CEITEC

Central European Institute of Technology
BRNO | CZECH REPUBLIC

Introduction to Bioinformatics (LF:DSIB01)

Week 1 : Introduction; Algorithm Basics

Panos Alexiou
panagiotis.alexiou@ceitec.muni.cz

Introduction to LF:DSIB01 - Course Goals

- Introductory course for Bioinformatics

- Students will:
  - become familiar with fundamental concepts
  - able to design, implement, and run basic analyses
  - decipher domain specific language in publications
  - understand the field and opportunities for development of skills

- Not goals:
  - solve specific research questions
  - cover entirety of Bionformatics field
  - teach programming skills

Introduction to LF:DSIB01 - Course Material

- Lecture Notes

- Useful Books
  - An Introduction to Bioinformatics Algorithms, Jones and Pevzner
  - Bioinformatics for Biologists, Pevzner and Shamir

Course Schedule

1. Introduction; Algorithm Basics
2. Sequence analysis introduction/common file formats
3. Sequence Alignment / Sequence pattern recognition 1/3
4. Sequence Alignment / Sequence pattern recognition 2/3
5. Sequence Alignment / Sequence pattern recognition 3/3
6. NGS / galaxy

---

7. Introduction to Data Analysis
8. Principles of Data visualisation
9. Clustering / PCA
10. Basic Statistics
11. Bayesian Inference/Bayesian classifier

---

12. Current Developments in Machine Learning
13. Colloquium Evaluation

Introduction to LF:DSIB01 - Student Responsibilities

- Attend Classes and Practicals
- Complete Practical Exercises
- Demonstrate Understanding of Material


And now for something completely different

Basics of Algorithms

- Definition of an algorithm
- Pseudocode Notation
- Exercise: The Coin Change Problem
- Brute force, Iterative, Recursive
- Big-O notation

What is an algorithm

- A sequence of instructions one must perform to solve a well formulated problem
- A step-by-step method of solving a problem
- A set of instructions designed to perform a specific task

Sequence of instructions Step-by-step method Set of instructions	Solve Perform	Well formulated problem Specific task

Sequence of instructions
Step-by-step method
Set of instructions

Solve
Perform

Well formulated problem
Specific task

MAKEPUMPKINPIE

$1\frac{1}{2}$ cups canned or cooked pumpkin
1 cup brown sugar, firmly packed
$\frac{1}{2}$ teaspoon salt
2 teaspoons cinnamon
1 teaspoon ginger
2 tablespoons molasses
3 eggs, slightly beaten
12 ounce can of evaporated milk
1 unbaked pie crust

Combine pumpkin, sugar, salt, ginger, cinnamon, and molasses. Add eggs and milk and mix thoroughly. Pour into unbaked pie crust and bake in hot oven (425 degrees Fahrenheit) for 40 to 45 minutes, or until knife inserted comes out clean.

Pseudocode Notation

```
MakePumpkinPie(pumpkin, sugar, salt, spices, eggs, milk, crust)
    PreheatOven(425)
    filling ← MixFilling(pumpkin, sugar, salt, spices, eggs, milk)
    pie ← Assemble(crust, filling)
    while  knife inserted does not come out clean
        Bake(pie)
    output  "Pumpkin pie is complete"
    return pie
```

Pseudocode Notation

Assignment

Format: $a \leftarrow b$

Effect: Sets the variable a to the value b.

Example: $b \leftarrow 2$
$a \leftarrow b$

Result: The value of a is 2

Pseudocode Notation

Conditional

Format:

```
if A is true
        B
else
        C
```

Effect: If statement A is true, executes instructions **B**, otherwise executes instructions **C**. Sometimes we will omit "**else C**," in which case this will either execute **B** or not, depending on whether A is true.

Example: $\textsc{Max}(a, b)$

```
if a < b
        return b
else
        return a
```

Pseudocode Notation

`for` loops

Format: **for** $i \leftarrow a$ **to** b

$\quad\quad$ **B**

Effect: Sets i to a and executes instructions **B**. Sets i to $a+1$ and executes instructions **B** again. Repeats for $i = a+2, a+3, \ldots, b-1, b$.

Example: SUMINTEGERS(n)

```
1  sum ← 0
2  for i ← 1 to n
3        sum ← sum + i
4  return sum
```

Pseudocode Notation

`while` loops

Format: **while** A is true
B

Effect: Checks the condition A. If it is true, then executes instructions **B**. Checks A again; if it's true, it executes **B** again. Repeats until A is not true.

Example: $\textsc{AddUntil}(b)$

```
i ← 1
total ← i
while total ≤ b
    i ← i + 1
    total ← total + i
return i
```

Pseudocode Notation

Array access

Format: a_i

Effect: The ith number of array $\mathbf{a} = (a_1, \ldots a_i, \ldots a_n)$. For example, if $\mathbf{F} = (1, 1, 2, 3, 5, 8, 13)$, then $F_3 = 2$, and $F_4 = 3$.

Example: $\textsc{Fibonacci}(n)$

$F_1 \leftarrow 1$

$F_2 \leftarrow 1$

for $i \leftarrow 3$ **to** n

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

return F_n

Pseudocode vs Computer Code

If you were to build a machine that follows these instructions, you would need to make it specific to a particular kitchen and be tirelessly explicit in all the steps (e.g., how many times and how hard to stir the filling, with what kind of spoon, in what kind of bowl, etc.)

This is exactly the difference between pseudocode (the abstract sequence of steps to solve a well-formulated computational problem) and computer code (a set of detailed instructions that one particular computer will be able to perform).

Pseudocode Exercise: Coin Change (Euro coins)

Convert an amount of money into the fewest number of coins

Input: Amount of money (M)
Output: the smallest number of 50c (a), 20c (b), 10c (c), 5c (d), 2c (e) and 1c (f) such that 50a+20b+10c+5d+2e+1f = M

```
while M > 0
    c ← Largest coin that is smaller than (or equal to) M
    Give coin with denomination c to customer
    M ← M − c
```

Try: M=60c; M=55c; M=40c

Pseudocode Exercise: Coin Change (Generalised)

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value ($c_1 > c_2 > \cdots > c_d$).

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

Pseudocode Exercise: Coin Change (US coins)

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value $(c_1 > c_2 > \cdots > c_d)$.

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

```
BetterChange(M, c, d)
r ← M
for k ← 1 to d
        i_k ← r / c_k
        r ← r − c_k · i_k
return (i_1, i_2, ..., i_d)
```

NB: Division = “floor”

Try
M = 40; c1=25, c2=10, c3=5, c4=1

Pseudocode Exercise: Coin Change (US coins)

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value $(c_1 > c_2 > \cdots > c_d)$.

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

BetterChange$(M, \mathbf{c}, d)$

$r \leftarrow M$
for $k \leftarrow 1$ **to** d
    $i_k \leftarrow r / c_k$
    $r \leftarrow r - c_k \cdot i_k$
return $(i_1, i_2, \ldots, i_d)$

NB: Division = "floor"

M = 40; c1=25, **c2=20**, c3=10, c4=5, c5=1

Discontinued 1875 for being too confusing

Pseudocode Exercise: Coin Change (US coins)

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value ($c_1 > c_2 > \cdots > c_d$).

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

```
BetterChange(M, c, d)
  r ← M
  for k ← 1 to d
      i_k ← r/c_k
      r ← r − c_k · i_k
  return (i_1, i_2, ..., i_d)
```

NB: Division = "floor"

BetterChange
40=
1x25 + 1x10 + 1x5=
3 coins

Incorrect!
40 = 2x20=
2 coins

M = 40; c1=25, **c2=20**, c3=10, c4=5, c5=1

Discontinued
1875
for being too
confusing

Pseudocode Exercise: Coin Change

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value ($c_1 > c_2 > \cdots > c_d$).

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

Tries every combination
Guaranteed to find optimal
Slow

Input: An amount of money M, and an array of d denominations $\mathbf{c} = (c_1, c_2, \ldots, c_d)$, in decreasing order of value $(c_1 > c_2 > \cdots > c_d)$.

Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

Brute Force Algorithm

BruteForceChange$(M, \mathbf{c}, d)$

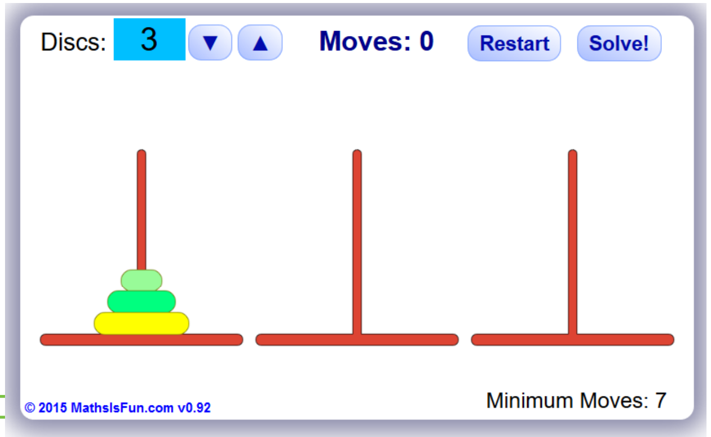
$\quad smallestNumberOfCoins \leftarrow \infty$
$\quad$ **for each** $(i_1, \ldots, i_d)$ **from** $(0, \ldots, 0)$ **to** $(M/c_1, \ldots, M/c_d)$
$\quad\quad valueOfCoins \leftarrow \sum_{k=1}^{d} i_k c_k$
$\quad\quad$ **if** $valueOfCoins = M$
$\quad\quad\quad numberOfCoins \leftarrow \sum_{k=1}^{d} i_k$
$\quad\quad\quad$ **if** $numberOfCoins < smallestNumberOfCoins$
$\quad\quad\quad\quad smallestNumberOfCoins \leftarrow numberOfCoins$
$\quad\quad\quad\quad \mathbf{bestChange} \leftarrow (i_1, i_2, \ldots, i_d)$
$\quad$ **return** $(\mathbf{bestChange})$

Tries every combination
Guaranteed to find optimal
Slow

Spoiler: (There is a better solution: Stay tuned for Week 4)

Recursive Algorithms

The *Towers of Hanoi* puzzle, introduced in 1883 by a French mathematician, consists of three pegs, which we label from left to right as 1, 2, and 3, and a number of disks of decreasing radius, each with a hole in the center. The disks are initially stacked on the left peg (peg 1) so that smaller disks are on top of larger ones. The game is played by moving one disk at a time between pegs. You are only allowed to place smaller disks on top of larger ones, and any disk may go onto an empty peg. The puzzle is solved when all of the disks have been moved from peg 1 to peg 3.



1 disc = 1 move
2 discs = 3 moves (1-2, 1-3, 2-3)
3 discs = 7 moves (1-3, 1-2, 3-2, 1-3, 2-1, 2-3,1-3)
…

https://www.mathsisfun.com/games/towerofhanoi.html

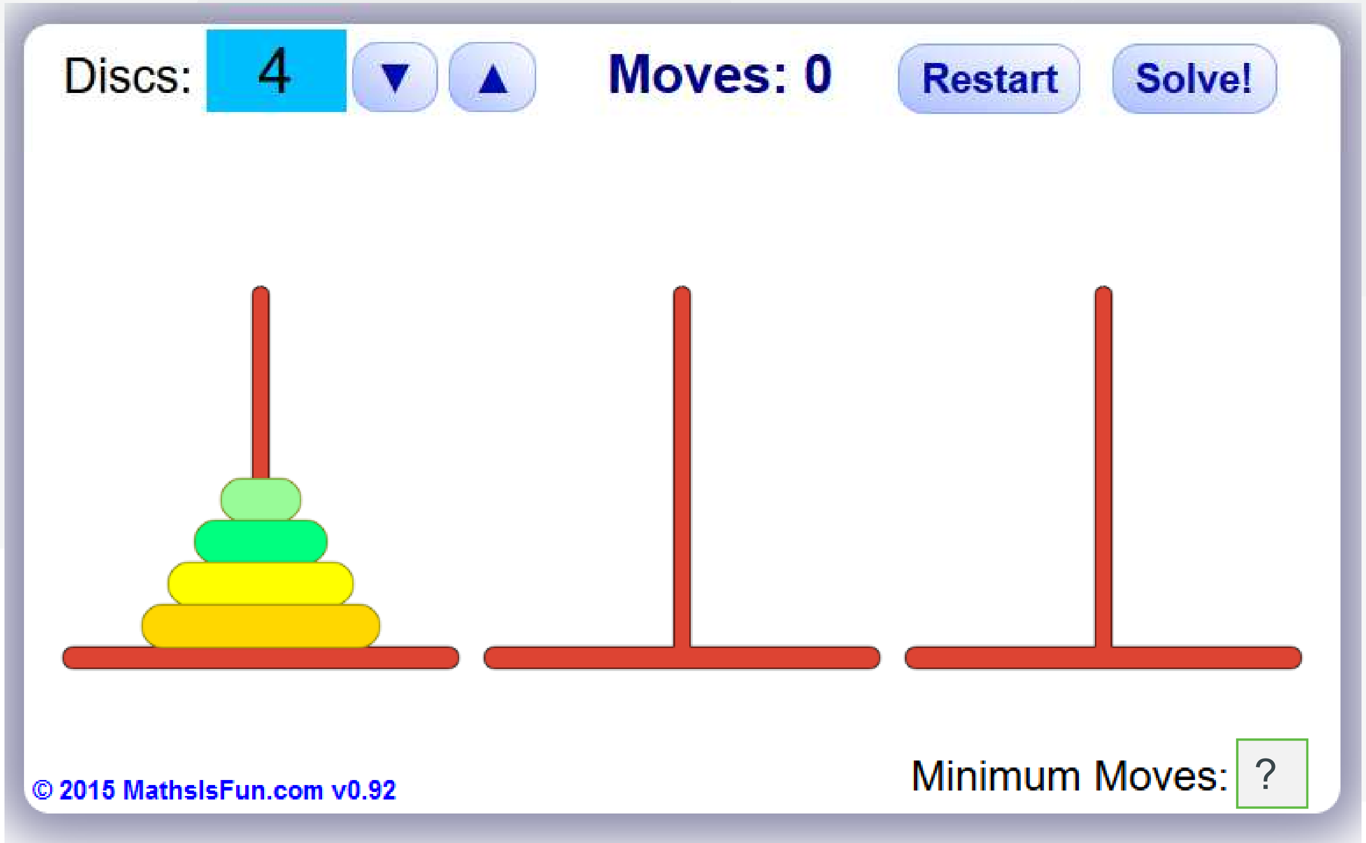
Towers of Hanoi (3 disks)

7 moves (1-3, 1-2, 3-2, 1-3, 2-1, 2-3,1-3)

More generally, to move a stack of size **n** from the **left** to the **right** peg, you first need to move a stack of size **n − 1** from the **left** to the **middle** peg, and then from the middle peg to the right peg once you have moved the nth disk to the right peg.

To move a stack of size **n − 1** from the **middle** to the **right**, you first need to move a stack of size **n − 2** from the **middle** to the **left**, then move the (n − 1)th disk to the right, and then move the stack of n − 2 from the left to the right peg, and so on.

Towers of Hanoi: N disks



Towers of Hanoi: N disks

$fromPeg$	$toPeg$	$unusedPeg$
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

```
HanoiTowers(n, fromPeg, toPeg)
if n = 1
    output "Move disk from peg fromPeg to peg toPeg"
    return
unusedPeg ← 6 − fromPeg − toPeg
return
```

Towers of Hanoi: N disks

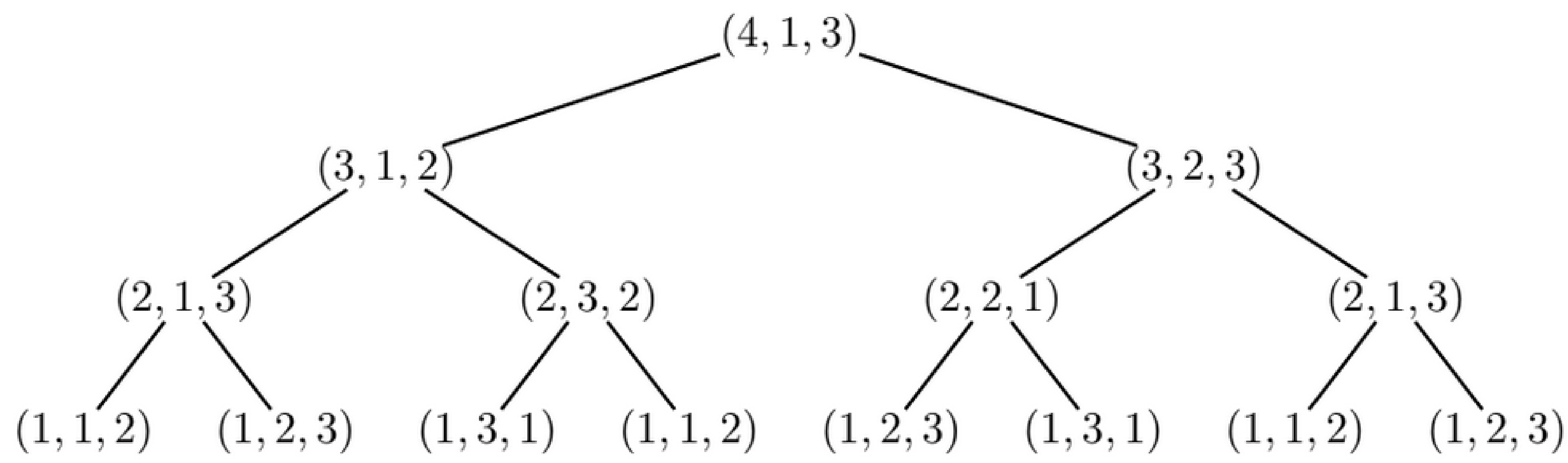
$fromPeg$	$toPeg$	$unusedPeg$
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

HANOITOWERS($n, fromPeg, toPeg$)

if $n = 1$
    output "Move disk from peg $fromPeg$ to peg $toPeg$"
    return
$unusedPeg \leftarrow 6 - fromPeg - toPeg$
HANOITOWERS($n - 1, fromPeg, unusedPeg$)
output "Move disk from peg $fromPeg$ to peg $toPeg$"
HANOITOWERS($n - 1, unusedPeg, toPeg$)
return

Towers of Hanoi: 4 disks

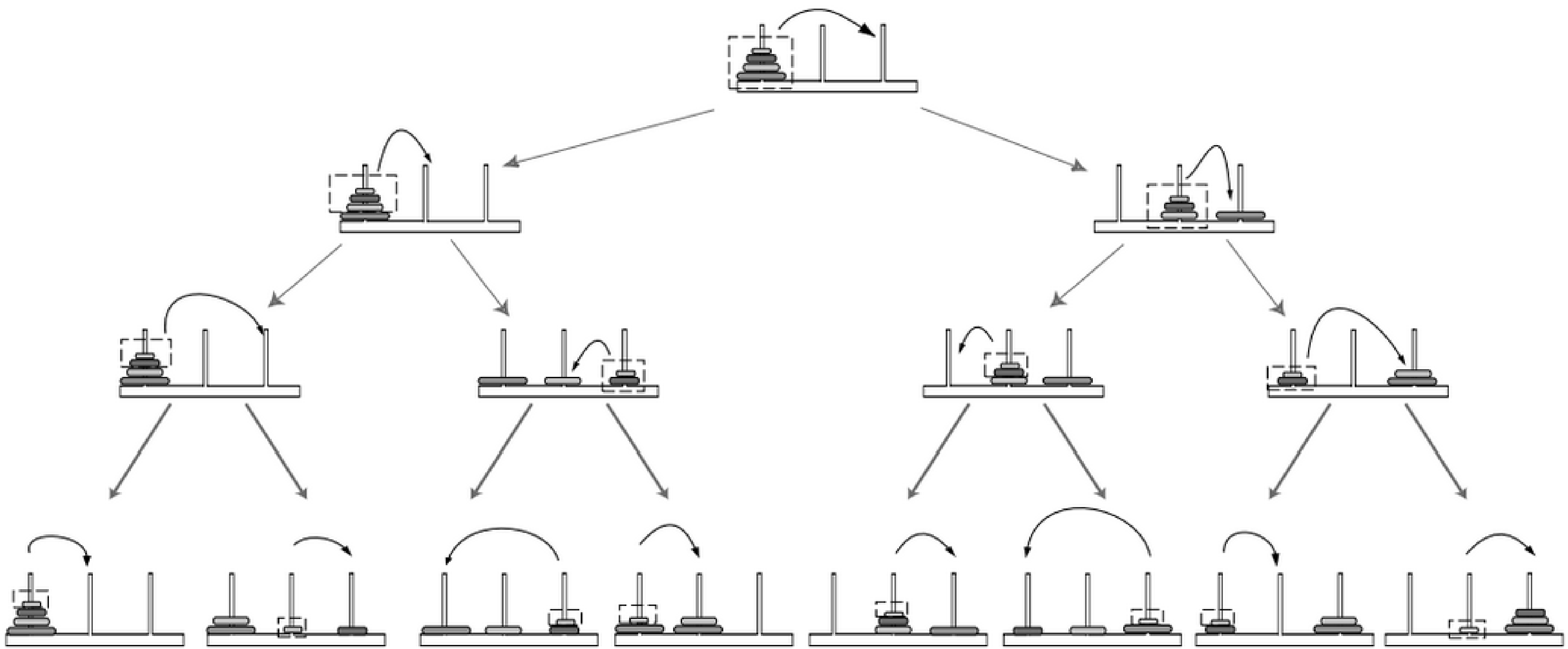
```
HANOITOWERS(n, fromPeg, toPeg)
if n = 1
    output "Move disk from peg fromPeg to peg toPeg"
    return
unusedPeg ← 6 − fromPeg − toPeg
HANOITOWERS(n − 1, fromPeg, unusedPeg)
output "Move disk from peg fromPeg to peg toPeg"
HANOITOWERS(n − 1, unusedPeg, toPeg)
return
```



https://www.mathsisfun.com/games/towerofhanoi.html

Towers of Hanoi: 4 disks

HANOITOWERS(n, $fromPeg$, $toPeg$)
if $n = 1$
  output "Move disk from peg $fromPeg$ to peg $toPeg$"
  return
$unusedPeg \leftarrow 6 - fromPeg - toPeg$
HANOITOWERS($n - 1$, $fromPeg$, $unusedPeg$)
output "Move disk from peg $fromPeg$ to peg $toPeg$"
HANOITOWERS($n - 1$, $unusedPeg$, $toPeg$)
return



https://www.mathsisfun.com/games/towerofhanoi.html

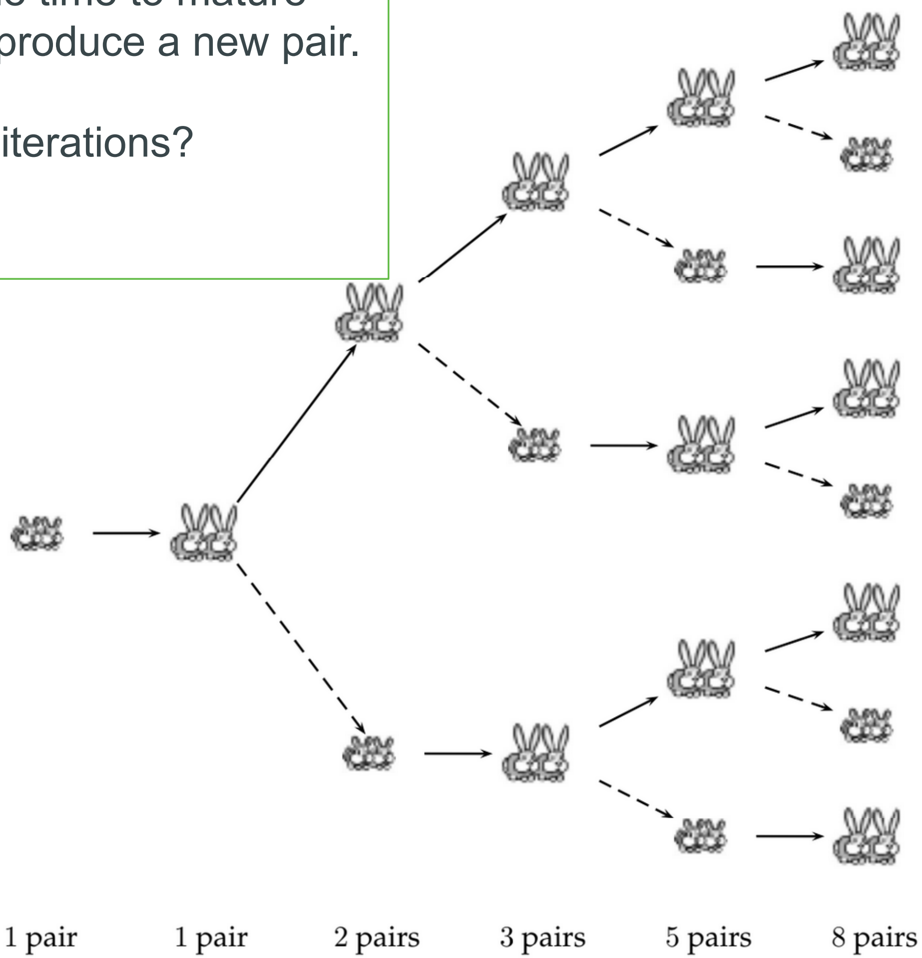
Iterative Algorithms – Fibonacci Series

Iterative Algorithms – Immortal Rabbits

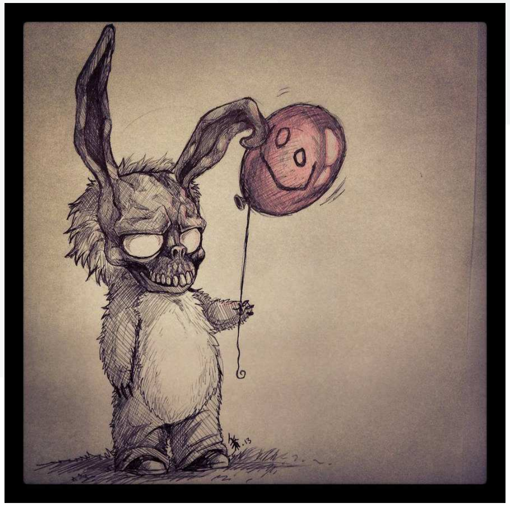
A baby pair of rabbits takes the same time to mature as a mature pair of rabbits takes to produce a new pair.

How many rabbits are there after N iterations?

PS: rabbits cannot die!



1,1,sum of previous two, …



Iterative Algorithms vs Recursive Algorithms

$\textsc{RecursiveFibonacci}(n)$

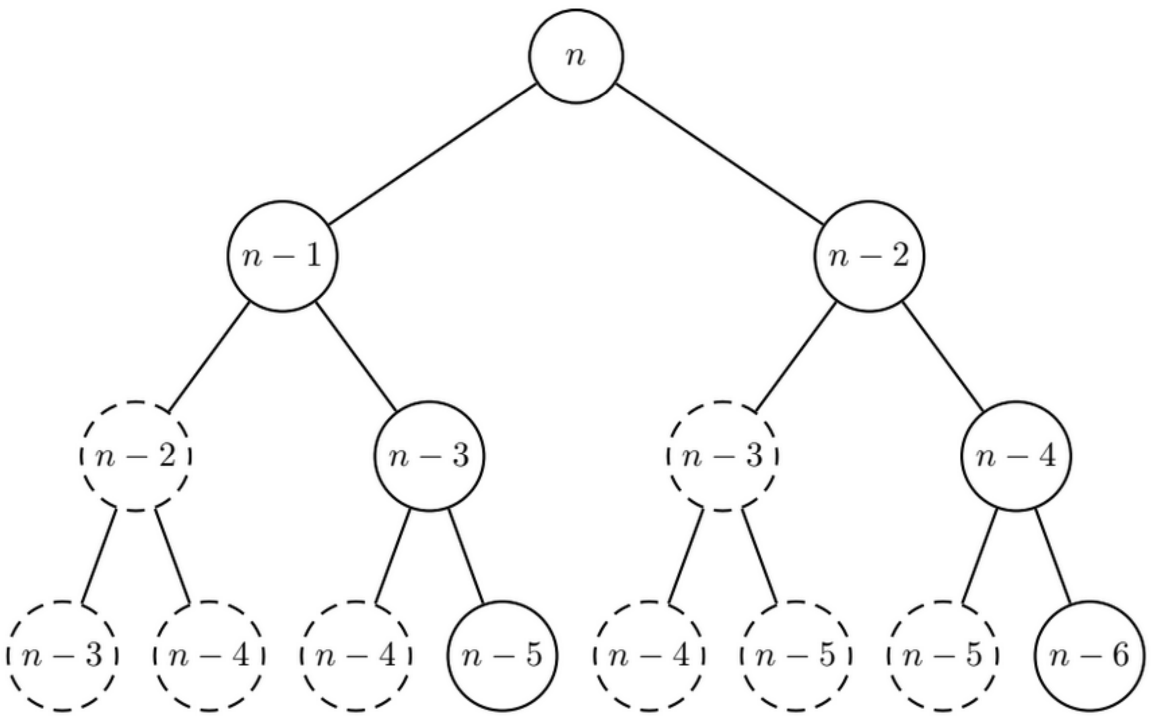
if $n = 1$ **or** $n = 2$
    return 1
else
    $a \leftarrow \textsc{RecursiveFibonacci}(n-1)$
    $b \leftarrow \textsc{RecursiveFibonacci}(n-2)$
    return $a + b$

Recursive: Slow (exponential)

$\textsc{Fibonacci}(n)$

$F_1 \leftarrow 1$
$F_2 \leftarrow 1$
for $i \leftarrow 3$ **to** n
    $F_i \leftarrow F_{i-1} + F_{i-2}$
return F_n

Iterative: Fast (linear)



RECURSIVEFIBONACCI(n)

if $n = 1$ **or** $n = 2$
  return 1
else
  $a \leftarrow$ RECURSIVEFIBONACCI($n-1$)
  $b \leftarrow$ RECURSIVEFIBONACCI($n-2$)
  return $a + b$

Recursive: Slow (exponential)

Algorithms

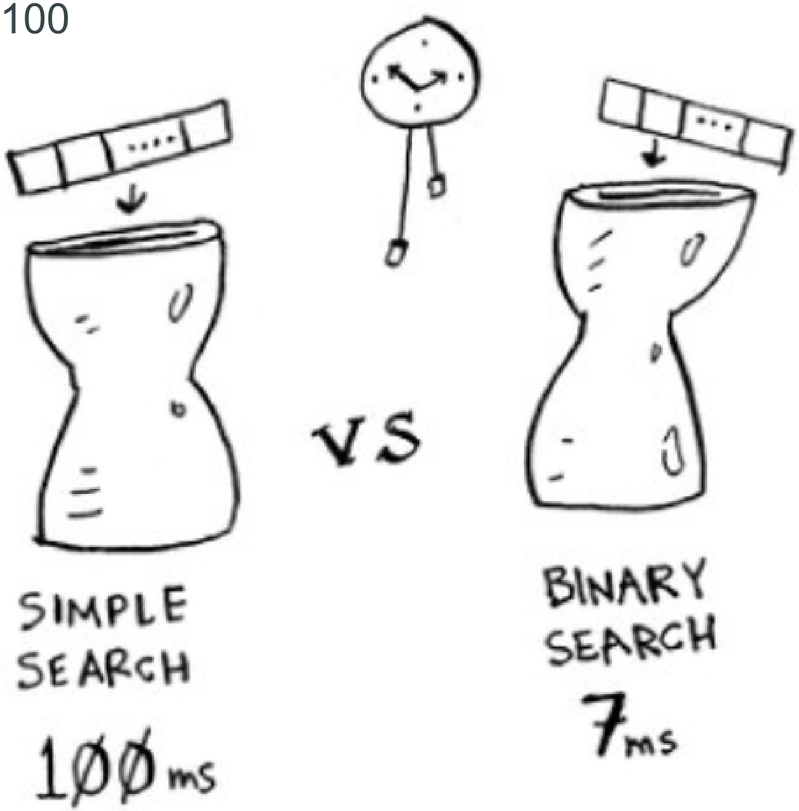
- Brute force : Try Everything, slow but always correct
- Recursive : To Solve for n, first solve for n-1
- Iterative : Loop on something, can be faster

Fast vs Slow algorithms

- How many operations does an algorithm take as N increases?
- Is the relationship linear? Quadratic? Exponential?
- What is the upper limit of the running time of an algorithm as N increases?

Guess random number (up/down)

1ms / check
N = 100



Simple search:
For i in 1 to N
If i == the number
  return i

Binary search:
Range-min = 1
Range-max = N
While ()
 i = middle number of range
  if i == the number; return I
  elsif i < number; Range-max=i;
  elsif i > number; Range-min=i;

Guess random number (up/down)

	SIMPLE SEARCH	BINARY SEARCH
100 ELEMENTS	100ms	7ms
10,000 ELEMENTS		
1,000,000,000 ELEMENTS		

~15 times faster

??? Guess ???

Guess random number (up/down)

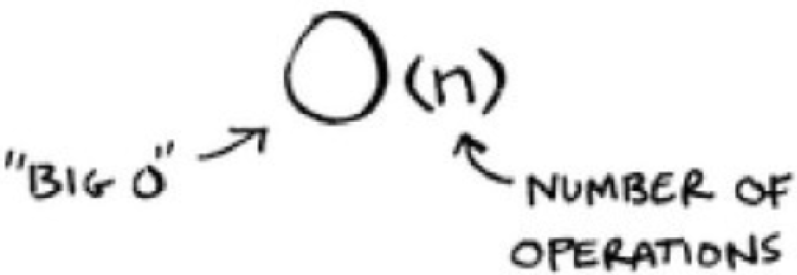
	SIMPLE SEARCH	BINARY SEARCH	
100 ELEMENTS	100ms	7ms	~15 times faster
10,000 ELEMENTS			
1,000,000,000 ELEMENTS	~450ms ?	32 ms	~15 times faster ?

Guess random number (up/down)

	SIMPLE SEARCH	BINARY SEARCH	
100 ELEMENTS	100 ms	7 ms	~15 times faster
10,000 ELEMENTS			
1,000,000,000 ELEMENTS	11 days	32 ms	~~~15 times faster~~ doesn’t make sense!

Guess random number (up/down)

	SIMPLE SEARCH	BINARY SEARCH
100 ELEMENTS	100 ms	7 ms
10,000 ELEMENTS	10 seconds	14 ms
1,000,000,000 ELEMENTS	11 days	32 ms
	Linear 1 ms/element O(n)	Logarithmic 1 ms/log2(element) O(log n)



(Worst case scenario)

A function $f(x)$ is "Big-O of $g(x)$", or $O(g(x))$, when $f(x)$ is less than or equal to $g(x)$ to within some constant multiple c. If there are a few points x such that $f(x)$ is not less than $c \cdot g(x)$, this does not affect our overall understanding of f's growth. Mathematically speaking, the Big-O notation deals with *asymptotic* behavior of a function as its input grows arbitrarily large, beyond some (arbitrary) value x_0.

Definition 2.1 *A function $f(x)$ is $O(g(x))$ if there are positive real constants c and x_0 such that $f(x) \leq cg(x)$ for all values of $x \geq x_0$.*

For example, the function $3x = O(.2x^2)$, but at $x = 1$, $3x > .2x^2$. However, for all $x > 15$, $.2x^2 > 3x$. Here, $x_0 = 15$ represents the point at which $3x$ is bounded above by $.2x^2$. Notice that this definition blurs the advantage gained by mere constants: $5x^2 = O(x^2)$, even though it would be wrong to say that $5x^2 \leq x^2$.

Like Big-O notation, which governs an upper bound on the growth of a function, we can define a relationship that reflects a lower bound on the growth of a function.

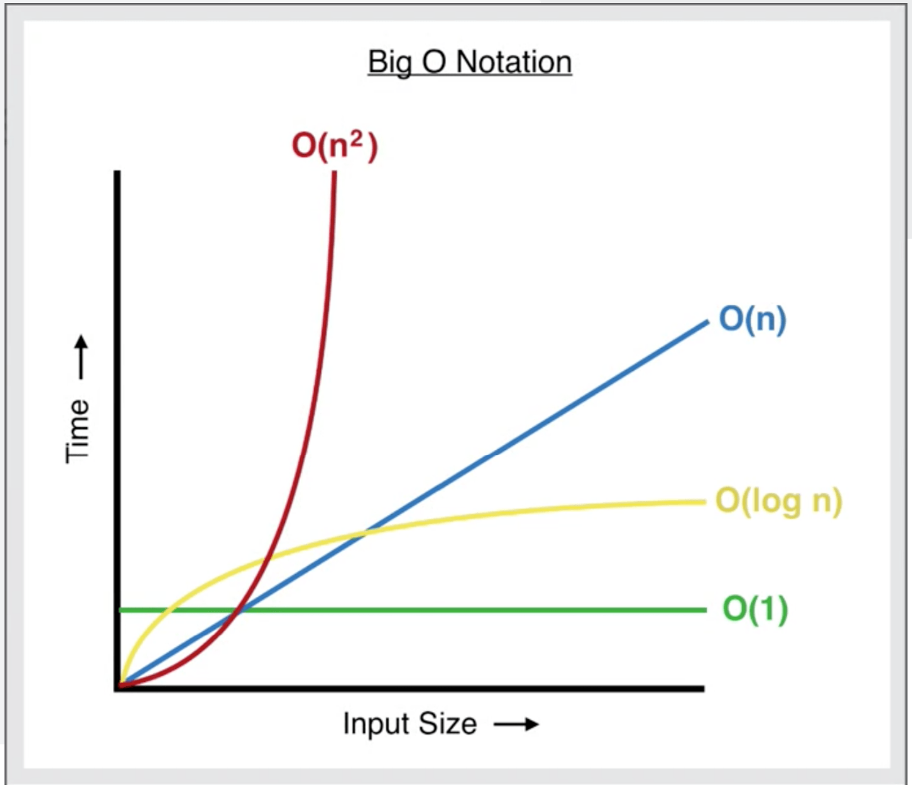
Definition 2.2 *A function $f(x)$ is $\Omega(g(x))$ if there are positive real constants c and x_0 such that $f(x) \geq cg(x)$ for all values of $x \geq x_0$.*

If $f(x) = \Omega(g(x))$, then f is said to grow "faster" than g.

Now, if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$ then we know very precisely how $f(x)$ grows with respect to $g(x)$. We call this the Θ relationship.

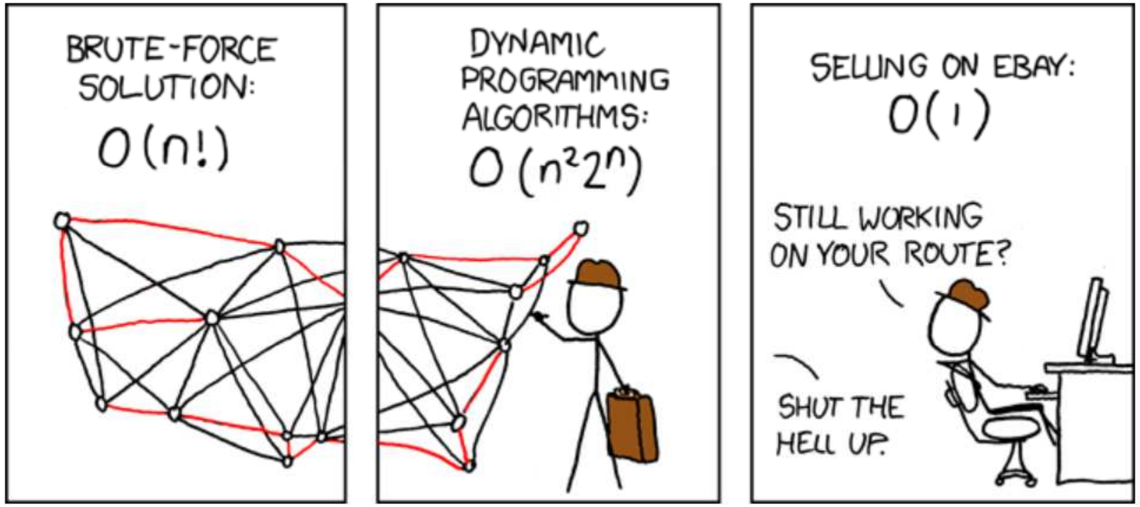
Definition 2.3 *A function $f(x)$ is $\Theta(g(x))$ if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.*

Common Big-Os



- O(log *n*), also known as *log time*. Example: Binary search.
- O(*n*), also known as *linear time*. Example: Simple search.
- O(*n* * log *n*). Example: A fast sorting algorithm, like quicksort.
- O(*n*2). Example: A slow sorting algorithm, like selection sort.
- O(*n*!). Example: A really slow algorithm, like the traveling salesperson.

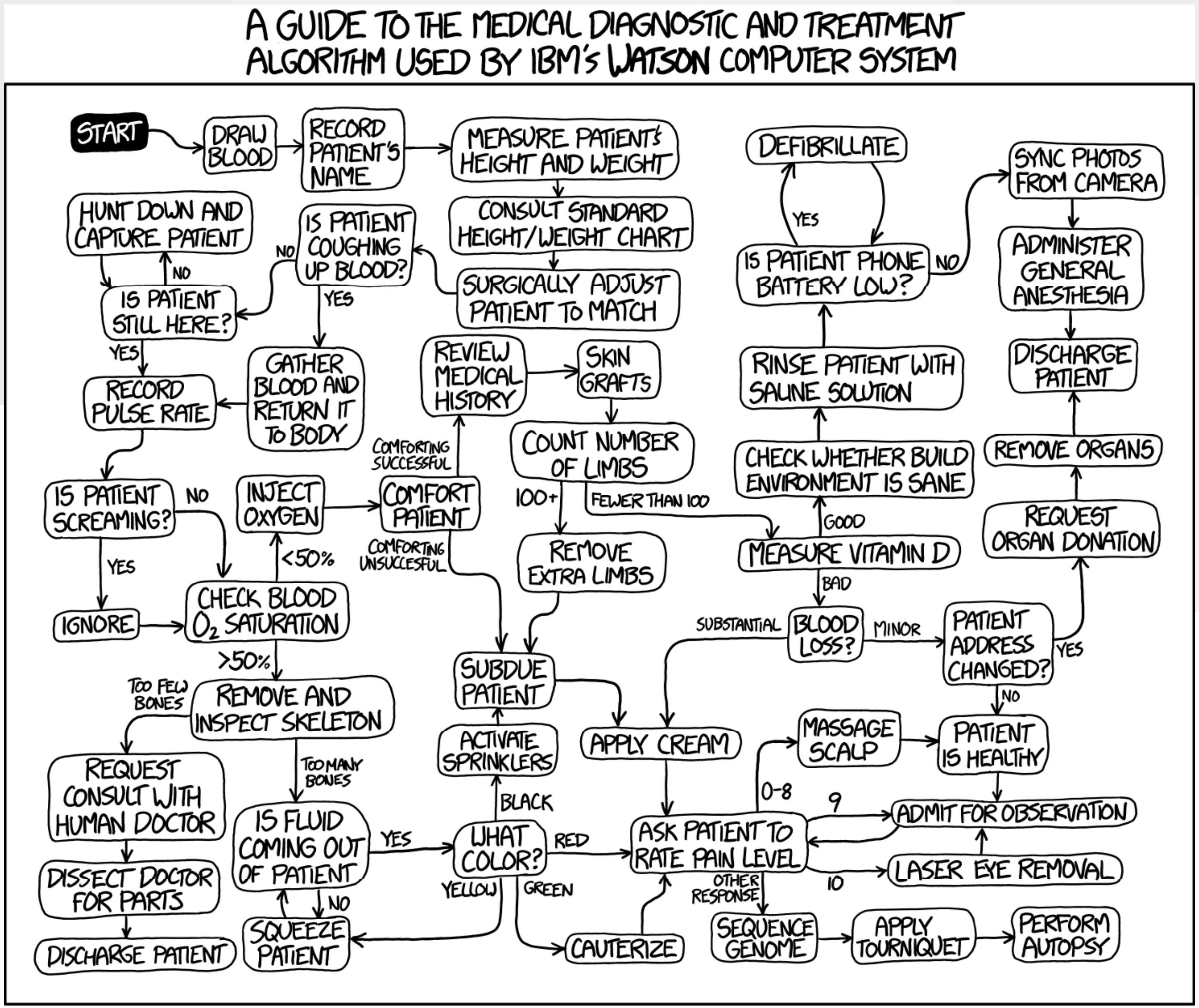
"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"



https://www.freecodecamp.org/news/big-o-notation-simply-explained-with-illustrations-and-video-87d5a71c0174/

Week 1 Summary

- I know what an algorithm is
- I can write pseudocode
- I understand
  - Brute force
  - Iterative
  - Recursive
- Big-O = how slow



CEITEC

@CEITEC_Brno

Thank you for your attention!
60 minutes lunch break.

Panos Alexiou
panagiotis.alexiou@ceitec.muni.cz

www.ceitec.eu