582670 Algorithms for Bioinformatics

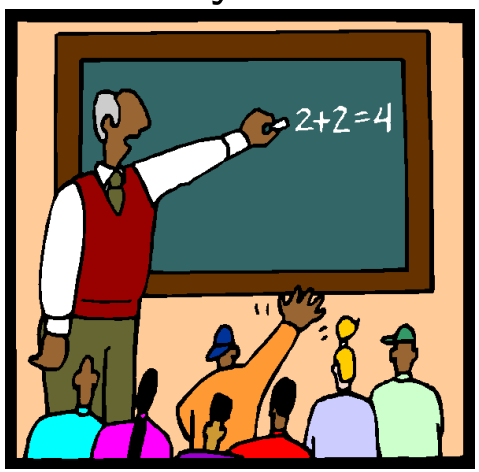
Lecture 1: Primer to algorithms and molecular biology

2.9.2014

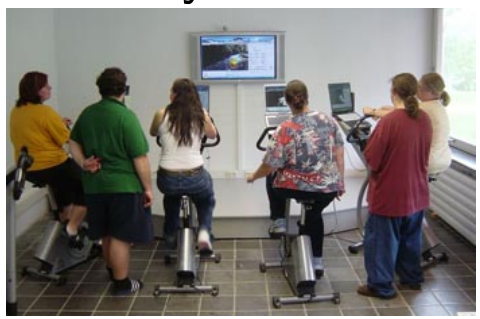
These slides are based on previous years' slides of
Alexandru Tomescu, Leena Salmela and Veli Mäkinen

Course format

Thursday 12-14



Tuesday 10-12



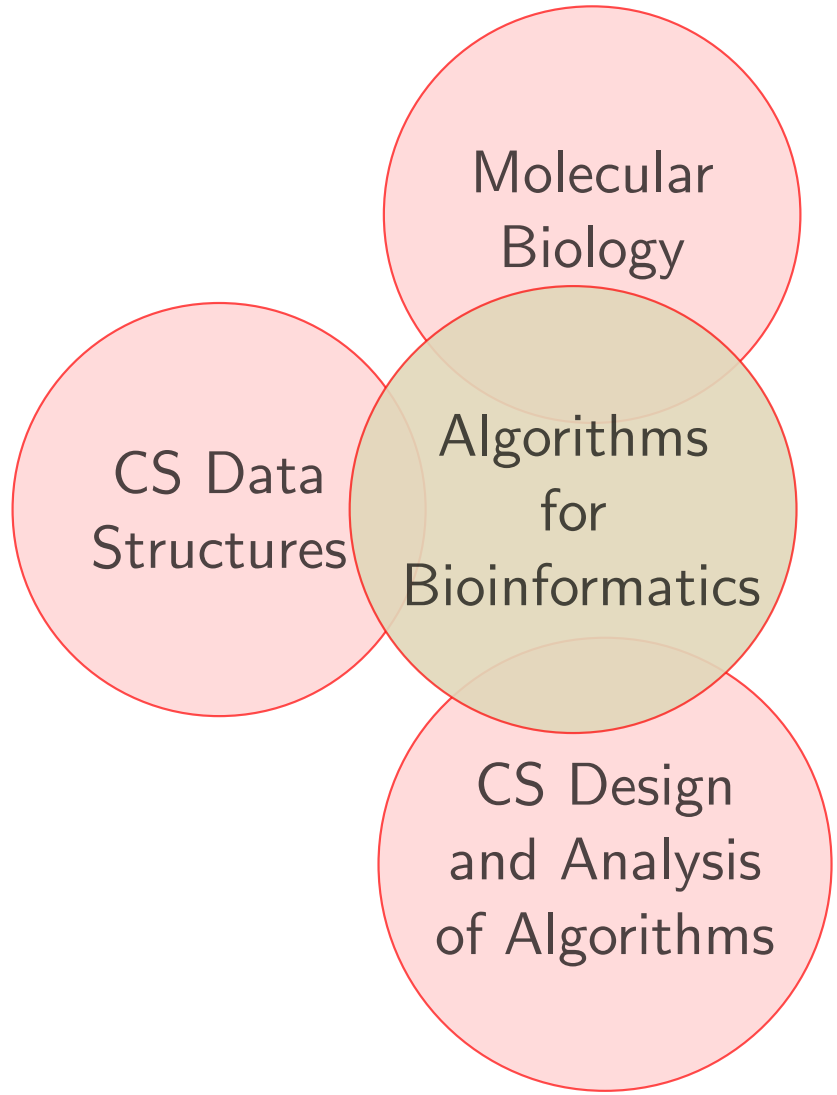
Tuesday 12-14

Grading

- Exam 48 points
- Exercises 12 points
  - 30% $\implies$ 1
  - 85% $\implies$ 12
- Grading $\sim 30 \implies 1$, $\sim 50 \implies 5$ (depending on difficulty of exam)
- Tuesday study group is mandatory!
  (Inform beforehand if you cannot attend)

Course overview

- Introduction to algorithms in the context of molecular biology
- Targeted for
  - biology and medicine students
  - first year bioinformatics students
  - CS / Math / Statistics students thinking of specializing in bioinformatics
- Some programming skills required
  - We will use Python in this course
- Not as systematic as other CS algorithm courses, emphasis on learning some design principles and techniques with the biological realm as motivation

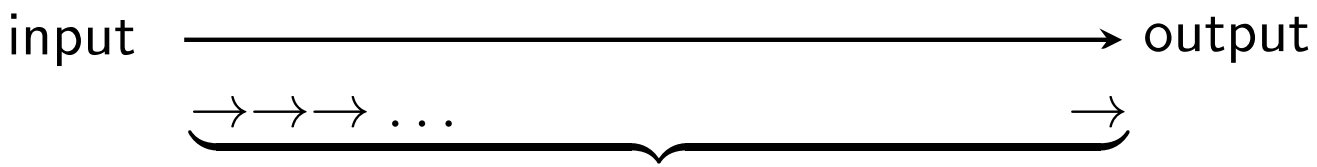


Algorithms *for* Bioinformatics

- State-of-the-art algorithms in bioinformatics are rather involved
- Instead, we study toy problems motivated by biology (but not too far from reality) that have clean and introductory level algorithmic solutions
- The goal is to arouse interest to study the real advanced algorithms in bioinformatics!
- We avoid statistical notions to give algorithmic concepts the priority
- Continue to further bioinformatics course to learn the practical realm

Algorithm

Well-defined problem Solution to problem



number of steps: $f(\text{size of input})$

Homework:
Find out what the following algorithm running time notions mean:

$$f(n) \in O(g(n)) \qquad g(n) \in \Omega(f(n))$$

$$f(n) \in o(g(n)) \qquad g(n) \in \omega(f(n))$$

$$f(n) \in \Theta(g(n))$$

Algorithms in Bioinformatics

Weakly defined problem → Solution to problem

input ⟶ output=input‘ ⟶ output‘=input“ ⟶ output“

- Reasons:
  - Biological problems usually too complex to admit a simple algorithmic formulation
  - Problem modeling sometimes leads to statistical notions
- Problematic for CS theory:
  - optimal solutions to subproblems do not necessarily lead to best global solution

Algorithms in Bioinformatics

Plenty of important subproblems where algorithmic techniques have been vital:

- Fragment assembly $\Longrightarrow$ human genome
- Design of microarrays $\Longrightarrow$ gene expression measurements
- Sequence comparison $\Longrightarrow$ comparative genomics
- Phylogenetic tree construction $\Longrightarrow$ evolution modeling
- Genome rearrangements $\Longrightarrow$ comparative genomics, evolution
- Motif finding $\Longrightarrow$ gene regulatory mechanism
- Biomolecular secondary structure prediction $\Longrightarrow$ function
- Analysis of high-throughput sequencing data $\Longrightarrow$ genomic variations in populations

Course prerequisites

- Programming skills
- High-school level biology++
  - 57780: Molecular genetics reading group recommended to be taken in parallel
  - To avoid overlap with other bioinformatics courses, we do not cover any more biology than is necessary to motivate the problems

Outline

Programming in this Course

- We will use Python
- What we need (in this course):
  - Built-in data types
  - Syntax for control flow statements
  - Function definitions
- What we can omit (i.e. software engineering):
  - Standard library, OOP, exceptions, I/O, etc.

Assignment

Pseudocode

$b \leftarrow 2$

$a \leftarrow b$

Python

```
b = 2

a = b

print a
```

Arithmetic

Pseudocode

DIST (x_1, y_1, x_2, y_2)

1. $dx \leftarrow (x_2 - x_1)^2$
2. $dy \leftarrow (y_2 - y_1)^2$
3. return $\sqrt{dx + dy}$

Python

```
from math import sqrt


def dist(x1, y1, x2, y2):
    dx = pow(x2-x1, 2)
    dy = pow(y2-y1, 2)
    return sqrt(dx+dy)


print dist(0, 0, 3, 4)
```

Conditional

Pseudocode

MAX (a, b)

1 if $(a < b)$

2 return b

3 else

4 return a

Python

```
def MAX(a, b):
    if a < b:
        return b
    else:
        return a

print MAX(1,99)
```

for loops

Pseudocode

SumIntegers (n)

1 $sum \leftarrow 0$

2 for $i \leftarrow 1$ to n

3 $\quad sum \leftarrow sum + i$

4 return sum

Python

```
def SUMINTEGERS(n):
    sum = 0
    for i in range(1,n+1):
        sum = sum + i
    return sum

print SUMINTEGERS(10)
```

while loops

Pseudocode

AddUntil (b)

1 $i \leftarrow 1$

2 $total \leftarrow i$

3 while $total \leq b$

4 $\quad i \leftarrow i + 1$

5 $\quad total \leftarrow total + i$

6 return i

Python

```
def ADDUNTIL(b):
    i = 1
    total = i
    while total <= b:
        i = i + 1
        total = total + i
    return i

print ADDUNTIL(25)
```

Recursion

Pseudocode

$$F(n) = \begin{cases} 0, & \text{when } n = 0 \\ 1, & \text{when } n = 1 \\ F(n-1) + F(n-2), & \text{otherwise} \end{cases}$$

Python

```
def RECURSIVEFIBONACCI(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        a = RECURSIVEFIBONACCI(n-1)
        b = RECURSIVEFIBONACCI(n-2)
        return a+b

print RECURSIVEFIBONACCI(8)
```

Lists

Python

```
l = [0] * 3

l[0] = 1    # list is mutable

l = range(1,4)

l.append('four')

l = [2**i for i in range(6)]

l2 = l[2:4]
```

Output

```
[0,0,0]

[1,0,0]

[1,2,3]

[1,2,3,'four']

[1,2,4,8,16,32]

[4,8]
```

List access

Pseudocode

FIBONACCI (n)

1 $F_0 \leftarrow 0$

2 $F_1 \leftarrow 1$

3 for $i \leftarrow 2$ to n

4 $\quad F_i \leftarrow F_{i-1} + F_{i-2}$

5 return F_n

Python

```
def FIBONACCI(n):
    F = [0]*(n+1)
    F[0] = 0
    F[1] = 1
    for i in range(2,n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]

print FIBONACCI(8)
```

Strings

Python

```
s='Hello'


s[0] = 'C'     # error: str
s.append('!') # is immutable


s = s + '!'


s = s[:5]


s = s.upper()
```

Output

```
Hello




Hello!


Hello


HELLO
```

Immutable vs Mutable

Immutable (int, str, ...)

```
a = 2
b = a
b = b + 1    # does not change a

s = 'Hello'
t = s
t = t + '!'  # does not change s
```

Mutable (list, set, dict, ...)

```
l = [0]
m = l          # shallow copy of l
m.append(1)    # changes also l

l = [0]
m = l
m = [0, 1]     # does not change l

l = [0]
m = l[:]       # deep copy of l
m.append[1]    # does not change l
```

Pass arguments by reference? - No.

Immutable (int, str, ...)

```
def ADDONE(x,y):
    x = x + 1    # x and y
    y = y + 1    # are local

# return a tuple instead
def ADDONE(x,y):
    return x+1, y+1

x,y = ADDONE(x,y)
```

Mutable (list, set, dict, ...)

```
def CLEAR(l):
    l = []    # l is local

# any mutable can still be
# changed in place, e.g.:
def CLEAR(l):
    l[:] = []

def ADDONE(l,i):
    l[i] = l[i] + 1
```

Multidimensional lists

Python

```
l = [[0] * 2] * 3  # Caution!
                   # You probably
                   # don't want
                   # to do this!
l[0][0] = 1



# This is safe:
l = [[0]*2 for i in range(3)]

l[0][0] = 1
```

Output (print l)

```
[[0, 0], [0, 0], [0, 0]]


[[1, 0], [1, 0], [1, 0]]



[[0, 0], [0, 0], [0, 0]]

[[1, 0], [0, 0], [0, 0]]
```

Sets and Dictionaries

Python

```
l = [0, 2, 1]
s = set(l)
s = {0, 1, 2}
s = s | {3, 4}
s = set('hello')
'e' in s

d = {'apple':2, 'orange':5}
d['apple']
d['orange'] = 8
```

Output

```
[0, 2, 1]
set([0, 1, 2])
set([0, 1, 2])
set([0, 1, 2, 3, 4])
set(['h', 'e', 'l', 'o'])
True

{'orange': 5, 'apple': 2}
2
{'orange': 8, 'apple': 2}
```

Large(r) data sets

- For mutable strings, use e.g.
  - `array.array('c', 'Hello')`
  - `bytearray('Hello')`
- `list` uses a lot of memory ($\sim$ 16 bytes per `int`). For homogeneous data, use e.g.
  - `array.array('l', [1,2,3,4])`
  - `numpy.array([1,2,3,4])`
- `list` is slow for operations at both ends. E.g. for a queue use
  - `collections.deque([1,2,3,4])`

Helpful links

- `http://openbookproject.net/thinkcs/python/english2e/` (Programming tutorial for those who have no programming experience)
- `http://docs.python.org/tutorial/`
- `http://docs.python.org/library/`
- `http://wiki.python.org/moin/BeginnersGuide/`
- `http://wiki.python.org/moin/TimeComplexity/`
- `http://docs.scipy.org/doc/` (NumPy documentation)

Outline

Group 1 (students with biology background)

- One of the fundamental and most deeply studied algorithmic problems is sorting. Before coming to the study group familiarize yourself with the problem (e.g. using Wikipedia) and be ready to explain the idea of couple of well-known sorting algorithms like *insertion sort*, *quicksort*, *merge sort*, and *radix sort*.
- At study group, try to understand the running time $O()$-notion of different sorting algorithms:
  - What happens if you are sorting a set of DNA sequences into lexicographic order instead of integers?
  - What if the set of DNA sequences consists of all *suffixes* of one DNA sequence?

Group 2 (students with CS background)

- Study the slides “molecular biology primer” (found on course web site) before coming to the study group.
- At study group, be ready to explain the material just using the “molecular biology cheat sheet”.