Additional Problems

VI.1 The following code computes the product of a **and** b**. What is its runtime?**

```
int product(int a, int b) {
    int sum = 0;
    for (int i = 0; i < b; i++) {
        sum += a;
    }
    return sum;
}
```

VI.2 The following code computes a^b**. What is its runtime?**

```
int power(int a, int b) {
    if (b < 0) {
```

```
        return 0; // error
    } else if (b == 0) {
        return 1;
    } else {
        return a * power(a, b - 1);
    }
}
```

VI.3 The following code computes a % b**. What is its runtime?**

```
int mod(int a, int b) {
    if (b <= 0) {
        return -1;
    }
    int div = a / b;
    return a - div * b;
}
```

VI.4 The following code performs integer division. What is its runtime (assume a and b are both positive)?

```
int div(int a, int b) {
    int count = 0;
    int sum = b;
    while (sum <= a) {
        sum += b;
        count++;
    }
    return count;
}
```

}

VI.5 **The following code computes the [integer] square root of a number. If the number is not a perfect square (there is no integer square root), then it returns -1. It does this by successive guessing. If n is 100, it first guesses 50. Too high? Try something lower – halfway between 1 and 50. What is its runtime?**

```
int sqrt(int n) {
    return sqrt_helper(n, 1, n);
}

int sqrt_helper(int n, int min, int max) {
    if (max < min) return -1; // no square root

    int guess = (min + max) / 2;
    if (guess * guess == n) { // found it!
        return guess;
    } else if (guess * guess < n) { // too low
        return sqrt_helper(n, guess + 1, max); // try higher
    } else { // too high
        return sqrt_helper(n, min, guess - 1); // try lower
    }
}
```

VI.6 **The following code computes the [integer] square root of a number. If the number is not a perfect square (there is no integer square root), then it returns -1. It does this by trying increasingly large numbers until it finds the right value (or is too high). What is its runtime?**

```
int sqrt(int n) {
    for (int guess = 1; guess * guess <= n; guess++) {
        if (guess * guess == n) {
            return guess;
```

```
        }
    }
    return -1;
}
```

VI.7 **If a binary search tree is not balanced, how long might it take (worst case) to find an element in it?**

VI.8 **You are looking for a specific value in a binary tree, but the tree is not a binary search tree. What is the time complexity of this?**

What is the time complexity of this?

VI.9 **The appendToNew method appends a value to an array by creating a new, longer array and returning this longer array. You've used the appendToNew method to create a copyArray function that repeatedly calls appendToNew. How long does copying an array take?**

```
int[] copyArray(int[] array) {
    int[] copy = new int[0];
    for (int value : array) {
        copy = appendToNew(copy, value);
    }
    return copy;
}

int[] appendToNew(int[] array, int value) {
    // copy all elements over to new array
    int[] bigger = new int[array.length + 1];
    for (int i = 0; i < array.length; i++) {
        bigger[i] = array[i];
    }

    // add new element
    bigger[bigger.length - 1] = value;
    return bigger;
}
```

VI.10 **The following code sums the digits in a number. What is its big O time?**

```
int sumDigits(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

VI.11 **The following code prints all strings of length k where the characters are in sorted order. It does this by generating all strings of length k and then checking if each is sorted. What is its runtime?**

```
int numChars = 26;

void printSortedStrings(int remaining) {
    printSortedStrings(remaining, "");
}

void printSortedStrings(int remaining, String prefix) {
    if (remaining == 0) {
        if (isInOrder(prefix)) {
            System.out.println(prefix);
        }
        } else {
            for (int i = 0; i < numChars; i++) {
                char c = ithLetter(i);
                printSortedStrings(remaining - 1, prefix + c);
            }
        }
    }

    boolean isInOrder(String s) {
        for (int i = 1; i < s.length(); i++) {
            int prev = ithLetter(s.charAt(i - 1));
            int curr = ithLetter(s.charAt(i));
            if (prev > curr) {
                return false;
            }
        }
        return true;
    }

    char ithLetter(int i) {
        return (char) (((int) 'a') + i);
    }
```

VI.12 The following code computes the intersection (the number of elements in common) of two arrays. It assumes that neither array has duplicates. It computes the intersection by sorting one array (array b) and then iterating through array a checking (via binary search) if each value is in b. What is its runtime?

```
    int intersection(int[] a, int[] b) {
        mergesort(b);
        int intersect = 0;

        for (int x : a) {
            if (binarySearch(b, x) >= 0) {
                intersect++;
            }
        }

        return intersect;
    }
```

Solutions

1. `O(b)`. The for loop just iterates through b.
2. `O(b)`. The recursive code iterates through b calls, since it subtracts one at each level.
3. `O(1)`. It does a constant amount of work.
4. `O(`$^a/_b$`)`. The variable `count` will eventually equal $^a/_b$. The while loop iterates count times. Therefore, it iterates $^a/_b$ times.
5. `O(log n)`. This algorithm is essentially doing a binary search to find the square root. Therefore, the runtime is `O(log n)`.
6. `O(sqrt(n))`. This is just a straightforward loop that stops when `guess*guess > n` (or, in other words, when `guess > sqrt(n)`).

7. `O(n)`, where n is the number of nodes in the tree. The max time to find an element is the depth tree. The tree could be a straight list downwards and have depth n.

8. `O(n)`. Without any ordering property on the nodes, we might have to search through all the nodes.

9. $O(n^2)$, where n is the number of elements in the array. The first call to `appendToNew` takes 1 copy. The second call takes 2 copies. The third call takes 3 copies. And so on. The total time will be the sum of 1 through n, which is $O(n^2)$.

10. `O(log n)`. The runtime will be the number of digits in the number. A number with d digits can have a value up to 10^d. If `n = ` 10^d, then `d = log n`. Therefore, the runtime is `O(log n)`.

11. $O(kc^k)$, where k is the length of the string and c is the number of characters in the alphabet. It takes $O(c^k)$ time to generate each string. Then, we need to check that each of these is sorted, which takes `O(k)` time.

12. `O(b log b + a log b)`. First, we have to sort array b, which takes `O(b log b)` time. Then, for each element in a, we do binary search in `O(log b)` time. The second part takes `O(a log b)` time.