Lecture 33 (Sorting 3)

Quicksort

CS61B, Spring 2025 @ UC Berkeley

Slides credit: Josh Hug

Quicksort

Lecture 32, CS61B, Spring 2025

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$	Link	Bad cache (61C) performance.
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Fastest of these.
Insertion Sort (in place)	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Link	Best for small N or almost sorted.

See this link for bonus slides on Shell's Sort, an optimization of insertion sort.

The Core Idea of Tony's Sort: Partitioning

A **partition** of an array, given a **pivot** x, is a rearrangement of the items so that:

- All entries to the left of x are <= x.
- All entries to the right of x are >= x.

Called the 'pivot'.

		i				
5	550	10	4	10	9	330

A.

			j			
4	5	9	10	10	330	550

B.

			j			
5	4	9	10	10	550	330

C.

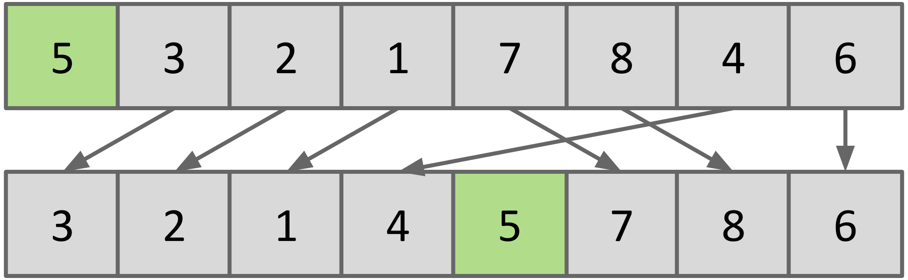
				j		
5	9	10	4	10	550	330

D.

		j				
5	9	10	~~4~~	10	550	330

Which partitions are valid?

Partition Sort, a.k.a. Quicksort



Q: How would we use this operation for sorting?

Observations:

- 5 is "in its place." Exactly where it'd be if the array were sorted.
- Can sort two halves separately, e.g. through recursive use of partitioning.

Job Interview Style Question (Partitioning)

Given an array of colors where the 0th element is white (and maybe a few more), and the remaining elements are red (less) or blue (greater), rearrange the array so that all red squares are to the left of the white square, the white squares end up together, and all blue squares are to the right. Your algorithm must complete in Θ(N) time (no space restriction).

- Relative order of red and blues does NOT need to stay the same.

Input

6	8	3	1	2	7	4	6

Example of a valid output

3	1	2	4	6	6	8	7

Another example of a valid output

3	4	1	2	6	6	7	8

Quick Sort

Quick sorting N items:

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.

unsorted

Input:

32	15	2	17	19	26	41	17	17

Quick Sort

Quick sorting N items:

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

partition(32)

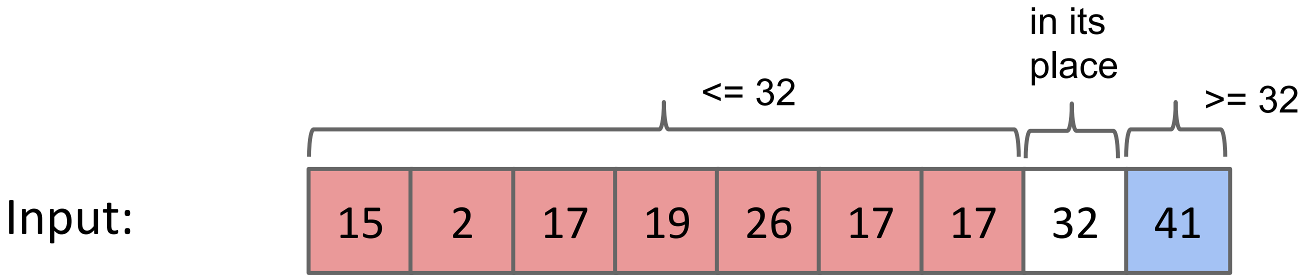
Input:	32	15	2	17	19	26	41	17	17

Quick Sort

Quick sorting N items:

- **Partition on leftmost item (32).**
- Quicksort left half.
- Quicksort right half.

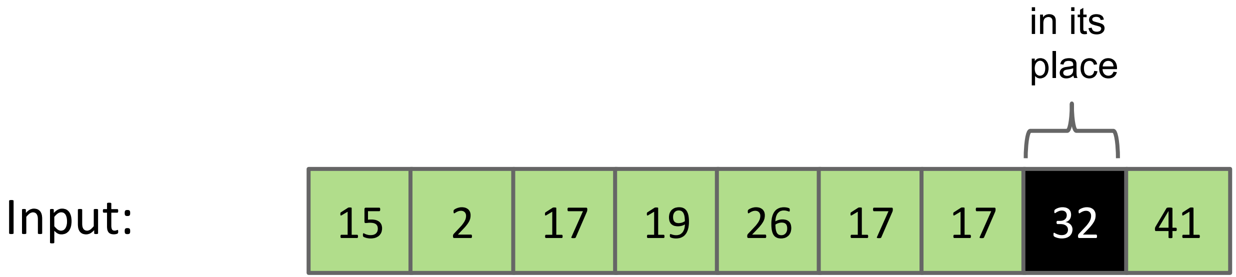
partition(32)



Quick sorting N items:

- **Partition on leftmost item (32) (done).**
- Quicksort left half.
- Quicksort right half.

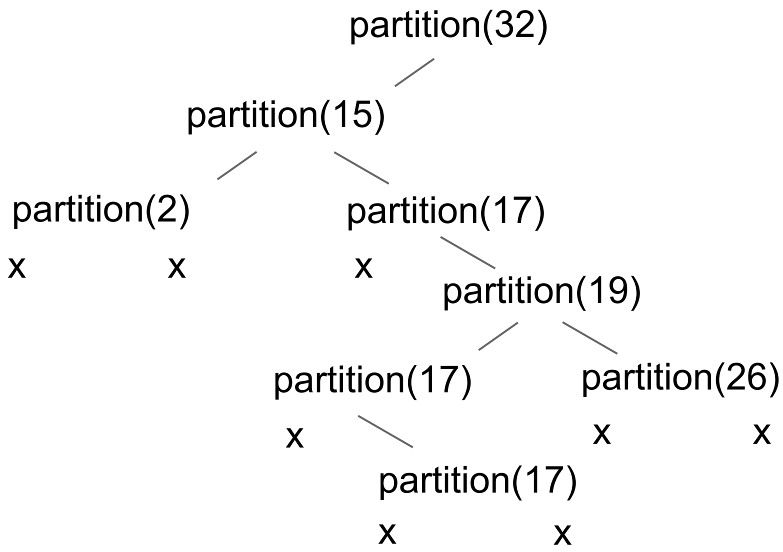
partition(32)



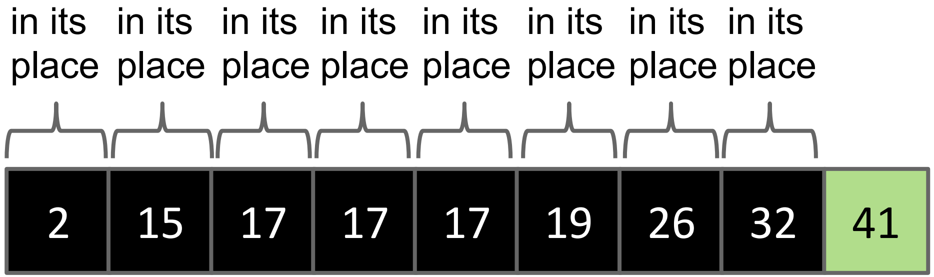
Quick Sort

Quick sorting N items:

- Partition on leftmost item (32) (done).
- **Quicksort left half (details not shown).**
- Quicksort right half.



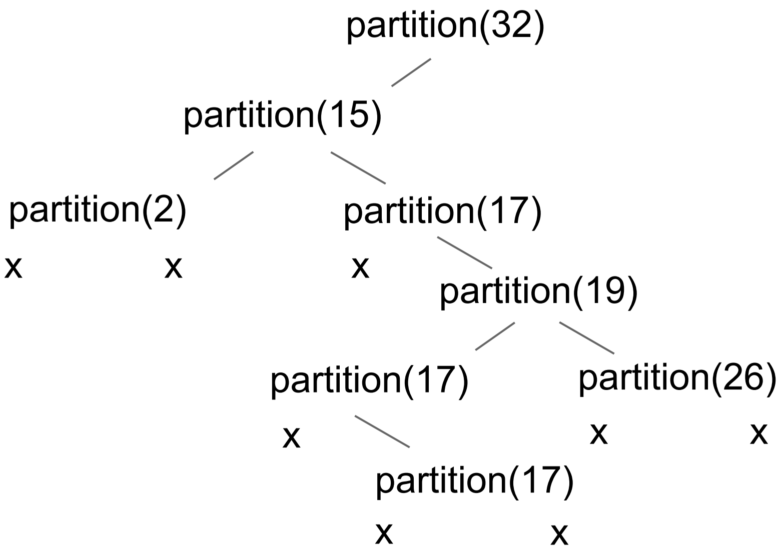
Input:



Quick Sort

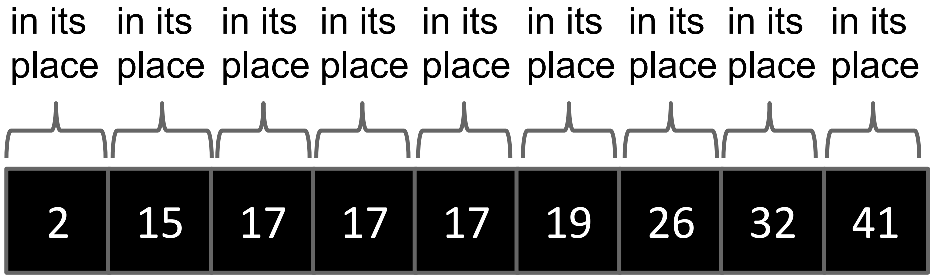
Quick sorting N items:

- Partition on leftmost item (32) (done).
- Quicksort left half (details not shown).
- **Quicksort right half (details not shown).**



If you don't fully trust the recursion, see [these extra slides] for a complete demo.

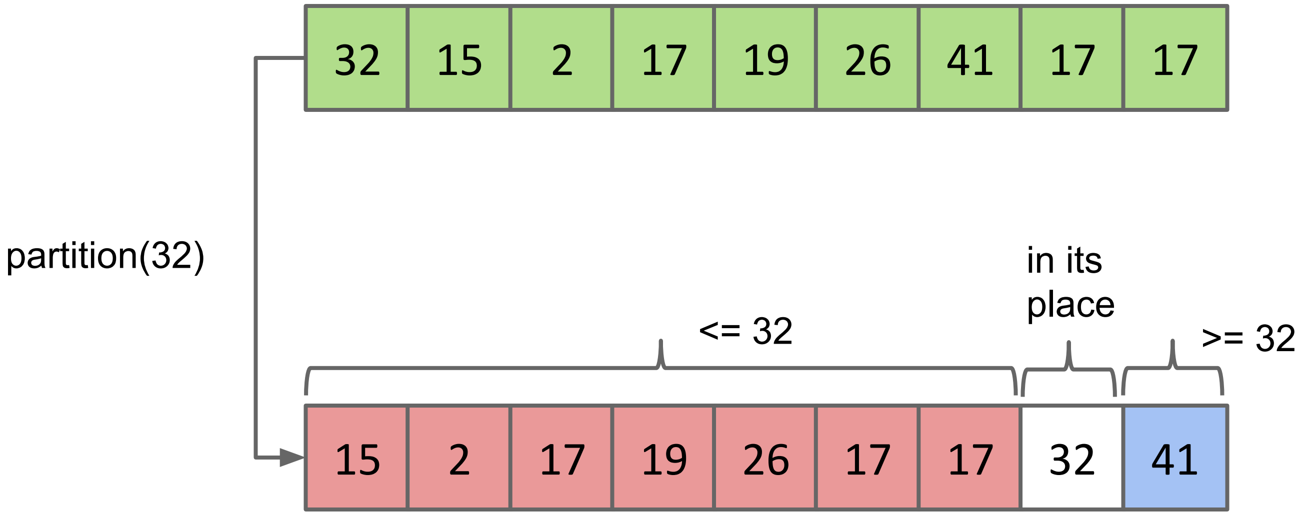
Input:



Partition Sort, a.k.a. Quicksort

Quick sorting N items:

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.



Quicksort

Quicksort was the name chosen by Tony Hoare for partition sort.

- For most common situations, it is empirically the fastest sort.
  - Tony was lucky that the name was correct.

How fast is Quicksort? Need to count number and difficulty of partition operations.

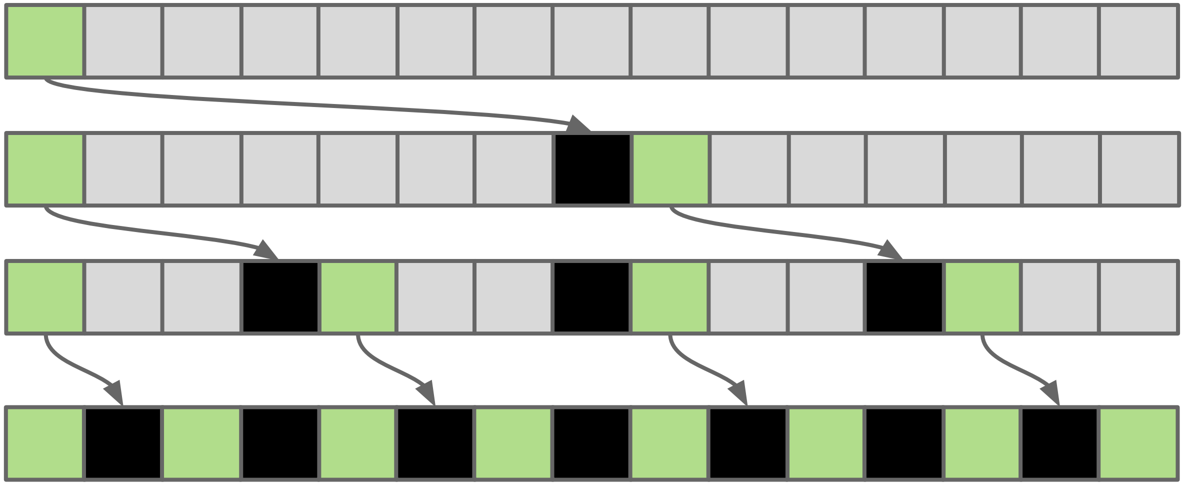
Theoretical analysis:

- Partitioning costs Θ(K) time, where Θ(K) is the number of elements being partitioned (as we saw in our earlier “interview question”).
- The interesting twist: Overall runtime will depend crucially on where pivot ends up.

Quicksort Runtime Analysis

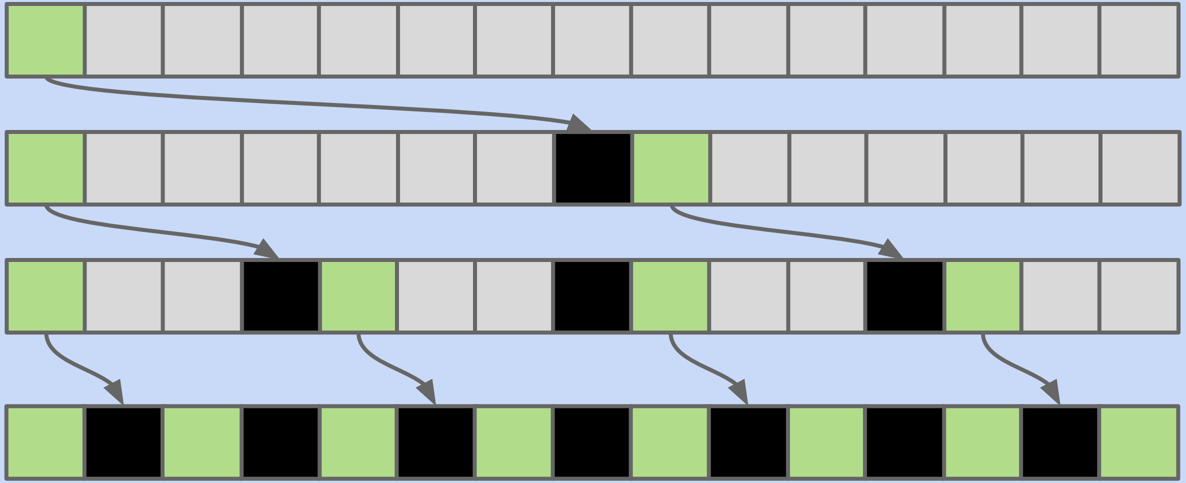
Lecture 32, CS61B, Spring 2025

Best Case: Pivot Always Lands in the Middle



Only size 1 problems remain, so we're done.

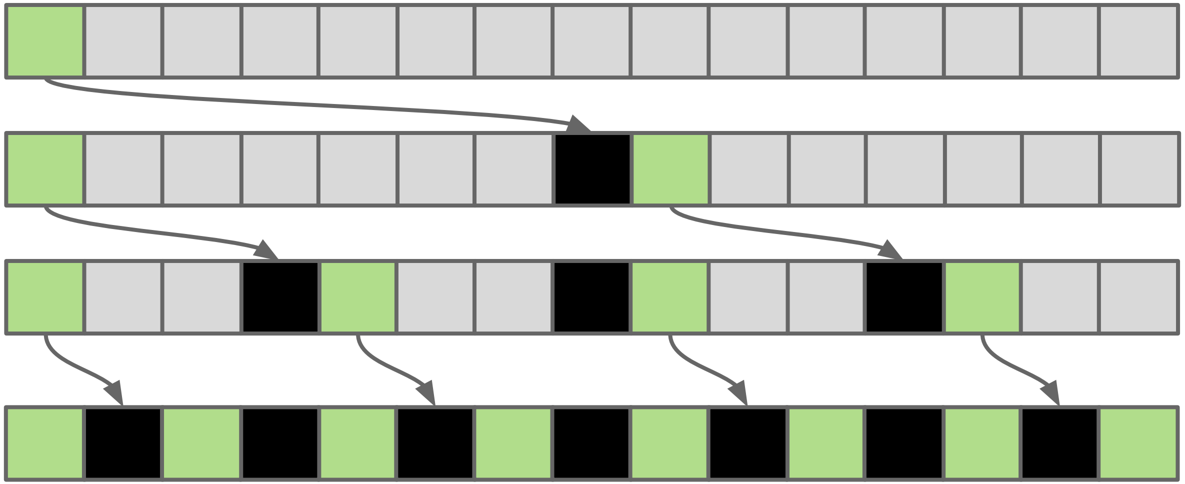
Best Case Runtime?



Only size 1 problems remain, so we're done.

What is the best case runtime?

Best Case Runtime?



Only size 1 problems remain, so we're done.

Total work at each level:

$\approx N$

$\approx N/2 + \approx N/2 = \approx N$

$\approx N/4 * 4 = \approx N$

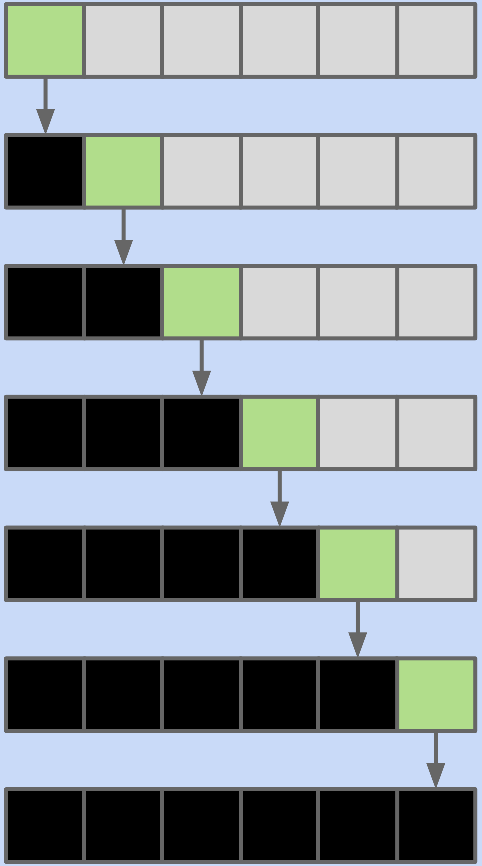
Overall runtime:
$\Theta(NH)$ where $H = \Theta(\log N)$

so: $\Theta(N \log N)$

Worst Case: Pivot Always Lands at Beginning of Array

Give an example of an array that would follow the pattern to the right.

What is the runtime $\Theta(\cdot)$?



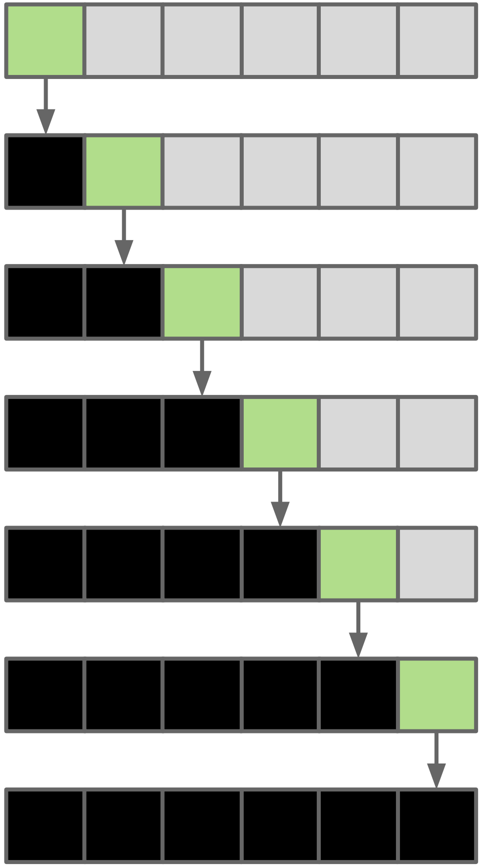
Worst Case: Pivot Always Lands at Beginning of Array

Give an example of an array that would follow the pattern to the right.

- 1 2 3 4 5 6

What is the runtime Θ(·)?

- N^2



Quicksort Performance

Theoretical analysis:

- Best case: Θ(N log N)
- Worst case: $\Theta(N^2)$

Compare this to Mergesort.

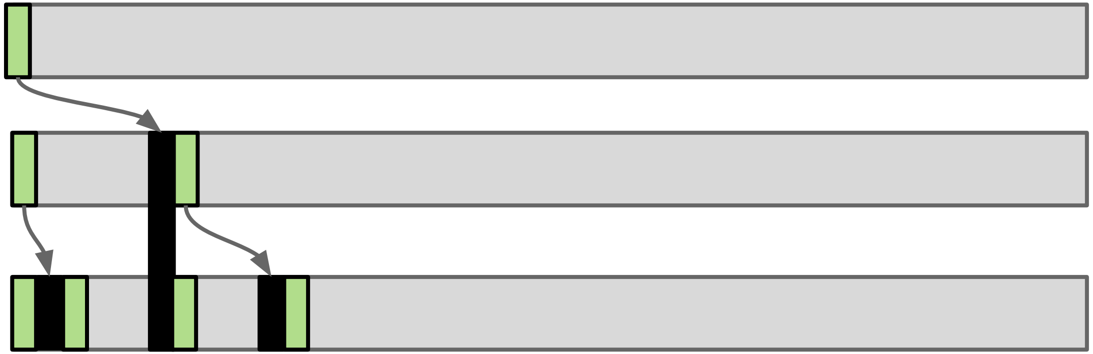
- Best case: Θ(N log N)
- Worst case: Θ(N log N)

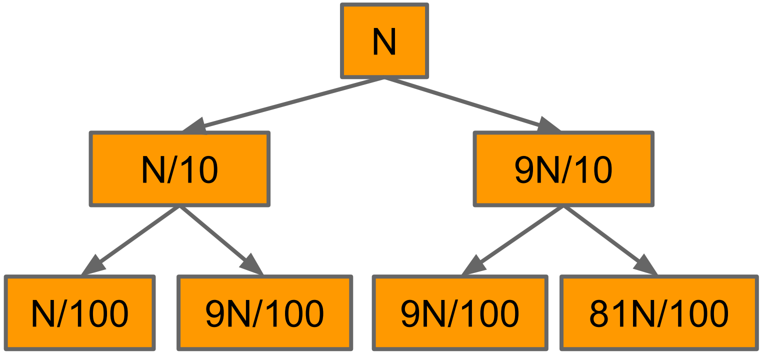
Recall that Θ(N log N) vs. $\Theta(N^2)$ is a **<u>really big deal</u>**. So how can Quicksort be the fastest sort empirically? Because on average it is Θ(N log N).

- Rigorous proof requires probability theory + calculus, but intuition + empirical analysis will hopefully convince you.

Argument #1: 10% Case

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



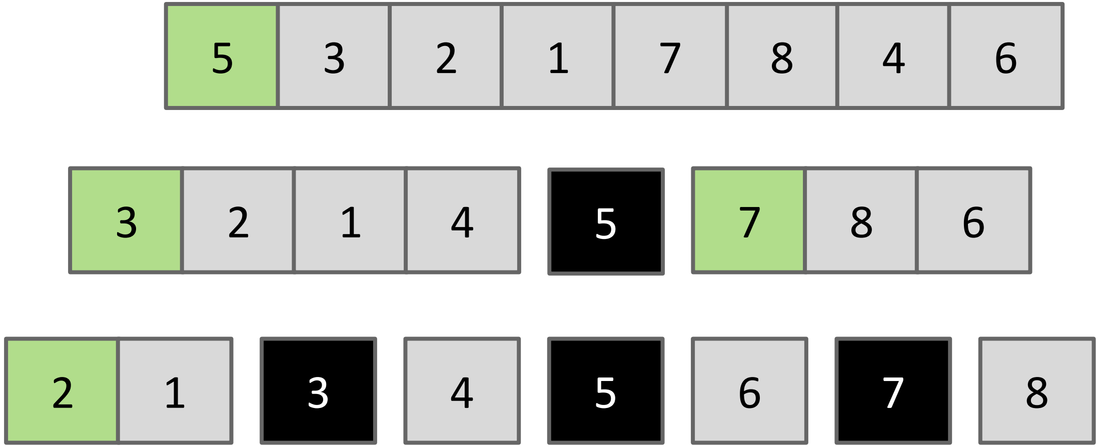


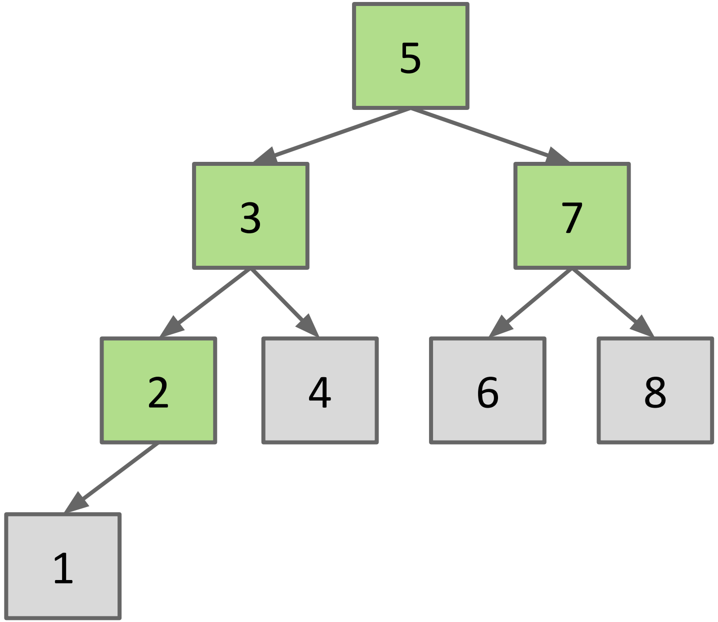
Work at each level: O(N)

- Runtime is O(NH).
  - H is approximately $\log_{10/9} N = O(\log N)$
- Overall: O(N log N).

Punchline: Even if you are unlucky enough to have a pivot that never lands anywhere near the middle, but at least always 10% from the edge, runtime is still O(N log N).

Argument #2: Quicksort is BST Sort





Key idea: compareTo calls are same for BST insert and Quicksort.

- Every number gets compared to 5 in both.
- 3 gets compared to only 1, 2, 4, and 5 in both.

Reminder: Random insertion into a BST takes O(N log N) time.

Empirical Quicksort Runtimes

For N items:

- Mean number of compares to complete Quicksort: ~2N ln N
- Standard deviation:

$$\sqrt{(21 - 2\pi^2)/3}N \approx 0.6482776N$$

Lots of arrays take 12,000ish compares to sort with Quicksort.

A very small number take 15,000ish compares to sort with Quicksort.

Empirical histogram for quicksort compare counts (10,000 trials with N = 1000)

Chance of taking 1,000,000ish compares is effectively zero.

For more, see: http://www.informit.com/articles/article.aspx?p=2017754&seqNum=7

Quicksort Performance

Theoretical analysis:

For our pivot/partitioning strategies: Sorted or close to sorted.

With extremely high probability!!

- Best case: $\Theta(N \log N)$
- Worst case: $\Theta(N^2)$
- **Randomly chosen array case: $\Theta(N \log N)$ expected**

Compare this to Mergesort.

- Best case: $\Theta(N \log N)$
- Worst case: $\Theta(N \log N)$

Why is it faster than mergesort?

- Requires empirical analysis. No obvious reason why.

Sorting Summary (so far)

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.

Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)
Insertion	$\Theta(1)$	$\Theta(N^2)$	$\Theta(N)$ if almost sorted
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	
Quicksort	$\Theta(\log N)$ (call stack)	$\Theta(N \log N)$ expected	Fastest sort

More Quicksort Origins

Amusingly, Quicksort was the wrong tool for the job. Two issues:

- Language that Tony was using didn't support recursion (so he couldn't easily implement Quicksort).
- Sentences are usually shorter than 15 words (So insertion sort is faster).

Tony Hoare 5/13/13

to jhug

You are quite right. But I am lucky that I sis not realise it at that time.

Remember, machines were then a million times slower than they are now.

Yours,

Tony.

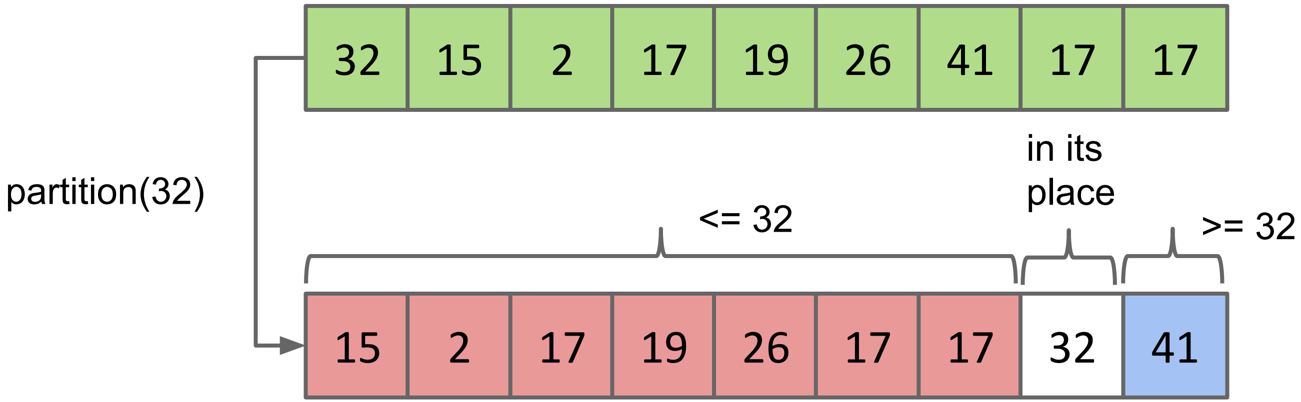
Philosophies for Avoiding Worst Case Quicksort Behavior

Lecture 32, CS61B, Spring 2025

Partition Sort, a.k.a. Quicksort

Quicksorting N items: (Demo)

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.



Run time is Θ(N log N) in the best case, $\Theta(N^2)$ in the worst case, and Θ(N log N) in the average case.

Avoiding the Worst Case: Question from Last Time

If pivot always lands somewhere “good”, Quicksort is Θ(N log N). However, the very rare Θ(N^2) cases do happen in practice, e.g.

What specific array inputs yield bad Quicksort runtime?

Recall, our version of Quicksort has the following properties:

- Leftmost item is always chosen as the pivot.
- Our partitioning algorithm preserves the relative order of <= and >= items.

6	8	3	1	2	7	4

3	1	2	4	6	8	7

Avoiding the Worst Case: Question from Last Time

If pivot always lands somewhere "good", Quicksort is Θ(N log N). However, the very rare Θ(N^2) cases do happen in practice, e.g.

- Bad ordering: Array already in sorted order (or almost sorted order).
- Bad elements: Array with all duplicates (when using Hoare Partitioning; see next section).

What can we do to avoid worst case behavior?

Recall, our version of Quicksort has the following properties:

- Leftmost item is always chosen as the pivot.
- Our partitioning algorithm preserves the relative order of <= and >= items.

6	8	3	1	2	7	4

3	1	2	4	6	8	7

Avoiding the Worst Case: My Answers

What can we do to avoid running into the worst case for QuickSort?

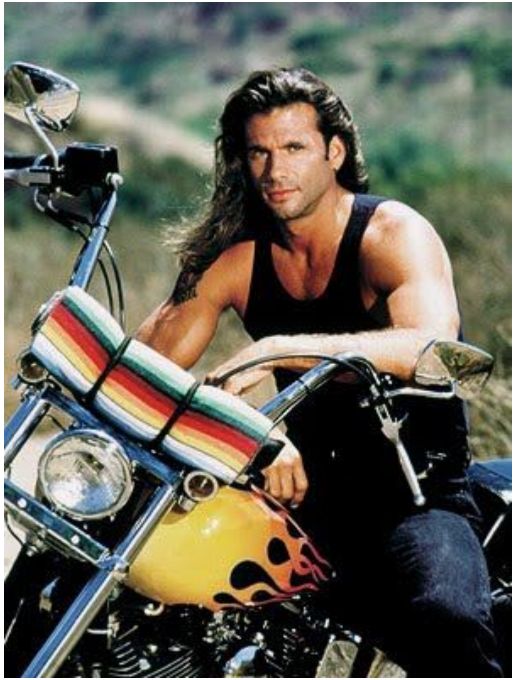
Four philosophies:

1. **Randomness**: Pick a random pivot or shuffle before sorting.
2. **Smarter pivot selection**: Calculate or approximate the median.
3. **Introspection**: Switch to a safer sort if recursion goes to deep.
4. **Preprocess the array**: Could analyze array to see if Quicksort will be slow. No obvious way to do this, though (can’t just check if array is sorted, almost sorted arrays are almost slow).

Philosophy 1: Randomness (My Preferred Approach)

If pivot always lands somewhere "good", Quicksort is $\Theta(N \log N)$. However, the very rare $\Theta(N^2)$ cases do happen in practice, e.g.

- Bad ordering: Array already in sorted order.
- Bad elements: Array with all duplicates.



Dealing with bad ordering:

- Strategy #1: Pick pivots randomly.
- Strategy #2: Shuffle before you sort.

The second strategy requires care in partitioning code to avoid $\Theta(N^2)$ behavior on arrays of duplicates.

- Common bug, even in a well known 2010s textbook.

Philosophy 2a: Smarter Pivot Selection (constant time pivot pick)

Randomness is necessary for best Quicksort performance! For any pivot selection procedure that is:

- Deterministic
- Constant Time

The resulting Quicksort has a family of dangerous inputs that an adversary could easily generate.

- See McIlroy's "A Killer Adversary for Quicksort"

7	2	3	4	5	6	1	8

Dangerous input

Philosophy 2b: Smarter Pivot Selection (linear time pivot pick)

Could calculate the actual median in linear time.

- "Exact median Quicksort" is safe: Worst case $\Theta(N \log N)$, but it is slower than Mergesort.

Raises interesting question though: How do you compute the median of an array? Will talk about how to do this later today.

Philosophy 3: Introspection

Can also simply watch your recursion depth.

- If it exceeds some critical value (say 10 ln N), switch to mergesort.

Perfectly reasonable approach, though not super common in practice.

Sorting Summary (so far)

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.

Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)
Insertion	$\Theta(1)$	$\Theta(N^2)$	$\Theta(N)$ if almost sorted
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	
Random Quicksort	$\Theta(\log N)$ expected	$\Theta(N \log N)$ expected	Fastest sort

Hoare Partitioning

Lecture 32, CS61B, Spring 2025

Quicksort Flavors

We said Quicksort is the fastest, but this is only true if we make the right decisions about:

- Pivot selection.
- Partition algorithm.
- How we deal with avoiding the worst case (can be covered by the above choices).

We'll call this **Quicksort L3S**

Suppose we run a speed test of Mergesort vs. Quicksort from last time, which had:

- Pivot selection: Always use **leftmost**.
- Partition algorithm: Make an array copy then do **three** scans for red, white, and blue items.
- **Shuffle** before starting (to avoid worst case).

Quicksort vs. Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints
Mergesort	N/A	N/A	N/A	1.3 seconds
Quicksort **L3S**	**Leftmost**	**3-scan**	**Shuffle**	3.2 seconds

Quicksort didn’t do so well!

Note: These are unoptimized versions of mergesort and quicksort, i.e. no switching to insertion sort for small arrays.

Tony Hoare’s In-place Partitioning Scheme

Tony originally proposed a scheme where two pointers walk towards each other.

- Left pointer loves small items.
- Right pointer loves large items.
- Big 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”.

(Note: The demo we'll show is not the exact scheme Tony used)

Using this partitioning scheme yields a very fast Quicksort.

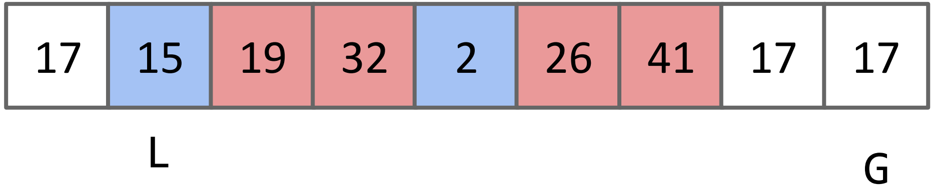
- Though faster schemes have been found since.
- Overall runtime still depends crucially on pivot selection strategy!

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

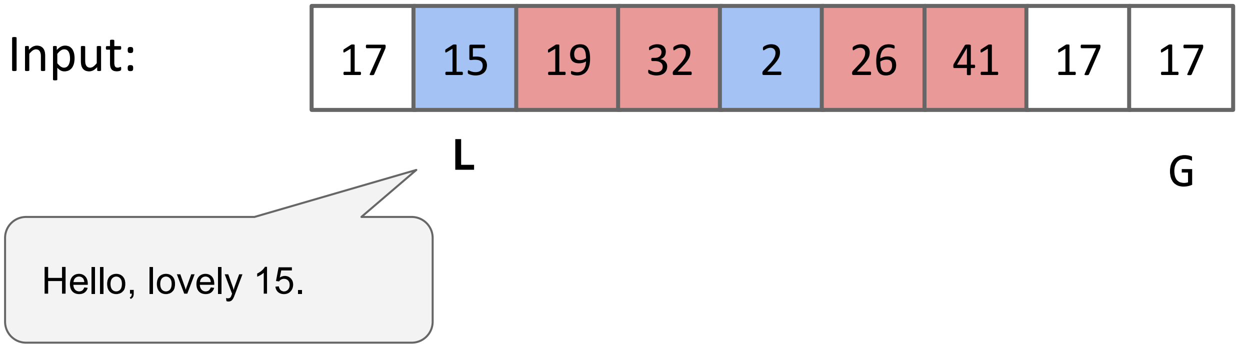
Input:



Hoare Partitioning

Create L and G pointers at left and right ends.

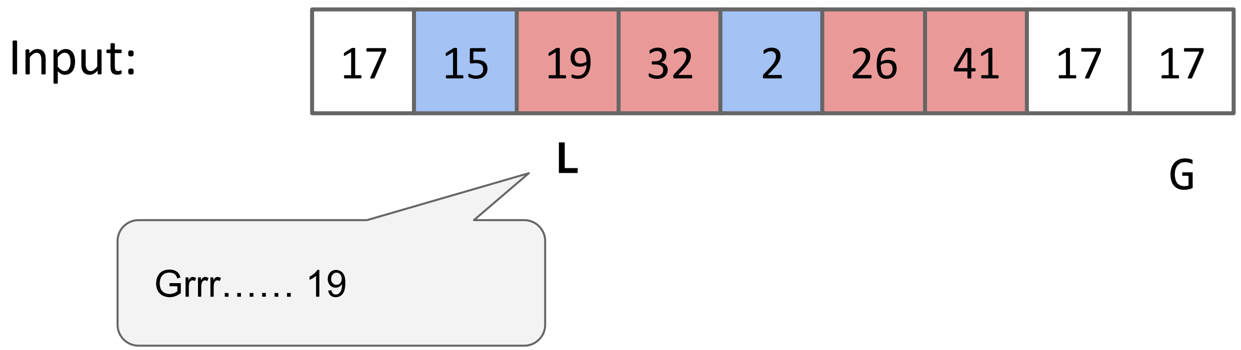
- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other**, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

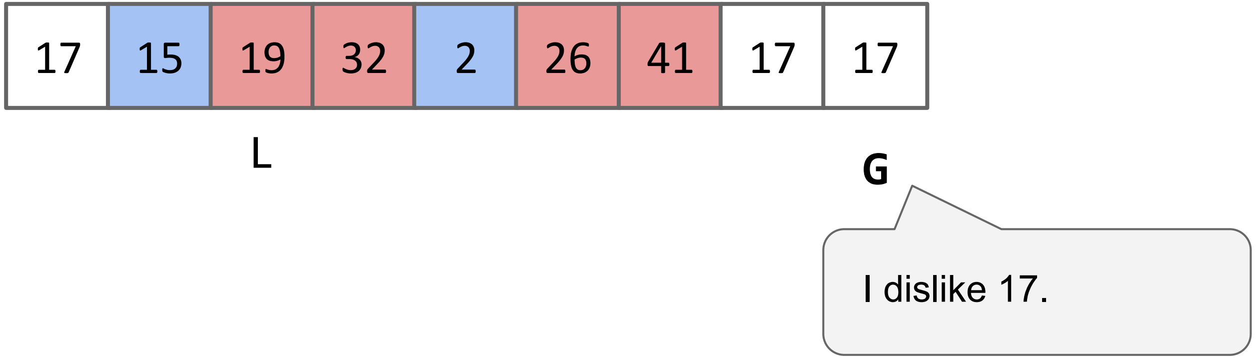


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap.

Input:

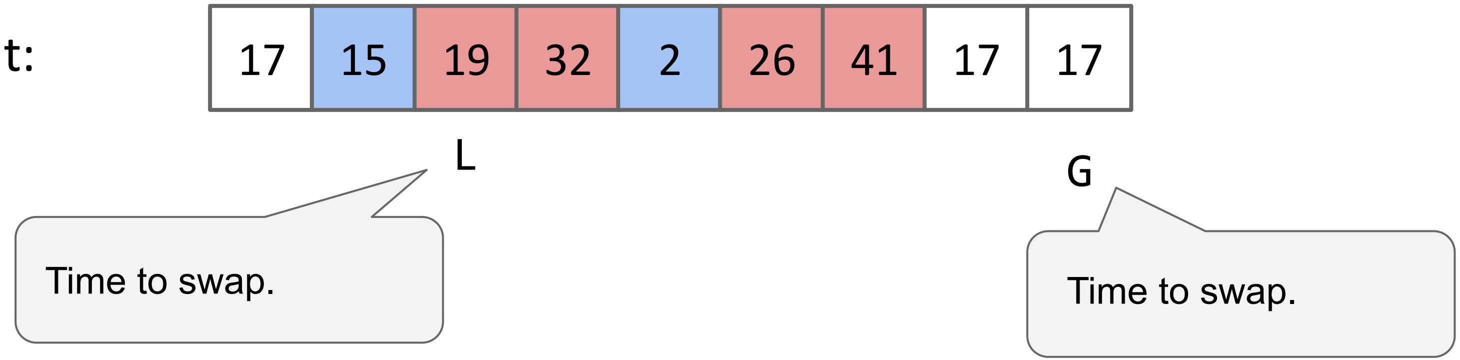


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
    - **When both pointers have stopped, swap and move pointers by one.**
- When pointers cross, you are done.

Input:



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - **When both pointers have stopped, swap and move pointers by one.**
- When pointers cross, you are done.

Input:

17	15	17	32	2	26	41	17	19

L — Swapped.

G — Swapped.

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
    - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

Input:

17	15	17	32	2	26	41	17	19
			L				G	

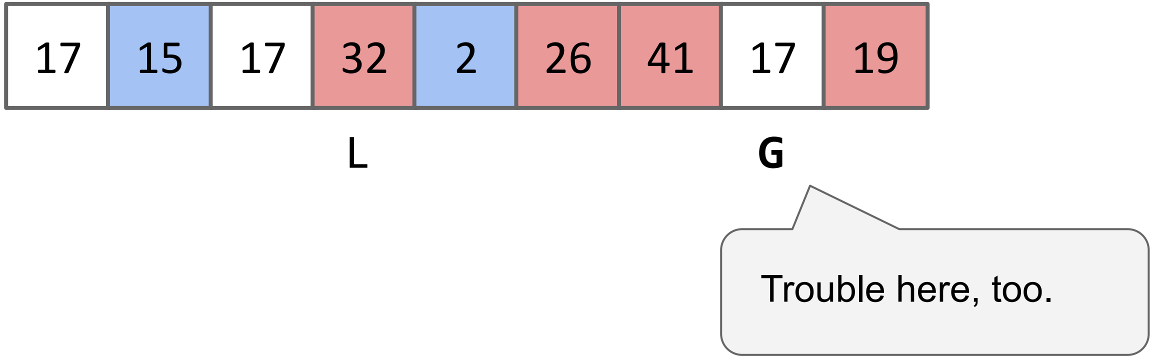
Immediate trouble.

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

Input:

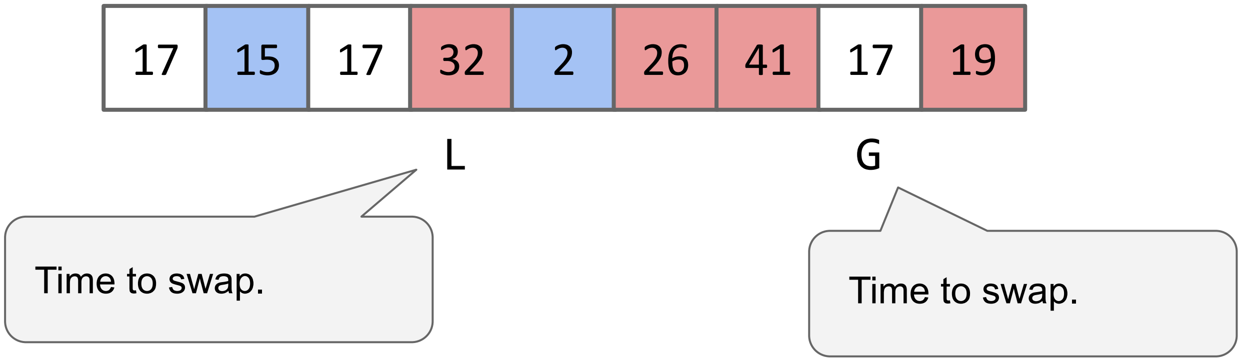


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - **When both pointers have stopped, swap and move pointers by one.**
  - When pointers cross, you are done walking.
- Swap pivot with G.

Input:

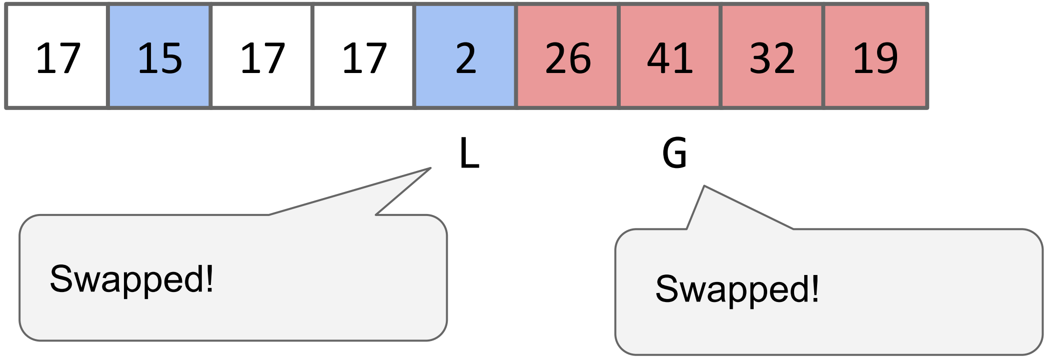


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - **When both pointers have stopped, swap and move pointers by one.**
  - When pointers cross, you are done walking.
- Swap pivot with G.

Input:

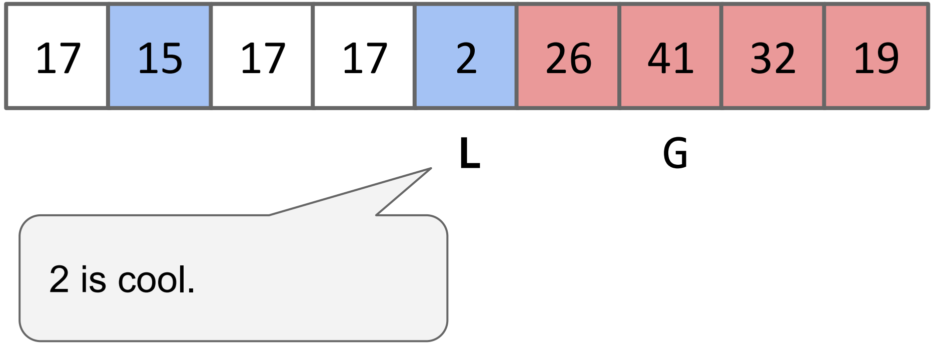


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- Swap pivot with G.

Input:

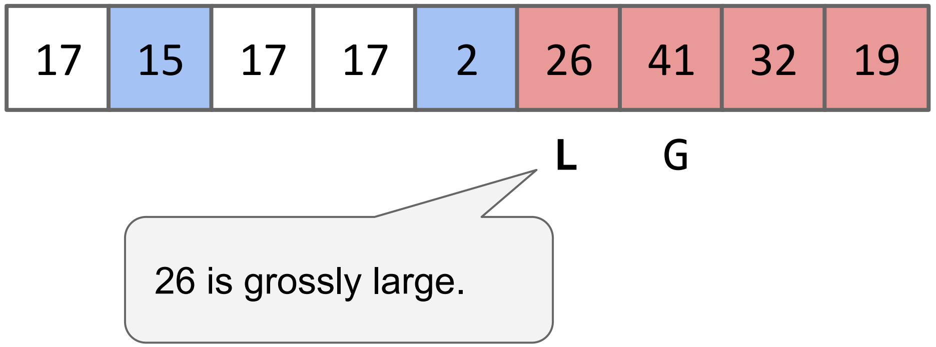


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- Swap pivot with G.

Input:

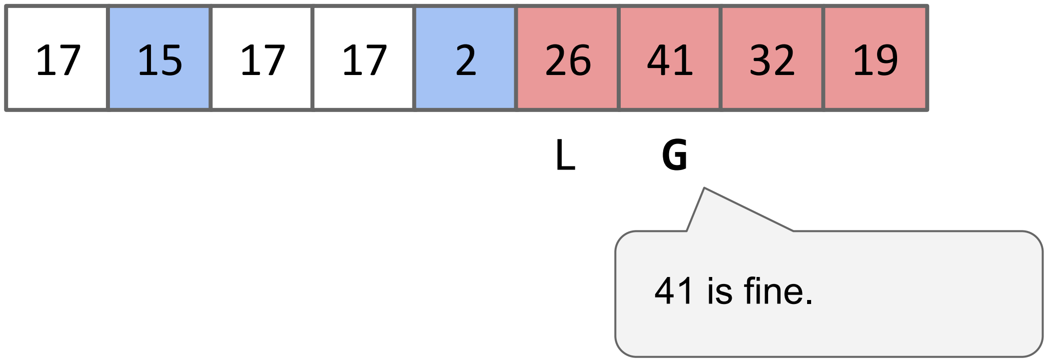


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- Swap pivot with G.

Input:

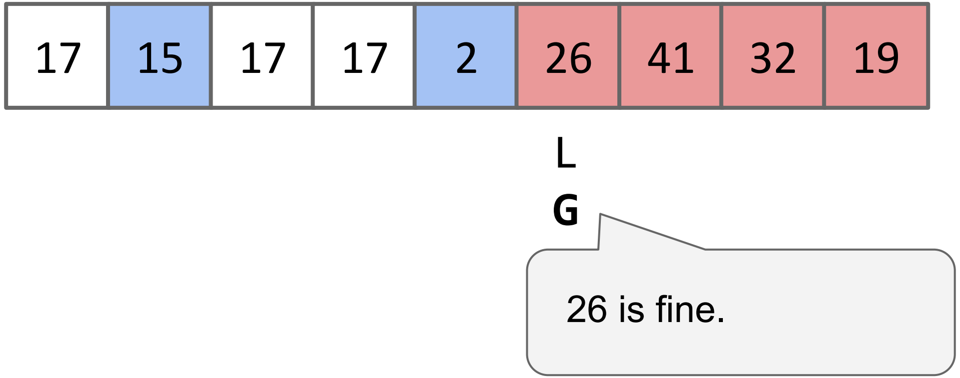


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- Swap pivot with G.

Input:

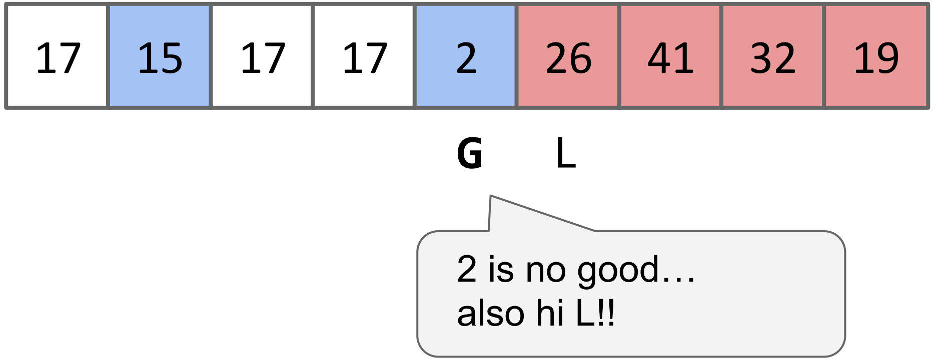


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- Swap pivot with G.

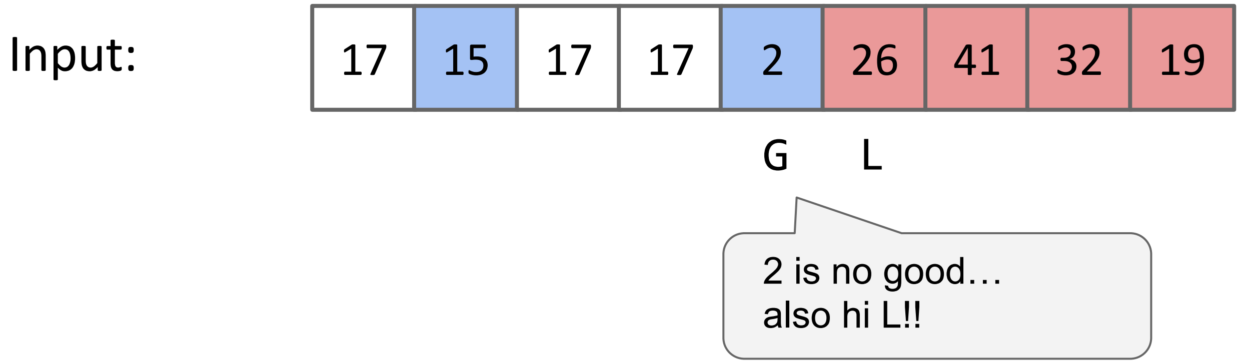
Input:



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
  - **When pointers cross, you are done walking.**
- Swap pivot with G.

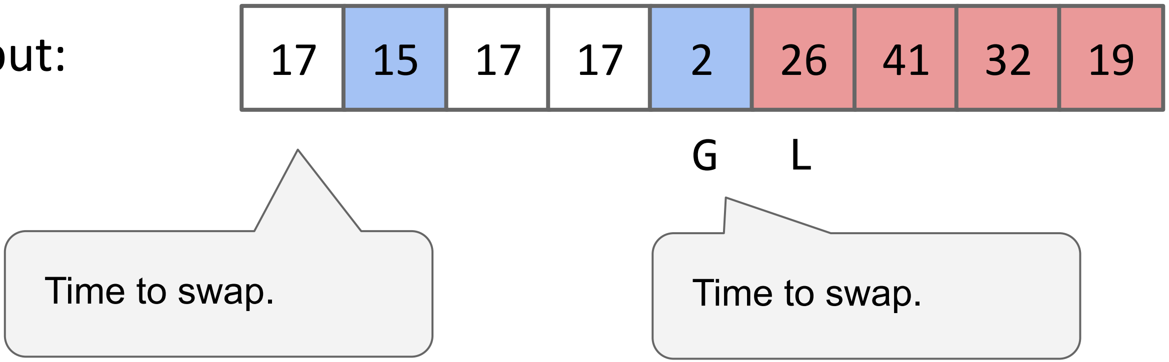


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- **Swap pivot with G.**

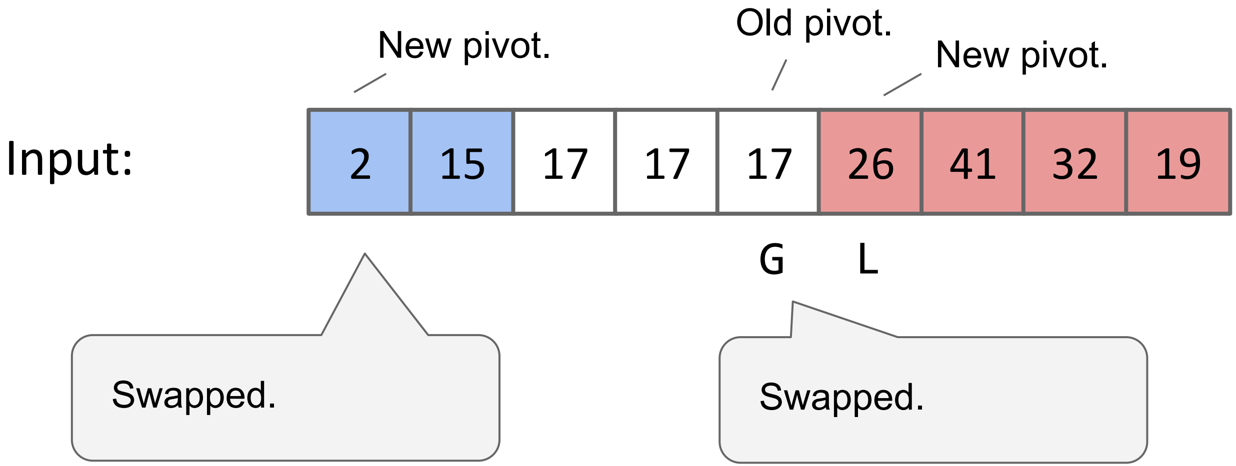
Input:



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- **Swap pivot with G.**



Quicksort vs. Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints
Mergesort	N/A	N/A	N/A	1.3 seconds
Quicksort **L3S**	**Leftmost**	**3-scan**	**Shuffle**	3.2 seconds
Quicksort **LTHS**	**Leftmost**	**Tony Hoare**	**Shuffle**	0.9 seconds

Using Tony Hoare's two pointer scheme, Quicksort is better than mergesort!

- More recent pivot/partitioning schemes do somewhat better.
  - Best known Quicksort uses a two-pivot scheme.
  - Interesting note, this version of Quicksort was introduced to the world by a previously unknown guy, in a Java developers forum (Link).

Note: These are unoptimized versions of mergesort and quicksort, i.e. no switching to insertion sort for small arrays.

Quick Select (median finding)

Lecture 32, CS61B, Spring 2025

What If We Don’t Want Randomness?

Our approach so far: Use randomness to avoid worst case behavior, but some people don’t like having randomness in their sorting routine.

Another approach: Use the median (or an approximation) as our pivot.

Four philosophies:

This is what we’ve been using.

1. **Randomness**: Pick a random pivot or shuffle before sorting.
2. **Smarter pivot selection**: Calculate or approximate the median.
3. **Introspection**: Switch to a safer sort if recursion goes to deep.
4. **Try to cheat**: If the array is already sorted, don’t sort (this doesn’t work).

Philosophy 2a: Smarter Pivot Selection (linear time pivot pick)

The best possible pivot is the median.

- Splits problem into two problems of size N/2.

Obvious approach: Just calculate the actual median and use that as pivot.

- But how?

Goal: Come up with an algorithm for finding the median of an array. Bonus points if your algorithm takes linear time.

Philosophy 2a: Smarter Pivot Selection (linear time pivot pick)

Goal: Come up with an algorithm for finding the median of an array. Bonus points if your algorithm takes linear time.

- Sort the array, return the middle item. Takes Θ(N log N) time to sort, though.

The Selection Problem

Computing the exact median would be great for picking an item to partition around. Gives us a “safe quick sort”.

However, it turns out that partitioning can be used to find the exact median.

- The resulting algorithm is the best known median identification algorithm.

Quick Select

Goal, find the median:

9	550	14	6	10	5	330	817	913

Partition, pivot lands at 2.

- Not the median. Why?
- So what next? Partition right subproblem, median can't be to the left!

6	5	9	550	14	10	330	817	913

			550	14	10	330	817	913

Now pivot lands at 6.

- Not the median.

			14	10	330	550	817	913

Pivot lands at 4. Are we done?

- Yep, 9/2 = 4.

			14	10	330			

			10	14	330			

Worst case performance?

What is the worst case performance for Quick Select? Give an array that causes this worst case (assuming we always pick leftmost item as pivot).

Worst case performance?

What is the worst case performance for Quick Select? Give an array that causes this worst case (assuming we always pick leftmost item as pivot).

Worst asymptotic performance $\Theta(N^2)$ occurs if array is in sorted order.

[1 2 3 4 5 6 7 8 9 10 … N]

[1 **2 3 4 5 6 7 8 9 10 … N**]

[1 2 **3 4 5 6 7 8 9 10 … N**]

…

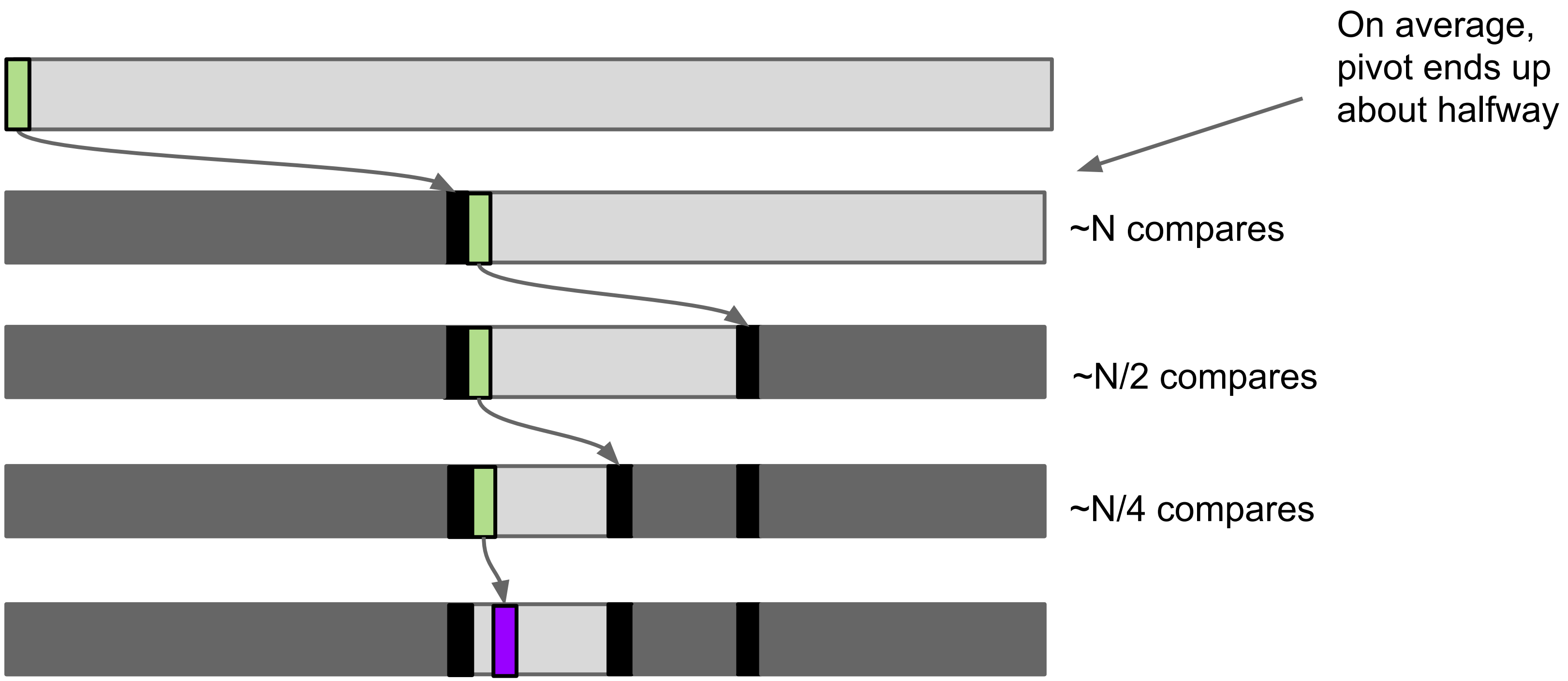
[1 2 3 4 5 … **N/2** … N]

Expected Performance

On average, Quick Select will take Θ(N) time.

- Intuitive picture (not a proof!):

$$N + N/2 + N/4 + \ldots + 1 = \Theta(N)$$



Quicksort With Quickselect?

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints	Worst Case
Mergesort	N/A	N/A	N/A	1.3 seconds	Θ(N log N)
Quicksort L3S	Leftmost	3-scan	Shuffle	3.2 seconds	$\Theta(N^2)$
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	0.9 seconds	$\Theta(N^2)$
Quicksort QSTH	QuickSelect	Tony Hoare	Exact Median	4.9 seconds	$\Theta(N^2)$

What if we used Quickselect to find the exact median?

- Resulting algorithm is very slow. Doesn't fix the worst case runtime issue
- A little strange to do a bunch of partitions to identify the optimal item to partition around.

Median Identification

Is it possible to find the median in Θ(N) time?

- Yes! Use '[BFPRT]' (called PICK in original paper).
- Algorithm developed in 1972
- In practice, rarely used.

Historical note: The authors of this paper include FOUR Turing Award winners (and Vaughan Pratt)

Time Bounds for Selection*

MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT,
RONALD L. RIVEST, AND ROBERT E. TARJAN

Department of Computer Science, Stanford University, Stanford, California 94305

Received November 14, 1972

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm—PICK. Specifically, no more than $5.430\dot{5}\, n$ comparisons are ever required. This bound is improved for extreme values of i, and a new lower bound on the requisite number of comparisons is also proved.

Let's see how Exact Median Quicksort performs.

Quicksort vs. Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints	Worst Case
Mergesort	N/A	N/A	N/A	1.3 seconds	$\Theta(N \log N)$
Quicksort L3S	Leftmost	3-scan	Shuffle	3.2 seconds	$\Theta(N^2)$
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	0.9 seconds	$\Theta(N^2)$
Quicksort PickTH	Exact Median	Tony Hoare	Exact Median	~15 seconds	$\Theta(N \log N)$

Quicksort using PICK to find the exact median (Quicksort PickTH) is terrible!

- Cost to compute medians is too high.
- Have to live with worst case $\Theta(N^2)$ if we want good practical performance.

Note: These are unoptimized versions of mergesort and quicksort, i.e. no switching to insertion sort for small arrays.