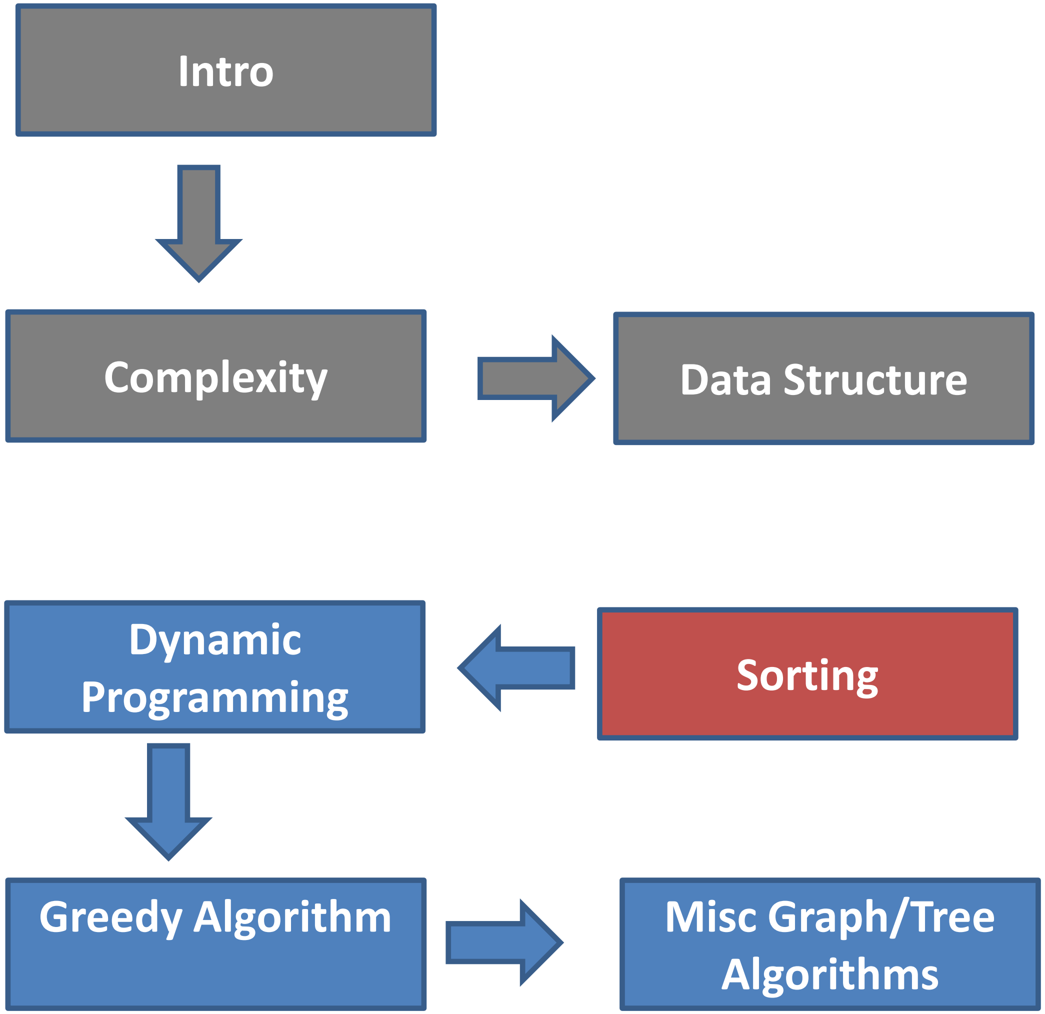
An Introduction to Algorithms

By

Hossein Rahmani

h_rahmani@iust.ac.ir

http://webpages.iust.ac.ir/h_rahmani/


Intro
Complexity
Data Structure
Trees
Hash Functions
Sorting
Dynamic Programming
Greedy Algorithm
Misc Graph/Tree Algorithms

Sorting Review

- Insertion Sort
  – $T(n) = \Theta(n^2)$
- Selection Sort
  - $T(n) = \Theta(n^2)$
- Merge Sort
  - $T(n) = \Theta(n \lg(n))$
- Heap Sort
  – $T(n) = \Theta(n \lg(n))$

Seems pretty good.
Can we do better?

Sorting

- Assumptions
    1. <u>No (Prior) knowledge</u> of the keys or numbers we are sorting on.
    2. Each key supports a <u>comparison</u> interface or operator.
    3. Each key is <u>unique</u> (just for convenience).

Comparison Sorting

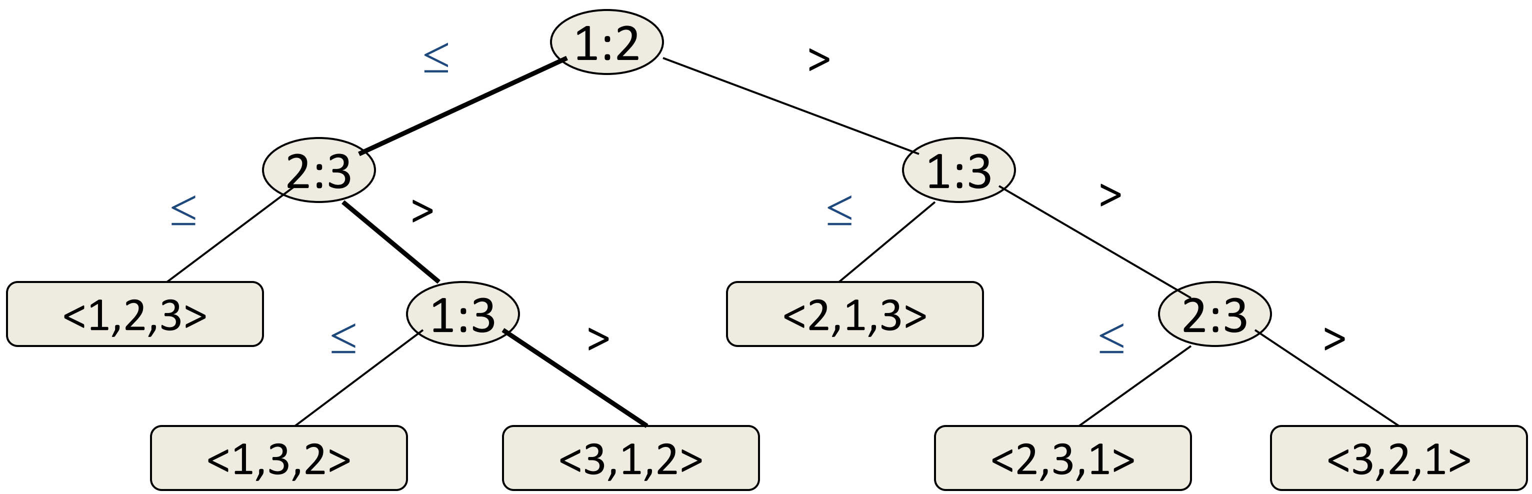
Comparison Sorting

- Given a set of *n* values, there can be *n!* permutations of these values.
- So if we look at the behavior of the sorting algorithm over all possible *n!* inputs we can determine the worst-case complexity of the algorithm.

Decision Tree

- Decision tree model
  - Full binary tree
    - A **proper binary tree** (or **2-tree**) is a tree in which every node other than the leaves has two children
  - Internal node represents a comparison.
    - Ignore control, movement, and all other operations, just see comparison
  - Each leaf represents one possible result (a permutation of the elements in sorted order).
  - The height of the tree (i.e., longest path) is the lower bound.

Decision Tree Model



Internal node i:j indicates comparison between a_i and a_j.
suppose three elements $< a1, a2, a3>$ with instance <6,8,5>
Leaf node $<\pi(1), \pi(2), \pi(3)>$ indicates ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq a_{\pi(3)}$.
Path of **bold lines** indicates sorting path for <6,8,5>.
There are total 3!=6 possible permutations (paths).

Decision Tree Model

- The longest path is the worst case number of comparisons. The length of the longest path is the height of the decision tree.
- **Theorem 8.1**: Any comparison sort algorithm requires $\Omega(n\lg n)$ comparisons in the worst case.
- **Proof:**
  - Suppose height of a decision tree is h, and number of paths (i,e,, permutations) is $n!$.
  - Since a binary tree of height h has at most 2^h leaves,
    - $n! \leq 2^h$, so $h \geq \lg(n!) \geq \Omega(n\lg n)$
- That is to say: **any comparison sort in the worst case needs at least *n*lg *n* comparisons.**

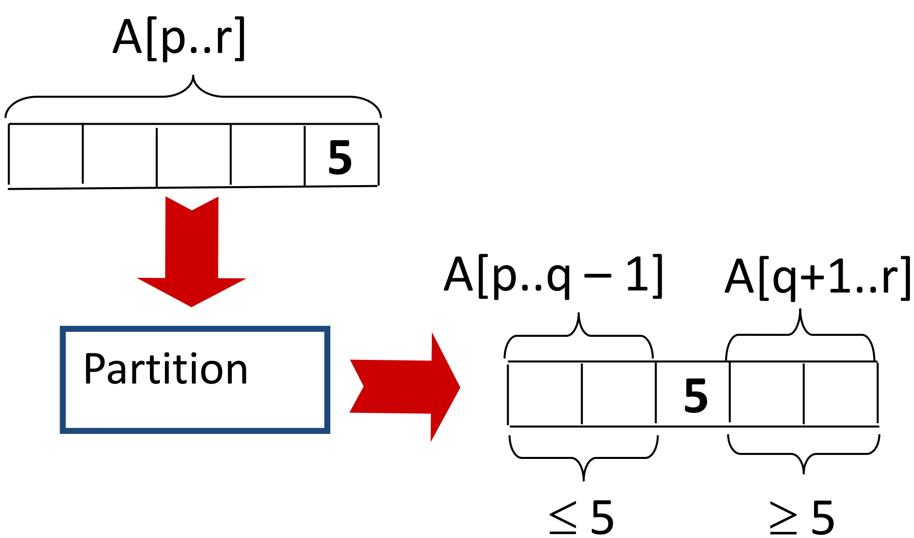
QuickSort Design

- Follows the **divide-and-conquer** paradigm.
- ***Divide***: Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$.
  - Each element in $A[p..q-1] < A[q]$.
  - $A[q] <$ each element in $A[q+1..r]$.
  - Index q is computed as part of the partitioning procedure.
- ***Conquer***: Sort the two subarrays by recursive calls to quicksort.
- ***Combine***: The subarrays are sorted in place – no work is needed to combine them.
- How do the divide and combine steps of quicksort compare with those of merge sort?

Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
```

```
Partition(A, p, r)
    x := A[r],
    i := p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```



Example

p r

initially: 2 5 8 3 9 4 1 7 10 **6** **note:** pivot (x) = 6

i j

next iteration: 2 5 8 3 9 4 1 7 10 **6**

i j

next iteration: 2 5 8 3 9 4 1 7 10 **6**

i j

next iteration: 2 5 8 3 9 4 1 7 10 **6**

i j

next iteration: 2 5 3 8 9 4 1 7 10 **6**

i j

```
Partition(A, p, r)
    x := A[r],
    i := p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

Example (Continued)

next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i		j					
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i			j				
next iteration:	2	5	3	4	9	8	1	7	10	**6**
				i			j			
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i			j		
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i				j	
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i					j
after final swap:	2	5	3	4	1	**6**	9	7	10	8
					i					j

Partition(A, p, r)

$x := A[r]$,

$i := p - 1$;

for $j := p$ **to** $r - 1$ **do**

if $A[j] \leq x$ **then**

$i := i + 1$;

$A[i] \leftrightarrow A[j]$

$A[i + 1] \leftrightarrow A[r]$;

return $i + 1$

Partitioning

- Select the last element A[r] in the subarray $A[p..r]$ as the *pivot* – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.
  1. $A[p..i\,]$ — All entries in this region are **$<$ *pivot***.
  2. $A[i+1..j-1]$ — All entries in this region are **$>$ *pivot***.
  3. $A[r] = pivot$.
  4. $A[j..r-1]$ — Not known how they compare to *pivot*.

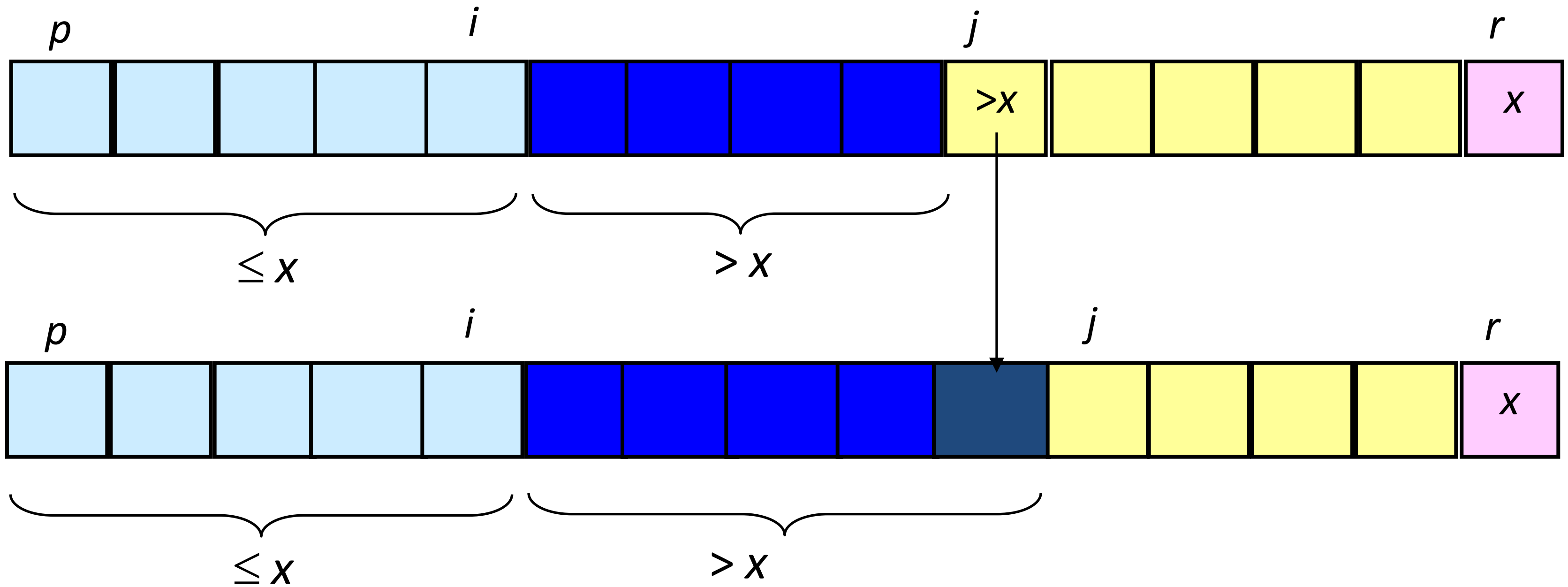
Correctness of Partition

- **Initialization:**
  - Before first iteration
    - $A[p..i]$ and $A[i+1..j-1]$ are empty – Conds. 1 and 2 are satisfied (trivially).
    - r is the index of the *pivot*
      - Cond. 3 is satisfied.
- **Maintenance:**
  - **Case 1:** $A[j] > x$
    - Increment j only.

```
Partition(A, p, r)
    x := A[r],
    i := p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

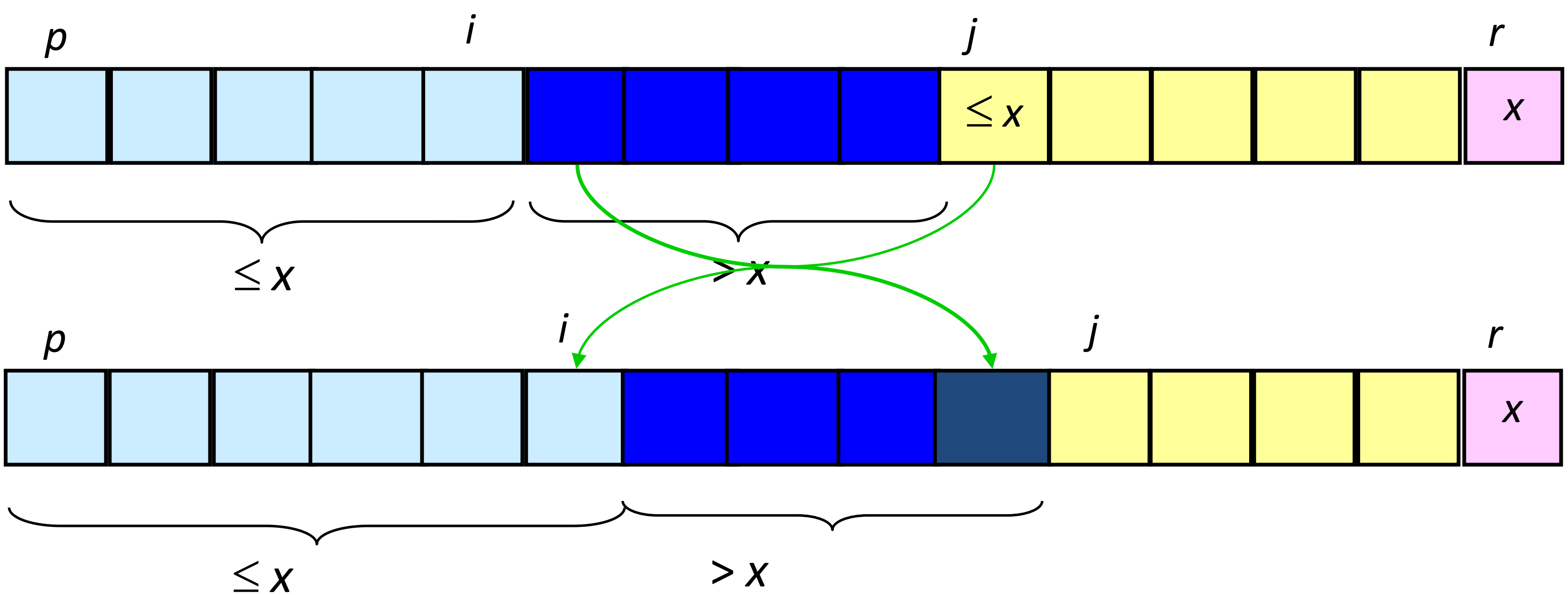
Correctness of Partition

Case 1:



Correctness of Partition

- **Case 2:** $A[j] \leq x$
  - Increment i
  - Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
  - Increment j
    - Condition 2 is maintained.
  - $A[r]$ is unaltered.
    - Condition 3 is maintained.



Correctness of Partition

- **Termination:**
  - When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p..i] \leq$ ***pivot***
    - $A[i+1..j-1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- The last two lines swap $A[i+1]$ and $A[r]$.
  - *Pivot* moves from the end of the array to between the two subarrays.
  - Thus, procedure *partition* correctly performs the divide step.

Complexity of Partition

- PartitionTime(n) is given by the number of iterations in the *for* loop.
- $\Theta(n)$: $n = r - p + 1$.

```
Partition(A, p, r)
    x := A[r],
    i := p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

Quicksort Overview

- To sort a[left...right]:

1. if left < right:
    1.1. Partition a[left...right] such that:

       all a[left...p-1] are less than a[p], and

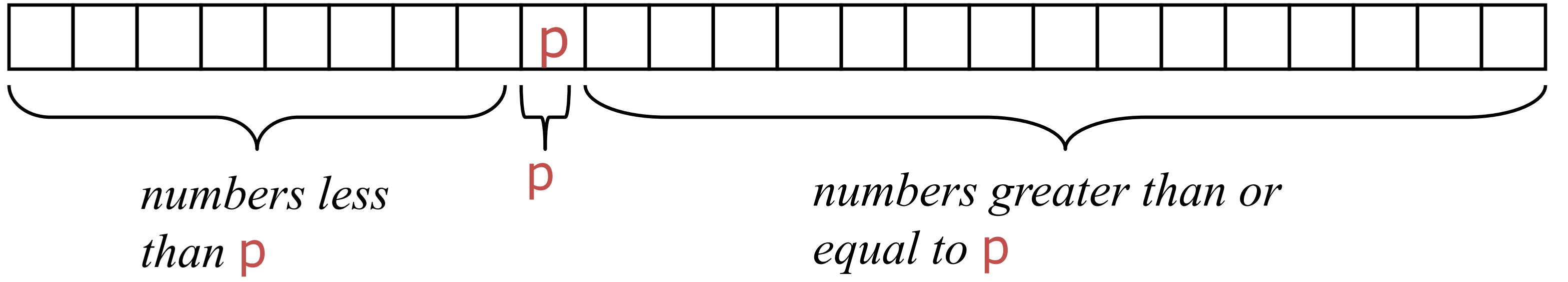
       all a[p+1...right] are >= a[p]

    1.2. Quicksort a[left...p-1]

    1.3. Quicksort a[p+1...right]
2. Terminate

Partitioning in Quicksort

- A key step in the Quicksort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



Alternative Partitioning

- Choose an array value (say, the first) to use as the pivot
- Starting from the left end, find the first element that is greater than or equal to the pivot
- Searching backward from the right end, find the first element that is less than the pivot
- Interchange (swap) these two elements
- Repeat, searching from where we left off, until done

Alternative Partitioning

- To partition a[left...right]:

1. Set pivot = a[left], l = left + 1, r = right;
2. while l < r, do
    2.1. while l < right & a[l] < pivot , set l = l + 1
    2.2. while r > left & a[r] >= pivot , set r = r - 1
    2.3. if l < r, swap a[l] and a[r]
3. Set a[left] = a[r], a[r] = pivot
4. Terminate

Example of partitioning

- choose pivot: <u>4</u> 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: <u>4</u> 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- swap: <u>4</u> 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: <u>4</u> 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- swap: <u>4</u> 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: <u>4</u> 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- swap: <u>4</u> 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- search: <u>4</u> 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- swap with pivot: 1 3 3 1 2 2 3 <u>4</u> 4 9 8 9 6 5 6

Partition Implementation (Java)

```
static int Partition(int[] a, int left, int right) {
    int p = a[left], l = left + 1, r = right;
    while (l < r) {
        while (l < right && a[l] < p)   l++;
        while (r > left && a[r] >= p)   r--;
        if (l < r) {
            int temp = a[l]; a[l] = a[r]; a[r] = temp;
        }
    }
    a[left] = a[r];
    a[r] = p;
    return r;
}
```

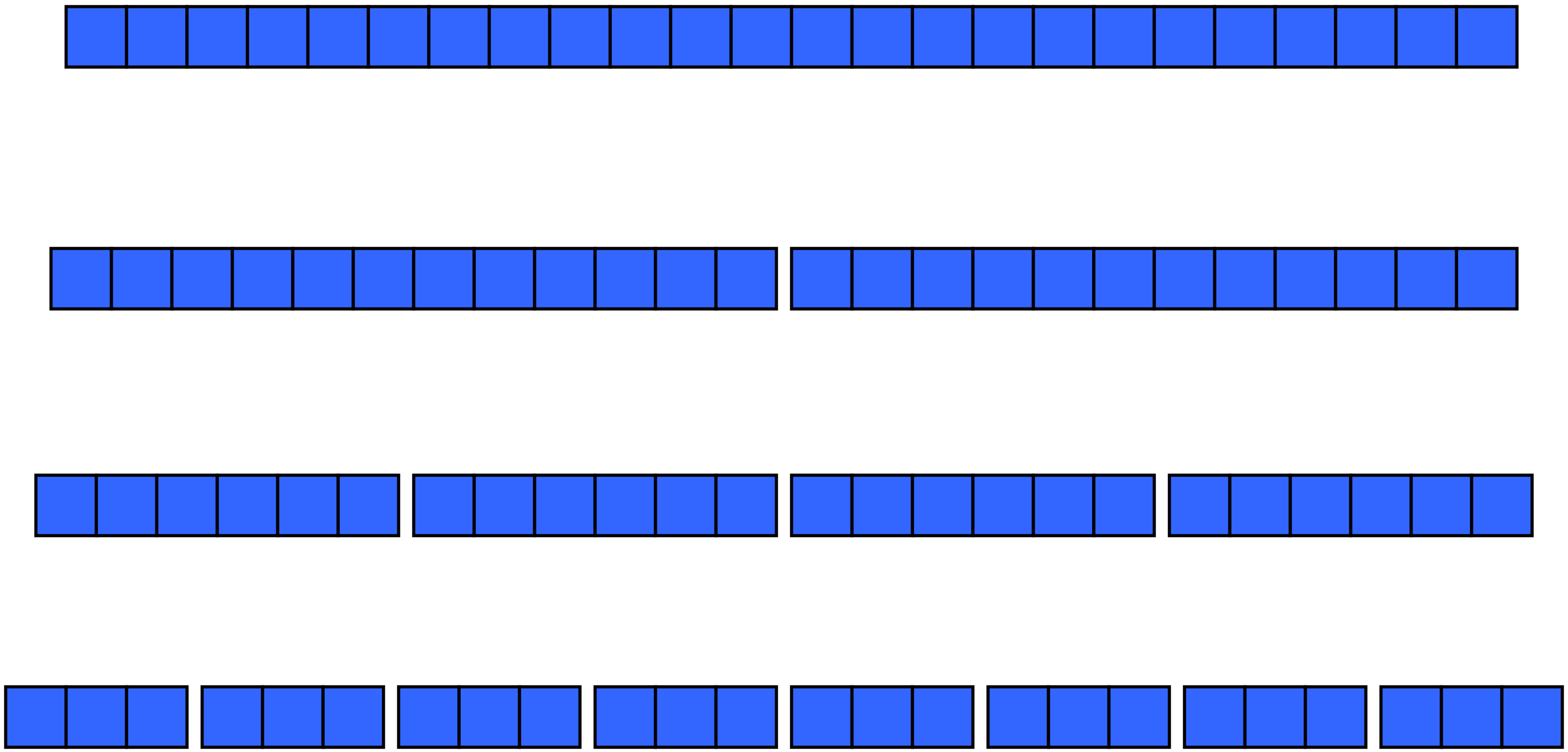
Quicksort Implementation (Java)

```
static void Quicksort(int[] array, int left, int right) {
    if (left < right) {
        int p = Partition(array, left, right);
        Quicksort(array, left, p - 1);
        Quicksort(array, p + 1, right);
    }
}
```

Analysis of quicksort—best case

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion in $\log_2 n$
  - Because that's how many times we can halve n
- We note that
  - Each partition is linear over its subarray
  - All the partitions at one level cover the array

Partitioning at various levels



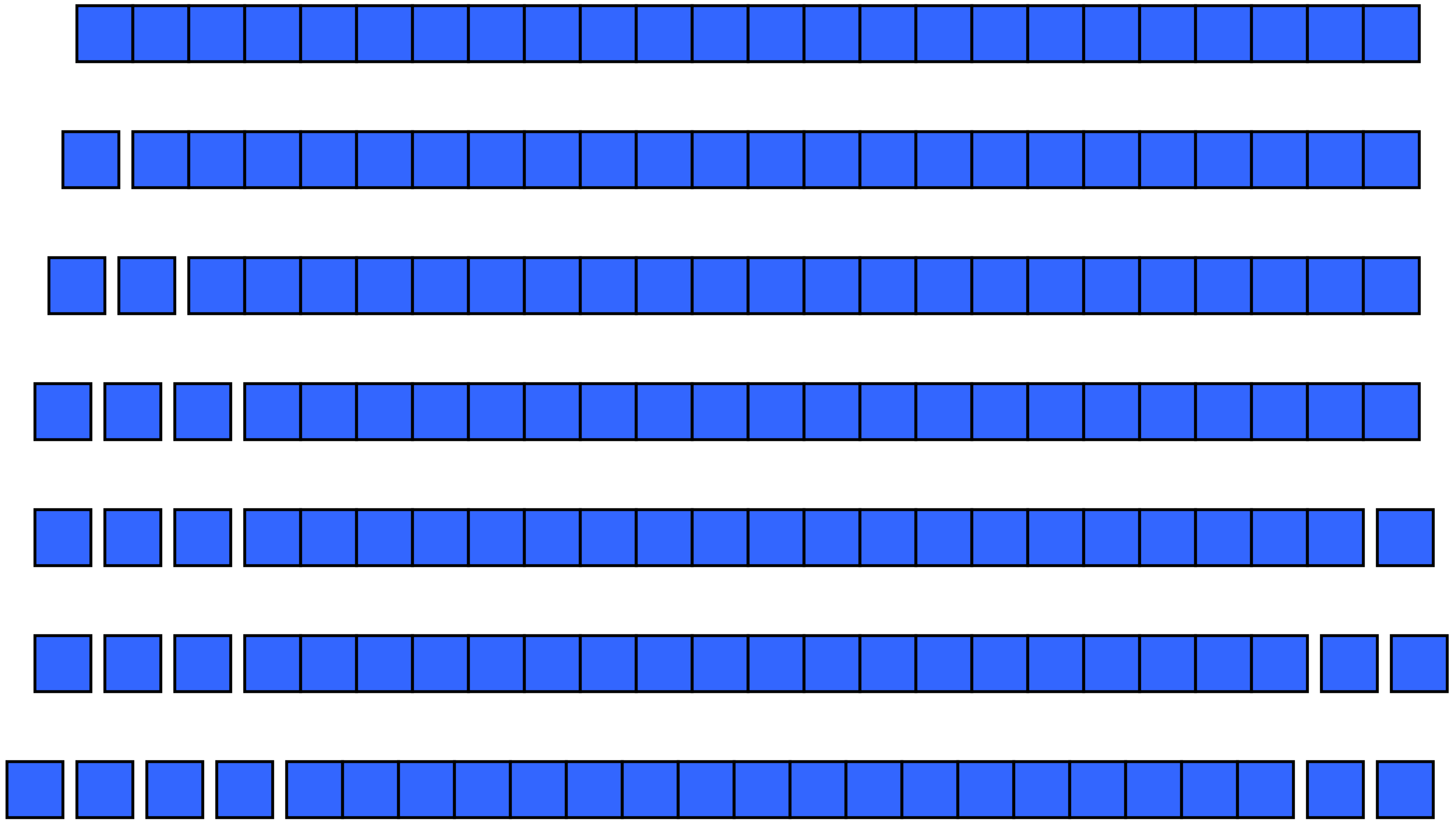
Best Case Analysis

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the best case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?

Worst case

- In the worst case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length `n-1` part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length `n-1` part requires (in the worst case) recurring to depth `n-1`

Worst case partitioning



Worst case for quicksort

- In the worst case, recursion may be <u>n levels deep</u> (for an array of size n)
- But the partitioning work done at each level is still n
- $O(n) * O(n) = \underline{O(n^2)}$
- So worst case for Quicksort is $O(n^2)$
- When does this happen?
  - There are many arrangements that *could* make this happen
  - Here are two common cases:
    - When the array is <u>already sorted</u>
    - When the array is <u>*inversely* sorted</u> (sorted in the opposite order)

Typical case for quicksort

- If the array is sorted to begin with, Quicksort is terrible: $O(n^2)$
- It is possible to construct other bad cases
- However, Quicksort is *usually* $O(n \log_2 n)$
- The constants are so good that Quicksort is generally the faster algorithm.
- Most real-world sorting is done by Quicksort

Picking a better pivot

- Before, we picked the *first* element of the subarray to use as a pivot
  - If the array is already sorted, this results in $O(n^2)$ behavior
  - It's no better if we pick the *last* element
- We could do an *optimal* quicksort (guaranteed $O(n \log n)$) if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a **median**: half of the values in the array are larger, half are smaller
  - The easiest way to find the median is to *sort* the array and pick the value in the middle (!)

Median of three

- Obviously, it doesn't make sense to sort the array in order to find the median to use as a pivot.
- Instead, compare just *three* elements of our (sub)array—the first, the last, and the middle
  - Take the *median* (middle value) of these three as the pivot
  - It's possible (but not easy) to construct cases which will make this technique $O(n^2)$

Quicksort for Small Arrays

- For very small arrays (N<= 20), quicksort does not perform as well as insertion sort
- A good cutoff range is N=10
- Switching to insertion sort for small arrays can save about 15% in the running time

Mergesort vs Quicksort

- Both run in $O(n \lg n)$
  - Mergesort – always.
  - Quicksort – on average
- Compared with Quicksort, <u>Mergesort</u> has <u>less</u> number of <u>comparisons</u> but <u>larger</u> number of <u>moving</u> elements
- In <u>Java</u>, an element <u>comparison</u> is <u>expensive</u> but <u>moving</u> elements is <u>cheap</u>. Therefore, <u>Mergesort</u> is used in the standard Java library for generic sorting

Mergesort vs Quicksort

In C++, <u>copying</u> objects can be <u>expensive</u> while <u>comparing</u> objects often is relatively <u>cheap</u>. Therefore, <u>quicksort</u> is the sorting routine commonly used in <u>C++</u> libraries

Partition Implementation (Java)


```java
static int Partition(int[] a, int left, int right) {
    int p = a[left], l = left + 1, r = right;
    while (l < r) {
        while (l < right && a[l] < p)   l++;
        while (r > left && a[r] >= p)   r--;
        if (l < r) {
            int temp = a[l]; a[l] = a[r]; a[r] = temp;
        }
    }
    a[left] = a[r];
    a[r] = p;
    return r;
}
```

Quicksort Implementation (Java)


```
static void Quicksort(int[] array, int left, int right) {
    if (left < right) {
        int p = Partition(array, left, right);
        Quicksort(array, left, p - 1);
        Quicksort(array, p + 1, right);
    }
}
```

`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)`

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

`pivot=6`

`left`

Partition Initialization...

quickSort(arr,0,5)

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

partition(arr,0,5)

pivot=6

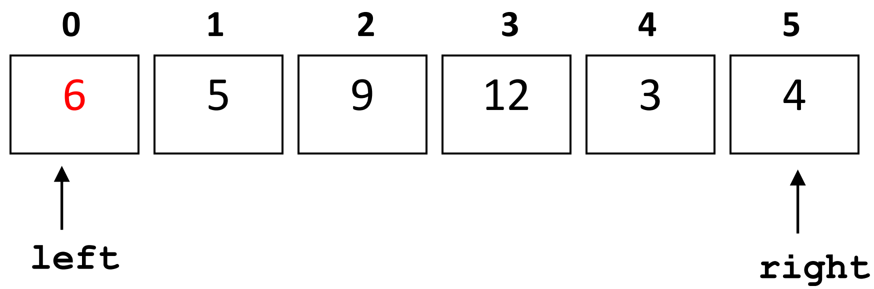
left (index 0) **right** (index 5)

Partition Initialization...

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`

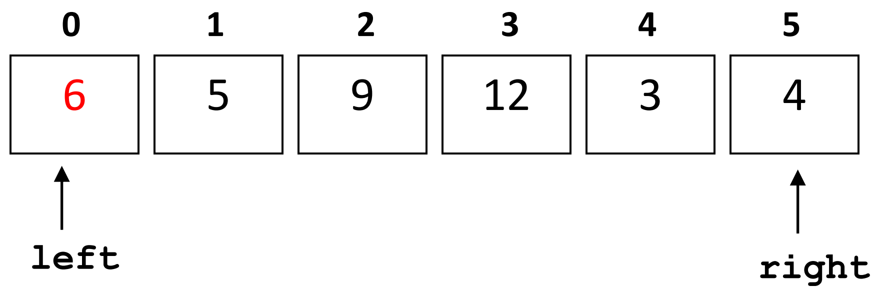


right moves to the left until value that should be to left of pivot...

`quickSort(arr,0,5)`

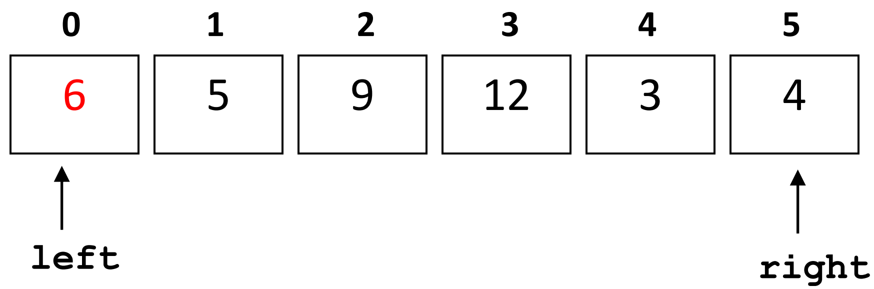
`partition(arr,0,5)`

`pivot=6`



`quickSort(arr,0,5)`

`partition(arr,0,5)`



`pivot=6`

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

quickSort(arr,0,5)

partition(arr,0,5)

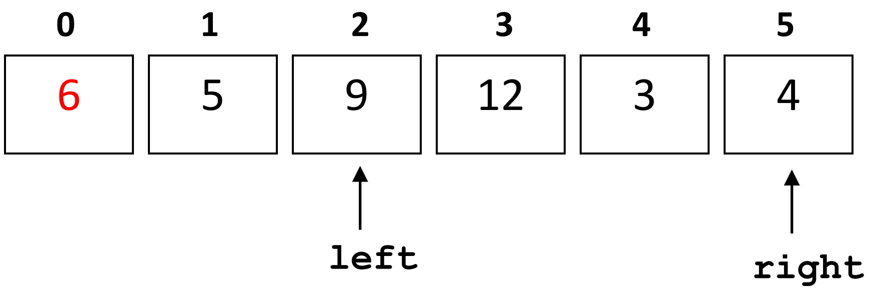
pivot=6

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

left **right**

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

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



pivot=6

quickSort(arr,0,5)

partition(arr,0,5)

0	1	2	3	4	5
6	5	9	12	3	4
		↑ left			↑ right

pivot=6

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)

0	1	2	3	4	5
6	5	4	12	3	9
		↑			↑
		left			**right**

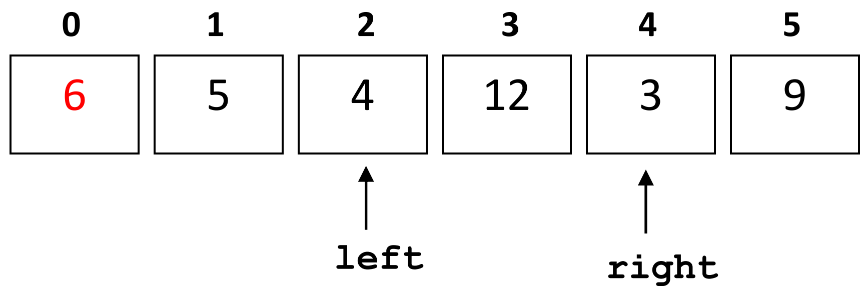
pivot=6

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

quickSort(arr,0,5)

partition(arr,0,5)

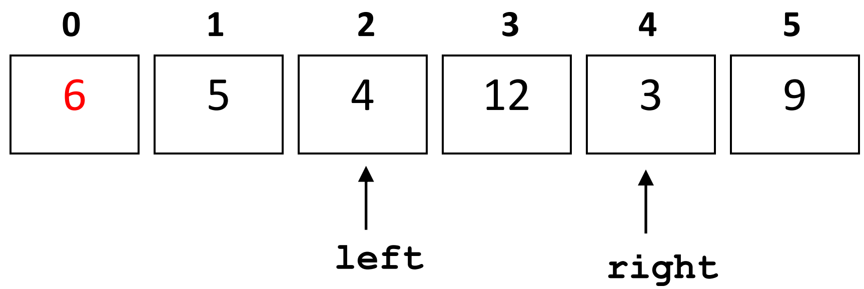
pivot=6



quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



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

`quickSort(arr,0,5)`

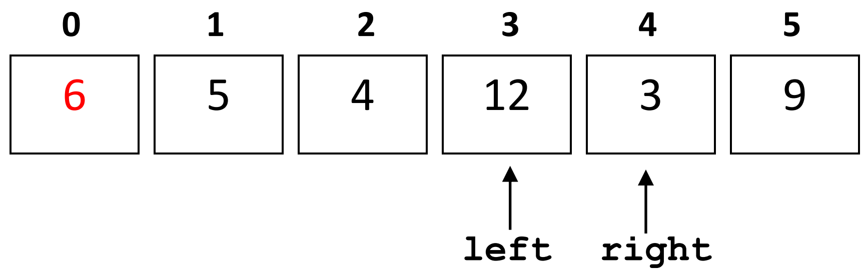
`partition(arr,0,5)`

`pivot=6`

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

quickSort(arr,0,5)

partition(arr,0,5)



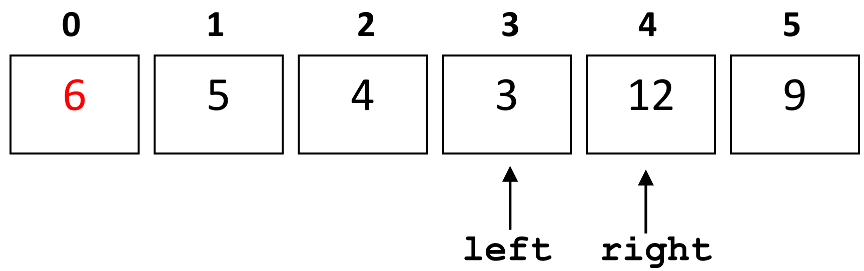
pivot=6

swap arr[left] and arr[right]

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`

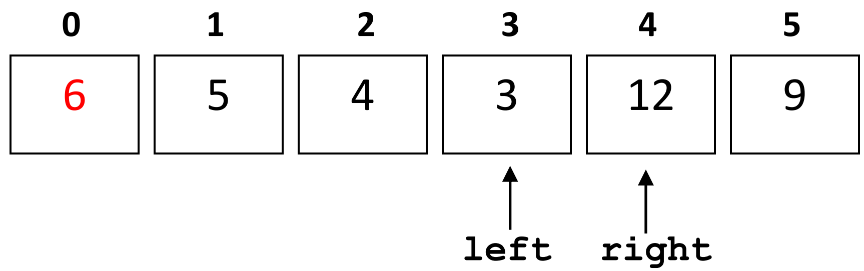


swap arr[left] and arr[right]

quickSort(arr,0,5)

partition(arr,0,5)

pivot=6



repeat right/left scan
UNTIL left & right cross

`quickSort(arr,0,5)`

`partition(arr,0,5)`

0	1	2	3	4	5
6	5	4	3	12	9
			`left`	`right`	

`pivot=6`

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

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

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

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

```
pivot=6
```

```
left
right
```

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`

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

`left`

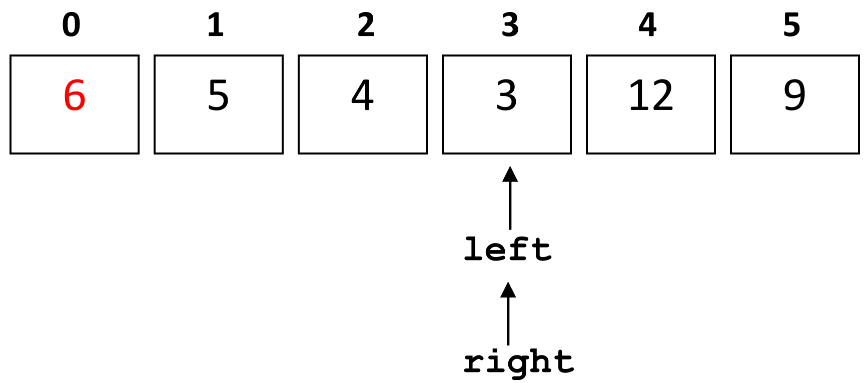
`right`

right & left CROSS!!!

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`



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

`left`

`right`

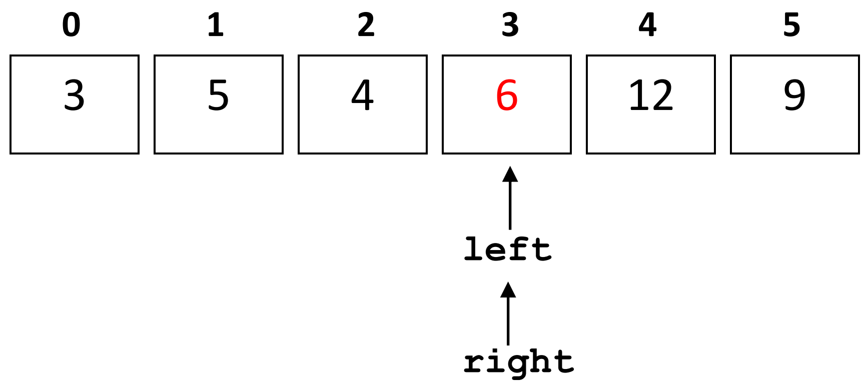
right & left CROSS!!!

1 - Swap pivot and arr[right]

`quickSort(arr,0,5)`

`partition(arr,0,5)`

`pivot=6`



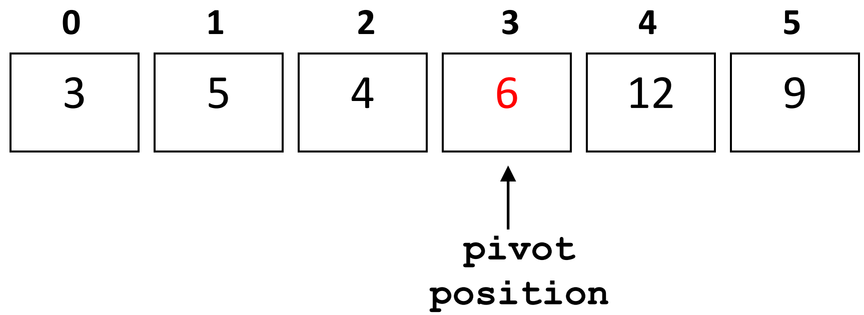
right & left CROSS!!!

1 - Swap pivot and arr[right]

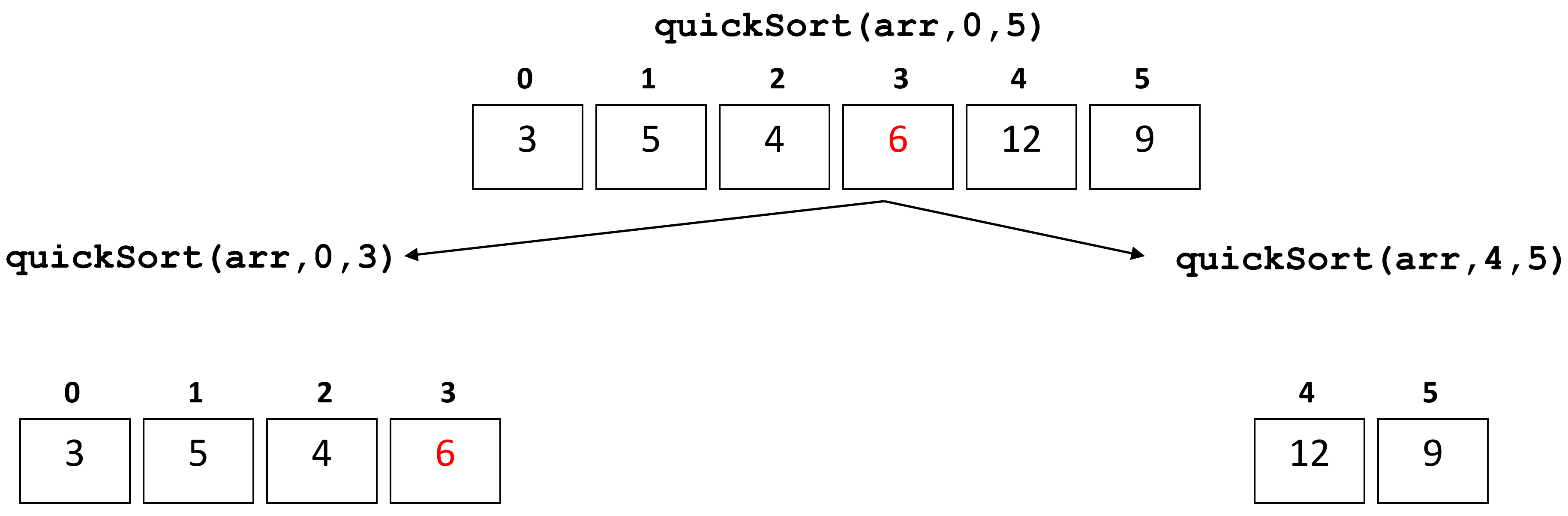
2 - Return new location of pivot to caller

`return 3`

`quickSort(arr,0,5)`



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)
quickSort(arr,4,5)
0 1 2 3
3 5 4 6
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)`

0	1	2	3
3	5	4	6

`quickSort(arr,4,5)`

4	5
12	9

Partition Initialization...

`quickSort(arr,0,5)`

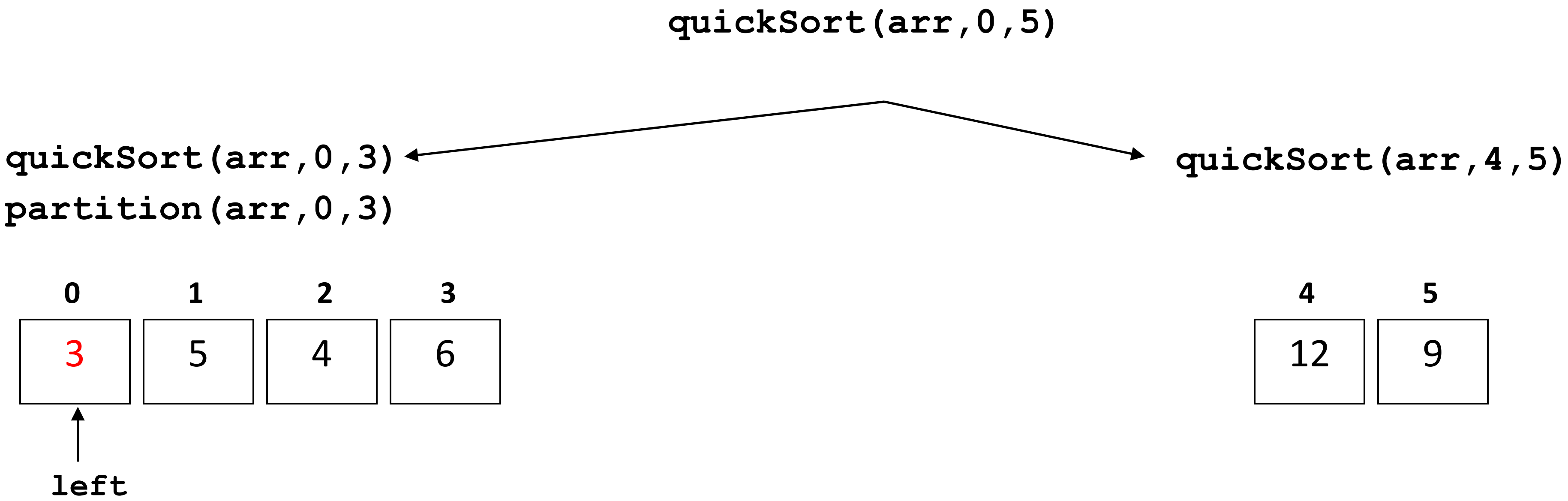
`quickSort(arr,0,3)`
`partition(arr,0,3)`

0	1	2	3
3	5	4	6

`quickSort(arr,4,5)`

4	5
12	9

Partition Initialization...



Partition Initialization...

`quickSort(arr,0,5)`

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

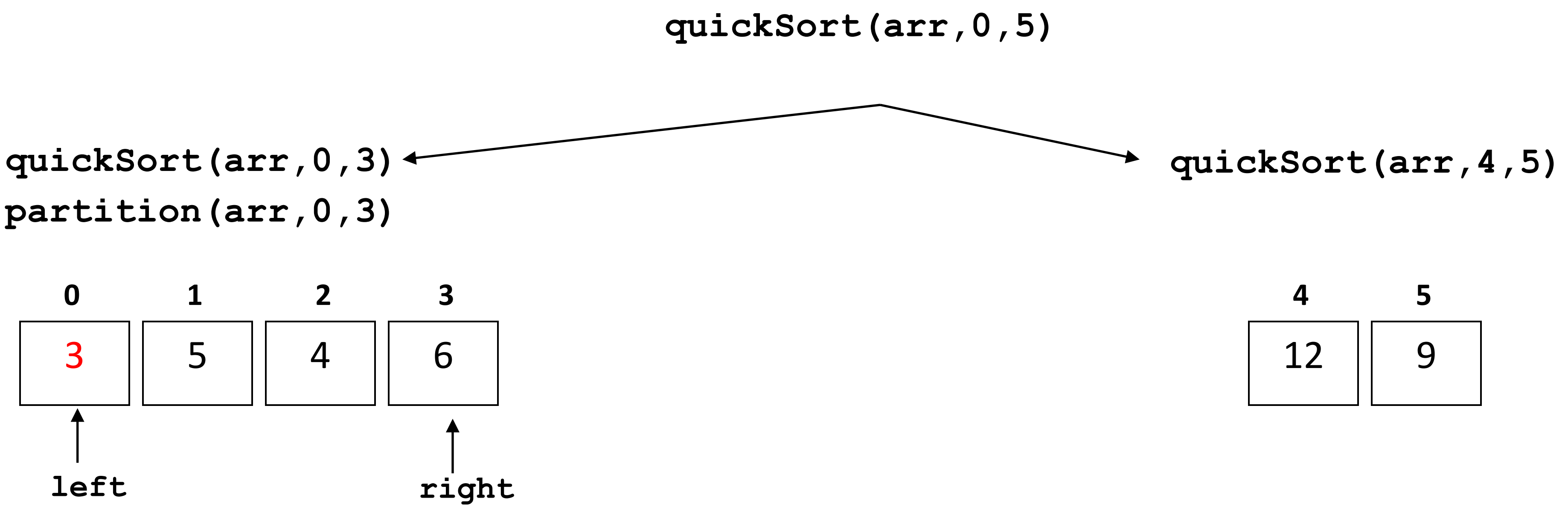
0	1	2	3
3	5	4	6

`left` `right`

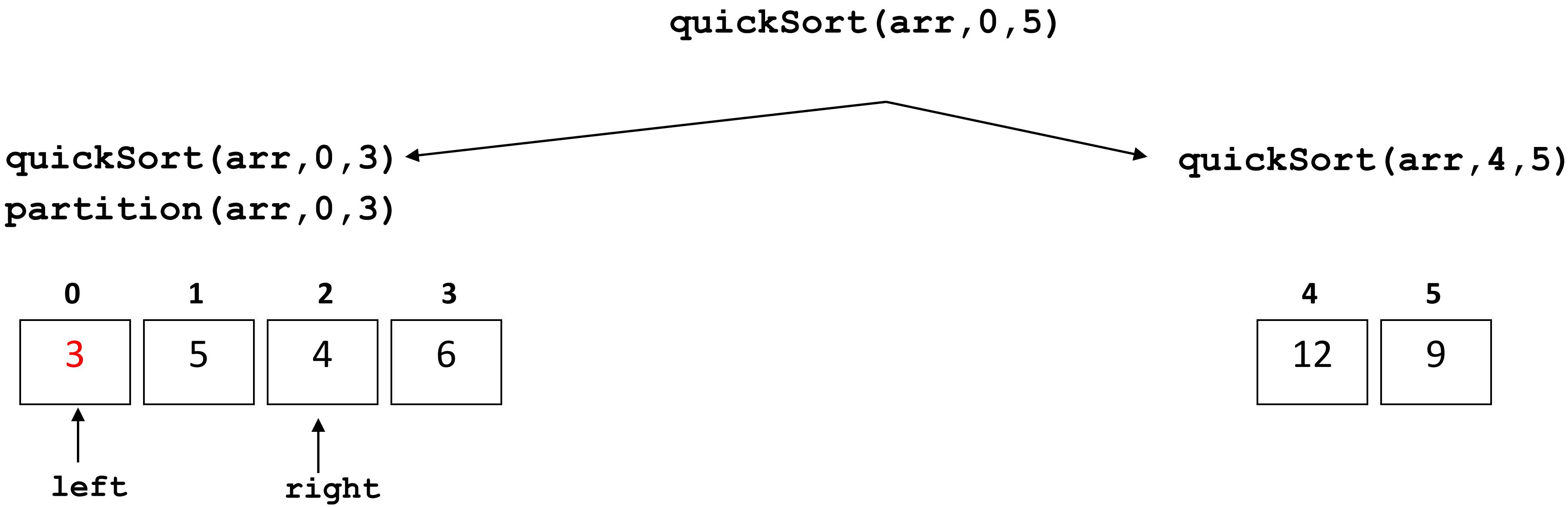
`quickSort(arr,4,5)`

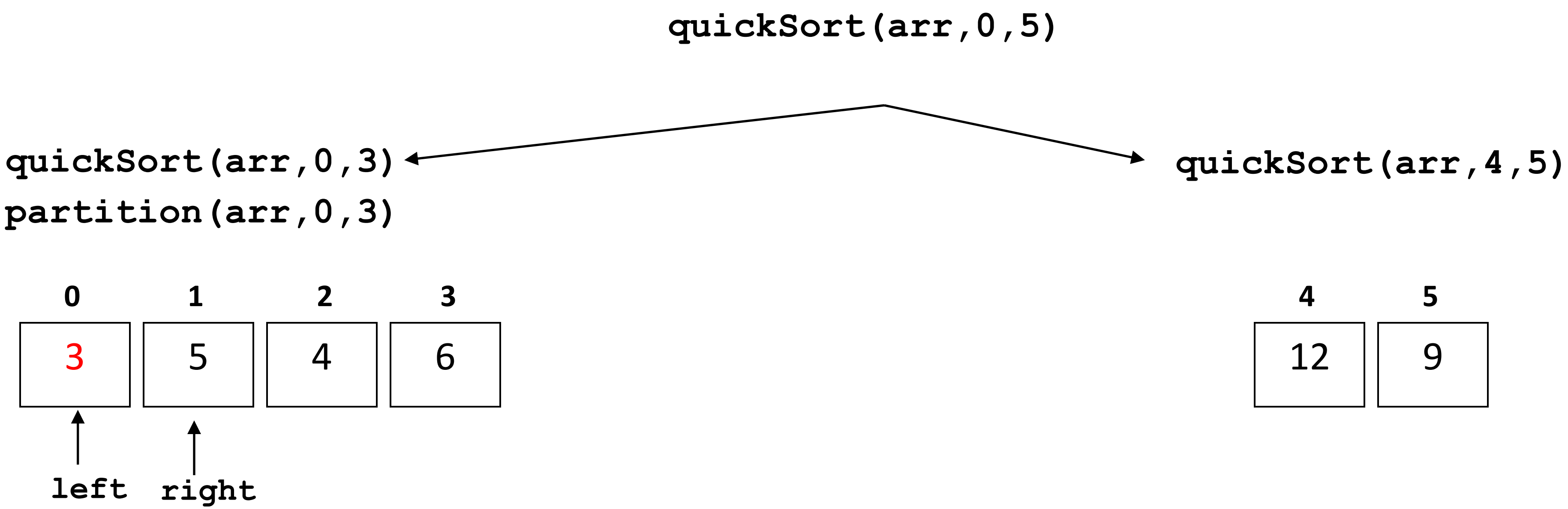
4	5
12	9

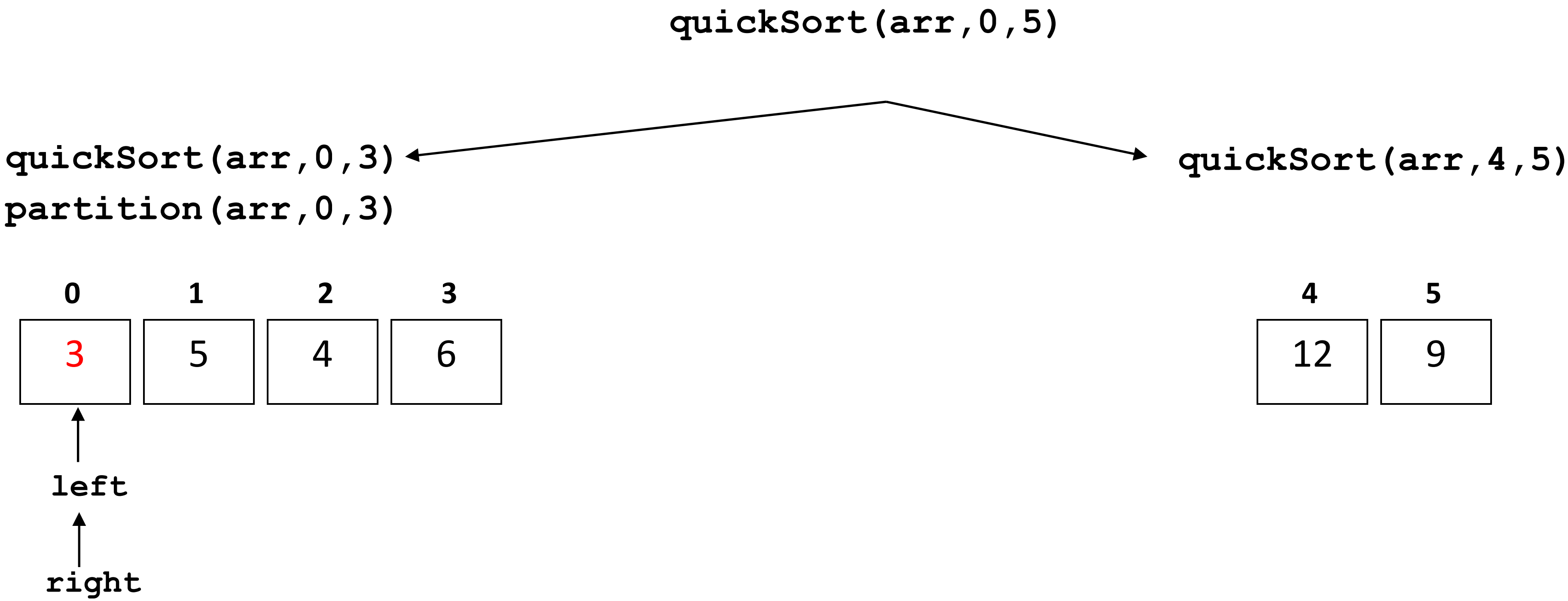
Partition Initialization...



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)
quickSort(arr,4,5)
0
1
2
3
3
5
4
6
left
right
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
left
right
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
left
right
4
5
12
9

`quickSort(arr,0,5)`

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

0	1	2	3
3	5	4	6

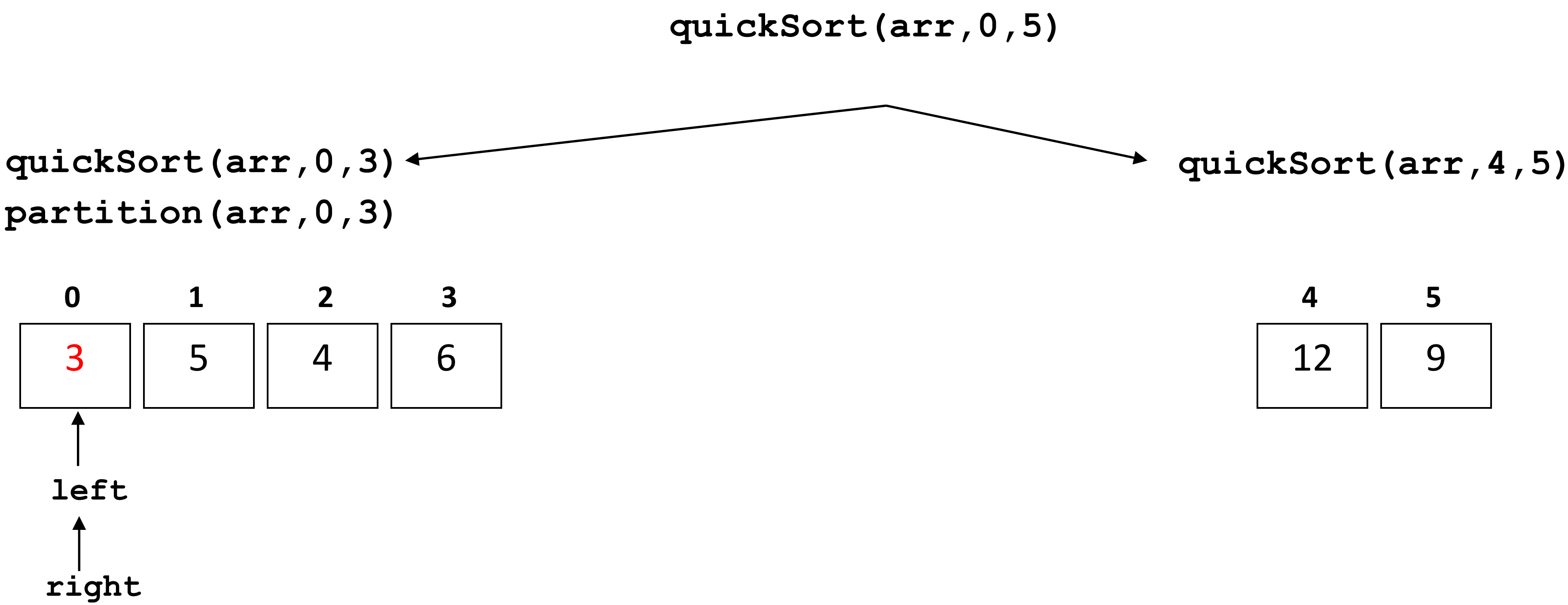
`left`

`right`

`quickSort(arr,4,5)`

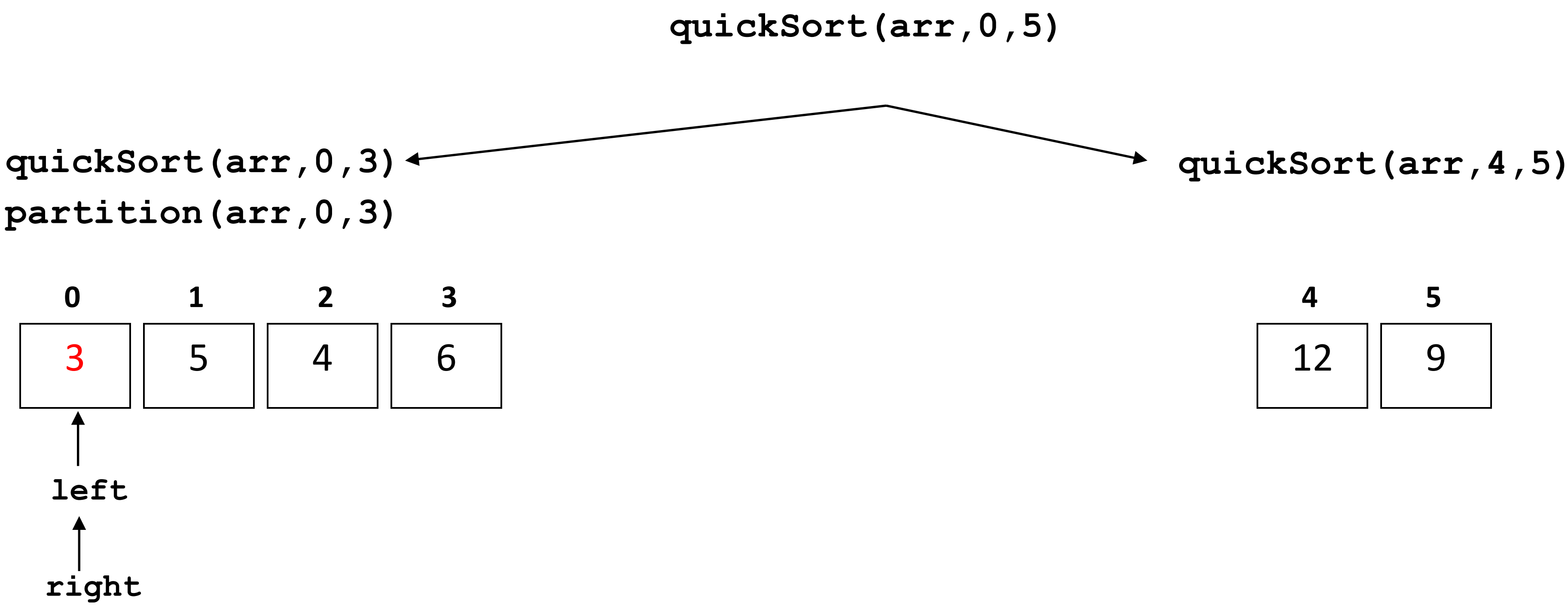
4	5
12	9

right & left CROSS!!!



right & left CROSS!!!

1 - Swap pivot and arr[right]



right & left CROSS!!!

1 - Swap pivot and arr[right]

2 - Return new location of pivot to caller

return 0

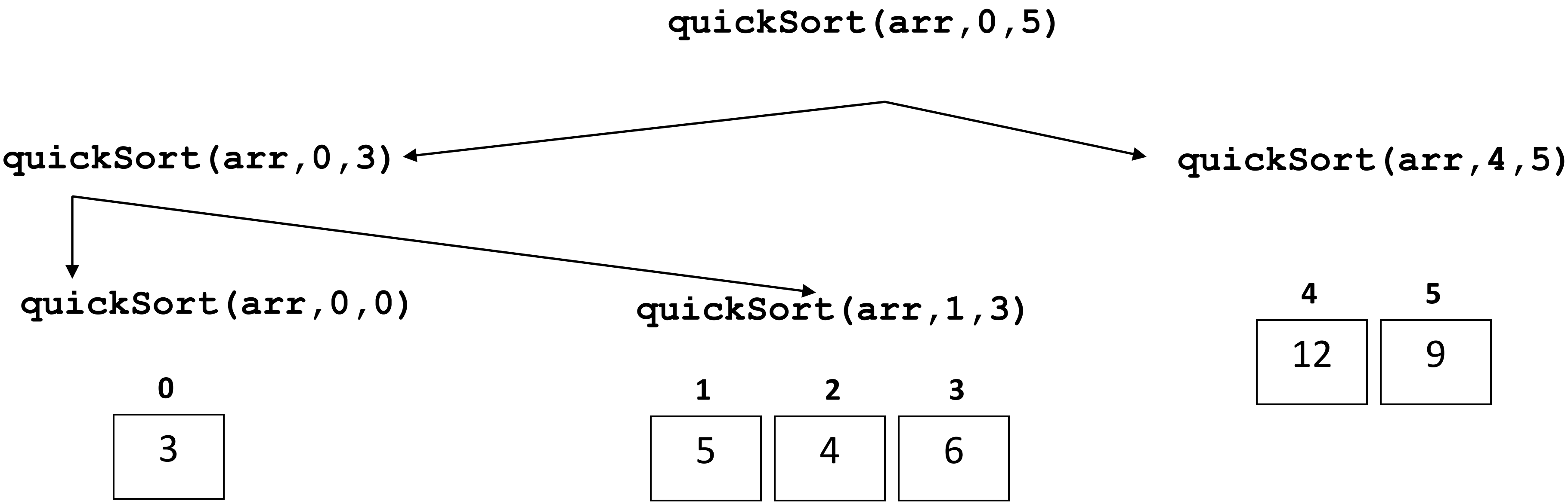
`quickSort(arr,0,5)`

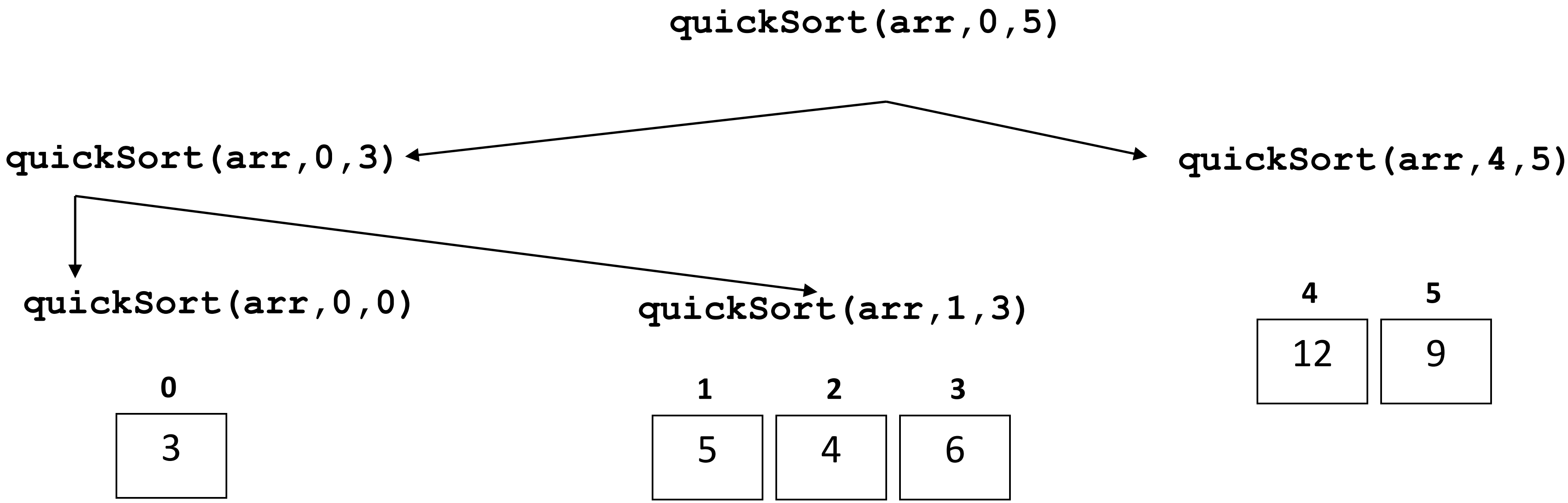
`quickSort(arr,0,3)` ← → `quickSort(arr,4,5)`

0	1	2	3
3	5	4	6

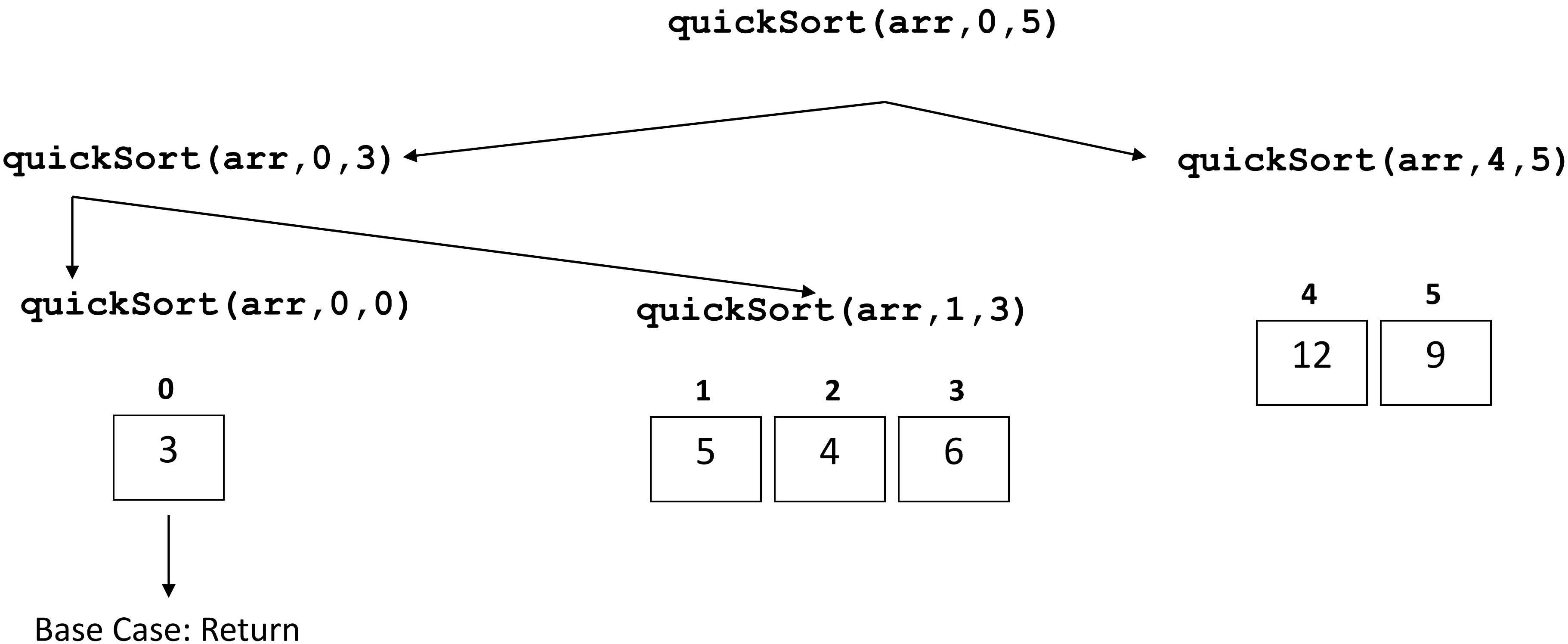
4	5
12	9

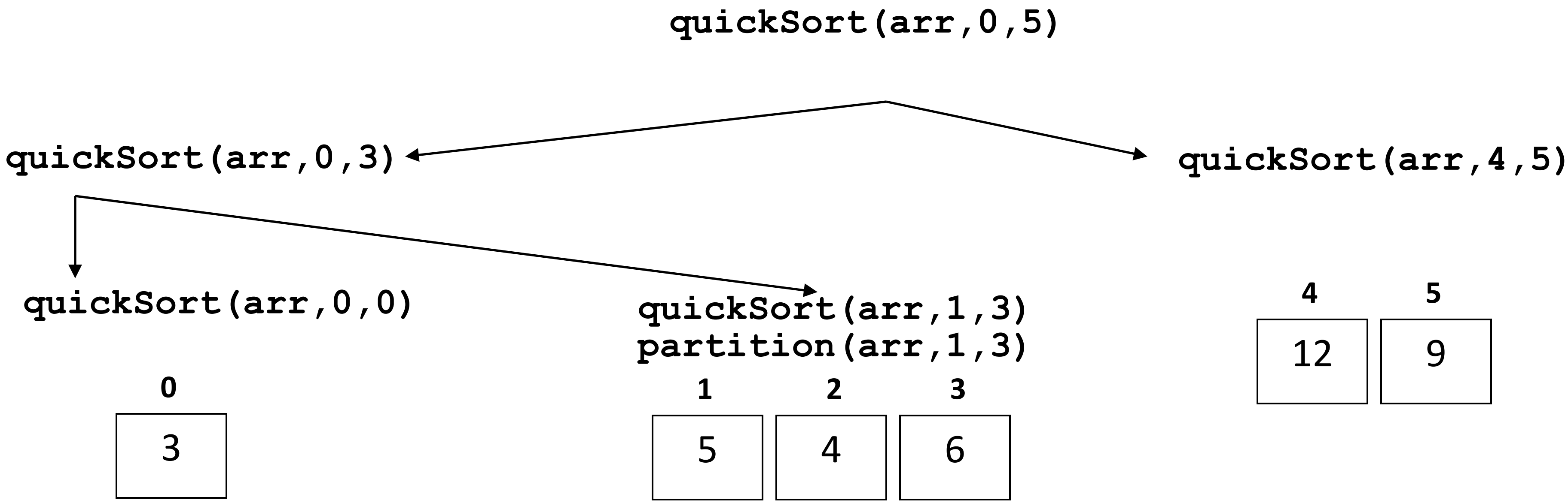
Recursive calls to quickSort()
using partitioned array...


quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
4
5
12
9
0
3
1
2
3
5
4
6

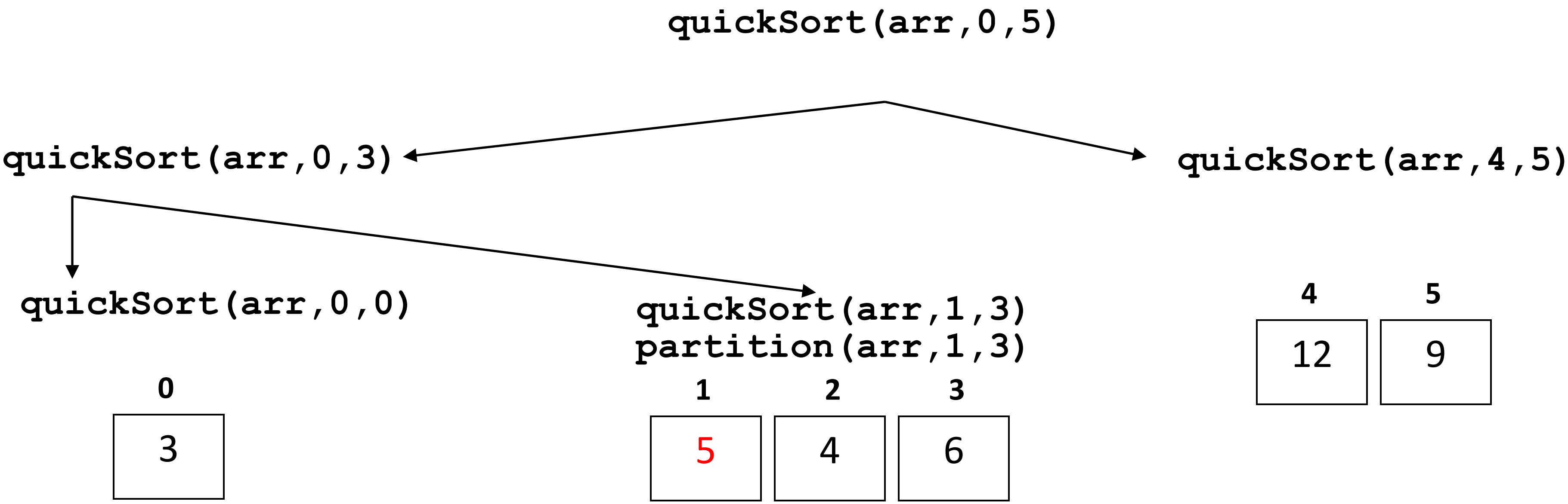


Base case triggered...
halting recursion.

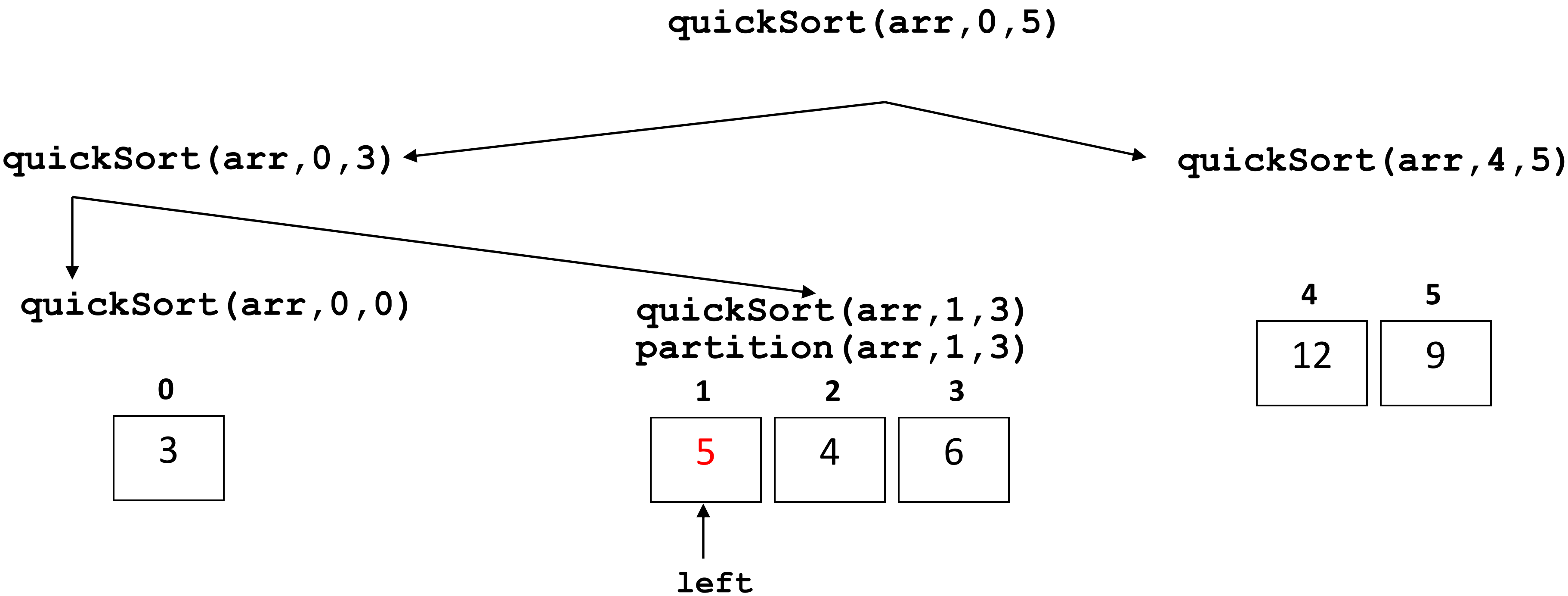

quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
4
5
12
9
0
3
1
2
3
5
4
6
Base Case: Return



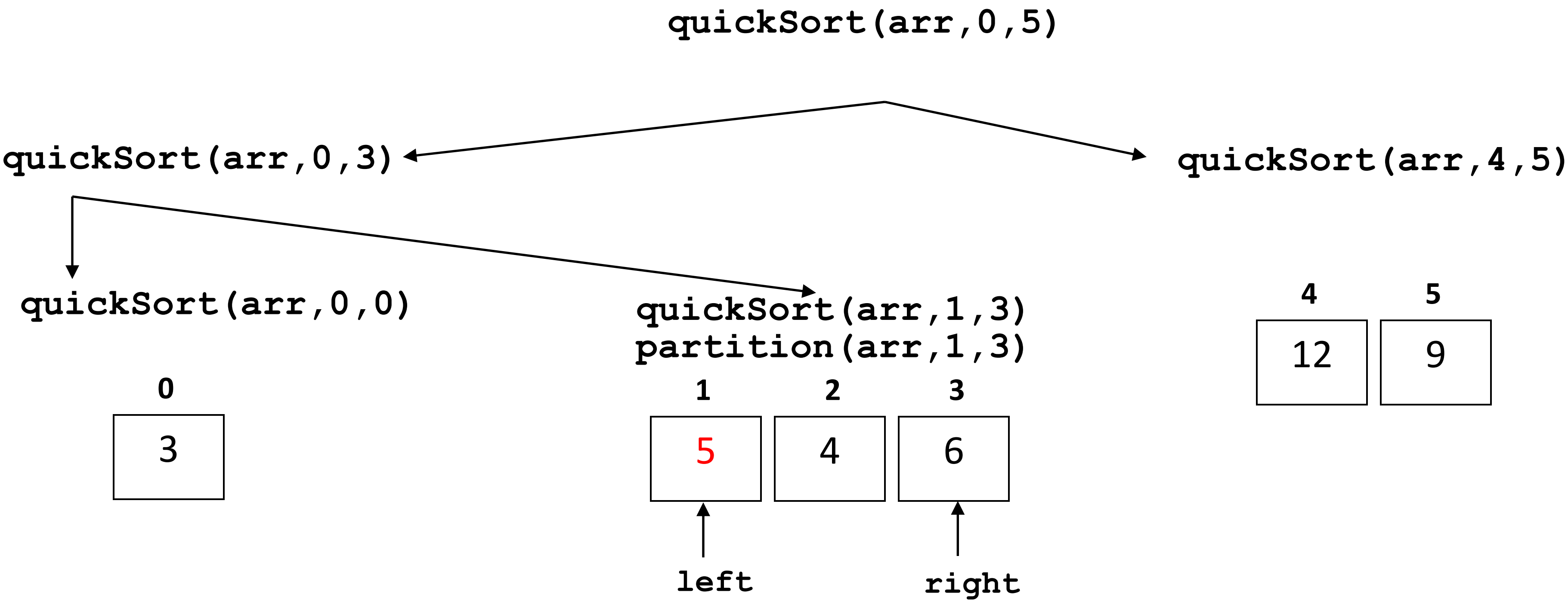
Partition Initialization...



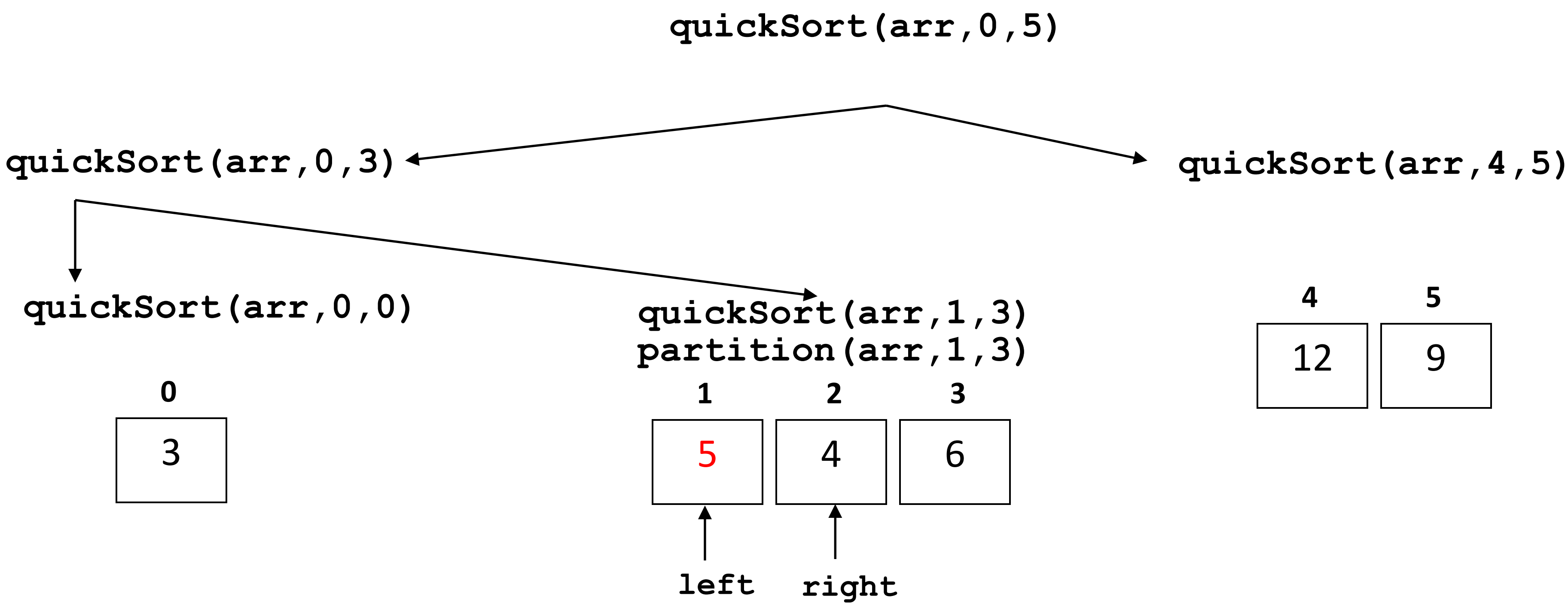
Partition Initialization...

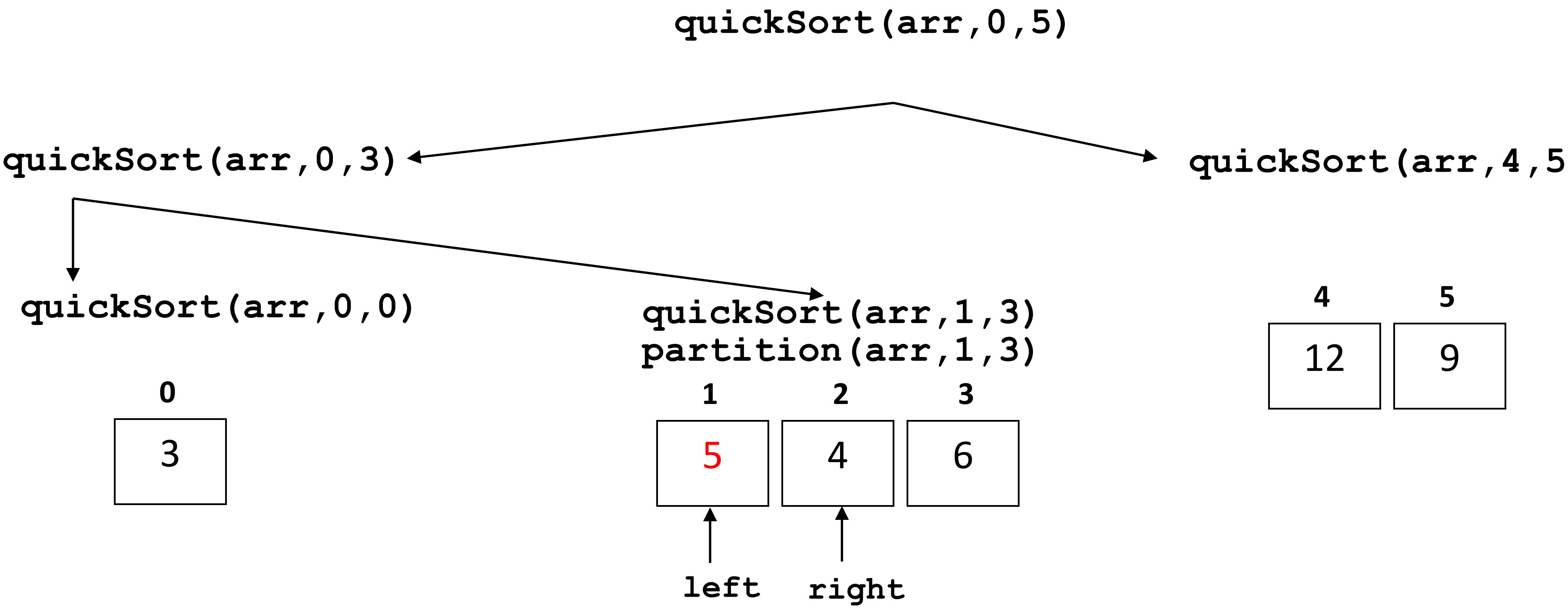


Partition Initialization...

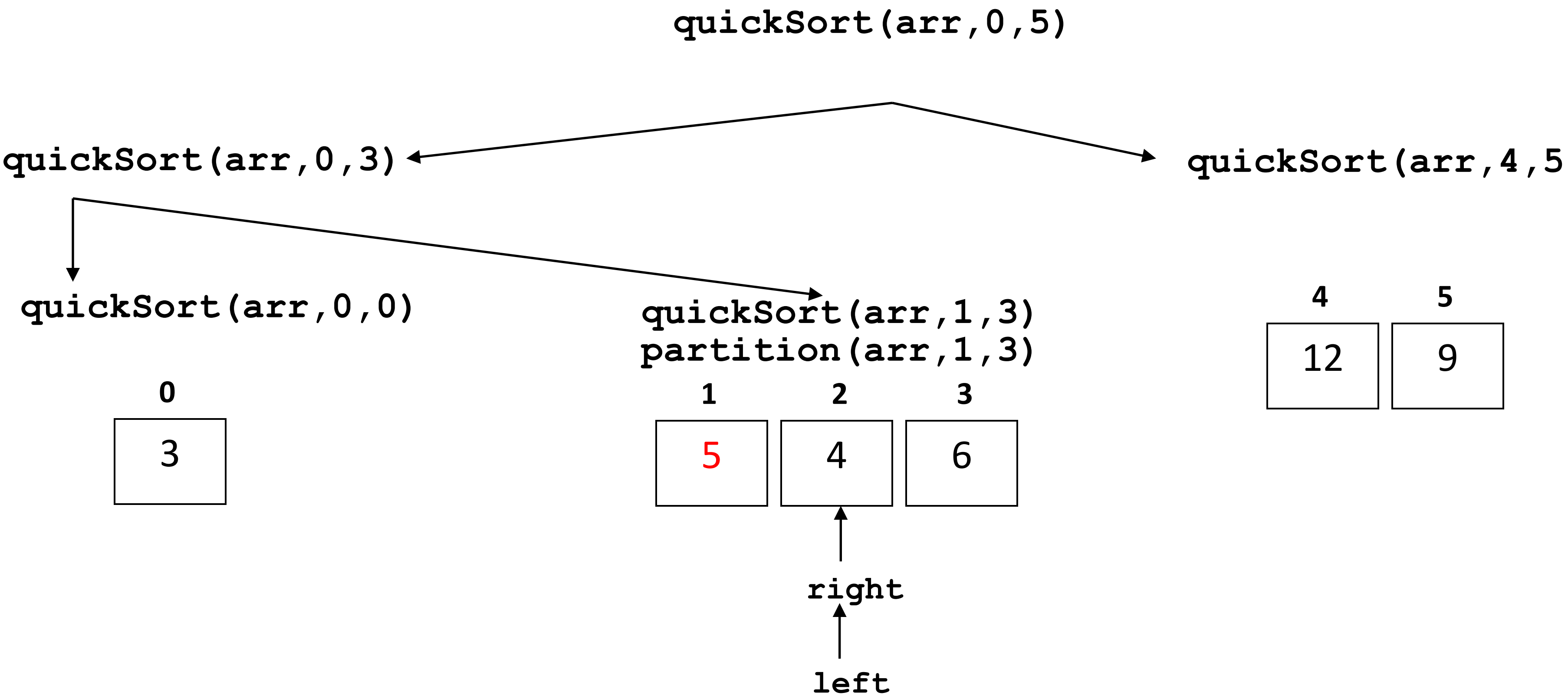


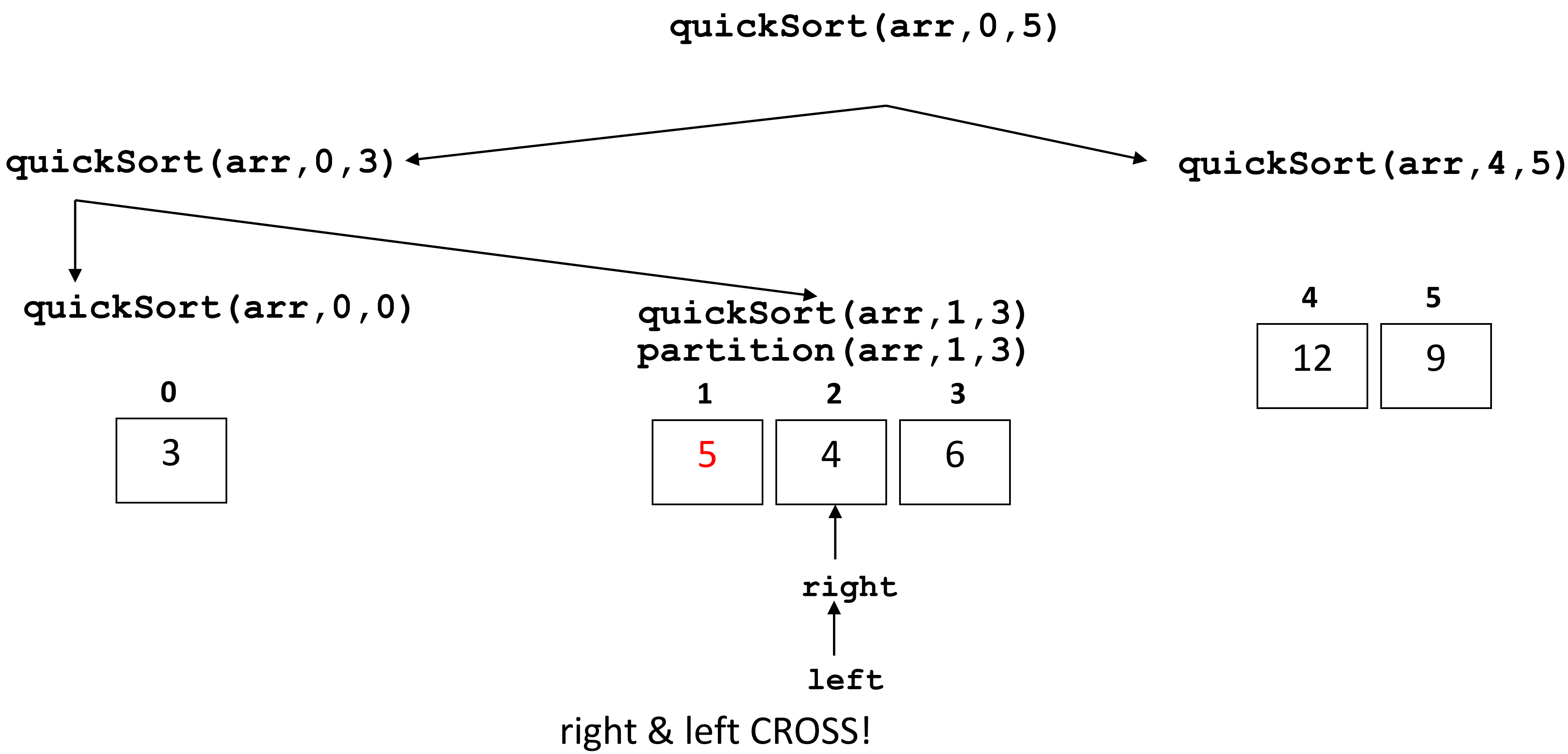
right moves to the left until value that should be to left of pivot...


quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
partition(arr,1,3)
0
3
1
2
3
5
4
6
left
right
4
5
12
9

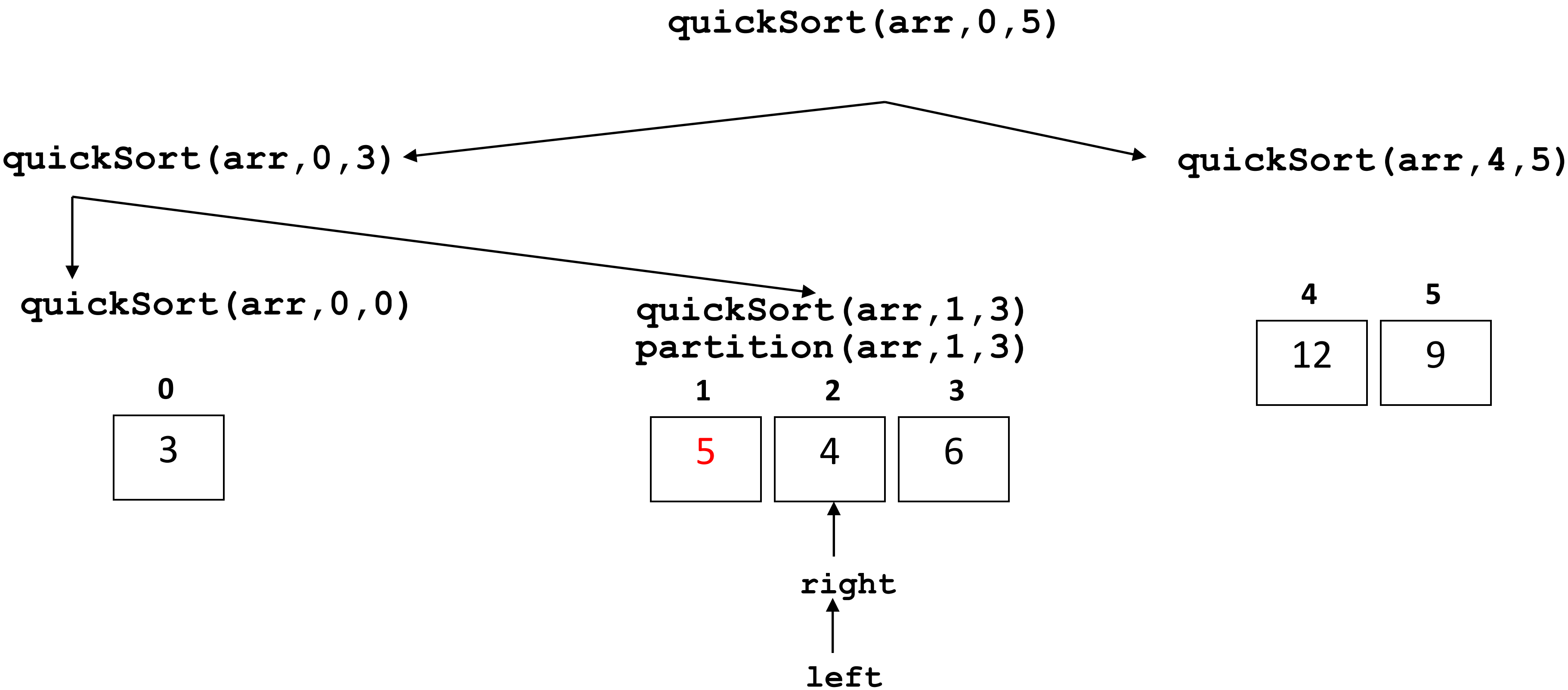


left moves to the right until value that should be to right of pivot...


quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
partition(arr,1,3)
0
3
1
2
3
5
4
6
4
5
12
9
right
left

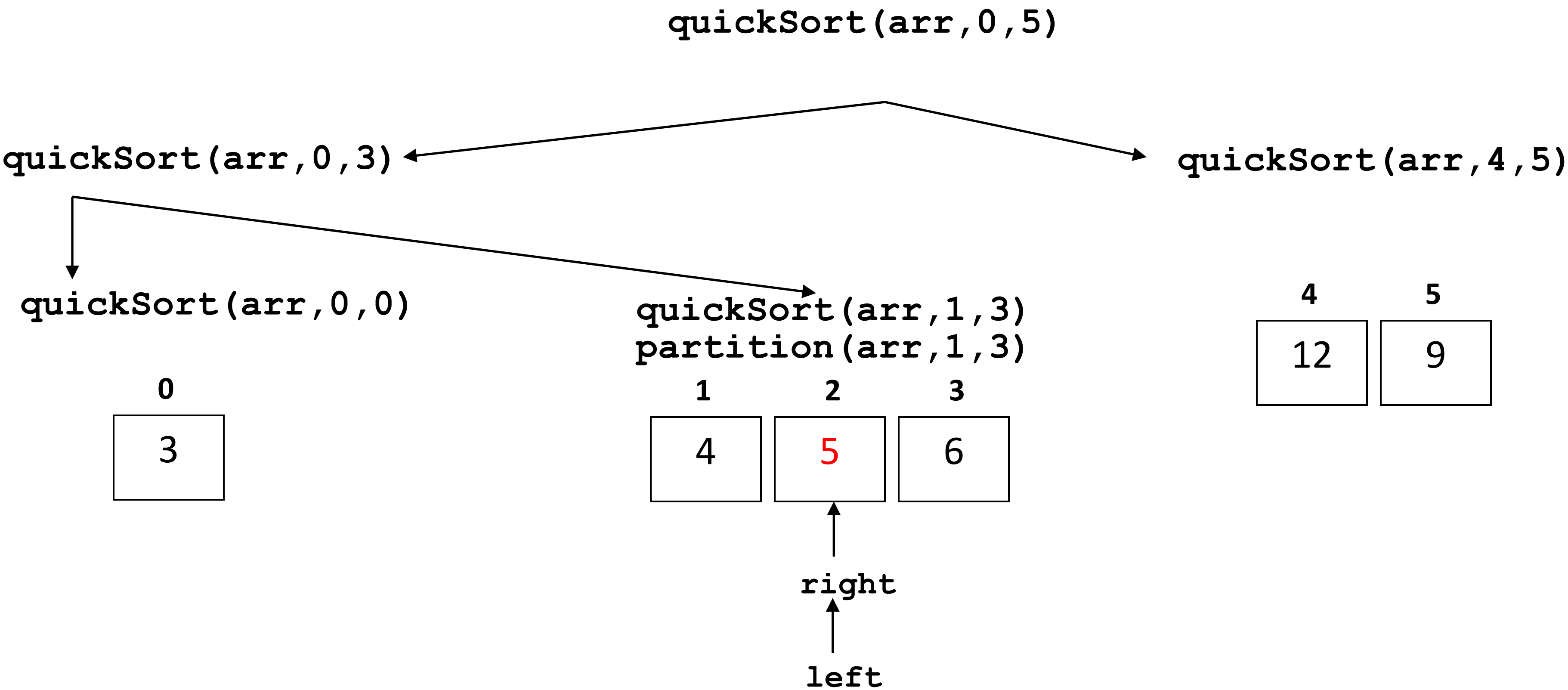


quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
partition(arr,1,3)
0
3
1
2
3
5
4
6
4
5
12
9
right
left
right & left CROSS!



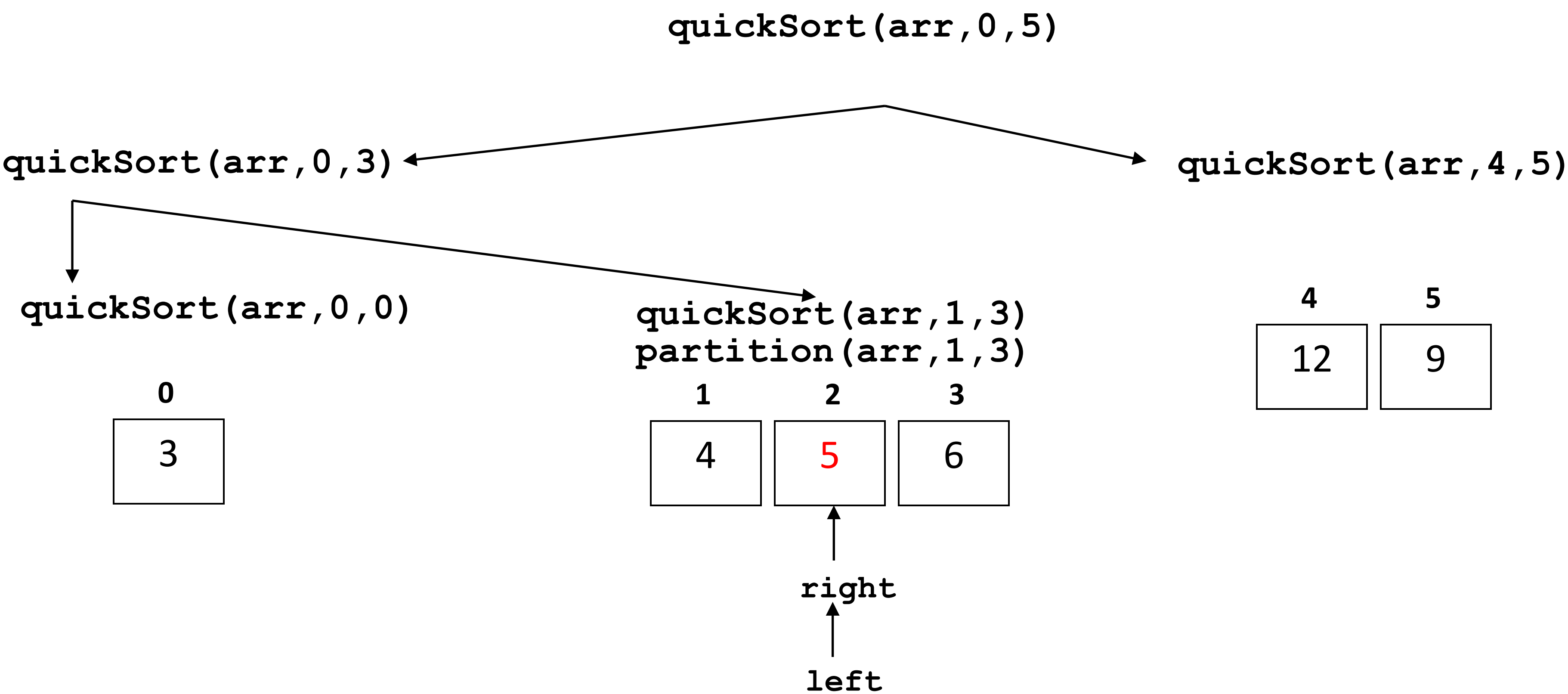
right & left CROSS!

1- swap pivot and arr[right]



right & left CROSS!

1- swap pivot and arr[right]

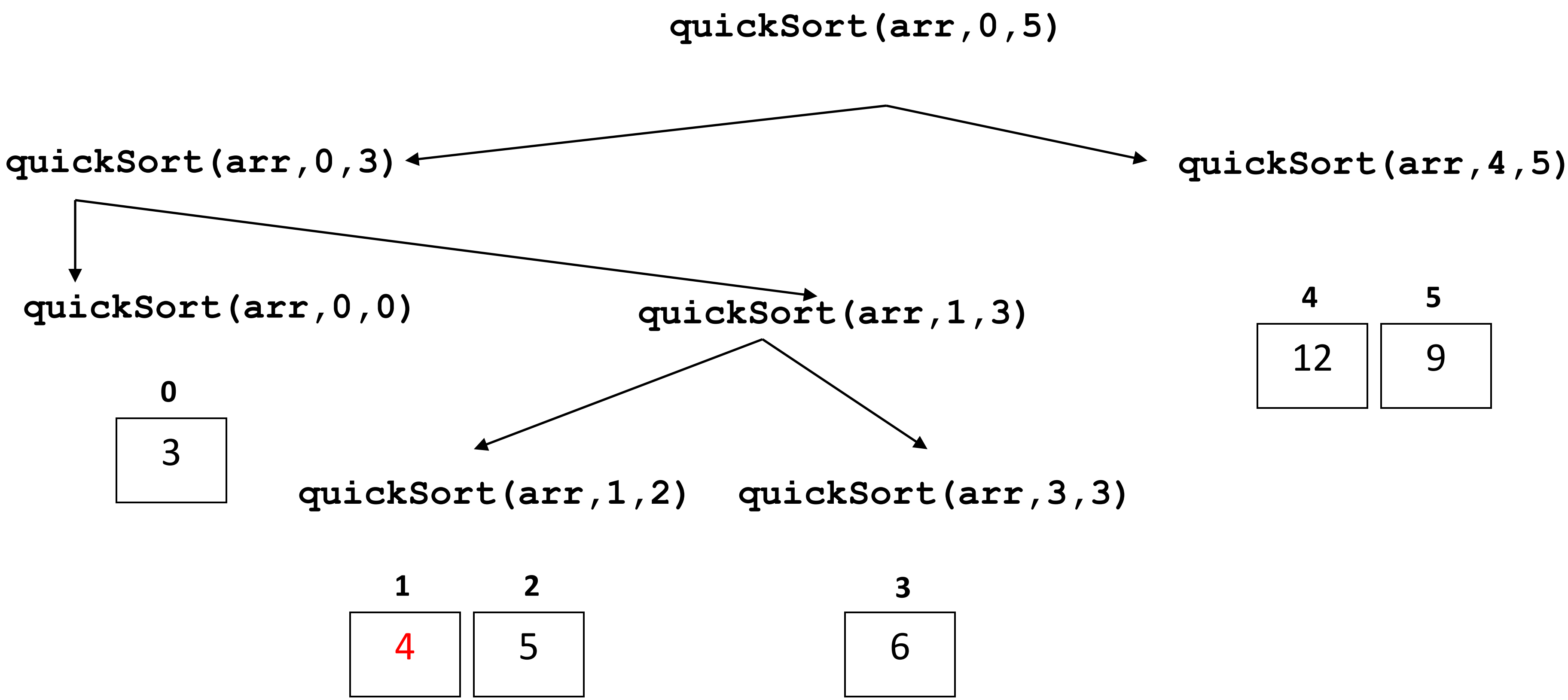


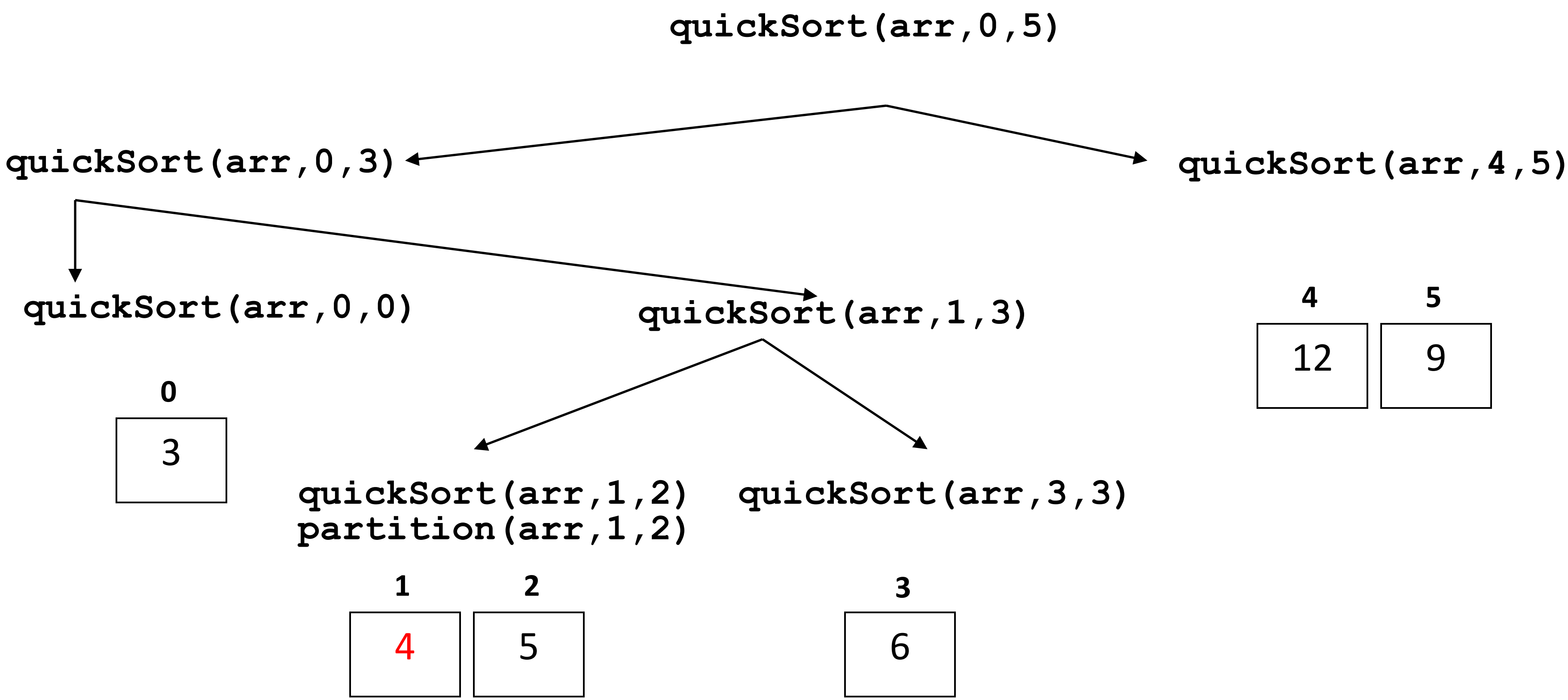
right & left CROSS!

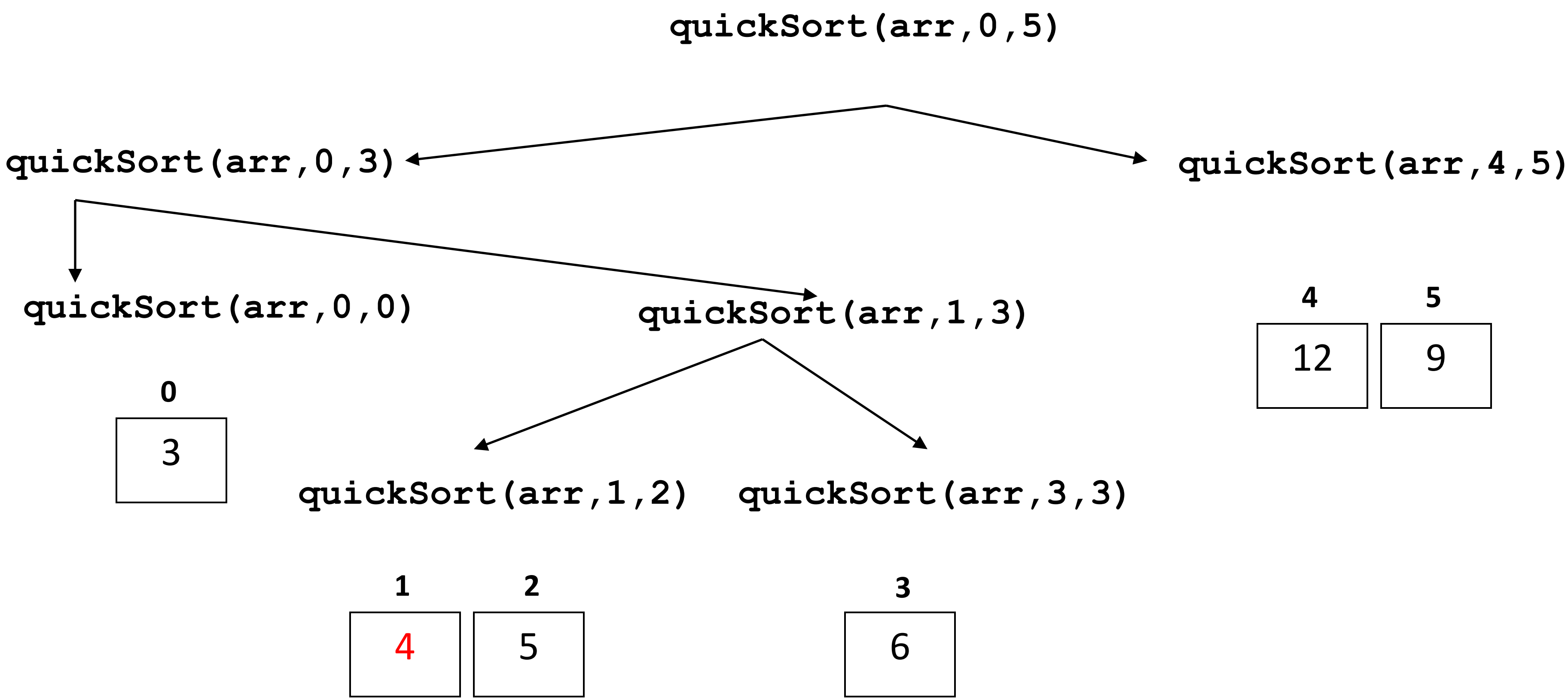
1- swap pivot and arr[right]

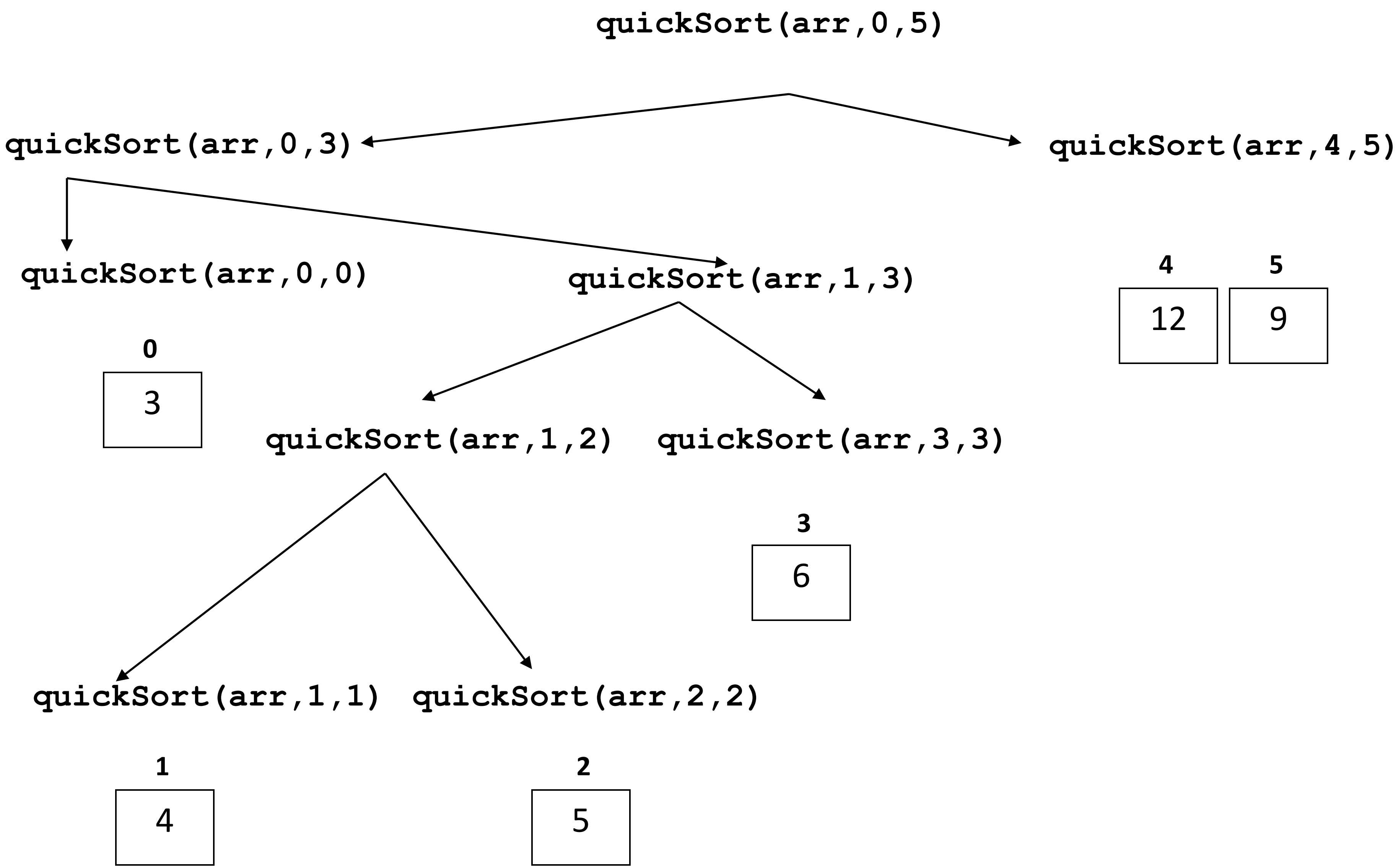
2 – return new position of pivot

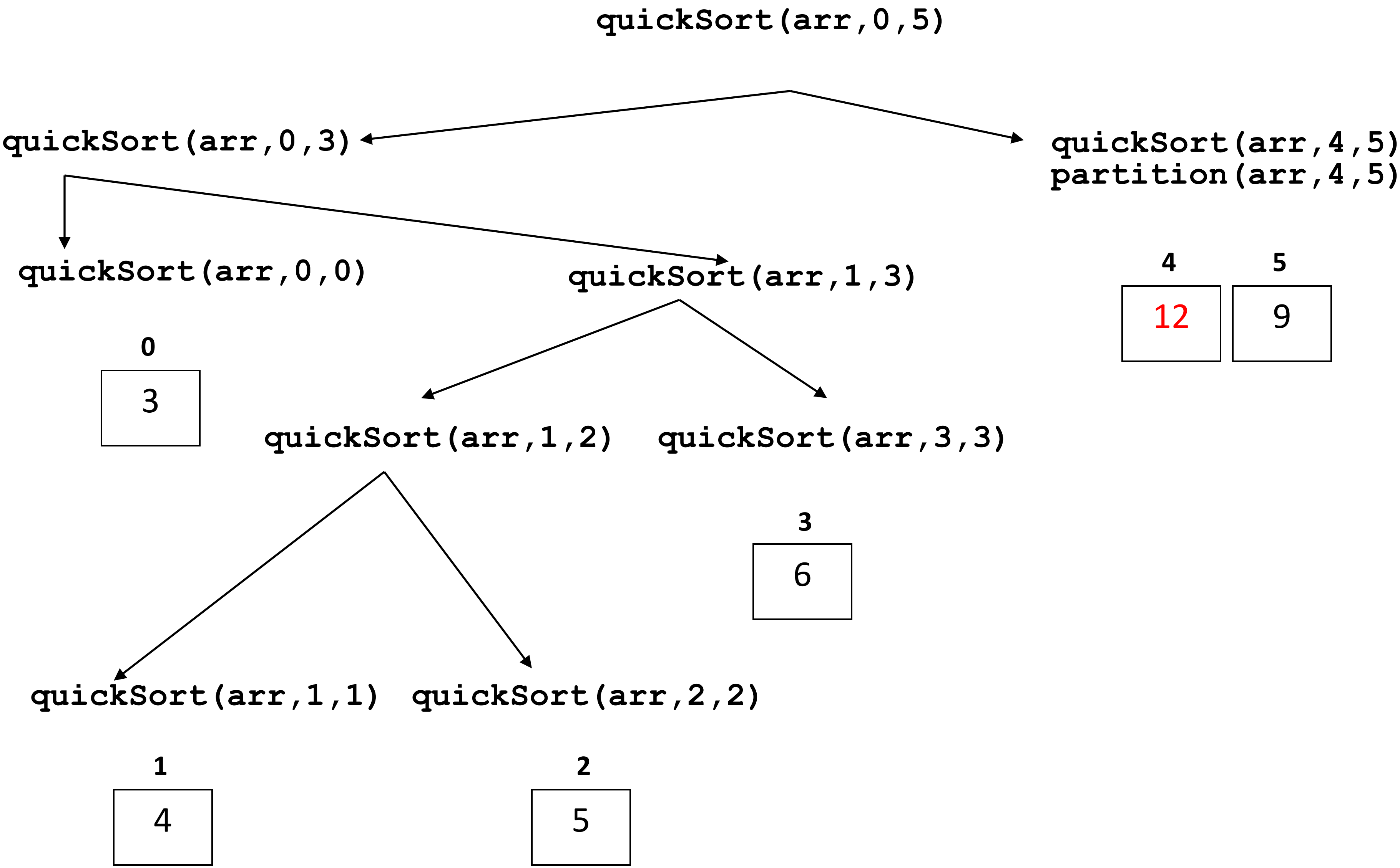
`return 2`

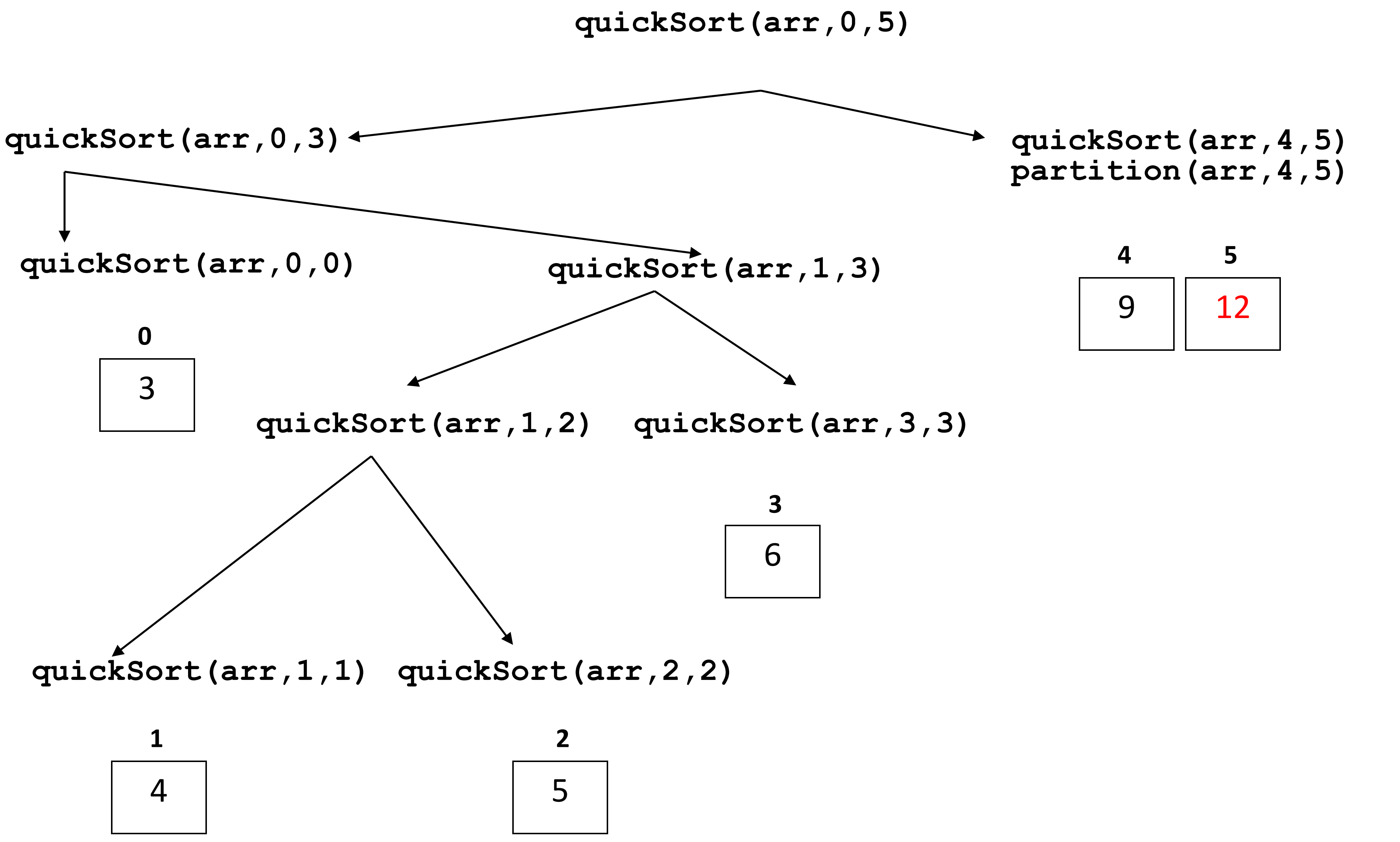

quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
4
5
12
9
0
3
quickSort(arr,1,2)
quickSort(arr,3,3)
1
2
4
5
3
6


quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
4
5
12
9
0
3
quickSort(arr,1,2)
partition(arr,1,2)
quickSort(arr,3,3)
1
2
4
5
3
6

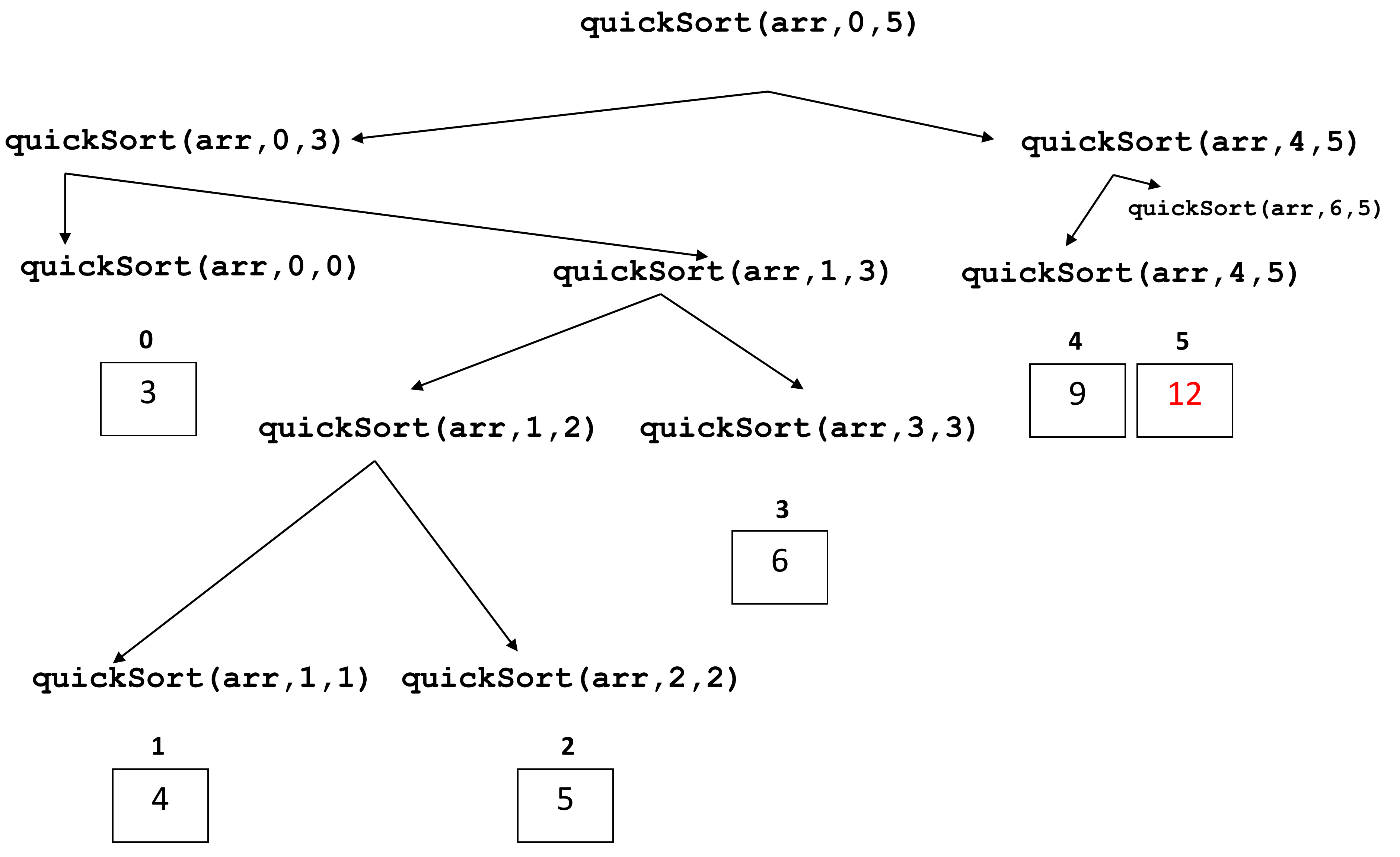

quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
4
5
12
9
0
3
quickSort(arr,1,2)
quickSort(arr,3,3)
1
2
4
5
3
6


quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
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

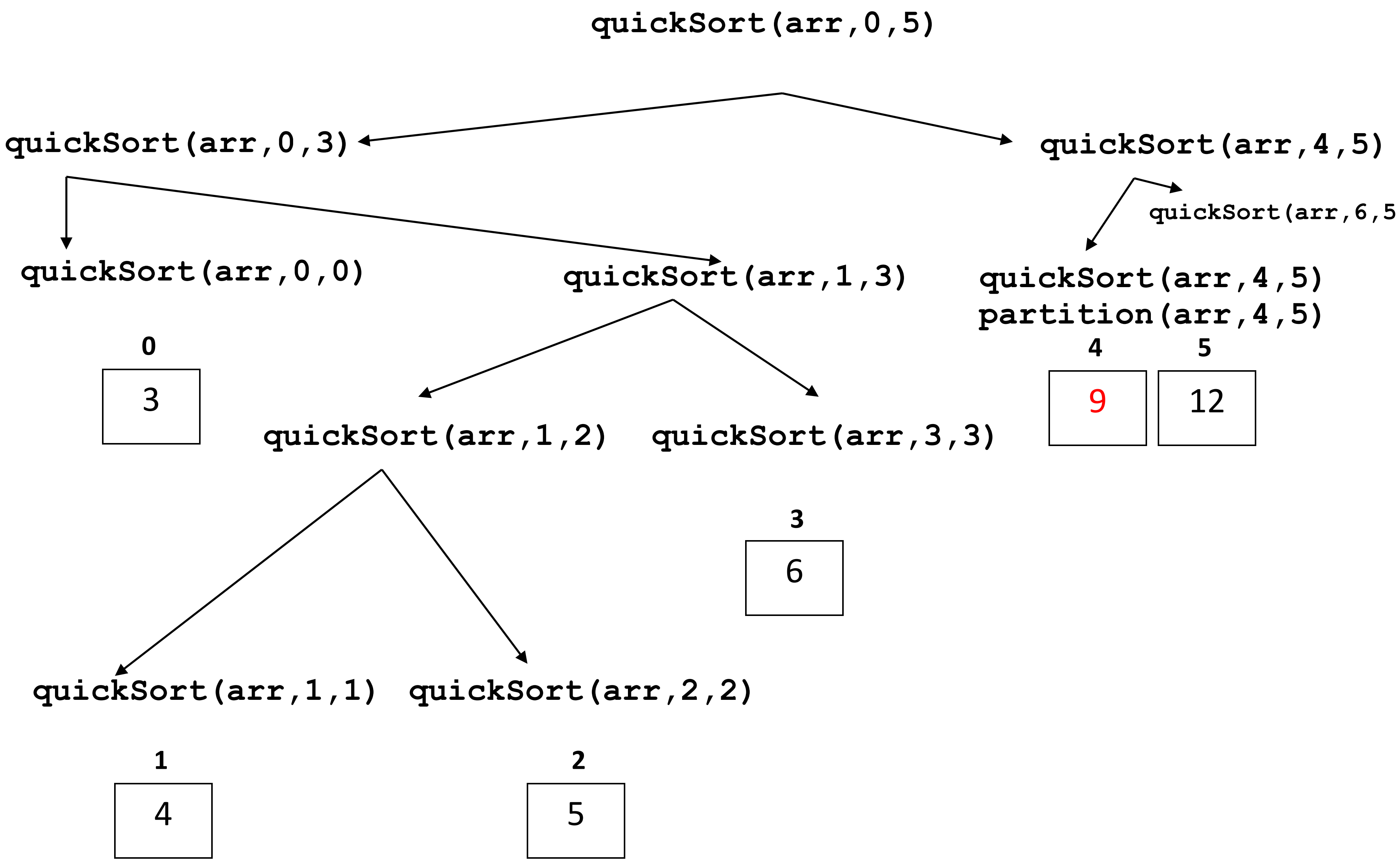

quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
partition(arr,4,5)
4
12
5
9
quickSort(arr,0,0)
0
3
quickSort(arr,1,3)
quickSort(arr,1,2)
quickSort(arr,3,3)
3
6
quickSort(arr,1,1)
1
4
quickSort(arr,2,2)
2
5



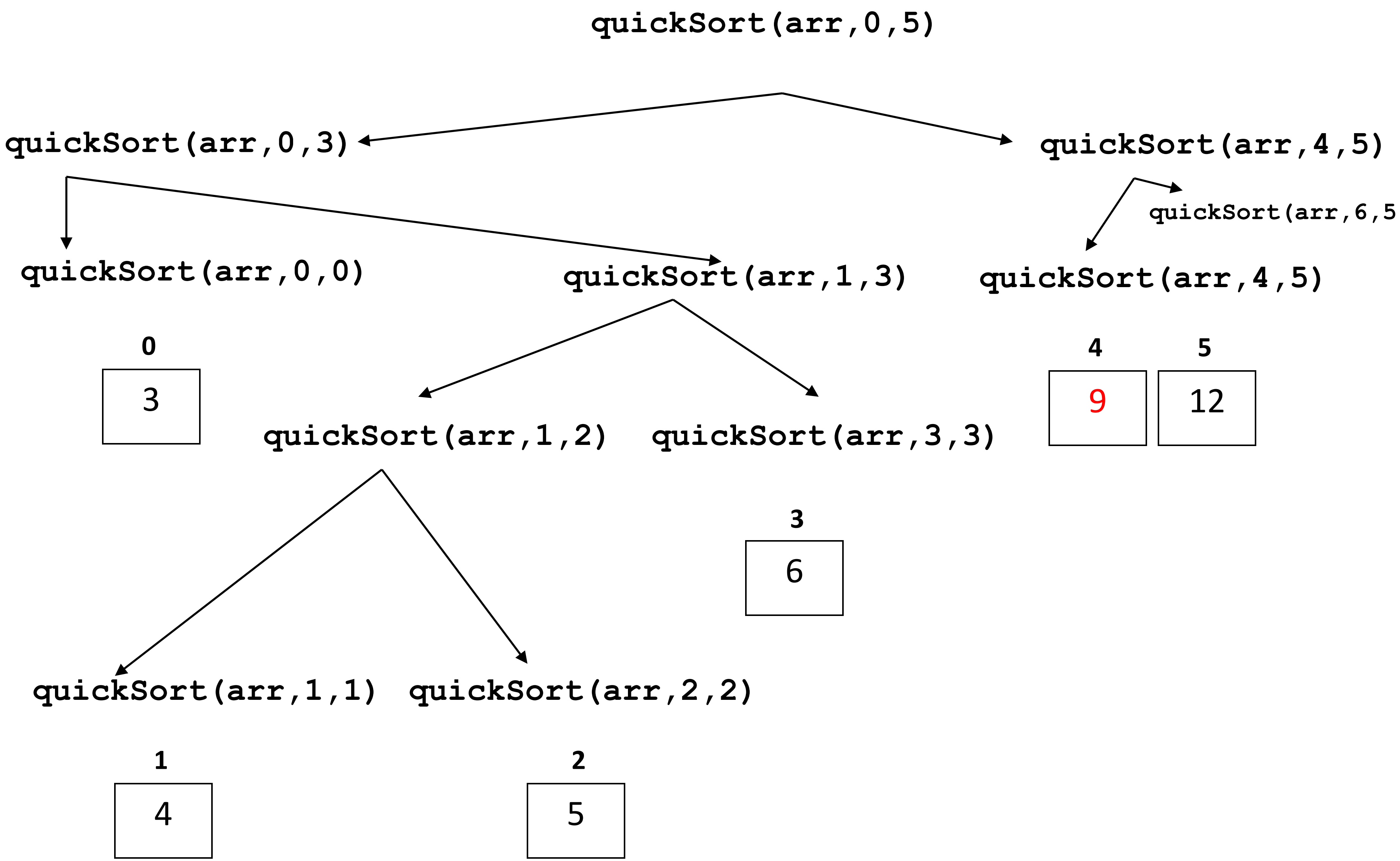
quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
partition(arr,4,5)
4
9
5
12
quickSort(arr,0,0)
0
3
quickSort(arr,1,3)
quickSort(arr,1,2)
quickSort(arr,3,3)
3
6
quickSort(arr,1,1)
1
4
quickSort(arr,2,2)
2
5



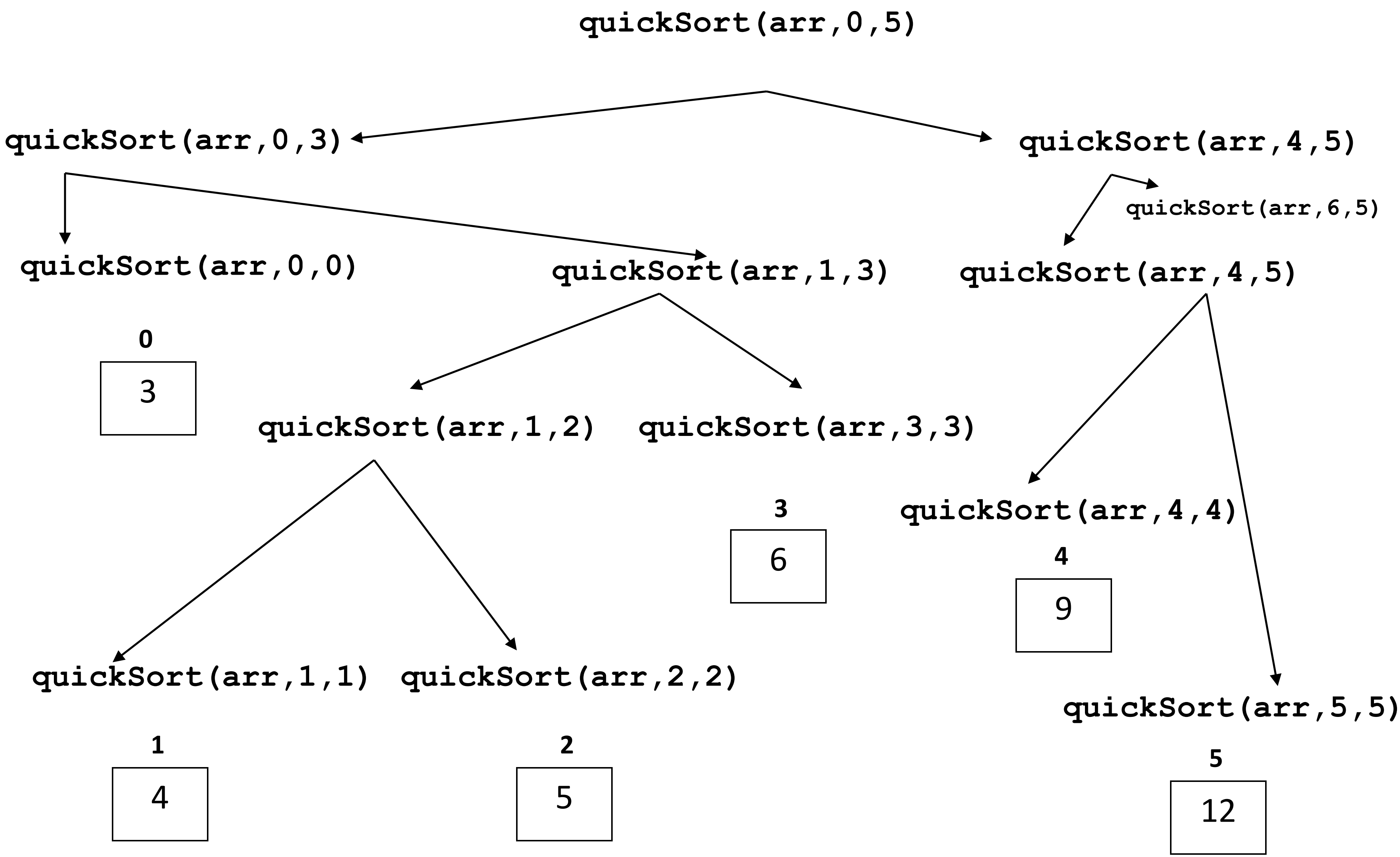
quickSort(arr,0,5)
quickSort(arr,0,3)
quickSort(arr,4,5)
quickSort(arr,6,5)
quickSort(arr,0,0)
quickSort(arr,1,3)
quickSort(arr,4,5)
0
3
4
5
9
12
quickSort(arr,1,2)
quickSort(arr,3,3)
3
6
quickSort(arr,1,1)
quickSort(arr,2,2)
1
2
4
5



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

Quiz 1

- Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this: 2 5 1 7 9 12 11 10 Which statement is correct?
- A The pivot could be either the 7 or the 9.
- B The pivot could be the 7, but it is not the 9
- C The pivot is not the 7, but it could be the 9
- D Neither the 7 nor the 9 is the pivot.

Quiz 2

- Let P be a QuickSort Program to sort numbers in ascending order using the first element as pivot. Let t1 and t2 be the number of comparisons made by P for the inputs {1, 2, 3, 4, 5} and {4, 1, 5, 3, 2} respectively. Which one of the following holds?
- A   t1 = 5
- B   t1 < t2
- C   t1 > t2
- D   t1 = t2

Quiz 3

- Discuss Which one of the following in place sorting algorithms needs the minimum number of swaps?

A	Quick sort
B	Insertion sort
C	Selection sort
D	Heap sort

Quiz 4

- You have an array of n elements. Suppose you implement quick sort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is ??

Quiz 5

- The Quicksort algorithm we have seen in class is the following:

```
QUICKSORT(A, p, r)

if p < r:
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)

i ← p − 1
for j ← p to r − 1
    if A[j] ≤ A[r]
        i ← i + 1
        exchange A[i] <−> A[j]
exchange A[i + 1] <−> A[r]
return i + 1
```

Quiz 5
Continues…

5-1- Make the <u>smallest possible change</u> to the algorithm such that it would <u>stop</u> with a <u>nearly-sorted array</u>. This is accomplished by not sorting subarrays with k elements or less. Mark your change on the copy of the algorithm given above.

5-2 With the change you made in part 1, and k=3, <u>how many recursive calls</u> to Quicksort are required to nearly-sort the following array?

3	6	1	4	7	2	8	5

5-3- With the change you made in part 1, k = 3, and the array given in part 2, compute the <u>final array</u> obtained by Quicksort