CS2040S: Data Structures and Algorithms
Tutorial 1

Problem 1.   Java Review

At this point, most of you should be comfortable enough to work with Java. Let's take some time to review a few concepts in Java so that we can limit our Java-related issues and, hence, focus on the algorithms when solving future Problem Sets.

(a) What is the difference between a class and an object? Illustrate with an example.

Answer: A class can be seen as a 'template', or a 'blueprint', specifying what kind of methods/operations should be supported and its behaviour. An object is an instance of a class.

(b) Why does the `main` method come with a `static` modifier?

Answer: Java program's main method has to be declared static because keyword static allows main to be called without creating an object of the class in which the main method is defined. If we omit static keyword before main Java program will successfully compile but it won't execute.

(c) Give an example class (or classes) that uses the modifier `private` incorrectly (i.e., the program will not compile as is, but would compile if `private` were changed to `public`).

Possible Answer:

```
class SecretHolder {
    private int secret;

    public SecretHolder(int value){
        this.secret = value;
    }
}
class Test {
    public static void main(String[] args){
        var holder = new SecretHolder(5);
        holder.secret = 6; // Compile-time error!
    }
}
```

(d) The following question is about Interfaces.

 (d)(i) Why do we use interfaces?
 (d)(ii) Give an example of using an interface.

(d)(iii) Can a method return an interface?

Answers:

(d)(i) An interface can be seen as a 'contract' that is signed by a class whenever it `implements` the interface. The reason for using interfaces is that whenever we see a class that implements that interface, we know for certain that it supports the operations specified by that interface.

(d)(ii) See Problem Set 1.

(d)(iii) Yes, see Problem Set 1: `ShiftRegisterTest.java`.

(e) Refer to `IntegerExamination.java` in Coursemology. Without running the code, predict the output of the `main` method. Can you explain the outputs?

Answer: The expected output consists of a string, *"I am in myOtherIntAddOne. The value of i is 8"*, followed by three numbers: 7, 8 and 7.

Most of the intuition is already mentioned in the code itself as comments. If you understand that, then you should understand the answer.

- Within the `addOne` method, the variable `i` refers to the argument of the method and not the variable `i` in the `main` method. As such, the variable `i` in the main method remains unchanged as 7.

- Because `myInteger` is a class (as opposed to a primitive data type like `int`), `j` is an object. That means when it is passed as an argument to the `myIntAddOne` method, the variable `i` in the method is pointing to the same object as `j` in the `main` method. As a result, when the `intState` of `i` is increased, that change is reflected in `j`, thus saving the value as 8.

- This is a (very) rough guide on what goes on under the hood of Java. The variable `k` in the `main` method does not store a `myInteger` object; it stores a pointer to that `myInteger` object, stored somewhere in memory (i.e. `k` stores the address). For example, let's say `k` contains something like `0x00AB38A0`.

  When `k` is passed as an argument to the `myOtherIntAddOne` method, what's really happening is that you're passing a copy of `0x00AB38A0`, which is what is stored in the variable `i` within the `myOtherIntAddOne` method. In other words, aside from the contents being the same, there is no link between `i` in the `myOtherIntAddOne` method and `k` in the `main` method.

  When a new `myInteger` object is constructed and assigned into `i`, `i` will contain the address of a brand new `myInteger` object. However, `k` will still be containing `0x00AB38A0`, the address of the original `myInteger` object, which still stores 7.

  This is why it's incorrect to call Java either 'pass-by-reference' or 'pass-by-value'; it's more accurate to call it 'pass-reference-by-value'.

(f) Can a variable in a parameter list for a method have the same name as a member (or static) variable in the class? If yes, how is the conflict of names resolved?

Use `this`.

```
class Example {
    int value = 5;

    public void clash(int value) {
        // Refers to the 'value' argument.
        System.out.println(value);
        // Refers to the 'value' member in the class.
        System.out.println(this.value);
    }
}
```

Problem 2.   Asymptotic Analysis

This is a good time for a quick review of asymptotic big-O notation. For each of the expressions below, what is the best (i.e. tightest) asymptotic upper bound (in terms of n)?

You might point out that for big-O, larger is always true, e.g., the function $f(n) = n$ is mathematically $O(2^n)$. But when we talk about the "best" asymptotic bound or a "tight" asymptotic bound, we mean $O(n)$ because $f(n) = \Theta(n)$.

$$f_1(n) = 7.2 + 34n^3 + 3254n \qquad\qquad f_2(n) = n^2 \log n + 25n \log^2 n$$
$$f_3(n) = 2^{4 \log n} + 5n^5 \qquad\qquad f_4(n) = 2^{2n^2 + 4n + 7}$$

Answers:

1. $f_1(n) = O(n^3)$

   The general strategy is to take the most dominant term and drop the coefficients. Here, that refers to the term of largest degree, $34n^3$.

2. $f_2(n) = O(n^2 \log n)$

   In general, a term of the from n^a for any positive constant a will be dominant over a term of the form $\log^b n$ for any positive constant b. For example, $n^{0.02}$ is dominant over $\log^{2040} n$. Hence, $n^2 \log n$ is the dominant term in $f_2(n)$.

3

3. $f_3(n) = O(n^5)$

Here, we use the following two properties to simplify the $2^{4\log n}$ term:

$$a^{mn} = (a^m)^n \quad a^{\log_a b} = b$$

Using these properties,

$$2^{4\log n} = \left(2^{\log n}\right)^4$$
$$= n^4$$

This makes $5n^5$ the dominant term, resulting in the final answer.

4. $f_4(n) = O(2^{2n^2+4n})$

We can move the 7 down from the exponent by doing the following:

$$2^{2n^2+4n+7} = 2^{2n^2+4n} \cdot 2^7$$

We can then remove the 2^7 coefficient. However, we cannot remove any of the other terms, neither can we remove any of the coefficients from the exponents.

For example, $2^{2n} \neq O(2^n)$. One possible intuition is that $2^{2n} = (2^n)^2$.

Problem 3.   More Asymptotic Analysis!

Let f and g be functions of n where $f(n) = O(n)$ and $g(n) = O(\log n)$. Find the best asymptotic bound *(if possible)* of the following functions.

(a) $h_1(n) = f(n) + g(n)$

(b) $h_2(n) = f(n) \times g(n)$

(c) $h_3(n) = \max(f(n), g(n))$

(d) $h_4(n) = f(g(n))$

(e) $h_5(n) = f(n)^{g(n)}$

Answers:

For simplicity, let's say that $f(n) \leq c_1 \cdot n$ and $g(n) \leq c_2 \cdot \log n$ for sufficiently large n, where c_1 and c_2 are some positive constants.

1. $h_1(n) \leq c_1 n + c_2 \log n$
    $= O(n)$

2. $h_2(n) \leq c_1 n \cdot c_2 \log n$
    $= c_1 c_2 n \log n$
    $= O(n \log n)$

3. $h_3(n) = \max(f(n), g(n))$
    $= O(f(n) + g(n))$
    $= O(n)$

4. $h_4(n) \leq c_1 g(n)$
    $\leq c_1 c_2 \log n \quad = O(\log n)$

    Note that this explanation is not very rigorous, but the intuitive idea should be there.

5. Trick question! The constant matters in the exponent here.

    For example, if $f(n) = n$ and $g(n) = \log n$, then $h_5(n) = O(n^{\log n})$. However, if $g(n) = 2 \log n$, then $h_5(n) = O(n^{2 \log n}) = O((n^{\log n})^2)$

Problem 4.   Time complexity analysis

Analyse the following code snippets and find the best asymptotic bound for the time complexity of the following functions with respect to n.

(a)
```
public int niceFunction(int n) {
    for (int i = 0; i < n; i++) {
        System.out.println("I am nice!");
    }
    return 42;
}
```

Answer: $O(n)$

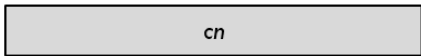This should be straightforward; it's a for-loop that runs for a total of n iterations.
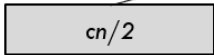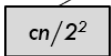
(b)
```
public int meanFunction(int n) {
    if (n == 0) return 0;
    return 2 * meanFunction(n / 2) + niceFunction(n);
}
```

Answer: $O(n)$

For a recursive function, the first step is to construct the recurrence relation.

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

There are many ways to solve this recurrence relation. One of the simplest ways is to draw the recurrence tree:

Level	Function Call	Work done	Total Work Done
0	$T(n)$	cn	cn
1	$T(n/2)$	$cn/2$	$cn/2$
2	$T(n/2^2)$	$cn/2^2$	$cn/2^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
h	$T(1)$	c	c

To solve the recurrence relation, simply sum up the total work done over all the levels. Additionally, when we encounter a geometric series, our lives are made simpler; A geometric series is upper bounded by the largest term.

$$
\begin{aligned}
T(n) &= cn + \frac{cn}{2} + \frac{cn}{2^2} + \cdots + c \\
&\leq cn + \frac{cn}{2} + \frac{cn}{2^2} + \cdots && \text{(Upper bounded by the infinite sum)} \\
&= \sum_{r=0}^{\infty} \frac{cn}{2^{r-1}} \\
&= \frac{cn}{1 - \frac{1}{2}} && \text{(Formula for infinite geometric series)} \\
&= O(n)
\end{aligned}
$$

(c)
```java
public int strangerFunction(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      System.out.println("Execute order?");
    }
  }
  return 66;
}
```
Answer: $O(n^2)$

This is a standard nested for-loop. Roughly speaking, during the i-th iteration of the outer for-loop, the inner for-loop runs for i iterations. If we add them all up, we get:

$$
\begin{aligned}
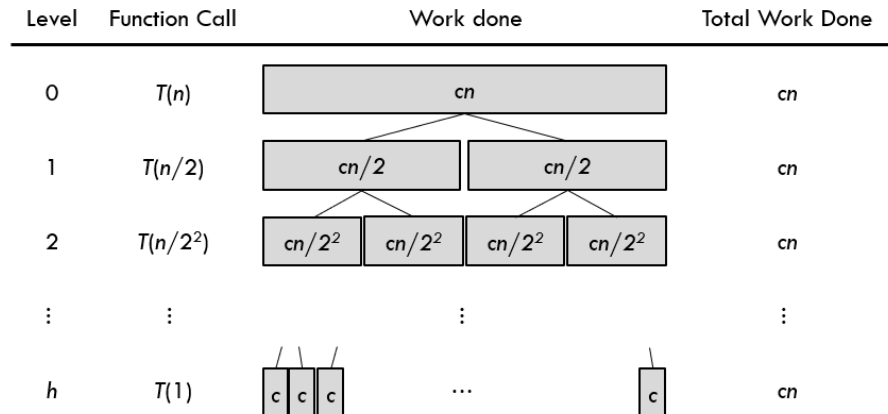1 + 2 + 3 + \cdots + n &= \frac{(n)(n+1)}{2} \\
&= O(n^2)
\end{aligned}
$$

(d)
```java
public int suspiciousFunction(int n) {
  if (n == 0) return 2040;

  int a = suspiciousFunction(n / 2);
  int b = suspiciousFunction(n / 2);
  return a + b + niceFunction(n);
}
```

Answer: $O(n \log n)$

Constructing the recurrence relation, we have:

You might have noticed that this is exactly the recurrence relation for Merge Sort, and hence, can immediately declare the answer to be $O(n \log n)$. Otherwise, we can draw the recurrence tree:

Notice how the total amount of work done in each level sums up to cn. Therefore, we can simply multiply cn by the height of this tree. The function call at the k-th level is $T(n/2^k)$. To determine the height of the tree, we have to determine the value of h for which $n/2^h = 1$ (i.e. the base case).

$$\frac{n}{2^h} = 1$$
$$2^h = n$$
$$h = \log_2 n$$

Therefore, the total amount of work done across all levels is $(cn)(\log_2 n + 1) = O(n \log n)$.

(e)
```
public int badFunction(int n) {
    if (n <= 0) return 2040;
    if (n == 1) return 2040;
    return badFunction(n - 1) + badFunction(n - 2) + 0;
}
```

Answer: $O(\phi^n)$

This is actually similar to the Fibonacci sequence. So the trivial bound is $O(2^n)$, the tigher bound is $O(\phi^n)$ where ϕ is the golden ratio. **But take note!** A faulty argument is to say "this program looks like Fibonacci, so it takes $F(n)$ time, where $F(n)$ is the n-th Fibonacci number". This is because the recurrence for the work done is actually $T(n) = T(n-1) + T(n-2) + O(1)$, which is not the same as $F(n) = F(n-1) + F(n-2)$ (there's an additional $+1$ term). Instead, we can prove it by induction, with the hypothesis that $T(i) = F(i) - 1, \forall i < n$. Then:

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= F(n-1) - 1 + F(n-2) - 1 + 1 \\
&= F(n-1) + F(n-2) + 1
\end{aligned}
$$

One more thing, notice that in this case, $T(n)$ seems to be 0, which would imply that our base case is wrong. This can be remedied by the fact that, we argue that this should only hold for $n \geq 2$. As in this bound on our runtime applies on the algorithm on problem sizes 2 and above. Whilst this might feel clunky, this argument is actually acceptable even in the context of algorithms. Why?

(f)
```java
public int metalGearFunction(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 1; j < i; j *= 2) {
      System.out.println("!");
    }
  }
  return 0;
}
```

Answer: $O(n \log n)$

We have another nested for-loop. But this time, during the i-th iteration of the outer for-loop, the inner for-loop runs for about $(\log i)$ iterations. If we add them up, we get:

$$
\begin{aligned}
\log(1) + \log(2) + \cdots + \log(n) &= \log(1 \cdot 2 \cdot \cdots \cdot n) \\
&= \log(n!) \qquad\qquad\qquad\quad = O(n \log n)
\end{aligned}
$$

The last statement has been mentioned in Lecture 1. For a simple way to upper bound it,

$$
\begin{aligned}
\log(1) + \log(2) + \cdots + \log(n) &\leq \log(n) + \log(n) + \cdots + \log(n) \\
&= n \log n \\
&= O(n \log n)
\end{aligned}
$$

9

Problem 5.   Another Application of Binary Search (Optional) Given a sorted array of $n-1$ unique elements in the range $[1, n]$, find the missing element? Discuss possible naive solutions and possibly faster solutions.

   Answer: The idea is to check the element in the middle of the the array first (at index $\lfloor \frac{n}{2} \rfloor$. If the element is $\lfloor \frac{n}{2} \rfloor$, then we know that all the elements in the left half of the array are present, so the missing element is in the right half, so we should recurse on that half instead, else the element what we are looking at should be $\lfloor \frac{n}{2} \rfloor + 1$, which would mean the missing element should be on the left half of the array.