1.5 Define an ADT for a set of integers (remember that a set may not contain duplicates). Your ADT should consist of the functions that can be performed on the sets, with each function defined in terms of its input and output.

Here are some possible operations.

(a) **Insert**. Input: a set $A$ and an integer $n$. Output: the set $A \cup \{n\}$.

(b) **Delete**. Input: a set $A$ and an integer $n$. Output: the set $A - \{n\}$, if $n \in A$, and the set $A$ together with a message to the effect that $n$ isn’t in $A$, if $n \notin A$.

(c) **Member**. Input: a set $A$ and an integer $n$. Output: true if $n \in A$, and false otherwise.

(d) **Union**. Input: two sets $A$ and $B$. Output: their union $A \cup B$. Note that implementing this may involve searching for duplicates.

(e) **Intersection**. Input: two sets $A$ and $B$. Output: their intersection $A \cap B$.

(f) **Set difference**. Input: two sets $A$ and $B$. Output: their difference $A - B$. That is, the output is

$$\{a \in A : a \notin B\}.$$

Note that in an object-oriented setting (e.g., Java), the set (or one of the sets in the case of union, intersection, and set difference) will probably be an implicit input, since the operation will most likely be a method on a set object.

1.13 A common problem for compilers and text editors is to determine if the parentheses (or other brackets) in a string are balanced and properly nested. For example, the string ”((((()))))” contains properly nested pairs of parentheses, but the string ”))()” does not; and the string ”(()” does not contain properly matching parentheses.

(a) Give an algorithm that returns true if a string contains properly nested and balanced parentheses, and false otherwise. Hint: At no time while scanning a legal string from left to right will you have encountered more right parentheses than left parentheses.

The algorithm should scan the string from left to right, keeping track of the number of unmatched left parentheses. If, at any time, the number is negative, there’s an unmatched right parenthesis and the algorithm can terminate with an error message. If, after the entire string has been scanned, the number of unmatched left parentheses is positive, there’s an unmatched left parenthesis and the algorithm should terminate with an error. If, after the entire string has been scanned, the number is 0, the string is OK. See Parens.java on the class website for details.
3.3 Arrange the following expressions by growth rate from slowest to fastest.

\[ 4n^2 \log_3(n) \quad 3^n \quad 20n \quad 2 \log_2(n) \quad n^{\frac{3}{4}} \]

Where does \( n! \) fit into this ordering?

\[ 2 < \log_3(n) \leq \log_2(n) < n^{\frac{3}{4}} < 20n < 4n^2 < 3^n. \]

You can directly calculate that \( 6! = 720, \ 3^6 = 729, \ 7! = 5040, \) and \( 3^7 = 2187. \) So for \( n \geq 7, \ n! > 3^n. \)

3.4 (a) Suppose that a particular algorithm has time complexity \( T(n) = 3 \cdot 2^n, \) and that executing an implementation of it on a particular machine takes \( T \) seconds for \( n \) inputs. Now suppose that we are presented with a machine that is 64 times as fast. How many inputs could we process on the new machine in \( T \) seconds?

(b) Suppose that another algorithm has time complexity \( T(n) = n^2, \) and that executing an implementation of it on a particular machine takes \( T \) seconds for \( n \) inputs. Now suppose that we are presented with a machine that is 64 times as fast. How many inputs could we process on the new machine in \( T \) seconds?

(c) Suppose that a third algorithm has time complexity \( T(n) = 8n, \) and that executing an implementation of it on a particular machine takes \( T \) seconds for \( n \) inputs. Now suppose that we are presented with a machine that is 64 times as fast. How many inputs could we process on the new machine in \( T \) seconds?

The key to working these problems is observing that the runtime \( T \) tells you roughly the number of instructions executed. For example, suppose \( T = 1000 \) seconds and the old machine will execute \( 10^6 \) instructions/second. Then in \( T \) seconds, the old machine will execute \( 10^6 \times 10^3 = 10^9 \) instructions, and the new machine will execute \( 64 \times 10^9 \) instructions. Thus, in any given period of time the new machine will do 64 times as much work as the old machine. Hence, if \( T_{\text{old}}(n) \) is the runtime of a program on the old machine, its runtime will be

\[ T_{\text{new}}(n) = \frac{1}{64} T_{\text{old}}(n) \]

on the new machine.

In the following discussion suppose that \( n_0 \) is the problem size on the old machine, and \( n_1 \) is the problem size on the new machine.

(a) We have \( T_{\text{old}}(n) = 3 \cdot 2^n, \) and \( T_{\text{old}}(n_0) = T. \) We want to know for what value of \( n = n_1 \) will we have \( T_{\text{new}}(n_1) = T. \) Using our formulas, we have

\[ 3 \cdot 2^{n_0} = \frac{1}{64} 3 \cdot 2^{n_1}. \]

So

\[ 64 \cdot 2^{n_0} = 2^{n_1}, \]

and since \( 64 = 2^6, \) we have

\[ 2^{6+n_0} = 2^{n_1}. \]

Thus, \( n_1 = 6 + n_0. \) If, for example, we could previously solve a problem of size 100, we can now solve a problem of size 106.
(b) Using reasoning similar to that in part (a), we have
\[ n_0^2 = \frac{1}{64} n_1^2. \]
So
\[ 64n_0^2 = n_1^2, \]
and
\[ 8n_0 = n_1. \]
So we can solve a problem 8 times larger on the new machine.

(c) Reasoning as in part (a) again, we have
\[ 8n_0 = \frac{1}{64} n_1, \]
and
\[ 64n_0 = n_1. \]
So we can solve a problem 64 times larger on the new machine.

3.6 Using the definitions of big-Oh and Omega, find the upper and lower bounds for the following expressions. Be sure to state appropriate values for \( c \) and \( n_0 \).

(a) \( c_1 n \)
(b) \( c_2 n^3 + c_3 \)

Since we’re interested in runtimes, I’ll assume that \( c_1 \) and \( c_2 \) are positive.

(a) \( c_1 n \) is \( O(n) \) and \( \Omega(n) \). For if we let \( c = c_1 \) and \( n_0 = 1 \), then we have that
\[ c_1 n \leq cn, \]
for all \( n > n_0 \), and
\[ c_1 n \geq cn, \]
for all \( n > n_0 \).

(b) \( c_2 n^3 + c_3 \) is \( O(n^3) \) and \( \Omega(n^3) \). First consider the case in which \( c_3 \geq 0 \). In this case we have
\[ c_2 n^3 \leq c_2 n^3 + c_3 \leq (c_2 + c_3)n^3. \]
for all \( n \). The second inequality holds because \( n \geq 1 \) and hence \( n^3 \geq 1 \). So \( c_3 n^3 \geq c_3 \).
If \( c_3 < 0 \), then clearly \( c_2 n^3 + c_3 \leq c_2 n^3 \), for all \( n \). So we only need to find \( c \) and \( n_0 \) so that
\[ cn^3 \leq c_2 n^3 + c_3, \]
for all \( n \geq n_0 \). To make things a little clearer, let \( c_4 = |c_3| \). Then
\[ c_2 n^3 + c_3 = c_2 n^3 - c_4, \]
and let \( T(n) = c_2 n^3 - c_4 \).
Now observe that if we choose \( c > 0 \) so that at some point \( n_1 > 0 \), we have
\[ cn_1^3 < c_2 n_1^3 - c_4. \]
then we’ll have
\[ cn^3 < c_2 n^3 - c_4, \]
for all \( n \geq n_1 \). We know this because of the following observations. Consider the function
\[ h(n) = c_2 n^3 - cn^3 - c_4 = (c_2 - c)n^3 - c_4. \]

- Then \( h(n) > 0 \) iff \( cn^3 < c_2 n^3 - c_4 \).
- So if \( cn_1^3 < c_2 n_1^3 - c_4 \), \( h(n_1) > 0 \).
- But since \( n_1 > 0 \), \( n_1^3 > 0 \), in order for \( h(n_1) > 0 \), we must have that \( c_2 - c > 0 \).
- But \( c_2 - c \) is constant, and if \( n_2 > n_1 \), then \( n_2^3 > n_1^3 \).
- So if \( n > n_1 \), we must have \( h(n) > 0 \).
- In other words, \( cn < c_2 n^3 - c_4 \), for all \( n > n_1 \).

So we only need to find a point \( n_1 \) where \( T(n_1) \) is positive, and then choose \( c \) so that \( cn_1^3 < T(n_1) \). But we know that \( T(n) \) is increasing (why?). So all we have to do is (say) solve the equation \( T(n_1) = 1 \). We can use basic algebra to get
\[ n_1 = \left( \frac{1 + c_4}{c_2} \right)^{\frac{1}{3}}. \]

Of course this may not be a whole number. So we just choose
\[ n_0 = \left[ \left( \frac{1 + c_4}{c_2} \right)^{\frac{1}{3}} \right]. \]

Note that since \( T(0) = -c_4 < 0 \), \( n_1 > 0 \) and hence \( n_0 \geq 1 \). Also note that since \( T(n) \) is increasing, we must have \( T(n_0) \geq 1 \). Now let
\[ c = \frac{1}{2n_0^3}. \]

Then \( c > 0 \) and
\[ cn_0^3 = \frac{1}{2n_0^3}n_0^3 = \frac{1}{2} < 1 = T(n_0). \]

So by our preceding observations we have
\[ cn^3 \leq c_2 n^3 + c_3 \]
for all \( n \geq n_0 \).

3.9 Determine big-Theta for the following code fragments in the average case. Assume that all variables are of type int. [We also assume that \( n \geq 1 \). Note that for these problems the number of statements executed is the same in both the best case, worst case, and average case.]

(a) \[ a = b+c; \]
\[ d = a+e; \]

This is two statements. Hence it’s \( \Theta(1) \).
(b) \[
\text{sum} = 0; \\
\text{for} \ (i = 0; \ i < 3; \ i++) \\
\quad \text{for} \ (j = 0; \ j < n; \ j++) \\
\quad \quad \text{sum}++; \\
\]

The inner loop is executed \(n\) times. So it executes \(an + b\) statements for some constants \(a\) and \(b\) with \(a > 0\). The outer loop executes 3 times. So when we add in the initialization of \(\text{sum}\), the entire code segment will execute

\[c(an + b) + d\]

statements for some constants \(c\) and \(d\) with \(c > 0\). Multiplying this out, we see that the number of statements executed is

\[en + f,\]

for some constants \(e\) and \(f\) with \(e > 0\). We saw in class that this function is \(\Theta(n)\).

(c) \[
\text{sum} = 0; \\
\text{for} \ (i = 0; \ i < n*n; \ i++) \\
\quad \quad \text{sum}++; \\
\]

Let \(m = n^2\). Then the number of statements executed is \(am + b\) for some constants \(a\) and \(b\) with \(a > 0\). So this is \(\Theta(m)\), and since \(m = n^2\), we see that this is \(\Theta(n^2)\).

(d) Assume array \(A\) contains \(n\) values.

\[
\text{for} \ (i = 0; \ i < n; \ i++) \{ \\
\quad \text{for} \ (j = 0; \ j < n; \ j++) \\
\quad \quad A[i] = \text{DSutil.random}(n) \quad // \text{random takes constant time} \\
\quad \quad \text{sort}(A, n) \quad // \text{sort takes } n \log n \text{ time} \\
\}\]

The inner loop executes \(an + b\) statements for constants \(a, b\) with \(a > 0\). The call to \text{sort} takes time \(n \log n\). So the total number of statements executed in the body of the loop is

\[cn \log(n) + dn + e,\]

for constants \(c, d, e\) with \(c > 0\). Since the outer loop is executed \(n\) times, the total number of statements executed is

\[fn(cn \log(n) + dn + e) + g\]

Multiplying this out, we see that the code is \(\Theta(n^2 \log(n))\).

Note that this argument can be considerably simplified by using the rules on page 56 of the text (which also apply to big-\(\Omega\) and big-\(\Theta\)). The inner loop is \(\Theta(n)\). The sort is \(\Theta(n \log n)\). So the body of the outer loop is \(\Theta(n \log n)\) (by rule 3). Thus, by rule 4, the code is \(\Theta(n^2 \log n)\).