Luiss
Libera Università Internazionale degli Studi Sociali Guido Carli

Algorithms A.Y. 2022/2023

Lab - Merge Sort

Irene Finocchi, Flavio Giorgi, Bardh Prenkaj
finocchi@luiss.it, fgiorgi@luiss.it, bprenkaj@luiss.it©

17 February 2023

courtesy of: *Andrea Coletta*

LUISS

Dipartimento di Impresa e Management

Sorting… Again

As you noticed sorting data structures is **very** important.

Besides, when we sort we have to be efficient!

A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

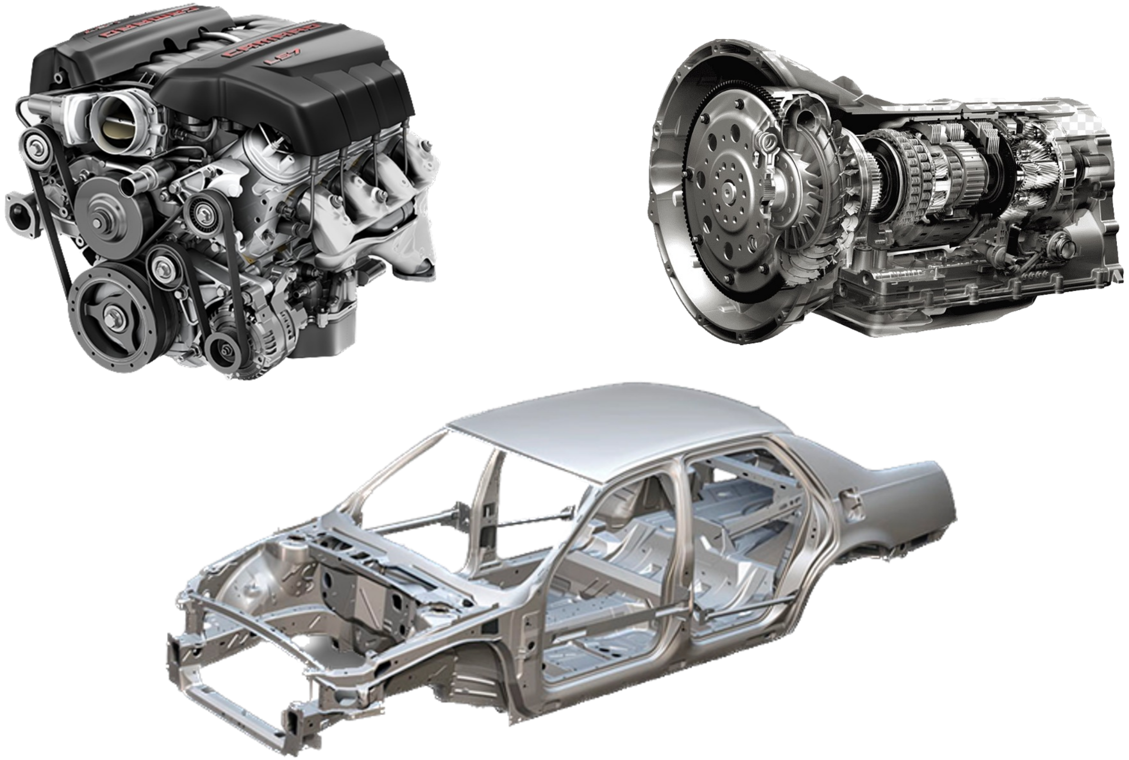
For example, if you have to build a car from scratch you start decomposing the problem into smaller problems:

A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

For example, if you have to build a car from scratch you start decomposing the problem into smaller problems:

1. Assemble the engine
2. Assemble the transmission
3. Assemble the car body
4. etc…



A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

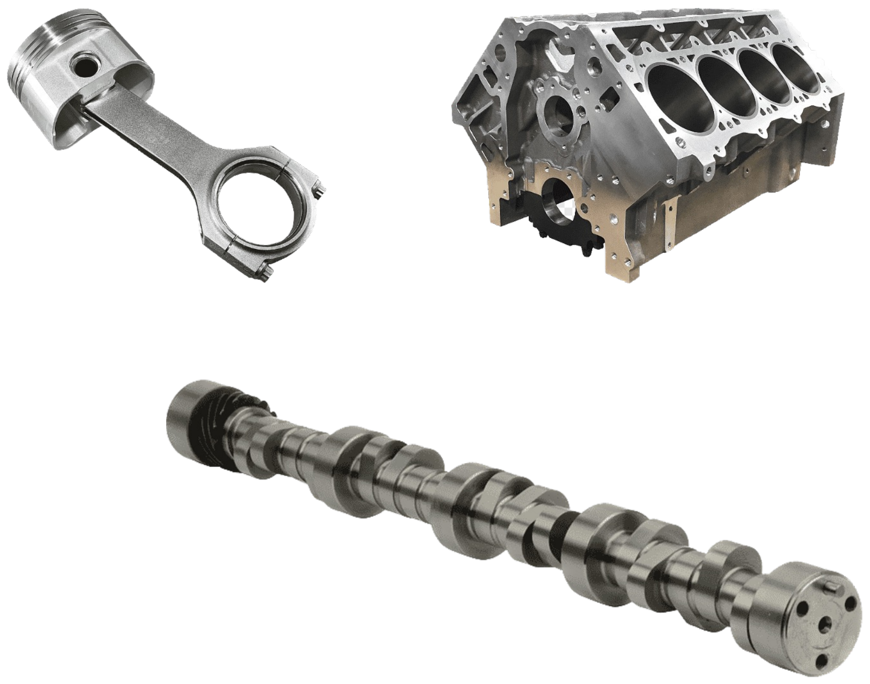
In turn each one of these problems can be split into smaller problems:

A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

In turn each one of these problems can be split into smaller problems:

1. Assemble the engine
    a. make the pistons
    b. make the engine block
    c. make the camshaft
    d. etc…



A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

And so on…

A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

Once you have all the components you can start assembling them to actually get the car!

So you start putting the engine parts together.

A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

Once you have all the components you can start assembling them to actually get the car!

So you start putting the engine parts together.

Then you put the engine within the car body.

A new paradigm - Divide and Conquer *(and Combine)*

A very efficient way to solve complex problems is to **divide** them into smaller pieces.

Once you have all the components you can start assembling them to actually get the car!

So you start putting the engine parts together.

Then you put the engine within the car body.

Etc…

Until you will have a car!

A new paradigm - Divide and Conquer *(and Combine)*

This paradigm is used also to solve more *“abstract”* problems like **sorting**.

A new paradigm - Divide and Conquer *(and Combine)*

This paradigm is used also to solve more *“abstract”* problems like **sorting**.

Today we explore a sorting algorithm called **Merge Sort** that is based on this paradigm!

Merge Sort: the idea

Algorithms based on divide-conquer-combine paradigm decompose **large and complex** problems into **small and simple** sub-parts.

Each sub-part in turn is solved separately, and the solutions are recombined to solve the original instance.

Steps:

Merge Sort: the idea

Algorithms based on divide-conquer-combine paradigm decompose **large and complex** problems into **small and simple** sub-parts.

Each sub-part in turn is solved separately, and the solutions are recombined to solve the original instance.

Steps:

1. **Divide**: decompose a large and complex problem into smaller and simple subproblems.

Merge Sort: the idea

Algorithms based on divide-conquer-combine paradigm decompose **large and complex** problems into **small and simple** sub-parts.

Each sub-part in turn is solved separately, and the solutions are recombined to solve the original instance.

Steps:

1. **Divide**: decompose a large and complex problem into smaller and simple subproblems.
2. **Conquer**: use a procedure to solve each one of the smaller subproblems.

Merge Sort: the idea

Algorithms based on divide-conquer-combine paradigm decompose **large and complex** problems into **small and simple** sub-parts.

Each sub-part in turn is solved separately, and the solutions are recombined to solve the original instance.

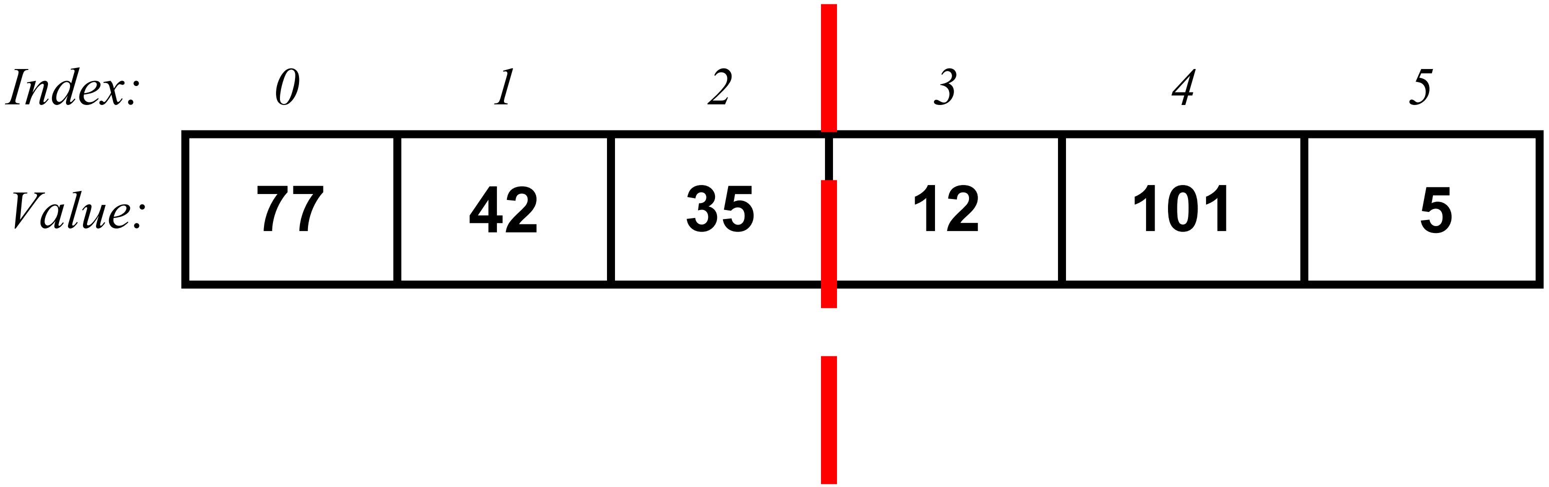
Steps:

1. **Divide**: decompose a large and complex problem into smaller and simple subproblems.
2. **Conquer**: use a procedure to solve each one of the smaller subproblems.
3. **Combine**: join the solutions returned by the procedure to solve the original problem.

Merge Sort: an example

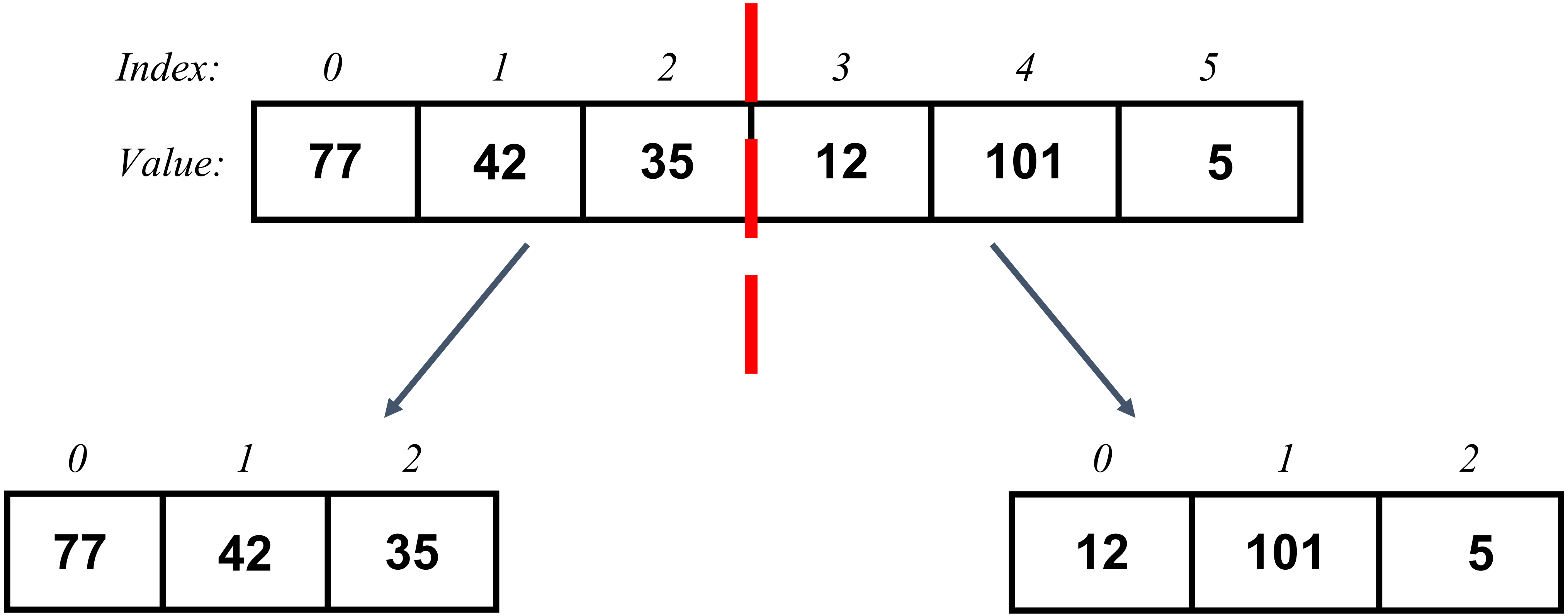
Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

Merge Sort: an example



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

Merge Sort: an example

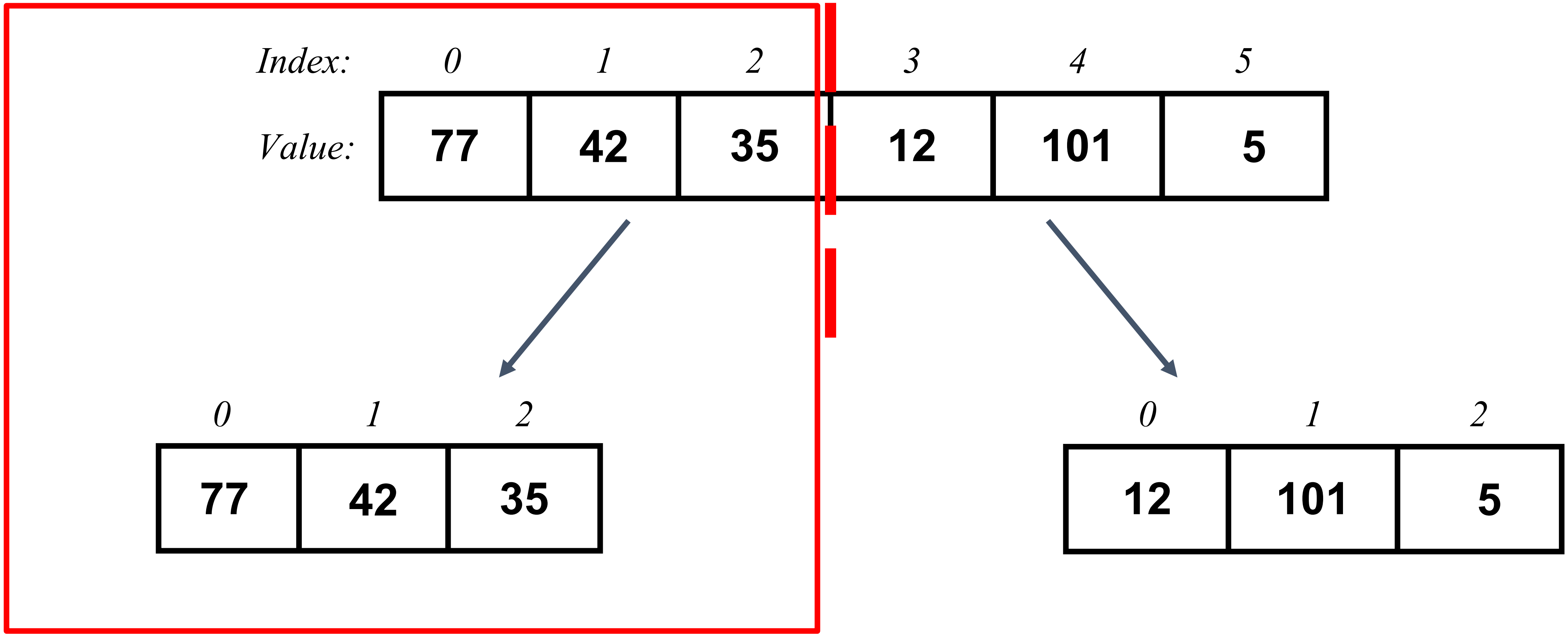


Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

0	*1*	*2*
77	**42**	**35**

0	*1*	*2*
12	**101**	**5**

Merge Sort: an example



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

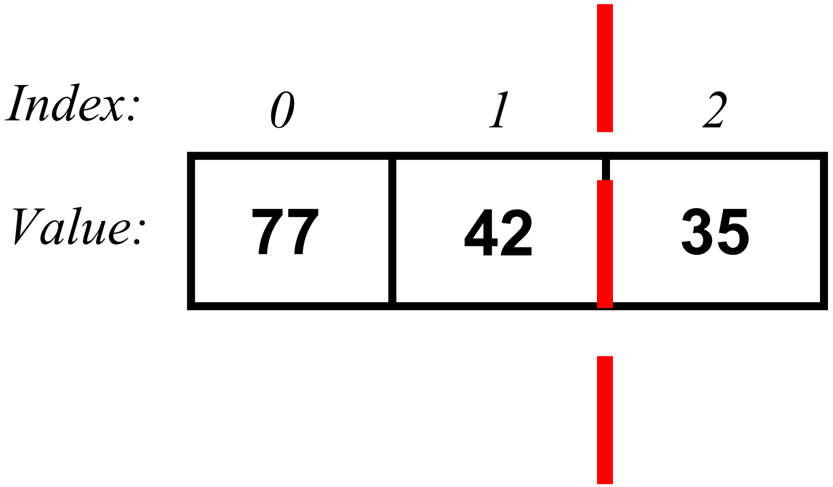
0	*1*	*2*
77	**42**	**35**

0	*1*	*2*
12	**101**	**5**

Merge Sort: an example

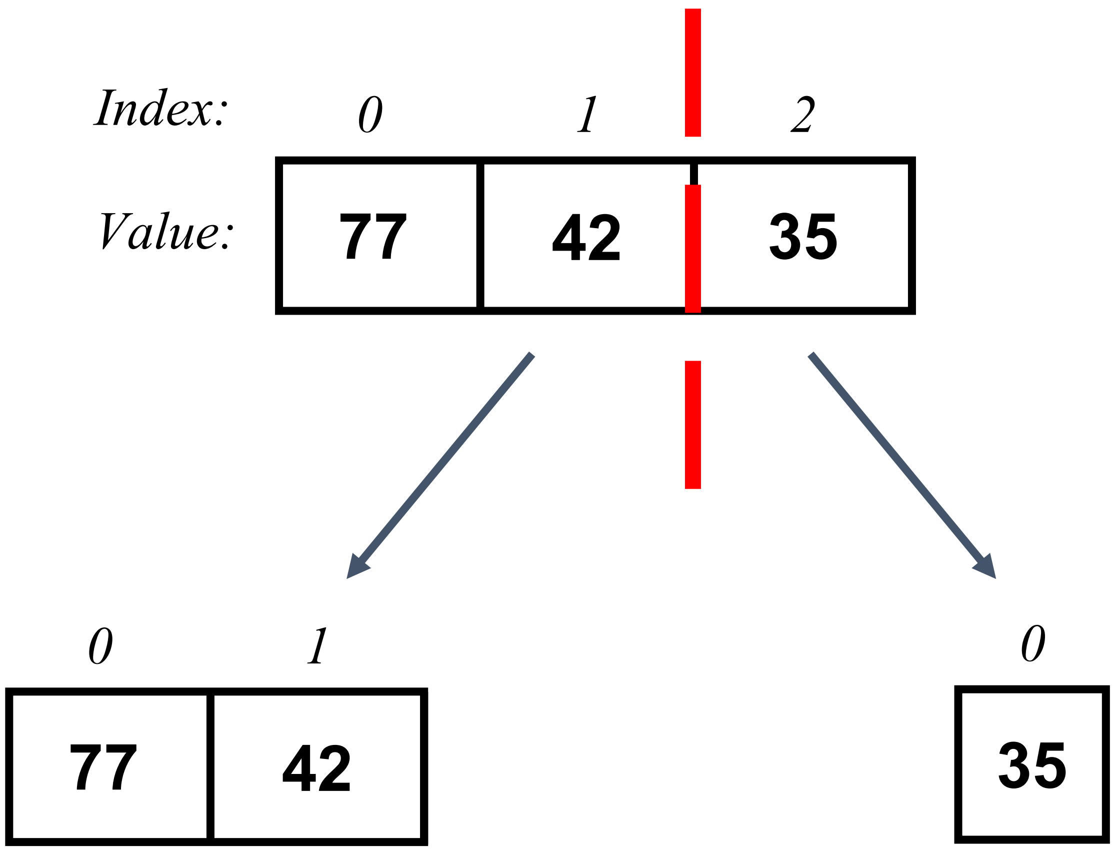
Index:	*0*	*1*	*2*
Value:	**77**	**42**	**35**

Merge Sort: an example



Index:	*0*	*1*	*2*
Value:	**77**	**42**	**35**

Merge Sort: an example



Index: 0 1 2

Value: 77 42 35

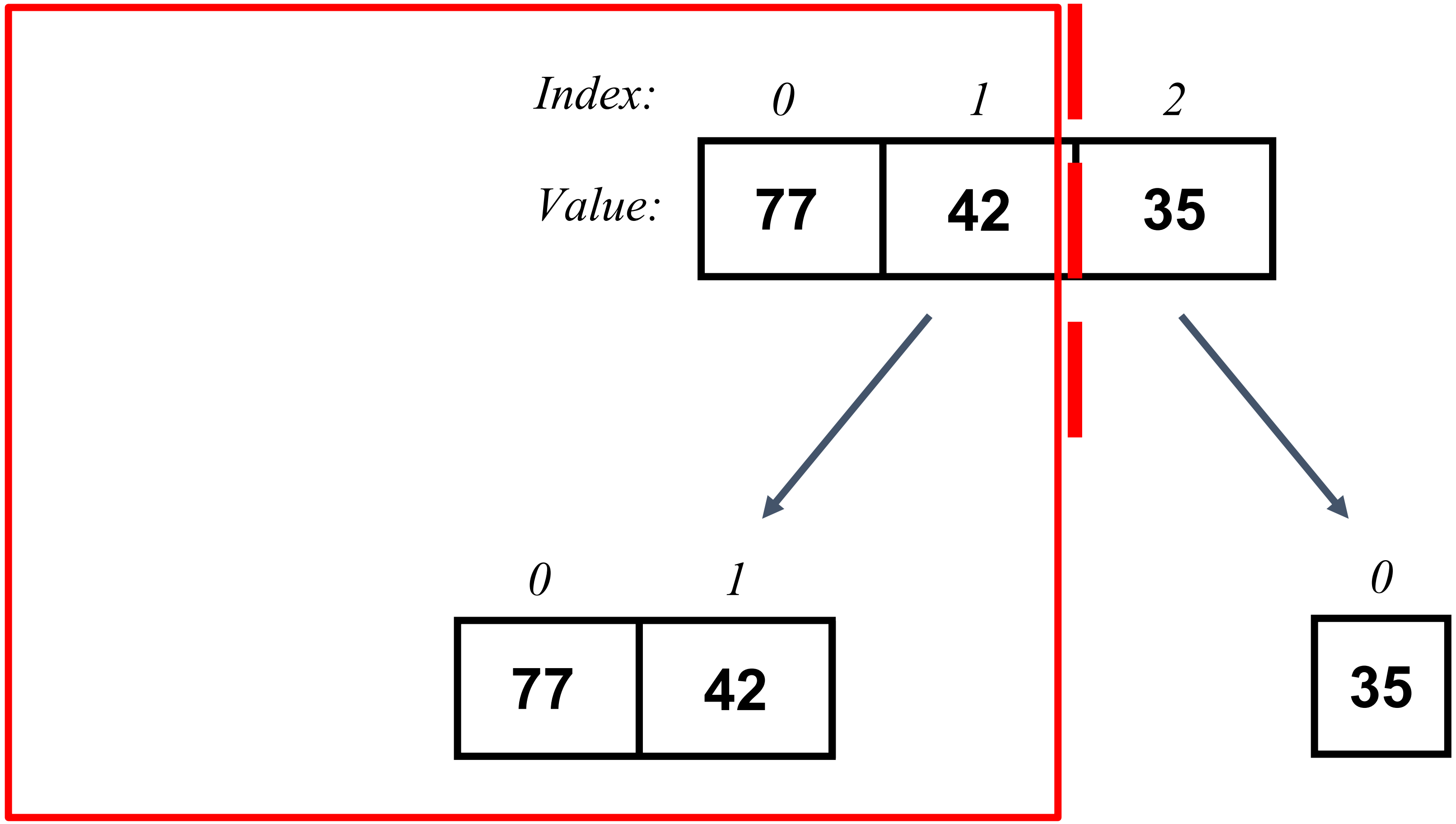
0 1

77 42

0

35

Merge Sort: an example



Index:	0	1	2
Value:	77	42	35

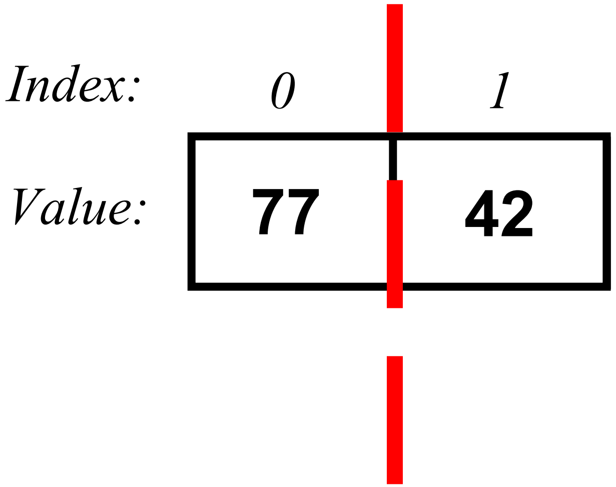
0	1
77	42

0
35

Merge Sort: an example

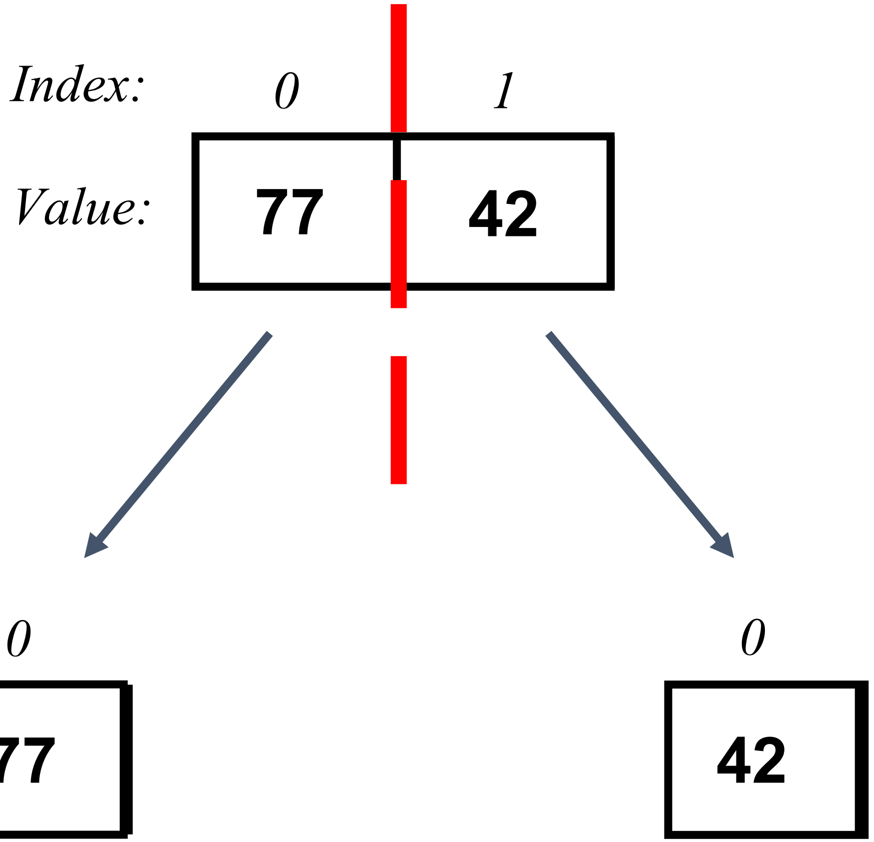
Index:	*0*	*1*
Value:	**77**	**42**

Merge Sort: an example



Index:	*0*	*1*
Value:	**77**	**42**

Merge Sort: an example



Index: 0 1

Value: 77 42

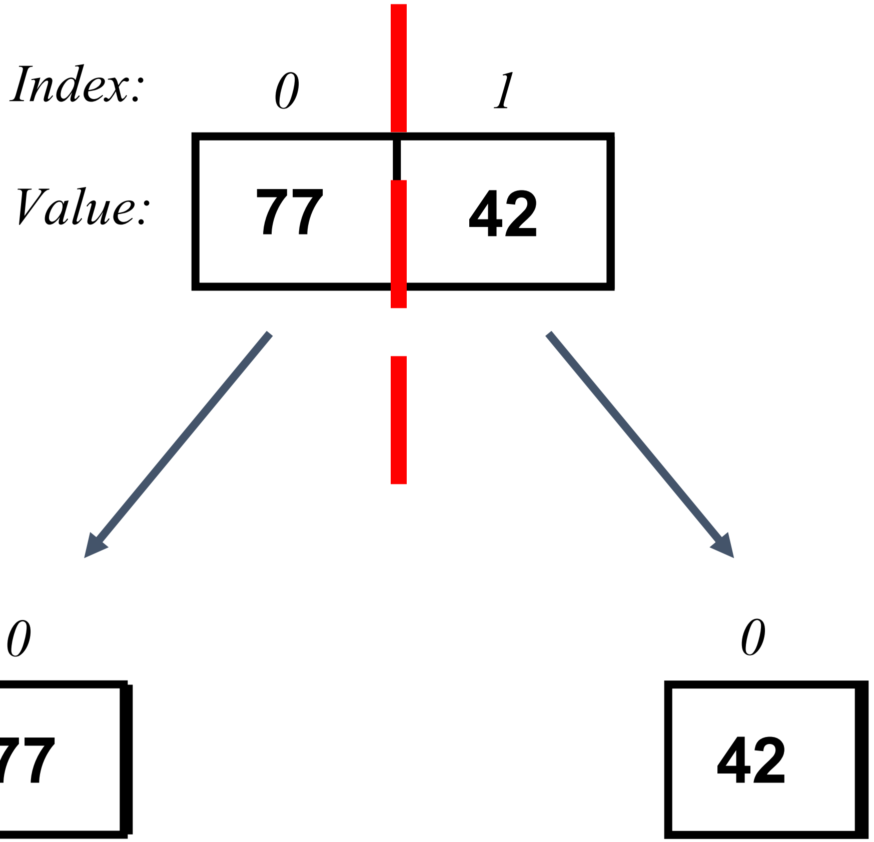
0

77

0

42

Merge Sort: an example



Index: *0* *1*

Value: **77** **42**

0 **77**

0 **42**

Here we reached the simplest possible case!
We cannot divide the list again!

Merge Sort: an example

0
77

0
42

Now we have to join the results!

Merge Sort: an example

0

77

0

42

Now we have to join the results!

How?

Merge Sort: an example

Now we have to join the results!

How?
We can check which element is the smaller one between the two and put it into the position 0 while the other one into position 1

Merge Sort: an example

0

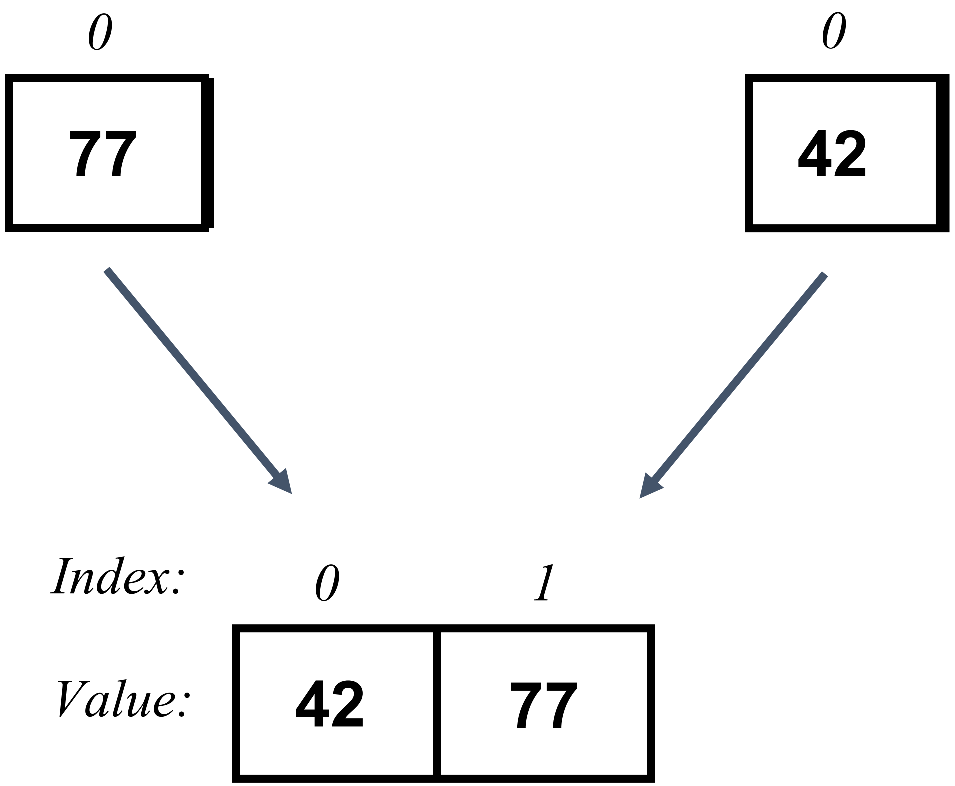
77

0

42

Since this case is trivial we are going to see the procedure used to merge during the next join step!

Merge Sort: an example



0

77

0

42

Index:	*0*	*1*
Value:	**42**	**77**

Merge Sort: an example

0	*1*
42	**77**

0
35

Again we have to join the results!

Merge Sort: an example

0	1
42	77

0
35

Again we have to join the results!

But how can we do that in linear time?

Merge Sort: the merge procedure

Before continuing with the Merge Sort execution we see a brief explanation about the **merge** procedure.

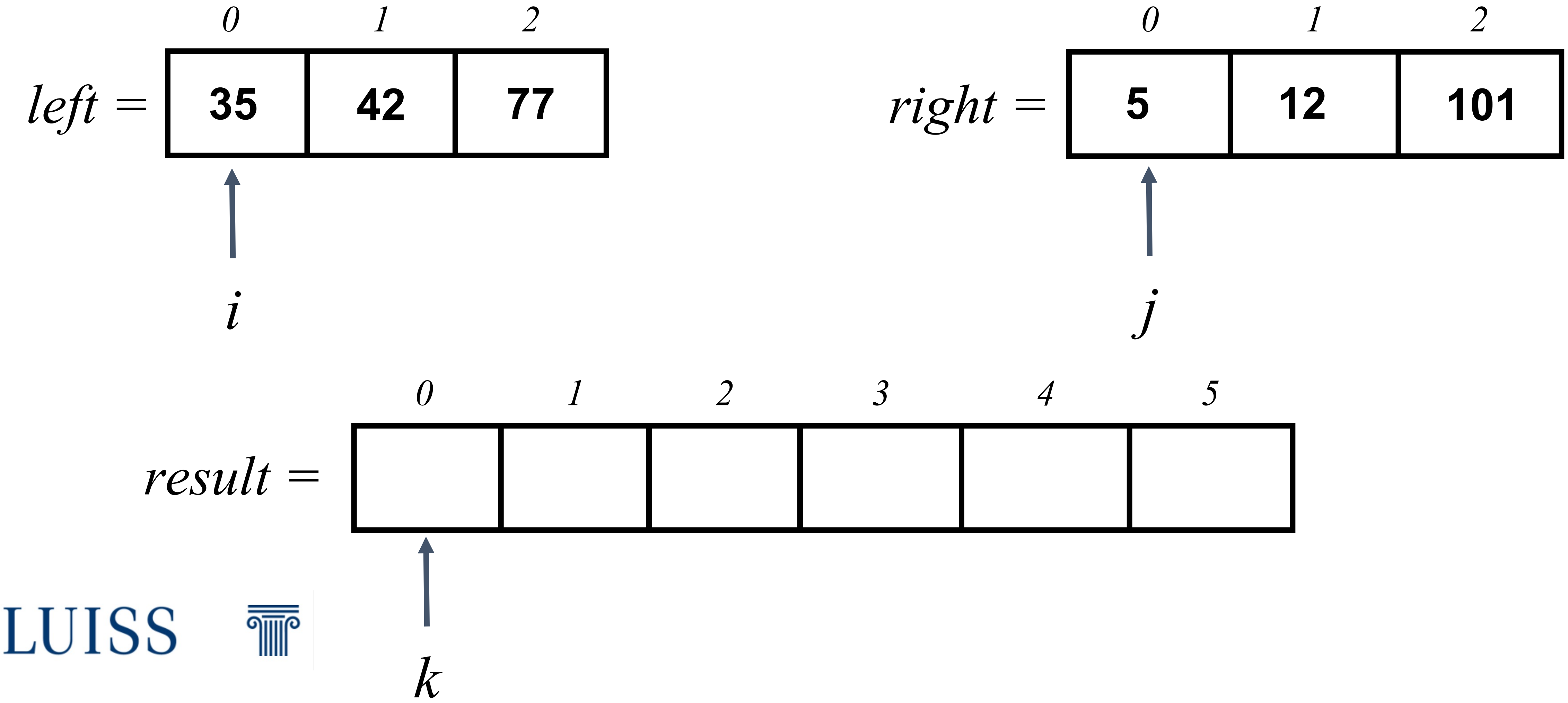
Merge Sort: the merge procedure

Before continuing with the Merge Sort execution we see a brief explanation about the **merge** procedure.

This procedure **joins** two **ordered** lists into a single **ordered** list!

Merge Sort: the merge procedure

Let’s declare an empty vector *result* that can contain the elements of both the sub-vectors!



$left =$

0	1	2
35	**42**	**77**

$right =$

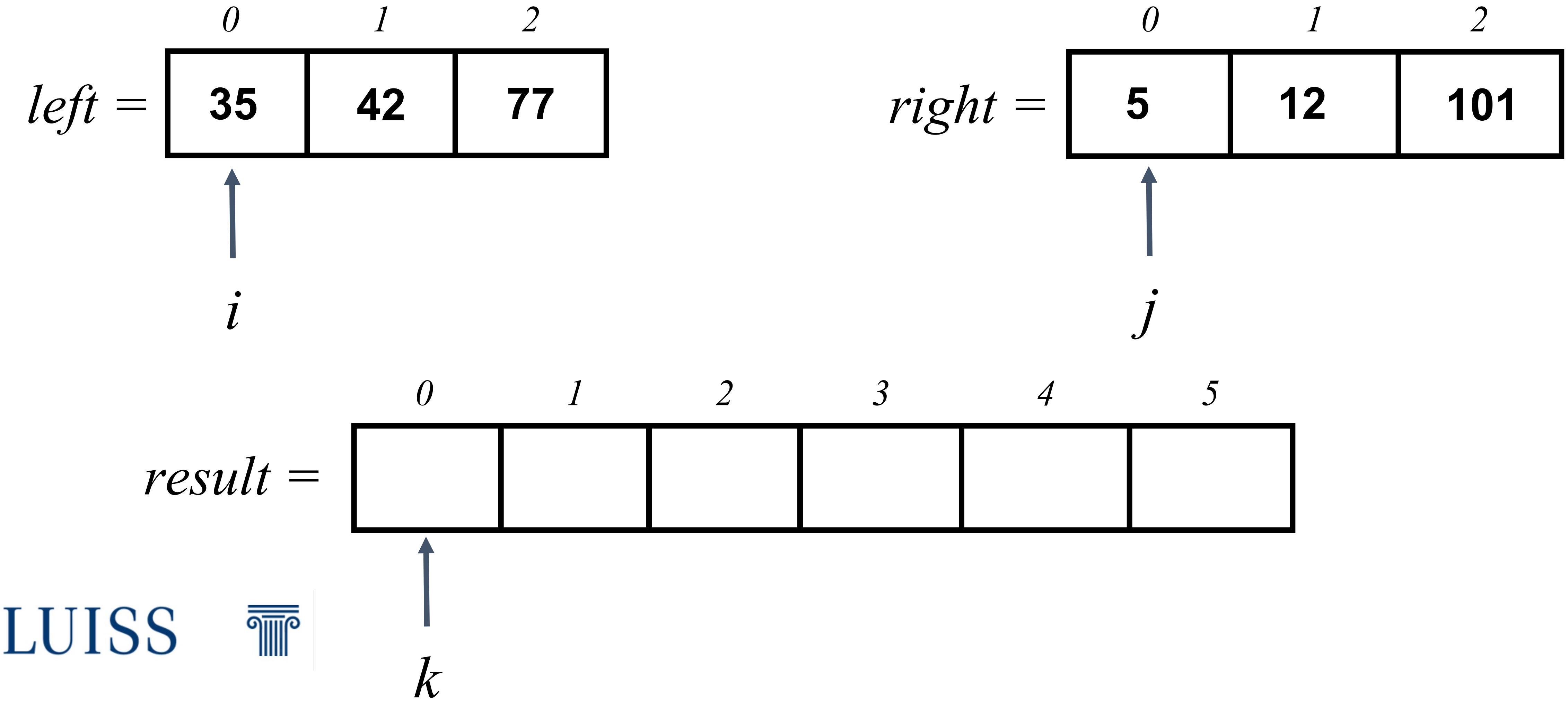
0	1	2
5	**12**	**101**

$result =$

0	1	2	3	4	5

Merge Sort: the merge procedure

Both *left* and *right* are ordered. We also use *i, j, k* as bookmarks



$left =$

0	1	2
35	42	77

$right =$

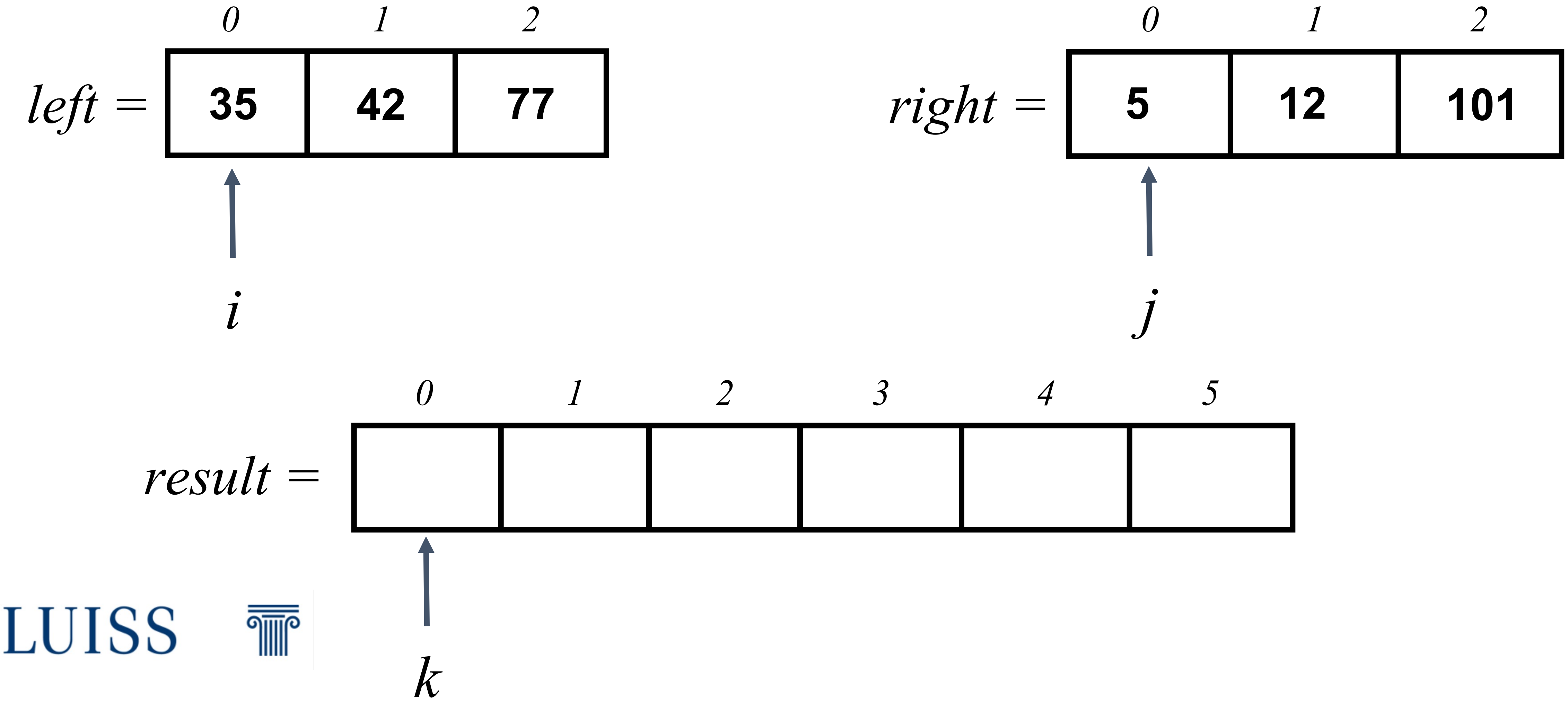
0	1	2
5	12	101

$result =$

0	1	2	3	4	5

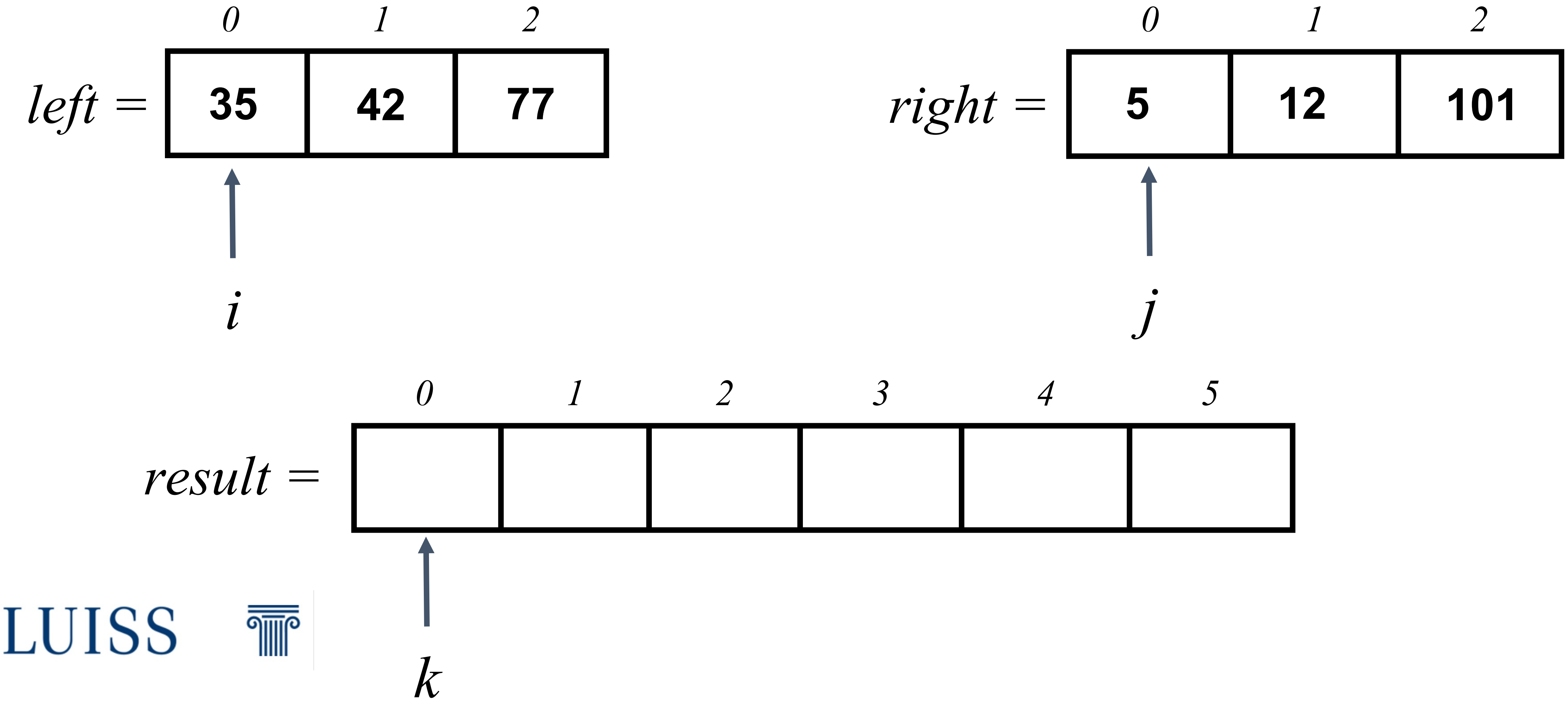
Merge Sort: the merge procedure

First of all we compare *left[0]* and *right[0]* to find the smallest value.



Merge Sort: the merge procedure

Once we find it we place it in the *result* list at position *0*



left =

0	1	2
35	**42**	**77**

right =

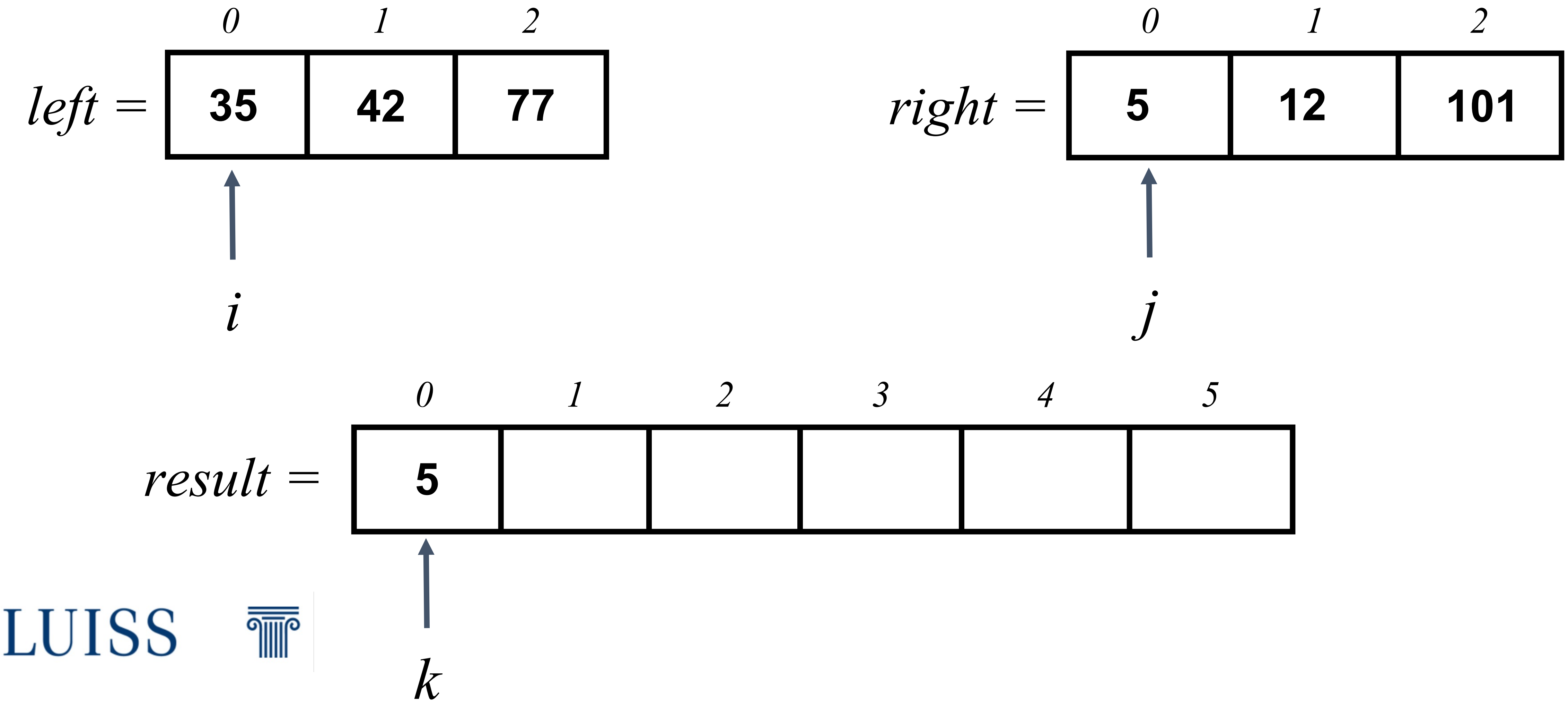
0	1	2
5	**12**	**101**

result =

0	1	2	3	4	5

Merge Sort: the merge procedure

Once we find it we place it in the *result* list at position *0*



left =

0	1	2
35	**42**	**77**

i

right =

0	1	2
5	**12**	**101**

j

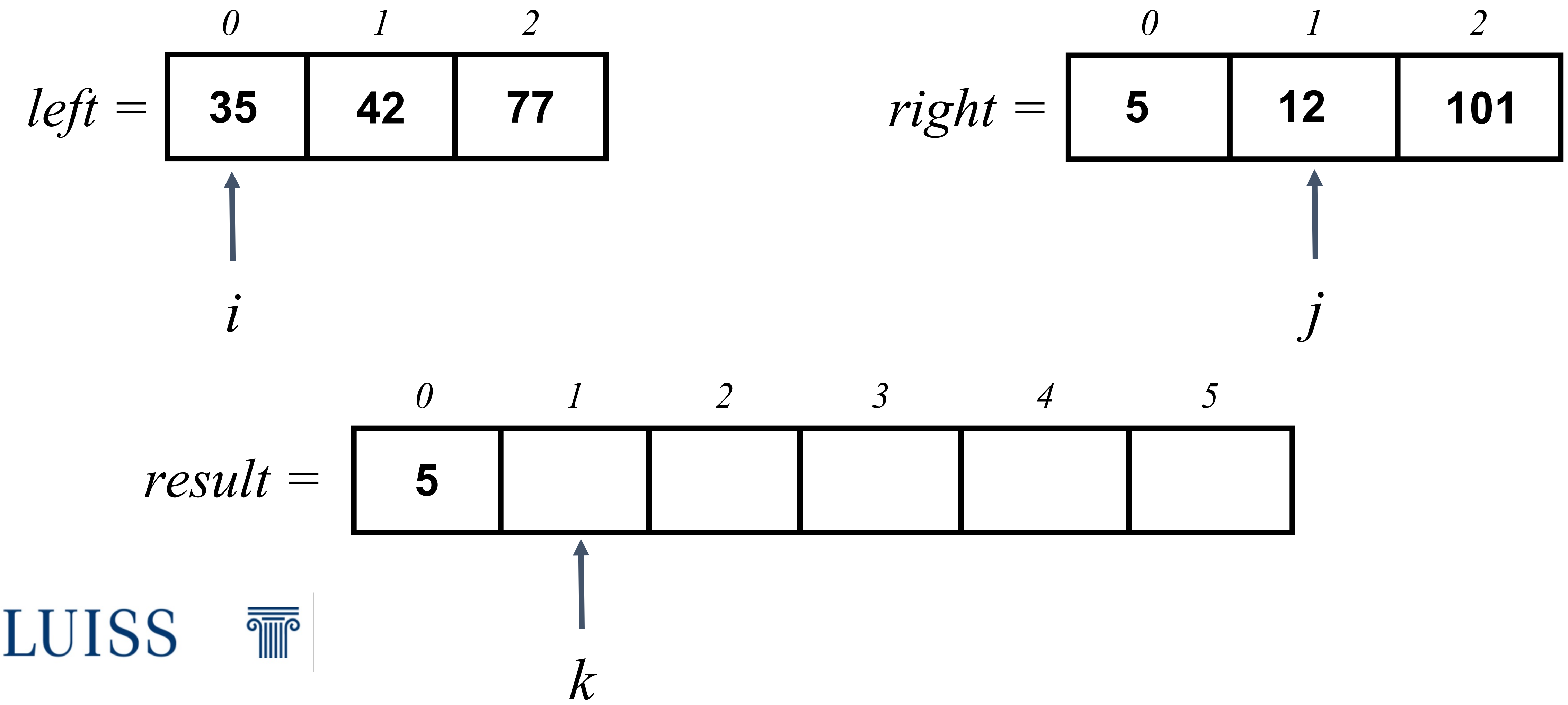
result =

0	1	2	3	4	5
5					

k

Merge Sort: the merge procedure

Then we increase the indices j and k



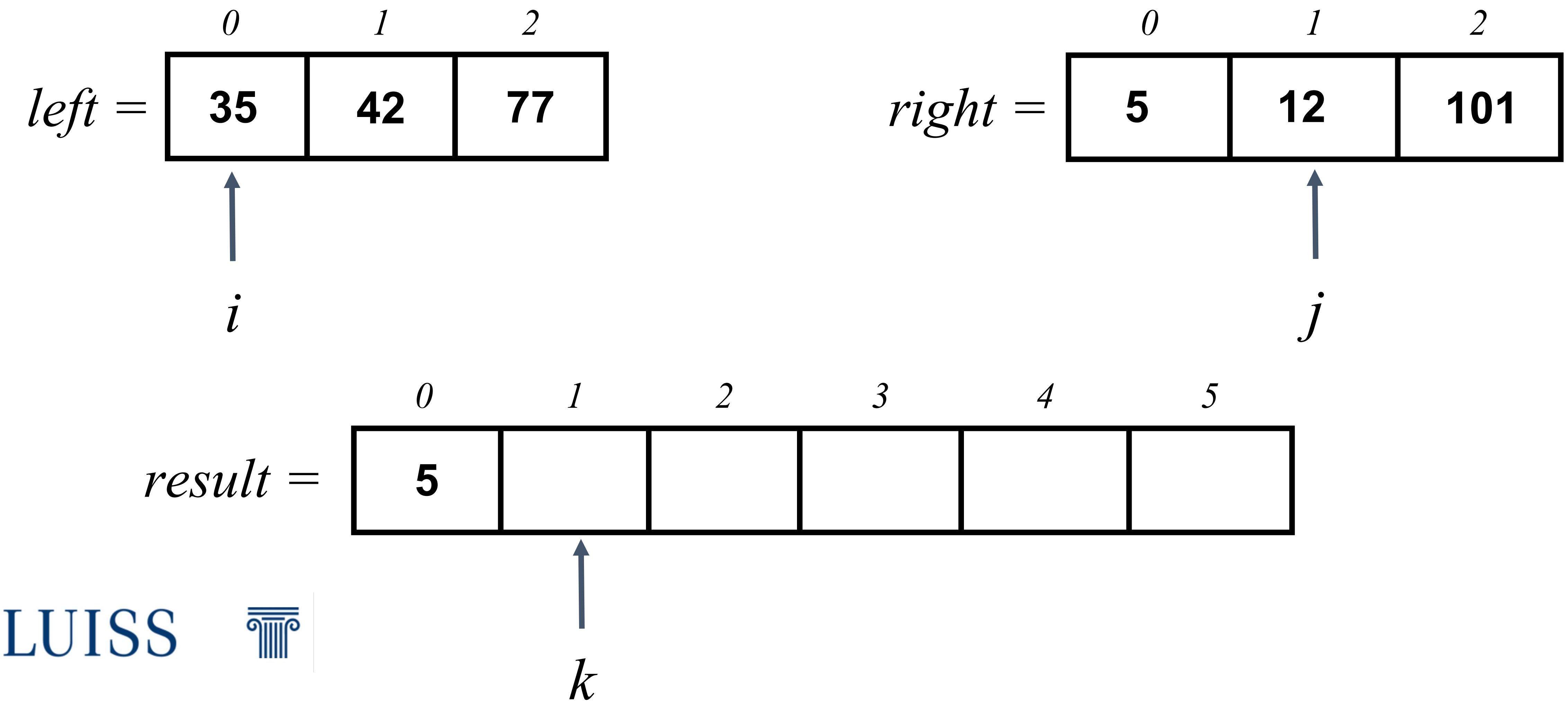
	0	1	2
$left =$	**35**	**42**	**77**
	i		

	0	1	2
$right =$	**5**	**12**	**101**
		j	

	0	1	2	3	4	5
$result =$	**5**					
		k				

Merge Sort: the merge procedure

Again, we have to compare *left[0]* and *right[1]* to find the smallest value.



left =

0	1	2
35	**42**	**77**

right =

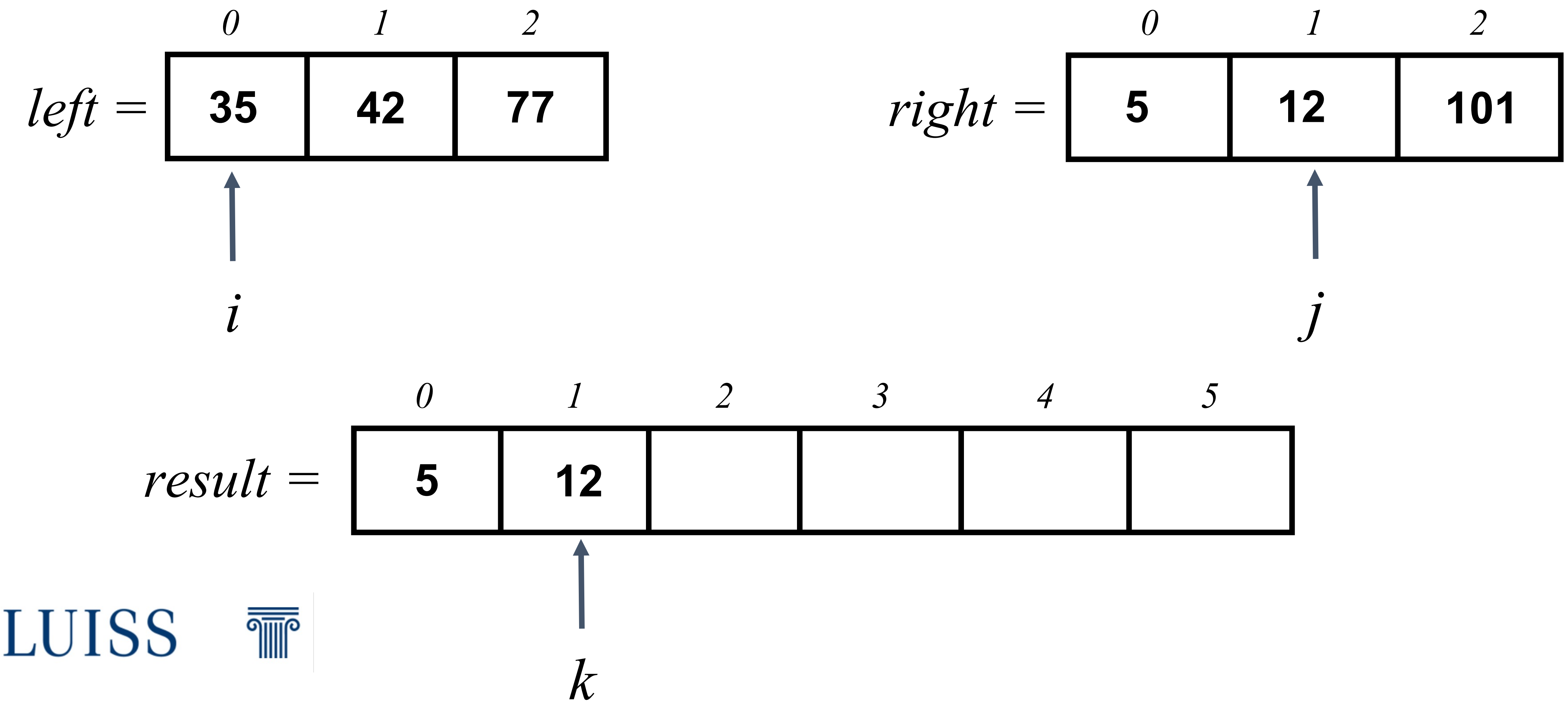
0	1	2
5	**12**	**101**

result =

0	1	2	3	4	5
5					

Merge Sort: the merge procedure

Once we find it we place it in the *result* list at position *1*



left =

0	1	2
35	42	77

right =

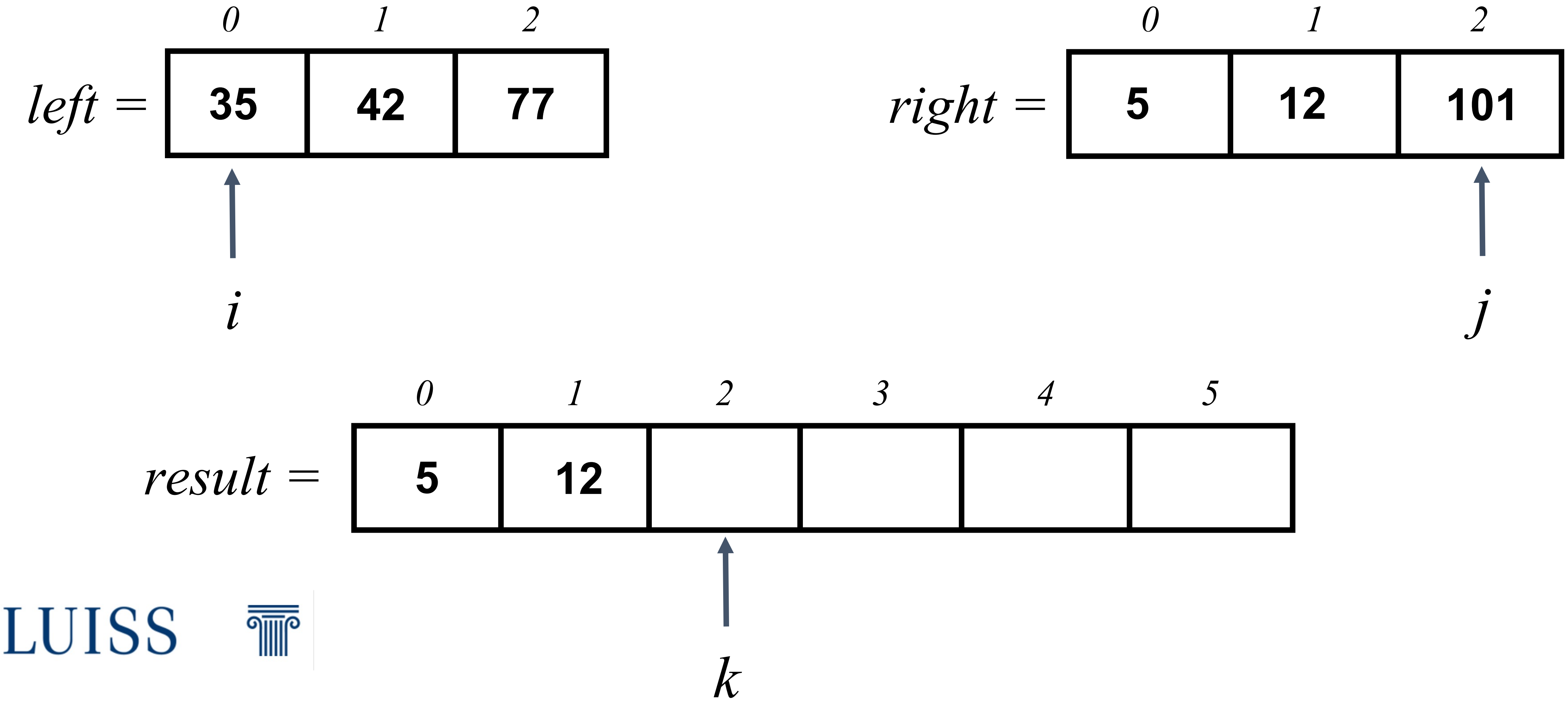
0	1	2
5	12	101

result =

0	1	2	3	4	5
5	12				

Merge Sort: the merge procedure

Then again we increase the indices j and k



$left =$

0	1	2
35	42	77

$right =$

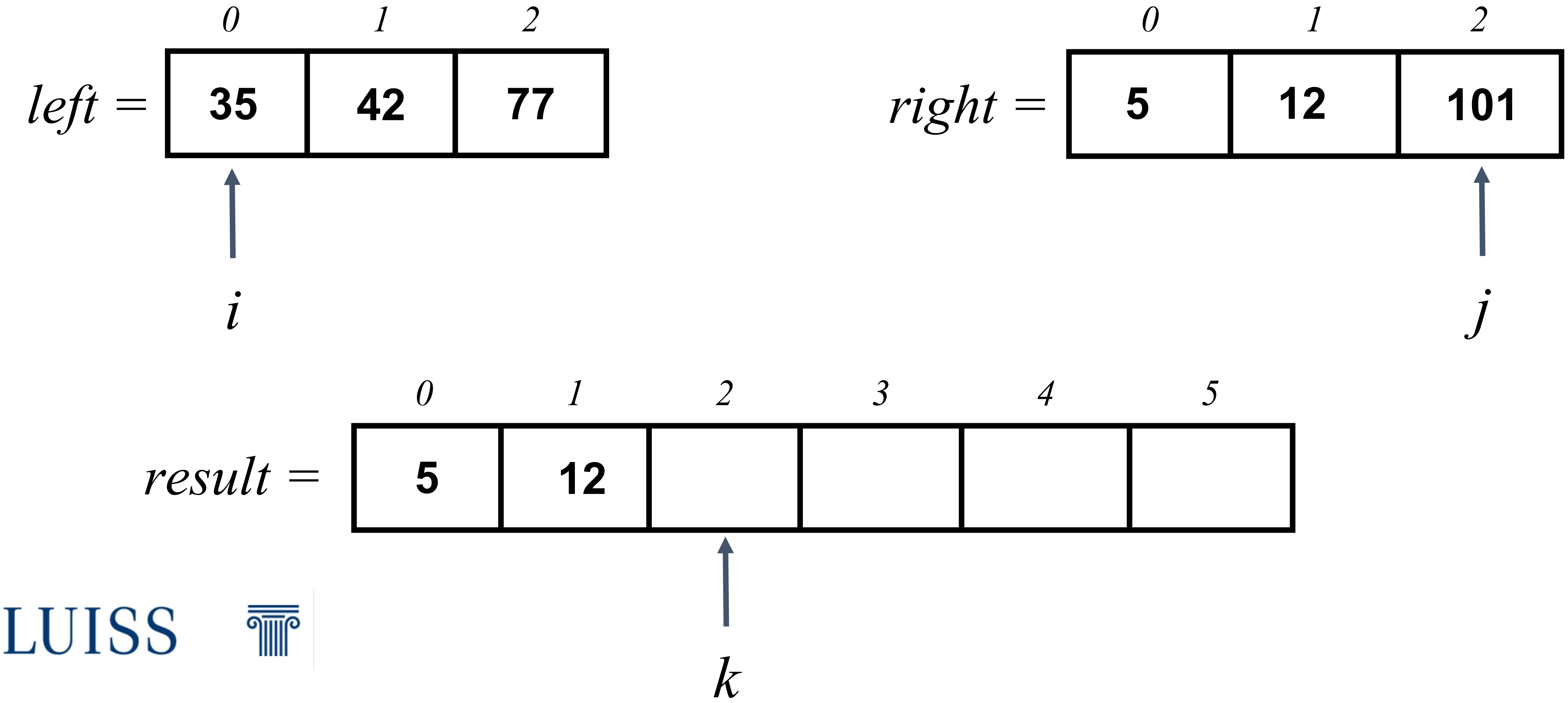
0	1	2
5	12	101

$result =$

0	1	2	3	4	5
5	12				

Merge Sort: the merge procedure

Now we have to compare *left[0]* and *right[2]* to find the smallest value.



left =

0	1	2
35	42	77

right =

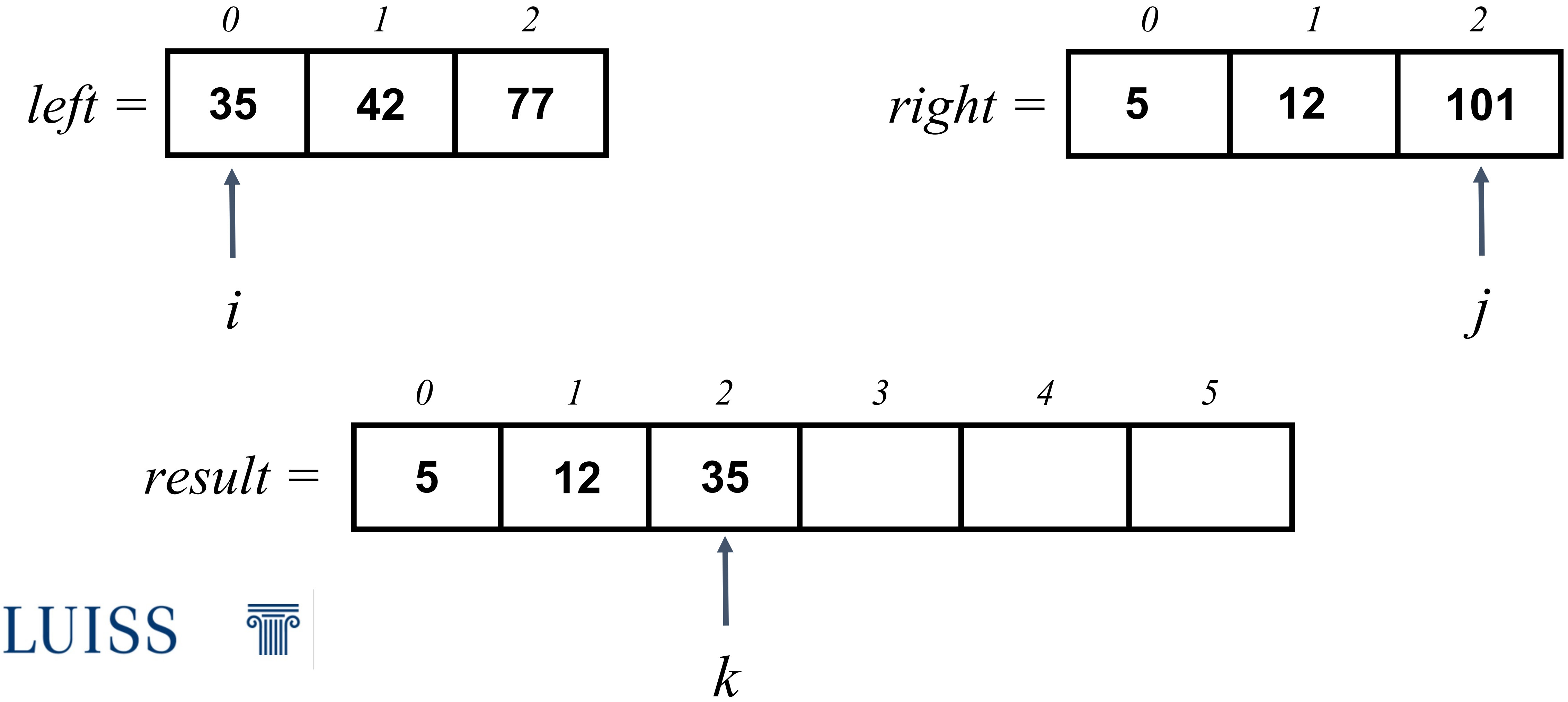
0	1	2
5	12	101

result =

0	1	2	3	4	5
5	12				

Merge Sort: the merge procedure

Once we find it we place it in the *result* list at position *2*



left =

0	1	2
35	42	77

right =

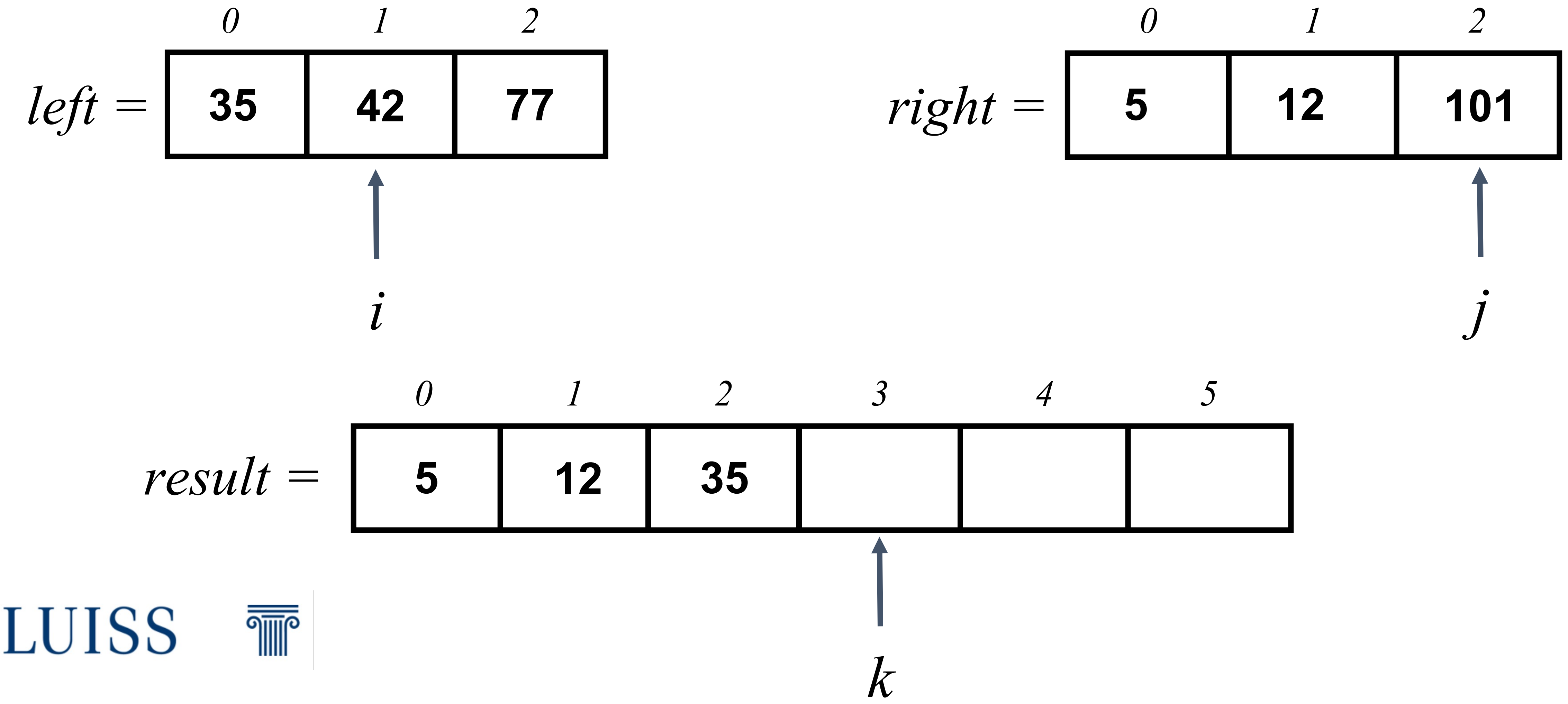
0	1	2
5	12	101

result =

0	1	2	3	4	5
5	12	35			

Merge Sort: the merge procedure

This time we increase the indices i and k



$left =$

0	1	2
35	42	77

i

$right =$

0	1	2
5	12	101

j

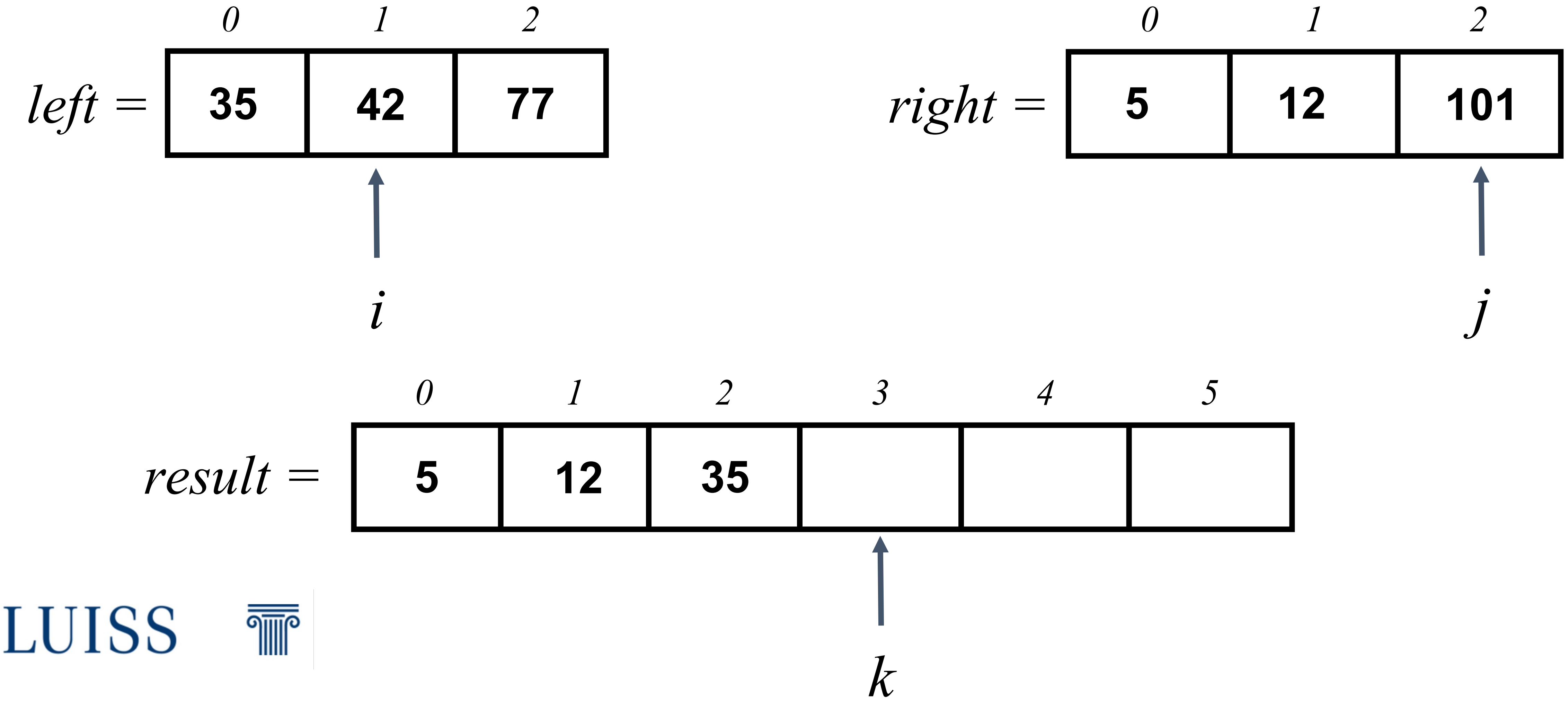
$result =$

0	1	2	3	4	5
5	12	35			

k

Merge Sort: the merge procedure

Now we have to compare *left[1]* and *right[2]* to find the smallest value.



left =

0	1	2
35	42	77

right =

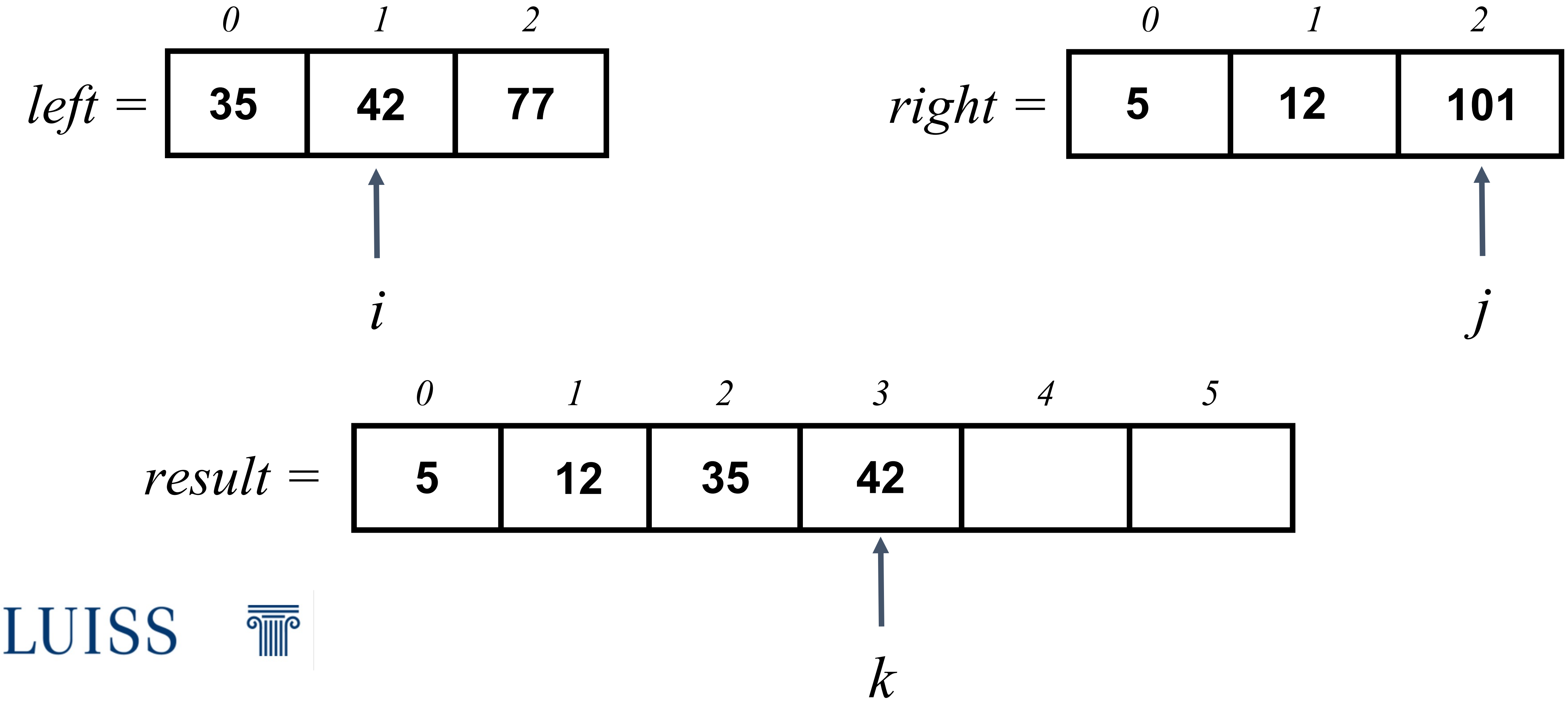
0	1	2
5	12	101

result =

0	1	2	3	4	5
5	12	35			

Merge Sort: the merge procedure

Once we find it we place it in the *result* list at position *3*



$left =$

0	1	2
35	**42**	**77**

$right =$

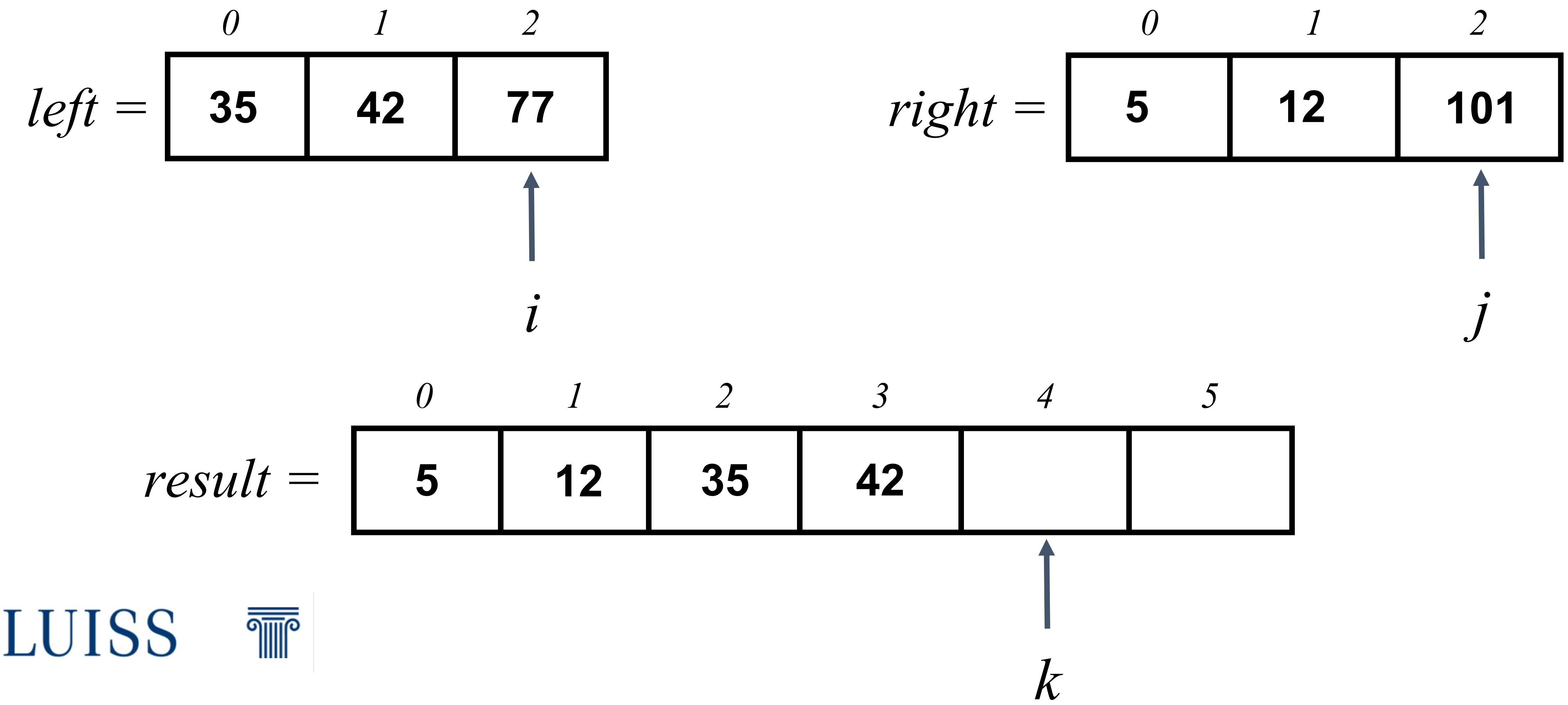
0	1	2
5	**12**	**101**

$result =$

0	1	2	3	4	5
5	**12**	**35**	**42**		

Merge Sort: the merge procedure

Again we increase the indices i and k



$left =$

0	1	2
35	**42**	**77**

i → 2

$right =$

0	1	2
5	**12**	**101**

j → 2

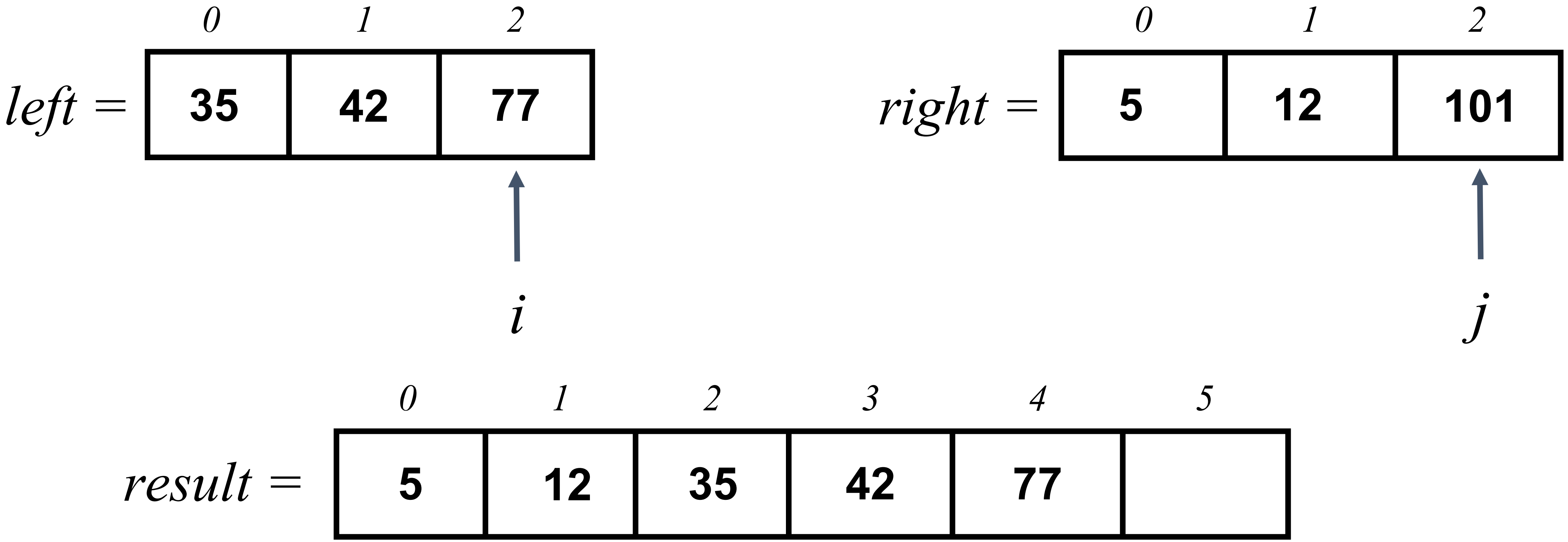
$result =$

0	1	2	3	4	5
5	**12**	**35**	**42**		

k → 4

Merge Sort: the merge procedure

Since we have just two elements now we can look for the smaller one and put it into position *4* and the larger one into position *5*



left =

0	1	2
35	**42**	**77**

right =

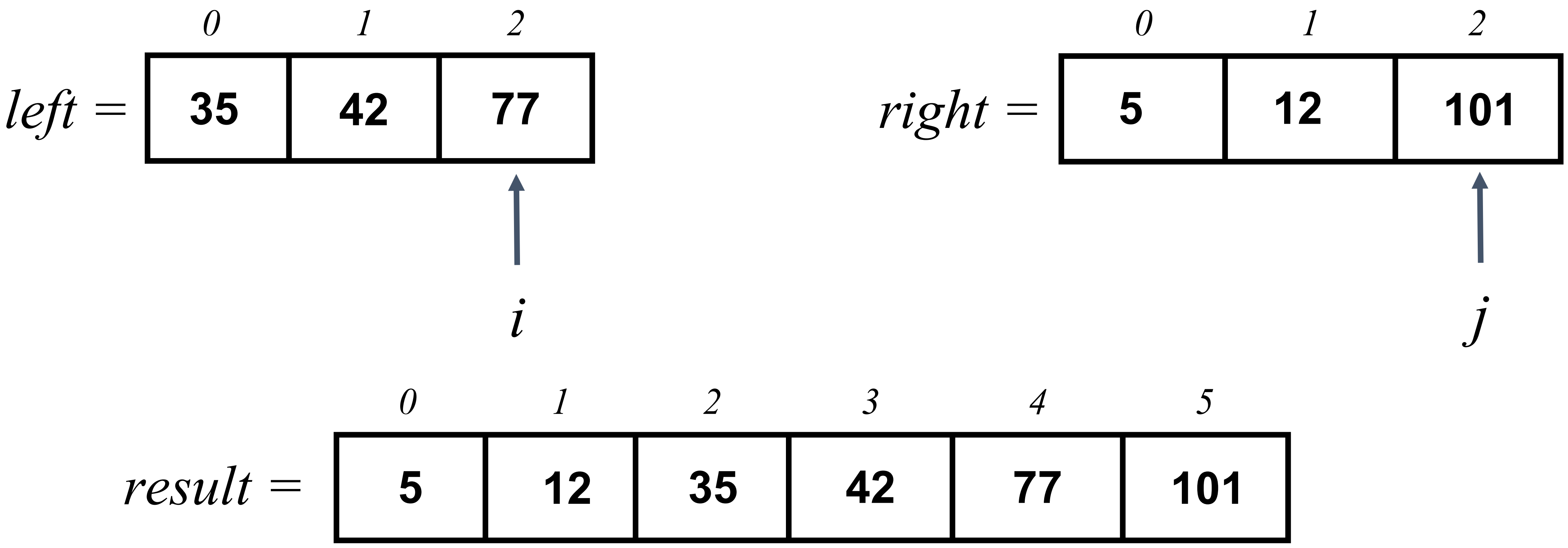
0	1	2
5	**12**	**101**

result =

0	1	2	3	4	5
5	**12**	**35**	**42**	**77**	

Merge Sort: the merge procedure

Since we have just two elements now we can look for the smaller one and put it into position *4* and the larger one into position *5*



$left =$

0	1	2
35	42	77

i

$right =$

0	1	2
5	12	101

j

$result =$

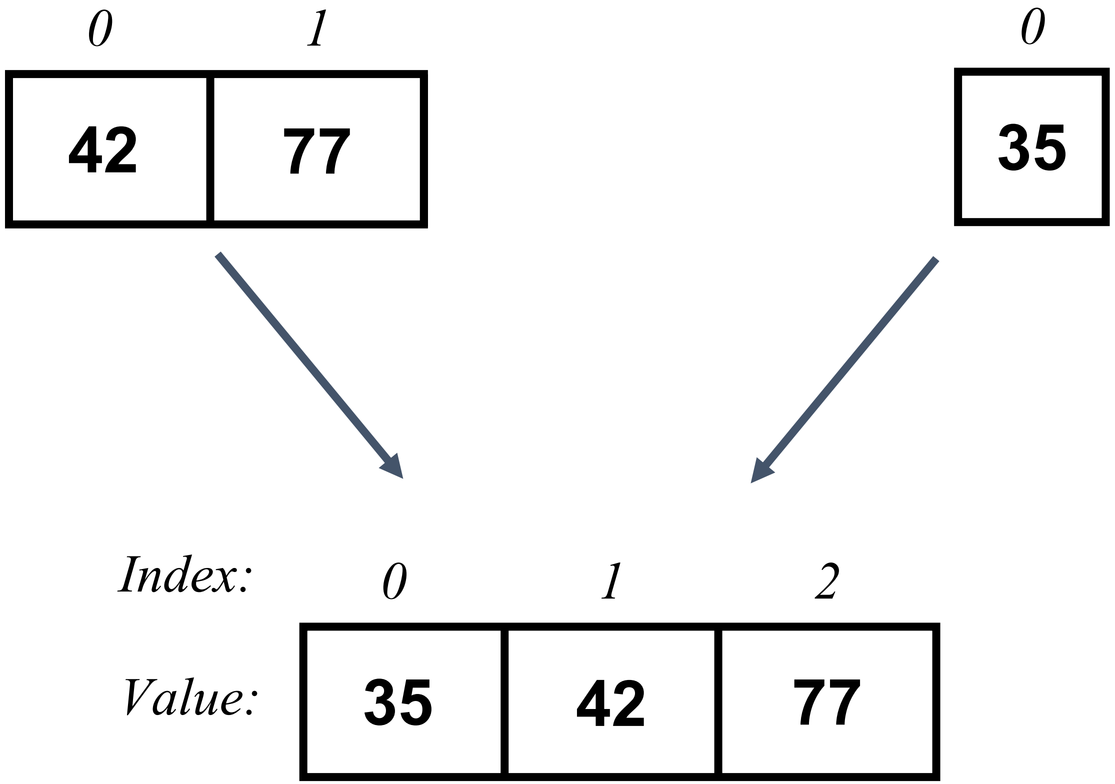
0	1	2	3	4	5
5	12	35	42	77	101

Merge Sort: the merge procedure

The computational complexity of this procedure is $\Theta(n)$

It works because each time we select the minimum among the smallest values!

Merge Sort: an example



0	1
42	77

0
35

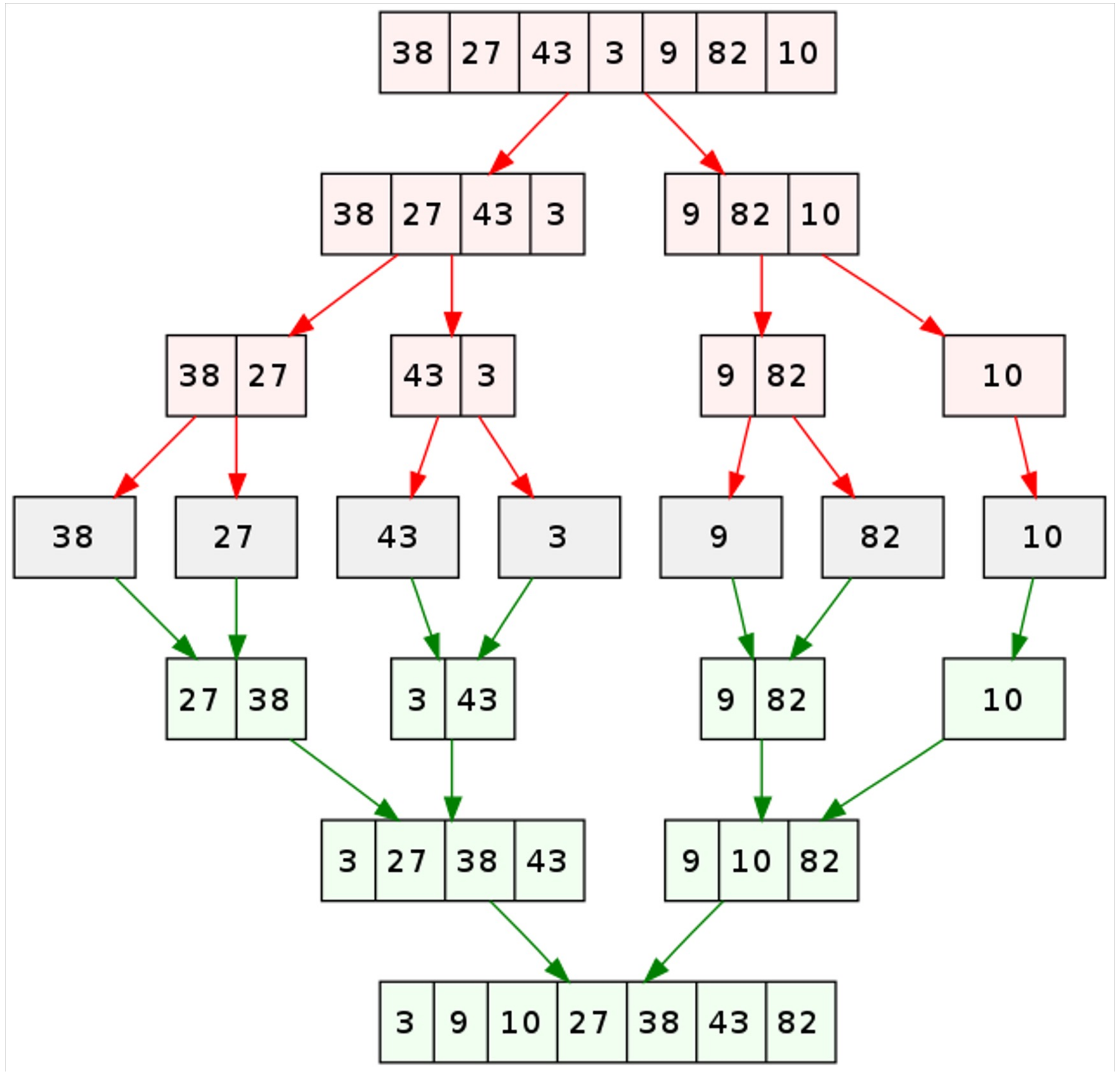
Index:	0	1	2
Value:	35	42	77

Merge Sort: an example

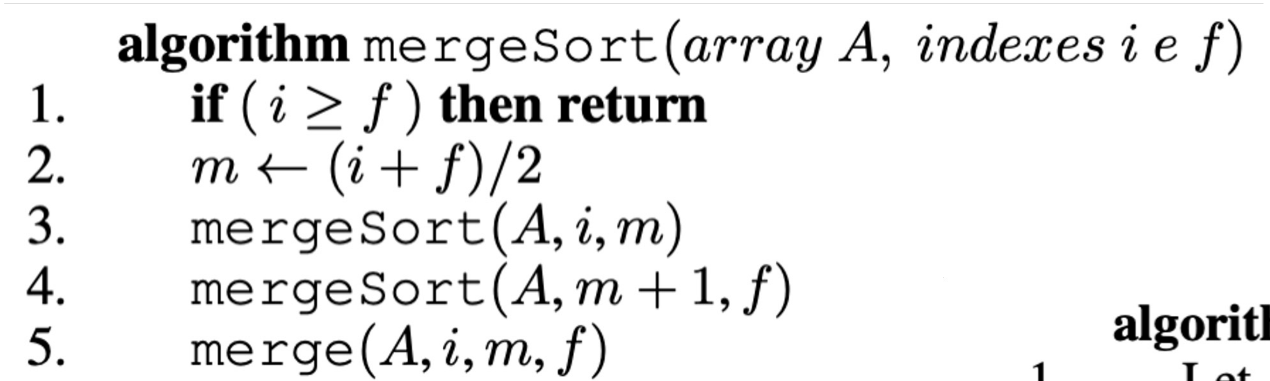
And so on!

Merge Sort: an example

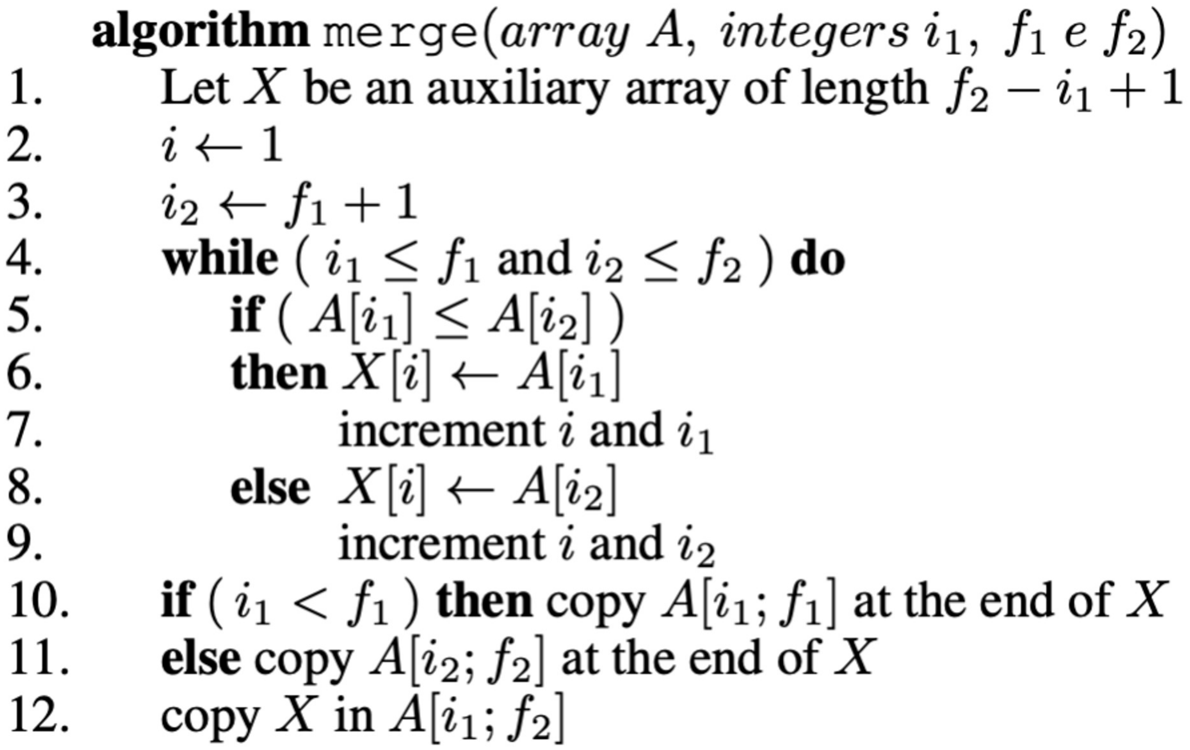
Visualization of the tree for a particular instance



Merge Sort: pseudocode



```
algorithm mergeSort($array\ A,\ indexes\ i\ e\ f$)
    if ( $i \geq f$ ) then return
    $m \leftarrow (i + f)/2$
    mergeSort($A, i, m$)
    mergeSort($A, m + 1, f$)
    merge($A, i, m, f$)
```



```
algorithm merge($array\ A,\ integers\ i_1,\ f_1\ e\ f_2$)
    Let $X$ be an auxiliary array of length $f_2 - i_1 + 1$
    $i \leftarrow 1$
    $i_2 \leftarrow f_1 + 1$
    while ( $i_1 \leq f_1$ and $i_2 \leq f_2$ ) do
        if ( $A[i_1] \leq A[i_2]$ )
        then $X[i] \leftarrow A[i_1]$
            increment $i$ and $i_1$
        else $X[i] \leftarrow A[i_2]$
            increment $i$ and $i_2$
    if ( $i_1 < f_1$ ) then copy $A[i_1; f_1]$ at the end of $X$
    else copy $A[i_2; f_2]$ at the end of $X$
    copy $X$ in $A[i_1; f_2]$
```

Merge procedure: Pseudocode

Computational Complexity?

Merge procedure: Pseudocode

Computational Complexity?

$$\boldsymbol{O(n\,log\,n)}$$

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

We need:

- the lenght of the sub list

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

We need:

- the lenght of the sub list
- A left index pointing out the starting point of the FIRST list

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

We need:

- the lenght of the sub list
- A left index pointing out the starting point of the FIRST list
- A mid index pointing out the ending point of the FIRST list and the starting point of the SECOND one

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

We need:

- the lenght of the sub list
- A left index pointing out the starting point of the FIRST list
- A mid index pointing out the ending point of the FIRST list and the starting point of the SECOND one
- A right index pointing out the ending point of the SECOND list

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

The iterative procedure starts from the smallest possible instance using a bottom-up approach

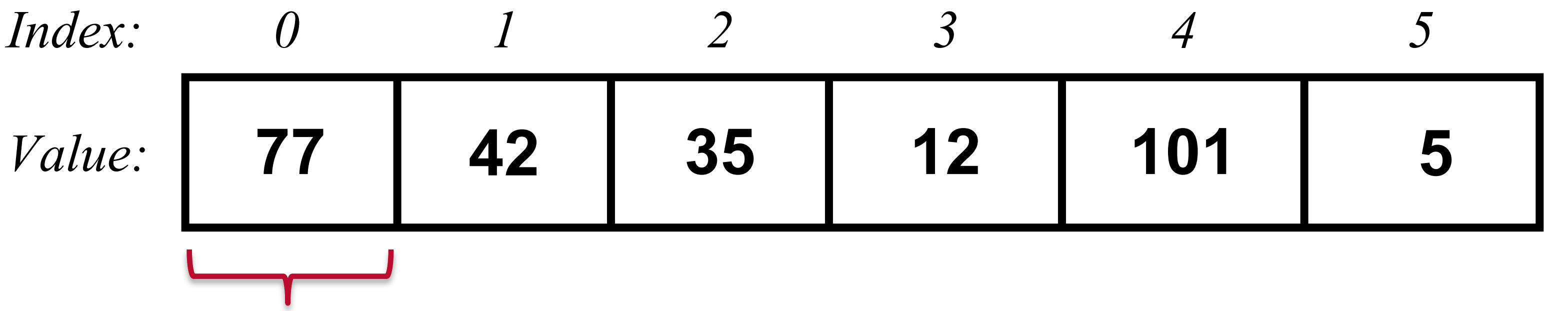
Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

The iterative procedure starts from the smallest possible instance using a bottom-up approach

The difference with the recursive solution is that we don’t need to go top-down and then bottom-up

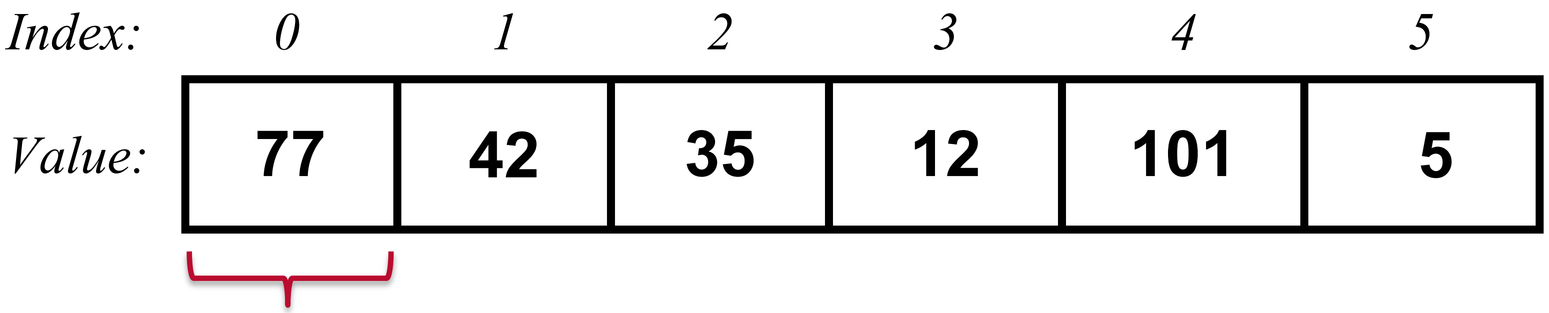
Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

width = 1

Merge Sort: An iterative solution

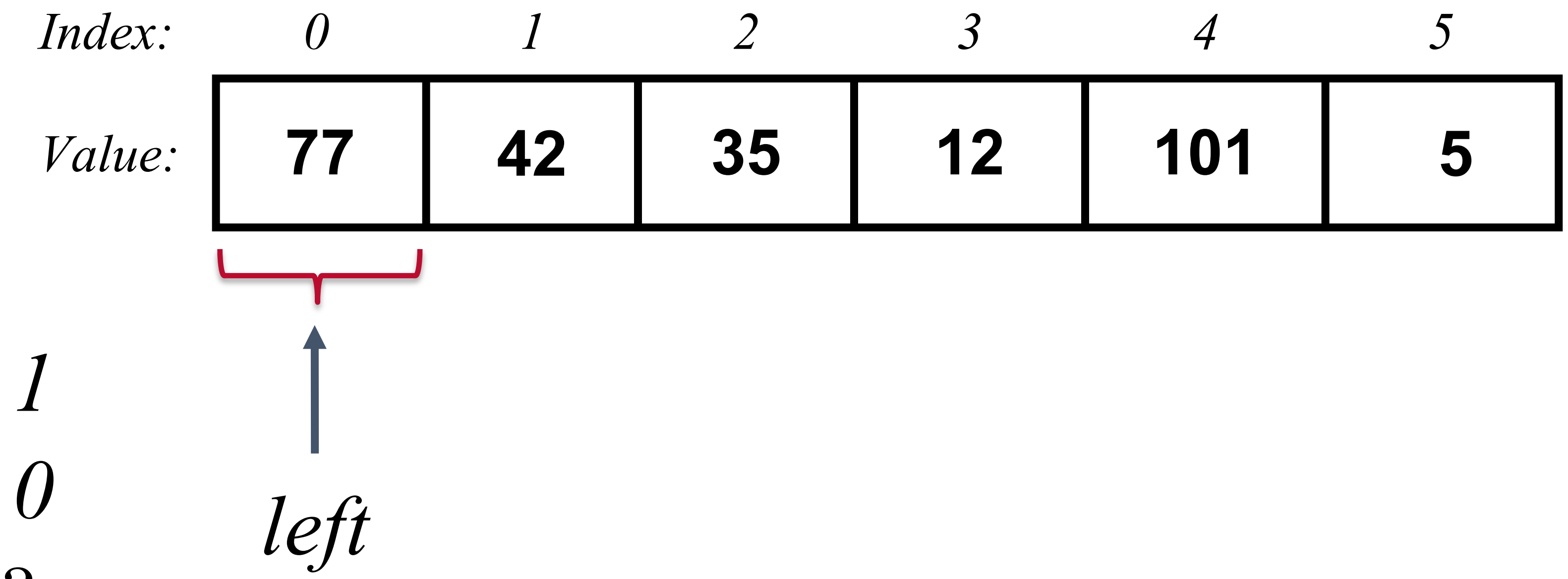


Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

$width = 1$

$left = ?$

Merge Sort: An iterative solution



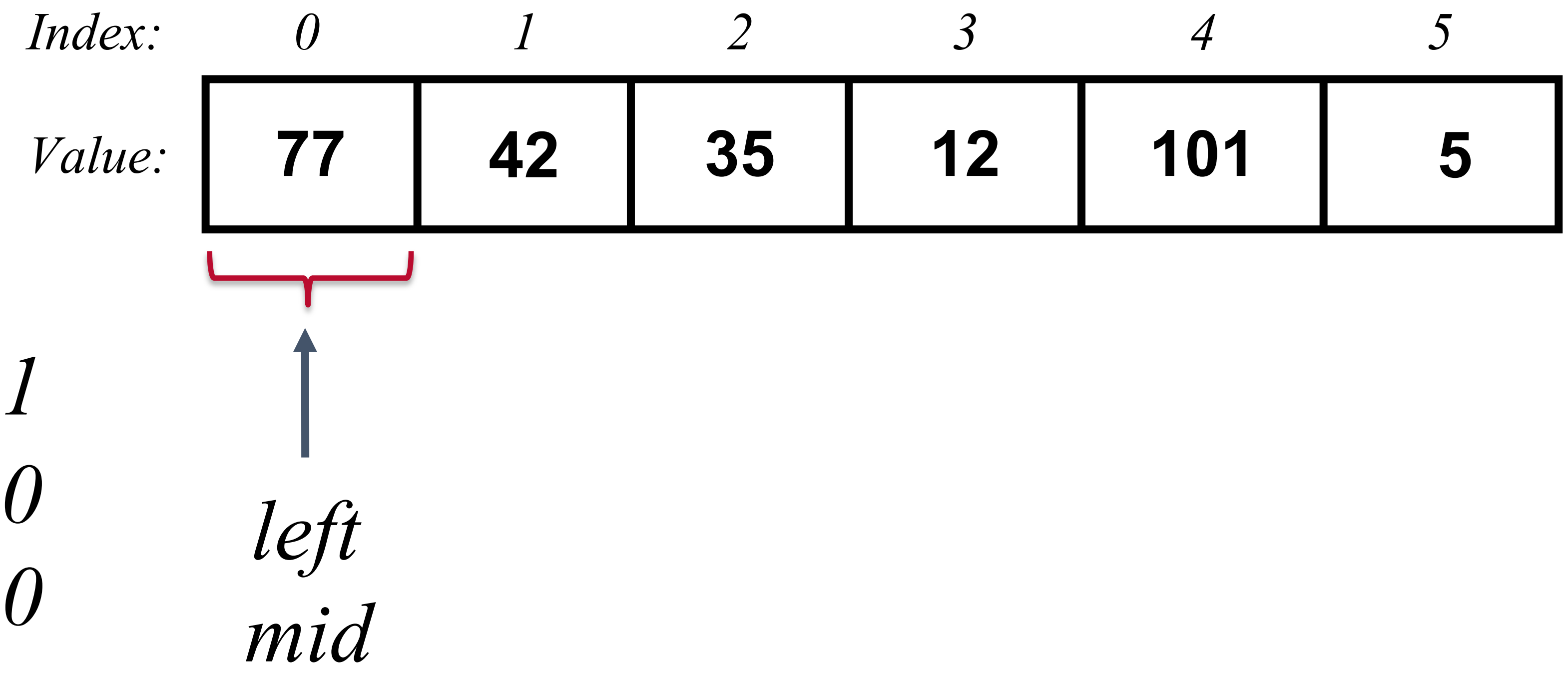
Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

width = *1*

left = *0*

mid = *?*

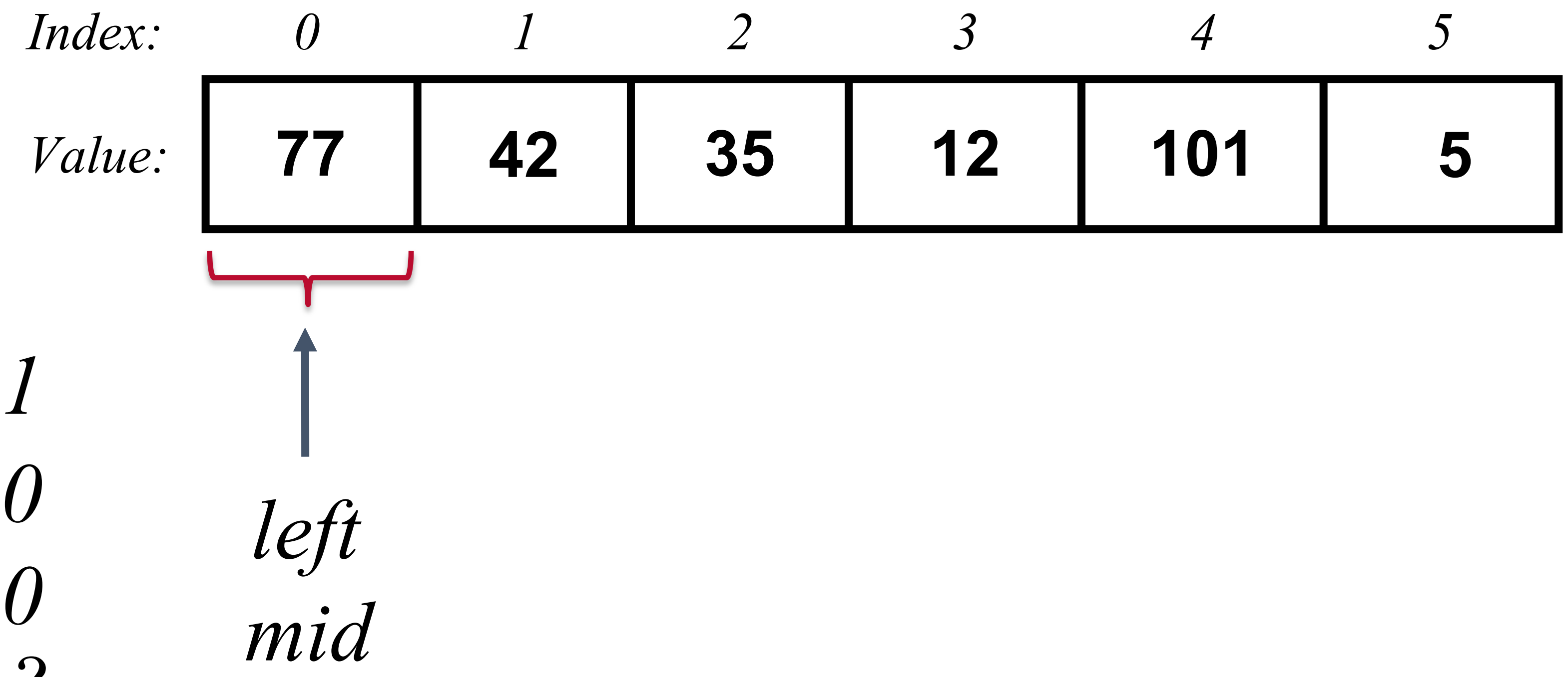
Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

width = *1*
left = *0*
mid = *0*

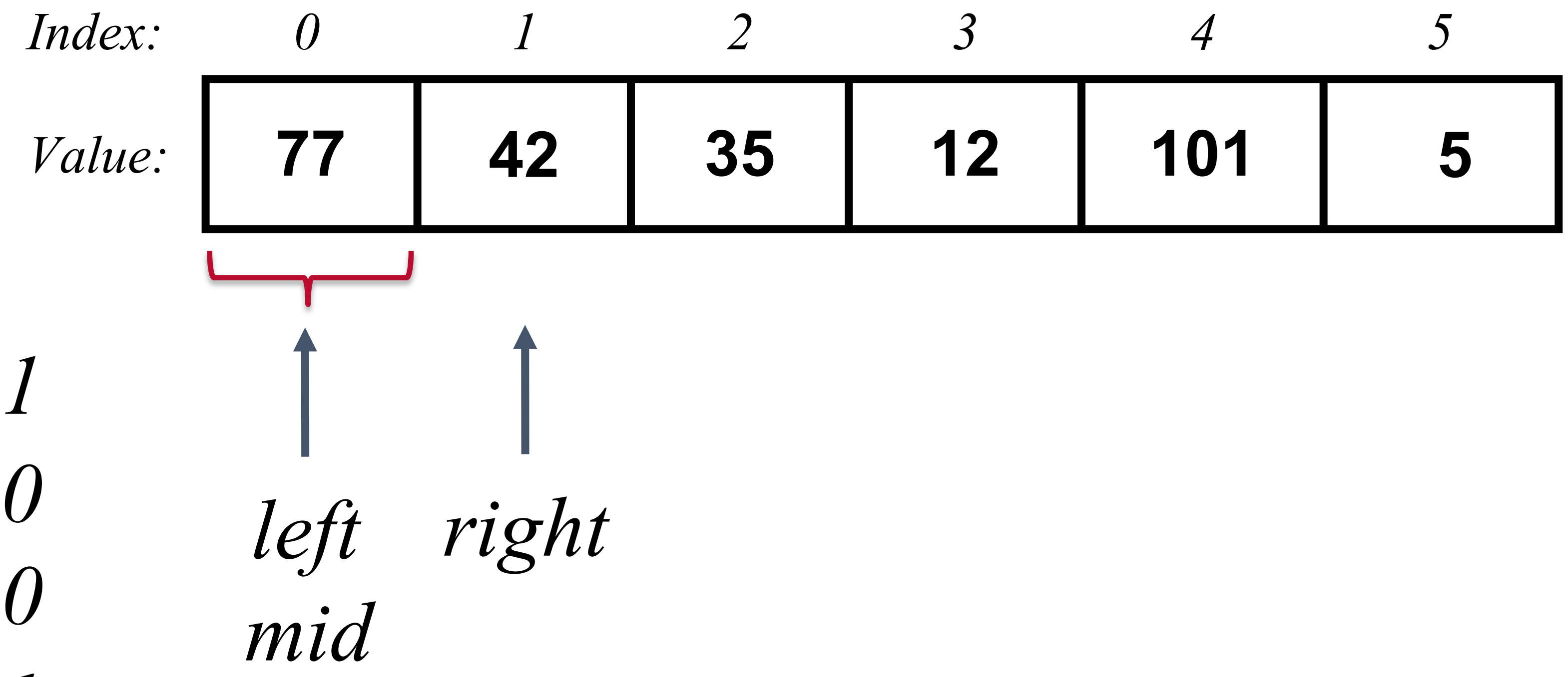
Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

width = 1
left = 0
mid = 0
right = ?

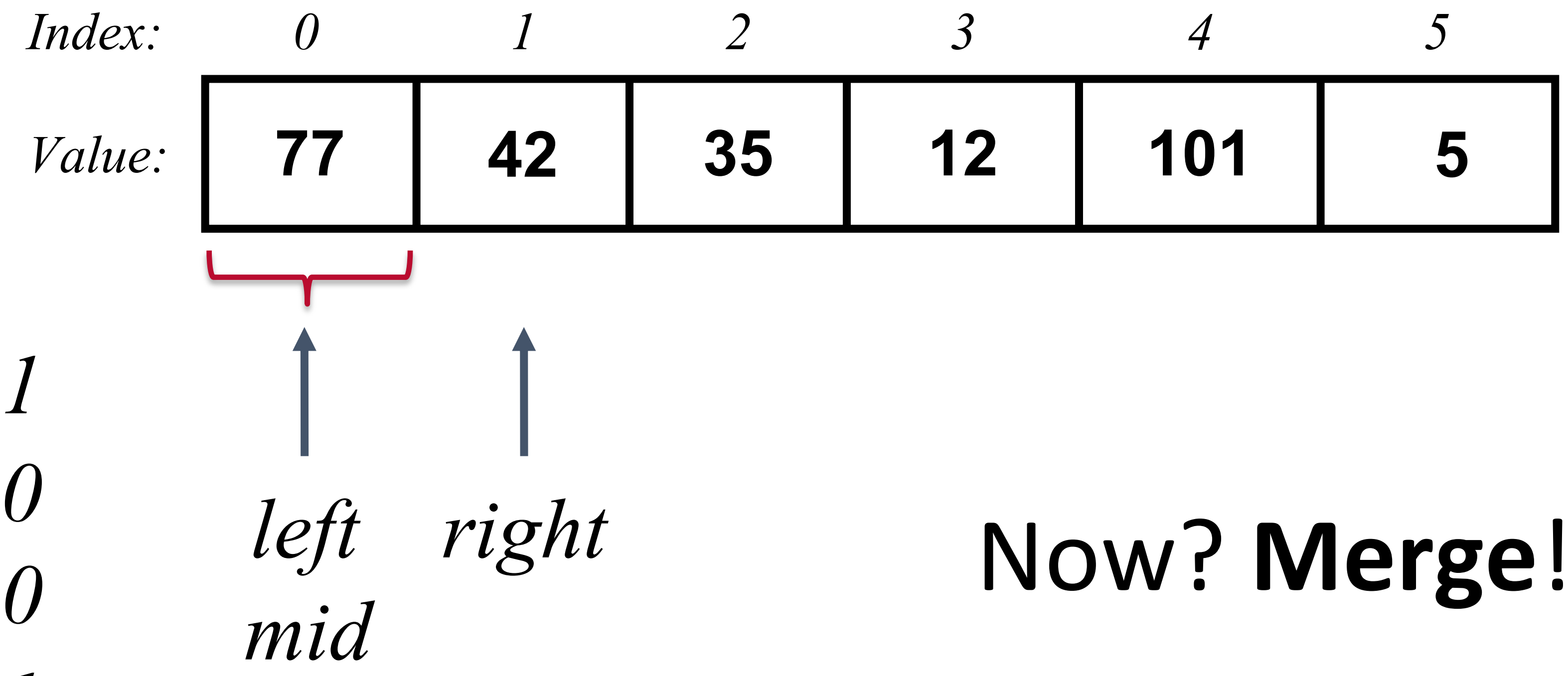
Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

width = 1
left = 0
mid = 0
right = 1

Merge Sort: An iterative solution

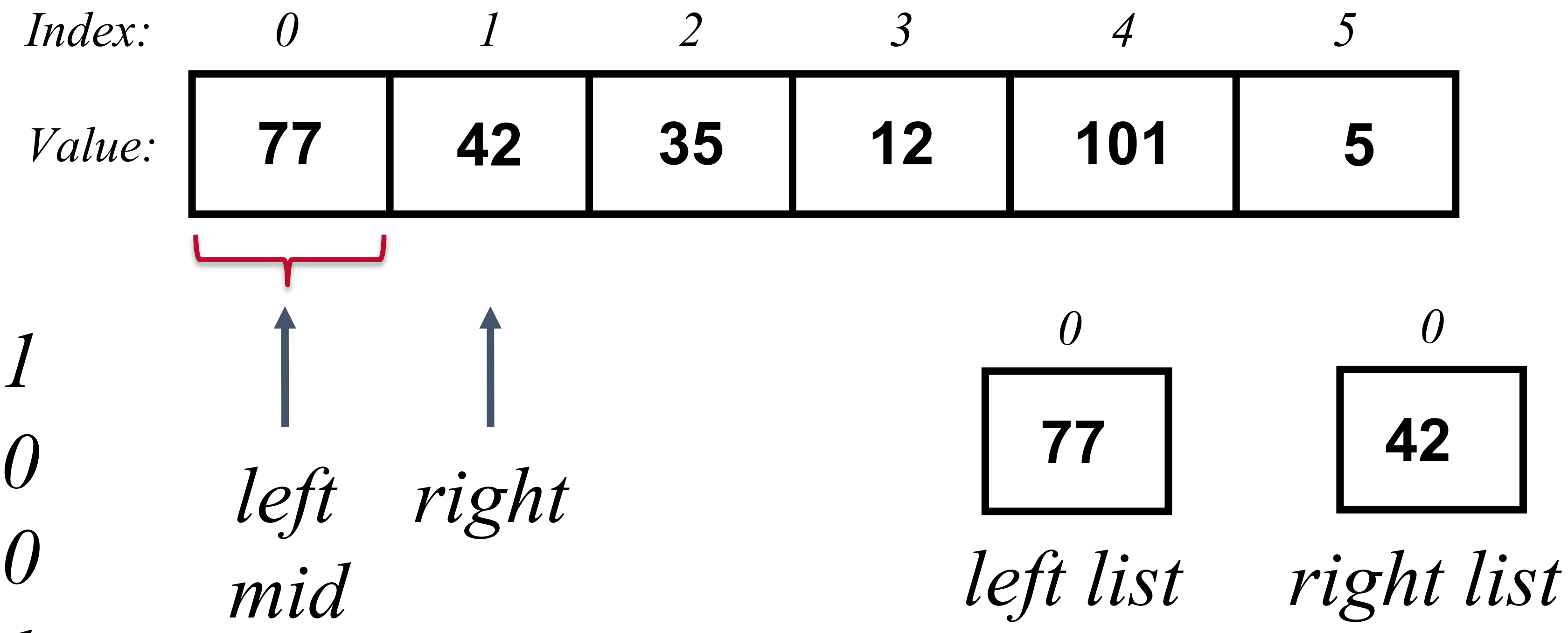


Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

width = *1*
left = *0*
mid = *0*
right = *1*

Now? **Merge!**

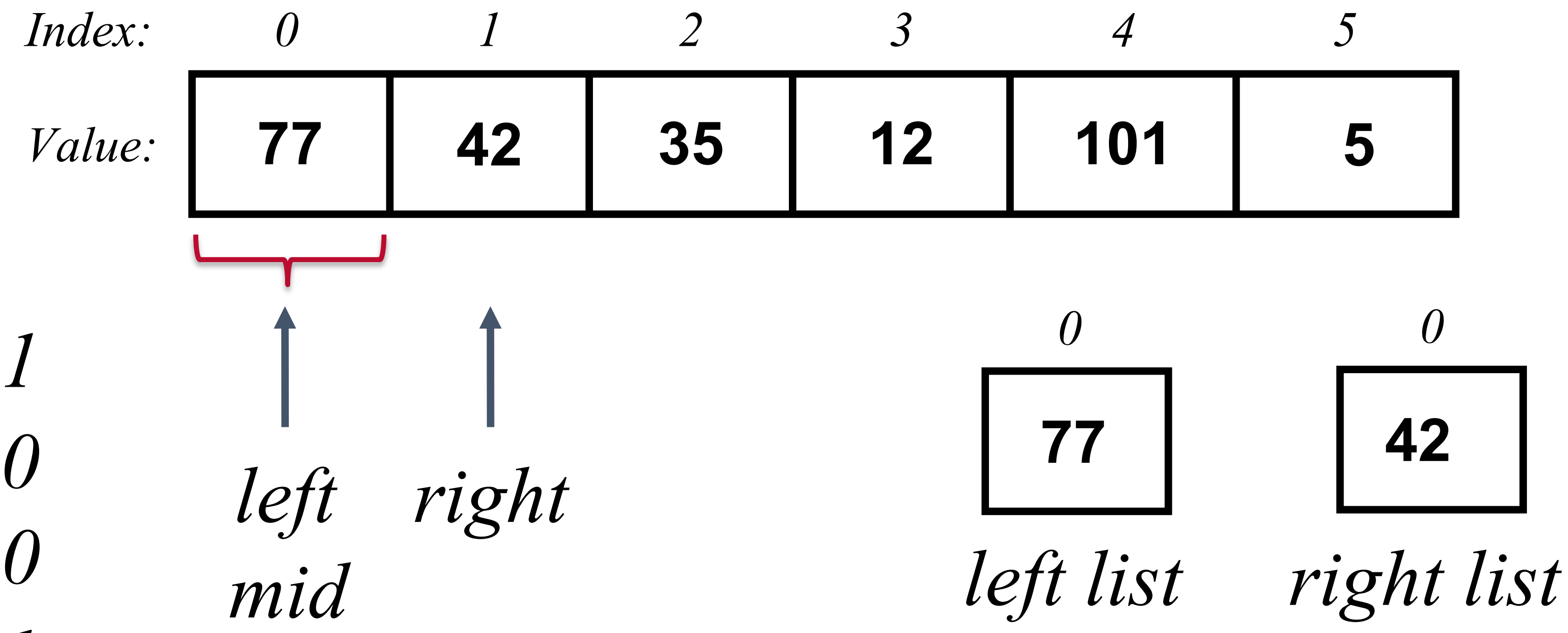
Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

width = 1
left = 0
mid = 0
right = 1

Merge Sort: An iterative solution

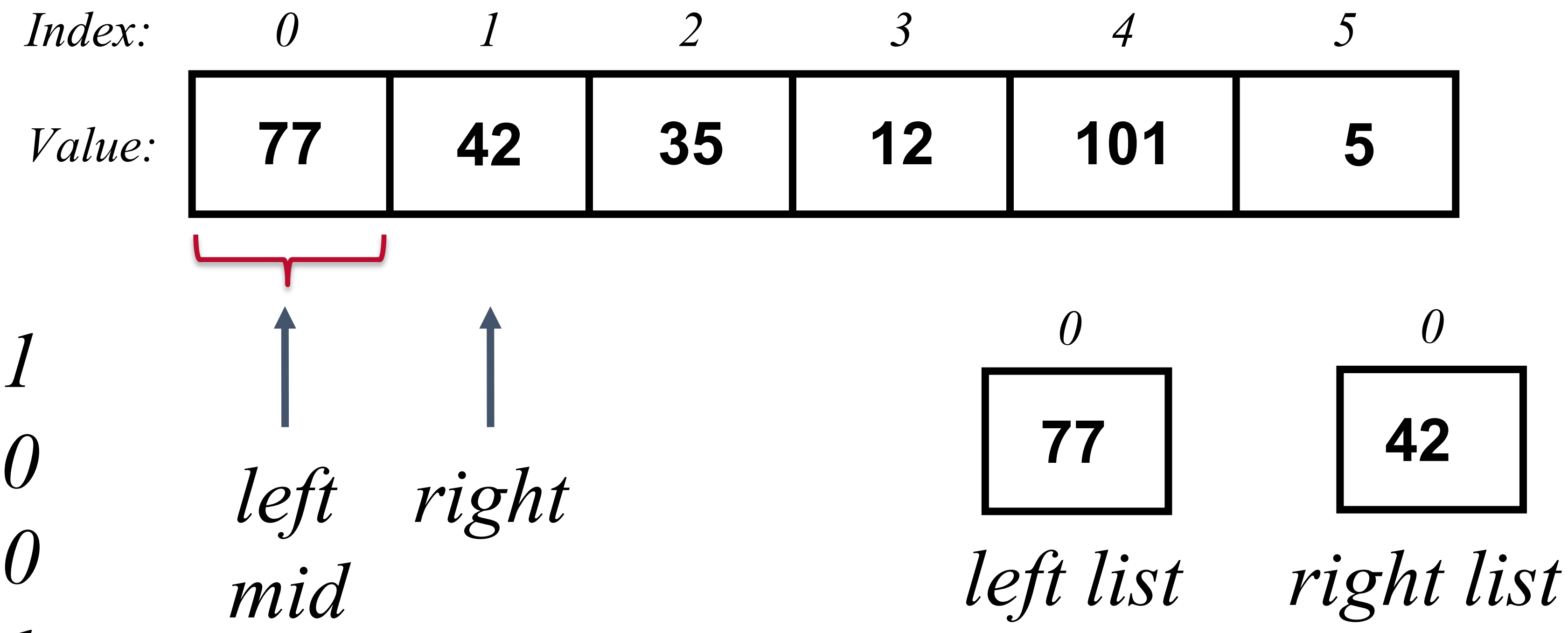


Index:	0	1	2	3	4	5
Value:	77	42	35	12	101	5

width = 1
left = 0
mid = 0
right = 1

Is 77 < 42?

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

left
mid

right

0	*0*
77	**42**
left list	*right list*

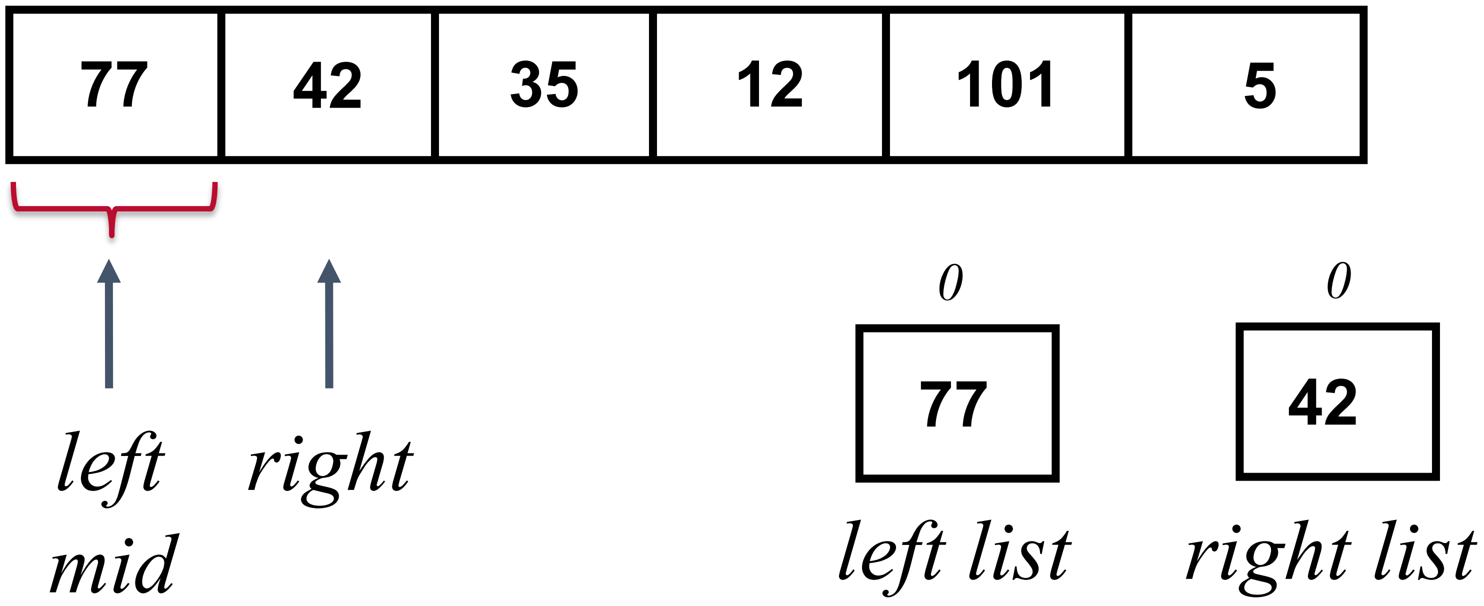
width = 1
left = 0
mid = 0
right = 1

Clearly not!

Merge Sort: An iterative solution

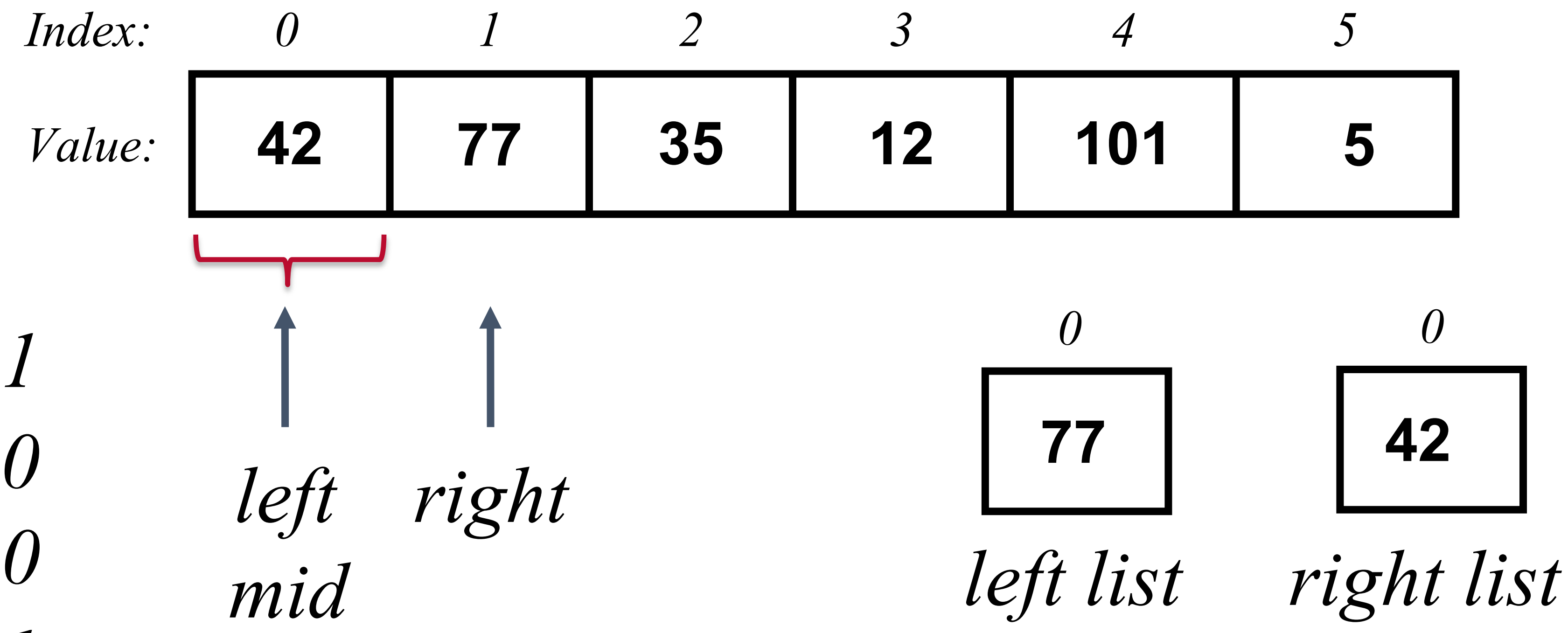
Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**77**	**42**	**35**	**12**	**101**	**5**

$width = 1$
$left = 0$
$mid = 0$
$right = 1$



Put in the original list 42 at index 0 and 77 at index 1

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**35**	**12**	**101**	**5**

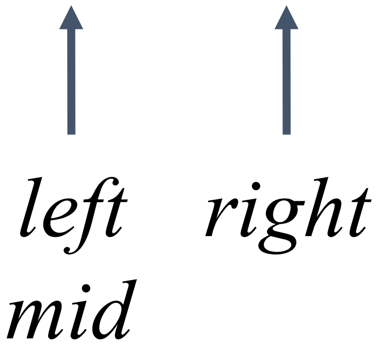
width = 1
left = 0
mid = 0
right = 1

Put in the original list 42 at index 0 and 77 at index 1

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**35**	**12**	**101**	**5**

width = *1*
left = *?*
mid = *?*
right = *?*

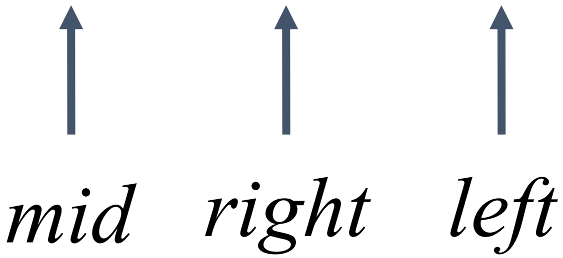


left
mid

right

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**35**	**12**	**101**	**5**



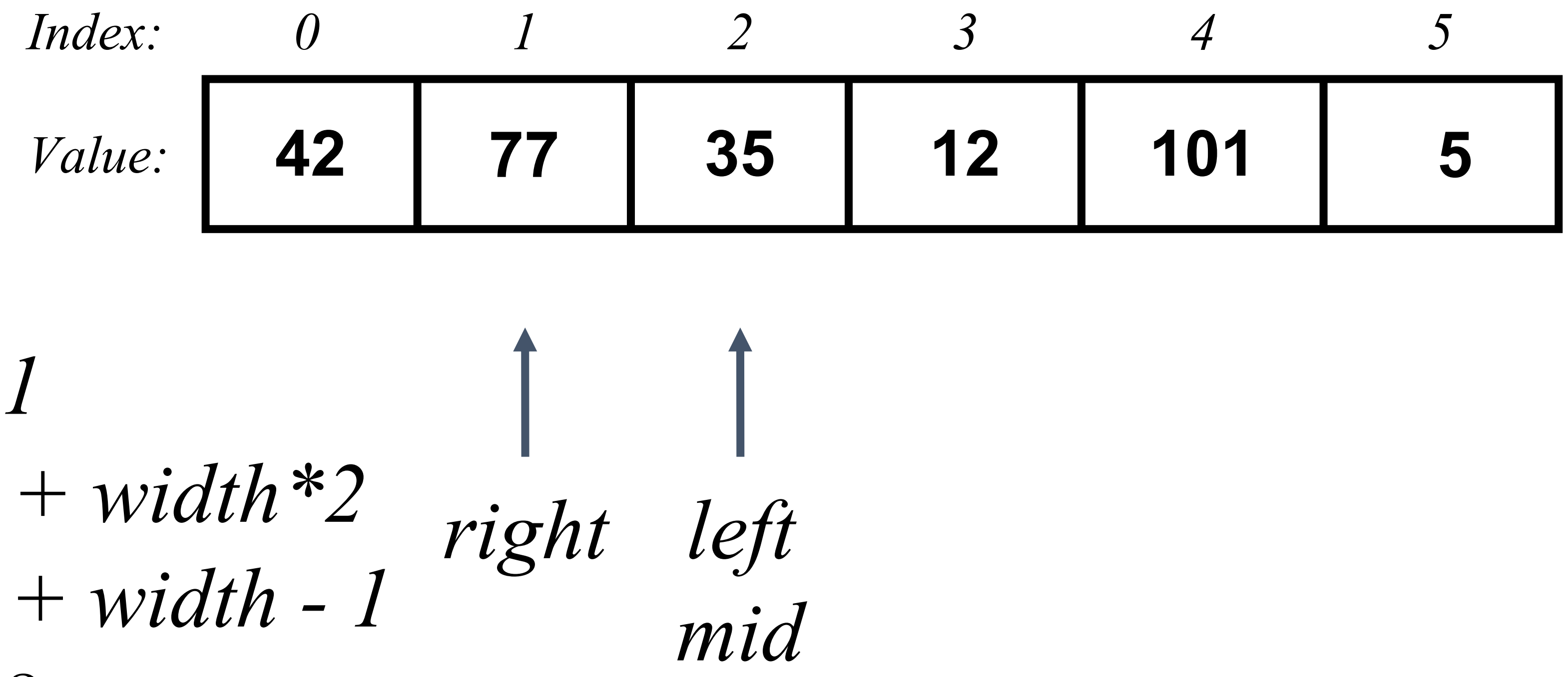
width = 1

*left = left + width*2*

mid = ?

right = ?

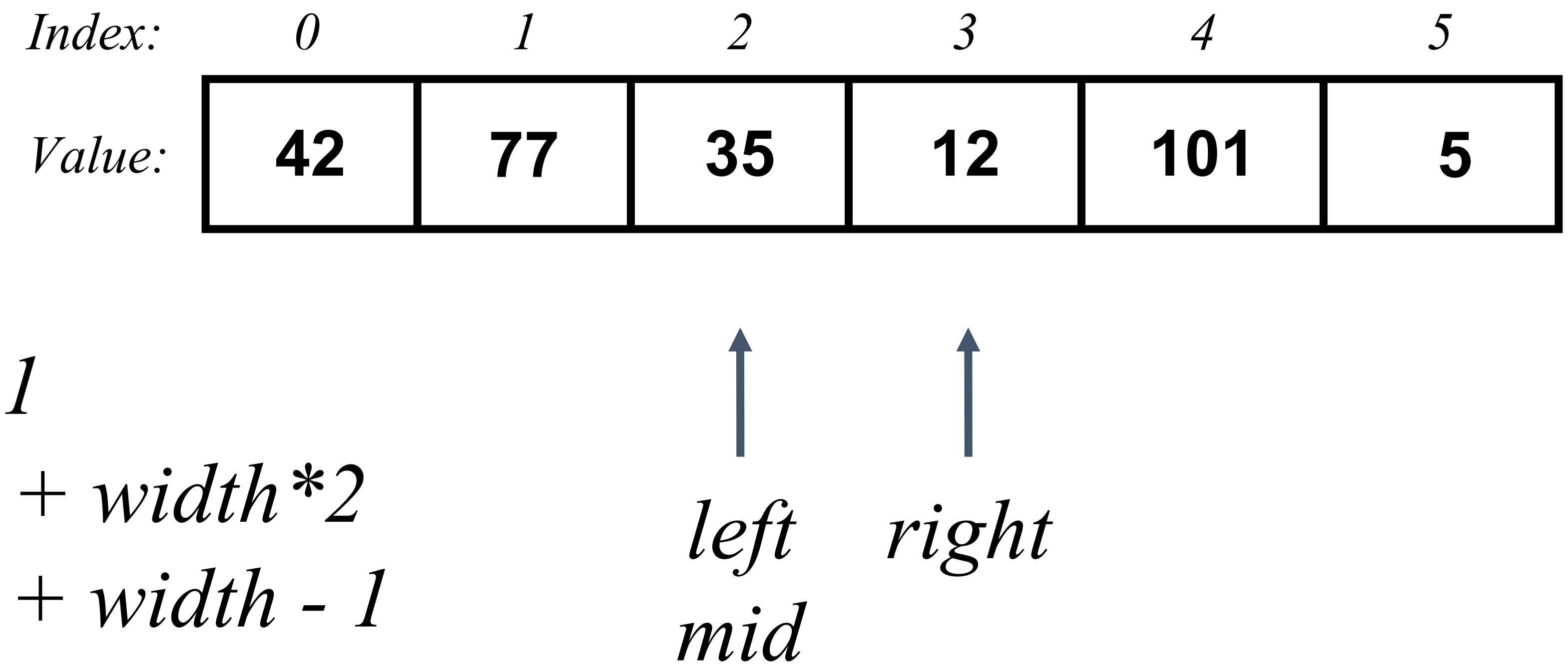
Merge Sort: An iterative solution



Index:	0	1	2	3	4	5
Value:	**42**	**77**	**35**	**12**	**101**	**5**

width = 1
*left = left + width*2*
mid = left + width - 1
right = ?

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**35**	**12**	**101**	**5**

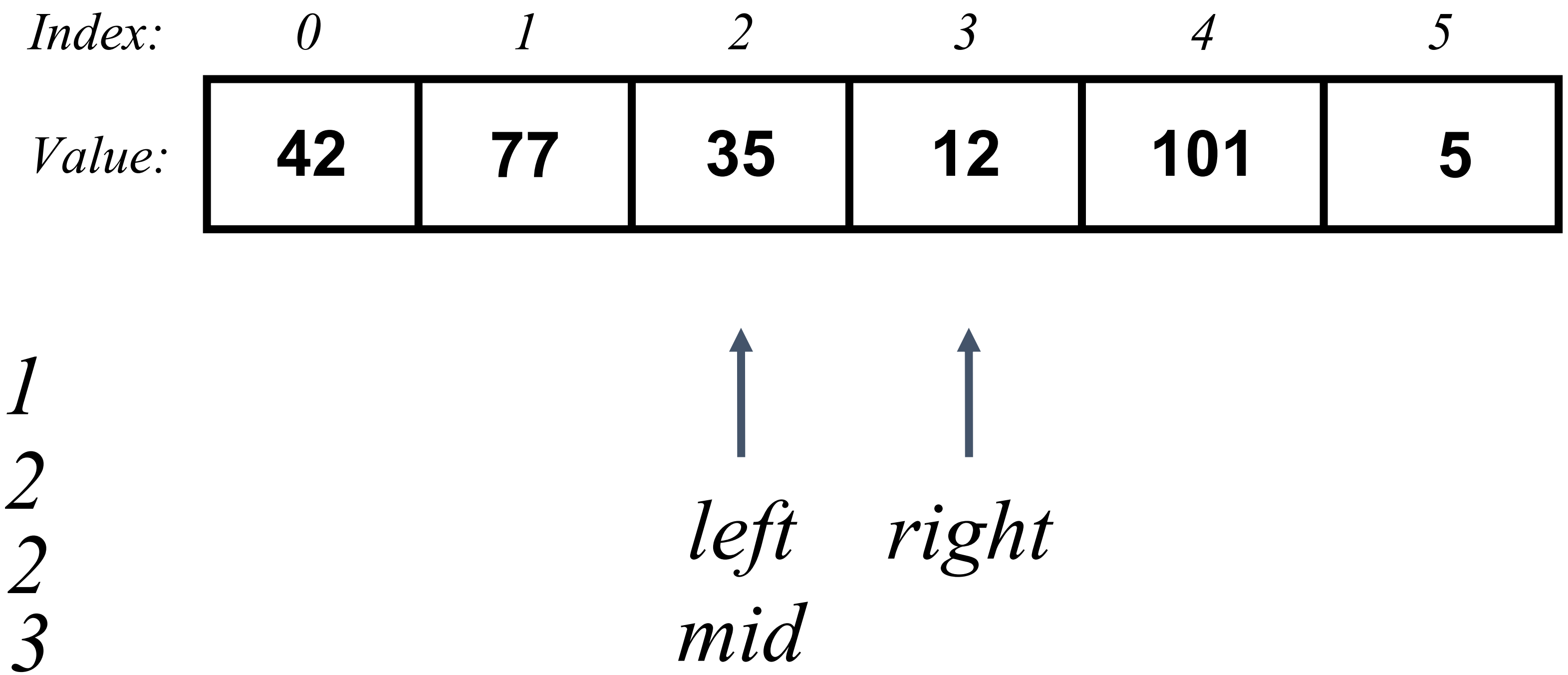
width = 1

*left = left + width*2*

mid = left + width - 1

*right = left + (width*2 -1)*

Merge Sort: An iterative solution



Index:	0	1	2	3	4	5
Value:	**42**	**77**	**35**	**12**	**101**	**5**

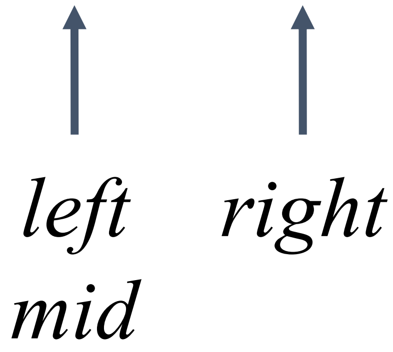
left *right*
mid

width = 1
left = 2
mid = 2
right = 3

Merge Sort: An iterative solution

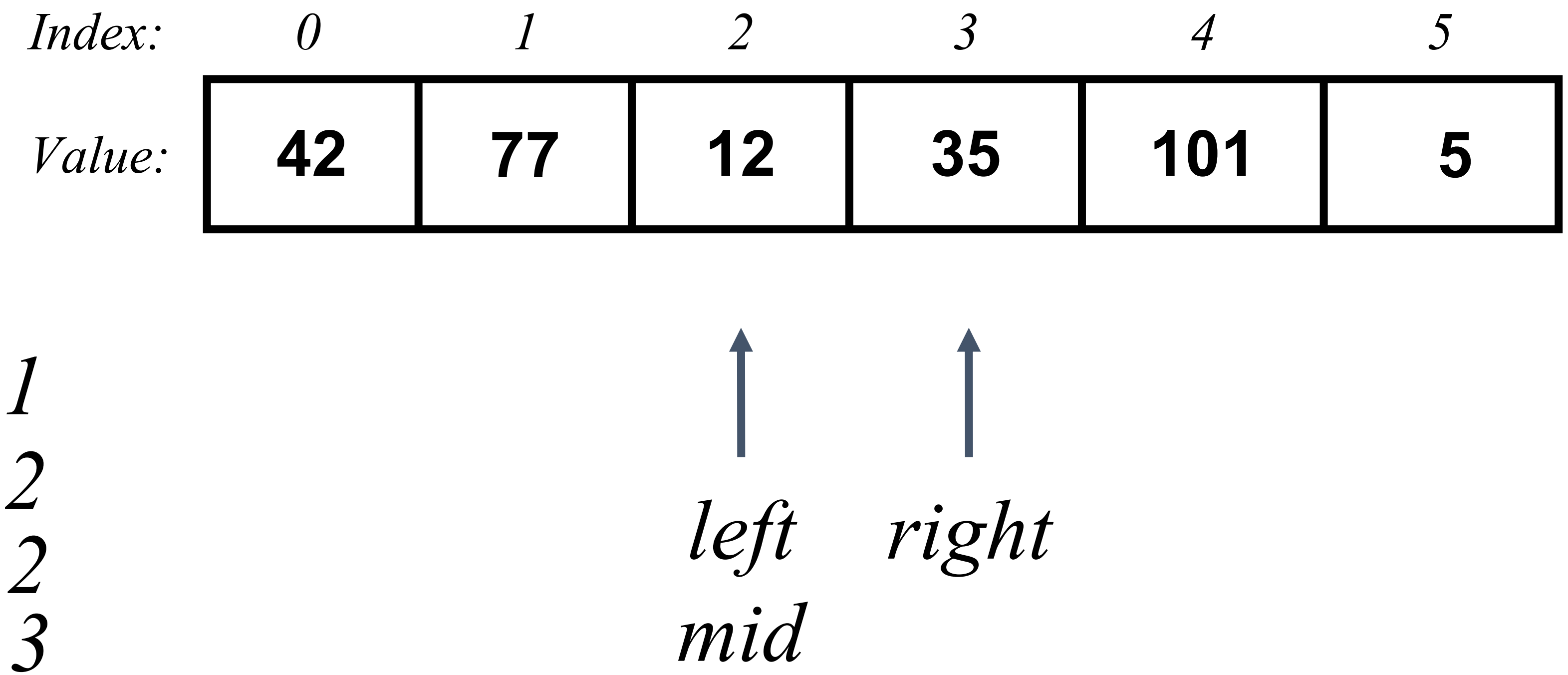
Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**35**	**12**	**101**	**5**

$width = 1$
$left = 2$
$mid = 2$
$right = 3$



Merge Again!

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**12**	**35**	**101**	**5**

left *right*

mid

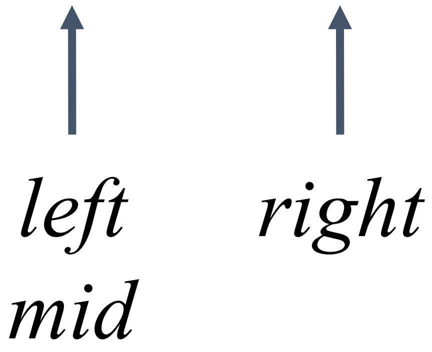
$width = 1$
$left = 2$
$mid = 2$
$right = 3$

Merge Again!

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**12**	**35**	**101**	**5**

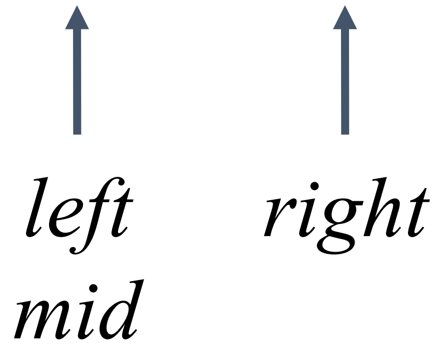
width = *1*
left = *4*
mid = *4*
right = *5*



Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**12**	**35**	**5**	**101**

width = *1*
left = *4*
mid = *4*
right = *5*



left
mid

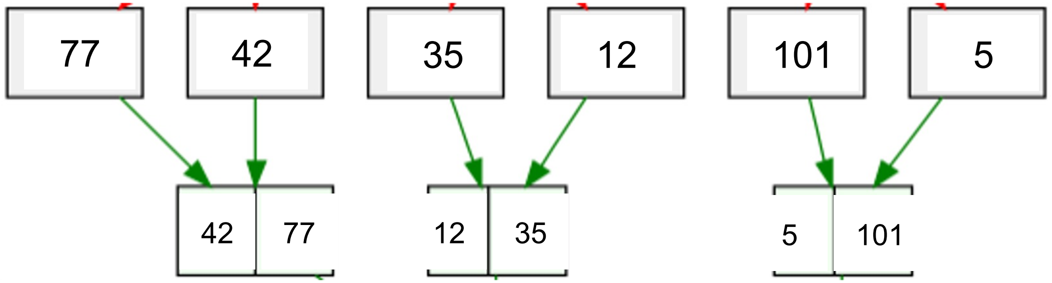
right

Merge Again!

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**12**	**35**	**5**	**101**

We just solved the problem for sub list of length 1 (width)



Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**12**	**35**	**5**	**101**

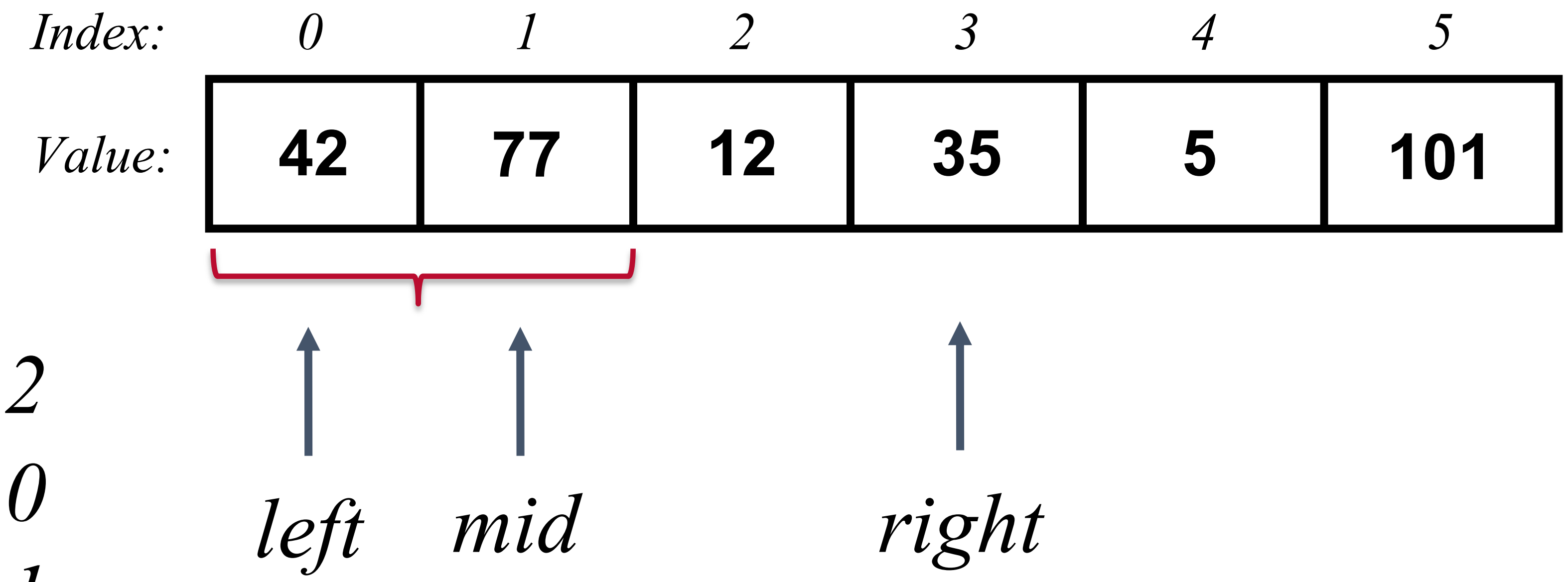
Now we have to solve the problem for width = 2

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**12**	**35**	**5**	**101**

Let’s start computing width, left, mid and right

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**42**	**77**	**12**	**35**	**5**	**101**

$width = 2$

$left = 0$

$mid = 1$

$right = 3$

Merge Sort: An iterative solution

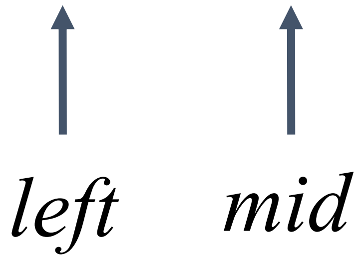
Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**12**	**35**	**42**	**77**	**5**	**101**

$width = 2$
$left = 0$
$mid = 1$
$right = 3$

Merge!

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**12**	**35**	**42**	**77**	**5**	**101**



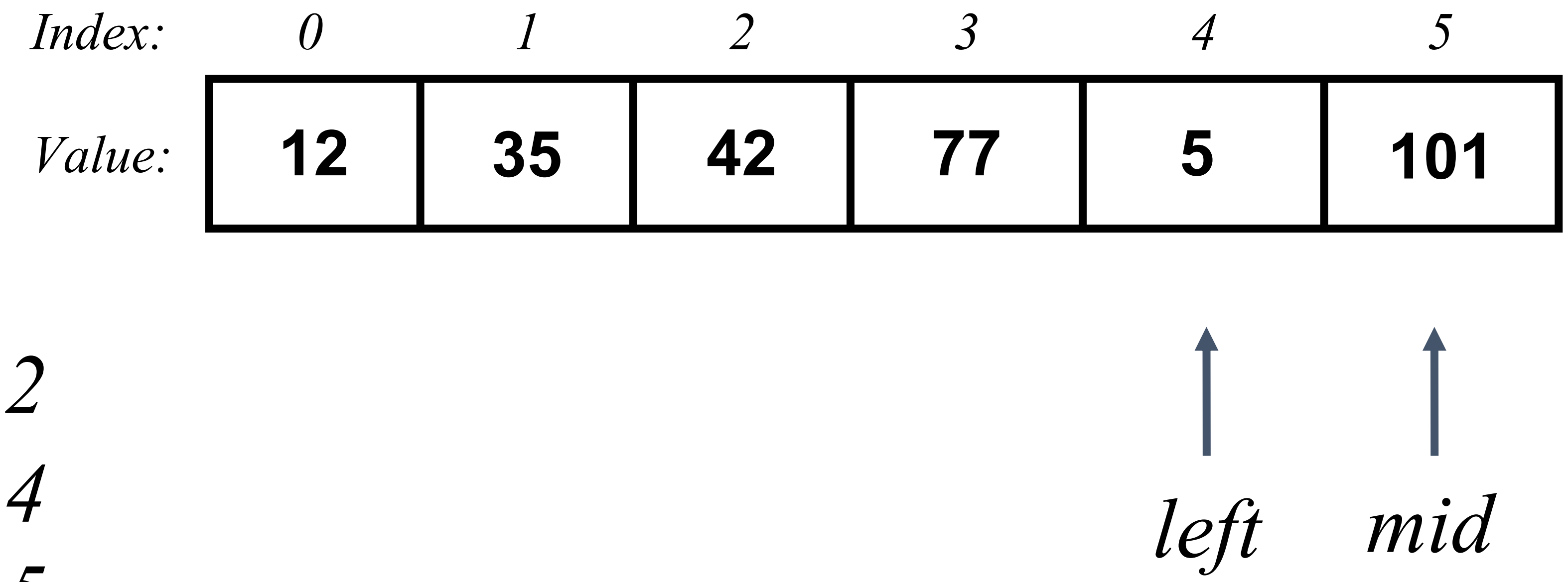
$width = 2$

$left = 4$

$mid = 5$

$right = 7$

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**12**	**35**	**42**	**77**	**5**	**101**

$width = 2$

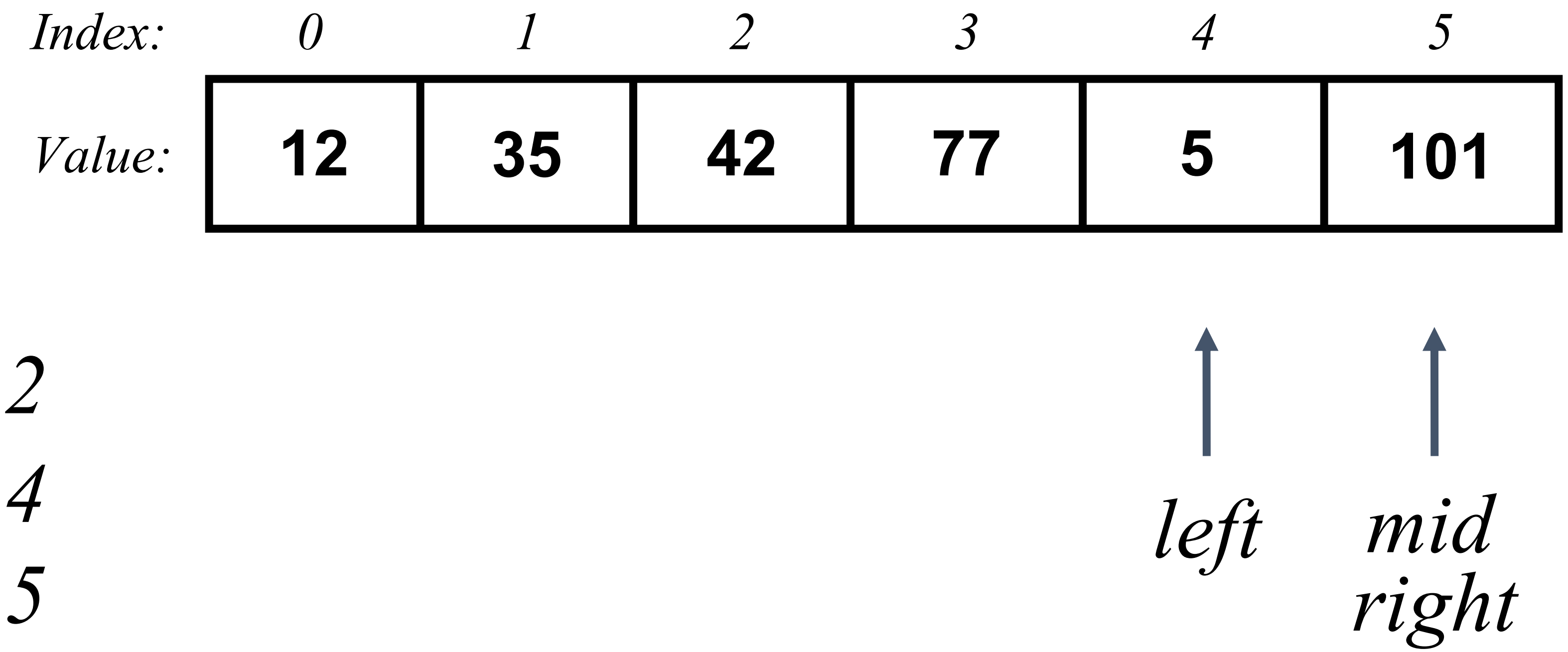
$left = 4$

$mid = 5$

$right = 7$

right is out of bound!

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**12**	**35**	**42**	**77**	**5**	**101**

$width = 2$

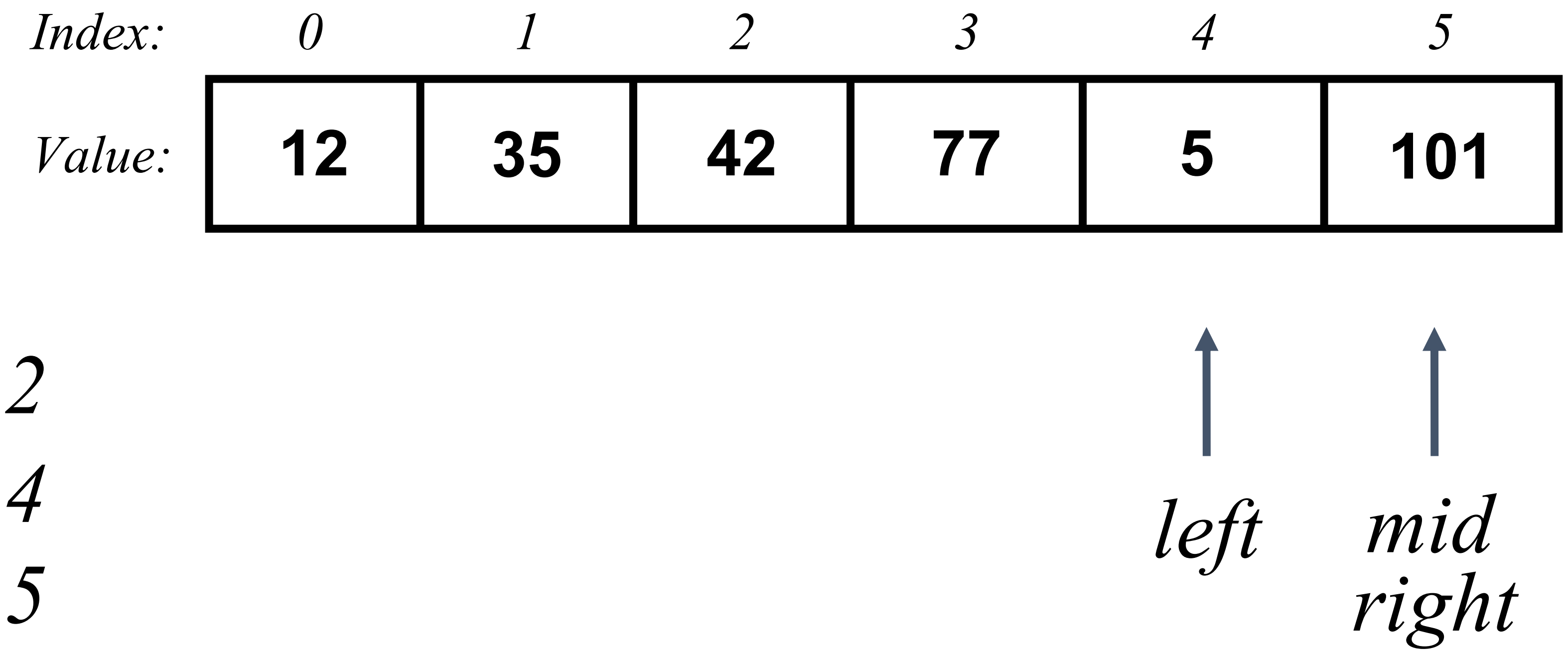
$left = 4$

$mid = 5$

$right = 5$

We set is as the list lenght in this case 5

Merge Sort: An iterative solution



Index:	0	1	2	3	4	5
Value:	**12**	**35**	**42**	**77**	**5**	**101**

$width = 2$

$left = 4$

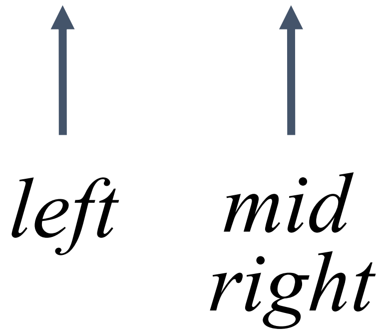
$mid = 5$

$right = 5$

In general every time an index goes out of bound we set it as the lenght of the list!

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**12**	**35**	**42**	**77**	**5**	**101**



$width = 2$

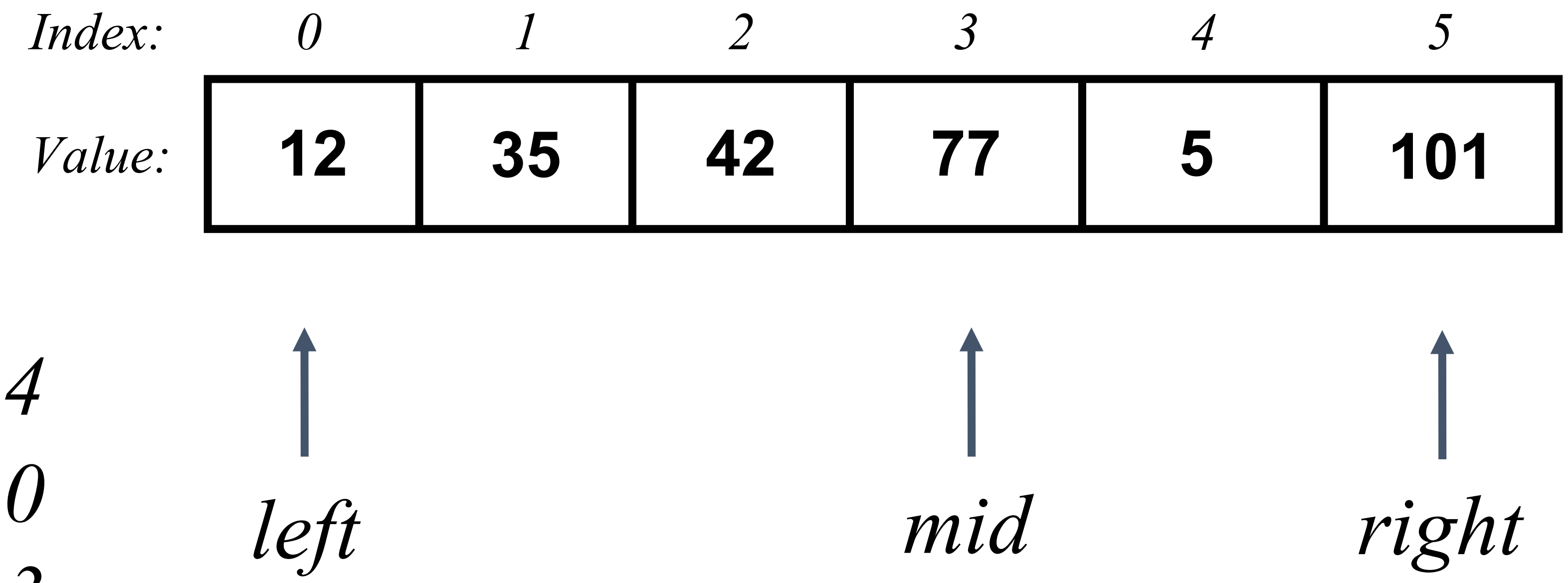
$left = 4$

$mid = 5$

$right = 5$

Now we merge again the sub lists

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**12**	**35**	**42**	**77**	**5**	**101**

$width = 4$

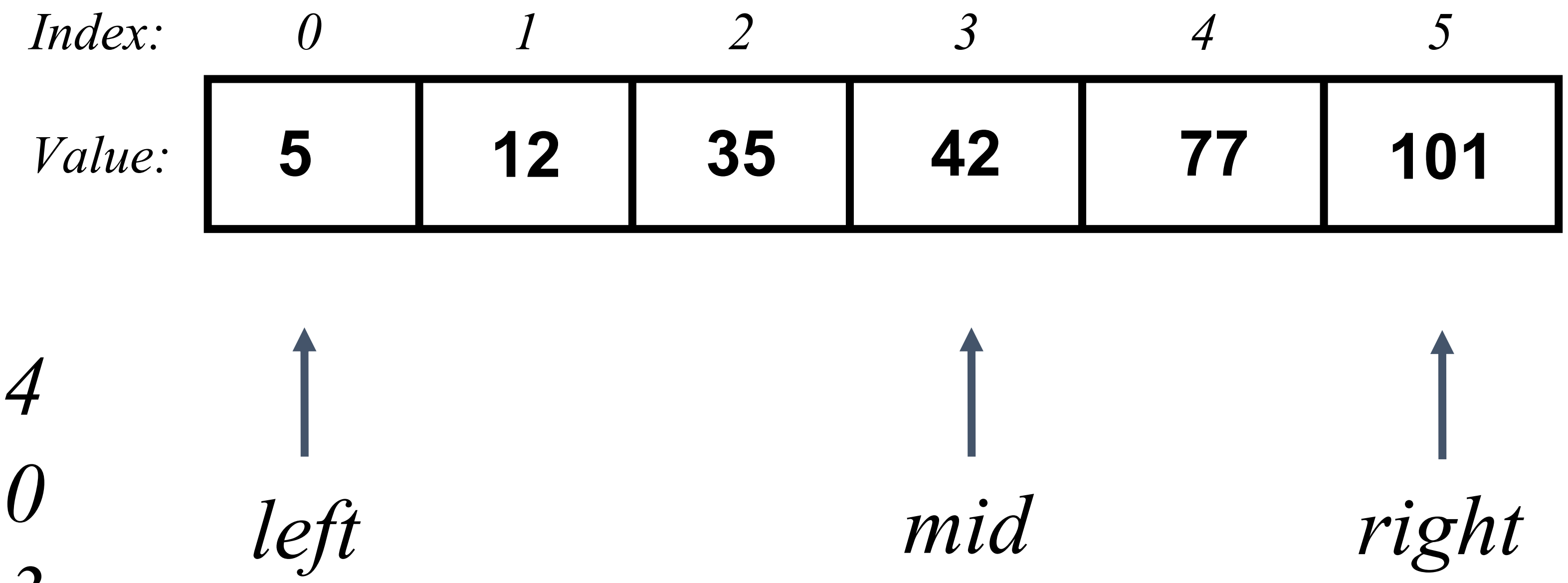
$left = 0$

$mid = 3$

$right = 5$

And perform the iteration 3

Merge Sort: An iterative solution



Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**5**	**12**	**35**	**42**	**77**	**101**

$width = 4$

$left = 0$

$mid = 3$

$right = 5$

Now we merge

Merge Sort: An iterative solution

Index:	*0*	*1*	*2*	*3*	*4*	*5*
Value:	**5**	**12**	**35**	**42**	**77**	**101**

And we are done! Why?

Because at the next iteration left becomes bigger than the list length so we exit the inner loop.

Than we double width but it is bigger than the list length too so the algorithm ends!

Merge Sort: An iterative solution

```
Procedure  iterative_mergesort(A: list):
    width = 1
    list_length = len(A)

    while (width < list_length):
        left = 0

        while (left < list_length):
            right = min(left + (width * 2 - 1), list_length - 1)
            middle = min(left + width - 1, list_length - 1)
            merge(A, left, middle, right)
            left += width*2
        width *= 2
    return A
```

Python Sort!

What is the algorithm behind python’s sorted?

Python Sort – TimSort (hybrid)

Official website: https://docs.python.org/3/library/functions.html

Idea:

- It takes an unsorted list and divides the elements in **“runs”**
- A small “run” is sorted by using the **insertion sort** algorithm**.**
- Eventually, it merges the sorted ”runs” (**Merge sort**).

Python Sort – TimSort (hybrid)

Official website: https://docs.python.org/3/library/functions.html

Idea: Based on **Insertion Sort** + **Merge Sort.**

Why we use the **Insertion Sort** if the **Merge sort** is asynthotically more efficient?

Python Sort – TimSort (hybrid)

Official website: https://docs.python.org/3/library/functions.html

Idea: Based on **Insertion Sort** + **Merge Sort.**

Why we use the **Insertion Sort** if the **Merge sort** is asynthotically more efficient?

Asymptotically faster means that there is a threshold **N** such that if n≥N then sorting n elements with **merge sort** is **faster** than with **insertion sort**

Python Sort – TimSort (hybrid)

Official website: https://docs.python.org/3/library/functions.html

Idea: Based on **Insertion Sort** + **Merge Sort.**

Why we use the **Insertion Sort** if the **Merge sort** is asynthotically more efficient?

Computational Complexity $O(n \log n)$
Space Complexity $O(n)$

Python Sort - Comparison

Num Items	Mergesort	TimSort
16,000	0.002	0.003
32,000	0.003	0.002
64,000	0.008	0.004
128,000	0.015	0.009
256,000	0.034	0.018
512,000	0.068	0.040
1,024,000	0.143	0.082
2,048,000	0.296	0.184
4,096,000	0.659	0.383
8,192,000	1.372	**0.786**

https://www.cs.utexas.edu/~scottm/cs314/handouts/slides/Topic17FastSorting.pptx