CSOR 4231 Analysis of Algorithms I (2018 Fall)
Problem Set 1

Jing Qian - jq2282@columbia.edu

September 20, 2018

Problem 1

(a) $T(n) = \Theta(n^2)$. Line 1 and line 5 take constant time, $\Theta(1)$. Line 2 runs $n+1 = \Theta(n)$ times. Line 3 runs $\sum_{l=2}^{n+1} l = \Theta(n^2)$ times. Line 4 runs $\sum_{l=1}^{n} l = \Theta(n^2)$ times. Therefore, we have:

$$\begin{aligned} T(n) &= 2\Theta(1) + \Theta(n) + 2\Theta(n^2) \\ &= \Theta(n^2) \end{aligned} \tag{1}$$

(b) $T(n) = \Theta(n^3)$. Line 1 and line 6 take $\Theta(1)$ time. Line 2 runs $n+1 = \Theta(n)$ time. Line 3 runs $\sum_{l=2}^{n+1} l = \Theta(n^2)$ time. Line 4 runs $\sum_{l=2}^{n+1} \frac{l(l-1)}{2} = \Theta(n^3)$ times. Line 5 runs $\sum_{l=1}^{n} \frac{l(l+1)}{2} = \Theta(n^3)$ time. Therefore, we have:

$$\begin{aligned} T(n) &= 2\Theta(1) + \Theta(n) + \Theta(n^2) + 2\Theta(n^3) \\ &= \Theta(n^3) \end{aligned} \tag{2}$$

Problem 2

The functions are ordered as following with [...] representing grouped functions that have the same order of growth:

$$\begin{aligned} &10, \log\log n, [\log_{10} n, \log(n^2)], (\log n)^2, \sqrt[4]{n}, n(\log n)^2, [2^{2\log n}, (3n-2)^2], \\ &n^4, [2^{(\log n)^2}, n^{\log n}], 4^{\sqrt{n}}, \sqrt{2^n}, 2^{3n}, 3^{2n}, n!, n^n \end{aligned} \tag{3}$$

(a)

$$\begin{aligned} \log_{10} n &= \Theta(\log n) \\ \log(n^2) &= 2\log n = \Theta(\log n) \end{aligned} \tag{4}$$

So $\log_{10} n = \Theta(\log(n^2))$.
(b)

$$\begin{aligned} 2^{2\log n} &= (2^{\log n})^2 = n^2 = \Theta(n^2) \\ (3n-2)^2 &= \Theta(n^2) \end{aligned} \tag{5}$$

So $2^{2\log n} = \Theta((3n-2)^2)$.
(c)

$$2^{(\log n)^2} = (2^{\log n})^{\log n} = n^{\log n} \tag{6}$$

So $2^{(\log n)^2} = \Theta(n^{\log n})$.

Problem 3

1. To solve this problem in linear time limit, we could use the Two Pointers Technique.

Let i point to the first element of array A which is the minimum of array A and let j point to the last element of array B which is the maximum of array B. Check whether $A[i] + B[j] = 100$. If their sum equals to 100, A[i] and B[j] are the elements we are looking for. If their sum is larger than 100, then we shift the pointer j to the left by $j-1$ and check the equation again. If their sum is smaller than 100, then we shift the pointer i to the right by $i+1$ and check the equation again. We keep moving these two pointers until we find a pair of elements with the sum equals to 100 or i points to the last element of A or j points to the first element of B. The loop invariant is $1 \leq i \leq n, 1 \leq j \leq n$.

1). If the sum of A[i] + B[j] equals to 100, we find the answer.

2). If i points to the last element of A and A[n] + B[j] > 100, keep moving j left to check whether there exists j to make A[n] + B[j] $= 100$ possible. If so, we find the answer. If A[n] + B[j] < 100, stop moving j left because the sum decreases with decreasing j. There is no pair of elements whose sum equals to 100.

3). If j points to the first element of B and A[i] + B[1] < 100, keep moving i right to check whether there exists i to make A[i] + B[1] $= 100$ possible. If so, we find the answer. If A[i] + B[1] > 100, stop moving i right because the sum increases with increasing i. There is no pair of elements whose sum equals to 100.

The worst case is that we check over array A from the first element to the last element and array B from the last element to the first element. We may not find the elements we are looking for, or we may find A[n] + B[1] $= 100$ in the final check. In the worst case, we check every element in array A and array B once. So the running time of **while** loop is $\Theta(n)$. Considering other operations only take constant time, the algorithm based on Two Pointers Technique runs in linear time in the worst case.

2. To solve this problem in $O(n \log n)$ time, we could sort array C with Merge Sort method which takes $O(n \log n)$ time, and then use the Two Pointers Technique mentioned in Problem 3.1, which takes $O(n)$ time.

Given array C sorted, we let pointer i to the first element of C and pointer j to the last element of C. If C[i] + C[j] $= 100$, the answer to this problem is yes with the two elements are C[i] and C[j]. If the sum of C[i] and C[j] is larger than 100, we move pointer j left by $j-1$ and check the equation again. If the sum is smaller than 100, we move pointer i right by $i+1$ and check the equation again. We keep moving pointers i and j until C[i] + C[j] $= 100$ or $i = j$. If $i = j$, there are not two elements that their sum equals to 100. The loop invariant is $1 \leq i \leq j \leq n$.

The worst case is that after we check the whole array C, the answer is no or the answer is yes only at the final check. In other words, the worst case is when we checked every element in C once. So the Two Pointer Technique takes linear time.

In this algorithm, $O(n \log n)$-time Merge Sort and $O(n)$-time Two Pointers are used, plus several constant-time operations. So the running time of this algorithm is $O(n \log n)$.

Problem 4

1. No, this algorithm does not run in polynomial time in n.

Given every operation takes polynomial time of n, in order to justify this answer, we need to show that the number of operations in this algorithm is not polynomial of n. Since line 1 and line 4 are 2 operations, the number of operations are mainly dependent on the **while** loop in line 2 and line 3. The worst case is that the greatest common divisor of a and b is 1. So t decreases from $\min(a, b)$ to 1 by step 1 and **while** loop runs $\min(a, b)$ times. Since $n = (\lfloor \log a \rfloor + 1) + (\lfloor \log b \rfloor + 1)$, the number of operations is $2 + 2\ O(\min(a, b)) = O(2^n)$.

Therefore, this algorithm does not run in polynomial time in n.

2.

a). To be loop invariant, the statements hold true before the loop and do not be changed by iterations.

Before the loop, x is the larger one in a and b while y is the smaller one. So $x \geq y$ must be true. Because a and b are x and y or reversed, the common divisors of a and b equal to those of x and y. So both loop invariant (i) and (ii) are true before the loop.

To clarify the discussion, we expand the algorithm as following:

$$
\begin{aligned}
& x_0 = \max(a, b) \\
& y_0 = \min(a, b) \\
& r_0 = x_0 \bmod y_0 \\
& x_1 = y_0 \\
& y_1 = r_0 \\
& \text{.....} \\
& r_{i-1} = x_{i-1} \bmod y_{i-1} \\
& x_i = y_{i-1} \\
& y_i = r_{i-1} \\
& \text{.....} \\
& x_N \bmod y_N == 0 \\
& \text{return } y_N
\end{aligned}
\tag{7}
$$

According to the definition of **mod**, for $r := x$ mod y, $r \in [0, y-1]$, so we have $r < y$ and $r \leq x$. Since y is assigned to the x in the next iteration and r is assigned to the y in the next iteration, $x \geq y$ holds after each iteration. So (i) is loop invariant.

We could show loop invariant (ii) in the following way: for all positive integers c, c divides both x_i and y_i if and only if c divides both x_{i-1} and y_{i-1}. If c divides both x_{i-1} and y_{i-1}, we could write as following: $x_{i-1} = m_{i-1}c$ and $y_{i-1} = n_{i-1}c$. Then:

$$\begin{aligned} r_{i-1} &= x_{i-1} \bmod y_{i-1} \\ &= (m_{i-1}c) \bmod (n_{i-1}c) \\ &= (m_{i-1} \bmod n_{i-1})\ c \end{aligned} \tag{8}$$

Since both y_{i-1} and r_{i-1} are multiples of c, x_i and y_i are multiples of c. Moreover, if c divides both x_i and y_i, $y_{i-1} = x_i$ is multiple of c. $x_{i-1} = \text{quotient} * y_{i-1} + r_{i-1} = \text{quotient} * x_i + y_i$, so x_{i-1} is also multiple of c. This "if and only if" relationship holds in every iteration of the loop, so (ii) is loop invariant.

b). According to the loop invariant (ii) in previous discussion, since the returning y_N divides x_N, y_N divides both x_i and y_i in $i-$th iteration and therefore is a common divisor of a and b. We only need to show there is no greater common divisor of a and b than y_N.

Let g be the greatest common divisor of a and b. So g is the greatest common divisor of x_0 and y_0. According to the loop invariant (ii) in **a** part, g is the common divisor of x_i and y_i in $i-$th iteration and therefore the common divisor of x_N and y_N. Then we have $g \leq y_N$. Since g is the greatest common divisor of $a,\ b$ and y_N is a common divisor of $a,\ b$, returned y_N equals to g. In other words, Euclids algorithm returns the greatest common divisor of $a,\ b$.

c). Since we could regard max, min, mod as primitive operations for this problem, the number of operations depends on the iteration number of the **while** loop. In the worst case, y does not divide x until y reduces to 1.

Then we should discuss how the **mod** operation shrink the size of x. As we shown in **a)**, $x \geq y$ is loop invariant. If $y \in (0, x/2]$, $(x \bmod y) \in [0, y-1]$. So $(x \bmod y) < x/2$ when $y \in (0, x/2]$. If $y \in (x/2, x]$, $(x \bmod y) = x - y$. So $(x \bmod y) < x/2$ when $y \in (x/2, x]$. Therefore, $(x \bmod y) < x/2$ when $x \geq y$.

The running time of **while** loop in the worst case could be written as: $T(x) = T(x/2) + c$. Using the master method, $T(x) = \mathrm{O}(\log x)$. Since $n = (\lfloor \log a \rfloor + 1) + (\lfloor \log b \rfloor + 1) > \log x$, $f(n) = \mathrm{O}(n)$.