Precept Outline
- Course Introduction
- Induction

Relevant Book Sections
- The (Inverse) Ackermann Function
- Union-by-Height and Path Compression

A. Introduction

We're pretty psyched for you to see what we've got in store, but, before we let you loose, here are just a few words about the format of precept in this course:

The exercises are meant to be done in pairs. We want to encourage you to talk about the details of algorithms and data structures with a peer so you can help fill in the blind spots of each other.

These exercises are not graded. You don't have to hand in any solutions and we won't grade any of your precept work. Students are strongly encouraged to ask questions about the problems. The solutions to each exercise will be released after all precepts are done.

You are not expected to complete all of the problems in each handout. These handouts are intended for practice, and explicitly designed to *exceed* what can feasibly be completed in precept. Indeed, it's very unlikely to happen in any precept. Some of the problems are marked as "optional", which means that they are outside the scope of the course and are intended to be bonus challenge problems.

Attendance is mandatory. Your preceptor will keep track of your attendance (except for this first week), which will count towards 2.5% of your grade.

B. Advanced Precept Problems

Part 1: Useful Identities

We will make *extensive* use (really) of two identities throughout this course: for partial sums of arithmetic and (special cases of) geometric progressions.

Let's start by proving these identities: *formally* show that

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

and

$$\sum_{i=0}^{n-1} 2^i = 1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1.$$

Part 2: More (Somewhat) Useful Identities

Prove that

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \sim \frac{n^3}{3}$$

and

$$H_n \coloneqq \sum_{i=1}^{n} \frac{1}{i} \sim \ln n.$$

Part 3: Union-by-Height and Path Compression

Prove that weighted quick-union with the weight of a set defined as the *height* of the tree it represents (instead of its size) also yields $O(\log n)$ worst-case array accesses per operation. How would you implement this in Java?

Now, prove that weighted quick-union with path compression, using *ranks* as weights, also yields $\Theta(\log n)$ worst-case array accesses per operation.