



Lecture 3: Code Analysis

CSE 373 Data Structures and
Algorithms

Administrivia

Three forms are going out later today:

- Partner form for Project 1 (due Monday night)
- Course background survey (help us optimize for your background/goals)
- Canvas quiz
 - Make sure you understand important details of the syllabus
 - Get extra credit!

Lecture 2 slides are updated on webpage.

- Usually will do this silently, if bugs were pointed out during lecture.

Testing Wrap Up

Computers don't make mistakes- people do!

"I'm almost done, I just need to make sure it works"

– Naive 14Xers

Software Test: a separate piece of code that exercises the code you are assessing by providing input to your code and finishes with an assertion of what the result should be.

1. Isolate - break your code into small modules
2. Build in increments - Make a plan from simplest to most complex cases
3. Test as you go - As your code grows, so should your tests

Testing Strategies

You can't test everything

- Break inputs into categories
- What are the most important pieces of code?

Test behavior in combination

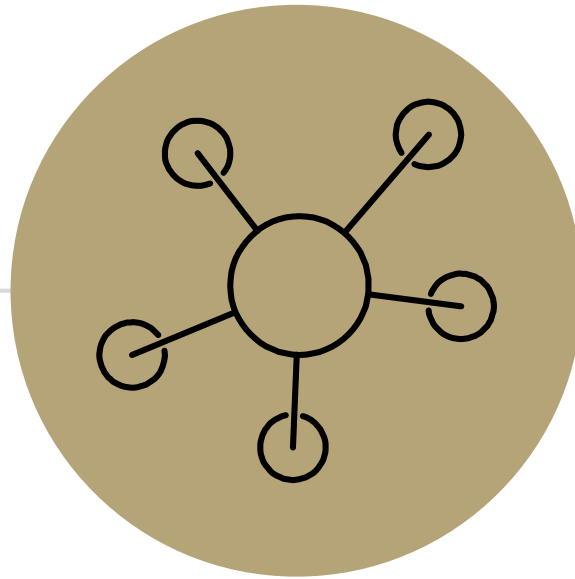
- Call multiple methods one after the other
- Call the same method multiple times

Trust no one!

- How can the user mess up?

If you messed up, someone else might

- Test the complex logic





Algorithm Analysis

Code Analysis

How do we compare two pieces of code?

Mathematically:

- Time needed to run 
- Memory used 
- Number of network calls
- Amount of data saved to disk

Tools we use can be applied to any aspect of your code you can measure.

With Words:

- Specialized vs generalized
- Code reusability
- Security

Comparing Algorithms with Mathematical Models

Consider overall trends as input/data set gets bigger

- Computers are fast – anything you do on your 5 item dataset is going to finish in the blink of an eye.
- Large inputs differentiate.

Identify trends without investing in testing

- Estimate how big of a dataset you can handle

asymptotic analysis – the process of mathematically representing runtime of an algorithm as a function of the number/size of inputs as the input grows (arbitrarily large)



Code Modeling

Disclaimer

This topic has lots of details/subtle relationships between concepts.

I'm going to try to introduce things one at a time (all at once can be overwhelming).

"We'll see that later" might be the answer to a lot of questions.

Code Modeling

code modeling – the process of mathematically representing how many operations a piece of code will run in relation to the number of inputs n

What counts as an “operation”?

Assume all basic operations run in equivalent time

Basic operations

- Adding ints or doubles
- Variable assignment
- Variable update
- Return statement
- Accessing array index or object field

Consecutive statements

- Sum time of each statement

Function calls

- Count runtime of function body
- Remember that `new` calls a function!

Conditionals

- Time of test + appropriate branch
 - We'll talk about which branch to analyze when we get to cases.

Loops

- Number of iterations of loop body x runtime of loop body

Modeling Case Study

Goal: return 'true' if a sorted array of ints contains duplicates

Solution 1: compare each pair of elements

```
public boolean hasDuplicate1(int[] array) {
    boolean found = false;
    int failedChecks = 0;
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array.length; j++) {
            if (i != j && array[i] == array[j])
                found = true;
            else
                failedChecks++
        }
    }
    return found;
}
```

Solution 2: compare each consecutive pair of elements

```
public boolean hasDuplicate2(int[] array) {
    boolean found = false;
    int failedChecks = 0;
    for (int i = 0; i < array.length - 1; i++) {
        if (array[i] == array[i + 1])
            found = true;
        else
            failedChecks++;
    }
    return found;
}
```

Modeling Case Study: Solution 2

Goal: produce mathematical function representing runtime $f(n)$ where n is the size of the array

Solution 2: compare each consecutive pair of elements

```
public boolean hasDuplicate2(int[] array) {
    boolean found = false; +1
    int failedChecks = 0; +1
    for(int i = 0; i < array.length - 1; i++) +2
        if (array[i] == array[i + 1])
            found = true; +1
        else
            failedChecks++; +1
    }
    return found; +1
}
```

loop = $(n - 1)(\text{body})$
Loop body
{ 5 (if) + 2 (loop checks) = 7
} If statement
4+1=5
+1 either branch

$$f(n) = 7(n - 1) + 4$$

linear $\rightarrow O(n)$

Approach

- \rightarrow start with basic operations, work inside out for control structures
- Each basic operation = +1
- Conditionals = test operations + appropriate branch (today branches equivalent)
- Loop = #iterations * (operations in loop body)

Finding a Big-O

We have an expression for $f(n)$.

How do we get the $O()$?

1. Make $f(n)$ "look nice"
2. Find the "dominating term" and delete all others.
 - The "dominating" term is the one that is largest as n gets bigger. In this class, often the largest power of n .
3. Remove any constant factors.
4. Write the final big-O

$$f(n) = \underline{7(n-1)} + 4$$

$$f(n) = 7(n-1) + 4 = 7n - 7 + 4 = \underline{7n - 3}$$

$$f(n) = 7n - 3 \approx \underline{7n}$$

$$f(n) \approx 7n \approx n$$

$$f(n) \text{ is } O(n)$$

Wait, what?

Why did we just throw out all of that information?

Big-O is the “significant digits” of computer science.

We care about what happens when n gets bigger

- All code is “fast enough” for small n in practice

For large enough n the dominant term decides how big the function is.

Why get rid of constants – we were counting “basic operations”

There is not a strong correlation between the number of basic operations and the time code actually takes to run.

Why aren't they significant?

```
public static void method1(int[] input)
{
    int n = input.length;
    input[n-1] = input[3] + input[4];
    input[0] += input[1];
}
```

public static void method1(int[]); Code:

0: aload_0	10: aload_0	20: iconst_1
1: arraylength	11: iconst_4	21: iaload
2: istore_1	12: iaload	22: iadd
3: aload_0	13: iadd	23: iastore
4: iload_1	14: iastore	24: return
5: iconst_1	15: aload_0	
6: isub	16: iconst_0	
7: aload_0	17: dup2	
8: iconst_3	18: iaload	
9: iaload	19: aload_0	

```
public static void method2(int[] input)
{
    int five = 5;
    input[five] = input[five] + 1;
    input[five]--;
}
```

public static void method2(int[]); Code:

0: iconst_5	10: aload_0
1: istore_1	11: iload_1
2: aload_0	12: dup2
3: iload_1	13: iaload
4: aload_0	14: iconst_1
5: iload_1	15: isub
6: iaload	16: iastore
7: iconst_1	17: return
8: iadd	
9: iastore	

Why aren't they significant?

It goes deeper.

The Java bytecode is converted (compiled) into your own machine's assembly code

- Might change the number of lines again.

The number of lines still isn't a perfect reflection of time taken by your laptop.

The amount of time it takes to look up a value in memory is **wildly** variable

Recently used values are probably "cached" and will have a quick lookup

If a value hasn't been used in a long time, might have to wait for main memory, which takes **thousands** of times as long.

Modern computers do lots of crazy things to speed up code.

- "pipelining" (execute parts of multiple instructions simultaneously)
- "branch prediction" (guess whether you're about to go down the if or else branch before it actually gets there)

Code Modelling

We can't accurately model the constant factors just by staring at the code.

And the lower-order terms matter even less than the constant factors.

So we just ignore them for the big-O.

If we ask for a model, we won't care about whether you count 4 operations per loop or 5 (or 10 or 1 or 28).

We want to be able to see your numbers weren't guesses and that you get the right big-O.

This does not mean you shouldn't care about constant factors ever – they are important in real code!

- Our theoretical tools aren't precise enough to analyze them well.

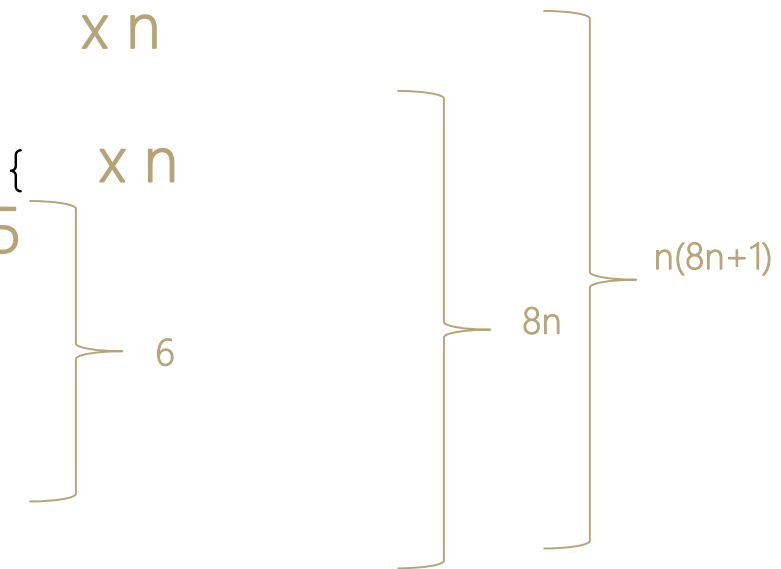
Modeling Case Study: Solution 1

Solution 1: compare each consecutive pair of elements

```
public boolean hasDuplicate1(int[] array) {
    boolean found = false;
    int failedChecks = 0;
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array.length; j++) {
            if (i != j && array[i] == array[j])
                found = true;
            else
                failedChecks++;
        }
    }
    return found;
}
```

$$f(n) = n(8n + 1) + 4$$

quadratic $\rightarrow O(n^2)$



Approach

- > *start with basic operations, work inside out for control structures*
- Each basic operation = +1
- Conditionals = test operations + appropriate branch (today branches equivalent)
- Loop = #iterations * (operations in loop body)

Your turn!

Write the specific mathematical code model for the following code and indicate the big-O runtime in terms of k .

```

public void foobar (int k) {
    int j = 0; +1
    while (j < k) { +k/5 (body)
        for (int i = 0; i < k; i++) { +k(body)
            System.out.println("Hello world"); +1
        }
        j = j + 5; +2
    }
}

```

Handwritten annotations: A large red bracket on the left side of the code. A red circle around the line `j = j + 5; +2`. A red $\frac{k}{5}$ next to the while loop. A red $3k$ next to the for loop. A red $\frac{k}{5}$ above the for loop.

$$f(k) = \frac{3k(k+2)}{5}$$

quadratic $\rightarrow O(k^2)$

Approach

- > *start with basic operations, work inside out for control structures*
- Each basic operation = +1
- Conditionals = test operations + appropriate branch (today branches equivalent)
- Loop = #iterations * (operations in loop body)

More Practice

Let `myLL` be a linked list (like we saw in lecture 1) with n nodes.

Suppose we're a client class. Let's try to print every element of the list.

Assume `get(i)` takes i steps

```
for(int i=0; i<myLL.size(); i++){
    System.out.println(myLL.get(i));
}
```

The number of operations changes each time through the loop.

Summations to the rescue!

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \rightarrow O(n^2)$$

Summations review and a bunch of identities:

<https://courses.cs.washington.edu/courses/cse373/19su/resources/>



Iterators

Traversing Data

We could get through the data much more efficiently in the Linked List class itself.

```
Node curr = this.front;
while (curr != null) {
    System.out.println(curr.data);
    curr = curr.next;
}
```

What if the client wants to do something other than just print?

We should provide giving each element in order as a service to client classes.

```
for (T item : list) { ← Iterator!
    System.out.println(item);
}
```

Review: Iterators

iterator: a Java interface that dictates how a collection of data should be traversed. Can only move in the forward direction and in a single pass.

Iterator Interface

behavior

hasNext() – true if elements remain

next() – returns next element

supported operations:

hasNext() – returns true if the iteration has more elements yet to be examined

next() – returns the next element in the iteration and moves the iterator forward to next item

```
ArrayList<Integer> list = new ArrayList<Integer>();  
//fill up list
```

```
Iterator itr = list.iterator();  
while (itr.hasNext()) {  
    int item = itr.next();  
}
```

```
ArrayList<Integer> list = new ArrayList<Integer>();  
//fill up list
```

```
for (int i : list) {  
    int item = i;  
}
```

Implementing an Iterator

Usually: you'll have a private class for the iterator object.

That iterator class will have a class variable to remember where you are.

`hasNext ()` – check if there's something left by examining the class variable.

`next ()` – return the current thing and update the class variable.

You have a choice:

- Variable might point to the thing you just processed
- Or the next thing that would be returned.

Both will work, one might be easier to think about/code up in some instances than others.

Punchline: Iterators make your client's code more efficient (which is what they care about)