



Week 7-8: Sorting Algorithms

CS-250 Data Structure and Algorithms

DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

Sorting

- We have gone to great trouble to keep lists of elements in sorted order:
 - Student records sorted by ID number
 - Integers sorted from smallest to largest
 - Words sorted alphabetically
- **Sorting** : Putting an unordered list of data items into order , is a very common and useful operation.
- The goal of sorting algorithms is mainly to facilitate searching.

Sorting

- As much as 25% of computing time is spent on sorting. Sorting aids searching and matching entries in a list.
- Sorting Definitions:
 - Given a list of records (R_1, R_2, \dots, R_n) where each record R_i has a key K_i .
 - An ordering relationship ($<$) between two key values, either $x = y$, $x < y$, or $x > y$. **Ordering relationships are transitive:** $x < y, y < z$, then $x < z$.
 - Find a permutation (p) of the keys such that $K_{p(i)} \leq K_{p(i+1)}$, for $1 \leq i < n$.
 - The desired ordering is: $(R_{p(1)}, R_{p(2)}, \dots, R_{p(n)})$

Sorting

- **Stability:** Since a list could have several records with the same key, the permutation is not unique. A permutation p is stable if:
 - sorted: $K_p(i) \leq K_p(i+1)$, for $1 \leq i < n$.
 - stable: if $i < j$ and $K_i = K_j$ in the input list, then R_i precedes R_j in the sorted list.
- An internal sort is one in which the list is small enough to sort entirely in main memory.
- An external sort is one in which the list is too big to fit in main memory.

Applications

- **Searching:** Binary search lets you test whether an item is in a dictionary in $O(\log n)$ time.
 - Speeding up searching is perhaps the most important application of sorting.
- **Closest pair:** Given n numbers, find the pair which are closest to each other.
 - Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.
- **Element uniqueness:** Given a set of n items, are they all unique or are there any duplicates?
 - Sort them and do a linear scan to check all adjacent pairs. This is a special case of closest pair above.

Applications

- **Frequency distribution:** Given a set of n items, which element occurs the largest number of times?
 - Sort them and do a linear scan to measure the length of all adjacent runs.
- **Median and Selection:** What is the k th largest item in the set?
 - Once the keys are placed in sorted order in an array, the k th largest can be found in constant time by simply looking in the k th position of the array.

Applications

- If you are trying to minimize the amount of space a text file is taking up, it is silly to assign each letter the same length (i.e., one byte) code.
 - Example: e is more common than q, a is more common than z.
- If we were storing English text, we would want a and e to have shorter codes than q and z.
 - To design the best possible code, the first and most important step is to sort the characters in order of frequency of use.

Sorting

- There are so many sorting algorithms
 - Selection Sort
 - Bubble Sort
 - Insertion Sort
 - Merge Sort
 - Quick Sort
 - And So on.

Selection Sort

1. for $i := 1$ to $n-1$ do
 2. begin
 3. $\text{min} := i$;
 4. for $j := i + 1$ to n do
 5. if $a[j] < a[\text{min}]$ then $\text{min} := j$;
 6. $\text{swap}(a[\text{min}], a[i])$;
 7. end;
- Selection sort is linear for files with large record and small keys

<u>3</u>	6	2	7	4	8	1	5
1	<u>6</u>	2	7	4	8	3	5
1	2	<u>6</u>	7	4	8	3	5
1	2	3	<u>7</u>	4	8	6	5
1	2	3	4	<u>7</u>	8	6	5
1	2	3	4	5	<u>8</u>	6	7
1	2	3	4	5	6	<u>8</u>	7
1	2	3	4	5	6	7	8

- n exchanges
- $n^2/2$ comparisons

Insertion Sort

1. for $i := 2$ to n do
 2. begin
 3. $v := a[i]$; $j := i$;
 4. while $a[j-1] > v$ do
 5. begin
 6. $a[j] := a[j-1]$; $j := j-1$
 7. end;
 8. $a[j] := v$;
 9. end;
- linear for "**almost sorted**" files
 - Binary insertion sort: Reduces comparisons but not moves.
 - List insertion sort: Use linked list, no moves, but must use sequential search.

3 6 2 7 4 8 1
5

2 3 6 7 4 8 1
5

2 3 4 6 7 8 1
5

1 2 3 4 6 7 8
5

1 2 3 4 5 6 7
8

$n^2/4$ exchanges

$n^2/4$ comparisons

Bubble Sort

1. for $i := n$ down to 1 do
 2. for $j := 2$ to i do
 3. if $a[j-1] > a[j]$
 4. then $\text{swap}(a[j], a[j-1]);$
- $n^2/4$ exchanges
 - $n^2/2$ comparisons
 - Bubble can be improved by adding a flag to check if the list has already been sorted.

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

Shell Sort

```
1. for(int gap = n / 2; gap > 0; gap /= 2)
2.     for(int i = gap; i < n; i++)
3.         temp = a[i];
4.         int j = i;
5.         for( ; j >= gap && temp < a[j - gap]; j
           -= gap )
6.             a[ j ] = a[ j - gap ];
7.             a[ j ] = temp;
8.     end
```

- Shell sort is a simple extension of insertion sort, which gains speeds by allowing exchange of elements that are far apart.
- Idea: rearrange list into gap-sorted (for any sequence of values of gap that ends in 1.)
- Shell sort never does more than $n^{1.5}$ comparisons.
- The analysis of this algorithm is hard. Two conjectures of the complexity are $n(\log n)^2$ and $n^{1.25}$

Shell Sort

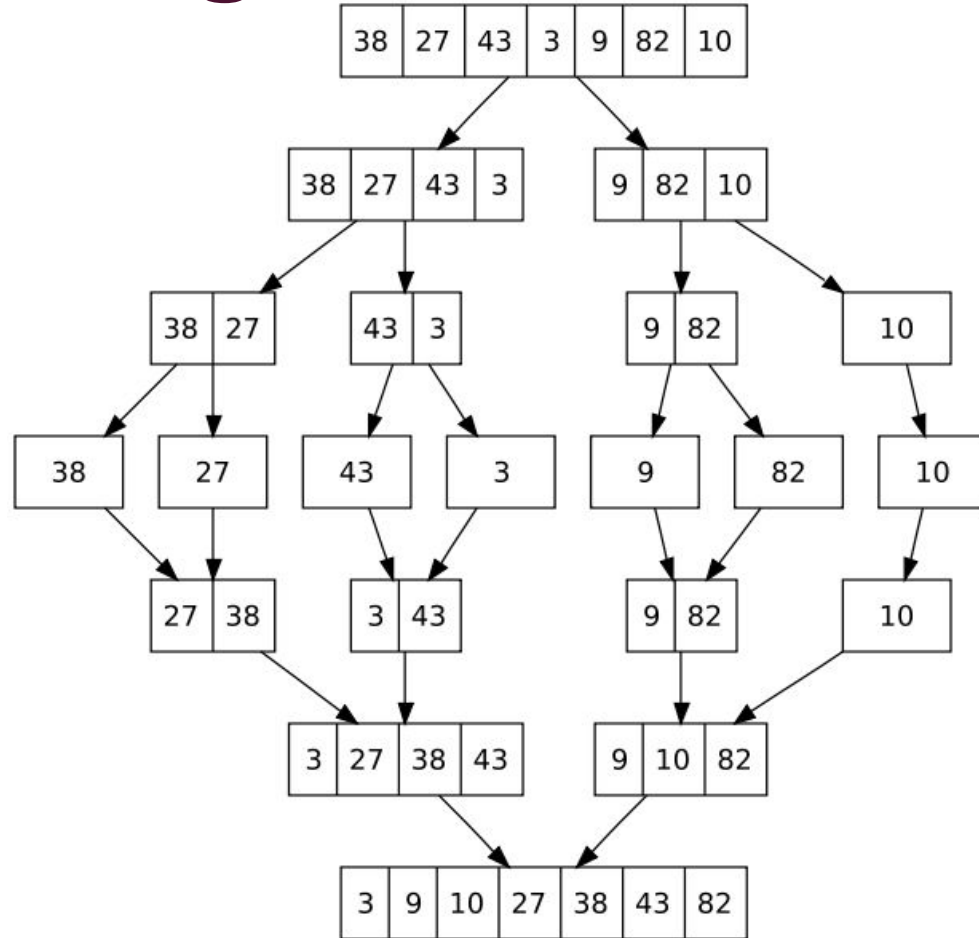
- 46 2 83 41 102 5 17 31 64 49 18
 - Gap of five. Sort sub array with 46, 5, and 18
5 2 83 41 102 18 17 31 64 49 46
 - Gap still five. Sort sub array with 2 and 17
5 2 83 41 102 18 17 31 64 49 46
 - Gap still five. Sort sub array with 83 and 31
5 2 31 41 102 18 17 83 64 49 46
 - Gap still five Sort sub array with 41 and 64
5 2 31 41 102 18 17 83 64 49 46
 - Gap still five. Sort sub array with 102 and 49
5 2 31 41 49 18 17 83 64 102 46
- 5 2 31 41 49 18 17 83 64 102 46
 - Gap now 2: Sort sub array with 5 31 49 17 64 46
5 2 17 41 31 18 46 83 49 102 64
 - Gap still 2: Sort sub array with 2 41 18 83 102
5 2 17 18 31 41 46 83 49 102 64
 - Gap of 1 (Insertion sort)
 - 2 5 17 18 31 41 46 49 64 83 102
 - Array sorted

Shell Sort

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25

- Initial gap = length / 2 = 16 / 2 = 8
- initial sub arrays indices:
 - {0, 8}, {1, 9}, {2, 10}, {3, 11}, {4, 12}, {5, 13}, {6, 14}, {7, 15}
 - next gap = 8 / 2 = 4
 - {0, 4, 8, 12}, {1, 5, 9, 13}, {2, 6, 10, 14}, {3, 7, 11, 15}
 - next gap = 4 / 2 = 2
 - {0, 2, 4, 6, 8, 10, 12, 14}, {1, 3, 5, 7, 9, 11, 13, 15}
 - final gap = 2 / 2 = 1

Merge Sort



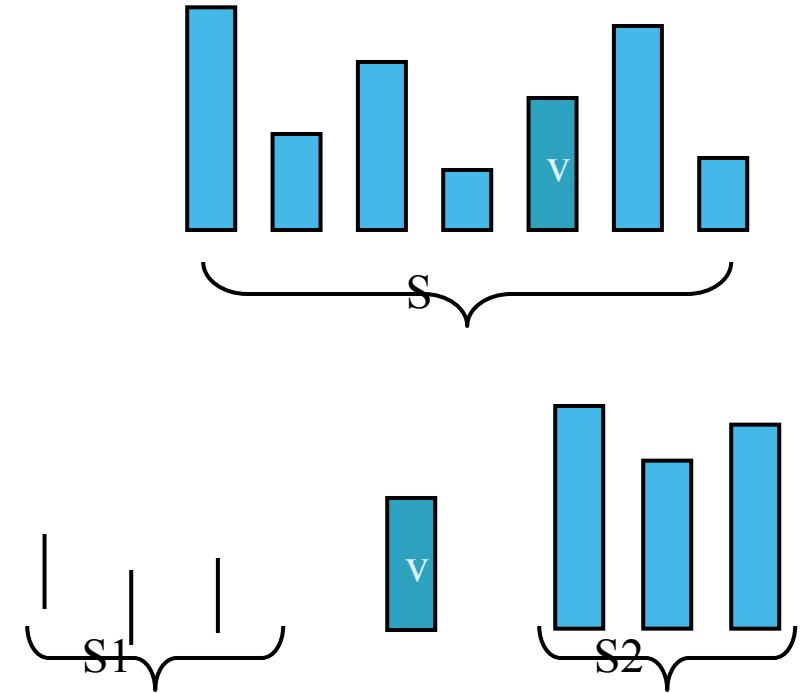
1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 2 split into into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together
 - When implementing one temporary array is used instead of multiple temporary arrays.
 - Why?

Merge Sort

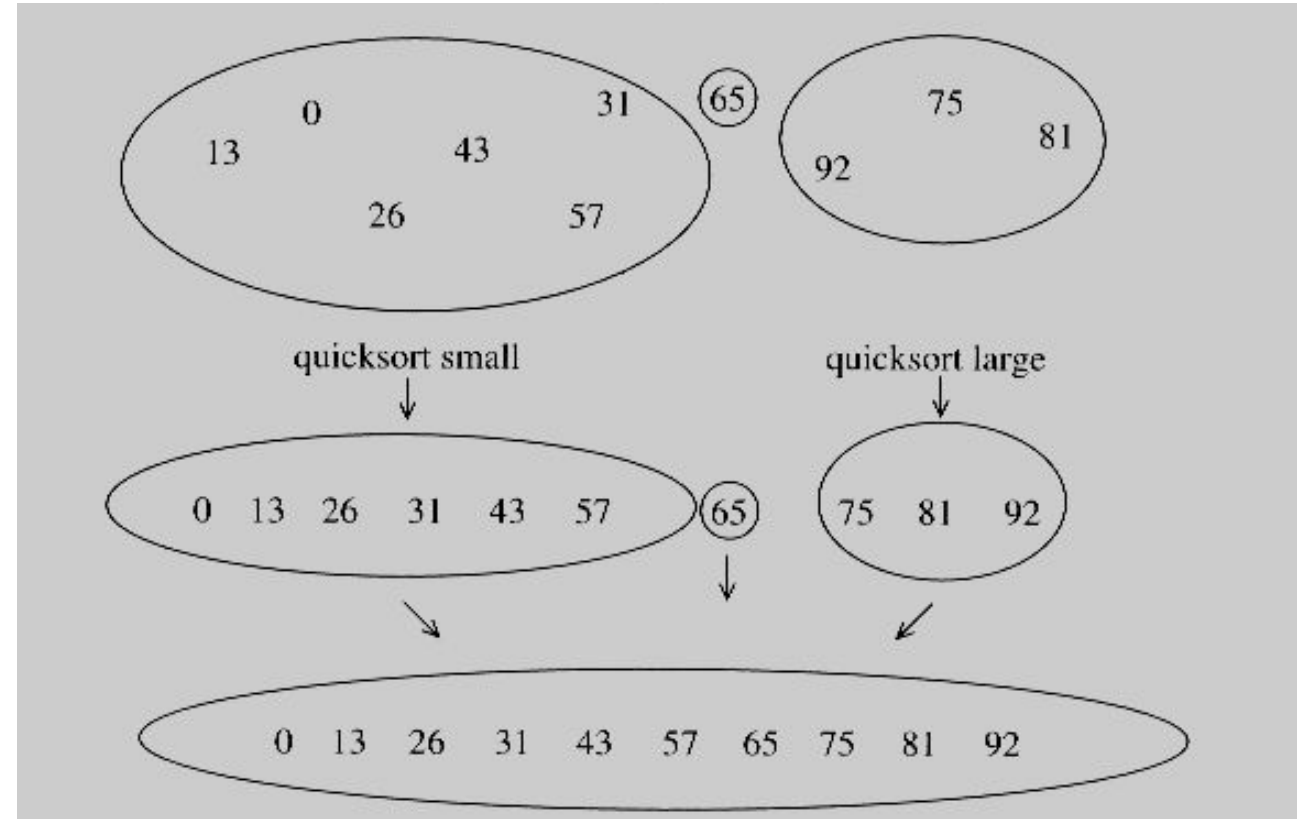
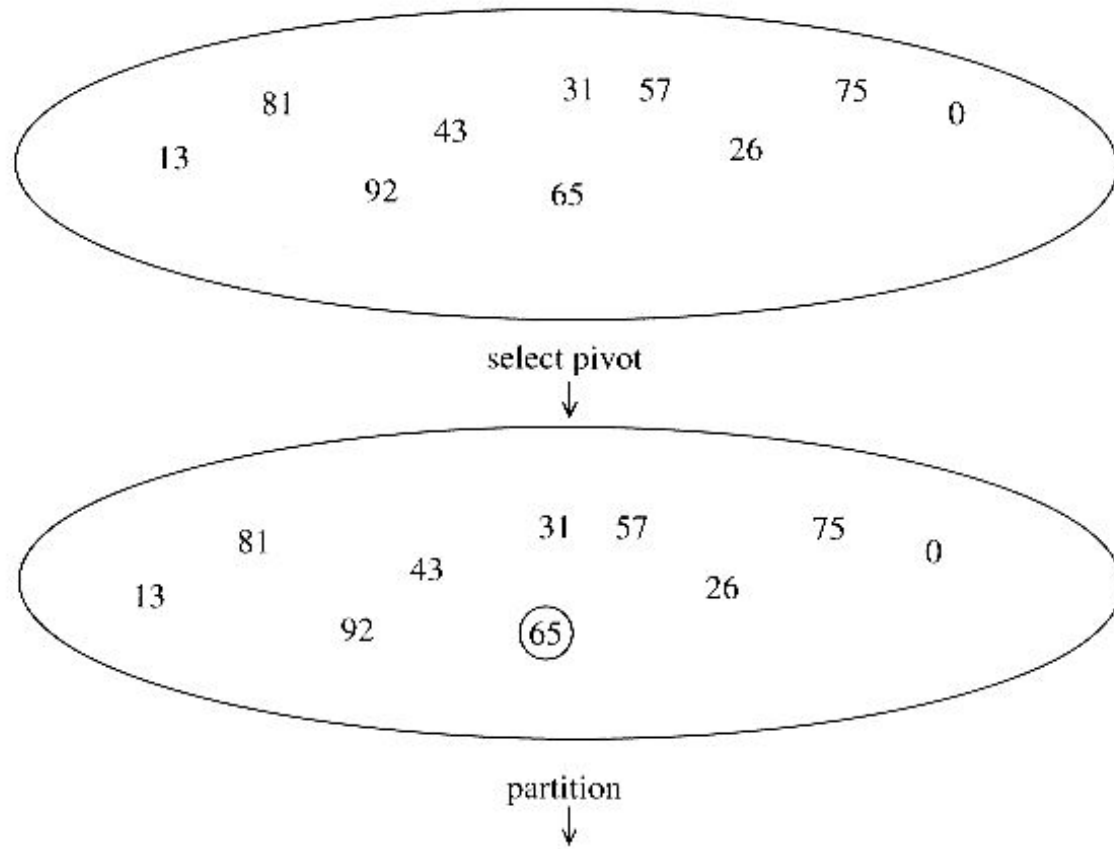
```
sort(int A[], int temp[], int low, int high)
1.  {  if( low < high)
2.      int center = (low + high) / 2;
3.      sort(list, temp, low, center);
4.      sort(list, temp, center + 1, high);
5.      merge(list, temp, low, center + 1,
6.          high);
7.  }
```


QUICKSORT

- Divide step:
 - Pick any element (pivot) v in S
- Partition $S - \{v\}$ into two disjoint groups
 - $S1 = \{x \in S - \{v\} \mid x \leq v\}$
 - $S2 = \{x \in S - \{v\} \mid x \geq v\}$
- Conquer step: recursively sort $S1$ and $S2$
- Combine step: the sorted $S1$ (by the time returned from recursion), followed by v , followed by the sorted $S2$ (i.e., nothing extra needs to be done)
- Although mergesort is $O(n \log n)$, it is quite inconvenient for implementation with arrays, since we need space to merge.
- In practice, the fastest sorting algorithm is Quicksort, which uses partitioning as its main idea.



To simplify, we may assume that we don't have repetitive elements, So to ignore the 'equality' case!



QUICKSORT

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

1. quickSort(array, lower, upper)
2. { if (lower >= upper) // Base Case
3. { we're done }
4. else
5. { partition array around pivot value array[lower]
6. pos contains the new location of pivot value
7. quickSort array up to pos: quickSort(array, lower, pos)
8. quickSort array after pos: quickSort(array, pos+1, upper)}
9. }

3 6 2 7 4 8 1 5
3 1 2 7 4 8 6 5
3 1 2 4 7 8 6 5
3 1 2 7 4 8 6 5
3 1 2 4 5 8 6 7
• • • •
3 1 2 4 5 6 7 8
• • • •
1 2 3 4 5 6 7 8

The efficiency of quicksort
can be measured by the
number of
comparisons.

```
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  $\leq$  pivot
            all values right of pivot are  $\geq$  pivot
            return pos
        end pos
    }
}
```

`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)`

`pivot = ?`

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

Partition Initialization...



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

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

pivot=6

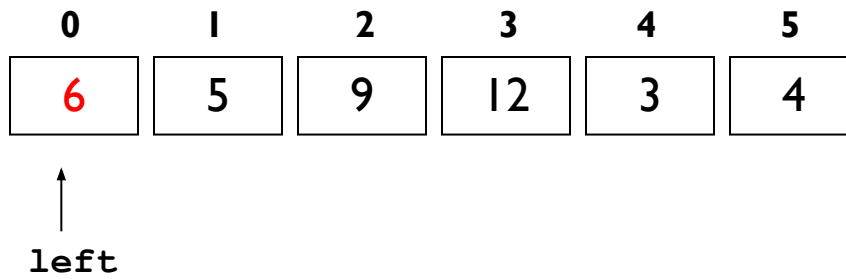
0	1	2	3	4	5
6	5	9	12	3	4

Partition Initialization...

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`



Partition Initialization...

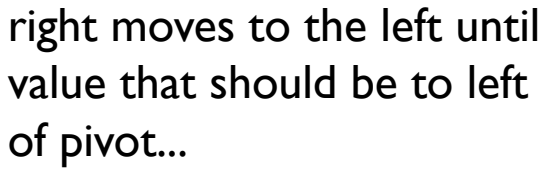


pivot=6





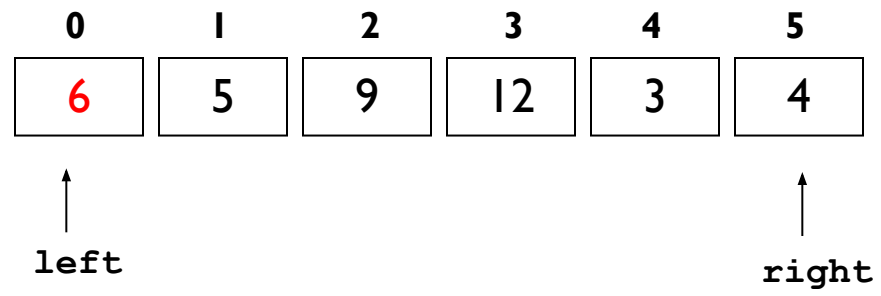
pivot=6



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

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

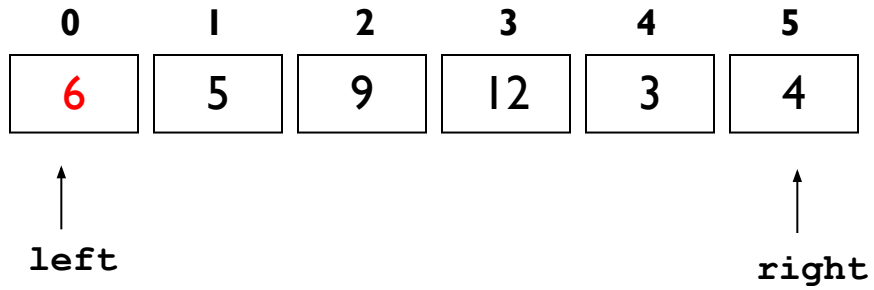
pivot=6



`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

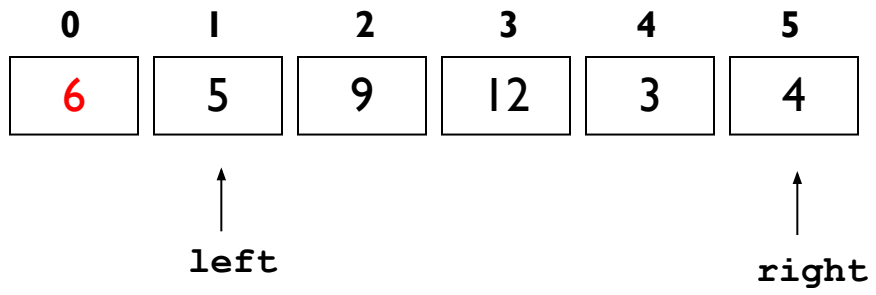


left moves to the right until
value that should be to right
of pivot...

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

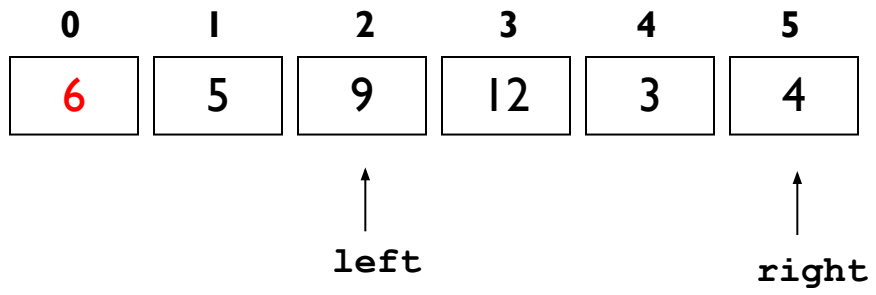
`pivot=6`



`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

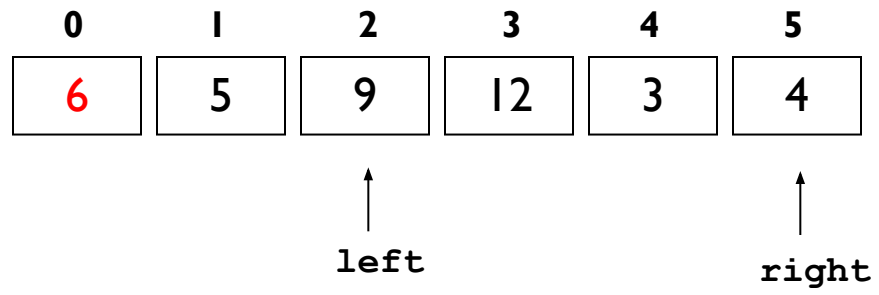
`pivot=6`



`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

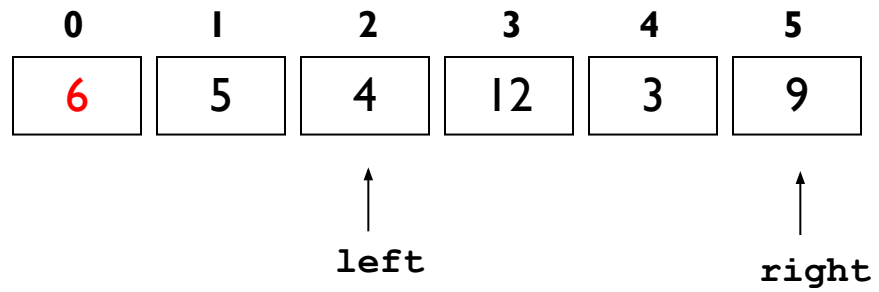


swap arr[left] and arr[right]

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

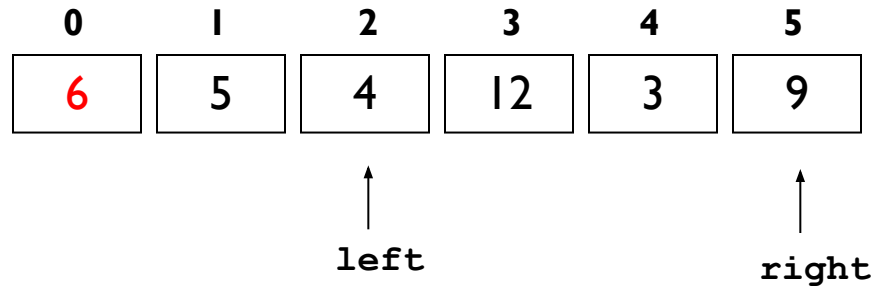


repeat right/left scan
UNTIL left & right cross

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

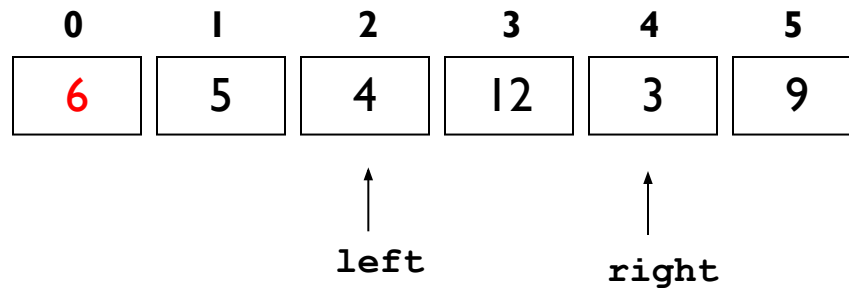


right moves to the left until
value that should be to left
of pivot...

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

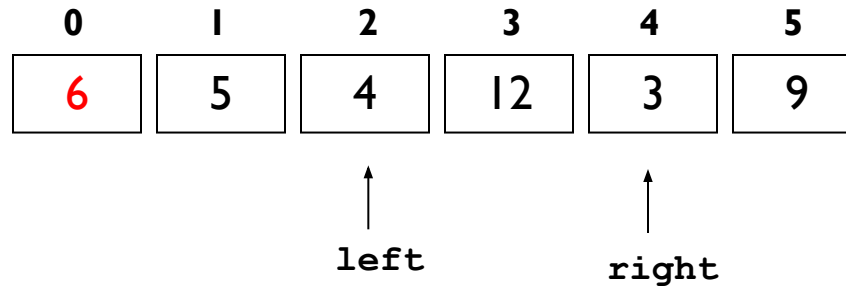
`pivot=6`



`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

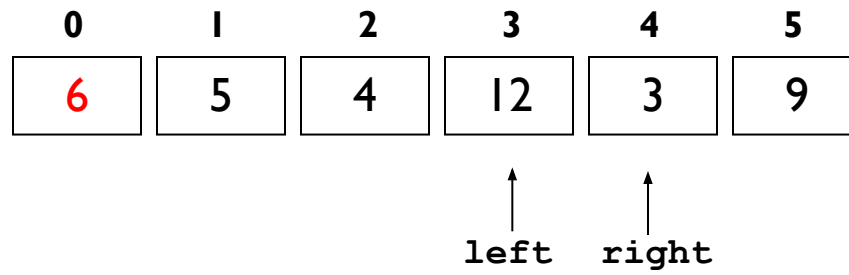


left moves to the right until
value that should be to right
of pivot...

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

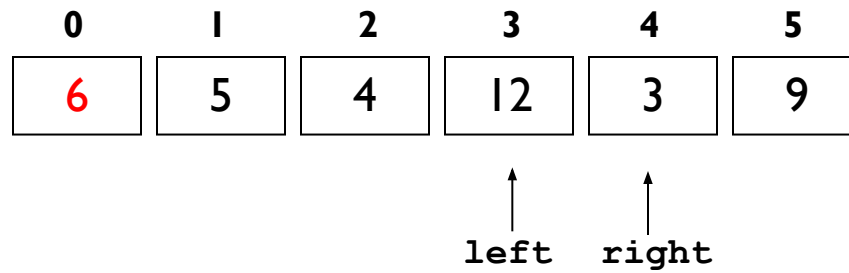
`pivot=6`



`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

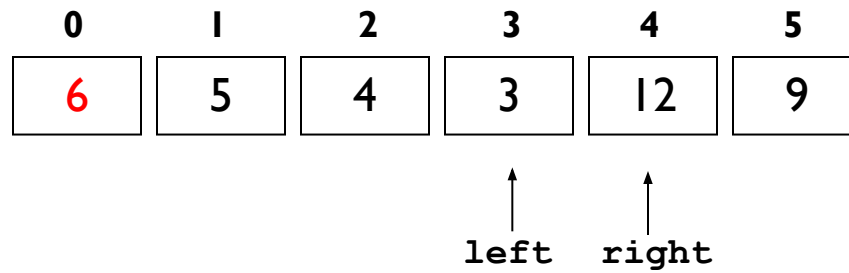


`swap arr[left] and arr[right]`

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

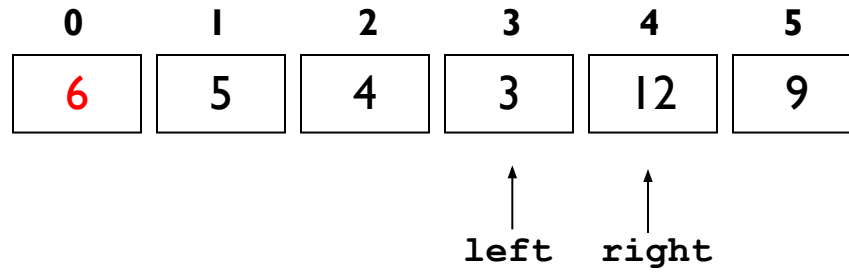


`swap arr[left] and arr[right]`

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

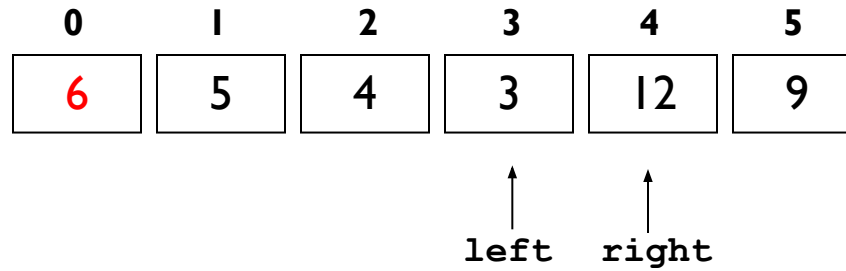


repeat right/left scan
UNTIL left & right cross

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`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`

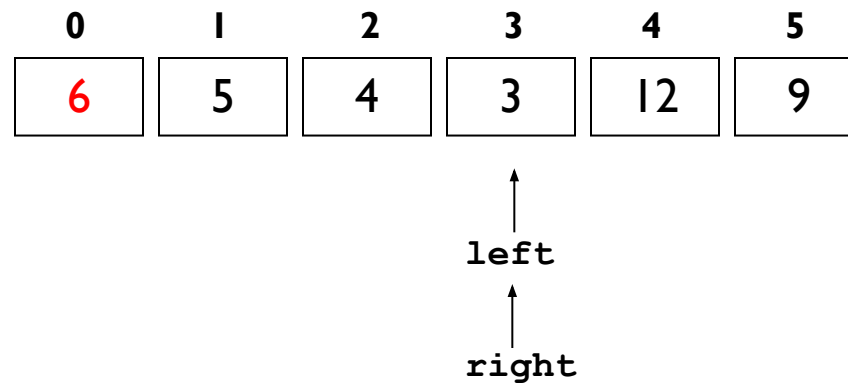
0	1	2	3	4	5
6	5	4	3	12	9

↑
left
↑
right

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

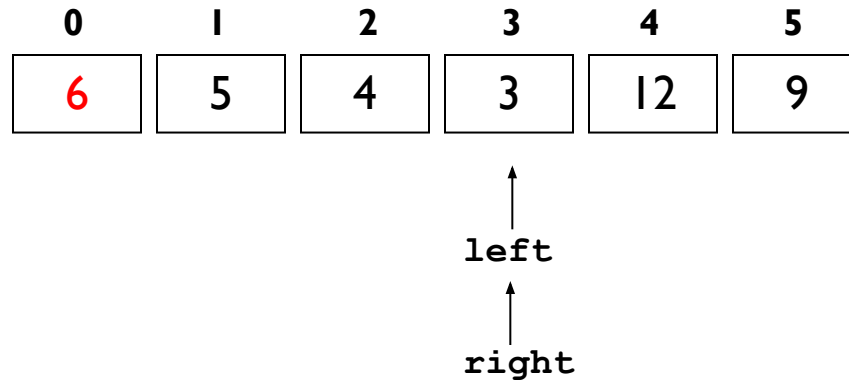


right & left CROSS!!!

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`



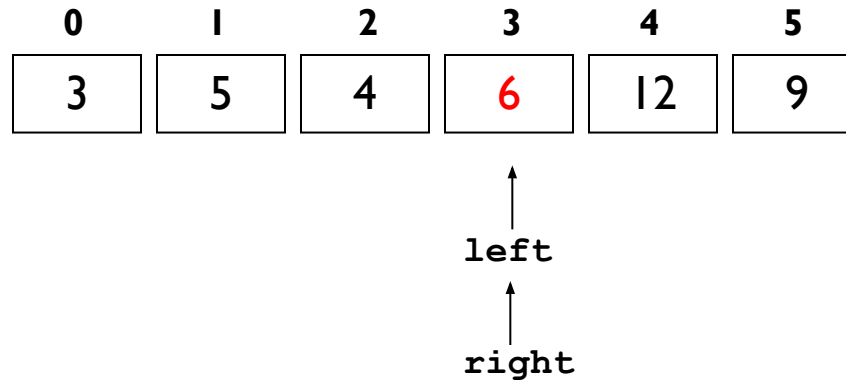
right & left CROSS!!!

1 - Swap pivot and arr[right]

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`



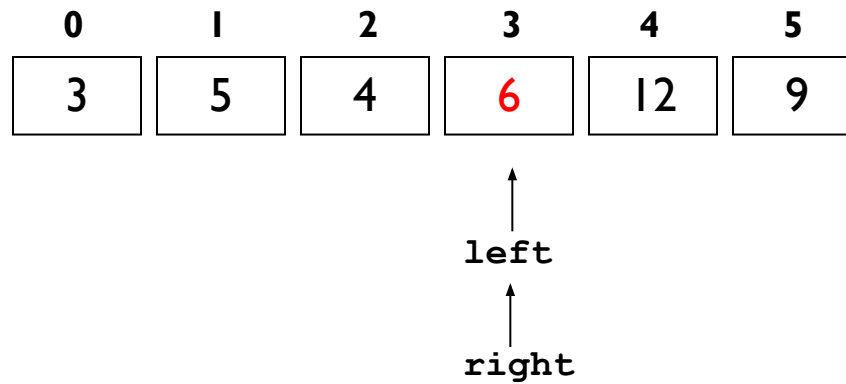
right & left CROSS!!!

! - Swap pivot and arr[right]

`quickSort(arr, 0, 5)`

`partition(arr, 0, 5)`

`pivot=6`

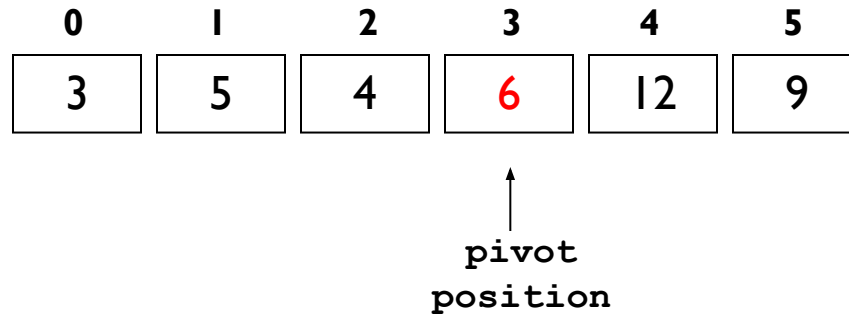


right & left CROSS!!!

- 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)`

`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

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)`

`quickSort(arr, 4, 5)`

0	1	2	3
3	5	4	6

↑
`left`

4	5
12	9

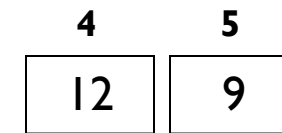
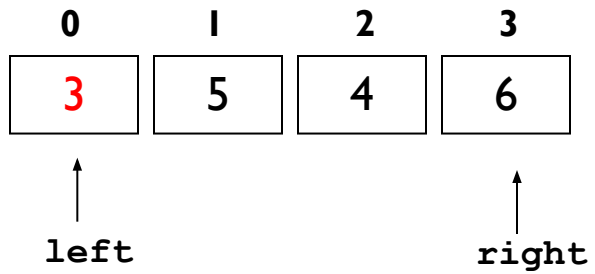
Partition Initialization...

`quickSort(arr, 0, 5)`

`quickSort(arr, 0, 3)`

`partition(arr, 0, 3)`

`quickSort(arr, 4, 5)`

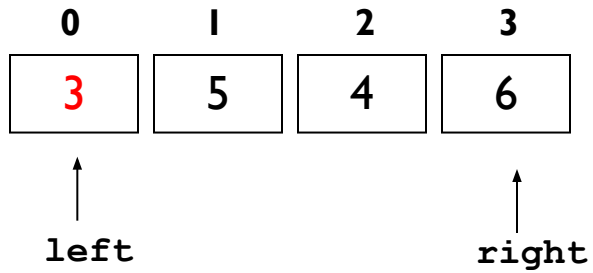


Partition Initialization...

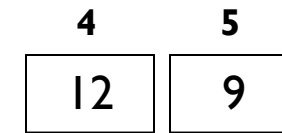
`quickSort(arr, 0, 5)`

`quickSort(arr, 0, 3)`

`partition(arr, 0, 3)`



`quickSort(arr, 4, 5)`



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)`

`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

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)`

`quickSort(arr, 4, 5)`

0	1	2	3
3	5	4	6

↑
left
↑
right

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)`

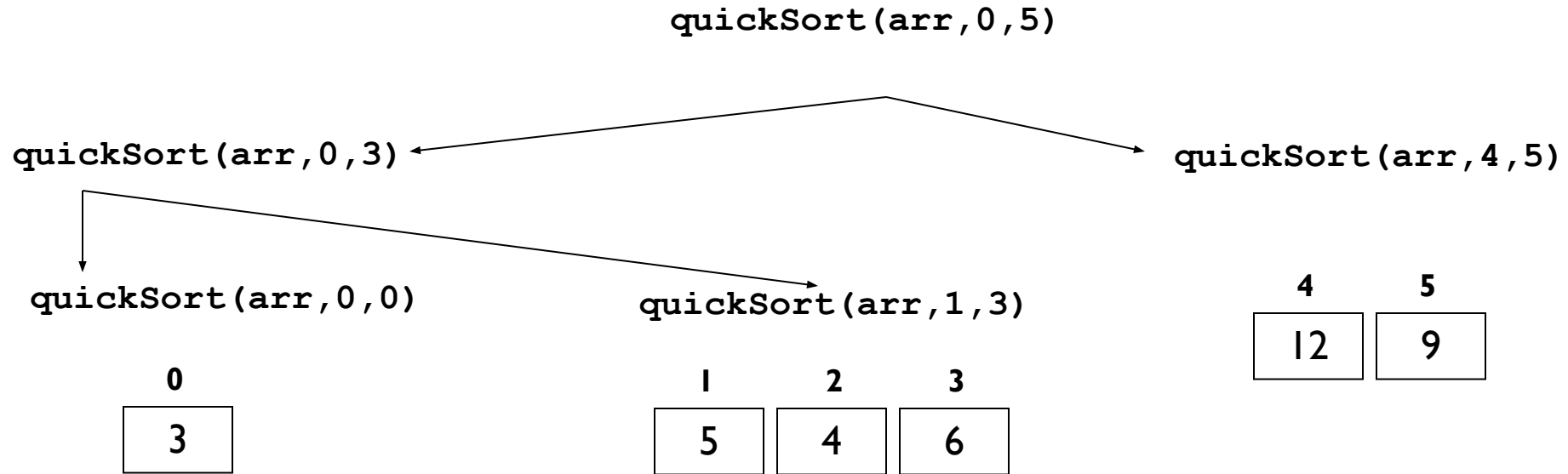
`quickSort(arr, 0, 3)`

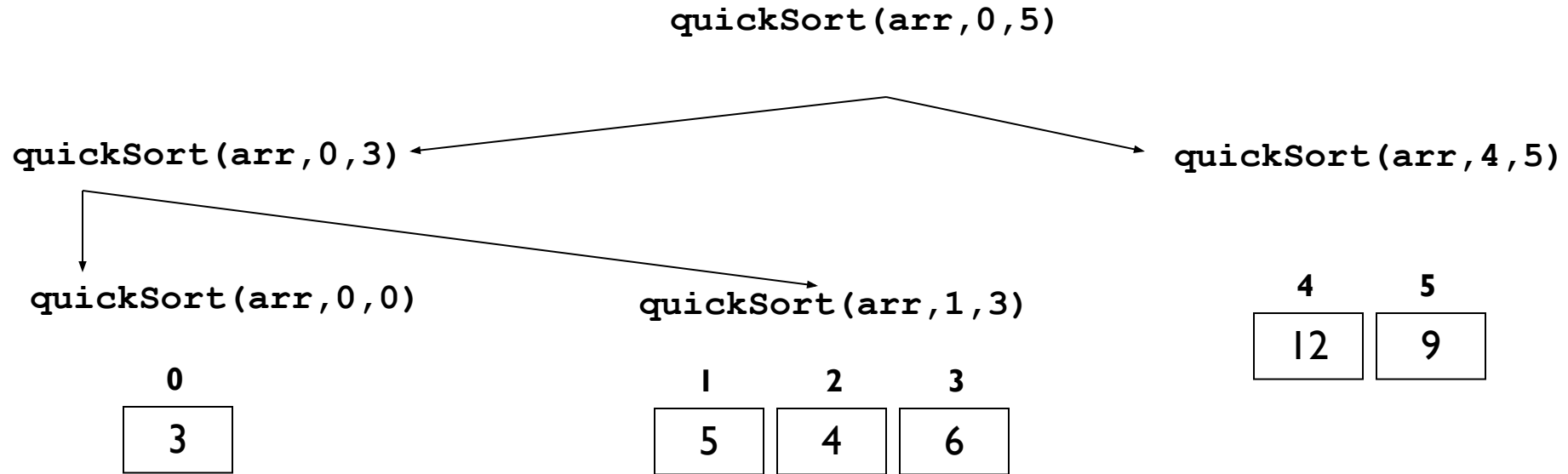
`quickSort(arr, 4, 5)`

0	1	2	3
3	5	4	6

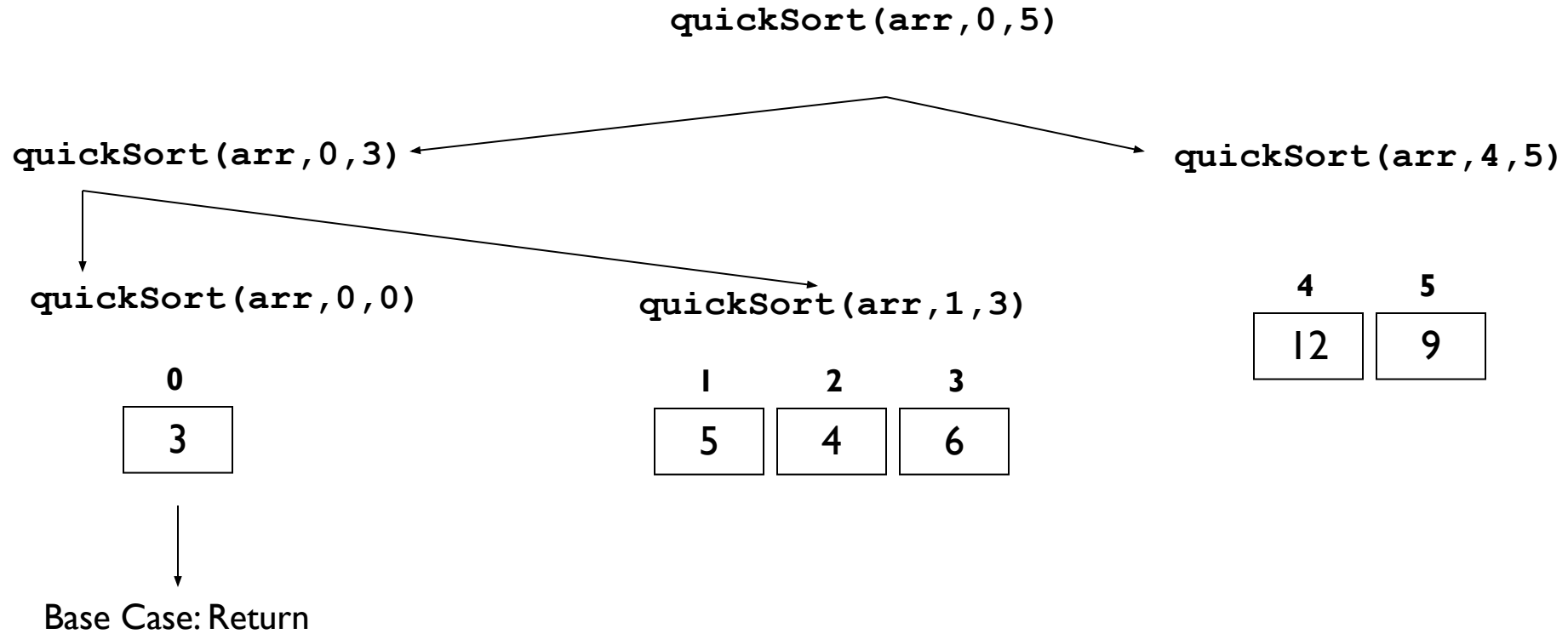
4	5
12	9

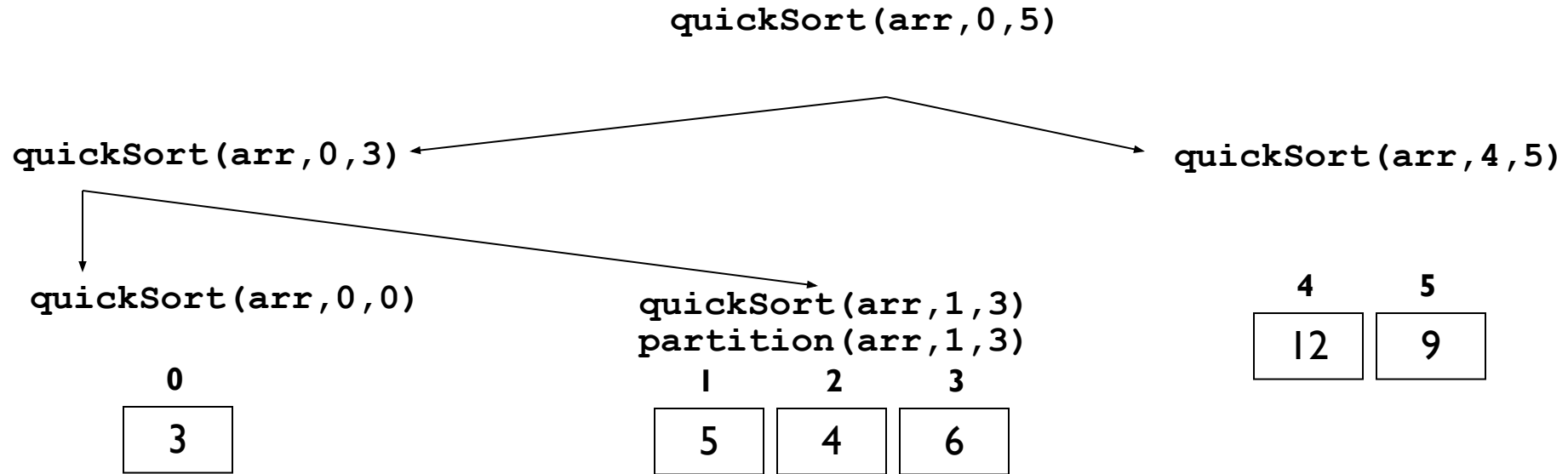
Recursive calls to `quickSort()`
using partitioned array...



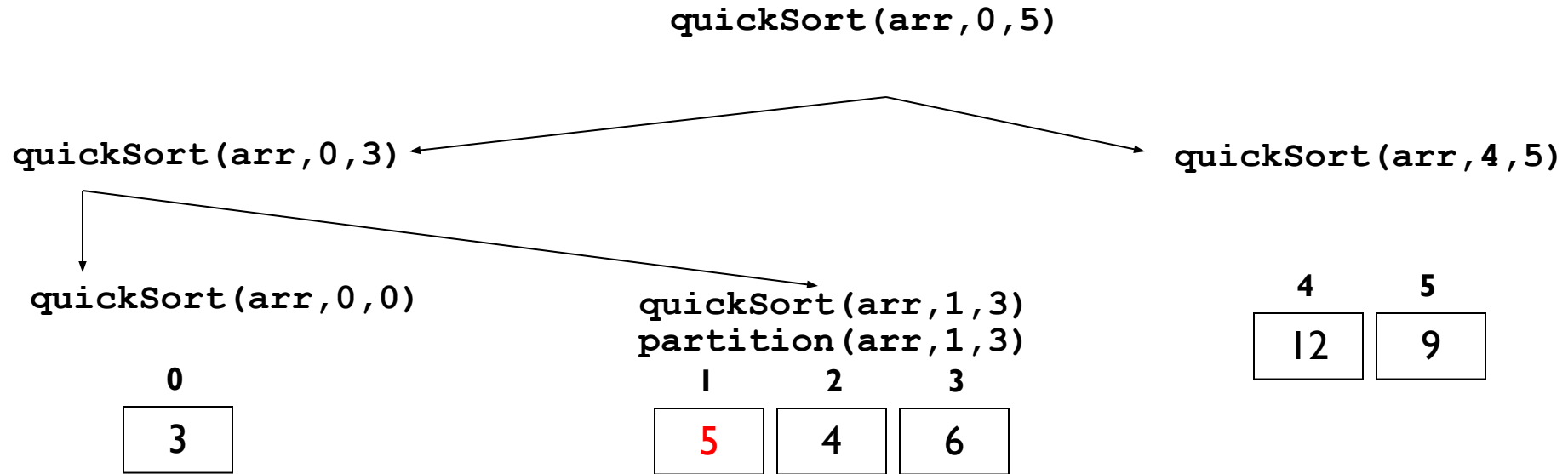


Base case triggered...
halting recursion.

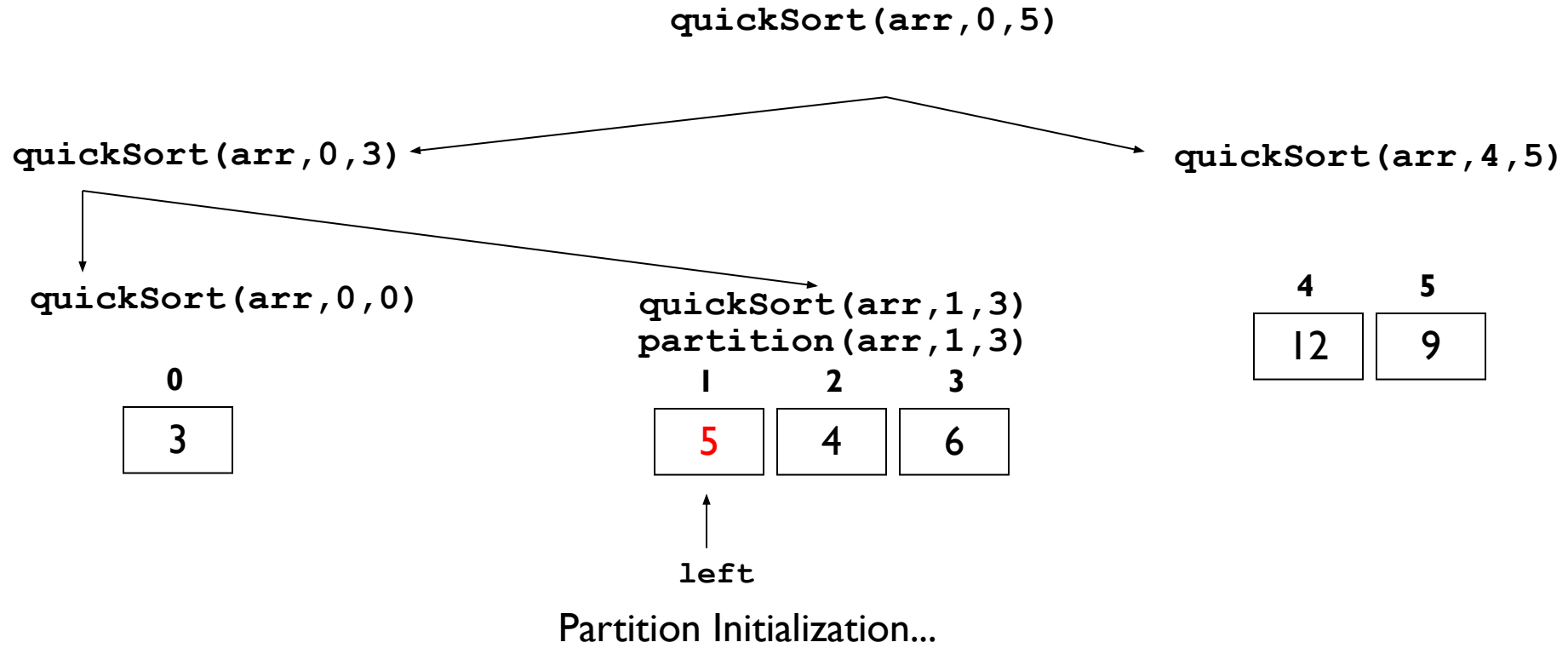


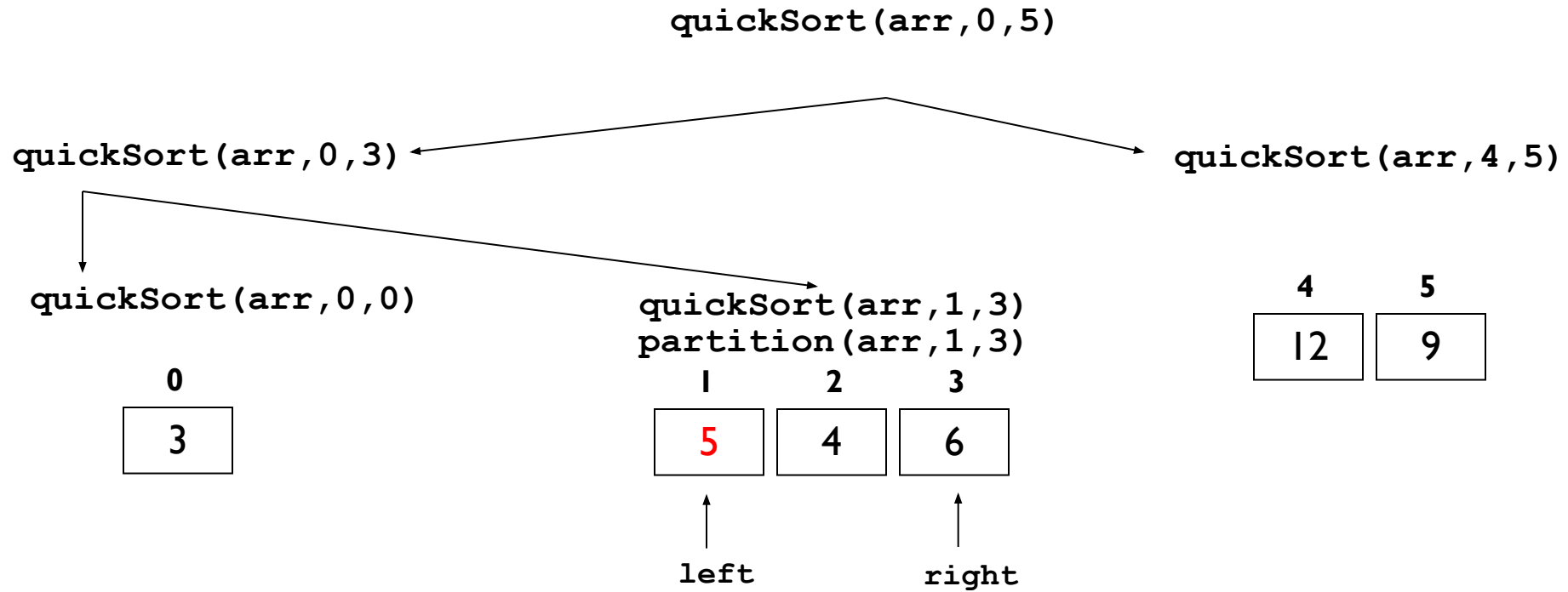


Partition Initialization...

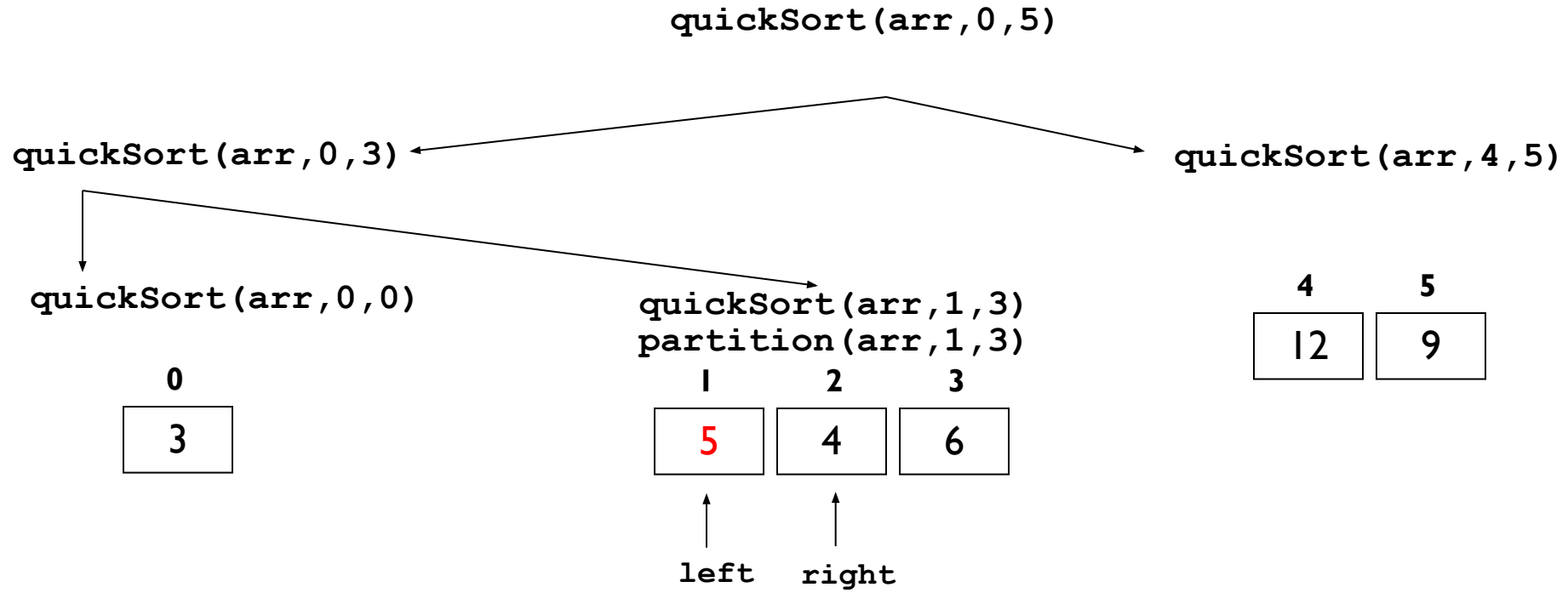


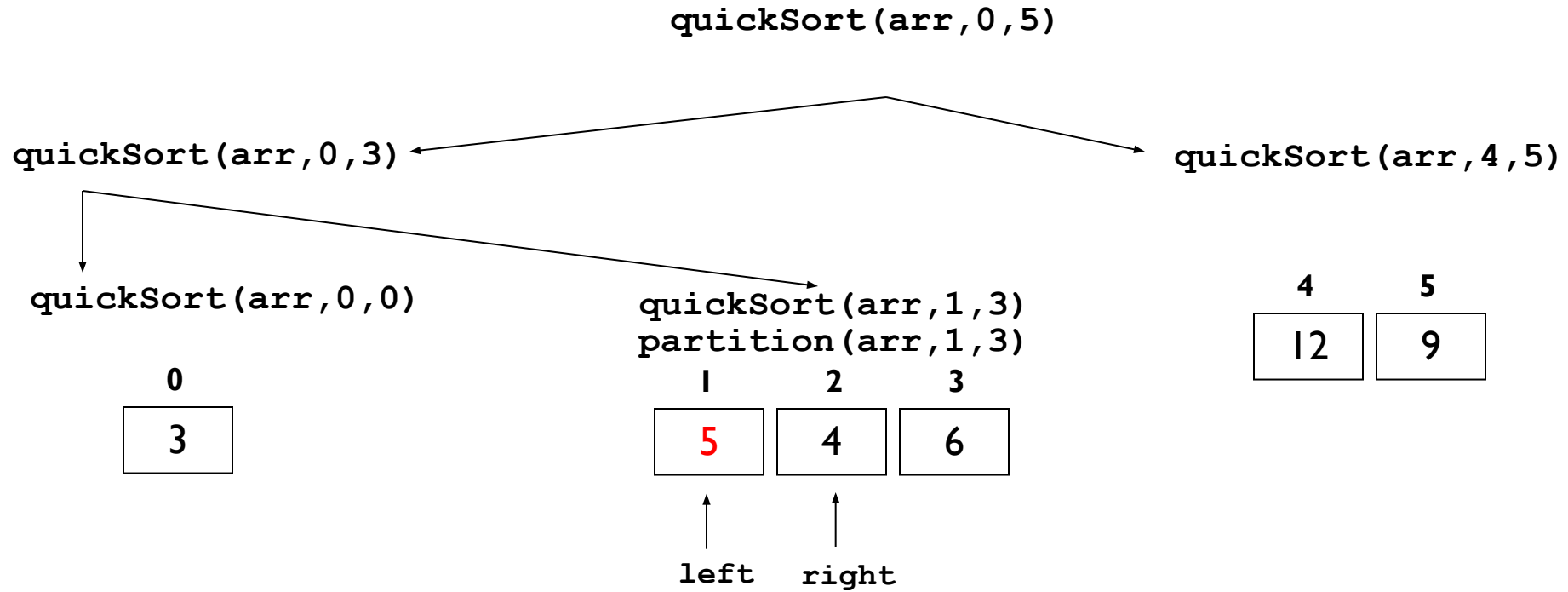
Partition Initialization...



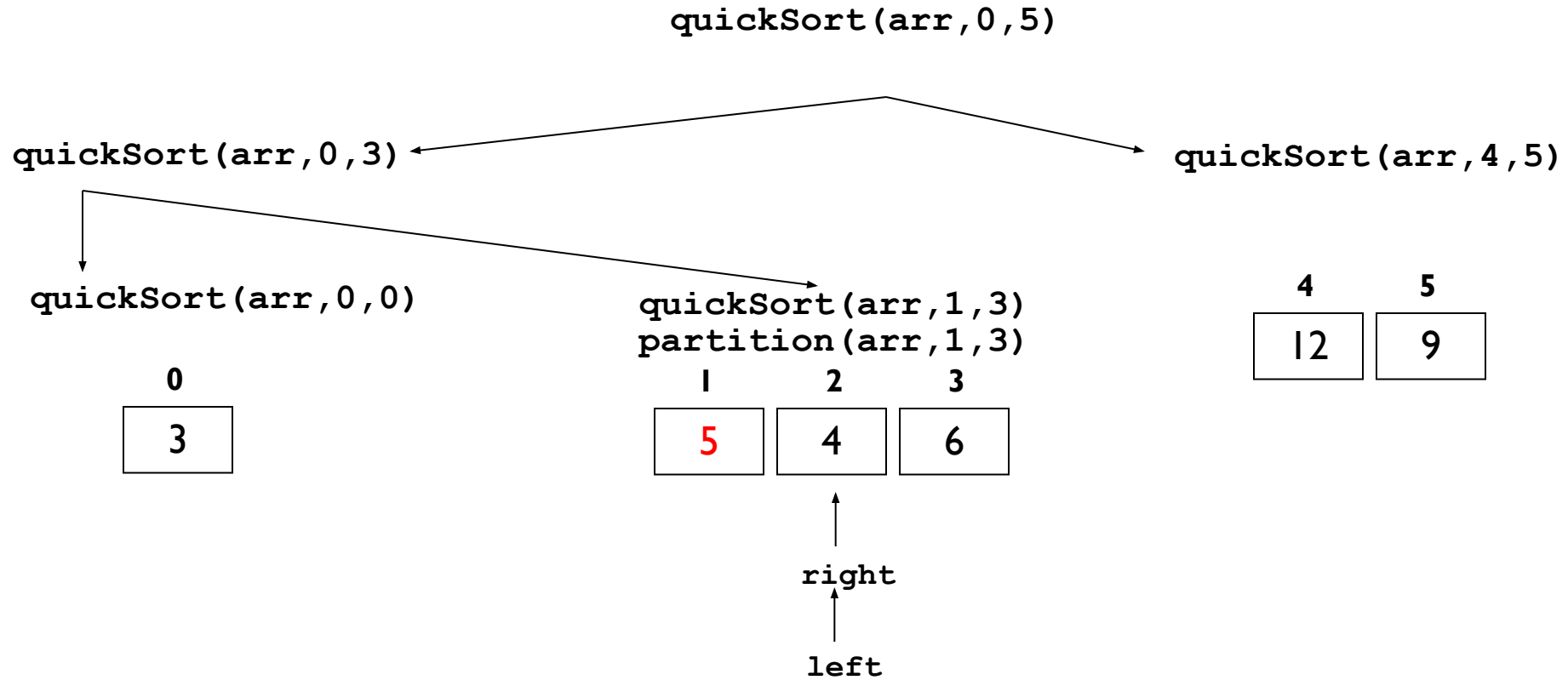


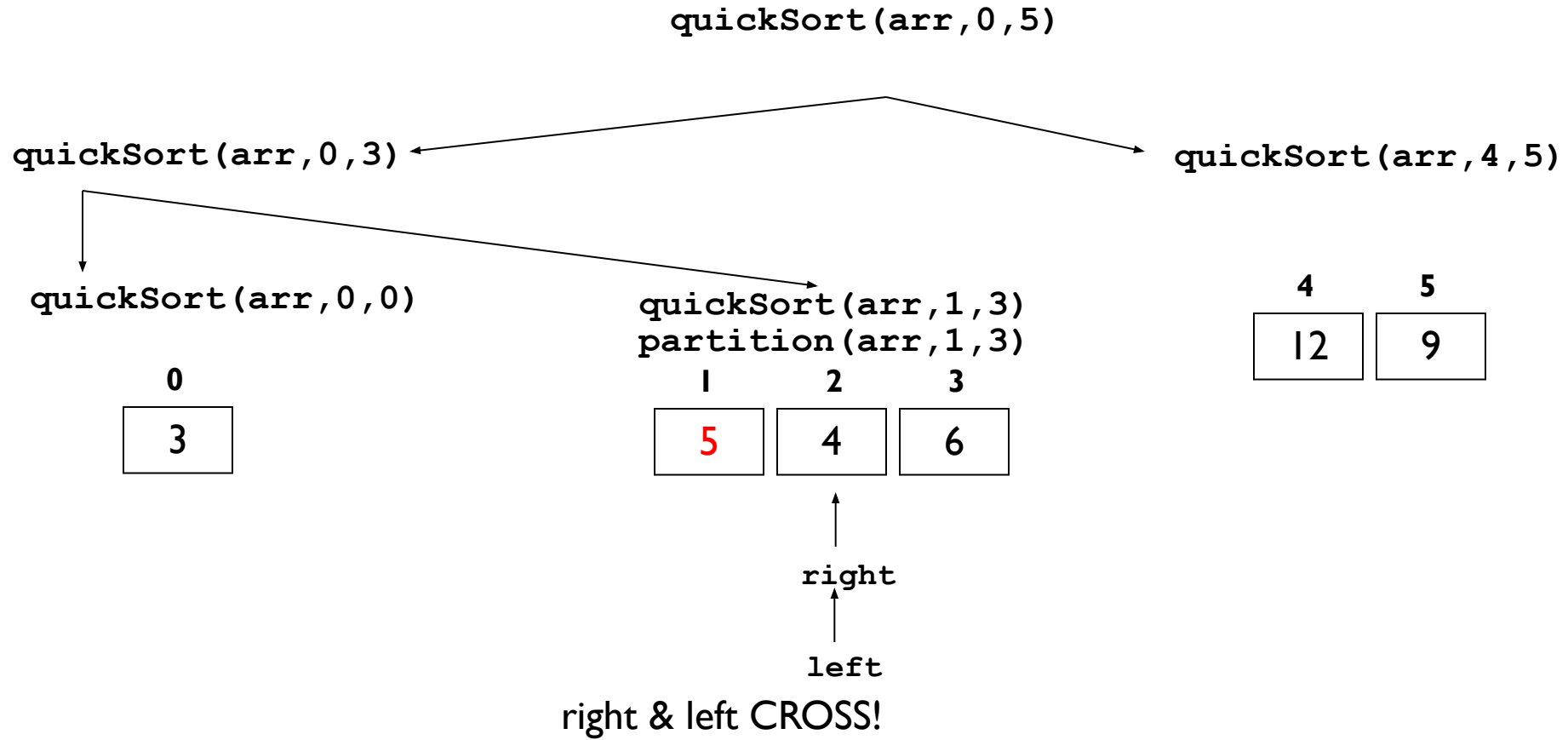
right moves to the left until
value that should be to left
of pivot...

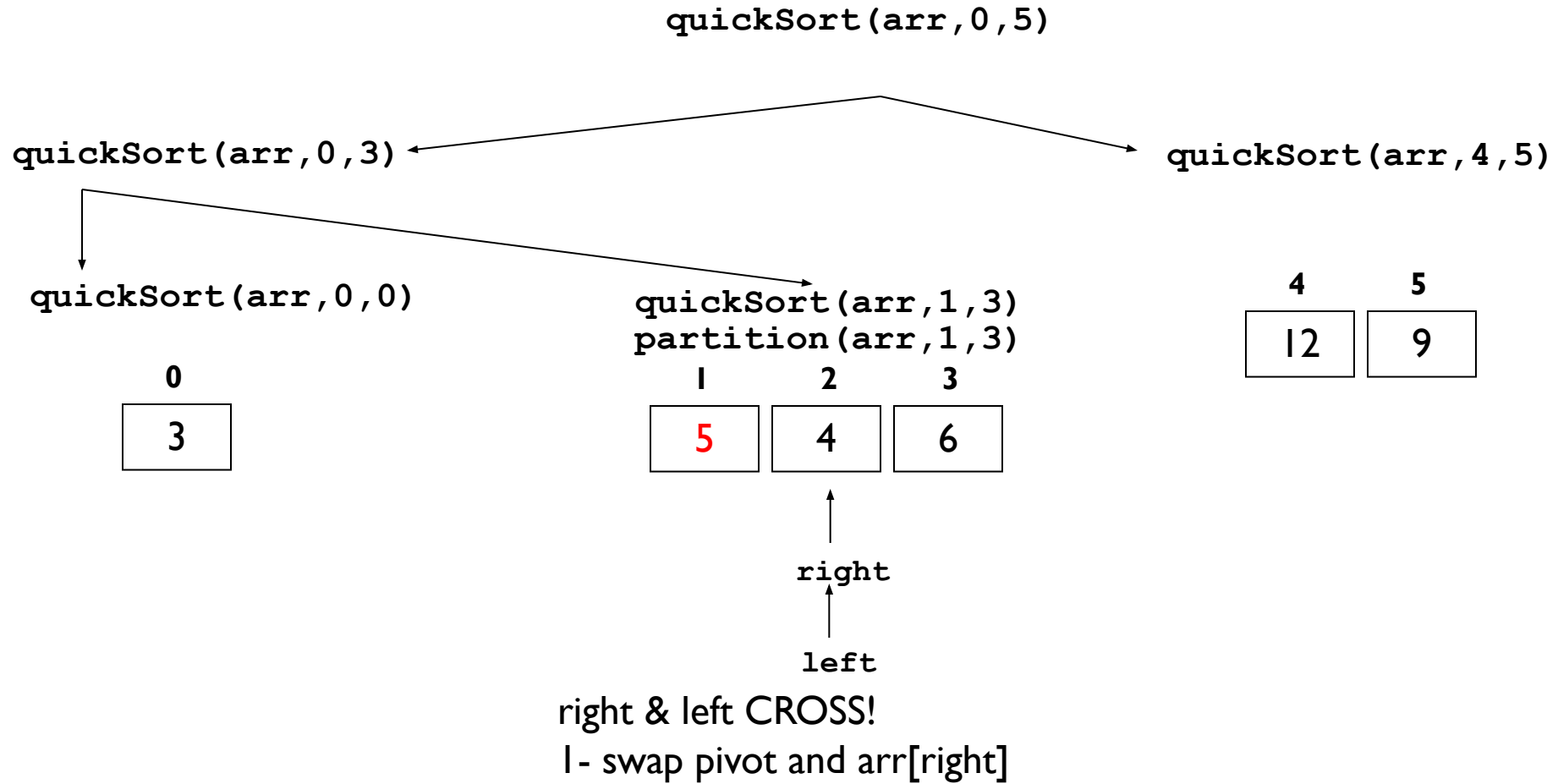


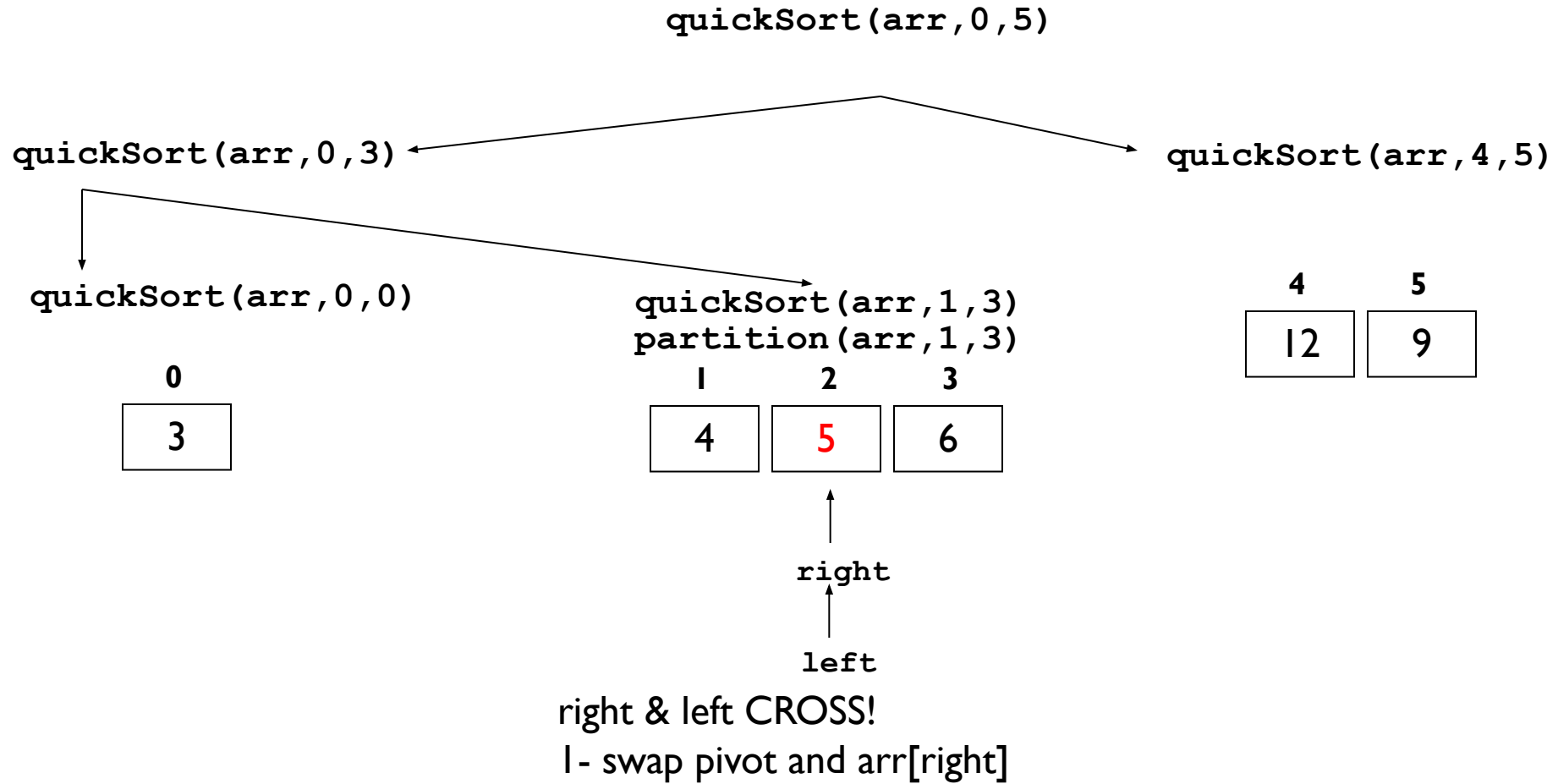


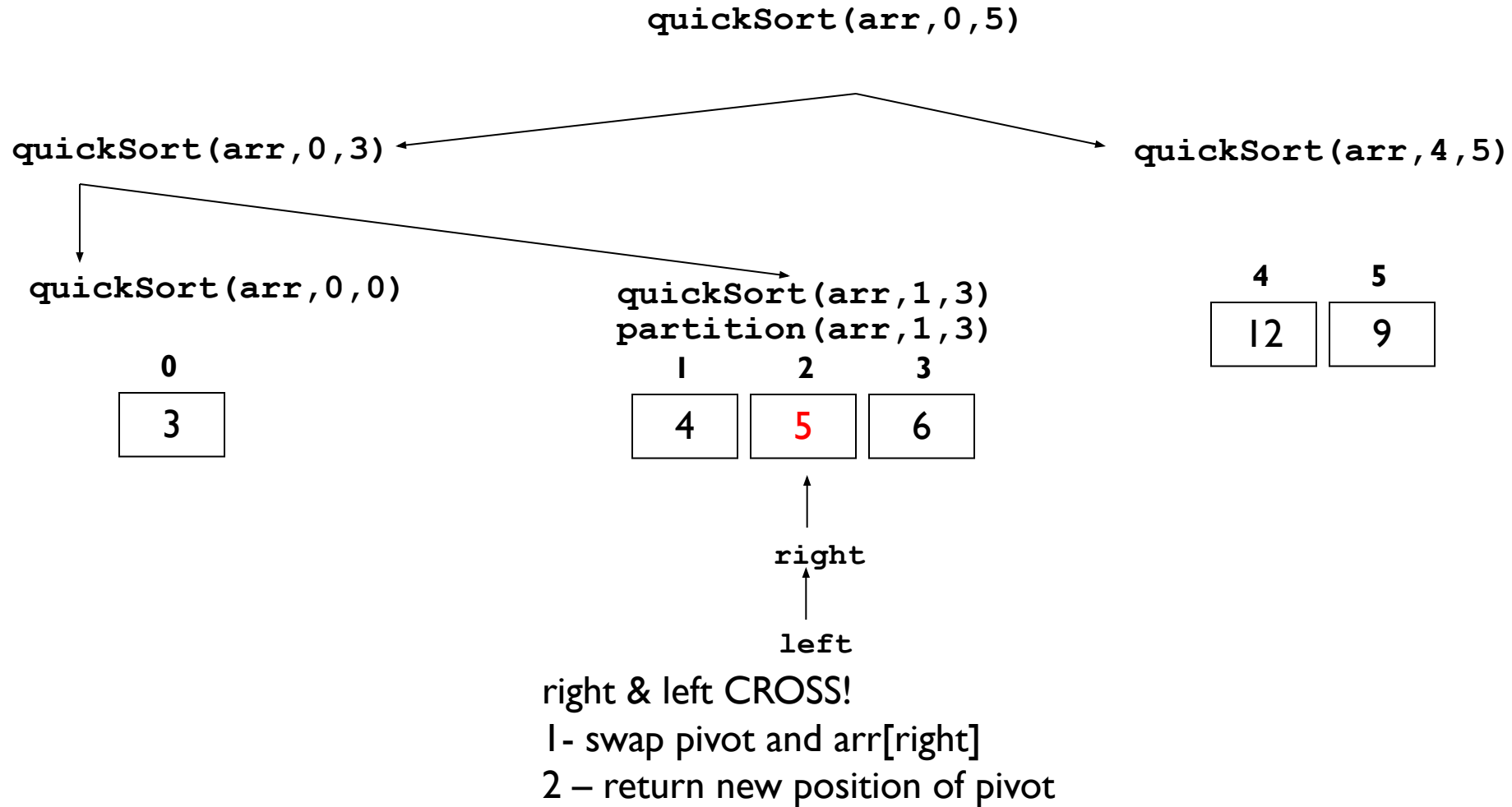
left moves to the right until
value that should be to right
of pivot...



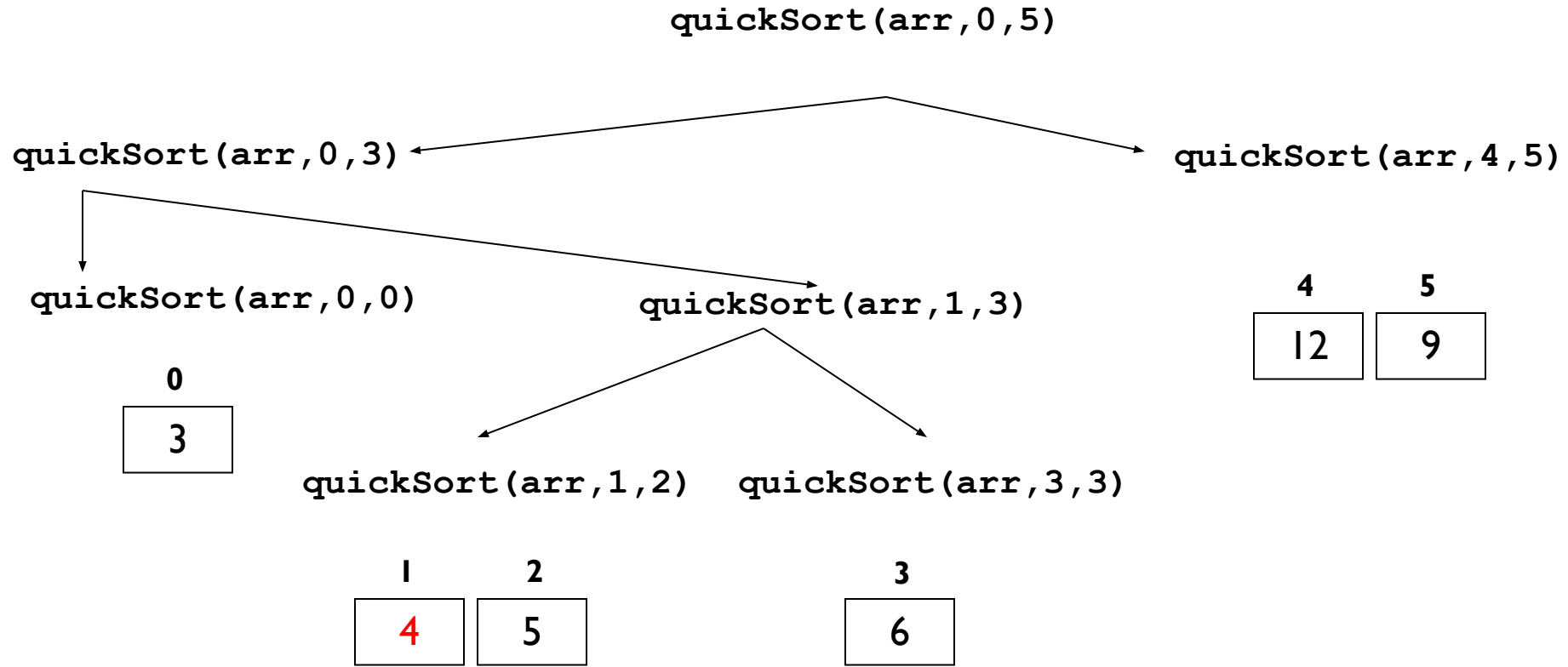


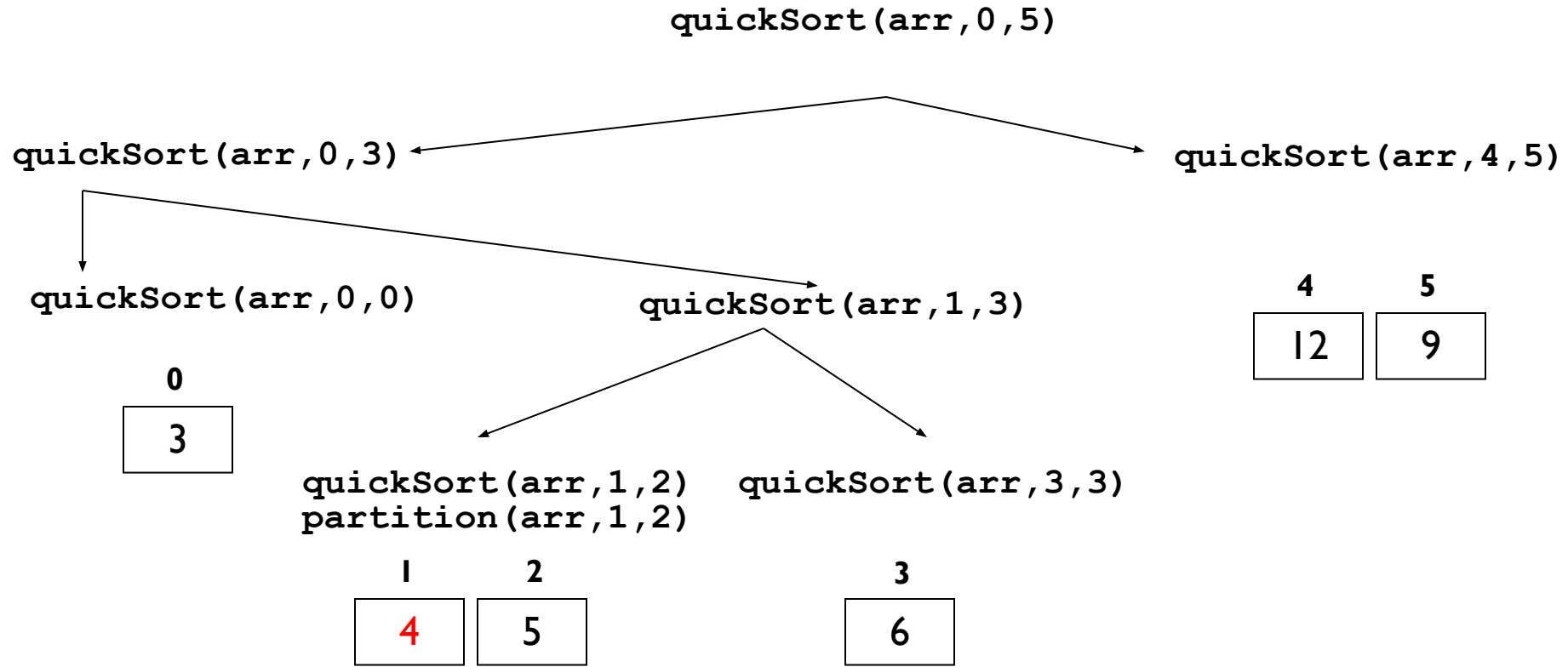


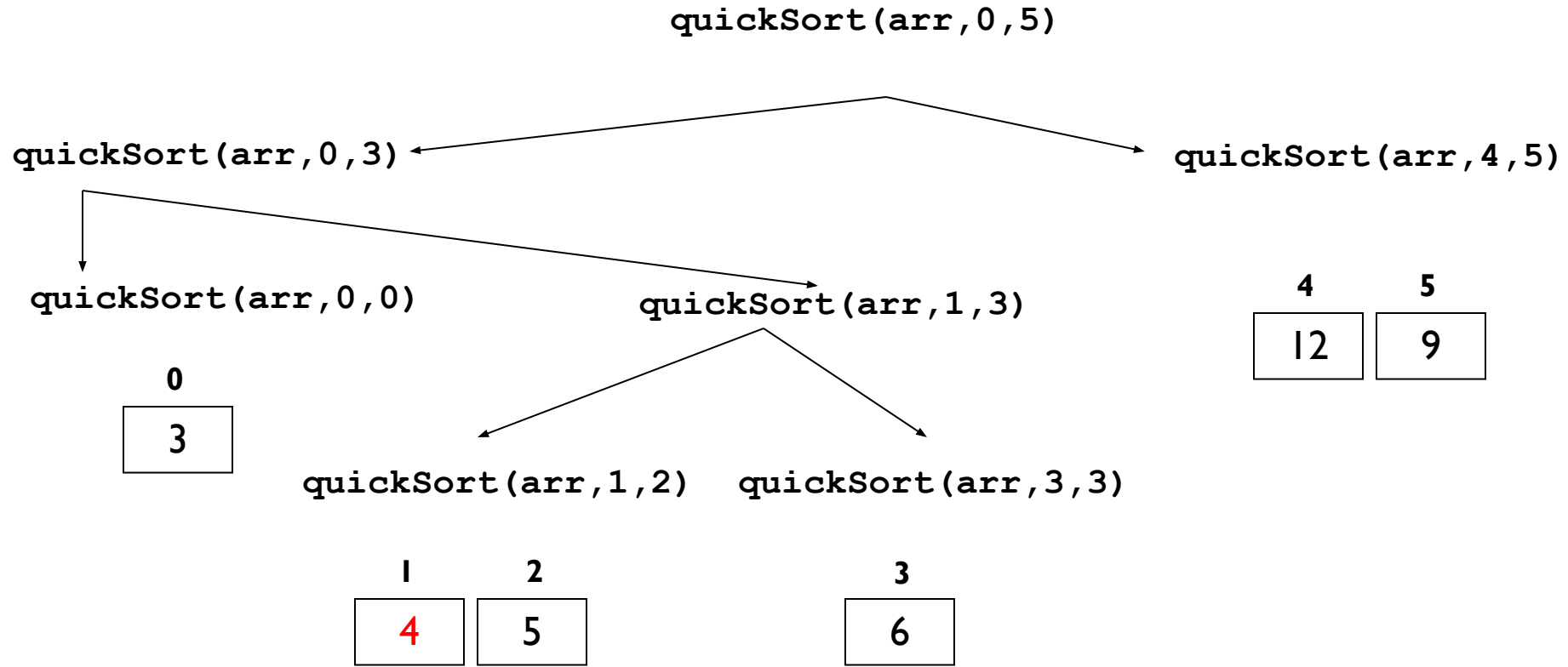


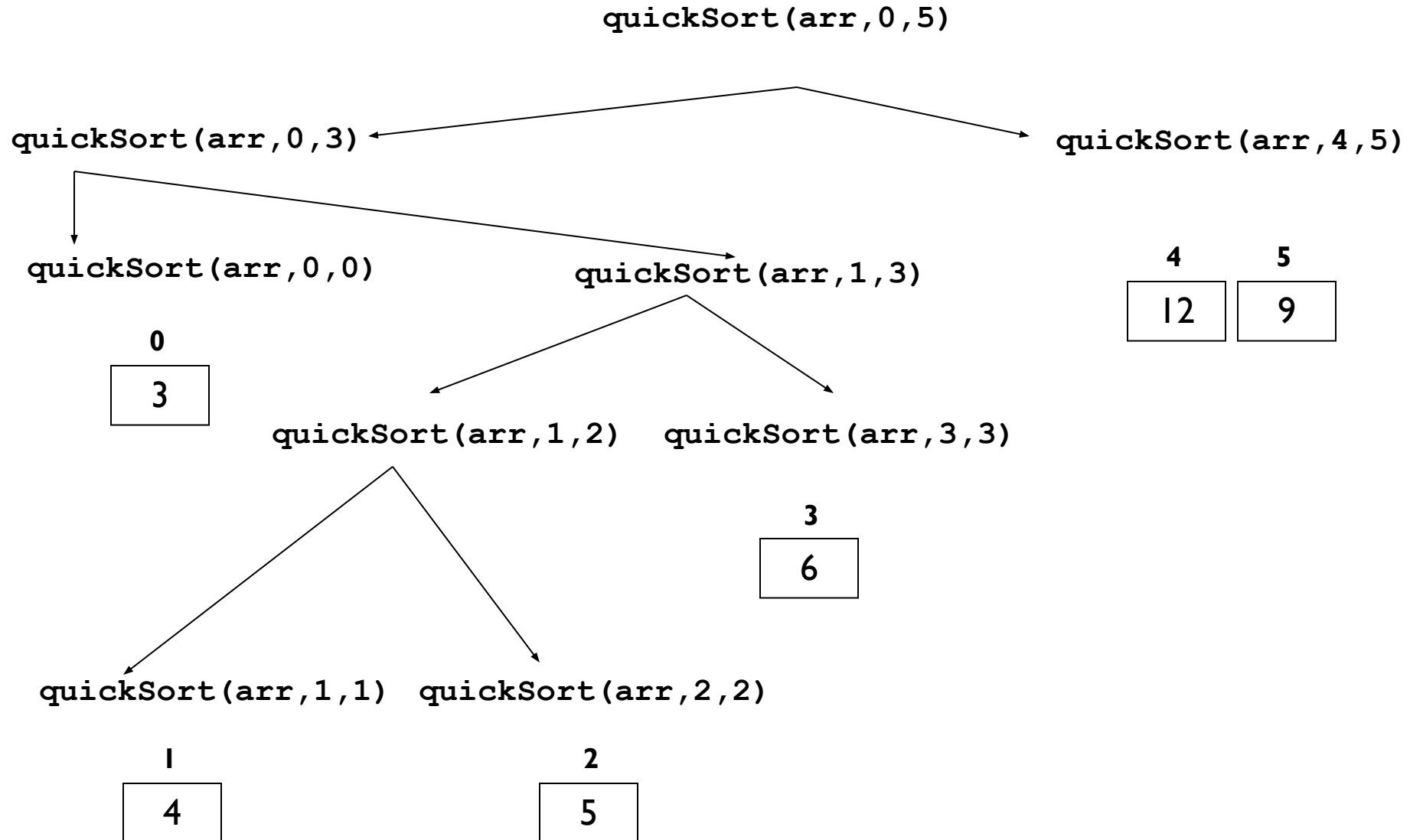


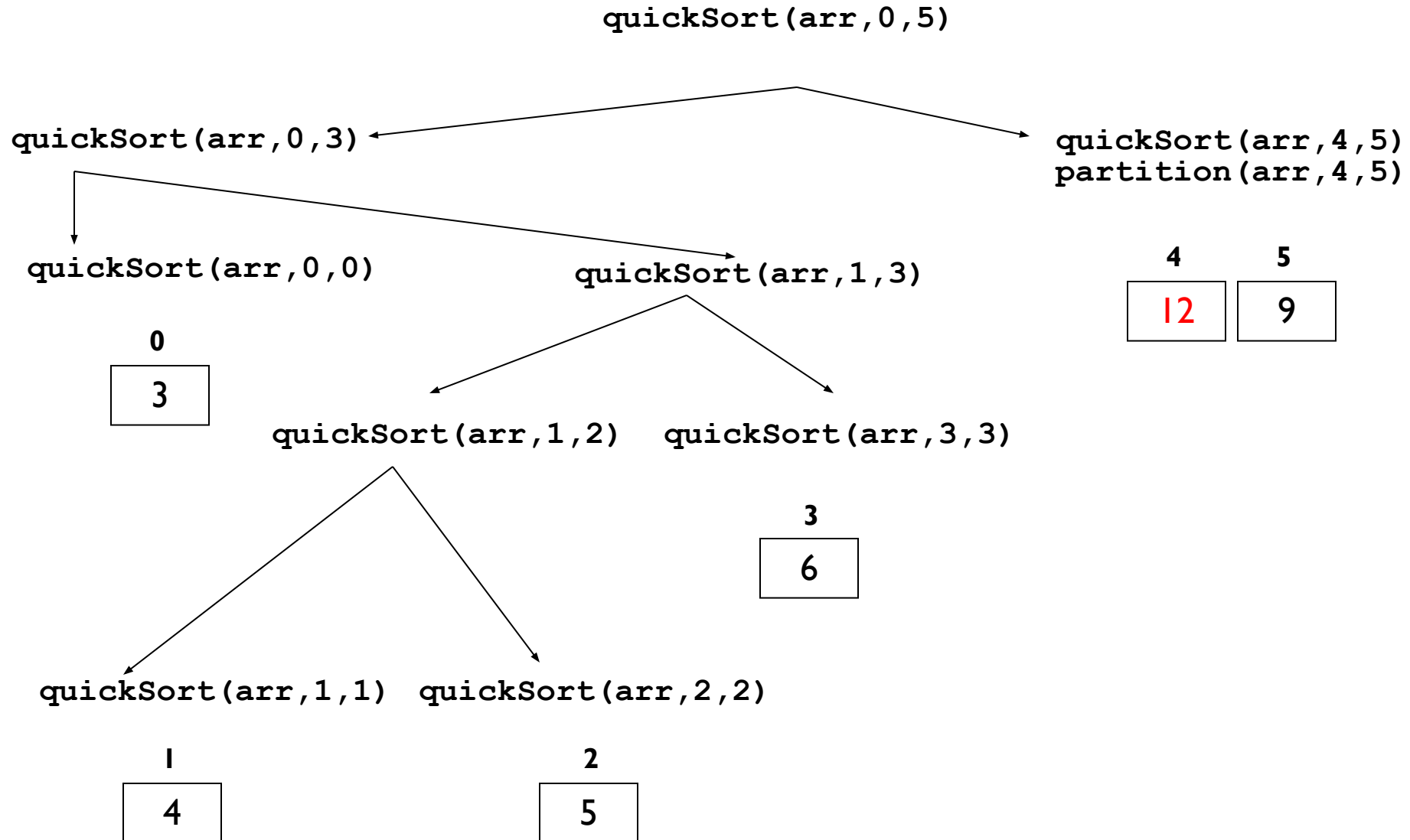
return 2

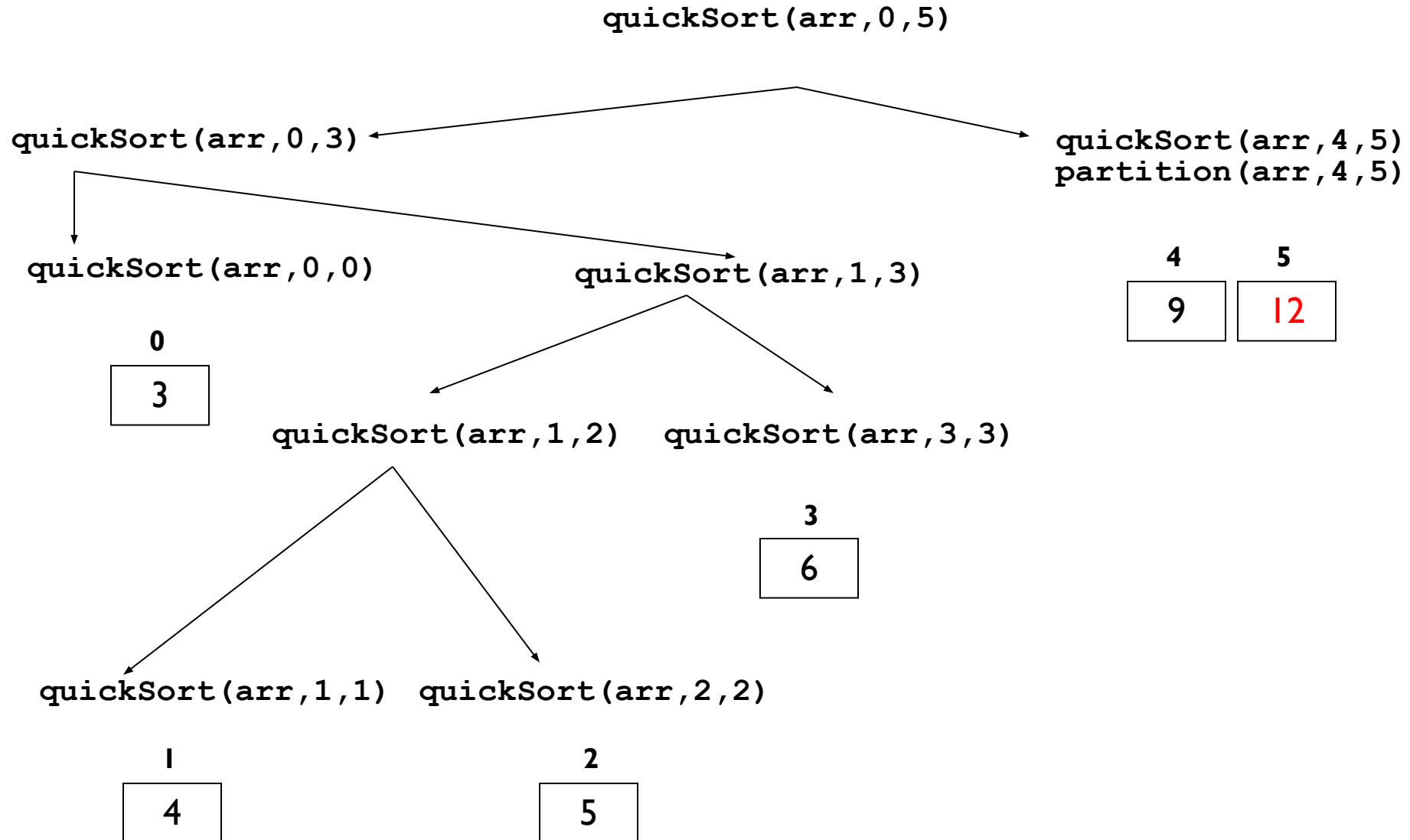


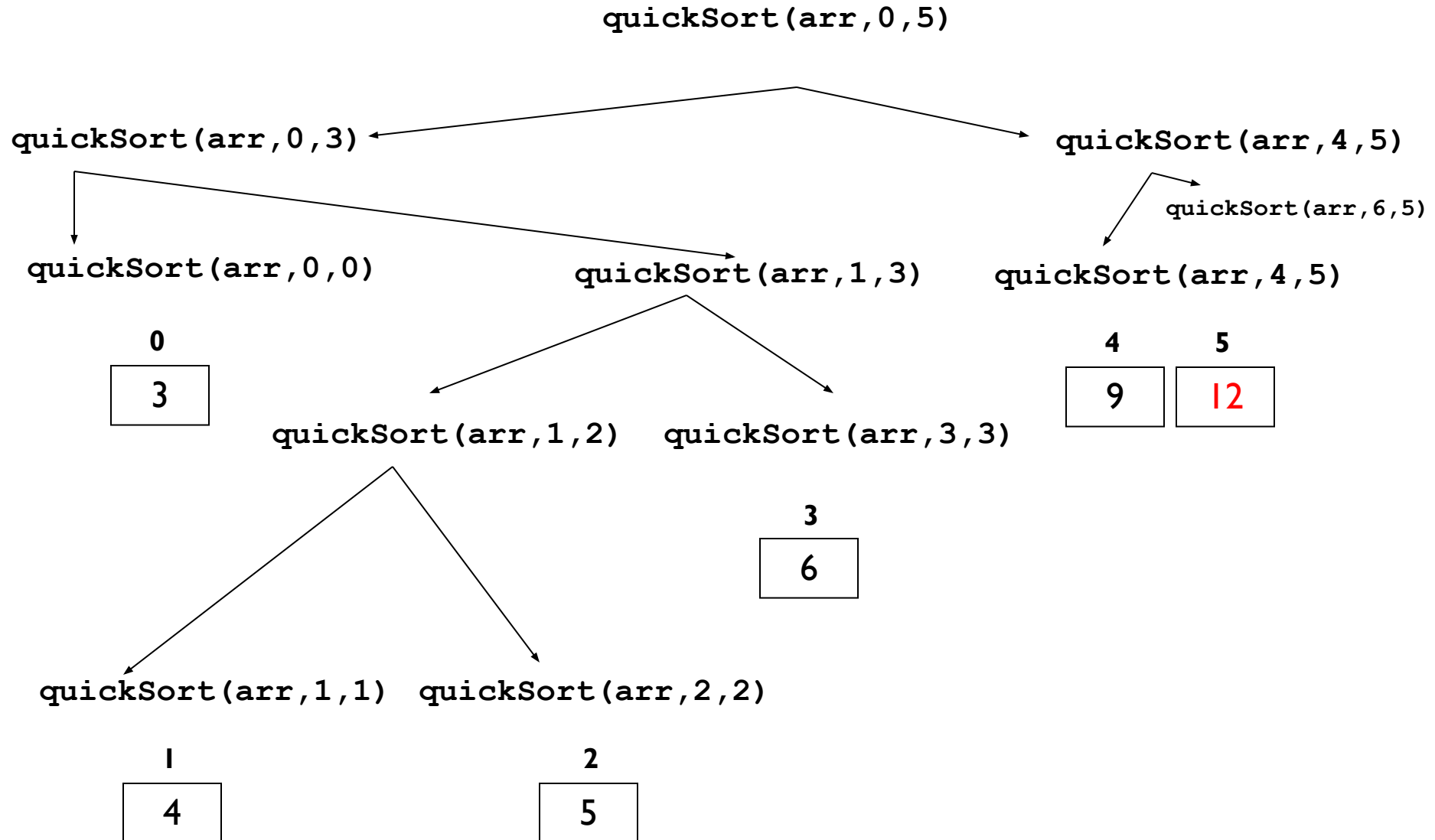


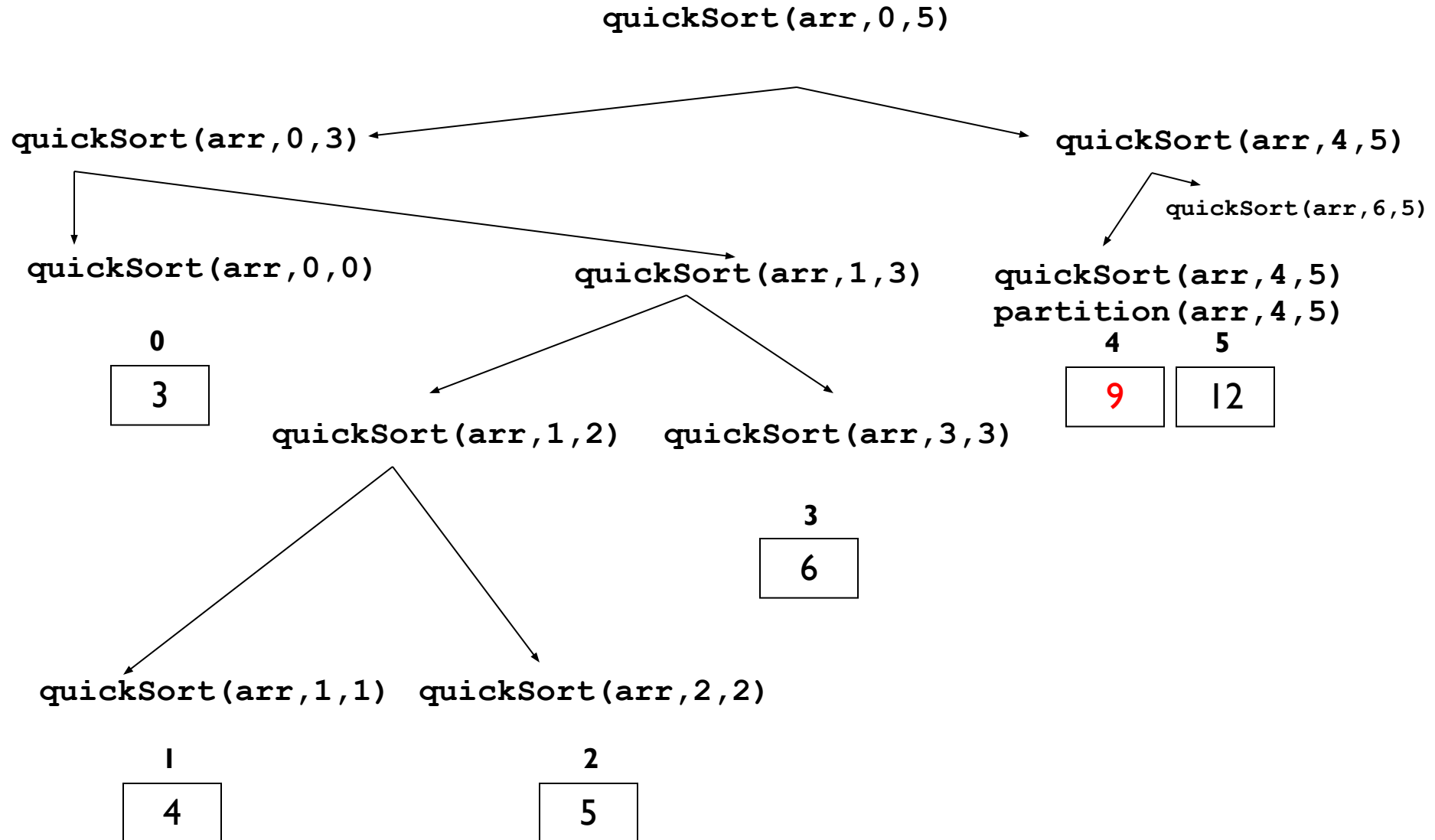


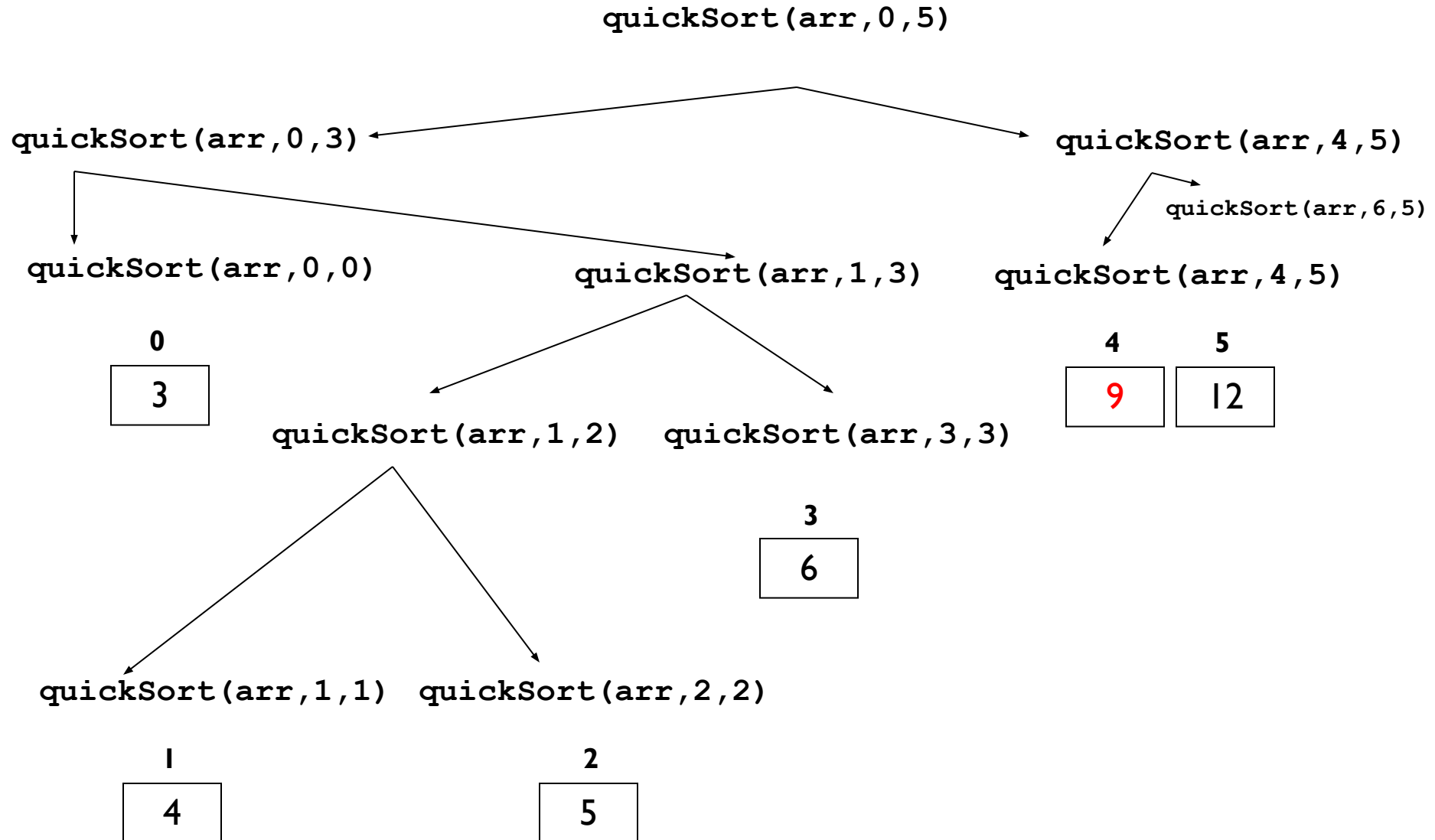


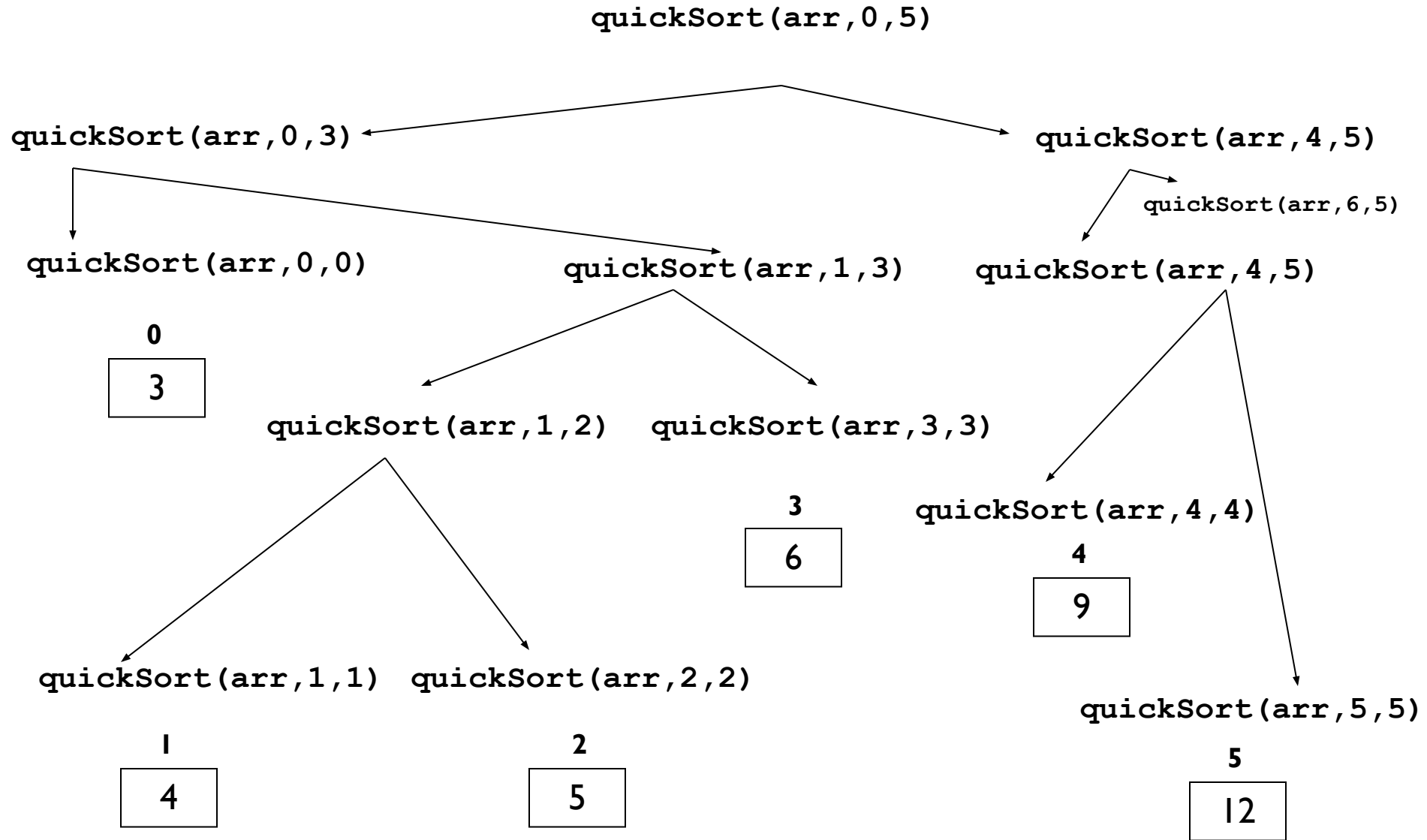












COMMENTS

- Sort smaller subfiles first reduces stack size asymptotically at most $O(\log n)$. Do not stack right subfiles of size < 2 in recursive algorithm -- saves factor of 4.
- Use different pivot selection, e.g. choose pivot to be median of first last and middle.
- **Randomized-Quicksort:** turn bad instances to good instances by picking up the pivot randomly
- Assume that keys are random, uniformly distributed.
- What is best case running time?

QUICKSORT ANALYSIS

- Assume that keys are random, uniformly distributed.
- What is best case running time?
 - Recursion:
 - Partition splits array in two sub-arrays of size $n/2$
 - Quicksort each sub-array
 - Depth of recursion tree? $O(\log_2 n)$
 - Number of accesses per partition? $O(n)$
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 - Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 - Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition? $O(n)$



Sorting

- The goal is to come up with better, more efficient, sorts.
- Because sorting a large number of elements can be extremely time consuming, a good sorting algorithm is very desirable.

Sorting

- How do we describe efficiency ?
- In our study of sorting algorithms , we will relate the number of comparisons to the number of elements in the list (N) as a rough measure of the efficiency of each algorithm.

Sorting

- Input : A sequence of numbers $a_1, a_2, a_3, \dots, a_n$
- Output : A reordering ($a_1, a_2, a_3, \dots, a_n$) of input such that $a_1 < a_2 < a_3 < \dots < a_n$
- Example
 - Input : 8 5 2 3 9 6 1
 - Output : 1 2 3 5 6 8 9

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
 - Recursion:
 1. Partition splits array in two sub-arrays:
 - one sub-array of size 0
 - the other sub-array of size $n-1$
 2. Quicksort each sub-array
 - Depth of recursion tree? $O(n)$
 - Number of accesses per partition? $O(n)$
 - Total Worst Case Complexity? $O(n^2)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

Improved Pivot Selection

Pick median value of three elements from data array:

`data[0]`, `data[n/2]`, and `data[n-1]`.

Use this median value as pivot.

Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
 - Sub-array of size 1: trivial
 - Sub-array of size 2:
 - if($\text{data}[\text{first}] > \text{data}[\text{second}]$) swap them
 - Sub-array of size 3: left as an exercise.