COMP 161 — Lecture Notes 17
Profiling and Complexity

In these notes we look at measuring performance in the real-world as the experimental counterpart to complexity analysis.

Efficiency in the Wild

Complexity theory and its tools tame the complicated world of efficient computation. It has proven itself to be an invaluable tool for understanding and discussing efficient computation. Efficient programs are designed to minimize complexity. However, minimizing complexity is only the first step. Within the realm of minimal complexity is a whole host of fine-grained details that drive actual performance. Let's return to the car comparison. Best practices might let you guarantee a car with 20–25 miles per gallon fuel efficiency, but the actual nuts and bolts of how you put that car together are likely going to be the difference between a real world performance of 20 and 25. The same is true with programming. We must first choose efficient algorithms[1], and then efficiently implement them on systems[2] that can achieve optimal performance. It's nearly impossible to build efficient programs with clever implementation of high complexity solutions. On the other hand, it's easy to achieve poor efficiency of low complexity solutions by making bad implementation choices.

[1] low complexity

[2] platforms and languages

When we assess the complexity of our programs we're analyzing the algorithms. When we assess the real-world efficiency of our program we're often PROFILING its performance. Profiling is concrete and from the scientific perspective it is the empirical verification of the underlying (complexity) theory. Profiling is also appealing because it deals in real world measures. Time is measured by the clock and space by the byte.

Profiling

Powerful profiling tools exist that for measuring real-world performance of programs. For programs written in C++ two candidates are *gprof* or *valgrind*. These tools collect profiling data about the program by taking measurements as the program runs. In this scenario you're attempting to see how the program, as a whole, behaves on typical data.

If, however, you're attempting to experimentally verify the efficiency of a specific procedure, then it's often better to write programs specifically designed to measure performance. Here we get to choose a specific range of data that covers the whole spectrum of inputs

from best to worst case. Our primary concern is understanding how much time it takes procedures to run.

Getting Execution time with chrono

The general pattern for timing code is to check the time prior to code execution and then again after code execution. You then subtract the start time from the end time to get the elapsed duration. The C++11 library *chrono* provides several classes that can be used to time the execution of code. There is a complete example for getting the execution time of a line of code in the example for the *now* function, which gets the current time[3]

To get the current time we'll use a *high_resolution_clock::time_point* object. As the name indicates, these objects measure time at a high resolution[4]. We can store the difference of two time points as a *duration<double>* which, by default, tells us the number of seconds as a double. We can access the number of seconds with the duration class method, *count*.

Let's say we wanted to time the *std::find* procedure[5]. Then we might build a main procedure with the following:

[3] `http://www.cplusplus.com/reference/chrono/high_resolution_clock/now/`

[4] nano seconds

[5] aka search `http://www.cplusplus.com/reference/algorithm/find/`

```
// search data
 std::vector<int> data{1,2,3,4,5,6,7,8,9,10,11};
 // get iterators now so they aren't captured in the timing data
 auto data_fst = std::begin(data);
 auto data_end = std::end(data);


 // Timing Data
 std::chrono::high_resolution_clock::time_point start;
 std::chrono::high_resolution_clock::time_point end;
 std::chrono::duration< double >  elapsed;


 // Gather a single time data point
 start = std::chrono::high_resolution_clock::now();
 std::find(data_fst,data_end,9);
 end = std::chrono::high_resolution_clock::now();

 elapsed = std::chrono::duration_cast< std::chrono::duration<double> >(end-start);

 std::cout << elapsed.count() << " secs\n";
```

This code is simply a modification of the example found in the documentation for *high_resolution_time::now*. What it illustrates is that the

time oriented classes provide all the functionality we need for getting execution time and that we can even do things like subtract time points using *operator-*. Operators also work with *duration<double>*, so you can add, subtract, multiply and divide durations as needed[6]. It's also important to note that the time reported by *elapsed.count()* is in seconds.

[6] See `http://www.cplusplus.com/reference/chrono/duration/operators/`

Some notes on chrono

It's pretty clear from our example the the namespaces and types involved in the chrono library are quite verbose. One way to make things easier would be to use a *using namespace* directive with the namespace *std::chrono*. In doing so we clearly highlight the types and considerably shorten the type declarations. If you go back to the above example and ignore all the instances of *std::chrono*, then we find the following types are used in our profiling code:

high_resolution_clock::time_point	`http://www.cplusplus.com/reference/chrono/time_point/`
duration<double>	`http://www.cplusplus.com/reference/chrono/duration/`

It's worth spending some time with the documentation for the classes in order to get a general sense of what they represent and how they work. We'll soon be dealing with vectors of durations and we'll want to know how to hard code a duration object's initial value in order to do things like testing. It's also a good idea to scan through the chrono documentation[7] in order to get a better understanding of what things like *high_resolution_clock::now()* and *duration_cast* do.

[7] `http://www.cplusplus.com/reference/chrono/`

Experimental Analysis of Algorithms

Experimental Algorithmics is the experimental arm of algorithm analysis. It empirically tests the theoretical performance bounds laid down by complexity-based analysis. If you have already done a complexity analysis and have an expected run time performance, then you can use experiment to validate your implementation against the underlying algorithm's complexity. Experimental studies can also guide theoretical analysis. You could build a through profile of the procedure's run-time and use that to guide a complexity analysis.

Whether we're using experimentation to validate theory or in attempt to discover underlying theoretical properties, we need to generate a sufficiently robust set of experimental data. Let's go back to thinking about *std::find* and the underlying search algorithm it

employs and think about the data we'd need to properly study this procedure.

The complexity of std::find

According to cplusplus.com, *std::find* has the following complexity[8]:

> Up to linear in the distance between first and last: Compares elements until a match is found.

[8] `http://www.cplusplus.com/reference/algorithm/find/`

Let's simplify the discussion a bit by thinking about searching the entire contents of a vector of integers. This means that the "distance between first and last" is the same as the size of the vector. In this case, the complexity is:

> Up to linear in the size of the vector: Compares elements until a match is found.

The first and most important thing to take away from this description is that *complexity is a function of the vector's size*. More specifically, we know that the function is linear. But if the vector's size is the function input, then what's it's output? We're talking about time complexity so what we're looking at is the number of elementary operations carried out by the computer. Let's fill some details in:

> The number of elementary operations carried out by the computer when computing *std::find* is up to linear in the size of the vector being searched because *std::find* compares elements until a match is found.

Earlier I said that we typically think in terms of the worst case. The language "up to linear" implies that worst case upper bound is linear. For this reason we call std::find a *linear time procedure*.

We can express all of this very concisely using mathematics. Let $\mathcal{T}$ be the function that computes the elementary operations carried out by std::find given the size n of the vector being searched. Then there must exist some $a > 0$ and b such that,

$$\mathcal{T}(n) \leq an + b \tag{1}$$

When we look at the math, we see how much we're simplifying matters. By saying that std::find is linear, we're ignore the details of the line[9] and simply saying, "I can draw a line such that all the values of the time function $\mathcal{T}$ are on or below that line". In the next set of notes we'll learn BIG-O NOTATION which allows us to express this kind of mathematical information in an even more concise manner. Big-O notation is the official "language" of complexity. You'll learn some basics in this class and continue to use and refine your understanding of it as you progress in the major.

[9] the value of a and b

Verifying the complexity of std::find

How can we experimentally verify that std::find is, in fact, a linear time procedure? Choose some arbitrary but "large" size for your vector[10]. Now, imagine you ran all possible searches for all the sizes up to your choosen size and recorded the time it took for each search. Here we assume that these elementary operations always take some fixed amount of time so that the real time is just some multiple of elementary operations. Finally, plot those time points on a graph where the x-axis is vector size and the y-axis is time. What do you expect to see?

[10] maybe something like 100000

If you said, "a line" you're both right and wrong. First off, for any given size there are many different searches. Look back at the complexity description:

> The number of elementary operations carried out by the computer when computing *std::find* is up to linear in the size of the vector being searched because *std::find* compares elements until a match is found.

The "compares elements" part implies that we traverse the vector until we find the item we're looking for. Sometimes that item is the first item and sometimes it's the last. Sometimes it's in the middle and sometimes it's not there at all. If we wanted to be more specific, then we could identify $n + 1$ cases for vectors of size n[11]. On our graph, we'll see $n + 1$ points at each size n. It seems highly unlikely that each of these sub-cases of n will take exactly the same time. Don't agree? Let n be 10000000. Do you think it will take the same amount of time to look at one element (the case where it's the first item) as it will to look at all of them (the case there it's not in the vector)? I don't.

[11] n locations and not there at all

Even if it were possible for all $n + 1$ sub-cases to have the same running time, we'd have to contend with experimental error. Running programs isn't really just a function of elementary operations. Lots of things can happen in the memory system that affect running time. The operating system could be multi-tasking and the other tasks could affect our procedure's execution. More formally, we have to expect some noise, some variance from one experiment to the next even when the inputs are exactly the same.

We won't see a line because we have multiple data-points (cases) per x-value (vector size) and our data will be a bit noisy. We also won't see "a line" in the data because (worst case) complexity isn't about the data, it's about *an upper bound on the data*. The line you should see is the one you can draw just above all of your data points. This is what we expect to see. This probably means we can find lines and a general linear shape in the data, but most importantly it means we can box all of our std::find data into a quadrilateral with sides the

x-axis, y-axis, max size value, and the upper-bound line. This general shape is what we want to see.

Gathering Data

The data set we need to verify the efficiency of std::find should be clear. The complete set involves execution times for all $n+1$ cases for each vector size n from $[0, m]$ where m is some suitably large upper bound on the vector size. This set is so regular that you can[12] write programs to gather it for you automatically. Even still, we need to understand some issues inherent in this data set and what we can do with it.

[12] and will!

The first problem we see in this approach is that we really don't know what happens after m. That's OK. Mathematics and statistics will let us make some pretty sound inferences about this unknown region of std::find's efficiency. It may also be OK because we simply don't care about vectors above a certain size. Either out problem deals with a fixed max size or sizes above m don't fit in the computer's memory and we need a different algorithm anyway.

The second limitation of the data set is that it quickly becomes unwieldy. We can actually compute exactly how many data points we're talking about using some basic discrete math[13].

[13] you don't need to know this math...yet

$$\begin{aligned} \sum_{n=0}^{m} n+1 &= m+1+\sum_{n=0}^{m} \\ &= \lceil \frac{m^2}{2} \rceil + m + 1 \end{aligned} \tag{2}$$

This quadratic function grows pretty fast.

m	size of data set
5	19
50	1301
500	125500
5000	12505000
500000	125000000000

Thankfully, we know from statistics that we don't really need all of that data to see the pattern. It's enough to have a representative sample. What constitutes a representative sample varies a bit from situation to situation and what, specifically, we're trying to determine from the data. In the case of our std::find analysis, we'd like to get some verification of the linearity of the worst case as well as the average behavior of the procedure. What we need is a representative sample of sizes and for each of those sizes a representative sample of sub-cases.

A classic method for choosing samples is to select at random. In this situation we'd choose some fraction of $[0, m]$ at random and for

each of those sizes we choose a random fraction of the sub-cases. A key advantage to this process is that randomness is not biased. If our goal is to verify the linear complexity of std::find, then we should see that linear pattern in any sufficiently large random sample. We might think to only test what analysis tells us is the worst case (key not found), but this assumes that our theoretical analysis of the algorithm mirrors the implementation. If we test only the key not found cases, then we don't know if some other case just happens to trigger some kind of unexpected worst case behavior. If we do it right, randomness saves us from this bias. We can then apply the same principle to the sub-cases for each size.

On the other hand, we do know about some underlying structure in this problem. Size 0 and size m really are bookends to the space that we're exploring, so it makes sense to test them specifically. With this in mind we could compromise by testing size 0 and m then doing a random sample of $(0, m)$. We can do the same thing with the size sub-cases. Test when the key is the first and when it's not found (our expected best and worst) and then test a random sample of when the key is at an index in $(0, n)$ for vector size n. The key here is that we're utilizing structures found in the overall problem to guide our sample selection.

If we know something about the implementation, then we might use those structures to guide the sampling as well. If the search proceeds in the standard zero to size fashion, then rather than randomly select key locations we might just choose evenly spaced out cases. In this case we might test the cases where the key is at an index that is a multiple of 3. This covers $\frac{1}{3}$ of the total cases and should paint a pretty clear picture of how the search plays out as we do more and more repetitions of the loop. Of course, this kind of targeted sampling is possible only when we can control the procedure inputs. For some problems it's hard to specifically select this intermediate, average cases. In these situations, we have to rely on randomly generated inputs.