(1)

Assume that lists are implemented using an array. For each of the following List methods, say whether (in the worst case) the number of operations is independent of the size of the list (is a constant-time method) or is proportional to the size of the list (is a linear-time method):

- the constructor
- add (to the end of the list)
- add (at a given position in the list)
- isEmpty
- contains
- Get

- ***constructor****: This method allocates the initial array, sets current to -1 and sets numItems to 0. This has nothing to do with the sequence size and is constant-time.*
- ***add*** *(to the end of the list): In the worst case, the array was full and you have to allocate a new, larger array, and copy all items. In this case the number of operations is proportional to the size of the list. If the array is not full, this is a constant-time operation (because all you have to do is copy one value into the array and increment numItems).*
- **add** (at a given position in the list): As for the other version of add, if the array is full, time proportional to the size of the list is required to copy the values from the old array to the new array. However, even if the array is not full, this version of add can require time proportional to the size of the list. This is because, when adding at position k, all of the items in positions k to the end must be moved over. In the worst case (when the new item is added at the beginning of the list), this requires moving all items over, and that takes time proportional to the number of items in the list.
- **isEmpty**: This method simply returns the result of comparing numItems with 0; this is a constant=time operation.
- **contains**: This method involves looking at each item in the list in turn to see if it is equal to the given item. In the worst case (the given item is at the end of the list or is not in the list at all), this takes time proportional to the size of the list.
- **get**: This method checks for a bad position and either throws an exception or returns the value in the given position in the array. In either case it is independent of the size of the list and so it is a constant-time operation.

(2)

Constant and linear times are not the only possibilities. For example, consider method createList:

```
ListADT<Object> createList(int n) {
   ListADT<Object> numbers = new SimpleArrayList();
   for (int k = 1; k <= n; k++)
       numbers.add(0, new Integer(k));
   return numbers;
}
```

Note that, for a given N, the for-loop above is equivalent to:

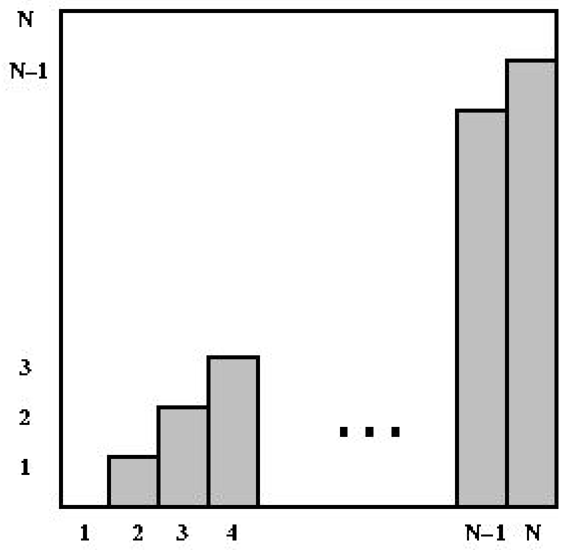
```
numbersL.add(0, new Integer(1) );
numbers.add(0, new Integer(2) );
numbers.add(0, new Integer(3) );
   ...
numbers.add(0, new Integer(n) );
```

If we assume that the initial array is large enough to hold N items, then the number of operations for each call to add is proportional to the number of items in the list when add is called (because it has to move every item already in the array one place to the right to make room for the new item at position 0). For the N calls shown above, the list lengths are: 0, 1, 2, ..., N-1. So what is the total time for all N calls? It is proportional to 0 + 1 + 2 + ... + N-1.

Recall that we don't care about the exact time, just how the time depends on the problem size. For method createList, the **problem size** is the value of N (because the number of operations will be different for different values of N). It is clear that the time for the N calls (and therefore the time for method createList) is ***not***independent of N (so createList is not a constant-time method). Is it proportional to N (linear in N)? That would mean that doubling N would double the number of operations performed by createList. Here's a table showing the value of 0+1+2+...+(N-1) for some different values of N:

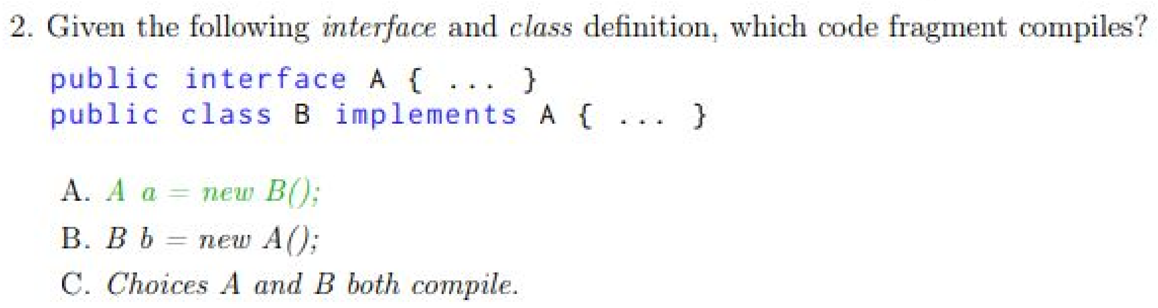
N	**0+1+2+...+(N-1)**
4	6
8	28
16	120

Clearly, the value of the sum does more than double when the value of N doubles, so createList is not linear in N. In the following graph, the bars represent the lengths of the list (0, 1, 2, ..., N-1) for each of the N calls.



The value of the sum (0+1+2+...+(N-1)) is the sum of the areas of the individual bars. You can see that the bars fill about half of the square. The whole square is an N-by-N square, so its area is N^2; therefore, the sum of the areas of the bars is about $N^2/2$. In other words, the time for method createList is proportional to the ***square*** of the problem size; if the problem size doubles, the number of operations will quadruple. We say that the worst-case time for createList is **quadratic** in the problem size.

(3)

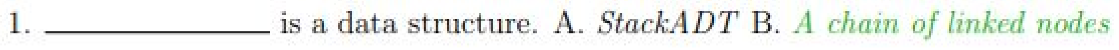


2. Given the following *interface* and *class* definition, which code fragment compiles?

```
public interface A { ... }
public class B implements A { ... }
```

A. *A a = new B();*
B. *B b = new A();*
C. *Choices A and B both compile.*

You could create the new subclass object but use super class to represent it however, you can not do it reversely.

(4) 

1. ________________ is a data structure. A. *StackADT* B. *A chain of linked nodes*

Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations. **In comparison, a data structure is a concrete implementation of the space provided by an ADT.**

(5)

[illegible]

3. Without modifying *BagADT interface*, it ____________ possible to have an indirect access iterator over the items in a *SimpleArrayBag*. Assume *SimpleArrayBag* implements *BagADT*. Recall that *BagADT* is:

```
interface BagADT {
          void add(Object item);
          Object remove();
          boolean isEmpty();
}
```

A. *is* B. *is not*

Lacking the get() method to implement next()

(6)

28. Given that you have implemented a **Queue** with a **circular array** and that the Strings **A,B,C,D** have been enqueued in that order (See figure).
Note: A circular array is one where index access wraps around the end of the array.

A	B	C	D	

What is the contents of the array after these operations:
dequeue(), enqueue("E"), dequeue(), enqueue("F"), dequeue(), enqueue("G"), enqueue("H")

A.

F	G	C	D	E

B.

D	E	F	G	H

C.

E	E	G	D	H

D.

F	G	H	D	E

a Queue with a circular array which means the end of the array will link the beginning of the array to store the extra items which goes out of the boundary

(7) Should the algorithm with the lowest time complexity always be used? Briefly explain why/why not?
No, many considerations go into choosing the most appropriate algorithm. For example, memory usage and development time are not captured by Big-O notation, but can be equally (or sometimes even more) important as running time. Furthermore, Big-O notation hides large constants.

In this example, Algorithm 3 has a very large constant term; if we know that our problem instances will be small (say N = 1), we might prefer Algorithm 1 or 2.

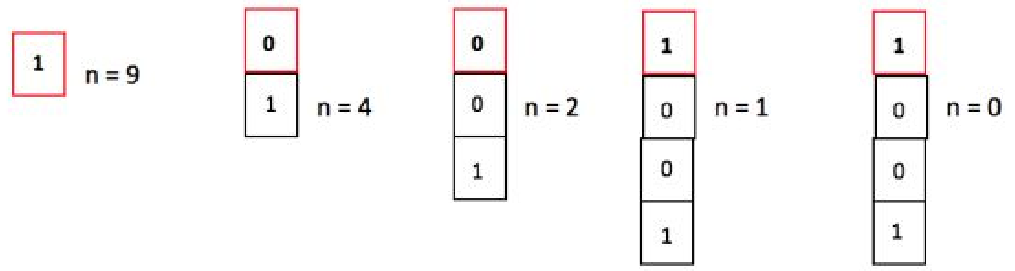
(8)

36. Read the below **secret** method thoroughly.

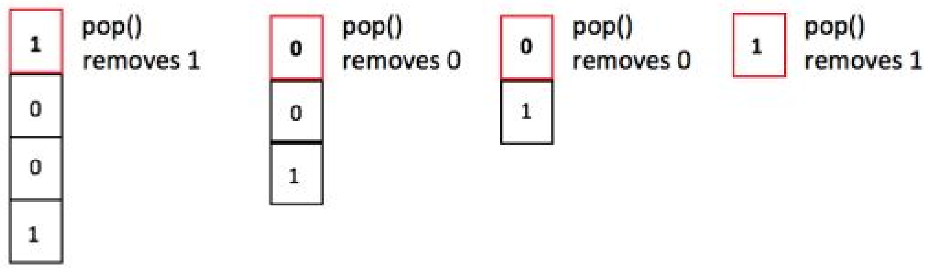
```
void secret(int n, Stack<Integer> st) {
        while (n > 0)
        {
                st.push(n % 2);
                n = n / 2;
        }

        while (!st.isEmpty())
                System.out.print(st.pop());
}
```

(a) **Trace** the code for this call: **secret(9, new Stack<Integer>())**.
In each iteration, show the value of **n** and the corresponding **st**; finally show the output of the function.

FIRST while loop (st indicated by the boxes):



SECOND while loop (st indicated by the boxes):

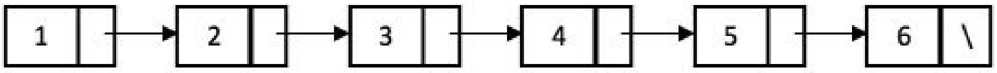


(b) What does the **secret** function do? **Hint**: Try generating the output for the call with a different value other than 9.
The **secret** function converts the given **decimal** number to its **binary** representation.
Exercise: Try tracing the method for few other values like 7, 8.

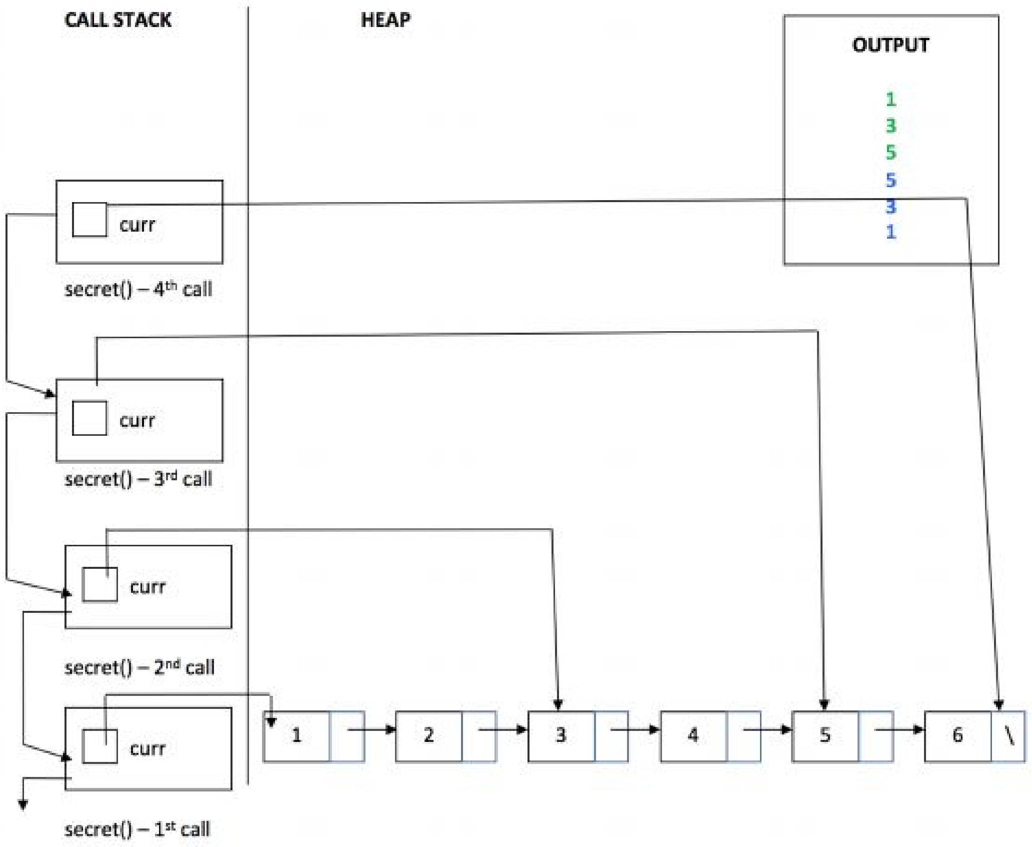
(9)

37. Read the below *recursive* **secret** method thoroughly. Show the **Call Stack**, **Heap** tracing and **Output** for the **secret** method given the following linked list.

```
void secret(Listnode<Integer> curr) {
        if(curr == null)
                return;
        System.out.println(curr.getData());

        if(curr.getNext() != null)
                secret(curr.getNext().getNext());
        System.out.println(curr.getData());
}
```



The output shown below in green color is the output from the first print statement. Observe the presence of the second print statement after the recursive call. The output of the second print statement is shown in blue color.


CALL STACK
HEAP
OUTPUT
1
3
5
5
3
1
curr
secret() – 4th call
curr
secret() – 3rd call
curr
secret() – 2nd call
curr
secret() – 1st call
1
2
3
4
5
6
\