Machine Representation Problem Set

This problem set is meant to help make sure that you understand how objects are represented in memory well enough to maintain an accurate conceptual model of how Java objects behave. As with the parameter problem set, these exercises are designed for pencil-and-paper solution. You are free to use the computer to check your work, but in principle you should be able to solve them without electronic assistance.

1. Assembly- to machine-language translation

Translate the following assembly-language program into its numeric machine-language equivalent, showing the contents of all memory addresses loaded by the program:

Assembly-language version

```
start:   INPUT   n
         LOAD    #0
         STORE   total
         LOAD    #1
loop:    STORE   i
         LOAD    n
         SUB     i
         JNEG    done
         LOAD    total
         ADD     i
         ADD     i
         SUB     #1
         STORE   total
         LOAD    i
         ADD     #1
         JUMP    loop
done:    OUTPUT  total
         HALT

i:       0
n:       0
total:   0
```

Machine-language version

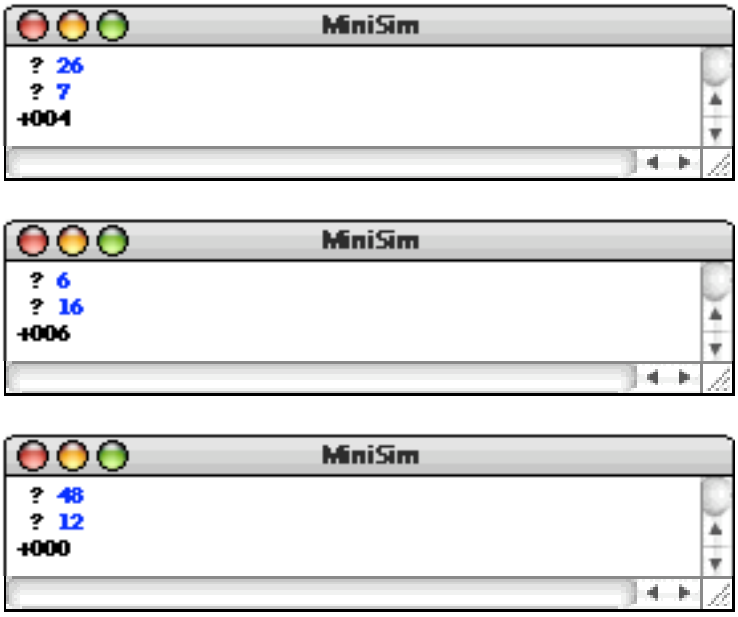
What output does this program produce if you enter 5 as the input value?

What value does this program compute in general?

2. MiniSim coding

Write a MiniSim program **`remainder.asm`** that requests two numbers from the user (which you may assume are both positive) and then computes the remainder of the first divided by the second. The problem, of course, is that MiniSim has only **`ADD`** and **`SUB`** instructions, and doesn't support multiplication and division. On the other hand, you can easily simulate the process of division by repeatedly subtracting the second number from the first until the result is negative. The remainder is the value immediately before the last subtraction.

Here are three independent sample runs to illustrate the operation of the program:



Answer to problem 2

3. Heap-stack diagrams

For each of programs on the two pages that follow, draw a diagram showing what values are stored on the heap and the stack at the specified point in the execution

3a)

Suppose that the class **`IndexList`** has been defined as follows:

```
public class IndexList {

   public IndexList(int n) {
      list = new int[n];
      for (int i = 0; i < n; i++) {
         list[i] = i;
      }
   }

   private int[] list;

}
```

and that the method **`testIndexList`** looks like this:

```
public void testIndexList() {
   IndexList list1 = new IndexList(3);
   IndexList list2 = list1;
}
```

←*Diagram at this point*

Using the heap-stack diagrams in Chapter 7 as a model, draw a diagram showing how memory is allocated just before **`testIndexList`** returns. You need not include explicit addresses in your diagram, but must indicate—either through addresses or arrows—where reference values point in memory. Your diagram should also include the names of any variables or fields.

heap | *stack*

3b)

Suppose that the class **`Domino`** has been defined as follows:

```
public class Domino {

   public Domino(int p1, int p2) {
      leftPips = p1;
      rightPips = p2;
   }

   private int leftPips, rightPips;
}
```

and that the method **`testDominos`** looks like this:

```
public void testDominos() {
   Dominos[] dominos = new Dominos[2];
   dominos[0] = new Domino(1, 3);
   dominos[1] = new Domino(2, 6);
}
```

←*Diagram at this point*

Using the heap-stack diagrams in Chapter 7 as a model, draw a diagram showing how memory is allocated just before **`testDominos`** returns. You need not include explicit addresses in your diagram, but must indicate—either through addresses or arrows—where reference values point in memory. Your diagram should also include the names of any variables or fields.

heap | *stack*