Last time:
1. Inheritance
2. comparison to composition

Today:
1. Complexity of Bubble Sort
2. Recursion
3. MergeSort
Main Sorting Algorithms

- Bubble sort
- Selection sort
- Insertion sort
- Heap sort
- Merge sort
- Quicksort
- …
Bubble Sort — what is the complexity?

```java
public static void BS(Comparable[] c) {
    for (int i = 0; i < c.length-1; i++) {
        for (int j = 0; j < c.length-i-1; j++) {
            if (c[j].compareTo(c[j+1]) > 0) {
                Comparable temp = c[j];
                c[j] = c[j+1];
                c[j+1] = temp;
            }
        }
    }
}
```

Number of times `compareTo` is called?
Analyzing Bubble Sort

- **Complexity analysis**
  - For each inner loop iteration:
    - One comparison
  - **How many iterations?**
    - Depends on i: n-i-1, where n is number of elements in array c
    - **Worst-case:** i = 0
    - **Pass 0:** n-1 iterations
      - **Pass 1:** n-2 iterations
      - etc.
    - 1+2+3+…+(n-2)+(n-1)
    - Almost looks like Gauss’ formula!
      - 1+2+3+…+(n-2)+(n-1)+n = n(n+1)/2
      - So 1+2+3+…+(n-2)+(n-1) = n(n+1)/2 − n
  - **We only care about the highest order term:** O(n^2)
There Are Better Sorting Algorithms!

- $O(n \log_2 n)$ is possible
- We will look at one such algorithm: **Merge sort**
First, more talk about recursion

- **Factorial**
  ```
  int factorial(int n) {
    if (n==1) { return 1; } // Base case to stop recursion
    return n * factorial(n-1);
  }
  ```

- Factorial uses “tail recursion” — one single recursive call at the end. Not too inefficient, but still less efficient than using loops (can get call stack overflow).

- **Fibonacci**
  ```
  int fib(int n) {
    if (n==1 || n==2) { return 1; } // Two base cases to stop recursion
    return fib(n-1) + fib(n-2);
  }
  ```

- Fib is doubly recursive—very hard to illustrate using a stack. Tree illustration:
MergeSort Pseudo-Code

- Pseudocode for MergeSort:
  ```
  Mergesort(x[]) {
    if (x.length == 1) {
      return
    }
    let a = left half of x
    let b = right half of x
    mergesort(a)
    mergesort(b)
    x = merge(a, b)
  }
  ```

- The sorting work is done in merge! Java prototype:
  ```
  Comparable[] merge(Comparable[] a, Comparable[] b)
  ```
  - Precondition for merge: a and b are already sorted
  - Postcondition for merge: answer is sorted
Analyzing MergeSort: Divide and Conquer Algorithm

- Divide into smaller problems of size n, n/2, n/4, n/8, etc.
- Conquer: This is the merge step of the Mergesort algorithm
  - First merge arrays of size 1 into arrays of size 2
  - Then arrays of size 2 into arrays of size 4
  - … until we merge arrays of size n/2 into one array of size n
- Complexity of Mergesort
  - At each level, there are n steps
  - How many levels are there?
    - $\log_2 n$
  - So how many steps total are there?
    - $n \log_2 n$
    - So MergeSort is $O(n \log n)$
- Is this a good run time??
  - Yes! Theorem: Fastest possible “comparison based sort” is $O(n \log n)$
  - Merge-sort is optimal (and many others too)
- Are there faster algorithms? Yes! $O(n)$—but these require further structure beyond “comparable.”
  - Example: Radix sort: $O(n)$ — Can sort students by ID number really quickly.
Show that MergeSort works using Induction

- **Base cases:**
  - Show that an array of size $k = 1$ is sorted by MergeSort: Trivially sorted (since one element). TRUE.
  - Show that an array of size $k = 2$ is sorted by MergeSort: Divide in half, each of the halves is sorted (see case above $k = 1$); the merge operation takes two numbers and puts the in the right order. TRUE.

- **Inductive step:** Show that if it is true for $1/2n$, it is also true for $n$. Since each of the $1/2n$ are sorted, the merge step will place elements in order in an array of size $n$ in sorted order. TRUE.

- Thus, by induction, MergeSort correctly produces a sorted array of size $n$. 