# Trace Tables

The first section gives a broad overview of what trace tables are and provides a range of examples for **WHILE** loops, **FOR** Loops and nested **FOR** Loops.

The next section provides a guide to completing a trace table for a linear search algorithm.

The final section provides guidance on developing student understanding of trace tables, with ideas to introduce them in the early stages of programming.

# What are trace tables?

Trace tables are tables that consist of columns, each representing a variable, a condition, or an output in an algorithm, although not every variable, condition or output need be included in a trace table.

The purpose of the table is that we can run through an algorithm and simulate what a computer would do if the program were executed. We complete the table to show how the values within the variables change, what the conditions would evaluate to, and what outputs would be generated.

There are two main reasons that trace tables are used.

* The first is to determine what an algorithm does by running through it to see what happens as the algorithm runs.
* The second is to test the logic of an algorithm in case there are errors that are not easily spotted.

The following examples show some basic algorithms and their trace tables.

### Example 1

The algorithm below contains 1 variable (num), 1 condition (num < 500) and 1 output (**OUTPUT** num). The trace table shows how this algorithm would run if the user entered an input of 4.

num ← **USERINPUT WHILE** num < 500

num ← num \* 2

**ENDWHILE**

**OUTPUT** num

|  |  |  |
| --- | --- | --- |
| **num** | **num < 500** | **OUTPUT** |
| 4 | True |  |
| 8 | True |  |
| 16 | True |  |
| 32 | True |  |
| 64 | True |  |
| 128 | True |  |
| 256 | True |  |
| 512 | False |  |
|  |  | 512 |

### Example 2

Although the algorithm above only contains one variable, trace tables can keep track of multiple variables by using additional columns. The algorithm below is an extension of the algorithm above, and this time the program will also keep track of how many times the while loop has repeated. In this example the user enters an input of 16.

num ← **USERINPUT**

count ← 0

**WHILE** num < 500

num ← num \* 2 count ← count + 1

**ENDWHILE**

**OUTPUT** num

**OUTPUT** count

|  |  |  |  |
| --- | --- | --- | --- |
| **num** | **count** | **num < 500** | **OUTPUT** |
| 16 | 0 | True |  |
| 32 | 1 | True |  |
| 64 | 2 | True |  |
| 128 | 3 | True |  |
| 256 | 4 | True |  |
| 512 | 5 | False |  |
|  |  |  | 512 |
|  |  |  | 5 |

### Example 3

Trace tables help you to determine how an algorithm will run and are especially useful when you have nested structures that require you to keep track of multiple variables. The algorithm below contains a simple **FOR** loop.

**FOR** i ← 1 **TO** 4

**OUTPUT** i \* 2

**ENDFOR**

The trace table for this example is shown below. Although no variable is re-assigned inside the **FOR** loop, there is an output which we can keep track of with a trace table.

|  |  |
| --- | --- |
| **i** | **Output** |
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |

### Example 4

The algorithm below has introduced additional elements which have an impact on how the algorithm runs. The best way to test this algorithm is to do a dry run using a trace table. The trace table, with a brief explanation is shown below.

num ← 0

total ← 0

**FOR** i ← 1 **TO** 3

**FOR** j ← 1 **TO** 3

num ← j \* i

total ← total + num

**ENDFOR ENDFOR**

**OUTPUT** total

Normally the order of variables in a trace table is the order in which they are first encountered in the algorithm or program.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **num** | **total** | **i** | **j** | **OUTPUT** |
| 0 | 0 | 1 | 1 |  |
| 1 | 1 |  | 2 |  |
| 2 | 3 |  | 3 |  |
| 3 | 6 | 2 | 1 |  |
| 2 | 8 |  | 2 |  |
| 4 | 12 |  | 3 |  |
| 6 | 18 | 3 | 1 |  |
| 3 | 21 |  | 2 |  |
| 6 | 27 |  | 3 |  |
| 9 | 36 |  |  | 36 |

The values for num and total are the first parts of this trace table that need to be completed. They are both set to 0 before the first **FOR** loops starts.

We then have two **FOR** loops, an outer **FOR** loop that uses i to count from 1 to 3, and an inner **FOR** loop that uses j to count from 1 to 3. When loops are nested, the inner **FOR** loop repeats entirely, for every time the outer **FOR** loop repeats once. You can think of this like the hours and minute hands on a clock. For every time the hour hand moves on one, the minute hand moves on sixty and goes round the entire clock face.

There are two variables that are re-assigned every time the inner **FOR** loop repeats. The trace table keeps track of how the values for i and j change, so keeping track of the way that num (in the line num ← j \* i) changes is straight forward. The trace table also keeps track of value stored in total, so adding to this total each time is much easier for us to follow.

# The flow of time in a trace table

In a trace table time flows in a left-to-right, top-to-bottom fashion, just like reading a book (at least in Western cultures). This means that finding the values of variables on which an expression depends, eg num ← j \* i, is just a case of going back through the table in a right-to-left, bottom-to-top fashion until you find the required column, then up to a cell with a value in it (the last time that variable's value changed).

The highlighted square in the table above shows how the new value of num has been arrived at. First you need to go up a row and then with the lower arrow just going right until it encounters a cell in the j column with a value in it (3) showing the latest value of j and the upper arrow going right to the i column and then up to the previous row, until it encounters a cell in the i column with a value in it (2).

## Step by step trace table for a linear search

The following breaks down the process of completing a trace table for a linear search. This document is not designed to offer a full and complete explanation of how a linear search works, but focuses on the steps that are required to take a specific algorithm and complete a trace table to prove the algorithm works correctly.

A linear search is an algorithm that is used to check if a specific item is present in a data structure. The algorithm loops over the data structure, comparing each item one at a time to the item that is being searched for. If the item is found the algorithm stops, but if every item in the structure is checked and no match is found, then the algorithm will alert the user that the item is not in the structure.

The pseudo-code for this algorithm is shown below and the line numbers have been included so they can be referred to at different stages of the explanation in the following pages. It is advised that you refer to this page to identify specific parts of the algorithm as the trace table is gradually completed.

**Note:** There are different ways of implementing a linear search, and while this is an accurate way of doing so, it is not the only way.

|  |  |
| --- | --- |
| 1 | arrayToSearch ← [4, 8, 15, 16, 23, 42] |
| 2 |  |
| 3 | found ← False |
| 4 | itemToFind ← **USERINPUT** |
| 5 |  |
| 6 | **FOR** i ← 0 **TO LEN**(arrayToSearch) – 1 |
| 7 | **IF** arrayToSearch[i] = itemToFind **THEN** |
| 8 | **OUTPUT** 'Item found at index', i |
| 9 | found ← True |
| 10 | **BREAK** # this will exit the loop structure |
| 11 | **ENDIF** |
| 12 | i ← i + 1 |
| 13 | **ENDFOR** |
| 14 |  |
| 15 | **IF** found = False **THEN** # or **IF NOT** found **THEN** |
| 16 | **OUTPUT** 'The item is not in the list' |
| 17 | **ENDIF** |

## Step by step trace table with the input 16

The following pages will step through the completion of a trace table for this algorithm when the input given will be 16. Rather than complete the entire trace table in one go, this will guide you through each stage of completing the trace table and reference the lines of code that are relevant as the trace table is being completed.

The structure that we are searching is called arrayToSearch, and is declared on line 1. In this implementation we will also start counting the data structure index at 0.

arrayToSearch ← [4, 8, 15, 16, 23, 42]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Index** | [0] | [1] | [2] | [3] | [4] | [5] |
| **Value** | 4 | 8 | 15 | 16 | 23 | 42 |

**Step 1**

The first step of a trace table is to complete the values that are declared at the start of the algorithm.

In this example the variables found and itemToFind are both given values at the start. The value for found is written into line 3 of the program, whereas the value for itemToFind is assigned a user input on line 4, and as already mentioned the input given to the program will be 16.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 |  |  |

**Note**: The array arrayToSearch is also declared at the start of the algorithm on line 1, but it does not have any column headings because its values do not change. If we were tracing a sort which rearranged the elements of the array then they would need to be included.

### Step 2

We then continue to move through the algorithm. The next line of the program that runs in line 6, where the **FOR** loop starts. Trace tables become more complex when programs include loops such as **FOR**, **WHILE** and **REPEAT**, but they also become more useful when testing such programs to see if an algorithm is working correctly and/or identifying where the problems in the logic of an algorithm might be.

**Hint**: When you see a loop in an algorithm that you are tracing, it is always useful to find where that loop ends, and then check which variables inside the loop you will have to trace. In this example the loop starts on line 6 and finishes on line 13. Inside the loop i, found and arrayToSearch[i] are all used (and, apart from i, could be modified) so we know those are the columns we need to include.

In this example the **FOR** loop initialises i to 0 on line 6, therefore we can enter that into the trace table. On line 7 the condition in the **IF** statement compares arrayToSearch[i] to itemToFind. We have just completed the table to state what the value of i is at this point, so we can now also determine that the value of arrayToSearch[i] at this point is the first item in the list (due to 0 based indexing).

4 (arrayToSearch[i] when i is 0) does not equal itemToFind (16) and so the two statements inside the **IF** statement are not executed.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 | 0 | 4 |

### Step 3

The algorithm then continues, and i increments by 1 on line 12, before looping back round to line 6. When i increments on line 12 we now create a new row on the trace table. The table now shows that i has started off with a value of 0 and then is changed to 1. On line 7 the program uses arrayToSearch[i] again. As mentioned above, the value of i was initially set to 0 and then incremented to 1, so now instead of being the first item in the list (4, at index 0) arrayToSearch[i] is now 8 (at index 1).

The condition on line 7 is still false as itemToFind is still 16 and

arrayToSearch[i] is 8. This means the code inside lines 8 to 10 still do not run.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 | 0 | 4 |
|  |  | 1 | 8 |

### Step 4

The next step in this example is very similar to Step 3. Line 12 increments the value of i by 1 again, which means it now has a value of 2. The loop repeats back round to line 6 and then the condition on line 7 is checked again.

With the value of i now being 2, arrayToSearch[i] is now the third item in the list, which has a value of 15. Again, this is not equal to itemToFind, the condition is False and lines 8 to 10 are skipped.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 | 0 | 4 |
|  |  | 1 | 8 |
|  |  | 2 | 15 |

### Step 5

With lines 8 to 10 being skipped, the next line executed is line 11, where i increments by 1 so it has a value of 3. Just as in step 2, step 3 and step 4, i is used to look at the next item in the list, and this time around arrayToSearch[i] has a value of 16.

Once again the condition on line 7 compares arrayToSearch[i] and itemToFind both of which have a value of 16. This condition is therefore True and so for the first time in this algorithm lines 8 to 10 are run.

In the **HINT** for Step 2 it was mentioned that it is essential to keep track of any variables that might be changed in a loop. This loop is now in its fourth iteration, and for the first time lines 8 to 10 run. Line 8 is an output which is not asked for in the trace table, but line 9 re-assigns the value for found, so we can change the value of found in the trace table to True.

Line 10 then runs for the first time in this program, and this breaks the **FOR** loop and takes us down to line 13 for the rest of the algorithm to be completed.

At this point we can see that none of the remaining lines of code will change any of the variables or values required in our trace table, so the trace is now complete.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 | 0 | 4 |
|  |  | 1 | 8 |
|  |  | 2 | 15 |
|  |  | 3 | 16 |
| True |  |  |  |

## Other possible correct trace tables

The trace table completed in step 5 contains 4 columns, and the key factor is that all four columns show the values changing in the correct way. The exact placement of the rows and cells is not always fixed, and so there can be different layouts that are still correct, and it may suit different people to complete trace tables with different styles. Some of the other ways of completing the trace table are outlined below with explanations.

### Additional layout 1

In this layout each change in a value is given a separate row in the table. This style shows a very clear run order, but more than double the number of rows are needed. With more complex algorithms, even more than this could be required.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False |  |  |  |
|  | 16 |  |  |
|  |  | 0 |  |
|  |  |  | 4 |
|  |  | 1 |  |
|  |  |  | 8 |
|  |  | 2 |  |
|  |  |  | 15 |
|  |  | 3 |  |
|  |  |  | 16 |
| True |  |  |  |

### Additional correct example 2

The original trace table completed in step 5 shows when the values for each column change. In this additional correct example, every cell is complete, even if the value does not change.

The advantage of this method is that making comparisons is clearer, but the disadvantage is that if the algorithm hasn’t been understood properly and changes need to be made, the trace table can begin to look very messy and difficult to read.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 |  |  |
| False | 16 | 0 | 4 |
| False | 16 | 1 | 8 |
| False | 16 | 2 | 15 |
| False | 16 | 3 | 16 |
| True |  |  |  |

The key point to both of these additional correct examples is that the values in each column change in the correct way.

* The column for found starts off as False as identified in line 3, and then is changed to True at the end when line 9 is executed.
* itemToFind starts off as 16 and does not change throughout the trace table.
* i starts as 0 and continues to count to 3 in increments of 1, but does not go beyond 3. This is because the item we are looking for is stored at index 3 in the array.
* arrayToSearch[i] starts off as 4 and then continues to show the value for each item one at a time until it gets to 16, but does not change again.

## Common mistakes

There are some common mistakes that can occur when completing trace tables, even if the person completing the table has fully understood the algorithm. Two of these common mistakes are outlined below, and both provide examples based on the linear search algorithm with an input of 16.

## Auto-completing rows and values without considering when the values should stop changing

In this example the person completing the trace table has identified that the loop could potentially count from 0 to 5, and that this will be used to cycle through each item in the list. Rather than tracing the values one at a time, they have assumed the program will continue and have completed the table without considering when the break may actually occur. They have also put the True value on the same row as the values of 4 and 16 for i and arrayToSearch[i] which, remembering the discussion about the flow of time in a trace table, suggests that the program set found to True when i was 2 and arrayToSearch[i] was 15 (alternatively the program could look into the future!)

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 |  |  |
|  |  | 0 | 4 |
|  |  | 1 | 8 |
|  |  | 2 | 15 |
| True |  | 3 | 16 |
|  |  | 4 | 23 |
|  |  | 5 | 42 |

## Completing columns up rather than down

Sometimes the person completing the trace table feels like they should complete every cell. As shown in the second additional correct example, this can be fine if the columns are completed down. In this example the person has completed the table with gaps, and then filled in any empty spaces by filling a cell with the value below the space.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **found** | | | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | | | 16 | 0 | 4 |
|  | True |  | 16 | 0 | 4 |
|  | True |  | 16 | 1 | 8 |
|  | True |  | 16 | 2 | 15 |
| True | | | 16 | 3 | 16 |

## Other examples

As stated, the trace table stepped through shows how the trace table is completed for the value of 16. There are 3 other complete examples of this algorithm shown below, but with different inputs.

Try completing the trace table for the following inputs, and then check them to see if you are correct.

1. user input of 26
2. user input of 4
3. user input of 42

### Example: User input of 26

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 26 | 0 | 4 |
|  |  | 1 | 8 |
|  |  | 2 | 15 |
|  |  | 3 | 16 |
|  |  | 4 | 23 |
|  |  | 5 | 42 |

**Example: User input of 4**

With a user input of 4, the algorithm will complete much more quickly. 4 is the first item in the list, so a linear search will check that item first, set found to True and break out of the loop.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 | 0 | 4 |
| True |  |  |  |

### Example: User input of 42

With a user input of 42, the algorithm will take the maximum number of comparisons possible with the end value of found still being True.

|  |  |  |  |
| --- | --- | --- | --- |
| **found** | **itemToFind** | **i** | **arrayToSearch[i]** |
| False | 16 | 0 | 4 |
|  |  | 1 | 8 |
|  |  | 2 | 15 |
|  |  | 3 | 16 |
|  |  | 4 | 23 |
|  |  | 5 | 42 |
| True |  |  |  |