Quicksort

First let's review two things from last class.

Figure 12.1
A circular array representing the positional collection \( \langle w, x, y, \text{null}, z \rangle \) where \( \text{start}=5 \).

* implementation in text guarantees that array slots not in use are null.

I called this offset last class.
Tracked Array holding \(<w, x, y, \emptyset, z>\)

underlying index (not position) stored. Then only need to update when element is physically moved...

redirect used to navigate to next element in iteration order from a tracked element even when it has been deleted.
Sorting Algorithms

Seen insertion sort - good for nearly sorted data but worst-case $\Theta(n^2)$ time

Seen merge sort - worst-case $\Theta(n \log n)$ time ($T(n) = 2T(n/2) + \Theta(n)$)

Today we'll study quicksort
**Quicksort**

Divide-and-Conquer Alg

Do all the hard work in splitting (at recursive call). No combine.

**Divide**: Partition array into two subarrays where all elements in left portion are less than all elements in right portion.
Example of Partition

\[ \ldots 11, 4, 9, 7, 3, 10, 2, 6, 8 \ldots \]

\[ \uparrow \]

\[ \ldots 6, 4, 9, 7, 3, 10, 2, 11, 8 \]

\[ \uparrow \]

\[ \ldots 6, 4, 2, 7, 3, \boxed{10, 9, 11, 8} \]

\[ \uparrow \]

\[ \ldots 6, 4, 2, 7, 3, 8, 9, 11, 10 \]
General invariant maintained

... left

< pivot

not yet processed

>= pivot

... right

pivot
Java Code For Partition

```java
int partition(int left, int right, Comparator<? super E> sorter) {
    E pivot = read(right); // pivot around the right element
    int i = left;
    int j = right;
    while (i < j) {
        while (i < j && sorter.compare(read(i), pivot) < 0) // pos. i element < pivot
            i++;
        while (j > i && sorter.compare(read(j), pivot) >= 0) // pos. j element ≥ pivot
            j--;
        if (i < j) // swaps pos i and pos j elements
            swapImpl(i, j);
    }
    swapImpl(i, right);
    return i;
}
```
Quicksort (Comparator sorter) {
    quicksortImpl (0, n-1, sorter)
}

quicksortImpl (left, right, sorter) {
    if (left < right) {
        swap what's in pos right using median-of-three
        mid = partition (left, right, sorter);
        quicksortImpl (left, mid-1, sorter);
        quicksortImpl (mid+1, right, sorter);
    }
}

method of positional collection interface
Time Complexity

Asymptotic time complexity for Partition is $\Theta(n)$

Best Case: Pivot is in middle

$T(n) = 2T\left(\frac{n-1}{2}\right) + \Theta(n) = \Theta(n \log n)$
What's the worst-case?

What if pivot is min or max?

What if positional collection was already sorted?

\[\text{pivot} \]

\[\text{mid} \]

n-1 elements in left

no elements in right

mid+1, right
Leads to recurrence: \[ T(n) = T(n-1) + \Theta(n) \]

\[ \sum_{n=1}^{N} C \cdot n \]

\[ C \cdot \frac{n(n+1)}{2} = \Theta(n^2) \]
How can we pick the partition to try to avoid bad behavior?

Median-of-three partitioning

\[ \frac{\text{left} + \text{right}}{2} \]

Find median of \( a, b + c \) and then if this is not \( c \), swap \( c \) with median
The other common solution is **Randomized Quicksort**

Pick a random element (uniformly) from subarray and swap that with the element in rightmost position of the subarray.

Show highlights that expected (average) time is \( \Theta(n \log n) \)
Expected Time Complexity

\[ E[T_A(x)] = \sum_t t \cdot \text{Prob}[T_A(x) = t] \]

\[ \frac{1}{2} \cdot 1 + \frac{1}{10} \cdot 2 + \ldots + \frac{1}{10} \cdot 6 = 2.5 \]
Overview of Analysis

- Dominant cost is comparison performed by partition
- Only aspect that affects time complexity is # elements in subarrays for recursive calls (at each level of recursion)

\[
\begin{align*}
i & \quad j \\
\text{j - i + 1 elements} & \\
\end{align*}
\]