Lecture 1: Introduction and a Review/Enrichment of CS 101

Introduction

Class Information

The title of this course is Data Structures, but in reality the name only tells half of the story. Part of the course covers the standard data structures theory along with an introduction to algorithms:

1. Basic Runtime and Space Asymptotic Analysis
2. Stacks, Queues and Deques
3. Linked Lists
4. Recursion
5. Trees, Tree Traversals, Balanced Trees
6. Binary Heaps and Priority Queues
7. Sorting and Searching Algorithms
8. Maps (hash and tree-based) and Sets
9. Graphs (time permitting)
10. Applying the above structures to solve problems

Although the above topics are important, data structures is also largely a Java programming course. Even more importantly, it is the second Java programming course in the major. This positions you (the student) to really improve your coding skills. Throughout the semester you will spend a lot of time programming. As such, part of the lectures will be devoted to improving your knowledge of the Java programming language. In class we will first quickly review and enrich what was covered last semester (in your Introduction to CS in Java course). For the remainder of the semester we will learn how to properly design, implement, and test our implementations in Java, and cover whatever language features are required. These include:

1. Using object-oriented techniques to design reusable data structures
2. Generics, Iterators, Exceptions
3. Using the Java API data structure implementations

4. Writing tests

We probably won’t have time to cover the functional programming features added in Java 8.

Class Logistics

Most of the grade will come from the homeworks and labs. There will be 1-2 written exams that will cover the material from class, and I will provide materials to help you prepare. Most if not all of the homeworks will be programming assignments. My lecture style is very interactive with a lot of student collaboration. In class computer usage will be restricted to the labs, which will take 45-75 minutes of each week. The course will be based primarily on the lecture notes. The listed textbook can be used as a learning aid to supplement the lectures.

Review/Enrichment of CS 101

Initial Exercises

1. FizzBuzz, a famous interview question: Write a Java program that prints out the numbers from 1 to 100 but, for multiples of 3 instead print Fizz, for multiples of 5 instead print Buzz, and for multiples of 3 and 5 instead print FizzBuzz.

2. Write a function factorial that takes an int as argument and outputs an int that is the factorial of the input. As a reminder, the factorial of 5 is \(5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1\) and in general

\[ n! = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1.\]

3. Write a function isPermutation that takes an array of ints and returns a boolean. Let \(n\) denote the length of the array. Your function should return true if each number between 0 and \(n - 1\), inclusive, occurs exactly once in the array. Otherwise return false. [In other words, we are checking if the array contains a reordering of the numbers 0,1,\ldots, n-1.]

For example,

\{0,1,2,4,3\} => True
\{2,3,1,0\} => True
\{0,1,2,4\} => False
\{-9\} => False
\{1,1,1,1\} => False
Solutions

1. This exercise let’s us warm up our Java skills. Here are two correct implementations:

   FizzBuzz.java
   ```java
   public class FizzBuzz
   {
      public static void main(String[] args)
      {
         for (int i = 1; i <= 100; ++i)
         {
            if (i % 15 == 0) System.out.println("FizzBuzz");
            else if (i % 3 == 0) System.out.println("Fizz");
            else if (i % 5 == 0) System.out.println("Buzz");
            else System.out.println(i);
         }
      }
   }
   
   FizzBuzz2.java
   ```
   ```java
   public class FizzBuzz2
   {
      public static void main(String[] args)
      {
         for (int i = 1; i <= 100; ++i)
         {
            if (i % 3 == 0) System.out.print("Fizz");
            if (i % 5 == 0) System.out.print("Buzz");
            if (i % 3 != 0 && i % 5 != 0) System.out.print(i);
            System.out.println();
         }
      }
   }
   
   Here is a typical incorrect solution:

   FizzBuzzWrong.java
   ```
   ```java
   public class FizzBuzzWrong
   {
      public static void main(String[] args)
      {
         for (int i = 1; i <= 100; ++i)
         {
            if (i % 3 == 0) System.out.println("Fizz");
            else if (i % 5 == 0) System.out.println("Buzz");
            else if (i % 15 == 0) System.out.println("FizzBuzz"); //Never happens
         }
      }
   }
   ```
else System.out.println(i);
}
}

Note that FizzBuzz is never printed in this one.

2. In this problem we remember how to write functions in Java, and think about the
limits of our data types. Below is a correct implementation (we included a main for
testing):

```java
public class Factorial {
    public static int factorial(int n) {
        int v = 1;
        for (int i = 2; i <= n; ++i) v *= i;
        return v;
    }

    public static void main(String[] args) {
        System.out.printf("factorial(%d) = %d
",3,factorial(3));
        System.out.printf("factorial(%d) = %d
",10,factorial(10));
        System.out.printf("factorial(%d) = %d
",12,factorial(12));
        System.out.printf("factorial(%d) = %d
",13,factorial(13));
        System.out.printf("factorial(%d) = %d
",50,factorial(50));
    }
}
```

Here is the output:

```
factorial(3) = 6
factorial(10) = 3628800
factorial(12) = 479001600
factorial(13) = 1932053504
factorial(50) = 0
```

It is clear from the above output that our function doesn’t work for an input of 50. It
also doesn’t work for 13. To see why, we must understand what an int is. The primitive
integral data types in Java are byte, char, short, int, and long. The most used of these
is int, and it uses 32 bits (or 4 bytes). Each bit takes the value 0 or 1, so there are \(2^{32}\)
possible different values that an int can take. By standard convention, an int uses half
of the values for non-negative numbers \((0, \ldots, 2^{31} - 1)\), and half for negative numbers
\((-2^{31}, \ldots, -1)\). If we recall that \(2^{10} = 1024 \approx 1000\) we see that

\[
2^{31} = 2 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10} \approx 2 \cdot 1000 \cdot 1000 \cdot 1000 = 2,000,000,000.
\]
Thus $2^{31}$ is a bit larger than 2 billion. As a rule of thumb, an integer can hold all 9-digit numbers. The value $13!$ is the first factorial that is larger than $2^{31}$ and thus we get an incorrect value. To get the correct value of $13! = 6227020800$ we could use the long data type which uses 64-bits and goes up to $2^{63} - 1$. The rule of thumb is that a long can represent any 18-digit number. As an aside, let’s try to understand why we get zero for $\text{factorial}(50)$. The answer is that when we add or multiply non-negative integers, the resulting int will always be the lowest order 32-bits of the result. Since $2^{32}$ divides 50! the lowest order 32-bits are all zero.

3. Our final question on isPermutation will allow us to think more about functions, and will get us started thinking about runtime analysis. Below we have 3 correct solutions:

```java
public class IsPermutation
{
    public static boolean isPermutation(int[] arr)
    {
        int n = arr.length;
        int[] counts = new int[n];
        for (int i = 0; i < n; ++i)
        {
            int v = arr[i];
            if (v < 0 || v > n-1) return false;
            counts[v]++;
        }
        for (int i = 0; i < n; ++i)
        {
            if (counts[i] != 1) return false;
        }
        return true;
    }
}
```

A few comments. First note that both for loops use the variable $i$. This is legal since the scope of each variable is their respective for loops. In other words, once each for loop ends the variable $i$ ceases to exist. Here is another solution that uses sorting:

```java
import java.util.Arrays;
public class IsPermutation2
{
    public static boolean isPermutation(int[] arr)
    {
        //arr = Arrays.copyOf(arr, arr.length);
        Arrays.sort(arr);
        for (int i = 0; i < arr.length; ++i)
        {
            if (arr[i] != i) return false;
        }
        return true;
    }
}
```
One thing to notice about this implementation is that input array is modified in the function which may be undesirable. We can avoid this by instead doing

```java
    arr = Arrays.copyOf(arr, arr.length);
    Arrays.sort(arr);
    // rest of code
```

Make sure you understand why this newly modified code is correct. Finally, let’s look at a third implementation:

```java
public class IsPermutation3 {
    public static int countOcc(int[] arr, int val) {
        int count = 0;
        for (int i = 0; i < arr.length; ++i) {
            if (arr[i] == val) count++;
        }
        return count;
    }

    public static boolean isPermutation(int[] arr) {
        for (int i = 0; i < arr.length; ++i)
            if (countOcc(arr, i) != 1) return false;
        return true;
    }

    public static void main(String[] args) {
        int[] arr = {5,4,3,2,1,0};
        System.out.println(isPermutation(arr));
        arr[0] = 1;
        System.out.println(isPermutation(arr));
        arr[0] = 9;
        System.out.println(isPermutation(arr));
    }
}
```

Here we included main to illustrate the following point. The function main has a variable named arr. The functions countOcc and isPermutation both have parameters
called arr. How can all of these coexist without conflicting? To understand how this works we will review the notion of scope (also called name visibility).

Each variable has a name that is used to access it. For instance, when you write `int v = 4;` you are creating a variable named v and giving it the value 4. The scope of a variable is where that name can be used once it is declared. All of the local variables (variables declared in functions) and parameters of a function can only be accessed in the function they were declared in. Thus there is no name conflict between the various variables named arr in IsPermutation3 since they live in different scopes.

**Runtime and Space Utilization**

There are many criteria we can use to evaluate a piece of code. These include:

1. Correctness
2. Memory Usage
3. Running Time
4. Reusability/Fragility: measure of how hard it is to add/change the functionality
5. Quality of Tests

For now we will focus on memory usage and running time. Memory usage describes the total amount of memory used by all of our variables, and all of the objects we refer to. In other words, all of the space we use to store our values. Running time refers to the number of operations/instructions we perform when solving a particular task. In both cases, a precise analysis can be very difficult to perform. For this reason we will use a rougher measure that ignores constant factors. Instead of going into an abstract definition, let’s try to evaluate our 3 implementations of isPermutation above.

1. We say the first version of isPermutation requires $\Theta(n)$ steps. The $\Theta(n)$ means that we are ignoring constants, and only care about how the number of steps grows with $n$, the size of the input. (This is a form of asymptotic analysis.) If we were being more precise we could look at each loop and ask how many operations it performs per iteration. But even this has its limitations, since each instruction may turn into many machine instructions once the program is compiled. Modern compilers are incredibly sophisticated, so trying to figure out what they will output is not always feasible.

   You could also argue that sometimes isPermutation takes far fewer than $\Theta(n)$ steps, like on the input $\{-9,1,2,3,4,...\}$ since we immediately return false. This brings us to another convention. By default we will focus on worst-case runtime and space utilization analyses. By this we mean that for each $n$ we look at the largest runtime and largest space utilization over any possible length $n$ input.

   Analogously we say that isPermutation uses $\Theta(n)$ space. This is determined by looking at all of the memory we allocate within our function. In addition to a few integer variables we allocate an array `counts` of size $n$ that represents our main memory usage.
2. In the second implementation we use only a constant amount of space (i.e., it doesn’t grow with \( n \)) if we don’t copy the input array. We write \( \Theta(1) \) to denote a constant that doesn’t depend on \( n \). If we do then we use \( \Theta(n) \) space. The runtime has two parts: the time required to sort, and the time required to loop over the array. If we also copy the input, that is an additional time consideration. Suppose we assume the (quicksort in this case) sort requires (worst-case) \( \Theta(n^2) \) steps. Then the total runtime is \( \Theta(n^2) + \Theta(n) \). We simplify this to \( \Theta(n^2) \) since we only care about the term that grows quickest. We will say more about this in a moment.

3. In the third implementation we use \( \Theta(1) \) space and have a runtime of \( \Theta(n^2) \). This is because we call countOcc \( n \) times, and each call to countOcc requires \( \Theta(n) \) operations.

There is a bit of necessary intuition that goes along with the analysis we performed above. Firstly, logarithms grow slowly. We use \( \lg n \) to denote the base-2 logarithm of \( n \). In other words, \( \lg n \) is the value \( v \) such that \( 2^v = n \). If \( n = 1024 \) then \( \lg n = 10 \). If \( n \) is a billion then \( \lg n \) is roughly 30. Secondly, powers (like \( n^{1/2}, n, n^2, n^3 \)) grow much faster than logarithms. Furthermore, if you increase the power by 1 (going from \( n \) to \( n^2 \) or \( n^2 \) to \( n^3 \)) it greatly increases the growth. Here is a table that illustrates these points:

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \lg n )</th>
<th>( n )</th>
<th>( n \lg n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>3.32</td>
<td>10</td>
<td>33.2</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>1000</td>
<td>9.96578</td>
<td>1000</td>
<td>9965.78</td>
<td>10^6</td>
<td>10^9</td>
</tr>
<tr>
<td>( 10^6 )</td>
<td>19.9315</td>
<td>( 10^6 )</td>
<td>19.9 \cdot ( 10^6 )</td>
<td>( 10^{12} )</td>
<td>( 10^{15} )</td>
</tr>
</tbody>
</table>

On large inputs, having a runtime of \( \Theta(n^2) \) or \( \Theta(n^3) \) instead of \( \Theta(n \log n) \) can make the program take an unreasonable amount of time. Another way to look at these growth functions is by changing the input length. Suppose we have a program with \( \Theta(n^2) \) runtime that takes roughly 10 seconds to process our input. How long do we expect it to take if we quadruple the input length? Roughly 160 seconds, or 16 times longer. If we had the same setup but had a \( \Theta(n) \) runtime, the quadrupled input would only require 4 times longer.

You may have heard of “Big-Oh” notation before and noticed its similarity with the \( \Theta \) notation we use here. They are actually slightly different. When you say the runtime is \( \Theta(n^2) \), you are making the statement that it grows like \( n^2 \) up to constants. When you make the statement “the runtime is \( O(n^2) \)” you are saying it grows at most like \( n^2 \) up to constants. That is, Big-Oh denotes an upper bound while the Big-Theta denotes what is called a tight bound (i.e., is more precise). As an example, suppose a function requires \( n \) steps. We can say it has a \( \Theta(n) \) runtime. We can also say it has a \( O(n) \) or \( O(n^2) \) or \( O(n^{1000}) \) runtime, since all of these are upper bounds.

Technical aside (can be ignored) : We will assume a simplified model where our indices and references take constant size. This isn’t technically true, since for an enormous array of size \( n \) (way larger than can fit on the machine say), simply holding an index to a position requires \( \lg n \) space. Alternatively, you can think of our space as omitting an implicit \( \lg n \) factor when dealing with indices/references.
Runtime, Space, and Procedural Design Exercises

1. What is the runtime and space utilization of the following function in terms of the input value $n$? [Don’t try to determine what the program is doing.]

   ```java
   public static int sumfunc(int n)
   {
       int sum = 0;
       for (int i = 1; i <= n; i += 2) sum += i;
       return sum;
   }
   ```

2. What is the runtime and space utilization of the following function in terms of the input value $n$? [Don’t try to determine what the program is doing.]

   ```java
   public static int func(int n)
   {
       int acc = 0;
       for (int i = 1; i < n; i *= 2) acc += i*i;
       return acc;
   }
   ```

3. What is the runtime and space utilization of the following function in terms of the length of the input (which we denote as $n$)? [Don’t try to determine what the program is doing.]

   ```java
   public static int[][] myFunction(int[] arr)
   {
       int n = arr.length;
       int[] tmp = new int[n];
       for (int i = 0; i < n; ++i) tmp[i] = 2*arr[i];
       int[] tmp2 = tmp;
       int[] sums = new int[n];
       for (int i = 0; i < n; ++i)
       {
           for (int j = 1; j < n; ++j) sums[j] += sums[j-1]+tmp[j];
       }
       return sums;
   }
   ```

4. (⋆⋆) Given the following code determine the value of $f(5)$. What are the runtime and space utilization of $f$ in terms of $n$, the input parameter?

   ```java
   public static int f(int n)
   ```
```java
if (n == 0) return 0;
int val = f(n-1);
return val + n;
```

**Solutions**

1. The runtime is $\Theta(n)$ and the space usage is $\Theta(1)$.

2. The runtime is $\Theta(\log n)$ and the space usage is $\Theta(1)$.

3. The runtime is $\Theta(n^2)$ and the memory usage is $\Theta(n)$. The runtime is dominated by the operations performed in the inner most for-loop on line 10. The space usage is dominated by the allocations on lines 4 and 7. Also note that line 6 only requires a constant amount of memory since it only points tmp2 at tmp and doesn’t allocate anything. One last detail. Even though we didn’t mention it above, allocating memory does take time proportional to the size of the memory (since Java zeros out any objects or arrays you allocate). Thus lines 4 and 7 also contribute $\Theta(n)$ to the runtime, but this is dominated by the work on line 10.

4. The value of $f(5)$ is $0 + 1 + 2 + 3 + 4 + 5 = 15$. The runtime is $\Theta(n)$ and the space utilization is $\Theta(n)$. We will explain this in the next section on recursion.

**Introduction to Recursion**

There are three main things one must learn to understand how recursive functions work:

1. (Implementation) How the program stack and activation records work.

2. (Theoretical) How to break a problem into subproblems of the same type.

3. (Coding) The structure that recursive functions usually have.

As this is only an introduction we will review the first and third parts (implementation & coding) and briefly discuss the second (theory). Later on we will spend more time on the theory when discussing trees and divide-and-conquer algorithms.

Each time you call a function, memory is allocated for the local variables and arguments in addition to other things (like where to go once the function returns). This memory is called the *activation record* for your function call. The activation records are organized as a *stack* (we will learn more about stacks later). To illustrate this, we will depict what happens when we call $f(1)$:

1. An activation record is allocated for $f(1)$ with the argument variable filled in:
2. The function is executed until we reach line 4 where we call $f$ again with argument 0. Another activation record is created for $f(0)$ on top of the activation record for $f(1)$. We say the activation record for $f(0)$ is pushed onto the program stack.

3. The call to $f(0)$ returns on line 3 with the value 0. The activation record for $f(0)$ is removed (popped) from the program stack. We return to line 4 in the call to $f(1)$. 
4. The call to \( f(1) \) reaches line 5 where the value 1 is returned. The activation record for \( f(1) \) is popped off the stack. What remains on top of the stack is the activation record for whichever function called \( f(1) \), which we have omitted.

The important take away from the above example is that each call to a function gets its own copies of its argument and local variables. You may have actually seen evidence of the program stack before if you ever looked at one of your runtime exceptions. Here is an example of such a stack trace for a program called IndexError:

```java
import java.util.ArrayList;
public class IndexError {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<>();
        al.get(4);
    }
}
```

Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 4, Size: 0
at java.util.ArrayList.rangeCheck(ArrayList.java:653)
at java.util.ArrayList.get(ArrayList.java:429)
at IndexError.main(IndexError.java:7)

As you can see, main is at the bottom of the stack and on line 7 it calls the get method of ArrayList. On line 429 of the ArrayList implementation the method rangeCheck is called, and then an exception is thrown. We will learn more about exceptions later and how they interact with the program stack.

Secondly, we show how to compute the runtime and memory usage of \( f \), and then investigate what it computes. When we call \( f(n) \) it will in turn call \( f(n-1) \), which in turn calls \( f(n-2) \), etc. This will go on until we get to \( f(0) \) for a total of \( n+1 \) calls to \( f \). Then all of the calls will return. Each call to \( f \) will have a runtime of \( \Theta(1) \) (excluding the recursive call), so the total runtime is \( \Theta(n) \). Furthermore, when we call \( f(0) \) the program stack will have \( \Theta(n) \) activation records on it giving us the \( \Theta(n) \) memory usage. To understand what
is doing, consider the task of adding the first numbers \( n \) positive integers:

\[
1 + 2 + \cdots + (n - 1) + n
\]

and compare it to the task of adding the first \( n - 1 \) positive integers:

\[
1 + 2 + \cdots + (n - 1).
\]

In pseudocode we could write:

```java
int SumIntsUpTo(N)
    If N is 0, return 0
    Else
        let s = SumIntsUpTo(N-1)
        return s + N
```

The key idea here is that the problem of adding the first \( n - 1 \) positive integers can be thought of as a subproblem for adding the first \( n \) positive integers. This is why \( f \) adds the first \( n \) positive integers.

Finally, note that our recursive function \( f \) has the form

Check Stopping Condition (also called base case)
Do work and perform recursive call
Return

You will see this structure again and again when looking at recursive function implementations.

**Program Arguments and the Terminal**

In the past you may have run all of your Java programs by clicking on run in Eclipse. Eclipse is actually doing a bunch of tasks behind the scenes. Suppose we were writing a Java program called Hmm.java with a public class Hmm and used a plain text editor instead of Eclipse. Once we finished editing Hmm.java we could go to the terminal and type

```
javac Hmm.java
```

This will compile Hmm.java into a binary file Hmm.class that is optimized for running. Furthermore, javac will let us know all of the errors the compiler found in the code. Eclipse is constantly compiling our Java files in the background and letting us know (with red underlines) whenever we make an error it can catch. The next step is to run our compiled class file. To do this we write

```
java Hmm
```

This runs a program called the Java Virtual Machine (JVM) and gives it our Hmm.class file to execute. If instead we write

```
java Hmm wow blah
```
then it will again run our class on the JVM but the main function will get a String[] args = 
{"wow","blah"} as an argument. One last bit about the terminal. You have probably written
programs using Scanner that take input from the user on the console (using nextInt and
such). The terminal allows us to use the syntax

    java Hmm < input.txt

This will run the class Hmm and will use input.txt to supply all of the console input. This
way we can test programs that need console input without having to repeatedly type in our
input. You can also type

    java Hmm < input.txt > output.txt

so that all of the input comes from input.txt and all of the output (all of your System.out
stuff) goes to output.txt. Using input.txt and output.txt in this way is called I/O redirection.
In the newest version of Eclipse we can get the same behavior by using an appropriate run
configuration.