Week 2 Studio 1 – Organizing Information [Graded]

This lab report is *graded*:

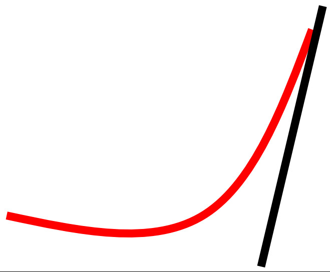
- **There are questions scattered in this handout that you are supposed to answer in your lab report. Each question will be clearly labelled.**
- **Submit a softcopy "W2S1_*Student_ID*.docx** or **.pdf" into your studio group submission folder ("Luminus Files" → "Student Submission" → "Week 2 – Studio 1" → "Studio Group"). The workbin will close <u>15 minutes after the studio ending time.</u>**

Core Objectives:

C1. Preliminary Understanding

C2. Applying Big-Oh notation

C3. Analyse common processing algorithms



Preparation (Before the studio):

- Go through the short lecture on **Code Complexity**.
- Make sure you have a ***Sunfire*** account:
    a. https://mysoc.nus.edu.sg/~myacct to (re)enable the access
    b. Verify by SSH into your account
- If you are new to ***Sunfire***, explore how to **edit-compile-run** a C program. [Note: ***Sunfire*** uses **Solaris OS**, which is a Unix based Operating System.]
- Unzip the `W2S1_Codes.zip` into a suitable folder on ***sunfire***.

Studio Setup:

- 2-3 students per sub group.
- Steps should be performed by **all sub group members individually**. You can then collate / compare / discuss the results you have.
- Share time measurement data among the sub group members, but **not explanation, example, deduction for the lab report question.**

Important Note

For consistency and comparable results, we will use **only Sunfire account** for this studio.

All C code edit + compilation + execution should be done on Sunfire.

Setup

1. Transfer and unzip the "**W2S1_Codes.zip**" at a location of your choice on the ***Sunfire*** account account if you have not done so.

2. Compile the file "**sanity.c**" and execute it for sanity check. The time measured for each round of the workload should be around 0.05 seconds. Alert the instructor if your workload is not within range of [0.04 to 0.06] seconds.

C1. Preliminary Understanding

1. Open the file "**testHarness.c**" and focus only on the main() for the time being.

```
//Read the "data size" as N
printf("Data Size = ");
scanf("%d", &N);

printf("\n**Work Starts.... ");
start = clock();

//Put actual workload here to measure time taken
                    Harness

end = clock();
printf("Done\n");
```

The two highlighted **clock()** allow us to measure the time elapsed of any instructions / functions between them. For ease of reference, we will call this location the **Harness**.

As a trial, enter the following code fragment into the **Harness**:

```
for (i = 0; i < N; i++){
      dummy = dummy + 1000;
}
```

Remember to declare **i** and **dummy** as integers.

Note that the value of **N** can be understood as a simple way for us to control the execution time. We will see more usage later.

Lab Report Question 1:

a. Find and report a **N** value that takes **about** 2 seconds to run.

b. Change "dummy" to a **double** value, report the execution time using the same **N** as in (a).

C. Observe the different timing generated by (b). Give an educated guess for this change in execution timing.

2. To facilitate experimentations, we have defined a number "workload functions" in the program. They have the form "`void workX(int N)`". Each of the function captures typical algorithm structures. Take a look at **workA()** and **workB()**:

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

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

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

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

The `unitWork()` function represents a single unit of work that takes **T** time unit to run. So, we can say that `workA()` function has about 567 x **T** time units of work, while `workB()` represents **N** x **T** time units of work.

In general, **N** represents the data size (problem size). Most `workX()` functions will scale in some fashion in response to the data size **N**.

3. Test `workA()` and `workB()` **separately** with N = 1 and note the execution time. (i.e. place `workA(N)` into the **Harness**, compile and test; then perform the same steps for `workB(N)`). To facilitate multiple testing, you may want to name the executable differently, e.g. "gcc testHarness.c –o A.exe" for workA(), etc. You may also want to run 2-3 times and get the average running time.

> **Lab Report Question 2**:
>
> a. Report the execution time for `WorkA()` and `WorkB()` when `N = 1.`
>
> b. Suppose `WorkA()` and `WorkB()` represents ***two different solutions*** to the ***same problem***. Based on the measurement result above, is it correct to say "We should ***always*** use workB() to solve this problem"? **Convince us** (with experiment data!)

4. Now, consider another scenario, where we have the following two contesting algorithms to solve the same problem:

Approach A	Approach B
//Place the following into **Harness** **`workB( 25 * N );`**	//Place the following into **Harness** **`workB( N );`** **`workC( N );`**

> **Lab Report Question 3**:
>
> a. Express the amount of work done by the two approaches as a formula with respect to the problem size **N.** Note that `unitWork()` takes **T** time units.
>
> b. Use a table to record the time taken by the two approaches for **N = 1 to 10**.
>
> c. Based on (b), which is the more efficient approach and why?
>
> - You can use the given "**`run.sh`**" shell script to help.
> Need to perform a "**`chmod 700 run.sh`**" before 1st use.
> Usage: "**`run.sh <executable> <Start N> <End N> [Interval]".`**
> Example: **`run.sh a.out 1 20`** will run **`a.out`** for **`N = 1`**, 2, …, 19, 20.
> Example: **`run.sh a.out 1 20 2`** will run **`a.out`** for **`N = 1`**, 3, 5, …, 19.
>
> - **Uncomment the more succinct output message in the `testHardness.c` to make your life easier. ☺**

Learning Point:

By now, you have all the necessary ingredient to explain the following elements of **Asymptotic Complexity Analysis** (see pre-studio material, slide 9):

a. Analyse problem of large size.
b. Consider only leading term.
c. Ignore coefficient of leading term.

Discuss with your group members. Use the experiments performed as a basis to understand why Asymptotic Complexity Analysis has such characteristics.

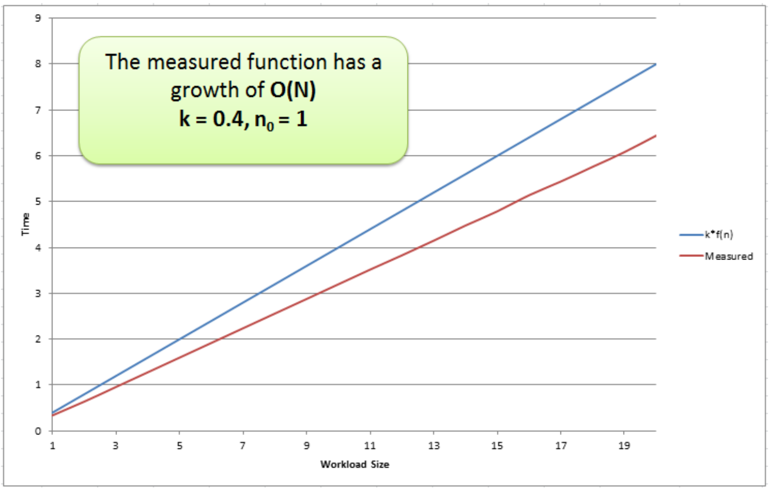
C2. Applying Big-Oh notation

> Algorithm *A* is of **O(*f*(*n*))**
> **if** there exist a **constant *k***, and a **positive integer n_0**
> **such that** Algorithm *A* requires
> **no more than *k* * *f*(*n*) time units** to
> solve a **problem of size *n* >= n_0**

In this section, we will determine the growth function (f(n)) for a few workload **experimentally**. Your task is to measure the time taken for a workload function for various input size, then manually try to fit the correct growth function to determine the **upper bound time complexity**. Note that we are interested in the **lowest** **upper bound** (just in case you listened to the ***eLecture*** and learned the "trick" ☺). Below is a worked example to illustrate the steps.

Example to measure [`workB(30 * N)`]

1. Placed the statement `workB(30 * N);` in the **harness.**
2. Record the timing for **N = 1 to 20** in the table.
   (See the "example" worksheet in **workloadExample.xlsx)**
3. Observe the timing trend and pick the most likely growth function (**f(n)**)from the "reference" worksheet. Copy the growth function values over to the appropriate column (**f(n)**) in the table.
4. Determine a suitable constant **k** and a **positive integer** N_0, such that the **k*f(n)** line **consistently higher than** your measured timing after a $N = N_0$
5. Summarize your findings in the text box on the chart. Screen capture the chart into your lab report. Below is an example of expected format.



N	O(n)	k	k*f(n)	Measured
1	1	0.4	0.4	0.34
2	2	0.4	0.8	0.64
3	3	0.4	1.2	0.96
4	4	0.4	1.6	1.28
5	5	0.4	2	1.6
6	6	0.4	2.4	1.92
7	7	0.4	2.8	2.24
8	8	0.4	3.2	2.56
9	9	0.4	3.6	2.88
10	10	0.4	4	3.2
11	11	0.4	4.4	3.52
12	12	0.4	4.8	3.83
13	13	0.4	5.2	4.15
14	14	0.4	5.6	4.48
15	15	0.4	6	4.79
16	16	0.4	6.4	5.14
17	17	0.4	6.8	5.44
18	18	0.4	7.2	5.76
19	19	0.4	7.6	6.08
20	20	0.4	8	6.44

Lab Report Question 4:

Perform the measurement and fitting procedure for the following workload. Note that the **constant values M_1, M_2, M_3 will be given to you during the studio itself.**

```
a. workC( M1 * N );
b. workD( M2 * N );
c. workE( M3 * N );
```

Notes:

- Distribute the work to members in your subgroup.
- As mentioned in Q3, You can use the given "`run.sh`" shell script and use the more succinct output format to simplify the procedure.

C3. Analyse common processing algorithms

In this section, let us apply ideas learned in **C1-C2** to implementation of real algorithms. We will focus on **searching** and related algorithms. To make the comparison easier, we present **3 different approaches** to solve the same problem.

Problem Context

Given a dictionary **D** (a collection of strings) in unsorted order, we wish to check whether a word **W** exists in **D**, i.e. search for **W** in **D**.

<u>Approach One: Just Search!</u>

Open up the file "`strSearch.c`" and take a look in the `main()` function.

```
//Read the "data size" as N
printf("Data Size = ");
scanf("%d", &N);

//Initialize the dictionary up to N words
// dSize is the actual size after initialization
1 dSize = readDictionary("words_random.txt", dictionary, N);
printf("Read %d words\n", dSize);

//Measure the time on the query operations
start = clock();

//Perform nQuery[10,000] searches
2 success = queryWordList("words_query.txt", &nQuery, dictionary, dSize);

end = clock();

//Print statistics
seconds = ((double)end - start) / CLOCKS_PER_SEC;
3 printf("##Time Elapsed = %u us [%.3f s]\n", end-start, seconds);

printf("Found %d out of %d queries in dictionary of %d words\n",
    success, nQuery, dSize);
```

Essentially:

1. The program reads up to **N** words into dictionary **D**.
2. Perform *nQuery* (hardcoded to 10,000) queries. Each query looks for a word **W** in the dictionary **D**.
3. Report the timing and the query outcome statistics.

A small note on string comparisons in c.
In the coming sections you will encounter a function called **strcmp**. Let's have a quick high-level look at what it does.

```
int strcmp ( const char * str1, const char * str2 );
```

The function **strcmp** compares two strings (str1 and str2) lexicographically. If the strings are identical, the function will return 0. If the strings are not identical, it will return a negative number if str1 comes before str2 lexicographically, or return a positive number if str2 comes before str1 lexicographically.

Some examples:
`strcmp("abc", "abc")` will return 0
`strcmp("abc", "def")` will return a negative number
`strcmp("def", "abc")` will return a positive number

It is a little more complicated than this description but it is sufficient for this studio. If you are interested you can read more here.

Now, zoom in the searching algorithm used in the `searchDictionary()` function.

Lab Report Question 5:

a. What searching algorithm is used?
b. User BigO notation to indicate the time complexity of the function.
c. Use 3-4 suitably chosen value of **N** to verify (b). Tabulate the results as a table with following columns:

N (Dictionary Size)	Time for 10,000 query	Average time per query	# of found words

Plot / draw a graph to show the growth of the time taken.

d. Why do you think we run 10,000 queries for timing measurement, instead of just 1 query?

Note:
- The largest **N** you can use is **220,000** due to memory limitation.

Approach Two: Sort and Binary Search

A common alternative to **approach one** consists of a 2-phase solution:

1. **Sort** the dictionary in ascending order.
2. Perform **binary search** for queries.

A sample implementation of this approach can be found in "`strSBS.c`" (string **s**ort + **b**inary **s**earch). Take a moment to read and revise your understanding of ***sorting*** and ***searching*** from CS1010.

Lab Report Question 6:

a. Which sorting algorithm is used?
b. Perform the similar steps as in Question 5(c) to find out the time complexity for **Sorting, Searching and the Overall approach**. Note that the code already measures the **sorting**, **searching** and **total time** separately for you.
c. Is this approach better than **approach one**? Briefly explain.

Learning Point:

You have a rare chance to actually verify complexity analysis in this studio. At the end of the studio, you should look back at the code and see whether you can determine the complexity analytically (i.e. purely on paper, without execution). If your understanding of Big-O analysis is correct, your analytical answer should match the experiment result perfectly.

Challenge Section

The questions below is provided for your own exploration only and will not be marked.

Approach Three: A different sort?

Search online (or refer to your CS1010 notes) for the outline / source code of other common sorting algorithms like **selection sort** and **bubble sort**. Discuss with your group members and make educated guess on their time complexity. It is natural to assume sorting simply cannot be done faster than the mysterious sort, selection or bubble sort.

Let us show you that is NOT the case ☺. Refer to the "`strMS.c`", there is yet another mysterious sorting function (`MSort()`).

Question 7:

Use the same procedure as in Q5, Q6 to measure and report the time complexity of the MSort() function.

Question 8:

a. With the new sorting function, do you think that the Sort + Binary Search approach is now more efficient than the **approach one**? Why?

b. How can we justify using **approach three**? You can describe a usage scenario where this approach gives the best overall performance.

~~~ End of Studio ~~~