[In Class] Lesson 15 (11/27)

What is "runtime"?

You're finally near the the end of the Introduction to Programming course series, so you have learned lots of different ways to solve any given problem. Now that you have a full tool box, it's time to learn how to pick the best tool for the job.

For example, we've learned about a number of different **data structures** such as ArrayIntList, LinkedIntList and IntTree. Each of these structures offers a way to add a value to the collection, access a value in the collection and remove a value from the collection. However, based on their design and your particular situation each of these functions may have very different **runtimes**.

We can use a timer to measure how long code takes to run, but this number will be highly variable based on the hardware and state of the computer the code is running on. You can use the code provided here to see how long it takes to add numbers to the different data structures we've learned about on your computer.

See graph from class **here.**

Intro to Runtime Analysis

As you've seen the data we get from timing code execution can be difficult to work with, so we need a conceptual way to assess how long it will take code to run independent of machine specifics.

Runtime analysis is a way of approximating how many instructions will be executed by a piece of code. We can use these approximations to compare different pieces of code based on how efficient they are. When approximating the instructions for code we assume all basic actions take a count of "1". We count the number of instructions in relation to the number of inputs or "n".

For example, consider the following code:

```java
public static void loopAnalysis(int n) {
    for (int i = 0; i < n; i++) {
        System.out.println("some basic action");
    }
}
```

Because this loop runs "n" times and executes 1 instruction each time, if n is 10 there will be approximately 10 instructions, if n is 100 there will be approximately 100 instructions and so on... Thus the runtime analysis of this code would be that is runs in **linear time** because the number of instructions are linearly related to the number of inputs.

Consider the code examples shown here. Approximate the number of instructions for a given data set then relate that approximation to the number of inputs.

Complexity Classes

The reason we only care about the approximate number of instructions executed by code is because the time it takes to run a single instruction is so small that getting a precise number doesn't give us that much more insight than the approximation.

When doing our analysis instead of focusing on counting the specific number of instructions, we instead only need to figure out the **complexity class**. In our previous examples, we met three complexity classes:

- **Constant** - $\mathcal{O}(1)$ - runtime is independent of the size of input - no repeating control structures
- **Linear** - $\mathcal{O}(n)$ - runtime is directly corelated to the size of input - single loop over input
- **Quadratic** - $\mathcal{O}(n^2)$ - runtime is squared corelated to the size of input - nested loop

See graph from class **here**

Let's explore another complexity class thanks to Trees.

- **Logarithmic -** $\mathcal{O}(\log(n))$ - runtime scales sub-linearly based on input size

And many many more! You can imagine there are an infinite number of polynomial classes $\mathcal{O}(n^3)$, $\mathcal{O}(n^4)$ and so on. A couple other common runtimes that show up in a lot of applications:

- **Loglinear** - $\mathcal{O}(n\log(n))$ - runtime of many algorithms to sort data
- **Exponential** - $\mathcal{O}(2^n)$ - runtime of many recursive-backtracking-algorithms that have choose-explore-unchoose behavior

---

Common Complexity Classes for Data Structures

One of the major benefits of spending this quarter implementing all of the foundational data structures, is that you can always understand what has to go on "under the hood" to complete an operation. It takes some practice to reason about data structure operations in terms of complexity classes, so we provide a reference sheet for common operations on *Java's implementations* of these data structures and their runtimes.

- `ArrayList`
  - `add(value)` (appending add) - $\mathcal{O}(1)$ (most of the time except when resizing)
  - `add(index, value)` - $\mathcal{O}(n)$
  - `get(index)` - $\mathcal{O}(1)$
  - `set(index, value)` - $\mathcal{O}(1)$
  - `remove(value)` - $\mathcal{O}(n)$
  - `remove(0)` (remove from front) - $\mathcal{O}(n)$
  - `indexOf(value)` - $\mathcal{O}(n)$
  - `contains(value)` - $\mathcal{O}(n)$
- `LinkedList`
  - `add(value)` - $\mathcal{O}(1)$
    - Java's linked list actually can do fast appending `add` since they also store a reference to the tail of the list
  - `add(index, value)` - $\mathcal{O}(n)$
  - `get(index)` - $\mathcal{O}(n)$
  - `set(index, value)` - $\mathcal{O}(n)$
  - `remove(value)` - $\mathcal{O}(n)$
  - `remove(0)` (remove from front) - $\mathcal{O}(1)$
  - `indexOf(value)` - $\mathcal{O}(n)$
  - `contains(value)` - $\mathcal{O}(n)$
- `TreeSet`
  - `add(value)` - $\mathcal{O}(\log(n))$
  - `remove(value)` - $\mathcal{O}(\log(n))$
  - `contains(value)` - $\mathcal{O}(\log(n))$
  - N/A Operations
    - `add(index, value)` - N/A
    - `get(index)` - N/A
    - `set(index, value)` - N/A

        - `remove(0)` (remove from front) - N/A
        - `indexOf(value)` - N/A
- `HashSet`
    - `add(value)` - $\mathcal{O}(1)$
    - `remove(value)` - $\mathcal{O}(1)$
    - `contains(value)` - $\mathcal{O}(1)$
    - N/A Operations
        - `add(index, value)` - N/A
        - `get(index)` - N/A
        - `set(index, value)` - N/A
        - `remove(0)` (remove from front) - N/A
        - `indexOf(value)` - N/A