1. The following class defines integer lists.

```
class List {
  private int   f;
  private List  r;

  public list(int element, List others) {
         f = element;
         r = others; }

  public int first() { //getter method
         return f; }

  public List rest() { //getter method
         return r; }
     }
```

Write a class method *reverse* that takes a list of integers as input, and returns a new list with the integers in reverse order. For example, if the input list is *(4 7 3 2)* (i.e., a list containing four elements, in which the first element is *4*, the second element is *7* and so on), it should return the list *(2 3 7 4)*. This problem can be done in roughly 8 lines of Java code, so excessively long solutions will not be graded.

```
 public static List Reverse (List l) {
   List out = null;
   while (l != null) do {
     out = new List(l.first(), out);
     l   = l.rest();
   }
   return out;
 }
```

2. A *balanced string of parentheses* is a string that contains only the characters '(' and ')', and satisfies the following condition:

   (a) Either the string is empty, or

   (b) the first character is '(', the last character is ')' and the substring obtained by deleting the first and last characters must itself be a balanced string of parentheses.

Write a recursive Java program to check if an input string is a balanced string of parentheses.

```
public static boolean Balance (String s) {

 if (s == null) return true;
 else if (s == "") return true;
 else {
```

```
        char first = s.charAt(0);
        char last  = s.charAt(s.length() - 1);
        boolean mycheck = (first == '(') && (last  == ')' );
        if (s.length() <= 2) return mycheck;
        else return mycheck && Balance(s.substring(1,s.length() - 1));
}
}
```

3. (a) Java Nagila, a CS 211 student, implemented a bunch of algorithms and came up with the following running times as a function of input size. Using big-O notation, write down the simplest and most accurate expressions for the complexity of these algorithms.

   i. *f(n) = 2nlog(n) + 4n + 17log(n)*
   ii. *g(n) = log(n^3)*

   (b) You have an algorithm that runs in time *O(log(n))*. Java Nagila claims to have found a better algorithm that runs in time *O(log(n-1))*. Is either algorithm better than the other in an asymptotic sense? Explain your answer briefly using the formal definition of big-O notation.

   (c) The following method returns the i^{th} element of an integer list.

```
public static int Element(List L, int i)
     { if (i == 1) return L.first();
       else        return Element(L.rest(), i-1); }
```

   What is the asymptotic time complexity of accessing the m^{th} element of a list using this method? Express the complexity as a function of m, and n, the length of the list.

```
(ai)   f(n) = O(nlog(n))
(aii)  g(n) = O(log(n))
(b) No.
      We say that a(n) = O(b(n)) if there is exist k,n0 such that
      a(n) < k*b(n) for all n > n0.

      Let f(n) = log(n) and g(n) = log(n-1).

      Since g(n) < f(n) for all n > 2, we pick k = 1, and n0 = 2,
      and use the formal definition to claim that g(n) = O(f(n)).

      Now, we will show that f(n) = O(g(n)).
      We must find k, n0 such that log(n) < k* log(n-1) for all n > n0
      Pick k = 2, and n0 = 2.
      For all n > 2, n < square(n-1), so log(n) < 2* log(n-1) as desired.
      Therefore, f(n) = O(g(n)).

      Since f(n) = O(g(n)) and g(n) = O(f(n)), neither algorithm is
      better than the other asymptotically.
```

```
(c) T(1) = k
    T(m) = k + T(m-1)
      => T(m) = k*m
      => T(m) = O(m).
```

4. The great Iraqi superspy Java Ghanooj is on a spying mission and wishes to make a tour of all the cities in the United States. Recall that a tour in a directed graph is a path that begins and ends at the same node in that graph, and that passes through all other nodes exactly once. What is the largest number of tours that can exist in a graph with n nodes? (Hint: consider a graph in which there is an edge from every node to every other node. Such a graph is said to be a *complete* graph.) You must explain the reasoning behind your answer. You may find it useful to draw a few small complete graphs.

   In a complete graph, every node is adjacent to every other node. If we start at any arbitrary city, we have (n-1) choices for which city to visit next, (n-2) choices for the city after that, and so on, which gives us a total of (n-1)! tours starting and ending at a particular city. So there can be as many as (n-1)! distinct tours in a graph with n nodes.