Question 1a.
```
/** Instances of subclasses of Nuc
    represent nucleotides */
public abstract class Nuc {

    private char symbol; // a symbol for a nucleotide

    /** Constructor: a Nuc with symbol sym.
        Precondition: sym is the character for a
        nucleotide */
    public Nuc(char sym) {
        symbol= sym;
    }

    /** = the symbol representing this nucleotide */
    public char getSymbol() {
        return symbol;
    }

    /** = "ob is a nucleotide (i.e. a Nuc) and
        is complementary to this one." */
    public abstract boolean isComplement
                                    (Object ob);
}
```

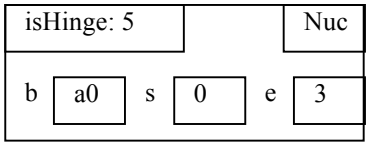
Question 1b. Make a class abstract so that instances of it cannot be created.

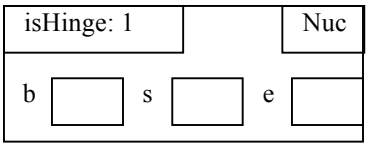
Question 1c.
```
/** an instance is a cytosine nucleotide
    (symbol 'C') */
public class CNuc extends Nuc {

    /** Constructor: a new cytosine molecule */
    public CNuc() {
        super('C');
    }
    /** = "ob is a Nuc whose symbol is 'G' (guanine)"
        (remember C, G are a complementary pair) */
    public boolean isComplement(Object ob) {
        if (!(ob instanceof Nuc)) {
            return false;
        }
        return ((Nuc)ob).getSymbol() == 'G';
    }
}
```

Question 2a.
```
/** = "b[s..e] is a perfect hinge." */
public static boolean isHinge(Nuc[] b, int s, int e) {
    if (e + 1 - s == 0)
        return true;
    if ((e + 1 - s) % 2 == 1)
        return false;
    return b[s].isComplement(b[e]) &&
        isHinge(b, s+1, e-1);
}
```

Question 2b.





Question 3.
```
/** = number of dips in b.
    Precondition: b contains at least one element. */
public static int numberOfDips(char[] b) {
    int n;

    // Set n to the number of dips in b.
    n= 0;
    // invariant: n = number of dips in b[0..k-1]
    for (int k= 1; k < b.length; k= k+1) {
        if (b[k-1] > b[k]) {
            n= n+1;
        }
    }

    // post: n = number of dips in b[0..b.length-1]
    return n;
    }
```

Question 4a.
```
/** = an array of Nucs corresponding
    to the symbols in s.
    Precond: The only characters that appear in s are
    'C', 'G', 'A', and 'U'. */
public static Nuc[] NucArray(String s) {
    Nuc[] b= new Nuc[s.length()];

    /* inv: Objects for s[0..i-1] have been placed
        in b[0..i-1] */
    for (int i= 0; i != s.length(); i= i+1) {
        char sym= s.charAt(i);
        if (sym == 'C') b[i]= new CNuc();
        else if (sym == 'G') b[i]= new GNuc();
        else if (sym == 'A') b[i]= new ANuc();
        else b[i]= new UNuc();
    }

    //post: Objects for chars in s have been placed in b
    return b;
}
```

Question 4b. nu may be cast to Object, Nuc, CNuc, and CNucS —and nothing else.

Since nu's apparent type is CNuc, upward (and identical) casts to CNuc, Nuc, and Object will be done automatically. A cast to CNucS must be done explicitly using (CNucS) nu.