**1.** Explain the meaning of the following expressions:

a. $f(n)$ is $O(1)$.

b. $f(n)$ is $\Theta(1)$.

c. $f(n)$ is $n^{O(1)}$.

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.** Assuming that $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, prove the following statements:

a. $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$.

b. If a number $k$ can be determined such that for all $n > k$, $g_1(n) \le g_2(n)$, then $O(g_1(n)) + O(g_2(n))$ is $O(g_2(n))$.

c. $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$ (rule of product).

d. $O(cg(n))$ is $O(g(n))$.

e. $c$ is $O(1)$.

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 $f_1(n) \le c_1 g_1(n)$ for all $n \ge N_1$;

$f_2(n)$ is $O(g_2(n))$ if there exist positive numbers $c_2$ and $N_2$ such that $f_2(n) \le c_2 g_2(n)$ for all $n \ge N_2$;

(a) From the above definitions we have

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

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

which implies that

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

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

(b) If $g_1(n) \le g_2(n)$, then for $c = \max(c_1, c_2)$, $cg_1(n) \le cg_2(n)$ and $cg_1(n) + cg_2(n) \le 2cg_2(n)$, 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 $f_1(n) \cdot f_2(n) \le c_1 c_2 g_1(n) \cdot g_2(n)$ for all $n \ge \max(N_1, N_2)$.

(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 $f(n) \le c_1 cg(n)$ for all $n \le N$; in this case, for a constant $c_2 = c_1 c$, $f(n) \le c_2 g(n)$; 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.** Prove the following statements:

a. $\sum_{i=1}^{n} i^2$ is $O(n^3)$ and more generally, $\sum_{i=1}^{n} i^k$ is $O(n^{k+1})$.

b. $an^k/\lg n$ is $O(n^k)$ but $an^k/\lg n$ is not $\Theta(n^k)$.

c. $n^{1.1} + n \lg n$ is $\Theta(n^{1.1})$.

d. $2^n$ is $O(n!)$ and $n!$ is not $O(2^n)$.

e. $2^{n+a}$ is $O(2^n)$.

f. $2^{2n+a}$ is not $O(2^n)$.

g. $2^{\sqrt{\lg n}}$ is $O(n^a)$.

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 $2^n \le c(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n)$, i.e., $\frac{2}{1} \cdot \frac{2}{2} \cdot \frac{2}{3} \cdot \ldots \cdot \frac{2}{n} \le 2 = c$.
For the other part of the exercise: $n!$ is $O(2^n)$ if there are constants $c$ and $N$ such that $n! \le c2^n$ for $n \ge N$. The inequality $n! \le c2^n$ implies $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \le c(2^n)$, i.e., $\frac{1}{2} \cdot \frac{2}{2} \cdot \frac{3}{2} \cdot \ldots \cdot \frac{n}{2} \le c$, 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 $c \ge 2^{2n+a}/2^n = 2n - a$.

(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.** Make the same assumptions as in Exercise 2 and, by finding counterexamples, refute the following statements:

a. $f_1(n) - f_2(n)$ is $O(g_1(n) - g_2(n))$.

b. $f_1(n)/f_2(n)$ is $O(g_1(n)/g_2(n))$.

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.** Find functions $f_1$ and $f_2$ such that both $f_1(n)$ and $f_2(n)$ are $O(g(n))$, but $f_1(n)$ is not $O(f_2)$.

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

**6.** Is it true that

a. if $f(n)$ is $\Theta(g(n))$, then $2^{f(n)}$ is $\Theta(2^{g(n)})$?

b. $f(n) + g(n)$ is $\Theta(\min(f(n),g(n)))$?

c. $2^{na}$ is $O(2^n)$?

6. (a) It is not true that if $f(n)$ 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 $\Theta(\min(f(n), g(n)))$, if, for example, $f(n) = n$, and $g(n) = \frac{1}{n}$.

(c) $2^{na}$ 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.** The algorithm presented in this chapter for finding the length of the longest subarray with the numbers in increasing order is inefficient, because there is no need to continue to search for another array if the length already found is greater than the length of the subarray to be analyzed. Thus, if the entire array is already in order, we can discontinue the search right away, converting the worst case into the best. The change needed is in the outer loop, which now has one more test:

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

What is the worst case now? Is the efficiency of the worst case still $O(n^2)$?

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 $i = 1, ..., n/2$, 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.** Find the complexity of the function used to find the *k*th smallest integer in an unordered array of integers

```
int selectkth(int a[], int k, int n) {
    int i, j, mini, tmp;
    for (i = 0; i < k; i++) {
        mini = i;
        for (j = i+1; j < n; j++)
            if (a[j]<a[mini])
                mini = j;
        tmp = a[i];
        a[i] = a[mini];
        a[mini] = tmp;
    }
    return a[k-1];
}
```

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

**9.** Determine the complexity of the following implementations of the algorithms for adding, multiplying, and transposing $n \times n$ matrices:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = b[i][j] + c[i][j];

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = a[i][j] = 0; k < n; k++)
            a[i][j] += b[i][k] * c[k][j];

for (i = 0; i < n - 1; i++)
    for (j = i+1; j < n; j++) {
        tmp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = tmp;
    }
```

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.** Find the computational complexity for the following four loops:

a.
```
for (cnt1 = 0, i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
          cnt1++;
```

b.
```
for (cnt2 = 0, i = 1; i <= n; i++)
      for (j = 1; j <= i; j++)
          cnt2++;
```

c.
```
for (cnt3 = 0, i = 1; i <= n; i *= 2)
      for (j = 1; j <= n; j++)
          cnt3++;
```

d.
```
for (cnt4 = 0, i = 1; i <= n; i *= 2)
      for (j = 1; j <= i; j++)
          cnt4++;
```

10. (a) The autoincrement `cnt1++` is executed exactly $n^2$ times.
    (b) $\sum_{i=1}^{n} i = O(n^2)$;
    (c) $\sum_{i=1}^{\lg n} i = \Theta(n \lg n)$.
    (d) $\sum_{i=1}^{\lg n} 2^i = \Theta(n)$.

**11.** Find the average case complexity of sequential search in an array if the probability of accessing the last cell equals $\frac{1}{2}$, the probability of the next to last cell equals $\frac{1}{4}$, and the probability of locating a number in any of the remaining cells is the same and equal to $\frac{1}{4(n-2)}$.

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

12. Consider a process of incrementing a binary $n$-bit counter. An increment causes some bits to be flipped: Some 0s are changed to 1s, and some 1s to 0s. In the best case, counting involves only one bit switch; for example, when 000 is changed to 001, sometimes all the bits are changed, as when incrementing 011 to 100.

| **Number** | **Flipped Bits** |
|---|---|
| 000 | |
| 001 | 1 |
| 010 | 2 |
| 011 | 1 |
| 100 | 3 |
| 101 | 1 |
| 110 | 2 |
| 111 | 1 |

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 |