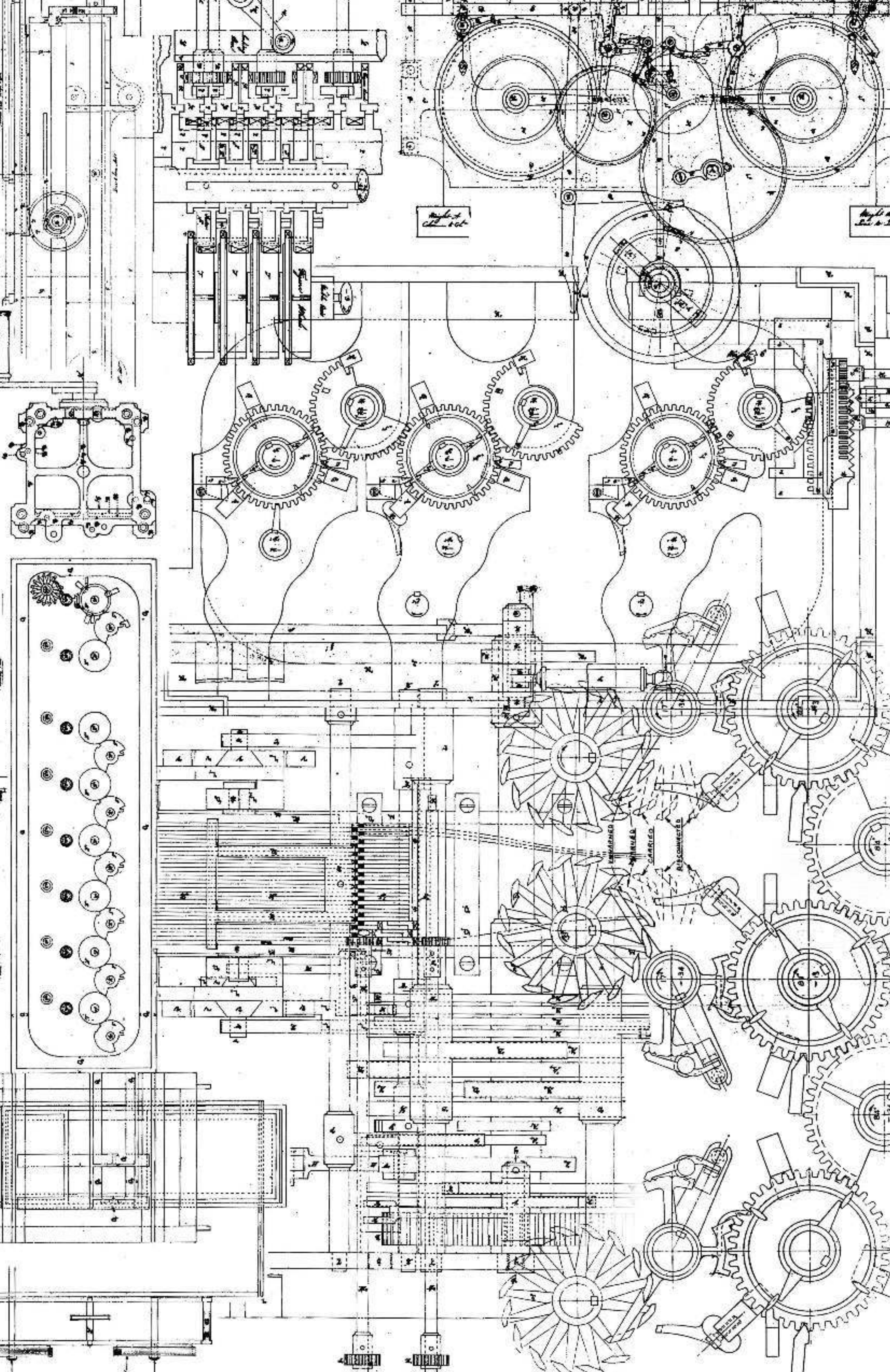
COMPUTER SCIENCE DISTILLED

LEARN THE ART OF SOLVING COMPUTATIONAL PROBLEMS

WLADSTON FERREIRA FILHO

COMPUTER SCIENCE DISTILLED


UNWARNED
WARNED
CARRIED
DISCONNECTED

COMPUTER SCIENCE DISTILLED

LEARN THE ART OF SOLVING COMPUTATIONAL PROBLEMS

Wladston Ferreira Filho

code energy

Las Vegas

Edited by Raimondo Pictet.

Published by Code Energy LLC

hi@code.energy
http://code.energy
http://twitter.com/code_energy
http://facebook.com/code.energy
304 S Jones Blvd # 401 Las Vegas NV 89107 🇺🇸

Publisher's Cataloging-in-Publication Data
Ferreira Filho, Wladston.
Computer science distilled: learn the art of solving computational problems / Wladston Viana Ferreira Filho. — 1st ed.
x, 168 p. : il.
ISBN 978-0-9973160-0-1
eISBN 978-0-9973160-1-8
1. Computer algorithms. 2. Computer programming. 3. Computer science. 4. Data structures (Computer science). I. Title.
004 – dc22 2016909247

First Edition, February 2018.

Friends are the family we choose for ourselves. This book is dedicated to my friends Rômulo, Léo, Moto and Chris, who kept pushing me to "finish the damn book already".

I know that two & two make four—and should be glad to prove it too if I could—though I must say if by any sort of process I could convert 2 & 2 into *five* it would give me much greater pleasure.

—Lord Byron

1813 letter to his future wife Annabella.
Their daughter Ada Lovelace was the first programmer.

Contents

Preface

Everybody in this country should learn to program a computer, because it teaches you how to think.

—Steve Jobs

As computers changed the world with their unprecedented power, a new science flourished: *computer science*. It showed how computers could be used to solve problems. It allowed us to push machines to their full potential. And we achieved crazy, amazing things.

Computer science is everywhere, but it's still taught as boring theory. Many coders never even study it! However, computer science is crucial to effective programming. Some friends of mine simply can't find a good coder to hire. Computing power is abundant, but people who can use it are scarce.

This is my humble attempt to help the world, by pushing *you* to use computers efficiently. This book presents computer science concepts in their plain distilled forms. I will keep academic formalities to a minimum. Hopefully, computer science will stick to your mind and improve your code.

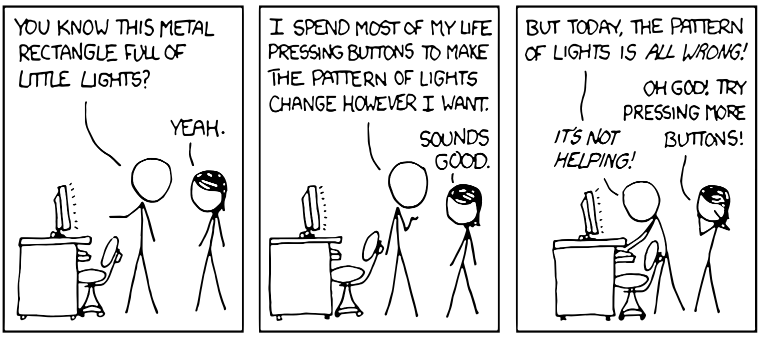

YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS ALL WRONG!
OH GOD! TRY PRESSING MORE BUTTONS!
IT'S NOT HELPING!

Figure 1 "Computer Problems", courtesy of http://xkcd.com.

Is this book for me?

If you want to smash problems with efficient solutions, this book is for you. Little programming experience is required. If you already wrote a few lines of code and recognize basic programming statements like `for` and `while`, you'll be OK. If not, online programming courses[1] cover more than what's required. You can do one in a week, for free. For those who studied computer science, this book is an excellent recap for consolidating your knowledge.

But isn't computer science just for academics?

This book is for everyone. It's about *computational thinking*. You'll learn to change problems into computable systems. You'll use computational thinking on everyday problems. Prefetching and caching will streamline your packing. Parallelism will speed up your cooking. Plus, your code will be awesome. 😉

May the force be with you,
Wlad

[1]http://code.energy/coding-courses.

Chapter 1

Basics

Computer science is not about machines, in the same way that astronomy is not about telescopes. There is an essential unity of mathematics and computer science.

—Edsger Dijkstra

Computers need us to break down problems into chunks they can crunch. To do this, we need some math. Don't panic, it's not rocket science—writing good code rarely calls for complicated equations. This chapter is just a toolbox for problem solving. You'll learn to:

- 💡 Model **ideas** into flowcharts and pseudocode,
- ✓ Know right from wrong with **logic**,
- 💯 **Count** stuff,
- 🎲 Calculate **probabilities** safely.

With this, you will have what it takes to translate your ideas into computable solutions.

1.1 Ideas

When you're on a complex task, keep your brain at the top of its game: dump all important stuff on paper. Our brains' working memory easily overflows with facts and ideas. Writing everything down is part of many organizing methods. There are several ways to do it. We'll first see how flowcharts are used to represent processes. We'll then learn how programmable processes can be drafted in pseudocode. We'll also try and model a simple problem with math.

Flowcharts

When Wikipedians discussed their collaboration process, they created a flowchart that was updated as the debate progressed. Having a picture of what was being proposed helped the discussion:

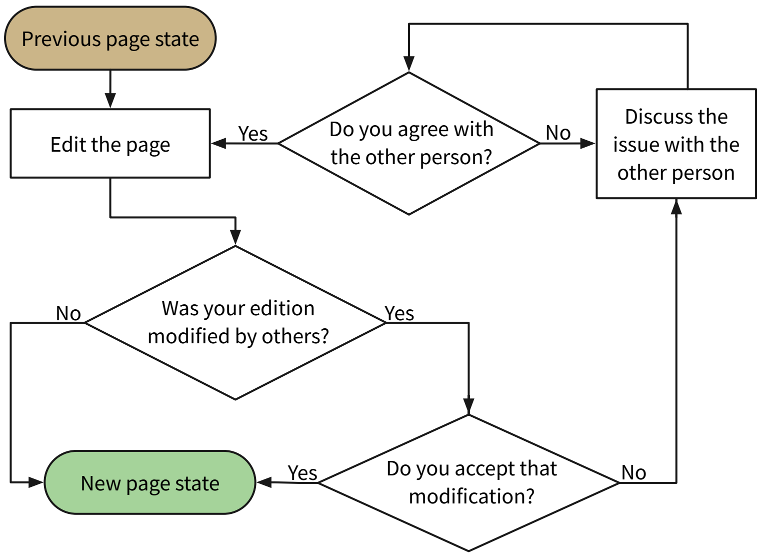

Previous page state
Edit the page
Yes
Do you agree with the other person?
No
Discuss the issue with the other person
No
Was your edition modified by others?
Yes
New page state
Yes
Do you accept that modification?
No

Figure 1.1 Wiki edition process (adapted from http://wikipedia.org).

Like the editing process above, computer code is essentially a process. Programmers often use flowcharts for writing down computing processes. When doing so, you should follow these guidelines[1] for others to understand your flowcharts:

- Write states and instruction steps inside rectangles.
- Write decision steps, where the process may go different ways, inside diamonds.
- Never mix an instruction step with a decision step.
- Connect sequential steps with arrows.
- Mark the start and end of the process.

[1]There's even an ISO standard specifying precisely how software systems diagrams should be drawn, called **UML**: http://code.energy/UML.

Let's see how this works for finding the biggest of three numbers:

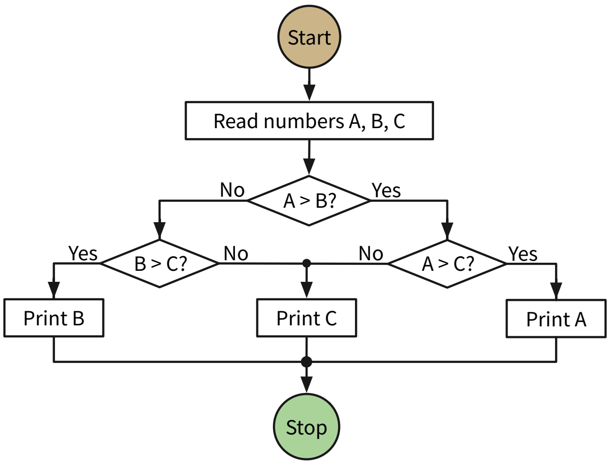


Figure 1.2 Finding the maximum value between three variables.

Pseudocode

Just as flowcharts, **pseudocode** expresses computational processes. Pseudocode is human-friendly code that cannot be understood by a machine. The following example is the same as fig. 1.2. Take a minute and test it out with some sample values of A, B, and C:[2]

```
function maximum(A, B, C)
    if A > B
        if A > C
            max ← A
        else
            max ← C
    else
        if B > C
            max ← B
        else
            max ← C
    print max
```

[2]Here, ← is the assignment operator: `x ← 1` reads *x is set to 1*.

Notice how this example completely disregards the syntactic rules of programming languages? When you write pseudocode, you can even throw in some spoken language! Just as you use flowcharts to compose general mind maps, let your creativity flow free when writing pseudocode (fig. 1.3 😄).

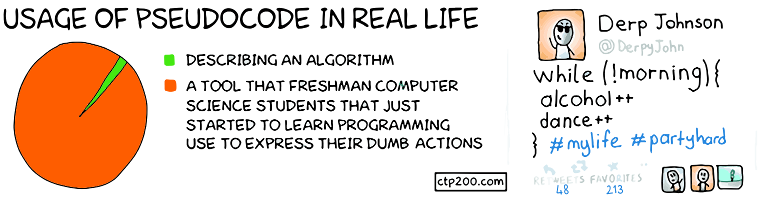

USAGE OF PSEUDOCODE IN REAL LIFE
DESCRIBING AN ALGORITHM
A TOOL THAT FRESHMAN COMPUTER SCIENCE STUDENTS THAT JUST STARTED TO LEARN PROGRAMMING USE TO EXPRESS THEIR DUMB ACTIONS
ctp200.com
Derp Johnson
@DerpyJohn
while (!morning){
alcohol++
dance++
} #mylife #partyhard
RETWEETS 48
FAVORITES 213

Figure 1.3 "Pseudocode in Real Life", courtesy of http://ctp200.com.

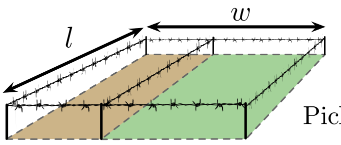
Mathematical Models

A **model** is a set of concepts that represents a problem and its characteristics. It allows us to better reason and operate with the problem. Creating models is so important it's taught in school. High school math is (or should be) about modeling problems into numbers and equations, and applying tools on those to reach a solution.

Mathematically described models have a great advantage: they can be adapted for computers using well established math techniques. If your model has graphs, use graph theory. If it has equations, use algebra. *Stand on the shoulders of giants* who created these tools. It will do the trick. Let's see that in action in a typical high school problem:

> LIVESTOCK FENCE 🐄 Your farm has two types of livestock. You have 100 units of barbed wire to make a rectangular fence for the animals, with a straight division for separating them. How do you frame the fence in order to maximize the pasture's area?

Starting with what's to be determined, w and l are the pasture's dimensions; $w \times l$, the area. Maximizing it means using all the barbed wire, so we relate w and l with 100:



$$A = w \times l,$$
$$100 = 2w + 3l.$$
Pick w and l that maximize the area A.

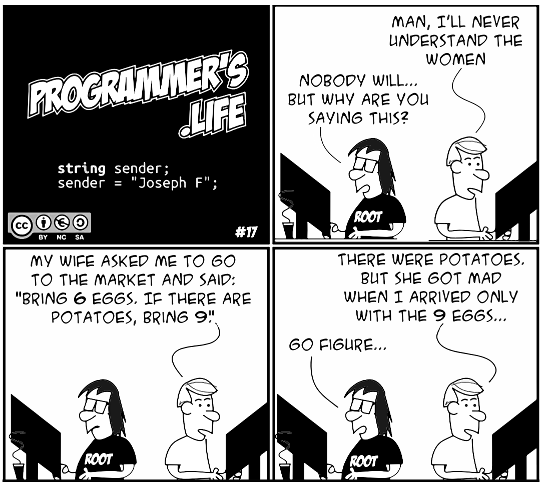
Plugging l from the second equation ($l = \frac{100-2w}{3}$) into the first,

$$A = \frac{100}{3}w - \frac{2}{3}w^2.$$

That's a quadratic equation! Its maximum is easily found with the high school *quadratic formula*. Set $A = 0$, solve the equation, and the maximum is the midway point between the two roots. Quadratic equations are important for you as a pressure cooking pot is valuable to cooks. They save time. Quadratic equations help us solve many problems faster. Remember, your duty is to solve problems. A cook knows his tools, you should know yours. You need mathematical modeling. And you will need logic.

1.2 Logic

Coders work with logic *so much* it messes their minds. Still, many coders don't really learn logic and use it unknowingly. By learning formal logic, we can deliberately use it to solve problems.



PROGRAMMER'S .LIFE
string sender;
sender = "Joseph F";
BY NC SA
#17
MAN, I'LL NEVER UNDERSTAND THE WOMEN
NOBODY WILL... BUT WHY ARE YOU SAYING THIS?
ROOT
MY WIFE ASKED ME TO GO TO THE MARKET AND SAID: "BRING 6 EGGS. IF THERE ARE POTATOES, BRING 9".
THERE WERE POTATOES. BUT SHE GOT MAD WHEN I ARRIVED ONLY WITH THE 9 EGGS...
GO FIGURE...

Figure 1.4 "Programmer's Logic", courtesy of http://programmers.life.

We will start playing around with logical statements using special operators and special algebra. We'll then learn to solve problems with truth tables and see how computers rely on logic.

Operators

In common math, variables and operators ($+$, $\times$, $-$,...) are used to model numerical problems. In mathematical logic, variables and operators represent the validity of things. They don't express numbers, but `True`/`False` values. For instance, the validity of the expression "*if the pool is warm, I'll swim*" is based on the validity of two things, which can be mapped to **logical variables** A and B:

$$A: \text{ The pool is warm.}$$
$$B: \text{ I swim.}$$

They're either `True` or `False`.[3] $A =$ `True` means a warm pool; $B =$ `False` means no swimming. B can't be *half-true,* because I can't half swim. Dependency between variables is expressed with $\rightarrow$, the **conditional operator**. $A \rightarrow B$ is the idea that $A =$ `True` implies $B =$ `True`:

$$A \rightarrow B: \text{ If the pool is warm, then I'll swim.}$$

With more operators, different ideas can be expressed. To negate ideas, we use `!`, the **negation operator**. $!A$ is the opposite of A:

$$!A: \text{ The pool is cold.}$$
$$!B: \text{ I don't swim.}$$

The Contrapositive Given $A \rightarrow B$ and I didn't swim, what can be said about the pool? A warm pool *forces* the swimming, so without swimming, it's impossible for the pool to be warm. Every conditional expression has a **contrapositive** equivalent:

$$\text{for any two variables } A \text{ and } B,$$
$$A \rightarrow B \text{ is the same as } !B \rightarrow !A.$$

[3]Values can be in between in fuzzy logic, but it won't be covered in this book.

Another example: *if you can't write good code, you haven't read this book*. Its contrapositive is *if you read this book, you can write good code*. Both sentences say the same in different ways.[4]

The Biconditional Be careful, saying *"if the pool is warm, I'll swim"* doesn't mean I'll only swim in warm water. The statement promises nothing about cold pools. In other words, $A \rightarrow B$ doesn't mean $B \rightarrow A$. To express both conditionals, use the **biconditional**:

$A \leftrightarrow B$: I'll swim if and only if the pool is warm.

Here, the pool being warm is equivalent to me swimming: knowing about the pool means knowing if I'll swim *and vice-versa*. Again, beware of the **inverse error**: never presume $B \rightarrow A$ follows from $A \rightarrow B$.

And, Or, Exclusive Or These logical operators are the most famous, as they're often explicitly coded. `AND` expresses all ideas are `True`; `OR` expresses any idea is `True`; `XOR` expresses ideas are of opposing truths. Imagine a party serving vodka and wine:

A : You drank wine. 🍷
B : You drank vodka. 🍸
A `OR` B : You drank. 🎉
A `AND` B : You drank mixing drinks. 😰
A `XOR` B : You drank without mixing. 😇

Make sure you understand how the operators we've seen so far work. The following table recaps all possible combinations for two variables. Notice how $A \rightarrow B$ is equivalent to `!`A `OR` B, and A `XOR` B is equivalent to `!`$(A \leftrightarrow B)$.

[4]And by the way, 🤓 they're both *actually* true.

Table 1.1 Logical operations for 4 possible values of A and B.

A	B	$!A$	$A \to B$	$A \leftrightarrow B$	A AND B	A OR B	A XOR B
✓	✓	✗	✓	✓	✓	✓	✗
✓	✗	✗	✗	✗	✗	✓	✓
✗	✓	✓	✓	✗	✗	✓	✓
✗	✗	✓	✓	✓	✗	✗	✗

Boolean Algebra

As elementary algebra simplifies numerical expressions, **boolean algebra**[5] simplifies logical expressions.

Associativity Parentheses are irrelevant for sequences of AND or OR operations. As sequences of sums or multiplications in elementary algebra, they can be calculated in any order.

$$A \text{ AND } (B \text{ AND } C) = (A \text{ AND } B) \text{ AND } C.$$
$$A \text{ OR } (B \text{ OR } C) = (A \text{ OR } B) \text{ OR } C.$$

Distributivity In elementary algebra we factor multiplicative terms from sums: $a \times (b + c) = (a \times b) + (a \times c)$. Likewise in logic, ANDing after an OR is equivalent to ORing results of ANDs, and vice versa:

$$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C).$$
$$A \text{ OR } (B \text{ AND } C) = (A \text{ OR } B) \text{ AND } (A \text{ OR } C).$$

DeMorgan's Law[6] It can't be summer *and* winter at once, so it's either *not* summer *or not* winter. And it's not summer and not winter *if and only if* it's *not* the case it's either summer *or* winter. Following this reasoning, ANDs can be transformed into ORs and vice versa:

[5] After George Boole. His 1854 book joined logic and math, starting all this.

[6] De Morgan was friends with Boole. He tutored the young Ada Lovelace, who became the first programmer a century before the first computer was constructed.

$$\texttt{!}(A \texttt{ AND } B) = \texttt{!}A \texttt{ OR } \texttt{!}B,$$
$$\texttt{!}A \texttt{ AND } \texttt{!}B = \texttt{!}(A \texttt{ OR } B).$$

These rules transform logical models, reveal properties, and simplify expressions. Let's solve a problem:

> Hot Server 💥 A server crashes if it's overheating while the air conditioning is off. It also crashes if it's overheating and its chassis cooler fails. In which conditions does the server work?

Modeling it in logical variables, the conditions for the server to crash can be stated in a single expression:

A :	Server overheats.
B :	Air conditioning off.
C :	Chassis cooler fails.
D :	Server crashes.

$$(A \texttt{ AND } B) \texttt{ OR } (A \texttt{ AND } C) \to D.$$

Using distributivity, we factorize the expression:

$$A \texttt{ AND } (B \texttt{ OR } C) \to D.$$

The server works when ($\texttt{!}D$). The contrapositive reads:

$$\texttt{!}D \to \texttt{!}(A \texttt{ AND } (B \texttt{ OR } C)).$$

We use DeMorgan's Law to remove parentheses:

$$\texttt{!}D \to \texttt{!}A \texttt{ OR } \texttt{!}(B \texttt{ OR } C).$$

Applying DeMorgan's Law again,

$$\texttt{!}D \to \texttt{!}A \texttt{ OR } (\texttt{!}B \texttt{ AND } \texttt{!}C).$$

This expression tells us that whenever the server works, either $\texttt{!}A$ (it's not overheating), or $\texttt{!}B \texttt{ AND } \texttt{!}C$ (both air conditioning *and* chassis cooler are working).

Truth Tables

Another way to analyze logical models is checking what happens in all possible configurations of its variables. A **truth table** has a column for each variable. Rows represent possible combinations of variable states.

One variable requires two rows: in one the variable is set `True`, in the other `False`. To add a variable, we duplicate the rows. We set the new variable `True` in the original rows, and `False` in the duplicated rows (fig. 1.5). The truth table size doubles for each added variable, so it can only be constructed for a few variables.[7]

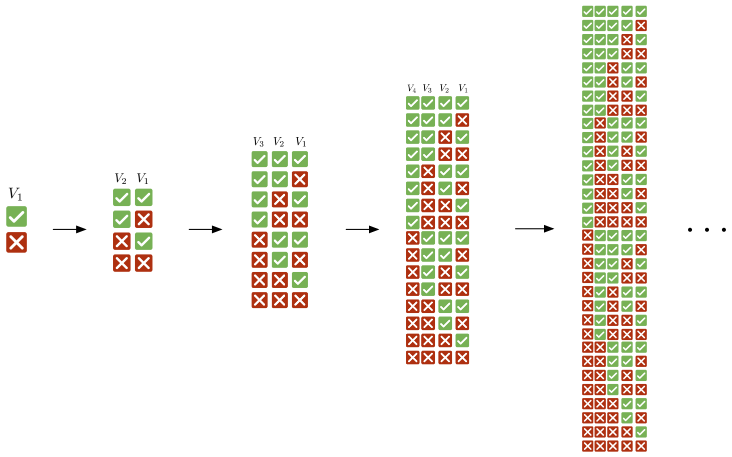


Figure 1.5 Tables listing the configurations of 1–5 logical variables.

Let's see how a truth table can be used to analyze a problem.

Fragile System We have to create a database system with the following requirements:

I : If the database is locked, we can save data.
II : A database lock on a full write queue cannot happen.
III : Either the write queue is full, or the cache is loaded.
IV : If the cache is loaded, the database cannot be locked.

Is this possible? Under which conditions will it work?

[7] A truth table for 30 variables would have more than a billion rows. 😱

First we transform each requirement into a logical expression. This database system can be modeled using four variables:

A :	Database is locked.	I :	$A \to B$.
B :	Able to save data.	II :	`!`$($$A$ `AND` $C$$)$.
C :	Write queue is full.	III :	C `OR` D.
D :	Cache is loaded.	IV :	$D \to$ `!`A.

We then create a truth table with all possible configurations. Extra columns are added to check the requirements.

Table 1.2 Truth table for exploring the validity of four expressions.

State #	A	B	C	D	I	II	III	IV	All four
1	✗	✗	✗	✗	✓	✓	✗	✓	✗
2	✗	✗	✗	✓	✓	✓	✓	✓	✓
3	✗	✗	✓	✗	✓	✓	✓	✓	✓
4	✗	✗	✓	✓	✓	✓	✓	✓	✓
5	✗	✓	✗	✗	✓	✓	✗	✓	✗
6	✗	✓	✗	✓	✓	✓	✓	✓	✓
7	✗	✓	✓	✗	✓	✓	✓	✓	✓
8	✗	✓	✓	✓	✓	✓	✓	✓	✓
9	✓	✗	✗	✗	✗	✓	✗	✓	✗
10	✓	✗	✗	✓	✗	✓	✓	✗	✗
11	✓	✗	✓	✗	✗	✗	✓	✓	✗
12	✓	✗	✓	✓	✗	✗	✓	✗	✗
13	✓	✓	✗	✗	✓	✓	✗	✓	✗
14	✓	✓	✗	✓	✓	✓	✓	✗	✗
15	✓	✓	✓	✗	✓	✗	✓	✓	✗
16	✓	✓	✓	✓	✓	✗	✓	✗	✗

All requirements are met in states 2–4 and 6–8. In these states, A = `False`, meaning the database can't ever be locked. Notice the cache will not be loaded only in states 3 and 7.

To test what you've learned, solve the Zebra Puzzle.[8] It's a famous logic problem wrongly attributed to Einstein. They say only 2% of people can solve it, but I doubt that. Using a big truth table and correctly simplifying and combining logic statements, I'm sure you'll crack it.

Whenever you're dealing with things that assume one of two possibilities, remember they can be modeled as logic variables. This way, it's easy to derive expressions, simplify them, and draw conclusions. Let's now see the most impressive application of logic: the design of electronic computers.

Logic in Computing

Groups of logical variables can represent numbers in binary form.[9] Logic operations on binary digits can be combined to perform general calculations. **Logic gates** perform logic operations on electric current. They are used in electrical circuits that can perform calculations at very high speeds.

A logic gate receives values through input wires, performs its operation, and places the result on its output wire. There are `AND` gates, `OR` gates, `XOR` gates, and more. `True` and `False` are represented by electric currents with high or low voltage. Using gates, complex logical expressions can be computed near instantly. For example, this electrical circuit sums two numbers:

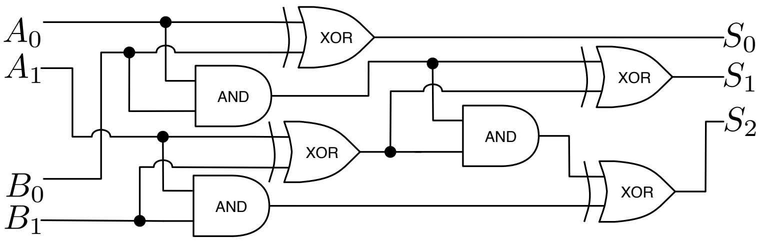


Figure 1.6 A circuit to sum 2-bit numbers given by pairs of logical variables (A_1A_0 and B_1B_0) into a 3-bit number ($S_2S_1S_0$).

[8]http://code.energy/zebra-puzzle.

[9]`True` $= 1$, `False` $= 0$. If you have no idea why `101` in binary represents the number 5, check Appendix I for an explanation of number systems.

Let's see how this circuit works. Take a minute to follow the operations performed by the circuit to realize how the magic happens:

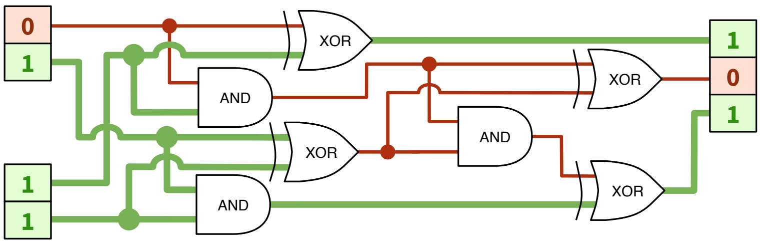


Figure 1.7 Calculating $2 + 3 = 5$ (in binary, `10` + `11` = `101`).

To take advantage of this fast form of computing, we transform numerical problems to their binary/logical form. Truth tables help model and test circuits. Boolean algebra simplifies expressions and thus simplifies circuits.

At first, gates were made with bulky, inefficient and expensive electrical valves. Once valves were replaced with transistors, logic gates could be produced en masse. And we kept discovering ways to make transistors smaller and smaller.[10] The working principles of the modern CPU are still based on boolean algebra. A modern CPU is just a circuit of millions of microscopic wires and logic gates that manipulate electric currents of information.

1.3 Counting

It's important to count things correctly—you'll have to do it many times when working with computational problems.[11] The math in this section will be more complex, but don't be scared. Some people think they can't be good coders because they think they're bad at math. Well, I failed high school math 😞, yet here I am 😅. The math that makes a good coder is not what's required in typical math exams from schools.

[10] In 2016, researchers created working transistors on a 1 nm scale. For reference, a gold *atom* is 0.15 nm wide.

[11] **Counting** and **Logic** belong to an important field to computer science called **Discrete Mathematics**.

Outside school, formulas and step-by-step procedures aren't memorized. They are looked up on the Internet when needed. Calculations mustn't be in pen and paper. What a good coder requires is intuition. Learning about counting problems will strengthen that intuition. Let's now grind through a bunch of tools step by step: multiplications, permutations, combinations and sums.

Multiplying

If an event happens in n different ways, and another event happens in m different ways, the number of different ways both events can happen is $n \times m$. For example:

> Cracking the Code 🔐 A PIN code is composed of two digits and a letter. It takes one second to try a PIN. In the worst case, how much time do we need to crack a PIN?

Two digits can be chosen in 100 ways (00-99) and a letter in 26 ways (A-Z). Therefore, there are $100 \times 26 = 2,600$ possible PINs. In the worst case, we have to try every single PIN until we find the right one. After 2,600 seconds (43 minutes), we'll have cracked it.

> Team Building 👥 There are 23 candidates who want to join your team. For each candidate, you toss a coin and only hire if it shows heads. How many team configurations are possible?

Before hiring, the only possible team configuration is you alone. Each coin toss then doubles the number of possible configurations. This has to be done 23 times, so we compute 2 *to the power 23*:

$$\underbrace{2 \times 2 \times \cdots \times 2}_{\text{23 times}} = 2^{23} = 8,388,608 \text{ team configurations.}$$

Note that one of these configurations is still you alone.

Permutations

If we have n items, we can order them in n factorial ($n!$) different ways. The factorial is explosive, it gets to enormous numbers for small values of n. If you are not familiar,

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1.$$

It's easy to see $n!$ is the number of ways n items can be ordered. In how many ways can you choose a first item among n? After the first item was chosen, in how many ways can you choose a second one? Afterwards, how many options are left for a third? Think about it, then we'll move on to more examples.[12]

> TRAVELING SALESMAN 🚚 Your truck company delivers to 15 cities. You want to know in what order to serve these cities to minimize gas consumption. If it takes a microsecond to calculate the length of one route, how long does it take to compute the length of all possible routes?

Each permutation of the 15 cities is a different route. The factorial is the number of distinct permutations, so there are $15! = 15 \times 14 \times \cdots \times 1 \approx 1.3$ trillion routes. That in microseconds is roughly equivalent to 15 days. If instead you had 20 cities, it would take *77 thousand years*.

> THE PRECIOUS TUNE 🎼 A musician is studying a scale with 13 different notes. She wants you to render all possible melodies that use six notes only. Each note should play once per melody, and each six-note melody should play for one second. How much audio runtime is she asking for?

We want to count permutations of six out of the 13 notes. To ignore permutations of unused notes, we must stop developing the factorial after the sixth factor. Formally, $n!/(n-m)!$ is the number of possible permutations of m out of n possible items. In our case:

[12] By convention, $0! = 1$. We say there's one way to order zero items.

$$\begin{aligned}\frac{13!}{(13-6)!} &= \frac{13\times 12\times 11\times 10\times 9\times 8\times 7!}{7!}\\ &= \underbrace{13\times 12\times 11\times 10\times 9\times 8}_{\text{6 factors}}\\ &= 1{,}235{,}520 \text{ melodies.}\end{aligned}$$

That's over 1.2 million one-second melodies—it would take 343 hours to listen to everything. Better convince the musician to find the perfect melody some other way.

Permutations with Identical Items

The factorial $n!$ overcounts the number of ways to order n items if some are identical. Identical items swapping their positions shouldn't count as a different permutation.

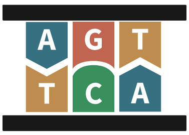
In a sequence of n items of which r are identical, there are $r!$ ways to reorder identical items. Thus, $n!$ counts each distinct permutation $r!$ times. To get the number of distinct permutations, we need to divide $n!$ by this overcount factor. For instance, the number of distinct permutations of the letters "`CODE ENERGY`" is $10!/3!$.

Playing with DNA A biologist is studying a DNA segment related to a genetic disease. The segment is made of 23 base pairs, where 9 must be `A-T`, 14 must be `G-C`. She wants to run a simulation task on every possible DNA segment having these numbers of base pairs. How many simulation tasks is she looking at?

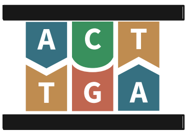
First we calculate all possible permutations of the 23 base pairs. Then we divide the result to account for the 9 repeated `A-T` and the 14 repeated `G-C` base pairs:

$$23!/(9!\times 14!) = 817{,}190 \text{ base pair permutations.}$$

But the problem isn't over. Considering orientation of base pairs:



isn't the same as



For each sequence of 23 base pairs, there are 2^{23} distinct orientation configurations. Therefore, the total is:

$$817,190 \times 2^{23} \approx 7 \text{ trillion sequences.}$$

And that's for a tiny 23 base pair sequence with a known distribution. The smallest replicable DNA known so far are from the minuscule *Porcine circovirus*, and it has 1,800 base pairs! DNA code and life are truly amazing from a technological point of view. It's crazy: human DNA has about 3 billion base pairs, replicated in each of the 3 trillion cells of the human body.

Combinations

Picture a deck of 13 cards containing all ♠ spades. How many ways can you deal six cards to your opponent? We've seen $13!/(13-6)!$ is the number of permutations of six out of 13 possible items. Since the order of the six cards doesn't matter, we must divide this by $6!$ to obtain:

$$\frac{13!}{6!(13-6)!} = 1,716 \text{ combinations.}$$

The binomial $\binom{n}{m}$ is the number of ways to select m items out of a set of n items, regardless of order:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}.$$

The binomial is read "n choose m".

Chess Queens ♛ You have an empty chessboard and 8 queens, which can be placed anywhere on the board. In how many different ways can the queens be placed?

The chessboard has 64 squares in an 8×8 grid. The number of ways to choose 8 squares out of the available 64 is $\binom{64}{8} \approx 4.4$ billion.[13]

Sums

Calculating sums of sequences occurs often when counting. Sequential sums are expressed using the **capital-sigma** (Σ) notation. It indicates how an expression will be summed for each value of i:

$$\sum_{\text{start } i}^{\text{finish } i} \text{expression of } i.$$

For instance, summing the first five odd numbers is written:

$$\sum_{i=0}^{4}(2i+1) = 1+3+5+7+9.$$

Note i was replaced by each number between 0 and 4 to obtain 1, 3, 5, 7 and 9. Summing the first n natural numbers is thus:

$$\sum_{i=1}^{n} i = 1+2+\cdots+(n-1)+n.$$

When the prodigious mathematician Gauss was ten years old, he got tired of summing natural numbers one by one and found this neat trick:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Can you guess how Gauss discovered this? The trick is explained in Appendix II. Let's see how we can use it to solve a problem:

> FLYING CHEAP ✈️ You need to fly to New York City anytime in the next 30 days. Air ticket prices change unpredictably according to the departure *and* return dates. How many pairs of days must be checked to find the cheapest tickets for flying to NYC and back within the next 30 days?

[13]Pro tip: Google `64 choose 8` for the result.

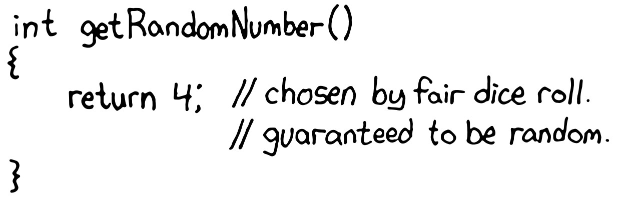
Any pair of days between today (day 1) and the last day (day 30) is valid, as long as the return is the same day of later than the departure. Hence, 30 pairs begin with day 1, 29 pairs begin with day 2, 28 with day 3, and so on. There's only one pair that begins on day 30. So 30+29+...+2+1 is the total number of pairs that needs to be considered. We can write this $\sum_{i=1}^{30} i$ and use our handy formula:

$$\sum_{i=1}^{30} i = \frac{30(30+1)}{2} = 465 \text{ pairs.}$$

Also, we can solve this using combinations. From the 30 days available, pick two. The order doesn't matter: the earlier day is the departure, the later day is the return. This gives $\binom{30}{2} = 435$. But wait! We must count the cases where arrival and departure are the same date. There are 30 such cases, thus $\binom{30}{2} + 30 = 465$.

1.4 Probability

The principles of randomness will help you understand gambling, forecast the weather, or design a backup system with low risk of failure. The principles are simple, yet misunderstood by most people.



```
int getRandomNumber()
{
        return 4;  // chosen by fair dice roll.
                   // guaranteed to be random.
}
```

Figure 1.8 "Random number", courtesy of http://xkcd.com.

Let's start using our counting skills to compute odds. Then we'll learn how different event types are used to solve problems. Finally, we'll see why gamblers lose everything.

Counting Outcomes

A die roll has six possible outcomes: ⚀, ⚁, ⚂, ⚃, ⚄ and ⚅. The chances of getting ⚃ are thus 1/6. How about getting an odd number? It can happen in three ways (⚀, ⚂ or ⚄), so the chances are $3/6 = 1/2$. Formally, the **probability** of an event to occur is:

$$P(\text{event}) = \frac{\text{\# of ways event can happen}}{\text{\# of possible outcomes}}.$$

This works because each possible outcome is equally likely to happen: the die is well balanced and the thrower isn't cheating.

> TEAM BUILDING, AGAIN 👥 There are 23 candidates who want to join your team. For each candidate, you toss a coin and only hire if it shows heads. What are the chances of hiring nobody?

We've seen there are $2^{23} = 8,388,608$ possible team configurations. The only way to hire nobody is by tossing 23 consecutive tails. The probability of that happening is thus $P(\text{nobody}) = 1/8,388,608$. To put things into perspective, the probability that a given commercial airline flight crashes is about one in five million.

Independent Events

If you toss a coin and roll a die, the chance of getting heads and ⚅ is $1/2 \times 1/6 = 1/12 \approx 0.08$, or 8%. When the outcome of an event does not influence the outcome of another event, they are **independent**. The probability that two independent events will happen is the product of their individual probabilities.

> BACKING UP 💾 You need to store data for a year. One disk has a probability of failing of one in a billion. Another disk costs 20% the price but has a probability of failing of one in two thousand. What should you buy?

If you use three cheap disks, you only lose the data if all three disks fail. The probability of that happening is $(1/2,000)^3 = 1/8,000,000,000$. This redundancy achieves a lower risk of data loss than the expensive disk, while costing only 60% the price.

Mutually Exclusive Events

A die roll cannot simultaneously yield ⚃ and an odd number. The probability to get either ⚃ or an odd number is thus $1/6 + 1/2 = 2/3$. When two events cannot happen simultaneously, they are **mutually exclusive**. If you need any of the mutually exclusive events to happen, just sum their individual probabilities.

> SUBSCRIPTION CHOICE ☑ Your website offers three plans: free, basic, or pro. You know a random new customer has a probability of 70% of choosing the free plan, 20% for the basic, and 10% for the pro. What are the chances a new user will sign up for a paying plan?

The events are mutually exclusive: a user can't choose *both* the basic and pro plans at the same time. The probability the user will pay is $0.2 + 0.1 = 0.3$.

Complementary Events

A die roll cannot simultaneously yield a multiple of three (⚂, ⚅) and a number *not* divisible by three, but it must yield one of them. The probability to get a multiple of three is $2/6 = 1/3$, so the probability to get a number *not* divisible by three is $1 - 1/3 = 2/3$. When two mutually exclusive events cover all possible outcomes, they are **complementary**. The sum of individual probabilities of complementary events is thus 100%.

> TOWER DEFENSE GAME 🏰 Your castle is defended by five towers. Each tower has a 20% probability of disabling an invader before he reaches the gate. What are the chances of stopping him?

There's $0.2 + 0.2 + 0.2 + 0.2 + 0.2 = 1$, or a 100% chance of hitting the enemy, right? *Wrong!* Never sum the probabilities of independent events, that's a common mistake. Use complementary events twice:

- The 20% chance of hitting is complementary to the 80% chance of missing. The probability that all towers miss is: $0.8^5 \approx 0.33$.
- The event "all towers miss" is complementary to "at least one tower hits". The probability of stopping the enemy is: $1 - 0.33 = 0.67$.

The Gambler's Fallacy

If you flip a normal coin ten times, and you get ten heads, then on the 11th flip, are you more likely to get a tail? Or, by playing the lottery with the numbers 1 to 6, are you less likely to win than playing with more evenly spaced numbers?

Don't be a victim of the gambler's fallacy. Past events never affect the outcome of an independent event. Never. *Ever*. In a truly random lottery drawing, the chances of any specific numbers being chosen is the same as any other. There's no "hidden law" that forces numbers that weren't frequently chosen in the past to be chosen more often in the future.

Advanced Probabilities

There's far more to probability than we can cover here. Always remember to look for more tools when tackling complex problems. For example:

> TEAM BUILDING, AGAIN AND AGAIN There are 23 candidates who want to join your team. For each candidate, you toss a coin and only hire if it shows heads. What are the chances of hiring seven people or less?

Yes, this is hard. Googling around will eventually lead you to the "binomial distribution". You can visualize this on Wolfram Alpha[14] by typing: `B(23,1/2) <= 7`.

[14]http://wolframalpha.com.

Conclusion

In this chapter, we've seen things that are intimately related to problem solving, but do not involve any actual coding. Section 1.1 explains why and how we should write things down. We create models for our problems, and use conceptual tools on the models we create. Section 1.2 provides a toolbox for handling logic, with boolean algebra and truth tables.

Section 1.3 shows the importance of counting possibilities and configurations of various problems. A quick back-of-the-envelope calculation can show you if a computation will be straightforward or fruitless. Novice programmers often lose time analyzing way too many scenarios. Finally, section 1.4 explains the basic rules of counting odds. Probability is very useful when developing solutions that must interact with our wonderful but uncertain world.

With this, we've outlined many important aspects of what academics call *Discrete Mathematics*. Many more fun theorems can be picked up from the references below or navigating Wikipedia. For instance, you can use the "Pigeonhole Principle" to prove at least two people in New York City have exactly the same number of hairs!

Some of what we learned here will be especially relevant in the next chapter, where we'll discover perhaps the most important aspect of computer science.

Reference

- Discrete Mathematics and its Applications, 7th Edition
  - Get it at https://code.energy/rosen
- Prof. Jeannette Wing's slides on computational thinking
  - Get it at https://code.energy/wing

CHAPTER 2

Complexity

In almost every computation, a variety of arrangements for the processes is possible. It is essential to choose that arrangement which shall tend to minimize the time necessary for the calculation.

—ADA LOVELACE

HOW MUCH TIME does it take to sort 26 shuffled cards? If instead you had 52 cards, would it take twice as long? How much longer would it take for a thousand decks of cards? The answer is intrinsic to the **method** used to sort the cards.

A method is a list of unambiguous instructions for achieving a goal. A method that always requires a finite series of operations is called an **algorithm**. For instance, a card-sorting algorithm is a method that will always specify some operations to sort a deck of 26 cards per suit and per rank.

Less operations need less computing power. We like fast solutions, so we monitor the number of operations in our algorithms. Many algorithms require a fast-growing number of operations when the input grows in size. For example, our card-sorting algorithm could take few operations to sort 26 cards, but four times as much operations to sort 52 cards!

To avoid bad surprises when our problem size grows, we find the algorithm's **time complexity**. In this chapter, you'll learn to:

- ⏱ Count and interpret **time** complexities,
- 📈 Express their growth with fancy **Big-O**'s,
- 😱 Run away from **exponential** algorithms,
- 💾 Make sure you have enough computer **memory**.

But first, how do we define time complexity?

Time complexity is written $\mathbb{T}(n)$. It gives the number of operations the algorithm performs when processing an input of size n. We also refer to an algorithm's $\mathbb{T}(n)$ as its **running cost**. If our card-sorting algorithm follows $\mathbb{T}(n) = n^2$, we can predict how much longer it takes to sort a deck once we double its size: $\frac{\mathbb{T}(2n)}{\mathbb{T}(n)} = 4$.

Hope for the best, prepare for the worst

Isn't it faster to sort a pile of cards that's almost sorted already? Input size isn't the only characteristic that impacts the number of operations required by an algorithm. When an algorithm can have different values of $\mathbb{T}(n)$ for the same value of n, we resort to cases:

- Best Case: when the input requires the minimum number of operations for any input of that size. In sorting, it happens when the input is already sorted.
- Worst Case: when the input requires the maximum number of operations for any input of that size. In many sorting algorithms, that's when the input was given in reverse order.
- Average Case: refers to the average number of operations required for typical inputs of that size. For sorting, an input in random order is usually considered.

In general, the most important is the worst case. From there, you get a guaranteed baseline you can always count on. When nothing is said about the scenario, the worst case is assumed. Next, we'll see how to analyze a worst case scenario, hands on.

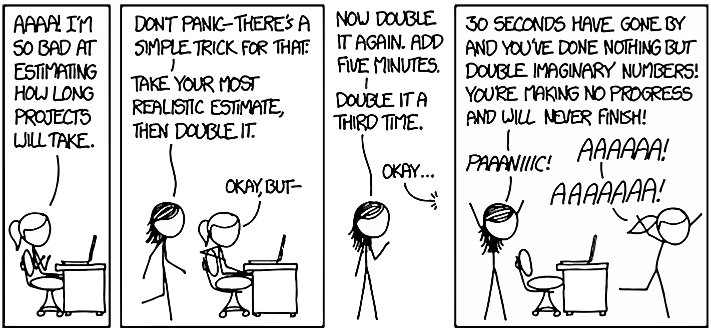

AAAA! I'M SO BAD AT ESTIMATING HOW LONG PROJECTS WILL TAKE.
DON'T PANIC–THERE'S A SIMPLE TRICK FOR THAT:
TAKE YOUR MOST REALISTIC ESTIMATE, THEN DOUBLE IT.
OKAY, BUT–
NOW DOUBLE IT AGAIN. ADD FIVE MINUTES.
DOUBLE IT A THIRD TIME.
OKAY...
30 SECONDS HAVE GONE BY AND YOU'VE DONE NOTHING BUT DOUBLE IMAGINARY NUMBERS! YOU'RE MAKING NO PROGRESS AND WILL NEVER FINISH!
PAAANIIIC!
AAAAAA!
AAAAAAA!

Figure 2.1 "Estimating Time", courtesy of http://xkcd.com.

2.1 Counting Time

We find the time complexity of an algorithm by counting the number of basic operations it requires for a hypothetical input of size n. We'll demonstrate it with **Selection Sort**, a sorting algorithm that uses a nested loop. An outer `for` loop updates the current position being sorted, and an inner `for` loop selects the item that goes in the current position:[1]

```
function selection_sort(list)
    for current ← 1 … list.length - 1
        smallest ← current
        for i ← current + 1 … list.length
            if list[i] < list[smallest]
                smallest ← i
        list.swap_items(current, smallest)
```

Let's see what happens with a list of n items, assuming the worst case. The outer loop runs $n-1$ times and does two operations per run (one assignment and one swap) totaling $2n-2$ operations. The inner loop first runs $n-1$ times, then $n-2$ times, $n-3$ times, and so on. We know how to sum these types of sequences:[2]

$$\text{\# of inner loop runs} = \overbrace{\underbrace{n-1}_{1^{\text{st}}\text{ pass of outer loop}} + \underbrace{n-2}_{2^{\text{nd}}\text{ pass of outer loop}} + \cdots + 2 + 1}^{n-1 \text{ total runs of the outer loop.}}$$

$$= \sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2-n}{2}.$$

In the worst case, the `if` condition is always met. This means the inner loop does one comparison and one assignment $(n^2-n)/2$ times, hence n^2-n operations. In total, the algorithm costs $2n-2$ operations for the outer loop, plus n^2-n operations for the inner loop. We thus get the time complexity:

$$\mathbb{T}(n) = n^2 + n - 2.$$

[1] To understand a new algorithm, run it on paper with a small sample input.
[2] From sec. 1.3, $\sum_{i=1}^{n} i = n(n+1)/2$.

Now what? If our list size was $n = 8$ and we double it, the sorting time will be multiplied by:

$$\frac{\mathbb{T}(16)}{\mathbb{T}(8)} = \frac{16^2 + 16 - 2}{8^2 + 8 - 2} \approx 3.86.$$

If we double it again we will multiply time by 3.90. Double it over and over and find 3.94, 3.97, 3.98. Notice how this gets closer and closer to 4? This means it would take four times as long to sort two million items than to sort one million items.

Understanding Growth

Say the input size of an algorithm is very large, and we increase it even more. To predict how the execution time will grow, we don't need to know all terms of $\mathbb{T}(n)$. We can approximate $\mathbb{T}(n)$ by its fastest-growing term, called the **dominant term**.

> INDEX CARDS 🗃️ Yesterday, you knocked over one box of index cards. It took you two hours of Selection Sort to fix it. Today, you spilled ten boxes. How much time will you need to arrange the cards back in?

We've seen Selection Sort follows $\mathbb{T}(n) = n^2 + n - 2$. The fastest-growing term is n^2, therefore we can write $\mathbb{T}(n) \approx n^2$. Assuming there are n cards per box, we find:

$$\frac{\mathbb{T}(10n)}{\mathbb{T}(n)} \approx \frac{(10n)^2}{n^2} = 100.$$

It will take you approximately $100 \times (2 \text{ hours}) = 200$ hours! What if we had used a different sorting method? For example, there's one called "Bubble Sort" whose time complexity is $\mathbb{T}(n) = 0.5n^2 + 0.5n$. The fastest-growing term then gives $\mathbb{T}(n) \approx 0.5n^2$, hence:

$$\frac{\mathbb{T}(10n)}{\mathbb{T}(n)} \approx \frac{0.5 \times (10n)^2}{0.5 \times n^2} = 100.$$

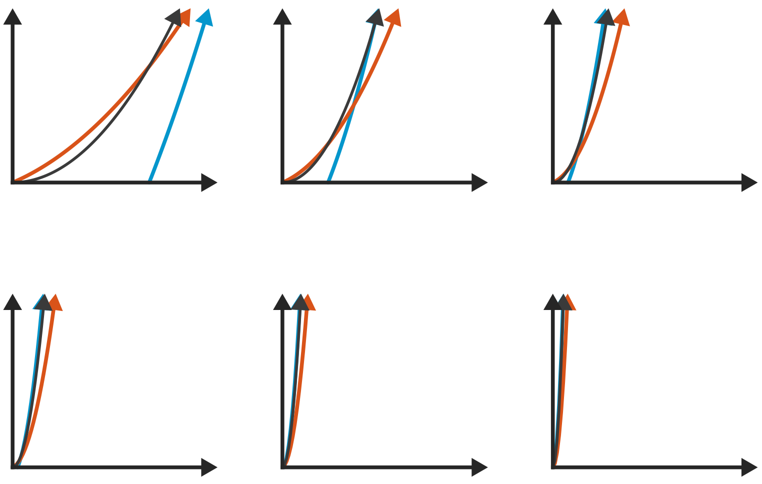


Figure 2.2 Zooming out n^2, $n^2 + n - 2$, and $0.5n^2 + 0.5n$, as n gets larger and larger.

The 0.5 coefficient cancels itself out! The idea that $n^2 + n - 2$ and $0.5n^2 + 0.5n$ both grow like n^2 isn't easy to get. How does the fastest-growing term of a function ignore all other numbers and dominate growth? Let's try to visually understand this.

In fig. 2.2, the two time complexities we've seen are compared to n^2 at different zoom levels. As we plot them for larger and larger values of n, their curves seem to get closer and closer. Actually, you can plug any numbers into the bullets of $\mathbb{T}(n) = \bullet\ n^2 + \bullet\ n + \bullet$, and it will still grow like n^2.

Remember, this effect of curves getting closer works if the fastest-growing term is the same. The plot of a function with a linear growth (n) never gets closer and closer to one with a quadratic growth (n^2), which in turn never gets closer and closer to one having a cubic growth (n^3).

That's why with very big inputs, algorithms with a quadratically growing cost perform a lot worse than algorithms with a linear cost. However they perform a lot better than those with a cubic cost. If you've understood this, the next section will be easy: we will just learn the fancy notation coders use to express this.

2.2 The Big-O Notation

There's a special notation to refer to classes of growth: the **Big-O notation**. A function with a fastest-growing term of 2^n *or weaker* is $\mathcal{O}(2^n)$; one with a quadratic *or weaker* growth is $\mathcal{O}(n^2)$; growing linearly *or less*, $\mathcal{O}(n)$, and so on. The notation is used for expressing the dominant term of algorithms' cost functions in the worst case—that's the standard way of expressing time complexity.[3]

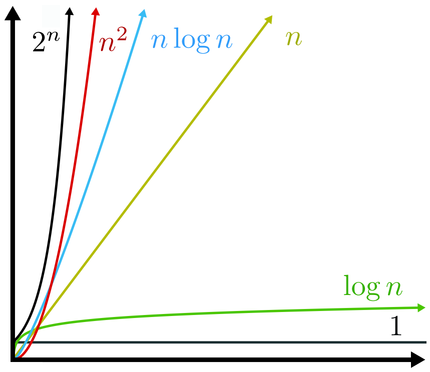


Figure 2.3 Different orders of growth often seen inside $\mathcal{O}$.

Both Selection Sort and Bubble Sort are $\mathcal{O}(n^2)$, but we'll soon discover $\mathcal{O}(n \log n)$ algorithms that do the same job. With our $\mathcal{O}(n^2)$ algorithms, $10\times$ the input size resulted in $100\times$ the running cost. Using a $\mathcal{O}(n \log n)$ algorithm, $10\times$ the input size results in only $10 \log 10 \approx 34\times$ the running cost.

When n is a million, n^2 is a trillion, whereas $n \log n$ is just a few million. Years running a quadratic algorithm on a large input could be equivalent to minutes if a $\mathcal{O}(n \log n)$ algorithm was used. That's why you need time complexity analysis when you design systems that handle very large inputs.

When designing a computational system, it's important to anticipate the most frequent operations. Then you can compare the Big-O costs of different algorithms that do these operations.[4] Also,

[3]We say *'oh'*, e.g., "that sorting algorithm is *oh-n-squared*".

[4]For the Big-O complexities of most algorithms that do common tasks, see http://code.energy/bigo.

most algorithms only work with specific input structures. If you choose your algorithms in advance, you can structure your input data accordingly.

Some algorithms always run for a constant duration regardless of input size—they're $\mathcal{O}(1)$. For example, checking if a number is odd or even: we see if its last digit is odd and *boom*, problem solved. No matter how big the number. We'll see more $\mathcal{O}(1)$ algorithms in the next chapters. They're amazing, but first let's see which algorithms are *not* amazing.

2.3 Exponentials

We say $\mathcal{O}(2^n)$ algorithms are **exponential time**. From the graph of growth orders (fig. 2.3), it doesn't seem the quadratic n^2 and the exponential 2^n are much different. Zooming out the graph, it's obvious the exponential growth brutally dominates the quadratic one:

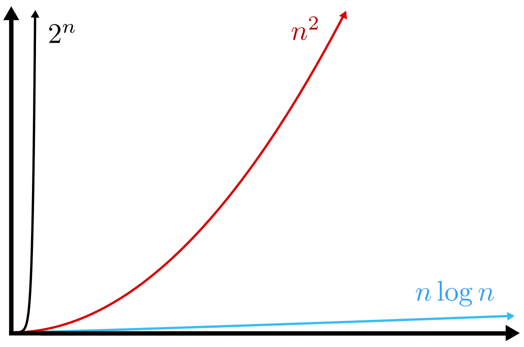


Figure 2.4 Different orders of growth, zoomed out. The linear and logarithmic curves grow so little they aren't visible anymore.

Exponential time grows *so much*, we consider these algorithms "not runnable". They run for very few input types, and require huge amounts of computing power if inputs aren't tiny. Optimizing every aspect of the code or using supercomputers doesn't help. The crushing exponential always dominates growth and keeps these algorithms unviable.

To illustrate the explosiveness of exponential growth, let's zoom out the graph even more and change the numbers (fig. 2.5). The exponential was reduced in power (from 2 to 1.5) and had its growth divided by a thousand. The polynomial had its exponent increased (from 2 to 3) and its growth multiplied by a thousand.

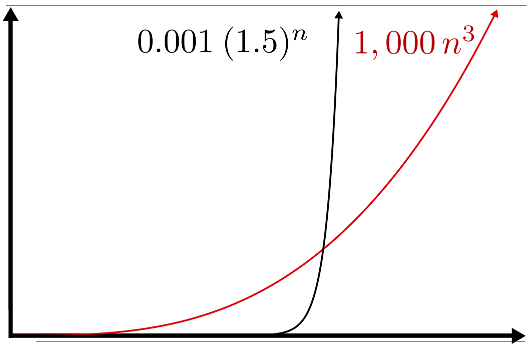


Figure 2.5 No exponential can be beaten by a polynomial. At this zoom level, even the $n \log n$ curve grows too little to be visible.

Some algorithms are even worse than exponential time algorithms. It's the case of **factorial time** algorithms, whose time complexities are $\mathcal{O}(n!)$. Exponential and factorial time algorithms are horrible, but we need them for the hardest computational problems: the famous **NP-complete** problems. We will see important examples of NP-complete problems in the next chapter. For now, remember this: the first person to find a non-exponential algorithm to a NP-complete problem gets a million dollars 💰[5] from the Clay Mathematics Institute.

It's important to recognize the class of problem you're dealing with. If it's known to be NP-complete, trying to find an optimal solution is fighting the impossible. Unless you're shooting for that million dollars.

[5]It has been proven a non-exponential algorithm for *any* NP-complete problem could be generalized to *all* NP-complete problems. Since we don't know if such an algorithm exists, you also get a million dollars if you prove an NP-complete problem cannot be solved by non-exponential algorithms!

2.4 Counting Memory

Even if we could perform operations infinitely fast, there would still be a limit to our computing power. During execution, algorithms need working storage to keep track of their ongoing calculations. This consumes **computer memory**, which is not infinite.

The measure for the working storage an algorithm needs is called **space complexity**. Space complexity analysis is similar to time complexity analysis. The difference is that we count computer memory, and not computing operations. We observe how space complexity evolves when the algorithm's input size grows, just as we do for time complexity.

For example, Selection Sort (sec. 2.1) just needs working storage for a fixed set of variables. The number of variables does not depend on the input size. Therefore, we say Selection Sort's space complexity is $\mathcal{O}(1)$: no matter what the input size, it requires the same amount of computer memory for working storage.

However, many other algorithms need working storage that grows with input size. Sometimes, it's impossible to meet an algorithm's memory requirements. You won't find an appropriate sorting algorithm with $\mathcal{O}(n \log n)$ time complexity *and* $\mathcal{O}(1)$ space complexity. Computer memory limitations sometimes force a trade-off. With low memory, you'll probably need an algorithm with slow $\mathcal{O}(n^2)$ time complexity because it has $\mathcal{O}(1)$ space complexity. In future chapters, we'll see how clever data handling can improve space complexity.

Conclusion

In this chapter, we learned algorithms can have different types of voracity for consuming computing time and computer memory. We've seen how to assess it with time and space complexity analysis. We learned to calculate time complexity by finding the *exact* $\mathbb{T}(n)$ function, the number of operations performed by an algorithm.

We've seen how to express time complexity using the Big-O notation ($\mathcal{O}$). Throughout this book, we'll perform simple time complexity analysis of algorithms using this notation. Many times, calculating $\mathbb{T}(n)$ is not necessary for inferring the Big-O complexity of an algorithm. We'll see easier ways to calculate complexity in the next chapter.

We've seen the cost of running exponential algorithms explode in a way that makes these algorithms not runnable for big inputs. And we learned how to answer these questions:

- Given different algorithms, do they have a significant difference in terms of operations required to run?
- Multiplying the input size by a constant, what happens with the time an algorithm takes to run?
- Would an algorithm perform a reasonable number of operations once the size of the input grows?
- If an algorithm is too slow for running on an input of a given size, would optimizing the algorithm, or using a supercomputer help?

In the next chapter, we'll focus on exploring how strategies underlying the design of algorithms are related to their time complexity.

Reference

- The Art of Computer Programming, Vol. 1, by Knuth
  - Get it at https://code.energy/knuth
- The Computational Complexity Zoo, by hackerdashery
  - Watch it at https://code.energy/pnp
- What is Big O, by Undefined Behavior
  - Watch it at https://code.energy/bigo-vid