Gabriel Angelo F. Limlingan
CoE 163 Computing Architectures and Algorithms
April 8, 2024

# Software Exercise 01: Profiling and Assembly

**Synopsis**: Formulate and profile two different algorithms to compute the minimum number of steps needed to unlock some combination locks.

---

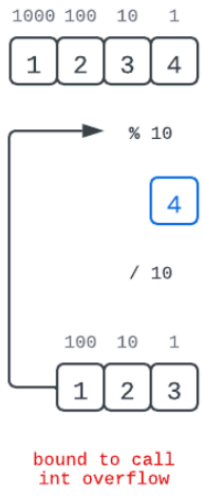
## Introduction

A lot of programming languages today can be used in problem-solution fits. Such languages are so flexible that they can solve the exact same exercises albeit in slightly different formatting and semantics. So, why not just stick to one standard like C/C++? As good as it may seem, when it comes to building more complex algorithms, modern languages are built in a higher level of abstraction to conform to use cases in more specific contexts.

For this software exercise, I implemented a simple algorithm that determines the minimum number of steps necessary to "brute force" the combinations of locks with varying sizes. The hard part of this exercise is that I have to develop the same algorithm in a mix of **C/C++** and **x86_64 ASM**, apart from **JavaScript**, my preferred programming language (with Node.js installed). I will discuss more on the comparison of the programs after going through the algorithm and runtime profiling.

## The `findMinCombi` Function

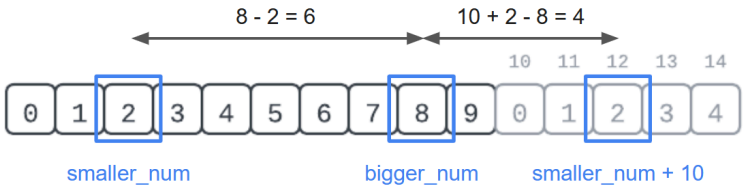
The function takes in the inputs of the program: `number_of_locks`, `init_combi`, `unlock_combi`. The first variable is taken as an `int`, while the latter two variables differ in data types depending on the algorithm.




The first program written in JavaScript [automatically] declares `init_combi` and `unlock_combi` as a `number`. What's good about JavaScript is that it automatically sets its datatype through the `let` function. Then, modulo 10 was done to select the "ones" value in the number for determining the minimum steps per digit. Then, the algorithm divides the value by 10 and removes the decimal place to move on to the next digit. The algorithm effectively works in JavaScript when dealing with combinations with many digits. However, this algorithm does not work when implemented in C++ since integers could only hold a maximum of 8 bytes or 64 bits (2^64 = 1.8447e+19, only until 20 digits!). Hence, it is bound to not handle values more than that.


One way to address this problem is to convert the numbers first to a `string` (which is also a `char` array). Selecting the digit is as easy as choosing the element from the string. This solves the issue of having lock combinations with a large number of locks.

Moving on, after we have selected the digits from init_combi and unlock_combi, the `findMinCombi` function determines which of the following numbers are smaller or bigger and assigns variables to it. It then compares the steps that it would take from one direction and to the other direction as shown in the picture below. It then determines the minimum value and adds it to `min_steps`. I came up with this algorithm so that I won't have to deal with negative numbers as well as wrapping the 0–9 digits as long as I know which number is smaller or bigger.




As for the x86_64 ASM implementation in C++, it follows a more bare-metal approach of the algorithm. While more lines are added to the code, it offers faster runtime because of the functionality that the x86_64 ASM program has that the C++ does not.

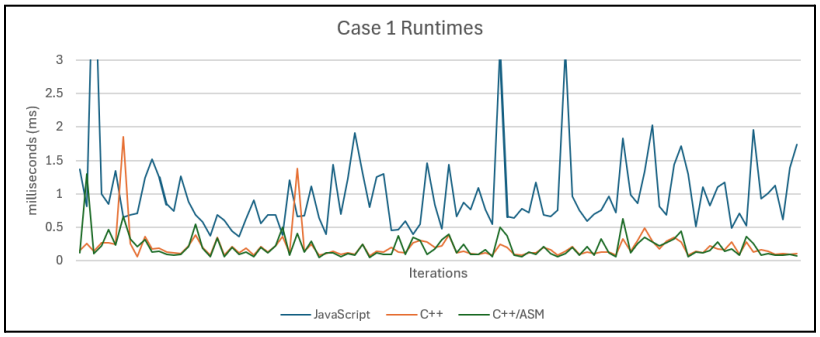
```
int findMinCombiASM(int number_of_locks, std::string init_combi, std::string unlock_combi){
    int bigger_num, smaller_num;
    int min_steps = 0;
    while(number_of_locks > 0){
        bigger_num = int(init_combi[number_of_locks - 1] - '0');
        smaller_num = int(unlock_combi[number_of_locks - 1] - '0');
        __asm__ (
            "cmp %3, %2\n"  // determine which is bigger/smaller, swap values if necessary
            "jge skip_swap\n"
            "xchg %3, %2\n"
            "skip_swap:\n"
            "mov %2, %%eax\n"  // eax = bigger_num
            "mov %3, %%ebx\n"  // ebx = smaller_num
            "sub %%ebx, %%eax\n"  // eax = bigger_num - smaller_num (steps to unlock in one direction)
            "add $10, %%ebx\n"
            "sub %2, %%ebx\n"  // ebx = smaller_num + 10 - bigger_num (steps to unlock in the other direction)
            "cmp %%ebx, %%eax\n"
            "jle skip_swap_again\n"
            "xchg %%ebx, %%eax\n"
            "skip_swap_again:\n"
            "add %%eax, %0\n"  // steps with smaller value is added to min_steps
            "dec %1\n"  // decrement number_of_locks
            : "+r" (min_steps), "+r" (number_of_locks), "+r" (bigger_num), "+r" (smaller_num)
            :
            : "eax", "ebx"
        );
```

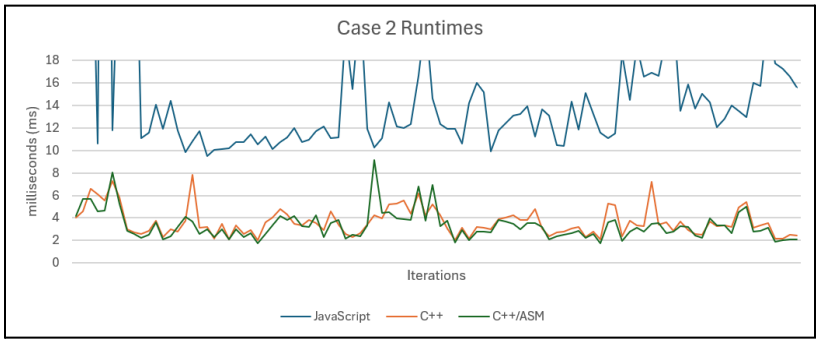
## Runtime Profiling

```
CoE 163 SE01: Profiling and Assembly (JavaScript)

Input                                   4 1234 9899
Output                                  15
Runtime Case 1: Same Length             3.8622 ms
Runtime Case 2: Different Length        42.9098 ms
```

```
CoE 163 SE01: Profiling and Assembly (C++)

Input                                   4 1234 9899
Output                                  15
Output (ASM)                            15
Runtime Case 1: Same Length             0.0837 ms
Runtime Case 1: Same Length (ASM)       0.0822 ms
Runtime Case 2: Different Length        2.8465 ms
Runtime Case 2: Different Length (ASM)  2.6018 ms
```

The images above show the results of the runtime profiling in the command line interface (CLI). `Runtime Case 1` shows the runtime of the `findMinCombi` function when iterated 1000 times with the same inputs. `Runtime Case 2` is also iterated 1000 times but with randomized inputs with `number_of_locks` ranging from 1 to 100, with its respective digits also being randomized from 0 to 9. It's clear from the images that the runtime of the algorithm is *a lot faster* if implemented in C++ and x86_64 ASM rather than JavaScript. The graph below shows the comparison of both programs after running the `runtimeProfiler` function 100 times. The C++/ASM implementation still shows better performance due to it optimizing the already-optimized findMinCombi algorithm.




| Programming Language | Average ms (Case 1) | Average ms (Case 2) |
|---|---|---|
| JavaScript | 0.999469 | 15.57235 |
| C++ | 0.209628 | 3.640558 |
| C++/ASM | 0.205595 | 3.326803 |

In determining the runtimes, I used `performance.now()` in JavaScript to determine the elapsed time in both cases of the runtime profiler. With C++/ASM, I used the high-resolution clock from the `<chrono>` library to have a more accurate calculation of the runtimes.

For both cases, the JavaScript implementation of the program is slower than the implementation in C++. While both programs follow the same algorithm for lock configurations, the innate architecture of both programming languages are a significant factor in the difference of runtimes for the following reasons:

1. C++ has a compiler that converts the C++ code to machine-readable code. And JavaScript instead has an interpreter that directly executes the code to show the desired results. While skipping the compilation to machine-readable code may seem nice, JavaScript's interpreter still presents a higher level of abstraction in the program which leads to less room for optimization.

2. JavaScript has a dynamic type system in declaring variables as compared to the static type system of C++. This would consequently make the JavaScript program slower because the interpreter is still left to decide on the data type that is to be implemented in every variable.

3. Lastly, C++ is the only one out of the two programming languages that has support for inline assembly functionalities. For example, the `xchg` instruction simplifies the swapping of variables to just one line. Furthermore, the inline assembly code allows the program to use less functions and registers as shown when it only used the `eax` and `ebx` registers. While it is not implicitly stated, I think that the assembly code also "short-circuits" the conditional statements which leads to faster runtime.

## Conclusion

The comparison between JavaScript, C++, and x86_64 Assembly implementations for solving a lock combination problem reveals notable distinctions in performance and approach. JavaScript, being a higher-level language with dynamic typing, more readable code, and interpreter-based execution, exhibits slower runtime performance compared to the compiled nature and static typing of C++. Moreover, including x86_64 Assembly in the C++ implementation enhances performance by employing low-level instructions and leveraging hardware features directly, resulting in even faster execution. Ultimately, while JavaScript offers convenience and flexibility, C++ and x86_64 ASM demonstrate better performance characteristics from their capabilities of lower-level control by the user, making them preferable choices for performance-critical applications and algorithmic optimizations. The table below summarizes the comparison between the three programming languages.

| | Runtime | Programming Effort | Code Organization / Readability |
|---|---|---|---|
| JavaScript | Slow | Easy | Best |
| C++ | Fast | Intermediate | Good |
| C++/ASM | Fastest | Difficult | Bad |