This test has 11 pages, be sure your test has them all. Do NOT spend too much time on one question.
In writing code you do not need to worry about specifying the proper `#include` header files. Assume that all the header files we've discussed are included in any code you write.
Some common recurrences and their solutions (the letters are used in one of the problems).

\[
\begin{align*}
A & \quad T(n) = T(n/2) + O(1) \quad O(\log n) \\
B & \quad T(n) = T(n/2) + O(n) \quad O(n) \\
C & \quad T(n) = 2T(n/2) + O(1) \quad O(n) \\
D & \quad T(n) = 2T(n/2) + O(n) \quad O(n \log n) \\
E & \quad T(n) = T(n-1) + O(1) \quad O(n) \\
F & \quad T(n) = T(n-1) + O(n) \quad O(n^2)
\end{align*}
\]

The declaration for binary search tree nodes on this test is:

```c
struct TreeNode
{
    int info;
    TreeNode * left;
    TreeNode * right;
    TreeNode(int val, TreeNode * lptr, TreeNode * rptr)
    : info(val), left(lptr), right(rptr)
    { }
};
```
PROBLEM 1: (Recurring Nightmares)

Part A (10 points)
For each algorithm/code fragment write the letter of the recurrence (from the front of this test) that applies in the worst-case for implementing the algorithm recursively. Some letters may be used more than once and some not at all.

1. binary search in a sorted vector
2. merge sort
3. quicksort
4. sequential search
5. printing values in a tree using a postorder traversal

Part B (7 points)
The function maxstack below correctly returns the largest element in its stack parameter. Write a recurrence for this function, explaining how each of the six labelled lines of the function maxstack contributes to the recurrence. What is the big-Oh solution to the recurrence?

```cpp
int max(int x, int y)
// post: returns largest of x and y
{
    if (x > y) return x;
    else return y;
}

int maxstack(tstack<int>& s)
// pre: s contains at least one element
// post: returns largest element in s, leaves elements in same order
{
    if (s.size() == 1) return s.top(); // line 1
    else {
        int topVal; // line 2
        s.pop(topVal); // line 3
        int maxRest = maxstack(s); // line 4
        s.push(topVal); // line 5
        return max(maxRest, topVal); // line 6
    }
}
```
Problem 2: (Game, set, match (12 points))

In class we studied the function below which returns the intersection of two sets. The intersection is the elements that are in both sets.

```cpp
tset<string> makeIntersection(const tset<string>& a, const tset<string>& b)
    // post: create and return a set that is the intersection of a and b
    { tset<string> result;
      Iterator<string> * it = a.makeIterator();
      for(it->Init(); it->HasMore(); it->Next()) {
        string s = it->Current();
        if (b.contains(s)) { result.insert(s); }
      }
      return result;
    }
```

Part A (6 points)
When calling `makeIntersection(x,y)` to create the intersection of sets `x` and `y`, which should be passed as the first parameter if the sets are drastically different in size, i.e., one has a few hundred elements and the other has millions. Why?

Part B (6 points)
Write a function that determines if two sets are disjoint (i.e., contain no elements in common). You can call `makeIntersection` or use it as a model for the code you write.

```cpp
bool areDisjoint (const tset<string>& a, const tset<string>& b)
    // post: returns true if a and b contain no common elements in common
    //        returns false if there is some element in common to a and b
    {
    }"
A vector contains the names of the people in an a specific undergraduate class/year: first year, sophomore, junior, senior. A vector of four such vectors contains the names of all undergraduates enrolled (assume the senior vector includes everyone not in the first three vectors) with the first year names in the first vector which has index zero and the senior names in the last vector whose index is three.

The function `sortNames` below creates a vector of all undergraduates sorted by name from such a vector of vectors.

**Part A (4 points)**

Assuming there are $N$ people in each undergraduate year/class, what is the big-Oh complexity of the code below, and why? Account for all the code in your reasoning.

```cpp
void sortNames(tvector<tvector<string> >& allClasses,
               tvector<string>& everyone)
// pre: allClasses contains 4 vectors, one per undergraduate year
// post: everyone contains the name of every student, sorted
{
    everyone.clear();
    for(int year=0; year < 4; year++) {
        for(int k=0; k < allClasses[year].size(); k++){
            everyone.push_back(allClasses[year][k]);
        }
    }
    sort(everyone.begin(), everyone.end());
}
```

**Part B (4 points)**

If the call

```cpp
sort(everyone.begin(), everyone.end());
```

is changed to/replaced by

```cpp
stable_sort(everyone.begin(), everyone.end());
```

then the complexity won’t change, but the order of the names in vector `everyone` will be different after the change. Describe the difference(s), be terse.
Part C (6 points)
Suppose that instead of four different years/classes of student there were $M$ different categories of student — each category containing $N$ students. There are two methods for generating a sorted list of students: use the method/code in `sortNames` above for sorting, replacing 4 with $M$; or use the following proposed method.

1. Sort each of the $M$ vectors using the standard `sort` function.
2. Merge the $M$ sorted vectors into a new vector to make a sorted vector of $NM$ students.

With the proper analysis of the two methods/algorithms (using big-Oh and reasoning about them) make a claim that one of the algorithms is faster than the other. Which is faster and why? Use big-Oh in answering this question. You may find it useful to recall that $\log(ab) = \log(a) + \log(b)$. 

Part D (8 points)
You’re given the task of finding the top $k$ numbers in a list of $N$ numbers. Three methods are proposed for solving this problem, each is correct.

1. Sort the elements using an efficient sort and then choose the top $k$.
2. Build a heap from the elements (this can be done in linear time) and then call deletemin $k$ times.
3. Make $k$ passes over the data finding the top element each time (taking steps to ensure the top element won’t be chosen again).

If $k$ is very small (under 10) and $N$ is very large (over one million) argue that one method is clearly superior to the others and why.

If $k == N$ argue that one method is superior to the others and why (assume $N$ is roughly 100,000).
PROBLEM 4:  (Nests in Trees)

In class we looked at the function nodeCount below.

```c
int nodeCount(TreeNode * tree)
// post: returns number of nodes in tree, 0 if tree is NULL/0
{
    if (tree == 0) return 0;
    return 1 + nodeCount(tree->left) + nodeCount(tree->right);
}
```

The function copyCount below returns a copy of its TreeNode parameter but where each node's value is replaced by the count of the number of nodes in the tree rooted at t. For example, if t is the tree shown on the left below, the call copyCount(t) returns the tree shown on the right.

```
TreeNode * copyCount (TreeNode * t)
// post: returns "counted" copy of t
{
    if (t == 0) return 0;
    else {
        return new TreeNode(nodeCount(t),
                            copyCount(t->left),
                            copyCount(t->right));
    }
}
```

**Part A (4 pts)**
The average case complexity of copyCount is not $O(n)$; what is the big-Oh complexity and why (briefly).
Part B (6 points)
Rewrite copyCount so that it runs in \(O(n)\) time. Your function should use a helper function as described below. The helper function should make two recursive calls and do \(O(1)\) other work.

```c
TreeNode * copyCount (TreeNode * t)
// post: returns counted copy of t
{
    int numNodes;
    return copyCountHelper(t, numNodes);
}

TreeNode * copyCountHelper (TreeNode * t, int& numNodes)
// post: returns (via reference parameter) number of nodes rooted at t
// returns (as value of function) counted copy of t
// e.g., info field replaced by count of nodes rooted at t
{
    if (t == 0) {
        numNodes = 0;
        return 0;
    }
    else {
        // fill in general case here, declare at least two variables
        // to obtain count of left/right subtrees when making
        // recursive calls. Be sure to set the value of numNodes and
        // to return a value
    }
}
```
Part C (6 points)
Suppose `TreeNode` objects are used to implement a structure such as that shown below in which nodes may be referenced by more than one pointer and there may be cycles in the structure. Such a structure is called a `nest`. In this problem nests only contain positive integers.

![Diagram of a nest structure](image)

Complete the function `nestSum` whose header is given below. `nestSum` returns the sum of all reachable nodes in its parameter `nest` — nodes should only be “summed” once. If passed a pointer to the node labeled 1 in the nest shown above, `nestSum` should return 36, if passed a pointer to node labeled 6 `nestSum` should return 29. Your code is permitted to overwrite the `info` fields of the nodes in `nest` as a way of “marking” nodes. Recall that in this problem nests store positive integers. In “marking” a node by overwriting the `info` field, you may use any integer value (positive, zero, or negative).

```c
int nestSum(TreeNode *nest)
// pre: nest represents a nest of integers
// post:returns the sum of the positive integers in nest, 
// a node is only summed once. Values in nest may be lost
// in the process of summing.
{
    if (nest == 0) return 0; // cover empty case
    if (nest->info < 0) return 0; // cover case seen before
    int val = nest->info;
    // Your code here
}
```

Part D (4 points)
The function `nestCount` correctly counts the number of nodes in a nest without altering the values stored in the nest. A set of nodes seen before is used to avoid counting the same node twice.

Write a recurrence relation for the function `nestCountAux` called by `nestCount` (for the average case). Explain how each line of `nestCountAux` is accounted for in the recurrence. **Do not solve the recurrence.**

```cpp
int nestCountAux(TreeNode * nest, tset<TreeNode *> & visited)
// pre: visited contains pointers to nodes already processed
// post: returns number of nodes in nest
{
    if (nest == 0) return 0;  // nothing to count
    if (visited.contains(nest)) return 0;  // seen before

    visited.insert(nest);
    return 1 + nestCountAux(nest->left, visited)
             + nestCountAux(nest->right, visited);
}

int nestCount(TreeNode * nest)
// post: returns # nodes in nest
{
    tset<TreeNode *> visited;
    return nestCountAux(nest, visited);
}
```
Part E (8 points) The function \texttt{copy} below is designed to create and return a copy of a nest without altering the nest. A map of pointers in the nest being copied to corresponding newly created nodes in the copied nest is maintained to avoid copying the same node twice. Complete \texttt{copyAux} so that it works as specified.

TreeNode * copyAux(TreeNode * nest, BSTMap<TreeNode *, TreeNode *> & seen)
// pre: seen is a map of pointers in nest to pointers that are copies
// of these nodes in nest already visited
// post: returns copy of nest
{
    if (nest == 0) return 0; // nothing to copy
    if (seen.contains(nest)) return seen.get(nest); // already copied

    // fill in code here
}

TreeNode * copy(TreeNode * nest)
{
    BSTMap<TreeNode *, TreeNode *> seen;
    return copyAux(nest, seen);
}