Trace Table Guidelines for Nested Loops

Trace tables are useful for understanding the execution of code. Each column represents the content of a variable, and each row represents a moment in time.

Earlier in the semester, we used trace tables to study conditional statements and single loops. Now we want to use trace tables to study nested loops. When doing this, it is important to follow a few guidelines so that our tables accurately represent how code is executed.

Guideline 1: Each value represents a variable assignment.

If a trace table includes a column for every variable, then the number of values should equal the number of assignments. This implies that a value will only appear more than once on consecutive rows of a column if a variable is assigned the same value that it already stores.

Consider the following code and trace table:

```
int x = 0;
int y = 1;
y = 1;
```

x	y
0	1
	1

Each value in the table corresponds to an assignment statement in the code. The value 1 appears twice in the y-column because the variable y is initialized to 1 and then overwritten with the same value.

The value 0 appears in the x-column because the variable x is initialized to 0. The value does not appear again on the last row because x is not assigned another value. *Do not copy a value to a lower row just to fill empty space.*

Guideline 2: Include the final value of each loop variable.

If a trace table includes a column for a variable that appears in a loop condition, include the value that causes the loop to stop.[1]

Consider the following code and table:

```
int var = 10;
for (int idx = 0; idx < 2; ++idx)
{
    var = var – idx;
}
```

idx	var
	10
0	10
1	9
2	

The loop stops after idx is assigned the value 2. When this happens, the body of the loop is skipped (so var is not assigned the value 7), but the final value of idx is included on the last row of the table.

[1] This is covered by Guideline 1; however, omitting the final value is a common mistake, so it's worth emphasizing.

Guideline 3: Each row represents a single moment in time.

Scanning horizontally across a trace table should tell us the simultaneous values of all the variables at some instant in time. If a column is empty, the value of the variable is indicated by the closest value on a higher row.

To give trace tables this property, it may be necessary to include vertical space to properly align the values. This is usually the case for nested loops, since the inner loop can change a variable many times before the outer loop changes a different variable once.

Aside on Alternative Value Alignments: There are often multiple correct ways to align the values in a trace table.

Consider the following two tables, which both trace the example code from Guideline 2:

idx	var
	10
0	10
1	9
2	

idx	var
0	10
1	10
2	9

In the first table, the top row shows the variable values before entering the loop. The variable var has been assigned 10, while idx has not been declared. The middle two rows show the values at the end of the loop after the first and second iteration. The last row shows the values after exiting the loop. The column for var is empty, but its value at this time is 9, which can be seen from the previous row.

The second table traces the same code, but the rows represent different moments in time. Each row shows the variable values immediately after evaluating the loop condition.

When following Guideline 3, don't get hung up on finding the best alignment. Instead, simply ensure that each row corresponds to some line in the code.

Example 1: Independent nested loops

```
int size = 4;
int sum = 0;
// First row shows values here.
while (size > 0)
{
    for (int j = 0; j < 3; ++j)
    {
        sum = sum + j;
        // First three rows of each
        // block show values here.
    }
    size = size - 1;
    // Last row of each block shows
    // values here.
}
```

size	j	sum
4		0
	0	0
	1	1
	2	3
3	3	
	0	3
	1	4
	2	6
2	3	
	0	6
	1	7
	2	9
1	3	
	0	9
	1	10
	2	12
0	3	

The code in this example has a nested inner loop that executes three times on each iteration of the outer loop. That is, the number of inner-loop iterations is independent of the outer-loop iteration.

To highlight this feature of the code, I've added a bold outline to the block of rows that traces each iteration of the outer loop. Notice that the number of rows in each block is the same.

In order to follow Guideline 3, vertical space is added to the first column in each block of rows. The space aligns the value assigned to size with the final value of j. This indicates that the value of size does not change while the inner loop is executing.

Aside on Rectangular Nested Loops: Dr. Trytten describes independent nested loops as having a "rectangular" shape. To understand why, consider this alternative trace table:

size\j	0	1	2	3
4	0	1	3	
3	3	4	6	
2	6	7	9	
1	9	10	12	
0				

Rather than show the values assigned to each variable in a separate column, the values of the outer loop variable (size) are shown in the first column, and the values of the inner loop variable (j) are shown in the first row. These values serve as labels for the remaining cells, which show the values assigned to the variable sum. For example, on the outer-loop iteration where size is 3 and the inner-loop iteration where j is 2, sum is assigned the value 6.

Notice that the block of values assigned to sum (outlined in bold) is shaped like a rectangle. This is because the number of inner-loop iterations is the same on each outer-loop iteration.

Example 2: Dependent nested loops

```
int[] data = {1, 3, 5, 9, 7};
for (int idxOut = 0; idxOut < data.length; ++idxOut)
{
    int outer = data[idxOut];
    for (int idxIn = idxOut + 1; idxIn < data.length; ++idxIn)
    {
        int inner = data[idxIn];
        // All but the last row of each block shows values here.
        if (outer > inner)
        {
            return false;
        }
    }
    // Last row of each block shows values here.
}
return true;
```

idxOut	outer	idxIn	inner
0	1	1	3
		2	5
		3	9
		4	7
		5	
1	3	2	5
		3	9
		4	7
		5	
2	5	3	9
		4	7
		5	
3	9	4	7

The code in this example has an inner loop that executes a different number of times on each iteration of the outer loop. That is, the number of inner-loop iterations depends on the outer-loop iteration.

Like the previous example, I have added a bold outline to the block of rows that traces each iteration of the outer loop. Notice that the number of rows in each block decreases by 1 on successive outer-loop iterations.

Vertical space is added to the first two columns in each block of rows. The space aligns the values assigned to idxOut and outer with the initial values of idxIn and inner. This indicates that idxOut and outer do not change while the inner loop is executing.

Aside on Triangular Nested Loops: Dr. Trytten describes dependent nested loops as having a "triangular" shape. To understand why, consider this alternative trace table:

idxOut\idxIn	1	2	3	4	5
0	false	false	false	false	
1		false	false	false	
2			false	false	
3				true	

As in the aside to Example 1, the values of the outer and inner-loop variables are shown in the first column and row respectively. These values label the remaining cells, which show the result of the comparison outer > inner.

The block of comparison results is shaped like a triangle. This is because the number of inner-loop iterations changes with each outer-loop iteration.

Example 3: Think-Pair-Share

```
String[] names = {"Abby", "Raven", "Jazz", "Daisy"};
int namesSize = 3;  // names is oversized (size is 3; capacity is 4)
// First row shows values here.
for (int idx1 = namesSize - 1; idx1 >= 0; --idx1)
{
    for (int idx2 = namesSize - 2; idx2 >= 0; --idx2)
    {
        names[idx1] = names[idx2];
        // All but the first and last row show values here.
    }
}
// Last row shows values here.
```

idx1	idx2	namesSize	names[0]	names[1]	names[2]	names[3]
		3	"Abby"	"Raven"	"Jazz"	"Daisy"
2	1				"Raven"	
	0				"Abby"	
	-1					
1	1			"Raven"		
	0			"Abby"		
	-1					
0	1		"Abby"			
	0		"Abby"			
	-1					
-1						

The trace table shows that these nested loops are independent (i.e., rectangular). The inner loop executes twice on each iteration of the outer loop.

As with the table in Example 1, the first row shows the values of variables initialized before entering the outer loop. (An alternative alignment would show the initial values of idx1 and idx2 on the first row, rather than the second.) Including an initialization row like this usually makes the table easier to read.