A

Introduction to Pseudocode

What is Pseudocode?

An **algorithm** is a sequence of instructions to solve a well-formulated computational problem specified in terms of its **input** and **output**. An algorithm uses the input to generate the output. For example, the algorithm **PATTERNCOUNT** uses strings *Text* and *Pattern* as input to generate the number COUNT(*Text*, *Pattern*) as its output.

In order to solve a computational problem, you need to carry out the instructions specified by the algorithm. For example, if we want you to count how many times *Pattern* appears in *Text*, we could tell you to do the following:

1. Start from the first position of *Text* and check whether *Pattern* appears in *Text* starting at its first position.
2. If yes, draw a dot on a piece of paper.
3. Move to the second position of *Text* and check whether *Pattern* appears in *Text* starting at its second position.
4. If yes, draw another dot on the same piece of paper.
5. Continue until you reach the end of *Text*.
6. Count the number of dots on the paper.

Since humans are slow, make mistakes, and hate repetitive work, we invented computers, which are fast, love repetitive work, and never make mistakes. However, while you may easily understand the above instructions for counting the number of occurrences of *Pattern* in *Text*, no computer in the world can execute them. The only reason you can understand them is because you have been trained for many years to understand human language. For example, we didn't specify that you should start from a *blank* piece of paper without any dots, but you assumed it. We didn't explain what it means to "reach the end of *Text*"; at what position of *Text* should we stop?

Because computers do not understand human language, algorithms must be rephrased in a **programming language** (such as Python, Java, C++, Perl, Ruby, Go, or dozens of others) in order to give the computer specific instructions. However, we don't want to describe algorithms in a specific language because it may not be your favorite

Our focus is on algorithmic ideas rather than on implementation details, which is why we will meet you halfway between human languages and programming languages by using **pseudocode**. By emphasizing ideas rather than implementation details, pseudocode is able to describe algorithms without being too formal, ignoring many of the tedious details that are required in a specific programming language. At the same time, pseudocode is more precise and less ambiguous than the instructions we gave above for counting a pattern in a text.

For example, consider the following pseudocode for an algorithm called **DISTANCE**, whose input is four numbers (x_1, y_1, x_2, y_2) and whose output is one number d. Can you guess what it does?

```
DISTANCE($x_1$, $y_1$, $x_2$, $y_2$)
    $d \leftarrow (x_2 - x_1)^2 + (y_2 - y_1)^2$
    $d \leftarrow \sqrt{d}$
    return $d$
```

The first line of pseudocode specifies the name of an algorithm (**DISTANCE**), followed by a list of **arguments** that it requires as input (x_1, y_1, x_2, y_2). Subsequent lines contain the statements that describe the algorithm's actions, and the operation **return** reports the result of the algorithm.

We can invoke an algorithm by passing it the appropriate values for its arguments. For example, **DISTANCE**$(1, 3, 7, 5)$ would return the distance between points $(1, 3)$ and $(7, 5)$ in two-dimensional space, by first computing

$$d \leftarrow 7 - 1)^2 + (5 - 3)^2 = 40$$

and then computing

$$d \leftarrow \sqrt{40}\ .$$

The pseudocode for **DISTANCE** uses the concept of a **variable**, which contains some value and can be assigned a new value at different points throughout the course of an algorithm. To assign a new value to a variable, we use the notation

$$a \leftarrow b\,,$$

which sets the variable a equal to the value stored in variable b. For example, in the pseudocode above, when we compute **DISTANCE**$(1, 3, 7, 5)$, d is first assigned the value $(7-1)^2 + (5-3)^2 = 40$ and then is assigned the value $\sqrt{40}$.

Furthermore, we can use any name we like for variable names. For example, the following pseudocode is equivalent to the previous pseudocode for **DISTANCE**.

```
DISTANCE(x, y, z, w)
    abracadabra ← $(z - x)^2 + (w - y)^2$
    abracadabra ← $\sqrt{abracadabra}$
    return abracadabra
```

Whereas computer scientists are accustomed to pseudocode, we fear that some biologists reading this book might decide that pseudocode is too cryptic and therefore useless. Although modern biologists deal with algorithms on a daily basis, the language they use to describe an algorithm may be closer to a series of steps described in plain English.

Accordingly, some bioinformatics books are written without pseudocode. Unfortunately, this language is insufficient to describe the complex algorithmic ideas behind various bioinformatics tools that biologists use every day.

To be able to explain complex algorithmic ideas, we will need to delve deeper into the details of pseudocode. As a result, you will be able not only to understand the algorithms in this book, but use pseudocode to design your own algorithms!

Nuts and Bolts of Pseudocode

We have thus far described pseudocode only superficially. We will now discuss some of the details of pseudocode that we use throughout this book. We will often avoid tedious details by specifying parts of an algorithm in English, using operations that are not listed below, or by omitting noncritical details.

***if** conditions*

The algorithm **MINIMUM2** shown below has two numbers (*a* and *b*) as its input and a single number as its output. What do you think that it does?

```
MINIMUM2(a, b)
    if $a > b$
        return b
    else
        return a
```

This algorithm, which computes the minimum of two numbers, uses the following construction:

```
if statement X is true
    execute instructions Y
else
    execute instructions Z
```

If statement *X* is true, then the algorithm executes instructions *Y*; otherwise, it executes instructions *Z*. For example, **MINIMUM2**$(1, 9)$ returns 1 because the condition "1 > 9" is false.

The following pseudocode computes the minimum of three numbers.

```
MINIMUM3(a, b, c)
    if $a > b$
        if $b > c$
            return c
        else
            return b
    else
        if $a < c$
            return a
        else
            return c
```

We may also use **else if**, which allows us to consider more than two different possibilities in the same **if** statement. For example, we can compute the minimum of three numbers as follows.

```
MINIMUM3(a, b, c)
    if $a > c$ and $b > c$
        return c
    else if $a > b$ and $c > b$
        return b
    else
        return a
```

Both of these algorithms are correct, but below is a more compact version that uses the **MINIMUM2** function that we already wrote as a **subroutine**, or a function that is called within another function. Programmers break their programs into subroutines in order to keep the length of functions short and to improve readability.

```
MINIMUM3(a, b, c)
    if $a > b$
        return MINIMUM2(b, c)
    else
        return MINIMUM2(a, c)
```

STOP and Think: Can you rewrite **MINIMUM3** using just a single line of pseudocode?

EXERCISE BREAK: Write pseudocode for an algorithm **MINIMUM4** that computes the minimum of four numbers.

Sometimes, we may omit the "**else**" statement.

for *loops*

Consider the following problem.

Summing Integers Problem:
Compute the sum of the first n positive integers.

Input: A positive integer n.
Output: The sum of the first n positive integers.

If n were a fixed number, then we could solve this problem using our existing framework. For example, the following program **SUM5** returns the sum of the first five integers (i.e., $1 + 2 + 3 + 4 + 5 = 15$).

```
SUM5()
    sum ← 0
    i ← 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    i ← i + 1
    sum ← sum + i
    return sum
```

We could then write **SUM6**, **SUM7**, and so on. However, we cannot endorse this programming style! After all, to solve the Summing Integers Problem for an *arbitrary* integer n, we will need an algorithm that takes n as its input. This is achieved by the following algorithm, which we call **GAUSS**.

```
GAUSS(n)
    sum ← 0
    for i ← 1 to n
        sum ← sum + i
    return sum
```

GAUSS employs a **for** loop that has the following format:

```
for $i \leftarrow a$ to $b$
    execute $X$
```

This **for** loop first sets i equal to a and executes instructions X. Afterwards, it increases i by 1 and executes X again. It repeats this process by increasing i by 1 until it becomes equal to b, when it makes a final execution of X. That is, i varies through the values $a, a+1, a+2, \ldots, b-1, b$ during execution of the loop.

***while** loops*

A different way of summing the first n integers, called **ANOTHERGAUSS**, is shown below.

```
ANOTHERGAUSS($n$)
    $sum \leftarrow 0$
    $i \leftarrow 1$
    while $i \leq n$
        $sum \leftarrow sum + i$
        $i \leftarrow i + 1$
    return $sum$
```

This algorithm uses a **while** loop having the following format:

```
while statement $X$ is true
    execute $Y$
```

The **while** loop checks condition X; if X is true, then it executes instructions Y. This process is repeated until X is false. (Note: if X winds up always being true, then the **while** loop enters an **infinite loop**, which you should avoid at all costs, because your algorithm will never end.) In the case of **ANOTHERGAUSS**, the loop stops executing after n trips through the loop, when i eventually becomes equal to $n+1$ and the numbers 1 through n have been added to *sum*.

Recursive algorithms

Below is yet another algorithm solving the Summing Integers Problem.

```
RECURSIVEGAUSS($n$)
    if $n > 0$
        $sum \leftarrow$ RECURSIVEGAUSS($n - 1$) $+ n$
    else
        $sum \leftarrow 0$
    return $sum$
```

You may be confused by the fact that **RECURSIVEGAUSS**(n) calls **RECURSIVEGAUSS**$(n - 1)$ as a subroutine. So imagine that you are computing the sum of the first 100 positive integers, but you are lazy and ask your friend to compute the sum of the first 99 positive integers for you. As soon as your friend has computed it, you will simply add 100 to the result, and you are done! Yet little do you know that your friend is just as lazy, and she asks her friend to compute the sum of the first 98 integers, to which she adds 99 and then passes to you. The story continues until a friend is assigned 1. Although every individual in this chain of friends is lazy, the group is nevertheless able to compute the sum.

RECURSIVEGAUSS presents an example of a **recursive algorithm**, which subcontracts a job by calling itself (on a smaller input).

EXERCISE BREAK: Can you write one line of pseudocode solving the Summing Integers Problem?

You have undoubtedly been wondering why we have named all these summing algorithms “Gauss”. In 1785, a primary school teacher asked his class to sum the integers from 1 to 100, assuming that this task would occupy them for the rest of the day. He was shocked when an 8 year old boy thought for a few seconds and wrote down the answer, 5,050. This boy was Karl Gauss, and he would go on to become one of the greatest mathematicians of all time. The following one-line algorithm implements his idea for solving the Summing Integers Problem. (Why does it work?)

```
GAUSS($n$)
    return $(n + 1) \cdot n/2$
```

Arrays

Finally, when writing pseudocode, we may also use an **array**, or an ordered sequence of variables. We often use a single letter to denote an array, e.g., $a = (a_1, a_2, \ldots, a_n)$.

EXERCISE BREAK: Consider the algorithm following this exercise. What does it do?

```
RABBITS(n)
    $a_1 \leftarrow 1$
    $a_2 \leftarrow 1$
    for $i \leftarrow 3$ to $n$
        $a_i \leftarrow a_{i-1} + a_{i-2}$
    return $a$
```

RABBITS(n) computes the first n Fibonacci numbers and places them in an array. Why do you think that we called it RABBITS?