[l..pos-1], k)$
7. If $pos > k$ return $kthSmallest(arr[pos+1..r], k - pos + l - 1)$

Using the above method ensures that the chosen pivot, in the worst case, has atmost 70% elements which are larger/smaller than the pivot. The proof of the same as well as the reason behind choosing the group size of 5 is given in the explanation of time complexity.

Java

```java
public class Solution {
    public void swap(int[] list, int i, int j) {
        int temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }
    public int partition(int[] list, int left, int right, int val) {
        int i;
        for (i = left; i < right; i++) {
            if (list[i] == val) {
                break;
            }
        }
        swap(list, i, right);
        int pivotValue = list[right];
        int storeIndex = left;
        for (i = left; i <= right; i++) {
            if (list[i] < pivotValue) {
                swap(list, storeIndex, i);
                storeIndex++;
            }
        }
        swap(list, right, storeIndex);
        return storeIndex;
    }
    int findMedian(int arr[], int l, int len) {
        Arrays.sort(arr, l, l + len);
```

Complexity Analysis

- Time complexity : $O(n)$. Worst case time complexity is $O(n)$.
- Space complexity : $O(n)$ to keep `medians` array of at most `(n + 4) / 5` elements.

Proof: Time Complexity $O(n)$:

The worst case time complexity of the above algorithm is $O(n)$. Let us analyze all steps.

The steps 1. and 2. take $O(n)$ time as finding median of an array of size 5 takes O(1) time and there are $\lceil \frac{n}{5} \rceil$ such arrays. The step 3. takes $T(n/5)$ time(if the whole algorithm takes $T(n)$ time). The step 4. is standard partition and takes $O(n)$ time. The interesting steps are 6. and 7. At most, one of them is executed. These are recursive steps. What is the worst case size of these recursive calls? The answer is maximum number of elements greater than medOfMed (obtained in step 3) or maximum number of elements smaller than medOfMed.

How many elements are greater than medOfMed and how many are smaller?

Let's assume that the list of medians obtained from step 2. in the sorted order be $m_1, m_2, m_3, ..., m_{x-1}, m_x, m_{x+1} ... m_{n-2}, m_{n-1}, m_n$, where m_x is the median chosen as the pivot. To find an upper bound on the number of elements in the given array smaller than our pivot, first consider the half of the medians from step 2($m_1, m_2, ..., m_{x-1}$) which are smaller than the pivot. It is possible for all five of the elements in the sublists corresponding to these medians to be smaller than the pivot(m_x, which leads to an upper bound of $\lceil \frac{n}{5} \rceil * 5 * \frac{1}{2}$ such elements. Now consider the half of the medians from step 2 which are larger than the pivot ($m_{x+1}, ..., m_{n-1}, m_n$). It is only possible for two of the elements(which are smaller than the respective medians) in the sublists corresponding to these medians to be smaller than the pivot(m_x), which leads to an upper bound of $\lceil \frac{n}{5} \rceil * 2 * \frac{1}{2} = \lceil \frac{n}{5} \rceil$ such elements. In addition, the sublist containing the pivot(m_x) contributes exactly two elements smaller than the pivot. It total, we may have at most:

$\frac{5}{2} \lceil \frac{n}{5} \rceil + \lceil \frac{n}{5} \rceil + 2 = \frac{7}{2} \lceil \frac{n}{5} \rceil + 2 \leq \frac{7n}{10} + 6$

elements smaller than the pivot, or approximately 70% of the list. The same upper bound applies the the number of elements in the list larger than the pivot. It is this guarantee that the partitions cannot be too lopsided that leads to linear run time.

Thus, the minimum number of elements which are smaller or larger than the chosen pivot(medOfMed) is given by $n - (\frac{7n}{10} + 6) = \frac{3n}{10} - 6$ or nearly 30% of the elements.

In the worst case, the function recurs for at most $\frac{7n}{10} + 6$ times.

Note that $\frac{7n}{10} + 6 < n$ for $n > 20$ and that any input of 80 or fewer elements requires $O(1)$ time. We can therefore obtain the recurrence:

$$T(n) \leq \begin{cases} \Theta(1), & n \leq 80 \\ T\lceil \frac{n}{5} \rceil + T(\frac{7n}{10} + 6) + O(n), & n > 80 \end{cases}$$

We show that the running time is linear by substitution. Assume that $T(n) = c \cdot n$ for some constant c and all $n > 80$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$T(n) \leq \frac{cn}{5} + c(\frac{7n}{10} + 6) + O(n) \leq \frac{cn}{5} + c + \frac{7cn}{10} + 6c + O(n) \leq \frac{9cn}{10} + 7c + O(n) \leq cn$ since we can pick c large enough so that $c(\frac{n}{10} - 7)$ is larger than the function described by the $O(n)$ term for all $n > 80$. The worst-case running time of is therefore linear.

Choosing the group size of 3 leads to at least half of the n/3 blocks having at least 2 elements $\geq \text{medOfMed}$, hence this gives a n/3 split, or 2n/3 in the worst case.

This gives $T(n)$ = $T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n)$, which reduces to $O(n \log n)$ in the worst case.

There is no reason why you should not use something greater than five; for example with seven the inequality would be

$T(n) \leq T(\frac{n}{7}) + T(\frac{5n}{7}) + O(n)$

$T(n) \leq T(\frac{n}{7}) + T(\frac{5n}{7}) + O(n)$

which also works, but five is the smallest odd number (useful for medians) which works.

Previous Next

DCXiaoBing ★45 May 29, 2020 9:23 PM

In approach 7, it needs extra array to store medians, so space complexity should not be O(1).

ChenyeXu ★3 January 8, 2020 8:24 PM

So brilliant!

Andrewli ★94 August 22, 2018 6:58 AM

@1337c0d3r actually, there is a problem about the `median of medians` strategy. Because the partition uses `two pointers` : all elements `smaller than` the pivot are at the left side, and all elements `equal or greater` are at the right. If `all elements equal the pivot`, every step the size of search set reduces `1` even if using median of median, because `there are 3 / 10 elments`

Anonymouso ★53 March 30, 2020 7:44 AM

Can anyone explain why in the "Approach 6: Using Quick-Select" the median is chosen to be: "nums.length / 2"?

Why is that true for arrays that are of even elements size? I can see that the result is the same whether you choose "nums.length / 2" or "(nums.length / 2) -1" but I don't understand the math behind this.