Password checker:

```
int count = 0;

        // Check for at least one digit
        if (!password.matches(".*\\d.*")) {
            count++;
        }

        // Check for at least one lowercase letter
        if (!password.matches(".*[a-z].*")) {
            count++;
        }

        // Check for at least one uppercase letter
        if (!password.matches(".*[A-Z].*")) {
            count++;
        }

        // Check for at least one special character
        if (!password.matches(".*[!@#$%^&*()\\-+].*")) {
            count++;
        }

        // Ensure the total length is at least 6
        int lengthNeeded = 6 - n;

        return Math.max(count, lengthNeeded);
    }

}
```

Pattern syntax checker:

```
public class Solution
{
        public static void main(String[] args){
                Scanner in = new Scanner(System.in);
    int testCases = Integer.parseInt(in.nextLine());

    while (testCases > 0) {
      String pattern = in.nextLine();
      try {
        Pattern.compile(pattern);
        System.out.println("Valid");
      } catch (PatternSyntaxException e) {
        System.out.println("Invalid");
      }
      testCases--;
    }

    in.close();
  }
}
```

Height of a binary tree:

```
int getHeight(struct node* root) {
    if (root == NULL) {
        return -1; // Height of an empty tree is -1
    } else {
        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);

        // Return the greater of the two heights plus one (for the current node)
        return (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;
    }
}
```

Java comparator:

```
class Checker implements Comparator<Player> {
    @Override
    public int compare(Player p1, Player p2) {
        // First compare by score in descending order
        if (p1.score != p2.score) {
            return Integer.compare(p2.score, p1.score);
        }
        // If scores are the same, compare by name in ascending order
        return p1.name.compareTo(p2.name);
    }
}
```

Insertion sort:

```
int value = arr.get(n - 1);  // The last element to be inserted
        int i = n - 2;  // The index of the element before the last element


        // Move elements of arr[0..i] that are greater than value to one position ahead of their current position
        while (i >= 0 && arr.get(i) > value) {
            arr.set(i + 1, arr.get(i));
            printArray(arr);
            i--;
        }


        // Insert the value at the correct position
        arr.set(i + 1, value);
        printArray(arr);
    }


    private static void printArray(List<Integer> arr) {
        System.out.println(arr.stream().map(String::valueOf).collect(joining(" ")));
    }
}
```