Algorithms and Data Structures in Biology

Algorithms and Their Complexity

Ugo Dal Lago

University of Bologna, Academic Year 2018/2019

Section 1

Defining The Algorithm

The Rock Pile Game

- Alice and Bob play a game, starting from two rock piles, each containing 10 rocks.
- In turn Alice and Bob either pick **one** rock from one of the two piles, or **two** rocks, one from each pile.
- *Who wins*? Whomever manage to remove *the last* pile.
- Alice starts.
- Is there a winning strategy?
- Bob realizes that if the rocks were just 2, he could easily win, independently on Alice's moves.
- But how about the general case?

The Rock Pile Game

- Alice and Bob play a game, starting from two rock piles, each containing 10 rocks.
- In turn Alice and Bob either pick **one** rock from one of the two piles, or **two** rocks, one from each pile.
- *Who wins*? Whomever manage to remove *the last* pile.
- Alice starts.
- Is there a winning strategy?
- Bob realizes that if the rocks were just 2, he could easily win, independently on Alice's moves.
- But how about the general case?

The Rock Pile Game

- Alice and Bob play a game, starting from two rock piles, each containing 10 rocks.
- In turn Alice and Bob either pick **one** rock from one of the two piles, or **two** rocks, one from each pile.
- *Who wins*? Whomever manage to remove *the last* pile.
- Alice starts.
- Is there a winning strategy?
- Bob realizes that if the rocks were just 2, he could easily win, independently on Alice's moves.
- But how about the general case?

	0	1	2	3	4	5	6	7	8	9	10
0	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
1	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
2	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
3	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
4	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
5	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
6	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
7	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
8	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$
9	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$	$\nwarrow$	$\uparrow$
10	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$	$\leftarrow$	$*$

But What if…

- What if the number of **rocks** we start from is higher than 10?
- And what if the number of **piles** is higher than 2?
- How could we determine the next move to make depending on the current state of the game (i.e., number of piles, number of rocks on each pile)?
- We are looking for an *effective strategy* for a combinatorial game. In other words, we are solving a particular kind a combinatorial problem.

But What if…

- What if the number of **rocks** we start from is higher than 10?
- And what if the number of **piles** is higher than 2?
- How could we determine the next move to make depending on the current state of the game (i.e., number of piles, number of rocks on each pile)?
- We are looking for an *effective strategy* for a combinatorial game. In other words, we are solving a particular kind a combinatorial problem.

Defining Combinatorial Problems

- A **combinatorial problem** is a *unambiguous* and *precise* problem concerning the production of some *outputs* from some *inputs*.
  - The *class* of possible input must be clearly specified.
  - *Which* output one gets from each input must itself be itself specified without any ambiguity.
  - Specifying *how* to obtain the output from the input is not part of the problem's definition.
- **Example**: The $n \times n$ rock pile problem
  - Input: n, and a state (m, k).
  - Output: a move that a player should make in (m, k) in order to win, *if possible*.

Defining Algorithms

- An **algorithm** is a sequence of instructions that one performs to *solve* a combinatorial problem.
- How should we *specify* an algorithm?
  - We could be *programming-language* dependent.
  - Or we could try to be *more abstract*.
- In this course, algorithms will be specified by way of **pseudocode**, namely by a notation which can be easily translated to concrete programming languages, including `Python`.
- We will not follow specific rules as for how pseudocode is specified. Rather, we will fit it to our needs whenever possible.
  - One should be **precise** without being **formal**.
  - The following basic requirements should be satisfied: *determinism*, *finiteness*, *unambiguity*.
  - There are certain constructions which are very common in pseudocode.

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

Arithmetic

Format: $a + b$, $a - b$, $a \cdot b$, a/b, a^b

Effect: Addition, subtraction, multiplication, division, and exponentiation of numbers.

Example: $\textsc{Dist}(x1, y1, x2, y2)$

$dx \leftarrow (x2 - x1)^2$

$dy \leftarrow (y2 - y1)^2$

return $\sqrt{(dx + dy)}$

Result: $\textsc{Dist}(x1, y1, x2, y2)$ computes the Euclidean distance between points with coordinates $(x1, y1)$ and $(x2, y2)$. $\textsc{Distance}(0, 0, 3, 4)$ returns 5.

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
```

Result: $\textsc{Max}(a, b)$ computes the maximum of the numbers a and b. For example, $\textsc{Max}(1, 99)$ returns 99.

`for` loops

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

$\quad\quad\mathbf{B}$

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

Example: SUMINTEGERS(n)

```
sum ← 0
for i ← 1 to n
    sum ← sum + i
return sum
```

Result: SUMINTEGERS(n) computes the sum of integers from 1 to n. SUMINTEGERS(10) returns $1 + 2 + \cdots + 10 = 55$.

`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: ADDUNTIL(b)

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

Result: ADDUNTIL(b) computes the smallest integer i such that $1 + 2 + \cdots + i$ is larger than b. For example, ADDUNTIL(25) returns 7, since $1+2+\cdots+7 = 28$, which is larger than 25, but $1+2+\cdots+6 = 21$, which is smaller than 25.

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: 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

Result: FIBONACCI(n) computes the nth Fibonacci number. FIBONACCI(8) returns 21.

United States Change Problem:
Convert some amount of money into the fewest number of coins.

Input: An amount of money, M, in cents.

Output: The smallest number of quarters q, dimes d, nickels n, and pennies p whose values add to M (i.e., $25q + 10d + 5n + p = M$ and $q + d + n + p$ is as small as possible).

United States Change Problem:

Convert some amount of money into the fewest number of coins.

Input: An amount of money, M, in cents.

Output: The smallest number of quarters q, dimes d, nickels n, and pennies p whose values add to M (i.e., $25q + 10d + 5n + p = M$ and $q + d + n + p$ is as small as possible).

```
USChange(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
```

Algorithm Correctness

- Are we sure that USCHANGE *indeed solves* the combinatorial problem it is supposed to solve, namely that it is **correct**?
- There are two ways one can use to convince herself of the correctness of an algorithm:
  1. **Testing** the algorithm.
     - Just check that the algorithm transforms inputs to outputs correctly.
     - This is an experimental methodology.
     - It is impossible to test an algorithm on *all* of the input instances.
  2. **Proving** the algorithm correct.
     - One needs to find a mathematical proof of the fact that the algorithm indeed does what it is supposed to do.
     - This is an analytical methodology.
     - Computer science has devised along the years so many methodology for proving algorithms correct.

Change Problem:
Convert some amount of money M into given denominations, using the smallest possible number of 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.

Change Problem:

Convert some amount of money M into given denominations, using the smallest possible number of 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)
```

Ouch!

- Unfortunately, algorithm BetterChange is simply **incorrect**, although being a generalisation of a correct algorithm.
  - Consider the case in whih $\mathbf{c} = (25, 20, 10, 5)$ and the amount of money M is 40. The algorithm would return the list $1, 0, 1, 1$, while there is a shorter one, namely $0, 2, 1, 1$.
  - What's the deep reason why the algorithm is not correct?
- Sometime, if one is not sure about the correctenss of the algorithm she has in mind, it is better to start with an algorithm which is **trivially correct**, although having perhaps other problems…

Change Problem:

Convert some amount of money M into given denominations, using the smallest possible number of 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 ($c_1 > c_2 > \cdots > c_d$).

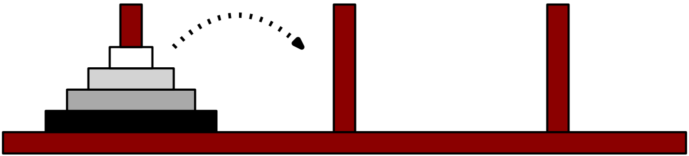
Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that $c_1 i_1 + c_2 i_2 + \cdots + c_d i_d = M$, and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

```
BruteForceChange(M, c, d)
  smallestNumberOfCoins ← ∞
  for each (i_1, ..., i_d) from (0, ..., 0) to (M/c_1, ..., M/c_d)
      valueOfCoins ← Σ_{k=1}^{d} i_k c_k
      if valueOfCoins = M
          numberOfCoins ← Σ_{k=1}^{d} i_k
          if numberOfCoins < smallestNumberOfCoins
              smallestNumberOfCoins ← numberOfCoins
              bestChange ← (i_1, i_2, ..., i_d)
  return (bestChange)
```

Direct Proofs of Correctness

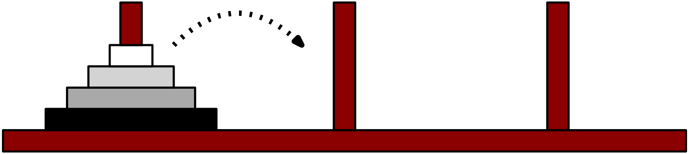
- Sometimes, the correctness of an algorithm can be proved by simply observing some simple facts, without any complicated mathematical arguments.
- This is the case of the algorithm BruteForceChange:
  - Any correct solution, and in particular, the optimal one, can be seen as a sequence between $(0, \ldots, 0)$ to $(M/c_1 \ldots, M/c_d)$.
  - The algorithm, simply, consider all such sequences one after the other.
  - At any iteration, the algorithm checks that the considered sequence indeed sums up to M.
  - It also keep track of *the best* sequence, namely the one with the fewest coins. This is done by two variables, *smallestNumberOfCoins* and **BestChange**. These are updated only when appropriate.

The Hanoi Puzzle



- One piece at a time.
- Never a larger piece stands above a smaller piece.

The Hanoi Puzzle



- One piece at a time.
- Never a larger piece stands above a smaller piece.

Towers of Hanoi Problem:
Output a list of moves that solves the Towers of Hanoi.

Input: An integer n.

Output: A sequence of moves that will solve the n-disk Towers of Hanoi puzzle.

Towers of Hanoi Problem:
Output a list of moves that solves the Towers of Hanoi.

Input: An integer n.

Output: A sequence of moves that will solve the n-disk Towers of Hanoi puzzle.

```
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
```

Proving the Correctness of Recursive Algorithms

- Recursively defined algorithm, like HanoiTowers, are particularly fit to be proved correct.
- The proof follows the structure of the algorithm, and consists in proving that:
  1. **Base Case**. Whenever the algorithm *does not* make any recursive call, it is correct.
  2. **Inductive Case**. If the algorithm *do make* recursive calls, it is correct *provided* all the recursive calls are themselves correct.
- It is of course crucial, in the inductive case, that the fact all the recursive calls are correct (called the **inductive hypothesis**) is *sufficient* to prove the algorithm correct.
  - Sometime this is not the case, and it is thus necessary to prove *a stronger* claim.

Proving the Correctness of HanoiTowers

- We can start by proving that HanoiTowers($n, \mathit{fromPeg}, \mathit{toPeg}$) correctly solves the Hanoi Problem, by induction on n
  1. The **base case** is easy.
  2. The inductive case fails, because the statement is too weak.
- We need a stronger statement, namely that HanoiTowers($n, \mathit{fromPeg}, \mathit{toPeg}$) correctly moves n (stacked) disks in *fromPeg* to *toPeg* *whenever* all the other disks in the three Peg are correctly stacked and of size higher than n.
  1. In this case, one can easily see that the inductive case works, too.

Proving the Correctness of HanoiTowers

- We can start by proving that HanoiTowers(n, *fromPeg*, *toPeg*) correctly solves the Hanoi Problem, by induction on n
  1. The **base case** is easy.
  2. The inductive case fails, because the statement is too weak.
- We need a stronger statement, namely that HanoiTowers(n, *fromPeg*, *toPeg*) correctly moves n (stacked) disks in *fromPeg* to *toPeg* *whenever* all the other disks in the three Peg are correctly stacked and of size higher than n.
  1. In this case, one can easily see that the inductive case works, too.

Fibonacci Problem:
Calculate the nth Fibonacci number.

Input: An integer n.

Output: The nth Fibonacci number $F_n = F_{n-1} + F_{n-2}$ (with $F_1 = F_2 = 1$).

Fibonacci Problem:
Calculate the nth Fibonacci number.

Input: An integer n.

Output: The nth Fibonacci number $F_n = F_{n-1} + F_{n-2}$ (with $F_1 = F_2 = 1$).

```
RecursiveFibonacci(n)
if n = 1 or n = 2
    return 1
else
    a ← RecursiveFibonacci(n − 1)
    b ← RecursiveFibonacci(n − 2)
    return a + b
```

Fibonacci Problem:
Calculate the nth Fibonacci number.

> **Input:** An integer n.
>
> **Output:** The nth Fibonacci number $F_n = F_{n-1} + F_{n-2}$ (with $F_1 = F_2 = 1$).

```
RecursiveFibonacci(n)
if n = 1 or n = 2
      return 1
else
      a ← RecursiveFibonacci(n − 1)
      b ← RecursiveFibonacci(n − 2)
      return a + b
```

```
Fibonacci(n)
F_1 ← 1
F_2 ← 1
for i ← 3 to n
      F_i ← F_{i−1} + F_{i−2}
return F_n
```

Correctness through Invariants

- The correctness of RECURSIVEFIBONACCI is very easy to be proved, since the algorithm's structure perfectly matches the definition of Fibonacci numbers.
  - There is not so much left to be proved.
- The algorithm FIBONACCI, is not recursive but rather *iterative*. Its proof of correctness is more delicate.
  - We need to find a statement, called an **invariant**, which is true before the *first* iteration of the **for** loop, which stays true after the execution of *any* such iteration, and which *implies* the correctness of the algorithm *as a whole*.
  - In our case such a statement can be

$$\forall j < i.F_j \text{ is a the } j\text{-th Fibonacci number.}$$

  - Could we find something slightly weaker?
- Why using FIBONACCI, then? Simply because it is *more efficient*!

Correctness through Invariants

- The correctness of RecursiveFibonacci is very easy to be proved, since the algorithm's structure perfectly matches the definition of Fibonacci numbers.
  - There is not so much left to be proved.
- The algorithm Fibonacci, is not recursive but rather *iterative*. Its proof of correctness is more delicate.
  - We need to find a statement, called an **invariant**, which is true before the *first* iteration of the **for** loop, which stays true after the execution of *any* such iteration, and which *implies* the correctness of the algorithm *as a whole*.
  - In our case such a statement can be

$$\forall j < i.F_j \text{ is a the } j\text{-th Fibonacci number.}$$

  - Could we find something slightly weaker?
- Why using Fibonacci, then? Simply because it is *more efficient*!

Correctness through Invariants

- The correctness of RECURSIVEFIBONACCI is very easy to be proved, since the algorithm's structure perfectly matches the definition of Fibonacci numbers.
  - There is not so much left to be proved.
- The algorithm FIBONACCI, is not recursive but rather *iterative*. Its proof of correctness is more delicate.
  - We need to find a statement, called an **invariant**, which is true before the *first* iteration of the **for** loop, which stays true after the execution of *any* such iteration, and which *implies* the correctness of the algorithm *as a whole*.
  - In our case such a statement can be

$$\forall j < i.F_j \text{ is a the } j\text{-th Fibonacci number.}$$

  - Could we find something slightly weaker?
- Why using FIBONACCI, then? Simply because it is *more efficient*!

Another Example

Sorting Problem:
Sort a list of integers.

Input: A list of n distinct integers $\mathbf{a} = (a_1, a_2, \dots, a_n)$.

Output: Sorted list of integers, that is, a reordering $\mathbf{b} = (b_1, b_2, \dots, b_n)$ of integers from $\mathbf{a}$ such that $b_1 < b_2 < \cdots < b_n$.

Another Example

Sorting Problem:
Sort a list of integers.

Input: A list of n distinct integers $\mathbf{a} = (a_1, a_2, \ldots, a_n)$.

Output: Sorted list of integers, that is, a reordering $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ of integers from $\mathbf{a}$ such that $b_1 < b_2 < \cdots < b_n$.

```
SELECTIONSORT(a, n)
for i ← 1 to n − 1
        a_j ← Smallest element among a_i, a_{i+1}, ... a_n.
        Swap a_i and a_j
return a
```

Fast and Slow Algorithms

- Different (correct) algorithms for the same problem can behave *very differently* when implemented as programs, even when using the same programming language and the same machine.
  - One can take much longer than the other to be executed!
  - The amount of memory one algorithm needs is perhaps much larger then the one the other needs.
- We will see in the Lab Module that a **purely empirical** approach to the benchmarking of algorithms makes a lot of sense.
- Benchmarking, being genuinely experimental, cannot however be exhaustive. Could we rather proceed analitically?

Fast and Slow Algorithms

- Different (correct) algorithms for the same problem can behave *very differently* when implemented as programs, even when using the same programming language and the same machine.
  - One can take much longer than the other to be executed!
  - The amount of memory one algorithm needs is perhaps much larger then the one the other needs.
- We will see in the Lab Module that a **purely empirical** approach to the benchmarking of algorithms makes a lot of sense.
- Benchmarking, being genuinely experimental, cannot however be exhaustive. Could we rather proceed analitically?

Fast and Slow Algorithms

- Different (correct) algorithms for the same problem can behave *very differently* when implemented as programs, even when using the same programming language and the same machine.
  - One can take much longer than the other to be executed!
  - The amount of memory one algorithm needs is perhaps much larger then the one the other needs.
- We will see in the Lab Module that a **purely empirical** approach to the benchmarking of algorithms makes a lot of sense.
- Benchmarking, being genuinely experimental, cannot however be exhaustive. Could we rather proceed analitically?

Measuring an Algorithm's Complexity

- With the (time) **complexity** of a given algorithm, A what we mean is an abstract measure of the execution time of A.
- An algorithm's complexity, being *a model*, is measured following a number of principles, namely the model's **axioms**:
  1. The complexity of A is simply the **number** of **basic** instructions which are executed when A is run on any of its instances. Each instruction costs *the same*.
  2. Since the number of instructions A executes may vary depending on the input, one expresses A's complexity as a function of some **parameters** of the input (typically, its *size*).
  3. In doing so, one is allowed to slightly **overapproximate** the amount of instructions involved in A's execution, for the sake of having simple expressions.
  4. Multiplicative and additive constants are typically **elided** themselves, and only the asymptotic behaviour of the involved functions matters.

Asymptotic Notation

- A function $f : \mathbb{N} \to \mathbb{N}$ is $O(g)$ if there are positive real constants c and x_0 such that $f(x) \leq c \cdot g(x)$ for all values of $x \geq x_0$.
  - **Example**: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $O(n^2)$, but also $O(n^3)$, and certainly $O(2^n)$. It is not, however, $O(n)$.
- A function $f : \mathbb{N} \to \mathbb{N}$ is $\Omega(g)$ if there are positive real constants c and x_0 such that $f(x) \geq c \cdot g(x)$ for all values of $x \geq x_0$.
  - **Example**: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $\Omega(n^2)$, but also $\Omega(n)$, but not $\Omega(n^3)$.
- A function f is $\Theta(g)$ if f is both $O(g)$ and $\Omega(g)$.

```
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
```

$$O(4^n)$$

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

$$O(4^n)$$

SelectionSort($\mathbf{a}$, n)

for $i \leftarrow 1$ **to** $n - 1$
    $a_j \leftarrow$ Smallest element among $a_i, a_{i+1}, \ldots a_n$.
    Swap a_i and a_j
return a

$$O(n^2)$$

Thank You!

Questions?