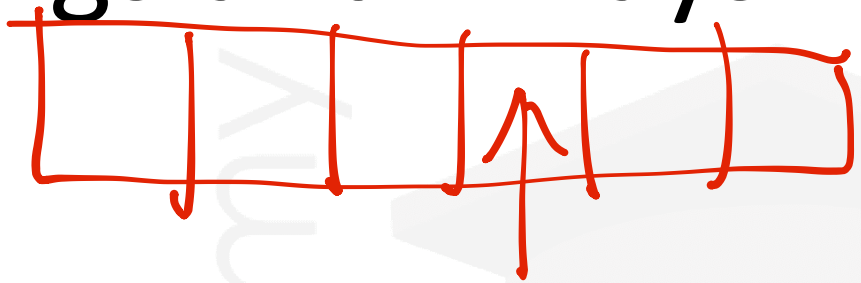
01 _ Strings and Arrays



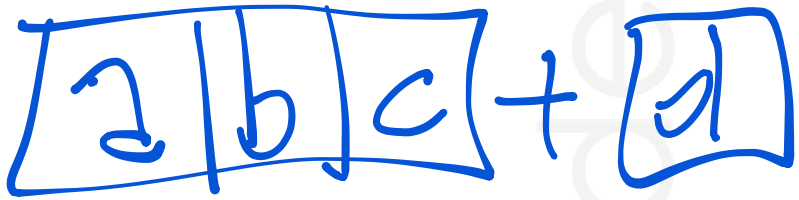
5

MYLIST = [ ]

READ O(1)

WRITE O(1)

APPEND O(1)



APPEND O(n)

```
S = []
for i in range(n):
    S += [i]
```

ITERATIVE APPEND $O(n^2)$

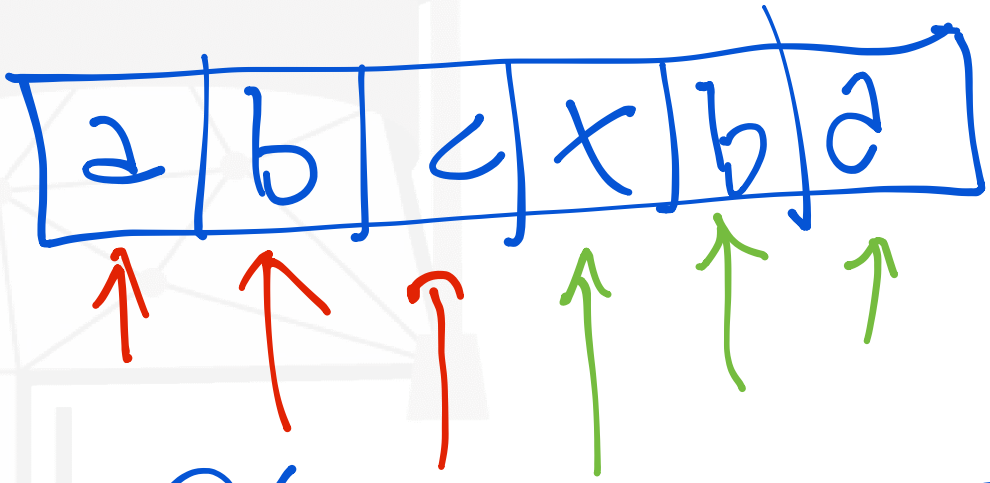
Write a function that checks if a string is palindrome

"" a ab abc abba abcba abcxba

```
s = 'abcd'
s1 = s[::-1]
return s == s1
```

tempo O(n)

spazio O(n)



t O(n)

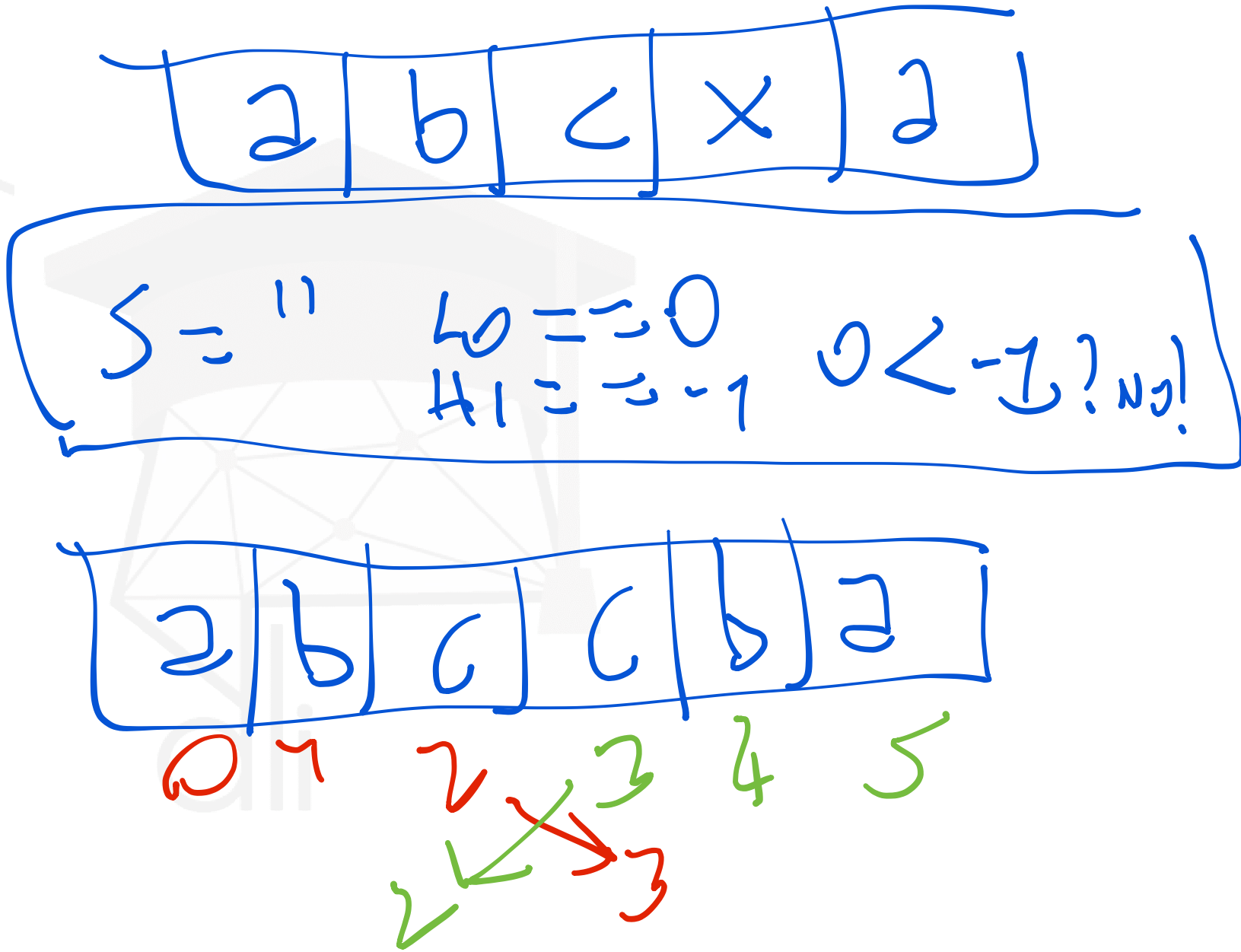
s O(1)

a == a ok

b == b ok

c == x no!

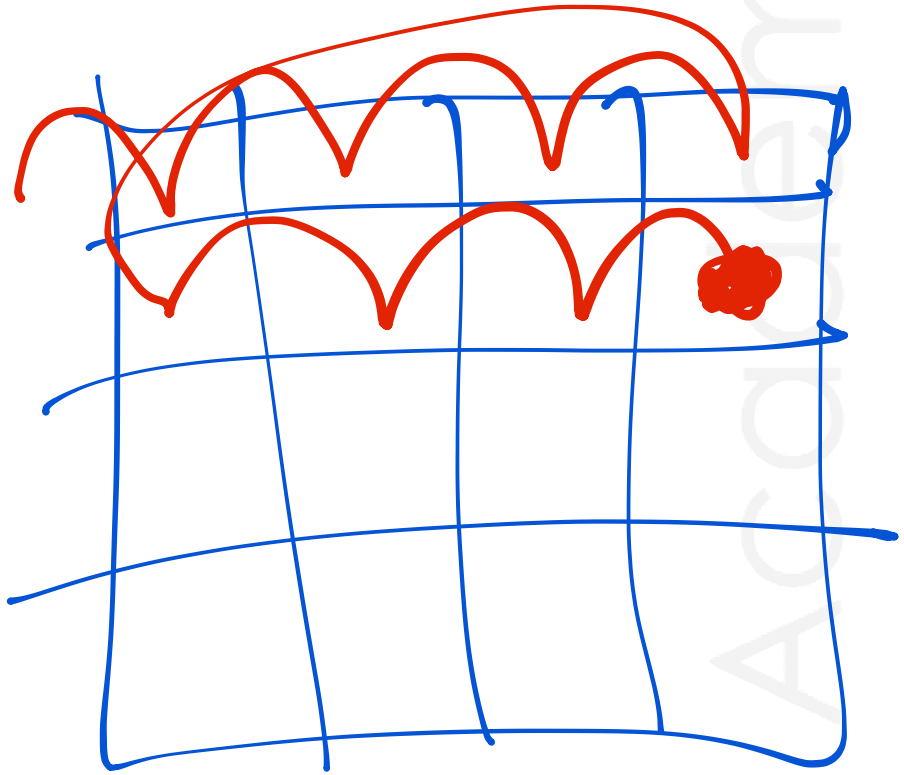
```
def is_palindrome(s):
    lo,hi = 0,len(s)-1
    while lo<hi:
        if s[lo]!=s[hi]:
            return False
        else:
            lo,hi=lo+1,hi-1
    return True
```



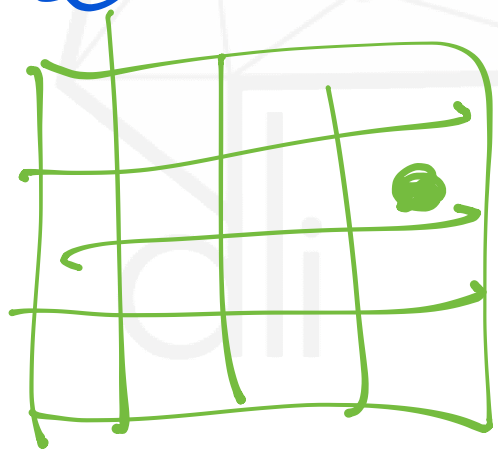
02 _ Matrices

LISTA DI LISTE [[...],[...]]
OPPURE NP.ARRAY

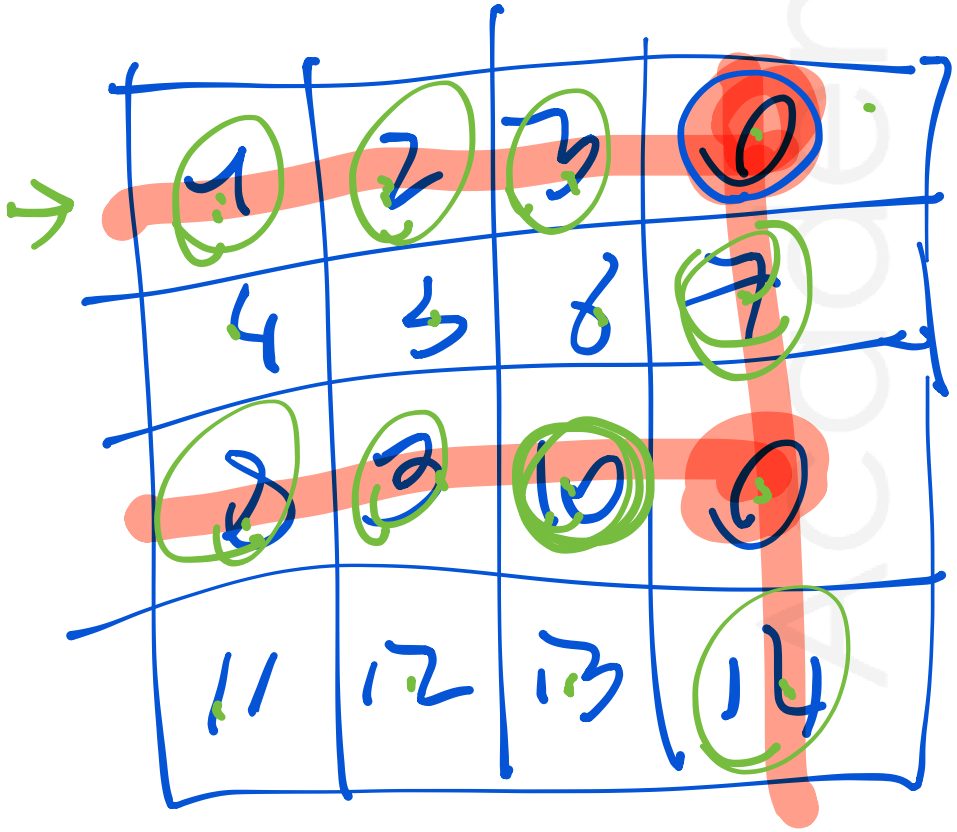
QUADRATA? NXN NXM

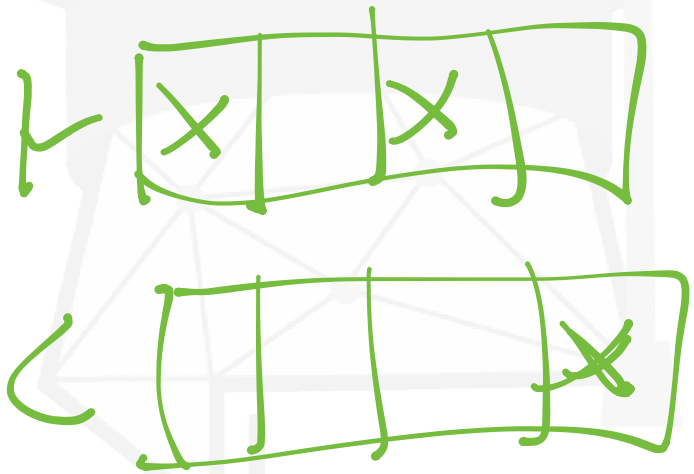


1. VISITA CON INDICI
2. RICORSIONE



Write a function that, given a matrix, finds all zeros. For each zero, all elements in the same row and column must be set to zero.



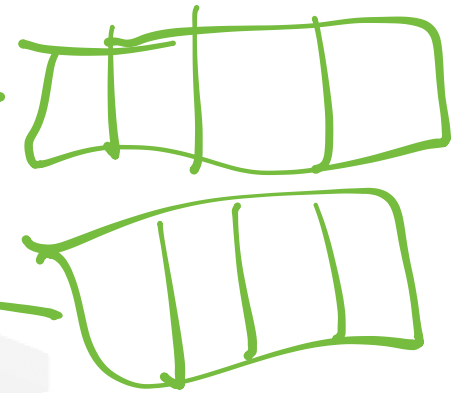


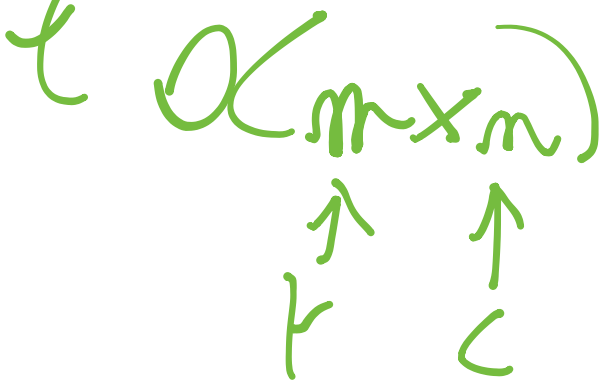
t O(m×n)
s O(m×n)

⇓

t O(m×n)
s O(m+n)

```
def nullify_rows_cols(m):
    r,c = m.shape
    null_rows = np.zeros(r,dtype=bool)
    null_cols = np.zeros(c,dtype=bool)
    for i in range(r):
        for j in range(c):
            if m[i,j]==0:
                null_rows[i]=True
                null_cols[j]=True
    for i in range(r):
        for j in range(c):
            if null_rows[i] or null_cols[j]:
                m[i,j]=0
    return m
```



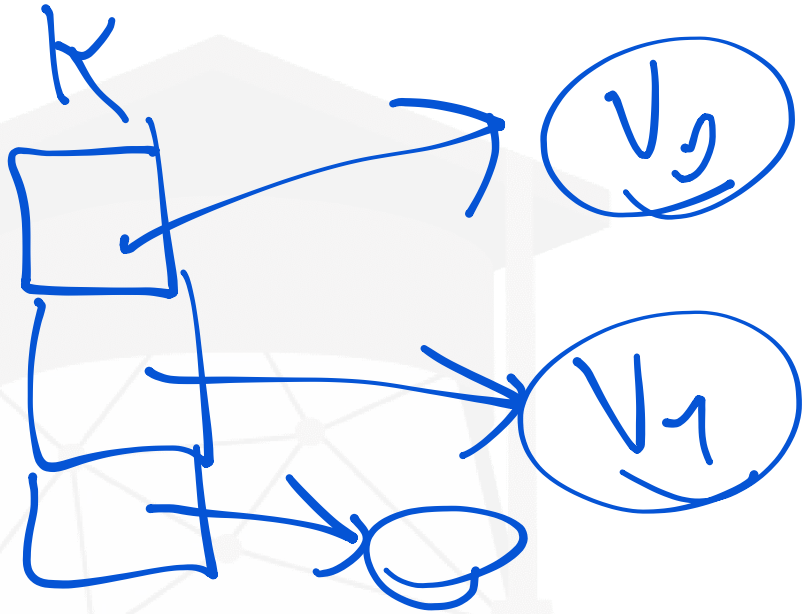


03 _ Dictionaries

HASH TABLES < DICTS

READ/WRITE

O(1)



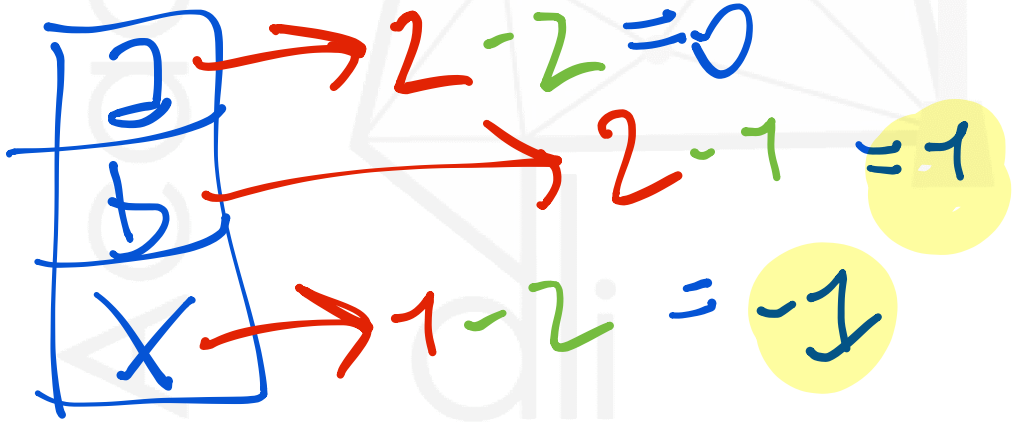
COUNT STUFF!

Check if the two strings have identical character counts.

return sorted(s1) == sorted(s1)

2b2bx

x22b1



t O(m+n)

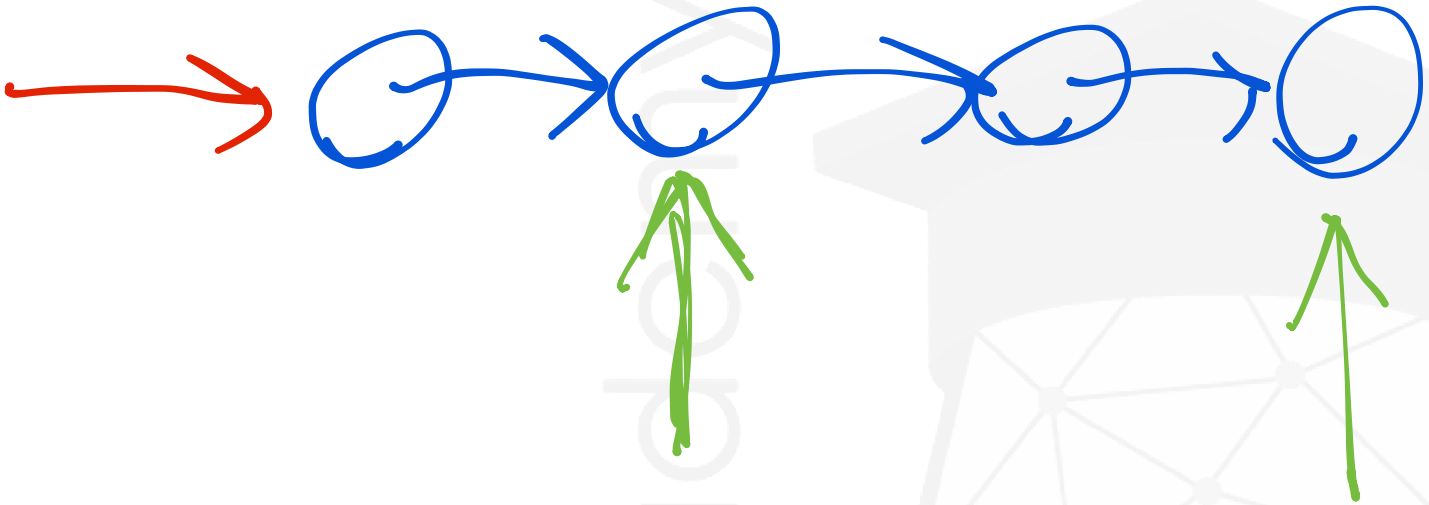
≤ O(m+n)

ALFABETO NOTO

⇒ S O(1)

```
def same_char_count(s1,s2):
    char_count = {}
    for c in s1:
        char_count[c] = char_count.get(c,0)+1
    for c in s2:
        char_count[c] = char_count.get(c,0)-1
    return set(char_count.values())=={0}
```

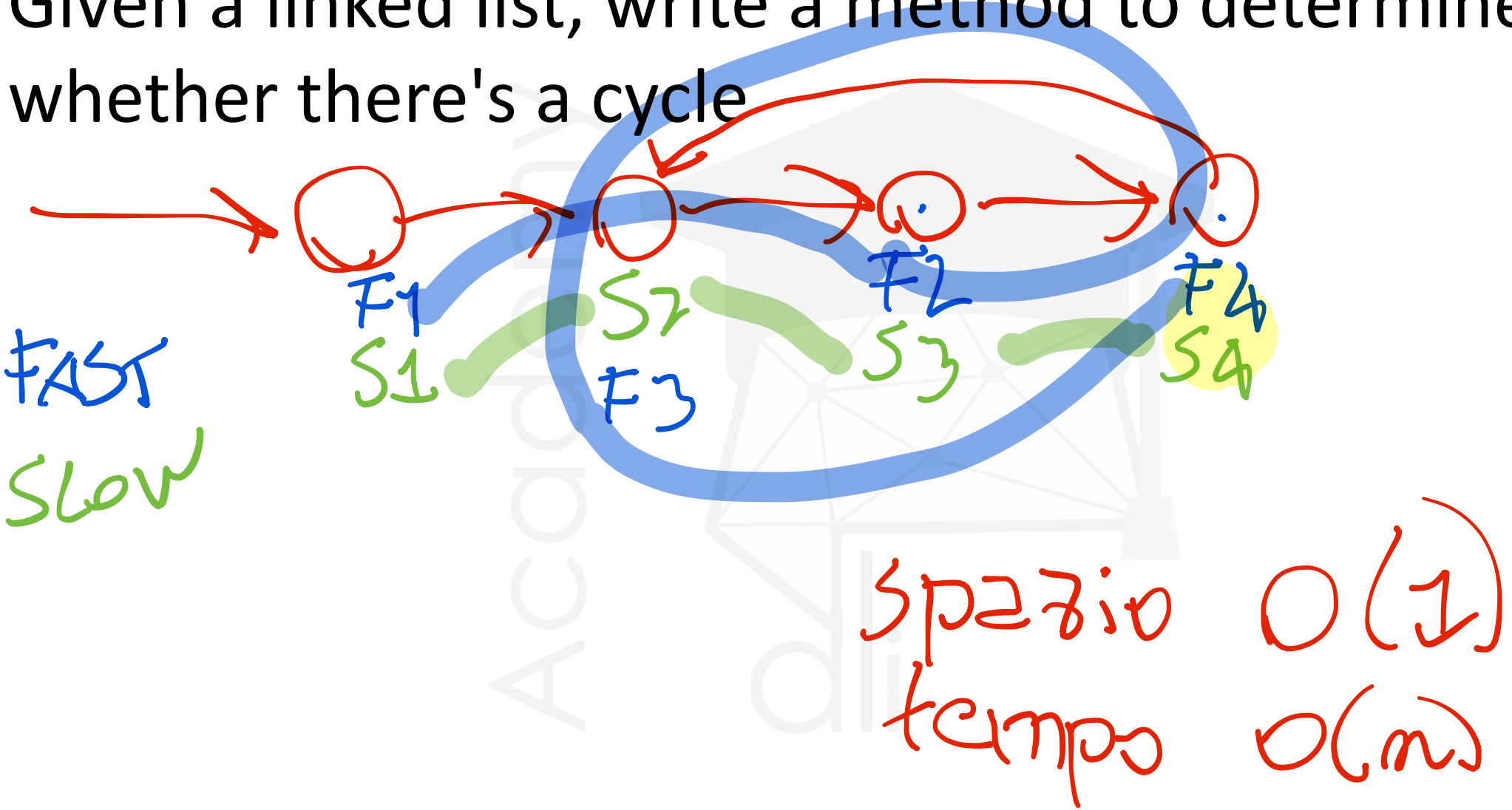
04 _ Linked Lists



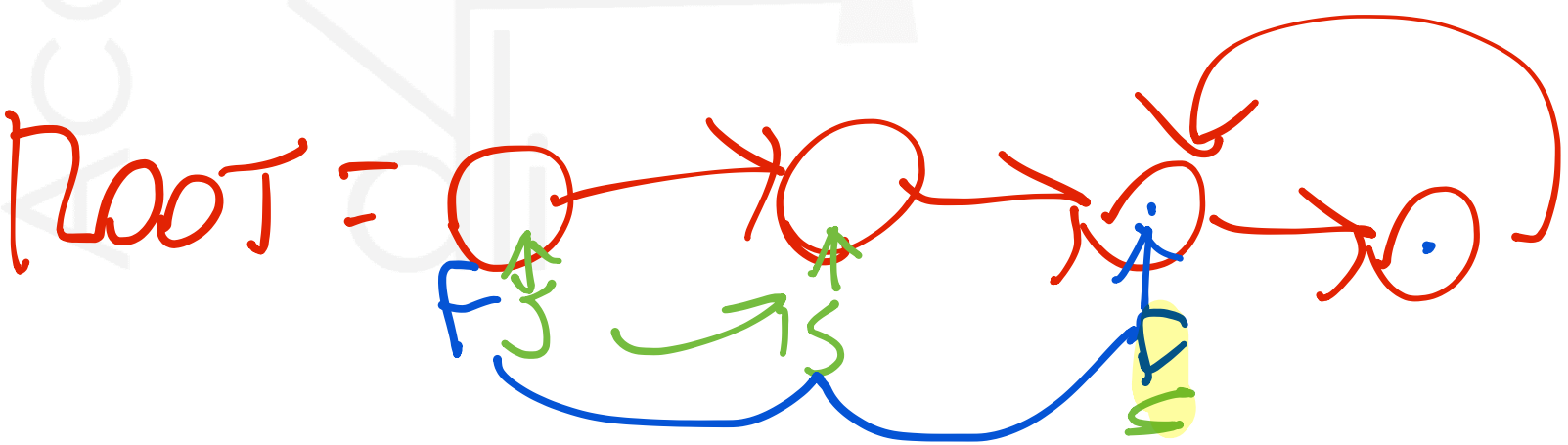
DEAD

O(n)

Given a linked list, write a method to determine whether there's a cycle

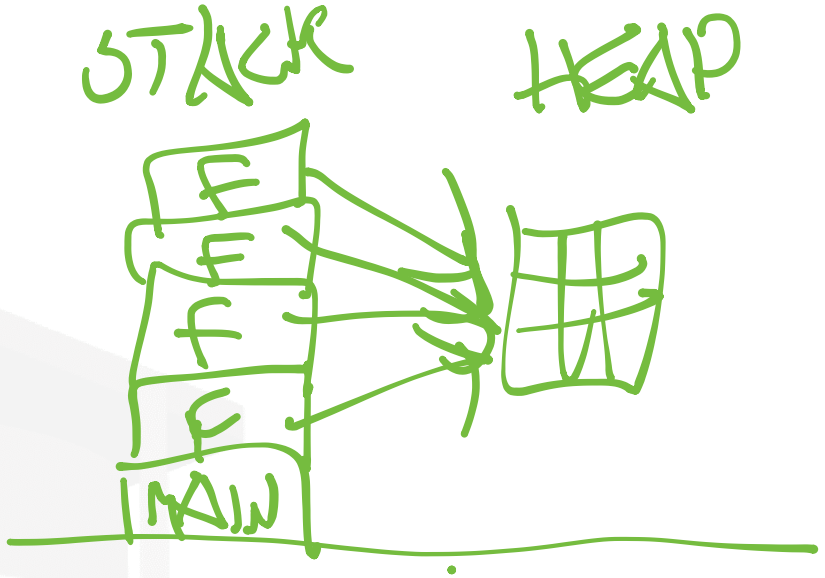


```
def check_for_loop(root):
    slow,fast=root,root
    while True:
        if fast.next is None or fast.next.next is None:
            return False
        slow = slow.next
        fast = fast.next.next
        if slow==fast:
            return True
```



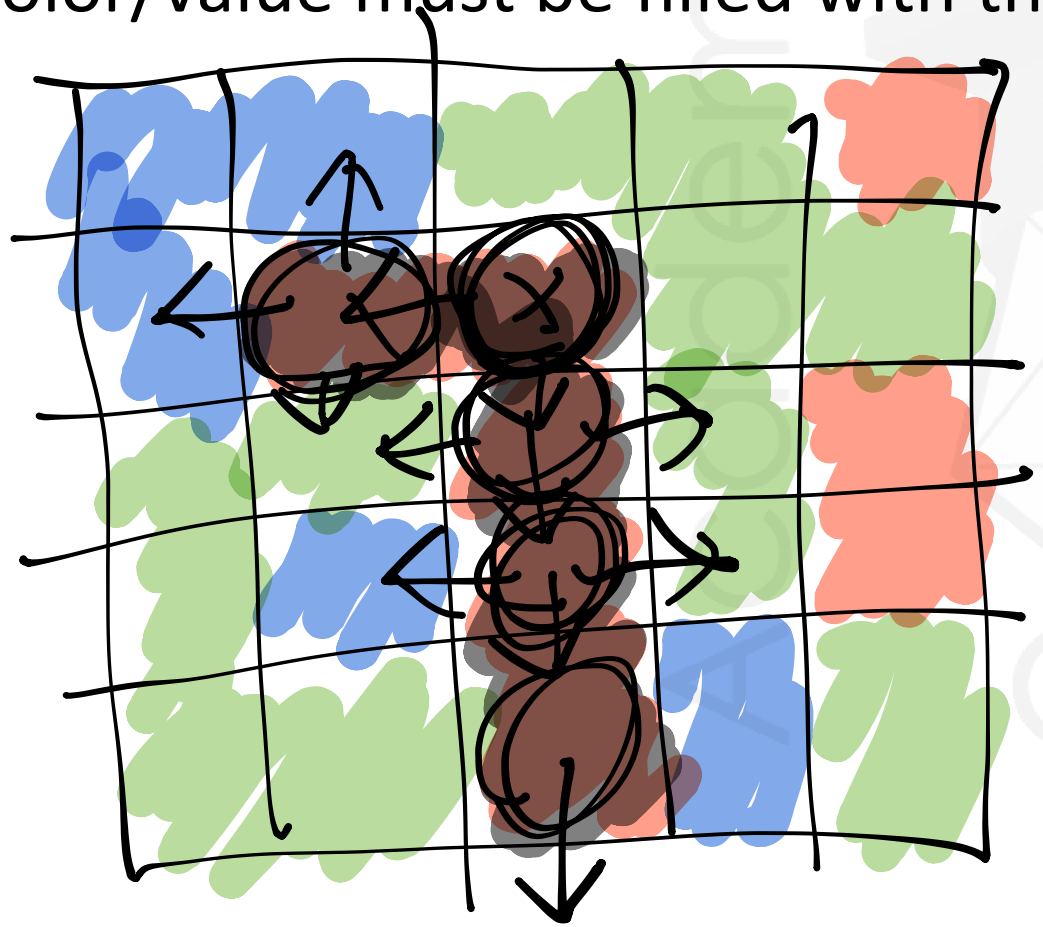
05 _ Recursion

beautiful!



EXPLOSIVE COMPLEXITY!

Write a method that implements the 'paint fill' in a matrix representing an image. That is, given a matrix, a new color value and a couple of coordinates (x,y), matrix[x,y] and all adjacent cells with the same color/value must be filled with the new color value.



→ DON'T FALL OUT!

→ REMEMBER WHAT COLOR TO REPLACE!

LIST OF LISTS OR NUMPY?

```
def paint_fill(matrix,r,c,new_col,prev_col):
    matrix[r,c] = new_col
    if r>0 and matrix[r-1,c] == prev_col:
        paint_fill(matrix,r-1,c,new_col,prev_col)
    if r<matrix.shape[0]-1 and matrix[r+1,c] == prev_col:
        paint_fill(matrix,r+1,c,new_col,prev_col)
    if c>0 and matrix[r,c-1] == prev_col:
        paint_fill(matrix,r,c-1,new_col,prev_col)
    if c<matrix.shape[1]-1 and matrix[r,c+1] == prev_col:
        paint_fill(matrix,r,c+1,new_col,prev_col)
```

GO UP

GO DOWN

GO LEFT

GO RIGHT

CAN I GO THERE?

$t: O(m \times n)$

$S: O(1)$

06 _ Fibonacci

fib(9)? 34

INDEXES	0	1	2	3	4	5	6	7	8	9
VALUES	0	1	1	2	3	5	8	13	21	34

$F_9 = F_8 + F_7$
$F_8 = F_7 + F_6$
F_7
⋮
$F_1 = 1$
$F_0 = 0$

RECURSIVE

fib(n) = fib(n-1) + fib(n-2)

ITERATIVE

```
def fibonacci_ricorsivo(n):
    if n<2:
        return n
    else:
        return fibonacci_ricorsivo(n-2)+fibonacci_ricorsivo(n-1)
```

fib(0) = 0
fib(1) = 1

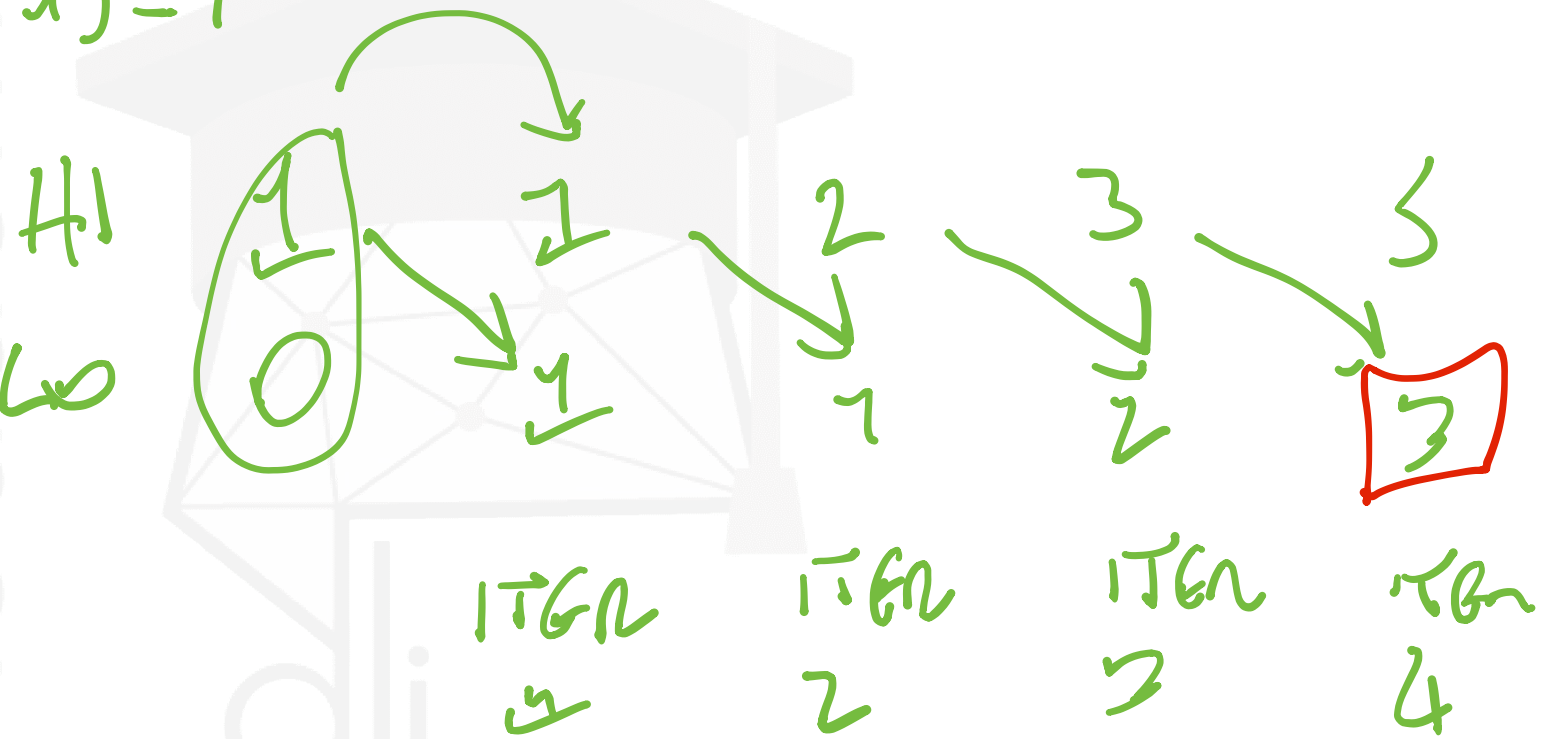
fib(4) = fib(2) + fib(3)
= fib(0) + fib(1) + fib(1) + fib(2)
0 + 1 + 1 + fib(0) + fib(1)
0 + 1 + 1 + 0 + 1 → 3

```
def fibonacci_iterativo(n):
    if n<2:
        return n
    else:
        lo,hi = 0,1
        for _ in range(n):
            lo,hi = hi,lo+hi
        return lo
```

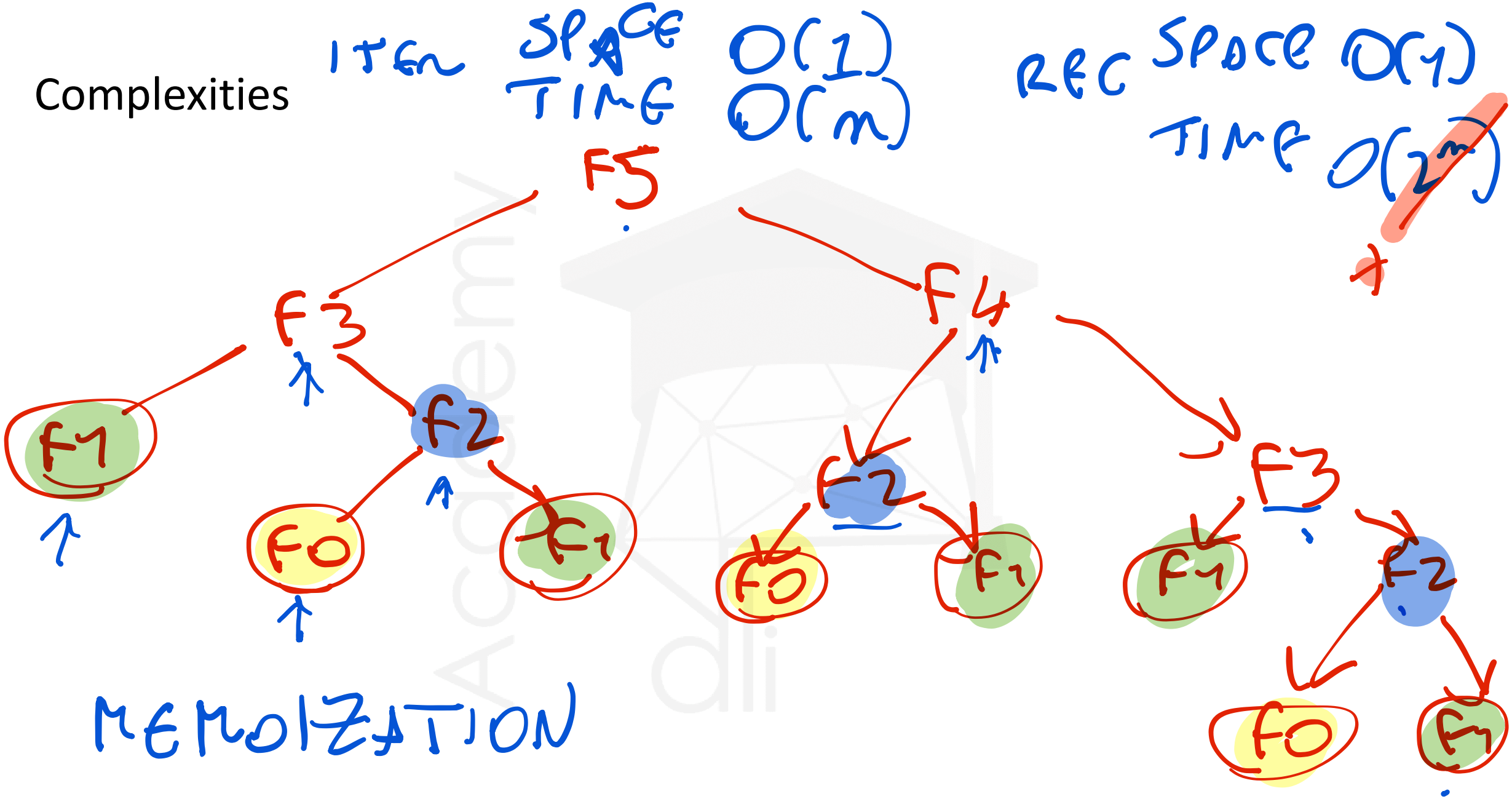
fib(4)? 3

fib(0) = 0

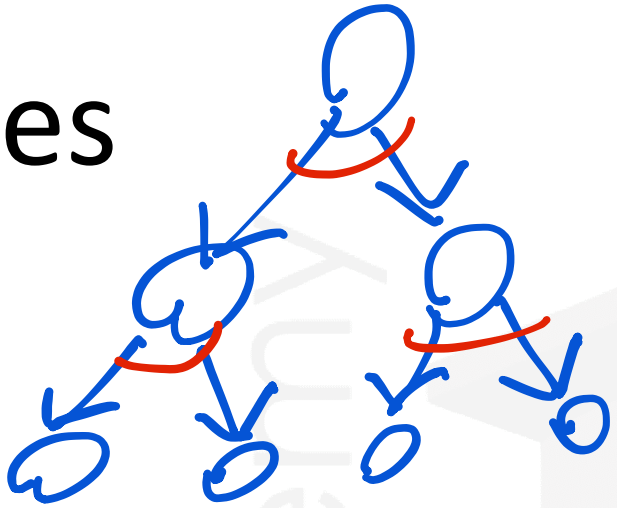
fib(1) = 1



Complexities



07 _ Trees



LEFT
RIGHT

BINARY SEARCH TREE

ALWAYS ASK!

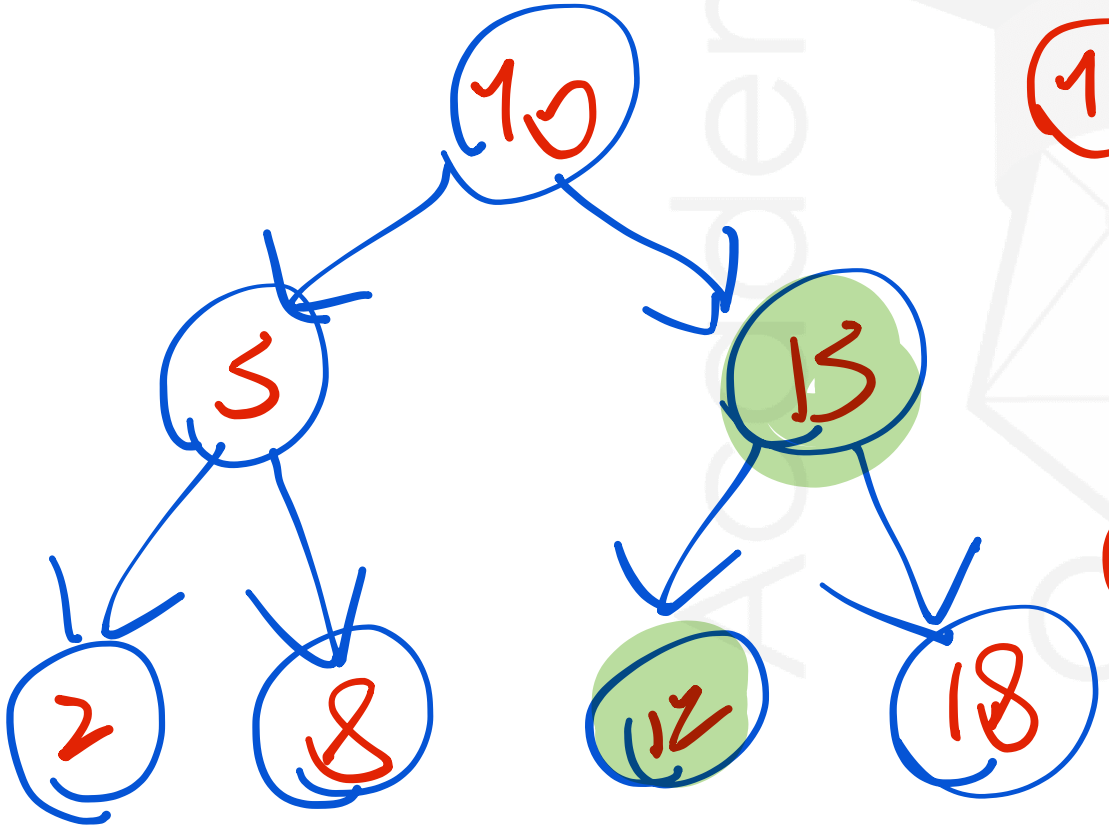
LOOK FOR BOTH

ITERATIVE AND RECURSIVE SOLUTIONS

IF BST SEARCH $O(\log n)$

Write a function that returns the lowest common ancestor of two nodes in a binary search tree.

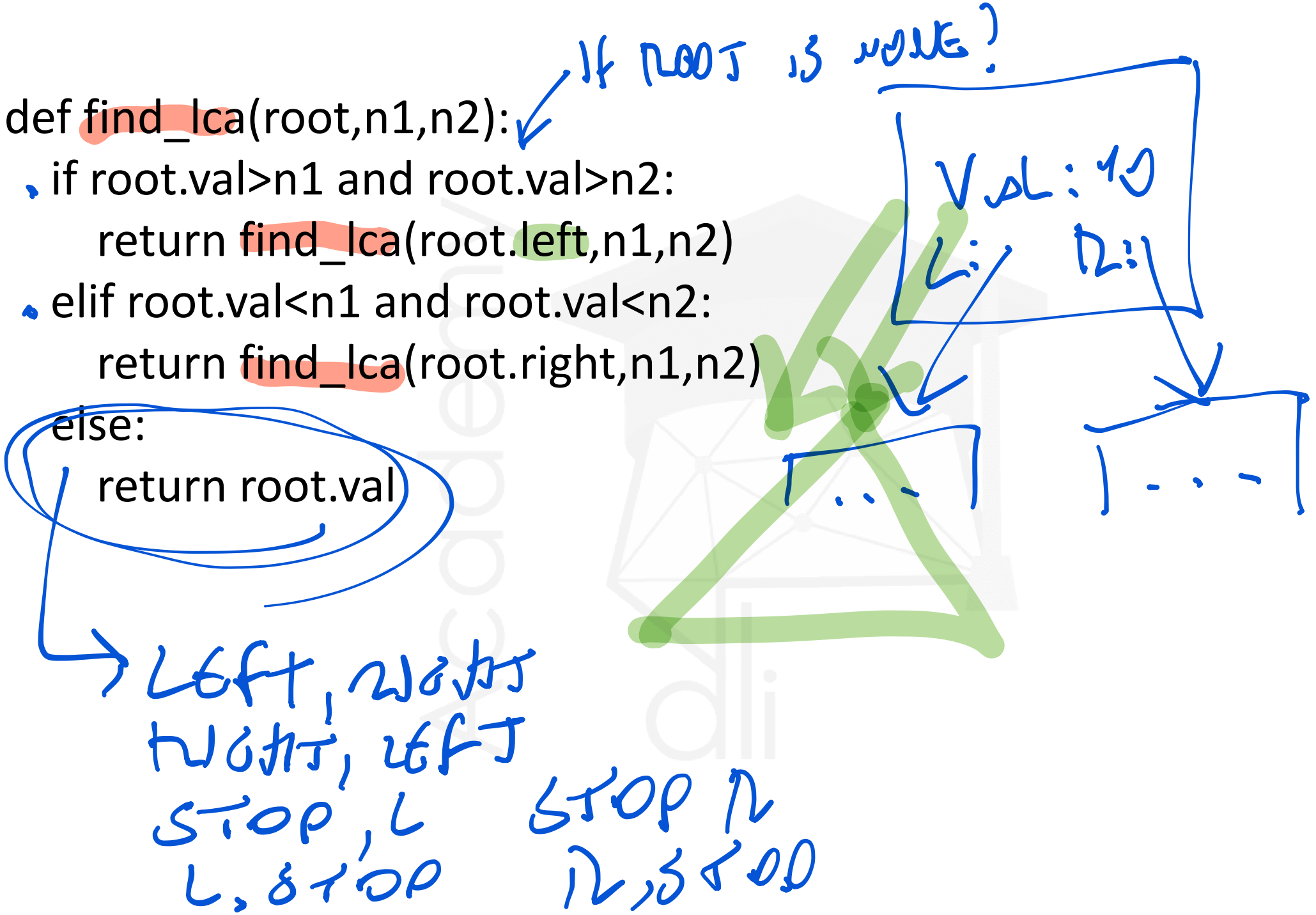
Assume both nodes are in the tree



① BOTH PATHS
10, 15
10, 15, 12
$\leq O(\log n)$ $\in O(\log n)$

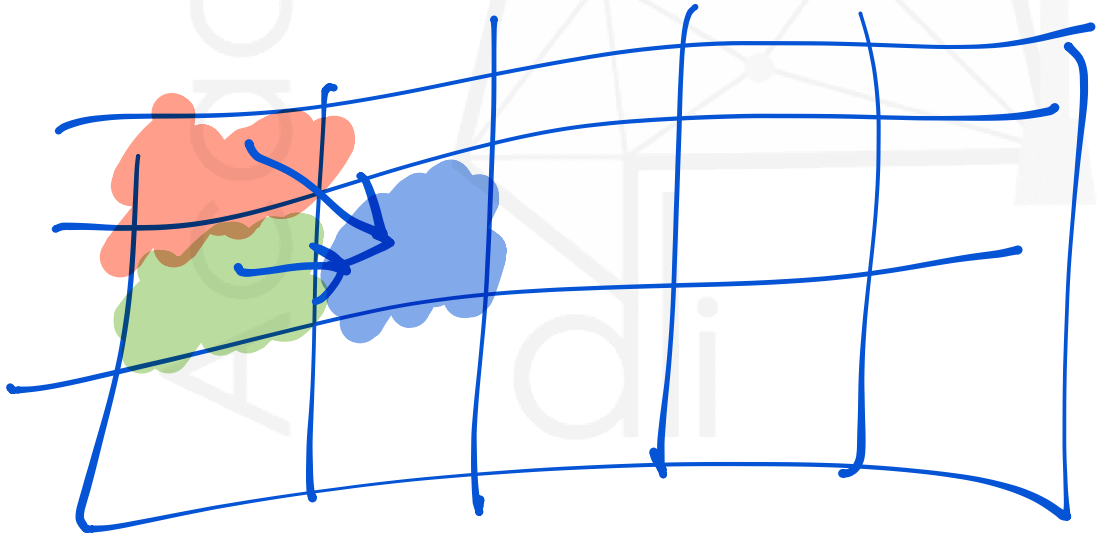
② SEARCH FOR BOTH
IN 10 → RIGHT, RIGHT
IN 15 → LEFT, STOP
$\leq O(2)$ $\in O(\log n)$

```
def find_lca(root,n1,n2):
    if root.val>n1 and root.val>n2:
        return find_lca(root.left,n1,n2)
    elif root.val<n1 and root.val<n2:
        return find_lca(root.right,n1,n2)
    else:
        return root.val
```



08 _ Dynamic Programming

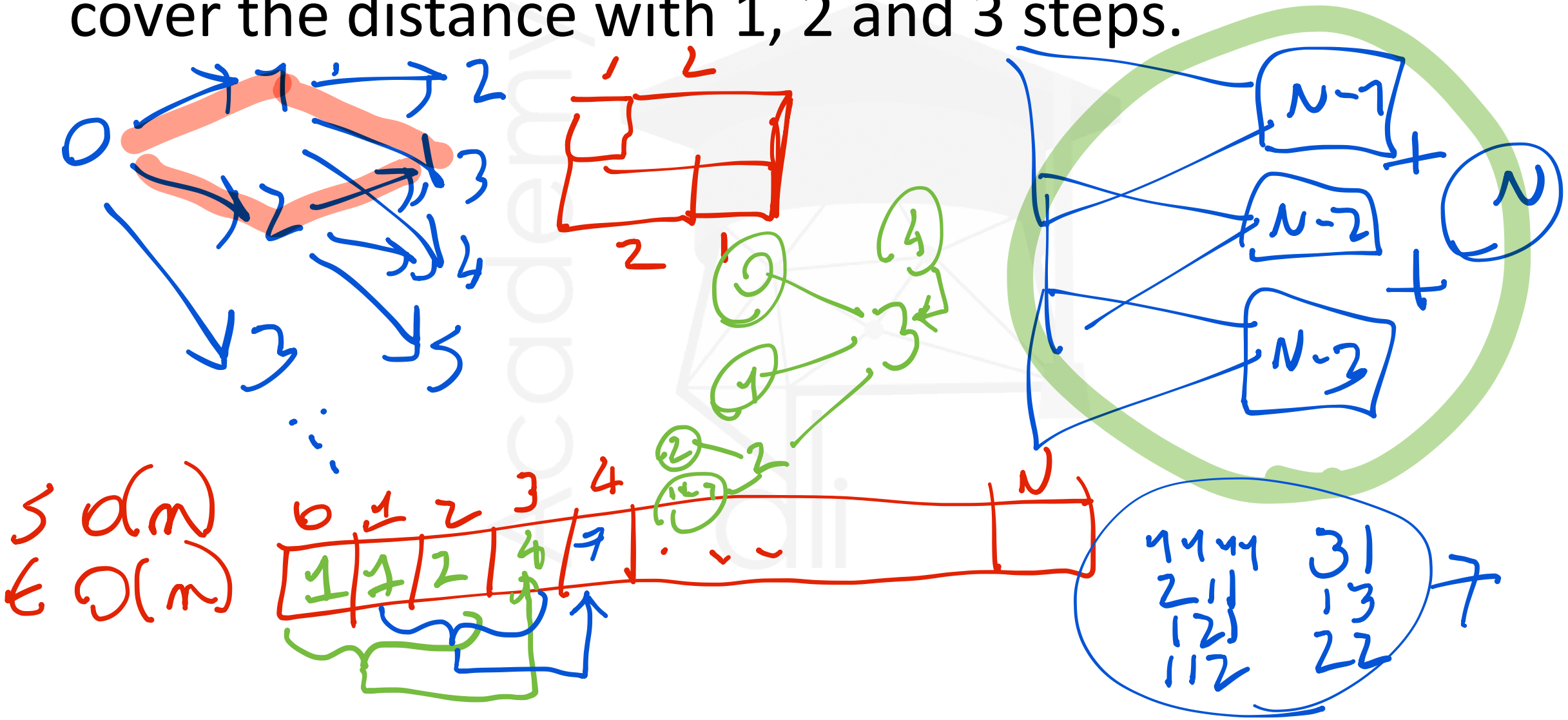
DECOMPOSE IN SUBPROBLEMS
APPLY ALGORITHM
RE-COMPOSE SOLUTION



TABLE

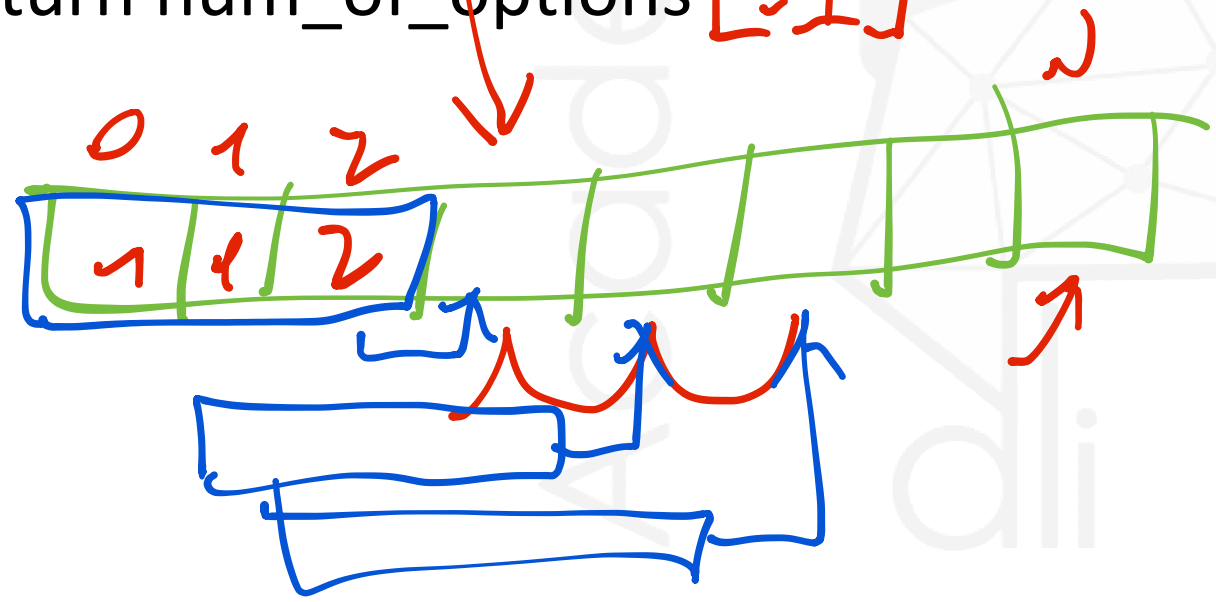
COMPUTATIONAL COMPLEXITY OFTEN POLINOMIAL

Given a distance, count the total number of ways to cover the distance with 1, 2 and 3 steps.



```
def cover_distance(n):
    num_of_options = np.zeros(n+1,dtype=int)
    num_of_options[:3] = [1,1,2]
    for itr in range(3,n+1):
        num_of_options[itr] = sum(num_of_options[itr-3:itr])
    return num_of_options[-1]
```

O(n)



09 _ More ADTs and quiz types

- Stacks and Queues
- Sorting and searching
- Bloom Filters
- Heaps, Union-Find, Red-Black trees
- Tries
- Graphs
- Bit Manipulation
- Probability
- Intuition/Logic

10 _ Testing!

- Discuss correctness (sketch proof?)
- List input assumptions
- Think about corner cases
- Analyse computational complexity
- Sketch test cases