Dot Game Dominator
Once again a greedy strategy can be used here. You want to eat the largest dot that is strictly smaller than your current size, since at each step, this makes your size grow the most. No competing sequence which eats a smaller dot can beat this greedy strategy. To implement this, a TreeSet can be used, since the class TreeSet has a method lower. However, be careful though - TreeSets don't store duplicates and an error will be caused in situations where there are multiple dots of the same size. To avoid this error, create a TreeSet of objects, where each object stores both the size of the dot and its ID number (you assign this as you read in the dots to be unique). Alternatively, it's very difficult to make data where the number of dots you eat is very large. Or if it is, where it takes you a long time to find the largest dot smaller than you. Thus, a relatively naive implementation which maintains a sorted list of the dot sizes will pass the data that was generated for this problem.

Ground Game
This question is the banger in the set. Keep a variable that tracks how many levels underground the player is and keep a second variable that stores the maximum levels underground the player has gone. When processing the input, ignore characters moving left (<) and right (>), add one to the count when moving down (v), and subtract one to the count when moving up (^). After each change to the counter, see if the new value is greater than the previously seen maximum. If so, update the maximum.

Strange Lottery Simulator
A regular trie where you store the number of words stored in each subtrie can handle the query asking for the number of participants with a particular prefix. However, if the names can be reversed, that same trie is not helpful. However, what can be done is that two different tries can be maintained simultaneously. Notice that reversing a name twice just brings it back to its original form. Thus, Whenever a name is added, add it to a trie forward as is and ALSO add its reversed version to a trie reverse. (So, "sharon" would get added to the trie forward AND "norahs" would get added to the trie reverse.) Finally, whenever you receive a query, you just need to know if the current state of the names is regular or reverse. Just keep either a boolean variable or an integer that toggles between 0 and 1, and then answer the query in the appropriate trie based on the value of the variable storing the current state of the names (forward or reverse).
**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.