Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes. You may use one 8.5x11 sheet of paper with hand-written notes (no photocopies or laser printed notes). No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 8 questions with points as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name ___________________________ Solution ___________________________

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

______________________________

signed
1. [12 points] Circle TRUE or FALSE according to whether the statement is true or false:

a) $1000 - 7n + 2n^2$ is $\Theta(n)$  **TRUE**  **FALSE**

b) $27n + 2 \log n$ is $\Theta(n)$  **TRUE**  **FALSE**

c) $2n^2 + 3n \log n$ is $\Theta(n \log n)$  **TRUE**  **FALSE**

d) $\sum_{i=1}^{n} \left( 4 + \sum_{j=1}^{i} 3j \right)$ is $\Theta(n^2)$  **TRUE**  **FALSE**

2. [12 points] Assuming that each assignment, arithmetic operation, comparison, and array index costs one unit of time, analyze the complexity of the following code fragment that transposes an $n \times n$ matrix, and give an exact count complexity function $T(n)$:

```java
for (int Row = 0; Row < n; Row++) {
    for (int Col = Row + 1; Col < n; Col++) {
        int tmp = Mtx[Row][Col];
        Mtx[Row][Col] = Mtx[Col][Row];
        Mtx[Col][Row] = tmp;
    }
}
```

Applying the analysis logic from the notes, we write summations for the two loops. Let $R$ and $C$ be the Row and Col counters, respectively. Then we get:

\[
T(n) = 1 + \sum_{R=0}^{n-1} \left( 2 + 1 + 1 + \sum_{C=R+1}^{n-1} (1 + 1 + 3 + 5 + 3 + 1) + 1 \right) + 1
\]

\[
= 2 + \sum_{R=0}^{n-1} \left( 5 + \sum_{C=R+1}^{n-1} 13 \right)
\]

\[
= 2 + \sum_{R=1}^{n} \left( 5 + \sum_{C=R+1}^{n-1} 13 \right)
\]

\[
= 2 + \sum_{R=1}^{n} (5 + 13(n - R - 1))
\]

\[
= 2 + \sum_{R=1}^{n} (5 + 13n - 13R - 13)
\]

\[
= 2 + 5n + 13n^2 + \frac{13n(n+1)}{2} + 13n
\]

\[
= 2 + 5n + \frac{39n^2}{2} + \frac{39n}{2}
\]
3. [10 points] A programmer must choose a data structure to store N elements, which will be supplied to the program in ascending (sorted) order. Give a big-Θ estimate for the number of operations required to create the structure if the programmer uses:

a) a sorted array of dimension N, inserting the N elements as they are supplied.

Since the elements are provided in sorted order, you would insert each element at the end of the list (in the first unused cell), so each insertion only costs one assignment and array indexing – \( \Theta(1) \). That's a constant cost per element, so the total cost to add all N elements would be \( N \times \Theta(1) = \Theta(N) \).

b) an AVL tree, inserting the N elements as they are supplied.

Each insertion adds a leaf, so the search cost for each insertion is determined by the depth of the tree. AVL trees guarantee a maximum depth of \( \log K \), where K is the number of nodes. Being a little sloppy, each search costs \( \Theta(\log N) \). The actual physical insertion is constant cost, and the cost of rebalancing is at worst \( \Theta(\log N) \), so the cost of each insertion is no worse than \( \Theta(\log N) \). Doing N insertions would thus have cost \( \Theta(N \log N) \).

4. [10 points] Using the relationship between big-Θ and limits, prove that \( T(N) = 7N^2 + N \log N \) is \( \Theta(N^2) \).

The theorem (Theorem 8 from the course notes) says that if the following limit is positive and finite, then \( T(N) \) is \( \Theta(N^3) \):

\[
\lim_{n \to \infty} \frac{T(n)}{n^2} = \lim_{n \to \infty} \frac{7n^2 + n \log n}{n^2} \quad \text{algebra}
\]

\[
= \lim_{n \to \infty} \left( 7 + \frac{\log n}{n} \right) \quad \text{1'Hopital's Rule}
\]

\[
= 7 + \lim_{n \to \infty} \frac{1}{n}
\]

\[
= 7 + 0
\]

\[
= 7
\]

Since \( 0 < 7 < \infty \) (i.e. 7 is a positive constant), then by the theorem, we can conclude that \( T(N) \) is \( \Theta(N^2) \).
5. [12 points] Consider the recursive function definition: 
\[ G(n) = \begin{cases} 
1 & n = 0 \\
2 \cdot G(n - 1) & n > 0 
\end{cases} \]

Use induction to prove that for all \( n \geq 0 \), \( G(n) = 2^n \).

**Proof:**

Let \( S \) be the set of all integers, \( n \geq 0 \), such that \( G(n) = 2n \).
We will prove that for all \( n \geq 0 \), \( n \in S \).

**Base case:** if \( n = 0 \), we have \( G(0) = 1 \) by definition, and \( 2^0 \) is 1, so \( G(0) = 2^0 \).

**Inductive assumption:** Assume that for some \( k \geq 0 \), \( G(k) = 2^k \).

**Induction step:** Consider \( G(k + 1) \). Since \( k + 1 \geq 1 \), the definition of \( G(n) \) implies that \( G(k + 1) = 2 \cdot G(k) \). The inductive assumption implies \( G(k) = 2^k \). Putting it together, we have:

\[
G(k + 1) = 2 \cdot G(k + 1 - 1) \quad \text{by the definition of } G(n) \\
= 2 \cdot G(k) \quad \text{by algebra} \\
= 2 \cdot 2^k \quad \text{by the Inductive assumption} \\
= 2^{k+1} \quad \text{by algebra}
\]

**Conclusion:** Therefore, by induction, \( G(n) = 2^n \) for all \( n \geq 0 \). Q.E.D.
For the next question, consider the partial BST and binary node template interfaces given below:

```cpp
template <typename T> class BinNodeT {
public:
    Data Element;
    BinNodeT<T>* Left;
    BinNodeT<T>* Right;

    BinNodeT();
    BinNodeT(const T& E, BinNodeT<T>* L, BinNodeT<T>* R);
    ~BinNodeT();
};
```

```cpp
template <typename T> class BST {
private:
    BinNodeT<T>* Root;

public:
    BST();
    bool Insert(const T& Elem);
    T* Find(const T& D);
    bool Delete(const T& D);
    ~BST();
};
```

6. [12 points] Write a BST member function `deleteMax()` which conforms to the interface and description below.

// The function deletes the maximum value from the BST, as efficiently as possible.
// The function may not call any other member functions of the BST template, and
// must not use recursion.
// You may assume that the BST does not contain any duplicate values.

```cpp
// template <typename T> void BST<T>::deleteMax() {

if ( Root == NULL ) return; // handle empty tree

BinNodeT<T>* Parent = Root; // will move this to parent of right-most node

if ( Root->Right == NULL ) { // root node is right-most
    Root = Root->Left; // reset Root to left subtree
    delete Parent; // deallocate old root node
    return; // done
}

while ( Parent->Right->Right != NULL ) { // walk to parent of right-most node
    Parent = Parent->Right;
}

BinNodeT<T>* toDel = Parent->Right; // save pointer to right-most node
Parent->Right = Parent->Right->Left; // reset parent's child pointer
delete toDel; // deallocate right-most node

return;
}
```

Note: the key is that the maximum value will always be in the right-most node in the BST. So, we need to find the parent of that node, and perform a simple deletion case (since the right-most node has at most one subtree). There are two special cases: an empty tree, and one in which the root node has no right subtree.
7. [22 points] Consider the AVL tree (Tree T) shown below.

Tree T:

a) [11 points] Starting with Tree T as shown at the top of this page, draw the AVL tree that would result from inserting 45 into Tree T.

After deleting 41

After deleting 59

Both cases are balanced after BST delete, no AVL rotations needed
8. [10 points] Recall the homework problem about devising a test to determine whether a given binary tree has the BST property. Consider the following proposed solution, which would be passed a pointer to the root of the binary tree:

```cpp
template <typename T> bool isBST( BinNodeT<T>* Root ) {
    if ( Root == NULL ) return true;
    if ( (Root->Left != NULL) && (Root->Element <= Root->Left->Element) )
        return false;
    if ( (Root->Right != NULL) && (Root->Element > Root->Right->Element) )
        return false;
    return ( isBST(Root->Left) && isBST(Root->Right) );
}
```

Is the solution correct? If not, give an example of a binary tree for which it would return the incorrect result.

Note there are two cases for possible logical errors: the function may say that a BST is not a BST, or it may say that a non-BST is a BST. You must consider both possibilities.

The given function essentially performs a pre-order traversal, comparing the value in each node to the values in its children (if it has children). The traversal logic is correct. The comparisons are correct, as far as they go. Some of you convinced yourselves the element comparisons in the second and third if conditions were incorrect --- they are not.

But… the given function performs ONLY a local test at each node. That is not enough. The given function will return true if given the root pointer of the following binary tree, even though it is not a BST:

```
   50
   / \
  75  
 /   \
40   80
```

Many other examples exist.

However, the given function will deal correctly with any true BST it's given.