In lecture we discussed two sorting methods: selection sort and insertion sort.

For each iteration:

Selection sort, SELECTS the next minimum in the unsorted part and puts it at the end of the sorted part using a SWAP.

You always have to look at all the (unsorted) elements to find the min.

Insertion sort, INSERTS the next element in the unsorted part in the appropriate place in the sorted part by SHIFTING elements to make room.
The runtime depends on how many elements need to be shifted.

Selection Sort:

Worst-case:	O(n^2)
Average:	O(n^2)
Best-case:	O(n^2)

Insertion Sort:

Worst-case:	O(n^2)
Average:	O(n^2)
Best-case:	O(n) (when the data is already sort, or very nearly sorted)

Sub-Quadratic Sorts

Why was binary search so fast? Because we halved the problem each round.

The idea of dividing a problem into smaller pieces and solving each piece separately is a powerful technique, especially if the smaller pieces are about the same size. The technique is called DIVIDE-AND-CONQUER.

Merge Sort

1. Divide the problem into the first half and second half (*Divide*)
2. Sort each half recursively (*Conquer*)
3. Merge the sorted halves together into one sorted list (*Combine*)

Simplest case is when there is 0 or 1 elements.

```
public static void mergesort(int[] A) {
    // create a temporary array that merge will use
    int[] temp = new int[A.length];
    mergesort(A, temp, 0, A.length-1];
}
```

```
// Sort elements between low and high inclusive
private static void mergesort(int[] A, int[] temp,
                              int low, int high) {
   if (high <= low) {
     return;
   }
   else {
      int mid = (low + high)/2;
      mergesort(A, temp, low, mid);
      mergesort(A, temp, mid+1, high);
      merge(A, temp, low, mid, high)
   }
}
```

```
// Precondition:
//     The elements of A from low to mid inclusive) are sorted.
//     The elements of A from mid+1 to high (inclusive) are sorted.
// Postcondition:
//     The elements of A from low to high inclusive) are sorted.
private static void merge(int[] A, int[] temp,
                          int low, int mid, int high) {
    // Copy elements to temp.
    for (int i = low; i <= high; i++) {
        temp[i] = A[i];
    }

    int i1 = low;    // where in first half of temp
    int i2 = mid+1;  // where in second half of temp
    int i = low;     // where in A

    // Copy back to A the smaller of the next element in first
    // half and the next element in the second half.
    while (i1 <= mid && i2 <= high) {
       if (temp[i1] < temp[i2]) {
           A[i] = temp[i1];
           i1++;
       }
       else {
           A[i] = temp[i2];
           i2++;
       }
       i++;
    }
    // Copy back any remaining elements is first half.
    while (i1 <= mid) [
            A[i] = temp[i1];
            i1++; i++;
     }
    // any remaining elements in second half are already in A
}
```

What is the runtime of merge? O(n)

Analysis of mergesort:

At every level of the call tree we do O(n) work
There are log n levels
Total runtime is O(n log n)

Uses O(n) extra space.

Merge improvements:

- You can avoid half of the copying and use the second half of A as temp.
- Note that if the last element (largest) of the first half is smaller then the first element (smallest) of the second half, then the elements are already sorted.

```
private static void fasterMerge(int[] A, int[] temp,
                                int low, int mid, int high) {
    if (A[mid] < A[mid+1]) return;

    // copy first half to temp
    for (int i = low; i <= mid; i++) {
        temp[i] = A[i];
    }

    int i1 = low;
    int i2 = mid+1;
    int i = low;
    while (i1 <= mid && i2 <= high) [
       if (temp[i1] < A[i2]) {
           A[i] = temp[i1];
           i1++;
       }
       else {
           A[i] = A[i2];
           i2++;
       }
       i++;
    }
    while (i1 <= mid) [
           A[i] = temp[i1];
           i1++; i++;
    }
}
```

Best case: O(1)
Worst case: O(n) (but smaller constants than the simple merge)

What is the best-case runtime for mergesort using fasterMerge? For what input?

Quicksort

1. Pick an element of the array as a *pivot,* and *partition* the array into elements less than the pivot, followed by the pivot, followed by the elements greater or equal to the pivot. (*Divide*)
2. Sort the two partitions recursively (*Conquer*)
3. Done (because every pivot is in its final position) (*Combine*)

Simplest case is when there is 0 or 1 elements

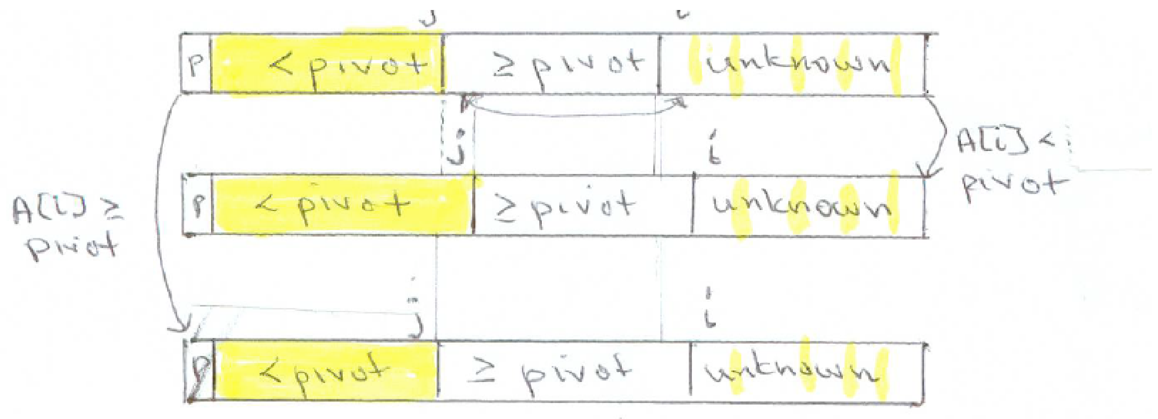
```
public static void quicksort(int[] A, int low, int high) {
   if (high <= low) {
     return;
   }
   else {
      int pivotIndex = partition(A, low, high);
      quicksort(A, low, pivotIndex - 1);
      quicksort(A, pivotIndex + 1, high);
   }
}
```

You can use any element as a pivot. The first one is convenient.

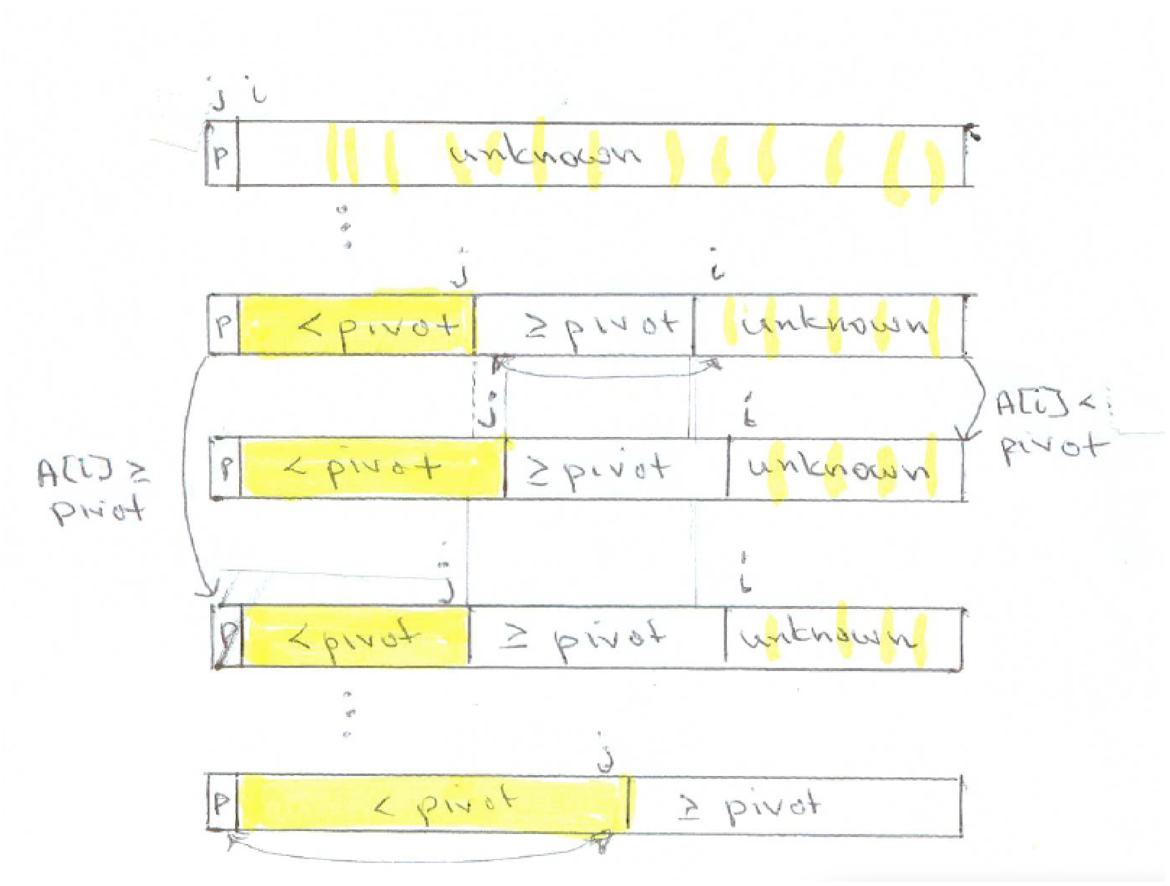
Below is the loop invariant for partition. The index j is the index of the last element known to be less than the pivot. The index i is the index of the next element to put into <pivot region or >= pivot region. There are two cases:

- The next element is < than the pivot. Swap it with the first >= pivot element, and increment both i and j.
- The next element >= pivot. Increment i.

In both cases the invariant stays the same for the next iteration.



What is the starting and ending conditions of the loop invariant?



Notice that j (or lessIndex in the code below) is conveniently the place where the pivot should go as its final position.

```
// Precondition: low < high
// Partitions array into elements < pivot, pivot, >= pivot
// Returns the index of the pivot's final position
public static int partition(int[] a, int low, int high) {
   int pivot = A[low];  // first element is the pivot
   int lessIndex = low; // index of rightmost element < pivot

   for (int i = low+1; i <= high; i++) {
        if (A[i] < pivot) {
           lessIndex++;
           swap(A, lessIndex, i);
       }
   }
   swap(A, low, lessIndex);
   return lessIndex;
}
```

Partition takes O(n) time.