Homework1 for EECS 340

Yu Mi

January 29, 2018

1 Warm-up: Big-Oh and Counting Primitive Operations

Show your work on the following questions. Use the limit-based definitions of asymptotic notation on the "Big-Oh Cheat Sheet" on Canvas wherever applicable.

1.1 Solve R-1.20, R-1.22, and R-1.23 in the text

1.1.1 R-1.20

Show that $(n+1)^5$ is $O(n^5)$.

Proof: Let $f(n) = (n+1)^5$ and $g(n) = n^5$ so that

$$\begin{aligned}\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{(n+1)^5}{n^5} &= \lim_{n\to\infty}\frac{n^5+5n^4+10n^3+10n^2+5n+1}{n^5}\\ &= 1 + \lim_{n\to\infty}\frac{5}{n}+\frac{10}{n^2}+\frac{10}{n^3}+\frac{5}{n^4}+\frac{1}{n^5} = 1\end{aligned}$$

Since $0 \le 1 < \infty$, $(n+1)^5$ is $O(n^5)$.

1.1.2 R-1.22

Show that n is $o(n\log n)$.

Proof: Let $f(n) = n$ and $g(n) = n\log n$ so that

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{n}{n\log n} = \lim_{n\to\infty}\frac{1}{\log n} = 0$$

Since $0 = 0$, n is $o(n\log n)$.

1.1.3 R-1.23

Show that n^2 is $\omega(n)$.

Proof: Let $f(n) = n^2$ and $g(n) = n$ so that

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{n^2}{n} = \lim_{n\to\infty} n = \infty$$

Since $\infty = \infty$, n^2 is $\omega(n)$.

1.2 Intuitively, $2^x \in O(3^x)$, since 3^x grows faster. Is $3^x \in O(2^x)$?

Answer: No, proof as follows:

Let $f(x) = 3^x$ and $g(x) = 2^x$ so that

$$\lim_{x\to\infty}\frac{f(x)}{g(x)} = \lim_{x\to\infty}\frac{3^x}{2^x} = \lim_{x\to\infty}(\frac{3}{2})^x = \infty$$

Since $\infty = \infty$, so that 3^x is $\omega(2^x)$, $3^x \notin O(2^x)$

1.3 Intuitively, $\log_3(x) \in O(\log_2(x))$. Is $\log_2(x) \in O(\log_3(x))$?

Answer: Yes, proof as follows:

Let $f(x) = \log_2(x)$ and $g(x) = \log_3(x)$ so that

$$\lim_{x\to\infty} \frac{\log_2(x)}{\log_3(x)} = \lim_{x\to\infty} \frac{\frac{\ln x}{\ln 2}}{\frac{\ln x}{\ln 3}} = \lim_{x\to\infty} \frac{\ln 3}{\ln 2} = \log_2(3)$$

Since $0 \leq \log_2(3) < \infty$, $\log_2(x)$ is $O(\log_3(x))$.

1.4 Use summations to derive tight asymptotic bounds $(\Theta(-))$ on the runtime of each algorithm

1.4.1 R-1.12

Answer: First, we need to rewrite this algorithm into *while* loop:

Algorithm Loop2(n):

$p \leftarrow 1$ ▷ 1 unit of time
$i \leftarrow 1$ ▷ 1 unit of time
while $i \leq 2n$ **do** ▷ $2n + 1$ units of time
  $p \leftarrow p \times i$ ▷ $2 \times 2n$ units of time
  $i \leftarrow i + 1$ ▷ $2 \times 2n$ units of time
end while

As is described in the comments of the algorithm, the run time of this algorithm should be $\Theta(10n + 3)$ units of time. So that this algorithm is $\Theta(n)$

1.4.2 R-1.14

Answer: First, we need to rewrite this algorithm into *while* loop:

Algorithm Loop4(n):

$s \leftarrow 0$ ▷ 1 unit of time
$i \leftarrow 1$ ▷ 1 unit of time
while $i \leq 2n$ **do** ▷ $2n + 1$ units of time
  $j \leftarrow 1$ ▷ $2n$ units of time
  while $j \leq i$ **do** ▷ $(i + 1) \times 2n$ units of time
    $s \leftarrow s + i$ ▷ $2 \times i \times 2n$ units of time
    $j \leftarrow j + 1$ ▷ $2 \times i \times 2n$ units of time
  end while
  $i \leftarrow i + 1$ ▷ $2n$ units of time
end while

To calculate the time cost of the inner loop, we need to focus on the value of i which changes with the outer loop. To make calculate easy to understand, we define C_1 as the actual units of time the outer loop will cost and C_2 as the actual units of time the inner loop will cost. So that:

$$C_1 = 1 + 1 + 2n + 1 + 2n + 2n = 6n + 3$$

$$C_2 = 2n \sum_{i=1}^{2n} (i + 1 + 2i + 2i) = 10n^2 + 7n$$

To sum up, the total units of time this algorithm will cost should be $time_{total} = C_1 + C_2 = 10n^2 + 13n + 3$. So that this algorithm is $\Theta(n^2)$

1.4.3 R-1.15

Answer: First, we need to rewrite this algorithm into *while* loop:

Algorithm Loop5(n):

$s \leftarrow 0$ $\triangleright$ 1 unit of time
$i \leftarrow 1$ $\triangleright$ 1 unit of time
while $i \leq n^2$ **do** $\triangleright$ $n^2 + 1$ units of time
  $j \leftarrow 1$ $\triangleright$ n^2 units of time
  while $j \leq i$ **do** $\triangleright$ $(i+1) \times n^2$ units of time
    $s \leftarrow s + i$ $\triangleright$ $2 \times i \times n^2$ units of time
    $j \leftarrow j + 1$ $\triangleright$ $2 \times i \times n^2$ units of time
  end while
  $i \leftarrow i + 1$ $\triangleright$ n^2 units of time
end while

To calculate the time cost of the inner loop, we need to focus on the value of i which changes with the outer loop. To make calculate easy to understand, we define C_1 as the actual units of time the outer loop will cost and C_2 as the actual units of time the inner loop will cost. So that:

$$C_1 = 1 + 1 + n^2 + 1 + n^2 + n^2 = 3n^2 + 3$$

$$C_2 = 2n \sum_{i=1}^{n^2} (i + 1 + 2i + 2i) = \frac{5}{2}n^4 + \frac{7}{2}n^2$$

To sum up, the total units of time this algorithm will cost should be $time_{total} = C_1 + C_2 = \frac{5}{2}n^4 + \frac{13}{2}n^2 + 3$. So that this algorithm is $\Theta(n^4)$

1.5 Explain why it is reasonable to ignore the overhead of a ranged *for* loop when you derived the tight asymptotic runtime bounds in the previous question.

Answer: When calculating the runtime bounds of a loop, the overhead of such loop will cost a constant time of computation, while the loop body will cost n times more than the overhead. As n grows big enough, the overhead is always significantly smaller than the loop body. Thus, when we are deriving the asymptotic runtime bounds, we can ignore the overhead of a ranged loop.

2 A Challenging Sum

Show that summation $\sum_{i=1}^{n} \lceil \log_2(n/i) \rceil$ is $O(n)$. You may assume that n is a power of 2.

Proof:

Base Case: Show that the statements holds for $n = 1$.
When $n = 1$, $\sum_{i=1}^{n} \lceil \log_2(n/i) \rceil = \lceil \log_2(1) \rceil = 0$, and $n = 1$, so we have $0 \leq 1$.
Inductive Step: Show that if $\sum_{i=1}^{n} \lceil \log_2(n/i) \rceil \leq cn$ holds, then also $\sum_{i=1}^{2n} \lceil \log_2(2n/i) \rceil \leq c2n$ holds.
Since we have $\sum_{i=1}^{n} \lceil \log_2(n/i) \rceil \leq cn$ holds, we can replace n by $2n$, thus we have:

$$\sum_{i=1}^{2n} \lceil \log_2(2n/i) \rceil = \sum_{i=1}^{2n} \lceil \log_2(n/i) + 1 \rceil = 2n + \sum_{i=1}^{2n} \lceil \log_2(n/i) \rceil$$

When $n < i \leq 2n$, $0.5 \leq (n/i) < 1$. Thus $-1 \leq \log_2(n/i) < 0$, thus we have

$$2n + \sum_{i=1}^{2n} \lceil \log_2(n/i) \rceil = 2n - 1 + \sum_{i=1}^{n} \lceil \log_2(n/i) \rceil \leq cn + 2n - 1$$

For any constant $c > 2$, we have $cn + 2n - 1 \le c2n$. Thereby that indeed $\sum_{i=1}^{2n} \lceil \log_2(2n/i) \rceil \le c2n$ holds.

Since both the base case and the inductive step have been performed, by mathematical induction, $\sum_{i=1}^{n} \lceil \log_2(n/i) \rceil$ is $O(n)$ holds for all n that is a power of 2. Q.E.D.

3 Theory: Big-Oh and Derivatives

In the following questions, suppose that $f, g : \mathbb{R} \to \mathbb{R}$ are differentiable and strictly increasing ($f'(x) > 0$ and $g'(x) > 0$ for all x). Prove the statement in each question, or construct a counterexample.

3.1 Is $f(x) \in O(g(x))$ if and only if $f'(x) \in O(g'(x))$?

Answer: No, the counterexample is shown as follow:

Suppose we have:

$$f(x) = 3 - \frac{1}{e^x}$$
$$g(x) = 4 - \frac{1}{e^{2x}}$$

Where $f(x) \in O(g(x))$ because:

$$\lim_{x\to\infty} \frac{f(x)}{(g(x))} = \lim_{x\to\infty} \frac{3 - \frac{1}{e^x}}{4 - \frac{1}{e^{2x}}} = \lim_{x\to\infty} \frac{\frac{3e^x - 1}{e^x}}{\frac{4e^{2x}-1}{e^{2x}}} = \lim_{x\to\infty} \frac{e^x(3e^x - 1)}{4e^{2x} - 1} = \lim_{x\to\infty} \frac{3e^{2x} - e^x}{4e^{2x} - 1}$$

According to L'Hospital's rule,

$$\lim_{x\to\infty} \frac{3e^{2x} - e^x}{4e^{2x} - 1} = \lim_{e^x\to\infty} \frac{6e^x - 1}{8e^x} = \lim_{e^x\to\infty} \frac{6}{8} = \frac{3}{4} = 0.75$$

Since $0 \le 0.75 < \infty$, it is indeed that $f(x) \in O(g(x))$.

However, when we have

$$f'(x) = \frac{1}{e^x}$$
$$g'(x) = \frac{2}{e^{2x}}$$

To examine $f'(x) \in O(g'(x))$, we can calculate $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ by:

$$\lim_{x\to\infty} \frac{f'(x)}{g'(x)} = \lim_{x\to\infty} \frac{\frac{1}{e^x}}{\frac{2}{e^{2x}}} = \lim_{x\to\infty} \frac{e^x}{2} = \infty$$

Such results indicate that $f'(x) \in \omega(g'(x))$, which stands against $f'(x) \in O(g'(x))$. So that the statement in this question is not true.

3.2 Is it true that if $\lim_{x\to+\infty} f'(x) = 0$, then $f(x) \in O(1)$?

Answer: No, the counterexample is shown as follow:

Suppose we have:

$$f(x) = \ln(x)$$

Where $f'(x) = \frac{1}{x}$, and $\lim_{x\to+\infty} f'(x) = 0$, however,

$$\lim_{x\to+\infty} \frac{\ln(x)}{1} = \lim_{x\to+\infty} \ln(x) = +\infty$$

So that $f'(x) \in \omega(1)$, which stands against $f'(x) \in O(1)$. So that the statement in this question is not true.

4 Theory: Properties of Big-Oh Notation

4.1 Let $f, g : \mathbb{R} \to \mathbb{R}$ be continuous and strictly increasing. Is it necessarily the case that $f \in O(g(x))$ or $g \in O(f(x))$? Prove, or provide a counterexample.

Answer: Yes, the statement is true, prove as follows:

According to the limit definition, the statement in the question is equivalent to: $\exists c_1 = \lim_{x\to+\infty} \frac{f(x)}{g(x)}$ or $c_2 = \lim_{x\to+\infty} \frac{g(x)}{f(x)}$, where $0 \le c_1 < +\infty$ or $0 \le c_2 < +\infty$

Thus we have:

$$1 = \lim_{x\to+\infty} \frac{f(x)}{g(x)} \times \frac{g(x)}{f(x)} = c_1 \times c_2$$

Since the definition field of $f(x)$ and $g(x)$ is $\mathbb{R}$, we can divide this field into these parts:

First, when $c_1 \in (-\infty, 0)$, it is not possible because in this case either $f(x)$ or $g(x)$ is not positive, which go against the condition given in the statement.

Second, when $c_1 = 0$, we can simple conclude that $f(x) \in O(g(x))$ because of the definition and $g(x) \in \omega(f(x))$ because in this case $c_2 = +\infty$.

Third, when $c_1 \in (0, +\infty)$, as is discussed above, we can also assert that $f(x) \in O(g(x))$ because of the definition.

Forth, when $c_2 = 0$, it is like the opposite of the second case that $f(x) \in \omega(g(x))$ and $g(x) \in O(f(x))$

To sum up, the statement is true.

4.2 R-1.18

Show that $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.

Proof:

$\to$: if we have $g(b) \in \Omega(f(n))$, which means $\exists 0 < c \le \infty, \lim_{n\to\infty} \frac{g(n)}{f(n)} = c$. We can assert that $\lim_{n\to\infty} g(n) \neq 0$. So when considering $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c'$, it is simply that $c' = \frac{1}{c}$. Since $0 < c \le \infty$, we can conclude that $0 \le c' < 0$, which means $f(n) \in O(g(n))$.

$\leftarrow$: if we have $f(n) \in O(g(n))$, which means $\exists 0 \le c < \infty, \lim_{n\to\infty} \frac{f(n)}{g(n)} = c$. We can assert that $\lim_{n\to\infty} f(n) \neq 0$. So when considering $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c'$, it is simply that $c' = \frac{1}{c}$. Since $0 \le c < \infty$, we can conclude that $0 < c' \le 0$, which means $g(n) \in \Omega(f(n))$.

4.3 R-1.19

Show that if $p(n)$ is a polynomial in n, then $\log p(n)$ is $O(\log n)$.

Proof:

Since $p(n)$ is a polynomial in n, which means

$$\exists a_0, a_1, a_2, a_3, ..., a_m \in \mathbb{R}, p(n) = a_m n^m + a_{m-1} n^{m-1} + ... + a_3 n^3 + a_2 n^2 + a_1 n^1 + a_0$$

Thus we have:

$$\lim_{n\to\infty} \frac{\log(p(n))}{\log(n)} = \lim_{n\to\infty} \frac{\log(\sum_{i=0}^m a_i n^i)}{\log(n)} = \lim_{n\to\infty} \frac{\ln(\sum_{i=0}^m a_i n^i)}{\ln(n)}$$

According to L'Hospital's rule, we have:

$$\lim_{n\to\infty} \frac{\ln(\sum_{i=0}^m a_i n^i)}{\ln(n)} = \lim_{n\to\infty} \frac{\sum_{i=1}^m i a_i n^{i-1} \times \frac{1}{\sum_{i=0}^m a_i n^i}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{\sum_{i=1}^m i a_i n^i}{\sum_{i=0}^m a_i n^i}$$

When we apply L'Hospital's rule for m times, we have:

$$\lim_{n\to\infty} \frac{\sum_{i=1}^m i a_i n^i}{\sum_{i=0}^m a_i n^i} = \lim_{n\to\infty} \frac{m \prod_{i=1}^m i}{\prod_{i=1}^m i} = m$$

Since m is the level of $p(n)$, $m \in \mathbb{N}_+$. Thus $0 \le m < +\infty$, so that $\log p(n) \in O(\log n)$.

5 Application: Matrix Multiplication

5.1 Provide a tight asymptotic upper bound on the runtime of easy-mutiply in terms of n.

Answer: To calculate the asymptotic upper bound, we need to rewrite this algorithm into *while* loop:

$Result \leftarrow$ an $n \times n$ matrix of zeros ▷ $n \times n$ units of time
$i \leftarrow 1$ ▷ 1 unit of time
while $i \leq n$ **do** ▷ $n + 1$ units of time
  $j \leftarrow 1$ ▷ n units of time
  while $j \leq n$ **do** ▷ $n \times (n + 1)$ units of time
    $k \leftarrow 1$ ▷ $n \times n$ units of time
    while $k \leq n$ **do** ▷ $n \times n \times (n + 1)$ units of time
      $Result[i][j] \leftarrow A[i][k] * B[k][j]$ ▷ $8 \times n^3$ units of time
      $k \leftarrow k + 1$ ▷ $2 \times n^3$ units of time
    end while
    $j \leftarrow j + 1$ ▷ $2 \times n^2$ units of time
  end while
  $i \leftarrow i + 1$ ▷ $2 \times n$ units of time
end while
return *Result* ▷ 1 unit of time

So the total time of execution is $11n^3 + 5n^2 + 5n + 3$, so that the upper bound should be $\Theta(n^3)$.

5.2 What is the asymptotic runtime of combined-multiply in terms of n?

Answer: Since when $n \geq 100$, this algorithm will use *Strassen Matrix Multiplication* method to calculate the multiplication, we can assert that the asymptotic runtime of combined-multiply is the same as *Strassen Matrix Multiplication*, so it is $O(n^{2.807})$.

6 "Application": Spaghetti Sort

6.1 Describe the asymptotic runtime of unmodified spaghetti-sort.

Answer: The *while* loop is executed for n times where n is the size of L. Inside of the *while* loop, there is a *for* loop which will be executed for m times where m maximum element in L, so that $m \leq 2^32 - 1$. As long as the append and remove operations are on the linked list, these operations and be done in constant time. To sum up, the runtime of unmodified algorithm should be $\Theta(n)$.

6.2 Suppose that we require that the input list is *not* a list of arbitrary natural numbers, but is instead a list of 64-bit unsigned integers. What is a tight bound on the asymptotic runtime of spaghetti sort now?

Answer: The tight bound should also be $\Theta(n)$, but the algorithm will possible try all the numbers in 64-bit unsigned integers, which means the actual runtime will be 2^{32} times longer than the original algorithm. Although 2^{32} is not related to the input size, we cannot see a direct reflection in the tight bound. However, the algorithm is much slower than other sorting algorithms even like bubble sort.

6.3 In the light of the answer to the previous question, why/ why not would we want to use spaghetti-sort for sorting 64-bit unsigned integers in practice?

Answer: When we consider the searching space size m into the complexity of algorithm, we can rewrite the runtime bound of this algorithm into $\Theta(n \times m)$, so that when we are sorting 64-bit unsigned integers, the value of m is significantly larger than n in most cases. At this time, the algorithm will cost most of the time trying to searching around the whole searching space, which contains mostly useless operation.

7 Algorithm Design: Finding Cycles

Answer: The pseudo-code of my algorithm is as follows:

```
fast ← head
slow ← head
while True do
    if fast = NULL then
        Break
    end if
    if fast.next = NULL then
        Break
    end if
    if fast.next.next = NULL then
        Break
    end if
    fast ← fast.next.next
    slow ← slow.next
    if fast.value = slow.value then
        return True
    end if
end while
return False
```

Here the *head* means the head node of the linked list, the $X.next$ means the next node of node X, and $X.value$ means the value stored in node X.

As is shown in the algorithm, the $fast$ node is one step faster than then *slow* node when performed every loop. When there is a cycle in the link list, the $fast$ node will need to go around the cycle for 2 times and the *slow* node will only go for one complete path through the cycle. Based on this observation, we can conclude that this *while* loop is executed for n times given the fact that the link list have a size of n. On the other hand, if there is no cycle in the linked list, we can see that $fast$ node will quickly get an $NULL$ value since it will jump two steps every loop, so that the loop is executed for less than $\frac{n}{2}$ times. To sum up, the loop is executed at most for n times. So that the asymptotic runtime is $O(n)$.

To examine the space usage, in this algorithm, we only used two temporarily nodes named $fast$ and *slow*, and the space usage is irrelevant to the size of n. We can conclude that the space usage is the size of two nodes, and bounded by $O(1)$.