Week 12 - Wednesday

COMP 2100

Last time

- What did we talk about last time?
  - Exam 2
- Before that:
  - NP-completeness
  - Review

Questions?

Project 4

Assignment 6

What do we want from sorting?

Characteristics of a sort

- Running time
  - Best case
  - Worst case
  - Average case
- Stable
  - Will elements with the same value get reordered?
- Adaptive
  - Will a mostly-sorted list take less time to sort?
- In-place
  - Can we perform the sort without additional memory?
- Simplicity of implementation
  - Relates to the constant hidden by Big Oh
- Online
  - Can sort as values arrive

Insertion Sort

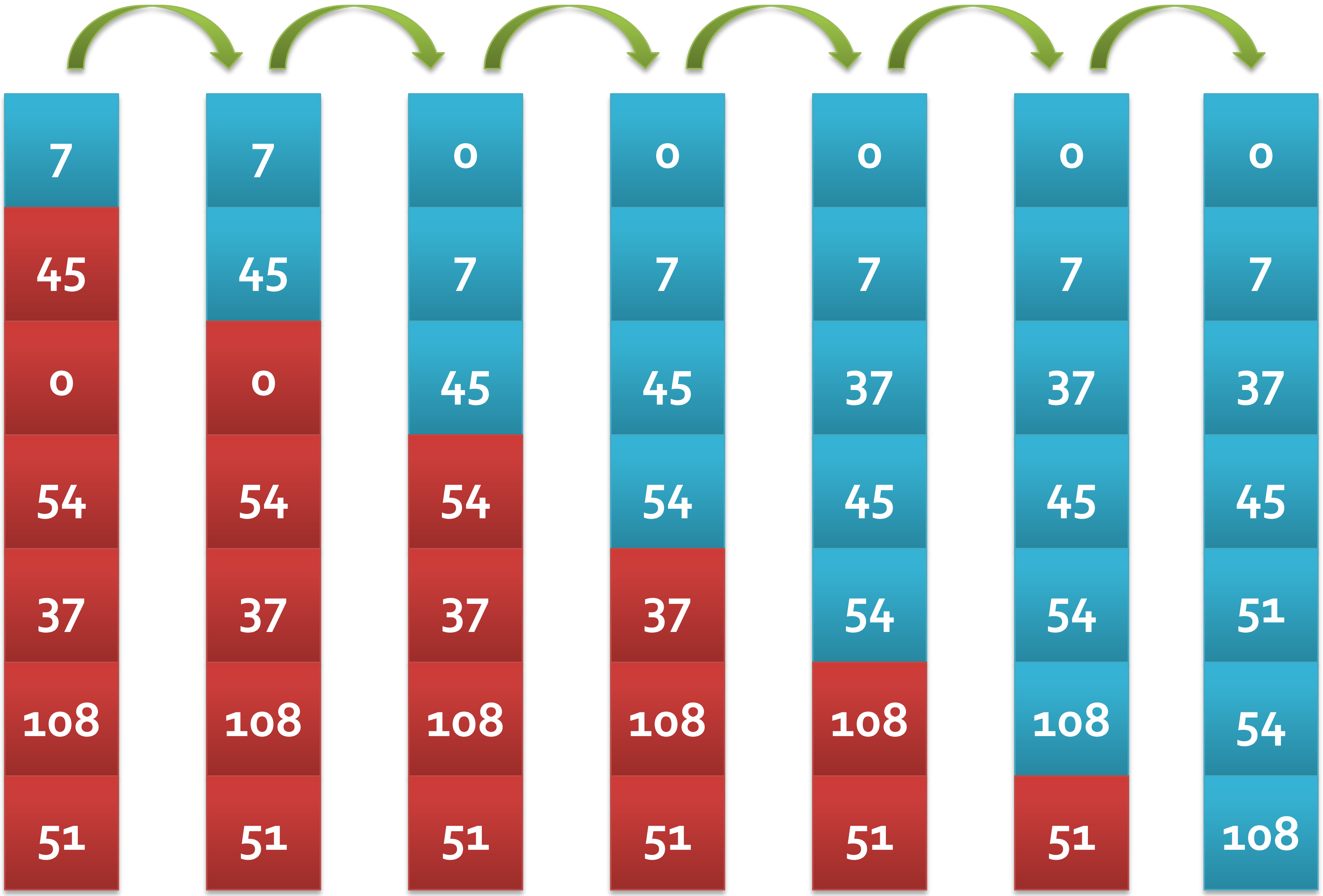
Insertion sort

- Pros:
  - **Best** case running time of O(***n***)
  - Stable
  - Adaptive
  - In-place
  - Simple implementation (one of the fastest sorts for 10 elements or fewer!)
  - Online
- Cons:
  - Worst case running time of O(n^2)

Insertion sort algorithm

- We do ***n*** – 1 rounds
  - For round ***i***, assume that the elements 0 through ***i*** – 1 are sorted
  - Take element ***i*** and move it up the list of already sorted elements until you find the spot where it fits

Insertion sort example



Insertion Sort Implementation

Merge Sort

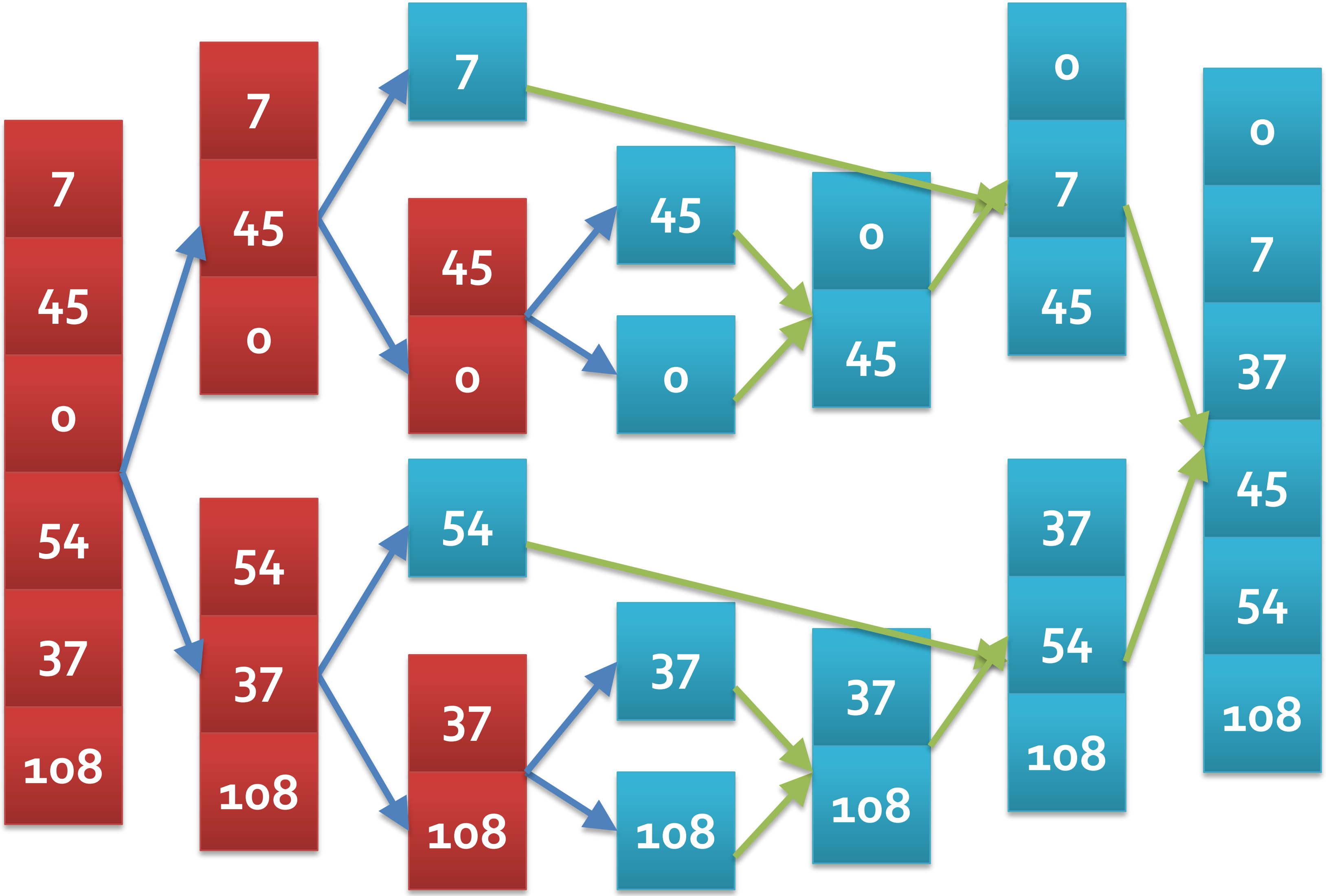
Merge sort

- Pros:
  - **Best**, **worst**, and **average** case running time of O(n log n)
  - Stable
  - Ideal for linked lists
- Cons:
  - Not adaptive
  - Not in-place
    - O(n) additional space needed for an array
    - O(log n) additional space needed for linked lists

Merge sort algorithm

- Take a list of numbers, and divide it in half, then, recursively:
  - Merge sort each half
  - After each half has been sorted, merge them together in order

Merge sort example



Merge Sort Implementation

Merge sort revisited

- We implemented merge sort before in a naïve way
  - Break the arrays down into smaller arrays
  - Recursively sort them
  - Merge them back together
- However, creating new arrays is an expensive memory operation
  - Creating very large arrays is expensive because they have to be cleared out in Java
  - Creating lots of small arrays has a lot of overhead
- A standard approach to improve performance is to use one extra scratch array that's the same size as the original array
- We won't need to do any other allocation beyond that

Merge sort methods

```
public static void mergeSort(double[] values) {
  double[] scratch = new double[values.length];
  mergeSort(values, scratch, 0, values.length);
}

private static void mergeSort(double[] values, double[]
  scratch, int start, int end) {
  …
}

private static void merge(double[] values, double[]
  scratch, int start, int mid, int end) {
  …
}
```

Quicksort issues

- Everything comes down to picking the right pivot
  - If you could get the median every time, it would be great
- A common choice is the first element in the range as the pivot
  - Gives O($\boldsymbol{n}^2$) performance if the list is sorted (or reverse sorted)
  - Why?
- Another implementation is to pick a random location
- Another well-studied approach is to pick three random locations and take the median of those three
- An algorithm exists that can find the median in linear time, but its constant is **HUGE**

Next time…

- Quicksort
- Counting sort

Reminders

- Start on Project 4
- Work on Assignment 6
  - Due Friday
- Read Sections 2.1 - 2.3 and 5.1