ICCS208: Assignment 5

Theeradon Sarawek
theeradon.sar@student.mahidol.edu
11 November 2024

1: Task 1: Hello, Definition

(I) Subtask I

To prove that $f(n) = n$ is $O(nlogn)$, we consider the definition of Big O, which states that $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{nlogn} \tag{1}$$

$$= \frac{1}{logn} \tag{2}$$

$$n \to \infty, log(n) \to \infty \tag{3}$$

$$\therefore = \frac{1}{\infty} = 0 \tag{4}$$

Which satisfies the Big O definition.

(II) Subtask II

If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then the product $d(n) \cdot e(n)$ is $O(f(n)g(n))$.

To prove this, we can take limits again. Consider that by definition,

$$\lim_{n\to\infty} \frac{d(n)}{f(n)} \to C_1 \quad \lim_{n\to\infty} \frac{e(n)}{g(n)} \to C_2 \qquad C_1, C_2 \in \mathbb{R}^+ \tag{5}$$

Therefore,

$$\lim_{n\to\infty} \frac{d(n) \cdot e(n)}{f(n) \cdot g(n)} = \left[\lim_{n\to\infty} \frac{d(n)}{f(n)}\right] \cdot \left[\lim_{n\to\infty} \frac{e(n)}{g(n)}\right] \tag{6}$$

$$= C_1 \cdot C_2 \tag{7}$$

Since we know both C_1 and C_2 are finite constants, we can make a new variable $C_3 = C_1C_2$ which fits the definition of Big O.

(III) Subtask III

Considering the following code (with comments),

```
void fnA(int S[]) {
    int n = S.length; // Theta(1)
    for (int i=0;i<n;i++) { // loops n times
        fnE(i, S[i]); // runtime -> 1000(i)
    }
}
```

We can ignore the part with constant runtime $\Theta(1)$ and consider the summation that is based on our for loop:

$$\sum_{i=0}^{n-1} 1000i = 1000 \cdot \sum_{i=0}^{n-1} i \tag{8}$$

$$= 1000 \cdot \frac{n(n-1)}{2} \tag{9}$$

Therefore, the total runtime is $\Theta(1000 \cdot \frac{n^2-n}{2})$. However, consider that the dominant factor n^2 will dominate for large values of n, therefore we can simplify this to $\Theta(n^2)$

(IV) Subtask IV

Show that $h(n) = 16n^2 + 11n^4 + 0.1n^5$ is not $O(n^4)$

Considering how Big O Notation works, it would make sense to only consider the dominant factor $0.1n^5$ which leaves us with a time complexity of $O(n^5)$

To cement that $h(n) \neq c \cdot n^4, c \in \mathbb{R}^+$ (per definition), we can take limits.

$$\lim_{n\to\infty} \frac{16n^2 + 11n^4 + 0.1n^5}{n^4} \tag{10}$$

$$= \lim_{n\to\infty} \frac{16n^2}{n^4} + \lim_{n\to\infty} \frac{11n^4}{n^4} + \lim_{n\to\infty} \frac{0.1n^5}{n^4} \tag{11}$$

$$= \lim_{n\to\infty} \frac{1}{16n^2} + \lim_{n\to\infty} 11 + \lim_{n\to\infty} 0.1 \tag{12}$$

$$= 0 + 11 + \infty \tag{13}$$

$$\to \infty \tag{14}$$

Since the limit is infinity, it does not meet the definition of Big O, which states that the limit must be some finite constant.

2: Task 2: Poisoned Wine

To implement a system to find the one poisoned wine using O(logn) testers, we shall implement a system using the binary representation of the i^{th} wine from 1 to n inclusive.

To demonstrate, assume $n = 16$ such that we will need $\lceil log_2(n+1) \rceil = 5$ bits to represent up to 16 in binary form. We must strictly use $log_2(16) = 4$ testers meaning we can assign each tester to the first four bits counting from the right. So let's label them:

-	T_4	T_3	T_2	T_1
0	0	0	0	0

(15)

Each tester T_j will only taste wine i (such that $1 \leq i \leq n$) if the binary representation of i has a 1-bit in the j^{th} column. To visualize this, say we have Wine 12, W_{12}, which can be visualized in binary form like so:

-	T_4	T_3	T_2	T_1
0	1	1	0	0

(16)

This means that T_4 and T_3 will try W_{12}. The key to finding out which wine is poisoned is to let all $log_2 n$ testers try all n wines and wait 31 days. We observe the testers to see which ones have been poisoned.

- Observe which testers have been poisoned and take count of their number j. We can then form a binary number from the poisoned tester(s) and place a 1 at their slot, leaving the slots of non-poisoned testers as 0.
- The number that is formed will be the bottle W_i that is poisoned.

! Notice how there isn't a T_5. The leftmost bit can only signify the largest possible representable number, which in this case would be W_{16}. If none of the testers are poisoned, then we can assume that the poisoned bottle is the only one that they haven't tried, which will be W_n.

To visualize this, suppose we notice that after 31 days, Tester 4, 3, and 1 are poisoned. Putting this into the binary table would leave us with:

-	T_4	T_3	T_2	T_1
0	1	1	0	1

(17)

The number that is formed would then be $8 + 4 + 1 = 13$, therefore we can conclude that W_{13} is the poisoned bottle (unlucky).

3: Task 3: How Long Does This Take?

```
void programA(int n) {
    long prod = 1; // Theta(1)
    for (int c=n;c>0;c=c/2) // log2(n) * 1
        prod = prod * c; // Theta(1)
}
```

From observation alone, I'll deduce that this likely has a logarithmic runtime due to the nature of the for loop, which reduces c by 2 until $c <= 0$, which considering how Java uses floor division, will terminate if it has to divide 1 by 2. Therefore, the number of times the function will loop would be $\lfloor log_2(n) \rfloor + 1$ (includes when Java decides to multiply by one), which simplifies to $log_2(n)$ in the grand scheme of things. We also consider the multiplication operation inside the loop will take constant time $\Theta(1)$.

Therefore, the total runtime will be $\Theta(1) + log_2(n) \cdot \Theta(1)$ which simplifies to $\Theta(logn)$

```
void programB(int n) {
    long prod = 1;
    for (int c=1;c<n;c=c*3)
        prod = prod * c;
}
```

Similar to *programA*, this should be logarithmic. The amount of times the for loop initiates would be $log_3(n)$ times, considering it loops x times based on how many times 3 can be multiplied to get a value that is at least n.

The code is identical to *programA* except for how the for loop iterates, therefore we can reuse what we found previously.

Therefore, the total runtime will be $\Theta(1) + log_3(n) \cdot \Theta(1)$ which is just $\Theta(logn)$

4: Task 4: Halving Sum

(I) Part I

To help me understand it better, here's my Python writeup I used based on the instructions inside the while loop. This is based on the idea that $len(x)$ must be some exponent of 2 so there would be no need to compensate for dividing odd numbers by 2.

```
def hsum(x: List[int]):
    while len(x) > 1:
        y = [0] * int(len(x) / 2) // step 1
        for i in range(int(len(x)/2)): // step 2
            y[i] = x[2*i] + x[(2*i) + 1] // step 2
        x = y // step 3
    return x[0]
```

- Step 1 will take $k_1 \cdot \frac{z}{2}$ since we allocate an array of size $\frac{z}{2}$
- Step 2 will take $7k_2 \cdot \frac{z}{2}$ as we will loop $\frac{z}{2}$ times doing seven operations that cost k_2 each:
  - Multiplying $2 \cdot i$
  - Retrieving element $2 \cdot i$ in X
  - Multiplying $2 \cdot i$
  - Adding 1
  - Retrieving element $2 \cdot i + 1$ in X
  - Adding those two elements together
  - Writing to the other array Y
- Step 3 takes $\frac{zk_2}{2}$ due to it being an array writing operation that we do $\frac{z}{2}$ times.

We can therefore write this as $k_1 \cdot \frac{z}{2} + 7k_2 \cdot \frac{z}{2} + k_2 \cdot \frac{z}{2}$ or simply $(\frac{k_1}{2} + 4k_2) \cdot z$ if we want to adhere as closely as we can to the format given in the assignment.

(II) Part II

We know that z is variable starting from the original $n = X.length$ that is divided by 2 until it yields 1. Therefore, our summation for runtime might look something along the lines of:

$$\left[(\frac{k_1}{2} + 4k_2) \cdot \frac{n}{2^1}\right] + \left[(\frac{k_1}{2} + 4k_2) \cdot \frac{n}{2^2}\right] + ... + \left[(\frac{k_1}{2} + 4k_2) \cdot \frac{n}{2^{log_2 n}}\right] \tag{18}$$

Where $\frac{n}{2^{log_2 n}}$ should just be 1. This means we can write a summation formula:

$$\sum_{i=1}^{log_2 n} \left[(\frac{k_1}{2} + 4k_2) \cdot \frac{n}{2^i}\right] \tag{19}$$

$$(\frac{k_1}{2} + 4k_2) \cdot \sum_{i=1}^{log_2 n} \frac{n}{2^i} \tag{20}$$

Gonna dissect the first summation separately since it looks like it will take a while. Assume $C = (\frac{k_1}{2} + 4k_2)$ for readability since it's a constant.

$$C \cdot n \cdot \sum_{i=1}^{log_2 n} \frac{1}{2^i} \tag{21}$$

$$= C \cdot n \cdot \left[\frac{1}{2} + \frac{1}{2^2} + ... + \frac{1}{2^{log_2 n}}\right] \tag{22}$$

Notice that this is a geometric sequence with $a = \frac{1}{2}$ and $r = 2^{-1}$ so we can use the geometric sum formula + consider that $2^{log_2 n} = n$

$$S = \frac{1}{2} \cdot \frac{1 - 2^{-log_2 n}}{1 - 2^{-1}} \tag{23}$$

$$= \frac{1}{2} \cdot 2 \cdot 1 - \frac{1}{n} \tag{24}$$

$$= 1 - \frac{1}{n} \tag{25}$$

Therefore, this part is:

$$C \cdot n \cdot (1 - \frac{1}{n}) \tag{26}$$

$$= (\frac{k_1}{2} + 4k_2) \cdot n \cdot (1 - \frac{1}{n}) \tag{27}$$

This means the total runtime as a summation would be:

$$hsum_{RT} = (\frac{k_1}{2} + 4k_2) \cdot (n - 1) \tag{28}$$

Which, considering only the dominant factor n in the sea of constants the final sum leaves us, leaves us with $\Theta(n)$ such that n is based on the length of input array x. Note that this is the runtime of my Python code, which I tried to base as close to the prompt as possible, so if the code is wrong then... :(

5: Task 5: More Running Time Analysis

(I) Part I

```
static void method1(int[] array) {
    int n = array.length; // O(1) constant
    for (int index=0;index<n-1;index++) { // n - 1 iterations
        int marker = helperMethod1(array, index, n - 1); // O(n -
            index)
        swap(array, marker, index); // O(1)
    }
}
static void swap(int[] array, int i, int j) { // O(1) constant
    int temp=array[i];
    array[i]=array[j];
    array[j]=temp;
```

```
}

static int helperMethod1(int[] array, int first, int last) { //
   O(z) such that z = |last-first| so basically O(n) anyway
    int max = array[first]; // O(1)
    int indexOfMax = first; // O(1)
    for (int i=last;i>first;i--) { // (last-first) * O(1)  = O(z)
        if (array[i] > max) { // O(1)
            max = array[i]; // O(1)
            indexOfMax = i; // O(1)
        }
    }
    return indexOfMax;
}
```

The runtime of *method*1 depends on *helperMethod*1 since *swap* takes constant runtime.

Each iteration will call helperMethod1 which scans through an array of size $n-index$ each time such that index goes from 0 to $n-1$. This means the total runtime should look something like

$$Runtime = O(n) + O(n-1) + O(n-2) + ... + O(1) = \frac{n(n-1)}{2} = O(n^2) \tag{29}$$

Considering that the for loop will loop $n-1$ times, calling *helperMethod*1 to scan through all of it (without some form of loop break) regardless of the order of size n inputs, the best and worst cases should both be the same at $O(n^2)$

(II) Part II

```
static boolean method2(int[] array, int key) {
    int n = array.length;
    for (int index=0;index<n;index++) {
        if (array[index] == key) return true;
            }
    return false;
}
```

*method*2 is not slick in hiding the fact that it is a linear search. So,

- The best case scenario for input size n is that the key we want to find just so happens to be in the first index. This means the best-case runtime will be $O(1)$
- The worst case scenario for input size n is that we need to traverse the entire array. This means worst-case runtime will be $O(n)$.

(III) Part III

```
static double method3(int[] array) {
    int n = array.length; // O(1)
    double sum = 0; // O(1)
    for (int pass=100; pass >= 4; pass--) { // 97 times -> Assume
       O(1) since it's constant
```

```
        for (int index=0;index < 2*n;index++) { // O(2n) -> Assume
            O(n)
            for (int count=4*n;count>0;count/=2) // O(log(n))
                sum += 1.0*array[index/2]/count; // O(1)
        }
    }
    return sum;
}
```

From the comments, we see total runtime should be $O(1) \cdot O(n) \cdot O(logn)$ which is $O(nlogn)$.

- The best and worst case runtime should be the same since the for loops will loop regardless of order.

6: Task 6: Recursive Code

I wish I saw this before I wrote my code in Task 4 :(

```
int halvingSum(int[] xs) {
    if (xs.length == 1) return xs[0];
    else {
        int[] ys = new int[xs.length/2];
        for (int i=0;i<ys.length;i++) // O(n/2) --> O(n)
            ys[i] = xs[2*i]+xs[2*i+1];
        return halvingSum(ys); // recursive call where len(ys) =
            len(xs)/2 therefore T(n/2)
    }
}
```

1. Input size depends on the length of array xs.
2. T(n) = T(n/2) + O(n),T(1) = 1
3. O(n)

```
int anotherSum(int[] xs) {
    if (xs.length == 1) return xs[0];
    else {
        int[] ys = Arrays.copyOfRange(xs, 1, xs.length); // O(n -
            1) --> O(n)
        return xs[0]+anotherSum(ys); // T(n - 1)
    }
}
```

1. Input size depends on the length of array xs.
2. T(n) = T(n-1) + O(n), T(1) = 1

3. O(n^2)

```
int[] prefixSum(int[] xs) {
    if (xs.length == 1) return xs;
    else {
        int n = xs.length;
        int[] left = Arrays.copyOfRange(xs, 0, n/2); // O(n/2)
        left = prefixSum(left); T(n/2)
        int[] right = Arrays.copyOfRange(xs, n/2, n); // O(n/2)
        right = prefixSum(right); T(n/2)
        int[] ps = new int[xs.length];
        int halfSum = left[left.length-1];
        for (int i=0;i<n/2;i++) { ps[i] = left[i]; } // O(n/2)
        for (int i=n/2;i<n;i++) { ps[i] = right[i - n/2] + halfSum;
           } // O(n/2)
        return ps;
    }
}
// In the grand scheme of things all the O(n/2) turns into O(n)
```

1. Input size depends on the length of array xs.
2. T(n) = 2T(n/2) + O(n)
3. O(nlogn)

7: Task 7: Counting Dashes

```
void printRuler(int n) {
    if (n > 0) {
        printRuler(n-1);
        // print n dashes
        for (int i=0;i<n;i++) System.out.print('-');
        System.out.println();
        // ----------------
        printRuler(n-1);
    }
}
```

Consider the following, where g(n) is printRuler and f(n) is the Tower of Hanoi sequence.

$$g(n) = 2g(n-1) + n, g(0) = 0 \tag{30}$$

$$f(n) = 2f(n-1) + 1, f(0) = 0 = 2^n - 1 \tag{31}$$

The relationship between them can be *guessed* as, where we will call this Equation 1:

$$g(n) = a \cdot f(n) + b \cdot n + c \tag{32}$$

To prove that this is true,

(I) Plugging $n = 0$ into Equation 1 would mean:

$$g(0) = a \cdot f(0) + b \cdot n + c \tag{33}$$
$$0 = 0 + 0 + c \tag{34}$$
$$\therefore c = 0 \tag{35}$$

(II) Plugging $g(n) = a \cdot f(n) + b \cdot n$ into the recurrence $g(n)$ would lead to:

$$g(n) = 2 \cdot [a \cdot f(n-1) + b \cdot (n-1)] + n \tag{36}$$
$$= 2 \cdot a \cdot f(n-1) + 2 \cdot b \cdot (n-1) + n \tag{37}$$

We can express $f(n-1)$ as $\frac{f(n)-1}{2}$ so,

$$g(n) = 2 \cdot a \cdot \frac{f(n)-1}{2} + 2 \cdot b \cdot (n-1) + n \tag{38}$$
$$= a \cdot f(n) - a + 2bn - 2b + n \tag{39}$$
$$= a \cdot f(n) + 2bn + n - a - 2b \tag{40}$$
$$= a \cdot f(n) + (2b+1) \cdot n - (a + 2b) \tag{41}$$

Which is in the same formula as the guess. Now, comparing with the guess,

- a = a
- $2b + 1 = b$, therefore $1 = -b$ so $b = -1$
- $c = a + 2b$, we established $c = 0$ so $a + 2b = 0$ therefore $a = -2b$
- Which therefore means $a = (-1)(-2) = 2$

Therefore, we say $g(n) = 2f(n) - n$

(III)

The closed form formula should look like g(n) = $2(2^n - 1) - n = 2^{n+1} - (n + 2)$

(IV)

We want to prove that $g(n) = g'(n)$ (note that the aprostophe does not mean derivative) where:

$$g(n) = 2g(n-1) + n \tag{42}$$
$$g'(n) = 2^{n+1} - (n+2) \tag{43}$$

Let's assume those are the inductive hypotheses.

Base case: g(0) = 0, $g'(0) = 2^{0+1} - (2 + 0) = 0$

Inductive step:

$$g(n+1) = 2(g(n)) + n + 1 \tag{44}$$
$$IH \rightarrow 2\left[2^{n+1} - (n+2)\right] + n + 1 \tag{45}$$
$$= 2^{n+2} - 2n - 4 + n + 1 \tag{46}$$
$$= 2^{n+2} - n - 3 \tag{47}$$
$$= 2^{[n+1]+1} - ([n+1] + 2) \tag{48}$$

Therefore, per induction, we know that $g(n) = g'(n)$ meaning that our closed form formula actually works.