# CS2040 Tutorial 1 Suggested Solution

Week 3, starting 22 Aug 2022

## *Q1 Big-O Notation*

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 < to indicate that the function on the left is upper-bounded by the function on the right, and == to indicate that two functions have the same big-O time complexity

| | | | |
|---|---|---|---|
| $4N^2$ | $\log_3 N$ | $20N$ | $N^{2.5}$ |
| $N^{0.00001}$ | $\log (N!)$ | $N^N$ | $2^N$ |
| $2^{N+1}$ | $2^{2N}$ | $3^N$ | $N \log N$ |
| $100N^{2/3}$ | $\log((\log N)^2)$ | $N!$ | $(N-1)!$ |

### *Answer*

Constant time < logarithmic time < fractional power (smaller than 1) time < linear time < linearthmic time < quadratic time < other polynomial (power more than 2) time < exponential time < factorial time

| | | | |
|---|---|---|---|
| $O(\log((\log N)^2))$ | $< O(\log_3 N)$ | $< O(N^{0.00001})$ | $< O(100N^{2/3})$ |
| $< O(20N)$ | $< O(\log (N!))$ | $== O(N \log N)$ | $< O(4N^2)$ |
| $< O(N^{2.5})$ | $< O(2^N)$ | $== O(2^{N+1})$ | $< O(3^N)$ |
| $< O(2^{2N})$ | $< O((N-1)!)$ | $< O(N!)$ | $< (N^N)$ |

Logarithmic functions are always bounded from above by a $N^k$ $(0 < k < 1)$ function. This can be proven by examining the ratio of the two terms as N tends to infinity, and using L'Hopital's rule (outside the scope of this module)

Why O(log (N!) == O(N log N):

$N^{N/2} < N! < N^N$ because $(N/2)^{N/2}2^{N/2} < N(N-1)(N-2)\ldots(N/2)\ldots(3)(2)(1) < NNN\ldots NNN$

Therefore, $(N/2) \log N < \log N! < N \log N$. This means NlogN and log(N!) bound each other from above

## *Q2 Time Complexity Analysis*

Find the tightest big-O time complexity of each of the following code fragments in terms of **n**

**a)**

```
for (int i = 1; i <= n; i++) {
   for (int j = 0; j < n; j++)
      System.out.print("*");
   System.out.println();
}
```

**b)**

```
for (int i = 1; i <= n; i++) {
   for (int j = 0; j < i; j++)
      System.out.print("*");
   System.out.println();
}
```

**c)**

```
for (int i = 1; i <= n; i *= 2) {
   for (int j = 0; j < i; j++)
      System.out.print("*");
   System.out.println();
}
```

**d)**

```
ArrayList<Integer> reverse(ArrayList<Integer> items) {
   ArrayList<Integer> copy = new ArrayList<>(); int n = items.size();
   for (int idx = 0; idx < n; idx++) copy.add(0, items.get(idx));
   return copy;
}
```

## *Answer*

**a)**

The number of times the inner loop runs is **independent** on the outer loop's control variable. We can simply factorize out the number of inner loop iterations that take place in outer loop iteration, and then multiply it back in after analysing the outer loop.

Outer loop passes: **n**
Inner loop passes: **n**

Every other operation runs in O(1) time. Therefore the time complexity of the code fragment is **O($n^2$)**

**b)**

Now, the number of times the inner loop runs is **dependent** on the outer loop's control variable, so we need to **carefully count the number of iterations** that take place. Analyzing the maximum number of passes in each loop independently does **NOT** always work

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

Every other operation runs in O(1) time. Therefore the time complexity of the code fragment is **O($n^2$)**

**c)**

Every other operation runs in O(1) time. Counting the number of inner loop iterations for every outer loop iteration:

$$1 + 2 + 4 + 8 + 16 \ldots + \frac{n}{2} + n = 2n - 1 = \boldsymbol{O(n)}$$

Therefore the time complexity of the code fragment is **O(n)**. This example shows that we can't just simply multiply the number of inner loop iterations with the number of outer loop iterations as the number of inner loop iterations is **dependent** on the outer loop control variable

**d)**

There are no nested loops in this part, but don't assume that every operation runs in O(1) time! Adding to the front of an ArrayList requires elements to be shifted backwards (from the last element down to the front). Adding to an ArrayList that would become size **i** requires O(**i**) time

Counting the number of inner loop iterations for every outer loop iteration:

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

## *Q3 Analyzing Recursive Algorithms*

For recursive problems, draw out the recursive tree. For each node in the tree:

- Identify how many calls are made directly from that node
- Identify how the **problem size** decreases each call
- Indicate how much **work is done per call**, aside from other recursive calls

Calls on the same level usually have similar problem sizes. Therefore, for each level:

- Calculate the work done per level
- Calculate the height of the tree if it helps you

These may help you to evaluate the Big-O time complexity quickly.

## *Worked Example*

```
public static long power(long x, long k, long M) {
   if (k == 0) return 1;
   long y = k / 2;
   if (2 * y == k) { // even power k
      long half = power(x, y, M); // (x^y) % M
      return half * half % M; // [(x^y % M)(x^y % M)] % M
   } else { // k == 2y + 1
      long next = power(x, 2 * y, M); // (x^2y) % M
      return x * next % M; // [x (x^2y % M)] % M
   }
}
```

Notice, x and M do not change, and are not useful to us. To find the next problem size, y may help.
Substituting k/2 for y and disregarding O(1) statements, the code can be **reduced** to:

```
void powerReduced(long k) {
    if (k == 0) return; // base case, to identify leaf nodes
    if (k % 2 == 0) powerReduced(k/2); // even k, recursive call
    else powerReduced(k-1); // odd k, recursive call
} // O(1) work done per call (aside from rec. calls)
```

Next, draw out the recursive tree (list, in this case, as only one call is made within another)

We here assume k is a power of 2 (e.g. 64, 1024, …)

| Level | Problem size | | # mtd calls in level | Work done per call | Work done in level |
|---|---|---|---|---|---|
| | Intuitive | Based on level | | | |
| 0 | k | $2^{\log(k)}$ | 1 | O(1) | O(1) |
| 1 | k/2 | $2^{\log(k)-1}$ | 1 | O(1) | O(1) |
| … | | | | | |
| h-1 | 2 | $2^1$ | 1 | O(1) | O(1) |
| Height h | 1 | $2^0$ | 1 | O(1) | O(1) |

How much work is done in total?

We can sum the work done in each level to get the answer.

However, since every level does O(1) work, we first need to find the height h, which is log(k)+1.

Therefore, time complexity is **O(log(k))**.

You may ask, what happens if k is not a power of 2?

The worst case occurs when k is a ((power of 2) -1).

Every call to powerReduced(x) results in a call to powerReduced(x-1) and powerReduced((x-1)/2)

Each call does O(1) work aside from other recursive calls

The height (length) of the list is 2log(k) + 1

Therefore, the time complexity is still O(log(k)).

**Your Turn!**

**a)**

```
boolean lookHere(ArrayList<Integer> items, int value) {
   int n = items.size();
   return lookHere(items, value, 0, n - 1);
}
boolean lookHere(ArrayList<Integer> items, int value, int low, int hi) {
   if (low > hi) return false;
   int mid = (low + hi) / 2;
   // do some O(1) stuff
   if (items.get(mid) > value)
      return lookHere(items, value, low, mid - 1);
   return lookHere(items, value, mid + 1, hi);
}
```

**b)**

```
void lookHere(ArrayList<Integer> items, int value) {
   int n = items.size();
   lookHere(items, value, 0, n - 1);
}
void lookHere(ArrayList<Integer> items, int value, int low, int hi) {
   if (low > hi) return;
   int mid = (low + hi) / 2;
   // do some O(1) stuff
   lookHere(items, value, low, mid - 1);
   lookHere(items, value, mid + 1, hi);
}
```

## *Answer*

**a)**

What algorithm this is similar to: Binary search with some parts omitted...

We are not concerned with items and value. Of course, we should first check that the code runs repeatedly and terminates, based on the values of low and hi. Our problem size is now (hi - low + 1).

```
void lookHereRec(int n) {
     if (n < 1) return;
     doOhOne();
     lookHereRec(n / 2); // condition removed, both branches combined
}
```

Next, draw out the recursive tree / list, assuming N is a power of 2 (e.g. 64, 1024, …)

| Level | Problem size | | # mtd calls in level | Work done per call | Work done in level |
|---|---|---|---|---|---|
| | Intuitive | Based on level | | | |
| 0 | n | $2^{\log(n)}$ | 1 | 1 | 1 |
| 1 | n/2 | $2^{\log(n)-1}$ | 1 | 1 | 1 |
| 2 | n/4 | $2^{\log(n)-2}$ | 1 | 1 | 1 |
| … | | | | | |
| h-2 | 2 | $2^1$ | 1 | 1 | 1 |
| h-1 | 1 | $2^0$ | 1 | 1 | 1 |
| Height h | 0 | 0 | 1 | 1 | 1 |

Height of the recursive list is log(**n**)+1

Adding the total work done across all levels gives log(**n**) + 2

Time complexity is **O(log(n))**. Even if **n** is not a power of 2, we have height of around 2log(**n**) = O(log(**n**)).

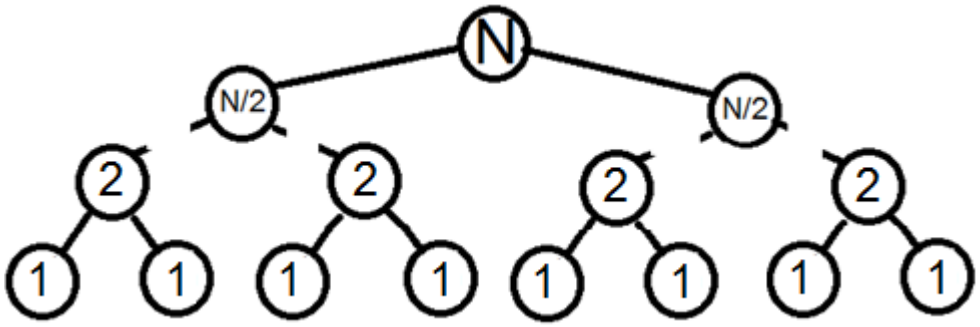
**b)**

What algorithm this is similar to: Traversal of a full binary tree…

Simplify the problem as we have been doing:

```
void lookHereRec2(int n) {
      if (n < 1) return;
      doOhOne();
      lookHereRec2(n / 2); lookHereRec2(n / 2);
}
```

If we examine the recursive calls that are made, we get a tree of calls with the problem size as each node. Remember, our objective is to find the sum of the **work done** by **all** recursive function calls




| Level | Problem size | | # mtd calls in level | Work done per call | Work done in level |
|---|---|---|---|---|---|
| | Intuitive | Based on level | | | |
| 0 | n | $2^{\log(n)}$ | 1 | 1 | 1 |
| 1 | n/2 | $2^{\log(n)-1}$ | 2 | 1 | 2 |
| 2 | n/4 | $2^{\log(n)-2}$ | 4 | 1 | 4 |
| … | | | | | |
| h-2 | 2 | $2^{1}$ | $2^{h-2}$ | 1 | $2^{h-2}$ |
| h-1 | 1 | $2^{0}$ | $2^{h-1}$ | 1 | $2^{h-1}$ |
| Height h | 0 | 0 | $2^{h}$ | 1 | $2^{h}$ |

h = log(**n**) + 1

Total work done forms a geometric series:

$$1 + 2 + 4 + \cdots + 2^{h-2} + 2^{h-1} + 2^{h} = 2^{h+1} - 1 = 2^{\log(n)+2} - 1 = \boldsymbol{O(n)} \neq O(N \log(N))$$

Common pitfalls:

- Multiplying tree height by initial problem size
- Forgetting that the function is recursive, n decreases at each lower level
- Adding problem size instead of amount of work done (number of operations)

### *Question 4 (Online Discussion) – Unangry Teams*

Answers for online discussion questions will NOT be given out. Your tutor may go through this question if there is time. Discuss these questions on piazza before/after the tutorial!

There are **N** people standing in a line and some of them are angry! You are given the array of the anger of these **N** people, `true` being angry. Find the number of groups of adjacent people in which no one angry is inside the group:

```
int numUnangryTeams(boolean[] angryPeople)
```

e.g. numUnangryTeams({T, F, F, F, T, F, F}) == 9

What is the time and space complexity of a brute force solution to solve this algorithm?
Can you think of any better solution than the brute force one?

There's an $O(N^3)$ brute forced algo, an $O(N^2)$ optimization and an O(N) optimization. Discuss on Piazza!