class 1: Measuring code performance

The driving question of this whole course is: what code is best at performing a given task, and how do we know that?

This is usually not an easy question to answer. There are many different measures of what "best" can be: easiest to implement, fastest, most memory efficient, requiring the least number of disk or network accesses, least number of cache misses and so on. In fact, the answer to the question depends not solely on the program, but also on the data, the environment and many other things. We shall explore this during this class.

Most of this particular class will be revolving around measuring the performance of some code. We shall begin by measuring time with a very rough method, by using the GNU `time` utility, available on the university machines. On Microsoft Windows, I suggest trying out powershell's `Measure-Command` (read about Measure-Command here).

Let's try to look at a program which does the following: given a file that looks like this

```
number1
operand1 number2
operand2 number3
operand3 number4
operand4 number5
operand5 number6
.
.
.
```

perform the computation :

```
(((numer1 operand1 number2) operand2 number3) operand3 number4) ...
```

So, for example, a file

```
10
* 3
/ 2
+ 1
- 7
```

should print out `9` as the result.

Let us try and write a program which does the following and use `time` (or `Measure-Command`) to test the time it takes to compute the answer. For instance, it might look like this:

```
 $ > time ./one in2
9

real    0m0.001s
user    0m0.001s
sys     0m0.000s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.003s
sys     0m0.001s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.003s
sys     0m0.002s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.002s
sys     0m0.002s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.003s
sys     0m0.002s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.003s
sys     0m0.002s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.001s
sys     0m0.003s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.003s
sys     0m0.001s
 $ > time ./one in2
9

real    0m0.002s
user    0m0.001s
sys     0m0.001s
 $ > time ./one in2
9
```

```
real    0m0.004s
user    0m0.003s
sys     0m0.001s
 $ > time ./one in2
9

real    0m0.004s
user    0m0.002s
sys     0m0.002s
```

The only relevant information for now is in the "real" row.

```
real    0m0.001s
real    0m0.004s
real    0m0.004s
real    0m0.004s
real    0m0.004s
real    0m0.004s
real    0m0.004s
real    0m0.004s
real    0m0.002s
real    0m0.004s
real    0m0.004s
```

First important observation: a program running on the same data will get different times. This is the first surprising result, but this is in fact normal.

Environment

Your operating systems are almost certainly multi user and multi task. So You can never be absolutely sure how long a process will take, as other processes might be running at the same time.

Just because of this simple fact, measuring things precisely is hard, and repeatable results are difficult to accomplish.

So let us run the program again, but this time I made sure that the processor was busy. I ran a CPU intensive task in the background that utilised all of my cores many times over.

```
real    0m0.021s
real    0m0.015s
real    0m0.020s
real    0m0.010s
real    0m0.015s
real    0m0.015s
real    0m0.018s
real    0m0.006s
real    0m0.017s
real    0m0.014s
```

And this time I got results which are 4-10 times slower.

I will be repeating this over and over again until You understand this deeply: the environment in which Your code runs influences how fast Your code runs. In the simplest case: running the same code on different machines will influence how fast it runs. So running the same code on two different processors will make one execute faster, and the other slower. Running the same code on two different architectures (say, x86-64 vs MIPS or ARM vs SPARC) might make one code run faster. The same goes with RAM (DDR2 vs DDR5, guess which one will be faster?), disk (reading data from a tape drive vs nvme). But even on the same machine, You can get very different results depending on multiple factors:

1. Are intensive tasks running in the background (for instance windows updater, a process which makes backups, or any other program that uses CPU?),
2. are You running the code on the same operating system: will the code run as fast on Windows as it will on Linux? Were any changes done to the operating system between tests, such as updates (for instance, an update might have put in place mitigation against Rowhammer or SPECTRE which might slow code)?
3. are You running the code in a VM? In different VMs? in VM vs a non-virtual system?
4. is the power cable connected (for instance, laptops with power saving features will slow down their CPUs when their power cables are detached)?
5. and many others...

What can be done about it? Always run the same code in the same environment: on the same machine, compiled with the same compiler, with the same updates, on the same VM and so on and so on.

Statistics

Even when we guarantee that the environment is always the same and unchanging, this still does not guarantee the same results, as we saw above. This is when statistical analysis of data comes into play. Let us perform the same task multiple times, and then we can use statistical tools on the data. We can calculate the minimum, maximum, mode, mean, standard deviation and variance of data, which may give us information on the quality of data. Is the data clustered around one point, or is it spread around? Are there any outliers -- data points which are so different from others, that they must be an error, and should be disregarded (but be careful with that)?

We could even use tools such as outlier detection to tell us which data points should not be taken into consideration at all.

Since You are not statisticians (and I am not one as well), be warned: statistics in inept hands make results worse, not better. It may make You more confident where caution should be exercised.

For our simplistic purposes, let us also do the simplest possible thing: let us perform the test multiple times, and average the results. This has several virtues: it's trivial to implement, and if there are enough data points then the outliers should have negligible effect. But keep in mind that there are also many caveats with this approach as well: as the old saying goes, a man and his dog have three legs, on average.

Let us see what happens when we perform the same test 100000 times, and average out the results.

This is the data, grouped by the time it took for the program to run.

```
time     count
======== =====
0m0.001s 22652
0m0.002s 76787
0m0.003s   362
0m0.004s   118
0m0.005s    41
0m0.006s    14
0m0.007s    11
0m0.008s     5
0m0.009s     3
0m0.011s     2
0m0.013s     2
0m0.015s     1
0m0.018s     1
0m0.020s     1
```

On average it took 0.00178318 seconds to get the result.

Judging by the above, it should probably take about 0.002s, perhaps faster, so the average is probably not far from the truth. As You may have noticed, there are some outliers: data points, which are probably false (once it took 0.02s instead of 0.0002 -- some other program probably started running in the background, or the hard drive was written to, things like that) . In this case, they might be safely ignored. Be aware, that this is not always the case, and by ignoring these You might be ignoring important data.

Be also aware that the number 0.00178318 looks very precise, when in fact it is anything but precise. The precision of execution time reported by `time` is probably around one millisecond -- and this is simply not enough. But we will come back to the topic of precision later.

How it all scales up...

So now we know how to get results that are less affected by accidents of multi-program operating systems. We also know how long a program runs for 5 operations (+ 1 initial value). The next thing that interests me is this: is this a general trend? That is: how will the program behave on larger inputs? If I have 10 lines, will the program run twice as long? Or

perhaps four times? If I have 100 lines, will the program run 20 times as long as the program which has 5 operations, will it run an hour, or the next million years? All are possible.

Let us run it, and time it!

Unfortunately, we get some very strange results:

```
operations time
========== ========
100        0m0.005s
200        0m0.005s
300        0m0.001s
400        0m0.001s
500        0m0.002s
600        0m0.002s
700        0m0.002s
800        0m0.002s
900        0m0.002s
1000       0m0.002s
```

It does seem that doing 100 operations takes longer than 1000 operations -- and there is very little difference between doing 500 operations, and doing 1000 operations! That is unfortunately the limit of two things: precision of `time`, and a problem of checking the time of running the whole program, instead of timing just a small portion of it.

How long does a part of code run...

So let us finally go back to coding. Let us suppose that we have this code that performs the required task:

```
#include <iostream>
#include <fstream>

int main(int, char **argv)
{
    std::ifstream f(argv[1]);
    char op;
    int num;
    int res;

    f >> res;
    f.ignore(1, '\n');

    while (f >> op >> num) {
        f.ignore(1, '\n');
        switch (op) {
        case('*'): res *= num; break;
        case('/'): res /= num; break;
        case('-'): res -= num; break;
        case('+'): res += num; break;
        }
    }

    std::cout << res << std::endl;
}
```

First, let us think very carefully about what are we going to test. Depending on what our task is, we can exclude certain things. For instance, what is the point checking how long we print the result? In fact, this happens only one time -- and printing the result is not a necessary part of the algorithm (we might as well be storing it in a database or someplace else entirely). Same thing about opening a file (creating an ifstream) and declaring variables: they are not a strictly necessary part of the algorithm, and variables are located on the stack, so we can't really measure their declaration time in code anyway. But the rest is a necessary part of the algorithm. We want to measure this part of the code:

```
    f >> res;
    f.ignore(1, '\n');

    while (f >> op >> num) {
        f.ignore(1, '\n');
        switch (op) {
        case('*'): res *= num; break;
        case('/'): res /= num; break;
        case('-'): res -= num; break;
        case('+'): res += num; break;
        }
    }
```

We shall now try to measure the time a piece of code takes. Doing this is can be... interesting.

First, let us try the C++ way.

std::chrono

`std::chrono` is a collection of types and functions which relate to time (as the name suggests). To use it, we first `#include <chrono>`, and then we are good to go. Let us try and measure how many nanoseconds the code takes.

```
#include <iostream>
#include <fstream>
#include <chrono>

int main(int, char **argv)
{
    std::ifstream f(argv[1]);
    char op;
    int num;
    int res;

    auto start = std::chrono::steady_clock::now();
    f >> res;
    f.ignore(1, '\n');

    while (f >> op >> num) {
        f.ignore(1, '\n');
        switch (op) {
        case('*'): res *= num; break;
        case('/'): res /= num; break;
        case('-'): res -= num; break;
        case('+'): res += num; break;
        }
    }
    auto end = std::chrono::steady_clock::now();
    std::chrono::nanoseconds time_taken(end - start);

    std::cout << time_taken.count() << std::endl;
}
```

First, we use `std::chrono::steady_clock::now()` to get the current time at that point in the execution of the code, and store the value in `start`. Then we perform the task at hand. And finally, we get the time after the task was performed, and we store it in `end`. The total time of performing the task must therefore be `end - start`. We create a variable of type `std::chrono::nanoseconds`, which is useful for storing time in nanoseconds, as the name suggests. Then we initialise it with the value `end - start`, which is automatically cast to `std::chrono::nanoseconds`. We then print it out -- we have to use the `.count()` method on `time_taken` to get a numerical representation.

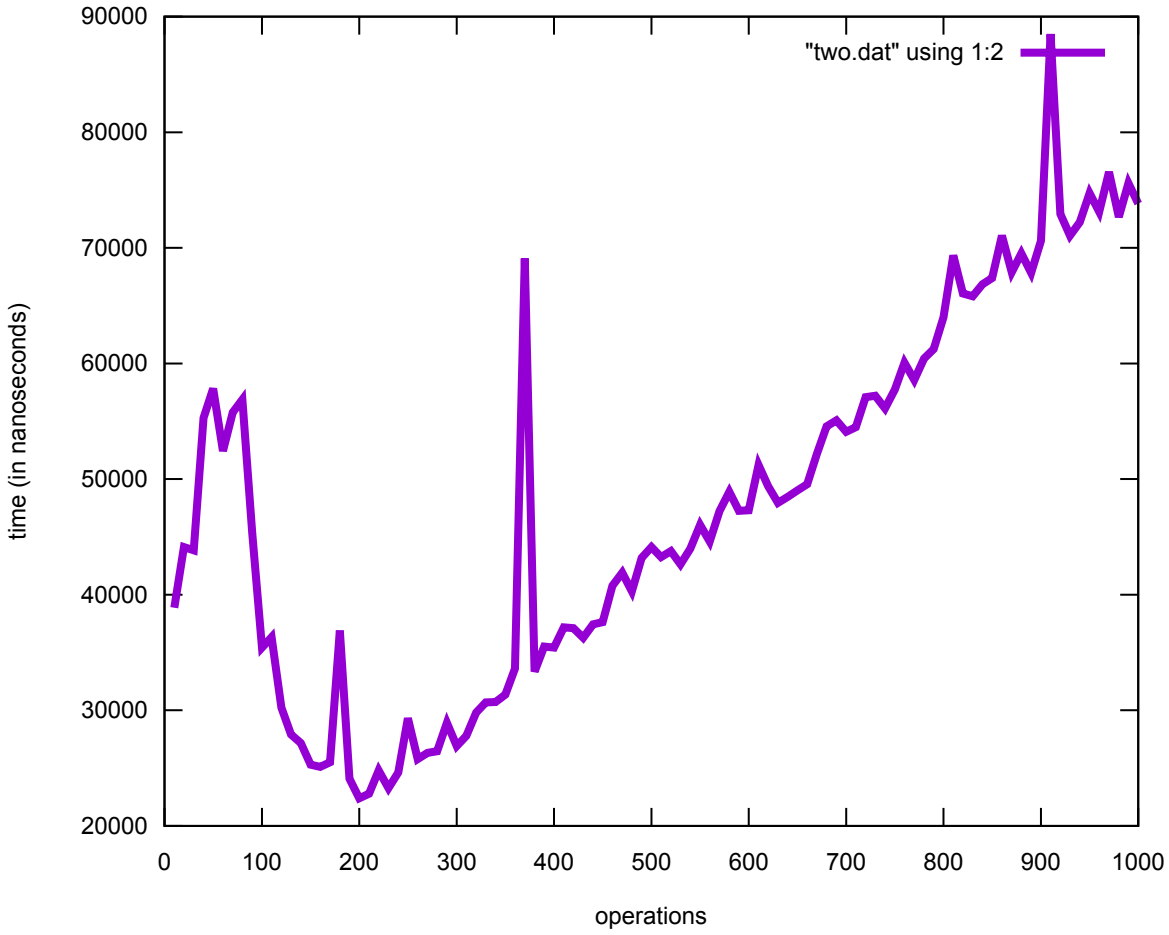
Let us try it on values from 10 up to 1000, incremented by 10.

```
operations  time (ns)
==========  =========
10              38864
20              44120
30              43834
40              55284
50              57832
60              52433
70              55777
80              56877
90              45403
100             35421
110             36327
120             30224
130             27910
140             27160
150             25303
160             25112
170             25498
180             36893
190             24094
200             22390
210             22783
220             24797
230             23277
240             24628
250             29319
260             25788
270             26319
280             26456
290             28920
300             26903
310             27788
320             29804
330             30686
340             30714
350             31362
360             33608
370             69088
380             33357
390             35506
400             35428
410             37184
420             37108
430             36273
440             37434
450             37630
460             40774
470             41903
480             40291
490             43181
500             44120
510             43253
520             43768
530             42633
540             43981
550             46047
```

```
560             44610
570             47227
580             48889
590             47253
600             47297
610             51221
620             49377
630             47936
640             48462
650             49010
660             49560
670             52211
680             54567
690             55078
700             54094
710             54490
720             57089
730             57207
740             56124
750             57750
760             60065
770             58570
780             60422
790             61239
800             63998
810             69336
820             66048
830             65805
840             66849
850             67381
860             71062
870             67892
880             69517
890             67837
900             70614
910             88495
920             72924
930             71064
940             72211
950             74735
960             73112
970             76558
980             72662
990             75598
1000            73829
```

I bet that You have not even looked at all the values, but I guarantee that they *are* interesting!

Fortunately, we have the perfect tool to visualise the results: a graph.



First, notice that we have the same problem again as we did before: look at 900, 910 and 920. 910 has less operations to perform than 900, and it takes far longer (which is expected) -- but it also takes longer than 920! And this is the same for others as well: 20 takes longer than 400, which is insane!

But there are some clear information that can be read: values from 10 to approximately 200 are quite probably garbage, since they take more than the larger values. From 200 to 1000 we can just barely see a linear progression, but it is obscured with noise in the data.

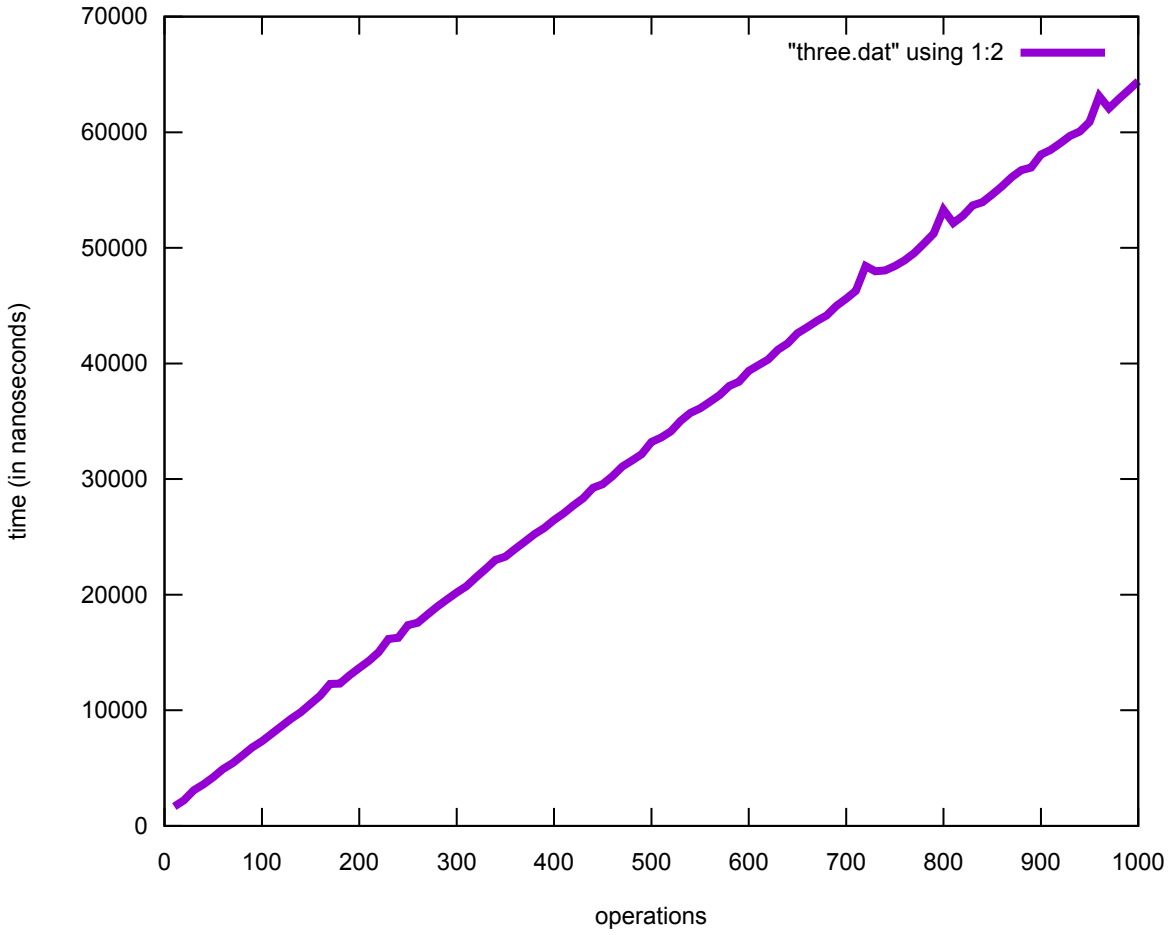
Fortunately, we know what to do: run the code multiple times, and average out the results!

```cpp
#include <iostream>
#include <fstream>
#include <chrono>

int main(int, char **argv)
{
    char op;
    int num;
    int res;
    std::chrono::nanoseconds total_time_taken(0);
    long times = 10000;
    for (long i = 0 ; i < times ; i++) {
        std::ifstream f(argv[1]);

        auto start = std::chrono::steady_clock::now();
        f >> res;
        f.ignore(1, '\n');

        while (f >> op >> num) {
            f.ignore(1, '\n');
            switch (op) {
            case('*'): res *= num; break;
            case('/'): res /= num; break;
            case('-'): res -= num; break;
            case('+'): res += num; break;
            }
        }
        auto end = std::chrono::steady_clock::now();
        std::chrono::nanoseconds time_taken(end - start);
        total_time_taken += time_taken;
    }
    std::cout << total_time_taken.count() / times << std::endl;
}
```

```
operations  time (ns)
==========  =========
10               1678
20               2220
30               3090
40               3603
50               4206
60               4925
70               5436
80               6102
90               6785
100              7330
110              7959
120              8625
130              9261
140              9833
150             10556
160             11269
170             12276
180             12307
190             13025
200             13668
210             14282
220             15025
230             16163
240             16282
250             17370
260             17565
270             18276
280             18958
290             19590
300             20187
310             20718
320             21509
330             22224
340             23002
350             23293
360             23945
370             24587
380             25248
390             25759
400             26461
410             27038
420             27728
430             28343
440             29240
450             29561
460             30243
470             31079
480             31592
490             32158
500             33201
510             33588
520             34127
530             35030
540             35711
550             36125
```

```
560             36667
570             37268
580             38057
590             38423
600             39351
610             39858
620             40343
630             41186
640             41740
650             42614
660             43135
670             43679
680             44155
690             44973
700             45610
710             46286
720             48434
730             47979
740             48051
750             48425
760             48910
770             49551
780             50379
790             51240
800             53298
810             52141
820             52781
830             53671
840             53946
850             54614
860             55322
870             56118
880             56714
890             56941
900             58069
910             58466
920             59051
930             59686
940             60055
950             60892
960             63129
970             62063
980             62885
990             63622
1000            64404
```

Now these look much better. Let us look at the graph as well:



There are some small "bumps" here and there, which might or might not be eliminated increasing the number of times we perform the test. But this time the result is a clean, clear, crisp, straight line.

The data that we have collected allow me to do even more. Based on that data, I predict, that on average it should take about 64.404 ns * number of operations to perform the desired computation. We can get to this number, by taking the time it took to calculate a 1000 operations, and then average out how long it took to do one of them. Computing 100 million operations should therefore take approximately 6440400000 nanoseconds, so I bet it should take approximately 6.4 seconds to perform 100 million operations. And indeed, one experiment took me 6317935910ns, approximately 6.3seconds. Mind You, we could average it out over 10000 tries, just as we did before -- but that would probably take around 64000 seconds, or more or less 18 hours. If You have that much time to spare, go ahead. Since I have not enough patience, I have averaged it out only over 100 tries -- it took no longer than 20 minutes to get the value 6325429904ns.

Obviously, the results You get will be different from the presented values.

however...

Unfortunately, the method presented above might not work, or the clock precision might be too small to be of any use. If this does not work for You, we can try to do two more things.

First thing is to use C. The C11 standard defines a function `clock_gettime` in the time.h header. Let us try to use it.

```
#include <iostream>
#include <fstream>
#include <ctime> // or <time.h>

constexpr long long nanoseconds = 1000000000L;

int main(int, char **argv)
{

    char op;
    int num;
    int res;

    long long total_time_taken = 0;

    long times = 10000;
    for (long i = 0 ; i < times ; i++) {
        std::ifstream f(argv[1]);

        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        f >> res;
        f.ignore(1, '\n');

        while (f >> op >> num) {
            f.ignore(1, '\n');
            switch (op) {
            case('*'): res *= num; break;
            case('/'): res /= num; break;
            case('-'): res -= num; break;
            case('+'): res += num; break;
            }
        }
        clock_gettime(CLOCK_MONOTONIC, &end);

        total_time_taken +=
            (end.tv_sec - start.tv_sec) * nanoseconds +
            (end.tv_nsec - start.tv_nsec);
    }
    // std::cout << total_time_taken << std::endl;
    std::cout << total_time_taken / times << std::endl;
}
```

The code looks very much the same with minor differences. The `clock_gettime` sets two fields in a `struct timespec`: `tv_sec`, and `tv_nsec`. `tv_sec` is the number of elapsed seconds, and `tv_nsec` is the elapsed fractions of a second (the maximum precision we can have is generally a nanosecond).

The problem with this approach, is that it might not work as well, or `CLOCK_MONOTONIC` might not be supported, or the precision might be too small to be of any use. If

`CLOCK_MONOTONIC` is not supported, `clock_gettime(CLOCK_MONOTONIC, &start);` returns -1, so be sure to check the return value if You get strange results.

Any of these approaches should work on Linux, Mac or similar, and Windows, when used with a recent Visual Studio compiler. If both of these approaches fail, You are probably using Code::Blocks or DevCpp on Windows (perhaps this would be a good time to try out some better alternatives than these two).

Fortunately there is also one more way for people using Windows: using Windows libraries to get high precision time.

```
#include <iostream>
#include <fstream>
#include <windows.h>

constexpr long long nanoseconds = 1000000000L;

int main(int, char **argv)
{
    LARGE_INTEGER fr;
    QueryPerformanceFrequency(&fr);
    double freq = fr.QuadPart/1.0;

    char op;
    int num;
    int res;

    long long total_time_taken = 0;

    long times = 10000;
    for (long i = 0 ; i < times ; i++) {
        std::ifstream f(argv[1]);
        LARGE_INTEGER start, end;
        QueryPerformanceCounter(&start);
        f >> res;
        f.ignore(1, '\n');

        while (f >> op >> num) {
            f.ignore(1, '\n');
            switch (op) {
            case('*'): res *= num; break;
            case('/'): res /= num; break;
            case('-'): res -= num; break;
            case('+'): res += num; break;
            }
        }
        QueryPerformanceCounter(&end);
        total_time_taken += end.QuadPart - start.QuadPart;
    }
    // std::cout << total_time_taken << std::endl;
    std::cout << total_time_taken / freq / times << std::endl;
}
```

This time we include `<windows.h>` to get `LARGE_INTEGER`, and two functions: `QueryPerformanceFrequency`, which should give You the counter frequency, and `QueryPerformanceCounter`, which shall function like a clock for us.

If this still fails to give You a proper time, definitely please contact me because Your setup must be pretty strange -- and we'll cook something up. In the meantime, connect to universities linux machines and code on those -- they should work.

conclusions

The data visualised in the above graph allows us to claim that the program takes linear time with regards to the size of the input. Not only that, it give us predictive abilities: based on the time taken, we may predict what will happen for 2000, 3000, 4000, 5000, 10000, 100000, 1000000, 10000000 or more operations. But always remember that we infer conclusions based on data. Remember that the data is almost always imprecise and incomplete (and may omit various important data points) -- so our predictions will almost always be inaccurate. We always predict within some error bounds.

summary

In this class we have learned how to measure the time a part of our program takes.