QuickSort

Recap: QuickSelect

It is a **selection algorithm**. It looks for the **k-th smallest element** in an unsorted array.

It uses a **partition strategy** to find if index of the partitioned element is more than k, then we recur for the left part. If index is the same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for the right part.

Using a **random pivot** when defining partitions we can **avoid the worst case** (largest/smallest element picked as pivot).

Quick Select - Time Complexity

Best Case: when we partition the list into two halves and continue with only the half we are interested in.

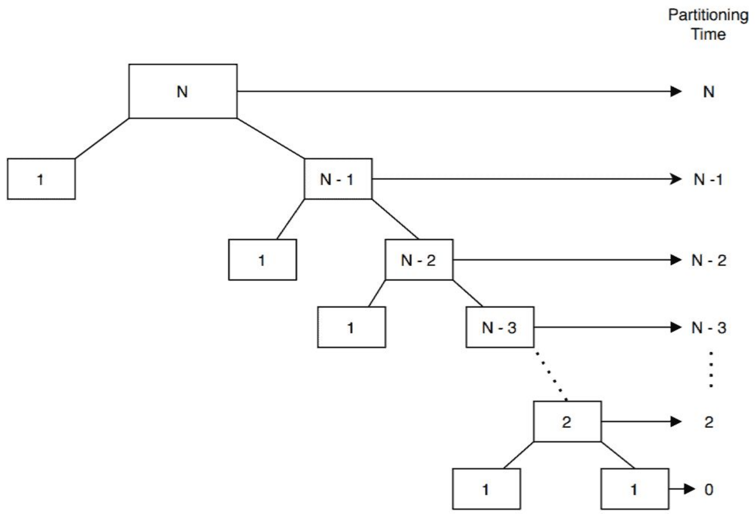
N + N/2 + N/4 + N/8 + N/16

This is a geometric series that evaluates approximately 2*N.

$\boxed{\textbf{O(n)}}$

Quick Select - Time Complexity

Worst case: larger/smallest element picked as pivot



$\boxed{O(N^2)}$

The **height of the tree** will be **N** and in **top node** we will be doing **N operations**

then N - 1 and so on until 1

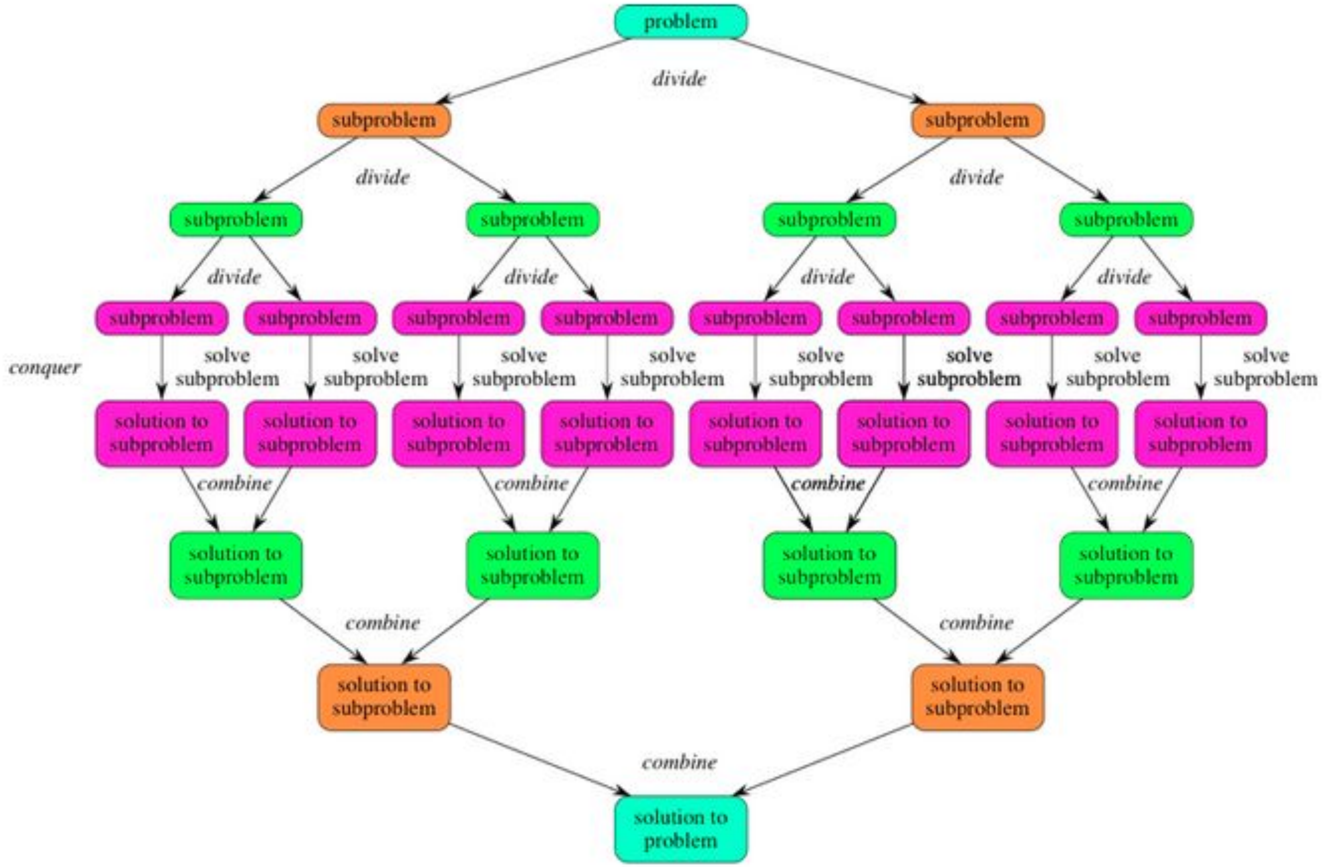
Divide and Conquer

Some algorithms, such as QuickSelect and QuickSort employ a common algorithmic paradigm based on recursion.

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. (Wikipedia)

Divide and conquer algorithm parts

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to the subproblems into the solution for the original problem.


problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
combine
combine
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
solution to subproblem
solution to subproblem
combine
solution to problem

QuickSort

QuickSort uses the divide and conquer strategy (recursive algorithm).

- **Divide** by choosing any element (the pivot) and creating the partitions.
- **Conquer** by recursively sorting the subarrays array[start … pivot - 1] (all elements to the left of the pivot, which must be less than or equal to the pivot) and array[pivot + 1 ... end] (all elements to the right of the pivot, which must be greater than the pivot).
- **Combine** by doing nothing. The conquer step recursively sorts the entire array.

QuickSelect and QuickSort

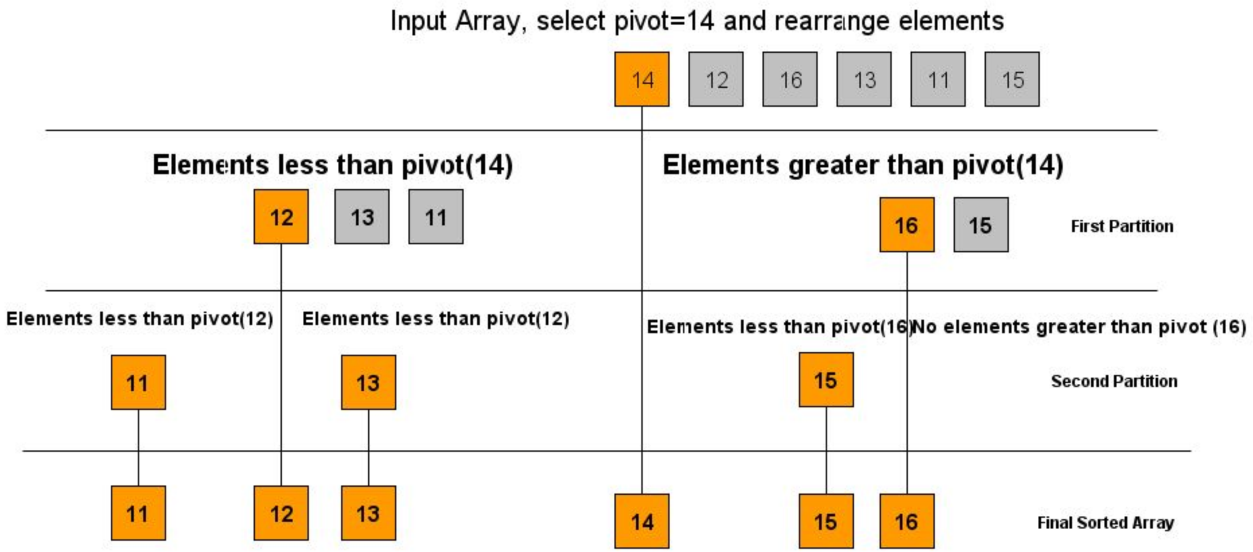
Both algorithms:

- It picks an element as a pivot
- It creates partitions based on the picked pivot

Difference:

- QuickSelect returns the value of the k-th element while the goal of the QuickSort is to use the logic partitions to sort the array

QuickSort



Input Array, select pivot=14 and rearrange elements
14
12
16
13
11
15
Elements less than pivot(14)
12
13
11
Elements greater than pivot(14)
16
15
First Partition
Elements less than pivot(12)
Elements less than pivot(12)
Elements less than pivot(16)
No elements greater than pivot (16)
11
13
15
Second Partition
11
12
13
14
15
16
Final Sorted Array

What does it happen after each call to partition?

QuickSort versions

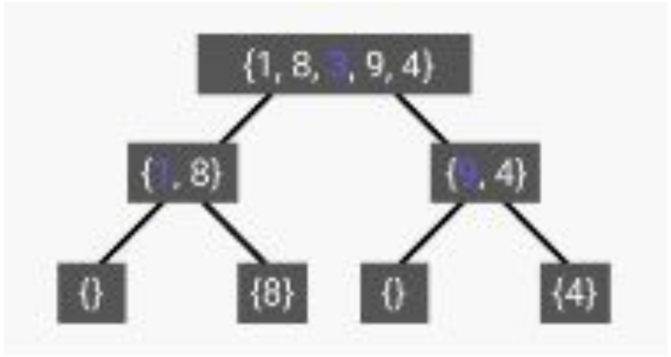
A pivot element for the QuickSort can be picked in different ways:

- Pick median as the pivot
- **Pick a random element as a pivot:** preferred because it avoids a predictable pattern that could lead to the worst-case scenario.
- Always pick the first element as a pivot
- Always pick the last element as a pivot

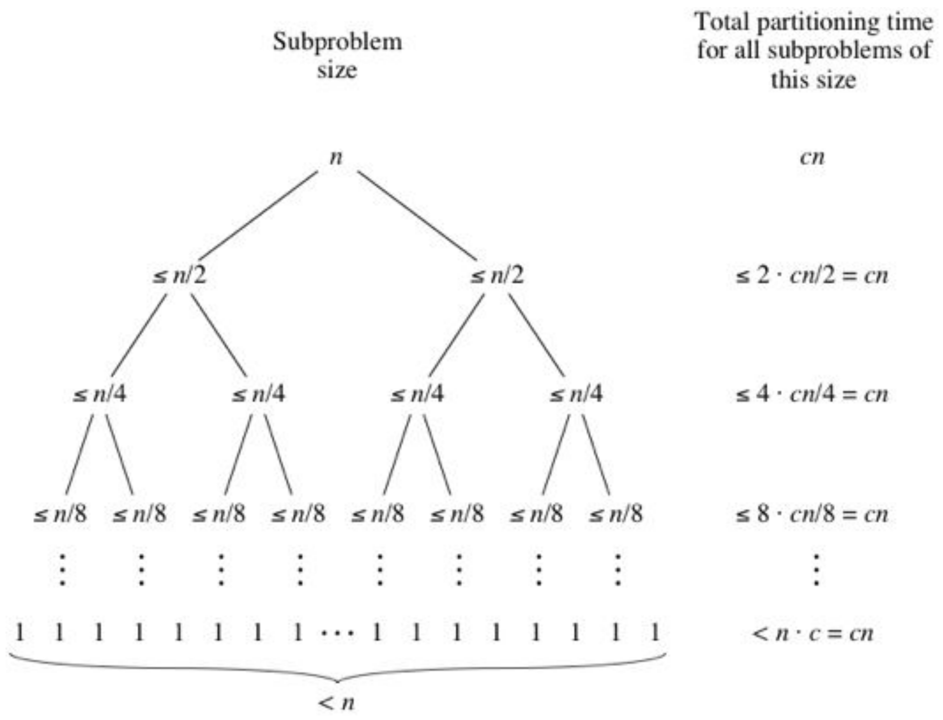
Time Complexity - Best Cases

The **best-case** occurs when the **pivot element is the middle element or near** to the middle element

In this case the recursion will look as shown in the diagram, as we can see the **height of tree is logN** and in each level we will be traversing to all the elements with total operations will be **logN * N**



Time Complexity - Best Case

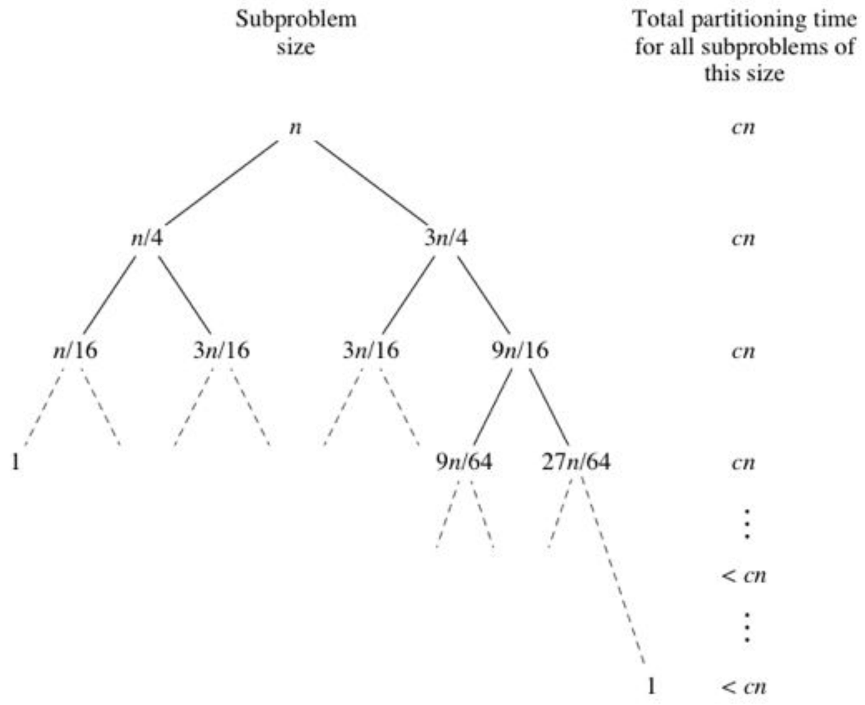


The tree levels represent the number of times we can repeatedly halve, starting at n, until we get the value 1, plus one.

c is a constant representing the work done per partition step.

Best case: O(n log(n))

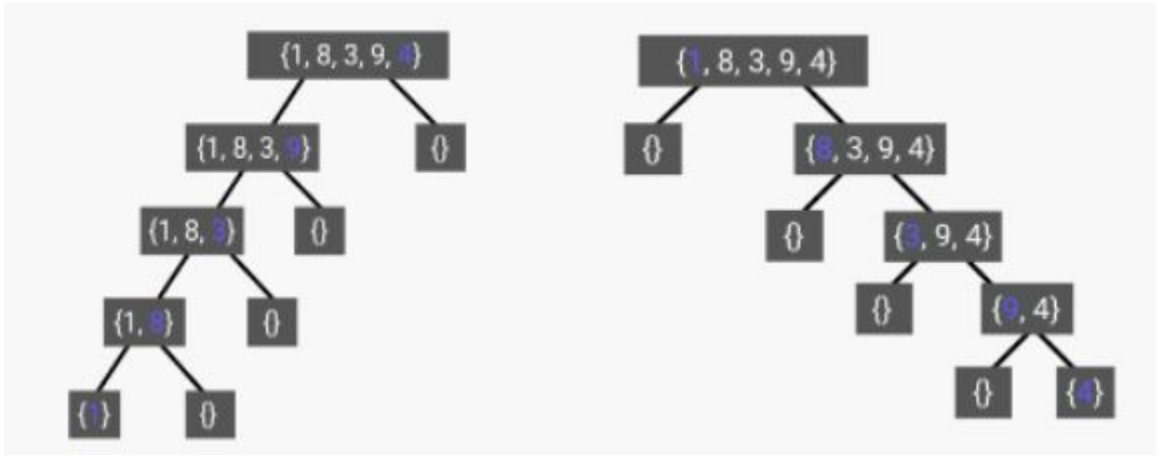
Time Complexity - Average Case



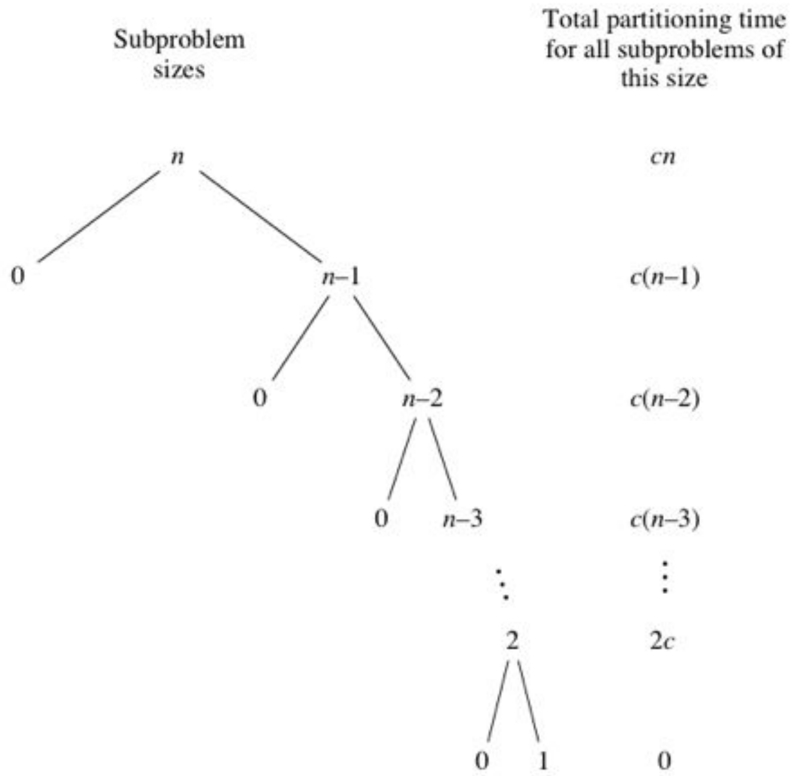
Average case: O(n log(n))

Time Complexity - Worst Case

The worst case occurs when the pivot element is either the greatest or the smallest element.



Time Complexity - Worst Case



$$\begin{aligned} T(n) &= cn + c(n-1) + c(n-2) + \dots + 2c \\ &= c(n + (n-1) + (n-2) + \dots + 2) \\ &= c((n+1)(n/2) - 1) \end{aligned}$$

Worst-case: $O(n^2)$

QuickSort: Performance issue

QuickSort exhibits poor performance for inputs that contain many repeated elements. The problem is visible when all the input elements are equal.

Then at each point in recursion, the left partition is empty (no input values are less than the pivot), and the right partition has only decreased by one element (the pivot is removed).

The algorithm takes quadratic time to sort an array of equal values.

Quicksort using Dutch National Flag Algorithm

It Implements QuickSort efficiently for inputs containing many repeated elements.

We can use an alternative linear-time partition routine to solve this problem that separates the values into three groups:

- The values less than the pivot,
- The values equal to the pivot, and
- The values greater than the pivot.

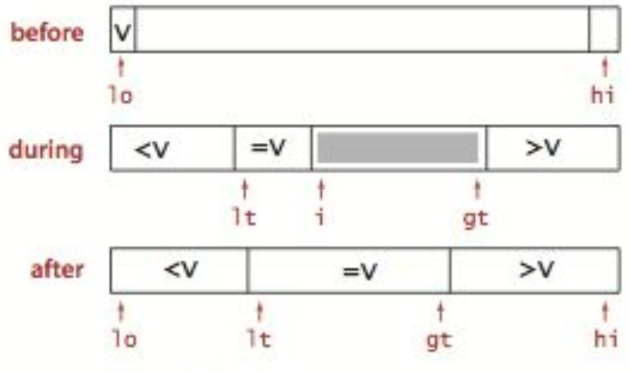
The values equal to the pivot are already sorted, so only the less-than and greater-than partitions need to be recursively sorted.

Quicksort using Dutch National Flag Algorithm

This method will not call QuickSort on the middle section which has the elements equals to pivot.

It must return the left and right index of the middle section.

The method must return an array with the the left and right indices. QuickSort calls will use those values.



3-way partitioning overview

Coding Time = Fun Time

Add the following method to your Quick.java file:

QuickSort

```
public static void quickSort(int[] data) {
    quickSort(data, 0, data.length - 1);
}

public static void quickSort(int[] data, int start, int end) {
    // Your code here
}
```

Dutch Partition is Optional

```
public static int[] partitionDutch(int[] data, int start, int end) {
    // Your code here

}
```