

Discussion Group Problems for Week 3

For: January 24–January 28

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.

Solution: 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?

Solution: Java program's `main` method has to be declared `static` because the keyword `static` allows `main` to be called without creating an object of the class in which the `main` method is defined. If we omit the `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 it is, but would compile if `private` was changed to `public`).

Solution:

Below is a possible solution:

```
class SecretHolder {
    private int secret;

    public SecretHolder(int value) {
        this.secret = value;
    }
}

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

- (d) The following question is about Interfaces.

- (d)(i) Why do we use interfaces?

Solution: 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) Give an example of using an interface.

Solution: See Problem Set 1.

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

Solution: Yes, see Problem Set 1: `ShiftRegisterTest.java`.

(e) Refer to `IntegerExamination.java`, which can be found in the same folder as this PDF. Without running the code, predict the output of the `main` method. Can you explain the outputs?

Solution:

The expected output consists of the following 6 strings:

I am in `addOne`. The value of `i` is 8

I am in `myIntAddOne`. The value of `j` is 8

I am in `myOtherIntAddOne`. The value of `k` is 8

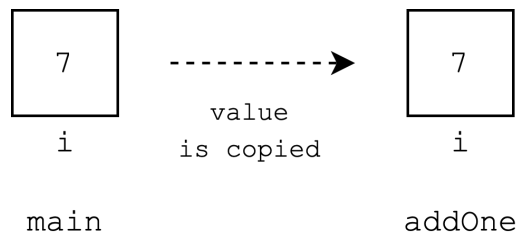
The final value of `i` back in `main` is 7

The final value of `j` back in `main` is 8

The final value of `k` back in `main` is 7

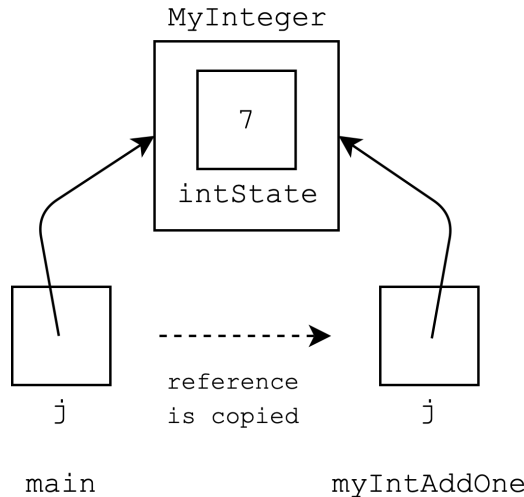
Most of the intuition is already mentioned in the code itself as comments. If you understand that, then you should understand the answer.

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



Solution: *(Continued)*

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



- We see something similar happening for variable `k` and `myOtherIntAddOne`. The variable `k` in `myOtherIntAddOne` initially points to the same object that `k` in `main` is pointing to. However, a new `MyInteger` object is subsequently constructed and assigned to `k` in `myOtherIntAddOne`. The original `MyInteger` object, which `k` in `main` is still pointing to, remains unchanged.
- In the above diagram, we have visualised the object reference as a box with an arrow pointing to the object instance. More precisely, this pointer is actually an address value. For example, `k` may contain something like `0x00AB38A0`, which is the value of the memory address where the referenced `MyInteger` object is stored.

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

When a new `MyInteger` object is constructed and assigned to `k`, `k` in `myOtherIntAddOne` will now contain the address of a brand new `MyInteger` object. However, `k` in `main` 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?

Solution: 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)?

(a) $f_1(n) = 7.2 + 34n^3 + 3254n$

Solution: $f_1(n) = O(n^3)$

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

(b) $f_2(n) = n^2 \log n + 25n \log^2 n$

Solution: $f_2(n) = O(n^2 \log n)$

In general, a term of the form 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 $f_2(n)$.

(c) $f_3(n) = 2^{4 \log n} + 5n^5$

Solution: $f_3(n) = O(n^5)$

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

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

Using these properties,

$$\begin{aligned} 2^{4 \log n} &= (2^{\log n})^4 \\ &= n^4 \end{aligned}$$

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

(d) $f_4(n) = 2^{2n^2+4n+7}$

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

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

Solution:
$$\begin{aligned} h_1(n) &\leq c_1 n + c_2 \log n \\ &= O(n) \end{aligned}$$

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

Solution:
$$\begin{aligned} h_2(n) &\leq c_1 n \cdot c_2 \log n \\ &= c_1 c_2 n \log n \\ &= O(n \log n) \end{aligned}$$

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

Solution:
$$\begin{aligned} h_3(n) &= \max(f(n), g(n)) \\ &= O(f(n) + g(n)) \\ &= O(n) \end{aligned}$$

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

Solution:
$$\begin{aligned} h_4(n) &\leq c_1 g(n) \\ &\leq c_1 c_2 \log n \\ &= O(\log n) \end{aligned}$$

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

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

Solution: 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;
}
```

```
}
```

Solution: $O(n)$

This should be straightforward; it's a for-loop that runs for a total of n iterations.

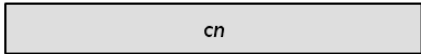
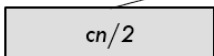
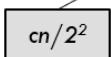

```
(b) public int meanFunction(int n) {  
    if (n == 0) return 0;  
    return 2 * meanFunction(n / 2) + niceFunction(n);  
}
```

Solution: $O(n)$

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 |
| 1 | $T(n/2)$ |  | $cn/2$ |
| 2 | $T(n/2^2)$ |  | $cn/2^2$ |
| \vdots | \vdots | \vdots | \vdots |
| h | $T(1)$ |  | 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.

$$\begin{aligned}
 T(n) &= cn + T\left(\frac{n}{2}\right) \\
 &= cn + \frac{cn}{2} + T\left(\frac{n}{2^2}\right) \\
 &= cn + \frac{cn}{2} + \frac{cn}{2^2} + T\left(\frac{n}{2^3}\right) \\
 &= \left(cn + \frac{cn}{2} + \frac{cn}{2^2} + \dots\right) + T(1) \\
 &\leq cn + \frac{cn}{2} + \frac{cn}{2^2} + \dots && \text{(Upper bounded by the infinite sum)} \\
 &= \sum_{r=0}^{\infty} \frac{cn}{2^r} \\
 &= \frac{cn}{1 - \frac{1}{2}} && \text{(Formula for infinite geometric series)} \\
 &= O(n)
 \end{aligned}$$

```
(c) 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;
}

```

Solution: $O(n^2)$

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:

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 \\
 &= \sum_{i=0}^{n-1} i \\
 &= 0 + 1 + 2 + 3 + \cdots + (n-1) \\
 &= \frac{n(n-1)}{2} \\
 &= O(n^2)
 \end{aligned}$$

```

(d) 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);
}

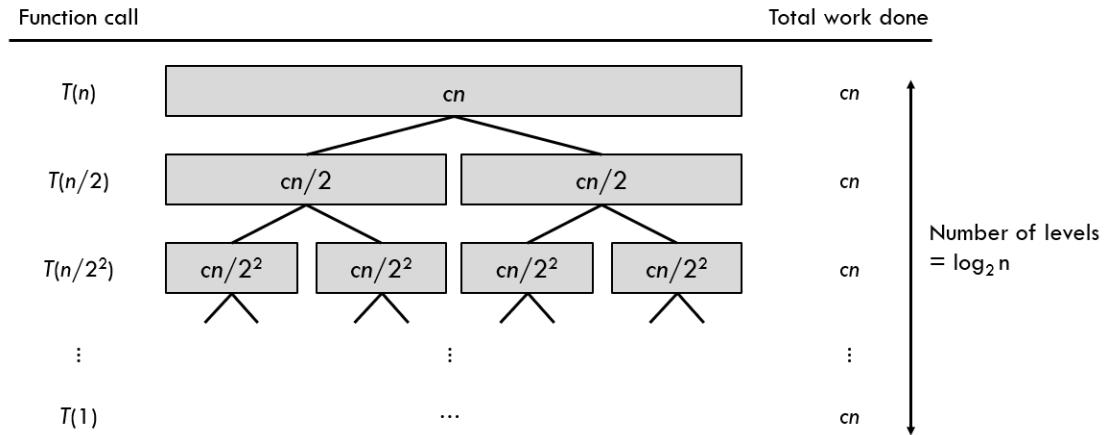
```

Solution: $O(n \log n)$

Constructing the recurrence relation, we have:

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

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).

$$\begin{aligned}\frac{n}{2^h} &= 1 \\ 2^h &= n \\ h &= \log_2 n\end{aligned}$$

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;
}
```

Solution: $O(\phi^n)$

The trivial bound is $O(2^n)$. This is actually similar to the Fibonacci sequence and the tighter bound is $O(\phi^n)$ where ϕ is the golden ratio.

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 using induction, with $F(0) = F(1) = 1, T(0) = T(1) = 0$. Suppose n is a non-negative integer such that $T(i) = F(i) - 1, \forall i < n$. Then, by basic algebra, we can also say that:

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

Hence, $T(n-1) = F(n-1) - 1$ implies $T(n) = F(n) - 1$.

There are several points that you need to take note of:

- First, notice that the base case for the relation T ($T(0) = T(1) = 0$), given the context that we are analysing computational cost of the algorithm, does not make sense. The computational cost can’t be 0. This contradiction can be resolved by the fact that, 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 in the context of algorithms. Why?
- Second, notice that it does not make intuitive sense that $T(n) < F(n)$ as $T(n)$ is the relation that has the additional +1 term. The relation between the relations is only true because the base cases of $T(n)$ and $F(n)$ are different. We are actually able to prove an alternative relation $T(n) = F(n+1) - 1$ with the base cases $T(0) = 0, T(1) = 1, F(0) = 1, F(1) = 1$, where $T(n) = F(n+1) - 1$ makes more intuitive sense. Just take note that we only decided to prove $T(n) = F(n) - 1$ because it’s the most convenient.

```
(f) 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;  
}
```

Solution: $O(n \log n)$

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:

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

For a simple way to upper bound $\log(n!)$,

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

```
(g) public String simpleFunction(int n) {  
    String s = "";  
    for (int i = 0; i < n; i++) {  
        s += "?";  
    }  
    return s;  
}
```

Solution: $O(n^2)$

We have string concatenation in a loop here. On each concatenation, a new copy of the string is created, which takes $O(\text{length of string})$ time. During the i -th iteration of the loop, the string is $i - 1$ characters long. The time complexity is very similar to that in part (c). If we add them all up, we get:

$$\begin{aligned}T(n) &= \sum_{i=0}^{n-1} i \\ &= 0 + 1 + 2 + 3 + \cdots + (n - 1) \\ &= \frac{n(n - 1)}{2} \\ &= O(n^2)\end{aligned}$$

Problem 5. Another Application of Binary Search

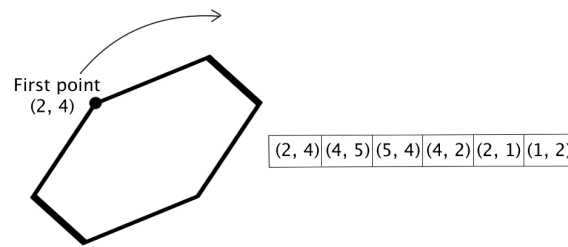
Given a sorted array of $n - 1$ unique integers in the range $[1, n]$, how would you find the missing element? Discuss possible naive solutions and possibly faster solutions.

Solution: 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 we are looking at should be $\lfloor \frac{n}{2} \rfloor + 1$, which would mean the missing element should be in the left half of the array.

Problem 6. Yet Another Application of Binary Search

(Optional) Given an array of n x and y -coordinates of an n -sided convex polygon in clockwise order, find a bounding box around the polygon. Discuss possible naive solutions and possibly faster solutions. A convex polygon is a polygon where all interior angles are less than 180 degrees.

An example of such an array is shown below:

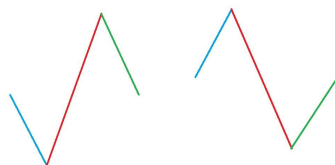


Solution: The problem is equivalent to finding the minimum and maximum x and y -coordinates of the polygon. We can start off by assuming that no two consecutive points will have the same x or y -coordinate value.

The key idea to note is that a **convex** polygon's x and y -coordinates will have **at most** two “changes in direction”, i.e. be either increasing-decreasing-increasing, or decreasing-increasing-decreasing. If you start at the minimum or maximum point, then it is possible to have only a single “change in direction”, i.e. it would be either increasing-decreasing or decreasing-increasing, or have none at all, i.e. simply increasing or decreasing. This question then becomes a problem of peak-finding, similar to Q1 of Problem Set 2.

The complexity of this question is greatly reduced by the fact that the points are in a clockwise order, and that the end points are adjacent points of a closed convex polygon. This lets us skip over a lot of the edge cases that come with binary searching an array with two “changes in directions”.

Let us illustrate this with some diagrams. Suppose we plot out x -coordinates against index of the points for an arbitrary polygon. We might see a curve that's similar to the two curves below:



One important insight is that the blue and green parts of the curve do not overlap, since it is a closed convex polygon. We are thus able to verify which part of the array we are in easily as long as we take note of the first element's value, the middle element's value and the direction that the array **starts with**. For example, in an array that goes increasing-decreasing-increasing, if our binary search lands on a decreasing portion of the array, we know that we are between the global maximum and minimum. If we land on an increasing portion of the array, we know that we are on the first increasing section if the value of the middle element is greater than that of the first element, and we know that we are on the second increasing section if the value of the middle element is smaller than that of the first element. A similar argument can be made for the decreasing-increasing-decreasing case. Knowing this, we can easily binary search in the right direction to find a peak/valley. You can then find the other extreme end with another simple binary search.

As for the scenario where two consecutive points have the same x or y -coordinates—in a convex polygon, if two adjacent points have the same x -coordinates, then those two points must be the top and bottom points of a fully vertical side, and this side would be the leftmost or rightmost side of the polygon. Similarly, same y -coordinates means they are the left and right points of a fully horizontal side that is at the top or the bottom of the polygon. As such, even if the array is not composed of unique values for consecutive points, you can be sure that the binary search won't be stuck at a plateau because any plateau is guaranteed to be a peak.