Recursion

Programmazione Avanzata 2025-26

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, there is 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)

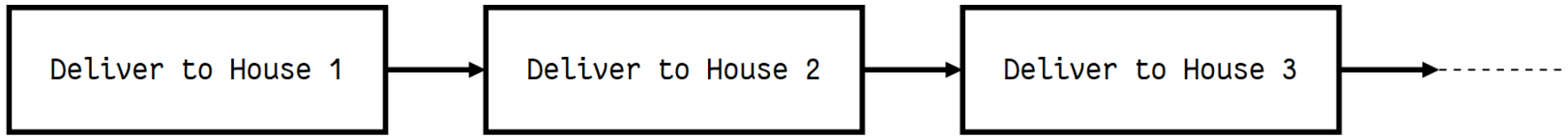
Example: Santa Claus deliveries

- It is Christmas time, and Santa Claus has a list of houses to visit to deliver presents

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

Example: Santa Claus deliveries

- He could loop through the houses, iteratively



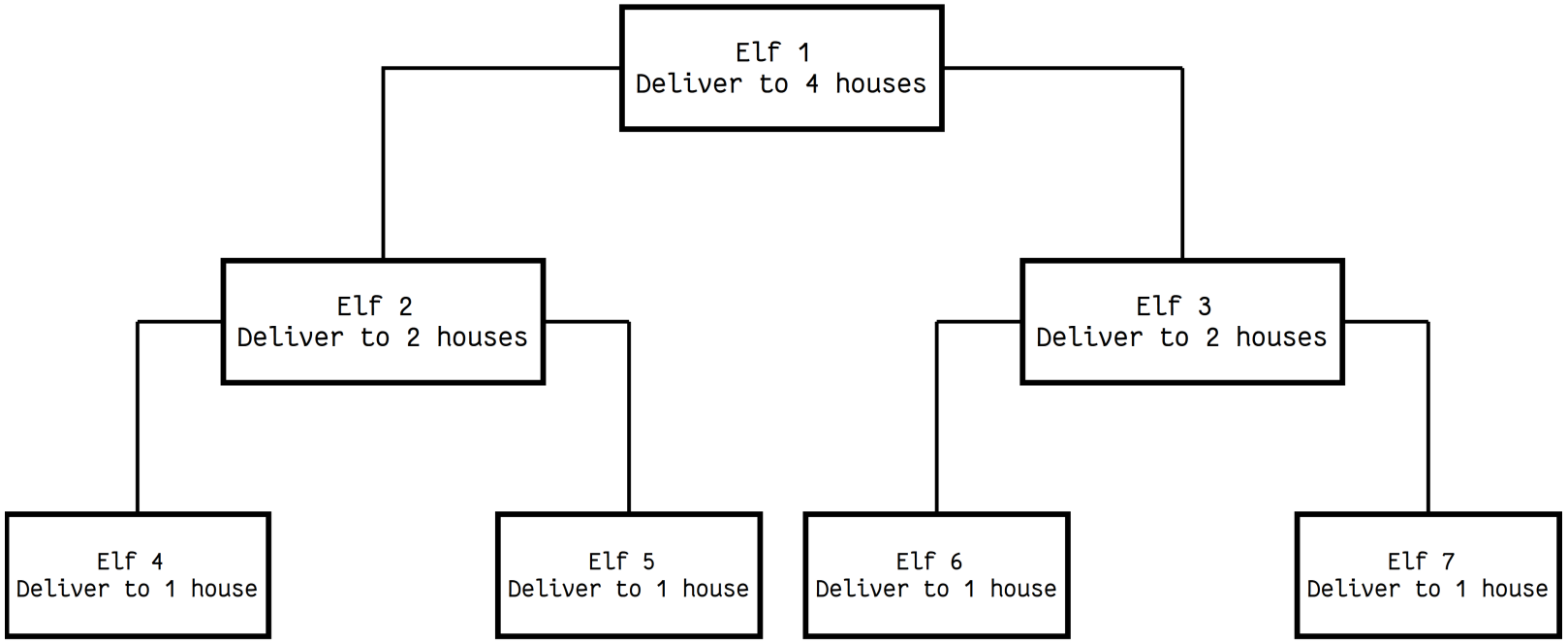
Iterative Present Delivery

Example: Santa Claus deliveries

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

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

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

How far can one go with recursions

- What happens executing the code below?

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

- This would go indefinitely, in theory; in practice, it would incur in a **`RecursionError`**
- It is possible to check how many iterations can be done using **`sys.getrecursionlimit()`**

Example: countdown

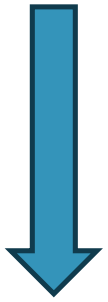
- Write a program implementing a countdown
- Can be coded
  - Iteratively, or
  - Recursively (with a method that calls itself)

Example: factorial

- Write a recursive program computing the factorial

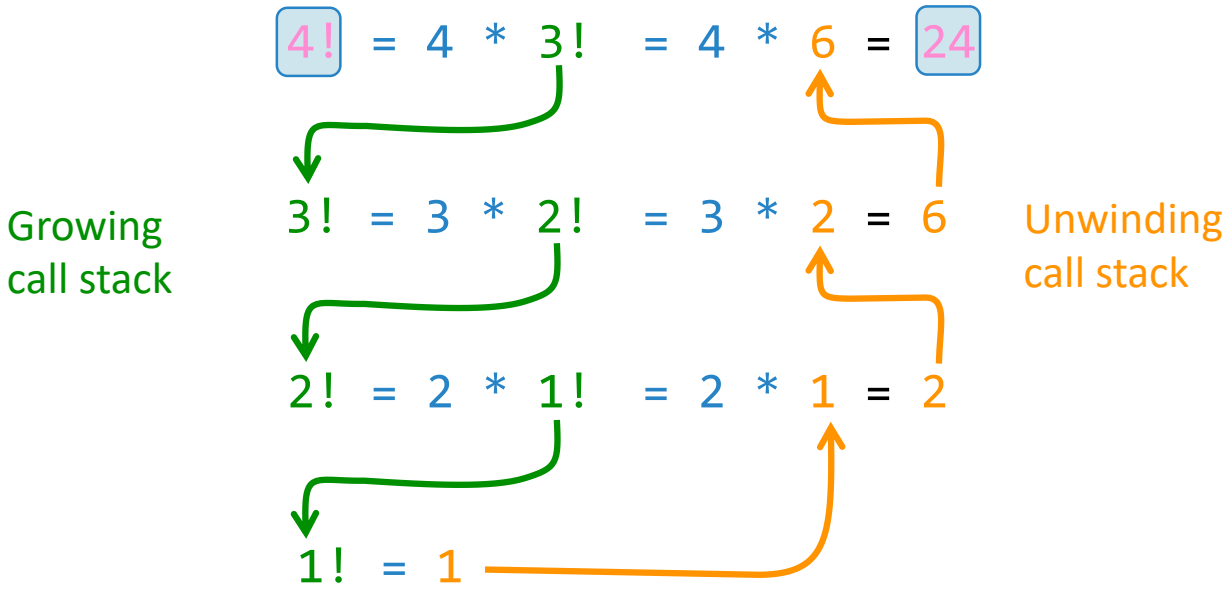
Factorial definition

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



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

- From the global context, that first invokes this method, the call stack will grow until reaching the banal case (1!)
- Then the call stack will unwind, by passing the results back until reaching the global context

Exmaple: binomial

- Write a recursive program computing the binomial coefficient exploiting the recurrence relations (derived from Tartaglia's triangle)

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

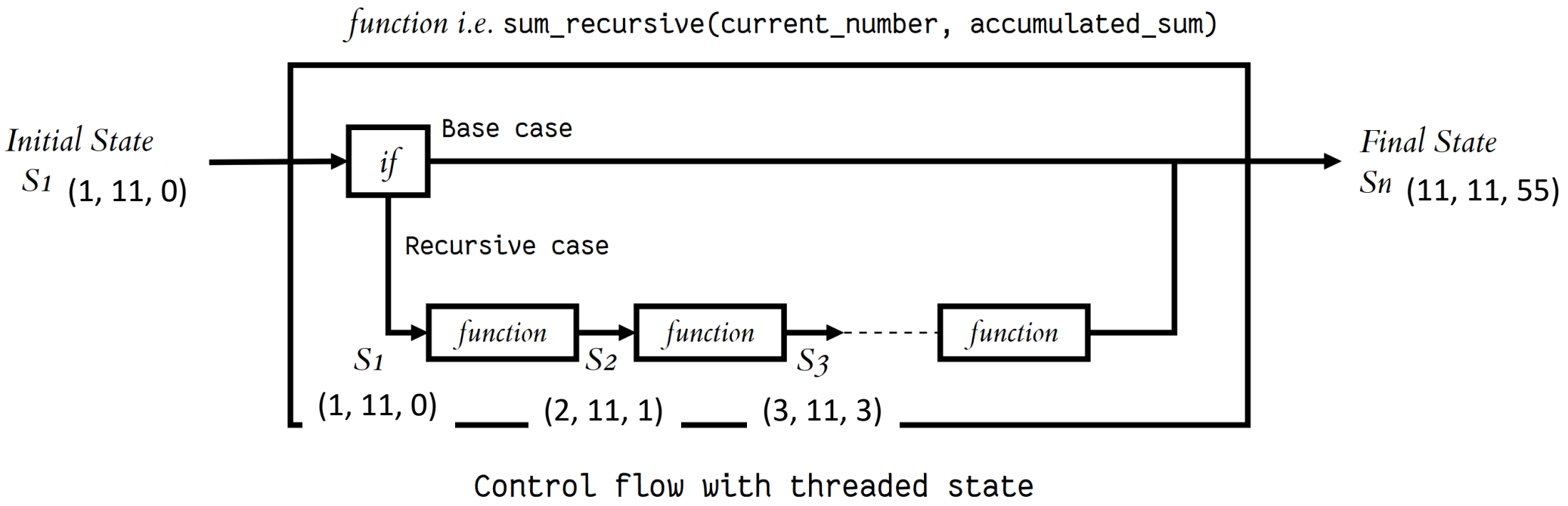
Maintaining the state

- 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 (discouraged)

Maintaining the state

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

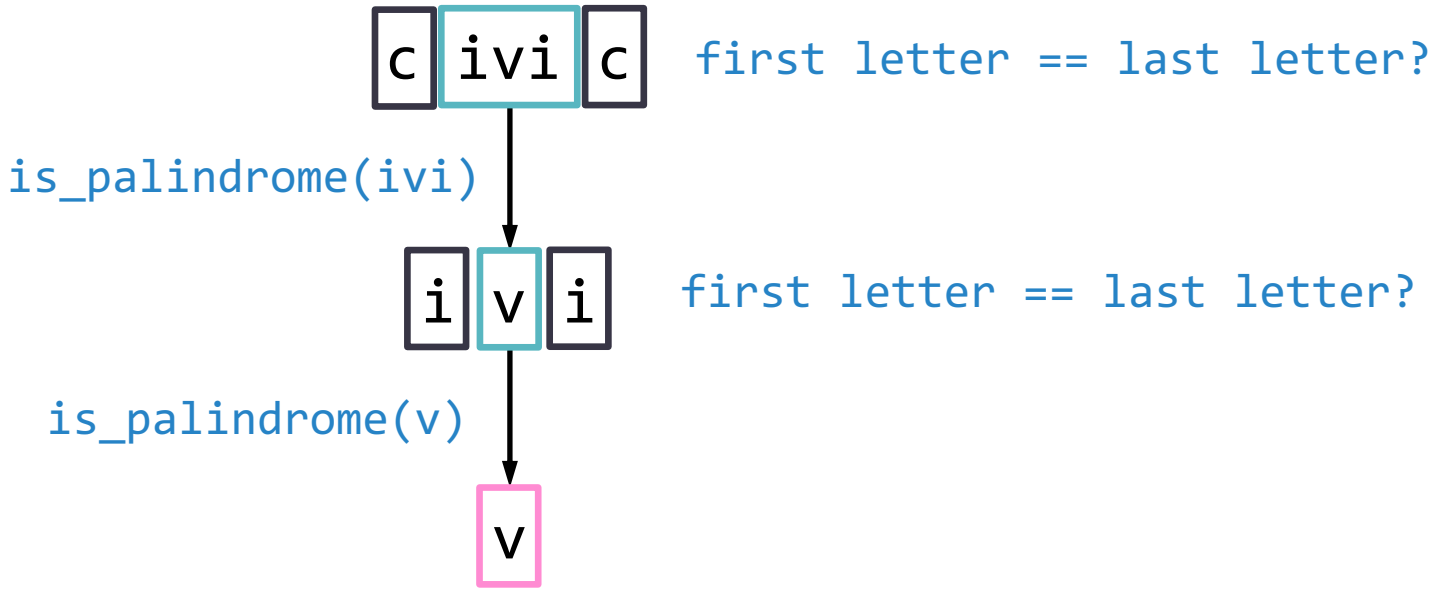
Maintaining the state



Control flow with threaded state

Example: palindrome checking

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



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:
  - Method defined to solve sub-problems like the initial one, but smaller
  - Method defined to combine the partial solutions into the overall solution of the original problem

Recursion: divide et impera

- Split a problem P into $\{Q_i\}$ where
  - Q_i are still complex, yet simpler instances of the same problem
  - Solve {Qi }, then merge the solutions
  - Merge & split must be “simple”
  - A.k.a., divide and conquer

Recursion: 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)
```

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 (e.g., 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
```

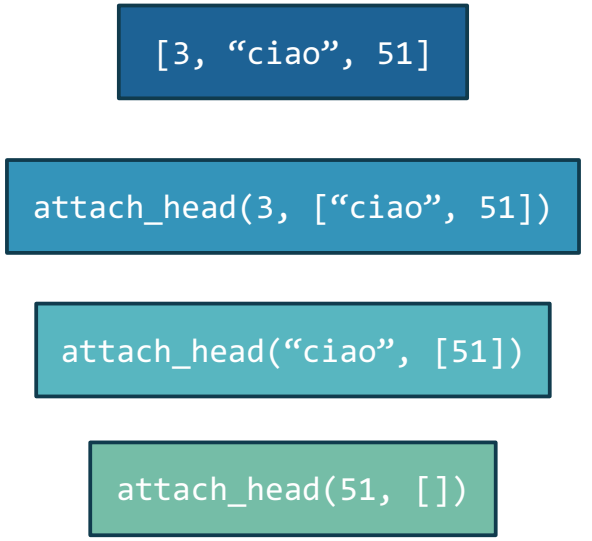
Exploration

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

Recursive data structures

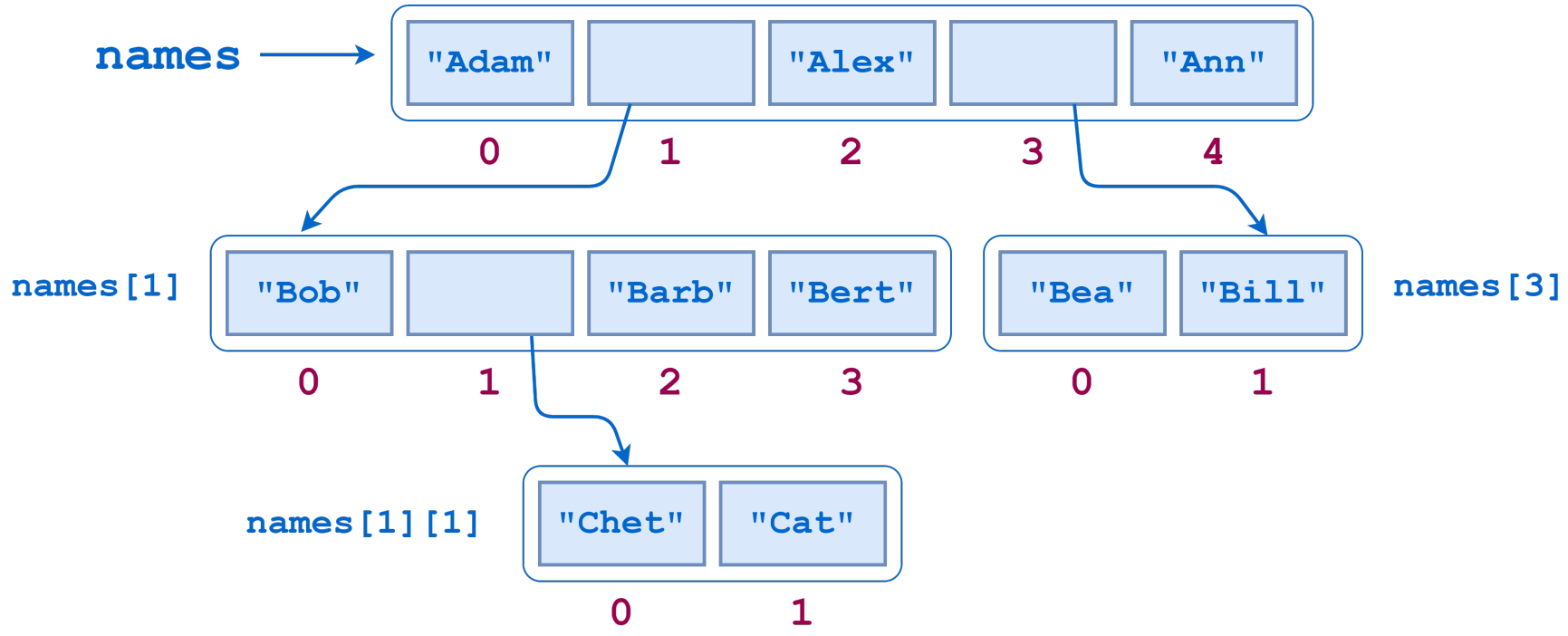
- A data structure is recursive if it can be defined in terms of a smaller version of itself
- Example: list

```
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



Example: nested list

- It can be implemented recursively

Example: nested list

```
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
```

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, save the state of the current level (count, list), and keep counting the elements of that level, until finished (while loop)

Example: nested list

```
def count_leaf_items(item_list):
    # Non-recursive implementation
    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
        elif isinstance(current_list[i], list):
            stack.append((current_list, i))
            current_list = current_list[i]
            i = 0
        else:
            count += 1
            i += 1
```

Recursion and efficiency

- How can we improve the runtime efficiency of the 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.)

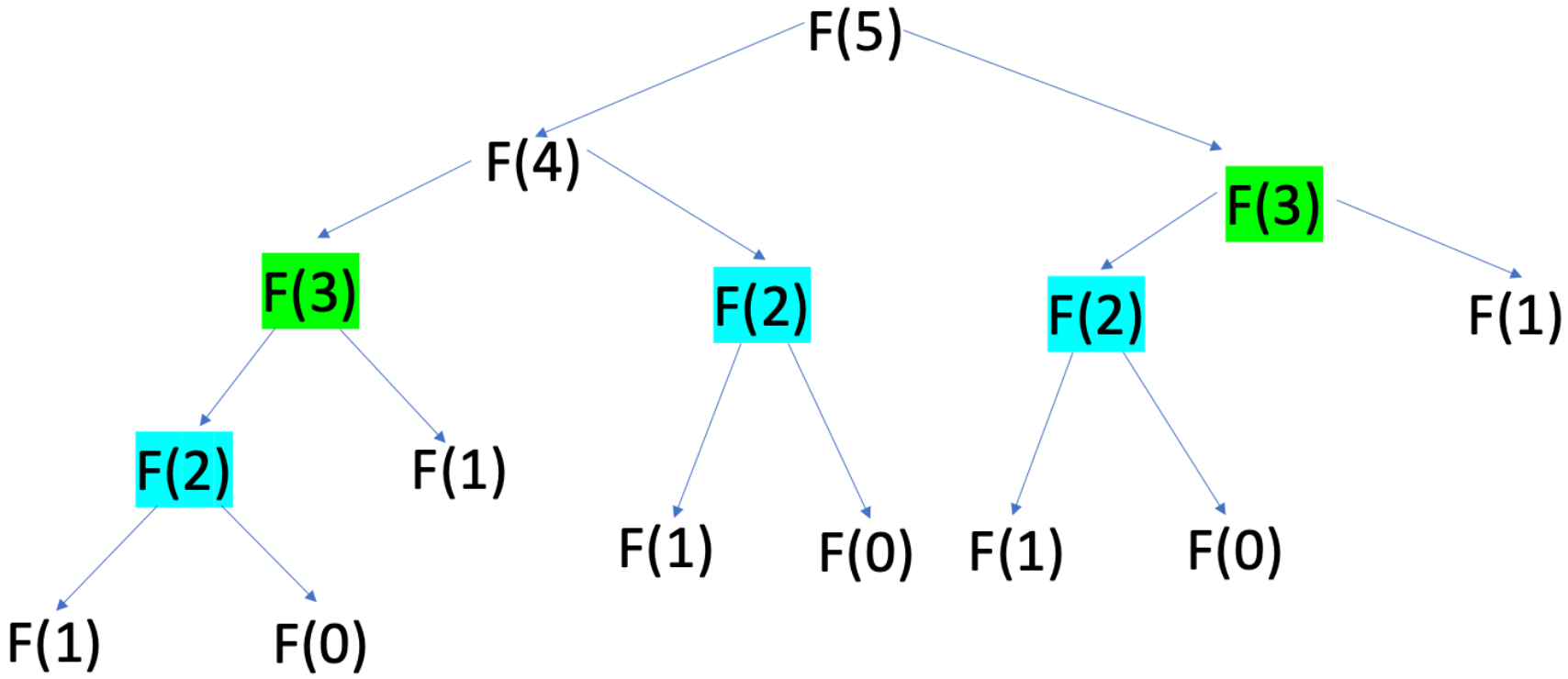
Fibonacci sequence

- The Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, …) is another mathematical construct that has a nice recursive expression

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

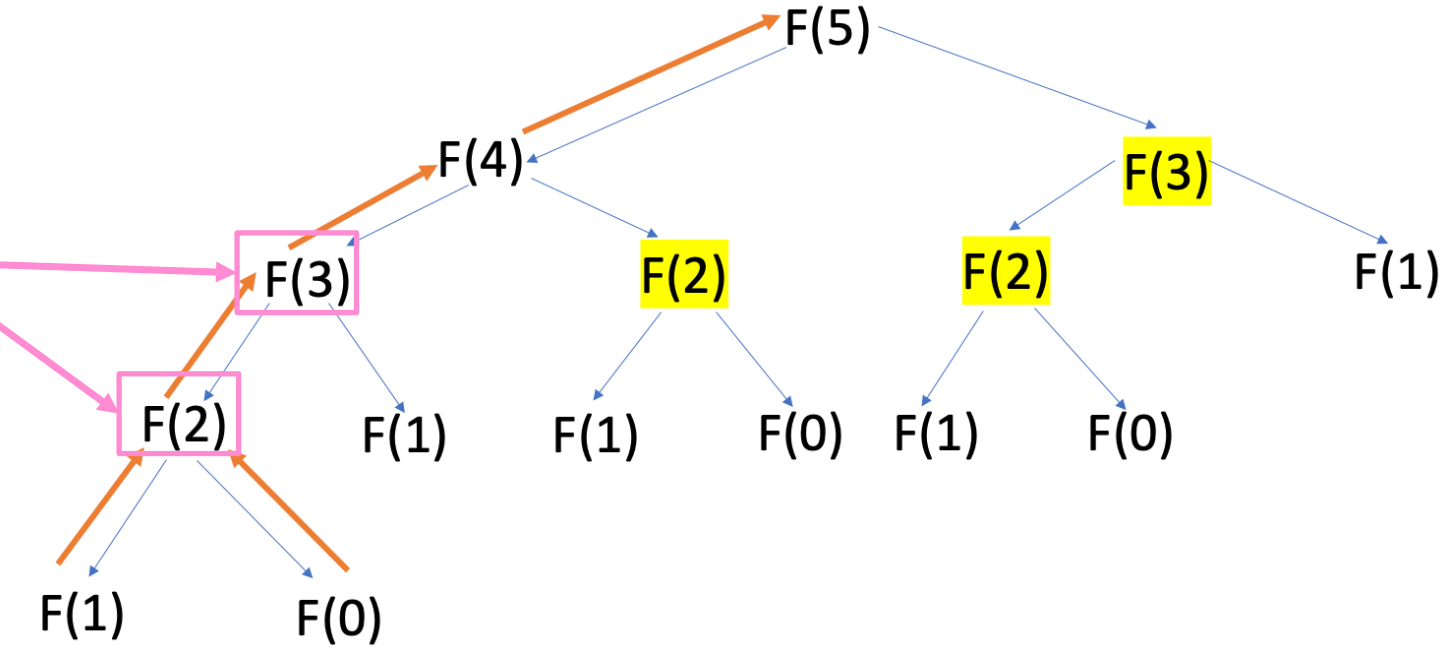
Fibonacci sequence

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



Fibonacci sequence

Cache the results the first time they are computed, so that to just read them later



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, available since Python 3.2, implements caching functionalities, that enable memoization
- **`@lru_cache`** is a decorator that wraps a function with a memoizing callable that saves up to a certain number of recent calls (defined using **`maxsize`** parameter)
- 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

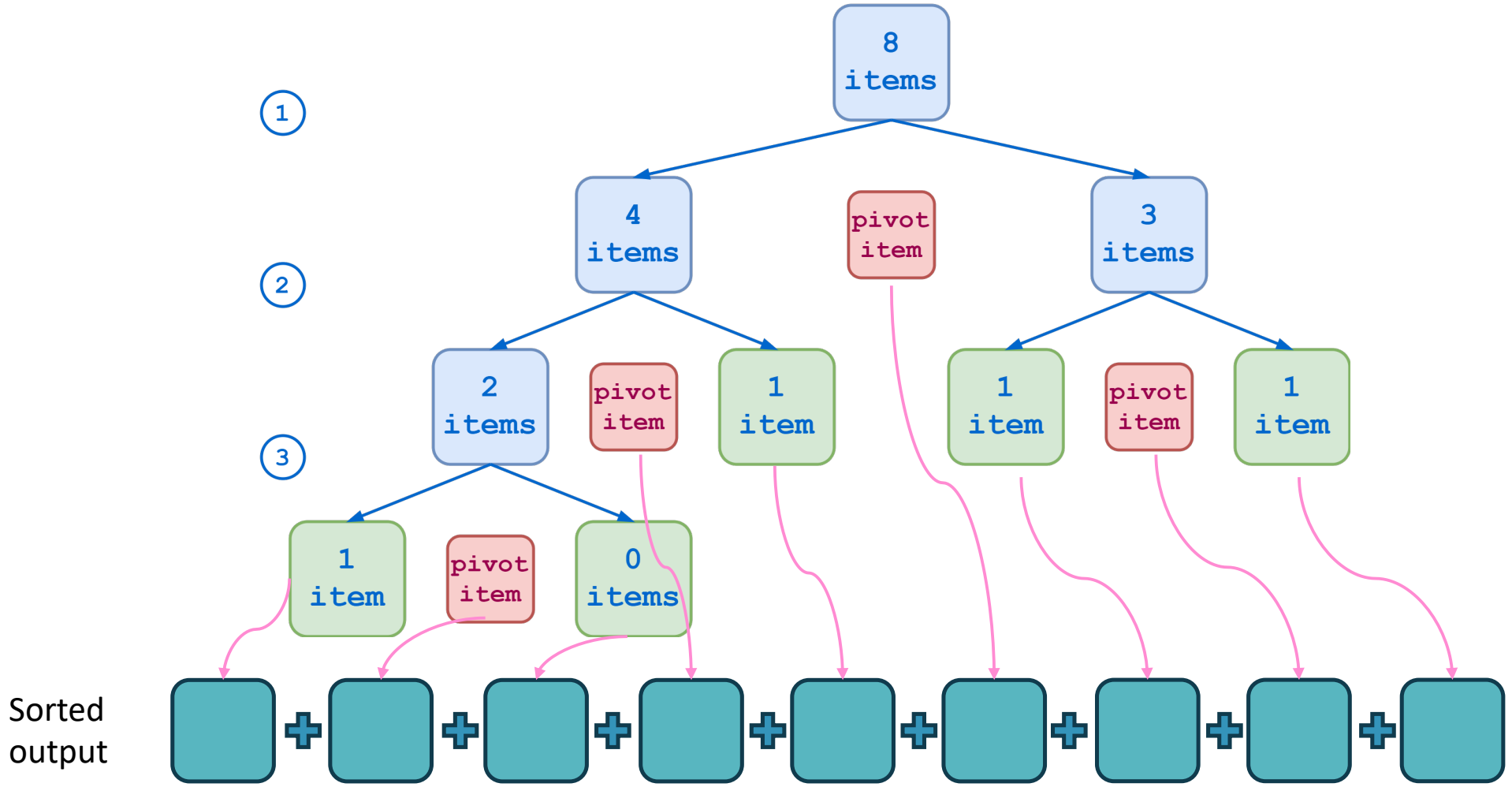
Sorting and searching with recursion

- Sorting and searching are perfect examples of algorithms that can be implemented with recursion

Example: quicksort

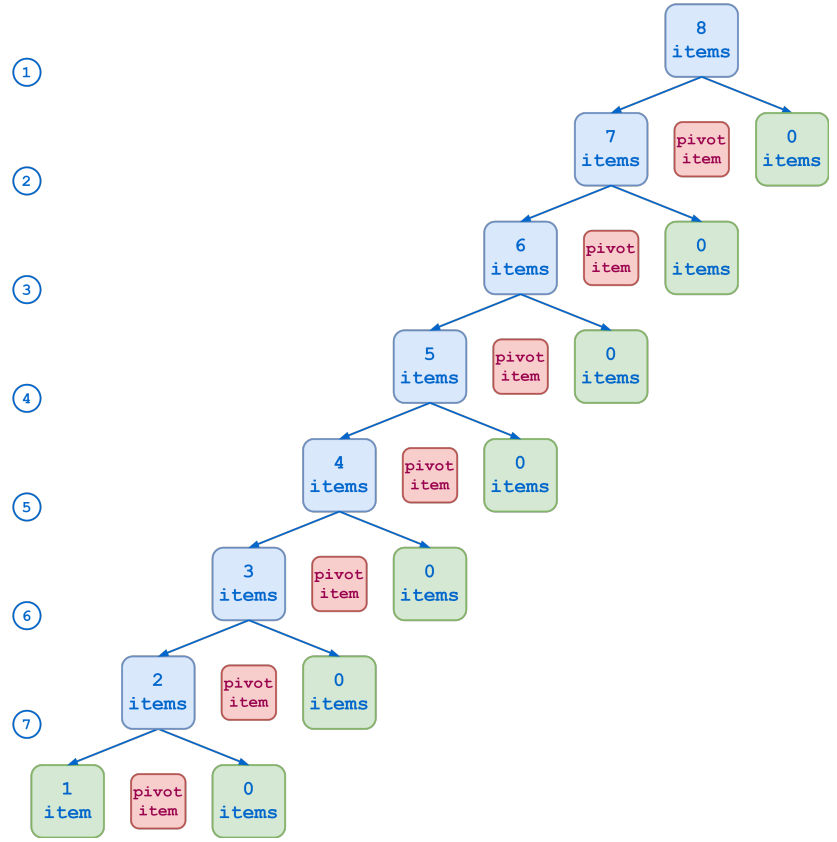
- Quicksort is a sorting algorithm based on the divide et impera principle:
  - Choose the pivot item.
  - Partition the list into two sublists:
    - Items that are less than the pivot item
    - Items that are greater than the pivot item
  - Quicksort the sublists recursively

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 something about the data would have to be known (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

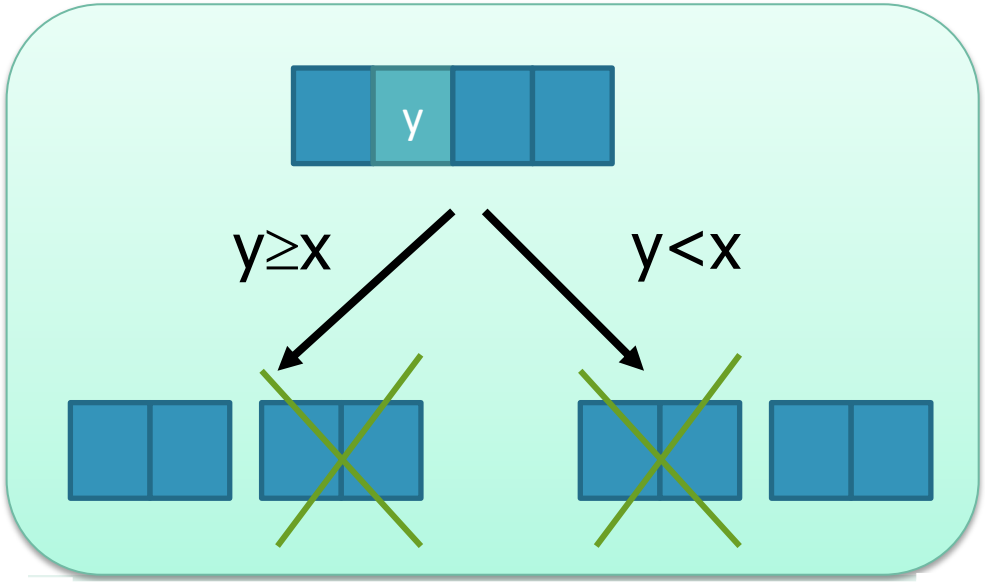
Example: dichotomic search

v

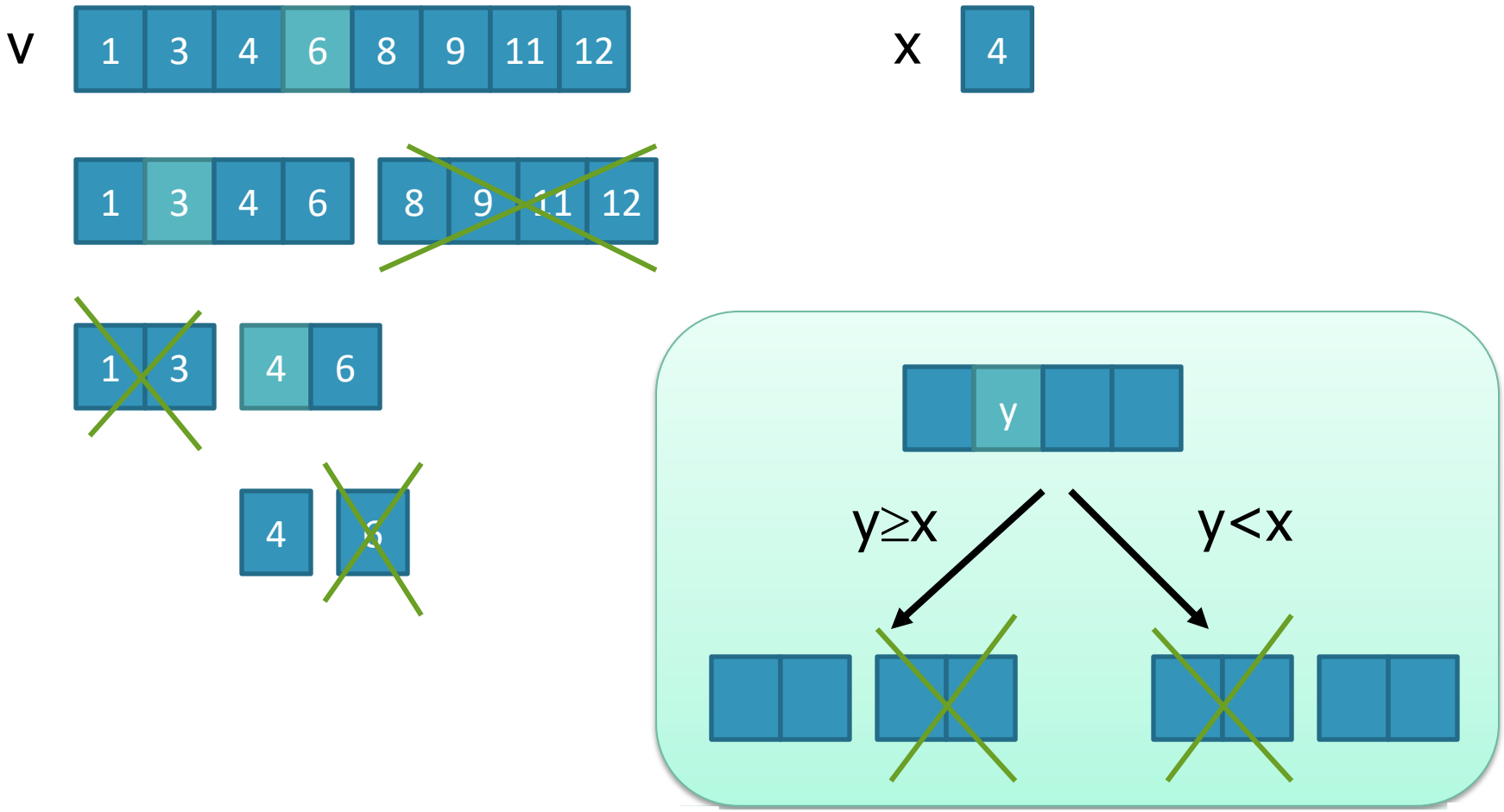
1	3	4	6	8	9	11	12

x

4



Example: dichotomic search



Example: dichotomic search

- Alternative iterative solution

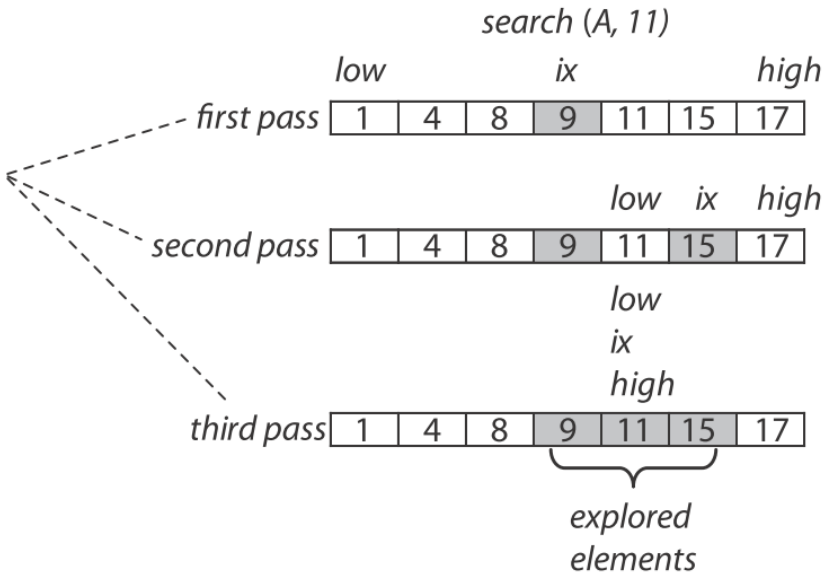
BINARY SEARCH

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

Array

Divide and Conquer

```
search (A, t)
1.    low = 0
2.    high = n –1
3.    while (low ≤ high) do
4.       ix = (low + high)/2
5.       if (t = A[ix]) then
6.             return true
7.       else if (t < A[ix]) then
8.             high = ix – 1
9.       else low = ix + 1
10.   return false
end
```



Design tips

- Analyze the problem
- Generate the possible solutions
- Identify valid solutions
- Choose the data structure

Analyze the problem

- How to 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 to recognize if a partial solution is also complete? (successful termination)
- How to start the recursion (level 0)?

Identify valid solutions

- Given a partial solution
  - How to know if it is valid (and thus it is possible to continue)?
  - How to know if it is not valid (and thus terminate the recursion)?
  - Maybe not possible
- Given a complete solution
  - How to know if it is valid?
  - How to know if it is not valid?
- What to 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 be used to store a solution (partial or complete)?
- What data structure should be used 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 filter: # C
                    recursion(…, level+1)
          # D
          back_tracking
```

Exercise: X-expansion

- Write a program that, given a binary string that includes characters 0, 1 and X, 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

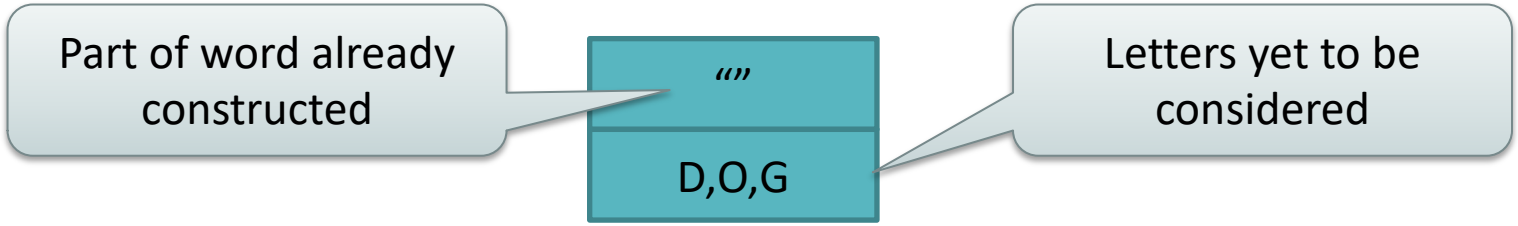
Exercise: X-expansion

- A recursive algorithm may be devised that explores the complete ‘tree’ of possible compatible combinations:
  - Transforming each X into a 0, and then into a 1
  - For each transformation, recursively seek other X in the string
- The number of final combinations (leaves of the tree) is equal to 2N, if N is the number of X
  - The tree height is N+1

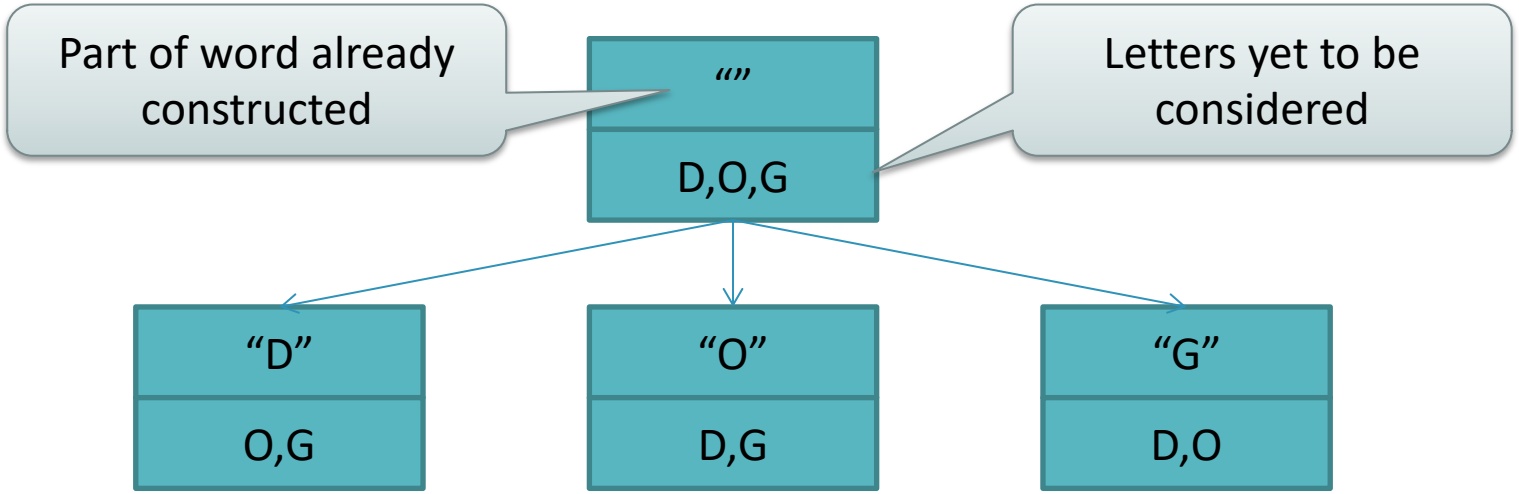
Exercise: anagrams

- Given a word, write a program that finds all possible anagrams of that word
  - Find all permutations of the elements in a set
  - Permutations are N!
  - E.g.: from “dog”, one gets dog, dgo, god, gdo, odg, ogd

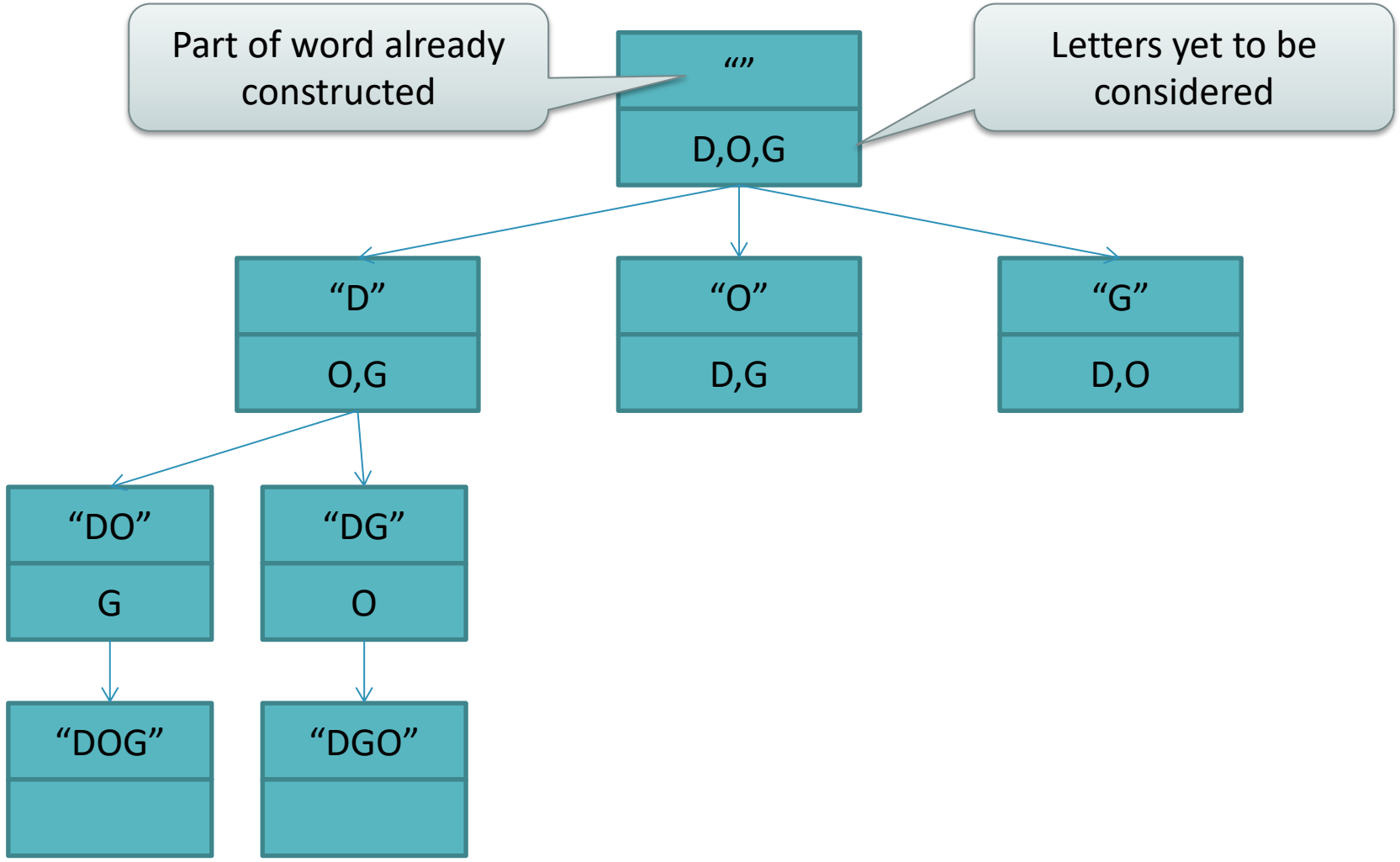
Exercise: anagrams



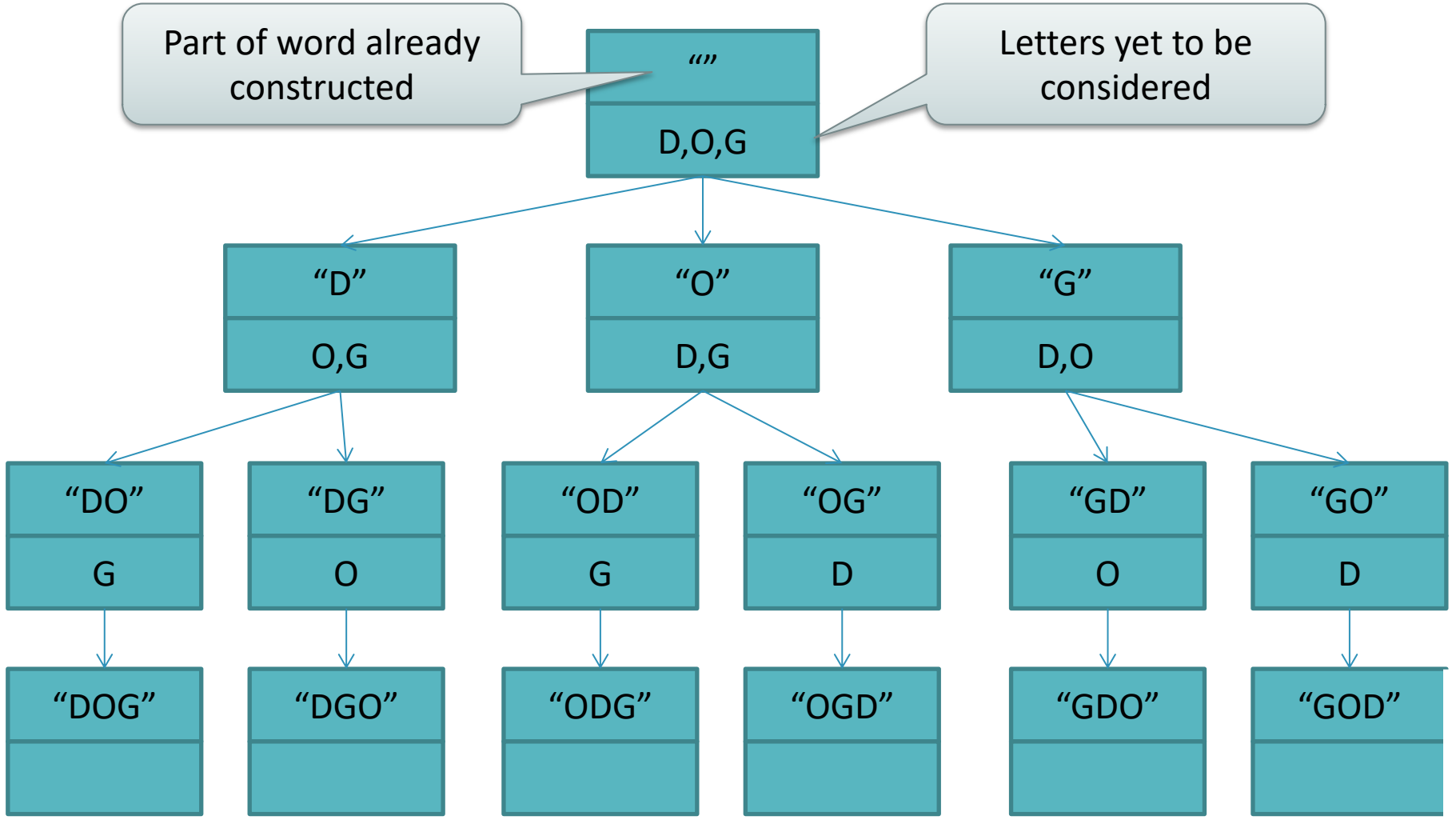
Exercise: anagrams



Exercise: anagrams



Exercise: anagrams

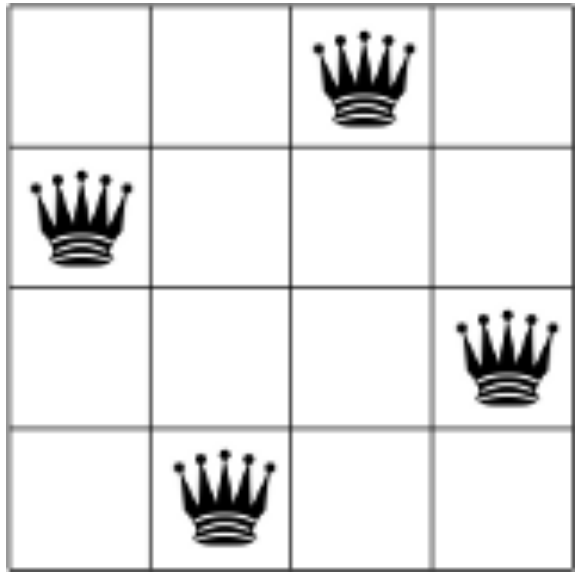


Exercise: anagram variants

- 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., from “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

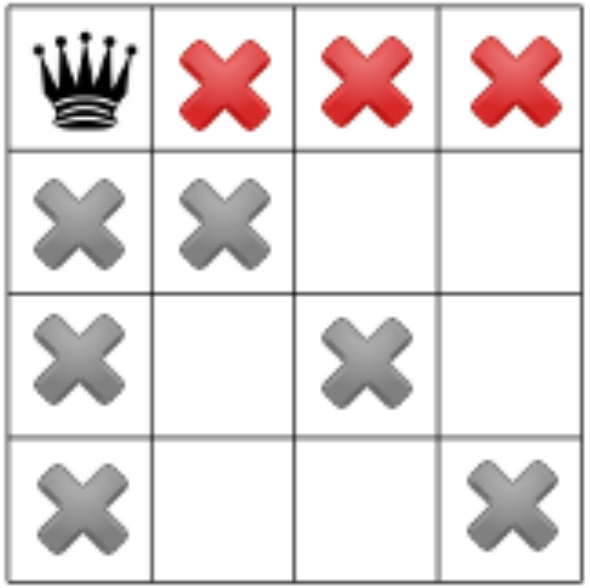
Exercise: N-Queens

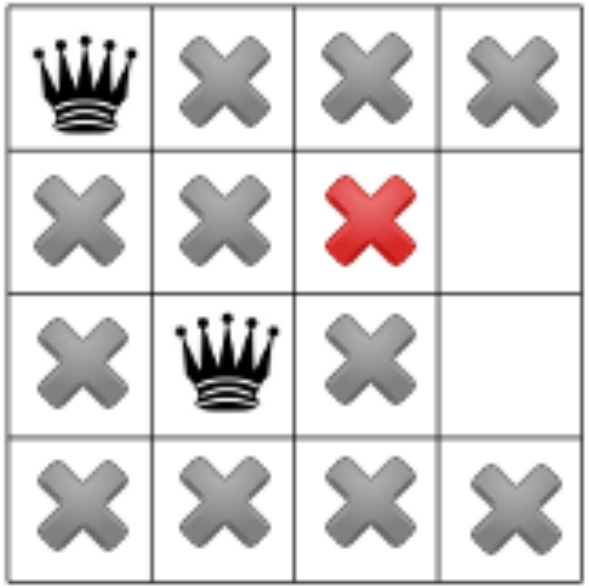
- In chess, a queen can attack horizontally, vertically, and diagonally
- The N-Queens problem asks:
  - How can N queens be placed on an NxN chessboard so that no two of them attack each other?



Exercise: N-Queens

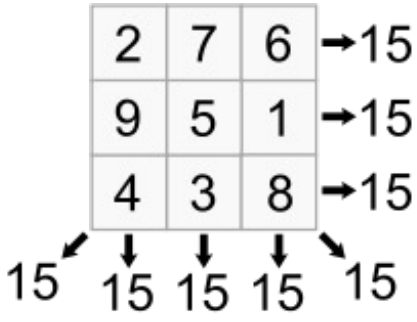
- Write a recursive program that adds a queen at a time and check whether a solution has been found





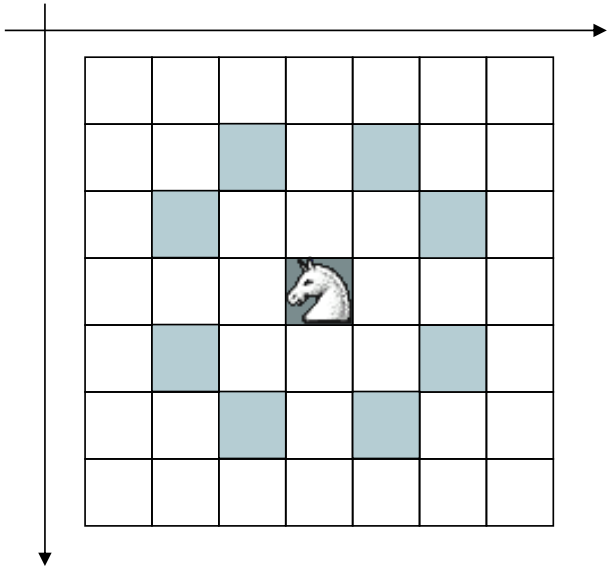
Magic square

- 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”



Exercise: Knight's tour

- Consider a NxN chessboard, with the Knight moving according to Chess rules
- The Knight may move in 8 different cells
- Write a program that finds 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



Exercise: a simple game

- The player beats the monster, if the sum of the scores of his or her squares is exactly 50