Sorting

Amiya Ranjan Panda

The Sorting Problem

- Input:
  - A sequence of n numbers $<a_1, a_2, . . . , a_n>$
- Output:
  - A reordering $<a_1', a_2', . . . , a_n'>$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$

Why Study Sorting Algorithms?

- There are a variety of situations that are encountered:
  - Do the keys randomly ordered?
  - Are all keys distinct?
  - How large is the set of keys to be ordered?
  - Need guaranteed performance?

- Various algorithms are better suited to some of these situations

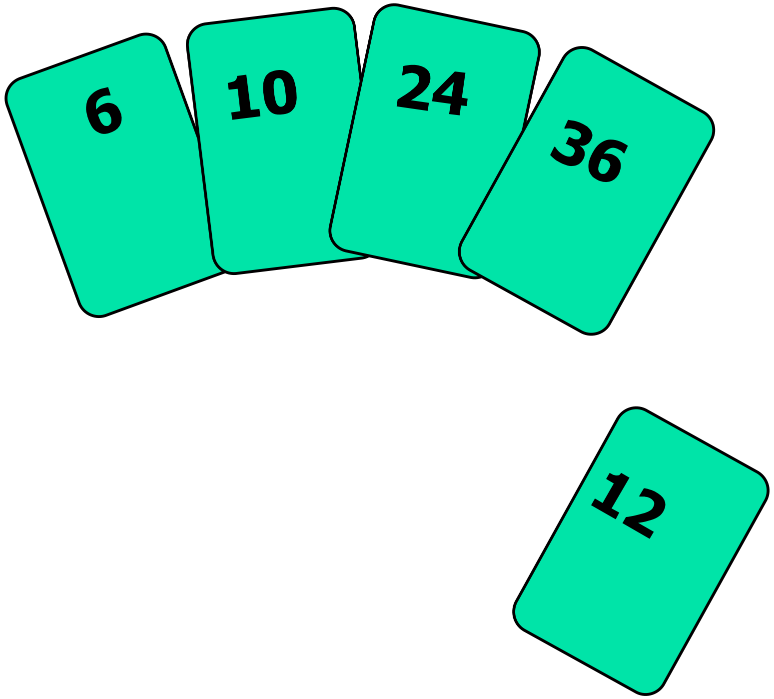
Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower device.

Insertion Sort

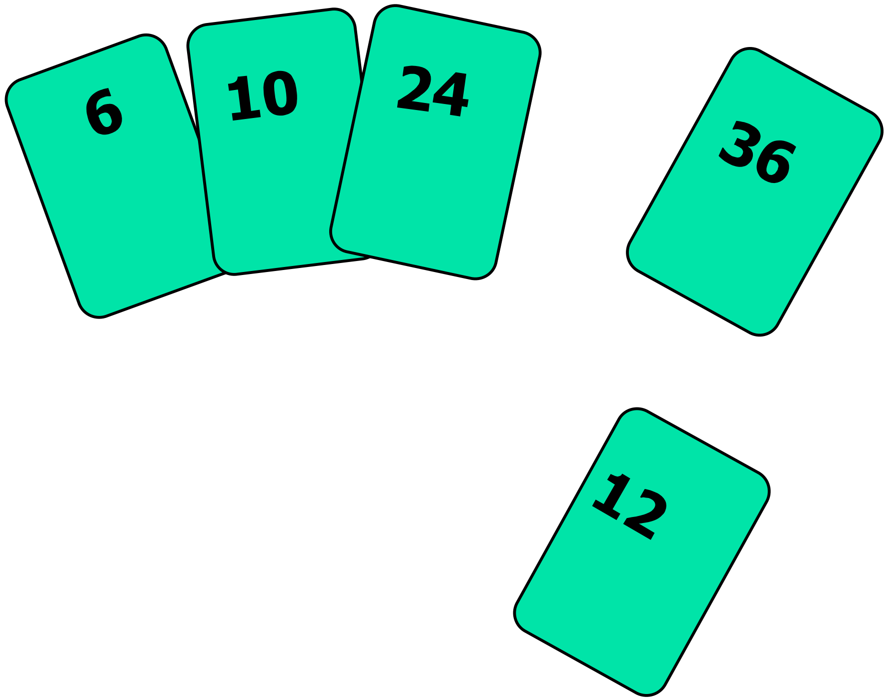
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

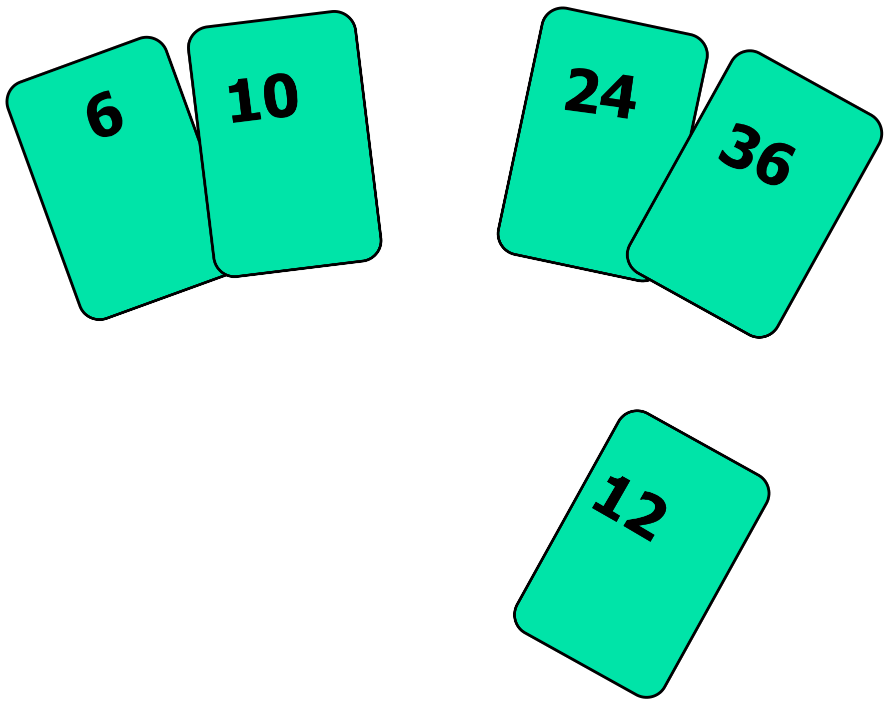


To insert 12, make a room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

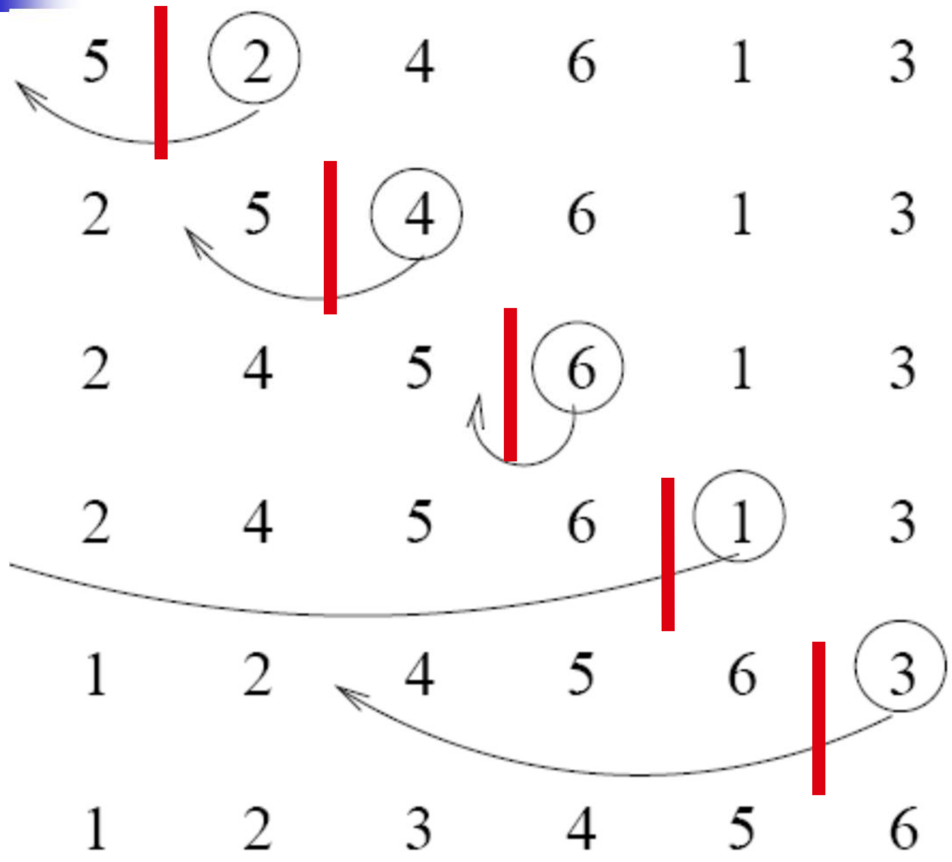
input array

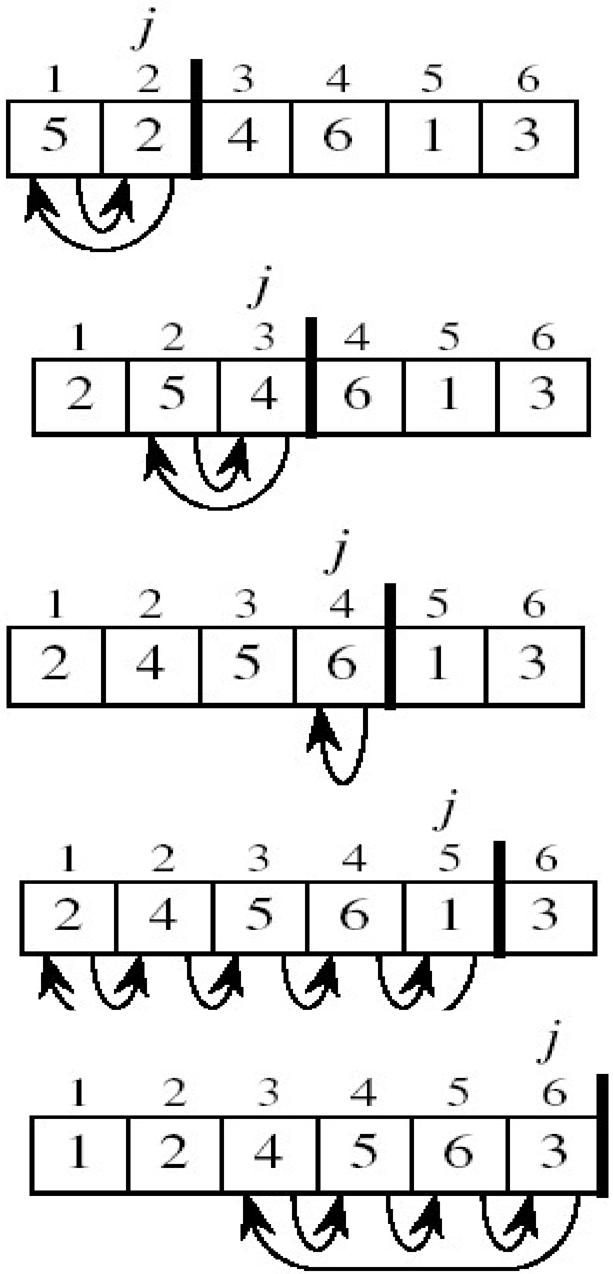
5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

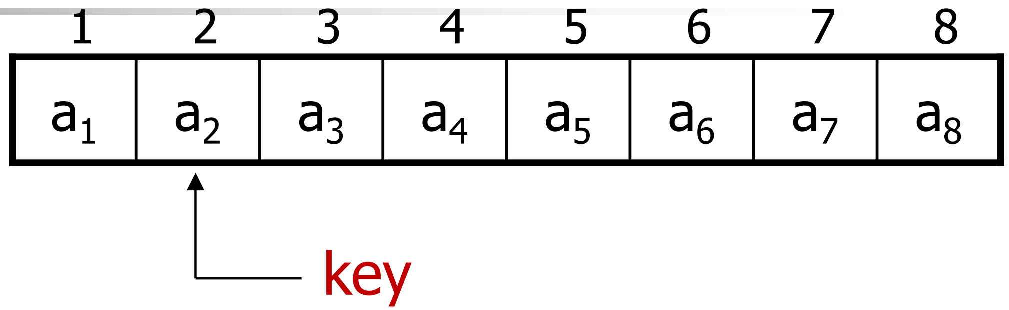
left sub-array	right sub-array
2 5	4 6 1 3
sorted	unsorted

Insertion Sort





Insertion Sort



Alg.: Insertion-Sort*(A)*

for j = 2 **to** n

do key = A[j]

Insert A[j] into the sorted sequence A[

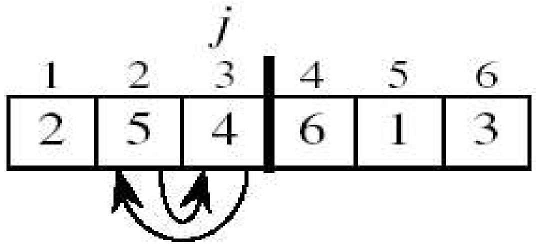
i = j - 1

while i > 0 and A[i] > key

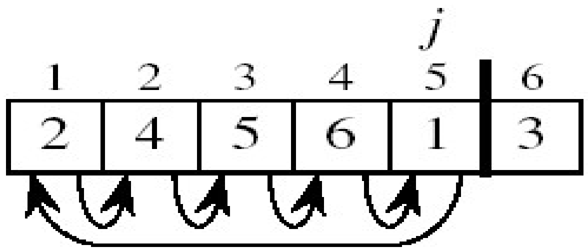
do A[i + 1] = A[i]

i = i – 1

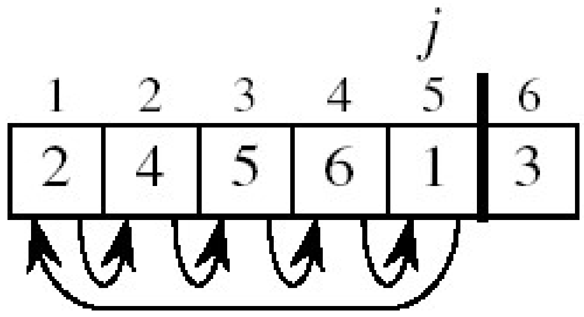
A[i + 1] = key



- Insertion sort – sorts the elements in place



Loop Invariant for Insertion Sort



Alg.: Insertion-Sort(A)

for j = 2 **to** n

do key = A[j]

Insert A[j] into the sorted sequence A[1 . . j -1]

i = j - 1

while i > 0 and A[i] > key

do A[i + 1] = A[i]

i = i – 1

A[i + 1] = key

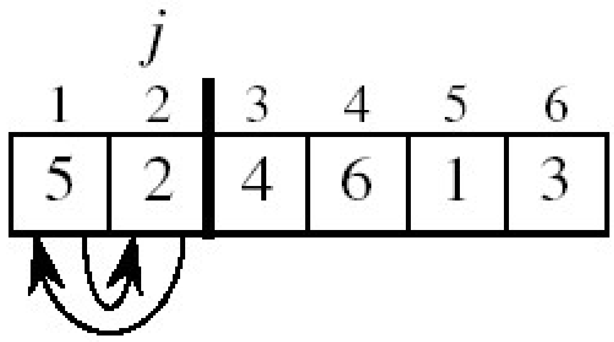
Invariant: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

Proving Loop Invariants

- Proving loop invariants works like induction
- **Initialization (base case):**
  - It is true prior to the first iteration of the loop
- **Maintenance (inductive step):**
  - If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:**
  - When the loop terminates, the invariant gives a useful property that helps show that the algorithm is correct
  - Stop the induction when the loop terminates

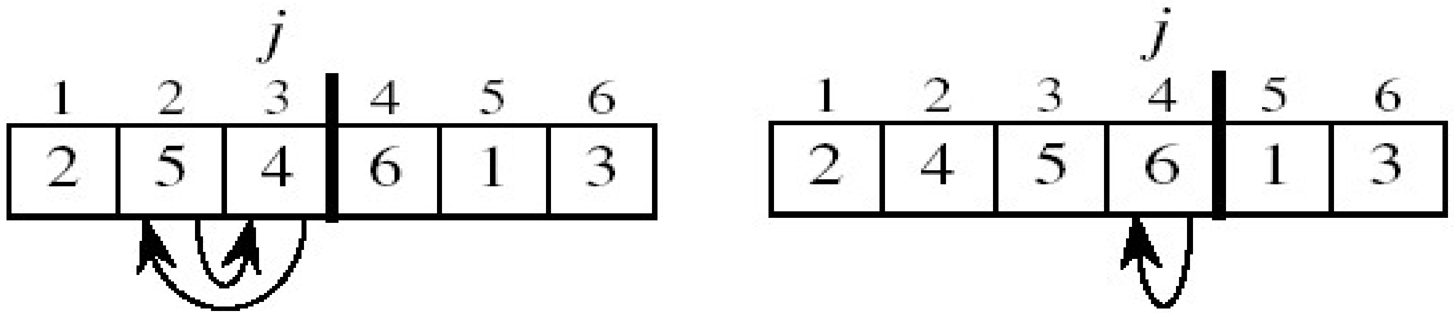
Loop Invariant for Insertion Sort

- **Initialization:**
  - Just before the first iteration, j = 2: the subarray A[1 . . j-1] = A[1], (the element originally in A[1]) – is sorted



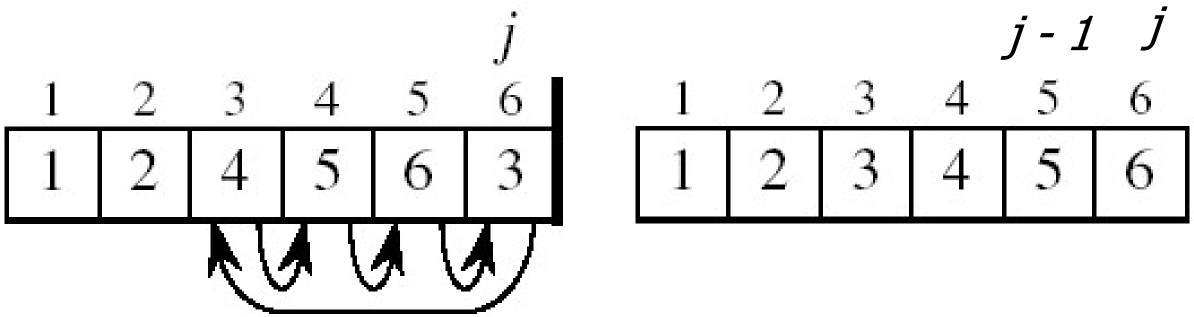
Loop Invariant for Insertion Sort

- **Maintenance:**
  - the **while** inner loop moves A[j -1], A[j -2], A[j -3], and so on, by one position to the right until the proper position for key (which has the value that started out in A[j]) is found
  - At that point, the value of key is placed into this position.



Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when j = n + 1 $\Rightarrow$ j-1 = n
  - Replace n with j-1 in the loop invariant:
    - the subarray A[1 . . n] consists of the elements originally in A[1 . . n], but in sorted order



- The entire array is sorted!

Invariant: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

Time complexity of Insertion Sort

- The worst case time complexity of Insertion sort is $O(n^2)$
  - when the elements are in reverse sorted order
- The average case time complexity of Insertion sort is $O(n^2)$
- The time complexity of the best case is $O(n)$.
  - when the elements are in sorted order
- The space complexity is $O(1)$

Bubble sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until no unsorted elements on the left

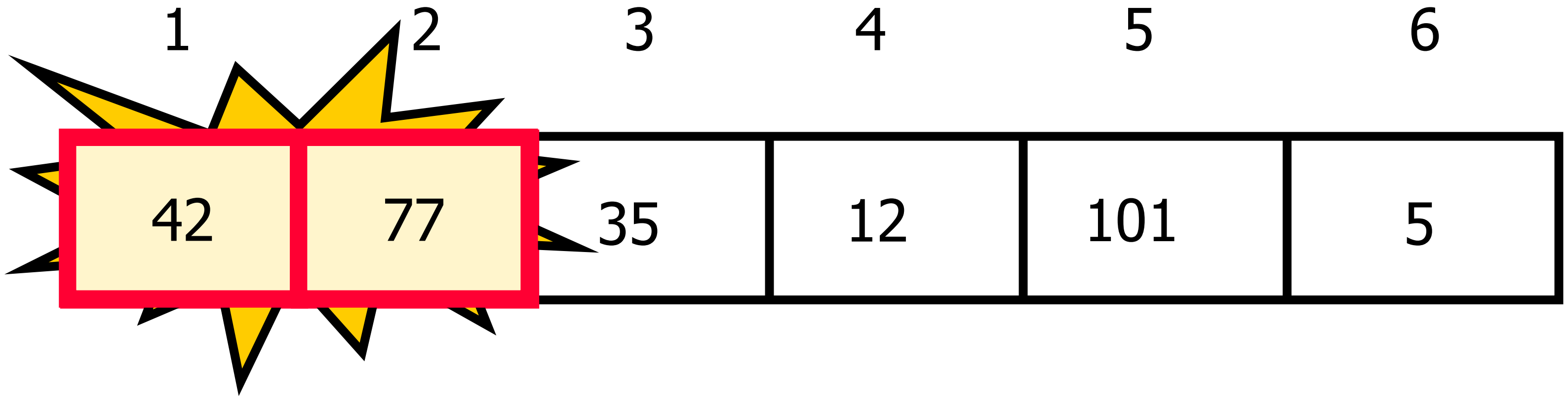
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
77	42	35	12	101	5

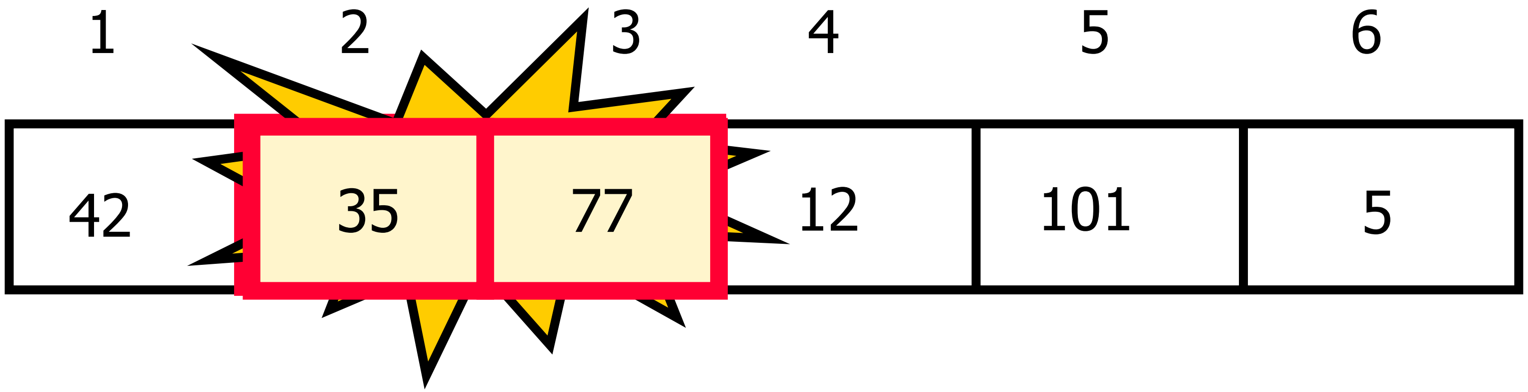
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



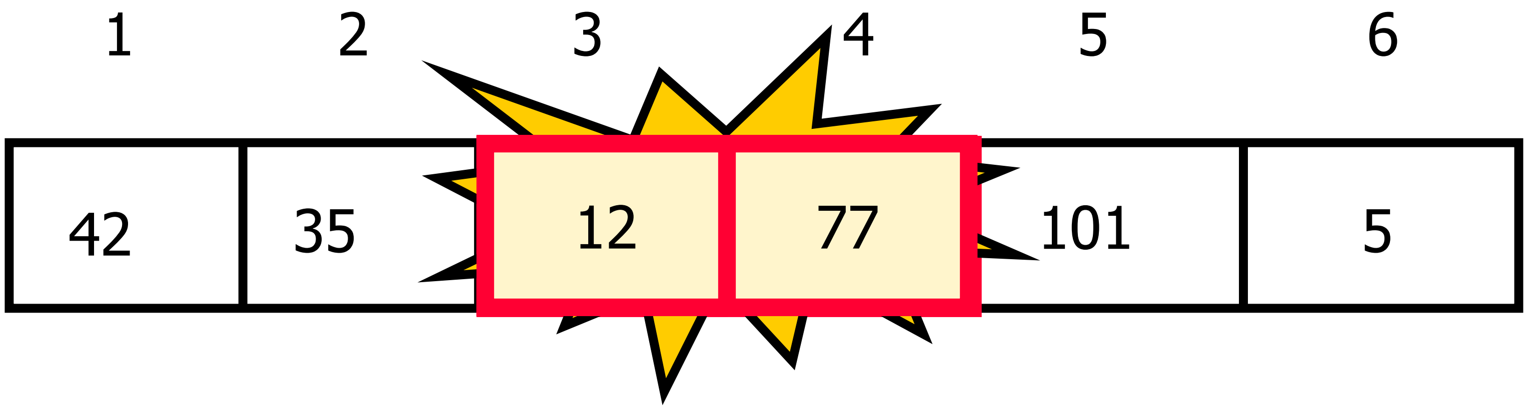
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

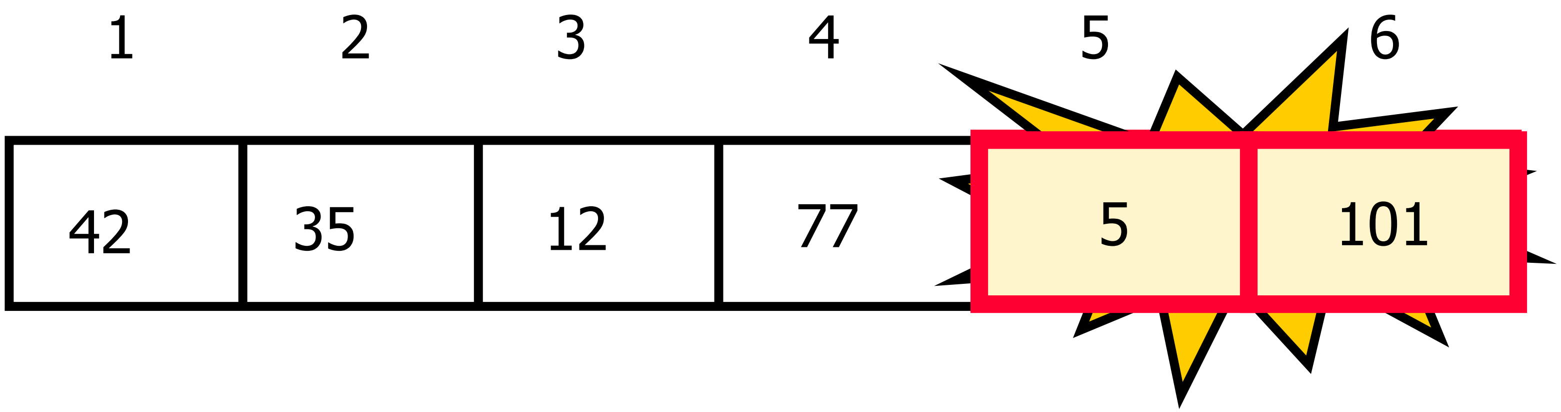
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



Reducing the Number of Comparisons

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

Reducing the Number of Comparisons

- On the n^{th} "bubble up", we only need to do MAX-n comparisons.

- For example:
  - This is the 4^{th} "bubble up"
  - MAX is 6
  - Thus we have 2 comparisons to do

1	2	3	4	5	6
12	35	5	42	77	101

Algorithm

```
BubbleSort(A: Arr_Type)
  iterations, index: integer
  iterations = n – 1

  loop
    exitif(iterations = 0)
    index = 1
    loop
      exitif(index > iterations)
      if(A[index] > A[index + 1]) then
           swap(A[index], A[index + 1])
      endif
      index = index + 1
    endloop
    iterations = iterations - 1
  endloop
endprocedure
```

Inner loop

Outer loop

Already Sorted Collections?

- What if the collection was already sorted?
- What if only a few elements were out of place and after a couple of "bubble ups," the collection was sorted?

- We want to be able to detect this and "stop early"!

1	2	3	4	5	6
5	12	35	42	77	101

Using a Boolean "Flag"

- Use a boolean variable to determine if any swapping occurred during the "bubble up."
- If no swapping occurred, then the collection is already sorted!
- This boolean "flag" needs to be reset after each "bubble up"

Revised Algorithm

```
flag: Boolean
flag = true
iterations = n – 1
loop
    exitif ((iterations = 0) OR NOT(flag))
    index = 1
    flag = false
    loop
        exitif(index > iterations)
        if(A[index] > A[index + 1]) then
            swap(A[index], A[index + 1])
            flag = true
        endif
        index = index + 1
    endloop
    iterations = iterations - 1
endloop
```

An Animated Example

n	8
iterations	7
index	
flag	true

98	23	45	14	6	67	33	42
1	2	3	4	5	6	7	8

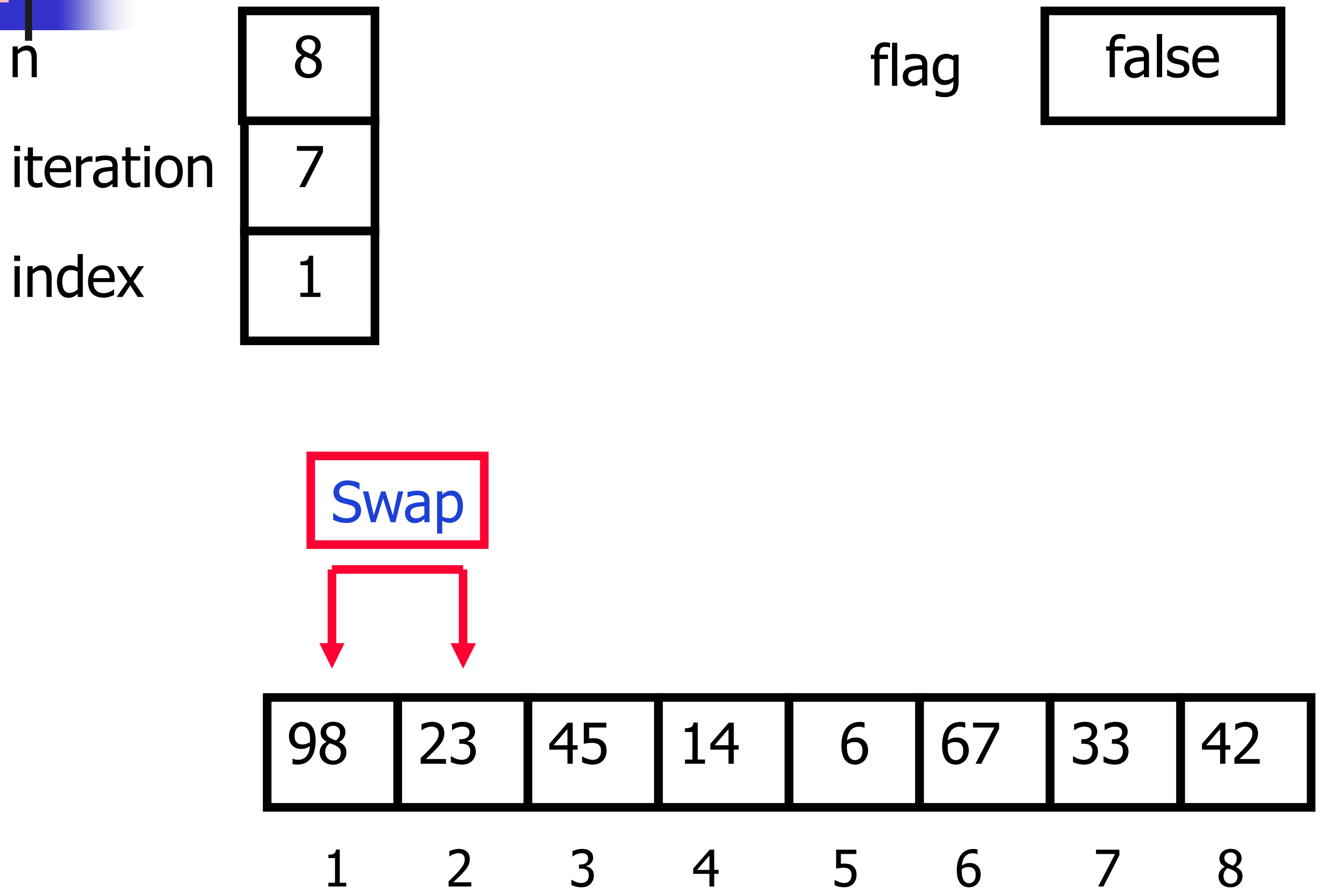
An Animated Example

n	8
iteration	7
index	1

flag false

98	23	45	14	6	67	33	42
1	2	3	4	5	6	7	8

An Animated Example



An Animated Example

- n: 8
- iteration: 7
- index: 1
- flag: true

Swap

23	98	45	14	6	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n 8

iterations 7

index 2

flag true

23	98	45	14	6	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n 8

iterations 7

index 2

flag true

Swap

23	98	45	14	6	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n	8
iterations	7
index	2
flag	true

Swap

1	2	3	4	5	6	7	8
23	45	98	14	6	67	33	42

An Animated Example

n	8
iterations	7
index	3

flag	true

23	45	98	14	6	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n	8
iterations	7
index	3

flag: true

Swap

23	45	98	14	6	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n: 8

iterations: 7

index: 3

flag: true

Swap

1	2	3	4	5	6	7	8
23	45	14	98	6	67	33	42

An Animated Example

n	8
iterations	7
index	4

flag true

1	2	3	4	5	6	7	8
23	45	14	98	6	67	33	42

An Animated Example

n	8
iterations	7
index	4
flag	true

Swap

23	45	14	98	6	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n	8
iterations	7
index	4

flag: true

Swap

23	45	14	6	98	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n 8

iterations 7

index 5

flag true

23	45	14	6	98	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n	8
iterations	7
index	5

flag	true

Swap

23	45	14	6	98	67	33	42
1	2	3	4	5	6	7	8

An Animated Example

n 8

iterations 7

index 5

flag true

Swap

23	45	14	6	67	98	33	42
1	2	3	4	5	6	7	8

An Animated Example

n	8
iterations	7
index	6

flag	true

1	2	3	4	5	6	7	8
23	45	14	6	67	98	33	42

An Animated Example

n 8

iterations 7

index 6

flag true

Swap

23	45	14	6	67	98	33	42
1	2	3	4	5	6	7	8

An Animated Example

n 8

iterations 7

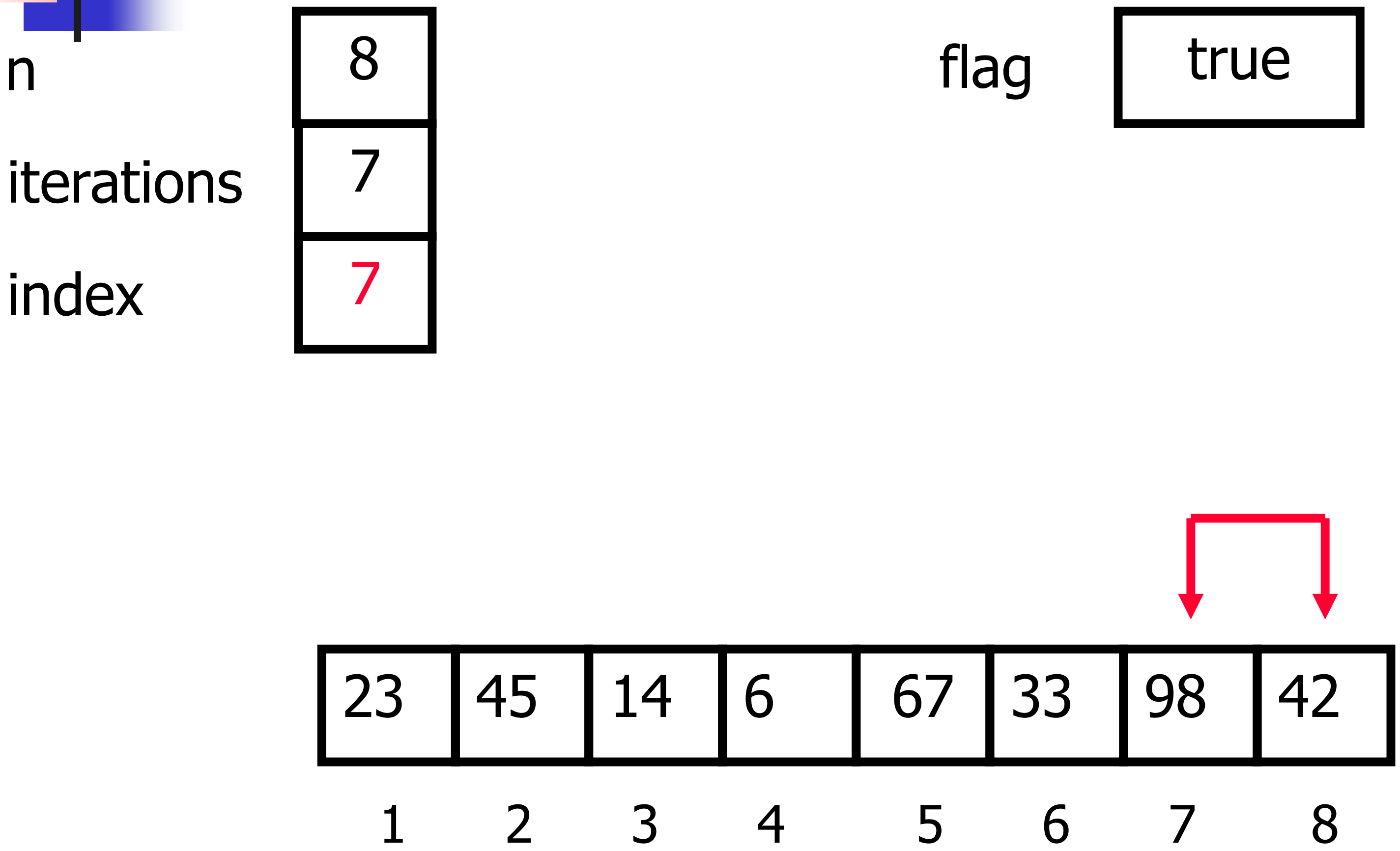
index 6

flag true

Swap

23	45	14	6	67	33	98	42
1	2	3	4	5	6	7	8

An Animated Example



An Animated Example

n	8
iterations	7
index	7
flag	true

Swap

23	45	14	6	67	33	98	42
1	2	3	4	5	6	7	8

An Animated Example

n	8
iterations	7
index	7

flag true

Swap

23	45	14	6	67	33	42	98
1	2	3	4	5	6	7	8

After First Pass of Outer Loop

n	8
iterations	7
index	8

flag: true

Finished first "Bubble Up"

1	2	3	4	5	6	7	8
23	45	14	6	67	33	42	98

The Second "Bubble Up"

n 8

iterations 6

index 1

flag false

23	45	14	6	67	33	42	98
1	2	3	4	5	6	7	8

The Second “Bubble Up”

n 8

iterations 6

index 1

flag false

No Swap

23	45	14	6	67	33	42	98
1	2	3	4	5	6	7	8

The Second “Bubble Up”

n	8
iterations	6
index	2

flag: false

23	45	14	6	67	33	42	98
1	2	3	4	5	6	7	8

The Second “Bubble Up”

n	8
iterations	6
index	2

flag: false

Swap

1	2	3	4	5	6	7	8
23	45	14	6	67	33	42	98

The Second "Bubble Up"

n	8
iterations	6
index	2

flag: true

Swap

23	14	45	6	67	33	42	98
1	2	3	4	5	6	7	8

The Second "Bubble Up"

n	8
iterations	6
index	3

flag: true

23	14	45	6	67	33	42	98
1	2	3	4	5	6	7	8

The Second “Bubble Up”

n 8

iterations 6

index 3

flag true

Swap

1	2	3	4	5	6	7	8
23	14	45	6	67	33	42	98

The Second "Bubble Up"

n	8
iterations	6
index	3

flag: true

Swap

23	14	6	45	67	33	42	98
1	2	3	4	5	6	7	8

The Second "Bubble Up"

n	8
iterations	6
index	4
flag	true

23	14	6	45	67	33	42	98
1	2	3	4	5	6	7	8

The Second “Bubble Up”

n 8

iterations 6

index 4

flag true

No Swap

1	2	3	4	5	6	7	8
23	14	6	45	67	33	42	98

The Second “Bubble Up”

n 8

iterations 6

index 5

flag true

1	2	3	4	5	6	7	8
23	14	6	45	67	33	42	98

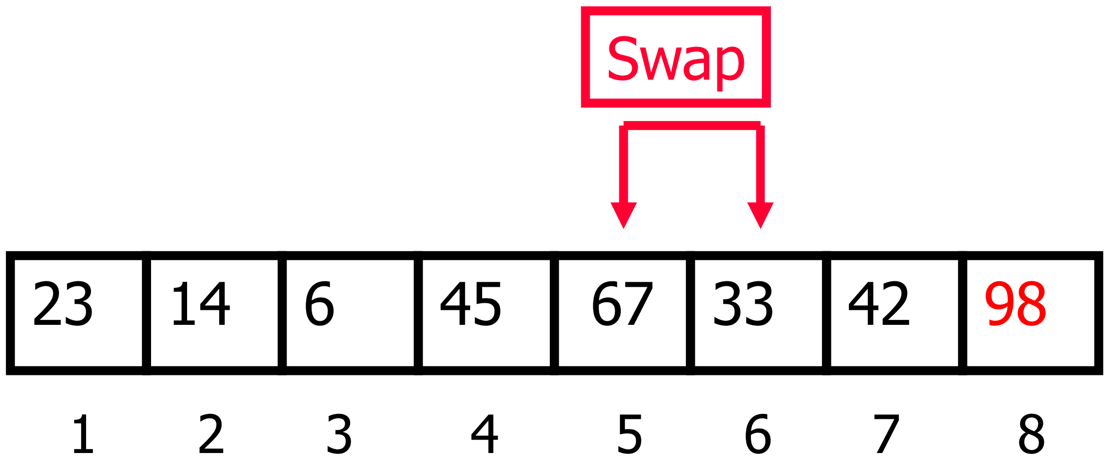
The Second “Bubble Up”

n 8

iterations 6

index 5

flag true



The Second “Bubble Up”

n 8

iterations 6

index 5

flag true

Swap

23	14	6	45	33	67	42	98
1	2	3	4	5	6	7	8

The Second "Bubble Up"

n 8

iterations 6

index 6

flag true

23	14	6	45	33	67	42	98
1	2	3	4	5	6	7	8

The Second “Bubble Up”

n 8

iterations 6

index 6

flag true

Swap

23	14	6	45	33	67	42	98
1	2	3	4	5	6	7	8

The Second "Bubble Up"

n 8

iterations 6

index 6

flag true

Swap

23	14	6	45	33	42	67	98
1	2	3	4	5	6	7	8

After Second Pass of Outer Loop

n	8
iterations	6
index	7

flag: true

Finished second "Bubble Up"

1	2	3	4	5	6	7	8
23	14	6	45	33	42	67	98

The Third "Bubble Up"

n	8
iterations	5
index	1

flag false

23	14	6	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third “Bubble Up”

n	8
iterations	5
index	1

flag: false

Swap

23	14	6	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n 8

iterations 5

index 1

flag true

Swap

14	23	6	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n	8
iterations	5
index	2

flag: true

14	23	6	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n 8

iterations 5

index 2

flag true

Swap

14	23	6	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n 8

iterations 5

index 2

flag true

Swap

14	6	23	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n	8
iterations	5
index	3

flag: true

14	6	23	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third “Bubble Up”

n	8
iterations	5
index	3

flag: true

No Swap

14	6	23	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third “Bubble Up”

n 8

iterations 5

index 4

flag true

14	6	23	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third “Bubble Up”

n	8
iterations	5
index	4

flag: true

Swap

14	6	23	45	33	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n	8
iterations	5
index	4

flag: true

Swap

14	6	23	33	45	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n	8
iterations	5
index	5
flag	true

1	2	3	4	5	6	7	8
14	6	23	33	45	42	67	98

The Third "Bubble Up"

n 8

iterations 5

index 5

flag true

Swap

14	6	23	33	45	42	67	98
1	2	3	4	5	6	7	8

The Third "Bubble Up"

n	8
iterations	5
index	5

flag: true

Swap

1	2	3	4	5	6	7	8
14	6	23	33	42	45	67	98

After Third Pass of Outer Loop

n 8

iterations 5

index 6

flag true

Finished third “Bubble Up”

14	6	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth “Bubble Up”

n 8

iterations 4

index 1

flag false

14	6	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth "Bubble Up"

n 8

iterations 4

index 1

flag false

Swap

14	6	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth "Bubble Up"

n 8

iterations 4

index 1

flag true

Swap

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth "Bubble Up"

n	8
iterations	4
index	2

flag: true

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth "Bubble Up"

n	8
iterations	4
index	2

flag: true

No Swap

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth “Bubble Up”

n	8
iterations	4
index	3

flag: true

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth "Bubble Up"

n 8

iterations 4

index 3

flag true

No Swap

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth “Bubble Up”

n	8
iterations	4
index	4

flag: true

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fourth "Bubble Up"

n	8
iterations	4
index	4

flag: true

No Swap

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

After Fourth Pass of Outer Loop

n	8
iterations	4
index	5

flag	true

Finished fourth "Bubble Up"

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fifth "Bubble Up"

n	8
iterations	3
index	1
flag	false

1	2	3	4	5	6	7	8
6	14	23	33	42	45	67	98

The Fifth "Bubble Up"

n	8
iterations	3
index	1

flag: false

No Swap

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fifth "Bubble Up"

n	8
iterations	3
index	2

flag false

1	2	3	4	5	6	7	8
6	14	23	33	42	45	67	98

The Fifth “Bubble Up”

n	8
iterations	3
index	2

flag: false

No Swap

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fifth “Bubble Up”

n 8

iterations 3

index 3

flag false

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

The Fifth "Bubble Up"

n 8

iterations 3

index 3

flag false

No Swap

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

After Fifth Pass of Outer Loop

n 8

iterations 3

index 4

flag false

Finished fifth "Bubble Up"

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

Finished "Early"

n	8
iterations	3
index	4

flag false

We didn't do any swapping, so all of the other elements must be correctly placed.

We can "skip" the last two passes of the outer loop.

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

Bubble Sort

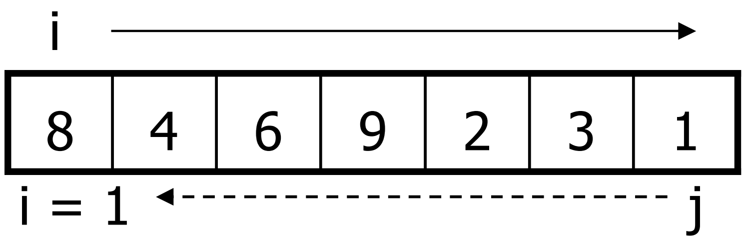
Alg.: Bubble-Sort(A)

for i ← 1 **to** length[A]

  do for j ← length[A] **downto** i + 1

    do if A[j] < A[j -1]

      then exchange A[j] ↔ A[j-1]



Bubble Sort: Analysis

- The biggest items "bubble up" to the end of the array, as the algorithm progresses.
- Efficiency:
  - if n is number of items,
  - n-1 comparisons in first pass, n-2 in second pass, so on and so forth.
  - The formula is : $(n-1) + (n-2) + \ldots + 1 = n \times (n-1)/2$.
  - hence runs in $O(n^2)$ time.

Time complexity of Bubble Sort

- The worst case time complexity of Bubble sort is $O(n^2)$
  - when the elements are in reverse sorted order
- The average case time complexity of Bubble sort is $O(n^2)$
- The time complexity of the best case is $O(n)$.
  - when thr elements are in sorted order
- The space complexity is $O(1)$

Selection Sort

- Idea:
  - Find the smallest/ largest element in the array
  - Exchange it with the element in the first/ last position
  - Find the second smallest/ largest element and exchange it with the element in the second position
  - Continue until the array is sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

- Comparison
- Data Movement
- Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

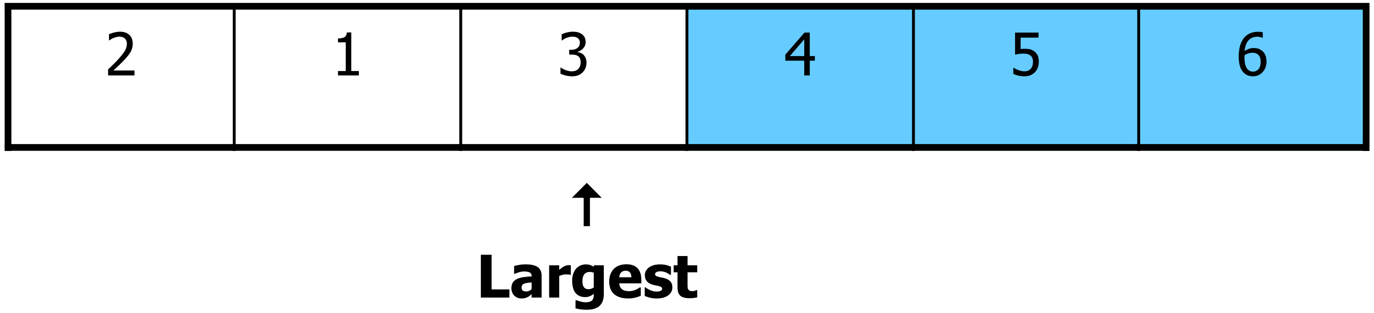
2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

2	1	3	4	5	6

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

1	2	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

1	2	3	4	5	6

DONE!

Comparison

Data Movement

Sorted

Selection Sort

```
Alg.: Selection-Sort(A)
   n ← length[A]
   for j ← 1 to n-1
      do smallest ← j
         for i ← j + 1 to n
            do if A[i] < A[smallest]
                  then smallest ← i
         exchange A[j] ↔ A[smallest]
```

```
Alg.: Selection-Sort(A)
   n ← length[A]
   for j ← n downto 2
      do largest ← 1
         for i ← 2 to j
            do if A[i] > A[largest]
                  then largest ← i
         exchange A[j] ↔ A[largest]
```

Time complexity of Selection Sort

- The worst case time complexity of Bubble sort is $O(n^2)$
  - when the elements are in reverse sorted order
- The average case time complexity of Bubble sort is $O(n^2)$
- The time complexity of the best case is $O(n^2)$.
  - when thr elements are in sorted order
- The space complexity is $O(1)$

The number of swaps in Selection Sort are as follows:

- Worst case: $O(n)$
- Average Case: $O(n)$
- Best Case: $O(1)$

Divide and Conquer

- Recursive in structure
  - Divide the problem into sub-problems that are similar to the original but smaller in size
  - Conquer the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - Combine the solutions to create a solution to the original problem

An Example: Merge Sort

Sorting Problem: Sort a sequence of n elements into non-decreasing order.

- Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

Mergesort

8	2	9	4	5	3	1	6

- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Merging

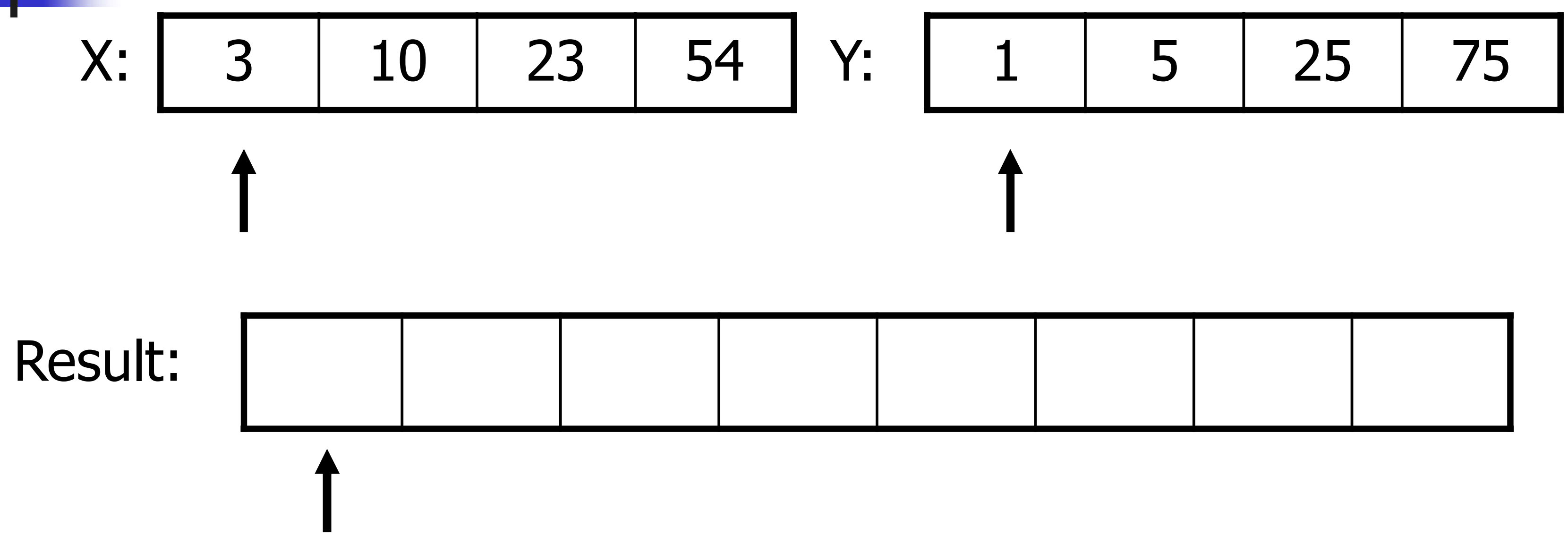
- The key point to Merge Sort is merging two sorted lists into one.
- Suppose, there are two lists X ($x_1 \le x_2 \le \cdots \le x_m$) and Y($y_1 \le y_2 \le \cdots \le y_n$) the resulting list is Z($z_1 \le z_2 \le \cdots \le z_{m+n}$)

- Example:

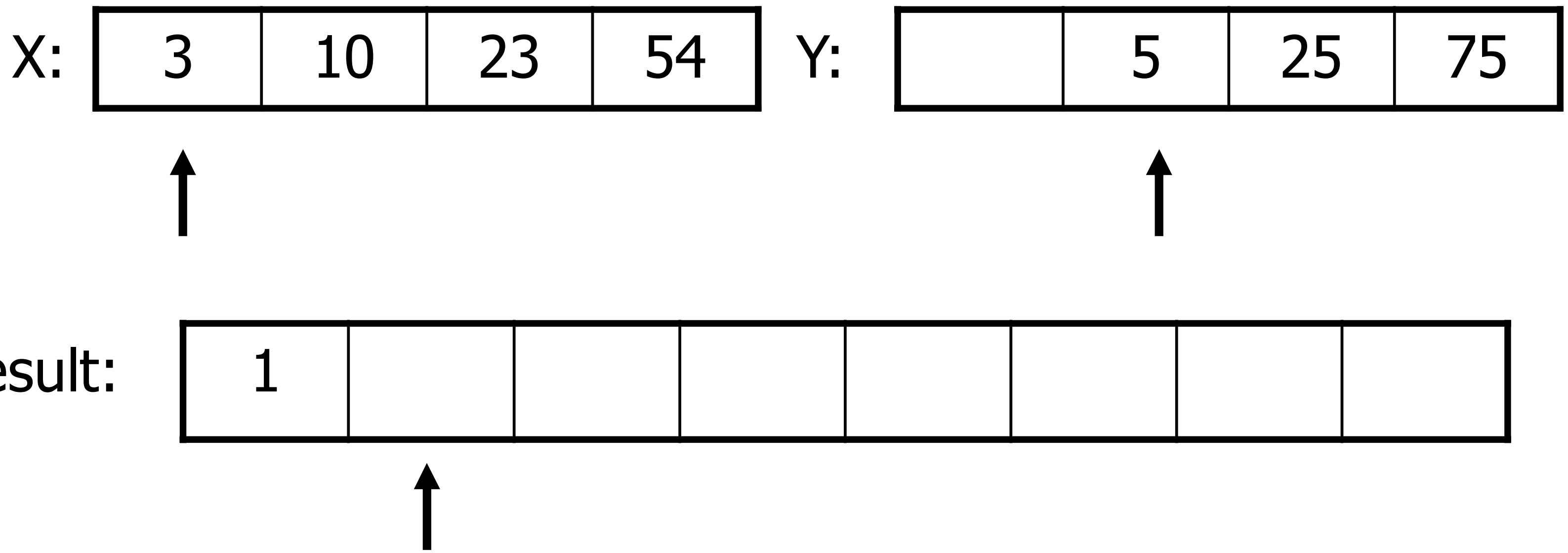
  $L_1 = \{3, 8, 9\}$ $L_2 = \{1, 5, 7\}$

  merge(L_1, L_2) = {1, 3, 5, 7, 8, 9}

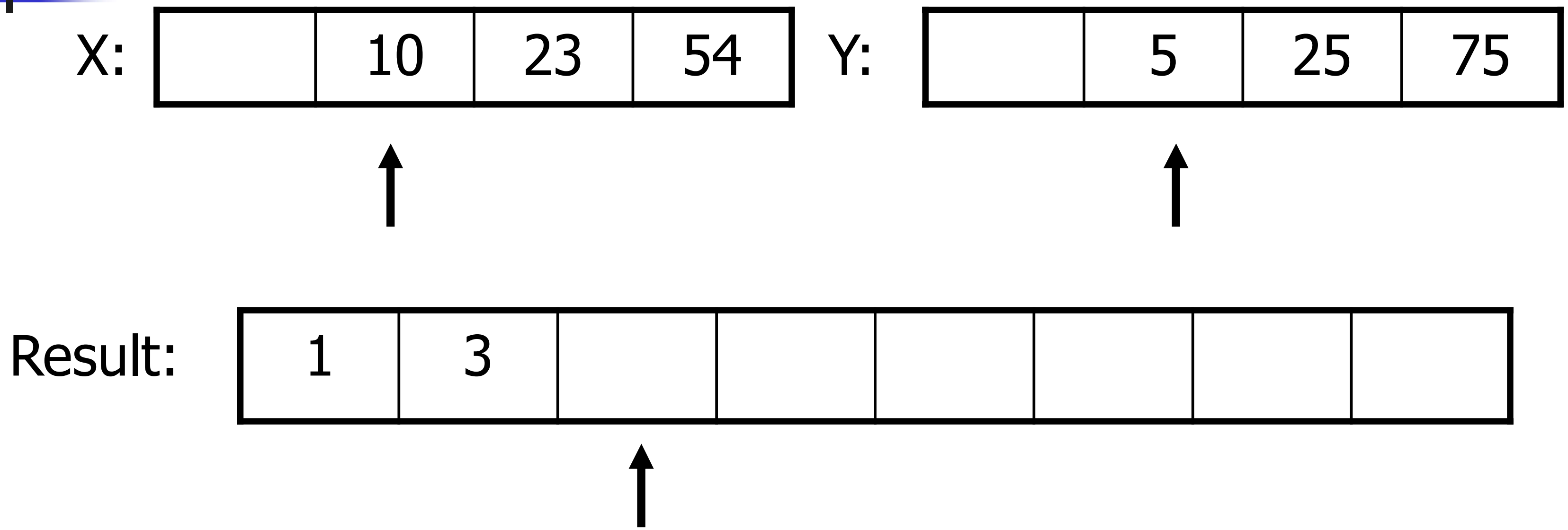
Merging (cont.)



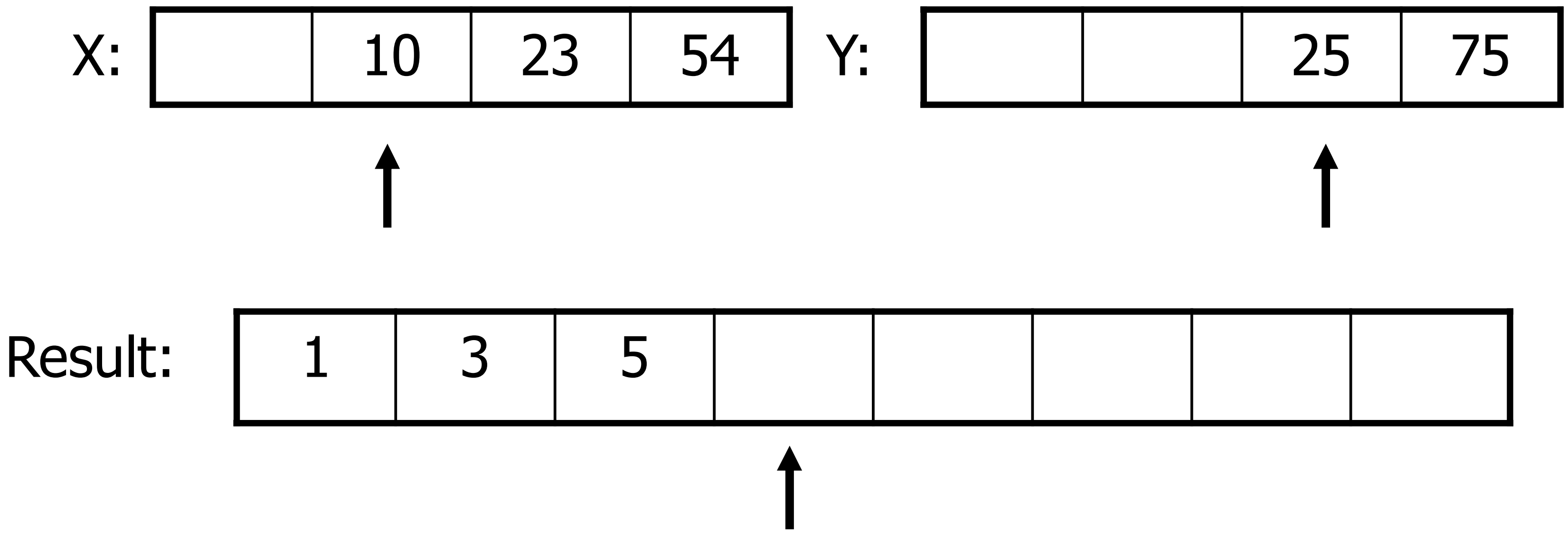
Merging (cont.)



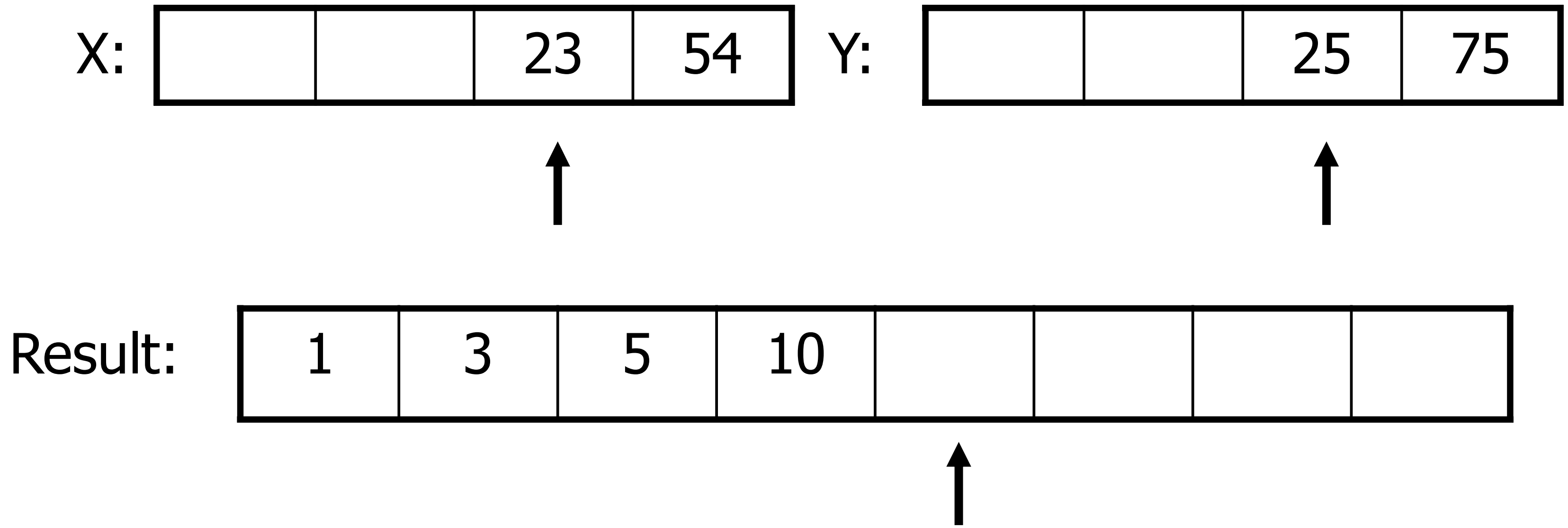
Merging (cont.)



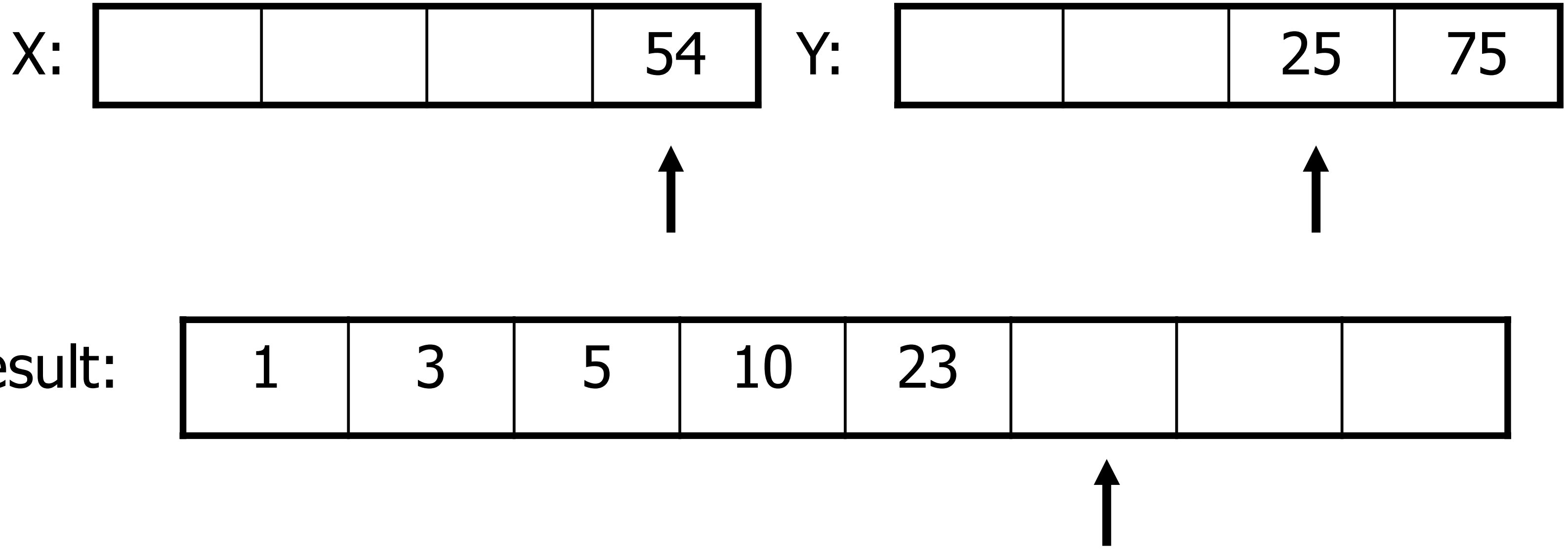
Merging (cont.)



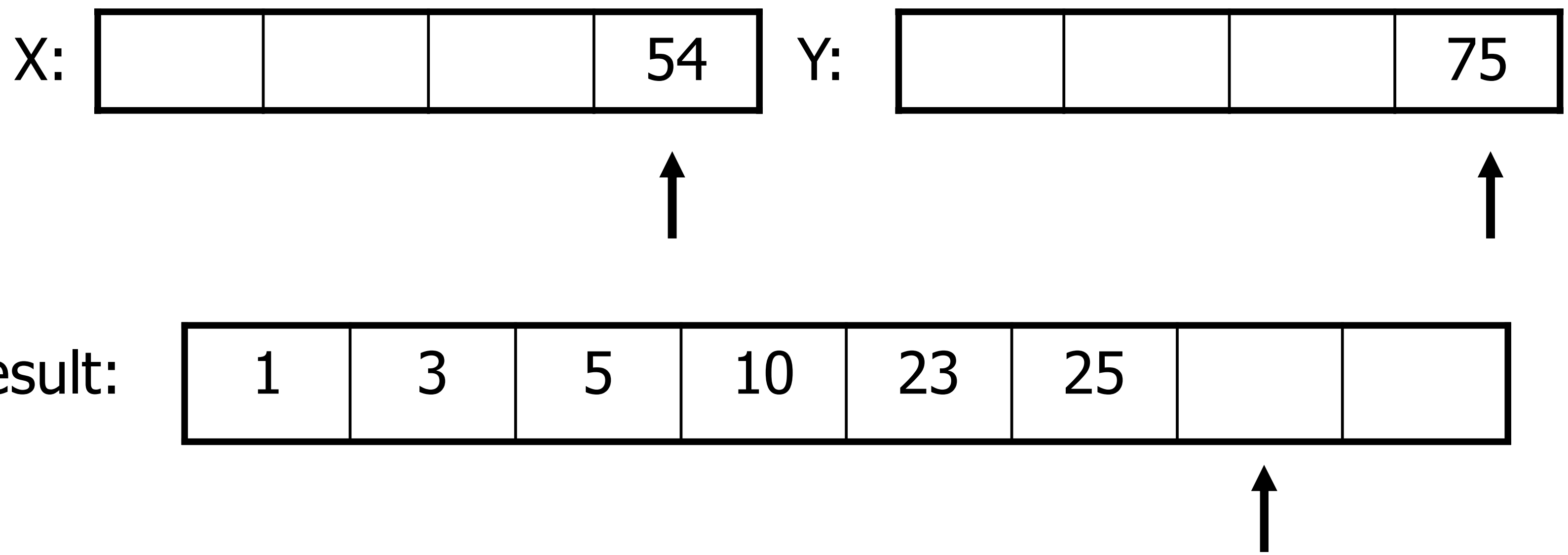
Merging (cont.)



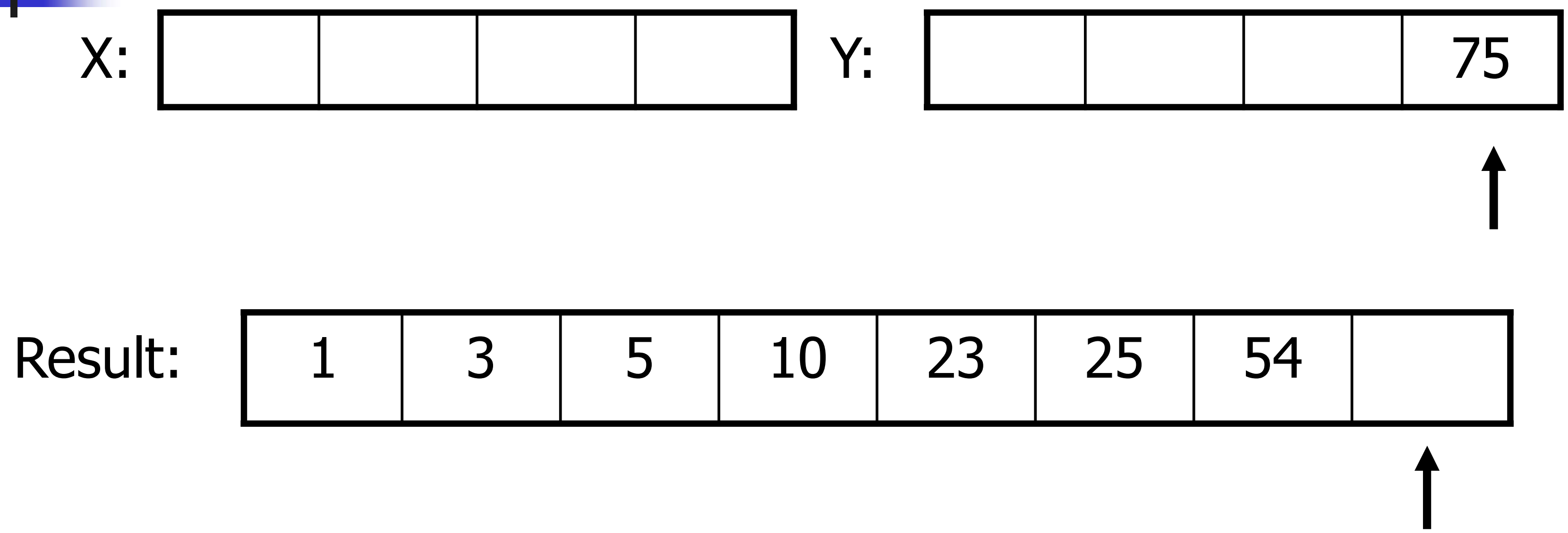
Merging (cont.)



Merging (cont.)



Merging (cont.)



Merging (cont.)

X:

Y:

Result:

1	3	5	10	23	25	54	75

Merge-Sort (A, p, r)

Input: a sequence of *n* numbers stored in array A

Output: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)                       // sort A[p...r] by divide & conquer
  if p < r
     then q ← ⌊(p+r)/2⌋
          MergeSort (A, p, q)
          MergeSort (A, q+1, r)
          Merge (A, p, q, r)              // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(*A*, 1, *n*)

Merge Sort Example

99	6	86	15	58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

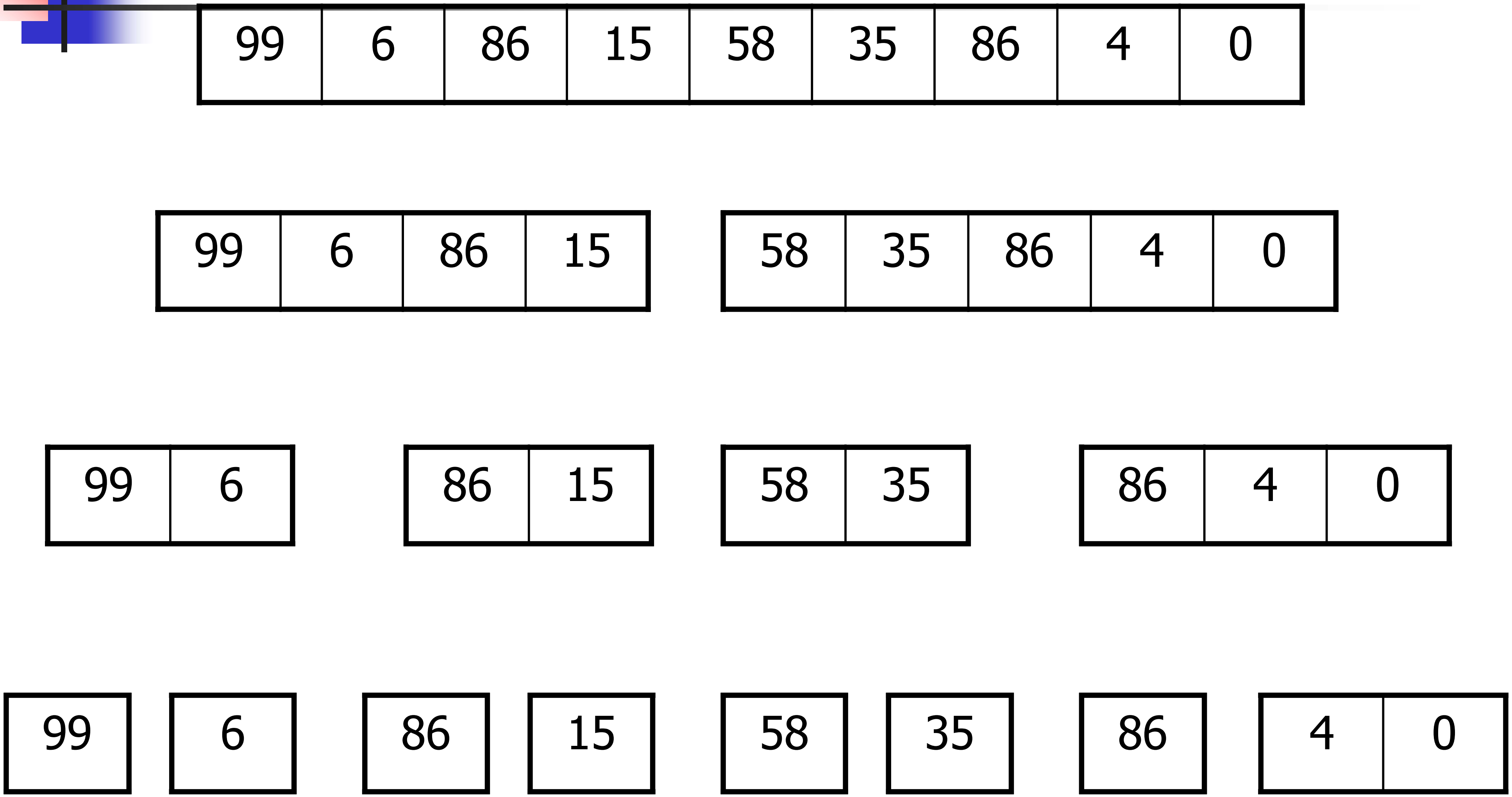
99	6

86	15

58	35

86	4	0

Merge Sort Example



Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

99	6

86	15

58	35

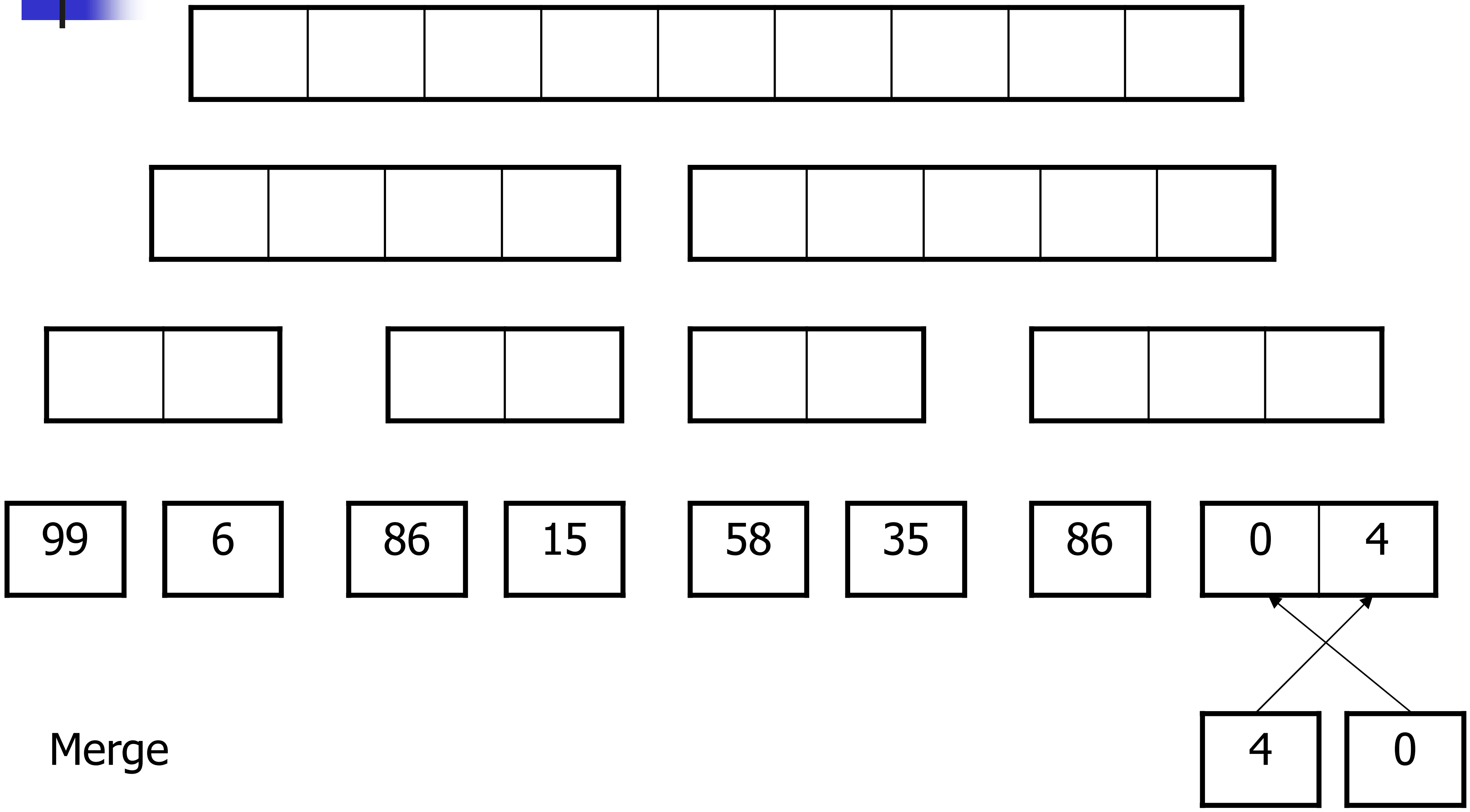
86	4	0

99	6	86	15	58	35	86

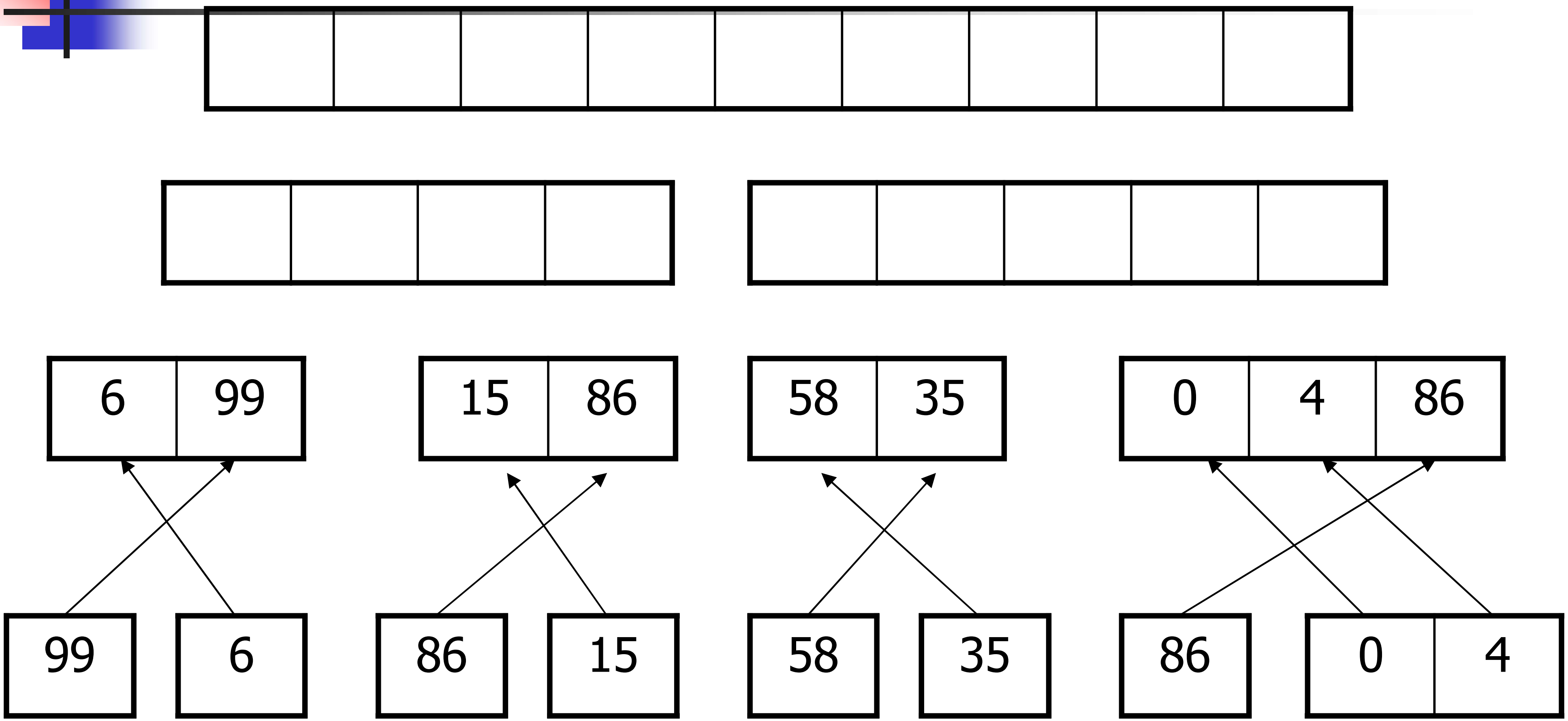
4	0

4	0

Merge Sort Example

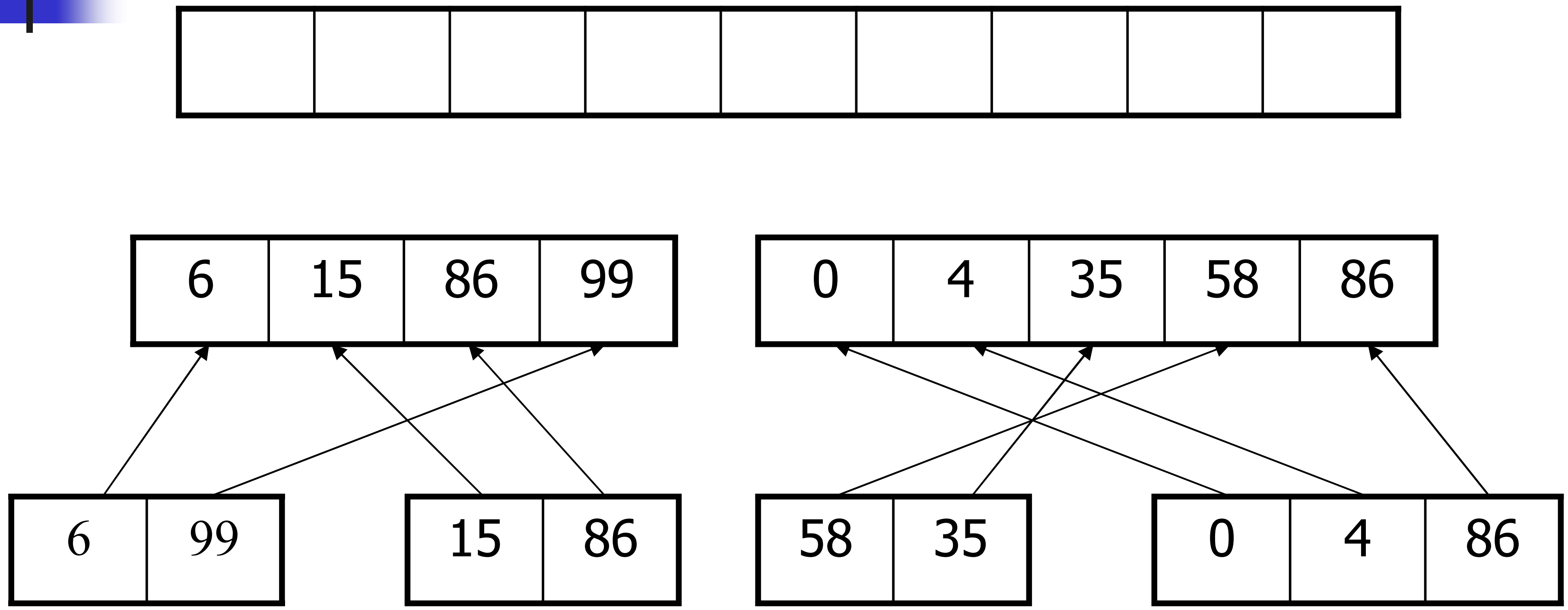


Merge Sort Example



Merge

Merge Sort Example



Merge Sort Example

0	4	6	15	35	58	86	86	99

6	15	86	99

0	4	35	58	86

Merge Sort Example

0	4	6	15	35	58	86	86	99

Merge Algorithm

```
void merge(int *arr, int beg, int mid, int end) {
        int temp[end - beg + 1];
        int i = beg, j = mid+1, k = 1;
        while(i <= mid && j <= end) {
                if(arr[i] <= arr[j]) {
                        temp[k] = arr[i];
                        k += 1; i += 1;
                }
                else {
                        temp[k] = arr[j];
                        k += 1; j += 1;
                }
        }
        // add elements left in the first interval
        while(i <= mid) {
                temp[k] = arr[i];
                k += 1; i += 1;
        }
// add elements left in the second interval
        while(j <= end) {
                temp[k] = arr[j];
                k += 1; j += 1;
        }
        // copy temp to original interval
        for(i = beg; i <= end; i++)
                arr[i] = temp[i - beg];
}
```

Implementing Merge Sort

Basic way to implement merge sort:

- Merging is done with a temporary array of the same size as the input array
  - Pro: Faster
  - Con: The memory requirement is doubled.

Merge Sort Analysis

The Double Memory Merge Sort runs O(n log n) for all cases, because of its Divide and Conquer approach.

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

QuickSort

- The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage.
- As a trade-off, it is possible that the list may not be divided in half.
- When this happens, it has been seen that performance is diminished.

Quick Sort

- **Fastest** known sorting algorithm in practice
- Average case: O(n log n)
- Worst case: $O(n^2)$
  - But the worst case can be made exponentially unlikely.
- Another divide-and-conquer recursive algorithm, like merge sort.

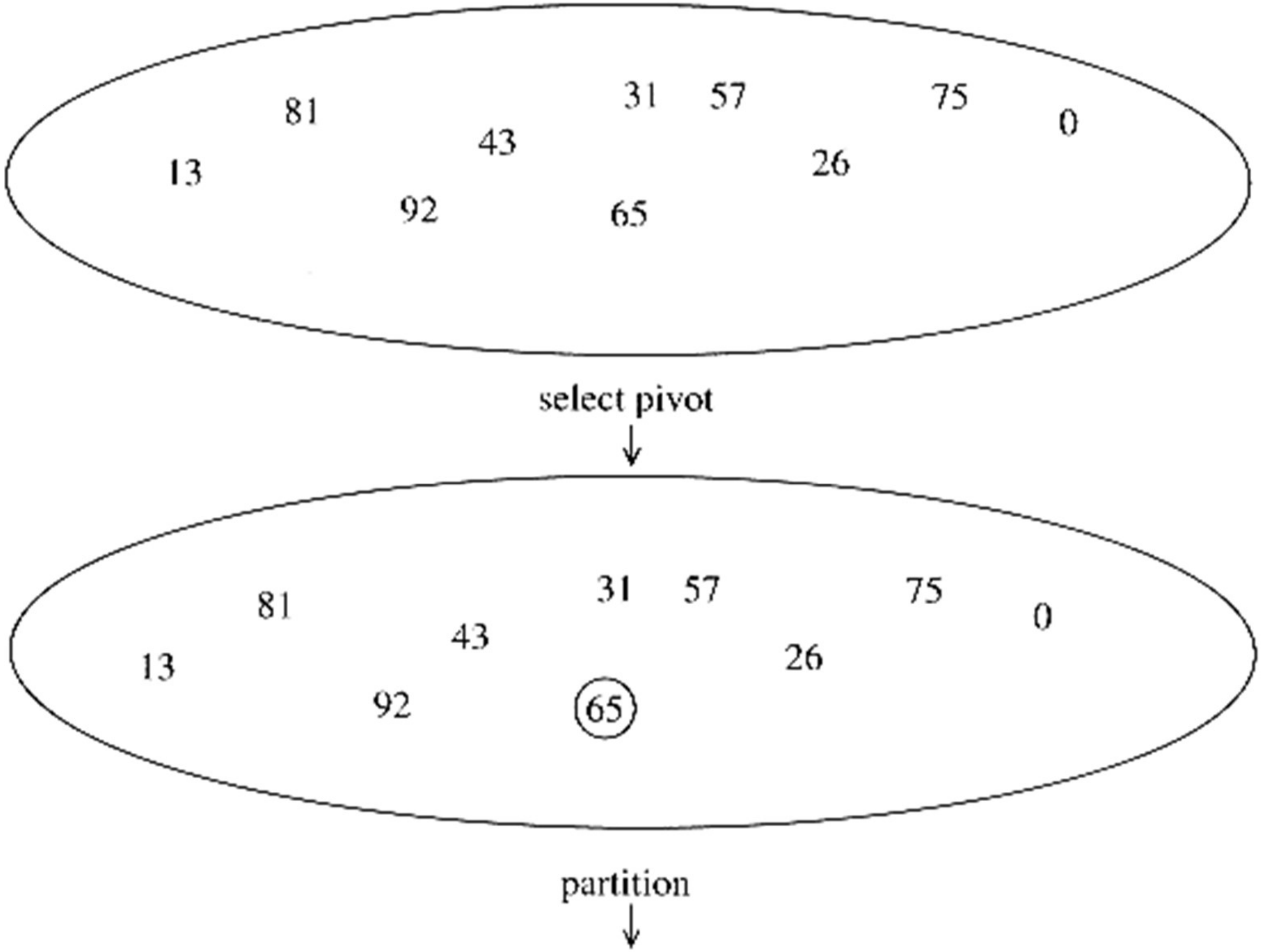
QuickSort

- A quick sort first selects a value, which is called the pivot value.
- Although there are many different ways to choose the pivot value, it is simply used the first/ last item in the list
- The role of the pivot value is to assist with splitting the list.
- The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

Quick Sort: Main Idea

1. If the number of elements in S is 0 or 1, then return (base case).
2. Pick any element v in S (called the pivot).
3. Partition the elements in S except v into two disjoint groups:
   1. $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
   2. $S_2 = \{x \in S - \{v\} \mid x \geq v\}$
4. Return $\{QuickSort(S_1) + v + QuickSort(S_2)\}$

Quick Sort: Example



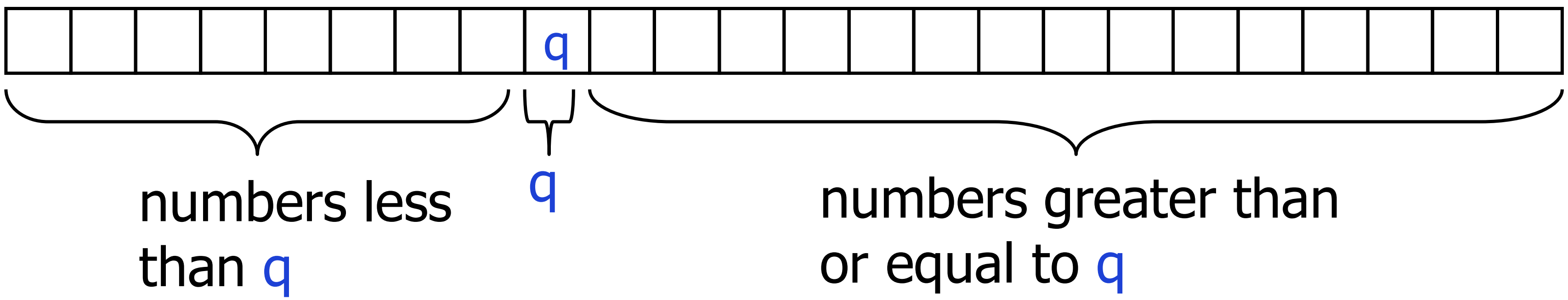
QuickSort

Divide-and-Conquer:

- Divide: partition A[p..r] into two subarrays A[p..q-1] and A[q+1..r] such that each element of A[p..q-1] is $\leq$ A[q], and each element of A[q+1..r] is $\geq$ A[q].
  - Compute q as part of this partitioning.
- Conquer: sort the subarrays A[p..q-1] and A[q+1..r] by recursive calls to Quicksort.
- Combine: the partitioning and recursive sorting leave us with a sorted A[p..r]

Partitioning (Quicksort)

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



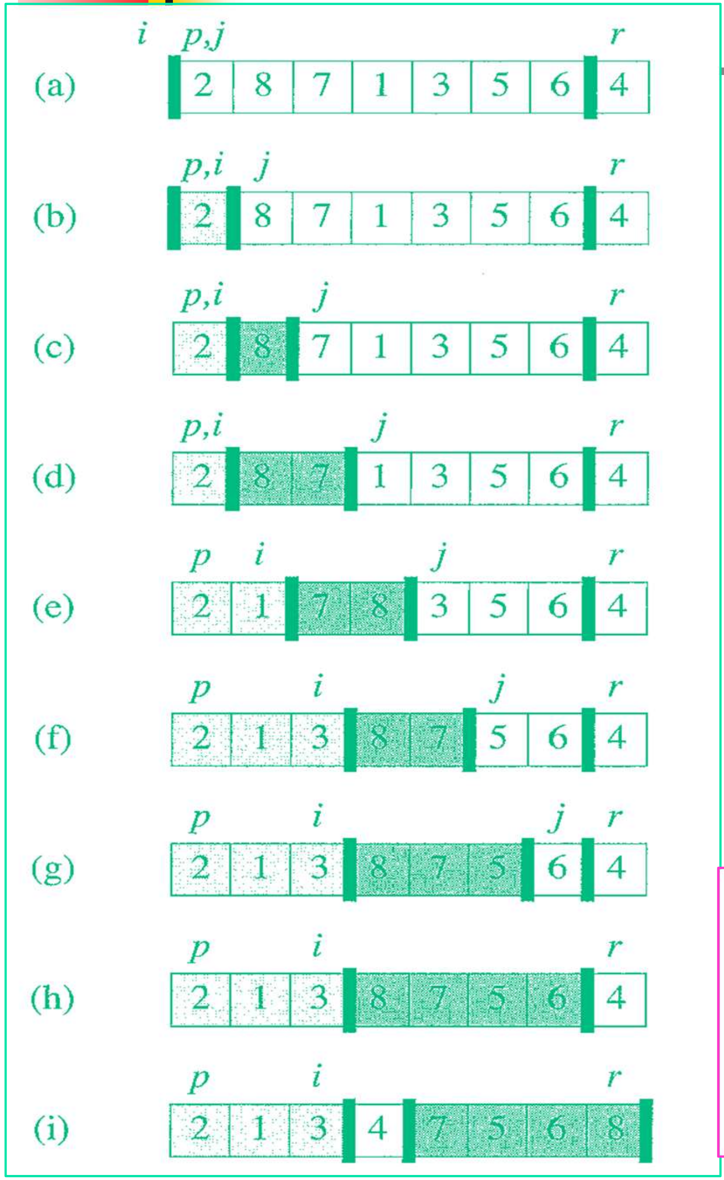
QuickSort

The Pseudo-Code

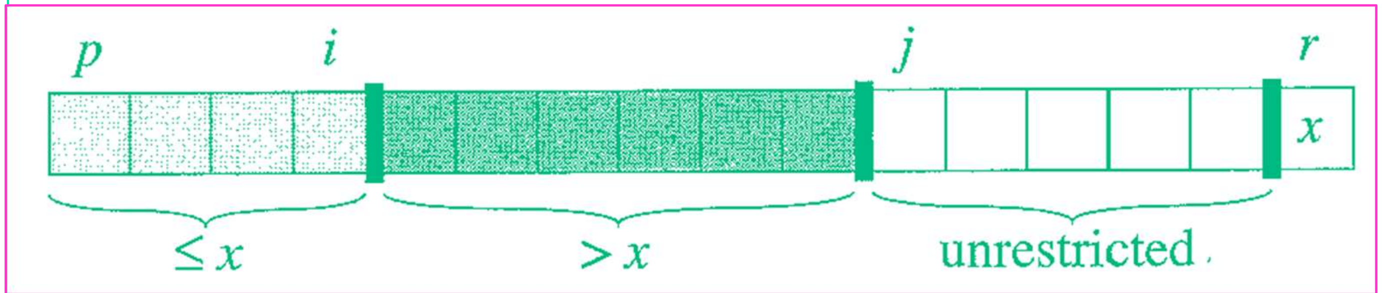
```
QUICKSORT(A, p, r)
if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

QuickSort




```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
        if A[j] ≤ x
              i = i + 1
              exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



Partitioning

- Choose an array value (say, the first/ last) 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

Another way: Partition method

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

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

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

Quicksort method

```
void quicksort(int a[], int left, int right) {
    if (left < right) {
        int p = partition(a, left, right);
        quicksort(a, left, p - 1);
        quicksort(a, p + 1, right);
    }
}
```

quickSort(a, 0, 5)

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

quickSort(a,0, 5)

partition(a, 0, 5)

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

quickSort(a, 0, 5)

partition(a, 0, 5)

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

pivot= ?

Partition Initialization...

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

Partition Initialization...

quickSort(a, 0, 5)

partition(a, 0, 5)

$\boxed{\text{pivot=6}}$

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

left

Partition Initialization...

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

Partition Initialization...

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

left

right

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

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

left

right

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

quickSort(a, 0, 5)

partition(a, 0, 5)

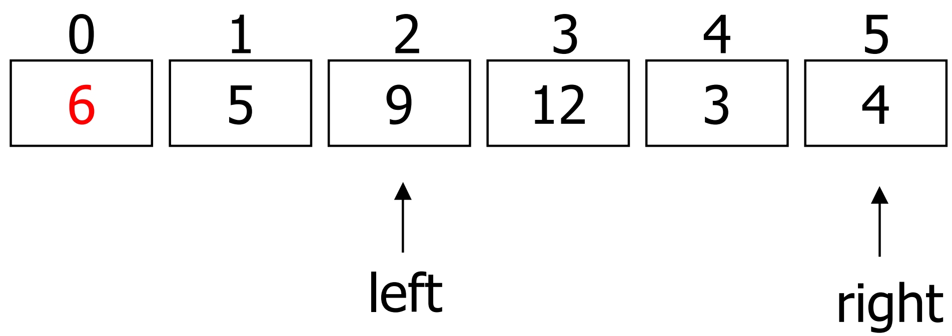
pivot=6

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

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6



quickSort(a, 0, 5)

partition(a, 0, 5)

$\boxed{\text{pivot=6}}$

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

left

right

swap arr[left] and arr[right]

quickSort(a, 0, 5)

partition(a, 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(a, 0, 5)

partition(a, 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

left

right

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

left

right

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

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

left right

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

left right

swap arr[left] and arr[right]

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

left right

swap arr[left] and arr[right]

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

repeat right/left scan
UNTIL left & right cross

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

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

quickSort(a, 0, 5)

partition(a, 0, 5)

pivot=6

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

left

right

quickSort(a, 0, 5)

partition(a, 0, 5)

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

pivot=6

left

right

right & left CROSS!!!

quickSort(a, 0, 5)

partition(a, 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(a, 0, 5)

partition(a, 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(a, 0, 5)

partition(a, 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]
2 - Return new location of pivot to caller

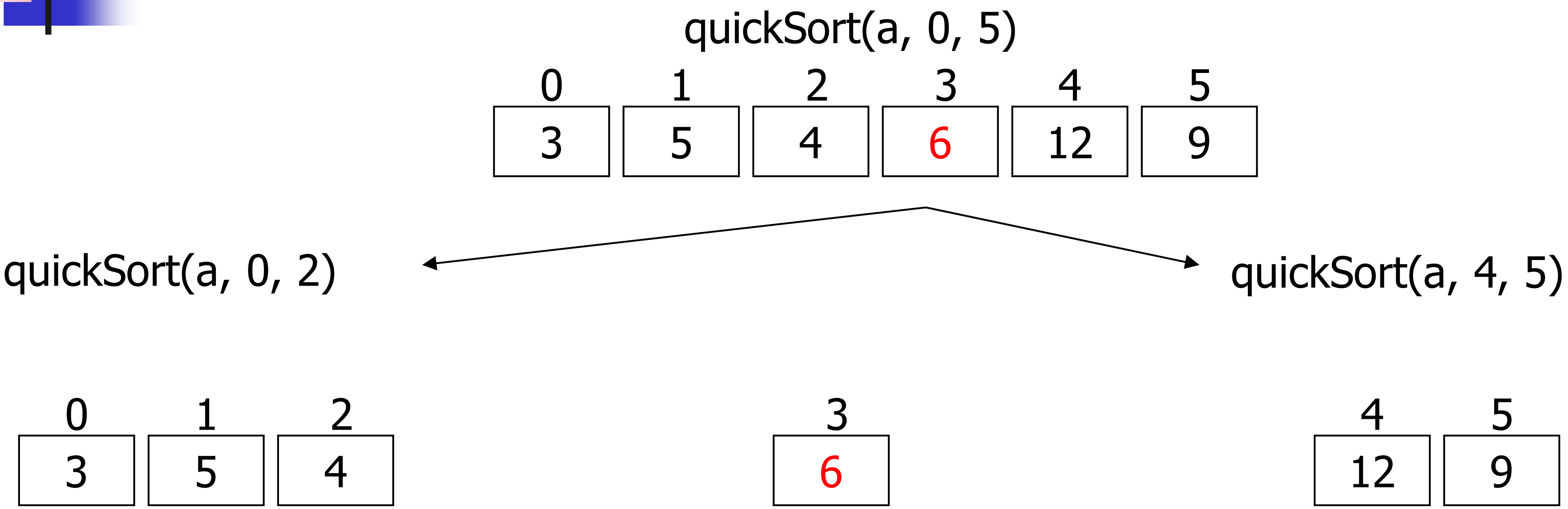
return 3

quickSort(a, 0, 5)

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

pivot
position

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


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

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

0	1	2
3	5	4

3
6

4	5
12	9

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

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

Partition Initialization...

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

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

Partition Initialization...

quickSort(a, 0, 5)

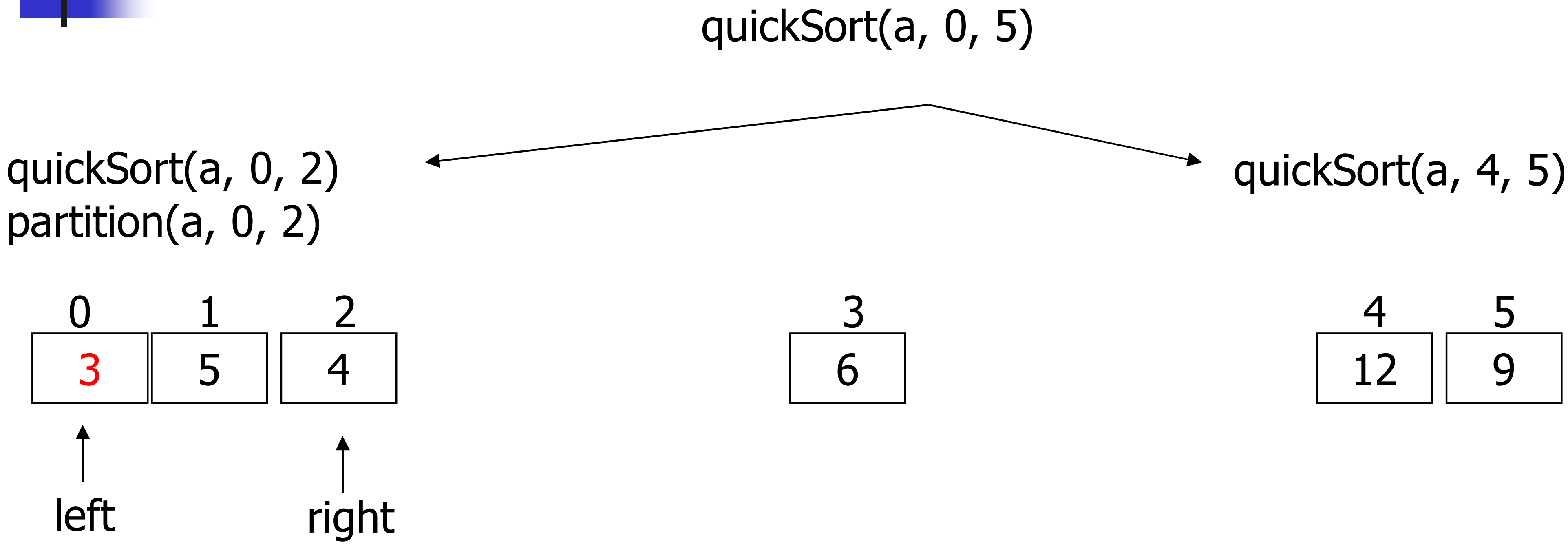
quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

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

left

Partition Initialization...



Partition Initialization...

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

0	1	2
3	5	4

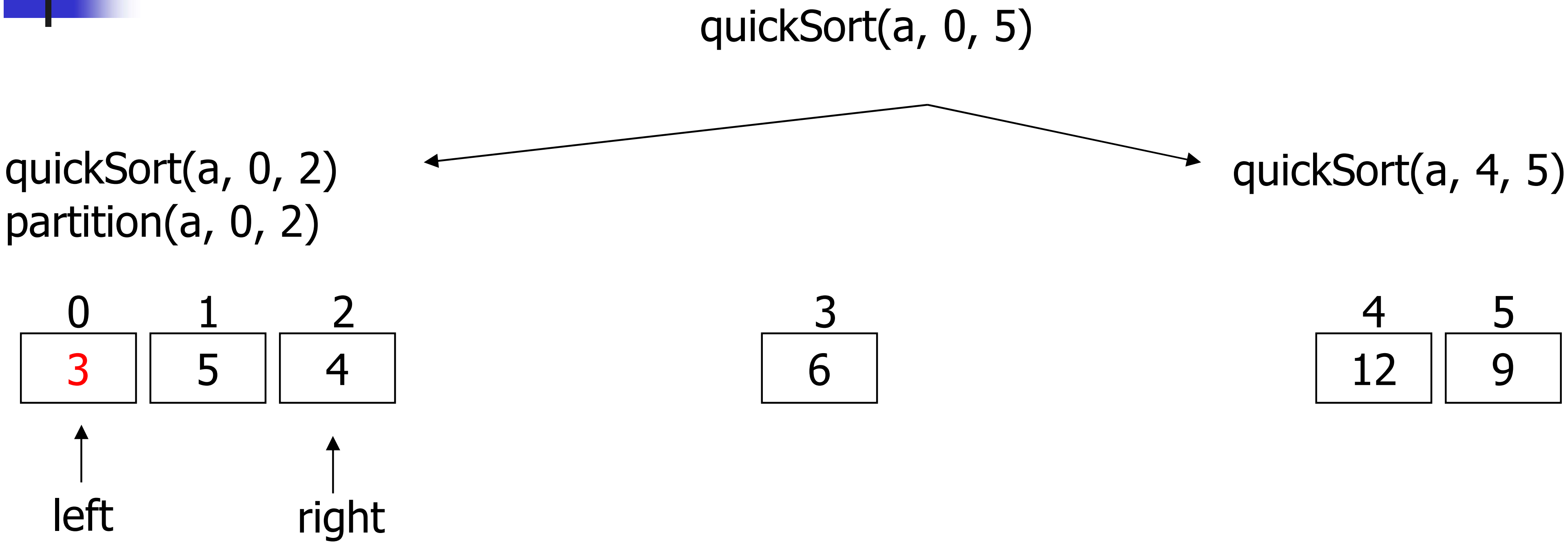
left

right

3
6

4	5
12	9

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


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

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

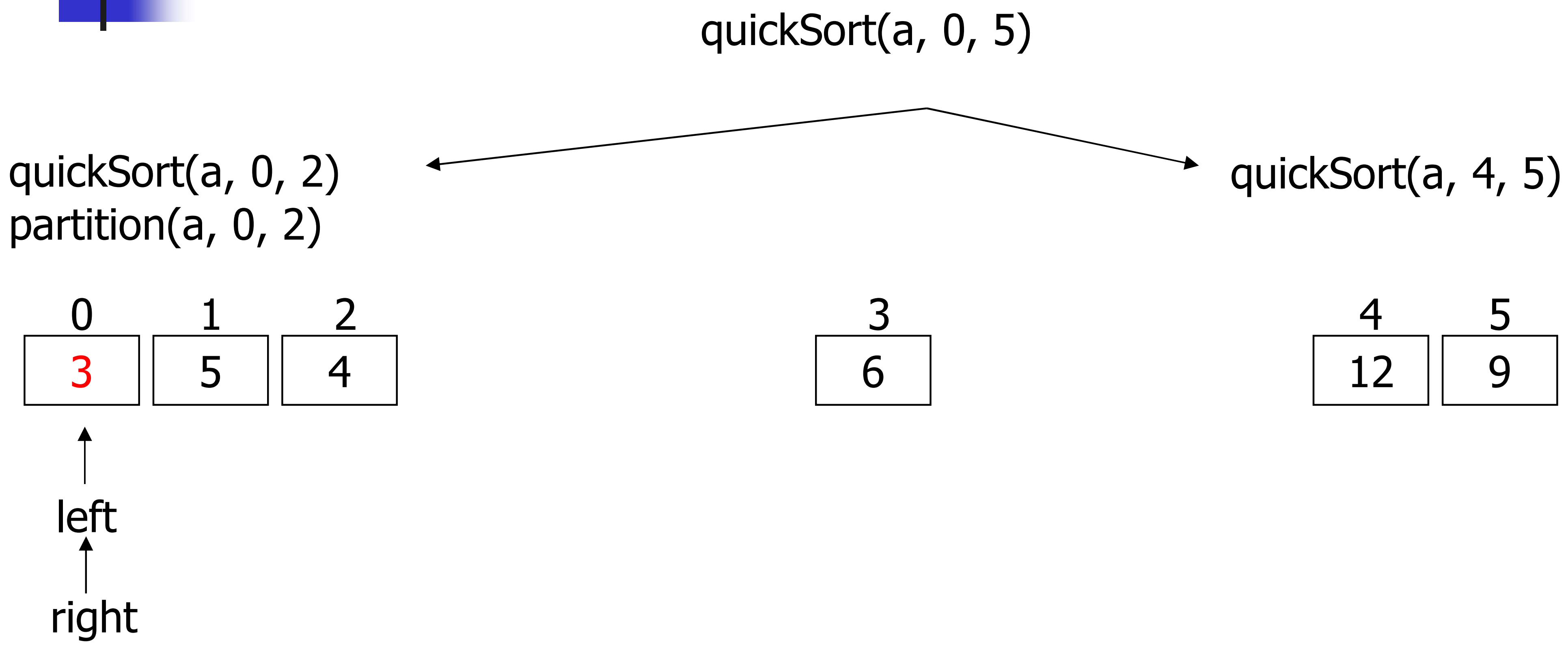
quickSort(a, 4, 5)

0	1	2
3	5	4

left right

3
6

4	5
12	9



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

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

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

left

right

right & left CROSS!!!

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

0	1	2
3	5	4

3
6

4	5
12	9

left

right

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

quickSort(a, 0, 5)

quickSort(a, 0, 2)
partition(a, 0, 2)

quickSort(a, 4, 5)

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

left

right

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

return 0

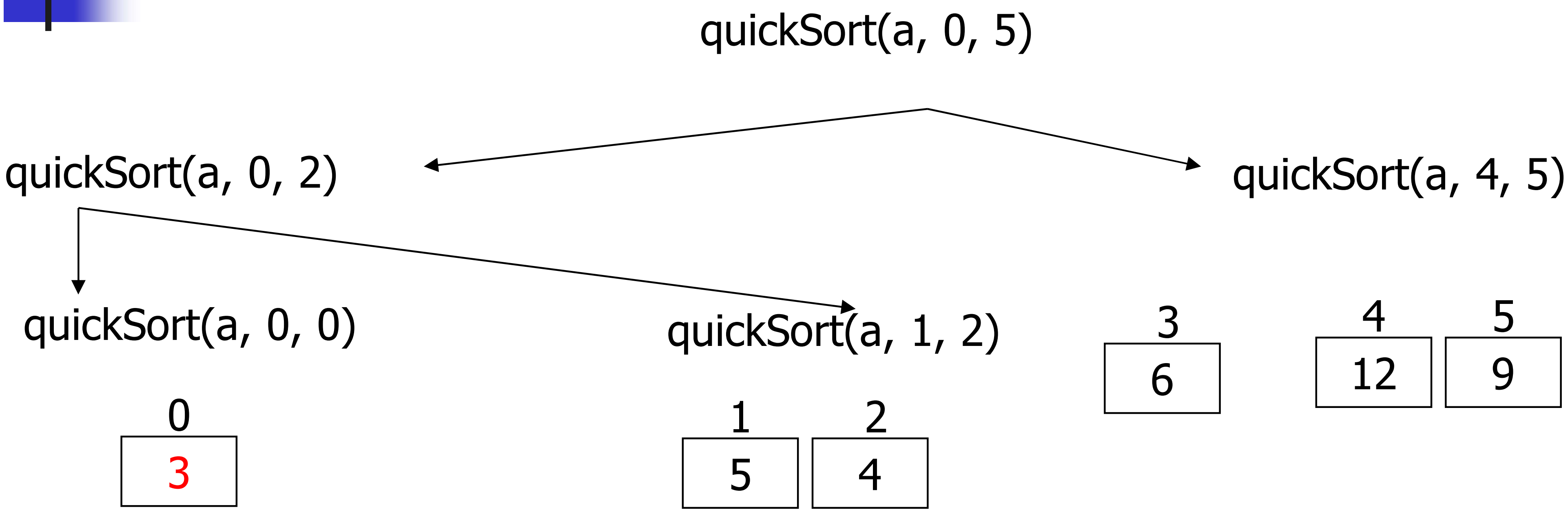
quickSort(a, 0, 5)

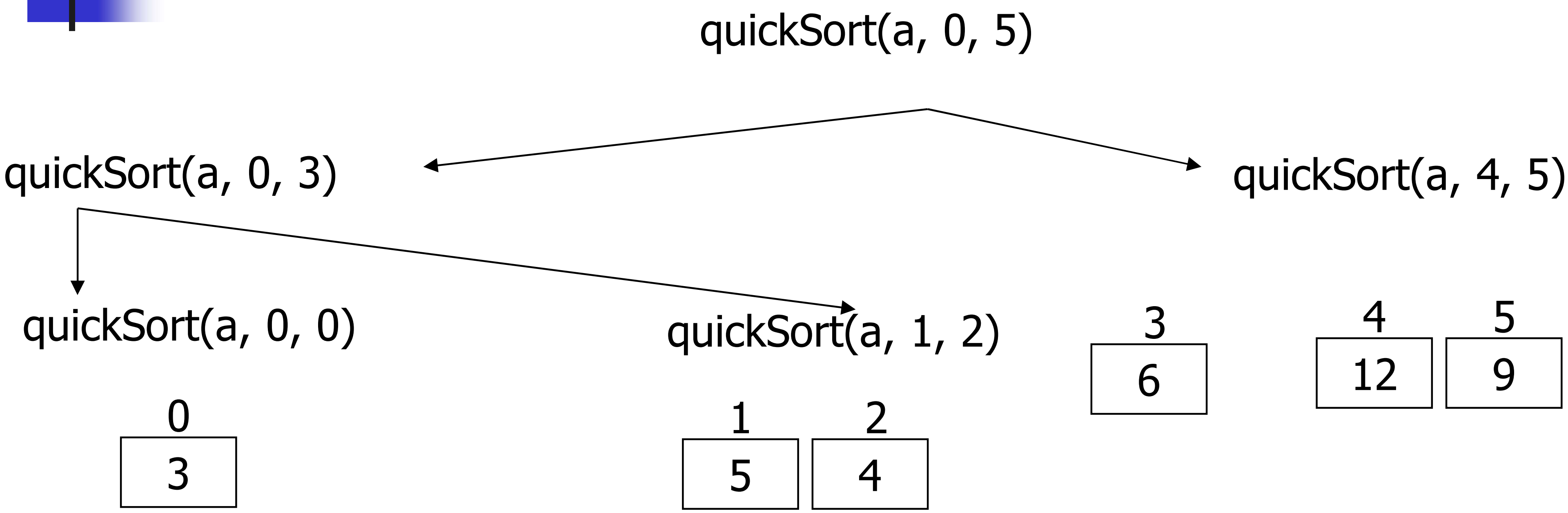
quickSort(a, 0, 2)

quickSort(a, 4, 5)

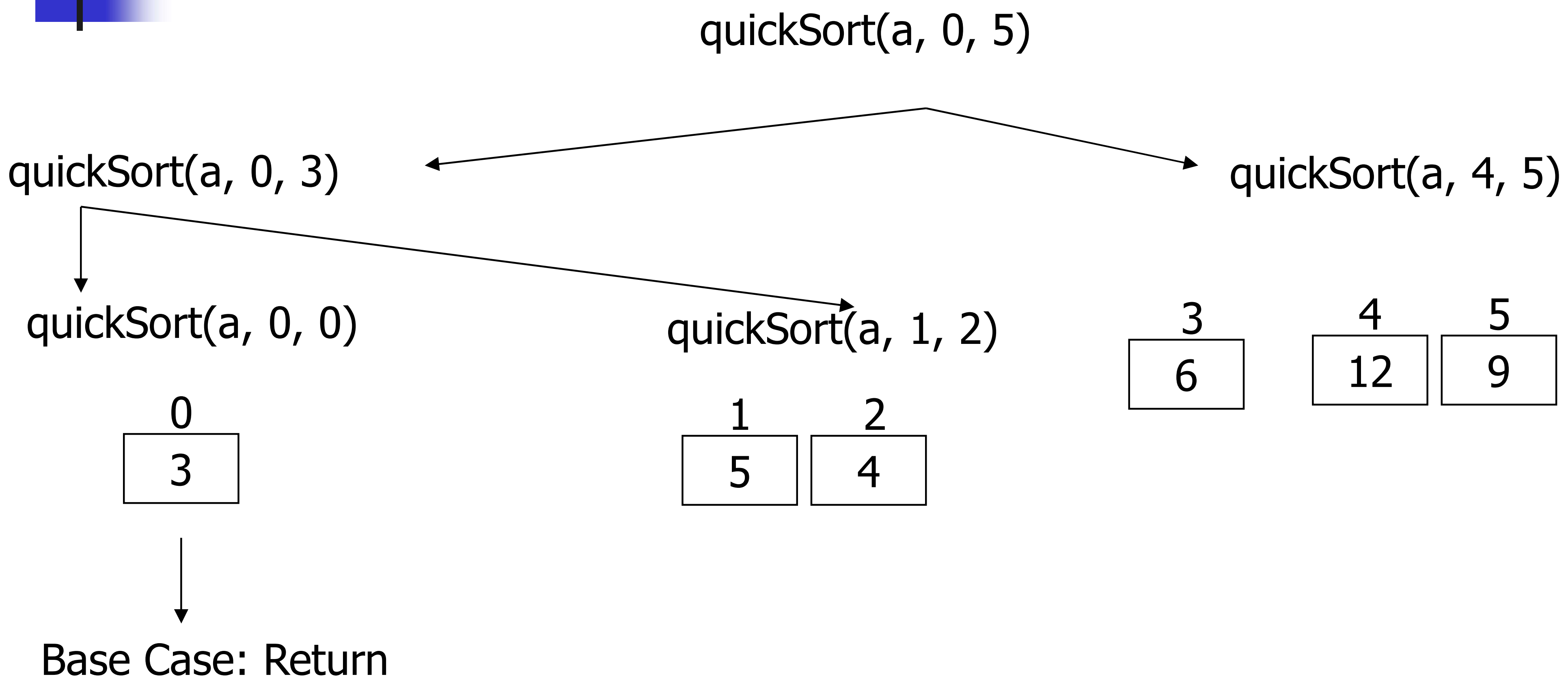
0	1	2	3	4	5
3	5	4	6	12	9

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


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



Base case triggered...
halting recursion.


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

quickSort(a, 0, 5)

quickSort(a, 0, 3)

quickSort(a, 4, 5)

quickSort(a, 0, 0)

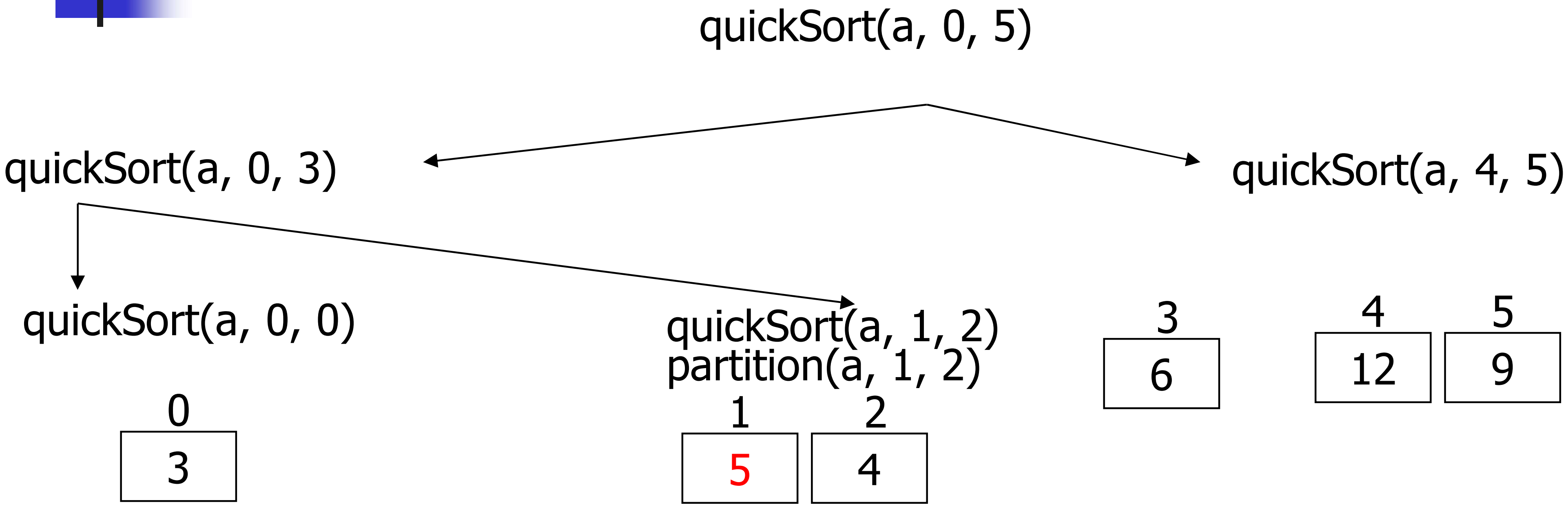
0
3

quickSort(a, 1, 2)
partition(a, 1, 2)

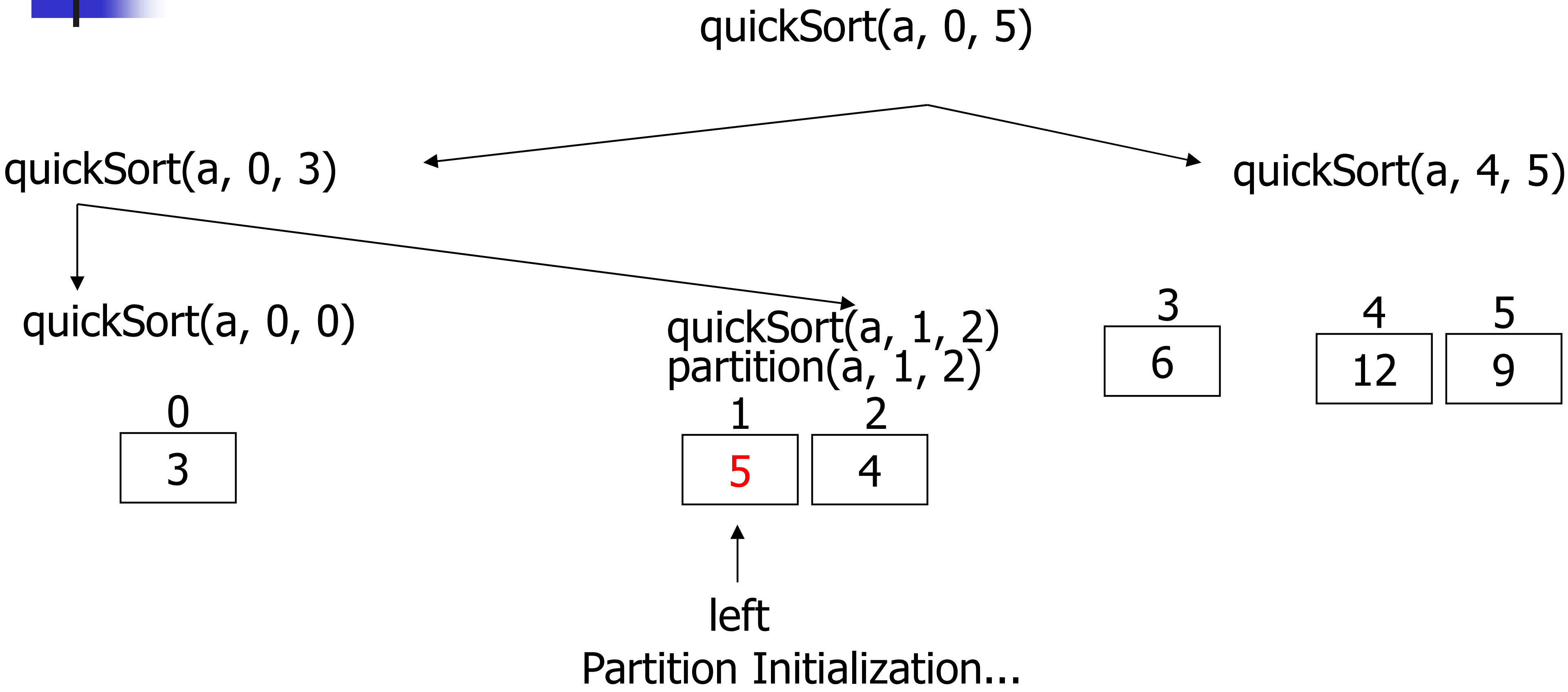
1	2
5	4

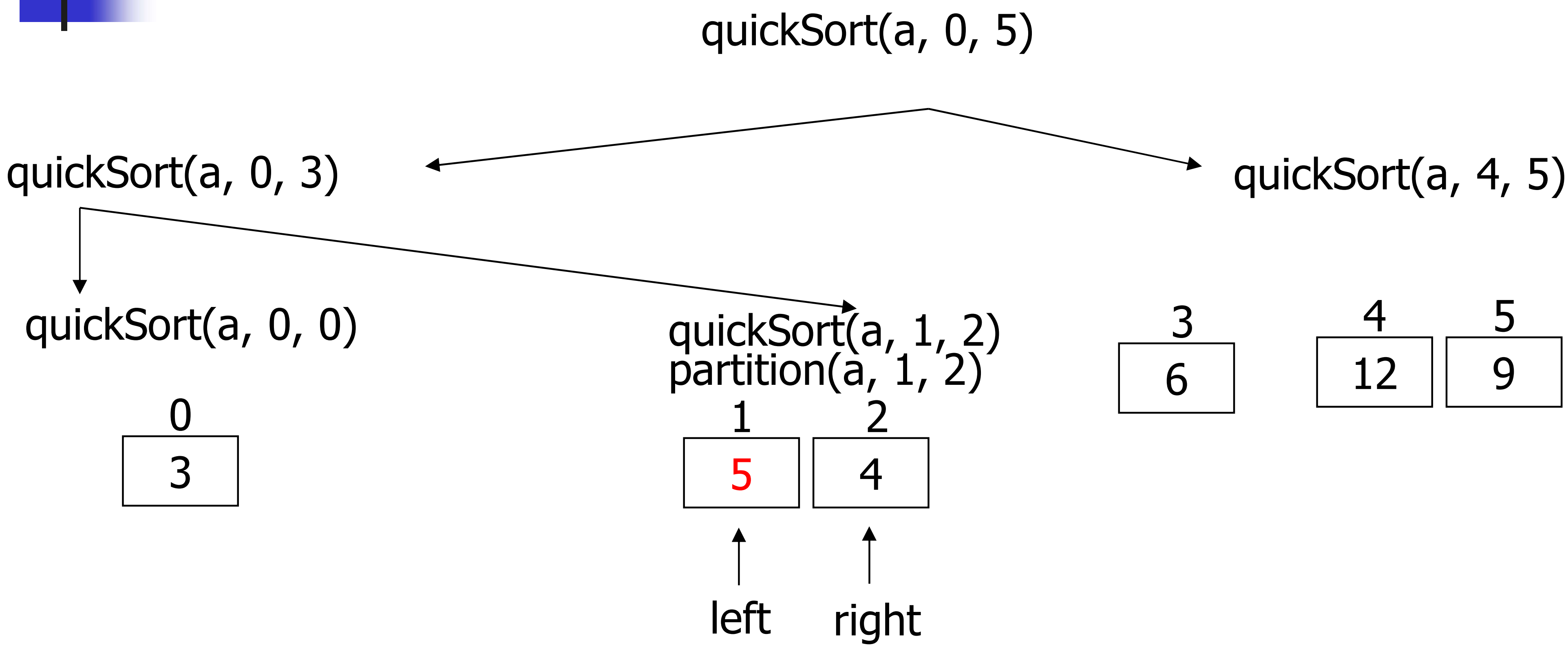
3	4	5
6	12	9

Partition Initialization...

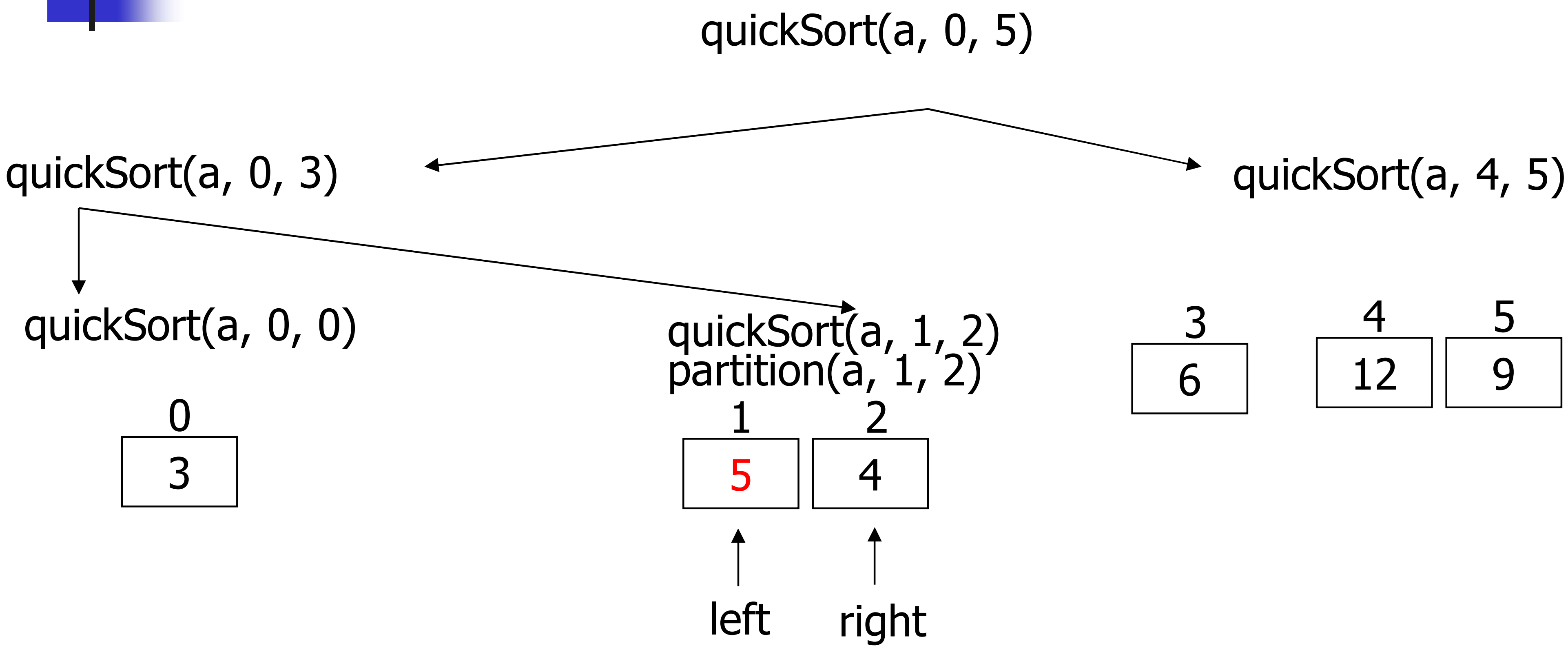


Partition Initialization...

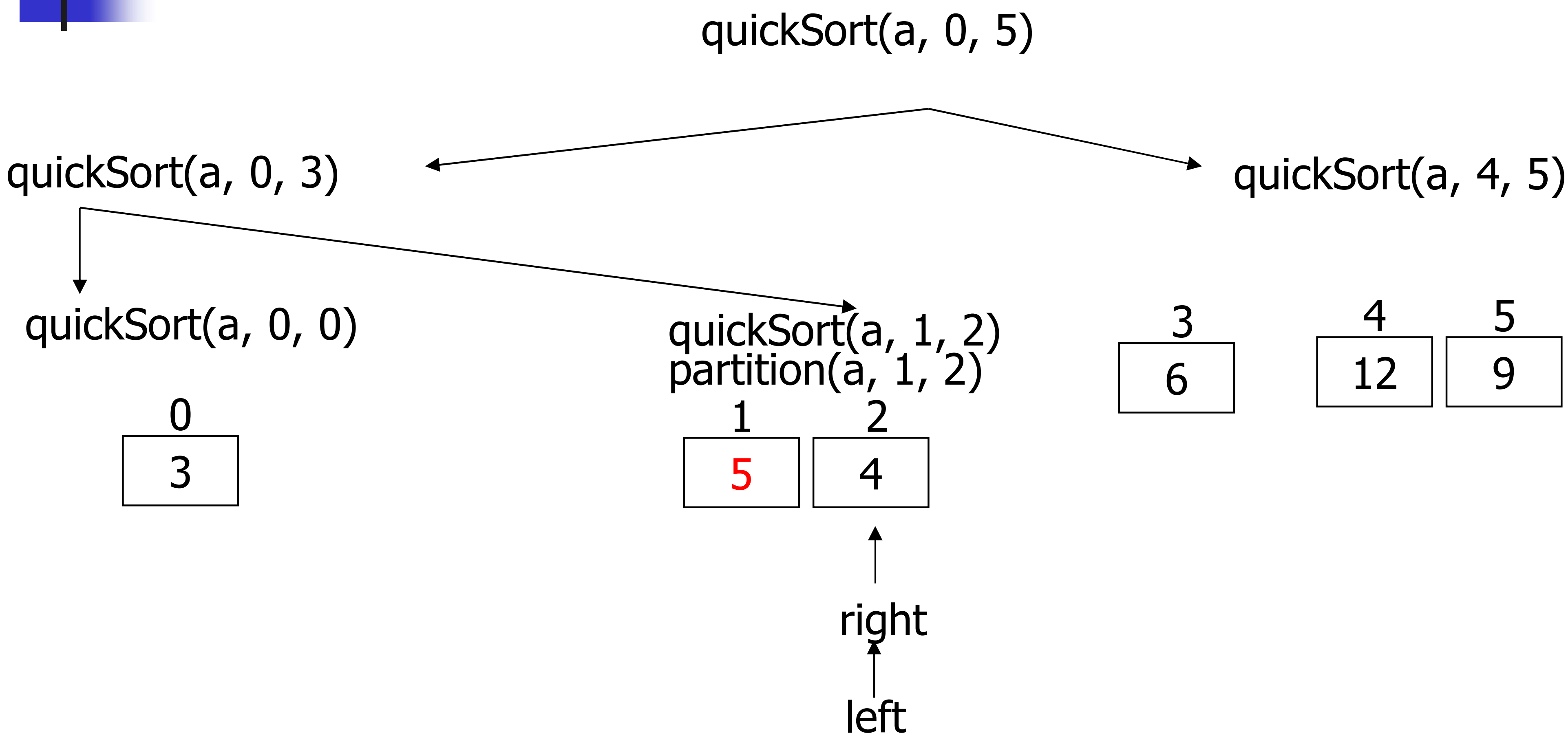

quickSort(a, 0, 5)
quickSort(a, 0, 3)
quickSort(a, 4, 5)
quickSort(a, 0, 0)
quickSort(a, 1, 2)
partition(a, 1, 2)
0
3
1
2
5
4
3
6
4
5
12
9
left
Partition Initialization...

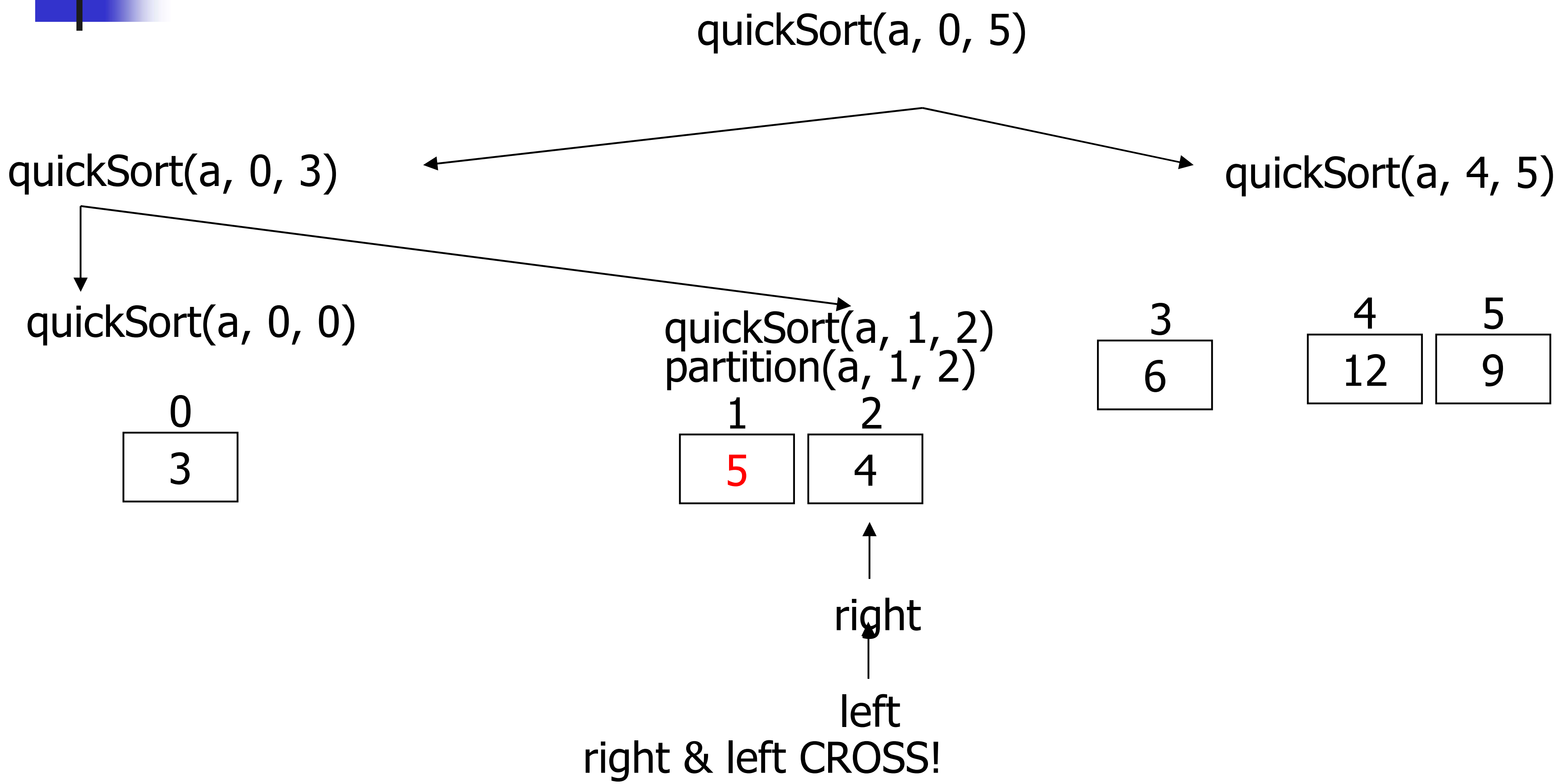


right moves to the left until value that should be to left of pivot...



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


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


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

quickSort(a, 0, 5)

quickSort(a, 0, 3)

quickSort(a, 4, 5)

quickSort(a, 0, 0)

0
3

quickSort(a, 1, 2)
partition(a, 1, 2)

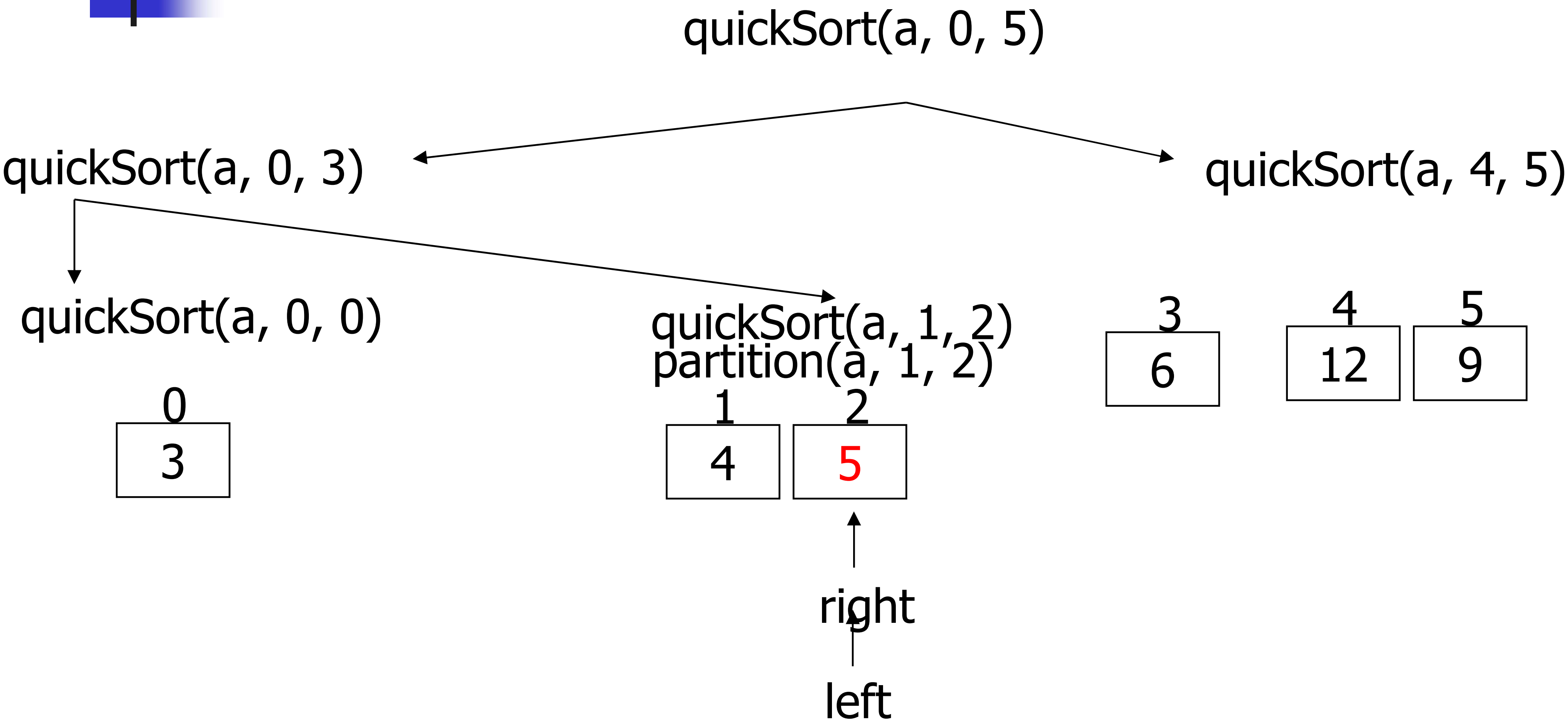
1	2
5	4

3	4	5
6	12	9

right

left

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



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

quickSort(a, 0, 5)

quickSort(a, 0, 3)

quickSort(a, 4, 5)

quickSort(a, 0, 0)

0
3

quickSort(a, 1, 2)
partition(a, 1, 2)

1	2
4	5

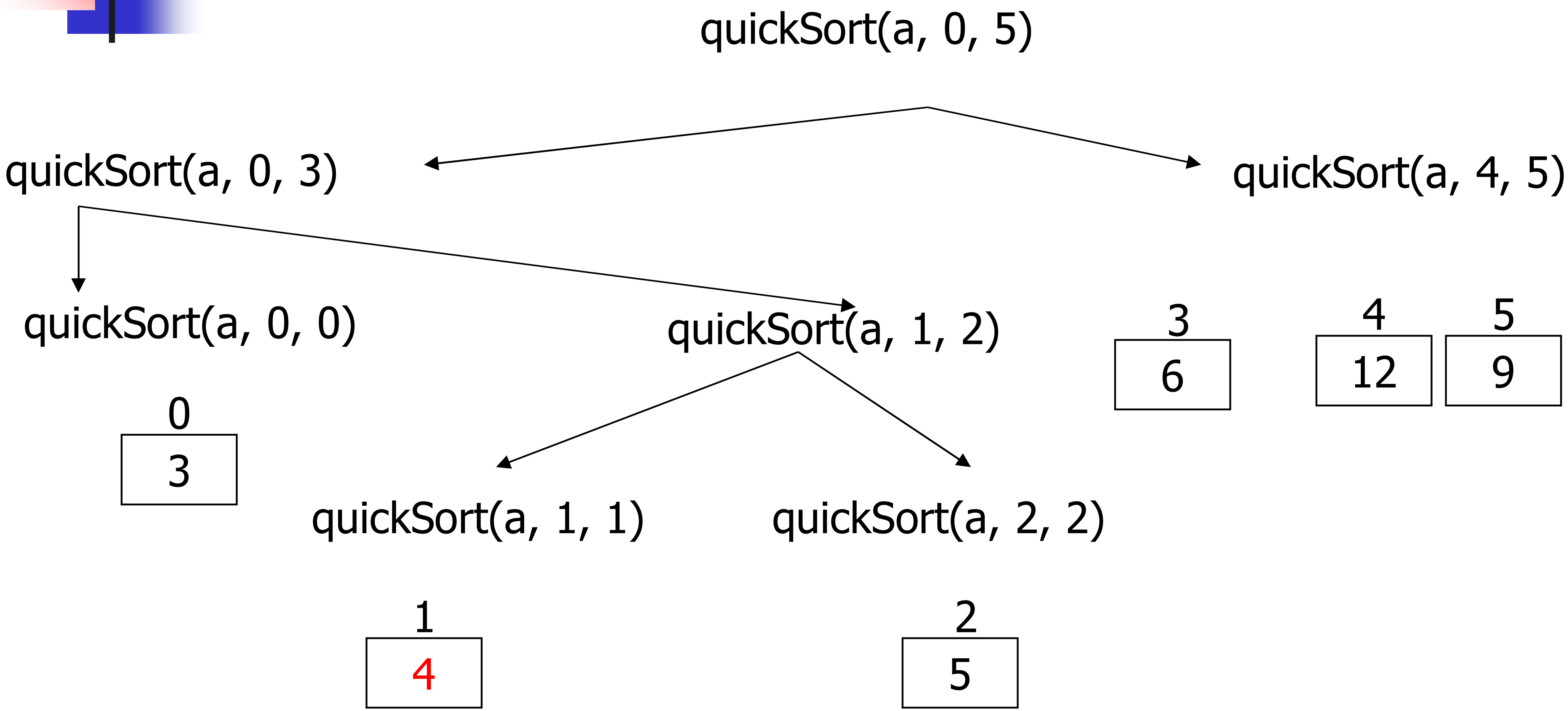
3	4	5
6	12	9

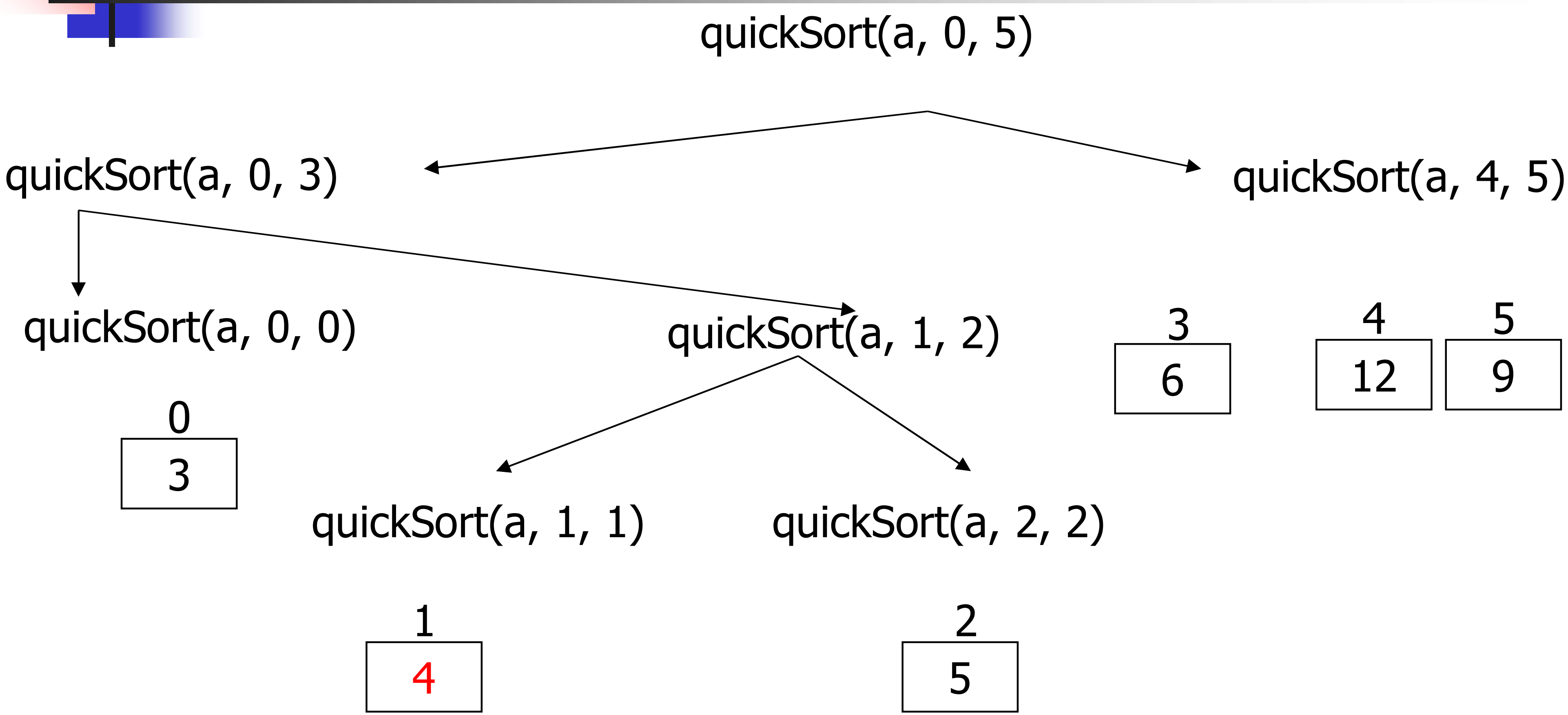
right

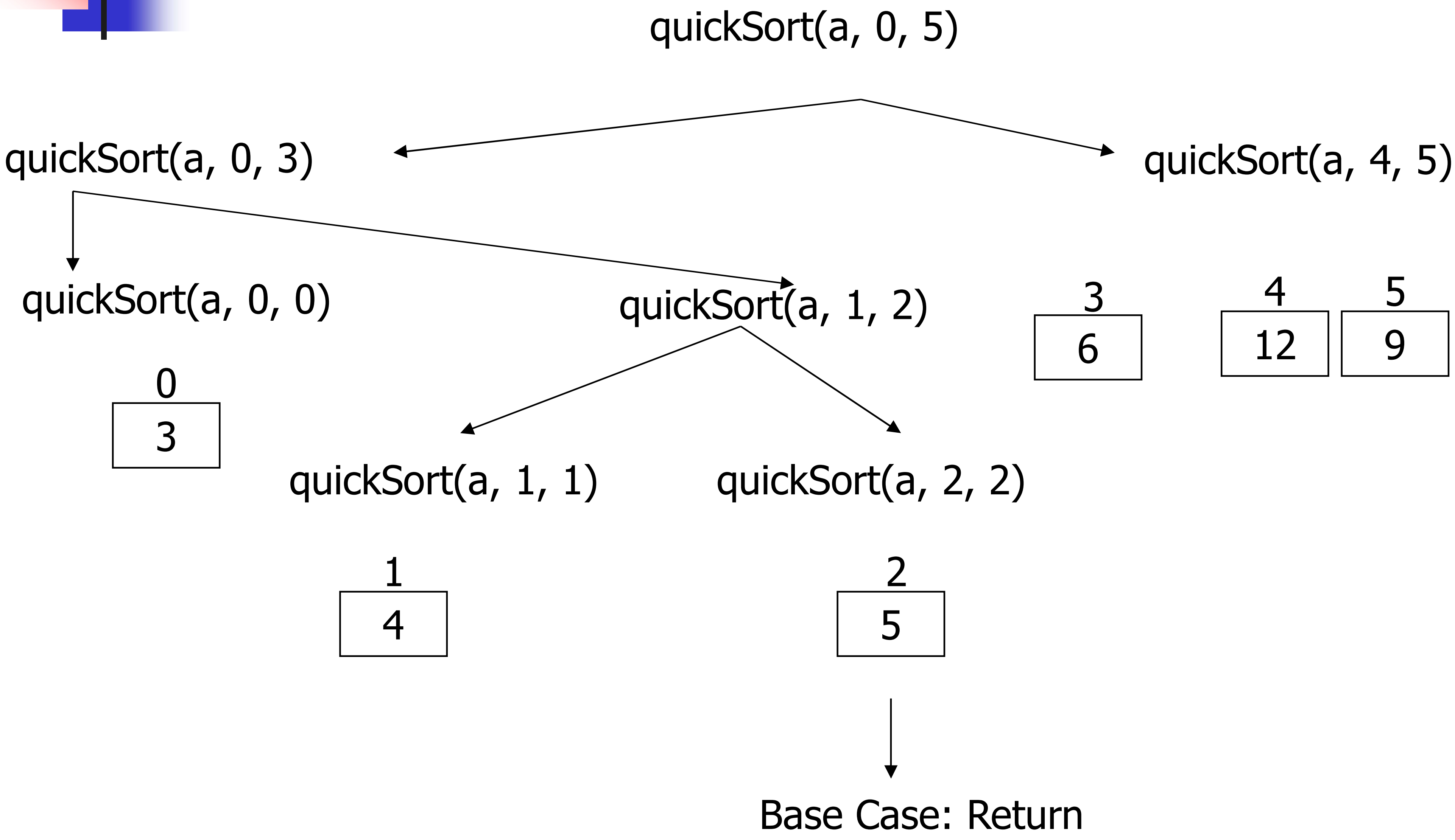
left

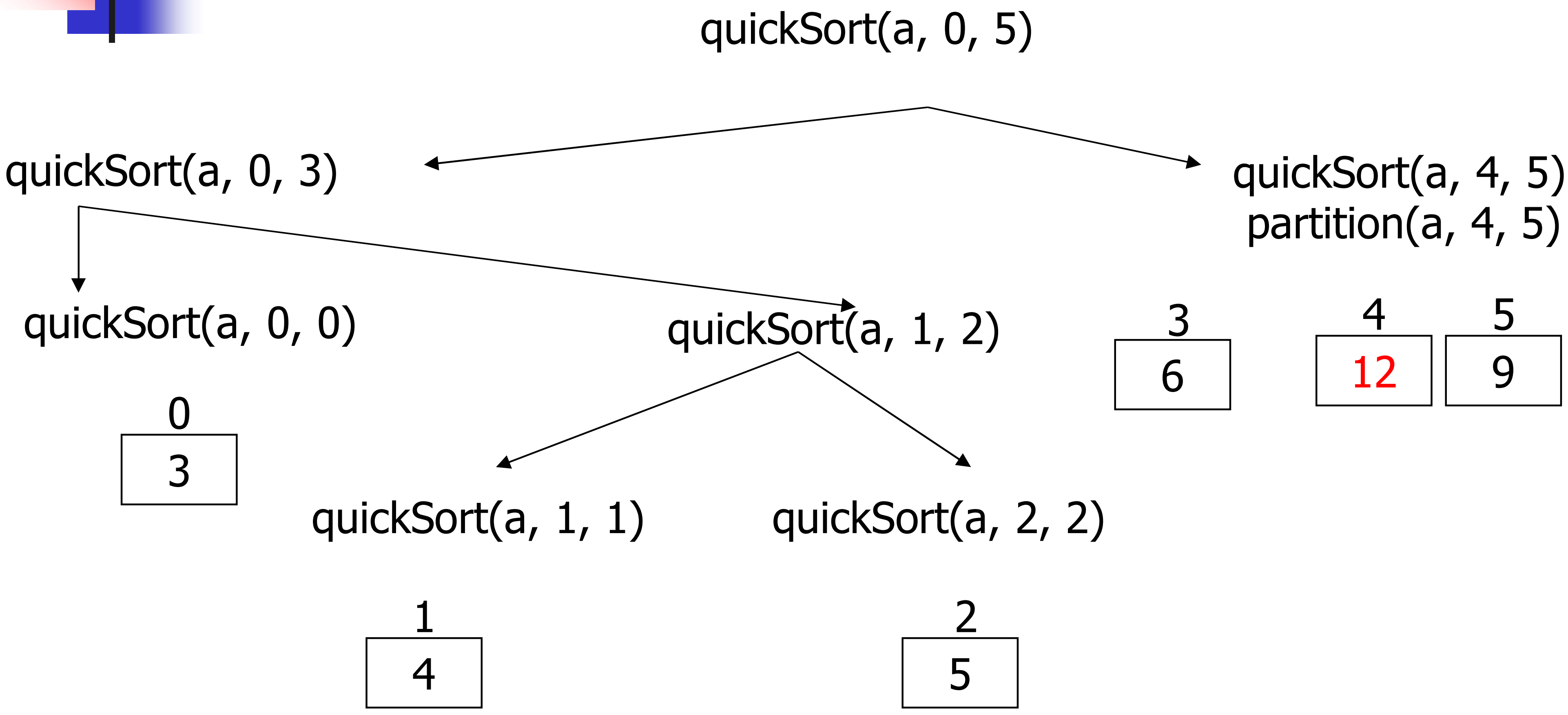
right & left CROSS!
1- swap pivot and arr[right]
2 – return new position of pivot

return 2

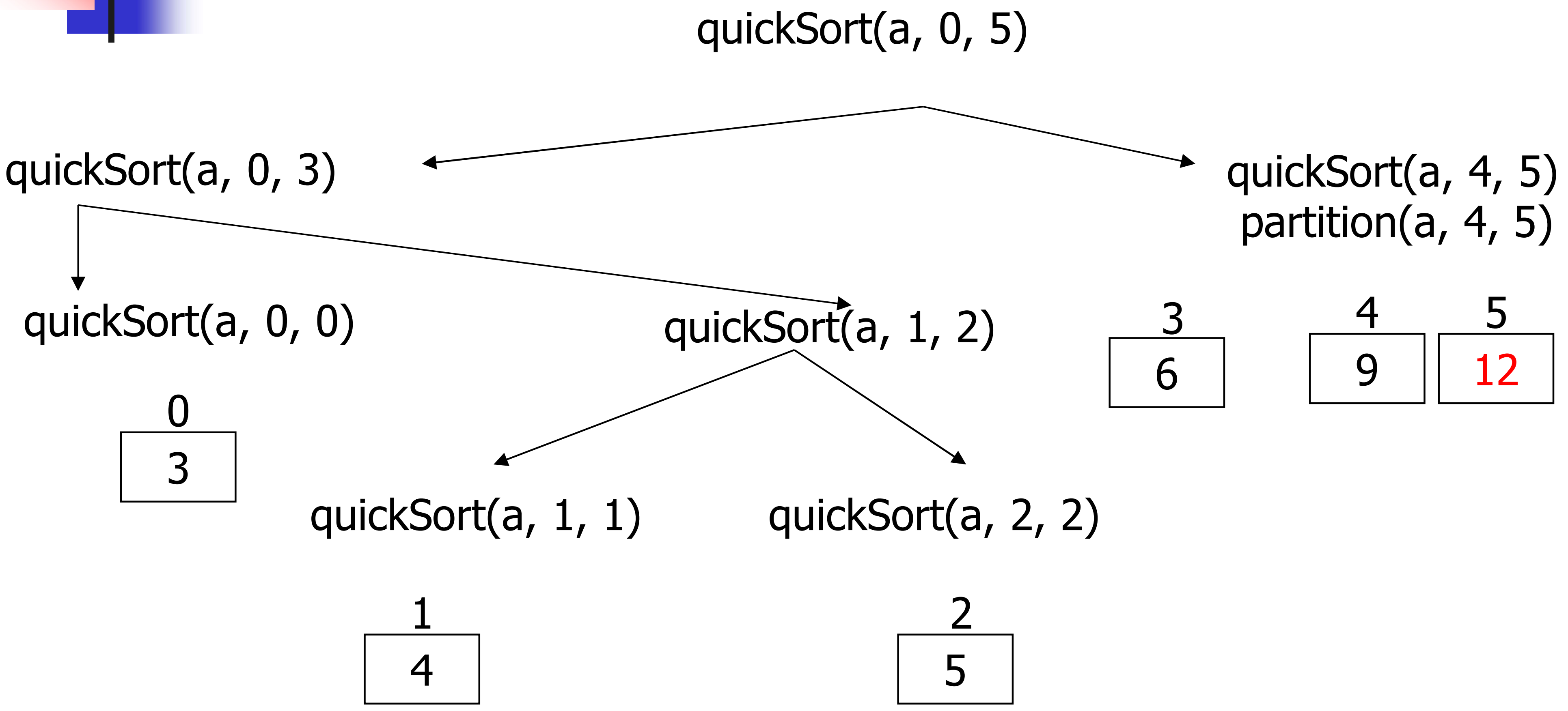

quickSort(a, 0, 5)
quickSort(a, 0, 3)
quickSort(a, 4, 5)
quickSort(a, 0, 0)
quickSort(a, 1, 2)
3
6
4
12
5
9
0
3
quickSort(a, 1, 1)
quickSort(a, 2, 2)
1
4
2
5

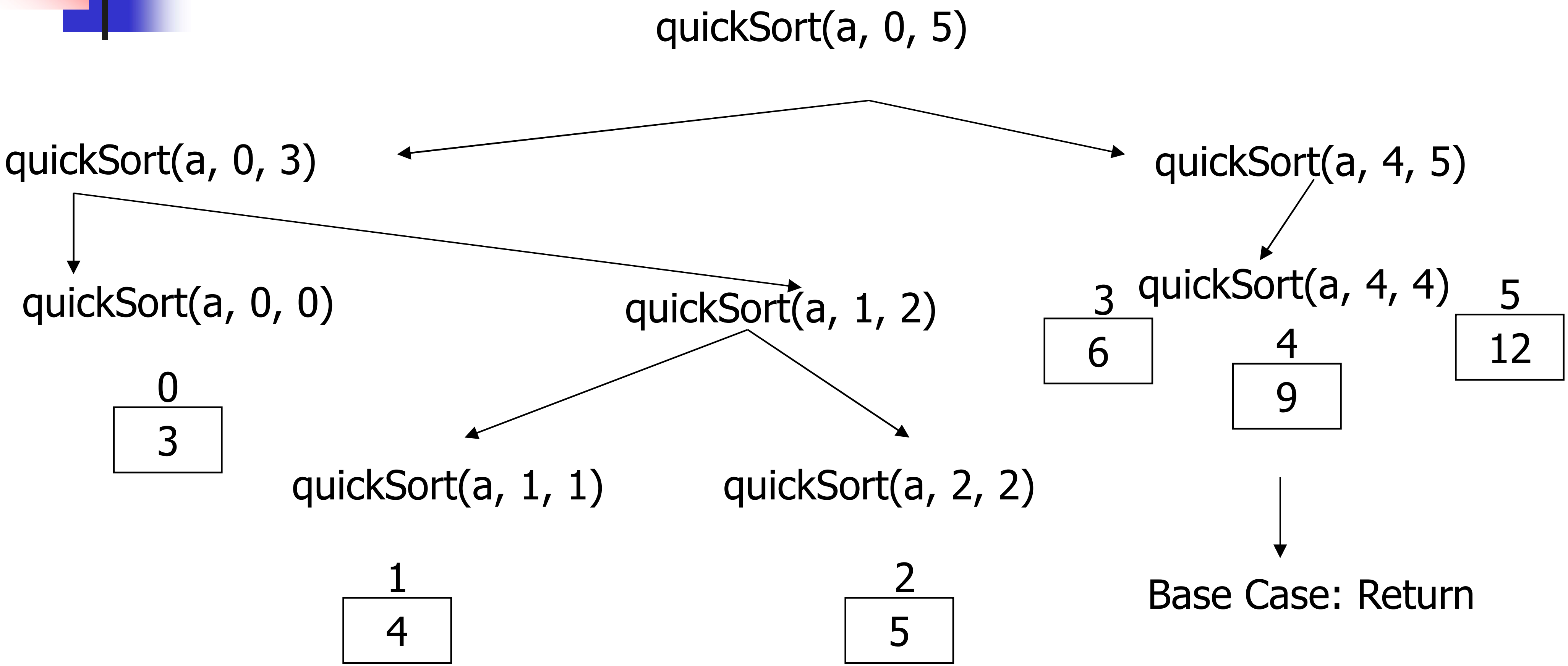

quickSort(a, 0, 5)
quickSort(a, 0, 3)
quickSort(a, 4, 5)
quickSort(a, 0, 0)
quickSort(a, 1, 2)
3
6
4
12
5
9
0
3
quickSort(a, 1, 1)
quickSort(a, 2, 2)
1
4
2
5


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



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


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


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

Example for Quick Sort

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	left	right
{ 26	5	37	1	61	11	59	15	48	19 }	0	9
{ 11	5	19	1	15 }	26	{ 59	61	48	37 }	0	4
{ 1	5 }	11	{ 19	15 }	26	{ 59	61	48	37 }	0	1
1	5	11	15	19	26	{ 59	61	48	37 }	3	4
1	5	11	15	19	26	{ 48	37 }	59	{ 61 }	6	9
1	5	11	15	19	26	37	48	59	{ 61 }	6	7
1	5	11	15	19	26	37	48	59	61	9	9
1	5	11	15	19	26	37	48	59	61		

Quicksort Analysis

- 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? $O(n)$

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].

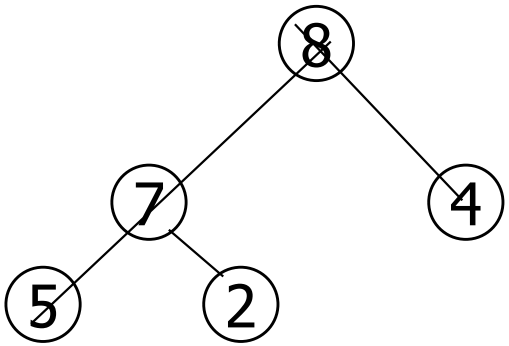
Use this median value as pivot.

The Heap Data Structure

Def: A heap is a nearly complete binary tree with the following two properties:

- Structural property: all levels are full, except possibly the last one, which is filled from left to right
- Order (heap) property: for any node x

$$\text{Parent}(x) \geq x$$



Heap

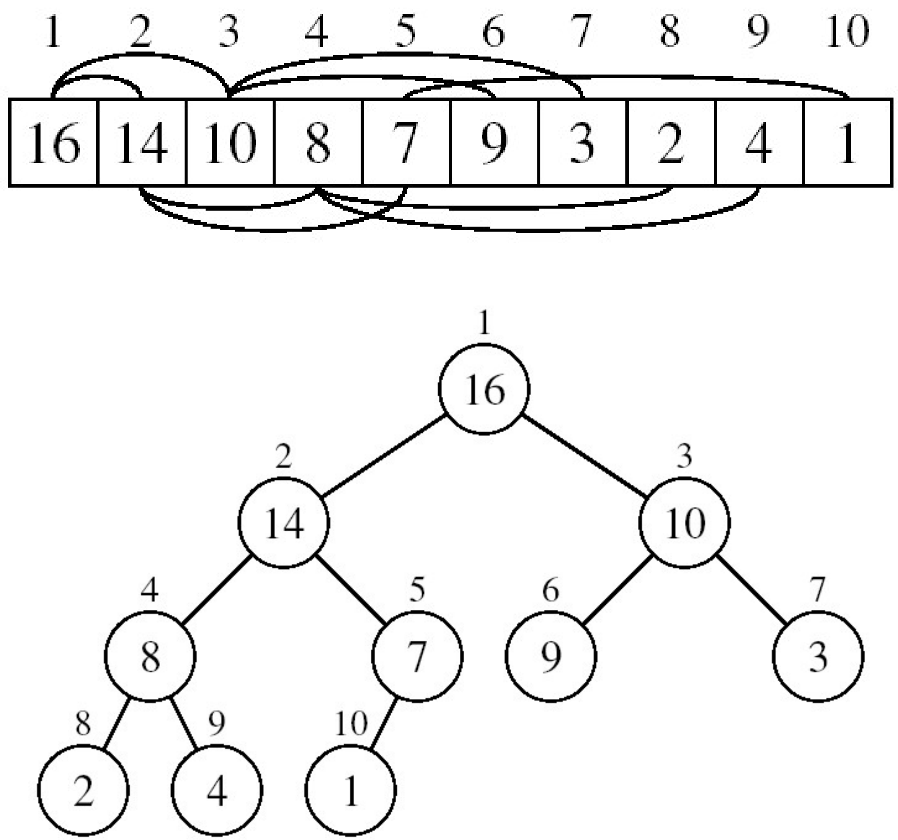
From the heap property, it follows that:

"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is A[1]
  - Left child of A[i] = A[2*i]
  - Right child of A[i] = A[2*i + 1]
  - Parent of A[i] = A[$\lfloor i/2 \rfloor$]
  - Heapsize[A] ≤ length[A]
- The elements in the subarray A[($\lfloor n/2 \rfloor$+1) .. n] are leaves



Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

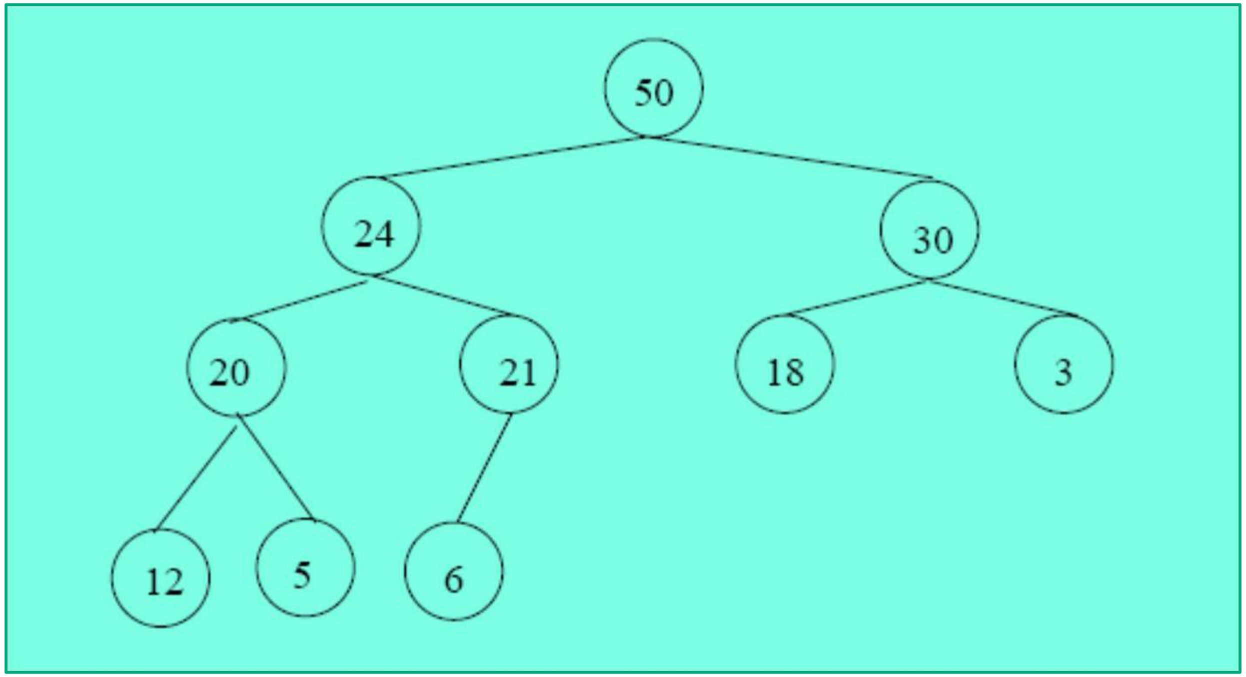
    A[PARENT(i)] ≥ A[i]

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

    A[PARENT(i)] ≤ A[i]

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

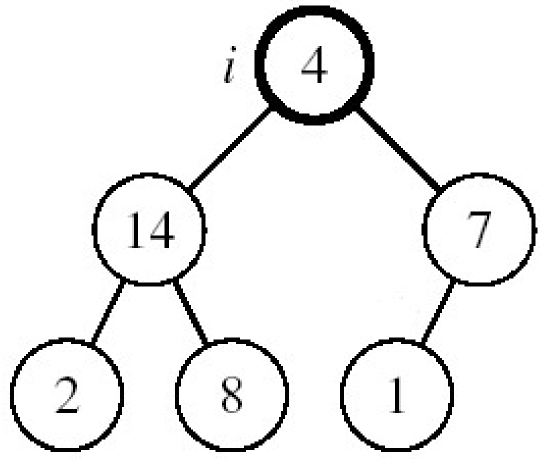


Operations on Heaps

- Maintain/Restore the max-heap property
  - Max-Heapify
- Create a max-heap from an unordered array
  - Build-Max-Heap
- Sort an array in place
  - Heapsort
- Priority queues

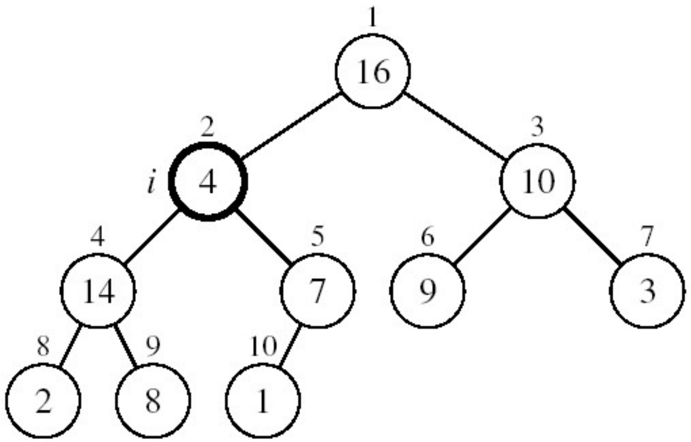
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children

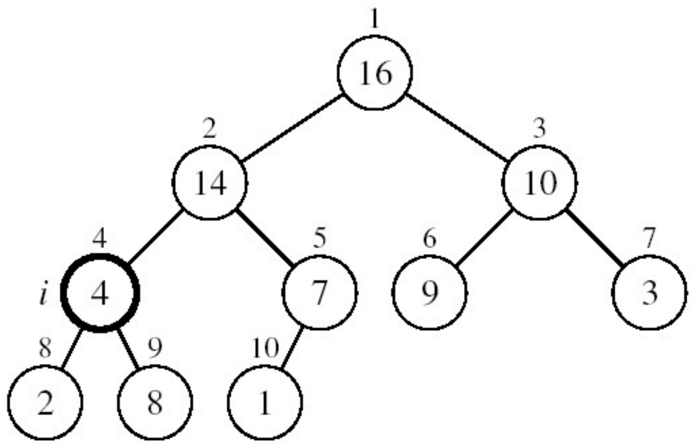


Example

Max-Heapify(A, 2, 10)



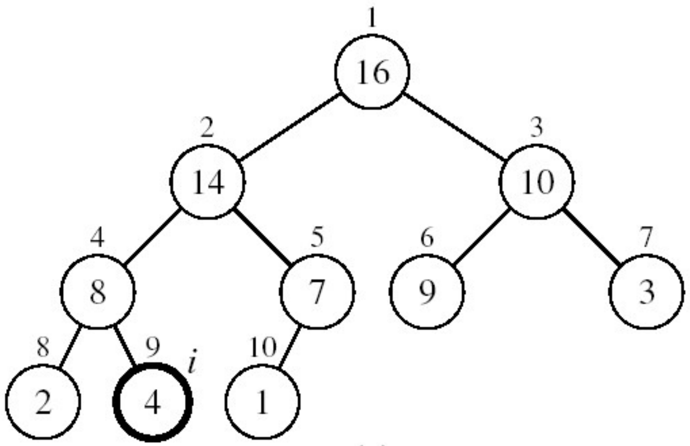
A[2] $\leftrightarrow$ A[4]



A[2] violates the heap property

A[4] violates the heap property

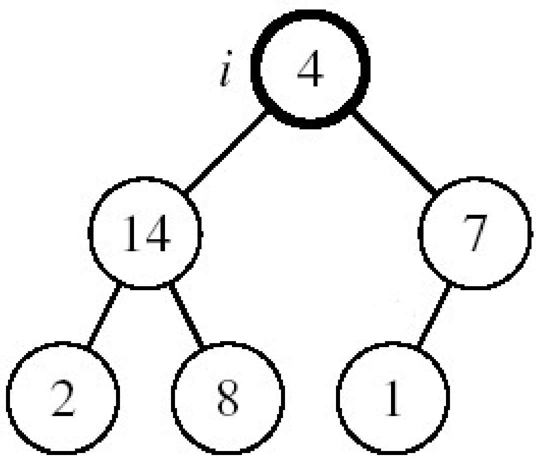
A[4] $\leftrightarrow$ A[9]



Heap property restored

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - A[i] may be smaller than its children



Alg: <u>Max-Heapify(A, i, n)</u>

```
l ← left(i)
r ← right(i)
if l ≤ n and A[l] > A[i]
   then largest ←l
   else largest ←i
if r ≤ n and A[r] > A[largest]
   then largest ←r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
         Max-Heapify(A, largest, n)
```

Max-Heapify Running Time

Intuitively:

- It traces a path from the root to a leaf (longest path length: h)
- At each level, it makes exactly two comparisons.
- Total number of comparisons is 2h
- Running time is O(h) or $O(\log_2 n)$

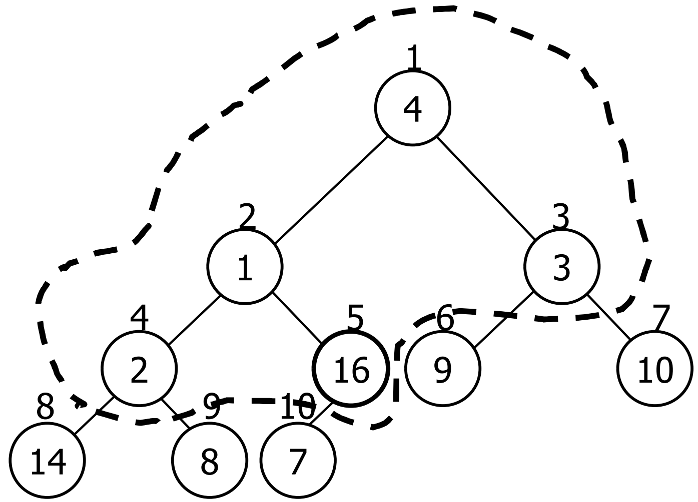
- Running time of Max-Heapify is $O(\log_2 n)$
- Can be written in terms of the height of the heap, as being O(h)
  - Since the height of the heap is $\lfloor \log_2 n \rfloor$

Building a Heap

- Convert an array A[1 … n] into a max-heap (n = length[A])
- The elements in the subarray A[($\lfloor n/2 \rfloor$+1) .. n] are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: <u>Build-Max-Heap(A)</u>

1. n = length[A]
2. **for** i ← $\lfloor n/2 \rfloor$ **downto** 1
3. **do** Max-Heapify(A, i, n)

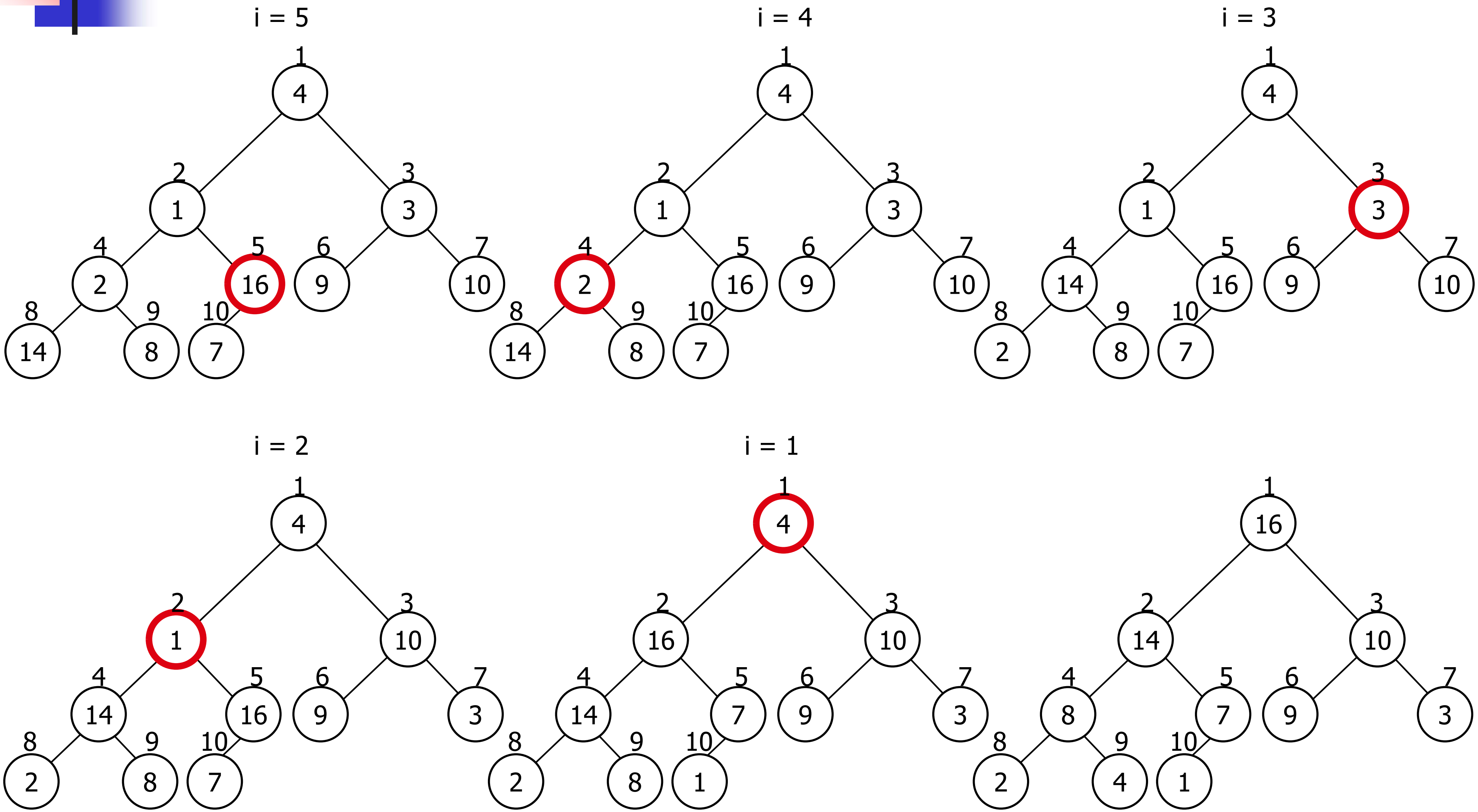


A:

4	1	3	2	16	9	10	14	8	7

Example: A

4	1	3	2	16	9	10	14	8	7



Running Time of Build-Max-Heap

Alg: <u>Build-Max-Heap(A)</u>

1. n = length[A]
2. **for** i ← $\lfloor n/2 \rfloor$ **downto** 1
3. **do** Max-Heapify(A, i, n)

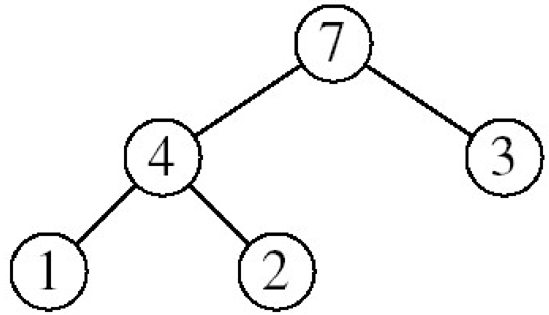
$O(\log_2 n)$ $\Big\}$ $O(n)$

⇒ Running time: $O(n\log_2 n)$

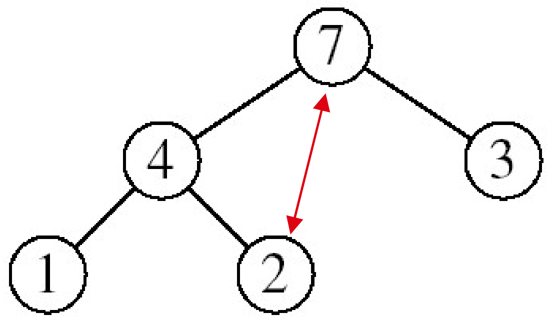
- This is not an asymptotically tight upper bound

Heapsort

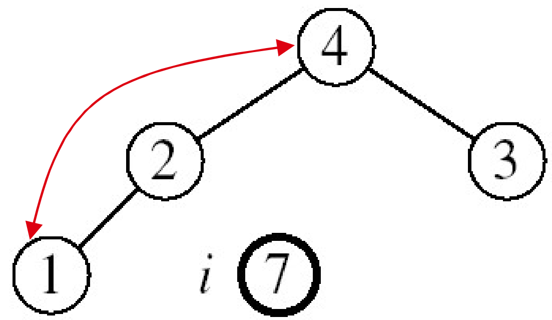
- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call Max-Heapify on the new root
  - Repeat this process until only one node remains



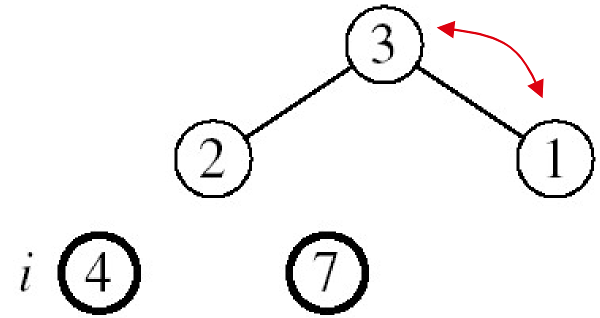
Example: A=[7, 4, 3, 1, 2]



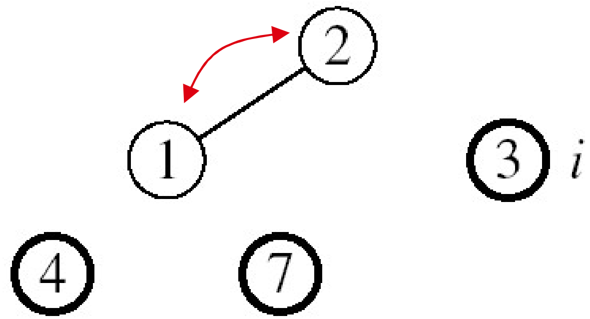
Max-Heapify(A, 1, 4)



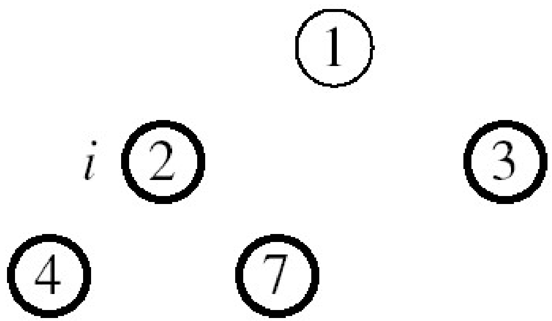
Max-Heapify(A, 1, 3)



Max-Heapify(A, 1, 2)



Max-Heapify(A, 1, 1)



A					
	1	2	3	4	7

Alg: Heapsort(A)

Build-Max-Heap(A)

for i $\leftarrow$ length[A] **downto** 2 $\quad O(n)$

  do exchange A[1] $\leftrightarrow$ A[i] $\quad$ n-1 times

    Max-Heapify(A, 1, i - 1) $\quad O(\log_2 n)$

- Running time: $O(n\log_2 n)$ --- Can be shown to be $O(n\log_2 n)$

Radix sort

- If integers are in a larger range then do bucket sort on each digit
- Start by sorting with the low-order digit using a STABLE bucket sort.
- Then, do the next-lowest, and so on

Radix Sort

- Extra information: every integer can be represented by at most k digits
  - $d_1 d_2 \ldots d_k$ where d_i are digits in base r
  - d_1: most significant digit
  - d_k: least significant digit

Radix Sort

- Algorithm
  - sort by the least significant digit first (counting sort)

    => Numbers with the same digit go to same bin
  - reorder all the numbers: the numbers in bin 0 precede the numbers in bin 1, which precede the numbers in bin 2, and so on
  - sort by the next least significant digit
  - continue this process until the numbers have been sorted on all k digits

Radix sort characteristics

- Each sorting step can be performed via bucket sort, and is thus $O(N)$.
- If the numbers are all b bits long, then there are b sorting steps.
- Hence, radix sort is $O(bN)$.

Radix sort

```
// Function to get the largest element from an array
int getMax(int arr[], int n) {
   int max = arr[0];
   for (int i = 1; i < n; i++)
      if (arr[i] > max)
         max = arr[i];
   return max;
}
```

Radix sort

```
// Counting sort
void countingSort(int arr[], int n, int place) {
    int i;
    int output[n + 1];
    int count[10] = {0};

    // Calculate count of elements
    for(i=0; i<n; i++)
        count[(arr[i] / place) % 10]++;

    // Calculate cummulative count
    for(i=1; i<10; i++)
        count[i] += count[i-1];

    // Place the elements in sorted order
    for(i=n-1; i>=0; i--) {
        output[count[(arr[i] / place) % 10] - 1] = arr[i];
        count[(arr[i] / place) % 10]--;
    }

    for(i=0; i<n; i++)
        arr[i] = output[i];
}
```

Radix sort

```
// Main function to implement radix sort
void radixsort(int arr[], int n) {
   int max = getMax(arr, n);   // Get maximum element
   // Apply counting sort to sort elements based on place value.
   for (int place=1; max / place > 0; place *= 10)
      countingSort(arr, n, place);
}

// Print an array
void printArray(int arr[], int n) {
  for (int i=0; i<n; ++i)
    printf("%d   ", arr[i]);
  printf("\n");
}
```

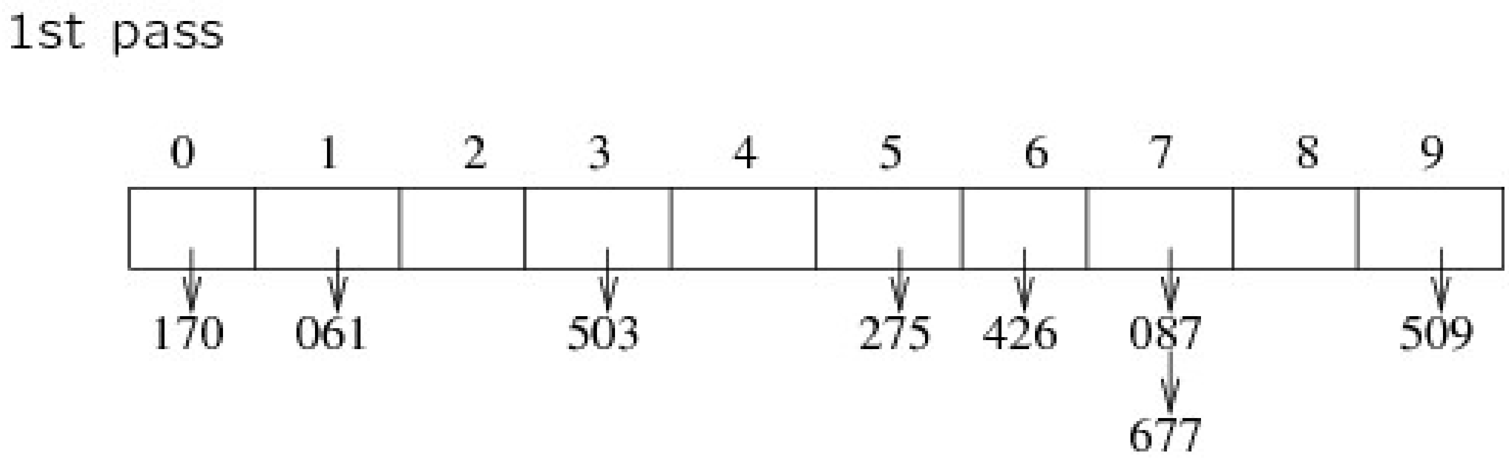
Radix sort

```
// Code
int main() {
  int arr[] = {121, 432, 564, 23, 1, 45, 788};
  int n = sizeof(arr) / sizeof(arr[0]);
  radixsort(arr, n);
  printArray(arr, n);
}
```

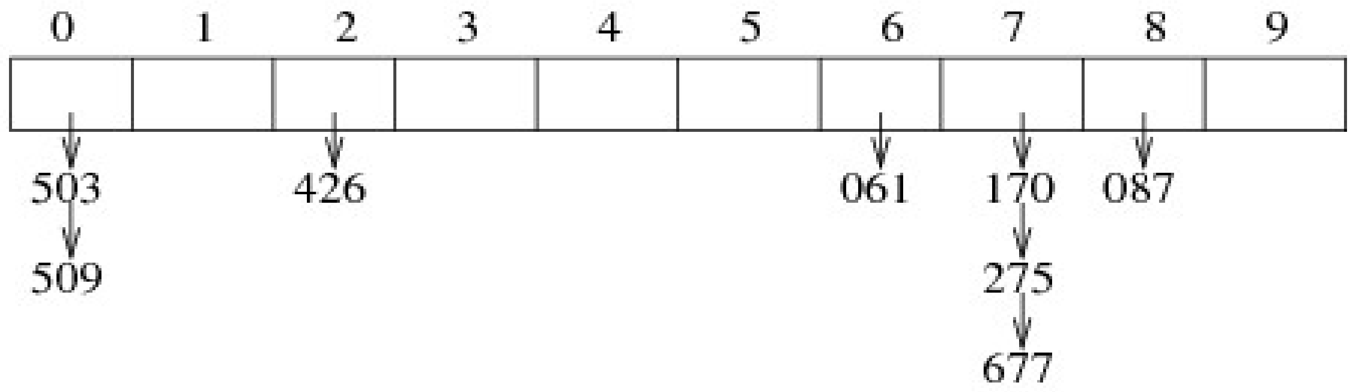
Radix Sort

- Least-significant-digit-first

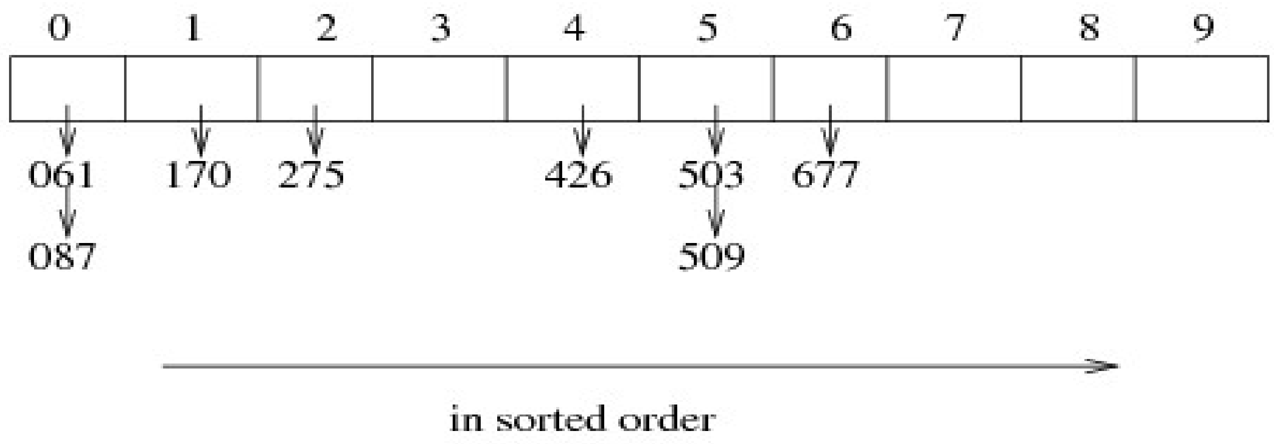
Example: 275, 087, 426, 061, 509, 170, 677, 503



2nd pass



3rd pass



Radix Sort

- Increasing the base r decreases the number of passes
- Running time
  - k passes over the numbers (i.e. k counting sorts, with range being 0...r)
  - each pass takes $O(n+r)$
  - total: $O(nk+rk)$
  - r and k are constants: $O(n)$