2

COMPLEXITY ANALYSIS

1.   (a)   The function f never exceeds the value of a certain constant c.

       (b) f is a constant function.

       (c)   The function f never exceeds the value of the power function n^c for some constant c.

2.   In the following answers, these two definitions are used:

> $f_1(n)$ is $O(g_1(n))$ if there exist positive numbers c_1 and N_1 such that $f_1(n) \leq c_1 g_1(n)$ for all $n \geq N_1$;
>
> $f_2(n)$ is $O(g_2(n))$ if there exist positive numbers c_2 and N_2 such that $f_2(n) \leq c_2 g_2(n)$ for all $n \geq N_2$;

       (a)   From the above definitions, we have

$$f_1(n) \leq c_1 \cdot \max(g_1(n), g_2(n)) \text{ for all } n \geq \max(N_1, N_2),$$

$$f_2(n) \leq c_2 \cdot \max(g_1(n), g_2(n)) \text{ for all } n \geq \max(N_1, N_2),$$

which implies that

$$f_1(n) + f_2(n) \leq (c_1 + c_2) \cdot \max(g_1(n), g_2(n)) \text{ for all } n \geq \max(N_1, N_2).$$

Hence for $c_3 = c_1 + c_2$ and $N_3 = \max(N_1, N_2)$, $f_1(n) + f_2(n) \leq c_3 \cdot \max(g_1(n) + g_2(n))$ for all $n \geq N_3$, that is, $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$.

       (b)   If $g_1(n) \leq g_2(n)$, then for $c = \max(c_1, c_2)$, $cg_1(n) \leq cg_2(n)$ and $cg_1(n) + cg_2(n) \leq 2cg_2(n)$, which implies that $O(g_1(n)) + O(g_2(n))$ is $O(g_2(n))$.

       (c)   The rule of product, $f_1(n) \cdot f_2(n)$ is $O(g_1(n) \cdot g_2(n))$ is true, since $f_1(n) \cdot f_2(n) \leq c_1 c_2 g_1(n) \cdot g_2(n)$ for all $n \geq \max(N_1, N_2)$.

       (d) $O(cg(n))$ is $(O(g(n))$ means that any function f which is $O(cg)$ is also $O(g)$. Function f is $O(cg)$ if there are two constants c_1 and N so that $f(n) \leq c_1 cg(n)$ for all $n \leq N$; in this case, for a constant $c_2 = c_1 c$, $f(n) \leq c_2 g(n)$; thus by choosing properly a constant c_2 (whose value depends on the value of c and c_1), f is $O(g)$.

(e) A constant c is $O(1)$ if there exist positive numbers c_1 and N such that $c \le c_1 \cdot 1$ for all $n \ge N$; that is, the constant function c is independent of n, and we can simply set $c_1 = c$.

3. (a) $\sum_{i=1}^{n} i^2$ is $O(n^3)$ if a constant can be found such that $\sum_{i=1}^{n} i^2 \le cn^3$; but $\sum_{i=1}^{n} i^2 \le n \cdot n^2 = n^3$, thus $c = 1$ for any n.

(b) Function $an^k / \lg n$ is $O(n^k)$ if there is a c and N for which $an^k / \lg n \le cn^k$ for all $n \ge N$. The inequality becomes $a / \lg n \le c$ for $N > 1$, and since $a / \lg n \to 0$ we can put $c = a$. But there is no *positive c* for which $c \le a / \lg n$ (it holds for $c = 0$), thus $an^k / \lg n$ is not $\Theta(n^k)$.

(c) Function $n^{1.1} + n \lg n$ is $\Theta(n^{1.1})$ if two constants can be found such that $c_1 n^{1.1} \le n^{1.1} + n \lg n \le c_2 n^{1.1}$. These inequalities can be transformed into $c_1 \le 1 + n^{-1} \lg n \le c_2$; by the rule of L'Hospital, $n^{-1} \cdot \lg n = (\lg n) / n^{1}$ has the same limit as $(\lg e) / (.1 \cdot n^{-1}) = (10 \cdot \lg e) / n^{-1}$, which is 0. Hence, $c_1 = 1, c_2 = 1 + 10 \lg e$.

(d) 2^n is $O(n!)$ if there is a c and N for which $2^n \le cn!$ for all $n \ge N$. If $N = 1$, then $2^n \le cn!$ implies that $2^n \le c(1 \cdot 2 \cdot 3 \cdot ... \cdot n)$, i.e., $\frac{2}{1} \cdot \frac{2}{2} \cdot \frac{2}{3} \cdot ... \cdot \frac{2}{n} \le 2 = c$.

For the other part of the exercise: $n!$ is $O(2n)$ if there are constants c and N such that $n! \le c2^n$ for $n \ge N$. The inequality $n! \le c2^n$ implies $1 \cdot 2 \cdot 3 \cdot ... \cdot n \le c(2^n)$, i.e., $\frac{1}{2} \cdot \frac{2}{2} \cdot \frac{3}{2} \cdot ... \cdot \frac{n}{2} \le c$, for all n's. But such a constant c cannot be found.

(e) We can find such a c that for some N and all $n \ge N, 2^{n+a} \le c2^n$, if $c \ge 2^{n+a} / 2^n = 2^a$.

(f) We cannot find such a c that for some N and all $n \ge N, 2^{n+a} \le c2^n$, because there exists no constant $c \ge 2^{n+a} / 2^n = 2n - a..$

(g) Because $n = 2^{\lg n}$, then $n^a = 2^{a \lg n}$; therefore, if $2^{a \lg n} > 2^{\sqrt{\lg n}}$, then $n^a > 2^{\sqrt{\lg n}}$. Now we have to find such a c that for some N and all $n \ge N, 2^{\sqrt{\lg n}} \le cn^a$, or $2^{\sqrt{\lg n}} \le c2^{a \lg n}$, i.e., $c \ge 2^{\sqrt{\lg n}} / 2^{a \lg n}$, which is possible because the function $2^{\sqrt{\lg n}} / 2^{a \lg n}$ is decreasing.

4. (a) Let $f_1(n) = a_1 n$, and $f_2(n) = a_2 n$; then both f_1 and f_2 are $O(n)$, but $f_1(n) - f_2(n) = (a_1 - a_2)n$ is not $O(n - n) = O(0)$. Hence, $f_1(n) - f_2(n)$ is not $O(g_1(n) - g_2(n))$.

(b) Take the same functions as before; $f_1(n) / f_2(n) = (a_1 / a_2)n$ is not $O(n / n) = O(1)$. Therefore, $f_1(n) / f_2(n)$ is not $O(g_1(n) / g_2(n))$.

5. For functions $f_1(n) = an^2 + O(n), f_2(n) = cn + d, g(n) = n^2$, both $f_1(n)$ and $f_2(n)$ are $O(g(n))$, but f_1 is not $O(f_2)$.

6.   (a)   It is not true that if $f_1(n)$ is $\Theta(g(n))$ then $2^{f(n)}$ is $\Theta(2^{g(n)})$, if, for example, $f(n) = n$, and $g(n) = 2n$.

       (b) $f(n) + g(n)$ is not $\Theta\big(\min\big(f(n), g(n)\big)\big)$, if, for example, $f(n) = n$, and $g(n) = \dfrac{1}{n}$.

       (c) 2na is not $O(2^n)$, because there is no constant $c \geq 2^{na}/2^n = 2^{n(a-1)}$ for any n and $a > 1$; for $a \leq 1$ the function $2^{n(a-1)}$ is decreasing so that a c meeting the specified condition can be found.

7.   If the line

```
for (i = 0, length = 1; i < n-1; i++)
```

is replaced by the line

```
for (i = 0, length = 1; i < n-1 && length < n-i; i++)
```

in the algorithm for finding the longest subarray with numbers in increasing order, then the best case, when all numbers in the array are in decreasing order, remains $O(n)$ since the inner loop is executed just once for each of the $n-1$ executions of the outer loop. For the ordered array, the outer loop is executed just once and the inner loop $n-1$ times, which makes it another best case.

But the algorithm is still $O(n^2)$. For example, if the array is [5 4 3 2 1 1 2 3 4 5], i.e., the first half of the array is in descending order, then the outer loop executes $n/2$ times and for each iteration $i = 1, \ldots, n/2$, the inner loop iterates i times, which makes $O(n^2)$ iterations in total. This inefficiency is due to the fact that the inner loop remembers only the length of the longest subarray, not its position, thereby checking subarrays of this subarray, e.g., after checking the subarray [5 4 3 2 1], it also checks its subarrays [4 3 2 1], [3 2 1], etc. To improve the algorithm more and make it $O(n)$, the inner loop

```
for (i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);
```

should be changed to

```
for (i1 = i2 = k; k < n-1 && a[k] < a[k+1]; k++, i2++);
```

8.   The complexity of `selectkth()` is $(n-1) + (n-2) + \cdots + (n-k) = (2n - k - 1)k/2 = O(n^2)$.

9.   The algorithm for adding matrices requires n^2 assignments. Note that the counter i for the inner loop does not depend on the counter j for the outer loop and both of them take values $0, \ldots, n-1$.

All three counters, i, j, and k in the algorithm for matrix multiplication are also independent of each other, hence the complexity of the algorithms is n^3.

To transpose a matrix, $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 3 = O(n^2)$ assignments are required.

10.   (a)   The autoincrement `cnt1++` is executed exactly n^2 times.

       (b) $\sum_{i=1}^{n} i = O(n^2)$
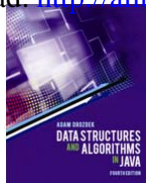
```
// DataStructures&AlgorithmsInJava (Adam Drozdek, 4th ed.)

// Instructor's Solutions Manual

// © 2013 Cengage Learning, all rights reserved.
```

(c) $\sum_{i=1}^{\lg n} i = \Theta(n \lg n)$

(d) $\sum_{i=1}^{\lg n} 2^i = \Theta(n)$

11. $\dfrac{1+\ldots+(n-2)}{4(n-2)} + \dfrac{n-1}{4} + \dfrac{n}{2} = \dfrac{n-1+2(n-1)+4n}{8} = \dfrac{7n-3}{8}$

12.   To prove the conjecture, the best thing to do is to devise an amortized cost that remains constant for each increment step. For example,

$$amCost(increment(x)) = 2 \cdot (\text{number of bits in } x \text{ set to } 1)$$

If the cost of setting one bit is one unit, then after setting a 1 bit there is one unit left for setting this bit back to 0, therefore, there is no need to charge anything for setting bits to 0. Note that the amortized cost for one increment is always 2.

Another definition is

$$amCost(increment(x)) = (\text{number of flipped bits}) + (\text{number of 1s added to } x)$$

The following table illustrates the application of this definition to increments of a 3-bit binary number.

Number	Flipped bits	Added 1s	Amortized cost
000			
001	1	1	2
010	2	0	2
011	1	1	2
100	3	−1	2
101	1	1	2
110	2	0	2
111	1	1	2

13.   For an alternative $A = (p_1 \vee p_2)$, generate an expression $A' = (p_1 \vee p_2 \vee z) \wedge (p_1 \vee p_2 \neg z)$. For an alternative $A = p_1$, generate the expression:

$$A' = (p_1 \vee y \vee z)\big((p_1 \vee \neg y \vee z) \wedge (p_1 \vee y \vee \neg z) \wedge (p_1 \vee \neg y \vee \neg z)\big).$$