Algorithms and Data Structures in Biology

Introduction to the Course

Ugo Dal Lago

University of Bologna, Academic Year 2018/2019

Section 1

Organization

Organization

- **Webpage**
  - `http://www.cs.unibo.it/~dallago/ADSB1819/`
- **Email**
  - `ugo.dallago@unibo.it`
- **Office Hours**
  - *Where*: see `http://www.cs.unibo.it/~dallago`
  - *When*: there is no fixed office hours, just write me an email and we will fix an appointment.
- **Teaching Assistant**
  - Thomas Leventis, `thomas.leventis@irif.fr`
  - He takes care of the lab sessions.

Structure and Schedule

- First Module: **Theory**
  - *Monday* 11.00am-1.00pm, *Friday* 9.00am-11.00am.
  - The basics of the theory of algorithms, and some design patterns: exhaustive search, dynamic programming, divide et impera, etc.
  - All lectures will be given by Ugo Dal Lago.
- Second Module: **Lab**
  - *Monday*, 2.00pm-6.00pm.
  - Performance evaluation of algorithms as implemented in the `Python` programming language through the `cProfile` library. You will also learn how to write a report by way of the $\LaTeX$ system.
  - Initially, both teachers will be present at the lab, while later on only Thomas Leventis will be there.

Textbooks and Exams

- **Textbook**
  - The first module, being theoretical, requires a textbook.
  - Neil C. Jones, Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms (Computational Molecular Biology)*. The MIT Press. 2004.
- **Exams**
  - The first module's exam will be *written*, and will be given *at the end* of the course.
  - The second module's exam will consist in three assignments, that needs to be completed in a week each. In *very exceptional* cases, in which students cannot complete one or more assignments on time, the teachers will decide how to

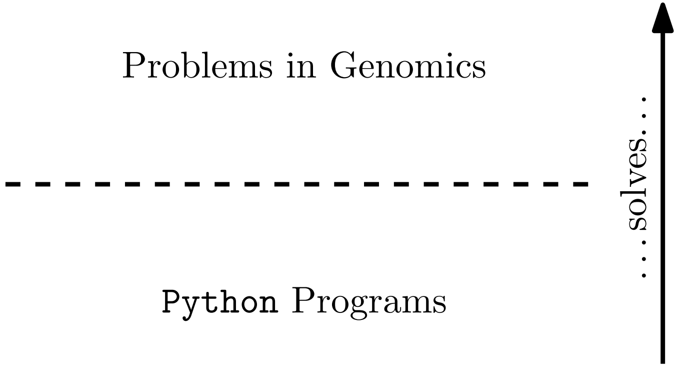
Section 2

The “What” and the “Why”

The Course's Objectives

- **Problems and Algorithms**
  - Algorithms are the middle ground between problems and programs.
- **Measuring Algorithms' Complexity**
  - Different algorithms (solving the same problem) may do so quite differently.
  - How should we decide *which one* of the possibly many algorithms is the right one for our needs?
- **How to Design Algorithms**
  - Although designing algorithmics requires a nontrivial dose of creativity, there are *a few recipes* that work remarkably well in many cases.
  - We will learn a few of them.
- **Proving Algorithms Correct**
  - Suppose you designed an algorithm for a

“Why Should I Learn Algorithmics?”



“Why Should I learn Algorithmics?”

- The workflow you have in mind is, very likely, the following one:
  1. In my daily work as an expert in genomics, I **stumble upon a problem** which seems to have a nice, nontrivial, computational content.
  2. I have some ideas about how to solve it via `Python`, but the problem seems to be **too complicated** to be solved *directly*.
  3. I thus **look for** a module which provides some dedicated function solving the kind of problem I encountered. Hey! There should be one, shouldn’t it?
  4. Once I find it, I simply **import** the module, **invoke** the function, and that’s it.
- But is all this going to *always* work?
  - The set of problems you will encounter is not fixed once and for all: it changes over time.
  - You cannot just wait until *someone* implements a module for you!

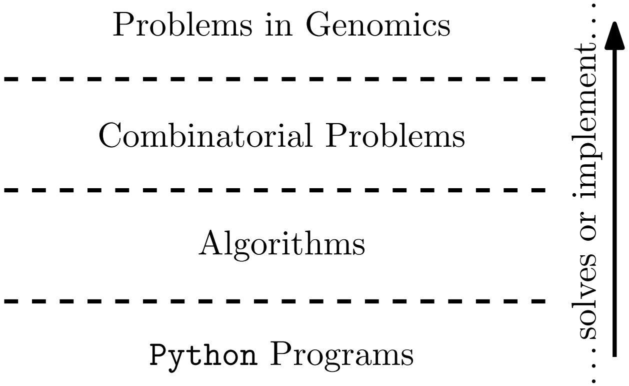
“Why Should I learn Algorithmics?”

- The workflow you have in mind is, very likely, the following one:
    1. In my daily work as an expert in genomics, I **stumble upon a problem** which seems to have a nice, nontrivial, computational content.
    2. I have some ideas about how to solve it via `Python`, but the problem seems to be **too complicated** to be solved *directly*.
    3. I thus **look for** a module which provides some dedicated function solving the kind of problem I encountered. Hey! There should be one, shouldn’t it?
    4. Once I find it, I simply **import** the module, **invoke** the function, and that’s it.
- But is all this going to *always* work?
    - The set of problems you will encounter is not fixed once and for all: it changes over time.
    - You cannot just wait until *someone* implements a module for you!

“Why Should I learn Algorithmics?”

- The workflow you have in mind is, very likely, the following one:
    1. In my daily work as an expert in genomics, I **stumble upon a problem** which seems to have a nice, nontrivial, computational content.
    2. I have some ideas about how to solve it via `Python`, but the problem seems to be **too complicated** to be solved *directly*.
    3. I thus **look for** a module which provides some dedicated function solving the kind of problem I encountered. Hey! There should be one, shouldn’t it?
    4. Once I find it, I simply **import** the module, **invoke** the function, and that’s it.
- But is all this going to *always* work?
    - The set of problems you will encounter is not fixed once and for all: it changes over time.
    - You cannot just wait until *someone* implements a module for you!

“Why Should I Learn Algorithmics?”



An Example: a Problem from Genomics

- Suppose you want to verify whether a strand of DNA you obtained from a database contains (or not, and how many times) a sub-strand which looks *similar* to a given sequence of interest, for example

`AACTTCGG`

- In other words, we not only want to look for exact occurrences of the sequence, but also for approximate ones, like the following:

`AACTCGG`   `AACTCCGG`   `AATCTTCGG`

An Example: a Problem from Genomics

- Suppose you want to verify whether a strand of DNA you obtained from a database contains (or not, and how many times) a sub-strand which looks *similar* to a given sequence of interest, for example

```
AACTTCGG
```

- In other words, we not only want to look for exact occurrences of the sequence, but also for approximate ones, like the following:

```
AACTCGG     AACTCCGG     AATCTTCGG
```

An Example: Approximate String Matching

- Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, the expression Σ^* indicates the set of all finite sequences of elements from Σ, called *strings from* Σ.
- Examples of strings from $\{0, 1\}$ are:

$$\varepsilon \qquad 0101 \qquad 00000 \qquad 1 \qquad 01101001$$

- Given two strings $s, t \in \Sigma^*$, their *edit distance* $\delta(s, t)$ is minimum number of insertions, erasure and modifications (of simbols from Σ) necessary to turn s into t. As an example

$$0001010 \rightarrow 000010 \rightarrow 000000 \rightarrow 1000000$$

$$\delta(00010101, 1000000) \leq 3$$

- **Approximate String Matching (ASM)**
  - Input: two strings s, t and a natural number n.
  - Output: all the substrings $s_1, \ldots, s_k$ of s such that $\delta(s_i, t) \leq n$.

An Example: Approximate String Matching

- Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, the expression Σ^* indicates the set of all finite sequences of elements from Σ, called *strings from* Σ.
- Examples of strings from $\{0, 1\}$ are:

$$\varepsilon \qquad 0101 \qquad 00000 \qquad 1 \qquad 01101001$$

- Given two strings $s, t \in \Sigma^*$, their *edit distance* $\delta(s, t)$ is minimum number of insertions, erasure and modifications (of simbols from Σ) necessary to turn s into t. As an example

$$0001010 \rightarrow 000010 \rightarrow 000000 \rightarrow 1000000$$

$$\delta(00010101, 1000000) \leq 3$$

- **Approximate String Matching (ASM)**
  - Input: two strings s, t and a natural number n.
  - Output: all the substrings $s_1, \ldots, s_k$ of s such that $\delta(s_i, t) \leq n$.

An Example: Approximate String Matching

- Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, the expression Σ^* indicates the set of all finite sequences of elements from Σ, called *strings from* Σ.
- Examples of strings from $\{0, 1\}$ are:

$$\varepsilon \qquad 0101 \qquad 00000 \qquad 1 \qquad 01101001$$

- Given two strings $s, t \in \Sigma^*$, their *edit distance* $\delta(s, t)$ is minimum number of insertions, erasure and modifications (of simbols from Σ) necessary to turn s into t. As an example

$$0001010 \rightarrow 000010 \rightarrow 000000 \rightarrow 1000000$$

$$\delta(00010101, 1000000) \leq 3$$

- **Approximate String Matching (ASM)**
  - Input: two strings s, t and a natural number n.
  - Output: all the substrings $s_1, \ldots, s_k$ of s such that $\delta(s_i, t) \leq n$.

Combinatorial Problems

- Approximate string matching is a typical example of a **combinatorial problem**.
- It generalises, thus helping to solve, the problem from genomics we mentioned a few slides ago.
- It is dubbed **combinatorial**, because it is formulated in the language of finite structures, and in particular as a problem on strings.
- We of course can try to solve the problem by directly writing some `Python` code, or by looking at a module providing a function which fits our needs.
- We could instead look at the problem in a principled way, and look for an *algorithm* which *efficiently* solves the ASM problem.
  - This is what we will do in this course.

Thank You!

Questions?