Programming Concepts

Part 2: A language for algorithms

Topics

Reading Suggestions

- Dowek. Chapter 1
- Harel. Chapters 1–3

Purpose of algorithms

- Algorithms are designed for person-to-person communication
- Programming languages are designed for person-to-computer communication
- Nevertheless algorithms are meant to be implemented on computers

Purpose of algorithms

- Algorithms are designed for person-to-person communication
- Programming languages are designed for person-to-computer communication
- Nevertheless algorithms are meant to be implemented on computers
- Algorithms sit halfway between
    - excruciating detail needed for person-to-computer communication
    - informal English used for person-to-person communication

Purpose of algorithms

- Algorithms are designed for person-to-person communication
- Programming languages are designed for person-to-computer communication
- Nevertheless algorithms are meant to be implemented on computers
- Algorithms sit halfway between
    - excruciating detail needed for person-to-computer communication
    - informal English used for person-to-person communication

We need to eliminate vagueness:

- make a note of . . .
- proceed through the list of records . . .
- find record of boss in list . . .

Essential features of algorithms

Data storage and manipulation

- make a note of number 0
- make note of name of boss
- add salary to noted number
- add up resulting numbers
- increase the counter

Control structures

- proceed through the employee list . . . . . .
- if salary of boss is less than . . . then . . .
- when the end of the list is finished . . . . . .

Pseudocode = a human-readable way to write algorithms
using exactly these features

Data storage: VARIABLES or little boxes

Examples:

- “noted number” is a variable
- “increase counter” – “counter” is a variable

Data storage: VARIABLES or little boxes

Examples:

- “noted number” is a variable
- “increase counter” – “counter” is a variable

In pseudo code

- VARIABLES are used to store data
- VARIABLES can be updated:
    - $\textsc{counter} \leftarrow 0$
- VARIABLES can be interrogated:
    - $\textsc{total} \leftarrow \textsc{salary} + \textsc{increment}$
    - $\textsc{counter} \leftarrow \textsc{counter} + 1$
    - $\textsc{salary} \leftarrow \textsc{salary} * 5$

Data storage: VARIABLES or little boxes

Examples:

- “noted number” is a variable
- “increase counter” – “counter” is a variable

In pseudo code

- VARIABLES are used to store data
- VARIABLES can be updated:
    - $\text{COUNTER} \leftarrow 0$
- VARIABLES can be interrogated:
    - $\text{TOTAL} \leftarrow \text{SALARY} + \text{INCREMENT}$
    - $\text{COUNTER} \leftarrow \text{COUNTER} + 1$
    - $\text{SALARY} \leftarrow \text{SALARY} * 5$

What operations can be performed with contents of variables?

Are variables really little boxes?

- If I put something in a box (SUM ← SALARY), do I still have hold of SALARY after?

Are variables really little boxes?

- If I put something in a box (SUM ← SALARY), do I still have hold of SALARY after?
- When I use SALARY is the box empty afterwards?

Are variables really little boxes?

- If I put something in a box ($\textsc{sum} \leftarrow \textsc{salary}$), do I still have hold of $\textsc{salary}$ after?
- When I use $\textsc{salary}$ is the box empty afterwards?
- What if I put two things in the box?

```
$\textsc{salary} \leftarrow 0$
$\textsc{salary} \leftarrow 1$
```

Are variables really little boxes?

- If I put something in a box ($\textsc{sum} \leftarrow \textsc{salary}$), do I still have hold of $\textsc{salary}$ after?
- When I use $\textsc{salary}$ is the box empty afterwards?
- What if I put two things in the box?

  $\textsc{salary} \leftarrow 0$

  $\textsc{salary} \leftarrow 1$

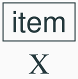
These are choices : a semantic description is needed

Data storage: arrays

- “proceed through the list of records”
- “search records for boss of current employee”

Data storage: arrays

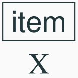
- “proceed through the list of records”
- “search records for boss of current employee”
- VARIABLE X:



Data storage: arrays

- “proceed through the list of records”
- “search records for boss of current employee”

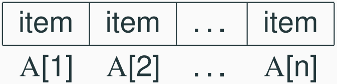
- VARIABLE X:



item

X

- ARRAY A[1...n]:



item	item	...	item
A[1]	A[2]	...	A[n]

Data storage: arrays

- “proceed through the list of records”
- “search records for boss of current employee”

- VARIABLE X:

item
X

- ARRAY A[1. . . n]:

item	item	. . .	item
A[1]	A[2]	. . .	A[n]

characteristics of arrays:

- Length of an array is always known:
  - A[1 . . . *n*] has *n* boxes
- Each box in an array is directly accessible, via index:
  - $\mathrm{A}[3] \leftarrow 27$
  - $\mathrm{A}[7] \leftarrow \mathrm{B}[2] + \mathrm{A}[1]$

Control structures

- Direct sequencing:

  “do A then do B then do C then do . . . ”

Control structures

- Direct sequencing:

  “do A then do B then do C then do . . . ”

- Conditional sequencing:

  “if something is true then do A” and
  “if something is true then do A otherwise do B ”

Control structures

- Direct sequencing:

  “do A then do B then do C then do . . . ”

- Conditional sequencing:

  “if something is true then do A” and

  “if something is true then do A otherwise do B ”

- Bounded iteration:

  “do something exactly *n* times”

  number of repetitions is known in advance

Control structures

- Direct sequencing:

  “do A then do B then do C then do . . . ”

- Conditional sequencing:

  “if something is true then do A” and
  “if something is true then do A otherwise do B ”

- Bounded iteration:

  “do something exactly *n* times”
  number of repetitions is known in advance

- Conditional iteration:

  “do something so long as some condition is true”
  number of repetitions depends on evaluation of “condition”

Control structures

- Direct sequencing:

  “do A then do B then do C then do . . . ”

- Conditional sequencing:

  “if something is true then do A” and

  “if something is true then do A otherwise do B ”

- Bounded iteration:

  “do something exactly *n* times”

  number of repetitions is known in advance

- Conditional iteration:

  “do something so long as some condition is true”

  number of repetitions depends on evaluation of “condition”

All available in all programming languages

Direct sequencing

General format:

```
do A
do B
do C
```

Direct sequencing

General format:

```
do A
do B
do C
```

Examples

```
$\textsc{curr} \leftarrow 1$
$\textsc{last} \leftarrow 10$
$\textsc{middle} \leftarrow$
$(\textsc{curr} + \textsc{last})$ div 2
```

Direct sequencing

General format:

```
do A
do B
do C
```

Examples

```
$\text{TEMP} \leftarrow \mathrm{A}[i]$
$\mathrm{A}[i] \leftarrow \mathrm{A}[j]$
$\mathrm{A}[j] \leftarrow \text{TEMP}$
```

Direct sequencing

General format:

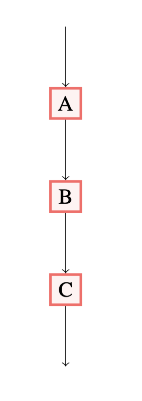
```
do A
do B
do C
```

Examples

```
// interchange $A[i]$ with $A[j]$
$\text{TEMP} \leftarrow A[i]$
$A[i] \leftarrow A[j]$
$A[j] \leftarrow \text{TEMP}$
```

Picturing direct sequencing

```
do A
do B
do C
```



Conditional sequencing

General formats

1.

```
if some condition is true then
    do something
```

2.

```
if some condition is true then
    do something
else
    do some other thing
```

Conditional sequencing

General formats

1.

```
if some condition is true then
  | do something
```

2.

```
if some condition is true then
  | do something
else
  | do some other thing
```

e.g.

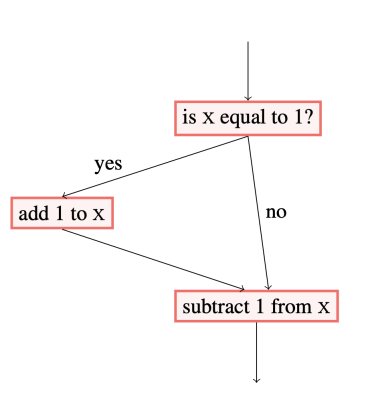
```
if sales have decreased
then
  | lower price by 10%
```

e.g.

```
if $price > limit$ then
  | pay x
else
  | pay y
```

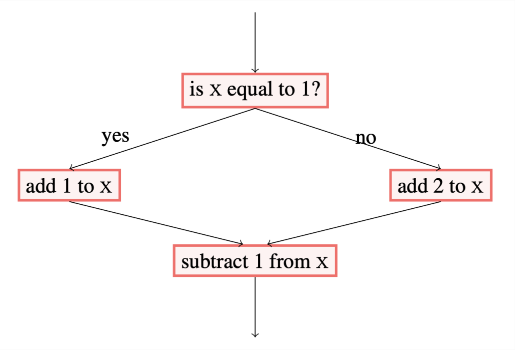
Picturing conditional sequencing

```
**if** $\mathrm{x} = 1$ **then**
  | $\mathrm{x} \leftarrow \mathrm{x} + 1$
$\mathrm{x} \leftarrow \mathrm{x} - 1$
```



Picturing conditional sequencing

```
**if** $x = 1$ **then**
  | $x \leftarrow x + 1$
**else**
  | $x \leftarrow x + 2$
$x \leftarrow x - 1$
```



Layout is important

```
if $item\ is\ taxable$ then
    if $price > limit$ then
        pay x
    else
        pay y
else
    pay z
```

Layout is important

```
if item is taxable then
    if $price > limit$ then
        pay x
    else
        pay y
else
    pay z
```

```
if item is taxable then
    if $price > limit$ then
        pay x
else
    pay y
```

Layout is important

```
if $condition$ then
    do A
    do B
else
    do C
do D
```

Layout is important

```
if condition then
    do A
    do B
else
    do C
do D
```

```
if condition then
    do A
    do B
else
    do C
    do D
```

Bounded iteration

Do something an exact number of times

General format:

```
**for** $i \leftarrow$ *start* **to** *finish* **do**
    *something*
```

Bounded iteration

Do something an exact number of times

General format:

```
for $i \leftarrow start$ to $finish$ do
    something
```

i: the *iterator*. Available to use within the *something*

start: value at which the iterator starts

finish: value at which the iterator ends

Bounded iteration: Example

Summing the first *n* positive numbers:

```
**Input**: positive number $n$
**Output**: sum of first $n$ positive numbers
$\textsc{sum} \leftarrow 0$
**for** $i \leftarrow 1$ **to** $n$ **do**
    $\textsc{sum} \leftarrow \textsc{sum} + i$
**return** $\textsc{sum}$
```

Bounded iteration: Example

Summing the first *n* positive numbers:

```
Input: positive number $n$
Output: sum of first $n$ positive numbers
$\textsc{sum} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $\textsc{sum} \leftarrow \textsc{sum} + i$
return $\textsc{sum}$
```

Explanation:

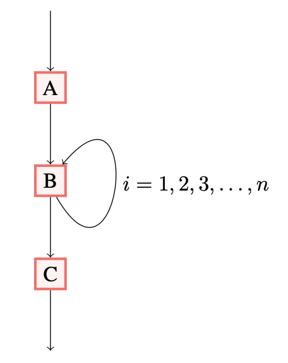
Input: description of expected input

Output: description of desired output

return *value*: an operation in pseudo-code

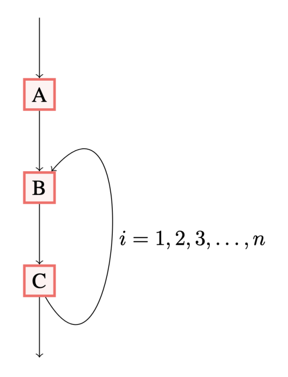
Picturing bounded iteration

```
A
for $i \leftarrow 1$ to $n$ do
    B
C
```



Picturing bounded iteration

```
A
for $i \leftarrow 1$ to $n$ do
    B
    C
```



Conditional iteration

Perform *something* repeatedly so long as some *condition* remains true

General format:

```
while condition do
    something
```

Conditional iteration

Perform *something* repeatedly so long as some *condition* remains true

General format:

```
while condition do
    something
```

Explanation:

- Executing *something* can affect the value of *condition*
- If *condition* is false *something* is not executed
- If *condition* is true *something* is executed and . . . evaluation repeats itself
- The value of *condition* may remain true forever

Conditional iteration

Perform *something* repeatedly so long as some *condition* remains true

General format:

```
while condition do
    something
```

Explanation:

- Executing *something* can affect the value of *condition*
- If *condition* is false *something* is not executed
- If *condition* is true *something* is executed and . . . evaluation repeats itself
- The value of *condition* may remain true forever

Not guaranteed to terminate

Conditional iteration: Example

- Summing numbers:

Input: positive number *n*

Output: sum of first *n* positive numbers

```
$\textsc{sum} \leftarrow 0$
$\textsc{iter} \leftarrow 1$
while $\textsc{iter} \leq n$ do
    $\textsc{sum} \leftarrow \textsc{sum} + \textsc{iter}$
    $\textsc{iter} \leftarrow \textsc{iter} + 1$
return $\textsc{sum}$
```

Conditional iteration: Example

- Summing numbers:

  Input: positive number *n*

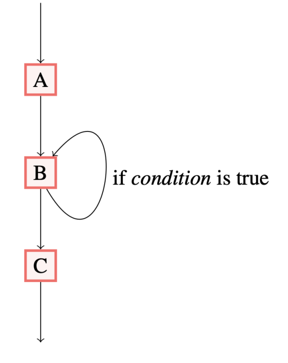
  Output: sum of first *n* positive numbers

  ```
  $\textsc{sum} \leftarrow 0$
  $\textsc{iter} \leftarrow 1$
  while $\textsc{iter} \leq n$ do
      $\textsc{sum} \leftarrow \textsc{sum} + \textsc{iter}$
      $\textsc{iter} \leftarrow \textsc{iter} + 1$
  return $\textsc{sum}$
  ```

- Note:
  - ITER needs to be explicitly managed
  - ITER is automatically furnished by for-loop construct

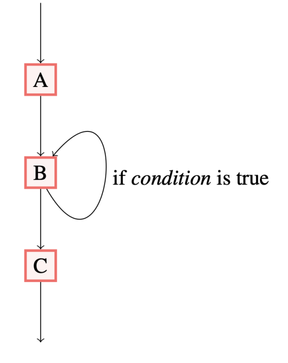
Picturing bounded iteration

```
A
while condition do
    B
C
```



Picturing bounded iteration

```
A
while condition do
    B
C
```



For more of these pictures, and some a recap of pseudocode, see the Overview of Pseudocode on Canvas

Layout is important

```
for $i \leftarrow 2$ to $k$ do
    A
    B
C
D
```

Layout is important

```
for $i \leftarrow 2$ to $k$ do
    A
    B
C
D
```

```
for $i \leftarrow 2$ to $k$ do
    A
    B
    C
D
```

Layout is important

```
if cond1 then
    while cond2 do
        A
        B
else
    C
```

Layout is important

```
if cond1 then
    while cond2 do
        A
        B
else
    C
```

```
if cond1 then
    while cond2 do
        A
    B
else
    C
```

Pseudocode

- Data storage: variables, arrays
- Control structures: direct sequencing, bounded iteration, conditional sequencing.
- **return** to give back a value at the end

Don’t forget: any algorithm must come with a valid specification!

How do you implement these in Python / Java?

Calculating salary bill

Legal inputs: any list of employee records; each record contains their salary

Required output: the total salary bill

Algorithm

(1) Make a note of number 0

(2) Proceed through the employee list, each time

- adding salary to noted number

(3) When end of list is reached

- output noted number

Name	Salary
Tom Jones	12000
Mary Clark	17000
Shaun Collins	16000
. . .	. . .
Lisa	23000

Pseudo-code for calculating salary bill

Input: an array $\mathrm{E}[1 \ldots n]$ of employee details

Output: total salary of all employees

Assumptions: array elements contain *salary* field

```
$\textsc{total} \leftarrow 0$
$\textsc{ptr} \leftarrow 1$
while $\textsc{ptr} \leq n$ do
    $\textsc{total} \leftarrow \textsc{total} + \mathit{salary}(\mathrm{E}[\textsc{ptr}])$
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{total}$
```

Pseudo-code for calculating salary bill

Input: an array $\mathrm{E}[1 \ldots n]$ of employee details

Output: total salary of all employees

Assumptions: array elements contain *salary* field

```
$\textsc{total} \leftarrow 0$
$\textsc{ptr} \leftarrow 1$
while $\textsc{ptr} \leq n$ do
    $\textsc{total} \leftarrow \textsc{total} + salary(\mathrm{E}[\textsc{ptr}])$
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{total}$
```

Somewhat easier with for-loop construct

Pseudo-code for calculating salary bill

Input: an array $\mathrm{E}[1 \ldots n]$ of employee details
Output: total salary of all employees
Assumptions: array elements contain *salary* field

```
$\textsc{total} \leftarrow 0$
$\textsc{ptr} \leftarrow 1$
while $\textsc{ptr} \leq n$ do
    $\textsc{total} \leftarrow \textsc{total} + salary(\mathrm{E}[\textsc{ptr}])$
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
return $\textsc{total}$
```

Somewhat easier with for-loop construct

Details of *salary* extraction ignored

Happy employees

Counting happy employees

Legal inputs: any list of employee records; each record contains their salary and name of boss

Required output: number of employees earning more than their boss

Algorithm ?

(1) Make a note of counter 0

(2) Proceed through the employee list, each time

- (a) Note name of boss, and salary of current employee
- (b) Find record of boss in list
- (c) If salary of boss is less than that of current employee, increase the counter

(3) When end of list is reached, output value of counter

Name	Salary	Boss
Tom	12000	James
Mary	17000	Cindy
Shaun	16000	Tom
…	…	…
Lisa	23000	Mary

Happy employees

Counting happy employees

Legal inputs: any list of employee records; each record contains their salary and name of boss

Required output: number of employees earning more than their boss

Algorithm ?

```
(1) Make a note of counter 0

(2) Proceed through the employee list, each time
    (a) Note name of boss, and salary of current employee
    (b) Find record of boss in list
    (c) If salary of boss is less than that of current
        employee, increase the counter

(3) When end of list is reached, output value of counter
```

Name	Salary	Boss
Tom	12000	James
Mary	17000	Cindy
Shaun	16000	Tom
…	…	…
Lisa	23000	Mary

Towards pseudo-code

Input: an array $\mathrm{E}[1 \ldots n]$ of employee details

Output: number of employees earning more than their boss

Assumptions: array elements contain *name*, *salary*, *boss* fields. Every employee has a *boss*

```
$\textsc{happy} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $\textsc{boss} \leftarrow boss(\mathrm{E}[i])$
    $\textsc{salary} \leftarrow salary(E[i])$
    find $\textsc{ptr}$ satisfying $\textsc{boss} = name(\mathrm{E}[\textsc{ptr}])$
    if $salary(\mathrm{E}[\textsc{ptr}]) < \textsc{salary}$ then
        $\textsc{happy} \leftarrow \textsc{happy} + 1$
return $\textsc{happy}$
```

Pseudo-code for counting happy employees

Input: an array $\mathrm{E}[1 \ldots n]$ of employee details

Output: number of employees earning more than their boss

Assumptions: array elements contain *name*, *salary*, *boss* fields. Every employee has a *boss*

```
$\textsc{happy} \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $\textsc{boss} \leftarrow boss(\mathrm{E}[i])$
    $\textsc{salary} \leftarrow salary(E[i])$
    $\textsc{ptr} \leftarrow 1$
    while $\textsc{boss} \neq name(\mathrm{E}[\textsc{ptr}])$ do
        $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
    if $salary(\mathrm{E}[\textsc{ptr}]) < \textsc{salary}$ then
        $\textsc{happy} \leftarrow \textsc{happy} + 1$
return $\textsc{happy}$
```

What about when an employee might not have a boss?

Summary

- A fixed language. Powerful enough to write all algorithms.
- Data storage: Variables and Arrays. Care with indexing.
- Difference between bounded and conditional iteration.
- Layout is very important!

To Do:

- Exercise sheet 1: check the solutions.
- Homework 1.
- Exercise sheet 2.

Ask any questions at the Helpdesk/Exercise sessions.