RES.TLL-004 STEM Concept Videos, Fall 2013

Transcript – Algorithm Efficiency

Information Flow: Efficient Algorithms

Host: We're in Cambridge, Massachusetts, MIT's backyard, for a side-by-side sorting speed challenge. We have a 1970's computer and a brand new 2013 laptop. Who do you think will sort this list first?
Actor: um, the new laptop?
Host: The old computer won!
Actor: woah!
Host: Algorithms beat hardware!

This video is part of the information flow series. A system is shaped and changed by the nature and flow of information into, within, and out of the system.

Hi, my name is Charles Leiserson, and I am a professor of Computer Science and Engineering at MIT.

Before watching this video, you should be familiar with computer programming, recursion, and algorithms.

After watching this video, you will be able to: Identify common resource limitations when programming; Understand speed and space and how they may be related; and Understand how efficiency affects modern problem solving.

Chapter 1: Measures of efficiency in computing

The goal of an algorithm is to solve a particular problem. But there are many ways to design and implement your algorithm—some more efficient than others. But what does "efficient" mean? In this video, we will talk about different types of efficiency and how they relate to limited computing resources.

To understand some measures of algorithm efficiency, let's imagine your computer as a kitchen with a cook and counter space, where our goal is to produce meals.

The first measure of efficiency we will consider is speed. Naturally, we are hungry, so we want our meals to be ready quickly. We can accomplish this by simply demanding the cook to cook faster. Similarly, computers can operate faster by utilizing faster processor chips. However, just as any person can only cook so fast, processor chips have a maximum speed.

As we will explore later, there are many other important strategies we can use to help our poor cook prepare meals more quickly.

Second, we will consider space. The cook has access to a limited amount of space to work in the kitchen. Have you ever prepared a very large meal and ran out of space on your counter? Computers also have access to a finite amount of workspace, known as random access memory or RAM.

Other than speed and space, what are some other resources we must use efficiently in programming? Pause the video here and brainstorm.

There are many important design aspects that may be unique to the particular problem you are solving. You may have thought of some of the following: data transmission size (such as when you are streaming music onto your computer or phone), long term storage space (such as the hard drive on your computer), and energy usage (does your phone run out of battery more quickly when you play certain apps or use 4G LTE?).

There are many meanings to the word efficiency in programming, and depending on your problem, certain types of efficiency will be more important to you. Today, we will focus on speed and space, which are two very important and often correlated limitations in designing algorithms.

Chapter 2: Speed and space usage in classic algorithm implementations

How do speed and space affect algorithms? Let's start by taking a look at some classic algorithms in a simplified setting.

Let's look at the Fibonacci sequence. This is a sequence of numbers beginning with 0 and 1, and each subsequent number is the sum of the previous two. So the first numbers of the sequence are, 0, 1, 1, 2, 3, 5, 8, and so on. Can we design an algorithm that will take an input n and tell us the nth number of the Fibonacci sequence?

Let's begin with a naïve solution to the problem: Fibonacci of n is simply equal to the sum of the two previous numbers in the sequence, Fibonacci of n-1 and Fibonacci of n-2. To calculate Fibonacci of n-1, we recursively calculate Fibonacci of n-2 and n-3. We also do the same for Fibonacci of n-2. Using this approach, how many function calls do we need for Fibonacci of 8? Pause the video here to try to work this out.

Each Fibonacci instance calls Fibonacci of n-1 and n-2. As we follow the chain of calls that are made, we see that the tree grows very quickly to be a lot of function calls! 67 to be exact. You may have noticed that the same function calls are also repeated several times. For example, Fibonacci of 2 is calculated _13_ times! This is equivalent to asking our cook to prepare separately, from scratch, a cup of chicken soup for 67 separate diners! Even though the time required to cook food grows only linearly with the amount of food we need to make, you can already see how wasteful this approach can be. Can you imagine if cooking time grew exponentially, as is the case with this Fibonacci solution?

Lets take a moment to pause the video here to discuss ways to prevent these wasteful repeat calculations.

We can save a lot of work by cooking a large pot of chicken soup. Whenever the soup is ordered next, we just ladle it from the pot! Similarly, we can cut down on function calls by saving--in memory--the calculations we've already done! Then, as we are going through the algorithm, we simply check to see if the result is already saved in our table. If not, we perform the calculation.

With this approach, which you may recognize as MEMOIZING, we've cut down the number of function calls from _67_ to _15_! We've replaced the majority of function calls with table lookups, which are generally much faster.

But consider now we want to calculate Fibonacci of 1,000,000,000. What happens to our saved results table? The larger the n we want to calculate, the larger the table! Even though memoizing has allowed us to calculate the Fibonacci number much faster, it requires a lot more space to work. Just like there is a limited amount of pre-cooked soup we can hold in our pot, there is a limited amount of memory for our table. Can you think of another way to approach this problem that saves space? Pause the video to discuss.

If we start calculating Fibonacci of 8 from the bottom up, that is starting with Fibonacci of 0 and 1 and adding until we reach the nth value, we only need to save the last two values we calculated.

This approach cuts down on both time, with 8 function calls, and memory space, with 2 values being saved.

It turns out that there are even faster algorithms for solving the Fibonacci problem. However, as you may find in your future work, not all problems can be solved so elegantly. But generally, by saving on the number of calculations and space you require, you can increase your algorithm's speed.

Chapter 3: Efficiency affects modern programs

We've seen some rather simplistic examples of how speed and space are utilized. But how do these little examples extrapolate to modern programming problems? Lets take a look at how speed and space have changed over the years.

An early home computer from the 1970s had a 1MHz processor. That means, within the span of one second, the computer will go through 1 million processing cycles. Sounds pretty good, right?

Well, a modern smart phone processor can run over 1000 times faster than a1970s computer.

And running at about 3Ghz, the state of the art desktop computer can triple this speed again. Plus, many computers work even faster by utilizing 4 to 16 cores, meaning there are multiple cooks operating at maximum speed in this modern kitchen.

Now let us consider random access memory, or RAM on a computer. Our 1977 computer had 4KB of RAM.

In comparison, the smartphone has 1GB of ram, or 1 million times more space!

And our desktop computer? Up to 64 GB!

Clearly, our resources have grown dramatically in the last few decades. But so have our demands.

Consider, for example, electronic trading, a method by which stocks, bonds, and derivatives are bought and sold over the internet.

Billions of trades can occur in a single day, about half of which are executed according to computer algorithms. What do you think is the most important measure of efficiency for this scenario? Pause the video here briefly to discuss.

In the fast-paced stock market world, speed is extremely important. To give you an idea of the timescale resolution financial analysts are interested in, a message traveling along a kilometer of fiber optic cable requires 3 microseconds, and algorithms running only tens of microseconds faster boast a significant advantage over competitors! This is why many companies are physically located close the New York Stock Exchange—so they can save the many microseconds of time!

Consider now the human genome. The DNA sequence that encodes your body consists of over 3 billion base pairs. A common task for computational biologists is to align, or find the closest sequence match for many short sequences to the genome. The time required to match each of these short sequences depends on the length of the short sequence and the length of the whole genome. An average task requires aligning 30 million sequences of length 50 base pairs to the genome. What do you think is the most constricting limitation in this scenario? Pause the video here to discuss.

In plain text, the genome takes up 3.1GB of space, and an additional 2GB is required for the short sequences to be aligned! Although we may run out of patience waiting for these programs to finish, the most limiting resource for the computer is space! Can you imagine doing this job in 1977?

Finally, let's consider designing a game or application for a smart phone. Studies have shown that if an application does not load within 6 seconds, the user deletes it!

Another major issue many smartphone users experience is limited battery life. But it turns out that the best way to save energy is to design the application to run faster! Thus we have even more incentive to design a fast application!

To Review

In this video, we saw how speed, space, power, and other programming constraints are often complementary. Generally, simply doing less—performing fewer calculations, storing fewer items in memory, uses fewer resources and results in a more efficient algorithm.

It will take some practice to recognize and address the constraints present in your programming problems. Learning to develop algorithms that use resources efficiently and effectively is the true art of programming.