# Answer of 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 6 | 2 | 4 | 3 | 5 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  |  |  |  |  |  |

Call #1:

quicksort(array, 0, 5)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 6 | 2 | 4 | 3 | 5 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  | l, pivot |  |  |  |  | r |

Pivot is picket the left-most one.

Now, do partition:

Swap, pivot with right position. i=0 and j=4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 5 | 6 | 2 | 4 | 3 | 1 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  | L, i |  |  |  | j | Pivot, r |

While arr[i] < pivot, i++. We get below result as arr[0] > pivot i.e. 5 > 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 5 | 6 | 2 | 4 | 3 | 1 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  | L, i |  |  |  | j | Pivot, r |

While arr[j] > pivot, j--. We get j = -1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 5 | 6 | 2 | 4 | 3 | 1 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  | L, i |  |  |  |  | Pivot, r |

Since, j crossed i. We stop and swap pivot with i. We get below results.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 6 | 2 | 4 | 3 | 5 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  | L, i |  |  |  |  | r |

Now we do self-calls:

quicksort(arr, 0, -1) -> This will stop as 0 > -1.

quicksort(arr, 1, 5)

Self-call #1:

quicksort(arr, 1, 5)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 6 | 2 | 4 | 3 | 5 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, pivot |  |  |  | r |

Partition:

Swap pivot element to right element. We have i=1 and j = 4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 5 | 2 | 4 | 3 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L,i |  |  | j | r, pivot |

While arr[i] < pivot, i++. We’ll get i= 5 as everything is less than pivot.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 5 | 2 | 4 | 3 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L |  |  | j | i, r, pivot |

I crossed j so we don’t loop over j. Now, swap pivot and I and return i=5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 5 | 2 | 4 | 3 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L |  |  |  | i, r |

We got new pivot location i=5 returned. Now, do self-calls on the left and right partitions.

quicksort(arr, 1, 4)

quicksort(arr, 6, 5) -> we don’t do this as 6 > 5.

Self-call #2:

quicksort(arr, 1, 4)

Pivot is always selected to be left.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 5 | 2 | 4 | 3 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, pivot |  |  | r |  |

Partition:

Swap pivot element to right element. We have i=1 and j = 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 3 | 2 | 4 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, i |  | j | R, pivot |  |

While arr[i] < pivot, i++. We’ll get i= 5 as everything is less than pivot.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 3 | 2 | 4 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L |  | j | I, R, pivot |  |

I crossed j so we don’t loop over j. Now, swap pivot and I and return i=5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 3 | 2 | 4 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L |  |  | i, R |  |

We got new pivot location i=4 returned. Now, do self-calls on the left and right partitions.

quicksort(arr, 1, 3)

quicksort(arr, 5, 4) -> we don’t do this as 5 > 4.

Self-call #3:

quicksort(arr, 1, 3)

Pivot is the left-most element.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 3 | 2 | 4 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, pivot |  | R |  |  |

Partition:

Swap pivot with right position.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 4 | 2 | 3 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, i | j | R, pivot |  |  |

While arr[i] < pivot, i++. We’ll get i= 1 as 4 > 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 4 | 2 | 3 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, i | j | R, pivot |  |  |

While arr[j] > pivot, j--. We’ll get j= 2 as 2 < 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 4 | 2 | 3 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, i | j | R, pivot |  |  |

As I and j are stuck, we swap the values and continue again.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 2 | 4 | 3 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L, i | j | R, pivot |  |  |

While arr[i] < pivot, i++. We’ll get i= 2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 2 | 4 | 3 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L | J, i | R, pivot |  |  |

I crossed j so we don’t loop over j. Now, swap pivot and I and return i=2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 2 | 3 | 4 | 5 | 6 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|  |  | L | i | R |  |  |

We got new pivot location i=2 returned. Now, do self-calls on the left and right partitions.

quicksort(arr, 1, 1) we stop as low == high

quicksort(arr, 3, 3) we stop as low == high

We get final result.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array | 1 | 2 | 3 | 4 | 5 | 6 |

# Answer of 2

Array: = [5, 1, 4, 3, 6, 2, 7, 1, 3]

Size, n = 9

For simplification, see the sorted array only to see what L, E and R would be.

Since, n =9, 3n/4 = 3 \* 9 / 4 ≈ 6.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array | 5 | 1 | 4 | 3 | 6 | 2 | 7 | 1 | 3 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sorted Array | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 6 | 7 |

When pivot is 1.

L = []

E = [1, 1]

R = [2,3,3,4,5,6,7]

L is of size 0 and R is of size 7.

When pivot = 2

L = [1, 1]

E = [2]

R = [3,3,4,5,6,7]

L is of size 2 and R is of size 6.

When pivot = 3

L = [1, 1, 2]

E = [3, 3]

R = [4,5,6,7]

L is of size 3 and R is of size 4.

When pivot = 4

L = [1, 1, 2, 3, 3]

E = [4 ]

R = [5,6,7]

L is of size 5 and R is of size 3.

When pivot = 5

L = [1, 1, 2, 3, 3, 4]

E = [5]

R = [6,7]

L is of size 6 and R is of size 2.

When pivot = 6

L = [1, 1, 2, 3, 3, 4, 5]

E = [6]

R = [7]

L is of size 7 and R is of size 1.

When pivot = 7

L = [1, 1, 2, 3, 3, 4, 5, 6]

E = [7]

R = []

L is of size 8 and R is of size 0.

1. From above, we can see that pivots [3,3, 4] are the only good pivots.

The pivots [1, 1, 2, 5, 6, 7] are bad pivots as either L or R is greater than 6.

1. No, if the array have duplicates, then there’s no guarantee that at least half of them would be good pivots.

# Answer of 3

The best case ideally would be to split the array in n/2 for both Left and Right partition. However, pivot itself always makes sure that neither Left and Right can be n/2.

A diagram of a flowchart

Description automatically generated

Running Time = no. of levels \* amount of work at each level

At each level, the array is divided into two partitions of < 3n/4. So, the height of the tree is the same as the number of levels.

Which will be O(logn).

At each level of the tree, the total processing time is O(n).

Hence Total Running Time = O(nlogn)

This is the best running time for quick sort O(nlogn).

# Answer of 4

class Solution {

public int findKthLargest(int[] nums, int k) {

int position = nums.length - k;

int start = 0;

int end = nums.length - 1;

while (start<end){

int cursor = quickSelect(nums, start, end);

if(cursor == position){

break;

} else if(position < cursor){

end = cursor - 1;

} else {

start = cursor + 1;

}

}

return nums[position];

}

int quickSelect(int[] arr, int start, int end) {

int pivot = arr[end];

int position = start;

for(int i = start; i < end; i++){

if(arr[i] < pivot){

if (position != i ){

swap(arr, i, position);

}

position++;

}

}

arr[end] = arr[position];

arr[position] = pivot;

return position;

}

void swap(int[] arr, int index1, int index2){

int temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

}