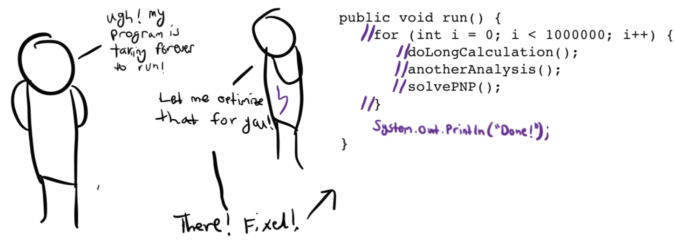
Adam Blank Lecture 6 Spring 2015

# CSE 143

## Computer Programming II



# Efficiency; Interfaces




## Oddly Prolific Questions. . .



- Is most of 143 "style" as opposed to "content"?
- How do TAs judge the "efficiency" of a solution?

## Efficiency



**What does it mean to have an "efficient program"?**

```
System.out.println("hello");
```

vs.

```
System.out.print("h");
System.out.print("e");
System.out.print("l");
System.out.print("l");
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
```
>> 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;
}
```

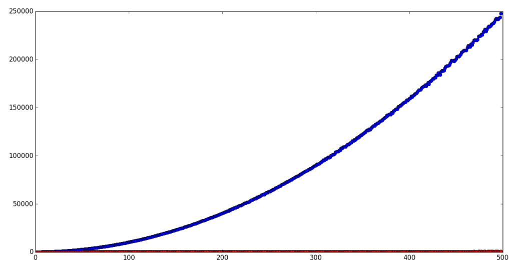
OUTPUT
```
>> hasDuplicate1 average number of steps is 9758172 steps.
>> hasDuplicate2 average number of steps is 170 steps.
```

## Comparing Programs: Plotting



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:




## 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).

```
statement1;
statement2;
statement3;

for (int i = 0; i < N; i++) {
   statement4;
}


for (int i = 0; i < N; i++) {
   statement5;
   statement6;
   statement7;
   statement8;
}
```

Lines 1–3: $3$; lines 5–7: $N$; lines 10–15: $4N$; total: $5N + 3$

## Big-Oh



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 $\mathcal{O}(N)$ ("Big-Oh-of-$N$") which means the number of steps it takes is **linear** in the input.

**Some Common Complexities**

| | | |
|---|---|---|
| $\mathcal{O}(1)$ | Constant | The number of steps doesn't depend on $n$ |
| $\mathcal{O}(n)$ | Linear | If you double $n$, the number of steps **doubles** |
| $\mathcal{O}(n^2)$ | Quadratic | If you double $n$, the number of steps **quadruples** |
| $\mathcal{O}(2^n)$ | Exponential | The number of steps gets infeasible at $n < 100$ |

## More Examples



```
statement1;
statement2;
statement3;

for (int i = 0; i < N; i++) {
   statement4;
   for (int j=0; j < N/2; j++) {
      statement5;
   }
}


for (int i = 0; i < N; i++) {
   statement6;
   statement7;
   statement8;
   statement9;
}
```

Lines 1–3: $3$; inner loop: $N/2$; lines 5–10: $N + N(N/2)$; lines 13–18: $4N$; total: $0.5N^2 + 5N + 3$

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

## `ArrayList` Efficiency



| | |
|---|---|
| `add(`**val**`)` | $\mathcal{O}(1)$ |
| `add(`**idx**, **val**`)` | $\mathcal{O}(n)$ |
| `get(`**idx**`)` | $\mathcal{O}(1)$ |
| `set(`**idx**, **val**`)` | $\mathcal{O}(1)$ |
| `remove(`**idx**`)` | $\mathcal{O}(n)$ |
| `size()` | $\mathcal{O}(1)$ |

## `ArrayList` Example



What are the time complexities of these functions?

```
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)
   }
}
```

Loop: $\mathcal{O}(n)$; whole function: $\mathcal{O}(n)$

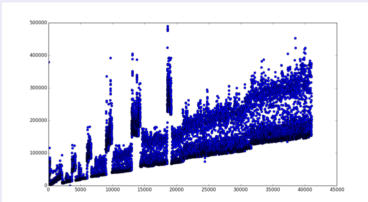
vs.

```
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)
   }
}
```

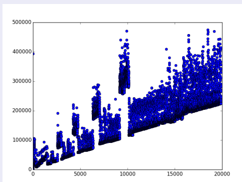
Loop body: $\mathcal{O}(1)$; loop: $\mathcal{O}(n)$; whole function: $\mathcal{O}(n)$

## Investigating Our Answer With Pictures 

numbers1




numbers2




## Find the Runtime 

```java
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);
      }
   }


   for (int i = 0; i < N; i++) {
      System.out.println(N - i);
   }
}
```

is10: $\mathcal{O}(1)$

two10s: $\mathcal{O}(1)$

nested loops: $\mathcal{O}(n^2)$; single loop: $\mathcal{O}(n)$; loops: $\mathcal{O}(n^2)$

## 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)
}
```

if block: $\mathcal{O}(1)$; for loop: $\mathcal{O}(n)$; has5: $\mathcal{O}(n)$

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 $\mathcal{O}(n)$.