Announcements

- PS 4 is ready
- Midterm on 10/15?
- Fast sorting algorithms (Quick sort)
- Break around 3:25pm

Mergesort review

- Divide and Conquer Algorithm
  1. Divide input in half
  2. Sort each half recursively
  3. Merge/join sorted halves together as recursion unwinds

- For Mergesort, dividing in half is easy!

- Merging is where the work occurs.

Quicksort

- A general purpose sorting algorithm developed by C.A.R. (Tony) Hoare
- Has the reputation as the fastest comparison-based in-place sort in the average case.
- Bad worst-case performance (as we shall see) but simple steps can be taken to make the worst-case unlikely.
- java.util.Arrays.sort(…) uses this for each of the primitive element type arrays

Quicksort algorithm overview

- Like Mergesort, Quicksort is a recursive divide and conquer algorithm but unlike Mergesort, the dividing step is where the work occurs while the merge or join step is trivial.

- In general:
  1. Partition array into “lower” and “higher” subarrays
  2. Sort each subarray recursively
  3. Join results (trivial)

Quicksort recursively

- Check for base case (array of size 1).
- Choose a pivot element from the array to serve as the boundary between the lower half and the upper half of the array.
- Partition the array such that all elements to the left of the pivot are less than the pivot and all elements to the right of the pivot are greater than the pivot.
- Recursively sort the lower and upper subarrays that result from the partitioning step.

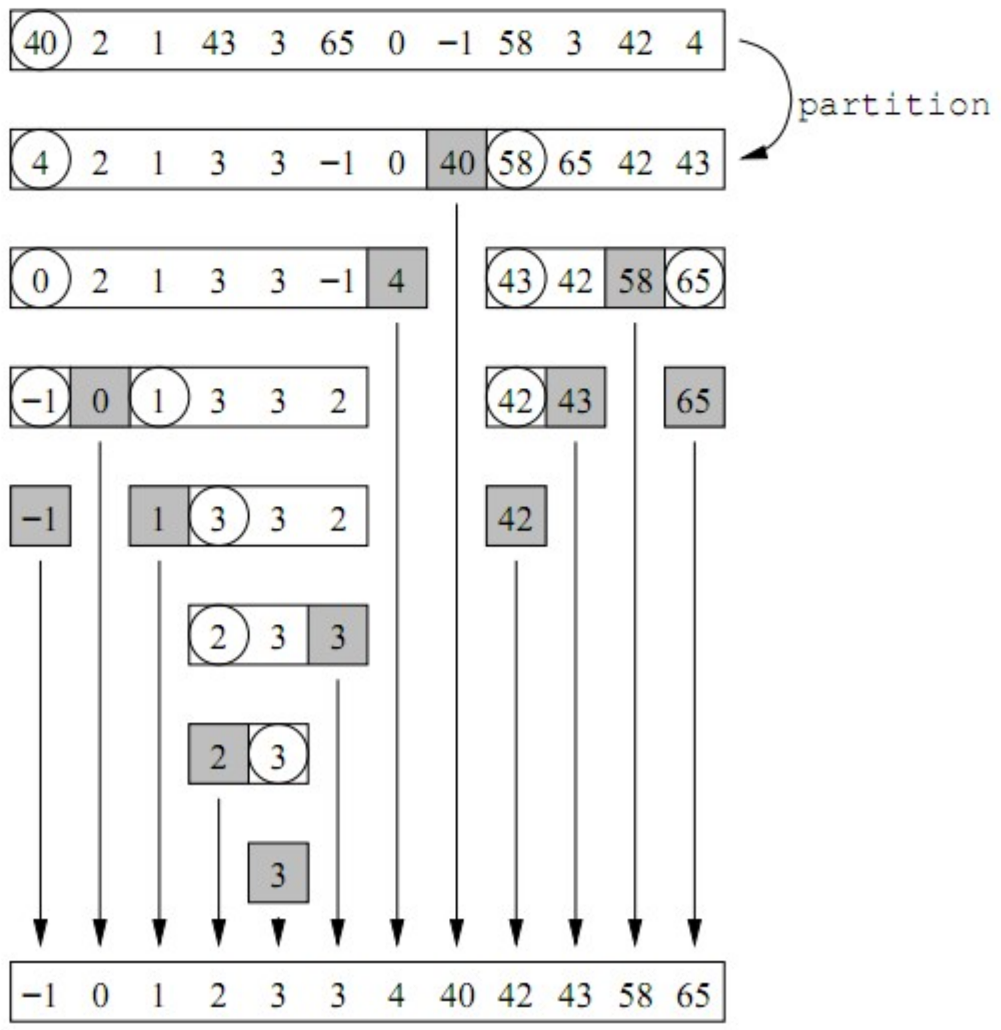
Partition

- The goal of partitioning is to move elements of the input array into the following form:

{ < pivot }	{ pivot }	{ > pivot }

- Choosing the pivot:
  - Ideal case is the median (‘the middle number’)
  - Simple scheme is to choose the first element since we know it’s at least in range.

Quicksort example



Quicksort pseudocode

```
quicksort(A, lo, hi)
  if lo < hi
    p = partition(A, lo, hi)
    quicksort(A, lo, p - 1)
    quicksort(A, p + 1, hi)

partition(A, lo, hi)
    pivot = A[hi]
    i = lo //place for swapping
    for j = lo to hi - 1
        if A[j] <= pivot
            swap A[i] with A[j]
            i = i + 1
    swap A[i] with A[hi]
    return i
```

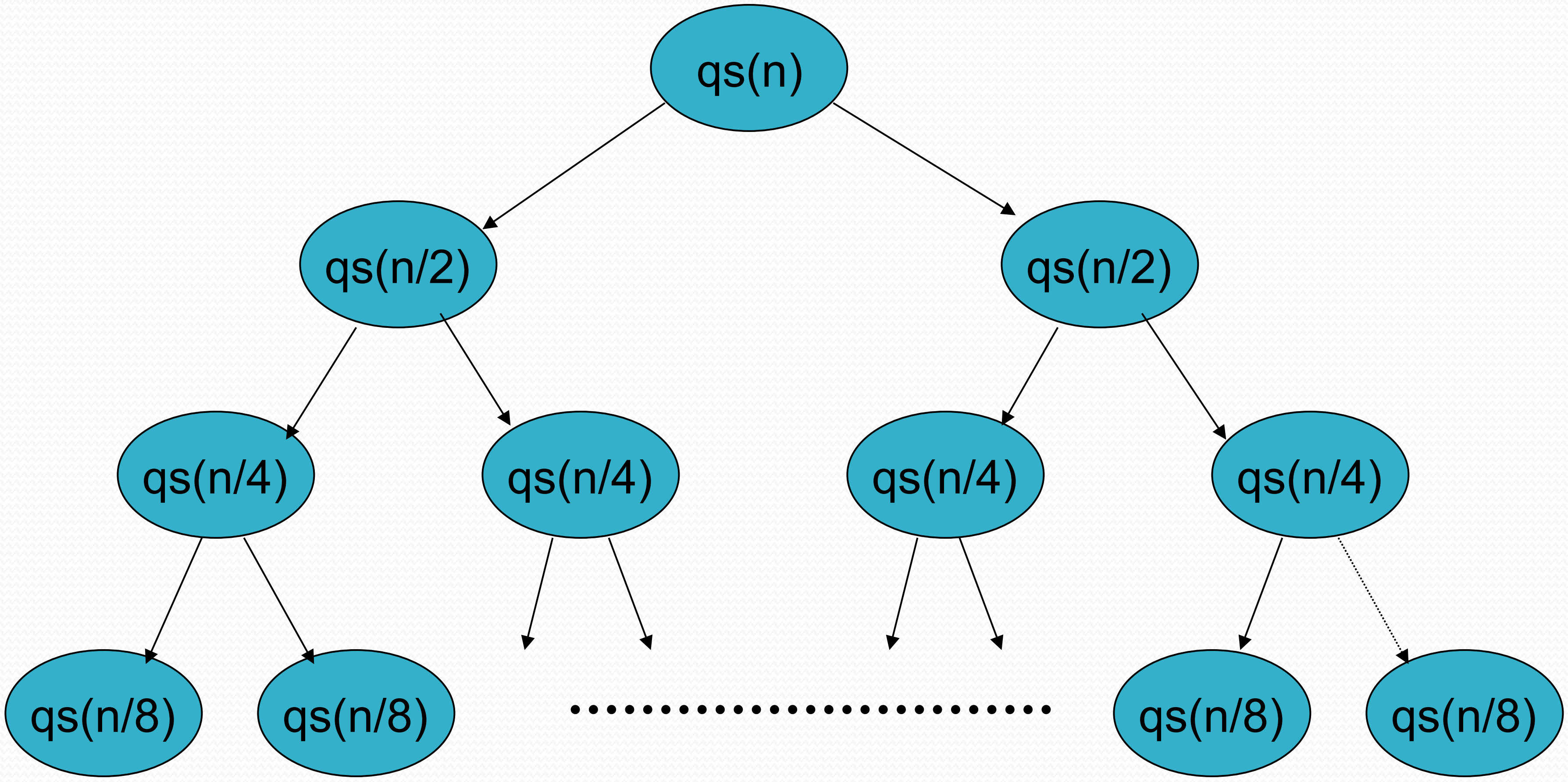
Quick sort code

- See Quick.java
- Also see the animation from last lecture
- Also as a folk dance if you like:
    - http://www.youtube.com/watch?v=ywWBy6J5gz8

Intuitive best case analysis

- To get an intuitive feel for the best case running time we'll assume the following for simplicity:
  - There are no duplicate elements.
  - We pick the pivot perfectly, i.e., we always pick the median of the input array as the pivot.
    - results in 2 subarrays of the "same" size (sound familiar?)
  - We'll ignore that the pivot is not in the subarrays, i.e., for our analysis, n-sized arrays are partitioned into subarrays of n/2 instead of (n-1)/2.

Quicksort recursive tree (best case)



Intuitive best case analysis (cont.)

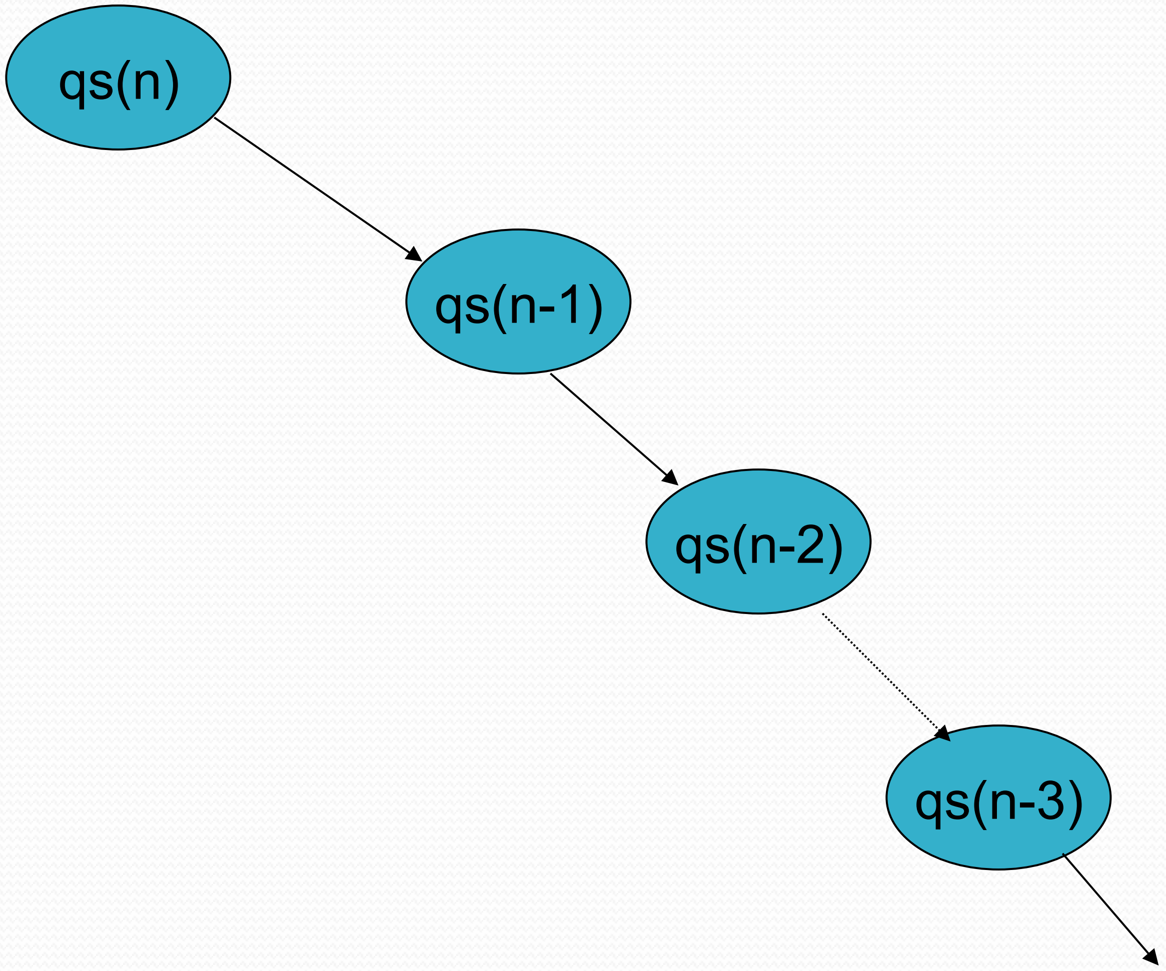
- In the recursion tree diagram for Quicksort's best case, each node represents recursive calls that are made and the node indicates the size of the array that the partition code will process.
    - This is similar to Mergesort! There are $\log_2 n$ levels and at each level, the number of elements partitioned is nearly identical to the number of elements that are merged in Mergesort.
- So, Quicksort is $\Theta(n \cdot \log_2 n)$ in the best case.

Intuitive worst case analysis

- What do you think causes the worst case running time? Think about the recursive tree representation again.

- Answer: when the pivot is the largest or smallest element in the array or subarray. One subarray has all the elements after partitioning!

Quicksort recursive tree (worst case)



Intuitive worst case analysis (cont.)

- Our recursive tree diagram is severely unbalanced! To determine the approximate running time, note the following:
  - At the 1st level, n elements will be partitioned.
  - At the 2nd level, n-1 elements will be partitioned.
  - At the 3rd level, n-2 elements will be partitioned.
  - Etc.

- The height of this recursion tree is n and assuming a simple model of computation, this results in:

$$\begin{aligned} T(n) &= n + n-1 + n-2 + \ldots + 3 + 2 + 1 \\ &= n(n+1)/2 \end{aligned}$$

- So, worst case, $T(n) = \Theta(n^2)$

Average case analysis

- To analyze the average-case behavior, we need to account for the cost of all possible arrangements of input and then divide by the number of cases.
  - To simplify things, we assume that all arrangements are equally likely, i.e., the pivot has equal probability of ending up at any spot.

- The running time is then:

$$T(n) = cn + (1/n) \sum_{k=0}^{n-1} [T(k) + T(n-1-k)], \quad T(0) = T(1) = c$$

where $T(k) + T(n-1-k)$ represents the cost for two recursive calls on arrays of size k and $n-1-k$ and cn is the cost of finding the pivot and partitioning.

Average case analysis (cont.)

- This hideous recurrence relation has a closed-form solution that's $\Theta(n \cdot \log_2 n)$.
- So, Quicksort's average-case behavior of $\Theta(n \cdot \log_2 n)$ is within a constant factor of the best case!
- What do we need to do to get average-case behavior?
  - Maybe if we can just avoid the worst-case, we can get "near" average-case!

Avoiding the worst case

- To get the best case behavior, we want the pivot to be the median of the array/subarrays at all recursive levels.
  - There are linear time algorithms for finding the median.
  - Could be overkill though. Because uneven (not extreme) splits still result in near average-case performance.

- So far, we've been using the first array element as the pivot. Given this scheme, what situation would cause the worst case behavior to occur?

- Answer: ironically, when the array is already sorted in either direction!
  - Picking the last element in the array as the pivot has the same problem.

Avoiding the worst case (cont.)

- One possible approach is to select the pivot at random.
  - There's a chance you'll get a degenerate split occasionally but you won't consistently get extreme splits.
  - Extra cost depends on the random number algorithm.

- Another common approach is the "Median of 3" approach where you select the median of 3 elements (typically the first, middle and last elements).
  - Handles the presorted or nearly sorted case.
  - Still degenerate cases but less likely to happen.
  - Obviously, can't use when there's less than 3 elements (use Insertion Sort for "small n" instead.

Quicksort optimizations

- In the real world, for “small” n, the arrays are often sorted or nearly sorted.
    - This can be bad for Quicksort so insertion sort is sometimes used as the finishing steps because of it’s Θ(n) behavior for nearly sorted sequences (similarly for Mergesort).

Quicksort vs. Mergesort

- For general purpose comparison-based sorting, $\Theta(n \cdot \log_2 n)$ is the best we can do theoretically.
- While Quicksort and Mergesort are both $\Theta(n \cdot \log_2 n)$ in the average case, Quicksort has “better” constants and is generally faster.
  - Quicksort is typically more efficient about moving elements around, hence the better performance.
- Quicksort’s worst case is bad though... mostly avoidable but still bad occasionally.
- Mergesort is steady... $\Theta(n \log n)$ no matter what the input is.
  - Uses extra memory though.

Quicksort best case analysis

- The best case occurs when we get an equal split (perfect pivots) at each recursive level. Then, we get the following recurrence relation:

$$T(n) = 2T(n/2) + n \quad \text{for } n > 1,\ T(1) = 1$$

Will solve it in class

- So Quicksort's Best Case Theta is $\Theta(n \cdot \log_2 n)$
  - This analysis is identical to Theta analysis for Mergesort

Quicksort worst case analysis

- The worst case occurs when one subarray has all the elements after the partition step. This results in the following recurrence relation for the running time:

  $T(n) = T(n-1) + n \quad \text{for } n > 1, T(1) = 1$

  [This is what you will get when you do `sSortRec` in the homework.]

- So Quicksort worst case is $\Theta(n^2)$

Next time

- We will start the Java Collections Framework