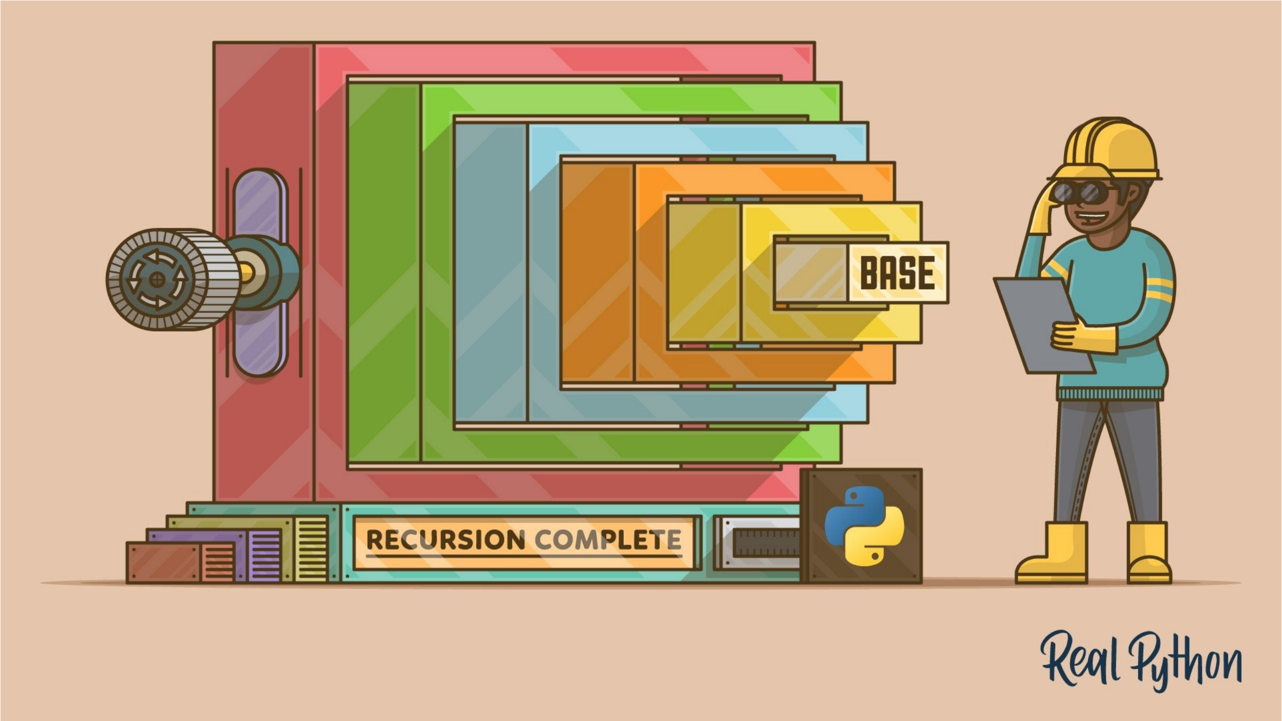
Recursion

Solving problems by dividing them in smaller, similar problems

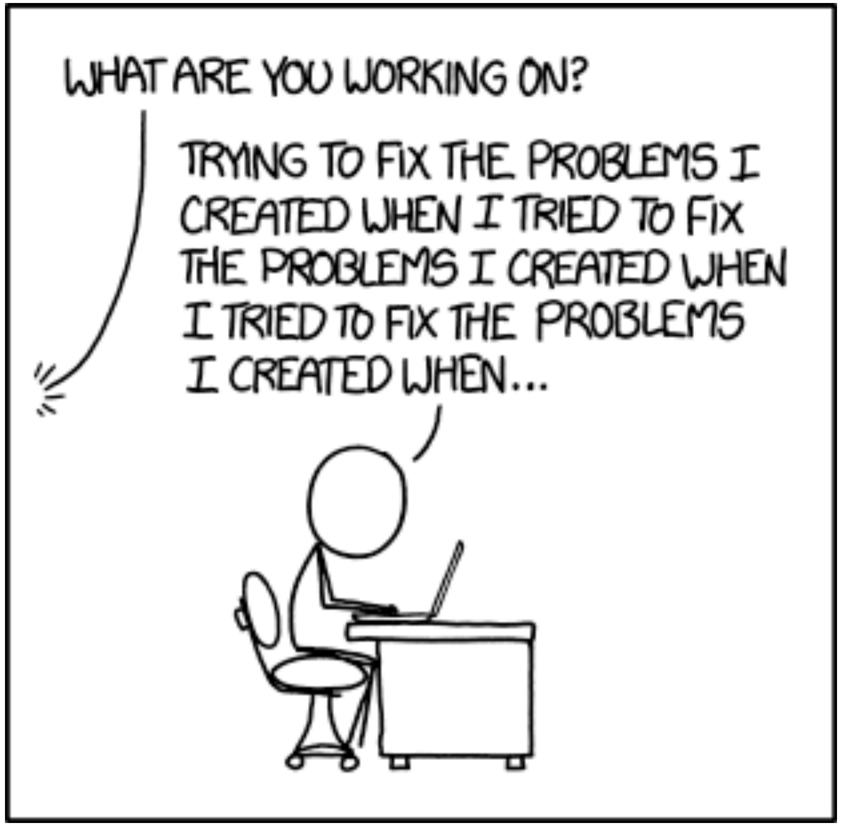
Fulvio Corno
Giuseppe Averta
Carlo Masone
Francesca Pistilli



Summary

- Introduction (definition, call stack, execution context, recursion limit)
  - Countdown, factorial, binomial, palindromes
- Iterative vs. recursive algorithms
  - Recursive data structures, nested lists
- Memoization/Caching (manually or using @lru_cache)
  - Fibonacci
- Sorting and Search algorithms
  - Quicksort, Dichotomic search
- Recursion applications
  - Recursive data structures, divide et impera, exploration
- Design tips
- Exercises
- Try it at home

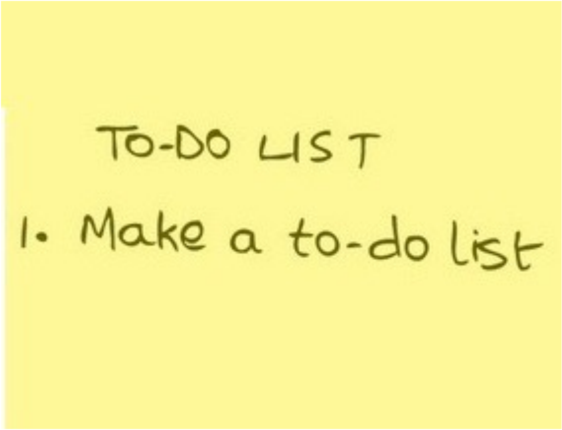
INTRODUCTION



Definition

- A **recursive** definition is one in which the defined term appears in the definition itself.

Your **ancestors** = (your parents) + (your parents' **ancestors**)



Definition

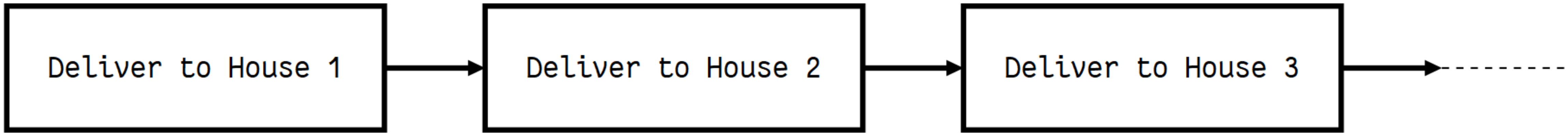
- In programming, recursion refers to a coding technique in which a function calls itself.
- A **method** (or a procedure or a function) is defined as **recursive** when:
  - Inside its definition, we have **a call to the same method** (procedure, function)
  - Or, inside its definition, there is a call to another method that, directly or indirectly, calls the method itself
- An algorithm is said to be recursive when it is based on recursive methods (procedures, functions)

Definition


Per capire la ricorsione prima devi capire la ricorsione
@vitadainformatici

Example: Santa Claus deliveries

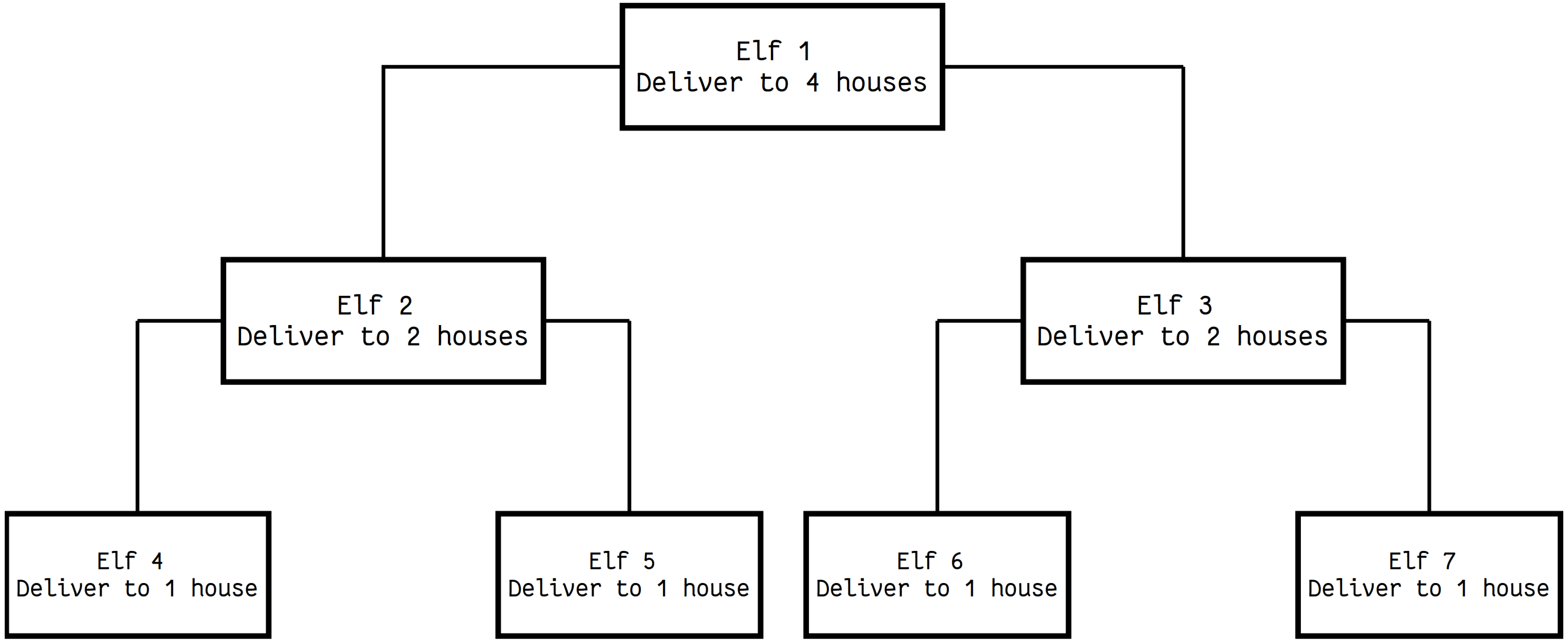
- It’s Christmas time, and Santa Claus has a list of houses to visit to deliver presents
- He could loop through the houses, **iteratively**



Iterative Present Delivery

Example: Santa Claus deliveries

- But it would probably be more effective to divide the work in chunks, among different workers



Recursive Present Delivery

Example: Santa Claus deliveries

```
houses = [“Eric’s house”, “Kenny’s house”, “Kyle’s house”, “Stan’s house”]
```

```
def deliver_presents_iteratively():
    for house in houses:
        deliver_to(house)
```

```
def deliver_presents_recursively(houses):
    if len(houses) == 1:
        house = houses[0]
        deliver_to(house)
    else:
        mid = len(houses) // 2
        first_half = houses[:mid]
        second_half = houses[mid:]
        deliver_presents_recursively(first_half)
        deliver_presents_recursively(second_half)
```

How far can we go with recursions

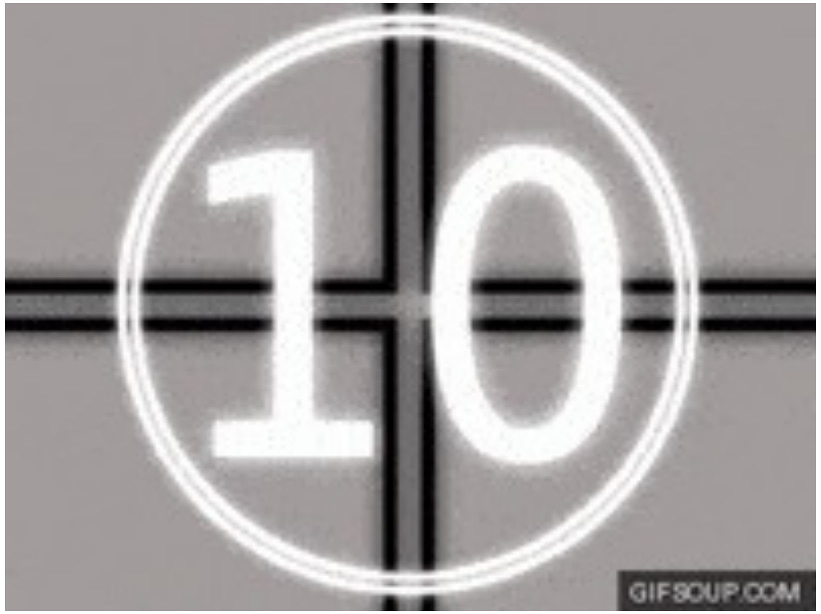
What happens executing this?

```python
def function():
    x = 10
    function()
```

- This would go indefinitely, in theory. In practice, we would incur in a `RecursionError`
- We can check how many iterations we can do using `sys.getrecursionlimit()`

Example: Countdown

- Let's try writing down a countdown, recursively



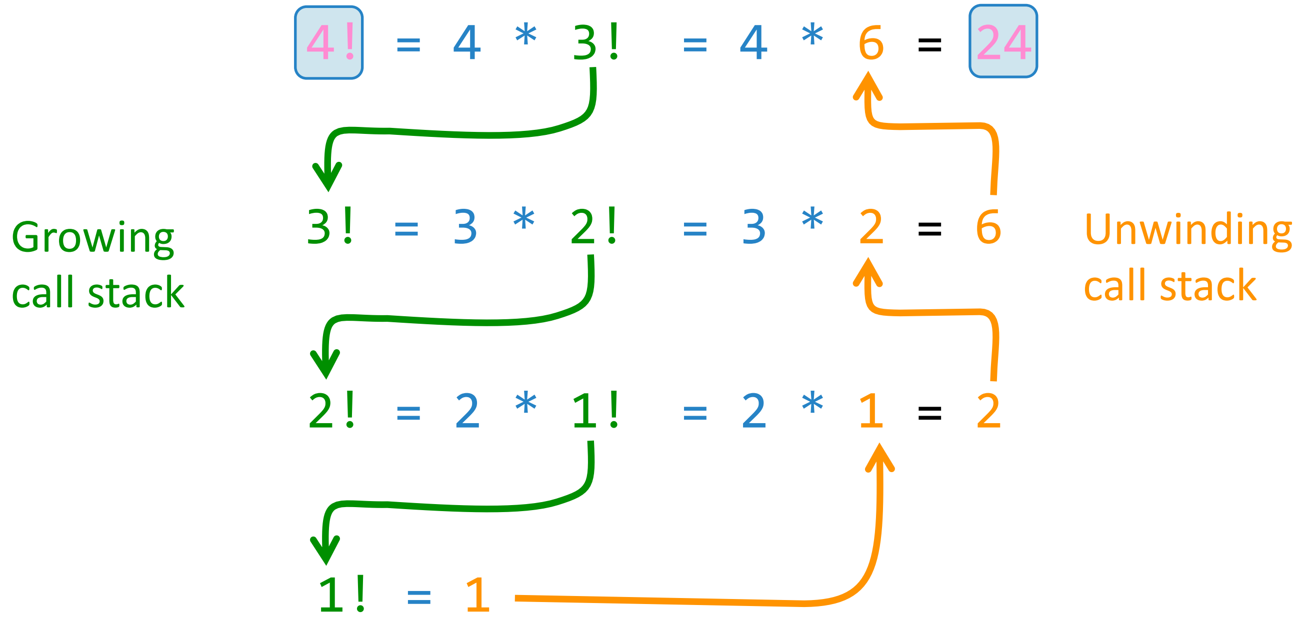
Example: Factorial

Factorial definition

$$n! = 1 \times 2 \times \ldots \times n$$

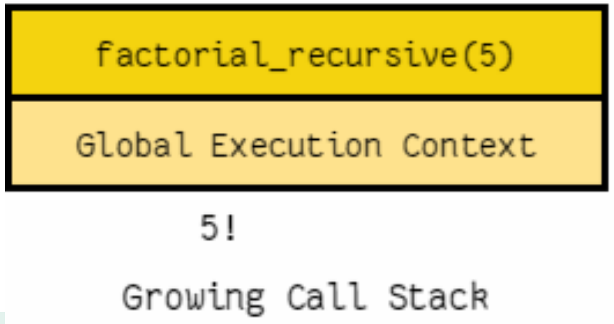
Equivalent recursive expression

$$n! = \begin{cases} 1 & \text{for } n = 0 \text{ or } n = 1 \\ n \times (n-1)! & \text{for } n \geqslant 2 \end{cases}$$



Example: Factorial

- We are going to implement this as a method that calls itself.
- From the global context, that first invokes this method, the call stack will grow until reaching the banal case (1!) and then the call stack will unwind, by passing the results back until reaching the global context



Example: Binomial

- Compute the Binomial Coefficient (n m) exploiting the recurrence relations (derived from Tartaglia's triangle):

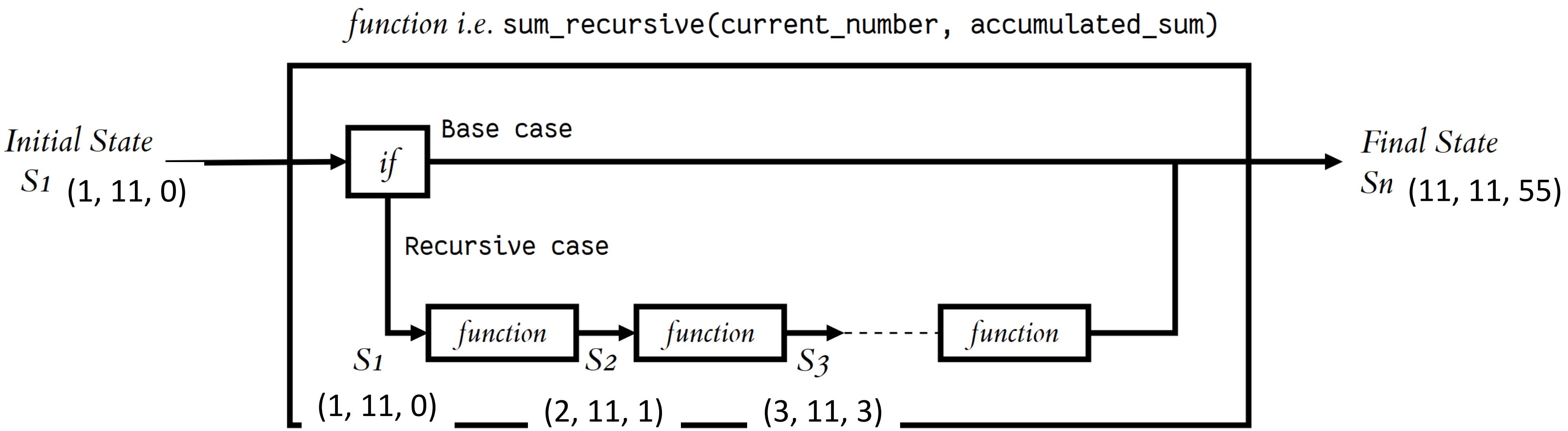
$$\begin{cases} \binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m} \\ \binom{n}{n} = \binom{n}{0} = 1 \\ 0 \le n, \quad 0 \le m \le n \end{cases}$$

Maintaining the state

- Each recursive call has its own execution context
- To maintain state, from one recursion level to another, one can:
  - Thread the state through each recursive call so that the current state is part of the current call's execution context
  - Encapsulate the recursive function within a class, using a class attribute to keep the state information
  - Keep the state in global scope 

Maintaining the state

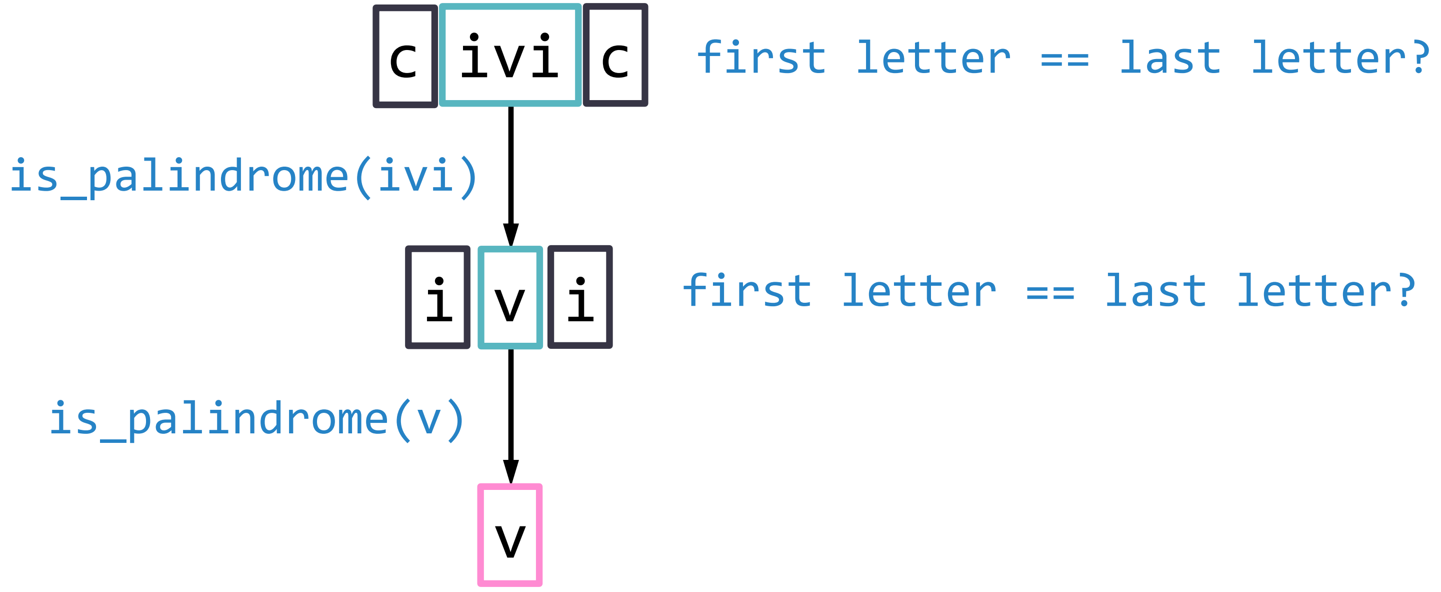
```python
def sum_recursive(level, N, accumulated_sum):
    if current_number == N:
        return accumulated_sum
    else:
        return sum_recursive(level + 1, N, accumulated_sum + level)
```



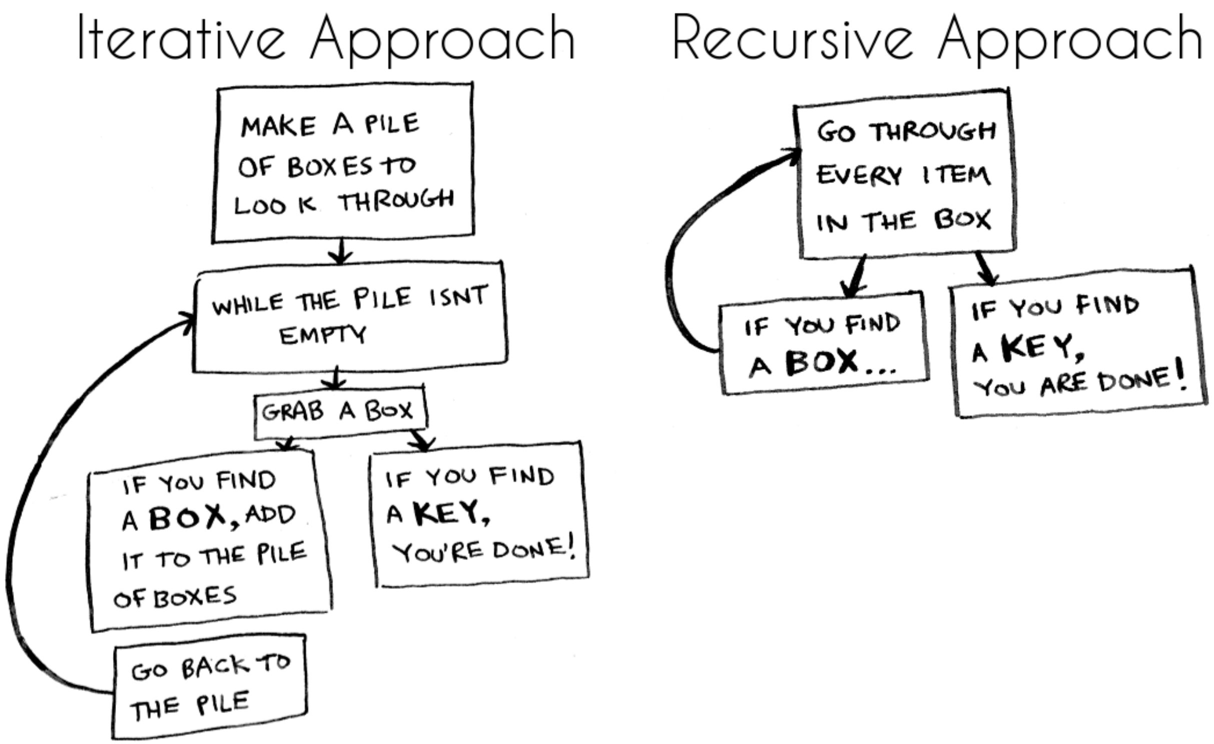
Control flow with threaded state

Example: Palindrome checking

- Write a recursive program to detect if a word is a palindrome or not
- A **palindrome** is a word that reads the same backward as it does forward (e.g., racecar, level, kayak, civic)



ITERATION VS. RECURSION



Iteration vs. Recursion

- Every **recursive** program can **always** be implemented in an **iterative** manner
- The best solution, in terms of efficiency and code clarity, depends on the problem

Why recursion?

Recursion comes handy in quite a few cases

- Divide et impera
- Systematic exploration/enumeration
- Handling recursive data structures

Motivation

- Many problems lend themselves, naturally, to a recursive description:
  - We define a method to solve sub-problems like the initial one, but smaller
  - We define a method to combine the partial solutions into the overall solution of the original problem

Gaius Julius Caesar

Recursion

- **Divide et Impera**

- Split a problem $\mathcal{P}$ into $\{\mathcal{Q}_i\}$ where $\mathcal{Q}_i$ are still complex, yet *simpler* instances of the same problem.
- Solve $\{\mathcal{Q}_i\}$, then merge the solutions
- Merge & split must be "simple"
- A.k.a., *Divide 'n Conquer*

- **Exploration**

- Systematic procedure to enumerate all possible solutions
- Solutions (built stepwise)
  - Paths
  - Permutations
  - Combinations
- Divide et Impera, by "dividing" the possible solutions

Divide et Impera – Divide and Conquer

```
def solve (problem):
    sub_problems = divide(problem)
    sub_solutions = []

    for sub_problem in sub_problems:
      sub_solutions.append(solve(sub_problem))

   solution = combine(sub_solutions)
   return solution


solution = solve(problem)
```

Divide et Impera – Divide and Conquer

```
def solve (problem):
    sub_problems = divide(problem)
    sub_solutions = []

    for sub_problem in sub_problems:
      sub_solutions.append(solve(sub_problem))

   solution = combine(sub_solutions)
   return solution

solution = solve(problem)
```

“a” sub-problems, each “b” times smaller than the initial problem

recursive call

How to stop recursion?

- Recursion **must not** be infinite
  - Any algorithm must always terminate!
- After a sufficient nesting level, sub-problems become so small (and so easy) to be solved:
  - Trivially (ex: sets of just one element, or zero elements)
  - Or, with methods different from recursion

Warnings

- Always remember the “termination condition”
- Ensure that all sub-problems are **strictly** “smaller” than the initial problem

Divide et Impera – Divide and Conquer

```
def solve (problem):
    if is_trivial(problem):
        solution = solve_trivial(problem)
        return solution
    else:
        sub_problems = divide(problem)
        sub_solutions = []
        for sub_problem in sub_problems:
            sub_solutions.append(solve(sub_problem))
        solution = combine(sub_solutions)
        return solution
```

check termination

do recursion

Exploration

- Explore (**S**) {
  - List<Step> steps = **PossibleSteps** (Problem, **S**) ;
  - for (each **p** in steps) {
    - **S**.**Do** (**p**)
    - **Explore** (**S**) ;
    - **S**.**Undo** (**p**) ;
  - }
- }

Exploration

The "status" of the problem

- Explore (**S**) {
  - List<Step> steps = **PossibleSteps** (Problem, **S**) ;
  - for (each **p** in steps) {
    - **S**.Do (**p**)
    - **Explore** (**S**) ;
    - **S**.Undo (**p**) ;
  - }
- }

Local variable

"Try" the step

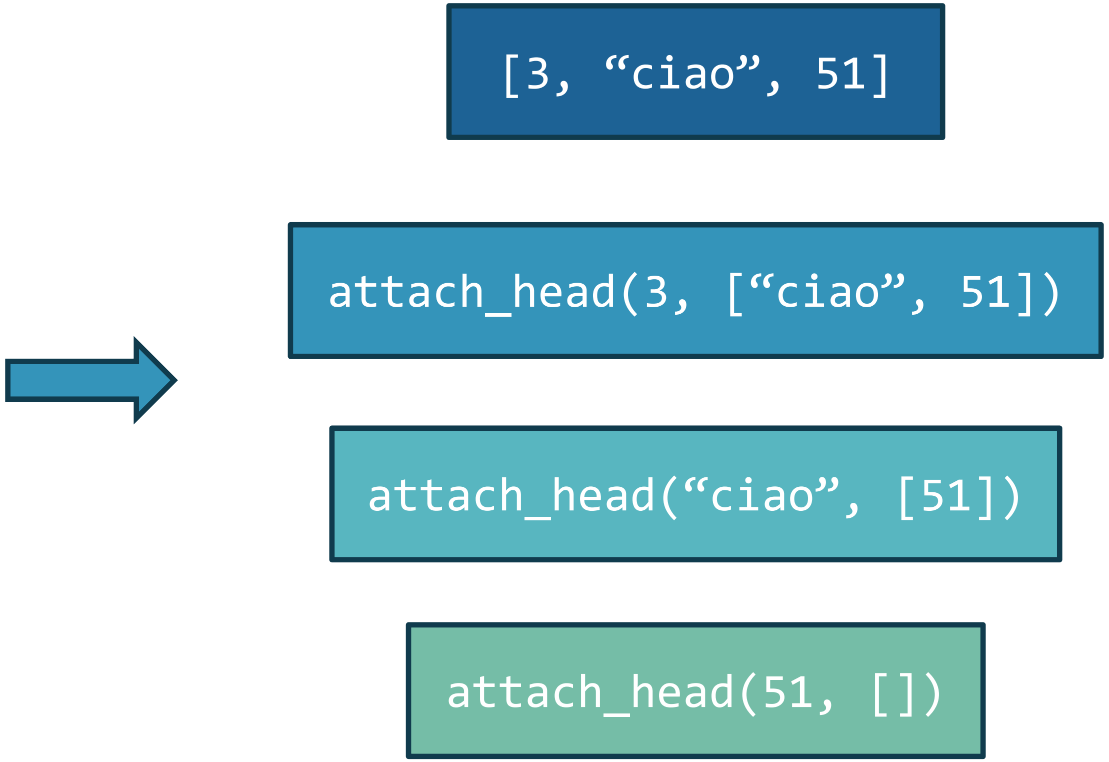
Recursion

Backtrack!

Recursive data structures

- A data structure is recursive if it can be defined in terms of a smaller version of itself.
- Example: list

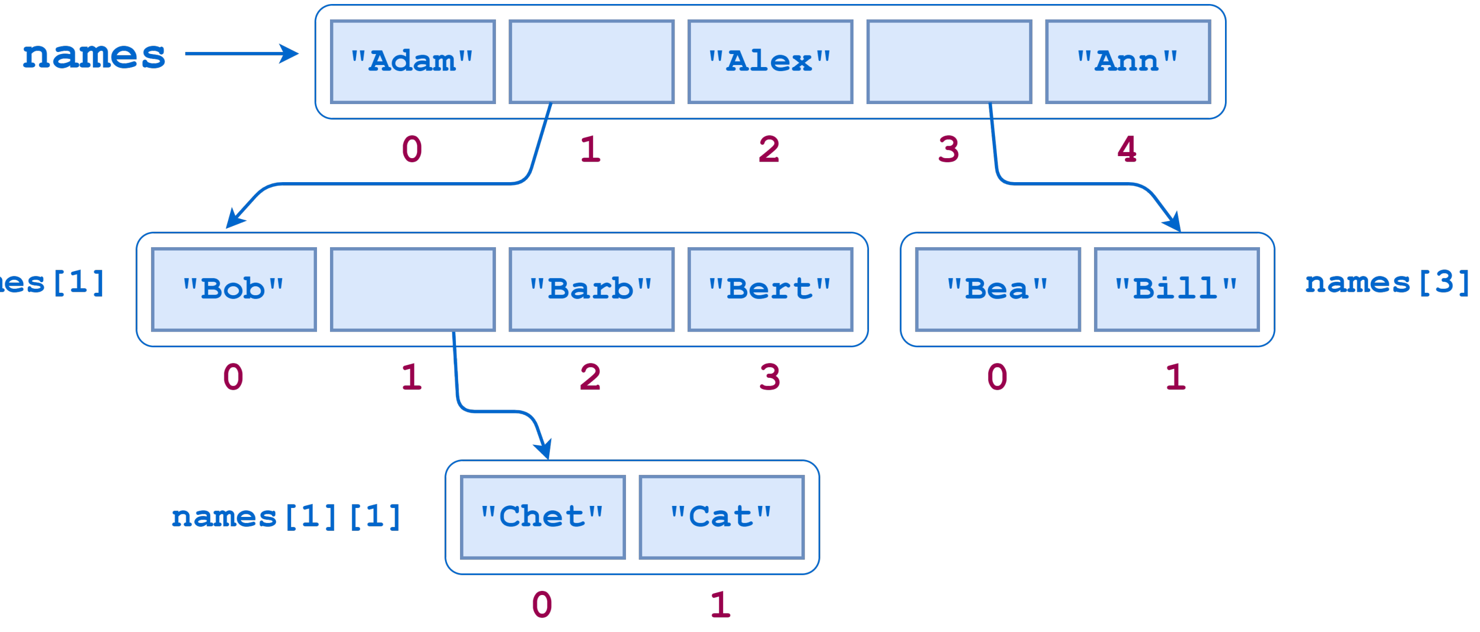
```python
def attach_head(element, input_list):
    return [element] + input_list
```



Example: nested list

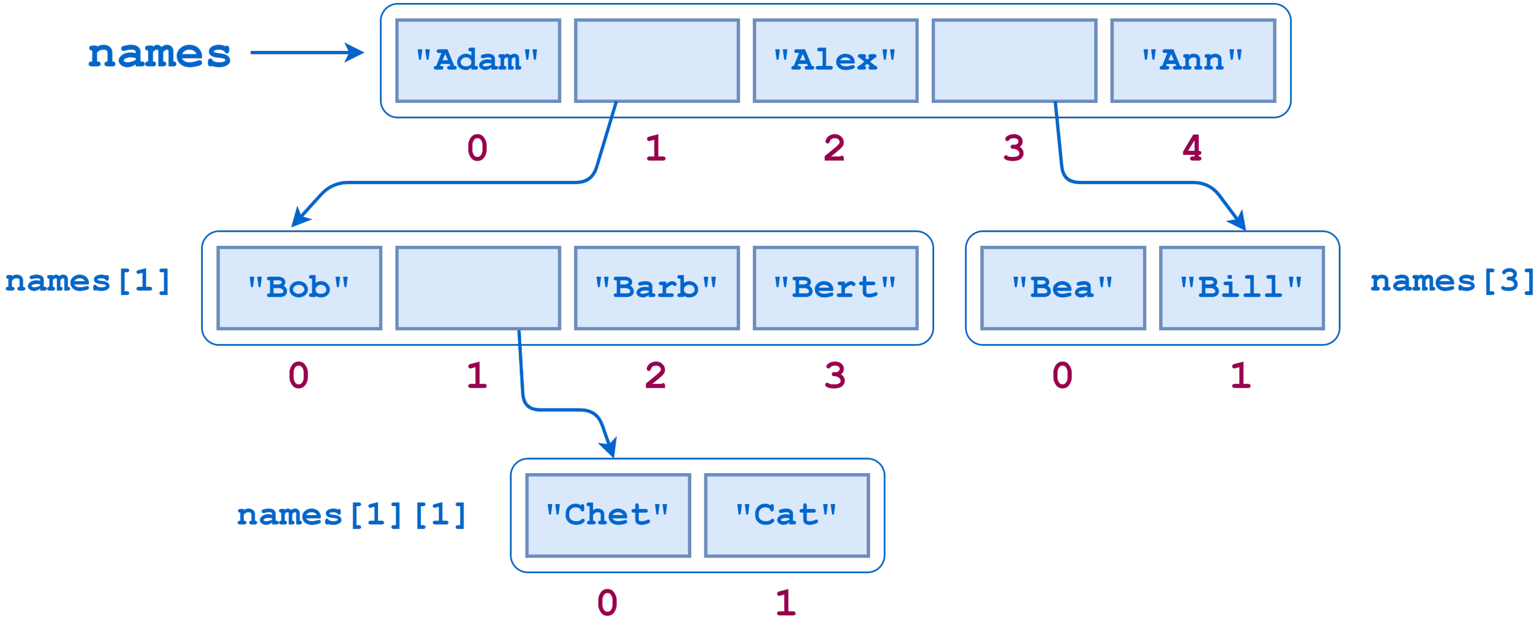
- Assume having a nested list, and having to count the leaf nodes.

```
names = ['Adam', ['Bob', ['Chet', 'Cat'], 'Barb', 'Bert'], 'Alex', ['Bea', 'Bill'], 'Ann']
```



Example: nested list

Let’s implement this method recursively!



Example: nested list

- The same functionality may also be implemented non-recursively.
  - Loop through the elements of a certain level of a list
  - Whenever a sub-list is encountered, <u>save the state of the current level</u> (count, list), and keep counting the elements of that level, until finished (while loop)

Example: nested list

```python
def count_leafs_iterative(item_list):
    count = 0
    stack = []
    current_list = item_list
    i = 0

    while True:
        if i == len(current_list):
            if current_list == item_list:
                return count
            else:
                current_list, i = stack.pop()
                i += 1
                continue

        if isinstance(current_list[i], list):
            stack.append([current_list, i])
            current_list = current_list[i]
            i = 0
        else:
            count += 1
            i += 1
```

Loop through all the elements in the list

Keep track of all the levels not yet completed

Keep track of all the partial result

Example: nested list

Recursive version

```python
def count_leaf_items(item_list):
    """Recursively counts and returns the
       number of leaf items in a (potentially
       nested) list.
    """
    count = 0
    for item in item_list:
        if isinstance(item, list):
            count += count_leaf_items(item)
        else:
            count += 1

    return count
```

Iterative version

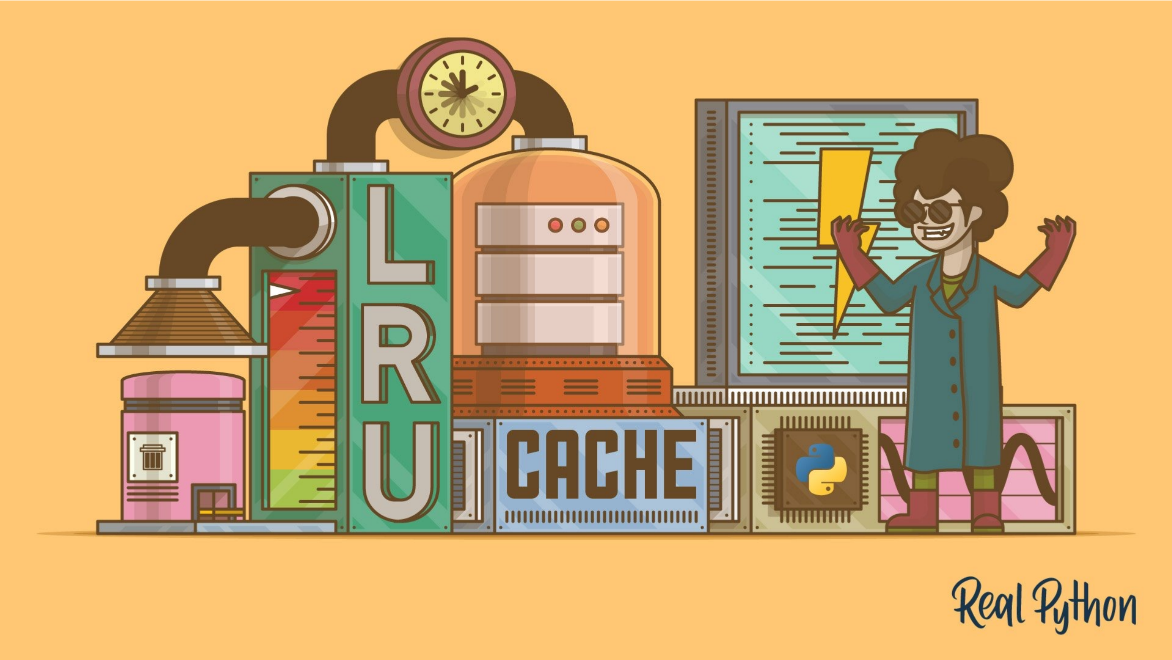
```python
def count_leaf_items(item_list):
    """Non-recursively counts and returns the
       number of leaf items in a (potentially
       nested) list.
    """
    count = 0
    stack = []
    current_list = item_list
    i = 0

    while True:
        if i == len(current_list):
            if current_list == item_list:
                return count
            else:
                current_list, i = stack.pop()
                i += 1

        if isinstance(current_list[i], list):
            stack.append([current_list, i])
            current_list = current_list[i]
            i = 0

        count += 1
        i += 1
```

IMPROVING EFFICIENCY



Recursion and efficiency

- How can we improve the runtime efficiency of our recursive method?
  - Use appropriate data structures (typically negligible improvements on small problems)
  - **Skip recursion threads that do not yield results** (can bring massive improvements)
  - **Cache intermediate results**, if the corresponding sub-problem is encountered multiple times (improvements depend on the problem, there is a memory cost.)

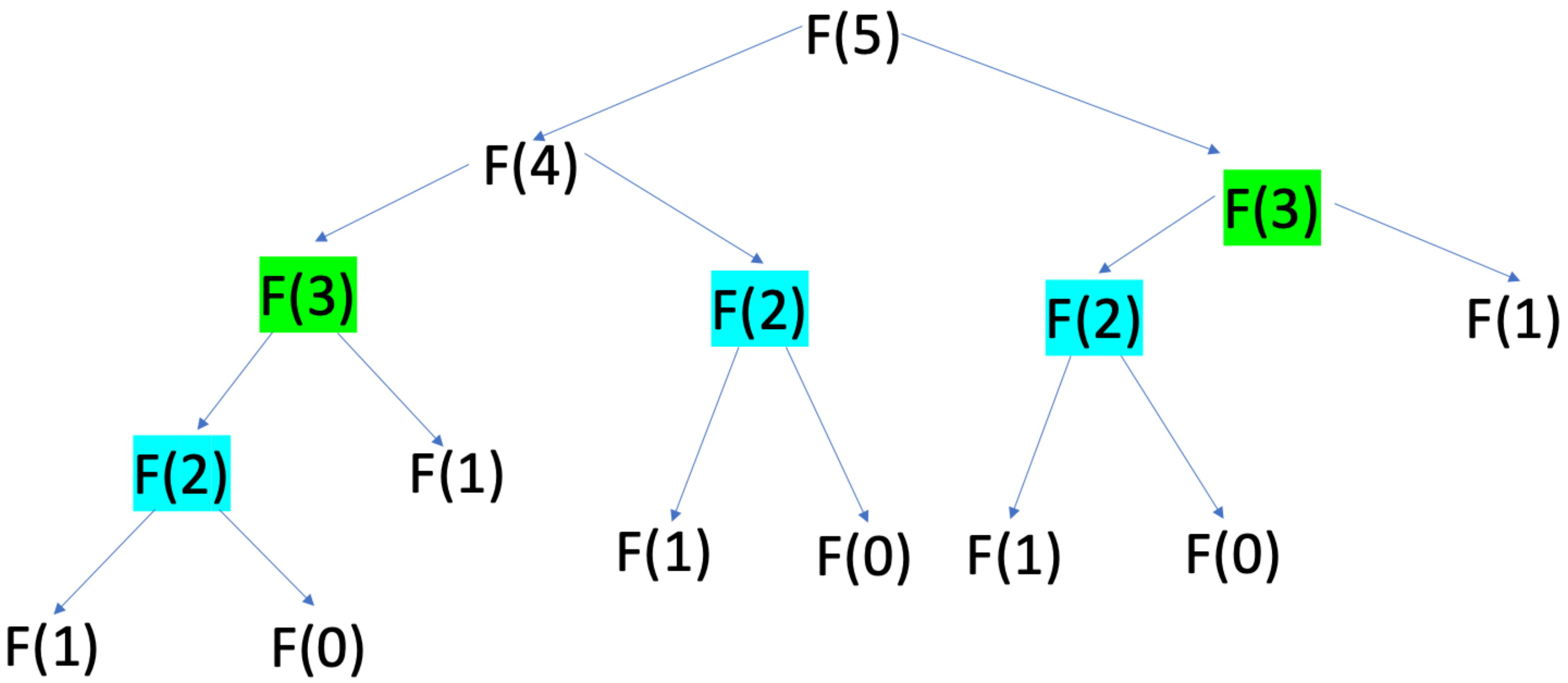
Fibonacci sequence

- The Fibonacci sequence is another mathematical construct that has a nice recursive expression

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

→ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 , 55, …

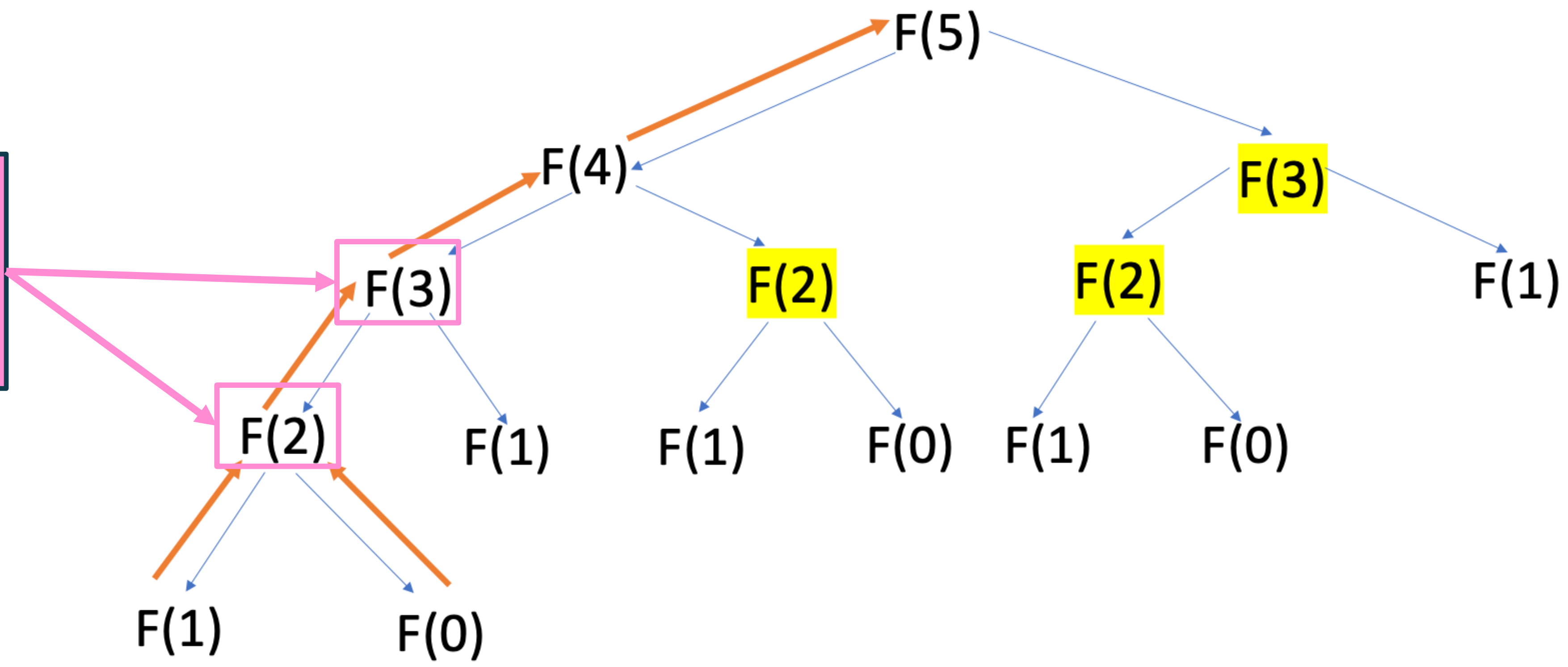
Fibonacci sequence



Computing F(5) recursively, implies computing F(2) three times and F(3) two times

Fibonacci sequence

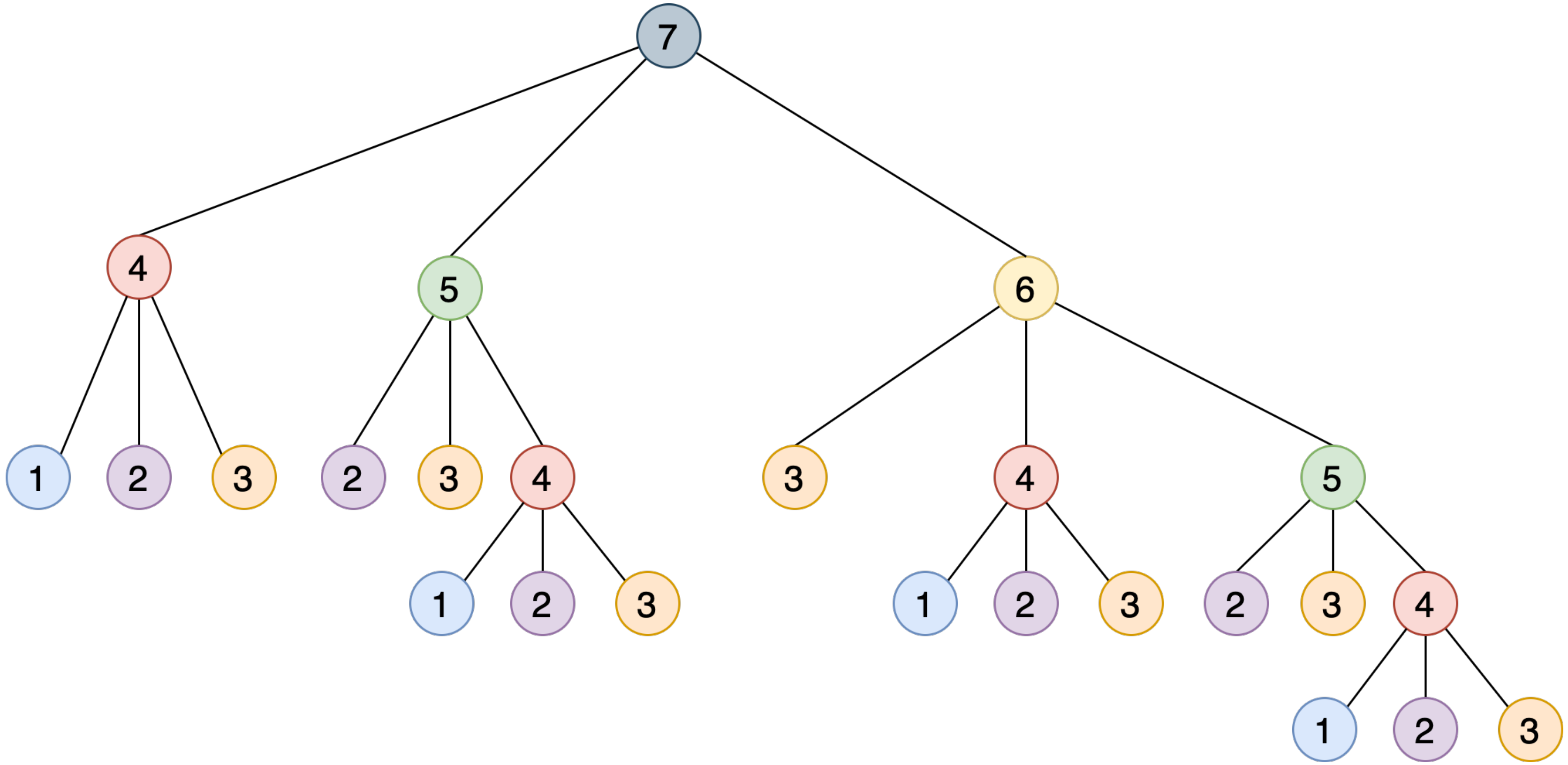
Cache these results the first time they are computed, so that later we can just read them



Let's implement this!

Memoization

Memoization: optimization technique used primarily to speed up computer programs by storing the results of expensive function calls to pure functions and returning the cached result when the same inputs occur again



Caching using @lru_cache

The `functools` package implements caching functionalities, that enable memoization

```
from functools import lru_cache

@lru_cache(maxsize=None)
def recursion(problem, …):
    # do operations
    return result
```

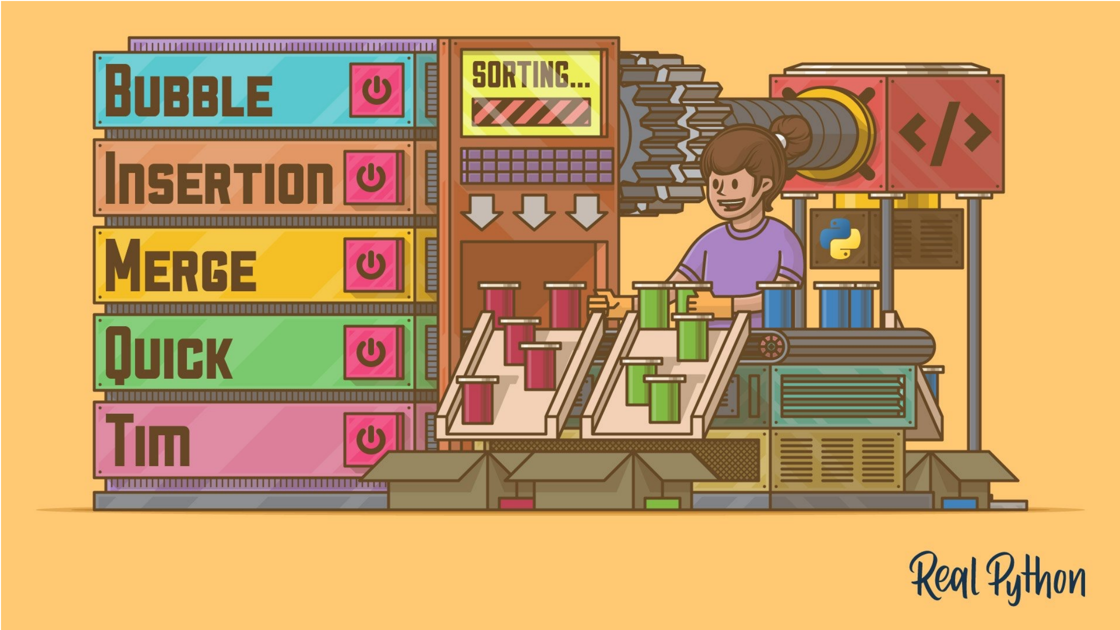
https://docs.python.org/3/library/functools.html

- `@lru_cache` is a decorator that wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls.
- Available since Python 3.2
- It uses a dictionary behind the scenes:
  - Key: the call to the function, including the supplied arguments
  - Value: the function's result
  - The function arguments have to be **hashable** for the decorator to work.

LRU cache

- The LRU cache should only be used when you want to reuse previously computed values.
- It doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call (such as generators and async functions)

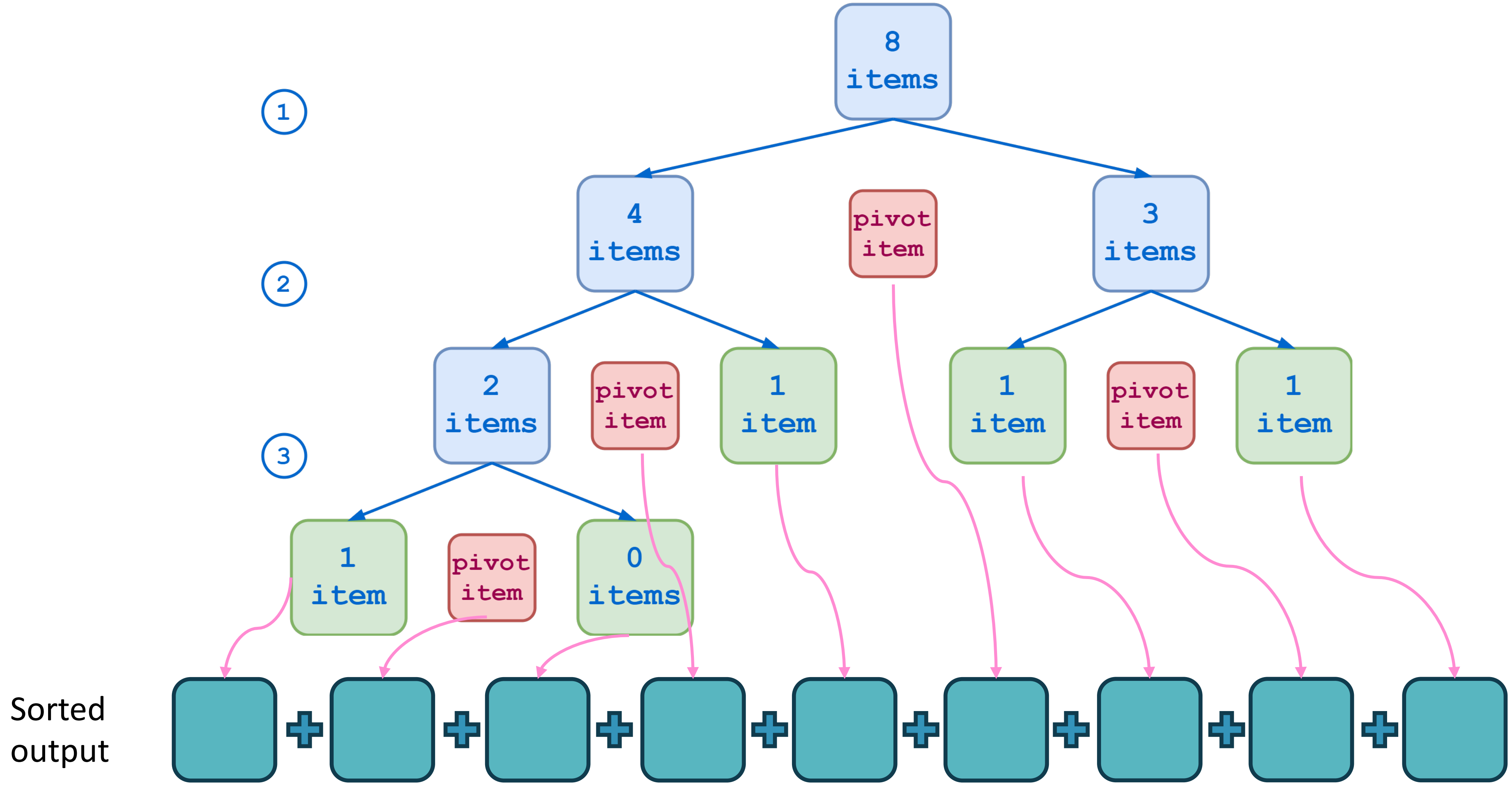
SORTING AND SEARCHING WITH RECURSION



Example: Quicksort

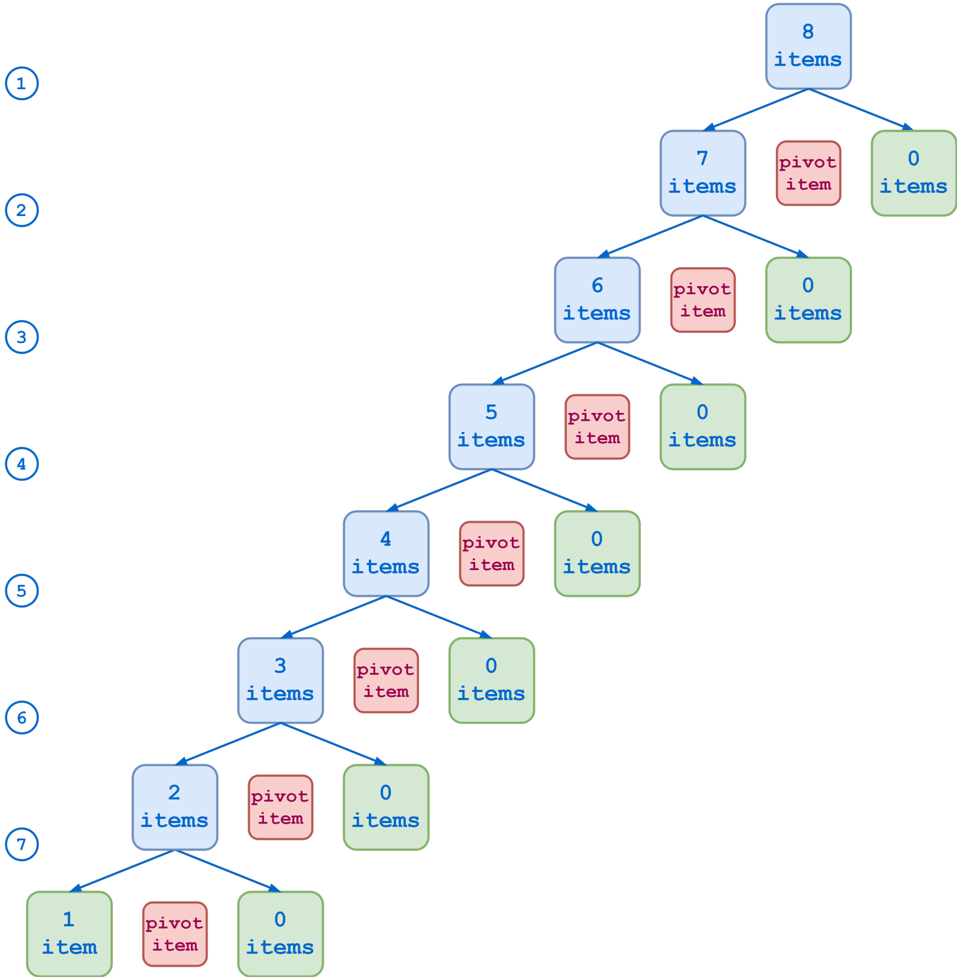
- Quicksort is a sorting algorithm based on the Divide et Impera principle:
    1. Choose the pivot item.
    2. Partition the list into two sublists:
        a. Those items that are less than the pivot item
        b. Those items that are greater than the pivot item
    3. Quicksort the sublists recursively

Example: Quicksort



Example: Quicksort

- The efficiency of the Quicksort algorithm depends on the choice of the pivot used to partition the list
- For an optimal partition we would need to know something about the data (e.g., looping through all the data, which may be very expensive)



Example: dichotomic search

- Problem
  - Determine whether an element x is **present** inside an ordered **vector** v[N]
- Approach
  - Divide the vector in two halves
  - Compare the middle element with x
  - Reapply the problem over one of the two halves (left or right, depending on the comparison result)
  - The other half may be ignored, since the vector is ordered

Example: dichotomic search

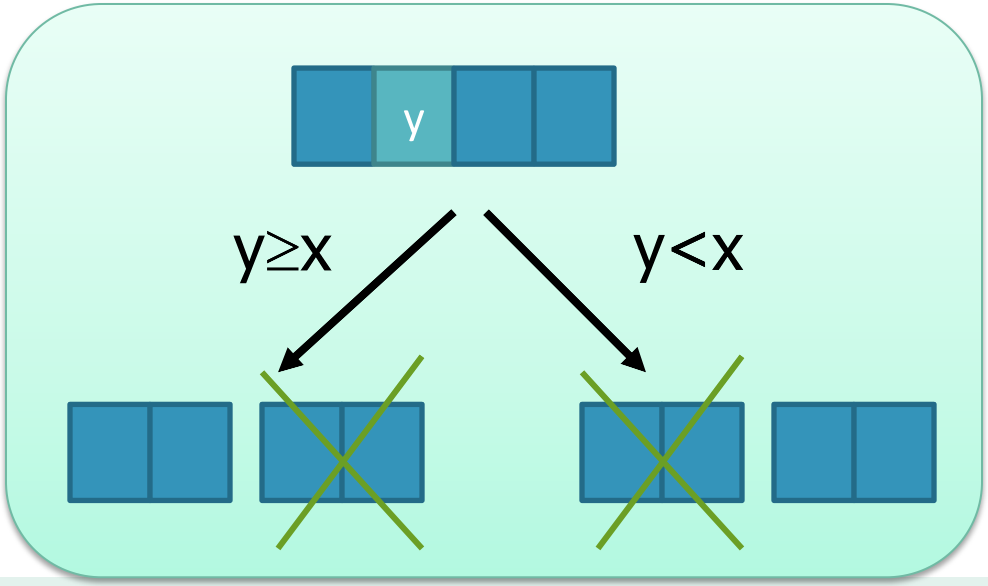
v

1	3	4	6	8	9	11	12

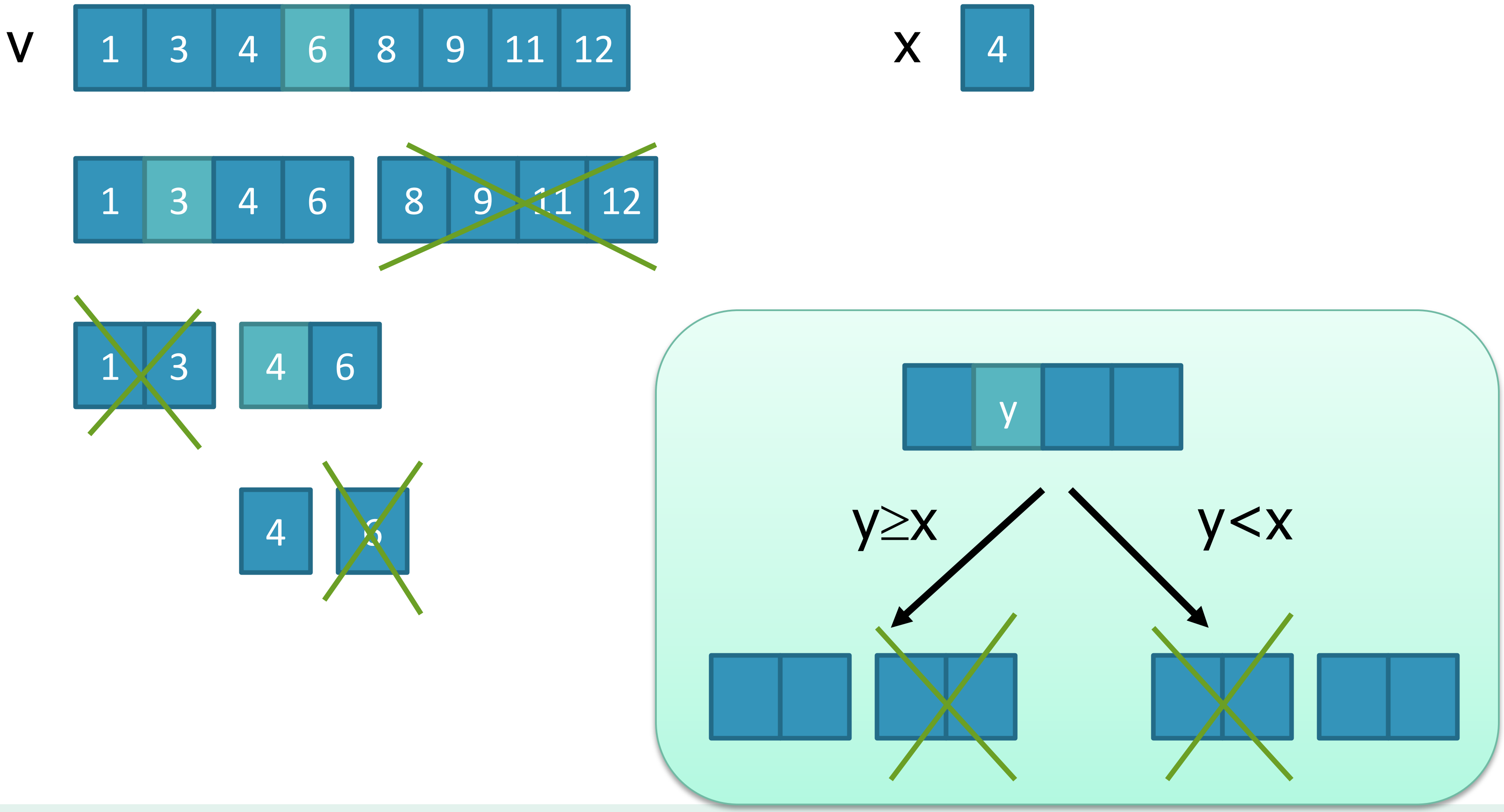
x

4

Example: dichotomic search



Example: dichotomic search



Example: dichotomic search

Alternative iterative solution

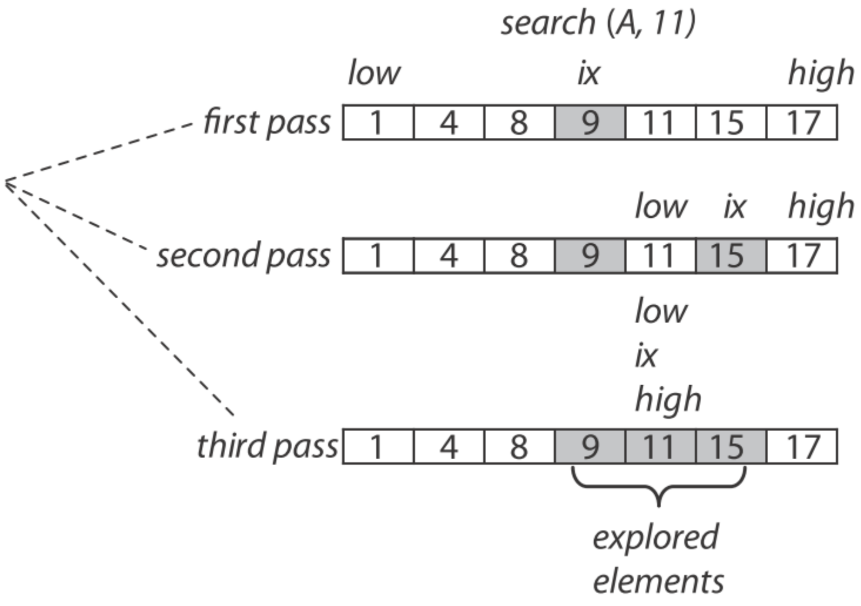
Binary Search

Best	Average	Worst
O (1)	O (log n)	O (log n)

Array

Divide and Conquer

```
search (A, t)
  low = 0
  high = n −1
  while (low ≤ high) do
    ix = (low + high)/2
    if (t = A[ix]) then
      return true
    else if (t < A[ix]) then
      high = ix − 1
    else low = ix + 1
  return false
end
```



DESIGN TIPS

Analyze the problem

- How do I structure a recursion in general?
- What does the *level* represent?
- What is a partial solution?
- What is a complete solution?

Generate the possible solutions

- What is the rule to generate all the solutions from *level+1*, starting from a partial solution of the current *level*?
- How can I recognize if a partial solution is also complete? (successful termination)
- How do I start the recursion? (*level 0*)?

Identify valid solutions

- Given a ***partial*** *solution*,
  - How can I know if it is valid (and thus I can continue)?
  - How can I know if it is not valid (and thus terminate the recursion)?
  - Maybe I cannot…
- Given a ***complete*** *solution*,
  - How can I know if it is valid?
  - How can I know if it is not valid?
- What should I do with the complete solutions that are valid?
  - Stop at the first one?
  - Compute them all?
  - Count them?

Choose the data structure

- What data structure should I use to store a solution (partial or complete)?
- What data structure should I use to keep track of the state of the research (of the recursion)?

Code Outline

```
def recursion(…, level):
    // E – instructions that should be always executed (rarely needed)
    do_always(…)

    // A
    if terminal_condition:
        do_something(…)
        return …

    for … //a loop, if needed
        //B
        compute_partial()

        if filtro: //C
            recursion(…, level+1)

        //D
        back_tracking
```

Code Outline

Blocco	Frammento di codice
A	
B	
C	
D	
E	

EXERCISES

X-Expansion

- We want to devise an algorithm that, given a binary string that includes characters `0`, `1` and `X`, will compute all the possible combinations implied by the given string.
- Example: given the string `01X0X`, algorithm must compute the following combinations
  - `01000`
  - `01001`
  - `01100`
  - `01101`

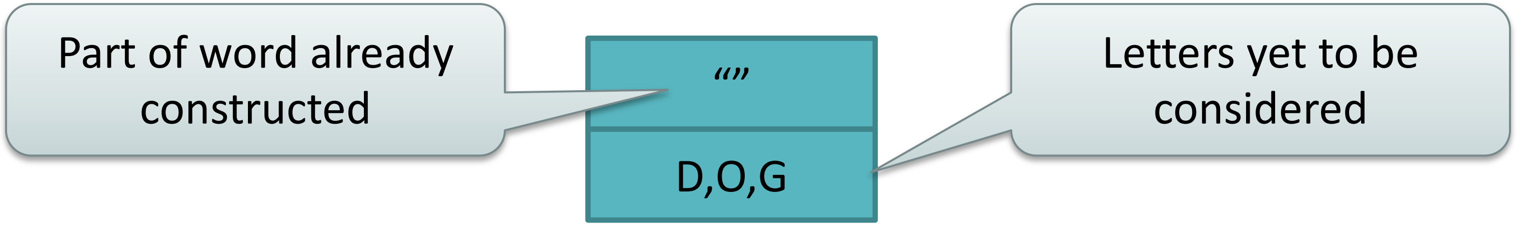
X-Expansion

- We may devise a recursive algorithm that explores the complete 'tree' of possible compatible combinations:
  - Transforming each X into a `0`, and then into a `1`
  - For each transformation, we recursively seek other X in the string
- The number of final combinations (leaves of the tree) is equal to 2^N, if N is the number of X.
- The tree height is N+1.

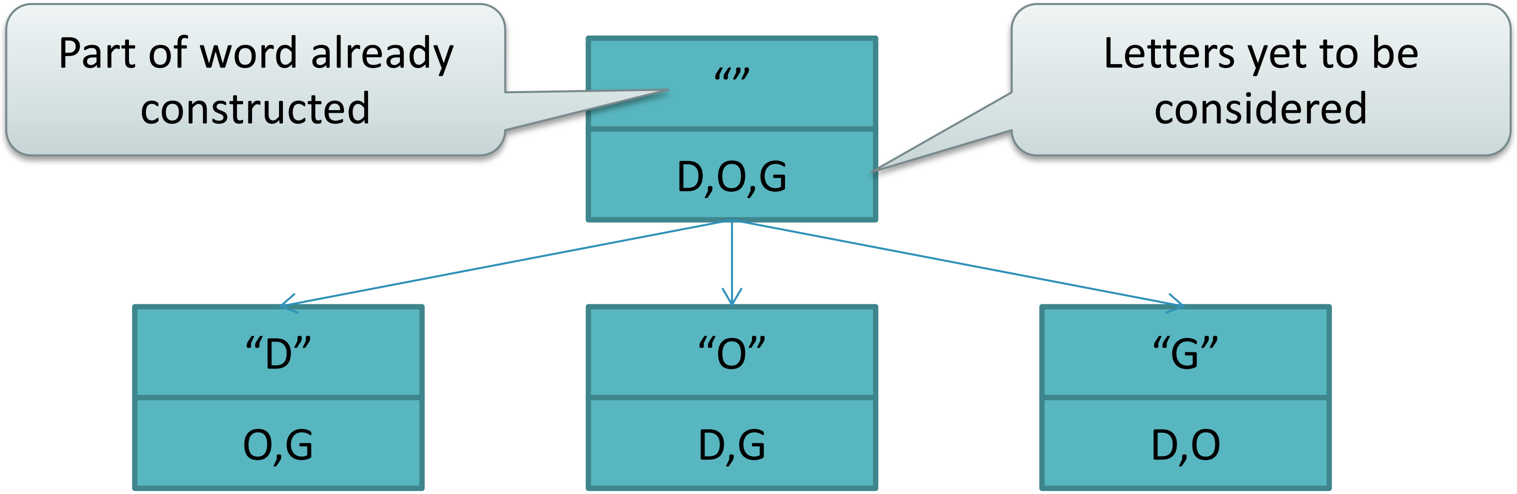
Anagrams

- Given a word, find all possible anagrams of that word
  - Find all permutations of the elements in a set
  - Permutations are N!
- E.g.: «Dog» → dog, dgo, god, gdo, odg, ogd

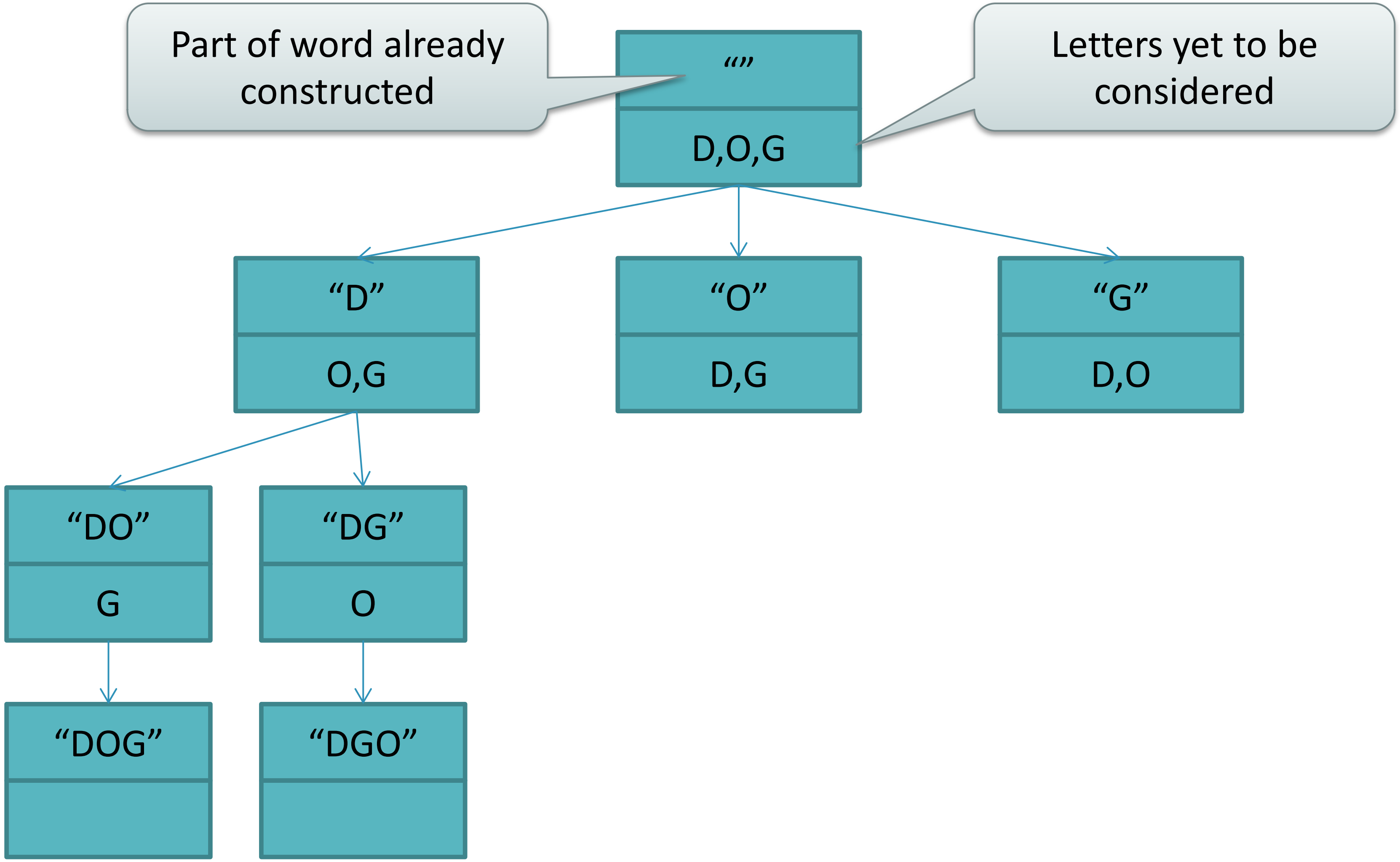
Anagrams: recursion tree



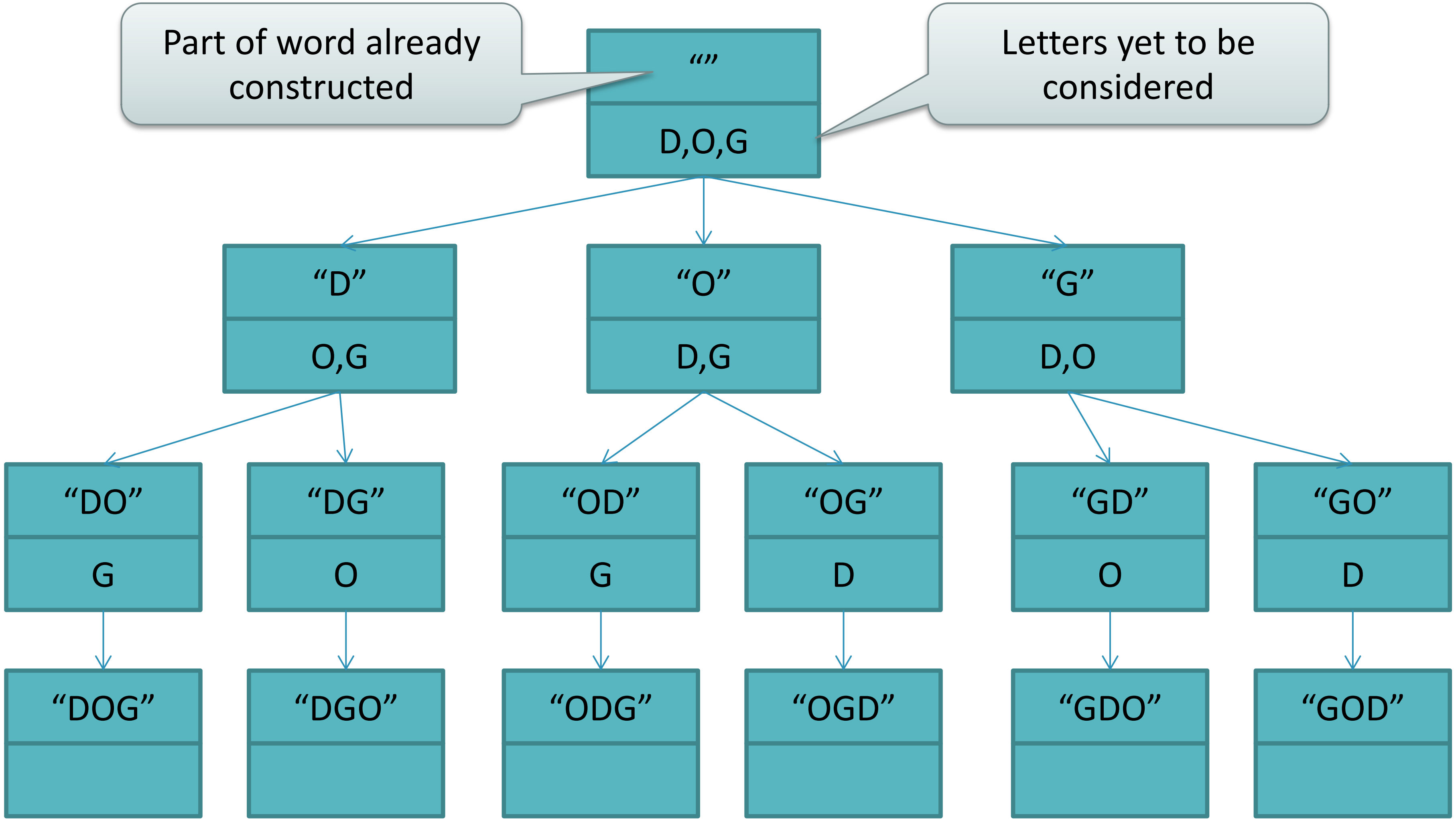
Anagrams: recursion tree



Anagrams: recursion tree



Anagrams: recursion tree

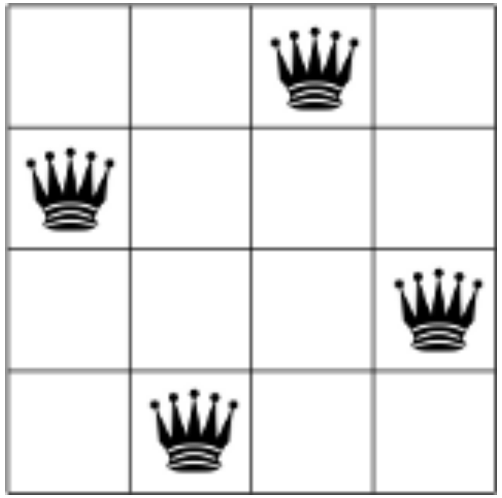


Anagrams variants

- Generate only anagrams that are "valid" words
  - At the end of recursion, check the dictionary
  - During recursion, check whether the current prefix exists in the dictionary
- Handle words with multiple letters: avoid duplicate anagrams
  - E.g., "seas" → seas and seas are the same word
  - Generate all and, at the end or recursion, check if repeated
  - Constrain, during recursion, duplicate letters to always appear in the same order (e.g, s always before s)
  - Use a set to avoid repetitions

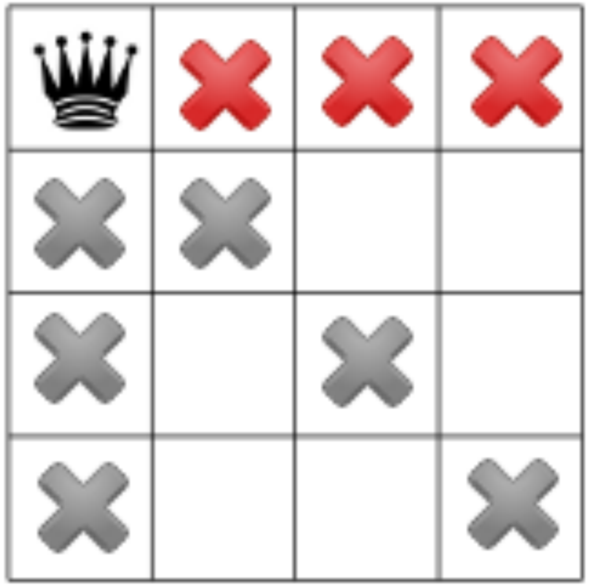
N-Queens

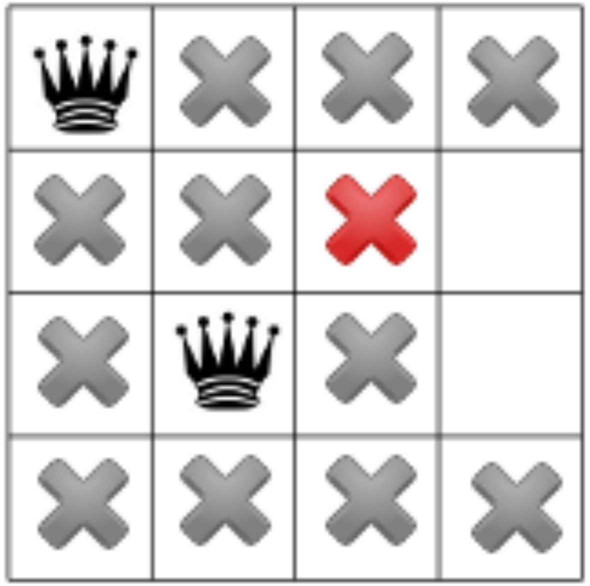
- In chess, a queen can attack horizontally, vertically, and diagonally. The N-queens problem asks:
- *How can N queens be placed on an NxN chessboard so that no two of them attack each other?*



N-Queens

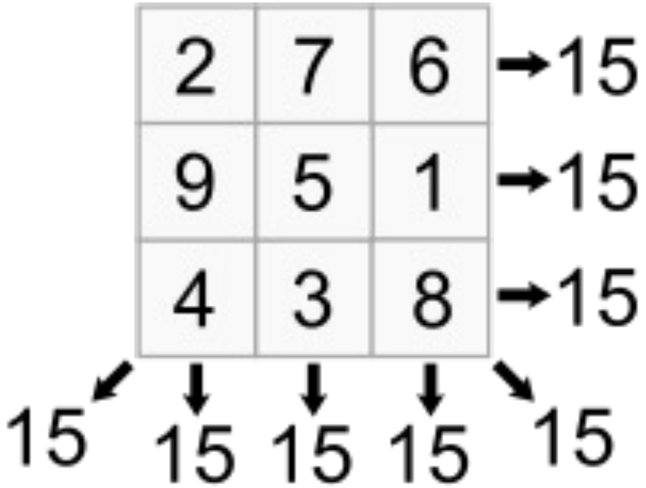
- We look for a recursive algorithm, that adds a queen at a time and check if we have found a solution





Magic Square

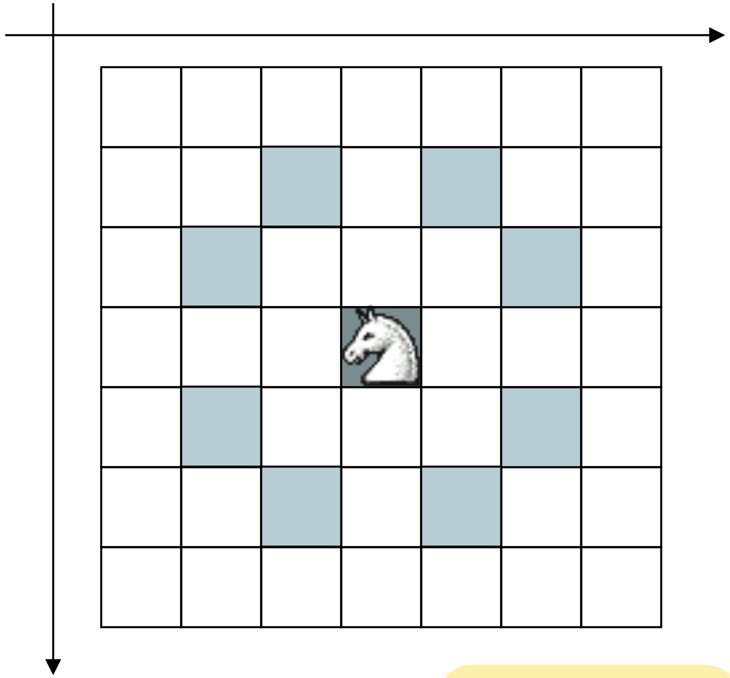
- A square array of numbers, usually **positive integers**, is called a **magic square** if the sums of the numbers in each row, each column, and both main diagonals are the same.
- The 'order' of the magic square is the number of integers along one side (*n*)
- The numbers in a magic square of order n are 1,2,..., n^2 and they are not repeated
- The constant sum is called the 'magic constant'.



EXERCISES FOR HOME

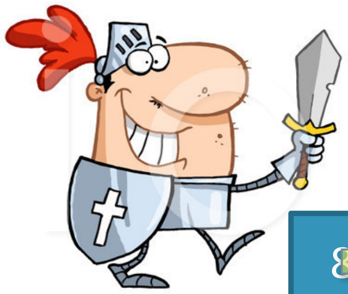
Knight’s tour

- Consider a NxN chessboard, with the Knight moving according to Chess rules
  - The Knight may move in 8 different cells
- We want to find a **sequence** of moves for the Knight where
  - **All** cells in the chessboard are visited
  - Each cell is touched exactly **once**
- The starting point is arbitrary



A simple game

You beat the monster, if the sum of the scores of your squares is exactly 50



8	2	5	5	6	7	3	9
1	2	4	1	9	2	3	1
2	2	5	2	4	7	9	7
8	2	5	6	6	6	3	9
1	2	4	1	9	2	3	1
2	7	1	1	4	7	8	9
2	3	5	3	1	8	9	9
8	2	3	1	6	7	3	9

License