Algorithms: Quick Sort -Part-2

Relevel
by Unacademy

Topics

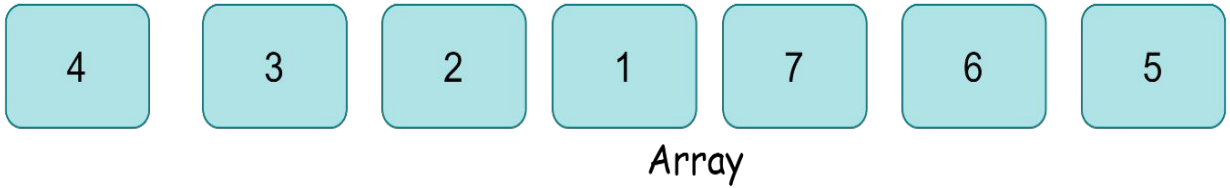
- Quick select algorithm
- Solving order statistics problem
- median of medians vs quick select
- Merge sort vs quick sort

Quick Select Algorithm

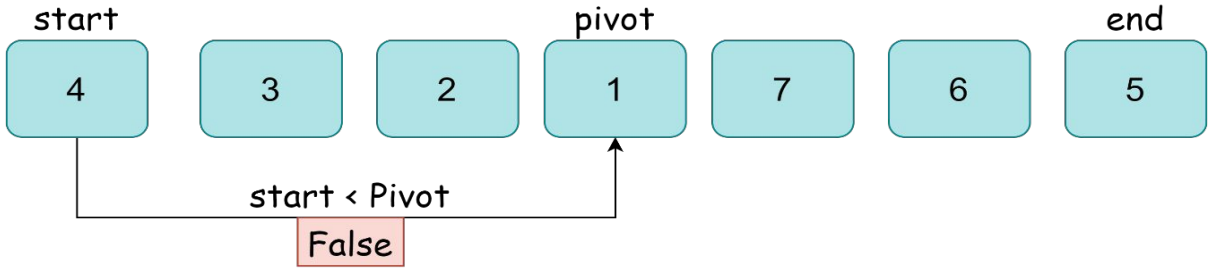
Quick Select is similar to the quicksort algorithm. but the only difference we no need to sort the entire array. It is an optimized way to find the kth smallest/largest element in an unsorted array.

- The partition (splitting) part of the algorithm is the same as that of quick sort algorithm.
- After the partition function split the array into two sub array then check the pivot element with Kth index ,so instead of recursing both sides of the pivot index, we recurse only for the part that contains our kth element

- Consider an array of element 4, 3, 2, 1, 7, 6, 5 and K is 5

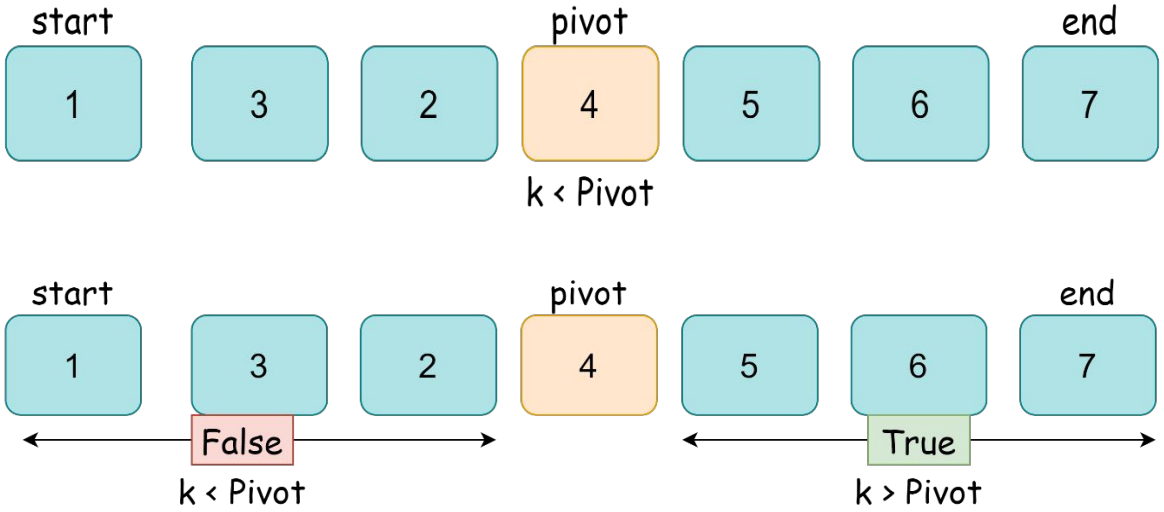


Consider a pivot element as 1 and partition the sub array ie. one having element less than pivot and one having element greater than pivot

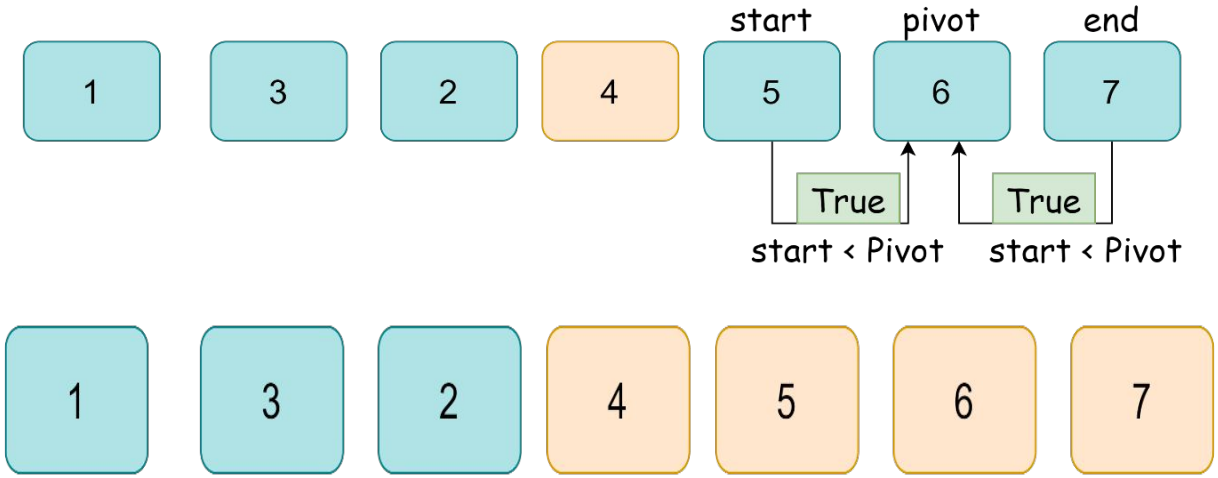


If K is less than pivot then consider the left sub array for next step sort

If K is greater than pivot then consider the right sub array for next step sort



Sort only the right sub array not the left array because sorted pivot element is in position 3rd index which is greater than K



Array
[1, 3, 2, 4, 5, 6, 7] a[k - 1]

Now you will get the Kth index element in the partially sorted array and by using index will find the Kth value

Code

Code link -

https://jsfiddle.net/saravananslb/q2uo41hc/

```
// Sort the one side of the array recursively
const quickSort = (arr, low, high, k) => {
    if (low >= high) {
        return;
    }
    let start = low;
    let end = high;
    const mid = Math.ceil((low + high) / 2);
    const pivot = arr[mid];
    while (start < end) {
        while (arr[start] < pivot) {
            start++;
        }

        while (arr[end] > pivot) {
            end--;
        }

        if (start <= end) {
            swap(arr, start, end);
            start++;
            end--;
        }
    }

    if (k === pivot){
        quickSort(arr, low, end, k);
        return;
    }
    if (k < pivot){
        quickSort(arr, low, end, k);
    }
    else {
        quickSort(arr, start, high, k);
    }

}
```

```
// quick select function to format the input
const quickSelect = (arr, k) => {
    const start = 0;
    const end = arr.length - 1;
    quickSort(arr, start, end, k)
}

// Swapping two index value in an array
const swap = (arr, a, b) =>{
    let temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}


const k = 8;
const unsortedArray = [4, 3, 10, 24, 2, 1, 7, 6, 5]
quickSelect(unsortedArray, k)
console.log(unsortedArray[k - 1])
```

Time Complexity Analysis

- Worst case: Worst case occurs when we pick the largest/smallest element as pivot.

  $T(n) = T(n-1) + cn$

  $= T(n-1) + T(n-2) + cn + c(n-1)$

  $= O(n^2)$

- Best case: The best-case occurs when QuickSelect chooses the K-th largest element as the pivot in the very first call. Then, the algorithm performs theta(n) steps during the first (and only) partitioning, after which it terminates. Therefore, QuickSelect is theta(n) in the best case.

  $T(n) = T(n/2) + cn$

  $= T(n/4) + c(n/2) + cn$

  $= n(1 + ½ + ¼ + ...) = 2n = \mathbf{\Omega}(n)$

Time Complexity Analysis

- Average case:
- As a quick select we don't need to sort all the element, sort the half of the earch sub array since our kth element is either greater than pivot or smaller than pivot.
- n + 1/2 n + 1/4 n + 1/8 n + ... < 2 n
- Quick select allows us to solve the problem in Θ (n) on an average. Using randomized pivot

T(n) = n + T(n/2)

= n + n/2 + T(n/4)

= n + n/2 + n/4 + ... n/n

= n(1 + 1/2 + 1/4 + ... + 1/2^log2(n)) = n (1/(1-(1/2))) = 2n

The time complexity for the average case for quick select is O(n) (reduced from O(nlogn) — quick sort). The worst case time complexity is still $O(n^2)$ but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(n) solution to find the kth largest/smallest element in an unsorted list.

Solving Order Static Problems

Quick Select is a variation of the quicksort algorithm. It is an optimized way to find the kth smallest/largest element in an unsorted array.

this should be the starting part for explaining order statistics problem.

Done

Code - https://jsfiddle.net/saravananslb/teabpsfk/

1) Sum and product of k smallest and k largest numbers in the array

Given input of array [4, 3, 2, 10, 24, 2, 1, 7, 6, 5]

```
// Sort the one side of the array recursively
const quickSort = (arr, low, high, kSmall, KHigh) => {
    if (low >= high) {
        return;
    }
    let start = low;
    let end = high;
    const mid = Math.ceil((low + high) / 2);
    const pivot = arr[mid];
    while (start < end) {
        while (arr[start] < pivot) {
            start++;
        }

        while (arr[end] > pivot) {
            end--;
        }

        if (start <= end) {
            swap(arr, start, end);
            start++;
            end--;
        }
    }

    if (kSmall === pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
        return;
    }
    if (kSmall < pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
    }
    else {
        quickSort(arr, start, high,  kSmall, KHigh);
    }
```

Explanation:

First split the array into two sub array for getting the smallest and largest element.

Next choose the pivot in the left array and again split into two sub array then check if the pivot is lesser or greater than K

If k less than pivot choose the left sub array if not choose the right sub array
Recursively do this process till find the Kth smallest element

Do the same for Kth largest element

```
    if (KHigh === pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
        return;
    }
    if (KHigh < pivot){
        quickSort(arr, low, end,  kSmall, KHigh);
    }
    else {
        quickSort(arr, start, high,  kSmall, KHigh);
    }

}

// quick select function to format the input
const quickSelect = (arr) => {
    const start = 0;
    const end = arr.length - 1;
    quickSort(arr, start, end, start, end)
    return (arr[start] * arr[end]);
}

// Swapping two index value in an array
const swap = (arr, a, b) =>{
    let temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}


const unsortedArray = [4, 3, 10, 24, 2, 1, 7, 6, 5]
const product = quickSelect(unsortedArray)
console.log(product)
```

median of medians

- median of medians is an approximate (median) selection algorithm, frequently used to supply a good pivot for an exact selection algorithm, mainly the quickselect, that selects the kth smallest element of an initially unsorted array.
- It is based on divide and conquer algorithm in that it will returns a pivot that in the worst case it will divide a list of unsorted array elements into sub-problems of size 3n/10 and 7n/10 assuming we choose a sublist size of 5.
- Let us consider 10 element and break this into 5 and 5 elements and for choosing the pivot it will make a partition based on the element (left and right sub array) then find the middle element as pivot for the first 5 elements.
- It will provide a good pivot that in the worst case will give a pivot in the range between 30% and 70 % of the list of size n.

  If you want to choose a good pivot, it is essential for it to be around the middle, 30-70% most of the case the pivot will be around the middle 40% of the list.

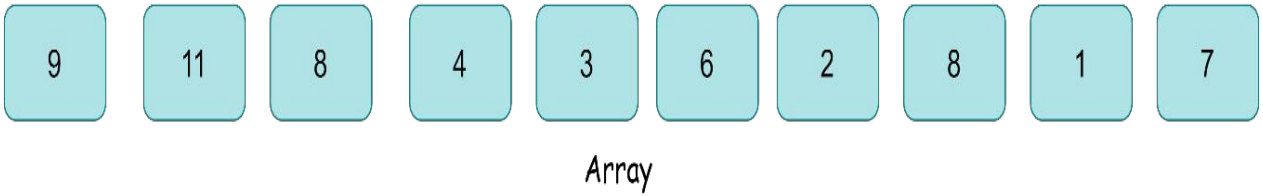
median of medians

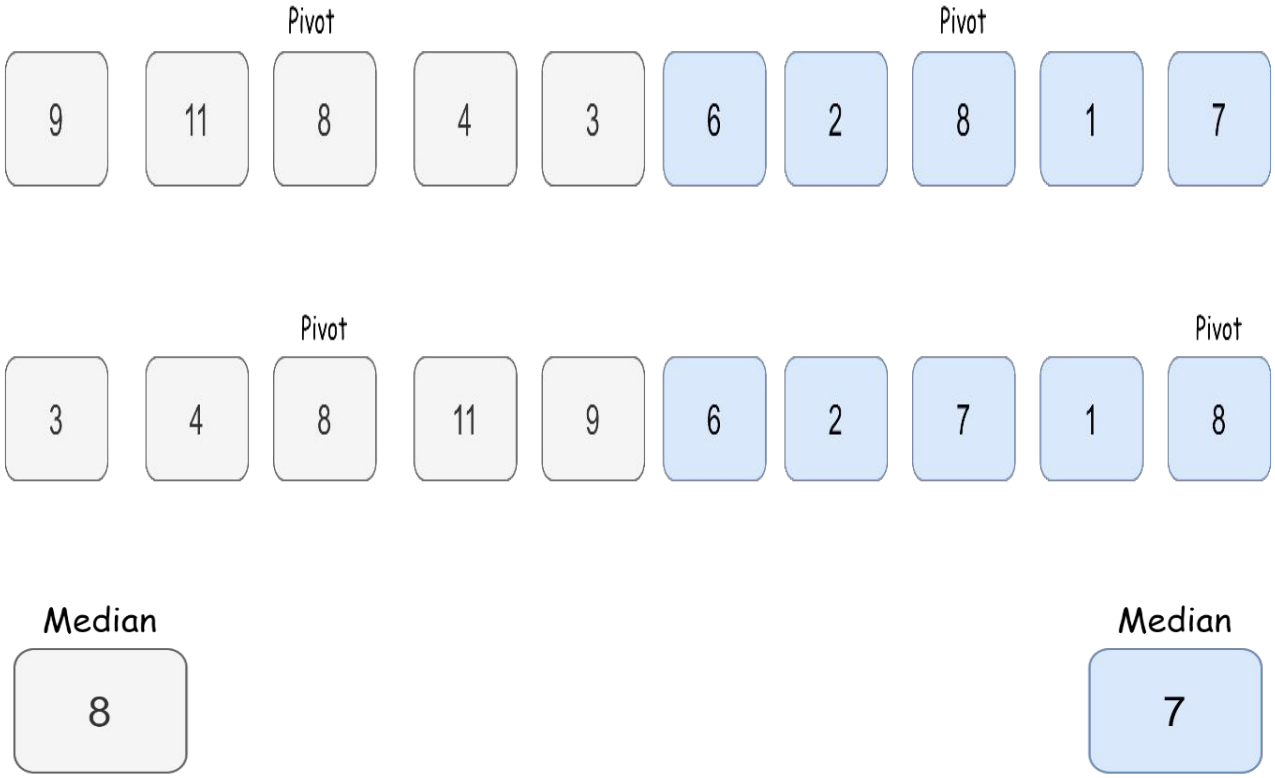
Time and Space Complexity of Median of Medians Algorithm

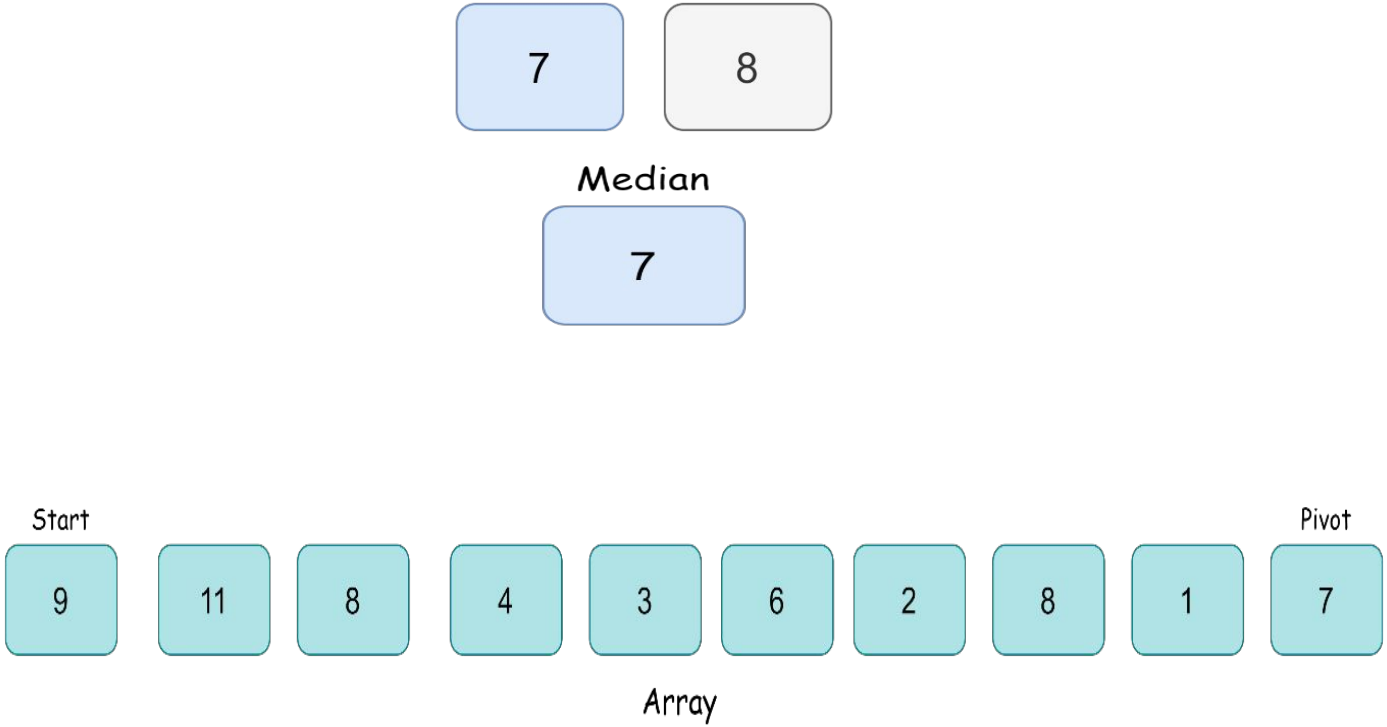
- This algorithm works in O(n) linear time complexity, we traverse the array once to find the medians in the sub array and another time to find the true median to be used as a pivot.

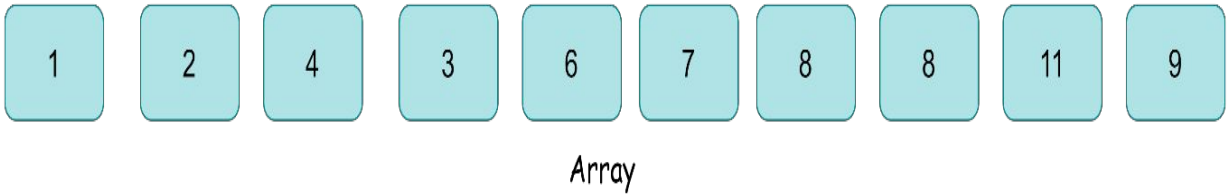
- The space complexity of mom is O(logn) and the memory used will be proportional to the size of the array.

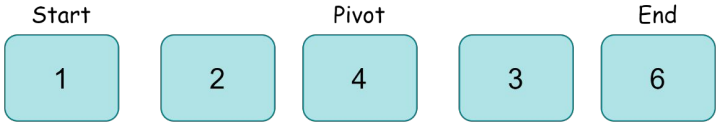
Code link - https://jsfiddle.net/saravananslb/kn2rzjf8/




Pivot
Pivot
9
11
8
4
3
6
2
8
1
7
Pivot
Pivot
3
4
8
11
9
6
2
7
1
8
Median
8
Median
7


7
8
Median
7
Start
Pivot
9
11
8
4
3
6
2
8
1
7
Array





Code link - https://jsfiddle.net/saravananslb/kn2rzjf8/

```
function quickselect_median(arr) {
    const L = arr.length, halfL = L/2;
    if (L % 2 == 1)
        return quickselect(arr, halfL);
    else
        return 0.5 * (quickselect(arr, halfL - 1) + quickselect(arr, halfL));
}

function quickselect(arr, k) {
    if (arr.length == 1)
        return arr[0];
    else {
        const pivot = arr[0];
        const lows = arr.filter((e)=>(e<pivot));
        const highs = arr.filter((e)=>(e>pivot));
        const pivots = arr.filter((e)=>(e==pivot));
        if (k < lows.length)
            return quickselect(lows, k);
        else if (k < lows.length + pivots.length)
            return pivot;
        else
            return quickselect(highs, k - lows.length - pivots.length);
    }
}

console.log(quickselect_median([7,3,5, 4, 10]));
```

The algorithm is as follows,

1. Divide the array into sub arrays of size n, assume 5.
2. Loop through the whole array in sizes of 5, considering our array is divisible by 5.
3. For n/5 sub arrays, use select brute-force subroutine algorithm to select a median m, which is in the 3rd place out of 5 elements.
4. Add medians obtained from the sub arrays to the array M.
5. Use quick Select recursively to find the median from array M, The median obtained is the best pivot.
6. Stop the recursion loop once the base case is hit, ie., when the sub array becomes small enough. Use Select brute-force subroutine to find the median.

Note: We used each splitted sub arrays of size 5 because selecting a median from a list when the size of the array is an odd number is easier. Even numbers require additional computation. We can also select 7 or any other odd number as we shall see in the proofs below.

median of medians vs quick select

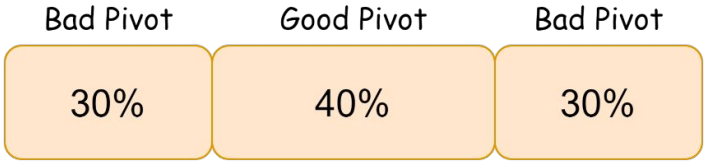
To avoid the O(n^2) worst case scenario for quick select, we will pick either of the below in the quick sort

1. Randomly choose a pivot index
2. Use median of medians (MoM) to select an approximate median and pivot around that

When using MoM with quick select, we can guarantee worst case O(n). When choosing pivot as random, we can't guarantee worst case O(n), but the probability of the algorithm going to O(n^2) should be extremely small.

The overhead cost of median of medians is much more than choosing random pivot, where the latter adds little to no additional complexity.

- Assuming quickselect is truly using a random choice of pivot at each step, you can prove that not only is the expected runtime O(n), but that the probability that its runtime exceeds Θ(n log n) is very, very small (at most 1 / n^k for any choice of constant k). So in that sense, if you have the ability to select pivots at random, quickselect will likely be faster.



- However, not all implementations of quickselect use true randomness for the pivots, and some use deterministic pivot selection algorithms. This, unfortunately, can lead to pathological inputs that trigger the Θ(n^2) worst-case runtime, which is a problem if you have adversarially-chosen inputs.

- Once nice compromise between the two is ***introselect***. The basic idea behind introselect is to use quickselect with a deterministic pivot selection algorithm. As the algorithm is running, it keeps track of how many times it's picked a pivot without throwing away at least 30% the input array.

If that number exceeds some threshold, it stops using a random pivot choice and switches to the median-of-medians approach to select a good pivot, forcing a 30% size reduction.

This approach means that in the common case when quickselect rapidly reduces the input size, introselect is basically identical to quickselect with a tiny bookkeeping overhead. However, in cases where quickselect would degrade to quadratic, introselect stops and switches to the worst-case efficient median-of-medians approach, ensuring the worst-case runtime is O(n).

This gives you, essentially, the best of both worlds - it's fast on average, and its worst-case is never worse than O(n).

Merge Sort vs Quick Sort

Merge Sort	Quick Sort
An efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order	An efficient, general purpose, comparison-based sorting algorithms
It is suitable sort and is ideally preferred for linked lists	It is a pervasively used sorting algorithm ideally preferred for arrays
The bulk of work is to merge two sub lists which takes place after the sub lists are sorted	The bulk of work is to partition the list into two sub lists which takes the place before the sub list are sorted
Divides the array into two sub arrays (n/2) again and again until one element is left	Sorts the element by comparing each element with the pivot
Works in consistent speed for all datasets	Works faster for small datasets
It is an external sorting method in which the element in the sorted array cannot be stored in the memory at the same time and some has to be kept in the auxiliary memory	It is an internal sorting method where the element in the array is adjusted within the main memory
More efficient for larger arrays	Not efficient for large arrays it is recursively partition the array
It requires a temporary array for merging two sub arrays	No additional array space is required
The worst case performance of the merge sort is O(nlog2 n)	The worst case performance of the Quick sort is O(n^2)
The Average case performance of merge sort is O(n log n)	The Average case performance of Quick sort is O(n log n)
The Best case performance of merge sort is O(n log n)	The Best case performance of Quick sort is O(n log n)

Practice Problem

1) Write a program to find the Kth largest element in the array using quick select.
2) Given an array. You need to sort an array using the iterative quick sort algorithm
   Input - [3,6,5,2,10]
   Output - [2,3,5,6,10]

Upcoming Teaser

- Problem Solving on Sorting

Thank you