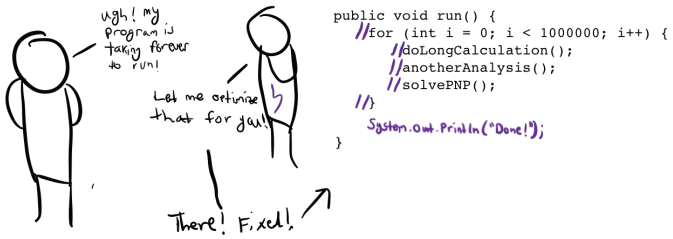
CSE 143X

Accelerated Computer Programming I/II

Efficiency



Efficiency 1

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."

Comparing Programs: Timing 2

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.
```

Comparing Programs: # Of Steps 3

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;
}
```

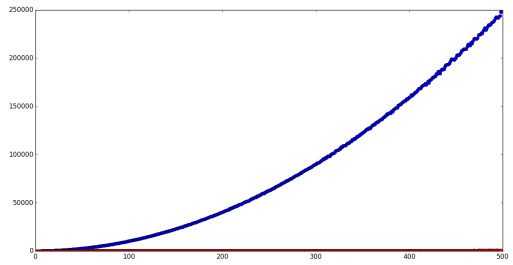
OUTPUT

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

Comparing Programs: Plotting 4

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).

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

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


```
1  statement1;
2  statement2;
3  statement3;
4
5  for (int i = 0; i < N; i++) {
6     statement4;
7     for (int j=0; j < 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 }
```

$0.5N^2 + 5N + 3$

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?

```
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  }
```

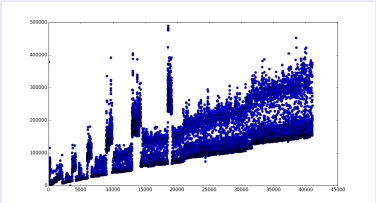
$\mathcal{O}(n)$ $\mathcal{O}(n)$

vs.

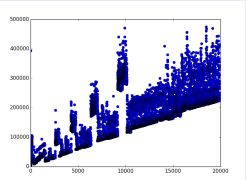
```
1  public static void numbers2(int max) {
2     ArrayList<Integer> list = new ArrayList<Integer>(); //O(1)
3     for (int i = 1; i < max; i++) {
4        list.add(i); //O(1)
5        list.add(i); //O(1)
6     }
7  }
```

Investigating Our Answer With Pictures

numbers1



numbers2



Find the Runtime


```
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 }
```

$\mathcal{O}(1)$ $\mathcal{O}(1)$ $\mathcal{O}(n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(1)$

It's the WORST CASE!


```
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  }
```

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 $\mathcal{O}(n)$.