Thanks to Prof. Betty O’Neil for parts of this solution.

1. For the following piece of code, calculate the runtime as a function of n:

```java
void foo(int n) {
    int i = 0, sum = 1;
    while (sum <= n) {
        i++;
        sum+=i;
    }
}
```

It looks tricky but definitely solvable with a little bit of thinking. Let us see who sum grows in each loop: At first, sum is 1, then 1+2, then 1+2+3 etc. We stop when \( sum \leq n \). However, sum grows as \( 1 + 2 + 3 + \ldots = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \). So, it grows in a quadratic rate to n, therefore we can intuitively assume it will surpass \( n \) at a rate that is more or less \( \sqrt{n} \). To give an exact number, let us calculate when \( \frac{sum \cdot (sum + 1)}{2} \geq n \rightarrow \frac{sum^2 + sum}{2} \geq n \rightarrow \frac{sum^2 + sum - 2n}{2} \geq 0 \). This is a quadratic equation in sum, so it has two solutions: \( -1 + \sqrt{1 + 8 \cdot n} \). We want the positive one (we only deal with positive numbers here). So it’s \( -1 + \sqrt{1 + 8 \cdot n} \), which is indeed the order of \( \sqrt{n} \).

2. Given the following piece of code:

```java
public static void mystery(int n)
{
    if(n==1) {
        System.out.println("Something");
    }
    else {
        System.out.println("Something");
        mystery(n/2);
    }
}
```

Let \( T(n) \) be the running time of mystery in terms of its argument \( n \).

(a) Every recursive call the size of \( n \) is halved and a constant operation is executed (the print). The boundary condition is just printing a line and hence a constant. So the formula is:
\[
T(n) = T\left(\frac{n}{2}\right) + c,
\]
for some constant \( c \).

(b) This is logarithmic, just like binary search (intuitively, doubling \( n \) adds one to the runtime).

(c) “Something” be printed three times, for \( n = 7, 3, 1 \) (remember integer division in Java).

3. Calculating power2
(a) `int power2(int n)
{
    if (n==0) return 1;
    return power2(n-1)+power2(n-1);
}

We have a double recursion here, so obviously the runtime is bad. The formula is $T(n) = C + 2T(n-1)$. So each run doubles the number of recursive calls while doing constant time operations outside the recursions. This is exponential, so $T(N) = O(2^N)$. It can be shown easily by expanding the next expression $T(n-1) = C + 2T(n-2)$, etc. The factors of 2 accumulate so that at the $k^{th}$ stage we have $2^k \cdot C$. Overall we have $n$ stages.

(b) A linear time algorithm would be the following:
```
int fib(int n)
{
    if (n==0) return 1;
    return 2*power2(n-1);
}
```

We simply run the recursion once and multiply by 2... You can easily show that the recurrence formula is linear, since it’s the same as the factorial function we showed in class: $T(n) = C + T(n-1)$.

(c) **Bonus:**
```
int power2New(int n)
{
    if (n==0) return 1;
    if (n % 2 == 0) {
        int result = power2New(n/2);
        return result*result;
    }
    else
        return 2*power2New(n-1);
}
```

Explanation: If $n$ is even, then $2^n = (2^{\frac{n}{2}})^2$, so we can calculate $2^{\frac{n}{2}}$ recursively and square, cutting half of $n$ in one move, which is similar to what we do in a binary search. So if we could always do that, the runtime formula would be the same as binary search: $T(n) = T(\frac{n}{2}) + C$, which is logarithmic. However, if $n$ is odd we resort to the previous method by subtracting one from $n$. We showed it is linear... So why is the overall runtime logarithmic despite the fact that we may have to run some “linear” stages? The answer lies in the following observation: If $n$ is even, we divide it by 2. The next stage we get $\frac{n}{2}$ which may be even or odd. If $n$ is odd, we subtract 1 and get $n-1$ which is always even! In other words – we have an odd $n$ at most one half of the times, so the runtime is at most twice as much as the “optimal” scenario when $n$ is always even, hence $2 \cdot (O(\log n)) = O(\log n)$.

4. Here is a possible solution:
```
class Passcode {
    private String pass;

    public Passcode(String s) {
        if (!valid(s)) return;
        pass = s;
    }

    public boolean authenticate(String s) {
        return s.equals(pass);
    }
}
public void changePass(String s) {
    System.out.println("Enter old password");
    Scanner oldP = new Scanner(System.in);
    String input = oldP.nextLine();
    if (!authenticate(input)) {
        System.out.println("Passwords do not match");
        return;
    }
    System.out.println("Enter new password");
    Scanner newP = new Scanner(System.in);
    input = newP.nextLine();
    if (!valid(input)) {
        System.out.println("Invalid password!");
        return;
    }
    pass = newP;
}

public boolean valid(String s) {
    return s.matches("[a-zA-Z0-9]{8,}");
}

public static void main(String[] args) {
    String s = args[0];
    System.out.println(s);
    if (s.matches("[a-zA-Z0-9]{8,}"))
        System.out.println("Valid!");
    else
        System.out.println("invalid!");
}

5. List<String> list1 with ("A","B","C","D")
   (a) 1. list1.iterator().next(): "A"
        2. list1.listIterator().next(): "A"
        3. list1.listIterator(2).next(): "C"
        4. list1.listIterator(4).previous(): "D"
   (b) The sequence of commands will result in the following: list1.listIterator(2).next(); list1.listIterator().remove();
        list1.listIterator(4).previous() At first “C” will be returned and then removed (the last call to next),
        so the list now has the members ”A”, ”B”, ”D”. The last command will throw an exception because
        the list has only 3 members.

6. Map’s get(), and put() methods can be implemented in O(1) time. Also, size(), containsKey() and
   remove() would be O(1), too. The Set’s add(), remove(), contains(), and remove() methods are O(1)
   time, if we use HashSet. Also size(), isEmpty(). A List is not a good candidate for using a hash
   table because the notion of order is not supported by the hash table ADT. One could argue that an
   immutable list could be hashed based upon position, and thereby preserves the order, but once we
   start inserting and deleting things in the hash, “maintaining order” becomes a pain, requiring deletes
   and re-_inserts of all the elements after the inserted or deleted one. The map operations that can be
   implemented with O(1) are get, put and remove. The set operations are add, remove and contains.
   We can’t implement a list since a list requires an order on the elements and a hash cannot provide it. 
   Additionally, lists allow duplicates and hash does not (in the key domain).

7. (a) A LinkedList<String> is best. Read each line, add to the list and then iterate backwards and
   print. This is O(N) in the number of lines.
(b) A `HashSet<String>` (a TreeSet is also OK but a bit slower and order is not important). It removes duplicates so each word appears just once. The runtime is $O(N)$ in the number of words for filling the hash table. Then $O(m)$ to print, where $m$ is the number of unique words (worst case $N$).

(c) A `TreeMap<String,Integer>` (This time order is important). Each time you read a word, check if it’s already in the Map. If so, increment the counter value. Otherwise, add it to the map with a value of 1. The runtime is $O(N \log N)$ where $N$ is the number of words for building the tree. Then $O(m \log m)$ for printing, where $m$ is the number of unique words. Worst case $O(N \log N)$. 