# CG1112

Tutorial 1

# 1. Comparing a Laptop/Desktop vs Rpi

You have now setup and used a Raspberry Pi 3B+. Build a table and contrast the RPi with a typical laptop computer in the following areas:

- Power Requirements (Voltage, Current for typical usage).
- Hardware Specification (CPU clock speed, cores, and runtime memory).
- Storage (SD Card vs. HDD vs SSD - read/write speed)
- Interfacing capabilities (to other devices, components, networking etc).
- Software environment

Based on the above, suggest 1-2 scenarios where Pi is more suitable than a laptop or desktop.

# 1. Suggested Answer

| | Example Laptop (Macbook Pro 2015) | Raspberry Pi 3 Model B+ |
|---|---|---|
| a | 16v, ~3A | 5v, ~1A (bare board) |
| b | 2.6 GHz 6-core Intel Core i7, 16GB DDR4 | 1.4 GHz x 4 (ARM processor quad core), 1GB SDRAM |
| c | HDD ~10 MB/s or SSD ~500 MB/s | SD Card Speed ~ 20 MB/s |
| d | USB Ports, HDMI port, Wifi, Thunderbolt | 40 I/O pins, USB ports, HDMI port, Ethernet Port, Wi-Fi. |
| e | Full-fledged OS (Mac OS X). | Fully-fledged operating system like a laptop/desktop. Raspbian (Debian Environment). |

# 2. Algorithm Choice Matters

Let’s consider the sum of the first 100 integers.

$$1 + 2 + \dots . + 98 + 99 + 100 = 5050$$

Describe or write out the simplest and most obvious algorithm to calculate this sum for the first N integers. What is the time complexity of your method? You can assume addition takes constant time O(1). If we double the problem size, how much more time does it take?

# 2. Suggested Answer

```
counter= 2
total = 1
while counter <= N
    total to counter+total
    counter = counter + 1
```

It takes 2(N-1) O(1) additions, and the rules of O() mean it is O(N), not O(2N-2). Complexity is therefore O(N). Linear time. If you double N, it will take about twice as long.

# 2. Algorithm Choice Matters

As a high school student, the mathematician Carl Fredrich Gauss, impressed his teacher by finding the sum of the integers from 1 to 100 very quickly. He used a different algorithm:

Gauss realised he had fifty pairs of numbers when he added the first and last number in the series, the second and second-to-last number in the series, and so on, and that each of these had the same size. For example (1+100), (2+99), and so on. All of these total to 101. Therefore, we know the sum is 101*50 or 5050. You can assume that multiplication takes O(1) time.

# 2. Suggested Answer

What is the time complexity of this method? If we double the problem size, how much more time does it take?

Only need to do addition once, calculate N/2 once, and do multiplication once, so it is O(1). Constant time. Double problem size it still takes the same time.

# 3. Complexity

a) WorkA( 54321 * N );

-> unitWork() is independent of N. Its always fixed at 567.

-> Constant Time

**-> O(1)**

```
void workA(int N)
{
    int i;

    for (i = 0; i < 567; i++){
        unitWork();
    }
}
```

# 3. Complexity

b) WorkB( 73 * N );

-> unitWork() is dependent on N in a single-loop.

-> O(73 * N)

**-> O(N)**

```
void workB(int N)
{
    int i;

    for (i = 0; i < N; i++){
        unitWork();
    }
}
```

# 3. Complexity

b) WorkC( 5 * N );

-> unitWork() is dependent on N in a nested loop.

-> $O((5 * N)^2) = O(25 * N^2)$

**-> $O(N^2)$**

```
void workC(int N)
{
    int i, j;

    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            unitWork();
        }
    }
}
```

# 3. Complexity

## b) WorkE( N );

- [WorkE(N)] → [WorkE(N/2)] → [WorkE(N/4)] → …. → [WorkE( 0)]
- The call is a 1D list with total floor($\log_2$(N)) + 2.
- Each call has 1 unit of work.
- Total complexity = O( floor($\log_2$(N)) + 2) → **O($\log_2$(N))**

Example: N = 8 -> floor($\log_2$(8)) + 2 = 3 + 2 = 5

$$8 \longrightarrow \underbrace{4 \longrightarrow 2 \longrightarrow 1}_{3} \longrightarrow 0$$

$$1 + 3 + 1 = 5$$

```
void workE(int N)
{
    if (N == 0){
        unitWork();
        return;
    }

    workE( N / 2 );
    unitWork();
}
```

# 3. Complexity

## b) WorkD( N );

- [WorkD(N)]
- [WorkD(N/2)] [WorkD(N/2)]
- [WorkD(N/4)] [WorkD(N/4)] [WorkD(N/4)] [WorkD(N/4)]
- …..
- [WorkD(0)] ………. [WorkD(0)]

```
void workD(int N)
{
    int i;

    if (N == 0){
        unitWork();
        return;
    }

    workD( N / 2 );
    workD( N / 2 );
    for (i = 0; i < N; i++){
        unitWork();
    }
}
```

# 3. Complexity

$N = 16$, Depth $= \log_2(16) + 2 = 4 + 2 = 6$

16 1

8 8

4 4 4 4

2 2 2 2 2 2 2 2 4

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

# 3. Complexity

- The call is a binary tree with height [**floor(Log$_2$(N)) + 2**]
- In this case, it is easier to note that each **level has the same amount of work in total.** e.g. Each of the WorkE(N/2) do N/2 work, so that level sum up to N/2 + N/2 = **N**, similarly for the WorkD(N/4) level, where each of the 4 calls do N/4 work ➔ total **N.**
- So, total complexity = O( N * height) = **O( N * log$_2$N )**

# 4. Time and Space Complexity

Given a character strings of N characters, tally the frequency of occurrences for every characters and print out the answer.

Example: "ab!da!" (N = 6 characters)
Output:
a = 2 times
b = 1 time
! = 2 times
d = 1 time
a = 2 times //note the result is printed for every characters in the
! = 2 times // input string, regardless of duplication.

Suggest **two algorithms** with the following restrictions:
a. Does not store any prior tally, i.e. recalculate the frequency for every characters
b. Use additional memory space to store the prior tally somehow.

# 4. Algorithm A

Approach A – Pseudo Code

```
For I = 0 to N-1
    Frequency = 0
    Current = String[I]
    For J = 0 to N-1
        if (Current is the same as String[J])
            Frequency ++
    Print result with String[I] and Frequency
```

Time complexity = $O(N^2)$

Space complexity = $O(1)$ (only I, J, Frequency and Current, independent of N)

# 4. Algorithm B

Approach B – Pseudo Code

```
Array Frequency[256], initialized to all zeroes

For I = 0 to N-1
    Frequency[ String[I] ]++  //Use Ascii as index

For I = 0 to N-1
    Print Result with String[I] and Frequency[ String[I]]
```

Time complexity = O(N)

Space complexity = O(1)   (only I, J, Frequency[256] and Current, independent of N)

# 4. Conclusion

- In this case, Approach B is the obvious winner.
- In general, time and space are two resources that are commonly in tradeoff relationship, i.e. we can spend more memory space in order to reduce the time spent or vice versa.
- For example, there are many cases where we can do pre-processing on the data and store the information to help with future calculation.

# 5. Git

- [Git] Consider the following scenario, suggest how to achieve the desired outcome by utilizing Git.
  - You are the working on a **solo C coding project**.
  - There is one **function X** in the project that has two **possible implementations A and B (e.g. different algorithms, different data structure etc).**
  - You want to **try both of the approaches separately**.

- Focus only on functionalities learned in the studio. Discuss the problems with this approach.

# 5. Suggested Answers

- One possible way is to:
  - Commit the original code without function X (say version 1.0).
  - Implement the first approach A and commit as version 2.0a.
  - **Checkout version 1.0**, then implement the second approach B and commit as version 2.0b.
  - **Check out version 2.0a or 2.0b as needed.**
- This is workable but very cumbersome especially when the alternative approaches are much bigger than a single function (e.g. consider the case where you need multiple commits for each version). Git support the **branching** function, where you can split off and maintain two separate lines of work. This is not covered in the studio / course, but you are encouraged to explore on your own.

# The End!

Q & A