Efficiency; Interfaces

public void run() {
    //for (int i = 0; i < 1000000; i++) {
    //    doLongCalculation();
    //    anotherAnalysis();
    //    solvePNP();
    //}
    System.out.println("Done!");
}
Questions From Last Time

- Does a constructor have to use all the fields specified in a class?
  Nope. It depends on what you’re trying to do.

- For class constants, why write “public”?
  We don’t technically have to. It’s just considered good style.

- public static final int vs. private static final int?
  If it’s private, clients can’t use it.

- Vim or Emacs?
  Vim is the way and the light.

- Waffles or Pancakes?
  Pancakes I guess?

- Is Euler self-aware?
  I’m not sure; I’ll have to ask him.

- (I’ll continue the game of Tic-Tac-Toe next time.)

- (Also, I’ll add this time’s pictures then too.)
Some Quick Debugging Tips

Style Tips

- Avoid “obvious comments”. The following is **BAD**.

```
1 //BAD BAD BAD BAD BAD BAD BAD
2 int count = 0; // Initializes a count of values
```

- Throw exceptions as early as possible in methods.

```
1 //BAD BAD BAD BAD BAD BAD
2 if (size > 0) {
3     //Do stuff
4 } else {
5     throw new IllegalArgumentException();
6 }        
```

- Avoid using constants that aren't clear. (Especially if there is a clearer way to write them.)

```
1 //BAD BAD BAD BAD BAD BAD
2 public static final int LENGTH_OF_JAVA = 4;
3 //BETTER
4 public static final int LENGTH_OF_JAVA = "JAVA".length();
```

- Don’t overcomment: a comment on every line is unreadable.
Some Quick Testing/Debugging Tips

![Testing/Debugging Tips](image)

- Check **EDGE CASES** (null, 0, capacity, etc.)
- Test running multiple methods one after another
  
  ```java
  (list.add(5); list.add(5); list.remove(0);
  list.add(5);...)
  ```
Oddly Prolific Questions.

- Is most of 143 “style” as opposed to “content”?
- How do TAs judge the “efficiency” of a solution?
What does it mean to have an “efficient program”? 

1 System.out.println("hello"); vs. 1 System.out.print("h");
2 System.out.print("e"); 2 System.out.print("e");
3 System.out.print("l"); 3 System.out.print("l");
4 System.out.print("l"); 4 System.out.print("l");
5 System.out.println("o"); 5 System.out.println("o");

OUTPUT

>> left average run time is 1000 ns.
>> right average run time is 5000 ns.

We’re measuring in NANOSECONDS!

Both of these run very very quickly. The first is definitely better style, but it’s not “more efficient.”
Comparing Programs: Timing

hasDuplicate

Given a sorted int array, determine if the array has a duplicate.

```java
public boolean hasDuplicate1(int[] array) {
    for (int i=0; i < array.length; i++) {
        for (int j=0; j < array.length; j++) {
            if (i != j && array[i] == array[j]) {
                return true;
            }
        }
    }
    return false;
}

public boolean hasDuplicate2(int[] array) {
    for (int i=0; i < array.length - 1; i++) {
        if (array[i] == array[i+1]) {
            return true;
        }
    }
    return false;
}
```

OUTPUT

```plaintext
>>> hasDuplicate1 average run time is 5254712 ns.
>>> hasDuplicate2 average run time is 2384 ns.
```
Comparing Programs: # Of Steps

Timing programs is prone to error:
- We can’t compare between computers
- We get noise (what if the computer is busy?)

Let’s **count** the number of steps instead:

```java
public int stepsHasDuplicate1(int[] array) {
    int steps = 0;
    for (int i=0; i < array.length; i++) {
        for (int j=0; j < array.length; j++) {
            steps++;
            // The if statement is a step
            if (i != j && array[i] == array[j]) {
                return steps;
            }
        }
    }
    return steps;
}
```

**OUTPUT**

```bash
>> hasDuplicate1 average number of steps is 9758172 steps.
>> hasDuplicate2 average number of steps is 170 steps.
```
This still isn’t good enough! We’re only trying a single array!

Instead, let’s try running on arrays of size 1, 2, 3, \ldots, 1000000, and plot:
Comparing Programs: Analytically

Runtime Efficiency

We’ve made the following observations:

- All “simple” statements (println("hello"), 3 + 7, etc.) take one step to run.
- We should look at the “number of steps” a program takes to run.
- We should compare the growth of the runtime (not just one input).

```java
1  statement1;
2  statement2;
3  statement3;
4
5  for (int i = 0; i < N; i++) {
6    statement4;
7  }
8
9  for (int i = 0; i < N; i++) {
10   statement5;
11   statement6;
12   statement7;
13   statement8;
14  }
```

- `3N` for the first loop.
- `5N + 3` for the second loop.
- `4N` for another loop.
- `N` for another loop.

```plaintext
1  statement1;
2  statement2;
3  statement3;
4
5  for (int i = 0; i < N; i++) {
6    statement4;
7  }
8
9  for (int i = 0; i < N; i++) {
10   statement5;
11   statement6;
12   statement7;
13   statement8;
14  }
```
We measure algorithmic complexity by looking at the growth rate of the steps vs. the size of the input.

The algorithm on the previous slide ran in $5N + 3$ steps. As $N$ gets very large, the “5” and the “3” become irrelevant.

We say that algorithm is $O(N)$ (“Big-Oh-of-$N$”) which means the number of steps it takes is linear in the input.

### Some Common Complexities

<table>
<thead>
<tr>
<th>$O(1)$</th>
<th>Constant</th>
<th>The number of steps doesn’t depend on $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
<td>If you double $n$, the number of steps doubles</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
<td>If you double $n$, the number of steps quadruples</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
<td>The number of steps gets infeasible at $n &lt; 100$</td>
</tr>
</tbody>
</table>
So, the entire thing is $O(N^2)$, because the quadratic term overtakes all the others.
### ArrayList Efficiency

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(val)</code></td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>add(idx, val)</code></td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td><code>get(idx)</code></td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>set(idx, val)</code></td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>remove(idx)</code></td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>$\mathcal{O}(1)$</td>
</tr>
</tbody>
</table>
What are the time complexities of these functions?

1. public static void numbers1(int max) {
   ArrayList<Integer> list = new ArrayList<Integer>(); // O(1)
   for (int i = 1; i < max; i++) {
      list.add(i); // O(1)
   }
}

2. public static void numbers2(int max) {
   ArrayList<Integer> list = new ArrayList<Integer>(); // O(1)
   for (int i = 1; i < max; i++) {
      list.add(i); // O(1)
      list.add(i); // O(1)
   }
}

vs.

1. public static void numbers1(int max) {
   ArrayList<Integer> list = new ArrayList<Integer>(); // O(1)
   for (int i = 1; i < max; i++) {
      list.add(i); // O(1)
   }
}

2. public static void numbers2(int max) {
   ArrayList<Integer> list = new ArrayList<Integer>(); // O(1)
   for (int i = 1; i < max; i++) {
      list.add(i); // O(1)
   }
}

O(n)
Investigating Our Answer With Pictures

numbers1

numbers2
public boolean is10(int number) {
    return number == 10;
}

public boolean two10s(int num1, int num2, int num3) {
    return (is10(num1) && is10(num2) && !is10(num3)) ||
           (is10(num1) && !is10(num2) && is10(num3)) ||
           (!is10(num1) && is10(num2) && is10(num3));
}

public void loops(int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.println(i + " " + j);
        }
    }
    System.out.println(N - i);
}

\(O(1)\)

\(O(n^2)\)

\(O(n^2)\)
It’s the WORST CASE!

```java
public static int has5(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);  // O(1)
        if (array[i] == 5) {  // O(1)
            return true;  // O(1)
        }
    }
    return false;  // O(1)
}
```

Sometimes, these will finish in fewer than `array.length` steps, but in the worse case, we have to go through the whole array. This makes both of them $O(n)$. 
//Inside the ArrayIntList class...

/** pre: size >= 1 otherwise throws IllegalStateException */

public int max() {
    if (this.size < 1) {
        throw new IllegalStateException();
    }

    int result = this.data[0];
    for (int i = 1; i < this.size; i++) {
        if (this.data[i] > result) {
            result = Math.max(result, this.data[i]);
        }
    }

    return result;
}

This code sucks! It's $O(n)$. Can we do it in $O(1)$?

Yes! Create a max field in the ArrayIntList class and update it when we add/remove.
private int slowMax() { //slowMax is \( O(n) \), because of the for loop.
    int result = this.data[0];
    for (int i = 1; i < this.size; i++) {
        if (this.data[i] > result) {
            result = Math.max(result, this.data[i]);
        }
    }
    return result;
}  

public void add(int index, int value) { //add is \( O(n) \)  
    this.size++; //\( O(1) \)
    this.grow(this.size); //\( O(n) \)
    this.checkIndex(index); //\( O(1) \)
    for (int i = this.size - 1; i > index; i--) { //\( O(n) \) (for loop)
        this.data[i] = this.data[i-1]; //\( O(1) \)
    }
    int oldValue = this.data[index]; //\( O(1) \)
    this.data[index] = value; //\( O(1) \)
    if (value > max) { this.max = value; } //\( O(1) \)
    else if (oldValue == max) { this.max = this.slowMax(); } //\( O(n) \) (slowMax)
}

public void remove(int index) { //remove is \( O(n) \)
    this.checkIndex(index); //\( O(1) \)
    int oldValue = this.data[index]; //\( O(1) \)
    for (int i = index; i < size - 1; i++) { //\( O(n) \) (for loop)
        this.data[i] = this.data[i+1]; //\( O(1) \)
    }
    this.size--; //\( O(1) \)
    if (this.max == oldValue) {
        this.max = this.slowMax(); //\( O(n) \) (slowMax)
    }
}
What are some different locking mechanisms for safes?

- Door Lock
- Combination Lock
- Padlock
- Digital Lock
- etc.

Note the following:

- All mechanisms have a way to “lock” and “unlock” the safe.
- Each mechanism works completely differently, is made up of different parts, and is used differently.
An interface specifies a group of behaviors and gives them a name. Classes can choose to implement interfaces which require them to implement all of the methods in the interface.

The idea is the same as with the safe: there might be multiple different ways to implement the interface.
public interface Shape {
    public double area();
    public double perimeter();
}

public class Circle implements Shape {
    int radius;
    public double area() {
        return Math.PI * r * r;
    }
    ...
}

public class Square implements Shape {
    int side;
    public double area() {
        return side * side;
    }
    ...
}

All shapes have an area and a perimeter, but they calculate them differently!
In Java, **List** is an interface:

1. `List<String> list = new ArrayList<String>();`
2. `List<String> list = new LinkedList<String>();`

By using the interface on the left instead of the specific class, we allow more general code!