Quicksort

- A recursive sorting algorithm
  - Invented by C.A.R. Hoare, 1960

- On average, the fastest algorithm
  - But occasionally explodes to $\mathcal{O}(n^2)$

- Splits the data in half, using a “pivot value”
- Then recurse into each half
Quicksort (by analogy)

- Consider performing a sort using a BST
  - Insert each element into the tree (no rebalancing)
  - At the end, do an in-order traversal to get the results

- Observations:
  - First element will be the root
  - If the root is (roughly) in the middle, the two sub-trees will be (roughly) the same size
  - Height of the tree often roughly $O(\lg n)$
Example 1

Build a BST.

Insert the following values in this order:
11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15
Typical Case:

Height of the tree is $O(\lg n)$.
Example 2

Insert the following values in this order:
100, -50, 90, -10, 30, 0
Worst Case:
Height of the tree is $O(n)$.
Quicksort

- Quicksort is analogous to the BST sort
  - Select a certain key value
  - Split the data items using the key
  - Recurse into both sides
- Same basic properties
  - Usually $\mathcal{O}(\lg n)$ recursion depth, occasionally $\mathcal{O}(n)$
- Quicksort is used to sort an array (in-place)
  - No tree needed!
Quicksort

• Quicksort has 4 basic steps
  - Choose a **pivot** (one of the values from the array)
  - **Partition** the array
    - Things less than the pivot on the left
    - Things more than the pivot on the right
  - Recurse left
  - Recurse right
What is a Pivot?

- Value from the array
- Chosen arbitrarily
  - Any value works (correctness)
  - Some are poor choices (efficiency)
- Goal: split the array in half (or close)
  - Not possible to get this perfect!
Choosing the Pivot

Strategies for choosing the pivot:

- **First or last element**
  - Simple, but often terrible – why?
- **Middle element**
  - Can you contrive a scenario where this would be terrible?
- **Random element**
- **Median-of-3**
Partitioning

• Goal: Partition the array
  – Small items to the left
  – Large items to the right
  – Pivot between them
  – Do it all in linear time!
  – Do it without any extra buffer array!
Partitioning

• Concept:
  - Put pivot at one end (temporarily)
  - Slowly grow “small” array from bottom, “big” array from top
  - When two arrays meet, swap pivot into place
Partitioning

- Equivalent design:
  - Save pivot value (but don't treat the element as special)
  - Slowly grow “small” array from bottom, “big” array from top
  - No explicit element for the pivot, just 2 arrays
Partition(A[p..r])
  x ← A[p]         # pivot *value* at head
  i ← p-1          # indx, lower end of TODO
  j ← r+1          # indx, upper end of TODO
  # i,j are both *exclusive*
  # bounds

  while True
    do j--           # decrement j until it
      while A[j] > x  # points to a small val

    do i++           # increment i until it
      while A[i] < x  # points to a large val

  if (i<j)
    swap(A[i],A[j]) # swap two; continue
  else
    return j        # return split point
Simulation

pivot value: \( x = 11 \)

Each step:
- \( i \) moves right until \( A[i] \geq x \)
- \( j \) moves right until \( A[j] \leq x \)
- Then swap
Simulation

\[ 11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15 \]

pivot value: \( x = 11 \)
Simulation

pivot value: \( x = 11 \)
Simulation

11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15

pivot value: \( x = 11 \)

4, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 11, 15

i \quad j
Simulation

11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15

pivot value: \( x = 11 \)

4, 7, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 13, 11, 15
Simulation

11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15

pivot value: \( x = 11 \)

4, 7, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 13, 11, 15

i

j
Simulation

11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15

pivot value: \( x = 11 \)

4, 7, 6, 10, 5, 9, 8, 19, 17, 16, 18, 12, 14, 20, 13, 11, 15

\[ i \quad \leftrightarrow \quad j \]
Simulation

11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15

pivot value: \( x = 11 \)

\[
4, 7, 6, 10, 5, 9, [8, 19, 17, 16, 18, 12, 14, 20, 13, 11, 15]
\]

\( j \) and \( i \) have crossed over, so the loop ends.
Simulation

11, 13, 6, 10, 5, 9, 16, 19, 17, 8, 18, 12, 14, 20, 7, 4, 15

pivot value: \( x = 11 \)

\[ \begin{align*}
4, & \quad 7, \quad 6, 10, \quad 5, \quad 9, \quad 8, \\
\leq & \quad x \\
19, & \quad 17, 16, 18, 12, 14, 20, 13, 11, 15, \\
\geq & \quad x
\end{align*} \]
Cost of Partitioning

- Compare every value to $x$
  - Sometimes as part of the “small” set
  - Sometimes as part of the “large” set
- Either:
  - Move on, to the next value
  - Swap, then move on
- Thus:
  - Constant-time max work per value
  - Linear time to partition the entire array
Quicksort In Overview

- We'll formally analyze the time-cost later
- Let's look at how the recursive algorithm works
Quicksort
(if it's perfect)

127 values

Running Quicksort on the entire array...

Stack
quicksort(0,127)
Quicksort
(if it's perfect)

127 values

Running Quicksort on the entire array...

Choose a pivot...

Stack
quicksort(0,127)
Quicksort
(if it's perfect)

63 values
(those <= the pivot)  63 values
(those >= the pivot)

Running Quicksort
on the entire array...

Choose a pivot...and partition
the array.

NOTE:
The pivot is in its final position
(kind of like selection sort)
Quicksort (if it's perfect)

Recurse!
Run Quicksort on the left half

Stack
quicksort(0,127)
quicksort(0,63)
Quicksort
(if it's perfect)

Choose a new pivot...

Recurse!
Run Quicksort on the left half

Stack
quicksort(0,127)
quicksort(0,63)
Quicksort  
(if it's perfect)

Choose a new pivot...and partition the array again.

Recurse!  
Run Quicksort on the left half

Stack 
quicksort(0,127) 
quicksort(0,63)
Quicksort
(if it's perfect)

31 values  31 values  63 values

Run Quicksort on the left quarter

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
QuickSort
(if it's perfect)

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(0,15)
Quicksort
(if it's perfect)

When the blocks get small enough, we stop recursing.

In the base case, we use an $n^2$ sort.

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(0,15)
quicksort(0,7)
Quicksort
(if it's perfect)

We return to the sort-15 call...

Stack
quicksort(0, 127)
quicksort(0, 63)
quicksort(0, 31)
quicksort(0, 15)
We return to the sort-15 call... and recurse into its right-hand partition.

<table>
<thead>
<tr>
<th>Stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>quicksort(0,127)</td>
<td>quicksort(0,63)</td>
</tr>
<tr>
<td>quicksort(0,31)</td>
<td>quicksort(0,15)</td>
</tr>
<tr>
<td>quicksort(8,7)</td>
<td></td>
</tr>
</tbody>
</table>
Quicksort
(if it's perfect)

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(0,15)
quicksort(8,7)

\[ n^2 \text{ sort again.} \]
Quicksort  
(if it's perfect)

We return to the sort-15 call, and now **all** of its values are sorted...

Stack
- quicksort(0,127)
- quicksort(0,63)
- quicksort(0,31)
- quicksort(0,15)
We return to the sort-15 call, and now all of its values are sorted... so it returns to the sort-31 call...

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
We return to the sort-15 call, and now all of its values are sorted... so it returns to the sort-31 call... which recurses into its own right-hand half.

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(16,15)
Quicksort
(if it's perfect)