CS61B

Lecture 30: Sorting II: Quicksort

- Insertion Sort (Continued)
- Backstory, Partitioning
- Quicksort
- Quicksort Performance Caveats and Conclusion

Insertion Sort Runtime: http://yellkey.com/process

What is the runtime of insertion sort?

A. $\Omega(1)$, $O(N)$
B. $\Omega(N)$, $O(N)$
C. $\Omega(1)$, $O(N^2)$
D. $\Omega(N)$, $O(N^2)$
E. $\Omega(N^2)$, $O(N^2)$

36 swaps:

```
S O R T E X A M P L E
S O R T E X A M P L E  (0 swaps)
O S R T E X A M P L E  (1 swap )
O R S T E X A M P L E  (1 swap )
O R S T E X A M P L E  (0 swaps)
E O R S T X A M P L E  (4 swaps)
E O R S T X A M P L E  (0 swaps)
A E O R S T X M P L E  (6 swaps)
A E M O R S T X P L E  (5 swaps)
A E M O P R S T X L E  (4 swaps)
A E L M O P R S T X E  (7 swaps)
A E E L M O P R S T X  (8 swaps)
```

Insertion Sort Runtime

What is the runtime of insertion sort?

A. $\Omega(1)$, $O(N)$
B. $\Omega(N)$, $O(N)$
C. $\Omega(1)$, $O(N^2)$
D. $\Omega(N)$, $O(N^2)$
E. $\Omega(N^2)$, $O(N^2)$

You may recall Ω is not “best case”.

So technnnniically you could also say $\Omega(1)$

36 swaps:

```
S O R T E X A M P L E
S O R T E X A M P L E  (0 swaps)
O S R T E X A M P L E  (1 swap )
O R S T E X A M P L E  (1 swap )
O R S T E X A M P L E  (0 swaps)
E O R S T X A M P L E  (4 swaps)
E O R S T X A M P L E  (0 swaps)
A E O R S T X M P L E  (6 swaps)
A E M O R S T X P L E  (5 swaps)
A E M O P R S T X L E  (4 swaps)
A E L M O P R S T X E  (7 swaps)
A E E L M O P R S T X  (8 swaps)
```

Picking the Best Sort: http://yellkey.com/standard

Suppose we do the following:

- Read 1,000,000 integers from a file into an array of length 1,000,000.
- Mergesort these integers.
- Select one integer randomly and change it.
- Sort using algorithm X of your choice.

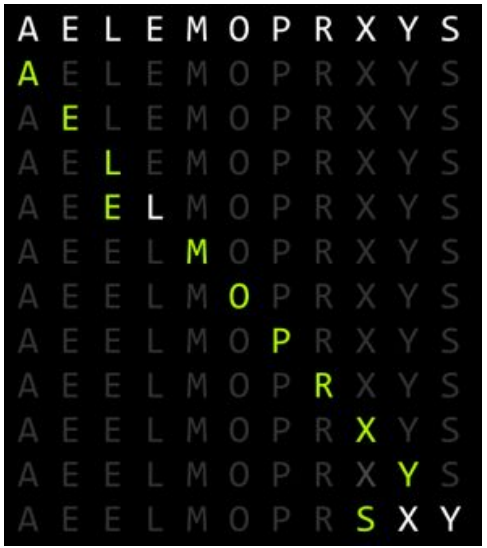
Which sorting algorithm would be the fastest choice for X?

A. Selection Sort: $O(N^2)$
B. Heapsort: O(N Log N)
C. Mergesort: O(N Log N)
D. Insertion Sort: $O(N^2)$

Observation: Insertion Sort on Almost Sorted Arrays

For arrays that are almost sorted, insertion sort does very little work.

- Left array: 5 inversions, so only 5 swaps.
- Right array: 3 inversion, so only 3 swaps.



Picking the Best Sort (Poll Everywhere)

Suppose we do the following:

- Read 1,000,000 integers from a file into an array of length 1,000,000.
- Mergesort these integers.
- Select one integer randomly and change it.
- Sort using algorithm X of your choice.
- In the worst case, we have 999,999 inversions: $\Theta(N)$ inversions.

Which sorting algorithm would be the fastest choice for X? Worst case run-times:

A. Selection Sort: $\Theta(N^2)$
B. Heapsort: $\Theta(N \log N)$
C. Mergesort: $\Theta(N \log N)$
D. Insertion Sort: $\Theta(N)$

Insertion Sort Sweet Spots

On arrays with a small number of inversions, insertion sort is extremely fast.

- One exchange per inversion (and number of comparisons is similar). Runtime is $\Theta(N + K)$ where K is number of inversions.
- Define an ***almost sorted*** array as one in which number of inversions $\leq cN$ for some c. Insertion sort is excellent on these arrays.

Less obvious: For small arrays ($N < 15$ or so), insertion sort is fastest.

- More of an empirical fact than a theoretical one.
- Theoretical analysis beyond scope of the course.
- Rough idea: Divide and conquer algorithms like heapsort / mergesort spend too much time dividing, but insertion sort goes straight to the conquest.
- The Java implementation of Mergesort does this (Link).

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.

Backstory, Partitioning

Sorting So Far

Core ideas:

- Selection sort: Find the smallest item and put it at the front.
  - Heapsort variant: Use MaxPQ to find max element and put at the back.
- Merge sort: Merge two sorted halves into one sorted whole.
- Insertion sort: Figure out where to insert the current item.

Quicksort:

- Much stranger core idea: Partitioning.
- Invented by Sir Tony Hoare in 1960, at the time a novice programmer.

Context for Quicksort's Invention (Source)

1960: Tony Hoare was working on a crude automated translation program for Russian and English.

"The cat wore a beautiful hat."

N words

...	...
beautiful	красивая
...	...
cat	кошка
...	...

Dictionary of D english words

"Кошка носил красивая шапка."

How would you do this?

- Binary search for each word.
  - Find "the" in log D time.
  - Find "cat" in log D time...
- Total time: N log D

Context for Quicksort's Invention (Source)

1960: Tony Hoare was working on a crude automated translation program for Russian and English.

Algorithm: N binary searches of D length dictionary.

- Total runtime: N log D
- ASSUMES log time binary search!

...	...
beautiful	красивая
...	...
cat	кошка
...	...

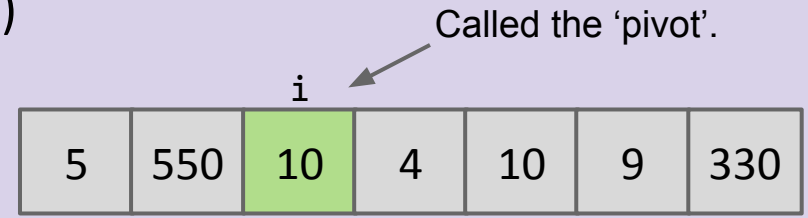
Limitation at the time:

- Dictionary stored on long piece of tape, sentence is an array in RAM.
  - Binary search of tape is not log time (requires physical movement!).
- Better: **Sort the sentence** and scan dictionary tape once. Takes N log N + D time.
  - But Tony had to figure out how to sort an array (without Google!)...

The Core Idea of Tony's Sort: Partitioning [no yellkey]

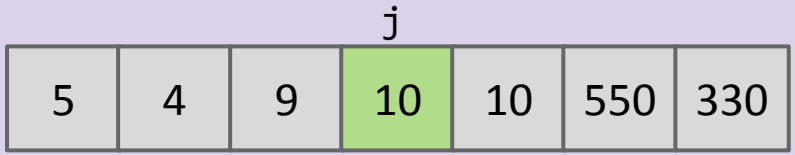
To partition an array a[] on element x=a[i] is to rearrange a[] so that:

- x moves to position j (may be the same as i)
- All entries to the left of x are <= x.
- All entries to the right of x are >= x.



A.

B.



C.

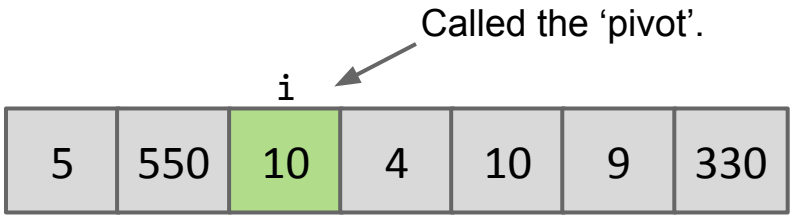
D.

Which partitions are valid?

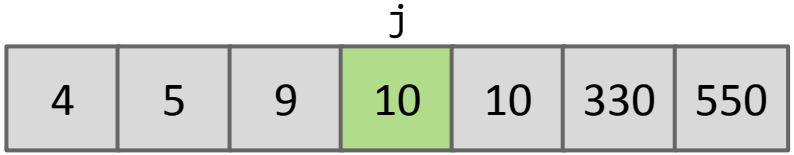
The Core Idea of Tony's Sort: Partitioning

To partition an array a[] on element x=a[i] is to rearrange a[] so that:

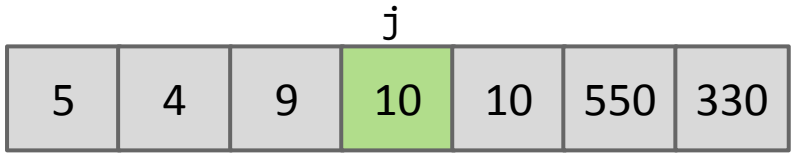
- x moves to position j (may be the same as i)
- All entries to the left of x are <= x.
- All entries to the right of x are >= x.



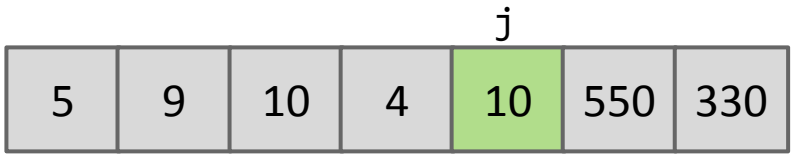
A.



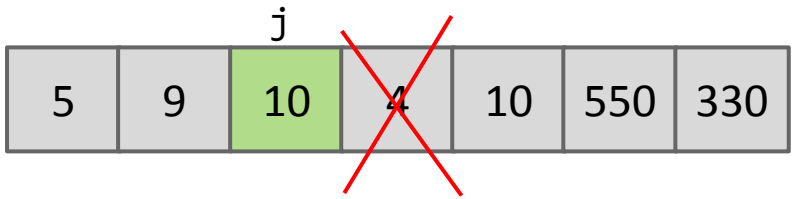
B.



C.



D.



Which partitions are valid?

Interview Question (Partitioning)

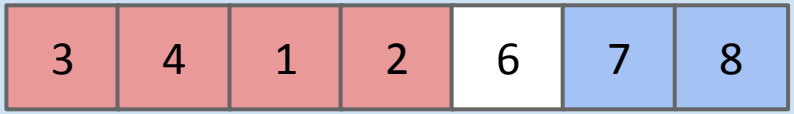
Given an array of colors where the 0th element is white, and the remaining elements are red (less) or blue (greater), rearrange the array so that all red squares are to the left of the white square, and all blue squares are to the right. Your algorithm must complete in Θ(N) time (no space restriction).

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

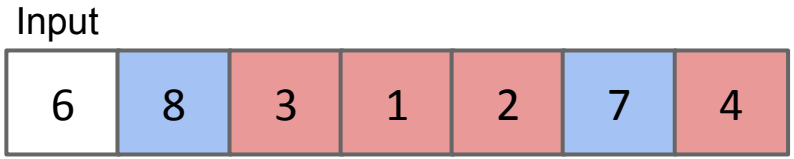
Input

Example of a valid output

Another example of a valid output



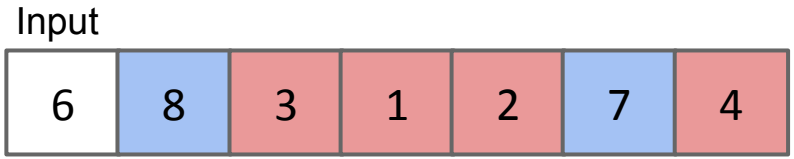
Interview Question, Student Answer #1



Scan from the right edge of the list. If anything is smaller, stick it in the leftmost position. Skip larger (blue) items.

- Very natural use for a double ended queue.
- Maybe I'll replace palindrome with this next semester.

Interview Question, Student Answer #1



Insert 6 into a BST. Then 8, then 3, then ..., then 4.

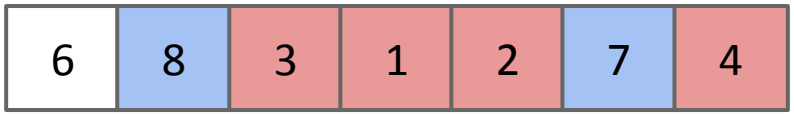
- All the small items are on the left.
- All the large items are on the right.

Simplest (but not fastest) Answer: 3 Scan Approach

Given an array of colors where the 0th element is white, 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, 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

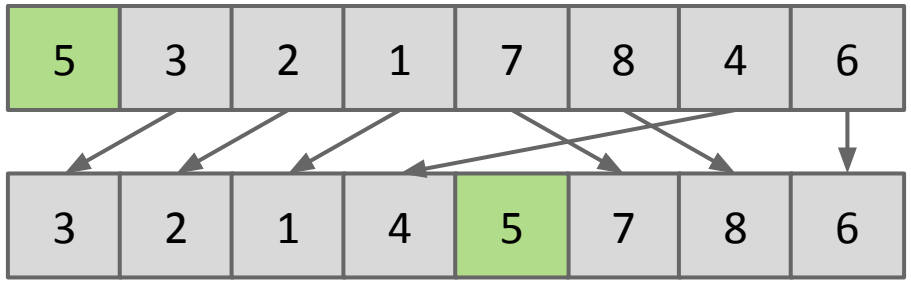


Algorithm: Create another array. Scan and copy all the red items to the first R spaces. Then scan for and copy the white item. Then scan and copy the blue items to the last B spaces.

Output

Quicksort

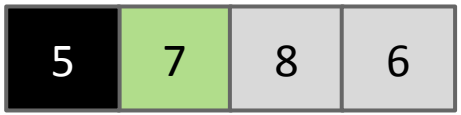
Partition Sort, a.k.a. Quicksort

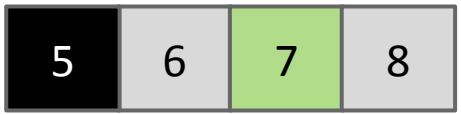


Q: How would we use this operation for sorting?

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.

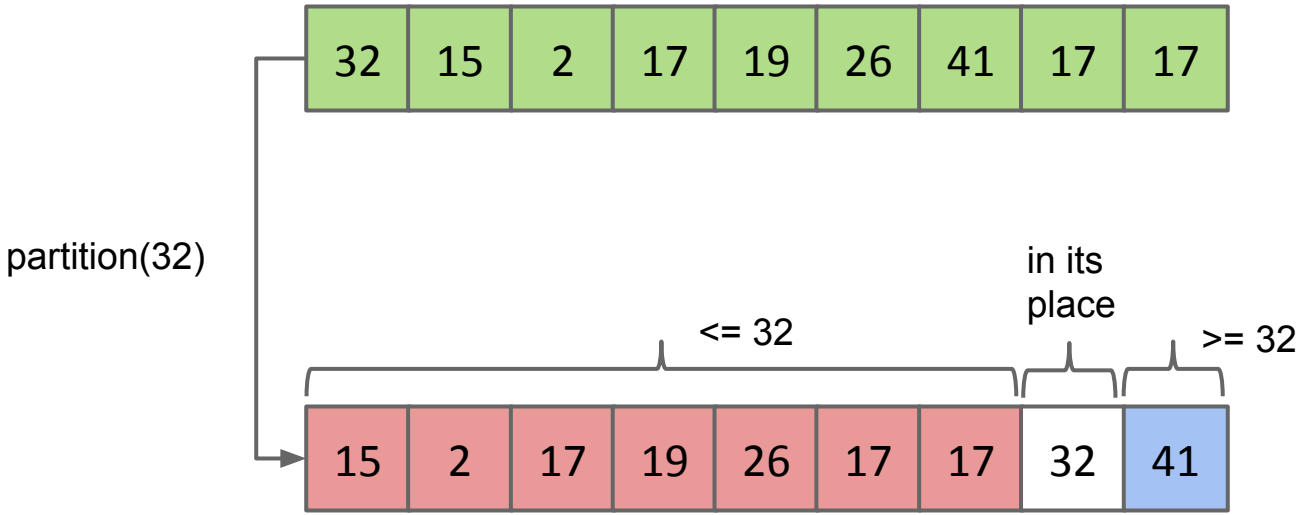




Partition Sort, a.k.a. Quicksort

Quicksorting N items: (Demo)

- 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.

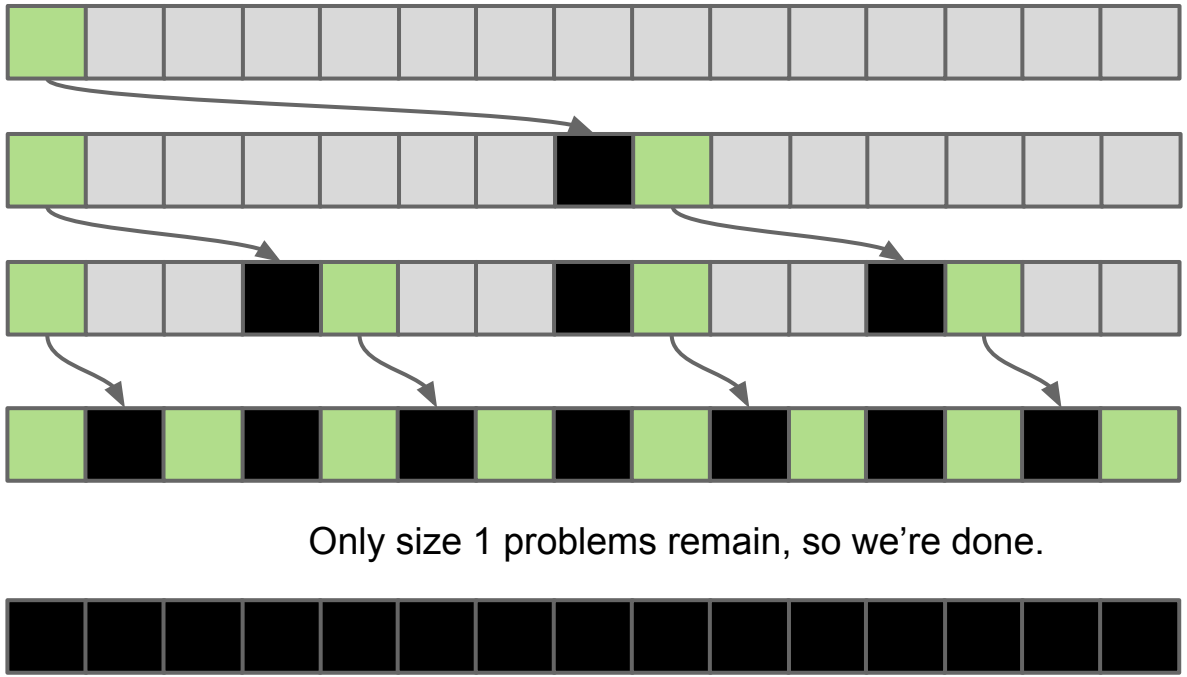
How fast is Quicksort? Need to count number and difficulty of partition operations.

Theoretical analysis:

- Partitioning costs $\Theta(K)$ time, where $\Theta(K)$ 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.

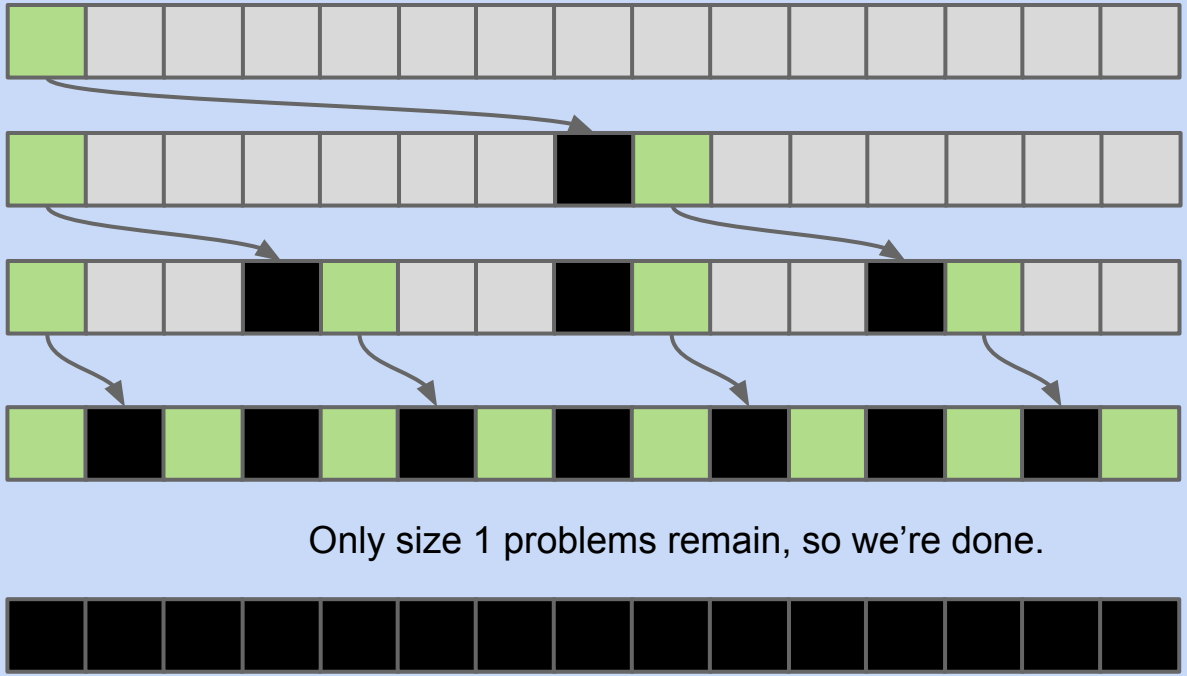
Quicksort Runtime

Best Case: Pivot Always Lands in the Middle



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

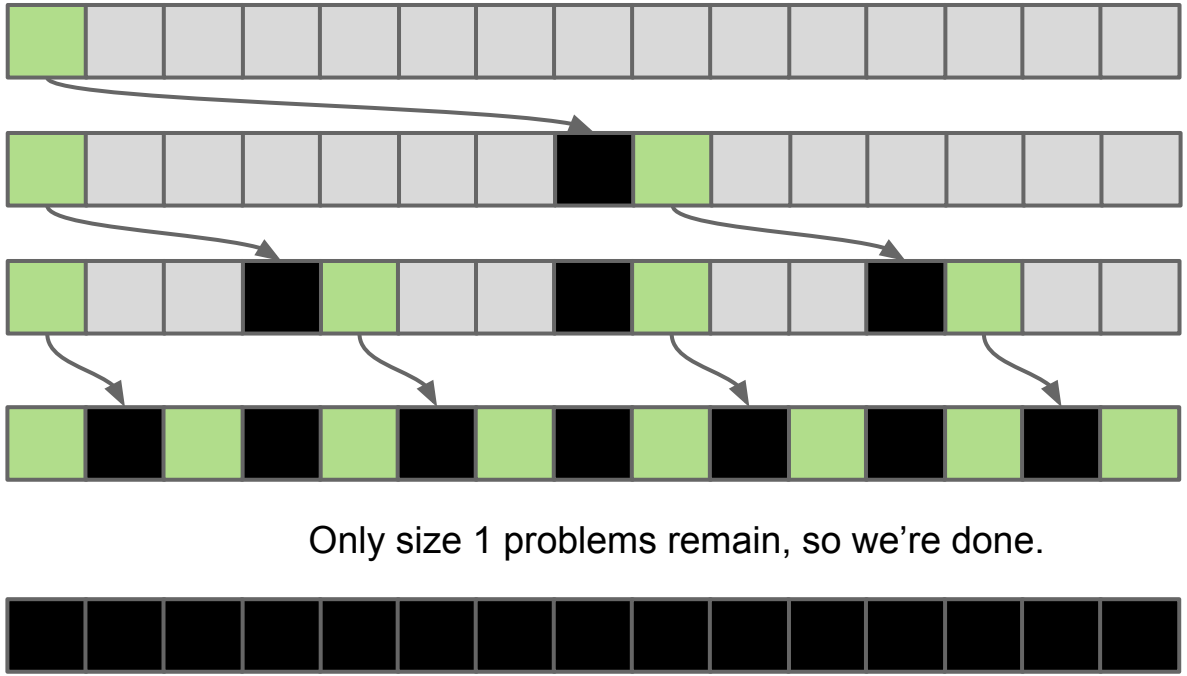
Best Case Runtime?



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:

$\approx N$

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

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

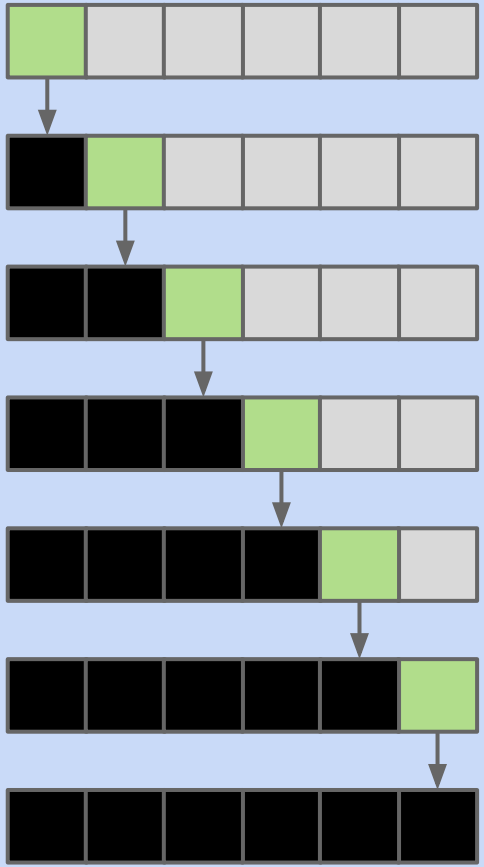
Overall runtime:
$\Theta(NH)$ where $H = \Theta(\log N)$

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 $\Theta(\cdot)$?



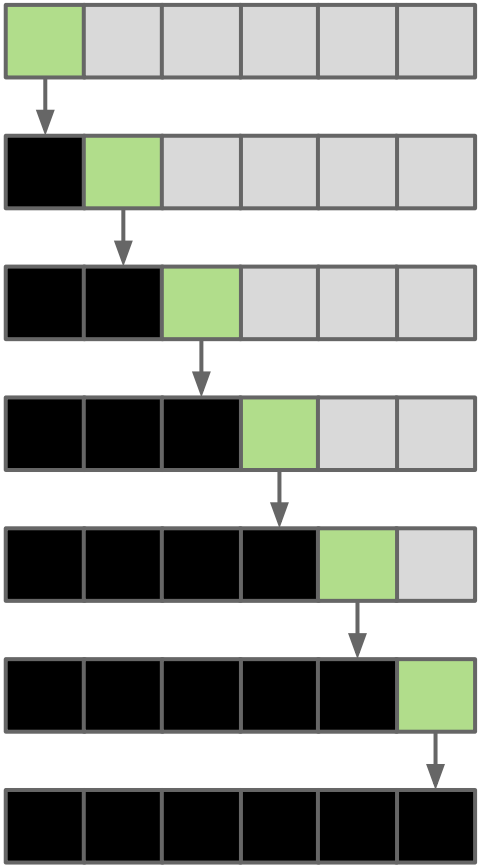
Worst Case: Pivot Always Lands at Beginning of Array

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

- 1 2 3 4 5 6

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

- N^2



Quicksort Performance

Theoretical analysis:

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

Compare this to Mergesort.

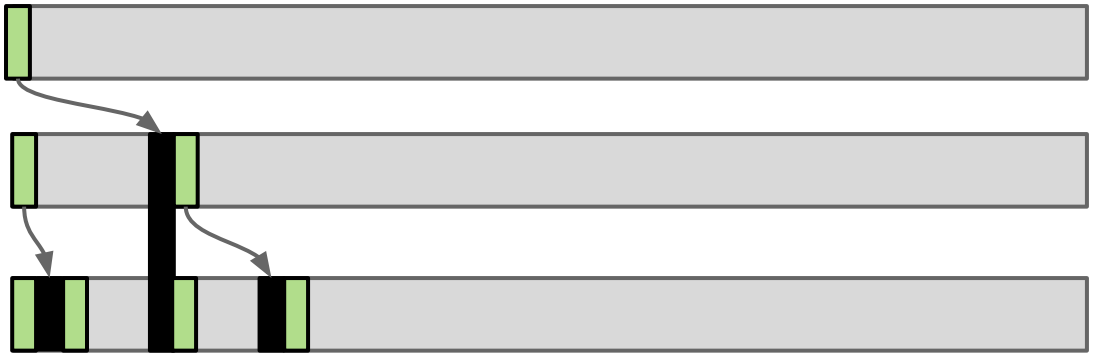
- Best case: $\Theta(N \log N)$
- Worst case: $\Theta(N \log N)$

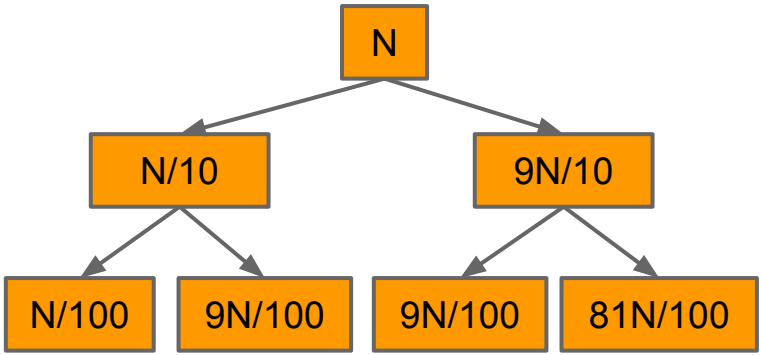
Recall that $\Theta(N \log N)$ vs. $\Theta(N^2)$ is a **really big deal**. So how can Quicksort be the fastest sort empirically? Because on average it is $\Theta(N \log N)$.

- 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).



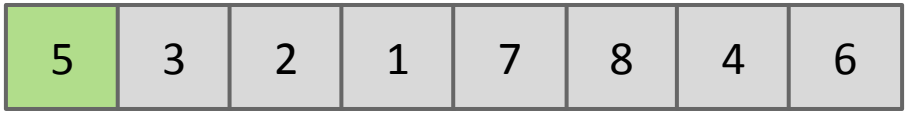


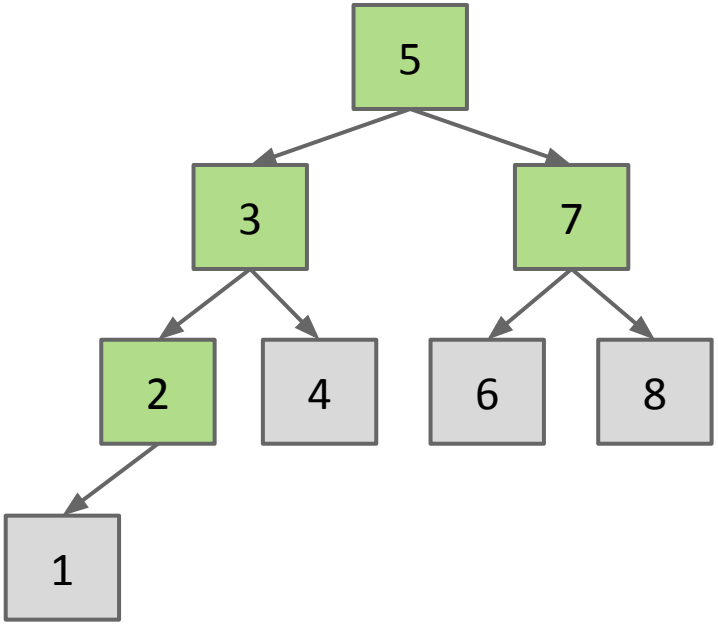
Work at each level: O(N)

- 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.

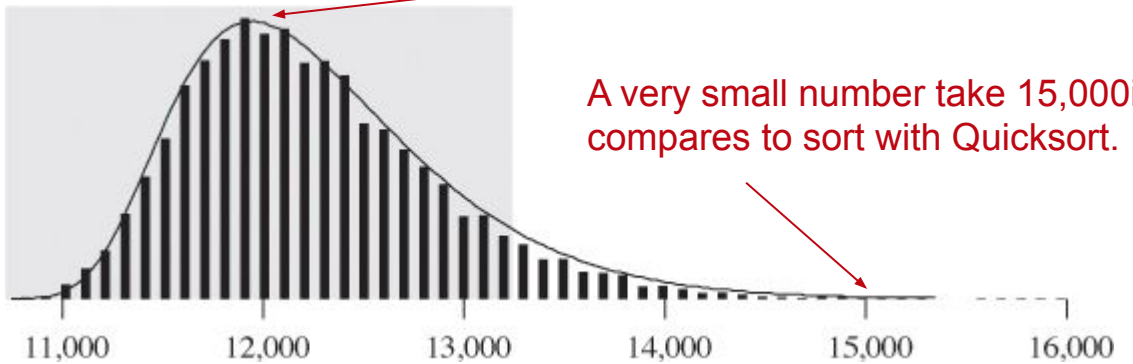
- Every number gets compared to 5 in both.
- 3 gets compared to only 1, 2, 4, and 5 in both.

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



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:

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

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

With extremely high probability!!

Compare this to Mergesort.

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

Avoiding the Quicksort Worst Case

Quicksort Performance

The performance of Quicksort (both order of growth and constant factors) depend critically on:

- How you select your pivot.
- How you partition around that pivot.
- Other optimizations you might add to speed things up.

Bad choices can be very bad indeed, resulting in $\Theta(N^2)$ runtimes.

Avoiding the Worst Case

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

- Bad ordering: Array already in sorted order (or almost sorted order).
- Bad elements: Array with all duplicates.

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.

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.

What can we do to avoid worst case behavior?

- Always use the median as the pivot -- this works.
- Randomly swap two indices occasionally.
  - Sporadic randomness. Maybe works?
- Shuffle before quicksorting.
  - This definitely works and is a harder core version of the above.
- Partition from the center of the array: Does not work, can still find bad cases.

Citations

Quickman from Mega Man 2

Deleted Slides

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.

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.

Citations

Quickman from Mega Man 2