**Assigned Reading:** Sec. 5.2 on pp. 155 – 160 and the discussion of call-by-name on p. 165 in Sethi. [Note: You should already understand pass/call by value and pass/call by reference, as these parameter passing modes are used in C++. So most of the material on pp. 155 – 158 should be familiar to you.]


Complete the table below to show the output that is produced when the following program is executed. When completing each row of the table, assume that parameters are passed by the indicated mode.

```java
class Example {
    static int e;
    static int a[] = new int[3];

    static void test (int x) {
        a[1] = 6;
        e = 2;
        x += 3;
    }

    public static void main(String[] args) {
        a[1] = 1; a[2] = 2; e = 1;
        test(a[e]);
        System.out.println(a[1]+" +a[2]="" +e);
    }
}
```

Output for each parameter passing mode:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value:</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>reference:</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result:</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result (Algol W):</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

**Problem 1:**

Output for each parameter passing mode:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value:</td>
<td>6</td>
<td>-1</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>reference:</td>
<td>9</td>
<td>-1</td>
<td>-1</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result:</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result (Algol W):</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Example 2** (based on an old final exam question):

Complete the table below to show the output that is produced when the following program is executed. When completing each row of the table, assume that parameters are passed by the indicated mode.

```java
class FinalExam {
    static int e = 1;
    static int a[] = {0,1,2};

    public static void main(String[] args) {
        test(a[e], a[e-1]);
        System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + e);
    }

    static void test (int x, int y) {
        a[1] = 6;
        e = 2;
        x += 3;
        y--;
        System.out.print(x + " " + y + " ");
    }
}
```

Output for each parameter passing mode:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value:</td>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>reference:</td>
<td>9</td>
<td>-1</td>
<td>-1</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result:</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result (Algol W):</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Problem 2:**

Output for each parameter passing mode:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value:</td>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>reference:</td>
<td>9</td>
<td>-1</td>
<td>-1</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result:</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result (Algol W):</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
The dump was on pp. 5 – 7 below was produced when TJasn.TJ compiled the TinyJ program on p. 2 and then executed the generated code with a debugging stop after execution of exactly 23,178 instructions with the following sequence of input values: 4, 5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0. The code that was generated is shown on pp. 3 – 5. Note that the INITSTKFRM instructions in this code are at code memory addresses 10, 328, 397, and 496.

**Some Examples of Possible Questions Relating to the Dump**

1. (a) For each method, say how many locations are allocated to local variables in its stackframe.

   ANSWER: main: 8; readRow: 2
   transpose: 5; writeOut: 3

   (b) Write down the size of a stackframe of readRow(), transpose(), and writeOut().

   ANSWER: readRow: 7
   transpose: 10; writeOut: 8

**At the time of the debugging stop:**

2. Consider the static variables mat, count, and tm. What values are stored in the following locations?

   (a) count
   ANSWER: 1

   (b) mat[2][4]
   ANSWER: 5

   (c) tm[0][3][2]
   ANSWER: 8

3. Which method is being executed?

   ANSWER: writeOut

4. Which data memory locations constitute the stackframe of the executing method?

   ANSWER: addresses 201 through 208

5. What values are stored in the stackframe locations of the local variables and formal parameters of the executing method?

   ANSWER: i = 742, j = 0, mm = null,
   rows = -2, cols = 0,
   matr[ ][ ] = PTR to 10085

6. Which method called the executing method?

   ANSWER: transpose

7. Which method called the caller?

   ANSWER: readRow

8. Which method called the caller's caller? And which method called that method?

   ANSWER: transpose; main

9. What are the addresses of main's local variables h and j?

   ANSWER: h's addr is 172; j's is 173

**Next, suppose the debugging stop had not occurred.**

10. What would be the code memory addresses of the next 10 instructions to be executed?

   ANSWER: 509 – 517, then 506

11. What output, if any, would be produced by execution of these instructions? ANSWER: None

12. Which data memory locations, if any, would be changed in value by execution of the 10 instructions? Name the variable(s) whose values are stored there.

   ANSWER: address 206; i

13. Write down what the PC, FP, ASP and ESP registers would contain after execution of the first 3 of the above 10 instructions. Also write down the value or values of EXPRSTACK[j] for 0 ≤ j < ESP.

   ANSWER: PC=512, FP=PTR TO 205,
   ASP=PTR TO 209, ESP=1
   EXPRSTACK[0]=PTR TO 206

14. When the currently executing method activation RETURNs to its caller, what will PC, FP, and ASP be set to?

   ANSWER: PC=574, FP=PTR TO 197
   ASP=PTR TO 201

15. Now suppose code memory address 509 contained the instruction GT instead of the instruction LT. Answer questions 10 – 13 again under this assumption.

   ANSWER: 509, 510, then 518 – 525
   Output: 1 then newline
   No data memory location's value is changed.

   After exec. of 509-10, 518,
   PC=519, FP=PTR TO 205,
   ASP=PTR TO 209, ESP=1
import java.io.*;

class DumpEx {
    static int mat[][][], count;
    static int tm[][][] = new int[5][ ][];

    public static void main (String args[]) throws IOException
    {
        int r[], c[] = new int[5], n=1;
        int layer = -1;
        while (n == 1) {
            if (layer < 4)
                layer = layer + 1;
            else
                layer = 0;

            BufferedReader input = new BufferedReader
                (new InputStreamReader(System.in));

            System.out.print("Enter number of rows: ");
            r[layer] = Integer.parseInt(input.readLine());
            System.out.print("Enter number of columns: ");
            c[layer] = Integer.parseInt(input.readLine());

            mat = new int[r[layer]][ ];
            tm[layer] = mat;

            int i = 0;
            while (i < r[layer]) {
                mat[i] = new int[c[layer]];
                readRow(i + 1, mat[i], c[layer]);
                i = i + 1;
            }

            int h = 0;
            while (h <= layer) {
                System.out.println("Given matrix: ");
                writeOut(r[h], c[h], tm[h]);
                System.out.println("Transposed matrix: ");
                writeOut(c[h], r[h],
                    transpose(tm[h], r[h], c[h]));
                h = h + 1;
            }

            System.out.println("Doubled matrices: ");

            h = 0;
            while (h <= layer) {
                i = 0;
                while (i < r[h]) {
                    j = 0;
                    while (j < c[h]) {
                        tm[h][i][j] = tm[h][i][j] * 2;
                        System.out.print(tm[h][i][j]);
                        j = j + 1;
                    }
                    System.out.println();
                    i = i + 1;
                }
                h = h + 1;
                System.out.println("\n");
            }
        }
        System.out.print("Type 1 to continue, 0 to quit: ");
        n = Integer.parseInt(input.readLine());
    }
}

static void readRow(int rowNum, int m[], int c) throws IOException
{
    int i = 0;
    while (i < c) {
        if (rowNum == -1) {
            int mm[][] = new int[1][];
            writeOut(-1, 10, mm);
            i = i + 1;
        } else {
            BufferedReader input = new BufferedReader
                (new InputStreamReader(System.in));
            System.out.print("Enter value in column ");
            System.out.print(i+1);
            System.out.print(" ");
            m[i] = Integer.parseInt(input.readLine());
            i = i + 1;
        }
    }
}

static int[][] transpose(int m[][], int r, int c) throws IOException
{
    int k, i, ml[][] = new int[c][];
    k = 0;
    while (k < c) {
        ml[k] = new int[1];
        k = k + 1;
    }
    i = 0;
    while (i < r) {
        j = 0;
        while (j < c) {
            ml[j][i] = m[i][j];
            int mm[] = new int[1];
            readRow(-1, mm, 10);
            j = j + 1;
        }
        i = i + 1;
    }
    return ml;
}

static void writeOut (int rows, int cols,
    int matr[][][]) throws IOException
{
    int i = 0;
    if (rows == -2) {
        while (i < 1000) i = i + 1;
        System.out.println(count);
        count = count+1;
    }
    while (i < rows | rows == -1 & i < cols) {
        j = 0;
        while (j < cols) {
            if (rows == -1) {
                int mm[][] = new int[1][];
                writeOut(-2,0,mm);
                j = j + 1;
            } else {
                System.out.print(matr[i][j]);
                System.out.print(" ");
                j = j + 1;
            }
        }
        if (rows >= 0) System.out.println();
        i = i + 1;
    }
}
0:  PUSHSTADDR 0 78:  SAVETOADDR 156:  LOADFROMADDR
1:  PUSHNUM 0 79:  PUSHLOCADDR 6 157:  PASSPARAM
2:  SAVETOADDR 80:  PUSHNUM 0 158:  PUSHSTADDR 2
3:  PUSHSTADDR 1 81:  PUSHLOCADDR 159:  LOADFROMADDR
4:  PUSHNUMBER 0 82:  PUSHLOCADDR 160:  PUSHLOCADDR 7
5:  SAVETOADDR 83:  LOADFROMADDR 161:  LOADFROMADDR
6:  PUSHSTADDR 2 84:  PUSHLOCADDR 162:  ADDTOPTR
7:  PUSHNUM 5 85:  LOADFROMADDR 163:  LOADFROMADDR
8:  HEAPALLOC 86:  PUSHLOCADDR 4 164:  PASSPARAM
9:  SAVETOADDR 87:  LOADFROMADDR 165:  CALLSTATMETHOD 496
10: INITSTKFRM 8 88:  ADDTOPTR 166:  NOP
11: PUSHLOCADDR 1 89:  LOADFROMADDR 167:  WRITESTRING 64 82
12: PUSHNUM 5 90:  LT 168:  WRITELNOP
13: HEAPALLOC 91:  JUMPONFALSE 133 169:  PUSHLOCADDR 2
14: SAVETOADDR 92:  PUSHSTADDR 0 170:  LOADFROMADDR
15: PUSHLOCADDR 2 93:  LOADFROMADDR 171:  PUSHLOCADDR 7
16: PUSHNUM 5 94:  LOADFROMADDR 172:  LOADFROMADDR
17: HEAPALLOC 95:  LOADFROMADDR 173:  ADDTOPTR
18: SAVETOADDR 96:  ADDTOPTR 174:  LOADFROMADDR
19: PUSHLOCADDR 3 97:  PUSHLOCADDR 2 175:  PASSPARAM
20: PUSHNUM 1 98:  LOADFROMADDR 176:  PUSHLOCADDR 1
21: PUSHLOCADDR 99:  LOADFROMADDR 177:  LOADFROMADDR
22: PUSHLOCADDR 4 100:  LOADFROMADDR 178:  PUSHLOCADDR 7
23: PUSHNUM 1 101:  ADDTOPTR 179:  LOADFROMADDR
24: CHANGESIGN 102:  LOADFROMADDR 180:  ADDTOPTR
25: PUSHLOCADDR 103:  PUSHLOCADDR 181:  LOADFROMADDR
26: PUSHLOCADDR 104:  SAVEADDR 182:  PASSPARAM
27: LOADFROMADDR 105:  PUSHLOCADDR 6 183:  PUSHSTADDR 2
28: PUSHNUM 1 106:  LOADFROMADDR 184:  LOADFROMADDR
29: EQ 107:  PUSHNUM 1 185:  PUSHLOCADDR 7
30: JUMPONFALSE 327 108:  ADD 186:  LOADFROMADDR
31: PUSHLOCADDR 4 109:  PASSPARAM 187:  ADDTOPTR
32: LOADFROMADDR 110:  PUSHSTADDR 0 188:  LOADFROMADDR
33: PUSHNUM 4 111:  LOADFROMADDR 189:  PASSPARAM
34: LT 112:  PUSHLOCADDR 6 190:  PUSHLOCADDR 1
35: JUMPONFALSE 43 113:  LOADFROMADDR 191:  LOADFROMADDR
36: PUSHLOCADDR 4 114:  ADDTOPTR 192:  PUSHLOCADDR 7
37: PUSHLOCADDR 4 115:  LOADFROMADDR 193:  LOADFROMADDR
38: LOADFROMADDR 116:  PASSPARAM 194:  ADDTOPTR
39: PUSHNUM 1 117:  PUSHLOCADDR 2 195:  LOADFROMADDR
40: ADD 118:  LOADFROMADDR 196:  PASSPARAM
41: SAVETOADDR 119:  PUSHLOCADDR 4 197:  PUSHLOCADDR 2
42: JUMP 46 120:  ADDTOPTR 198:  LOADFROMADDR
43: PUSHLOCADDR 4 121:  ADDTOPTR 199:  PUSHLOCADDR 7
44: PUSHNUM 0 122:  LOADFROMADDR 200:  LOADFROMADDR
45: SAVETOADDR 123:  PASSPARAM 201:  ADDTOPTR
46: WRITESTRING 3 24 124:  CALLSTATMETHOD 328 202:  LOADFROMADDR
47: PUSHLOCADDR 1 125:  NOP 203:  PASSPARAM
48: LOADFROMADDR 126:  PUSHLOCADDR 6 204:  CALLSTATMETHOD 397
49: PUSHLOCADDR 127:  PUSHLOCADDR 6 205:  PASSPARAM
50: LOADFROMADDR 128:  LOADFROMADDR 206:  CALLSTATMETHOD 496
51: ADDTOPTR 129:  PUSHNUM 1 207:  NOP
52: READINT 130:  ADD 208:  PUSHLOCADDR 7
53: SAVETOADDR 131:  SAVETOADDR 209:  PUSHLOCADDR 7
54: WRITESTRING 25 49 132:  JUMP 82 210:  LOADFROMADDR
55: PUSHLOCAL 2 133:  PUSHLOCADDR 7 211:  PUSHNUM 1
56: PUSHLOCAL 134:  PUSHNUM 0 212:  ADD
57: PUSHLOCAL 135:  SAVETOADDR 213:  SAVETOADDR
58: LOADFROMADDR 136:  PUSHLOCADDR 7 214:  JUMP 136
59: ADDTOPTR 137:  LOADFROMADDR 215:  WRITESTRING 83 100
60: READINT 138:  PUSHLOCADDR 216:  WRITELNOP
61: SAVETOADDR 139:  LOADFROMADDR 217:  PUSHLOCADDR 7
62: PUSHLOCADDR 0 140:  LE 218:  PUSHNUM 0
63: PUSHLOCADDR 1 141:  JUMPONFALSE 215 219:  SAVETOADDR
64: LOADFROMADDR 142:  WRITESTRING 50 63 220:  PUSHLOCADDR 7
65: PUSHLOCADDR 143:  WRITELNOP 221:  LOADFROMADDR
66: LOADFROMADDR 144:  PUSHLOCADDR 222:  PUSHLOCADDR 4
67: ADDTOPTR 145:  LOADFROMADDR 223:  LOADFROMADDR
68: LOADFROMADDR 146:  PUSHLOCADDR 7 224:  LE
69: HEAPALLOC 147:  LOADFROMADDR 225:  JUMPONFALSE 322
70: SAVETOADDR 148:  ADDTOPTR 226:  PUSHLOCADDR 6
71: PUSHLOCAL 2 149:  LOADFROMADDR 227:  PUSHNUM 0
72: LOADFROMADDR 150:  PASSPARAM 228:  SAVETOADDR
73: PUSHLOCAL 151:  PUSHLOCADDR 2 229:  PUSHLOCADDR 6
74: LOADFROMADDR 152:  LOADFROMADDR 230:  LOADFROMADDR
75: ADDTOPTR 153:  PUSHLOCADDR 7 231:  PUSHLOCADDR 1
76: PUSHLOCAL 154:  LOADFROMADDR 232:  LOADFROMADDR
77: LOADFROMADDR 155:  ADDTOPTR 233:  PUSHLOCADDR 7
471:  | 472:  | 473:  | 474:  | 475:  |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE</td>
<td>PASSPARAM</td>
<td>PUSHLOCADDR</td>
<td>LOADFROMADDR</td>
<td>PASSPARAM</td>
</tr>
<tr>
<td>519:</td>
<td>520:</td>
<td>521:</td>
<td>522:</td>
<td>523:</td>
</tr>
<tr>
<td>LOADFROMADDR</td>
<td>WRITETIME</td>
<td>WRITETIME</td>
<td>PUSHSTATADDR</td>
<td>PUSHSTATADDR</td>
</tr>
<tr>
<td>567:</td>
<td>568:</td>
<td>569:</td>
<td>570:</td>
<td>571:</td>
</tr>
<tr>
<td>PASSPARAM</td>
<td>PUSHNUM</td>
<td>PUSHNUM</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
</tr>
<tr>
<td>572:</td>
<td>573:</td>
<td>574:</td>
<td>575:</td>
<td>576:</td>
</tr>
<tr>
<td>PASSPARAM</td>
<td>CALLSTATMETHOD</td>
<td>ADD</td>
<td>PUSHLOCADDR</td>
<td>PUSHLOCADDR</td>
</tr>
<tr>
<td>577:</td>
<td>578:</td>
<td>579:</td>
<td>580:</td>
<td>581:</td>
</tr>
<tr>
<td>PUSHNUM</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>582:</td>
<td>583:</td>
<td>584:</td>
<td>585:</td>
<td>586:</td>
</tr>
<tr>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>ADDTOPTR</td>
<td>LOADFROMADDR</td>
</tr>
<tr>
<td>587:</td>
<td>588:</td>
<td>589:</td>
<td>590:</td>
<td>591:</td>
</tr>
<tr>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>WRITEINT</td>
<td>WRITEINT</td>
</tr>
<tr>
<td>592:</td>
<td>593:</td>
<td>594:</td>
<td>595:</td>
<td>596:</td>
</tr>
<tr>
<td>RETURN</td>
<td>3</td>
<td>OR</td>
<td>JUMPONFALSE</td>
<td>PUSHLOCADDR</td>
</tr>
<tr>
<td>597:</td>
<td>598:</td>
<td>599:</td>
<td>600:</td>
<td>601:</td>
</tr>
<tr>
<td>PUSHLOCADDR</td>
<td>1</td>
<td>613</td>
<td>PUSHLOCADDR</td>
<td>PUSHLOCADDR</td>
</tr>
<tr>
<td>602:</td>
<td>603:</td>
<td>604:</td>
<td>605:</td>
<td>606:</td>
</tr>
<tr>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>PUSHLOCADDR</td>
</tr>
<tr>
<td>607:</td>
<td>608:</td>
<td>609:</td>
<td>610:</td>
<td>611:</td>
</tr>
<tr>
<td>RETURN</td>
<td>3</td>
<td>ADDTOPTR</td>
<td>JUMPONFALSE</td>
<td>PUSHLOCADDR</td>
</tr>
<tr>
<td>612:</td>
<td>613:</td>
<td>614:</td>
<td>615:</td>
<td>616:</td>
</tr>
<tr>
<td>JUMP</td>
<td>506</td>
<td>CALLSTATMETHOD</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
</tr>
<tr>
<td>617:</td>
<td>618:</td>
<td>619:</td>
<td>620:</td>
<td>621:</td>
</tr>
<tr>
<td>PUSHSTATADDR</td>
<td>1</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
<td>LOADFROMADDR</td>
</tr>
</tbody>
</table>

******* Debugging Stop ******

Data memory dump

Data memory--address 0 to top of stackframe stack, and allocated heap locations:

```
0: 2147428131 = PTR TO 10019  21: 119 = 'w'
1: 1 = Ctrl-A  22: 115 = 's'
2: 2147428113 = PTR TO 10001  23: 58 = ':'
3: 69 = 'E'  24: 32 = '
4: 110 = 'n'  25: 69 = 'E'
5: 116 = 't'  26: 110 = 'n'
6: 101 = 'e'  27: 116 = 't'
7: 114 = 'r'  28: 101 = 'e'
8: 32 = ' '  29: 114 = 'r'
9: 110 = 'n'  30: 32 = '
10: 117 = 'u'  31: 110 = 'n'
11: 109 = 'm'  32: 117 = 'u'
12: 98 = 'b'  33: 109 = 'm'
13: 101 = 'e'  34: 98 = 'b'
14: 114 = 'r'  35: 101 = 'e'
15: 32 = ' '  36: 114 = 'r'
16: 111 = 'o'  37: 32 = ' '
17: 102 = 'f'  38: 111 = 'o'
18: 32 = ' '  39: 102 = 'f'
19: 114 = 'z'  40: 32 = ' '
20: 111 = 'o'  41: 99 = 'c'
```
202: 0 = Ctrl-@ 
203: 2147428197 = PTR TO 10085 
204: 574 
205: 2147418309 = PTR TO 197 
206: 742 
207: 0 = Ctrl-@  
208: 0 = Ctrl-@  
10000: 2147428118 = PTR TO 10006 
10001: 2147428131 = PTR TO 10019 
10002: 0 = Ctrl-@  
10003: 0 = Ctrl-@  
10004: 0 = Ctrl-@  
10005: 0 = Ctrl-@  
10006: 2147428124 = PTR TO 10012 
10007: 4 = Ctrl-D  
10008: 0 = Ctrl-@  
10009: 0 = Ctrl-@  
10010: 0 = Ctrl-@  
10011: 0 = Ctrl-@  
10012: 2147428130 = PTR TO 10018 
10013: 5 = Ctrl-E  
10014: 0 = Ctrl-@  
10015: 0 = Ctrl-@  
10016: 0 = Ctrl-@  
10017: 0 = Ctrl-@ 
10018: 2147428135 = PTR TO 10023 
10019: 2147428136 = PTR TO 10024 
10020: 2147428142 = PTR TO 10030 
10021: 2147428148 = PTR TO 10036 
10022: 2147428154 = PTR TO 10042 
10023: 2147428141 = PTR TO 10029 
10024: 1 = Ctrl-A  
10025: 2 = Ctrl-B  
10026: 3 = Ctrl-C  
10027: 4 = Ctrl-D  
10028: 5 = Ctrl-E  
10029: 2147428147 = PTR TO 10035 
10030: 6 = Ctrl-F  
10031: 7 = Ctrl-G  
10032: 8 = Ctrl-H  
10033: 9 = Ctrl-I  
10034: 0 = Ctrl-@  
10035: 2147428153 = PTR TO 10041 
10036: 1 = Ctrl-A  
10037: 2 = Ctrl-B  
10038: 3 = Ctrl-C  
10039: 4 = Ctrl-D  
10040: 5 = Ctrl-E  
10041: 2147428159 = PTR TO 10047 
10042: 6 = Ctrl-F  
10043: 7 = Ctrl-G  
10044: 8 = Ctrl-H  

PC=509  ESP=2  FP= PTR TO 205  ASP= PTR TO 209  HMAX= PTR TO 15000 

Total number of instructions executed: 23178 

Last instruction to be executed: 508: PUSHNUM 1000 

Expression evaluation stack: 

1: 1000 
0: 742
Comments on the Answers

1(a) The answers are deduced from the operands of the methods' INITSTKFRM instructions at code memory addresses 10, 328, 397, 496. It is also possible to work out the answers from the local variable declarations in each method. In main(), for example, the local variables r, c, n, and layer are given the stackframe offsets 1, 2, 3, and 4; input is given the offset 5; i is given offset 6; h is given offset 7; and j is given offset 8. Note that the scopes of local variable declarations need to be taken into account. Thus if we add a declaration of a local variable hh inside the block of the while (h <= layer) { ... } loop that follows the declaration of h, then both hh and j will be given the offset 8 because the scopes of the declarations of hh and j will not overlap.

(b) Stackframe size = no. of parameters + 2 + no. of locations for local vars. The 2 extra locations are for the dynamic link (at offset 0) and the return address (at offset -1).

Note: This rule does not apply to main's stackframe--which is always the bottom stackframe on the stackframe stack. main has no caller and hence no return address. main's stackframe also does not need a dynamic link. But in this implementation the dynamic link location in main's stackframe (at offset 0) has a dummy value: It always points to the illegal data memory address 20000. (The highest legal data memory address is 19999; moreover, addresses 10000 - 19999 are reserved for use as heap memory.)

2. mat's address is 0, count's address is 1, and tm's address is 2. (b) and (c) are intended to test your understanding of arrays. (c) is solved as follows: tm's address is 2. That location points to tm[0], so tm[0][0]'s addr is 10001, and hence tm[0][0][0]'s addr is 10002. That location points to tm[0][3][0], so tm[0][3][0]'s addr is 10042, and hence tm[0][3][2]'s addr is 10044. That location contains the answer, 8.

3. From the addresses of the INITSTKFRM instructions, we see that main's code is at 10 - 327, readRow's code is at 328 - 396, transpose's code is at 397 - 495, writeOut's code is at 496 - 613.

The last instruction to be executed was at 508 (as stated on the 5th last line of the dump). This is within writeOut's code.

4. We see from FP that offset 0 of the stackframe is at 205. The beginning and end of the stackframe can be deduced from this and the answers to 1(a) and (b) for writeOut.

5. The answers are deduced from the stackframe offsets of the parameters and variables, and the fact that offset 0 is at 205. (In fact the variables j and mm are not in scope in the "while (i < 1000)" loop that is being executed at this time. So the values stored in the locations of j and mm are just "garbage" values!)

6. Return addr (at offset -1, addr 204) is 574. This is within writeOut's code.

7,8. The dynamic link in the stackframe of the currently executing method points to addr 197. That location points to 190. That location points to 180. That location points to 165. Thus 197, 190, 180, and 165 are the addresses of the offset 0 locations in the stackframes of the caller, the caller's caller, the caller's caller's caller, and the caller's caller's caller's caller. The return addresses stored in the first three of these stackframes (at addresses 196, 189, and 179) are 367, 479, and 205, which are instructions in the code of readRow, transpose, and main, respectively.

Note: Another way to tell that the caller's caller's caller's caller is main is to observe that offset 0 in its stackframe (addr 165) points to the illegal data memory address 20000--recall the note in the above comment on question 1(b).

9. Offset 0 in main's frame is at addr 165 (see comments on questions 7,8). h's stackframe offset is 7 and j's is 8.

10. PC contains 509, so 509: LT is the first of the 10 instructions. We see from p. 7 that at this time ESP = 2, EXPRSTACK[0] = 742, and EXPRSTACK[1] = 1000. Thus 1000 is on top of EXPRSTACK and 742 is the second item from the top. Since 742 < 1000, execution of LT replaces these two integers with the value 1 (which represents true), so the JUMPONFALSE at 510 does not jump after popping off this value.

11. Only WRITEINT, WRITESTRING, WRITELNOP, and FLUSH produce output.

12. Data memory is changed only by SAVETOADDR, PASSPARAM, CALLSTATMETHOD, INITSTKFRM, and HEAPALLOC. The only one of these that is executed here is SAVETOADDR (at 516). When this is executed, the pointer that is second from top on EXPRSTACK was put there by 511: PUSHLOCADDR 1. This refers to offset 1 in the currently executing method's stackframe, which is the location of i and has address 206 (from the answer to question 4).

[Note: HEAPALLOC changes data memory only because it sets the location that immediately precedes the block of heap memory it allocates to point to the location that immediately follows the block. This allows allocated blocks of heap memory to be deallocated by the garbage collector, and makes it possible to check at runtime that every array index is less than the length of the array.]

13,14. Questions like these are intended to test your understanding of what specific machine instructions do to the virtual machine. Here the instructions you are being tested on are LT, JUMPONFALSE, PUSHLOCADDR, and RETURN.
The dump below was produced when TJasn.TJ compiled the TinyJ program on p. 2 and executed the generated code with a debugging stop after execution of 1,209,794 instructions. The sequence of input values was 4, 3, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2. The INITSTKFRM instructions in the generated code are: 10: INITSTKFRM 7 345: INITSTKFRM 4 414: INITSTKFRM 6 515: INITSTKFRM 5

The instructions at addresses 357 – 413 in the generated code are shown on page 3.

Some Examples of Possible Questions Relating to the Dump

1. (a) For each method, say how many locations are allocated to local variables in its stackframes.

   ANSWER: main: 7; readRow: 4
            transpose: 6; writeOut: 5

   (b) Write down the size of a stackframe of readRow(), transpose(), and writeOut().

   ANSWER: readRow: 10
            transpose: 13; writeOut: 10

At the time of the debugging stop:

2. Consider the static variables mat, count, and tm. What values are stored in the following locations?

   (a) count    ANSWER: 100
   (b) mat[3][2] ANSWER: 2
   (c) tm[0][1][2] ANSWER: 6

3. Which method is being executed?

   ANSWER: readRow

4. Which data memory locations constitute the stackframe of the executing method?

   ANSWER: addresses 171 through 180

5. What values are stored in the stackframe locations of the formal parameters and first two local variables of the executing method?

   ANSWER: rowNum = –1
           m = PTR TO 10059
           c = 10  d = 6
           i = 0  mm = PTR TO 10061

6. Which method called the executing method?

   ANSWER: transpose

7. Which method called the caller?

   ANSWER: main

8. What are the addresses of main’s local variables layer and hhhhh?

   ANSWER: layer’s addr is 152;
            hhhhh’s is 155

Next, suppose the debugging stop had not occurred.

9. What would be the code memory addresses of the next 10 instructions to be executed?


10. What output, if any, would be produced by execution of these instructions?  ANSWER: None

11. Which data memory locations, if any, would be changed in value by execution of the 10 instructions? Name the variable(s) stored there and say what its/their value(s) is/are after execution of the 10 instructions.

   ANSWER: address 177, i, 1

12. Write down what the PC, FP, ASP and ESP registers would contain after execution of the first 3 of the above 10 instructions. Also write down the value or values of EXPRSTACK[j] for 0 ≤ j < ESP.

   ANSWER: PC=412, FP=PTR TO 176,
            ASP=PTR TO 181, ESP=0

13. When the currently executing method activation returns to its caller, what will PC, FP, and ASP be set to?

   ANSWER: PC=498, FP=PTR TO 164
            ASP=PTR TO 171

14. What is the next instruction to be executed after the execution of the 10 instructions listed in your answer to question 9?

   ANSWER: 365: PUSHLOCADDR –5

15. What will be on top of EXPRSTACK after execution of the instruction in your answer to question 14?

   ANSWER: PTR TO 171
import java.io.*;

class DumpEx {
    static int tm[][], [], count;
    static int mat[], [], count;
    static BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

    public static void main(String args[]) throws IOException {
        int r[], c[], n=1;
        int layer = -1;

        while (n == 1) {
            if (layer < 4)
                layer = layer + 1;
            else
                layer = 0;

            System.out.print("Enter number of rows: ");
            r[layer] = Integer.parseInt(input.readLine());
            System.out.print("Enter number of columns: ");
            c[layer] = Integer.parseInt(input.readLine());

            mat = new int[r[layer]][];
            tm[layer] = mat;

            int i = 0;
            while (i < r[layer]) {
                int iiiii = i;
                mat[i] = new int[c[layer]];
                readRow(i + 1, mat[i], c[layer], iiiii);
                i = i + 1;
            }

            int h = 0;
            while (h <= layer) {
                int hhhhh = h*2;
                System.out.println("Given matrix: ");
                writeOut(r[h], c[h], tm[h]);
                System.out.println("Transposed matrix: ");
                writeOut(c[h], r[h], transpose(tm[h], r[h], c[h], h, hhhhh));
                System.out.println(" ");
                h = h + 1;
            }
        }
        int jjjjj;
        System.out.print("\n\nType 1 to continue, 0 to quit: ");
        n = Integer.parseInt(input.readLine());
    }
}

static int[][] transpose(int m[], int r, int c, int p, int q) throws IOException {
    int temp, k, i, m1[],[];
    while (k < c) {
        m1[k] = new int[r];
        k = k + 1;
    }
    i = 0;
    while (i < r) {
        j = 0;
        while (j < c) {
            m1[j][i] = m[i][j];
            j = j + 1;
        }
        i = i + 1;
    }
    return m1;
}

static void writeOut (int rows, int cols, int matrix[[]]) throws IOException {
    int i = 0, tmp, tmp1;
    if (rows == -2) {
        while (i < 1000) i = i + 1;
        System.out.println(count);
        count = count+1;
    }
    while (i < rows | rows == -1 & i < cols) {
        int j = 0;
        while (j < cols) {
            if (rows == -1) {
                int mm[] = new int[1];
                writeOut(-2,0,mm);
            }
            else {
                System.out.print(matrix[i][j]);
                System.out.println(" ");
            }
            j = j + 1;
        }
        if (rows >= 0) System.out.println();
        j = 1;
        i = i + j;
    }
}

static void readRow (int rowNum, int m[[]],
    int c, int p, int d) throws IOException {
    if (rowNum >= 0) {
        System.out.print("Row ");
        System.out.println(rowNum);
    }
    int i = 0;
    while (i < c) {
        if (rowNum == -1) {
            int mm[] = new int[1];
            writeOut(-1, 10, mm);
            i = i + 1;
        }
        else {
            System.out.print("Enter value in column ");
            System.out.print(i+1);
            System.out.println(" ");
            m[i] = Integer.parseInt(input.readLine());
            i = i + 1;
        }
    }
}

static int[][] transpose(int m[[]], int r, int c, int p, int q) throws IOException {
    int temp, k, i, m1[[]],[];
    k = 0;
    while (k < c) {
        m1[k] = new int[r];
        k = k + 1;
    }
    i = 0;
    while (i < r) {
        j = 0;
        while (j < c) {
            m1[j][i] = m[i][j];
            i = i + 1;
        }
        i = i + 1;
    }
    return m1;
}

static void writeOut (int rows, int cols, int matrix[][]) throws IOException {
    int i = 0, tmp, tmp1;
    if (rows == -2) {
        while (i < 1000) i = i + 1;
        System.out.println(count);
        count = count+1;
    }
    while (i < rows | rows == -1 & i < cols) {
        int j = 0;
        while (j < cols) {
            if (rows == -1) {
                int mm[] = new int[1];
                writeOut(-2,0,mm);
                j = j + 1;
            }
            else {
                System.out.print(matrix[i][j]);
                System.out.println(" ");
            }
            j = j + 1;
        }
        if (rows >= 0) System.out.println();
        i = 1;
        i = i + 1;
    }
}
357:  PUSHNUM     0 17: 111 = 'o'
358:  SAVETOADDR 18: 102 = 'E'
359:  PUSHLLOCADDR 1 19: 110 = 'n'
360:  LOADFROMADDR 100: 110 = 'n'
361:  PUSHLLOCADDR 20: 114 = 't'
362:  SAVETOADDR 101: 116 = 't'
363:  LT          102: 105 = 'i'
364:  JUMPONFALSE 116: 105 = 'i'
365:  PUSHLLOCADDR -5 117: 58 = ':'
366:  LOADFROMADDR 118: 32 = ' ' 107: 32 = ' ' 10027: 2 = Ctrl-D
367:  PUSHLLOCADDR 119: 82 = 'R'
368:  PASSPARAM  120: 111 = 'o'
369:  LOADFROMADDR 121: 119 = 'w'
370:  PUSHLLOCADDR 122: 32 = ' ' 10022: 7 = Ctrl-G
371:  LOADFROMADDR 123: 69 = 'E'
372:  PUSHLLOCADDR 124: 110 = 'n'
373:  SAVETOADDR 125: 116 = 't'
374:  PUSHLLOCADDR 126: 117 = 'e'
375:  PUSHLLOCADDR 127: 114 = 't'
376:  SAVETOADDR 128: 32 = ' ' 10028: 4 = Ctrl-D
377:  ADDTOPTR  129: 118 = 'v'
378:  PUSHLLOCADDR 130: 97 = 'a'
379:  SAVETOADDR 131: 108 = 'E'
380:  JUMP  412: 51: 71 = 'G'
381:  SAVETOADDR -4 132: 117 = 'u'
382:  PUSHLLOCADDR 133: 101 = 'e'
383:  SAVETOADDR 134: 32 = ' ' 10034: 9 = Ctrl-I
384:  SAVETOADDR 135: 105 = 'i'
385:  SAVETOADDR 136: 110 = 'n'
386:  ADD   137: 32 = ' ' 10036: 6 = Ctrl-F
387:  WRITETINT 138: 99 = 'c'
388:  WRITETSTRING 139: 111 = 'o'
389:  LOADFROMADDR 140: 108 = 'i'
390:  ADDTOPTR 141: 117 = 'u'
391:  SAVETOADDR 142: 109 = 'm'
392:  SAVETOADDR 143: 110 = 'n'
393:  SAVETOADDR 144: 114 = 't'
394:  LOADFROMADDR 145: 117 = 'u'
395:  SAVETOADDR 146: 32 = ' '
396:  ADD   147: 32 = ' '
397:  WRITETINT 148: 32 = ' '
398:  WRITETSTRING 149: 32 = ' '
399:  LOADFROMADDR 150: 110 = 'n'
400:  LOADFROMADDR 151: 116 = 't'
401:  ADDTOPTR 152: 117 = 'e'
402:  LOADFROMADDR 153: 4 = Ctrl-D
403:  ADDTOPTR 154: 58 = ':'
404:  READING 155: 58 = ':'
405:  SAVETOADDR 156: 32 = ' '
406:  SAVETOADDR 157: 4 = Ctrl-D
407:  SAVETOADDR 158: 158 = 'a'
408:  ADDTOPTR 159: 4 = Ctrl-D
409:  PUSHLLOCADDR 160: 3 = Ctrl-C
410:  ADDTOPTR 161: 0 = Ctrl-@ 0: PTR TO 10001
411:  SAVETOADDR 162: 0 = Ctrl-@ 1: PTR TO 10001
412:  JUMP 359: 6 = Ctrl-F
413:  RETURN 4 163: 224

Data memory dump
76: 109 = 'm' 164: 105 = 'i'
77: 97 = 'a' 165: 58 = ':'
locations 10045 – 10259 omitted here

Data memory--addr 0 to
top of stack, and
allocated heap locs:
0:  PTR TO 10001
1:  PTR TO 10009
2: 100 = 'd'
3: 0 = Ctrl-@ 107: 101 = 'e'
4: 69 = 'E'
5: 110 = 'm'
6: 116 = 't'
7: 101 = 'e'
8: 114 = 't'
9: 32 = '
10: 110 = 'm'
11: 117 = 'u'
12: 109 = 'm'
13: 98 = 'b'
14: 101 = 'e'
15: 114 = 't'
16: 32 = '