Spike: Task 7.P
Title: Performance Measurement

Author: Mitchell Wright, 100595153

Goals / deliverables:

Demonstrate the ability to collect and analyse software performance data, applying said knowledge to the comparison and evaluation of functions, IDE settings and compiler settings.

Items created during task:

- Code, see: \07 - Spike – Performance Measurement\t7-performance\

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

- Visual Studio 2022
- SourceTree
- GitHub
- Lecture 3.1 – Code Performance

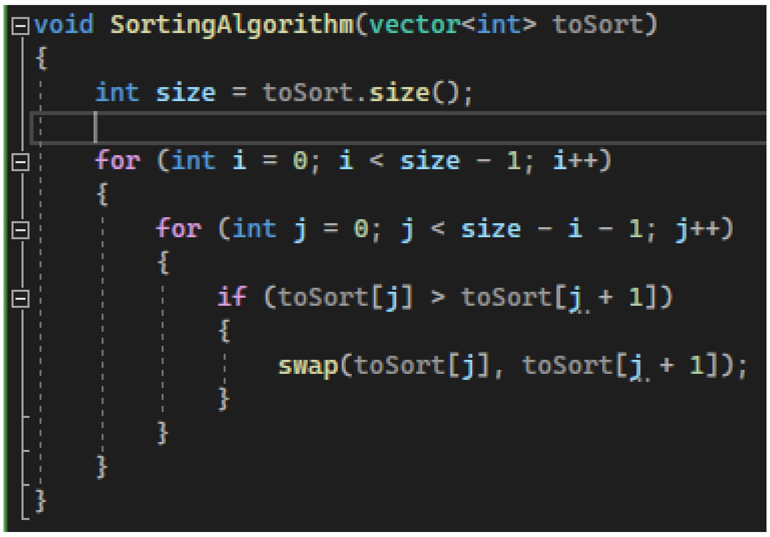
Tasks undertaken:

- Single Tests
- Ramp-up Test
- Repeatability
- Function Comparison
- IDE Settings
- Compiler Settings
- Commit to Git

What we found out:

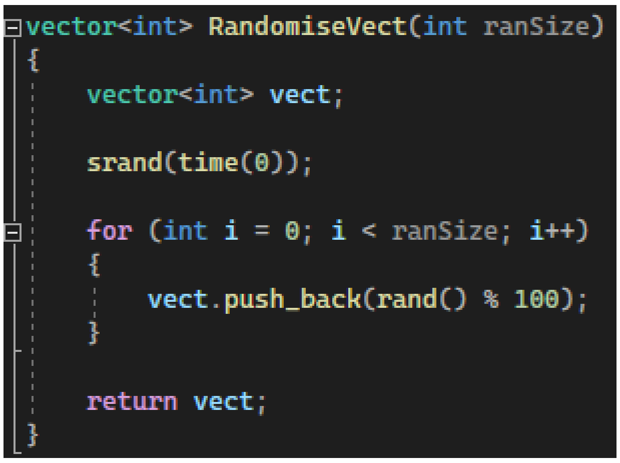
1. Single Tests:

For the tests within this section, a simple bubble sort was used on a vector containing 10 random elements. The bubble sort algorithm is shown below. Please note bubble sort was chosen due to it being simple to code, as well as leaving the vector passed through to it unsorted at the end (for repeatability's sake). Please also note this was a Chat GPT special. Since a sorting algorithm is a sorting algorithm and nothing fancy was needed here, I opted for the quick and dirty option so I could focus on the important parts of this task.



```
void SortingAlgorithm(vector<int> toSort)
{
    int size = toSort.size();

    for (int i = 0; i < size - 1; i++)
    {
        for (int j = 0; j < size - i - 1; j++)
        {
            if (toSort[j] > toSort[j + 1])
            {
                swap(toSort[j], toSort[j + 1]);
            }
        }
    }
}
```

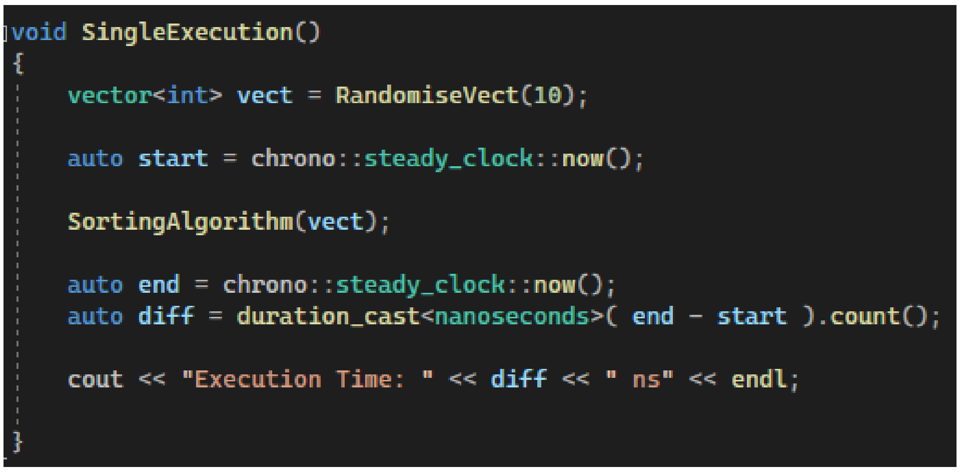
To fill the vector with random numbers, the below code is used.



```
vector<int> RandomiseVect(int ranSize)
{
    vector<int> vect;

    srand(time(0));

    for (int i = 0; i < ranSize; i++)
    {
        vect.push_back(rand() % 100);
    }

    return vect;
}
```

The random number generation has been lifted from task 2 and performs the same basic function, though now in conjunction with a vector rather than an array. Numbers generated are between 0 and 100.

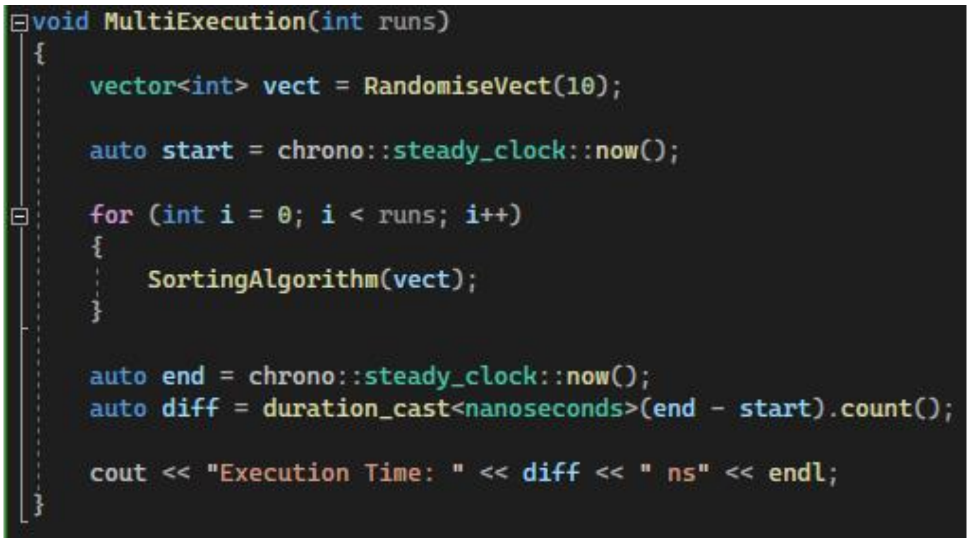
For the single execution test, a start time is taken, the sorting algorithm then runs, then the end time is taken.



```cpp
void SingleExecution()
{
    vector<int> vect = RandomiseVect(10);

    auto start = chrono::steady_clock::now();

    SortingAlgorithm(vect);

    auto end = chrono::steady_clock::now();
    auto diff = duration_cast<nanoseconds>( end - start ).count();

    cout << "Execution Time: " << diff << " ns" << endl;

}
```

Following the sample code provided in the lectures, steady_clock::now() is used to get the current time, then the execution time is calculated by casting the difference between start and end into nanosecond format. After which, the results are printed to console.

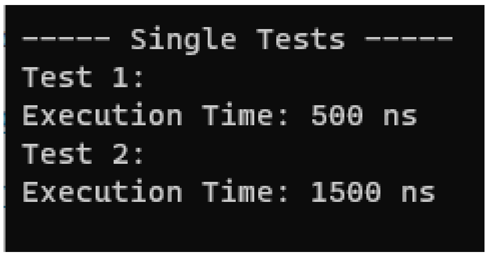
Multi-execution testing is performed essentially the same way, though with one extra for loop.



```cpp
void MultiExecution(int runs)
{
    vector<int> vect = RandomiseVect(10);

    auto start = chrono::steady_clock::now();

    for (int i = 0; i < runs; i++)
    {
        SortingAlgorithm(vect);
    }

    auto end = chrono::steady_clock::now();
    auto diff = duration_cast<nanoseconds>(end - start).count();

    cout << "Execution Time: " << diff << " ns" << endl;
}
```

The sorting algorithm is executed multiple times now, rather than just once. Note that the vector does not need to be reset in this case, as the changes to the values only occur on a copy of the vector, not the original vector.

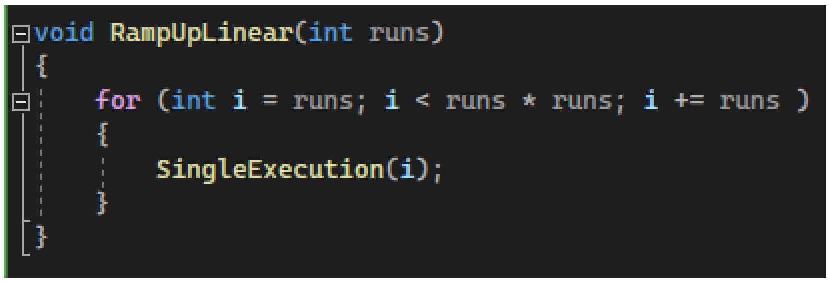
The results for both tests are shown below.



```
----- Single Tests -----
Test 1:
Execution Time: 500 ns
Test 2:
Execution Time: 1500 ns
```

2. Ramp-up Test:

The ramping tests were done using the Single Execution from before, placed inside a for loop, with the loop start and end points, as well as step size, being determined by parameters. In the case of the exponential test, this was done with a scale parameter and a power parameter (with scale^power being used to find the end point). It should also be noted that the size of the vector being sorted is what is being ramped here.



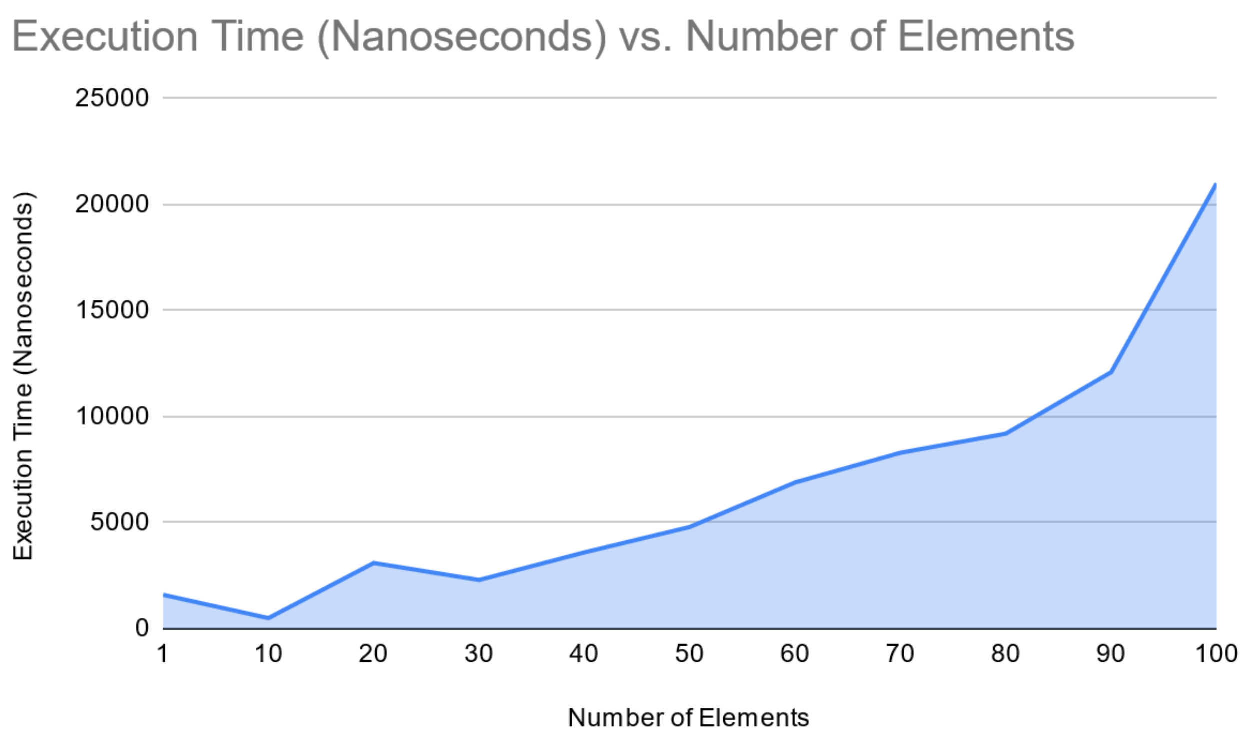
```
void RampUpLinear(int runs)
{
    for (int i = runs; i < runs * runs; i += runs )
    {
        SingleExecution(i);
    }
}
```

Based solely on numbers, the linear ramp-up test seems to show steady numbers. Please note that the results for both this and the ramp-down test start/end at 10.

Number of Elements	Execution Time (Nanoseconds)
1	1600
10	500
20	3100
30	2300
40	3600
50	4800
60	6900
70	8300
80	9200
90	12100
100	21000

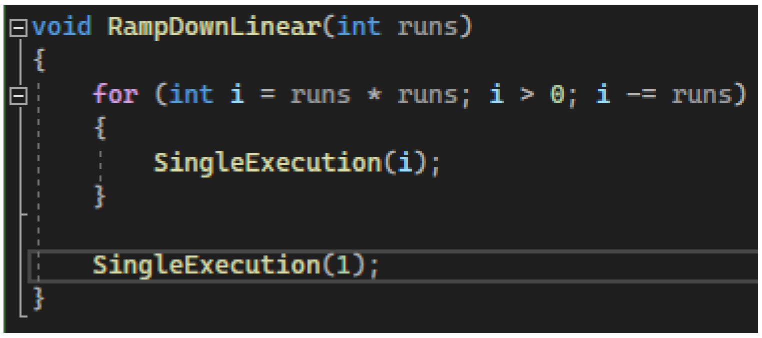
Towards the end of the chart, it becomes obvious that there is a bit of exponential growth in execution time occurring, with the numbers starting to double.

The chart for while looks like this:



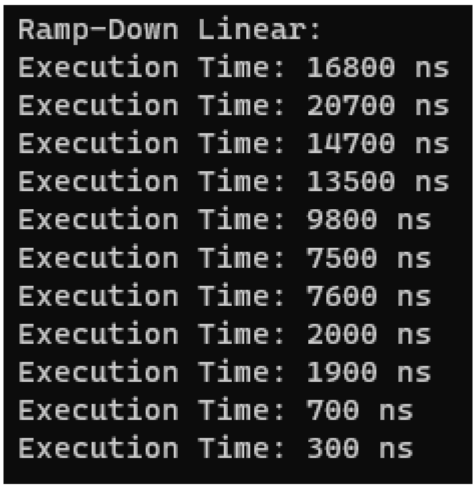
The chart highlights the exponential growth of execution time much more effectively. It should also be noted that during the 20-element test, there seems to be some sort of spike. This was consistent across runs.

The ramp-down test is essentially the ramp-up linear but in reverse, as shown below.



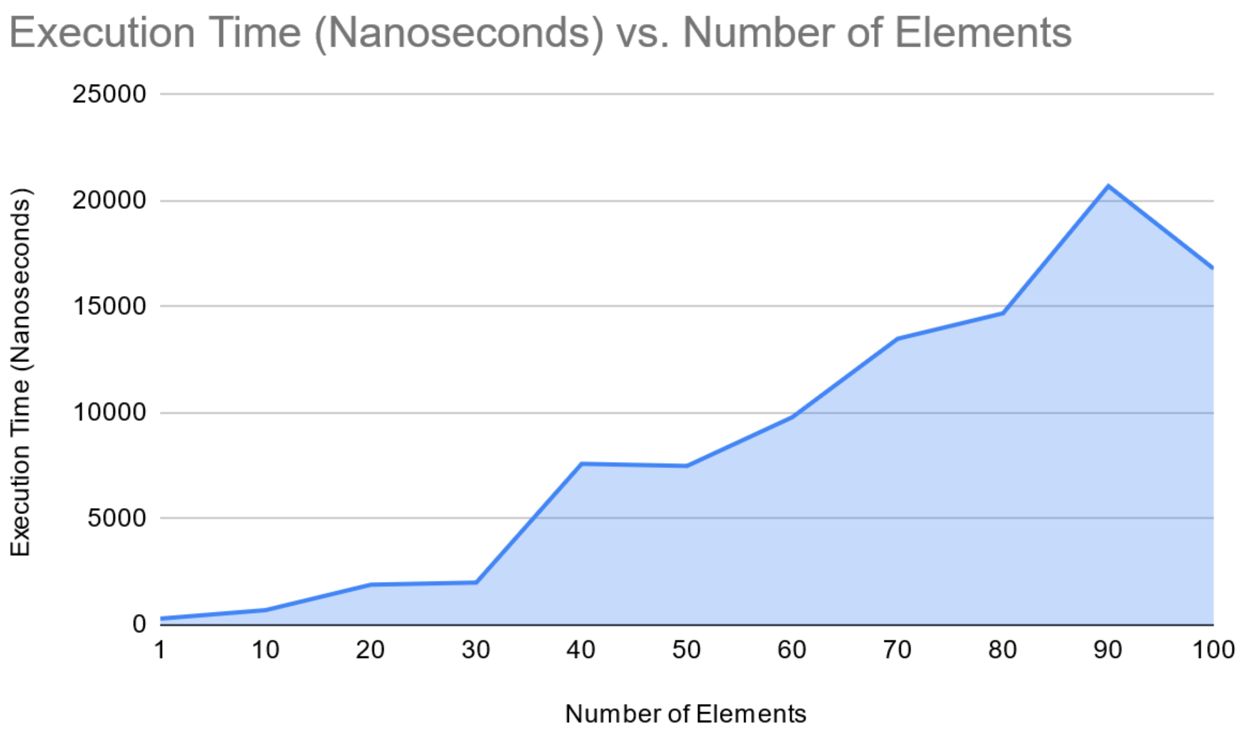
```
void RampDownLinear(int runs)
{
    for (int i = runs * runs; i > 0; i -= runs)
    {
        SingleExecution(i);
    }

    SingleExecution(1);
}
```

For which the results look like the below.



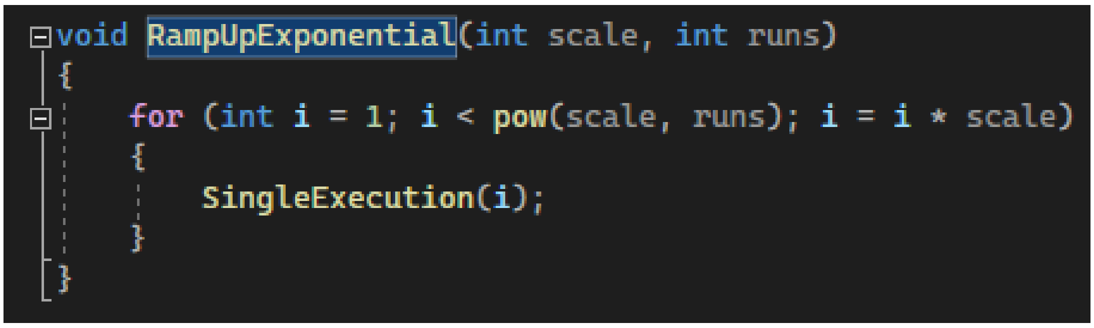
```
Ramp-Down Linear:
Execution Time: 16800 ns
Execution Time: 20700 ns
Execution Time: 14700 ns
Execution Time: 13500 ns
Execution Time: 9800 ns
Execution Time: 7500 ns
Execution Time: 7600 ns
Execution Time: 2000 ns
Execution Time: 1900 ns
Execution Time: 700 ns
Execution Time: 300 ns
```

These results start to highlight a pattern with my laptop suddenly spiking in execution time upon the second run of the test, which is very interesting.



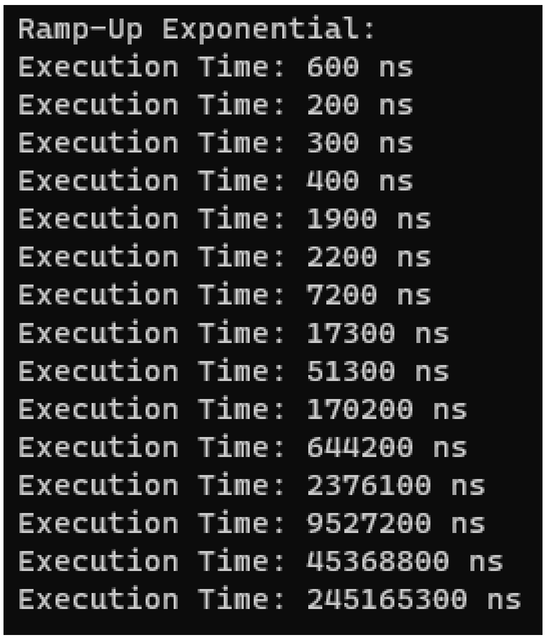
These results still do show a (skewed) exponential growth, though now with a representative value for a single element vector. Generally, it seems that ramp-up tests stabilise in terms of results when reaching the higher bounds of the testing, whereas ramp-down testing achieves the opposite. These types of testing used in conjunction can give a more accurate picture of how an algorithm reacts over a range of inputs.

In order to test an exponential growth in element size as an input, the ramp test was modified as below.



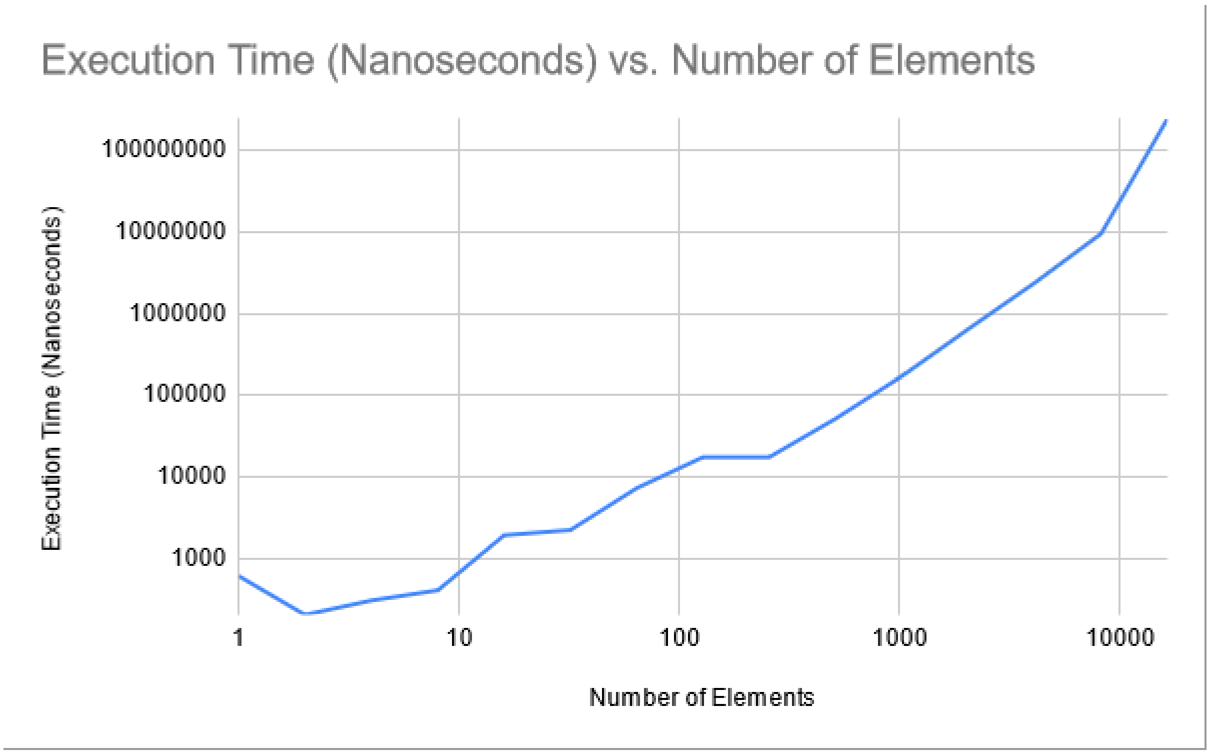
```
void RampUpExponential(int scale, int runs)
{
    for (int i = 1; i < pow(scale, runs); i = i * scale)
    {
        SingleExecution(i);
    }
}
```

This produced the below results.



```
Ramp-Up Exponential:
Execution Time: 600 ns
Execution Time: 200 ns
Execution Time: 300 ns
Execution Time: 400 ns
Execution Time: 1900 ns
Execution Time: 2200 ns
Execution Time: 7200 ns
Execution Time: 17300 ns
Execution Time: 51300 ns
Execution Time: 170200 ns
Execution Time: 644200 ns
Execution Time: 2376100 ns
Execution Time: 9527200 ns
Execution Time: 45368800 ns
Execution Time: 245165300 ns
```

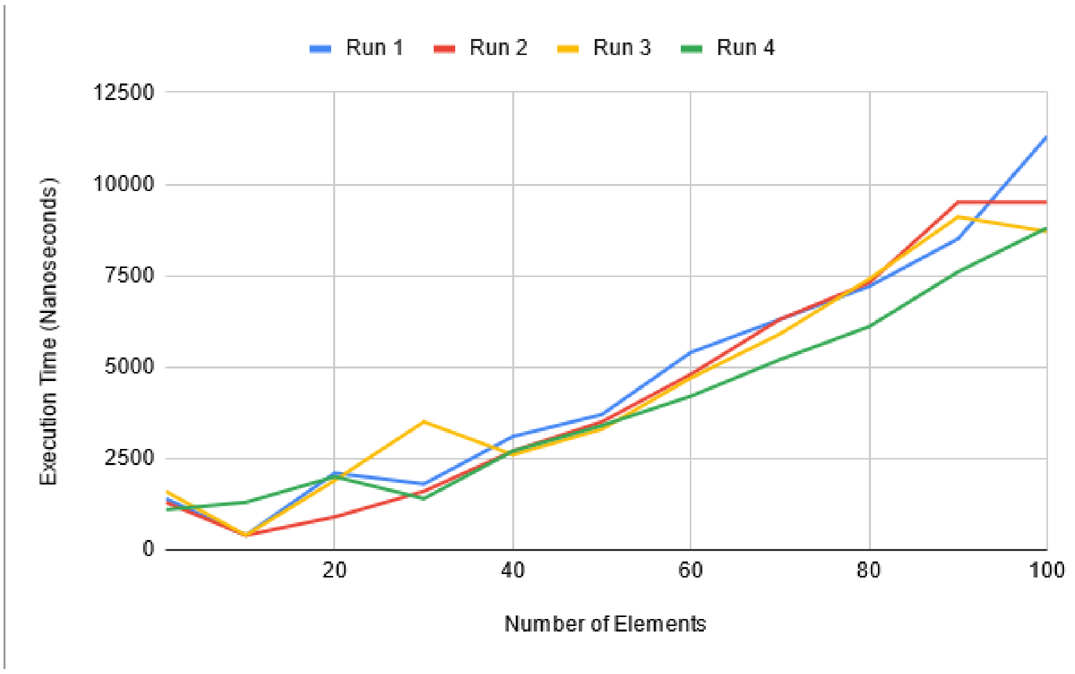
And charted, looks like the below.



While more test points will highlight this better, there does seem to be the beginnings of an exponential curve forming. Please note that graph scales up 1 to 2^15. Unfortunately measuring using powers of 10 was not realistically feasible on a laptop.

3. Repeatability:

Repeatability can be quite difficult when it comes to getting exact results. Many variables can cause differences, such as Windows updates, background tasks, other programs, OS overhead, etc. Re-using the linear ramp test from before, the results on the chart below show a reasonable amount of variance, even from such a small test. Please note that the programs were run consecutively, with no new programs opened or closed between runs.

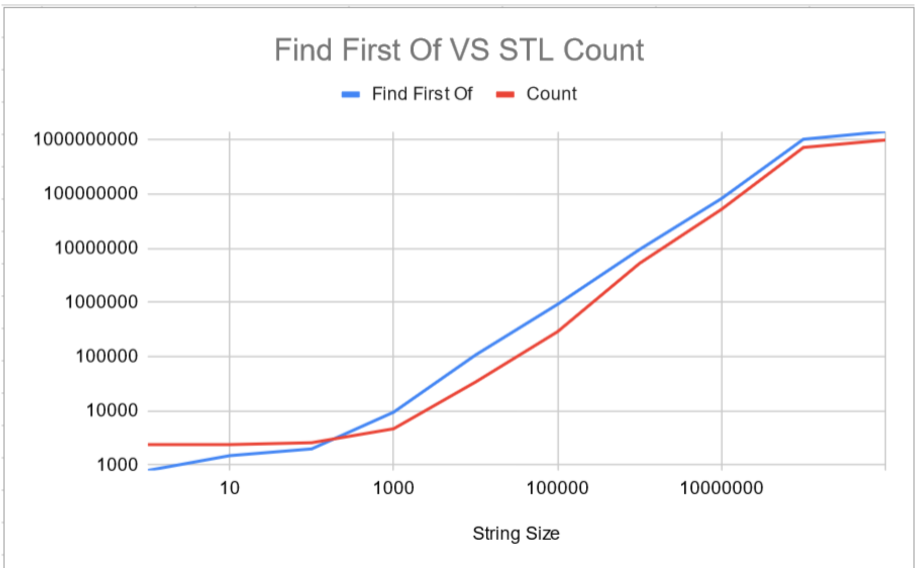


4. Function Comparison:

Comparing the 2 functions provided in the sample code, the one comment as slower seems to be performing a lot faster than the STL version. The below tests were done using a portion of the Bee Movie script.

Find First Of	Count
7700	13400
3800	5000
2900	4600

Testing this again with a ramp-up based on string length using a random string generator, these are the results.



Using an exponential ramp up test, it's notable that the Find First Of function starts off much faster, but soon loses out to the Count when getting up to the larger strings. Please note that both axes have been log scaled for visibility.

5. IDE Settings:

When testing Debug vs Release modes with the multi-execution test from section 1, the results are as below.

Debug	Release
16100	1700
17000	1300
15100	1500

Clearly Release mode, running without all the debugging tools runs a heck of a bit faster, aye.

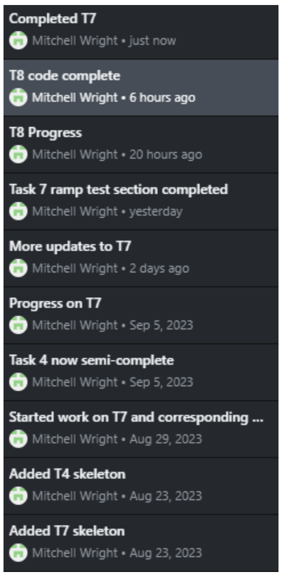
6. Compiler Settings:

In order to enable compiler optimization, follow the menus to the setting:

Project -> Properties -> Configuration Properties -> C/C++ -> Optimization.

These settings cannot be enabled during Debug mode without causing an error. However in Release mode they're enabled by default, so the results will be similar to above.

7. Commit History:



Completed T7
Mitchell Wright • just now

T8 code complete
Mitchell Wright • 6 hours ago

T8 Progress
Mitchell Wright • 20 hours ago

Task 7 ramp test section completed
Mitchell Wright • yesterday

More updates to T7
Mitchell Wright • 2 days ago

Progress on T7
Mitchell Wright • Sep 5, 2023

Task 4 now semi-complete
Mitchell Wright • Sep 5, 2023

Started work on T7 and corresponding ...
Mitchell Wright • Aug 29, 2023

Added T4 skeleton
Mitchell Wright • Aug 23, 2023

Added T7 skeleton
Mitchell Wright • Aug 23, 2023