Big O and Sort (2)

Lecture 8

Hyung-Sin Kim

SNU Graduate School of Data Science

Announcement

- HW #7 due on 10/12

Review

- Search and sort are essential functions to process data
- Linear search
- Binary search
- Selection sort
- Insertion sort

Contents

- Big *O*
  - Time Complexity
  - Big *O* Notation
- Sort (2) Merge Sort and Recursion
  - Merge Sort Idea & Operation
  - Recursion
  - Merge Sort Implementation

Big *O*

- **Time Complexity**
- **Big O Notation**

Two Types of Program Cost

- Execution cost (our focus while learning algorithms)
  - Time complexity of a program (how much time?)
  - Memory complexity of a program (how much memory?)

- Programming cost (very important in practice, but not a focus in this course)
  - Development time
    - What if you develop a very nice program a year later than your competitor?
  - Readability, modifiability, and maintainability
    - Super important for real-world products (majority of cost actually…)

Measuring Time Complexity

- Measure execution time in seconds using a client program (e.g., time module)
  - **Pros**: Easy to measure. Gives actual time
  - **Cons**: large amounts of time might be required. Results depend on lots of factors (machine, compiler, data…)

- Count possible operations in terms of input list size **N**
  - **Pros**: Machine independent. Gives algorithm's scalability
  - **Cons**: Tedious to compute… Does not give actual time

⇒ Fortunately, we usually care only about asymptotic behavior (with a very large **N** – Big Data!)

Count Possible Operations

```
def linear_search_for(L: list, value: Any) -> int:
    for i in range(len(L)):
        if L[i] == value:
            return i
    return -1
```

```
def selection_sort(L: list) -> None:
    for i in range(len(L)):
        smallest = i
        for j in range(i+1, len(L)):
            if L[j] < L[smallest]:
                smallest = j
        L[i], L[smallest] = L[smallest], L[i]
```

- Assume that input list size is **N**

Operation	Count
==	1 to N

Operation	Count
Smallest = i	
<	
Smallest = j	
Swapping	

Count Possible Operations

```
def linear_search_for(L: list, value: Any) -> int:
    for i in range(len(L)):
        if L[i] == value:
            return i
    return -1
```

- Assume that input list size is **N**

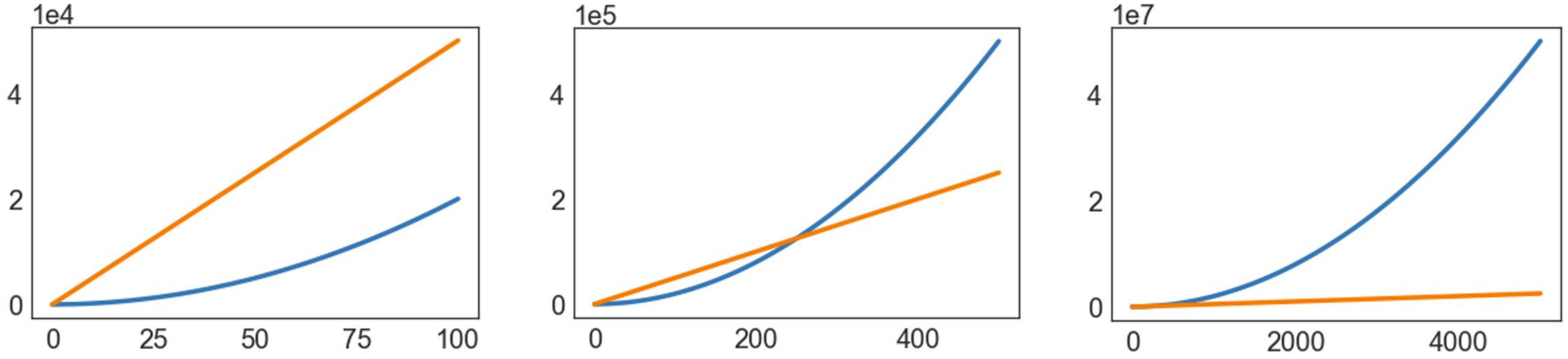
Operation	Count
==	1 to N

```
def selection_sort(L: list) -> None:
    for i in range(len(L)):
        smallest = i
        for j in range(i+1, len(L)):
            if L[j] < L[smallest]:
                smallest = j
        L[i], L[smallest] = L[smallest], L[i]
```

Operation	Count
Smallest = i	N
<	
Smallest = j	
Swapping	N

What is Important for Asymptotic Analysis?

- Compare the two algorithms below:
  - Algorithm 1 requires $2N^2$ operations
  - Algorithm 2 requires 500N operations
- Algorithm 1 is faster than Algorithm 2 for a small N, but becomes much slower for a very large N
  - What is important?: Not a specific value but a function **shape**! (parabola vs. line)
  - **Order of growth**



The figures are from 61B course material at UC Berkeley

How can we characterize an algorithm’s time complexity more ***formally*** *and* ***simply****?*

Simplification to Find "Order of Growth"

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance

Operation	Count
Smallest = i	N
<	
Smallest = j	
Swapping	N

Simplification to Find "Order of Growth"

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance
- 2. Focus on only one operation that has the highest order of growth
  - There could be multiple good choices. Then, just choose any of them.

Operation	Count
Smallest = i	~~N~~
<	
Smallest = j	
Swapping	~~N~~

Simplification to Find "Order of Growth"

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance
- 2. Focus on only one operation that has the highest order of growth
  - There could be multiple good choices. Then, just choose any of them.
- 3. Remove lower order terms

Operation	Count
Smallest = j	

Simplification to Find "Order of Growth"

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance
- 2. Focus on only one operation that has the highest order of growth
  - There could be multiple good choices. Then, just choose any of them.
- 3. Remove lower order terms
- 4. Remove constants
  - We have already thrown away information at step 2. At this stage, constants are not meaningful

- Worst-case order of growth of **selection sort**
  - N^2

Operation	Count
Smallest = j	—

Big *O*

- Time Complexity
- **Big O Notation**

Formal Definition

- If a function $T(N)$ has its order of growth less than or equal to $f(N)$,
- we write this as $\boldsymbol{T(N) \in O(f(N))}$
- where $\boldsymbol{O}$ is called **Big-O** notation

- More mathematically, $\boldsymbol{T(N) \in O(f(N))}$ means that
- there exists positive constants k such that
- $T(N) \leq k \cdot f(N)$ for all values of N greater than some N_0 (i.e., very large N)

Examples

- Simplify T(N) to find f(N) and use the Big-O notation

Function T(N)	Order of Growth in terms of Big-O

Examples

- Simplify T(N) to find f(N) and use the Big-O notation

Function T(N)	Order of Growth in terms of Big-O

Summary

- Given a program, we can express its time complexity as a function T(N) where N is an input characteristic (usually the input size)
- We can extract **Big O** from T(N) by focusing only on its worst-case order of growth

Sort (2) Merge Sort and Recursion

- **Merge Sort Idea & Operation**
- Recursion
- Merge Sort Implementation

Motivation

- Insertion sort and selection sort work but too slow – proportional to n^2
  - Does not matter when handling small data, but we want to handle **big data**!

- Recall linear search vs. binary search – Divide the whole task into **two parts**
  - Is there a way something similar?

Merge sort!

index	0	1	2	3	4	5	6	7
values	5	-2	0	100	-6	7	4	9

Merge Sort – Idea

- Step 1: **Divide** the whole list into two sub-lists

Sublist1 *Sublist2*

index	0	1	2	3	4	5	6	7
values	5	-2	0	100	-6	7	4	9

Merge Sort – Idea

- Step 1: **Divide** the whole list into two sub-lists
- Step 2: **Sort** the left sublist and the right sublist separately
  - Smells like **binary** something…

Sublist1 – sorted! *Sublist2 – sorted!*

index	0	1	2	3	4	5	6	7
values	-2	0	5	100	-6	4	7	9

Merge Sort – Idea

- Step 1: **Divide** the whole list into two sub-lists
- Step 2: **Sort** the left sublist and the right sublist separately
    - Smells like **binary** something…
- Step 3: **Merge** the two sorted sublist in a sorted way

Merge sublist1 and sublist2!

index	0	1	2	3	4	5	6	7
values	-6	-2	0	4	5	7	9	100

How to sort sublists?

Merge Sort – Idea

- Step 1: **Divide** the whole list into two sub-lists
- Step 2: **Sort** the left sublist and the right sublist separately, by using **merge sort**
  - Smells like **binary** something…
- Step 3: **Merge** the two sorted sublist in a sorted way

*Sublist1 – **mergesort**!*

*Sublist2 – **mergesort**!*

index	0	1	2	3	4	5	6	7
values	5	-2	0	100	-6	7	4	9

Merge Sort – Operation (Breakdown)

0	1	2	3	4	5	6	7
5	-2	0	100	-6	7	4	9

Mergesort(L)

Merge Sort – Operation (Breakdown)

Mergesort(L)

0	1	2	3	4	5	6	7
5	-2	0	100	-6	7	4	9

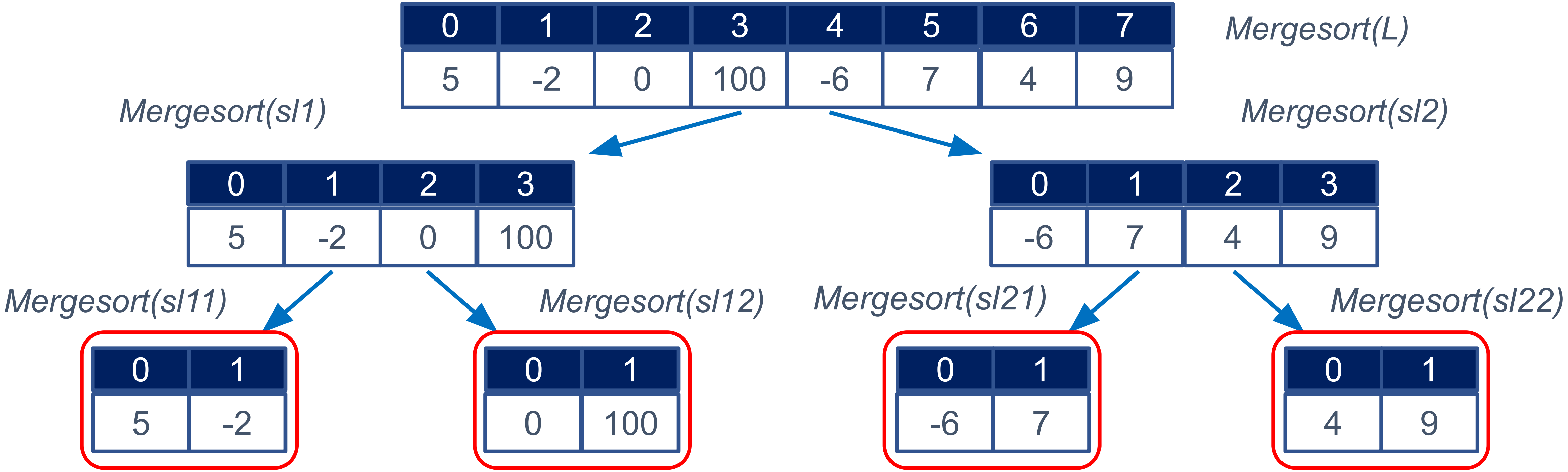
Mergesort(sl1)

0	1	2	3
5	-2	0	100

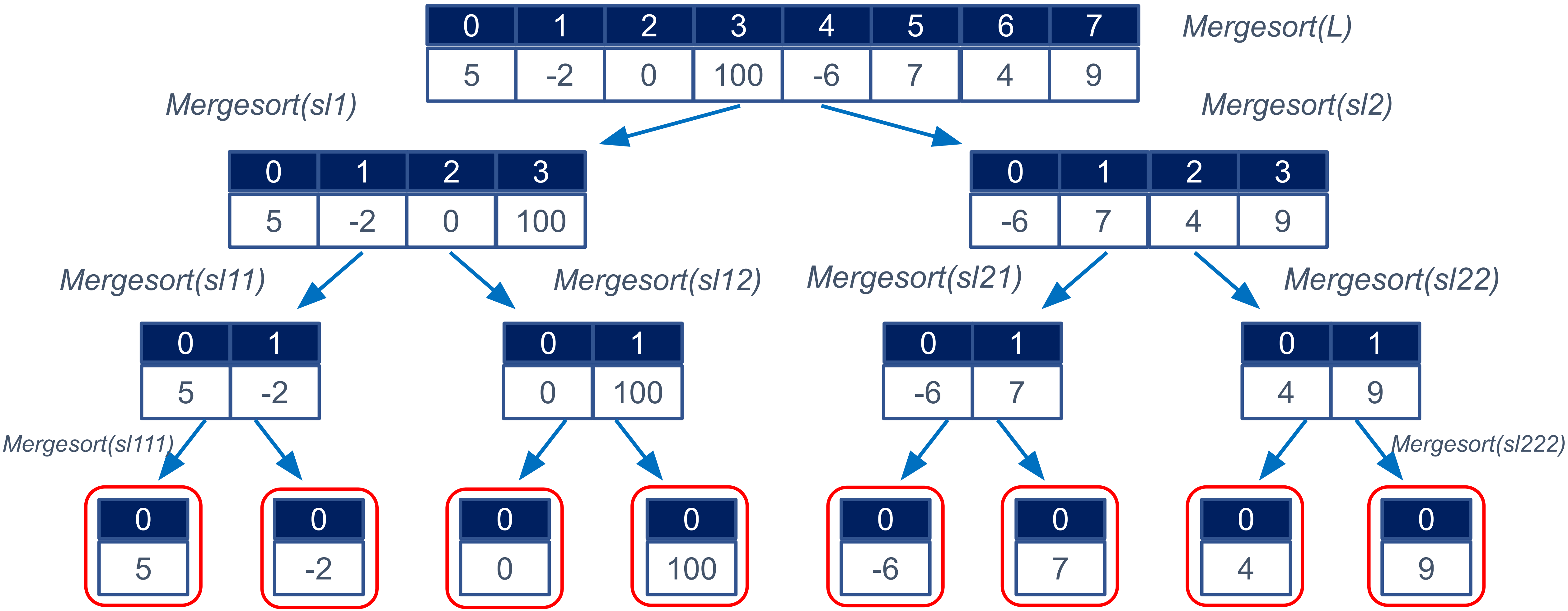
Mergesort(sl2)

0	1	2	3
-6	7	4	9

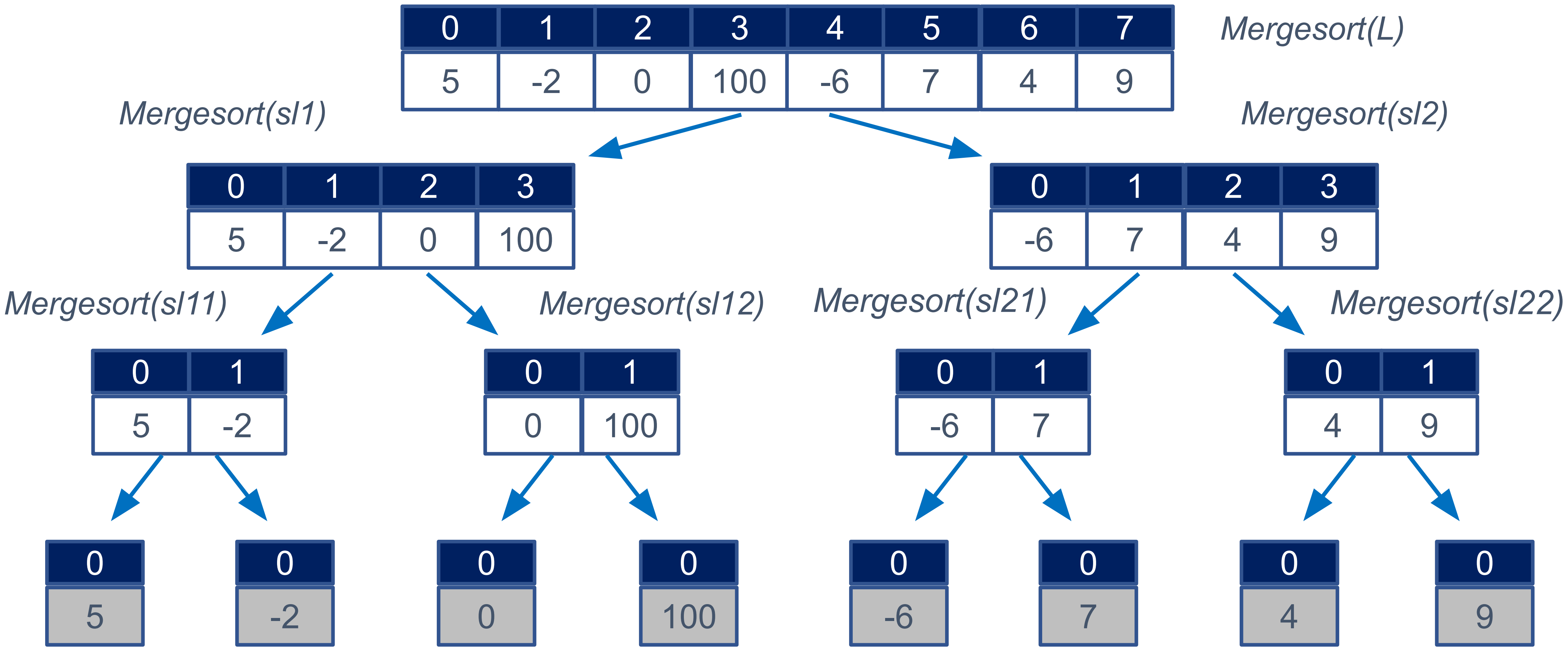
Merge Sort – Operation (Breakdown)



Merge Sort – Operation (Breakdown)

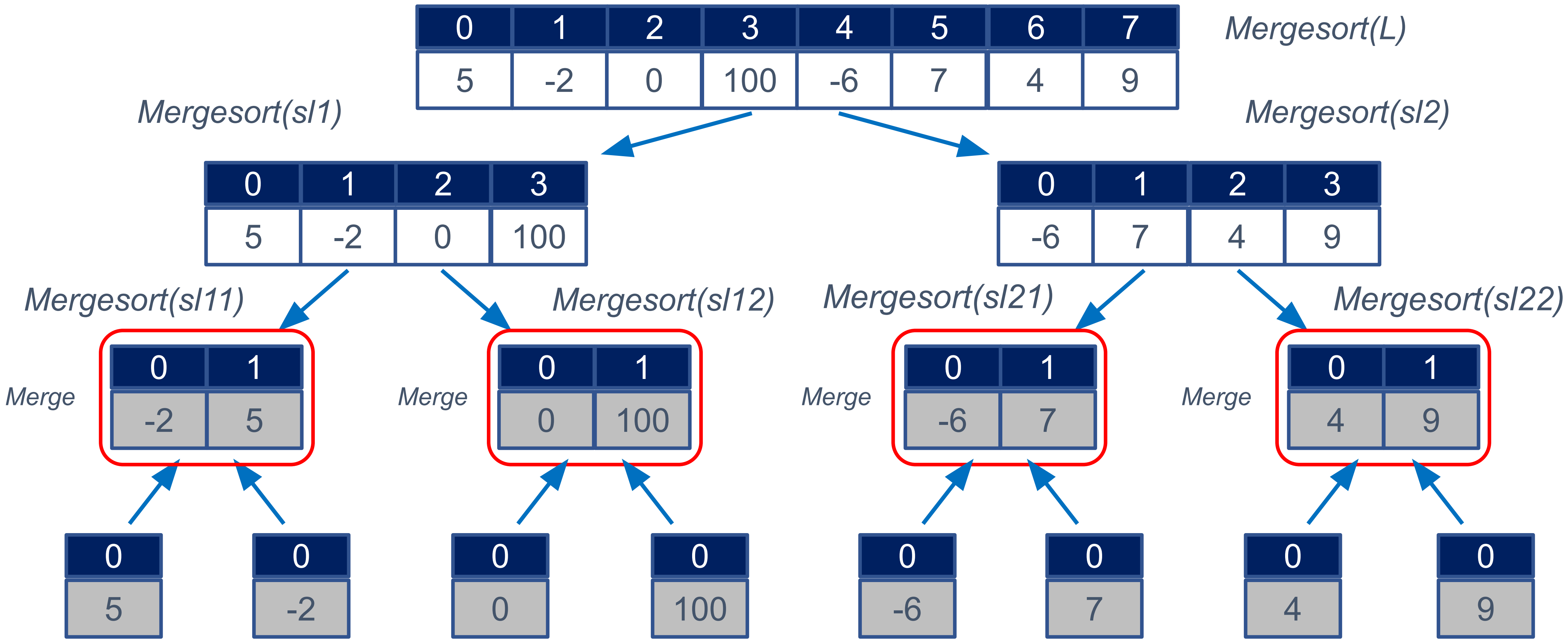


Merge Sort – Operation (Breakdown)

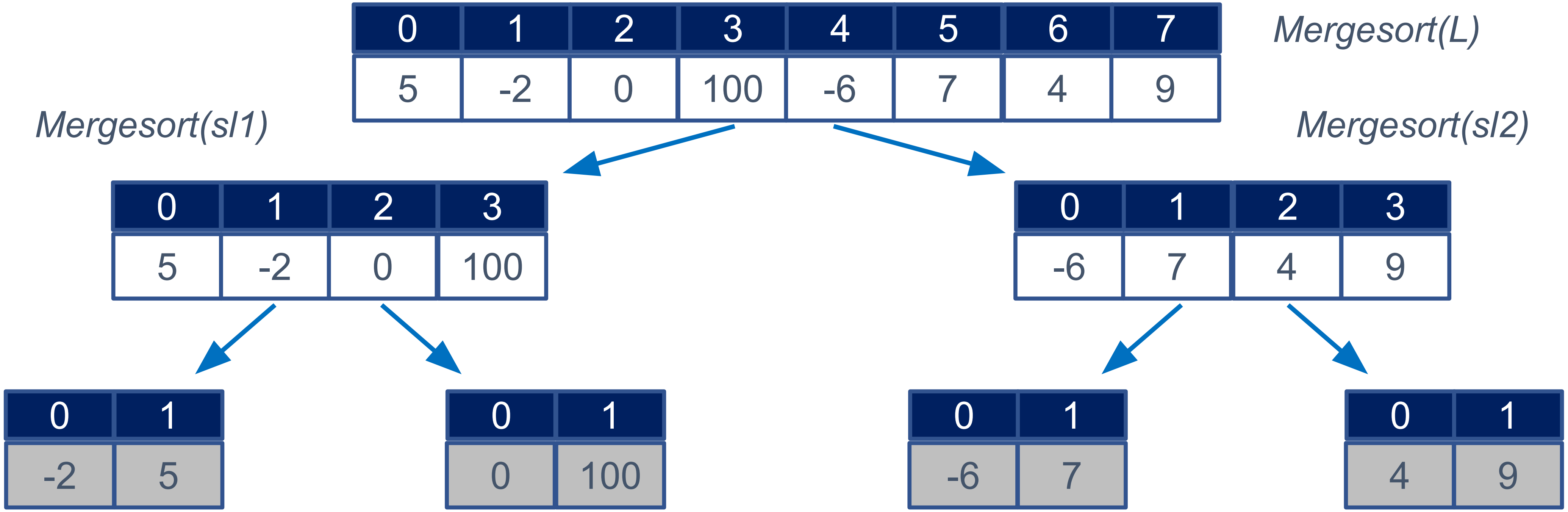


Time to go upwards!

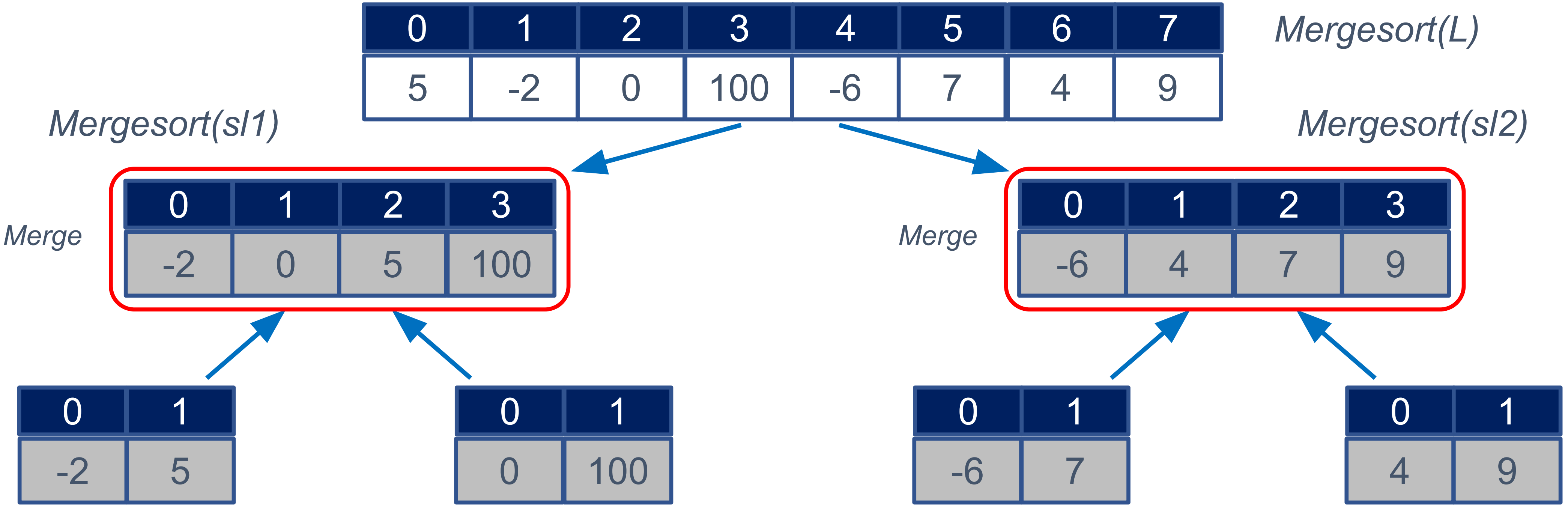
Merge Sort – Operation (Merge)



Merge Sort – Operation (Merge)



Merge Sort – Operation (Merge)



Compare the ***leftmost items*** *of the two sublists, given that the two lists are* ***already sorted****!*

Merge Sort – Operation (Merge)

Mergesort(L)

0	1	2	3	4	5	6	7
5	-2	0	100	-6	7	4	9

0	1	2	3
-2	0	5	100

0	1	2	3
-6	4	7	9

Merge Sort – Operation (Merge)

Merge

0	1	2	3	4	5	6	7
-6	-2	0	4	5	7	9	100

Mergesort(L)

0	1	2	3
-2	0	5	100

0	1	2	3
-6	4	7	9

*Again, compare the **leftmost items** of the two sublists, given that the two lists are **already sorted**!*

Merge Sort – Operation (Merge)

0	1	2	3	4	5	6	7
-6	-2	0	4	5	7	9	100

Sort (2) Merge Sort and Recursion

- Merge Sort Idea & Operation
- **Recursion**
- Merge Sort Implementation

Recursion

- Function that calls itself during execution -

mergesort calls mergesort that calls mergesort that calls mergesort…

Recursion

- Let's implement factorial function (n! = 1x2x3x…x(n-1)xn)
    - >>> def facto(n: int) -> int:
    - …   ans = 1
    - …   for i in range(1,n+1):
    - …   ans = ans * i
    - …   return ans
- How about this?
    - >>> def facto(n: int) -> int:
    - …   if n == 0:
    - …   return 1
    - …   else:
    - …   return n***facto(n-1)**

Recursion

- Recursion can happen when solving a problem includes solving **subproblems** having the **same structure**
  - Easier to implement (if you can think of this way ever)
  - Results of subproblems can be reused (called dynamic programming, <u>out of scope</u>)

- Structure
  - >>> def facto(n: int) -> int:
  - …   if n == 0:
  - …   return 1
  - …   else:
  - …   return n*facto(n-1)

#Conditional statements check for base cases

#Base case (evaluated without recursive calls)

#Recursive case (evaluated with recursive calls)

Practice 6

Recursion

Problem

- Implement **Fibonacci sequence**, starting from n=1
  - 1,2,3,5,8,13,21,34,55,89 …

- What are
  - (1) the conditional statement,
  - (2) the base case, and
  - (3) the recursive case?

Sort (2) Merge Sort and Recursion

- Merge Sort Idea & Operation
- Recursion
- **Merge Sort Implementation**

Let’s go back to merge sort

Merge Sort – Recursive Call

- def mergeSort(L: list) -> None:
-             mergeSortHelp(L, 0, len(L) – 1)

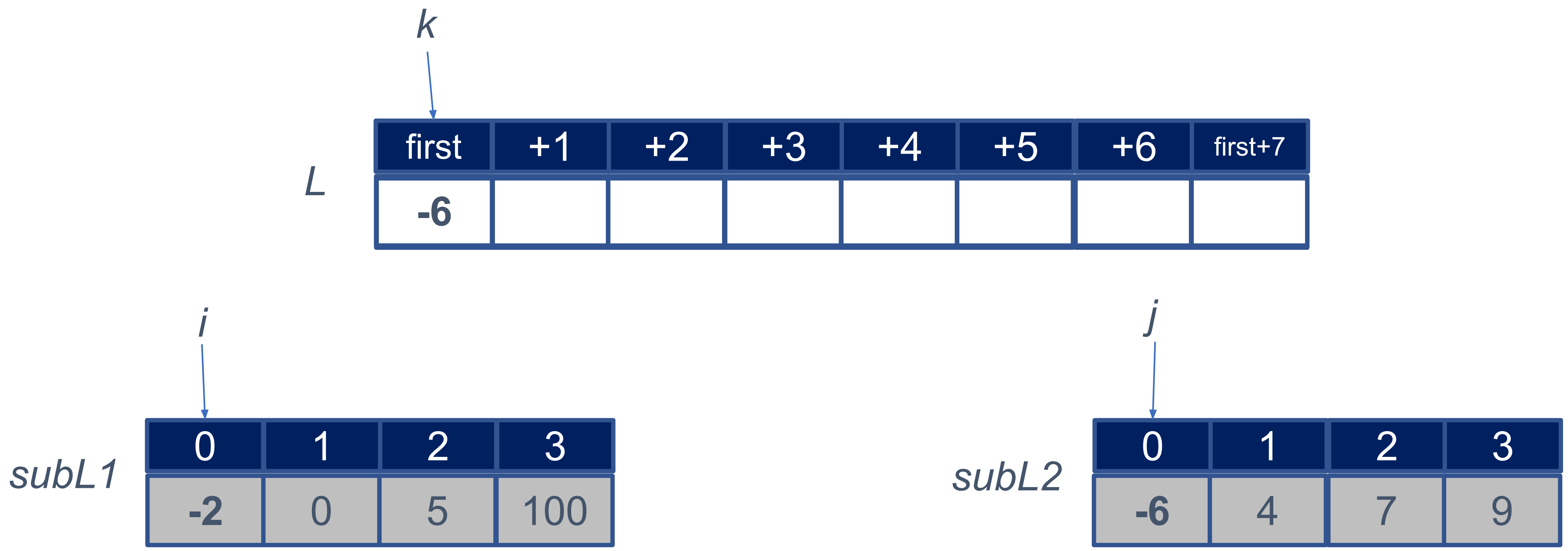
Merge Sort – Recursive Call

- def mergeSortHelp(L: list, first: int, last: int) -> None:
- if first == last: # Conditional statements
- return # Base case
- else:
- mid = first + (last – first) // 2
- **mergeSortHelp(L, first, mid)** # Recursive call for sublist1
- **mergeSortHelp(L, mid+1, last)** # Recursive call for sublist2
- **merge(L, first, mid, last)** # Merge the two (**sorted**) sublists

Parameters to indicate where are two sublists and the whole list

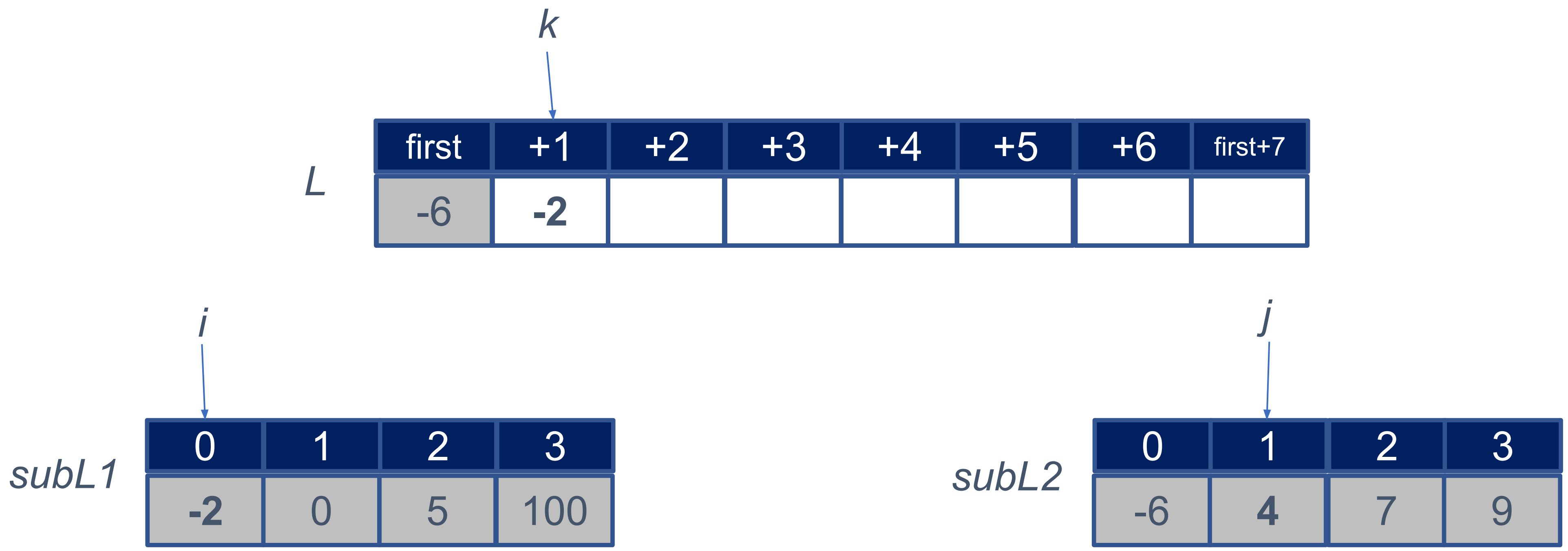
Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



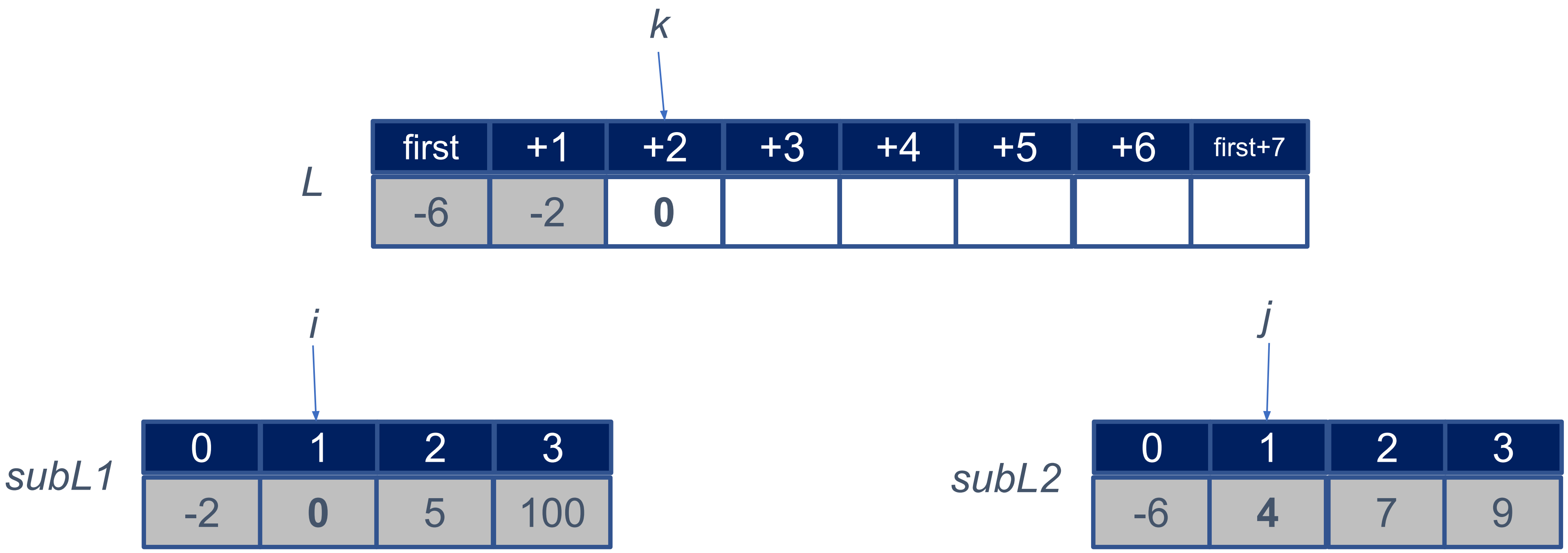
Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



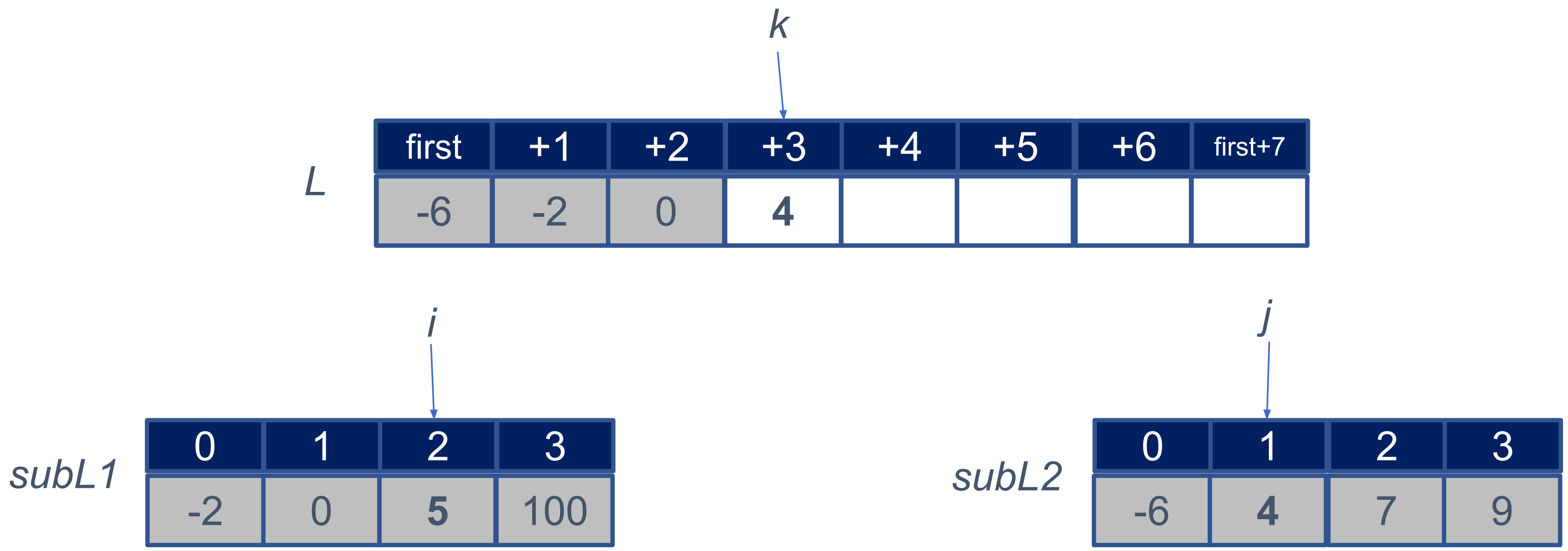
Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



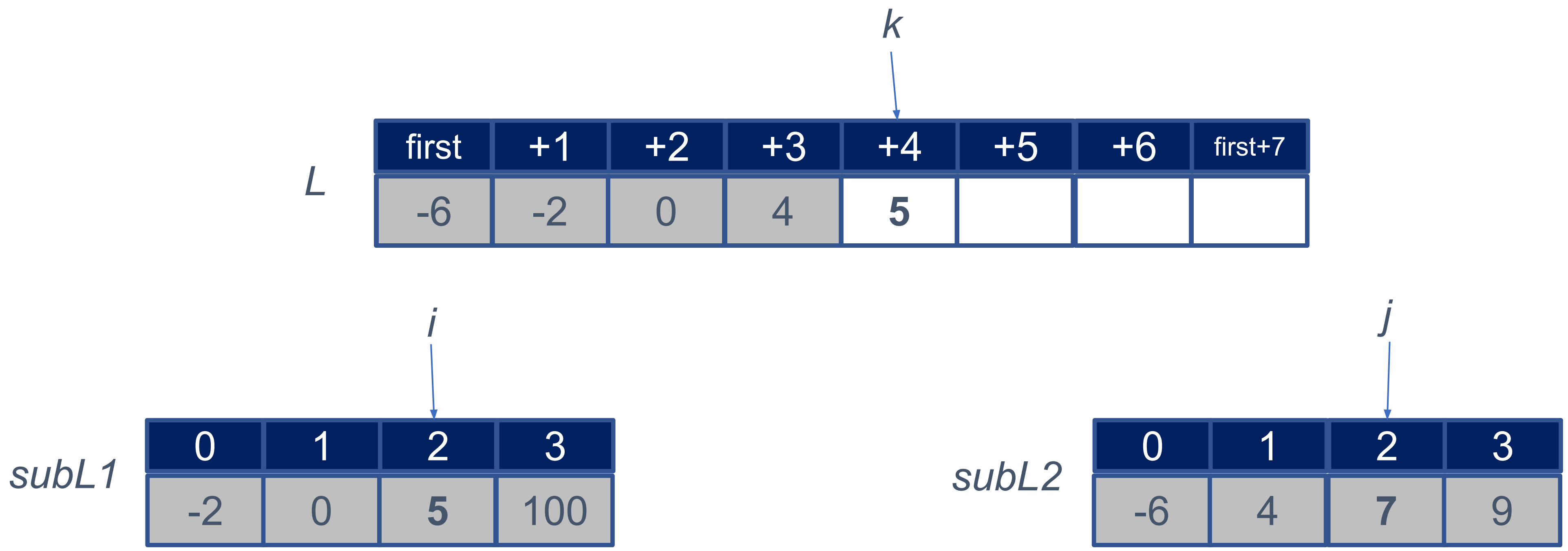
Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



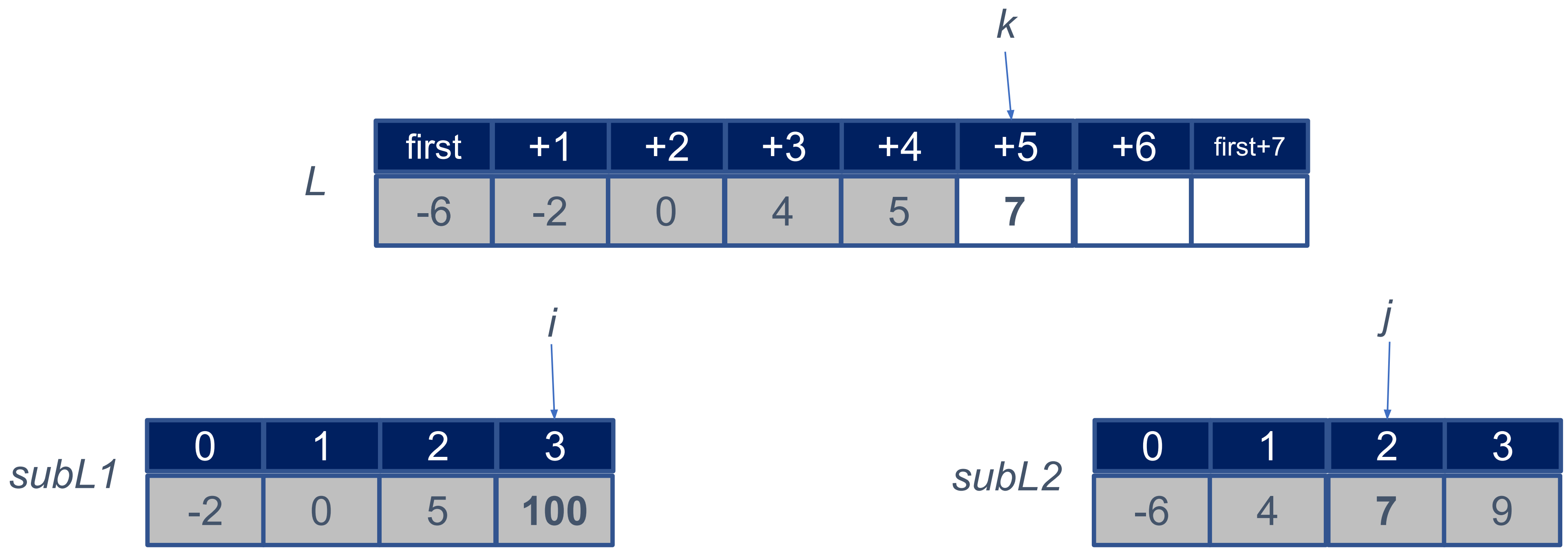
Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



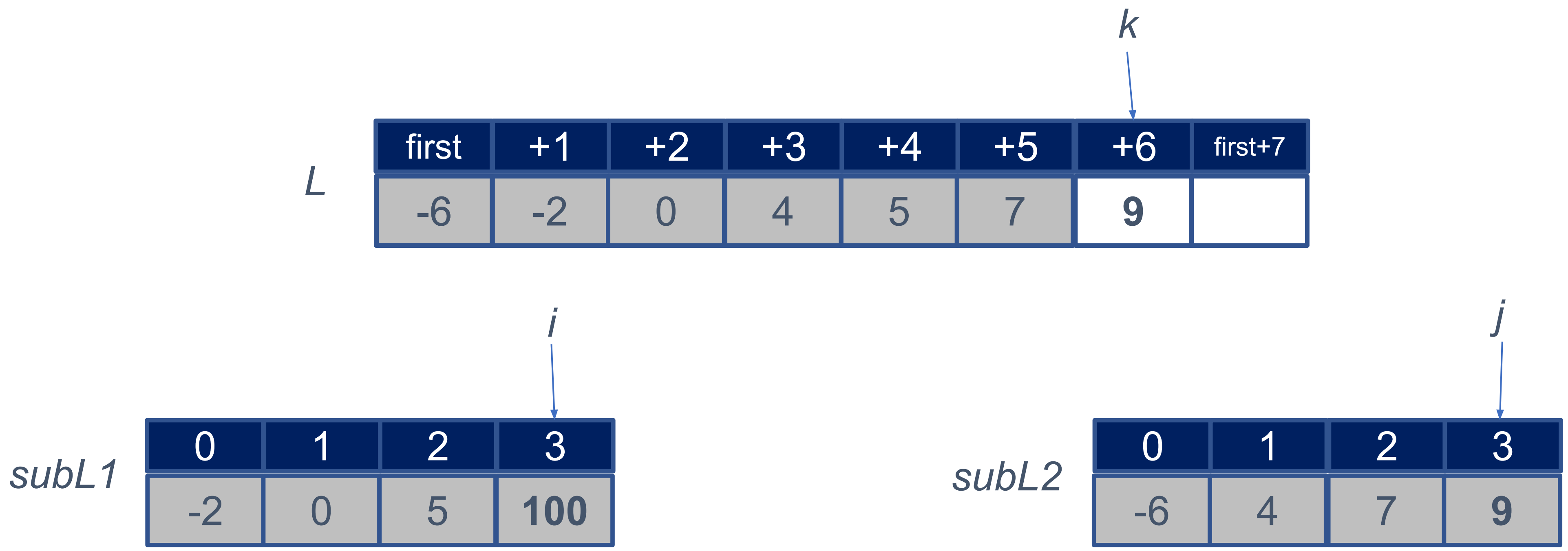
Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



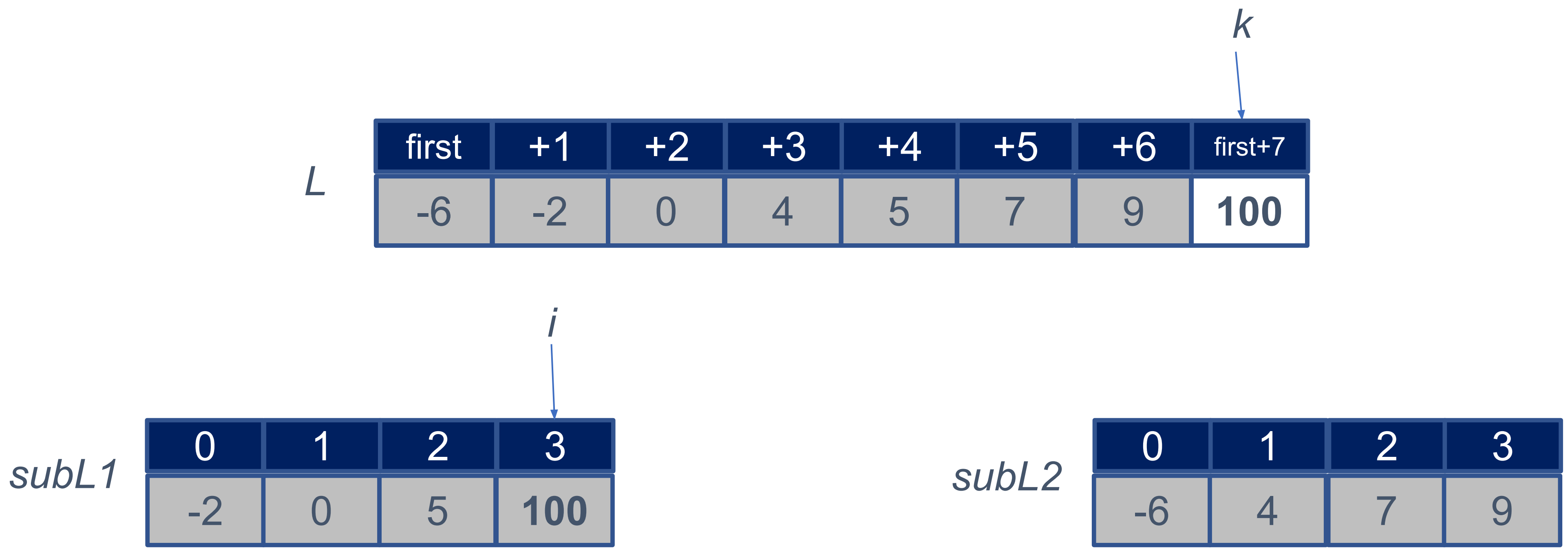
Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]



Merge Sort – Merge Algorithm

- Merge(L, first, mid, last)
  - Memory complexity: O(len(L)) for subL1=L[first:mid+1] and subL2=L[mid+1:last+1]
  - Time complexity of O(**len(L)**), instead of O($\mathbf{len(L)^2}$)

L

first	+1	+2	+3	+4	+5	+6	first+7
-6	-2	0	4	5	7	9	100

subL1

0	1	2	3
-2	0	5	100

subL2

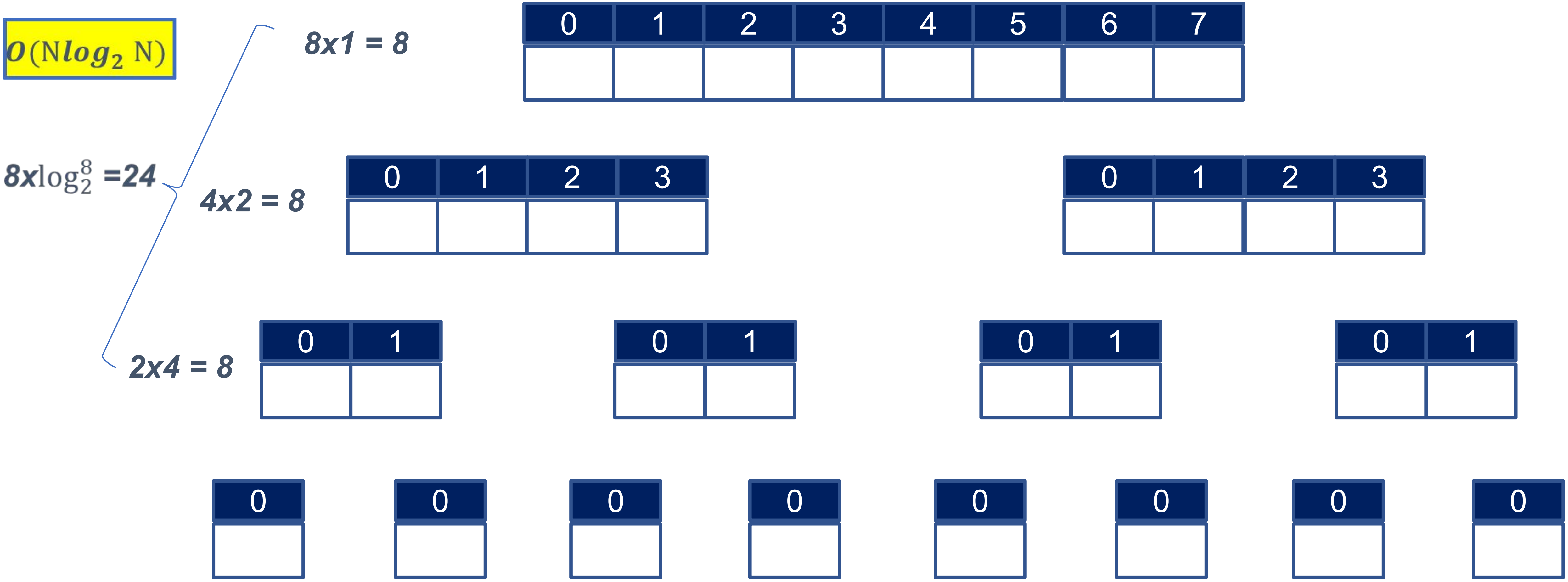
0	1	2	3
-6	4	7	9

Merge Sort – Merge Code

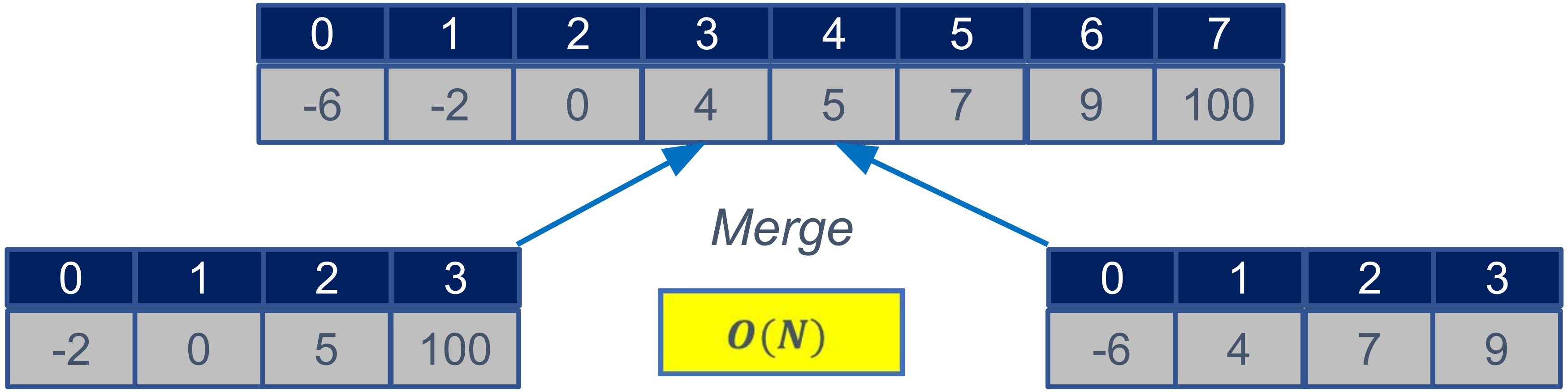
```
>>> def merge(L: list, first: int, mid: int, last: int) -> None:
...         k = first
...         sub1 = L[first:mid+1]
...         sub2 = L[mid+1:last+1]
...         i = j = 0
...         while i < len(sub1) and j < len(sub2):
...             if sub1[i] <= sub2[j]:
...                 L[k] = sub1[i]
...                 i = i+1
...             else:
...                 L[k] = sub2[j]
...                 j = j+1
...             k = k+1
```

```
...         # Checking if any element is left
...         if i < len(sub1):
...             L[k:last+1] = sub1[i:]
...         elif j < len(sub2):
...             L[k:last+1] = sub2[j:]
```

Merge Sort – Time Complexity



Merge Sort – Memory Complexity



Performance Comparison

- Despite messy implementation and somewhat complex logic, Merge Sort is much faster than selection/insertion sort (O(N$\boldsymbol{log_2 N}$) vs. O(N^2))
- Built-in sorting function is still faster, but its complexity **grows similar** to Merge Sort

List Length	Selection sort	Merge Sort	list.sort
1000	148	7	0.3
2000	583	15	0.6
3000	1317	23	0.9
4000	2337	32	1.3
5000	3699	41	1.6
10000	14574	88	3.5

So… What Sort Algorithm is Used for Python?

- **Tim Sort** in 2002 – a hybrid sorting algorithm (merge sort + insertion sort)
  - Divide and conquer like merge sort
  - When a sublist becomes smaller than a threshold, sort the sublist by using insertion sort
    - Insertion sort is faster than merge sort for a small list

- Visualization
  - https://www.youtube.com/watch?v=NVIjHj-lrT4

Thanks!