CSE 143

Computer Programming II
Efficiency; Interfaces

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

ugh! my program is taking forever to run.

Let me optimize that for you!

There! Fixed!
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.  
2 1 `System.out.print("h");`
3 2 `System.out.print("e");`
4 3 `System.out.print("l");`
5 4 `System.out.print("l");`
6 5 `System.out.println("o");`

<table>
<thead>
<tr>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt; left average run time is 1000 ns.</td>
</tr>
<tr>
<td>&gt;&gt; right average run time is 5000 ns.</td>
</tr>
</tbody>
</table>

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.

```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**

```shell
>> 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:
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    }
```

- `statement1` takes one step.
- `statement2` takes three steps.
- The loop in line 5 takes `N` steps.
- The loop in line 10 executes `4N` times.
- The total runtime is `5N + 3`.

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>Complexity</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
<td>The number of steps doesn’t depend on $n$</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
<td>If you double $n$, the number of steps <strong>doubles</strong></td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
<td>If you double $n$, the number of steps <strong>quadruples</strong></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>add(val)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(idx, val)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get(idx)</td>
<td>O(1)</td>
</tr>
<tr>
<td>set(idx, val)</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove(idx)</td>
<td>O(n)</td>
</tr>
<tr>
<td>size()</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
What are the time complexities of these functions?

```java
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 }
```

vs.

```java
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
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);
    }
}
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 \( O(n) \).