**Terminology:**

- **problem** - question we seek an answer for, e.g., "what is the largest item in a list/array?"
- **parameters** - variables with unspecified values
- **problem instance** - assignment of values to parameters, i.e., the specific input to the problem

```
myList:  0 1 2 3 4 5 6  
       5 10 2 15 20 1 11
largest: ?

n:  7
(number of elements)
```

- **algorithm** - step-by-step procedure for producing a solution
- **basic operation** - fundamental operation in the algorithm (i.e., operation done the most) Generally, we want to derive a function for the number of times that the basic operation is performed related to the **problem size**.
- **problem size** - input size. For algorithms involving lists/arrays, the problem size is the number of elements ("n").

```python
import time
def main():
    aList = range(1,1000001)
    start = time.time()
    sum = sumList(aList)
    end = time.time()
    print "Time to sum the list was %.3f seconds" % (end-start)

def sumList(myList):
    """Returns the sum of all items in myList""
    total = 0
    for item in myList:
        total = total + item
    return total

main()
```

**Execution time, T(n), of sumList = (time to perform code which is done once) + n x (time to perform loop once).** Computers of different speeds would not effect this formula, but only the constants:

- time to perform code which is done once, and
- time to perform loop once

The big-oh and big-theta definitions that follow provide a computer independent measure of an algorithm’s performance.
**Big-oh Definition** - asymptotic upper bound
For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant $c$ and some nonnegative integer $N$ such that for all $n \geq N$,
$$g(n) \leq c \times f(n).$$

Execution Time

```
T(n) = c_1 + c_2 \cdot n = 100 + 10 \cdot n \text{ is } O(n).
```

"Proof": Pick $c = 110$ and $N = 1$, then $100 + 10 \cdot n \leq 110 \cdot n$ for all $n \geq 1$.

100 + 10n ≤ 110n  
100 ≤ 100n  
1 ≤ n

**Problem with big-oh:**
If $T(n)$ is $O(n)$, then it is also $O(n^2)$, $O(n^3)$, $O(2^n)$, ..., since these are also upper bounds.

**Omega Definition** - asymptotic lower bound
For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant $c$ and some nonnegative integer $N$ such that for all $n \geq N$,
$$g(n) \geq c \times f(n).$$

Execution Time

```
``
2. Let $T(n) = c_1 + c_2 n = 100 + 10 n$. Show that $100 + 10 n$ is $\Omega(n)$.

"Proof": We need to find a $c$ and $N$ so that the definition is satisfied, i.e., $100 + 10 n \geq c n$ for all $n \geq N$.

What $c$ and $N$ will work?

I would pick $c = 10$, so we need to find an $N$ such that $100 + 10 n \geq 10 n$ for all $n \geq N$. By picking $c = 10$, this will be easy since

$$100 + 10 n \geq 10 n$$
$$100 + 10 n - 10 n \geq 10 n - 10 n$$
$$100 \geq 0$$

which is always true for an value of $n$.

I'll pick $N = 1$.

(Note: Other values of $c$ would work too, like $c = 9, c = 8, ..., c = 1$)

**Theta Definition** - asymptotic upper and lower bound, i.e., a "tight" bound or "best" big-oh

For a given complexity function $f(n)$, $\theta(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constants $c$ and $d$ and some nonnegative integer $N$ such that for all $n \geq N$,

$$d \times f(n) \leq g(n) \leq c \times f(n).$$

3. Suppose that you have an $\theta(n^2)$ algorithm that required 10 seconds to run on a problem size of 1000. How long would you expect the algorithm to run on a problem size of 10,000?

A $\theta(n^2)$ algorithm implies execution time, $T(n) = c_1 n^2 + (\text{slower growing terms like } c_2 n + c_3)$, where $c_1$, $c_2$, $c_3$ are constants determined by the speed of your computer.

For large $n$'s, $T(n) \approx c_1 n^2$. We are given that the problem requires 10 seconds to run on a problem size of 1000, so $T(1000) = 10 \text{ seconds } \approx c_1 \times 1000^2$, solving for $c_1 = 10 \text{ seconds } / 1000^2 = 10 \text{ seconds } / 10^6 = 1 \text{ second } / 10^5 = 10^{-5} \text{ seconds}$. For a larger size problem, $c_1$ remains constant on the same computer.

Thus, $T(10,000) \approx c_1 \times 10000^2 = 10^{-5} \times 10000^2 = 10000 \text{ seconds}$.

The problem size got 10 times bigger -- it went from 1000 to 10,000. However, the execution time got 100 times longer -- it went from 10 seconds to 1000 seconds, since the algorithm was $\theta(n^2)$. 
4. Analyze the below algorithm to determine its theta notation, \( \theta(\cdot) \).

```python
i = n
while i > 0:
    for j in xrange(n):
        # something of \( O(1) \)
    i = i / 2
# end while
```

The outer while-loop executes \( \theta(\log_2 n) \) times. IMPORTANT: Any time \( n \) is repeatedly halved down to 1 or 0, the loop executes \( \theta(\log_2 n) \) times. Similarly, any time a variable \( i \) starts at 1 and is repeatedly doubled to reach the value \( n \), the loop executes \( \theta(\log_2 n) \) times.

For each iteration of the outer-loop, the inner-loop executes \( n \) times, so the whole algorithm is \( \theta(n \log_2 n) \).

Pronounced as “theta of \( n \log n \)”.

5.

```python
def linearSearch(target, aList):
    
    """Returns the index position of target in aList or -1 if target
    is not in aList""
    for position in xrange(len(aList)):
        if target == aList[position]:
            return position
    return -1
```

a) For sequential search, what is the best-case time complexity \( B(n) \)? In the best-case the target value is in position 0 of the list, so the for-loop stops after one iteration. This will take a constant amount of time so the best-case time complexity is \( \theta(1) \), which is “constant time”.

b) For sequential search, what is the worst-case time complexity \( W(n) \)? In the worst-case the target is NOT in the list so the for-loop must loop through all \( n \) list positions. Thus, the worst-case time complexity \( \theta(n) \), which is “linear”.

c) If the probability of a successful sequential search is \( p \), then what is the probability on an unsuccessful search? \( 1-p \)

d) If the probability of a successful sequential search is \( p \), then what is the probability of finding the target value at a specific index in the array? All thinks being equal, we can spread the probability \( p \) across all \( n \) positions in the list, so the probability of a target a specific index would be \( p/n \).

e) For each position in the list of size \( n \), how many comparisons would be needed for a successful search?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>n-1</th>
<th>All unsuccessful searches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># compares:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>probability:</td>
<td>( p/n )</td>
<td>( p/n )</td>
<td>( p/n )</td>
<td></td>
<td>( p/n )</td>
<td>(1-p)</td>
</tr>
</tbody>
</table>
e) Write a summation for the average number of comparisons.

\[
\frac{p}{n}1 + \frac{p}{n}2 + \frac{p}{n}3 + \ldots + \frac{p}{n}n + (1 - p)n
\]

\[
= \frac{p}{n}(1 + 2 + 3 + \ldots + n) + (1 - p)n
\]

\[
= \frac{p}{n} \frac{n(n+1)}{2} + (1 - p)n
\]

\[
= \frac{p}{2} (n + 1) + (1 - p)n
\]

\[
= \frac{p}{2} n + \frac{p}{2} + n - pn
\]

\[
= n - \frac{p}{2} n + \frac{p}{2}
\]

\[
= n(1 - \frac{p}{2}) + \frac{p}{2}
\]

f) What is the average time complexity, \( A(n) \)?

\( n(1 - \frac{p}{2}) + \frac{p}{2} \) which is \( \Theta(n) \) since any given \( p \) causes \( (1 - \frac{p}{2}) \) to be a constant.