Divide & Conquer

10.020 Data Driven World

Natalie Agus

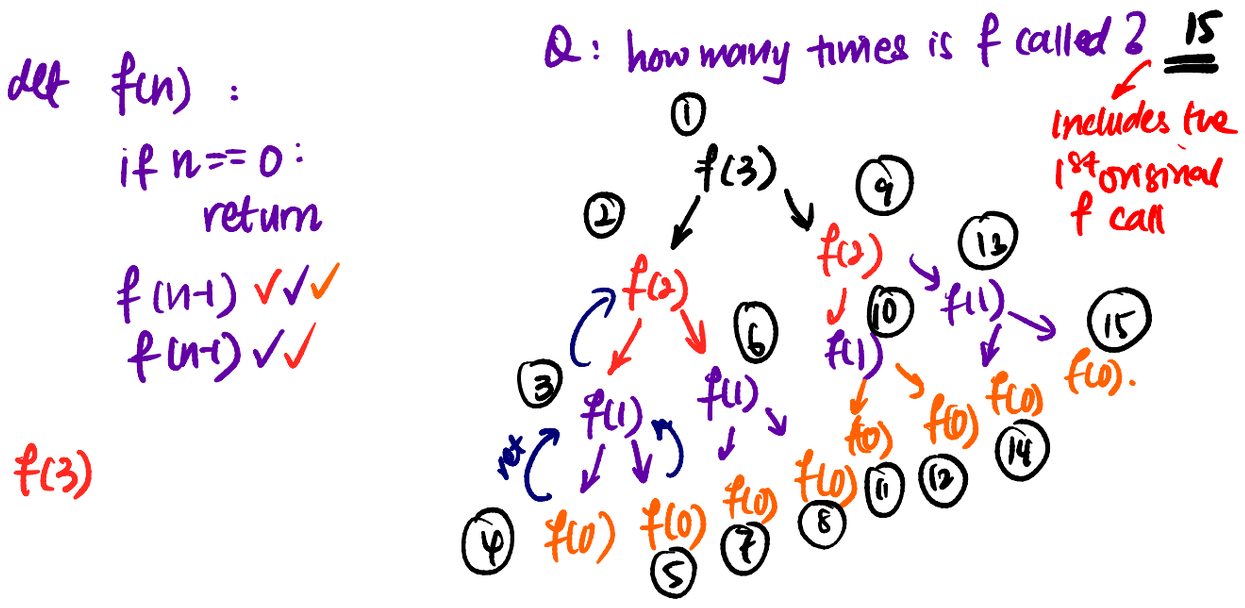
Learning Objectives

- **Recursion**:
  - Solve problems using recursion.
  - Identify problems that has recursive solutions.
  - Identify **base case** and **recursive case** in a recursive problem and its solution.
  - Explain and implement the recursive solution of Tower of Hanoi.
  - Derive solution for **recurrence of Tower of Hanoi** using recursion-tree method
- **Merge-Sort**:
  - Explain and **implement** merge sort algorithm.
  - Derive solution of recurrence of merge sort using **recursion-tree method**.
  - Measure computation time of merge sort and compare it with the other sort algorithms.

Recursion

Definition

- It is a function that **calls itself** to solve **smaller problems**
- Requires a **base case** to stop recursion and prevent infinite loops, then it has **recursive case to solve repetitive substructure**
- Often used for problems that have **repetitive substructure** (e.g., factorial, Fibonacci, tree traversal, Tower of Hanoi)



Identify the base case and recursive case

```
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

eg: 5

2 6

2, 3, 4, 5

res = 1 * 2 = 2

res = 2 * 3 = 6

res = 6 * 4 = 24

res = 24 * 5 = 120

5! → 5 x 4 x 3 x 2 x 1

substructure: (n * n-1)

```
def fact_r(n):
    if n <= 1:
        return n
    else:
        return n * fact_r(n-1)
```

BASE CASE

recursive case

1, 2, 3

3 * 2

2 * 1

out = fact_r(1) → 1

out = fact_r(2) → 2*1

fact_r(2) calls fact_r(1) returns 1

out = fact_r(3) → 6

fact_r(3) → 3*2

fact_r(2) → 2*1=2

fact_r(1) returns 1

fact_r(1) → 1

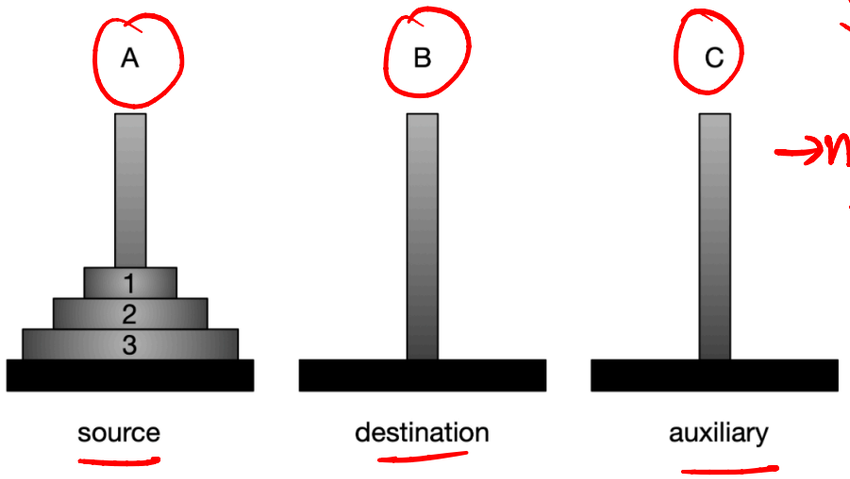
fact_r(2) → 1*2

fact_r(3) → 2*3

Identify the base case and recursive case

```
1 def sum(array):
2     result = 0
3     for number in array:
4         result += number
5     return result
6
7 input_array = [4, 3, 2, 1, 7]
8 print(sum(input_array))
```

Identify the base case and recursive case



when n is 1 :
→ move that only disk to dest tower.

when n > 1

① move all n-1 disks to aux tower
② move biggest (nth) disk to dest tower
③ move all n-1 disks in aux tower to destination tower

Show Pseudocode

```
Input:
  - n, number of disks
  - source tower
  - destination tower
  - auxiliary tower
Output:
  - sequence of steps to move n disks from source to destination tower using auxiliary to
Steps:
1. if n is 1 disk:
    1.1 Move the one disk from source to destination tower
2. otherwise, if n is greater than 1:
    2.1 Move the first n - 1 disks from source to auxiliary tower
    2.2 Move the last disk n from source to destination tower
    2.3 Move the first n - 1 disks from the auxiliary tower to the destination tower
```

steps to code: (can rename variable around to make sense.)

① start w/ a base case test case:
move_disk(1, A, C, B)
f t aux.
② (generate) check answer, implement
③ increase the test case complexity, eg: 2 disks.
④ generate answer key & implement
⑤ check w/ other test cases (normal). n=3,4,5
⑥ check w/ weird test cases: n=0, n<1

Complexity

Time Complexity (there's also space complexity)

O(n)

O(n).

at one point, we have n-1 fact functions waiting for fact(n) to return → each function needs O(1) of space to hold its current state:
∴ O(n)

guess: n log(n)

Factorial?

Simplify & draw the recurrence tree

```
def fact(n):
    if n==1 or n==0: → O(1)      (constant time to run, doesn't depend on size of n.)
        return 1 → O(1)
    return n * fact(n-1)
           O(1)
```

O(1) fact(n)
↓ calls
O(1) fact(n-1)
↓
O(1) fact(n-2)
⋮
O(1) fact(1)

n calls of fact, each costs O(1)
∴ O(n)

assume n>1

```
def fact_iterative(n):
    result = 1   O(1)
    for i in range(2, n+1):   } O(n)
        O(1) result *= i
    return result → O(1)
```

repeat n times

time complexity: O(n).
space complexity: O(1).
* look at var declaration

Complexity

Time Complexity (there's also space complexity)

Sum?

Simplify & draw the recurrence tree

Complexity

Time Complexity (there's also space complexity)

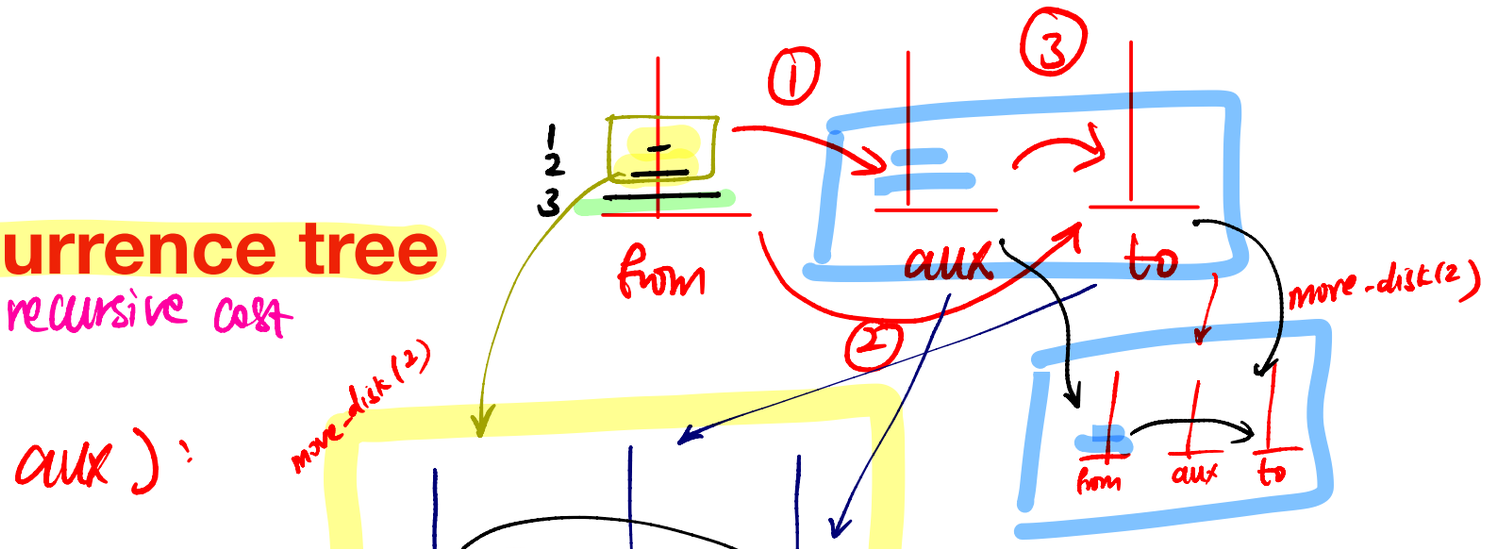
every func call is gonna to append a string (or two) to the result list.
# of strings = # of func calls, which is also $O(2^n)$

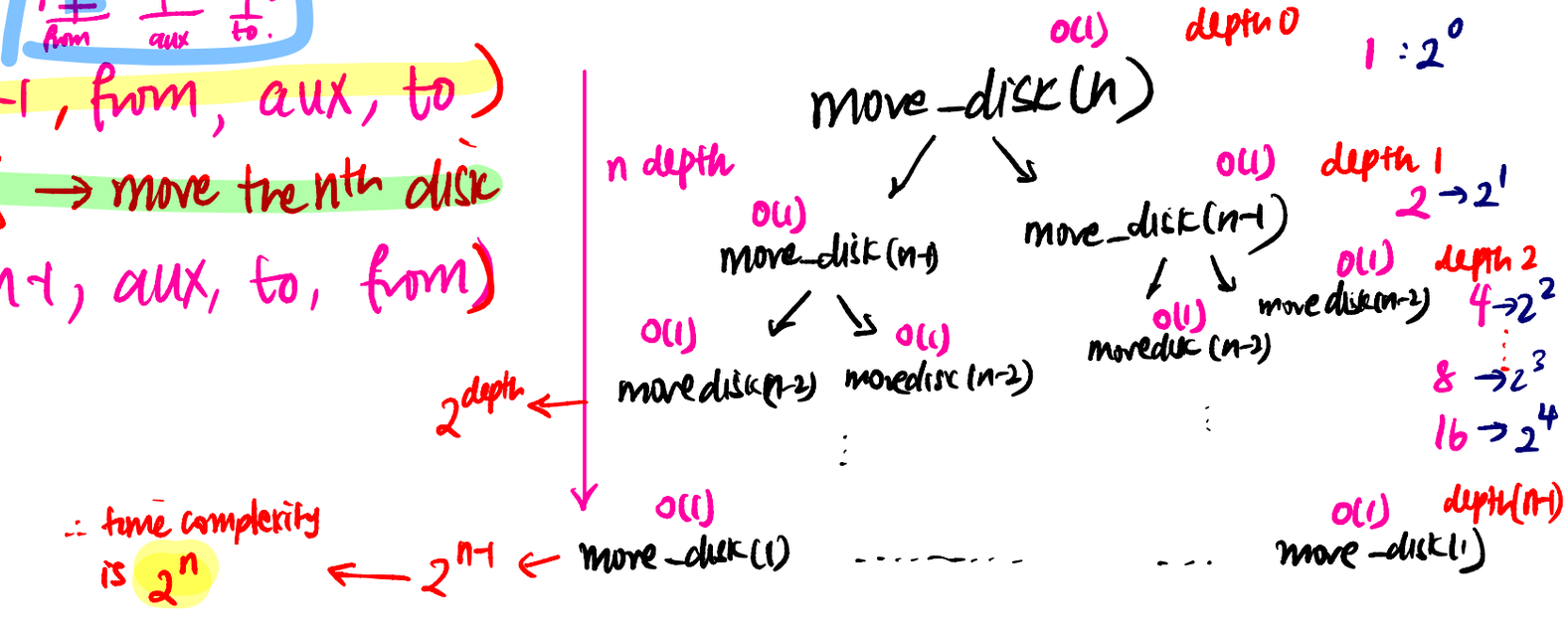
Tower of Hanoi?

Simplify & draw the recurrence tree

constant cost of time, minus recursive cost
constant ≠ CHEAP

```
def move_disks(n, from, to, aux):
O(1)    result = []
O(1)    if n == 1:
O(1)        return [f"{1} → {to}"]
        else:
O(1)    ① result = move_disk(n-1, from, aux, to)
O(1)    ② result += [f"{n} → {to}"]
O(1)    ③ result += move_disk(n-1, aux, to, from)
```





Merge Sort

Others: Insertion sort $O(n^2)$, Bubble sort $O(n^2)$, Heapsort $O(n \log n)$ } Tc

A sorting algorithm that implements the principle of divide and conquer

merge_sort() → mc

n = 10 → 4x

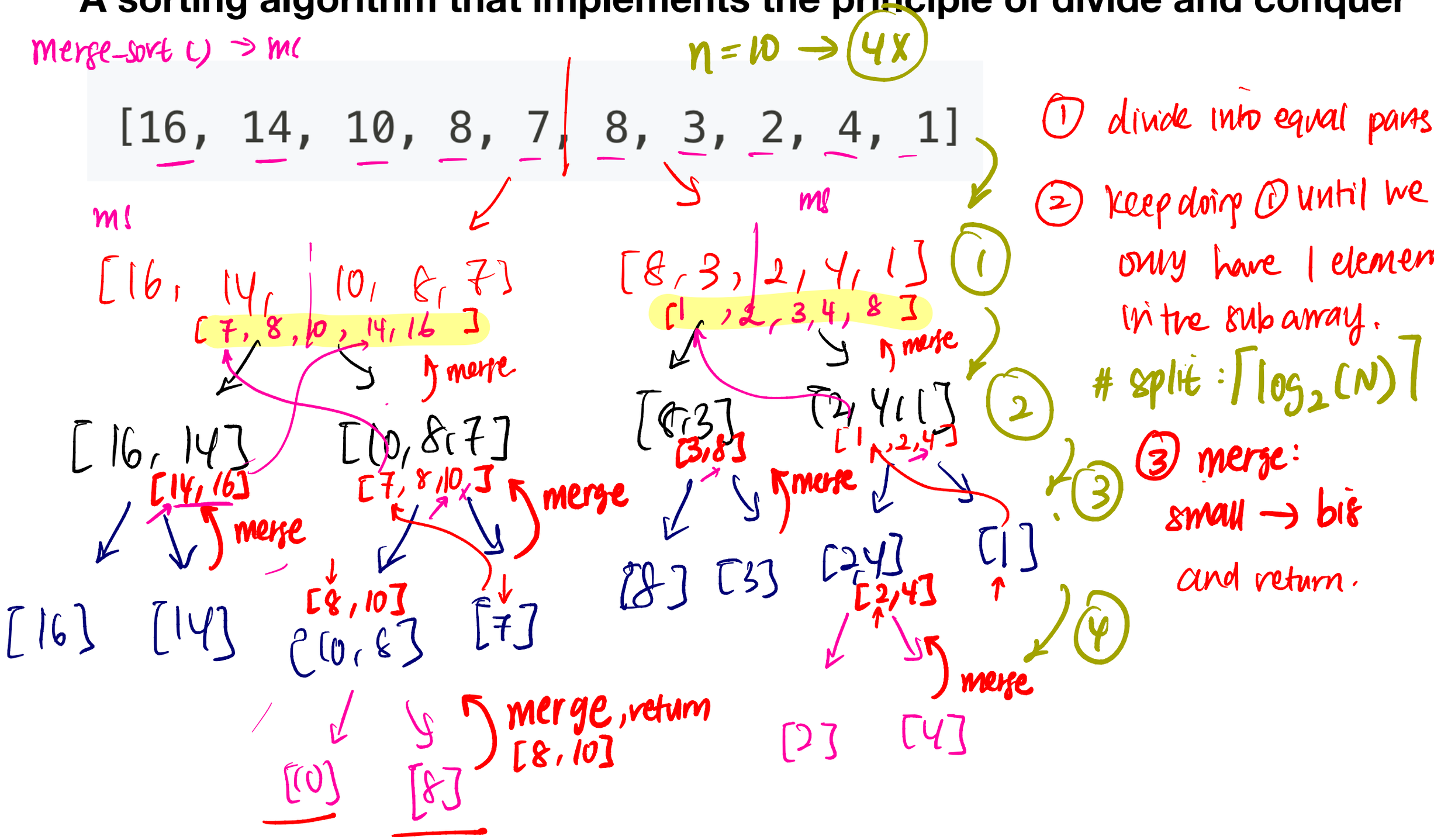
[16, 14, 10, 8, 7, 8, 3, 2, 4, 1]

① divide into equal parts

② keep doing ① until we only have 1 element in the sub array.

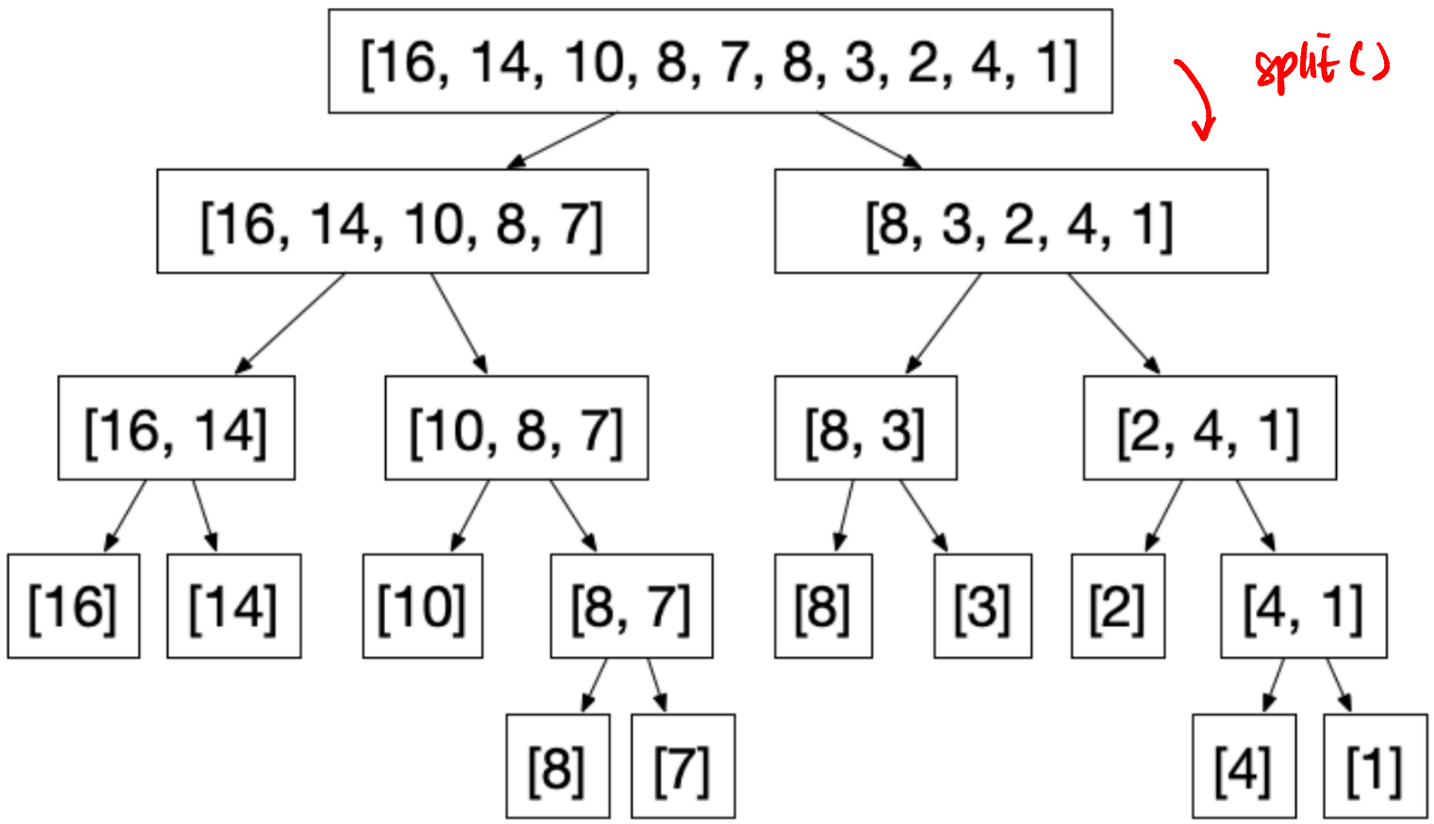
split : $\lceil \log_2(N) \rceil$

③ merge: small → big and return.



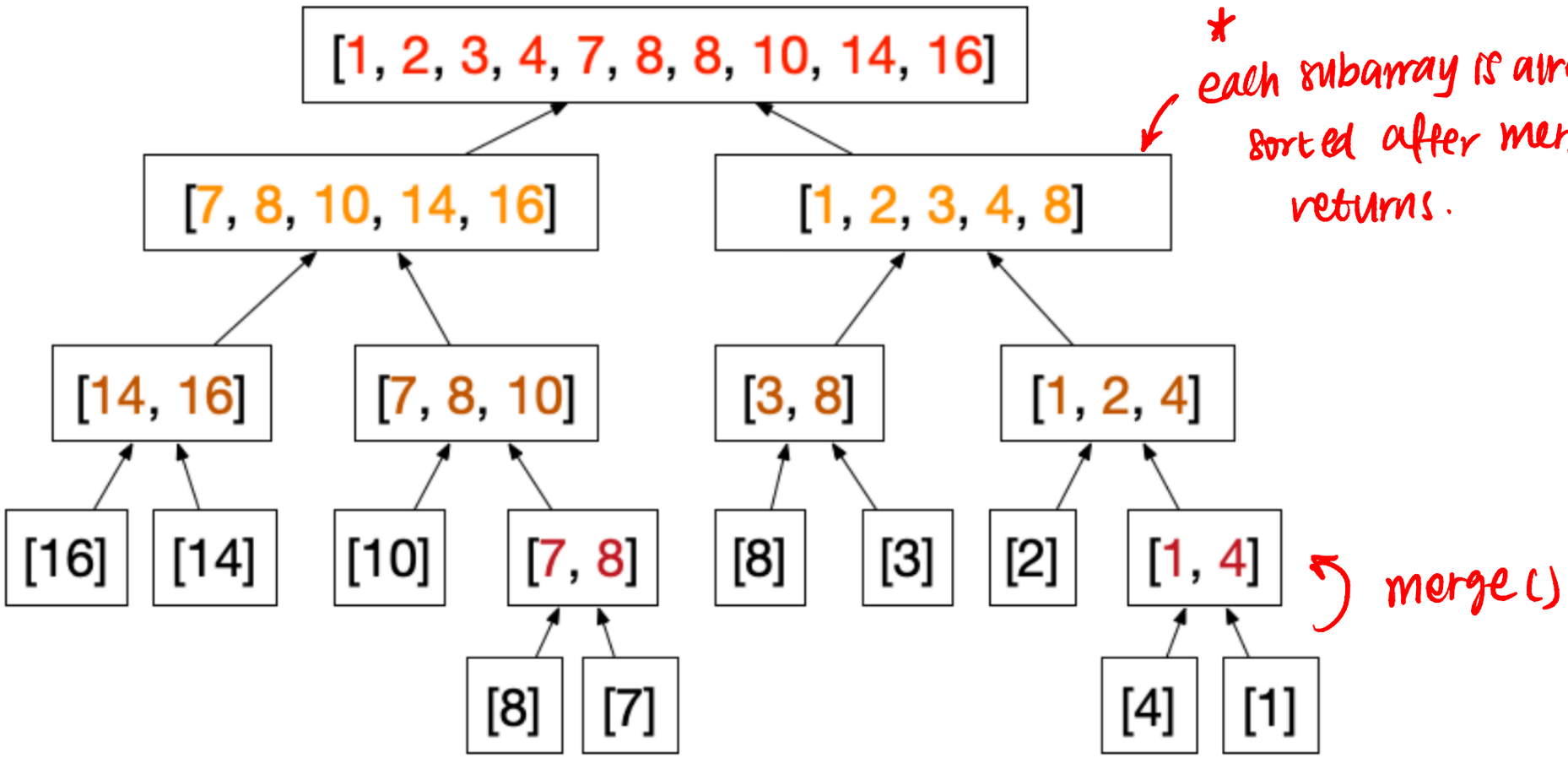
Step 1: Split

Stop until you just have 1 element



Step 2: Merge

Compare and merge two arrays

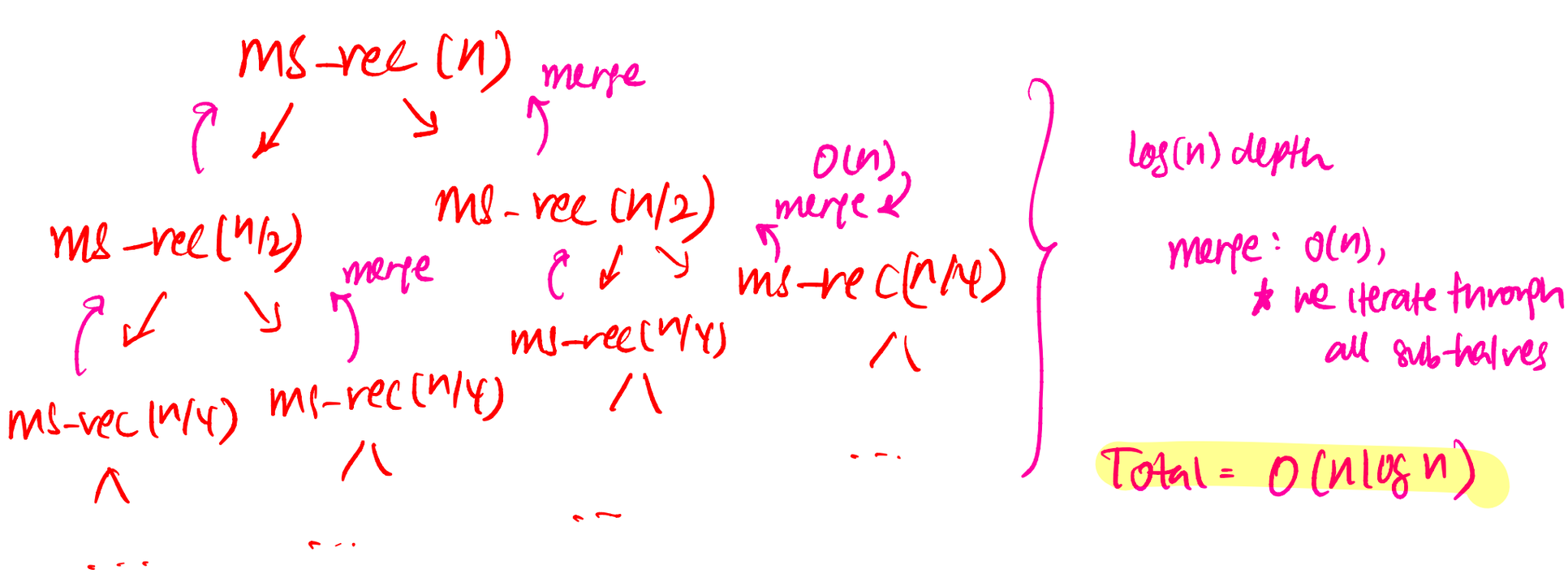


Complexity

Time complexity of merge-sort algorthm

↳ O(n log n)

Simplify & draw the recurrence tree



Complexity Summary Table

Algorithm	Time Complexity	Space Complexity
Sum (recursive)	O(n)	O(n) → because of func call stack
Sum (iterative)	O(n)	O(1)
Factorial (recursive)	O(n)	O(n)
Factorial (iterative)	O(n)	O(1)
Tower of Hanoi	$O(2^n)$	~~O(n)~~ $O(2^n)$ because we append string to the list, # strings == # func call
Merge Sort	O(n log n)	O(n)

also + O(log n): recursion depth. to hold funcstack, but log n < n, so we just report O(n)

standard mergesort, we create temp left right array at each merge, so we use O(n) space at a time in worst case scenario.

Why Recursion?

We can use iterative method + some helper data structure

- However, recursion is **superior** when the problem has a **self-similar structure** **(can divide & conquer)**: trees, nested data

- Recursive calls mirror the problem shape ***naturally***
  - Back-tracking and divide and conquer tasks
  - Iterative solutions require manual stacks and control flow

- Recursive functions are **modular** and **composable**, making them easier to **read**, **test**, and **reuse** than loop-based or stack-simulated alternatives

Summary

- **Recursion**:
  - Solve problems using recursion.
  - Identify problems that has recursive solutions.
  - Identify **base case** and **recursive case** in a recursive problem and its solution.
  - Explain and implement the recursive solution of Tower of Hanoi.
  - Derive solution for **recurrence of Tower of Hanoi** using recursion-tree method
- **Merge-Sort**:
  - Explain and **implement** merge sort algorithm.
  - Derive solution of recurrence of merge sort using **recursion-tree method**.
  - Measure computation time of merge sort and compare it with the other sort algorithms.