



Recursion

Solving problems by dividing them in smaller, similar problems

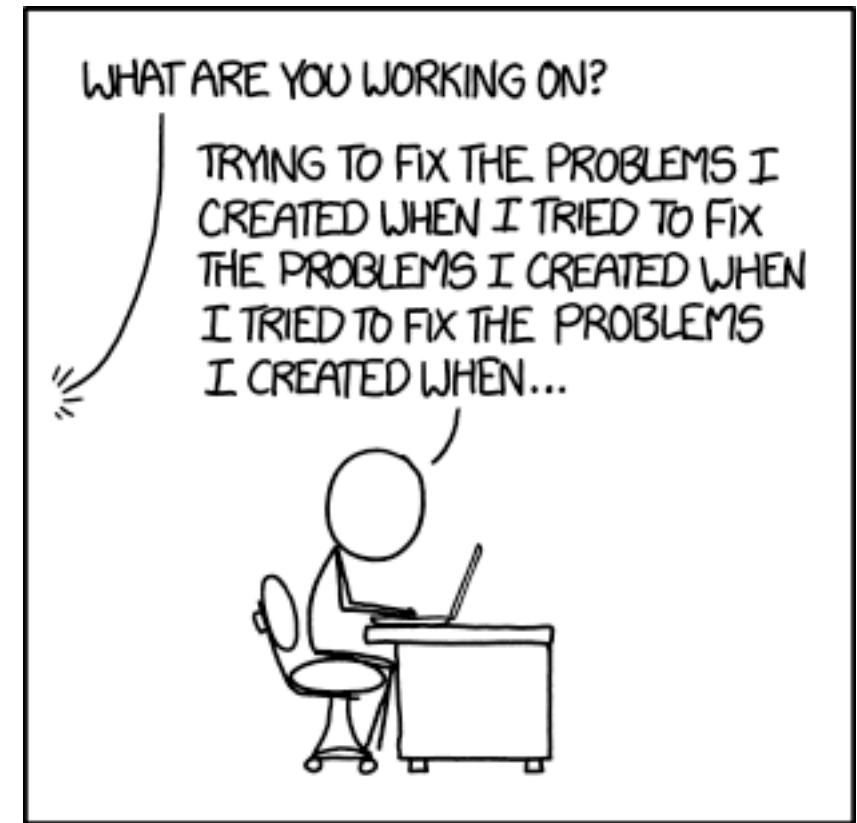
Fulvio Corno
Giuseppe Averta
Carlo Masone
Francesca Pistilli



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

Una definizione **ricorsiva** è una definizione in cui il termine definito compare nella definizione stessa!



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

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

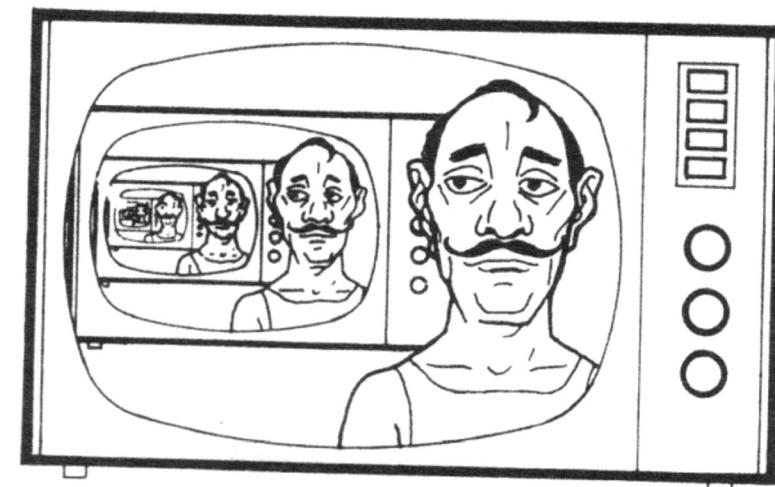
TO-DO LIST

1. Make a to-do list

Definition

Metodo ricorsivo: metodo / procedura che richiama se stesso nella sua definizione!

- 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)

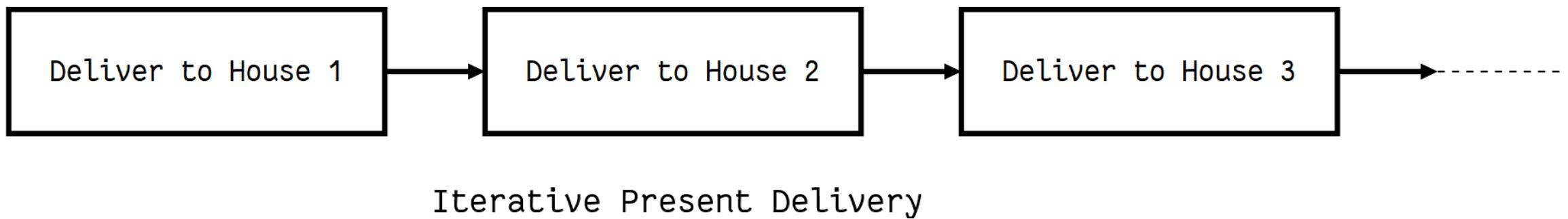


Definition



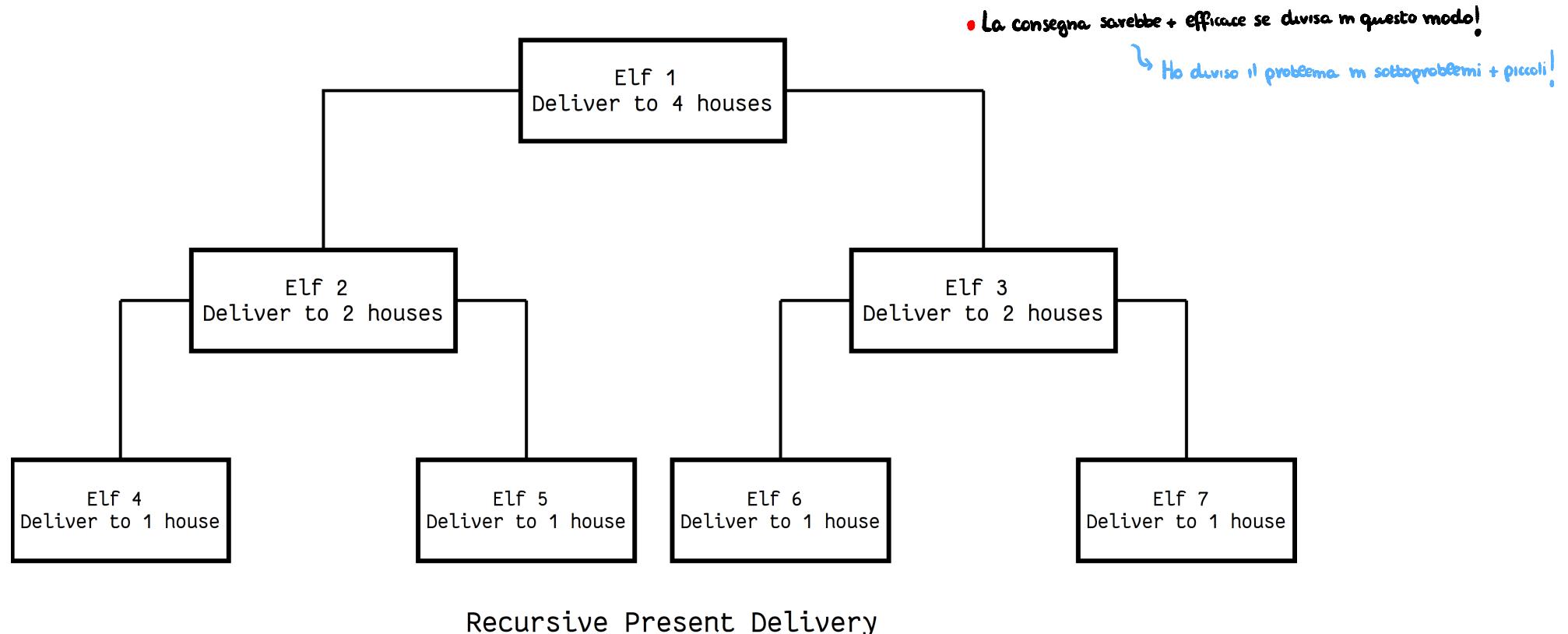
Example: Santa Claus deliveries

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



Example: Santa Claus deliveries

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



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)
```

Consegna **iterativa**: 1° metodo

```
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)
```

Consegna **ricorsiva**
2° metodo

How far can we go with recursions

What happens executing this?

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

→ Avrei un **RecursionError** perché la funzione continuerebbe ricorsivamente a chiamarsi all'infinito. Nelle funzioni ricorsive è fondamentale avere una **condizione di terminazione**!

- 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()**

↓
Questo metodo mi restituisce il numero di ricorsioni massimo che posso avere prima che Python mi segnali un **RecursionError**!

Example: Countdown

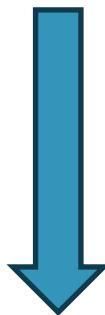
- Let's try writing down a countdown, recursively



Example: Factorial

Factorial definition

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



Equivalent recursive expression

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

Condizione terminale della ricorsione

Growing call stack

$$\begin{aligned} 4! &= 4 * 3! &= 4 * 6 &= 24 \\ 3! &= 3 * 2! &= 3 * 2 &= 6 \\ 2! &= 2 * 1! &= 2 * 1 &= 2 \\ 1! &= 1 \end{aligned}$$

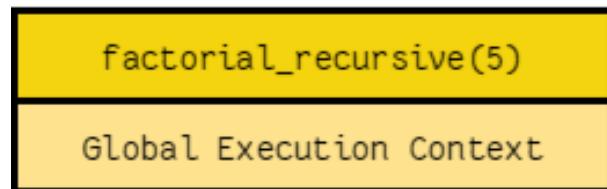
Unwinding call stack

Le chiamate superiori del metodo ricorsivo vengono terminate quando ottengo il risultato delle chiamate del metodo a livello inferiore!

Il fattoriale è un esempio di definizione ricorsiva!

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



5!

Growing Call Stack

Example: Binomial

- Compute the Binomial Coefficient $\binom{n}{m}$ exploiting the recurrence relations (derived from Tartaglia's triangle):

$$\begin{cases} \binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m} \\ \binom{n}{n} = \binom{n}{0} = 1 \quad \text{Condizione terminale della ricorsione} \\ 0 \leq n, \quad 0 \leq m \leq n \end{cases}$$

• Anche il coefficiente binomiale ha una definizione ricorsiva!

Maintaining the state

Quello che abbiamo fatto nella funzione fattoriale e in quella binomiale è passare lo stato da un'iterazione ad un'altra, passando all'interno della funzione ricorsiva un nuovo valore in input. In generale, queste funzioni possono avere degli input e potenzialmente anche uno stato interno. Come facciamo a gestire questa informazione sullo stato interno della funzione e passare lo stato interno di una funzione ricorsiva alla stessa funzione ricorsiva che chiamo all'interno della funzione stessa?

1. Possiamo utilizzare una variabile globale → DA EVITARE!
2. Nella chiamata della funzione ricorsiva possiamo passare degli input che trasferiscono le informazioni alla chiamata successiva.
3. In alcuni casi può essere utile mettere il nostro metodo ricorsivo in una classe che dentro abbia un attributo di stato, che quindi può essere visibile dentro la ricorsione.

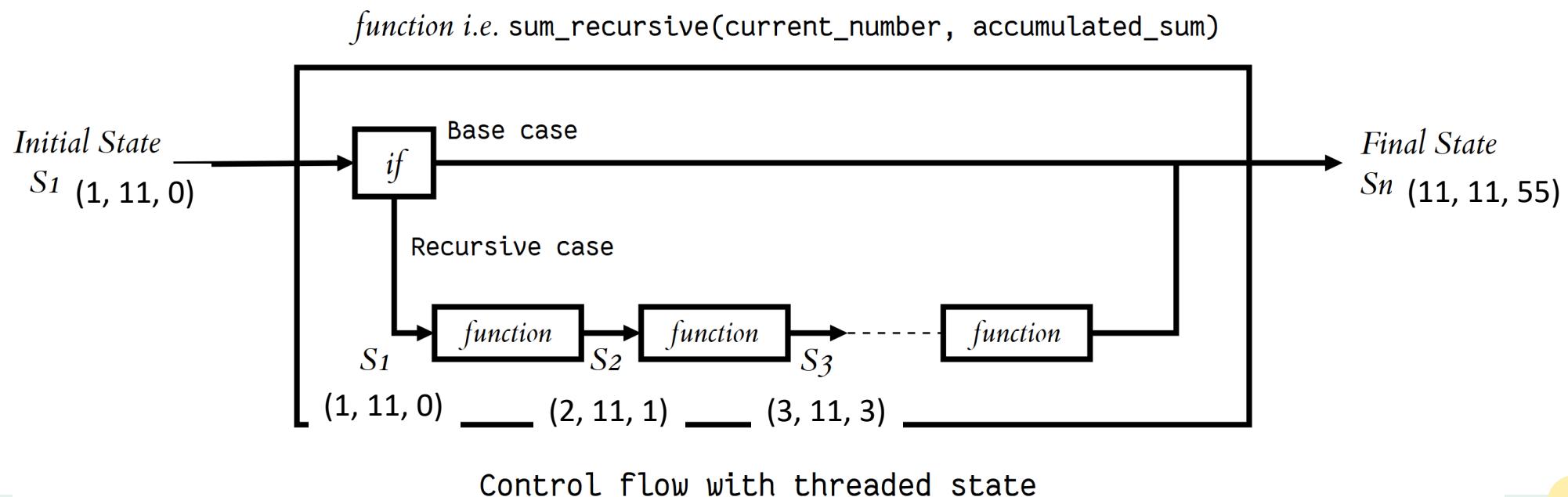
- Each recursive call has its own execution context
- To maintain state, from one recursion level to another, one can:
 - Thread the state through each recursive call so that the current state is part of the current call's execution context
 - Encapsulate the recursive function within a class, using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



Maintaining the state

• Esempio di passaggio di informazioni sullo stato di una funzione tramite `input` alla chiamata!

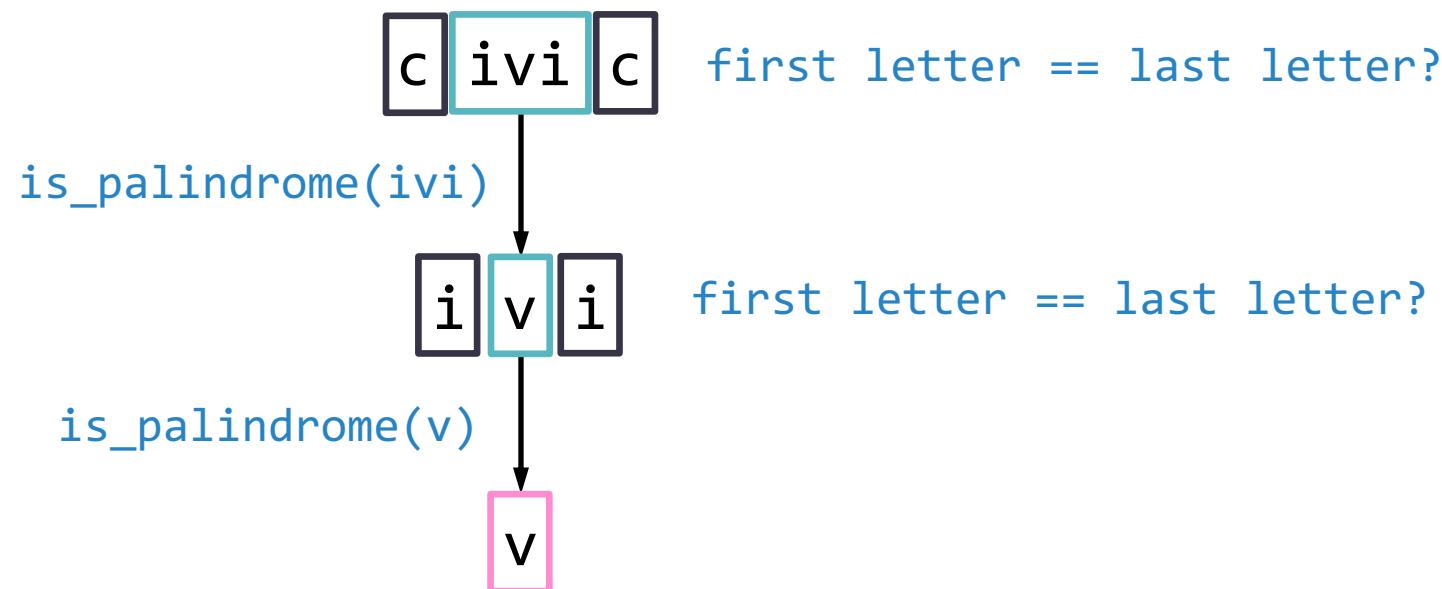
```
def sum_recursive(level, N, accumulated_sum):  
    if current_number == N:  
        return accumulated_sum  
    else:  
        return sum_recursive(level + 1, N, accumulated_sum + level)
```



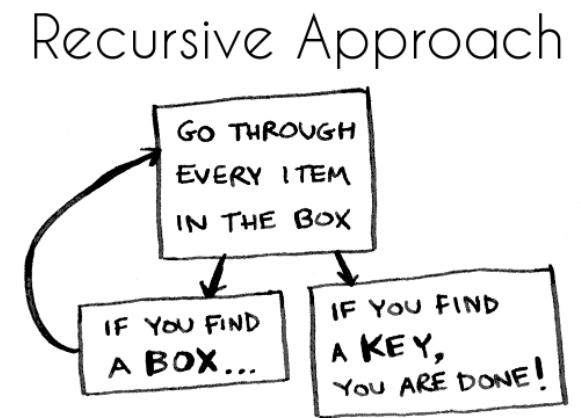
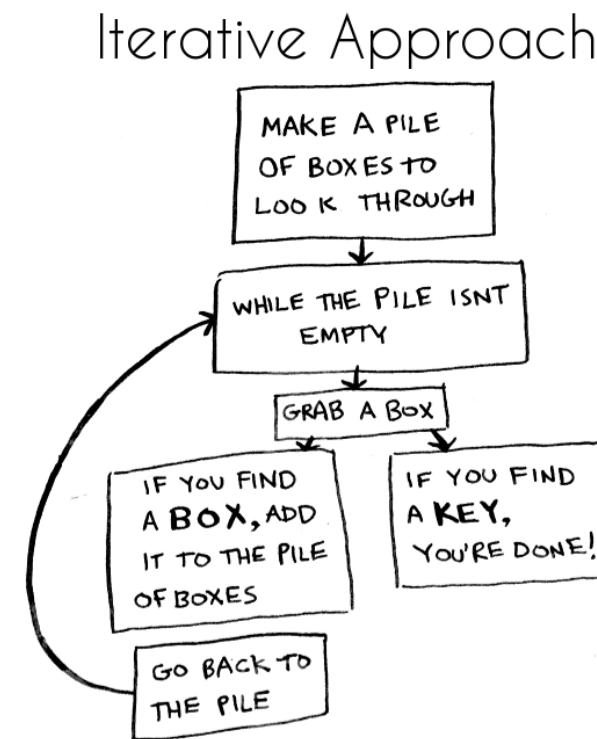
Example: Palindrome checking

- 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)

• Verifica delle parole **palindrome** in maniera
vicorsiva!



ITERATION VS. RECURSION



Tipi di Ricorsione

Ci sono tre tipi di problemi principali in cui possiamo applicare la ricorsione in modo:

- dividere et impera.
- enumerazione
- strutture dati.

• **Dividi ET IMPERA:** Lo schema ricorsivo di questa strategia ricorsiva è il seguente:

Questa strategia coinvolge la suddivisione del problema in sottoproblemi + piccoli, risolvendo ciascun sottoproblema separatamente (tramite chiamate alla funzione ricorsiva), risolvendo ciascun sottoproblema separatamente e quindi combinando le soluzioni x ottenere la soluzione al problema originale.

```
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
```

Praticamente si ha un problema grande con una soluzione non banale e vogliamo ridurlo al problema in sottoproblemi sempre + piccoli fino ad ottenere un problema la cui soluzione è banale e quindi possiamo risolverlo senza ulteriori chiamate ricorsive al metodo.

In questi metodi avremo (come anche negli altri) una condizione di terminazione che rappresenteremo con un **if** e rappresenta il caso in cui si è arrivati al problema banale, in cui possiamo ritornare direttamente il risultato del problema. Quando nelle iterazioni ricorsive si arriva al problema banale e quindi alla condizione di terminazione, la funzione "ritorna indietro" ritornando valori alle rispettive chiamate ricorsive fino ad arrivare alla soluzione del problema iniziale. Non abbiamo **backtracking** in questo caso.

• **ENUMERAZIONE:** Lo schema ricorsivo di questa strategia ricorsiva è il seguente:

Questa strategia coinvolge l'esplorazione di tutte le possibili soluzioni del problema e la selezione di quelle ottimali o quelle che soddisfano tutti i requisiti.

Soltanamente in questi metodi lavora sempre con una struttura dati "parziale" che il metodo riceve in input & chiamata effettuata; A livello di ricorsione, andrò ad aggiungere un elemento a questa struttura dati e quando raggiunge una certa lunghezza "entra" nella condizione di terminazione, all'interno della quale posso aggiungere la mia soluzione parziale ad una struttura dati che conterrà tutte le + soluzioni (solitamente conviene aggiungere una copy.deepcopy(parziale) nella lista di soluzioni, e non solo le riferimenti a parziale)

```
def recursion(..., level):  
    // E - instructions that should be  
    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
```

N.B: Date che l'obiettivo dell'algoritmo ricorsivo è aggiungere un elemento a parziale A livello di ricorsione, la condizione di terminazione sarà sempre qualcosa del tipo **if len(parziale) == n:** dove n è il numero di elementi che mi aspetto esserci all'interno di una soluzione qualsiasi. Nel for, prima di chiamare la ricorsione a livello successivo, posso anche impostare dei filtri con degli if. Nei cicli andrò a sviluppare l'algoritmo ricorsivo; ci sarà solitamente un for. All'interno del ciclo, andrò ad aggiungere solitamente un elemento a parziale, e richiamerò il metodo ricorsivo (entro nel livello successivo di ricorsione passando in input parziale con l'elemento appena aggiunto) Dopo di che, devo continuare a uscire nel for nel livello di ricorsione in cui mi trovo, e x poterlo fare devo tornare alla situazione prima del for, e quindi nel for faccio **backtracking**, rimuovendo da parziale l'elemento aggiunto a quell'iterazione.

Iteration vs. Recursion

Ogni problema **ricorsivo** ha SEMPRE un'implementazione **iterativa** dello stesso problema!

- 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

Bisogna scegliere sempre la soluzione + efficiente!

Why recursion?

Recursion comes handy in quite a few cases

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

Quando va usata la ricorsione?

Motivation

• **Dividi et Impera:** Scompongo un problema grosso in problemi + piccole; combinandone le soluzioni, ottengo il risultato del problema iniziale!

- 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



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 $\{Q_i\}$, then merge the solutions *↳ i problemi avranno soluzioni banali!*
- Merge & split must be “simple”
- A.k.a., *Divide 'n Conquer*

Approccio "Dividi et Impera"

↳ i sottoproblemi devono essere dello stesso tipo del problema di partenza!

• Exploration

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

Approccio "esplorativo"

↳ Anche in questo caso divide un problema in sottoproblemi!

Divide et Impera – Divide and Conquer

```
Funzione ricorsiva
def solve (problem):
    sub_problems = divide(problem) ~~~~~ Divido il problema in sottoproblemi
    sub_solutions = [] ~~~~~ Lista delle soluzioni vuota inizialmente

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

    solution = combine(sub_solutions)
    return solution ~~~~~ Combinazione delle sotto-soluzioni!

solution = solve(problem)
```

• **MANCA CONDIZIONE DI TERMINAZIONE!**

↳ A sottoproblema, chiamo il metodo ricorsivo **solve**,
a cui passerò in input il sottoproblema rispettivo.

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

• La recursion va terminata con una **condizione di terminazione!**
Quando i problemi diventano molto piccoli, avranno una soluzione
bella e che verrà restituita x cui non occorrerà dividere ulteriormente
il problema in sotto-problemi!

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
```

La condizione terminale va SEMPRE messa nei metodi ricorsivi!

check termination

I metodi ricorsivi avranno sempre una condizione di terminazione dentro a un if, che verrà chiamata una volta che il problema diventa talmente banale da restituire direttamente la soluzione nell'if, oppure da poter risolvere il problema con una funzione esterna non ricorsiva, e ci sarà sempre un else, quando il problema non ha raggiunto ancora il livello di banalità necessario per poter essere risolto facilmente, e per questo motivo viene ancora suddiviso in ulteriori sotto problemi.

do recursion

Exploration

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

Il problema dell'esplorazione è abbastanza simile: avremo un set di dati, una collection, una lista ecc. e tipicamente il problema dell'esplorazione consiste nel valutare possibili combinazione di questi dati, percorsi, ecc. Quello che faremo è: dato il problema, verificare quali sono i step ammissibili, e per ognuno di questi step ammissibili eseguo lo step, lo esploro e necessiterò di un Undo: l'Undo è un cosiddetto backtracking, e serve perché se ad esempio voglio esplorare un cammino, quindi ho 10 possibili mete e voglio trovare il modo di raggiungere tutte e 10 le mete nel modo più vantaggioso possibile, considerando autobus, voli, auto ecc. —> Il metodo ricorsivo può essere: dalla lista di 10 possibili mete, ne prendo una a caso e lo metto nella mia soluzione; tra quelli rimanenti ne prendo uno a caso e lo metto nella soluzione e così via. Alla fine raggiungerò una soluzione che avrà tutte le mete. La valuto, poi quando faccio la winning stack (?), se questa soluzione è un vettore condiviso tra le varie chiamate, allora dovrò fare un passo indietro rimuovendo uno step quando torno alla chiamata precedente del metodo ricorsivo.

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

↳ *. La ricorsione può essere utile anche quando lavori con strutture dati ricorsive!*

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

funzione **ricorsiva** che chiama iterativamente partendo dalla lista vuota e passandole elementi che verranno attaccati in cima alla lista!



```
[3, "ciao", 51]
```

```
attach_head(3, ["ciao", 51])
```

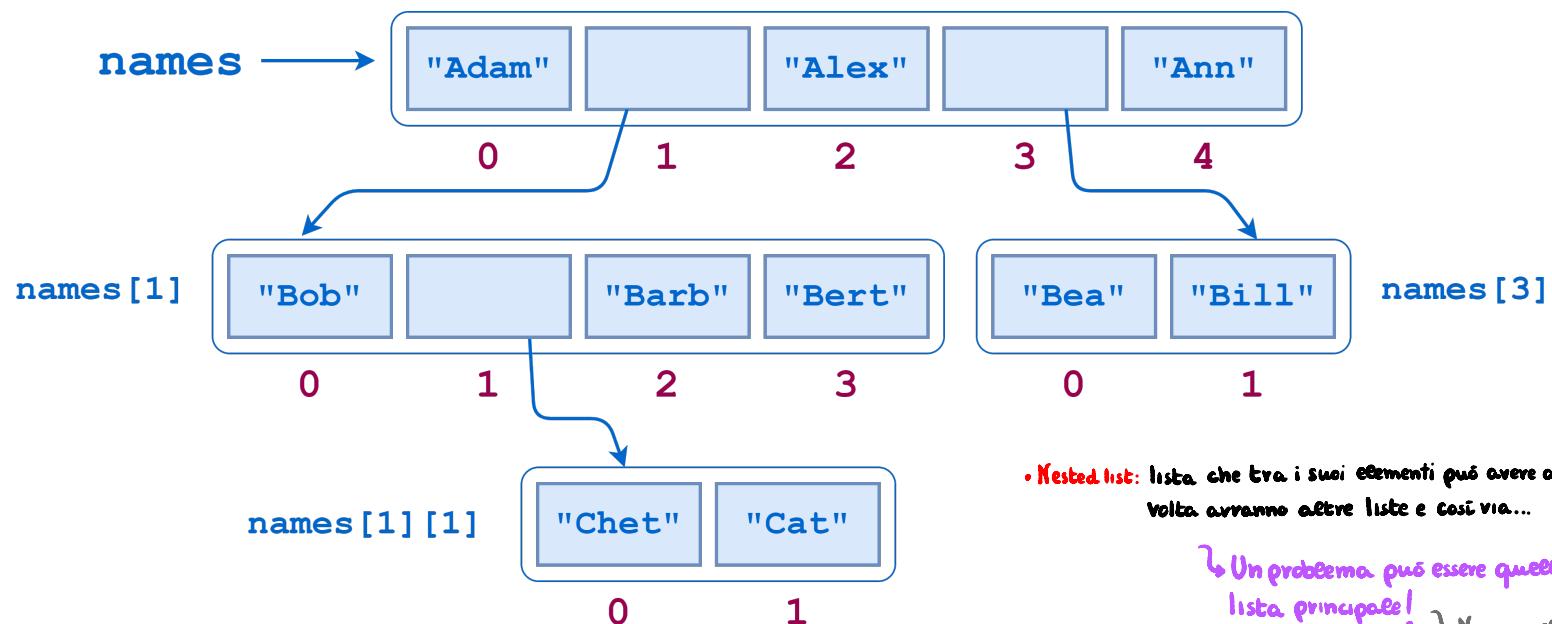
```
attach_head("ciao", [51])
```

```
attach_head(51, [])
```

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']
```



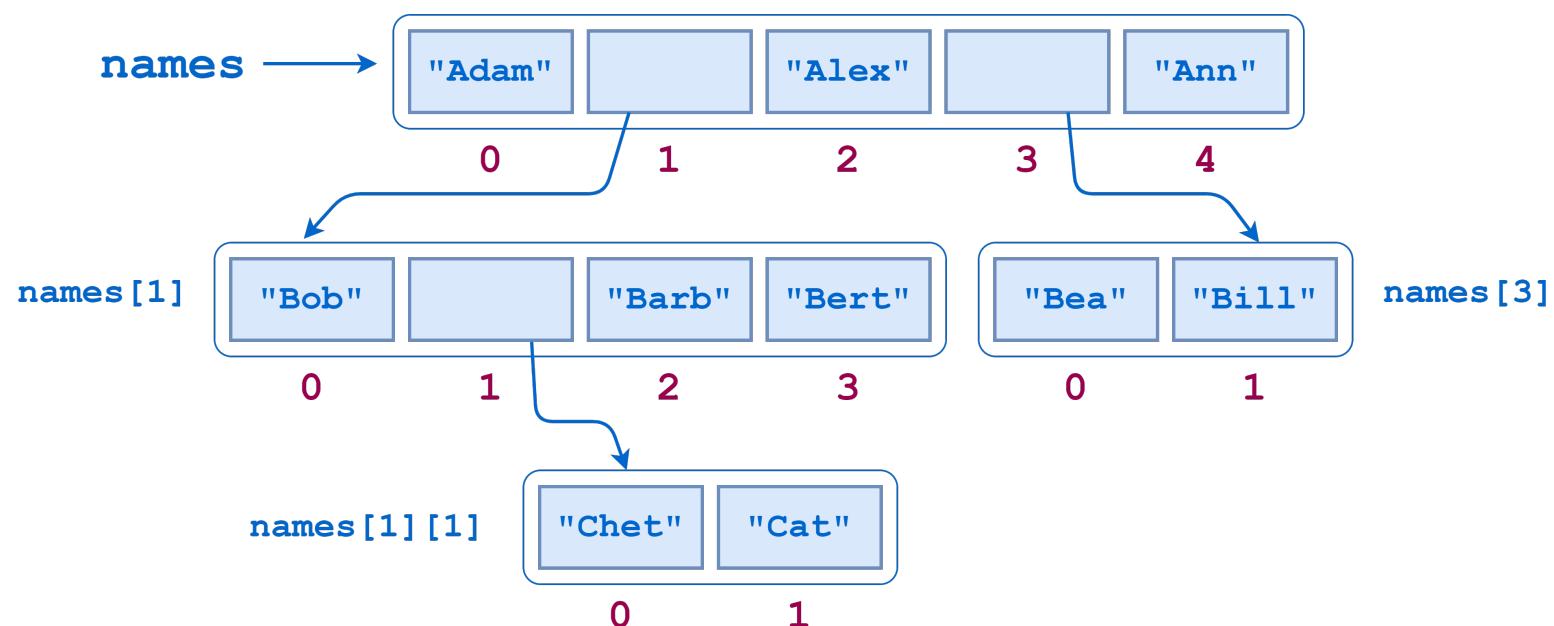
• Nested list: lista che tra i suoi elementi può avere altre liste, che a loro volta avranno altre liste e così via...

↳ Un problema può essere quello di contare i "nodi terminali" della lista principale!

↳ Numero effettivo di nomi presenti nella lista!

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

↳ *Esiste anche una soluzione iterativa per la nested list, ma risulta molto più complicata!*

Example: nested list

```
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

• La soluzione **ricorsiva** è molto + easy rispetto
alla soluzione **iterativa**!

Recursive version

```
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

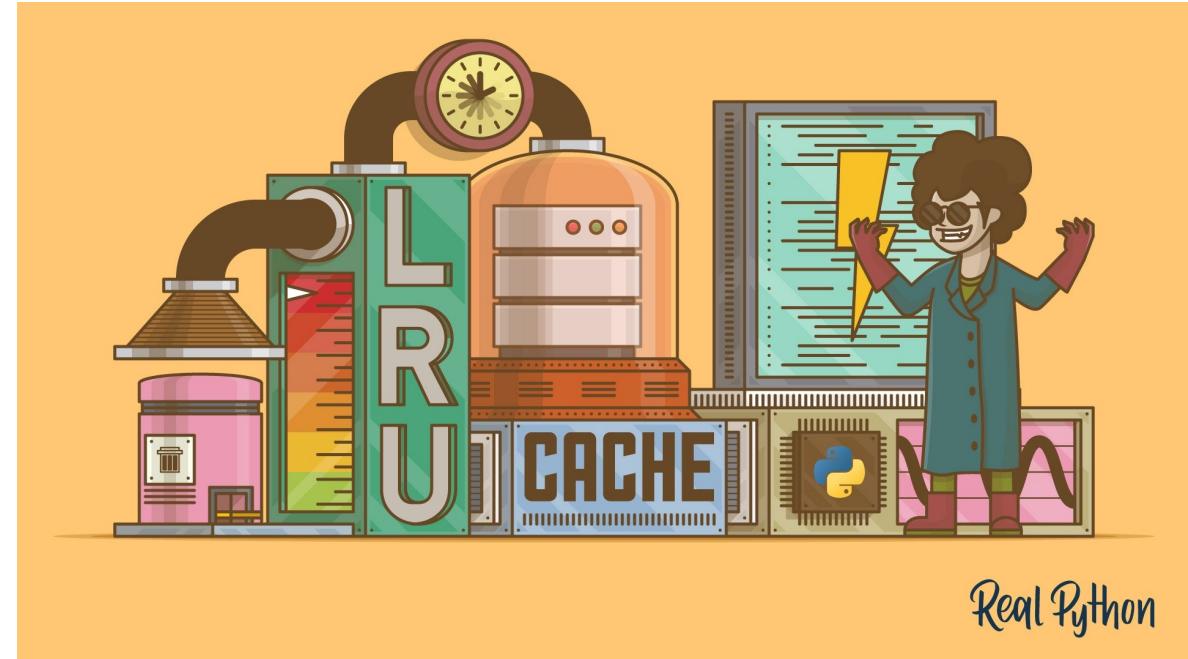
```
def count_leaf_items(item_list):
    """Non-recursively counts and returns the
       number of leaf items in a (potentially
       nested) list.
    """
    count = 0
    stack = []
    current_list = item_list
    i = 0

    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

• L'algoritmo ricorsivo farà tante chiamate, x questo motivo è importante ottimizzare l'efficienza dei metodi ricorsivi!

- How can we improve the runtime efficiency of our recursive method?
 - Use appropriate data structures (typically negligible improvements on small problems) ↗ Usando strutture dati + appropriate è possibile ottimizzare di poco l'efficienza dei metodi ricorsivi!
 - **Skip recursion threads that do not yield results** (can bring massive improvements) ↗ Evitare di fare chiamate ricorsive là dove non serve può portare grossi miglioramenti nell'efficienza del metodo ricorsivo! ↗ Lo posso fare aggiungendo ulteriori condizioni con if nell'else del metodo ricorsivo!
 - **Cache intermediate results**, if the corresponding sub-problem is encountered multiple times (improvements depend on the problem, there is a memory cost.)

↳ **Caching:** tecnica di programmazione in cui si memorizzano dei risultati intermedi ; in questo modo, se re-incontro gli stessi parametri / risultati, altre volte, non c'è bisogno che me lo calcoli perché avrò già il risultato salvato da qualche parte.

Fibonacci sequence

• Sequenza di Fibonacci: esempio in cui le Caching migliora molto l'efficienza del metodo ricorsivo.

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

$$F(0) = 0$$

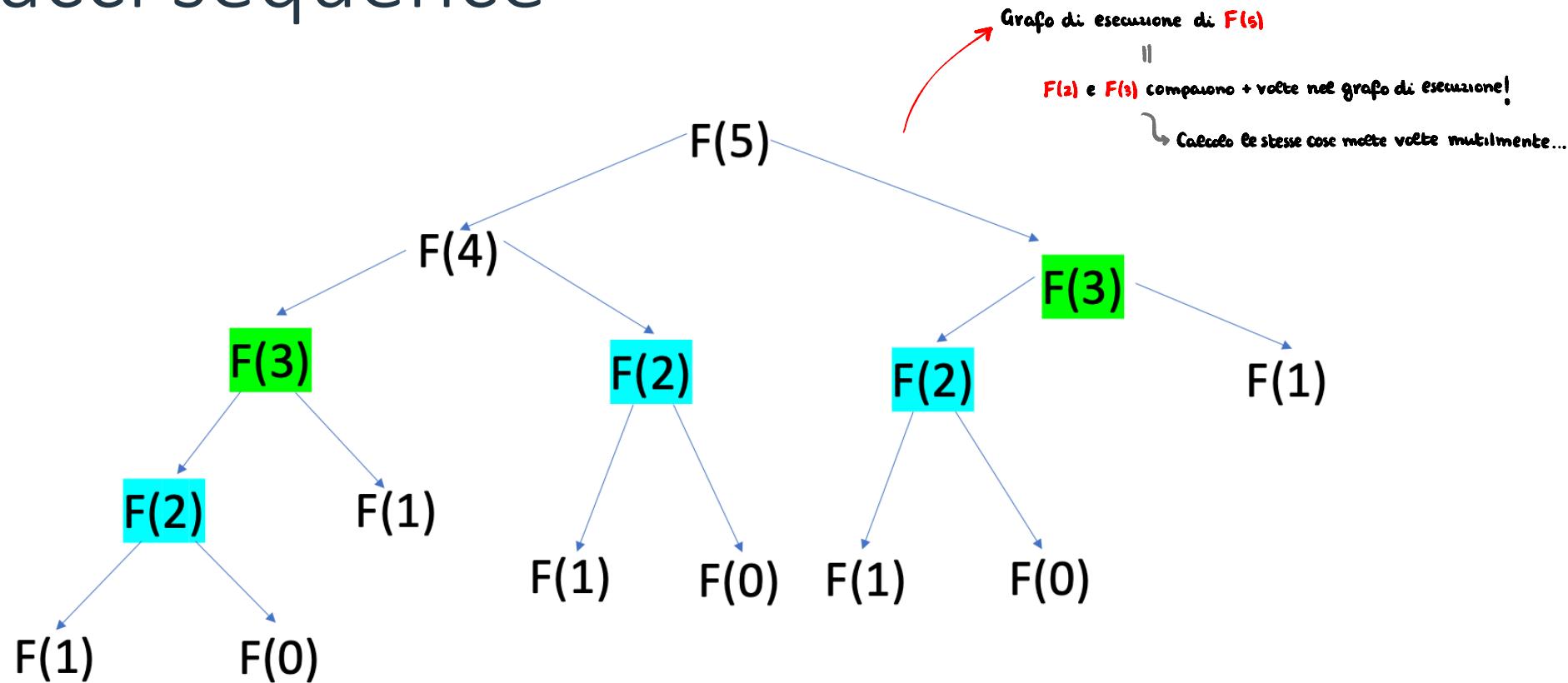
$$F(1) = 1$$

$$F(n) = F(n - 1) + F(n - 2)$$



0, 1, 1, 2, 3, 5, 8, 13, 21, 34 , 55, ...

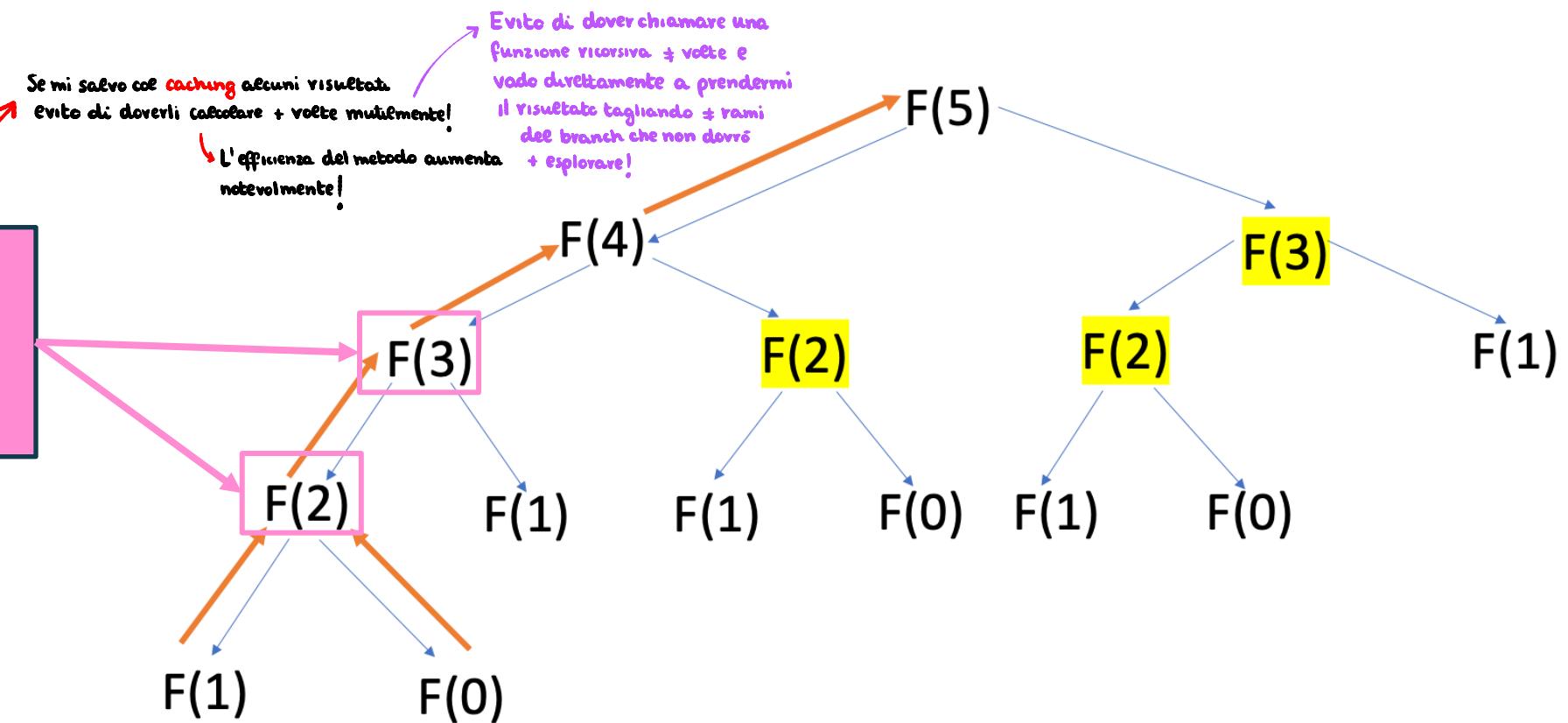
Fibonacci sequence



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

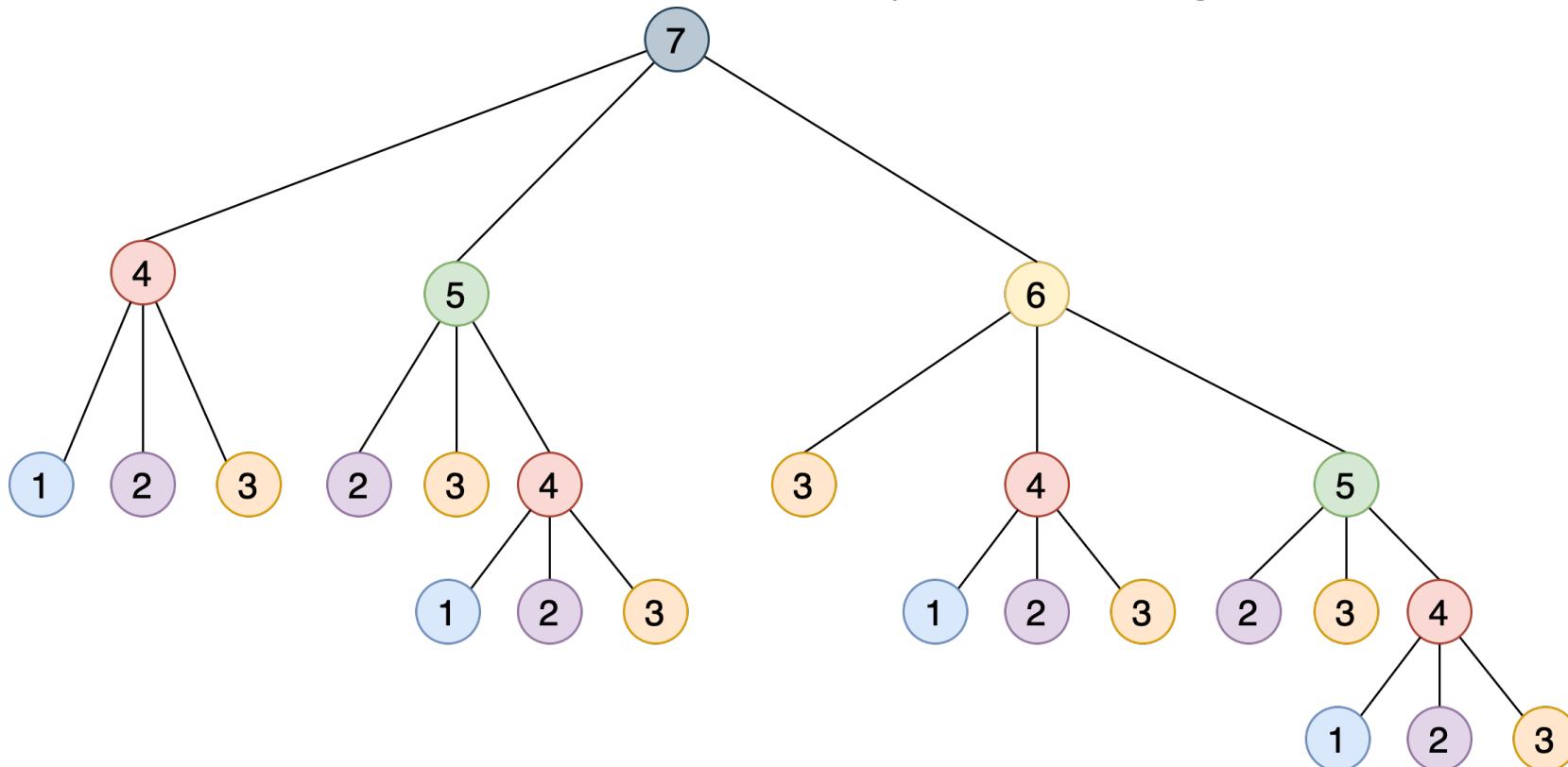


Let's implement this!

Memoization

~ *Tecnica di Caching*

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

Per effettuare **caching** ho due strade:

- usare un **dizionario**, dove le chiavi sono l'argomento della funzione e il valore è il risultato della funzione con quegli argomenti.
- usare **functools**, libreria di Python che implementa alcune funzionalità che fanno già il caching tramite dei decoratori: **@lru_cache**, è una cache che si salva le chiamate + recenti del metodo ricorsivo. La dimensione di questa cache è definita con il parametro **maxsize**. Se metto **maxsize=None**, allora la cache si salverà tutte le chiamate del metodo ricorsivo.

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
```

lru_cache implementa un dizionario in cui si salva
la sequenza di argomenti il valore di ritorno della
funzione => **N.B.**: Gli argomenti della funzione devono
essere **hashable**, altrimenti non potrei usarli come
chiavi del dizionario.

- **@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.

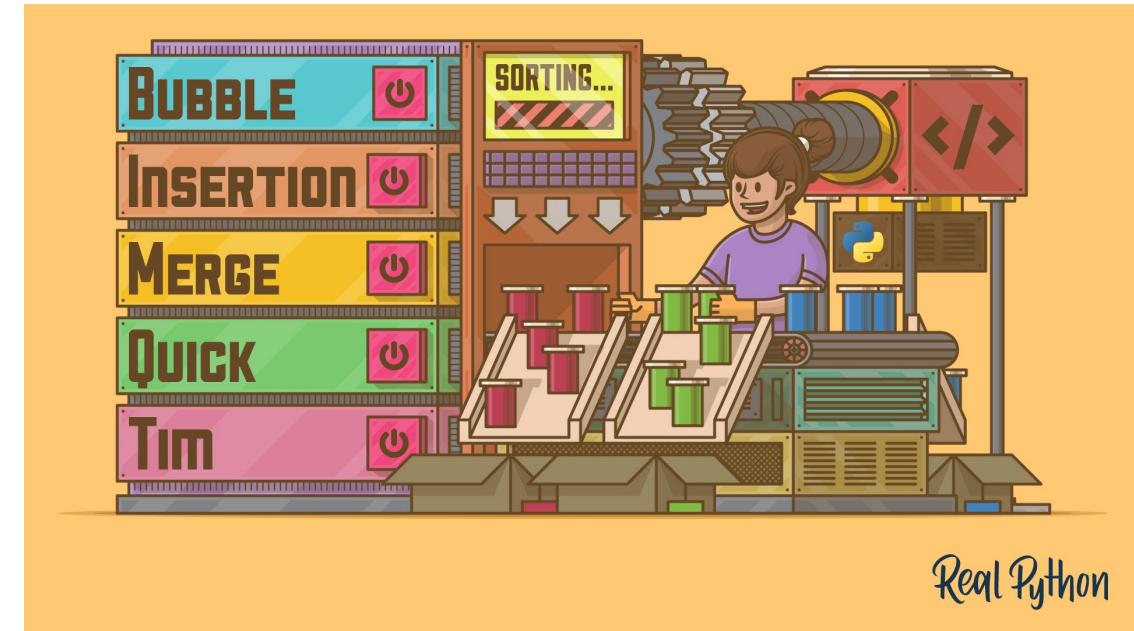
<https://docs.python.org/3/library/functools.html>

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)

 Il **caching** ha senso solo se devo chiamare la stessa funzione con gli stessi parametri + volte.

SORTING AND SEARCHING WITH RECURSION



Real Python

Example: Quicksort

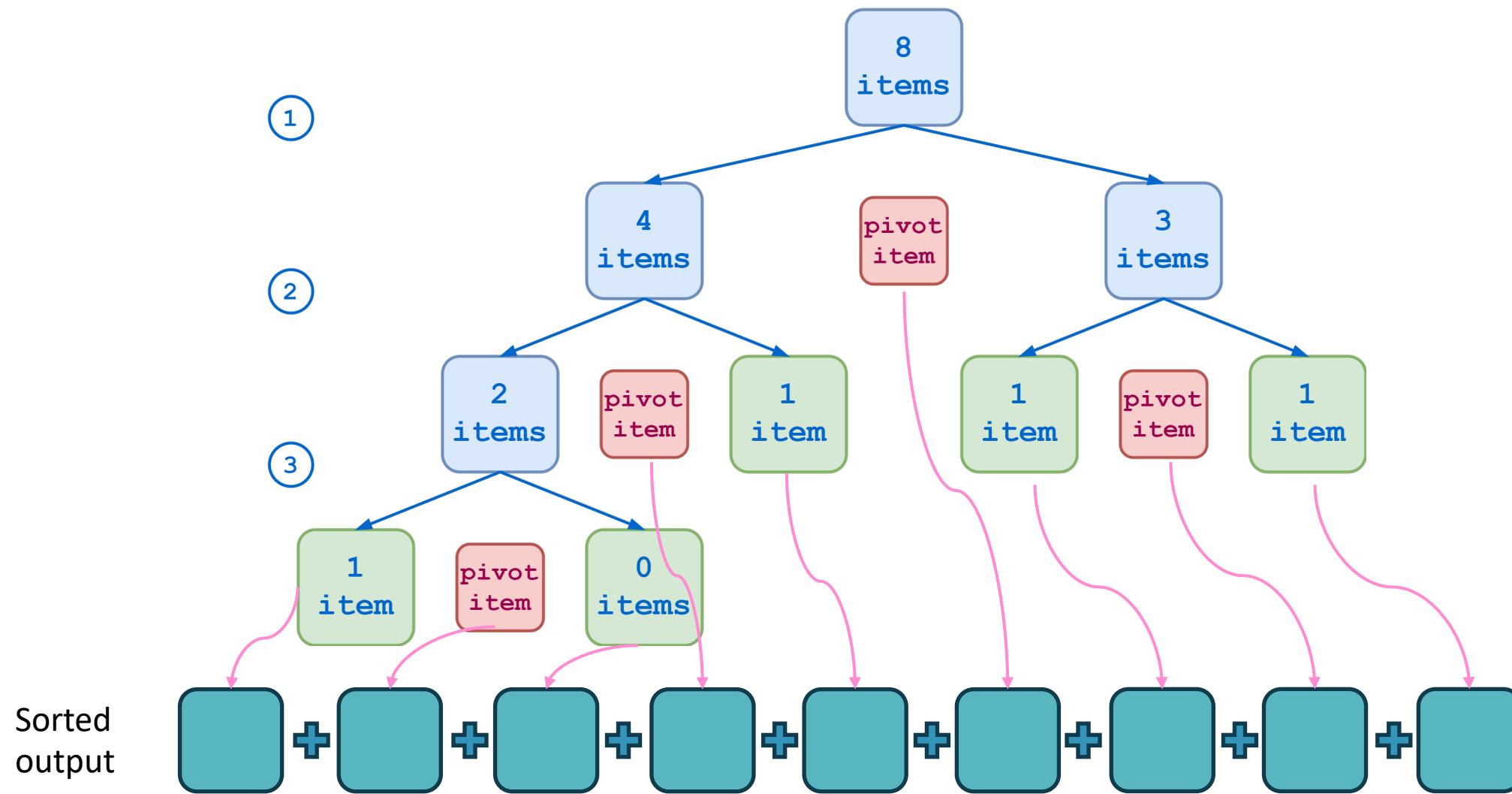
• Esempio applicativo in cui la ricorsione risulta effettivamente utile!

↳ L'obiettivo della **quicksort** è quello di ordinare in modo crescente una lista di **N** elementi.

- 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

• Esempio di **quicksort** su una lista di 8 elementi.

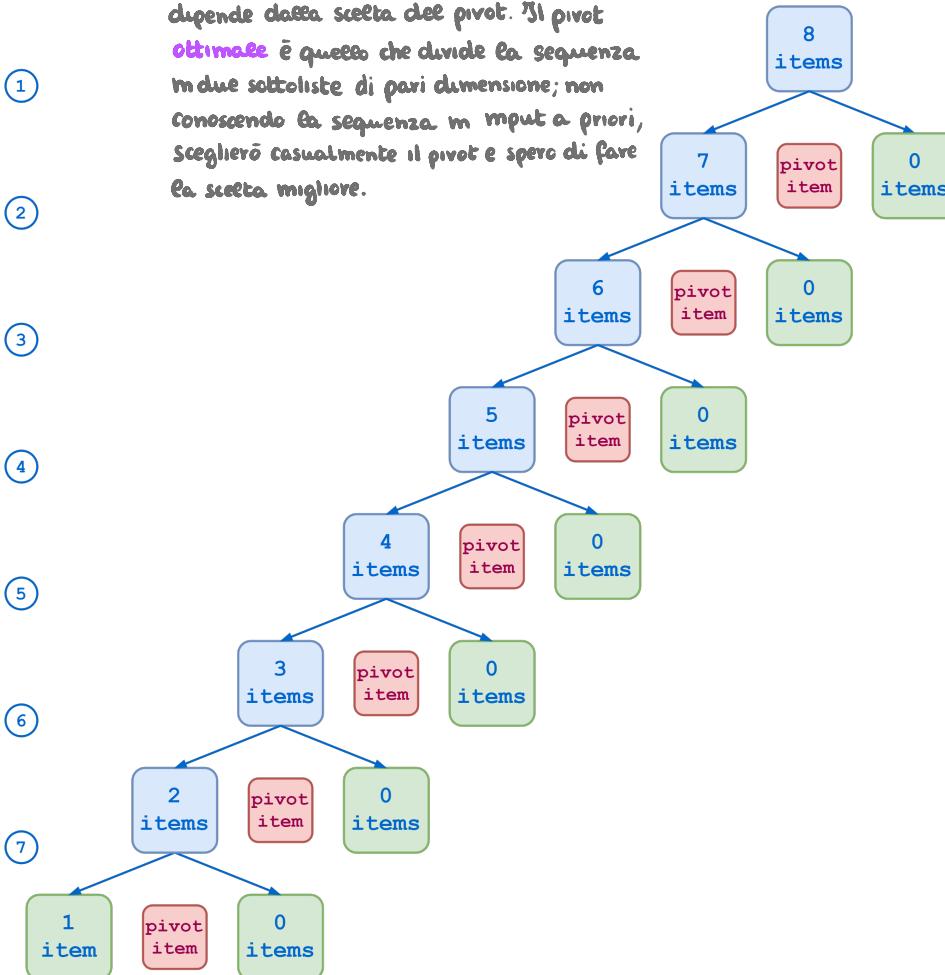


Example: Quicksort

- 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)

• La scelta del **pivot** sta a noi: ad esempio, possiamo scegliere l'elemento a metà!

L'efficienza dell'algoritmo di **quicksort** dipende dalla scelta del pivot. Il pivot **ottimale** è quello che divide la sequenza in due sottosequenze di pari dimensione; non conoscendo la sequenza in input a priori, sceglierò casualmente il pivot e spero di fare la scelta migliore.

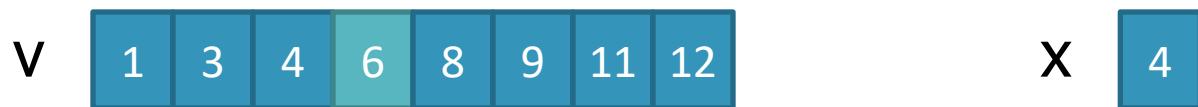


Example: dichotomic search

In cosa consiste la **Dichotomic Search**? Devo vedere se un elemento è presente in una determinata lista. Come faccio? Ricevo una lista ordinata in input, divido la lista in due sottoliste; se l'elemento è presente nella sottolista 1 (lo posso sapere confrontando l'elemento cercato con l'elemento in mezzo tra le due sottoliste in cui ho fatto la divisione), continuo, dividendo la sottolista in due ulteriori sottoliste, altrimenti, la sottolista può essere ignorata. Vado a cercare quindi l'elemento in sottoliste sempre più piccole, fino a rendere il problema banale quando ottengo una lista con un solo elemento.

- 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

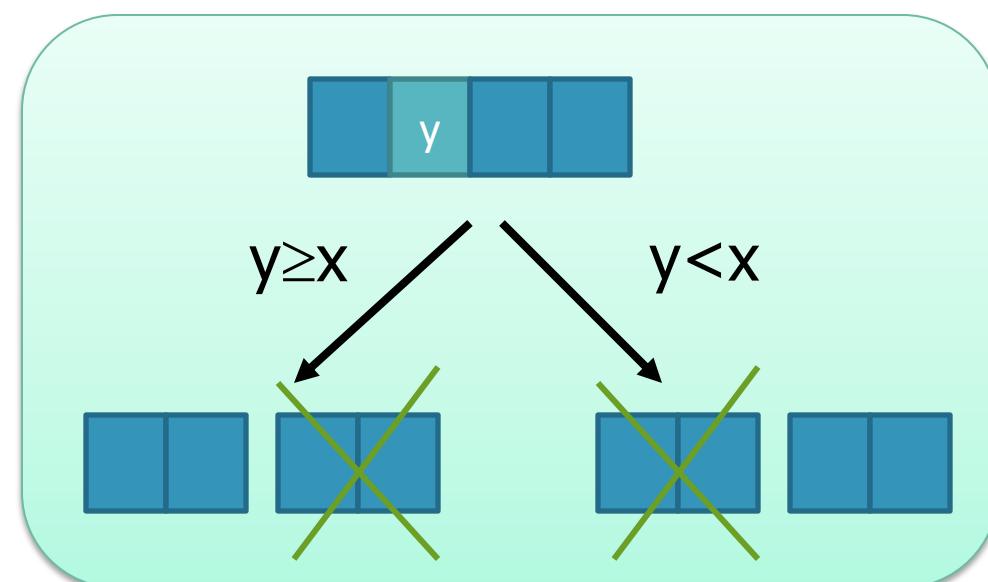
Example: dichotomic search



Example: dichotomic search

v [1 | 3 | 4 | 6 | 8 | 9 | 11 | 12]

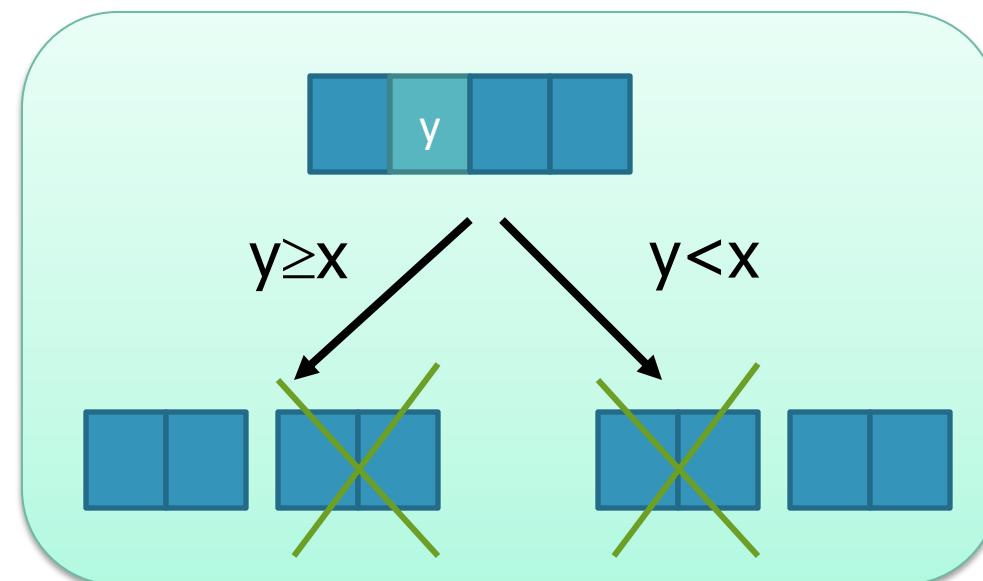
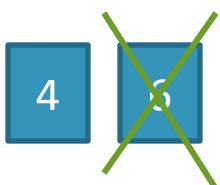
x [4]



Example: dichotomic search

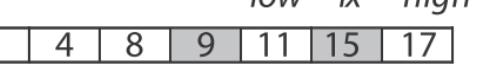
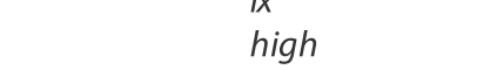
v [1 3 4 6 8 9 11 12]

x 4



Example: dichotomic search

Alternative iterative solution

BINARY SEARCH			Array	Divide and Conquer
Best	Average	Worst		
O (1)	O ($\log n$)	O ($\log n$)		
search (A, t)			<i>search (A, 11)</i>	
1. $low = 0$			<i>first pass</i> 	
2. $high = n - 1$			<i>second pass</i> 	
3. while ($low \leq high$) do			<i>third pass</i> 	
4. $ix = (low + high)/2$				
5. if ($t = A[ix]$) then				
return true				
7. else if ($t < A[ix]$) then				
$high = ix - 1$				
9. else $low = ix + 1$				
10. 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
```

• IMPOSTAZIONE METODO RICORSIVO:

- All'inizio, andrò a mettere eventuali istruzioni che devono sempre essere runnate inizialmente nel metodo; solitamente queste istruzioni NON sono presenti.
- **Condizione terminale:** all'inizio del metodo ricorsivo va SEMPRE aggiunta la condizione terminale che, se verificata, mi permette di uscire dal metodo ricorsivo.
- **else:** Se non entro nell'if della condizione terminale, entro nell'else dove andrò a scorporre il problema + grosso in sottoproblemi + semplici su cui andrò a chiamare nuovamente il metodo ricorsivo. Nell'else posso aggiungere eventualmente dei filtri con **if** x evitare di chiamare il metodo ricorsivo multilente x aumentarne l'efficienza.

• Back tracking:

Il backtracking è una tecnica di programmazione che viene spesso utilizzata insieme alla ricorsione per risolvere problemi che richiedono di esplorare tutte le possibili combinazioni di una soluzione. Consiste nell'esplorare tutte le possibili opzioni disponibili, ma, quando si raggiunge un punto in cui si determina che una soluzione parziale non può portare a una soluzione finale accettabile, si "torna indietro" (backtrack) e si esplora una strada diversa.

In termini più semplici, il backtracking funziona esplorando tutte le scelte possibili fino a quando non si trova la soluzione desiderata o si scopre che non esiste una soluzione. Se si raggiunge uno stato in cui non si può procedere ulteriormente, si fa marcia indietro e si riprova con un'altra opzione.

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.

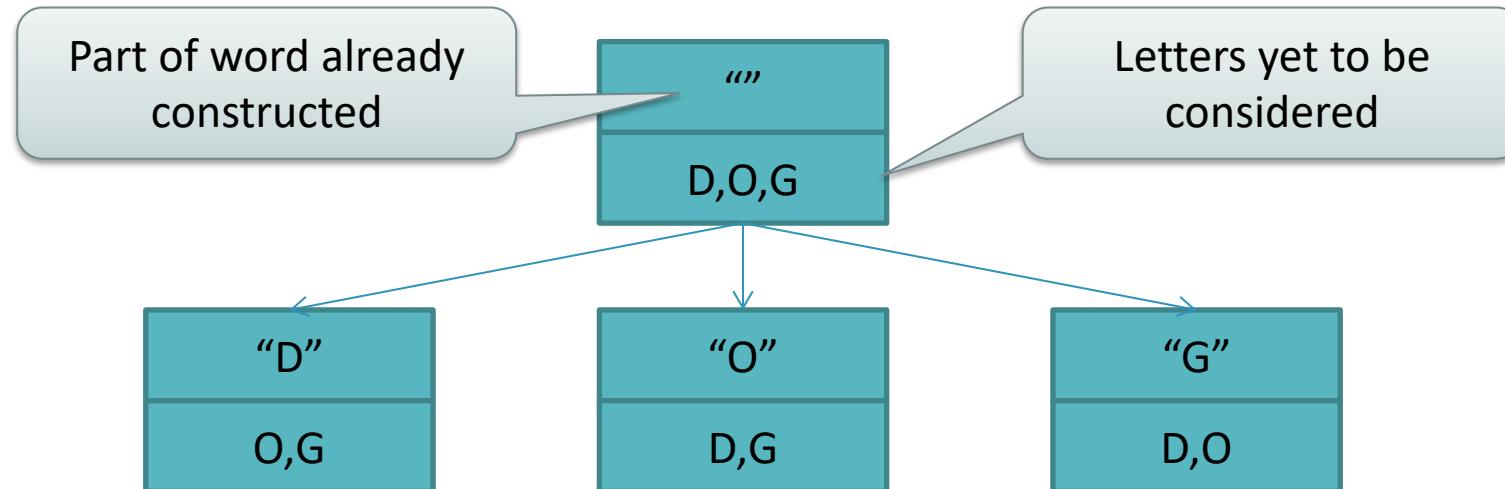
Anagrams

- 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

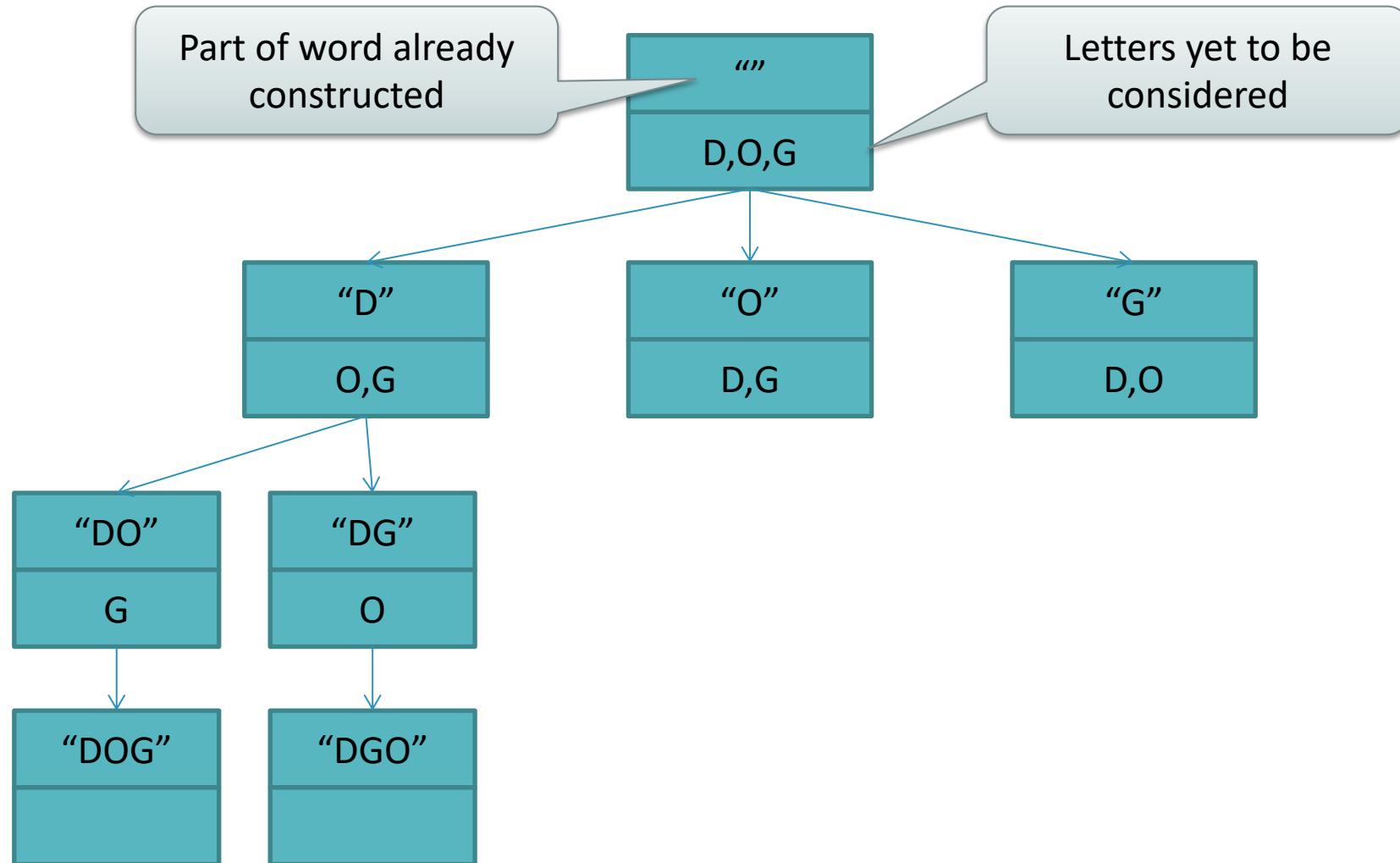
Anagrams: recursion tree



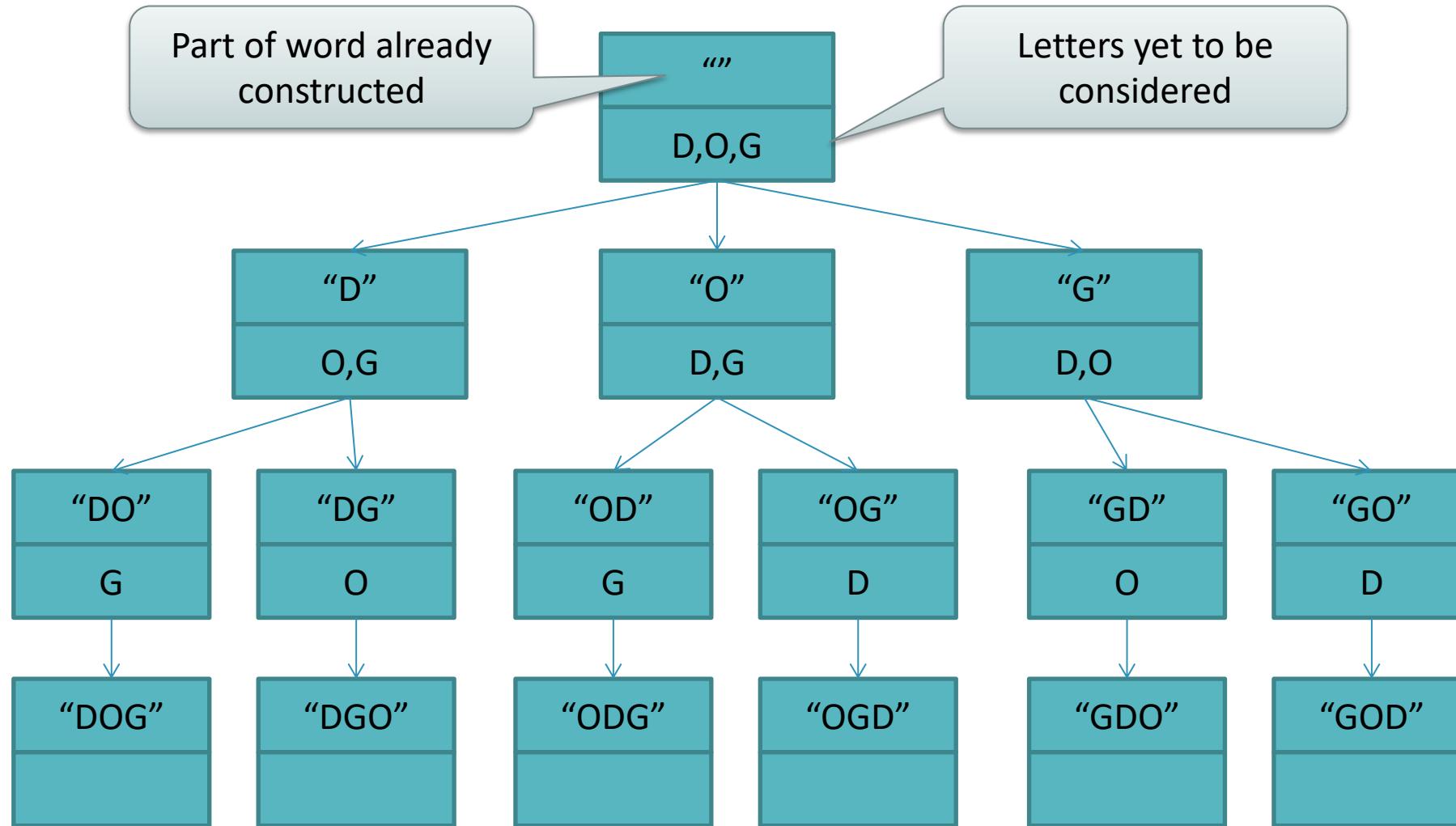
Anagrams: recursion tree



Anagrams: recursion tree



Anagrams: recursion tree

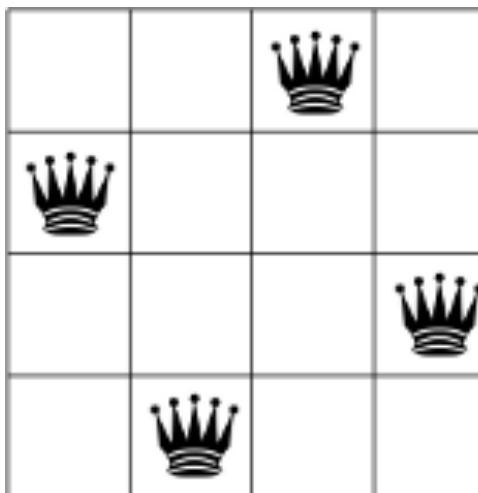


Anagrams variants

- 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” → **s**e**a**s and **s**e**a**s 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 **a**)
 - Use a set to avoid repetitions

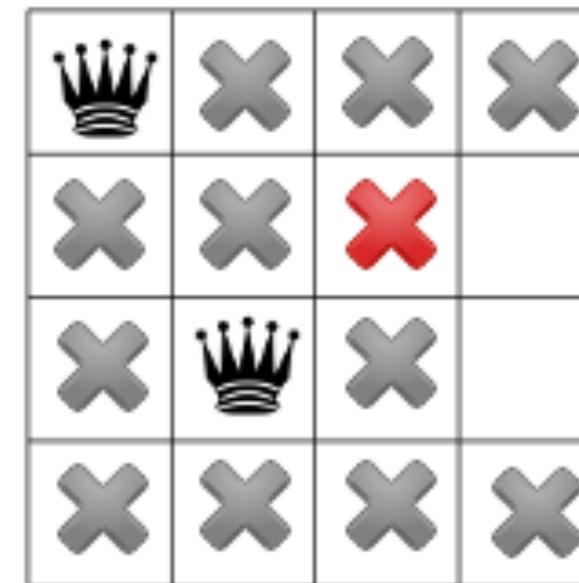
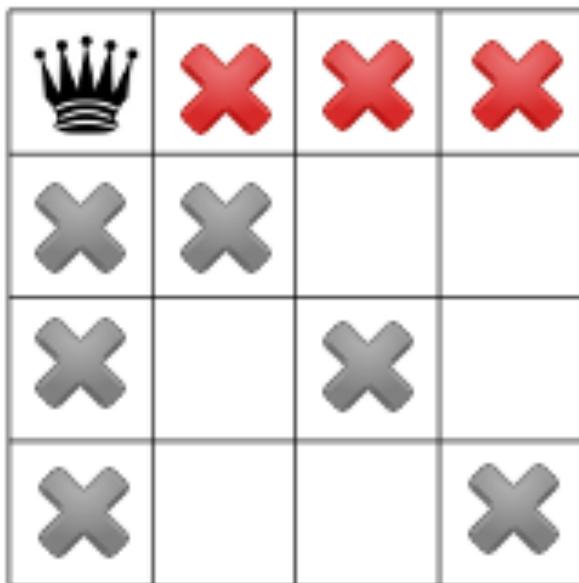
N-Queens

- In chess, a queen can attack horizontally, vertically, and diagonally.
The N-queens problem asks:
- *How can N queens be placed on an $N \times N$ 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 $1, 2, \dots, n^2$ and they are not repeated
- The constant sum is called the 'magic constant'.

2	7	6	→ 15
9	5	1	→ 15
4	3	8	→ 15

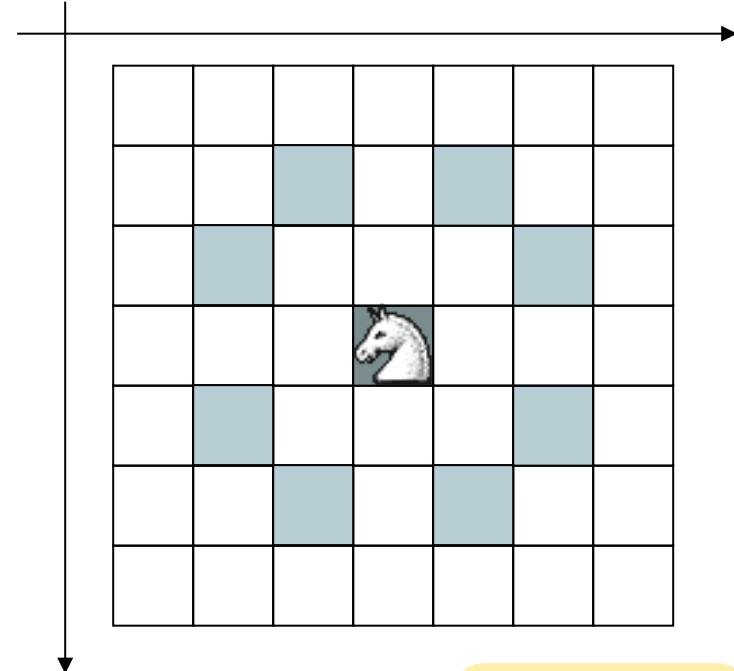
15 ↘ ↓ 15 15 15 15 ↗ 15



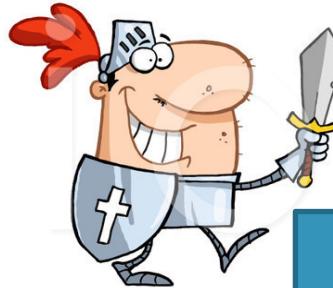
EXERCISES FOR HOME

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



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

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





License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
 - **Share** — copy and redistribute the material in any medium or format
 - **Adapt** — remix, transform, and build upon the material
 - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
 - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - **NonCommercial** — You may not use the material for commercial purposes.
 - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
 - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>