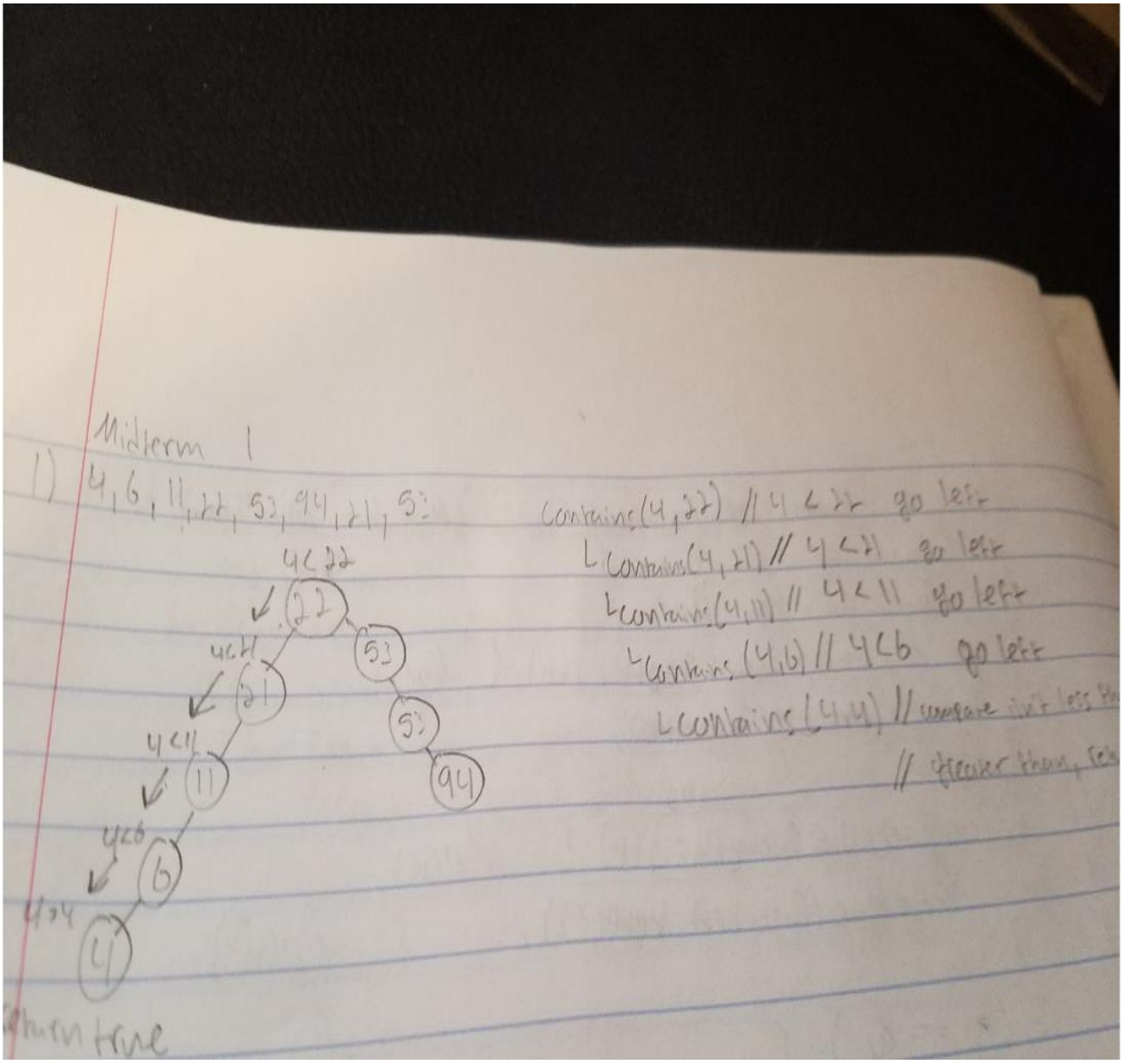
1)



The first check that the contains method does will compare the result of the node to the value that's being searched for, in this case, 4. The result returned will be < 0, so the next recursive call will be 21. After the next call, 4 is less than 21 so it will go down the tree and to the left again. The next value checked will be 11, and 4 is less than 11 so it will again return < 0 and go to the left. The next value will be 4 and 6, so again < 0 will be returned and the next value will be 4. 4 and 4 and it'll then return true after receiving a compareTO value of neither <0 and >0.

```
public static void doubleArray(int[] arr, myStack s) {

   for(int i = 0; i <= arr.length; i++) {
      for (int j = 0; j <= arr.length; j*= 2) {
         s.push(arr[j]);
       }
     }
  }
```

2)

This method takes an array and doubles the int values and then pushes them into a stack. The parameters used are an int array arr, and myStack s. The return value would be the result of the array being doubled and pushed into the stack. The O(nlog(n)) run time comes from the two for loops. The first for loop is O(n), and when multiplied with the second for loop, which is O(log(n)) due to the *= sign. An edge case for this algorithm could be someone passing ion characters, or special characters, as my code doesn't detect these or prevent them. A possible solution not this would be an if statement at the start to check the input from the array to check for a special character and then print out an invalid number and resume parsing through the integer.

```
public static Boolean isPalindrome(String[] s, Node fast, Node slow) {

      Queue q1 = new Queue;
      Queue q2 = new Queue;

      for (int i = 0; i <= s.length; i++) {
         q1.enqueue(slow.next(i));
       }
      for (int j = 0; j <= s.length; j++) {

         q2.enqueue(fast.next.next(i));
        }
      if (q1 == q2) {
         return true;
     } else {
         return false;
       }

}
```

3)

This algorithm uses the slow and fast runner concept learned in class to check if a string is a palindrome or not. It receives a fast and slow Node from a linked list and a string array to parse through. It will return either true or false. A possible edge case is in case an empty array is being passed in due to having no check for this case. A quick fix would be an if statement at the start to check if the head or the array is emmpty and returning null.

4) A way to create a run time exception would be to change the for loop checks to less than, rather than <= s.length. Additionally, I would chane the iteration process to ++I, which is pre-conditional, causing the loop to increment before the loop. This causes the array to go out of

bounds, causing an exception. Finally, you can remove the braces for one of the loops, causing it to run and run without breaks, or a check to stop.

```
public static void findCat(String[] para) {

   int hideSpot = 0;

   for (int i = 0; i <= para.length; i++) {
      if(para.substring(i, i+2).equals("(("); {
         for(int j = para.length; j >= 0; j--) {
            if(para.substring(j, j-2).equals("))"); {
               hideSpot++;
              }
            }
          }
        }
   }
```

5)

The number of spots the cat can hide in is 3

```
public static void findCat(String[] para) {

   int hideSpot;
   String s = " ";

   for (int i = 0; i <= para.length; i++) {
      if(para.substring(i, i+2).equals("((") {
         s += "((";
      if (s.equals("((") {
         if(para.substring(i,i+2).equals("))"){
            s += "))";
          }
        }


     }
      hideSpot++;
       s += "";
  }
```

6)

The change to the code to make this O(n) is using one for loop to iterate through once and using if statements to keep track of a temp string variable. The loop will keep track of s and append the parenthesis to it, and once ((is reached it will then look for)) to add to the count. At the end of these Ifs it will then reset s to a blank space and begin again.