Worksheet Questions

1. If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on some inputs?

> **Solution:**
>
> Yes, there is no requirement for the function in the big-Oh to be tight. In addition the big-Oh bound refers to the worst-case input and some inputs may not elicit the worst-case time.

2. If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on all inputs?

> **Solution:**
>
> Yes, there is no requirement for the function in the big-Oh to be tight. So we might say $O(n^2)$ but it's possible that all inputs take $O(n)$ time.

Exercises 10.4

For the code segments in Exercises 7-12, determine which of the orders of magnitude given in this section is the best O to use to express the worst-case computing time as a function of n.

8.

```
        // Matrix addition
(1)     for (int i = 0; i < n; i++)
(2)        for (int j = 0; j < n; j++)
(3)           c[i][j] = a[i][j] + b[i][j]
```

> **Solution:** For ease of discussion, the lines of the algorithm have been numbered in the code above.
>
> - The for loop on line (1) iterates $O(n)$ times in the worst-case.
> - The for loop on line (2) iterates $O(n)$ times, for each iteration of the for loop on line (1), in the worst-case.
> - Line (3) takes $O(1)$ time for each iteration of the for loop on line (2).
>
> Therefore, the worst-case time complexity of this algorithm is:
>
> $$\begin{aligned} T(n) &= O(n * (n * (1))) \\ &= O(n^2) \end{aligned}$$
>
> *Notice that since the for loops are nested we multiply their time-complexities.*

10.

```
         // Bubble sort
(1)      for (int i = 0; i < n - 1; i++)
         {
(2)         for (int j = 0; j < n - 1; j++)
(3)            if (x[j] > x[j + 1])
               {
(4)               temp = x[j];
(5)               x[j] = x[j+1];
(6)               x[j+1] = temp;
               }
         }
```

Solution: For ease of discussion, the lines of the algorithm have been numbered in the code above.

- The for loop on line (1) iterates $O(n)$ times in the worst-case.
- The for loop on line (2) iterates $O(n)$ times, for each iteration of the for loop on line (1), in the worst-case.
- The conditional on line (3) takes $O(1)$ time for each iteration of the for loop on line (2).
- If the conditional on line (3) evaluates to `TRUE`, then lines (4), (5), and (6) each take $O(1)$ time.

Therefore, the worst-case time complexity of this algorithm is:

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

Notice that since the for loops are nested we multiply their time-complexities. Also notice that since lines (4), (5), (6) occur in series we add their time complexities.

11.

```
(1)      while (n >= 1)
(2)         n /= 2;
```

Solution: For ease of discussion, the lines of the algorithm have been numbered in the code above.

Each iteration of the while loop on line (1) halves n (by line (2)). This continues until $n < 1$. How many times can you halve n until you get to 1? $\lg n$

Therefore, the worst-case time complexity of this algorithm is:

$$T(n) = O(\lg n)$$

12.

```
(1)   x = 1;
(2)   for (int i = 1; i <= n; i++)
      {
(3)      for (int j = 1; j <= x; j++)
(4)         cout << j << endl;
(5)      x *= 2
      }
```

Solution: For ease of discussion, the lines of the algorithm have been numbered in the code above.

- Line (1) takes $O(1)$ time.
- The for loop on line (2) iterates $O(n)$ times in the worst-case.
- The for loop on line (3) iterates x times, for each iteration of the for loop on line (2). Since x is not an input variable (it is defined on line (1)), we must determine what x is in terms of the input variable n.
  - The first time we encounter the for loop on line (3), $x = 1 = 2^0$
  - The second time we encounter the for loop on line (3), $x = 1 * 2 = 2^1$
  - The third time we encounter the for loop on line (3), $x = 1 * 2 * 2 = 2^2$
  - The fourth time we encounter the for loop on line (3), $x = 1 * 2 * 2 * 2 = 2^3$
  - ...
  - The final time we encounter the for loop on line (3), $x = 2^n$

  Therefore, in the worst case $x = O(2^n)$
- Line (4) takes $O(1)$ time for each iteration of the for loop on line (3).
- Line (5) takes $O(1)$ time for each iteration of the for loop on line (2).

Therefore, the worst-case time complexity of this algorithm is:

$$\begin{aligned} T(n) &= O\left(1 + n\left(x\left(1\right) + 1\right)\right) \\ &= O\left(1 + n\left(2^n\left(1\right) + 1\right)\right) \\ &= O\left(\left(n\right)\left(2^n\right)\right) \end{aligned}$$

Notice that we cannot drop the n (even though it is a lower order term than 2^n) since the two terms are multiplied together. We only drop lower order terms and constants that occur in series (i.e., are added together).