Java 2 Sorting

- We talked about Selection and Insertion in 1210

- **Input:** List a of N elements, with no assumptions of order.

- **Output:** A permutation of the elements in a such that it is in ascending order.

- **Algorithms:** Selection sort, insertion sort, bubble sort, shaker sort, quick sort, merge sort, heap sort, sample sort, shell sort, solitaire sort, …….

- An **inversion** is a pair of elements that are out of order
  - **exchange** = swap the location of the inverted elements.
  - *sorting could be seen as a sequence of exchanges.*
- A list in reverse order would have the maximum number of inversions, which, at worst will have $O(n^2)$ complexity.

Detecting an inversion:

```
private boolean less(Comparable x, Comparable y) {
    return x.compareTo(y) < 0;
}
```

Correcting an inversion:

Swap the data found from less()

Properties of sorts:

- **Comparison sort:** the only assumption about the data being sorted is that the data elements can be compared to each other.
- **In-place:** The list itself is rearranged and only a constant amount of extra space is required
- **Adaptive:** Running time is affected by initial state of input.
- **Stable:** Equal elements maintain the same relative order.

Selection Sort:
- Walk from left to right through the array.
- On each step, **select** the element that in the current location in sorted order and put it there.
- After k steps, the first k elements are in sorted order an are in their final position.

```
public static void selectionSort(Comparable[] a) {
    int N = a.length;


    for (int i = 0; i < N-1; i++) {
        int min = i;
        for (int j = i + 1; j < N; j++) {
            if (less(a[j], a[min]))
                min = j;
        }
        swap(a, i, min);
    }
}
```

- **Selection sort is $O(N^2)$**

- This is **not adaptive** to its input, so all arrangements of data in the array will require a quadratic amount of work.

Insertion Sort:
- Walk from left to right through the array.
- On each step, **insert** the element in the current location in sorted order to its left.
- After k steps, the first k+1 elements are in sorted order relative to each other.

```
public static void insertionSort(Comparable[] a) {
    int N = a.length;

    for (int i = 0; i < N; i++) {
        int j = i;
        while ((j > 0) && (less(a[j], a[i]))) {
            swap(a, j, j-1);
            j—;
        }
    }
}
```

- **Insertion Sort is O(N^2)**
- This **is adaptive** to its input, so some arrangements of data in the array will require less work than others.

Neither insertion sort or selection sort scale well at all.
Ways to make this more efficient:
1. Impose additional constraints on the problem.
   a. EX: The values being sorted must be integers in a given range. => **Counting Sort O(N).**
2. Use a divide-and-conquer algorithm.
   a. EX: Divide the array in half, sort each half, then combine the sorted halves. => **Merge Sort O(NlogN)**

We have 3 in toolbox:
1. Linear Scan
2. Sort-First
3. Divide-and-Conquer

Divide and Conquer
- Is an algorithm design technique where we **divide** (partition) the problem into two or more smaller parts, solve (**conquer**) each part, and then **combine** the solutions for the parts into a solution for the whole problem.
  - EX: Find the maximum element in an array.
  - *Typical strategy: iteration*
  - *Divide and Conquer Strategy:*
    - **Divide:** Partition in two halves
    - **Conquer:** Find the largest in each half.
    - **Combine:** Pick the larger of these two halves.
  - Divide and Conquer algorithms are usually expressed **recursively**, and the division is repeated until each part is small enough to be solved directly or trivially.
  - Change the signature.
    - public int max(int[] a, int left, int right) {. . .}

Recursion:
- Is a means of specifying the solution to a problem in terms of solutions to smaller instances of the same problem.
- The *smallest* instance of the problem must have a solution that is known or trivial to compute; that is,

one that does not involve recursion.

- Any solvable problem can be with either iterative or recursive techniques.

- The factorial of a positive integer n!, is the product of all positive integers less than or equal to n.
  - Useful for finding permutations.

This is the iterative form of the factorial:

```
public int factorial(int n) {
	int fact = n;
	for (int i = n - 1; i > 0; i—) {
		fact = fact * i;
	}
return fact;
}
```

This is recursion:

- 5! = 5 * 4! = 5 * 4 * 3! = 5 * 4 * 3 * 2! = 5 * 4 * 3 * 2 * 1! = 5 * 4 * 3 * 2 * 1 = 120
- In general:
  - n! = 1 if n = 1 or n! = n * (n - 1)! if n > 1

1. A solution to the smallest instance of the problem:
   a. This is the **base case**
2. A rule for reducing all other instances of the base
   a. This is called the **recursive step** or the **reduction step**

This the recursive form of the factorial problem:

```
public int factorial(int n) {
	if (n == 1) {
		return 1;
	}
	return n * factorial(n - 1);
}
```

Both of the algorithms require the same amount of work.

Rewriting the search to have a starting value:
The iterative version won't differ except i = start instead of 0 initially.

Written recursively:

```
public boolean search(int[] a, int target, int start) {
		if (start == a.length) { //BASE CASE//
			return false;
		}
		if (a[start] == target) {
			return true;
		}

		return search(a, target, start + 1);
	}
}
```

Writing max with recursion:

```
public static int max(int[] a, int l, int r) {
```

```
    if (l == r) {
        return a[l];
    }

    int mid = (l + r) / 2;
    int lm = max(a, l, mid);
    int rm = max(a, mid + l, r);

    if (lm > rm)
        return lm;
    else
        return rm;
}
```

D&C Sorting

- A merge Sort is an example of a **Coordinated Linear Scan**

```
public void mergeSort(Comparable[] a, int left, int right) {
    if (right <= left) return;

    int mid = left + (right -left) / 2;
    mergeSort(a, left, mid);
    mergeSort(a, mid + 1, right);

    merge(a, left, mid, right);
}
```

Disadvantages of Merge Sort
- Needs an extra array as a field.

Quicksort
- Another example of a Divide and Conquer sort
- It has **O(NlogN)**, but only in the average case. In the worst case, **O(N^2)**. We can almost always guarantee the faster time complexity.

Divide: Select a pivot then partition the array so that:
- pivot in is its correct sorted position
- no larger element is to the left of the pivot
- no smaller element is to the right of the pivot

Conquer:
- Sort each partition(recursively)

Combine:
- Nothing to do

```
public void qsort(Comparable[] a, int left, int right) {
    if (right <= left)
        return;

    int j = partition(a, left, right);
    qsort(a, left, j-1);
    qsort(a, j+1, right);
}

private int partition(T[] a, int left, int right, int pivotIndex) {
    T pivot = a[pivotIndex];
```

```
        swap(a, pivotIndex, right); //move pivot to the end
        int  p = left; // p will become the final index of pivot

        for (int i = left; i < right; i++) {
                if (less(a[i], pivot)) {
                        swap(a, i, p);
                        p++;
                }
        }
        swap(a, p, right); // move pivot to its correct location
        return p;
}
```

The choice of pivot value determines the size of each partition, and therefore determines the number of divide steps that will be necessary.

- **Best case** pivot choices at each step lead to partitions being about the same size. Gives Average Case time complexity.
- **Worst case** pivot choices at each step lead to one partition that is empty. Gives Worst Case time complexity.

Choosing a pivot value:

- Find the median value. (Gives **O(NlogN)** doing it with linear scan.)
  - Give **O(N)** using a selection algorithm
- Find the median of three (Choose three elements), first, middle, and last, and then use the median of those 3 values.
- Randomly pick, PRNG
- Shuffle once, pick first element
  - Randomize the order of elements in the array once up front.

A “Randomized” quicksort:

```
public void quicksort(Comparable[] a) {
        shuffle(a);
        qsort(a, 0, a.length - 1);
}
```

- Unless it was an array of duplicates, then it is very difficult to cause the Worst Case Time Complexity.
- Java uses quicksort for primitives.
- Java uses mergesort for references (for stability reasons because order is preserved).