Efficiency

What does it mean to have an “efficient program”?

1. System.out.println("hello");
2. System.out.println("e");
3. System.out.println("l");
4. System.out.println("o");

SYSTEM OUT

>> 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.

1. public boolean hasDuplicate(int[] array) {
   2.     for (int i = 0; i < array.length; i++) {
   3.         for (int j = 0; j < array.length; j++) {
   4.             if (i != j && array[i] == array[j]) {
   5.                 return true;
   6.             }
   7.         }
   8.     }
   9.     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:

1. public int stepsHasDuplicate(int[] array) {
   2.     int steps = 0;
   3.     for (int i = 0; i < array.length; i++) {
   4.         for (int j = 0; j < array.length; j++) {
   5.             steps++; // The if statement is a step
   6.         }
   7.     }
   8.     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 output).

```java
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        statement6;
10    }
11    statement7;
12    statement8;
13 }
```

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

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>

ArrayList Efficiency

- `add(val)` $O(1)$
- `add(idx, val)` $O(n)$
- `get(idx)` $O(1)$
- `set(idx, val)` $O(1)$
- `remove(idx)` $O(n)$
- `size()` $O(1)$

Investigating Our Answer With Pictures

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

### It's the WORST CASE!

```
1 public static int has5(int[] array) {
2    for (int i = 0; i < array.length; i++) {
3        if (array[i] == 5) {
4            return true;
5        }
6    }
7    return false;
8 }
9
10 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).
```