Berjhan Rose Alpeche

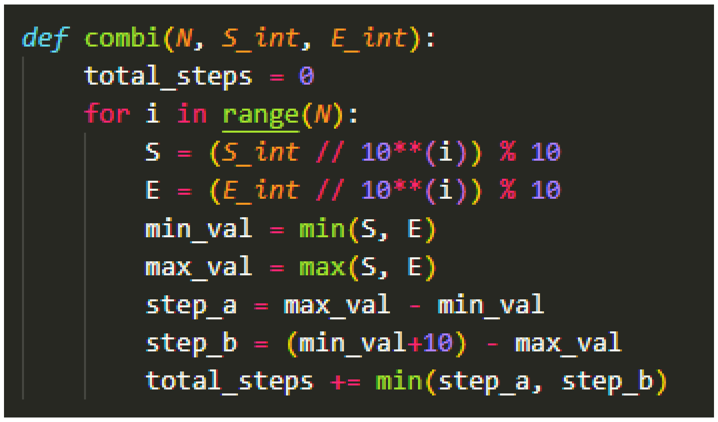
2019-01592

COE 163 SE01

For Software Exercise 01, I opted to use Python and C because these are the most familiar to me.

Algorithm Correctness

PYTHON



```python
def combi(N, S_int, E_int):
    total_steps = 0
    for i in range(N):
        S = (S_int // 10**(i)) % 10
        E = (E_int // 10**(i)) % 10
        min_val = min(S, E)
        max_val = max(S, E)
        step_a = max_val - min_val
        step_b = (min_val+10) - max_val
        total_steps += min(step_a, step_b)
```

```
def combi(N, S_int, E_int):
--> the function "combi" takes 3 integers as argument

total_steps = 0
--> initializes the total_steps as 0 in the beginning of the function

for i in range(N):
--> the function repeats N times starting 0 to N-1. N is the number of digits a
combination lock has.

S = (S_int // 10**(i)) % 10
E = (E_int // 10**(i)) % 10
--> extracts the i-th digit from the integers and stores them into S and E
respectively. Initially done using arrays, but I figured using arithmetic instead
of memory referral should be more efficient.

min_val = min(S, E)
max_val = max(S, E)
--> determines the smaller and larger of the digits
```

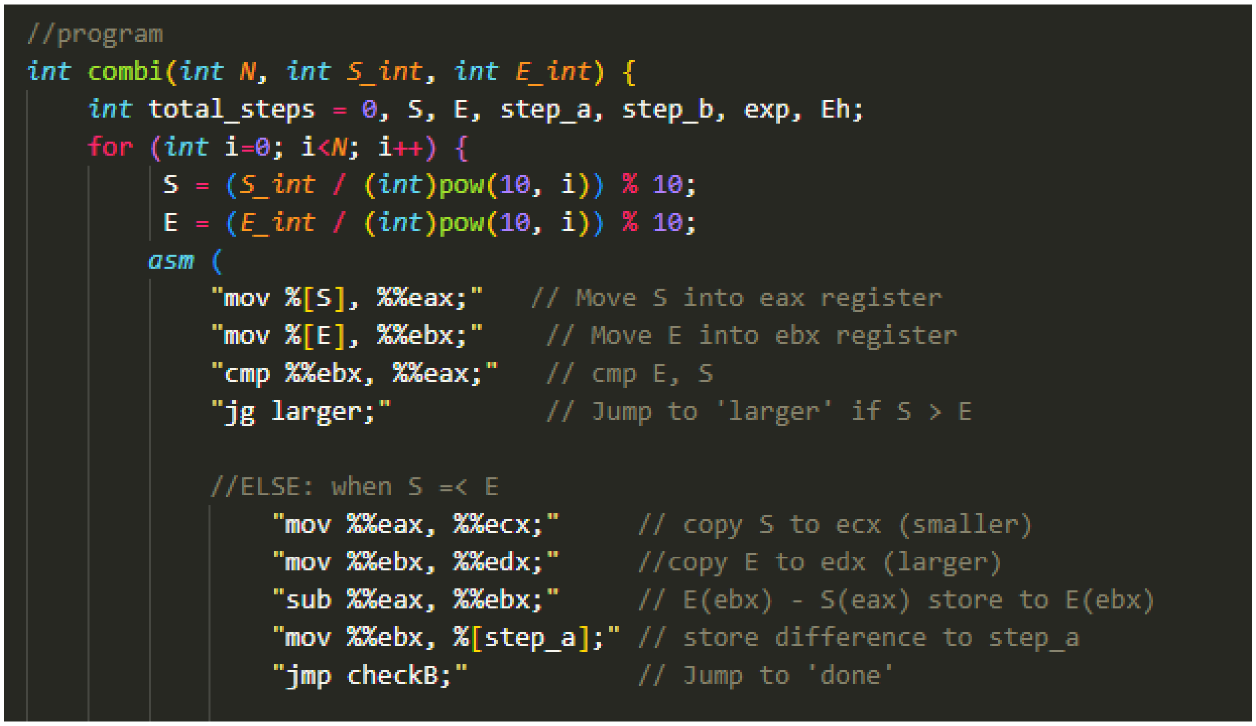
Because the dial numbers wrap around such that 0 and 9 are connected, there are two ways to measure the distance (step_a and step_b). The lesser step will be stored. The dials can turn clockwise and counterclockwise, so decrement and increment are both counted as unsigned step; there is no positive or negative direction.

```
step_a = max_val - min_val
```
--> subtracts smaller digit from larger digit

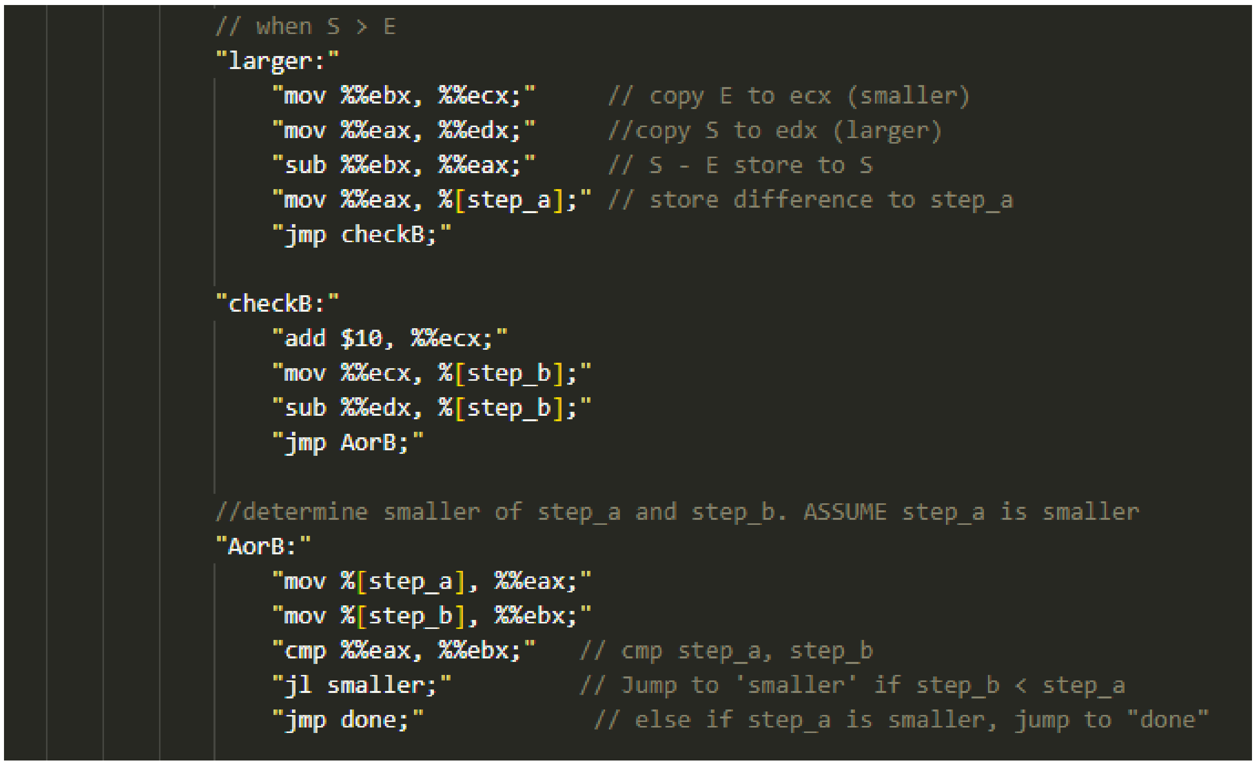
```
step_b = (min_val+10) - max_val
```
--> adds 10 to smaller digit and subtracts larger digit from it. Simulates the scenario of getting the distance going the opposite way.

```
total_steps += min(step_a, step_b)
```
--> takes the smaller of step_a and step_b and stores it. Once the loop is done, total_steps holds the minimum number of steps needed to get to the unlock configuration

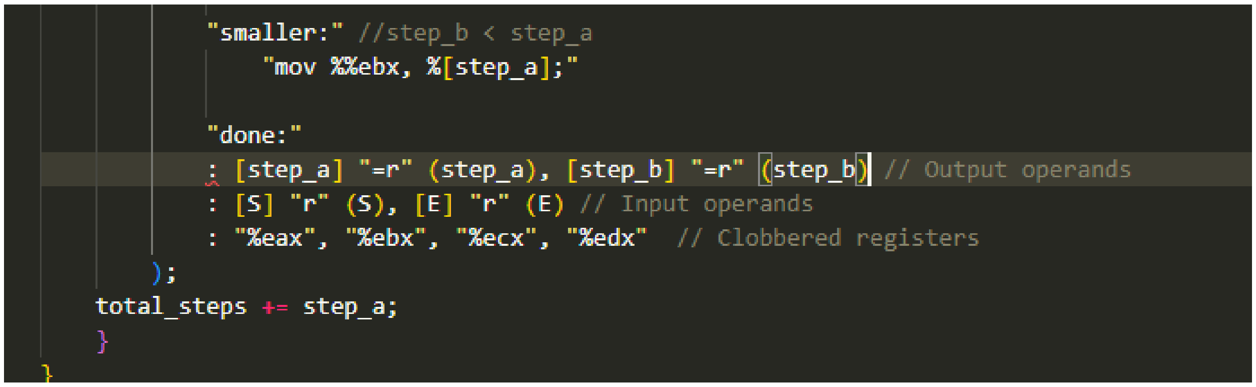
C + ASSEMBLY



```
//program
int combi(int N, int S_int, int E_int) {
    int total_steps = 0, S, E, step_a, step_b, exp, Eh;
    for (int i=0; i<N; i++) {
        S = (S_int / (int)pow(10, i)) % 10;
        E = (E_int / (int)pow(10, i)) % 10;
        asm (
            "mov %[S], %%eax;"   // Move S into eax register
            "mov %[E], %%ebx;"    // Move E into ebx register
            "cmp %%ebx, %%eax;"   // cmp E, S
            "jg larger;"          // Jump to 'larger' if S > E

            //ELSE: when S =< E
                "mov %%eax, %%ecx;"     // copy S to ecx (smaller)
                "mov %%ebx, %%edx;"     //copy E to edx (larger)
                "sub %%eax, %%ebx;"     // E(ebx) - S(eax) store to E(ebx)
                "mov %%ebx, %[step_a];" // store difference to step_a
                "jmp checkB;"           // Jump to 'done'
```



```
            // when S > E
            "larger:"
                "mov %%ebx, %%ecx;"     // copy E to ecx (smaller)
                "mov %%eax, %%edx;"     //copy S to edx (larger)
                "sub %%ebx, %%eax;"     // S - E store to S
                "mov %%eax, %[step_a];" // store difference to step_a
                "jmp checkB;"

            "checkB:"
                "add $10, %%ecx;"
                "mov %%ecx, %[step_b];"
                "sub %%edx, %[step_b];"
                "jmp AorB;"

            //determine smaller of step_a and step_b. ASSUME step_a is smaller
            "AorB:"
                "mov %[step_a], %%eax;"
                "mov %[step_b], %%ebx;"
                "cmp %%eax, %%ebx;"   // cmp step_a, step_b
                "jl smaller;"         // Jump to 'smaller' if step_b < step_a
                "jmp done;"            // else if step_a is smaller, jump to "done"
```



```
            "smaller:" //step_b < step_a
                "mov %%ebx, %[step_a];"

            "done:"
            : [step_a] "=r" (step_a), [step_b] "=r" (step_b) // Output operands
            : [S] "r" (S), [E] "r" (E) // Input operands
            : "%eax", "%ebx", "%ecx", "%edx"  // Clobbered registers
        );
    total_steps += step_a;
    }
}
```

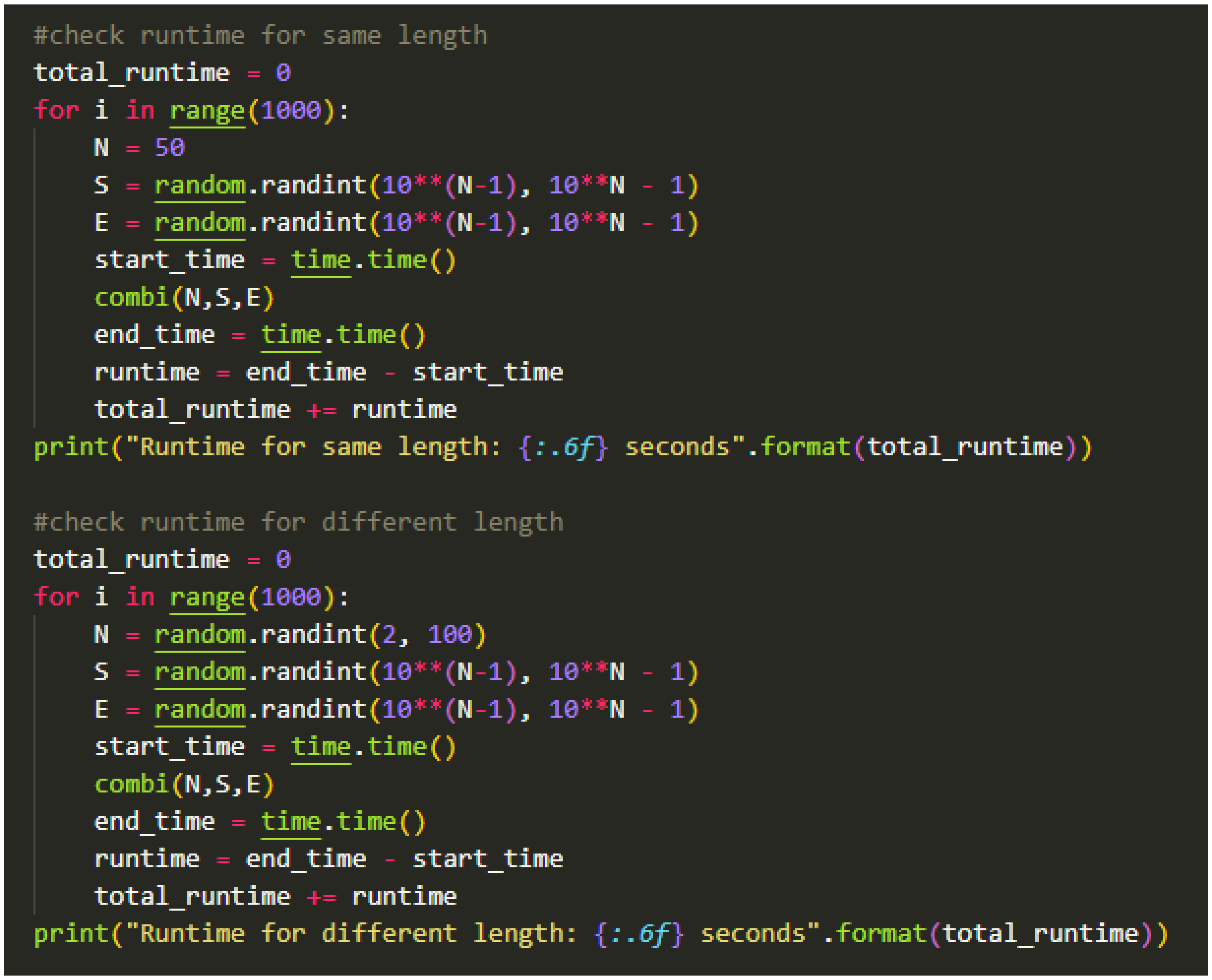
For the algorithm in C+assembly, I followed the same logic done in python.
Extracting the digits was done in C. This entails heavy arithmetic that would be tedious to code in Assembly.

The comparison of S and E digits is done in assembly.
First block of assembly code stores S and E data into registers then a "cmp" is executed to determine which has the larger value.
The smaller value is subtracted from the larger value and the difference is stored in step_a.
Next is in branch "checkB". Ten is added to the smaller value and their sum is then subtracted by the larger value. The difference is stored in step_b.

Finally, in the last branch "AorB", we get the smaller of step_a and step_b. This is the final output of the assembly block. It is then then iteratively added to total_steps in C to get the minimum steps needed to get the unlock configuration.

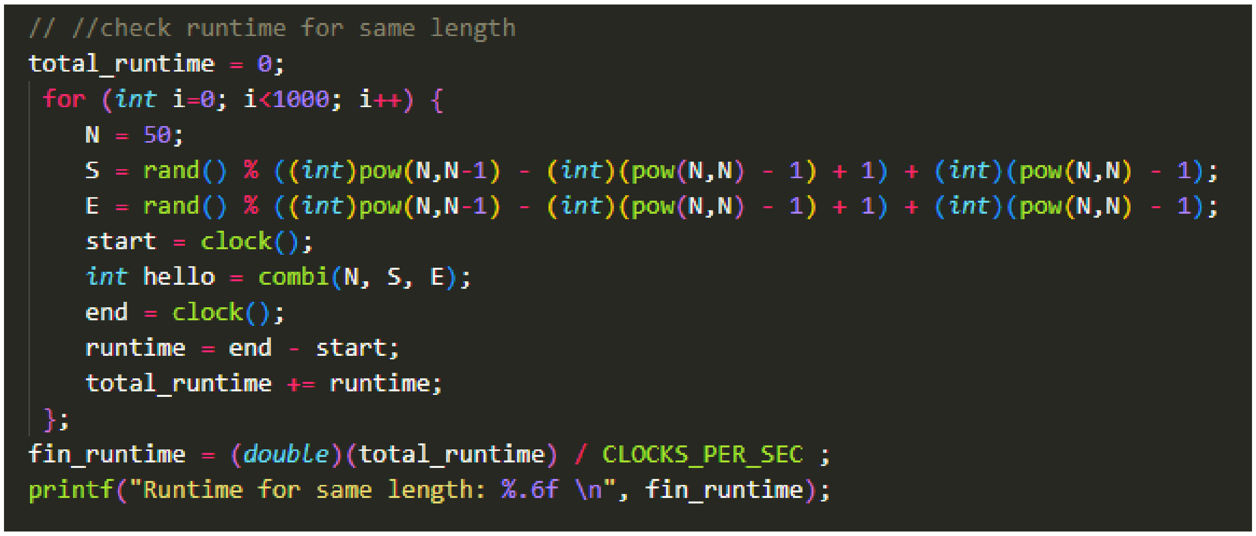
Profiling

PYTHON

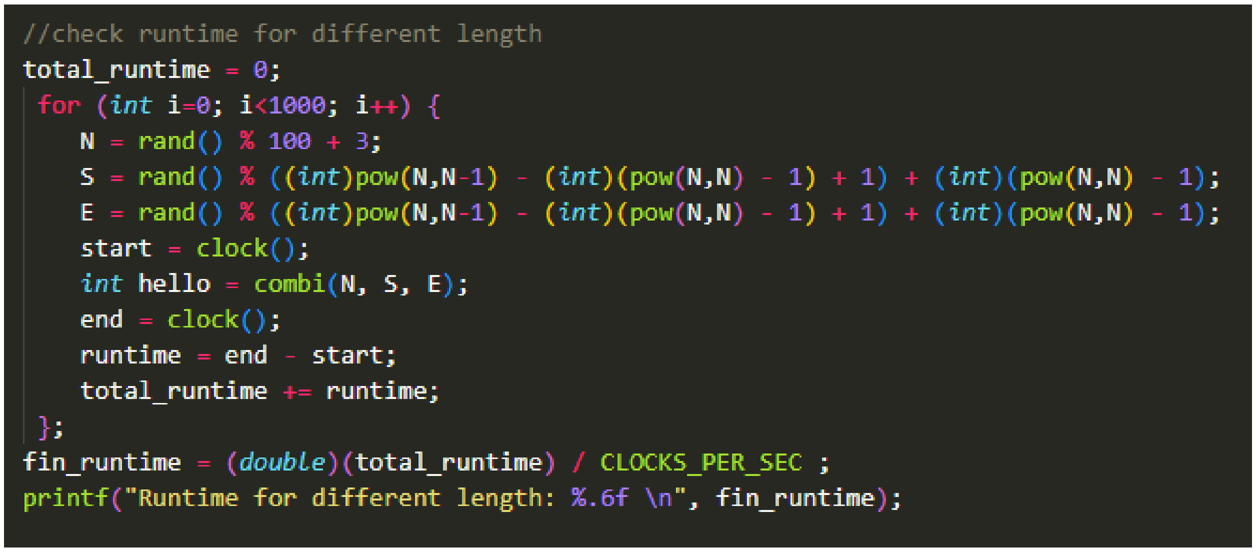


```python
#check runtime for same length
total_runtime = 0
for i in range(1000):
    N = 50
    S = random.randint(10**(N-1), 10**N - 1)
    E = random.randint(10**(N-1), 10**N - 1)
    start_time = time.time()
    combi(N,S,E)
    end_time = time.time()
    runtime = end_time - start_time
    total_runtime += runtime
print("Runtime for same length: {:.6f} seconds".format(total_runtime))

#check runtime for different length
total_runtime = 0
for i in range(1000):
    N = random.randint(2, 100)
    S = random.randint(10**(N-1), 10**N - 1)
    E = random.randint(10**(N-1), 10**N - 1)
    start_time = time.time()
    combi(N,S,E)
    end_time = time.time()
    runtime = end_time - start_time
    total_runtime += runtime
print("Runtime for different length: {:.6f} seconds".format(total_runtime))
```

C



```c
// //check runtime for same length
total_runtime = 0;
 for (int i=0; i<1000; i++) {
    N = 50;
    S = rand() % ((int)pow(N,N-1) - (int)(pow(N,N) - 1) + 1) + (int)(pow(N,N) - 1);
    E = rand() % ((int)pow(N,N-1) - (int)(pow(N,N) - 1) + 1) + (int)(pow(N,N) - 1);
    start = clock();
    int hello = combi(N, S, E);
    end = clock();
    runtime = end - start;
    total_runtime += runtime;
 };
fin_runtime = (double)(total_runtime) / CLOCKS_PER_SEC ;
printf("Runtime for same length: %.6f \n", fin_runtime);
```



```c
//check runtime for different length
total_runtime = 0;
 for (int i=0; i<1000; i++) {
    N = rand() % 100 + 3;
    S = rand() % ((int)pow(N,N-1) - (int)(pow(N,N) - 1) + 1) + (int)(pow(N,N) - 1);
    E = rand() % ((int)pow(N,N-1) - (int)(pow(N,N) - 1) + 1) + (int)(pow(N,N) - 1);
    start = clock();
    int hello = combi(N, S, E);
    end = clock();
    runtime = end - start;
    total_runtime += runtime;
 };
fin_runtime = (double)(total_runtime) / CLOCKS_PER_SEC ;
printf("Runtime for different length: %.6f \n", fin_runtime);
```

The same logic was used when profiling the python and C+assmbly code. Test case 1 executes the program with a fixed N. Test case 2 executes it at variable N.

The profiling runs the program in a loop that iterates 1000 times, and the runtime is recorded.
N, S, and E values (except for test case 1 where N is fixed) are randomly generated per iteration.
S and E will be integers of N digits. Time is only recorded when the "combi" function is called. So, the runtime for generating the inputs is not included.
In the test case where N is constant, N is fixed at 50.
For the test case where N is variable, N is capped at 100 (this was a given constraint).

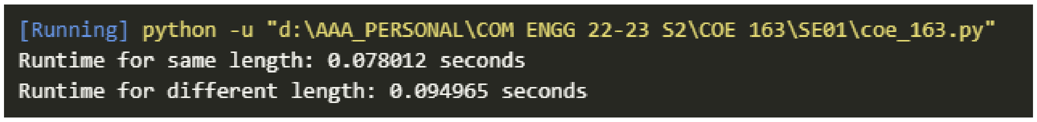
Due to variability, I roughly assumed that N would average to its median, 50. So for test case 1, N is 50, for a more equal comparison. This is a huge assumption, as we don't know the distribution of random numbers in test case 2 per runtime.

On both test cases, the test case 2 with variable N runs slower than test case 1 with fixed N. An inherent limitation of this approach is the random generator, as it may select values that are skewed on the larger side.

Algorithm Comparison

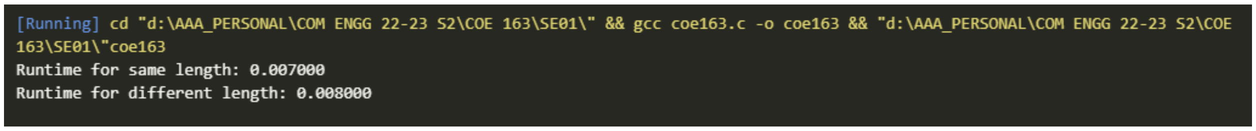
<u>RUNTIME</u>

PYTHON



```
[Running] python -u "d:\AAA_PERSONAL\COM ENGG 22-23 S2\COE 163\SE01\coe_163.py"
Runtime for same length: 0.078012 seconds
Runtime for different length: 0.094965 seconds
```

C+ASSEMBLY



```
[Running] cd "d:\AAA_PERSONAL\COM ENGG 22-23 S2\COE 163\SE01\" && gcc coe163.c -o coe163 && "d:\AAA_PERSONAL\COM ENGG 22-23 S2\COE 163\SE01\"coe163
Runtime for same length: 0.007000
Runtime for different length: 0.008000
```

Based on the recorded runtime, C+assembly runs significantly faster on both test cases. At fixed length, python runs for 0.078012 s and C+asm runs only for 0.00700 s. At variable length, python runs for 0.094965 s and C runs for 0.008000 s.

Python is a high level language that is interpreted at runtime to machine code instruction. This added layer on execution adds overhead time which makes it slower.

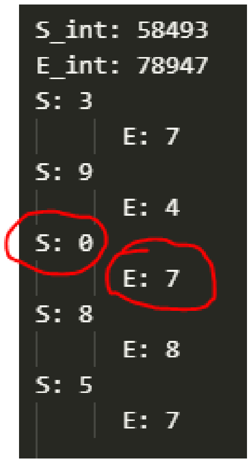
C+assembly is faster because it's a low-level language. Its codes directly corresponds to machine code instructions so it is executed directly by the computer without need for interpretation or translation.

Furthermore, python is also dynamically typed. Unlike static-typed languages, python's variable type is not determined prior to execution and the interpreter spends time type checking, which further slows down program execution.

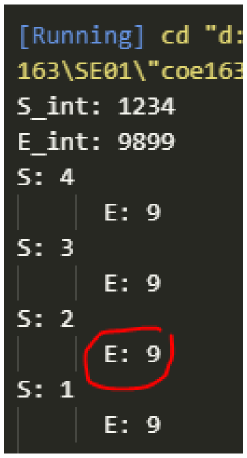
<u>PROGRAMMING EFFORT</u>

I worked side by side with chatGPT to get the correct syntax for the logic I formulated. I was able to execute the algorithm in Python immediately. It took me an entire day to get it done in Assembly. The language was less intuitive and I encountered a lot of bugs. Had I known some of the constraints ahead of time, I would not have spent on it too long. Example, the values of S and E keeps getting jumbled even if the logic was correct. Turns out the solution to this was to first store them into temporary registers and do the manipulation from there.

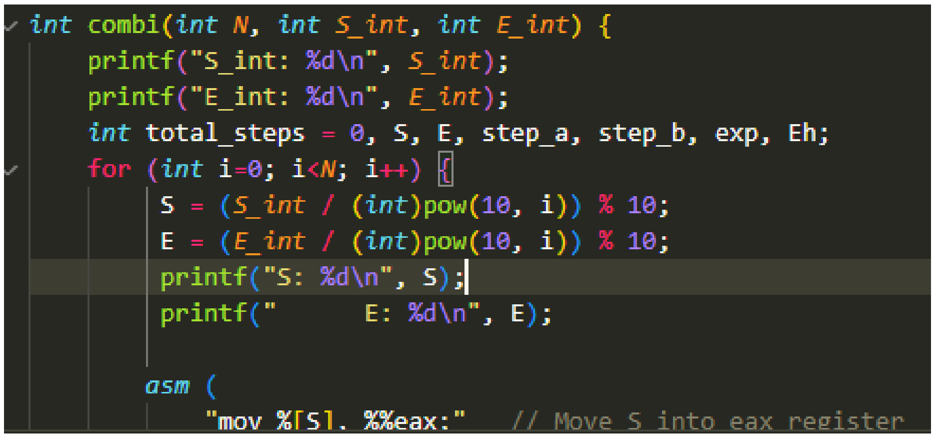
**Unresolved bug in C



```
S_int: 58493
E_int: 78947
S: 3
        E: 7
S: 9
        E: 4
S: 0
        E: 7
S: 8
        E: 8
S: 5
        E: 7
```



```
[Running] cd "d:
163\SE01\"coe163
S_int: 1234
E_int: 9899
S: 4
        E: 9
S: 3
        E: 9
S: 2
        E: 9
S: 1
        E: 9
```



```c
int combi(int N, int S_int, int E_int) {
    printf("S_int: %d\n", S_int);
    printf("E_int: %d\n", E_int);
    int total_steps = 0, S, E, step_a, step_b, exp, Eh;
    for (int i=0; i<N; i++) {
        S = (S_int / (int)pow(10, i)) % 10;
        E = (E_int / (int)pow(10, i)) % 10;
        printf("S: %d\n", S);
        printf("      E: %d\n", E);

        asm (
            "mov %[S], %%eax;"    // Move S into eax register
```

There is an unresolved issue on the C code when it was run on my machine. It may extract the wrong digit from the given integer input. As can be seen on the sample code above, the values are printed before the assembly code block is executed. The error lies in how C performs the arithmetic. That's why, given a test case, the program may output the wrong answer because the input is wrong.