BC205: Algorithms for Bioinformatics.

I. An Introduction

Christoforos Nikolaou

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

Evaluation

- Assignments (50%)
- Final Project (50%)

Reading

- **Computational Biology. Christoforos Nikolaou [http://computational-genomics.weebly.com/computational-biology-book.html]**
- **Introduction to Computation and Programming with Python (John V. Guttag)** *general topics on computation but with a lot of useful python code*
- **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) [https://www.bioinformaticsalgorithms.org/read-the-book]** *very good choice for both disciplines, with a lot of practicals*
- **Genome-scale Algorithm desing (Tomescu, Bellazzougui, Cunial and Makinen)** *NGS-related but quite tecnhincal*

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
- Optional (if we have time). Hidden Markov Models, Clustering Algorithms

Part I. Basic Concepts

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

> **Algorithm:** "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]

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

- Analyzing Sequence Composition (Algorithmic Introduction)
- Data Structures (Arrays, Hashes, Trees)
- Searching/Matching/Extracting Motifs in Sequences (Randomized Algorithms)
- Comparing Sequences through Alignments (Dynamic Programing)
- Next-generation Sequencing Analysis (Branch and Bound)
- Biological Networks (Graph Algorithms)

Some (Simple) Problems

1. Iteration and Exhaustive Searches (Brute Force Methods)
    a. Finding the positive root of a quadratic equation
2. Dichotomous (clever) Searches (Divide and Conquer Methods)
    a. Finding the square root of a number
    b. Finding if a k-mer exists in a sequence
3. Iteration vs Recursion
    a. Finding the largest common divisor of two numbers
    b. Calculating the Fibonacci series for up to a number N
4. Randomized Algorithms
    a. Estimating the value of π with a randomized process
5. Application: Sorting a set of integers

Step 1. 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?

Step 2. Formulate the problem

- Break the problem into pieces
- Identify (in detail) a process of simpler problems
- Work out the simpler problems in order

Types of Algorithms

1. Simple, exhaustive iteration

This is the simplest, more obvious computational approach in which we try to solve a problem by enumerating all possible solutions (or something that is close to all possible solutions) and then **exhaustively search** among them for the best one to our problem.

Consider the following example: Someone gives you a DNA sequence and asks you to report whether a given tetranucleotide is present in the sequence. How do you work?

Lets try with the following sequence:

$$S = AC\,AC\,A\,GT\,AC\,AC\,GT\,AT\,AC\,CC\,AGT\,T\,TGC\,AC\,AGT\,T\,T\,T$$

in which we need to check for the existence of:

$$P = AGTT$$

Below we see a program that checks every tetranucleotide in S for being a match with P

```
sequence = 'ACACAGTACACGTATACCCAGTTTGCACAGTTTT'
pattern = 'AGTT'
matches = 0
for i in range(len(sequence)-3): # consider that any string has n-k+1
substrings of length k
    string = sequence[i:i+4]
    if (string == pattern):
        matches += 1
        print("match found at position", i)
print("Pattern was found ", matches, " times.")
```

```
match found at position 19
match found at position 28
Pattern was found  2  times.
```

Lets make this more interesting by checking for a pattern in the entire E. coli genome. And make it a bit more interesting by searching for a 10-mer instead of a simple 4-mer.

Take a pause and think: What is the probability of finding a given 10-mer in the sequence of the E.coli genome which is ~3.1Mbp long?

```
# Reading ecoli genome

file = open('files/ecoli.fa', 'r')
ecoli = ''
```

```
count = 0

for line in file:
    count += 1
    if (count > 1): # the first line contains the non-sequence header
so we discard it
        ecoli += line.replace("\n", "") # we string the newline
character from the end of each line

# Using time to measure time of execution
import time
start_time = time.time()

# Pattern search

pattern = 'AGTTAGGCCT'
#pattern = 'TCGGCATCAG'
matches = 0
k = 10

for i in range(len(ecoli)- k + 1): # consider that any string has n-
k+1 substrings of length k
    string = ecoli[i:i+k]
    if (string == pattern):
        matches += 1
print("Pattern was found ", matches, " times.")

print("--- %s seconds ---" % (time.time() - start_time))
```

```
Pattern was found  0  times.
--- 0.7970407009124756 seconds ---
```

So it takes a bit less than a second to search through the whole genome and report that the given 10-mer doesn't exist in the sequence. This is not bad. But can it be done quicker?

2. Divide and Conquer. Dichotomous Searches

Below we take a slightly different approach. We first split all the k-mers we find in the genome and sort them in a list of kmers.

```
# Reading ecoli genome

file = open('files/ecoli.fa', 'r')
ecoli = ''
count = 0
k = 10
for line in file:
    count += 1
    if (count > 1): # the first line contains the non-sequence header
so we discard it
```

```
        ecoli += line.replace("\n", "") # we string the newline
character from the end of each line

# Creating a sort list of all k-mers in the genome
kmers = [ecoli[i:i+k] for i in range(len(ecoli)-k+1)]
kmers.sort()
# or we can create a set of unique kmers instead
setkmers = set(kmers)
setkmers = list(setkmers)
```

This may appear, at first, as unnecessary and basically an overkill that takes up too much time and memory. But sometimes, this sort of data transformations are beneficial because they speed up downstream processess that are expected to be performed many times.

We next, want to try to search this list for our desired k-mer.

Take a pause and think! Does this remind you of the way we look for things in a certain context.

```
## Dichotomous Search for k-mers

# Using time to measure time of execution
import time
start_time = time.time()

# Pattern search

pattern = 'AGTTAGGCCT'
#pattern = 'TCGGCATCAG'
matches = 0

# Depending on what we want to search for
list_of_kmers = kmers # full list
#list_of_kmers = setkmers # set only

iter = 0
min = 1
max = len(list_of_kmers)

midpoint = int((max+min)/2)

import math

while iter <= math.log2(len(list_of_kmers)):
    iter += 1
    if (pattern == list_of_kmers[midpoint]):
        matches = list_of_kmers.count(list_of_kmers[midpoint])
        print("Pattern matched ", matches, " times")
        break
```

```
    if (pattern > list_of_kmers[midpoint]):
        min = midpoint
        midpoint = int((max+min)/2)
    if (pattern < list_of_kmers[midpoint]):
        max = midpoint
        midpoint = int((max+min)/2)
if (matches == 0):
    print("No matches found")

print("--- %s seconds ---" % (time.time() - start_time))
```

```
No matches found
--- 0.0009846687316894531 seconds ---
```

This approach, which is called **dichotomous search**, proceeds by splitting the ordered space of kmers in two equal parts, depending on whether the searched string is alphabetically before or after a fixed midpoint.

You see that it is somewhat faster than the exhaustive search. Of course there are things we can do to speed up both processes (can you think of which?) and we have to keep in mind that the dichotomous search in this case, works on a precalculated ordered list that has taken more time to be created.

The point that we want to make here is that there is a big difference in the two approaches, which guarantees that even for the worst case scenario (especially for the worst case scenario), the dichotomous search will be much faster.

Take a pause and think! What makes the dichotomous search faster?

3. Iteration vs Recursion

Case 1: The Largest Common Divisor Problem

- Given two integer numbers *a* and *b*
- Find an integer *lcd* that:
  - divides both *a* and *b* with 0 remainder
  - is the largest possible number

A solution: Euclid's Algorithm for LCD

- Euclid is said to have proposed an elegant solution.
- The basis of the solution is both *a* and *b* should be able to be represented as products of *lcd*
- In this sense, the best case scenario for *lcd* is that *mod(a/b)=0, lcd=a/b*, which means that the smaller of the two numbers is actually the *lcd*
- If this is not the case and *mod(a/b)=c* then *lcd* should be smaller or equal to the remainder of the division. The problem now is to find the *lcd* of the remainder *c* and the smaller number *b*. It is basically **the same problem**.
- Through a repetitive process in which *a, b* are substituted by *b, mod(a/b)* in each step we stop when *mod(a/b)=0*, declaring *b* as the *lcd*.

Euclid's Algorithm (Process)

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 as the LCD of a and b.

Let's make it more formal. Pseudocode

```
Input: A, B

    # C=remainder(A/B)

      if (C is greater than 0) {B->A; C->B; goto #}
```

```
      if (C equals 0) {print LCD=C; end}
```

LCD Take#1 (with Iteration)

```
def simple_euclid(a,b):
  while (b > 0): # as long as the smallest of the two (or the
remainder) is not zero
    a, b = b, a%b # switch a and b to b and mod(a/b)
  print(a) # print the last a (since b is now 0)

x = 1920
y = 1080
simple_euclid(x, y)
```

```
120
```

What does it do?

The solution above goes through a simple (but clever) iteration just as Euclid suggested

```
def simple_euclid(a,b):
     # enters an iterative process if b > 0
     while (b > 0): # Checks if the smallest of the numbers is > 0
               # it's basically the remainder of the division
                    # # swaps a and b with b and the remainder of the
division
                    a, b = b, a%b
           # Calculates the division
           # Extracts the remainder
           # Makes the swap
     return(a) # returns the result as the last divisor that (gave 0
remainder)

x = 1920
y = 1080

simple_euclid(x, y)
```

```
120
```

LCD Take#2 (with Recursion)

```
def rec_euclid(a,b):
    if a % b == 0:
        return b
    else:
        return rec_euclid(b, a%b)

x = 1920
y = 1080
rec_euclid(x, y)
```

What does it do?

- The most interesting part is the place where the function calls itself

```
# else:
#   rec_euclid(b, a%b)
```

- This is a nice example of **recursion**, a process through which we take advantage of our algorithmic process by **calling it** from within itself.
- While very useful it is not always the best way to proceed (as for instance in this case)

Differences of Iteration vs Recursion

- In iteration control is performed by the value of b, while in recursion it is a more general control statement *(mod(a/b) == 0)*
- Infinite iteration means waiting forever but infinite recursion means trouble, so **be careful!**
- In many cases (not this one) recursion appears to be more elegant, however iteration is always simpler, easier to follow and with smaller burden on the system

Case 2: Sorting a series of integers

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

Sorting Take #1: Simple Sort (Pseudocode)

```
Input N[i] list of numbers
Output S[i] list of sorted N

while(there is a list)
for i in 1:l # l is the size of the list
  minimum<-N[i] # assign an initial minimum value
    for j in 1:l # loop over all elements
      if (minimum >= N[j]) # check for minimum constraint
        minimum <- N[j]
      else
        continue
    remove minimum from N
    append minimum to S
return(S)
```

Simple Sort (Iterative)

```
def SimpleSort(N):

    i=0
    S=[]
    minind=0
    while (i < len(N)):
        minimum = N[i]
        j = i
```

```
        while (j < len(N)):
            if minimum >= N[j]:
                minimum = N[j]
                minind = j
                j = j + 1
            else:
                j = j + 1
        S.append(N[minind])
        N.remove(N[minind])
    return(S)

numbers = [14,7,3,12,9,11,6,2]

SimpleSort(numbers)
```

```
[2, 3, 6, 7, 9, 11, 12, 14]
```

Now let's try with a greater set and record how long it takes to sort them out

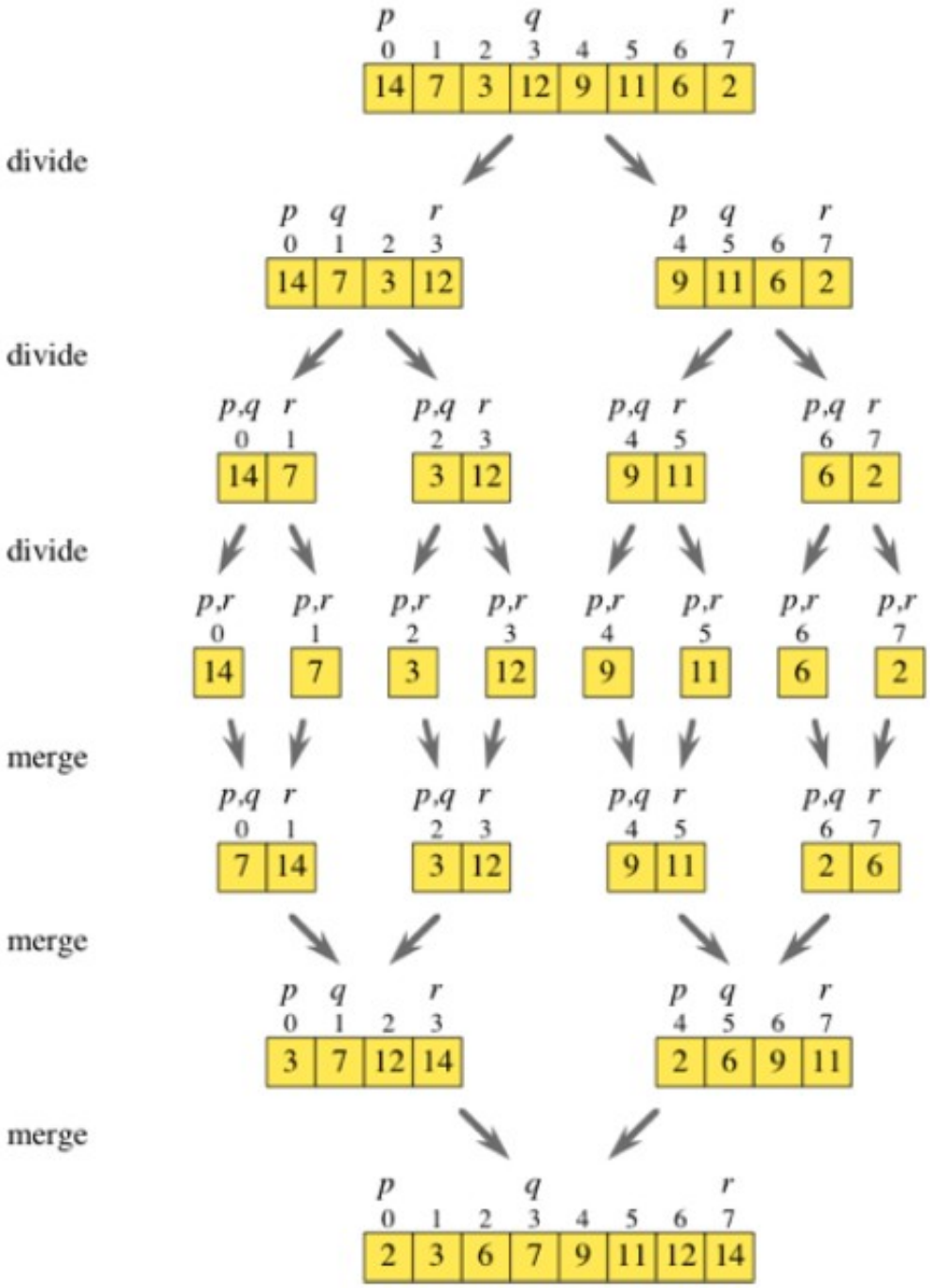
```
import random

N = 100000
numbers = []

for i in range(N):
    numbers.append(random.randint(0,N*10))

import time
start_time = time.time()

# Simple Sort

sortedNs = SimpleSort(numbers)

print("--- %s seconds ---" % (time.time() - start_time))
```

```
---------------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
<ipython-input-26-92fdcc4db0b2> in <module>
      4 # Simple Sort
      5
----> 6 sortedNs = SimpleSort(numbers)
      7
      8 print("--- %s seconds ---" % (time.time() - start_time))

<ipython-input-16-e5ee81518c18> in SimpleSort(N)
      8         j = i
      9         while (j < len(N)):
---> 10             if minimum >= N[j]:
```

```
    11                  minimum = N[j]
    12                  minind = j

KeyboardInterrupt:
```

Sorting Take #2: MergeSort

- Instead of taking each element and checking if it is the smallest in a list of gradually decreasing length, MergeSort implements a different (and faster strategy)
- It starts by **dividing** the list of numbers into two sublists and trying to sort these smaller lists before joining them back to the full list.
- You can imagine consecutive splits that come down to sublists of (N=2) in which the sorting is trivial: We basically need to check which is the greater of two numbers.
- The key in the process is to carefully implement the consecutive splits and then merging of the sublists (which gives the algorithm its name)
- This is done with a clever use of **recursion**.



MergeSort. A case of Recursion

Pseudocode (Recursion) The pseudocode below shows how the recursive sorting is done

```
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.B[j] # add B[j] to a list C[N]
          remove B[j]
```

MergeSort

MergeSort can be broken down into two parts. We first need to describe a function that will merge the sorted sublists. We call this **merge()**.

The main part, though, is a recursive call to a sort function, which we call **mergesort()**. This recursive call splits lists in ever-shorter sublists down to the rudimentary size of 2 and uses the merge() function to sort pairs, then quadruples, then octets etc.

```
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

def mergesort(x):
    # Function to sort an array using merge sort algorithm
    if len(x) == 0 or len(x) == 1:
        return x
    else:
        # print(len(x)) # use this line to get an idea how this works
        middle = int(len(x)/2)
        a = mergesort(x[:middle])
        b = mergesort(x[middle:])
        return merge(a,b)

import time
start_time = time.time()

# MergeSort

sortedNs = mergesort(numbers)
```

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 14.727187633514404 seconds ---
```

Evaluating the Complexity

Take a pause and think:

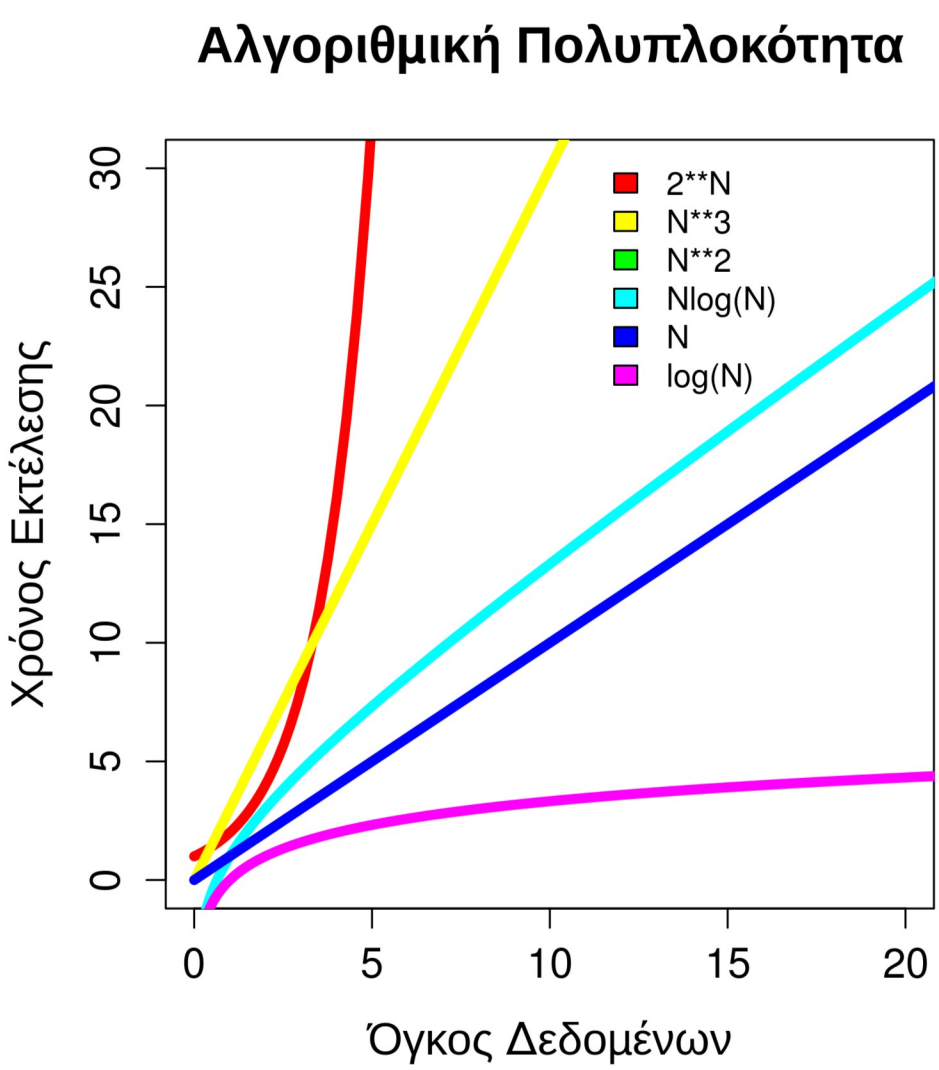
- How many calculations did SimpleSort require?
- How does the number of calculations scale with the size of the list N?
  - For each element in the list we need N-1 comparisons
  - We then shorten the list by -1 and repeat
  - We thus need (N-1)+(N-2)+(N-3)...+1 calculations
  - This is the sum of a series with period=1 and is equal to:((N-1)(N-2))/2
- This means that as N increases, the number of calculations increases by N^2.

Brute Force Approaches

- SimpleSort belongs to a type of algorithms that are called "Exhaustive" or "Brute Force".
- This means that they proceed with a simple "all out" approach that attacks the problem directly, in the hope that it is not too complex and expecting that the mere "force" of computation can solve it.
- Brute Force approaches work in a satisfactory way if the problem is not very complex.
- However they can be problematic if the problem is not simple as we will see in the following.

Big-O, O() notation

- O()-notation is a means to express the complexity of an algorithm, in particular the way with which the number of calculation increases with the size of the input
- O() is shown as a function of input size (n) depending on the way processing time scales with n.
- For example SimpleSort is O(n^2) because as we saw it scales with n-quadratic



Big-O Notation (Merge Sort)

- **MergeSort**: 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 **MergeSort is O(nlogn)**.

- Question: Which is faster? SimpleSort or MergeSort?
- Question: Can you figure out the time complexity of Euclid's algorithm?

Divide and Conquer approaches

- MergeSort belongs to a different type of approaches that are called "Divide and Conquer"

- Divide and Conquer approaches proceed (as their name implies) by dividing a complex problem into simpler subproblems and solving them either recursively or iteratively (usually the first)
- In many cases they are the only way to go around difficult problems that would require a prohibitive amount of calculations using Brute Force.

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[0]=0
N[1]=1
N[2]=N[0]+N[1]=0+1=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
```

Fibonacci Sequence

The problem: Calculate the Fibonacci element number N

Fibonnaci Take #1: Using an Array

```
def SimpleFibonacci(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]

import time
start_time = time.time()
N = 40
myFib = SimpleFibonacci(N)
print(myFib)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
102334155
--- 0.0004940032958984375 seconds ---
```

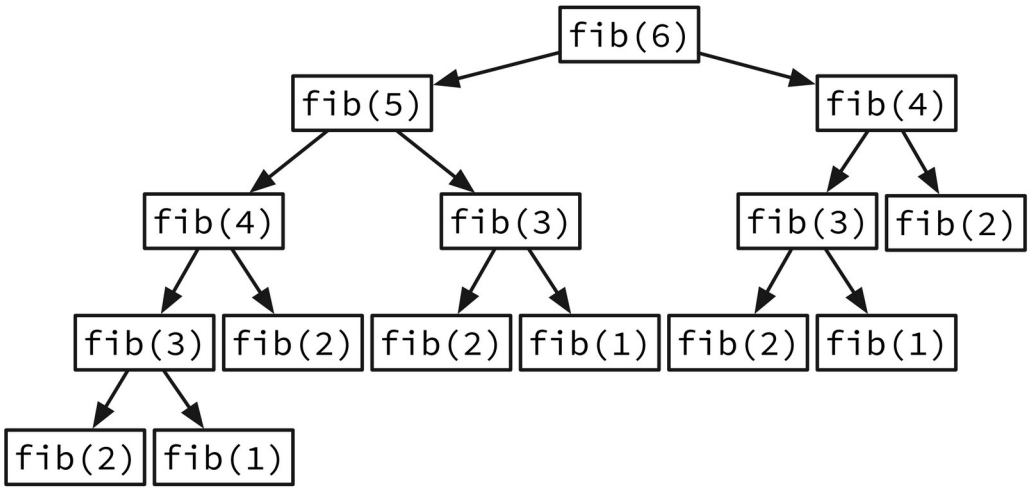
Fibonacci Take #2: Using Recursion

```
def RecursiveFibonacci(N):
    if N in {0,1}:
        return N
    if N > 1:
        return RecursiveFibonacci(N-2) + RecursiveFibonacci(N-1)

import time
start_time = time.time()
N = 40
```

```
myFib = RecursiveFibonacci(N)
print(myFib)
print("--- %s seconds ---" % (time.time() - start_time))

102334155
--- 85.16786909103394 seconds ---
```

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
4. What do you think about recursion now?

Other cases of algorithms

1. Randomized algorithmic approaches
2. Using data structures

4. Randomized approaches

Sometimes using a random approach to solve a problem is much more efficient than we may think. A nice example to demonstrate this is the calculation of π with a randomized algorithm.

Imagine the following "experiment":

- Imagine a square with a side of length $\alpha=1$ and
- A circle of radius $\alpha/2$ that is inscribed inside the square
- Now consider a random sample of points x,y from the square
- The proportion of points that fall within the circle is equal to the ratio of the two areas R which is a function of π.
- Design a randomized process that will calculate π

```
import random
import math
success=0
tries=10000000
for i in range(tries):
    # drawing point
    (x,y)=(random.random(), random.random())
    # checking if x,y are part of the circle
    d=math.sqrt(abs(x-0.5)**2+abs(y-0.5)**2)
    if (d <= 0.5):
        success=success+1
print(4*success/tries)
```

```
3.1416952
```

5. Data structures

Data structures (tables, networks, trees etc) go beyond simple entities in programming. The choice of the correct data structure can be crucial into solving a problem. A nice example may be found in simple question regarding multiplicity. For instance consider the following question:

- Given a set of dates corresponding to the birthdays of all employees in a company
- Find the date on which most birthdays occur with the fastest possible way

```
import datetime
import random

employees=5000
random_dates=[]

start_date = datetime.date(2020, 1, 1)
end_date = datetime.date(2020, 12, 31)

for i in range(employees):
    time_between_dates = end_date - start_date
    days_between_dates = time_between_dates.days
    random_number_of_days = random.randrange(days_between_dates)
    random_date = start_date +
datetime.timedelta(days=random_number_of_days)
    random_dates.append(random_date.strftime("%b %d"))
```

```
bday={}
for day in random_dates:
    bday[day]=random_dates.count(day)

def get_key(dictionary, val):
    for key, value in dictionary.items():
         if val == value:
            return key
    return "key doesn't exist"

mostbdays=max(bday.values())
print("The date with most birthdays is:", get_key(bday, mostbdays),
".\n", mostbdays, "employees have their bday.")
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-38-3a480e111ecb> in <module>
     25     return "key doesn't exist"
     26
---> 27 mostbdays=max(bday.values())
     28 print("The date with most birthdays is:", get_key(bday,
mostbdays), ".\n", mostbdays, "employees have their bday.")
     29

TypeError: 'int' object is not callable
```

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 (coming up soon)

- Given a long DNA sequence can you locate a given string of characters within it.
- Can you say how many times a subsequence is found in a sequence 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?

Exercises

1. Implement a strategy to search for all possible 10-mers in the genome of E.coli and report the ones that are not found in the genome. You can download the E. coli genome from this link.

2. Implement the dichotomous search in the genome of E. coli for any given 10-mer, this time with the additional function of reporting the positions of the found elements.

3. Test both SimpleSort and MergeSort with a increasing sizes of integer numbers, record the time of execution and plot their behaviour on a graph to see differences in their complexity.