CSCI 230
Test 1, Spring 2013

Name Key

1. (10 points) Write a function **T(n)** which measures **exactly** the complexity of the following algorithm, where the basic operations are the assignments to variable sum, and the addition done at the line marked by *. You may assume that n is a positive, even number. (Show your reasoning or work – if you can figure this out in your head, write a sentence supporting your answer. Again, I want an **exact** function – not an estimate.)

```
int sum = 0;
for (int i = n; i > 0; i=i-2)
   sum += i;  // *
```

1 assignment

2 ops - 1 asgn + 1 +

loop iterate $n/2$ times

T(n) = $2 \cdot \frac{n}{2} + 1 = \boxed{n+1}$

2. (4 points) Order the following complexity categories from smallest to largest (as a single group):

5 $\theta(n^3)$ 3 $\theta(n \log n)$ 4 $\theta(n^2)$ 2 $\theta(\log n)$

1 $\theta(100000)$ 8 $\theta(2^n)$ 7 $\theta(n^{15})$ 6 $\theta(n^3 \log n)$

$\frac{1}{2}$ pt per

3. (5 points) You have two algorithms to solve the same problem. Algorithm 1 has complexity **4n + 10**, while Algorithm 2 has complexity **$n^2 - 3$**. Is Algorithm 1 always a better choice? Why or why not?

No. For small values of n, Alg. 2 is a better choice.

$\boxed{n \leq 6}$

n	4n + 10	$n^2 - 3$
6	36	33
7	38	46

4. (10 points) What does the following algorithm compute? (**Show your trace.**)

```
mystery(n)
{
     if (n == 1)
         return 1;
     else
         return 2*n + mystery (n-1) – 1;
}
```

n = 4

mystery(4)
return 2 * 4 + mystery(3) - 1 = 8 + 9 - 1 = 16
mystery(3)
return 2 * 3 + mystery(2) - 1 = 6 + 4 - 1 = 9
mystery(2)
return 2 * 2 + mystery(1) - 1 = 4 + 1 - 1 = 4
mystery 1
return 1

myst(1) = 1
mystery(2) = 4
mystery(3) = 9
mystery(4) = 16
⋮

In plain English, the algorithm above computes n^2

5. (2 points) Which of the following recurrence relations, accurately describes the amount of work done in the ***mystery*** algorithm above, assuming that the basic operations counted are the *, + , and – operators. (Note – only one below is correct.) Explain why you chose the one you did.

Explain

(a) $T(n) = 4 + T(n-1)$, $T(1) = 0$ (b) $T(n) = T(n/2) + 4$, $T(1) = 0$ (c) $T(n) = T(2n + n-1) - 1$, $T(1) = 0$

Why? Bec. eff. for size n depends on (eff. of) cost of size n-1 + the 4 ops *, +, - + - in return stmt

6. (3 points) What is the complexity of the ***mystery*** algorithm above – here I want a growth rate. Support your answer.

O(n)

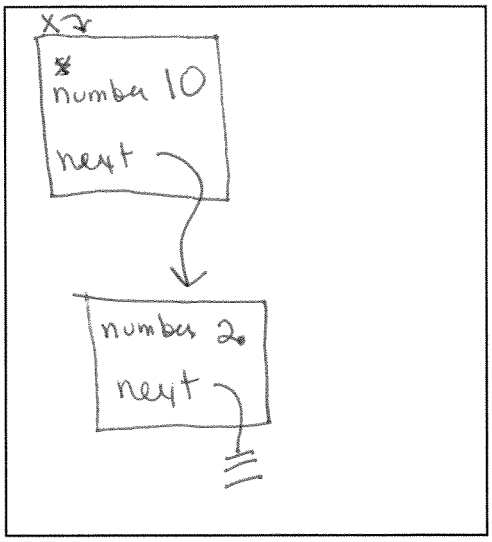
Counts down from n to 1

7. (10 points) Consider the following class Mystery:

```
public class Mystery {
    private int number;
    private Mystery next;

    public Mystery (int one, int two){
        if ( two < one ){
            number = one;
            next = new Mystery(two);
        }
        else{
            number = two;
            next = new Mystery(one);
        }}
    public Mystery (int one){
        number = one; next = null;}}
```

Varies slightly in tests



Suppose the following line of code "`Mystery x = new Mystery(10, 2)`" is executed.

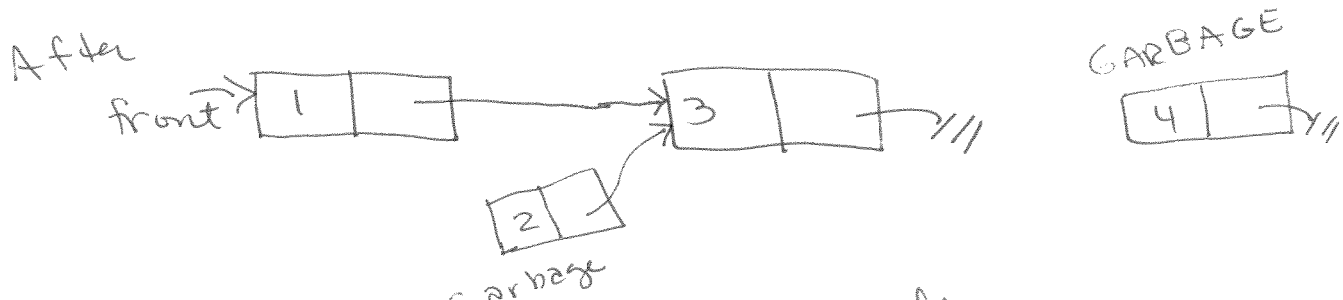
(a) How many objects are created of class "Mystery"? 2

(b) In the box **above right**, draw a diagram representing the object or objects created by that line of code.

8. (10 points) Assume an externally defined **ListNode** that provides public methods (**p.getNext( )** returns a reference to the node that p's next field stores, and **p.setNext( )** reassigns p's next field). Consider the following method.

```
public static void puzzle(ListNode p) {
  if (p != null){
     puzzle(p.getNext().getNext());
     p.setNext(p.getNext().getNext());
  }}
```

Assume **puzzle** is called with the front reference to a singly-linked list: **puzzle(front)** that contains **4** nodes. What is the result of the method? (Show work your doodles.)

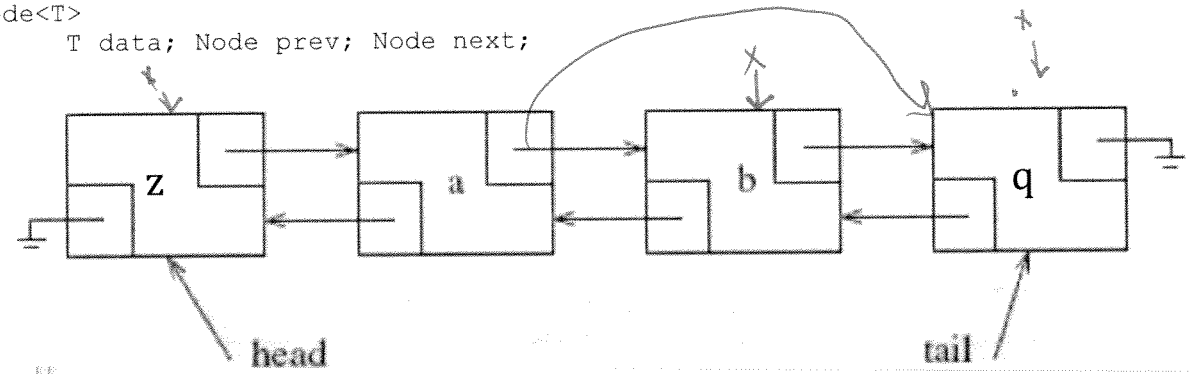


Result - list changed - 2nd + 4th nodes bypassed

9. (20 points) A doubly-linked list, as shown below, exists. Assume that a variable **x** references some node in such a list. Write code that will remove the node referenced by **x**. (Note, **x** could reference **ANY** node in the list, the list could be of any length but it does contain <u>at least one</u> node.) The code that you write is within a member method in the class that stores the list, so your code can directly access list nodes and parts of nodes. You may assume the node declaration shown next:

```
Node<T>
    T data; Node prev; Node next;
```

4 cases each 5 pts



#9

```
if (head == tail ){  // list has 1 element
    head = null; tail = null;
}
else if (head == x) {
    x.next.prev = null;
    head = x.next;
}
else if (tail == x) {
    x.prev.next = null;
    tail = x.prev;
}
else{   // x is an internal node
    x.prev.next = x.next;
    x.next.prev = x.prev;
}
```

10. (10 points) Do any **two** of the following (clearly label your answers) – assume existence of a **Stack<T>** class that provides the usual functionality, including an **isEmpty** method. An instance of a **Stack<Integer>** called **myStack** has been instantiated and is non-empty. You may declare additional Stack instances to solve the problems. (Bonus credit will be awarded for doing b and c, correctly.) Write a code segment that:

a. Sets a variable **bottom** equal to the bottom element in the stack, leaving **myStack** empty.
b. Sets a variable **bottom** equal to the bottom element in the stack, leaving **myStack** unchanged.
c. Make a copy of **myStack**, leaving **myStack** unchanged.

a)
```
while (!myStack.isEmpty())
    bottom = myStack.pop();
```

b)
```
Stack<Integer> otherStack = new Stack<Integer>;
while (!myStack.isEmpty())
    otherStack.push(myStack.pop());
bottom = otherStack.top();
while (!otherStack.isEmpty())
    myStack.push(otherStack.pop());
```

c) Same as b, but create another stack, called copy & replace 2nd loo
```
while(!other.isEmp
{ Integer i = other.po
  myStack.push(i
  copy.push(i
}
```

11. (10 points) Do any **two** of the following (clearly label your answers) – assume existence of a **Queue<T>** class that provides the usual functionality, including an **isEmpty** method. An instance of a **Queue<Integer>** called **myQueue** has been instantiated and is non-empty. You may declare additional Queue instances to solve the problems. (Bonus credit will be awarded for doing b and c, correctly.) Write a code segment that:

a. Sets a variable **last** equal to the rear/back element in the queue, leaving **myQueue** empty.
b. Sets a variable **last** equal to the rear/back element in the queue, leaving **myQueue** unchanged.
c. Make a copy of **myQueue**, leaving **myQueue** unchanged.

a)
```
while (!myQueue.isEmpty())
    bottom = myQueue.dequeue();
```

b)
```
Queue<Integer> otherQ = new Queue<Integer>;
while (!myQueue.isEmpty())
    otherQ.enqueue(myQueue.dequeue());
last = otherQ.dequeue();
myQueue.enqueue(last);
while (!otherQ.isEmpty())
    myQueue.enqueue(otherQ.dequeue())
```

c – need a 3rd queue – enqueue at some time, to rebuild myQueue

check for BONUS on ... and