Precept Outline
- Course Introduction
- Review of Lectures 1 and 2
- Problem Solving

Relevant Book Sections
- 1.4 (Analysis) and 1.5 (Union-Find)
- 1.1 and 1.2 (Java review)

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:

Precepts will be a mix of review, problem solving, and discussion. Each precept will start with a brief review of the lecture contents, followed by solving a mix of exercises in this handout. Many of the exercises follow the same format as the ones you will find in the midterm and final exams, so precepts will be good practice for those.

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. Review: Analysis and Union-Find

Your preceptor will briefly review key points of this week's lectures.

C. Analysis

Part 1: Loops

Runtime analysis can be tricky; even short and simple code can be hard to analyze correctly. The key to mastering this skill is practice, practice, practice. That's what we'll do in this part.

Determine the number of times the function `op()` is called *asymptotically*, as a function of n, using **both** tilde notation ($\sim$) and big Theta notation (Θ, i.e. order of growth).

1.

```
for (int i = 10; i < n + 5; i += 2)
    op();
```

2.

```
for (int i = 1; i <= n * n * n; i *= 2)
    op();
```

3.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < 100; j++)
        op();
```

4.

```
for (int i = 0; i * i < n; i++)
    for (int j = 1; j < n; j *= 3)
        op();
```

5.

```
for (int i = 0; i < n; i++)
    for (int j = 1; j < n; j *= 2)
        op();
```

6.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < 100; j++)
        for (int k = 0; k < n; k++)
            for (int l = k; l < n; l++)
                op();
```

D. Union-Find

Part 1: Find the Bug!

Consider the following (incorrect) implementation of `union()` in the *quick-find* data structure. Recall that the length-n `leader[]` array is initialized with `leader[i] = i` for all i, and that `find(i)` returns `leader[i]`.

```java
public void union(int p, int q) {
    for (int i = 0; i < leader.length; i++)
        if (leader[i] == leader[p])
            leader[i] = leader[q];
}
```

Find a number of elements n, a sequence of `union()` operations, an integer $0 \leq i < n$, and an integer $0 \leq j < n$, such that elements i and j belong to the same set but `find(i)` and `find(j)` return different values.

Part 2: Fall'22 Midterm Question

For the items below, assume we initialize a union-find data structure with n elements. Then, we perform the following sequence of `union()` operations: `union(0, 1)`, `union(0, 2)`, `union(0, 3)`, ..., `union(0, n - 1)`.

(a) How many total connected components (i.e., disjoint sets) does the resulting data structure contain?

(b) Assume that the data structure implementation is quick-find. How many array updates are made by these `union()` operations, as a function of n, in tilde notation? (Recall that our quick-find implementation of `union(p,q)` never changes `leader[q]`.)

(c) Assume that the data structure implementation is quick-union, and that we call `find(0)` after the sequence of operations above. How many array accesses would `find(0)` make as a function of n in Θ notation? (Recall that our quick-union implementation of `union(p,q)` operation never changes `parent[q]`.)

(d) Assume that the data structure implementation is weighted quick-union, and that we call `find(0)` after the sequence of operations above. How many array accesses, as a function of n in Θ notation, would `find(0)` make? (Recall that our weighted quick-union implementation of `union(p,q)` operation changes `parent[q]` if both trees have the same size.)