Monkey Vines
Just like the postorder example shown in the class, the key is to write a recursive function that takes in (or has access to) the input string as well as a start index and end index:

```java
int solve(String s, int sI, int eI) {
    ...
}
```

The goal of the function would be to solve the subproblem for the string s[SI..eI]. The base case cases correspond to leaf nodes. In the recursive case, we must scan through the substring to find two subtrees (we are guaranteed to have two). Any substring that represents a subtree must have an equal number of [s and ]s. So, just start a counter of open brackets and close brackets. The first time you hit a close that makes the number equal to zero is your left subtree. So let's say we call this index endLeft. Then we want to make the following two recursive calls to the left and right subtrees, respectively:

```java
solve(s, sI+1, endLeft)
solve(s, endLeft+1, eI-1)
```

Note that the first and last characters of the original recursive call must be the enclosing brackets for both of those subtrees.

Then, between these trees, the larger tree is the one that must be balanced out, so if one call returns 8 and the other returns 4, that means you have to make the other tree "weigh" 8 so that the whole thing weighs 16.

Ordering Paper
Each sheet of 11" x 17" paper you order stores 4 pages of a single exam. If an exam booklet needs n pages, then you must order \[\lceil \frac{n}{4} \rceil\] sheets of paper. (In code this is just \((n+3)/4\) using integer division.) Then, for each exam, calculate the number of sheets of paper and multiply that by the number of students taking that exam. Finally, sum this product over all exams in a test case to get the result for the test case.

Sorting Exams
This problem is **identical** to the Week 2 Practice Problem: Add All. The key observation is that our cost function of merging two stacks is the sum of those stacks, which is the same exact cost function in Add All. The observation in that problem was that all things being equal, we'd rather have our current merge operation be of minimum cost, which means merging the two minimum stacks. (The exchange argument formally proves this.) Then, we want to find the two new minimum stacks and repeat until there is only one stack. Thus, store all numbers in a priority queue, delete min twice, add these two numbers together and add that to the total cost, then put this sum back in the priority queue and repeat this \(n-1\) times total, where \(n\) was the original number of stacks of exams.
**Spreading News**

Consider solving this problem for the root of some tree of employees. At the root, you get to choose which order to tell each of your direct subordinates. Naturally, you want them in order of which subtree will take the longest to spread the message to the shortest. For example, say that there are four subtrees, and it would take 3, 9, 8 and 8 seconds respectively for the message to completely spread through each of those subtrees. We want to tell the subtree that takes 9 seconds first, followed by one of the subtrees that takes 8 seconds, followed by the next one that takes 8 seconds, followed by the one that takes 3 seconds. This is because waiting to tell the "slowest" subtree could potentially delay the total amount of time the message takes to spread throughout the whole tree. Given the times above and the ordering given, the message actually gets through the second subtree in $1 + 9 = 10$ seconds, since it took one second to tell that subordinate, $2 + 8 = 10$ seconds for the third subtree since that subordinate waited 2 seconds to get the message, $3 + 8 = 11$ seconds for the fourth subtree, since that subordinate waited 3 seconds to get the message, and $4 + 3 = 7$, since the last subordinate waited 4 seconds to get the message. Thus, for this case, it would take a minimum of 11 seconds for the message to spread.

If we reordered these calls as say, 9, 3, 8 and 8, then the corresponding finish times would $1+9 = 10$ seconds, $2 + 3 = 5$ seconds, $3 + 8 = 11$ seconds and $4 + 8 = 12$ seconds. Thus, this ordering of telling the subordinates would not lead to the minimum answer.

Thus, in the general case do the following:

1. Recursively solve the problem for each child, storing the answer from each recursive call in an array list.

2. Sort that list in reverse order. (So, for our example, it would be 9, 8, 8, and 3.)

3. Add $i$ to the $i^{th}$ of the list, starting with $i = 1$. (So, for our example it would be 10, 10, 11 and 7.)

4. Return the maximum of the values listed in step 3.

The base case is a single node, which takes time 0.