1. [10 points] Indicate the outcome of executing each of the following (one of “compilation error”, “runtime error”, or whatever is printed). In each case, briefly justify your answer:

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
</table>
| a. ```
String s = "Hello ";
String s2 = s;
s = s + "there";
if( s == s2 )
    System.out.println( "equal" );
else
    System.out.println( "not equal" );
``` | "not equal" – the addition creates a new string, so s and s2 point to different things. |
| b. ```
class Test {
    public void greet () {
        System.out.println( "Hello!" );
    }
}

class Test {
    public void greet () {
        System.out.println( "Hello!" );
    }
}
``` | compilation error – cannot call an instance method from a static method |
| c. ```
class Thing {
    int x;
    public Thing( int x ) {
        this.x = x;
    }
    public boolean equals( Object p ) {
        return x == ((Thing) p).x;
    }
}

class Thing {
    int x;
    public Thing( int x ) {
        this.x = x;
    }
    public boolean equals( Object p ) {
        return x == ((Thing) p).x;
    }
}
``` | "equal" – Thing's equals method is called and returns true |
d. class Thing {
    int x;
    public Thing( int x ) {
        this.x = x;
    }
    public boolean equals( Object p ) {
        return x == ((Thing) p).x;
    }
}

public static void main(String[] args) {
    Thing p1 = new Thing(3),
    p2 = new Thing(3);
    if( p1.equals( p2 ))
        System.out.println("equal");
    else
        System.out.println("not equal");
}

}  

"equal" – the equals method returns true

---

e. class Thing {
    int x;
    public Thing( int x ) {
        this.x = x;
    }
    public boolean equals( Object p ) {
        return x == ((Thing) p).x;
    }
}

public static void main(String[] args) {
    Thing p1 = new Thing(3);
    if( p1.equals( "(3)" ))
        System.out.println("equal");
    else
        System.out.println("not equal");
}

}  

runtime error – cannot cast ",(3)" to Thing
in equals method.
2. [10 points] For each of the following assertions, indicate whether it is true or false. Give a clear and brief argument for your answer.

a. \( |\frac{x \sin x}{2}| \in \Theta(x) \)
   
   False \( |\frac{x \sin x}{2}| \) is bounded above by \( y = \frac{x}{2} \), but it goes to zero at \( x = k\pi \) (\( k \) is an integer). We cannot bound it below with any \( kx \), \( k \neq 0 \) so it is not \( \Theta(x) \).

b. \( 1000 - \frac{1}{n} \in O(1) \)
   
   True \( 1000 - \frac{1}{n} < 1000 \) for all positive \( n \). Therefore, this function has a constant upper bound.

c. \( 2^{-n^3} \in \Omega(n) \)
   
   False For very large values of \( n \), \( 2^{-n^3} \) approaches 0. No function \( y = kn \) (\( k \neq 0 \)) can bound it from below.

d. The worst-case time for executing F(A, 0) is \( \Omega(2^n) \), where \( n \) is A.length
   
   void F(int[] A, int k) {
     if (A.length <= k)
       return;
     for (int i = k+1; i < A.length; i += 1) {
       if (i == A[k]) {
         System.out.println(A[i]);
         F(A, k+1);
       }
     }
   }

   (Examine the loop carefully on this one!)

   False The loop will only ever execute F once, as \( i \) changes each time but \( A[k] \) stays the same.
   Worst case: \( n/2 \) calls to \( F \), loops are \( n/2 \) long on average \( \Rightarrow O(n^2) \)

   The upper bound on the worst case is \( O(n^2) \) so the lower bound cannot be \( \Omega(2^n) \).

e. In the best case, the number of calls to \( f \) in the following loop is \( O(n^2) \):
   
   for (int i = 0; i < n, i += 1)
     for (int j = i; i < n; j += 1)
       if (A[j] > Q)
         f(A[j]);

   True We can never have more than \( n^2 \) calls, so we can use \( n^2 \) as an upper bound for the best case.
3. [10 points] The class `Powerset` represents a **power set**: the set of all subsets of some given set of items (initially empty). The **add** method adds an item to the given set. The **iterator** method returns an `Iterator<Object[]>` - that is, an Iterator whose next method yields an array containing the items in one of the subsets (a unique one each time, in some unspecified order).

For example, in

```java
Powerset S = new Powerset();
S.add("Bob"); S.add("Alice");
for( Iterator<Object[]> i = S.iterator(); S.hasNext();)
    Do something with i.next();
```

`i.next()` takes on the `Object[]` values '{ }', '{ "Bob" }', '{ "Alice" }', '{ "Alice", "Bob" }' (not necessarily in that order).

a. Fill in the skeleton below and on the next page to get this effect.

```
public class Powerset {
    /** The powerset of the empty set */
    public PowerSet() { // FILL THIS IN
        elements = new List<Object>();
    }

    /** Add X to the current given set.  Throws an exception
    * if the size of the given set exceeds 32. */
    public void add( Object x ) { // FILL THIS IN
        if( elements.size() >= 32 )
            throw next Exception( "Powerset full" );
        elements.add( x );
    }

    /** An iterator over all subsets of the given set.  The
    * remove method throws UnsupportedOperationException. */
    public Iterator<Object[]> iterator () { // FILL THIS IN
        return new SetIterator(elements);
    }
}
```

More space on the next page.
//ADD ANY PRIVATE INSTANCE METHODS, VARIABLES, AND NESTED CLASSES
//YOU NEED HERE

private List<Object> elements;

private static class SetIterator extends Iterator {
    public SetIterator( List<Object> elements ) {
        lst = elements;
        current = 0;
    }
    List<Object> lst;
    unsigned int current;
    public void remove() throws UnsupportedOperationException {
        throw new UnsupportedOperationException();
    }
    public boolean hasNext() {
        return current >= ( 1 << lst.size() );
    }
    public Object[] next() {
        ArrayList<Object> ret = new ArrayList<Object>();
        int temp = current;
        int count = 0;
        while( temp != 0 ) {
            if( temp & 1 == 1 )
                ret.add( lst.get(current) );
            count +=1;
            temp = temp >>> 1;
        }
        current += 1;
        return ret.toArray();
    }
}
}
b. The *subset sum* problem considers the question of whether, given an integer \( k \) and a set \( S \) of integers, there exists a subset of \( S \) whose members add up to \( k \). Using class `Powerset` from part a, fill in the skeleton below to do the obvious, brute-force computation (yes, there is a better method, but let's not worry about it):

```
interface SubsetAccum {
    void add( Object[] x );
    Object result();
}

class SubsetSum {
    void accumulate(Iterator<Object[]> iterator, SubsetAccum accum ) {
        while( iterator.hasNext() )
            accum.add( iterator.next() );
    }

    /** True iff some subset of the integers in INTS sums to K,
    * where the integers in INTS are all distinct. */
    boolean subsetSum( int[] ints, int k ) {
        Powerset P = new Powerset();
        for( int x : ints )
            P.add(x);
        IntAccum a = new IntAccum( k );
        accumulate( P.iterator(), a );
        return a.result() == k ;
    }
}
```

//PRIVATE METHODS, NESTED CLASSES GO HERE

```
private static class IntAccum implements SubsetAccum {
    public IntAccum( int toMatch ) {
        match = toMatch;
        matched = false;
    }

    private int match;
    private boolean matched;
    public void add( Object[] x ) {
        int sum = 0;
        for( int i = 0; i < x.length; ++i ) {
            sum += (int) x[i];
        }
        if( sum == match )
            matched = true;
    }

    public Object result() {
        if( matched )
            return match;
        else
            return match +1;
    }
}
```
c. Assuming that next and hasNext on Powerset iterators take constant time, what is the worst-case time for subsetSum (Be as precise as possible).

\[ n2^{n-1} + n \quad \text{where } n \text{ is ints.size()} \]

4. [1 point] Where or what is Sark?
   Sark is one of the Channel Islands.

5. [10 points] Using the following class definition:
   
   ```java
   class IntList {
     // 'final' means head can't be changed after the constructor
     // sets it.
     public final int head;
     public IntList tail;
     public IntList( int head, IntList tail ) {
       this.head = head; this.tail = tail;
     }
   }
   ```
   
   fill in the methods on below and on the next page to agree with their comments. Define any additional methods you'd like.

   /* a. */
   /** Move the first occurrence of the maximum item in L to the end of
    * the list, maintaining the order of all other items and returning
    * the modified list. The operation is destructive and creates no new
    * IntList elements. E.g., if L is { 4, 3, 9, 2, 9, 8 }, returns
    * { 4, 3, 2, 9, 8, 9 } */
   static IntList moveMaxToEnd( IntList L ) { // FILL THIS IN
     IntList max = L.findMax();
     if( max.next == null )
       return L;
     if( L == max ) {
       IntList h = max.tail;
       L.findTail().next = max;
       max.next = null;
       return h;
     } else {
       IntList p = L;
       while( p.next != max ) {
         p = p.next;
       }
       p.next = max.next;
       L.findTail.next = max;
       max.next = null;
       return L;
     }
   }
   
   }
/* b. */
/** A list that is identical to L, except that the first
 * occurrence of the maximum element is remove to the end.
 * Nondestructive: has no effect on the original list L. */
static IntList maxToEnd( IntList l ) { // FILL THIS IN
    return moveMaxToEnd( l.copy() );
}

//ADDITIONAL METHODS MAY GO HERE

private IntList findMax() {
    if( tail == null )
        return this;
    IntList rest = findMax( next );
    if( rest != null && rest.head > head )
        return rest;
    else
        return this;
}

private IntList findTail() {
    IntList p = this;
    while( p.next != null ) {
        p = p.next;
    }
    return p;
}

private IntList copy() {
    if( tail == null ) {
        return new IntList( head, null );
    } else {
        return new IntList( head, tail.copy() );
    }
}