CASTOR, John Danielle T
20[redacted]
3-31-23
SE01

I. Task

The task basically involves combination locks.

- Inputs (integers): N (0 < N <= 100) = number of combination locks, S = initial configuration, E = unlock configuration
- Output (integer): P = minimum steps to go from initial to unlock configuration (Step- one integer increment or decrement)

Use any preferred language (Python) and a mixture of C/C++/Rust and x86_64 assembly (C and x86_64 Intel).

Enclose the algorithm (for both languages) in a function, then in a loop that runs >=1000 times (100000 times). Then time profile the two programs using two test cases: 1. Same length, 2. Different lengths.

II. Algorithm

1. Take the inputs.
    a. Import random
    b. For same lengths, N was set to 4, then randomize (Python: randint(), C: rand()) integers within 1000-9999 for S and E.
    c. For different lengths, randomize an integer from 1-100 for N, then randomize integers from $10^{(N-1)}$ to (10^N – 1) for S and E.
2. Initialize output (global P, starts from 0)
3. Call the unlocker function.
    a. Takes in N, S, E
    b. Returns P
    c. Iterate through every digit of S and E from MSB to LSB.
        i. Floor divide the integer by $10^{(\text{place value})}$.
        ii. Divide the result by 10 then get remainder via modulo.
    d. Compare S_new and E_new (current digits) to obtain the minimum number of steps needed, use the conditions below:

```
if (E_new >= S_new):
    if (((E_new - S_new) >= 0) and ((E_new - S_new) <= 5)):
        P += E_new - S_new
    else:
        P += (10 - E_new) + S_new
else:
    if (((S_new - E_new) >= 0) and ((S_new - E_new) <= 5)):
        P += S_new - E_new
    else:
        P += (10 - S_new) + E_new
```

e. Add the minimum steps to P, for all iterations.
f. Return P

III. Algorithm Correctness

1. The conditional part of the code is the trickiest part. This was solved via divide and conquer.
    a. Case 1: E_new >= S_new
        i. Minimum step is basically their difference if their difference is within 0-5.
        ii. If not, the minimum step is the least distance of E_new to zero plus the least distance of S_new to zero plus one (since numbers are wrapped around).
        iii. If greater than 5 (E.g. 6), it is better to decrement (just need 4 steps).
    b. Case 2: E_new < S_new
        i. Same as Case 1 but S_new is now the minuend.
2. Sample test cases (at module) were correctly solved.
3. For the randomization (from II. 1):
    a. If N=5, the range for S and E is within $10^{(5-1)} – (10^5 - 1)$ = 10000-99999, which are all 5 digit numbers.
4. For obtaining the digits (from II. 3. c):
    a. E.g. floor($1239/10^3$) = floor(1.239) = 1
    b. 1 % 10 = 1
    c. E.g. floor($1239/10^2$) = floor(12.39) = 12
    d. 12 % 10 = 2
5. Adding current minimum steps for a digit to P for all iterations, will yield the minimum steps needed.

IV. Algorithm Issues

1. For Python, since the input is strictly integer, leading zeros are discarded (E.g. if S=01234, it will be read as 1234), thus the obtaining of digits will produce errors.
2. For C, numbers with ~>10 digits are processed incorrectly (capped at -2147483648), so minimum steps needed on some digits results to zero.

V. Time Profiling

Timers start from the start of loop, and end at end of loop, within the loop are the input randomizations and function call.

1. Import timer
2. Take a snapshot of time before the loop (100000 times).
3. Take another snapshot at end.
4. Subtract the final snapshot to initial to get t (time)
5. Python: perf_counter(), measured in ms; C: clock(), measured in s; both from time library
6. Below are the times obtained:

	Times (s)			
	Python (N=4)	C and x86_64 Intel (N=4)	Python (0<N<=100)	C and x86-64 Intel (0<N<=100)
Execution 1	0.2076	0.0150	3.5901	0.1730
Execution 2	0.1972	0.0190	3.5236	0.1790
Execution 3	0.2029	0.0190	3.5319	0.1690
Average	~0.2027	~0.0177	~3.54853	~0.1737

Fig. 1 Execution times of programs with N=4 and randomized N (s)

From the figure above, it can be inferred that the algorithm coded using C and x86_64 Intel ran ~10-13x faster than Python. For 0<N<=100, the difference became much significant reaching up to ~20x faster. The significant difference in runtimes is due to C being statically natured (faster compilation, lower level) and Python being dynamic. Both requires a compiler but Python also requires an interpreter. Adding the inline assembly code further improves the runtimes of the C program by ~1.5x, and this might be due to assembly involving less instructions. I chose to write a subtractor function in assembly as my algorithm involves lots of subtractions.

But, note that the C and x86_64 Intel program was not able to process numbers with too many digits, thus this might have affected the greater time differences.

VI. Trade-offs

C with an inline x86_64 Intel assembly greatly and consistently improves runtimes. However, programming in C is a bit more tedious for me due to its syntax and also me being less familiar with the language. This is a minor issue as it involves the skills of the programmer and not really the language itself. It is apparent though that writing in Python is much more clean, short, and organized as the syntax involves less special symbols. Assembly programming is another story as coding it really is much difficult and complex. For me, coding and learning C is worth it as the program speed up is very significant, and the code itself can be organized with proper indentations, spacings, and groupings. However, assembly language is complex and I think that it really is true that assembly coding is the final resort with regards to program optimization.