CSE 143

Computer Programming II

Efficiency; Interfaces


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

- Does a constructor have to use all the fields specified in a class?
  Nope. It depends on what you're trying to do.
- For class constants, why write "public"?
  We don't technically have to. It's just considered good style.
- `public static final int` vs. `private static final int`?
  If it's private, clients can't use it.
- Vim or Emacs?
  Vim is the way and the light.
- Waffles or Pancakes?
  Pancakes I guess?
- Is Euler self-aware?
  I'm not sure; I'll have to ask him.
- (I'll continue the game of Tic-Tac-Toe next time.)
- (Also, I'll add this time's pictures then too.)

Some Quick Debugging Tips

Style Tips

- Avoid "obvious comments". The following is **BAD**.

```
//BAD BAD BAD BAD BAD BAD
int count = 0; // Initializes a count of values
```

- Throw exceptions as early as possible in methods.

```
//BAD BAD BAD BAD BAD BAD
if (size > 0) {
   //Do stuff
}
else {
   throw new IllegalArgumentException();
}
```

- Avoid using constants that aren't clear. (Especially if there is a clearer way to write them.)

```
//BAD BAD BAD BAD BAD BAD
public static final int LENGTH_OF_JAVA = 4;
//BETTER
public static final int LENGTH_OF_JAVA = "JAVA".length();
```

- Don't overcomment: a comment on every line is unreadable.

Some Quick Testing/Debugging Tips

Testing/Debugging Tips

- Check **EDGE CASES** (`null`, 0, `capacity`, etc.)
- Test running multiple methods one after another (`list.add(5); list.add(5); list.remove(0); list.add(5);...`)

- 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"?

```
System.out.println("hello");
```

vs.

```
System.out.print("h");
System.out.print("e");
System.out.print("l");
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."

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.
```

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:

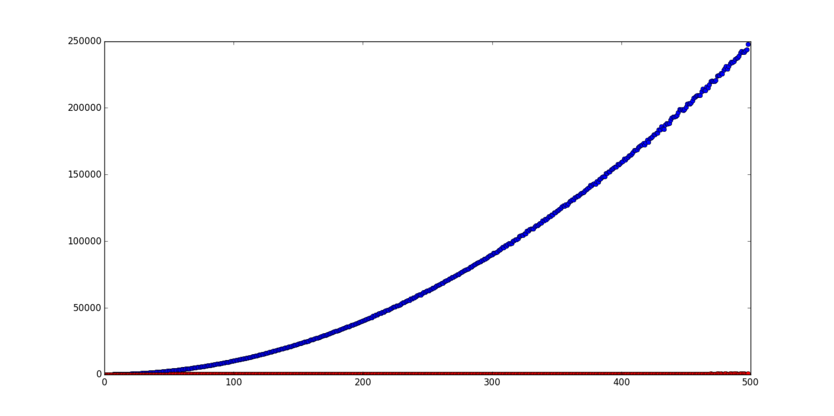
```
public int stepsHasDuplicate1(int[] array) {
   int steps = 0;
   for (int i=0; i < array.length; i++) {
      for (int j=0; j < array.length; j++) {
         steps++; // The if statement is a step
         if (i != j && array[i] == array[j]) {
            return steps;
         }
      }
   }
   return steps;
}
```

OUTPUT

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

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).

```
statement1;
statement2;
statement3;

for (int i = 0; i < N; i++) {
   statement4;
}


for (int i = 0; i < N; i++) {
   statement5;
   statement6;
   statement7;
   statement8;
}
```

statement1–statement3: 3

First loop: N

Second loop: $4N$

Total: $5N + 3$

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 $\mathcal{O}(N)$ ("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

```
statement1;
statement2;
statement3;

for (int i = 0; i < N; i++) {
   statement4;
   for (int j=0; i < N/2; j++) {
      statement5;
   }
}


for (int i = 0; i < N; i++) {
   statement6;
   statement7;
   statement8;
   statement9;
}
```

3

$N/2$

$N + N(N/2)$

$4N$

$0.5N^2 + 5N + 3$

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

`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?

```java
public static void numbers1(int max) {
   ArrayList<Integer> list = new ArrayList<Integer>(); //O(1)
   for (int i = 1; i < max; i++) {
      list.add(i); //O(1)
   }
}
```

Lines 3–5 (loop): $\mathcal{O}(n)$

Whole function: $\mathcal{O}(n)$

vs.

```java
public static void numbers2(int max) {
   ArrayList<Integer> list = new ArrayList<Integer>(); //O(1)
   for (int i = 1; i < max; i++) {
      list.add(i); //O(1)
      list.add(i); //O(1)
   }
}
```

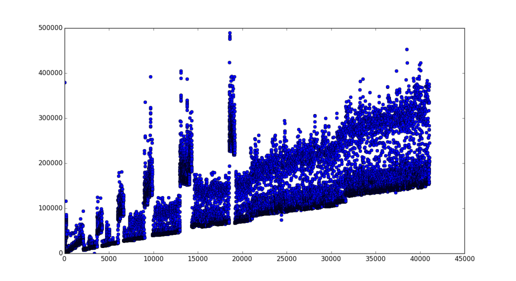
Lines 4–5: $\mathcal{O}(1)$

Lines 3–6 (loop): $\mathcal{O}(n)$

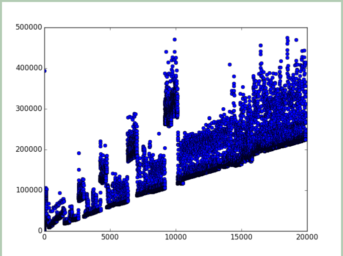
Whole function: $\mathcal{O}(n)$

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);
   }
}
```

is10: $\mathcal{O}(1)$

two10s: $\mathcal{O}(1)$

Nested loops: $\mathcal{O}(n^2)$

Single loop: $\mathcal{O}(n)$

loops: $\mathcal{O}(n^2)$

It's the WORST CASE!

```
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)
}
```

Lines 4–5 (if / return true): $\mathcal{O}(1)$; the for loop (lines 2–7): $\mathcal{O}(n)$; the whole method: $\mathcal{O}(n)$

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 $\mathcal{O}(n)$.

```java
//Inside the ArrayIntList class...
/** pre: size >= 1 otherwise throws IllegalStateException */
public int max() {
   if (this.size < 1) {
      throw new IllegalStateException();
   }

   int result = this.data[0];
   for (int i = 1; i < this.size; i++) {
      if (this.data[i] > result) {
         result = Math.max(result, this.data[i]);
      }
   }
   return result;
}
```

This code sucks! It's $\mathcal{O}(n)$. Can we do it in $\mathcal{O}(1)$?

Yes! Create a max field in the ArrayIntList class and update it when we add/remove.

```
private int slowMax() { //slowMax is O(n), because of the for loop.
   int result = this.data[0];
   for (int i = 1; i < this.size; i++) {
      if (this.data[i] > result) {
         result = Math.max(result, this.data[i]);
      }
   }
   return result;
}
public void add(int index, int value) { //add is O(n)
   this.size++;                                              //O(1)
   this.grow(this.size);                                     //O(n)
   this.checkIndex(index);                                   //O(1)
   for (int i = this.size - 1; i > index; i--) {             //O(n) (for loop)
      this.data[i] = this.data[i-1]; //O(1)
   }

   int oldValue = this.data[index];                          //O(1)
   this.data[index] = value;                                 //O(1)

   if (value > max) { this.max = value; }                    //O(1)
   else if (oldValue == max) { this.max = this.slowMax(); } //O(n) (slowMax)
}
public void remove(int index) { //remove is O(n)
   this.checkIndex(index);                                   //O(1)
   int oldValue = this.data[index];                          //O(1)
   for (int i = index; i < size - 1; i++) {                  //O(n) (for loop)
      this.data[i] = this.data[i+1]; //O(1)
   }
   this.size--;                                              //O(1)
   if (this.max == oldValue) {
      this.max = this.slowMax();                             //O(n) (slowMax)
   }
}
```

Locking a Safe

What are some different locking mechanisms for safes?

- Door Lock
- Combination Lock
- Padlock
- Digital Lock
- etc.

Note the following:

- All mechanisms have a way to "lock" and "unlock" the safe.
- Each mechanism works completely differently, is made up of different parts, and is used differently.

Interfaces

Interface

An **interface** specifies a group of behaviors and gives them a name. Classes can choose to **implement** interfaces which require them to implement all of the methods in the interface.

The idea is the same as with the safe: there might be multiple different ways to implement the interface.

```
public interface Shape {
   public double area();
   public double perimeter();
}

public class Circle implements Shape {
   int radius;
   public double area() {
      return Math.PI * r * r;
   }
   ...
}

public class Square implements Shape {
   int side;
   public double area() {
      return side * side;
   }
   ...
}
```

All shapes have an area and a perimeter, but they calculate them differently!

Lists

In Java, List is an interface:

```
List<String> list = new ArrayList<String>();
List<String> list = new LinkedList<String>();
```

By using the interface on the left instead of the specific class, we allow more general code!