Digraph of transformations between NP-complete problems.¹

(5) is subset sum
(13) is Hamiltonian circuit
(33) is the Travelling Salesman Problem
(130) is knapsack problem
(291) is 3-coloring

Acknowledgements

These notes have been developed based off previous professors’ work over many years. It was originally produced by Dr. Esfandiar Haghverdi, then modified by Dr. Saúl Blanco, and also Dr. Shabnam Kavousian.

First and foremost, I would like to thank Dr. Shabnam Kavousian and Dr. Nathan Ensmenger for their immense support throughout this effort.

Additionally, I would like to thank all the professors and faculty who have put in resources toward developing the content for this course. I would also like to thank Dr. Michael Shindler and Dr. Aaron Cote for their support and inspiration to write this set of lecture notes.

Finally, I would like to thank Julia Turner and Anna Lubienski for investing their time in helping review this iteration.

Usage

This set of lecture notes is not meant to be used in isolation and be a comprehensive set of notes for the course. This set of notes is meant to be used in conjunction with the lectures to fill in the exercises and understand the material in depth. This is not a substitution for lecture.

Errata

If you notice any mistakes, grammatical errors, unclear explanations / definitions, please contact Stephen Tsung-Han Sher at stsher@iu.edu.

SAMPLE

This is a sample of the full set of course notes. For the full version, please contact me at stsher@iu.edu
Introduction

It is perhaps too easy to just motivate this entire course with “we are learning mathematics because it is the foundation for everything we do.” We often hear about how important mathematics is. However, I’ve found it is quite difficult to understand why we are learning the contents of this course. From talking to numerous students over the years about why mathematics is the foundations of informatics and computing, the one story that seems to motivate the most students is, quite surprisingly, rooted in the history of computing. And thus, I would like to take a quick moment to go through the history of how the computer came to be.

The Journey to the Turing Machine

Computational theory, which is the field of study that over-arches this course, began with Alan Turing\(^2\) and his conception of the *Turing Machine* in 1936.\(^3\) The Turing Machine is the theoretical model in which all of our modern computing is based off of, from our cellphones to laptops, printers, servers, microwaves, and digital watches. The goal of this course is to teach you how to think like computers; in other words, teach you to think like a Turing Machine.

However, Turing did not develop this conceptual model of a computer overnight. It also would be horribly confusing to jump into his work immediately. And so, to really understand computers, we need to spend time understanding the work of his predecessors: Aristotle → Leibniz → Boole → Frege → Cantor → Hilbert → Gödel → Turing

In **Unit 1** we will learn propositional logic, the method in which great thinkers and philosophers have and continue to use to discern true and false statements. Propositional logic is the work done by Aristotle, Leibniz, and Boole, and thus will begin our journey towards Turing’s work. **Unit 2**, which focuses on predicate logic and algorithms, will touch upon the work of Frege, as well as Cantor, and just tangentially Hilbert and Gödel’s work, too. Finally, in **Unit 3**, we will look at graph theory, used to model the Turing Machine.

By the end of this course you will have travelled through the journey from Aristotle to Turing, from basic logic to algorithms to graph theory. While this may sound like a daunting task or a dull endeavor, I hope, nonetheless, you will accompany me through the journey of computation.

---

\(^2\)If you’re unfamiliar with Alan Turing, watch the movie *The Imitation Game* starring Benedict Cumberbatch.

\(^3\)In Turing’s paper *On Computable Numbers, with an Application to the Entscheidungsproblem*, which is a surprisingly pleasant read, despite the title of the paper.
This Course is “Hard”

You may have heard that this course is hard, and yes, I agree that this course is “hard”, but not in the way you think it is hard.

If you’ve ever learned how to speak a second language, learn an instrument, or play a sport, you understand that no one expects you to be proficient your very first day. We spend years honing our skills for language, music, and athletics because they are difficult tasks. This course is this type of “hard”. In reality, this course is more akin to a language course than a mathematics one. If there is one thing I would like you to remember throughout the course, it is:

You are learning to think like a computer, and your brain is not used to thinking like a computer.

Open any computer, phone, tablet, and at the very center of any computational machine is the central processing unit (CPU):

This is the equivalent to the brains of a computer; it does all the logical processes and calculations. And, you may notice that your human brain has no commonalities with the brain of a computer. They are different in size, structure, and material. Computational thinking is not natural for a human brain, but the more you can think like a computer, the better you are able to work with them. As future informaticians, designers of technology systems, and software developers, your greatest asset is the ability to understand how a computer thinks/operates, and knowing the limitations of a computer.

Just like how one semester of introductory to violin is insufficient to turn a beginning violinist into a concert soloist, we do not expect you to master all of computational thinking in this course. This course is meant to develop your foundations in computational thinking. By the end of the semester you will be prepared to venture into more advanced topics in computation and informatics. Want to learn more about databases and networking? You’ll have learned about sets, relations, basic algorithms. Want to take a course on artificial intelligence? You’ll have enough knowledge on propositional and predicate logic, as well as rudimentary AI algorithms and heuristic algorithms in graph theory. Want to become a software engineer? You’ll have a strong foundation in writing algorithms to take the courses you need to become one.

Similar to learning a second language, a musical instrument, or a sport, this course will require time and practice. If you approach this course with patience and perseverance, you’ll be able to learn the language of computation, and I hope you’ll have great fun along the way.

Without further ado, shall we begin our journey?
# Table of Contents

4 Algorithms ..................................................... 6
  4.1 What are Algorithms? ...................................... 7
  4.2 Algorithm Basics .......................................... 8
    4.2.1 Name, Input, Output ................................ 9
    4.2.2 Syntax .............................................. 10
    4.2.3 Variables and Assignment ........................... 10
    4.2.4 Input and Return Values ............................. 11
    4.2.5 Operations .......................................... 14
    4.2.6 Arrays ............................................. 15
  4.3 Algorithm Structure and Flow ............................... 17
    4.3.1 Conditional Statements ............................... 17
    4.3.2 Loops ............................................. 19
  4.4 Algorithm Tracing .......................................... 24
  4.5 Types of Algorithms ....................................... 27
    4.5.1 Search Algorithms .................................. 27
    4.5.2 Sorting Algorithms .................................. 32
  4.6 Algorithmic Complexity .................................... 37
    4.6.1 Analyzing Time Complexity .......................... 37
    4.6.2 Algorithmic Growth .................................. 40
    4.6.3 Unreasonable Or Uncomputable? ..................... 43
  4.7 Recursive Algorithms ...................................... 44
4 Algorithms

“If you come across a tin of Royal Dansk Danish Butter Cookies, you expect to find some delicious buttery Danish cookies. However, if you open the tin and instead find a bunch of random sewing supplies, you’d be disappointed, infuriated, and confused all at the same time.”

If you were brave enough to venture into Cantor’s diagonal argument at the end of set theory, you would have come across the entscheidungsproblem, or the “decision problem”. Recall that this problem, roughly translated and very much simplified, is:

There are an infinite number of computer problems and an infinite number of computer programs we can write. Can we solve all computer problems?

Alan Turing and Alonzo Church, using Cantor’s discovery of some infinities being bigger than other infinities, proved:

There are infinitely more computer problems than there are programs to solve them.

Your immediate reaction may be a rather pessimistic one, thinking:

“Why do we bother then? If there are infinitely many computer problems we cannot solve, what’s the point in trying?”

But we cannot deny the tremendous impact computation has on humanity. The Anthropocene (the age of humans) is defined by humans’ technological advances, and computation is the one tool that has arguably shaped the human nature more so than any other technological advances.

Despite its mathematical limits, computation is still, undeniably, a powerful tool that we need to learn how to use, analyze, and apply to harness its full potential. So of the minuscule subset of infinite computer problems that we can solve, we want to study:

• What can a computer do?
• Given a problem, how do we use a computer to solve it?
• What are the common problems computers solve everyday?
• How do we tell if an algorithm is “good”?
• Which problems are easy for a computer to solve, which are hard, and which are impossible?
• What makes computers so powerful?

This section will answer each of these questions. Just as with propositional and predicate logic, the language of computers will feel very foreign to you. Approach it with patience and perseverance, and you will soon discover the fascinating world of computation. Let’s get started!
4.1 What are Algorithms?

If you’ve ever made any baked goods, used a recipe to make a meal, or followed an instructional, then you will be innately familiar with what an **algorithm** is.

**Exercise 4.1** On a separate sheet of paper, write a recipe for a peanut butter and jelly sandwich.

**Algorithm**

An algorithm is a finite set of precise instructions for performing a computation or solving a problem that can be repeated with consistent results.

*Note: The formal definition of an algorithm can be defined as a 7-tuple \((Q, \Gamma, b, \Sigma, \delta, q_0, F)\), but we don’t need to worry about that right now.*

The word “**algorithm**” derives from a mangled transliteration of *al-Khwārizmi* (c. 780 – 850), a great Persian mathematician and scholar. In 825, *al-Khwārizmi* wrote a famous treaty on numbers and allowable operations. The treaty was translated into Latin and read throughout Europe, greatly influencing many modern mathematicians. He was the one who introduced Arabic numerals to the Western civilization, the exact same numerals we use today.

You have been using **algorithms** without realizing it. Do the following very common calculation that you’ve all done before:

**Exercise 4.2** Calculate the sum from 1 to 10.

However now imagine you’re tutoring an elementary school student who gets overwhelmed by addition. Add 1 to 10!? So many numbers!? How would you now explain, step-by-step, how to sum up 1 to 10 to an elementary schooler?

**Exercise 4.3** Explain how to calculate the sum from 1 to 10 to an elementary school student.

The step-by-step instruction you just wrote is an **algorithm**. It is a **finite** and **precise** set of instructions on how to perform a calculation. Unfortunately, in many ways, a computer is dumber than an elementary school student; a computer is in some ways like a baby. Just as how you cannot tell an elementary schooler “just add up 1 to 10”, you cannot tell a computer “just add these numbers up for me”. We will need to learn how to write **precise** instructions for a computer.

As we study algorithms we will see how algorithms can **efficiently**, and often very elegantly, solve very important computational problems.
4.2 Algorithm Basics

There are many different computer languages to write algorithms in. Here is a small subset of computer languages writing an algorithm on how to add up the numbers 1 to 10:

**Javascript** (INFO-101, web scripting)
```javascript
var sum = 0;
for (var i = 1; i <= 10; ++i) {
    sum += i;
}
```

**Python** (INFO-210, general-purpose)
```python
sum = 0
for i in range(1, 11):
    sum += i
print(x)
```

**C++** (very famous and commonly used)
```cpp
int sum = 0;
for (int i = 1; i <= 10; ++i) {
    sum += i;
}
```

**Racket** (CSCI-211, an inherently recursive language)
```racket
(define (sum-up lst)
    (cond
        [(empty? lst) 0]
        [else + (first (lst)) (sum-up (rest lst))]]

(sum-up (list 1 2 3 4 5 6 7 8 9 10))
```

**Erlang** (language used in networking)
```erlang
sum(L) ->
    sum(L, 0).

sum([H|T], Acc) ->
    sum(T, H + Acc);

sum([], Acc) ->
    Acc.
> sum:sum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

You can see that there are a wide variety of computer languages. Which one do we choose? You may notice that Python, Javascript, and C++ are all very similar. In this course, we will be learning pseudocode. It is not a specific language, but a generalization of Python, Javascript, C++, and other similar languages that will allow you to write and communicate your algorithms clearly.

**Pseudocode**

An informal high-level description of a computer program or algorithm.
4.2.1 Name, Input, Output

When we write an algorithm using pseudocode, we need to clearly indicate what the algorithm is created to do. Often times it’s easy to tell what an algorithm is trying to do:

**Algorithm 1** Mystery Algorithm 1

```plaintext
s = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
return s
```

and other times it’s unclear what’s happening:

**Algorithm 2** Mystery Algorithm 2

```plaintext
s = 0
i = 1
while i ≤ 5 do
    s = i + (11 - i)
    i = i + 1
end while
return s
```

Therefore, we need provide some information to preempt the reader what the algorithm is going to do. For every algorithm you write, you must provide:

- an **algorithm name** (general description of the purpose of the algorithm);
- a **list of inputs** (what the algorithm is taking in to calculate);
- an **output** (the final calculated outcome).

Below is the exact same “Mystery Algorithm 2” you saw above, but with some preemptive information:

**Algorithm 3** Sum from 1 to 10

**Input:** Nothing  
**Output:** An integer sum

```plaintext
s = 0
i = 1
while i ≤ 5 do
    s = s + i + (11 - i)
end while
return s
```

Now with the preemptive information, even though you may not be able to fully comprehend the algorithm yet (Gaussian summation), you still get a really good idea of what the algorithm is doing, because you trust (hopefully) me that the description I’ve written is accurate.
4.2.2 Syntax

Just as how the following propositional formulae need to be well-formed and follow syntactic rules:

\[(p \land \underline{\land} q)\]

and how English also has syntactic rules:

“Speak like Yoda, we do not.”

pseudocode also has syntactic rules because it has to be easily readable by others. Since pseudocode is closely tied with mathematics, many of these syntactic rules we are already familiar with and expect.

<table>
<thead>
<tr>
<th>Algorithm 4 Bad Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> nothing</td>
</tr>
<tr>
<td><strong>Output:</strong> Integer sum</td>
</tr>
<tr>
<td>0 = \textbf{sum}</td>
</tr>
<tr>
<td>\textbf{sum} = 1 + +2 3 + all the way to 10</td>
</tr>
<tr>
<td>return The sum</td>
</tr>
</tbody>
</table>

Now there are small variations between different pseudocode styles and between the writers of pseudocode, and we do not expect you to have perfect syntax in this new language, however, we do expect that you conform to the major syntactic rules as they are introduced.

4.2.3 Variables and Assignment

One of the required components of a computer is a memory system to store and retrieve information. We can access this memory system by declaring variables and assigning values to them.

**Variables**

A symbol used to represent a unique unit of storage. This can come in the form of letter or singular words. The value of a variable is assigned with the equals sign with the variable on the left and the value on the right. We declare a variable \(x\) in the following way:

\[x = \text{<value>}\]

Exercise 4.4 *What are the values of \(x\), \(y\), and \(z\) at the end of the following algorithm?*

<table>
<thead>
<tr>
<th>Algorithm 5 Variables and Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Nothing</td>
</tr>
<tr>
<td><strong>Output:</strong> Nothing</td>
</tr>
<tr>
<td>(x = 5)</td>
</tr>
<tr>
<td>(y = x + 5)</td>
</tr>
<tr>
<td>(z = x - y)</td>
</tr>
</tbody>
</table>
We can also overwrite the values of variables. Because the computer follows algorithms top to bottom, variables will have the retain its last-assigned value.

**Exercise 4.5** What are the values of \( x \), \( y \), and \( z \) at the end of the following algorithm?

---

**Algorithm 6 Variables and Assignments**

**Input:** Nothing

**Output:** Nothing

- \( x = 5 \)
- \( y = x + 5 \)
- \( x = 20 \)
- \( z = x - y \)
- \( x = 5 \)

---

**4.2.4 Input and Return Values**

Every algorithm needs to have a purpose. Every variable you declare in your algorithm (program) will be deleted and disappear from existence after the algorithm ends, meaning if you don’t have a way to retain the outcome of your calculations then your algorithm is useless. The way in which an algorithm communicates the results of its calculation is through a **return value**.

---

**Return Value**

The value that is communicated back to the algorithm’s caller.

Likewise, an algorithm that always communicates the same value every single time is redundant. We would like an algorithm to be able to return different values depending on the inputs we give them. Therefore we also specify the inputs into a function.

---

**Input Values**

The value(s) that is specified as input(s) by the algorithm’s caller.

This is specified by the **input** and **output** descriptions we write at the beginning of each algorithm. With inputs, we will need to specify the specific variable names used to refer to these inputs. With the output, we need to declare a final **return statement** on the last line of our algorithm.
Example 4.1  Given three integers \( x, y, \) and \( z \), find the sum of these three integers

Algorithm 7 Sum of Three Integers

**Input:** Three integers \( x, y, \) and \( z \)

**Output:** An integer sum

\[
\begin{align*}
\text{sum} &= 0 \\
\text{sum} &= \text{sum} + x \\
\text{sum} &= \text{sum} + y \\
\text{sum} &= \text{sum} + z \\
\text{return} \quad \text{sum}
\end{align*}
\]

Exercise 4.6  Identify what is wrong (a bug) with the following algorithm:

Algorithm 8 Sum of Three Integers

**Input:** Three integers \( x, y, \) and \( z \)

**Output:** An integer sum of \( x, y, \) and \( z \)

\[
\begin{align*}
x &= 5 \\
y &= 10 \\
z &= 15 \\
\text{return} \quad x + y + z
\end{align*}
\]

This is an unspoken rule, but a very good rule to follow:

**DO NOT, EVER EVER EVER, change the value of your inputs.**

If you come across a tin of Royal Dansk Danish Butter Cookies, you expect to find some delicious buttery Danish cookies. However, if you open the tin and instead find a bunch of random sewing supplies, you’d be disappointed, infuriated, and confused all at the same time. That is the same feeling as changing the value of an input variable. Don’t do it.

Exercise 4.7  Given three integers \( a, b, \) and \( c \), find the sum of the square of these three integers.
Exercise 4.8 Given two integers \( x \) and \( y \), write an algorithm to calculate \( x^2 + 2xy + y^2 \).
4.2.5 Operations

Now that we’ve learned the very basics of an algorithm, we are now ready to answer the question:

What can a computer do?

So far you’ve seen and used the addition and subtraction operators in pseudocode, and you expect computers to be able to perform basic calculations. Generally speaking, computers can perform two types of operations: numerical and comparative operations.

Numerical operations should already be very familiar:

<table>
<thead>
<tr>
<th>Numerical Operations</th>
<th>Operations that instruct the computer to perform numerical calculations. These include:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Symbol</td>
</tr>
<tr>
<td>Addition</td>
<td>$x + y$</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$x - y$</td>
</tr>
<tr>
<td>Multiplication</td>
<td>$x \ast y$</td>
</tr>
<tr>
<td>Division*</td>
<td>$x / y$</td>
</tr>
<tr>
<td>Modulo (remainder)</td>
<td>$x % y$</td>
</tr>
</tbody>
</table>

* IMPORTANT: In computation, division will automatically round down to the nearest integer. This means $5/2 = 2$, $40/7 = 5$, $1/2 = 0$ etc.

However, be aware that there are operations that we have used in mathematics that we do not have access to without the use of special libraries. You can not use the following operations:

<table>
<thead>
<tr>
<th>Unavailable Operations</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponents or power</td>
<td>$x^2$</td>
</tr>
<tr>
<td>Square roots</td>
<td>$\sqrt{x}$</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$\log(x)$</td>
</tr>
</tbody>
</table>

Comparative operators will also be familiar to you, however they will operate in a way that may be confusing at first. Here you will also see why we spent so much time on logic in the first unit of the course.

<table>
<thead>
<tr>
<th>Comparative (Boolean) Operations</th>
<th>Operations that instruct the computer to perform Boolean (true or false) calculations. These operators will always output a true or a false:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Symbol</td>
</tr>
<tr>
<td>Equality</td>
<td>$x == y$</td>
</tr>
<tr>
<td>Inequality</td>
<td>$x != y$</td>
</tr>
<tr>
<td>Greater than</td>
<td>$x &gt; y$</td>
</tr>
<tr>
<td>Greater or equal</td>
<td>$x &gt;= y$</td>
</tr>
<tr>
<td>Less than</td>
<td>$x &lt; y$</td>
</tr>
<tr>
<td>Less or equal</td>
<td>$x &lt;= y$</td>
</tr>
<tr>
<td>Logical AND</td>
<td>$x &amp;&amp; y$</td>
</tr>
<tr>
<td>Logical OR</td>
<td>$x</td>
</tr>
</tbody>
</table>

We will see the use of comparative operators when we get to conditional statements and loops. Be aware that there are also other types of operators, one of which you’ve been using already. The assignment operator (=) is a unique operator that is neither a numerical nor comparative operator.
4.2.6 Arrays

Let’s say we want to add up the sum of 10 integers given to us as input. Our pseudocode will be:

Example 4.2 Given 10 input variables \(a, \cdots, j\), find the sum of the variables.

Algorithm 9 Sum of 10 Input Variables

**Input:** 10 integers \(a, b, c, d, e, f, g, h, i,\) and \(j\).

**Output:** An integer sum

\[
\text{return } a + b + c + d + e + f + g + h + i + j
\]

This seems rather silly to use so many different variables. What if we need to write an algorithm with 100 input variables? 1000 input variables? Do we really need to define separate variables for every single unique input?

To resolve this problem, we introduce a *data structure* called an **array**.

Array

A single continuous section of memory used as data storage and retrieval. We represent the **members** of an array by encapsulating them between “[” and “]”.

An array has very similar properties as a **tuple**.

Consider the following array \(A\) with five elements:

\[
A = [17, 1, 19, 13, 61]
\]

We access the contents of an array by specifying the **index** of the member we want. In computation, we start the index of arrays at 0 for very good reasons that will become more apparent later:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>17</td>
<td>1</td>
<td>19</td>
<td>13</td>
<td>61</td>
</tr>
</tbody>
</table>

We say that the member stored at location 0 of the array \(A\) is 17. Likewise, we say that the member stored at location 1 is 1. This process of accessing the contents of an array is called **indexing**. We have a shorthand for this:

\[
A[0] = 17
A[1] = 1
\]

**Exercise 4.9** Given the array \(A\) defined above, what is \(A[2], A[3], A[4],\) and \(A[5]\)?
Let’s see how this would be used in pseudocode:

**Example 4.3** Given an array of integers $A$ of size 5, write an algorithm to find the sum of the array.

**Algorithm 10** Sum of $A$

**Input:** An array $A$ of size 5

**Output:** An integer sum

```
sum = 0
sum = sum + A[0]
sum = sum + A[1]
sum = sum + A[2]
sum = sum + A[3]
return sum
```

**Exercise 4.10** Given an array of integers $A$ of size 5, write an algorithm to find the product of the array.

**Exercise 4.11** Given an array of integers $B$ of size 5, write an algorithm to find the average of the numbers in the array.
4.3 Algorithm Structure and Flow

Currently all of our algorithms run linearly – line by line by line. However linear algorithms have very limited use. Here we will introduce two components of algorithms that allow us to alter the structure and flow of our algorithms: **conditional statements** and **loops**. This will allows us to answer the question:

Given a problem, how do we use a computer to solve it?

Once you’ve learned how to use **conditional statements** and **loops**, along with the **variables** and **operations** we just covered, you can write algorithms for everything. Every single computer program in existence boils down to these four structures.

### 4.3.1 Conditional Statements

Consider the following array $A$:

$$A = [3, -5, 7]$$

Let’s say we want to find the sum of the positive integers in $A$. How do we determine if an integer is positive or negative? As a human we can just look at it and see if it has a negative sign, but a computer cannot do that.

We know an integer is positive if and only if it is greater than 0. Here we introduce the **conditional statement** to achieve this exact goal.

**Conditional Statement**

A type of instruction where the algorithm will take differing predetermined actions depending on a **Boolean value** (true or false). This is achieved through the use of **if... else...** statements:

```
if (<boolean condition>) then
    <instructions if condition is true>
else
    <instructions if condition is false>
end if
```

**Example 4.4** Given an array of integers $A$ of size 3, find the sum of only the positive integers.

**Algorithm 11** Sum of Only the Positive Integers in $A$

**Input:** An integer array $A$ of size 2.

**Output:** An integer sum of only the positive integers.

```plaintext
sum = 0
if (A[0] > 0) then
    sum = sum + A[0]
end if
if (A[1] > 0) then
    sum = sum + A[1]
end if
if (A[1] > 0) then
    sum = sum + A[1]
end if
return sum
```
Exercise 4.12  Given an array of integers (both positive and negative) $A$ of size 3, write an algorithm to find the sum of the absolute value of each integer.
You may not use the mathematical notation for absolute value $|x|$.

Exercise 4.13  Given an array of integers $A$ of size 5, write an algorithm to find the sum of only the even integers (hint: What is the remainder of an even number when divided by 2?)
4.3.2 Loops

By now you’ve noticed that you’ve written a lot of redundant lines of pseudocode. Consider the following problem we want to solve:

**Example 4.5** Given an array of integers $A$ of size 10, write an algorithm to find the sum of only the odd integers.

**Algorithm 12** Sum of all odd integers in an Array of Size 10

**Input:** An array $A$ of size 10.

**Output:** An integer sum of only the odd integers.

```plaintext
sum = 0
if (A[0] % 2 == 1) then
    sum = sum + A[0]
end if
if (A[1] % 2 == 1) then
    sum = sum + A[1]
end if
if (A[2] % 2 == 1) then
    sum = sum + A[2]
end if
if (A[3] % 2 == 1) then
    sum = sum + A[3]
end if
if (A[4] % 2 == 1) then
end if
if (A[5] % 2 == 1) then
    sum = sum + A[5]
end if
if (A[6] % 2 == 1) then
end if
if (A[7] % 2 == 1) then
    sum = sum + A[7]
end if
if (A[8] % 2 == 1) then
    sum = sum + A[8]
end if
if (A[9] % 2 == 1) then
    sum = sum + A[9]
end if
return sum
```

Well this is really really silly isn’t it? We’re just repeating the same steps over and over and over again. There has to be a more convenient way, and there is! Here we introduce the **loop** to achieve this exact goal.
Loop
A type of instruction where the algorithm will repeat the same specified set of instructions depending on a Boolean value (true or false). This is achieved through the use of a while statement:

```
while (<boolean condition>) do
    <instructions as long as condition is true>
end while
```

Note: There are other types of loops such as for loops and do while loops. They achieve the same behavior just with different syntax.

Let’s see how we can use a loop to simplify the algorithm we wrote on the previous page.

First, we identify the lines that are not repeated. These will not be included in our loop:

```
sum = 0
...
...
...
...
...
...

return sum
```

Now let’s look at the lines that are being repeated. We notice that the following chunk of instructions is begin repeated 10 times, once for each integer:

```
if (A[ 0 ] % 2 == 1) then
    sum = sum + A[ 0 ]
end if
```

However, it is not an exact replica ever single time. There are are small changes being made each iteration. Specifically, the A[0] changes to A[1], then to A[2]··· all the way up to A[9]. So instead of using an integer, let’s use a variable instead:

```
if (A[ i ] % 2 == 1) then
    sum = sum + A[ i ]
end if
```

By convention, we use the variable i to keep track of how many times we need to repeat a set of instructions. The i stands for “iterator”.

20
We know that $i$ will start from 0 and increment by one until we reach 9, so we need to declare $i = 0$ at the beginning of our algorithm:

**Input:** An array $A$ of size 10.
**Output:** An integer sum of only the odd integers.
```
sum = 0
i = 0
... if (A[i] % 2 == 1) then
    sum = sum + A[i]
end if
... return sum
```

Next, let’s use a **while loop** statement to tell our algorithm to repeat the middle section of the algorithm:

**Input:** An array $A$ of size 10.
**Output:** An integer sum of only the odd integers.
```
sum = 0
i = 0
while (<condition>) do
    if (A[i] % 2 == 1) then
        sum = sum + A[i]
    end if
    i = i + 1
end while
return sum
```

What is our **loop condition**? In other words, under what **condition** do we want to repeat the instruction? We know that $i$ starts at 0 since we declared it as such. $i$ will then need to increase by 1 each time we repeat the instruction, so we need to tell the algorithm to add 1 to $i$ right before the loop repeats (right before the `end while` line). Finally, we know that the condition will be to repeat as long as $i$ is less or equal to 9, or equivalently as long as $i > 10$.

Putting all this together, we get the following:

**Algorithm 13** Sum of all odd integers in an Array of Size 10
```
Input: An array $A$ of size 10.
Output: An integer sum of only the odd integers.
sum = 0
i = 0
while (i < 10) do
    if (A[i] % 2 == 1) then
        sum = sum + A[i]
    end if
    i = i + 1
end while
return sum
```
With loops we can now very efficiently write algorithms to perform very large tasks. You now have all the tools you need to write any algorithm you wish!

**Exercise 4.14** Write an algorithm to find the sum of the first 1000 integers (0 included, meaning sum from 0 to 999).

**Exercise 4.15** Modify your previous algorithm to find the sum of the first \( n \) integers (0 included).

**Exercise 4.16** Given an array of integers \( A \) of size \( n \), write an algorithm to find the average of the array.
Exercise 4.17  Given an array of integers $A$ of size $n$ and a target integer $k$, write an algorithm to that will return $true$ if $k$ is in the array and will return $false$ if $k$ is not in the array.

Exercise 4.18  Given an array of integers $A$ of size $n$, write an algorithm to return the largest integer in the array.
4.4 Algorithm Tracing

Now that we've written quite a few algorithms, let's turn our attention to a problem that you've most likely faced in the previous few exercises:

How do I know my algorithm is correct and behave the way I want it to?

The process in which we analyze the behavior of an algorithm is called **tracing**.

---

**Algorithm Tracing**

A step-by-step breakdown of an algorithm's behavior given a specific set of inputs to analyze what an algorithm does or if it is accurately solving the given problem.

There are many ways to trace an algorithm. The most common method is to use a table listing out every variable and calculation of interest and tracking how they change as the algorithm executes.

---

Let's take a look at sample algorithm to the previous exercise:

**Example 4.6** Given an array of integers $A$ of size $n$, write an algorithm to return the maximum integer in the array.

**Algorithm 14** Find Largest

**Input:** An array $A$ of size $n$.

**Output:** The maximum integer in $A$

max = 0
i = 0
while $(i \leq n)$ do
  if $(\text{max} < A[i])$ then
    max = $A[i]$
  end if
i = i + 1
end while
return max

To tell if this algorithm is correct, we will trace the above algorithm with a sample input:

**Example 4.7** Trace the above algorithm with the given the sample input $A$:

$A = [3, 15, 2, 20, -5]$

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>max before loop</td>
<td>0</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>i ≤ n</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>$A[i]$</td>
<td>3</td>
<td>15</td>
<td>2</td>
<td>20</td>
<td>-5</td>
<td>???</td>
</tr>
<tr>
<td>max &lt; $A[i]$</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>max after loop</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>20</td>
<td>20</td>
<td>—</td>
</tr>
</tbody>
</table>
From this trace, we’ve found a bug in the algorithm. The array \( A \) is of size 5, meaning its last member is at index 4. However, because the while loop’s condition is \((i \leq n)\), our algorithm will iterate one too many times to \( i = 5 \), which \( A[5] \) does not exist.

To fix this, we will ensure that the while loop condition is changed to \((i < n)\):

**Algorithm 15** Find Largest

**Input:** An array \( A \) of size \( n \).

**Output:** The largest integer in \( A \)

\[
\begin{align*}
\text{max} &= 0 \\
i &= 0 \\
\text{while } (i < n) \text{ do} \\
&\quad \text{if } (\text{max} < A[i]) \text{ then} \\
&\quad\quad \text{max} = A[i] \\
&\quad \text{end if} \\
i &= i + 1 \\
\text{end while} \\
\text{return } \text{max}
\end{align*}
\]

**Exercise 4.19** Trace the above algorithm with the given the sample input \( A \):

\[
A = [-5, -10, -2, -1, -8]
\]

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>max before loop</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>i &lt; n</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>A[i]</td>
<td>-5</td>
<td>-10</td>
<td>-2</td>
<td>-1</td>
<td>-8</td>
<td>—</td>
</tr>
<tr>
<td>max &lt; A[i]</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>max after loop</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
<td>\</td>
</tr>
</tbody>
</table>

**Exercise 4.20** What is the bug in the algorithm you just traced? How can you fix it?
Exercise 4.21  *Trace the following algorithm with the input* \( n = 15 \).

**Algorithm 16** Unkonwn Algorithm Ohhhh So Mysterious

**Input:** An positive integer \( n \)

**Output:** An integer

\[
\begin{align*}
x &= 0 \\
i &= 1 \\
\text{while } (i \leq (n/2)) \text{ do} \\
x &= x + i + (n + 1 - i) \\
i &= i + 1 \\
\text{end while} \\
\text{if } (n \% 2 == 1) \text{ then} \\
x &= x + ((n / 2) + 1) \\
\text{end if} \\
\text{return } x
\end{align*}
\]

*Hint 1:* Remember, division will round down to the nearest whole number.

*Hint 2:* Trace the following variables: \( i, n, n/2, x \) before loop, \( i \leq (n/2), n + 1 - i, i + (n + 1 - i), \) and \( x \) after loop.

Exercise 4.22  **Bonus:** What does this algorithm do?
4.5 Types of Algorithms

Now that you’ve written quite a few algorithms to solve problems, let’s turn our attention to the next question:

What are the common problems computers solve everyday?

We will look at the two of the most common problems: search and sorting problems. While these may seem very simple problems to solve, computer scientists have spent a lot of time coming up with better strategies and are still finding new ways to optimize computers to solve these problems.

4.5.1 Search Algorithms

Search Algorithm
Given an array of integers $A$ and a target integer $k$, write an algorithm to return true if and only if $k \in A$.

You’ve already written the solution to this problem in Exercise 1.17! This is known as the linear search algorithm.

Linear Search
Given an array $A$ and target $k$:
1. Create an index $i = 0$.
2. Repeat in a loop as long as $i < n$:
   (a) If $A[i] == k$, then return true.
   (b) $i = i + 1$
3. Return false if the loop never returned true.

Algorithm 17 Linear Search
Input: An array $A$ of size $n$ and a target $k$.  
Output: true if and only if $k \in A$.
\[
i = 0 \\
\text{while } (i < n) \text{ do} \\
\quad \text{if } (A[i] == k) \text{ then} \\
\quad \quad \text{return true} \\
\quad \text{end if} \\
\quad i = i + 1 \\
\text{end while} \\
\text{return false}
\]
Example 4.8 Trace linear search on the following array \( A \) with the target \( k = 33 \):

\[
A = [3, 15, -7, 33, 17, 15, 22, 24, 23]
\]

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k )</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>( n )</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>( A[i] )</td>
<td>3</td>
<td>15</td>
<td>-7</td>
<td>33</td>
</tr>
<tr>
<td>( A[i] == k )</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Exercise 4.23 Trace linear search on the above array \( A \) with the target \( k = 23 \):
From the previous example, we notice a small problem with **linear search**: if the target $k$ is at the very end of the array $A$, it’s going to take us a long time to find it. So naturally, the next question we want to ask is: can we do better?

**Exercise 4.24** Trace linear search (just roughly) on the following array $S (n = 17)$ with the target $k = 23$:

$$S = [1, 2, 3, 4, 6, 8, 9, 10, 13, 14, 15, 16, 18, 19, 21, 22, 23]$$

**Exercise 4.25** What do you notice about the array $S$?

**Exercise 4.26** What’s a better strategy than linearly looking at every single integer in sequential order?
Here we experience one of the most important rules in computation:

The more information we know about the problem (e.g., structure, behavior, scope), the more efficiently we can solve the problem.

This rule is definitely applied to the searching problem. We can find our target integer more efficiently if and only if the starting array is sorted. We will also later discuss exactly what it means for an algorithm to be efficient.

The strategy you came up with for searching through a sorted array is called binary search.

---

**Binary Search**

Given a sorted array $A$ and target $k$:

1. Repeat in a loop until no elements left in $A$:
   
   (a) If $(A[n/2] == k)$:
       
       Return true
   
   (b) If $(A[n/2] > k)$:
       
       Check left of n/2
   
   (c) If $(A[n/2] < k)$:
       
       Check right of n/2

2. Return false

---

**Algorithm 18 Binary Search**

**Input:** An array $A$ of size $n$ and a target $k$

**Output:** true if and only if $k \in A$.

```plaintext
left = 0
right = n - 1
while (left ≤ right) do
    mid = (left + right) / 2
    if (A[mid] == k) then
        return true
    else if (A[mid] > k) then
        right = mid - 1
    else
        left = mid + 1
    end if
end while
return false
```

We will see in the last portion of algorithms how to write a much more elegant version of binary search using recursion.
Example 4.9  Trace binary search with the following array $A$ with the target $k = 59$.

\[ A = [2, 3, 4, 7, 11, 16, 19, 50, 55, 59, 80, 97] \]

<table>
<thead>
<tr>
<th>A</th>
<th>n</th>
<th>n/2</th>
<th>A[n/2]</th>
<th>A[n/2] == 59</th>
<th>Left or Right?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2,3,4,7,11,16,19,50,55,59,80,97]</td>
<td>12</td>
<td>6</td>
<td>19</td>
<td>F</td>
<td>Right</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>59</td>
<td>T</td>
<td>—</td>
</tr>
</tbody>
</table>

Exercise 4.27  Trace binary search with the array $A$ from Example 1.8 with the target $k = 3$.

Exercise 4.28  Trace binary search with the following array $B$ with target $k = 89$.

\[ B = [40, 60, 65, 77, 89, 120, 200] \]

Remember, division will round down to the nearest whole number.

Exercise 4.29  Trace binary search with the array $b$ from Exercise 1.28 and the target $k = 200$. 

4.5.2 Sorting Algorithms

Now that we know performing binary search on a sorted array is much more efficient than linear search, it lends itself nicely to the next question:

**Given an unsorted array A, how do we sort A?**

This is a problem that has been examined by many computer scientists. There are currently 11 main sorting algorithms that have been developed:

<table>
<thead>
<tr>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>Merge Sort</th>
<th>Heapsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>Shellsort</td>
<td>Bubble Sort</td>
<td>Comb Sort</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>Bucket Sort</td>
<td>Radix Sort</td>
<td></td>
</tr>
</tbody>
</table>

Here we will look at two of the simpler sorting algorithms: **selection sort** and **insertion sort**.

---

### Selection Sort

Given an array A:

1. Create an empty array $S$
2. Repeat in a loop until no elements are left in $A$:
   - (a) Find the smallest member in $A$ and move it to the end of $S$
3. Return $S$

---

**Algorithm 19 Selection Sort**

**Input:** An array $A$ of size $n$.  
**Input:** A sorted array.  

```
S = [ ]
while (!empty(A)) do
  min = findMin(A)
  A.remove(min)
  S.append(min)
end while
return S
```

*Note: There are a few more advanced instructions written in the pseudocode for selection sort. See if you can make sense of them!*

---

Notice that in the pseudocode of selection sort we have the line:

$$\text{min} = \text{findMin}(A)$$

$\text{findMin}(A)$ here is a function call to run the algorithm $\text{findMin}$ with $A$ as input. As the name suggests, $\text{findMin}$ will take an array as input and return the smallest member of that array.
Exercise 4.30 Given an array $A$ of size $n$, write an algorithm to return the minimum integer in the $A$.

Example 4.10 Trace selection sort with the following array $A$:

\[ A = [8, 5, 3, 1, 2, 1, 21, 13] \]

<table>
<thead>
<tr>
<th>A</th>
<th>Smallest</th>
<th>Sorted Array $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[8, 5, 3, 1, 2, 1, 21, 13]</td>
<td>1</td>
<td>[ ]</td>
</tr>
<tr>
<td>[8, 5, 3, 2, 1, 21, 13]</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>[8, 5, 3, 21, 13]</td>
<td>2</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>[8, 5, 21, 13]</td>
<td>3</td>
<td>[1, 1, 2]</td>
</tr>
<tr>
<td>[8, 21, 13]</td>
<td>5</td>
<td>[1, 1, 2, 3]</td>
</tr>
<tr>
<td>[21, 13]</td>
<td>8</td>
<td>[1, 1, 2, 3, 5]</td>
</tr>
<tr>
<td>[ ]</td>
<td>13</td>
<td>[1, 1, 2, 3, 5, 8]</td>
</tr>
<tr>
<td>[ ]</td>
<td>21</td>
<td>[1, 1, 2, 3, 5, 8, 13]</td>
</tr>
</tbody>
</table>

Exercise 4.31 Trace selection sort with the following array $B$:

\[ B = [3, 13, 61, 17, 2] \]
Selection sort is pretty neat, but there’s a small non-ideal characteristic: the extra array $S$. Sometimes memory is a very scarce resource and we do not want to create an extra array if we can avoid it. Let’s see if we can make a few interesting observations and improve upon selection sort.

Exercise 4.32 Use any method you wish to sort the following array $E$:

$$E = [ ]$$

Exercise 4.33 Use any method you wish to sort the following array $HR$:

$$HR = [ 1729 ]$$

Stephen’s favorite number is 1729 but that has nothing to do with the exercise.

Exercise 4.34 Sort ONLY the first element of the following array $A$

$$A = [36, 3, 10, 28, 21, 15, 6]$$

Exercise 4.35 Sort ONLY the first two elements of the above array $A$

Exercise 4.36 Working off from Exercise 1.35, sort the third element of the array.

If we continue this process for the remainder of the array, this algorithm is called insertion sort.
**Insertion Sort**
Given an array \( A \):
1. Create an index \( i = 1 \)
2. Repeat in a loop as long as \( i < n \):
   (a) Insert \( A[i] \) into its sequential order between \( A[0] \) and \( A[i-1] \).
   (b) \( i = i + 1 \)
3. Return \( A \)

**Algorithm 20 Insertion Sort**

**Input**: An array \( A \) of size \( n \).

**Input**: A sorted array.

\[
i = 1
\]
while \( (i < n) \) do
   \[
   \text{key} = A[i]
   \]
   \[
   j = i - 1
   \]
   while \( (j \leq 0 \&\& A[j] > \text{key}) \) do
      \[
      \]
      \[
      j = j - 1
      \]
end while
\[
A[j + 1] = \text{key}
\]
end while

return \( A \)

Here we see that the algorithm for **insertion sort** is a little bit more complicated than the algorithm for **selection sort**. Generally speaking (though not always the case):

The **more you want to optimize** an algorithm (e.g., avoid creating another array) the **more complex** the algorithm becomes.

The algorithms that both optimizes a solution and remain simple in implementation are the truly elegant algorithms and is where we see some really beautiful algorithms (e.g. merge sort).

**Example 4.11** Trace insertion sort on the following array \( A \):

\[
A = [36, 3, 10, 28, 21, 15, 6]
\]

<table>
<thead>
<tr>
<th>( A ) (sorted sub-array in blue)</th>
<th>Next member to insert (in red)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[36, 3, 10, 28, 21, 15, 6]</td>
<td>3</td>
</tr>
<tr>
<td>[3, 36, 10, 28, 21, 15, 6]</td>
<td>10</td>
</tr>
<tr>
<td>[3, 10, 36, 28, 21, 15, 6]</td>
<td>28</td>
</tr>
<tr>
<td>[3, 10, 28, 36, 21, 15, 6]</td>
<td>21</td>
</tr>
<tr>
<td>[3, 10, 21, 28, 36, 15, 6]</td>
<td>15</td>
</tr>
<tr>
<td>[3, 10, 15, 21, 28, 36, 6]</td>
<td>6</td>
</tr>
<tr>
<td>[3, 6, 10, 15, 21, 28, 36]</td>
<td>—</td>
</tr>
</tbody>
</table>
Exercise 4.37  Trace selection sort on the following array $B$:

$$B = [49, 25, 4, 9, 16, 121, 1, 25]$$

Exercise 4.38  Trace insertion sort on the array $B$ defined above
4.6 Algorithmic Complexity

Now that we’ve seen two different algorithms to sort an array, we want to ask the question:

Which algorithm is better, selection sort or insertion sort?

or in more generally:

How do we tell if an algorithm is “good”?

You may be tempted to say insertion sort because it does not need to create a new array like selection sort does, saving memory. And you’d be correct if we’re talking about better in terms of memory. However, when analyzing how good, we are typically talking in terms of time complexity.

4.6.1 Analyzing Time Complexity

**Time Complexity**

Given a problem with input size $n$, in the worse possible case, what is the required runtime of an algorithm $A$ to solve the question?

We denote the worst-case scenario of an algorithm using Big $O$ Notation:

$$O(f(n))$$

where $f(n)$ is any function of the input size $n$. The lower the order of $f(n)$, the more efficient the algorithm is.

Let’s first see what’s the time complexity or worst-case runtime of selection sort:

**Example 4.12 Analyze the time complexity of selection sort.**

Let’s consider the following array $A$ of size $n = 10$:

$A = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$

We know that to sort the entire array we’ll need to individually put each member of $A$ into our sorted array $S$. Since there are $n = 10$ members in our array, this will take us at least:

$$O(n)$$

However that’s all selection sort does. To find the first number to move from $A$ into $S$, we need to run the `findMin` algorithm we wrote. How long does `findMin` take? Well `findMin` will loop through the entire array to find the smallest number, meaning `findMin` will also take $O(n)$ time.

If `findMin` takes $O(n)$ time and we need to run `findMin` ($O(n)$) once for each of $A$’s $n$ members, then our worst-case runtime is:

$$O(n \times n) = O(n^2)$$
Example 4.13 Analyze the time complexity of insertion sort.

Let’s consider the same array $A$ of size $n = 10$:

$$A = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$$

Once again we know that to sort the entire array we’ll need to individually insert every single member of $A$ into its rightful position. Since there are $n = 10$ members in our array, this will take us at least:

$$O(n)$$

However, just as with selection sort, there’s a bit more nuance to insertion sort. Each time we are trying to insert a member into its rightful place, it needs to loop through the array to find the correct position. This process of finding the right position will take another $O(n)$ time.

If insertion sort will take $O(n)$ time to find the right position for each of $A$’s $n$ members, then our worst-case runtime is:

$$O(n \times n) = O(n^2)$$

From our analysis, it turns out that the time complexity of both selection sort and insertion sort are $O(n^2)$, meaning they are equally efficient.

What about linear search and binary search? The analysis of search algorithms are a bit more straightforward to do:

Example 4.14 Analyze the time complexity of linear search.

Let’s consider the situation where our target $k$ is at the very last position of our array of size $n$. In this situation, our algorithm will look through one-by-one from the start of the array until it reaches the end, looking at a total of $n$ members. Therefore the worst-case runtime is:

$$O(n)$$

Example 4.15 Analyze the time complexity of binary search

This is a bit harder to figure out, but once again let’s consider a sorted array $S$ of size $n$. The best case is when $A[n/2]$ is exactly our target $k$. However remember that time complexity is looking at the worst-case runtime. If we don’t find it on our first try, we will then look to the left or right of the midpoint depending on whether $A[n/2]$ is smaller or larger than $k$.

Each time we don’t find $k$, we will look to the left and right of the midpoint again, splitting the size of the array by half each time.

The worst-case scenario is if our target $k$ is in the very last place where we look. However, how long did it take us to get here? Since each loop of binary search reduces the size of the array by half until we get to a size of 1, our worst-case runtime is:

$$O(\log(n))$$

Note: In computation, log is automatically interpreted as $\log_2$. 

38
In the following section we will see why $O(n)$ is more efficient than $O(\log(n))$. However for now just keep in mind that binary search is more efficient than linear search. This is why earlier we said:

> The more information we know about the problem
> the more efficiently we can solve the problem.

**Exercise 4.39** If binary search is more efficient than linear search, should we just use binary search instead of linear search every single time?

**Exercise 4.40** If selection sort and insertion sort are equally efficient, does it matter if we use selection sort or insertion sort to sort an array?

The following sorting algorithm is beyond the scope of the course, but it’s still a bit important to bring up:

**Merge Sort**
Given an array $A$:
1. Split $A$ in a left half and right half.
2. Keep splitting the halves until the size of each half is 1.
3. Sort the small halves together and merge them.
4. Keep merging until all of $A$ is sorted.

As it turns out, **merge sort**’s worst-case runtime is $O(n \log(n))$, significantly better than **selection sort** and **insertion sort**’s $O(n^2)$. This is to show the following:

> Sometimes there are algorithms that solve the same problem with more efficiency.

In the actual practice, no one uses **selection sort** nor **insertion sort**. Programmers will opt to use more efficient sorting algorithms such as **merge sort**, **quicksort**, **heap sort**, and **shell sort** to complete the same task with more efficiency.
4.6.2 Algorithmic Growth

Now let’s turn our attention to answer the following question:

Which problems are easy for a computer to solve, which are hard, and which are impossible?

From the definition of time complexity that the complexity of an algorithm is dependent on its input size \( n \). This means if an algorithm has a time complexity of \( O(n) \), then as \( n \) gets larger, the time it takes for an algorithm to finish the problem will grow at a rate of \( f(n) = n \). If another algorithm has a time complexity of \( O(n^2) \), the time it takes will grow at a rate of \( f(n) = n^2 \). Let’s do a quick comparison of each time complexity with different input sizes \( n \):

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Approximate Required Time for Input Size ( n ) (in number of operations, smaller is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>1 1 1 1 1 1</td>
</tr>
<tr>
<td>( O(\log(n)) )</td>
<td>4 7 10 14 16 19</td>
</tr>
<tr>
<td>( O(n) )</td>
<td>10 100 1,000 10,000 100,000 1,000,000</td>
</tr>
<tr>
<td>( O(n \log(n)) )</td>
<td>34 670 7,800 130,000 1,600,000 20,000,000</td>
</tr>
<tr>
<td>( O(n^2) )</td>
<td>100 10,000 1,000,000 1.0 \times 10^8 1.0 \times 10^{10} 1.0 \times 10^{12}</td>
</tr>
<tr>
<td>( O(2^n) )</td>
<td>1024 1.3 \times 10^{10} Effectively ( \infty )</td>
</tr>
<tr>
<td>( O(n!) )</td>
<td>3,600,000 Effectively ( \infty )</td>
</tr>
</tbody>
</table>

We can now clearly see why binary search \( O(\log(n)) \) is significantly more efficient than linear search \( O(n) \) as well as why merge sort \( O(n \log(n)) \) is significantly more efficient than selection and insertion sort \( O(n^2) \).

We also have a few other time complexities that show up in algorithms that solve really important problems. Here is a graph summarizing them:
There can be some pretty wonky time complexities. For example, Ailon, Jaiswal, and Monteleoni’s algorithm for Streaming k-means Approximation has a time complexity of:

\( O(dnk \log(n) \log(k)) \)

We need an approach to analyze more complicated runtimes and compare their algorithmic growths to see how one algorithm’s efficiency compares to another algorithm’s efficiency.

**Calculating Time Complexity**

Given two time complexity functions \( f(n) \) and \( g(n) \):

1. If an algorithm grows at \( f(n) + g(n) \), then the it grows at whichever rate is faster (domination).
2. If an algorithm grows at \( f(n) \times g(n) \), then it compounds the growth of both functions.

Here we also want to make the distinction between terminology used to describe algorithmic growth:

- Efficient = Slower Growing = Faster Runtime
- Not Efficient = Faster Growing = Slower Runtime

**Exercise 4.41** Which of the two algorithms is more efficient?

- **Algorithm A**: \( f(n) = n \times \log(n) \times n^2 \)
- **Algorithm B**: \( g(n) = n^2 \times n \)

**Exercise 4.42** Which of the two algorithms is more efficient?

- **Algorithm A**: \( f(n) = n^3 + \log(n) \)
- **Algorithm B**: \( g(n) = n^2 \times \log(n) \)

---

Exercise 4.43 Which of the two algorithms is more efficient?

Algorithm A: \( f(n) = n(\log(n) + n) \)
Algorithm B: \( g(n) = \log(n)(100 + n^2) \)

Exercise 4.44 Which of the two algorithms is more efficient?

Algorithm A: \( f(n) = (2^n + n \log(n))(n^3 + 2n + 2) \)
Algorithm B: \( g(n) = (n^4 + 10000n^3 + 90)(10n^4 + n^8 + 32) \)
4.6.3 Unreasonable Or Uncomputable?

Here I want to make a quick distinction between unreasonable and uncomputable problems.

1. **Unreasonable**: A problem that will take too long to compute (years or even millenniums).

2. **Uncomputable**: A problem that is impossible to compute. Ever.

In the previous section we saw that polynomial-runtime algorithms ($n^2$) will reach unreasonable runtimes when the input size gets large. Exponential ($2^n$) and factorial ($n!$) runtimes quickly explode into numbers that exceed the number of atoms in the universe.

However, even if an algorithm has a runtime in the billions, or even $10^{100}$, it is still computable. A computer can solve the problem, it’s at an unreasonable length of time.

But it’s technically possible.

We call these problems computable problems. We can write an algorithm and the computer can solve it. But there is set of problems that is absolutely impossible for a computer to solve. Not because it will take too long, but because it is mathematically impossible. It doesn’t matter what you do,

They are impossible.

But what if we used supercomputers and —

Still impossible.

What about quantum —

No.

Remember how Alan Turing disproved the entscheidungsproblem and showed that there are infinitely more computer problems than there are computer programs to solve them? These are those impossible problems.

One of the most famous uncomputable problems is called the Halting Problem.

---

### The Halting Problem

Given an algorithm $H$, write an algorithm to determine if $H$ will halt (if the program will eventually end).

---

It’s a little beyond this course to explain why the halting problem is uncomputable. However the YouTube channel Computerphile has a great video simplifying the proof using terrific animations. If you’re interested, search up Turing & The Halting Problem - Computerphile on YouTube.
4.7 Recursive Algorithms

In this section we will examine the last question posed in the introduction of algorithms:

What are computers so powerful?

We will look at a type of algorithm that uses a process that us humans cannot utilize to its full potential. This process is called recursion.

---

**Recursion (General Idea)**

Go to Section 4.7 on page 44 for the definition of recursion.

Or alternatively, search “recursion” on Google.

---

All joking aside, it is quite difficult to define recursion in a way that makes sense to humans. It is a process that is very non-intuitive for humans to grasp when first introduced. Here let’s actually formally define it:

---

**Recursion (Formal)**

A process that is defined by means of self-referencing.

---

and in a very similar, ambiguous fashion, we can also define what a recursive algorithm is:

---

**Recursive Algorithm**

A self-referencing algorithm where a small change or calculation is applied every self-reference.

---

You may get the idea that recursion is the same thing as loops. While it is true that all recursive algorithms can be re-implemented as a loop, there are some key differences that make recursion a much more powerful process.

To help understand what recursion is and help illustrate how to write a recursive algorithm, let’s imagine how a forgetful student approaches the following problem with the help of an all-knowing oracle.

**Example 4.16** Given an array of integers $A$, write an algorithm to find the sum from $n$ to down to 1. For example, if $n = 5$:

$$5 + 4 + 3 + 2 + 1$$

*Student:* Ok let’s give it a try:

---

**Input:** An array of integers $A$.

**Output:** An integer sum.

---

*Student:* ... I’ve forgotten how to do loops.
Oracle: I am the oracle. I am an omniscient being here to give you any question you ask me.

Student: First a fairy and now an oracle. What a weird day. Well, can you tell me what is \( \text{sum}(5, 4, 3, 2, 1) \)?

Oracle: I could, but that would defeat the purpose of the problem.

Student: You’re not omniscient then.

Oracle: I am. To help you I can tell you what \( \text{sum}(4, 3, 2, 1) \) is.

Student: Oh well that is actually helpful. Because if you tell me that, I can answer the problem with the following simple calculation:

\[
\begin{align*}
\text{sum}(5, 4, 3, 2, 1) &= 5 + \text{sum}(4, 3, 2, 1) \\
\text{sum}(4, 3, 2, 1) &= 4 + \text{sum}(3, 2, 1) \\
\text{sum}(3, 2, 1) &= 3 + \text{sum}(2, 1) \\
\text{sum}(2, 1) &= 2 + \text{sum}(1) \\
\end{align*}
\]

Student: Oh well that is actually helpful. Because if you tell me that, I can answer the problem with the following simple calculation:

\[
\begin{align*}
\text{sum}(4, 3, 2, 1) &= 4 + \text{sum}(3, 2, 1) \\
\text{sum}(3, 2, 1) &= 3 + \text{sum}(2, 1) \\
\text{sum}(2, 1) &= 2 + \text{sum}(1) \\
\end{align*}
\]

Student: Well, call you tell me what is \( \text{sum}(3, 2, 1) \)?

Oracle: I could, but that would defeat the purpose of the problem.

Student: You’re not omniscient then.

Oracle: I am.

Student: Huh, déjà vu.

Oracle: To help you I can tell you what \( \text{sum}(3, 2, 1) \) is.

Student: Oh well that is actually helpful. Because if you tell me that, I can answer the problem with the following simple calculation:

\[
\begin{align*}
\text{sum}(3, 2, 1) &= 3 + \text{sum}(2, 1) \\
\text{sum}(2, 1) &= 2 + \text{sum}(1) \\
\end{align*}
\]

Student: Well, call you tell me what is \( \text{sum}(2, 1) \)?

Oracle: I could, but that would defeat the purpose of the problem.

Student: You’re not omniscient then.

Oracle: I am.

Student: Huh, déjà vu.

Oracle: To help you I can tell you what \( \text{sum}(2, 1) \) is.

Student: Oh well that is actually helpful. Because if you tell me that, I can answer the problem with the following simple calculation:

\[
\begin{align*}
\text{sum}(2, 1) &= 2 + \text{sum}(1) \\
\end{align*}
\]

Student: Well, call you tell me what is \( \text{sum}(1) \)?

... wait, why am I even asking you? \( \text{sum}(1) \) is just 1! I don’t need to do any calculations! If I know what \( \text{sum}(1) \) is, I now know...
\[
\begin{align*}
\text{sum}(2, 1) &= 2 + \text{sum}(1) \\
&= 2 + 1 \\
&= 3
\end{align*}
\]

**Student:** And if I know what \(\text{sum}(2, 1)\) is, I now know...

\[
\begin{align*}
\text{sum}(3, 2, 1) &= 3 + \text{sum}(2, 1) \\
&= 3 + 3 \\
&= 6
\end{align*}
\]

**Student:** And if I know what \(\text{sum}(3, 2, 1)\) is, I now know...

\[
\begin{align*}
\text{sum}(4, 3, 2, 1) &= 4 + \text{sum}(3, 2, 1) \\
&= 4 + 6 \\
&= 10
\end{align*}
\]

**Student:** And if I know what \(\text{sum}(4, 3, 2, 1)\) is, I now know...

\[
\begin{align*}
\text{sum}(5, 4, 3, 2, 1) &= 5 + \text{sum}(4, 3, 2, 1) \\
&= 5 + 10 \\
&= 15
\end{align*}
\]

**Student:** The answer is 15!

**Oracle:** You didn’t need me to begin with. All you needed was **recursion**.

Using this as inspiration, our student writes the following algorithm:

---

**Algorithm 21 sum**

**Input:** An integer \(n\).

**Output:** An integer sum.

\[
\begin{align*}
\text{if } (n == 1) & \text{ then} \\
\quad \text{return } 1 \\
\text{end if} \\
\text{return } n + \text{sum}(n - 1)
\end{align*}
\]

---

This algorithm, in my personal opinion, is elegant and beautiful. This is **recursion**.
To help you with writing recursive algorithms, here are the three main questions you want to answer:

1. **How can I break the problem down into a smaller similarly-computable component?**
   In the above example, it is breaking down \( \text{sum}(n) \) into \( \text{sum}(n-1) \).

2. **What extra computation do I need to do to combine the answer of the smaller component?**
   In the above example, it is \( n + \text{sum}(n-1) \).
   This is called the **recursive step**.

3. **What is the smallest component I can break down?**
   In the above example, it is simply when \( n == 1 \). There are no calculations to be made, the sum is trivially 1.
   This is called the **base case**.

**Exercise 4.45** Given an integer \( n \), write a recursive algorithm to find \( n! \).

**Exercise 4.46** Given an odd integer \( n \), write a recursive algorithm to find the sum of all odd integers from 1 to \( n \).
With recursion, we can rewrite some of the previous algorithms we learned in a much more elegant manner. Notice the return of logical operators! These algorithms also require a bit more advanced pseudocode which you will learn in INFO-210.

Example 4.17 Write a recursive algorithm for linear search.

Algorithm 22 linear_search
\[\text{Input: } \text{An array of integers } A \text{ of size } n \text{ and a target } k.\]
\[\text{Output: } \text{true if and only if } k \in A.\]
\[\text{if } (n == 1) \text{ then}\]
\[\quad \text{return } A[0] == k\]
\[\text{end if}\]
\[\text{return } (A.pop() == k) \lor \text{linear_search}(A)\]

Example 4.18 Write a recursive algorithm for binary search.

Algorithm 23 binary_search
\[\text{Input: } \text{An array of integers } A \text{ of size } n \text{ and a target } k.\]
\[\text{Output: } \text{true if and only if } k \in A.\]
\[\text{if } (n == 1) \text{ then}\]
\[\quad \text{return } A[0] == k\]
\[\text{end if}\]
\[\text{if } ([A[n/2] > k) \text{ then}\]
\[\quad \text{return } (A[n/2] == k) \lor \text{binary_search}(A[0:(n/2)])\]
\[\text{else}\]
\[\quad \text{return } (A[n/2] == k) \lor \text{binary_search}(A[(n/2)+1:n])\]
\[\text{end if}\]

Example 4.19 Write a recursive algorithm for insertion sort.

Algorithm 24 insertion_sort
\[\text{Input: } \text{An array of integers } A \text{ of size } n.\]
\[\text{Output: } A \text{ sorted array.}\]
\[\text{if } (n == 1) \text{ then}\]
\[\quad \text{return } A\]
\[\text{end if}\]
\[\text{return } \text{insert}(A.pop()), \text{insertion_sort}(A)\]

Be aware that not all algorithms are recursion friendly! It is very difficult to convert selection sort into a recursive algorithm. Likewise, some algorithms like binary search are much easier to write recursively than iteratively (using a loop).

Computers are inherently recursive. It is much easier for a computer to solve a problem recursively because it is physical hardware is designed to perform recursion. Many important computation problems are difficult for humans to solve to do because our brains are not built for recursion, but to a computer’s naturally recursive “brain”, the problem becomes very simple to solve.