Recursion

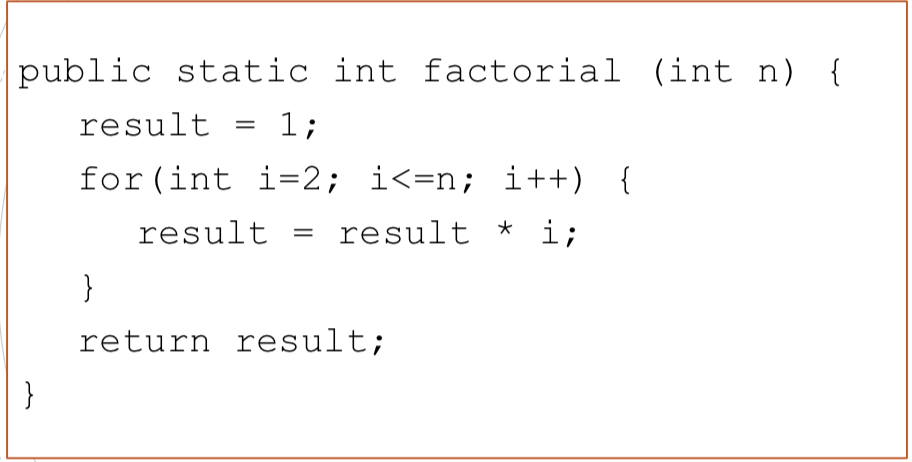
* *Base Case(s): one (or a finite number) of terminating scenario that does not use recursion to produce an answer.*
* *Recursive or Inductive step(s): rules that determine how to produce an answer from simpler cases.*

Note that if there is no base case in a recursive method, or if the base case is never reached, the execution will never end.

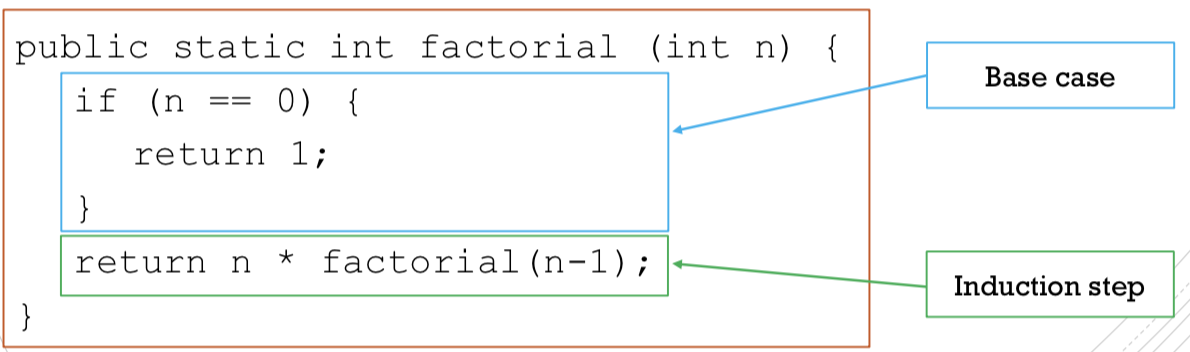
* Example 1: Factorial

𝑛! = 𝑛 ∗ (𝑛 − 1) ∗ (𝑛 − 2) ∗ (𝑛 − 3) ∗ ... ∗ 1

-iterative

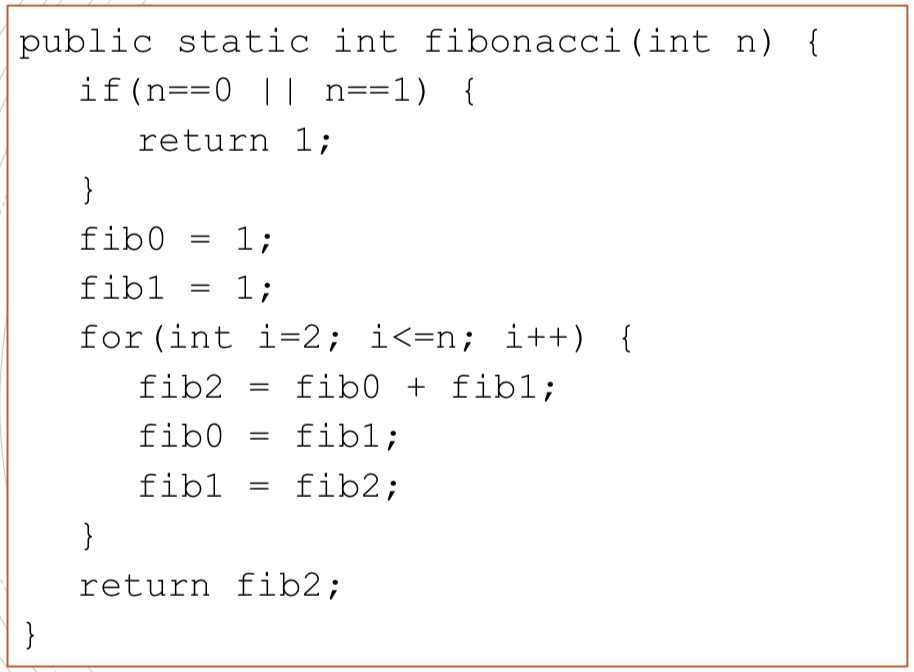


-recursive

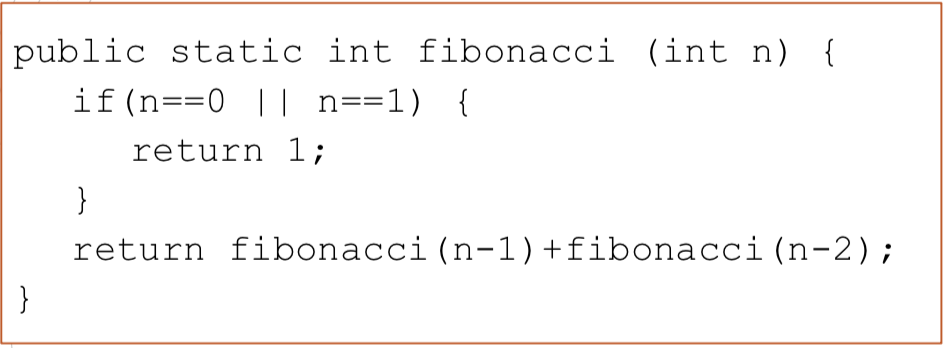


* Example 2: Fibonacci Numbers

-iterative

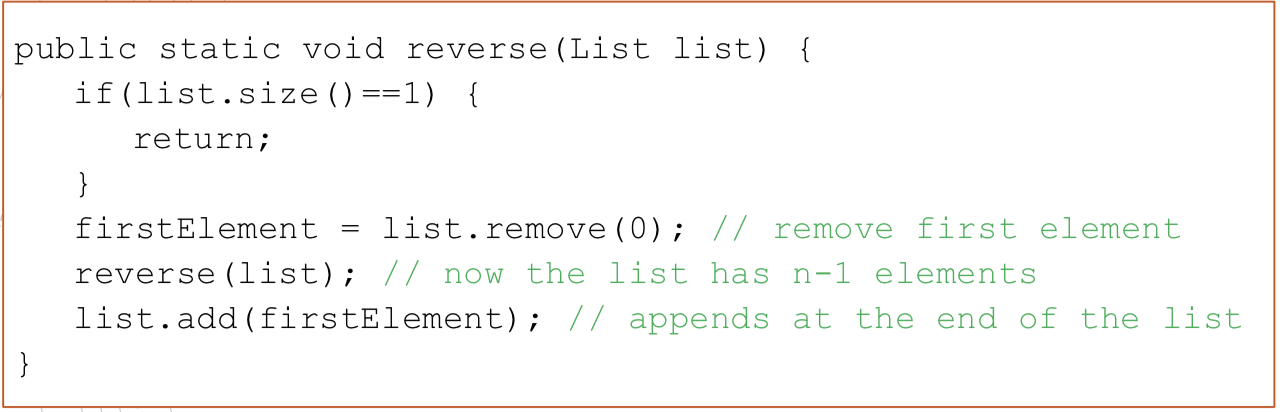


-recursive

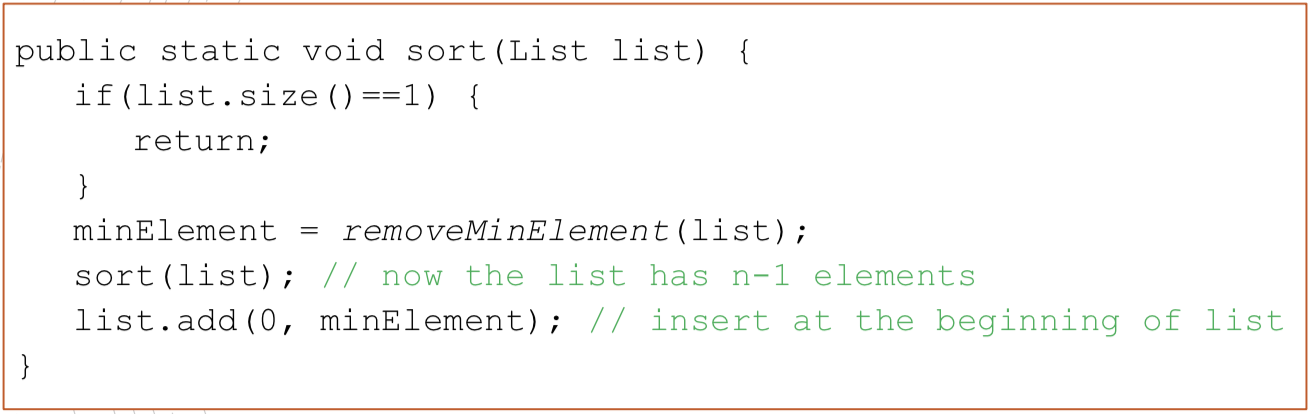


However, the recursive Fibonacci algorithm is very inefficient. It computes the same quantity many times

* Example 3: Reversing a List



* Example 4: Sorting a List

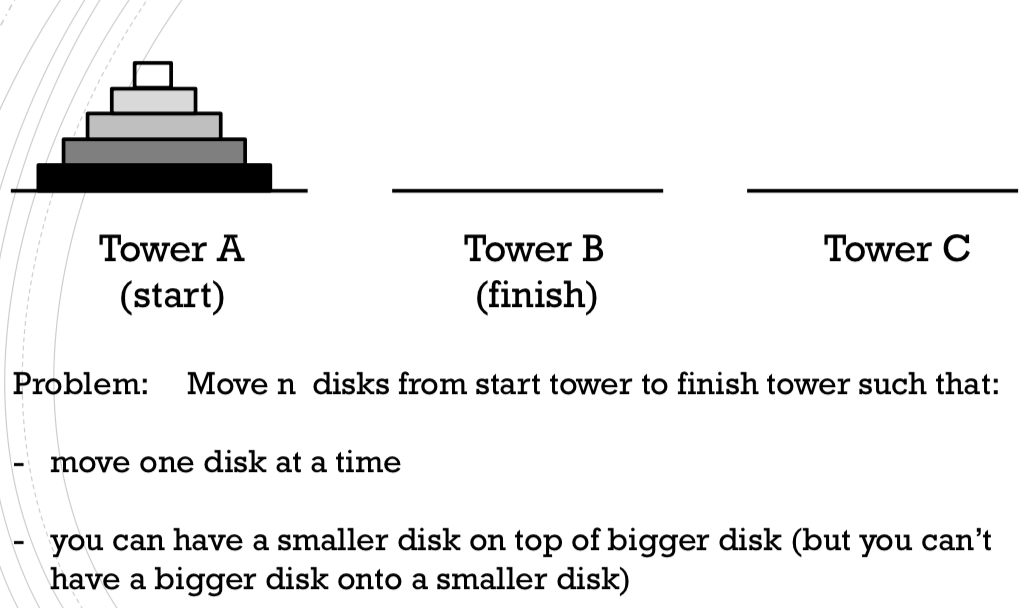


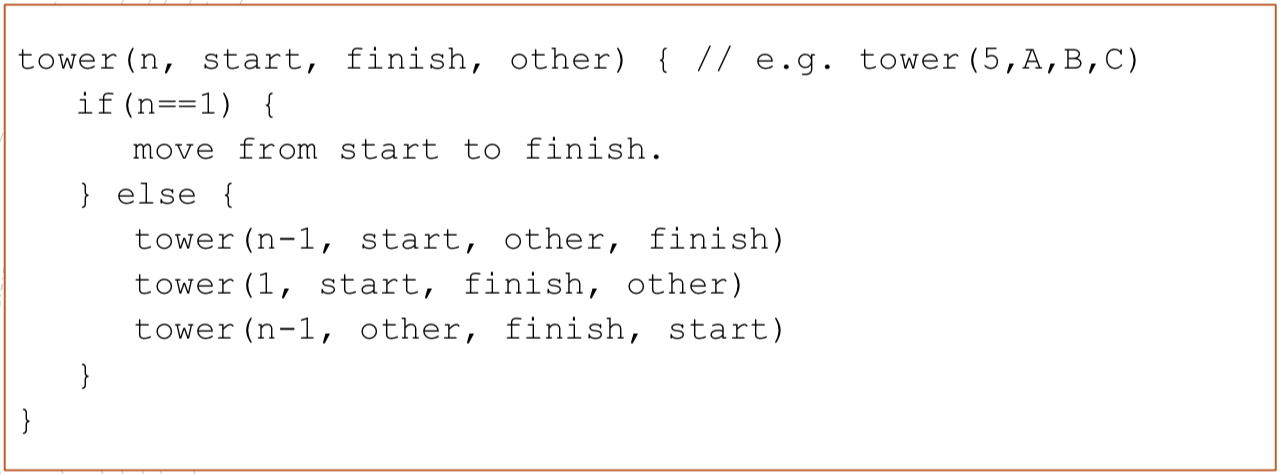
* Example 6: Tower of Hanoi

Problem: Move n disks from start tower to finish tower such that:

- move one disk at a time

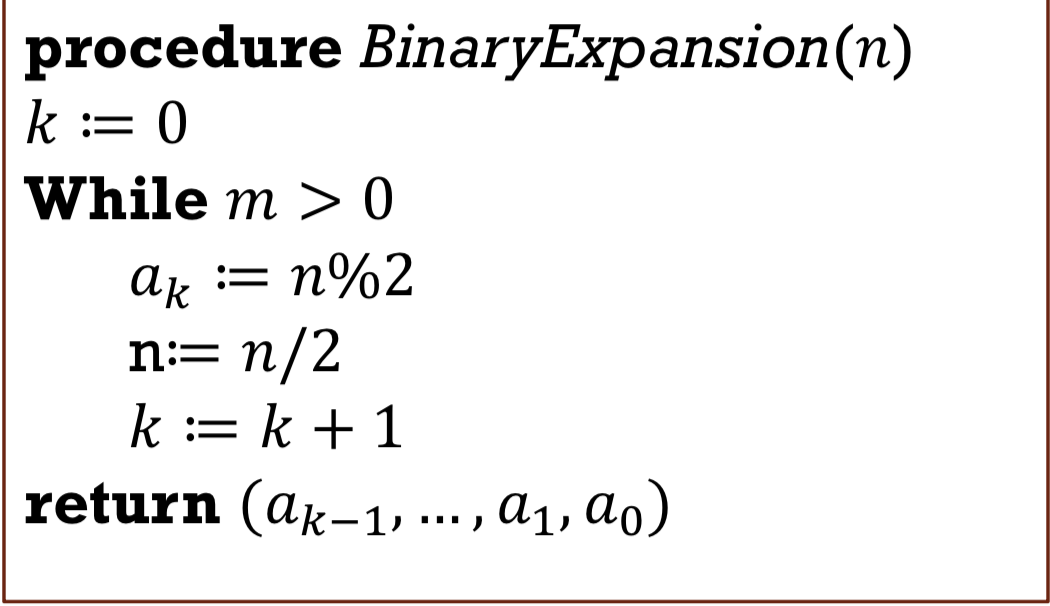
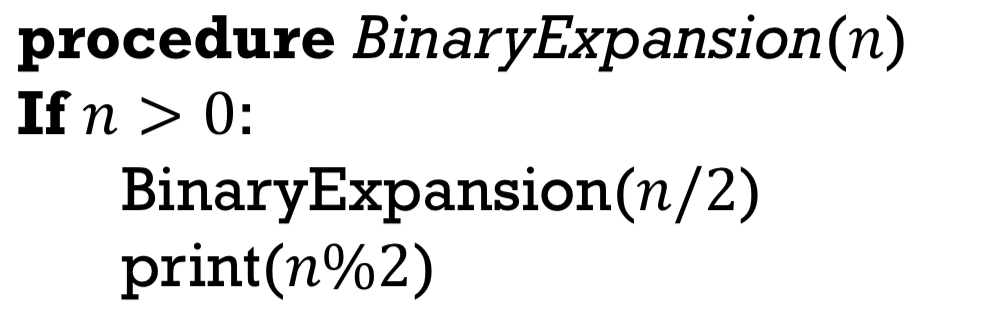
- you can have a smaller disk on top of bigger disk (but you can’t have a bigger disk onto a smaller disk)



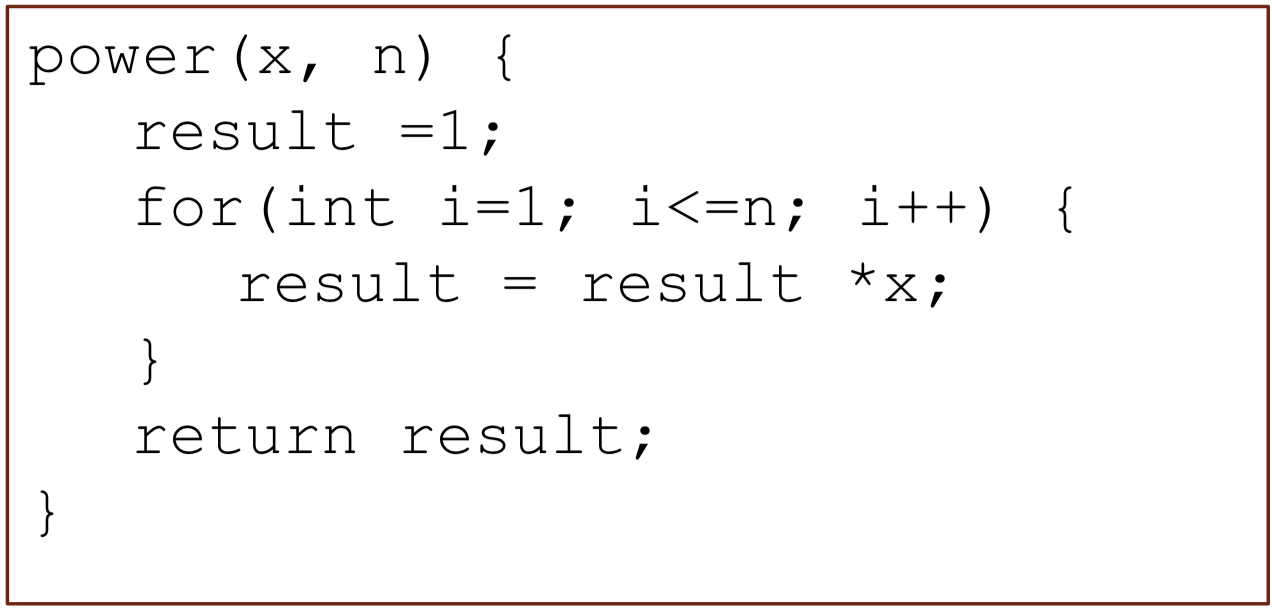
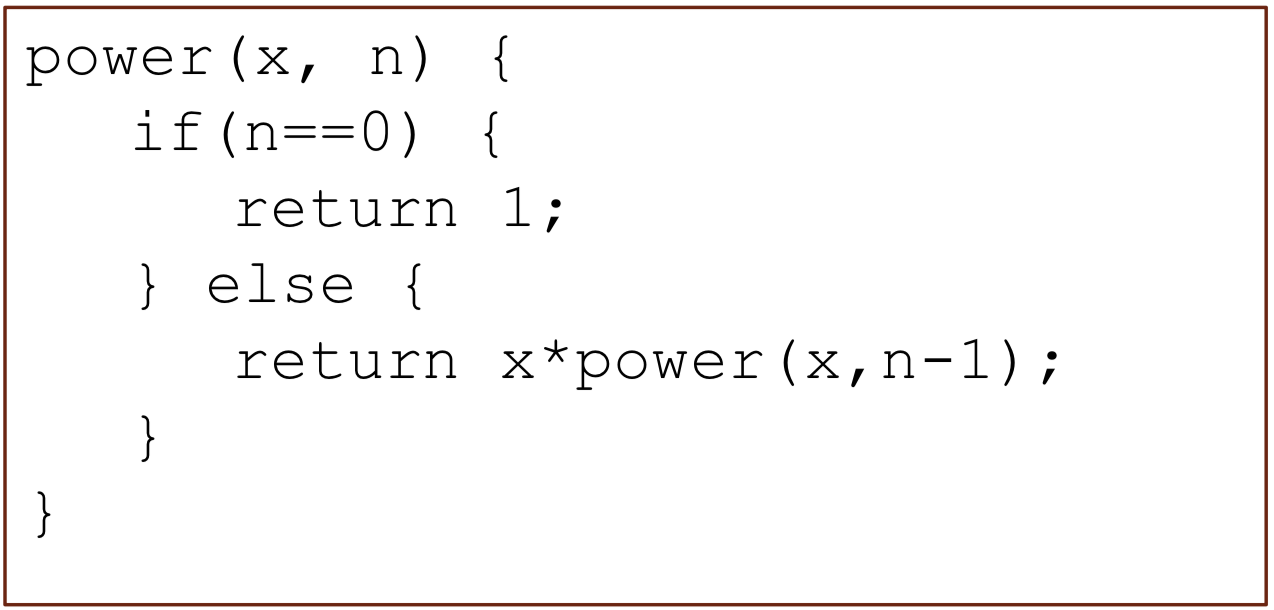


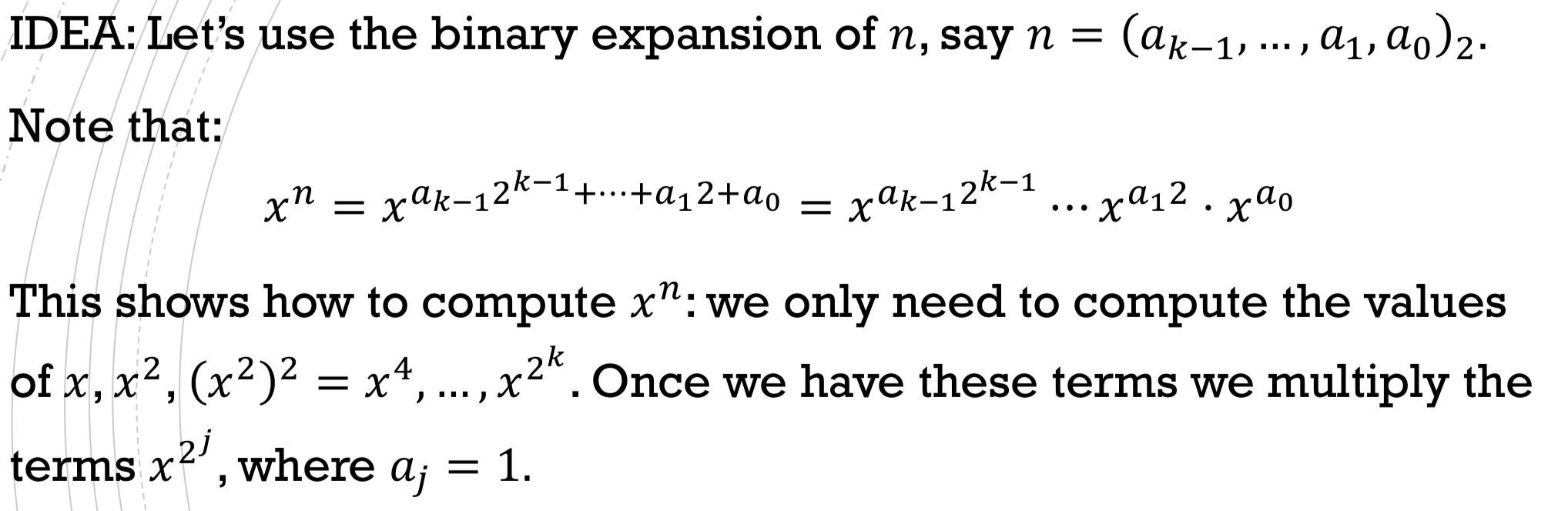
1+2+4+ ...+ 2𝑛−1 = 2𝑛 −1

* Example 7: Decimal to Binary



* Example 8: power



𝑥243: 𝑛 = (243)10 = (11110011)2

Q: How many multiplications do we need?

Method 1: Recursive method: 5\*2 + 2\*1 = 12 Iterative method: 7 + 7 = 14

The highest order bit in the recursive method doesn’t require multiplication.

The lowest order bit in the iterative method does not require multiplication.

Method 2: 𝑂(log2𝑛)---not faster

Let 𝑥 be a positive integer with M digits.

𝑥2 has about 2M digits.

𝑥3 has about 3M digits.

𝑥𝑛 has about 𝑛 ∗ M digits.

We cannot assume that multiplication takes ‘constant’ time.

Taking large powers gives very large numbers and multiplications becomes more expensive.

Binary Search

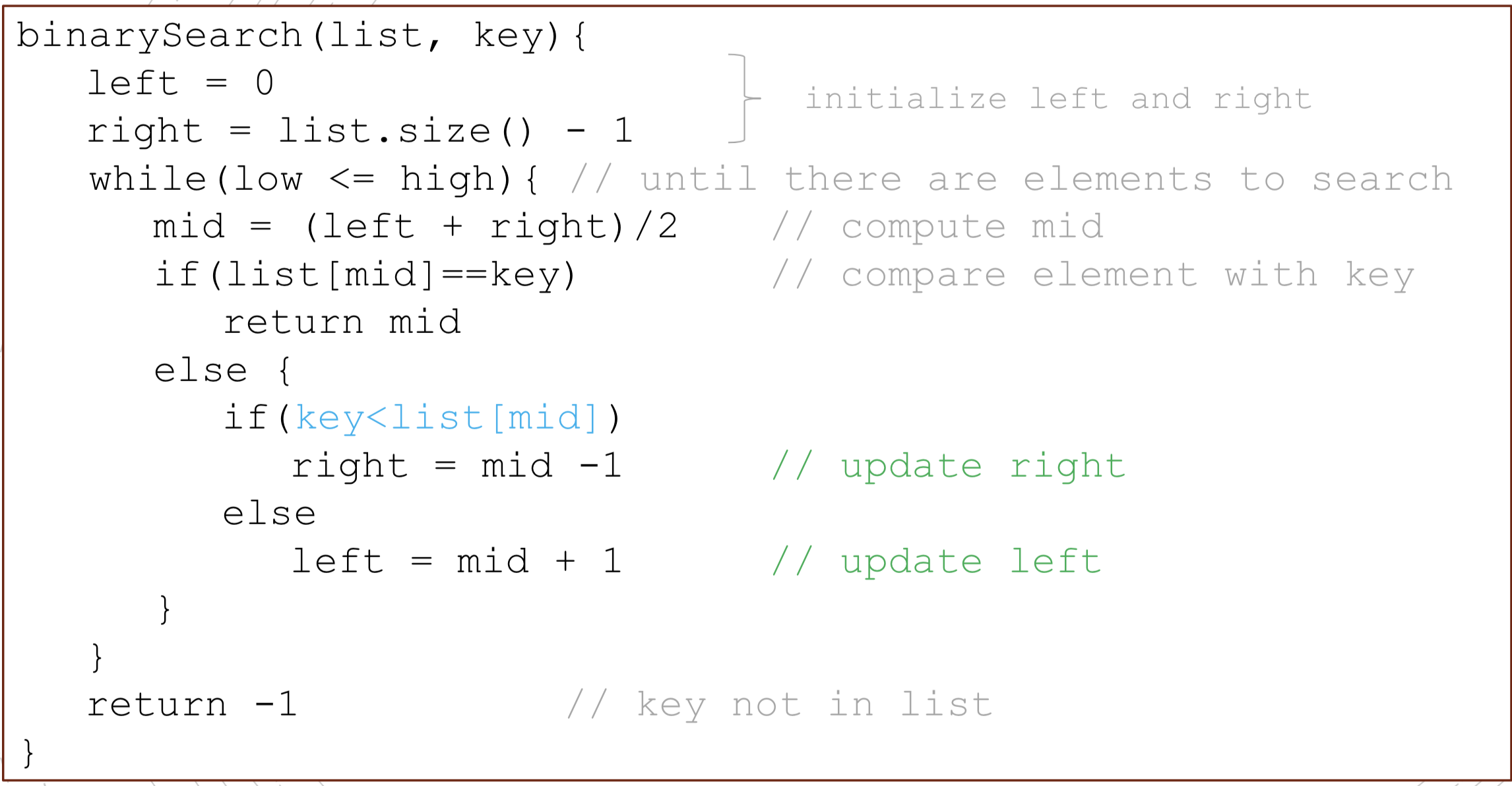
Inputs: A sorted list.

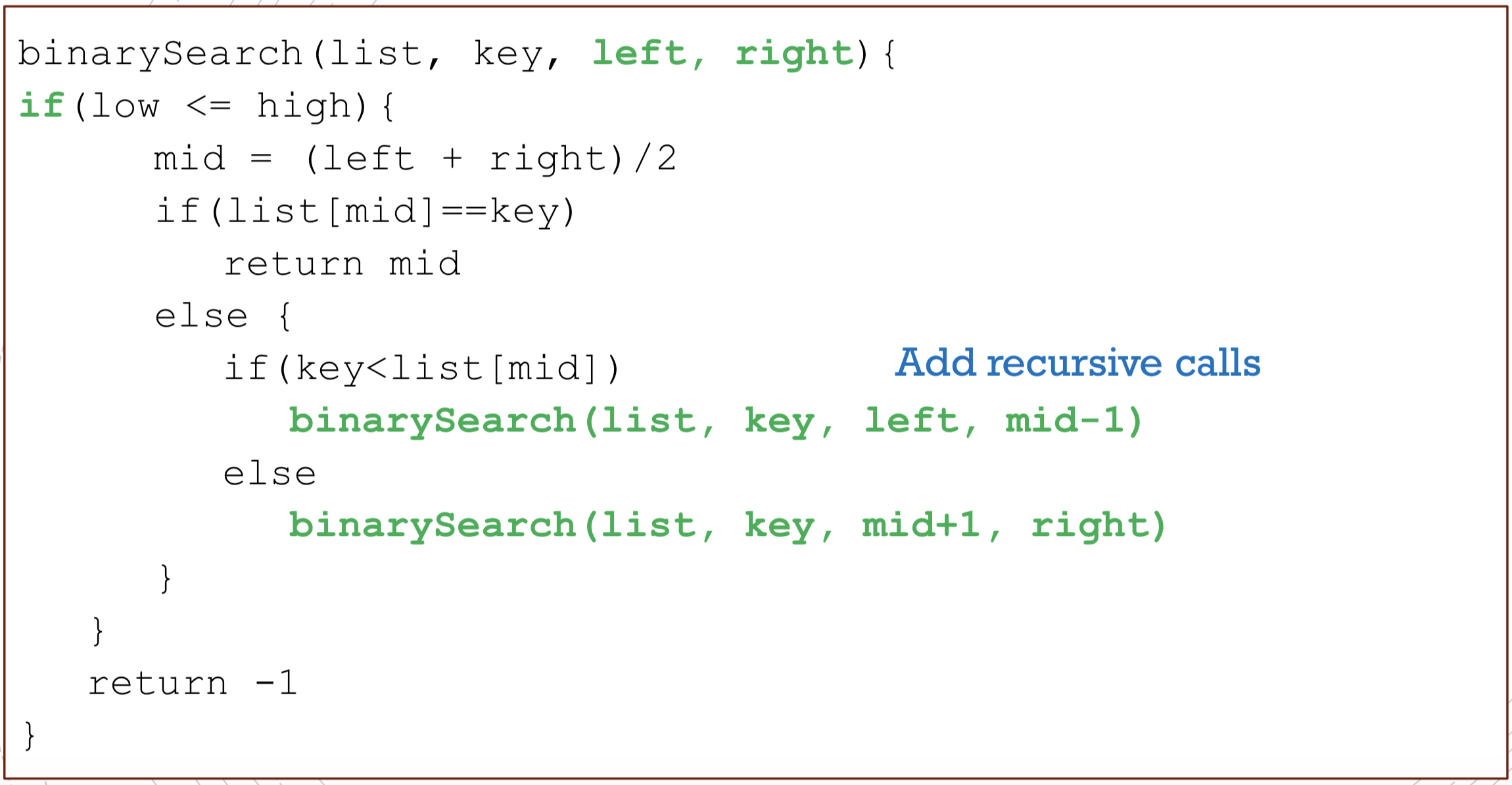
The element we are looking for (the key)

IDEA:

* First compare the key with the element in the middle of the list
* If the key is less than the middle element, we only need to search the first half of the list, so we continue searching on this smaller list.
* If the key is greater than the middle element, we only need to search the second half of the list, so we continue searching on this smaller list.
* If the key equals the middle element, we have a match – return its index.

Implementation : keep track of the left and right indices denoting the section to be searched.





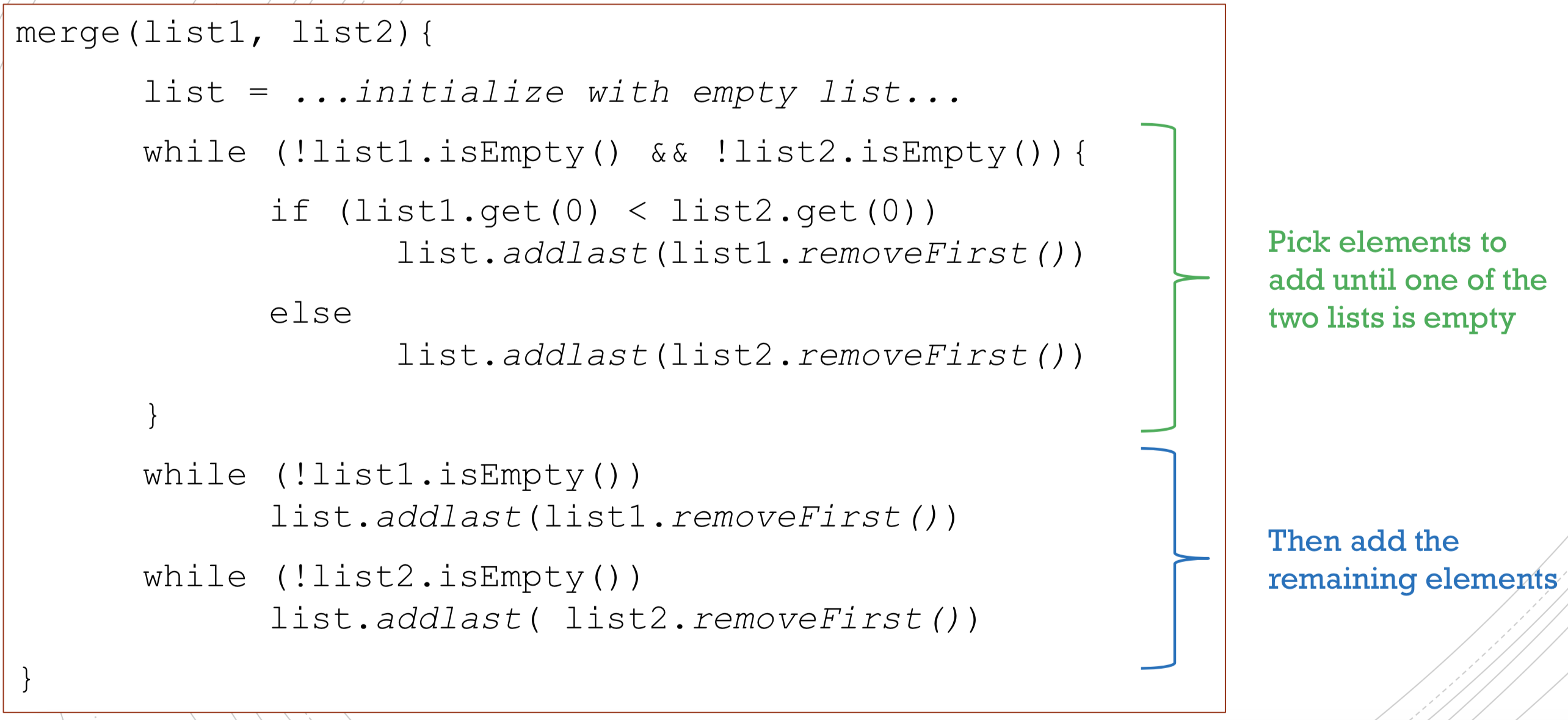
Time Complicity Worst Case: the element cannot be found. Then, worst time is 𝑂(𝑙𝑜𝑔2 𝑛) where 𝑛 is size of the list. Why? Because each time we are approximately halving the size of the list.

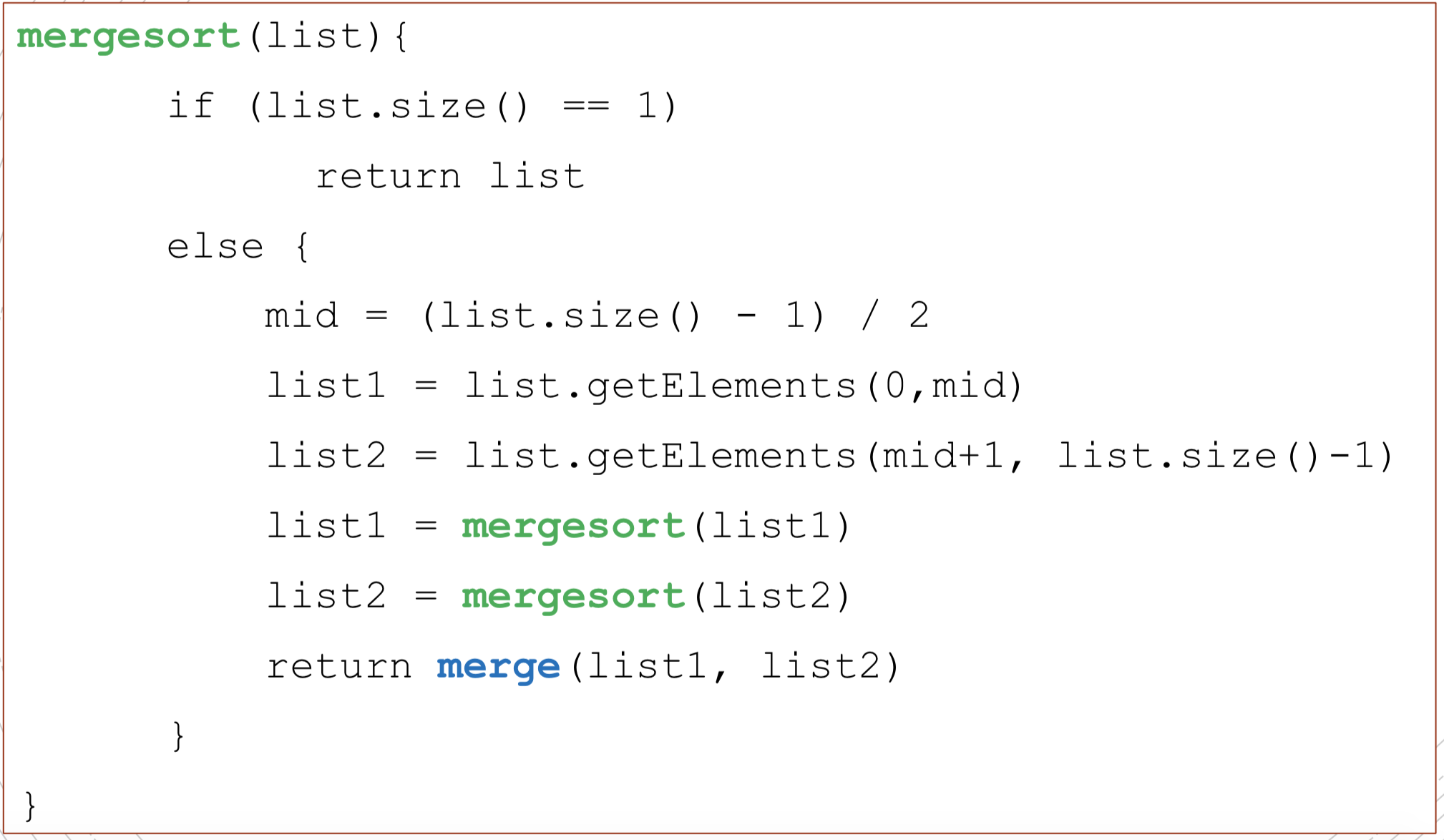
**MERGE SORT**

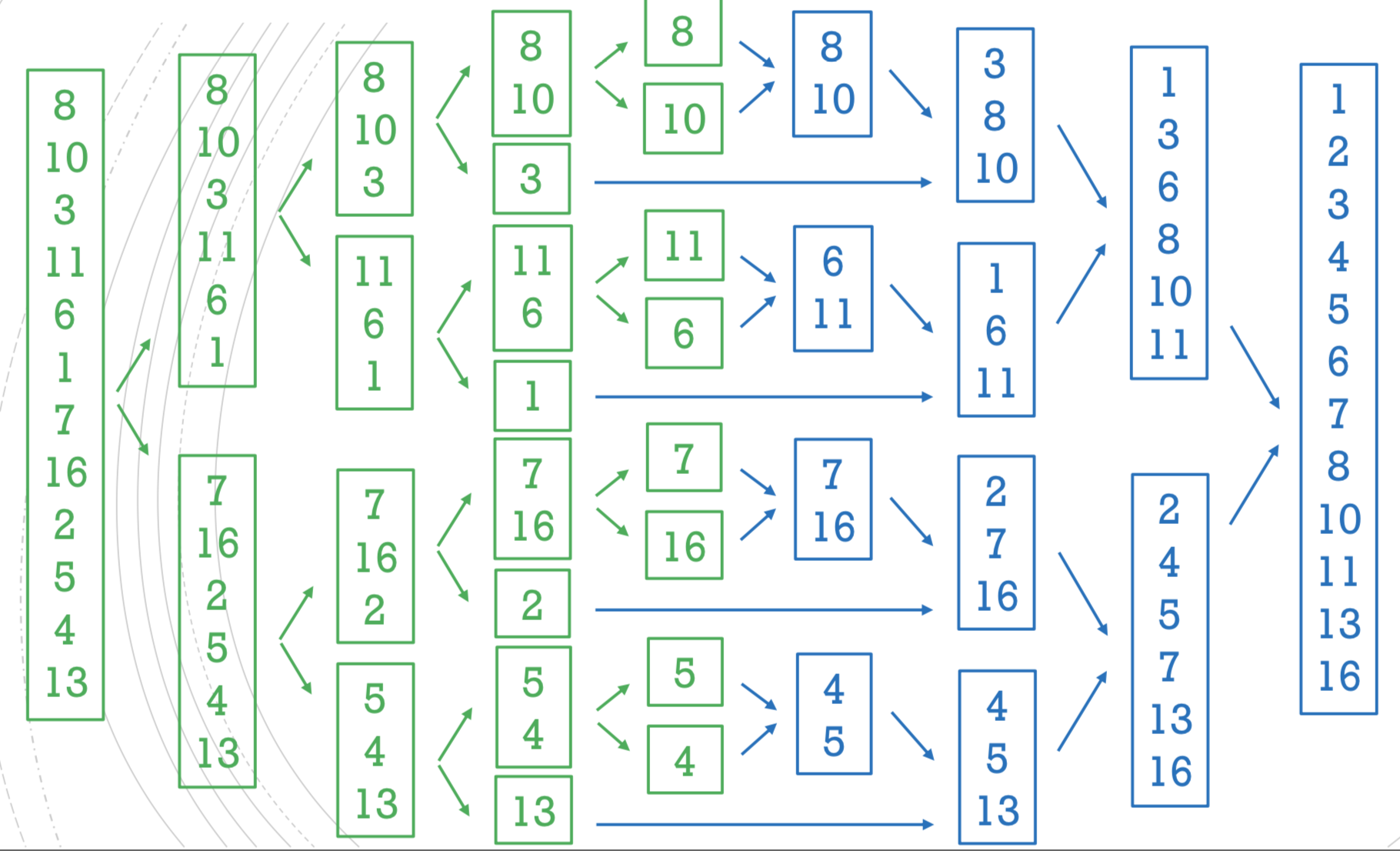
Merge Sort is a divide and conquer algorithm.

* Partition the list into two halves.
* Sort each half recursively
* Merge the sorted half maintaining the order.

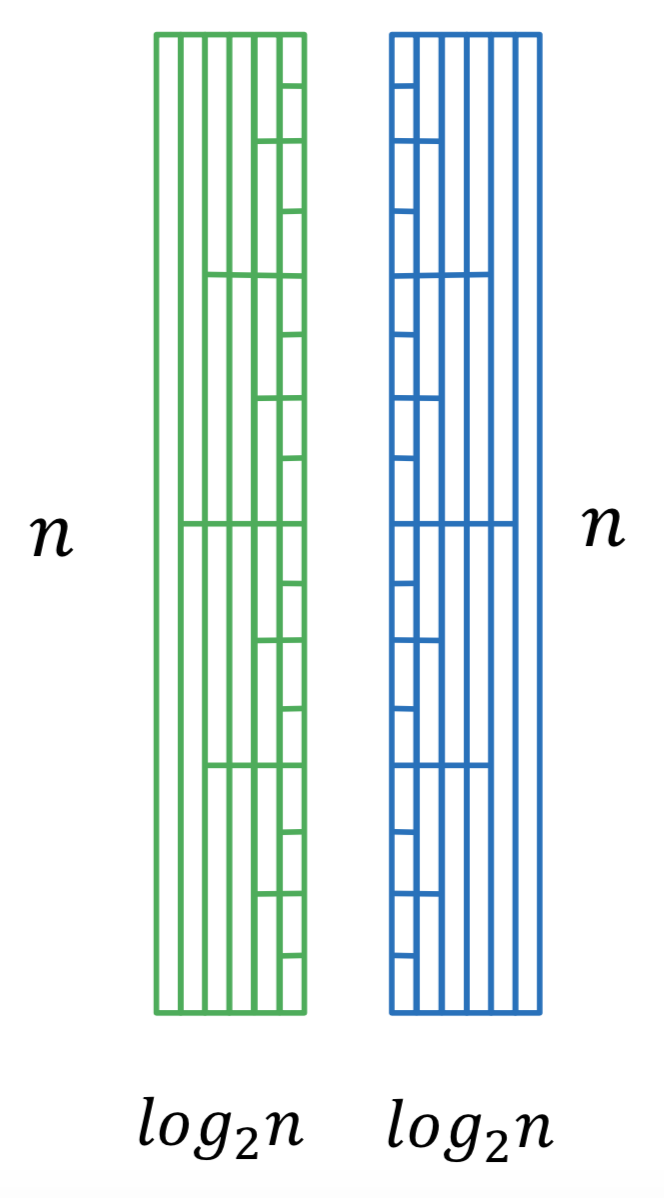
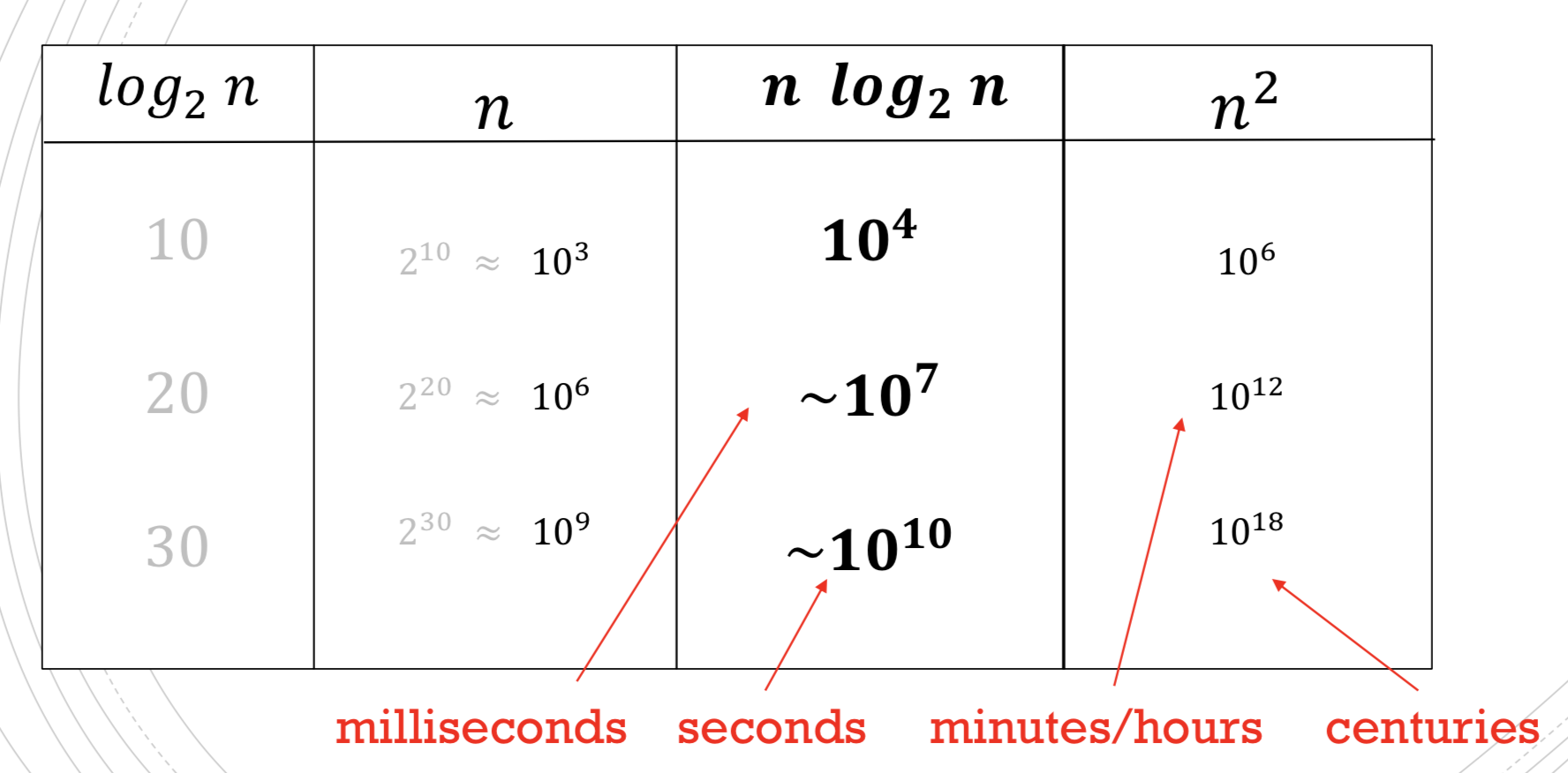








Time complexity: 𝑂(𝑛\*𝑙𝑜𝑔2𝑛)



QUICK SORT

Quick Sort is a divide and conquer algorithm.

 Pick an element of the array (the pivot).

 Partition the list moving the pivot to its correct position making sure that all the lower elements are on its left and all the larger elements are on its right.

 Sort the left part AND the right part of the list recursively. [recursive step]

 Keep doing it until there’s nothing left to sort. [base case]

Different versions of Quick Sort pick the pivot in different ways(affect running time):

 Always pick the first element as the pivot

 Always pick the last element as the pivot

 Pick a random element

 Pick the “median” as pivot (e.g. pick three random elements and choose its median)

implementation

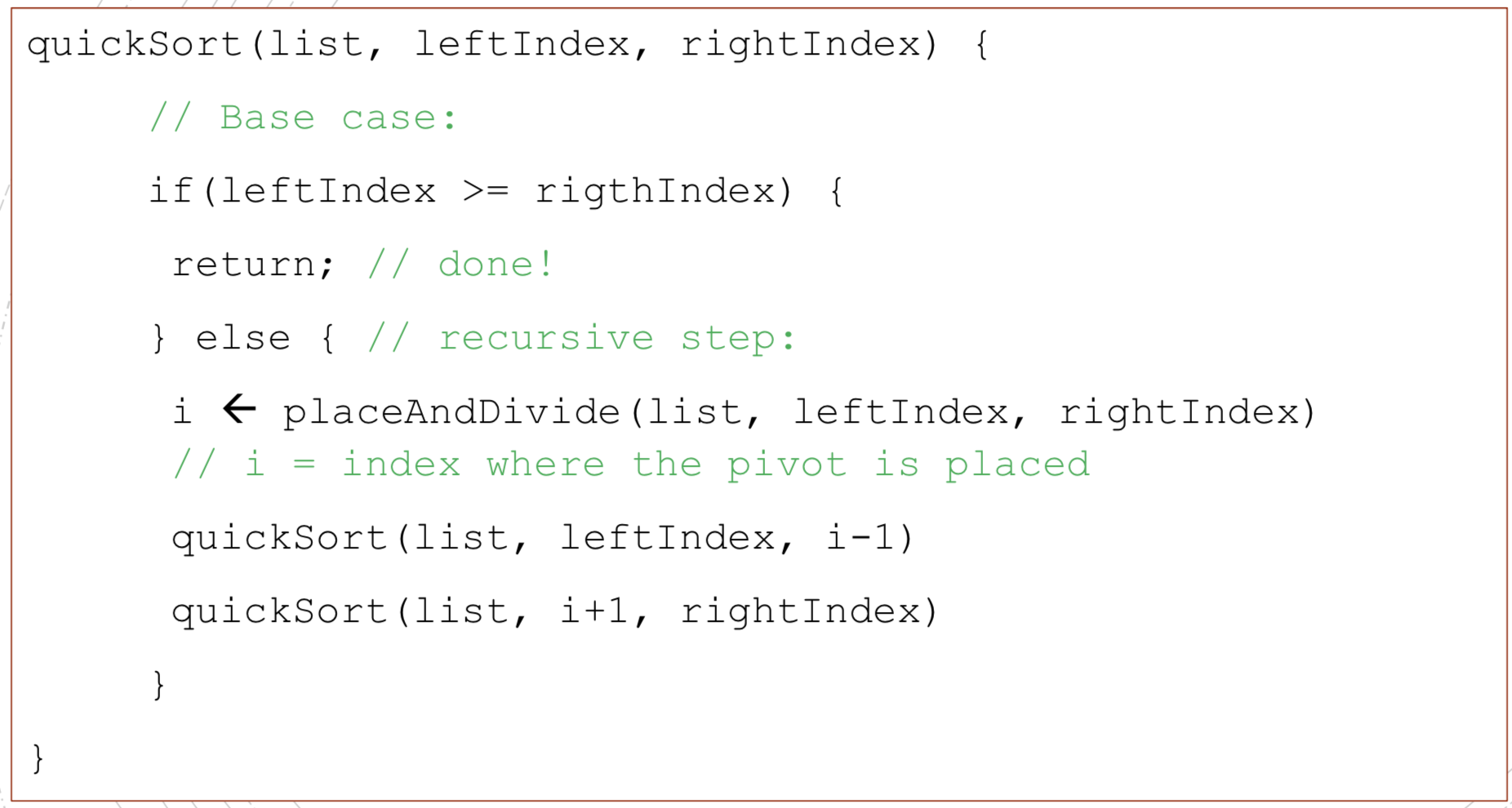
* A method that swaps two elements
* A way to refer to parts of the list
* A method that places the pivot in its correct position and moves the elements around so that all the lower elements are on the left, and all the larger elements are on the right. Call it placeAndDivide
* A method that implements the Quick Sort, that is:

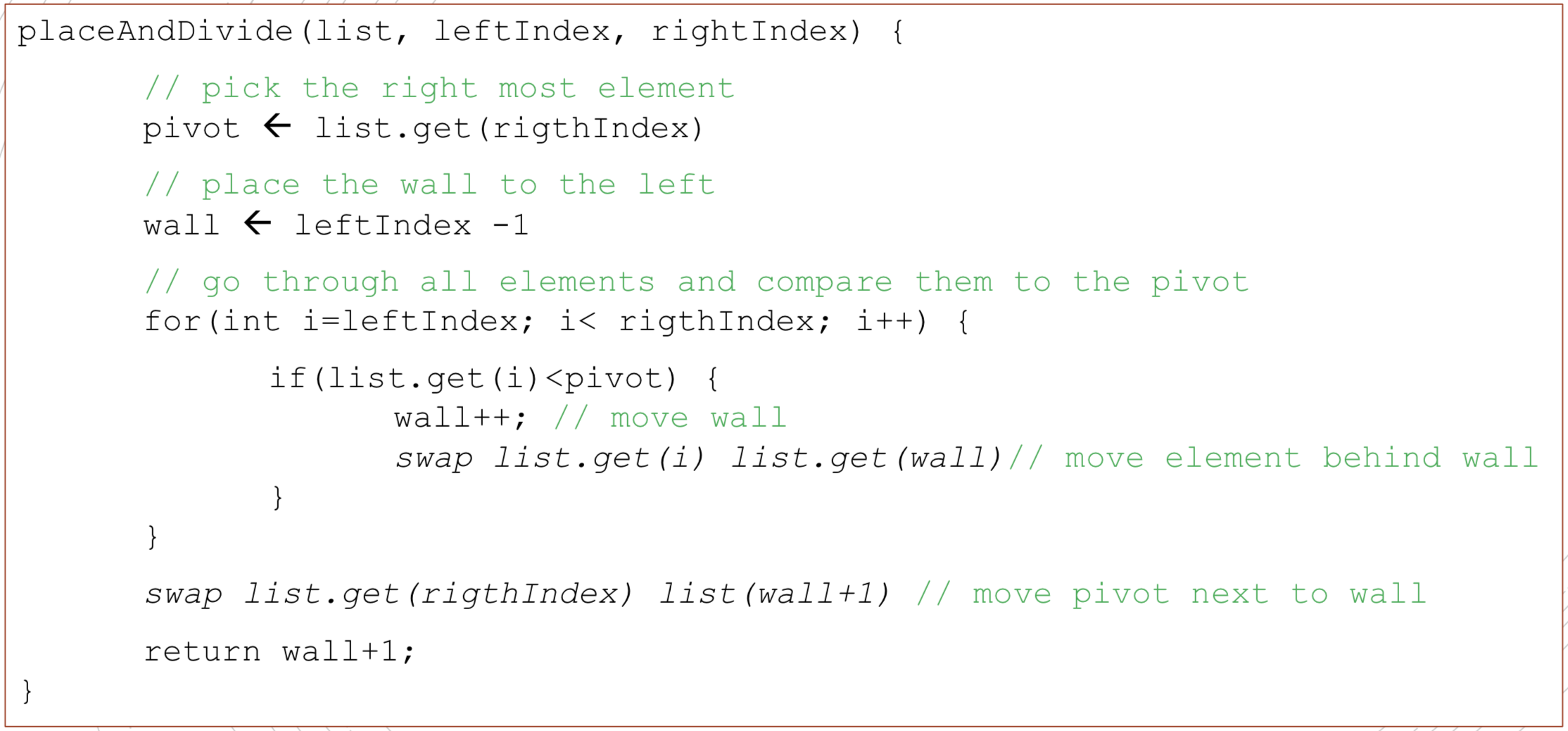
Pick a pivot

placeAndDivide

quickSort left part

quickSort right part





Merge sort VS Quick sort

* If we always choose the last element as our pivot:

The worst case for quick sort: always pick the largest element---O(N2)

The best case for quick sort: always pick the median element---O(n\*log2(N))

But we could choose another pivot to avoid worst case.

* However, merge sort typically uses an extra list. More space can hurt performance for big lists.