Oddly Prolific Questions...

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

Efficiency; Interfaces

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.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;
}
```

```java
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;
}
```

```java
public int stepsHasDuplicate2(int[] array) {
    int steps = 0;
    for (int i=0; i < array.length - 1; i++) {
        if (array[i] == array[i+1]) {
            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:

![Graph showing growth rate of algorithmic complexity](image)

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 $O(N)$ ("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>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
</tr>
</tbody>
</table>

Comparing Programs: Analytically

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 for (int i = 0; i < N; i++) {
5      statement4;
6      statement5;
7      statement6;
8      statement7;
9      statement8;
10     }
11 }
```

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

$N$ $5N + 3$

$N/2$ $N + N(N/2)$ $0.5N^2 + 5N + 3$

$4N$ $4N$ $4N$

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

ArrayList Efficiency

<table>
<thead>
<tr>
<th>Operation</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>

**More Examples**

```
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 }
```

```
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 }
```

$O(n)$ $O(n)$

ArrayList Example

What are the time complexities of these functions?
Find the Runtime

```java
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.   for (int i = 0; i < N; i++) {
17.     System.out.println(N - i);
18.   }
19. }
20.```

O(n^2)

O(n)

O(n^2)

O(1)

O(1)

O(1)

O(1)

O(n^2)

O(n)

It's the WORST CASE!

```java
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.         System.out.println("== 5"); //O(1)
   6.         return true; //O(1)
   7.     }
   8.     }
   9.     return false; //O(1)
}
```

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).