2 Algorithms

"We are artists ourselves – our medium is that of the computer – and we should practice our craft as such."

After venturing through propositional logic, we now will explore how logic is applied to computing. This topic on algorithms aims to develop your "computational brain" as opposed to your "human brain."

The goal of this topic is <u>not</u> to have you become proficient programmers. You'll have plenty of time to develop your programming skills in I210, I211, and future courses. Our goal here is to introduce foundational computing concepts for *algorithmic thinking*. You'll notice that two core concepts in algorithms – **conditional branching** and **loops** – have propositional logic inherently built into them.

We will be focused on understanding exaclty how to instruct computers to solve simple computational problems through providing a series of precise computing instructions. To do so, we will need to learn the basic operations of a computer and how to organize these operations properly. Here is a quick summary of our journey ahead:

- How does a computer remember information? (Variable assignment, arrays)
- How does a computer "compute"? (Numerical operations)
- How does a computer make simple decisions? (Comparative operations and conditional branching)
- How do we more efficiently solve large problems with a computer? (Looping)
- How does a computer solve a problem of *any* size? (Dynamic memory)
- What are more advanced algorithmic techniques to solve more complex problems? (Flags and nested loops)
- What are common computation problems and the algorithms used to solve them? (Search and sorting algorithms)

By the end of this topic you will have developed a strong groundwork for algorithmic thinking. In the beginning, keep in mind that the <u>language of computers will feel very foreign to you</u>. Approach it with patience and perseverance, and you will soon discover the fascinating world of computation. Let's get started!

2.1 What are Algorithms?

If you've ever made any baked goods, made Lego models, or assembled a piece of IKEA furniture, then you will be innately familiar with what an **algorithm** is.

> **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.
>
> **Exercise 2.1** *On a separate sheet of paper, write a recipe for a peanut butter and jelly sandwich.*

The word "**algorithm**" derives from a mangled transliteration of *al-Khwarizmi* (c. 780 – 850), a great Persian mathematician and scholar. In 825, *al-Khwarizmi* 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.

In this sense, you have been using **algorithms** throughout your math career without realizing it. Do the following calculation:

Exercise 2.2 *Calculate the sum from* 1 *to* 10.

Now imagine you're tutoring an elementary school student who easily gets overwhelmed by addition. Add 1 to 10!? So many numbers!?

Exercise 2.3 *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 to get the right answer every time. Unfortunately, in many ways, a computer is dumber than an elementary school student; a computer is more 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.

Before we can start solving computation problems with a computer, we need to learn the language of computers.

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. Some you may be familiar with, some you may not.

Javascript (INFO-101, web scripting)

```
var sum = 0;
for (var i = 1; i <= 10; ++i) {
  sum += i;
}
```

Python (INFO-210, general-purpose)

```
sum = 0
for i in range(0, 10):
    sum += i + 1
print(x)
```

C++ (standard language in industry)

```
int sum = 0;
for (int i = 1; i <= 10; ++i) {
    sum += i;
}
```

Racket (CSCI-211, a recursively structured language)

```
(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 (a concurrent functional language)

```
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])
```

With so many languages, which one do we choose? You may notice that **Python**, **Javascript**, and **C++** are all very similar; these are all **object-oriented programming languages** – a standard in many universities and companies.

In this course, we will be learning **Coral**, a simplified **object-oriented** language that is terrific for working on foundational algorithmic thinking. It is not a robust language, but it lends itself well into **Python**, **Javascript**, **C++**, and other **object-oriented** languages.

We will writing a considerable amount of Coral instructions on the Coral Simulator. It'll be a good idea to bookmark this website for easy access for when we need to use it.

2.2 Algorithm Basics

Consider the following calculation:

Exercise 2.4 *Calculate* $2 + 5$

When we do simple calculations we juggle numbers in our head with ease. This juggling of numbers is not as inherently simple for computers. Let's break down what we do when we do $2 + 5$ into more granular steps:

Example 2.1 *Calculate* $2 + 5$ *step-by-step.*

1. Read the number 2.
2. Remember the number 2 in one part of your brain.
3. Read the number 5.
4. Remember the number 5 in a second part of your brain.
5. Add $2 + 5 = 7$. Remember the solution 7 in a third part of your brain.
6. Say 7 out loud for the answer.

As much as this feels tedious, this is exactly how a computer thinks. A task as easy as $2 + 5$ that you can do in your head as a human will require 6 discrete steps for a computer. This definitely feels more complicated to do — which it is — however the one advantage of a computer is that a computer will never get tired. We will use this to our advantage when we solve for larger problems later.

For now, let's what the 6-step process looks like when we write it as an algorithm in Coral. Here we will look at the 3 basic components of an algorithm: **variables**, **input**, and **output**.

2.2.1 Variables and Data Types

When we were calculating $2 + 5$ in our head, we were holding 3 numbers in our head throughout this calculation. The first number is 2, the second number is 5, and the third number is the sum $2 + 5 = 7$. The equivalent action of remembering a number for a computer is assigning values to variables.

> **Variable Declaration**
> A symbol used to represent a discrete unit of storage. This can come in the form of letter or singular words. We declare a variable by stating the **data type** followed by the **variable name**:
>
> `<DATA TYPE> <VARIABLE NAME>`

Here we also need to look at what a **data type** is.

> **Data Type**
> A classification of data that specifies what kind of data the computer is expected to store.

There are different data types depending on the programming language. Coral only has two main data types: **integers** and **floats**.

> **Integer**
> The **data type** representing whole numbers (positive and negative).

> **Float**
> The **data type** representing rational numbers (decimals).

You will be introduced to more **data types** in I210 (Python). The types of basic data types differ between languages, but for Coral we will only be concerend with **integers** and **floats**.

2.2.2 Variable Value Assignment

Let's now try to "remember" 2 and 5 in Coral.

Example 2.2 *Write an algorithm in Coral to calculate* $2 + 5$.

We are adding two numbers together, which means we need two units of storage (variables). Since both numbers are **integers**, we'll need to use the **integer** data type.

Let's declare our first unit of storage for our first integer. Because it is an **integer**, we need to declare it as an **integer data type**. Let's assign this unit of storage the **variable name** `a`:

```
integer a
```

Let's also declare our second unit of storage for our second integer. Because it is an **integer**, we need to declare it as an **integer data type**. Let's assign this unit of storage the **variable name** `b`:

```
integer a
integer b
```

Let's take a look at our current algorithm in **Coral**. Look at the top right hand corner of the Coral Simulator:

```
1 integer a
2 integer b
```

Variables

0	a	integer
0	b	integer

We indeed have two variables declared both of types integers. But notice that the **values** they're holding are both 0. This is not what we want! We want `a` to hold the value of 2 and `b` to hold the value of 5. This is because when we first **declare a variable**, the value of the variable is assigned to 0 by default (for the integer data type). This means after we declare our variables, we need to **assign** a value to them.

Value Assignment
The **value** of a **variable** is **assigned** with the equals sign `=` operator with the **variable name** on the left and the **value** on the right.

```
<VARIABLE NAME> = <VALUE>
```

We can now assign the values of 2 and 5 to our variables.

```
integer a
integer b

a = 2
b = 5
```

If we check our variable values on Coral, we can confirm that `a` and `b` contain the correct values:

```
1 integer a
2 integer b
3
4 a = 2
5 b = 5
```

Variables

2	a	integer
5	b	integer

Now that we have the two numbers stored as variables, let's now add them together. Adding the variables `a` and `b` behaves as you would expect:

```
a + b
```

However, we now have a new problem. `a` + `b` will yield $2 + 5 = 7$, however the number `7` is just floating in the void and disappears. We need to remember the sum! This means we need a new variable to to capture the computation result. Let's declare a new **integer** variable called `sum` and assign it to the sum.

```
integer a
integer b
integer sum

a = 2
b = 5

sum = a + b
```

Let's type this into Coral and see what we get:

Variables

2	a	integer
5	b	integer
7	sum	integer

Exercise 2.5 *Write an algorithm in Coral to calculate* $1 + 7 + 2 + 9$

2.2.3 Reading in Inputs

The algorithm you wrote in the previous exercise will calculate the sum of $1, 7, 2$, and 9 correctly, but it will only calculate the sum of $1, 7, 2$, and 9 and nothing else. Instead of writing a custom algorithm for every type of calculation we encounter, we want to write a more generalized algorithm that will do any kind of calculation. This way our algorithm will work for any number it receives. Instead of *hard-coding* numbers into our algorithm, we will want to read **input values**.

Input
A type of instruction where the algorithm will receive information (numbers, letters, strings) from the user.

In Coral, this is achieved with the `Get next input` instruction:

```
integer x

x = Get next input
```

Example 2.3 *Write an algorithm in Coral to calculate the sum of three input integers.*

We are going to calculate the sum of three integers, so let's declare three integer variables. We will also need an extra variable to store the sum, so let's declare an integer variable for the sum as well.

```
integer a
integer b
integer c
integer sum
```

We do not immediately know the values of the three integers we are going to be summing up – the values will be provided by the user. In Coral, the input values are entered in the *"Input"* text box. Let's go ahead and enter some example numbers:

Input

9 10 1729

When we type these three numbers in, Coral will queue these input values for when we use the `Get next input` command. The first time we use `Get next input` Coral will fetch the value 9; the second time we use `Get next input` Coral will fetch 10; the third time we use `Get next input` Coral will fetch 1729. Let's use `Get next input` command and **assign** our variables to their values.

```
integer a
integer b
integer c
integer sum

a = Get next input
b = Get next input
c = Get next input
```

Let's also run this in the Coral Simulator and make sure our variables are assigned the correct values from our input:

```
1 integer a
2 integer b
3 integer c
4 integer sum
5
6 a = Get next input
7 b = Get next input
8 c = Get next input
```

Variables

9	a	integer
10	b	integer
1729	c	integer
0	sum	integer

Input

9 10 1729

Now we are ready to calculate our sum:

```
integer a
integer b
integer c
integer sum

a = Get next input
b = Get next input
c = Get next input

sum = a + b + c
```

Once again, let's run this in the Coral Simulator and make sure our `sum` variable is assigned the correct value:

```
1  integer a
2  integer b
3  integer c
4  integer sum
5
6  a = Get next input
7  b = Get next input
8  c = Get next input
9
10 sum = a + b + c
```

Variables

9	a	integer
10	b	integer
1729	c	integer
1748	sum	integer

Input

9 10 1729

2.2.4 Writing out Outputs

We are faced with one last issue. Currently, our algorithm reads in 3 input values from the user, it calculates the correct sum, and it stores the value in the variable `sum`. We're missing one last piece: we need the algorithm to report the answer back to us! This is were we need to tell the algorithm to **output** the sum back to us.

> **Output**
> A type of instruction where the algorithm will output information (numbers, letters, strings) back to the user.
>
> In Coral, this is achieved with the `Put <VALUE> to output` instruction:
>
> ```
> Put <VALUE> to output
> ```

Let's finish our algorithm by instructing the computer to report the contents of the `sum` variable back to us:

```
integer a
integer b
integer c
integer sum

a = Get next input
b = Get next input
c = Get next input

sum = a + b + c

Put sum to output
```

And once again, let's check in the Coral Simulator that the correct value is being outputted in the **"Output"** text box:

```
1  integer a
2  integer b
3  integer c
4  integer sum
5
6  a = Get next input
7  b = Get next input
8  c = Get next input
9
10 sum = a + b + c
11
12 Put sum to output
```

Variables

9	a	integer
10	b	integer
1729	c	integer
1748	sum	integer

Input

9 10 1729

Output

1748_

2.2.5 Arrays

For now we've only been adding two or three numbers together. What if we scale up the size of our problem? Let's say we want to add up the sum of 10 input integers. Our algorithm will be:

```
// Declare variables
integer a
integer b
integer c
integer d
integer e
integer f
integer g
integer h
integer i
integer j
integer sum

// Read in inputs
a = Get next input
b = Get next input
c = Get next input
d = Get next input
e = Get next input
f = Get next input
g = Get next input
h = Get next input
i = Get next input
j = Get next input

// Calculate sum
sum = a + b + c + d + e + f + g + h + i + j

// Output sum
Put sum to output
```

This seems rather excessive 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 unique variables for every single input?

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

Array
A single contiguous section of memory used as data storage and retrieval.

In Coral, we declare a static array (an array of a fixed size) as a variable:

```
<DATA TYPE> array(<SIZE>) <VARIABLE NAME>
```

The memory contents of the array are accessed with the indexing operator (square brackets):

```
<VARIABLE NAME>[<INDEX>]
```

With this we can alleviate some of this redundancy. Consider the following integer array `A` of size 5 in Coral.

```
// Declare an integer array of size 5
integer array(5) A

// Populate the array with integers
A[0] = 17
A[1] = 5
A[2] = 19
A[3] = 13
A[4] = 61
```

For ease of reading, we will represent the array written as a list:

```
A = [ 17, 5, 19, 13, 61 ]
```

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

```
A = [  17,  5,  19,  13,  61  ]
        ↑   ↑    ↑    ↑    ↑
Index   0   1    2    3    4
```

We say that the member stored at the 0th location of the array `A` is 17. Likewise, we say that the member stored at the first location is 5. This process of **accessing** the contents of an array is called **indexing**. The syntax for accessing an array is to use the square brackets:

$$\texttt{A[0]} = 17$$
$$\texttt{A[1]} = 5$$

Exercise 2.6 *Given the array `A` defined above, what is `A[2]`, `A[3]`, and `A[4]`? What about `A[5]`?*

Let's use an array to sum up five input integers:

Example 2.4 *Given five input integers, write an algorithm in Coral to find the total sum.*

```
// Declare variables
integer array(5) A
integer sum

// Read in inputs
A[0] = Get next input
A[1] = Get next input
A[2] = Get next input
A[3] = Get next input
A[4] = Get next input

// Calculate the sum
sum = A[0] + A[1] + A[2] + A[3] + A[4]

// Output the sum
Put sum to output
```

Exercise 2.7 *Given five input integers, write an algorithm in Coral to read these integers into an array and find the average of the integers in the array.*

This still feels quite tedious since we need to call the `Get next input` instruction multiple times for each number we're reading in. We will see the solution to this headache when we look at **loops**. For now, we have enough foundational knowledge on algorithms to move forward to look at some of the more interesting algorithmic features.

2.3 Swapping and Tracing

Let's do a classic exercise on **variable assignment** to make solidify the algorithm basics we've covered so far before we move on. The beginning of the algorithm is given to you. You only need to fill out the section with the "`TODO`" comment (you will also need to declare a new variable, but don't worry about that too much for now).

Exercise 2.8 *Write an algorithm in Coral to swap the values of the two variables* **x** *and* **y** *and output them.*

```
// Declare variables
integer x
integer y

// Assign values
x = 17
y = 29

// Output values before swapping
Put "Before swapping\n" to output
Put "x is: " to output
Put x to output
Put ", y is: " to output
Put y to output

// TODO: Swap the values of the first and second variable











// Output values after swapping
// the "\n" is the "new line" character just for aesthetics
Put "\nAfter swapping\n" to output
Put "x is: " to output
Put x to output
Put ", y is: " to output
Put y to output
```

What are the values of `x` and `y` before swapping?

`x =` `y =`

What are the **expected** values of `x` and `y` after swapping?

`x =` `y =`

Let's consider the following solution for swapping the values of two variables:

```
x = y
y = x
```

If we use the above logic for swapping and run the algorithm in Coral, our output will be:

```
Output
Before swapping
x is: 17, y is: 29
After swapping
x is: 29, y is: 29_
```

Our output values are not what we expected. `x = 29` is correct, but `y` should be `17` and not remain the same value at `29`. To see why this is the case, we need to do a **trace** of our algorithm:

> **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 accurately solves 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 keep track of our two variables `x` and `y`. In the very beginning, the two variables are initialized to `17` and `29` respsectively.

```
x = 17
y = 29
```

x	17
y	29

Because an algorithm runs **sequentially** – line-by-line one at a time – let's see what happens after our first line `x = y`. Here we assign the value of `x` to be `27`, the value of `y`.

```
x = y
```

x	~~17~~	29
y	29	29

Here we see an error in our logic. When the line `x = y` is run, the value of `x = 17` is overwritten and lost forever. We need a way to temporarily store the value of `x` for later use. To do so, let's create another variable for this express purpose.

```
// Declare variables
integer x
integer y
integer temp
```

Now we can use the `temp` variable to hold onto the value of `x` so we don't lose track of it, then use it again to assign the value of `y`.

```
// TODO: Swap the values of the first and second variable
temp = x
x = y
y = temp
```

We can use the Coral Simulator to see if we get the right solution, but let's do this by hand and trace through our algorithm. First, the variables are initialized to their starting value (`temp` is 0 by default):

```
x = 17
y = 29
```

x	17
y	29
temp	0

We then assign the value of `temp` with the value of `x`:

```
temp = x
x = y
y = temp
```

x	17	17
y	29	29
temp	~~0~~	17

With the original value of `x` safely stored in the `temp` variable, we can assign `x` the value of `y` without worry:

```
temp = x
x = y
y = temp
```

x	17	~~17~~	29
y	29	29	29
temp	~~0~~	17	17

And finally, let's assign `y` the original value of `x` that is stored in `temp`:

```
temp = x
x = y
y = temp
```

x	17	~~17~~	29	29
y	29	29	~~29~~	17
temp	~~0~~	17	17	17

And here we have successfully swapped the values of `x` and `y`. Write the entire algorithm in the Coral Simulator and see for yourself!

2.4 Algorithm Errors

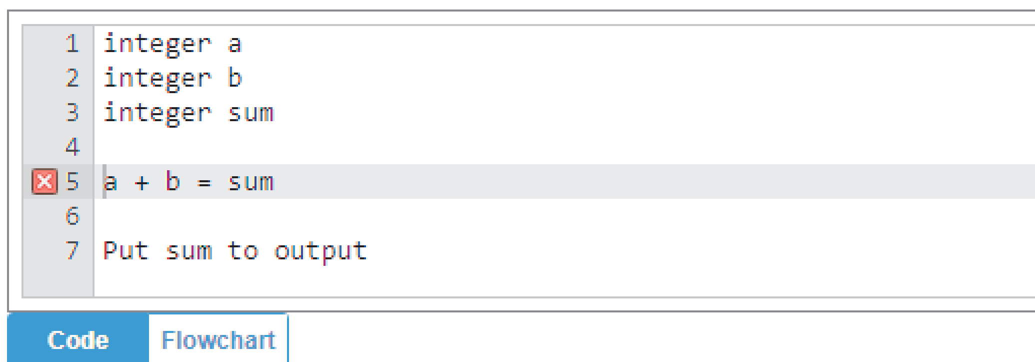
Computers are very picky "eaters". They will accept algorithms if they are "cooked" a specific way. If you don't prepare an algorithm just the way they like it, they'll reject it like a kid does broccoli.

2.4.1 Syntactic Errors

Exercise 2.9 *Try the following algorithm in the Coral Simulator*

```
// Declare variables
integer a
integer b
integer sum

// Initialize the variable values
a = 2
b = 5

// Add a and b together
a + b = sum

// Output
Put sum to output
```

Coral is going to complain about the above algorithm because it's not written just the way they like it:



Coral is nice enough to tell us what they were upset about by marking the incorrectly-written line and giving us an error message: *"Line 5: Must assign to a variable."* The problem here is that we did not use the **assignment operator** (equals sign) correctly. Recall that the target variable for storage is on the left-hand side of the equals sign, and the value to store is on the right-hand side. To fix this, we should write:

```
sum = a + b
```

These errors where we typed something that the computer did not expect are called **syntax errors**.

> **Syntax and Syntax Errors**
> **Syntax** is the pre-definied set of rules on what structure of symbols or words are allowed in the algorithm. A **syntax error** is an error in which the algorithm contains an expression that is not allowed by the pre-defined **syntax** of a language.

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

$$(p \wedge \wedge q$$

and how English also has syntactic rules:

"Speak like Yoda, we do not."

Coral also has its own set of syntax rules. As we learn more features of Coral in the upcoming sections on **conditional branches**, **loops**, and **arrays**, keep in mind that you have to write these lines exactly as specified.

2.4.2 Semantic Errors

Not all errors are syntactic, nor will the computer catch all the errors for you and complain at you to fix them. Consider the following problem and the algorithm to solve it:

Example 2.5 *Write an algorithm in Coral to find the difference between two integer inputs. Subtract the second input integer from the first.*

```
// Declare variables
integer a
integer b
integer diff

// Get inputs
a = Get next input
b = Get next input

// Calculate difference
diff = b - a

// Output
Put diff to output
```

If we enter two input values in Coral and run the above algorithm, the algorithm will run without any errors. However, there is an error in the algorithm regardless.

Exercise 2.10 *What is the error in the above algorthm?*

There errors are errors in the logic of our algorithm and not with how we write them. Our syntax is correct, but the computation work is incorrect. These are called **semantic errors**.

> **Semantic Errors**
> An error in the computation logic of an algorithm, resulting in the algorithm not consistently producing the correct answer.

You'll need to be wary of making sure both your **syntax** and **semantic** (logic) are correct as your write your algorithm. This is best achieved by frequent checking, and more checking.

2.5 Testing and Organization

Up until now, and for the large majority of this topic, the problems we are writing algorithms for are small in scale. The amount of inputs are manageable, there's only one calculation to do, and the output is simply reporting the answer. However, the type of computation problems you will be asked to solve in later courses will increase in difficulty quickly. There will be multiple input types, computation and logic will be multi-staged and complex, and outputs will need formatting.

Before we get overwhelmed by complex computation problems, we will want to develop some good algorithmic practices from the start. These may feel redundant and unnecessary for the problems we are working on this course, however these habits will prove to be invaluable in later courses. Take them into serious consideration and you will be graced by their utility later.

2.5.1 Check, Check, Check... and Check Again

You may notice that we are using the Coral Simulator very frequently, almost to the point of excess. This may seem unnecessary, but this is a very, very good practice when writing algorithms. You want to check your algorithm's behavior frequently as you write each step of the algorithm, **do not** do your checking all at once at the end. If you check your algorithm only once at the end, you will likely run into all sort of weird errors.

A chef will taste their food throughout the cooking process to ensure the seasoning is correct. A still-life painter will frequently take a step back to look at their painting to check their piece looks they way they want it to from afar. A composer will play snippets of their music aloud to check that it is melodious to listen to. In many ways, we are artists ourselves – our medium is that of the computer – and we should practice our craft as such. It would be unwise for artists to check their work at the very end, and we are no exception.

Of course, there are exceptions to this rule, but they are exceptions. Mozart can write a symphony for breakfast without any checking; Kandinsky would paint feverishly on his abstract works. Likewise, an experienced programmer can write 10, 20, or hundreds of lines of an algorithm without needing to run their code. However, this comes with practice – you'll get there in time. For now, be patient, be diligent, and take it step-by-step.

2.5.2 Organizing Your Algorithm

Generally speaking, algorithms follow a basic three-part structure. You can think of this as the head, body, and tail of an algorithm:

Head: Read in the input data (and storing them properly in variables).
Body: Apply computation and logic to the data (using computational operators and logic structures).
Tail: Output the results.

These three components should be constructed in that order. 1) Read the inputs in, then 2) work on the computation logic to calculate the answer, and finally 3) output the answer. There are times when the output instructions are folded into the logic of the algorithm, but this concept of compartmentalizing your algorithm still holds. A good rule to follow is:

Only proceed to the next portion of your algorithm after you've ensured that your previous portion are working properly as intended (through excessive checking).

This way you can write each section of your algorithm with the peace of mind that you are error-free in your previous sections.

To help us organize our algorithms' sections and keep track of our work, we can use **comments** to annotate our algorithm.

Comments
Portions of your algorithm that will automatically be ignored by the computer. The computer will skip over these lines.

In Coral, we denote comment lines with the double front-slash command `//`:

```
// This line is a comment and will be ignored
```

Let's go back and organize our summing example with comments to help us navigate what we wrote.

Example 2.6 *Write an algorithm in Coral to calculate the sum of three input integers (this time with comments!)*

```
// Declare our variables: 3 integers and a sum
integer a
integer b
integer c
integer sum

// Read the three input integers
a = Get next input
b = Get next input
c = Get next input

// Compute the sum
sum = a + b + c

// Output the solution
Put sum to output
```

As our computation problems get more complex and require more steps, comments will help us organize our thoughts and algorithmic structure[6]. If you're working in teams, your comments will also help your teammates understand each step of your algorithm as they're not mind-readers. If you don't comment your algorithm in industry, you will be fired very, very quickly.

For this course, you might feel that commenting out the **input**, **logic**, and **output** portions of your algorithms are unnecessary. This is true to an extent, but only because the problems we are working on are simplistic by design. In I210 and beyond you will be working with large amounts of input data, complex logic structures, and picky output formatting. Making sure your algorithm works on these step-by-step and not be intertwined with one another will be to your great benefit.

[6]Comments can also be a source of great comic relief.

2.6 Algorithmic Operators

We've seen the algorithmic instructions on how to calculate the sum (`+`) and difference (`-`) between two variables. Let's take a comprehensive look at the other computation instructions available for us to use. Generally speaking, computers can perform two types of operations: **numerical** and **comparative** operations.

Numerical Operations
Operations that instruct the computer to perform numerical calculations. These include:

Operation	Symbol
Addition	`x + y`
Subtraction	`x - y`
Multiplication	`x * y`
Division	`x / y`
Modulo (remainder)	`x % y`

Be aware that there are operations that we have used in mathematics that we <u>do not</u> have access to without the use of in Coral. You <u>can not</u> use the following operations:

Unavailable Operations	Example
Exponents or power	x^2
Square roots	$\sqrt{x}$
Logarithmic	$\log(x)$

Comparative operators will also be familiar to you. A solid grasp of how **logical operators** work lends itself perfectly with **comparative operators** in algorithms.

Comparative (Boolean) Operations
Operations that instruct the computer to perform **Boolean** (true or false) calculations. These operators will always output a **true** or a **false**:

<table>
<tr><th>Operation</th><th>Symbol</th><th>Description</th></tr>
<tr><td>Equality</td><td><code>x == y</code></td><td>Will return <code>true</code> if x and y are the same value.</td></tr>
<tr><td>Inequality</td><td><code>x != y</code></td><td>Will return <code>true</code> if x and y are not the same value.</td></tr>
<tr><td>Greater than</td><td><code>x > y</code></td><td>Will return <code>true</code> if x is strictly greater than y.</td></tr>
<tr><td>Greater or equal</td><td><code>x >= y</code></td><td>Will return <code>true</code> if x is greater than or equal to y.</td></tr>
<tr><td>Less than</td><td><code>x < y</code></td><td>Will return <code>true</code> if x is strictly less than y.</td></tr>
<tr><td>Less or equal</td><td><code>x <= y</code></td><td>Will return <code>true</code> if x is less than or equal to y.</td></tr>
<tr><td>Logical AND</td><td><code>x && y</code></td><td>Will return <code>true</code> if x and y are both true.</td></tr>
<tr><td>Logical OR</td><td><code>x || y</code></td><td>Will return <code>true</code> if either x or y are true.</td></tr>
</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.

Among these operators, we need to look at more closely at the **division** and **modulo** operators in particular as they behave a little differently in computation.

2.6.1 Computational Division

Let's take a look at the following example. Pay close attention to the **data type** of our variables.

```
// Declare variables
integer a
integer half

// Instantiate values
a = 5

// Compute
half = a / 2

// Output
Put half to output
```

Exercise 2.11 *What will the above algorithm output?*

Now consider the following. Once again, pay attention to the **data types**.

```
// Declare variables
float a
float half

// Instantiate values
a = 5

// Compute
half = a / 2

// Output
Put half to output
```

Exercise 2.12 *What will the above algorithm output? What is the difference to the first variation?*

Here we see an important behavior when it comes to **integer division** in computation. Because the **integer data type** can only hold **integer** values (whole numbers), if we try to store a value with decimals into an **integer data type**, the computer will truncate off the decimal values.

Exercise 2.13 *Compute the following integer divisions:*

`4 / 2` =	`7 / 2` =	`38 / 3` =	`1729 / 9` =

While this may seem like this will cause issues because we are eliminating information, many computer problems cannot have fractional values in their computation. Integer division allows us to create algorithms that fit neatly into the discrete behaviors of a computer.

2.6.2 Modulo Operator

If integer division truncates the decimal values, what if that is information we need? We don't just throw them away right? Not to worry, the **modulo** operator calculates the remainder of an **integer division** computation. The **modulo** operator `%` will calculate only the remainder of a division.

Exercise 2.14 *Compute the following modulo calculations:*

`4 % 2` =	`7 % 2` =	`38 % 3` =	`1729 % 9` =

Similar to **integer division**, the usefulness of **modulus** is not immediately apparent. Let's explore one of the many utilities of **modulus**:

Exercise 2.15 *Given an integer input, how do we determine, computationally, if an integer is odd or even?*

Input	1	2	3	4	1729
Odd / Even?					
`input / 2`					
`input % 2`					

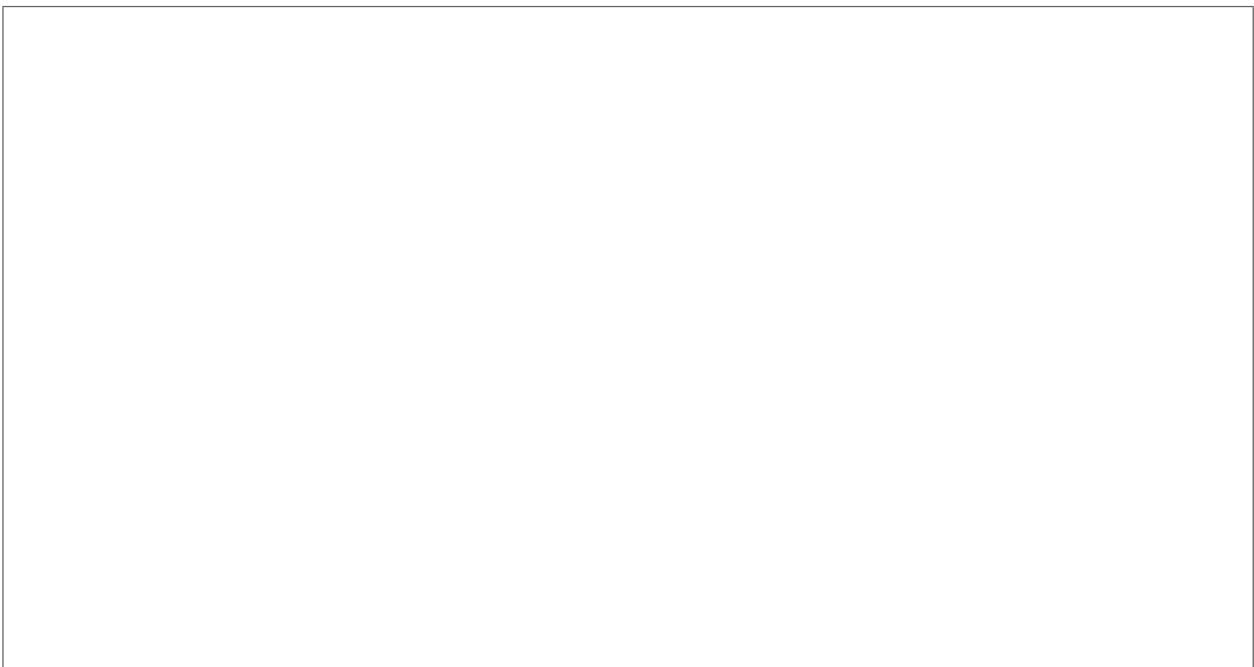
Modulus is the perfect computation operation to compute whether an integer is odd or even. As to how to properly detect an odd or even integer, we will do so when we explore **conditional branching**. We can extend this use beyond just checking for odd or even integers:

Exercise 2.16 *Given an integer input, how do we determine, computationally, if an integer is divisible by 3? By 7?*

In addition to calculating divisibility, modulus is integral to the fields of *cryptography*, *data structures*, and other important field in computation.

Exercise 2.17 *Given two input floats in Coral, write an algorithm in Coral that will find the average of the two inputs.*

Exercise 2.18 *Given two input integers in Coral, write an algorithm in Coral that will first output the quotient of the two integers (first divided by second), then output the remainder (modulo) of this division.*



2.7 Conditional Statements

Currently all of our algorithms run linearly: line-by-line-by-line. However we often might want to do different computational steps depending on the scenario. To introduce decision-making into our algorithms, we will need to use **conditional statements** to **branch** our algorithm structure.

2.7.1 Conditionals Basics

Consider the following array `A`:

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

Let's say we want to find the sum of only 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.

By its mathematical definition, we know an integer is positive if and only if it is greater than 0. A computer uses a **conditional statement** to be able to make branching decisions.

Conditional Statement
A type of instruction where the algorithm will take **branching** predetermined actions depending on a **Boolean value** (true or false). There are three basic variations of **branching** instructions:

In Coral, a **conditional statement** is written with **if... else...** statements:

```
if (<BOOLEAN CONDITION>)
    // Instructions if condition is true
else
    // Instructions if condition is false
```

If we only need to check a conditional and have no need for the an `else...` statement, we can have a stand-alone `if...` statement:

```
if (<BOOLEAN CONDITION>)
    // Instructions if condition is true
```

If there are cascading checks to be made for a computation, we can extend an `if... else...` statement with `if... elseif... else`:

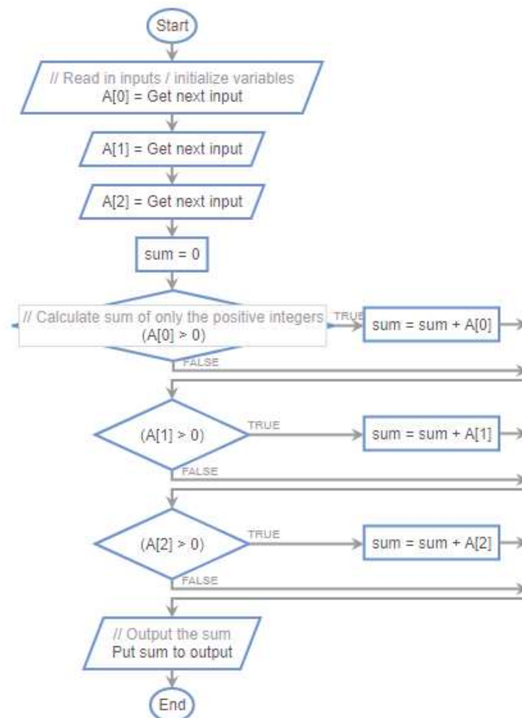
```
if (<BOOLEAN CONDITION>)
    // Instructions if the above condition is true
elseif (<BOOLEAN CONDITION>)
    // Instructions if the above condition is true and the preceeding
        ones fail
elseif (<BOOLEAN CONDITION>)
    // Instructions if the above condition is true and the preceeding
        ones fail
...
else
    // Instructions if all the above conditions fail
```

Every line that is indented after an `if` or `else` statement is **scoped** in that statement.

Let's try our hand at some conditional statements. Remember, for each one we want to identify what is the **true / false** condition that allows us to make the right decision.

Example 2.7 *Given three input integers, write an algorithm in Coral to find the sum of only the positive integers.*

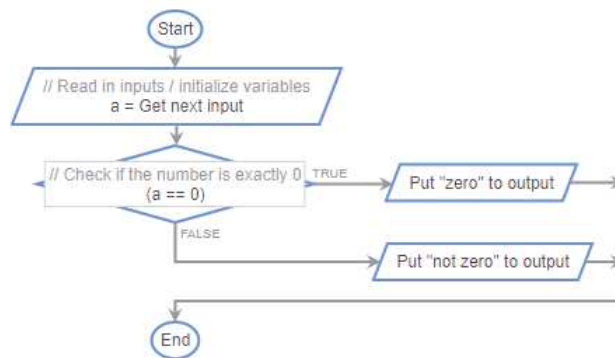
```
// Declare variables
integer array(3) A
integer sum

// Read in inputs / initialize variables
A[0] = Get next input
A[1] = Get next input
A[2] = Get next input
sum = 0

// Calculate sum of only the positive integers
if (A[0] > 0)
    sum = sum + A[0]
if (A[1] > 0)
    sum = sum + A[1]
if (A[2] > 0)
    sum = sum + A[2]

// Output the sum
Put sum to output
```



Exercise 2.19 *Given three input integers, write an algorithm in Coral that counts how many of the three integers are negative.*

Example 2.8 *Given an input integer, write an algorithm in Coral that outputs "zero" if the integer is a zero and "not zero" otherwise.*

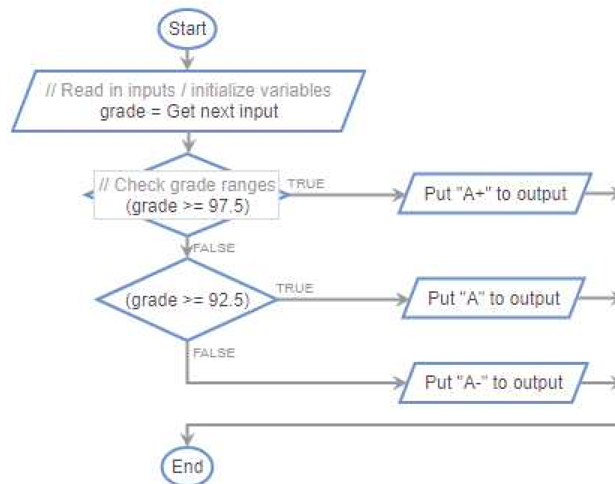
```
// Declare variables
integer a

// Read in inputs / initialize variables
a = Get next input

// Check if the number is exactly 0
if (a == 0)
    Put "zero" to output
else
    Put "not zero" to output
```



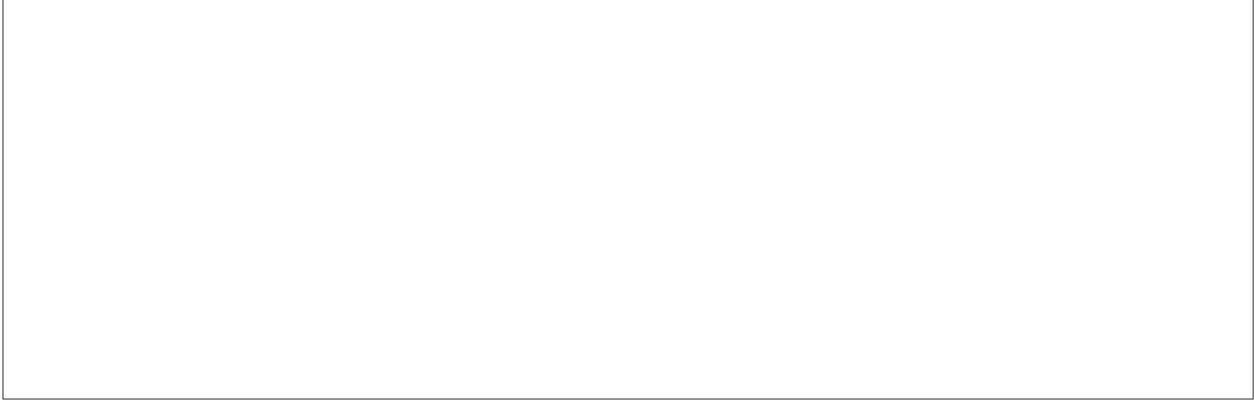
Exercise 2.20 *Given an input integer, write an algorithm in Coral that outputs "even" if the integer is a even and "odd" otherwise (what arithmetic operator can we use to check if an integer is even / divisible by 2?).*

Example 2.9 *Given an input float, write an algorithm in Coral that outputs "A+" if the number is 97.5 or above, "A" if the number is between 92.5 (inclusive) and 97.5 (exclusive), and "A-" if the number is between 90 (inclusive) and 92.5 exclusive. If the number is less than 90 exclusive or greater than 100 (exclusive), output "Invalid number".*

```
// Declare variables
float grade

// Read in inputs / initialize variables
grade = Get next input

// Check grade ranges
if (grade > 100)
    Put "Invalid number" to output
elseif (grade < 90)
    Put "Invalid number" to output
elseif (grade >= 97.5)
    Put "A+" to output
elseif (grade >= 92.5)
    Put "A" to output
else
    Put "A-" to output
```



Exercise 2.21 *Given an input integer, write an algorithm in Coral that outputs "negative" if the integer negative, "zero" if the integer is 0, and "positive" if the integer is positive.*



2.7.2 Nested and Compound Conditionals

Branching may occasionally not be as straightforward as a singular conditional check, sometimes we need to combine multiple conditionals together in order to achieve the behavior we want. Consider the following problem:

Example 2.10 *Given an input integer, write an algorithm in Coral to check if the integer is between 25 and 75 inclusive. Output "true" if it is so, and "false" otherwise.*

Written mathematically, our conditional check is:

$$25 \leq x \leq 75$$

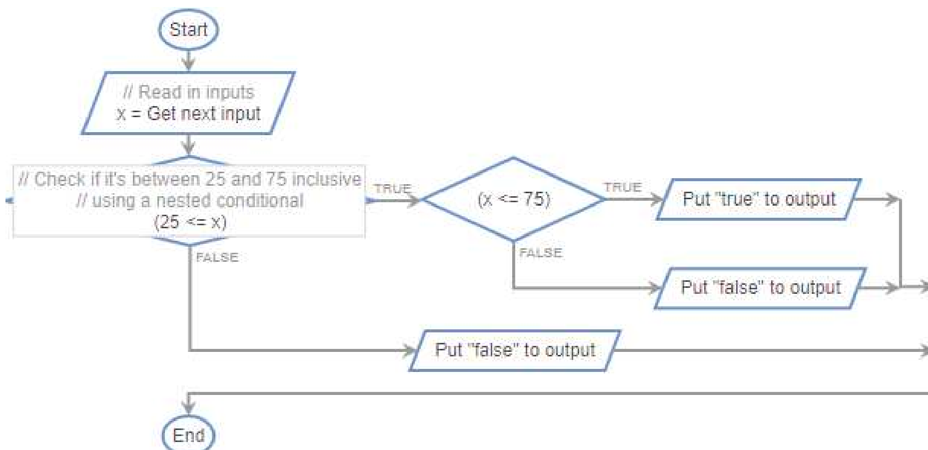
How do we represent this in Coral? Instead of trying to do both conditional checks at once, we can split it up into two parts:

$$25 \leq x \quad \texttt{AND} \quad x \leq 75$$

We can achieve this in two ways: using a **nested conditional** or a **compound conditional**.

> **Nested Conditional**
> A nested conditional constructed by writing a conditional statement within the **scope** of another conditional statement.

```
// Declare variables
integer x

// Read in inputs
x = Get next input

// Check if it's between 25 and 75 inclusive using a nested conditional
if (25 <= x)
    if (x <= 75)
        Put "true" to output
    else
        Put "false" to output
else
    Put "false" to output
```

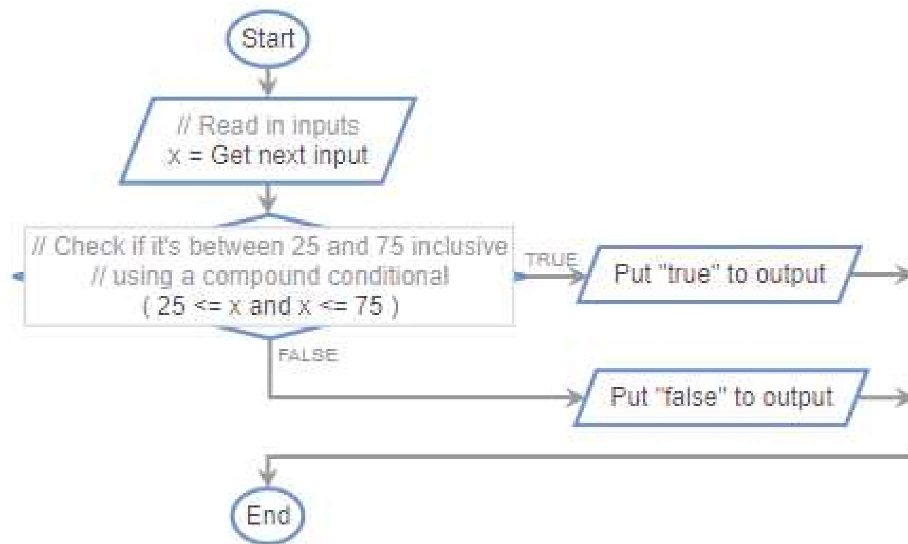


Notice that the inner conditional `if(x <= 75)` will only be executed if and only if the outer conditional `if(25 <= x)` is first satisfied. This emulates a **logical conjunction** between these two conditionals, achieving the behavior we want.

While this behavior is correct, the structure is rather complex and not ideal for a problem as simple as this one. Instead, we can use a **compound conditional** to achieve the same result.

> **Compound Conditional**
> A **conditional statement** constructed through two or more **conditional statement** logically joined together using **conjunction** or **disjunction**.
>
> In Coral, this is achieved using the `and` and `or` keywords in the conditional statement.

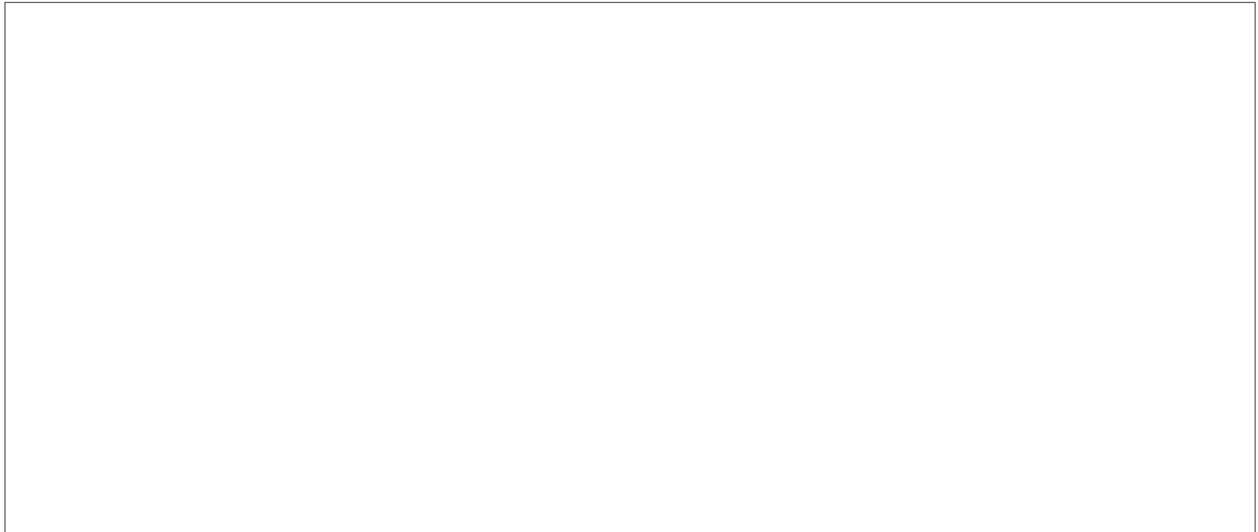
```
// Declare variables
integer x

// Read in inputs
x = Get next input

// Check if it's between 25 and 75 inclusive using a compound conditional
if ( 25 <= x and x <= 75 )
    Put "true" to output
else
    Put "false" to output
```



The use cases of each varies depending on the problem you are working on. For large complex multi-branching structures, a **nested conditional** can help keep your work organized by separating out each branching condition to its own `if...` statement. For more mathematically oriented problems (like the one above), a **compound conditional** can make your algorithm elegant and clean. Knowing both will be to your advantage to use the best-suited structure.

Exercise 2.22 *Modify the above code to output "Under range" if $x < 25$, "In range" if $x \geq 25$ and $x \leq 75$, and "Above range" if $x > 75$.*

Exercise 2.23 *Given an input integer, write an algorithm in Coral to check if it is either strictly less 25 or strictly greater than 75. Output "true" if the integer is within this range, and "false" otherwise. Why won't nested conditionals work here?*



Exercise 2.24 *Given two input integers, write an algorithm in Coral that will behave in the following way: if both integers are even, output the product of the two integers; if only one of the two integers are even, output the sum of the integers; if neither are even, output 0.*

2.8 Loops

By now you've noticed that you've written a lot of redundant lines in our algorithms. Consider the following problem we want to solve:

Example 2.11 *Given 10 input integers, write an algorithm in Coral to find the sum of only the odd integers.*

```
// Declare variables
integer array(10) A
integer sum

// Read in inputs and initialize values
A[0] = Get next input
A[1] = Get next input
A[2] = Get next input
A[3] = Get next input
A[4] = Get next input
A[5] = Get next input
A[6] = Get next input
A[7] = Get next input
A[8] = Get next input
A[9] = Get next input

sum = 0

// Compute sum of only odd integers
if (A[0] % 2 == 1)
    sum = sum + A[0]

if (A[1] % 2 == 1)
    sum = sum + A[1]

if (A[2] % 2 == 1)
    sum = sum + A[2]

if (A[3] % 2 == 1)
    sum = sum + A[3]

if (A[4] % 2 == 1)
    sum = sum + A[4]

if (A[5] % 2 == 1)
    sum = sum + A[5]

if (A[6] % 2 == 1)
    sum = sum + A[6]

if (A[7] % 2 == 1)
    sum = sum + A[7]

if (A[8] % 2 == 1)
    sum = sum + A[8]

if (A[9] % 2 == 1)
    sum = sum + A[9]

// Output sum
Put sum to output
```

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 as long as the **loop condition** is **true**.

In Coral, a loop is written with a **while** statement:

```
while (<BOOLEAN CONDITION>)
    // instructions if condition is true
```

*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.

2.8.1 Read Inputs with Loops

Let's simplify the ten lines of `Get next input`. Notice that in those ten lines, everything is written exactly the same except the index numbers used to access the array. For now, let's write down this line once instead of ten times:

```
A[???] = Get next input
```

Because we're going to repeat this line ten times, we need to put this inside of a **while loop**:

```
while (<BOOLEAN CONDITION>)
    A[???] = Get next input
```

Next, for the repeated lines, notice that the only changing part is the index integer. Rather than typing out `0, 1, 2, ..., 9`, let's instead use a **variable**. By convention, we will use the variable name `i` (for iterator):

```
while (<BOOLEAN CONDITION>)
    A[ i ] = Get next input
```

Looking at our repeated lines, we can see that our iterator `i` starts at the value `0`. Let's initiate our iterator variable to `0` in the beginning:

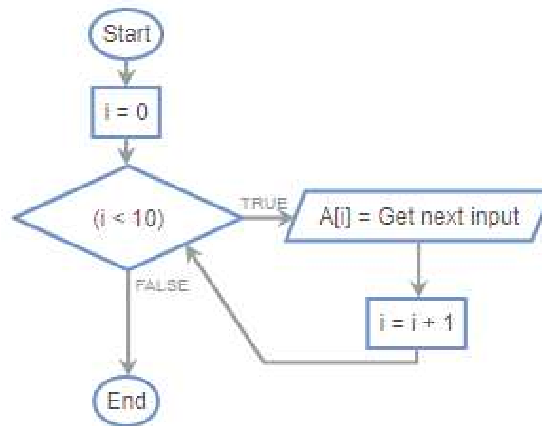
```
i = 0
while (<BOOLEAN CONDITION>)
    A[i] = Get next input
```

What is our **loop condition**? Under what **true condition** do we want to repeat the instruction? We know that `i` starts at `0` since we're starting at the 0th index. `i` will then need to increase by 1 each time to get `1`, then `2`, then `3`... until we get to `9`. This means that we want our instruction `A[i] = Get next input` to repeat as long as `i` is strictly less than `10`, or written computationally as `i < 10`:

```
i = 0
while (i < 10)
    A[i] = Get next input
```

Finally we need to increment the **iterator**. Right now, each time the instruction under the **while loop** executes, the value of `i` starts at `0` and never changes. We need to tell the computer to add 1 to the **iterator** to progress through the loop:

```
i = 0
while (i < 10)
    A[i] = Get next input
    i = i + 1
```



When writing loops, you want to look out for the following pieces of information:

- Which computational steps are being repeated?
- Of these repeated computations, which portions remain the same? Which portions are changing?
- For the changing portions, what value does it start at? What value does it end at? How does the value change each time?

For many (not all) computation problems, you'll want to make sure you have the following three pieces for a loop: 1) an iterator `i` initialized to the correct starting value, 2) a loop with the loop condition comparing the iterator to the ending value of the iterator, and 3) a line instructing the algorithm to increment the iterator by the amount you want.

2.8.2 Computing with Loops

Let's now work on the second repeated portion of our algorithm to sum up the odd integers of 10 input integers.

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

if (A[1] % 2 == 1)
    sum = sum + A[1]

...

if (A[9] % 2 == 1)
    sum = sum + A[9]
```

Exercise 2.25 *What are the portions that are being repeated?*

Exercise 2.26 *What are the portions that are changing?*

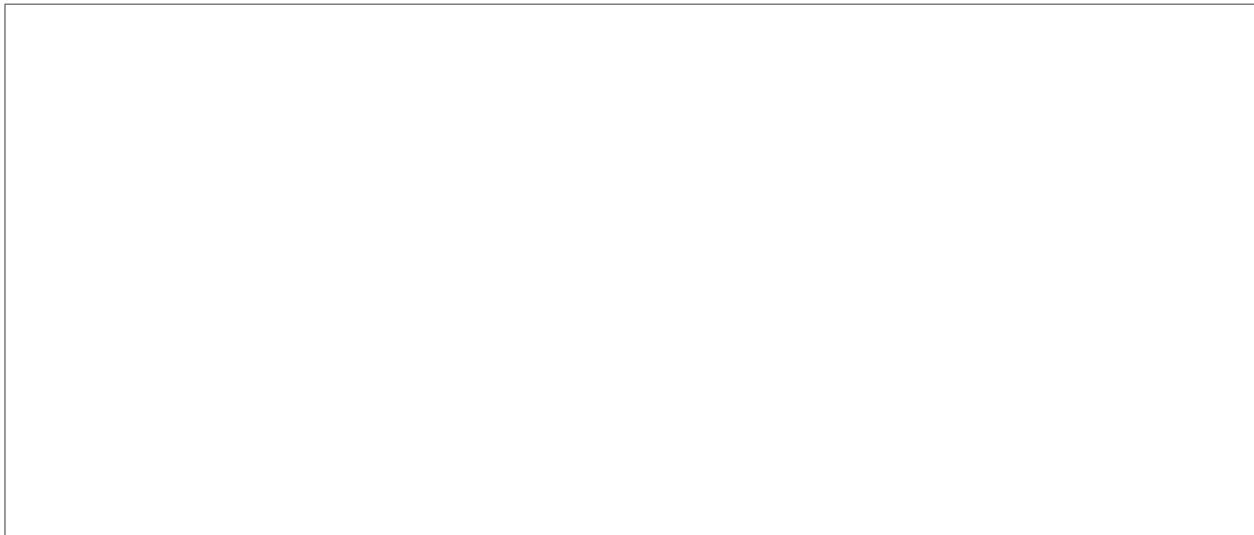
Exercise 2.27 *What value should our iterator start at?*

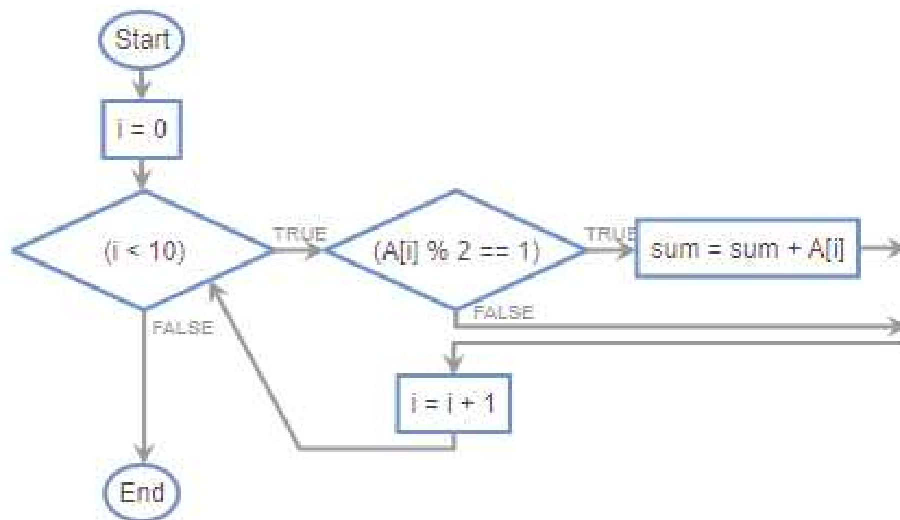
Exercise 2.28 *What value should our iterator end at?*

Exercise 2.29 *How much is our iterator increasing each time by?*

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

if (A[1] % 2 == 1)
    sum = sum + A[1]

...

if (A[9] % 2 == 1)
    sum = sum + A[9]
```

Exercise 2.30 *Write an algorithm in Coral for the above repeated instructions.*

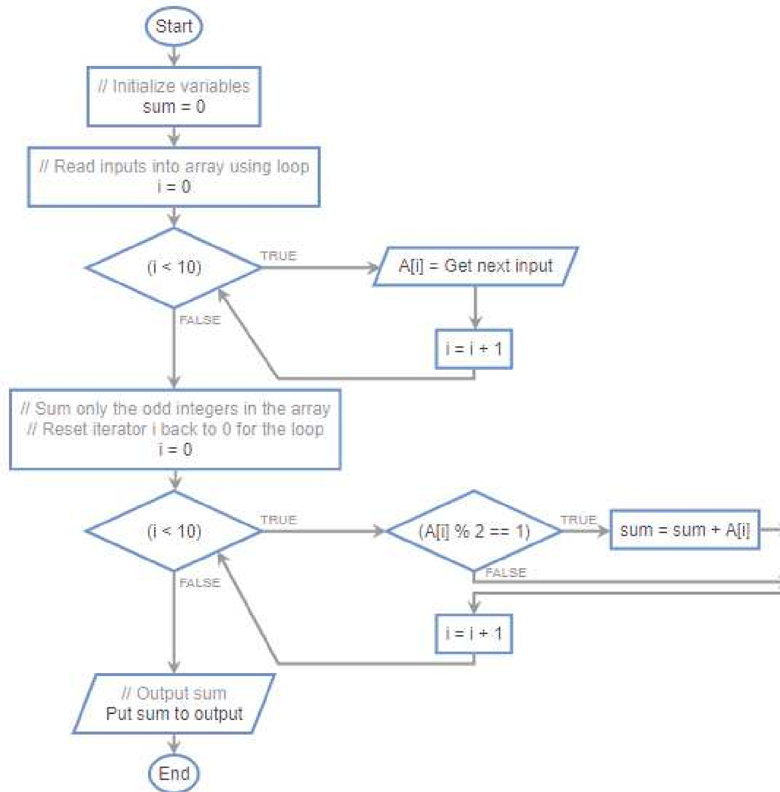




Putting it all together, we can now greatly simplify our original algorithm.

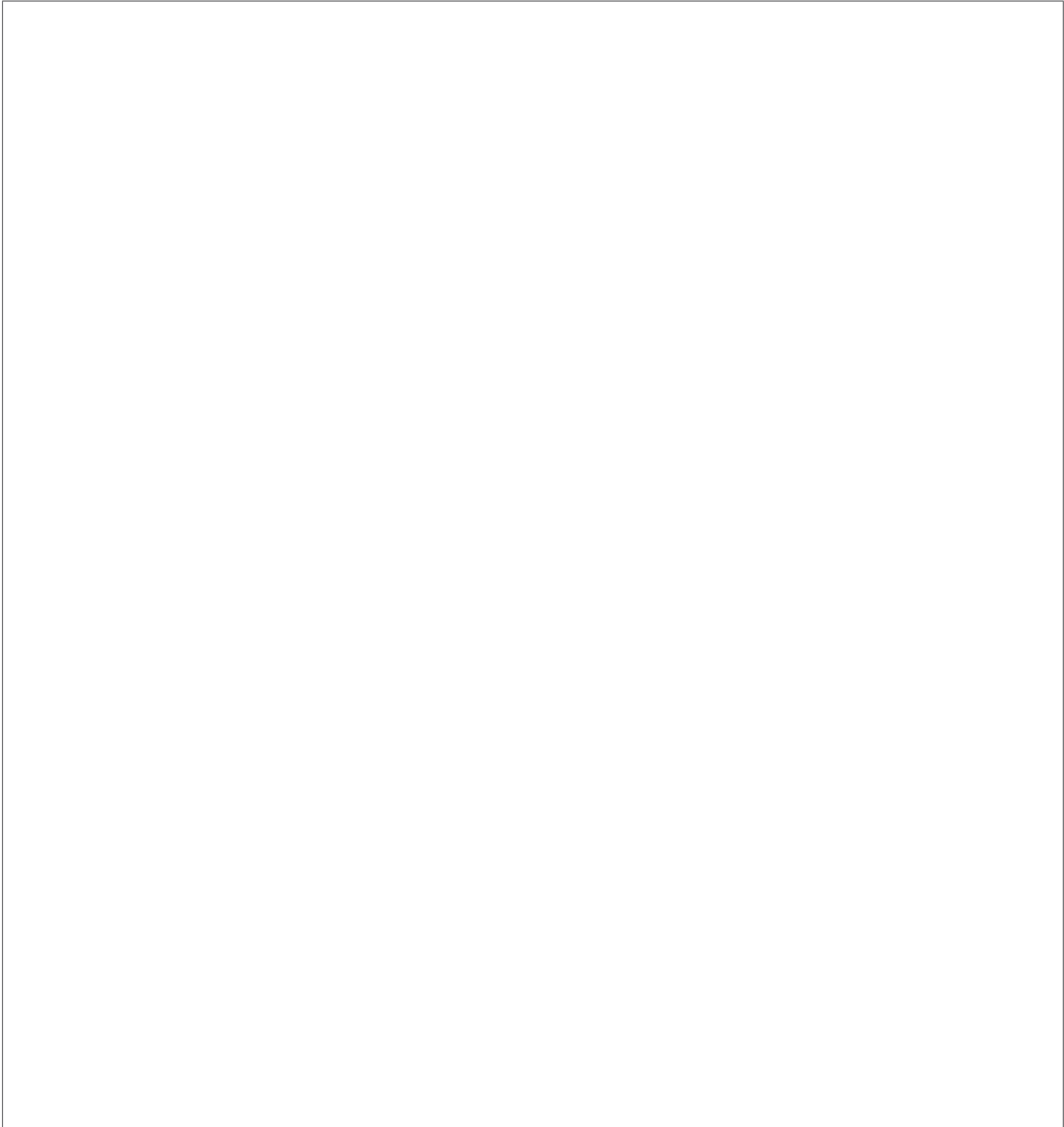
Example 2.12 *Given 10 input integers, write an algorithm in Coral to find the sum of only the odd integers.*

```
// Declare variables
integer array(10) A
integer sum
integer i

// Initialize variables
sum = 0

// Read inputs into array using loop
i = 0
while (i < 10)
    A[i] = Get next input
    i = i + 1

// Sum only the odd integers in the array
// Reset iterator i back to 0 for the loop
i = 0
while (i < 10)
    if (A[i] % 2 == 1)
        sum = sum + A[i]
    i = i + 1

// Output sum
Put sum to output
```



We are now fearless to the number of inputs! Whether it's 10 inputs, or 15, or 100, we can write our algorithm with elegance.

Exercise 2.31 *Given 1000 input integers, write an algorithm in Coral to sum up only the negative integers.*



2.8.3 Dynamic Arrays with Loops

Sometimes we know exactly how much space we need for an array when it is explicitly given to us (e.g. Given 10 input integers). When we know this exact amount, the array we allocate has a fixed size, called a **static array**.

Static Array
An array of a fixed size. This size cannot be changed.

In Coral, a static array created by specifying a specific integer during the array's declaration:

```
<DATA TYPE> array(n) <VARIABLE NAME>
```

Sometimes we do not know how much space we need to allocate to an array until later. Consider the following problem:

Example 2.13 *Given an input integer* `n`*, write an algorithm in Coral to read in an additional* `n` *integers and calculate the sum.*

First, we need an integer variable to store the input `n` that we are going to read in.

```
// Declare variables
integer n
```

Next, we know we are calculating a sum, so we need an integer variable for that too.

```
// Declare variables
integer n
integer sum
```

Finally, we know that we will need an array to store all of our integers.

```
// Declare variables
integer n
integer sum
integer array(...?) A
```

But how many integers do we have? We know we are going to have `n` integers, but how large is `n`? 10? 20? 100? We don't know until we read that integer in with `Get next input`. So how large should we make our array? 10? 20? 10000 to be safe? But `n` can be 100000. Why not allocate a space for 1 million? Is that enough space? Is that too much space?

We want to allocate the exact amount of space we need, no more no less. If `n = 15`, then we will allocate an array of size 15. If `n = 1729`, then we will allocate an array of size 1729. Once again, we don't know the exact size of `n` until we run our algorithm, therefore we need to **dynamically allocate** memory.

Dynamic Arrays
An array of dynamic size. The size of the array can be resized during the *runtime* of the algorithm.

In Coral, a dynamic array created by using a question mark `?` during the array's declaration, then later assigning the array's size:

```
<DATA TYPE> array(?) <VARIABLE NAME>
...
<VARIABLE NAME>.size = <ARRAY SIZE>
```

Let's see how this is used. Once again, let's declare our two integer variables:

```
// Declare variables
integer n
integer sum
```

Now, let's declare our array. Remember, we do not know the size of the array at this current moment in time, so we're going to tell the algorithm that we do not know by putting a `?` for the array's size. But, we're going to promise to tell the algorithm how large the array is when we find out what the value of `n` is later:

```
// Declare variables
integer n
integer sum
integer array(?) A
```

Notice that in Coral, our array's size is set to `?` with no memory slots at all:

```
1 // Declare variables
2 integer n
3 integer sum
4 integer array(?) A
5
6
```

Variables
0 n integer
0 sum integer
? .size A
not set integer array

Now that we have our variables declared, let's read in our inputs. Recall in the problem statement, the first input is the size `n`. Let's read that and assign it to our variable accordingly:

```
// Declare variables
integer n
integer sum
integer array(?) A

// Read inputs
n = Get next input
```

At this current moment in time, we now know the value of `n`, which means we know how large the array `A` should be! We need to fulfill our promise and update the algorithm the size of the array. We achieve this by assigning `A.size` (read as *"the size of A"*) the value of `n`:

```
// Declare variables
integer n
integer sum
integer array(?) A

// Read inputs
n = Get next input
A.size = n
```

Consider the sample input `3 9 10 1729`. This means the first input integer `3` is the value of `n`, indicating to us that there are 3 more integers of `9 10 1729` to read into our array. Our algorithm will resize our array to have a size of exactly 3:

```
1 // Declare variables
2 integer n
3 integer sum
4 integer array(?) A
5
6 // Read inputs
7 n = Get next input
8 A.size = n
```

Variables

Value	Name	Type
3	n	integer
0	sum	integer
3	.size	A integer array
0	[0]	
0	[1]	
0	[2]	

Input

3 9 10 1729

Let's try a different size of inputs! What if our input is instead `5 9 10 1 12 1729`. This means `n = 5` and we'll have 5 more integers `9 10 1 12 1729` to read into our array, so we need to **dynamically allocate** five slots of memory:

```
1 // Declare variables
2 integer n
3 integer sum
4 integer array(?) A
5
6 // Read inputs
7 n = Get next input
8 A.size = n
```

Variables

Value	Name	Type
5	n	integer
0	sum	integer
5	.size	A integer array
0	[0]	
0	[1]	
0	[2]	
0	[3]	
0	[4]	

Input

5 9 10 1 121729

The beauty of this is that we now have one algorithm that works for any size of inputs. Whether it's needing to sum 10 integer, or 100, or 10000, our algorithm will be able to dynamically adjust the size of our array to be a perfect fit.

Exercise 2.32 *Complete the following algorithm – Given an input integer* `n`*, write an algorithm in Coral to read in an additional* `n` *integers and calculate the sum. You may need to declare more variables, feel free to make any additions you see fit.*

```
// Declare variables
integer n
integer sum
integer array(?) A




// Read inputs
n = Get next input
A.size = n













// Compute the sum

















// Output the sum
Put sum to output
```

Exercise 2.33 *Given an input integer `n` followed by another integer `k`, write an algorithm in Coral to read in an additional `n` integers into an array. Then check to see if the integer `k` can be found in the array. If so, output `true` to the output, otherwise output `false`.*

If you try running your algorithm in Coral, you might notice that there are some odd kinks in the behavior of the algorithm. We'll sort out very soon.

2.8.4 Looping with Flags

The previous algorithm is a common task for computers to do: looking to see whether a target `k` exists within an array. Let's call this the **searching algorithm**.

Let's trace the algorithm from the previous exercise with some examples. Remember, we have an input `n`, the size of our array, an input `k`, the target integer we want look for in our array, and `n` additional integers, the contents of our array.

Example 2.14 *Trace the **searching algorithm** with the following input. How many steps did it take?*

Input: [5 7 2 9 7 2 5]

The first input integer is the size of our array `n = 5`, and the second input integer is our target `k = 7`. Let's setup our tracing:

n	5	5	5	5	5
k	7	7	7	7	7
i	0	1	2	3	4
A[i]	2	9	7	2	5
k == A[i]	F	F	T	F	F

In total, it took 5 steps for the algorithm to complete.

Exercise 2.34 *Trace the **searching algorithm** with the following input. How many steps did it take?*

Input: [7 9 5 17 6 3 26 1 9]

`n =` `k =`

`A =`

Exercise 2.35 *Trace the **searching algorithm** with the following input. How many steps did it take?:*

Input: [20 5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 5]

n =   k =

A =

Exercise 2.36 *Trace the **searching algorithm** with the following input. How many steps did it take?:*

Input: [100 5 5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

n =   k =

A =

Notice that for our current algorithm, there's a lot of redundant steps. As a human, we automatically optimize our work. However, for computers, they will mindlessly do the work we tell them to.

Our search algorithm has two behavioral scenarios:

- If we haven't found our target `k` in our array yet, we have to keep looking because it might appear later in the array.
- If we found or target `k` in our array, we don't need to keep looking any more and ignore the rest of the array.

Right now our **searching algorithm** will continue to check the array even if we found our target `k`. How do we tell the computer to stop looking once we've found our target `k`? To do so, we need some kind of indicator, a **flag**, that will change the behavior of our algorithm.

> **Flags**
> A variable, usually of a **boolean type**, used dictate the behavior of an algorithm.

Let's see how this is used to improve the **efficiency** of our **searching algorithm**. First is the base algorithm we've already designed:

```
// Declare variables
integer n
integer k
integer array(?) A
integer i

// Read inputs, allocate memory for array
n = Get next input
A.size = n
k = Get next input

// Read n inputs into array
i = 0
while ( i < n )
    A[i] = Get next input
    i = i + 1

// Search for target k in array
i = 0
while ( i < n )
    if ( k == A[i] )
        Put "true" to output
    else
        Put "false" to output
    i = i + 1
```

Let's add in our **flag variable**. We do need a bit of prep work to assign meaning to its value:

- If `flag = 0` (false), it means we have not yet found our target `k`
- If `flag = 1` (true), it menas we have found our target `k`

Therefore, by default, we should initialize the our **flag variable** to 0.

```
// Declare variables
integer n
integer k
integer array(?) A
integer i
integer flag

// Read inputs, allocate memory for array
n = Get next input
A.size = n
k = Get next input
flag = 0

// Read n inputs into array
i = 0
while ( i < n )
    A[i] = Get next input
    i = i + 1

// Search for target k in array
i = 0
while ( i < n )
    if ( k == A[i] )
        Put "true" to output
    else
        Put "false" to output
    i = i + 1
```

Next, where should we flip our **flag** to `1`? If we check back at how we are using the flag, `flag = 1` means we have found our target `k`. Finding our target `k` in our array occurs in our conditional `if ( k == A[i] )`, so we'll flip the value of our flag in the **scope** of that if statement.

```
// Declare variables
integer n
integer k
integer array(?) A
integer i
integer flag

// Read inputs, allocate memory for array
n = Get next input
A.size = n
k = Get next input
flag = 0

// Read n inputs into array
i = 0
while ( i < n )
    A[i] = Get next input
    i = i + 1

// Search for target k in array
i = 0
while ( i < n )
    if ( k == A[i] )
        flag = 1
        Put "true" to output
    else
        Put "false" to output
    i = i + 1
```

Now we need to figure out how to tell our algorithm to stop looking if we've already found our target `k`. The information of *"we've already found our target* `k`*"* is encoded by `flag = 1`, and we've made sure to add in our computation in our previous step.

How do we tell the algorithm to stop looking? The *"keep looking behavior"* is determined by our **while loop**, therefore we need to modify our **while statement**. If we dissect our **looping condition**, it has two parts:

1. Keep looking if there are more elements in our array to check `(i < n)`
2. Keep looking if we haven't found our target `k` yet `flag == 0`

Let's add the second condition to our **while statement**:

```
// Declare variables
integer n
integer k
integer array(?) A
integer i
integer flag

// Read inputs, allocate memory for array
n = Get next input
A.size = n
k = Get next input
flag = 0

// Read n inputs into array
i = 0
while ( i < n )
    A[i] = Get next input
    i = i + 1

// Search for target k in array
i = 0
while ( i < n and flag == 0 )
    if ( k == A[i] )
        flag = 1
        Put "true" to output
    else
        Put "false" to output
    i = i + 1
```

Let's see what this means. As long as we haven't found our target `k` yet, the flag will always remain at `flag = 0`, its default value, and we'll keep looping through our loop regularly with `(i < n)`.

However, the moment we find our target `k` in our array with `(k == A[i])`, the flag is flipped to `flag = 1` indicating that we have found our target `k`. The next time the algorithm checks the **loop condition**, because our flag is now `flag = 1`, the comparison `flag == 0` will resolve to `false`, causing the entire loop condition `( i < n and flag == 0 )` to fail as a result, ending our looping immediately, saving us time of not needing to check the rest of the array.

There is one final adjustment to make. Our outputting of `true` or `false` is now contingent upon our `flag`. So, let's move our output statements outside of the loop so we will output once and only once.

```
// Declare variables
integer n
integer k
integer array(?) A
integer i
integer flag

// Read inputs, allocate memory for array
n = Get next input
A.size = n
k = Get next input
flag = 0

// Read n inputs into array
i = 0
while ( i < n )
    A[i] = Get next input
    i = i + 1

// Search for target k in array
i = 0
while ( i < n  and flag == 0 )
    if ( k == A[i] )
        flag = 1
        Put "true" to output
    else
        Put "false" to output
    i = i + 1

if ( flag == 0 )
    Put "false" to output
else
    Put "true" to output
```

Keep in mind that it's difficult to notice that we will need a **flag variable** until the moment before we need it. Don't start an algorithm wondering if you need a flag variable – its use and necessity will make itself apparent naturally. Remember, this is a tool for you to use in your algorithm, not something to memorize its use.

Exercise 2.37 ***Challenge:*** *Can you implement the* ***searching algorithm*** *without a flag variable and still have the correct behavior? Where can you "encode" a flag in the algorithm without a dedicated flag variable?*

Exercise 2.38 *Given an input integer* `n`*, read in an additional* `n` *input integers into an array. Write an algorithm in Coral to detect if there is an even integer in the array. If so, output* `true`*, otherwise output* `false`*.*

2.8.5 Nested Loops

Let's add another layer of complexity to our searching problem. Instead of searching for one target `k`, what if we want to search for multiple targets?

Example 2.15 *Given and integer input* `n` *followed by five target integers* k_1 *to* k_5*, then followed by* `n` *integers for an array. Write an algorithm in Coral to check if each of the five targets can be found in the array. For each target,* `true` *if it can be found and* `false` *otherwise.*

Before we start writing the algorithm, let's see if we can work through a small example to orient ourselves. Consider the following input for the problem:

```
Input:  [ 5 2 4 6 8 10 3 4 5 6 7 ]
```

Let's read through this example and **parse** the input bit by bit.

Exercise 2.39 *Parse the above example input and populate the variables below:*

n =

---

k_1 = k_2 = k_3 = k_4 = k_5 =

A =

Now that we've parsed the input, let's manually work through the problem:

Exercise 2.40 *Answer the following using the parsed information in the previous exercise*

(a) Can we find our first target k_1 in our array `A`?

(b) Can we find our second target k_2 in our array `A`?

(c) Can we find our third target k_3 in our array `A`?

(d) Can we find our fourth target k_4 in our array `A`?

(e) Can we find our fifth target k_5 in our array `A`?

We want to also make sure we know what the output of our algorithm should be. Recall that we should output `true` if the target is found and `false` otherwise.

Exercise 2.41 *What is the output for the above example input?*

As we were doing this problem manually, notice that we had repeated actions. We were doing the exact same algorithmic steps for each of our five targets $k_1 - k_5$. More specifically, we were running our **searching algorithm** for each of the five targets $k_1 - k_5$.

The behavior of the entire algorithm is a **nested loop**: we were looping five times, once for each of our five targets $k_1 - k_5$, and for each target k_i we had to run our **searching algorithm**, which in itself has a loop.

> **Nested Loop**
> An algorithmic structure wherein a loop (**inner loop**) is written within the scope of another loop (**outer loop**). The **inner loop** will run its entirety, followed by one incremental step of the **outer loop**.

Before worrying about implementing a **nested loop**, let's see if we can write our algorithm with some redundancy first, then eliminate the redundancy using a **nested loop**.

First, we need to properly read in our inputs. Let's declare the necessary variables. One integer variable for `n`. Instead of five separate variables for $k_1 - k_5$, let's use a **static array** `K` of size 5 for efficiency. Lastly, we need a **dynamic array** `A` which will by sized to `n` later. For our two arrays `K` and `A` we'll need separate iterators `i` and `j` as well. Finally, we need a **flag variable** to optimize our **searching algorithm**.

```
// Declare variables
integer n
integer array(5) K
integer array(?) A
integer i
integer j
integer flag
```

Now let's read our input, parse it, and populate our variables accordingly.

```
// Read first input n, allocate array, set flag to 0
n = Get next input
A.size = n
flag = 0

// Read the five targets into array K
i = 0
while ( i < 5 )
    K[i] = Get next input
    i = i + 1

// Read the next n integers into array A
j = 0
while ( j < n )
    A[j] = Get next input
    j = j + 1
```

Now that we've read in our inputs and stored them properly, let's use our existing **searching algorithm** on our first target `K[0]`. Notice that we now have to juggle two iterators: `i` is reserved for iterating through our target array `K`, and `j` is reserved for iterating through our array `A`.

```
// Reset iterators back to 0
i = 0
j = 0

// Search for K[0] in A
while ( j < n and flag == 0 )
    if ( K[i] == A[j] )
        flag = 1
    j = j + 1

if ( flag == 0 )
    Put "false" to output
else
    Put "true" to output
```

That's target number one k_1 (stored in `K[0]`) finished. Now let's repeat this process for targets $k_2 - k_5$:

```
// Reset flag and j, increment i to prepare for target K[1]
flag = 0
j = 0
i = i + 1

while ( j < n and flag == 0 )
    if ( K[i] == A[j] )
        flag = 1
    j = j + 1

if ( flag == 0 )
    Put "false" to output
else
    Put "true" to output

// Reset flag and j, increment i to prepare for target K[2]
flag = 0
j = 0
i = i + 1

while ( j < n and flag == 0 )
    if ( K[i] == A[j] )
        flag = 1
    j = j + 1

if ( flag == 0 )
    Put "false" to output
else
    Put "true" to output

// Reset flag and j, increment i to prepare for target K[3]
flag = 0
j = 0
i = i + 1

while ( j < n and flag == 0 )
    if ( K[i] == A[j] )
        flag = 1
    j = j + 1

if ( flag == 0 )
    Put "false" to output
else
    Put "true" to output

// Reset flag and j, increment i to prepare for target K[4]
flag = 0
j = 0
i = i + 1

while ( j < n and flag == 0 )
    if ( K[i] == A[j] )
        flag = 1
    j = j + 1

if ( flag == 0 )
    Put "false" to output
else
    Put "true" to output
```

That was literally copy and pasting our **searching algorithm** five times. We now encounter the same situation as before: how many targets do we have? 10? 20? 100? Déjà Vu much?

Let's resolve this Déjà Vu with *algorithmic Déjà Vu* – **loops**! Instead of copy and pasting our **sorting algorithm**, let's wrap it in a loop around it:

```
// Declare variables
integer n
integer array(5) K
integer array(?) A
integer i
integer j
integer flag

// Read first input n, allocate array, set flag to 0
n = Get next input
A.size = n
flag = 0

// Read the five targets into array K
i = 0
while ( i < 5 )
    K[i] = Get next input
    i = i + 1

// Read the next n integers into array A
j = 0
while ( j < n )
    A[j] = Get next input
    j = j + 1

// Reset our iterators
i = 0
j = 0

// Outer loop: iterates through the targets K[0] to K[4]
// iterator i is reserved for iterating through K
while ( i < 5 )

    // Inner loop: our sorting algorithm logic
    while ( j < n and flag == 0 )
        if ( K[i] == A[j] )
            flag = 1
        j = j + 1

    if ( flag == 0 )
        Put "false" to output
    else
        Put "true" to output
    // End of inner loop
```

There's a few things we need to iron out – we cannot just plug and play and expect everything to work. First, our **outer loop** `while ( i < 5 )` uses iterator `i`, however we have not incremented `i` yet! Let's add that at the tail end of our **outer loop** after our **searching algorithm** logic:

```
// Declare variables
integer n
integer array(5) K
integer array(?) A
integer i
integer j
integer flag

// Read first input n, allocate array, set flag to 0
n = Get next input
A.size = n
flag = 0

// Read the five targets into array K
i = 0
while ( i < 5 )
    K[i] = Get next input
    i = i + 1

// Read the next n integers into array A
j = 0
while ( j < n )
    A[j] = Get next input
    j = j + 1

// Reset our iterators
i = 0
j = 0

// Outer loop: iterates through the targets K[0] to K[4]
// iterator i is reserved for iterating through K
while ( i < 5 )

    // Inner loop: our sorting algorithm logic
    while ( j < n and flag == 0 )
        if ( K[i] == A[j] )
            flag = 1
        j = j + 1

    if ( flag == 0 )
        Put "false" to output
    else
        Put "true" to output
    // End of inner loop

    i = i + 1
```

Also, we need to make sure our **inner loop** iterator `j` is reset to `0` before our **sorting algorithm** begins. The `flag` needs to be reset to `0` each time as well. Let's add these variable resets at the beginning of our **outer loop**, before our **searching algorithm** logic starts:

```
// Declare variables
integer n
integer array(5) K
integer array(?) A
integer i
integer j
integer flag

// Read first input n, allocate array, set flag to 0
n = Get next input
A.size = n
flag = 0

// Read the five targets into array K
i = 0
while ( i < 5 )
    K[i] = Get next input
    i = i + 1

// Read the next n integers into array A
j = 0
while ( j < n )
    A[j] = Get next input
    j = j + 1

// Reset our iterators
i = 0
j = 0

// Outer loop: iterates through the targets K[0] to K[4]
// iterator i is reserved for iterating through K
while ( i < 5 )
    // Reset variables j and flag to prepare for inner loop
    j = 0
    flag = 0

    // Inner loop: our sorting algorithm logic
    while ( j < n and flag == 0 )
        if ( K[i] == A[j] )
            flag = 1
        j = j + 1

    if ( flag == 0 )
        Put "false" to output
    else
        Put "true" to output
    // End of inner loop

    i = i + 1
```

Exercise 2.42 *Let's now generalize this from five target inputs K_1 – K_5 to* `m` *target inputs $K_i, 1 \leq i \leq m$. Modify the following algorithm to search for* `m` *target inputs over an array of size* `n`*.*

```
// Declare variables
integer n
integer m
integer array(?) K
integer array(?) A
integer i
integer j
integer flag

// Read first input n, allocate array, set flag to 0
n = Get next input
m = Get next input
A.size = n
K.size = m
flag = 0

// Read the next m inputs into target array K
i = 0
while ( i < 5 )
    K[i] = Get next input
    i = i + 1

// Read the next n integers into array A
j = 0
while ( j < n )
    A[j] = Get next input
    j = j + 1

// Reset our iterators
i = 0
j = 0

// Outer loop: iterates through the targets K[0] to K[m-1]
// iterator i is reserved for iterating through K
while ( i < 5 )
    // Reset variables j and flag to prepare for inner loop
    j = 0
    flag = 0

    // Inner loop: our sorting algorithm logic
    while ( j < n and flag == 0 )
        if ( K[i] == A[j] )
            flag = 1
        j = j + 1

    if ( flag == 0 )
        Put "false" to output
    else
        Put "true" to output
    // End of inner loop

    i = i + 1
```

For as daunting as this algorithm looks, as long as we keep everything organized we will not get lost. **Commenting** each section and **encapsulating** the behaviors (keeping the input reading separate from the logic etc.) gives the algorithm structure.

For this course, you will **not** need to write a nested loop algorithm from scratch, you'll get plenty of practice for that in I210. However, you will need to be able to look at a completed **nested loop** algorithm and understand it's structure: which loop is the **outer loop**? Which is the **inner loop**? How are they behaving independently / interacting with one another?

Exercise 2.43 *For the following snippet of algorithm, answer the following questions:*

```
m = 5
n = 7
i = 0
while ( i < m )
    j = 0
    while ( j < n )
        sum = sum + j
        j = j + 1
    i = i + 1
```

(a) What is the outer loop? How many times does the outer loop iterate?

(b) What is the inner loop? How many times does the inner loop iterate?

(c) How many times will the line `sum = sum + j` execute?

Exercise 2.44 *For the following snippet of algorithm, answer the following questions:*

```
m = 5
i = 0
while ( i < m )
    j = 0
    while ( j < i )
        sum = sum + j
        j = j + 1
    i = i + 1
```

(a) What is the outer loop? How many times does the outer loop iterate?

(b) What is the inner loop? How many times does the inner loop iterate?

(c) How many times will the line `sum = sum + j` execute?

2.9 Types of Algorithms

Recall the following **searching problem** we looked at extensively in the previous section:

> *Given an input integer* `n` *followed by a target integer* `k`*, then by* `n` *additional input integers, write an algorithm in Coral to determine if the target* `k` *can be found in the* `n` *input integers.*

And to solve this, we designed a **searching algorithm** to solve this problem:

```
// Searching algorithm logic
while ( i < n and flag == 0 )
    if ( k == A[i] )
        flag = 1
    i = i + 1

if (flag == 0)
    Put "false" to output
else
    Put "true" to output
```

As it turns out, this type of **searching** problem is very common in computation. As such, computational scholars have spent a lot of time exploring if these problems can be solved more **efficiently** and have come up with a few interesting designs.

In this section, we will look at two common computational algorithm: the **searching algorithms** and **sorting algorithms**.

2.9.1 Search Algorithms

By this point we're quite familiar with a **search algorithm**. Rather than writing the quite convoluted problem definition for Coral, let's define a more generalized **search algorithm**:

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

The **search algorithm** that we designed previously is a legitimate solution to this problem called **linear search**, since we look through the array **linearly** one element at a time. Just as we defined a generalized **search algorithm** definition, let's also write a generalized algorithm, called a **pseudocode**, for **linear search**:

> **Linear Search**
> Given an array `A` of size `n` and target `k`:
>
> 1. Create an iterator `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`.

Here let's get familiar with the computational steps of **linear search**.

Example 2.16 *Trace linear search on the following array* `A` *with the target* `k = 33`:

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

`k`	33	33	33	33
`n`	9	9	9	9
`i`	0	1	2	3
`A[i]`	3	15	-7	33
`A[i] == k`	F	F	F	T

Exercise 2.45 *Trace linear search on the above array* `A` *with the target* `k = 23`:

Exercise 2.46 *How many steps did it take for linear search to find the target* `k = 23` *on the above array* `A`*?*

Exercise 2.47 *How many steps will it take for linear search to find the target* `k = 3` *on the above array* `A`*?*

Exercise 2.48 *How many steps did it take for linear search to find the target* `k = 99` *on the above array* `A`*?*

In the previous example, we notice a coincidental convenience and inconvenience with **linear search**. If the target `k` is at the beginning of the array, then we have the happy coincidence and **linear search** will find the `k` very quickly. However, if the target `k` is at the very end of the array, then we have the unfortunate circumstance of requiring a long time to find it. So naturally the next question we ask is: can we do better?

Exercise 2.49 *How many steps will it take linear search to find the target `k = 23` on the following array `S` of size `n = 17`?*

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

Exercise 2.50 *What do you notice about the array `S`?*

Exercise 2.51 *For this specific array, what's a better strategy than linearly looking at every single integer in sequential order?*

Here we experience one of the golden rules in computation:

The more information we know about a 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 given array is **sorted**.

For a **sorted array**, the more efficient algorithm to search for a target `k` 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`

The logic for **binary search** in Coral is more complex, which is an addendum to the golden rule we mentioned above:

The more information we know about a problem,
the more efficiently we can solve the problem,
but the logic of the algorithm will be more complex.

If you're so inclined, here is the core logic of **binary search** written in Coral. The variable declaration and input reading has been removed for brevity. Note that you do not need to know how implement **binary search**; you just need to understand the behavior of **binary search**.

```
// Searching algorithm logic
left = 0
right = n - 1
while (left <= right and flag == 0)
    mid = (left + right) / 2
    if (A[mid] == k)
        flag = 1
    else if (A[mid] > k)
        right = mid - 1
    else
        left = mid + 1
if (flag == 0)
    Put "false" to output
else
    Put "true" to output
```

Example 2.17 *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 ]`

A	n	n/2	A[n/2]	A[n/2] == 59	Left or Right?
[2,3,4,7,11,16,19,50,55,59,80,97]	12	6	19	F	Right
[50,55,59,80,97]	5	2	59	T	—

Exercise 2.52 *Trace binary search with the array* `A` *from* ***Example 1.8*** *with the target* `k = 3`*.*

Exercise 2.53 *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 2.54 *Trace binary search with the array* `b` *from* ***Exercise 1.28*** *and the target* `k = 200`*.*

2.9.2 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, with some more efficient than others:

Selection Sort   Insertion Sort   Merge Sort   Heapsort
Quicksort   Shellsort   Bubble Sort   Comb Sort
Counting Sort   Bucket Sort   Radix Sort

Here we will only look at one of the simpler algorithms: **selection 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 append it to `S`
   (b) Remove the smallest member in `A`
3. Return `S`

The following is the core logic of **selection sort** written in Coral. Like **binary search** you do not need to know how to write this algorithm from scratch, but you will need to understand its behavior.

```
// Selection sort algorithm logic
i = 0
while (i < n)
    smallest = INT_MAX
    j = 0
    while (j < n)
        if (A[j] < smallest)
            smallest = A[j]
            s_i = j
        j = j + 1
    S[i] = smallest
    A[s_i] = INT_MAX
    i = i + 1
```

Exercise 2.55 *For the above snippet of Coral, highlight the nested loop. Which portions are the inner loop, and which portions are the outer loop?*

Despite the complexity of **selection sort**, we can still understand the main portion of it. Let's design the inner loop used in **selection sort**.

Exercise 2.56 *Given a* n *input integers, write an algorithm in Coral to output the minimum integer.*

Example 2.18 *Trace selection sort with the following array `A`:*

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

`A`	Smallest	Sorted Array `S`
`[8, 5, 3, 1, 2, 1, 21, 13]`	1	`[ ]`
`[8, 5, 3, 2, 1, 21, 13]`	1	`[1]`
`[8, 5, 3, 2, 21, 13]`	2	`[1, 1]`
`[8, 5, 3, 21, 13]`	3	`[1, 1, 2]`
`[8, 5, 21, 13]`	5	`[1, 1, 2, 3]`
`[8, 21, 13]`	8	`[1, 1, 2, 3, 5]`
`[21, 13]`	13	`[1, 1, 2, 3, 5, 8]`
`[21]`	21	`[1, 1, 2, 3, 5, 8, 13]`
`[ ]`	—	`[1, 1, 2, 3, 5, 8, 13, 21]`

Exercise 2.57 *Trace selection sort with the following array `B`:*

`B = [ 3, 61, 17, 2 ]`

Exercise 2.58 *Trace selection sort with the following array `C`:*

`C = [ 17, 2, 82, 17, 33 ]`

Exercise 2.59 *Trace selection sort with the following array `D`:*

`D = [ 1, 2, 3, 4 ]`