YEAH A5

Bag'O Big-O

Assignment Logistics

- This assignment is due Friday, Feb 19th at class time, with the same late policy as usual.

Assignment Logistics

- This assignment is due Friday, Feb 19th at class time, with the same late policy as usual.
- You’re also welcome to work in pairs on this assignment.

Assignment Logistics

- This assignment is due Friday, Feb 19th at class time, with the same late policy as usual.
- You're also welcome to work in pairs on this assignment.
- We **strongly** recommend that you finish the midterm **before** starting this assignment!

Assignment Logistics

- This assignment is due Friday, Feb 19th at class time, with the same late policy as usual.
- You're also welcome to work in pairs on this assignment.
- We **strongly** recommend that you finish the midterm **before** starting this assignment!
- Don't worry, we designed this assignment to be lighter than the others!

Assignment Overview

This assignment has two parts:

Assignment Overview

This assignment has two parts:

1. Big-O Analysis

Assignment Overview

This assignment has two parts:

1. Big-O Analysis
2. Recursive Combine

Part 1: Big-O Analysis

- In this part, we’ve written 13 C++ functions for you to analyze.

Part 1: Big-O Analysis

- In this part, we've written 13 C++ functions for you to analyze.
- We've also given you access to a runtime plotter that displays the relationship between *n*, the size of the input, and the execution time of the function calls.

Part 1: Big-O Analysis

- In this part, we've written 13 C++ functions for you to analyze.
- We've also given you access to a runtime plotter that displays the relationship between *n*, the size of the input, and the execution time of the function calls.
- With these two things, all you need to do is determine the runtime of each of the 13 functions, and log your answers in `BigOAnswers.txt`

Part 1: Big-O Analysis

Let’s review some Big-O Basics!

Part 1: Big-O Analysis

- Big-O notation is a way of quantifying the rate at which some quantity grows.
- We can use it to roughly estimate how well our program scales with increasingly large input.
- Let’s see some examples!

Part 1: Big-O Analysis - O(1)

```
void foo1(const Vector<int>& input) {
    cout << "hello world!" << endl;
}
```

Some functions exhibit behaviors independent of the size of their inputs.

Part 1: Big-O Analysis - O(1)

```
void foo1(const Vector<int>& input) {
    cout << "hello world!" << endl;
}
```

```
void foo2(const Vector<int>& input) {
    for (int i = 0; i < 100000000; i++) {
        cout << i << endl;
    }
}
```

They can do a lot of work, but in Big-O they are still constant.

Part 1: Big-O Analysis - O(N)

```
void foo2(const Vector<int>& input) {
    for (int i = 0; i < input.size(); i++) {
        cout << input[i] << endl;
    }
}
```

Other functions have runtimes that grow LINEARLY with the size of the input.

Part 1: Big-O Analysis - O(N)

```
void foo2(const Vector<int>& input) {
    for (int i = 0; i < input.size(); i++) {
        cout << input[i] << endl;
    }
}
```

```
void foo2(const Vector<int>& input) {
    for (int i = 0; i < input.size(); i++) {
        cout << input[i] << endl;
    }
    for (int i = 0; i < input.size() * 2; i++) {
        cout << input[i/2] << endl;
    }
    for (int i = 0; i < 137; i++) {
        cout << "more printing!" << endl;
    }
}
```

Again, constants don't matter.

Part 1: Big-O Analysis - O(N^2)

```
void foo5(const Vector<int>& input) {
    for (int i = 0; i < input.size(); i++) {
        for (int j = 0; j < input.size(); j++) {
            cout << i << j << endl;
        }
    }
}
```

Runtimes can be quadratic, and potentially much higher.

Part 1: Big-O Analysis - O(N^2)

```
void foo5(const Vector<int>& input) {
    for (int i = 0; i < input.size(); i++) {
        for (int j = 0; j < input.size(); j++) {
            cout << i << j << endl;
        }
    }
}
```

```
void foo6(Vector<int>& input) {
    while (!input.isEmpty()) {
        input.remove(0);
    }
}
```

However, a function's runtime is not always determined by the number of functions it calls.

Part 1: Big-O Analysis

Some general rules:

- Two parallel blocks of code add their runtime.

Part 1: Big-O Analysis

Some general rules:

- Two parallel blocks of code add their runtime.

```
void foo2(const Vector<int>& input) {
    for (int i = 0; i < input.size(); i++) {
        cout << input[i] << endl;
    }
    for (int i = 0; i < input.size() * 2; i++) {
        cout << input[i/2] << endl;
    }
    for (int i = 0; i < 137; i++) {
        cout << "more printing!" << endl;
    }
}
```

Part 1: Big-O Analysis

Some general rules:

- Two parallel blocks of code add their runtime.
- Nested code multiplies its runtime.

Part 1: Big-O Analysis

Some general rules:

- Two parallel blocks of code add their runtime.
- Nested code multiplies its runtime.

```
void foo5(const Vector<int>& input) {
    for (int i = 0; i < input.size(); i++) {
        for (int j = 0; j < input.size(); j++) {
            cout << i << j << endl;
        }
    }
}
```

Part 1: Big-O Analysis

Some general rules:

- Two parallel blocks of code add their runtime.
- Nested code multiplies its runtime.

```
void foo6(Vector<int>& input) {
    while (!input.isEmpty()) {
        input.remove(0);
    }
}
```

Part 1: Big-O Analysis

Some general rules:

- Two parallel blocks of code add their runtime.
- Nested code multiplies its runtime.
- Recursive function:
  - How many calls are there (in terms of the size of input N) * how much work is done per call

Part 1: Big-O Analysis

A few notes:

Part 1: Big-O Analysis

A few notes:

- You’ll probably want to read up on the documentation for the various functions you find in the code. Ensure you know the runtimes of each, and don’t take anything for granted!

Part 1: Big-O Analysis

A few notes:

- You'll probably want to read up on the documentation for the various functions you find in the code. Ensure you know the runtimes of each, and don't take anything for granted!
- Just in case it wasn't clear, there's no need to write/modify any code in this part of the assignment!

Part 1: Big-O Analysis

A few notes:

- You’ll probably want to read up on the documentation for the various functions you find in the code. Ensure you know the runtimes of each, and don’t take anything for granted!
- Just in case it wasn’t clear, there’s no need to write/modify any code in this part of the assignment!
- With respect to runtime plots, expect somewhat noisy data plots for small-input levels. Additionally, you might see odd spikes at certain points in your graphs -- this is probably because your computer had other things going on in the background!

Questions about Part 1?

Part 2: Combine

- For part 2 of this assignment, your job is to combine a **collection of sorted lists** into a **single sorted list.**

Part 2: Combine

- For part 2 of this assignment, your job is to combine a **collection of sorted lists** into a **single sorted list.**
- Let's say that again to be clear -- starting with a `Vector<Vector<DataPoint>>`, your job is to merge the internal sorted `Vector<DataPoint>`'s into a single sorted `Vector<DataPoint>`.

Part 2: Combine

- For part 2 of this assignment, your job is to combine a **collection of sorted lists** into a **single sorted list.**
- Let's say that again to be clear -- starting with a **Vector<Vector<DataPoint>>**, your job is to merge the internal sorted **Vector<DataPoint>**'s into a single sorted **Vector<DataPoint>**.

```
Vector<DataPoint> combine(const Vector<Vector<DataPoint>>& dataPoints)
```

Part 2: Combine

- For part 2 of this assignment, your job is to combine a **collection of sorted lists** into a **single sorted list.**
- Let's say that again to be clear -- starting with a **`Vector<Vector<DataPoint>>`**, your job is to merge the internal sorted **`Vector<DataPoint>`**'s into a single sorted **`Vector<DataPoint>`**.
- For savy algorithmic fiends, what you'll really be doing is *implementing the **merge** part of the recursive sorting algorithm **mergesort**!* Cool beans!

Part 2: Combine

- For part 2 of this assignment, your job is to combine a **collection of sorted lists** into a **single sorted list.**
- Let's say that again to be clear -- starting with a **Vector<Vector<DataPoint>>**, your job is to merge the internal sorted **Vector<DataPoint>**'s into a single sorted **Vector<DataPoint>**.
- For savy algorithmic fiends, what you'll really be doing is *implementing the **merge** part of the recursive sorting algorithm **mergesort**!* Cool beans!
- For those interested, a DataPoint is a little struct that looks like this:

```
struct DataPoint {
  string name; // Name of this data point; varies by application
  int weight; // "Weight" of this data point. Points are sorted by weight.
};
```

Part 2: Combine

Here's how we want you to approach this problem:

Part 2: Combine

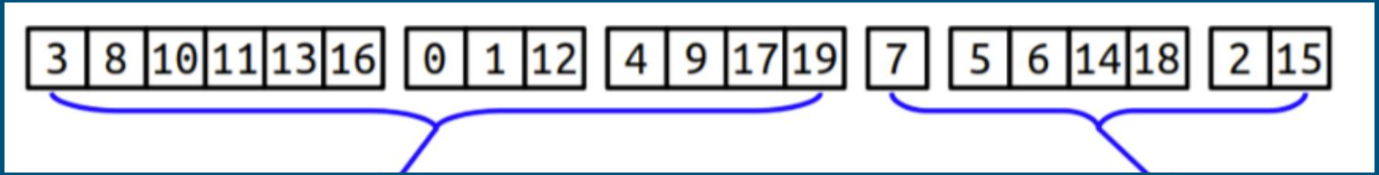
Here's how we want you to approach this problem:

1. Assuming you start with a **`Vector`** of *k* sequences, begin by splitting this collection into 2 **`Vector`**'s of *k/2* sequences.

Part 2: Combine

Here's how we want you to approach this problem:

1. Assuming you start with a **`Vector`** of *k* sequences, begin by splitting this collection into 2 **`Vector`**'s of *k/2* sequences.



[Above: A collection of sorted sequences being regrouped into 2 collections of sequences of size roughly *k/2*.]

Part 2: Combine

Here's how we want you to approach this problem:

1. Assuming you start with a **`Vector`** of *k* sequences, begin by splitting this collection into 2 **`Vector`**'s of *k/2* sequences.
2. Call your **`combine()`** function recursively on both of the **`Vector`**'s

Part 2: Combine

Here’s how we want you to approach this problem:

1. Assuming you start with a **`Vector`** of *k* sequences, begin by splitting this collection into 2 **`Vector`**’s of *k/2* sequences.
2. Call your **`combine()`** function recursively on both of the **`Vector`**’s

```
Vector<DataPoint> combine(const Vector<Vector<DataPoint>>& dataPoints)
```

As you can see, each call returns a **`Vector<DataPoint>`**’s that is sorted!

Part 2: Combine

Here's how we want you to approach this problem:

1. Assuming you start with a **`Vector`** of *k* sequences, begin by splitting this collection into 2 **`Vector`**'s of *k/2* sequences.
2. Call your **`combine()`** function recursively on both of the **`Vector`**'s

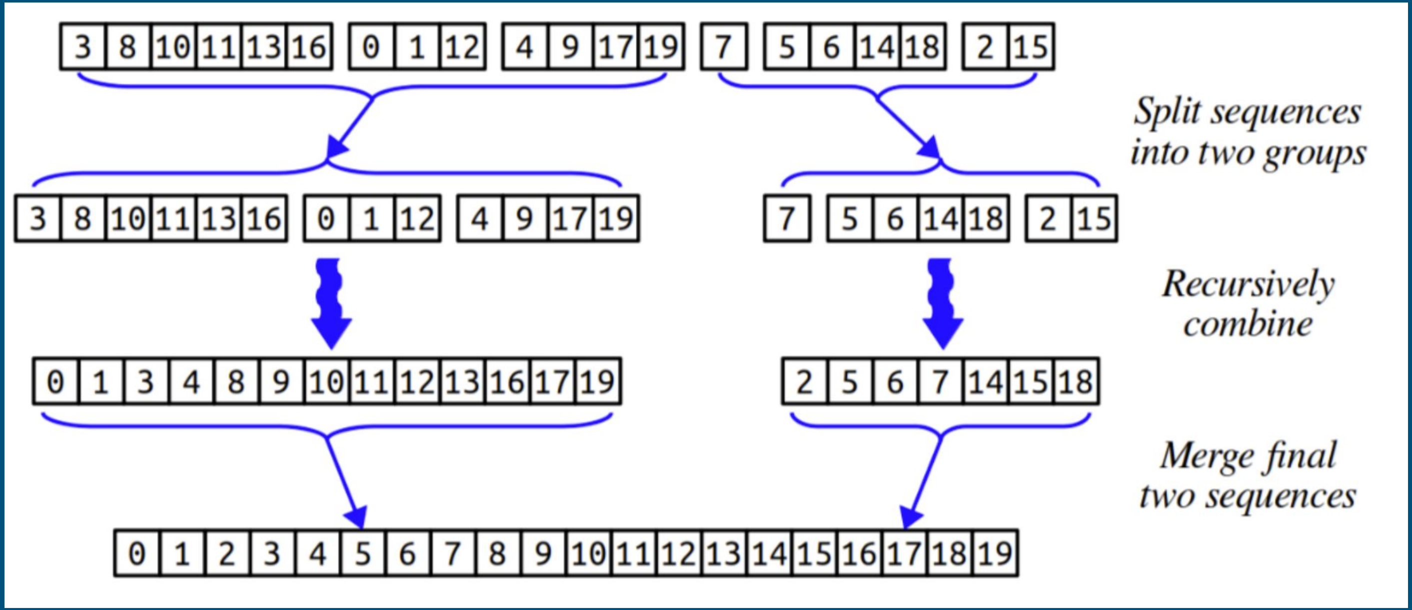
```
Vector<DataPoint> combine(const Vector<Vector<DataPoint>>& dataPoints)
```

   As you can see, each call returns a **`Vector<DataPoint>`**'s that is sorted!

3. Use the **`merge()`** algorithm from class to combine these two sorted sequences into a single sorted sequence, which you'll return!

Part 2: Combine

At a high level, here's a visual of what's going on:



Part 2: Combine

Some implementation notes:

Part 2: Combine

Some implementation notes:

- You'll want this code to run in `O(n log k)` time, where *n* is the number of elements total and *k* is the number of sequences originally. Think about why this is the case.

Part 2: Combine

Some implementation notes:

- You'll want this code to run in `O(n log k)` time, where *n* is the number of elements total and *k* is the number of sequences originally. Think about why this is the case.
- If you use the `merge()` routine from class, recall that removing from the front of a `Vector` does not run in O(1) time. You're going to need to figure out a workaround that preserves the required runtime.

Part 2: Combine

A few more implementation notes:

- Just to be clear -- you'll be determining ordering based on the **weight** field of the **`DataPoint`** struct. Ties can be broken arbitrarily.

Part 2: Combine

A few more implementation notes:

- Just to be clear -- you'll be determining ordering based on the **weight** field of the **`DataPoint`** struct. Ties can be broken arbitrarily.
- The sequences you merge together might not be the same size -- some might even be empty! Your code should correctly combine them all.

Part 2: Combine

A few more implementation notes:

- Just to be clear -- you'll be determining ordering based on the **weight** field of the `DataPoint` struct. Ties can be broken arbitrarily.
- The sequences you merge together might not be the same size -- some might even be empty! Your code should correctly combine them all.
- Before you submit, run the "Time Tests" portion of the GUI and verify visually that your code runs in `O(n log k)` time. After completing part 1 of this assignment, we think you'll know how to identify this runtime on a plot!

Any Questions?