BC205: Algorithms for Bioinformatics. I. An Introduction

Christoforos Nikolaou

March 1, 2018

Algorithms in Bioinformatics

The course will cover:

- An introduction to the concept of Algorithms
- A listing of *some* of the major problems of Bioinformatics
- A detailed description of algorithmic approaches to these problems

Instructors

- Christoforos Nikolaou, Dept. of Biology
  - Introduction
  - Motif Search
  - NGS algorithms
- Ioannis Tsamardinos, Dept. of Computer Science
  - Sequence Alignment and Dynamic Programming
- Kleio Lakiotaki, TA, Dept. of Computer Science

Evaluation

- Exercises to be handed in (depends on the instructor)
- Final exam

Reading

- **Introduction to Algorithms. (Cormen, Leiserson, Rivest and Stein)** *for a general intro, but may be rather tecnhical for biologists*
- **Introduction to Bioinformatics Algorithms. (Pevzner and Jones)** *covers basic Bioinformatics algorithms with a right balance between CS and Biology*
- **Bioinformatics Algorithms. A practical approach (Pevzner and Compeau)** *very good choice for both disciplines, with a lot of practicals*
- **Introduction to Computation and Programming with Python (Guttag)** *basic principles with a Python edge*
- **Genome-scale Algorithm desing (Tomescu, Bellazzougui, Cunial and Makinen)** *NGS-related but quite tecnhical*

Course Outline

- Introduction, concepts and algorithmic warm-up
- Analyzing Sequence Composition
- Motifs: Search, Evaluation and Discovery
- Sequence Alignment
- Data structures for NGS applications
- Algorithms inspired by NGS problems (mapping, peak finding and differential expression)
- Graph Algorithms

What is an Algorithm

Formally

> ***Algorithm***: *A systematic and well-defined procedure that produces, in a finite number of steps, the answer to a question or the solution of a problem.*
> *[Encyclopaedia Britannica]*

Informally

> *"Any well-defined computational procedure that takes some value, or sets of values as input and produces some value, or sets of values as output."*
> *[Cormen, Leiserson, Rivest & Stein]*

> *"Transforming input into output is also good to make some sense!" [me]*

Problems of Bioinformatics *(that we will be discussing)*

- Analyzing Sequence Composition (Algorithmic Introduction)
- Searching/Matching/Extracting Motifs in Sequences (Randomized Algorithms)
- Comparing Sequences through Alignments (Dynamic Programing)
- Phylogenetic Reconstruction (Clustering Algorithms)
- Next-generation Sequencing Analysis (Data Transformations)
- Biological Networks (Graph Algorithms)

(Simple) Examples of Algorithms

1. Finding the largest common divisor of two numbers
2. Sorting a set of integers
3. Calculating the Fibonacci series for up to a number N
4. Finding the square root of an integer

Thinking the problem through

- The hardest part:
  - What is the input?
  - What is the (expected) output?
  - How can we do it?
  - How can we do it faster?

Types of Algorithms (with a more serious problem)

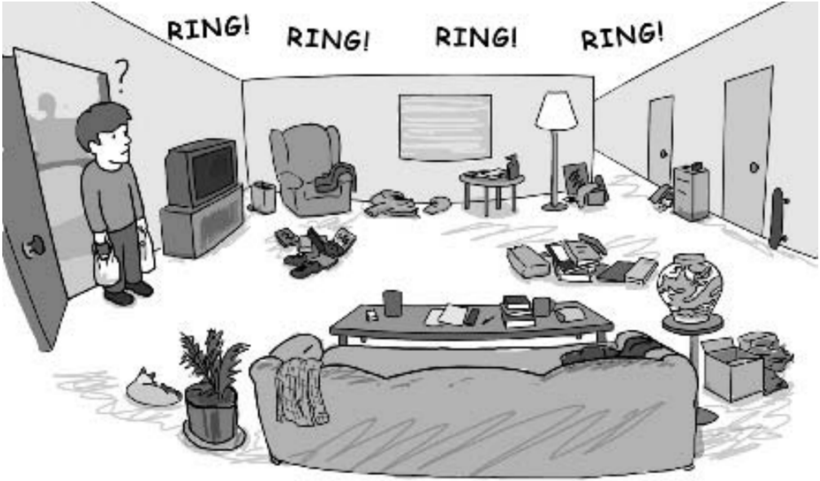


Figure 1: Where's my phone

How should we go about finding the phone?

- Go through every room starting from the closest to the one that is farther away (Exhaustive Search/Brute Force)
- Exclude some/many obvious impossible position (e.g. phone is heard through the ceiling, so you skip searching the floor you are now) (Branch-and-Bound/Pruning)
- Proceed directly towards the sound regardless of obstacle (walls, furniture) in between (Greedy Approach)

How should we go about finding the phone?

- Suppose someone/something can tell you if the phone is/is not in a part of your flat (e.g. “-Mom, is my phone on the kitchen table? -No, it’s not!”). You can then use a series of questions to narrow down the available search space (Divide-and-Conquer, Dynamic Programming)
- Suppose this has happened to you before. Drawing from your experience with real data you search specific “highly likely” places (e.g. between the sofa cushions, under my son’s pillow). (Machine Learning)
- You can always flip a coin before you choose which direction to go search for your phone. It may appear counter-intuitive but sometimes it works (Randomized Algorithms)

Case 1: The Largest Common Divisor Problem

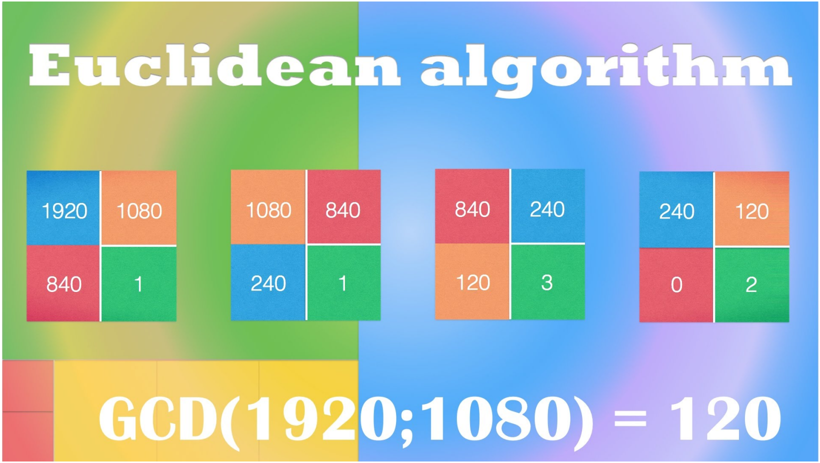


Figure 2: Euclid's LCD Algorithm

Euclid's Algorithm for LCD

1. Start with two numbers a, b ($a > b$)
2. Divide a/b and keep the remainder c
3. Now, divide b/c and keep the remainder d
4. Repeat the division until there is no remainder
5. Report the last divisor and the LCD of a and b.

Let's make it more formal. Pseudocode

Input: A, B

```
if (A == 0) {LCD = B}
if (B == 0) {LCD = A}
if (A & B != 0){
  C=remainder(A/B)
  if (C is greater than 0) {B->A; C->B; goto #}
  if (C equals 0) {print LCD=C; end}
}
```

Writing the whole thing up: Programming

- You can choose your language of choice.
- Our focus will not be so much on coding as on the design and the general approach

Euclid’s Algorith for the Largest Common divisor

(Python Implementation)

```
def euclid(a,b)

# Euclid's Algorithm for the largest common divisor
while (b > 0):
    a, b = b, int(a)%int(b)

print a
```

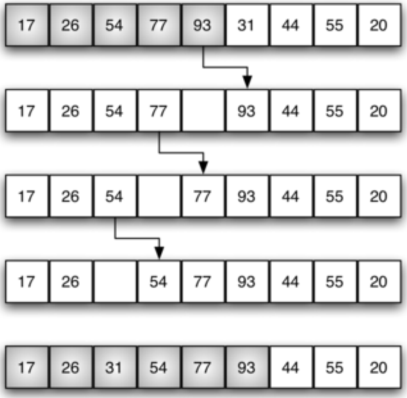
What does this algorithm do?

```
# enters an iterative process if b > 0
while (b > 0): # Checks if the smallest of the
               # numbers is > 0
               # b is basically the remainder #
               #of the division

    # swaps a and b with b and the
    # remainder of the division
    a, b = b, int(a)%int(b)
          # Calculates the division
          # Extracts the remainder
          # Makes the swap

print a # returns the result as the
        # last divisor that (gave 0 remainder)
```

Case 2: Sorting a series of integers

Starting with N integers, order them from the smallest to the largest

Take #1: InsertionSort (Exhaustive)



Need to insert 31
back into the sorted list

93>31 so shift it
to the right

77>31 so shift it
to the right

54>31 so shift it
to the right

26<31 so insert 31
in this position

Figure 3: Insertion Sort

Insertion Sort. Pseudocode

```
Start with a list of L[N] numbers:
    For i in 2 to N:
        value<-L[i]
        j=i-1
        while j>0 and L[j]>value:
          L[j+1]=L[j]
          j=j-1
        L[j+1]<-value
```

Insertion Sort. (Python code)

```python
def insertionSort(alist):
   # starts with the second element till the end of
   #the list
   for index in range(1,len(alist)):
     # assigns the ith-element of the list to a value
     currentvalue = alist[index]
     # marks its position
     position = index
     # if a) the position of the element is not
     # the first in the list
     # and b) if the value of the element
     # is bigger than the previous
     while position>0 and alist[position-1]>currentvalue:
         # exchange positions with the previous element
         alist[position]=alist[position-1]
         position = position-1
     alist[position]=currentvalue
```

Take #2: MergeSort (Divide and Conquer)

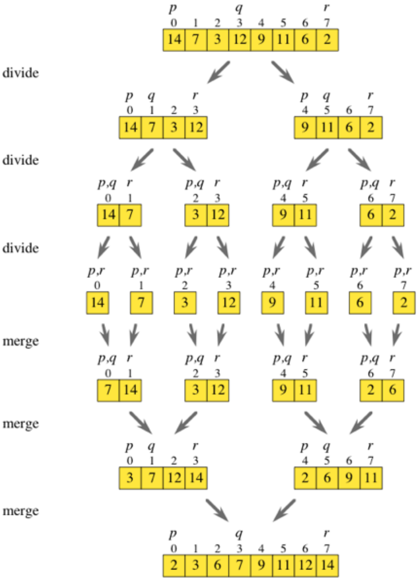


Figure 4: Merge Sort

MergeSort Pseudocode (Recursion)

```
Start with a list of L[N] numbers:
    # Split L[N] into two half-lists: A[N/2] and B[N/2]
    A[N/2]<- Goto #(A[N/2])
    B[N/2]<- Goto #(B[N/2])
    for i in 1:length(A) and j in 1:length(B):
        if (A[i]<B[j]):
          C=C.A[i] # add A[i] to a list C[N]
          remove A[i]
        if (A[i]>B[j]):
          C=C.A[i] # add A[i] to a list C[N]
          remove A[i]
```

Merge Sort (Merging)

```python
def merge(a,b):
    """ Function to merge two arrays """
    c = []
    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]:
            c.append(a[0])
            a.remove(a[0])
        else:
            c.append(b[0])
            b.remove(b[0])
    if len(a) == 0:
        c += b
    else:
        c += a
    return c
```

Merge Sort (Recursive Call)

```
# Code for merge sort
def mergesort(x):
    if len(x) == 0 or len(x) == 1:
        return x
    else:
        middle = len(x)/2
        a = mergesort(x[:middle])
        b = mergesort(x[middle:])
        return merge(a,b)

print mergesort(x)
```

Questions to ask?

- Is the algorithm correct?
    - We can test the correctness of an algorithm by checking a property called *loop invariant*. A loop invariant is (as its name implies a property that remains unchanged through the succession of loops).
- Is it fast?
    - We test the speed of the algorithm by estimating the number of calculations it performs (assignments+logical+arithmetic operations).

A. Correctness

1. Can you think of a loop invariant for the Sorting algorithms?
   1.1 Does it hold **before** starting the iteration?
   1.2 Does it hold **after** every iteration process?
   1.3 Does it provide a control to show you that the process is over?

Insertion Sort. A Loop Invariant

The elements at position L[1..i] are the same before and after each iteration

1. **Before** the beginning of the iteration there is only one element
2. **After** every iteration, they are the same elements, only now they are sorted
3. **At the end** of the run we should have iterated N times

The above 3 conditions prove the correctness of the algorithm

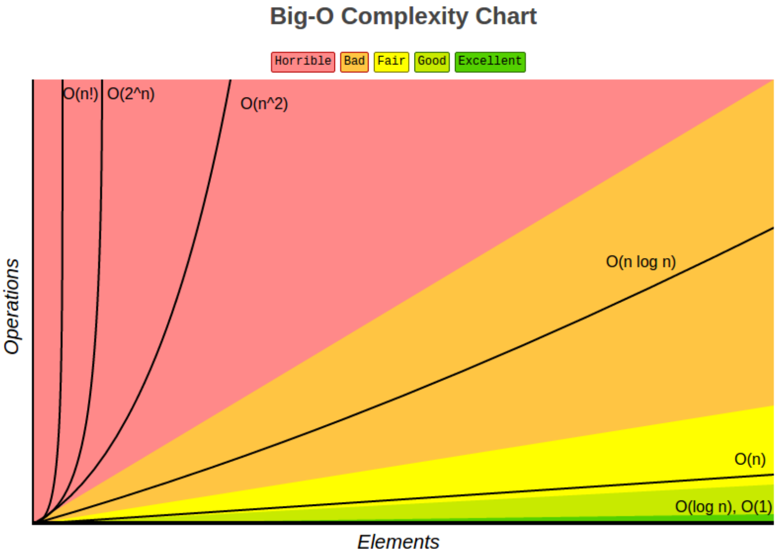
B. Speed

- Calculating the number of processes/calls/calculations made by an algorithm can be strictly formalized. We are interested in the general principle which is:

  How does the number of calculations scale with increasing size of input?

- Computer scientists use a concept called "Time Complexity" to define the type of the algorithm in terms of scalability. This is a property that is dependent on the *structure* of the algorithm and not on the input (of which it is a function).
- We use a notation called Big-O to designate algorithmic time complexity. For instance: $O(n^2)$ means that the algorithm takes time proportional to the square of the input, while $O(n)$ means it is a "linear-time" algorithm

Big-O Notation



`http://bigocheatsheet.com/`

Big-O Notation (Insertion Sort)

- **Insertion Sort**: takes an array of N and scans it N times. Each time it takes an i-subset and makes i-1 comparisons. This is proportional to: T=1+2+3. . . +N-1
  which we know to be equal to [n(n+1)]/2
  and thus **Insertion Sort is O(n^2)**

Big-O Notation (Merge Sort)

- **Merge Sort**: takes an array of N and splits it in half, then sorts each half by recursive calls of the merge function. Let's break this into the two components:
  - Splitting is done into halves which means that for a list of N, log2(N) splits will be required
  - The merging process is done by parsing the elements of A and B lists one at a time, thus for N values it takes O(n) time.

Combination of the two gives that **Merge Sort is O(nlogn)**, which means it is much better than Insertion Sort.

Case 3: Fibonacci Series

Calculate a sum of N numbers where each one is produced as the sum of the two that came immediately before it. (the first two numbers are by definition set to 1)

```
N[1]=1
N[2]=1
N[3]=N[2]+N[1]=1+1=2
N[4]=N[3]+N[2]=2+1=3
N[5]=N[4]+N[3]=3+2=5
etc
```

The problem: Calculate the Fibonacci element number N

Take #1: Using an Array

```
N=int(raw_input("Give number for Fibonacci:"))
def fibonacci(N):
    fib=[]
    fib.append(1)
    fib.append(1)
    for i in range(2,N):
        fib.append(fib[i-1]+fib[i-2])
    return fib[i]

output=fibonacci(N)
print "Fibonacci sum for:",N," is ", output
```

Take #2: Using Recursion

```python
N=int(raw_input("Give number for Fibonacci:"))
def fibonacci(N):
    a, b = 1, 1
    for i in range(N-2):
        a, b = b, a+b
    return b

for i in range(N):
  fib=fibonacci(i-2)+fibonacci(i-1)

print "Fibonacci sum for:",N," is ", fib
```

Fibonacci: Analysis

- Take #1
  - We create an array of length N
  - We go through the array calculating the i-th element with a simple addition of i-1, i-2
- Take #2
  - We swap the values of a, b with b and the sum of the two
  - We recursively call the algorithm for i-1 and i-2

Ask yourself

1. How does array-Fibonacci scale with N?
2. How does recursive-Fibonacci scale with N?
3. What is the Big-O notations of the two

Case 4: Finding the Square Root of an integer

- You are given an integer you need to find its square root. How do you work?
- Consider two approaches:

1. An exhaustive search for the root
2. A more clever way to faster converge to the solution

- It may be of help. Try to play a guessing game with the computer.

Exhaustive Search

1. Provide the integer x

- Restrictions: set limitations on the number (eg. it has to be positive) and the answer (only positive roots will be reported)

2. Start a counter k from a very low value (e.g. 0)
3. Increment k with a small step s (e.g. 0.01)
4. Ask for abs(k**2-x) to be smaller that a given constant (small number) (this has to be at most equal to the step s but not bigger [why?])
5. Provide answer if Condition 4 is fulfilled

Exhaustive Root Search

```
x = float(input('give number:'))
epsilon = 0.01
step = epsilon**2
guess = 0
ans = 0.0
while abs(ans**2 - x) >= epsilon and ans <= x:
    ans += step
    guess +=1
print "iterations="+str(guess)
if abs(ans**2-x) >= epsilon:
    print "failure"
else:
    print "the sqrt of x is approximately:"+str(ans)
```

Another take on root finding. A Guessing Game

1. I am thinking of an integer between 1 and N
2. You can try and guess it. I can only answer if your guess is greater, equal or smaller to my number
3. How many guesses (at most) are required?

The Guessing Game

```
import random
x = random.randint(1,101)

guess=int(raw_input("Guess a number between 1 and 100:"))

while (guess != x):
    if (guess < x):
        print "My number is greater than ", guess
        guess=int(raw_input("Guess the number:"))
    if (guess > x):
        print "My number is smaller than ", guess
        guess=int(raw_input("Guess the number:"))

if (guess == x):
    print "Congratulations! The number was:", guess
```

Binary Search

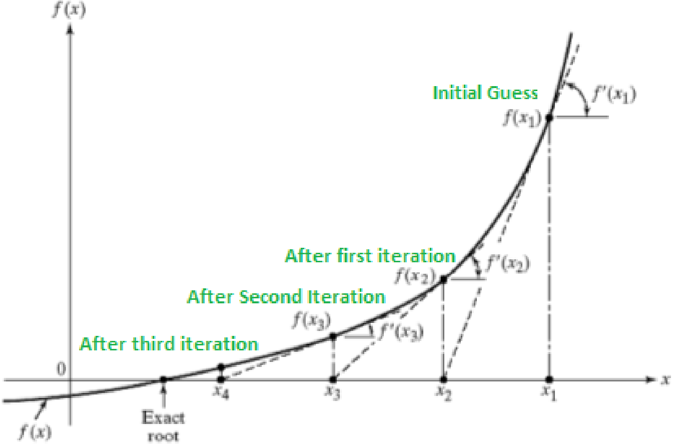
- A much more appropriate technique for estimation, instead of exhaustive enumeration, is binary search. In this, you split the space search in half with every iteration, which means that you will nead at most log_2(N) iterations to parse a space of size N.
- Think of how to apply binary search in the square root finding problem.

Binary Square Root Finding

```
x = float(input('give number:'))
epsilon = 0.01
step = epsilon**2
guess = 0
low = 0
high = max(1.0, x)
ans = (high+low)/2.0

while abs(ans**2 - x) >= epsilon:
    ans += step
    guess +=1
    if ans**2 < x:
        low = ans
    else:
        high = ans
    ans = (high + low)/2.0
print "the sqrt of x is approximately:"+str(ans)
```

Newton-Raphson Method

- It is based on a theorem according to which the approximation of $x_1 = x_0 - f(x_0)/f'(x_0)$ converges faster to the root of f(x).



Newton-Raphson Method

1. Define a search space [A..B]
2. Split search space in 2 -> initialize x=(B-A)/2
3. Set a (sufficiently small) accuracy threshold (below which you will accept solution)
4. Iterate over $x_i = x_{i-1} - f(x_{i-1})/f'(x_{i-1})$ for as long as the answer is above threshold

Newton-Raphson Method for finding the Square Root

```
x = float(input('give number:'))
ans=0
epsilon = 0.01
k = x
guess = k/2.0

while abs(guess**2 - x) >= epsilon:
    ans = ans + 1
    guess = guess - (((guess**2)-k)/(2*guess))

print "iterations="+str(ans)
print "the sqrt of x is approximately:"+str(guess)
```

Exercises

1. Implement the binary search strategy to find the solution of the equation: $2x^2 - 5x - 12 = 0$
2. Implement the Newton-Raphson method to find the solution of the same equation.
3. Compare the number of iterations necessary for the two approaches. Which converges faster?

Next week:

- Enough with this. What about Bioinformatics?
- What we will be discussing in this class may appear detached from the above but it is *not* so.
- Issues like recursion, time complexity and efficiency will matter
- The way we transform the problem into formal sets of questions is crucial.

Some (not so simple) problems

- Given a long DNA sequence can you locate a given string of characters within it.
- Can you say how many times it is there, and where?
- Given two strings of characters can you find the longest common subsequence of a) un-interrupted characters b) characters with gaps c) characters with gaps and also some mismatches?

Bioinformatics Warm-Up

1. You are given a DNA sequence
   - Can you count the number of nucleotides of each of the four bases (A, G, C, T)?
   - How many calculations will you need?
   - How will you implement it?
2. Now consider the same problem only instead of nucleotides we need to count the number of all 8-nucleotides. What do you need to consider to attack the problem?

Why should you care?

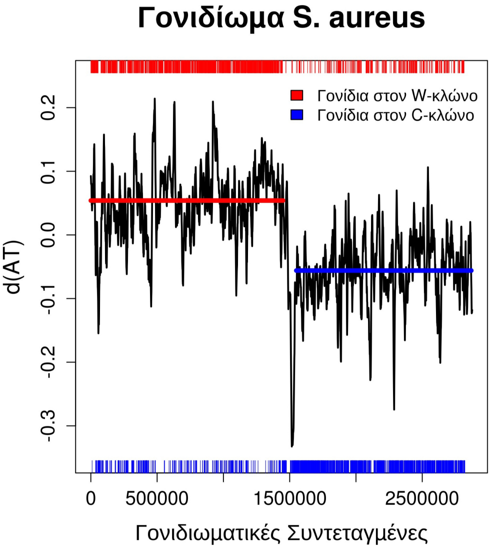


Figure 5: Nucleotide Parity

Nucleotide Skew Analysis of Bacterial Genomes

- Given a bacterial genome
  - Count nucleotides in windows of N base pairs
  - Calculate the scaled AT-skew as (A-T)/(A+T)
  - Create an array of the skew values along the genome
  - Locate the transition point

Exercises (to think about)

1. What is the Big-O of Euclid's algorithm?
2. Euclid's algorithm is slowest for a=F(b), that is when a and b are members of the Fibonacci series. Any idea why should that happen?
3. Find a loop invariant for MergeSort
4. Write a program that calculates the Chargaff Parity Deviation (for the A,T pair) for a given DNA sequence.