CS2040 Tutorial 1

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	$N \log N$
$100N^{2/3}$	$\log((\log N)^2)$	$N!$	$(N-1)!$

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;
}
```

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			
1	k	$2^{\log(k)}$	1	O(1)	O(1)
2	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);
}
```

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?