Practical Computer Science

This workshop helps new developers get acquainted with the best parts of computer science.

I intentionally used the phrase "get acquainted with" in the same manner as we "get acquianted" with someone new for the first time. You can't get to know everything there is to know about a person over a single cup of coffee. But it's enough time to *get acquainted*. To think, "I think I could get along with this person. I'd be willing to meet again sometime." And it's also enough time to plan your getaway. To think to yourself, "omg how do I end this boring conversation," excuse yourself to the washroom, and discreetly escape out the back door.

By the end of this workshop, you won't be an expert in every topic in computer science. But you will be *acquainted* with the best parts. You'll have been immersed just enough in the deep parts of the pool to know whether or not you want to go for another swim.

We will start with the best part of computer science (no sense saving the best for last). We start with Chapter 1: *How to Think*. Whether or not you decide to become a serious computer scientist, web designer, or perhaps a sculptor, musician, or politician, there is no better takeaway from the field of computer science than the skill of computational thinking.

How To Think

Let's get right to the best part of computer science: *computational thinking*. This is the name often given to the particular way of reasoning about the world that has punctuated the entire history of computer science. It is a manner of approaching every problem we need to solve, and it is rooted in the long, arduous evolution of the scientific method.

In some ways it would be proper to say that the inventors of computer science came well before

Ada Lovelace and Charles Babbage; we must go back to the beginning of the Enlightenment, when evidence, proof, and logical reasoning came to be understood as the best path of progress. When scientific progress bloomed into the industrial revolution, we inherited not only locomotives and factories, but also a new way of reasoning about the things we are trying to build.

Schools in the US have historically limited their teaching of the scientific method and outcomes-based reasoning skills to the science and mathematics classrooms. The more recent push toward computer science in the early grades will hopefully be transformed into a push toward computational thinking skills, rather than specific programming language requirements. Not everyone wants to grow up to be a software engineer; but everyone should learn how to think; how to investigate claims based on data; how to discern the difference between correlation and causation; and how to detect assumptions and falacies in rhetorical argument.

Computational thinking is, therefore, not limited to the endeavor of computer science or programming, but a skill with a wide range of applicability regardless of vocation. It is therefore, arguable, the *best* part of computer science, and it manifests itself in many different ways. This chapter will highlight the best ways to put computational thinking into practice.

Breaking Things Into Pieces

There's an old joke that goes, "How do you eat an elephant? One piece at a time."

There are a few principles that run through every branch of computer science, but perhaps none are as pervasive or fundamental as the concept of divide-and-conquer. When a large problem can be decomposed into a set of smaller ones, it is already on its way to being solved.

If the set of small problems can, in turn, be divided into even smaller problems, each tiny problem can often be solved trivially. When all of the tiny problems have been solved, the large problem has been solved as well.

Finding the Boundaries Within

However, knowing how to break a problem into pieces can be a problem unto itself. The insight required to split a problem into pieces can be more art than science.

Imagine a very large boulder that has just rolled down a cliff, and is now blocking the road. Moving the boulder would be nearly impossible without very large equipment. But after some experience with large boulders, natural fault lines can quickly be recognized. It is along these lines that the boulder can be split apart; and after repeating this process a few times, the small chunks of leftover

rock are easily removed.

Finding the natural fault lines of an intractable bug or complex feature implementation is an essential practice of applied software engineering. Much of the rest of this chapter, and indeed the rest of this book, is devoted to offering a wealth of strategies and techniques for breaking problems down to size.

Example: Alan Turing's Instruction Tables

At the close of World War II, Alan Turing was itching to build the first manifestation of his imagined "universal computing machine." His unique perspective enabled him to recognize that a machine's ability to follow instructions of a logical sort was far superior to one that was restricted to numbers and brute calculations. Coming from a pencil-and-paper world, he envisioned his machine as reading an "instruction table" for the machine to follow. These tables would describe the microsteps necessary to accomplish a certain task, like adding two numbers together, storing a number in memory, or retrieving a stored value.

But almost as soon as he'd begin sketching out his plans, way before he had a chance to build the machine itself, he realized that developing one giant instruction table was neither practical nor intelligent. Finally, in a flash of insight, his imagination revealed to him that any section of instructions could be made into its own table. If the machine could remember where it left off, it could switch to the another table of instructions when needed, later to resume from where it originally left off.

These subtables provided a way to store a reusable set of instructions which could be called upon as needed.

He took this idea one step further, imagining an entire "heirarchy of tables," with the first "main" table loading the other tables based on conditionals or some other manner. Alan Turing had used pure imagination to invent the concept of *subroutines*.

In one broad stroke, Turing had invented both a practical solution to a complex problem (maintaining long lists of instructions) and a novel way of thinking that has endured for 75 years.

Cause and Effect

One of the first scientific principles that we learn in school is that of *cause and effect*. But we also know this from everyday experience. If I hit a tennis ball with my racquet, the ball will be propelled forward (hopefully!) through the air. If I drink water, I'll be less thirsty. If I put on a coat, I'll be

warmer. If I plug my phone into the wall, it will recharge.

Computer programming is just as predictable, or *deterministic* as people like to say. The computer operates by strictly following a set of instructions, and the instructions are written by us. It cannot make things up. It cannot even ignore the instructions that have been given to it. The computer is completely incapable of improvising on its own. Even so-called "artificial intelligence" still requires the computer to follow a set of predefined rules (see the chapter on Neural Networks for more).

Yet, modern software has become so complex, and consist of so many possible instruction paths, that it may feel non-deterministic. Bugs reported by customers often seem unreproducible. Confidence in our ability to deliver code that works exactly as intended can be shaky at best.

We must take heart: things are not random. Bugs do not happen by chance. Adding that next feature need not be daunting.

We have many tools at our fingertips to help us cope with complexity. The proper tool, wielded with skill, should make the code seem so simple as to immediately reveal the direct cause and effect.

All programming languages are abstractions. The only "real" computer language is the long stream of ones and zeros, which are themselves just representations for certain instructions belonging to a finite set.

When was say that a section of code is "hard to read", we mean that it the connection between the code and its effect is unclear. This is why so many languages exist: there is no single programming language abstraction that will fit everyone's brain in the same way. To some, Java is easy to read and well-written Java code exhibits behaviors that are obvious from its source code. For others, Ruby provides the clear cause-and-effect relationship. For still others, functional languages that do not maintain state seem best.

Whatever your choice, remember that everything you do is just a virtual billiard table: the number of balls and angles of the rails are determined by your progamming language, but the fundamental laws remain the same. What goes up must come down. A line of code must execute, and it must do exactly what it says it will do.

Practical Computer Science: Containers for Things

- Linked Lists
- Arrays
- Queues
- FIFO
- LIFO
- Priority
- Trees
- Binary
- Heaps
- Graphs
- Directed
- Neural Networks

Digital computing was invented in the 1940's, when physical computing devices (such as adding machines and simple calculators) were the state of the art.

Those devices were pre-wired to perform their function. A calculator built to add and subtract integers, which would then transform itself into a machine that could also multiply and divide, would have sounded like some sort of dark magic.

The invention of electronic programmability changed that worldview forever. Computing machiens could be now pre-wired to follow instructions that would be provided later, electronically.

Electronic data storage pioneered the age of software. The days of "unitasker" computing devices were immediately numbered and the realm of computing took a giant leap forward. But soon it became clear that the art of organizing electronic data was not at all obvious.

How should data be organized? Shoud all data be organized in the same way? Must the technique of storing data be connected to the manner in which that data will be used to perform the calculations?

Over 70 years of such questions - and answers - have defined the data structures we use today.

A raw inventory of data structures is not helpful. They can be found in any computer programming textbook worth it's weight in paper (or bandwidth).

But to be able to glimpse the insights that produced such structures in the first place is priceless. The dual dedication to laws of both simplicity and power is one of the best parts of the tradition of computer science.

This chapter aims to illuminate the fundamental structures that we still use today, and to make clear the utmost importance these structures have on modern computational architectures today.

Memory Locations

The most fundamental of all data containers is the *memory location*. This usually refers to the kind of electronic memory that's built into the computer circuitry, as opposed to hard drive storage or flash storage.

A memory location was initially conceived by Alan Turing in his quest to invent a technique for computing devices to temporarily store certain numbers for a few moments, so that they could be retrieved later during a certain step of a process.

Today, modern computers use molecular substrates to reliable store and recall numerical data, but we still imagine them as mail slots, cubby holes, or other familiar physical metaphors.

Each memory location is associated with a specific *address* that uniquely identifies that location. Modern operating systems provide a layer of abstraction on top of the physical hardware by giving each process a "virtual" set of memory addresses, which are mappyed by the operating system to physical addresses. This virtualization allows each process to access memory consistently, even though the underlying system may actually "page" the memory onto other storage devices (like the hard drive) as needed in order to maintain the illusion of multitasking in a shared memory space.

The size of a single memory location is measured in bits. The number of bits that can fit has increased over the years. The personal computing era begin with 8-bit values per location. Modern smartphones and tablets can now store an entire 64-bit value in a single location. (For the remainder of this book, we will assume that a single memory location can store a 64-bit value.)

If we only used computer memory to represent numbers, computers wouldn't be very useful. The breakthrough in computing came with the realization that numbers could represent other things.

At first, certain numerical values were a shorthand to refer to a particular "instruction" which was a hardcoded built-in ability of the computer itself (for example, the ability to add two numbers together).

Later, we extended this idea of interpretation to allow numbers to represent letters of the alphabet, custom instructions, color values, geolocation coordinates, and almost anything else we can imagine.

However, despite the infinite variety of possible interprations for any given memory address, a value

stored in a given memory location is always approached in one of two ways:

1. as data: that is, a numerical value or equivalent interpretation of some real-world entity; or,
2. the address of another memory location

Those are our only two choices. When we choose to interpret a location's value to be the address of another location, we call that value a "pointer," "memory pointer," or "reference." The concept of pointers is one of the most bizarre and intimidating concepts that new programmers face. But it represnets a breakthrough concept in computer science, and i still a fundamental particle in today's programming universe.

Pointers enable complex, heterogeneous structures to emerge from an otherwise linear slate of data buckets.

Some programming languages, like `C`, require the programmer to be quite preoccupied with the tasks of allocating memory locations as needed by the program, and then releasing them back to the operating system when they are no longer needed.

A special syntax enables the programmer to designate certain values as pointers as opposed to scalar values. This burden results in various headaches during programming, but offers speed and ultimate flexibility when writing algorithms.

Other languages, like `Java`, `Ruby`, and `Python`, abstract the gory details of memory allocation away from the programmer, reducing complexity and removing a common source of errors.

However, this convenience comes at the cost of efficiency and a (potentially) non-deterministic "garbage collector" to ensure that memory locations can be reused efficiently.

Linked Lists

If we put together a simple series of memory locations, we can make them work together to form a *linked list*. A linked list is the simplest possible container for a collection of data elements.

A linked list is a linear series of *nodes*. Each node is constructed from a pair of memory locations. The first location is responsible for storing the desired data (a single 64-bit numerical value); the other location is responsible for storing the memory location of the next node.

These memory locations are commonly referred to as the node "data" and "next pointer", respectively.

In order to represent a list, the following strategy is used. One node is arbitrarily designated as the

"head" node. Starting from the head, we attach subsequent nodes by using the "next pointer" as a way to refer to the next node in the list. The ending node is therefore easily identified, as it will store a value of zero for it's "next" pointer.

Modern programming languages offer higher-level abstractions: strings, structures, and objects of arbitrary size and complexity. So why are we talking about a linked list?

Every data structure described in the remainder of this book, and in fact that you will ever use in your software, can be defined in terms of linked lists. It is the fundamental building block of all data structures.

One nice thing about a linked list is that it's pretty easy to maintain a notion of *order*. When adding new items, the "next" pointers can be easily rearranged to "insert" a new item in the midst of the existing chain. Removing items is also simple. Imagine a line of elephants, each connected trunk-to-tail. Removing an elephant simply requires the elephant behind to let go for a moment while the elephant is led away; after which the chain can be reformed.

If you can solve the challenge of building a your own linked list to manage some data that's meaningful - your MP3 collection, for example - then your brain will come to understand the mindset and the patterns that have been most crucial for the endeavor of computer science to advance.

1. a collection of things is made up of smaller, atomic containers
2. data has no intrinsic meaning - it's all about representation and interpretation
3. memory pointers represent the breakthrough concept of_indirection_, a powerful technique to build software artifacts
4. indirection is the core idea of all modern software architecture

Arrays

For all the power provided by linked lists, they suffer from one crucial drawback: speed. We do not create lists of items merely to contain a list of items. The point of a list is do something creative with it. And this is where the linked list structure is less than ideal.

Imagine that we used a linked list to keep track of a list of business contacts. One day, you forget the phone number for your boss, Margaret Smith. Fortunately, you wrote a program to retrieve a contact from the list given their name. Unfortunately, you used a linked list. Your program will need to search for Margaret's node by starting at the head node, and walking the chain one node at a time until it arrives at the right node.

If you're lucky, Margaret's node is stored at the near the front of the list, and it takes only a few hops down the chain to get to the right node.

But if you're unlucky, Margaret's information is further along, or worst case at the very end. The more contacts you add to the list, the greater your chances of having to wait a long time to retrieve any given contact.

This dilemma gave rise to an improvement over the linked list: the *array*. An array is a series of memory locations that guarantees access to any node in the same amount of time.

Even arrays are not panaceas. Access to any random element can now happen in a predictable fashion, but for array sizes are traditionally fixed - you can't grow the array beyond the initial size. Some languages allow you to bypass this requirement, at the cost of additional time and memory if you choose to grow the array after its initial construction.

Still, the advent of arrays made linked lists nearly obsolete for all but the lowest-level programming tasks.

Queues

Now that we've seen linked lists and arrays, a simple twist yields a most useful invention, the *queue*. A queue has the added characteristic that new elements are added at one end of the list, but removed from the other. Consider a line in the grocery store. Customers enter at the back of the line, and are serviced at the front.

A queue does not offer access to any random element, so it is unlike an array. But it is often implemented by using an array and keeping track of the "front" and "back" indices, avoiding the slowness associated with linked lists.

There are actually three common queue flavors, which we will now look at.

FIFO

The *FIFO* queue, short for "First In, First Out", is the kind of queue we typically find in the real world. At the grocery store, boarding an airplane, driving to work - these are all FIFO queues in which the person to get in line first will be the first to be granted their freedom.

Like all queues, only two primary operations exist:

1. you can *push* (i.e. add) a new element onto the end of the queue

2. you can *pop* (i.e. remove) the front element from the queue

LIFO

The *LIFO* queue, or "Last In, First Out" queue, is more commonly called a *Stack*. Stacks are extremely useful containers for many programming tasks, such as implementing calculators and expression evaluation. We refer to a chained series of function calls a *stack* because that's exactly how your favorite programming language keeps track of the execution point in your program. Local variables are generally allocated in memory using a stack structure.

Priority

One of the most useful types of queues is the *priority* queue. In this kind of queue, the *push* and *pop* operations are still supported, but the elements being pushed have some kind of additional characteristic, generally called a priority value or priority level, that serves to modify the position of the item within the queue. Higher-priority items are allowed to "jump" the queue and bypass lower-priority items that were added first.

<<Apollo 11 landing example here>>

Trees

Let's go back to the design of a linked list: each node in the list consists of two parts: one part to store data, and the other to store a pointer.

Let's rename *pointer* to *branch* to help us visualize things a bit better. If we expand our definition of a node to allow more than just one branch, we end up with an entirely new kind of structure called a *tree*. Trees have some amazing properties that open up all kinds of algorithmic possibilities.

Trees are used to implement database systems, solve complex routing problems, store ordered data efficiently, represent complicated heirarchical data, and more.

Binary

We start with the original linked list node and modify it slightly by adding a second branch. Now, each node can point at two neighbors instead of just one. A tree in which each node contains two branches is called a *binary tree*.

By naming the two branches "left" and "right", we have devised a more efficient structure for holding data. Consider the following two structures, one a linked list and the other a tree.

1. How many steps of operation would it require to navigate from the head of the linked list to the node containing the number 10?
2. How many steps of operation would it require to navigate from the root of the linked list to the node containing the number 10?
3. How many steps of operation would it require to insert the number 7 into the linked list?
4. How many steps of operation would it require to insert the number 7 into the binary tree?

It should be clear that using a tree can drastically reduce the number of steps required to store and retrieve data.

Nodes which have no branches are called *leaves*. Leaf nodes end up with some startling properties, which we will examine further in the chapter on algorithms.

Heaps

A *heap* is a tree that follows the specific rule: parent node values are always ordered with respected to their children. If the parent value is always greater than the children, it is called a *max heap*. If the parent value is always less than the children, it is a *min heap*.

Unlike a typical binary tree, it is not possible to traverse all of the nodes of the tree in order. Although parent-child order is guaranteed, *sibling* order is not.

If you can't traverse a heap in absolute sorted order, why use a heap? There are actually several wonderful uses of heaps. Perhaps the most famous is as a very efficient implementation of a priority queue. Highest-priority nodes will sort ot the top (or vice-versa), make it very easy to consume the nodes in priority order. Sibling nodes will have equal priority, and so by definition order doesn't matter.

Therefore, traversing a heap in order is equivalent to popping items from a priority queue; and pushing new elements will update the queue while preserving priority order.

Using a heap is also amongst the fastest ways to sort a collection of data, using the so-called *heapsort algorithm*.

Graphs

When I first encountered the term "graph" in computer programming, I thought it meant the kinds of graphs we learned to draw in school: bar graphs, pie charts, and so on. In computer science, we use the term "graph" to mean something completely different.

Let us return to the concept of the binary tree. Recall that the shape of a binary tree makes searching for a specific element easier than a linear linked list, and it's also easier to keep the container in a particular sorted order.

If we enhance the tree by removing the only-up-to-two branch restriction, and allow each node to have an unlimited number of branches, certain remarkable consequences begin appear. We have now discovered a a new kind of container, the *graph*.

When we transition from a binary tree to a graph, we also shift our terminology: branches are now called *edges*. Edges connect the nodes in the graph. Every node must have at least one edge, otherwise it would become disconnected from the other nodes and no longer be part of the graph.

Graphs can be used to represent city streets, geographic systems, electrical grids, transportation systems, and social networks, to name a few. The graph can represent real-world constructs and phenomena remarkably well.

There are some amazing algorithms that operate on graphs as we've described them, but there are a few important variations that can turn this container into a supercontainer.

Directed

We can do amazing things with our graph if we enhance the edges with one additional aspect, *directionality*. A *directed graph* is one in which each edge is associated with one particular direction. Imagine a city in which every street was a one-way street.

It is also possible to have edges that can allow for movement in both directions, but this is much less common.

When we add the notion of direction to the graph, a number of interesting consequential properties can result:

1. An *directed acyclic graph* is a directed graph in which no "loops" are possible. Once you begin travelling from a particular node, you'll never get back to where you started. Spreadsheets, source code version histories, certain types of electronic circuits, scheduling systems, and garbage collectors are examples of directed acyclic graphs.
2. A *weighted directed graph* is a graph in which each edge is assigned a "weight" or value. Weighted directed graphcs become very useful for real-time traffic analysis, transportation logistics planning, least-cost algorithms, and more.

Neural Networks

Graphs are sometimes called *networks* because they are often used to simulate the behavior of a flow network, whether it's a city's water pipe system, an electronic circuit, or a computer network.

One type of network that's often used as an introduction to the field of artificial intelligence is the *neural network*.