BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 12 – Recap

Camelia Chira

Course content

Programming in the large

1. Software development process *(Lectures 1, 2)*
2. Procedural programming
3. Modular programming
4. Abstract data types
5. Software development principles
6. Testing and debugging

Programming in the small

7. Recursion
8. Complexity of algorithms
9. Search and sorting algorithms
10. Problem solving methods

1 Software development process

Programming in the large

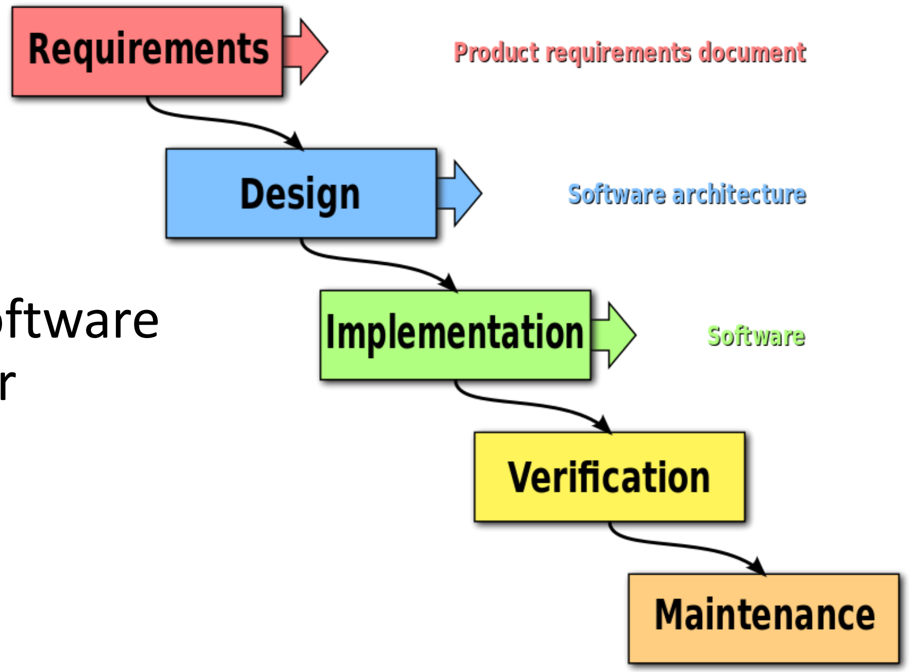
- Programming
  - Python programs
  - Data types
  - List operations
  - Assignments
  - Statements
  - Control flow
- Software development process
- Feature-driven development

1 Software development process

Programming in the large

- Roles in software engineering
  - Programmer / software developer
  - Client
  - User
- Software development process:
  - Includes creation, launch and maintainence of software
  - Indicates the steps to be followed and their order
- Steps in solving a problem:
  - Problem definition
  - Requirements
  - Use case scenario
  - Identify the features and separate them on iterations
  - Identify the activities or tasks (for each feature) and describe them

The waterfall model



1

Feature-driven development

Programming in the large

- Build a feature list from problem statement
- Plan iterations
- For each iteration
  - Model planned features
  - Implement and test the features
  - Obs:
    - At the beginning of each iteration: analyze each feature – determine the activities (tasks) required – schedule the tasks – implementat and test each independently.
    - *An iteration will result in a working program for the client (will interact with the user, perform some computation, show results)*

Course content

Programming in the large

Programming in the small

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming
4. Abstract data types
5. Software development principles
6. Testing and debugging
7. Recursion
8. Complexity of algorithms
9. Search and sorting algorithms
10. Problem solving methods

2

Procedural programming

Programming in the large

- Programming paradigms
- Procedural programming
- Function definition
- Variable scope
- Test-driven development

2 Procedural programming

Programming in the large

- Programming paradigms
  - Imperative programming
    - Computations described through statements that modify the state of a program (control flow – sequence of statements exexuted by the computer)
  - Examples: **Procedural programming,** Object oriented programming
- **Procedural programming** – each program is formed by several procedures (subroutines or functions)
- Function
  - A block of statements that can be reused
  - Are run in a program only when they are called

2 Functions in Python

Programming in the large

Function characteristics:

Has a **name**

Has a list of **parameters**

Can **return** a value

Has a body (a block of statements)

```
def get_max(a, b):
    """
    Compute the maximum of 2 numbers a, b - numbers
    Return a number - the maximum of two integers.
    Raise TypeError if parameters are not integers.
    """
    if a>b:
        return a
    return b
```

```
get_max(2, 3)
```

Has a specification (**docstring**) formed of:

- A description
- Type and description of parameters
- Conditions imposed on input parameters (pre-conditions)
- Type and description of return value
- Conditions imposed on output values (post-conditions)
- Exceptions that can occur during its execution

Variable scope

- Types of variables
  - **Local** – a name (of variable) defined in a block
  - **Global** – a name defined in a module
  - Free – a name used in a block but defined somewhere else

```
g1 = 1 # g1 - global variable (also local, a module being a block)

def fun1(a): # a is a formal parameter
    b = a + g1 # b - local variable, g1 - free variable
    if b > 0: # a, b, and g1 are visible in all blocks of this function
        c = b - g1 # b is visible here, also g1
        b = c # c is a local variable defined in this block
    return b # c is not visible here

def fun2():
    global g1
    d = g1 # g1 - global variable
    g1 = 2 # g1 must be declared global, before
    return d + g1 # any references to g1 in this function

print(fun1(1))
print(fun2())
```

2

Variable scope

Programming in the large

- Where is a variable visible?
  - Rules to determine the scope of a name (variable or function)
    - A name is visible only inside the block where it is defined
    - The formal parameters of a function belong to the body of the function (are visible only inside the function)
    - Names defined outside of a function (at module level) belong to the module scope
    - When a name is used in a block, its visibility is determined using the nearest scope (that contains that name)

```python
a = 100
def f():
    a = 300
    print(a) # 300

f()
print(a) # 100
```

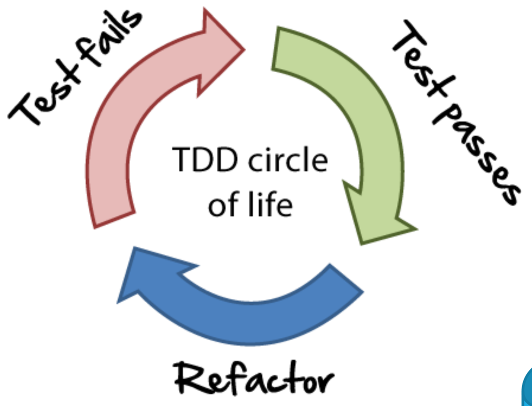
```python
a = 100
def f():
    global a
    a = 300
    print(a) # 300

f()
print(a) # 300
```

2

Test-Driven Development (TDD)

Programming in the large

- Implies creation of tests (that clarify the requirements) before writing the code of the function
- Steps to create a new function:
  1. Add a new test / several tests
  2. Execute tests and verify that at least one of them failed
  3. Write the body of the function
  4. Run all tests
  5. Refactor the code
     - Extraction method
     - Substitution of an algorithm
     - Replacing a temporary expression with a function



- Think first (what each part of the program has to do), write code after
- Analyse boundary behaviour, how to handle invalid parameters before writing any code

Course content

Programming in the large

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types
5. Software development principles
6. Testing and debugging

Programming in the small

7. Recursion
8. Complexity of algorithms
9. Search and sorting algorithms
10. Problem solving methods

3

Modular programming

Programming in the large

- Modules
- Packages
- import statement

- How to organize an application
- Layered architecture
- Exceptions
  - raise statement
  - try..except(..finally) statement

3

Modular programming

Programming in the large

- Module
  - Structural unit (that can communicate with other units), changeable
  - Collections of functions and variables that implement a well-defined feature
- Based on decomposing the problem in subproblems considering:
  - Separating concepts
  - Layered architectures
  - Maintenance and reuse of code
  - Cohesion of elements in a module
  - Link between modules

3

Python module

Programming in the large

- Module: a file that contains Python statements and definitions
  - Variables – global names, visible at the level of the module
  - Function definitions – available in that module and in other modules that import the current module
  - Other statements - initialization
- A module has:
  - Name (**__name__**)
  - Docstring (**__doc__**)
  - A symbol table with all the names introduced by the module – dir(moduleName)
- Python module must be imported in order to use it
  - The `import` statement
    - `import [path.]moduleName`
    - `from moduleName import itemName`

3 Organize the code of an application

Programming in the large

- Layered architectures

  - Decomposing by features – 2 perspectives:
    - Functional perspective – identifying different features specified by the problem
    - Technical perspective – introducing technical features (such as user interaction, file management, databases, networks , etc)

  - Recommended solution:
    - Decompose a complex application on layers
    - Concentrate the code related to the domain of the problem in a single layer and isolate it
    - Ensure cohesive layers

3

Layered architecture

Programming in the large

- How to organize the code
  - Create modules for
    - User interface
      - Functions dealing with user interaction
      - Contains read operations and display methods
      - The only module used to read and output data
    - Domain of the application
      - Functions dealing with the problem domain
    - Infrastructure
      - Useful functions that are highly to be reused (e.g. logging, network I/O)
    - Application coordinator
      - Initializes and starts the application

Course content

Programming in the large

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types *(Lectures 5, 6)*
5. Software development principles
6. Testing and debugging

Programming in the small

7. Recursion
8. Complexity of algorithms
9. Search and sorting algorithms
10. Problem solving methods

5

Abstract Data Types

Programming in the large

- Abstract Data Type
- Classes
- Data abstraction
- Encapsulation
- Information hiding
- Class attributes vs. instance attributes
- Static methods
- UML diagrams

5 Abstract Data Types (ADT)

Programming in the large

- Abstract Data Type
  - Export a name (a data type)
  - Define a domain of values for the data
  - Define an interface (the operation possible with the new data type)
  - Restrict access to the components of the new type
  - Hide the implementation of the new type
- Create the class vs. using an instance of the class
- `class` keyword

```
class Flower:
    '''
    a flower is a structure of two elements:
name (a string) and price (an integer)
    '''
    def __init__(self, n, p = 0):
        '''
            creates a new instance of Flower
        '''
        self.name = n
        self.price = p
        self.size = None

myFlower = Flower("rose", 5)
```

5

Classes

Programming in the large

- Creating a new class:
  - Creates a new *type* of an object
  - Allows *instances* of that type
- A class instance can have:
  - *Attributes* (to maintain its state)
  - *Methods* (to modify its state)
- Class name is the type e.g. `class Flower:`
- Instance is one specific object e.g. `f1 = Flower("rose", 5)`
- A class introduces a new namespace

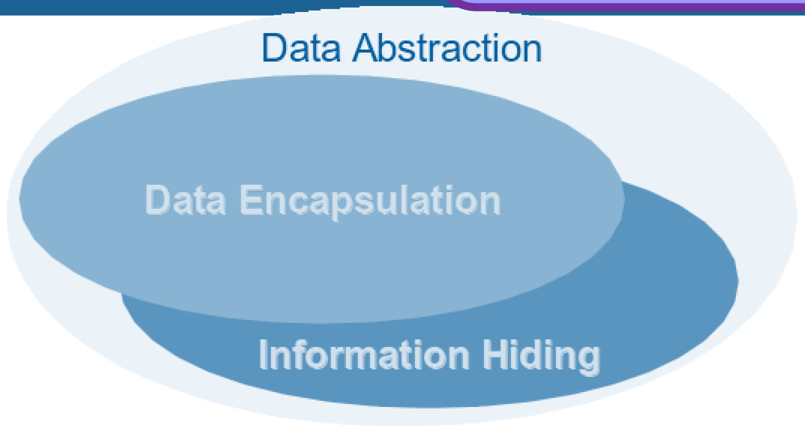
5 ADT Recommendations

Programming in the large

- Create getter and setter methods to access the data attributes
- Hide the implementation details
  - The class is an abstraction (a black box)
  - The interface of the class stays the same while internal changes can occur
  - Client code should work without any changes even when internal changes occur in the class
- Document each class
  - Short description
  - What objects can be created (based on the data attributes)
  - Restrictions that apply to data
- Create classes using ***test-driven development***
  - Create test functions for
    - Creating an instance of the class
    - Each method of the class

6 Data Abstraction

Programming in the large



- Encapsulation
  - *bundling of data with the methods that operate on that data*
  - Getter/setter methods
- Information hiding
  - *the principle that some internal information or data is "hidden" so that it can not be changed by accident*
  - The internal representation of an object
    - Needs to be hidden outside the object's denition
    - Protect object integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state
- **Data Abstraction = Data Encapsulation + Data Hiding**

6 Information hiding

Programming in the large

- Data hiding in Python
  - public and private members
  - based upon convention
- Attribute types
  - Private attributes __name
  - Protected (restricted) attributes _name
  - Public attributes name
- *Class* attributes vs *instance* attributes

```
class Student:

    __studentCount = 0

    def __init__(self, name=""):
        self.__name = name

        Student.__studentCount += 1

    def setName(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    @staticmethod
    def getStudentCount():
        return Student.__studentCount
```

6

UML Class Diagram

Programming in the large

Flower
+name: String +price: Integer
+__init__() +getName(): String +setName(String) +getPrice(): Integer +setPrice(Integer) +compare(Flower): Boolean

- Visibility
  - + -> public
  - - -> private
  - # -> protected

```
class Flower:
    def __init__(self):
        self.name = ""
        self.price = ""

    def getName(self):
        return self.name

    def setName(self, n):
        self.name = n

    def getPrice(self):
        return self.price

    def setPrice(self, p):
        self.__price = p

    def compare(self, other):
        if ((self.name == other.name) and
            (self.price == other.price)):
            return True
        else:
            return False
```

Course content

Programming in the large

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types *(Lectures 5, 6)*
5. Software development principles *(Lecture 4, 7)*
6. Testing and debugging

Programming in the small

7. Recursion
8. Complexity of algorithms
9. Search and sorting algorithms
10. Problem solving methods

5

Software development principles

Programming in the large

- Design principles
- Layered architecture
- GRASP

7

Software development principles

Programming in the large

- Good software design:
  - ✓ Code easy to understand
  - ✓ Easy to test
  - ✓ Easy to maintain
  - ✓ Easy to develop and modify (e.g. add features)
- Key design principles
  - Single Responsibility Principle
  - Separation of Concerns Principle
  - Reuse Principle
  - Cohesion and Coupling Principle

7

Design Principles

Programming in the large

- Single Responsibility Principle
  - Responsibility: of a function = to compute something, of a module = responsibilities of all functions in the module
  - The principle of a single responsibility: a function / module should have one responsibility
- Reuse Principle
  - Using modules improves the maintenance of an application
  - Using modules facilitates reuse of elements defined in the application
  - Managing the dependencies increases reuse
    - Dependencies between functions
      - A function invokes (calls) another function / other functions
    - Dependencies between modules
      - The functions from a module invoke functions from other modules

Design Principles

- Cohesion and Coupling Principle
  - The cohesion degree:
    - the degree to which a module has a single, well-focused purpose
    - High cohesion – the elements of the module are highly dependent on each other
    - Low cohesion – the elements relate more to other activities (and not to each other)
  - The coupling degree:
    - how dependent is a module on other modules
    - High coupling – modules that are highly dependent on each other
    - Low coupling – independent modules

7

Organize the code of an application

Programming in the large

- Organizing the application on layers should consider:
  - Each layer communicates with the previous layer
  - Each layer has a well-defined interface that is used by the superior layer (implementation details are hidden)

  - Low coupling between modules
    - Modules do not need to know details about other modules – futures changes are easier to make
  - High cohesion of each module
    - The elements of a module should be highly related

7

GRASP

Programming in the large

- General Responsibility Assignment Software Patterns (GRASP)
  - Guidelines for assigning responsibility to classes and objects

  - High Cohesion
  - Low Coupling
  - Information Expert
  - Creator
  - Pure Fabrication
  - Controller

Layered architecture:

- **High cohesion**

  To increase cohesion: break programs into classes and subsystems

  Low cohesion means that an element has too many unrelated responsibilities => problems: hard to understand, hard to reuse, hard to maintain

- **Low coupling**

  Low dependency between classes

  Low impact in a class of changes in other classes

  High reuse potential

7

GRASP

Programming in the large

- General Responsibility Assignment Software Patterns (GRASP)
  - Guidelines for assigning responsibility to classes and objects

  - High Cohesion
  - Low Coupling
  - Information Expert
  - Creator
  - Pure Fabrication
  - Controller

- Assign a responsibility to the class that has the information necessary to fulfill the responsibility

7 GRASP

Programming in the large

- General Responsibility Assignment Software Patterns (GRASP)
  - Guidelines for assigning responsibility to classes and objects

  - High Cohesion
  - Low Coupling
  - Information Expert
  - Creator
  - Pure Fabrication
  - Controller

Which class is responsible for creating objects?
Class B should be responsible for creating instances of class A if:

- Instances of B contain instances of A
- Instances of B closely use instances of A
- Instances of B have the initializing information for instances of A and can use it for creation

7

GRASP

Programming in the large

- General Responsibility Assignment Software Patterns (GRASP)
  - Guidelines for assigning responsibility to classes and objects

  - High Cohesion
  - Low Coupling
  - Information Expert
  - Creator
  - Pure Fabrication
  - Controller

A class added to an application in order to achieve low coupling, high cohesion and reuse

Repository

- Represents all objects of a certain type as a conceptual set
- Objects can be added, updated, removed and retrieved from the repository (persistent storage)

7 GRASP

Programming in the large

- General Responsibility Assignment Software Patterns (GRASP)
  - Guidelines for assigning responsibility to classes and objects

  - High Cohesion
  - Low Coupling
  - Information Expert
  - Creator
  - Pure Fabrication
  - Controller

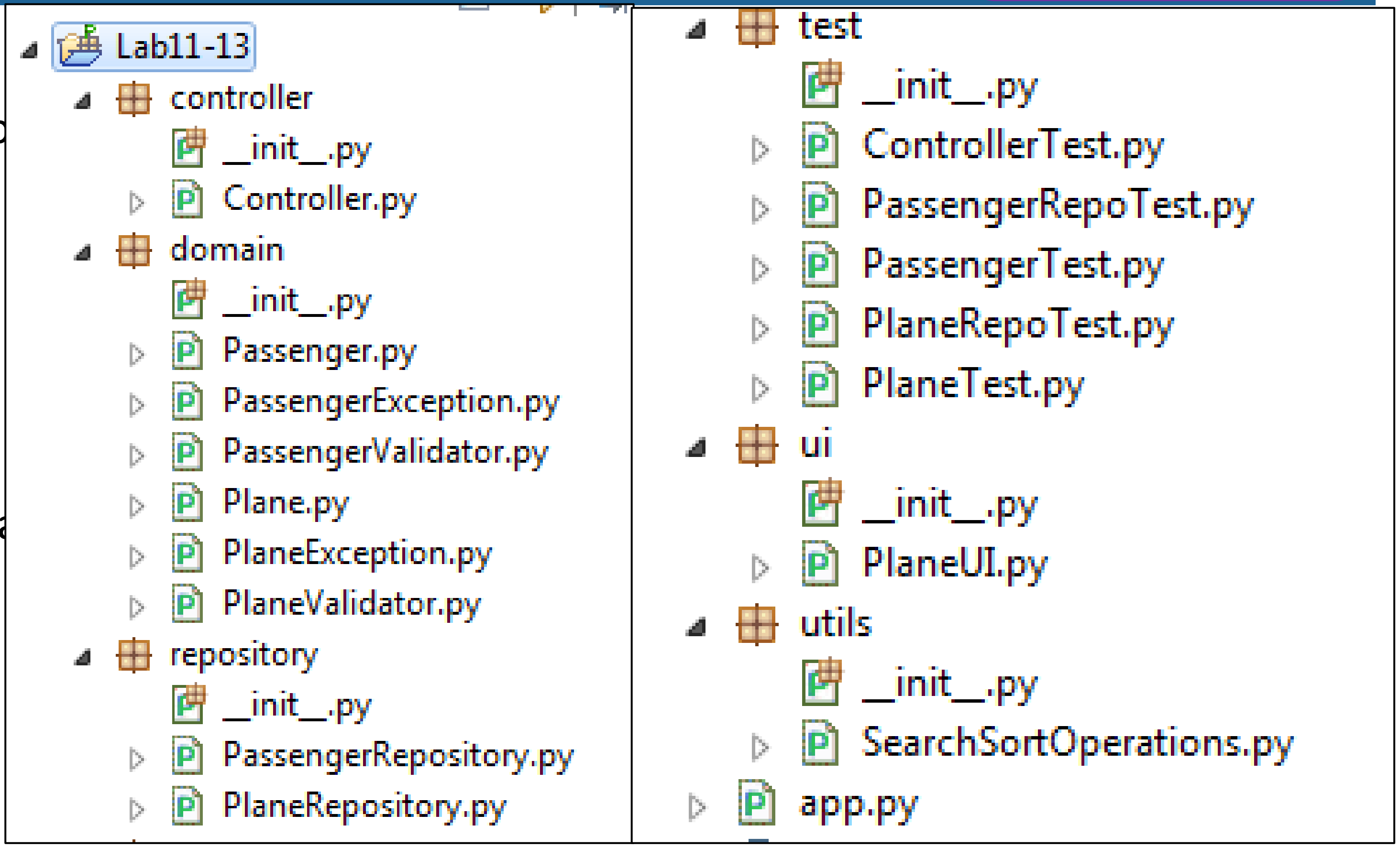
Controller:
the first object beyond the UI layer that receives and coordinates ("controls") a system operation
- Delegates to other objects the work that needs to be done
- Coordinates or controls the activity
- It does not do much work itself

7

Layered architecture

Programming in the large

- User Interface
  - Functions, modules, classes fo
  - *UI / View / Presentation*
- Domain
  - The logic of the application
  - *Business Logic / Model*
- Infrastructure
  - Functions with a general chara
  - *Utils*
- Coordinator
- Controller
- Repository
- Test



Course content

Programming in the large

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types *(Lectures 5, 6)*
5. Software development principles *(Lecture 4, 7)*
6. Testing and debugging *(Lecture 7)*

Programming in the small

7. Recursion
8. Complexity of algorithms
9. Search and sorting algorithms
10. Problem solving methods

Testing and debugging

- Blackbox testing
- Whitebox testing

- Testing levels
- Testing in Python: unittest

7

Testing methods

Programming in the large

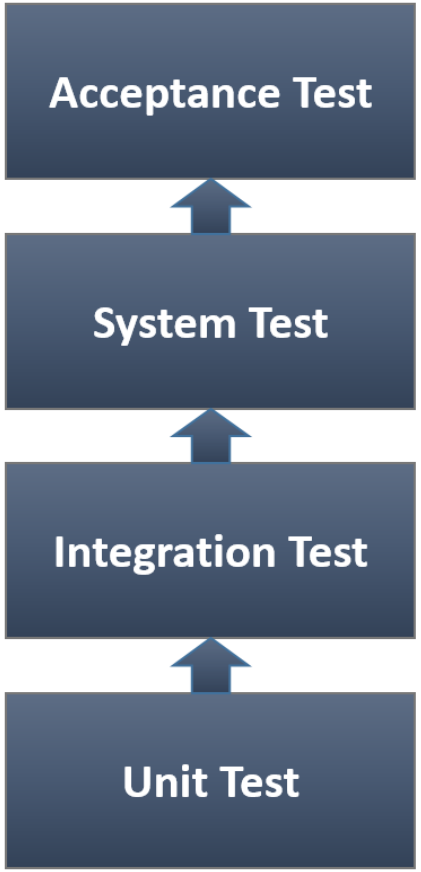
- Blackbox testing
  - Choose the testing data based on the specification of algorithms (data, results) without looking at the code
  - Test if the application does what is supposed to do
    - Normal values
    - Boundary conditions on the values e.g empty list, large numbers, etc
    - Error conditions
- Whitebox testing
  - Choose testing cases based on the code of the algorithms
  - `If:` Test all parts of conditionals
  - `While, for:` Test all cases to exit the loop , Loop not entered, Loop executed only once or several times

7

Testing Levels

Programming in the large

- Unit Test
  - Verify the functionality of a specific section of code e.g. a function
  - Test small parts of the program independently
- Integration Test
  - Test different parts of the system in combination
  - Bottom-up approach: based on the results of unit testing
- System Test
  - Tests how the program works as a whole after all modules have been tested
- Acceptance Test
  - Check that the system complies with user requirements and is ready for use



7 Testing in Python

Programming in the large

- Automated Testing
- **unittest**
  - Python framework for writing Unit Tests
  - provides a class **TestCase** and a **main()** method

    ```
    from unittest import TestCase, main OR import unittest
    ```
- Test classes in Python
  - Test classes should extend TestCase and contain at least one method starting with test_
  - Test methods contain assertions (assertEqual, assertTrue, etc)
- Running the tests
  - **unittest.main** method looks for all classes derived from TestCase
  - Runs all the tests and reports

Course content

Programming in the large

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types *(Lectures 5, 6)*
5. Software development principles *(Lecture 4, 7)*
6. Testing and debugging *(Lecture 7)*

Programming in the small

7. Recursion *(Lecture 8)*
8. Complexity of algorithms
9. Search and sorting algorithms
10. Problem solving methods

Recursion

- A programming technique where a function calls itself
- Basic concepts
  - Recursive element – an element that is defined by itself
  - Recursive algorithm – an algorithm that calls itself
- Recursion can be:
  - **Direct** – a function calls itself (`f` calls `f`)
  - **Indirect** – a function `f` calls a function `g`, function `g` calls `f`
- Main idea of developing a recursive algorithm for a problem of size n
  - **Base case**
    - How to stop recursion
    - Identify the base case solution (for n=1)
  - **Inductive step**
    - Break the problem into a simpler version of the same problem plus some other steps

Course content

Programming in the large

Programming in the small

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types *(Lectures 5, 6)*
5. Software development principles *(Lecture 4, 7)*
6. Testing and debugging *(Lecture 7)*
7. Recursion *(Lecture 8)*
8. Complexity of algorithms *(Lecture 8, 9)*
9. Search and sorting algorithms
10. Problem solving methods

8

Complexity of algorithms

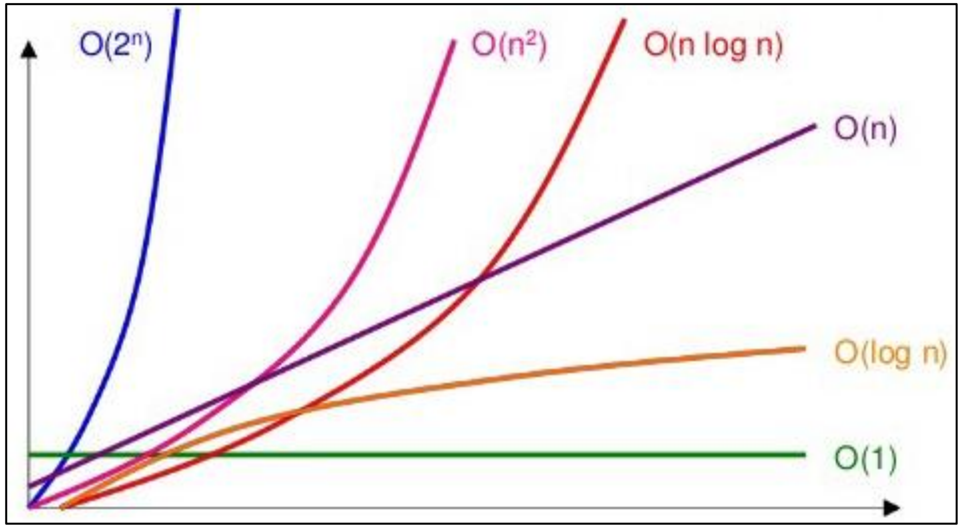
Programming in the small

- Complexity in time
  - Running time of an algorithm:
    - It is not a fixed number but a function *T(n)* that depends on the size *n* of the input data
    - Measures the basic steps the algorithm makes
  - Best case, Worst case, Average case
  - Exact steps vs Big Oh or O() notation
  - O(n) measure
    - How the running time grows depending on the input data size
    - Expression for the number of operations -> asymptotic behavior as the problem gets bigger
- Complexity in space
  - Estimates the **space** (memory) that an algorithm needs to store input data, output data and any temporary data

Complexity classes

O(1)	Constant running time	e.g. 1, 47, 100	Add an element to a list
O(log n)	Logarithmic running time	e.g. 10 + log n	Find an element in a sorted list
O(n)	Linear running time	e.g. n, 3n, 10n+100	Find an entry in an unsorted list
O(n log n)	Log-linear running time	e.g. n + n log n	Sort a list (MergeSort, QuickSort)
$O(n^c)$, c is constant	Polynomial running time	e.g. n^2+1, n^3+n^2+5n	Shortest path between two nodes
$O(c^n)$, c is constant	Exponential running time	e.g. 2^n+1, 3^n	Traveling Salesman Problem (TSP)

$O(n^2)$ - quadratic time
$O(n^3)$ - cubic time



Course content

Programming in the large

1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types *(Lectures 5, 6)*
5. Software development principles *(Lecture 4, 7)*
6. Testing and debugging *(Lecture 7)*

Programming in the small

7. Recursion *(Lecture 8)*
8. Complexity of algorithms *(Lecture 8, 9)*
9. Search and sorting algorithms *(Lecture 9, 10)*
10. Problem solving methods

9

Search methods

Programming in the small

- **Sequential search**
  - Basic idea: the elements of the list are examined one by one (the list can be ordered or not)
  - O(n)
- **Binary search**
  - Basic idea: the problem is divided in two similar but smaller subproblems (the list has to be ordered)
  - O(log n)
- Python
  - Functions index, count, find

Sorting methods

- Selection sort
  - Swap the smallest element with the first one & repeat for all elements
  - $O(n^2)$
- Insertion sort
  - Insert each element at the correct position in a sublist with the elements already sorted
  - $O(n^2)$
- Bubble sort
  - Compare any 2 consecutive elements and swap them if not in correct order
  - $O(n^2)$
- Quick sort
  - Divide and conquer: divide the list in 2 parts and sort the sublists
  - $O(n \log n)$

10

Sorting in Python

Programming in the small

- `list.sort()`
- `sorted(lista)`
- Lambda expressions
- Writing general functions

```python
def mySearch(l, cond):
    '''
    ...
    '''
    result = []
    for i in range(0, len(l)):
        if (cond(l[i])):
            result.append(l[i])
    return result

def mySort(l, relation):
    '''
    ...
    '''
    for i in range(0, len(l) - 1):
        for j in range(i + 1, len(l)):
            if (not relation(l[i], l[j])):
                aux = l[i]
                l[i] = l[j]
                l[j] = aux
```

```python
def sortMembersByInfo(self):
        member_list = self.__data[:]
        mySort(member_list, lambda x, y:
                                x.getInfo() < y.getInfo())
        return member_list
```

Course content

Programming in the large

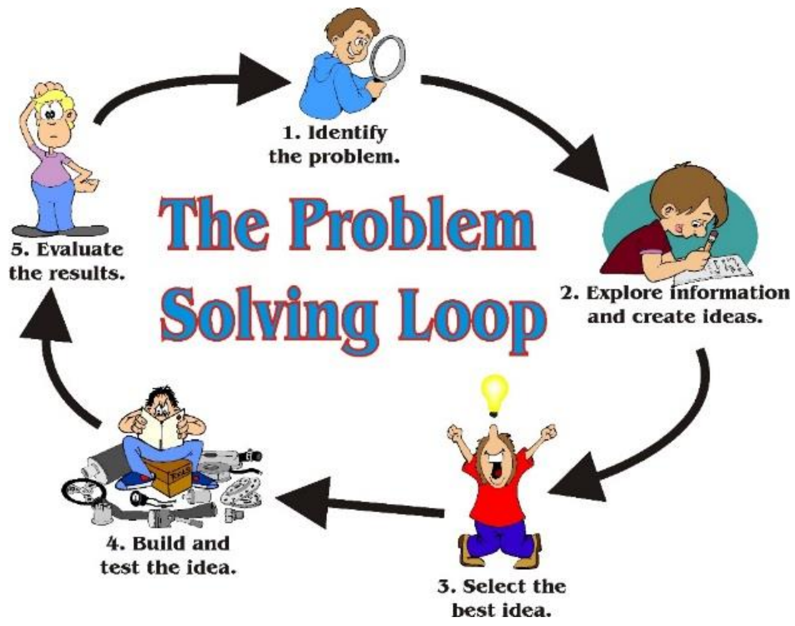
1. Software development process *(Lectures 1, 2)*
2. Procedural programming *(Lecture 2)*
3. Modular programming *(Lecture 3)*
4. Abstract data types *(Lectures 5, 6)*
5. Software development principles *(Lecture 4, 7)*
6. Testing and debugging *(Lecture 7)*

Programming in the small

7. Recursion *(Lecture 8)*
8. Complexity of algorithms *(Lecture 8, 9)*
9. Search and sorting algorithms *(Lecture 9, 10)*
10. Problem solving methods *(Lectures 11, 12)*

Problem solving methods

- Solving problems by search using standard methods
  - Exact methods
    - **Generate and test**
    - **Backtracking**
    - **Divide and conquer**
    - **Dynamic programming**
  - Heuristic methods
    - **Greedy method**



11

Generate and test

Programming in the small

- Basic idea
  - Generate a possible solution and verify it
- Mechanism
  - Generate: determine all possible solutions
  - Test: search solutions that are correct (satisfy some conditions)
- When to use it?
  - Problems that can have multiple solutions
  - Problems with restrictions (solutions need to satisfy some conditions)
- Examples
  - Generate permutations with 3 elements

```
#D = D(D1) = D(D1(D2))...
def generate_test(D):
    while (True):
        sol = generate_solution()
        if (test(sol) == True):
            return sol
```

11 Backtracking

Programming in the small

- The ability to undo – *backtrack* - when a potential solution is not valid
- Basic idea:
  - Try every possibility to see if it's a solution: Search space of a solution **s**
  - Sequence of choices: A solution is formed of several elements *s[0], s[1], s[2],...*
- Implementation elements:
  - *init:* generates an empty value for the definition domain
  - *getNext:* returns the next element from the definition domain
  - *isConsistent:* verifies if a (partial) solution is consistent
  - *isSolution:* verifies if a (partial) solution is a complete solution
- Examples
  - Generate permutations with n elements
  - 8 queens problem

Divide and conquer


```
#D = d1 U d2 U d3...U dn
def div_imp(D):
    if (size(D) < lim):
        return rez
    rez1 = div_imp(d1)
    rez2 = div_imp(d2)
    ...
    rezn = div_imp(dn)
    return combine(rez1, rez2, ..., rezn)
```

- Basic idea
  - Divide the problem in sub-problems similar to the initial problem but smaller in size and get the final solution by combining sub-solutions
- Mechanism
  - **Divide**: breaking the problem in sub-problems
  - **Conquer**: solve the sub-problems
  - **Combine**: combine sub-solutions to obtain final solution
- When it can be used
  - A problem ***P*** with the input data ***D*** can be solved by solving the same problem ***P*** but with input data ***d***, where ***d* < *D***
- Examples
  - Find the maximum of a list
  - Cover a chessboard with L shapes

12 Programming in the small

Dynamic Programming

- Basic idea:
  - Break the problem in nested sub-problems P(P1(P2(P3(...(Pn))...)
  - Solve the sub-problems
  - Compute the final solution by combining the sub-solutions
- Applicable in solving problems where:
  - Problems where one needs to find the best decisions one after another
  - The solution is the result of a sequence of decisions dec_1; dec_2; ...; dec_n.
  - The principle of optimality holds (whatever the initial state is remaining decisions must be optimal with regard the state following from the first decision)
- Examples
  - find the longest increasing subsequence from a list of integer numbers
  - Stagecoach problem

Greedy

- Basic idea
  - Break the problem in successive sub-problems & solve them
  - Determine the final solution by successively selecting the best sub-solutions
  - Global optimum = a sequence of local optimas
- Mechanism
  - Divide the problem in successive sub-problems P1, P2, ...Pn
  - Progress to the final solution by selecting at each step the best decision
- When to use Greedy?
  - Problem P (optimization)
  - Solution is the result of a successive selections of local optima
- Examples
  - Coins Problem
  - Knapsack Problem

Reading materials and useful links

1. The Python Programming Language - https://www.python.org/
2. The Python Standard Library - https://docs.python.org/3/library/index.html
3. The Python Tutorial - https://docs.python.org/3/tutorial/
4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
6. K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. http://refactoring.com/catalog/index.html

Exam

- **“Written exam” (40% of final grade)** -> online exam

12 January 2022, 7:30AM

  - ✓ You need to have the minimum required attendance for seminar (75%) and lab (90%)

- **“Practical exam” (30% of final grade)** -> physical presence

Date and time of your last lab (with your own semigroup)

Exams

CAMELIA CHIRA 08/12/2021 16:53

Exam Dates

2021_MIE1_Algorithms_Programming

The written exam will take place online in the last week of the semester: **January 12th 2022, 7:30 AM** (online, Microsoft Teams)

The retake written exam date is: **February 17th 2022, 10:00 AM** (online, Microsoft Teams)

The **practical exam** will take place in **the last week of the semester** during the lab hours for each semigroup (with physical presence of students).

The **retake practical exam** will take place on the same date as the written exam as follows: **February 17th 2022, 16:00, Lab 338** (with physical presence of students).

Written exam – online

- **Quiz**
- At least 15 questions
- Time: 20 minutes
- Both theoretical and practical questions
- Types of questions:
  - Open-ended
  - Multiple choice
  - File upload

Written exam – online

- Examples of theoretical questions
  - *What is test-driven development?*
  - *What is the difference between a local variable and a global variable?*
- Examples of practical questions

```python
class Person():
    '''
    classdocs
    '''


    def __init__(self, name, age):
        '''
        Constructor
        '''
        self.__name = name
        self.age = age
```

Flower

+name: String
+price: Integer

+__init__()
+getName(): String
+setName(String)
+getPrice(): Integer
+setPrice(Integer)
+compare(Flower): Boolean

```python
def bubbleSort(lst):
    for i in range(len(lst)):
        ...
            if lst[j] > lst[j+1]:
                lst[j], lst[j+1] = lst[j+1], lst[j]
    return lst
```

Simulation exam….