Data Driven World
Week 3

Simon Perrault– Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

Week 3 – Divide and Conquer

“In order to understand recursion, one must first understand recursion.”

Divide and Conquer

Take a problem too large to solve, break it down into **smaller problems**

Example 1: Sum of Elts in Array (Human Version)

```
l1 = [1, 3, 53, 65, 75, 68, 99, 45, 39, 23, 53,
29, 72, 94, 32]
```

Can anyone sum that ***quickly***?

A: No.

Example 1: Sum of Elts in Array (Human Version)

```
l2 = [1, 3, 53, 65]
```

Can anyone sum that ***quickly***?

A: Not really

Example 1: Sum of Elts in Array (Human Version)

```
l3 = [1, 3]
```

Can anyone sum that ***quickly***?

A: Yes but still need to think

Example 1: Sum of Elts in Array (Human Version)

```
l4 = [1]
```

Can anyone sum that ***quickly***?

A: Yes

Also consider:

```
l5 = []
```

Recursion Philosophy

- Consider the **easiest** cases of the problem one can solve
  - …and only the easiest cases
- Consider the general case of the problem
  - …and find a way to make it simpler, **one step at a time**

If you can do both, the problem will eventually get simple enough and be solved

Finding an Easiest Case [Base Case]

1. An input for which the expected value is known
    - List with one element
    - For mathematical functions, a remarkable number
2. An empty input
    - Empty list

Making the Problem Simpler [General Case]

- Reducing the size of the data:
  - Removing one element
  - Dividing the data into smaller chunks (2, 3, 4 or more)

Summing Elements of an Array

- Easiest case:
  1. The sum of an array with one element is the value of the element
  2. The sum of an empty array is 0

Specific cases (we know the expected results)

- Making the problem simpler:
  1. The **sum of an array** equals to the value of its first element, and the **sum of the subsequent array** starting at index 1

General case, the data size will reduce by 1 every call

Sum of Array Algorithm

Input: Array or list of numbers

Output: the Sum of the array

Steps:

1. if the number of element is one only
   1.1 Return that element as the sum of the array
2. Otherwise,
   2.1 Return the addition of the first element with the sum of the rest of the array

How to code runs

```
array = [4, 3, 2, 1, 7]

sum(array) = 4 + sum([3, 2, 1, 7])              unknown
      sum([3, 2, 1, 7])) = 3 + sum([2, 1, 7])   unknown
            sum([2, 1, 7]) = 2 + sum([1, 7])    unknown
                  sum([1, 7]) = 1 + sum([7])    unknown
```

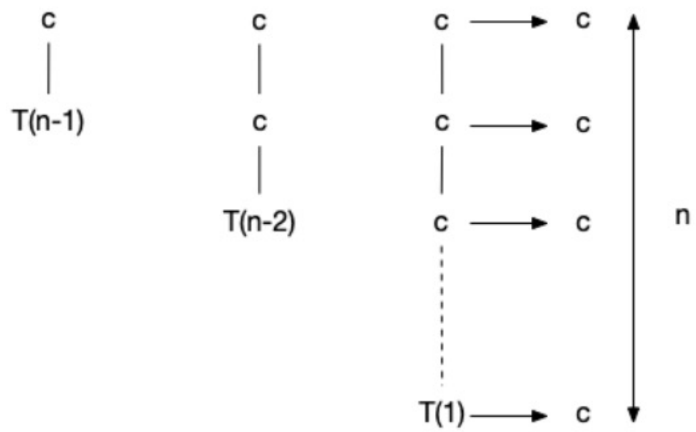
`sum([7]) = 7`

How to code runs (step 2)

```
array = [4, 3, 2, 1, 7]
sum(array) = 17
sum(array) = 4 + 13
      sum([3, 2, 1, 7])) = 3 + 10
            sum([2, 1, 7]) = 2 + 8
                  sum([1, 7]) = 1 + 7
                        sum([7]) = 7
```

Computation Time

- Each step is usually straightforward (O(1), constant time)
- **So if you follow the above pattern, complexity is O(n)**



Factorial

$$n! = n \times \underbrace{(n-1) \times (n-2) \times \ \ldots \times 2 \times 1}_{(n-1)!}$$

Therefore, $n! = n \times (n-1)!$

Easiest cases: By definition, $1! = 1$ and $0! = 1$

Factorial

```
Input: n, an integer
Output: factorial of n, an integer
Steps:
1. if n is equal to 0 or to 1
    1.1 return 1
2. otherwise,
    2.1 return n * factorial of n-1
```

Palindrome (Recursive)

- Is a given word a palindrome?

Example: “`datadrivenworld`”

- Base cases:
  - An empty string/word is a palindrome
  - A string with one element is a palindrome
- General Case: Check first and last letter in the word
  - If they are the same, this might be a palindrome
  - If they are different, the word is not a palindrome

Palindrome General Case

“If they are the same, this might be a palindrome”

Meaning:

If both first and last character are the same, this word is a palindrome if the subword (i.e. the remaining characters) are palindrome

By removing two characters each time, we will gradually reach an base case.

Palindrome in Cohort Problem

```
def palindrome(s):
    pass
```

This function works if we make sure to call it recursively and “trim” the string ourselves

Palindrome (Elegant)

- Instead of trimming, we can also keep track of the indices/indexes in a helper function:

```
def is_palindrome(s, left, right):
    pass
```

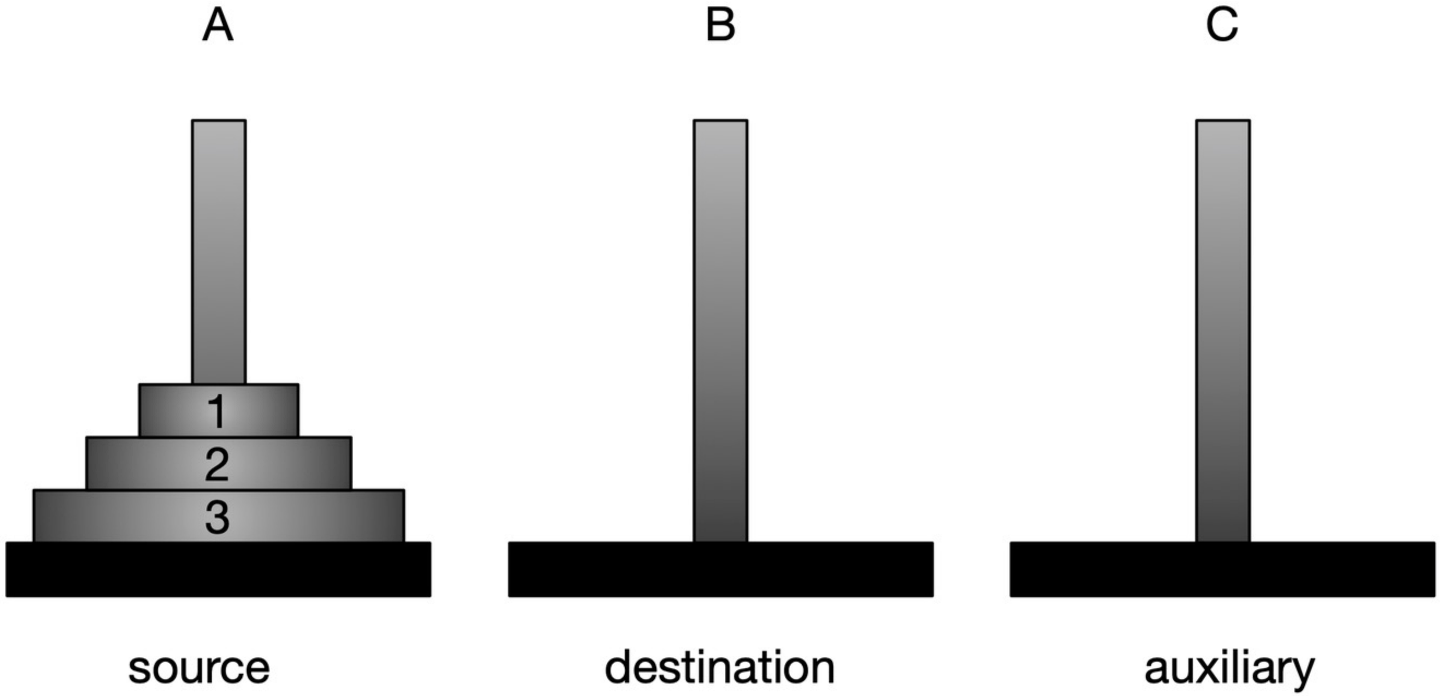
- Here, ”left” and “right” are used to store the index of resp. the first and last character for this call
- **Note: depending on the number of characters (odd or even), we will reach one of the two base cases.**

Using Input to Store Information

- We use this technique a lot when doing recursion
- In some cases, the recursion approach may not work without it

Tower of Hanoi

- Three towers, `n` disks
- Move all n disk from the source tower to the destination tower, using an auxiliary tower



Tower of Hanoi - rules

- One move at a time
- A bigger disk cannot be place on top of a smaller disk

- Simulator: https://www.mathsisfun.com/games/towerofhanoi.html

Easiest Case [Base Case]

- 1 disk:

Solution: move disk 1 from source to destination

- 2 disks:
- Move disk 1 from A (source) to C (auxiliary)
- Move disk 2 from A (source) to B (destination)
- Move disk 1 from C (auxiliary) to B (destination)

Use solution for n = 1 disk

Going to General Case

- If we can solve for `n = 2` disks, then for `n = 3` disks
  - we can use this solution to move the first two disks from source (A) to the auxiliary (C)
  - Then move the 3rd (largest) disk from source (A) to destination (B)
  - Then use the solution for `n = 2` disks from (C) to (B)
- To now solve for `n = 4` disks, apply solution for `n = 3` disks
  - …

Dividing the Problem into two parts

- Solve for `n - 1` to move all `n - 1` disks to auxiliary
- Move the last disk to destination
- Move the `n - 1` disks from auxiliary to destination

Tower of Hanoi Algorithm

```
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 tower
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
```

Base Case

General Case

Complexity

- How many steps (moves) are needed to solve the problem?
- `n = 1:`
  - 1 step
- `n = 2:`
  - 3 steps
- `n = 3`
  - 7 steps
- `n = 4`
  - 15 steps

$$f(n) = 2^n - 1$$

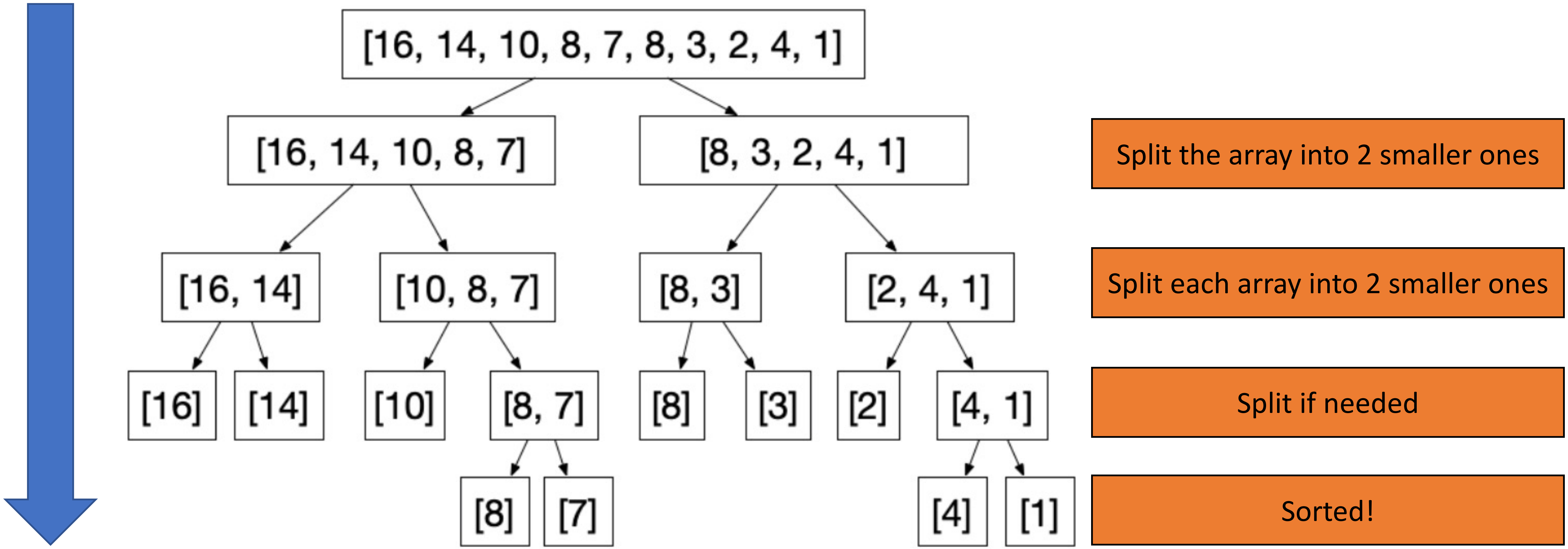
Week 3 - Merge Sort

“*Aucune idée pour une bonne blague sur ce sujet*” – French proverb

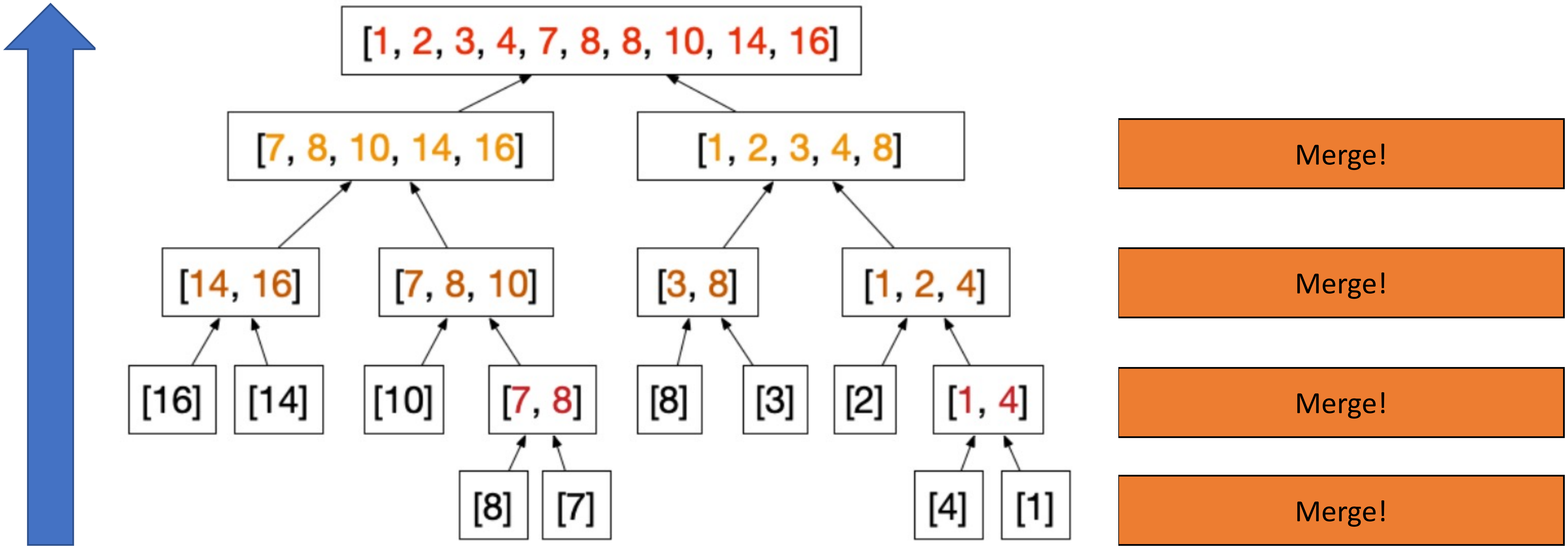
General Idea

- It’s troublesome to sort a list
- …Except if it only has 1 element
- Let’s split the large array into smaller ones until they only have 1 element

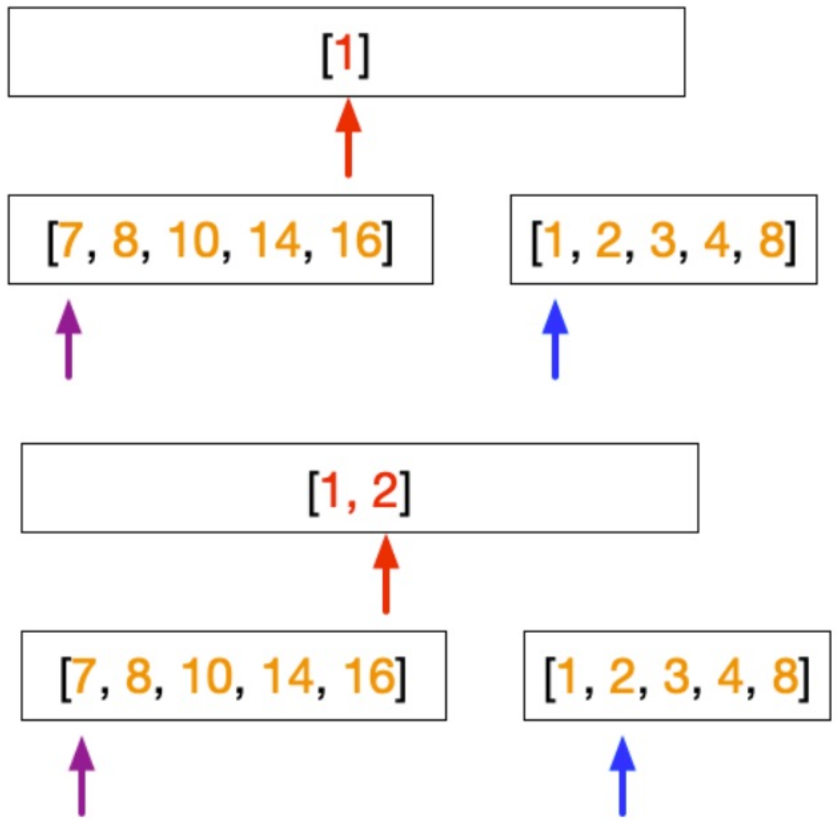
Merge Sort Illustration



Merge Sort Illustration (cont'd)



How to merge two sorted arrays?



- Start from the first element of both lists
  - Find the smallest one (1)
  - Insert it at the end of the new array
  - Move the “cursor” of the second list to the next element
- New iteration:
  - Find the smallest element (2)
  - Insert it
  - Move the “cursor” to the next

Base Case of merging two lists

- **If one of the two array is empty, insert all the remaining elements of the non-empty array**

Note: if we want to not use more space, we'll need to play with indexes a lot.

Merge_two_lists

- Let us implement merge_two_lists
- It takes two lists and returns a new list that contains all the elements of l1 and l2 sorted

MergeSort Algorithm

```
Merge Sort (recursive)
Input:
- array = sequence of integers
- p = index of beginning of array
- r = index of end of array
Output: None, sort the array in place
Steps:
1. if length(array) > 1, do:
1.1 calculate q = (p + r) / 2 # element in the middle
1.2 call MergeSort(array, p, q)
1.3 call MergeSort(array, q+1, r)
1.4 call Merge(array, p, q, r)
```

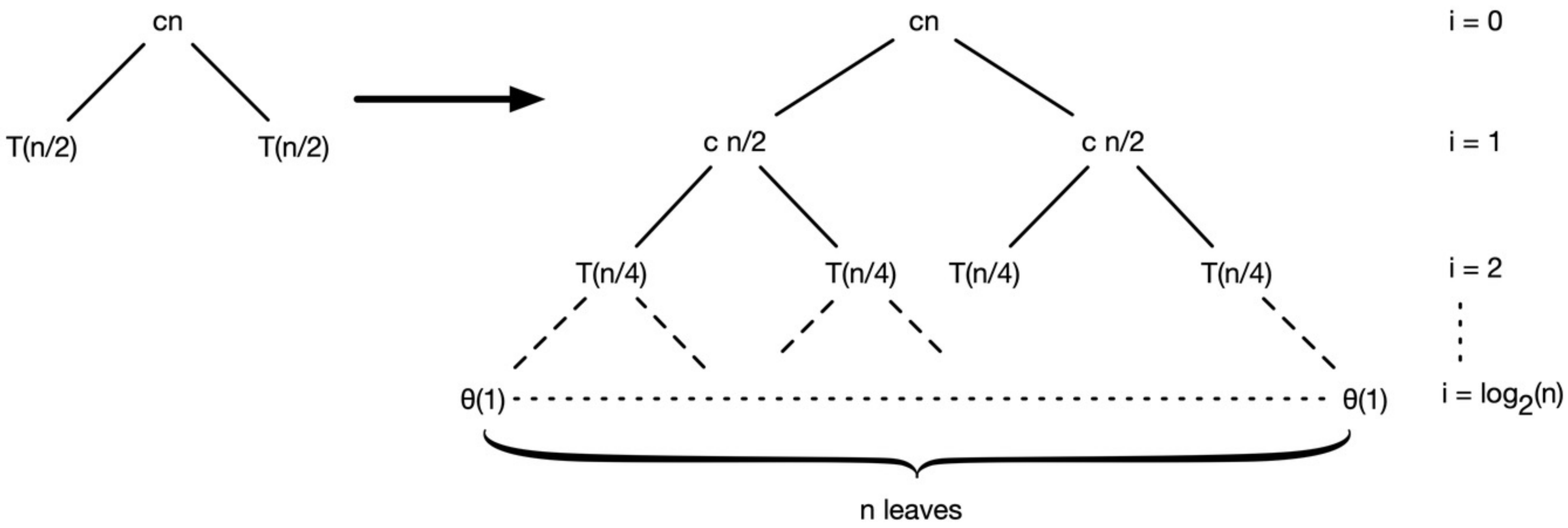
This is the code for mergesort_recursive

Merge Algorithm

Note: the algorithm was too long to fit a slide, let's discuss it on the whiteboard.

Compilation Time

- Every step, we divide the array by 2, to end up with n arrays of 1 element



Compilation Time

- Dividing by 2 means, if we have <= 2 elements, we need 1 step to break down the list into 2 sublists
- If we have <= 4 elements, we need 2 steps
- If we have <= 8, we need 3 steps
- If we have n elements we need $\log_2(n)$
- Every merge operation is of complexity n (need to go through n elements)

Overall complexity is there $n \times \log(n)$ (see notes)