Section 7.3 Additional Exercises

1. int CountValuesLessThanT(int a[], int n, int T) {
   int count = 0;
   for (int i = 0; i < n; i++)
      if (a[i] < T) count++;
   return count;
}

   In the worst case, the body of the for loop will always execute count++.
   The body of the for loop has constant run-time, and, if we include the loop test and
   the increment,
   i < n;
   if (a[i] < T) count++;
   i++;

   each iteration of the loop also has constant run-time. Since there are n iterations, we
get that the total run-time including loop variable initialization, final test, branch out
of the loop, and return is linear. So the run-time is \( \Theta(n) \).

2a. int MaxSubseqSum(int a[], int n) {
    int maxSum = 0;
    for (int i = 0; i < n; i++) {
        int thisSum = 0;
        for (int j = i; j < n; j++) {
            thisSum += a[j];
            if (thisSum > maxSum) maxSum = thisSum;
        }
    }
}
The body of the inner loop, including the loop test and loop variable increment has constant runtime. Since the loop iterates from \(i\) to \(n-1\), it has a total of \((n-1)-i+1 = n - i\) iterations. Including loop variable initialization, final loop test, and branch out, we get that the inner loop has run-time that is \(\Theta(n-i)\).

The initialization of \(thisSum\), the outer loop test, and the outer loop increment only add a constant to this. So the run-time of one iteration of the outer loop is also \(\Theta(n-i)\). This implies that the total cost of all the iterations of the outer loop is

\[
\Theta (n + (n-1) + \cdots + 3 + 2 + 1)
\]

which is

\[
\Theta \left( \frac{n(n+1)}{2} \right) \text{ or } \Theta(n^2).
\]

Initialization of the outer loop variable, final loop test, branch out of the loop, and return is constant. So the run-time of the entire algorithm is also \(\Theta(n^2)\).

3. int FindMaxFcnVal(int a[], int n) {
    int max = M(a[0], a[0], a[0]);
    
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; i++)
            for (int k = 0; k < n; i++) {
                int new = M(a[i], a[j], a[k]); // Theta(1)
                if (new > max) max = new;
            }
    
    return max;
}

We’ll omit the references to constants added by the \texttt{for} statements, since they won’t change the run-time.

Since \(M\) has run-time that is \(O(1)\), it’s run-time is also \(\Theta(1)\). So the body of the innermost loop is \(\Theta(1)\), and since there are \((n-1) - 0 + 1 = n\) iterations of the innermost loop, the run-time of the innermost loop is \(\Theta(n)\).

The middle loop also has \(n\) iterations. So the total run-time of middle loop is \(\Theta(n \cdot n)\) or \(\Theta(n^2)\).

The outer loop has \(n\) iterations. So its total run-time is \(\Theta(n \cdot n^2)\) or \(\Theta(n^3)\).

Since the run-time of the initialization of \texttt{max} and the return statement are constant, the run-time of the algorithm is \(\Theta(n^3)\).
int FindProduct(int a[], int n, int prod) {
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            if (a[i]*a[j] == prod) return 1;
    return 0;
}

(a) In order to maximize the number of statements executed, the algorithm should return 0, and this happens when there is no pair of distinct indices \( i \neq j \), with the property that \( a[i]*a[j] = \text{prod} \).

(b) The body of the inner loop has constant run-time. So it is \( \Omega(1) \). In the worst case there will be \( (n - 1) - (i + 1) + 1 = n - i - 1 \) iterations of the inner loop. So the total cost of the inner loop is \( \Omega(n - i) \).

In the worst case there are \( n \) iterations of the outer loop. So the run-time is bounded below by the sum

\[
n + (n - 1) + (n - 2) + \cdots + 3 + 2 + 1 = \frac{n(n + 1)}{2}.
\]

The return statement has constant run-time. So the total run-time is \( \Omega(n^2) \).

(c) In the best case, \( a[0]*a[1] = \text{prod} \), and the algorithm will have constant run-time, which is not \( \Omega(n^2) \). So it was important that we analyzed worst-case run-time.

(d) The analysis for big-\( O \) is essentially the same as the analysis for big-\( \Omega \): the run-time is \( O(n^2) \).
Extra Problems

1. Recall that in class we showed that the following algorithm was
\[ O \left( \sum_{i=0}^{n-1} f_i(n) \right). \]

Show that it is also
\[ \Omega \left( \sum_{i=0}^{n-1} f_i(n) \right). \]

```c
for (i = 0; i < n; i++) {
    // Runtime of func(i,n) is T_i(n)
    // which is Theta(f_i(n))
    func(i,n);
}
```

Since `func(i,n)` has run-time `T_i(n)`, and `T_i(n)` is \( \Theta(f_i(n)) \), `T_i(n)` is \( \Omega(f_i(n)) \), and there exist positive constants \( c_0, c_1, \ldots, c_{n-1}, m_0, m_1, \ldots, m_{n-1} \) such that
\[ T_i(n) \geq c_i f_i(n), \forall n \geq m_i, \]
and for \( i = 0, \ldots, n-1 \). If we choose \( n_0 = \max\{m_0, m_1, \ldots, m_{n-1}\} \) and \( c = \min\{c_0, c_1, \ldots, c_{n-1}\} \), then
\[ T_i(n) \geq c f_i(n), \forall n \geq n_0, \]
and for \( i = 0, \ldots, n-1 \).

The `for` statement adds in a fixed constant to the run-time of each iteration. So the sequence
```
    test i < n;
    func(i, n);
    i++;
```
has run-time \( T_i(n) + d \) for some positive constant \( d \). So the total run-time of the \( n \) iterations of the `for` statement is
\[
\sum_{i=0}^{n-1} (T_i(n) + d).
\]

Since
\[ T_i(n) + d \geq T_i(n) \geq c f_i(n), \forall n \geq n_0, \]
We get that the total run-time of the $n$ iterations of the \texttt{for} statement is

$$\sum_{i=0}^{n-1} (T_i(n) + d) \geq \sum_{i=0}^{n-1} cf_i(n), \forall n \geq n_0.$$ 

The initialization of $i$, the final test $i < n$, and the branch out of the \texttt{for} add a constant to this time. So the total run-time $S(n)$ of the algorithm satisfies

$$S(n) \geq c \sum_{i=0}^{n-1} f_i(n), \forall n \geq n_0,$$

and we get that $S(n)$ is

$$\Omega \left( \sum_{i=0}^{n-1} f_i(n) \right).$$

2. Suppose that \texttt{func1(n)} and \texttt{func2(n)} are algorithms that have runtimes $T(n)$ and $U(n)$, respectively. Suppose also that both $T(n)$ and $U(n)$ are $O(f(n))$.

Use the definition of big-$O$ to show that the runtime of the algorithm

\begin{verbatim}
func1(n);
func2(n);
\end{verbatim}

is also $O(f(n))$. If $T(n)$ and $U(n)$ are both $\Theta(f(n))$, is it also true that the runtime of this algorithm is $\Omega(f(n))$?

Since $T(n)$ and $U(n)$ are both $O(f(n))$, there exist positive constants $c_1, c_2, n_1, n_2$ such that

$$T(n) \leq c_1 f(n), \forall n \geq n_1, \text{ and } U(n) \leq c_2 f(n), \forall n \geq n_2.$$ 

So if we let $n_0 = \max\{n_1, n_2\}$, then

$$T(n) \leq c_1 f(n), \forall n \geq n_0, \text{ and } U(n) \leq c_2 f(n), \forall n \geq n_0.$$ 

Adding these formulas, we get that

$$T(n) + U(n) \leq c_1 f(n) + c_2 f(n) = (c_1 + c_2) f(n), \forall n \geq n_0.$$ 

Since the run-time of

\begin{verbatim}
func1(n);
func2(n);
\end{verbatim}

is also $O(f(n))$. If $T(n)$ and $U(n)$ are both $\Theta(f(n))$, is it also true that the runtime of this algorithm is $\Omega(f(n))$?
is $T(n) + U(n)$, we get that the runtime of this algorithm is $O(f(n))$.

Essentially the same arguments show that the run-time of

```
func1(n);
func2(n);
```

is $\Omega(f(n))$ if $T(n)$ and $U(n)$ are both $\Theta(f(n))$ and hence both $\Omega(f(n))$. 