CMPS 10
Introduction to Computer Science
Lecture Notes

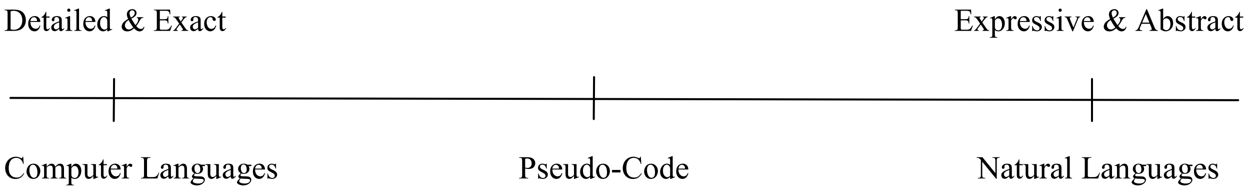
Chapter 2: Algorithm Design

How should we present algorithms?

Natural languages like English, Spanish, or French which are rich in interpretation and meaning are not ideal for this purpose. We need a more precise notation to present algorithms that is not subject to differing interpretations.

At the other extreme *formal programming languages* (such as C, C++, Java, Perl, ADA, Python, etc.) are very inflexible in their syntax, and require much attention to detail. These details may be irrelevant in the initial design phases of an algorithm.

Instead we use an informal and flexible language called *pseudo-code* which uses English language constructs and conventions modeled to look like commands available in most computer languages.



Pseudo-code is a language in which all necessary operations can be expressed, but which is independent of any computer language, or of any computer for that matter.

Sequential Operations are of three basic kinds: *computation*, *input*, and *output*. The instruction for performing a calculation and storing the result is written as

$$\text{variable} \leftarrow \text{expression}$$

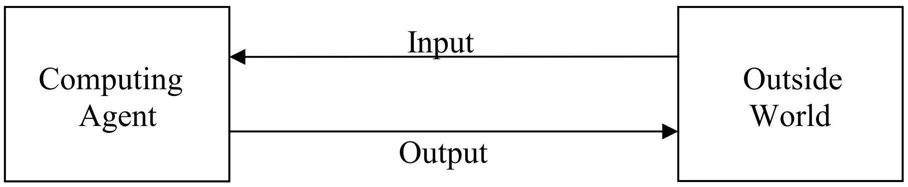
This instruction tells the computing agent to evaluate the expression on the right and assign its value to the variable on the left. The left arrow $\leftarrow$ in this context is called the *assignment operator*.

Example $c_i \leftarrow a_i + b_i + \text{carry}$

Example $x \leftarrow \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$

Unless otherwise stated, we will assume that the computing agent is at least as capable as any scientific calculator, so that the above expressions are considered primitive operations.

Input and output operations allow the computing agent to receive data values from, and send results to the outside world, which we also call the "user".



The operations used for input and output are *get* and *print* respectively.

```
get   variable
print   variable
print   expression
```

Example get x

Example print x

Example print $x+5$

Text to be printed will be placed in single quotes.

Example print 'error: division by zero'

Example print 'the value of x is: ' x 'and the value of y is: ' y

An algorithm which uses only sequential operations is sometimes called a *straight line* algorithm.

Example

```
1) get  $a, b, c, d$
2) sum $\leftarrow a+b+c+d$
3) average $\leftarrow$ sum / 4
4) print  average
```

What does this algorithm do?

Conditional operations (also called *branching operations*) are expressed using the *if* statement or the *if-else* statement. The if statement is written as

```
if  condition
      do something
      do something
      ..
      do something
do something
..
```

In this structure “condition” stands for a *logical* (or *Boolean*) expression, i.e. an expression that can be evaluated as true or false. If condition is true, then the indented lines are executed, followed by the unindented lines. If on the other hand condition is false, then the indented lines are skipped, and execution picks up at the next unindented line. The indented commands are known as the *true branch* of the conditional. An if statement may also come with an accompanying else clause.

```
if condition
    do something
    ..
    do something
else
    do something
    ..
    do something
do something
..
```

If condition is true, then the first group of indented instructions is executed, and the second group is skipped, followed by the next unindented instruction. If condition is false, then the first group is skipped, and the second group of indented instructions is executed, followed by the next unindented instruction. In this structure the second group of indented instructions is known as the *false branch* of the conditional, while the first group is know as the true branch, as before.

Example What does the following algorithm do?

```
1) get $a$, $b$
2) if $b = 0$
3)       print 'Error: division by zero'
4) else
5)       quotient ← $a / b$
6)       print quotient
7) stop
```

Iterative operations (also called *loops*) will be expressed in pseudo-code through three commands: *while*, *do-while*, and *for*.

```
while condition
    do something
    ..
    do something
do something
..
```

First condition is tested, and if found to be true, the indented lines (called the *loop body*) is executed. The condition is tested again, and if found to be true, the loop body is executed again. This process continues until the condition is found to be false, in which case execution picks up at the next unindented line. The while loop condition is sometimes called a *loop repetition condition* since the loop body repeats only so long as this logical expression is true.

Example What does the following algorithm do?

```
1) response ← 'yes'
2) while response = 'yes'
3)     get $a, b$
4)     if $b = 0$
5)         print 'Error: division by zero'
6)     else
7)         print $a / b$
8)     print 'Do you wish to continue?'
9)     get response
10) stop
```

The do-while loop structure puts the loop repetition condition after the loop body.

```
do
    do something
    ..
    do something
while condition
do something
..
```

First the loop body, which consists of just those lines that are indented, is executed. Then the condition is tested, and if found to be true, the loop body is repeated. We continue executing the loop body until condition becomes false, then execution picks up at the next unindented line after while. The loop repetition condition is called a *post test* condition in this case since it is tested *after* the first execution of the loop body. The condition in the while loop on the other hand is called a *pre test* condition, since it is tested *before* the first execution of the loop body.

Example Notice that the loop body in this example is identical to the one in the preceding example.

```
1) do
2)     get $a, b$
3)     if $b = 0$
4)         print "Error: division by zero'
5)     else
6)         print $a / b$
7)     print 'Do you wish to continue?'
8)     get response
9) while response = 'yes'
10) stop
```

We often encounter while loops with the following general structure

```
$i \leftarrow 1$    **Initialize the loop control variable *i***
while $i \le n$    **Test the loop repetition condition**
    do something with $i$
    $i \leftarrow i + 1$    **Increment the loop control variable**
```

We can do these three steps, initialize loop control variable, test loop repetition condition, and increment loop control variable, more succinctly with the *for* loop structure below, which is equivalent to the preceding while loop.

```
for $i \leftarrow 1$ to $n$
    do something with $i$
```

Example Here are two equivalent algorithms, one uses a while loop, and the other uses an equivalent for loop. A trace of both algorithms would be identical.

```
1) $i \leftarrow 1$
2) while $i \le 5$
3)         print $i^2$
4)         $i \leftarrow i+1$
5) stop
```

```
1) for $i \leftarrow 1$ to 5
2)         print $i^2$
3) stop
```

It has been proved that one can represent *any* valid algorithm with the operations so far described. We now consider some well known algorithms which perform certain basic tasks in computer science.

Example Consider the problem of looking up a name and corresponding telephone number in a directory

$$\begin{tikzcd}
 & \underline{\textbf{Names}} & \underline{\textbf{Numbers}} \\
 & N_1 & T_1 \\
 & N_2 & T_2 \\
 & N_3 & T_3 \\
 & \vdots & \vdots \\
 & \vdots & \vdots \\
\text{NAME} \arrow[r] & N_i \arrow[r] & T_i \\
 & \vdots & \vdots \\
 & \vdots & \vdots \\
 & N_n & T_n
\end{tikzcd}$$

We assume that the names are in no particular order (i.e. not necessarily in alphabetical order) so that we must look at each name in succession. If we find a name N_i which matches our target NAME, we return the corresponding number T_i. If no N_i is a match, we print a message to that effect. We call this algorithm **Sequential Search**.

<u>Input</u>: $n \ge 1$ (the number of names in the list), $N_1, \cdots, N_n$ and $T_1, \cdots, T_n$ (the list of names and corresponding telephone numbers), and NAME (the target name we are searching for.)
<u>Output</u>: the telephone number T_i for which $\text{NAME} = N_i$, or if no such name exists, print a message to the user.

Sequential Search

```
1.)  Get $n, N_1, \ldots, N_n, T_1, \ldots, T_n$, NAME from the user
2.)  $i \leftarrow 1$
3.)  found $\leftarrow$ false
4.)  while $i \leq n$ **and** **not** found
5.)      if $N_i$ = NAME
6.)          found $\leftarrow$ true
7.)          print $T_i$
8.)      else
9.)          $i \leftarrow i+1$
10.) if **not** found
11.)     print 'sorry' NAME 'not in directory'
12.) stop
```

The variable **found** is called a *logical variable* or *Boolean variable* (named after the Logician George Boole). It stores one of the two logical values **true** or **false**. A *logical expression* is an expression which evaluates to one of these two values. In order to understand how these expressions are evaluated, we must first define the *logical operators* **and**, **or**, and **not**.

A *proposition* is a statement or assertion which can (at least in principle) be evaluated as being either true or false. For instance 'today is Wednesday' and '$6 < 7$' are propositions, while 'hello' and '$x < 7$' are not propositions. Let A and B be *propositional variables*, i.e. variables which stand for unspecified propositions. We can form the *compound propositions* 'A and B', 'A or B', and 'not A', using the operators **and**, **or**, and **not**, which are defined as follows.

A	**B**	**A and B**
F	F	F
F	T	F
T	F	F
T	T	T

In this *truth table*, 'T' stands for true and 'F' stands for false. In words, 'A and B' is true only in the case when both operands are true, and is false in every other case.

A	**B**	**A or B**
F	F	F
F	T	T
T	F	T
T	T	T

Again in words, 'A or B' is false only in the case when both operands are false, and is true in every other case.

A	**not A**
F	T
T	F

Thus 'not A' just reverses the logical value of its operand A. The operators **and** and **not** have the same meanings that any speaker of English would expect. However there is some ambiguity in the English language concerning the word ‘or’. The above truth table for **or** defines an operation which is more properly called **inclusive or**. It asserts that either A, or B, or possibly both are true There is another meaning of the word ‘or’ in the English language called **exclusive or**, abbreviated **exor**. This operation is defined by the following truth table.

A	**B**	**A exor B**
F	F	F
F	T	T
T	F	T
T	T	F

Thus ‘A exor B’ asserts that either A, or B, but not both are true. Unfortunately for speakers of English, there is no such word as ‘exor’, and both meanings are denoted by the word ‘or’. The listner is expected to gather the meaning from the context.

Example A man with a gun walks up to you and says ‘your money or your life’. Which ‘or’ do you hope he is using?

Fortunately in Mathematics and Computer Science there is no ambiguity. Whenever you see the word ‘or’ used in a Math textbook, it means **inclusive or**. If we ever mean **exclusive or**, we explicitly write ‘exor’ (or perhaps some other symbol).

Now we can see exactly how the logical expressions on lines 4 and 10 or Sequential Search will be evaluated. There are many other examples of searching problems where the data to be searched is in no particular order. The algorithmic solution is essentially the same as that of the phone book example, i.e. step through the data from beginning to end until the target data item is found. It is conceptually very useful and elegant to consider all these examples as being different instances of the same problem, all solvable by the same algorithm. Since everything in Computer Science is ultimately represent able by numbers, we can present this Generalized Sequential Search algorithm as a search of an unsorted list of numbers for some target number.

Input: $n \geq 1$ (the number of numbers in the list), $a_1, \cdots, a_n$ (the list of numbers itself), and target (the number to search for.)
Output: The first index i such that target $= a_i$, or 0 if no such index exists.

Sequential Search

```
1.)  Get $n, a_1, \ldots, a_n$, target from the user
2.)  $i \leftarrow 1$
3.)  found $\leftarrow$ false
4.)  while $i \leq n$ and not found
5.)      if $a_i$ = target
6.)          found $\leftarrow$ true
7.)      else
8.)          $i \leftarrow i+1$
9.)  if not found
10.)     $i \leftarrow 0$
11.) print $i$
12.) stop
```

Example Trace this algorithm on the input: $n = 5$, target $= 2$, and the list 3, -1, 2, 5, 12.

Exercise Modify this algorithm to find the last (i.e. rightmost) occurrence of the target. How can we modify it to find the 2nd, or 5th, or jth occurrence of the target? How can we modify it to print the number of occurrences of the target?

Another common problem in processing a list of numbers is to find the largest or smallest number in the list. The following algorithm, called Find Largest, solves this problem.

Input: $n \geq 1$ (the number of numbers), and $a_1, \cdots, a_n$ (the list of numbers, which we assume to be distinct).
Output: The largest value in the list, together with the position (i.e. the index i) where it is located.

Find Largest
```
1.)  max ← $a_1$    (max is the largest value seen so far)
2.)  $j \leftarrow 1$        ($j$ is the index of the largest value seen so far)
3.)  $i \leftarrow 2$        ($i$ is the index of the element we are presently looking at)
4.)  while  $i \leq n$
5.)      if  $a_i >$ max
6.)          max ← $a_i$
7.)          $j \leftarrow i$
8.)      $i \leftarrow i+1$
9.)  print  max, $j$
10.) stop
```

Example Trace this algorithm on the list 2, 1, -7, 5, 3, 11, 0.

Exercise What happens if we run Find Largest on a list whose elements are not distinct? Try it on the list 1, 3, 0, 10, 3, 5, 10. How can it be modified to find all occurrences of the maximum? How can it be modified to find the minimum?

Exercise Assume that addition of integers is a primitive operation for the computing agent, but that multiplication of integers is not. Write an algorithm which takes as input two (non-negative) integers a and b, and prints out their product $a \cdot b$. Obviously one cannot write 'print $a \cdot b$' since we are assuming here that multiplication is not primitive. The problem is to define multiplication algorithmically in terms of repeated addition, as follows.

```
1.)  get  $a, b$
2.)  product ← 0
3.)  $i \leftarrow 1$
4.)  while  $i \leq b$
5.)      product ← product + $a$
6.)      $i \leftarrow i+1$
7.)  print  product
8.)  stop
```

This algorithm could be written a little more succinctly using a for loop.

```
1.)  get  $a, b$
2.)  product $\leftarrow 0$
3.)  for  $i \leftarrow 1$  to  $b$
4.)       product $\leftarrow$ product $+ a$
5.)  print  product
8.)  stop
```

Exercise Now assume that multiplication is a primitive operation for the computing agent, but that exponentiation (i.e. raising a number to a power) is not. Write an algorithm which takes as input two (non-negative) integers *a* and *b*, then prints out the value a^b. Again it is obvious that statement 'print a^b' is not a valid operation. The problem is to define exponentiation in terms of repeated multiplication.

A classical problem in Computer Science is Pattern Matching. There are many versions of this problem dealing with patterns in graphics, sound, pictures, and other types of data. We will consider a simple form of this problem: finding patterns in text data. More precisely, given *n* characters of text

$$T_1 T_2 T_3 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots T_n$$

and given a pattern of *m* characters, where $m \le n$

$$P_1 P_2 P_3 \ldots\ldots P_m$$

find every occurrence of the pattern within the text, i.e. find each index position in the text at which the pattern begins.

Example $n = 18,\ m = 2$
text: 'to be or not to be'
pattern: 'be'
answer: 4, 17

Example $n = 10,\ m = 3$
text: 'xxxaaaaaxx'
pattern: 'aaa'
answer: 4, 5, 6

Notice that if $m = 1$, i.e. if the pattern has just one character in it, then this problem is not essentially different from the searching problem. We may expect that our algorithm will show some similarity to Sequential Search. Our solution will be to simply compare the pattern, one character at time, to *m* contiguous text characters at every possible starting location within the text. We start by comparing P_1 to T_1, and if they match compare P_2 to T_2, and if they match compare P_3 to T_3, etc. We continue in this manner until reach the end of the pattern and thus establish that the pattern matches the text at position 1, or until we find two characters which don't match, indicating a mismatch at position 1. We then slide the pattern over one position in the text and compare P_1 to T_2, then P_2 to T_3, etc. as before until we either establish a match or a mismatch at position 2 in the text. Every time a match occurs, we print out the index within the text at which the pattern and text matched.

Example $n = 7,\ m = 3$

$$\begin{array}{llllllllll}
 & & & & & & & & & \text{check for match at position} \\
T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & & & i = 1 \\
P_1 & P_2 & P_3 & \rightarrow \text{ slide over} & & & & & & \\
 & & & & & & & & & \\
T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & & & i = 2 \\
 & P_1 & P_2 & P_3 & \rightarrow & & & & & \\
 & & & & & & & & & \\
T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & & & i = 3 \\
 & & P_1 & P_2 & P_3 & \rightarrow & & & & \\
 & & & & & & & & & \\
T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & & & i = 4 \\
 & & & P_1 & P_2 & P_3 & \rightarrow & & & \\
 & & & & & & & & & \\
T_1 & T_2 & T_3 & T_4 & T_5 & T_6 & T_7 & & & i = 5 \\
 & & & & P_1 & P_2 & P_3 & & &
\end{array}$$

In this case we see that the possible answers, i.e. the set of possible indices *i* at which a match can occur, are $i = 1$ to $i = 5$, since to slide the pattern over one more time would cause it to fall off the end of the text. In general the indices *i* to be tested range from $i = 1$ to $i = n - m + 1$.

<u>Input</u>: Integers *n* and *m* satisfying $1 \le m \le n$, the text $T_1T_2T_3 \ldots\ldots\ldots T_n$ to be searched, and the pattern $P_1P_2P_3 \ldots P_m$ to search for.
<u>Output</u>: All indices *i* such that $P_1P_2P_3 \ldots P_m$ matches $T_iT_{i+1}T_{i+2} \ldots T_{i+m-1}$.

<u>Pattern Match</u>
```
1.)  $i \leftarrow 1$
2.)  while  $i \le n - m + 1$
3.)       $j \leftarrow 1$
4.)       match $\leftarrow$ true
5.)       while  $j \le m$  and  match
6.)           if  $P_j \ne T_{i+j-1}$
7.)               match $\leftarrow$ false
8.)           else
9.)               $j \leftarrow j + 1$
10.)      if  match
11.)          print  ‘match found at position ’  $i$
12.)      $i \leftarrow i + 1$
13.) stop
```

Exercise Trace this algorithm on some simple examples like ‘to be or not to be’ or ‘xxxaaaaaxx’ above. Search for various patterns within these texts.

Exercise How can this algorithm be used to find words rather than patterns? For instance consider the text ‘hand the band to randy and I’, and search for the pattern ‘and’. The answers will be positions 2, 11, 18, and 23. However the word ‘and’ appears only once, at position 23. How can we use this algorithm to find only the word ‘and’ and not all occurrences of the pattern ‘and’ within other words?