Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. 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 9 questions, priced 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
printed

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×n matrix, and give an exact count complexity function T(n):

```
for (int Row = 0; Row < n; Row++) {        // 1 before, 2 each pass, 1 after

   for (int Col = 0; Col < Row; Col++) {   // 1 before, 2 each pass, 1 after

      int tmp = Mtx[Row][Col];             // 3 (2 for indexing, 1 assign)

      Mtx[Row][Col] = Mtx[Col][Row];       // 5 (4 for indexing, 1 assign)

      Mtx[Col][Row] = tmp;                 // 3 (2 for indexing, 1 assign)

   }
}
```

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:

$$T(n) = 1 + \sum_{R=0}^{n-1}\left(2 + 1 + \sum_{C=0}^{R-1} 13 + 1\right) + 1$$

$$= 2 + \sum_{R=0}^{n-1}\left(4 + \sum_{C=0}^{R-1} 13\right) =$$

$$= 2 + \sum_{R=1}^{n}\left(4 + \sum_{C=1}^{R} 13\right) =$$

$$= 2 + \sum_{R=1}^{n}(4 + 13R)$$

$$= 2 + 4n + \frac{13n(n+1)}{2}$$

$$= 2 + \frac{21}{2}n + \frac{13}{2}n^2$$

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.

   Obviously, 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. That's a constant cost per element, so the total cost would be Θ(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 Θ(logN). The actual physical insertion is constant cost, and the cost of rebalancing is at worst Θ(logN), so the cost of each insertion is no worse than Θ(logN). Doing N insertions would thus have cost Θ(N logN).

---

4. [12 points] Consider the recursive function definition: $G(n) = \begin{cases} 1 & n = 0 \\ 2 * G(n-1) & n > 0 \end{cases}$

   Use induction to <u>prove</u> that for all n ≥ 0, $G(n) = 2^n$.

   <u>Base case</u>: if n = 0, we have G(0) = 1 by definition, and 2^0 is 1, so $G(0) = 2^0$.

   <u>Inductive assumption</u>: for some k ≥ 0, $G(k) = 2^k$.

   <u>Induction step</u>: Consider G(k + 1). Since k + 1 ≥ 1, the definition of G(n) implies that G(k + 1) = 2 * G(k). But the inductive assumption implies $G(k) = 2^k$. Putting it together, we have:

$$G(k + 1) = 2 * G(k) = 2 * 2^k = 2^{k+1}$$

   Therefore, by induction, $G(n) =2^n$ for all n ≥ 0.

QED

5. [10 points] Using the relationship between big-Θ and limits, <u>prove</u> that T(N) = $7N^2$ + N log N is $\Theta(N^2)$.

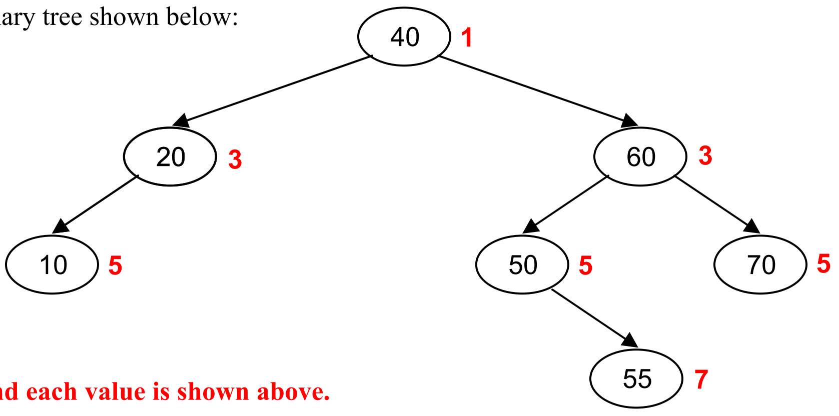
The theorem says that if the following limit is positive and finite, then T(N) is $\Theta(N^2)$:

$$
\begin{aligned}
\lim_{n\to\infty}\frac{T(N)}{N^2} &= \lim_{n\to\infty}\frac{7N^2+N\log N}{N^2} && \text{algebra}\\
&= \lim_{n\to\infty}\left(7+\frac{\log N}{N}\right) && \text{l'Hopital's Rule}\\
&= 7+\lim_{n\to\infty}\frac{1/N}{1}\\
&= 7+0\\
&= 7
\end{aligned}
$$

6. [10 points] Suppose a data structure holds N different data values: d_1, d_2, … d_N. Assume that if a search is performed then each data value is just as likely as any other to be the target of the search. If x is a data value, let C(x) be the number of comparisons performed in searching the data structure for x.

The average search cost is then defined to be: $\displaystyle \frac{1}{N}\sum_{k=1}^{N} C(d_k)$

Calculate the average search cost for the binary tree shown below:



The number of comparisons needed to find each value is shown above.

The total is 29.

So, the average is 29/7 or about 4.14.

For the next three questions, consider the partial BST and binary node template interfaces given below:

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

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

```
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();
};
```

7. [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.
//
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>* toKill = Parent->Right;         // save pointer to right-most node
   Parent->Right = Parent->Right->Left;         // reset parent's child pointer
   delete toKill;                               // 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.

8. [12 points] Write a BST member function which conforms to the description and interface given below. You may write one or more recursive helper functions if you like.

```
// The function deletes all the leaves from the BST and returns a vector containing
// the values that were in the leaves.
//
template <typename T> vector<T> BST<T>::pickLeaves( ) {

   vector<T> Trimmings;            // vector to hold contents of leaves

   if ( Root != NULL )             // nothing to do if tree is empty
      Picker(Root, Trimmings);     // trim off the leaves and get their contents

   return Trimmings;               // return leaf contents
}


template <typename T> void Picker(BinNodeT<T>*& sRoot, vector<T>& Trimmings) {

   if ( sRoot == NULL ) return;    // empty subtree, nothing to do

   if ( sRoot->Left == NULL && sRoot->Right == NULL ) {  // we're at a leaf!

      Trimmings.push_back(sRoot->Element);               // save contents
      delete sRoot;                                      // deallocate leaf node
      sRoot = NULL;                                      // fix pointer in parent
      return;                                            // we're done here
   }

   Picker(sRoot->Left, Trimmings);         // process left subtree
   Picker(sRoot->Right, Trimmings);        // process right subtree
}
```

Notes:

- **The node pointer is passed to the helper function by reference so the helper can properly update the pointer (which is a child pointer in the parent) when a leaf is removed.**
- **The test for an empty tree in the first function is, strictly speaking, unnecessary since the helper function does its own NULL test. But, putting the test in the first function will eliminate the overhead of a function call if the initial tree is empty.**

9. [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:

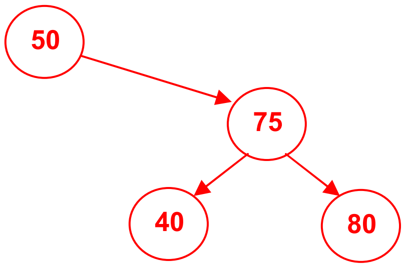
```
template <typename T> bool isBST( BinNodeT<T>* Root ) {

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

Note there are two cases for <u>possible</u> 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:



Many other examples exist.

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