Binary Search

Insert Into Sorted Array

1	2	3	5	7	8	2

Recall the basic subroutine in insertion sort:

$$\mathtt{vec}[0] \leq \mathtt{vec}[1] \leq \cdots \leq \mathtt{vec}[\mathtt{i}-1]$$

and we want to insert $\mathtt{vec}[\mathtt{i}]$ into its correct position.

We gave an algorithm with running time $\Theta(i)$ to do this.

Is there a better way?

Binary Search

$\texttt{vec} = $

1	2	3	5	7	8

$a = 2$

Let's abstract out the problem. Say we have a sorted array

$$\texttt{vec}[0] \leq \cdots \leq \texttt{vec}[\texttt{n} - 1]$$

We also have a number a. We want to find an index i such that

$$\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$$

Binary Search

$-\infty$	1	2	3	5	7	8	∞
-1	0			. . .		n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Let $\texttt{vec}[-1] = -\infty$ and $\texttt{vec}[\texttt{n}] = \infty$ so that such an index i always exists.

(We won't actually do this in the algorithm, but it is helpful to imagine these sentinel values for the analysis.)

Examples

$-\infty$	1	2	3	5	7	8	∞
-1	0			…		n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Examples

$-\infty$	1	2	3	5	7	8	∞
-1	0		…			n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-\texttt{1}] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

$a = 2$ then the output should be 2.

Examples

$-\infty$	1	2	3	5	7	8	∞
-1	0			$\cdots$		n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

$a = 2$ then the output should be 2.

$a = -3$ then the output should be 0.

Examples

$-\infty$	1	2	3	5	7	8	∞
-1	0		...			n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

$a = 2$ then the output should be 2.

$a = -3$ then the output should be 0.

$a = 6$ then the output should be 4.

Examples

$-\infty$	1	2	3	5	7	8	∞
-1	0		$\cdots$			n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-1] \le \texttt{a} < \texttt{vec}[\texttt{i}]$.

$a = 2$ then the output should be 2.

$a = -3$ then the output should be 0.

$a = 6$ then the output should be 4.

$a = 8$ then the output should be 6.

Binary Search: Invariant

$-\infty$	1	2	3	5	7	8	∞
-1	0		...			n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Binary Search: Invariant

$-\infty$	1	2	3	5	7	8	∞
-1	0		...			n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left}-1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Initialization: Let $\texttt{left} = 0, \texttt{right} = \texttt{n}$.

The invariant holds!

Binary Search: Invariant

$-\infty$	1	2	3	5	7	8	∞
-1	0		…			n-1	n

We want to find an index i such that $\texttt{vec}[\texttt{i}-1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left}-1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Termination: When $\texttt{left} = \texttt{right}$ we are done.

Return `left` as the answer.

Binary Search: Invariant

$-\infty$	1	2	3	5	7	8	∞
-1	0			...		n-1	n

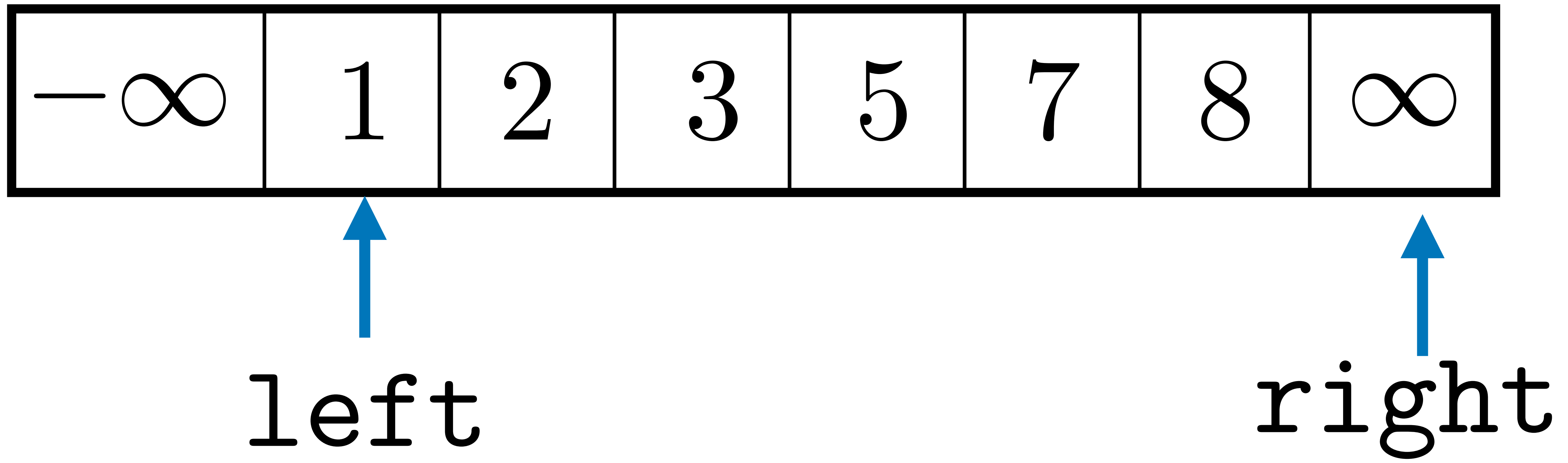
We want to find an index i such that $\texttt{vec}[\texttt{i} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{i}]$.

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Maintenance: We want to bring `left` and `right` closer together while maintaining the invariant.

Update



$a = 2$

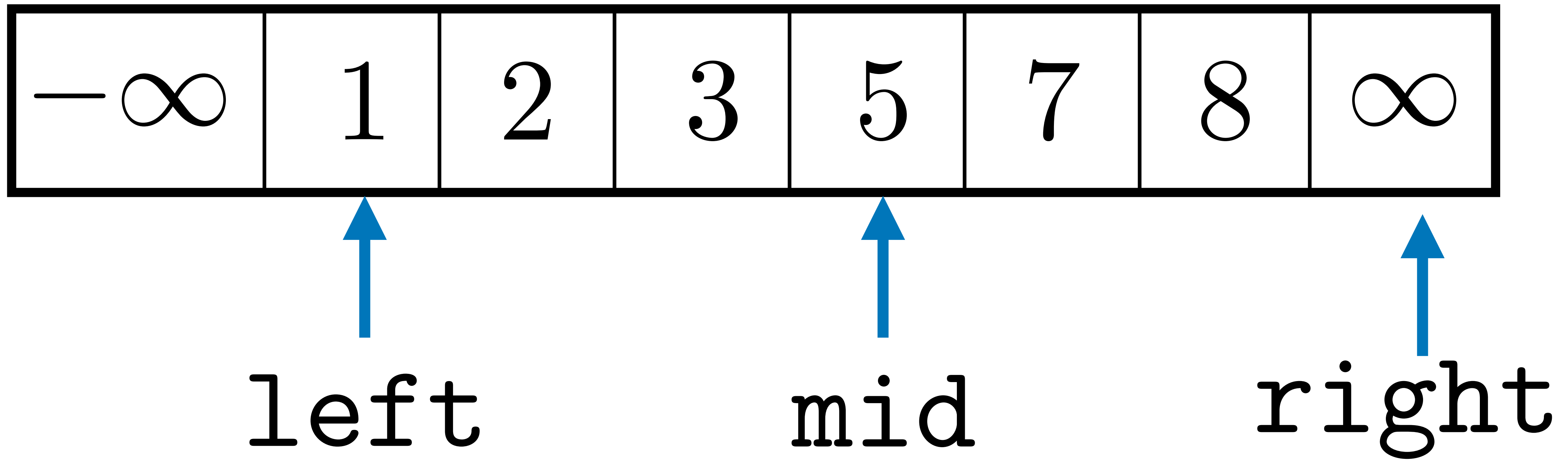
Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Update Idea: Probe the middle element between `left` and `right`.

If $a < \texttt{vec}[\texttt{mid}]$ we can update $\texttt{right} = \texttt{mid}$ and maintain the invariant.

Update



$a = 2$

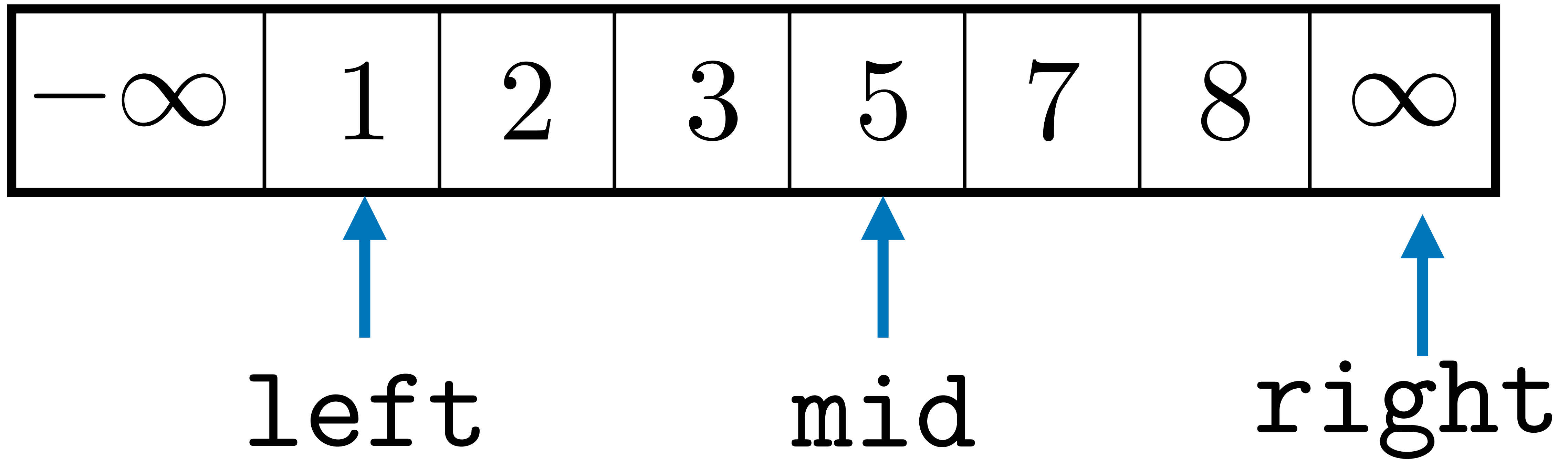
Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Update Idea: Probe the middle element between $\texttt{left}$ and $\texttt{right}$.

If $a < \texttt{vec}[\texttt{mid}]$ we can update $\texttt{right} = \texttt{mid}$ and maintain the invariant.

Update



$a = 2$

Invariant: Maintain two indices $\texttt{left} \leq \texttt{right}$ such that

$$\texttt{vec}[\texttt{left} - 1] \leq \texttt{a} < \texttt{vec}[\texttt{right}]$$

Update Idea: Probe the middle element between $\texttt{left}$ and $\texttt{right}$.

If $a \geq \texttt{vec}[\texttt{mid}]$ we can update $\texttt{left} = \texttt{mid} + 1$ and maintain the invariant.

Algorithm

```
std::size_t insertionPoint(const std::vector<int>& vec, int a) {
  std::size_t left = 0;
  std::size_t right = vec.size();
  while(left < right) {
    std::size_t middle = left + (right - left)/2;
    if(a < vec[middle]) {
      right = middle;
    } else {
      left = middle + 1;
    }
  }
  return left;
}
```

https://godbolt.org/z/e7T7nTzzs

Binary Search: Time

1	2	3	5	7	8

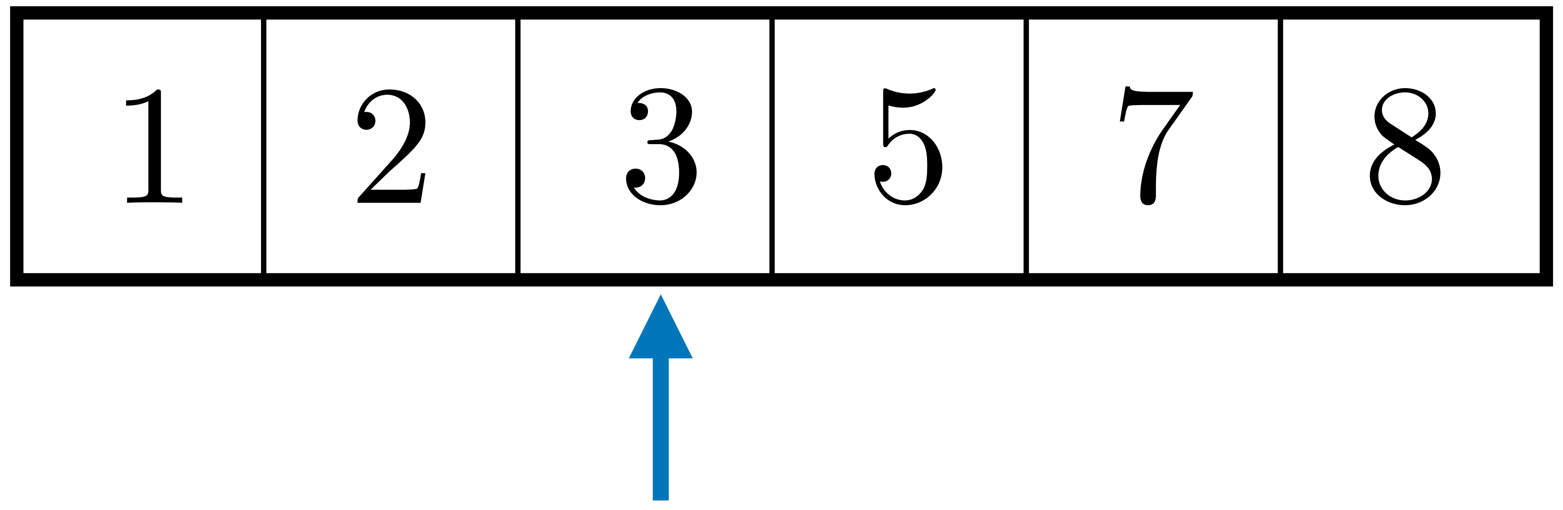
$a = 2$

The algorithm terminates when $\texttt{left} = \texttt{right}$.

The initial distance between them is n , and the distance halves in each iteration.

The worst-case running time of the algorithm is $\Theta(\log n)$.

Inserting

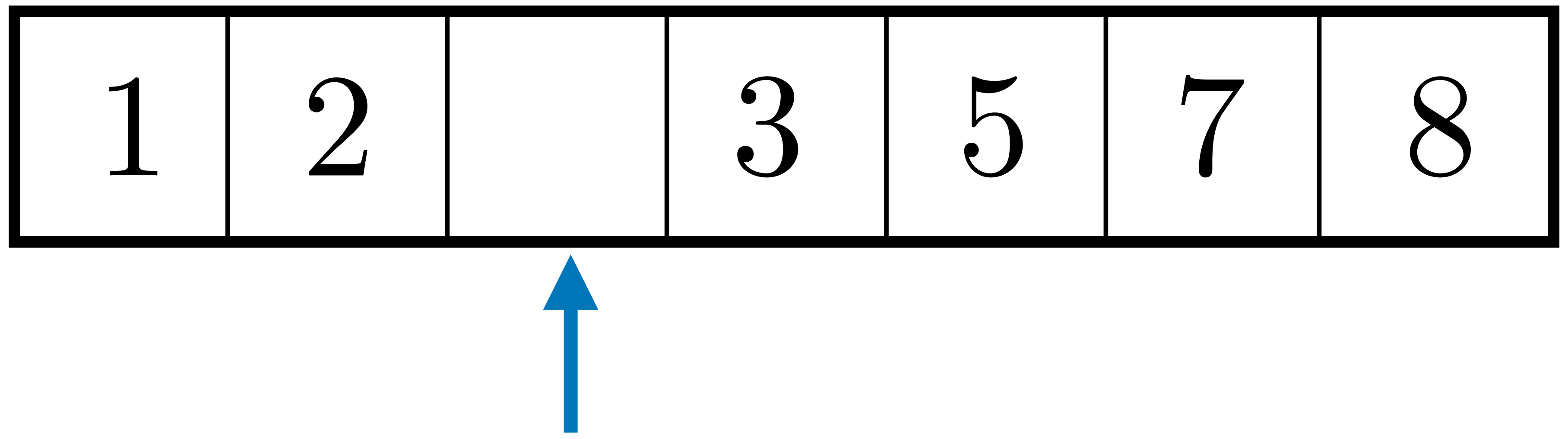


$a = 2$

We have now found where a should be inserted.

But what about the complexity of actually inserting a?

In a resizable array, we have to shift over all the elements to the right of the insertion point.



Inserting

1	2	3	5	7	8

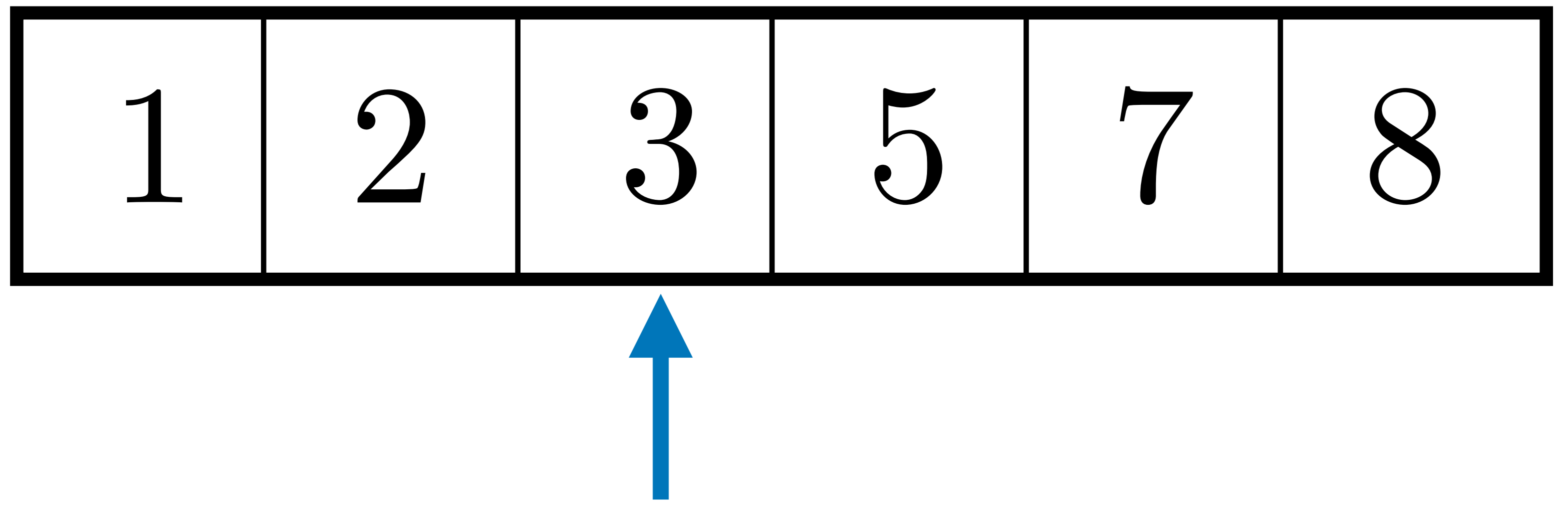
$a = 2$

We have now found where a should be inserted.

But what about the complexity of actually inserting a?

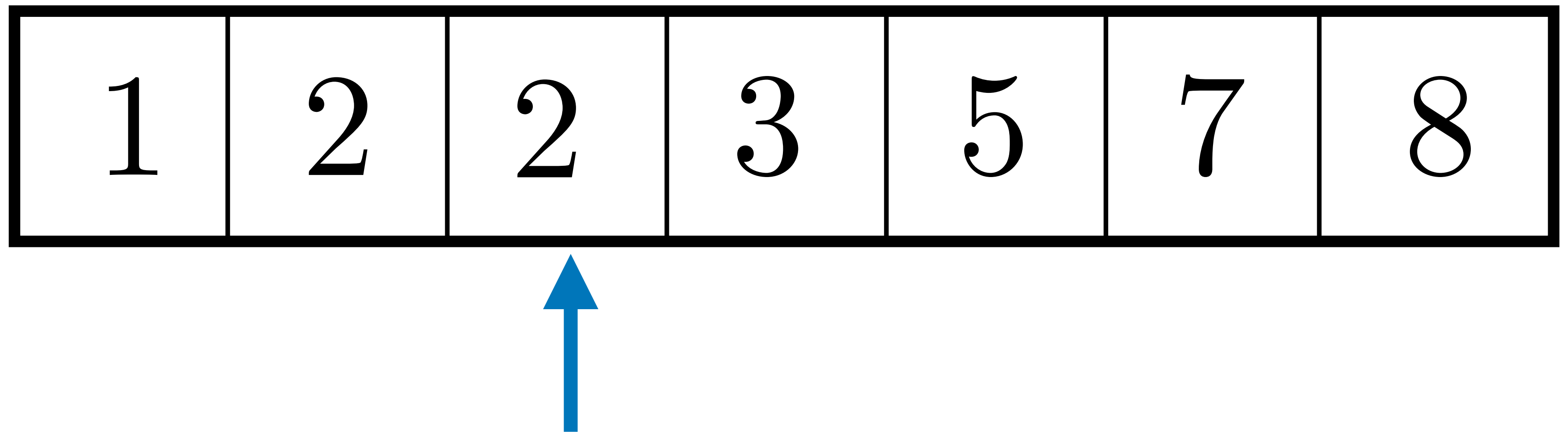
In a resizable array, we have to shift over all the elements to the right of the insertion point.

1	2	2	3	5	7	8



$a = 2$

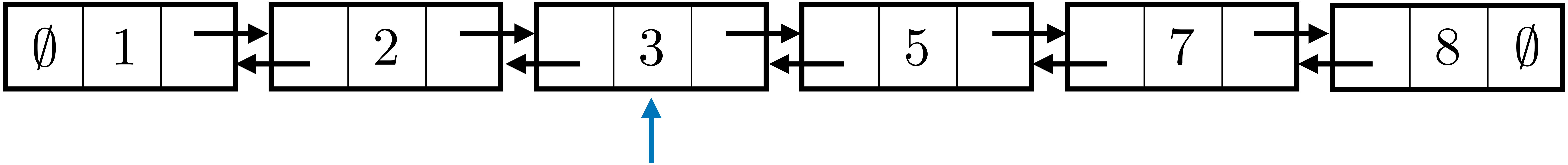
In a resizable array, we have to shift over all the elements to the right of the insertion point.



This still has a worst-case complexity of $\Theta(n)$.

We do not realize an improvement for insertion sort.

Doubly Linked List



Can this idea work if we use a linked list instead?

In a linked list we can insert a new node into the list in constant time.

However, we do not have random access to the elements so we cannot do binary search in $O(\log n)$ time.

Later we will look at balanced binary search trees which can maintain an ordered list with $O(\log n)$ insertion time.

Divide and Conquer: Example

Buy and Sell Stock

We are first going to illustrate divide and conquer with an example.

Leetcode 121: Best time to buy and sell stock (Easy, Blind 75)

We are given a vector $\texttt{prices}$ where $\texttt{prices}[i]$ is the price of a stock on day i. We can buy the stock once, and a later date sell it.

What is the maximum profit we can make?

Buy and Sell Stock

We are first going to illustrate divide and conquer with an example.

Leetcode 121: Best time to buy and sell stock (Easy, Blind 75)

We are given a vector $\texttt{prices}$ where $\texttt{prices}[i]$ is the price of a stock on day i. We can buy the stock once, and a later date sell it.

What is the maximum profit we can make?

In other words, we want to compute

$$\max_{i<j} \texttt{prices}[j] - \texttt{prices}[i]$$

Examples

prices

9	2
0	1

Examples

`prices`

9	2
0	1

max profit: 0

Examples

`prices`

9	2
0	1

max profit: 0

The answer is not just the maximum value minus the minimum value.

Examples

`prices`

9	2
0	1

max profit: 0

The answer is not just the maximum value minus the minimum value.

`prices`

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

Examples

prices

0	1
9	2

max profit: 0

The answer is not just the maximum value minus the minimum value.

prices

0	1	2	3	4	5	6	7
4	6	9	3	2	5	8	1

max profit: 6

Divide and Conquer

prices

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

There are 3 possible cases:

1) The best time to buy and sell both occur in the first half of the array.

2) The best time to buy and sell both occur in the second half of the array.

3) The best time to buy occurs in the first half of the array and the best time to sell occurs in the second half of the array.

Divide and Conquer

There are 3 possible cases:

1) The best time to buy and sell both occur in the first half of the array.

2) The best time to buy and sell both occur in the second half of the array.

The first two cases are instances of the buy and sell stock problem on an array of half the size.

Divide and Conquer

There are 3 possible cases:

1) The best time to buy and sell both occur in the first half of the array.

2) The best time to buy and sell both occur in the second half of the array.

The first two cases are instances of the buy and sell stock problem on an array of half the size.

This is the divide part of divide and conquer. We express the original problem in terms of instances of the original problem on smaller inputs.

Additional Work

We do not completely express the problem in terms of the same problem on smaller inputs because there is the third case.

3) The best time to buy occurs in the first half of the array and the best time to sell occurs in the second half of the array.

We have to solve this problem separately. Do you see how to solve it?

Additional Work

We do not completely express the problem in terms of the same problem on smaller inputs because there is the third case.

3) The best time to buy occurs in the first half of the array and the best time to sell occurs in the second half of the array.

We have to solve this problem separately. Do you see how to solve it?

For this case the best time to buy is the minimum value in the first half.

The best time to sell is the maximum value in the second half.

Additional Work

3) The best time to buy occurs in the first half of the array and the best time to sell occurs in the second half of the array.

For this case the best time to buy is the minimum value in the first half.

The best time to sell is the maximum value in the second half.

The time to solve this case is $\Theta(n)$.

Example

prices

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

The maximum profit is going to be the maximum of what we can achieve in the three cases.

1) maximum profit on

4	6	9	3

Example

prices

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

The maximum profit is going to be the maximum of what we can achieve in the three cases.

1) maximum profit on

4	6	9	3

2) maximum profit on

2	5	8	1

Example

prices

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

The maximum profit is going to be the maximum of what we can achieve in the three cases.

1) maximum profit on

4	6	9	3

2) maximum profit on

2	5	8	1

3) maximum profit when we buy first half, sell second. This is 5.

`prices`

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

We recursively solve the first two cases:

1) maximum profit on

4	6	9	3

This is the maximum profit from the three cases:

prices	4	6	9	3	2	5	8	1
	0	1	2	3	4	5	6	7

We recursively solve the first two cases:

1) maximum profit on

4	6	9	3

This is the maximum profit from the three cases:

4	6

first half

$\texttt{profit} = 2$

prices	4	6	9	3	2	5	8	1
	0	1	2	3	4	5	6	7

We recursively solve the first two cases:

1) maximum profit on

4	6	9	3

This is the maximum profit from the three cases:

4	6

first half

$\texttt{profit} = 2$

9	3

second half

$\texttt{profit} = 0$

prices

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

We recursively solve the first two cases:

1) maximum profit on

4	6	9	3

This is the maximum profit from the three cases:

4	6

first half

$\texttt{profit} = 2$

9	3

second half

$\texttt{profit} = 0$

buy first half
sell second half

$\texttt{profit} = 5$

prices

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

We recursively solve the first two cases:

1) maximum profit on

4	6	9	3

This is the maximum profit from the three cases:

4	6

first half

$\texttt{profit} = 2$

9	3

second half

$\texttt{profit} = 0$

buy first half
sell second half

$\texttt{profit} = 5$

The maximum profit from this case is 5.

`prices`

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

We recursively solve the first two cases:

2) maximum profit on

2	5	8	1

This is the maximum profit from the three cases:

prices

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

We recursively solve the first two cases:

2) maximum profit on

2	5	8	1

This is the maximum profit from the three cases:

2	5

first half

$\texttt{profit} = 3$

`prices`

0	1	2	3	4	5	6	7
4	6	9	3	2	5	8	1

We recursively solve the first two cases:

2) maximum profit on

2	5	8	1

This is the maximum profit from the three cases:

2	5

first half

$\texttt{profit} = 3$

8	1

second half

$\texttt{profit} = 0$

`prices`

4	6	9	3	2	5	8	1
0	1	2	3	4	5	6	7

We recursively solve the first two cases:

2) maximum profit on

2	5	8	1

This is the maximum profit from the three cases:

2	5

first half

$\texttt{profit} = 3$

8	1

second half

$\texttt{profit} = 0$

buy first half
sell second half

$\texttt{profit} = 6$

`prices`	4	6	9	3	2	5	8	1
	0	1	2	3	4	5	6	7

We recursively solve the first two cases:

2) maximum profit on

2	5	8	1

This is the maximum profit from the three cases:

2	5

first half

$\texttt{profit} = 3$

8	1

second half

$\texttt{profit} = 0$

buy first half
sell second half

$\texttt{profit} = 6$

The maximum profit from this case is 6.

Wrap Up

prices

0	1	2	3	4	5	6	7
4	6	9	3	2	5	8	1

1) maximum profit on

4	6	9	3

$\texttt{profit} = 5$

2) maximum profit on

2	5	8	1

$\texttt{profit} = 6$

3) maximum profit when we buy first half, sell second: $\texttt{profit} = 5$

The answer is the maximum of the three cases so 6.

Code

```
int maxProfit(std::vector<int>::iterator begin, std::vector<int>::iterator end) {
    if (end - begin <= 1) {
      return 0;
    }
    std::vector<int>::iterator mid = begin + (end - begin)/2;
    int buyFirstHalfSellSecond = *std::max_element(mid, end) -
                                 *std::min_element(begin, mid);
    return std::max({maxProfit(begin, mid), maxProfit(mid, end), buyFirstHalfSellSecond});
}
```

Base case: $\texttt{end} - \texttt{begin} \leq 1$.

In this case the max profit is 0.

https://godbolt.org/z/r9b1zM1e6

Code

```cpp
int maxProfit(std::vector<int>::iterator begin, std::vector<int>::iterator end) {
    if (end - begin <= 1) {
      return 0;
    }
    std::vector<int>::iterator mid = begin + (end - begin)/2;
    int buyFirstHalfSellSecond = *std::max_element(mid, end) -
                                 *std::min_element(begin, mid);
    return std::max({maxProfit(begin, mid), maxProfit(mid, end), buyFirstHalfSellSecond});
}
```

Compute the midpoint to set up the divide step:

$$\mathtt{mid} = \mathtt{begin} + (\mathtt{end} - \mathtt{begin})/2;$$

Code

```
int maxProfit(std::vector<int>::iterator begin, std::vector<int>::iterator end) {
    if (end - begin <= 1) {
        return 0;
    }
    std::vector<int>::iterator mid = begin + (end - begin)/2;
    int buyFirstHalfSellSecond = *std::max_element(mid, end) -
                                 *std::min_element(begin, mid);
    return std::max({maxProfit(begin, mid), maxProfit(mid, end), buyFirstHalfSellSecond});
}
```

Compute the maximum profit from case 3:

$$\texttt{buyFirstHalfSellSecond} = *\texttt{max\_element}(\texttt{mid}, \texttt{end}) \\ - *\texttt{min\_element}(\texttt{begin}, \texttt{mid})$$

Code

```
int maxProfit(std::vector<int>::iterator begin, std::vector<int>::iterator end) {
    if (end - begin <= 1) {
      return 0;
    }
    std::vector<int>::iterator mid = begin + (end - begin)/2;
    int buyFirstHalfSellSecond = *std::max_element(mid, end) -
                                  *std::min_element(begin, mid);
    return std::max({maxProfit(begin, mid), maxProfit(mid, end), buyFirstHalfSellSecond});
}
```

Return the maximum of the two recursive calls and `buyFirstHalfSellSecond`.

Divide and Conquer: Recurrence

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Create the division into subproblems:

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Create the division into subproblems:

In the buy and sell stock problem this step was trivial, we just computed the midpoint.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Create the division into subproblems:

In the buy and sell stock problem this step was trivial, we just computed the midpoint.

Later we will see quicksort, where this step is substantial work.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Complete the cases:

Handle any case that is not covered by the division into subproblems.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Complete the cases:

Handle any case that is not covered by the division into subproblems.

This was the main work of the buy and sell stock algorithm.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Combine:

Combine the answers to the subproblems into an answer to the original problem.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Combine:

Combine the answers to the subproblems into an answer to the original problem.

In buy and sell stock we combined with the maximum of 3 values.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Combine:

Later we will see mergesort where the combine step is substantial work.

Divide and Conquer

Divide: Express the problem in terms of the same (or similar) problems on smaller inputs.

Create/Complete/Combine: The extra work to create the subproblems, complete additional cases, and combine answers into an overall solution.

Combine:

Later we will see mergesort where the combine step is substantial work.

Create/Complete/Combine is the spice of a D&C algorithm.

Time Complexity

How fast is our divide and conquer algorithm for the buy and sell stock problem?

Let us see how to analyze the time complexity of a divide and conquer algorithm.

Their recursive nature leads to a recurrence relation for the time complexity.

Time Complexity

Let $T(n)$ be the time it takes to solve the buy and sell stock problem on a vector of size n.

Our buy and sell stock algorithm computes the maximum of the three possible cases:

1) Maximum profit on first half: this takes time $T(\lfloor n/2 \rfloor)$.

2) Maximum profit on second half: this takes time $T(\lceil n/2 \rceil)$.

3) Maximum profit to buy in the first half and sell in the second: this takes time $O(n)$.

Time Complexity

1) Maximum profit on first half: this takes time $T(\lfloor n/2 \rfloor)$.

2) Maximum profit on second half: this takes time $T(\lceil n/2 \rceil)$.

3) Maximum profit to buy in the first half and sell in the second: this takes time $O(n)$.

For the combine step we take the maximum of the 3 values from these steps, and to create the subproblems we find the midpoint.

This additional work just takes constant time.

Time Complexity

The time to solve the problem is the sum of the time to solve the three cases, plus an $O(1)$ term to compute the division and combine by taking the max.

This gives a recurrence relation for the running time.

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

$$T(1) = O(1) \qquad \text{base case}$$

To figure out the running time we need to solve for $T(n)$.

Time Complexity

The time to solve the problem is the sum of the time to solve the three cases, plus an $O(1)$ term to compute the division and combine by taking the max.

This gives a recurrence relation for the running time.

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

create, complete, combine

$$T(1) = O(1)$$

base case

To figure out the running time we need to solve for $T(n)$.

Let's assume n is a power of 2 so we don't have to worry about the floors and ceilings.

Our recurrence relation then becomes

$$T(n) = 2T(n/2) + O(n)$$

$$T(1) = O(1)$$ base case

Let's assume n is a power of 2 so we don't have to worry about the floors and ceilings.

Our recurrence relation then becomes

$$T(n) = 2T(n/2) + O(n)$$

$$T(1) = O(1)$$ base case

Anatomy of the recurrence:

$$T(n) = 2T(n/2) + O(n)$$

2 ← number of subproblems

$n/2$ ← size of subproblems

$O(n)$ ← time for create, complete, combine

Recursion Tree

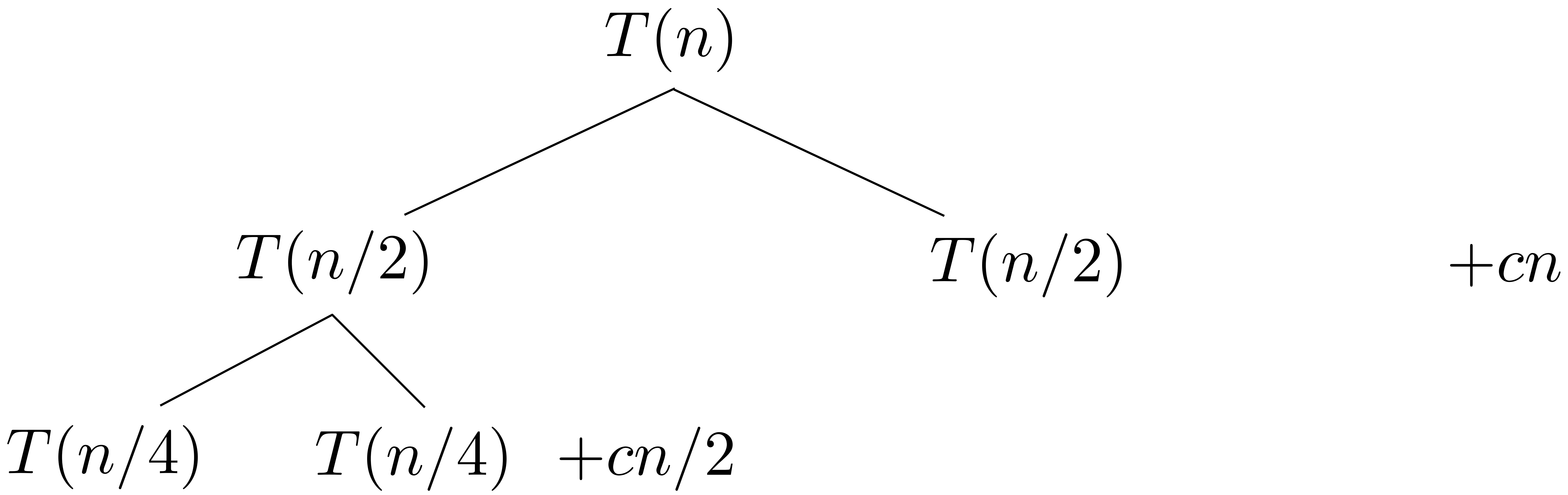
$T(n)$

Recursion Tree

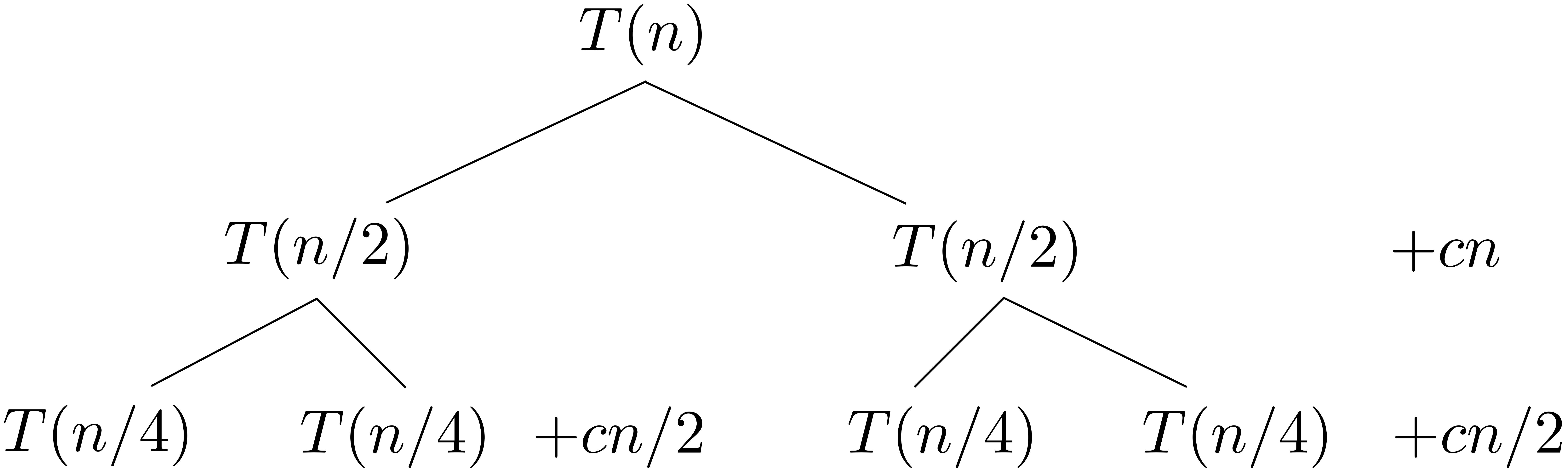
$$T(n)$$

$$T(n/2) \qquad T(n/2) \qquad +cn$$

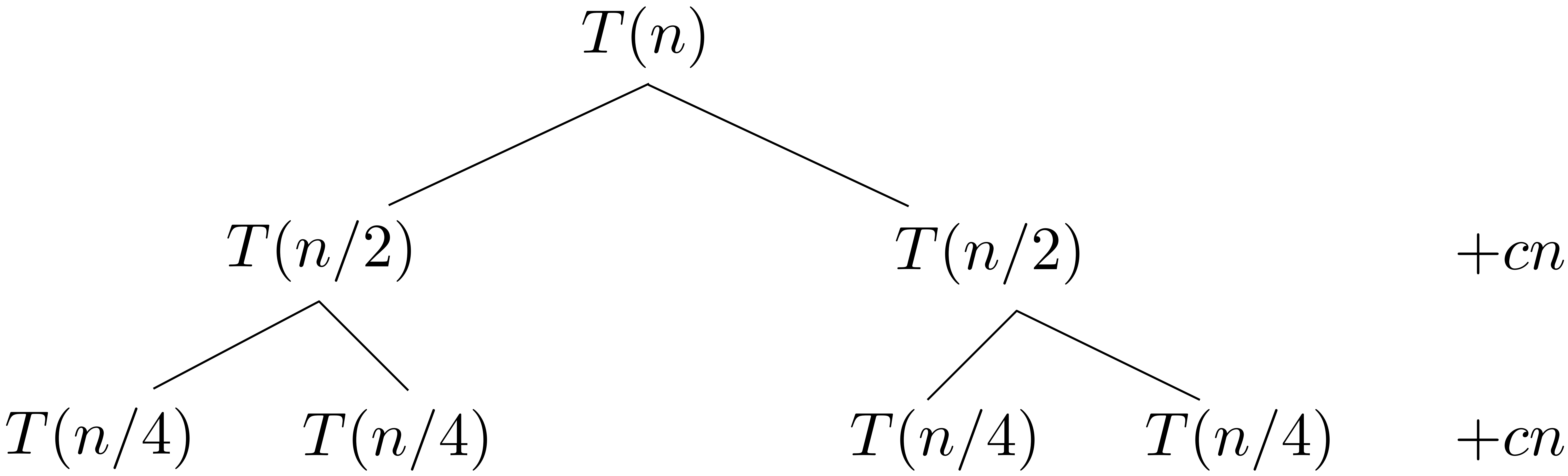
Recursion Tree



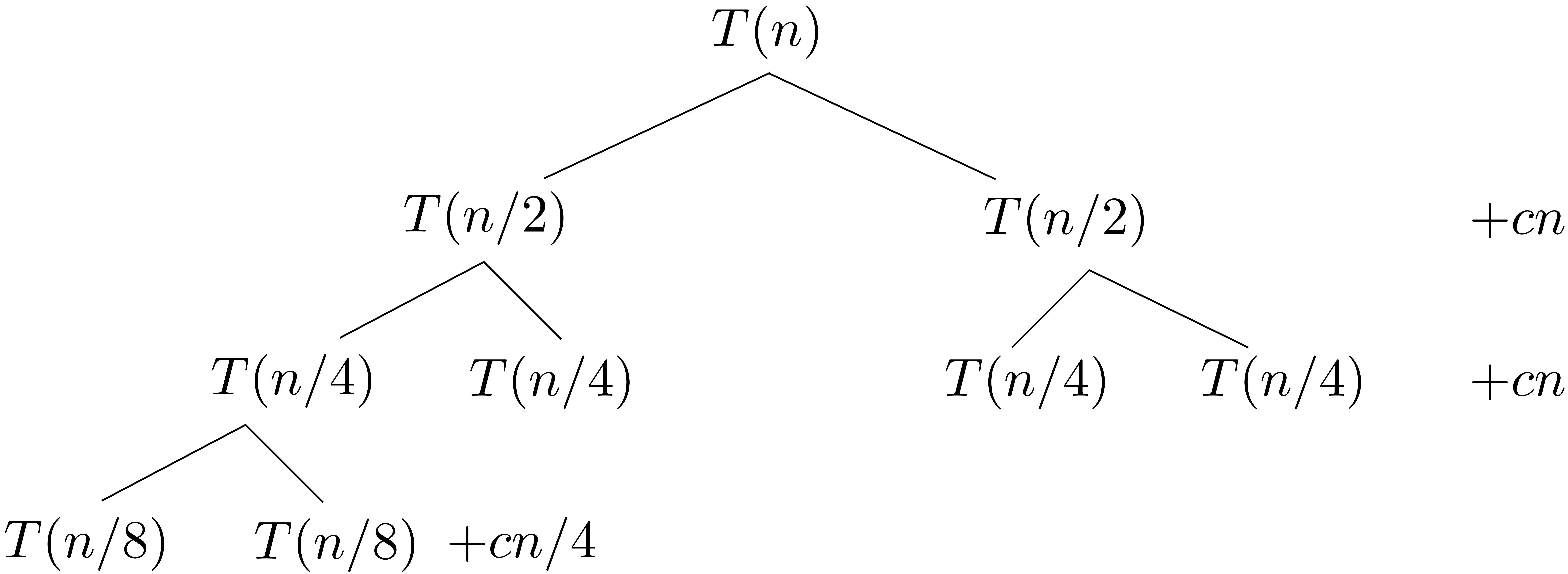
Recursion Tree



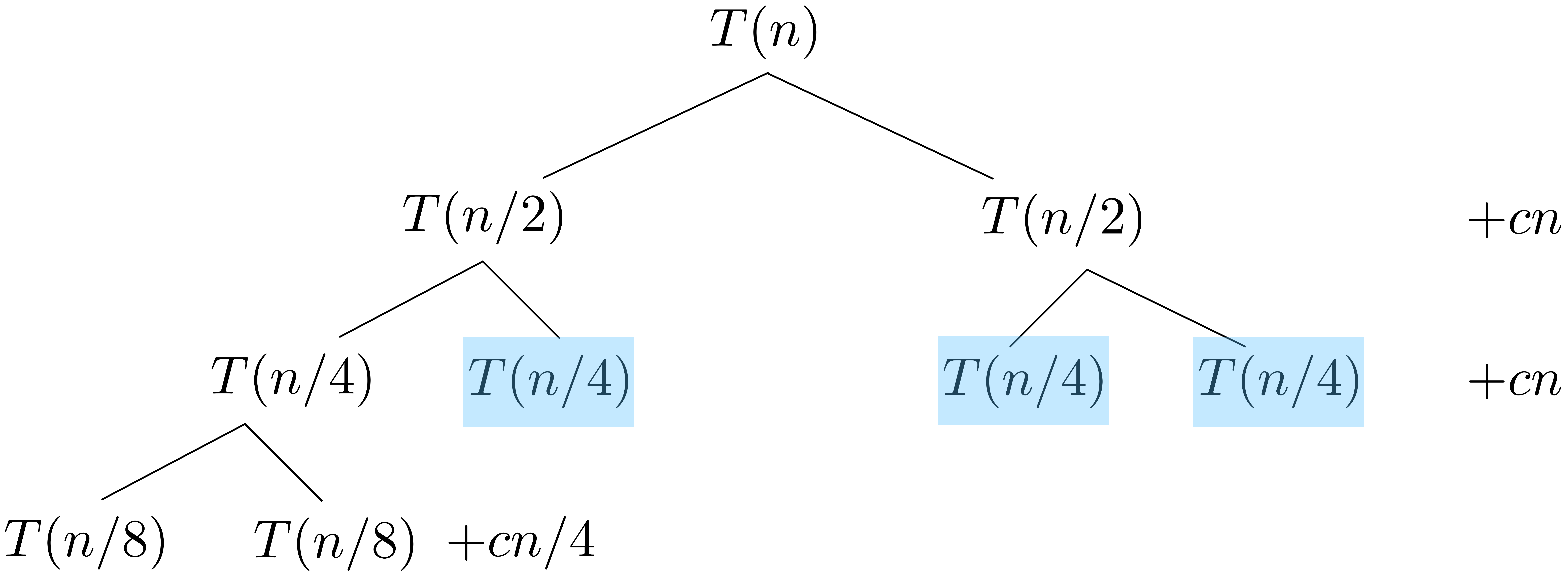
Recursion Tree

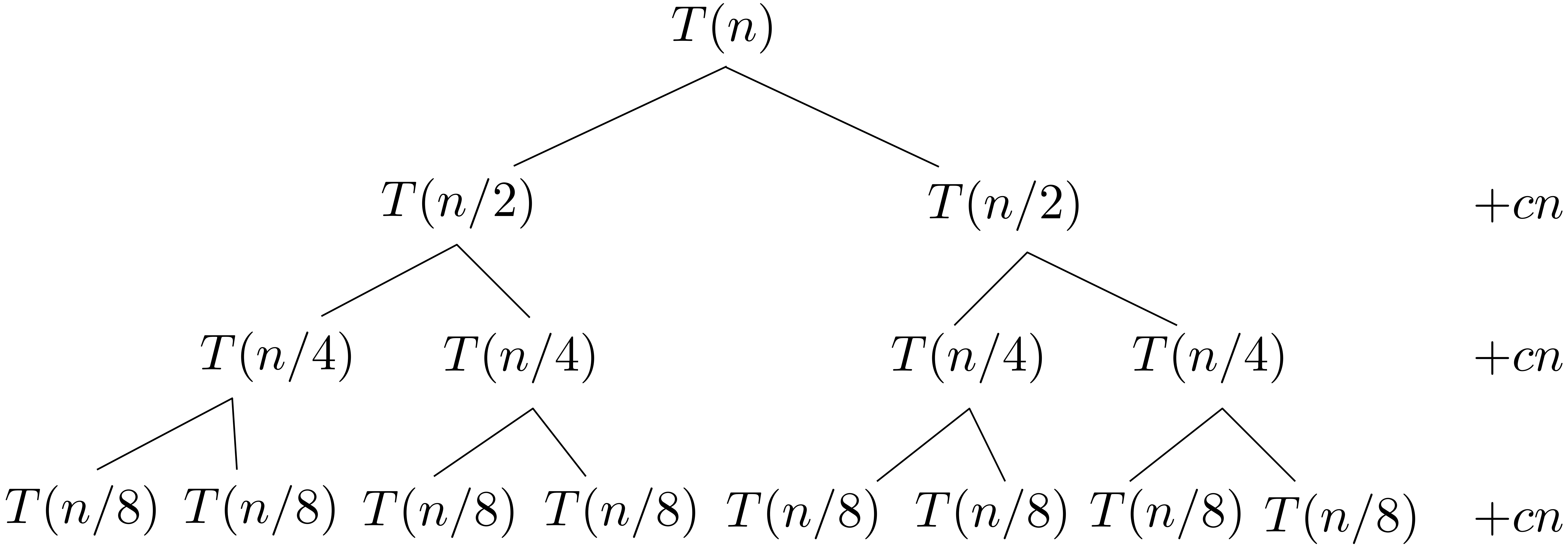


Recursion Tree

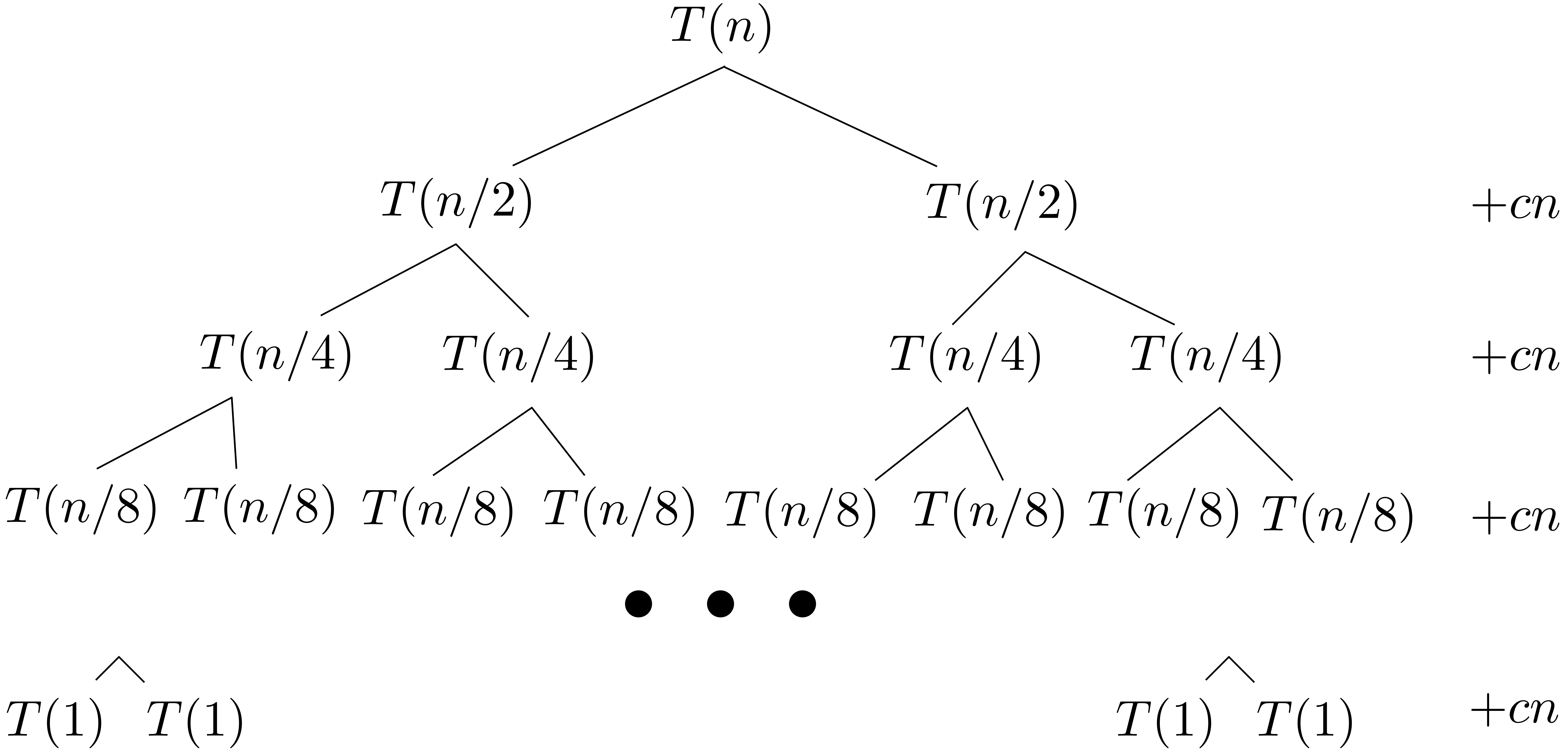


Recursion Tree

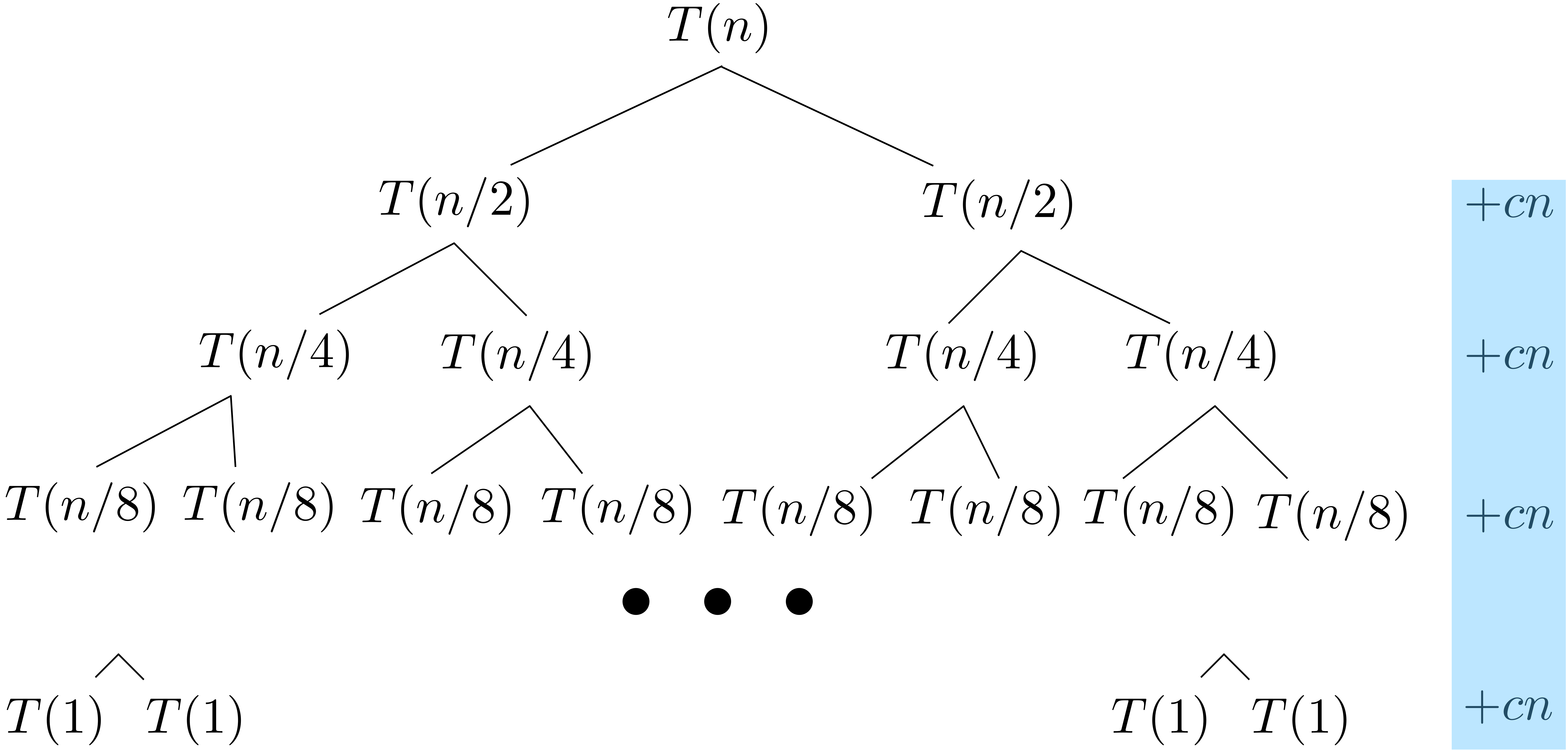




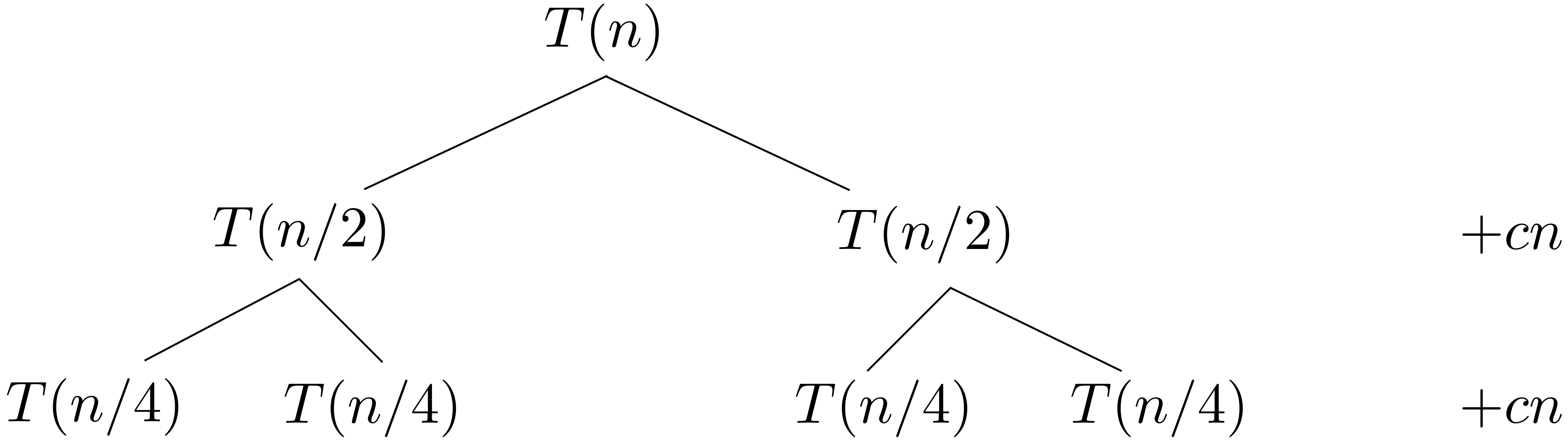
T(n)
T(n/2) T(n/2) +cn
T(n/4) T(n/4) T(n/4) T(n/4) +cn
T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) +cn

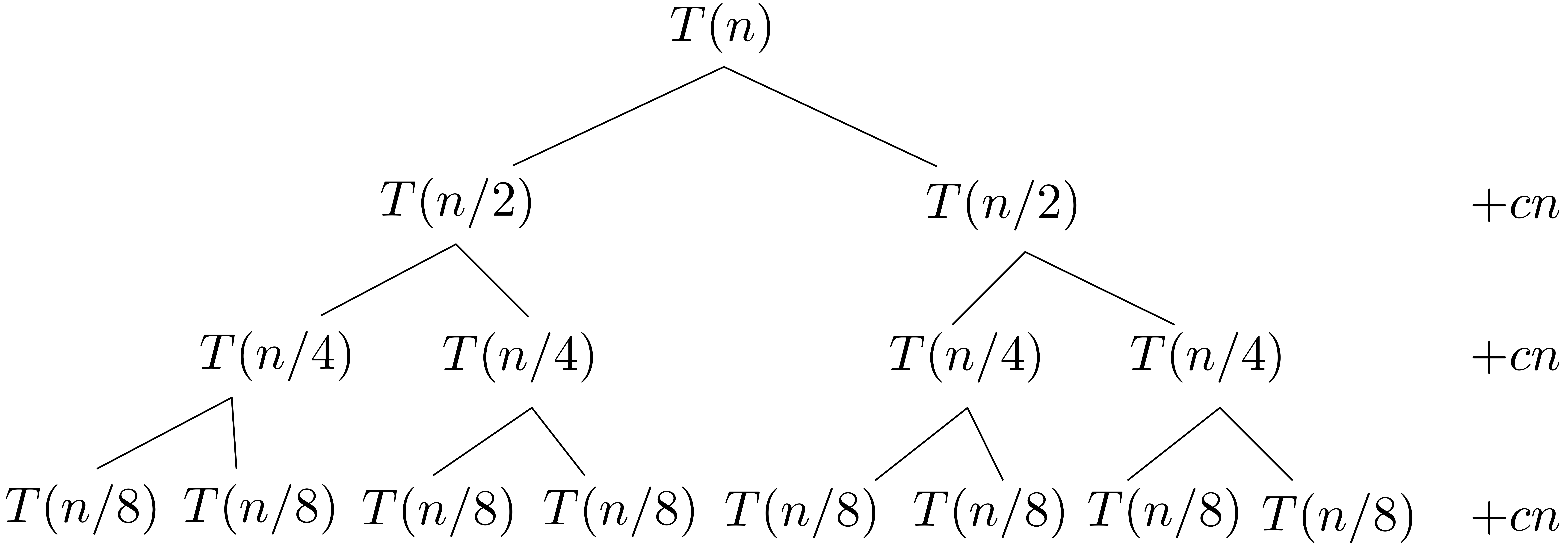


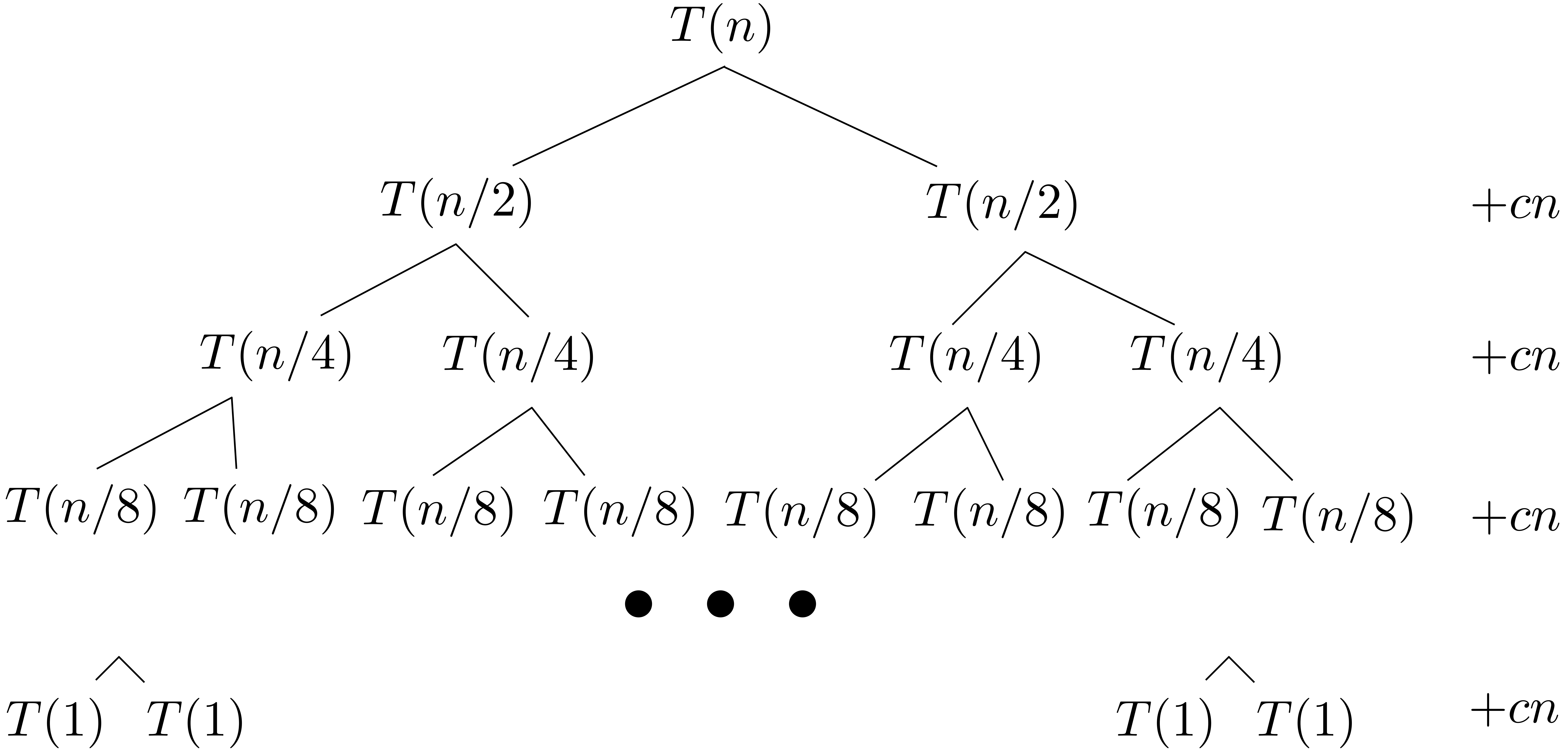
$T(n)$
$T(n/2)$ $T(n/2)$ $+cn$
$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$ $+cn$
$T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $+cn$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $+cn$



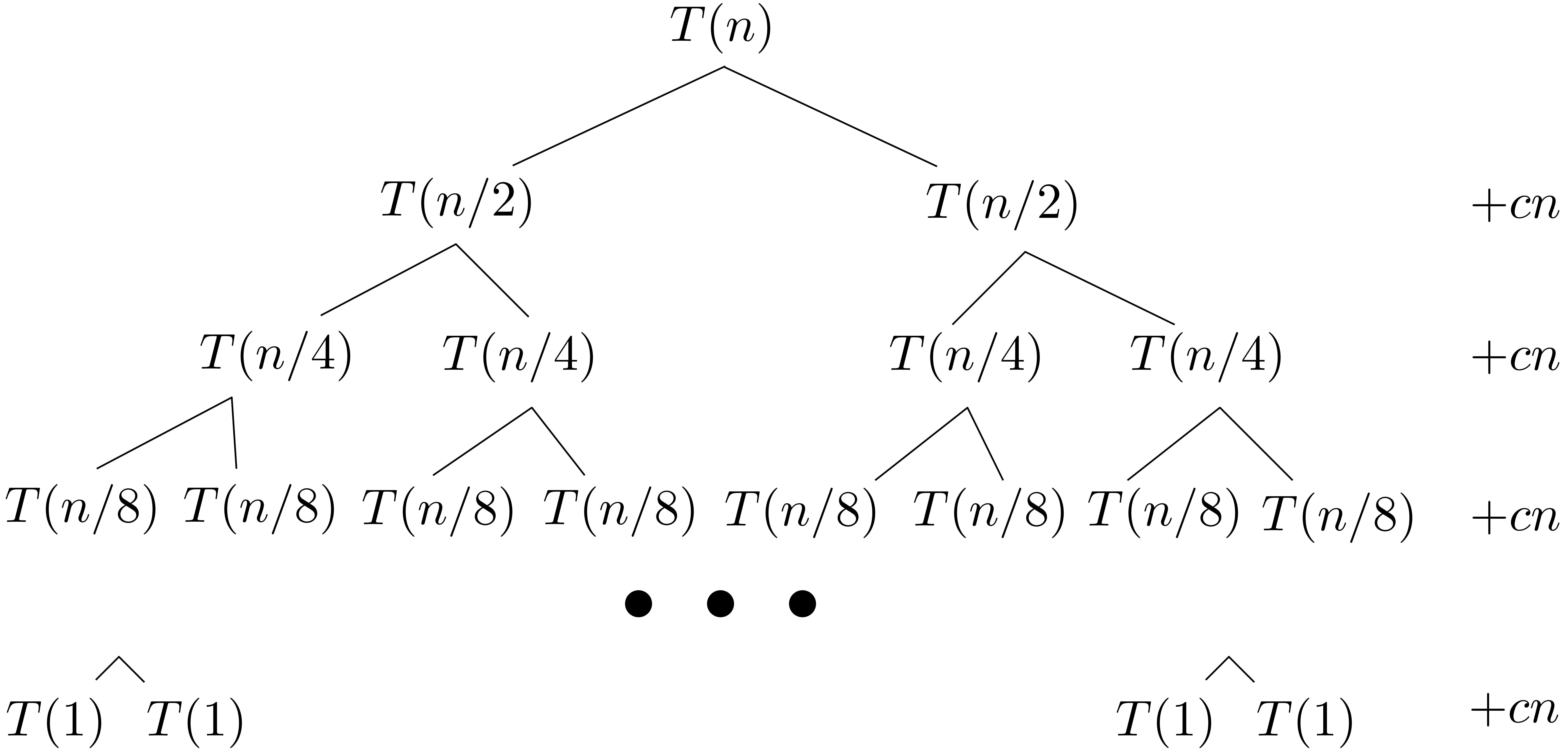
There are $\log n$ levels. The "complete" term contributes $cn \log n$.


$T(n)$
$T(n/2)$
$T(n/2)$
$+cn$
$T(n/4)$
$T(n/4)$
$T(n/4)$
$T(n/4)$
$+cn$


T(n)
T(n/2) T(n/2) +cn
T(n/4) T(n/4) T(n/4) T(n/4) +cn
T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) T(n/8) +cn



$T(n)$
$T(n/2)$ $T(n/2)$ $+cn$
$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$ $+cn$
$T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $T(n/8)$ $+cn$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $+cn$



We have n terms of $T(1)$ at the bottom level. This contributes $O(n)$.

Summary

$$T(n) = 2T(n/2) + O(n)$$

$$T(1) = O(1)$$ base case

The solution to this recurrence is $T(n) = O(n \log n)$.

The running time of our divide and conquer algorithm for the best time to buy and sell stock is $O(n \log n)$.

Mergesort

Mergesort

Mergesort is a comparison based sorting algorithm with worst-case running time $\Theta(n \log n)$.

This is optimal for a comparison-based method.

Mergesort is stable but is not in place.

Mergesort is a great example of a divide and conquer algorithm.

Building Block

The heart of mergesort is merging together two sorted arrays.

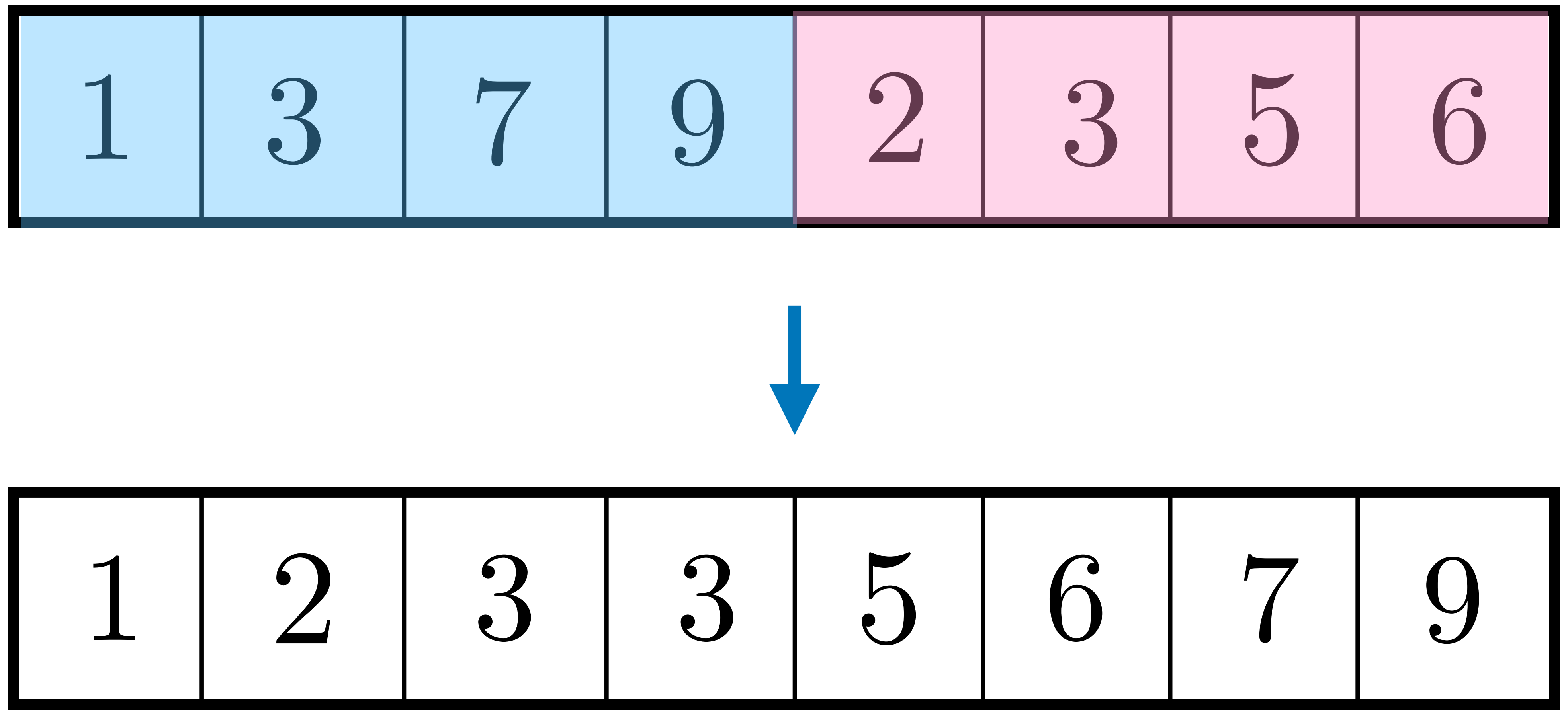
Say we have an array of size n where the first half is sorted and the second half is sorted.

1	3	7	9	2	3	5	6

Building Block

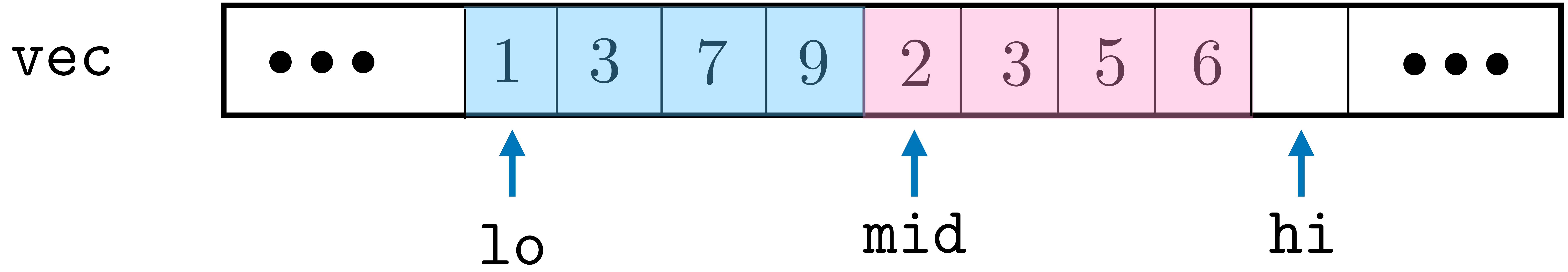
The heart of mergesort is merging together two sorted arrays.

Say we have an array of size n where the first half is sorted and the second half is sorted.



We want to merge these to completely sort the array.

Merge Function



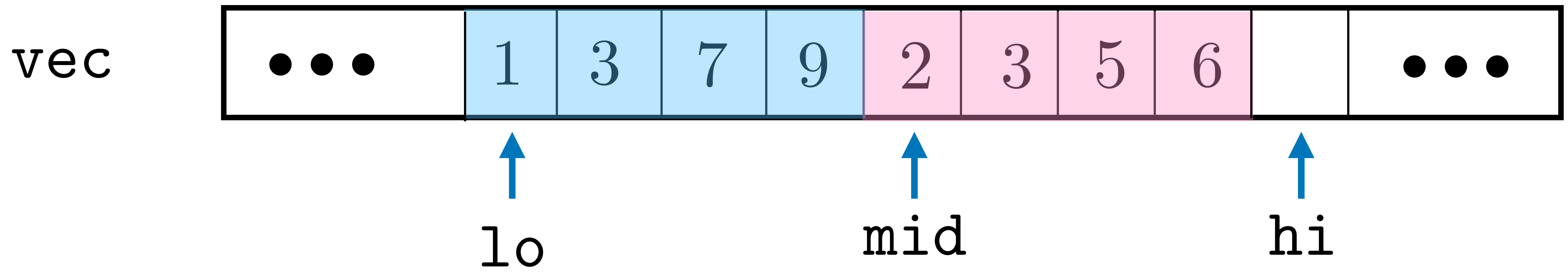
Let us specify in more detail what the merge function should do:

We are given three iterators `lo` $\leq$ `mid` $\leq$ `hi` into a vector `vec` .

We are promised that `vec` is sorted in $[\texttt{lo}, \texttt{mid})$, that is from `lo` up to but not including `mid` .

This is called a half-closed interval.

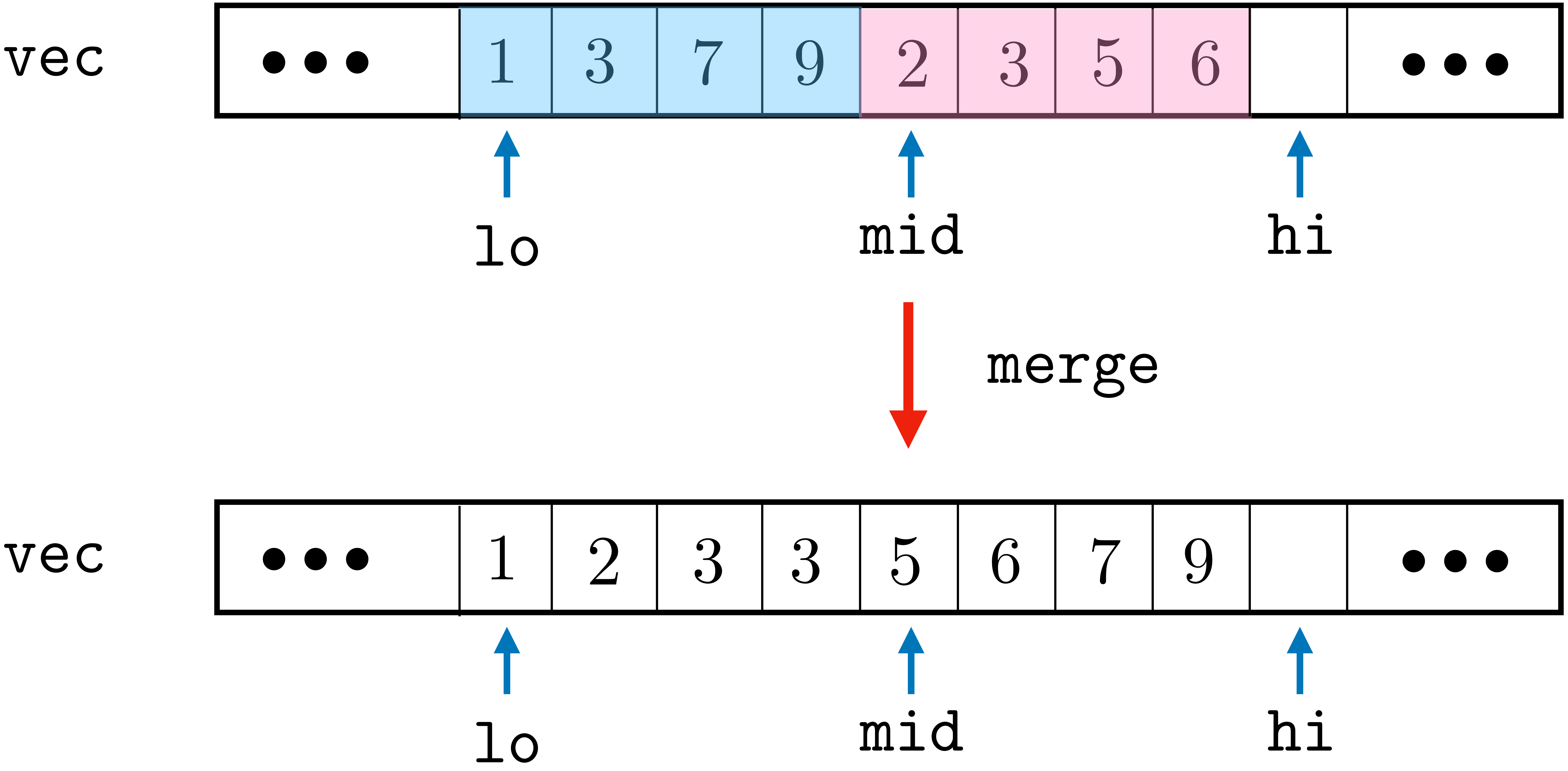
Merge Function



We are given three iterators $\texttt{lo} \leq \texttt{mid} \leq \texttt{hi}$ into a vector `vec`.

We are also promised that `vec` is sorted in $[\texttt{mid}, \texttt{hi})$, that is from `mid` up to but not including `hi`.

Merge: Goal

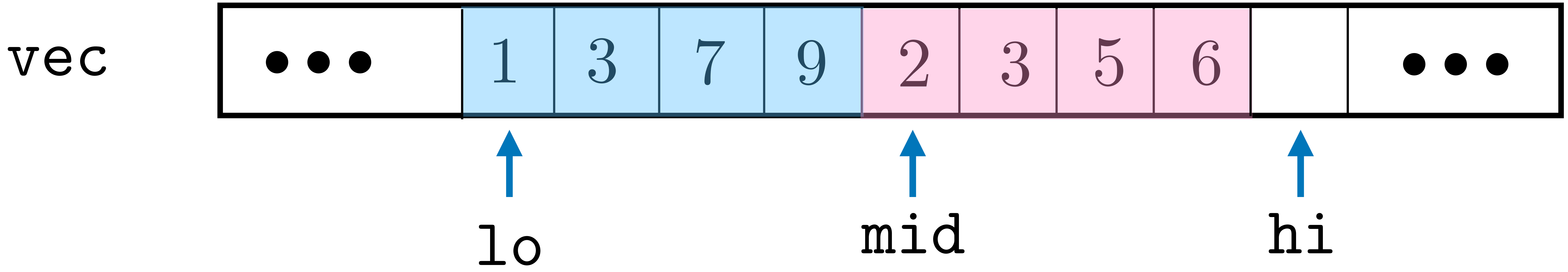


After `merge` the vector should be sorted in the interval $[\texttt{lo}, \texttt{hi})$.

Merge: Signature

```
using vecIt = std::vector<int>::iterator;

// Assumptions: lo <= mid <= hi
// Vector is sorted in [lo, mid) and [mid, hi)
// Result: After merge, vector is sorted in [lo, hi)
void merge(vecIt lo, vecIt mid, vecIt hi);
```

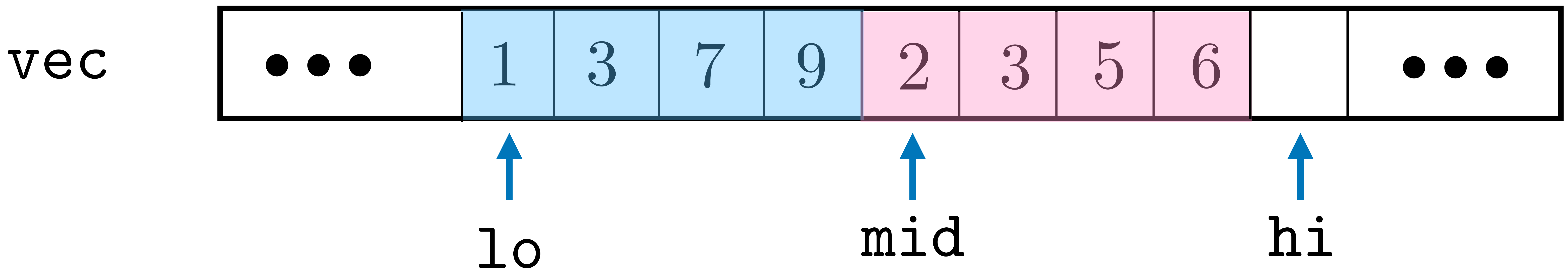


Merge: Signature

```
using vecIt = std::vector<int>::iterator;

// Assumptions: lo <= mid <= hi
// Vector is sorted in [lo, mid) and [mid, hi)
// Result: After merge, vector is sorted in [lo, hi)
void merge(vecIt lo, vecIt mid, vecIt hi);
```

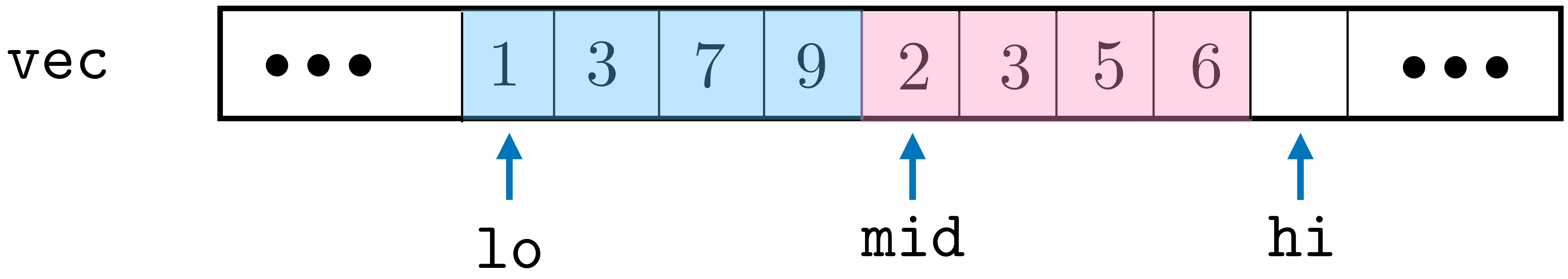
← using declaration



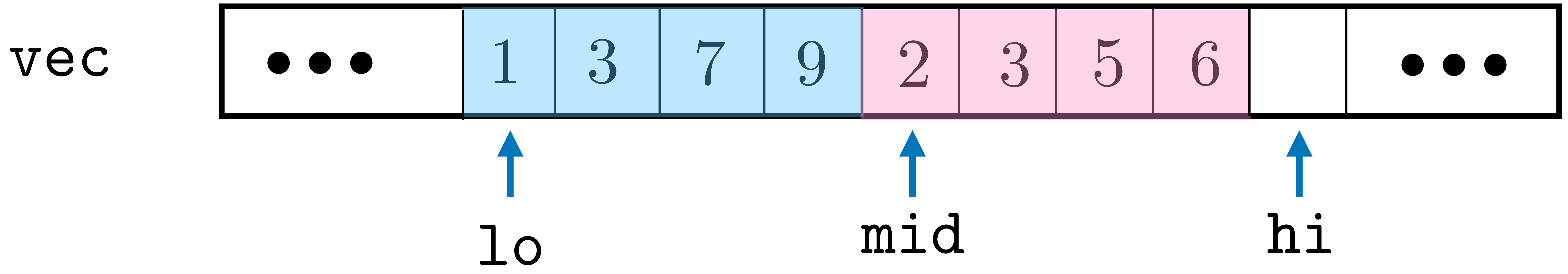
Merge: Signature

```
using vecIt = std::vector<int>::iterator;

// Assumptions: lo <= mid <= hi
// Vector is sorted in [lo, mid) and [mid, hi)
// Result: After merge, vector is sorted in [lo, hi)
void merge(vecIt lo, vecIt mid, vecIt hi);
```



Merge: Complexity



We can implement `merge` to run in time $\Theta(\texttt{hi} - \texttt{lo})$ and to use $\Theta(\texttt{hi} - \texttt{lo})$ additional space.

I'm going to leave this as an exercise.

Now let's continue designing mergesort using `merge` as a black box.

Mergesort

How can we sort this vector making use of the merge subroutine?

7	3	2	5	6	3	1	9

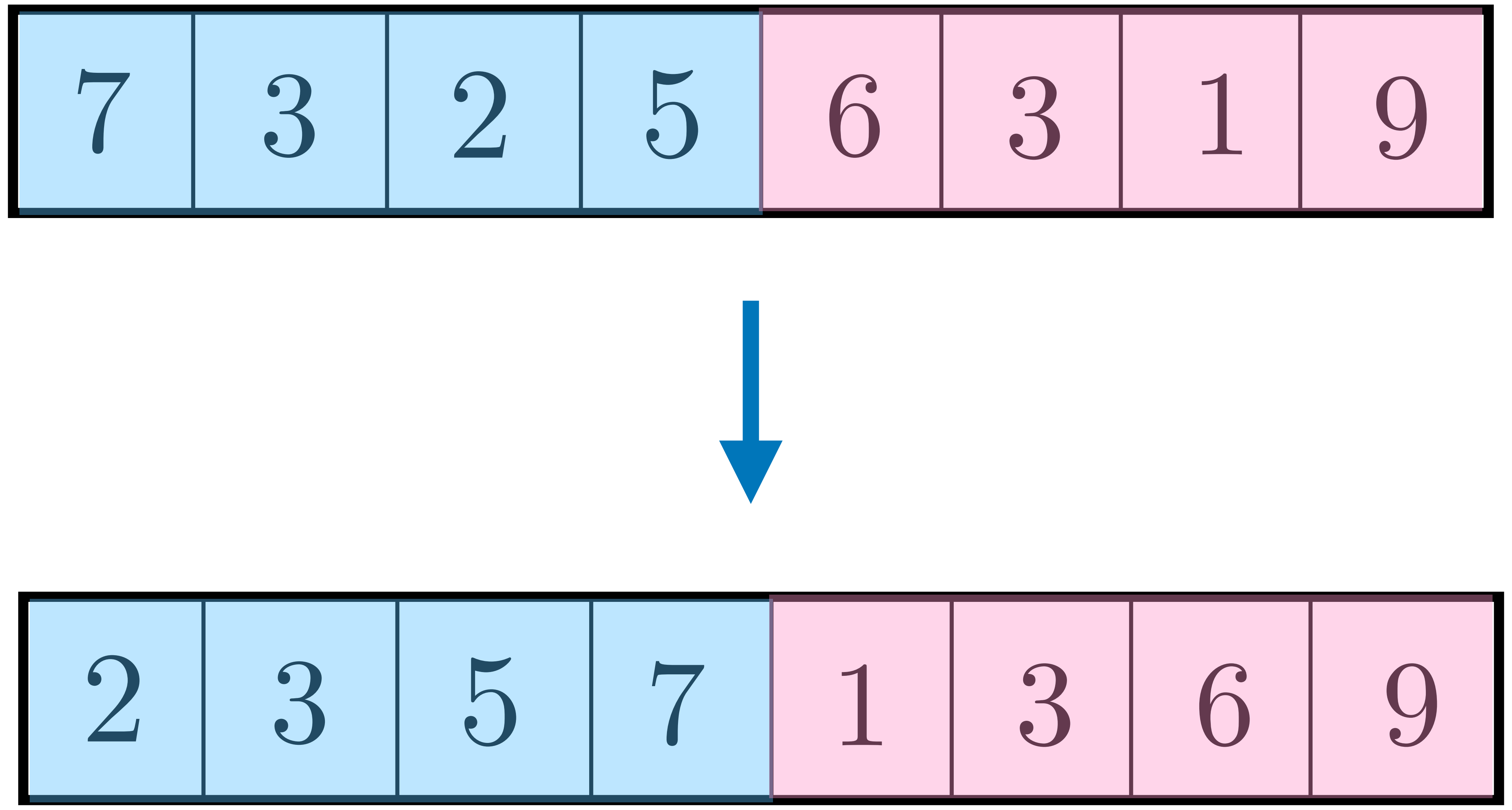
Mergesort

How can we sort this vector making use of the merge subroutine?

7	3	2	5	6	3	1	9

Mergesort

How can we sort this vector making use of the merge subroutine?



Sort the left
half and the right half.

Mergesort

How can we sort this vector making use of the merge subroutine?

7	3	2	5	6	3	1	9

Sort the left half and the right half.

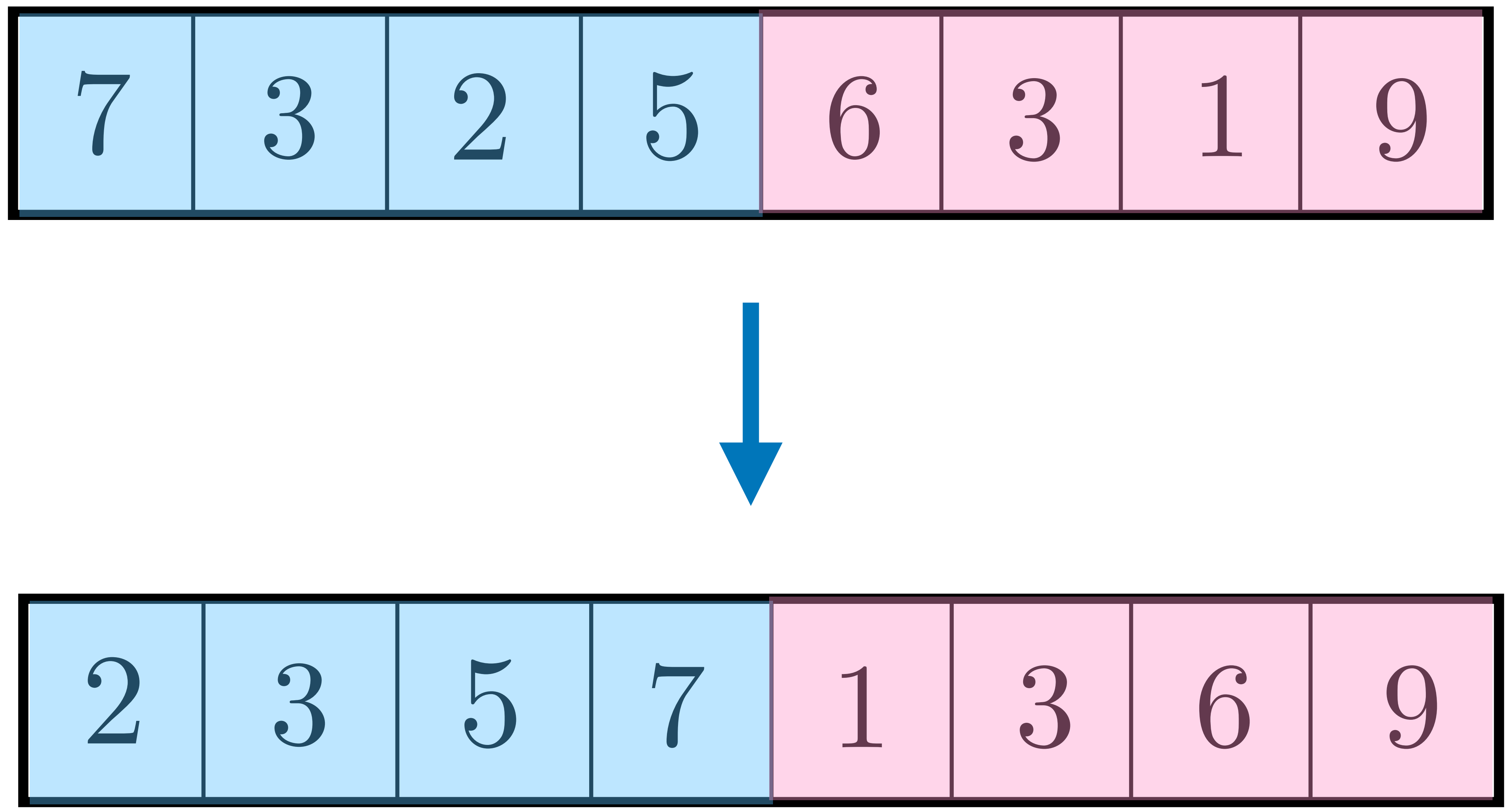
2	3	5	7	1	3	6	9

Merge the two sorted halves.

1	2	3	3	5	6	7	9

Mergesort

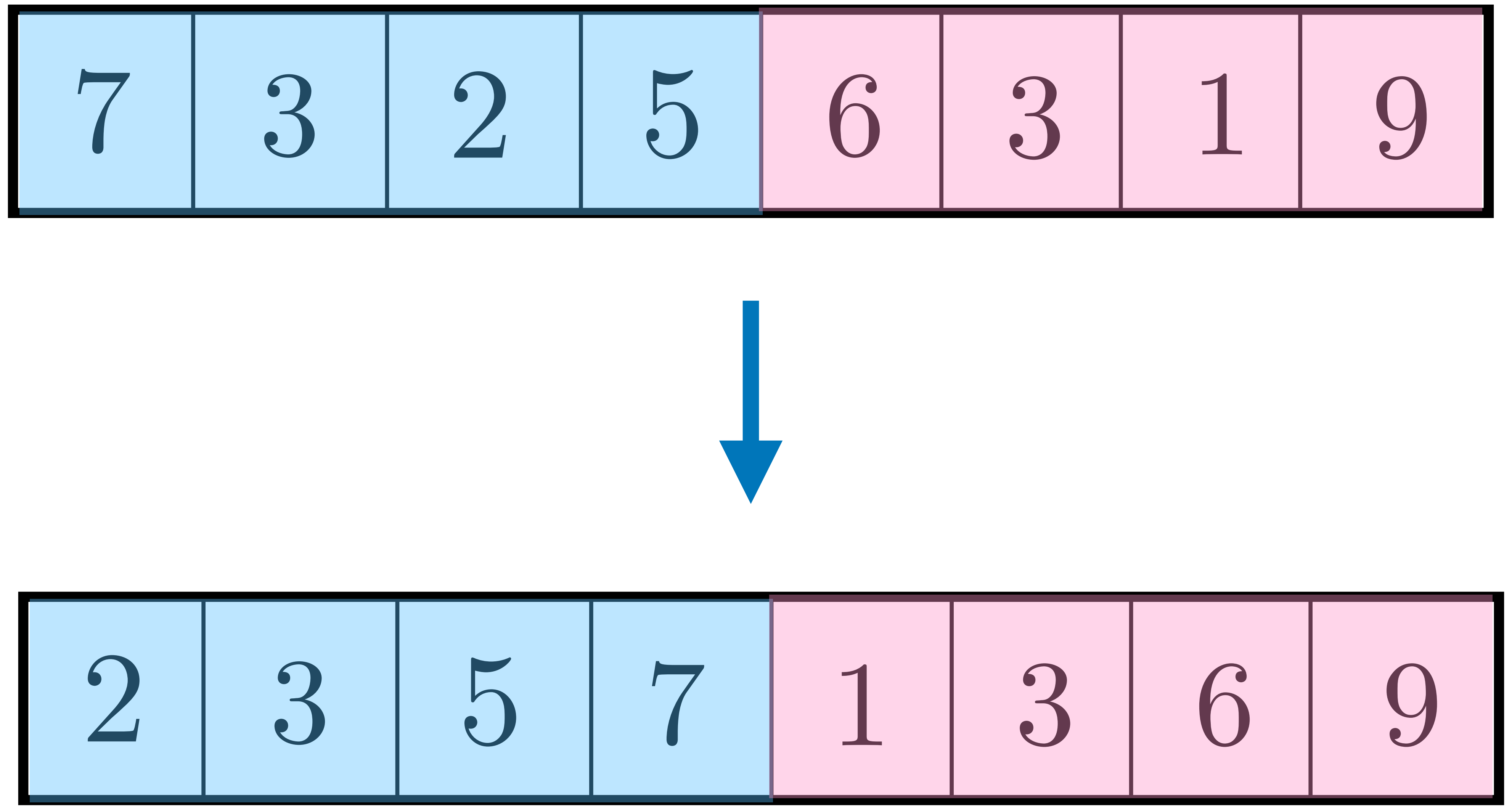
How can we sort this vector making use of the merge subroutine?



Sort the left
half and the right half.

Mergesort

How can we sort this vector making use of the merge subroutine?

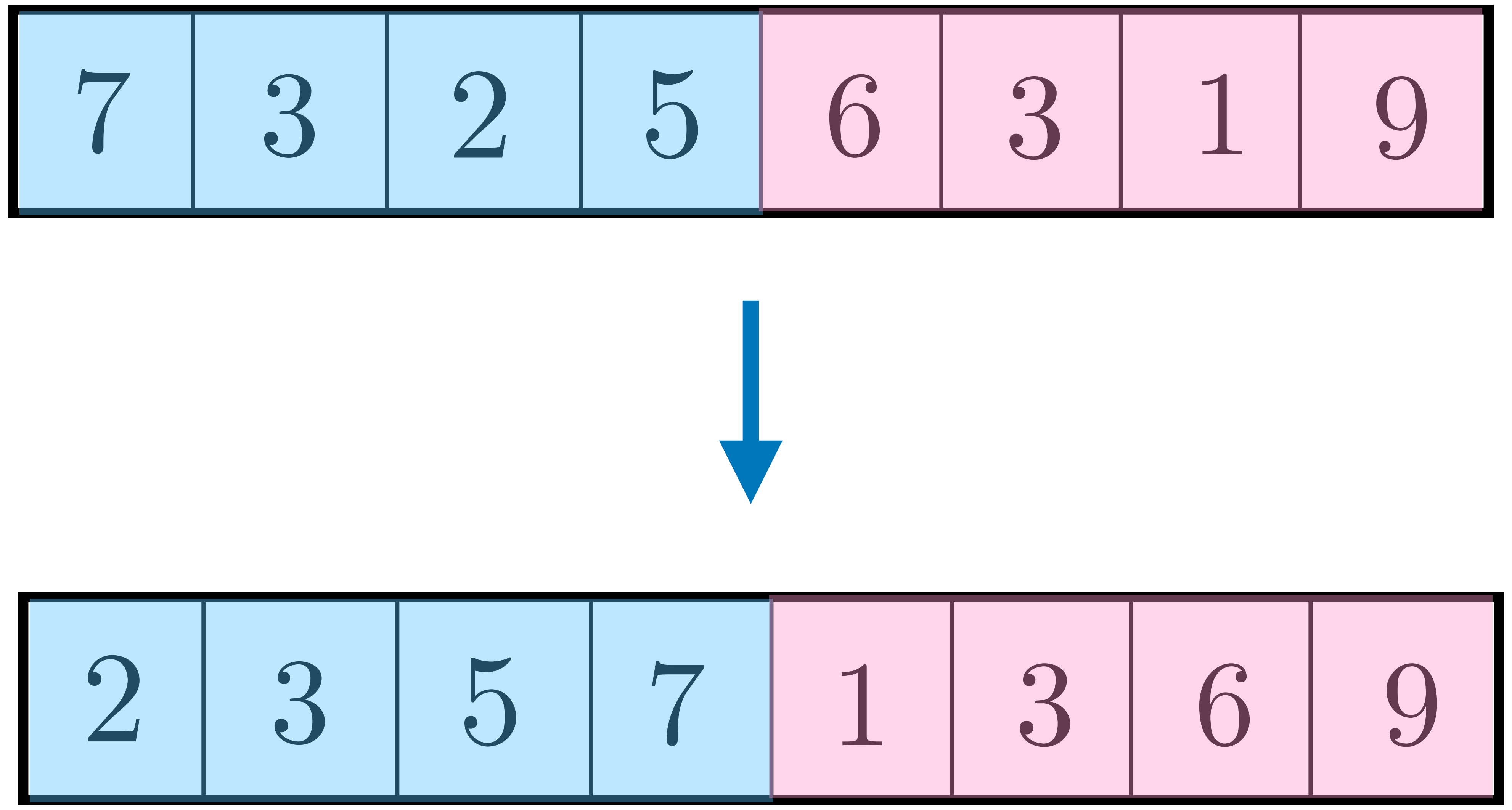


Sort the left half and the right half.

How do we sort the left and right halves?

Mergesort

How can we sort this vector making use of the merge subroutine?



Sort the left
half and the right half.

How do we sort the left and right halves?

Use mergesort!

Mergesort: D&C

Let's put mergesort in the context of divide and conquer algorithms.

7	3	2	5	6	3	1	9

Original problem: sort a vector of size n.

Divide: Sort the first half and sort the second half.

Two subproblems of size roughly $n/2$.

Mergesort: D&C

Let's put mergesort in the context of divide and conquer algorithms.

7	3	2	5	6	3	1	9

Original problem: sort a vector of size n.

Divide: Sort the first half and sort the second half.

Two subproblems of size roughly $n/2$.

2	3	5	7	1	3	6	9

Mergesort: D&C

2	3	5	7	1	3	6	9

Divide: Sort the first half and sort the second half.

Let's look at the work to Create, Complete, and Combine

Create: The subproblems are the first half and second half of the vector.

Easy! We just have to compute the midpoint.

Mergesort: D&C

2	3	5	7	1	3	6	9

Divide: Sort the first half and sort the second half.

Let's look at the work to Create, Complete, and Combine

Complete: We don't have to do any work here.

All the information we need is in solution to the subproblems.

Mergesort: D&C

2	3	5	7	1	3	6	9

Divide: Sort the first half and sort the second half.

Let's look at the work to Create, Complete, and Combine.

Combine: Combine solutions to subproblems to solve original problem.

This is done by the `merge` function!

The combine step is where the main work of mergesort is done.

Mergesort: Code

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

Sort the vector in $[\texttt{begin}, \texttt{end})$

Mergesort: Code

Let $T(n)$ be the running time of `mergesort` when $\texttt{end} - \texttt{begin} = n$.

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

← Base case of the recursion.
A vector of size one is already sorted.

$$T(1) = O(1)$$

Mergesort: Code

Let $T(n)$ be the running time of `mergesort` when $\texttt{end} - \texttt{begin} = n$.

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

← Create step: Find the midpoint.

Takes time $O(1)$.

Mergesort: Code

Let $T(n)$ be the running time of `mergesort` when $\texttt{end} - \texttt{begin} = n$.

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

Solve the subproblems.

← Sort the left half.

Mergesort: Code

Let $T(n)$ be the running time of `mergesort` when $\texttt{end} - \texttt{begin} = n$.

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

Solve the subproblems.

← Sort the right half.

Mergesort: Code

Let $T(n)$ be the running time of `mergesort` when $\texttt{end} - \texttt{begin} = n$.

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

Solve the subproblems.

← Sort the right half.

These two lines take time

$$T(\texttt{mid} - \texttt{begin}) + T(\texttt{end} - \texttt{mid})$$

Mergesort: Code

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

Combine step.

Merge the sorted intervals $[\texttt{begin}, \texttt{mid})$ and $[\texttt{mid}, \texttt{end})$

Time $O(\texttt{end} - \texttt{begin})$.

Mergesort: Code

```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

Total time:

$T(\texttt{mid} - \texttt{begin})$ solve subproblems

$+T(\texttt{end} - \texttt{mid})$

$+O(\texttt{end} - \texttt{begin})$ combine

$+O(1)$ create

Mergesort: Running Time

Let us assume the size of the original vector is a power of 2.

Then we have the recurrence

$$T(n) = 2T(n/2) + O(n)$$

$$T(1) = O(1)$$ base case

Mergesort: Running Time

Let us assume the size of the original vector is a power of 2.

Then we have the recurrence

$$T(n) = 2T(n/2) + O(n)$$

$$T(1) = O(1) \qquad \text{base case}$$

This is the exact same recurrence we had for the buy and sell stock problem.

Mergesort: Running Time

Let us assume the size of the original vector is a power of 2.

Then we have the recurrence

$$T(n) = 2T(n/2) + O(n)$$

$$T(1) = O(1)$$ base case

This is the exact same recurrence we had for the buy and sell stock problem.

The running time of mergesort is $O(n \log n)$.

Mergesort Example

Mergesort: Code

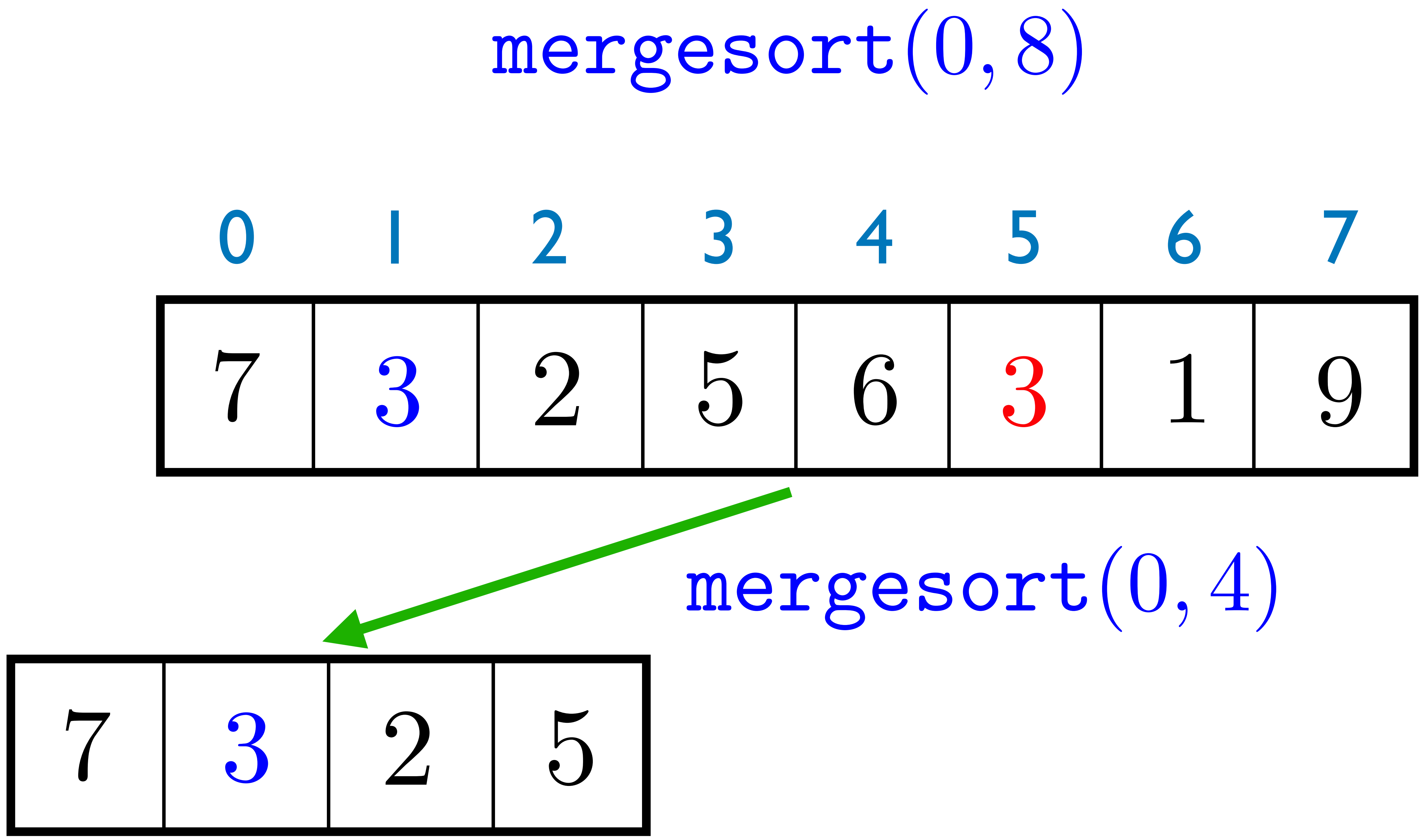
```
void mergesort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt mid = begin + (end - begin)/2;
  mergesort(begin, mid);
  mergesort(mid, end);
  merge(begin, mid, end);
}
```

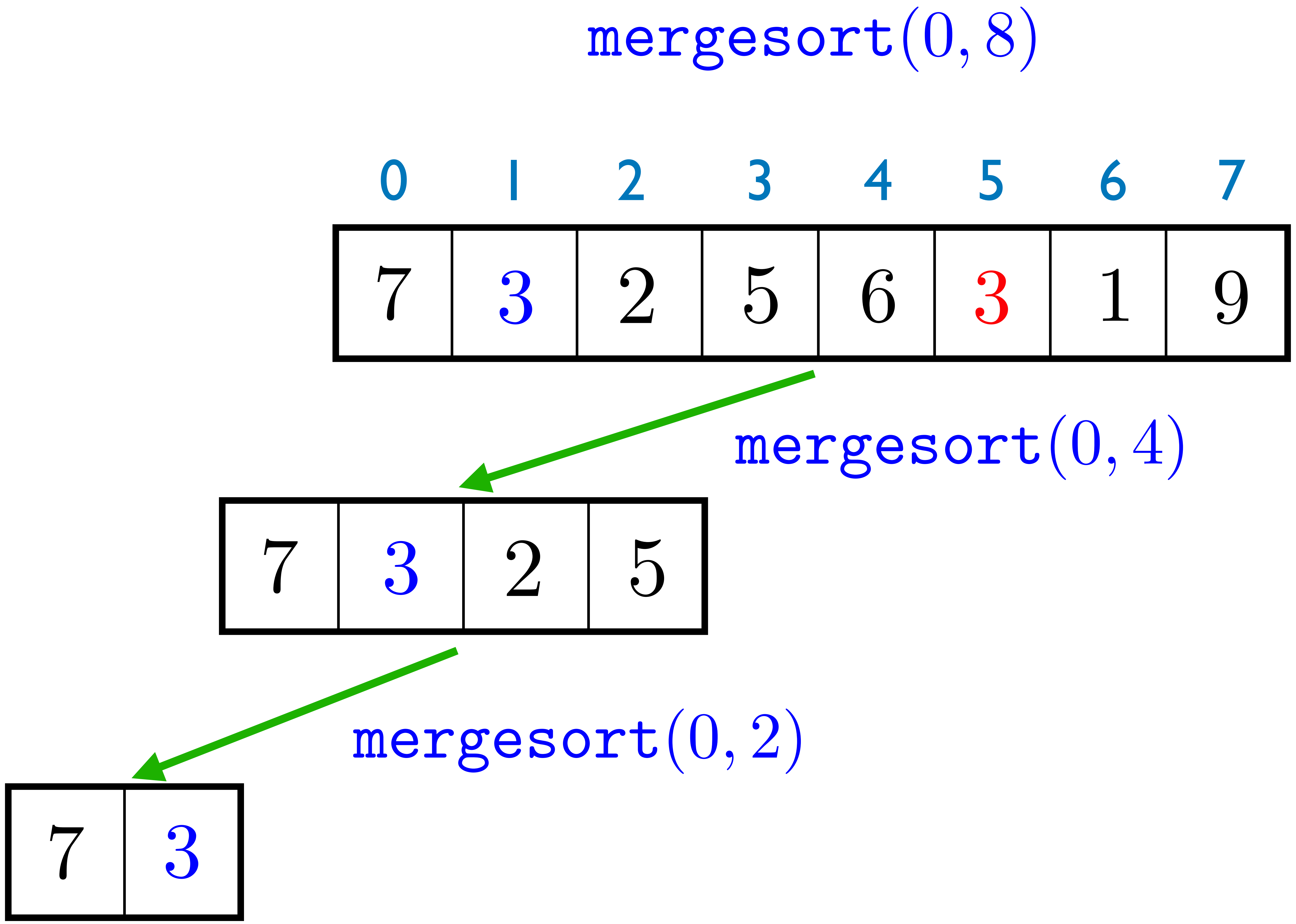
base case

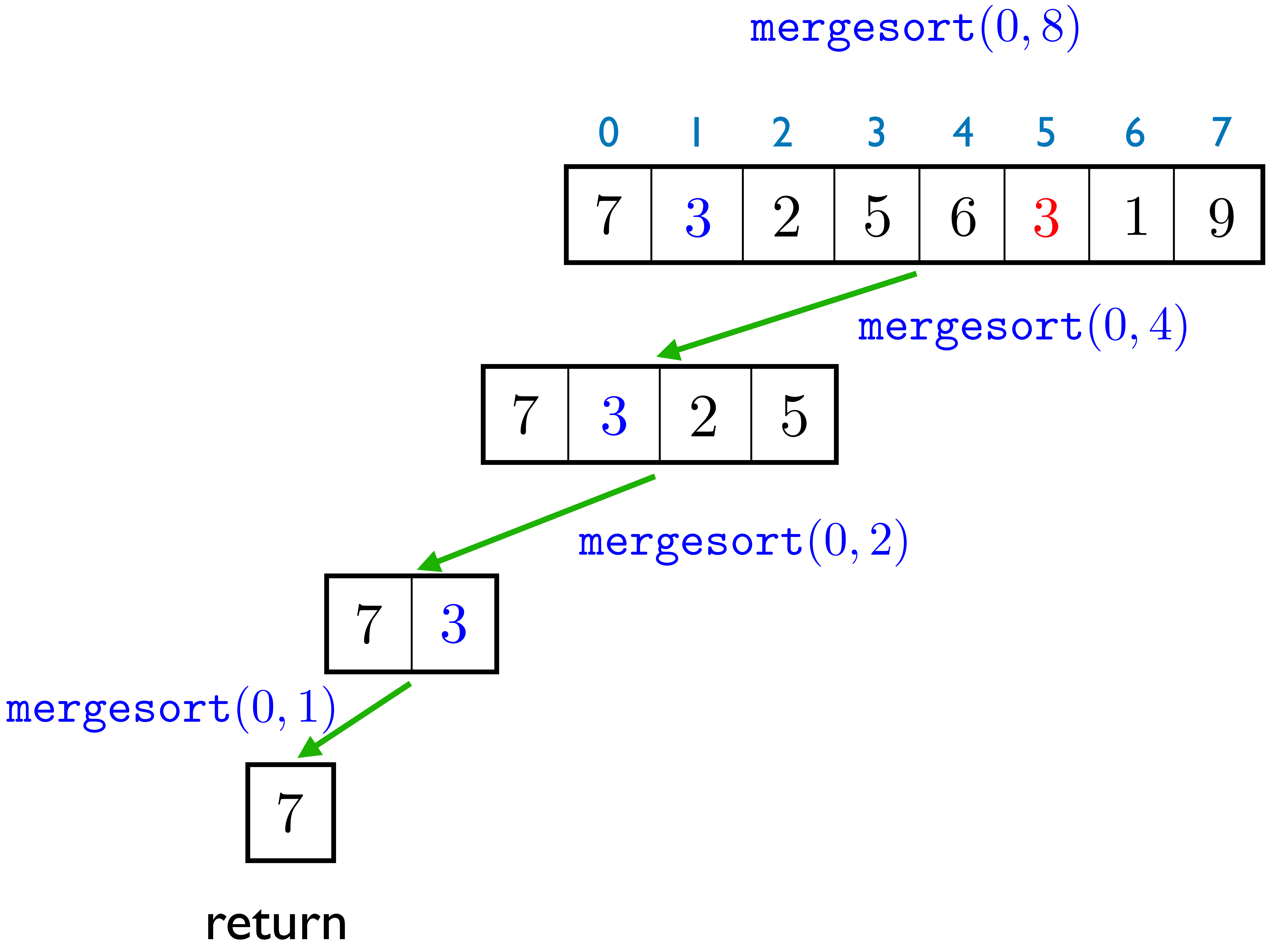
create subproblems
sort first half
sort second half
combine solutions with merge

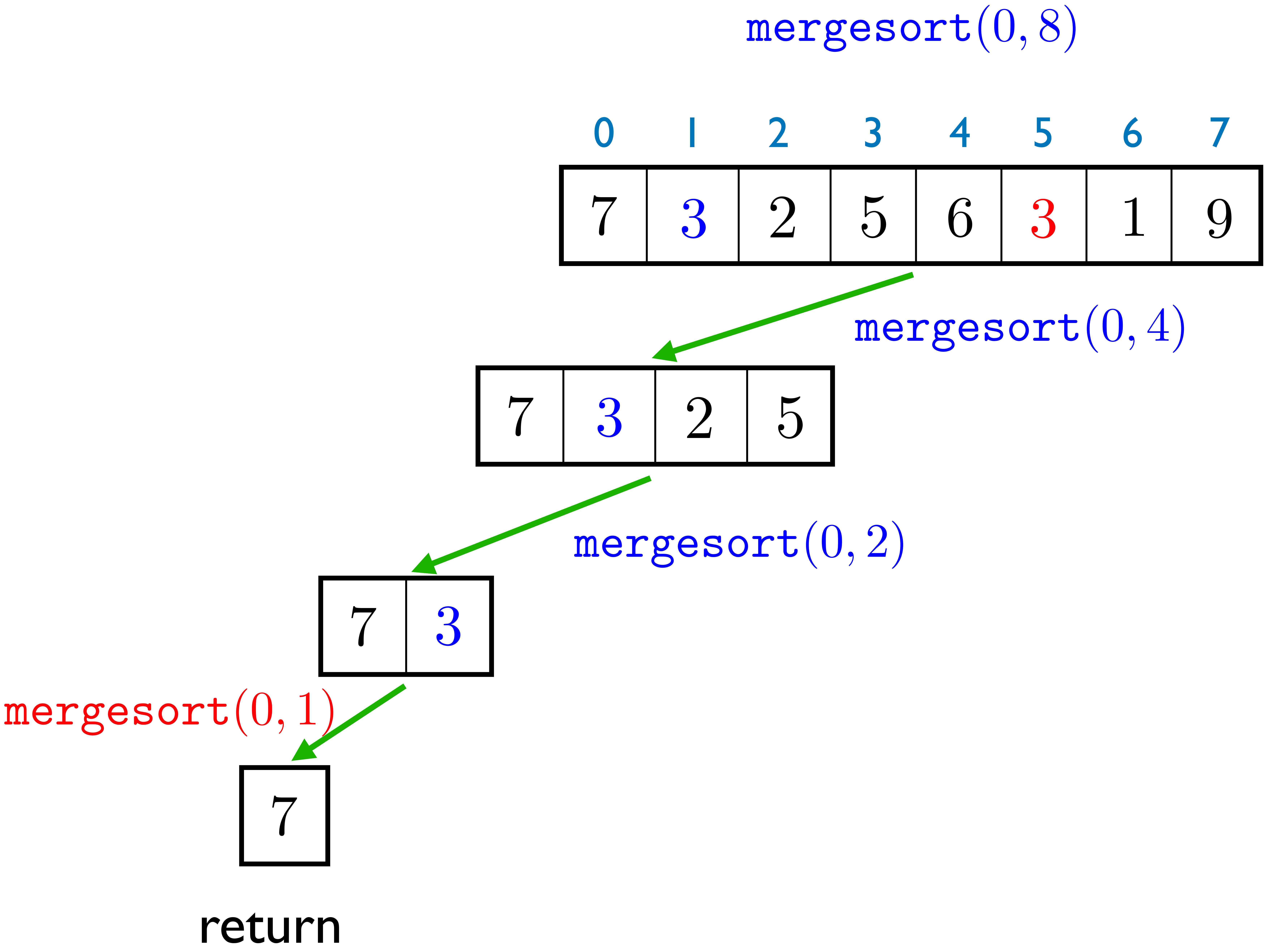
$\texttt{mergesort}(0, 8)$

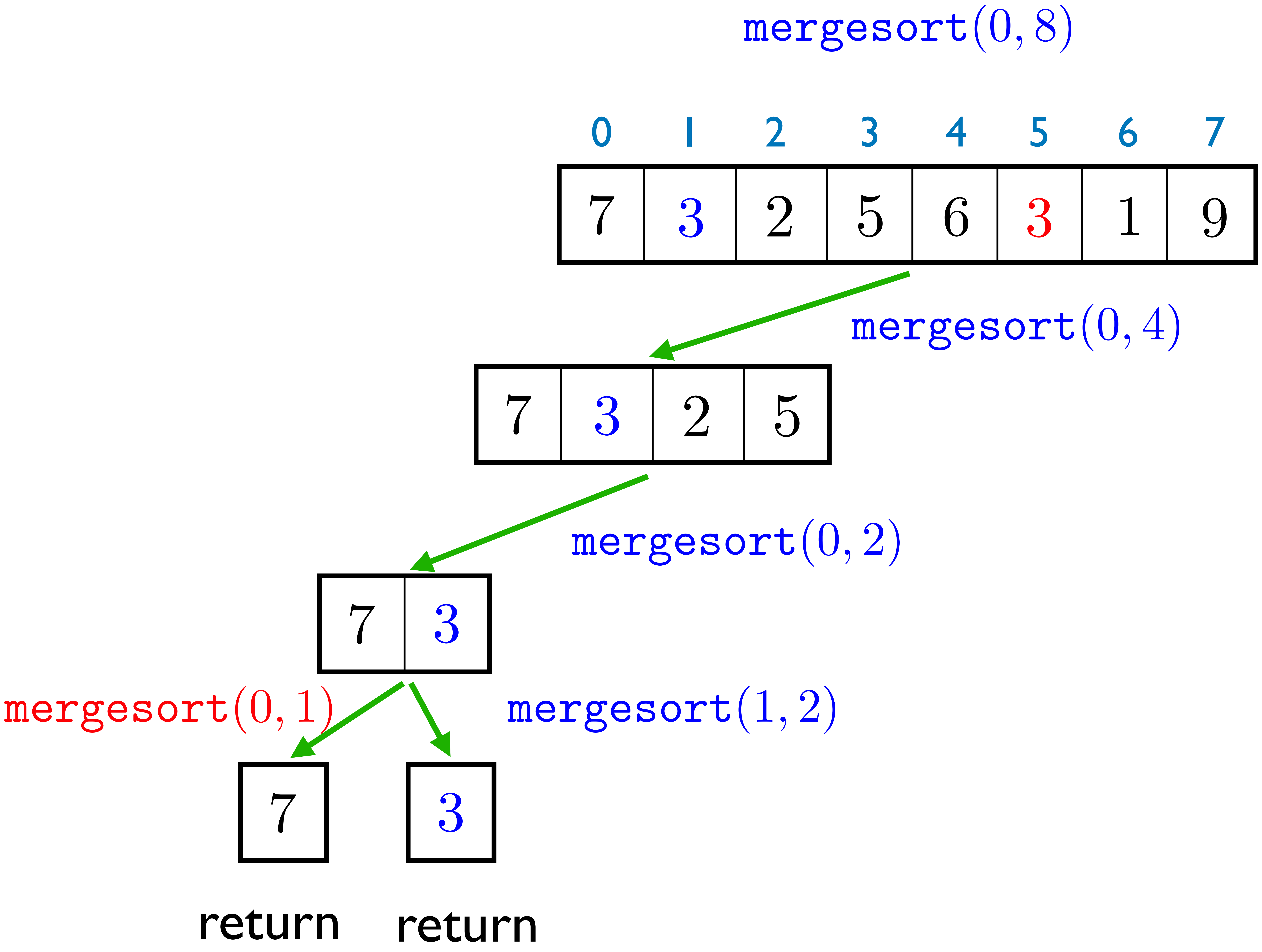
0	1	2	3	4	5	6	7
7	3	2	5	6	3	1	9


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5

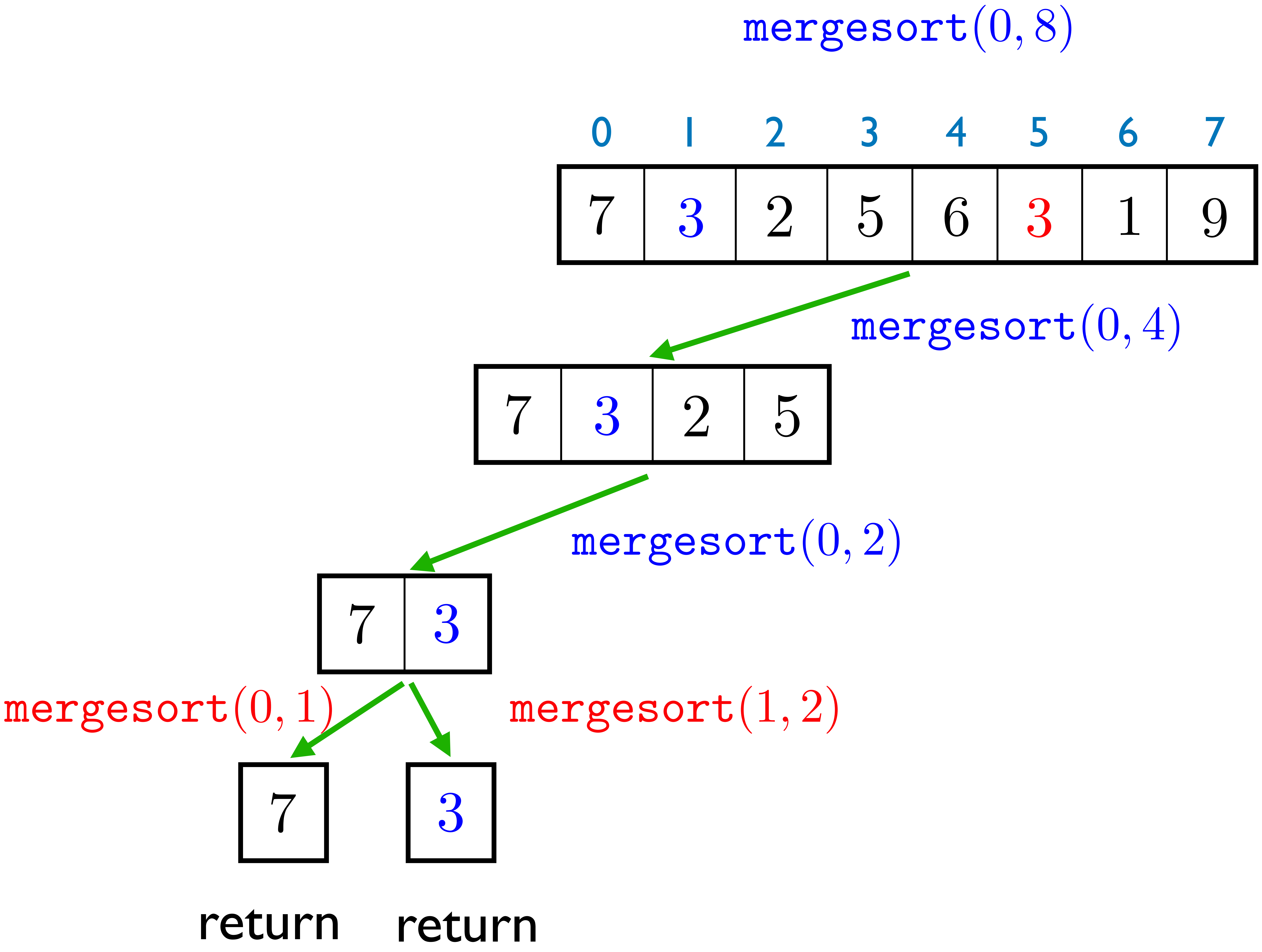

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(0, 2)
7 3

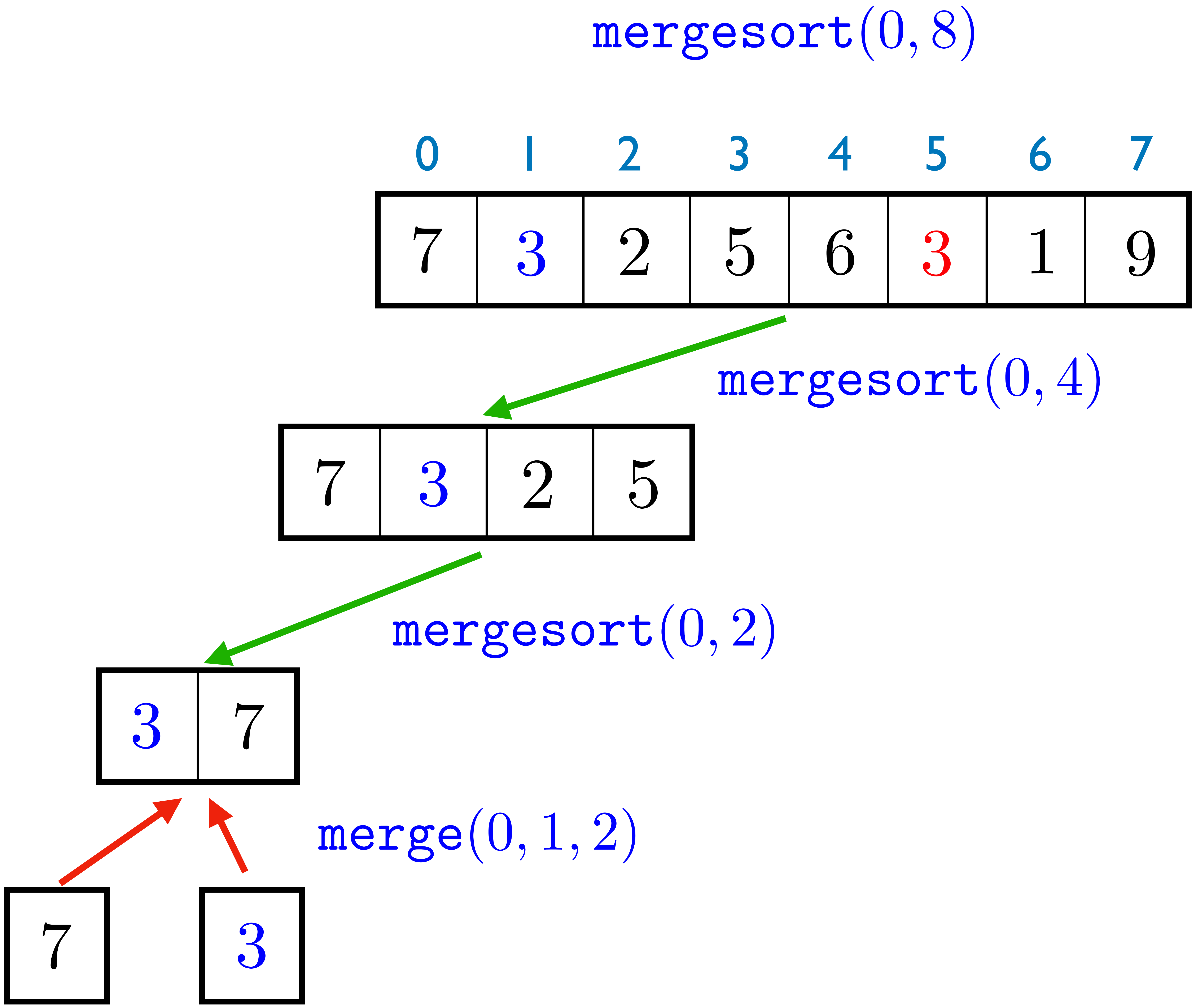

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(0, 2)
7 3
mergesort(0, 1)
7
return

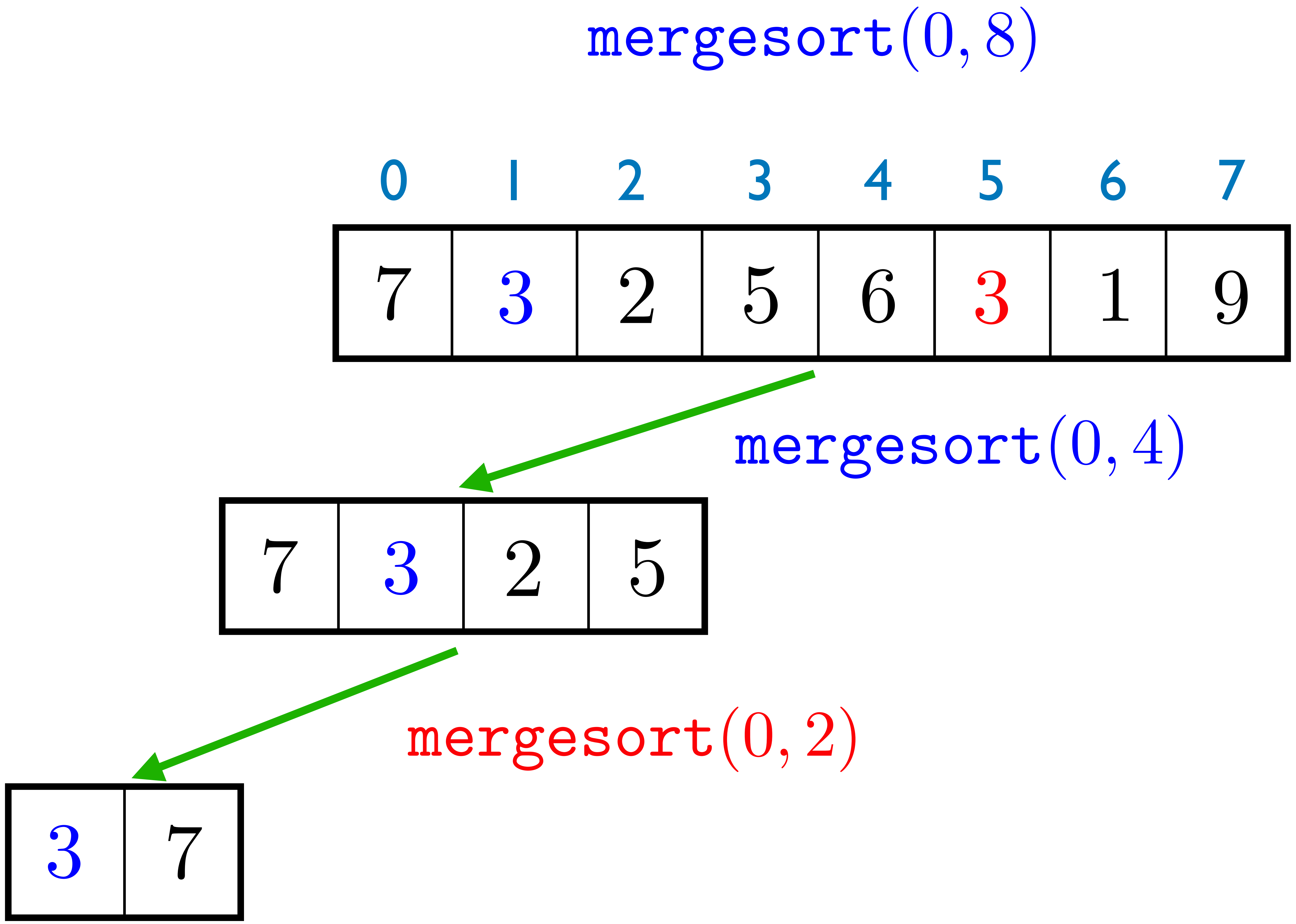

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(0, 2)
7 3
mergesort(0, 1)
7
return

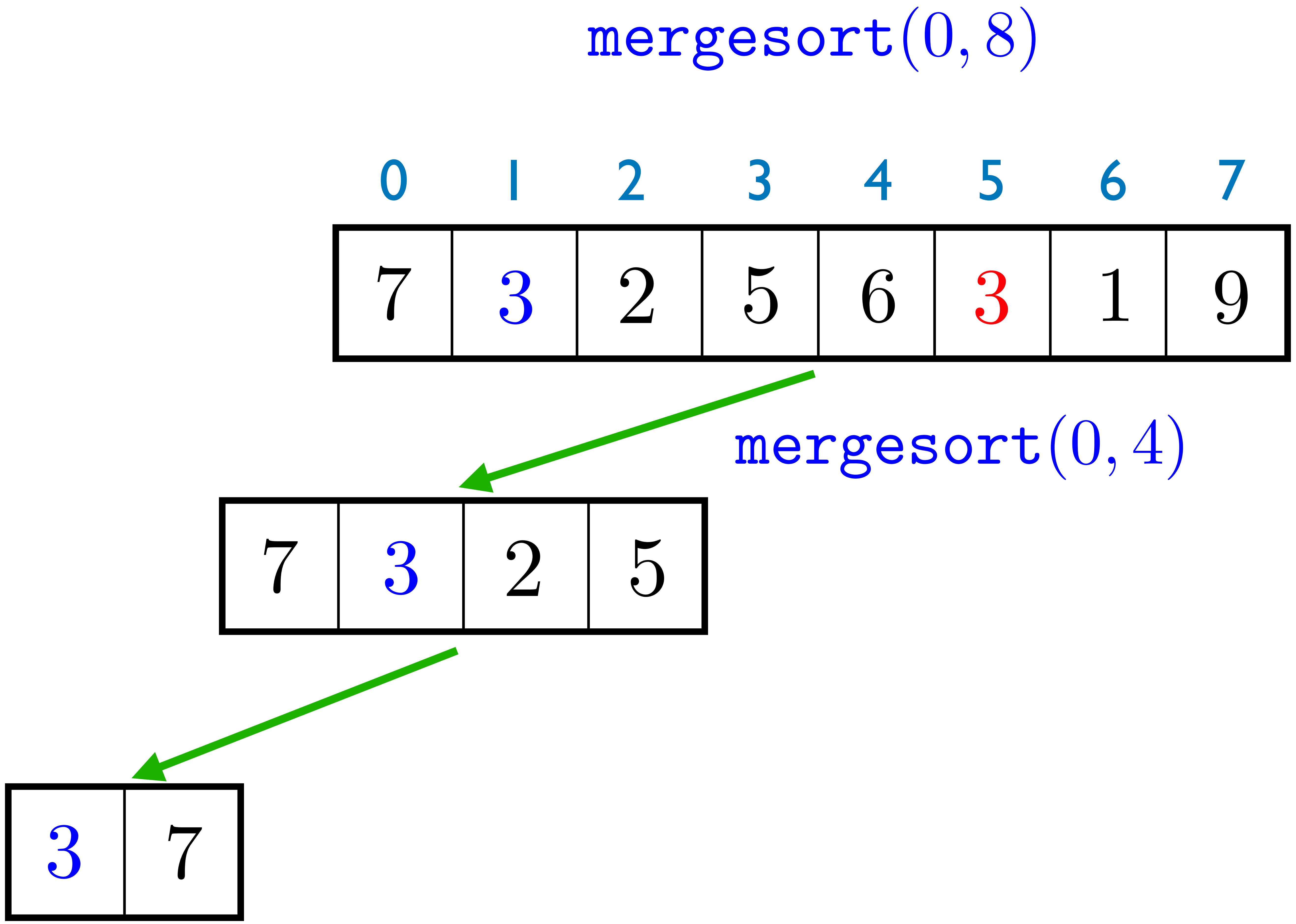


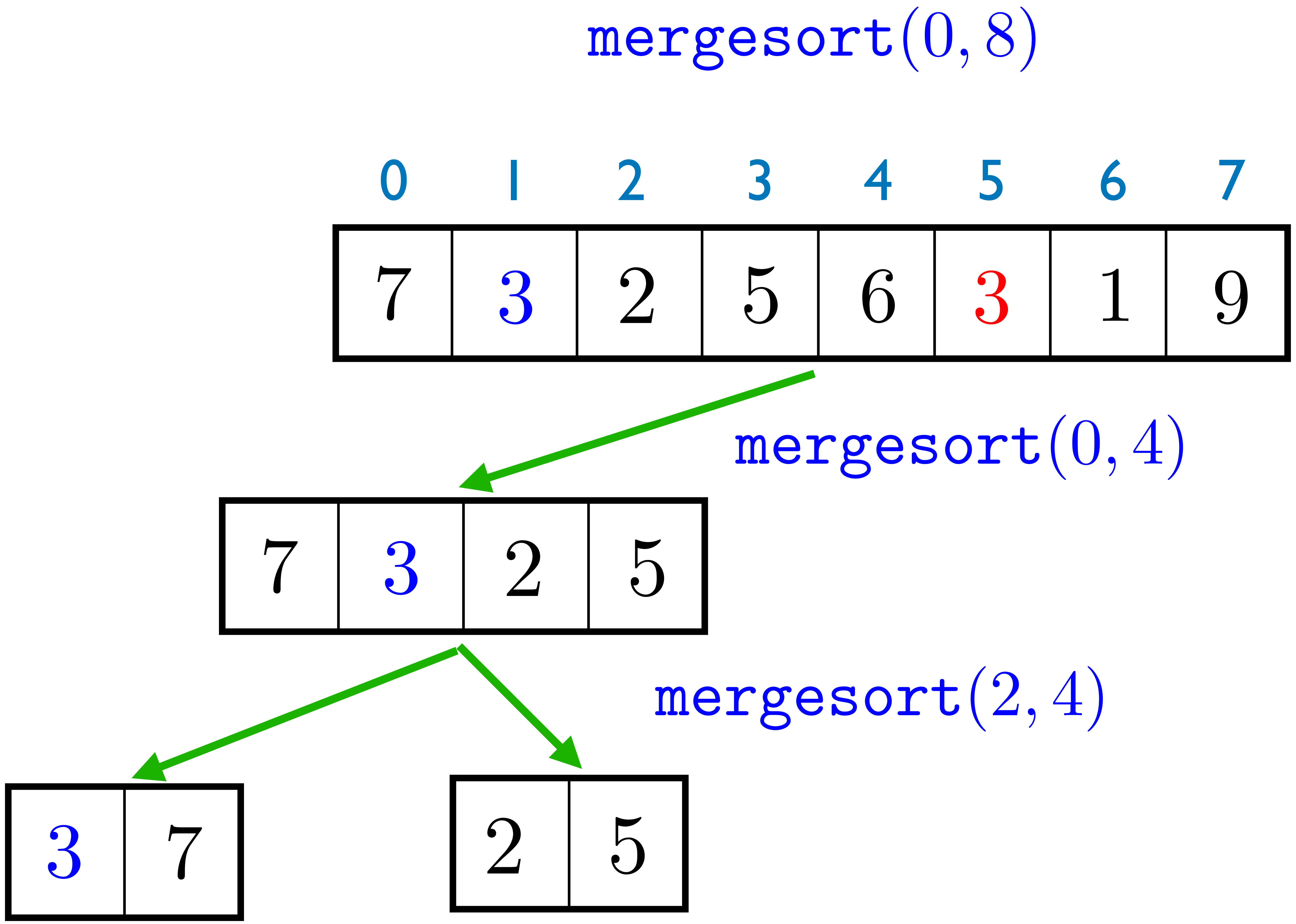
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(0, 2)
7 3
mergesort(0, 1)
mergesort(1, 2)
7
3
return
return


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(0, 2)
7 3
mergesort(0, 1)
mergesort(1, 2)
7
3
return
return

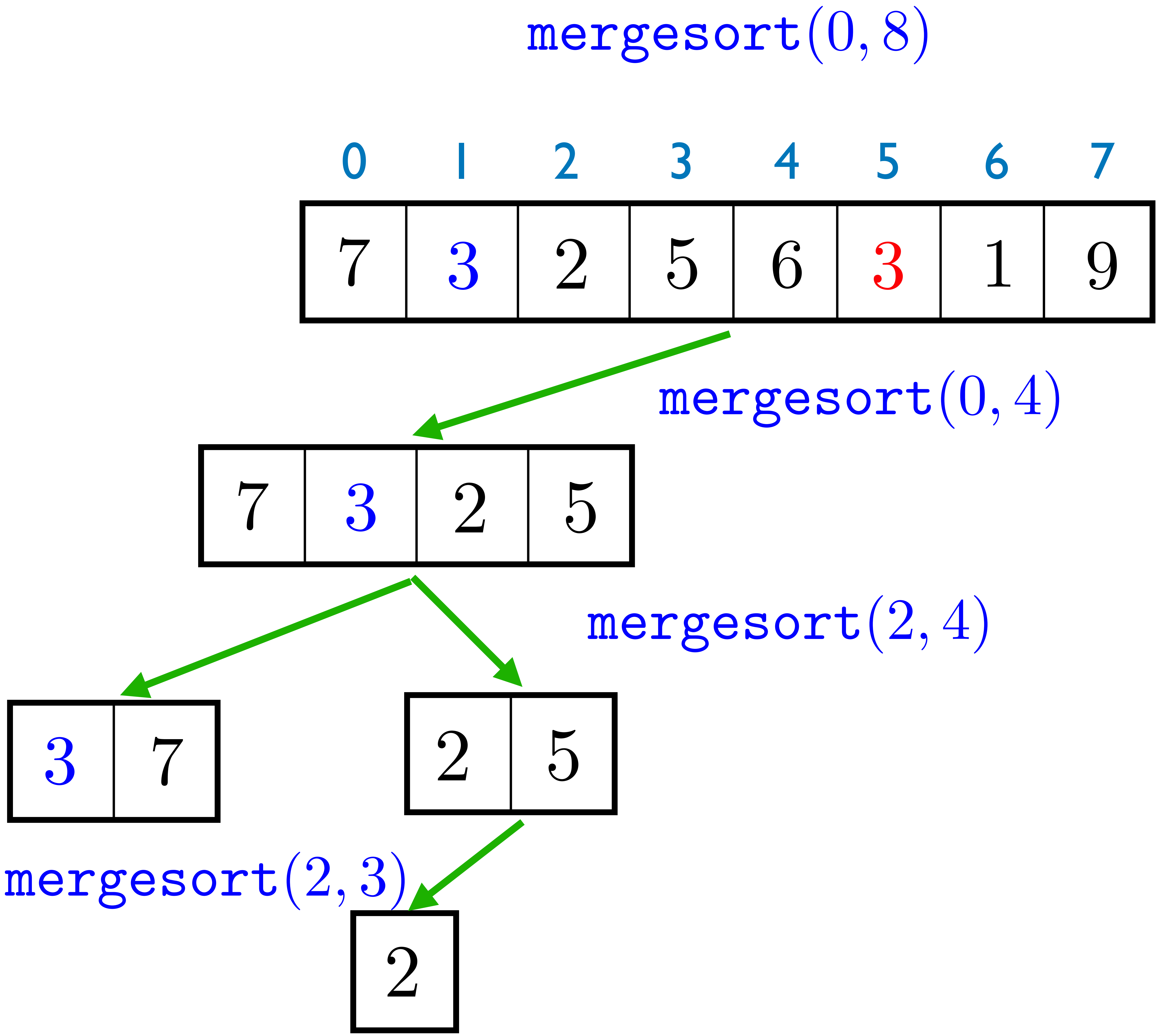

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(0, 2)
3 7
merge(0, 1, 2)
7
3

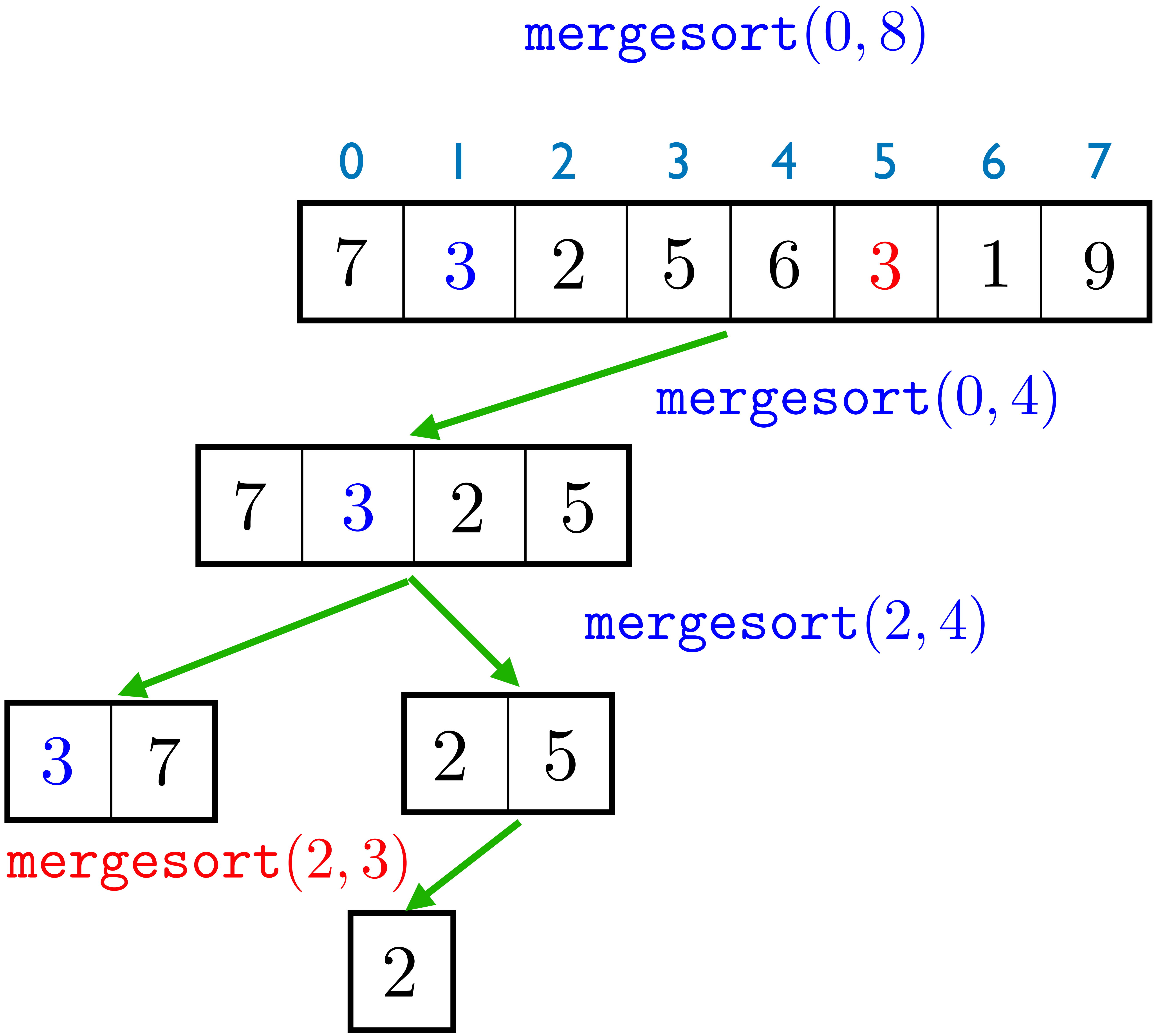

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(0, 2)
3 7


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
3 7

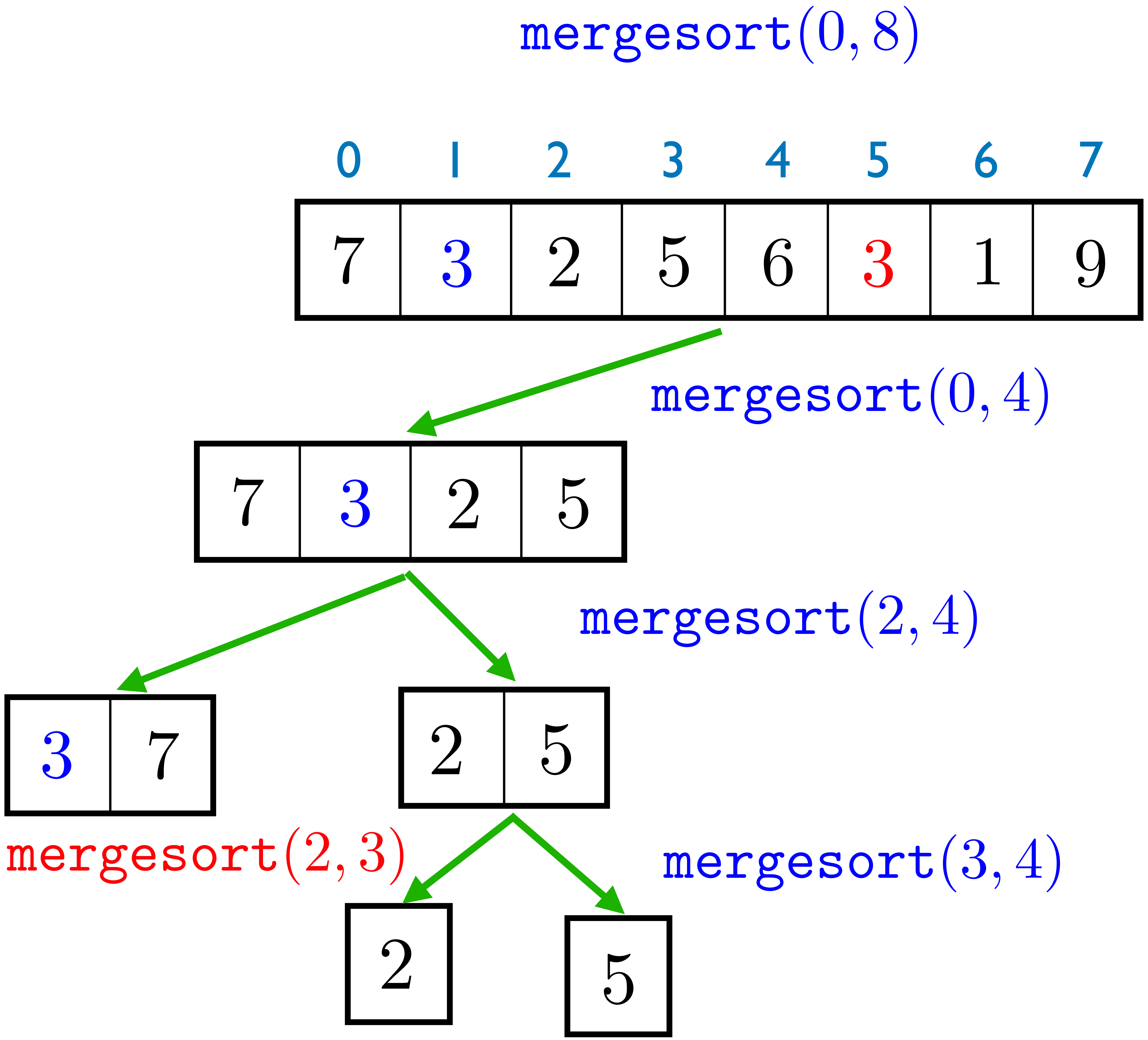


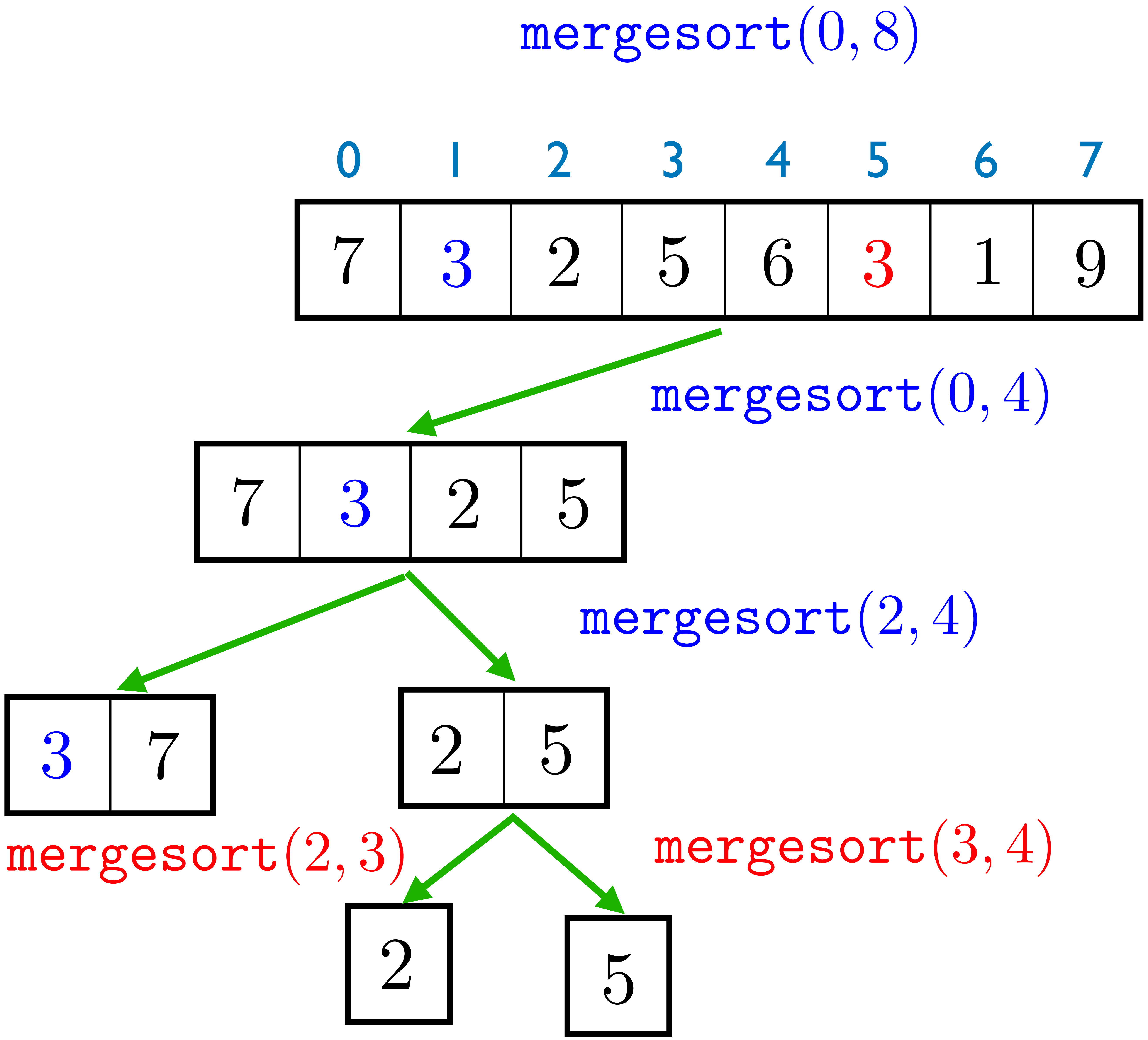
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(2, 4)
3 7
2 5

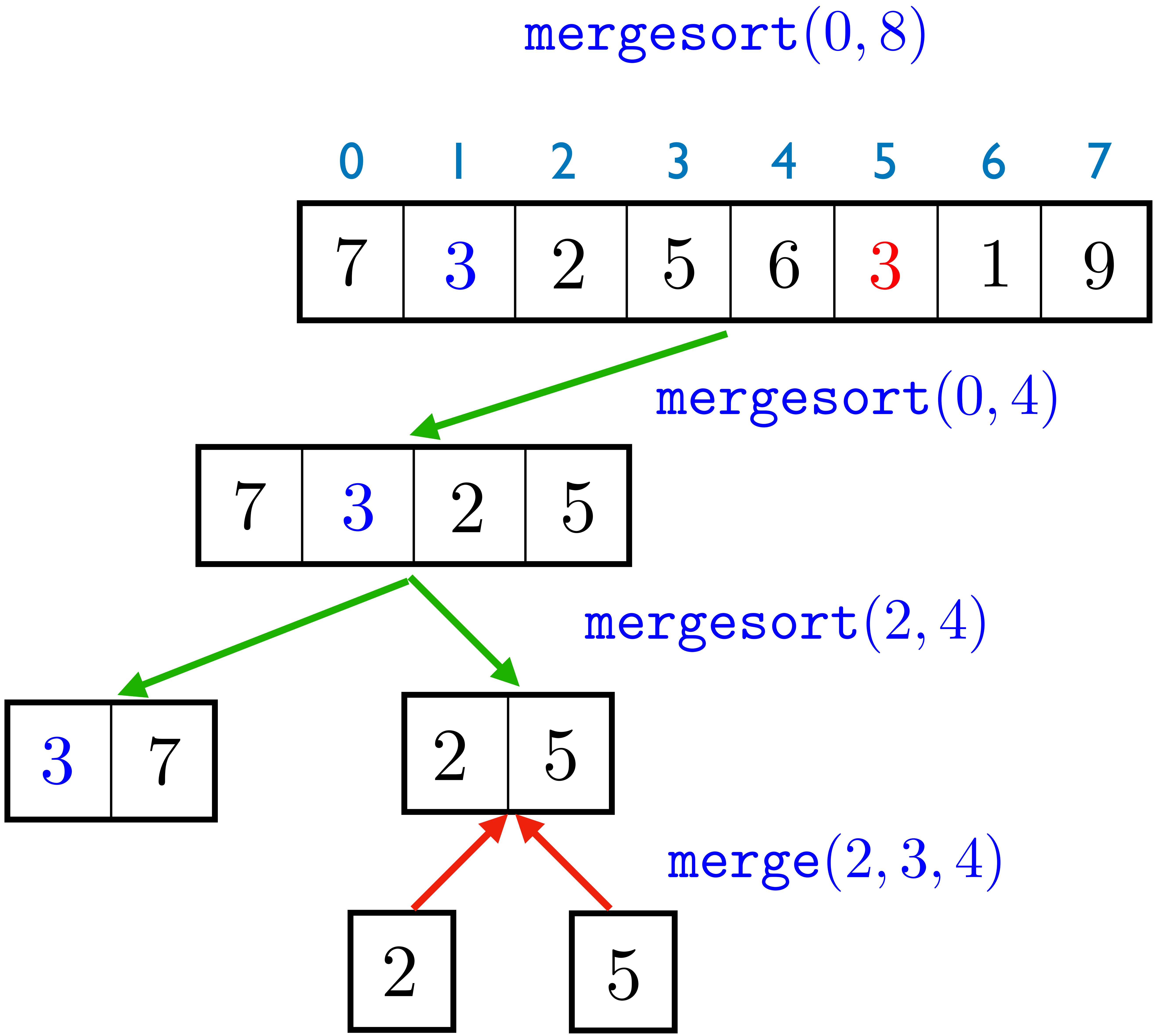

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(2, 4)
3 7
2 5
mergesort(2, 3)
2

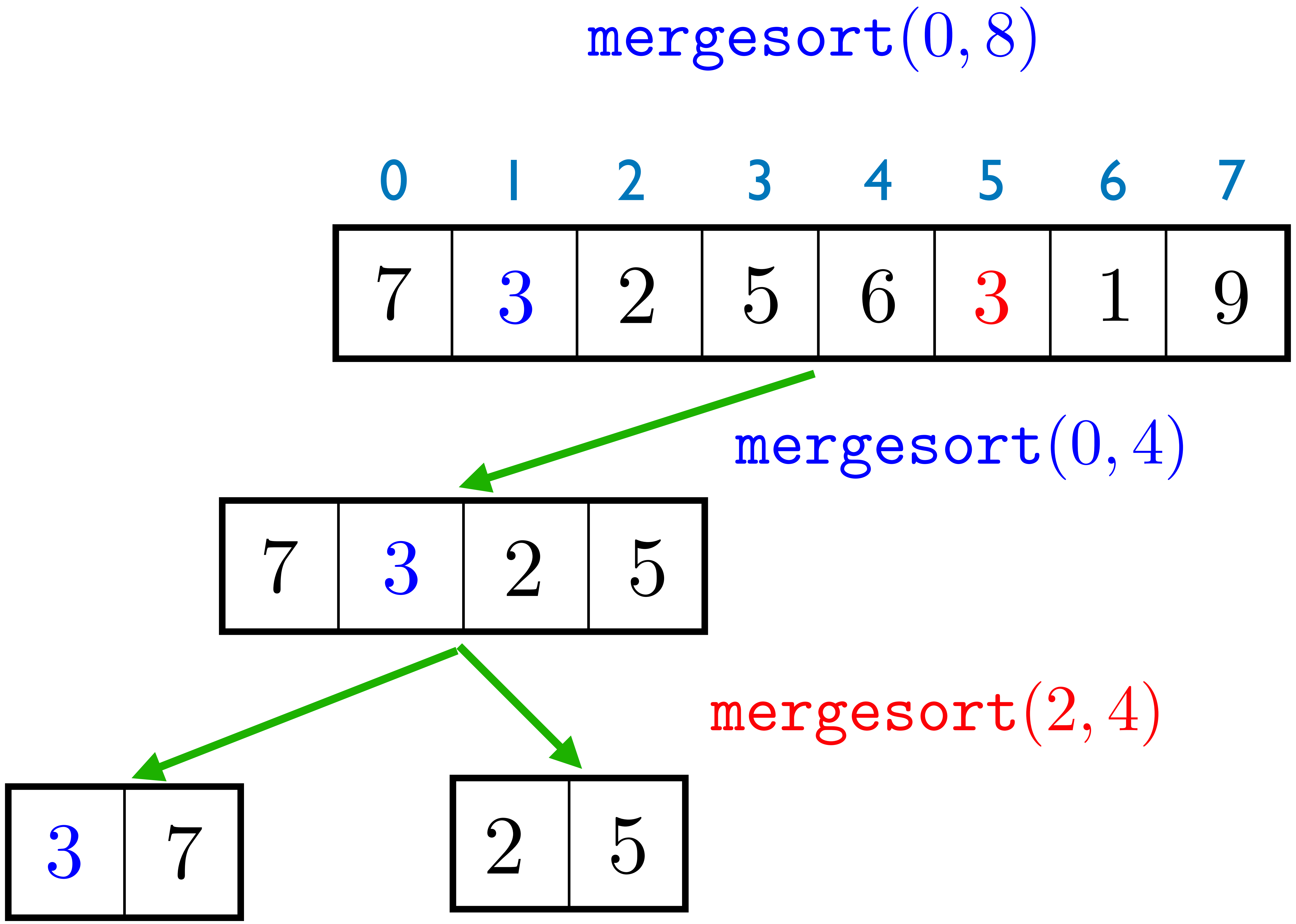


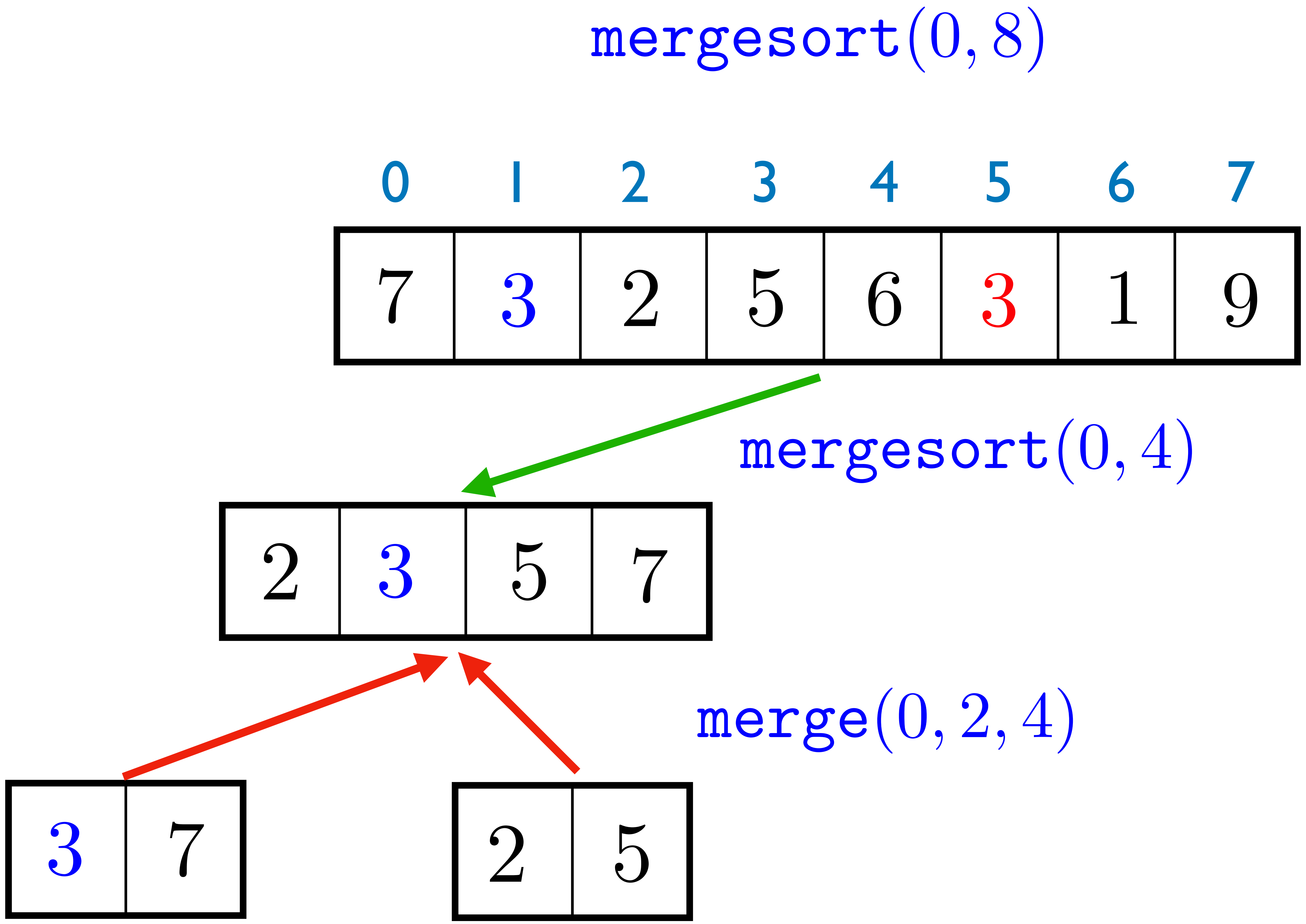
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(2, 4)
3 7
2 5
mergesort(2, 3)
2

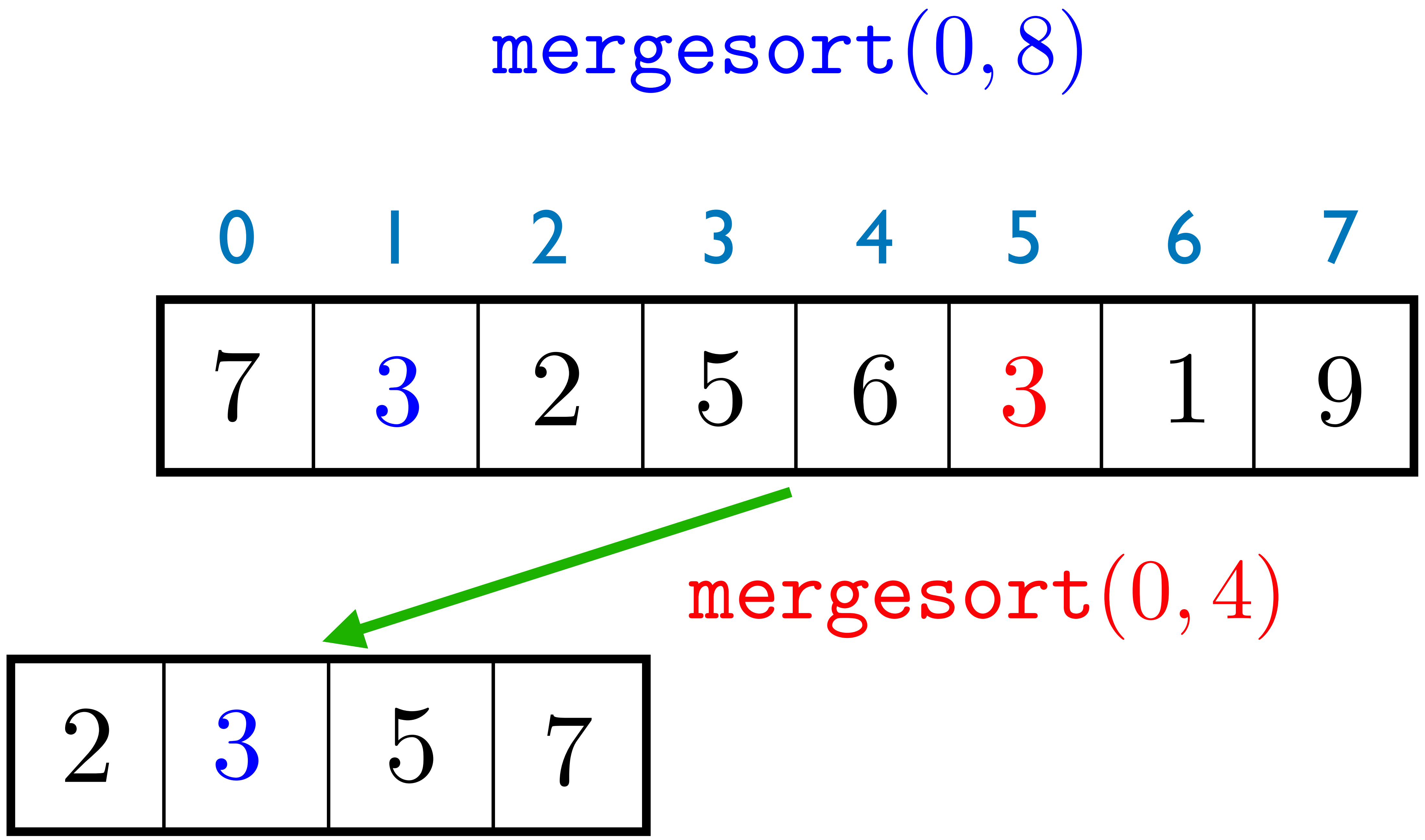

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(2, 4)
3 7
2 5
mergesort(2, 3)
mergesort(3, 4)
2
5


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(2, 4)
3 7
2 5
mergesort(2, 3)
mergesort(3, 4)
2
5

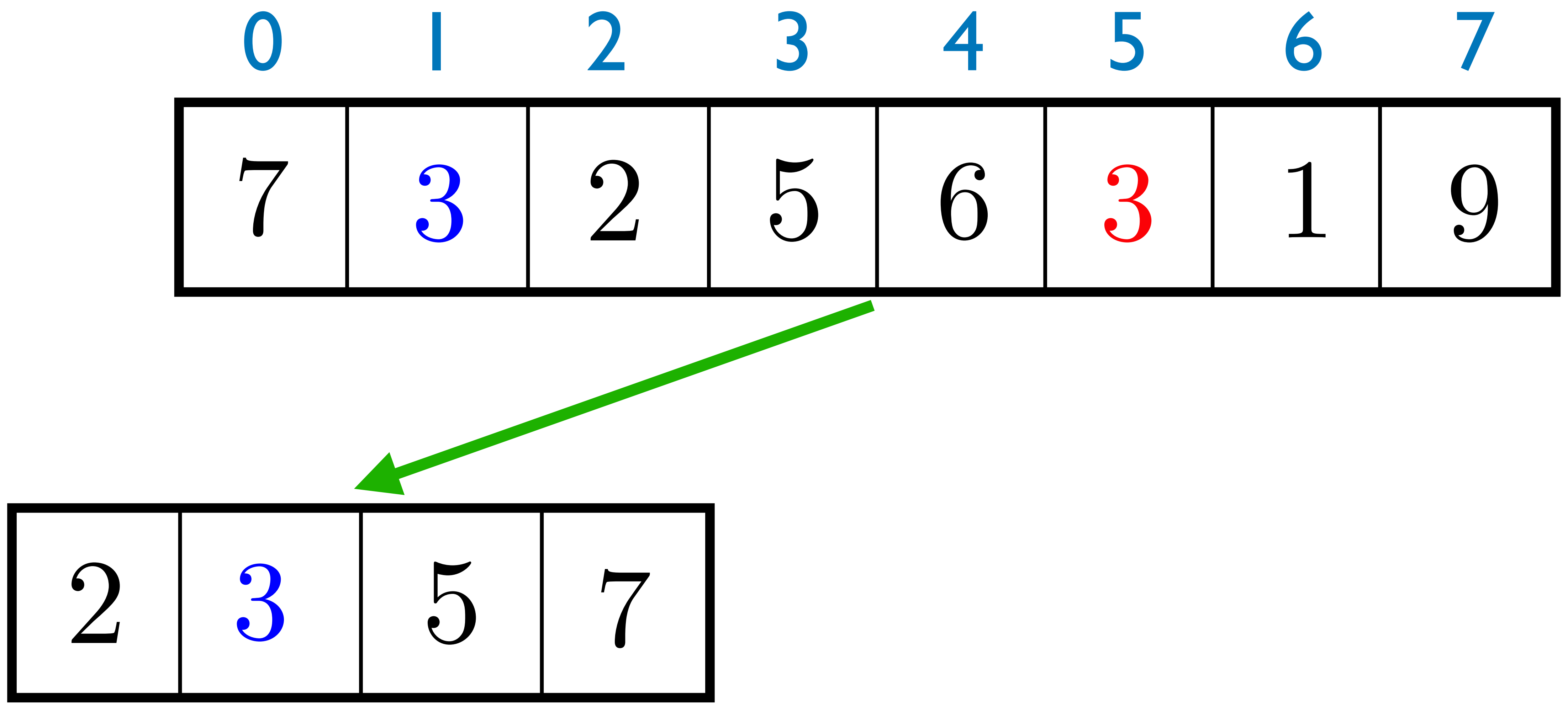

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(2, 4)
3 7
2 5
merge(2, 3, 4)
2
5

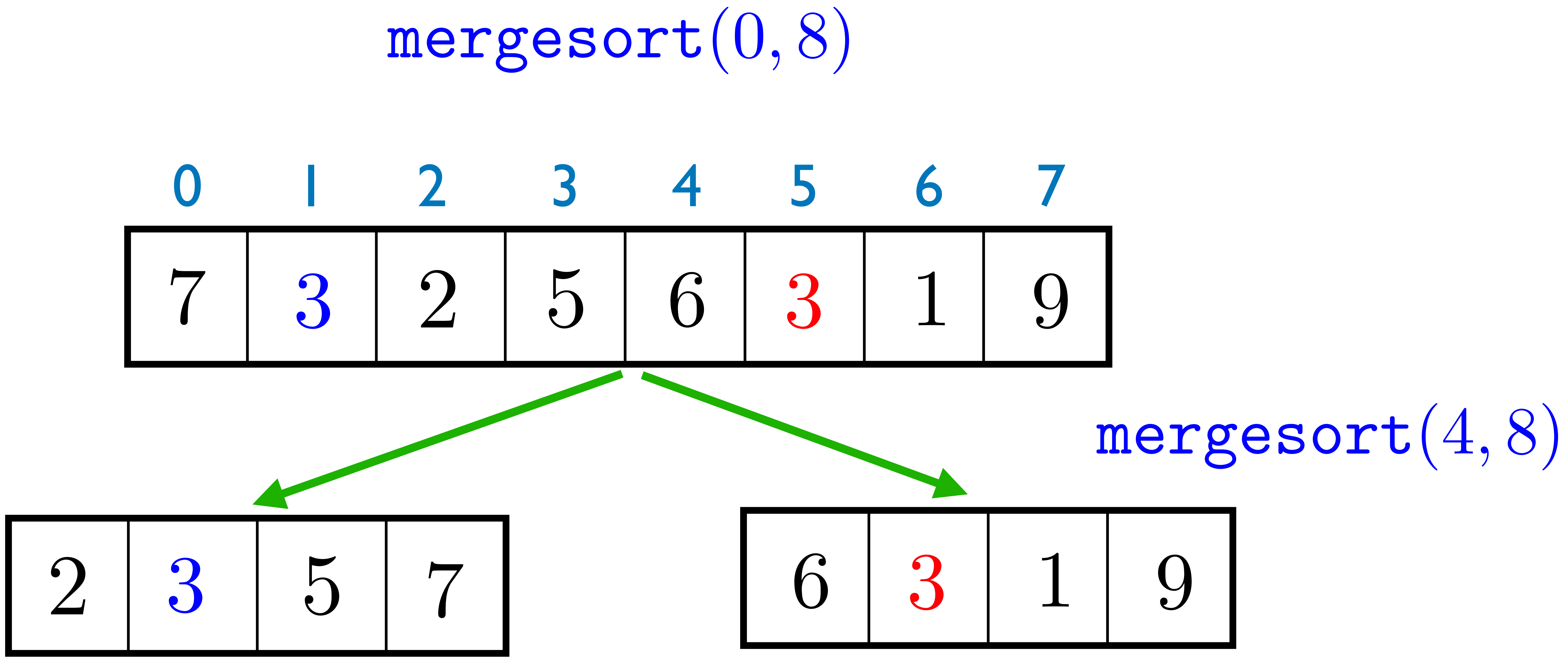

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
7 3 2 5
mergesort(2, 4)
3 7
2 5


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
2 3 5 7
merge(0, 2, 4)
3 7
2 5

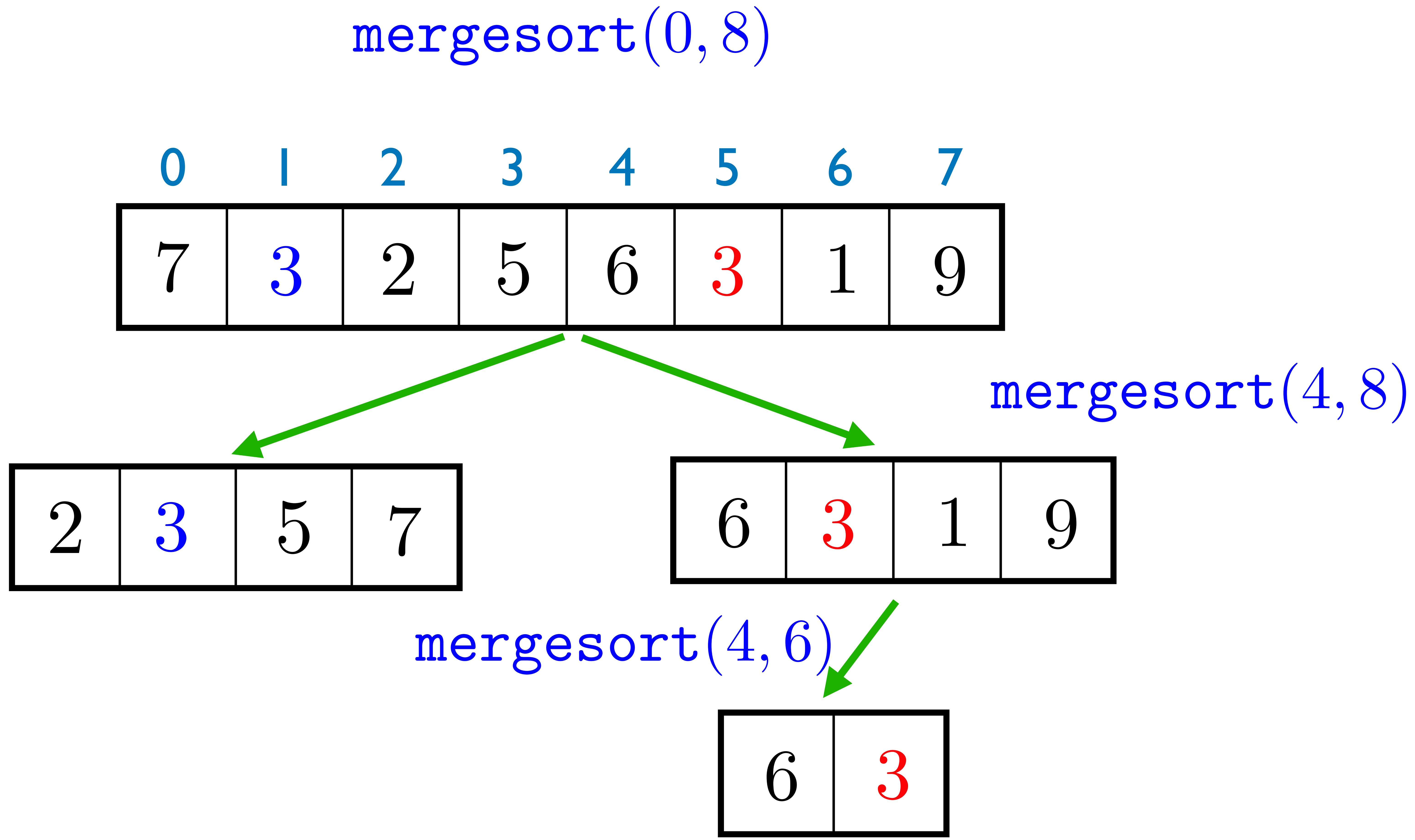

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(0, 4)
2 3 5 7

$\texttt{mergesort}(0, 8)$

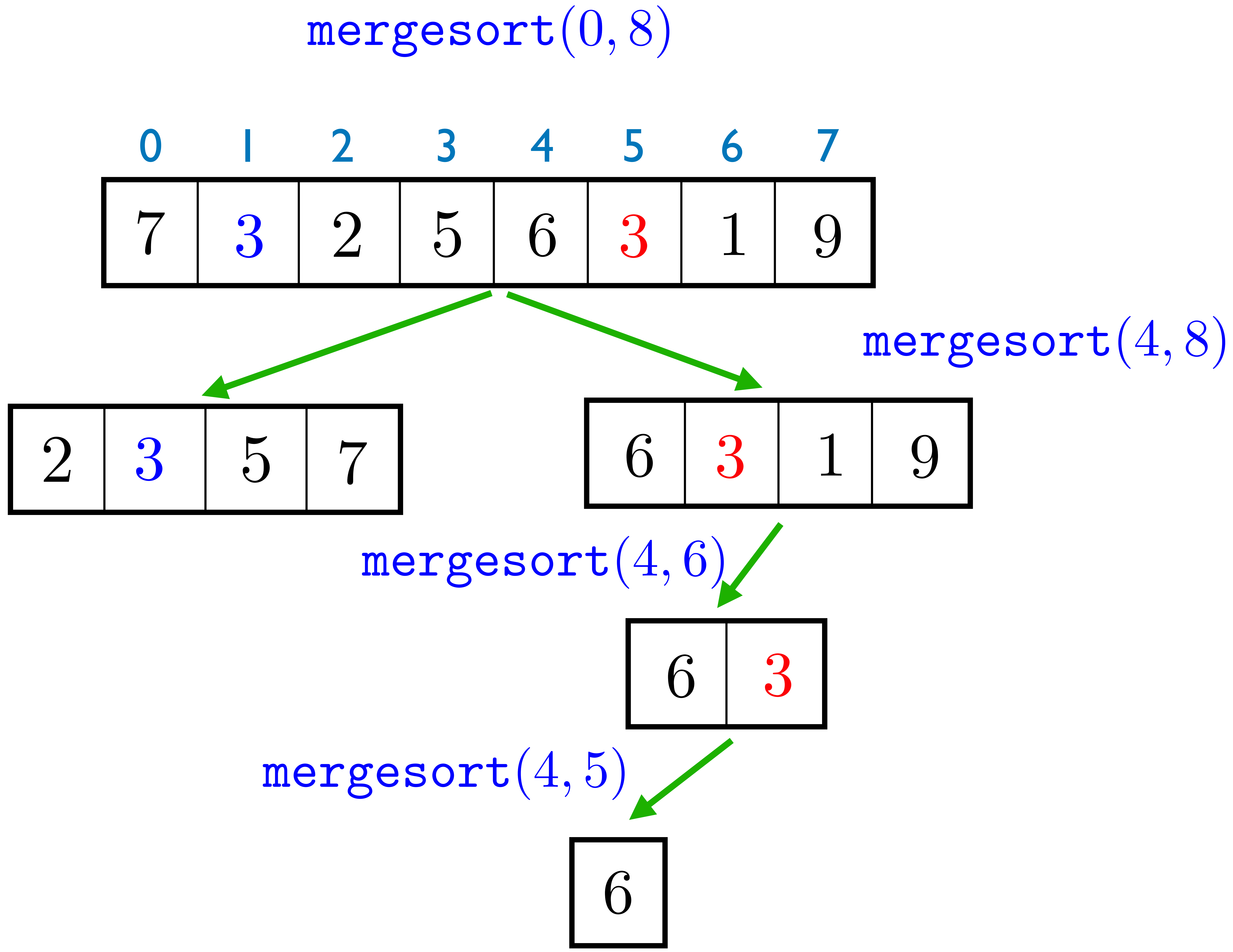




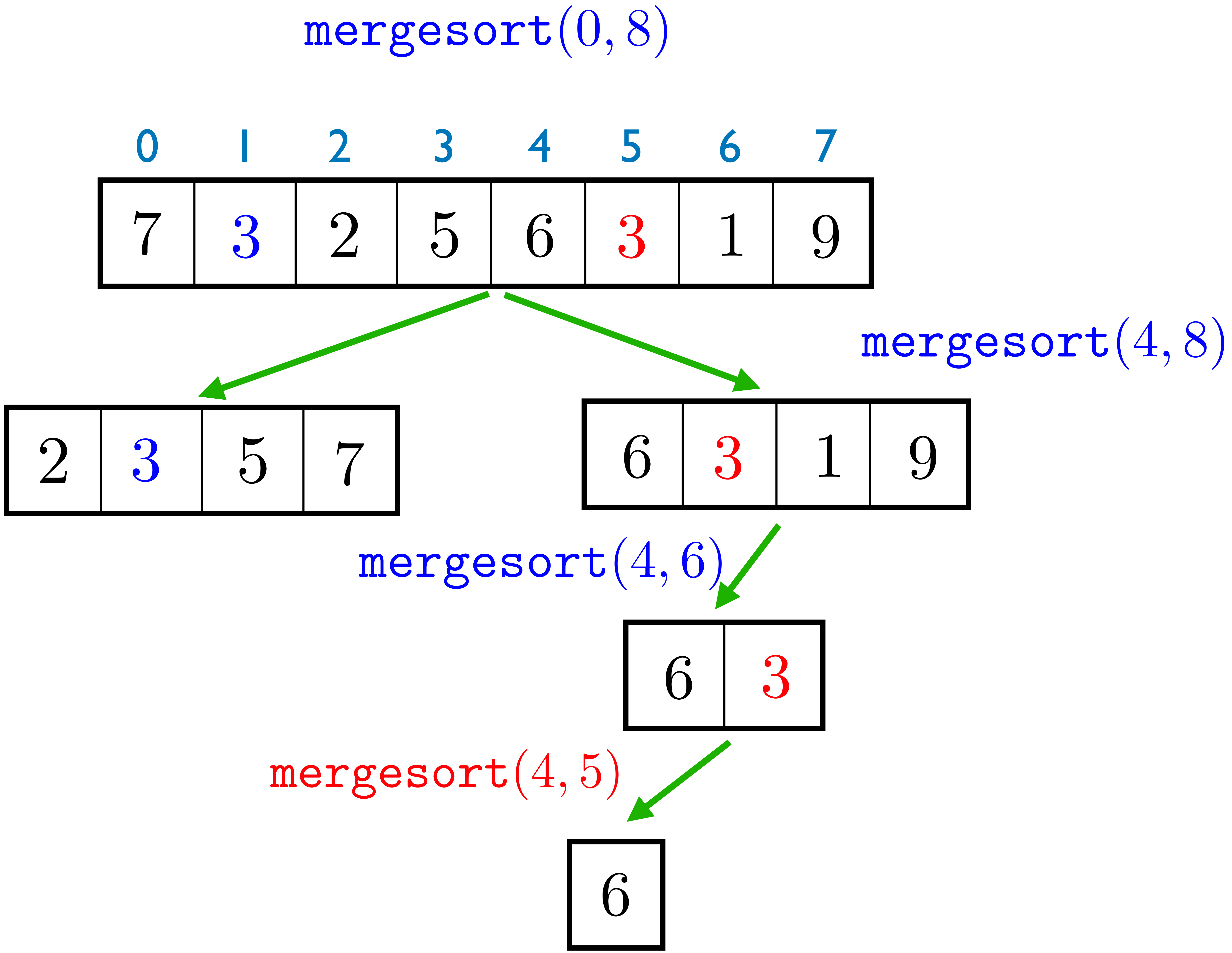
$\texttt{mergesort}(0, 8)$
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
$\texttt{mergesort}(4, 8)$
2 3 5 7
6 3 1 9



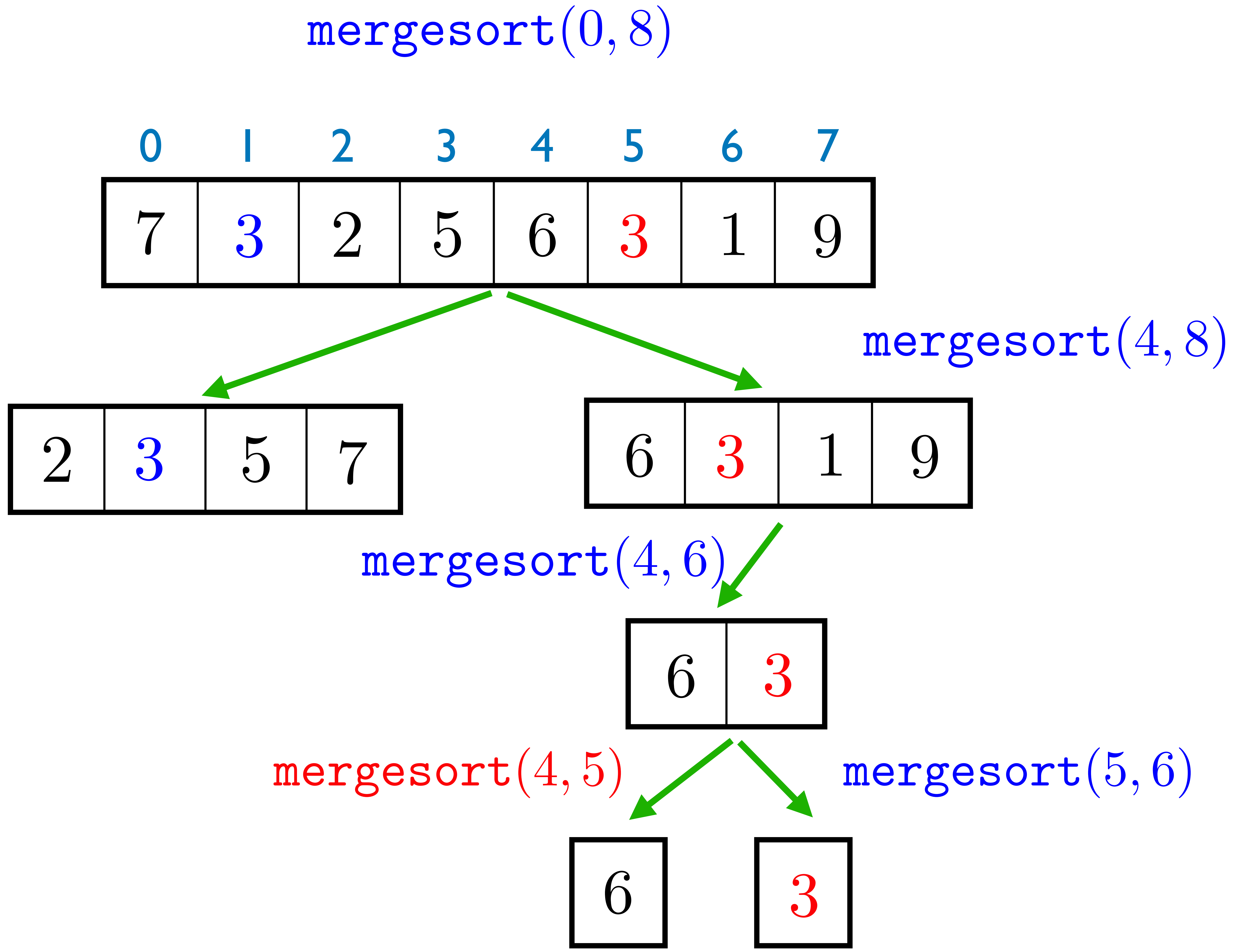
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
2 3 5 7
mergesort(4, 8)
6 3 1 9
mergesort(4, 6)
6 3

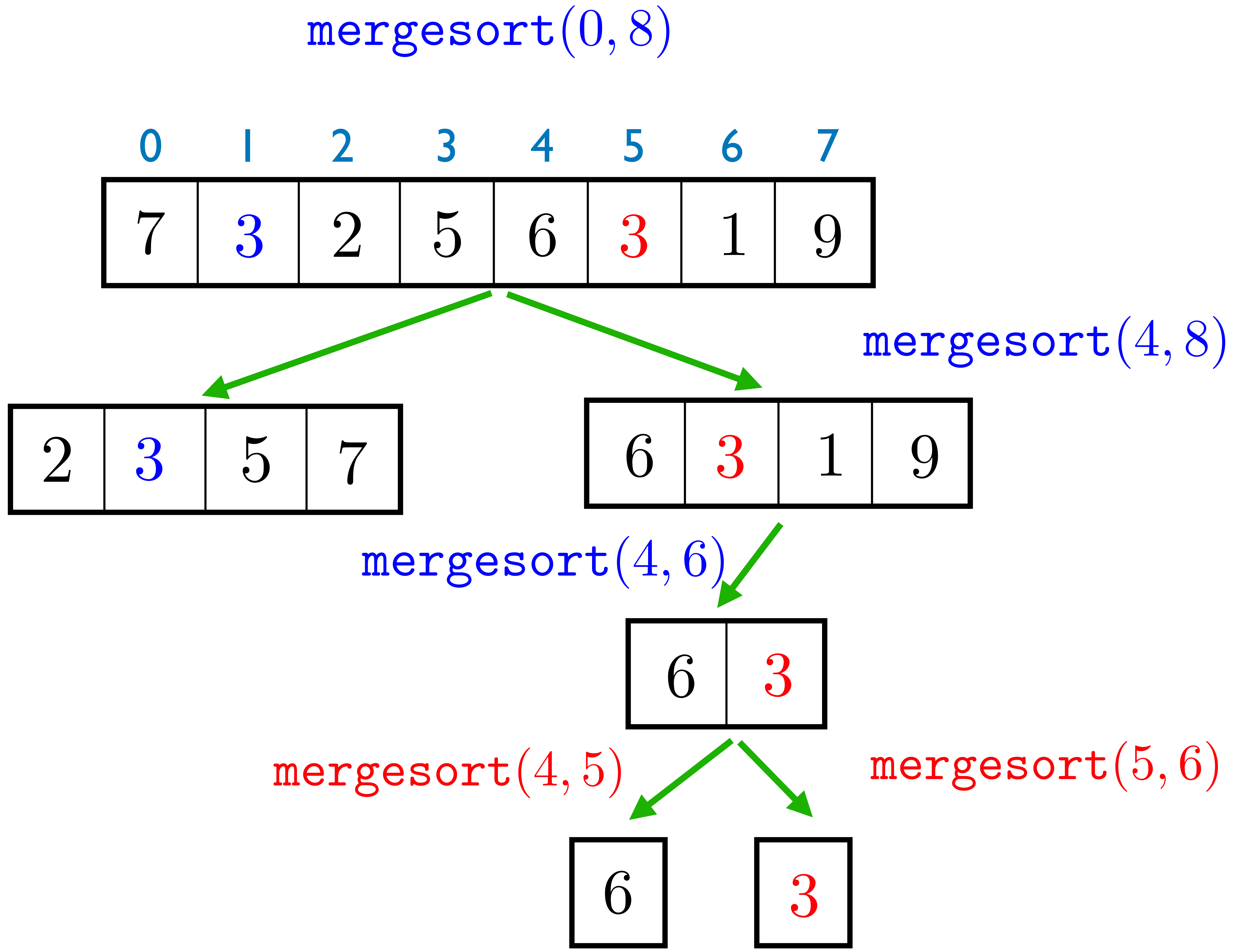


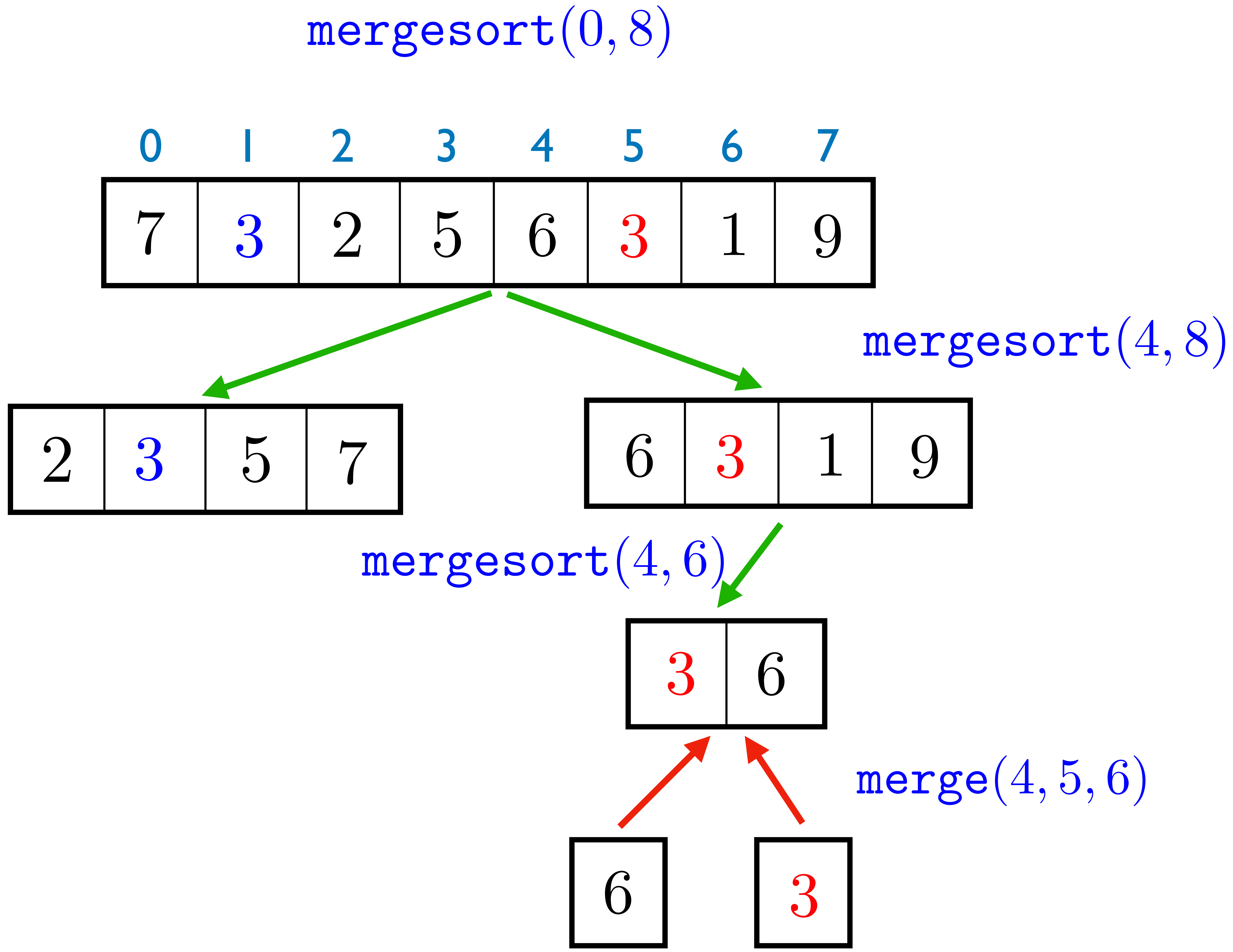
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(4, 6)
6 3
mergesort(4, 5)
6



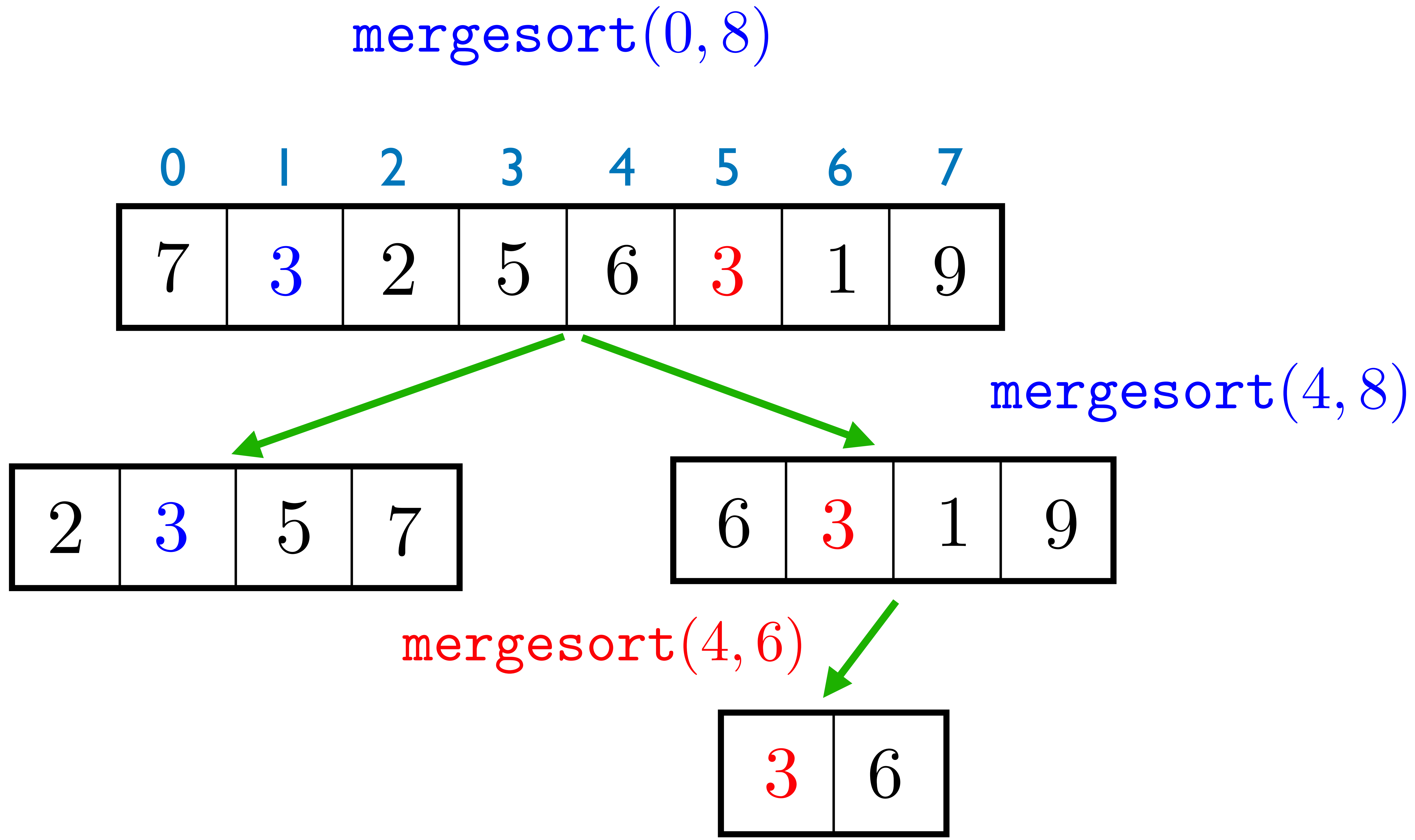
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(4, 6)
6 3
mergesort(4, 5)
6


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(4, 6)
6 3
mergesort(4, 5)
mergesort(5, 6)
6
3

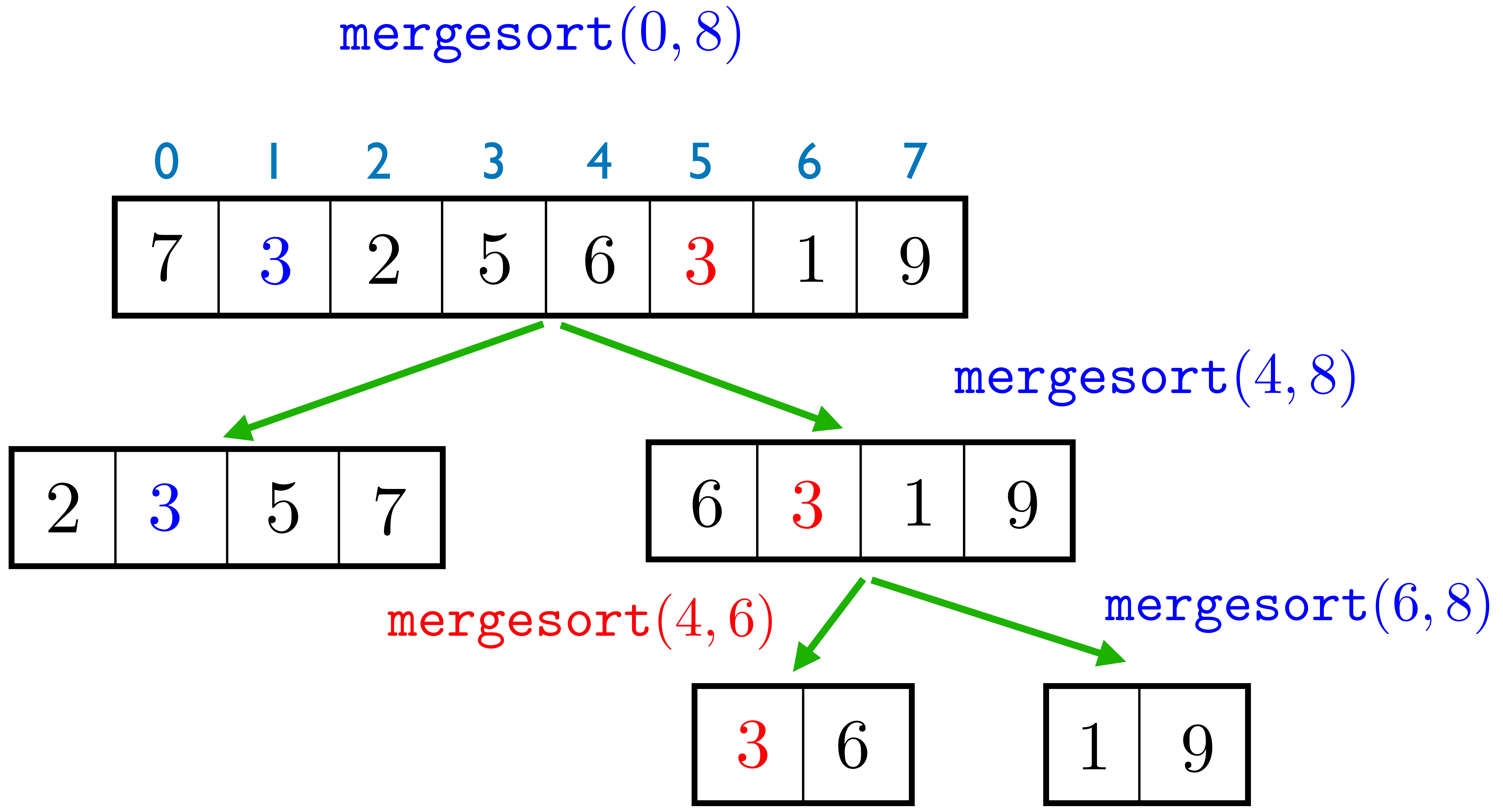

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(4, 6)
6 3
mergesort(4, 5)
mergesort(5, 6)
6
3

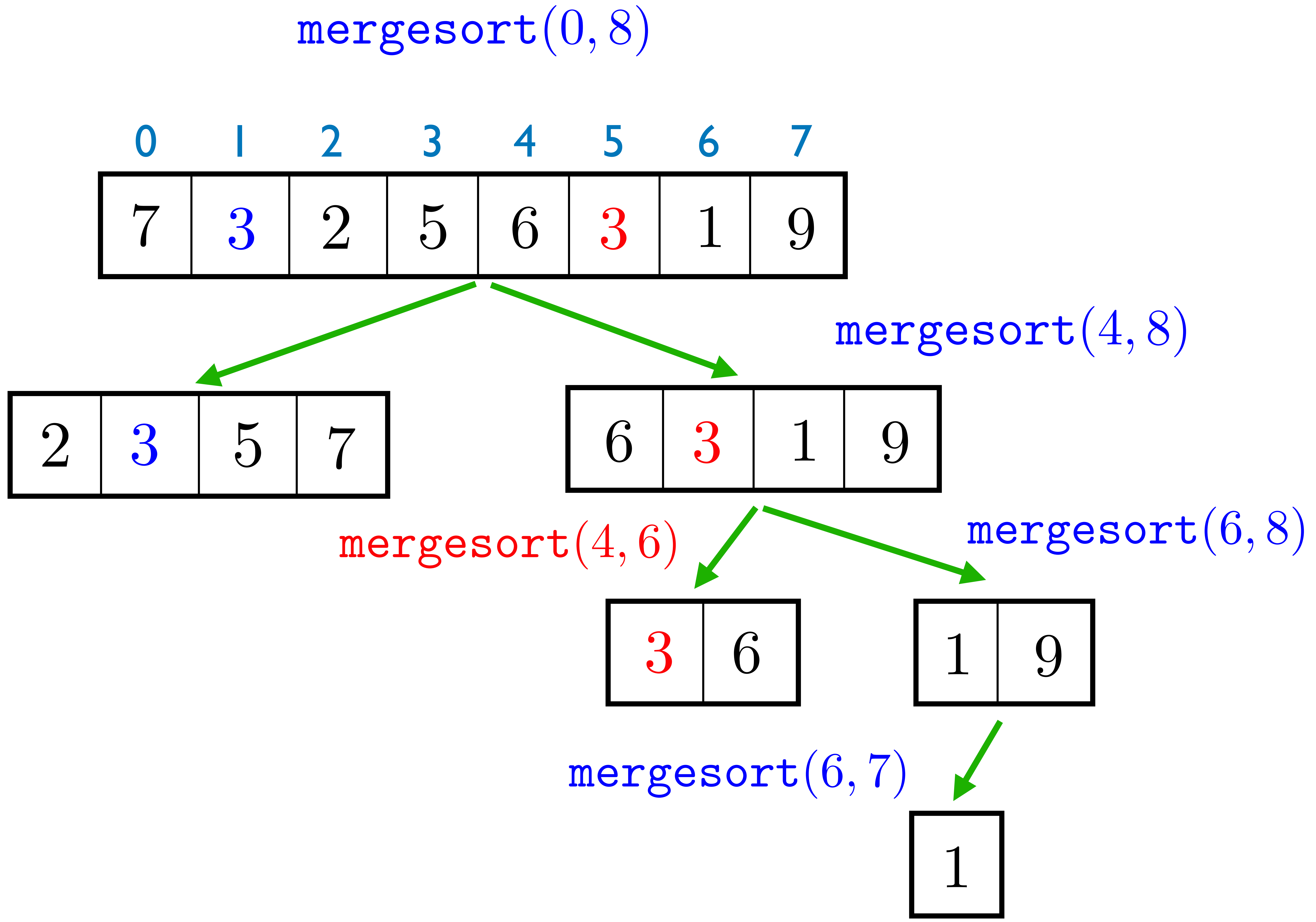


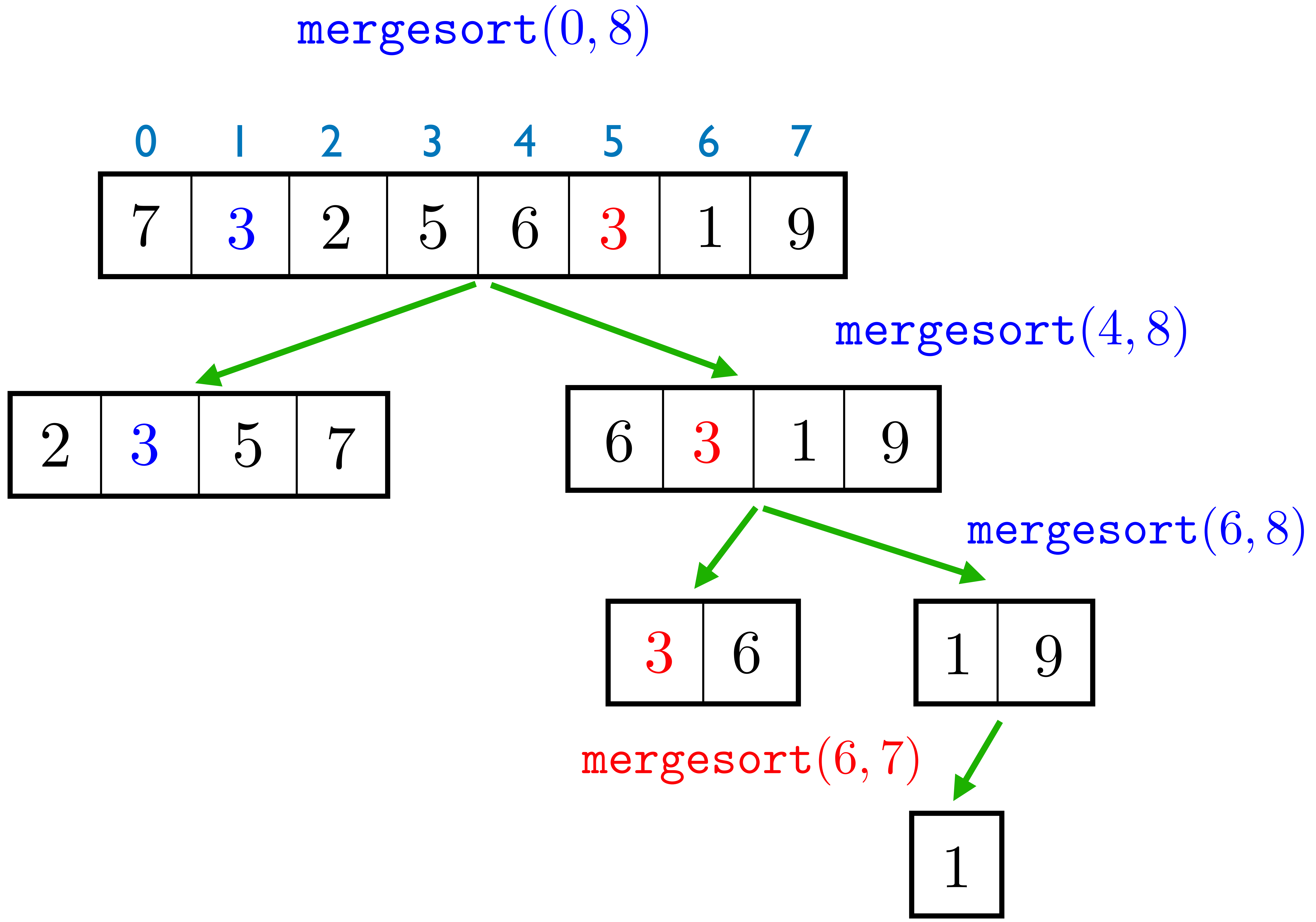
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(4, 6)
3 6
merge(4, 5, 6)
6
3

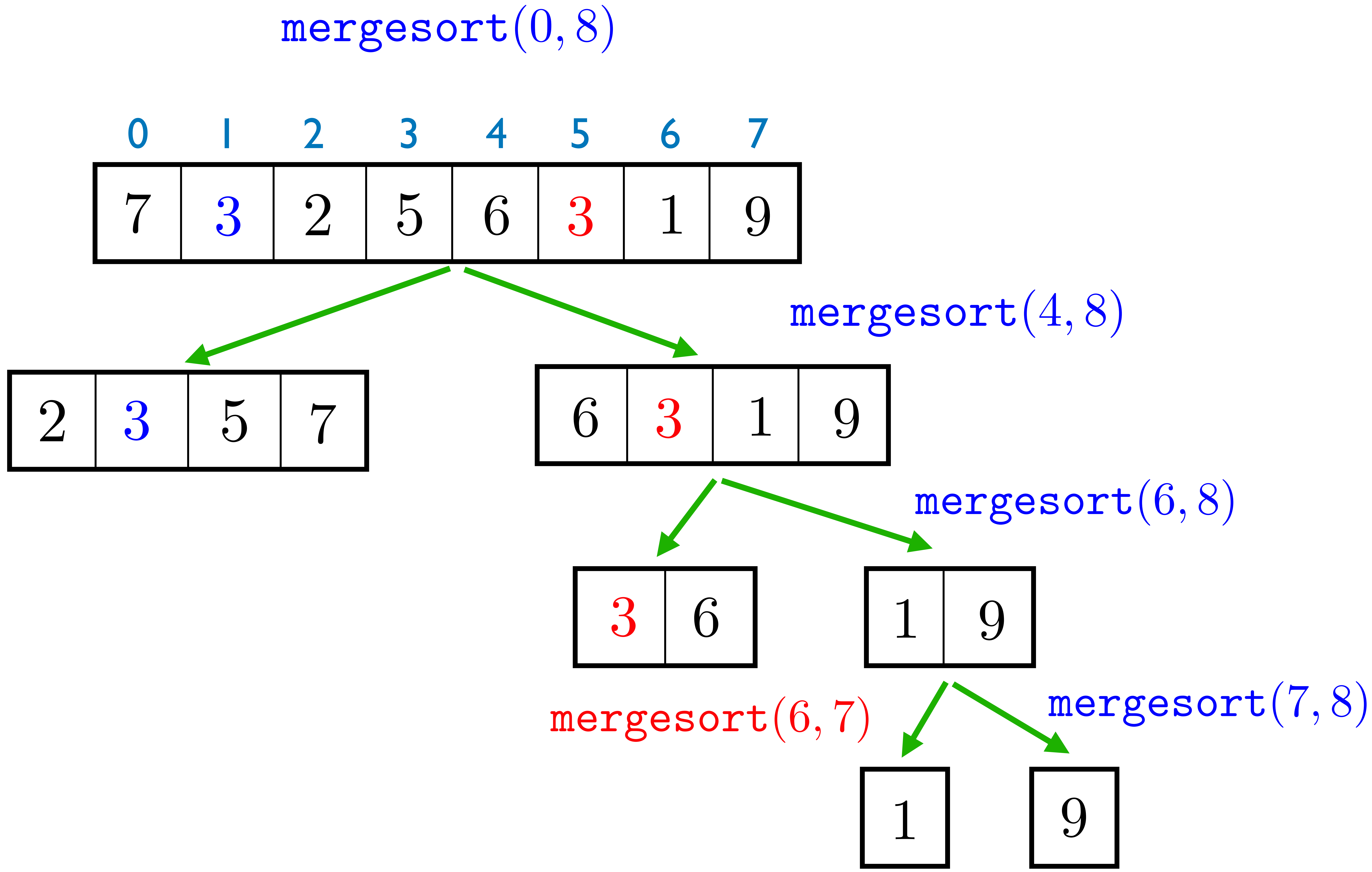


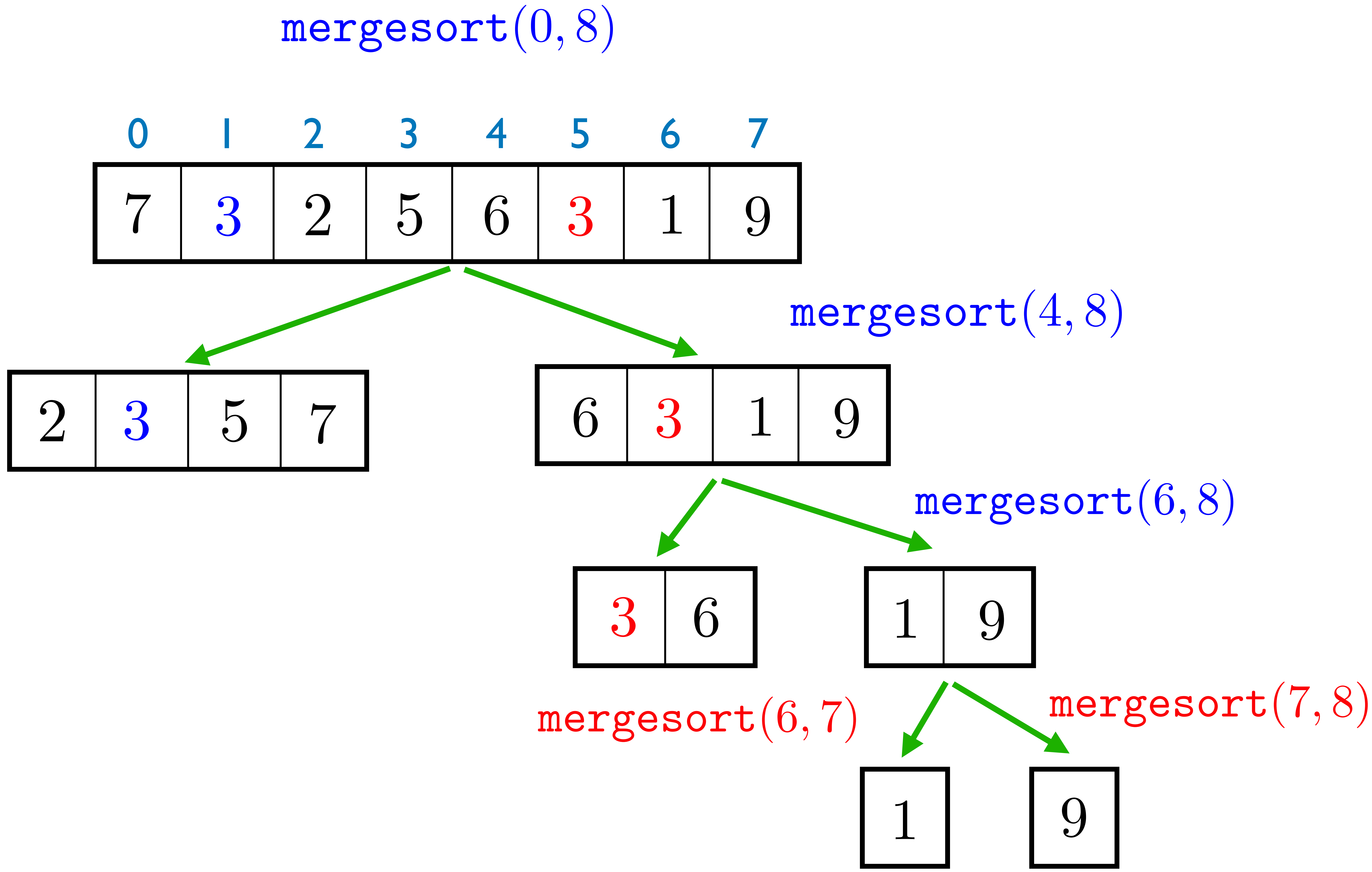
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(4, 6)
3 6

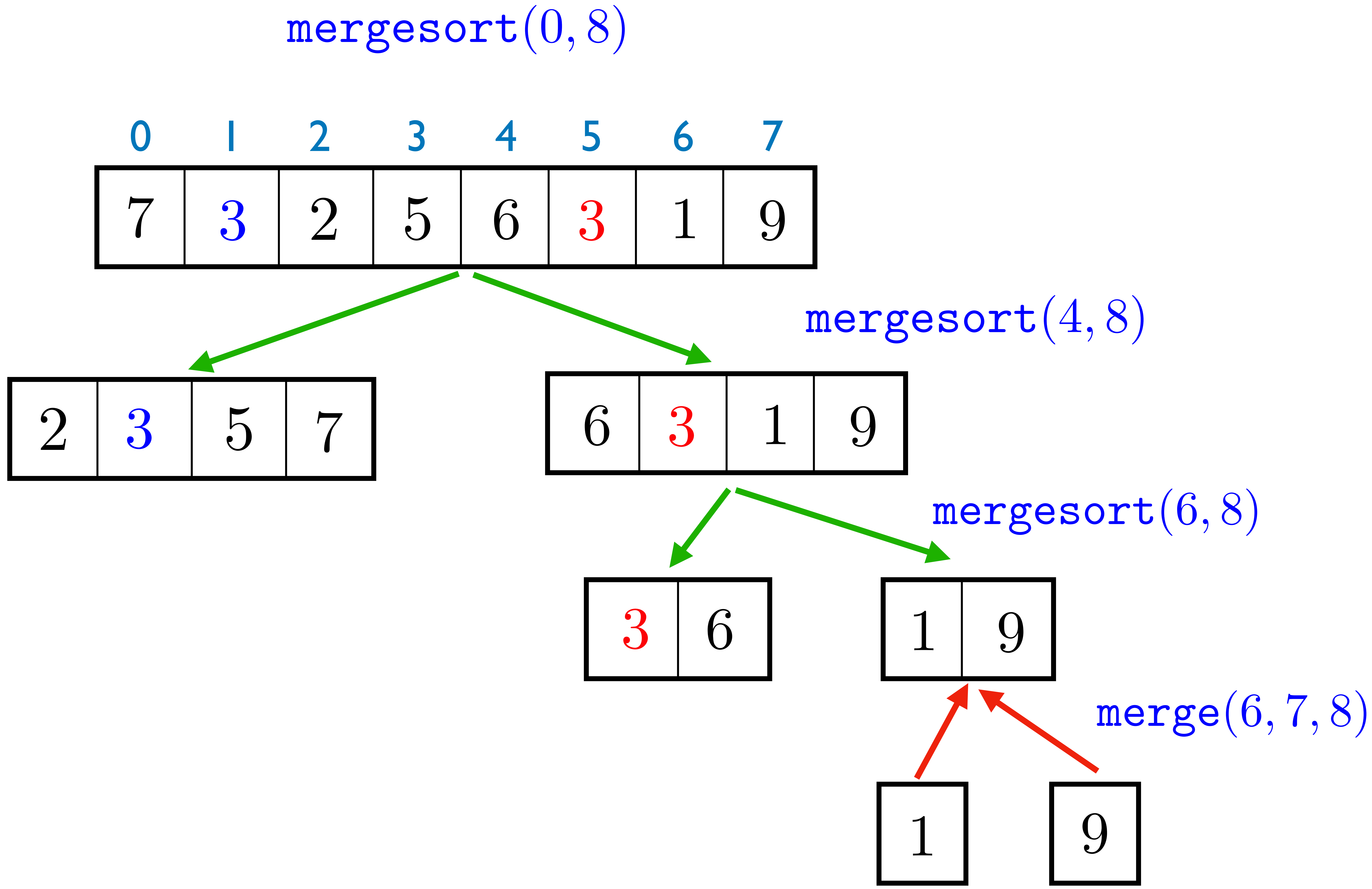

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
2 3 5 7
6 3 1 9
mergesort(4, 8)
mergesort(4, 6)
mergesort(6, 8)
3 6
1 9

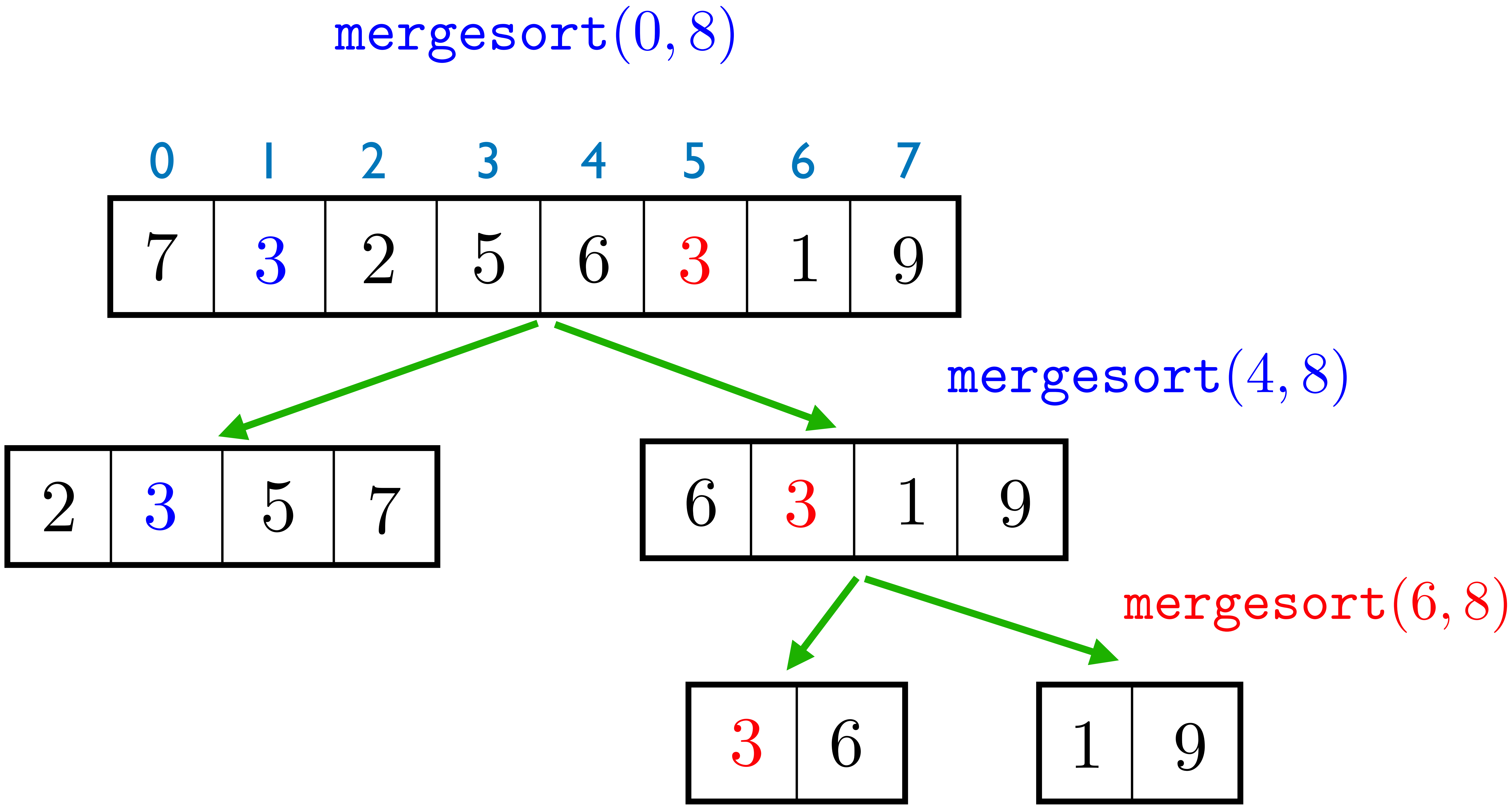

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(4, 6)
mergesort(6, 8)
3 6
1 9
mergesort(6, 7)
1

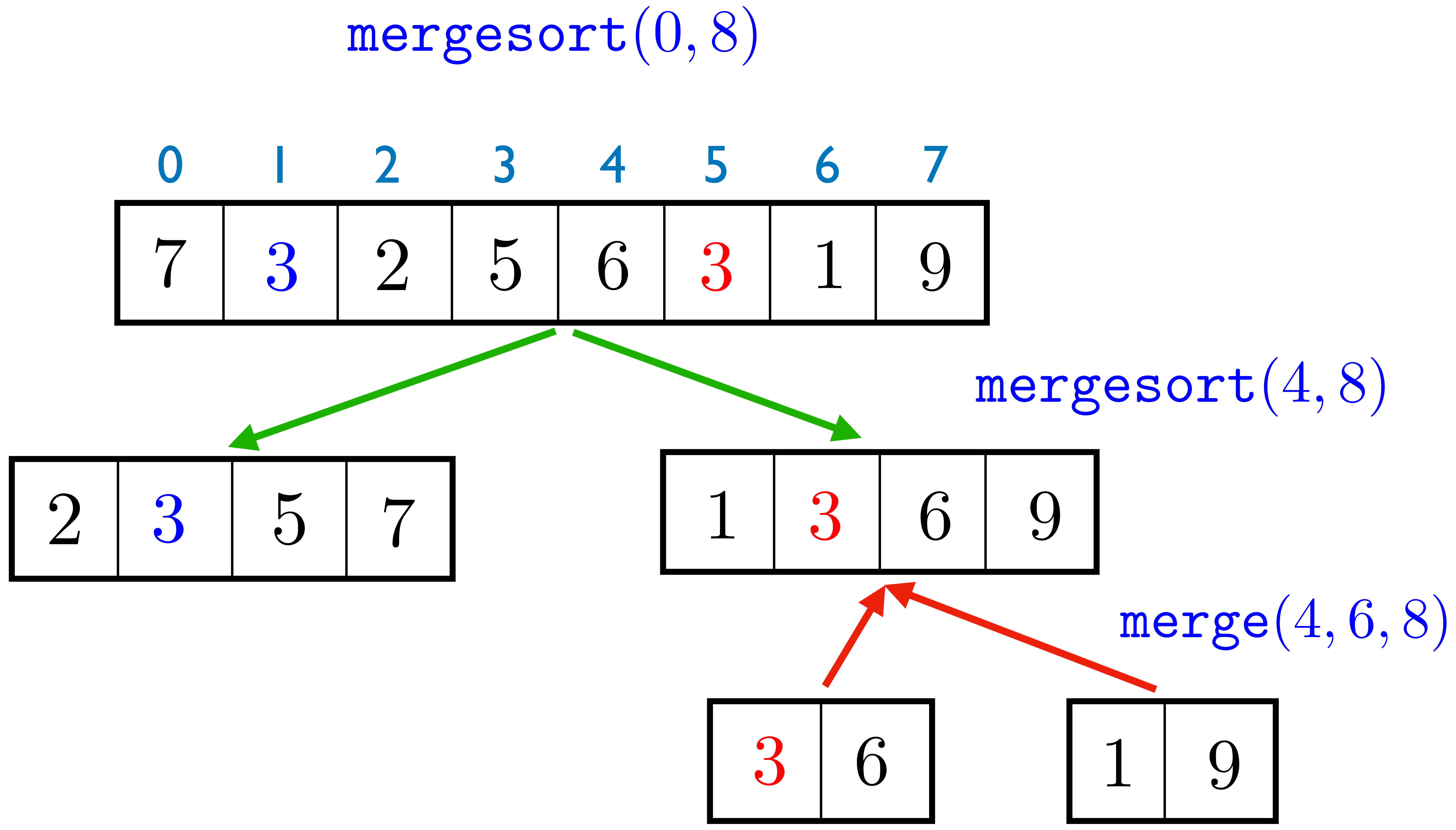

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(6, 8)
3 6
1 9
mergesort(6, 7)
1


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(6, 8)
3 6
1 9
mergesort(6, 7)
mergesort(7, 8)
1
9

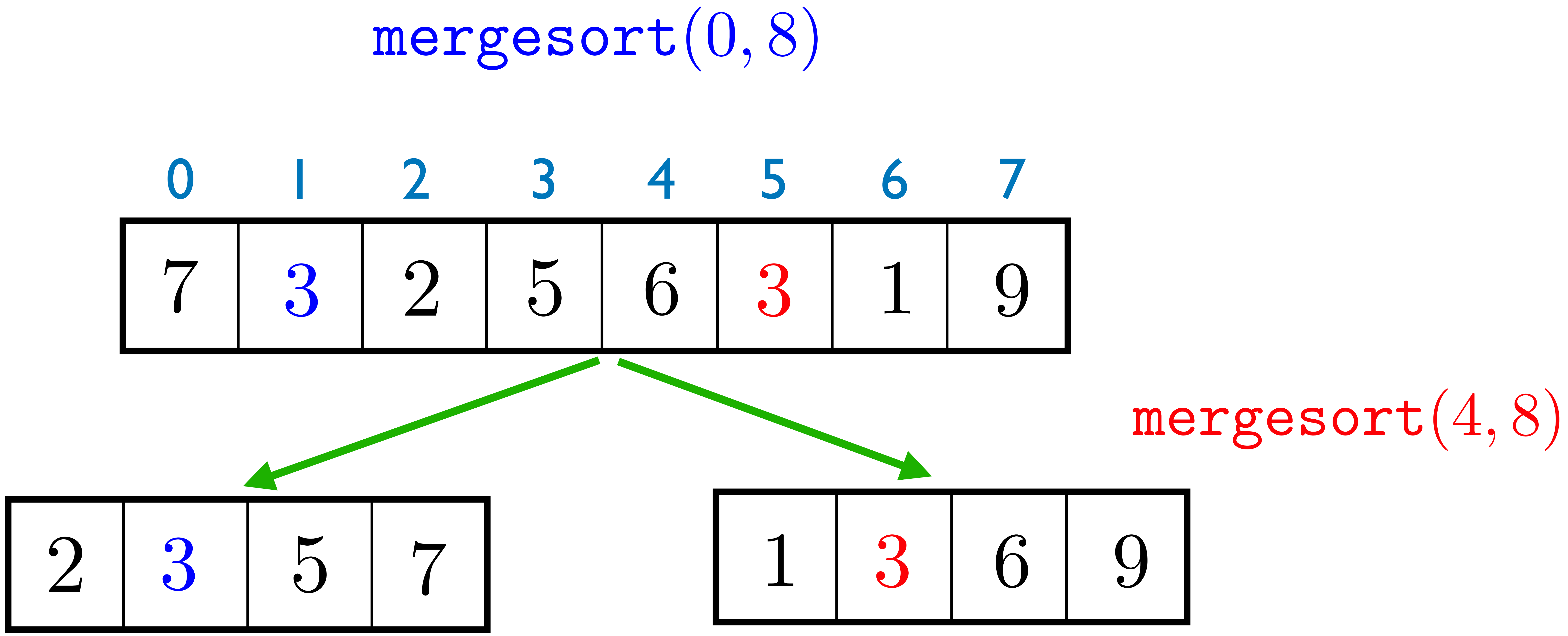

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
2 3 5 7
mergesort(4, 8)
6 3 1 9
mergesort(6, 8)
3 6
1 9
mergesort(6, 7)
mergesort(7, 8)
1
9

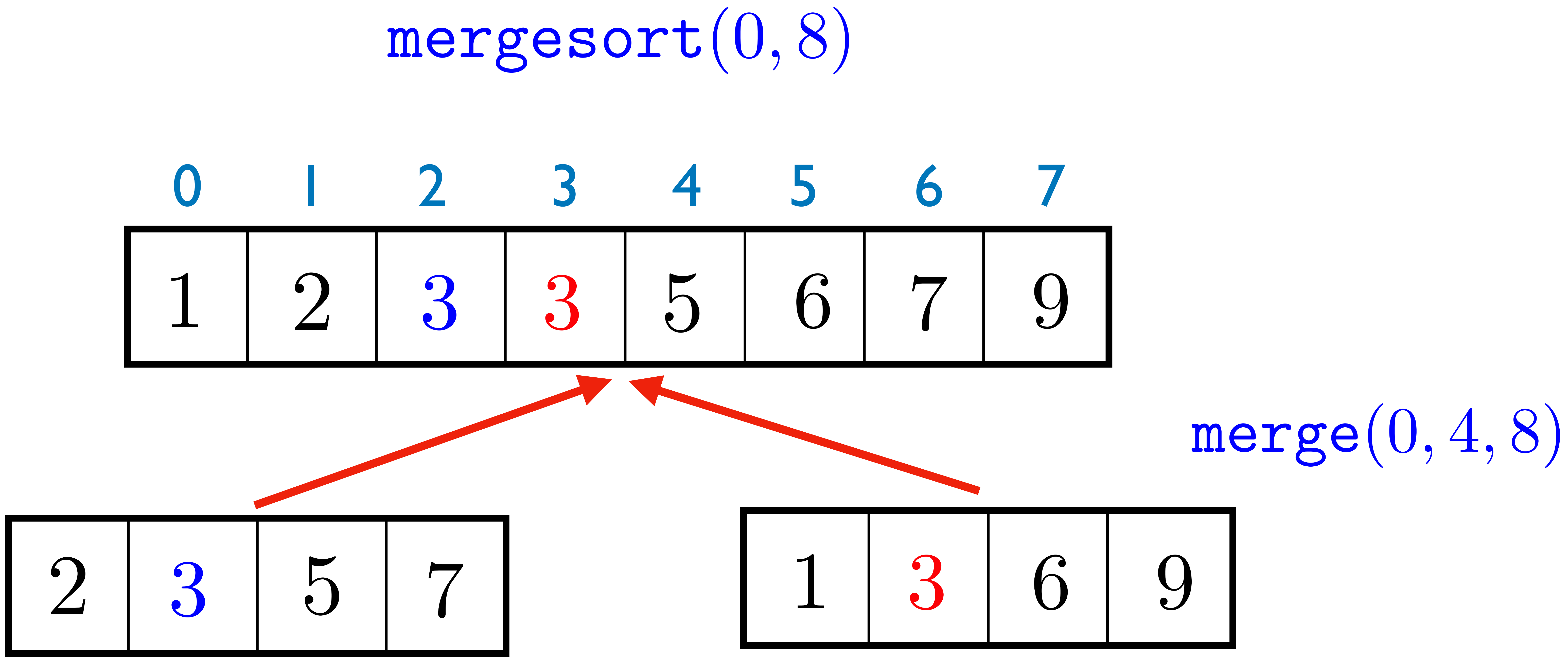

mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(6, 8)
3 6
1 9
merge(6, 7, 8)
1
9


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
6 3 1 9
mergesort(6, 8)
3 6
1 9



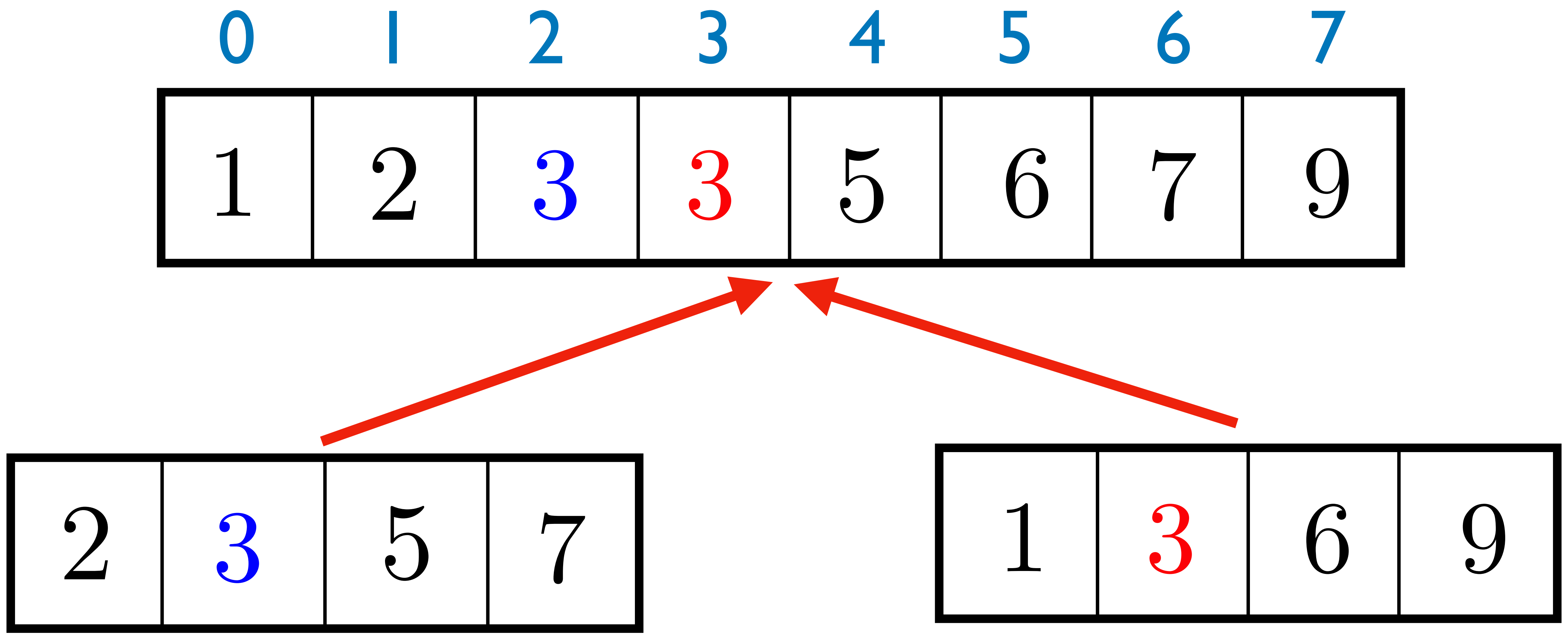
mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
1 3 6 9
merge(4, 6, 8)
3 6
1 9


mergesort(0, 8)
0 1 2 3 4 5 6 7
7 3 2 5 6 3 1 9
mergesort(4, 8)
2 3 5 7
1 3 6 9



For mergesort to be stable the merge algorithm needs to put equal values from the left subproblem before those from the right subproblem.

$\texttt{mergesort}(0, 8)$



Now the algorithm finishes, and the vector is sorted.

Quicksort

Quicksort

Quicksort is one of the most widely used sorting algorithms in practice.

Its worst-case running time is $\Theta(n^2)$.

The average-case running time, however, is $O(n \log n)$.

Quick sort is comparison based and in place, but not stable.

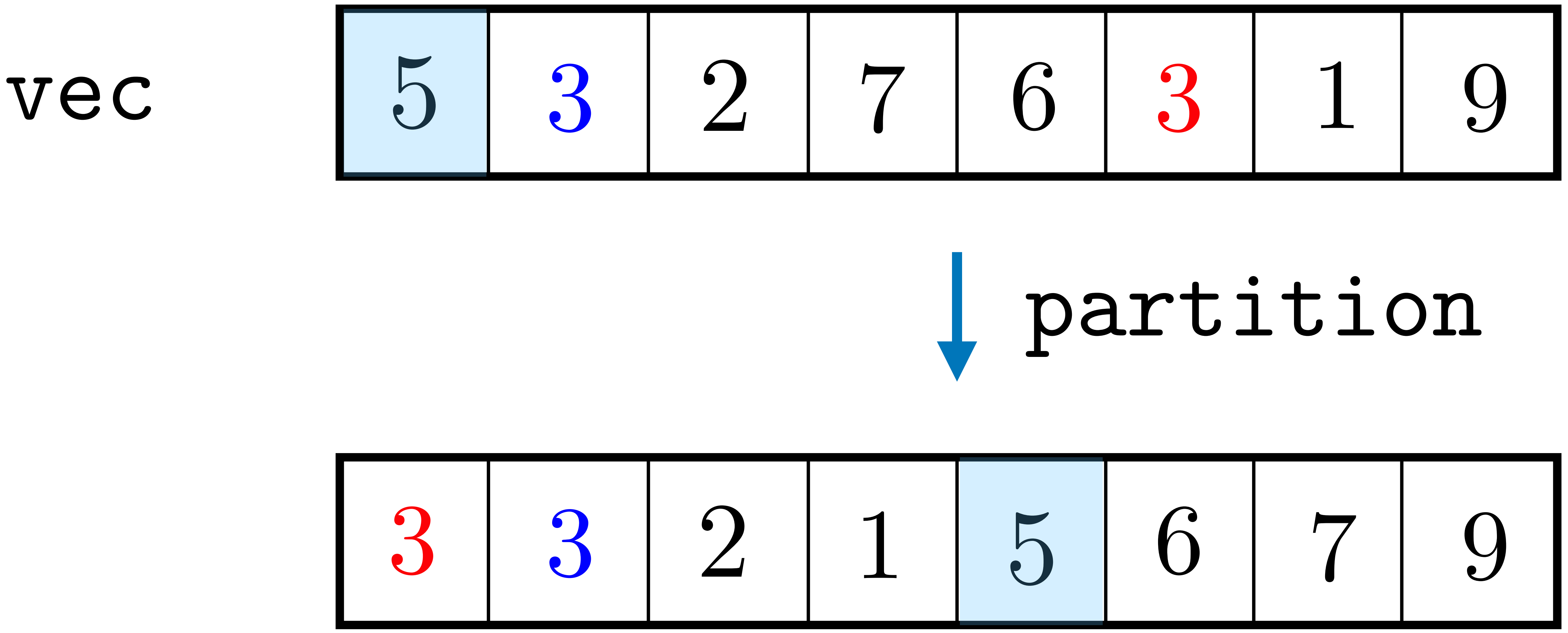
Like mergesort, quicksort is a divide and conquer algorithm.

Quicksort

	0	1	2	3	4	5	6	7
vec	5	3	2	7	6	3	1	9

Step 1: Choose a pivot. Let's take $\texttt{vec}[0]$.

Partition

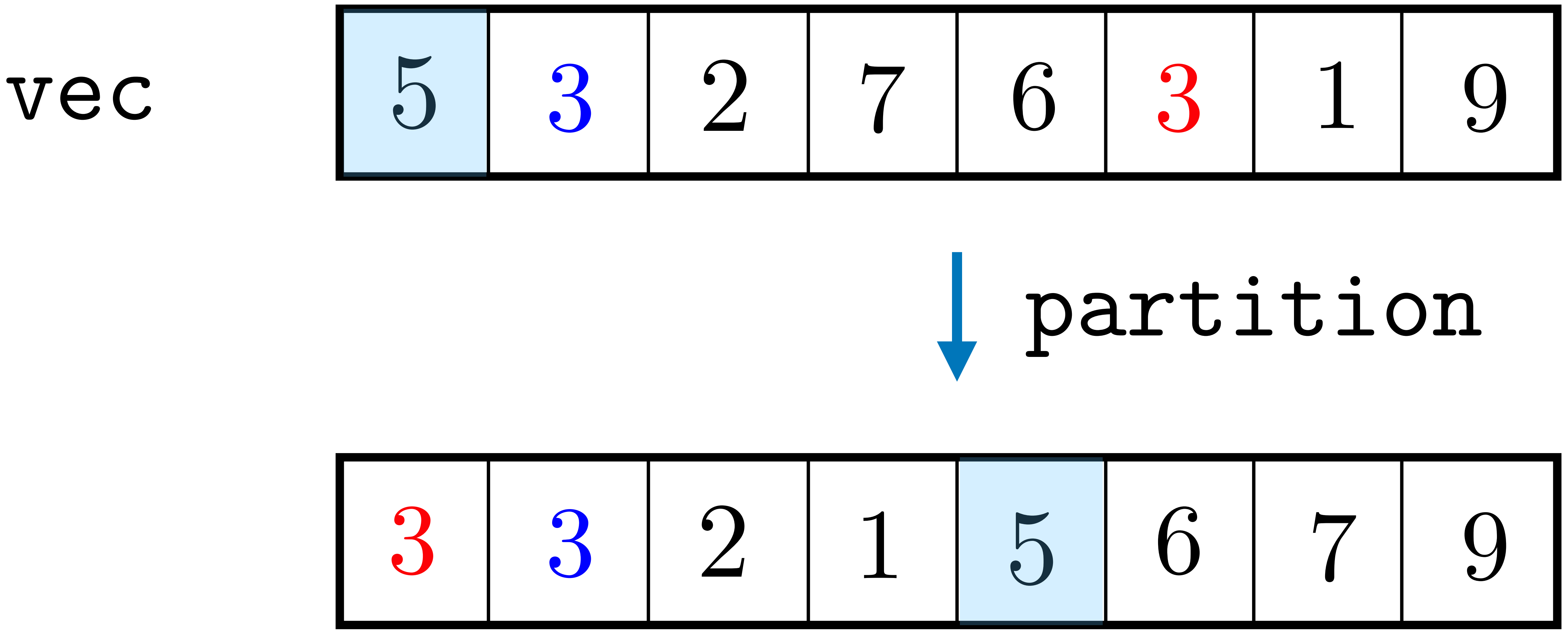


Step 2: Partition—put the pivot in a position such that

everything to the left is $\leq$ the pivot

everything to the right is $\geq$ the pivot

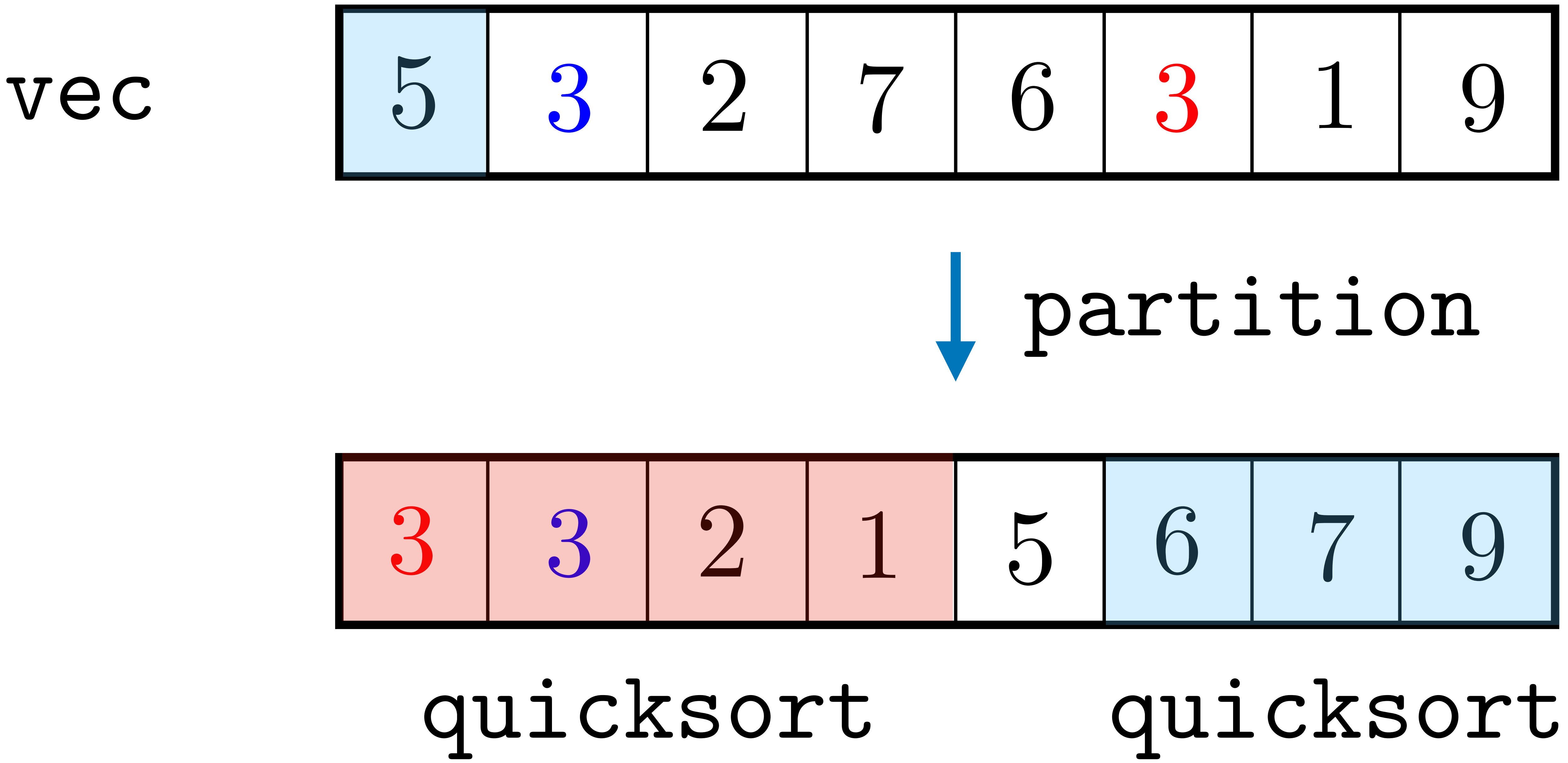
Partition



Step 2: Partition

The pivot is now in a valid final position for a sorted array.

Recurse



Step 3: Recursively use quicksort to sort the portion to the left of the pivot and to the right of the pivot.

Divide And Conquer

Divide: Two subproblems

Sort the elements to the left of the pivot element.

Sort the elements to the right of the pivot element.

Create/Complete/Combine:

Create: The main work in quicksort is to create the subproblems.

This is done with the `partition` function.

Divide And Conquer

Divide: Two subproblems

Sort the elements to the left of the pivot element.

Sort the elements to the right of the pivot element.

Create/Complete/Combine:

Complete/Combine: No work to be done!

Partition

The main work of quicksort is in the partition function.

The partition function creates the subproblems.

Let's look at the signature of the partition function:

```
using vecIt = std::vector<int>::iterator;
vecIt partition(vecIt begin, vecIt end);
```

We take two iterators which define the half-closed interval where we work.

We return an iterator which points to final position of the pivot.

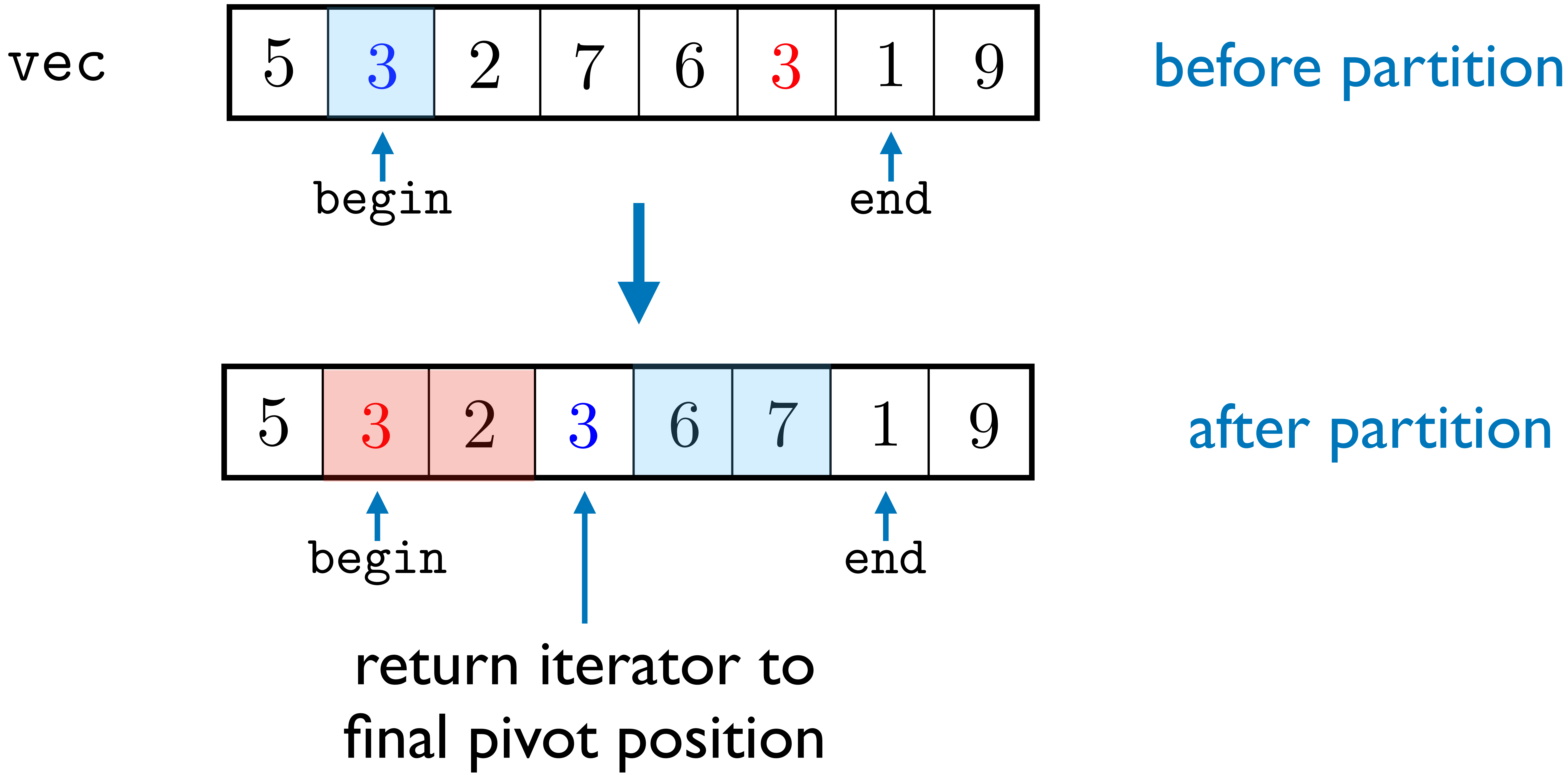
Partition

```
vecIt partition(vecIt begin, vecIt end)
```

The input iterators define a half-closed interval—we want to partition the elements in this interval.

We use `*begin` as the pivot element, in this case 3.

```
vecIt partition(vecIt begin, vecIt end)
```



Partition can be done in place in time $\Theta(\texttt{end} - \texttt{begin})$.

Quicksort

Let's set the implementation of partition aside for the moment and see how to finish writing quicksort.

```
void quicksort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt pivotIt = partition(begin, end);
  quicksort(begin, pivotIt);
  quicksort(pivotIt+1, end);
}
```

← base case: vector of size zero or one is already sorted.

Quicksort

Let's set the implementation of partition aside for the moment and see how to finish writing quicksort.

```
void quicksort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt pivotIt = partition(begin, end);
  quicksort(begin, pivotIt);
  quicksort(pivotIt+1, end);
}
```

← create the subproblems.

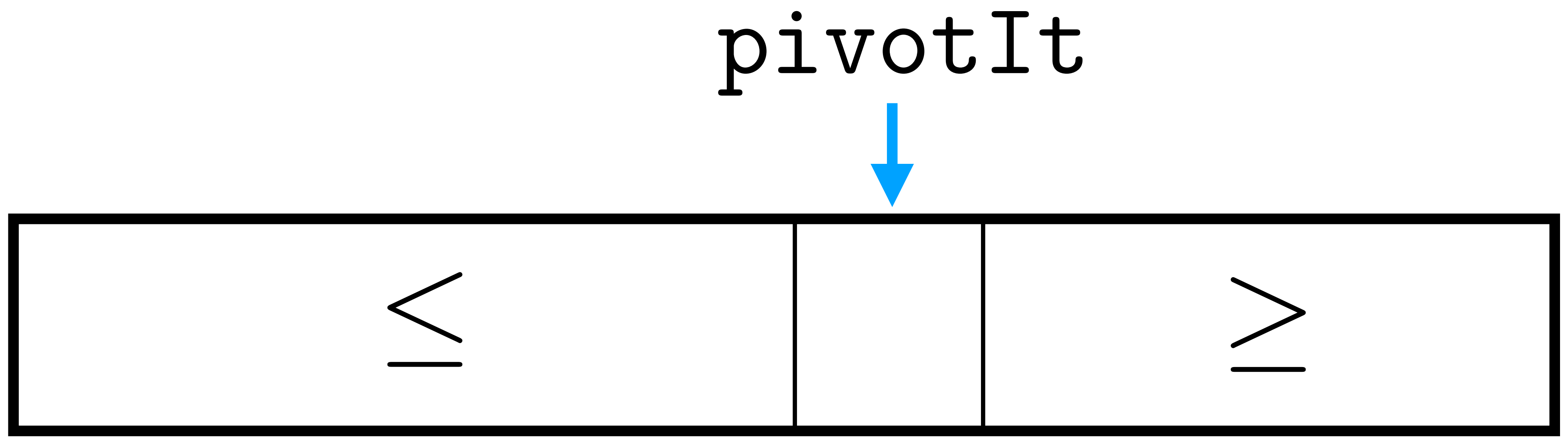
`partition` puts the pivot in its correct location, pointed to by `pivotIt`.

Quicksort

Let's set the implementation of partition aside for the moment and see how to finish writing quicksort.

```
void quicksort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt pivotIt = partition(begin, end);
  quicksort(begin, pivotIt);
  quicksort(pivotIt+1, end);
}
```

← create the subproblems.

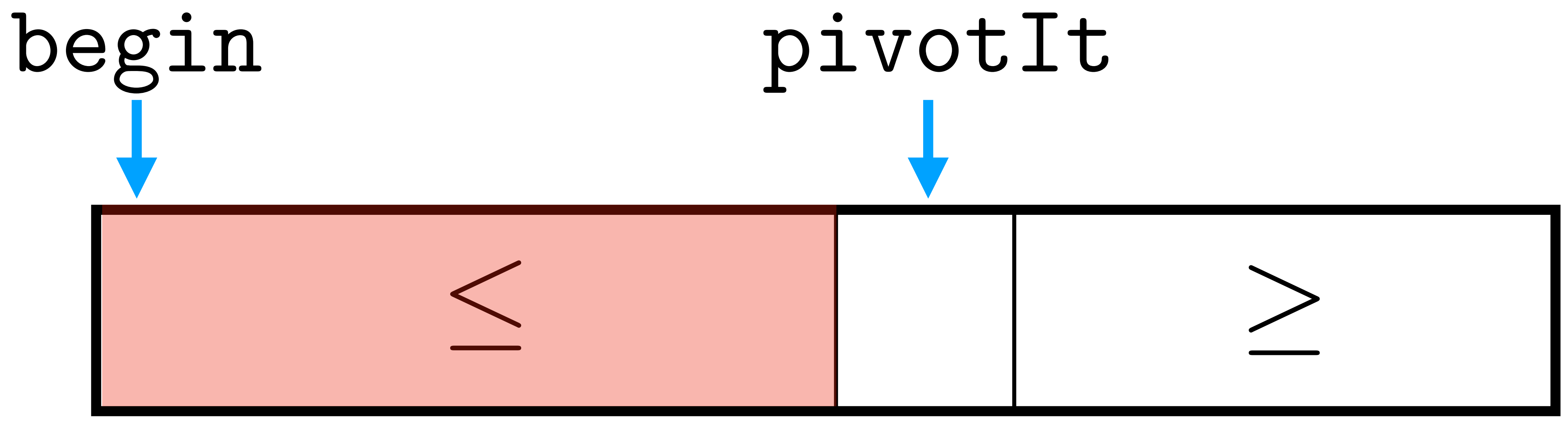


Quicksort

Let's set the implementation of partition aside for the moment and see how to finish writing quicksort.

```
void quicksort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt pivotIt = partition(begin, end);
  quicksort(begin, pivotIt);
  quicksort(pivotIt+1, end);
}
```

← recursively solve left subproblem.

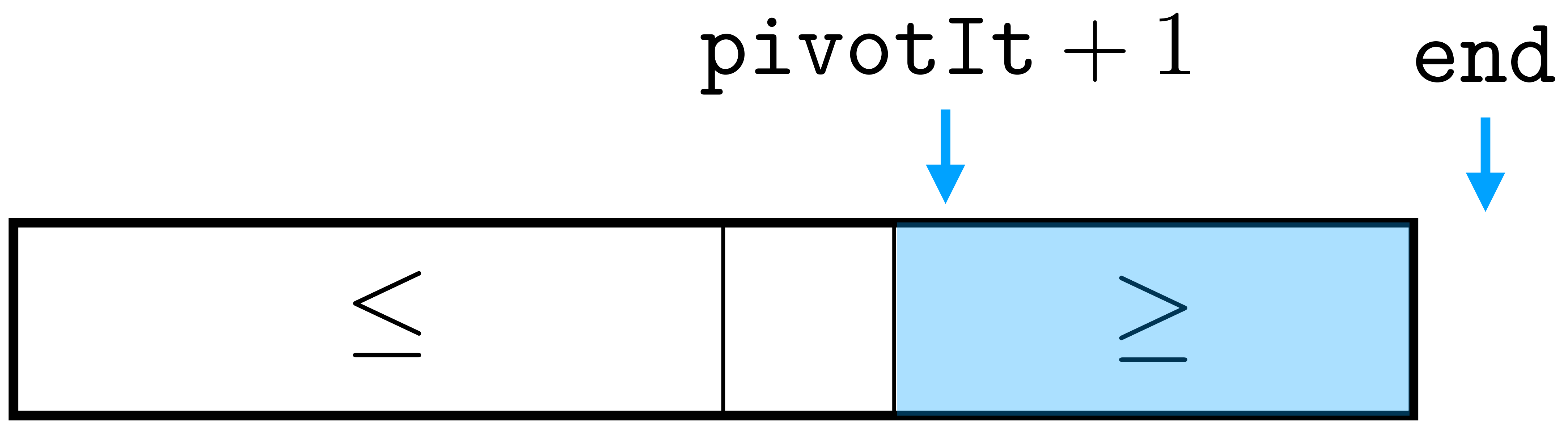


Quicksort

Let's set the implementation of partition aside for the moment and see how to finish writing quicksort.

```
void quicksort(vecIt begin, vecIt end) {
  if (end - begin <= 1) {
    return;
  }
  vecIt pivotIt = partition(begin, end);
  quicksort(begin, pivotIt);
  quicksort(pivotIt+1, end);
}
```

← recursively solve right subproblem.



Quicksort: Running Time

Quicksort: Running Time

Let's assume we are sorting a vector where all elements are distinct.

Quicksort: Running Time

Let's assume we are sorting a vector where all elements are distinct.

Let $T(n)$ be the time to sort a vector of size n with quicksort when we always pick the perfect pivot.

The perfect pivot makes the two subproblems (nearly) equal in size.

Quicksort: Running Time

Let's assume we are sorting a vector where all elements are distinct.

Let $T(n)$ be the time to sort a vector of size n with quicksort when we always pick the perfect pivot.

The perfect pivot makes the two subproblems (nearly) equal in size.

$$T(n) = T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + \Theta(n)$$

one subproblem

other subproblem

partition: create subproblems

Quicksort: Running Time

Let $T(n)$ be the time to sort a vector of size n with quicksort when we always pick the perfect pivot.

The perfect pivot makes the two subproblems (nearly) equal in size.

$$T(n) = T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + \Theta(n)$$

one subproblem

other subproblem

partition:
create subproblems

With $T(1) = O(1)$ this has the familiar solution $T(n) = \Theta(n \log n)$.

Pretty Good Pivot

Say a pivot is pretty good when it creates leads to subproblems that are both larger than $n/10$.

Now let $T(n)$ be the running time when we always pick a pretty good pivot.

We get a recurrence relation like the following:

$$T(n) \leq T(n/10) + T(9n/10) + O(n)$$

The solution to this recurrence is still $T(n) = O(n \log n)$.

Usually Pretty Good

Always choosing a pretty good pivot is also unrealistic. Sometimes we will have bad pivots.

More realistic is that, say, half the time, we will choose a good pivot. This still leads to an $O(n \log n)$ time algorithm.

Usually Pretty Good

Always choosing a pretty good pivot is also unrealistic. Sometimes we will have bad pivots.

More realistic is that, say, half the time, we will choose a good pivot. This still leads to an $O(n \log n)$ time algorithm.

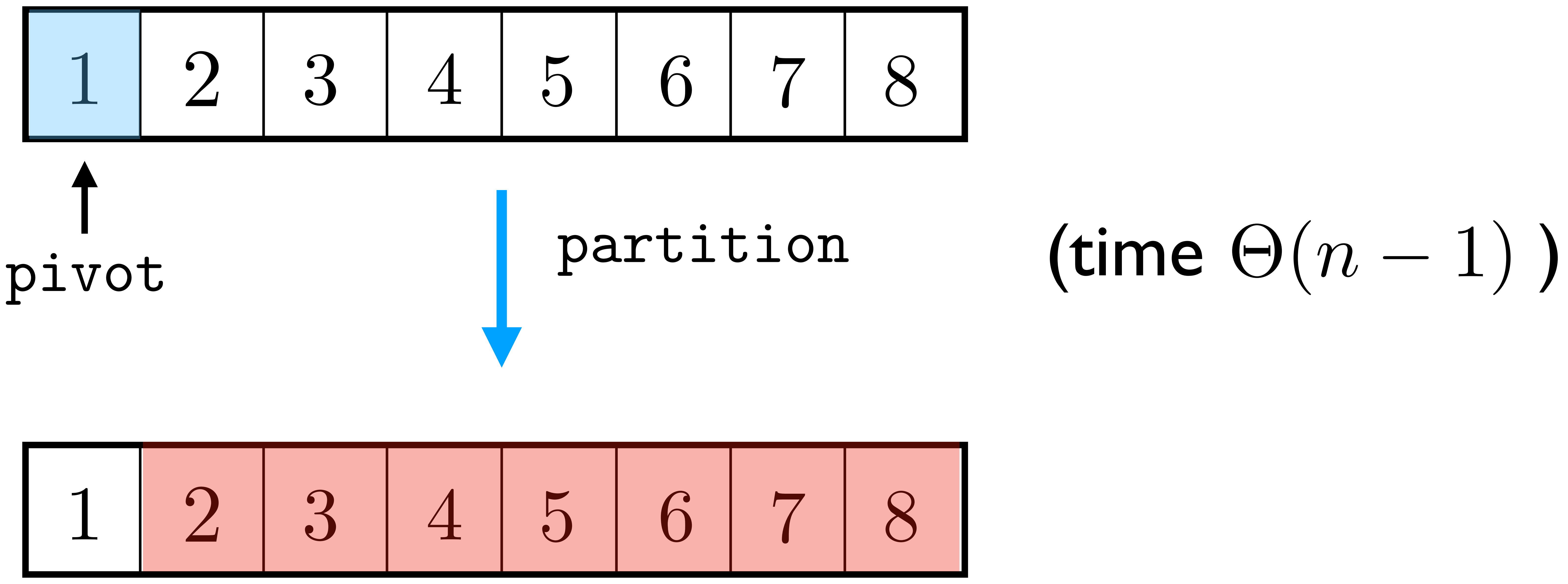
Take n distinct integers and look at the average running time of quicksort $n!$ over all permutations of them.

Usually the pivots will be pretty good—the average running time of quicksort over all possible permutations is $\Theta(n \log n)$.

Quicksort: Worst Case

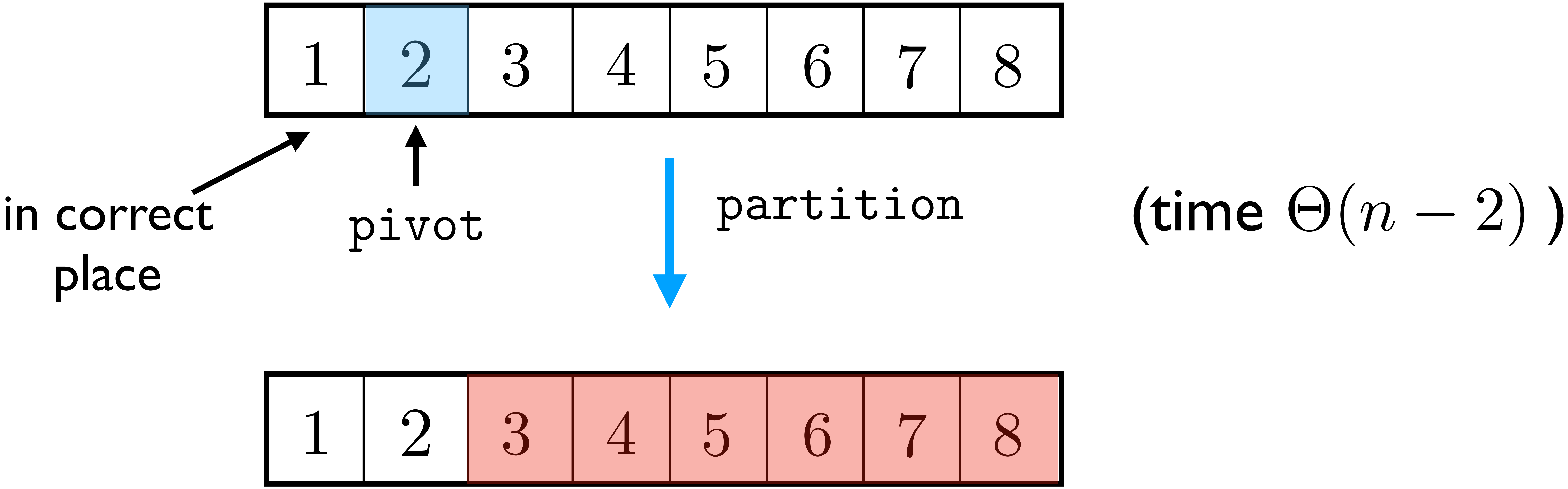
In the worst case quicksort can take time $\Theta(n^2)$.

A bad case for our version of quicksort is when the vector is already sorted.



Left subproblem has size 0, right subproblem has size $n-1$.

Quicksort: Worst Case



Left subproblem has size 0, right subproblem has size $n-2$.

We only decrease the problem size by one each time.

Quicksort: Worst Case

When the vector is already sorted we only decrease the problem size by after each round of `partition`.

The running time is proportional to

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$$

After `partition` we always put at least one element in the correct position—at most n rounds of `partition`.

The worst-case running time of quicksort is $\Theta(n^2)$.

Partition

Lomuto Partition

Several different algorithms have been suggested do the partition step of quicksort.

The original algorithm of Hoare from 1961 uses two approaching indices.

We will describe a simpler (but slightly slower) algorithm due to Lomuto.

Lomuto Partition

Several different algorithms have been suggested do the partition step of quicksort.

The original algorithm of Hoare from 1961 uses two approaching indices.

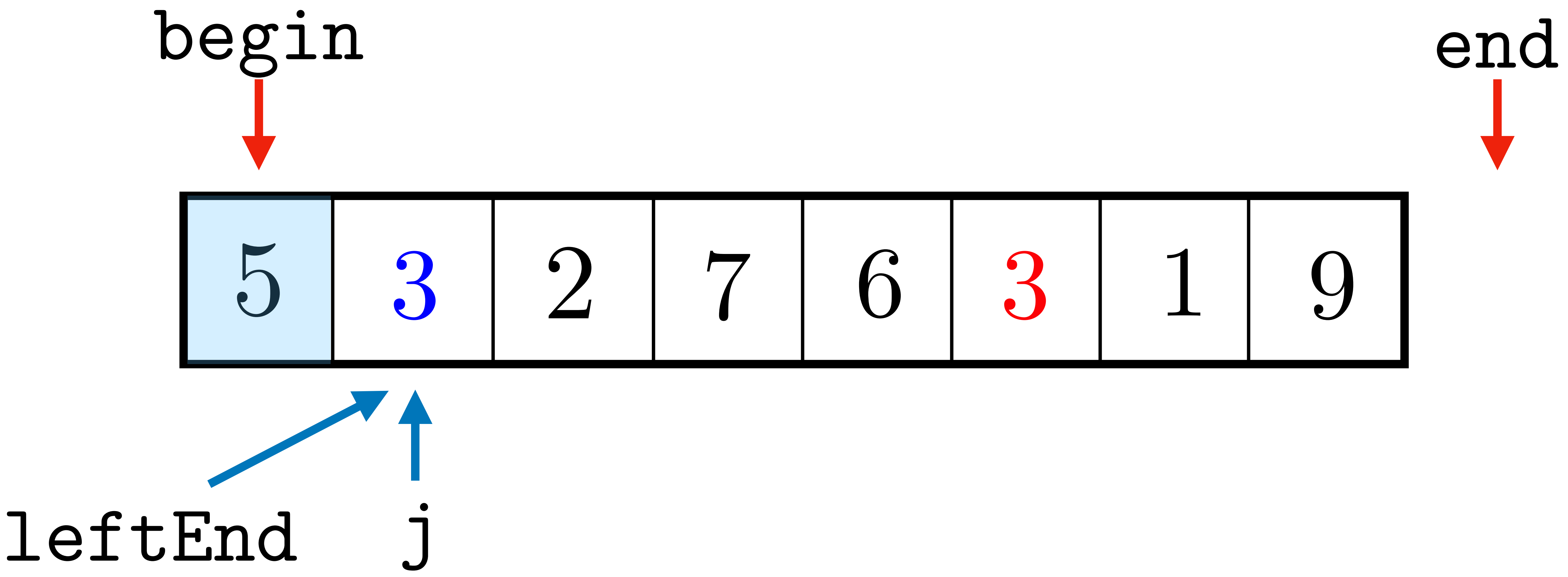
We will describe a simpler (but slightly slower) algorithm due to Lomuto.

---

[2] Most discussions of Quicksort use a partitioning scheme based on two approaching indices like the one described in Problem 3. Although the basic idea of that scheme is straightforward, I have always found the details tricky—I once spent the better part of two days chasing down a bug hiding in a short partitioning loop. A reader of a preliminary draft complained that the standard two-index method is in fact simpler than Lomuto's, and sketched some code to make his point; I stopped looking after I found two bugs.

—Jon Bentley, Programming Pearls

Lomuto Example



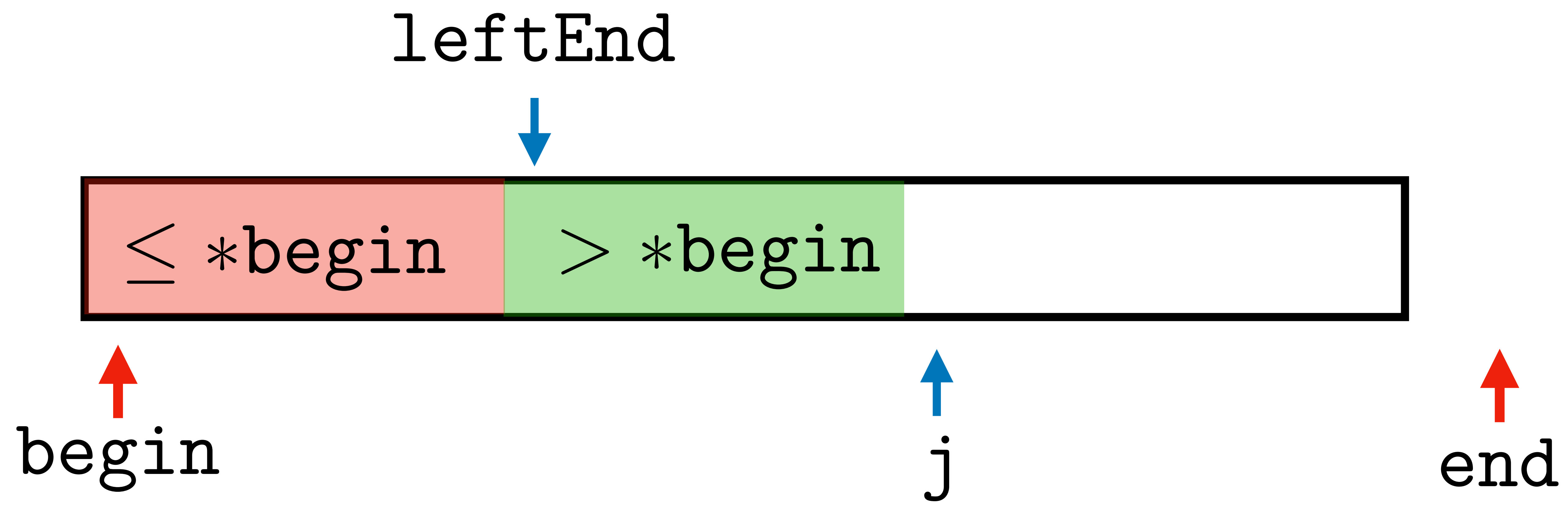
We use `*begin` as the pivot and initialize $\texttt{leftEnd} = \texttt{begin} + 1$.

The iterator `j` starts at $\texttt{begin} + 1$ and runs over the vector.

$\texttt{begin} < \texttt{leftEnd} \leq \texttt{j}$ partition the vector into three parts in general.

Lomuto Example

$\texttt{begin} < \texttt{leftEnd} \leq \texttt{j}$ partition the vector into three parts in general.

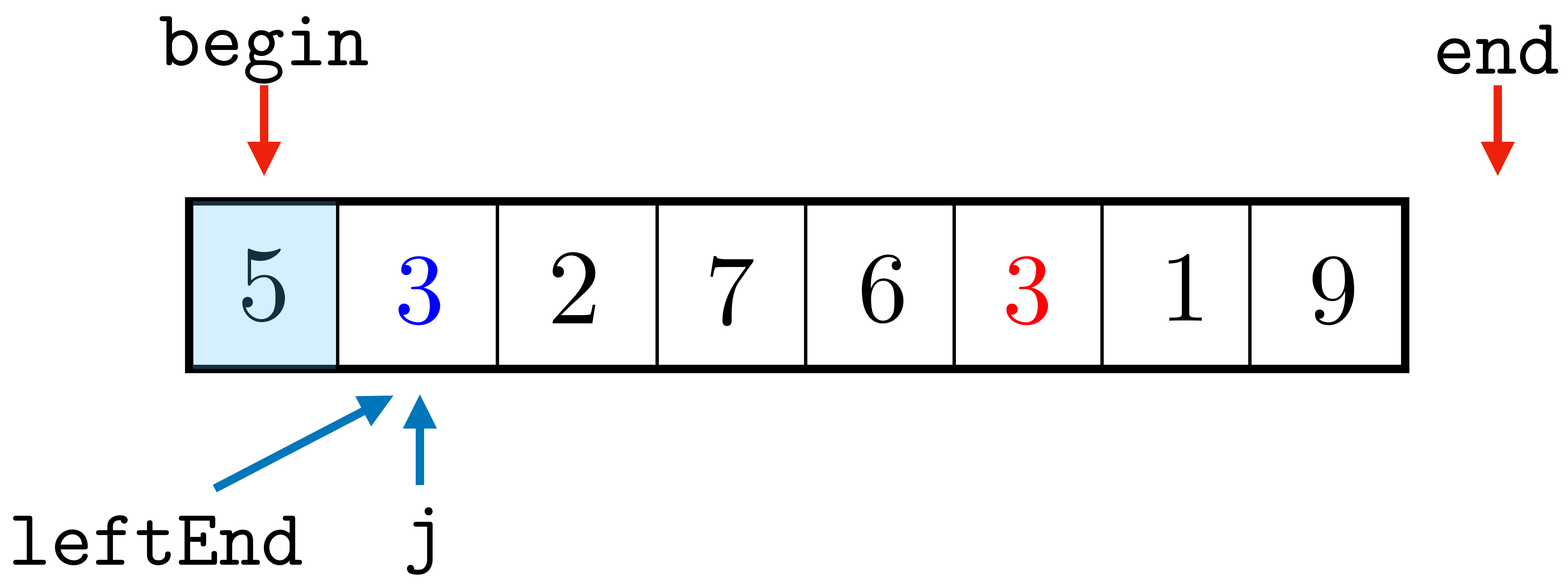


Elements in $[\texttt{begin}, \texttt{leftEnd})$ are at most the pivot.

Elements in $[\texttt{leftEnd}, \texttt{j})$ are greater than the pivot.

Elements in $[\texttt{j}, \texttt{end})$ are still to be processed.

Initialization



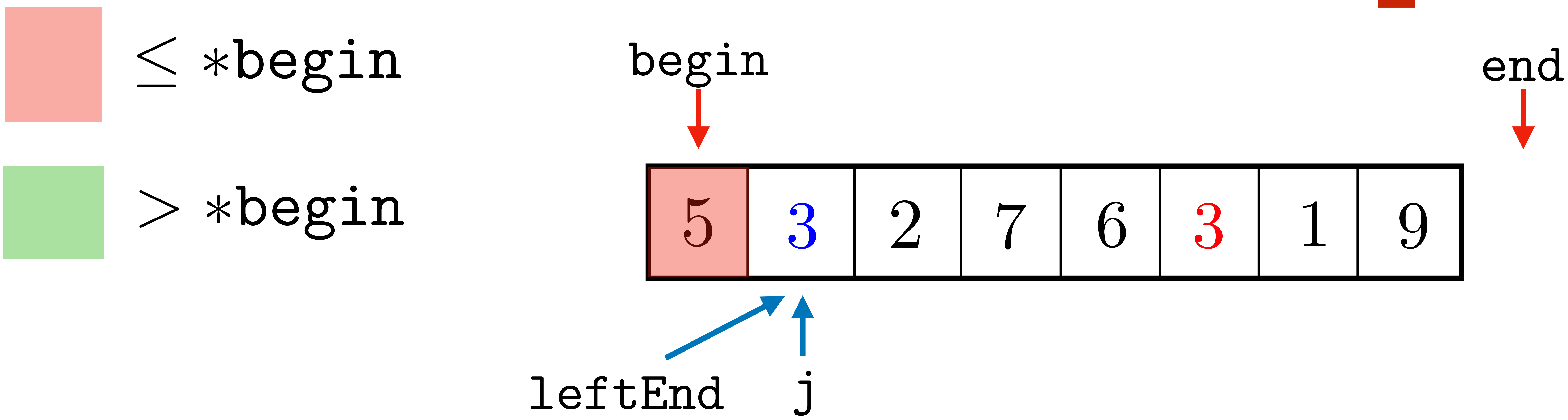
Elements in $[\texttt{begin}, \texttt{leftEnd})$ are at most the pivot. Just the pivot 😀.

Elements in $[\texttt{leftEnd}, \texttt{j})$ are greater than the pivot. Empty 😀.

Elements in $[\texttt{j}, \texttt{end})$ are still to be processed. Everything but the pivot 😀.

Lomuto Loop

Godbolt Link

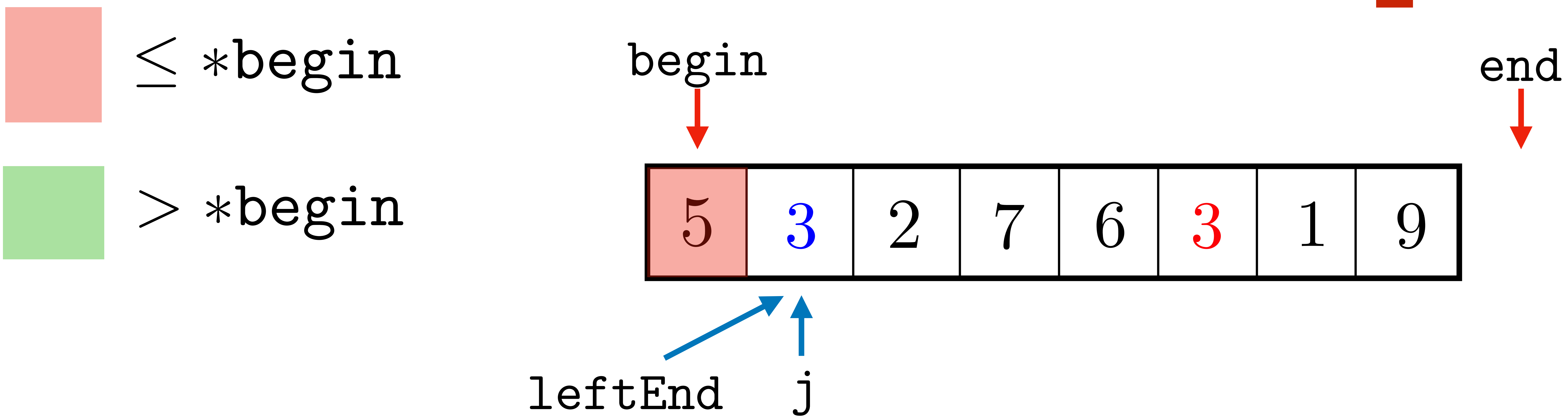



```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

Lomuto Loop

[Godbolt Link]



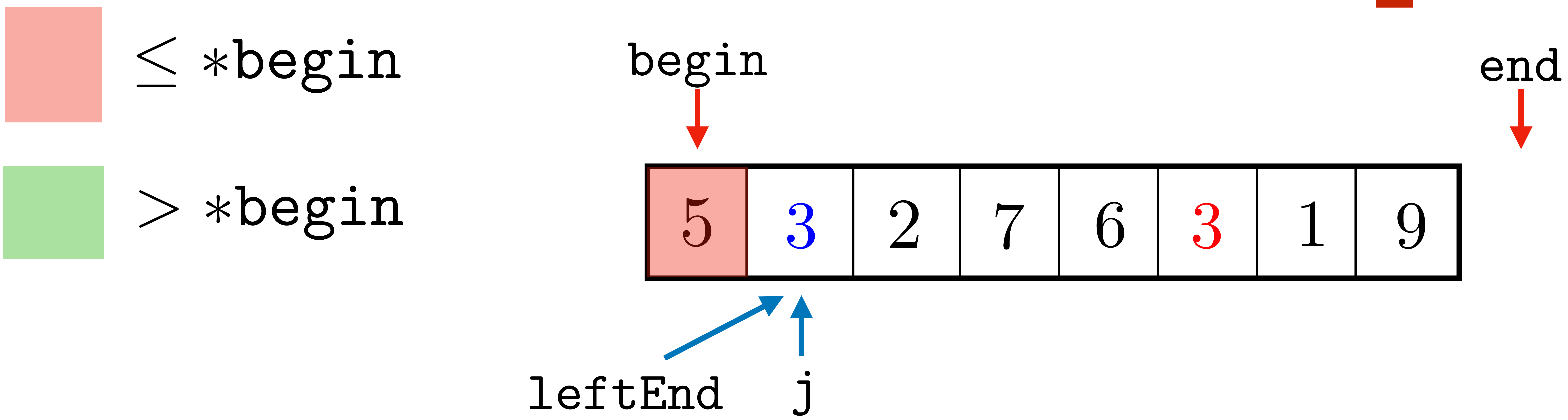

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

First iteration: $*\texttt{j} \leq *\texttt{begin}$

Let's see why this loop maintains the invariant.

Lomuto Loop

Godbolt Link




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

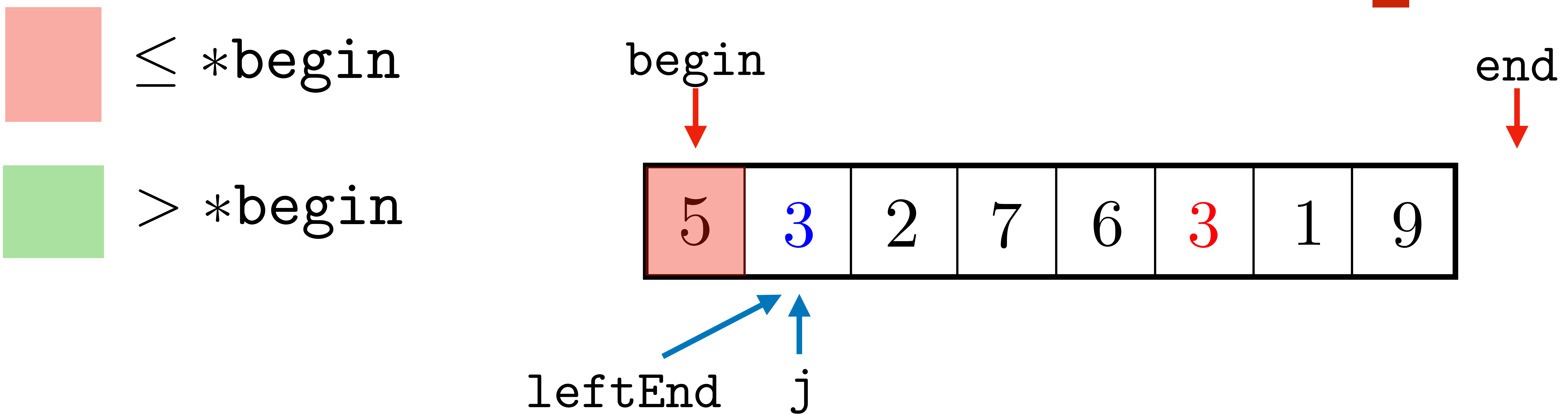
Let's see why this loop maintains the invariant.

First iteration: $*j \leq *begin$

We swap, which does nothing in this case.

Lomuto Loop

Godbolt Link




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

First iteration: $*j \leq *begin$

We swap, which does nothing in this case.

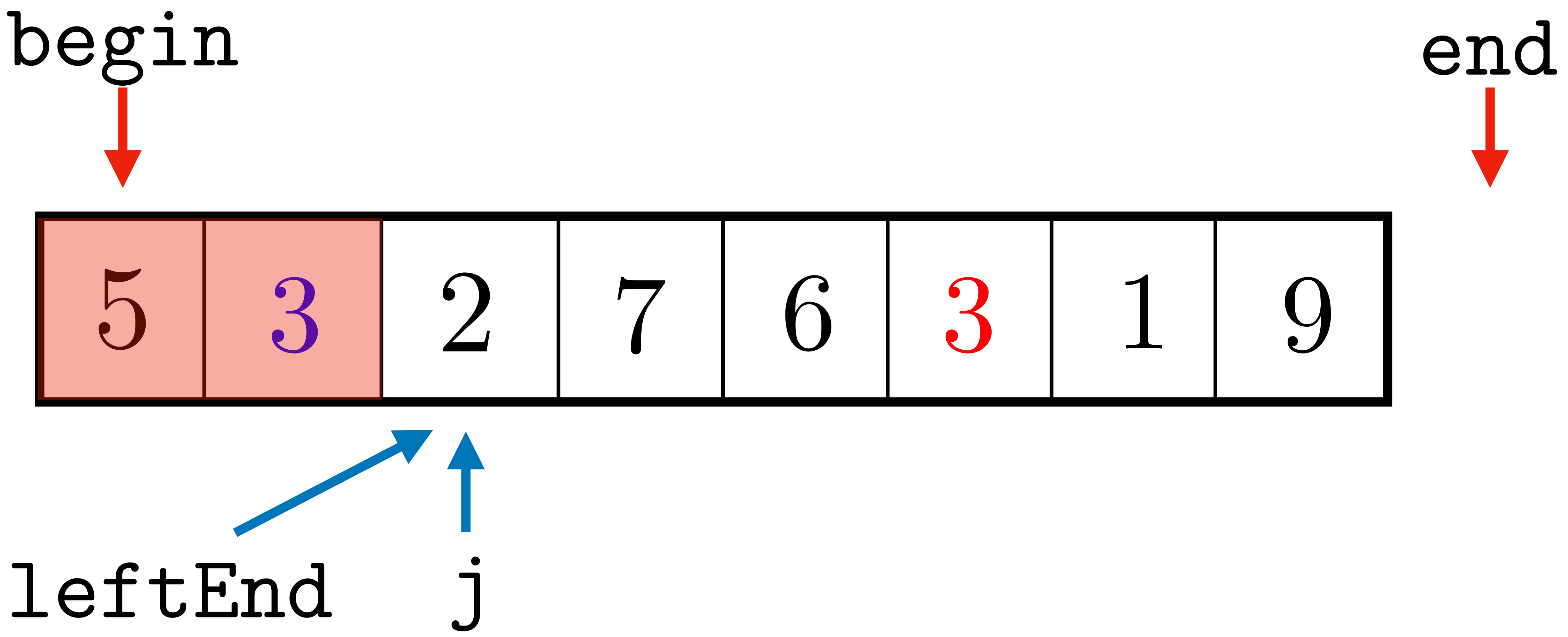
We increment `leftEnd` .

Lomuto Loop

Godbolt Link

$\leq$ *begin

$>$ *begin

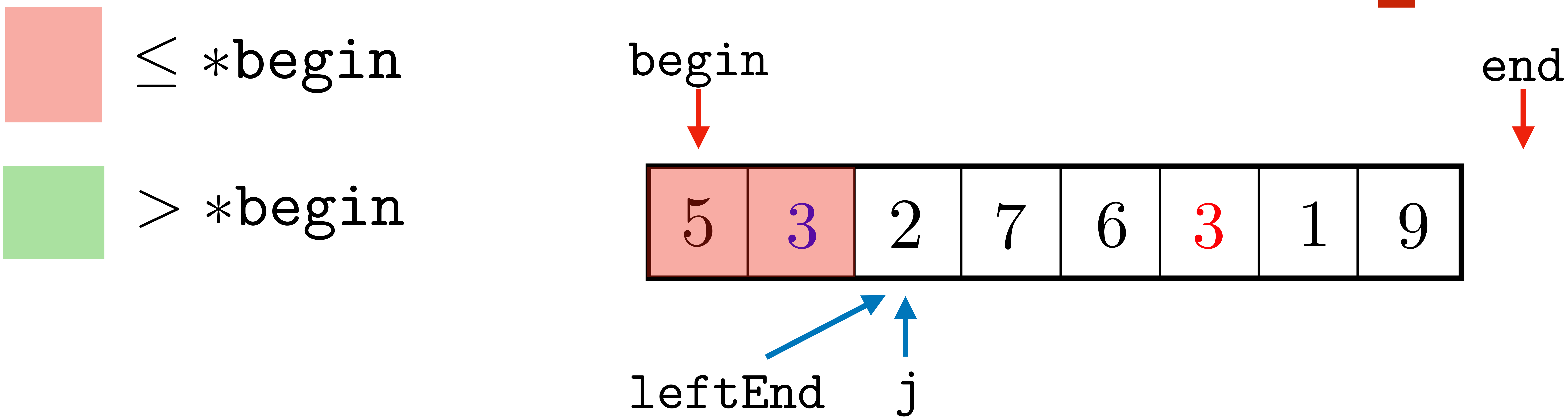



```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

Lomuto Loop

Godbolt Link



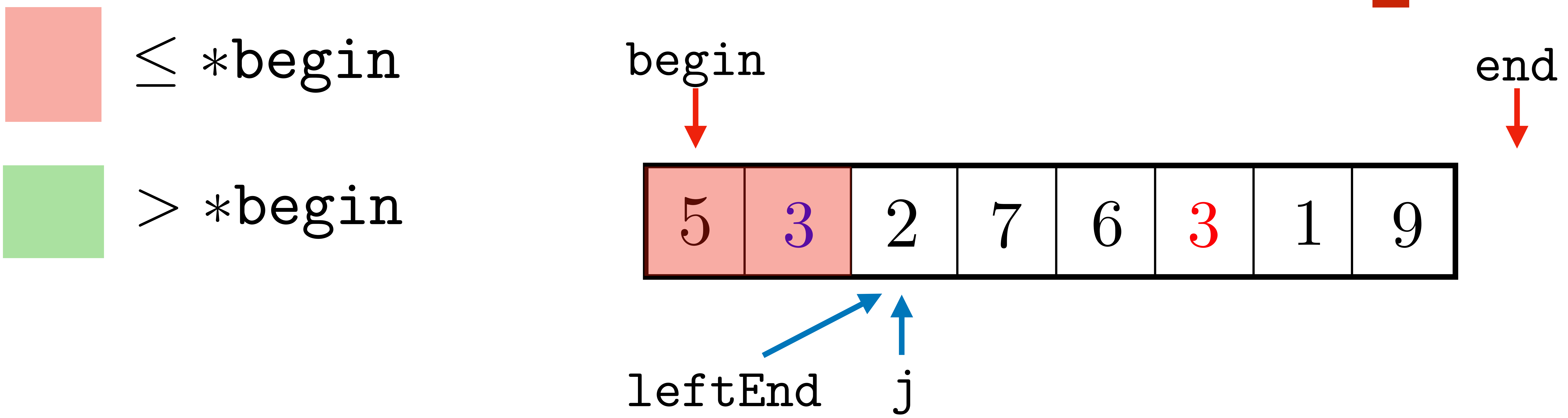

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

Second iteration: $*j \leq *begin$

Lomuto Loop

Godbolt Link




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

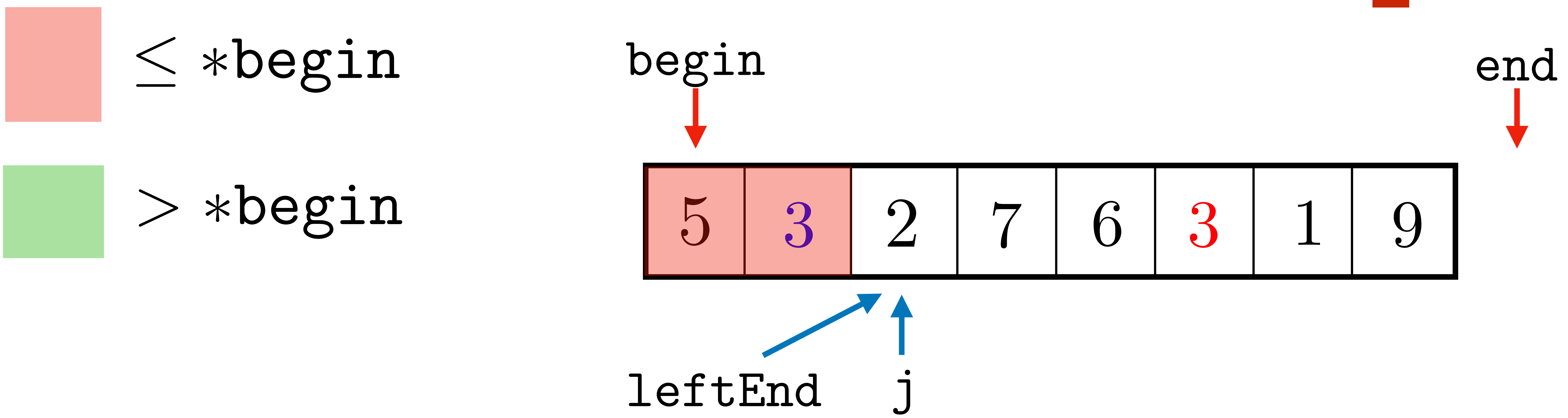
Let's see why this loop maintains the invariant.

Second iteration: $*j \leq *begin$

We swap, which again does nothing.

Lomuto Loop

[Godbolt Link]




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

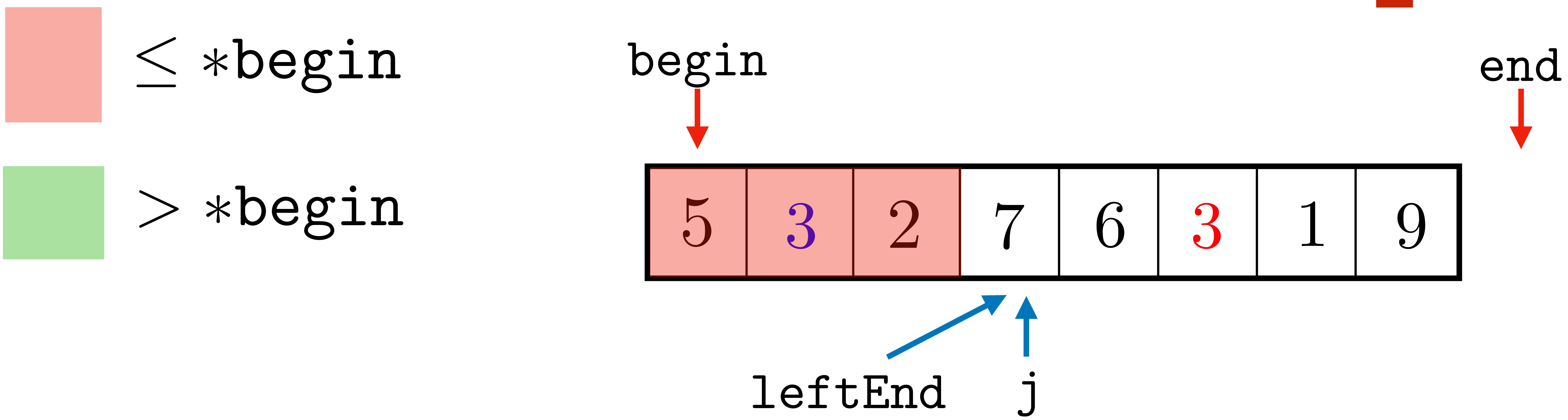
Second iteration: $*\texttt{j} \leq *\texttt{begin}$

We swap, which again does nothing.

We increment `leftEnd` .

Lomuto Loop

[Godbolt Link]

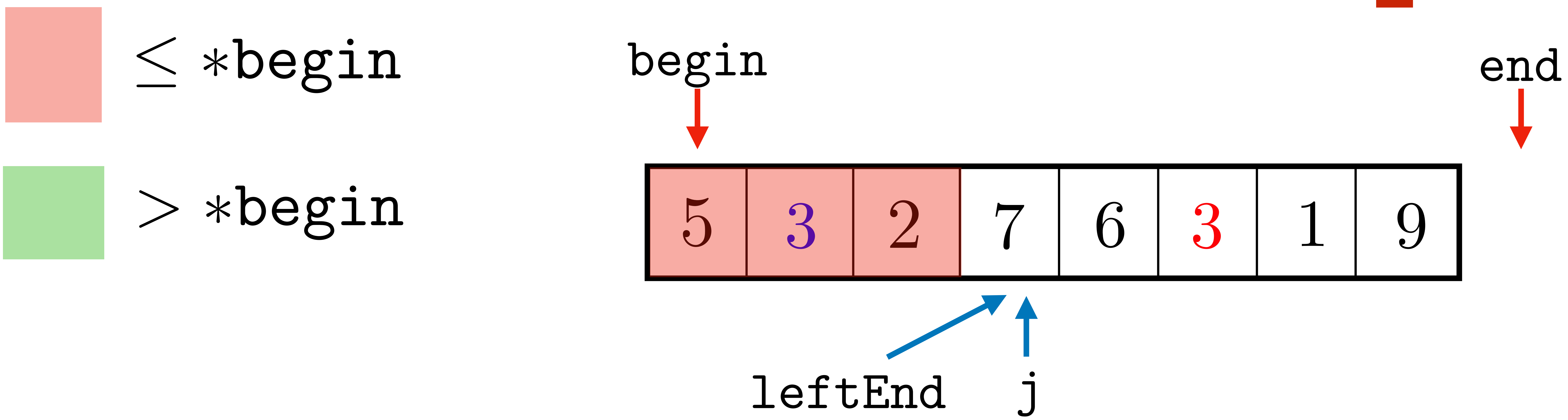



```cpp
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

Lomuto Loop

Godbolt Link




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

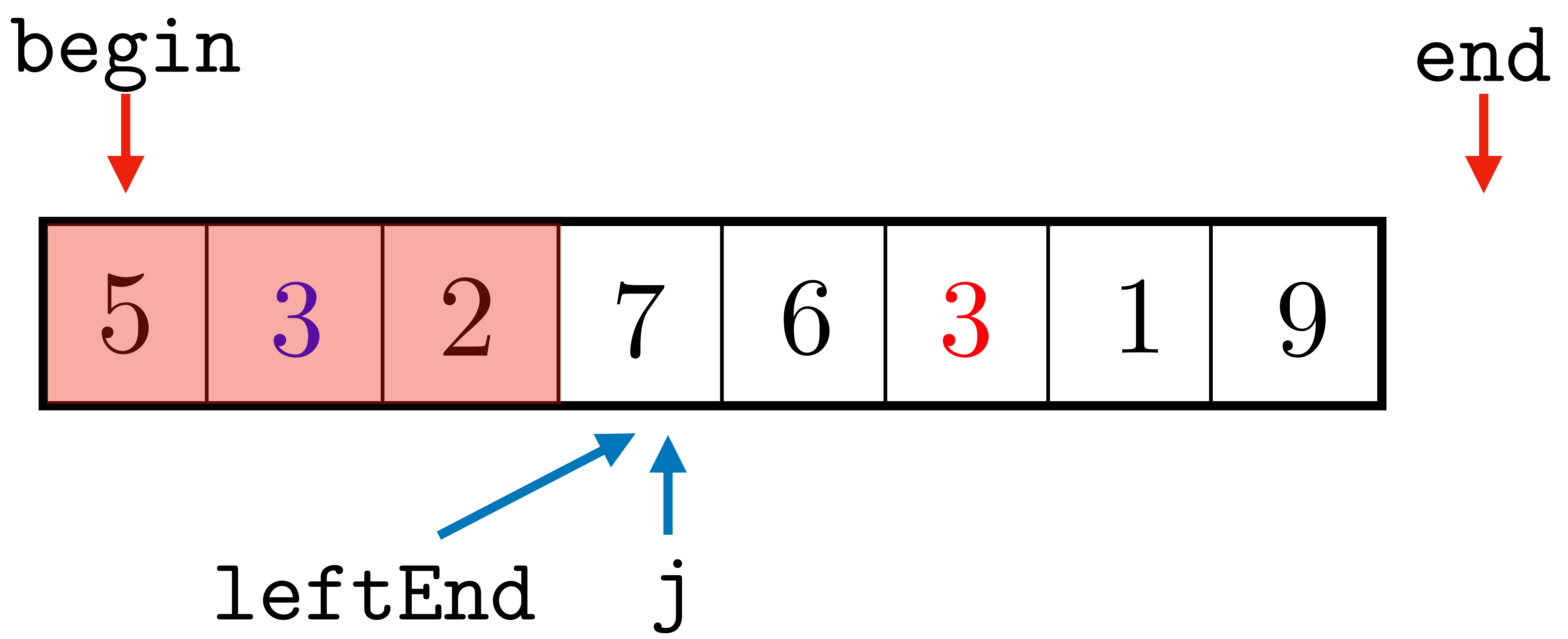
Third iteration: $*j > *begin$

Lomuto Loop

[Godbolt Link]

$\leq$ *begin

$>$ *begin




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

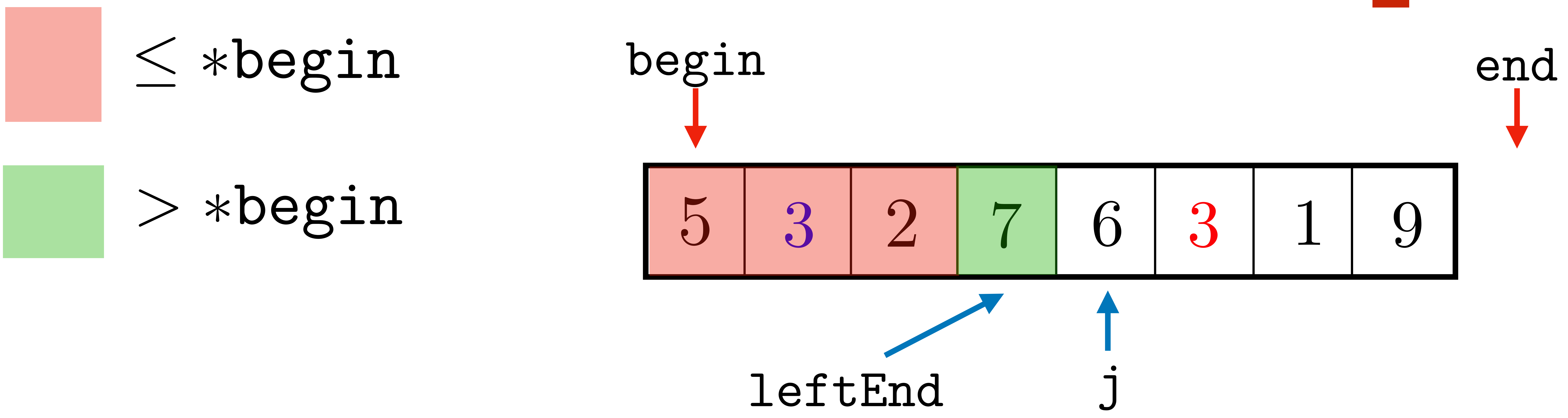
Let's see why this loop maintains the invariant.

Third iteration: $*j > *begin$

No swap, do not increment `leftEnd`.

Lomuto Loop

[Godbolt Link]




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

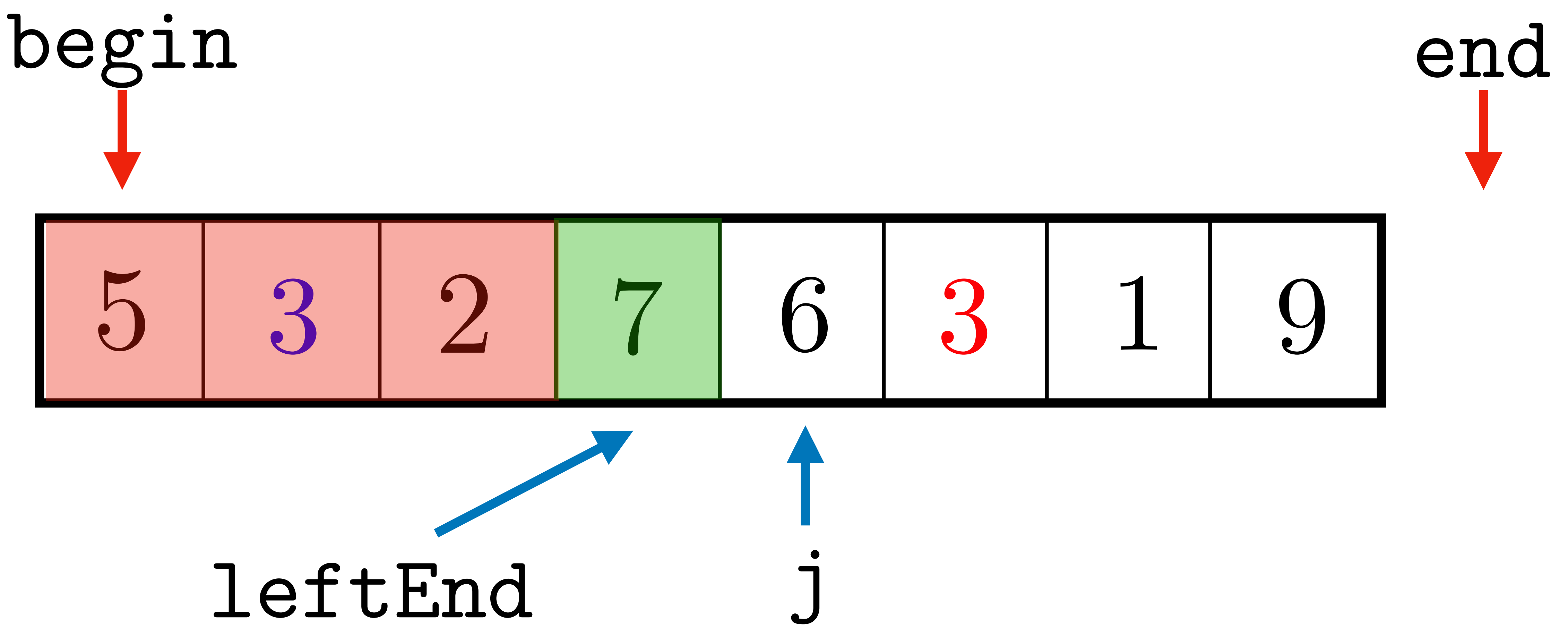
No swap, do not increment `leftEnd`.

Lomuto Loop

[Godbolt Link]

$\leq$ *begin

$>$ *begin




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

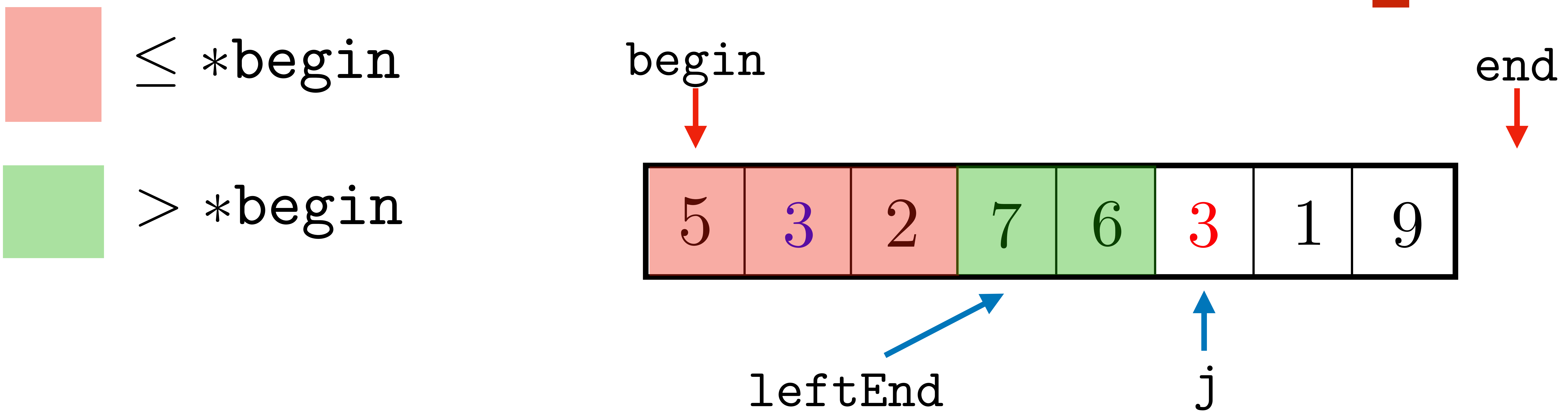
Let's see why this loop maintains the invariant.

Fourth iteration: $*\texttt{j} > *\texttt{begin}$

No swap, do not increment `leftEnd`.

Lomuto Loop

Godbolt Link




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

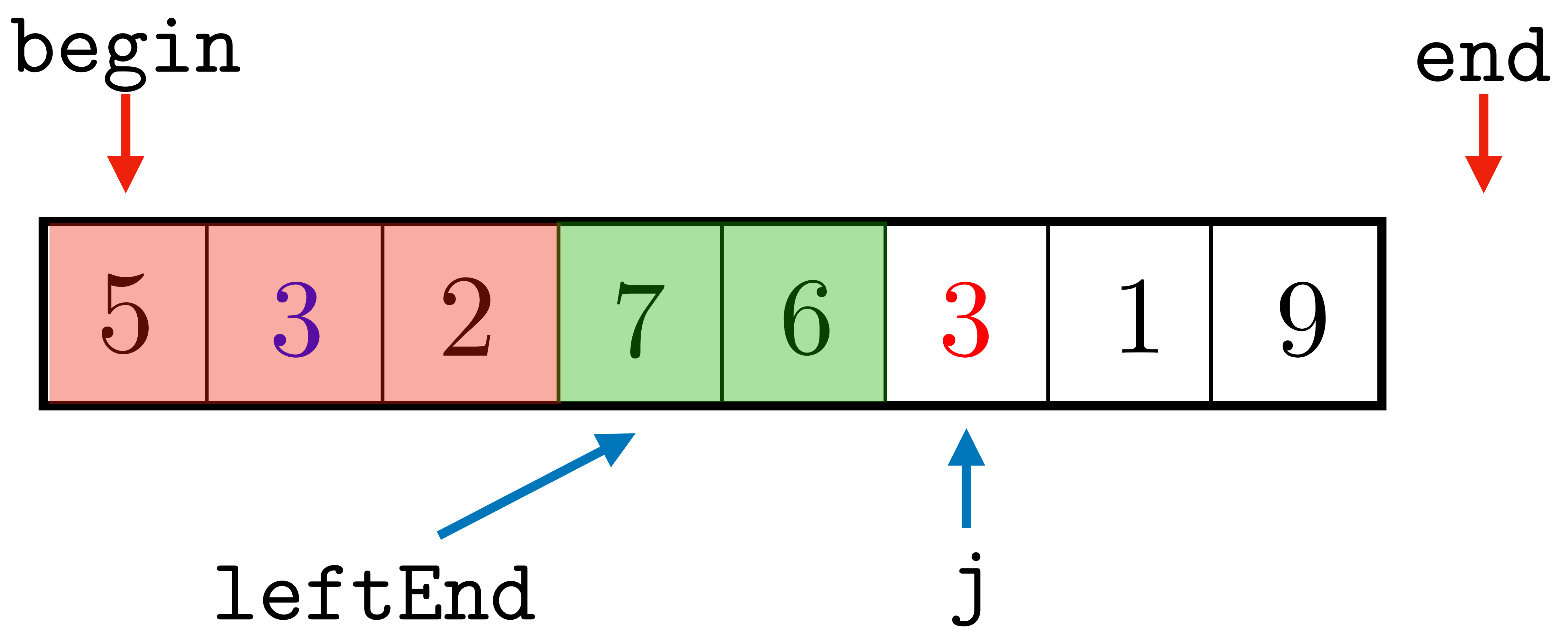
Let's see why this loop maintains the invariant.

Lomuto Loop

[Godbolt Link]

≤ *begin

> *begin




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

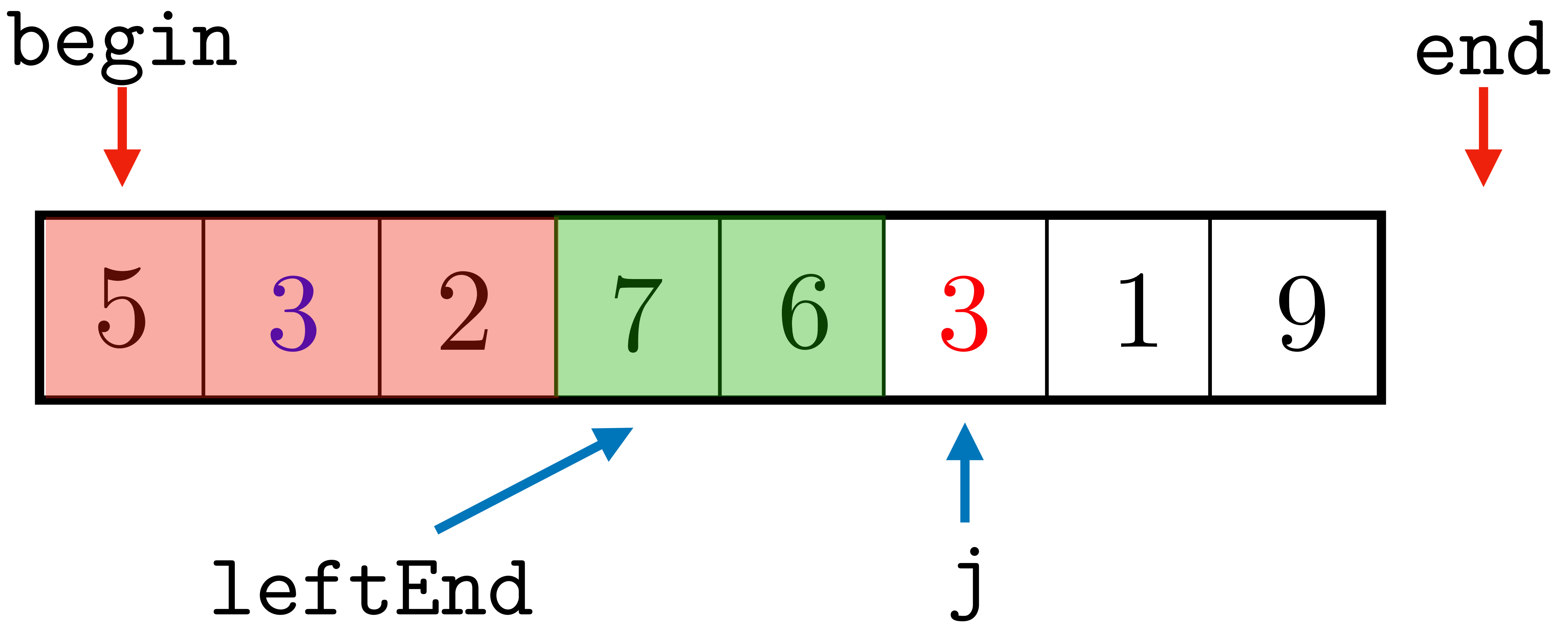
Fifth iteration: $*j \leq *begin$

Lomuto Loop

[Godbolt Link]

$\leq$ *begin

$>$ *begin




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Let's see why this loop maintains the invariant.

Fifth iteration: $*j \leq *begin$

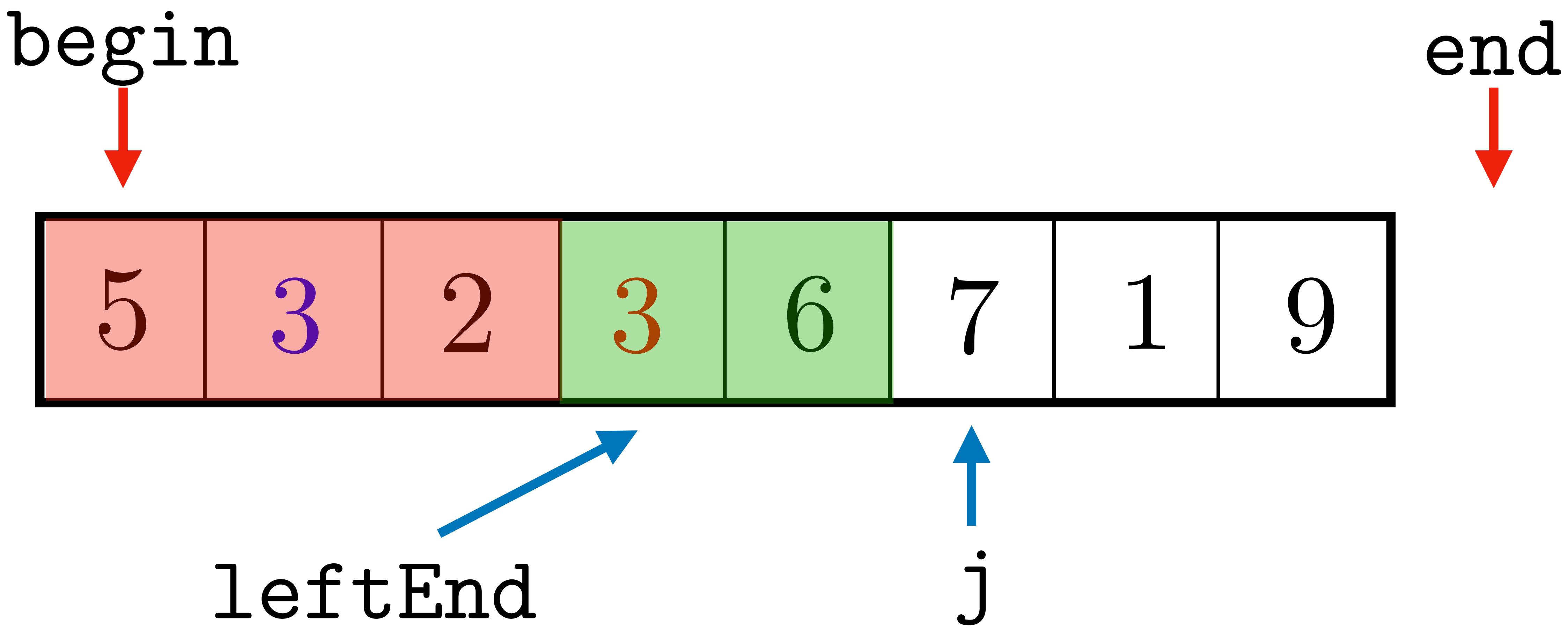
Swap $*$leftEnd and $*$j .

Lomuto Loop

[Godbolt Link]

$\leq$ *begin

$>$ *begin




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

After the swap we have $*\texttt{leftEnd} \leq *\texttt{begin}$ and $*\texttt{j} > *\texttt{begin}$.

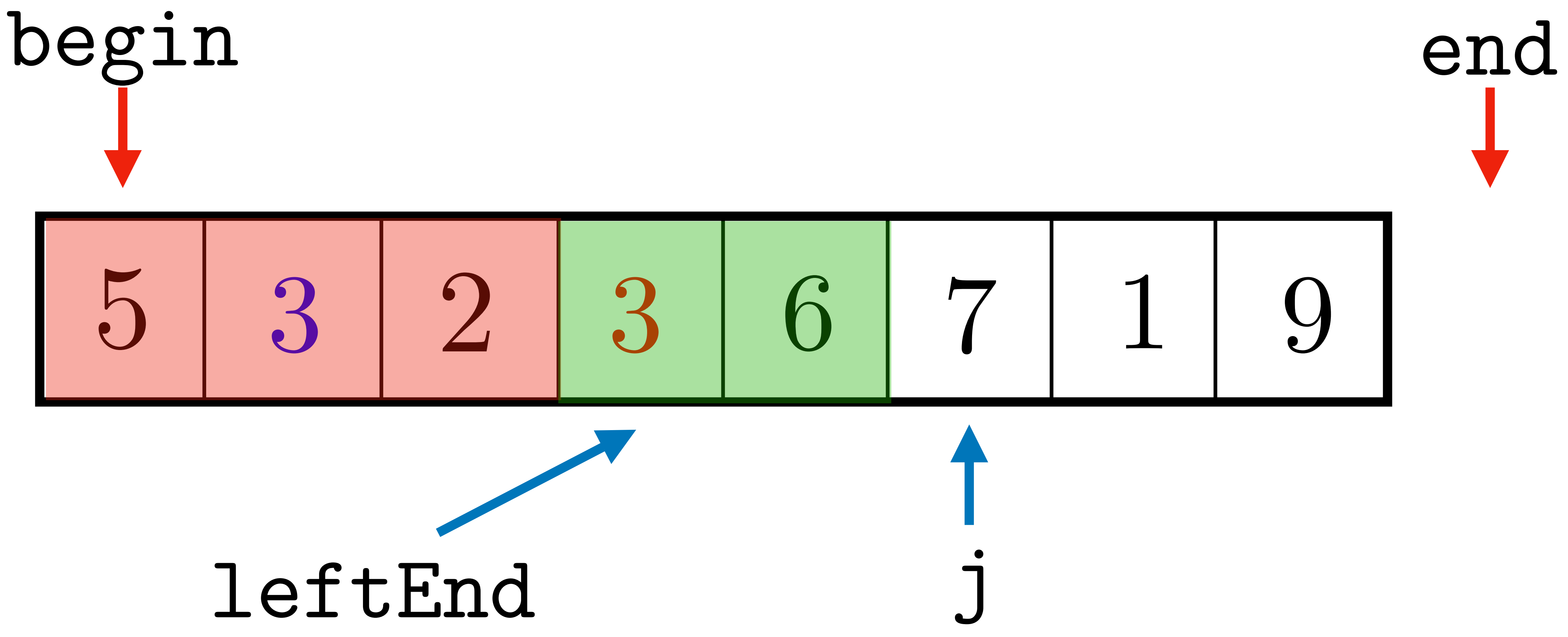
Increment `leftEnd`.

Lomuto Loop

[Godbolt Link]

$\leq *\texttt{begin}$

$> *\texttt{begin}$




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

After the swap we have $*\texttt{leftEnd} \leq *\texttt{begin}$ and $*\texttt{j} > *\texttt{begin}$.

Fifth iteration: $*\texttt{j} \leq *\texttt{begin}$

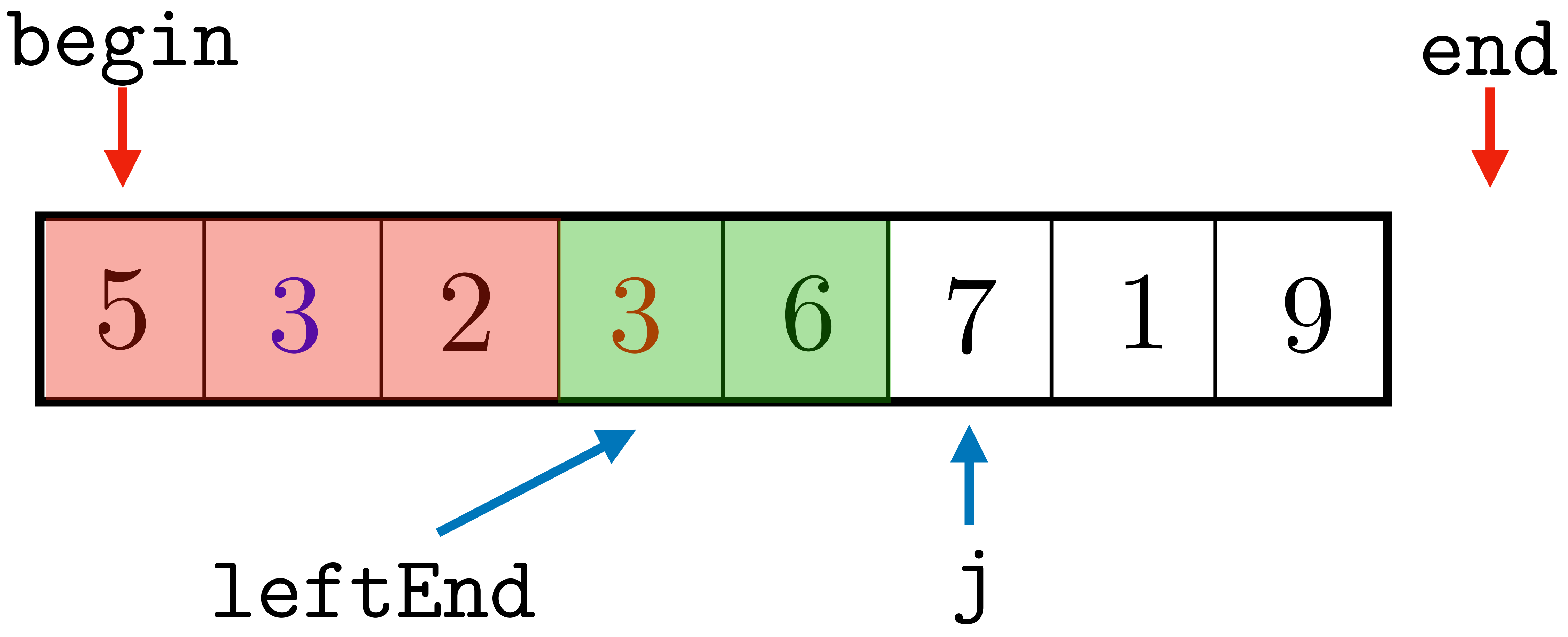
Increment `leftEnd`.

Lomuto Loop

[Godbolt Link]

$\leq$ *begin

$>$ *begin




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

After the swap we have $*\texttt{leftEnd} \leq *\texttt{begin}$ and $*\texttt{j} > *\texttt{begin}$.

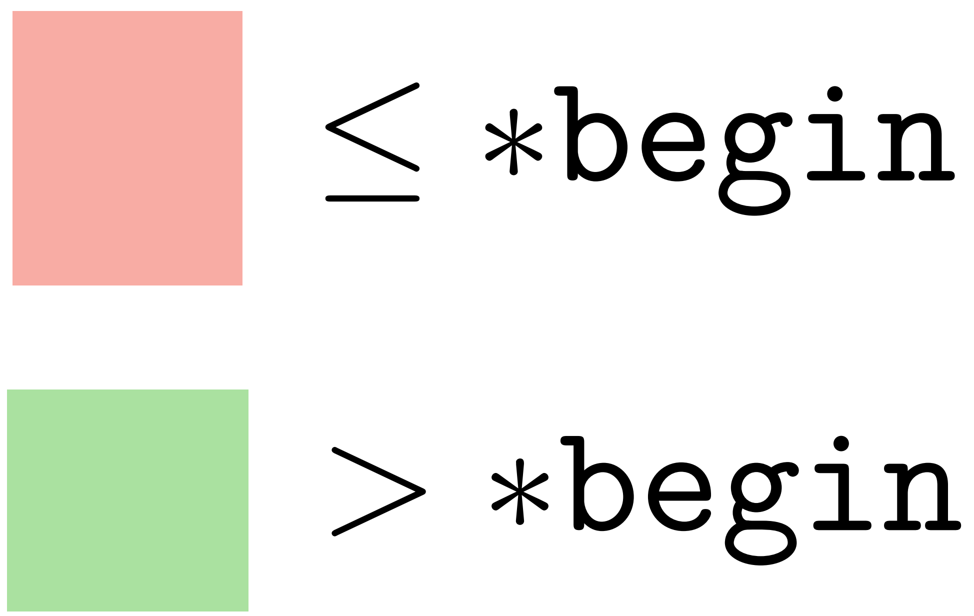
Fifth iteration: $*\texttt{j} \leq *\texttt{begin}$

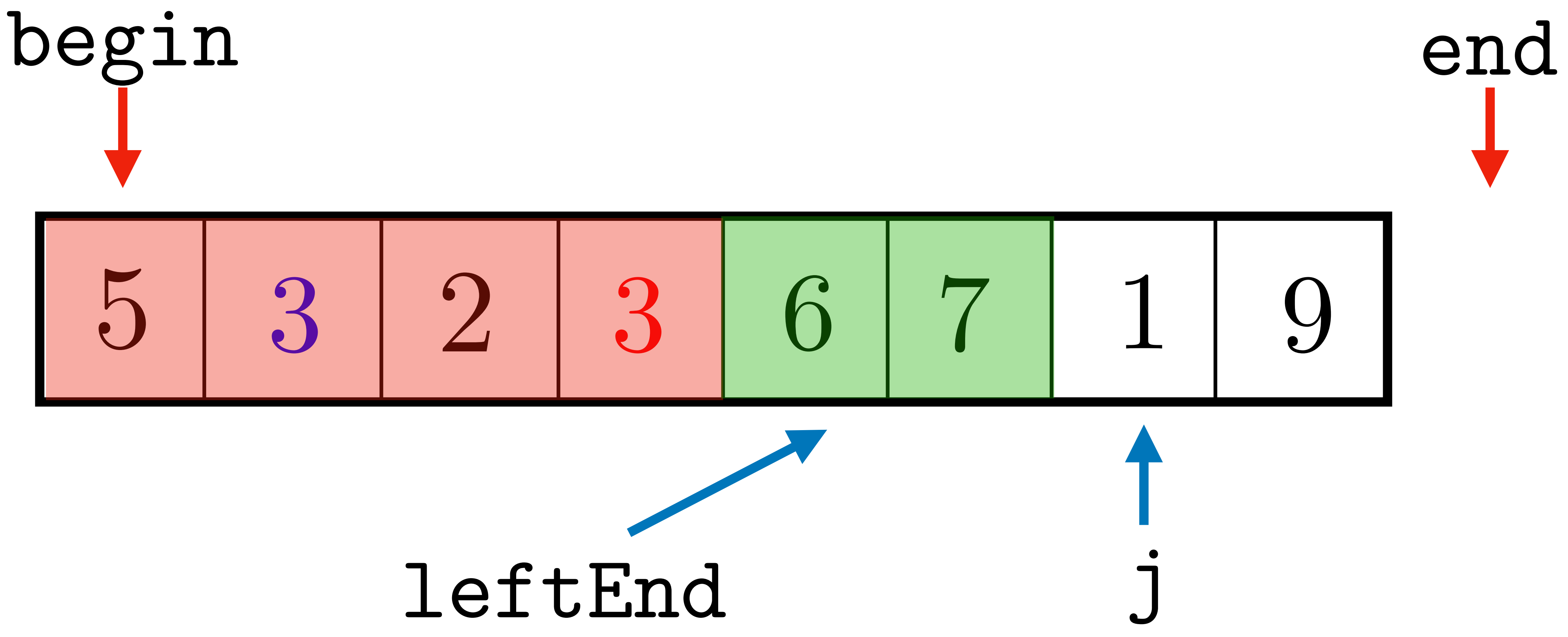
Swap *leftEnd and *j .

Increment leftEnd.

Lomuto Loop

[Godbolt Link]





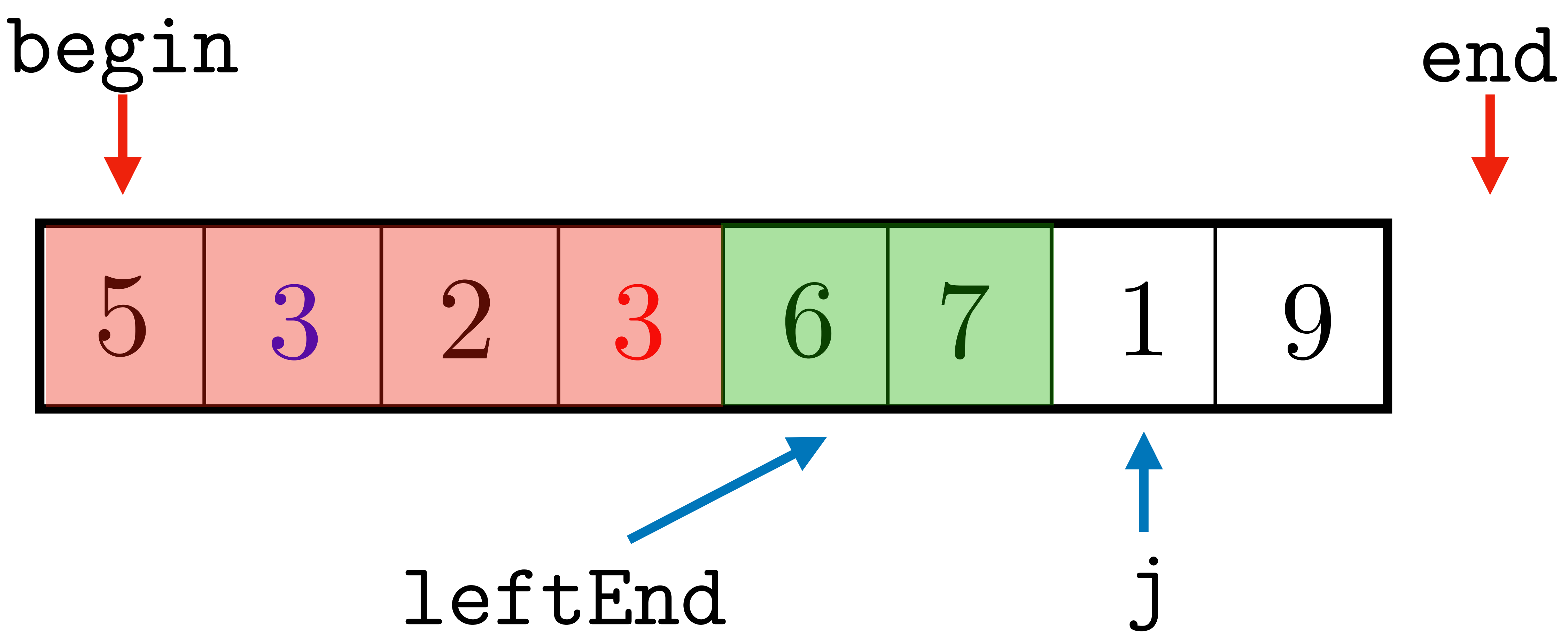

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Lomuto Loop

Godbolt Link

$\leq$ *begin

$>$ *begin



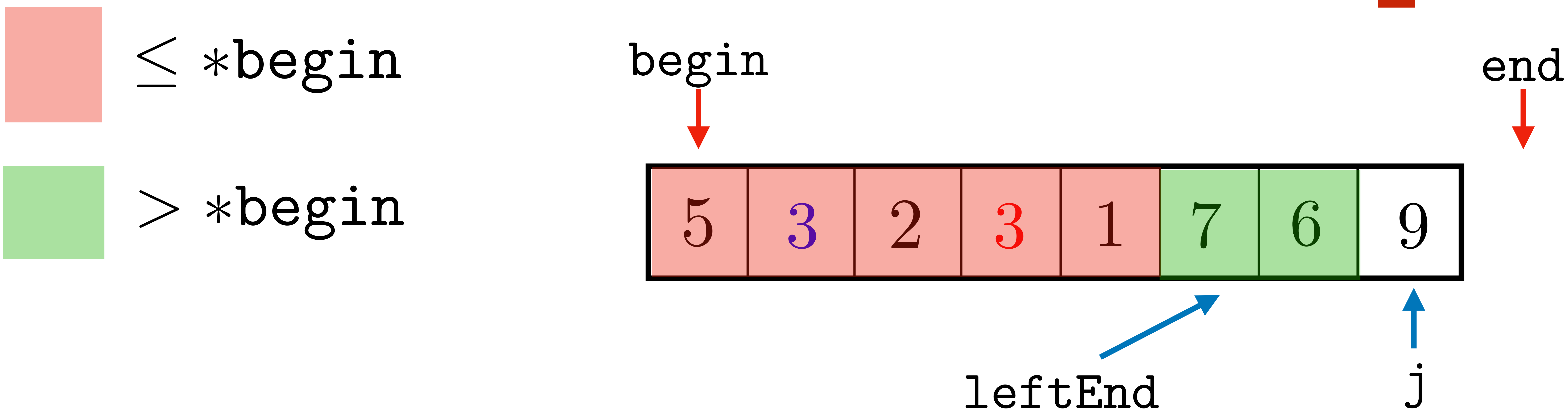

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Sixth iteration: $*j \leq *\texttt{begin}$

Swap, and increment `leftEnd`.

Lomuto Loop

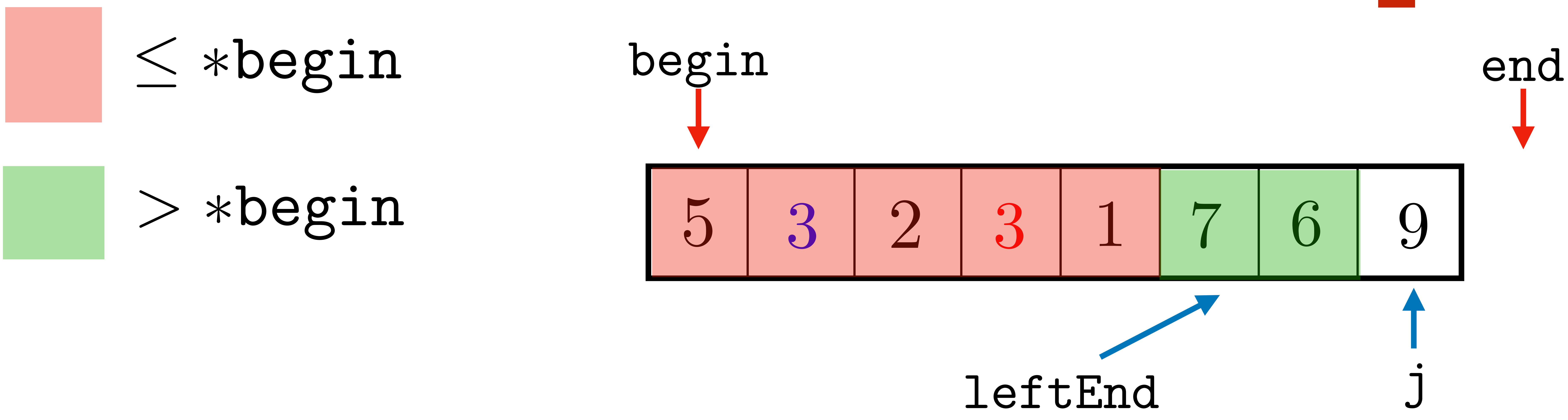
[Godbolt Link]




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Lomuto Loop

[Godbolt Link]



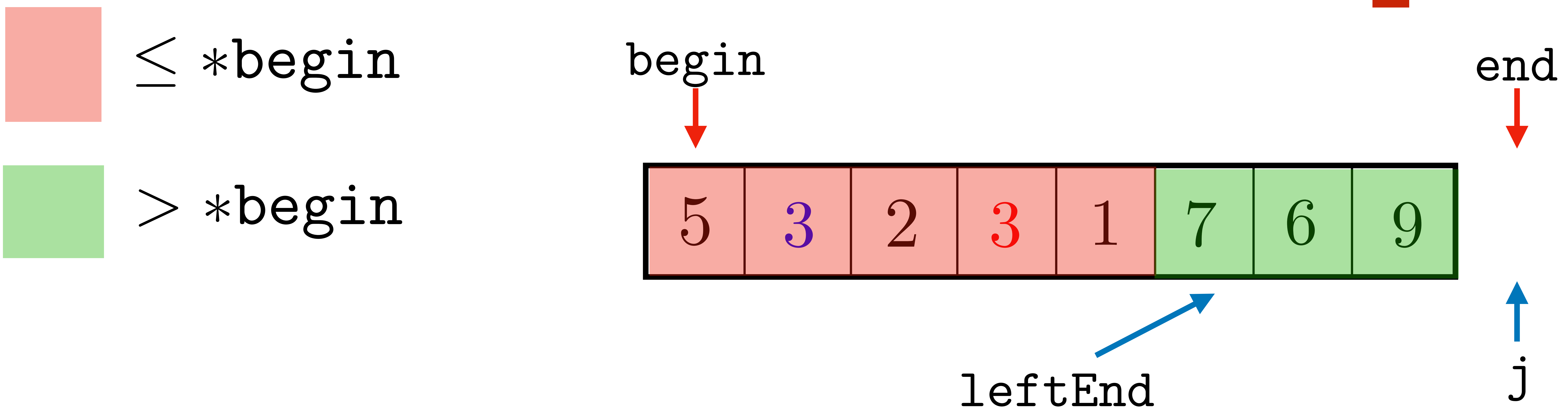

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Seventh iteration: $*j > *begin$

No swap, do not increment `leftEnd`.

Lomuto Loop

[Godbolt Link]

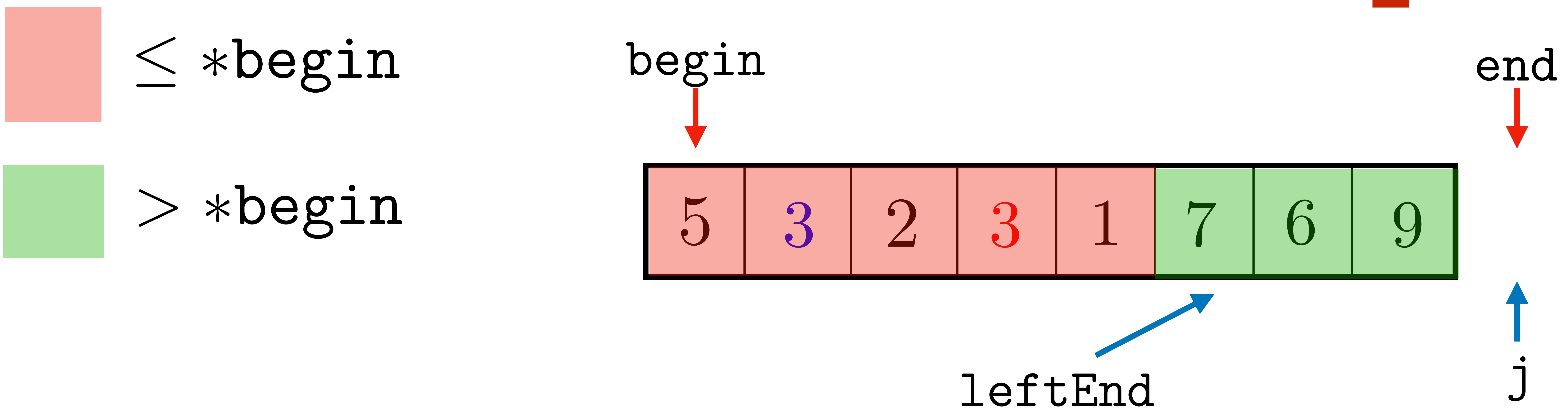



```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Finally, `j == end` and the for loop terminates.

Lomuto Loop

[Godbolt Link]

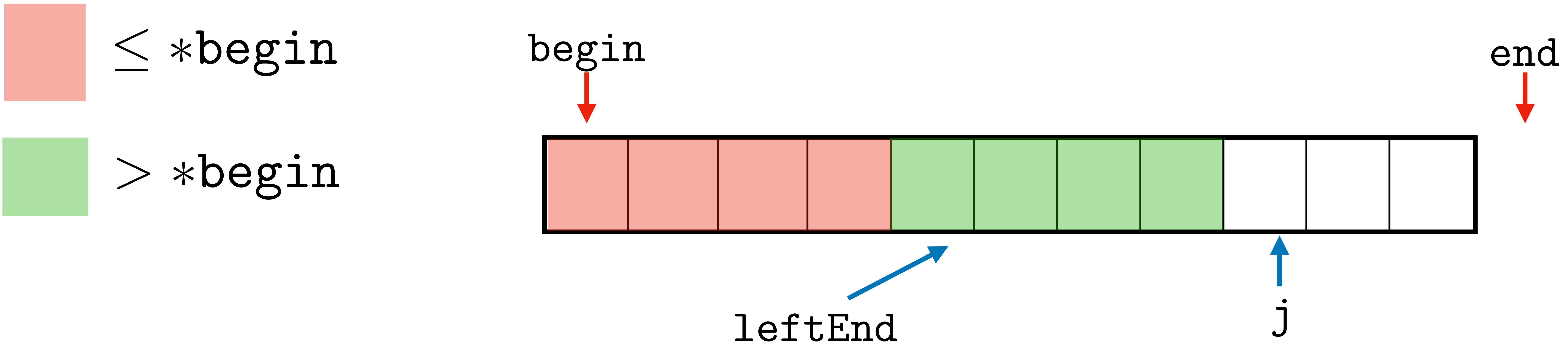



```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Finally, `j` $==$ `end` and the for loop terminates.

We still need to check two things: why the loop maintains the invariant in general, and what the invariant gives us at the end of the loop.

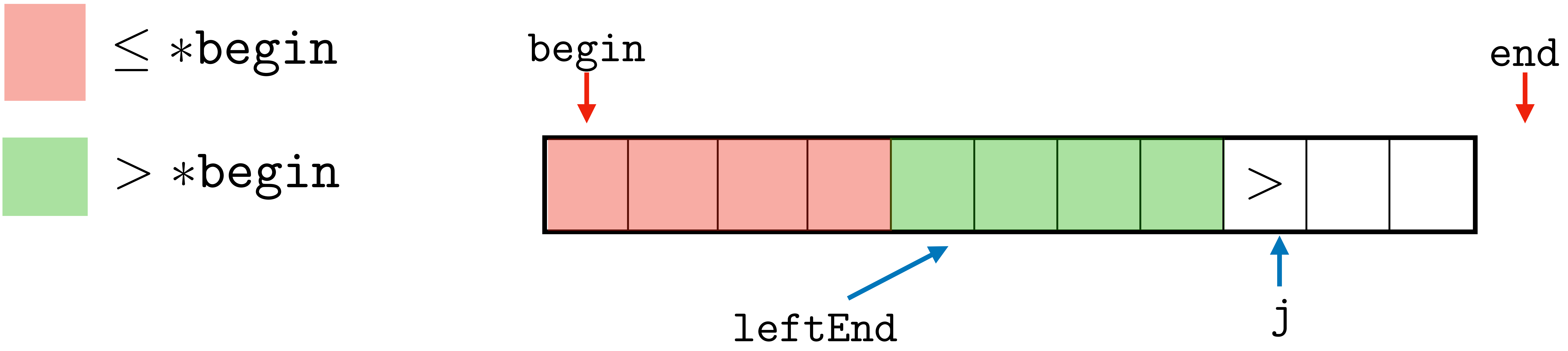
Maintenance




```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

In general, there are two cases:

Maintenance



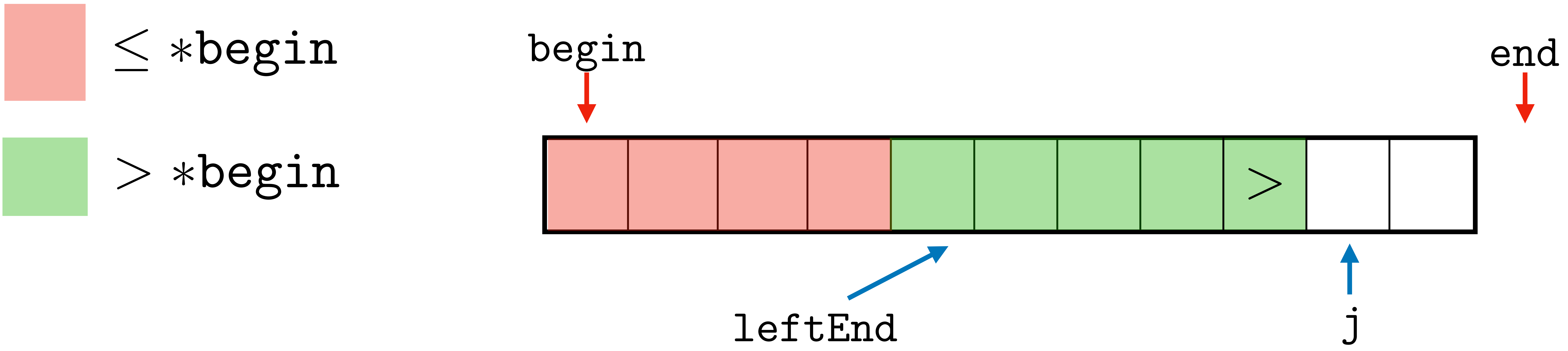

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

In general, there are two cases:

Case 1: $*j > *begin$

No swap, no increment of `leftEnd`.

Maintenance



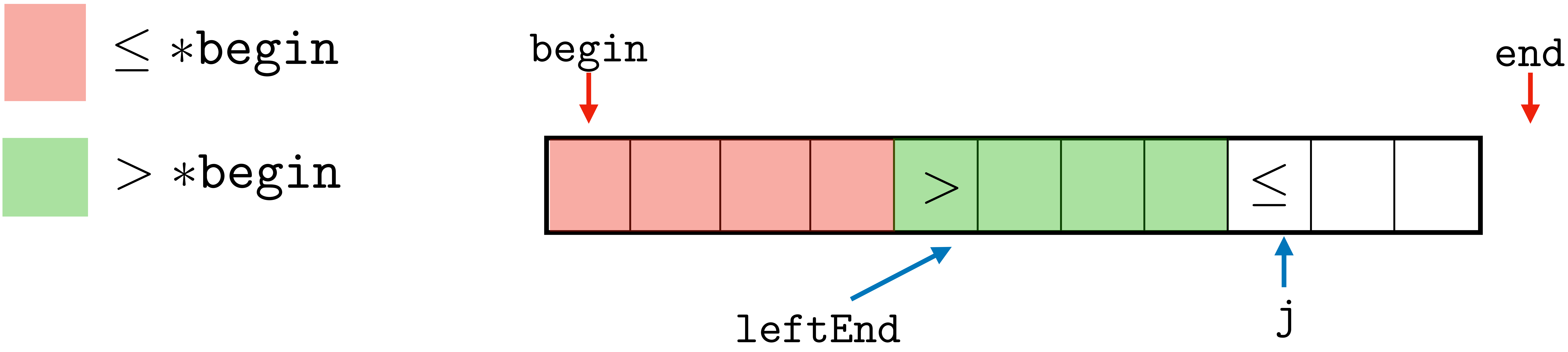

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

The invariant still holds.

Case 1: $*j > *begin$

No swap, no increment of `leftEnd`.

Maintenance



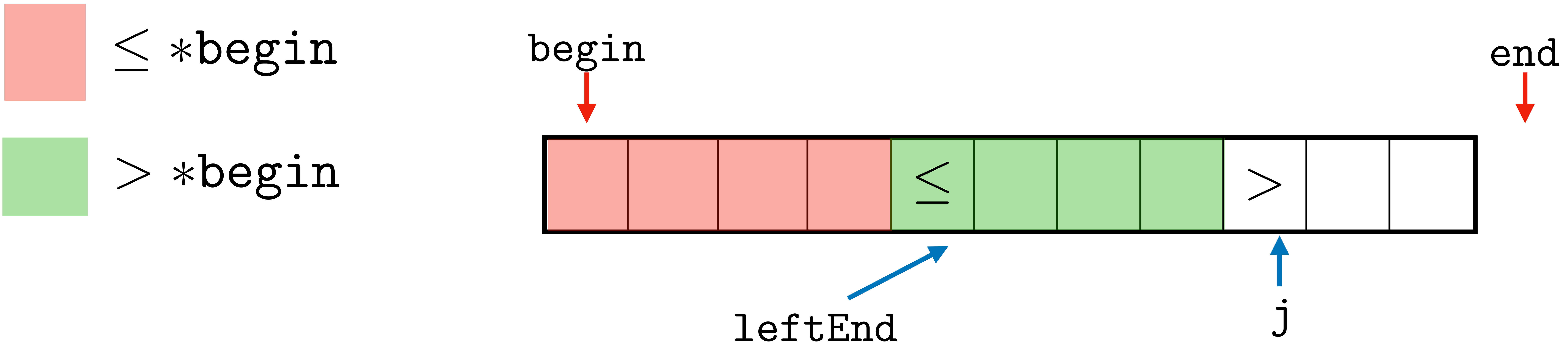

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

In general, there are two cases:

Case 2: $*j \leq *begin$

Swap $*j$ and $*leftEnd$.

Maintenance

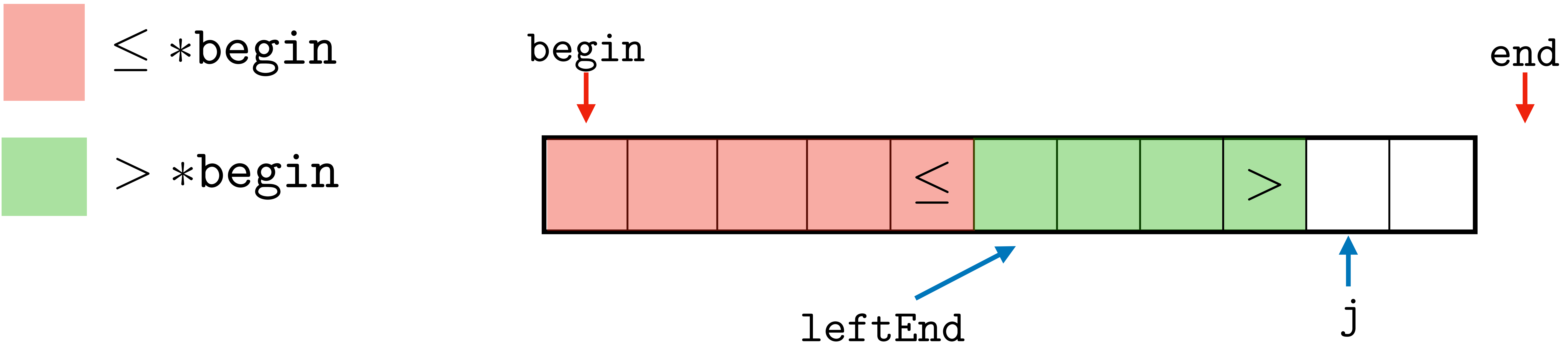



```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

Case 2: $*j \leq *begin$

Increment `leftEnd` and `j`.

Maintenance



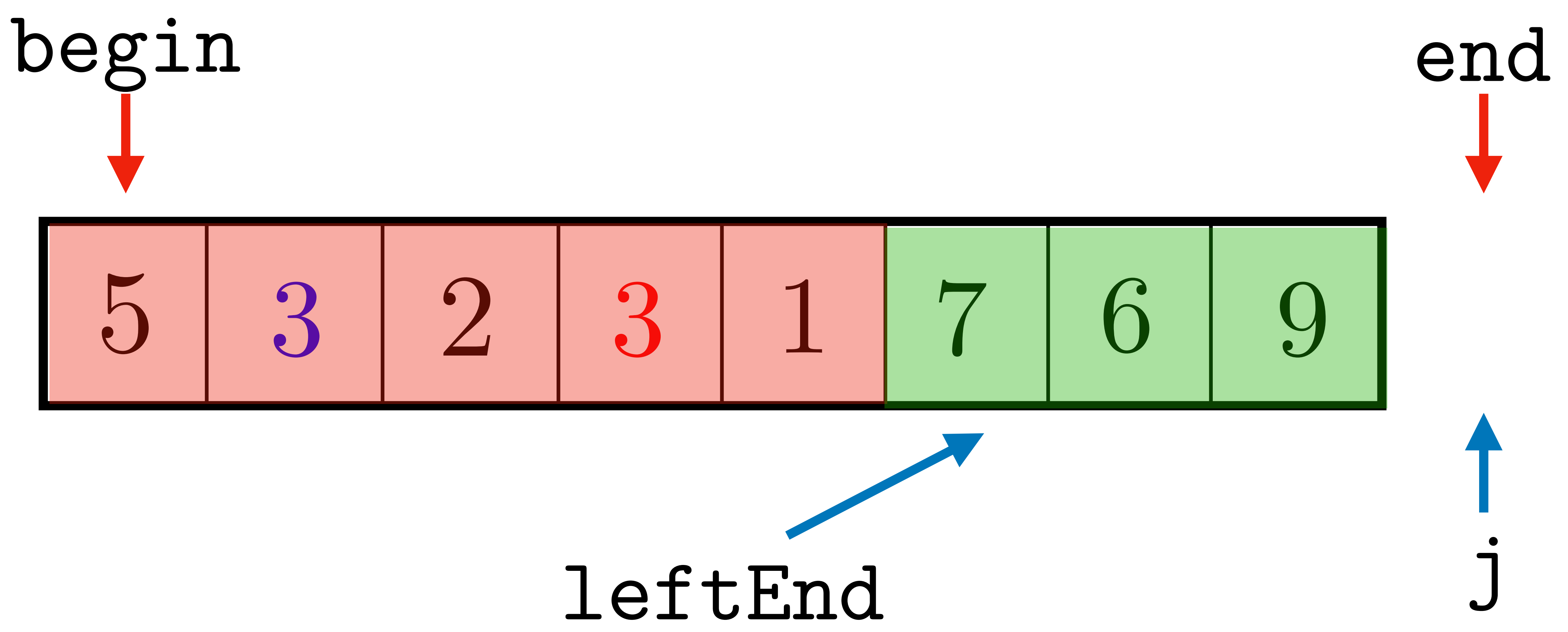

```
for (vecIt j = begin + 1; j < end; ++j) {
  if (*j <= *begin) {
    std::swap(*leftEnd, *j);
    ++leftEnd;
  }
}
```

The invariant still holds.

Case 2: $*j \leq *begin$

Increment `leftEnd` and `j`.

Termination



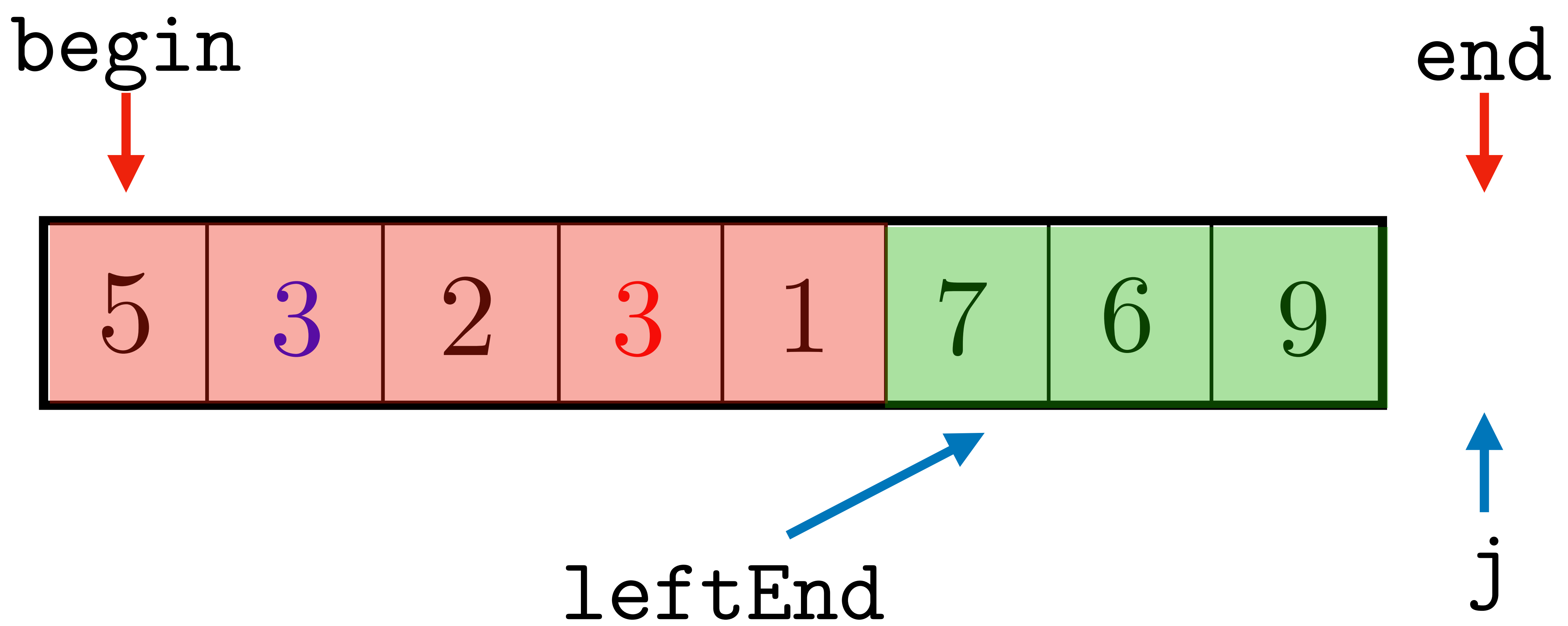
Elements in $[\texttt{begin}, \texttt{leftEnd})$ are at most the pivot.

Elements in $[\texttt{leftEnd}, \texttt{j})$ are greater than the pivot.

Elements in $[\texttt{j}, \texttt{end})$ are still to be processed. Empty 😀.

We have now partitioned the vector.

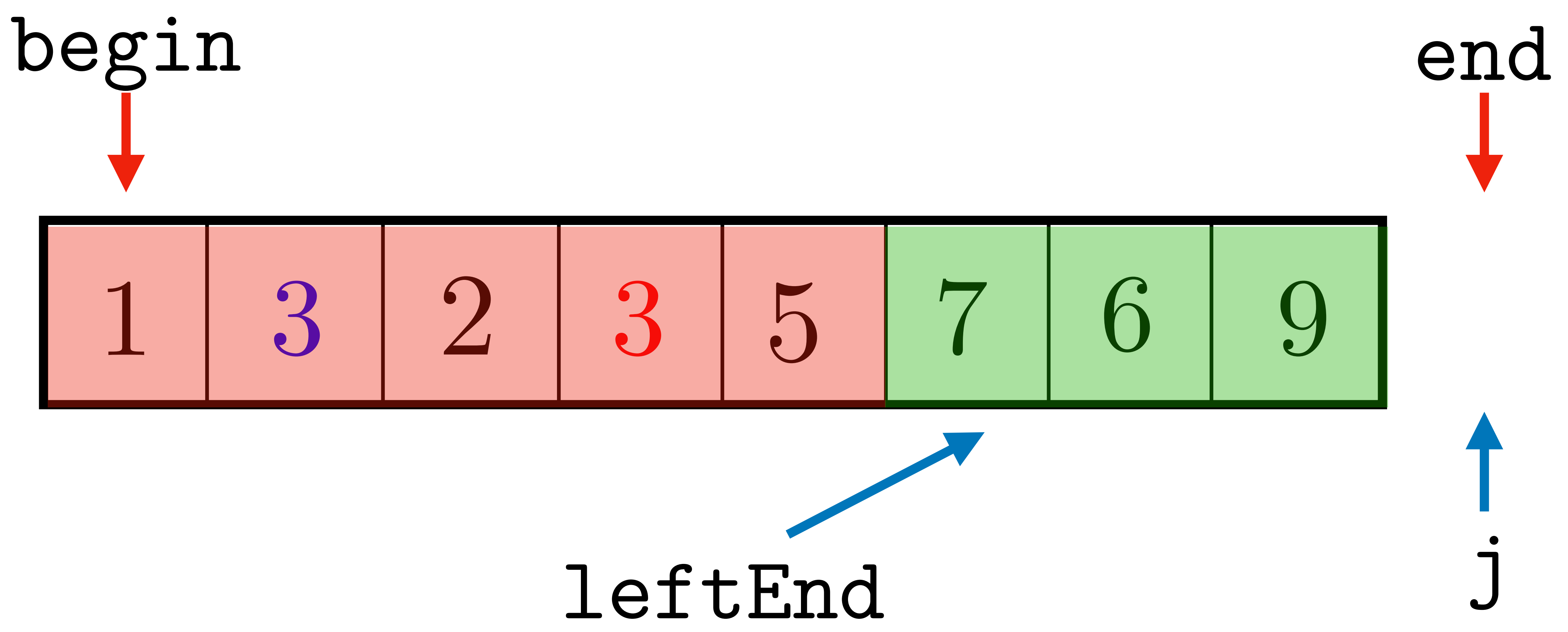
Put Pivot in Its Place




```
std::swap(*begin, *(leftEnd-1));
return leftEnd - 1;
```

If we put the pivot in position $\texttt{leftEnd} - 1$ then it is less than everything to its right, and greater than or equal to everything to its left.

Put Pivot in Its Place




```
std::swap(*begin, *(leftEnd-1));
return leftEnd - 1;
```

This is a valid final position for the pivot in a sorted vector.

We then return the pivot position $\texttt{leftEnd} - 1$.

Running Time

Godbolt Link

```
vecIt lomutoPartition(vecIt begin, vecIt end) {
  vecIt leftEnd = begin + 1;
  for (vecIt j = begin + 1; j < end; ++j) {
    if (*j <= *begin) {
      std::swap(*leftEnd, *j);
      ++leftEnd;
    }
  }
  std::swap(*begin, *(leftEnd-1));
  return leftEnd - 1;
}
```

The body of the for loop does a constant amount of work.

The running time is $\Theta(\texttt{end} - \texttt{begin})$.

This is the bound we used in the previous lecture to argue quicksort has $\Theta(n^2)$ worst-case complexity and $\Theta(n \log n)$ average-case complexity.