CMPT 225: Data Structures & Programming – Unit 27 – Quick-Sort

Dr. Jack Thomas
Simon Fraser University
Spring 2021

Today's Topics

- Divide-and-Conquer (again)
- Quick-Sort
- Analyzing Quick-Sort
- Quick-Sort in Java
- Implementing Quick-Sort

Dividing up Divide-and-Conquer

- Merge-Sort isn't the only comparison-based sorting algorithm based on divide-and-conquer, there's also Quick-Sort.
- Whereas Merge-Sort divided the data first, then did the work of sorting as it merged them back together again, Quick-Sort will sort the data through dividing it.

Quick-Sort as Divide-and-Conquer

- **Divide**: If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the pivot. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
  - L, storing the elements in S less than x.
  - E, storing the elements in S equal to x.
  - G, storing the elements in S greater than x.
- **Recur**: Recursively sort sequences L and G.
- Conquer: Put back the elements into S in order by first **inserting** the elements of L, then those of E, and finally those of G.

Quick-Sort in a Nutshell

- Quick-Sort is based around picking one value from the set of data (called the pivot), then splitting the rest of the set into three subsets based on whether they're larger, smaller, or equal to the pivot.
- You repeat the process on the subsequences until you've broken everything down to sets of zero, one, or multiple equal values, laid out in order so that re-combining them will give you a sorted sequence.

Cool Let's Get Another One Of Those Fun Wikipedia Visualizations

- Eh, this one's not as good.

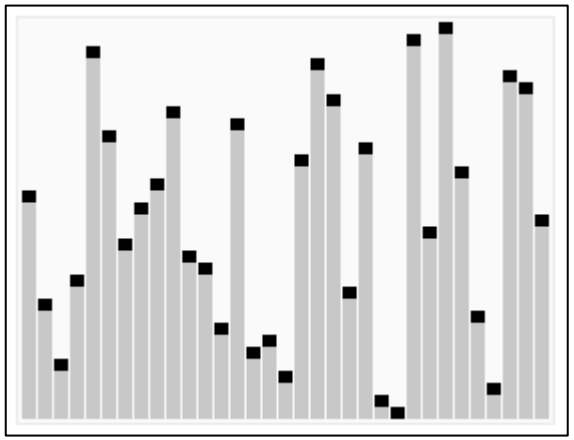
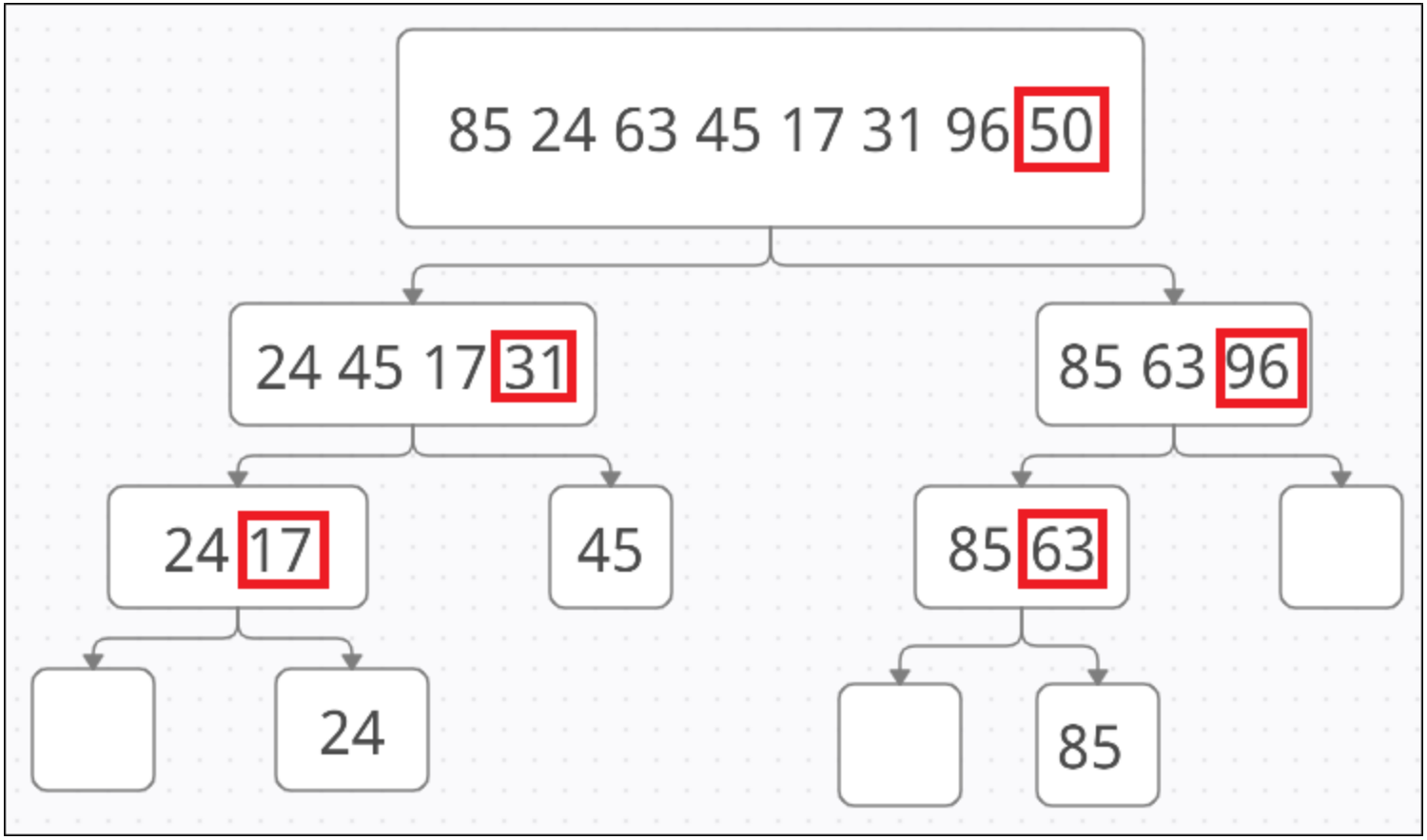
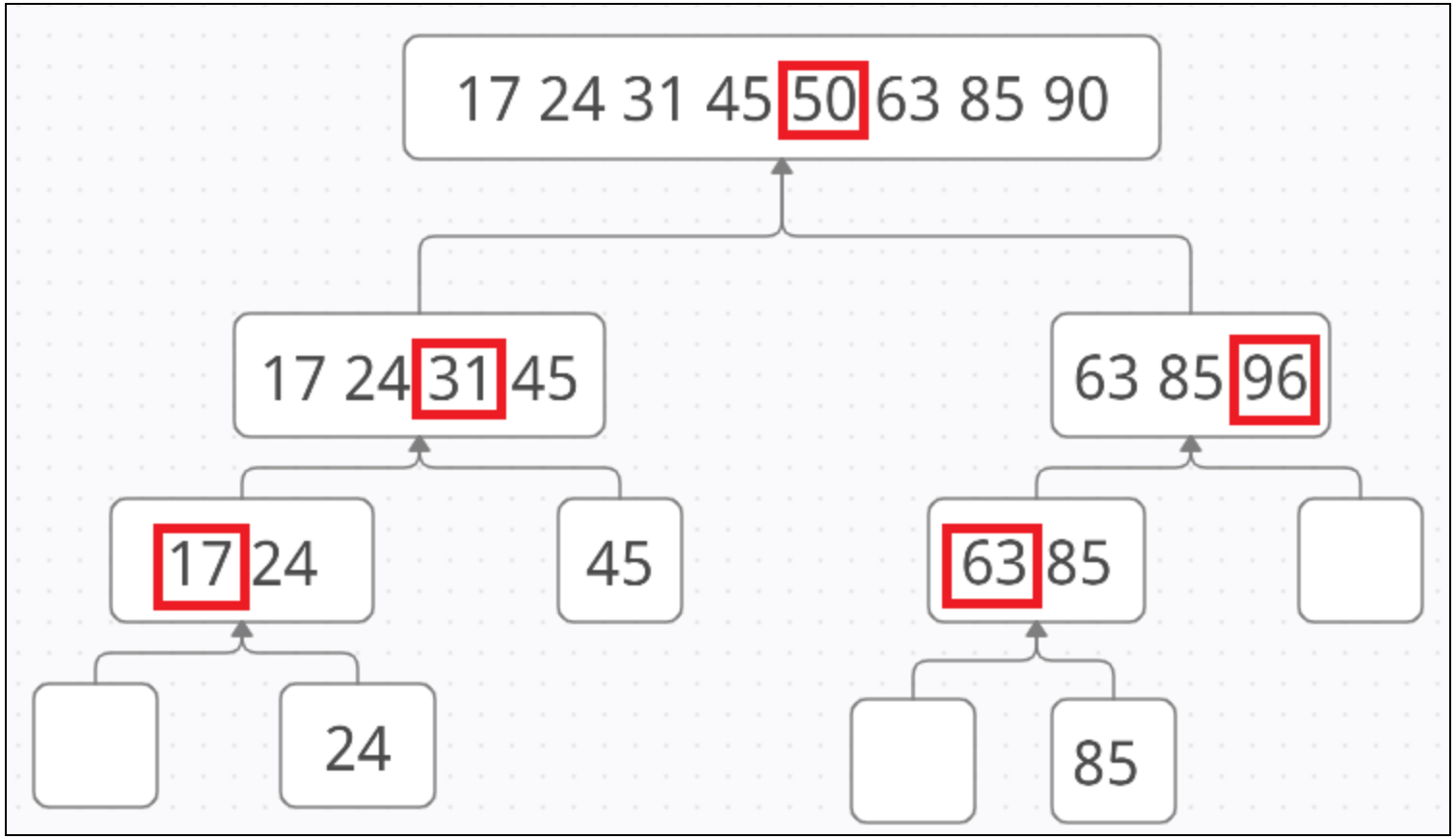


Image credit:
https://upload.wikimedia.org/wikipedia/commons/6/6a/Sorting_quicksort_anim.gif

Divide…



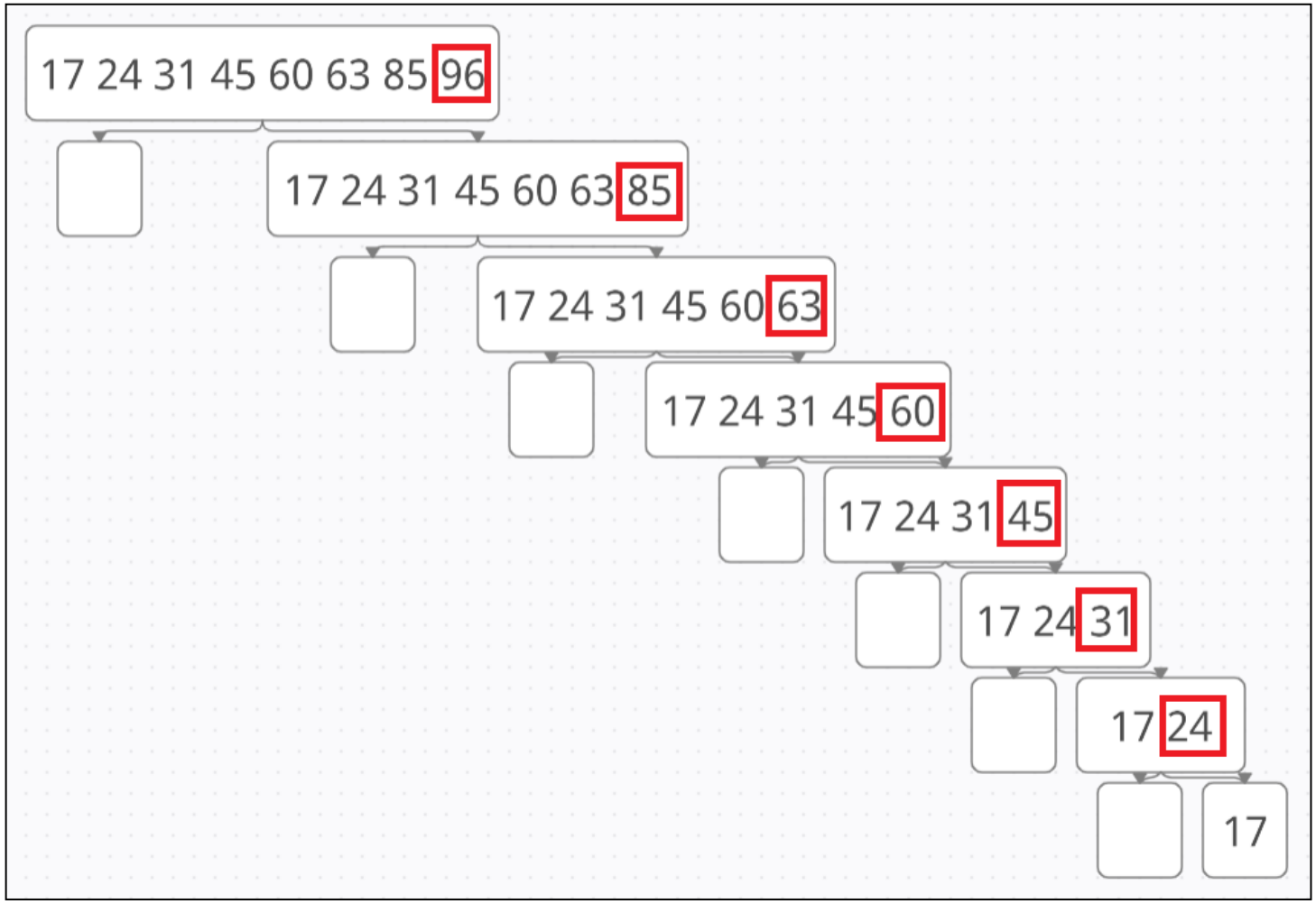
…and Conquer



Analysis

- Each node of our Tree represents a recursive call of Quick-Sort that needs to be resolved.
- It also produces a number of comparisons equal to the number of entries.
- The more equally-divided each sequence is into sub-sequences, the fewer calls we'll need to completely divide up the data, and the fewer comparisons it'll take to divide each sub-sequence.
- Taken together, this means the efficiency of our sorting depends on the compactness of the Tree.

Consider a Worst-Case



Quick-Sort's Random Element

- Merge-Sort controlled the size of the tree by guaranteeing each node had half the number of elements as its parent, but Quick-Sort's use of pivots introduces some randomness to the size of the subsequences.
- We want to choose a pivot value that will divide the sequence neatly in half, but we don't actually want to search through the sequence to find that value – searching the whole sequence would essentially require sorting it, after all.

Partitions and Pivots

- The next best thing to a perfect pivot is a randomly-chosen pivot, instead of the last element.
- Regardless of the actual values of the elements, we can assert that half of them would divide the sequence into sub-sequences of between n/4 and 3n/4 – that's just a product of there being an ordering at all.
- This is called Randomized Quick-Sort, and we're allowed to expect it to be an O(n log n) algorithm, even though technically the true worst-case is $O(n^2)$.

Quick-Sort in Java

- Unsurprisingly at this point, Java doesn't have a built-in Quick-Sort method.
- However, for reasons we'll discuss later, Quick-Sort is a fairly popular sorting method, which means it's often the implementation for .sort() methods in standard library classes.
- This is still subject to change, however – for example, the .sort() for arrays of primitive data types is now the Dual-Pivot-Quick-Sort, a variant of Quick-Sort.

How About Implementation? Are Arrays and Lists a Problem Again?

- Not really, no.
- This time around, we can express the algorithm for Quick-Sort generally, without being specific to a list or array.
- Note that in the following algorithm, I do use list-based language (addLast(), for example), but you can implement that pseudocode statement with an array as well. There's just no array-specific adaptations as there was with Merge-Sort's two counters.

```
Algorithm QuickSort(S):
Input: A sequence S implemented as an array or linked list.
Output: The sequence S in sorted order.
If S. size() < = 1 then
        return
P <- S.last().element()
Let L, E, and G be empty list-based sequences
While !S.isEmpty() do
        if S.last().element() < p then
                L.addLast(S.remove(S.getLast())
        Else if S.last().element() = p then
                E.addLast(S.remove(S.getLast()))
        Else
                G.addLast(S.remove(S.getLast()))
QuickSort(L)
QuickSort(G)
While !L.isEmpty() do
        S.addLast(L.remove(L.getFirst()))
While !E.isEmpty() do
        S.addLast(E.remove(E.getFirst()))
While!G.isEmpty() do
        S.addLast(G.remove(G.getFirst()))
return
```

Recap: Let's Sort This Quick

- Quick-Sort is another comparison-based Divide-and-Conquer sorting algorithm.
- Where Merge-Sort did the sorting work while merging data back together, Quick-Sort does it while dividing it up.
- The efficiency of Quick-Sort depends on the choice of pivot values, and we can generally expect O(n log n) with a randomly-chosen pivot.
- Java doesn't offer an explicit Quick-Sort, but often uses it to implement .sort() methods.
- The implementation for Quick-Sort is general to either arrays or lists.