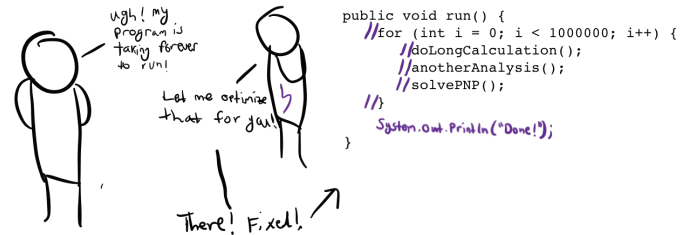


# CSE 143

## Computer Programming II

## Efficiency; Interfaces



### Questions From Last Time

1

- 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

2

#### Style Tips

- Avoid "obvious comments". The following is **BAD**.

```
1 //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 }
5 else {
6     throw new IllegalArgumentException();
7 }
```

- 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

3

#### Testing/Debugging Tips

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

### Oddly Prolific Questions...

4

- 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");
                                         3 System.out.print("l");
                                         4 System.out.print("l");
                                         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."

## hasDuplicate

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

```

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

```

>> hasDuplicate1 average run time is 5254712 ns.
>> hasDuplicate2 average run time is 2384 ns.

```

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:

```

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

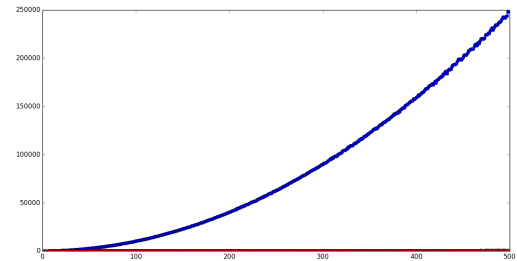
```

>> 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, ..., 1000000, and plot:



## 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).

```

1 statement1; }
2 statement2; } 3
3 statement3; }
4
5 for (int i = 0; i < N; i++) { } N
6     statement4; }
7 }
8
9
10 for (int i = 0; i < N; i++) { } 4N
11     statement5; }
12     statement6; }
13     statement7; }
14     statement8; }
15 }

```

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

$O(1)$	Constant	The number of steps doesn't depend on $n$
$O(n)$	Linear	If you double $n$ , the number of steps <b>doubles</b>
$O(n^2)$	Quadratic	If you double $n$ , the number of steps <b>quadruples</b>
$O(2^n)$	Exponential	The number of steps gets infeasible at $n < 100$

## More Examples

11

```

1 statement1; }
2 statement2; } 3
3 statement3; }
4
5 for (int i = 0; i < N; i++) {
6     statement4;
7     for (int j=0; i < N/2; j++) {
8         statement5;
9     }
10 }
11
12
13 for (int i = 0; i < N; i++) {
14     statement6;
15     statement7;
16     statement8;
17     statement9;
18 }

```

$\left. \begin{array}{l} N/2 \\ 4N \end{array} \right\} N + N(N/2)$   
 $\left. \begin{array}{l} N + N(N/2) \\ 4N \end{array} \right\} 0.5N^2 + 5N + 3$

So, the entire thing is  $O(N^2)$ , because the quadratic term overtakes all the others.

## ArrayList Efficiency

12

add(val)	$O(1)$
add(idx, val)	$O(n)$
get(idx)	$O(1)$
set(idx, val)	$O(1)$
remove(idx)	$O(n)$
size()	$O(1)$

## ArrayList Example

13

What are the time complexities of these functions?

```

1 public static void numbers1(int max) {
2     ArrayList<Integer> list = new ArrayList<Integer>(); //O(1)
3     for (int i = 1; i < max; i++) {
4         list.add(i); //O(1)
5     }
6 }

```

$\left. \begin{array}{l} O(1) \\ O(1) \end{array} \right\} O(n)$

vs.

```

1 public static void numbers2(int max) {
2     ArrayList<Integer> list = new ArrayList<Integer>(); //O(1)
3     for (int i = 1; i < max; i++) {
4         list.add(i); //O(1)
5         list.add(i); //O(1)
6     }
7 }

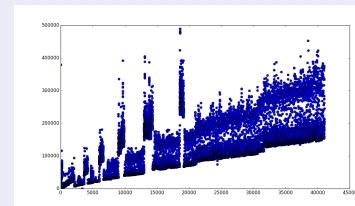
```

$\left. \begin{array}{l} O(1) \\ O(1) \end{array} \right\} O(n)$

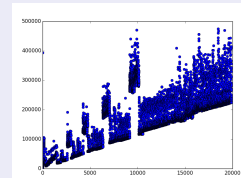
## Investigating Our Answer With Pictures

14

numbers1



numbers2



## Find the Runtime

15

```

1 public boolean is10(int number) {
2     return number == 10;
3 }
4
5 public boolean two10s(int num1, int num2, int num3) {
6     return (is10(num1) && is10(num2) && !is10(num3)) ||
7           (is10(num1) && !is10(num2) && is10(num3)) ||
8           (!is10(num1) && is10(num2) && is10(num3));
9 }
10
11 public void loops(int N) {
12     for (int i = 0; i < N; i++) {
13         for (int j = 0; j < N; j++) {
14             System.out.println(i + " " + j);
15         }
16     }
17 }
18
19 for (int i = 0; i < N; i++) {
20     System.out.println(N - i);
21 }
22 }

```

$\left. \begin{array}{l} O(1) \\ O(1) \end{array} \right\} O(1)$   
 $\left. \begin{array}{l} O(1) \\ O(1) \end{array} \right\} O(1)$   
 $\left. \begin{array}{l} O(n^2) \\ O(n^2) \end{array} \right\} O(n^2)$   
 $\left. \begin{array}{l} O(n^2) \\ O(n) \end{array} \right\} O(n^2)$

## It's the WORST CASE!

16

```

1 public static int has5(int[] array) {
2     for (int i = 0; i < array.length; i++) {
3         System.out.println(array[i]); //O(1)
4         if (array[i] == 5) { //O(1)
5             return true; //O(1)
6         }
7     }
8     return false; //O(1)
9 }

```

$\left. \begin{array}{l} O(1) \\ O(1) \end{array} \right\} O(1)$   
 $\left. \begin{array}{l} O(1) \\ O(n) \end{array} \right\} O(n)$   
 $\left. \begin{array}{l} O(n) \\ O(1) \end{array} \right\} O(n)$

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  $O(n)$ .

```

1 //Inside the ArrayIntList class...
2 /** pre: size >= 1 otherwise throws IllegalStateException */
3 public int max() {
4     if (this.size < 1) {
5         throw new IllegalStateException();
6     }
7
8     int result = this.data[0];
9     for (int i = 1; i < this.size; i++) {
10        if (this.data[i] > result) {
11            result = Math.max(result, this.data[i]);
12        }
13    }
14    return result;
15 }

```

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

Yes! Create a max field in the ArrayIntList class and update it when we add/remove.

```

private int slowMax() { //slowMax is  $\mathcal{O}(n)$ , 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  $\mathcal{O}(n)$ 
    this.size++; // $\mathcal{O}(1)$ 
    this.grow(this.size); // $\mathcal{O}(n)$ 
    this.checkIndex(index); // $\mathcal{O}(1)$ 
    for (int i = this.size - 1; i > index; i--) { // $\mathcal{O}(n)$  (for loop)
        this.data[i] = this.data[i-1]; // $\mathcal{O}(1)$ 
    }

    int oldValue = this.data[index]; // $\mathcal{O}(1)$ 
    this.data[index] = value; // $\mathcal{O}(1)$ 

    if (value > max) { this.max = value; } // $\mathcal{O}(1)$ 
    else if (oldValue == max) { this.max = this.slowMax(); } // $\mathcal{O}(n)$  (slowMax)
}

public void remove(int index) { //remove is  $\mathcal{O}(n)$ 
    this.checkIndex(index); // $\mathcal{O}(1)$ 
    int oldValue = this.data[index]; // $\mathcal{O}(1)$ 
    for (int i = index; i < size - 1; i++) { // $\mathcal{O}(n)$  (for loop)
        this.data[i] = this.data[i+1]; // $\mathcal{O}(1)$ 
    }
    this.size--; // $\mathcal{O}(1)$ 
    if (this.max == oldValue) {
        this.max = this.slowMax(); // $\mathcal{O}(n)$  (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.

Interface

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.

```

1 public interface Shape {
2     public double area();
3     public double perimeter();
4 }
5
6 public class Circle implements Shape {
7     int radius;
8     public double area() {
9         return Math.PI * r * r;
10    }
11    ...
12 }
13
14 public class Square implements Shape {
15     int side;
16     public double area() {
17         return side * side;
18    }
19    ...
20 }

```

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!