Thank goodness! It’s Quicksort Man! Help me!

I’m on my way, Bubble Sort Man.
Quick-Sort

• To understand quick-sort, let’s look at a high-level description of the algorithm

• 1) **Divide**: If the sequence $S$ has 2 or more elements, select an element $x$ from $S$ to be your pivot. Any arbitrary element, like the last, will do. Remove all the elements of $S$ and divide them into 3 sequences:
  - $L$, holds $S$’s elements less than $x$
  - $E$, holds $S$’s elements equal to $x$
  - $G$, holds $S$’s elements greater than $x$

• 2) **Recurse**: Recursively sort $L$ and $G$

• 3) **Conquer**: Finally, to put elements back into $S$ in order, first inserts the elements of $L$, then those of $E$, and those of $G$.

• Here are some pretty diagrams....
Idea of Quick Sort

1. Select
   pick *an* element

2. Devide
   rearrange elements so that
   • *x* goes to its final position *E*

3. Recurse and Conquer
   recursively sort
Quick-Sort Tree

7 6 2 10 4 5 9 8

7 6 2 4 5 8 10 9

7.4
Quick-Sort Tree
Quick-Sort Tree
Quick-Sort Tree

2 4

5 7 6

8 10 9

2 4

5 7 6

8 10 9

2 4

5 7 6

8 10 9
Quick-Sort Tree

2 4 5 6 7 8 10 9

Diagram of a Quicksort algorithm as a tree structure.
Quick-Sort Tree

Skipping ...
... Finally

2 4 5 6 7

2 4 5 6 7 8 9 10

2 4 5 6 7 8 9 10
In-Place Quick-Sort

• **Divide step**: $l$ scans the sequence from the left, and $r$ from the right.

```
85  24  63  45  17  31  96  50
  l                        r
```

• A swap is performed when $l$ is at an element larger than the pivot and $r$ is at one smaller than the pivot.

```
31  24  63  45  17  85  96  50
  l                        r
```
• A final swap with the pivot completes the divide step
In Place Quick Sort code

public class ArrayQuickSort implements SortObject {

    public void sort(Sequence S, Comparator c) {
        quicksort(S, C, 0, S.size() - 1);
    }

    private void quicksort(Sequence S, Comparator c, int leftBound, int rightBound) {
        // left and rightmost ranks of // sorting range
        if (S.size() < 2) return; // a sequence with 0 or // 1 elements is already sorted
        if (leftBound >= rightBound) return; // terminate // recursion

        // pick the pivot as the current last // element in range
        Object pivot = S.atRank(rightBound).element();

        // indices used to scan the sorting range
        int leftIndex = leftBound; // will scan // rightward
        int rightIndex = rightBound - 1; // will scan // leftward
In Place Quick Sort code
(contd.)

// outer loop
while (leftIndex <= rightIndex) {

  // scan rightward until an element larger than
  // the pivot is found or the indices cross
  while ((leftIndex <= rightIndex) &&
    (c.isLessThanOrEqualTo
    (S.atRank(leftIndex).element(),pivot))
    leftIndex++;

  // scan leftward until an element smaller than
  // the pivot is found or the indices cross
  while (rightIndex >= leftIndex) &&
    (c.isGreaterThanOrEqualTo
    (S.atRank(rightIndex).element(),pivot))
    rightIndex--;

  // if an element larger than the pivot and an
  // element smaller than the pivot have been
  // found, swap them
  if (leftIndex < rightIndex)
    S.swap(S.atRank(leftIndex),S.atRank(rightIndex));

} // the outer loop continues until
// the indices cross. End of outer loop.
In Place Quick Sort code (contd.)

// put the pivot in its place by swapping it with the element at leftIndex
S.swap(S.atRank(leftIndex),S.atRank(rightBound));

// the pivot is now at leftIndex, so recur on both sides
quicksort(S, c, leftBound, leftIndex-1);
quickSort(S, c, leftIndex+1, rightBound);

} // end quicksort method

} // end ArrayQuickSort class
Analysis of Running Time

- Consider a quick-sort tree $T$:
  - Let $s_i(n)$ denote the sum of the input sizes of the nodes at depth $i$ in $T$.
- We know that $s_0(n) = n$ since the root of $T$ is associated with the entire input set.
- Also, $s_1(n) = n - 1$ since the pivot is not propagated.
- Thus: either $s_2(n) = n - 3$, or $n - 2$ (if one of the nodes has a zero input size).

The worst case running time of a quick-sort is then:

$$O\left(\sum_{i=0}^{n-1} s_i(n)\right)$$

Which reduces to:

$$O\left(\sum_{i=0}^{n-1} (n - i)\right) = O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

Thus quick-sort runs in time $O(n^2)$ in the worst case.
Analysis of Running Time (contd.)

- Now to look at the best case running time:

- We can see that quicksort behaves optimally if, whenever a sequence S is divided into subsequences L and G, they are of equal size.

- More precisely:
  - \( s_0(n) = n \)
  - \( s_1(n) = n - 1 \)
  - \( s_2(n) = n - (1 + 2) = n - 3 \)
  - \( s_3(n) = n - (1 + 2 + 2^2) = n - 7 \)
  
  ...  

  - \( s_i(n) = n - (1 + 2 + 2^2 + ... + 2^{i-1}) = n - 2^i + 1 \)
  
  ...

- This implies that \( T \) has height \( O(\log n) \)

- Best Case Time Complexity: \( O(n\log n) \)
Randomized Quick-Sort

• Select the pivot as a *random* element of the sequence

• The expected running time of randomized quick-sort on a sequence of size $n$ is $O(n \log n)$

• The time spent at a level of the quick-sort tree is $O(n)$

• We show that the *expected height* of the quick-sort tree is $O(\log n)$

• good vs. bad pivots

  - **good**: $1/4 \leq n_L/n \leq 3/4$
  - **bad**: $n_L/n < 1/4$ or $n_L/n > 3/4$

• the probability of a good pivot is 1/2, thus we expect $k/2$ good pivots out of $k$ pivots

• after a good pivot the size of each child sequence is at most 3/4 the size of the parent sequence

• After $h$ pivots, we expect $(3/4)^{h/2} \ n$ elements

• the expected height $h$ of the quick-sort tree is at most:

  $$2 \log_{4/3} n$$