The code Walter wrote is a C++ program which is difficult to read and maintain because no basic coding practices were followed: no author name, dates, or program description; no proper code indentation; no meaningful variable and function names; no comments to explain the logic or that add context; and it repeats itself unnecessarily.

1. The purpose of the code is to search for a user-provided number within the program's array of numbers using three different searching algorithms. If the number is found, its position is displayed for each algorithm; otherwise the user is informed that it is not found.

2. The name of each algorithm used by Walter is: linear search in function **`f1`**, binary search in function **`f2`** and jump search in function **`f3`**.

3. After the element **`419`** is entered by the user, it becomes the integer variable **`key`**, which is passed to the first searching algorithm function **`f1`** (linear search). This function searches for **`419`** by comparing it to each numbers in the array, starting from the first element and moving one by one through the array. If it finds **`419`**, it returns its index in the array; if it doesn't, it returns -1.

   Next, the key is passed to the second search algorithm function **`f2`** (binary search). Since the array is sorted, the binary search divides the array into halves, comparing the middle value with the **`key`**. If the middle value equals **`419`**, the function returns the index + 1. If **`419`** is greater than the middle value, the search continues in the right half; if smaller, in the left half. The process repeats over and over until **`419`** is found or not. If **`419`** is found, the function returns the index + 1; otherwise, it returns -1.

   Then, the key is passed to the third search algorithm function **`f3`** (jump search). This algorithm searches for **`419`** by jumping ahead in blocks of size √n (where n is the length of the array). It first jumps forward in steps of √100 (which is 10) until it either finds a block where the value is larger than **`419`** or it reaches the end of the array. Then, it performs a linear search within that block to find **`419`**. If it finds **`419`**, it returns the index. If **`419`** is not found, it returns -1.

   Finally, if the sum of the results from the three algorithms is not -3 (meaning that the element was found in at least one of the searches), the program will display the location of **`419`** for each of the three algorithms (adjusted as per function index calculation).

4. After the element **541** is entered by the user, it will become the integer variable **key**, which is then passed to the first searching algorithm function **f1** (linear search). The function will search for the element **541** by comparing it to each of the numbers in the program's array, starting from the first element and going through all the elements until it reaches the last one. Once it finds **541** at index 99, the function returns the index.

   Next, just as question 3 described it, the program calls **f2**, which performs a binary search. Since the array is sorted, the binary search algorithm will divide the array in half over and over, narrowing down the search space until it reaches the position of **541**, which is at index 99. The function then returns the index of **541** plus one, which would be 100.

   The key is then passed into the third search function, **f3**, which uses jump search. It will jump in blocks, checking a subset of the array at each step. Eventually, after jumping to the block that contains **541**, the search narrows down to find the element at index 99.

   Finally, if the sum of the results from the three algorithms is not -3, the program will display the position of **541** for each of the three algorithms. Since **541** is found in all three, the program will display its location (adjusted as per function index calculation).

5. When the user enters **two** as the input, it is a string. However, the program expects an integer input (using **cin >> key**). Since **two** is not an integer, the input operation fails, and the value stored in key remains unchanged (either set to 0 or whatever was in key before). The functions **f1**, **f2**, and **f3** expect an integer value to perform the search, but because the input is invalid, the functions won't find the string **two** in the array, as they are designed to work with integers. When the invalid input is passed to the search functions, they will either use the default or invalid integer stored in key and none of the functions will find this value in the array. As a result, the sum of the results from the three algorithms will be -3 (since all three functions return -1 when the value is not found), triggering the else statement in the **main** function, which displays "Requested key not found.

6. After entering **31*1**, the program reads only the numeric part before the non-numeric character *****, so the value **31** will be stored in **key**. Afterward, the program processes **31** using the three search algorithms as previously described. The result will be the location of **31** in the array, displayed by each algorithm (adjusted as per function index calculation).

7. The sequence of numbers stored in array **`a`** is the first 100 prime numbers, starting from 2 and ending at 541, which are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541.

8. There are three user-defined functions (**`f1`**, **`f2`**, and **`f3`**), which implement three different searching algorithms. Also, the standard library function **`min`** is used within **`f3`**. So it is safe to say that there are 4 functions used in the code, unless we consider **main()** a function as well, in which case there are 5 functions.

9. One of the weaknesses of this code is **input validation**. If the user enters non-integer values like **`two`** or **`31*1`**, it behaves unpredictably and may return incorrect results. To fix this, we could add input validation to ensure only integers are accepted, and prompt the user to enter a valid integer if the input is anything other than an integer.

   Another weakness is that the program **terminates after a single search**. This means that the user must restart the program to search for another number, which is inconvenient. A fix for this would be to implement a loop that allows multiple searches without restarting the program. Additionally, providing an option to quit would improve the convenience of using the program.

   A third weakness is the **output format**. The same position is displayed for each algorithm on three different lines without distinguishing between the algorithms. This can be confusing, as it looks like the same result is being repeated. To improve this, we could modify the output to clarify which search algorithm returned which result, providing more context, which goes in line with "comments that add context" practice described in 6 coding best practices link within the Activity 1 read this article links. Additionally, if we wanted to compare the performance of each algorithm, we could include the **execution time** for each search.

   Another issue is that the **binary search function `f2`** returns mid + 1, meaning its index is inconsistent with the other two algorithms. In C++ arrays, indexing starts from 0, and the "+1" could cause confusion. To fix this, we could adjust **`f2`** to return mid without the "+1" so it aligns with the indexing behavior of the other functions. Or edit the other 2 functions so they all use the same notation/logic.

The logic in the **`main()`** function also has readability issues. The condition **`if((l1 + l2 + l3) != -3)`** is confusing. The variable names l1, l2, l3 look similar to 11, 12, and 13, which can lead to misreading the meaning of what they represent. On top of that, this condition assumes that all three algorithms will return **`-1`** when the key isn't found. A better approach would be to use separate conditions for each search result, which would make the code easier to understand and maintain. This would also align with good coding practices, such as **low coupling and high cohesion**.

Finally, if we are going through the code itself (fixme.cpp), there is a lack of **comments or documentation** among other things. The code does not explain what each function does, which makes it harder to maintain or debug. To fix it, we should add comments to explain the purpose of each function and important lines of code.