1. 

A. Suppose S is a Stack<Integer> that starts off empty and we do the following sequence of operations:

```java
S.push(1);
S.push(2);
S.pop();
S.push(3);
S.pop();
S.push(4);
S.push(5);
```

With the same stack we do the following loop:

```java
while (!S.isEmpty()) {
    System.out.print(S.pop());
}
```

**What will this loop print?**  

5 4 1

B. Next, we do this with a queue. Suppose Q is an empty queue and we do

```java
Q.enqueue(1);
Q.enqueue(2);
Q.dequeue();
Q.enqueue(3);
Q.enqueue(4);
Q.dequeue();
Q.dequeue();
Q.enqueue(5);
```

What will the following loop print?

```java
while (!Q.isEmpty()) {
    System.out.print(Q.dequeue());
}
```

4 5
2. Here is a loopy chunk of code:

```java
int total = 0;
for (int i = 0; i < N; i++) {
    total = total + i*i;
    for (int j = N-1; j >= 0; j--) {
        total = total - j;
        for (int k = 1; k < 10; k++) {
            total = total + k;
        }
    }
    for (int m = 0; m < N; m++) {
        total = total + 1;
    }
}
System.out.println(total);
```

Give a Big Oh upper bound for the runtime of this in terms of N.

The loop on k runs 10 steps. This is \(O(1)\).

The loop on j runs \(N\) times; each time it runs the loop on k. This is \(O(N)\)

The loop on m runs \(N\) steps so this is also \(O(N)\).

The loop on i runs \(N\) steps; in each step it runs first the loop on j (\(O(N)\) steps) and the loop on m (\(O(N)\) steps). So the body of the loop on i is \(O(N)\) and the body is run \(N\) times. Altogether this is \(O(N^2)\).
3. What good are interfaces? **Describe (in 3 sentences or less) any situation where it is useful to have an interface.**

Interfaces allow you to assert that a class has certain methods. In Lab3 we defined MyStack and MyQueue to implement the StackADT and QueueADT interfaces. If you have a list of objects of various classes and all that you want to do with these objects is to call a method Print( ) for each one, you can make an interface Printable which says that its implementing classes have a method Print, and then you can use Printable as the base type of the list.
In Lab 3 you implemented Queues with a linked structure. Suppose we use an ArrayList instead, as we did with Stacks. Here is a start:

```java
public class MyQueue<T> implements QueueADT<T> {
    ArrayList<T> data;
    int size;
    public MyQueue( ) {
        data = new ArrayList<T>();
        size = 0;
    }
    ...
}
```

A. Write `public void enqueue(T item)` for this implementation.

```java
public void enqueue(T item) {
    data.add(item);
    size += 1;
}
```

B. Write `public T dequeue() throws NoSuchElementException` for this implementation.

```java
public T dequeue() throws NoSuchElementException {
    if (size ==0)
        throw new NoSuchElementException();
    else {
        T temp = data.get(0);
        data.remove(0);
        size -= 1;
        return temp;
    }
}
```

Note that you could pack some of the statements in the else-clause together:

```java
    size -= 1;
    return data.remove(0);
```
5. For this question I will make a new data structure that I call a BobList<T>. A BobList is just like an ArrayList, with the addition of a method removeRandom(), which removes and returns a random element of the list:

```java
public class BobList<T> extends ArrayList<T> {
    T removeRandom () {
        Random rand = new Random();
        int i = random.nextInt(size());
        return remove(i);
    }
}
```

Now think about the Maze lab. I want to make a new MazeSolverBob class that uses a BobList<Square> as its worklist, with the BobList add() and removeRandom() for its add() and next() methods. In other words, we’ll store the worklist as a list, and each time we take an element from it we will get a random element of the worklist rather than the first or last element.

Here is the question: **will this still solve the maze?** Why or why not?

Yep, it will still solve the maze. Take any path from the start square to the exit and number the squares on it 0, 1, 2, ..., N where the start is numbered 0 and the exit is numbered N. We start by putting square 0 (Start) into the worklist. Sooner or later it will come out and its neighbors, including square 1, will go into the worklist. Eventually square 1 will come out and its neighbors, including square 2, will go into the worklist. This continues until square N, the exit square, comes out of the worklist.
Write method sums(L) whose signature is
ArrayList<Integer> sums(ArrayList<Integer> L)

This method takes a list L as an argument and it returns a list that I will call M. The first entry of M is the first entry of L. The second entry of M is the sum of the first two entries of L. The third entry of M is the sum of the first three entries of L, and so forth. For example, if L has entries 3 5 7 then the list returned by sums(L) will have entries 3 8 1

Here are 3 different solutions

Version 1: Get the entries of M by adding the next entry of L to the previous entry of M

```java
ArrayList<Integer> sums(ArrayList<Integer> L) {
    ArrayList<Integer> M = new ArrayList<Integer>();
    if (L.size() == 0)
        return M;
    else {
        M.add(L.get(0));
        for (int i = 1; i < size; i++) {
            M.add(M.get(i-1)+L.get(i));
        }
        return M;
    }
}
```

Version 2: Just loop through the entries of L adding them together, saving each sum in M

```java
ArrayList<Integer> sums(ArrayList<Integer> L) {
    ArrayList<Integer> M = new ArrayList<Integer>();
    int sum = 0;
    for (int i=0; i < L.size(); i++) {
        sum += L.get(i);
        M.add(sum);
    }
    return M;
}
```

Version 3: Compute the ith entry of M by adding entries 0 through i of L

```java
ArrayList<Integer> sums(ArrayList<Integer> L) {
    ArrayList<Integer> M = new ArrayList<Integer>();
    for (int i=0; i < L.size; i++) {
        int sum = 0;
        for (int j = 0; j <= i; j++)
            sum += L.get(j);
        M.add(sum);
    }
    return M;
}
```