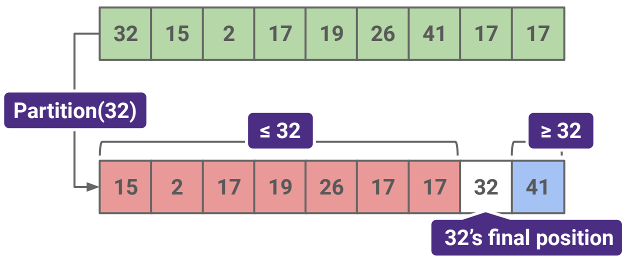
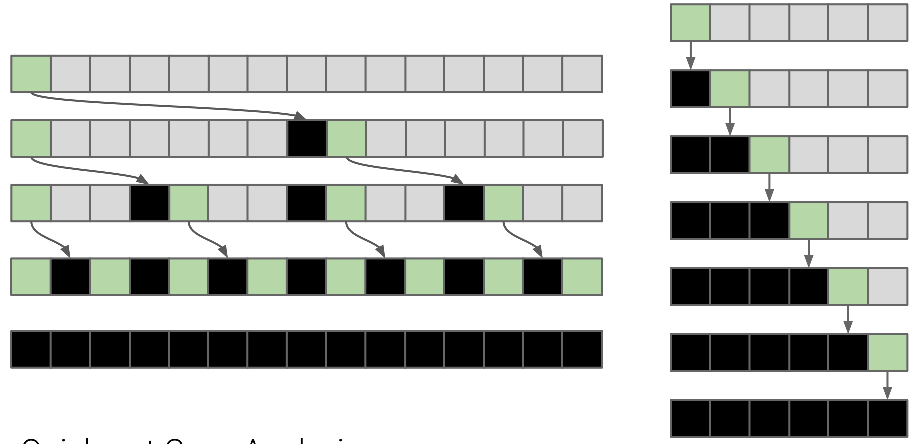
Naive Quicksort

Demo

1. Partition around a **pivot item**, e.g. leftmost item.
2. Quicksort left side, all keys ≤ pivot.
3. Quicksort right side, all keys ≥ pivot.





Quicksort Case Analysis

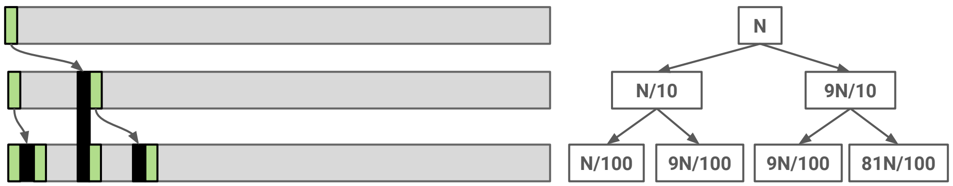
Give the tight asymptotic time complexity of naive quicksort assuming no duplicate keys.

- $\Omega(N), O(N \log N)$
- $\Omega(N), O(N^2)$
- $\Omega(N \log N), O(N \log N)$
- $\Omega(N \log N), O(N^2)$
- $\Omega(N^2), O(N^2)$
- Not sure

Sort	Best-Case	Worst-Case	Space	Stable	Notes
Selection Sort	Θ(N^2)	Θ(N^2)	Θ(1)	No	
Heapsort	Θ(N)	Θ(N log N)	Θ(1)	No	Slow in practice.
Merge Sort	Θ(N log N)	Θ(N log N)	Θ(N)	Yes	Fastest stable sort.
Insertion Sort	Θ(N)	Θ(N^2)	Θ(1)	Yes	Best for small or almost sorted inputs.
Naive Quicksort	Θ(N log N)	Θ(N^2)	Θ(N)	Yes	2x or more slower than merge sort.
Java Quicksort	Ω(N)	O(N^2)	?	No	Fastest comparison sort.

Argument 1: 10% Case

Suppose the pivot is always at least 10% from either edge (not to scale).



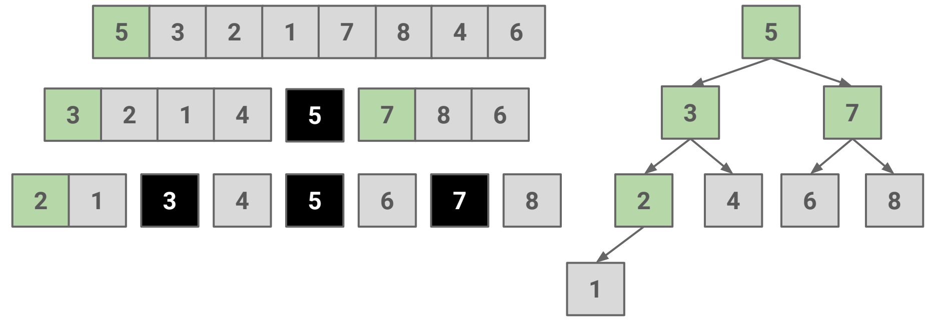
Work at each level is in O(N).

Height is about $\log_{10/9} N \in O(\log N)$.

Overall: **O(N log N)**.

Argument 2: Binary Search Tree Analogy

Random insertion into a binary search tree is expected to take O(N log N) time.



Optimizing Quicksort

Naive Quicksort.

Recursive Depth. Ω(log N), O(N).

Pivot choice. Leftmost item. Θ(1)

Common worst-case: **sorted array!**

Partitioning. Allocate a new array. Θ(N)

Slow but stable.

Common worst-case: **all duplicates!**

Java Quicksort–5x or more faster.

Recursive Depth. Ω(log N), O(N).

Pivot choice. Approximate median. Θ(1)

Resilient to worst-case inputs.

Partitioning. Long-distance swaps. Θ(N)

In-place, fast, but unstable.

3-way partition to handle duplicates.

Q Median-Finding

Goal. Find the median item in O(N) time.

Reduces to the **selection problem**.

Selection. Given an array of N items, find item of rank **K**.

For median, find **K** = N / 2.

How difficult is this problem?

- Why is the time complexity of selection in Ω(N)?
- Describe an O(N log N) runtime algorithm for selection with any **K**.
- Describe an O(N) runtime algorithm for selection with **K** = 0, 1, 2.

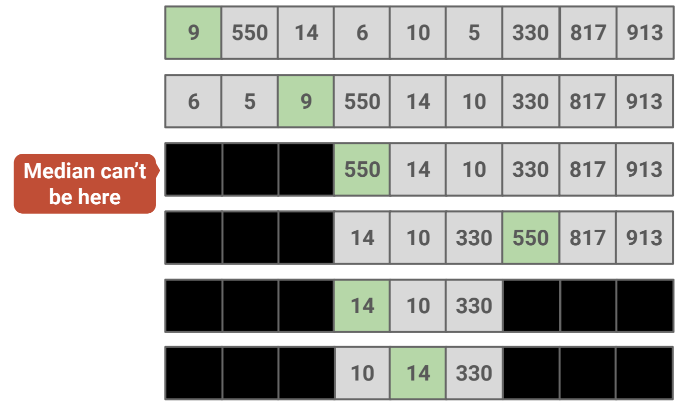
When poll is active, respond at **PollEv.com/kevinl**

If selection reduces to sorting, which of the following statements about problem difficulty is true?

- $\text{Selection} \leq \text{Sorting}$
- $\text{Sorting} \leq \text{Selection}$
- $\text{Selection} > \text{Sorting}$
- $\text{Sorting} > \text{Sorting}$
- Not sure

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**

Total Results



Quickselect: Partition-selection with leftmost item as pivot

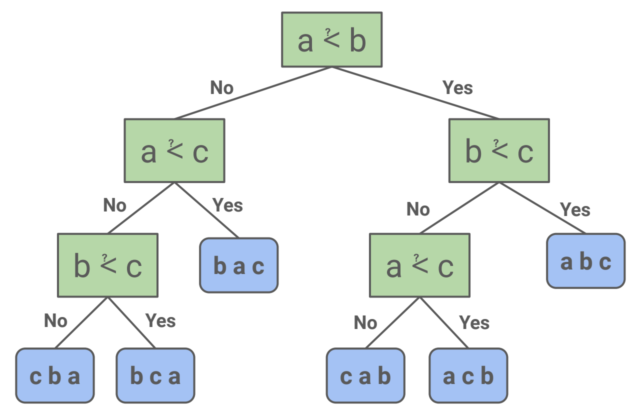
Approximate Median-Finding

Unfortunate reality: Quicksort with quickselect pivots is significantly slower than merge sort.

Goal. Find the **approximate median** item in Θ(1) time.

Median-of-3. Pick 3 items and take the median of the sample.

```
if (a < b)
  if      (b < c) return b;
  else if (a < c) return c;
  else            return a;
else
  if      (a < c) return a;
  else if (b < c) return c;
  else            return b;
```

Median-of-3 Decision Tree

Hoare Partitioning

Demo

Hoare partitioning. In-place, **unstable** partitioning algorithm. Initialize an int **L** and an int **G**.

- **L.** Left pointer that loves small items < pivot.
- **G.** Right pointer that loves big items > pivot.

Idea. Walk towards each other, swapping anything they don't like.

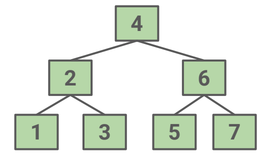
End result is that things on left are "small" and things on the right are "large".

Hoare partitioning improves real-world runtime and space complexity.

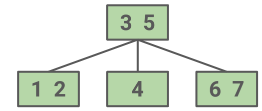
Asymptotic time complexity still depends on pivot choice!

Dual-Pivot Quicksort: Data Structure Analogy

If classic quicksort is analogous to BSTs, then dual-pivot quicksort is analogous to 2-3 trees.



"Classic Quicksort"

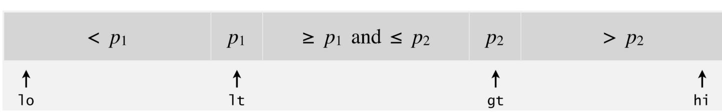


Dual-Pivot Quicksort

Dual-Pivot Quicksort

Use **two partitioning keys** p_1 and p_2 and partition into three subarrays:

- Keys less than p_1.
- Keys between p_1 and p_2.
- Keys greater than p_2.



Recursively quicksort the three subarrays (skip middle subarray if $p_1 = p_2$).

Now widely used. Java 8, Python unstable sort, Android, ...

Algorithms (Robert Sedgewick, Kevin Wayne/Princeton)

Sort	Best-Case	Worst-Case	Space	Stable	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	No	
Heapsort	$\Theta(N)$	$\Theta(N \log N)$	$\Theta(1)$	No	Slow in practice.
Merge Sort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Yes	**Fastest stable sort.**
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Yes	Best for small or almost sorted inputs.
Naive Quicksort	$\Theta(N \log N)$	$\Theta(N^2)$	$\Theta(N)$	Yes	**2x or more slower than merge sort.**
Java Quicksort	$\Theta(N)$	$\Theta(N^2)$	$O(\log N)$	No	**Fastest comparison sort.**