1. Testing: Please briefly discuss how you went about testing your heap implementation.

I created a ThreeHeap, add several doubles to the heap to check if insert() is working, then checked the size, organization of the heap, and if the size of the array expanded etc. Then I removed the elements and checked the above once again to make sure deleteMin() is working. I also tested buildHeap() with an ArrayList. I wrote a function display() which print the values of the heap in order to make checking easier.

2. Big-O Analysis of your Heap: What is the worst case big-O running time of buildHeap, isEmpty, size, insert, findMin, and deletemin operations on your heap. [For this analysis you should ignore the cost of growing the array. That is, assume that you have enough space when you are inserting a value.]

	runtime
buildHeap():	O(n)
isEmpty():	O(1)
size():	O(1)
insert():	O(log3 n)
findMin():	O(1)
deleteMin():	O(log3 n)

3. Asymptotic runtime: For each of the following program fragments, determine the asymptotic runtime in terms of n.

a.
```
public void mysteryOne(int n) {
  int sum = 0;
  for (int i = n; i >= 0; i--) {
    if ((i % 5) == 0) {
      break;
    } else {
      for (int j = 1; j < n; j*=2) {
        sum++;
      }
    }
  }
}
```

The runtime for this function is: O(logn).
The for loop breaks when i%5 == 0, so the for loop runs for at most 5 times. Inside the for loop, there is another for loop, which loops for long times. So the total runtime would be 5log(n). Therefore O(logn).

b.
```
public void mysteryTwo(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < (n * (n + 1) / 3); j++) {
```

```
      x += j;
    }
  }
}
```

The runtime for this function is O(n^3).
The first for loop runs n times, the second for loop runs for n(n+1)/3 times. addition is O(1). So in total there are n*n(n+1)/3 O(1) operations. Therefore O(3).

c.
```
public void mysteryThree(int n) {
  for (int i = 0; i < n; i++) {
    printCats(n);
  }
}


public void printCats(int n) {
  for (int i = 0; i < n; i++) {
    System.out.println("catsmoop");
  }
}
```

The runtime for this function is O(n^2).
The loop inside mysteryThree runs n times, then it calls another function, which also runs n times when called. So the total run time would be n^2. Therefore O(n^2).

4. Psuedocode and Runtime:
a.Write pseudocode for a function that calculates the largest difference between any two numbers in an array of positive integers with a runtime in Θ(n2).
For example, the largest difference between any two numbers in the array storing the values [4, 6, 3, 9, 2, 1, 20] would be 19.

Pseudocode:
```
public int maxDiff(int array[]){
        int max = 0;
        for (int i = 0; i < array.length; i++){
                for (int j = 1; j < array.length; j++){
                        int temp = abs(array[i] - array[j]);
                        max = Math.max(max, temp);
                }
        }
        return max;
}
```
The runtime for this function is O(n^2).
The two for loops both run n times. So total runtime is O(n^2).

b. Can this function be written with a runtime Θ(n)? If yes, write the pseudocode. If no, why? What would have to be different about the input to do so?

Pseudocode:

```
public int maxDiff(int arr[]) {
        int max = arr[1] - arr[0];
        int min = arr[0];
        int i;
        for (i = 1; i < arr.length; i++) {
            if (arr[i] - min > max){
                max_diff = arr[i] - min_element;
                 }
            if (arr[i] < min){
                min = arr[i];
                 }
        }
        return max;
}
```

Runtime for this function is O(n).
There is only one for loop and inside the for loop are multiple O(1) operations.

Precondition is that there must be at least two elements in the array.

c. Can this function be written with a runtime Θ(1)? If yes, write the pseudocode. If no, why? What would have to be different about the input to do so?

No, we would have to loop through the array at least once to find the max difference.

5. Amortized Runtime Analysis: For this analysis you did in question 2, you ignored the cost of growing the array. You assumed you had enough space in the array when inserting the value. Now, for each of the following resize algorithms, re-evaluate the runtime of the insert operation, without assuming that the array has enough space. Explain why each answer is the same or different than your answer for question 2. You do not have to write pseudocode for these algorithms, just explain their asymptotic amortized runtime and why it is or isn't different than your answer for question 2.

a. You start with an array of size 5. When the array is full, you make an array that has 5 extra slots and copy over your elements to the new array.

	runtime
i) buildHeap():	O(n)

When inserting elements into the array (before percolating), the runtime for assign value in the list to the array is O(1), but every 5 insertions would cause one enlargement of the array,

which is O(n). So (4/5n*O(1) + 1/5n*O(n))/n = O(n) Percolation is O(log3 n), smaller than O(n). Therefore, the runtime for buildHeap is still O(n).

ii) isEmpty(): O(1)
This method does not cause any change to the heap. So the analysis stays the same.

iii) size(): O(1)
This method does not cause any change to the heap. So the analysis stays the same.

iv) insert(): O(n)
For every 5 elements, the array enlarges its size by 5 slots. So (4/5n*log3(n) + 1/5n*O(n))/n = O(n).

v)findMin(): O(1)
This method does not cause any change to the heap. So the analysis stays the same.

vi) deleteMin(): O(log3 n)
Array does not shrink so array size does not change. Analysis stays the same.

b. You start with an array of size 10. When the array is full, you make an array that is 1.5 times larger and copy over your elements.

runtime
i) buildHeap(): O(log3 n)
When inserting elements into the array (before percolating), the runtime for assign value in the list to the array is O(1), but when the array is full, it would copy everything to a array that's 1.5 times large, which is O(n). So ((n-1)*O(1) + 1*O(n))/n = O(1) Percolation is O(log3 n), larger than O(1). Therefore, the runtime for buildHeap is O(log3 n).

ii) isEmpty(): O(1)
This method does not cause any change to the heap. So the analysis stays the same.

iii) size(): O(1)
This method does not cause any change to the heap. So the analysis stays the same.

iv) insert(): O(log3 n)
For every n, elements are copied to a larger array. ((n-1)*log3(n) + 1*O(n))/n = O(log3 n). So the amortized runtime for insert is still O(log3 n).

v)findMin(): O(1)
This method does not cause any change to the heap. So the analysis stays the same.

vi) deleteMin(): O(log3 n)
Array does not shrink so array size does not change. Analysis stays the same.

c. The algorithm you used for insert. You start with an array of some size. When the array is full you double the array and copy over your elements.

	runtime
i) buildHeap():	O(log3 n)
ii) isEmpty():	O(1)
iii) size():	O(1)
iv) insert():	O(log3 n)
v)findMin():	O(1)
vi) deleteMin():	O(log3 n)

I think this would have the same runtime with part b, which copies every element to a new array with 1.5 size. Because both algorithms calls resize when the array is full. The new array that is 1.5 times larger will be smaller than the one that is 2 times larger, thus for a sufficiently large n, there would be more calls of resize(), which would result in more O(n) operation. However since array size is smaller, the number of elements copied over will be smaller. This should be able to compensate the difference between the number of calls of resize(). Overall, the runtime for both algorithms should be the same.

6. Above and Beyond: Did you implement anything for extra credit?
Yes. My buildQueue() function is using Floyd's method instead of n insert()s.