CSC230

Intro to C++ Lecture 24

Outline

- Review Sorting
  - Insertion Sort
  - Selection Sort
  - Merge Sort

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 0: step 0.

InsertionSort

```
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i= 1; i < b.length; i= i+1) {
    Push b[i] down to its sorted position
    in b[0..i]
}
```

- Worst-case: O(n^2)
  (reverse-sorted input)
- Best-case: O(n)
  (sorted input)

Pushing b[i] down can take i swaps. Worst case takes

$$1 + 2 + 3 + \dots n-1 = (n-1)*n/2$$

Swaps.

Selection Sort

5	1	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

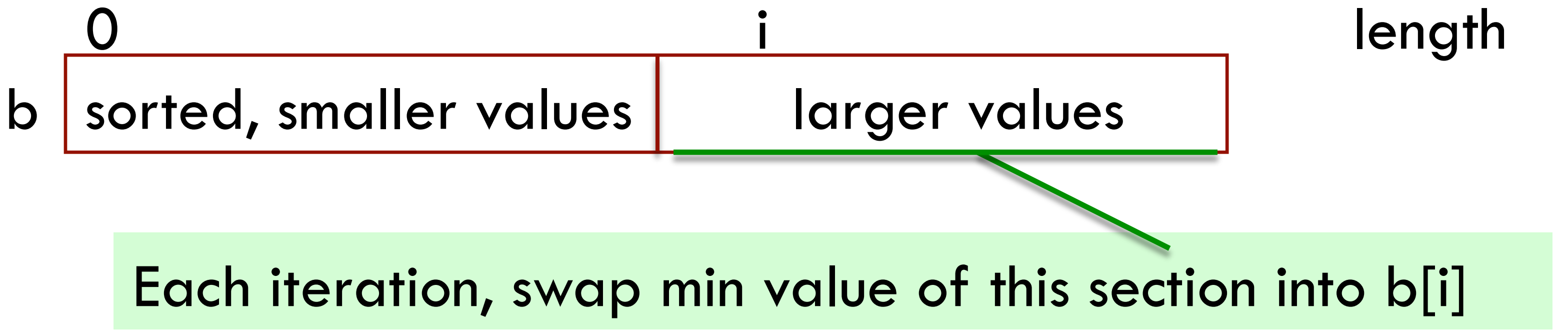
SelectionSort

```
//sort b[], an array of int
// inv: b[0..i-1] sorted
//        b[0..i-1]  <=  b[i..]
for (int i= 1; i < length; i= i+1) {
   int m= index of minimum of b[i..];
   Swap b[i] and b[m];
}
```

Another common way for people to sort cards

Runtime

- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$



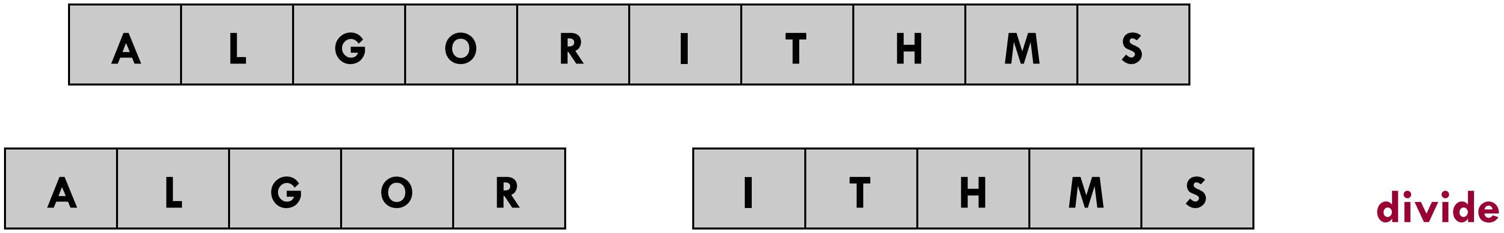
0 i length

b sorted, smaller values | larger values

Each iteration, swap min value of this section into b[i]

Mergesort

- Mergesort (divide-and-conquer)
  - Divide array into two halves.



A	L	G	O	R	I	T	H	M	S

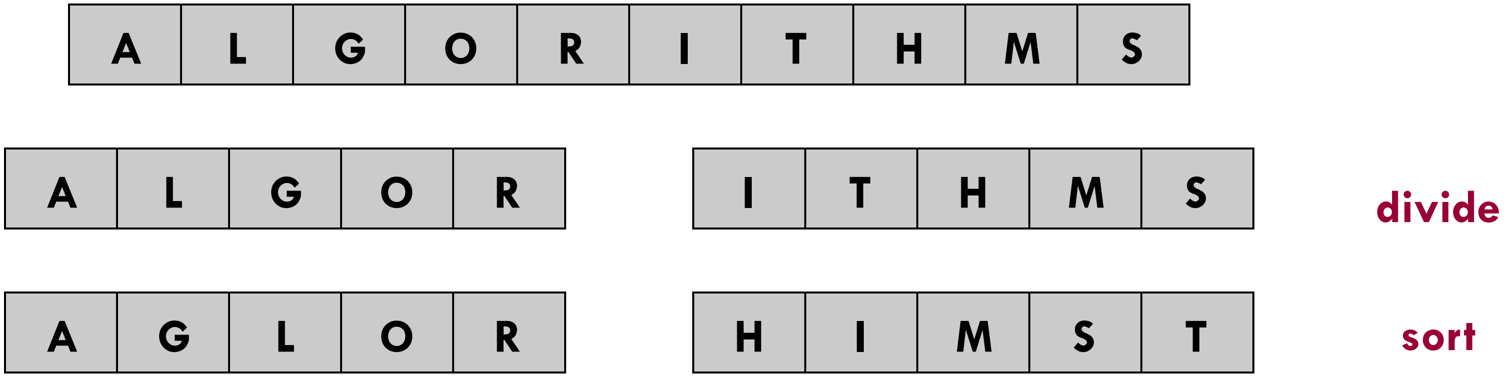
A	L	G	O	R

I	T	H	M	S

divide

Mergesort

- Mergesort (divide-and-conquer)
  - Divide array into two halves.
  - Recursively sort each half.



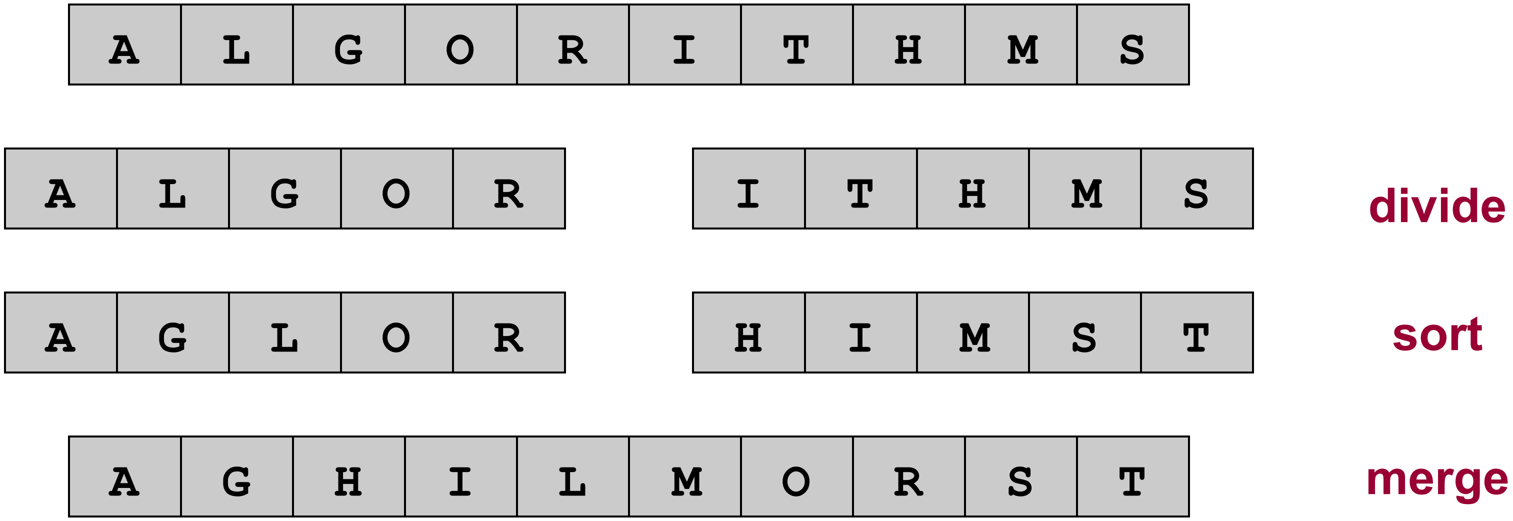
A L G O R I T H M S

A L G O R   I T H M S   divide

A G L O R   H I M S T   sort

Mergesort

- Mergesort (divide-and-conquer)
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.



A L G O R I T H M S

A L G O R   I T H M S   divide

A G L O R   H I M S T   sort

A G H I L M O R S T   merge

How to Merge

Here are two lists to be merged:

First: **(12, 16, 17, 20, 21, 27)**

Second: **(9, 10, 11, 12, 19)**

Compare**12** and **9**

First: **(12, 16, 17, 20, 21, 27)**

Second: **(10, 11, 12, 19)**

New: **(9)**

Compare **12** and **10**

First: **(12, 16, 17, 20, 21, 27)**

Second: **(11, 12, 19)**

New: **(9, 10)**

Merge Example

Compare **12** and **11**

- **First:** **(12, 16, 17, 20, 21, 27)**
- **Second:** **(12, 19)**
- **New:** **(9, 10, 11)**

Compare **12** and **12**

- **First:** **(16, 17, 20, 21, 27)**
- **Second:** **(12, 19)**
- **New:** **(9, 10, 11, 12)**

Merge Example

Compare **16** and **12**

First:	**(16, 17, 20, 21, 27)**
Second:	**(19)**
New:	**(9, 10, 11, 12, 12)**

Compare **16** and **19**

First:	**(17, 20, 21, 27)**
Second:	**(19)**
New:	**(9, 10, 11, 12, 12, 16)**

Merge Example

Compare **17** and **19**

First: **(20, 21, 27)**

Second: **(19)**

New: **(9, 10, 11, 12, 12, 16, 17)**

Compare **20** and **19**

First: **(20, 21, 27)**

Second: **( )**

New: **(9, 10, 11, 12, 12, 16, 17, 19)**

Merge Example

Checkout **20** and **empty list**

First:( )

Second: **( )**

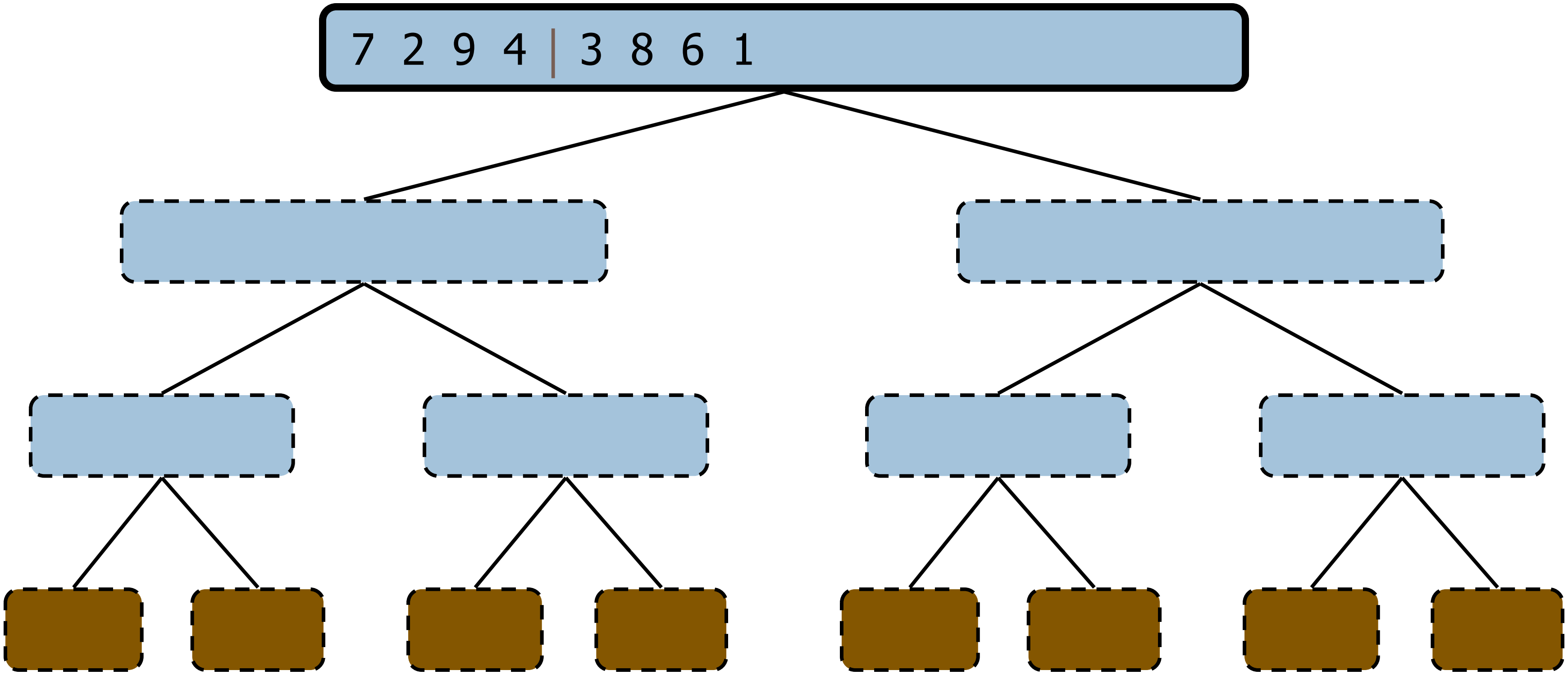
New: **(9, 10, 11, 12, 12, 16, 17, 19, 20, 21, 27)**

Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1

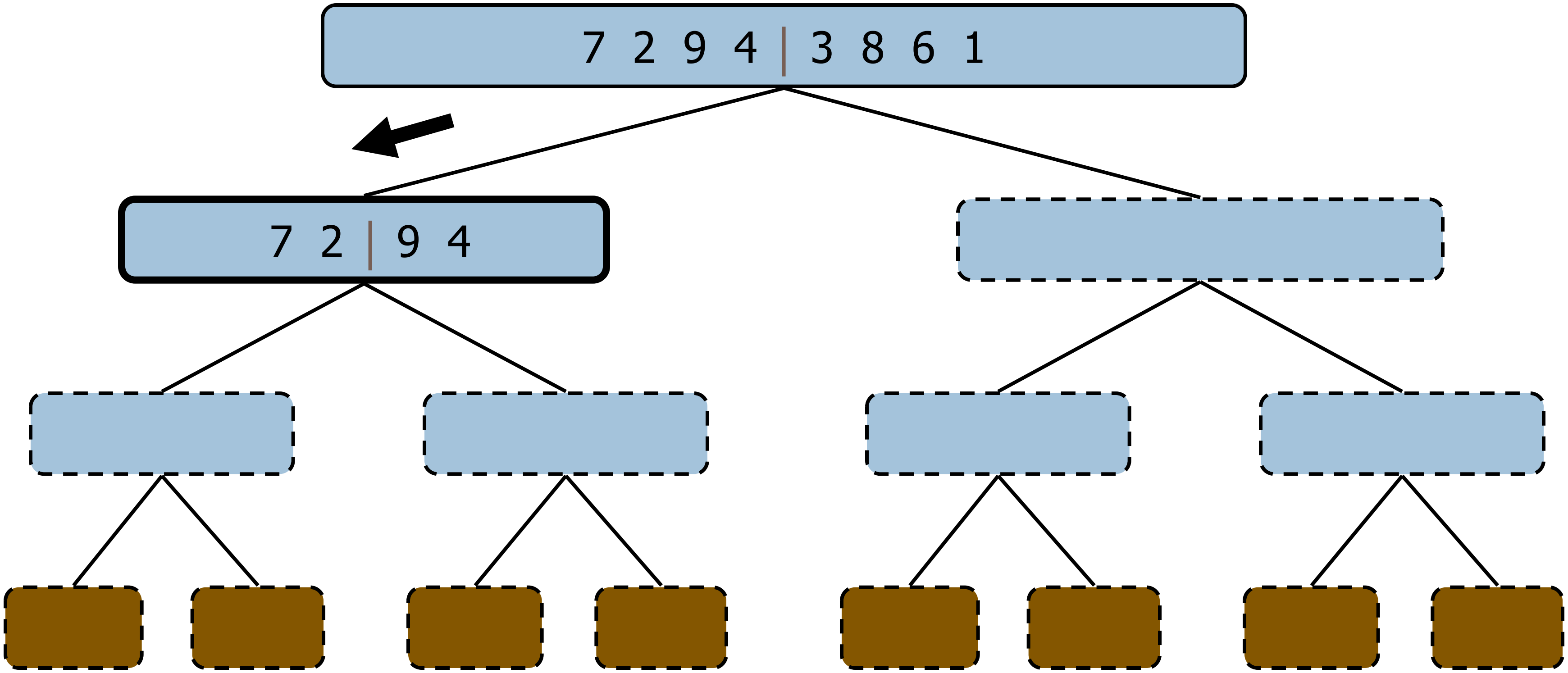
Execution Example

- Partition



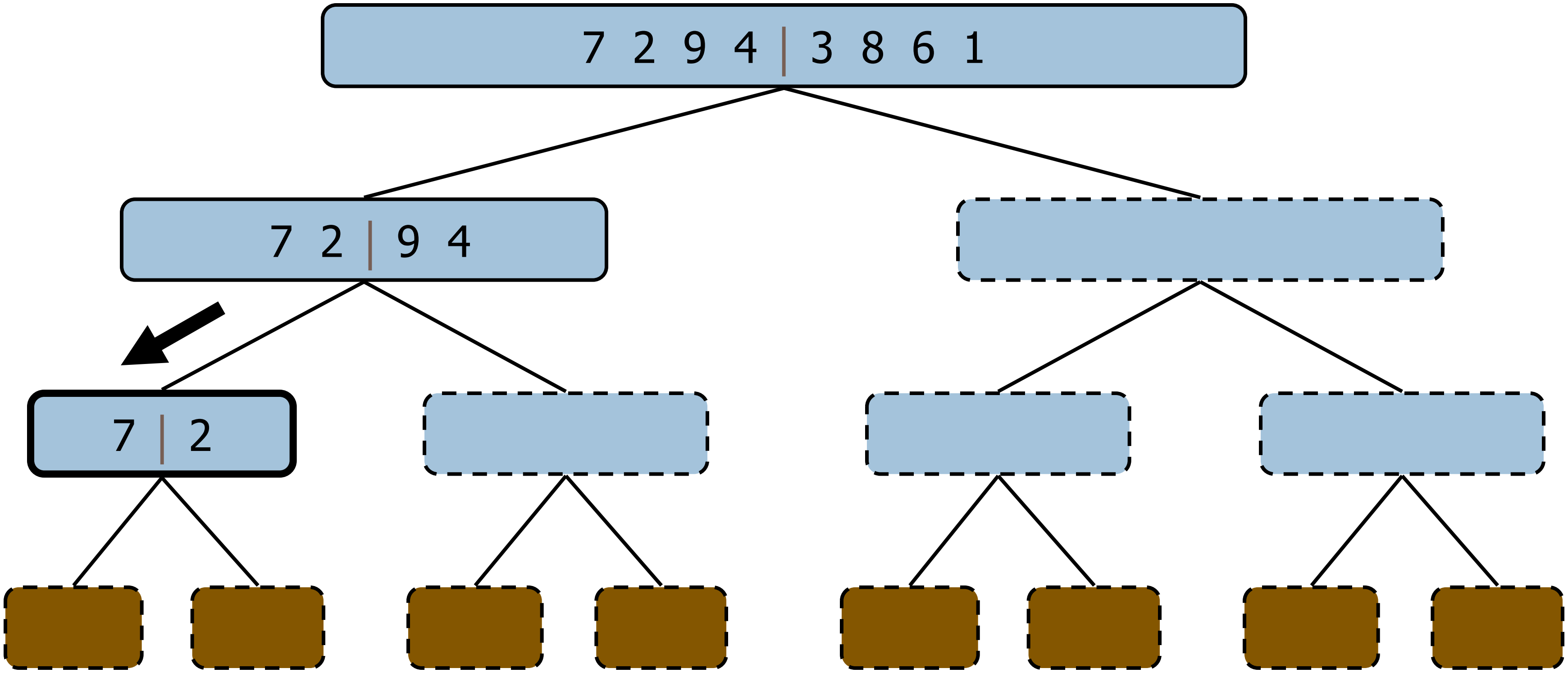
Execution Example (cont.)

- Recursive call, partition



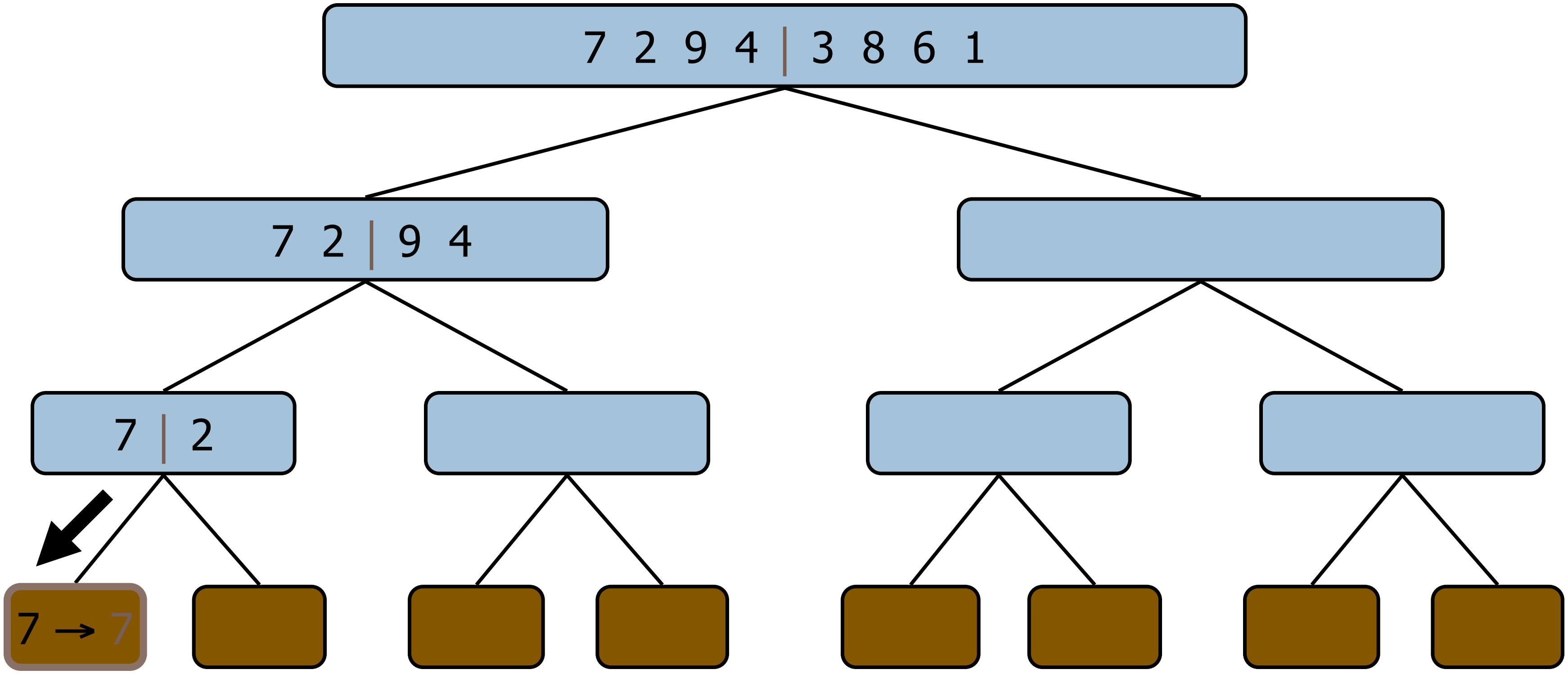
Execution Example (cont.)

- Recursive call, partition



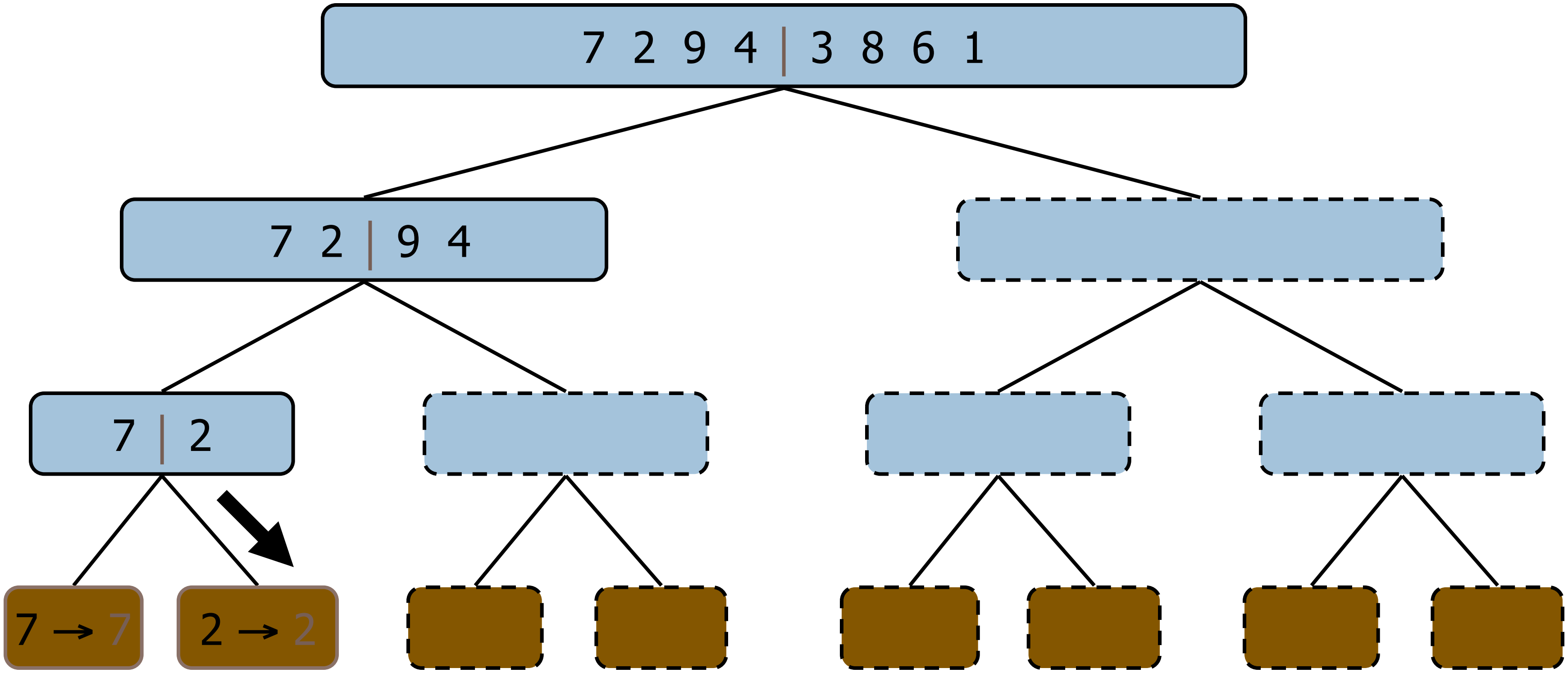
Execution Example (cont.)

- Recursive call, base case



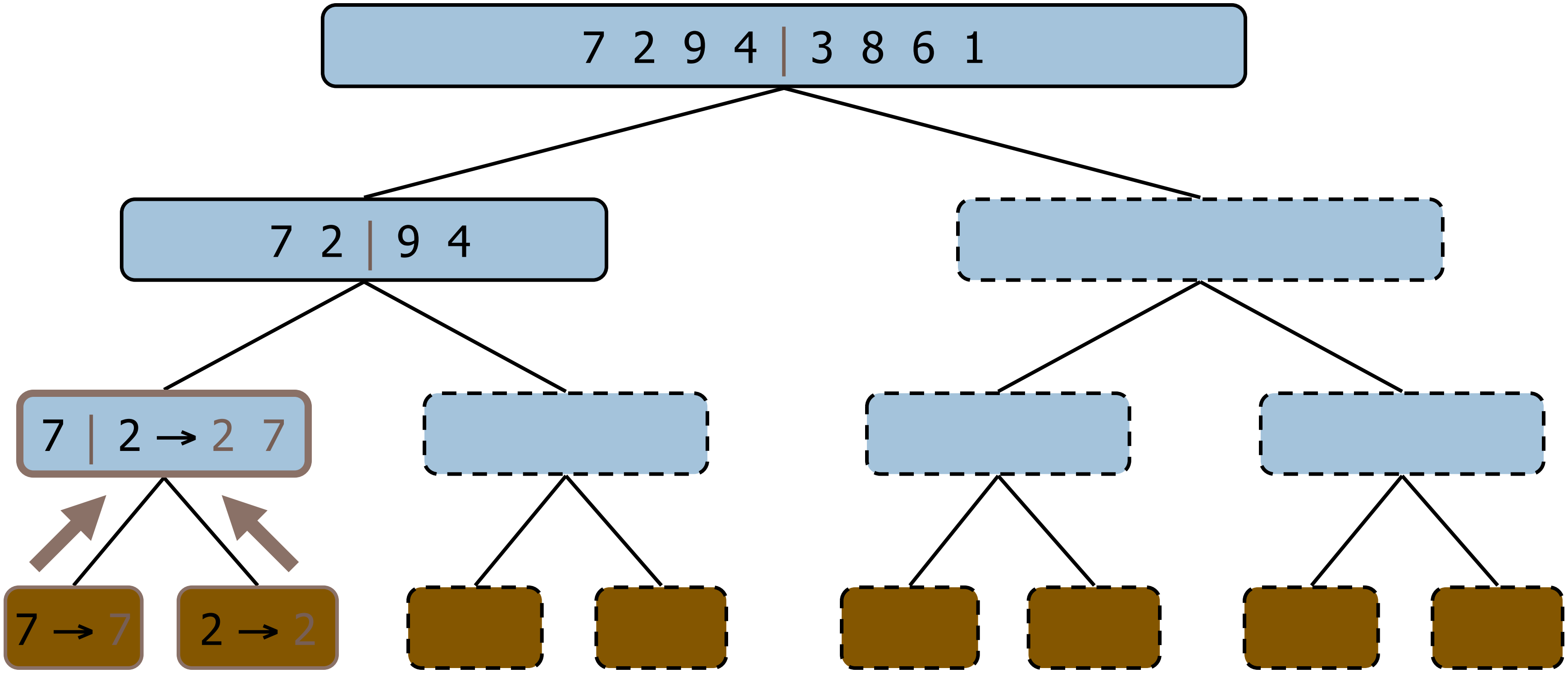
Execution Example (cont.)

- Recursive call, base case



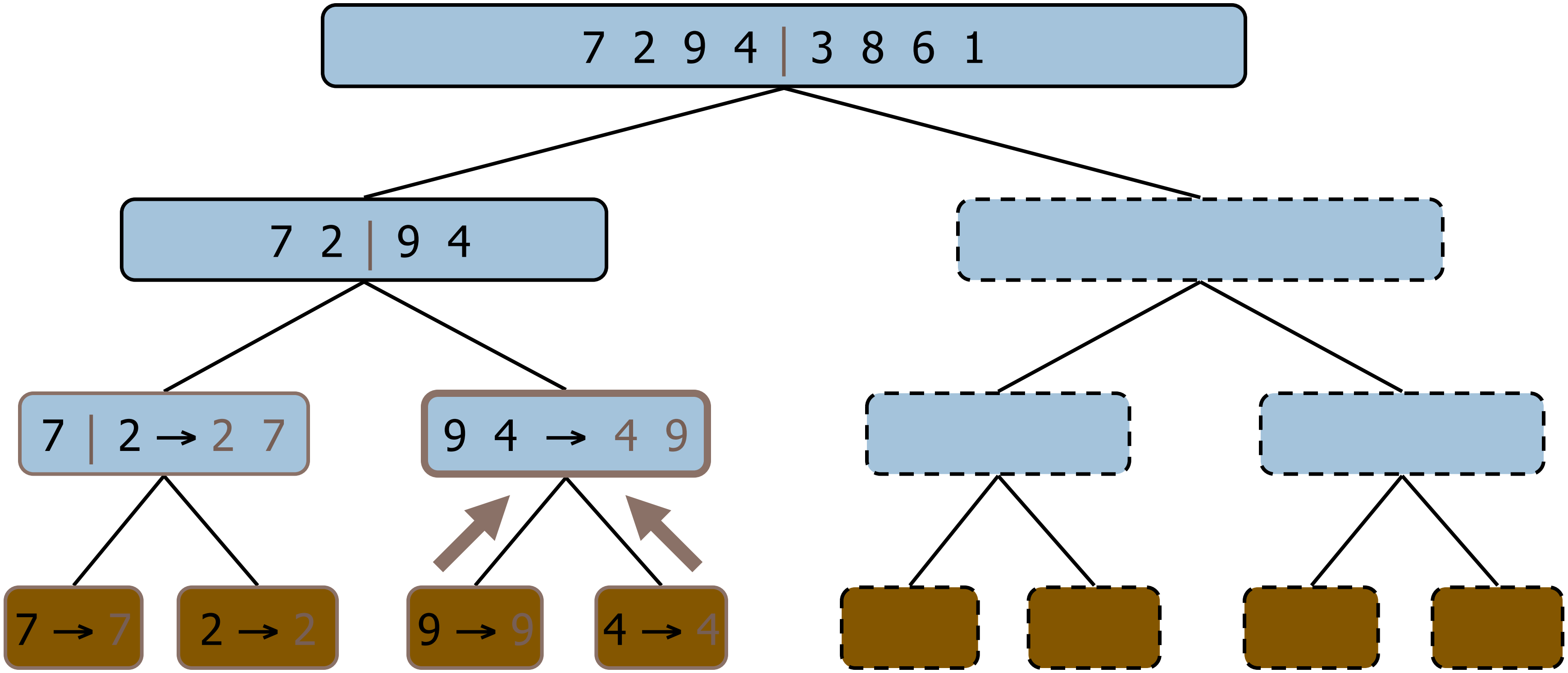
Execution Example (cont.)

- Merge



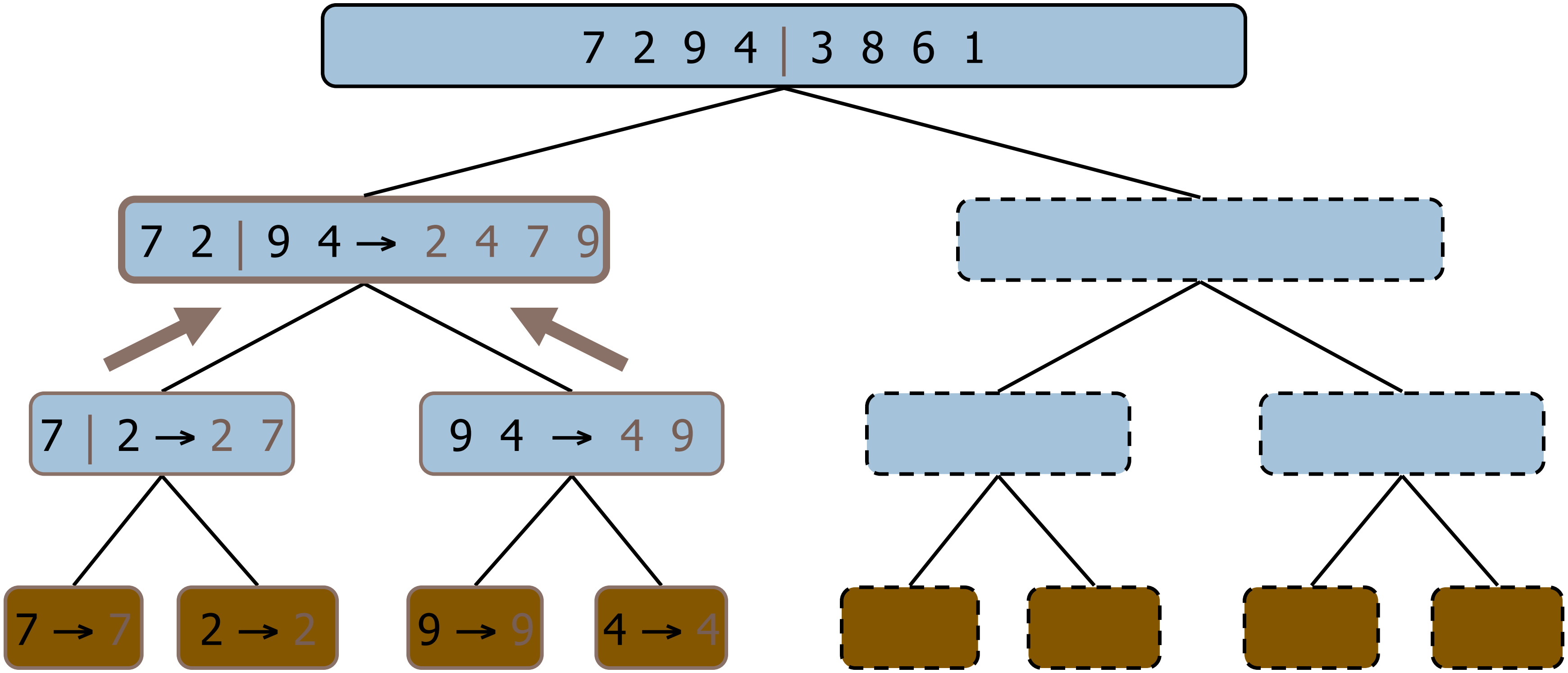
Execution Example (cont.)

- Recursive call, …, base case, merge



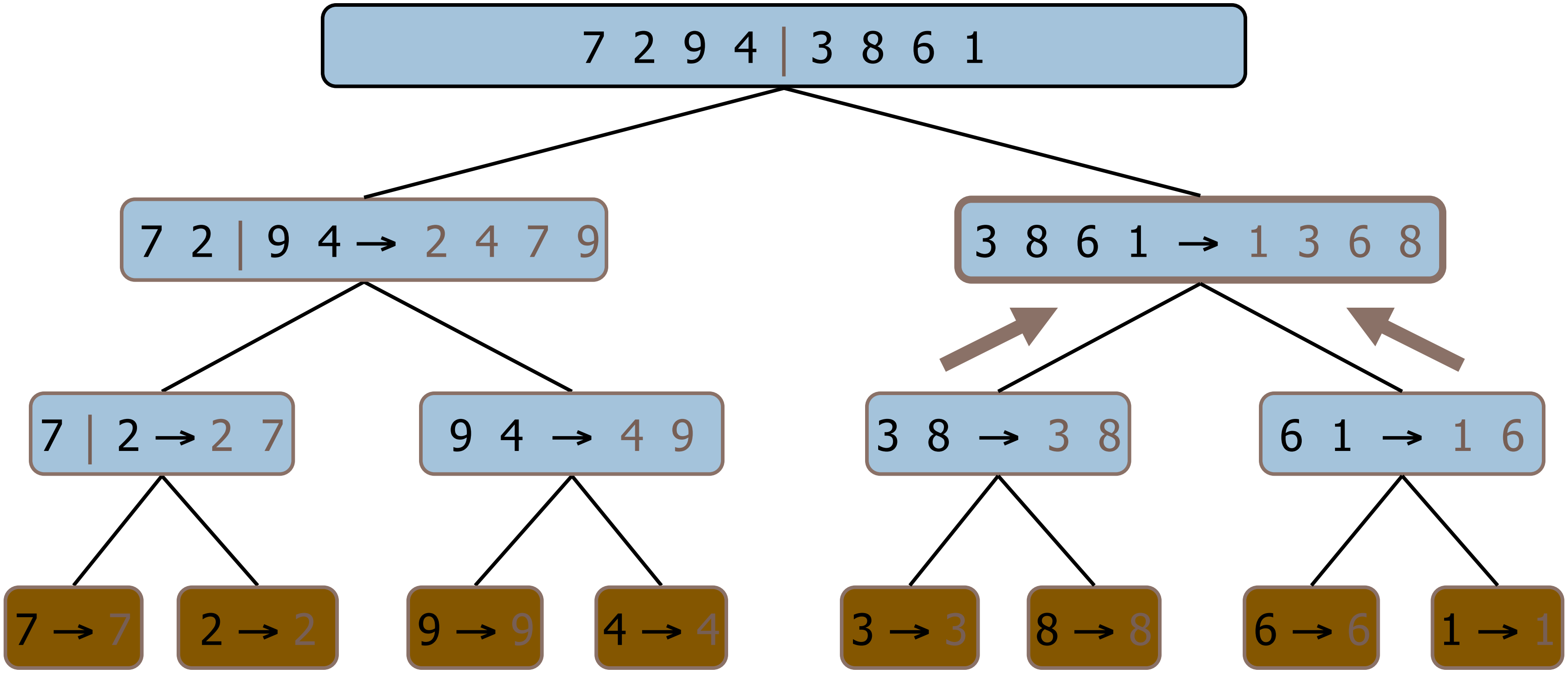
Execution Example (cont.)

- Merge



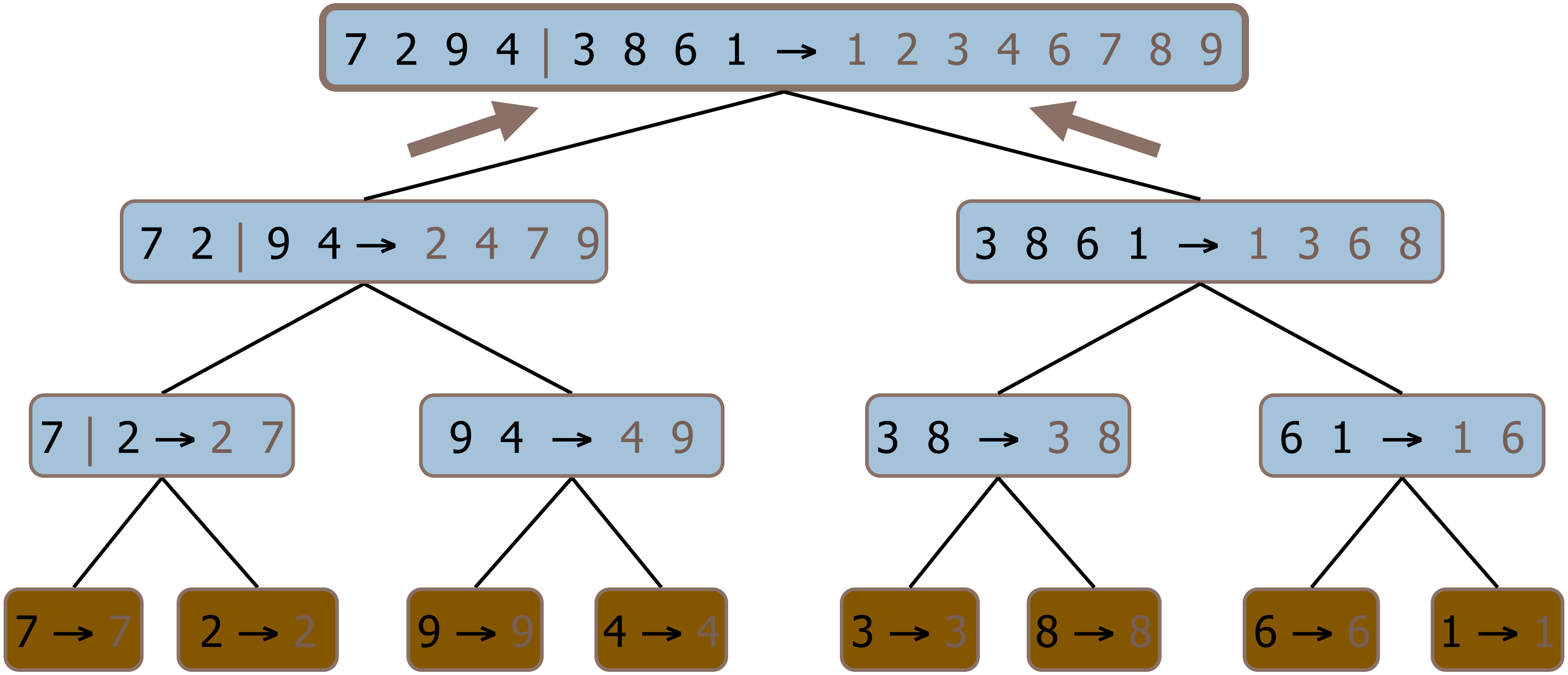
Execution Example (cont.)

- Recursive call, …, merge, merge

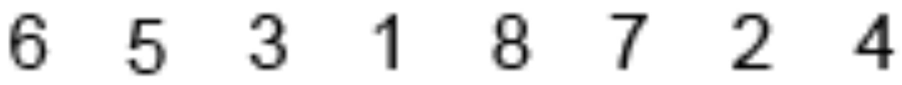


Execution Example (cont.)

- Merge



Implementing Mergesort



6 5 3 1 8 7 2 4

Merge Sort

- Apply divide-and-conquer to sorting problem
- Problem: Given *n* elements, sort elements into non -decreasing order
- Divide-and-Conquer:
  - If n=1 terminate (every one-element list is already sorted)
  - If n>1, partition elements into two or more sub-collections; sort each; combine into a single sorted list

Implementing Merge

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = $(l+r)/2$
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

- EXAMPLE: MergeSort.cpp

Implementing Mergesort

Mergesort

```
Item aux[MAXN];
```

uses scratch array

```
void mergesort(Item a[], int left, int right) {
   int mid = (right + left) / 2;
   if (right <= left)
      return;
   mergesort(a, left, mid);
   mergesort(a, mid + 1, right);
   merge(a, left, mid, right);
}
```

Outline

- □ *Merge Sort*

- □ Quicksort Algorithm

Partitioning - Choice 1

- First n-1 elements into set A, last element set B
- Sort A using this partitioning scheme recursively
  - B already sorted
- Combine A and B using method Insert() (= insertion into sorted array)
- Leads to recursive version of InsertionSort()
  - Number of comparisons: O(n^2)
    - Best case = n-1
    - Worst case = $c\sum_{i=2}^{n} i = \frac{n(n-1)}{2}$

Partitioning - Choice 2

- Put element with largest key in B, remaining elements in A
- Sort A recursively
- To combine sorted A and B, append B to sorted A
  - Use Max() to find largest element → recursive SelectionSort()
  - Use bubbling process to find and move largest element to right-most position → recursive BubbleSort()
- All $O(n^2)$

Partitioning - Choice 3

- Let' s try to achieve balanced partitioning
- A gets $n/2$ elements, B gets rest half
- Sort A and B recursively
- Combine sorted A and B using a process called *merge*, which combines two sorted lists into one
  - How? We will see soon

Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

Partitioning Array

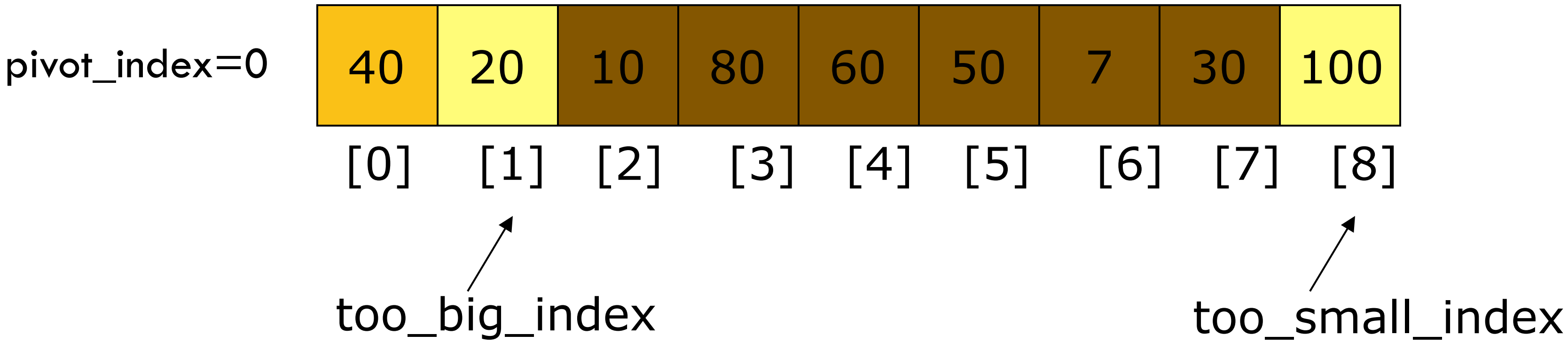
Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below /above pivot.

Quick Sort



pivot_index=0

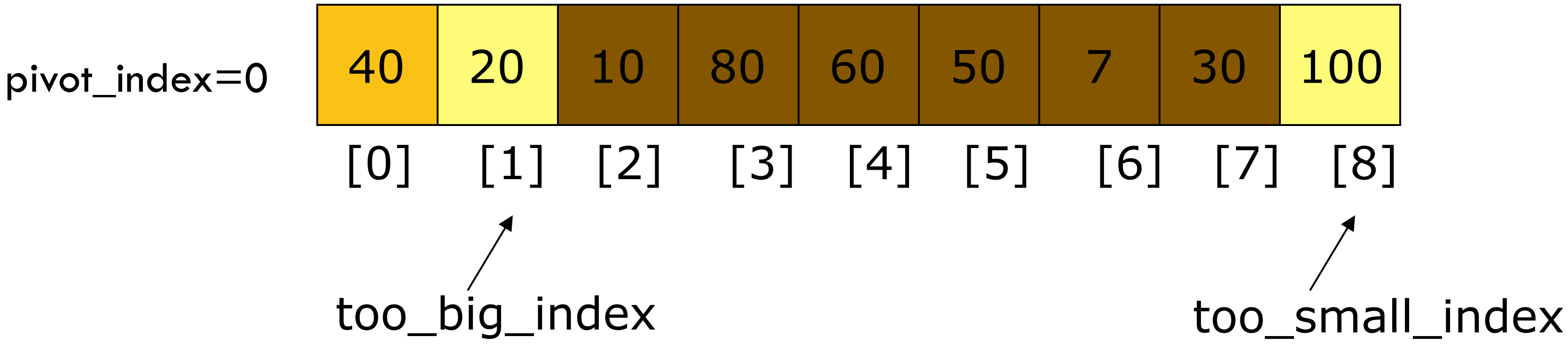
40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
```

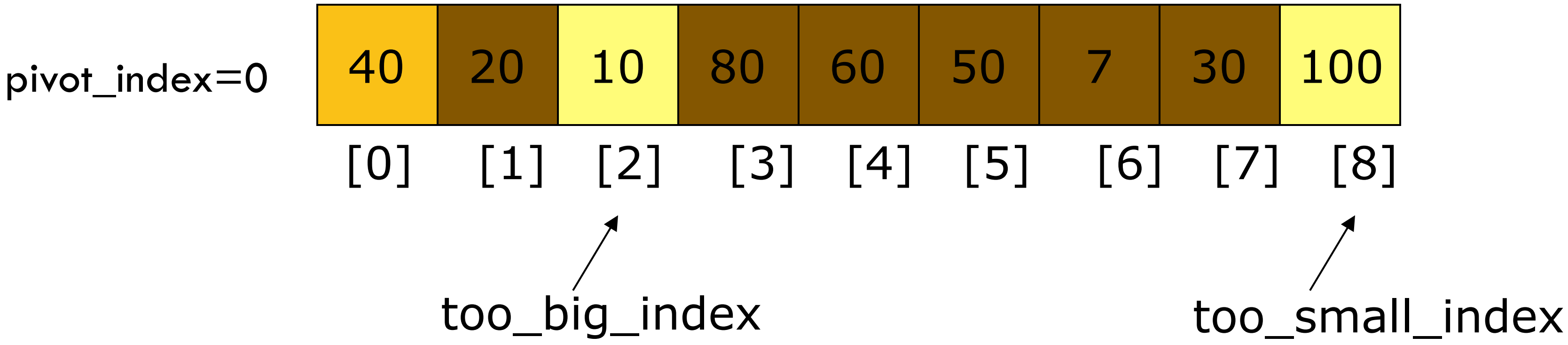


pivot_index=0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
```

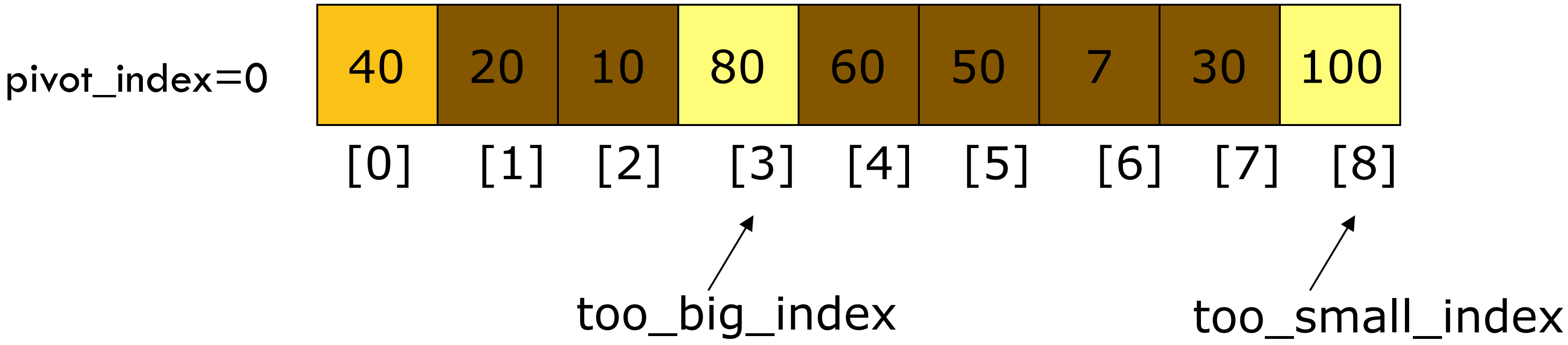


pivot_index=0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
```



pivot_index=0

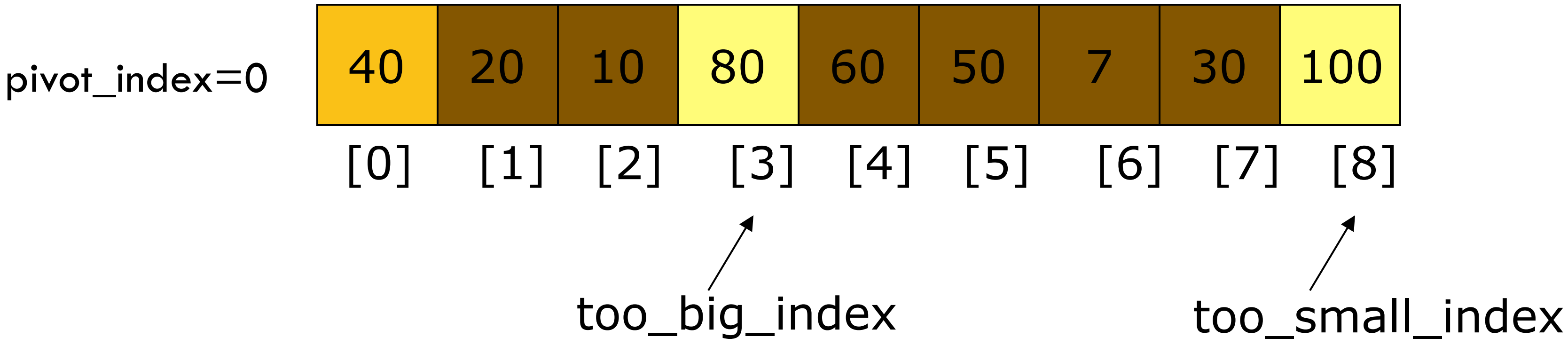
40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
```



pivot_index=0

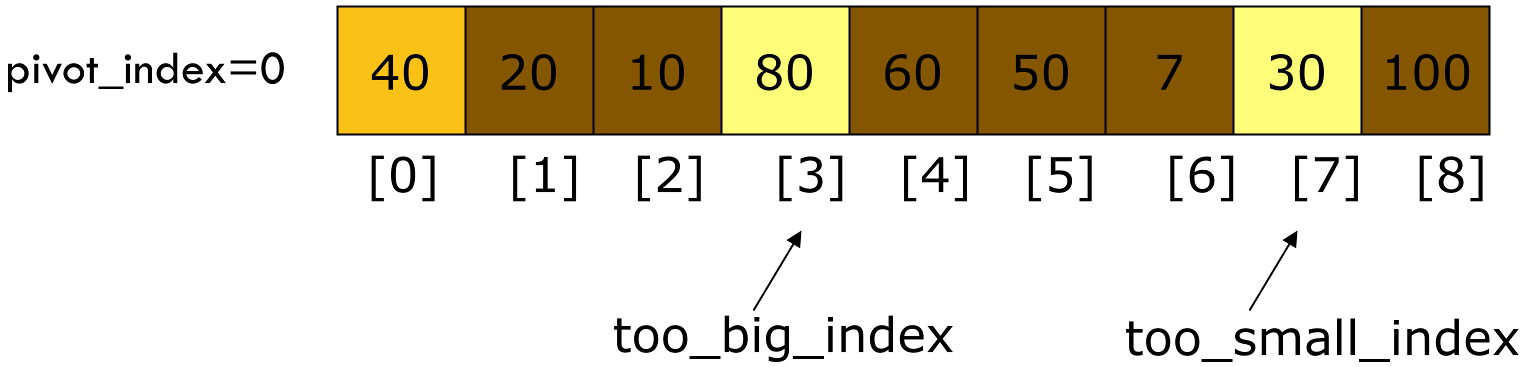
40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
```



pivot_index=0

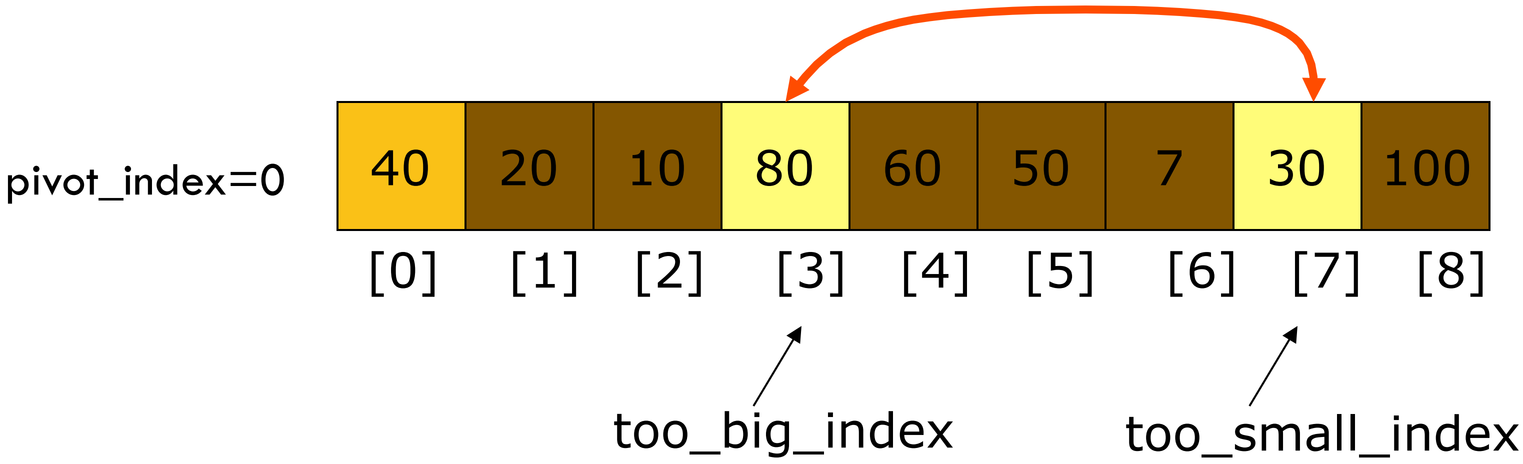
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	80	60	50	7	30	100

too_big_index

too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
```

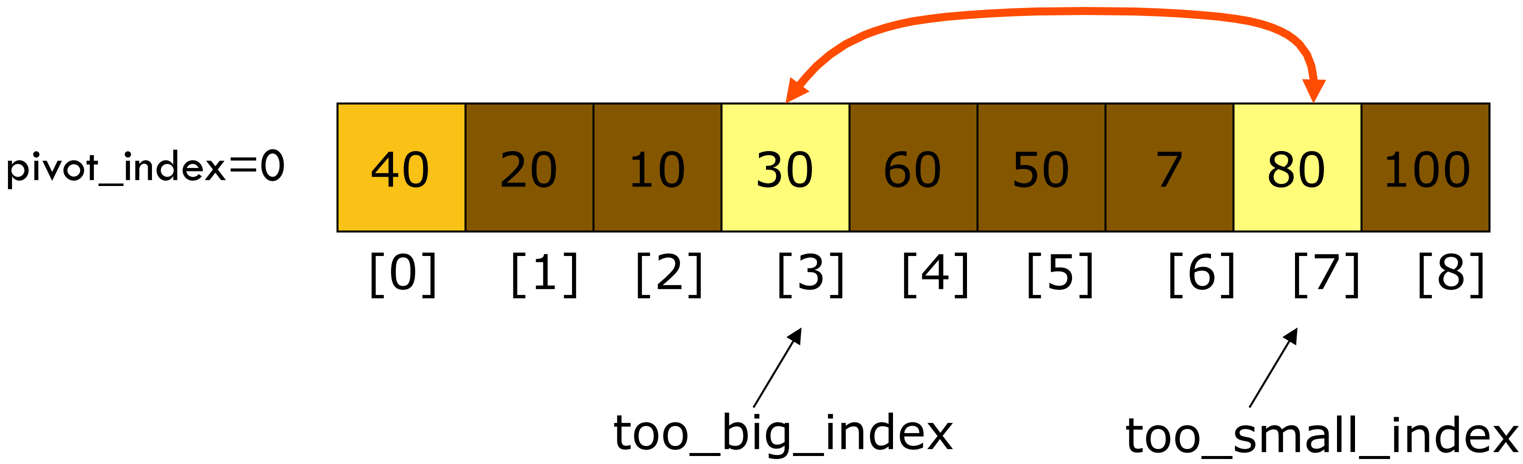


pivot_index=0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
```



pivot_index=0

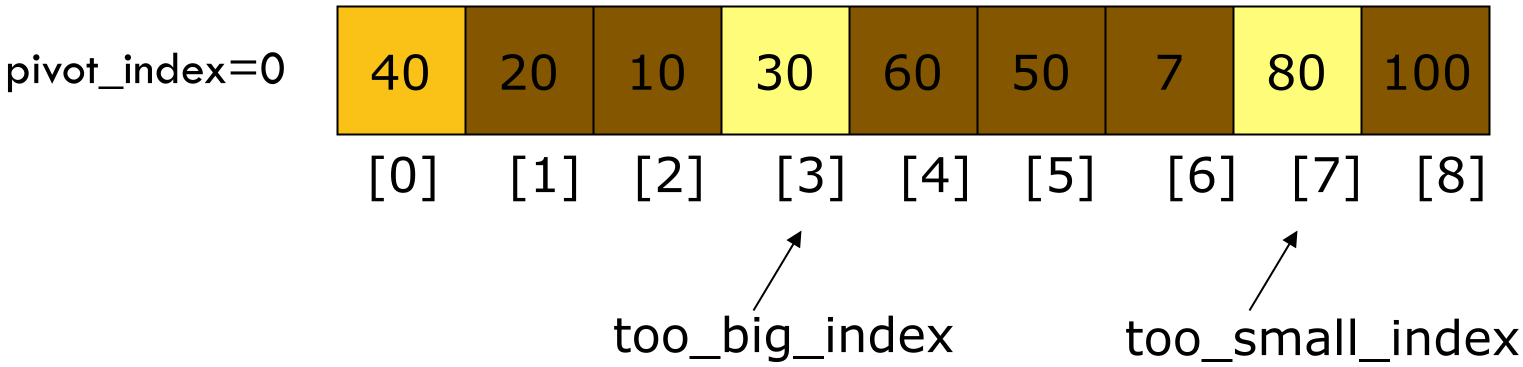
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

too_big_index

too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
```



pivot_index=0

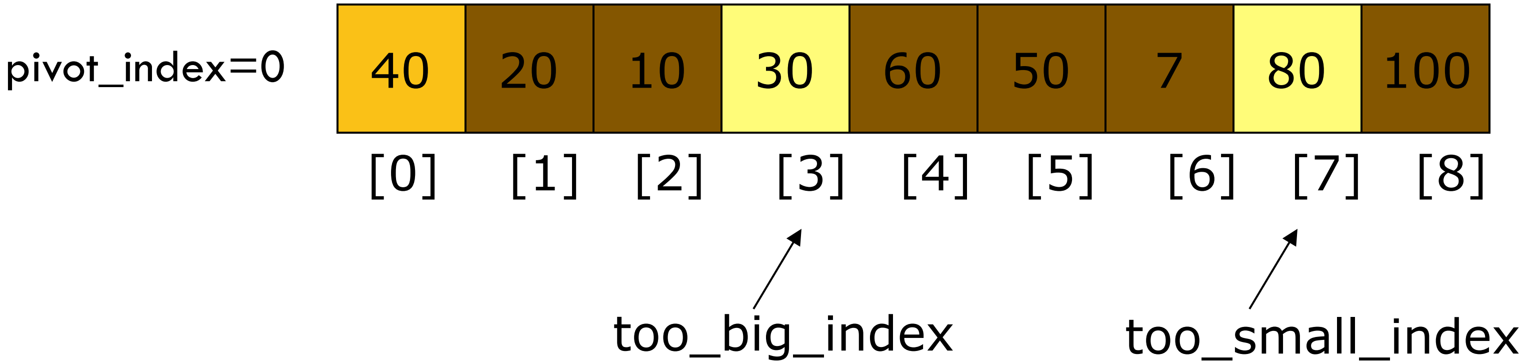
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

too_big_index

too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
```



pivot_index=0

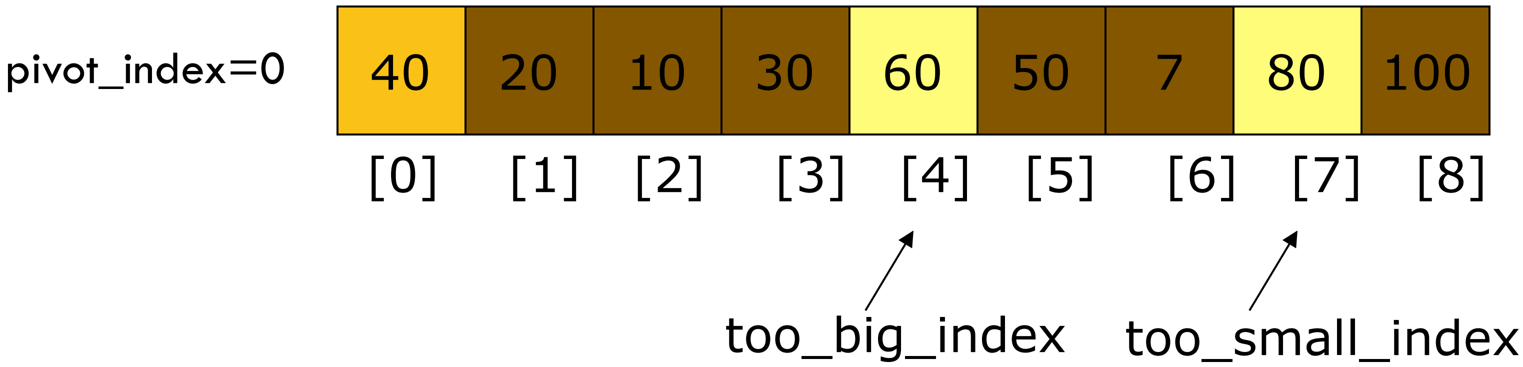
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

too_big_index

too_small_index

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  While too_small_index > too_big_index, go to 1.
```



pivot_index=0

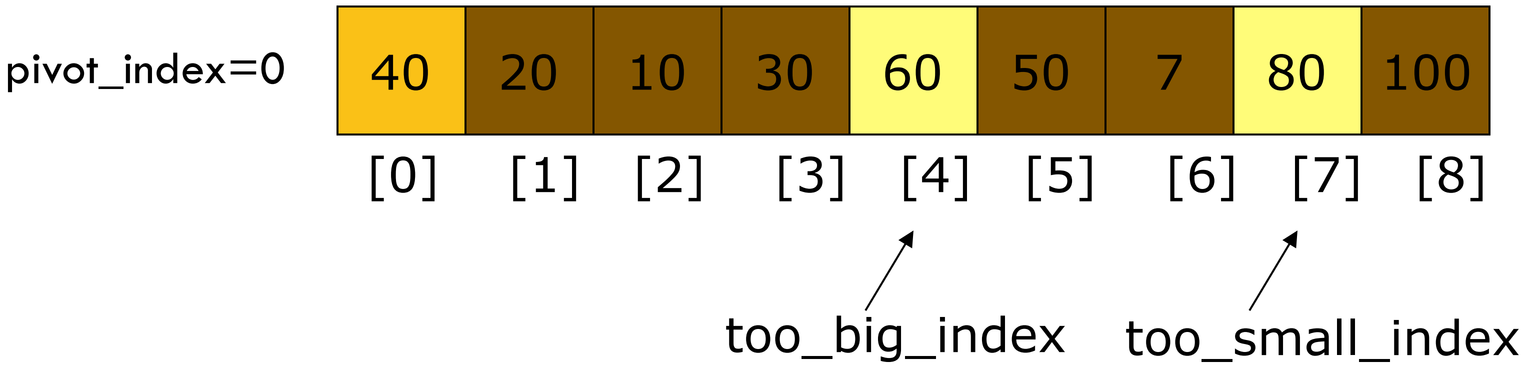
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

too_big_index

too_small_index

Quick Sort

```
   1. While data[too_big_index] <= data[pivot]
          ++too_big_index
→  2. While data[too_small_index] > data[pivot]
          --too_small_index
   3. If too_big_index < too_small_index
          swap data[too_big_index] and data[too_small_index]
   4. While too_small_index > too_big_index, go to 1.
```

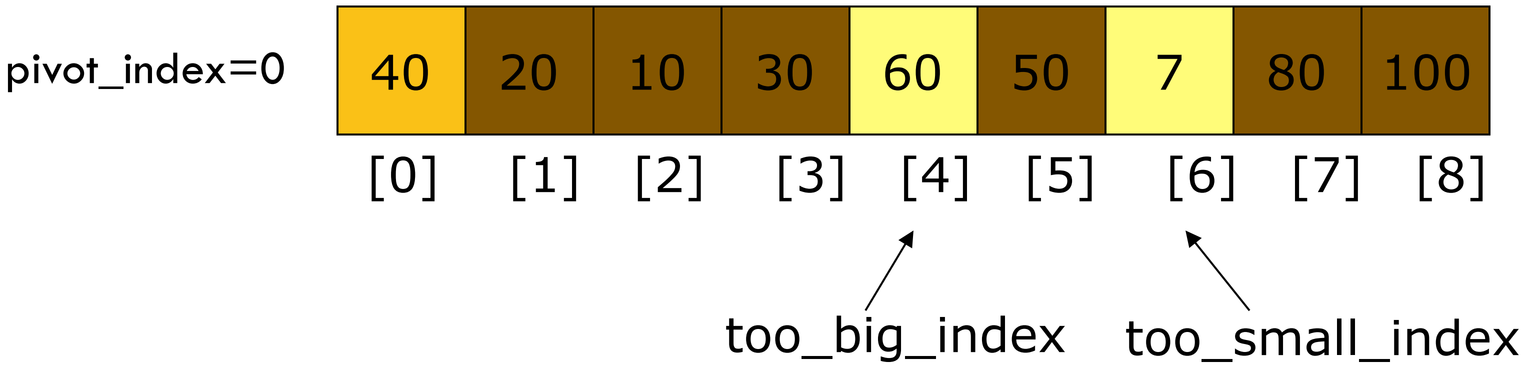


pivot_index=0

40	20	10	30	60	50	7	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Quick Sort

```
  1. While data[too_big_index] <= data[pivot]
         ++too_big_index
→ 2. While data[too_small_index] > data[pivot]
         --too_small_index
  3. If too_big_index < too_small_index
         swap data[too_big_index] and data[too_small_index]
  4. While too_small_index > too_big_index, go to 1.
```

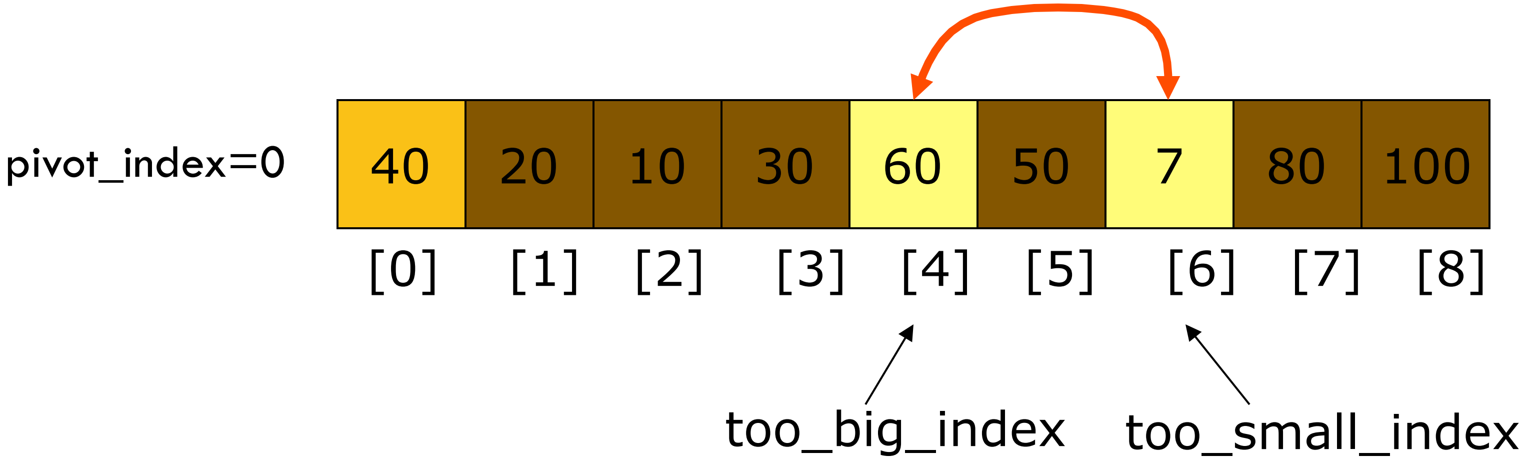


pivot_index=0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

Quick Sort

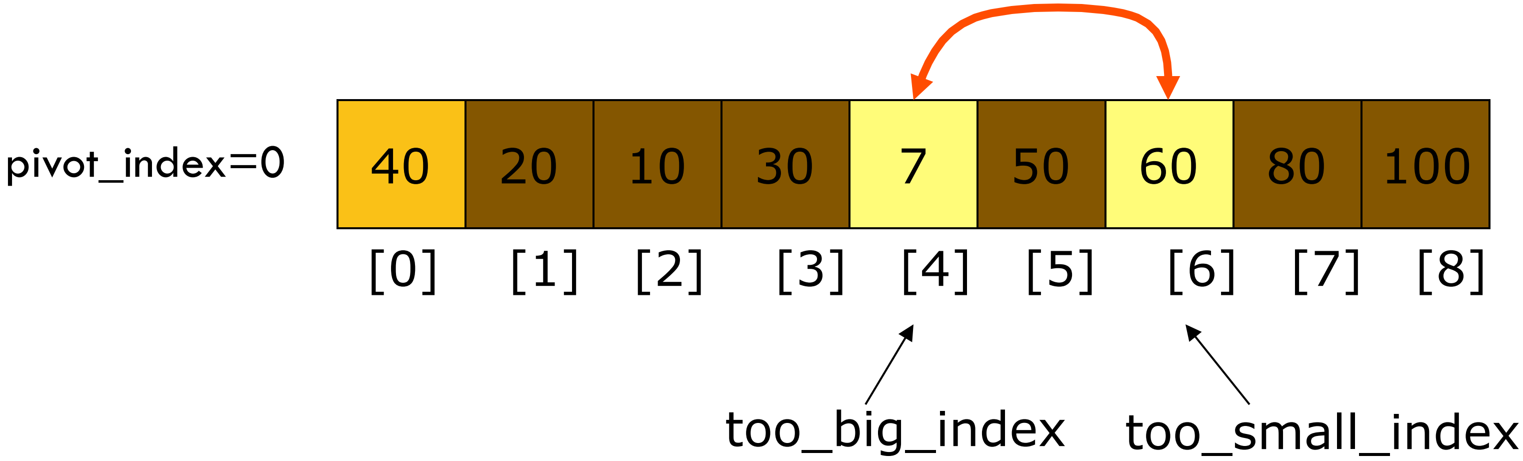
```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
```



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

Quick Sort

```
   1. While data[too_big_index] <= data[pivot]
          ++too_big_index
   2. While data[too_small_index] > data[pivot]
          --too_small_index
→  3. If too_big_index < too_small_index
          swap data[too_big_index] and data[too_small_index]
   4. While too_small_index > too_big_index, go to 1.
```



pivot_index=0

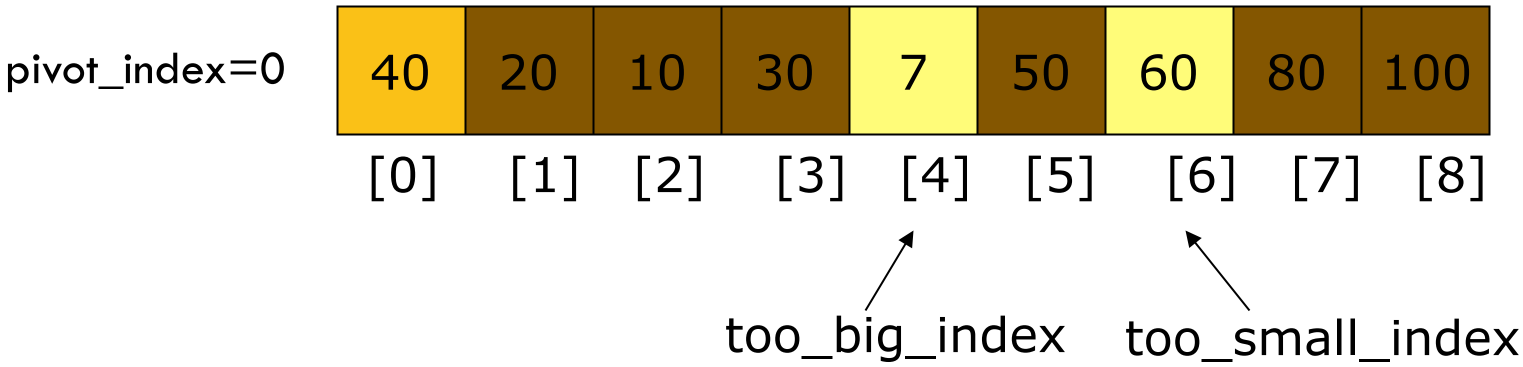
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index

too_small_index

Quick Sort

```
  1. While data[too_big_index] <= data[pivot]
         ++too_big_index
  2. While data[too_small_index] > data[pivot]
         --too_small_index
  3. If too_big_index < too_small_index
         swap data[too_big_index] and data[too_small_index]
→ 4. While too_small_index > too_big_index, go to 1.
```

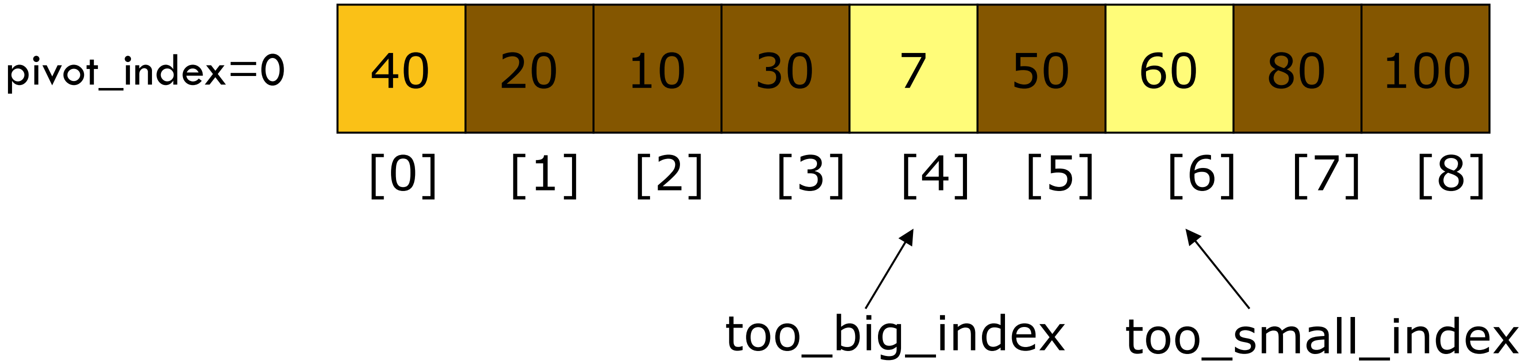


pivot_index=0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  While too_small_index > too_big_index, go to 1.
```

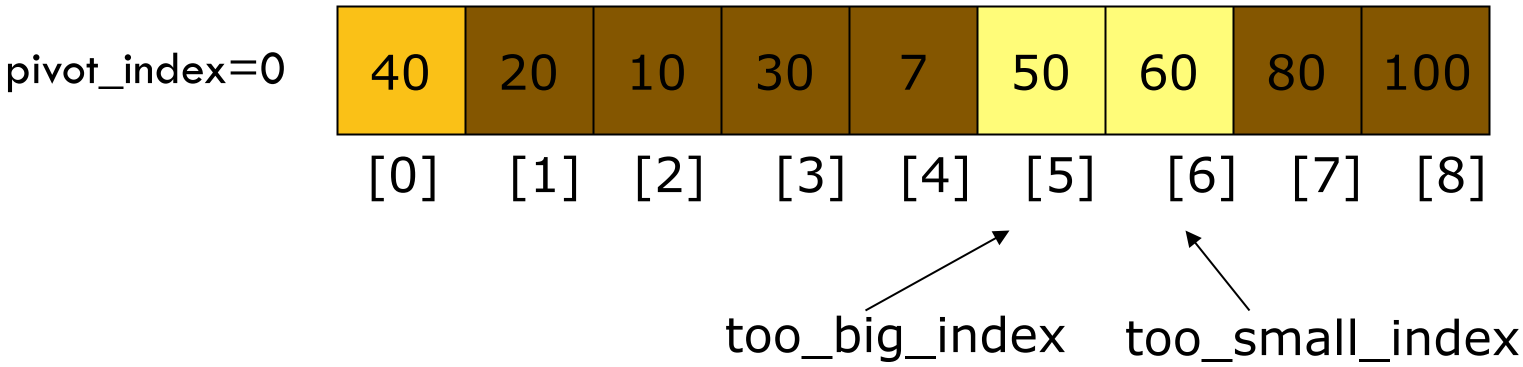


pivot_index=0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  While too_small_index > too_big_index, go to 1.
```

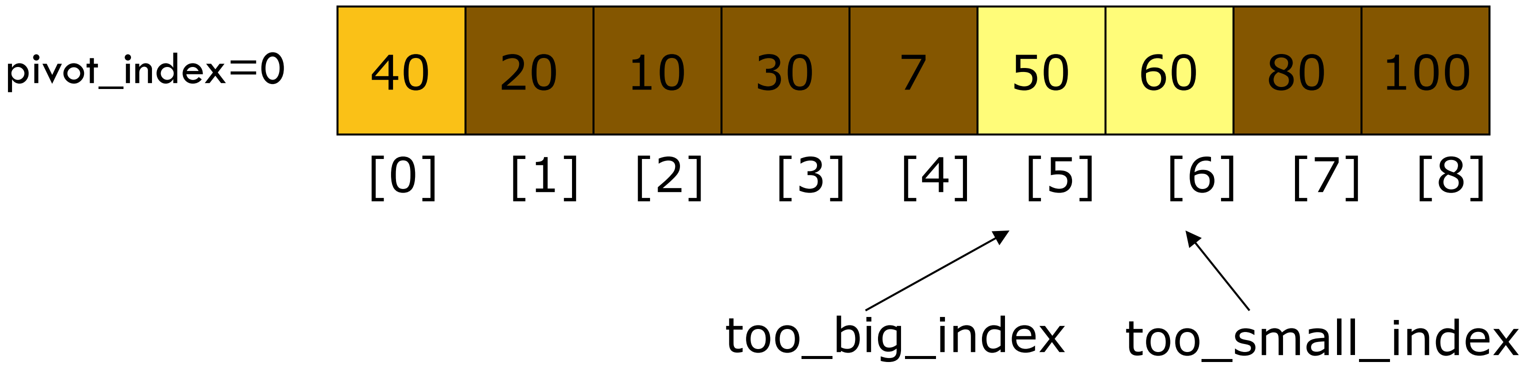


pivot_index=0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  While too_small_index > too_big_index, go to 1.
```



pivot_index=0

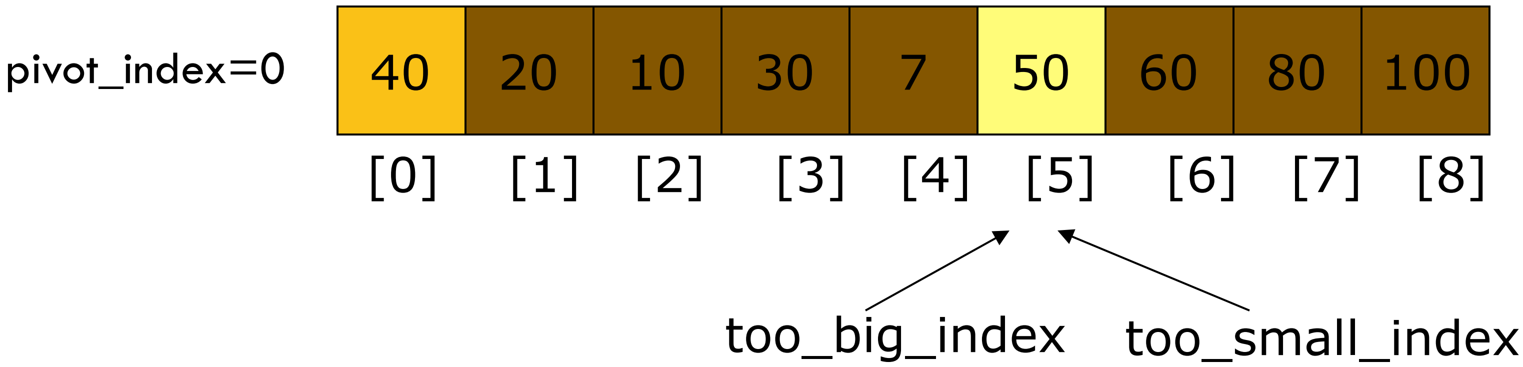
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index

too_small_index

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  While too_small_index > too_big_index, go to 1.
```



pivot_index=0

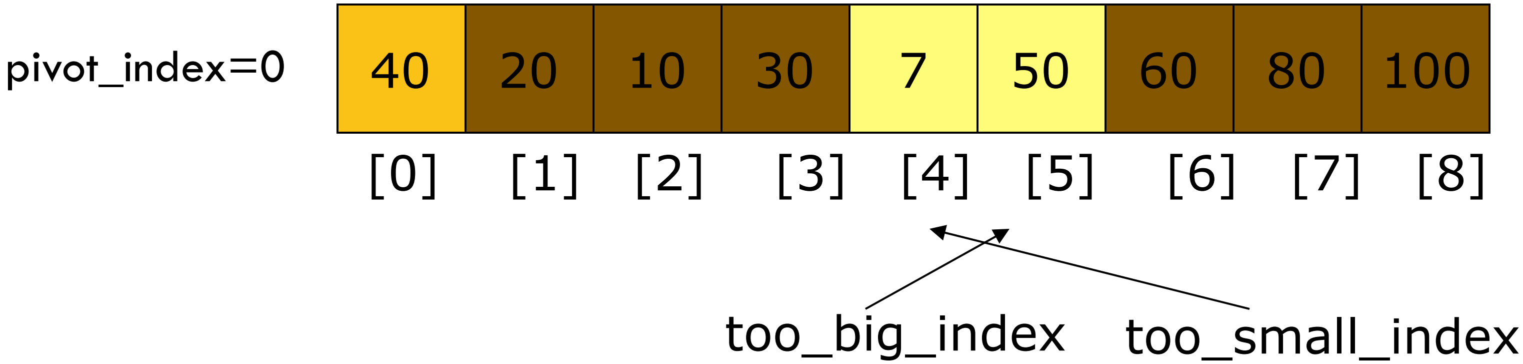
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index

too_small_index

Quick Sort

```
1.  While data[too_big_index] <= data[pivot]
        ++too_big_index
2.  While data[too_small_index] > data[pivot]
        --too_small_index
3.  If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4.  While too_small_index > too_big_index, go to 1.
```



pivot_index=0

40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

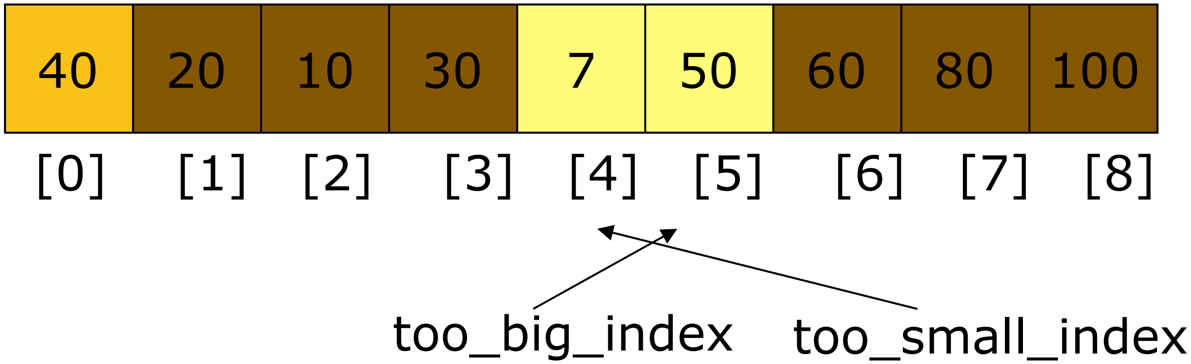
too_big_index

too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
        ++too_big_index
2. While data[too_small_index] > data[pivot]
        --too_small_index
3. If too_big_index < too_small_index
        swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
```

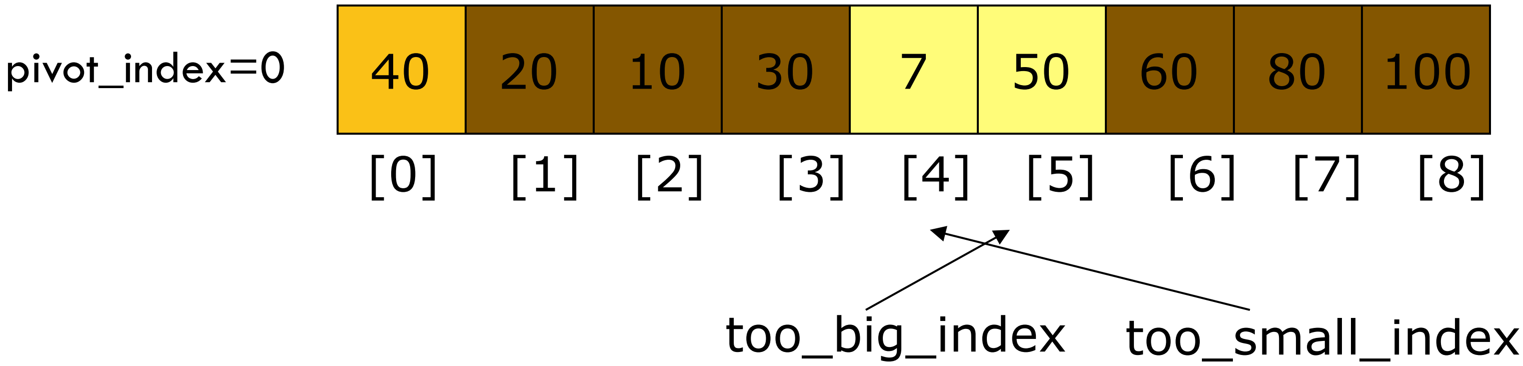
pivot_index=0



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

Quick Sort

```
  1.  While data[too_big_index] <= data[pivot]
            ++too_big_index
  2.  While data[too_small_index] > data[pivot]
            --too_small_index
  3.  If too_big_index < too_small_index
            swap data[too_big_index] and data[too_small_index]
→ 4.  While too_small_index > too_big_index, go to 1.
```



pivot_index=0

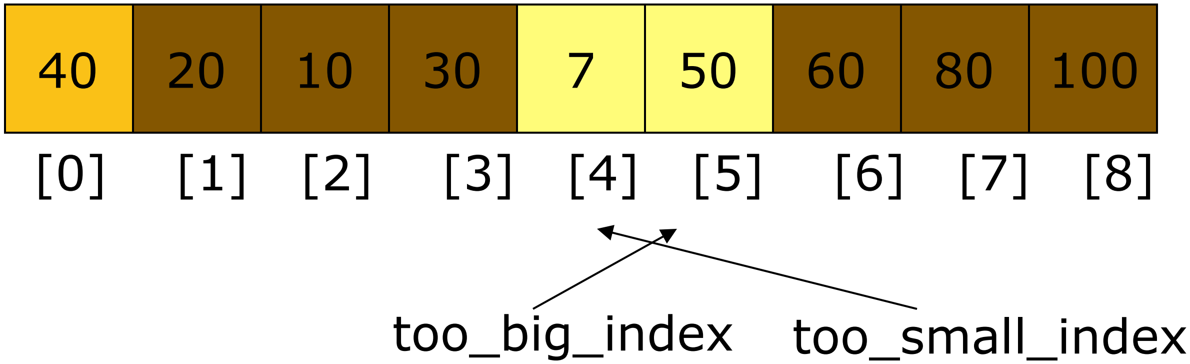
40	20	10	30	7	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]
```

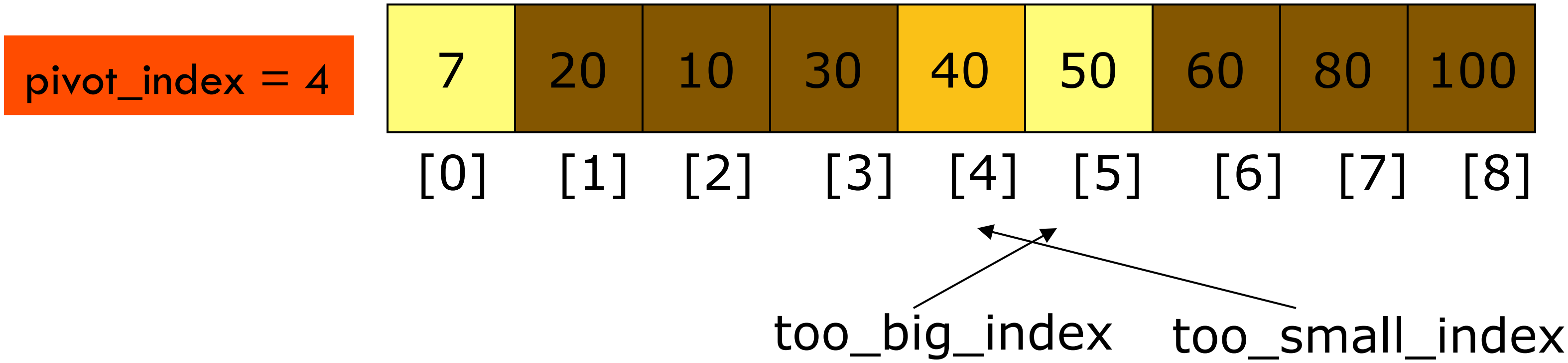
pivot_index=0



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

Quick Sort

```
1. While data[too_big_index] <= data[pivot]
       ++too_big_index
2. While data[too_small_index] > data[pivot]
       --too_small_index
3. If too_big_index < too_small_index
       swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
→ 5. Swap data[too_small_index] and data[pivot_index]
```

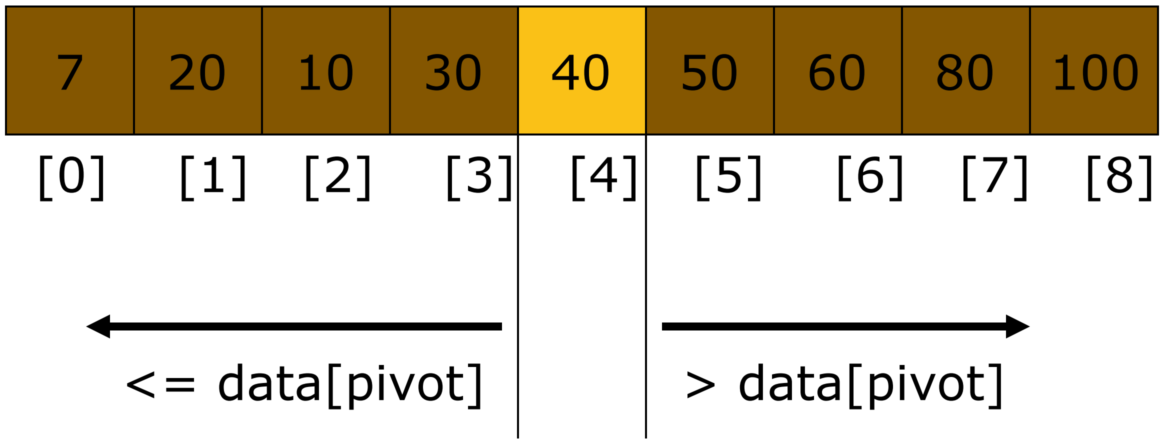


pivot_index = 4

7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

too_big_index too_small_index

Partition Result

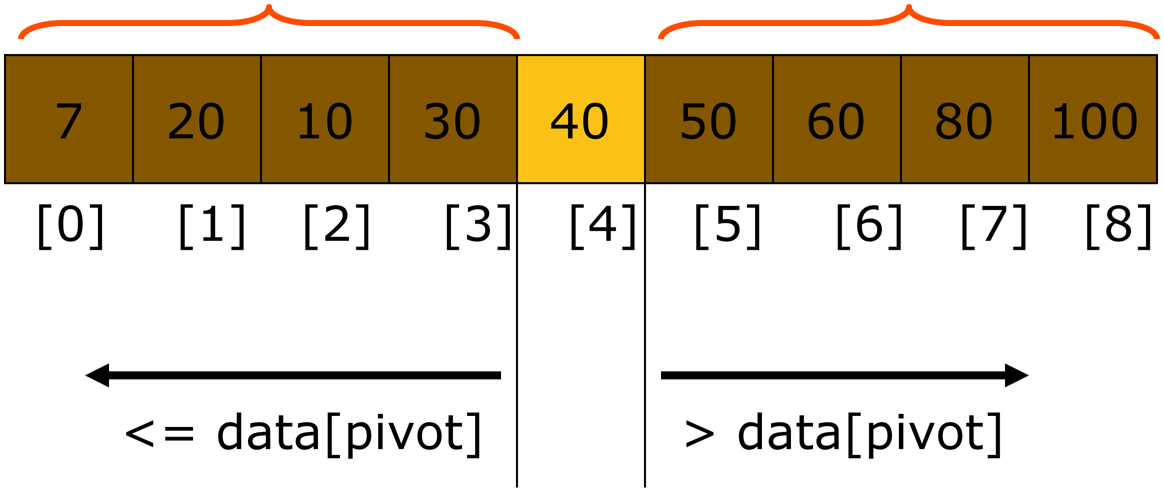


7	20	10	30	40	50	60	80	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

<= data[pivot]

> data[pivot]

Recursion: Quicksort Sub-arrays

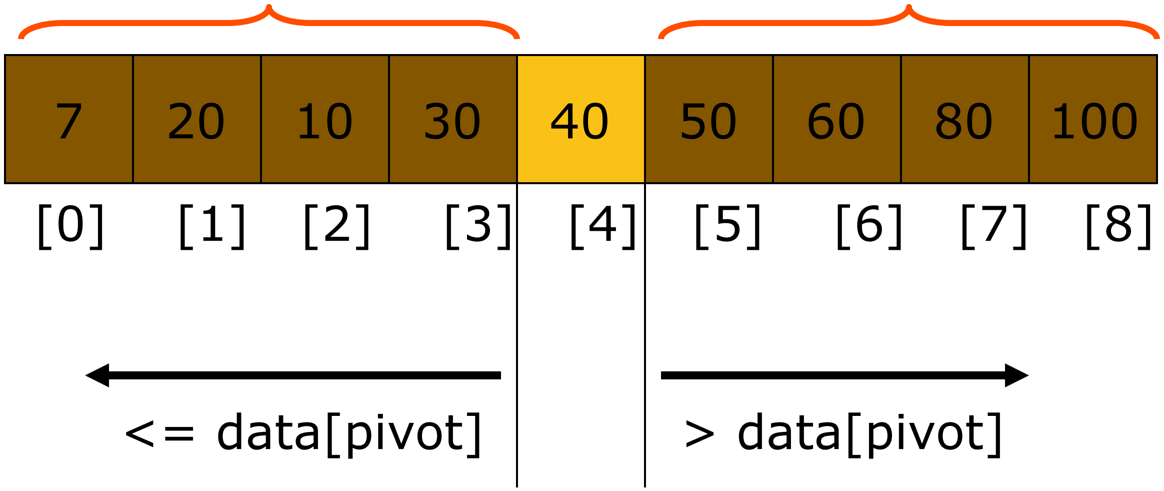


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
7	20	10	30	40	50	60	80	100

<= data[pivot]

> data[pivot]

Recursion: Quicksort Sub-arrays



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
7	20	10	30	40	50	60	80	100

`quickSort(arr,0,5)`

0	1	2	3	4	5
6	5	9	12	3	4

```
quickSort(arr,0,5)
```

```
partition(arr,0,5)
```

0	1	2	3	4	5
6	5	9	12	3	4

quickSort(arr,0,5)

partition(arr,0,5)

0	1	2	3	4	5
6	5	9	12	3	4

pivot= ?

Partition Initialization...

quickSort(arr,0,5)

partition(arr,0,5)

0	1	2	3	4	5
6	5	9	12	3	4

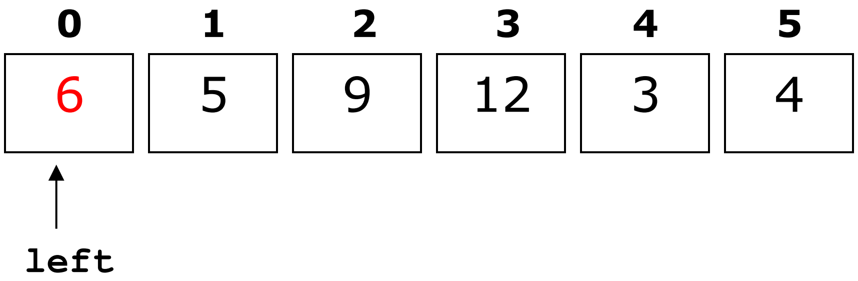
pivot=6

Partition Initialization...

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`



0	1	2	3	4	5
6	5	9	12	3	4

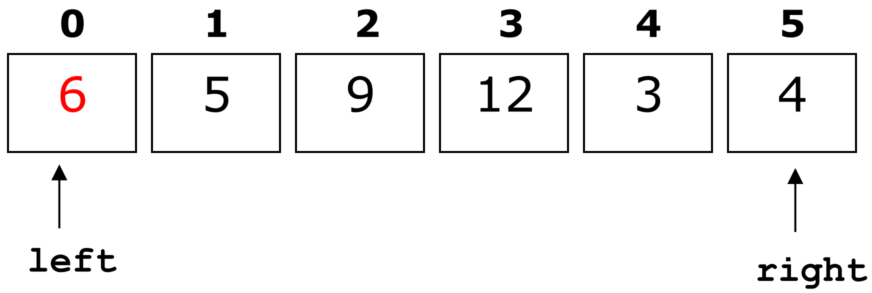
`left`

Partition Initialization...

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`



0	1	2	3	4	5
6	5	9	12	3	4

left

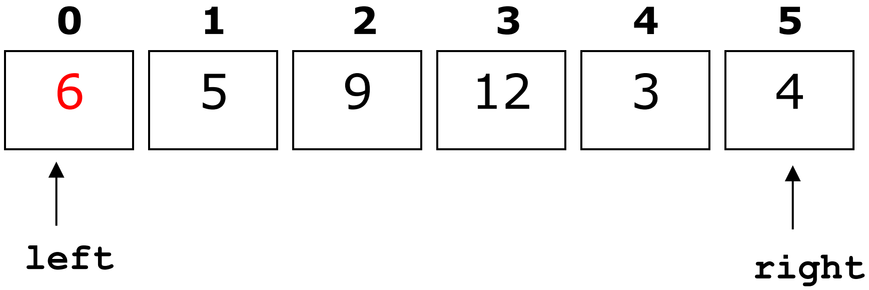
right

Partition Initialization...

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`



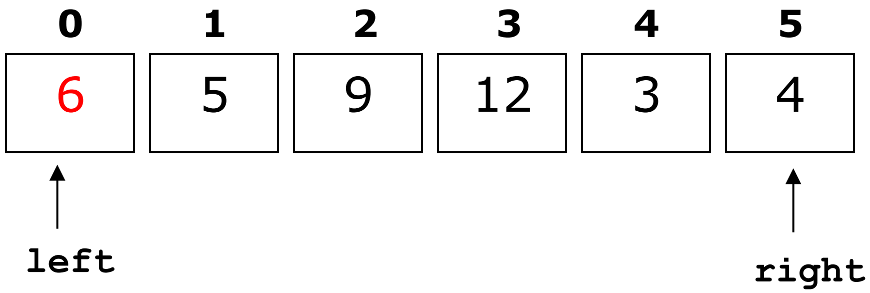
0	1	2	3	4	5
6	5	9	12	3	4

right moves to the left until value that should be to left of pivot...

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



0	1	2	3	4	5
6	5	9	12	3	4

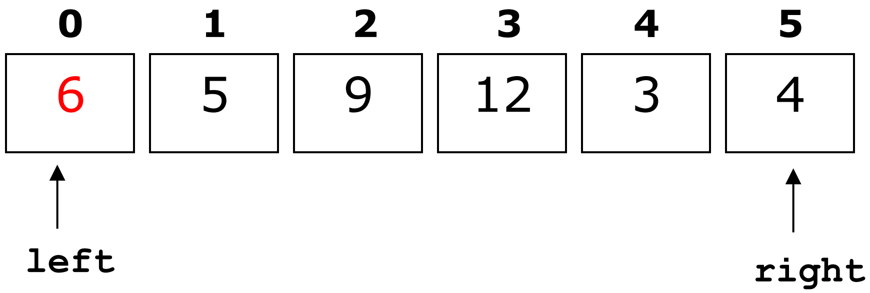
left

right

```
quickSort(arr,0,5)

partition(arr,0,5)

pivot=6
```



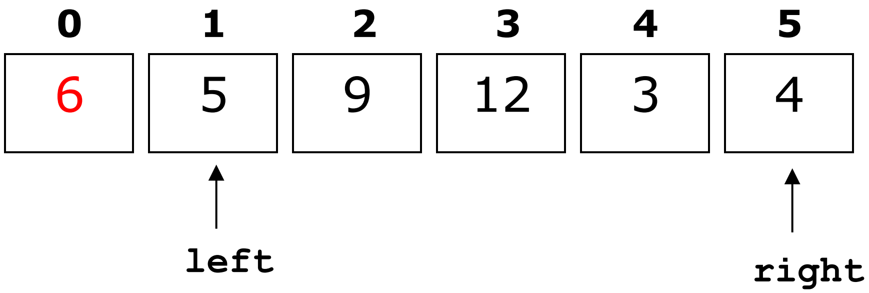
0	1	2	3	4	5
6	5	9	12	3	4

left moves to the right until value that should be to right of pivot...

```
quickSort(arr,0,5)
```

```
partition(arr,0,5)
```

```
pivot=6
```

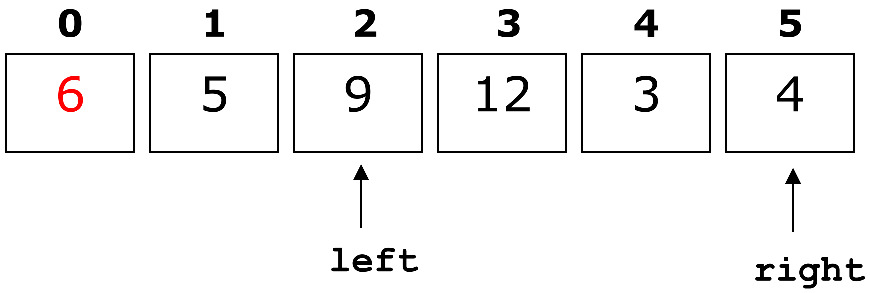


0	1	2	3	4	5
6	5	9	12	3	4

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`

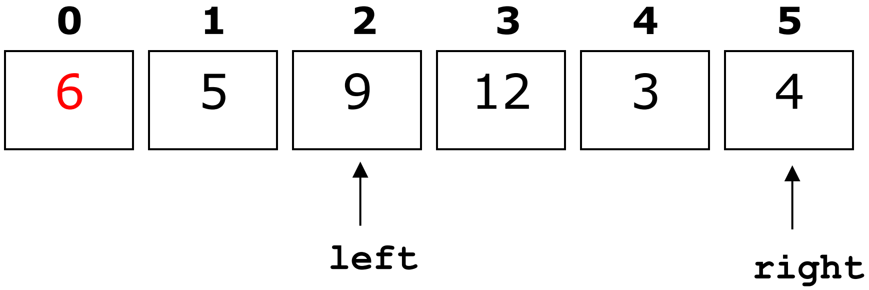


0	1	2	3	4	5
6	5	9	12	3	4

```
quickSort(arr,0,5)
```

```
partition(arr,0,5)
```

```
pivot=6
```



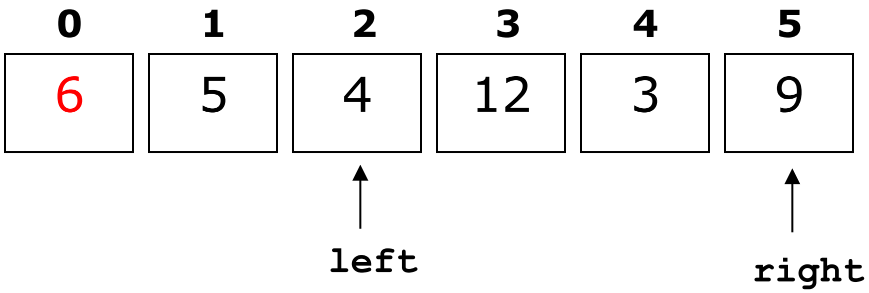
0	1	2	3	4	5
6	5	9	12	3	4

swap arr[left] and arr[right]

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



0	1	2	3	4	5
6	5	4	12	3	9

left

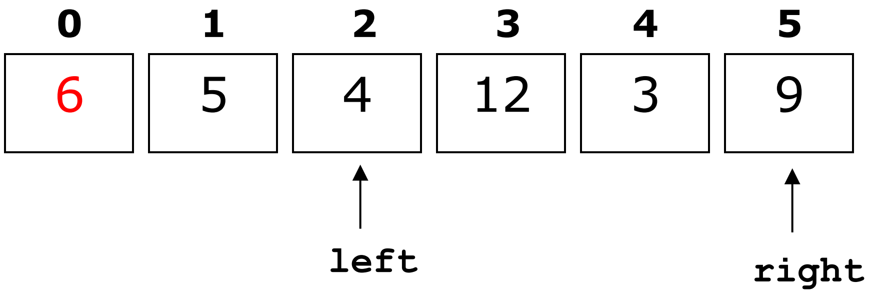
right

repeat right/left scan
UNTIL left & right cross

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



0	1	2	3	4	5
6	5	4	12	3	9

left

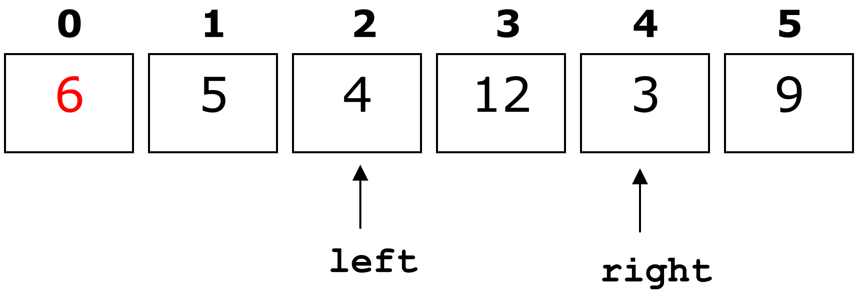
right

right moves to the left until value that should be to left of pivot...

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6

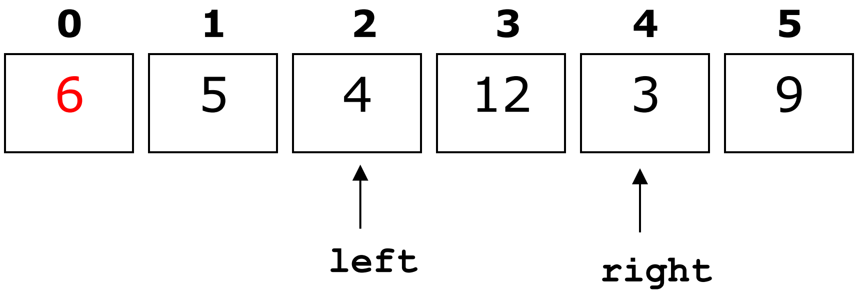


0	1	2	3	4	5
6	5	4	12	3	9

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`



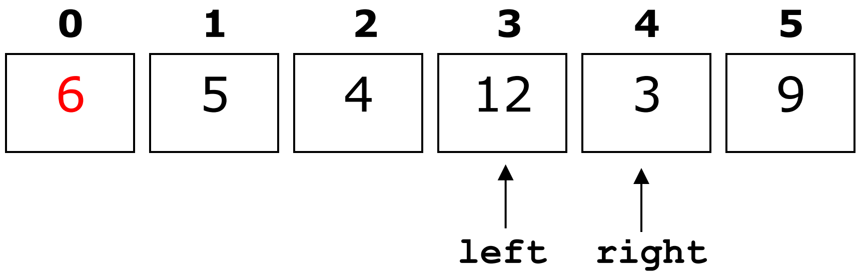
0	1	2	3	4	5
6	5	4	12	3	9

left moves to the right until value that should be to right of pivot...

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6

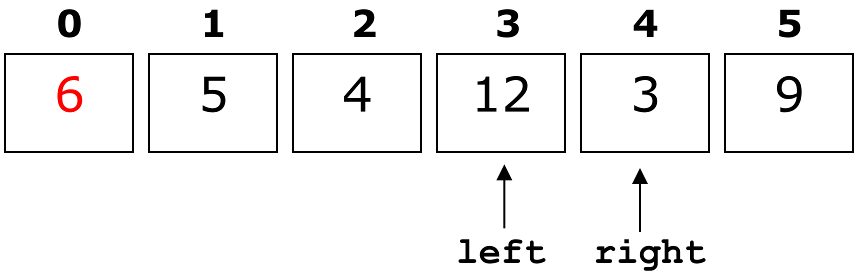


0	1	2	3	4	5
6	5	4	12	3	9

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



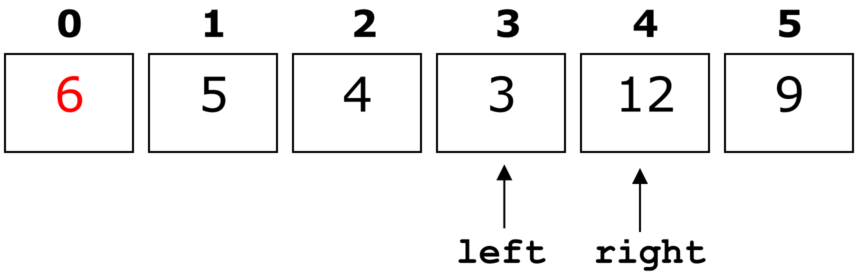
0	1	2	3	4	5
6	5	4	12	3	9

swap arr[left] and arr[right]

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



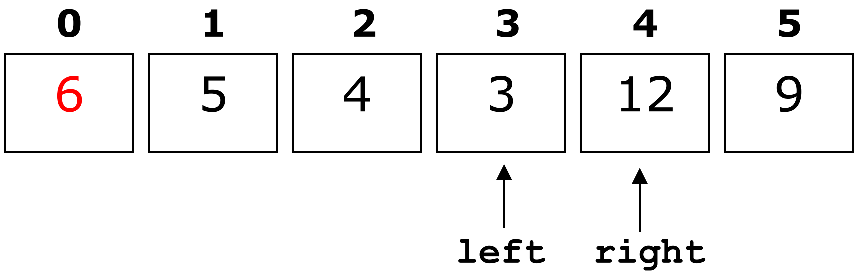
0	1	2	3	4	5
6	5	4	3	12	9

swap arr[left] and arr[right]

```
quickSort(arr,0,5)
```

```
partition(arr,0,5)
```

```
pivot=6
```



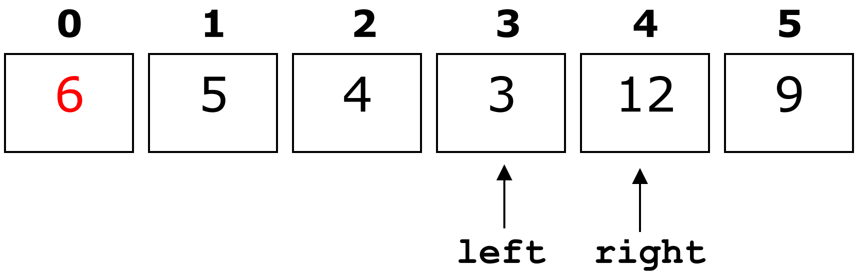
0	1	2	3	4	5
6	5	4	3	12	9

repeat right/left scan
UNTIL left & right cross

```
quickSort(arr,0,5)
```

```
partition(arr,0,5)
```

```
pivot=6
```



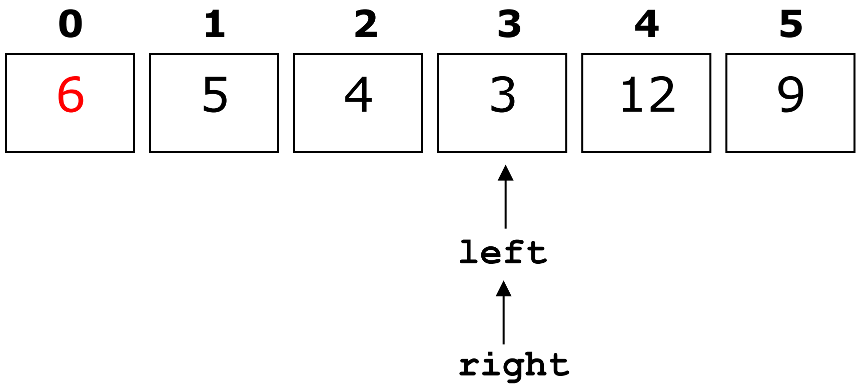
0	1	2	3	4	5
6	5	4	3	12	9

right moves to the left until value that should be to left of pivot...

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6

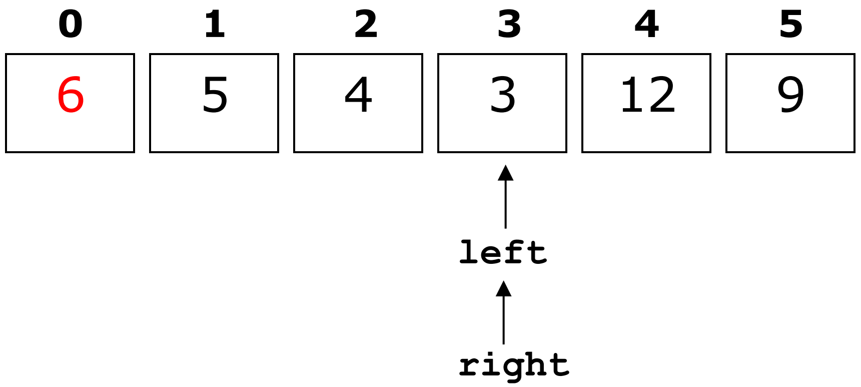


0	1	2	3	4	5
6	5	4	3	12	9

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



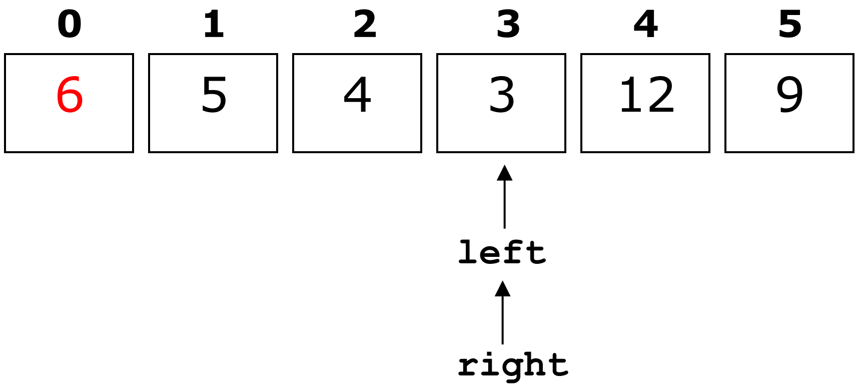
0	1	2	3	4	5
6	5	4	3	12	9

right & left CROSS!!!

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`



0	1	2	3	4	5
6	5	4	3	12	9

left

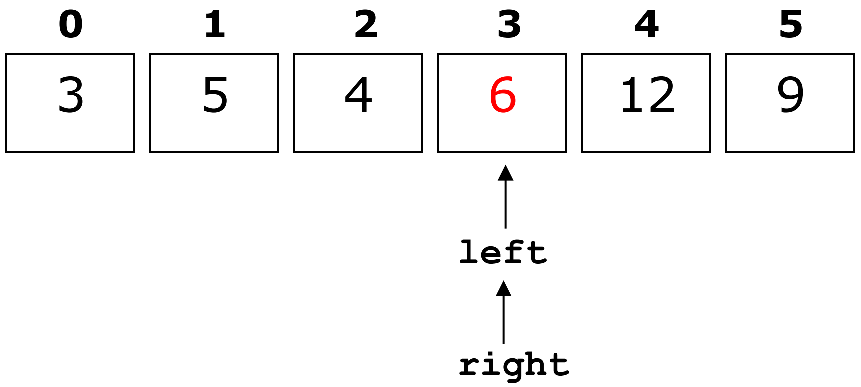
right

right & left CROSS!!!
1 - Swap pivot and arr[right]

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



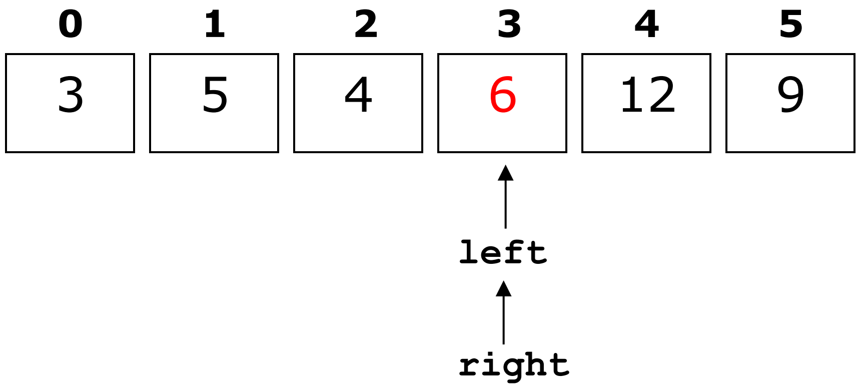
0	1	2	3	4	5
3	5	4	6	12	9

right & left CROSS!!!
1 - Swap pivot and arr[right]

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6

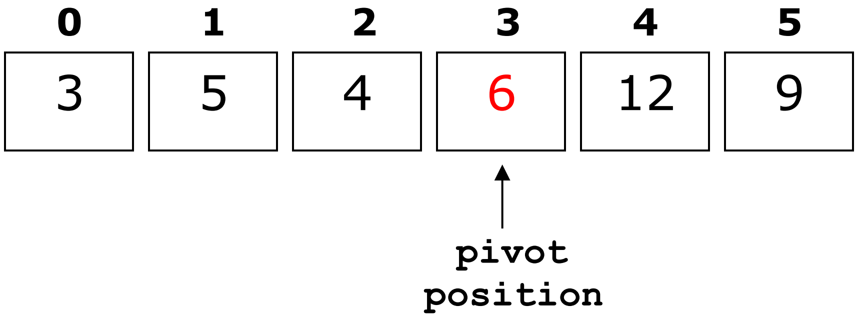


0	1	2	3	4	5
3	5	4	6	12	9

right & left CROSS!!!
1 - Swap pivot and arr[right]
2 - Return new location of pivot to caller

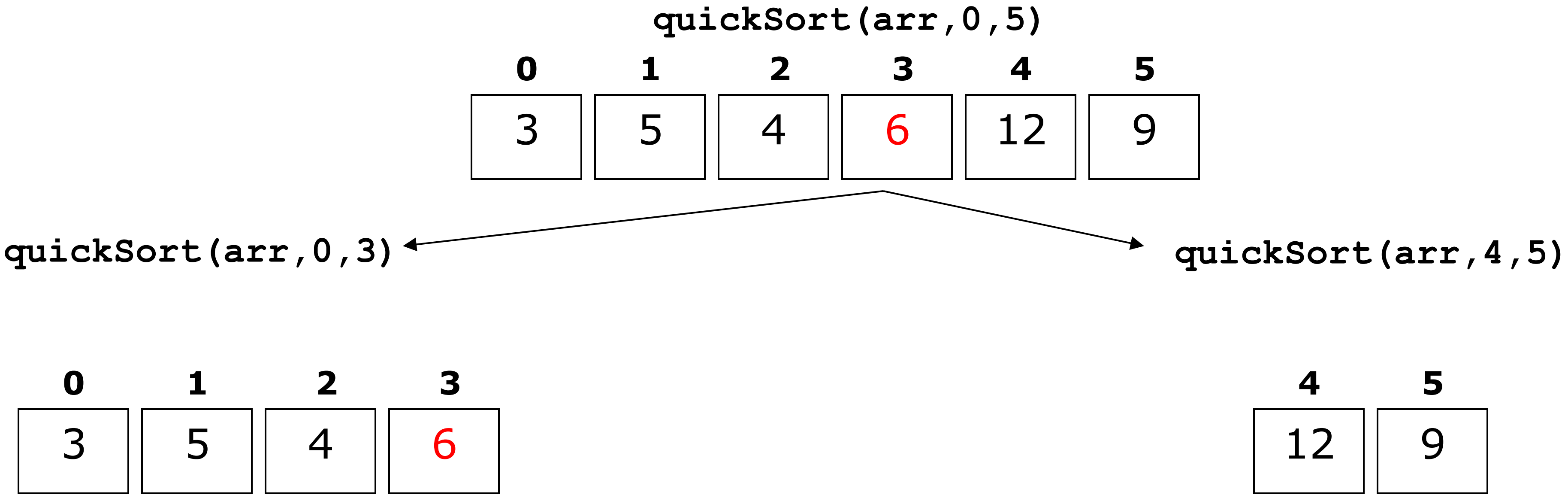
return 3

`quickSort(arr,0,5)`



0	1	2	3	4	5
3	5	4	6	12	9

Recursive calls to quickSort()
using partitioned array...



quickSort(arr,0,5)

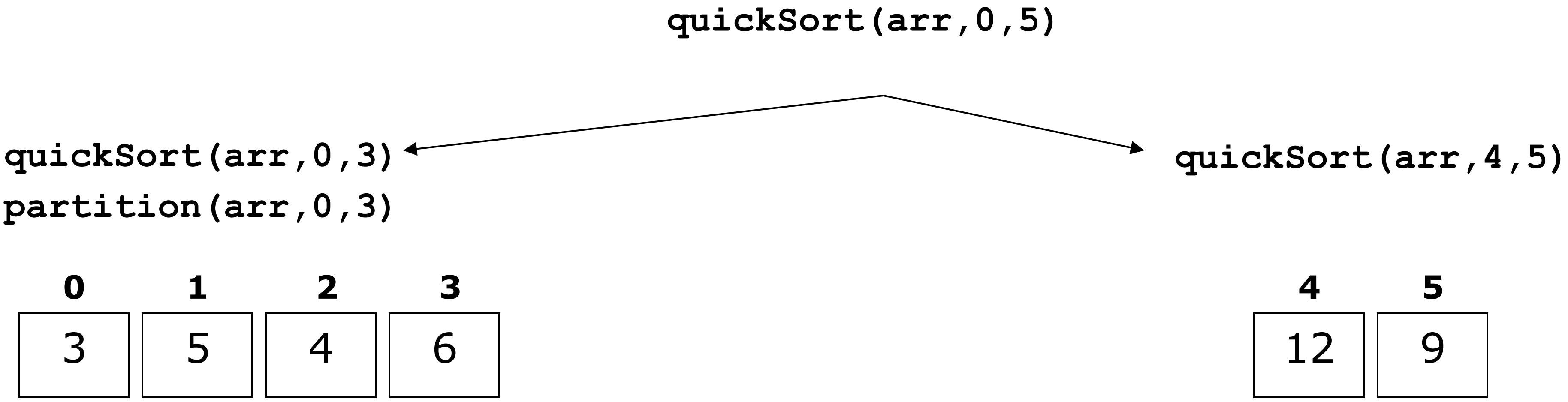
0	1	2	3	4	5
3	5	4	6	12	9

quickSort(arr,0,3)

0	1	2	3
3	5	4	6

quickSort(arr,4,5)

4	5
12	9



quickSort(arr,0,5)

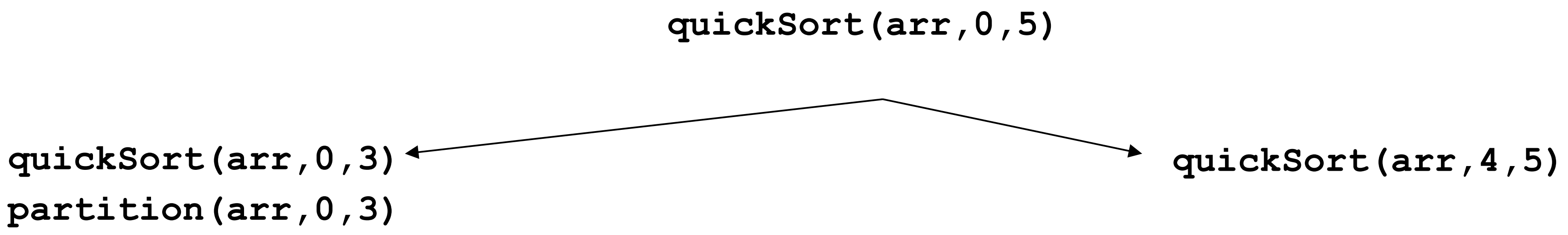
quickSort(arr,0,3)

partition(arr,0,3)

0	1	2	3
3	5	4	6

quickSort(arr,4,5)

4	5
12	9



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

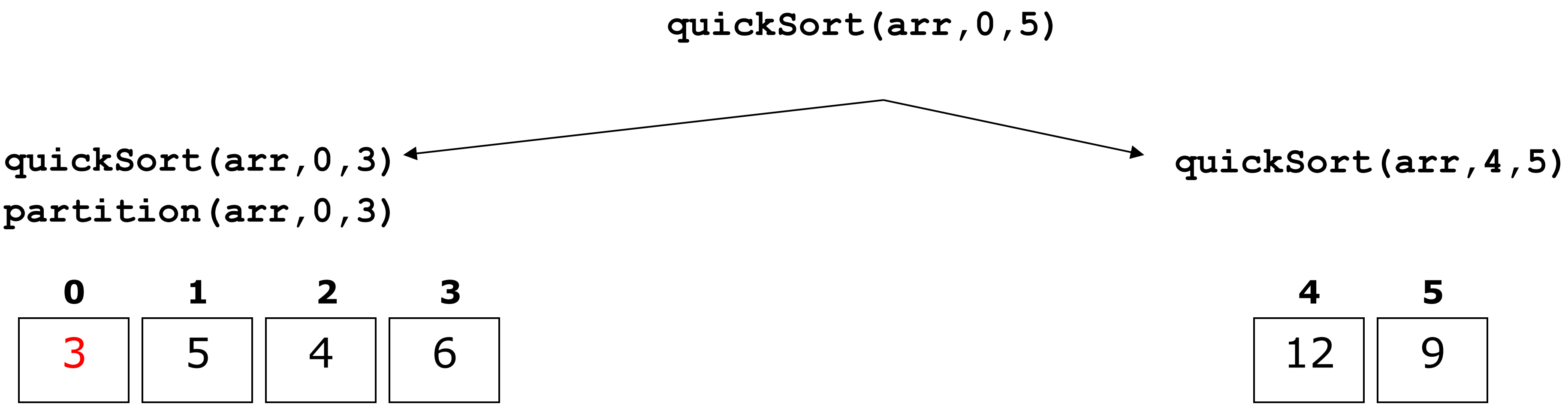
`partition(arr,0,3)`

`quickSort(arr,4,5)`

0	1	2	3
3	5	4	6

4	5
12	9

Partition Initialization...



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

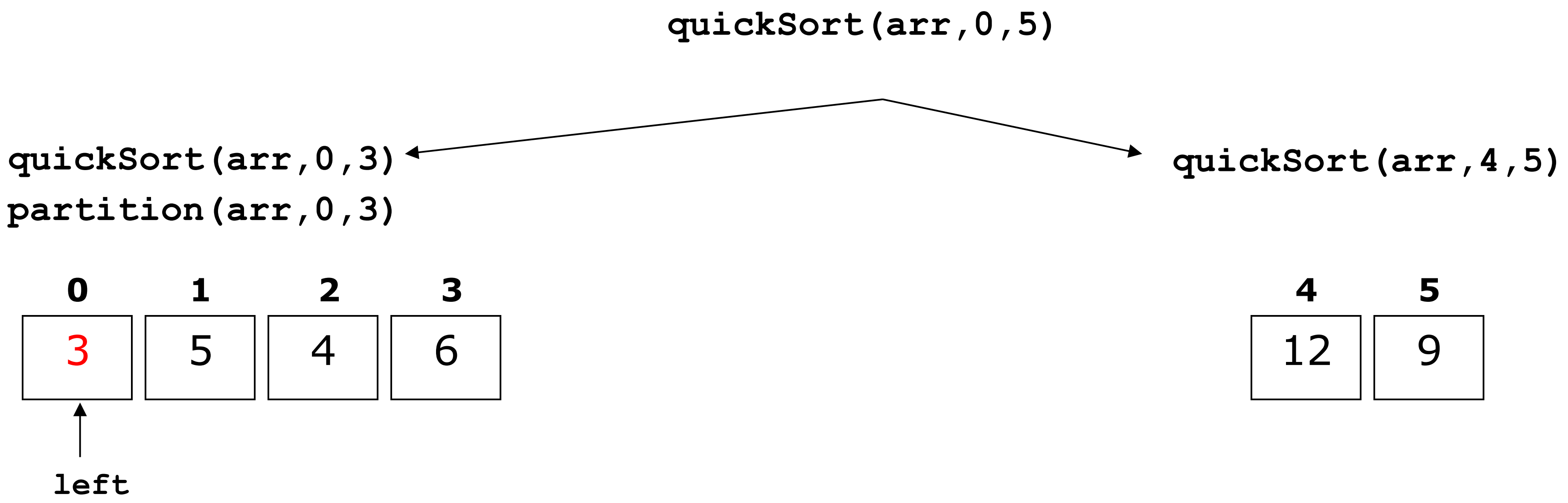
`partition(arr,0,3)`

0	1	2	3
3	5	4	6

`quickSort(arr,4,5)`

4	5
12	9

Partition Initialization...



quickSort(arr,0,5)

quickSort(arr,0,3)

partition(arr,0,3)

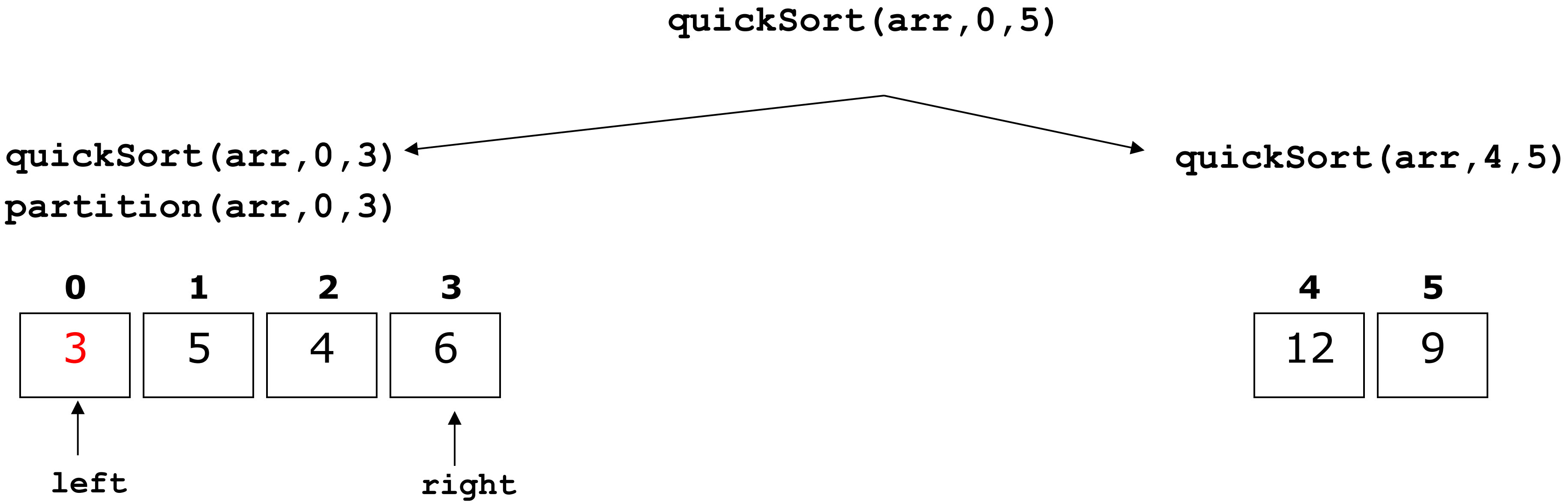
0	1	2	3
3	5	4	6

left

quickSort(arr,4,5)

4	5
12	9

Partition Initialization...



```
quickSort(arr,0,5)
```

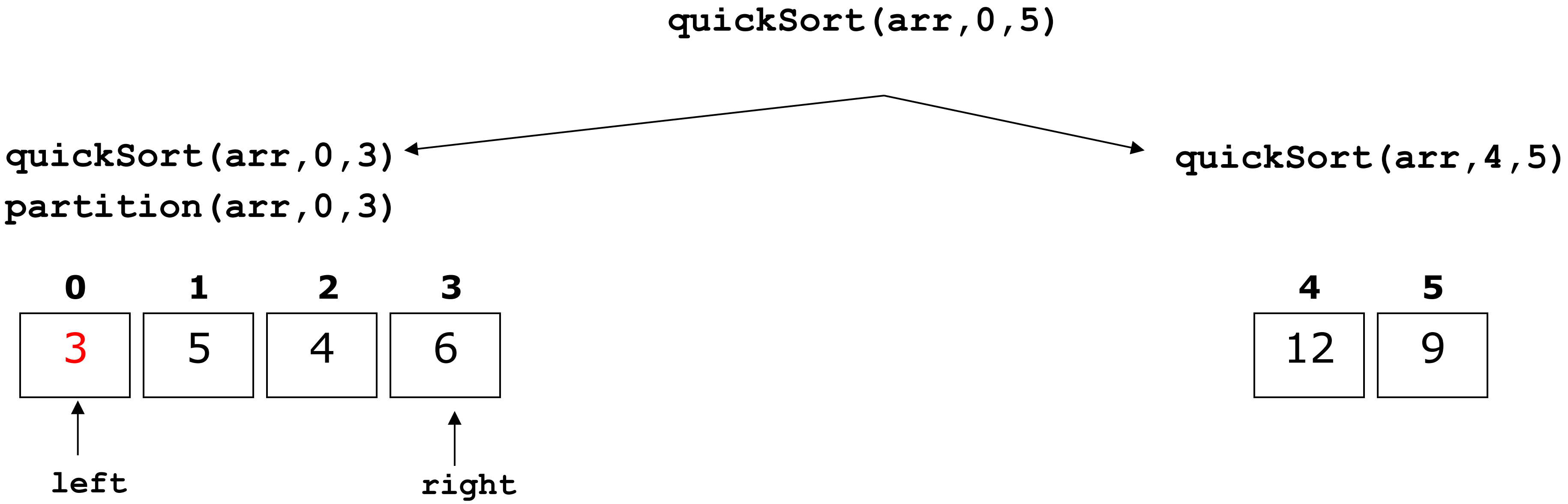
```
quickSort(arr,0,3)
partition(arr,0,3)
```

0	1	2	3
3	5	4	6

```
quickSort(arr,4,5)
```

4	5
12	9

Partition Initialization...



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

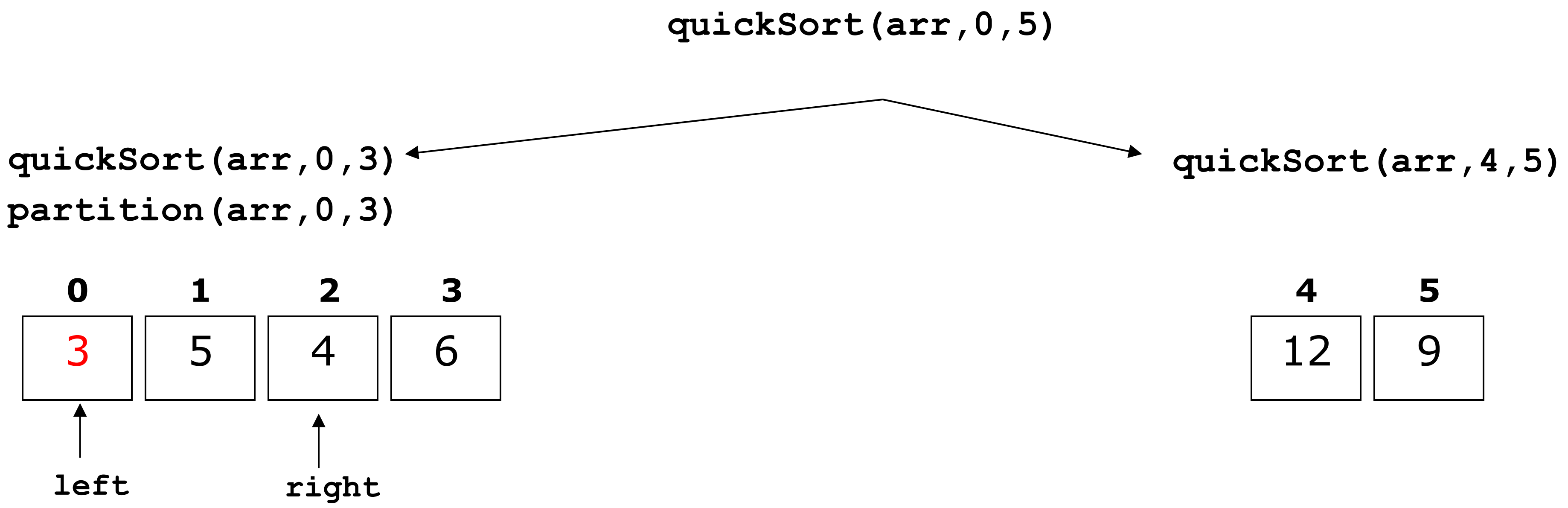
`partition(arr,0,3)`

0	1	2	3
3	5	4	6

`quickSort(arr,4,5)`

4	5
12	9

right moves to the left until value that should be to left of pivot...



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

`partition(arr,0,3)`

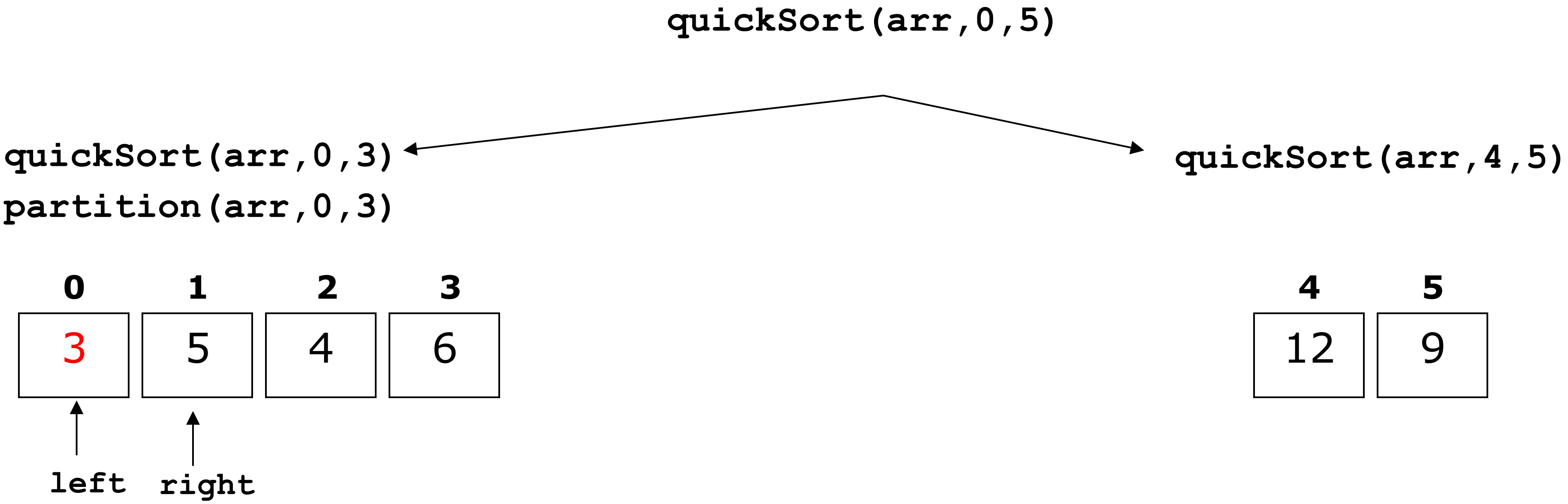
0	1	2	3
3	5	4	6

`left`

`right`

`quickSort(arr,4,5)`

4	5
12	9



quickSort(arr,0,5)

quickSort(arr,0,3)

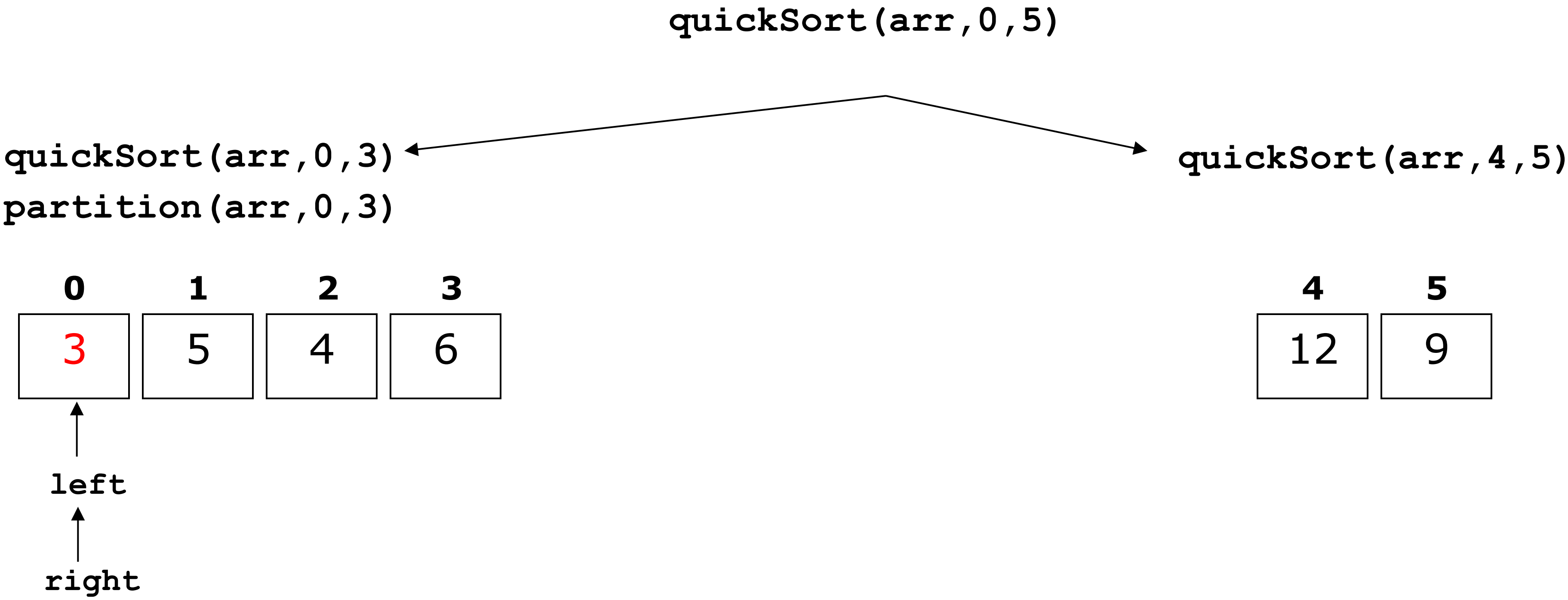
partition(arr,0,3)

0	1	2	3
3	5	4	6

left right

quickSort(arr,4,5)

4	5
12	9



quickSort(arr,0,5)

quickSort(arr,0,3)

partition(arr,0,3)

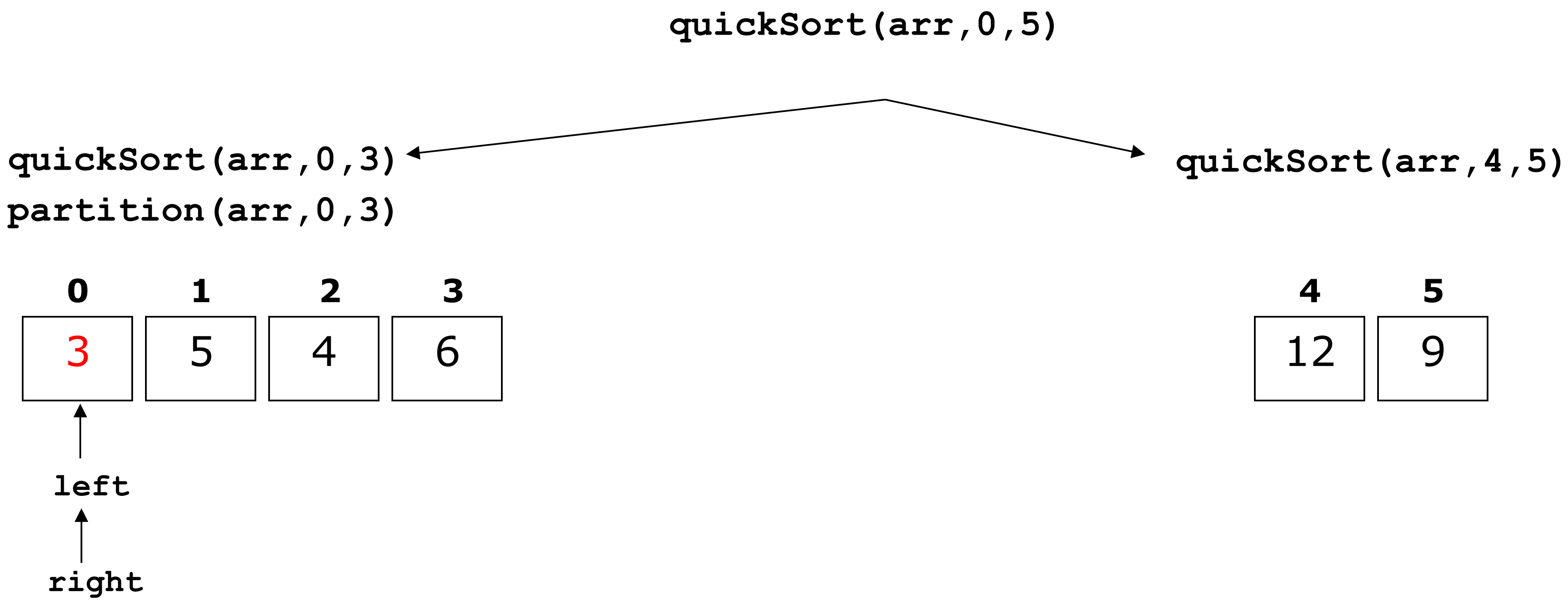
0	1	2	3
3	5	4	6

left

right

quickSort(arr,4,5)

4	5
12	9



quickSort(arr,0,5)

quickSort(arr,0,3)

partition(arr,0,3)

quickSort(arr,4,5)

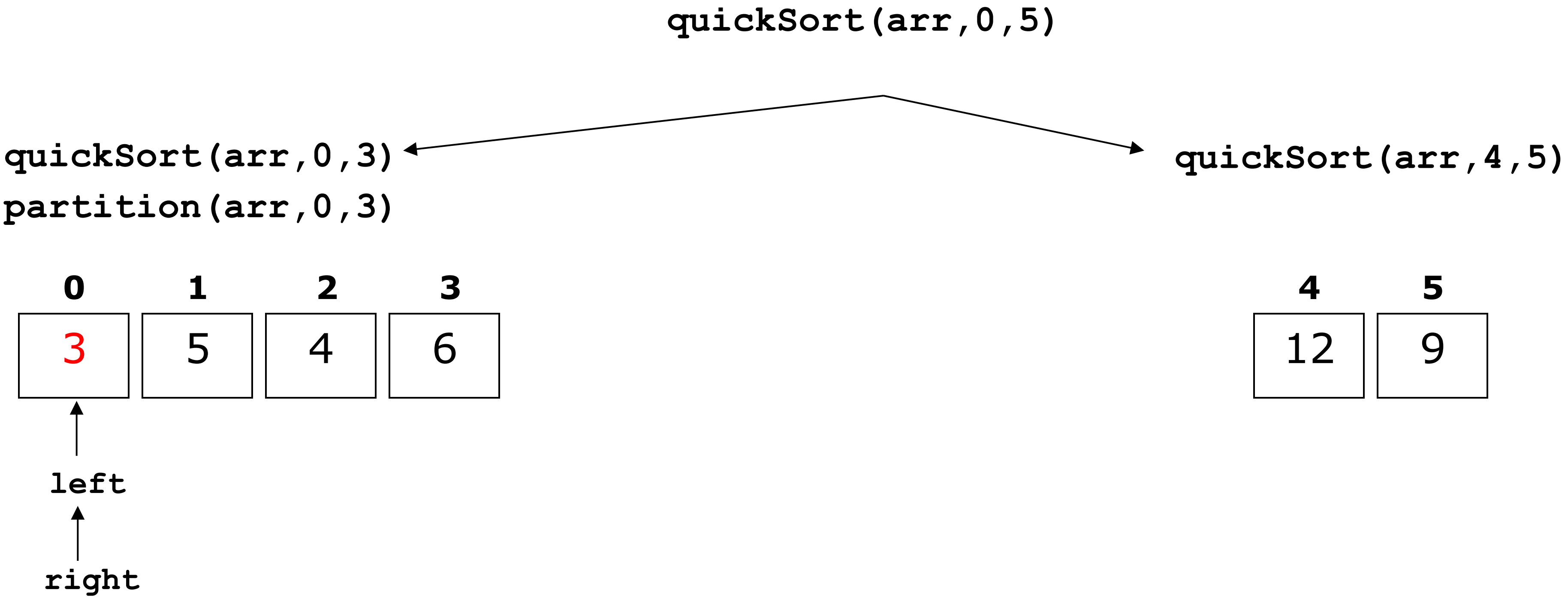
0	1	2	3
3	5	4	6

4	5
12	9

left

right

right & left CROSS!!!



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

`partition(arr,0,3)`

0	1	2	3
3	5	4	6

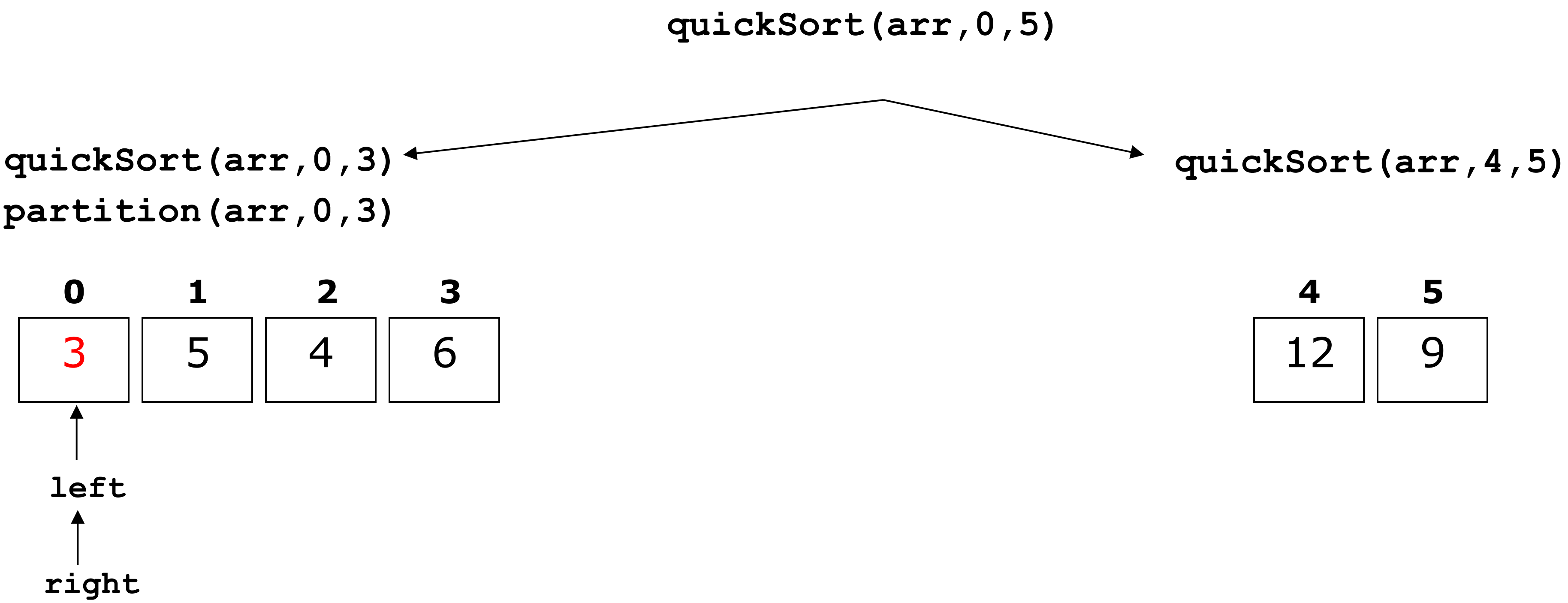
`left`

`right`

`quickSort(arr,4,5)`

4	5
12	9

right & left CROSS!!!
1 - Swap pivot and arr[right]



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`
`partition(arr,0,3)`

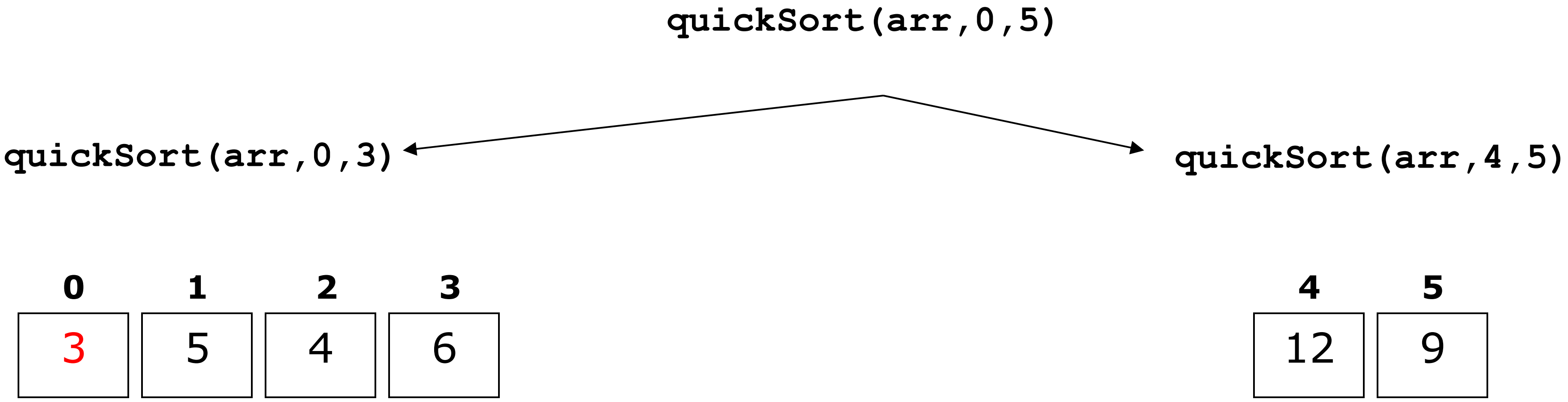
0	1	2	3
3	5	4	6

`quickSort(arr,4,5)`

4	5
12	9

right & left CROSS!!!
1 - Swap pivot and arr[right]
2 - Return new location of pivot to caller

`return 0`



`quickSort(arr,0,5)`

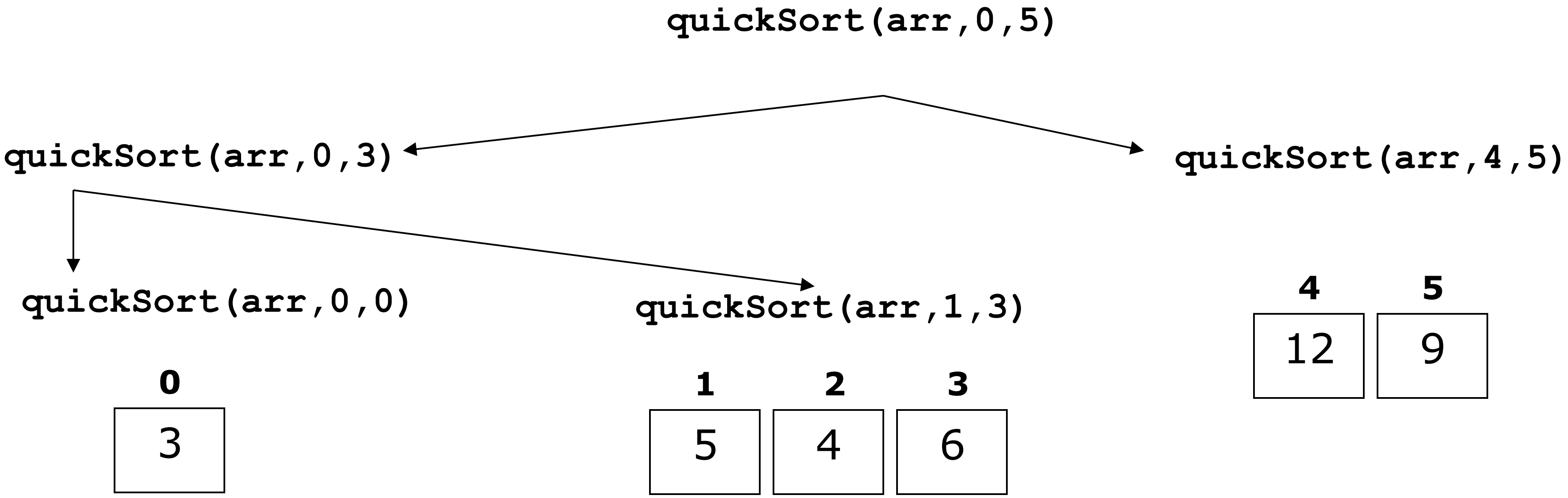
`quickSort(arr,0,3)`

`quickSort(arr,4,5)`

0	1	2	3
3	5	4	6

4	5
12	9

Recursive calls to quickSort()
using partitioned array...



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

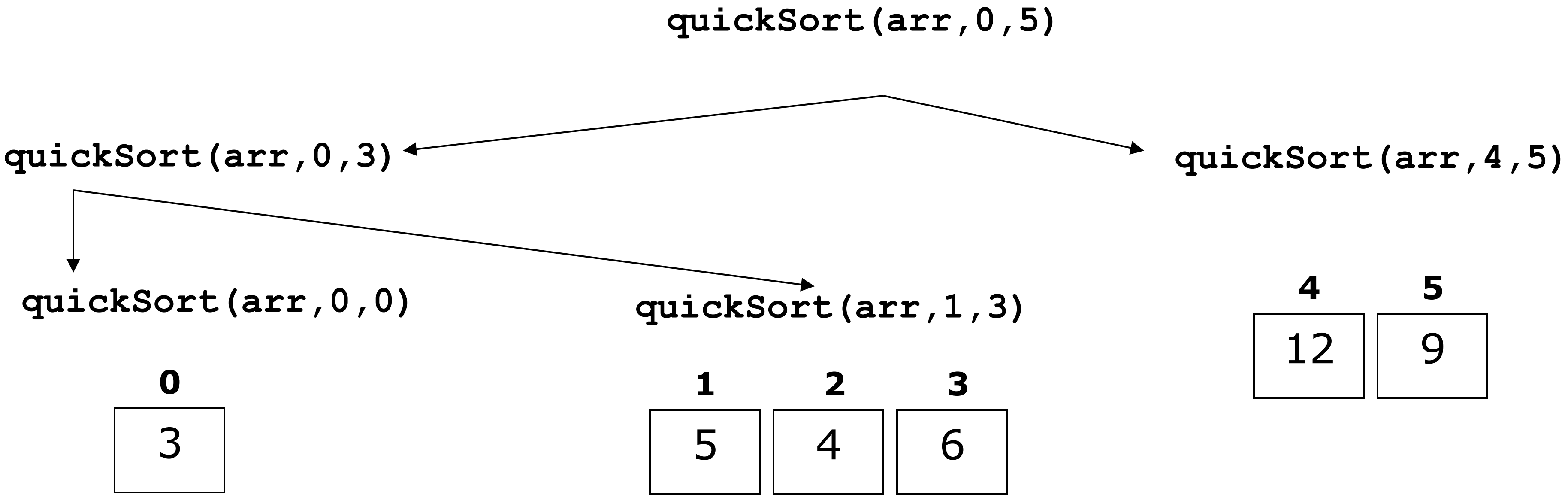
4	5
12	9

quickSort(arr,0,0)

0
3

quickSort(arr,1,3)

1	2	3
5	4	6



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

4	5
12	9

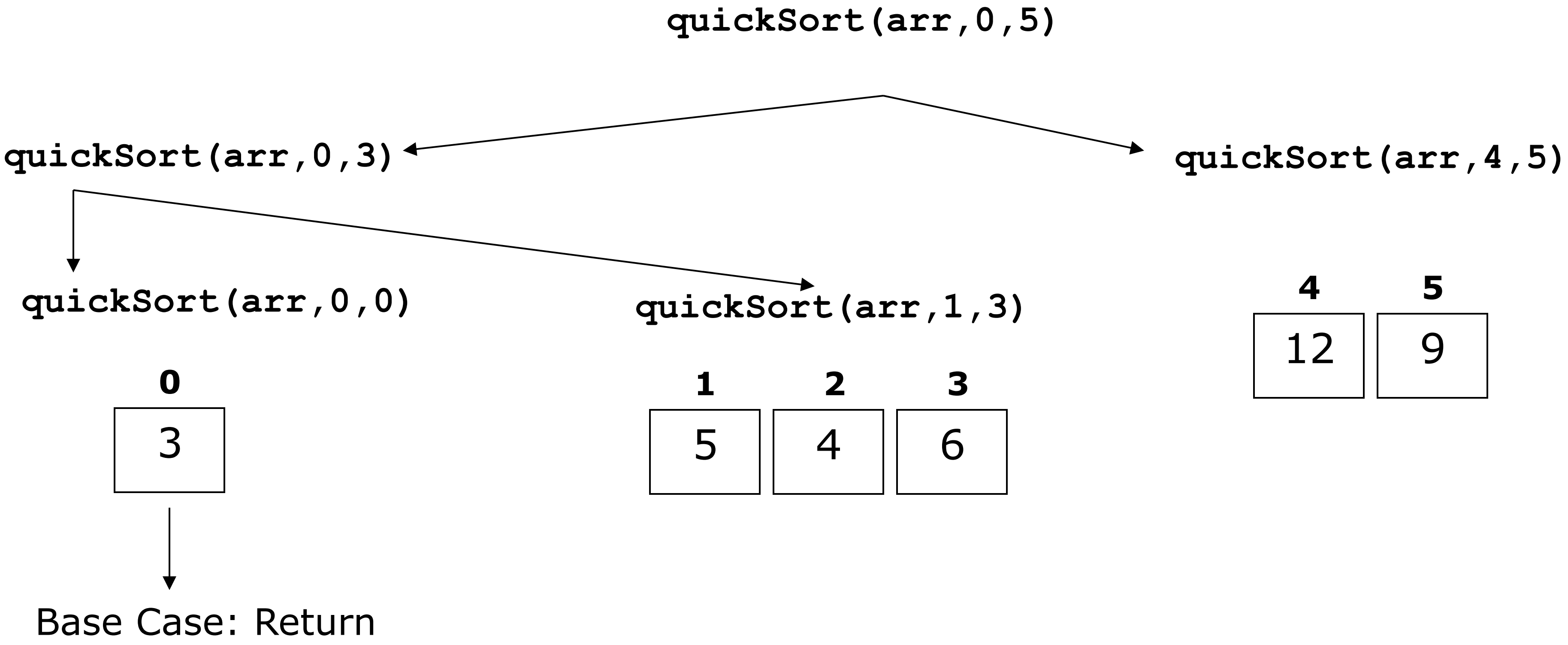
quickSort(arr,0,0)

0
3

quickSort(arr,1,3)

1	2	3
5	4	6

Base case triggered...
halting recursion.

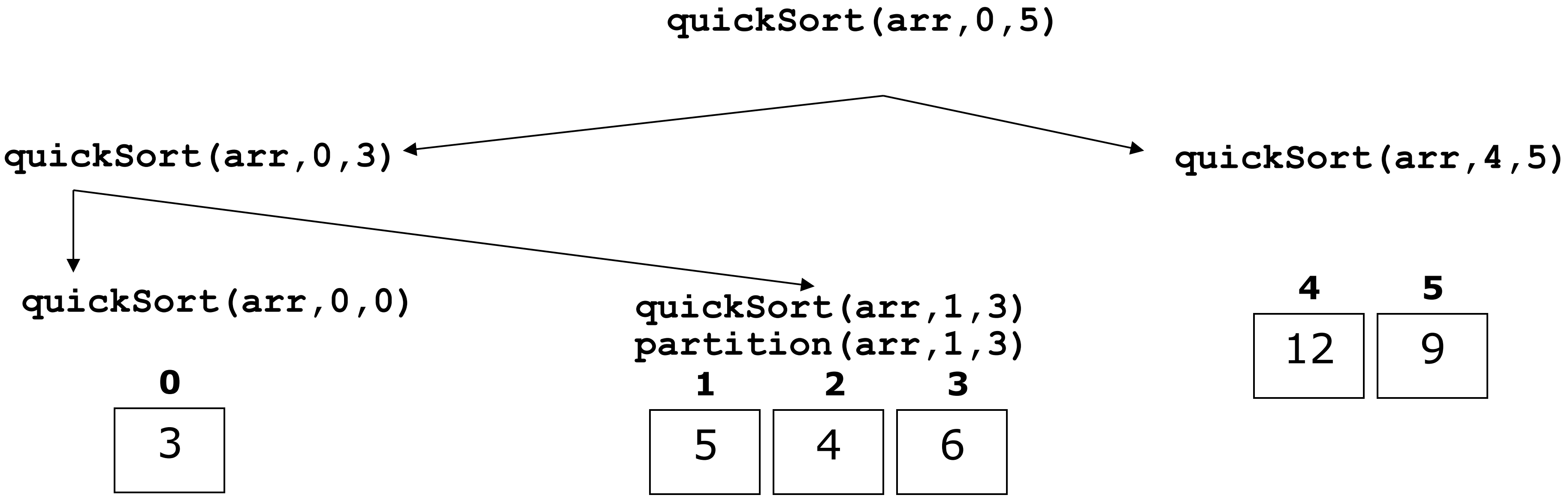


0
3

Base Case: Return

1	2	3
5	4	6

4	5
12	9



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

4	5
12	9

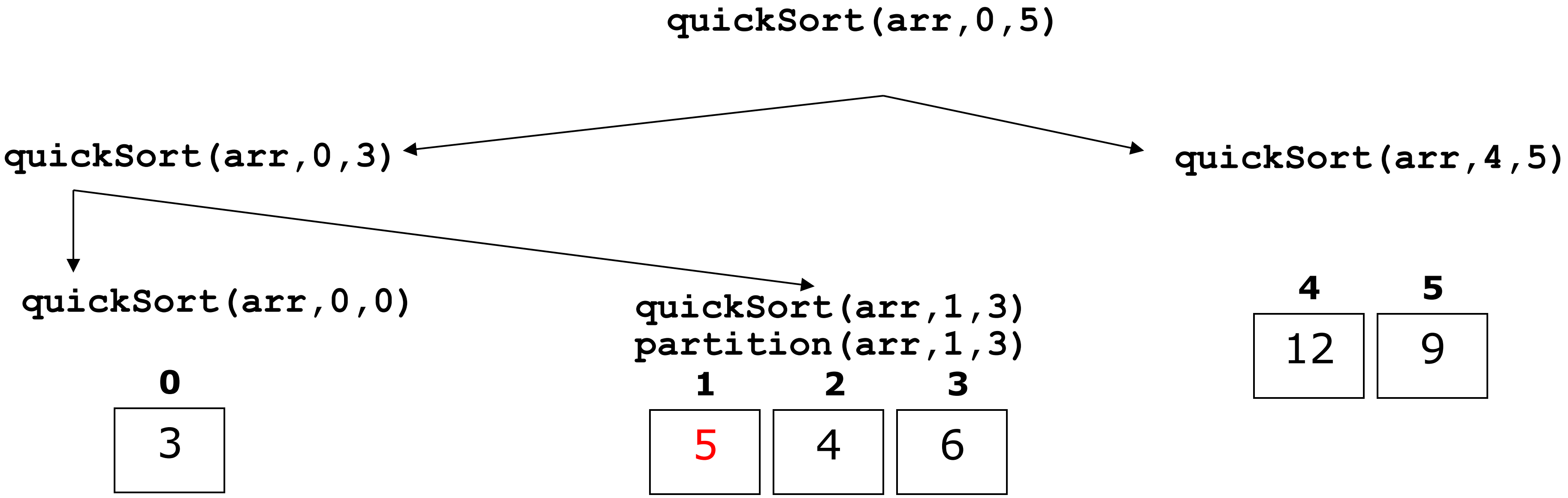
quickSort(arr,0,0)

0
3

quickSort(arr,1,3)
partition(arr,1,3)

1	2	3
5	4	6

Partition Initialization...



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

`quickSort(arr,4,5)`

4	5
12	9

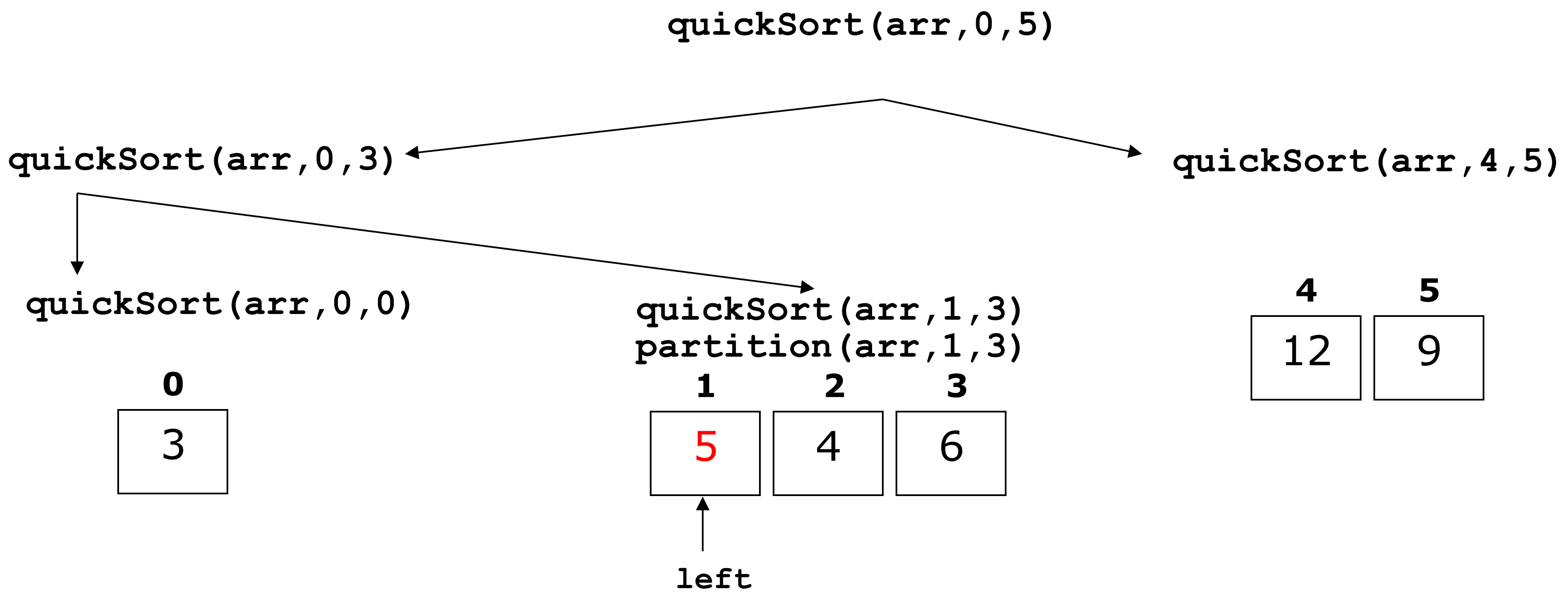
`quickSort(arr,0,0)`

0
3

`quickSort(arr,1,3)`
`partition(arr,1,3)`

1	2	3
5	4	6

Partition Initialization...



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

0
3

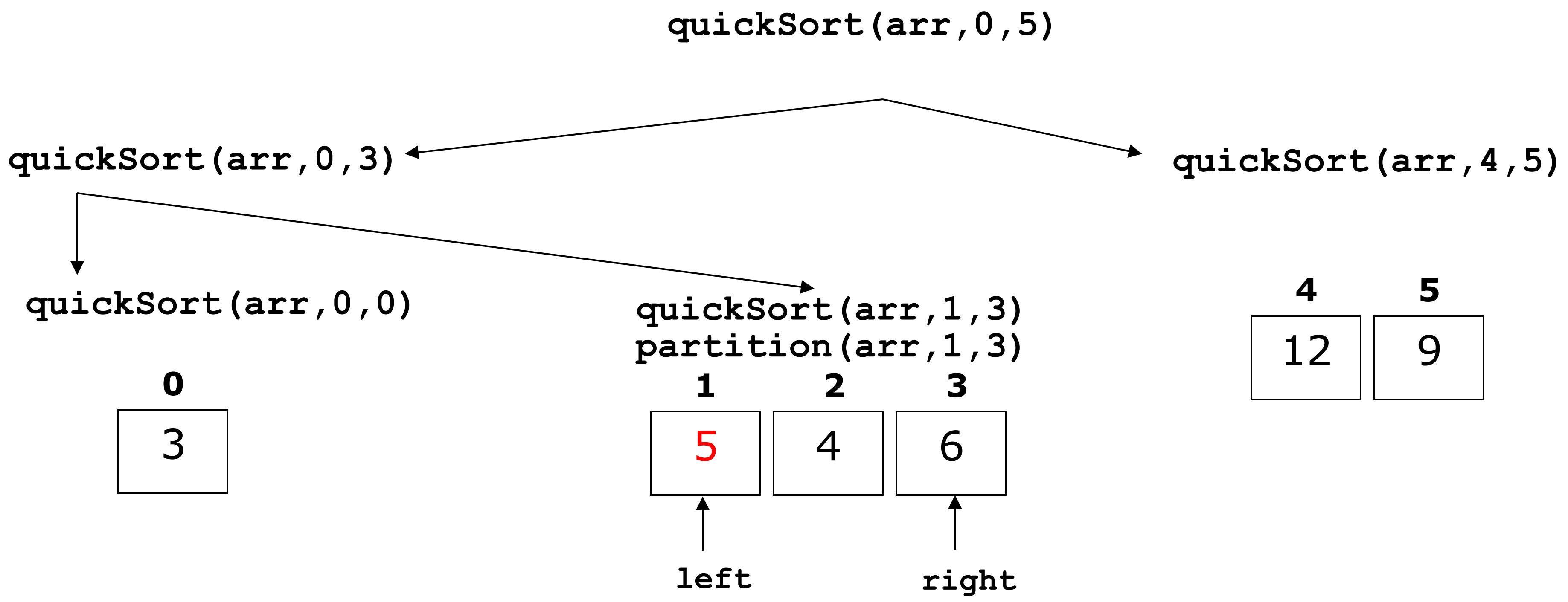
quickSort(arr,1,3)
partition(arr,1,3)

1	2	3
5	4	6

left

4	5
12	9

Partition Initialization...



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

`quickSort(arr,4,5)`

`quickSort(arr,0,0)`

0
3

`quickSort(arr,1,3)`
`partition(arr,1,3)`

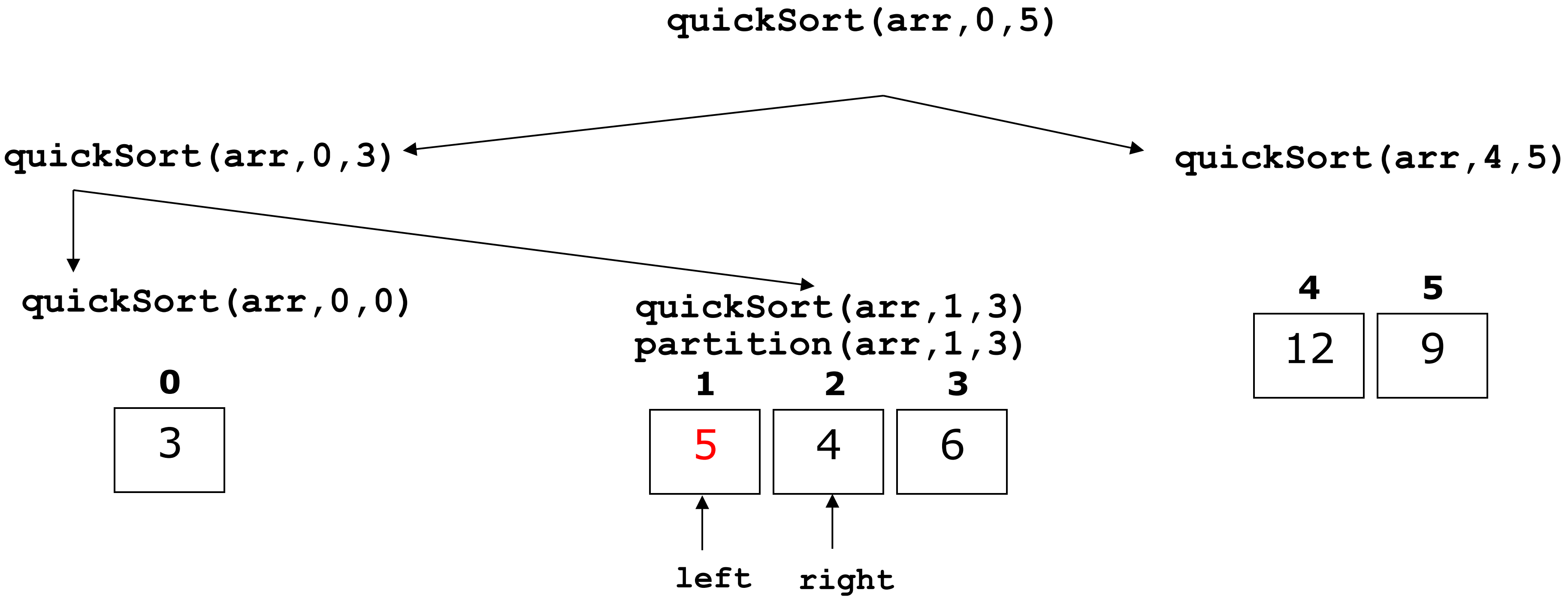
1	2	3
5	4	6

`left`

`right`

4	5
12	9

right moves to the left until
value that should be to left
of pivot...



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

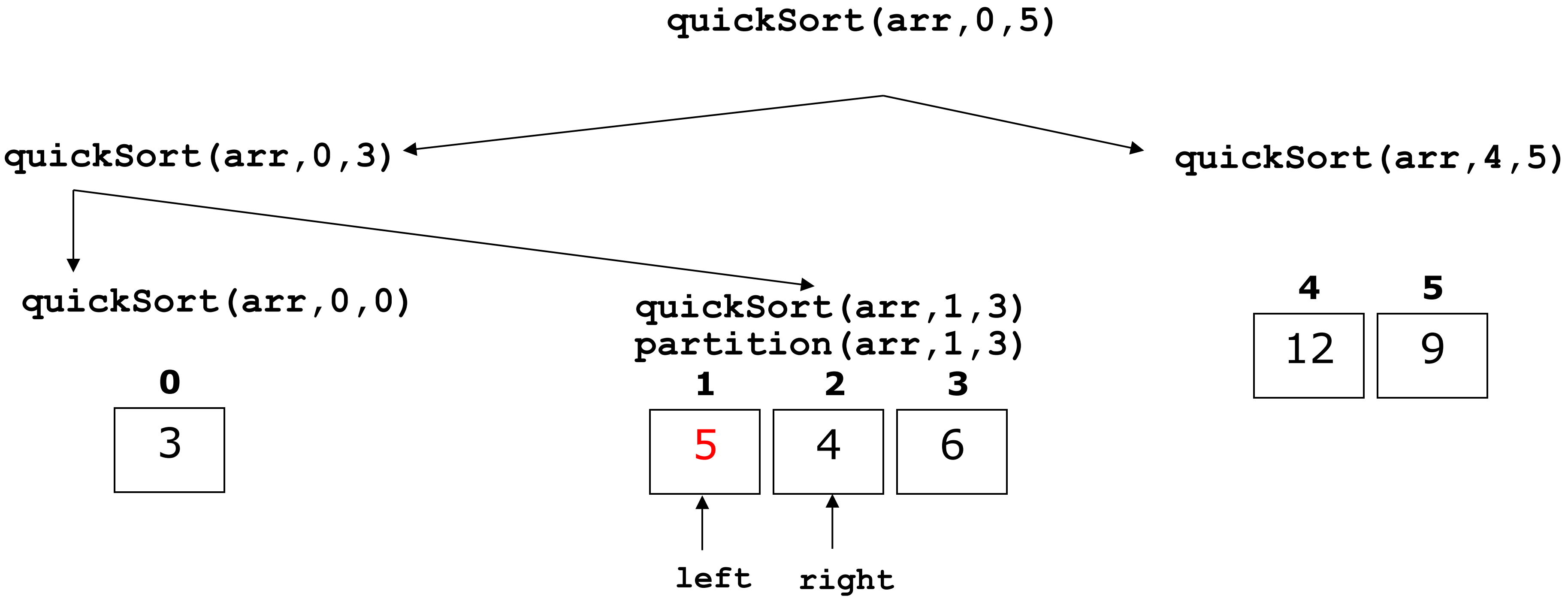
quickSort(arr,0,0)

0
3

quickSort(arr,1,3)
partition(arr,1,3)

1	2	3
5	4	6

4	5
12	9



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

0
3

quickSort(arr,1,3)
partition(arr,1,3)

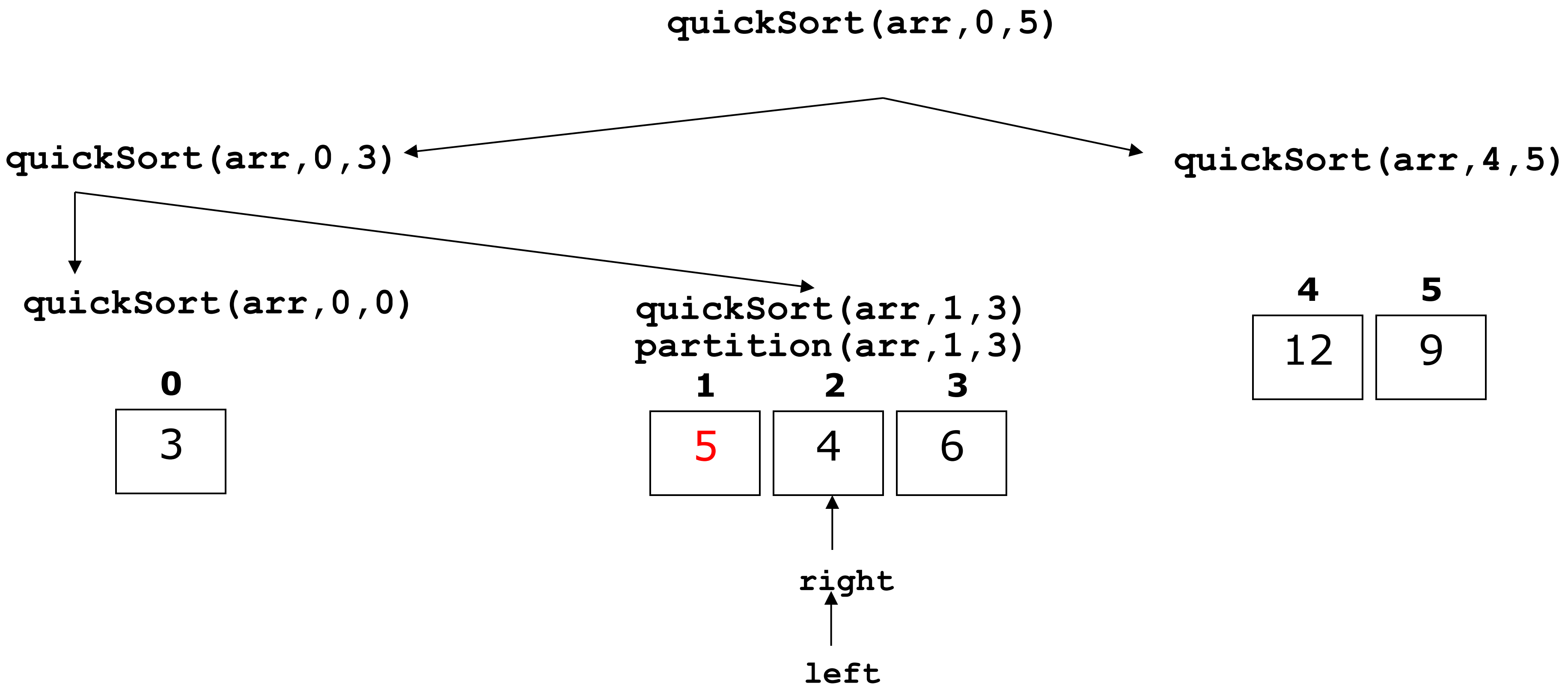
1	2	3
5	4	6

left

right

4	5
12	9

left moves to the right until
value that should be to right
of pivot...



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

0
3

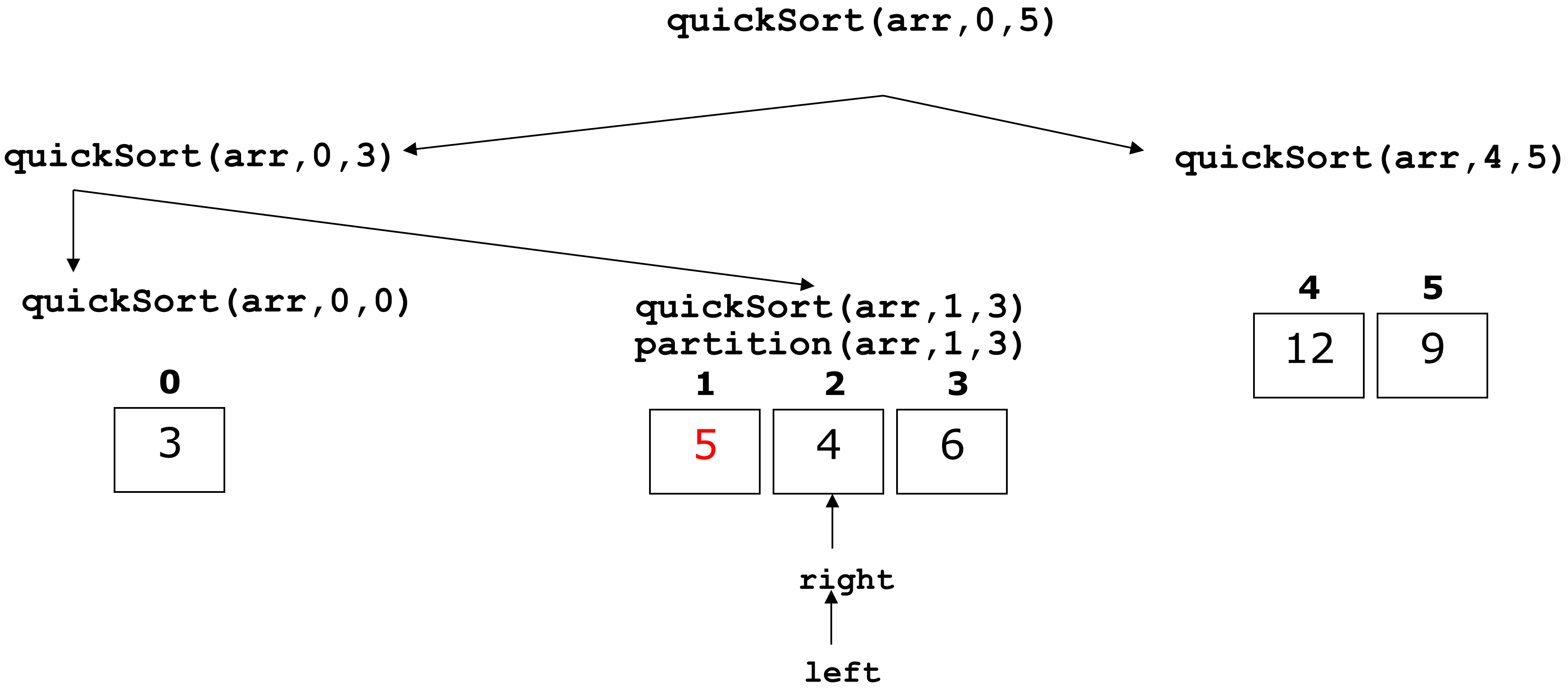
quickSort(arr,1,3)
partition(arr,1,3)

1	2	3
5	4	6

right

left

4	5
12	9



`quickSort(arr,0,5)`

`quickSort(arr,0,3)`

`quickSort(arr,4,5)`

4	5
12	9

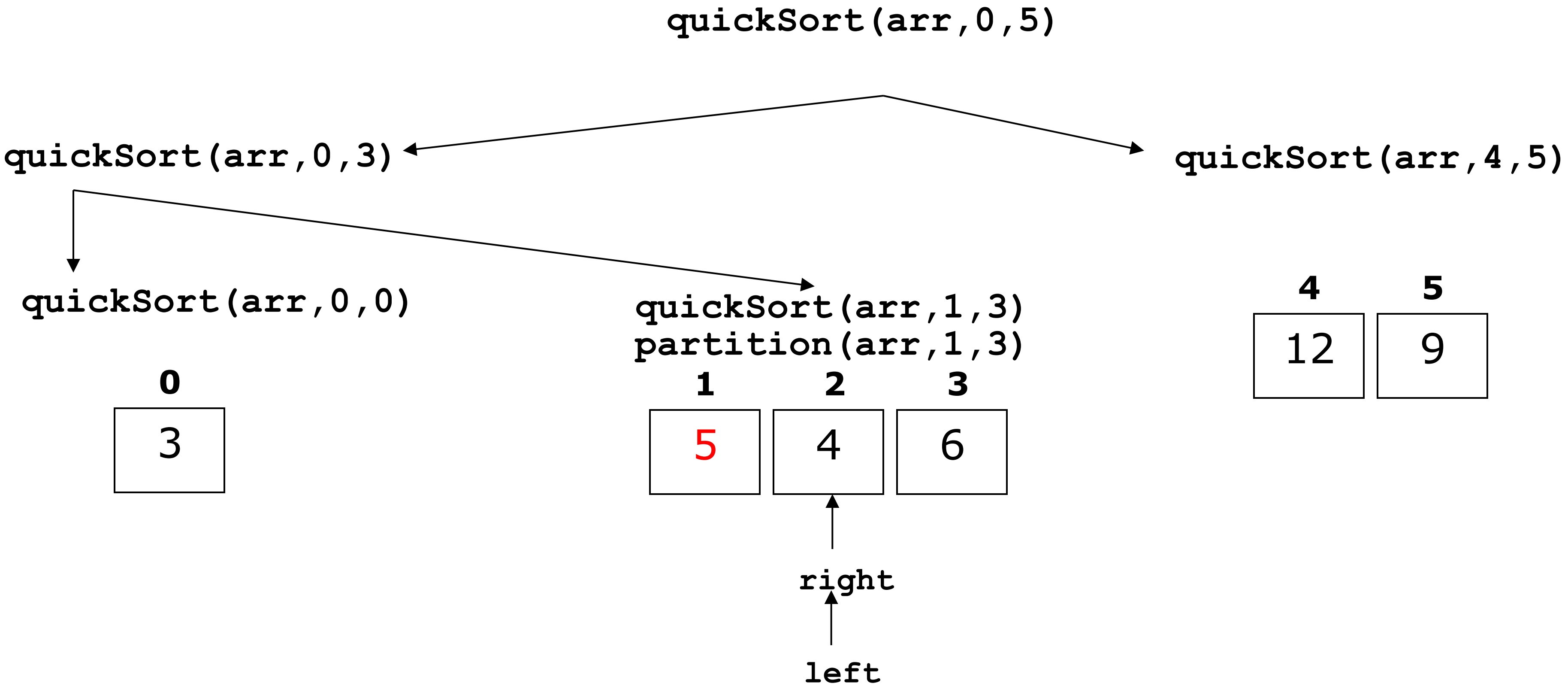
`quickSort(arr,0,0)`

0
3

`quickSort(arr,1,3)`
`partition(arr,1,3)`

1	2	3
5	4	6

right & left CROSS!



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

0

3

quickSort(arr,1,3)
partition(arr,1,3)

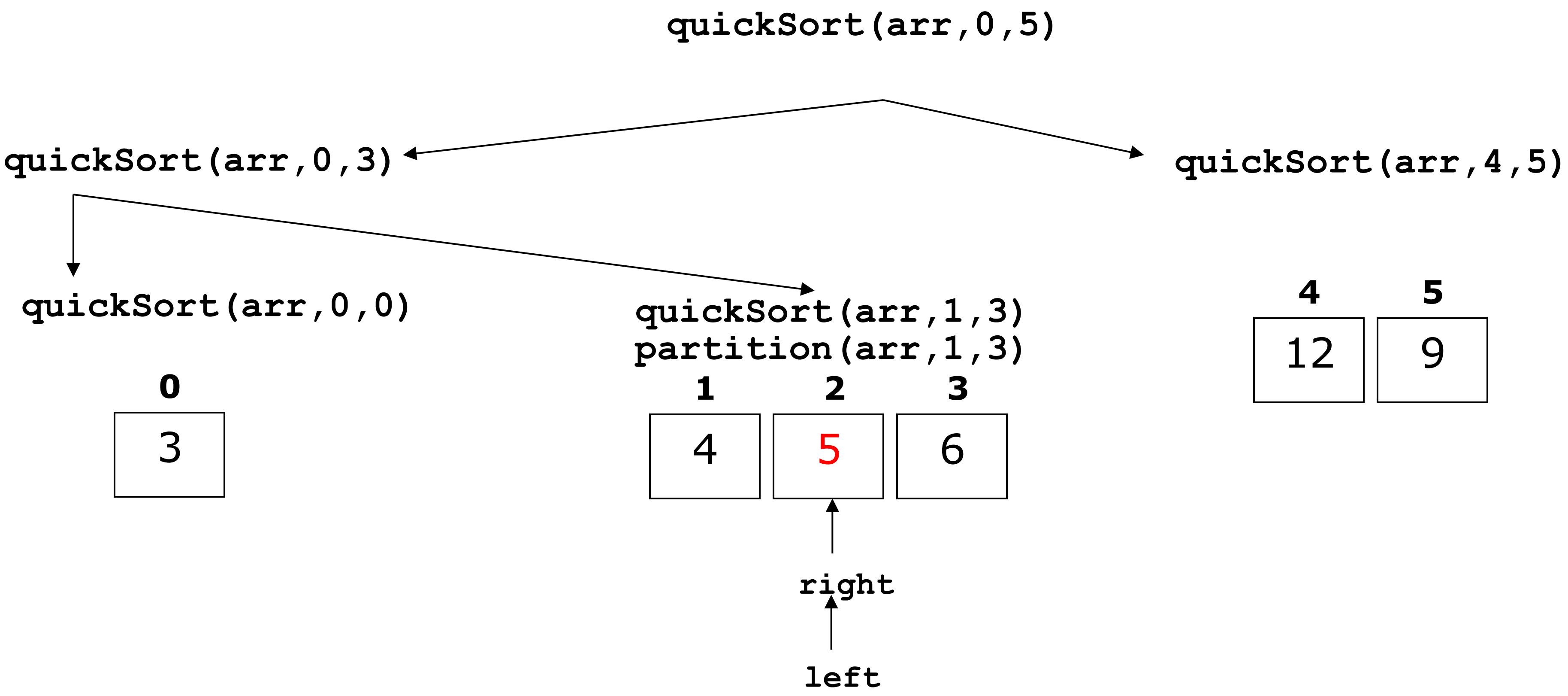
1	2	3
5	4	6

right

left

4	5
12	9

right & left CROSS!
1- swap pivot and arr[right]



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

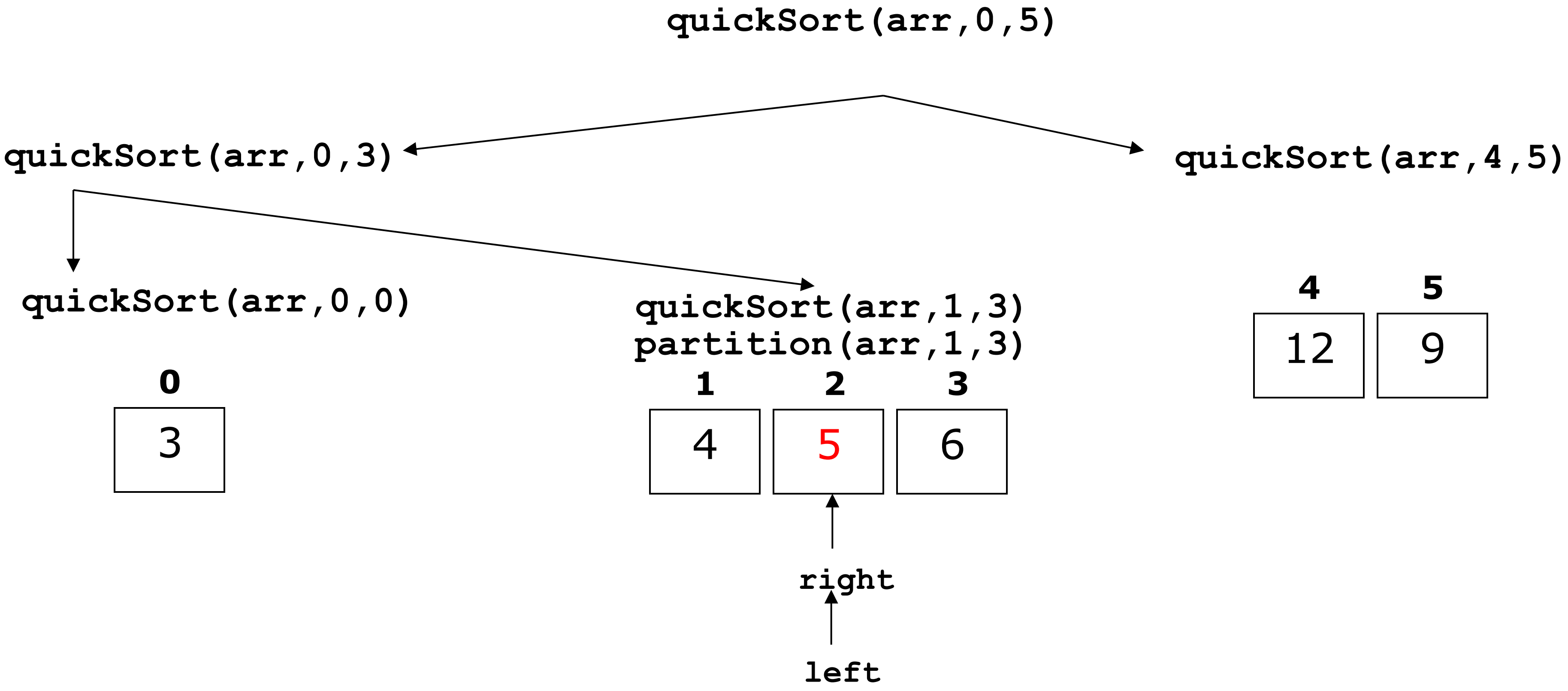
0
3

quickSort(arr,1,3)
partition(arr,1,3)

1	2	3
4	5	6

4	5
12	9

right & left CROSS!
1- swap pivot and arr[right]



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

0
3

quickSort(arr,1,3)
partition(arr,1,3)

1	2	3
4	5	6

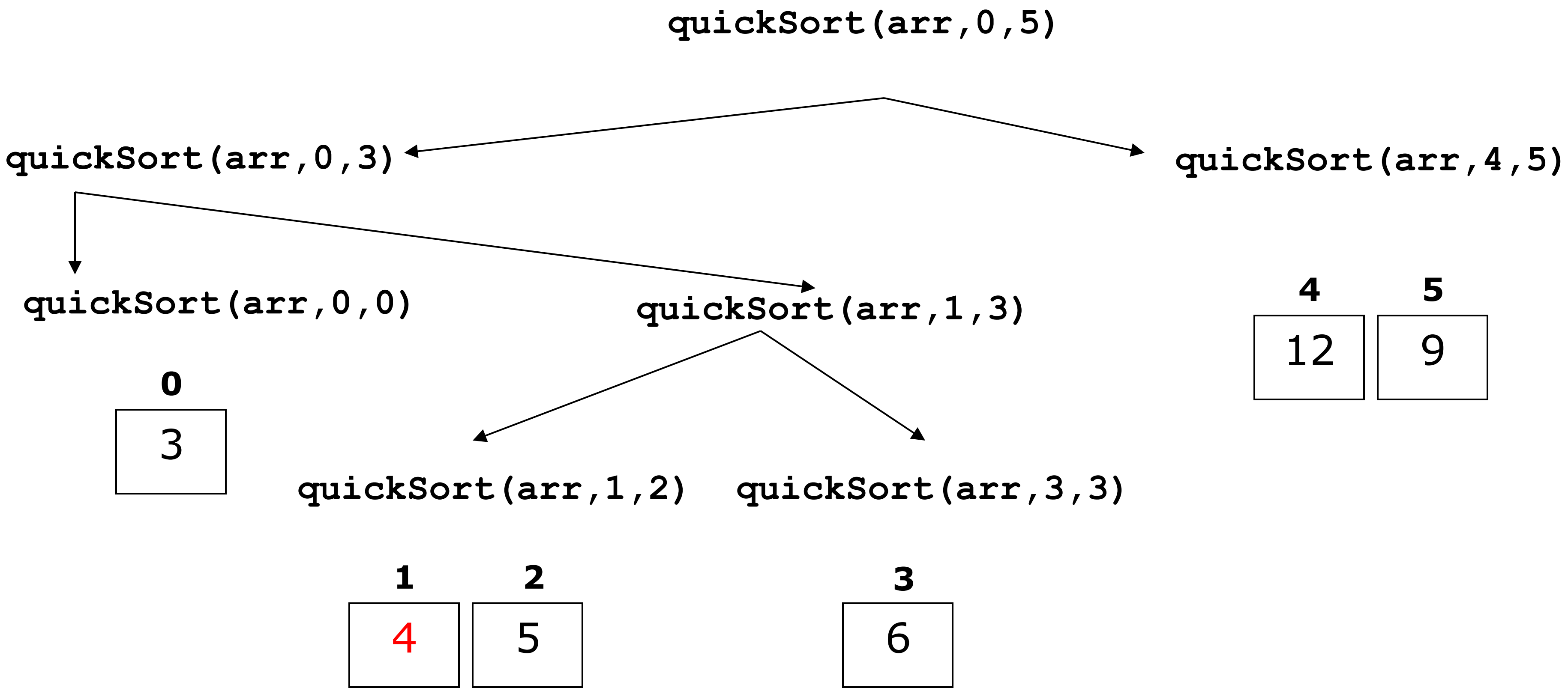
right

left

4	5
12	9

right & left CROSS!
1- swap pivot and arr[right]
2 – return new position of pivot

return 2



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

4	5
12	9

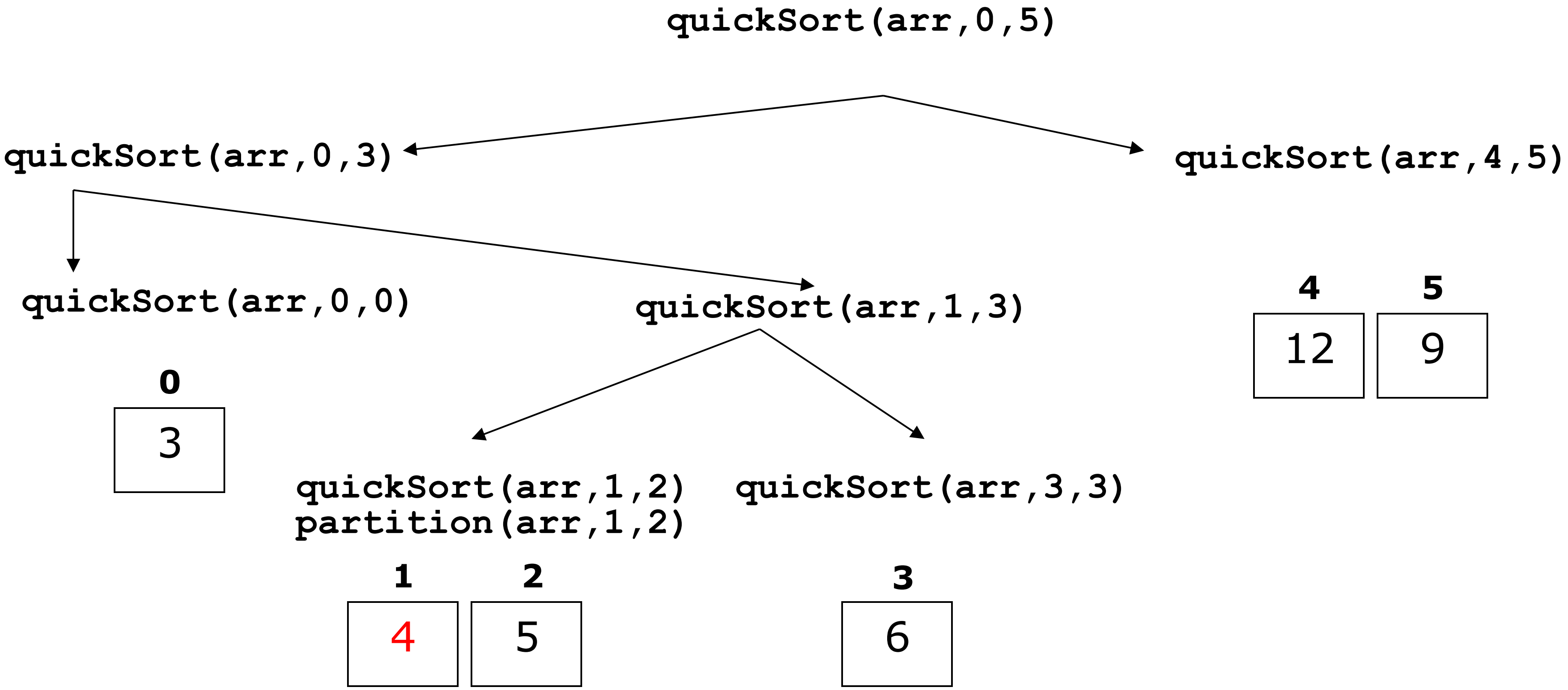
0
3

quickSort(arr,1,2)

quickSort(arr,3,3)

1	2
4	5

3
6



quickSort(arr,0,5)

quickSort(arr,0,3)

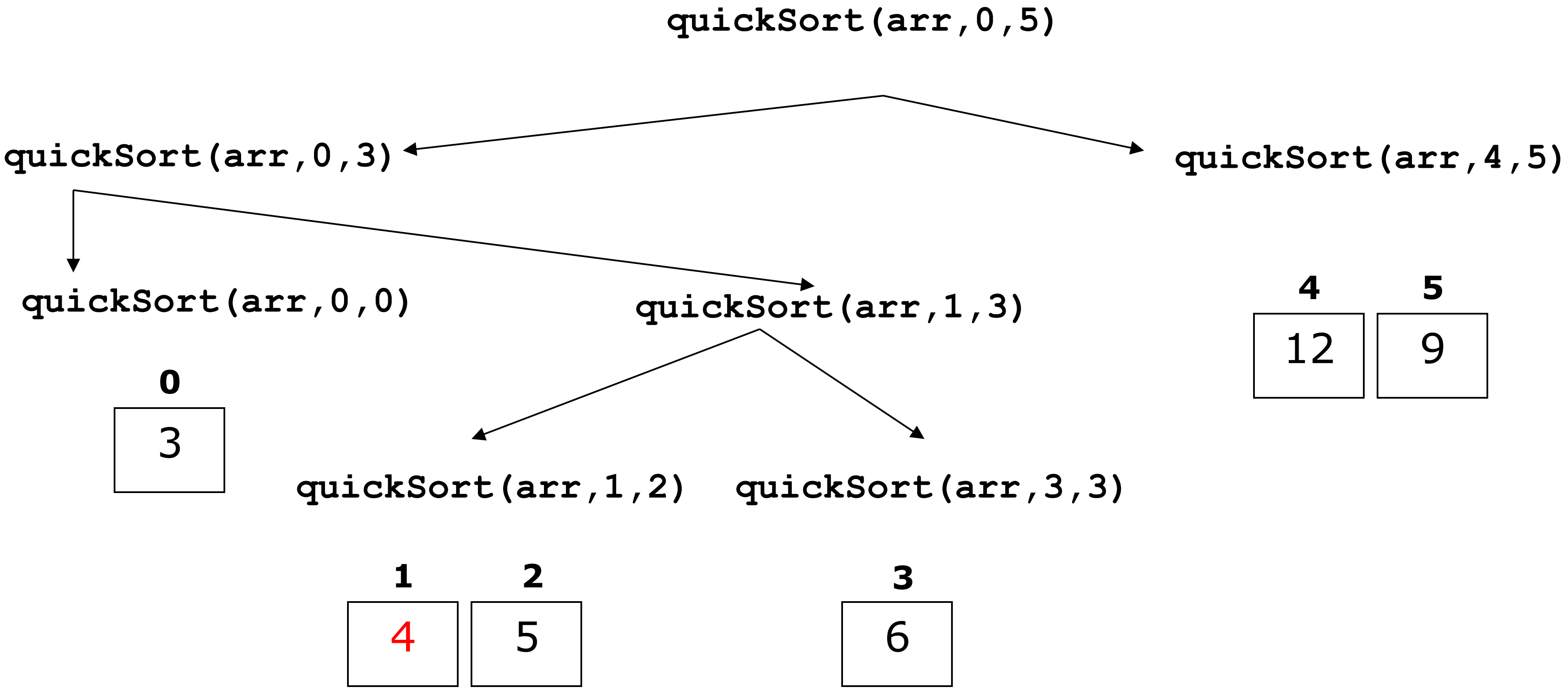
quickSort(arr,4,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

quickSort(arr,1,2)
partition(arr,1,2)

quickSort(arr,3,3)



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

quickSort(arr,1,2)

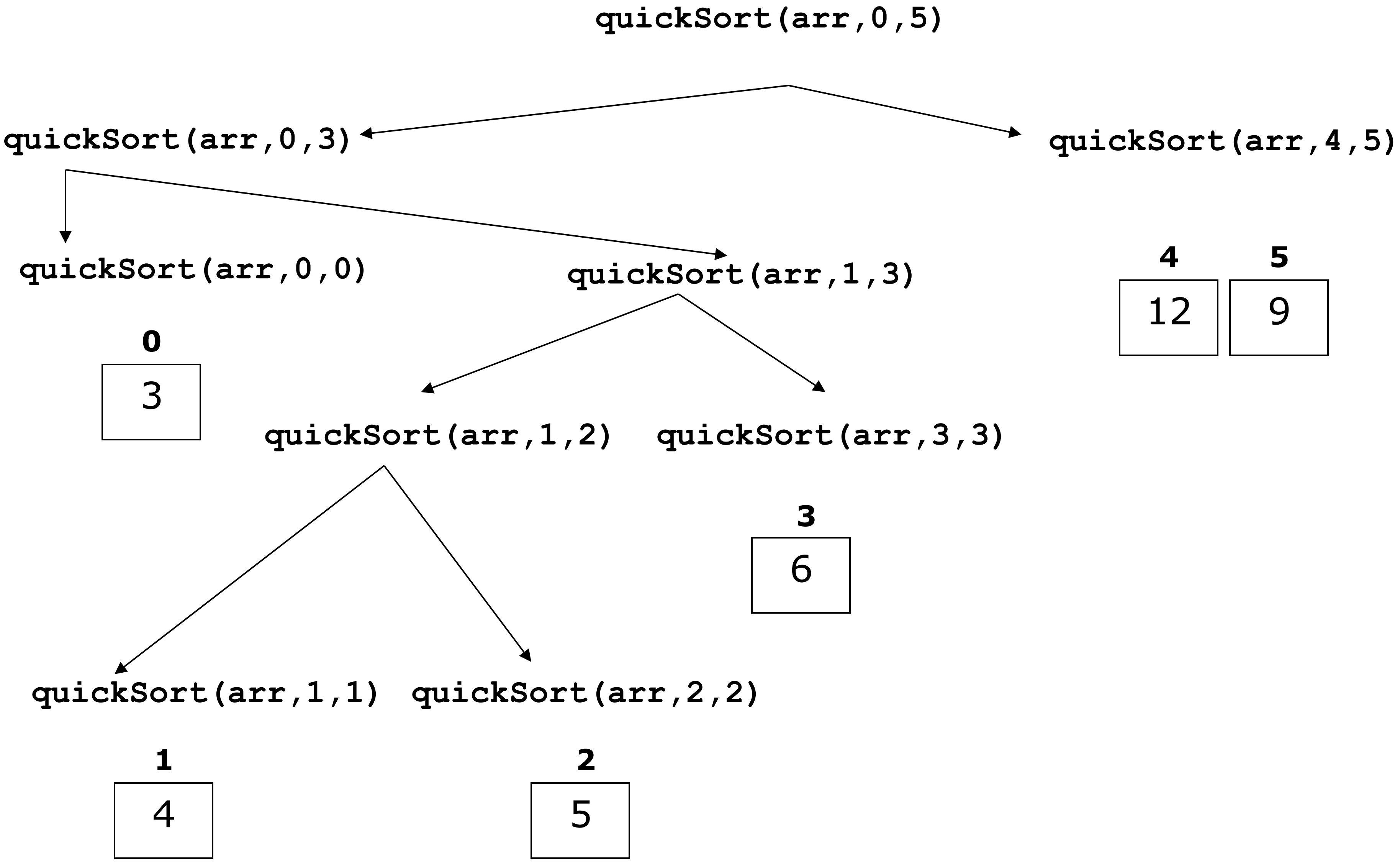
quickSort(arr,3,3)

0
3

1	2
4	5

3
6

4	5
12	9



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

4 5

12 9

0

3

quickSort(arr,1,2)

quickSort(arr,3,3)

3

6

quickSort(arr,1,1)

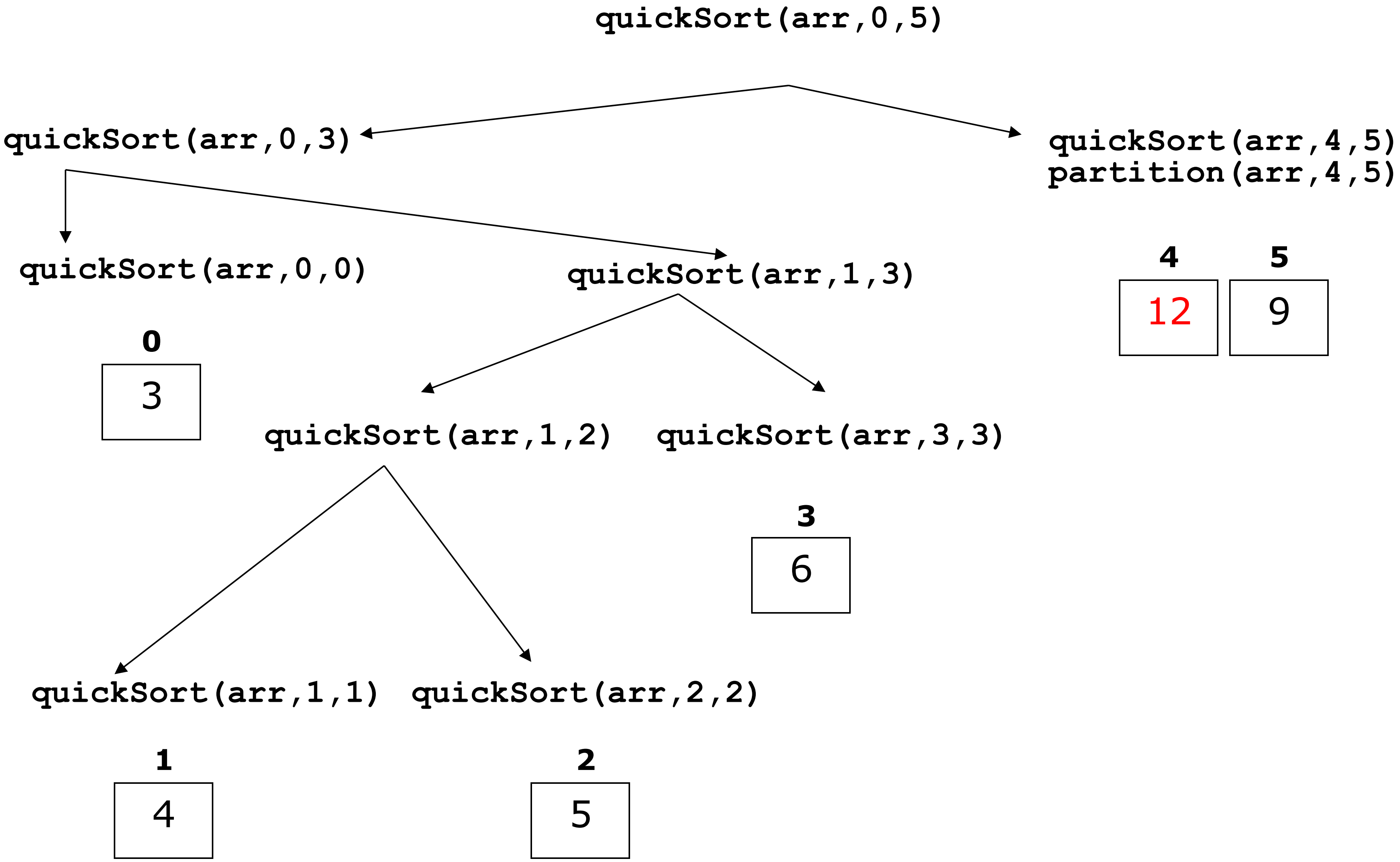
quickSort(arr,2,2)

1

4

2

5



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)
partition(arr,4,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

4	5
12	9

0
3

quickSort(arr,1,2)

quickSort(arr,3,3)

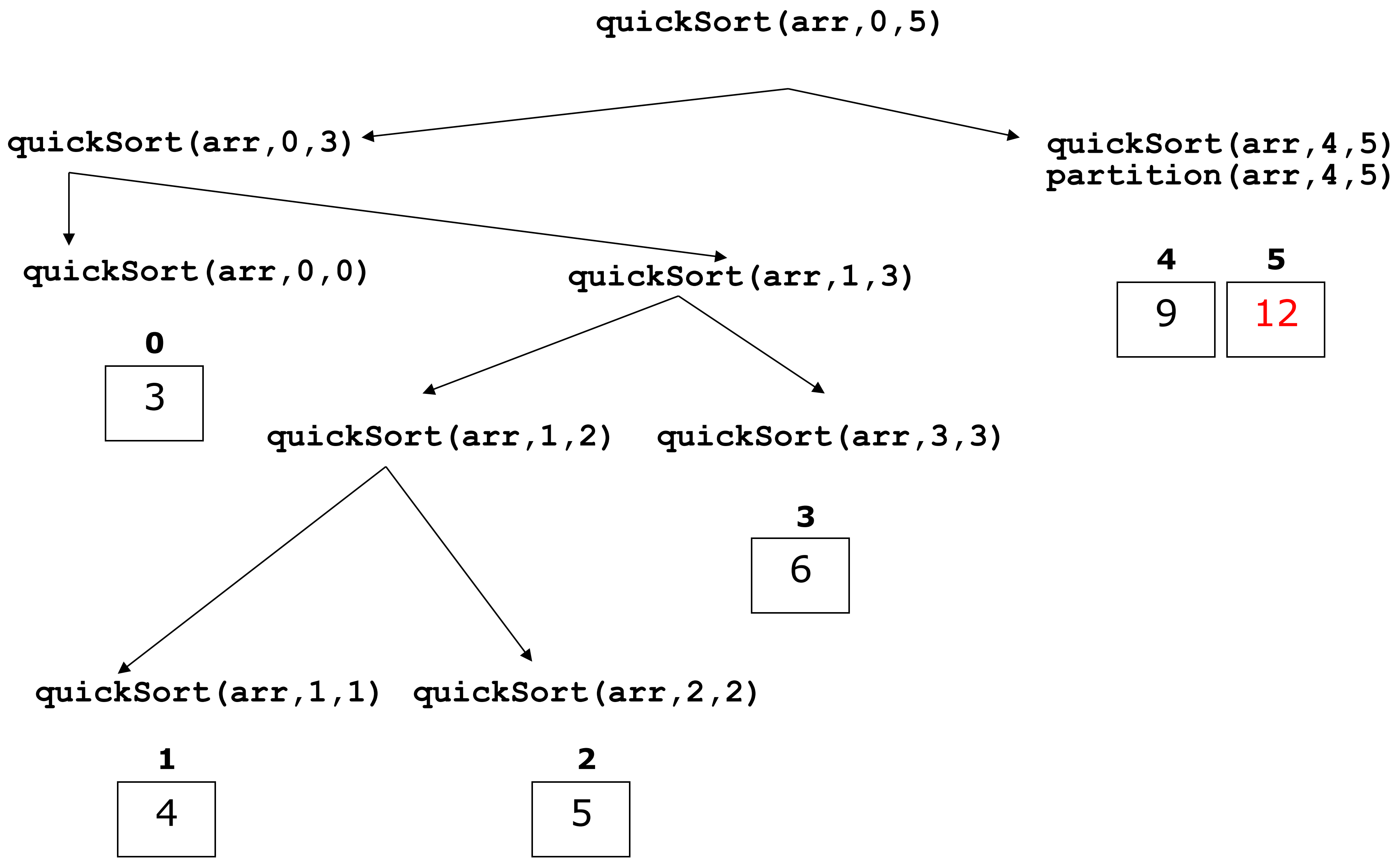
3
6

quickSort(arr,1,1)

quickSort(arr,2,2)

1
4

2
5



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)
partition(arr,4,5)

4	5
9	12

quickSort(arr,0,0)

0
3

quickSort(arr,1,3)

quickSort(arr,1,2)

quickSort(arr,3,3)

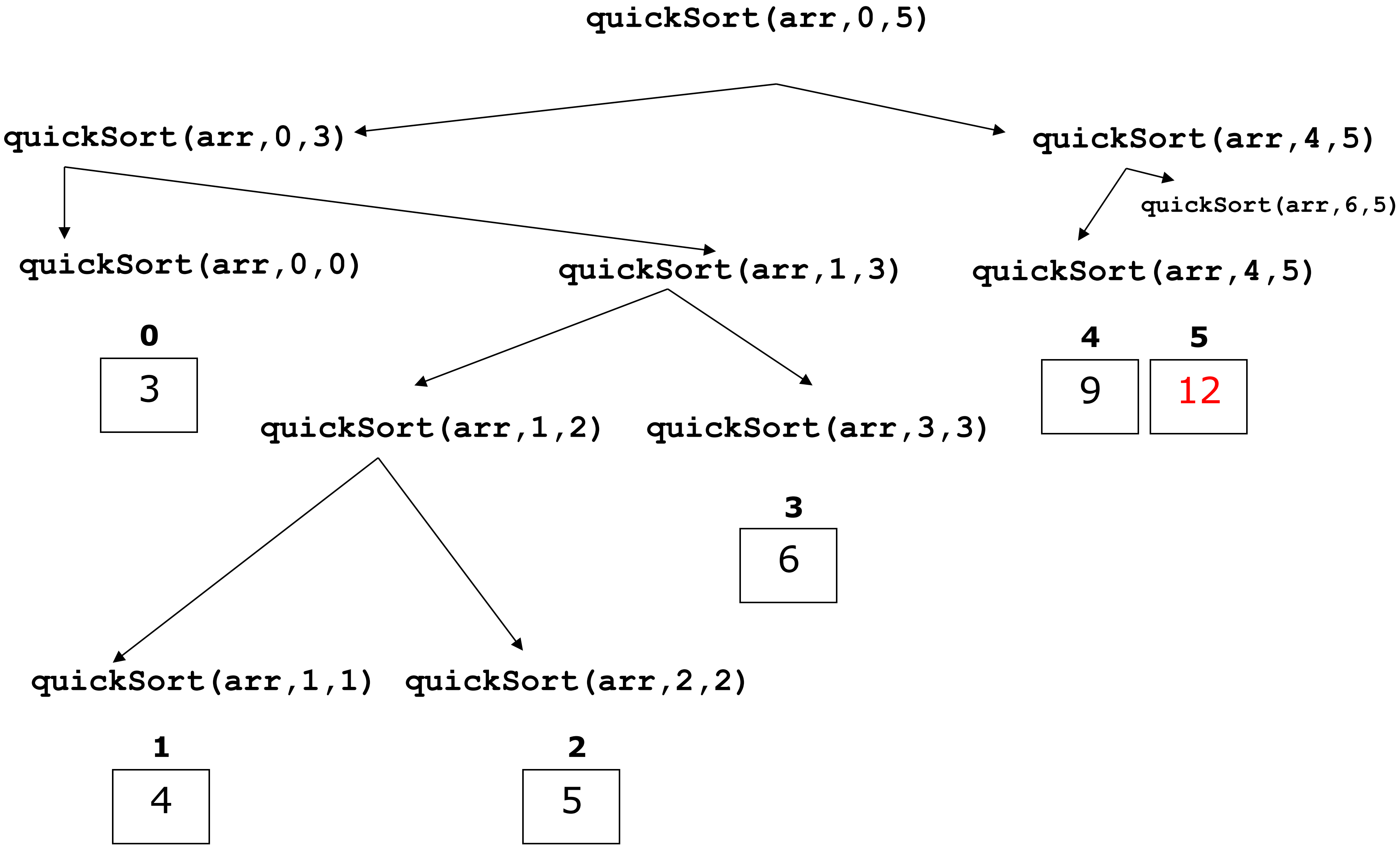
3
6

quickSort(arr,1,1)

quickSort(arr,2,2)

1
4

2
5



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,6,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

quickSort(arr,4,5)

0

3

4 5

9 12

quickSort(arr,1,2)

quickSort(arr,3,3)

3

6

quickSort(arr,1,1)

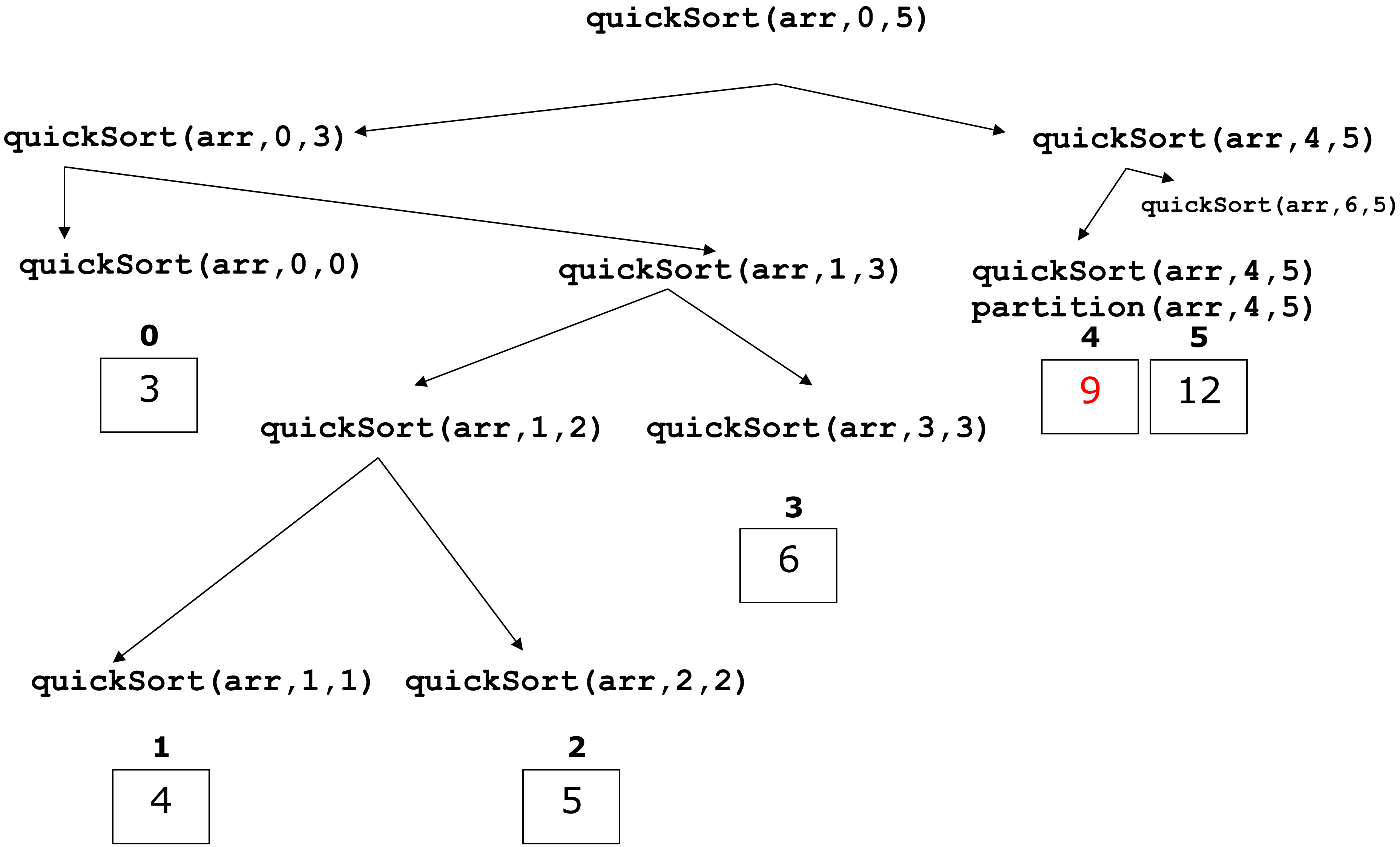
quickSort(arr,2,2)

1

4

2

5



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,6,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

quickSort(arr,4,5)
partition(arr,4,5)

0

3

4 5

9 12

quickSort(arr,1,2)

quickSort(arr,3,3)

3

6

quickSort(arr,1,1)

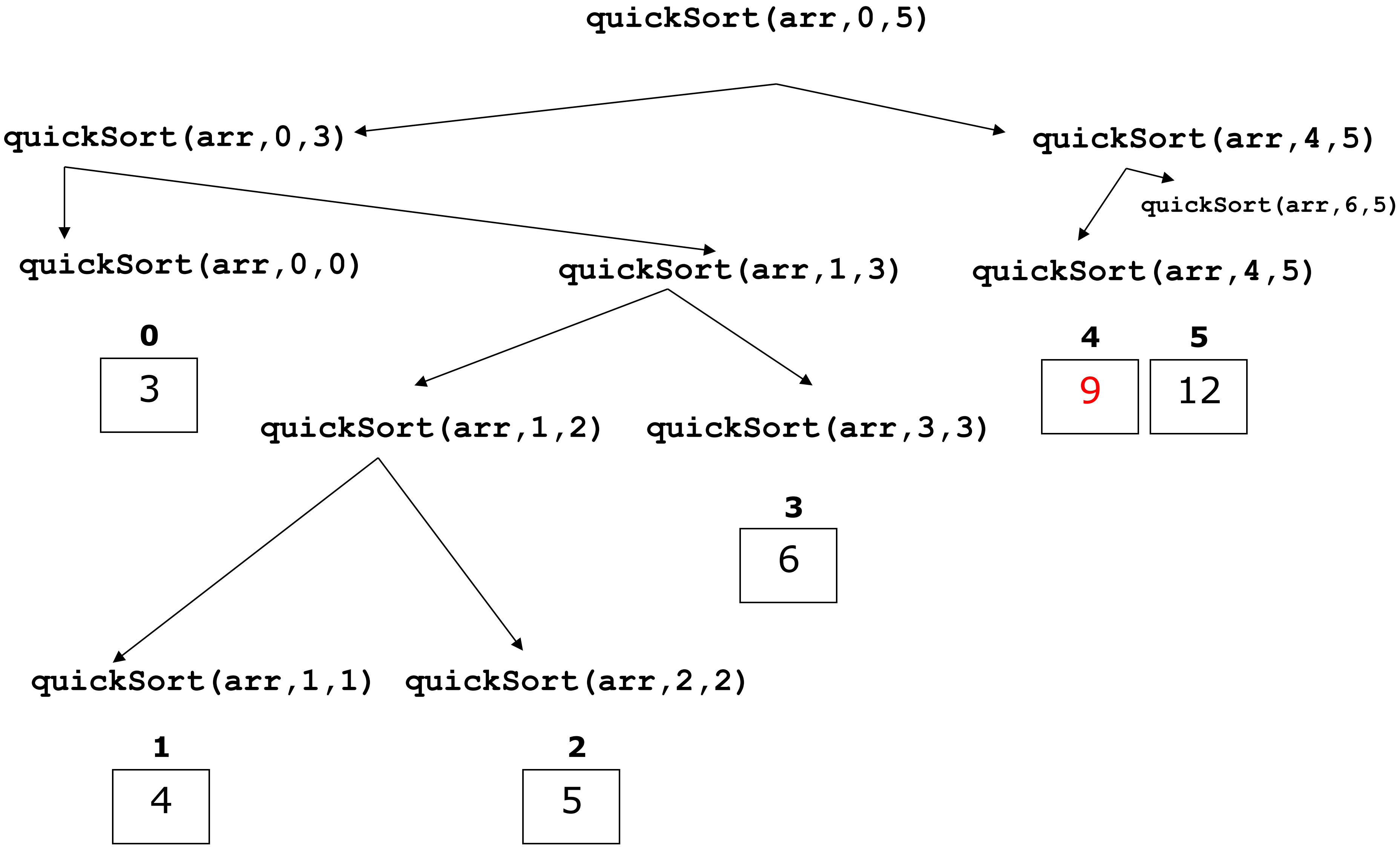
quickSort(arr,2,2)

1

4

2

5



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,6,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

quickSort(arr,4,5)

0
3

4	5
9	12

quickSort(arr,1,2)

quickSort(arr,3,3)

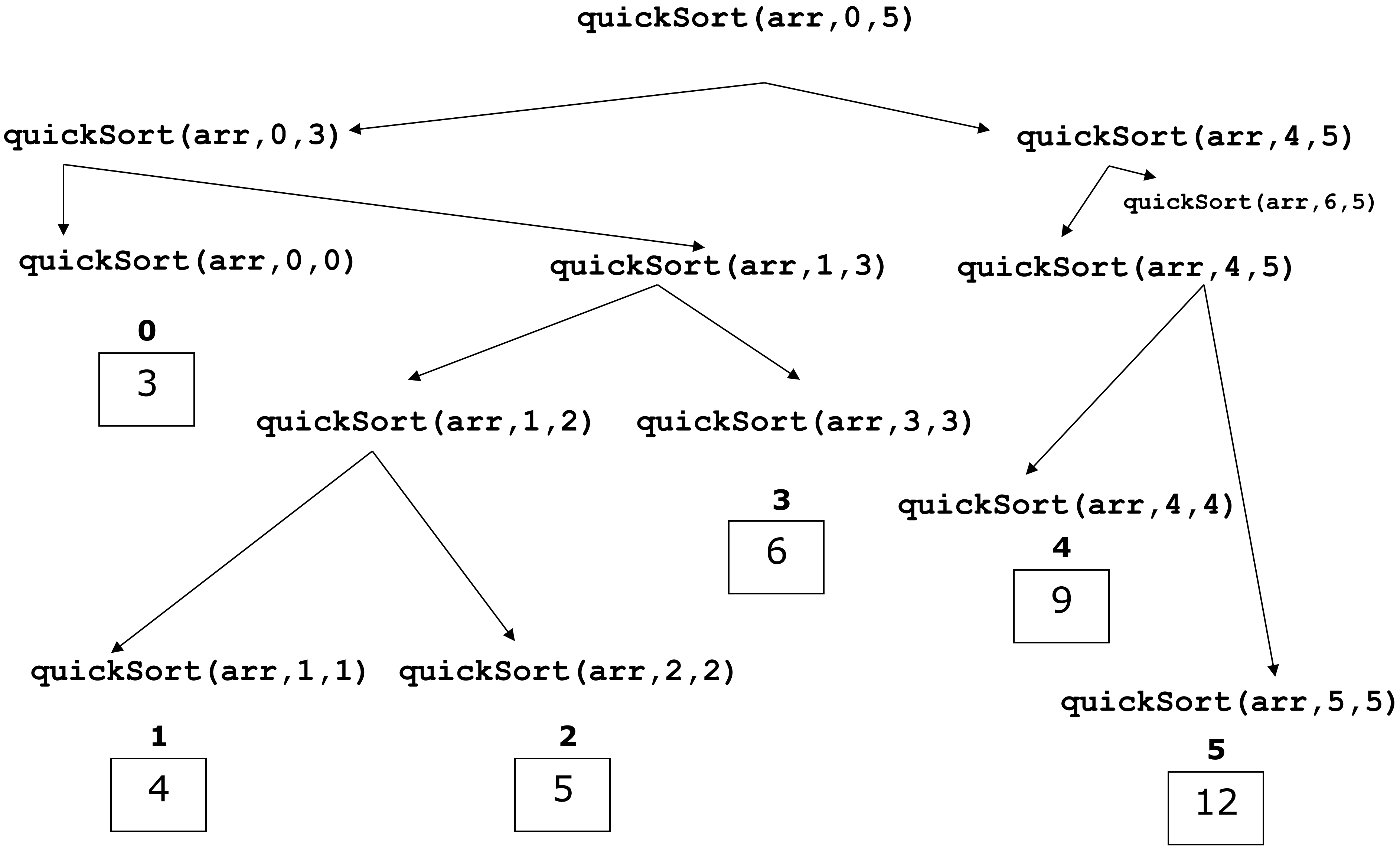
3
6

quickSort(arr,1,1)

quickSort(arr,2,2)

1
4

2
5



quickSort(arr,0,5)

quickSort(arr,0,3)

quickSort(arr,4,5)

quickSort(arr,6,5)

quickSort(arr,0,0)

quickSort(arr,1,3)

quickSort(arr,4,5)

0

3

quickSort(arr,1,2)

quickSort(arr,3,3)

3

6

quickSort(arr,4,4)

4

9

quickSort(arr,1,1)

quickSort(arr,2,2)

quickSort(arr,5,5)

1

4

2

5

5

12

```
quickSort(array, lower, upper)
{
   // Base Case
   if (lower >= upper)
   {
      we’re done
   }
   else
   {
     partition array around pivot value array[lower]
     pos contains the new location of pivot value
     quickSort array up to pos: quickSort(array,lower,pos)
     quickSort array after pos: quickSort(array,pos+1,upper)
   }
}
```

```
partition(array, lower, upper)
{
   pivot is array[lower]
   while (true)
   {
      scan from right to left using index called RIGHT
      STOP when locate an element that should be left of pivot

      scan from left to right using index called LEFT
      stop when locate an element that should be right of pivot

      swap array[RIGHT] and array[LEFT]

      if (RIGHT and LEFT cross)
         pos = location where LEFT/RIGHT cross
         swap pivot and array[pos]
         all values left of pivot are <= pivot
         all values right of pivot are >= pivot
         return pos
      end pos
   }
}
```