Advanced Programming
INFO135

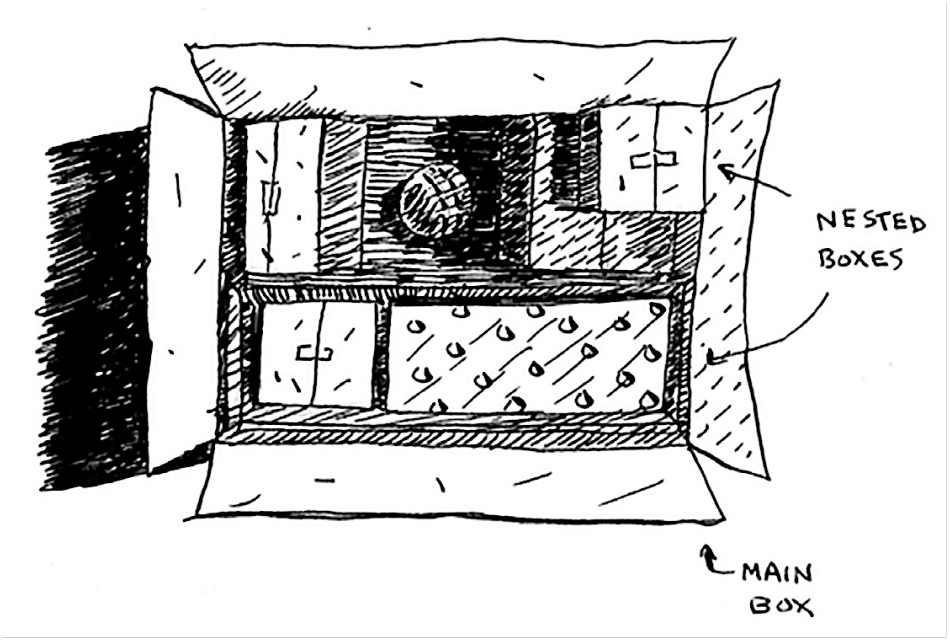
Lecture 5: Recursion

Mehdi Elahi

University of Bergen (UiB)

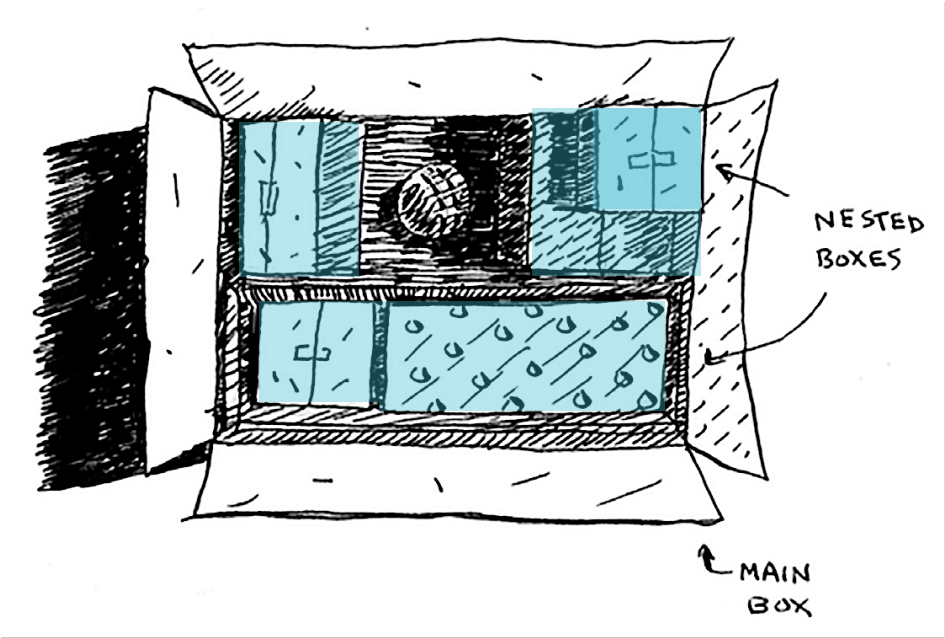
Recursion

- Suppose you're searching for the **key of a suitcase**.
- Grandma tells you that **the key is in this box**.



Recursion

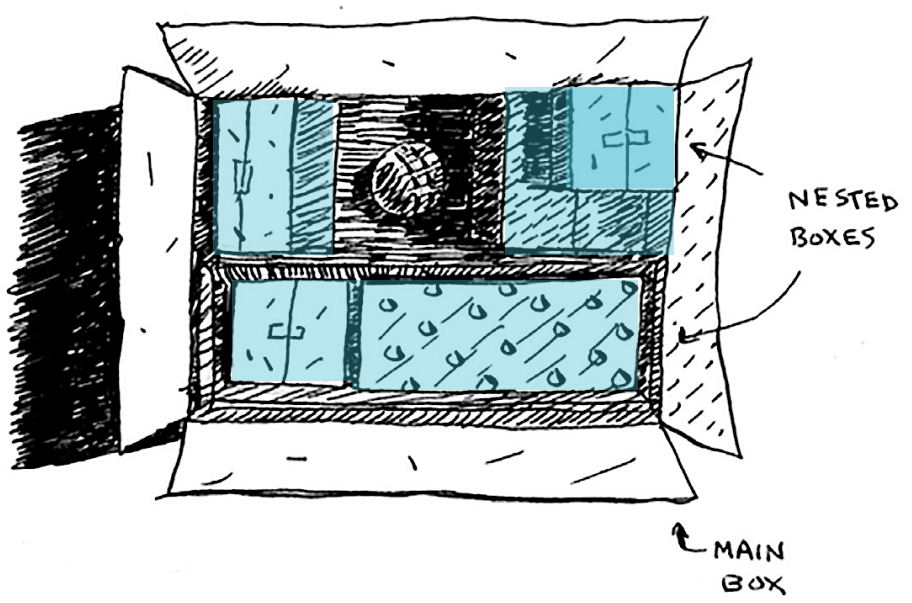
- The box contains **more boxes**, & more boxes inside those boxes!
- The **key** is in a box **somewhere**.
- What can be an **algorithm to search** for the key?



Recursion

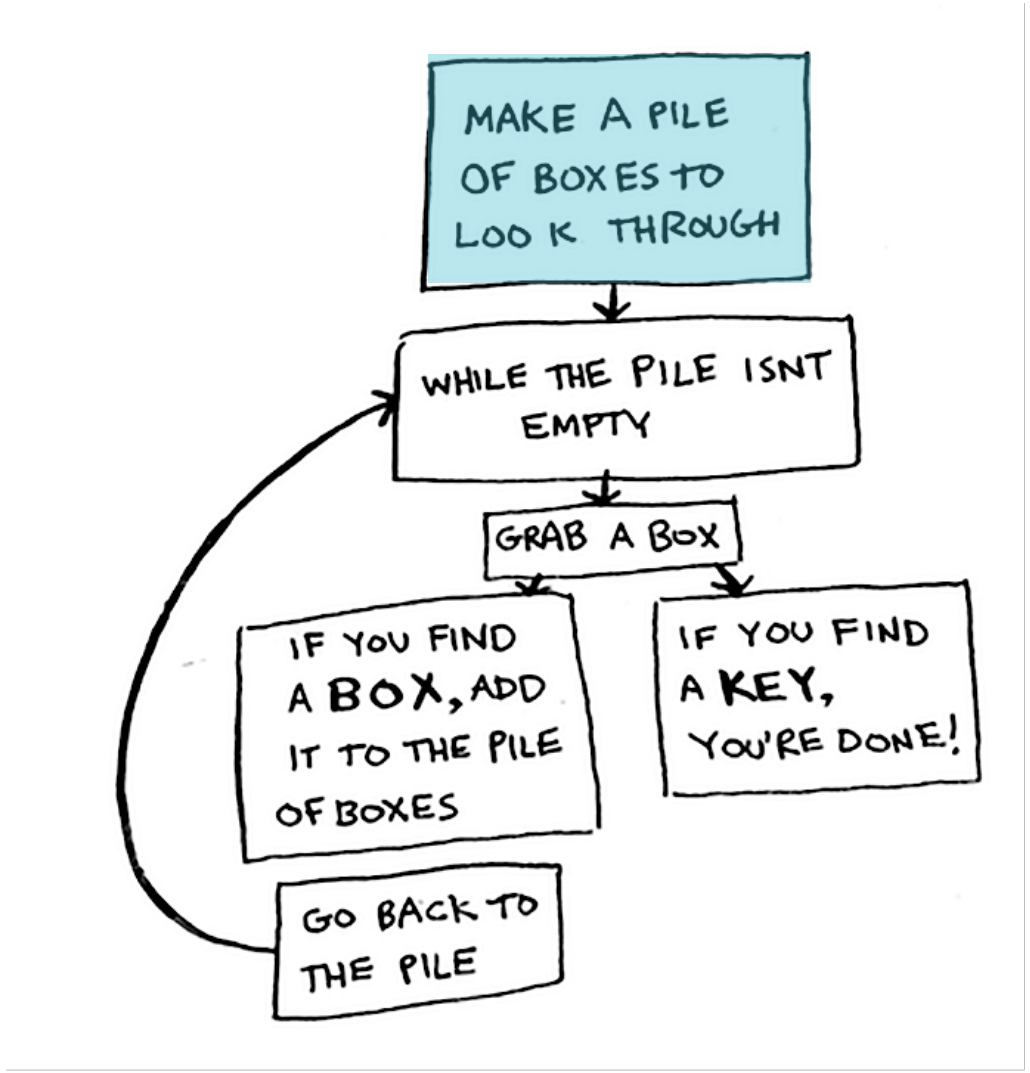
❖ Here's an **example:** lets call it **algorithm 1**:

1) make a pile of boxes **to look through**.
2) **grab a box**, and look through it.
3) **if you find** another box inside,
   a) add **it to the pile** to look through later.
   b) if you **find a key**, you're done!
4) **repeat**.



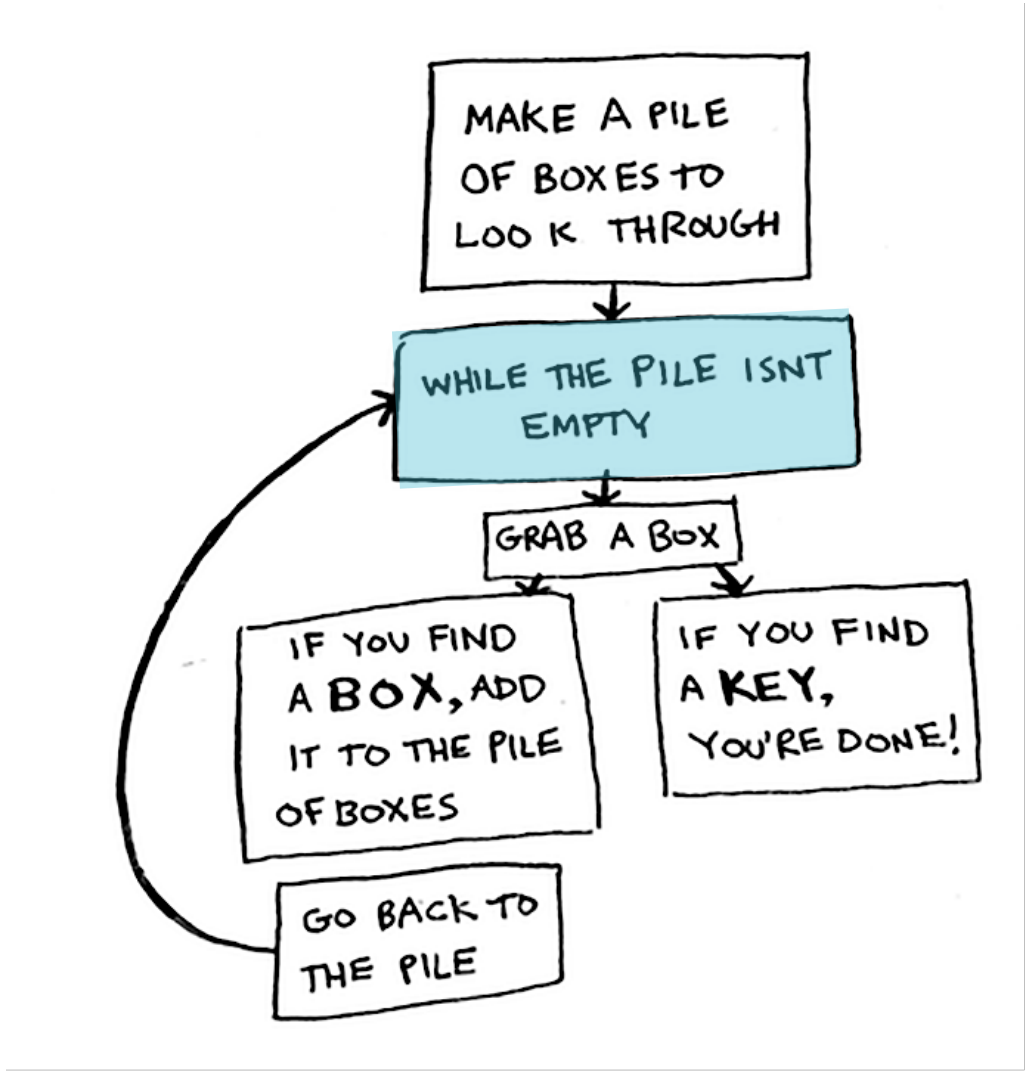
Recursion

- Here's **algorithm 1**:



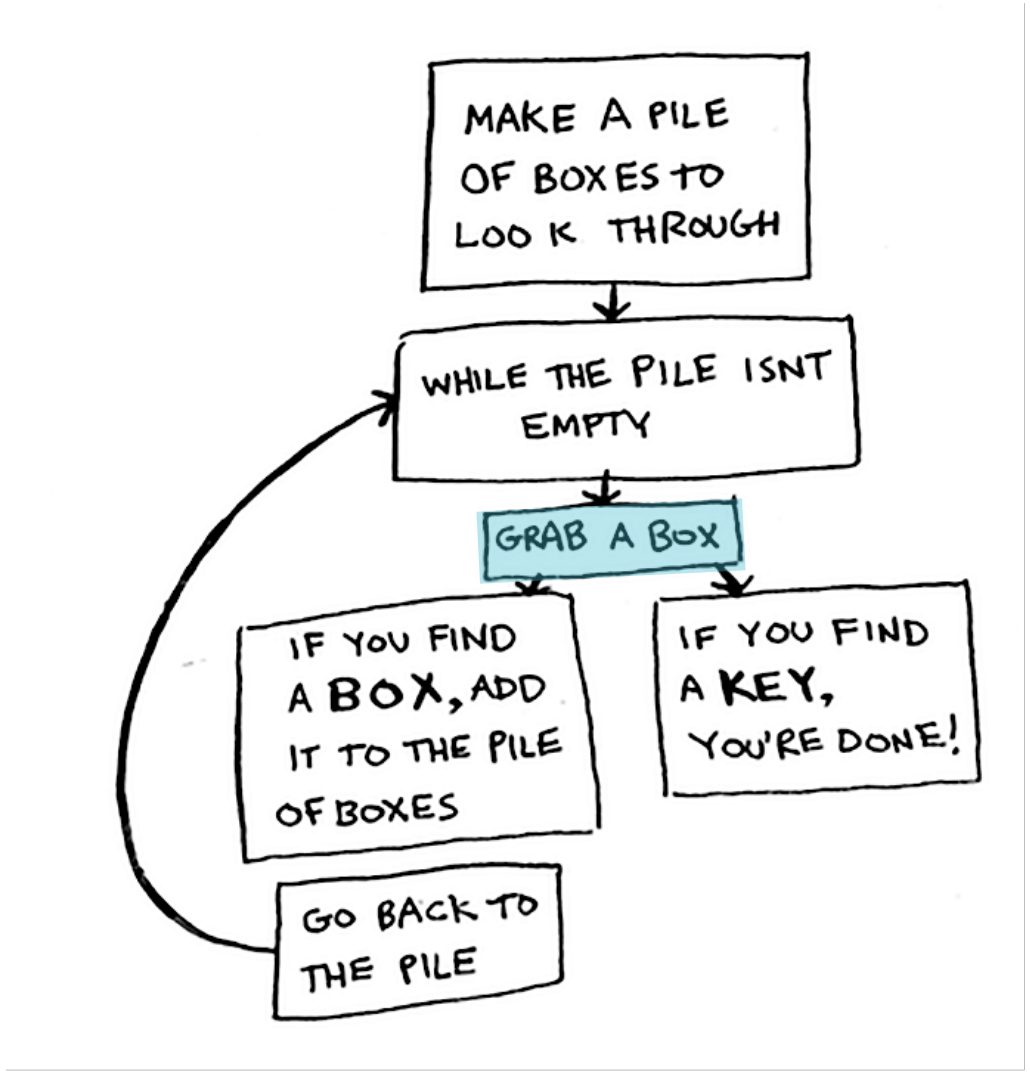
Recursion

- Here's **algorithm 1**:



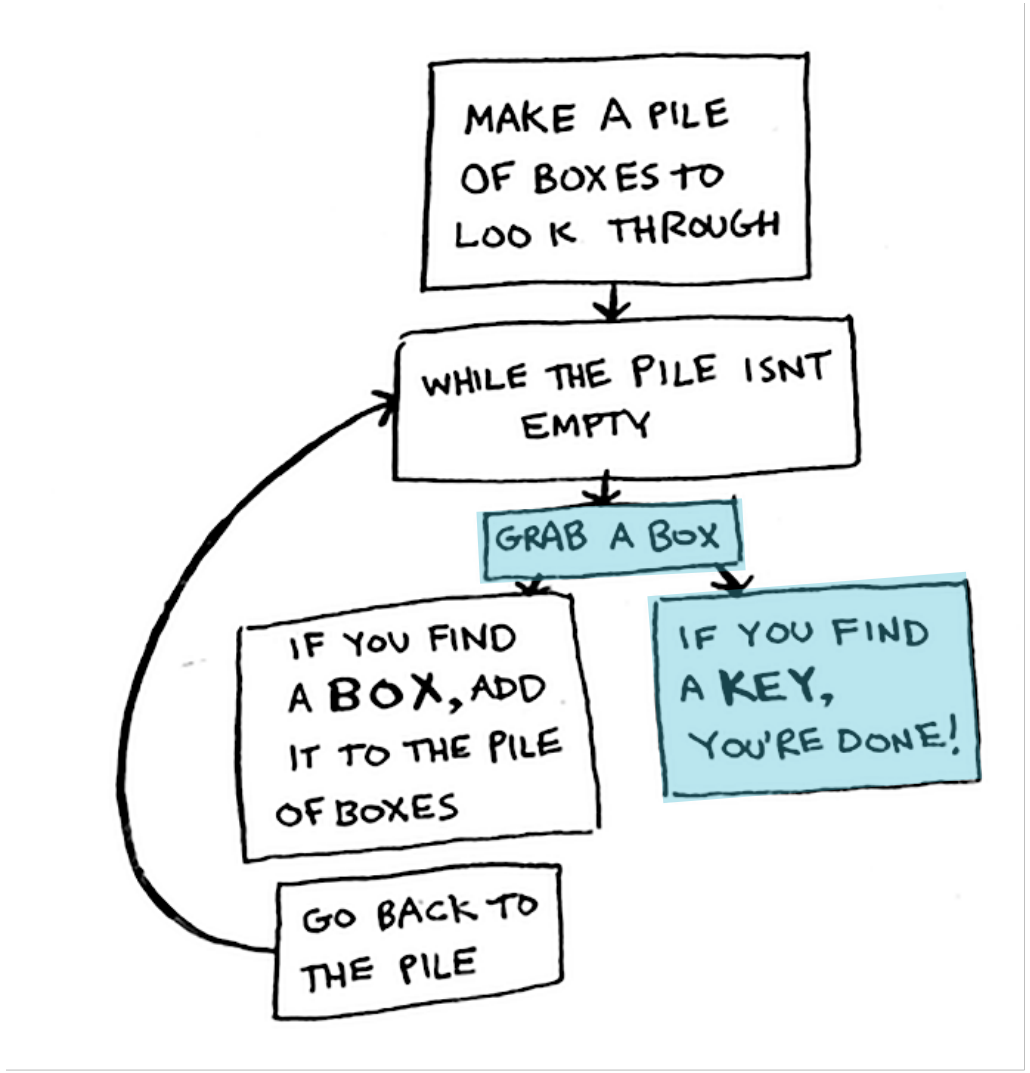
Recursion

- Here's **algorithm 1**:



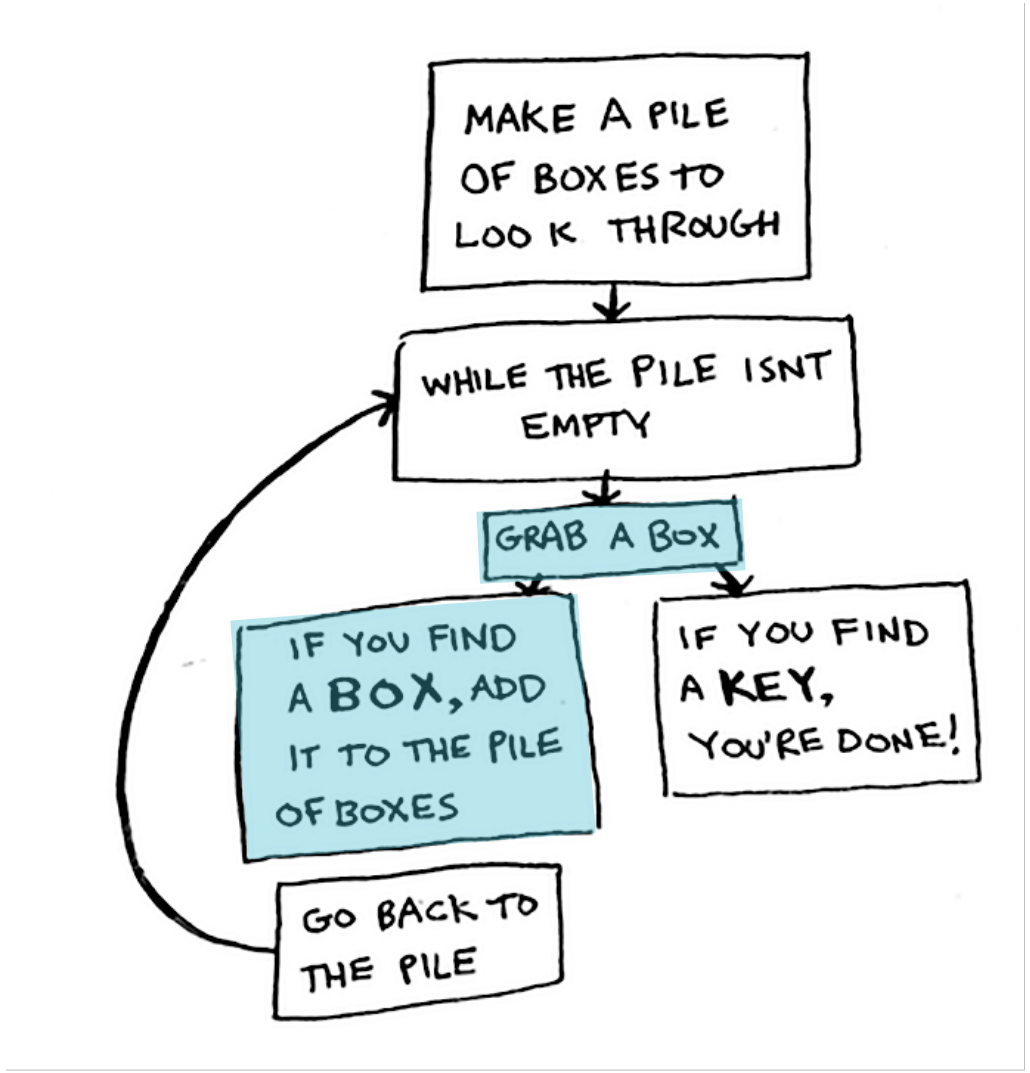
Recursion

- Here's **algorithm 1**:



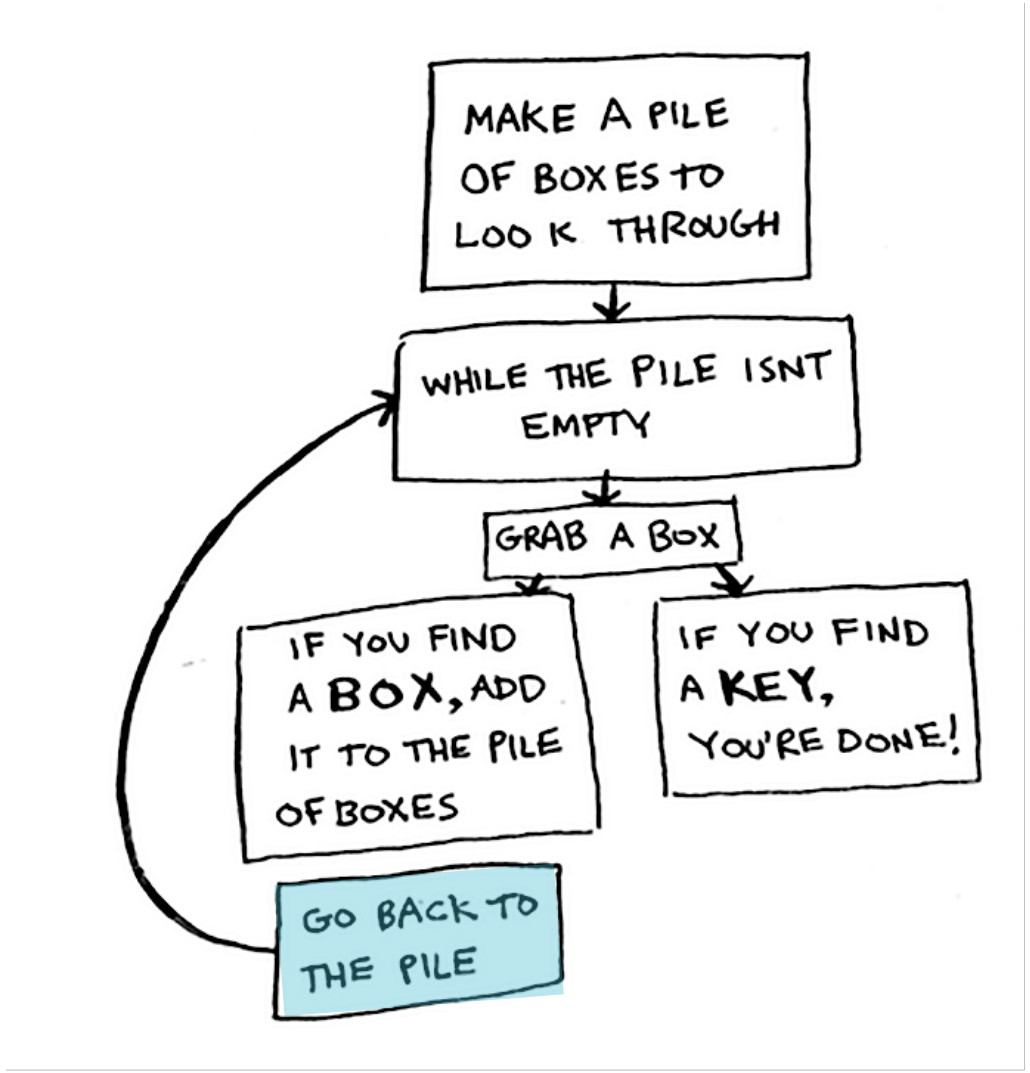
Recursion

- Here's **algorithm 1**:



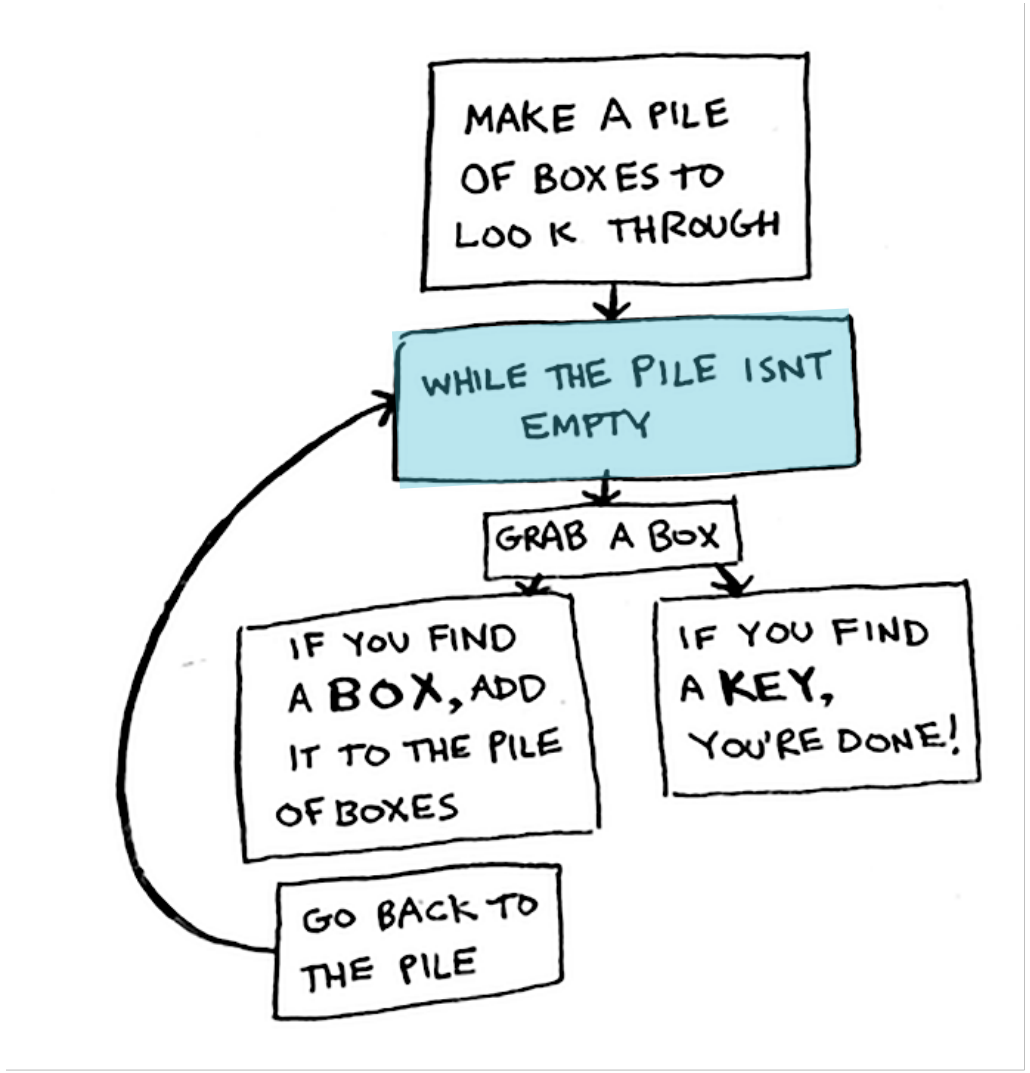
Recursion

- Here's **algorithm 1**:



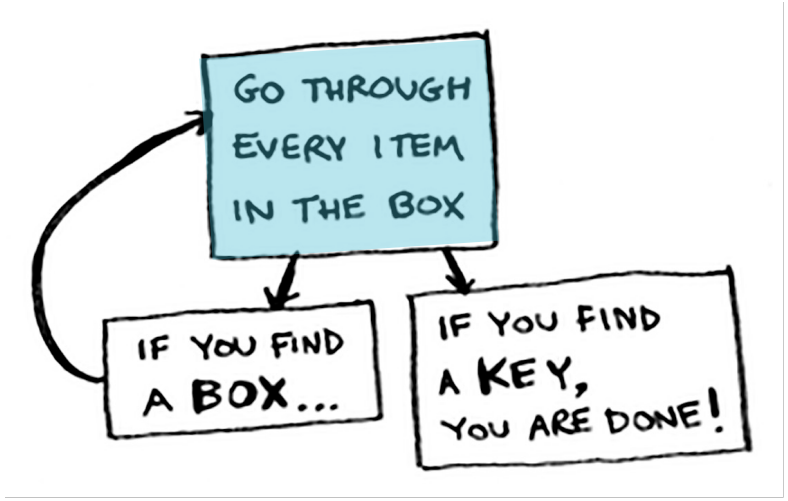
Recursion

- Here's **algorithm 1**:



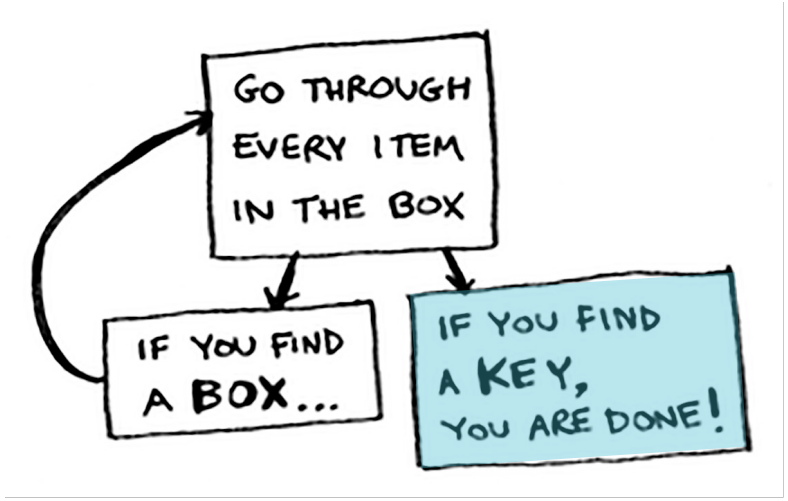
Recursion

- Here's another example: lets call it **algorithm 2**:



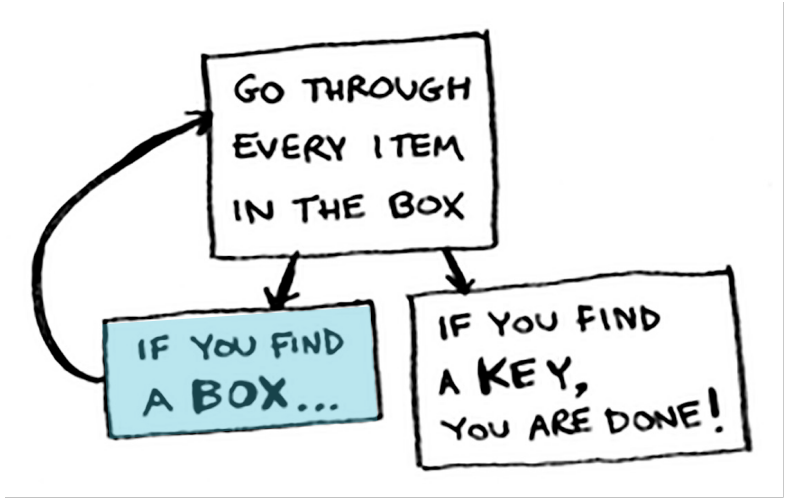
Recursion

- Here's **algorithm 2**:



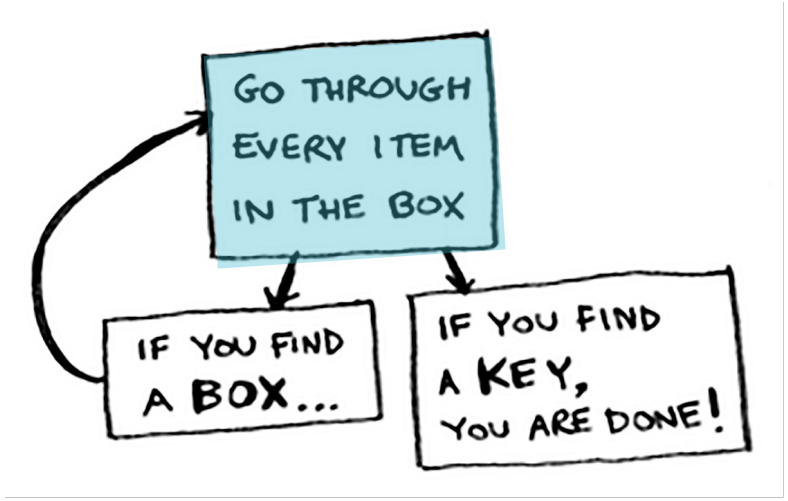
Recursion

- Here's **algorithm 2**:



Recursion

- Here's **algorithm 2**:



Recursion

- Here's the implementation of **algorithm 1**:

```
def look_for_key(main_box):
    pile = main_box.make_a_pile()
    while pile is not None:
        box = pile.grab_a_box()
        for item in box:
            if item.is_a_box():
                pile.append(item)
            elif item.is_a_key():
                print('found the key!')
```

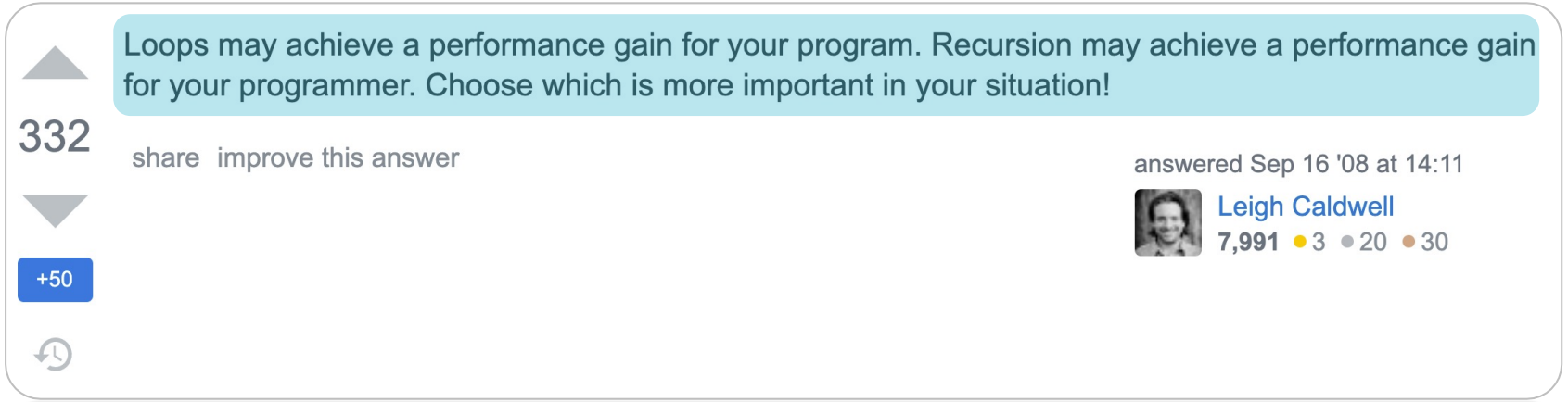
- Here's the implementation of **algorithm 2**:

```
def look_for_key2(box):
    for item in box:
        if item.is_a_box():
            look_for_key2(item)
        elif item.is_a_key:
            print('found the key!')
```

Which one is clearer ?

Recursion

- **Recursion** is used when it makes the solution **clearer**.
- Recursion is when a **function calls itself**.
- But **loops** are sometimes better for **performance**.



- Many **important algorithms use recursion.**

Quiz

- Write a **recursive function** ***sum(n)*** that receives an input ***n*** and sums all the nonnegative integers up to ***n***

Answer

Write a recursive function to

ref : Youtube

Recursion

- **Recursive algorithms** must obey **three laws**:
  1) a recursive algorithm must have a **base case**.
  2) a recursive algorithm must **move toward** base case.
  3) a recursive algorithm must **call itself**, recursively.

- **Attention:** because a **recursive function** calls itself, it's easy to write a function that ends up in an **infinite loop**.

- Sometimes they can be **computationally** expensive. Lets check an example!

Recursion

- **Fibonacci sequence:** is a sequence of integer numbers in which each subsequent value is the sum of the two previous values.

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2), & \text{if } n > 1 \\ n, & \text{if } n = 1 \text{ or } n = 0 \end{cases}$$

- The first 11 numbers of the sequence are:

1, 1, 2, 3, 5, 8, 13, 21, 34, …

Recursion

- **Fibonacci sequence:**

fib(0) **1**

fib(1) **1**

1+1=**2**

1+2=**3**

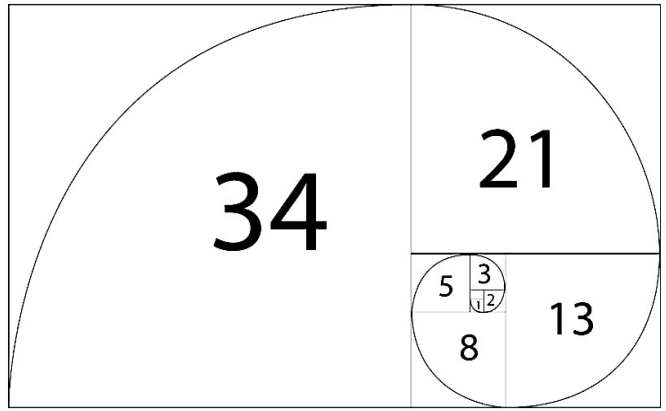
2+3=**5**

3+5=**8**

5+8=**13**

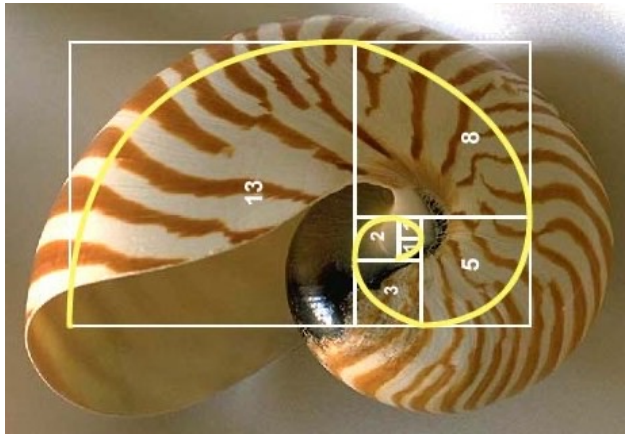
8+13=**21**

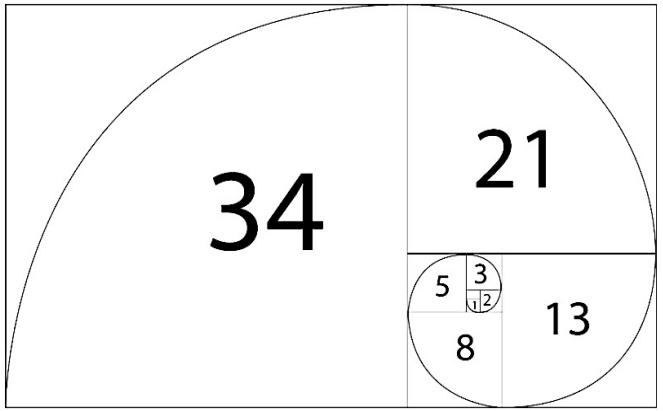
13+21=**34**



Recursion

Fibonacci sequence:





1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1

1 1 2 3 5 8 13 21 34 55 89

Recursion

❖ **Fibonacci sequence** with recursion:

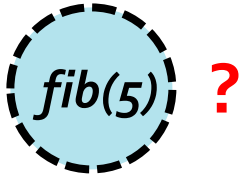
```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)


print('The 40th Fibonacci: ', fib(40))
```

[output:]

```
The 40th Fibonacci:  102334155
```

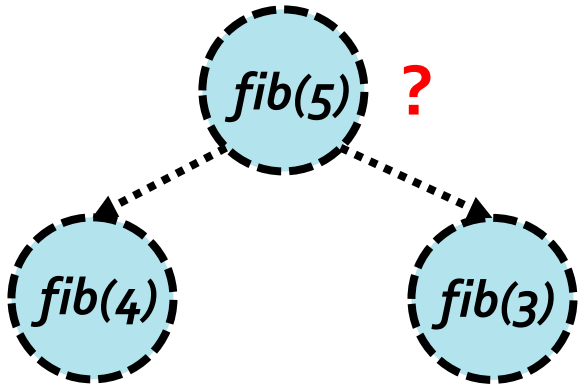
Recursion

- **Fibonacci sequence** recursion tree.
- Lets compute ***fib(5)***



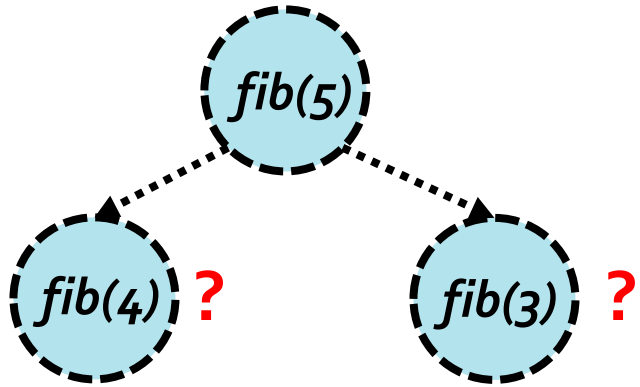
Recursion

- Lets compute ***fib(5)*** = ***fib(4)*** + ***fib(3)***



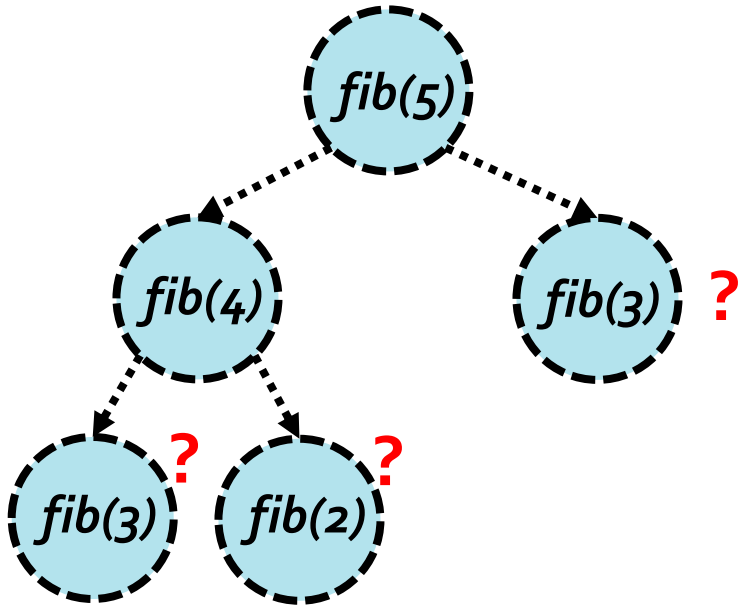
Recursion

- So we need to compute:
  - *fib(4)*
  - *fib(3)*



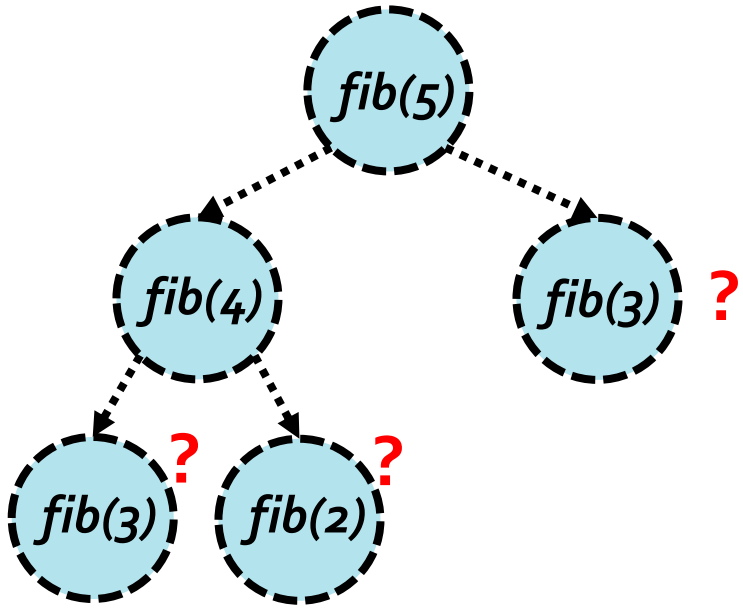
Recursion

- So we need to compute:
  - ***fib(4) = fib(3) + fib(2)***



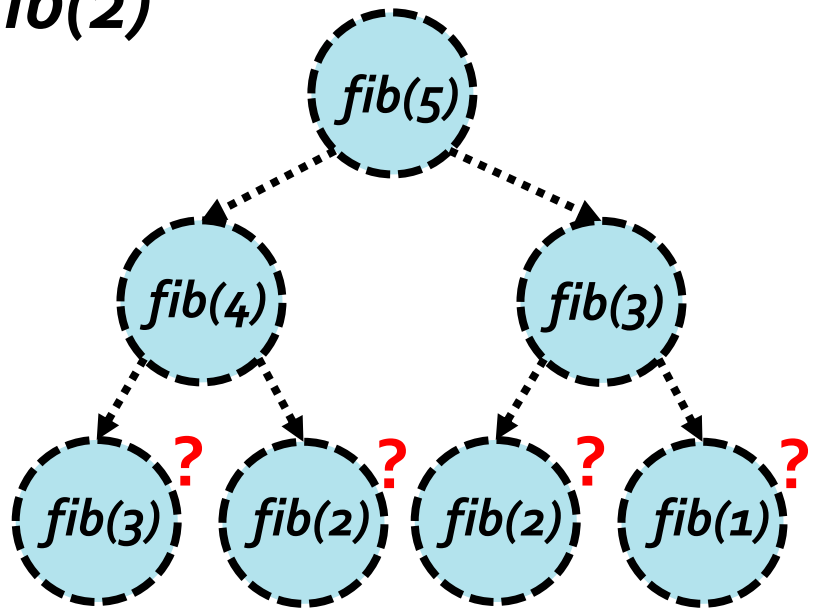
Recursion

- So far we need compute:
  - *fib(4)*
  - again *fib(3)*
  - *fib(2)*



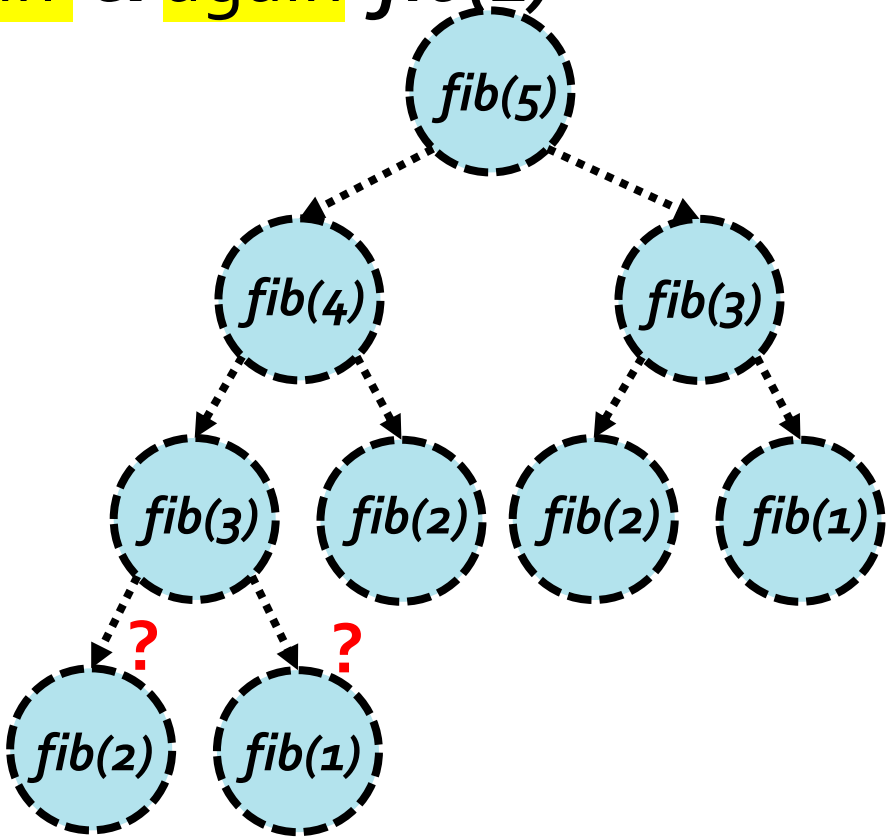
Recursion

- So far we need compute:
  - ***fib(4)***
  - again ***fib(3)***
  - again ***fib(2)***
  - ***fib(1)***



Recursion

- If we don't store, we need to compute:
  - …
  - again & again *fib(2)*
  - again & again *fib(1)*



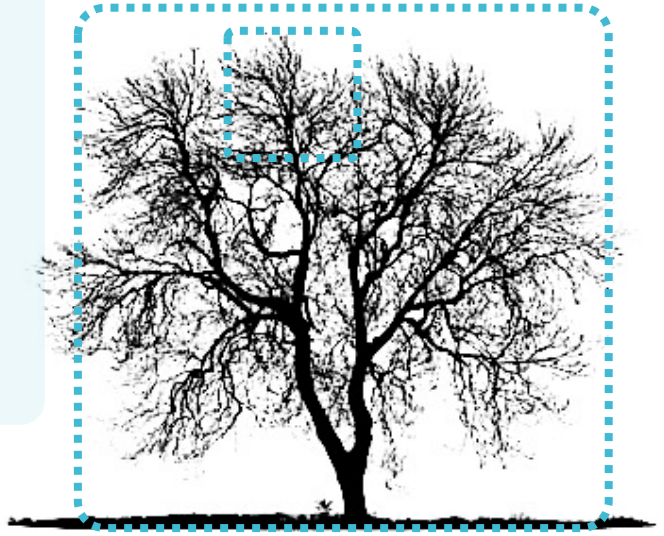
In future lectures we will deal this and find a way **to optimize such functions!**

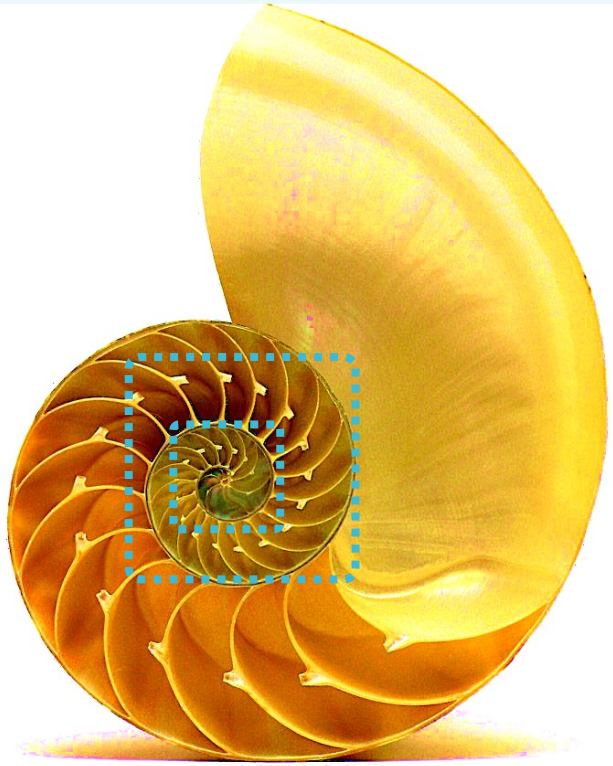
Recursion

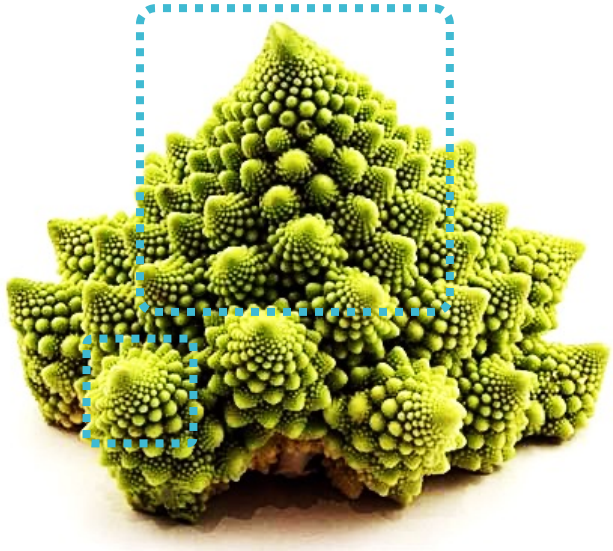
- There are more such types of **reclusive functions** in nature. Examples are **fractals**.
- It might be interesting to use Python to **visualize some of them**.

Visulazing Recurrsion

- Examples in nature:
- **Fractal** is a structure that looks the same at all different levels of magnification.

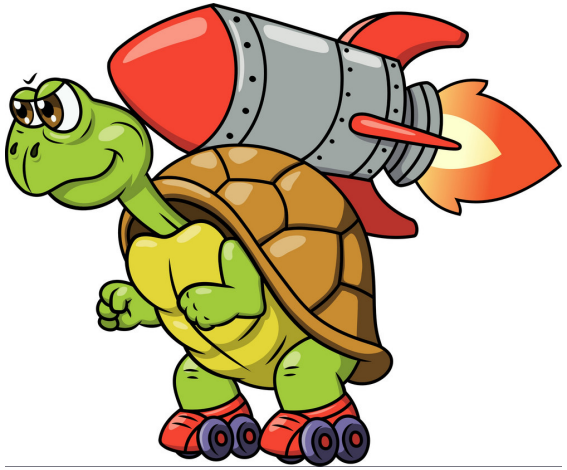






Visulazing Recurrsion

- **Turtle graphics** module:
  - is standard with Python.
  - is easy to use and fun.



- You can create a **turtle** that can **move forward, backward**, turn left, turn right, with the tail up or down.

- When the turtle's tail is down it **draws a line** as it moves (with different **colors**).

Visualizing Recursion

- Lets try an example of Python Turtle.

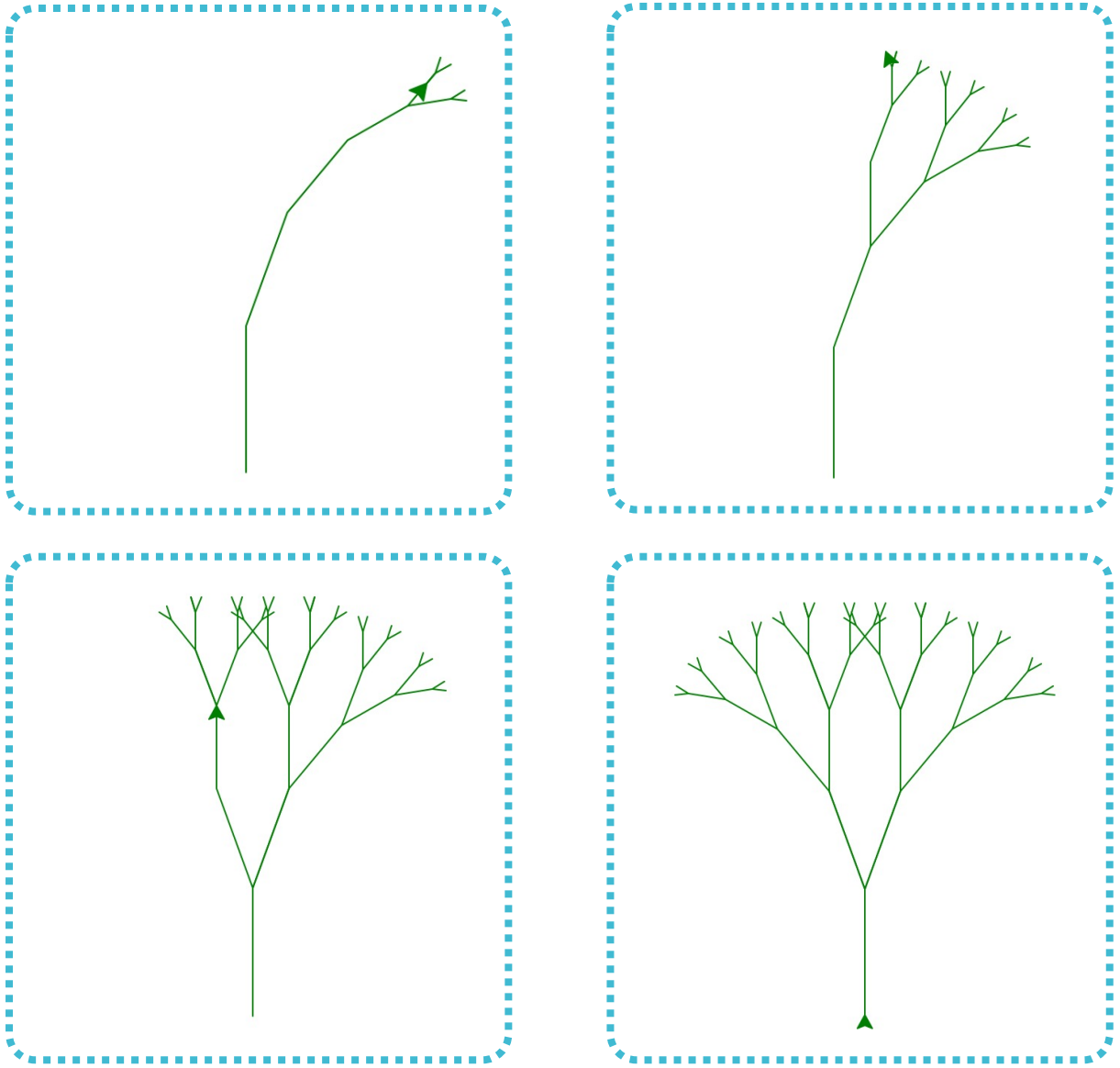
```
import turtle


def tree(branch_len, my_turtle):
    if branch_len > 5:
        my_turtle.forward(branch_len)
        my_turtle.right(20)
        tree(branch_len - 15, my_turtle)
        my_turtle.left(40)
        tree(branch_len - 15, my_turtle)
        my_turtle.right(20)
        my_turtle.backward(branch_len)
```

Visualizing Recursion

- Lets try an example of Python Turtle.

```
def run():
        my_turtle = turtle.Turtle()
        my_win = turtle.Screen()
        my_turtle.left(90)
        my_turtle.up()
        my_turtle.backward(150)
        my_turtle.down()
        my_turtle.color("green")
        tree(85, my_turtle)
        my_win.exitonclick()


run()
```

Visualizing Recursion



Quiz

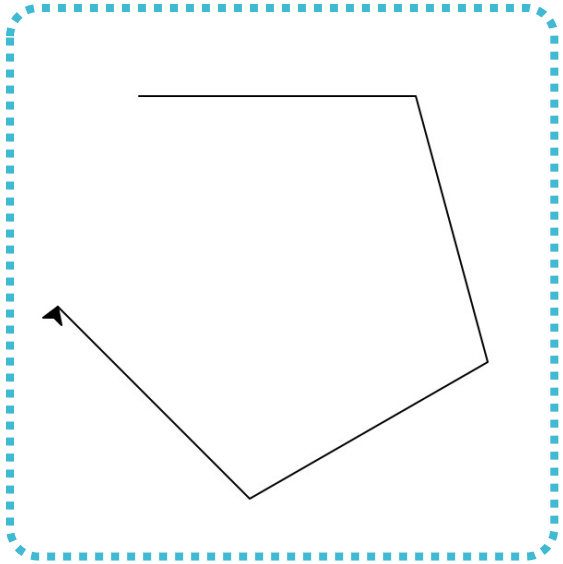
- What is the output of the following code:

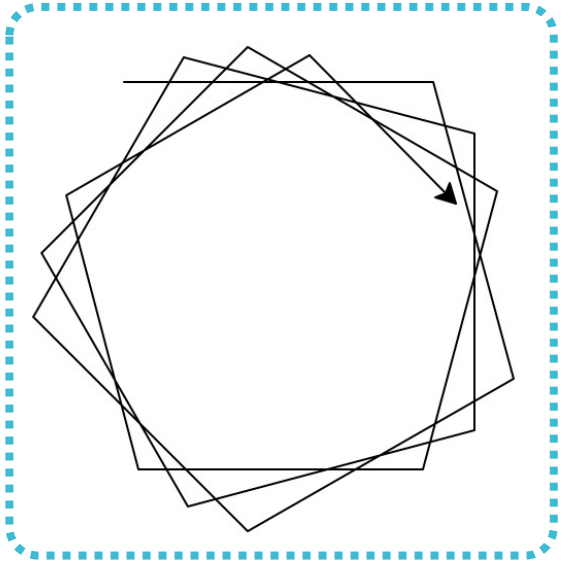
```
import turtle

my_turtle = turtle.Turtle()
my_win = turtle.Screen()
```

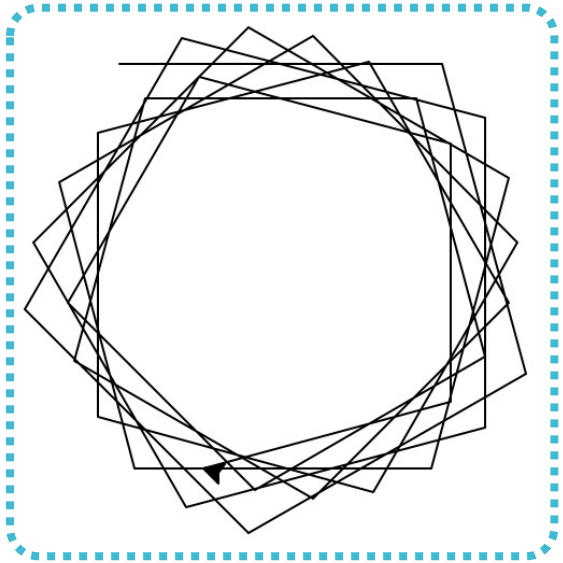
```
def draw_spiral(my_turtle, length):
    if length > 0:
        my_turtle.forward(length)
        my_turtle.right(75)
        draw_spiral(my_turtle, length - 1)
```

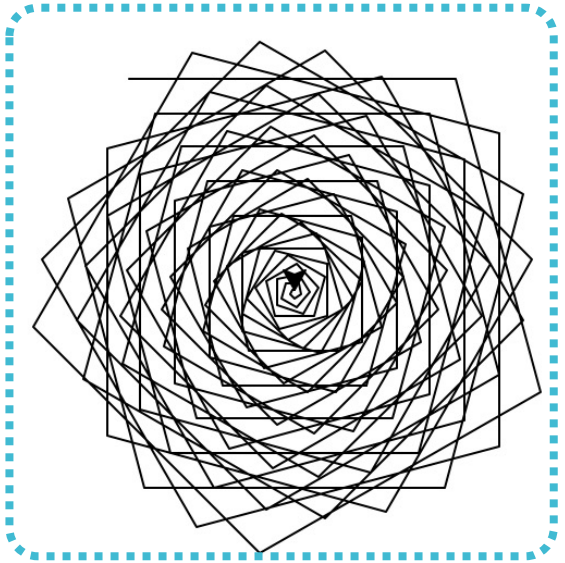
```
draw_spiral(my_turtle, 150)
my_win.exitonclick()
```

Answer









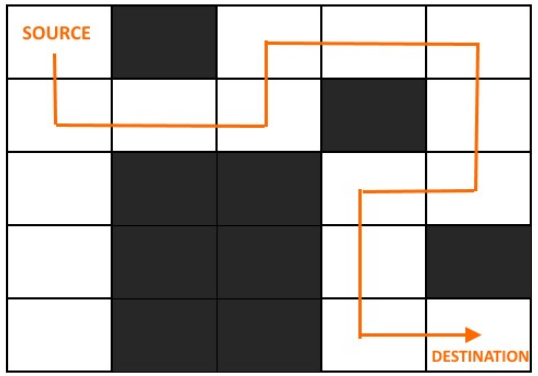
Recursion

- One of the interesting applications of recursion is **backtracking**.
- Now we will check a problem that can be solved by **backtracking** algorithms.

Backtracking

- **Backtracking** is a popular problem solving technique that builds up **partial solutions** that get increasingly closer to the goal.

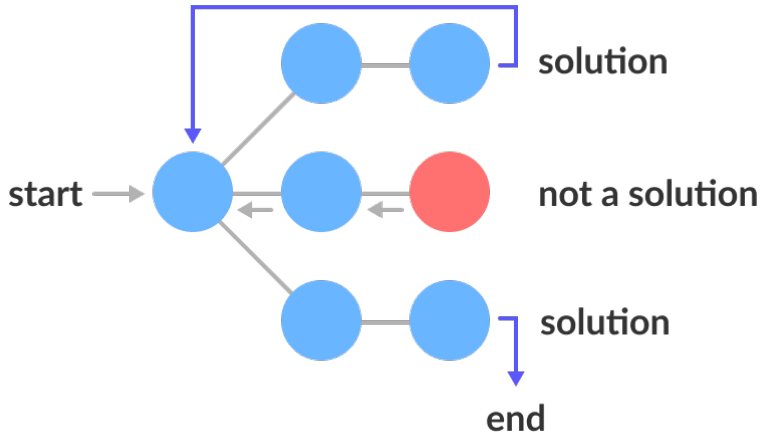
- Backtracking can be used to **solve game problems** such as crossword puzzles, escape from mazes.



Backtracking

- If a partial solution cannot be completed, one **abandons it** and returns to examining the other candidate solutions.

- In order to employ backtracking for a particular problem, we need **two characteristic properties**.



Backtracking

- **Two characteristic properties.**
  1. A procedure to **examine** **a partial solution** to:
     - **"Accept"** it as a solution.
     - **"Abandon"** it (either because it violates rules or it can never lead to a valid solution).
     - **"Continue"** extending it.

Backtracking

- In order to employ backtracking, we need **two characteristic properties.**

  2. A procedure to **extend a partial solution**, by generating more solutions that come closer to the goal.

- Note that the processes of examining and extending a partial solution depend on the **nature of the problem**.

Backtracking

- Backtracking can then be expressed with the following **recursive algorithm**:

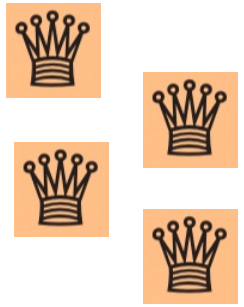
```
solve(partial_solution)
   examine(partial_solution)
   if accepted
       add partial_solution to solutions
   else if not abandoned
      for each p in
      extend(partial_solution)
         solve(p).
```

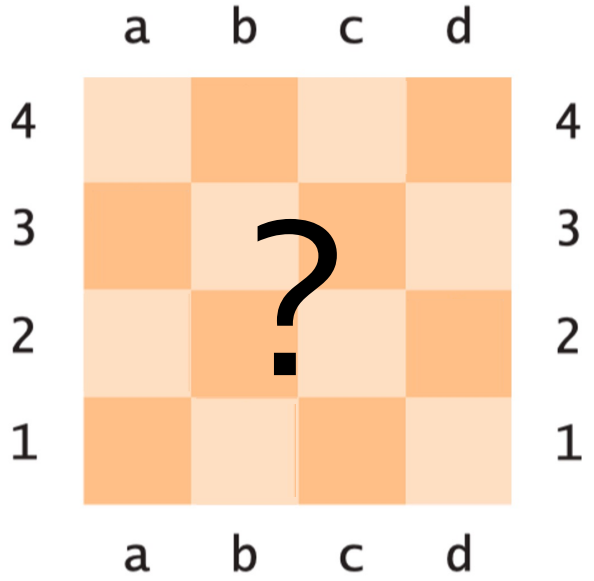
N Queens Problem

- If you have evenr played chess, you might be aware of **N Queen puzzle**.
- Historically, a chess composer Max Bezzel published the **8 queens puzzle** in 1848.
- Franz Nauck published the **first solutions in 1850**.

N Queens Problem

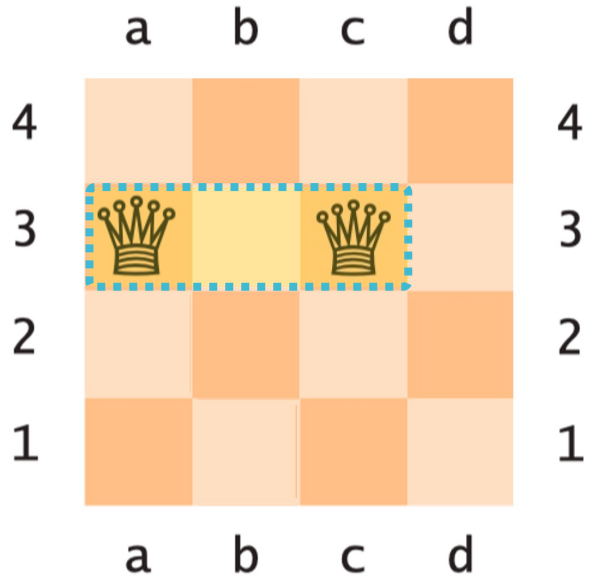
- **Example:** develop a Python program to find solutions to the following problem.
- There are **N queens** on a chess board of **NxN.**
- How can they be positioned so that **none of them attacks** another according to the rules of chess.





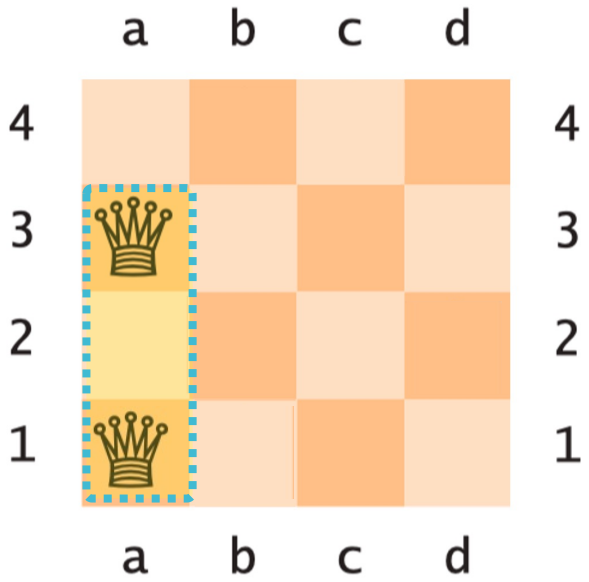
N Queens Problem

- The rule is that **2 queens** can **not** be on:
  - **the same row**
  - the same column
  - the same diagonal.



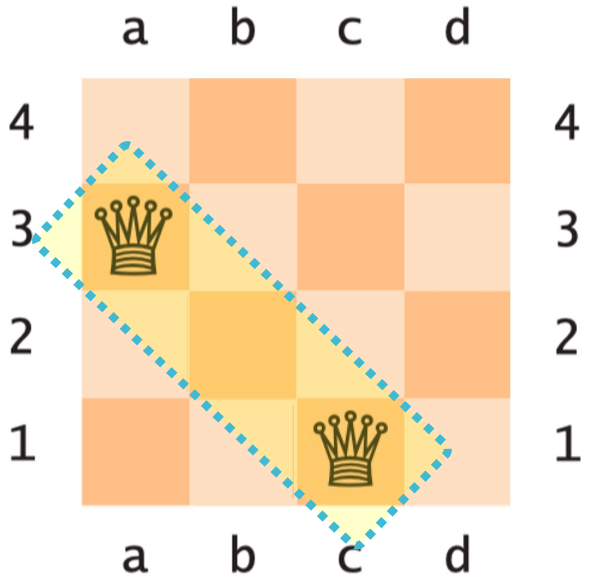
N Queens Problem

- The rule is that **2 queens** can **not** be on:
  - the same row
  - **the same column**
  - the same diagonal.



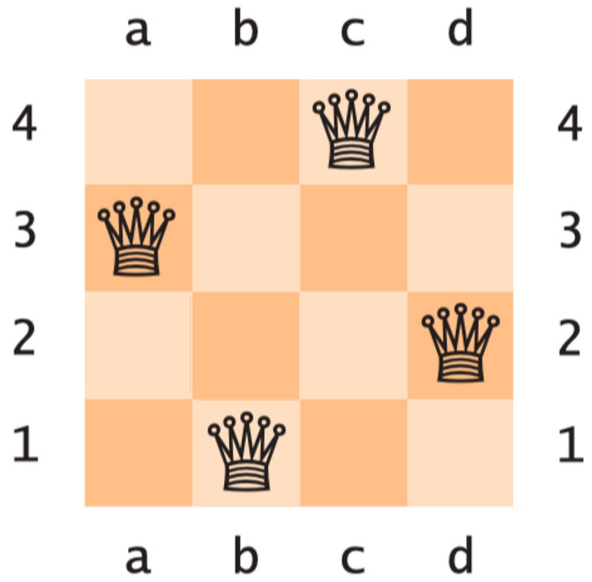
N Queens Problem

- The rule is that **2 queens** can **not** be on:
  - the same row
  - the same column
  - **the same diagonal.**



4 Queens Problem

- **Example solution** for 4 queens in 4x4 chess.
- We represent a partial solution as a **list of strings.**
- **Such as** ["a3", "b1", "c4", "d2"]
- Each string gives a **queen position** in the chess.

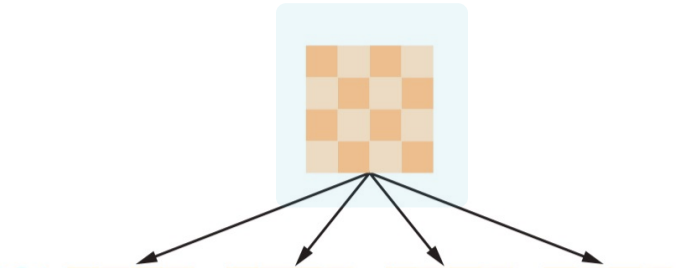


4 Queens Problem

- In this problem, easy to **examine partial solution**:
  - If two queens attack each other, **reject** it.
  - Otherwise, if it has N=4 queens, **accept** it as solution.
  - Otherwise, **continue**.

4 Queens Problem

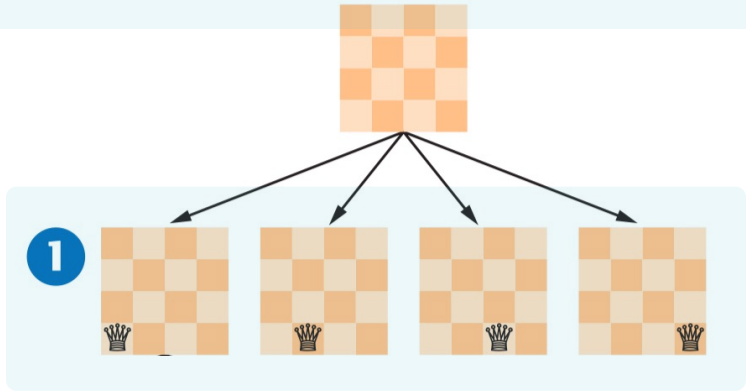
- We would start with a blank and **extend a partial solutions** by add a queen in this empty square.



?

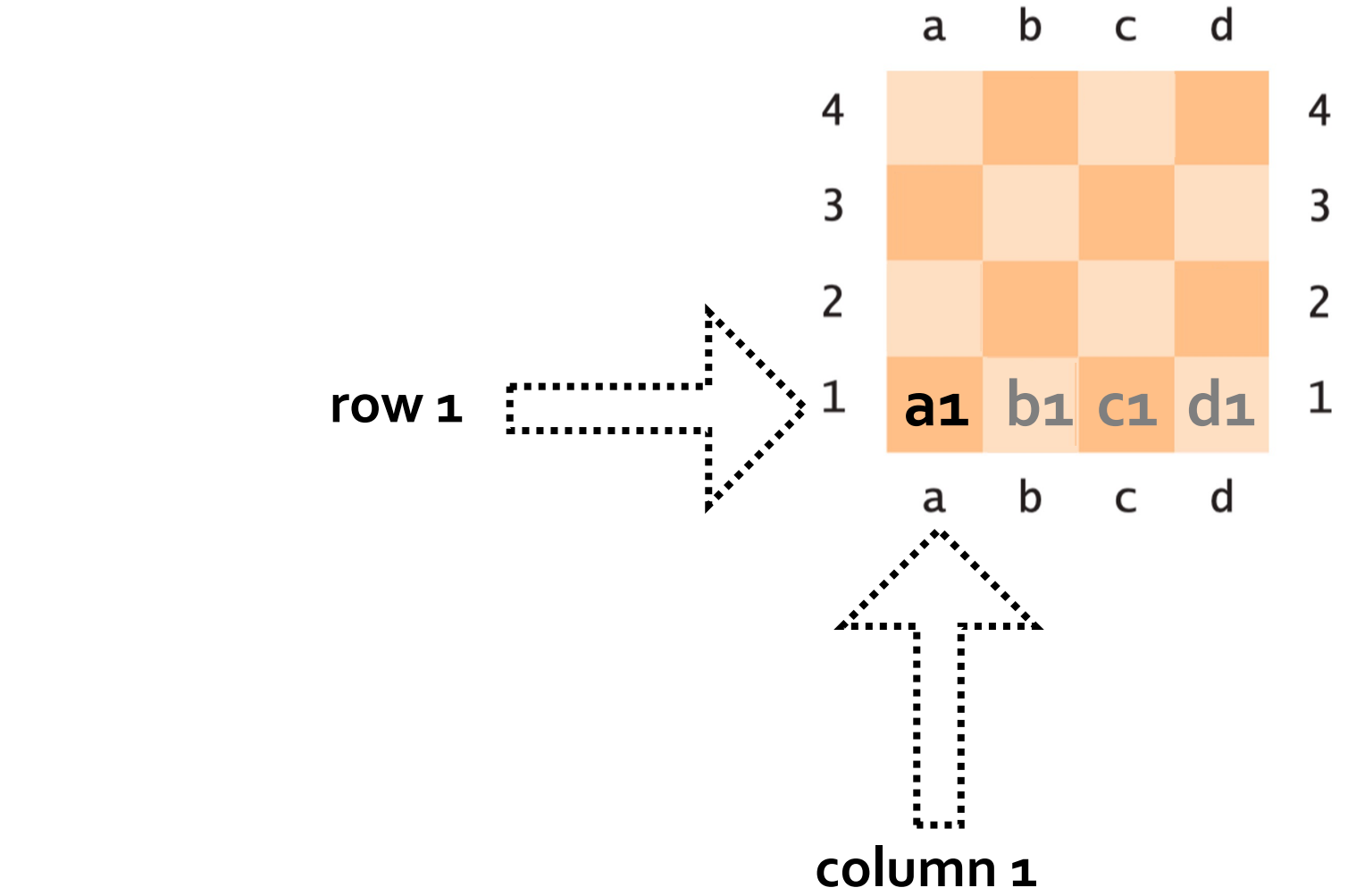
4 Queens Problem

- **Question:** How many partial solutions are there if a queen is in position (for instance) in row 1?
- **Answer:** 4 partial solutions.



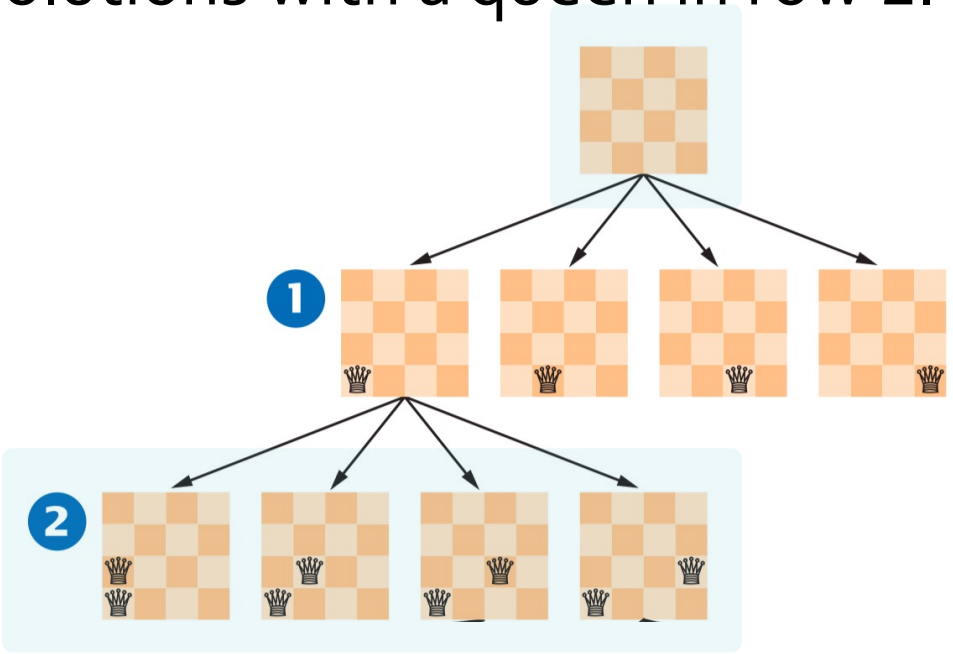
4 Queens Problem

- Then the **algorithm begins with positioning** a queen in position **a1** (row 1, column 1).



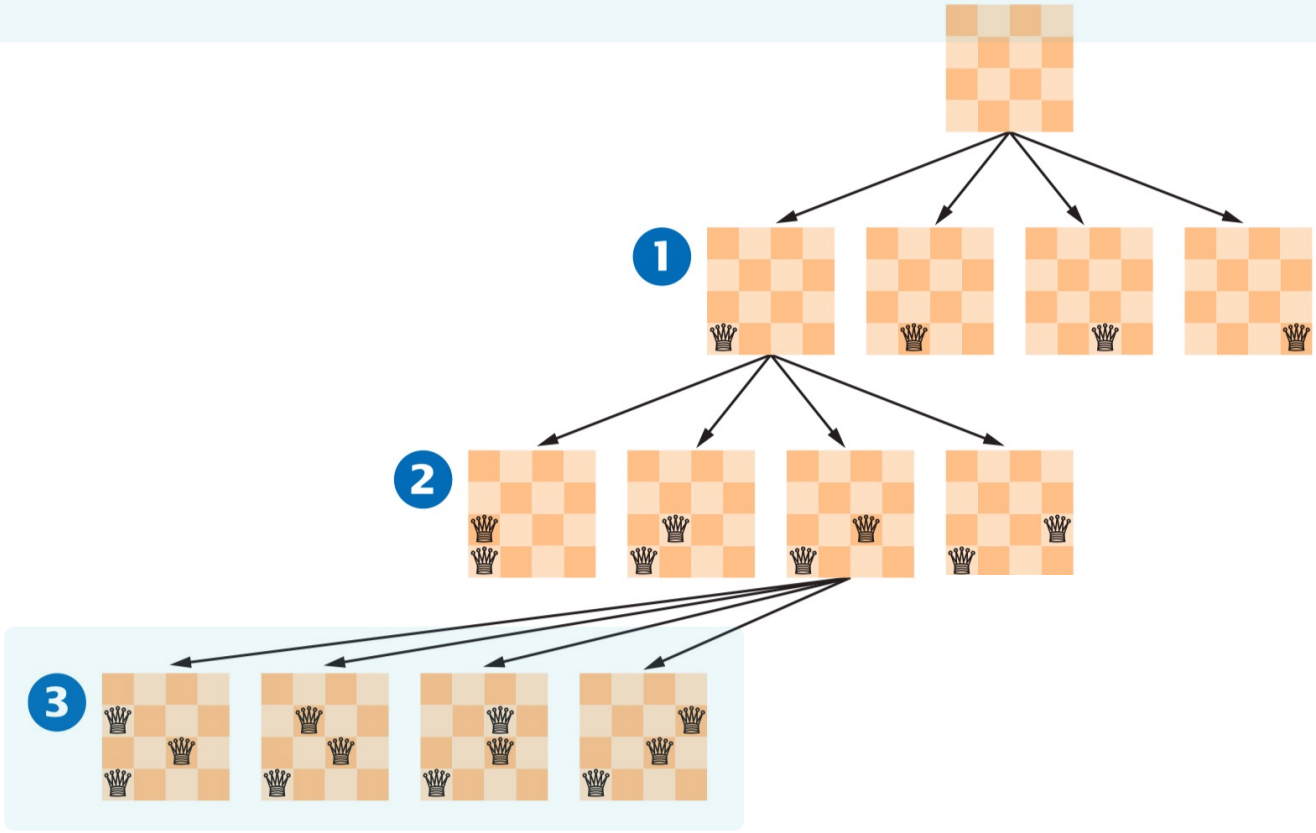
4 Queens Problem

- We need to check all **possible scenarios**.
- If the **queen is in row 1 & column 1**, there are again 4 partial solutions with a queen in row 2.



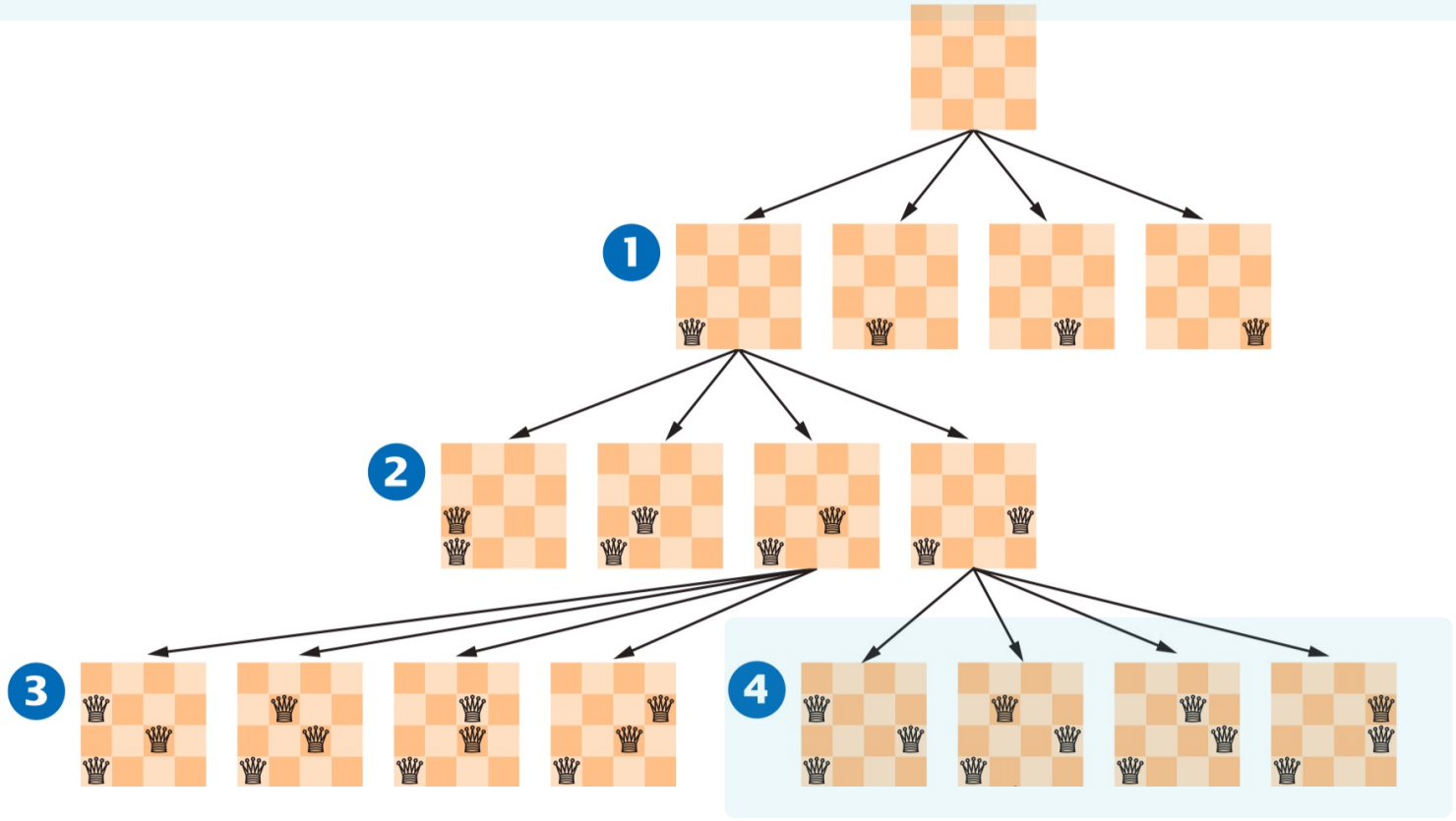
4 Queens Problem

- We can take one of the partial solutions & **extend it with more queens**.
- If you check them carefully, all abandoned again. **Why?**



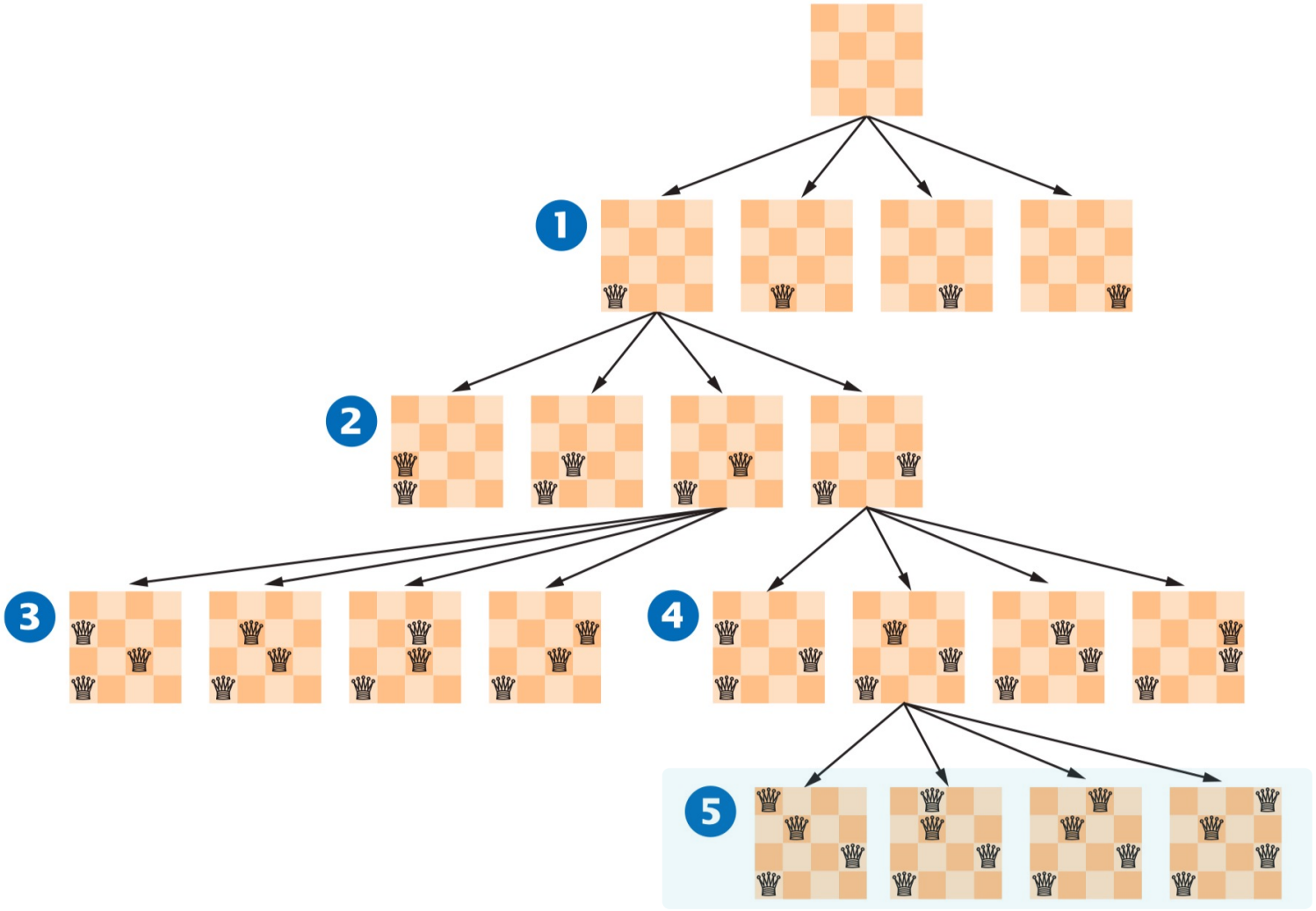
4 Queens Problem

- The other two lead to partial solutions **with 3 queens.**
- But all are abandoned **except one**. Which one?



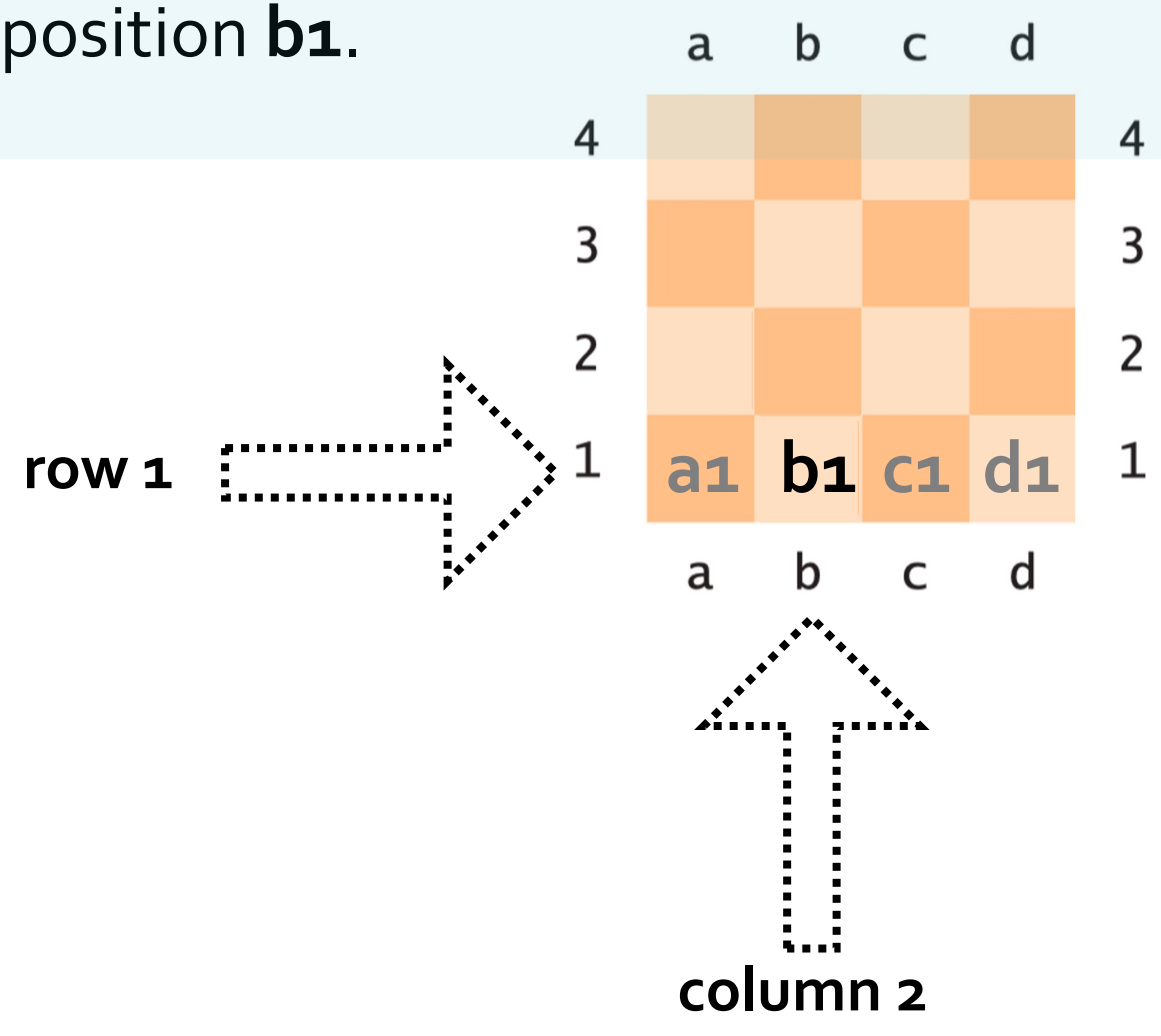
4 Queens Problem

- One partial solution is **extended to 4 queens**, but all of those are abandoned as well.



4 Queens Problem

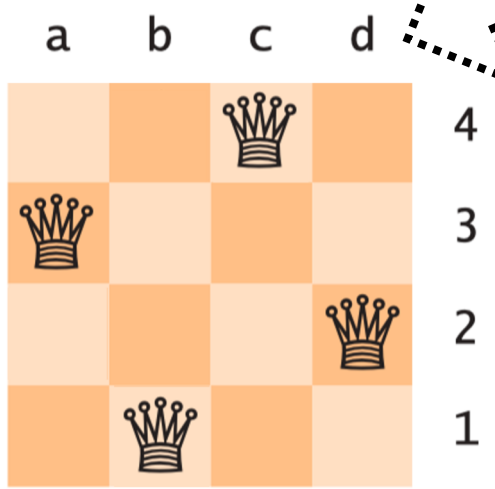
- Then the **algorithm backtracks**, giving up on a queen in position **a1**.
- instead **extending the solution** with the queen in position **b1**.



4 Queens Problem

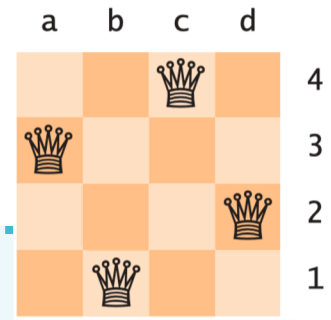
- Example implementation of the **backtracking algorithm** for N queen problem (part 1)
- Lets assume **N=4**. We define some **constant values** that specify the problem at hand.

```
COLUMNS = "abcd"
NUM_QUEENS = len(COLUMNS)
ACCEPT = 1
CONTINUE = 2
ABANDON = 3
```



4 Queens Problem

Example implementation (part 1)

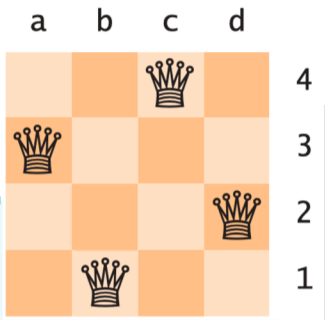



```
def extend(partial_sol):
    results = []
    row = len(partial_sol) + 1

    for column in COLUMNS:
        new_solution = list(partial_sol)
        new_solution.append(column + str(row))
        results.append(new_solution)

    return results
```

We implement an **extend** function that takes a **partial solution** and makes different copies of it.
Each copy gets **a new queen** in a different column.

4 Queens Problem

Example implementation (part 2)

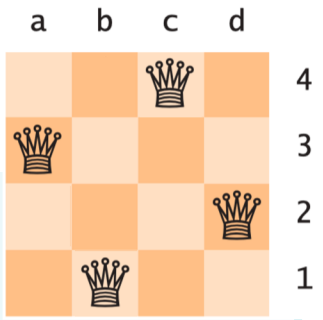



```
def examine(partial_sol):
    for i in range(len(partial_sol)):
        for j in range(i + 1, len(partial_sol)):

            if attacks(partial_sol[i],
                       partial_sol[j]):
                return ABANDON

    if len(partial_sol) == NUM_QUEENS:
        return ACCEPT
    else:
        return CONTINUE
```

The **examine** function checks whether two queens in a partial solution **can attack** each other or not.

4 Queens Problem

Example implementation (part 3)

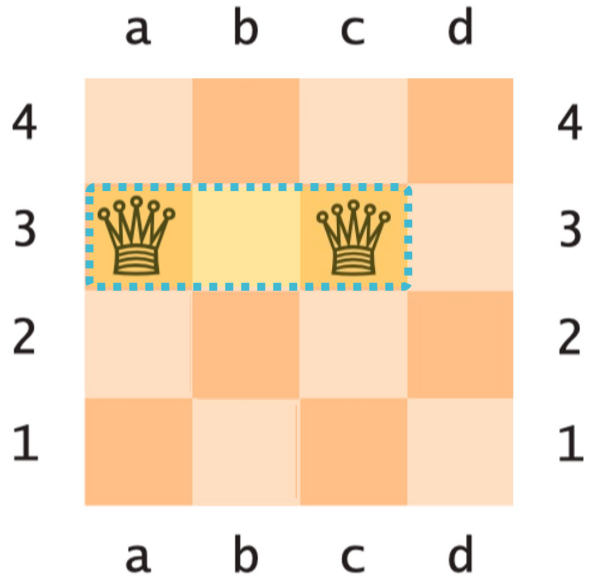



```
def attacks(p1, p2):
    column1 = COLUMNS.index(p1[0]) + 1
    row1 = int(p1[1])

    column2 = COLUMNS.index(p2[0]) + 1
    row2 = int(p2[1])

    return (row1 == row2 or
            column1 == column2 or
            abs(row1-row2) == abs(column1-column2))
```

The remaining challenge is to determine when **two queens** can actually **`attack`** each other. But how?

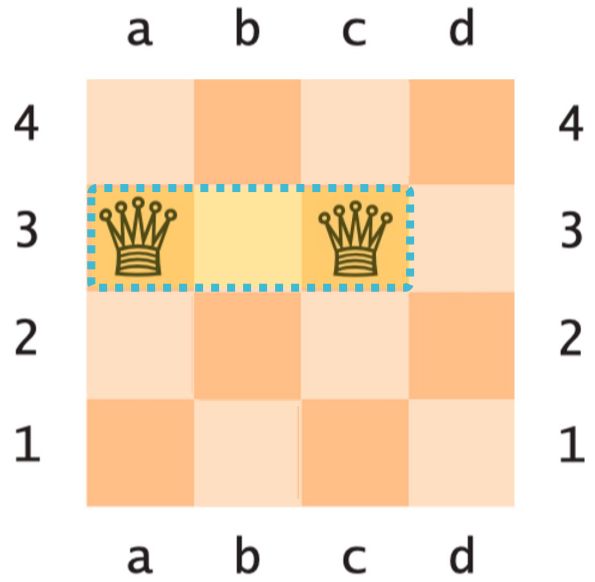
N Queens Problem

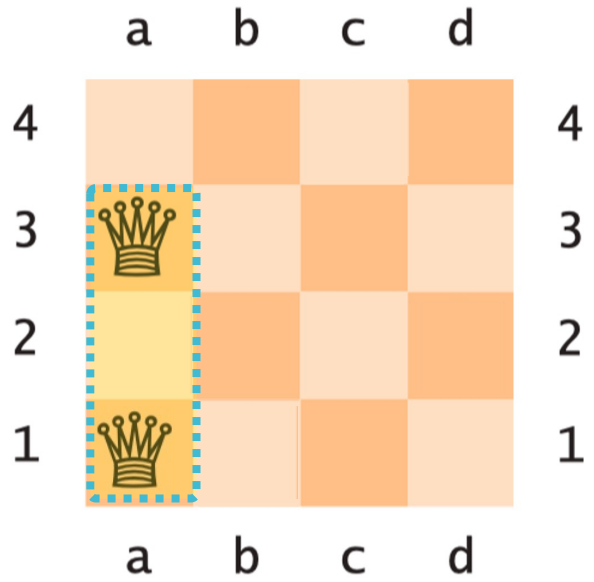
- The rule is that **2 queens** can **not** be on:
  - **the same row**



N Queens Problem

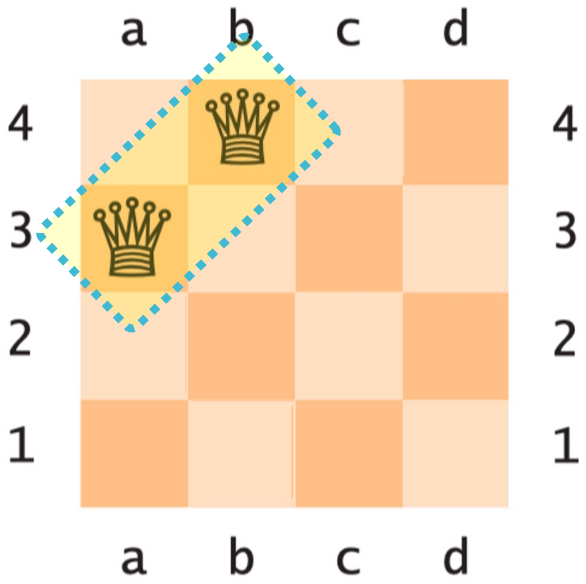
- Easy to check:
  - **the same row**
  - **the same column**
  - the same diagonal.

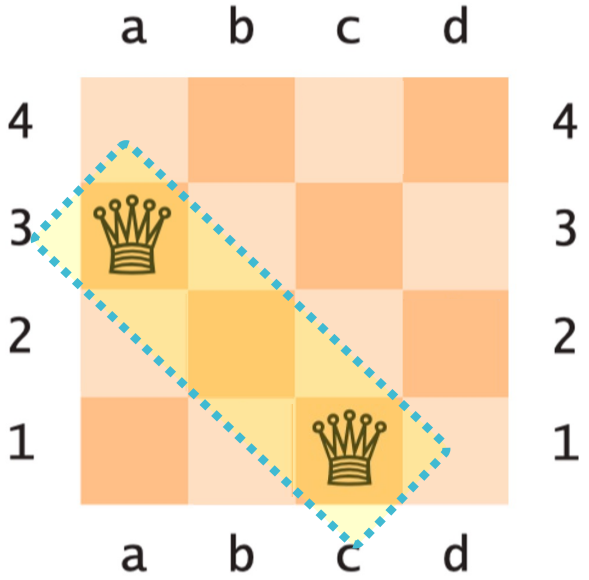




N Queens Problem

- Possible scenarios for:
  - the same row
  - the same column
  - **the same diagonal.**





N Queens Problem

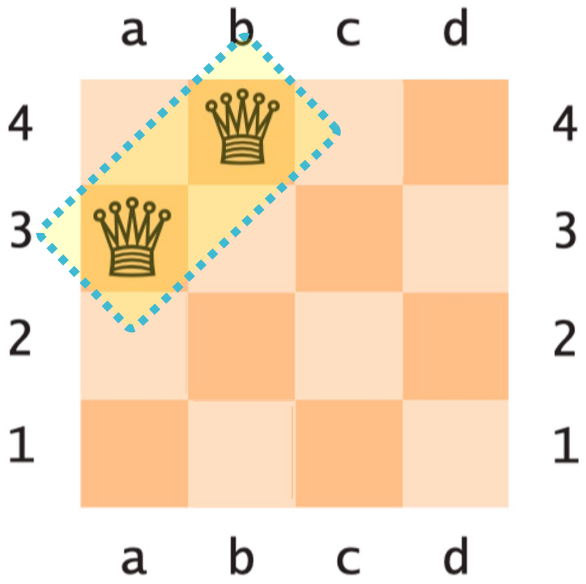
- What about **computing the slope** and check whether it is ±1.
- **This condition** can be simplified as follows:

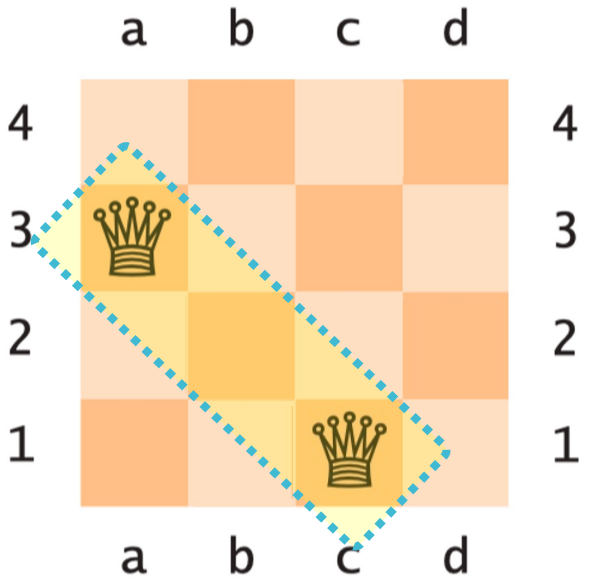
$$(\text{row}_2 - \text{row}_1)/(\text{column}_2 - \text{column}_1) = \pm 1$$

slope

$$\text{row}_2 - \text{row}_1 = \pm(\text{column}_2 - \text{column}_1)$$

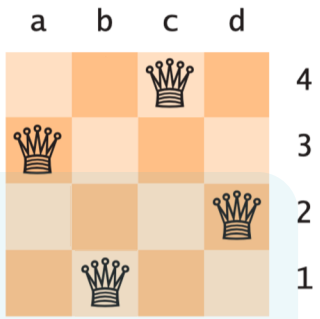
$$|\text{row}_2 - \text{row}_1| = |\text{column}_2 - \text{column}_1|$$





4 Queens Problem

Example implementation (part 4)




```
def attacks(p1, p2):
    column1 = COLUMNS.index(p1[0]) + 1
    row1 = int(p1[1])

    column2 = COLUMNS.index(p2[0]) + 1
    row2 = int(p2[1])

    return (row1 == row2 or
            column1 == column2 or
            abs(row1-row2) == abs(column1-column2))
```

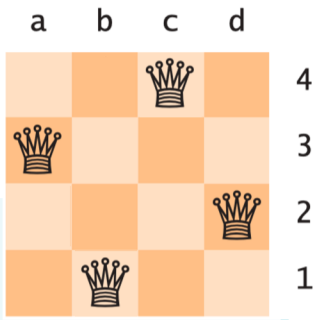
$$(\text{row}_2 - \text{row}_1)/(\text{column}_2 - \text{column}_1) = \pm 1$$

$$\text{row}_2 - \text{row}_1 = \pm(\text{column}_2 - \text{column}_1)$$

$$|\text{row}_2 - \text{row}_1| = |\text{column}_2 - \text{column}_1|$$

4 Queens Problem

Example implementation (part 5)




```
def solve(partial_sol):
    exam = examine(partial_sol)

    if exam == ACCEPT:
        print(partial_sol)

    elif exam != ABANDON:
        for p in extend(partial_sol):
            solve(p)
```

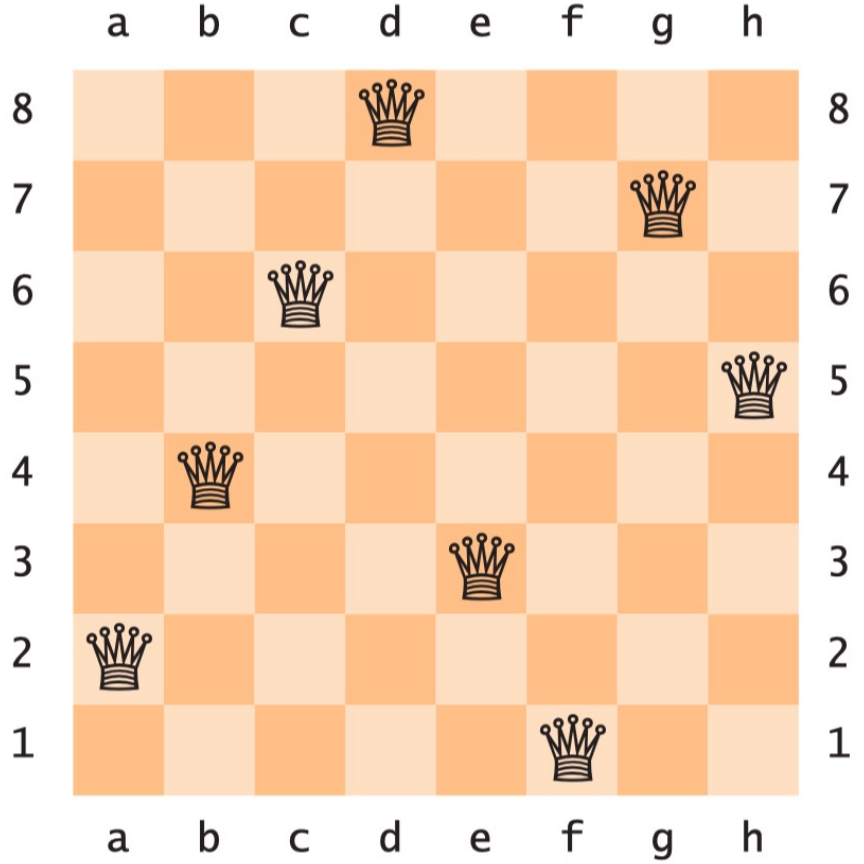
```
solve([])
```

[Output]:

```
['b1', 'd2', 'a3', 'c4']
['c1', 'a2', 'd3', 'b4']
```

Quiz

- Now try finding solution for **8 queens** in 8x8 chess.
- How **many solutions** are possible?



Answer

❖ Here are the solutions with the **8 queens**.

```
COLUMNS = "abcdefgh"
NUM_QUEENS = len(COLUMNS)
ACCEPT = 1
CONTINUE = 2
ABANDON = 3
```

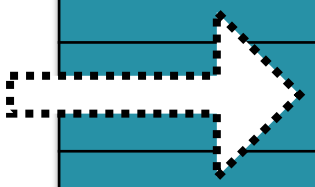
[Output]:

```
['a1', 'e2', 'h3', 'f4', 'c5', 'g6', 'b7', 'd8']
['a1', 'f2', 'h3', 'c4', 'g5', 'd6', 'b7', 'e8']
['a1', 'g2', 'd3', 'f4', 'h5', 'b6', 'e7', 'c8']
['a1', 'g2', 'e3', 'h4', 'b5', 'd6', 'f7', 'c8']
                    .  .  .
['h1', 'b2', 'e3', 'c4', 'a5', 'g6', 'd7', 'f8']
['h1', 'c2', 'a3', 'f4', 'b5', 'e6', 'g7', 'd8']
['h1', 'd2', 'a3', 'c4', 'f5', 'b6', 'g7', 'e8']
```

?

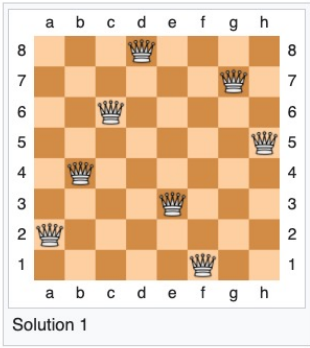
Answer

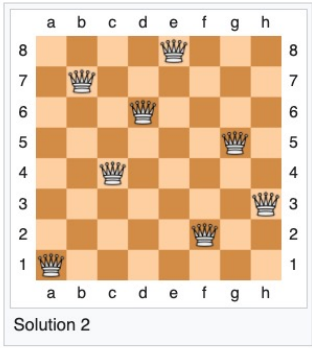
N	Num of Solutions
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
…	…

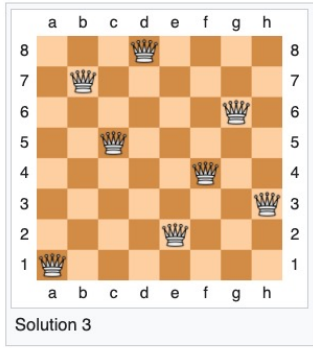


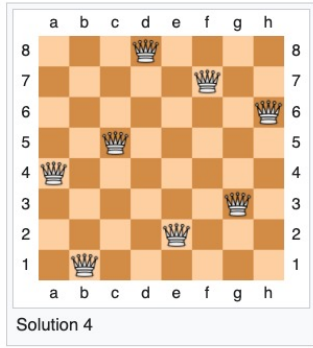
Answer

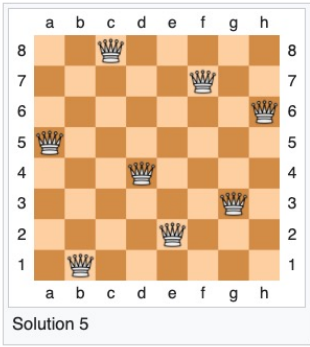
- Some of the example solutions

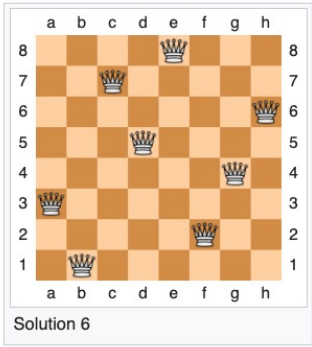

Solution 1

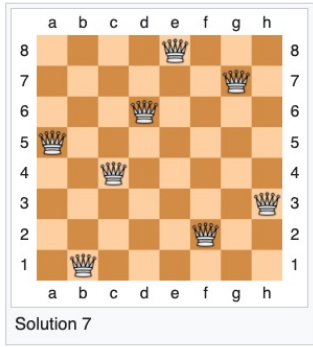

Solution 2

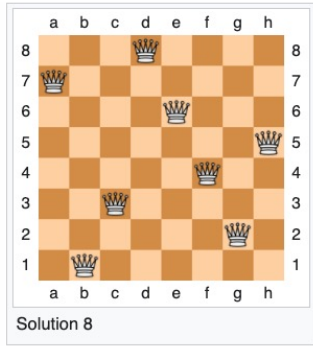

Solution 3

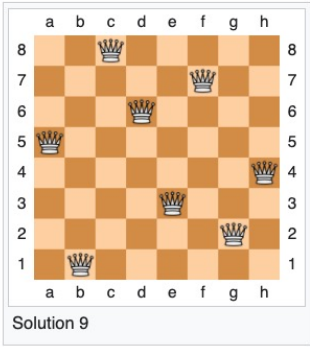

Solution 4

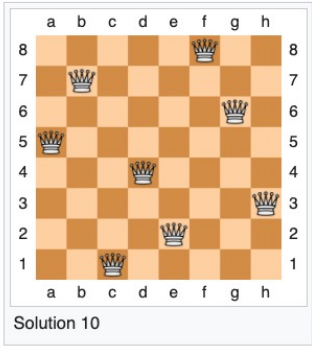

Solution 5

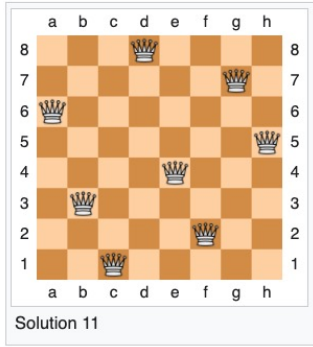

Solution 6

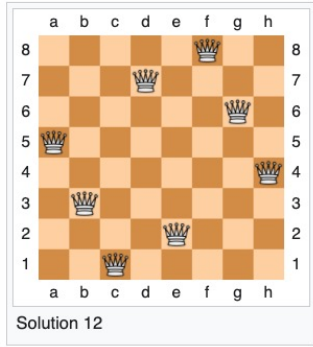

Solution 7


Solution 8


Solution 9


Solution 10

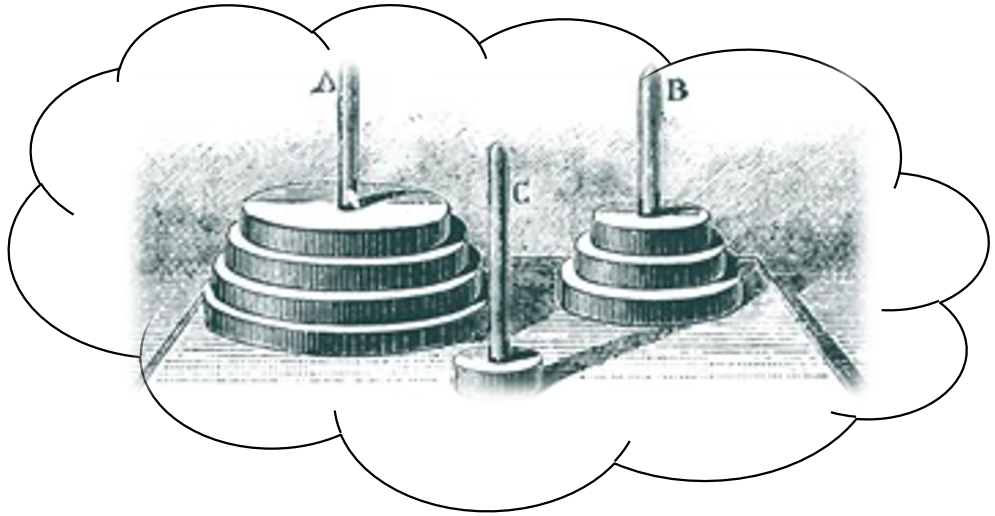

Solution 11


Solution 12

Tower of Hanoi

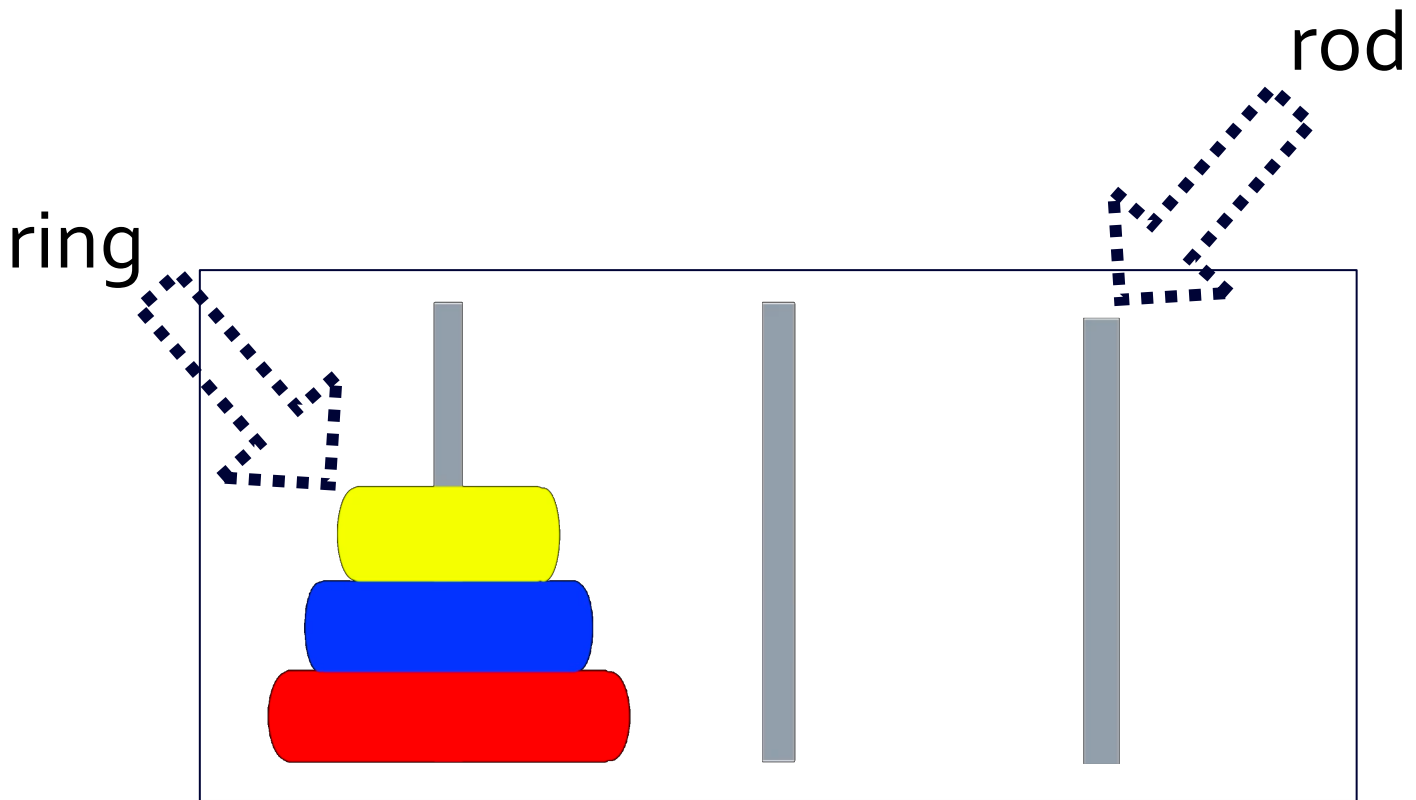
- **Tower of Hanoi** (or tower of Brahma) is a popular math puzzle**.**

- Invented by a French mathematician **Édouard Lucas** in the 19th century.



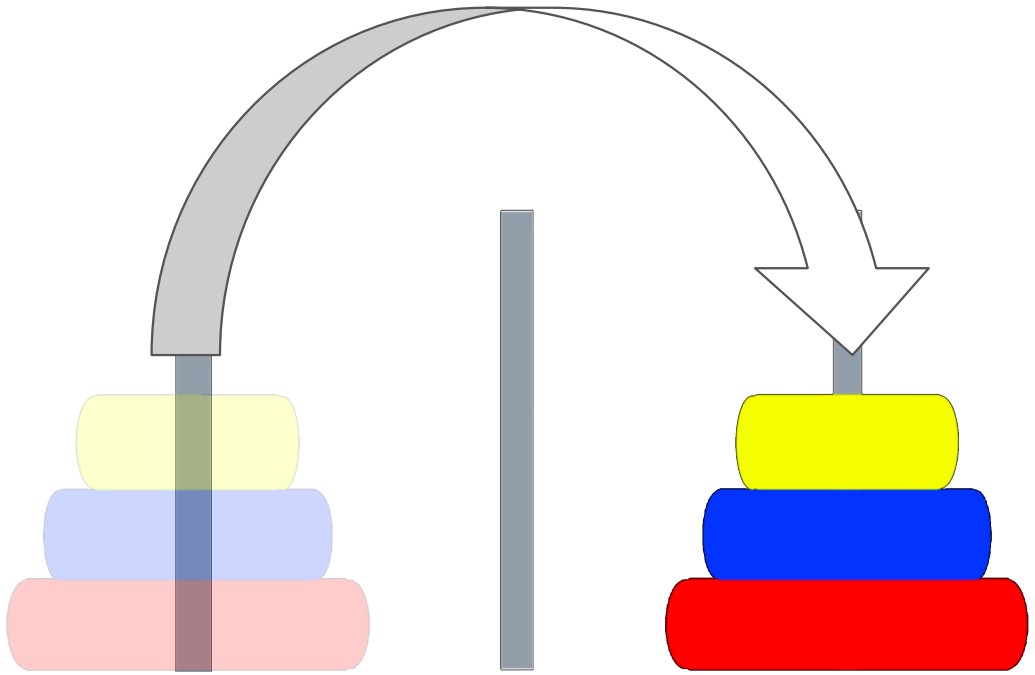
Tower of Hanoi

- Puzzle:
  - **Given: 3** rods and **n** rings (or disks)



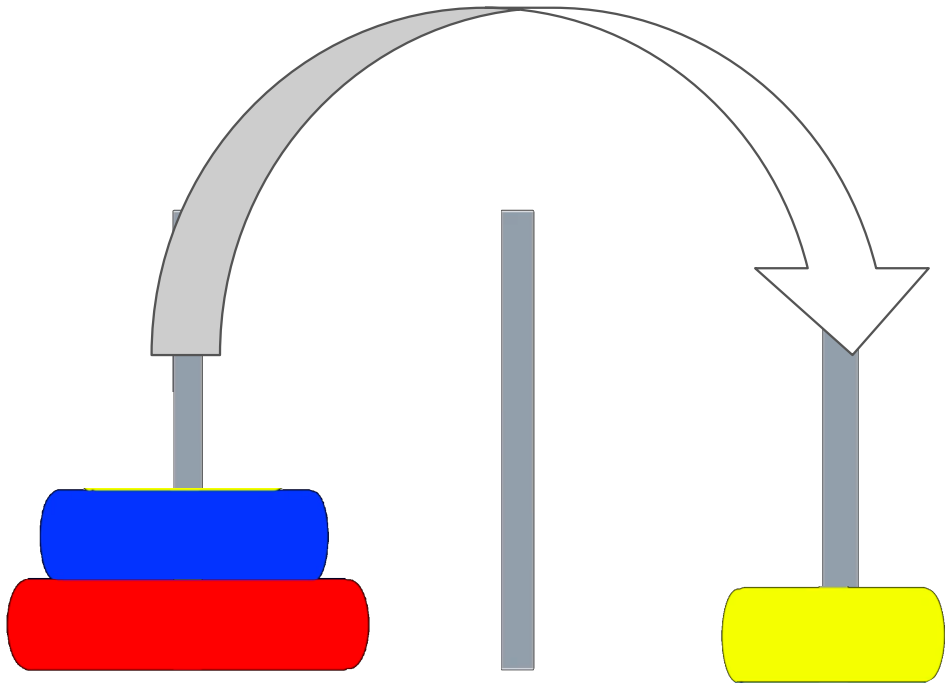
Tower of Hanoi

- Puzzle:
  - **Goal:** move the entire stack of rings from one rode to another rod



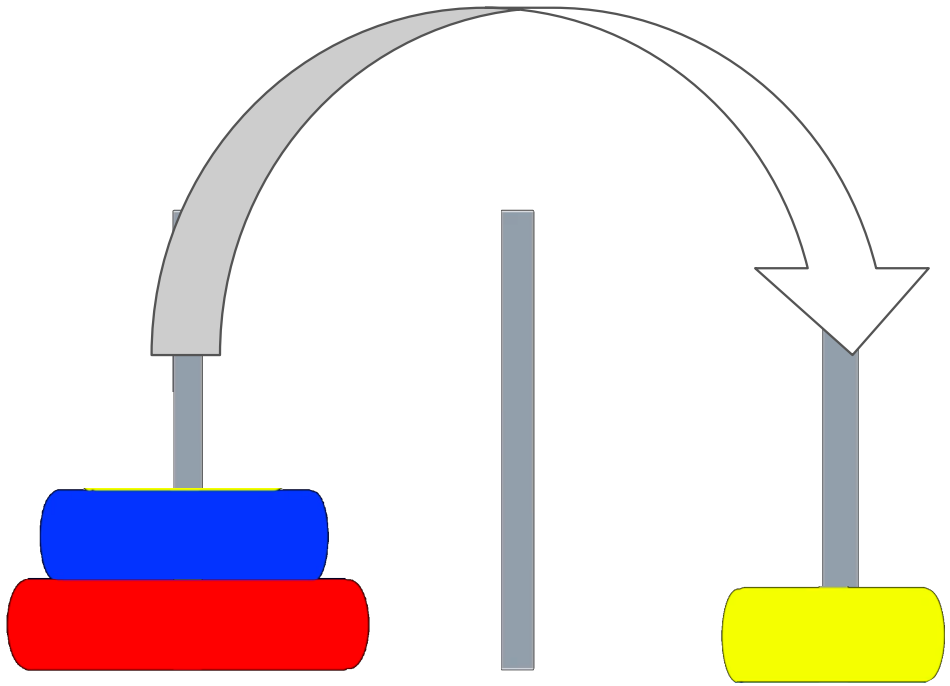
Tower of Hanoi

- Rules:
  a) Only **one ring** can be moved at a time



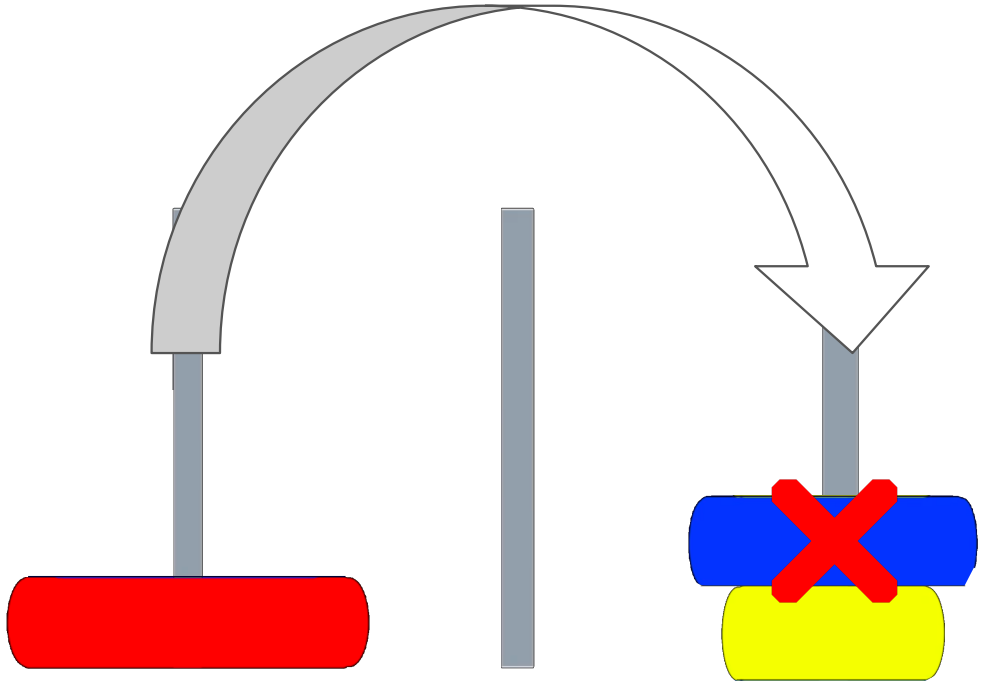
Tower of Hanoi

- Rules:
  a) Only **one ring** can be moved at a time
  b) A ring can only be moved if it is the **uppermost** ring on a stack



Tower of Hanoi

- Rules:
  a) Only **one ring** can be moved at a time
  b) A ring can only be moved if it is the **uppermost** ring on a stack
  c) **No ring** may be put on top of a smaller ring



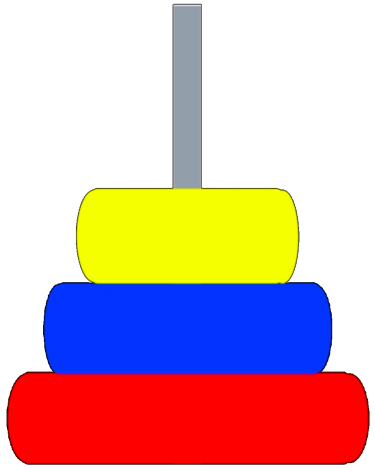
Tower of Hanoi

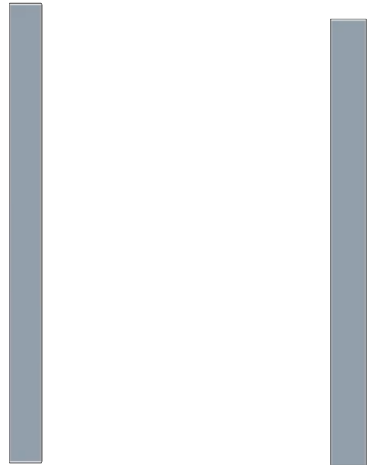
- Consider the problem of moving ***d*** **disks** from **rode *p*1 to rod *p*2**, where $p1$ and $p2$ are 1, 2, or 3, & $p1 \neq p2$.

- Because $1 + 2 + 3 = 6$, we can get the **index of the remaining rods** as $p3 = 6 - p1 - p2$.

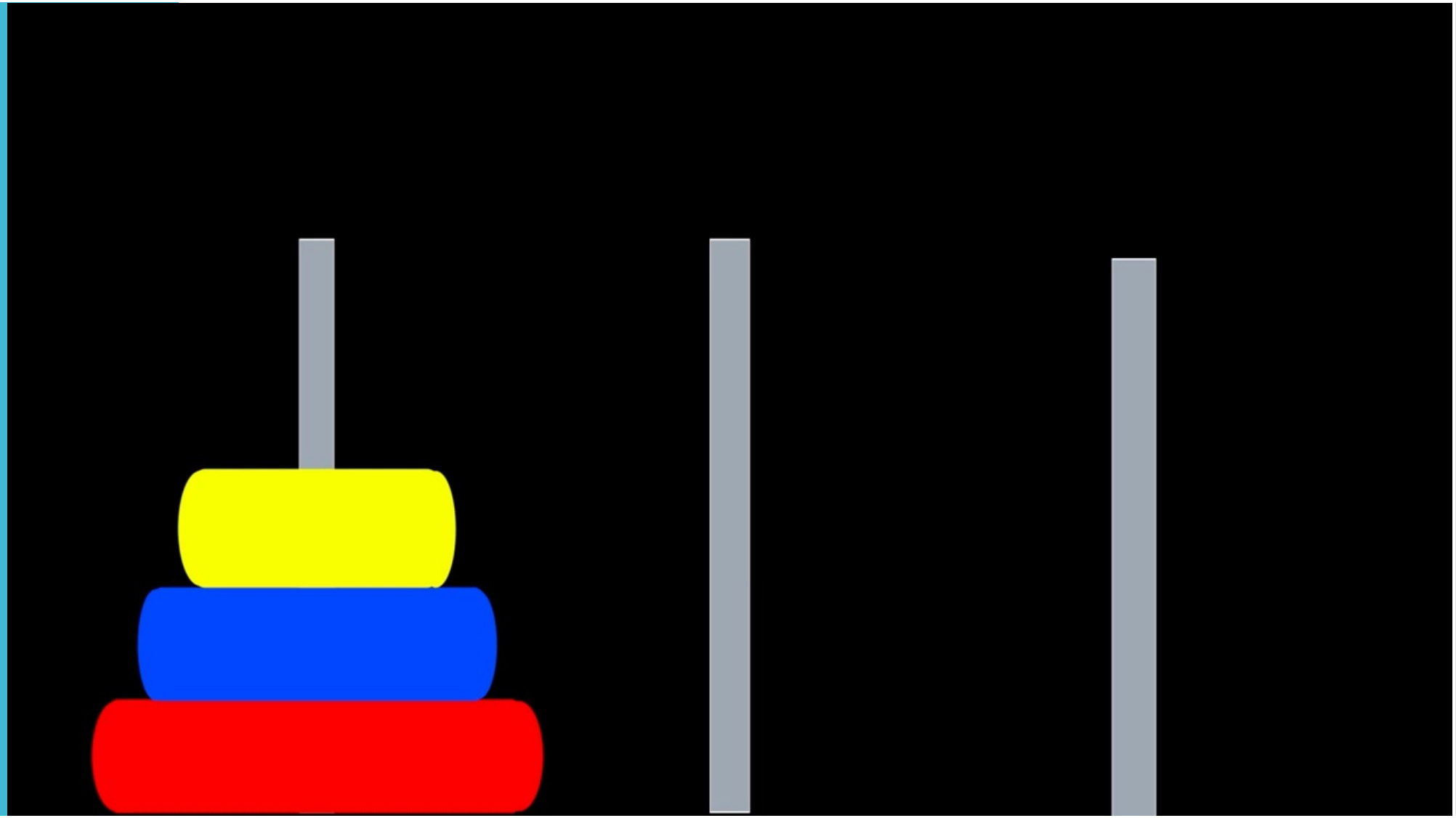
Quiz

- Write a class **`Tower`** that has:
  - A constructor method **`__init__()`** that creates 3 lists representing the rods and the rings in there
  - A method **`solve()`** that can solve the problem





Answer



ref : Youtube

Search

- **Search** is the process of selecting specific data from a **collection of data** based on specific criteria.

- Everyday life, we are performing many searches.

- **Examples:**
  - searching in **Google** to find web pages with certain words or phrases
  - searching in a **phonebook** to find a number.

Search

- **Sequence search:** finding an item within a sequence of data based on a search key.
- **Search key:** a unique value used to identify the data elements of a collection:
  - in collection of **simple types** (integers or reals), the values are the keys.
  - in collections of **complex types**, a specific data has to be identified as the key.

Search

Sequence search

```
def sequential_search(a_list, item):
    pos = 0
    found = False
    while pos < len(a_list) and not found:
        if a_list[pos] == item:
            found = True
        else:
            pos = pos + 1
    return found
```

Search

Sequence search

```
def sequential_search(a_list, item):
    pos = 0
    found = False
    while pos < len(a_list) and not found:
        if a_list[pos] == item:
            found = True
        else:
            pos = pos + 1
    return found
```

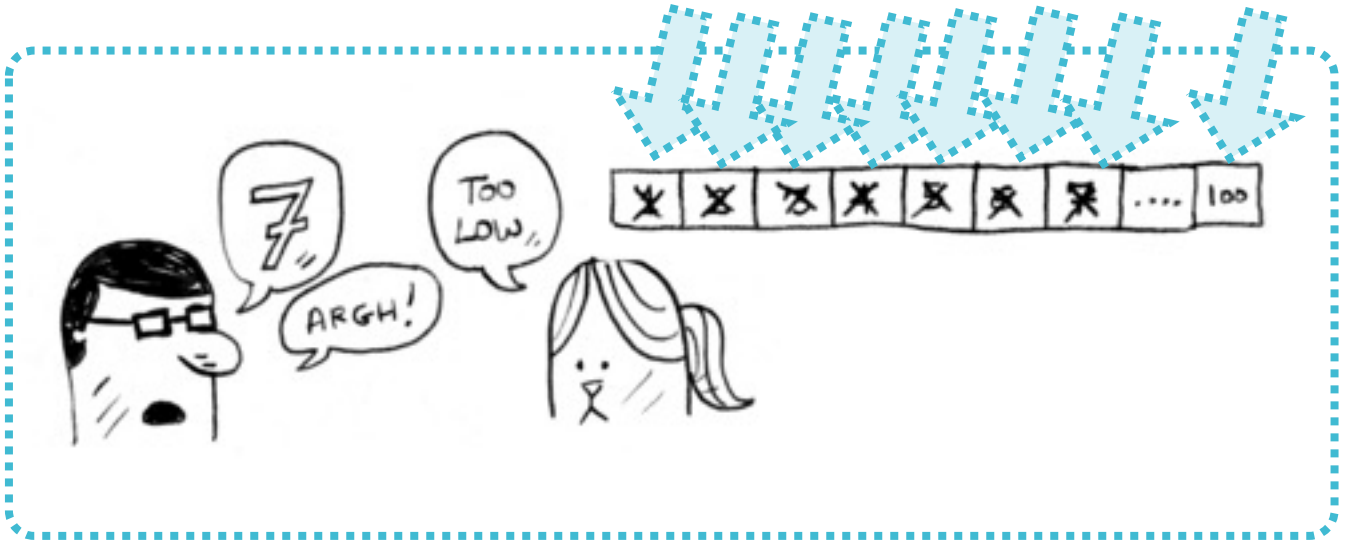
```
music_count = [156, 141, 35, 94, 88, 61, 111]
print(sequential_search(music_count, 88))
```

[Output:]

```
True
```

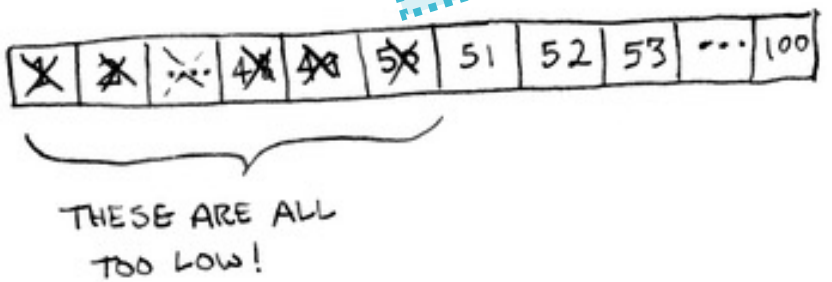
Search

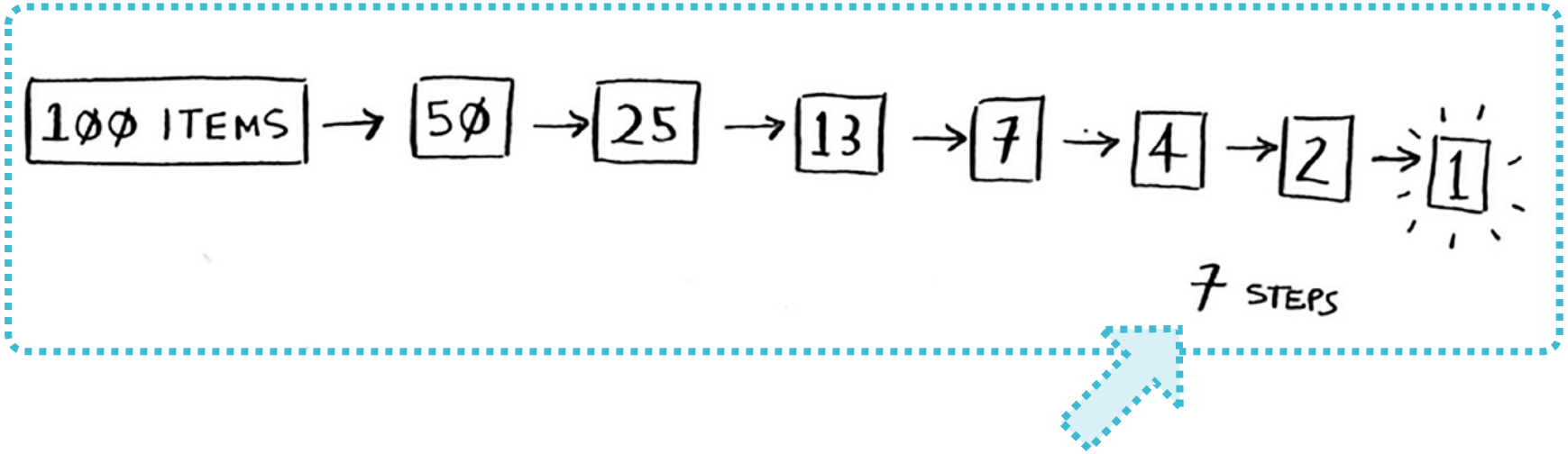
- Remember the **Search** game!
- This was the **basic solution**.



Search

- There was a **smarter solution**.





Search

Binary search (iterative)

```
def binary_search(a_list, item):
    begin = 0
    end = len(a_list) - 1
    found = False
    while begin <= end and not found:
        mid = (begin + end) // 2
        if a_list[mid] > item:
            end = mid - 1
        elif a_list[mid] < item:
            begin = mid + 1
        else:
            found = True
    return found
```

Binary search (Recursive)

Search

```
def binary_search(a_list, item):
    if len(a_list) == 0:
        return False
    else:
        mid = len(a_list) // 2
        if a_list[mid] > item:
            return binary_search(a_list[:mid], item)
        elif a_list[mid] < item:
            return binary_search(a_list[mid + 1:], item)
        else:
            return True
```

Search

❖ Binary search (Recursive)

```
def binary_search(a_list, item):
    if len(a_list) == 0:
        return False
    else:
        mid = len(a_list) // 2
        if a_list[mid] > item:
            return binary_search(a_list[:mid], item)
        elif a_list[mid] < item:
            return binary_search(a_list[mid + 1:], item)
        else:
            return True
```

```
music_count = [156, 141, 35, 94, 88, 61, 111]
music_count.sort()
print(binary_search(music_count, 88))
```

[Output:]

```
True
```

Quiz

1) Try to search for **88** in the **`music_count`** list list.

- **It will not find** it and will return False!
- Why does it happen? Can you fix it?

```
music_count = [156, 141, 35, 94, 88, 61, 111]
print(binary_search(music_count, 88))
```

[Output:]

```
False
```

Quiz

2) Suppose you have the following sorted list:

[3, 5, 6, **8**, 11, 12, 14, 15, 17, 18]

- Using recursive **binary search algorithm**, which option shows the correct sequence of comparisons to find the **key 8?**

a) 11, 5, 6, 8
b) 12, 6, 11, 8
c) 3, 5, 6, 8
d) 18, 12, 6, 8

Answer

1) Because the list is not sorted.

❖ Binary search only works with sorted lists.

```
music_count = [156, 141, 35, 94, 88, 61, 111]
music_count.sort()
print(binary_search(music_count, 88))
```

[Output:]

```
True
```

Quiz

2) **Answer:**

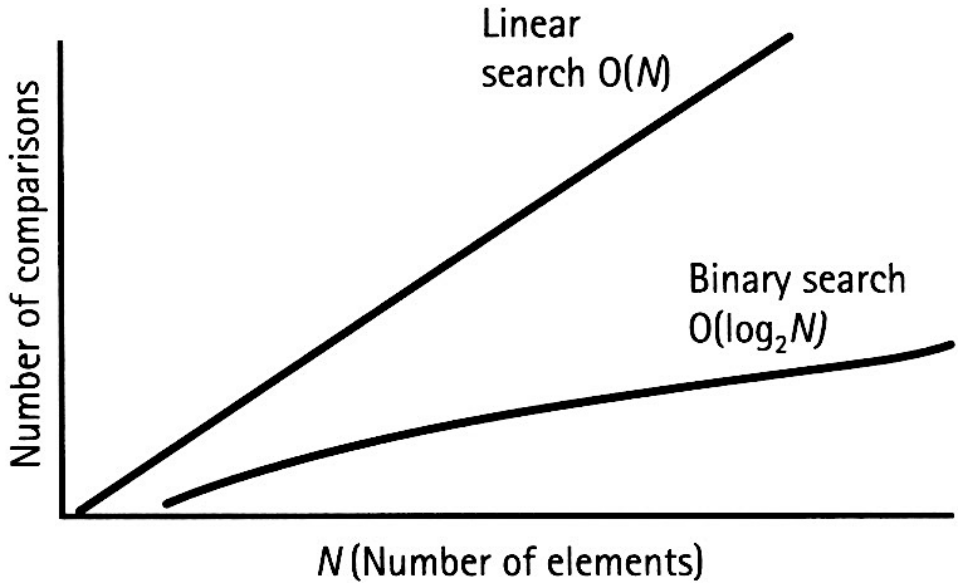
a) 11, 5, 6, 8

b) 12, 6, 11, 8

c) 3, 5, 6, 8

d) 18, 12, 6, 8

Search



	Best case	Average case	Worse case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

Next Lesson

- **Hashing**