Data Structures - Test 1

Question 1. (10 points) Determine the theta notation $\Theta()$ for the following Python code.

\[
\begin{align*}
\text{for } i \text{ in } \text{xrange}(n): & \quad \text{loops } n \text{ times} \\
& \quad \text{for } j \text{ in } \text{xrange}(i): \\
& \quad \quad \text{sum} = i + j \\
& \quad \quad \text{# end for } j \\
& \quad \text{# end for } i \\
\end{align*}
\]

\[
= \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \quad \text{i.e. } \Theta(n^2)
\]

Question 2. (10 points) Suppose a $\Theta(n^2)$ algorithm takes 10 seconds when $n = 1,000$. How long would you expect the algorithm to run when $n = 10,000$?

\[\Theta(n^2) \text{ means } \text{Exec. time } T(n) \approx cn^2\]

\[
T(1000) = c \times 1000^2 = 10 \text{ sec} \quad c = \frac{10}{1000^2} = \frac{10}{(10^3)^2} = 10^{-5}
\]

\[
T(10000) = c \times 10000^2 = (10^{-5}) \times (10^4)^2 = 10^{-5} \times 10^8 = 10^3 \text{ sec}
\]

\[
= 1000 \text{ sec}
\]

Question 3. (15 points) For the two implementations of fibonacci given below, explain why fibA is so much slower than fibB.

\[
\text{def fibA(n):}
\quad \text{if } n \leq 1:
\quad \quad \text{return } n
\quad \text{else:}
\quad \quad \text{return fibA(n-1) + fibA(n-2)}
\]

\[
\text{def fibB(n):
\quad fibs = [0, 1]
\quad \text{for } i \text{ in } \text{range}(2, n+1):
\quad \quad \text{fibs.append(fibs[i-1]+fibs[i-2])}
\quad \text{return fibs[n]}
\]

The fibA implementation solves the same smaller problems over and over during the recursion. For example, fibA(n-2) in:

\[
\text{fibA}(n) \quad \quad \text{fibA}(n-1) \quad \quad \text{fibA}(n-2)
\]

fibB however solves each problem once and stores the answer in the list fibs so these answers can be looked up when needed again.
Question 4. (5 points) What is the difference between unit testing and integration testing?

Unit testing you test each module (e.g. class, function, etc.) separately. In integration testing you put these units together and test them together.

Question 5. (15 points)

a) In the following recursive binary search code, what would be a precondition on the binarySearch function?

The list myList is in ascending order.

```python
def binarySearch(myList, target):
    """Returns the position of the target in myList or -1 if not found""
    if myList == []: return -1
    elif target < myList[0]:
        return binarySearch(myList[1:], target)
    elif target > myList[-1]:
        return binarySearch(myList[:-1], target)
    else:
        midpoint = len(myList) // 2
        if myList[midpoint] == target:
            return midpoint
        elif target < myList[midpoint]:
            return binarySearch(myList[:midpoint], target)
        else:
            return binarySearch(myList[midpoint+1:], target)

b) Show the output of the following program which calls binarySearch. (INCLUDE the output of the debugging print statement in the binarySearchHelper function)

```aList = [10, 20, 30, 40, 50, 60, 70, 80]```

print "The list is: ", aList

target = 50
location = binarySearch(aList, target)
if location == -1:
    print target, "NOT found"
else:
    print target, "FOUND at index", location
```

Output of the above program which calls binarySearch:

The list is: [10, 20, 30, 40, 50, 60, 70, 80]
first is 0 last is 7
first is 4 last is 7
first is 4 last is 7
50 FOUND at index 4"
Question 6. (25 points) Consider the following AltStack class that uses an Array to store the items in the stack. The “top” item on the stack is always stored at index 0. (NOTE: this is different from the ArrayStack class of section 14.4)

a) Complete the theta notation \( \Theta() \) for each stack methods of the above AltStack implementation: (Let us define "n" as the # items in the stack)

<table>
<thead>
<tr>
<th>Theta notation</th>
<th>_init_ (constructor)</th>
<th>push(item)</th>
<th>pop()</th>
<th>peek()</th>
<th>len()</th>
<th>isEmpty()</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(1) )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td></td>
</tr>
</tbody>
</table>

b) Assume that the array size DOES NOT grow during the push method, but has a fixed physical capacity from the \_init\_ constructor. What would be the precondition on the push method.

(\$) The stack is not full.

(\$) The stack is not full.

c) Write the code for the push method of the AltStack class.

```python
def push(self, newItem):
    # "Inserts newItem at the top of stack."
    for index in xrange(len(self._items), 0, -1):
        # self._items[index] = self._items[index-1]
        self._items[index] = self._items[index-1]

    self._items[0] = newItem
    self._size += 1
```
Question 7. (20 points) Consider the following `AltLinkedStack` class which uses the `Node` class (from the text and listed above) to dynamically create storage for a new item added to the stack. Conceptually, an `AltLinkedStack` object would look like the below picture. (NOTE: this is different from the `LinkedStack` class in section 14.4)

a) Complete the theta notation $\Theta()$ for each stack method of the above `AltLinkedStack` implementation: (Let us define "n" as the # items in the stack)

<table>
<thead>
<tr>
<th>Method</th>
<th>Theta notation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>push(item)</code></td>
<td>$\Theta(1)^2$</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>$\Theta(n)^2$</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>len()</code></td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

b) Write the code for the `push` method of the `AltLinkedStack` class.

```python
def push(self, newItem):
    """Inserts newItem at the top of stack."""
    temp = Node(newItem) + 3
    if self._size == 0:
        self._bottom = temp
    else:
        self._top.next = temp
    self._top = temp + 2
    self._size += 1 + 2
```