# *COMP 161 - Lecture Notes 14 - Efficiency and Complexity*

In these notes we look at the basic ideas of program and algorithm efficiency and some techniques for profiling the run time requirements of a C++ procedure.

## *Efficiency, Complexity, and Computational Resources*

Being efficient means optimizing the use of one or more resources. A fuel efficient car will travel far while minimizing the consumption of fuel. This means the ratio of miles traveled per gallon of fuel is high relative to a less efficient vehicle[1]. When comparing cars for fuel efficiency we tend to look at things like the expected miles per gallon or perhaps the highest achievable miles per gallon. In both cases, we're operating the car not under a guarantee of peak efficiency but with an understanding that we can expect a certain level of efficiency when driving. In short, we understand that manufactures typically provide us with theoretically achievable efficiency for our vehicle and that in practice, mileage may vary.

[1] conversely the ratio of gallons consumed per mile is lower

Of course efficiency is not a singular measure. It is relative to the consumption of a specific resource. Perhaps the consumption of fuel is not your concern, but time is. A time efficient car would exhibit a higher acceleration rate and/or top speed. We typically expect that this increased time efficiency came at the cost of fuel efficiency. So not only is efficiency subjective, but the overall efficiency of a vehicle involves trade offs. Less time means more fuel, less fuel means more time.

### *Complexity*

Enough about cars. What about computers and computation? The most discussed and emphasized computational resource around is TIME. Time efficient computations take less time. We then tend to look at other resources in terms of their relationship with time. By increasing the SPACE[2] needs of a computation we can often decrease the *time* needs. Conversely, decreasing space often requires increased time. In this way space might be viewed as the fuel of computation. There are still more resources used in computation. When we get into parallel computation, we start to think about things like COMMUNICATION as a resource. The less computers working in parallel have to communicate, the less time they need and vice versa.

[2] memory

COMPLEXITY THEORY is the branch of theoretical computer science in which problems are classified by the resource needs of their solutions. Go look at the documentation for C++ library procedures and class methods on cplusplus.com and you'll see a section title *Complexity*. This is exactly the complexity theoretic view of the efficiency of that procedure. It provides the programmer with a broad classification of the efficiency of the procedure in favor of a fine-grained view. From the programmer's perspective, complexity measures of efficiency provide you with an expectation of efficiency in much the same way that a car manufacture's reported miles per gallon does for fuel efficiency.

Unless otherwise specified, complexity classifications are based on the WORST CASE resource needs of the procedure. The worst and best case are often generated by a very specific variation of the input and it's easier to be concrete about their resource needs. For example, the basic search procedure's best case is when the item we're looking for is the first item we look at and it's worst case is when it's the last. The average case is much less specific; it's when the item we're looking for is neither the first nor the last. Given a choice between best case and worst case thinking, we err on the side of worst case. Knowing that your program needs at most some amount of time means that on average we can can expect that efficiency or better. This UPPER BOUND on the resource needs is ultimately more useful to the programmer than the *lower bound* offered by the best case.

Complexity theory is carried out in a machine and language agnostic fashion. We measure time by counting basic operations, not by the clock. We measure space sometimes by bits and bytes but sometimes by abstract, discrete units of data. By classifying procedures in this way we are able to rule out the language used and the machine executing the code and instead focus on some underlying abstract model of computation. In doing so, the assessment of a procedure's complexity becomes an assessment of the underlying *algorithm* and not the actual implementation. As a discipline of study, complexity theory is smack dab at the intersection of mathematics and computer science[3]. For our purposes, we simply with to apply the basic principles, broad categorization of program efficiency based on the the expected worst case performance of the procedures, to program design and evaluation.

[3] The work itself is done in mathematics where the objects of study are those of computer science. The distinction between the two disciplines gets awful blurry in this realm

## *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[4], and then efficiently implement them on systems[5] 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.

[4] low complexity

[5] 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 theory. Profiling is also appealing because it deals in real world measures. Time is measured by the clock and space by the byte.

## *Profiling*

Our previous efforts at profiling code did not require any changes to the code that we wished to profile. All tAt most, we'd write some contrived *main* procedures to make specific calls to the target code. Profiling without modifying existing library code allows us to profile a program in context. Often this lets you get a good idea of what's going on under the hood. On the other hand, it produces data about a lot of code we're not concerned with and can prevent us from getting the kind of detailed data we're looking for.

Sometimes we're not worried about code within a specific application, but instead we're focused on detailed experimentation. For example, to get a good picture of the behavior of a specific vector procedure we're likely to want to run the procedure on a large number of inputs and gather individualized data about each execution.

When optimizing a program, it's often better to use dedicated profiling tools such as *gprof* or *valgrind*. These tools collect profiling data about the program by gathering data as it runs. In this scenario you're attempting to see how the program 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.

## 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[6]

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

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[7]. 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*.

[7] nano seconds

Let's say we wanted to time the *std::find* procedure[8]. Then we might build a main procedure with the following:

[8] 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 = begin(data);
 auto data_end = 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[9]. It's also important to note that the time reported by *elapsed.count()* is in seconds.

[9] 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[10] in order to get a better understanding of what things like *high_resolution_clock::now()* and *duration_cast* do.

[10] `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[11]:

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

[11] 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[12] 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.

[12] 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[13]. 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?

[13] 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 $n$[14]. 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.

[14] $n$ 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[15] 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.

[15] 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[16].

[16] 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