CSE 143

Computer Programming II
Efficiency

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

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**
```
>> 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).

```
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 }
```

- \( \text{3 steps} \) vs. \( \text{5N + 3 steps} \)
- \( \text{N steps} \) vs. \( \text{4N steps} \)
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.

<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>Time 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?

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 }

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 }

O(n)
Investigating Our Answer With Pictures

(numbers1)

(numbers2)
1 public boolean is10(int number) {
2     return number == 10;
3 }

4 public boolean two10s(int num1, int num2, int num3) {
5     return (is10(num1) && is10(num2) && !is10(num3)) ||
6            (is10(num1) && !is10(num2) && is10(num3)) ||
7            (!is10(num1) && is10(num2) && is10(num3));
8 }

9 public void loops(int N) {
10     for (int i = 0; i < N; i++) {
11         for (int j = 0; j < N; j++) {
12             System.out.println(i + " " + j);
13         }
14     }
15 }
16 }
17 }
18 }
19 }
20 System.out.println(N - i);
21 }
22 }
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 $O(n)$. 