its ICTAcademy

INFORMATION AND COMMUNICATIONS TECHNOLOGY

MergeSort

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.

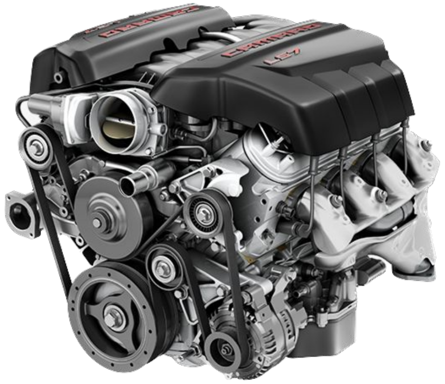
For example, if you have to build a car from scratch you start decomposing the problem into smaller problems:

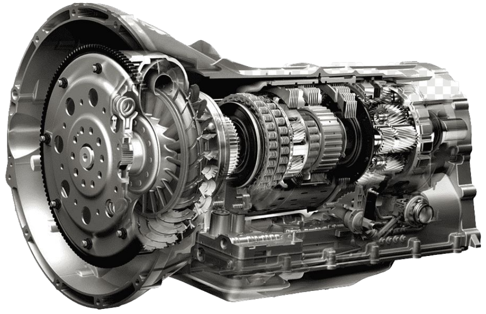
A new paradigm - Divide and Conquer *(and Combine)*

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.

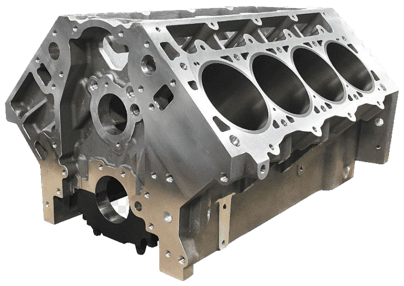
In turn each one of these problems can be split into smaller problems:

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.

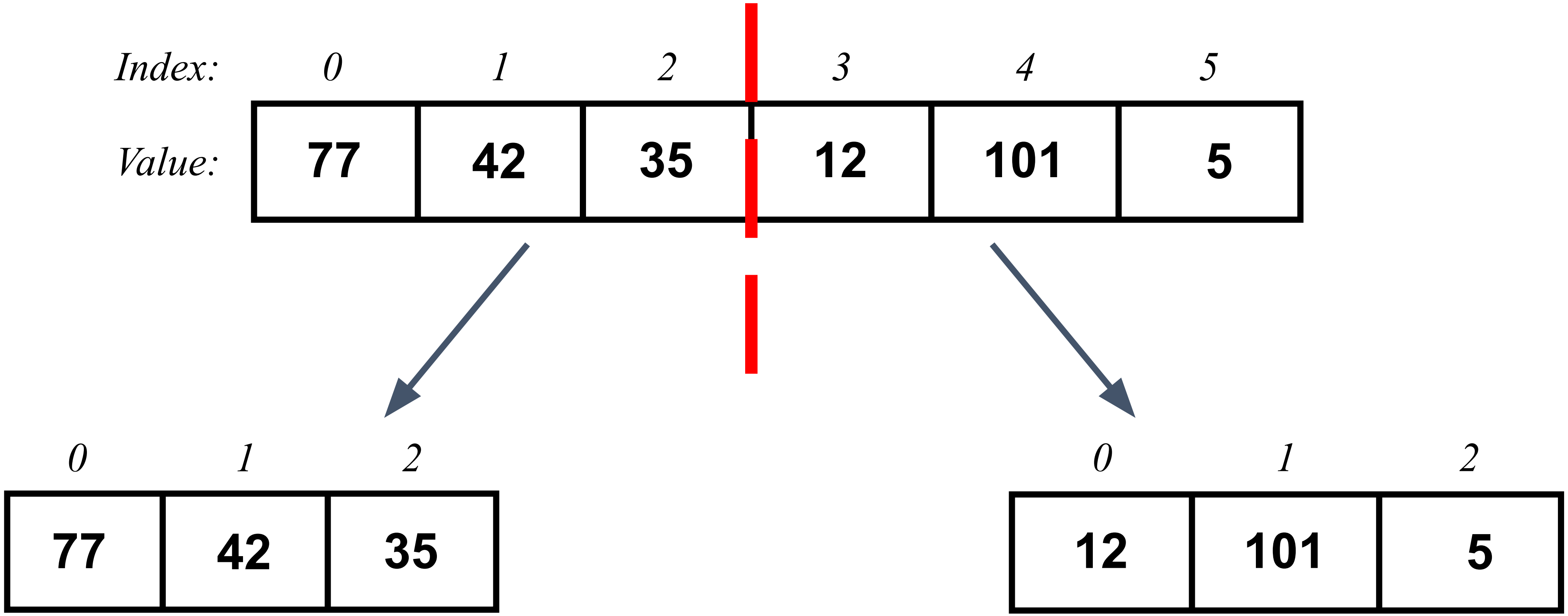
Merge Sort: an example

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

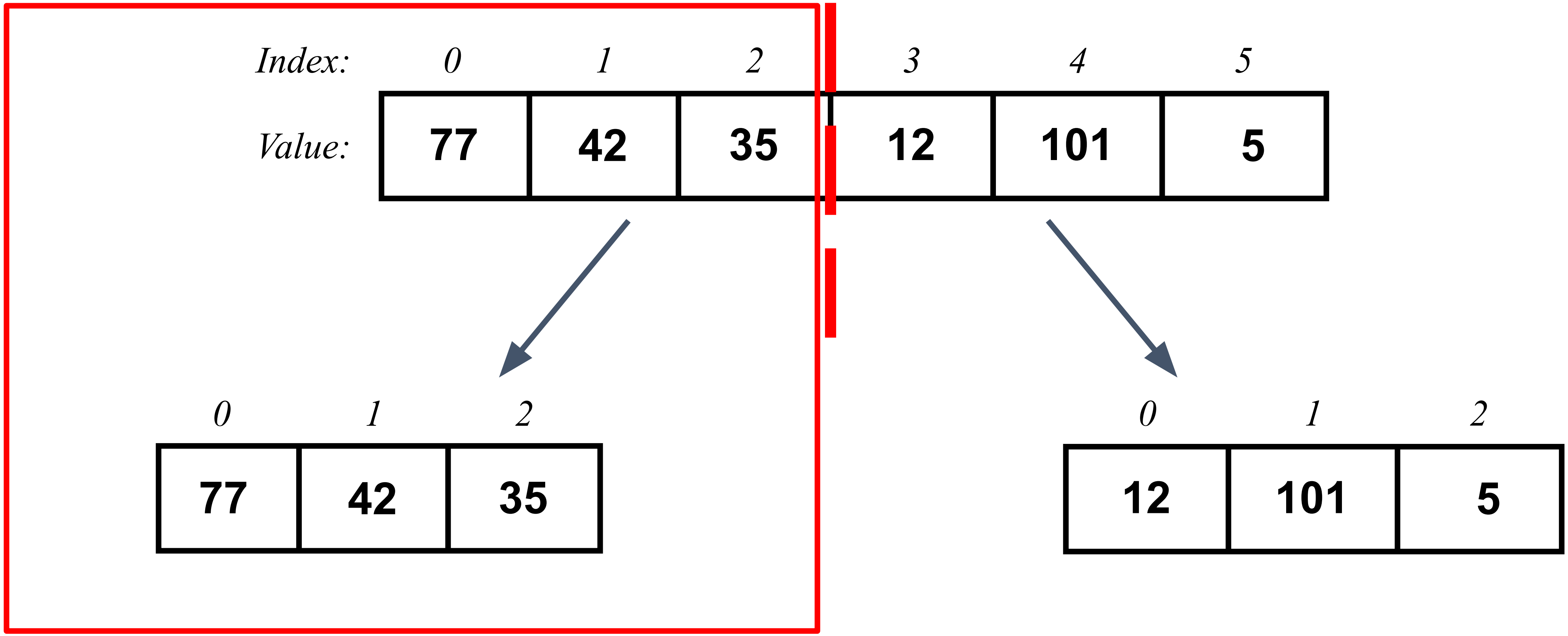
Merge Sort: an example

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

Merge Sort: an example



Merge Sort: an example



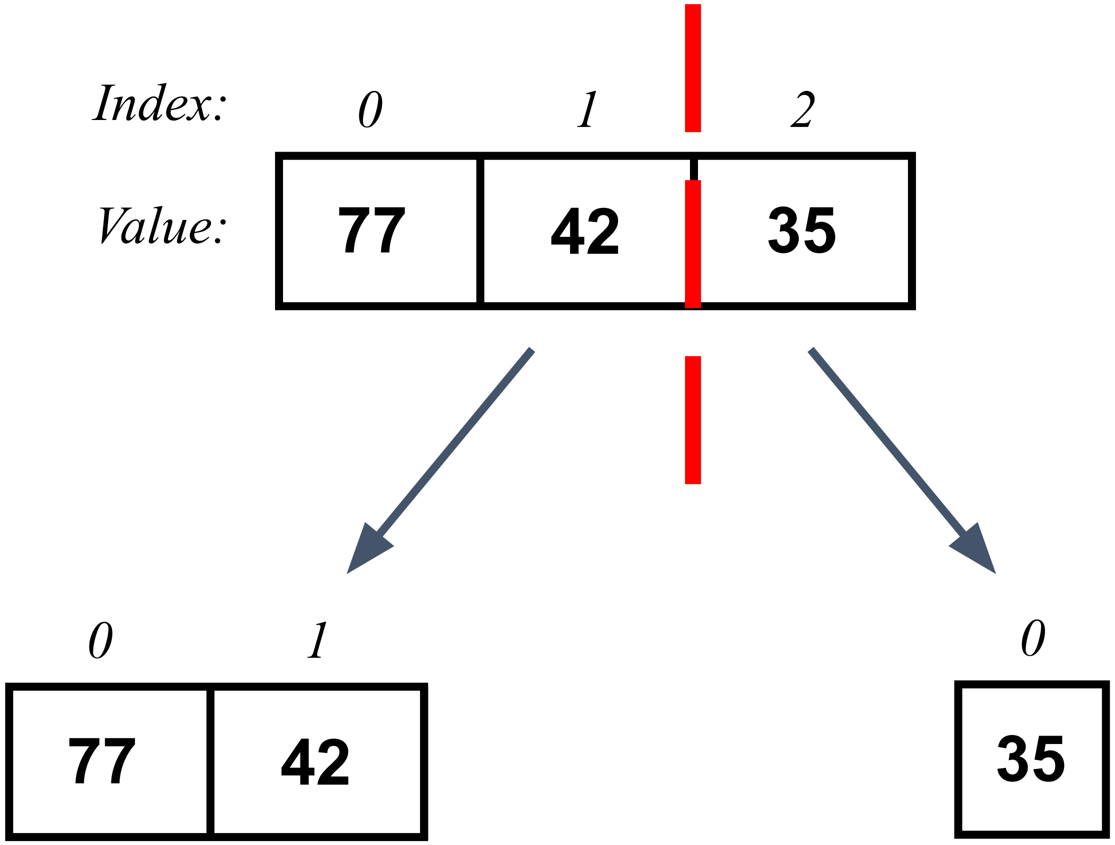
Merge Sort: an example

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

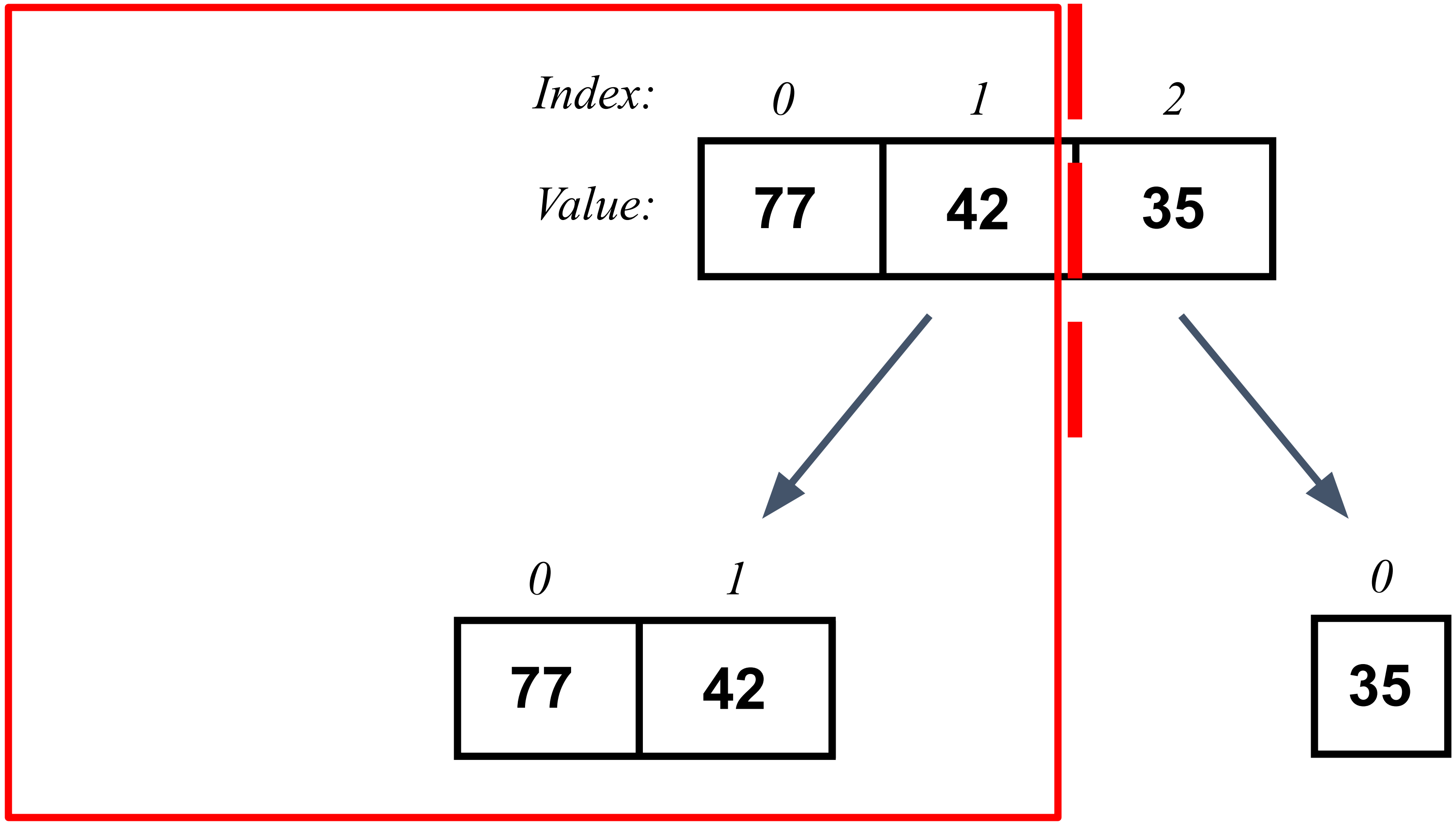
Merge Sort: an example

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

Merge Sort: an example



Merge Sort: an example



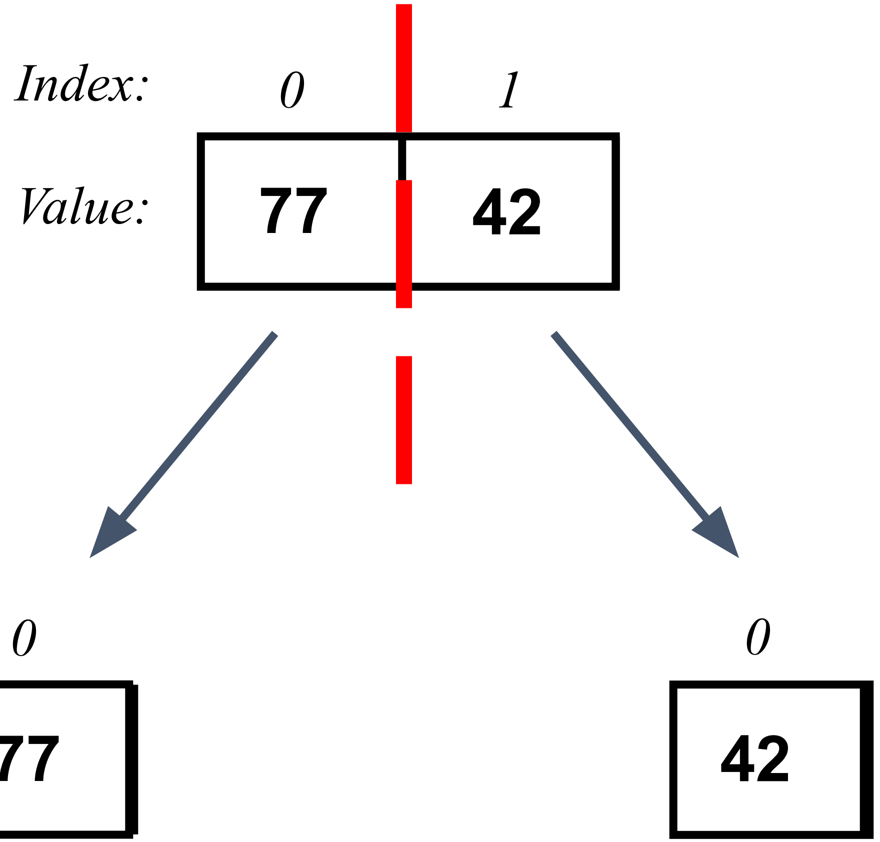
Merge Sort: an example

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

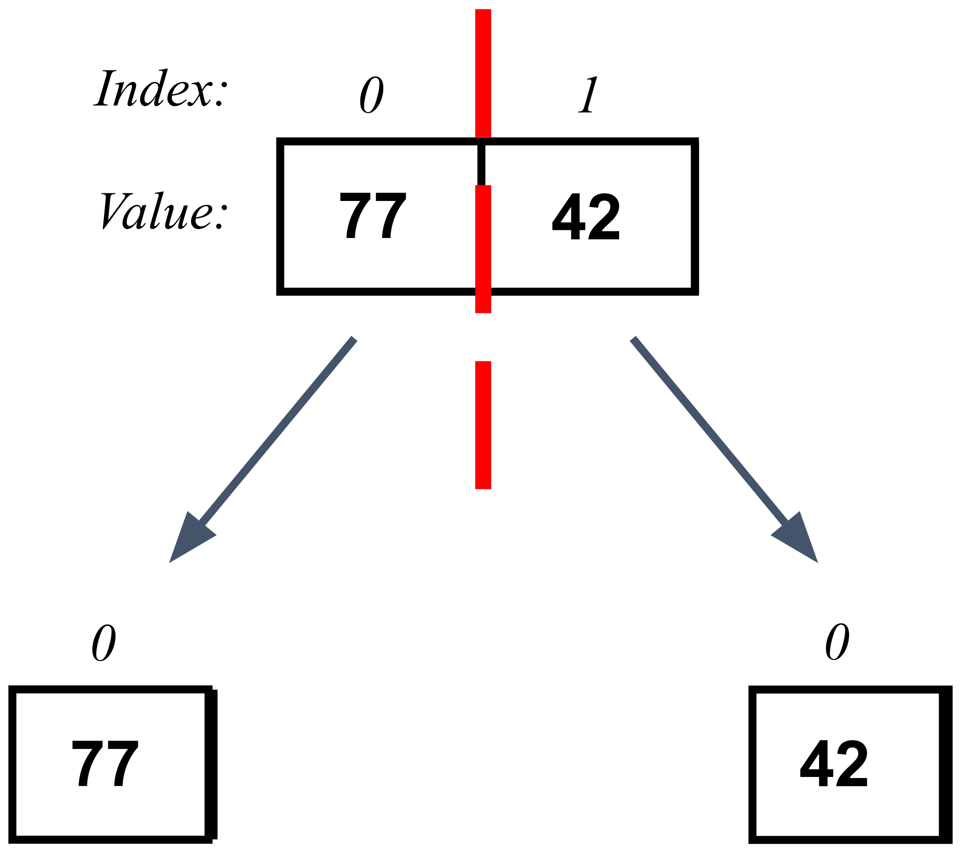
Merge Sort: an example

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

Merge Sort: an example



Merge Sort: an example



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		*0*
77		**42**

Now we have to join the results!

How?

Merge Sort: an example

0	0
77	**42**

Now we have to join the results!

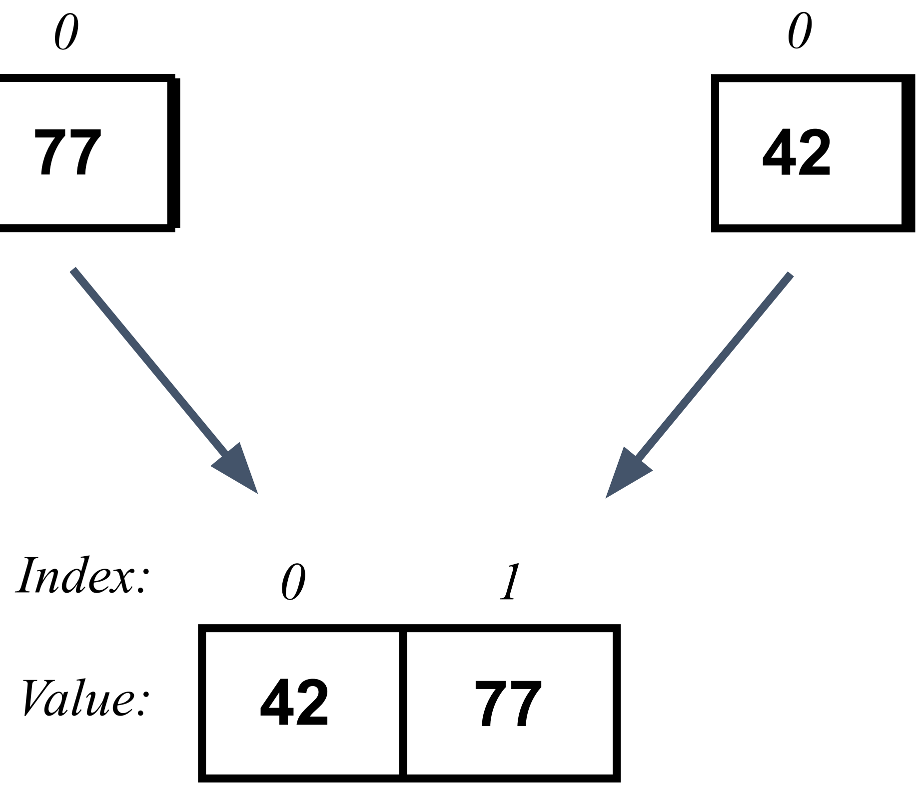
How?
We can check which element is the smaller one between the two and put it into the position 0 while the other one into position 1

Merge Sort: an example

0	0
77	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



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.

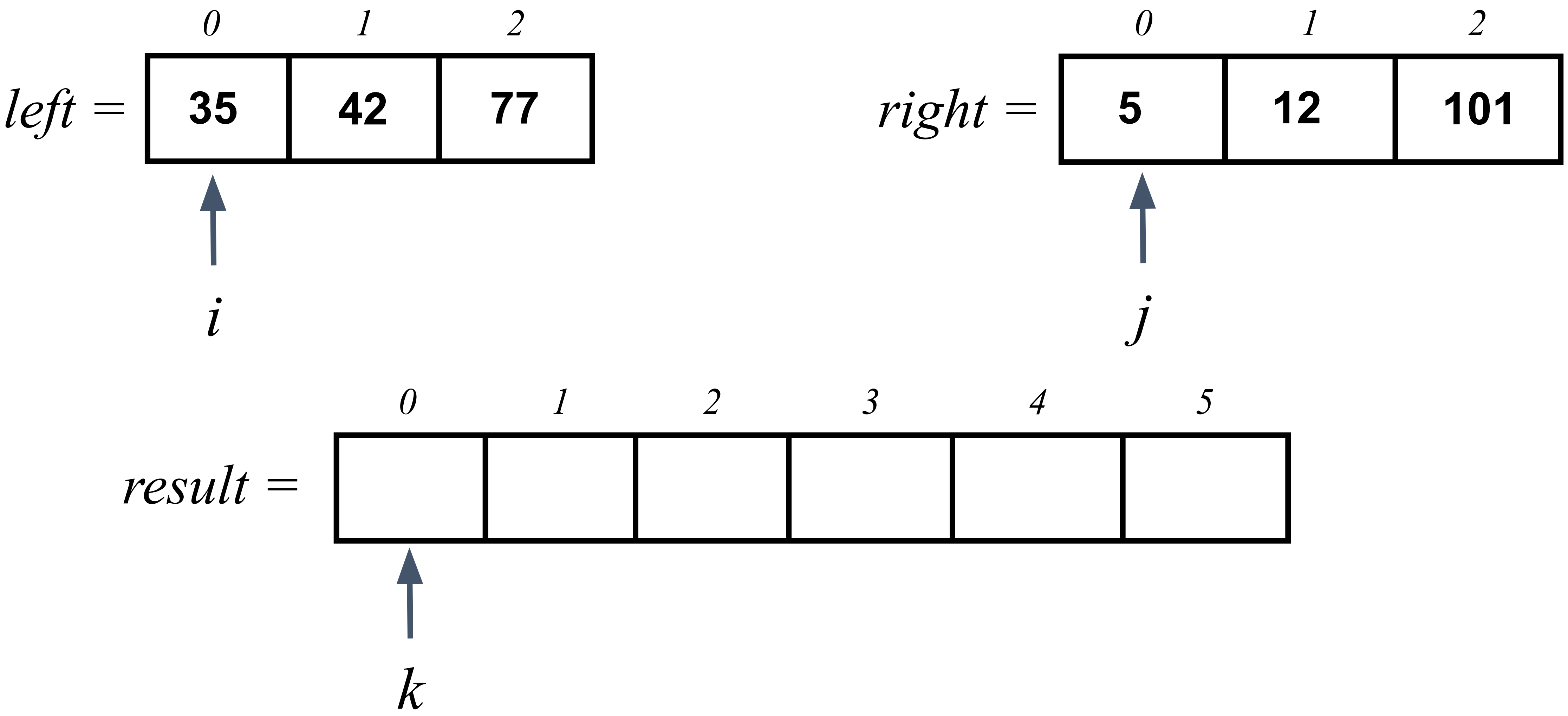
Merge Sort: the merge procedure

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!

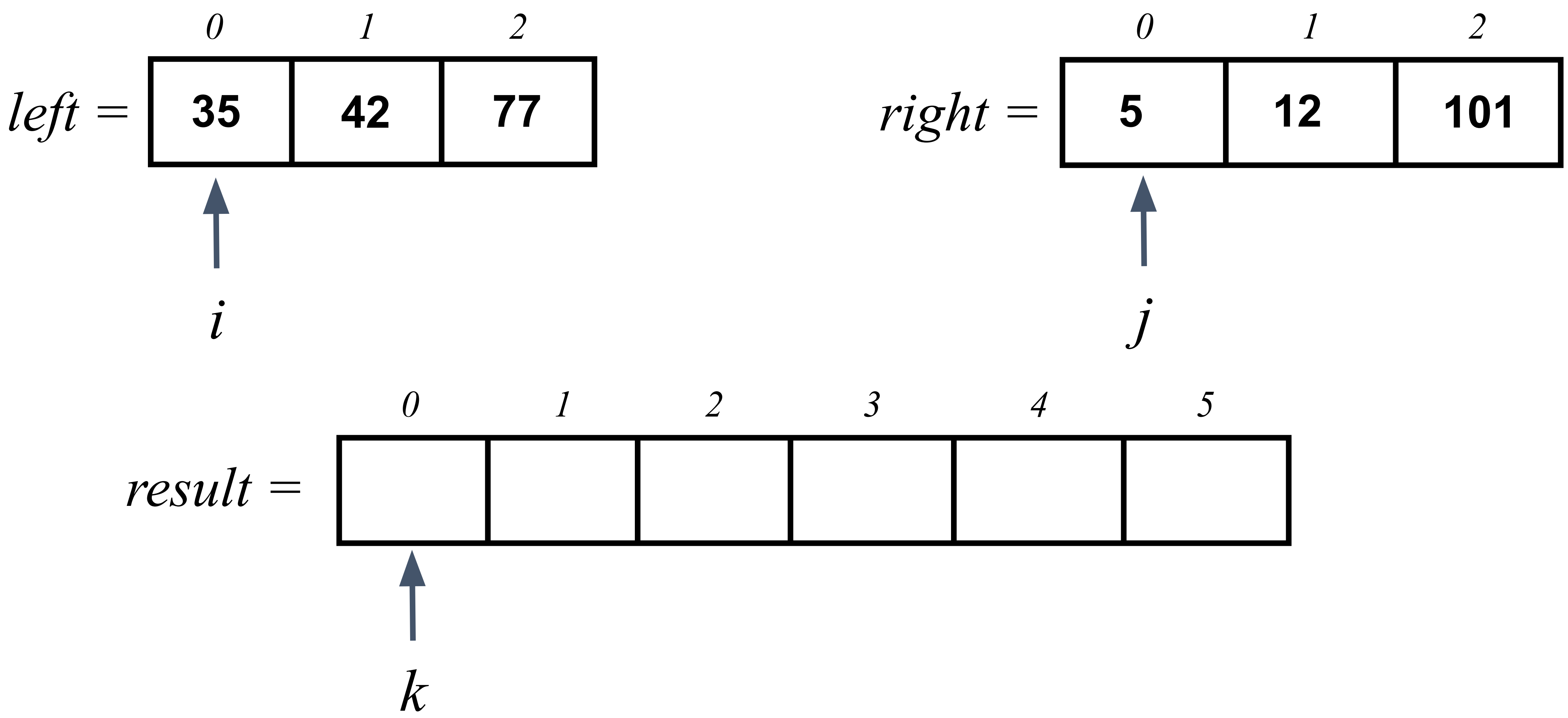
Merge Sort: the merge procedure

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



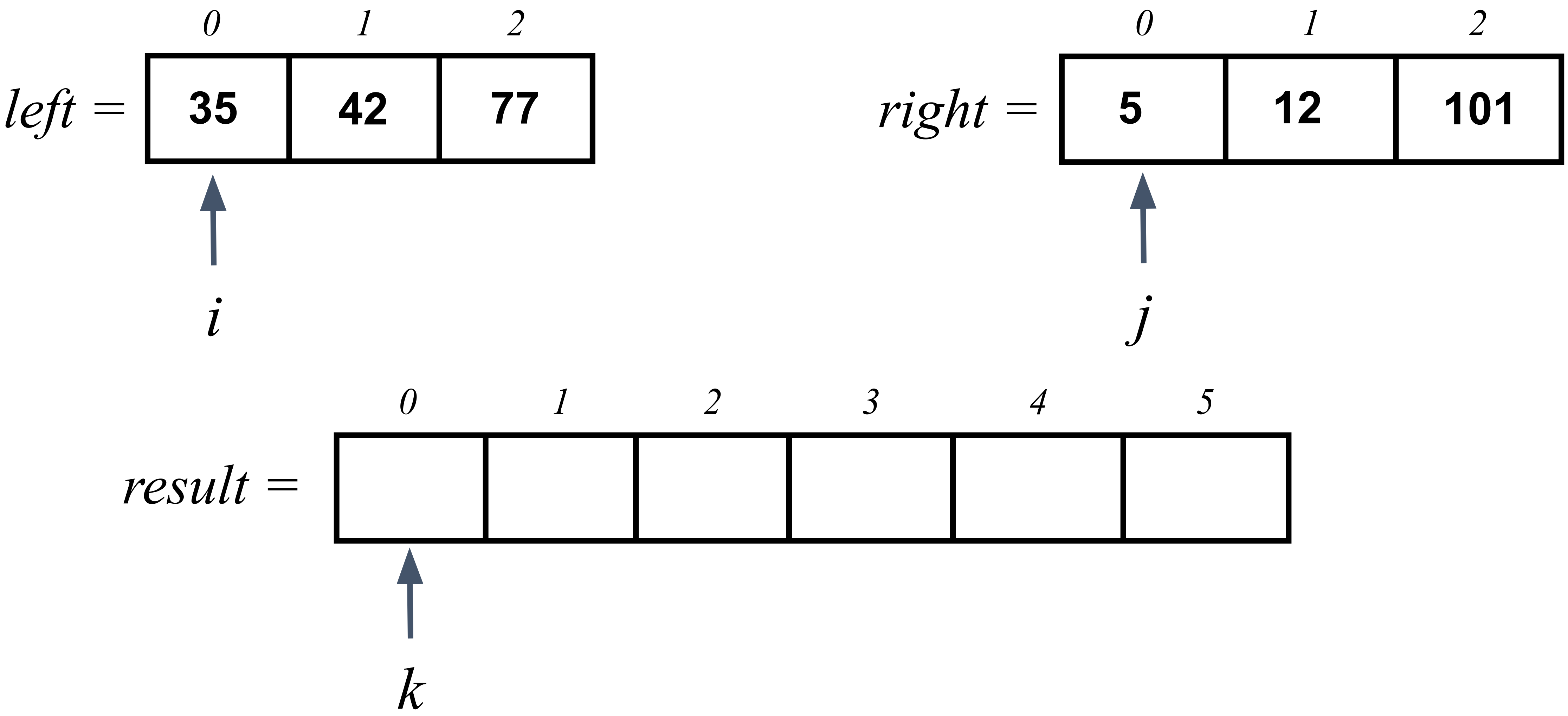
Merge Sort: the merge procedure

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



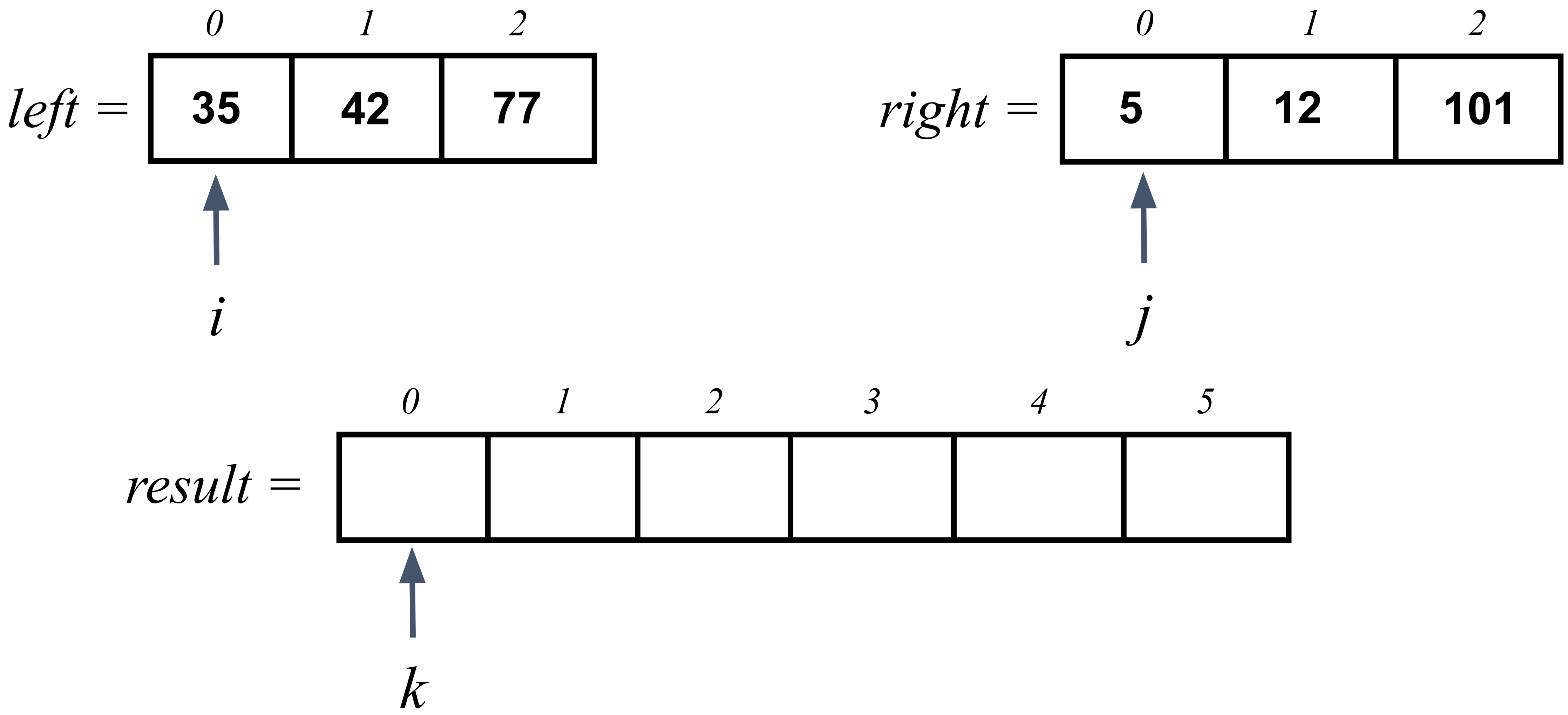
Merge Sort: the merge procedure

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



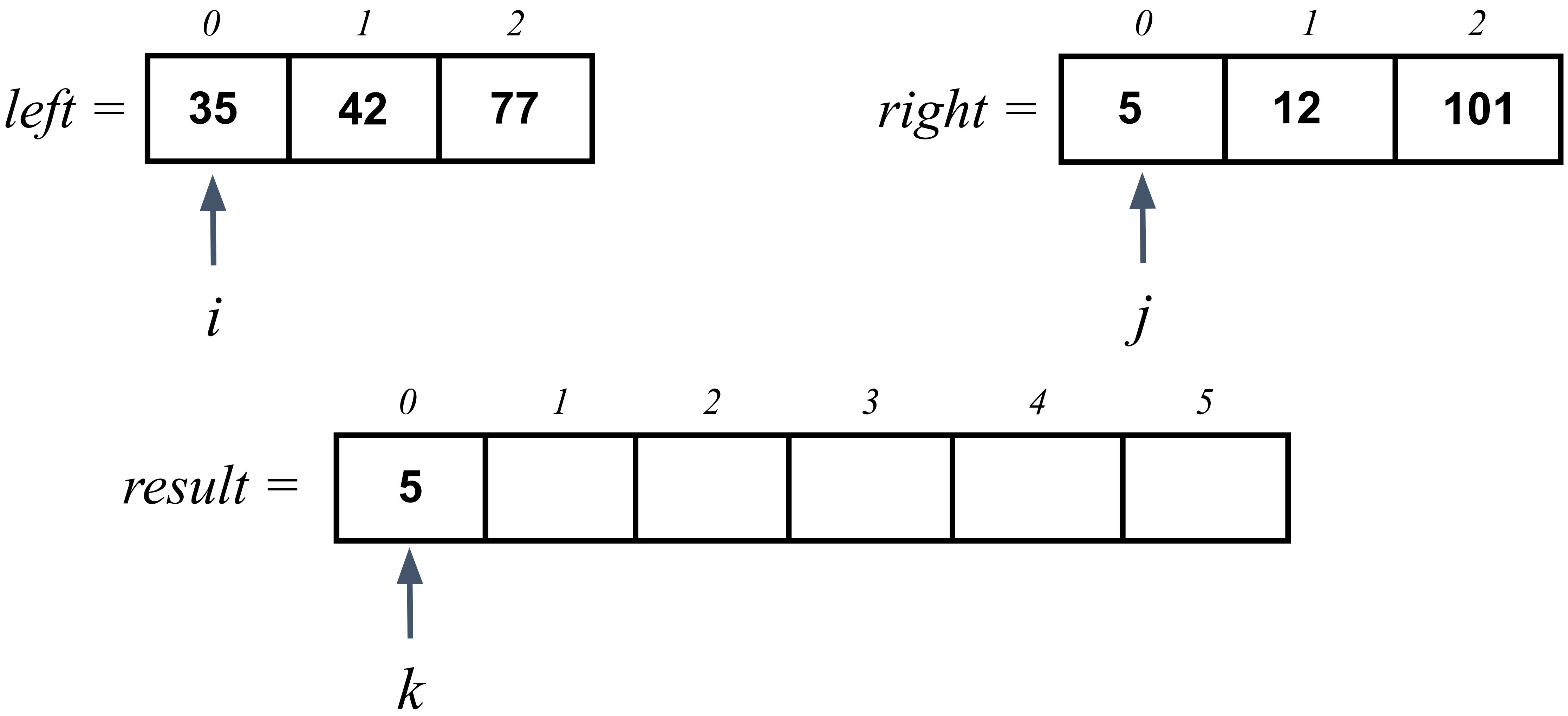
Merge Sort: the merge procedure

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



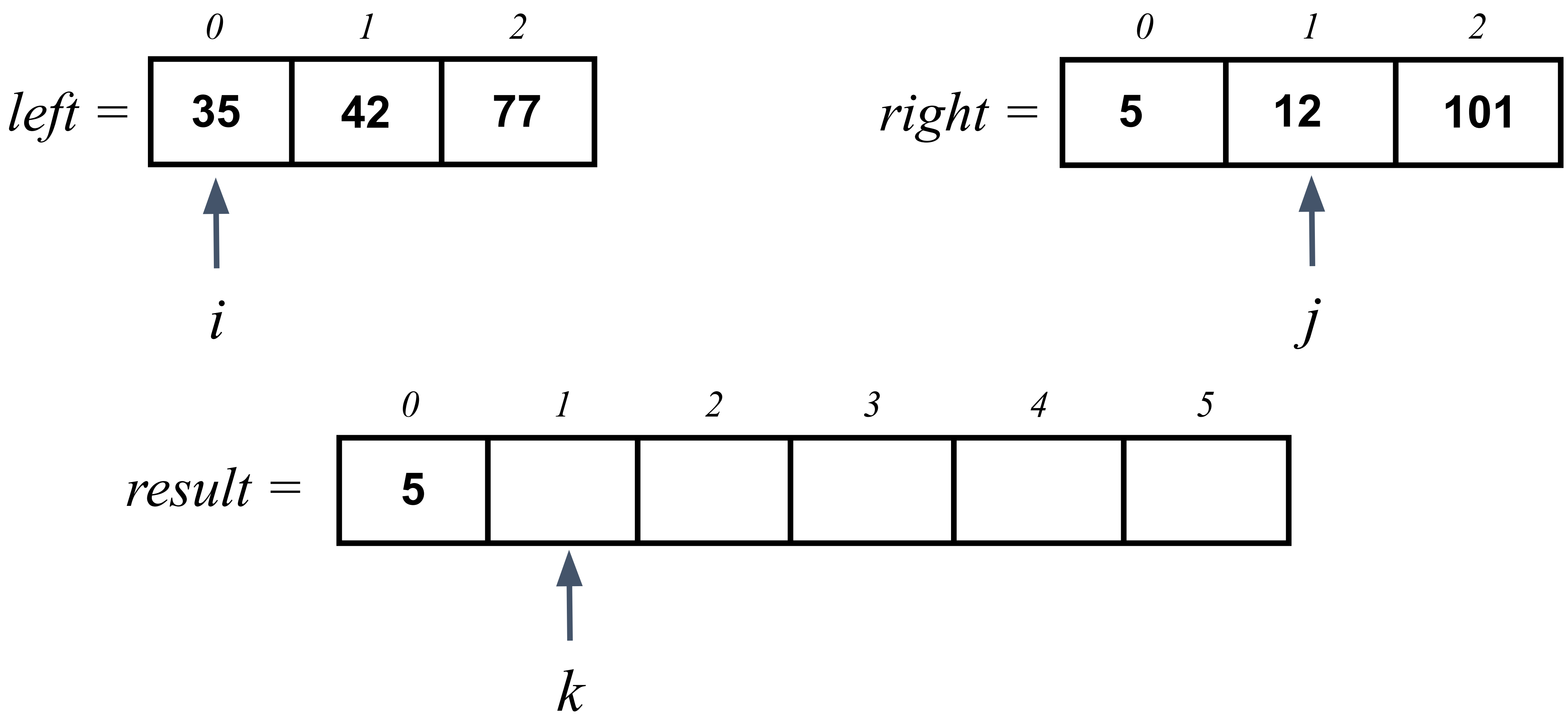
Merge Sort: the merge procedure

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



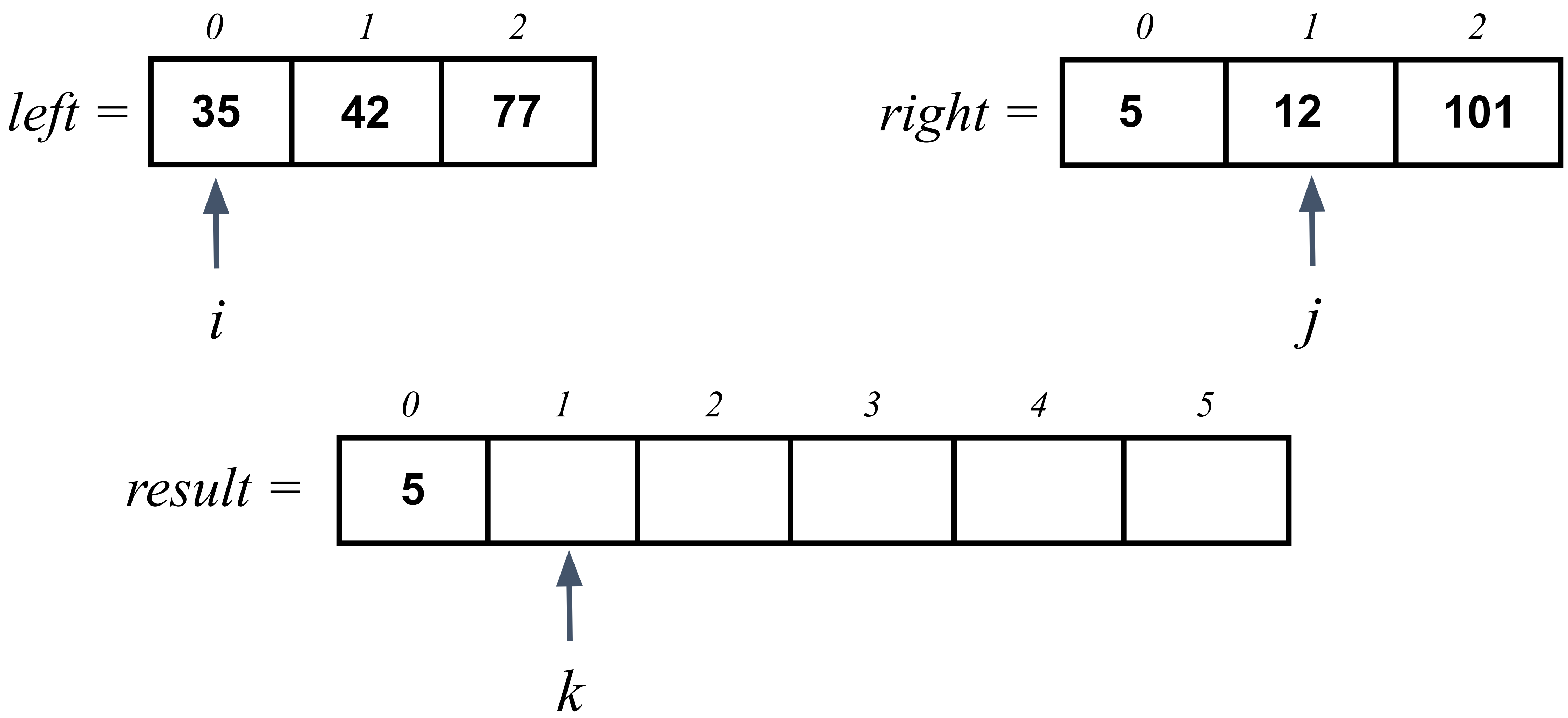
Merge Sort: the merge procedure

Then we increase the indices j and k



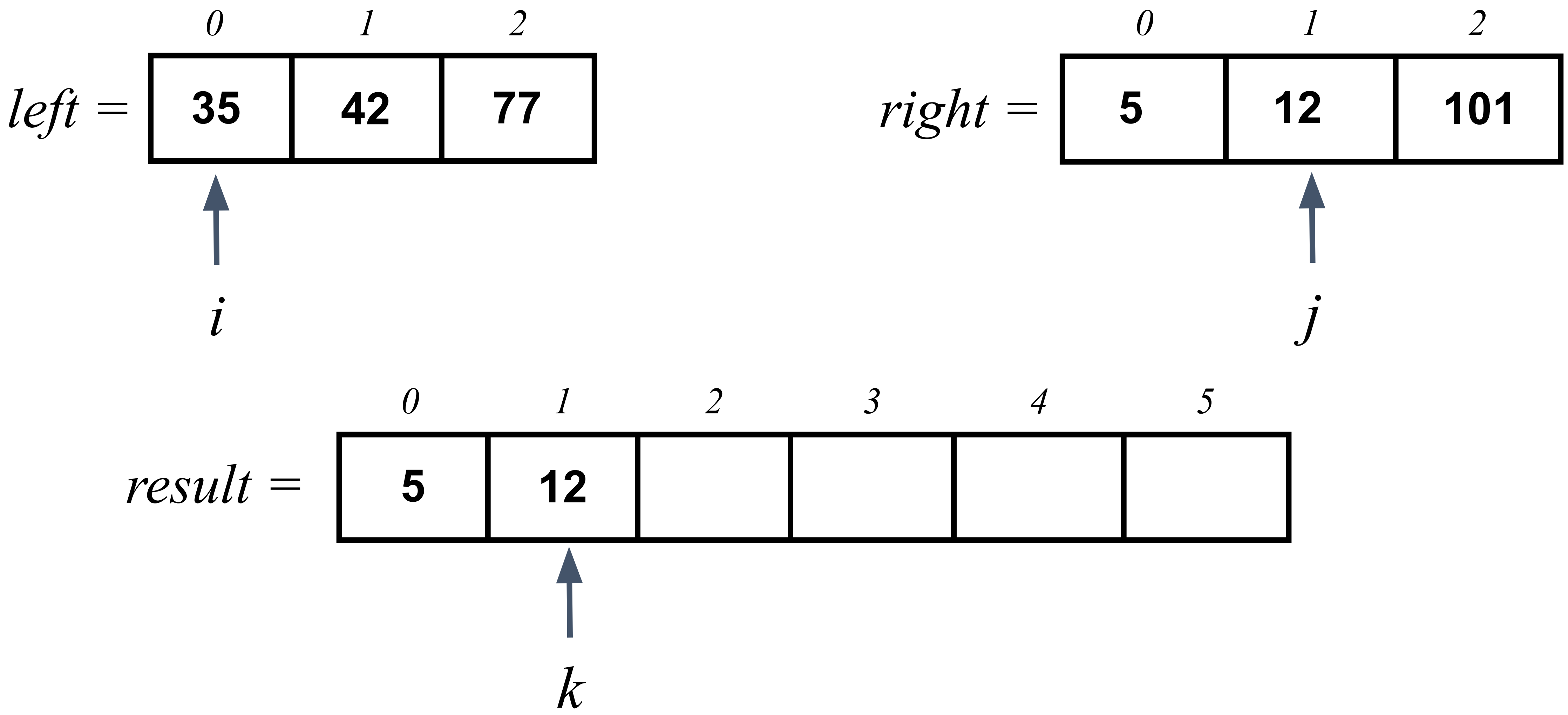
Merge Sort: the merge procedure

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



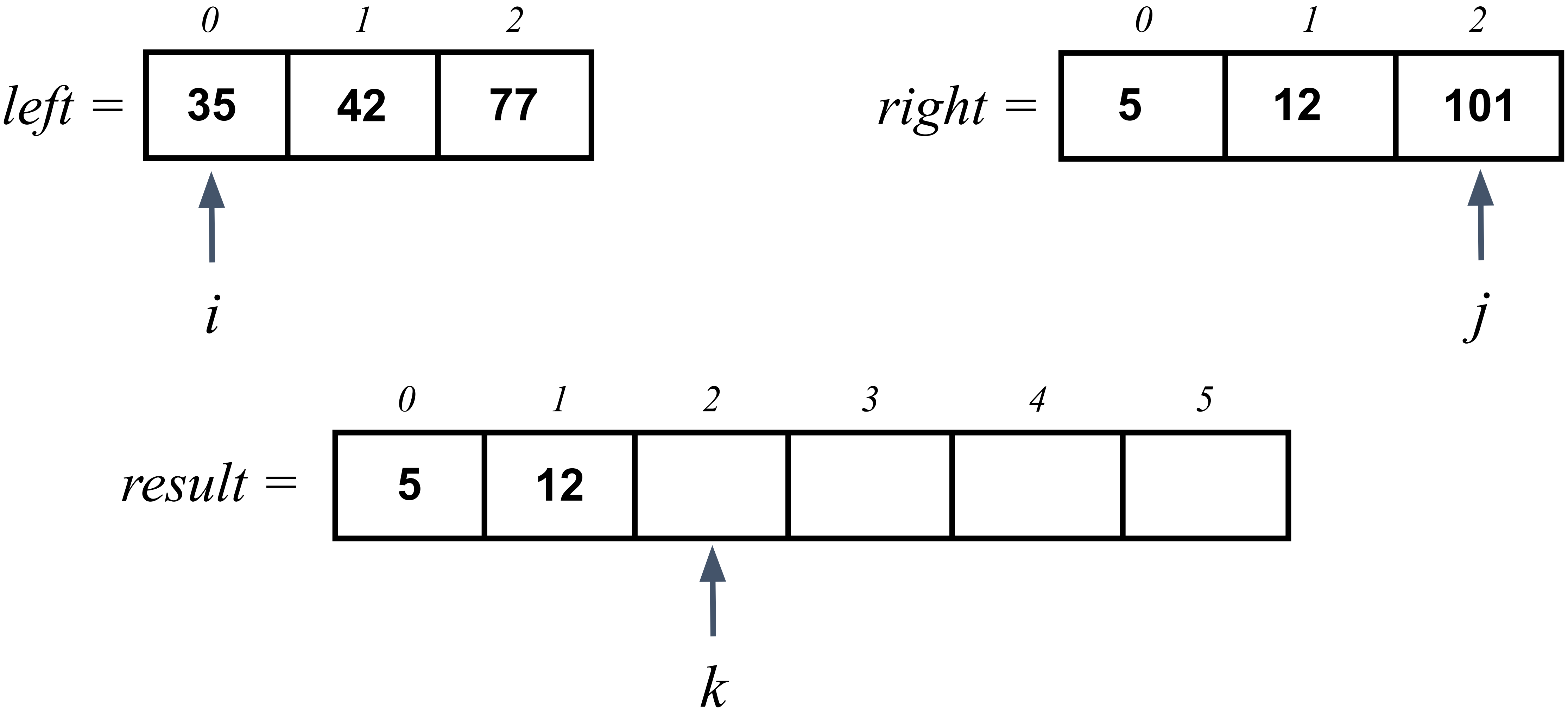
Merge Sort: the merge procedure

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



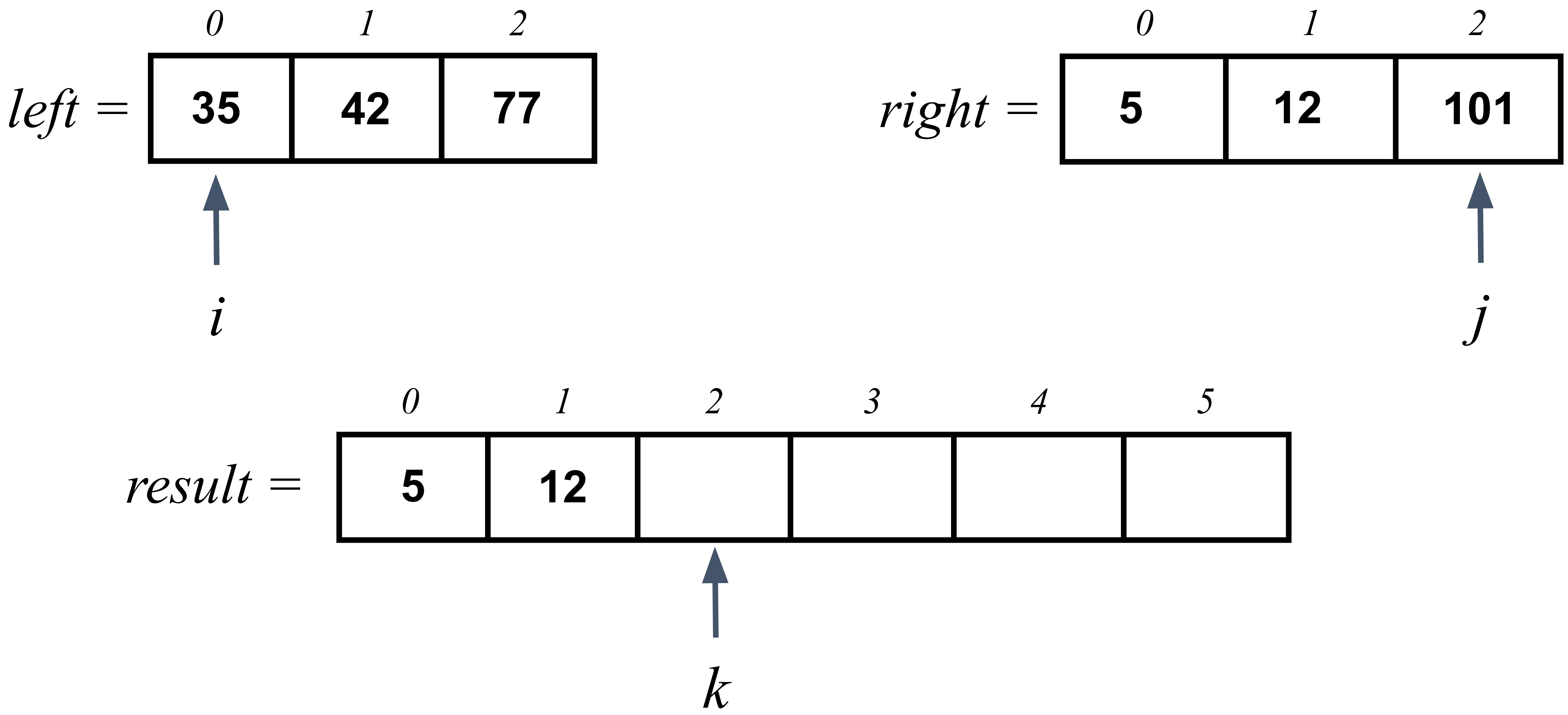
Merge Sort: the merge procedure

Then again we increase the indices j and k



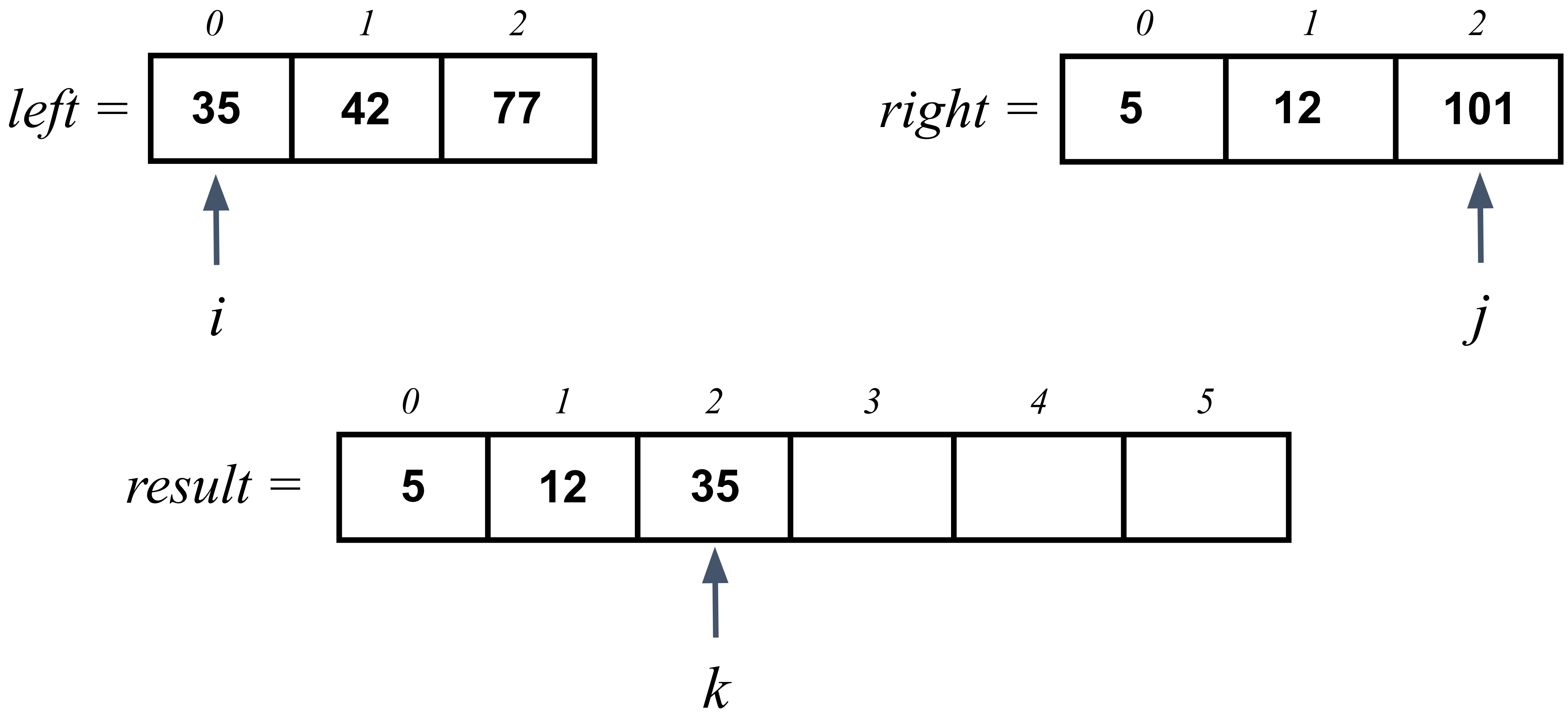
Merge Sort: the merge procedure

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



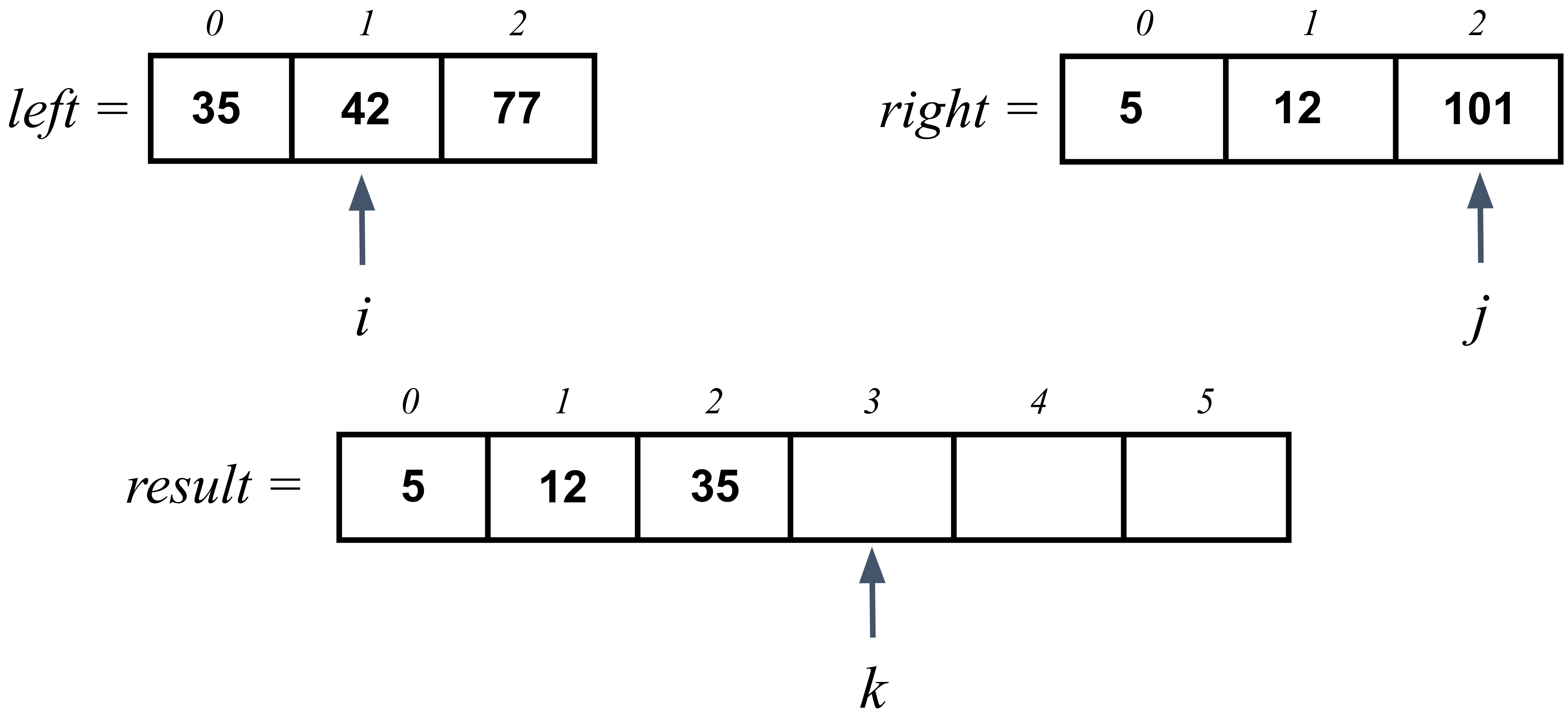
Merge Sort: the merge procedure

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



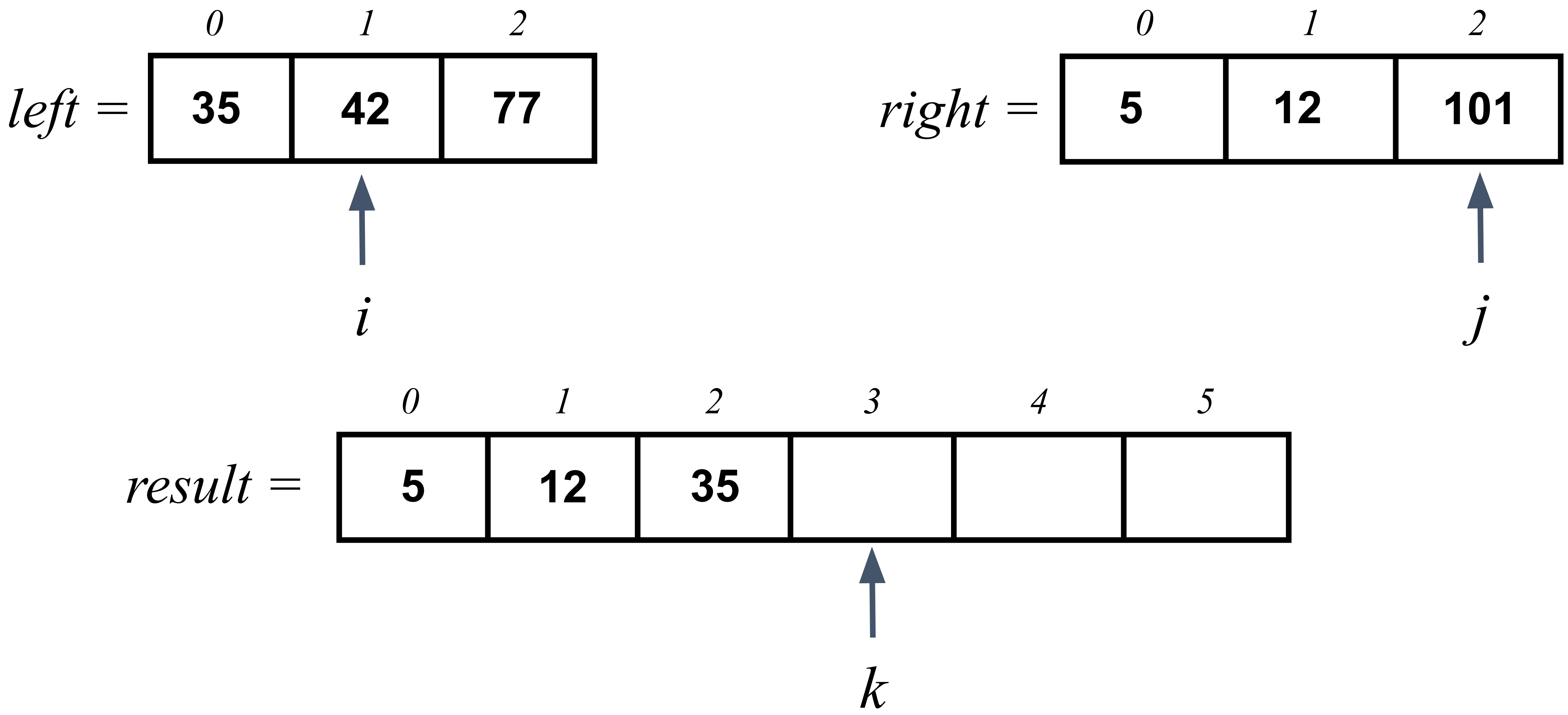
Merge Sort: the merge procedure

This time we increase the indices i and k



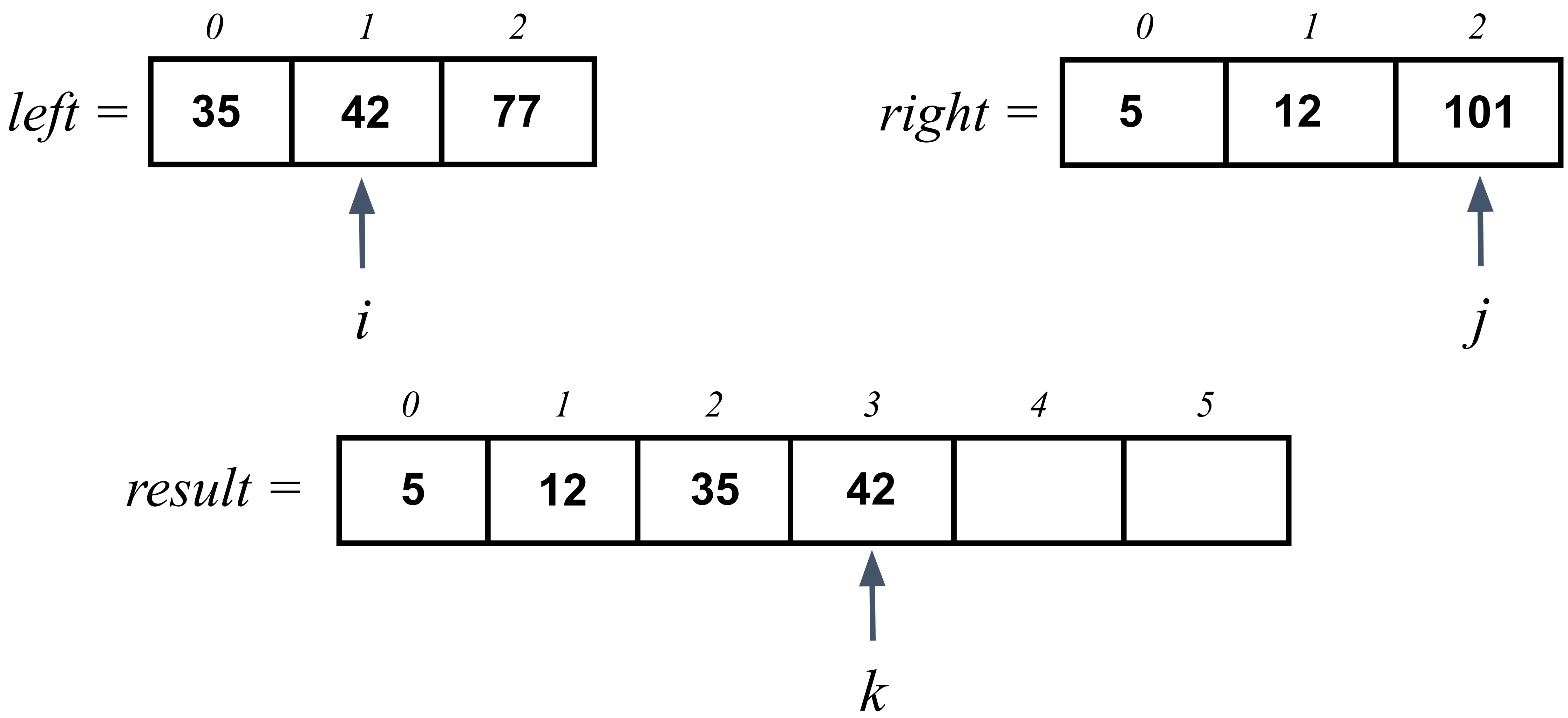
Merge Sort: the merge procedure

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



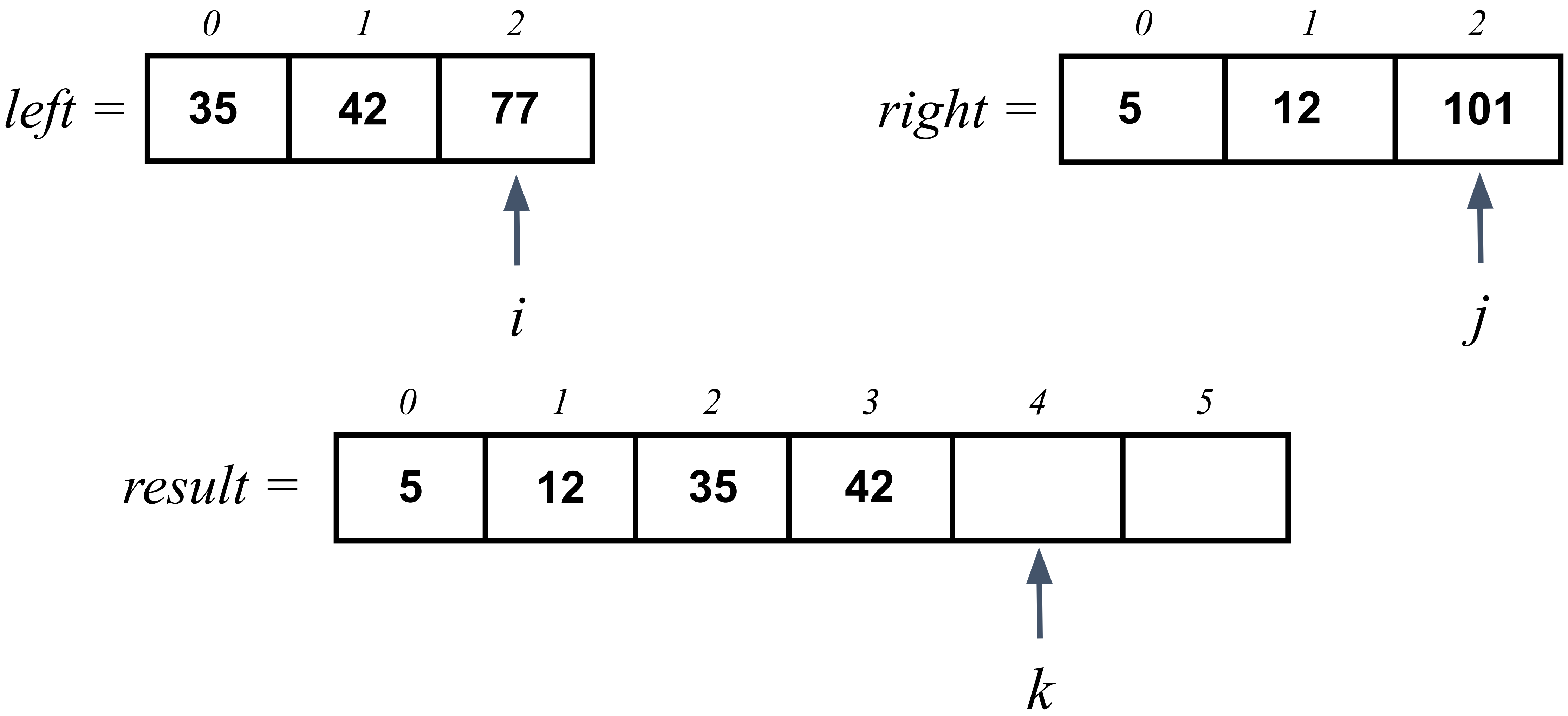
Merge Sort: the merge procedure

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



Merge Sort: the merge procedure

Again we increase the indices i and k



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

$result =$

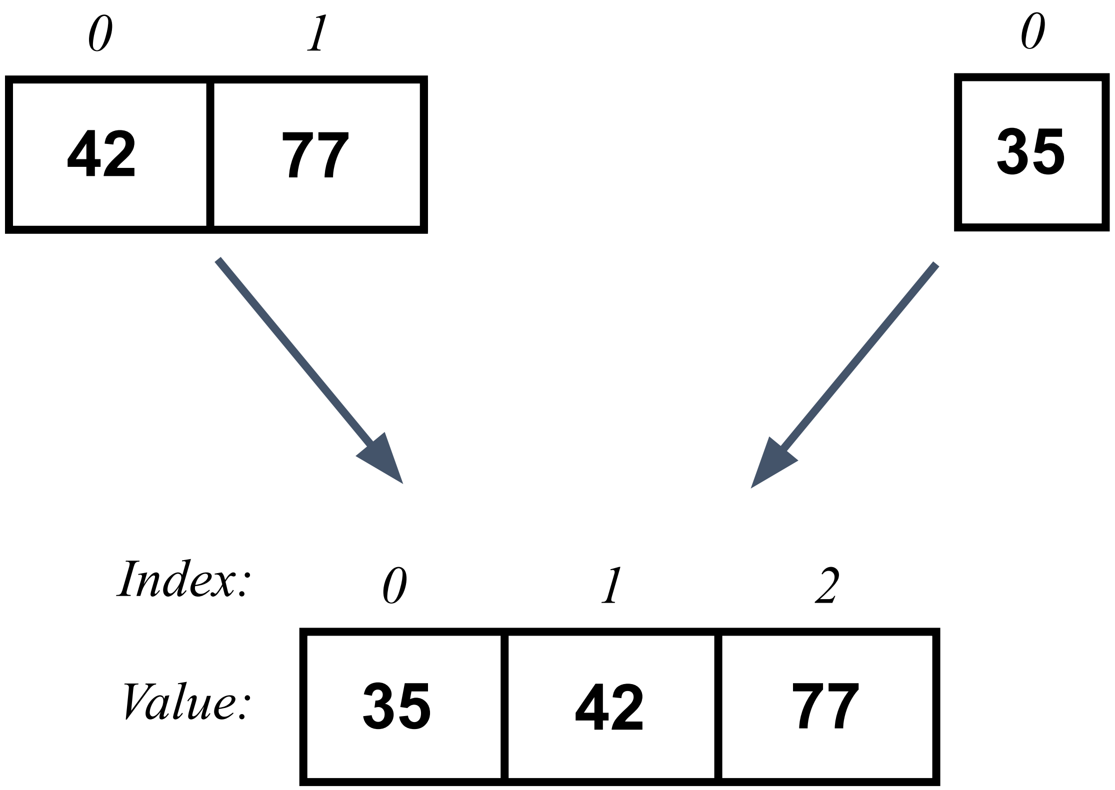
0	1	2	3	4	5
5	12	35	42	77	101

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 smaller values!

Merge Sort: an example

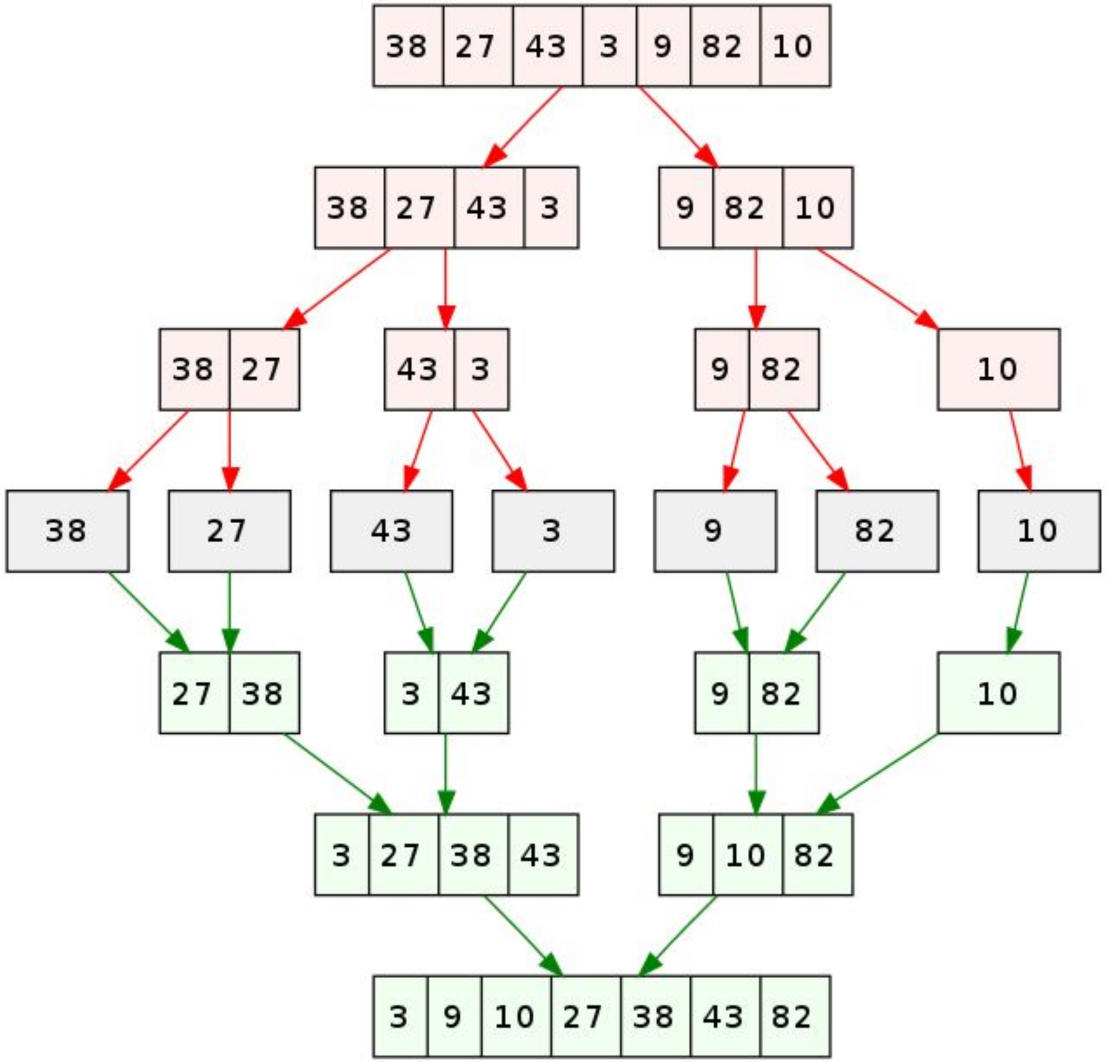


Merge Sort: an example

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)

Merge procedure: Pseudocode

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 $A[i_1; f_2]$

Python Sort!

What is the algorithm behind python’s sorted?

Python Sort – TimSort (hybrid)

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

Python but also Java!

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” (similar to **Merge sort**).

Python Sort – TimSort (hybrid)

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

Python but also Java!

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

TimSort Comment: https://mail.python.org/pipermail/python-dev/2002-July/026837.html

Python Sort – TimSort (hybrid)

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

Python but also Java!

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.

TimSort Comment: https://mail.python.org/pipermail/python-dev/2002-July/026837.html

Python Sort – TimSort (hybrid)

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

Python but also Java!

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

- Time complexity ?
- Space-Complexity ?

TimSort Comment: https://mail.python.org/pipermail/python-dev/2002-July/026837.html

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ 12, 10, 7, 5,\ \ 7, 10, 14, 25, 36,\ \ 3, 5, 11, 14, 15, 21, 22,\ \ 20, 15, 10, 8, 5, 1\ )$$

Tip: Real-world data is not too much randomly distributed: it's common to have sorted runs in the data to be sorted.

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ \underbrace{12, 10, 7, 5,}_{\text{first run}}\ 7, 10, 14, 25, 36,\ 3, 5, 11, 14, 15, 21, 22,\ 20, 15, 10, 8, 5, 1\ )$$

Tip: Real-world data is not too much randomly distributed: it's common to have sorted runs in the data to be sorted.

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ \underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{\overbrace{7, 10,} 14, 25, 36,}_{\text{second run}}\ 3, 5, 11, 14, 15, 21, 22,\ 20, 15, 10, 8, 5, 1\ )$$

Tip: Real-world data is not too much randomly distributed: it's common to have sorted runs in the data to be sorted.

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ \underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{7, 10, 14, 25, 36,}_{\text{second run}}\ \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}}\ 20, 15, 10, 8, 5, 1\ )$$

Tip: Real-world data is not too much randomly distributed: it's common to have sorted runs in the data to be sorted.

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ \underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{7, 10, 14, 25, 36,}_{\text{second run}}\ \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}}\ \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}}\ )$$

Tip: Real-world data is not too much randomly distributed: it's common to have sorted runs in the data to be sorted.

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ \underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{7, 10, 14, 25, 36,}_{\text{second run}}\ \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}}\ \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}}\ )$$

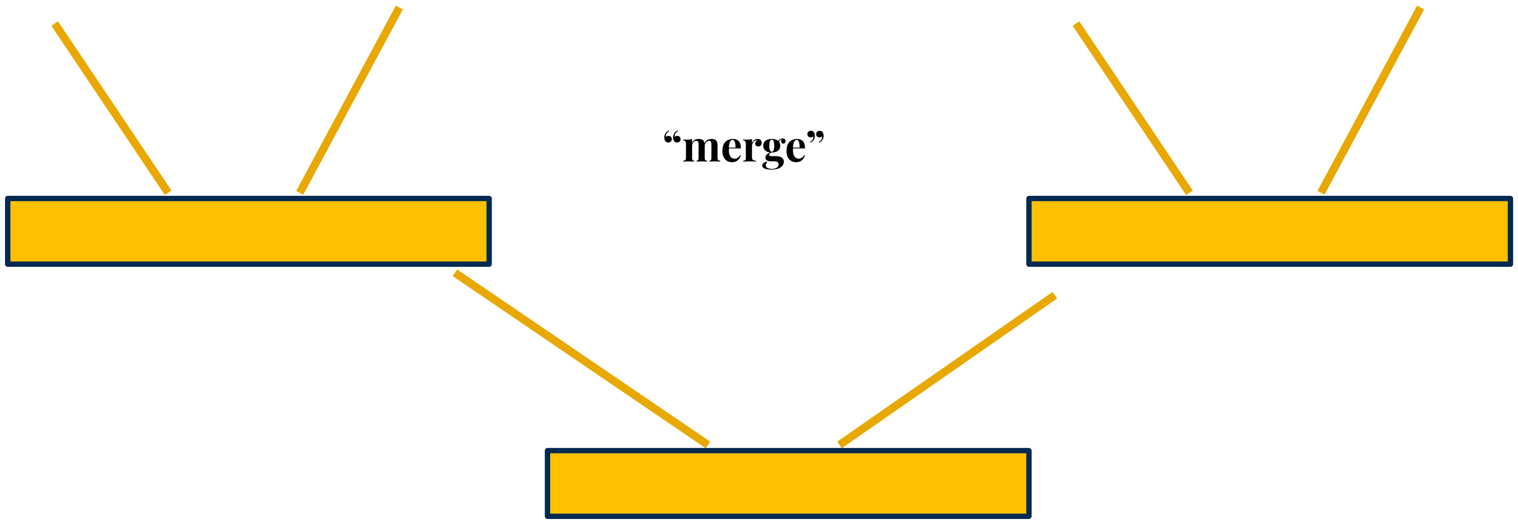
"run" decomposition

Complexity ?

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ \underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{7, 10, 14, 25, 36,}_{\text{second run}}\ \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}}\ \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}}\ )$$

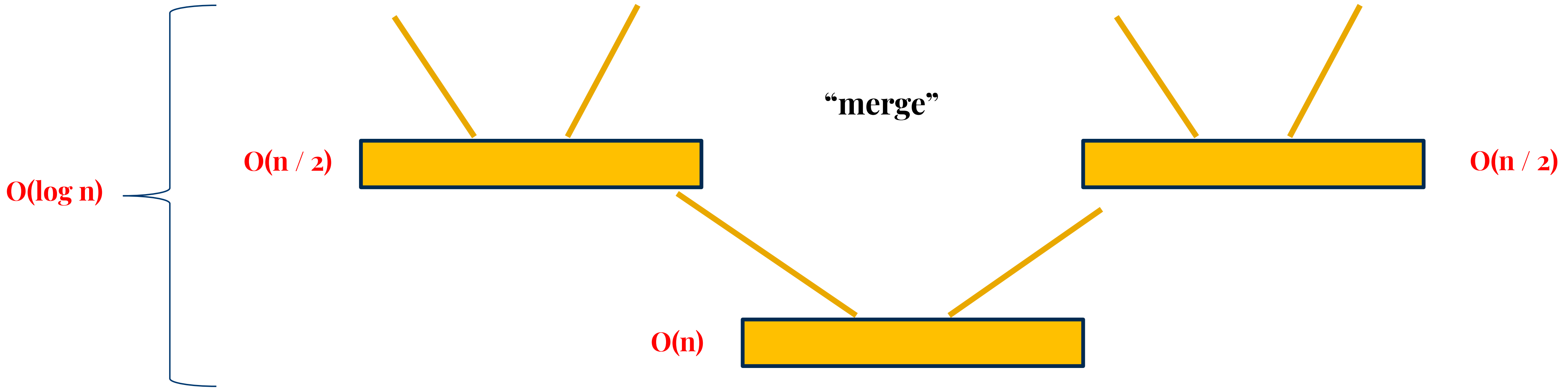


Merge Complexity ? Total Complexity ?

Python Sort – TimSort (hybrid)

General Idea:

O(n) $$S = (\underbrace{12, 10, 7, 5,}_{\text{first run}} \underbrace{7, 10, 14, 25, 36,}_{\text{second run}} \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}} \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}})$$

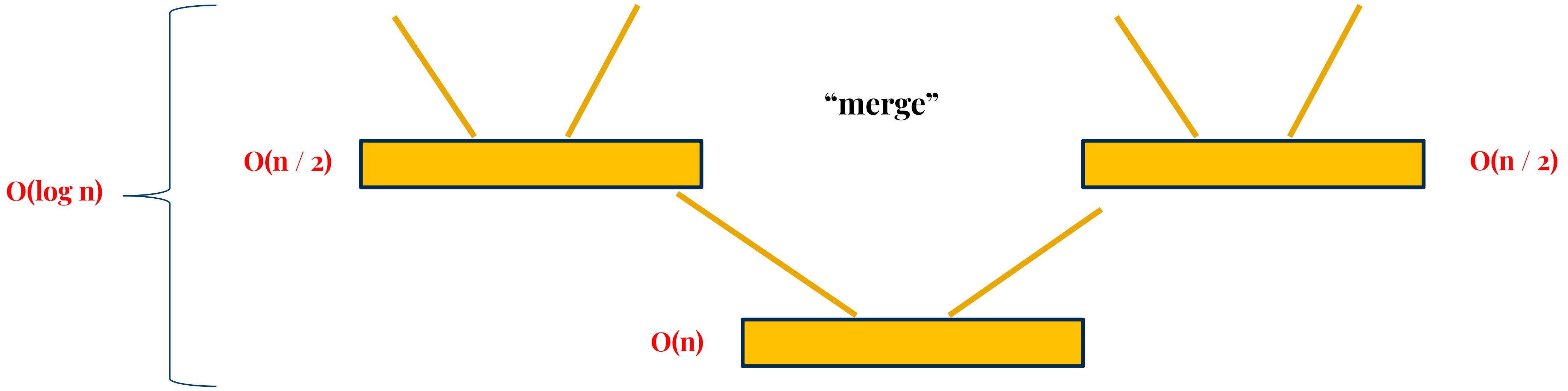


Complexity ? O(n log n)

Python Sort – TimSort (hybrid)

General Idea:

O(n) $S = (\underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{7, 10, 14, 25, 36,}_{\text{second run}}\ \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}}\ \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}})$

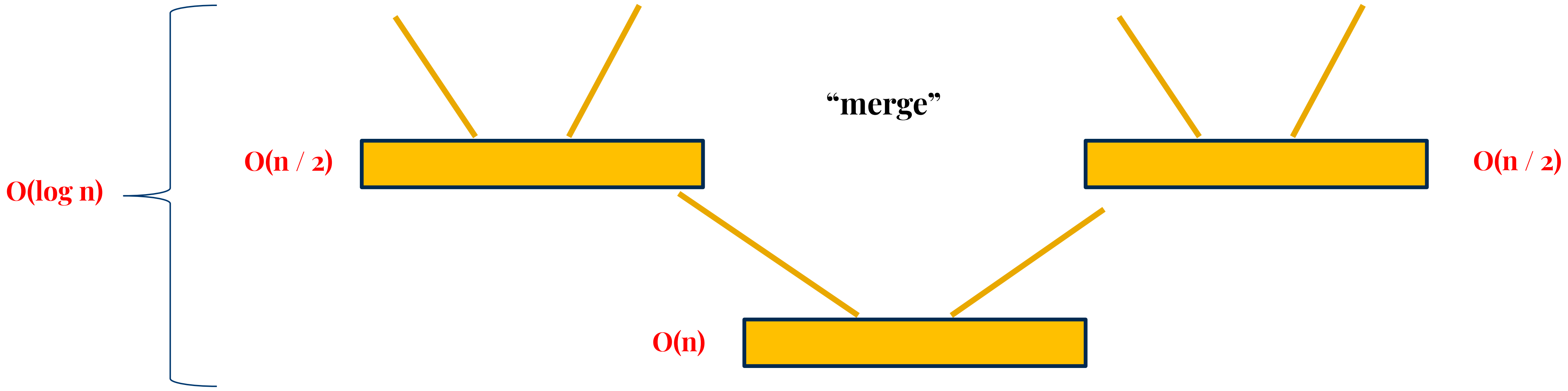


Which is the <u>Best case</u>? Cost?

Python Sort – TimSort (hybrid)

General Idea:

O(n) $S = (\underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{7, 10, 14, 25, 36,}_{\text{second run}}\ \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}}\ \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}})$



Which is the <u>Best case</u>? O(n) we just execute the run-decomposition

Python Sort – TimSort (hybrid)

General Idea:

$$S = (\ \underbrace{12, 10, 7, 5,}_{\text{first run}}\ \underbrace{7, 10, 14, 25, 36,}_{\text{second run}}\ \underbrace{3, 5, 11, 14, 15, 21, 22,}_{\text{third run}}\ \underbrace{20, 15, 10, 8, 5, 1}_{\text{fourth run}}\ )$$

- What if the size of each run is too small (or too large)?
- When we use the **insertion sort**?

Insertion Sort: We guarantee that the initial runs are not to small using the insertion sort! Usually there is a minimum size (32 or 64) that we want to have!

Python Sort – TimSort (hybrid)

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

Python but also Java!

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

- Worst-case time complexity - O(n log n)
- Best-case time complexity – O(n)
- Space complexity - O(n)

Python Sort – TimSort (hybrid)

Pseudo-code:

Algorithm 3: TimSort: translation of Algorithm 1 and Algorithm 2.

Input : A sequence to S to sort

Result: The sequence S is sorted into a single run, which remains on the stack.

Note: At any time, we denote the height of the stack $\mathcal{R}$ by h and its i^{th} top-most run (for $1 \leqslant i \leqslant h$) by R_i. The length of this run is denoted by r_i.

runs $\leftarrow$ the run decomposition of S
$\mathcal{R} \leftarrow$ an empty stack
while runs $\neq \emptyset$ **do** // main loop of TimSort
  remove a run r from **runs** and push r onto $\mathcal{R}$ // #1
  while true do
    if $h \geqslant 3$ **and** $r_1 > r_3$ **then** merge the runs R_2 and R_3 // #2
    else if $h \geqslant 2$ **and** $r_1 \geqslant r_2$ **then** merge the runs R_1 and R_2 // #3
    else if $h \geqslant 3$ **and** $r_1 + r_2 \geqslant r_3$ **then** merge the runs R_1 and R_2 // #4
    else break
while $h \neq 1$ **do** merge the runs R_1 and R_2

Python Sort - Comparison

Num Items	Selection	Insertion	Quicksort	Mergesort	TimSort (Built-in sort
1,000	<0.001	<0.001	-	-	-
2,000	0.001	<0.001	-	-	-
4,000	0.004	0.003	-	-	-
8,000	0.017	0.010	-	-	-
16,000	0.065	0.040	0.002	0.002	0.003
32,000	0.258	0.160	0.002	0.003	0.002
64,000	1.110	0.696	0.005	0.008	0.004
128,000	4.172	2.645	0.011	0.015	0.009
256,000	16.48	10.76	0.024	0.034	0.018
512,000	70.38	47.18	0.049	0.068	0.040
1,024,000	-	-	0.098	0.143	0.082
2,048,000	-	-	0.205	0.296	0.184
4,096,000	-	-	0.450	0.659	0.383
8,192,000	-	-	0.941	1.372	**0.786**

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

Project-v2 Q/A ?

Iterative Merge-Sort (Bottom-up) !

Iterative Merge-Sort (Bottom-up) !

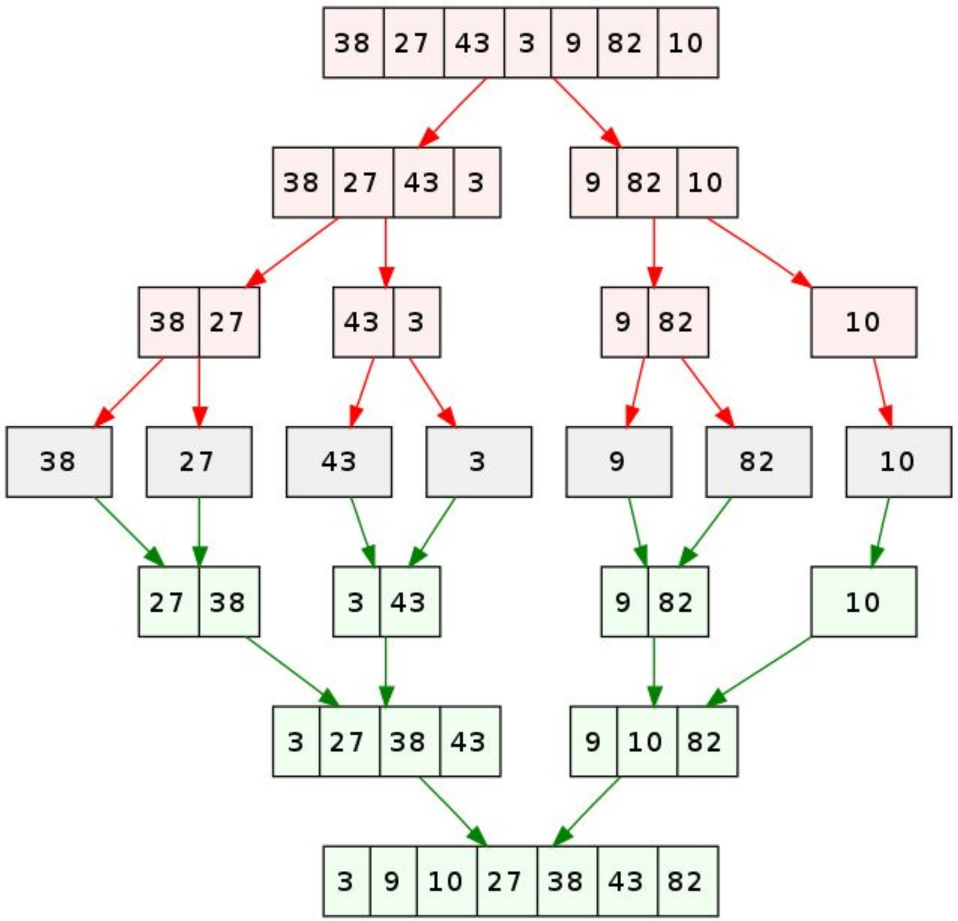
Iterative Merge-Sort.

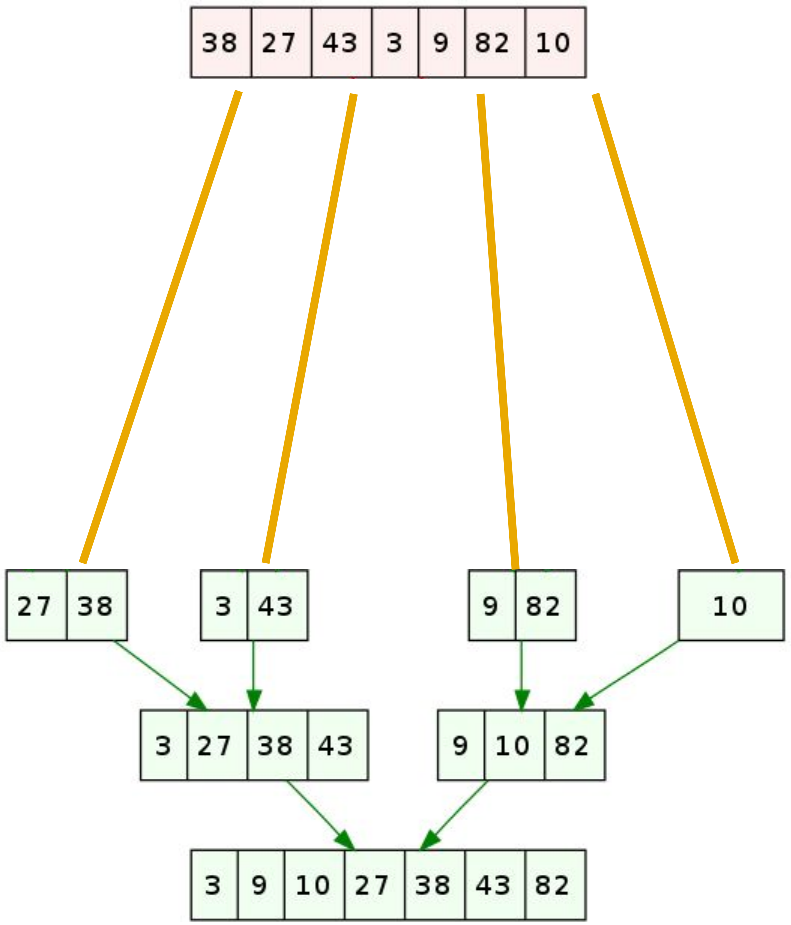
Idea:

- We start by sorting all subarrays of 1 element;
- Then we merge results into subarrays of 2 elements,
- Then we merge results into subarrays of 4 elements.

Likewise, perform successive merges until the array is completely sorted.

Merge-Sort iterative Implementation





Thank you!