Chapter 2
Algorithm Design

Before we discuss how to design algorithms, we need to make up our minds as how to represent them, i.e., what notation should we use to express algorithms so that they are clear, precise, and unambiguous.

We don’t want to use a programming language at this stage since we want to be flexible but not have to deal with details. 😞

Natural language is not a good choice either, since 1) the verbosity leads to an unstructured description; 2) it is live, thus unstable, so that it cannot be finitely specified; and 3) the context sensitivity could lead to ambiguity;
Too much details...

The following Java program implements the addition algorithm that we went through in the last chapter, on Page 11.

```java
{  
    Scanner inp = new Scanner(System.in);
    int i, m, carry;
    int[] a = new int[100];
    int[] b = new int[100];
    int[] c = new int[100];
    m = inp.nextInt();
    for (int j = 0; j <= m-1; j++) {
        a[j] = inp.nextInt();
        b[j] = inp.nextInt();
    }
    carry = 0;
    i = 0;
    while (i < m) {
        c[i] = a[i] + b[i] + carry;
        if (c[i] >= 10)
            .
            .
            .
    }
}
```

We will eventually come here, but not yet.
What do you mean?

Given the following statement:

“Call me a taxi.”

**Question:** Does this person ask someone to get a taxi, or wants to be called a “taxi”? 😁

Another example could be

“I saw someone on the hill with a telescope.”

**Question:** Who has a telescope? 😞

Remember the non-ambiguous and feasible requirement?
What will we use?

*Pseudocode* is a subset of English language constructs that looks like the statements available in most programming languages.

It is simple, flexible, and highly readable. With its well-defined structure, it is easier to visualize the organization of a pseudocode algorithm. Finally, it is also easier to transform a pseudocode algorithm into a computer program, since its syntax resembles many programming languages.

To start, we will present the three basic structures: *sequential, conditional and, iterative constructs*, in pseudocode.

We will then see a few exemplary algorithms expressed in pseudocode.
Sequential operations

The fundamental operation sequence involved in every algorithm is *input, computation, output*.

*Assignment* is an instruction that performs a computation and then saves the result.

Set the value of "variable" to "expression"

It *evaluates* the arithmetic expression first and gets a result, which is then stored in the variable. The latter corresponds to a named storage location.

For example,

Set the value of Carry to $3 \times 4 - 5$

More generally,

variable = expression;
Input, compute, output

*Input* operations allow the computing agent to receive data from the outside world, which can then be used in a *computation*, while *output* operations allow the agent to send out results of the computation for future use.

The following algorithm, in pseudocode, computes *average miles per gallon*.

1. Get values for *gallons*, *start* and *end*

2. Set the value of *distance* to (*end* – *start*)

3. Set the value of *average* to (*distance*/*gallons*)

4. Print the value of *average*

5. Stop
One more example

Write an algorithm that gets the values of radius $r$ of a circle as input, then output both the circumference and the area of such a circle.

1. Get value for $r$

2. Set $\text{circumference}$ to $2 \times 3.14 \times r$

3. Set $\text{area}$ to $3.14 \times r \times r$

4. Print ”Circumference is ” + $\text{circumference}$

5. Print ”Area is ” + $\text{area}$

6. Stop

Here, ‘+’ represents the concatenation operator.

**Homework:** Exercise 1, think about Exercise 2
Conditional operation

A sequence structure starts at the beginning, goes forward, until the end, then stops. On the other hand, a *conditional operation* lets the algorithm ask a question, and, based on the answer, selects the next operation to perform. Below is the most commonly used structure.

If "a true-false" condition is true
Then
    first set of operations
Else second set of operations.

It *evaluates* the condition first to see if it is *true* or *false*, and then executes the first, or the second set, of operations, accordingly.

In either case, the execution of the program continues with the *next* operation right after this conditional operation.
An example

Both Honda CR-V and Subaru Forester reach a respectable 28 MPG.

**Question:** How to decide if another car is even better?

**Answer:** We can run the following algorithm to decide (The first part (1-4) came up on Page 6):

1. Get values for *gallons, start* and *end*
2. Set the value of *distance* to (*end* − *start*)
3. Set the value of *average* to (*distance/gallons*)
4. Print the value of *average*
5. If *average* > 28 Then
6. Print the massage “You are getting good gas mileage.”
7. Else Print the massage “You are not getting a good gas mileage.”
8. Stop
How does it work?

Below shows the flowchart of a conditional structure, telling us what does it do..., where $S$ is the next thing to do once this conditional structure is completed.

Assignment: Have a look at, and try, some of the problems in Practice Problems on Page 61 of the textbook.

You don't need to send in your work for these assignments.
Another example

Write an algorithm that inputs your current credit card balance, the total dollar amount of new purchases, and the total amount of all payments. The following algorithm computes the new balance, including a 20.94% interest charge on any unpaid balance.

1. Get values for \textit{balance}, \textit{purchase}, and \textit{payment}
2. Set \textit{unpaid} to \textit{balance} – \textit{payment}
3. If (\textit{unpaid} > 0)
4. Set \textit{charge} to \textit{unpaid} * 0.2094
5. Else set \textit{charge} to 0
6. Set \textit{newBalance} to \textit{charge} + \textit{purchase}
7. Print ”New balance is ” + \textit{newBalance}

\textbf{Homework:} Exercise 5
Iterative operation

The *While..Do* loop structure is pretty popular:

```
While "a condition" remains true do
  operation
  ...
  operation
```

We can also use *Do..While*:

```
Do
  operation
  ...
  operation
While "a condition" remains true
```

The *Do..While* loop goes through at least once, then checks the condition.

On the other hand, the *While..Do* loop checks condition first, thus it may not execute even once. It is thus a more cautionary structure. 😊
How does it work?

Below shows the flowchart of a `while..do` loop structure, which keeps on going through the loop until the loop condition becomes false.

Here $S_n$ is the next thing to do after this loop is done.

Remember that the well-founded requirement says that the algorithm knows where to start, and when one step is done, an algorithm needs to know what to do next.
An example

The following algorithm measures the mileage of multiple cars until we get bored with it. 😊

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>response = Yes</td>
</tr>
<tr>
<td>2</td>
<td>While (response = Yes) do Steps 3 through 11</td>
</tr>
<tr>
<td>3</td>
<td>Get values for gallons used, starting mileage, ending mileage</td>
</tr>
<tr>
<td>4</td>
<td>Set value of distance driven to (ending mileage – starting mileage)</td>
</tr>
<tr>
<td>5</td>
<td>Set value of average miles per gallon to (distance driven ÷ gallons used)</td>
</tr>
<tr>
<td>6</td>
<td>Print the value of average miles per gallon</td>
</tr>
<tr>
<td>7</td>
<td>If average miles per gallon &gt; 28 then</td>
</tr>
<tr>
<td></td>
<td>Print the message ‘You are getting good gas mileage’</td>
</tr>
<tr>
<td>8</td>
<td>Else</td>
</tr>
<tr>
<td></td>
<td>Print the message ‘You are NOT getting good gas mileage’</td>
</tr>
<tr>
<td>9</td>
<td>Print the message ‘Do you want to do this again? Enter Yes or No’</td>
</tr>
<tr>
<td>10</td>
<td>Get a new value for response from the user</td>
</tr>
<tr>
<td>11</td>
<td>Stop</td>
</tr>
</tbody>
</table>

Algorithm design is often an incremental process, following the *Divide and Conquer* principle: Start with something simple, then keep on adding more and more until we have got the complete algorithm for the task at hand.

We start with one car (Page 6), add on comparison (Page 10), and finally, the loop part for multiple cars.
How about *Do..While*?

The *Do..While* loop puts the test at the end, thus executes at least once. 😐
How about \textit{for}?

The popular \textit{for} loop runs a \textit{fixed number of time}.

If $S(i)$ is any statement which changes the loop variable $i$, then the following loop runs exactly $n$ times.

\begin{verbatim}
for (i=1; i<=n; i++)
    S(i);
\end{verbatim}

A \textit{for} loop is not real, but implemented in a \textit{while} loop:

\begin{verbatim}
i=1;
while(i<=n){
    S(i); i++;
}
\end{verbatim}

Wait until \textit{CS 2470 System Programming in C/C++} for a more general syntax of the \textit{for} loop; and \textit{CS 3221 Algorithm analysis} for its analysis.
What should you do?

While still thirsty
  Keep on drinking

Keep on drinking
While still thirsty

Repeat
  drinking
Until not thirsty

Drink three cups

**Theorem:** If a problem can be solved algorithmically, then it can be solved using only the sequential, conditional, and iterative operations.

Thus, *an algorithm is just a combination of these three structures.* 😊
Revisit multiplication

**Question:** How to multiply two numbers through repeated addition? (Exercise 12 in Chapter 1)

Given two nonnegative integer values, \( a \geq 0 \), and \( b \geq 0 \), compute and output the product \((a \times b)\) using the technique of repeated addition.

That is,

\[
P = a \times b = a + a + a + \cdots + a.
\]

The first step is to bring in the values of both \( a \) and \( b \). We then repeatedly add \( a \) to a partial product. Thus, it is natural to use a loop structure to get it done.

We will add \( a \) a total of \( b \) times, we could use a variable, \( count \), with an initial value of 0. We will increment \( count \) once whenever we add another \( a \) until \( count \) reaches \( b \).

**Question:** What is the first crack?
A first attempt

1. Get values for \(a\) and \(b\)
2. Set \(product\) to 0
3. Set the value of \(count\) to 0
4. While \((count < b)\) do
5. Set \(product\) to \(product + a\)
6. Set the value of \(count\) to \((count + 1)\)
7. Print the value of \(product\)

<table>
<thead>
<tr>
<th>(product)</th>
<th>(count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2a</td>
<td>1</td>
</tr>
<tr>
<td>3a</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(ba)</td>
<td>(b - 1)</td>
</tr>
<tr>
<td>(ba)</td>
<td>(b)</td>
</tr>
</tbody>
</table>

Notice we multiply first then update \(count\). Once \(count\) contains \(b\), we have got \(a \times b\) done, when the loop condition fails. We wrap up and print out the value of \(product\), namely, \(ab \ (\equiv ba)\) since multiplication is commutative.

😊
A little tweaking

The assumption of the original multiplication problem is that both $a$ and $b$ are non-negative, i.e., $a \geq 0$ and $b \geq 0$. The code that we just got certainly works when both $a$ and $b$ are positive.

**Question:** Will it work correctly when either $a = 0$ or $b = 0$?

**Answer:** Yes.

The case of $b = 0$ works out nicely. When $a = 0$, although we do get the right answer of 0 back, the algorithm is not *efficient*, as it keeps on adding 0 $b$ times to *product*. 😞

What we could do is to add the following before getting into the costly loop

If(either $a = 0$ or $b = 0$)
   Set the value of *product* to 0
Else Solve the original problem
A final solution

1. Get values for $a$ and $b$
2. Set the value of $product$ to 0
3. If ($a! = 0$ and $b! = 0$)
4. Set the value of $count$ to 0
5. While ($count < b$) do // Not yet
6. Set $product$ to $product + a$
7. Set the value of $count$ to ($count + 1$)
8. Print the value of $product$
9. Stop

**Question:** Does it work?

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$product$</th>
<th>$count$</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>4</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>60</td>
<td>4</td>
</tr>
</tbody>
</table>
Who’s calling?

The second problem we want to discuss in detail is to search for somebody’s name, if we know her telephone number, often called the reverse telephone lookup.

**Question:** Should you pick up the phone, when a number pops up?

**Answer:** Let’s find out who is calling?

**Assignment:** Have a look at Practice Problems on Page 64 of the textbook, and try at least Problems 3 and 4.
Keep on looking...

Assume that we have 10,000 names, $N_1, \cdots, N_{10000}$, along with that many phone numbers, $T_1, \cdots, T_{10000}$.

We further assume that we are using a normal phone book, which is sorted in the alphabetical order of last names.

Let's come up with an algorithm that bring in a phone number (input), and find out the owner's name, using a phone book (output).

The general idea (algorithm) is to start with the first record, to see if its phone number matches with the input number. If it is, we are done; otherwise, move down to the next record, ..., until we either find it somewhere 😊, or declare a failure when we have checked the last record, but still could not find it.. 😞

Let's turn this idea into an algorithm.
A first try

Get $Number, T_1, \cdots, T_{10000} N_1, \cdots, N_{10000}$,
If $Number == T_1$ Then write out $N_1$
If $Number == T_2$ Then write out $N_2$
...
If $Number == T_{10000}$ Then write out $N_{10000}$
Stop

This algorithm is extremely long. (Eight and half million people live in NYC.) It also tells us nothing when the number is not in the list. 😞

Since we will be doing the same thing repeatedly, the first problem can be fixed by using a loop, while the second can also be solved by checking the index at the end.

**Question:** What will this lead us to?

**Homework:** Exercise 13 and think about Exercise 15
An improvement

Get $\text{Number}, T_1, \ldots, T_{10000}, N_1, \ldots, N_{10000}$,
Set $i$ to 1 and set $\text{Found}$ to NO
While $\text{Found} == \text{NO}$ and $i \leq 10000$ Do
  If $\text{Number} == T_i$ Then
    Print $N_i$
    Set $\text{Found}$ to YES
  Else Add 1 to $i$
  If $(\text{Found} == \text{NO})$ Then
    Print “Sorry, the number is not in the book.”
Stop

*This is essentially the algorithm that we will play with in Lab 2 with the Search Animator.*

**Questions:** Is this the way we look for a number in the phone book?

How could we adjust our records to make such a search much faster?
Sequential search

When we want to look for \textit{target} in an \textit{unordered} array \(A[1..n]\), we use the following \textit{sequential search algorithm}.

Get \textit{target} \\
Set \(i\) to 1 and \textit{Found} to NO \\
While both \textit{Found} \(==\) NO and \(i \leq n\) Do \\
\hspace{1em} If \(A[i] == \textit{target}\) Then \\
\hspace{2em} Print \(i\) \\
\hspace{2em} Set \textit{Found} to YES \\
\hspace{1em} Else Add 1 to \(i\) \\
If (\textit{Found} \(==\) NO) Then \\
Print \text{“Sorry, the number is not in the book.”}

As we found out in Lab 2, it will make at least one comparison, when \(A[1] = \textit{target}\); and will make at most \(n\) comparisons, when either \(A[n] = \textit{target}\), or \textit{target} is not in \(A\).

On average, it makes \((n + 1)/2\) comparisons.
Big, bigger, biggest

The next problem we want to solve is similar to the previous one, in the sense that we still look for something in a list of items.

This time we will look for the biggest value, but not a particular one. It is not only useful by itself, but can be used to sort a list of items (Selection sort, Page 9 in the next chapter).

Formally, the problem can be specified as follows: Given a value $n \geq 2$, and a list containing exactly $n$ unique numbers, $A_1, \ldots, A_n$. find and print out both the largest value in the list and its position.

For example, given 19, 41, 12, 63, 22. The algorithm should print out 63 and 4.
Bear in the corn field

A bear is hungry. He walks into a corn filed, trying to get the biggest ear of corn.

What he would do is to get the first ear, then walk forward. Whenever he sees a bigger ear, he drops the one from his mouth, and gets the bigger one.

He keeps on walking, picking, dropping, and snatching, ... until he is out of the filed, when he has got the biggest ear.

**Question:** What should we learn from this bear? ☺️
Here it goes ...

Intuitively, we have to search the whole list to find out the biggest value, and we have to save this value and its position somewhere.

With respect to the searching, let’s begin with the first item, check the values one at a time, until we have looked at all the values.

As we can’t be sure about the answer until the last step, let’s also keep the biggest value we have seen *so far* in a pair of variables, and *keep on updating this pair* during the searching process.

These ideas lead to the following alternative, and straightforward, algorithm.
A general algorithm

Get $n, A_1, \ldots, A_n$
Set $Location$ to 1
Set $Largest$ to $A_1$
Set $i$ to 2
While ($i \leq n$) Do
  If $A_i > Largest$ Then
    Set $Largest$ to $A_i$
    Set $Location$ to $i$
  Add 1 to $i$
Print out $Largest$ and $Location$

Questions: 1) How can we modify the algorithm so that we can find the smallest value in the list? 2) What happens if $n$ equals 0 or 1?

We will play with this algorithm in Lab 3 with the Search Animator

It always makes $n - 1$ comparisons, $n \geq 2$, because the loop runs this many times, and each time it runs, one comparison is made.

Homework: Exercises 16 and 17.