Spike: Task 7
Title: Performance Measurement

Author: Luke Valentino, 103024456

Goals / deliverables:

- Code see /COS30031-2023-103024456/07-Spike-Performance Measurement/task07.cpp
- Spike Report (including graphs)

Technologies, Tools, and Resources used:
List of information needed by someone trying to reproduce this work

- Swinburne Lecture 3.1
- Sample code provided by Swinburne.

Tasks undertaken:
Step 1: Retrieve sample code files task07_sample1.cpp and task07_sample2.cpp from CANVAS. Also retrieve Task 07 – Spike – Performance Measurement.pdf from CANVAS.

Step 2: Open sample code files in Visual Studio Code. (Make sure the latest C++ extension is installed).

Step 3: Create your own cpp file, task07.cpp. This will be where we write our code.

Step 4: We will now complete the 6 tasks set out for us in Task 07 – Spike – Performance Measurement.pdf, starting with the first task, Single Tests. By inspecting the sample code and watching lecture 3.1, we can understand the use of std::chrono to time tasks. The goal here is to perform an action and time it by taking the time immediately before its commencement and after its completion. To make the time more readable, we can cast the time to nanoseconds using the example code in task07_sample1.cpp.

Step 5: From task07_sample1.cpp, we can use the exponential_rampup_test() and linear_rampup_test() functions to complete part of the “Ramp-up Test” of this spike report. After implementing them into our code, we must cast the time to nanoseconds. We still need to compare the ramp-up tests to the ramp-down tests. We can do this using the code we used for the ramp-up tests by flipping the operations and increasing/decreasing some of the variables.

Step 6: Run this test multiple times with small changes on test conditions. E.g. change the order in which the functions are called in main.

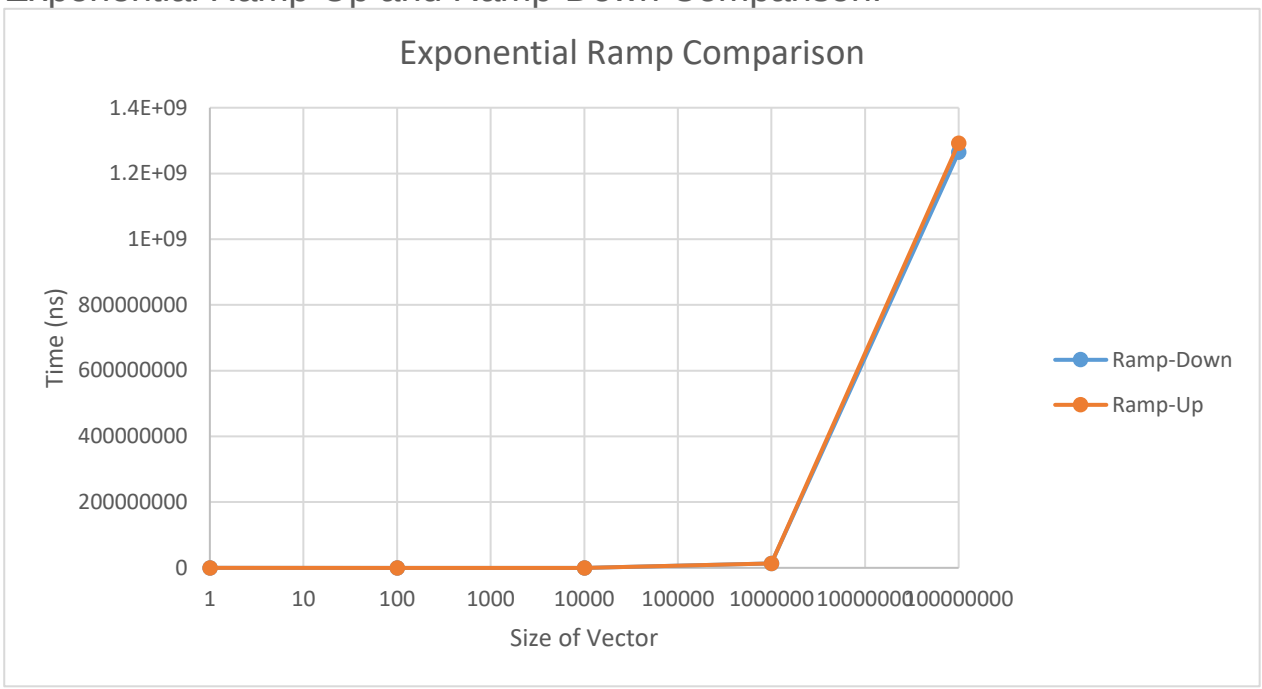
Step 7: Using code from task07_sample1.cpp, we can tackle the function comparison task. Edit the code from the sample file to be able to time these characters counting actions and create a string for them to analyze.

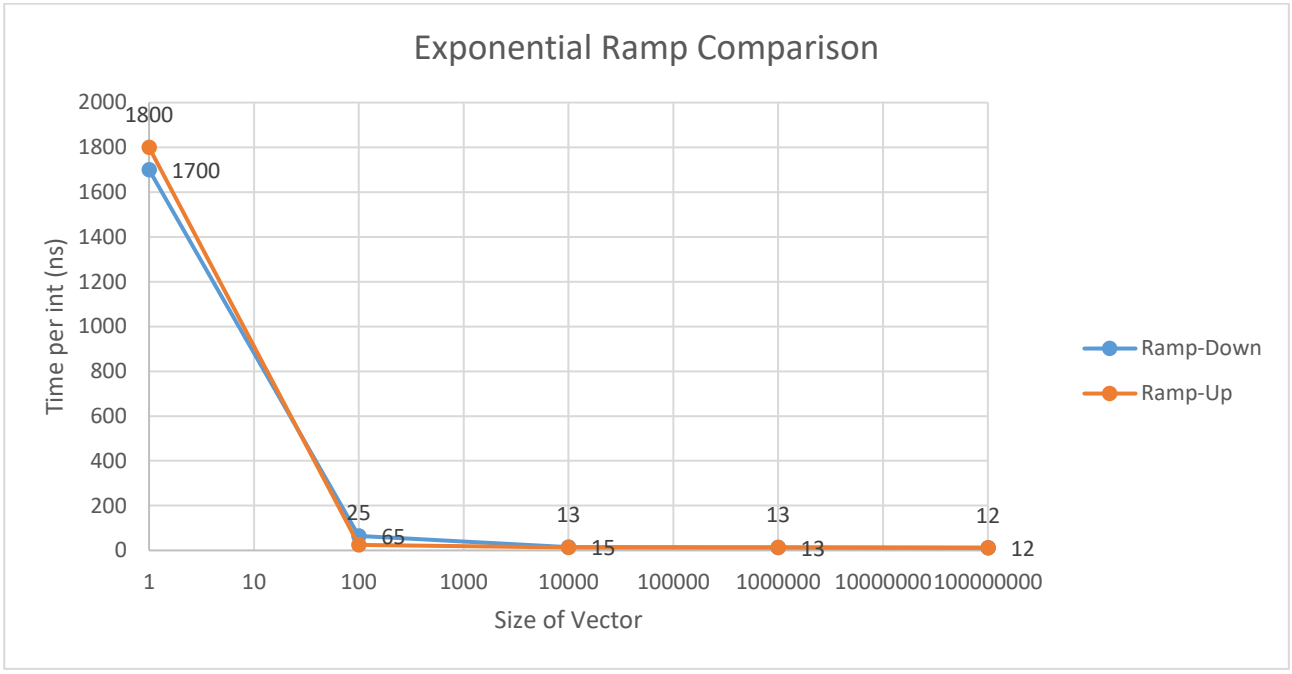
Step 8: By clicking on the run arrow in the top right of Visual Studio Code, we can change from debugging mode to release mode. Check if there are any differences in efficiency between the two modes.

Step 9: To disable compiler optimizations in Visual Studio Code, navigate to the tasks.json file in your project folder. Then add “/0d” to the args array. Save the file and run the tests.

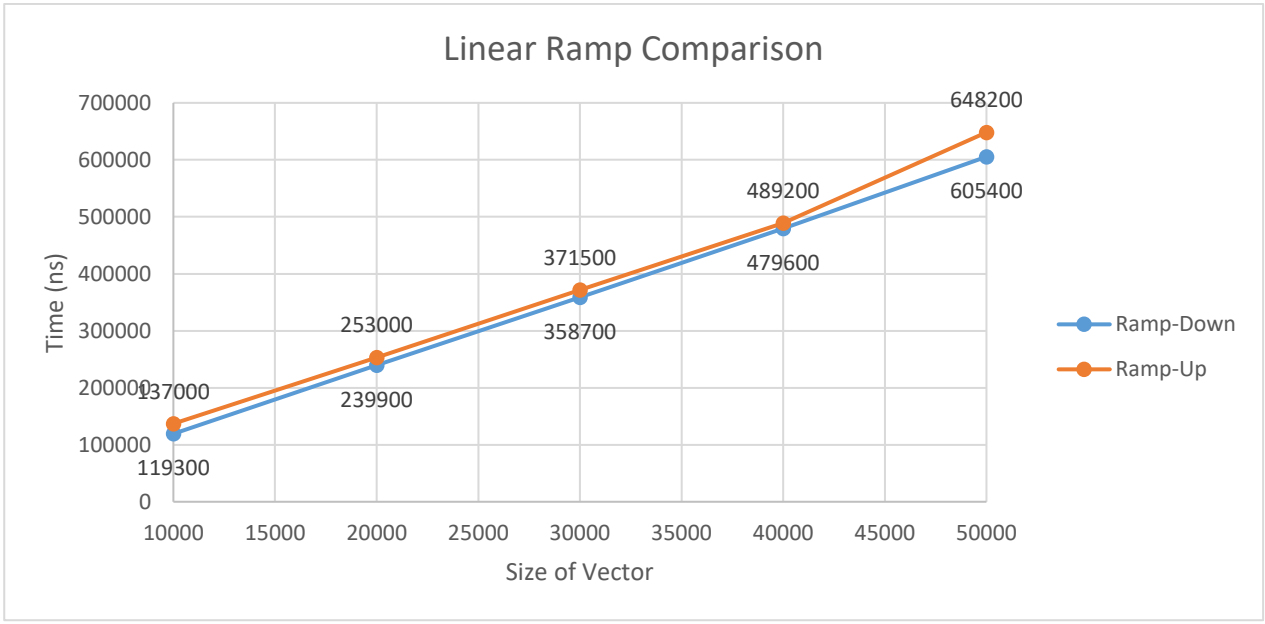
What we found out:

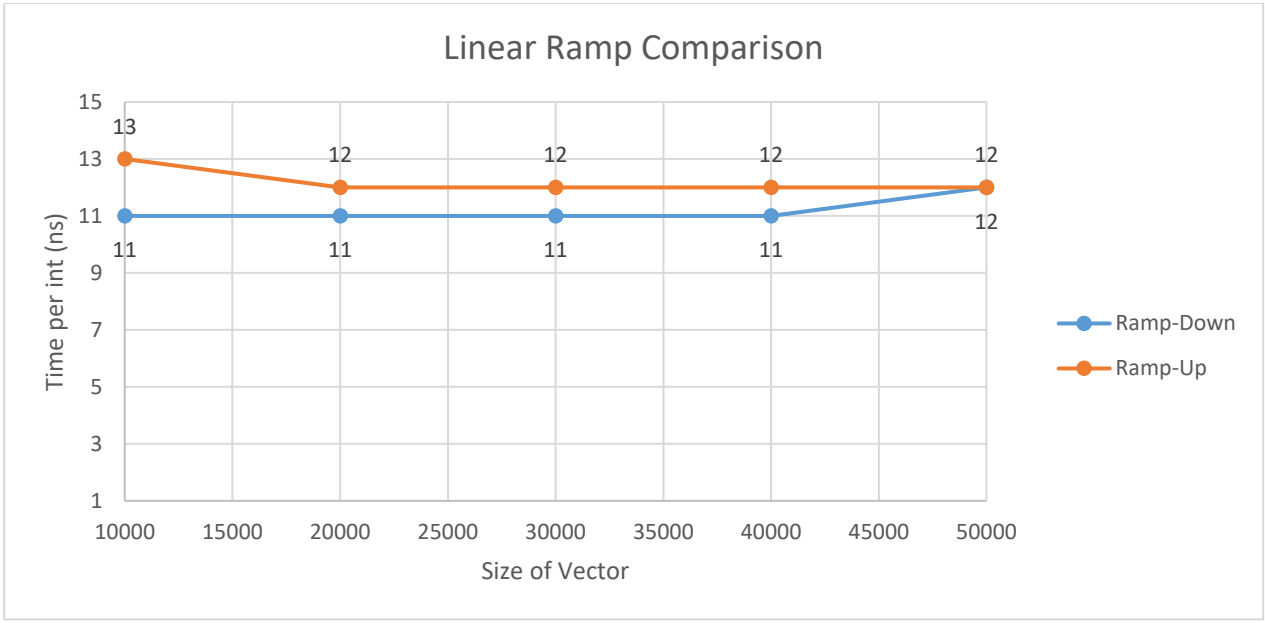
Exponential Ramp-Up and Ramp-Down Comparison:





Linear Ramp-Up and Linear Ramp-Down Comparison:





As can be seen from the graphs, the difference between ramp-up and ramp-down in both linear and exponential is minimal. This difference could very well be explained by inconsistences.

Function Comparison:

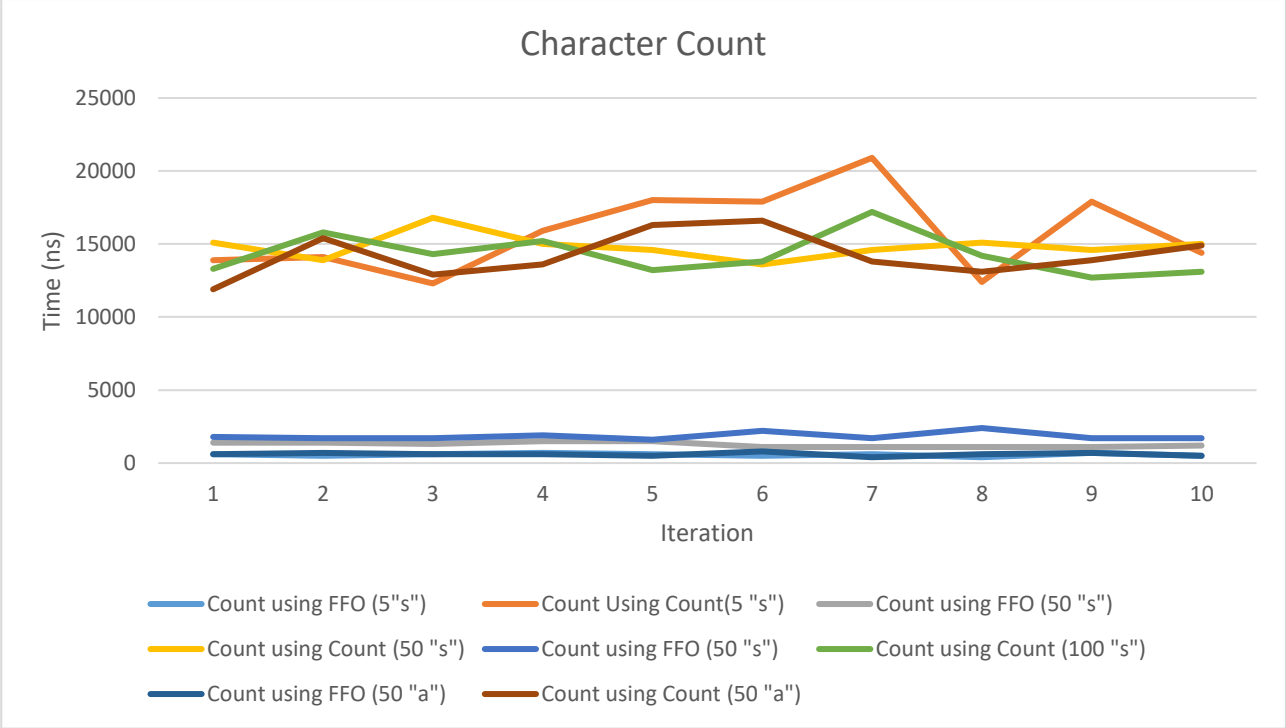
The two character counting functions provided in sample 1 vary greatly in their approach.

Iterations	Count using FFO	Count using Count
1	600	13900
2	500	14100
3	600	12300
4	700	15900
5	600	18000
6	500	17900
7	600	20900
8	400	12400
9	700	17900
10	500	14400
Average	**570**	**15770**

As can be seen from the summary table, the character count using FFO is considerably faster.

To determine if execution time differs linearly with respect to string size, I decided to test this by changing the string the functions analyse. I created three test strings. For reference, the target letter that the function is counting is “s”:

- String with 5 “s” characters
- String with 50 “s” characters
- String with 100 “s” characters
- String with 50 “a” characters



As can be seen from the graph, execution time only increases if the string contains more “s” characters. Adding non target characters (e.g. “a”) does not increase the time of “Count using FFO” but it does increase the time of “Count using Count”. When adding additional “s” characters to the string, we can see that the time does not increase linearly.

How Repeatability Varies Depending on Test Conditions:

To test how different test conditions impact runtime, I decided to vary the order in which the functions are called. The first set of data (labelled “After functions”) is executed after all of the ramp-up/down functions from the previous tests. The second set of data (labelled “Before functions”) is executed before these functions (Count using FFO is executed first).

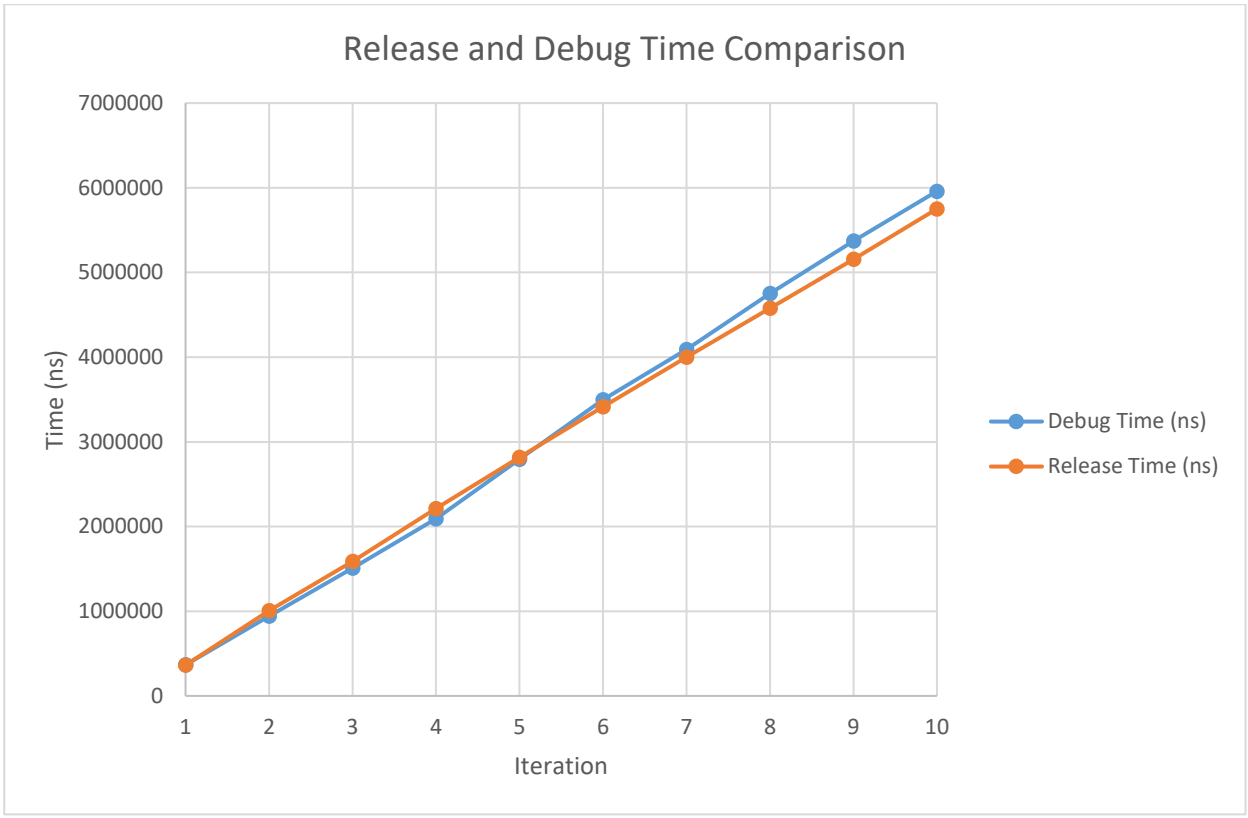
Iterations	Count using FFO (After functions)	Count using Count (After functions)	Count using FFO (Before functions)	Count using Count (Before functions)
1	600	13900	800	9300
2	500	14100	800	6300
3	600	12300	1000	8300
4	700	15900	900	6900
5	600	18000	800	6300
6	500	17900	900	6700
7	600	20900	800	6400

8	400	12400	1000	6400
9	700	17900	800	9900
10	500	14400	700	7400
Average	**570**	**15770**	**850**	**7390**

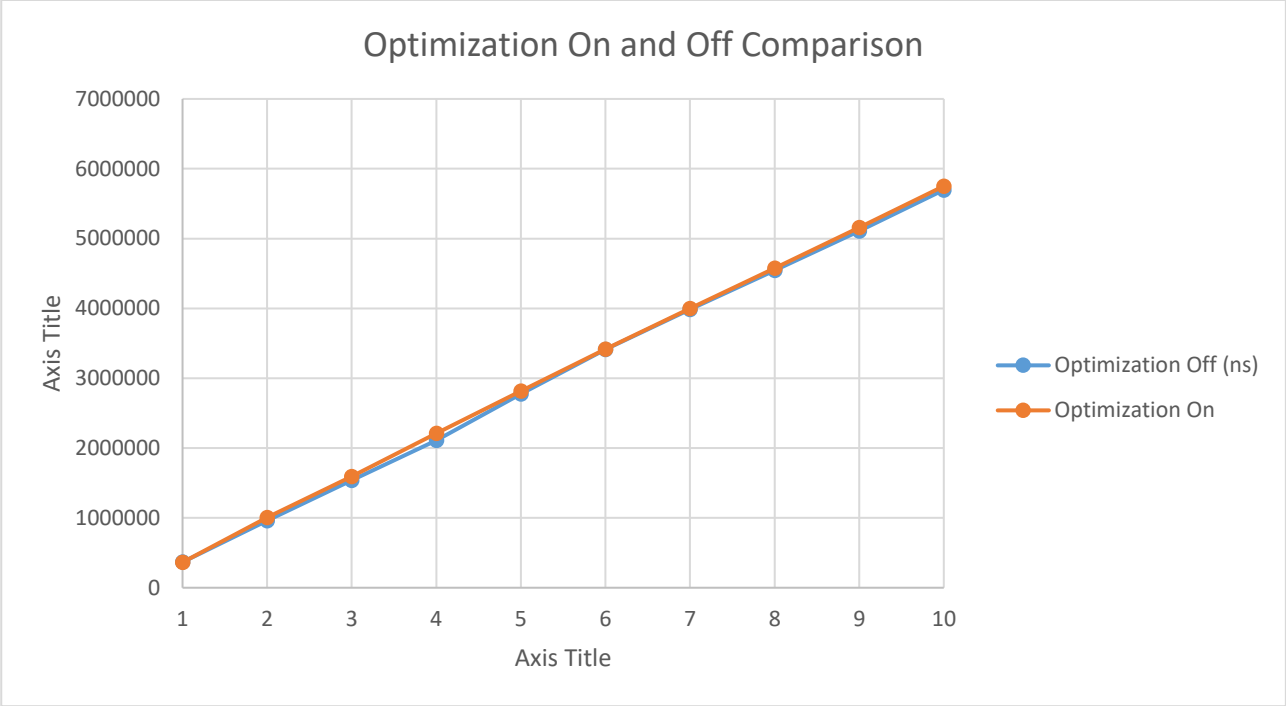
As can be seen from this data, executing the “Count using Count) before the ramping functions decreases the runtime by over 50%. However, executing “Count using FFO” before the ramping functions increases the runtime.

Release and Debug Comparison:

As can been seen from the graph below, running the program with release settings and debug settings makes no difference that cannot be attributed to normal variations.



Optimization Settings:



As can be seen from this graph, turning optimization off did not have an impact on the runtime.