**Lecturer: John Williamson**

# Week 1

## Lecture 1

Weekly unit:

* 10-15 video before Friday
* 1 hour lecture
* 2 hour lab – **4 hours before the lab**
* 1 hour follow up lecture

Assessment:

* Quizzes each lecture 5%
* Written class exam: 15%
* Computer lab exam: 10%
* 2nd sem computer lab exam: 10%
* Degree exam in May: 60%

Python:

* Dynamic typing: values have types
* One line comment symbol - #

HW:

* Watch video till Friday
* Read lecture notes
* Prepare for first lab:

## Lecture 2: definitions (expr, statem, operators, operands)

Expression – sequence of code that returns a value

Statement – sequence of code that does smth

* Assignment st – right-hand side is an expression/variable

Dynamic typing: values have tags (type) and values

Behaviour changes depending on types of values in expression

Duck typing: “if it walks like a duck and quacks like a duck, it’s a duck”

Operand: values that operators operate on

* 1 operand: unary operator: *prefix*, e.g., -
* 2 operands: binary operator: *infix*

String comparison operators compare lexicographically (dict order)

Identifiers (names for variables):

* Start with underscore or letter
* Consist only of underscores, letters and numbers

Parallel assignment:

* a, b = 1, 2
* swap: a, b = b, a

input(…) string

String interpolation:

* “string bla {placeholder}”.format(placeholder = …)

# Week 2

## Lecture 3

Literal: way of writing a value directly in code

Hack-job code monkey: bad  
Professional software developer: good

Classes will have code criticism sections

### Criticism

Bad idea to feature bare constants (0.1, 0.999) without explanation, **use variables for constants**

“Number of [smth]”: n\_smth

**The constant name explains its purpose, not the value it contains**

Be precise and concise.

## Lecture 4: branching and looping

Branching: ability to do different things when given different things

* Branch: element of the program where the flow of exe splits into two possible paths. The path selected depends on **data**

Flow diagrams are used to show where and how branches appear.

**Python is whitespace-sensitive.**

*If* and *else* are **mutually exclusive** branches.

Sometimes it’s better to use another data structure (list, dict) instead of if, else

Conditional: expression that evaluates to True/False

Truthy and falsey:

* Everything is true, except:
  + False
  + None
  + Empty list
  + Empty string
  + Int 0
  + Floating point 0

*pass:* does nothing, but doesn’t cause syntax error

Iteration:

* Definite it: fixed number of times
  + Loop variable: changes as loop runs
  + Iterable: smth that provides values to iterate over
  + Start: inclusive
  + End: exclusive
  + Skip: how much to increment by
  + Nested loops
  + Range(): counted loops
  + Modifying loop variables modifies them for just one iteration
  + For: natural iterator
* Indefinite it:
  + Can be a way to write for loops (but not recommended)
  + Can count the number of times the loop executes (also true for definite it)
* Infinite it:
  + If condition is always true, executes forever
* Escape from escaping:
  + Break: breaks innermost loop for condition
  + Continue: skips over the rest of the body for one iteration
  + Do not use these as crutch for sloppy thinking

# Week 3

## Lecture 5:

Flag: Boolean value that’s used to determine a program’s behaviour

Fencepost error: off-by-one error. Frequently found in for loops

## Lecture 6: Functions and Recursion

Functions:

* Decomposition: breaking down into small, manageable chunks (which can be reused)
* Parametrisation: allows problems to be modified in a controlled way
  + Parameters: variables you receive inside a function
  + Argument: values you send to a function by calling it
  + Optional parameters: if they are assigned in the definition of the function
  + If arguments are named the same as parameters and assigned when called, they can be in a different order
* Isolation: isolate variables used in one part of the program from another
  + When variables are created inside a function, they exist in a **local scope** (only in the function)
  + Variables created outside a function is in a **global scope**
  + Shadowing: creating a variable in local scope with the same name as one in global scope
  + If you want to create a global variable locally: **global** prefix

Actions:

* Calling: transfers the flow of control

Functions are **first-class**: you can do anything with them

Functions without brackets are themselves, not their returned value.

A function isn’t bound to any objects, a method is.

### Recursion: function calling itself.

Base case: a condition which will stop the recursion.

Recursive call: function calling itself if the base case does not apply

# Week 4

## Lecture 7: Abstraction: Functions, parameters, arguments, and scope

## Lecture 8: Lists: []

Lists are heterogenous: one can contain multiple data types

Lists can be used as queues with pop(0) and append() (push)

Stack: LIFO, pop()

Queue: FIFO

Lists:

* Dynamically sized
* Dynamically typed (anything goes in)
* len()
* Slicing:
  + Indexing: [-2] : second to last element
  + [-2:] : second to last to last
  + [:-2] : first to second to last
* sum()
* Enumerate:
  + for i, elt, in enumerate(list\_name):
    - # i: index; elt: element
* Deleting:
  + remove(value) (deletes first occurrence)
  + pop(index) (by default: last, returns the value)
  + del list[index] (just deletes it)
* Membership test
  + Element **in** list
* index(element)
* count(element)
* sort() (modifies the list)
* sorted(list) (function that doesn’t change the list)
* Copying:
  + b = a[:] (slice of everything from a is copied)
* Equality:
  + == (same values)
  + is (whether they refer to same memory space)

Immutable: integers, strings, tuples  
Mutable: lists, dictionaries

# Week 5

## Lecture 9: Tuples: (), Do Not Repeat Yourself (DRY)

Tuples: like lists, but immutable

* tuple(), list() conversions

Functions can return “multiple values” by returning a tuple

Unpacking: assigning multiple values into multiple variables in one line

* first, last = [2, 4]
* Can use tuples on RHS

DRY:

* Create functions

## Lecture 10: Dictionaries: {}

Keys: values dictionaries are indexed by, must be unique

Values: anything, the thing that is mapped to

{key1: value1, key2: value2, …}

Uses of dictionaries:

* Lookup tables
* Memoization and caching
  + Memoization: remembering results that have been computed before
  + Web cache with URLs
* Mini-database (dictionaries in a list)
* Graphing
* Parameters in functions
* Set (bc of no repetition)
* Counters

Lists have O(n) time

If list is sorted, binary search takes O(log(n))

**Dictionaries have O(1) time**

Hashing: every key is translated into a number

Hash function: converts a value used as a key to an integer which can be used as an index into an array

Every key has to be:

* Unique
* Hashable
* Immutable

Keys cannot be:

* Lists
* Dictionaries
* Streams

Deleting:

* dictionary[key]
* del dictionary[key]
* dictionary.pop(key): returns and deletes
* dictionary.popitem(): deletes random key-value pair

dic.keys(): all keys only

dic.values(): all values only

for key, value in dic.items()

dic1.update(dic2): merges and changes duplicate keys (to those of dic2)

defaultdict(list): **RESEARCH**

Copying:

* dict.copy()
* Otherwise, just like for lists, new dict2 is a reference, not a copy to dict1
* copy() is shallow: changes just one level of values
* deepcopy() as an alternative

# Week 6

## Lecture 11: Hashing, splat

Coderbyte, leetcode: interview questions

Splat operator \*:

* Defining
  + Takes any number of arguments
* Calling
  + Unpacks a sequence into separate arguments

Double splat \*\*:

* Defining
  + Will collect all keyword arguments into a dictionary
* Calling
  + Unpacks keywords

Hashing:

* Hash function maps a value used as a key to an integer. Hash(key) -> integer index
* Collision: hash key same, value tries to be stored in a full spot
* Linear probing: if value isn’t in key, check next one, repeat

## Lecture 12: Strings, Files

### Strings

Can do “in” for substrings (and chars, ofc)

ASCII: 128 chars (7 bit ASCII codes)

* Subset of utf-8
* 0-31 are control chars
* 32: space
* 10: \n (newline
* 9: \t
* 13: \r (CR)

chr(): converts an integer into char (if possible)

.split(): separates string into a list (default delimiter: space

* .split(“\n”) = .splitlines()

“delimiter”.join(list): takes list, joins into a string with delimiter

Capitals:

* .capitalize(): capitalises first letter of first word
* .title(): capitalises first letter of all words
* .swapcase(): swaps uppercase and lowercase (WHY)

Justify:

* String.ljust()
* String.rjust()

Interpolation methods:

* :f floating
* :e floating, scientific notation
* :d integer
* :s string
* :x integer, hex

.find(): same as .index(), but gives -1 if not found

.replace(target, argument)

### Files

Why use files:

* Make information persistent
* To interchange data
* To organise data (filesystem)

Why not use files:

* Writing and access if slow (~1 million times slower)
* Made with old computers (sequential) in mind

File: a block of data managed by OS

Stream: connection to a file, represents a state (stream position)

Context managers:

* Files must be opened to read and write
* Files must be closed
* CMs establish a connection and guarantee it will be closed
  + with open(“file”) as f:
  + file mode possible to change
    - open(“file”, “w”) to write (deletes everything)
    - open(“file”, “a”) to write at end
    - open(“file”, “r”) read-only, by default
  + ALWAYS USE THIS SYNTAX

Reading and writing:

* read(): returns entire file as string
* readline(): reads one
* readlines(): reads all lines, puts into list
* print(…, file=f): prints into a file (newlines at the end)
* for line in f: (goes to each line)
* .strip(): removes spacing

# Week 7

## Lecture 13: Tape Recorder Demonstration, Data Types

Indirection: when a table is used to translate one result before looking it up in another table

## Lecture 14: Debugging

Instrumenting code: measuring behaviour

* Explicit instrumentation: adding logging code, adding assertions
* External instrumentation: instrumentation will be injected into code by external
  + Debugger
  + Profiler

Testing:

* Proving code is correct (almost never happens)
* Be confident if everything is written with standards
* Actual testing
  + Automated testing
    - Live-testing: testing whether values are in expected forms. Documents expectations
    - Regression testing:
    - Test-driven dev:
* Hoping

Testing methods:

* Assert: “be right or die now”
  + Guard: assert before crucial part
  + Identify:
    - assert <statement>, “This error string will appear”
  + Testing with known good references (import math vs self-made functions)
  + Unit testing: testing parts of code (e.g., functions)
  + Fuzzing: testing with automatically generated values

Test conditions:

* Edge cases: values that are either at the extremes of what is likely, or values which don’t make sense in context

Debugging:

* Replicate the bug
* Read the code and understand it
* Run and see if everything works
* Refine: identify which part is causing the problem
* Ruminate it over
* Rubber duck: explain thinking out loud
* Rest
* Repair
* Re-evaluate: test

Debugging strategies:

* Worst: debug-by-guessing
* Slightly better: debug-by-bisection (the binary search of debugging)
  + Have to constantly disable parts of code
* Logging
  + Recording values
  + Usually done with peppering code with print() statements
  + Purposes:
    - to know which parts of a program are being executed; and
    - to know what values are bound to what variables during execution.
  + Tracing:
    - Branches
    - Function calls
    - Function returns
  + Principles
    - Arguments for functions
    - Function return
    - Computation results
    - Branches
  + What to log
    - Reachability
    - Values
    - Status (did a file open?
* Using a debugger: a special tool which can interrupt, inspect and modify running code:
  + Breakpoints
  + Commands
    - s: execute line and enter function call
    - n: execute line (skipping function calls) and go to next
    - c: go to next breakpoint
    - r: continue until end of current function
    - q (quit): quits debugger

### Optimisation

“Premature optimisation is the root of all evil”:

* Most inexperienced optimisers will make mistakes

Rules

* Never write optimised code first
* The biggest gains are always algorithmic -- this can require fundamental rethinking and lead to huge improvements.
* After you have correct code, and you are sure you have the right algorithmic approach, you can tune things to improve performance.

Optimisation strategies:

* Pull operations out of innermost loops
* Precompute values
  + Trade-off between runtime and space
* Using the right algorithms and data structures

# Week 8

## Lecture 15: Time Complexity

Big O notation

Leading order term: the term that will be the most relevant as the problem gets larger (the one that grows the fastest)

Underscores can be interspersed in integers freely

## Lecture 16: Arrays

Arrays are efficient:

* Compact (storing data in a memory efficient way)
* Computationally efficient (GPU-efficient)

1D: vector  
2D: shape  
3+D: *n*-rank tensor

Ndarrays:

* Fixed, predefined size
* Fixed, predefined type
* Shape: dimensions
* Type of elements

Creating arrays:

* Give existing list [of lists of lists …]
* np.full(shape, value)
* np.arange(start, (end, (step)))
  + can increment by fractional values

Indexing:

* x[row, col] if 2D
* x[el] if 1D

Joining arrays:

* Stack: two 1D arrays to make 2D
* Concatenate: two 1D to make a longer 1D

# Week 9

## Lecture 17: Array Reductions, Data Type Strengths

np.min, np.max, np.mean, np.std, np.sum

Reduction happens to all elements by default

Axis: specific dimensions

* axis = 0: reduced along the rows

Map (every element), filter, reduce (operations in between operators: sum, mean, etc)

Use numerical arrays:

* when all elements have to be changed simultaneously

Why use ndarrays:

* code simpler, faster, takes much less memory

Don’t use:

* when data is not numerical

Data type priorities:

* List: sequence
  + Queues, stacks
* Dictionary: associations, fast lookup (for membership, too)
  + Sets, counters, “databases”
* Array:
  + Tables, images, sounds
* String: interchange data

Code Criticism:

* Abstractions express things precisely and concisely

## Lecture 18: Comprehensions, Closures, and Lambda

Functional:

* Spreadsheets (no hidden data)
* Map, filter, reduce
* Recursion

Functional programming: without side effects and mutability

* Side effect: aliasing – unintentionally affecting other parts of code when changing one part

Pure function: return value depends **only** on the value passed in.

Comprehensions:

* Mapping: applying a function to each element
* Filtering: removing unneeded elements (if statement)

Can’t be done in a comprehension:

* indefinite iterations
* break, continue, return
* assignment to variables
* non-expression statements (*del*)
* always one return value

Ternary *if*:

* x if condition else y

First-class functions can:

* be put into variables
* stored in compound data structures
* passed to other functions
* be returned from other functions

Closures:

* Functions with inner functions to define other functions with similar properties

Non-local variables: belong to outer function

Higher-order function: a function that operates on functions

* map
* filter
* reduce: reduce(fn, item) – applies fn to pairs from pairs
* zip: return pairs of elements from sequences with same index

Lambda: anonymous functions

* Deferred execution:
  + Postpone execution by evaluating only when calling function

# Week 10

## Lecture 19: Recursive Solutions

numpy:

* .all() – returns True if all elements in array are True(thy)

The only way to achieve repetition in pure functional programming is with recursion.

Tree – a data structure with branches

* Tree traversal is easier with recursion

Recursion strategies:

* Default arguments
* Wrapper (outer) functions

Python accepts multi-line expressions if and only if there is some type of brackets around it.

Book recommendations: Code by Charles Petzgold, An Eternal Golden Braid, The Code Book by Simon Singh

## Lecture 20: Revision

List concatenation: +

Dict concatenation: dict1.update(dict2)

# Week 11

## Lecture 21: Revision+, Class Exam

Advice for exam:

* Read instructions
* Write in black pen
* Do easier questions, then tackle harder questions
* Read the whole paper before writing anything
* Check marks for tasks

Task 4: Tables for function calls

call

parameter argument

return value

sub(13, 8)  
x 13  
y 8  
5

operator(sub, 1, 1)  
fn sub  
x 1  
y 1  
None

double(double(2))  
x 2  
x 2  
2

switch(double(2), sub(8))  
x 2  
x 8  
y 0  
x 2  
y 8  
(8, 2)

## Lecture 22: Revision++

Question: strip() in first question?

def test\_eq(fn\_a, fn\_b):

def inner(seq):

res = True

for I in seq:

if fn\_a(i) != fn\_b(i):

res = False

break

return res

return inner

# Term 2

**Lecturer: Mohamed Khamis**

# Week 12

## Lecture 23: Intro, Revision

**Lecturer: Mohamed Khamis**

* [Mohamed.khamis@glasgow.ac.uk](mailto:Mohamed.khamis@glasgow.ac.uk)
* Room 204 Sir Alwyn Williams Building

## Lecture 24: More Revision, Compound Datatypes

Simple datatypes: int, float, bool

Compound: strings, lists, dictionaries, tuple

* Used as a single element or by parts
  + Iterable

Higher order functions:

* map(): applies a function to each element in an iterable
* filter(): tests each element in an iterable against a condition, evaluates to True if all elements are True
* reduce(): Applies a function of two parameters consecutively to an iterable
  + Only in “functools” library

List comprehensions are a bit faster than lambda functions + filter(),map()

Sets

* Collection of **distinct** elements
* Order doesn’t matter
* Mutable
* mySet = {1,2,3}
* len(set): number of distinct values
* {1,2,3} == {1,3,3,2,1}
  + True
* Adding: set.add(el)
* Removing:
  + set.remove(el): error if not found
  + set.discard(el): no error if not found
  + set.pop(): removes and returns a random element from the set
* Membership test: el in mySet
* Operations
  + A|B: union (A.union(B))
  + A&B: intersection (A.intersection(B))
  + A – B: removes elements from A that are in B (A.difference(B))
  + A –= B: changes A to the difference (A.difference\_update(B))
  + A |= B: changes A to union (A.update(B))

def split(\*args):

return [x[0] for x in args], [x[1] for x in args]

# Week 13

## Lecture 25: More on Sets, Sparse Data Structures

Sets:

* no indexing
* iterable

Dictionaries are good for sparse data structure representation.

## Lecture 26: Sorting

Measuring efficiency:

* Running time: not optimal, dependent on machine speed
* Big O

Bubble sort:

* O(n2)
* Algorithm
  + Compare first two items
  + If not in order, swap
  + Move to next pair
  + Repeat until end
  + Repeat until done

Selection sort

* O(
* Algorithm:
  + Find the smallest item
  + Swap it with the first item
  + Repeat

# Week 14

## Lecture 27: More Sorting Algorithms

Insertion sort:

* Algorithm
  + Go once through the list
  + Check next element and compare it to previous

Invariant: a statement that remains unchanged as the algorithm runs

* Bubble sort: the last *I* list elements are sorted
* Selection sort: the first *I*

## Lecture 28: Space Complexity, Merge Sort

Space complexity:

* Bubble sort: O(1) (temp value for swapping is reused)
* Selection sort: O(1) (technically O(2)) (minimum and temp)
* Insertion sort: O(1)

Merge sort (1945):

* Algorithm
  + Merging two sorted lists (merge()):
    - Compare first two values
    - Insert smallest
    - Compare next item in list with smallest first with first of other list
    - Repeat until both lists exhausted
  + Split into two halves
  + Sort each half by repeating merge sort
  + Merge halves
* Big O: O(nlog2n)
* Space complexity: O(n)

Quick sort:

* Algorithm:
  + Pick one el (pivot)
    - First/last/middle/median of all 3/randomly
  + Move all elements smaller than pivot to the left
  + Move all elements larger than pivot to the right
  + Looking for 2 items
    - Item from left of list larger than pivot
    - Item from right of list smaller than pivot
    - Swap
  + When pivot is in the right place, recursively repeat for each half around the pivot
* Time complexity:
  + Worst case: O(n2)
  + Best case: O(nlogn)

# Week 15

## Lecture 29: Searching

Unstructured list: O(n)

Quantum computers: O(sqrt(n)) with Grover’s algorithm

## Lecture 30: Searching (Actually)

Structured list:

* Binary search
  + Halve list, compare, repeat

# Week 16

## Lecture 31: Files

Stress testing:

* Testing sys against random events, chaos

Recap on files:

* Commands
  + r: read
  + w: (over)write
  + a: append
  + x: creates a file
  + t: opens file as text
  + b: opens file as binary
  + Commands can be stacked: “wt” (overwriting to text)
* Reading
  + read(): returns entire content as string
  + read(int): returns next int chars
  + readline(): returns next line as string
  + readlines(): returns lines as a list of strings
* Writing
  + write(str): writes str to line and doesn’t put linebreak
  + writelines(list): writes list of strings as lines of strings
  + Special characters: \t (tab, \ (escape for quotation marks), \n (newline)

## Lecture 32: Error Handling

Types of errors:

* Parsing: not understandable code
* Run-time (Exceptions): illegal operations
* Semantic: result not the intended result

Handling exceptions:

* try:
* except [optional exception]:
* Multiple except statements with different exceptions can be used (and a general except at the end)
* finally:
  + Executes regardless
* else:
  + Executes if try is successful

Ways of error handling:

* Easier to Ask for Forgiveness than Permission (EAFP):
  + Performance penalties when raising exceptions
  + Encouraged in Python
* Look Before You Leap (LBYL)
  + Checking conditions before calling them

# Week 17

## Lecture 33: Class Inheritance, Raising Exceptions

class CarError(Exception):

def \_\_init\_\_(self, car, msg=None): # the constructor

raise MyError()

## Lecture 34: Graphical User Interface (GUI), tkinter

Widget: GUI element (button, menu, label, etc)

The main loop is provided by the module

The functions are responses to events (clicks, hovers, etc)

Tkinter:

* import tkinter
* top = tkinter.Tk()
  + Create the top-level (root) window
* Elements:
  + quitButton = tkinter.Button(top, text=’Quit’, command=top.destroy)
  + msgLabel = tkinter.Label(top, text=’Hi’)
  + showBtn = tkinter.Button(top, text=’Show’, command=display)
  + Arguments:
    - width
    - row
    - column
  + Input:
    - textVar = tkinter.StringVar(“”)
    - textEntry = tkinter.Entry(top, textvariable=textVar, width=12)
    - textVar.get() # returns the string
  + Radio button:
    - choice = tkinter.IntVar(0)
    - hiBtn = tkinter.Radiobutton(top, text, variable=choice, value=1, command)
    - byeBtn = tkinter.Radiobutton(top, text, variable=choice, value=2, command)
* quitButton.grid()
  + Adds it to the grid
* tkinter.mainloop()
  + Tracks mouse
* Defining events:
  + define function
  + use command=func

# Week 18

## Lecture 35: Random Dict Exercises

dates: {“day”: 9, “month”: 4, “year”: 2008}

def before(date1, date2):

return date1[year] < date2[year] or   
 ( date1[year] == date2[year] and ( date1[month] < date2[month] or   
 ( date1[month] == date2[month] and date1[day] < date2[day] )))

def age(today, bday):

res = today[year] – bday[year] – 1  
 if today[month] > bday[month] or today[month] == bday[month] and today[day] >= bday[day]:  
 res += 1  
 return res

def dateFromString(s):

data = s.split(‘/’)  
 date = {}  
 date[‘day’] = data[0]  
 date[“month”] = data[1]  
 date[‘year’] = data[2]  
 return date

def printDate(d):

return ‘/’.join(list(d.items()))

## Lecture 36: GUI+, MVC

top.title(“Title”)  
top.geometry(“[pixels]x[[pixels]”)

Configure a label: label.configure(…)

MVC: Model, View, Controller

* Software design pattern
  + Model (computation)
  + View (display)
  + Controller (Event handler)

# Week 19

## Lecture 37: Sudoku Program

…

## Lecture 38: Object Oriented Programming

Idea of OOP: real life objects represented as instances of classes (those instances called objects)

* The objects have attributes and methods

Procedural vs OOP:

* Procedural
  + Data and functions have separate identities
* OOP:
  + A single object has both data (state) and functions that operate on the data (methods)

Object:

* a combination of data/state and methods/behaviour
* correspond to real life objects (cars, people, etc)
* constructor:
  + class Person:  
     def \_\_init\_\_(self, name, age):  
     self.name = name  
     self.age = age
  + x = Person(‘Joe’, 42)
* Getters and setters also possible (accessors, mutators)

Inheritance:

* A (child/sub) class inherits the methods and properties of another (parent/super) class, changing the specifics
  + Inheriting the constructor:  
    super().\_\_init\_\_(…)
* Notation
  + class Student(Person):
* Overriding: changing a specific method in a child class

Python vs. Java:

* Python’s Interpreter - runs code until meets an error
* Java’s Compiler - checks for errors in entire code
* Python dynamically typed
* Java statically typed

# Week 20

## Lecture 39: Sudoku+

Write down possibilities for each cell, eliminate possibilities by going through the program.

Mutual recursion: a function calling another function, which in turn calls the original function.

## Lecture 40: Revision of Sem 2

Assessed exercise:

* on Moodle
* due 25th of March

Complexity:

* Constant: O(1)
* Linear: O(n)
* Quadratic: O(n2)
* Exponential: O(2n)
* Factorial: O(n!)

Invariant: an unchanging statement

Sorting:

* Bubble
  + Behaviour: swaps by comparing pairs and then next pairs:
  + Complexity: O(n2)
  + Invariant: the last *i* items are sorted
* Selection
  + Behaviour: finds next smallest in whole list, swaps
  + Complexity: O(n2)
  + Invariant: the first *i* items are sorted
* Insertion
  + Behaviour: insert next value into sorted sublist
  + Complexity: O(n2)
  + Invariant: the first *i* items are sorted
* Merge
  + Behaviour: recursive merging until swapping two elements
  + Complexity: O(nlog(n))