**CS2040S: Data Structures and Algorithms**

# Problems for Week 3: Asymptotic Analysis

*For: 23 Aug 2021, Tutorial 1*

**Solution: Secret! Shhhh... This is the solutions sheet.**

## Problem 1. Big-O Time Complexity

Big-$O$ time complexity gives us an idea of the growth rate of a function. In other words, for a large input size $N$, as $N$ increases, in what order of magnitude is the volume of statements executed expected to increase?

Rearrange the following functions in increasing order of their Big-$O$ complexity. Use $\prec$ to indicate that the function on the left is upper-bounded by the function on the right, and $\equiv$ to indicate that two functions have the same big-$O$ time complexity. An example is given below.

**Example.** For the following functions:

| | | | |
|---|---|---|---|
| $5n$ | $\frac{1}{2}n^3$ | $n$ | $3n^2$ |

The correct arrangement is

$$n \equiv 5n \prec 3n^2 \prec \frac{1}{2}n^3$$

because $n = O(n)$, $5n = O(n)$, $3n^2 = O(n^2)$ and $\frac{1}{2}n^3 = O(n^3)$.

**Now, you try!** Rearrange the following 16 functions in ascending order using $\prec$ and $\equiv$.

| | | | |
|---|---|---|---|
| $4n^2$ | $\log_3 n$ | $20n$ | $n^{2.5}$ |
| $n^{0.00000001}$ | $\log n!$ | $n^n$ | $2^n$ |
| $2^{n+1}$ | $2^{2n}$ | $3^n$ | $n \log n$ |
| $100n^{\frac{2}{3}}$ | $\log[(\log n)^2]$ | $n!$ | $(n-1)!$ |

**Solution:** A systematic way of ordering the 16 functions is as follows.

**Step 1.** Write the **tightest** upper bound of all 16 functions using big-$O$ notation.

| | | | |
|---|---|---|---|
| $4n^2 = O(n^2)$ | $\log_3 n = O(\log n)$ | $20n = O(n)$ | $n^{2.5} = O(n^{2.5})$ |
| $n^{0.00000001} = O(n^{0.00000001})$ | $\log n! = O(n \log n)$ | $n^n = O(n^n)$ | $2^n = O(2^n)$ |
| $2^{n+1} = O(2^n)$ | $2^{2n} = O(2^{2n}) = O(4^n)$ | $3^n = O(3^n)$ | $n \log n = O(n \log n)$ |
| $100n^{\frac{2}{3}} = O(n^{\frac{2}{3}})$ | $\log[(\log n)^2] = O(\log \log n)$ | $n! = O(n!)$ | $(n-1)! = O((n-1)!)$ |

Some important points to highlight below:

- **In general, if the base of a logarithm is a constant, it does not matter.** This is because if the base of a logarithm is a constant, we can always change it to any other constant using the change of base rule. For example,

$$\log_3 n = \frac{\log_k n}{\log_k 3} = \frac{1}{\log_k 3} \cdot \log_k n = O(\log_k n)$$

  for any positive constant $k$. For this reason, we usually omit the base of logarithms in big-$O$ notation.

  **Question.** What if, instead of $\log_3 n$, we have $\log_b n$ instead, for a **variable** $b$? Does the above simplification still work?

- $\log n! = O(n \log n)$ because:

$$\log n! = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n = O(n \log n)$$

  where we use the fact that $\log ab = \log a + \log b$ to obtain the second step from the first.

  **Question.** Notice that from the above, $\log n! \leq n \log n$. Yet, we say that $O(n \log n)$ is a tight upper bound for $\log n!$. Compare this to the case of $(n-1)!$ — even though $(n-1)! \leq n!$, we are not using $O(n!)$ as a tight upper bound for $(n-1)!$ instead. Why is this so?

(Solution continued on next page)

**Step 2.** Group the functions into different classes. The goal of grouping the functions into different classes is to allow us to compare functions in smaller groups more easily, instead of comparing all 16 functions in one go. Note that the classification below is fairly detailed — you might not need so many classes to help you obtain the correct ordering!

- **Logarithmic Functions.** $\log[(\log n)^2] = O(\log \log n)$, $\log_3 n = O(\log n)$
- **Sublinear Power Functions.** $n^{0.00000001} = O(n^{0.00000001})$, $100n^{\frac{2}{3}} = O(n^{\frac{2}{3}})$
- **Linear Functions.** $20n = O(n)$
- **Linearithmic Functions.** $n \log n = O(n \log n)$, $\log n! = O(n \log n)$
- **Quadratic Functions.** $4n^2 = O(n^2)$
- **Polynomial Functions.** $n^{2.5} = O(n^{2.5})$
- **Exponential Functions.** $2^n = O(2^n)$, $2^{n+1} = O(2^n)$, $3^n = O(3^n)$, $2^{2n} = O(4^n)$
- **Factorial Functions.** $(n-1)! = O((n-1)!)$, $n! = O(n!)$
- **Tetration.** $n^n = O(n^n)$

**Step 3.** Arrange the functions in increasing order. Note that any logarithmic function grows slower than any power function.

$$\log[(\log n)^2] \prec \log_3 n \prec n^{0.00000001} \prec 100n^{\frac{2}{3}} \prec 20n \prec n \log n \equiv$$
$$\log n! \prec 4n^2 \prec n^{2.5} \prec 2^n \equiv 2^{n+1} \prec 3^n \prec 2^{2n} \prec (n-1)! \prec n! \prec n^n$$

**Problem 2. Time Complexity Analysis**

Find the **tightest** big-$O$ time complexity of each of the following code fragments.

**Problem 2.a.** The big-$O$ time complexity of the following code fragment, in terms of $n$.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("*");
    }
}
```

**Solution:** $O(n^2)$

First, we need to count how many statements are executed, relative to the input size $n$. In particular, we need to count how many times the statement "`System.out.println("*");`" is executed, relative to the input size $n$.

Observe that the outer loop runs for $n$ iterations, and in the $i$th iteration of the outer loop, the inner loop executes for $i$ iterations, and so the statement "`System.out.println("*");`" is executed for $i$ times:

| $i$ | # iterations of inner loop | # times `System.out.println("*");` is executed |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| ... | ... | ... |
| $n-1$ | $n-1$ | $n-1$ |

Therefore, the total number of times "`System.out.println("*");`" is executed is:

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

Each execution of "`System.out.println("*");`" runs in $O(1)$ time, hence, the code fragment runs in $O(n^2)$ time.

**Problem 2.b.** The big-$O$ time complexity of the following code fragment, in terms of $n$.

```
int i = 1;
while (i <= n) {
    System.out.println("*");
    i = 2 * i;
}
```

**Solution:** $O(\log n)$

First, let's try to figure out how many iterations the while loop executes. Notice that the while loop stops after the value of $i$ exceeds $n$. Also, notice that after each iteration of the while loop, the value of $i$ doubles.

| iteration of while loop | value of $i$ at beginning of iteration |
|---|---|
| 1 | $1 = 2^0$ |
| 2 | $2 = 2^1$ |
| 3 | $4 = 2^2$ |
| 4 | $8 = 2^3$ |
| ... | ... |
| ??? | $n$ |

Notice that at the beginning of the $k$th iteration of the while loop, the value of $i = 2^{k-1}$. The loop stops when the value of $i = 2^{k-1} > n$. Solving the inequality:

$$\begin{aligned} 2^{k-1} > n &\Rightarrow \log_2 2^{k-1} > \log_2 n \\ &\Rightarrow (k-1)\log_2 2 > \log_2 n \\ &\Rightarrow (k-1) > \log_2 n \\ &\Rightarrow k > \log_2 n + 1 \end{aligned}$$

So the while loop terminates after the $O(\log n)$ iterations. Since the statements inside the while loop run in $O(1)$ time, the code fragment runs in $O(\log n)$ time overall.

**In general, if the variable controlling the loop increases or decreases exponentially after every iteration, the loop will execute for $O(\log n)$ iterations.**

**Problem 2.c.** The big-$O$ time complexity of the following code fragment, in terms of $n$.

```
int i = n;
while (i > 0) {
    for (int j = 0; j < n; j++)
        System.out.println("*");
    i = i / 2;
}
```

**Solution:** $O(n \log n)$

First, let's try to figure out how many iterations the while loop executes. Notice that the while loop stops after the value of $i$ reaches 0. Also, notice that after each iteration of the while loop, the value of $i$ halves. This is identical to Problem 2b: the value of $i$, which is controlling the loop, decreases exponentially, and thus the while loop runs for $O(\log n)$ iterations.

Next, in every iteration of the while loop, we have a for loop that runs for $O(n)$ iterations. Note that the two loops here are nested, but the **number of iterations the inner loop runs is independent of the outer loop**. Therefore, the total number of statements executed can be taken by multiplying both values together. Hence, the time complexity of this code fragment is $O(n \log n)$.

**Problem 2.d.** The big-$O$ time complexity of the following code fragment, in terms of $n$.

```
while (n > 0) {
    for (int j = 0; j < n; j++)
        System.out.println("*");
    n = n / 2;
}
```

**Solution:** $O(n)$

Here, the number of iterations of the while loop is still $O(\log n)$. However, we cannot examine each loop one at a time and then multiply the number of iterations together. This is because after each iteration of the outer loop, the value of $n$ changes, and hence the number of times the inner loop executes also changes. Therefore, we have to fall back to summing up the number of inner loop iterations to find the total number of statements executed.

| iteration of while loop | value of $n$ at beginning of iteration | # for loop iterations |
|---|---|---|
| 1 | $n$ | $n$ |
| 2 | $\frac{n}{2}$ | $\frac{n}{2}$ |
| 3 | $\frac{n}{4}$ | $\frac{n}{4}$ |
| ... | ... | ... |
| $O(\log n)$ | 1 | 1 |

Summing the number of for loop iterations, we obtain a geometric series:

$$\begin{aligned} n + \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + 1 &\leq n(1 + \frac{1}{2} + \frac{1}{4} + \dots) \\ &= n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= n \cdot \frac{1}{1 - \frac{1}{2}} = 2n = O(n) \end{aligned}$$

Hence, the code fragment runs in $O(n)$ time.

**Problem 2.e.** The big-$O$ time complexity of the following code fragment, in terms of $n$ and $m$.

```
String x; // String x has length n
String y; // String y has length m
String z = x + y;
System.out.println(z);
```

**Solution:** $O(n+m)$

This problem is used to highlight a common misconception: concatenating two strings of variable length does **not** take $O(1)$ time. Java note: There is **nothing** in the Java API that allows you to concatenate two variable length strings in $O(1)$ time. Be careful when working with strings!

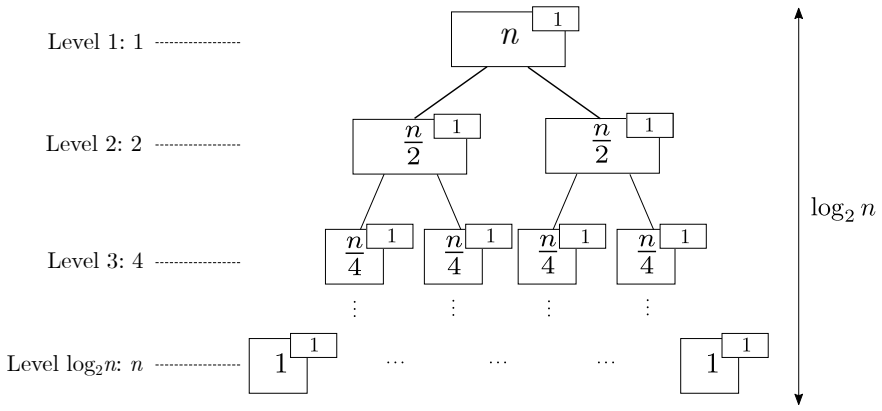
**Problem 2.f.** The big-$O$ time complexity of the following function, in terms of $n$.

```
void foo(int n){
    if (n <= 1)
        return;
    System.out.println("*");
    foo(n/2);
    foo(n/2);
}
```

**Solution:** $O(n)$

When given a recursive function, it is often useful to draw a **recursion tree** to analyze the function. We draw a tree of function calls, figure out how much work is done per node, then per level in the tree, before adding everything up to find out the overall time complexity.

The recursion tree for the function in Problem 2f is shown below. Every node in the recursion tree represents a single call of the `foo` function. The input to the function is written in each node in the recursion tree. The initial function call receives an input of $n$, from which two function calls with $\frac{n}{2}$ as input are generated. A single function call runs in $O(1)$ time, indicated in the box on the top right of every node. The recursion tree has a height of $\log_2 n$, as the input to the function calls decrease exponentially down the recursion tree.




If we sum up the values in the small boxes at the top right of every node for every level, we can see that the work done on the $i$th level is $2^{i-1}$. Summing up the work done across all $\log_2 n$ levels in the recursion tree gives us the time complexity of the function. Using the geometric series formula,

$$\begin{aligned} 1 + 2 + 4 + \cdots + n &= \sum_{i=0}^{\log_2 n} 2^i \\ &= \frac{1 - 2^{\log_2 n + 1}}{1 - 2} \\ &= 2n - 1 \\ &= O(n) \end{aligned}$$

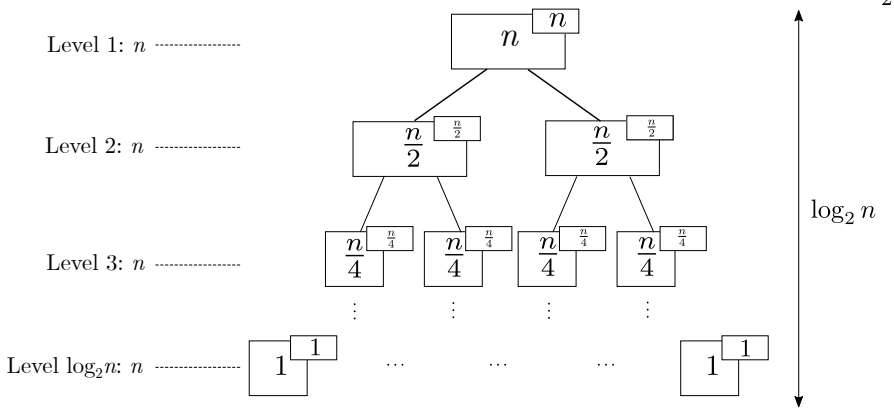
In the above, we have used the identity $a^{\log_c b} = b^{\log_c a}$.
Therefore, the time complexity of the function is $O(n)$.

**Problem 2.g.** The big-$O$ time complexity of the following function, in terms of $n$.

```
void foo(int n){
    if (n <= 1)
        return;
    for (int i = 0; i < n; i++) {
        System.out.println("*");
    }
    foo(n/2);
    foo(n/2);
}
```

**Solution:** $O(n \log n)$

Similar to Problem 2f, we draw a recursion tree for the function. The recursion tree still has a height of $\log_2 n$. However, note that the number of iterations the for-loop in the function runs is different depending on which level the function call is on. For example, the for-loop in the initial function call runs for $n$ iterations, but the for-loops generated from the initial function call each run for $\frac{n}{2}$ iterations. In particular, the for-loop in each node on the $i$th level runs for $\frac{n}{2^{i-1}}$ iterations.



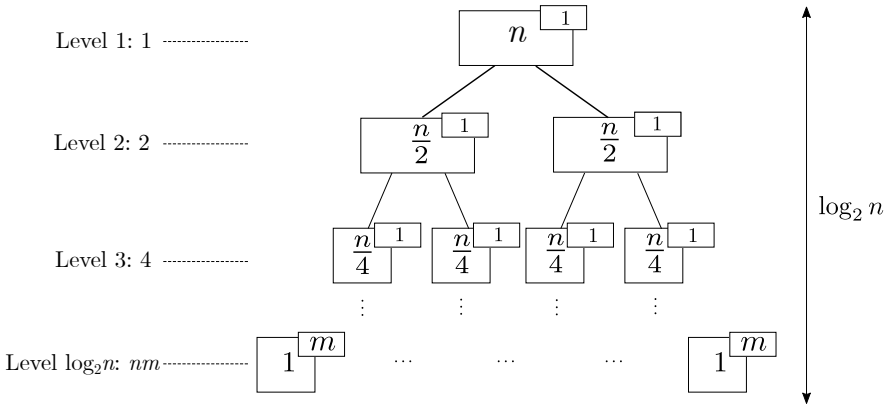

There are $2^{i-1}$ nodes on the $i$th level, so the total work done on the $i$th level is $2^{i-1} \cdot \frac{n}{2^{i-1}} = n$. As there are $\log_2 n$ levels in the recursion tree, the total work done across all levels is $n \cdot \log_2 n = O(n \log n)$.

**Problem 2.h.** The big-$O$ time complexity of the following function, in terms of $n$ and $m$.

```
void foo(int n, int m){
    if (n <= 1) {
        for (int i = 0; i < m; i++) {
            System.out.println("*");
        }
        return;
    }
    foo(n/2, m);
    foo(n/2, m);
}
```

**Solution:** $O(nm)$

We draw a recursion tree for the given function. The recursion tree looks similar to the one in Problem 2f.




Observe that the function is identical to the one in Problem 2f, except that in the base case of the recursion, we run a for-loop for $m$ iterations. This is reflected in the recursion tree above: every node in the last level performs $m$ work. Summing up the work done across all levels of the recursion tree, we have

$$\begin{aligned} 1 + 2 + \cdots + 2^{\log_2 n - 1} + m \cdot 2^{\log_2 n} &= \sum_{i=0}^{\log_2 n - 1} 2^i + nm \\ &= \frac{1 - 2^{\log_2 n}}{1 - 2} + nm \\ &= n - 1 + nm \\ &= O(nm) \end{aligned}$$

Therefore, the time complexity of the function is $O(nm)$.