1. Github
2. Txt file in github
3. Idk man
4. Idk man
5. Code: github

The readability of my code is about the same as original code. Both uses similar logic to iterate through the matrix. The difference is how it determines rejects. Instead of looking for looking for not zeros it looks for zeros and increments though the row till it hits a non-zero. One it hits a non-zero it breaks out of the row and moves to the next. If it increments to where j is equal to the length of the array, then it breaks out of incrementing though the matrix because then it knows it has pass through a row of all zeros. It then outputs “i” which is what row the program is on. This is less intuitive than reding though a go to statement because you have to run though the loop and see what break leaks where to determine what the output would be. The original code and the new code has the same time and space complexity because both could have to run though the whole matrix resulting in a complexity of n^2. The original code is slightly more compact and only has a minor advantage on readability. This means that it is up to the preference of the programmer on which to use.

1. Code: github

In the case of non-nested and nested both are reliable, but the nested version has higher reliability. This increases with scaling the input with more than 3 integers. This works similar to a branch and bound algorithm. The problem does not have to go down the whole decision tree in direction it knowns are already incorrect. This it is able to greatly reduce the time complexity, especially as the input size increases. The non-nested version has a worst case time complexity of n!, because it could be the last case checked. This is less efficient than nesting. The non-nested version has a readability advantage. It is very easy to see which conation al is being tested, while in the nested version it would take more time tracing the code.

1. Tombstones have a higher time a space complexity, because tombstones are never deallocated from memory and reclaimed. This creates ineffective memory usage. Accessing a heap dynamic variable though this method requires an additional cycle to access. This introduces another level of time complexity. From a safety perspective it does prevent access to deallocated memory as the pointer will point to unchanging tombstone. The lock-and-key method has pointers represented as ordered pairs, keys and address. Heap dynamic variables are represented as the storage for the variable plus a cell that stores the “lock” integer value. When a Heap Dynamic variable is allocated a lock value is also created in the heap dynamic variable, the lock cell, and the key cell of the pointer. Accessing the dereferenced pointer compares the key value to the lock value. Matching values are allowed access. In the event of mismatched values, it returns a run-time error. When a heap dynamic variable is deallocated its lock value is set to an illegal value. When it’s pointer dereferenced, tis address value will still be not change, but the key value will not work anymore, so access is denied. This prevents memory leaks and dangling pointers. However it does has time and space complexity overhead, all be it less than tombstones. Each block and pointer in the heap requires an extra word of memory space. Also, the comparing lock and key values has extra time complexity associated with it. The lock and key method has higher security associated as without the proper key it is impossible to access the locked memory. While with tombstone there is nothing preventing access to deallocated memory areas. This is easily done if you have the address of the memory segment you are looking for.