Review

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

Computing Bootcamp

Recursion

Lecture 11-1

Hyung-Sin Kim

SNU Graduate School of Data Science

Recursion

- Function that calls itself during execution -

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, out of scope)

- Structure
  - >>> def facto(n: int) -> int:
  - …   if n == 0:   *#Conditional statements check for base cases*
  - …   return 1   *#Base case (evaluated without recursive calls)*
  - …   else:
  - …   return n***facto(n-1)**   *#Recursive case (evaluated with recursive calls)*

Example – Fibonacci Sequence

- 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?

Example – Fibonacci Sequence

- Another example: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
  - 1, 2, 3, 5, 8, 13, 21, 34 …
  - >>> def fibonacci(n: int) -> int:
  - …   if n == 1 or n == 2:   *#Conditional statements*
  - …   return n   *#Base case*
  - …   else:
  - …   return **fibonacci(n-1) + fibonacci(n-2)**   *#Recursive case*

Example – Fibonacci Sequence

- Another example: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
  - 1, 2, 3, 5, 8, 13, 21, 34 …
  - >>> def fibonacci(n: int) -> int:
  - …   if n == 1 or n == 2:   *#Conditional statements*
  - …   return n   *#Base case*
  - …   else:
  - …   return **fibonacci(n-1) + fibonacci(n-2)**   *#Recursive case*

- Is Fibonacci implemented correctly?
  - Verify the base case
  - Assuming that fibonacci(n-1) and fibonacci(n-2) are correct, verify if fibonacci(n) is correct

Computing Bootcamp

Merge Sort

Lecture 11-2

Hyung-Sin Kim

SNU Graduate School of Data Science

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

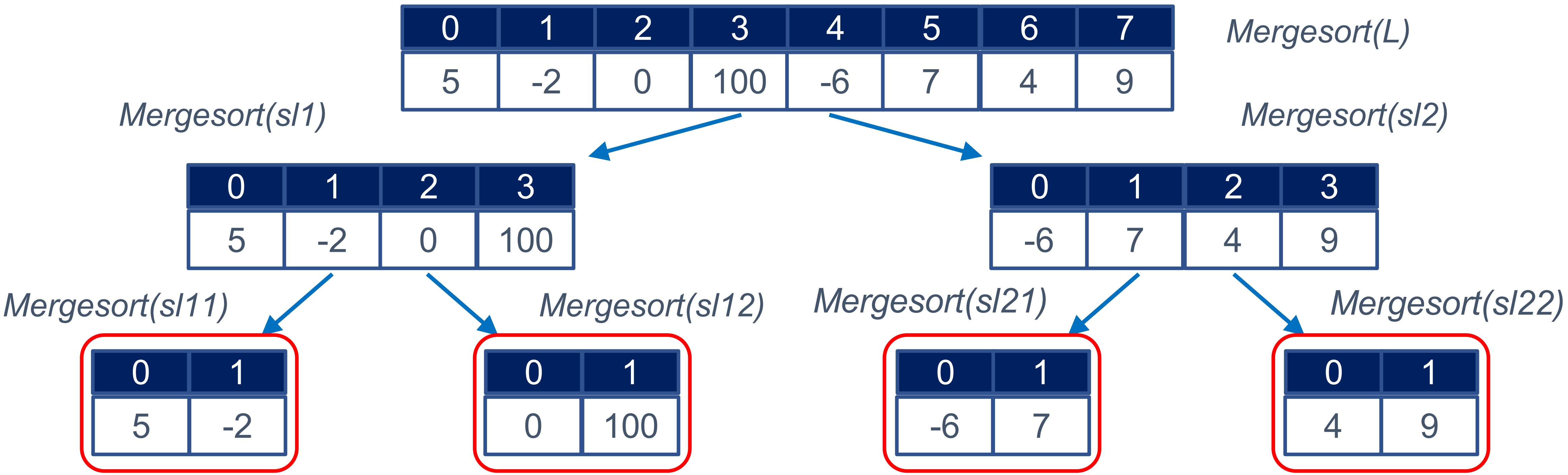
Mergesort(sl1)

0	1	2	3
5	-2	0	100

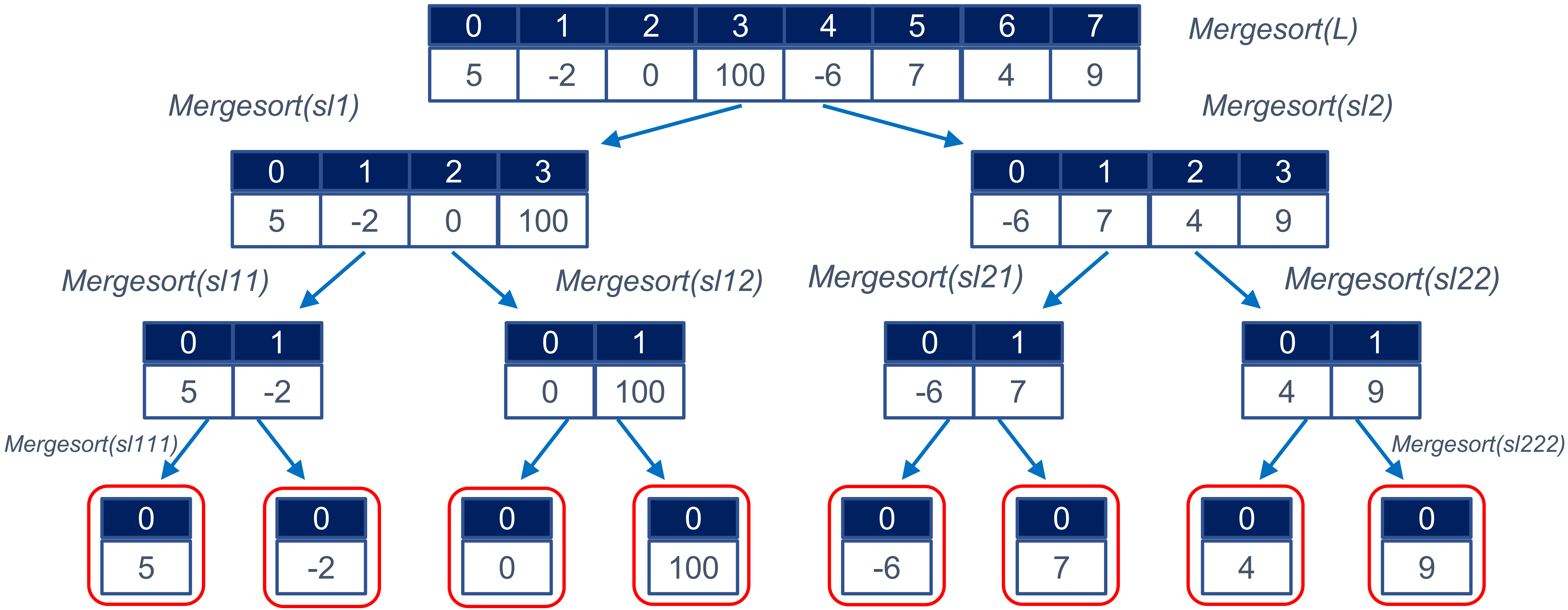
Mergesort(sl2)

0	1	2	3
-6	7	4	9

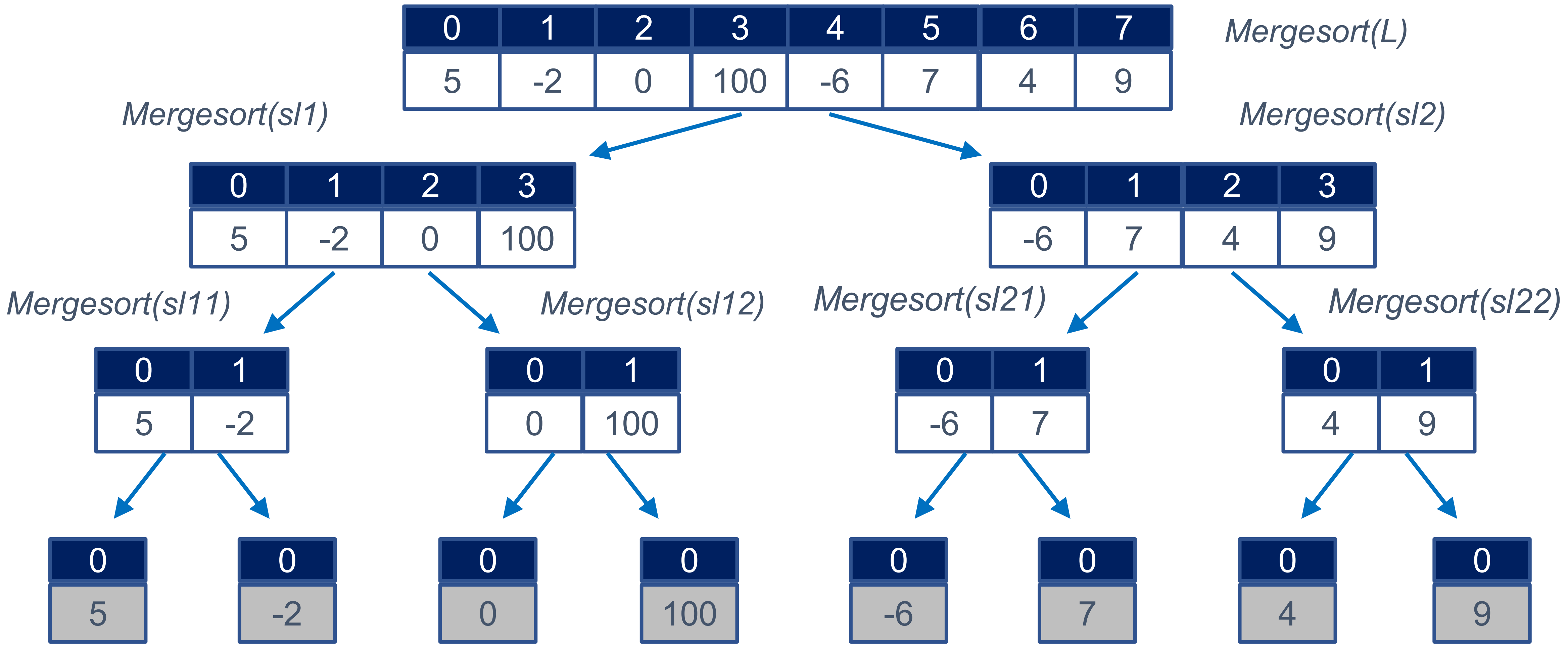
Merge Sort – Operation (Breakdown)



Merge Sort – Operation (Breakdown)

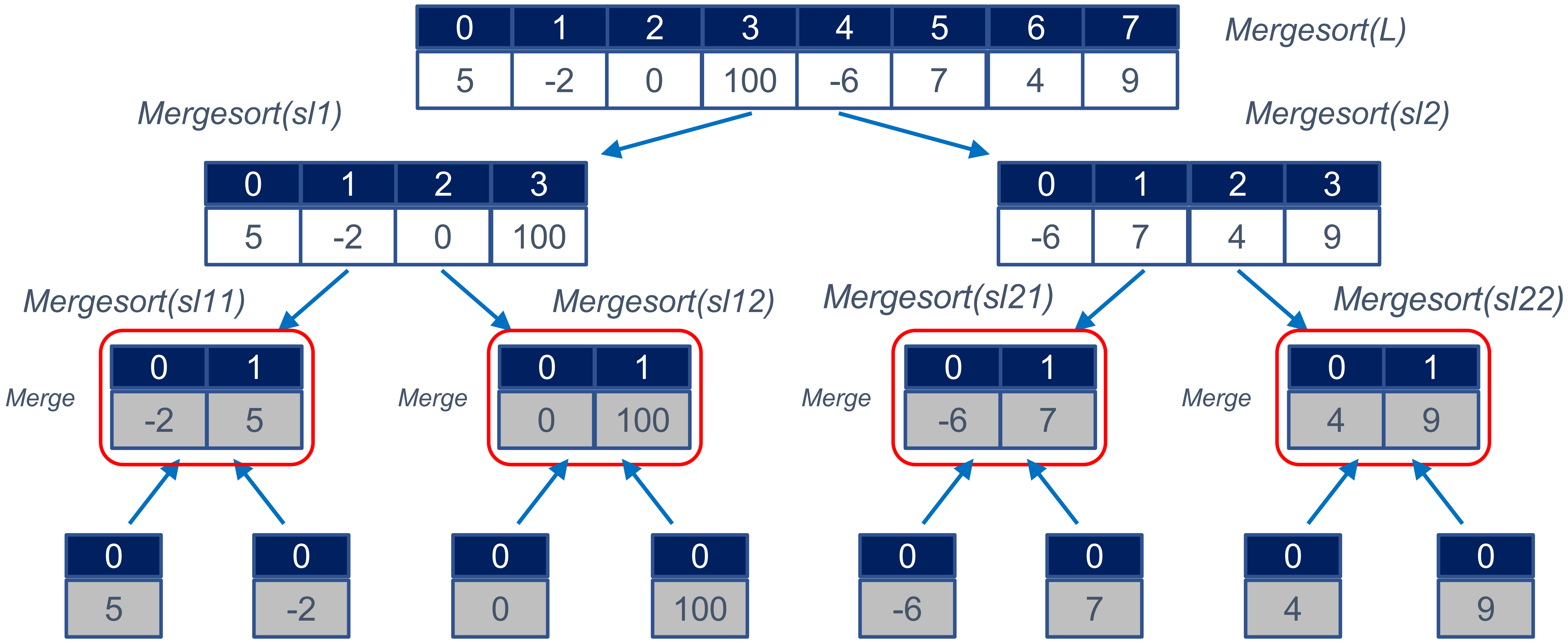


Merge Sort – Operation (Breakdown)

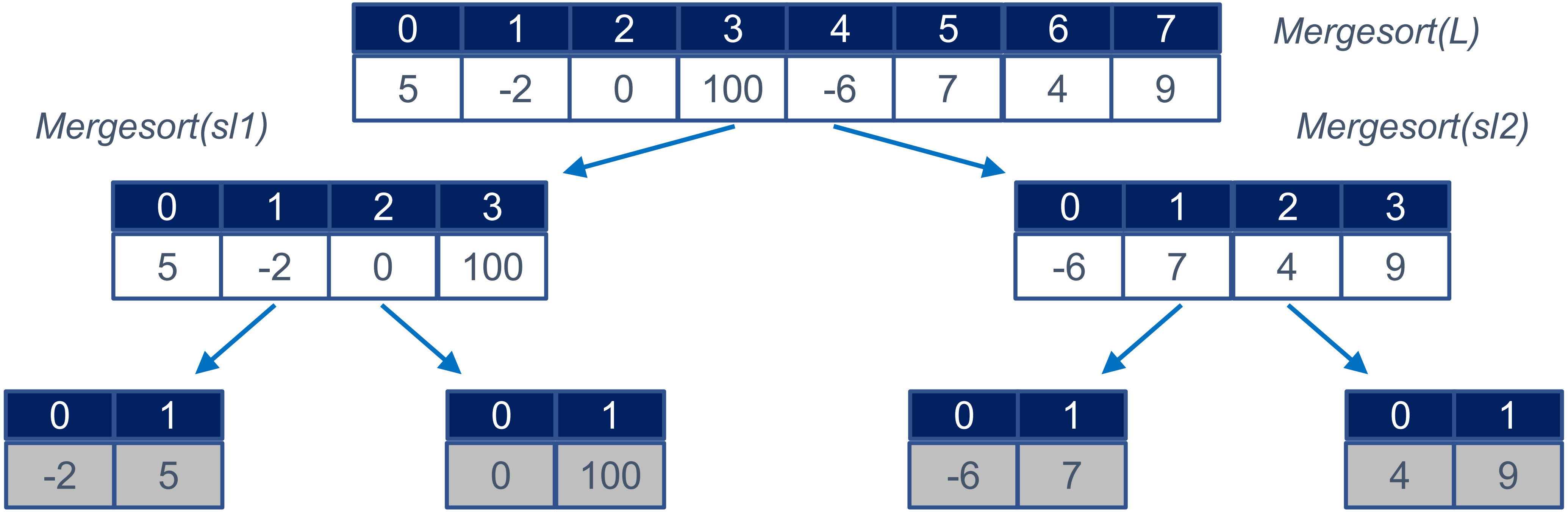


Time to go upwards!

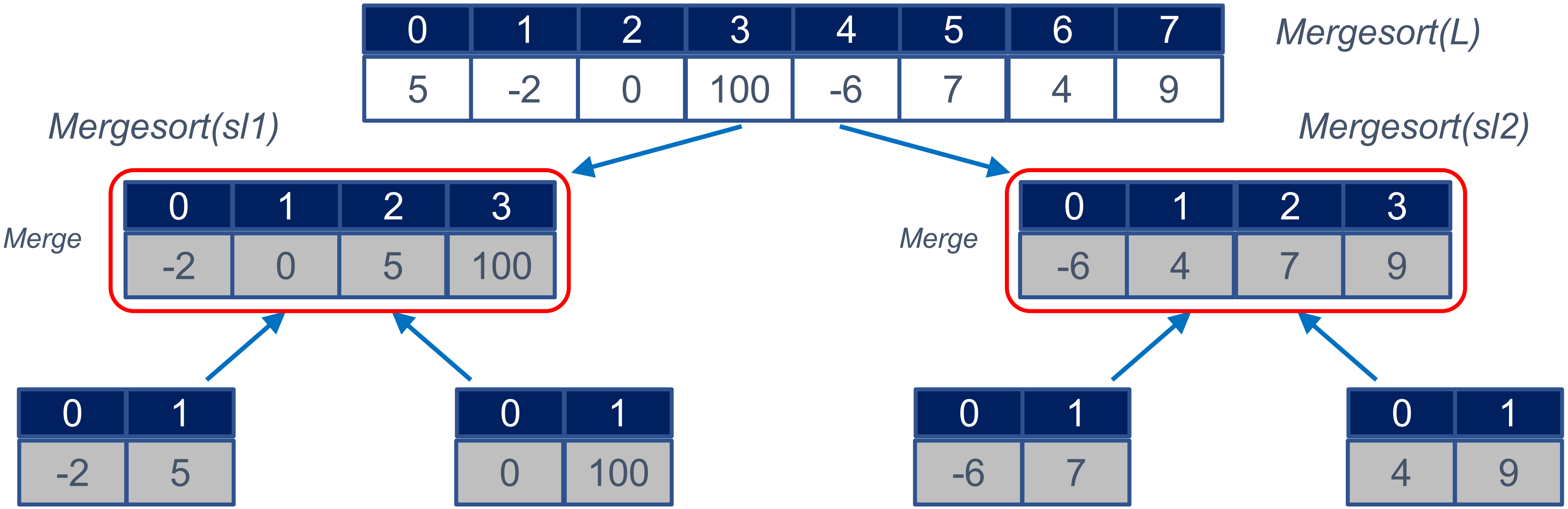
Merge Sort – Operation (Merge)



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)

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

Mergesort(L)

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

Summary

- Mergesort – a sorting algorithm using recursion

Merge Sort Implementation

Lecture 11-3

Hyung-Sin Kim

SNU Graduate School of Data Science

Merge Sort – Recursive Call

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

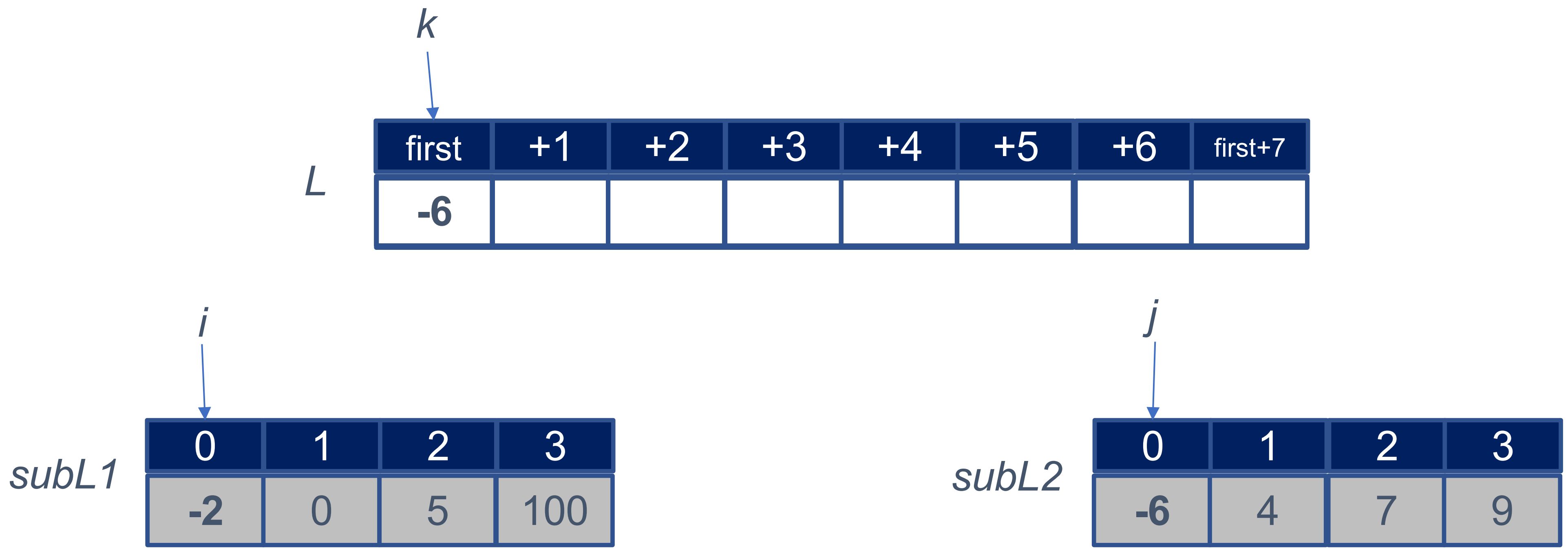
Merge Sort – Recursive Call

- 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

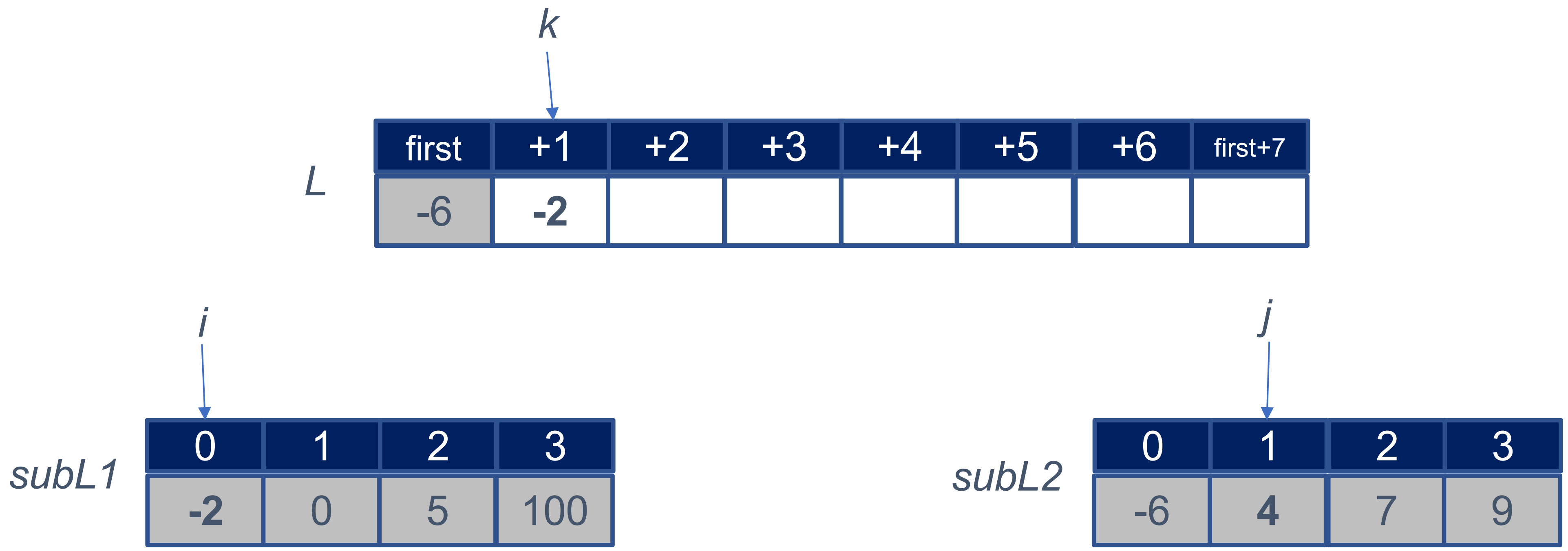
Merge Sort – Merge Algorithm

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



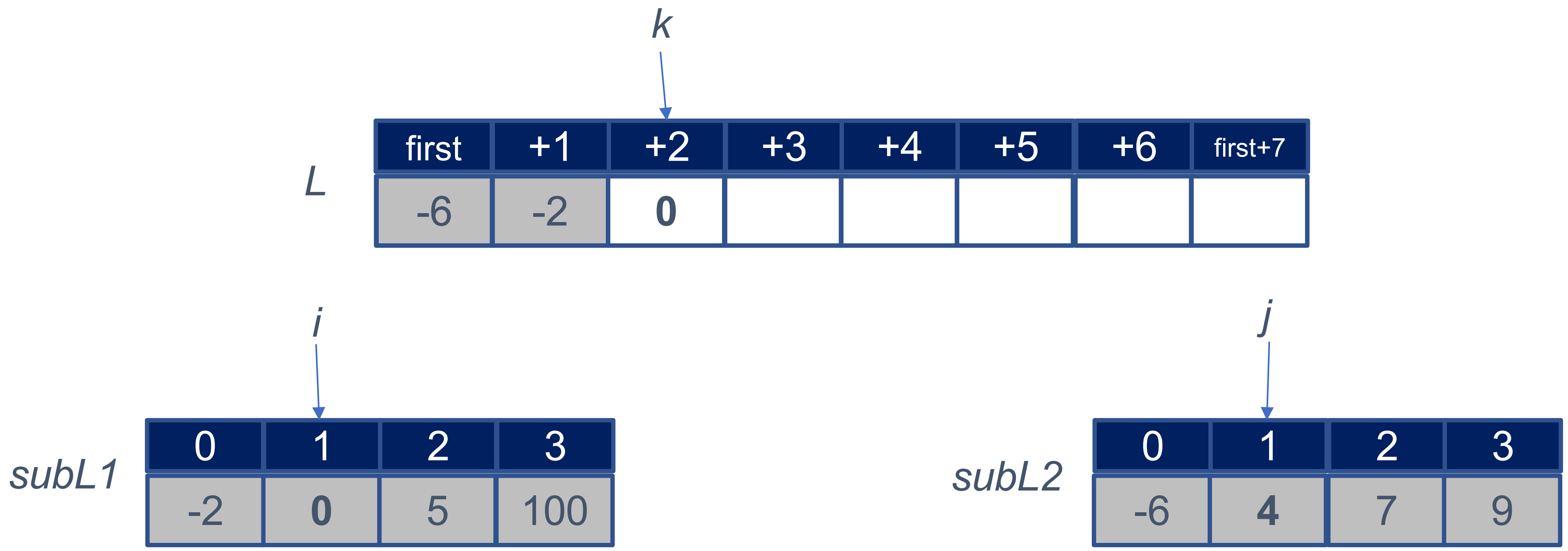
Merge Sort – Merge Algorithm

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



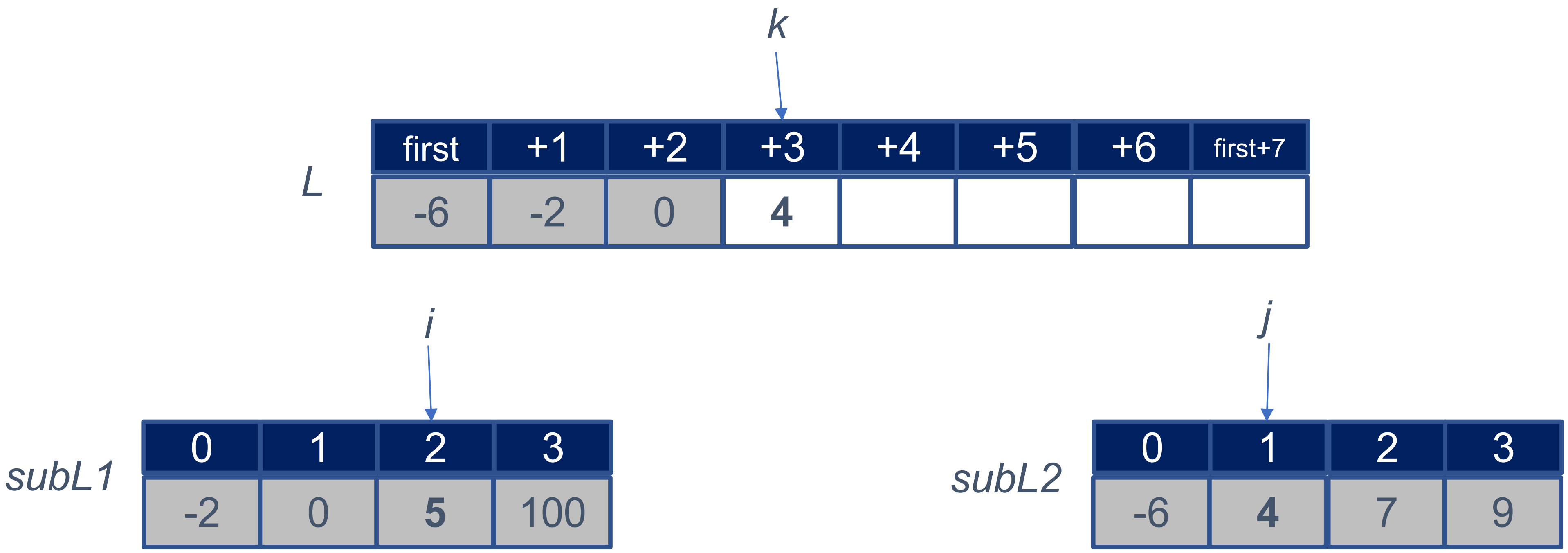
Merge Sort – Merge Algorithm

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



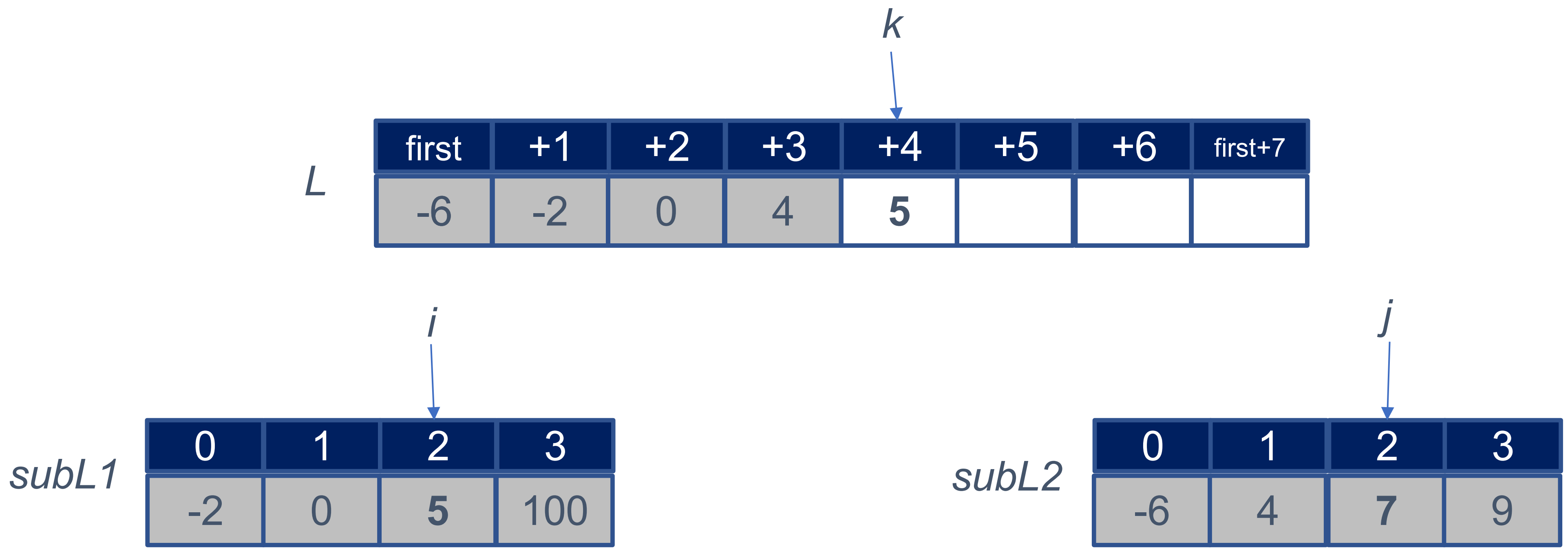
Merge Sort – Merge Algorithm

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



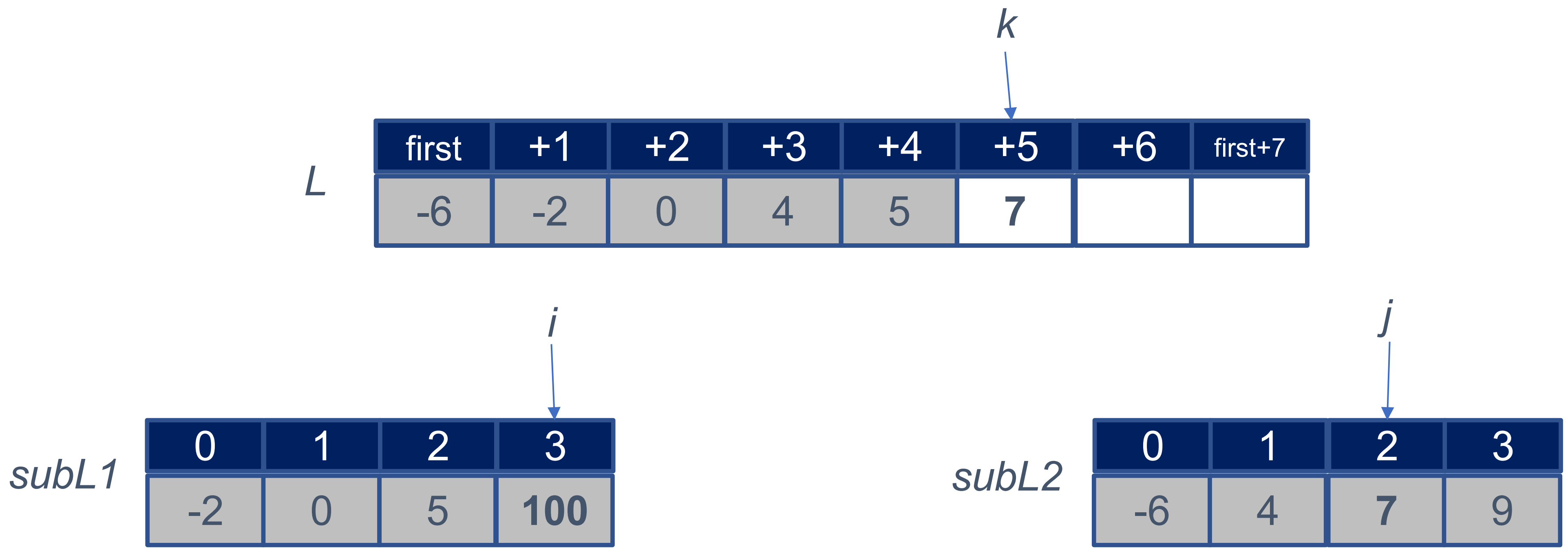
Merge Sort – Merge Algorithm

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



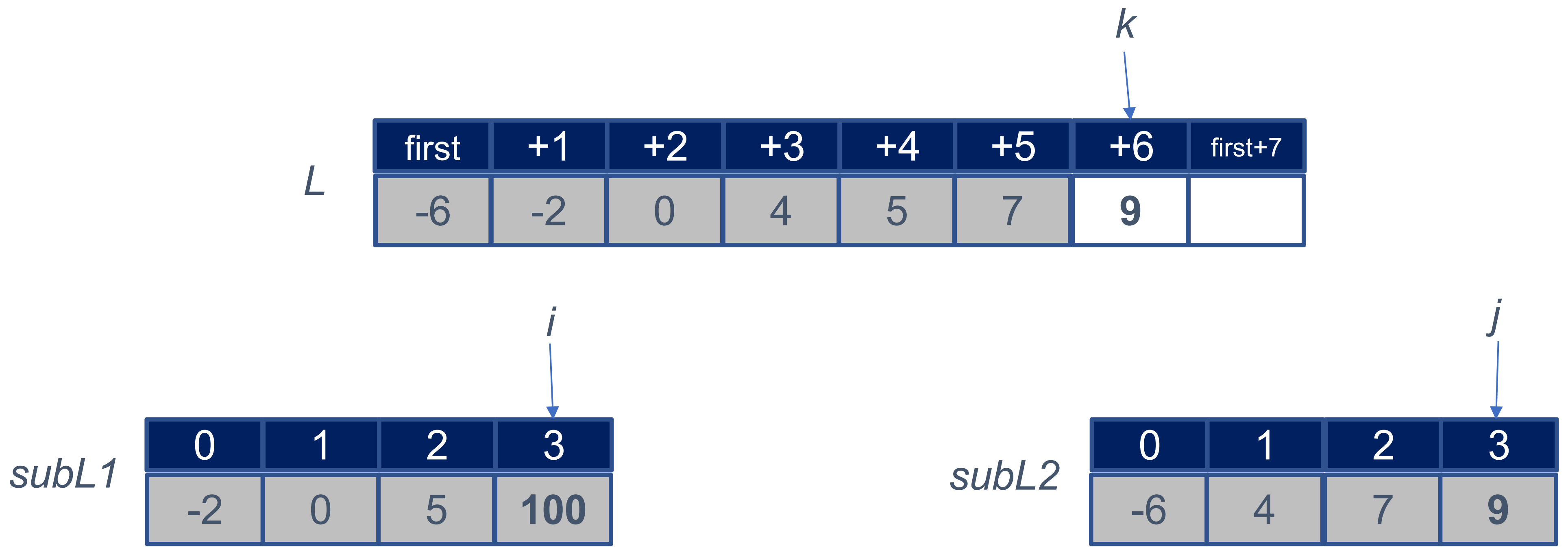
Merge Sort – Merge Algorithm

- 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]

k

L

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

i

subL1

0	1	2	3
-2	0	5	**100**

subL2

0	1	2	3
-6	4	7	9

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(**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

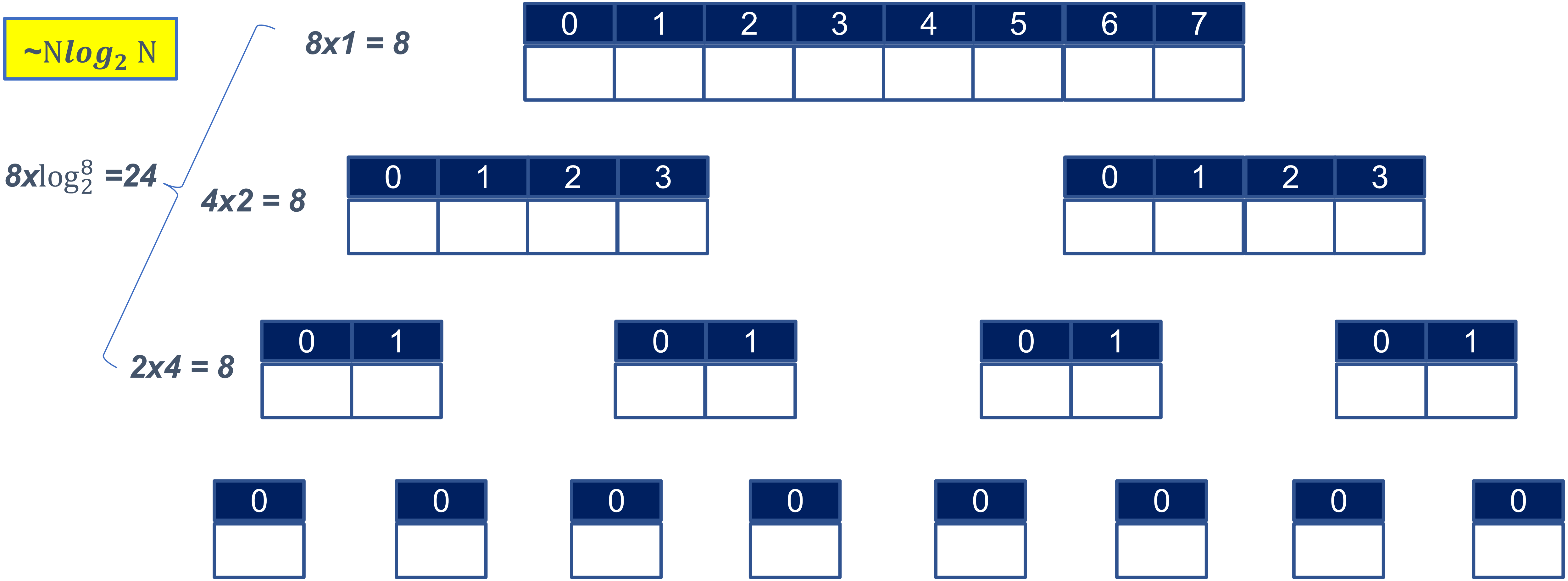
0	1	2	3
-6	4	7	9

Merge Sort – Merge Code

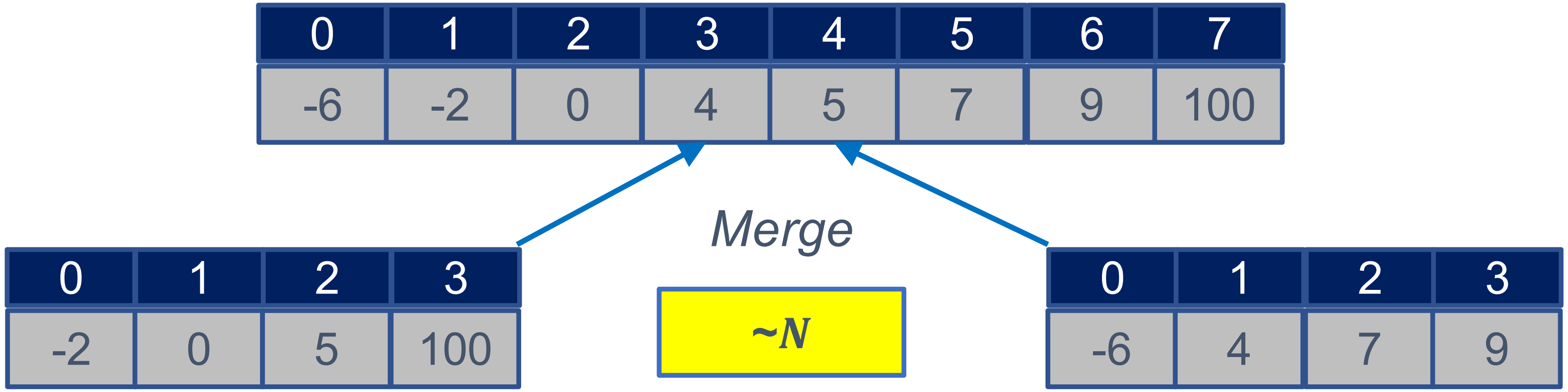
```
>>> def merge(L: list, first: int, mid: int, last: int) -> None:
...         k = first
...         sub1 = L[first:mid+1]
...         sub2 = L[mid+1:last+1]
...         i = j = 0
...         while i < len(sub1) and j < len(sub2):
...             if sub1[i] <= sub2[j]:
...                 L[k] = sub1[i]
...                 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 ($N\boldsymbol{log_2 N}$ vs. 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

Summary

- Mergesort
  - Divide and recursive calls
  - Merge sorted sublists

- Time complexity O(NlogN)
  - O(logN) levels for recursion
  - O(N) for merging at each level

- Memory complexity O(N)
  - For copying sublists

Thanks!