ICCS208: Assignment 5

Chanat Keratiyutwong
chanat.ker@student.mahidol.edu
June 11, 2022

1: Hello, Definition

---

(1) Using the definition of Big-O, $f(n) \in O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

We have $f(n) = n$ and $g(n) = n \cdot log(n)$.

Verify the limit above.

$$\lim_{n\to\infty} \frac{n}{n \cdot log(n))} = \lim_{n\to\infty} \frac{1}{log(n)} = 0 < \infty$$

Therefore, $n \in O(n \cdot log(n))$.

(2) Using the definition of Big-O, $d(n) = O(f(n))$and $e(n) = O(g(n))$, we have $\lim_{n\to\infty} \frac{d(n)}{f(n)} < \infty$ and $\lim_{n\to\infty} \frac{e(n)}{g(n)} < \infty$.

$$\begin{aligned} d(n) \cdot e(n) &= O(f(n)) \cdot O(g(n)) \\ &= \lim_{n\to\infty} \frac{d(n)}{f(n)} \cdot \lim_{n\to\infty} \frac{e(n)}{g(n)} < \infty \\ &= \lim_{n\to\infty} \frac{d(n)e(n)}{f(n)g(n)} < \infty \end{aligned}$$

Therefore, $d(n) \cdot e(n) \in O(f(n)g(n))$.

(3)

```
void fnA(int S[]) {
    int n = S.length;
    for (int i=0;i < n;i++) {
        fnE(i, S[i]);
    }
}
```

i) determining length of array = $O(1)$
ii) for loop - linear increments (add by 1) = $O(n)$
iii) inside for loop - fnE always runs $1000i$ steps = $O(1000n) = O(n)$

The total running time for fnA is $O(1) + O(n) \cdot O(n) = O(n^2)$.

(4)

Show that $h(n) = 16n^2 + 11n^4 + 0.1n^5 \notin O(n^4)$.

Using the definition of Big-O, $h(n) \in O(n^4)$ if

$$\lim_{n\to\infty} \frac{h(n)}{n^4} < \infty.$$

We take the limit and verify this condition.

$$\begin{aligned}
\lim_{n\to\infty} \frac{h(n)}{n^4} &= \lim_{n\to\infty} \frac{16n^2 + 11n^4 + 0.1n^5}{n^4} \\
&= \lim_{n\to\infty} \left(\frac{16}{n^2} + 11 + 0.1n\right) \\
&= 0 + 11 + \infty \\
&= \infty
\end{aligned}$$

Because the evaluated limit is not finite (goes to ∞), $h(n) \notin O(n^4)$.

2: Poisoned Wine

1) Assign each bottle to bits: 1, 2, 3, ... , n
2) Assign each tester a number 1, 2, 3, ... , log(n) 3) Convert each bit into binary form. The number of bits for each binary number will be log(n) as we have log(n) testers. For example:

- 1 = ...00000001
- 2 = ...00000010
- 3 = ...00000011
- 4 = ...00000100
- 5 = ...00000101
- 6 = ...00000110

4) The bits in a binary number will determine which testers will test a specific bottle. For example, if we have 8 testers and the 5^{th} bottle is the poisonous bottle:

- bottle 5 will be represented as 00000101 in binary form.
- Wherever the 1-bits are in the binary number will indicate which testers will test that bottle.
- In this case, 1-bits are on the 6^{th} bit and the 8^{th} bit (from the left), so we will serve wine from this specific bottle to the 6^{th} and the 8^{th} testers.
- If the 6^{th} and the 8^{th} testers are the only testers to experience hysteria after a month, the 5^{th} is the poisonous bottle.

To generalize this example, these numbered bottles represented in binary forms determine which testers will test a specific bottle, depending on the positions of the 1-bits in the binary representations. After a month (30 days), we check to see which testers are experiencing hysteria and use that combination of diagnosed testers to match a bottle's binary representation depending on the positions of the 1-bits (remember that we also numbered the testers).

This scheme meets the $O(log(n))$-tester requirement as we are using binary representations to label each wine bottle, so the number of testers $= log_2(n) + 1$ (using the idea of how a decimal number (base 10) is represented as a binary number (base 2)) where n is the number of wine bottles.

3: How Long Does This Take?

---

(1)

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

i) assignment of prod = $\Theta(1)$
ii) multiplication inside loop = $\Theta(1)$
iii) for loop - loop variable is halved = $\Theta(log_2(n))$

So the running time of programA is $\Theta(log(n))$.

(2)

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

i) assignment of prod = $\Theta(1)$
ii) multiplication inside loop = $\Theta(1)$
iii) for loop - multiplication to update variable = $\Theta(n \cdot log_3(n))$

So the running time of programB is $\Theta(n \cdot log(n))$.

4: Halving Sum

```
def hsum(X): # assume len(X) is a power of two
    while len(X) > 1:
        (1) allocate Y as an array of length len(X)/2
        (2) fill in Y so that Y[i] = X[2i] + X[2i+1]
        for i = 0, 1, ..., len(X)/2 - 1
        (3) X = Y
    return X[0]
```

(1)

For each iteration:

i) allocating array Y - $\frac{k_1}{2} \cdot z$
ii) filling in array Y

- accessing X[2i] = k_2
- accessing X[2i + 1] = k_2
- adding X[2i] and X[2i + 1] together = k_2
- setting that sum = Y[i] = k_2
- setting array X = array Y = k_2

So the total time to fill in array Y is: $5k_2$.

Work done in each iteration: $f(z) = \frac{k_1}{2} \cdot z + 5k_2$**.**

(2)

The length of the input array (assumed to be a power of 2) will be halved each time until the length is 1, in which case it will return just the element inside that array, taking constant time to do so.

For each iteration, reading and writing into a new array Y will always take constant time ($5k_2$), so for an array X which has a very large size, the amount of time to read and write into another array will be insignificant in comparison to the the amount of time it takes to allocate an array X ($k_1 \cdot z$). Because of this, we will ignore the time it takes to read and write into an array (constant term), and only consider the running time based on the allocation of the array, at the start of each iteration.

Allocating an array has a running time of $O(n)$, which in this case is $k_1 \cdot z$, with k_1 and z representing a positive constant integer and the length of an array respectively. For simplicity we will let $k_1 = 1$, so that we can analyze the running time based on the array size z itself.

Because the while loop will run as long as $z > 1$, running times of array sizes larger than 1 will add the running times of all the iterations, as follows:

z	Running time (ignoring reading and writing time and letting $k_1 = 1$)
1	1
2	2 + 1
4	4 + 2 + 1
8	8 + 4 + 2 + 1
16	16 + 8 + 4 + 2 + 1
32	32 + 16 + 8 + 4 + 2 + 1
64	64 + 32 + 16 + 8 + 4 + 2 + 1
...	...
n	$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + 1$

From this table, we see that for array of size z, the running time is a geometric sum with 2 as the common ratio, and $log_2(z) - 1$ as the exponent (because z is a power of 2). We can write this as a function $g(n)$ (for the sake of generalization, $z = n$) as the summation below:

$$g(n) = \sum_{i=0}^{log_2(n)-1} 2^i$$

Using the geometric sum formula $S(n) = \frac{a_0(r^n-1)}{r-1}$ (r = common ratio, n = number of terms), we can simplify this function further, as shown below.

$$\begin{aligned} g(n) &= \sum_{i=0}^{log_2(n)-1} 2^i \\ &= \frac{1 \cdot (2^{log_2(n)-1} - 1)}{2-1} \\ &= 2^{log_2(n)-1} - 1 \\ &= \frac{2^{log_2(n)}}{2} - 1 \\ g(n) &= \frac{n}{2} - 1 \end{aligned}$$

Therefore, $g(n) \in \Theta(n)$, meaning that the code will take an amount of time proportional to the array size of n (linear time complexity).

5: More Running Time Analysis

---

(1)

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

static void swap(int[] array, int i, int j) {
    int temp=array[i];
    array[i]=array[j];
    array[j]=temp;
}

static int helperMethod1(int[] array, int first, int last) {
    int max = array[first];
    int indexOfMax = first;
    for (int i=last;i>first;i--) {
        if (array[i] > max) {
            max = array[i];
            indexOfMax = i;
        }
    }
    return indexOfMax;
}
```

Consider each helper methods' running times:

- swap - $\Theta(1)$ (initializing values and assigning vales to indices in array).
- helperMethod1 - $\Theta(n)$ (for loop with loop variable updating by +1, with all the other lines inside and outside the for loop taking constant time to execute)

Breaking down method1:
i) declaring and initializing int n - $\Theta(1)$
ii) for loop - linear (loop variable updates by +1) - $\Theta(n)$
iii) inside for loop - helpermethod1 and swap - $\Theta(n)$

The total running time of method1 is $\Theta(n^2)$.

Best-case: $\Theta(n)$ - this occurs if helperMethod1's "first" and "last" integer inputs are equal, which will prevent the for loop in that method from running, making helperMethod1 run in constant time.

Worst-case: $\Theta(n^2)$ - occurs if "first" and "last" inputs in helperMethod1 are not equal, so the for loop will run as normal.

(2)

```
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;
}
```

i) assigning n to be the value of the length of the array, return at the end - $\Theta(1)$
ii) for loop - linear search - $\Theta(n)$
iii) inside for loop - boolean + return - $\Theta(1)$

Total running time of method2 = $\Theta(n)$

Best-case: $\Theta(1)$ (the key is immediately found at the first index - returns true instantly).

Worst-case: $\Theta(n)$ (having to run through the array only to find that nothing in the array matches the key, thereby returning false at the end)

(3)

```
static double method3(int[] array) {
    int n = array.length;
    double sum = 0;
    for (int pass=100; pass >= 4; pass--) {
        for (int index=0;index < 2*n;index++) {
            for (int count=4*n;count>0;count/=2)
                sum += 1.0*array[index/2]/count;
        }
    }
    return sum;
}
```

fixed # of iterations (97)

i) Assigning values for int n and double sum, return at the end - $\Theta(1)$
ii) for loop (pass) - ~~linear - $\Theta(n)$~~ constant - $\Theta(1)$, as the number of iterations is fixed.

iii) for loop (index) - linear - $\Theta(n)$.
iv) for loop (count) - binary - $\Theta(log_2(n))$

Total running time = $\Theta(\cancel{n^2 log(n)})$ $n \log_2(n)$

Best-case: In the case where the array is empty ($n = 0$), only the outermost for loop (pass) will keep running as it does not depend on n, which is the length of the array. The outer for loop (pass) has a running time of $\cancel{O(n)}$, and added on to the other lines that take constant time, the best case running time $\cancel{\text{is } O(n)}$. $\theta(1)$ $\theta(1)$

Worst-case: The array is not empty, and there are no conditions to instantly break each loop while it is still running, so the worst case running time is $\Theta(\cancel{n^2 log_2(n)})$.
$n \log_2(n)$

6: Recursive Code

(1)

```
// assume xs.length is a power of 2
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++)
            ys[i] = xs[2*i]+xs[2*i+1];
        return halvingSum(ys);
    }
}
```

i) The input size is measured by the length of array xs, which also defines the length of the array ys.

ii) Breaking each line down:

- xs.length == 1 - Boolean takes constant time - $O(1)$.
- initializing int[] ys - linear - $O(n)$.
- the whole for loop - linear - $O(n)$
  - for loop - loop variable updates by 1 - linear - $O(n)$.
  - setting a value to an index of array ys - constant time - $O(1)$.
- recursion with array ys, which is half the length of array xs - $T(\frac{n}{2})$.

Putting everything together, the total running time of halvingSum is:

$$T(n) = O(1) + O(n) + O(n) + T(\frac{n}{2})$$
$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

iii) This recurrence relation solves to $O(n)$.

(2)

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

i) the input size is determined by the length of the array xs.
ii) Breaking down each line:

- base case - Boolean of checking array length - constant time - $O(1)$.
- initializing array ys by using Arrays.copyOfRange(...) - $O(n)$.
- recursion with array ys, whose length is 1 shorter than the input array - $T(n-1)$.

Putting everything together, we have:

$$T(n) = O(1) + O(n) + T(n-1)$$
$$T(n) = T(n-1) + O(n)$$

iii) This recurrence relation solves to $O(n^2)$.

(3)

```
int[] prefixSum(int[] xs) {
    if (xs.length == 1) return xs;
    else {
        int n = xs.length;
        int[] left = Arrays.copyOfRange(xs, 0, n/2);
        left = prefixSum(left);
        int[] right = Arrays.copyOfRange(xs, n/2, n);
        right = prefixSum(right);
        int[] ps = new int[xs.length];
        int halfSum = left[left.length-1];
        for (int i=0;i<n/2;i++) {
            ps[i] = left[i];
        }
        for (int i=n/2;i<n;i++) {
            ps[i] = right[i - n/2] + halfSum;
        }
        return ps;
    }
}
```

i) the input size is measured by the length of the array xs.
ii) Breaking down the code method by method:

- base case (boolean) - constant time - $O(1)$.
- initializing int n - constant time - $O(1)$.
- initializing arrays "left" and "right" - both use Arrays.copyOfRange(...) - $O(n)$ each.

- recursive calls prefixSum(left) and prefixSum(right) - both arrays are essentially half the size of the input array - $T(\frac{n}{2})$ each.
- initializing array ps - $O(n)$.
- int halfSum - constant time - $O(1)$.
- for loop 1 - assigning values to indices in the array, while loop variable updates linearly (+1) - $O(n).$
- for loop 2 - follows on from where for loop 1 left off, and fills the rest of the array - $O(n)$.

Putting everything together, we have:

$$T(n) = O(1) + O(1) + O(n) + T\left(\frac{n}{2}\right) + O(n) + T\left(\frac{n}{2}\right) + O(n) + O(1) + O(n) + O(n)$$
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

iii) This recurrence relation solves to $O(n \cdot log(n))$.

7: Counting Dashes

(i) Given that $g(0) = f(0) = 0$, plugging n = 0 into equation 1 gives us,

$$\begin{aligned} g(n) &= a \cdot f(n) + b \cdot n + c \\ g(0) &= a \cdot f(0) + b \cdot 0 + c \\ 0 &= a \cdot 0 + c \\ 0 &= c \end{aligned}$$

(ii) Plugging in $g(n)$ from equation 1 into $g(n) = 2g(n-1) + n$, we have

$$\begin{aligned} a \cdot f(n) + bn &= 2[af(n-1) + b(n-1)] + n \\ a \cdot f(n) + bn &= 2a \cdot f(n-1) + 2bn - 2b + n \\ a \cdot f(n) - 2a \cdot f(n-1) &= bn - 2b + n \\ a \cdot [f(n) - 2f(n-1)] &= bn - 2b + n \end{aligned}$$

Because $f(n) = 2f(n-1) + 1$, we know that $f(n) - 2f(n-1) = 1$.

$$\begin{aligned} a &= bn - 2b + n \\ a + 2b - bn - n &= 0 \\ -(b+1)n + (a+2b) &= 0 \end{aligned}$$

We have $P = -(b+1)$ and $Q = a + 2b$. We now solve for a and b such that $P = 0$ and $Q = 0$.

$$\begin{cases} -b - 1 = 0 \\ a + 2b = 0 \end{cases}$$

From the first equation, $b = -1$. Substituting that into the second equation, we get $a = 2$.

(iii) Given that $f(n)$ has a closed form of $2^n - 1$, and we've solved for a and b that $a = 2$ and $b = -1$, our closed form of $g(n)$ is:

$$\begin{aligned} g(n) &= a \cdot f(n) + b \cdot n + c \\ &= 2 \cdot (2^n - 1) - n + 0 \\ &= 2 \cdot (2^n) - n - 2 \\ g(n) &= 2^{n+1} - n - 2 \end{aligned}$$

(iv) Proof by induction:

- Inductive Predicate: $P(i) \equiv g(i) = 2^{i+1} - i - 2 \ \forall i \geq 0$.
- Base Case: $P(0) \equiv g(0) = 2^{0+1} - 0 - 2 = 0$.
  So $P(0)$ is true.
- Inductive Hypothesis: Assume that $P(k) \equiv g(k) = 2^{k+1} - k - 2 \ \forall k \geq 0$.
- Inductive Step:

  We want to show that $g(k+1) = 2^{k+2} - k - 3$.

  Given the recurrence $g(n) = g(n-1) + n$, we substitute our closed form for $g(n)$ where $n = k + 1$:

$$\begin{aligned} \text{LHS: } g(n) &= g(k+1) \\ &= 2^{k+2} - k - 3 \end{aligned}$$

$$\text{RHS: } 2g(n-1) + n = 2g(k) + (k+1)$$

$$\begin{aligned} \text{Using the inductive hypothesis: } g(i) &= 2^{i+1} - i - 2 \\ &= 2[2^{k+1} - k - 2] + k + 1 \\ &= 2^{k+2} - 2k - 4 + k + 1 \\ &= 2^{k+2} - k - 3 \\ &= \text{LHS} \end{aligned}$$

Therefore, by mathematical induction, $g(n) = 2^{n+1} - n - 2 \ \forall n \geq 0$.