Sorting Algorithms

Objectives

- Examine different sorting algorithms
- Learn in-place approaches as well as those that use auxiliary collections
- Apply previously learned topics like stacks, queues, and recursion in the sorting algorithms
- Analyze the algorithms (in terms of memory and time complexity)

Sorting Problem

- **Consider an unordered list of *n* objects that we wish to have sorted into ascending order**

7	1	8	3	5	⇒	1	3	5	7	8
F	K	W	C	P	⇒	C	F	K	P	W
obj1	obj2	obj3	obj4	obj5	⇒	?	?	?	?	?

Sorting Problem

- **Numbers and strings are inherently orderable**
  - **Numerical and alphabetical ordering is built in to Java**

- **How do we sort objects of our own classes?**
  - **Remember the Comparable interface and the compareTo() method we used for the OrderedList?**
  - **It can be used in sorting algorithms too**
  - **Make your class implement Comparable and customize the compareTo() method to suit the needs of the program**

Sorting Problem

- **Speaking of OrderedList, why can't we use that instead of learning sorting algorithms?**
  - **If memory or execution time are limited, it may not be feasible to create an OrderedList**
  - **If we don't need items being sorted automatically as they're added, then don't bother with it!**

- **If memory is very limited, use *in-place* approaches to sorting**
  - **This means no additional arrays or data structures are required to perform the sort**

Sorting Problem

- We will study the following sorting algorithms:
  - *Insertion sort* using stacks and in-place
  - *Selection sort* using queues and in-place
  - *Quick sort* using recursion

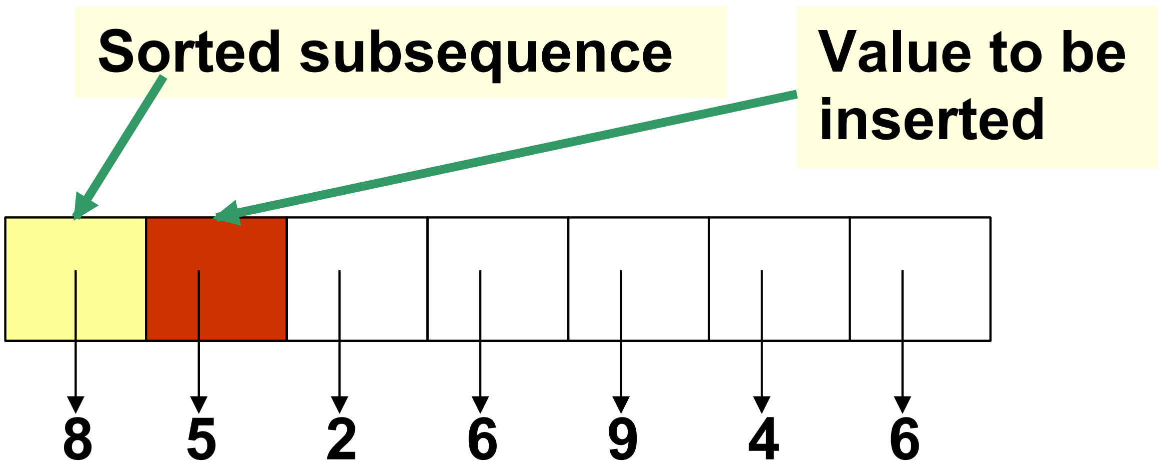
- There are many other sorting algorithms, i.e.
  - Bubble sort
  - Merge sort
  - Radix sort
  - Bucket sort

Insertion Sort

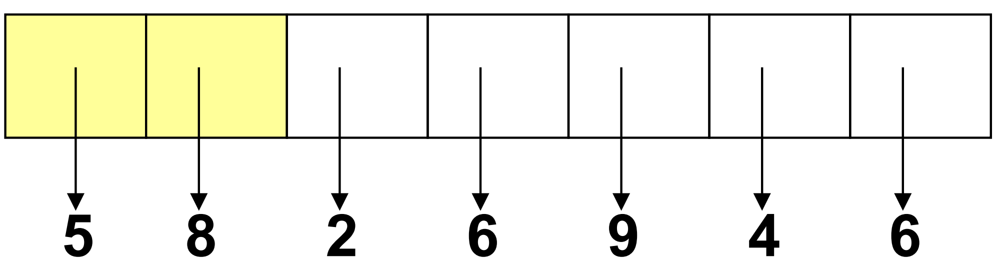
- ***Insertion Sort*** **orders a sequence of values by repeatedly taking each value and inserting it in its proper position within a *sorted subset* of the sequence.**
- **More specifically:**
  - **Consider the first item to be a *sorted subsequence* of length 1** because there's only one item.
  - **Insert the second item into the *sorted subsequence*, now of length 2**
  - **Repeat the process for each item, always inserting it into the current *sorted subsequence*, until the entire sequence is in order**

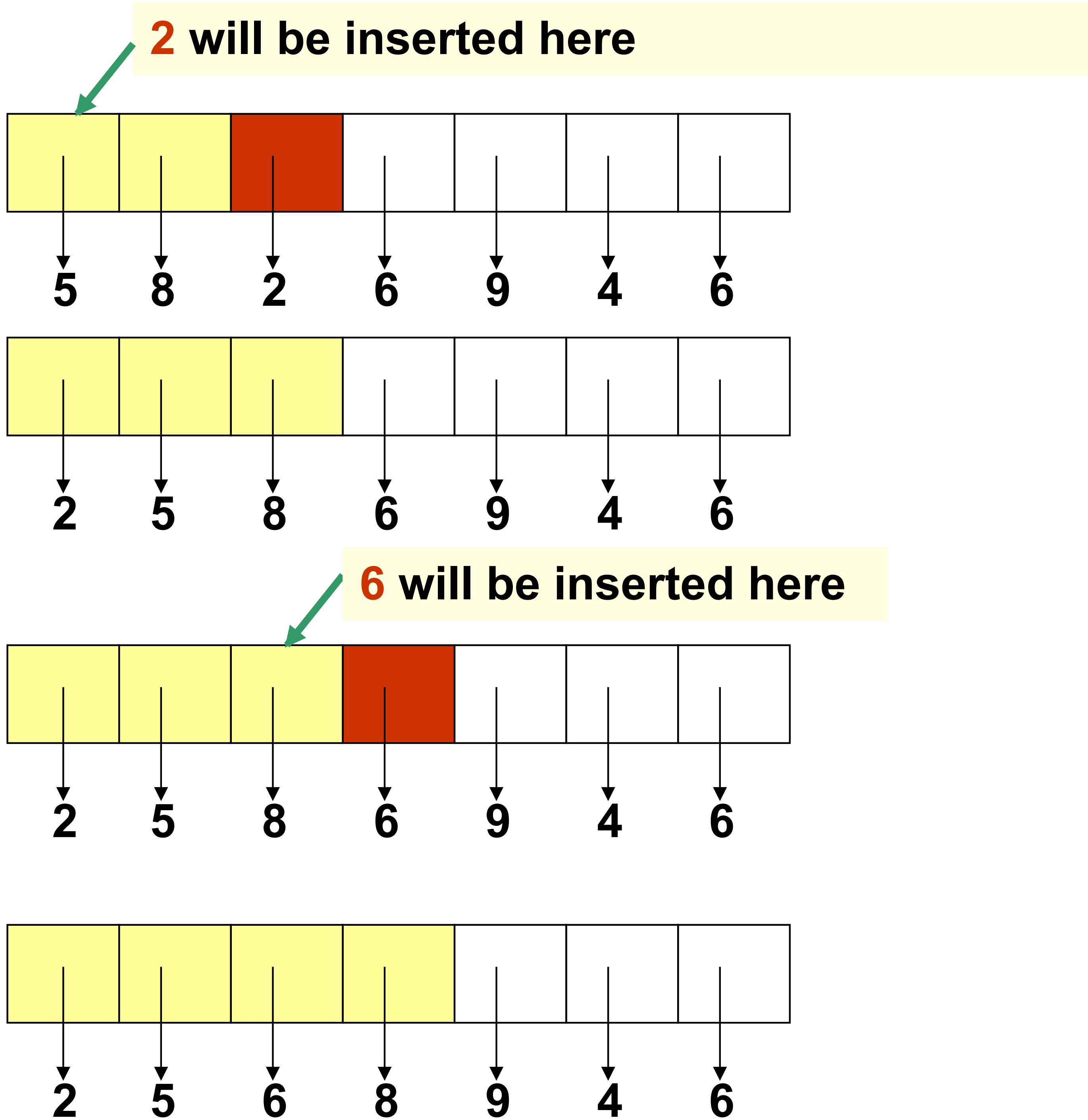
Insertion Sort Algorithm

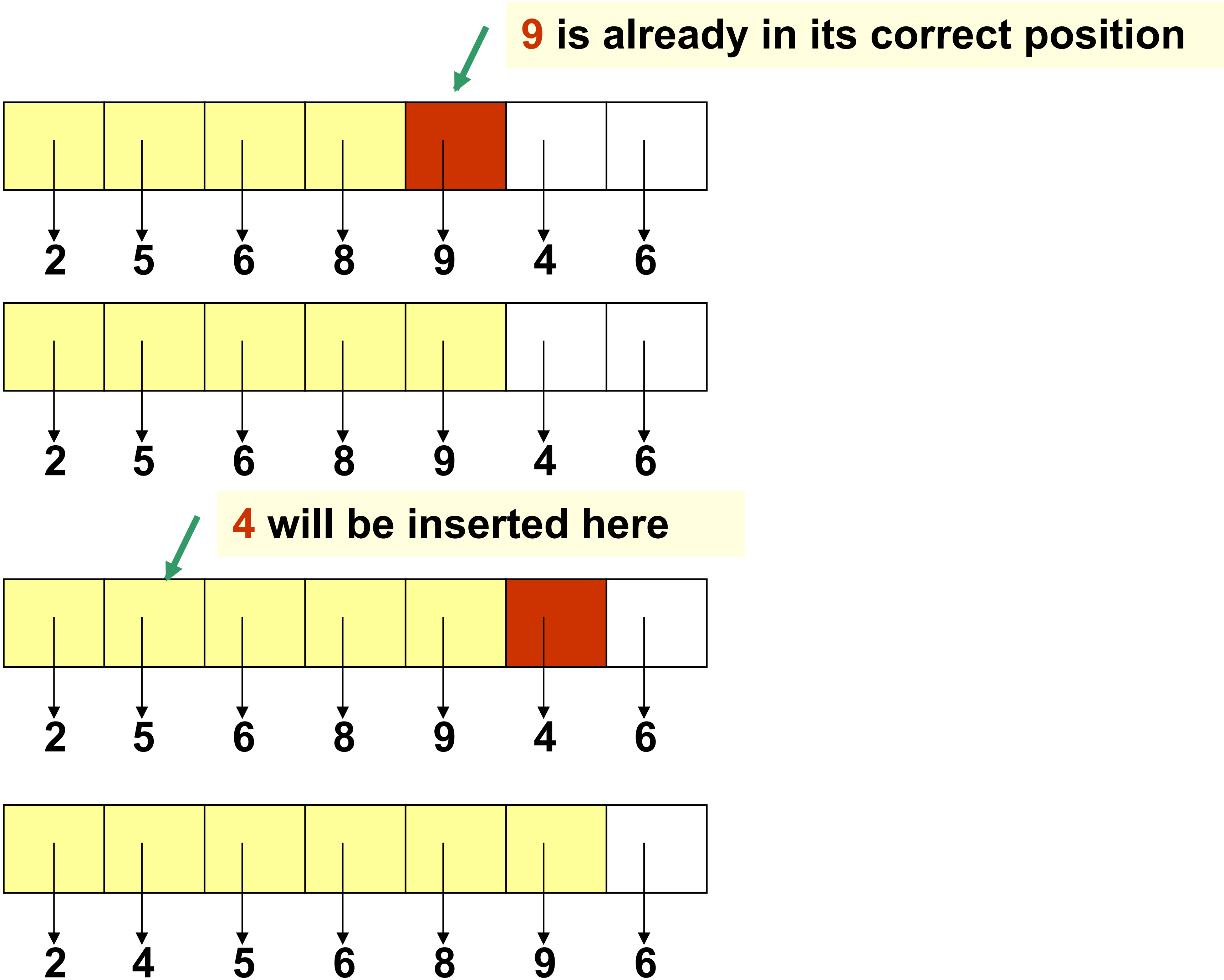
Example: **sorting a sequence of Integer objects**



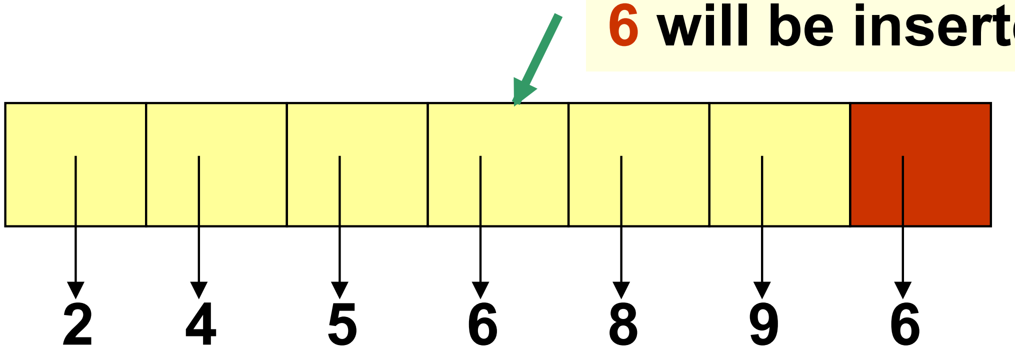
Value 5 is to be inserted in the sorted sequence to its left. Since 5 is smaller than 8, then 8 needs to be shifted one position to the right and then 5 can be inserted on the first position of the array.



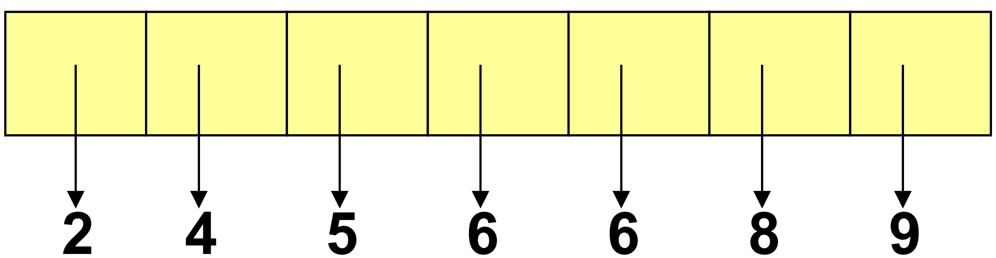

2 will be inserted here
5 8 2 6 9 4 6
2 5 8 6 9 4 6
6 will be inserted here
2 5 8 6 9 4 6
2 5 6 8 9 4 6


9 is already in its correct position
2 5 6 8 9 4 6
2 5 6 8 9 4 6
4 will be inserted here
2 5 6 8 9 4 6
2 4 5 6 8 9 6

6 will be inserted here



And we’re done!



Insertion Sort using Stacks

- Use two temporary stacks called **sorted** and **temp**, both of which are initially empty
- The contents of **sorted** will always be in order, with the **smallest** item on the top of the stack
  - This will be the "sorted subsequence"
- **temp** will temporarily hold items that need to be "shifted" out in order to insert the new item in the proper place in stack **sorted**

while (item < sorted.top) {
temp.push(sorted.top)
}
sorted. push(item)
while(! temp.isEmpty()) {

InsertionSort.

sorted.push(temp.pop())
}

Algorithm insertionSort (A,n)

In: ~~Array A storing n elements~~

Out: Sorted array

greater than sorted.peek()
the

```
sorted = empty stack
temp = empty stack
for i = 0 to n-1 do {
   while (sorted is not empty) and (sorted.peek() < A[i]) do
         temp.push (sorted.pop())
   sorted.push (A[i])
   while temp is not empty do
         sorted.push (temp.pop())
}
for i = 0 to n-1 do
   A[i] = sorted.pop()
return A
```

if stack is empty,
stack.peek() will throw an Exception.

loop through the original array.

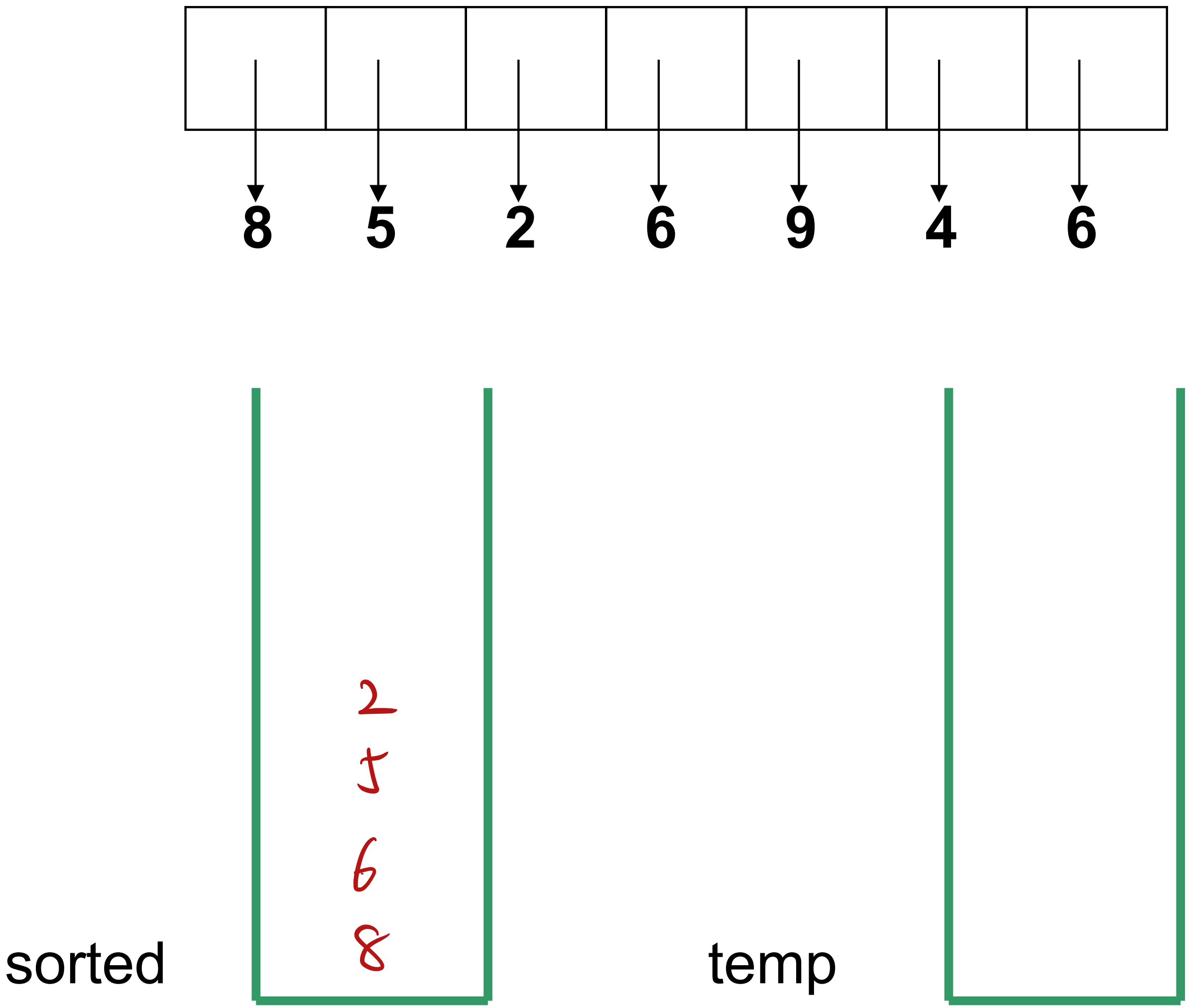
move elements smaller than A[i] from sorted to temp

A[i] becomes the smallest

return elements in temp in order.

move elements back to array.

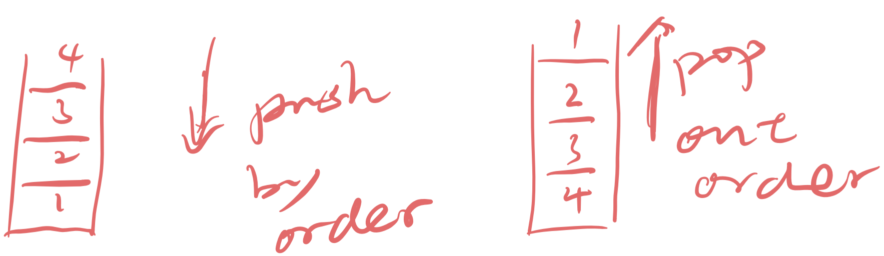
Insertion Sort

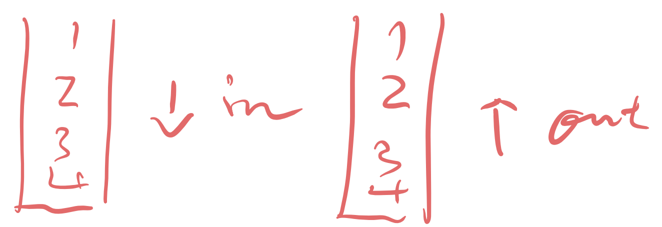


Analyzing the Stack Insertion Sort

- Let's look at 2 extreme cases that represent the best and worst cases:
  - A pre-sorted array worst
  - A reverse-sorted array best

every time adding a new element, we have to pop out all elements in the current stack:





Insertion Sort

operations.

1

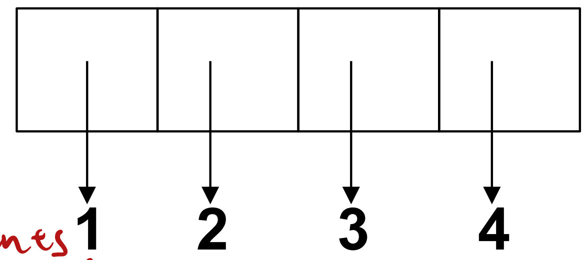
3

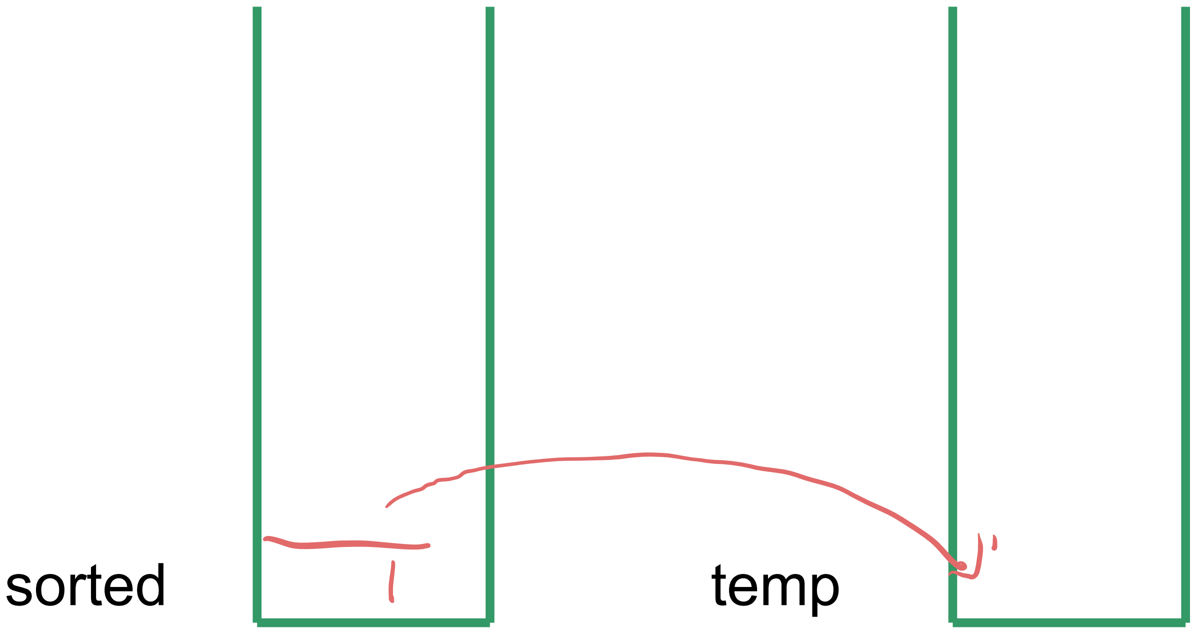
5

7

16 operations for 4 elements!

$O(n^2)$.



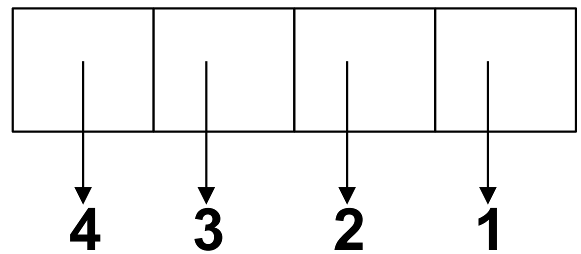


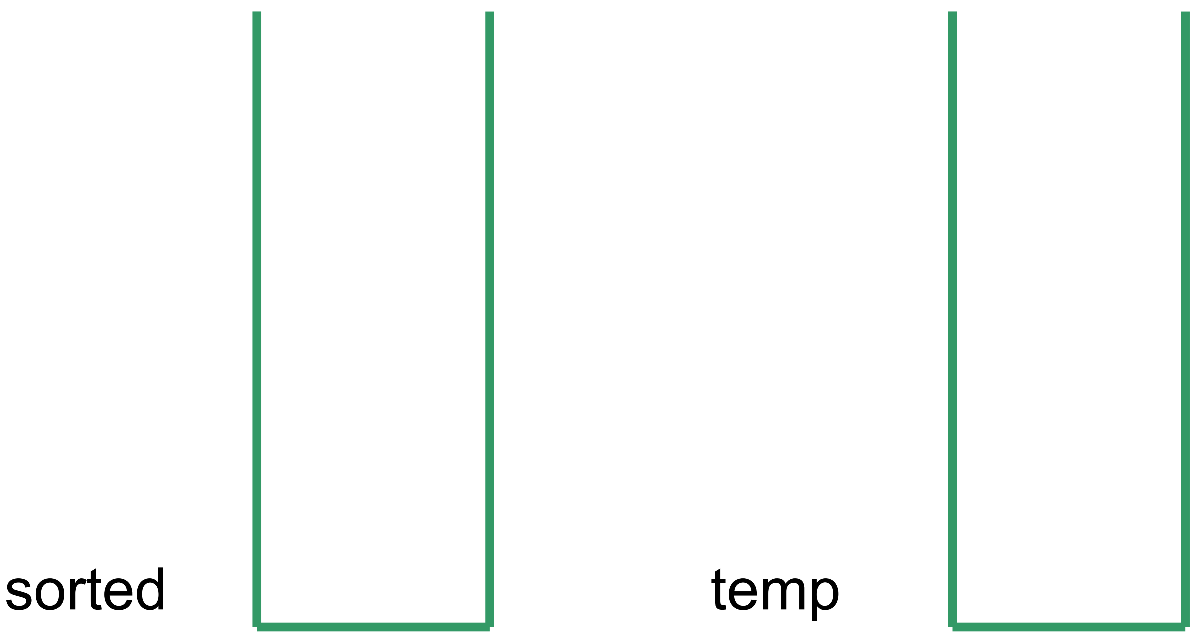
Insertion Sort

operation.

4 at all for 4 elements.

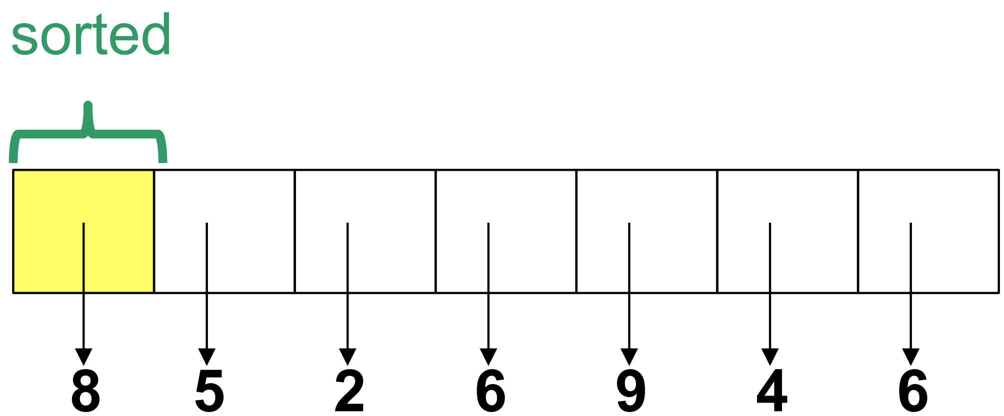
O(n).





In-Place Insertion Sort

no extra data structure.



5

Consider the next value: 5

It is not In-Place (no extra data structure is created).

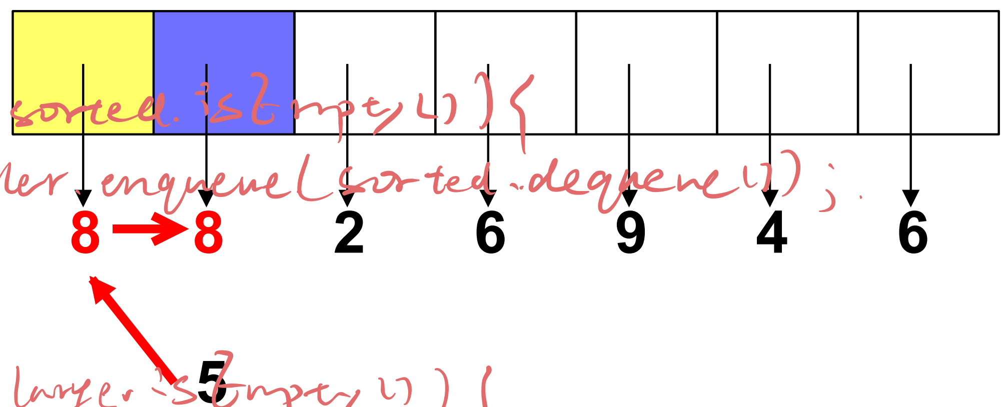
In-Place Insertion Sort

```
For (int i = 0; i < A.length-1; i++){
    if (!sorted.isEmpty()){        && (!sorted.isEmpty()).
        while(sorted.element() > A[i]){
            larger.enqueue(sorted.dequeue())
        }
    }
    while(!sorted.isEmpty()){
        smaller.enqueue(sorted.dequeue());
    }
    while(!larger.isEmpty()){
        sorted.enqueue(larger.dequeue());
    }
    sorted.enqueue(A[i]);
    while(!smaller.isEmpty()){
        sorted.enqueue(smaller.dequeue());
    }
```

1 5 6

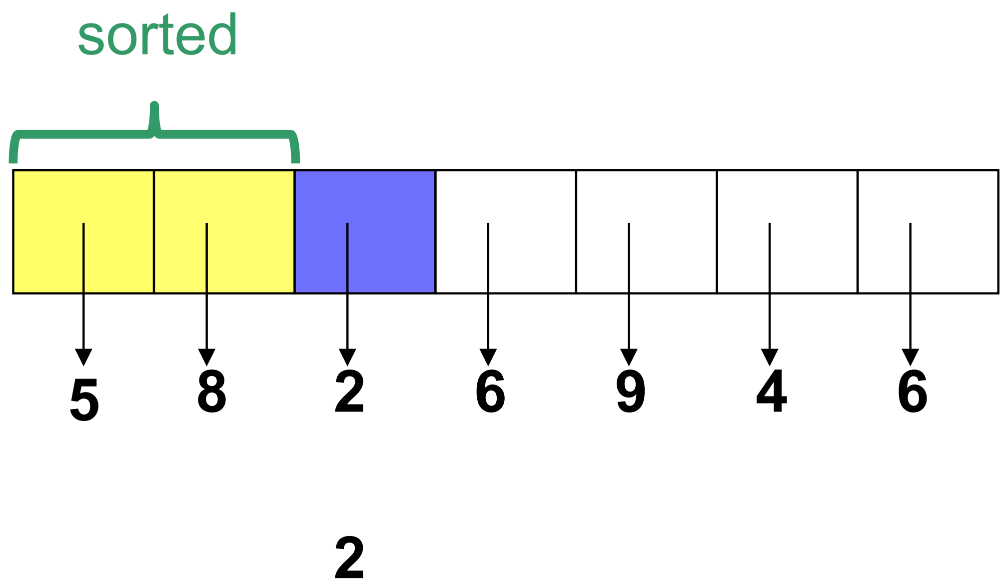
1 2 3 5 6

4



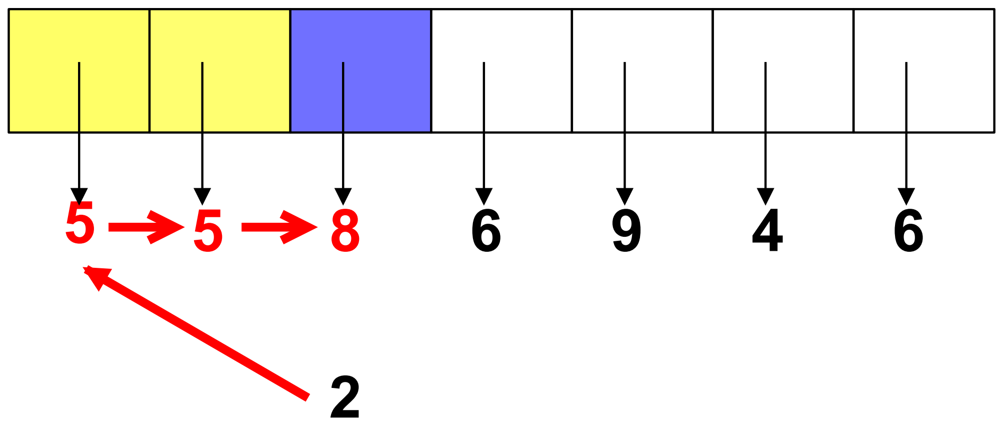
Shift 8 to make room for 5

In-Place Insertion Sort



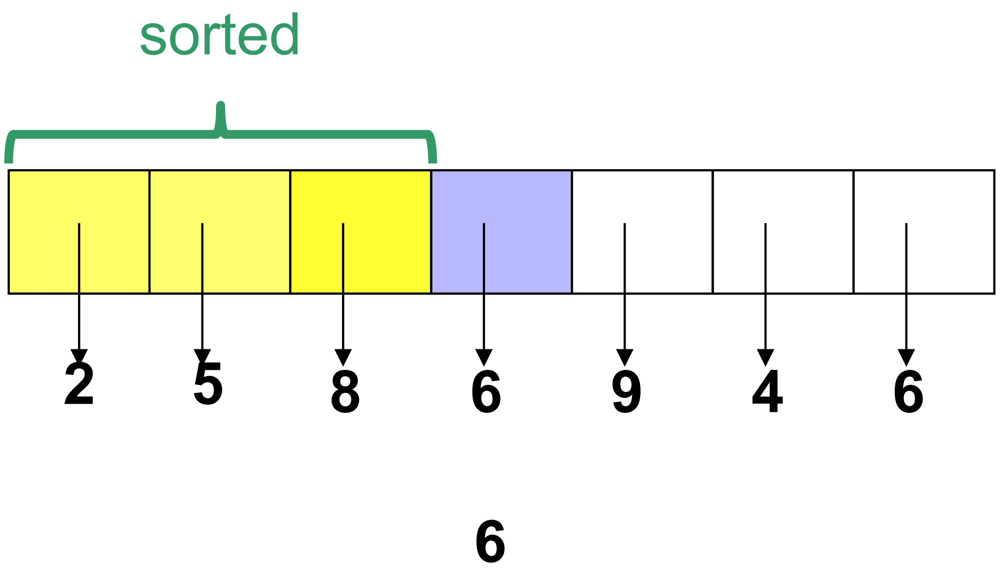
Consider the next value: 2

In-Place Insertion Sort



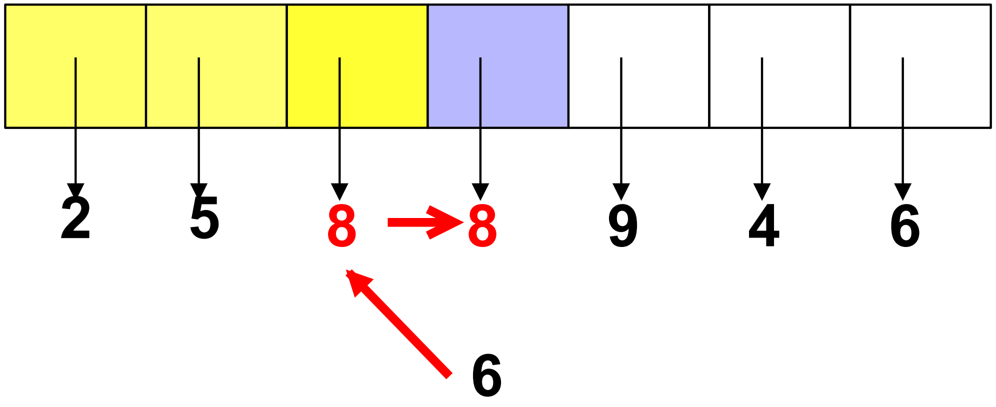
Shift 8 and 5 to the right

In-Place Insertion Sort



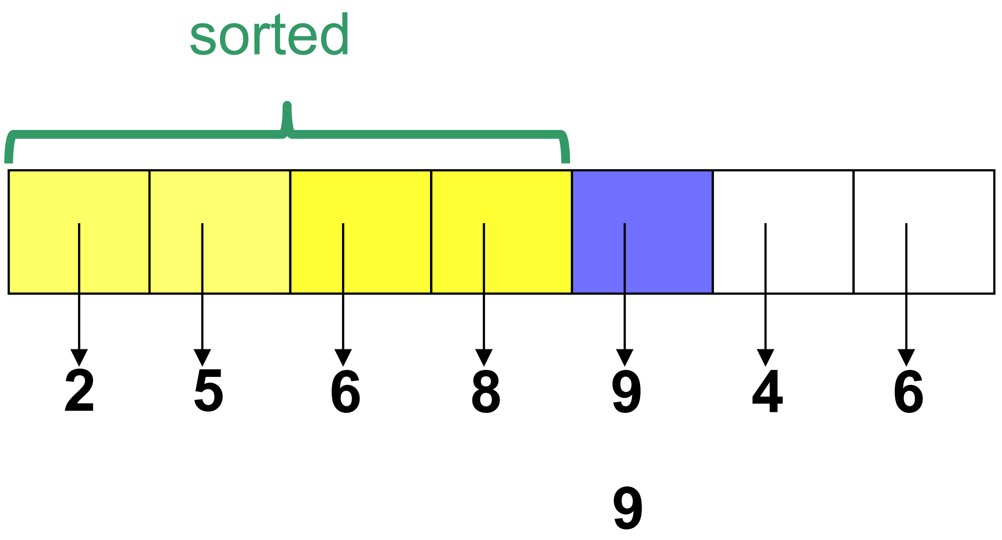
Consider the next value: 6

In-Place Insertion Sort



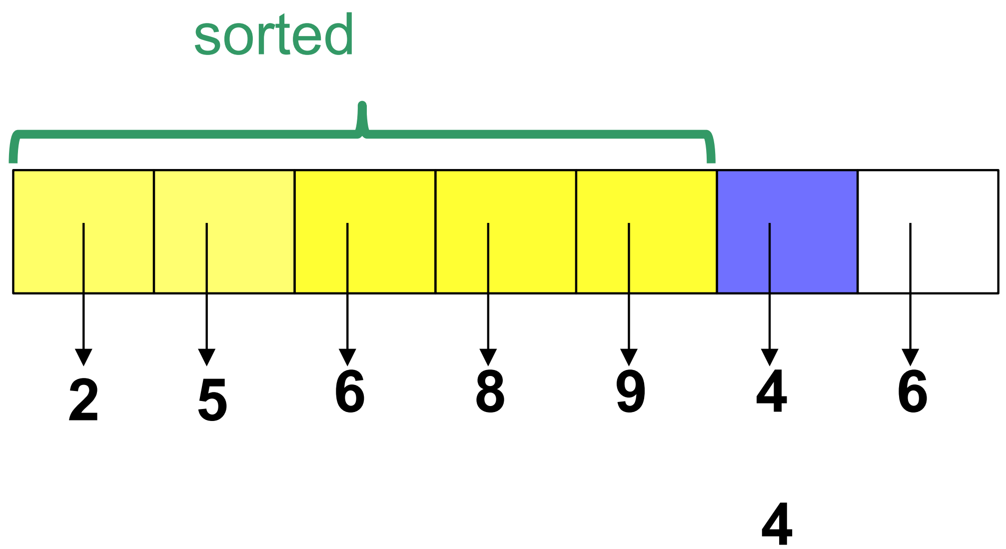
Shift 8 to the right

In-Place Insertion Sort



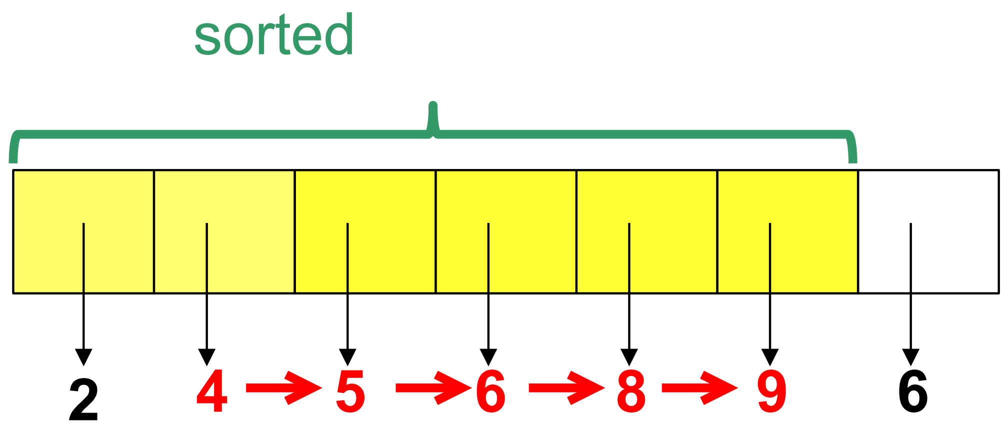
9 is already in its correct position

In-Place Insertion Sort



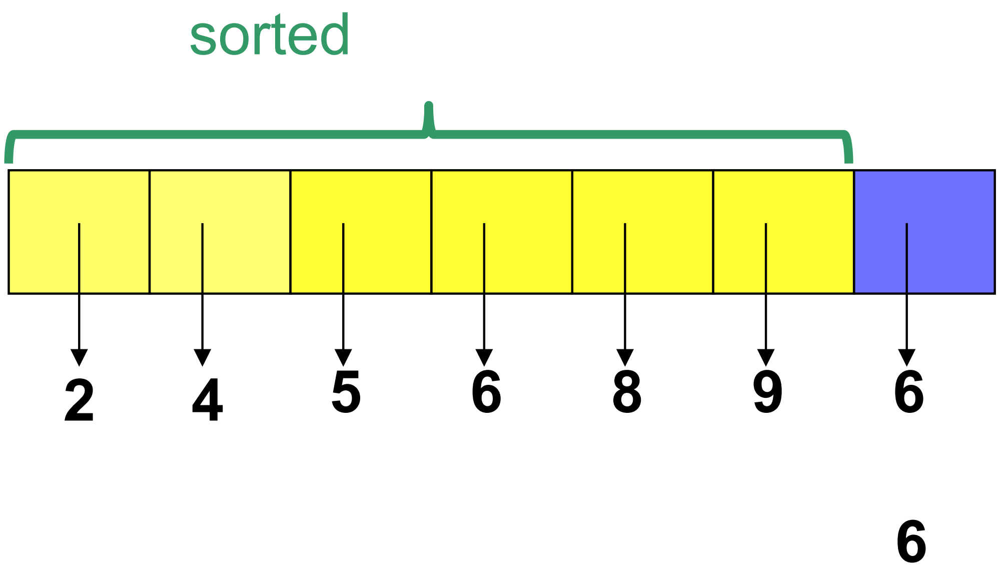
Consider the next value: 4

In-Place Insertion Sort



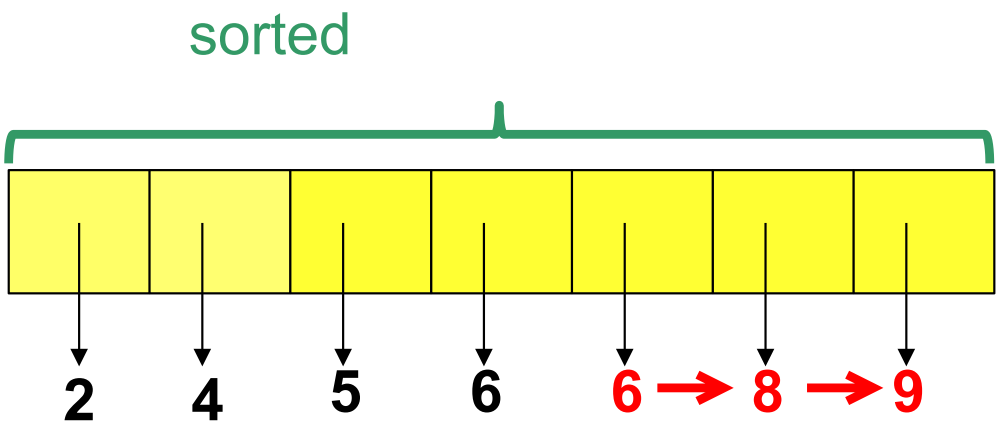
Shift 5, 6, 8, 9 to the right and
insert 4 in the second position

In-Place Insertion Sort



Finally, consider the last value: 6

In-Place Insertion Sort



Shift 8 and 9 to the right and insert 6 in the fifth position. The array is sorted!

giving an unorder Array and create a sorted array base on queue.

Algorithm *insertionSort* (*A*,*n*)

In: Array *A* storing n values

Out: {Sort *A* in increasing order}

the first element is sorted.

n-1. **for** *i* = 1 **to** *n*-1 **do** {

// Insert *A*[*i*] in the sorted sub-array *A*[0..*i*-1]

temp = *A*[*i*]

$\boxed{j = i - 1}$ counter for the subset.

n **while** (*j* >= 0) **and** (*A*[*j*] > *temp*) **do** {

A[*j*+1] = *A*[*j*]

j = *j* – 1

move backward larger elements.

}

A[*j*+1] = *temp*

put the element.

}

j=0

i=1. temp

2 5 1 3

What is the time complexity of the in-place insertion sort?

O(n²).

n·n

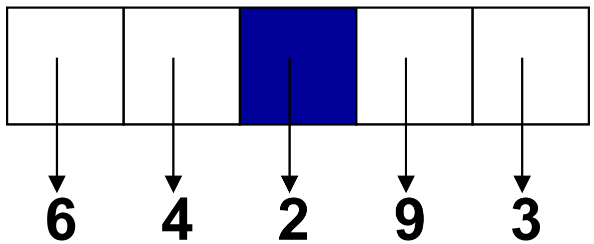
Selection Sort

- ***Selection Sort*** orders a sequence of values by repetitively putting a particular value into the *final* position of its *sorted subsequence*
- More specifically:
  - Find the smallest value in the sequence
  - Switch it with the value in the first position
  - Find the next smallest value in the sequence
  - Switch it with the value in the second position
  - Repeat until all values are in their proper places

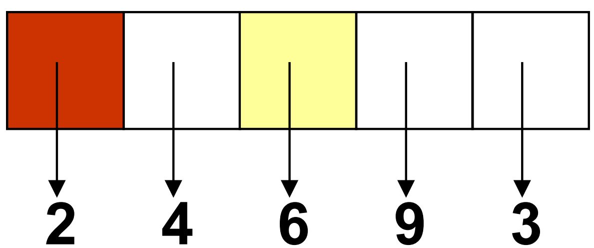
Selection Sort Algorithm

Initially, the *entire* array is the "*unsorted portion*"

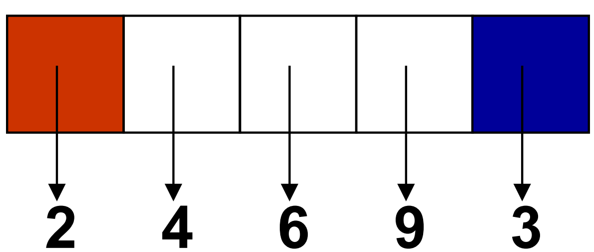
The sorted portion is in red.



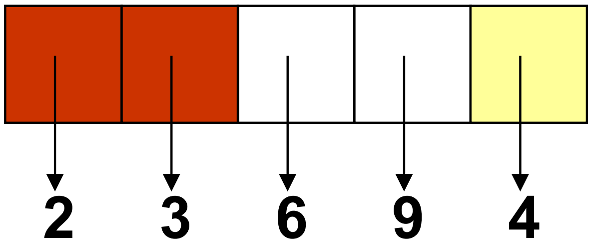
Find the smallest element in the unsorted portion of the array



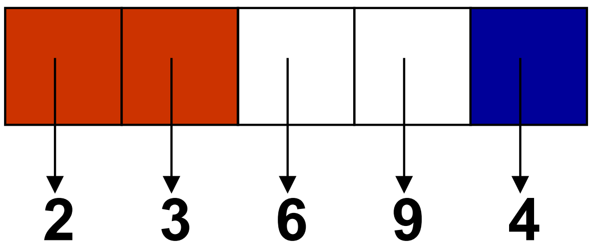
Interchange the smallest element with the one at the first position of the array



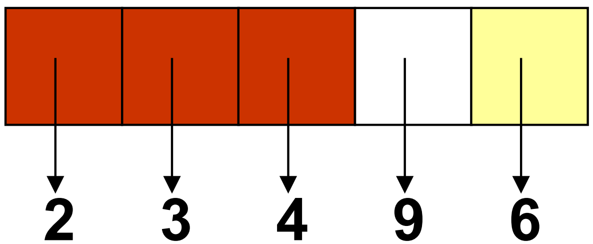
Find the smallest element in the unsorted portion of the array



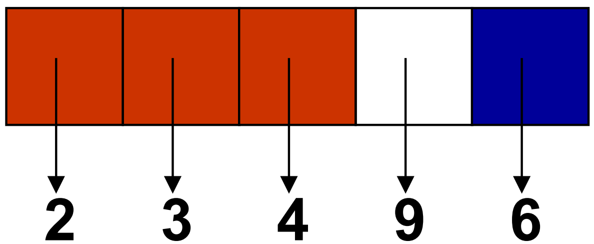
Interchange the smallest element with the one at the second position



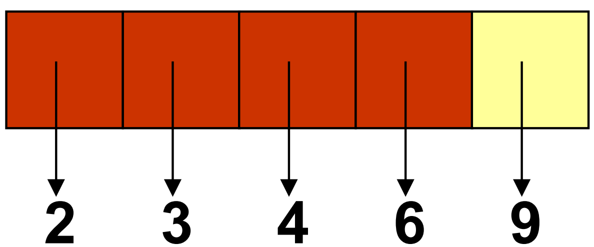
Find the smallest element in the unsorted portion



Interchange the smallest element with the one at the third position



Find the smallest element in the unsorted portion



Interchange the smallest element with the one at the fourth position

After n-1 repetitions of this process, the last item has automatically fallen into place!

Selection Sort Using a Queue

- Create a queue called sorted, initially empty, to hold the items that have been sorted *so far*
- The contents of sorted will always be in order, with new items added at the end of the queue

Selection Sort Using Queue Algorithm

- **While the unordered list list is not empty:**
  - ***remove*** **the smallest item from list and *enqueue* it to the end of sorted**
- **At the end of the while loop the list is empty, and sorted contains the items in ascending order, from front to rear**
- **To restore the original list, *dequeue* the items one at a time from sorted, and *add them to* list**

12367

Algorithm selectionSort(*list*)
In: Unsorted list
Out: Sorted list

What is the time complexity of the queue-based selection sort?

$O(n^2)$.

sorted = empty queue

_n_ = number of data items in _list_

```
while list is not empty do {
      smallestSoFar = get first item in list
      for i = 1 to n – 1 do {   find the smallest item in current list
         item = get item in the i-th position of list   A[i].
         if item < smallestSoFar then smallestSoFar = item
      }                         A[i].
      sorted.enqueue(smallestSoFar)   add to the sorted list.
      remove smallestSoFar from list
      n = n - 1
}
for i = 0 to n – 1 do   move items back to the array.
      insert sorted.dequeue() in the i-th position of list
return list
```

n

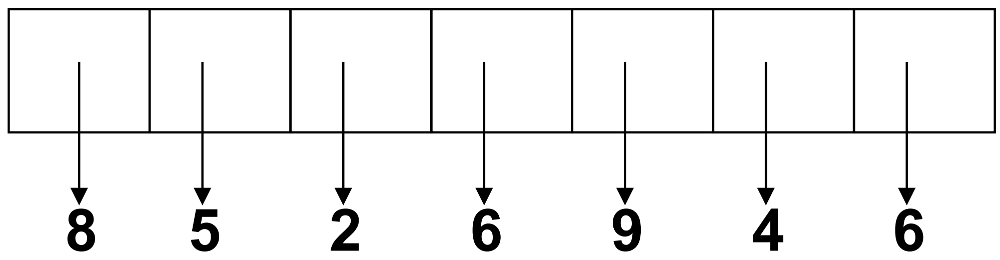
n

~~8~~	~~5~~	~~2~~	~~6~~	~~9~~	~~4~~	~~6~~

Q 2 4 5 6 6 8 9.

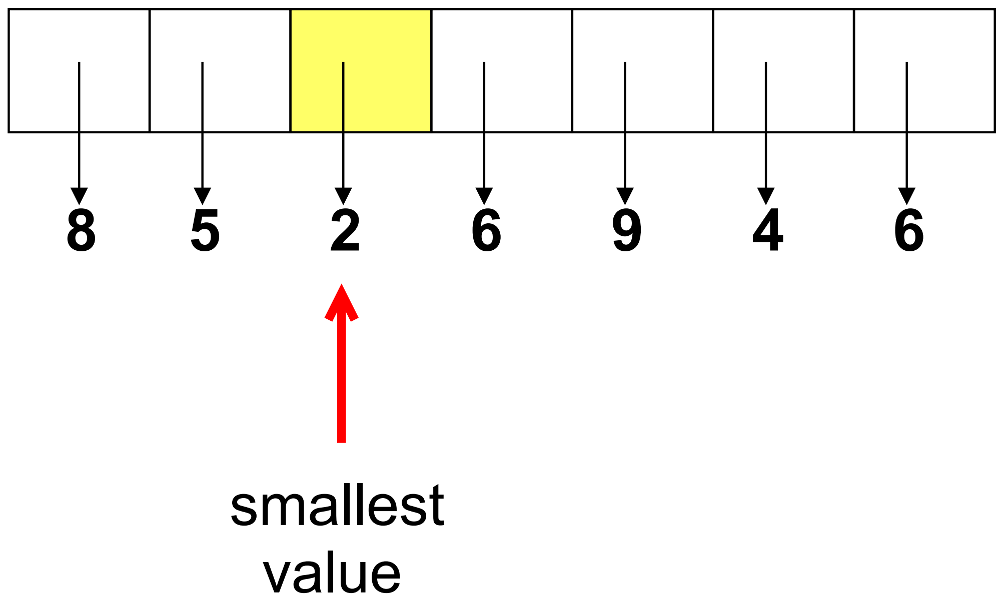
In-Place SelectionSort

Selection sort without using any additional data structures.
Assume that the values to sort are stored in an array.



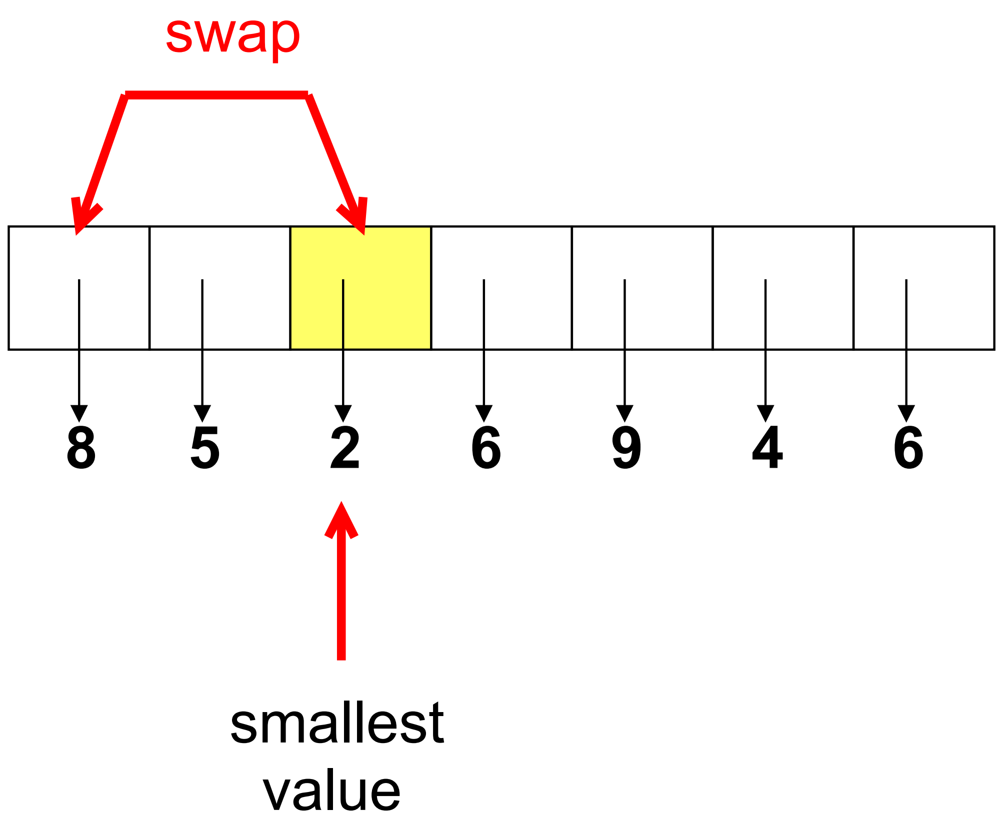
In-Place SelectionSort

First, find the smallest value



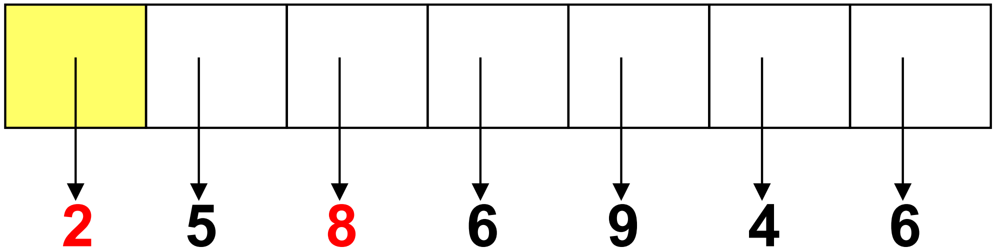
In-Place SelectionSort

Swap it with the element in the first position of the array.

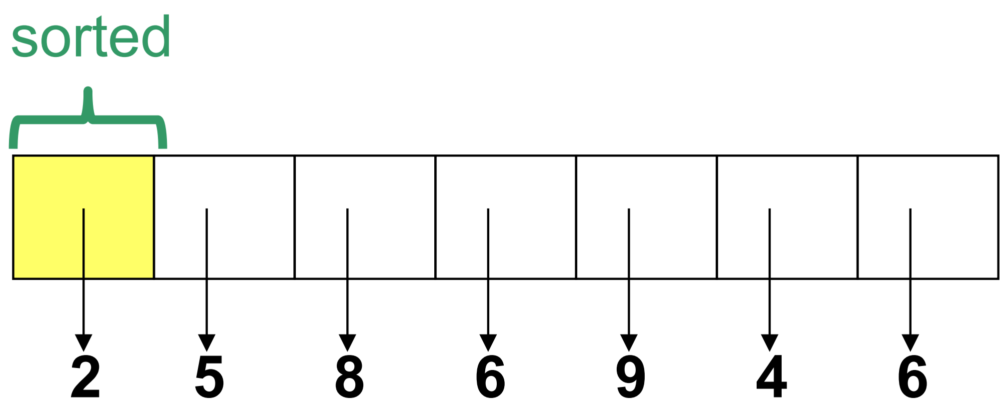


In-Place SelectionSort

Swap it with the element in the first position of the array.

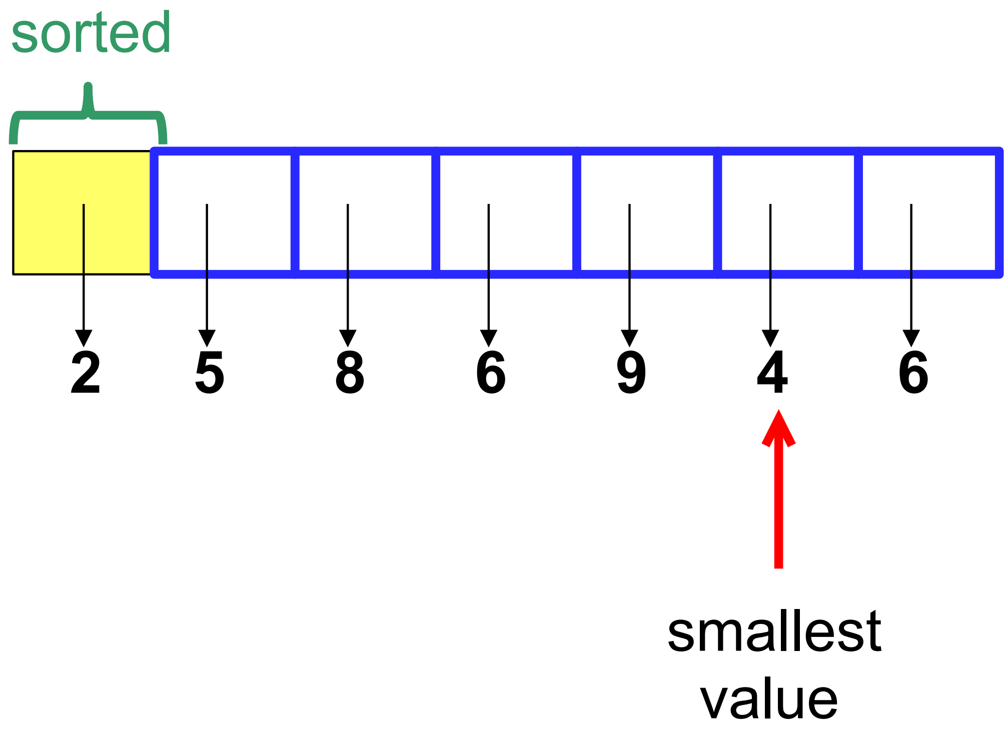


In-Place SelectionSort



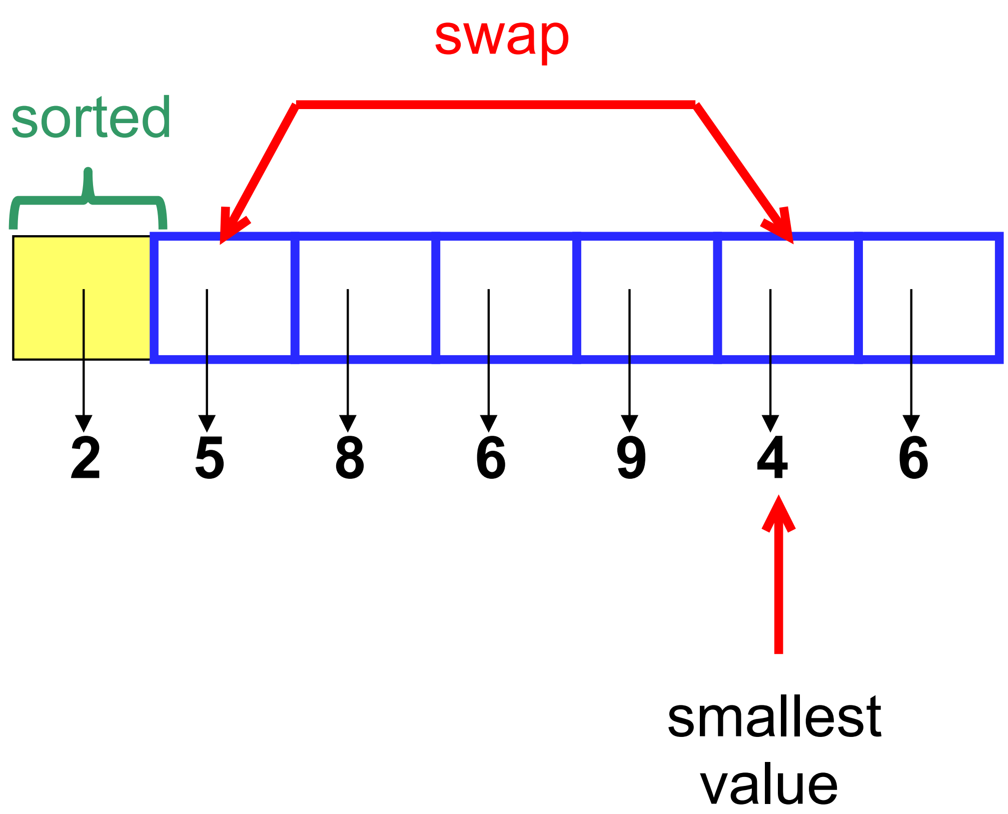
In-Place SelectionSort

Now consider the rest of the array and again find the smallest value.

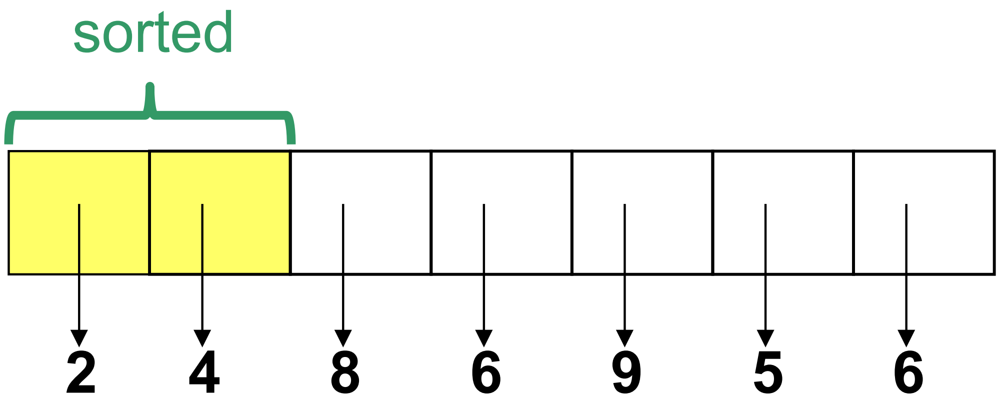


In-Place SelectionSort

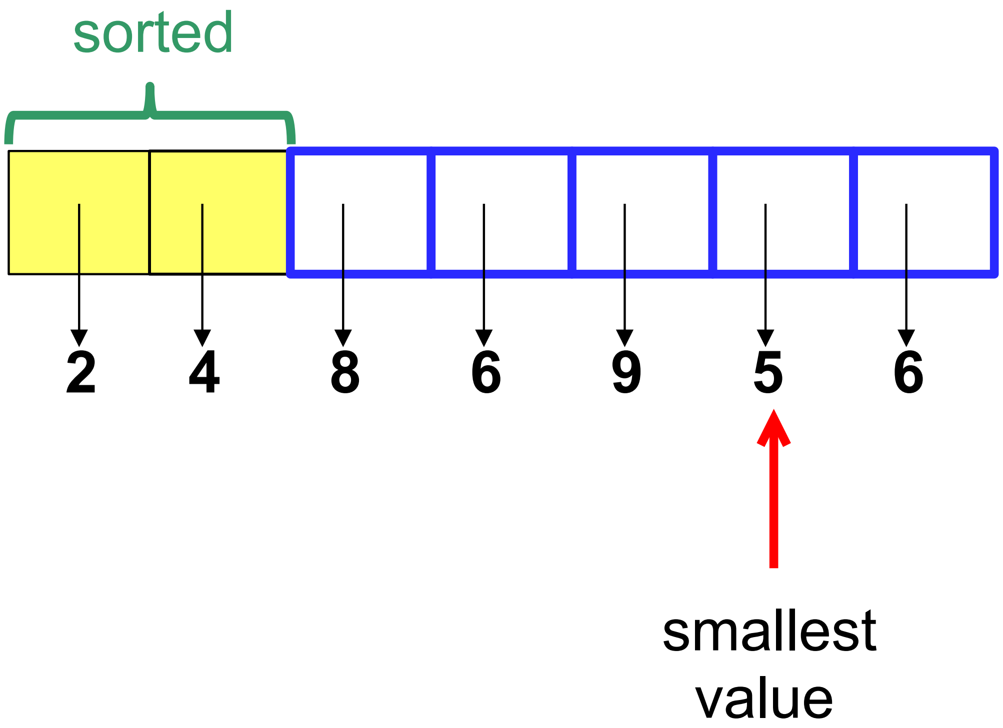
Swap it with the element in the second position of the array, and so on.



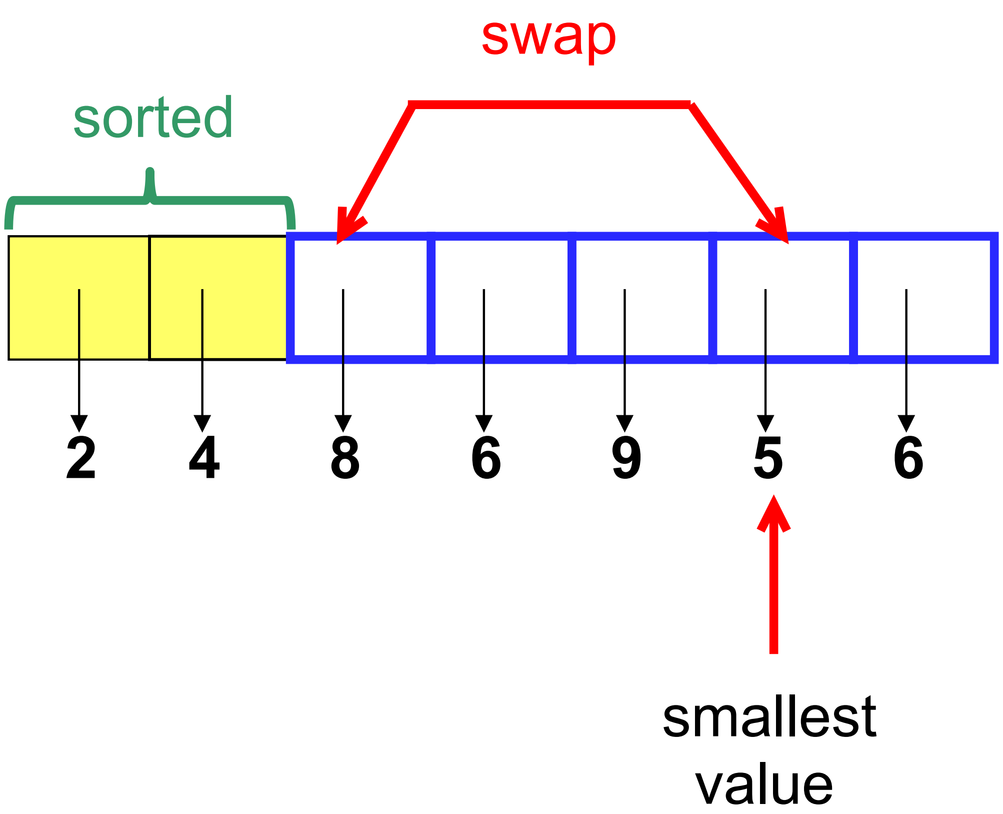
In-Place SelectionSort



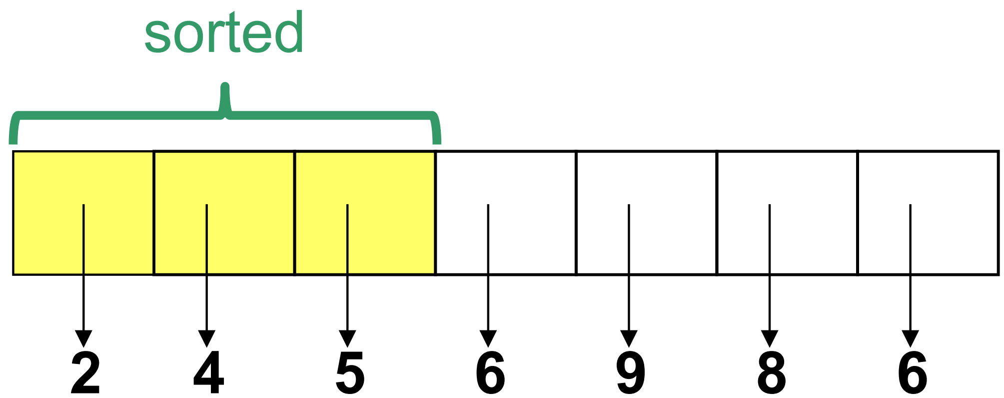
In-Place SelectionSort



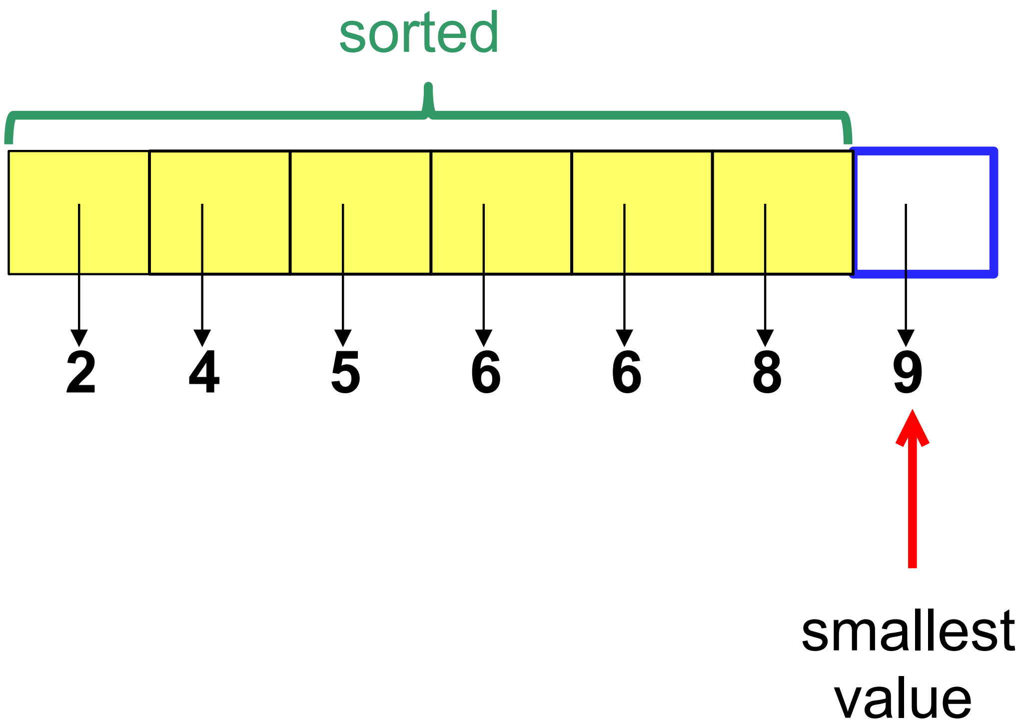
In-Place SelectionSort



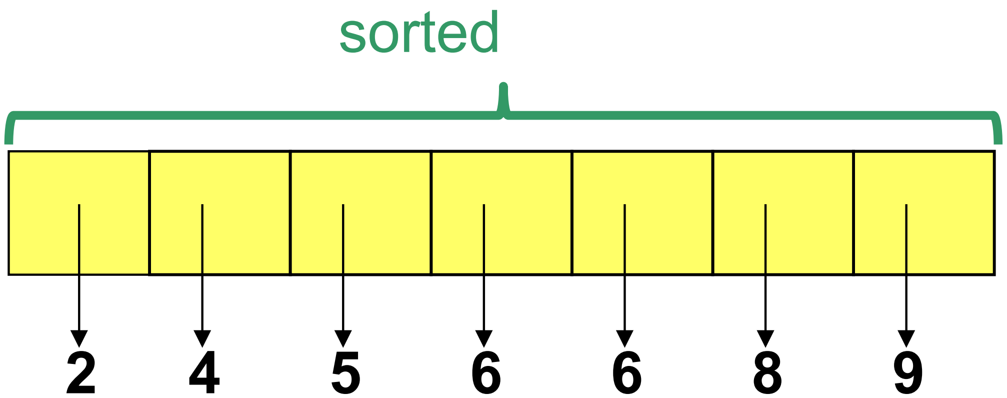
In-Place SelectionSort



In-Place SelectionSort



In-Place SelectionSort



~~for(int i=0; i< A.length; i++){~~

~~int Smallest = A[0];~~

~~int Sublength=0;~~

~~for (int j=0; j< sublength; j++){~~

~~if A[i] < A[j]{ //if curr < sortedSubsequenceElement.~~

~~temp=A[i]; //switch position.~~

~~A[i]=A[j];~~

~~A[j]=temp~~

~~}~~

~~sublength++;~~

~~}~~

ordered.

A[i].

|0 1 4 |2 3 8 7 5

sublength:3.

A[i]: curr.

A[j] loop the ordered sublist.

⇒

A[i]

0 2 4 1 3 8 6 5 7.

i=3.

0 1 4 2 3

0 1 2 4

sc=0.

sc=1.

sc=2.

```
for(int i=0; i< A.length; i++){
    int subCount = sc = 0;
    while (sc< i){
        if A[i] < A[sc] {
            Switch(A[i], A[sc]);
        }
        sc++.
    }
}
```

Algorithm ***selectionSort*** **(*A*,*n*)**

In: Array *A* storing *n* values

Out: {Sort *A* in increasing order}

```
for i = 0 to n-2 do {
   // Find the smallest value in unsorted subarray A[i..n-1]
   smallest = i
   for j = i + 1 to n - 1 do {
      if A[j] < A[smallest] then
         smallest = j
   }
   // Swap A[smallest] and A[i]
   temp = A[smallest]
   A[smallest] = A[i]
   A[i] = temp
}
```

n

n

What is the time complexity of the in-place selection sort?

$O(n^2)$.

Quick Sort

- ***Quick Sort*** **orders a sequence of values by *partitioning* the list around one element (called the *pivot* or *partition element*), then sorting each partition**
- **More specifically:**
  - **Choose one element in the sequence to be the pivot**
  - **Organize the remaining elements into three groups (*partitions*): those *greater than* the pivot, those *less than* the pivot, and those *equal* to the pivot**
  - **Then sort each of the first two partitions (recursively)**

Quick Sort

Partition element **or** ***pivot*****:**

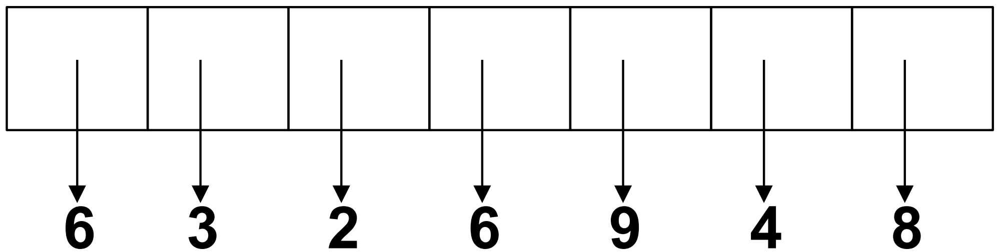
- **The choice of the pivot is arbitrary**
- **For efficiency, it would be nice if the pivot divided the sequence roughly in half**
  - **However, the algorithm will work in any case**

Quick Sort

- Put all the items to be sorted into a container (e.g. an array)
- We will arbitrarily choose the pivot (partition element) as the first element from the container
- Use a container called smaller to hold the items that are smaller than the pivot, a container called larger to hold the items that are larger than the pivot, and a container called equal to hold the items of the same value as the pivot
- We then *recursively* sort the items in the containers smaller and larger
- Finally, copy the elements from smaller back to the original container, followed by the elements from equal, and finally the ones from larger

```
public int[] QuickSort(int[] A){
    if A.length%2==0  pivot = A.length%2.
    else pivot=(A.length-1.)%2.
```

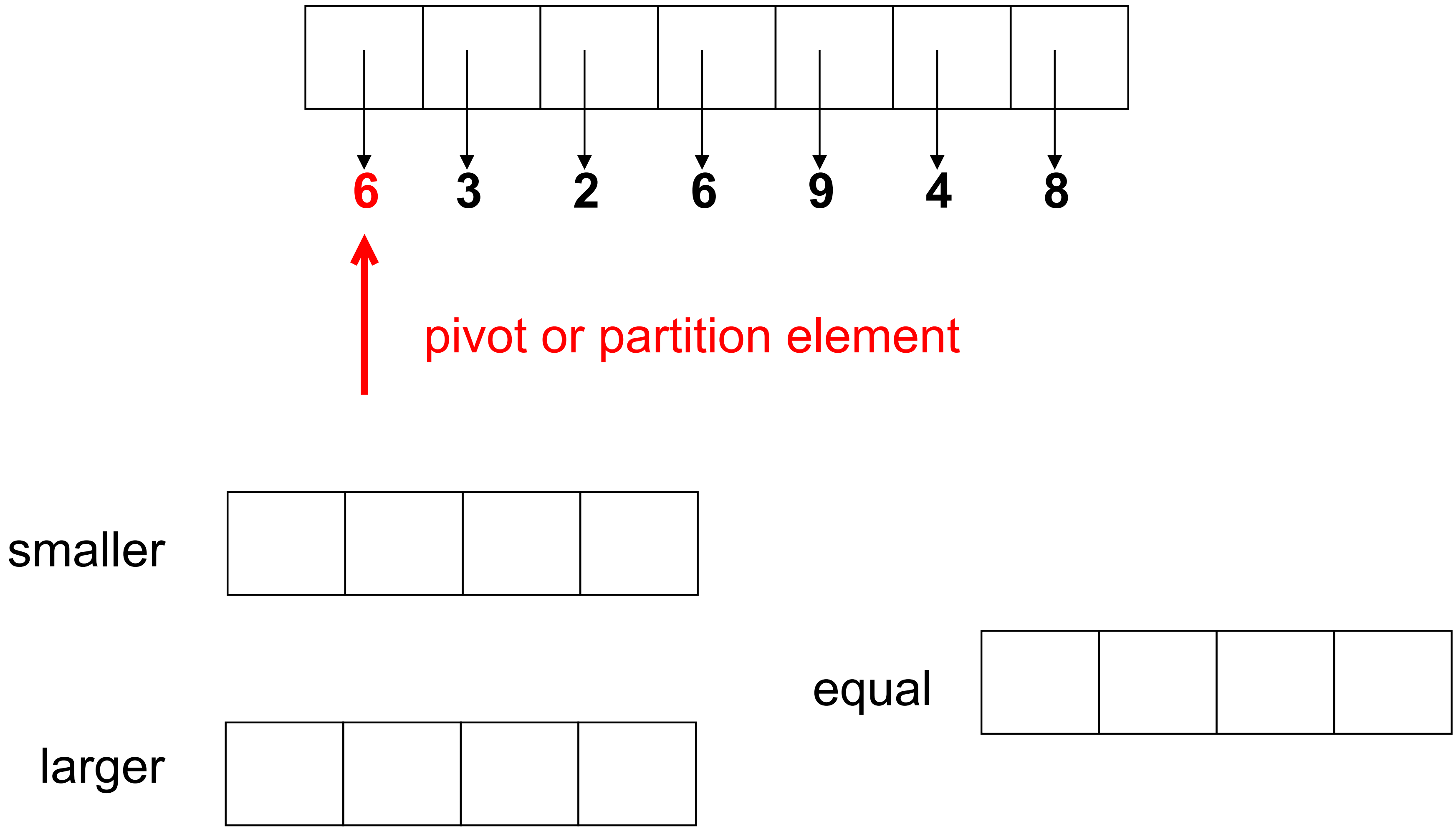
QuickSort



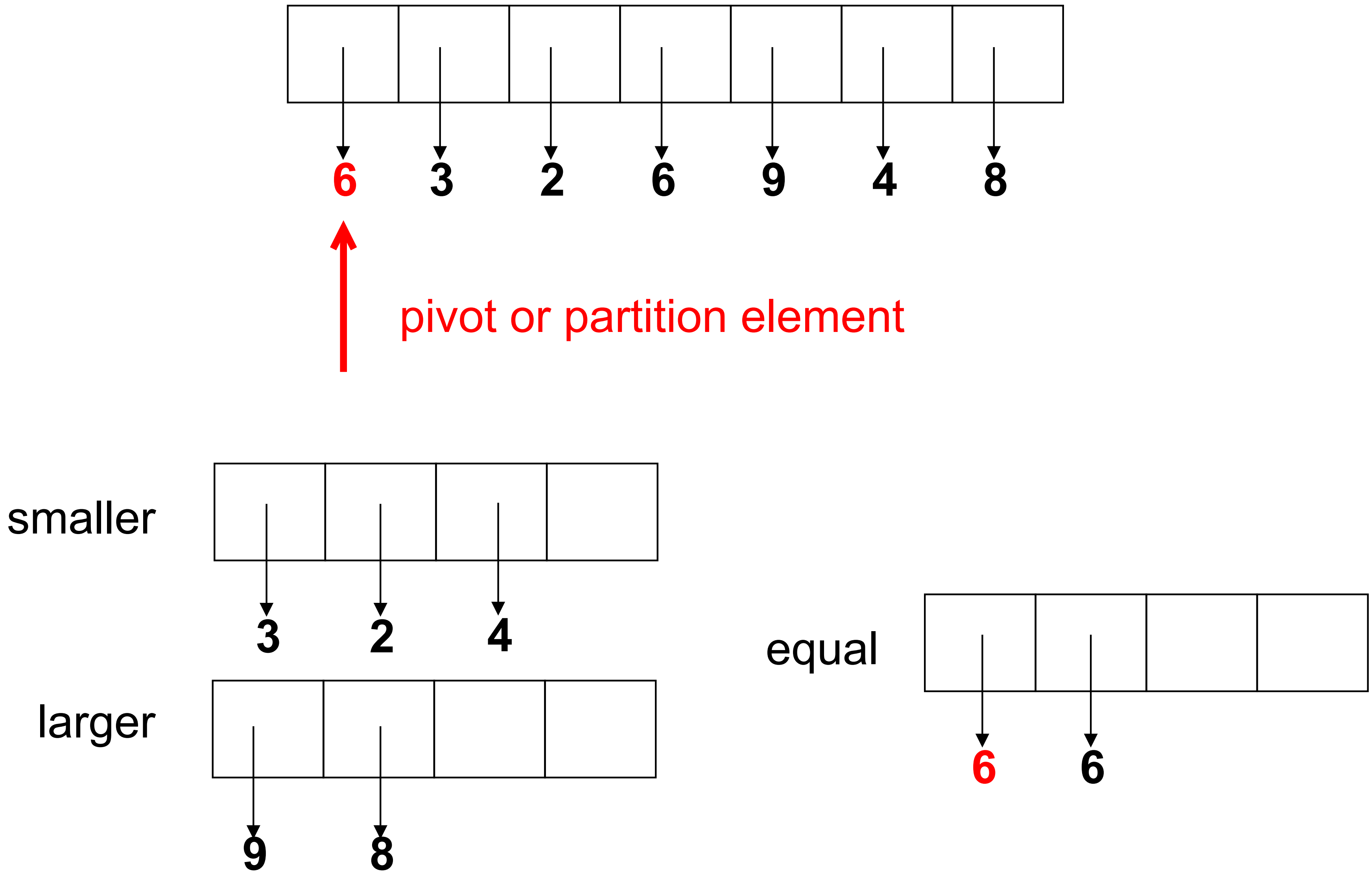

```
public quicksort (Stack A){
pivot= A[A.length/2]
for (int i=0; i< A.length; i++){
    if A[i] > pivot (smaller.additem (A[i]);
    elseif   <          greater.
    else                equal
    }
    A = |smaller|equal|larger

    quicksort(smaller)
    quicksort(larger)
              A;
```

return A }

QuickSort

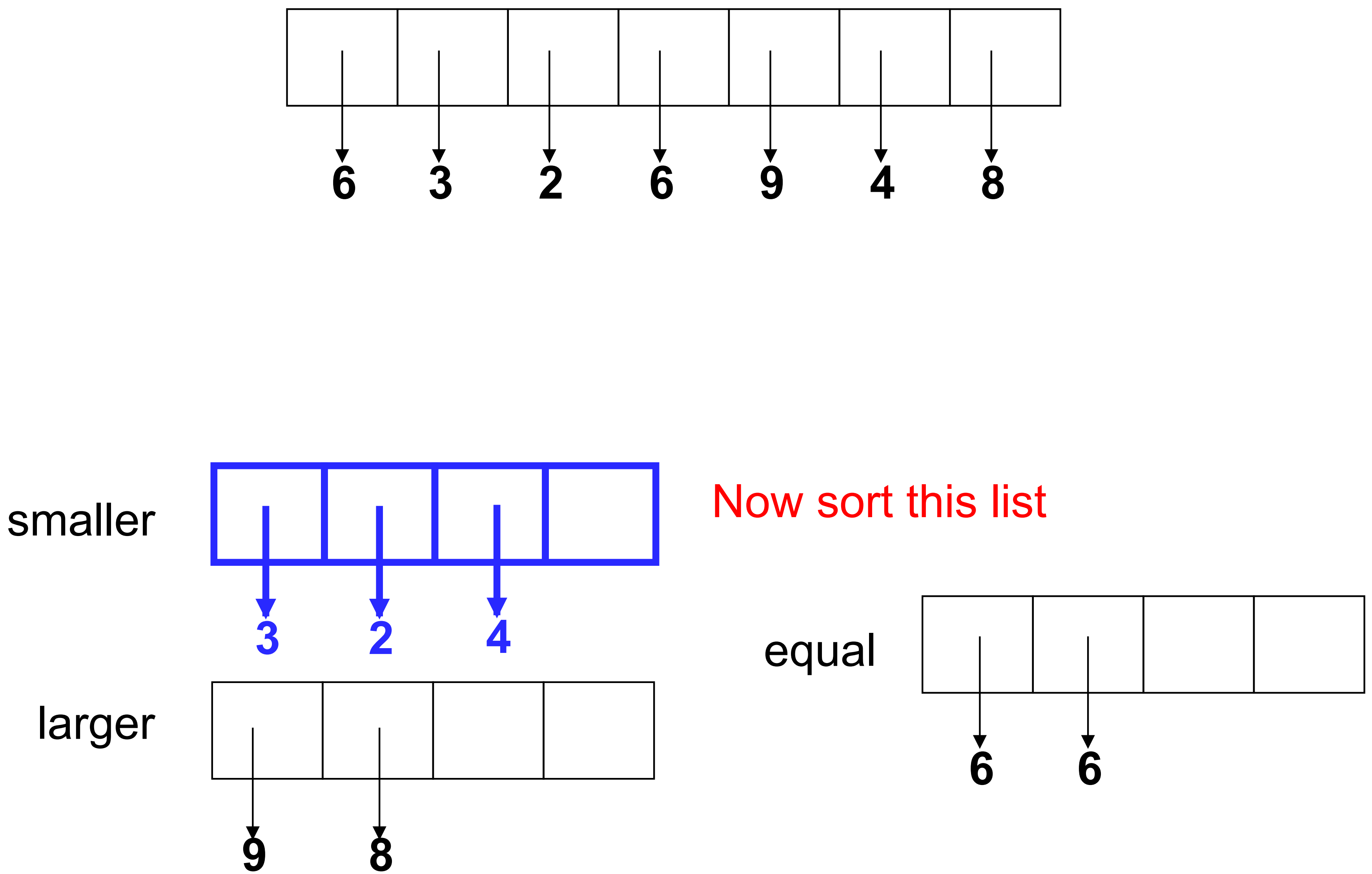


QuickSort

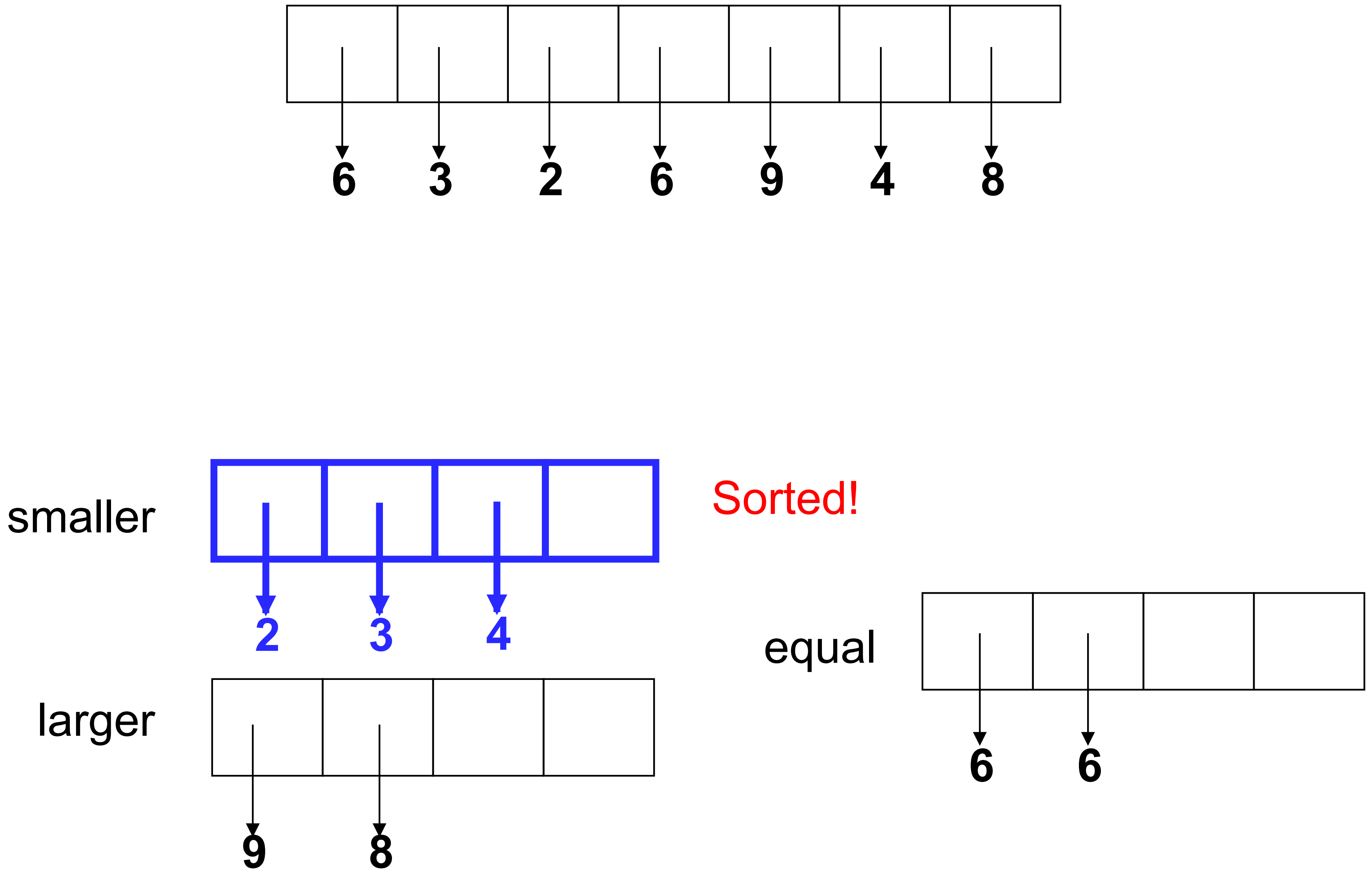


6
3
2
6
9
4
8
pivot or partition element
smaller
3
2
4
larger
9
8
equal
6
6

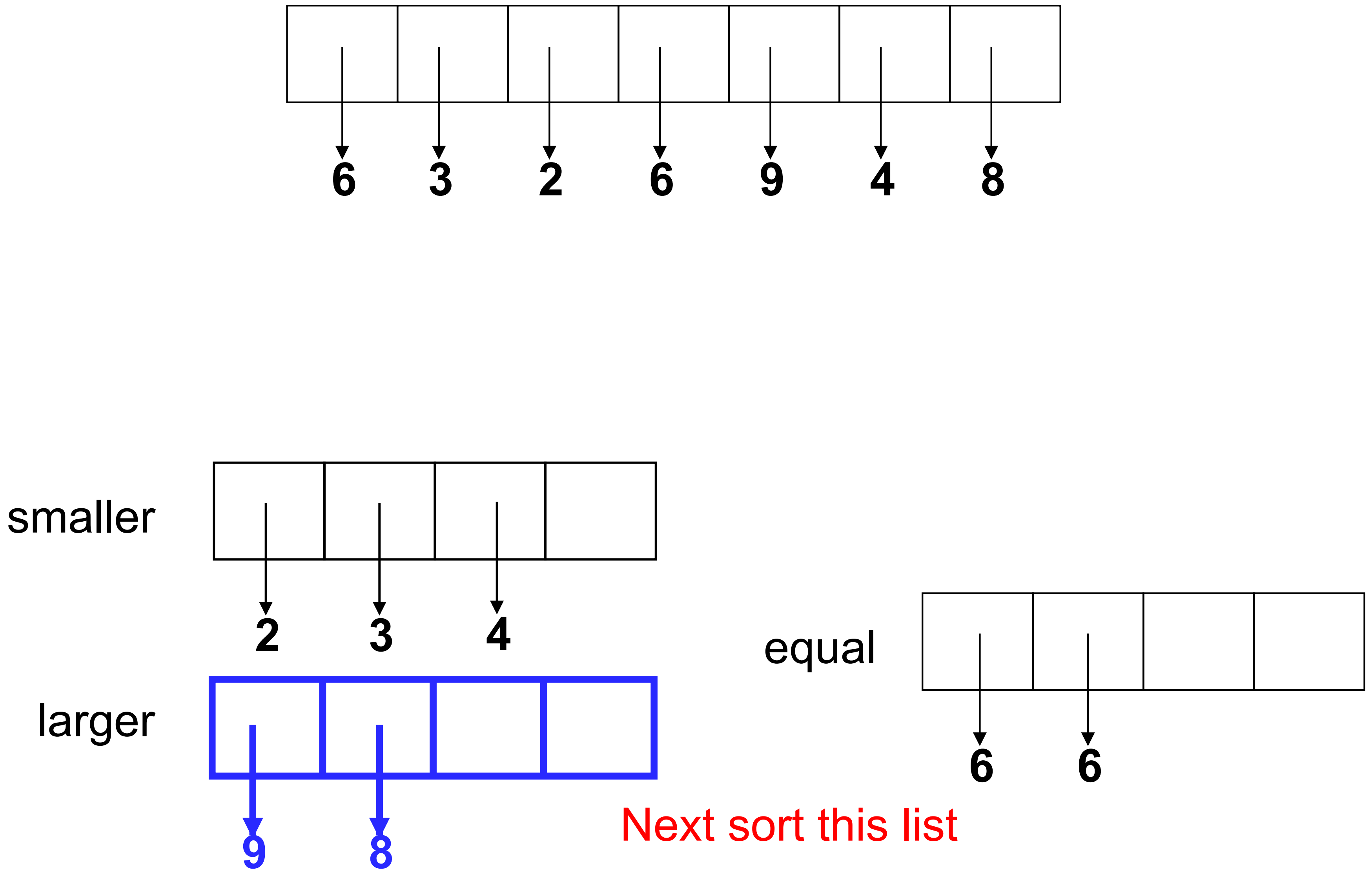
QuickSort



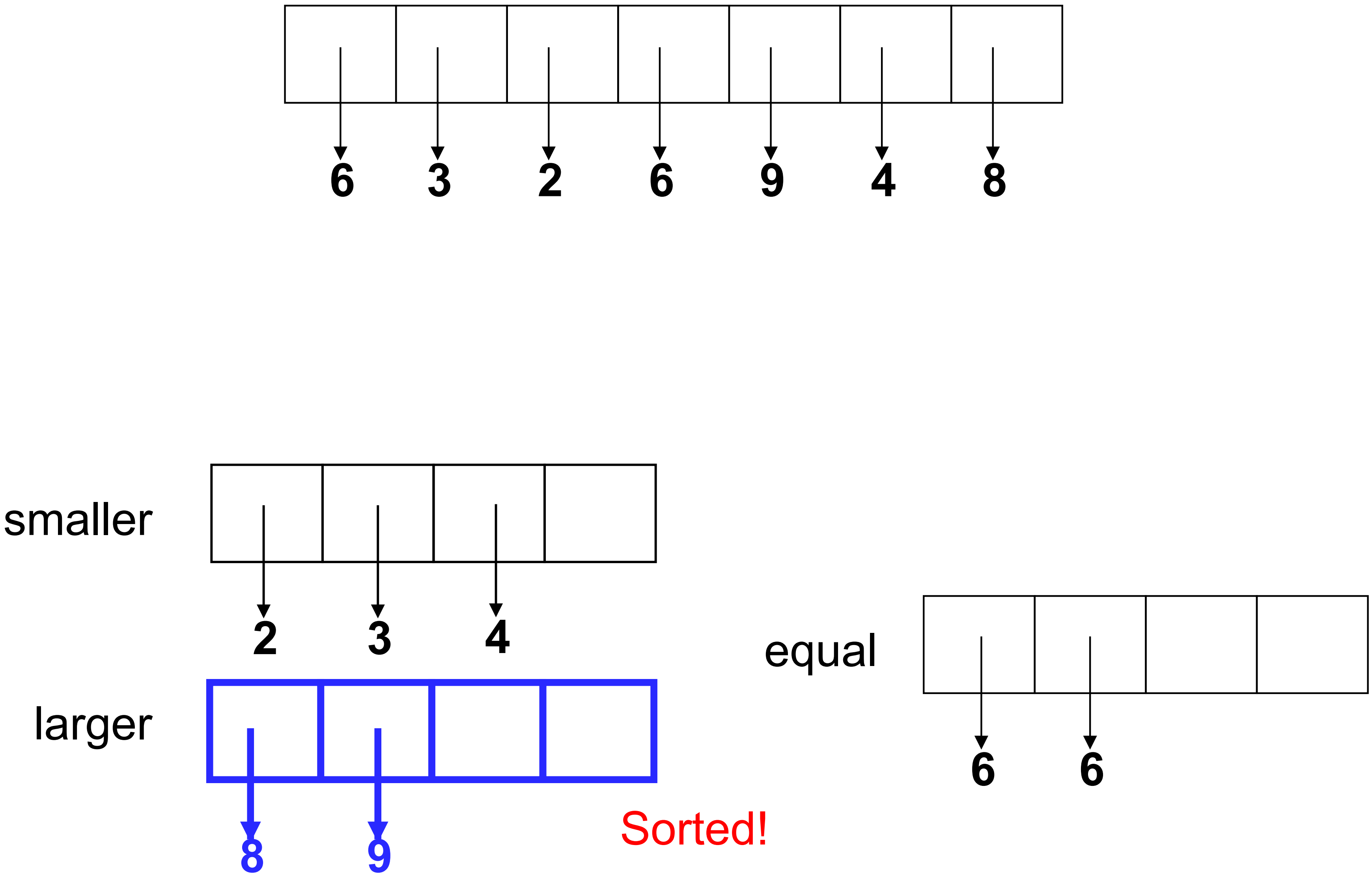
QuickSort



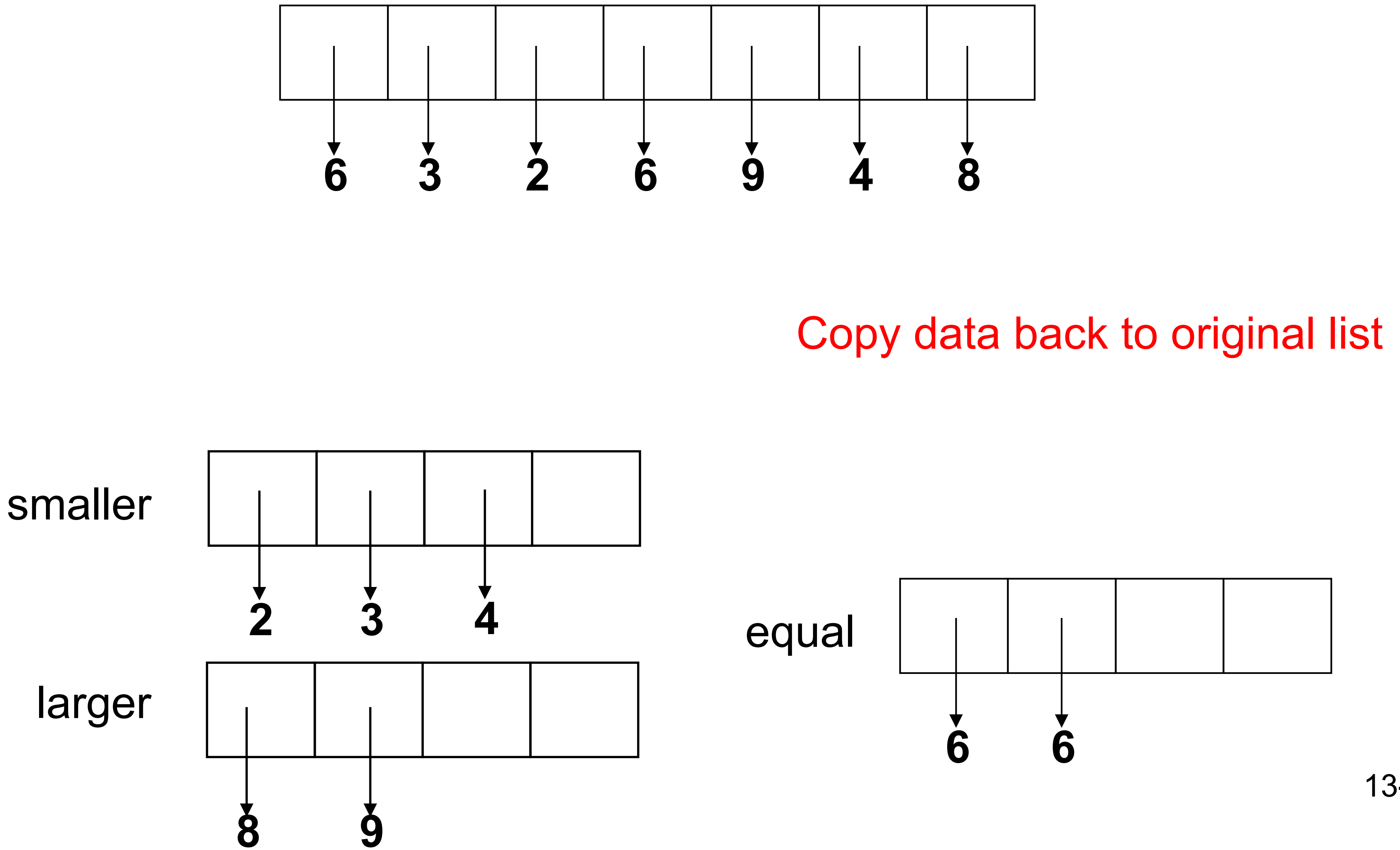
QuickSort



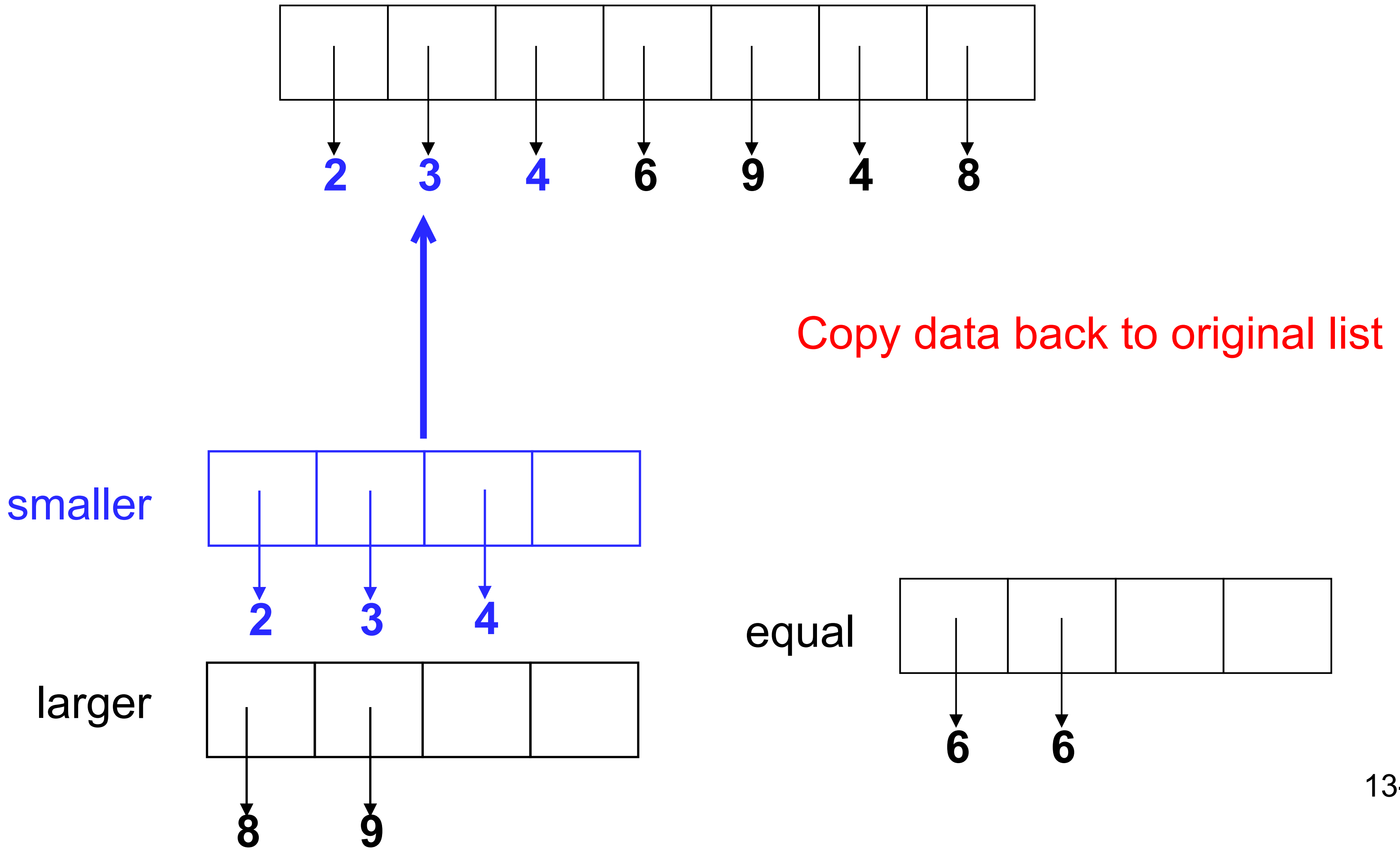
QuickSort



QuickSort

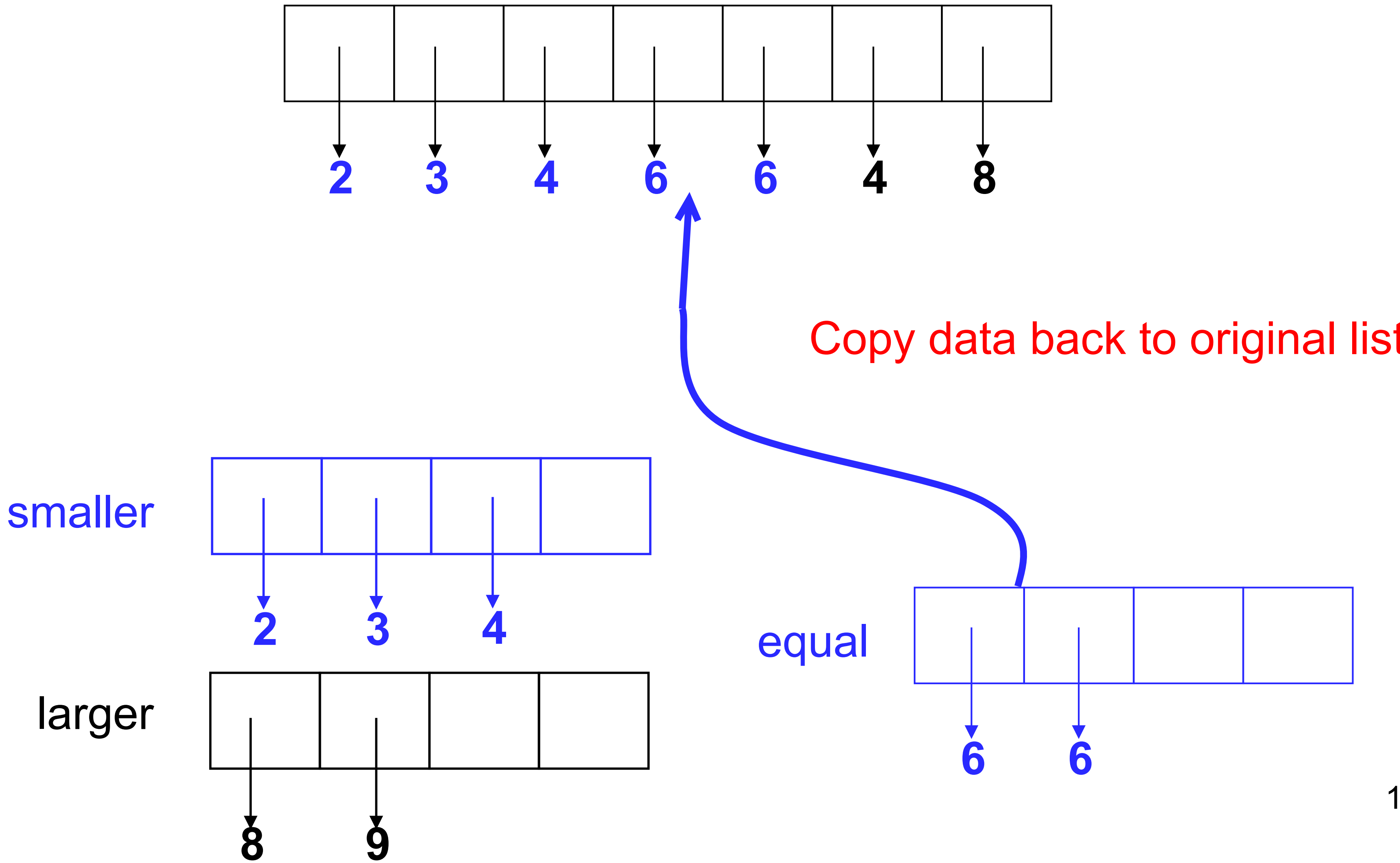

6
3
2
6
9
4
8
Copy data back to original list
smaller
2
3
4
larger
8
9
equal
6
6

QuickSort

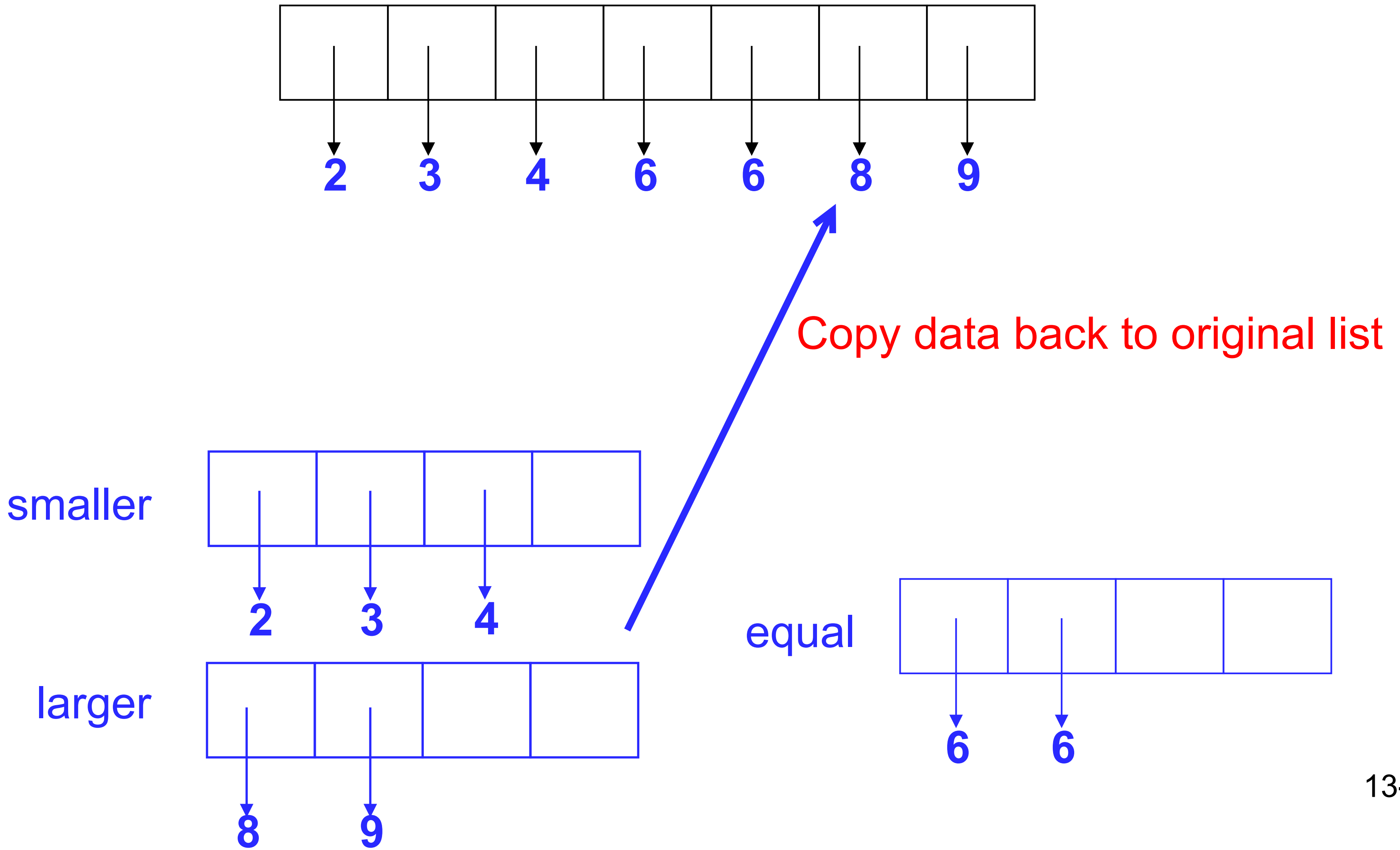


2 3 4 6 9 4 8
Copy data back to original list
smaller
2 3 4
larger
8 9
equal
6 6

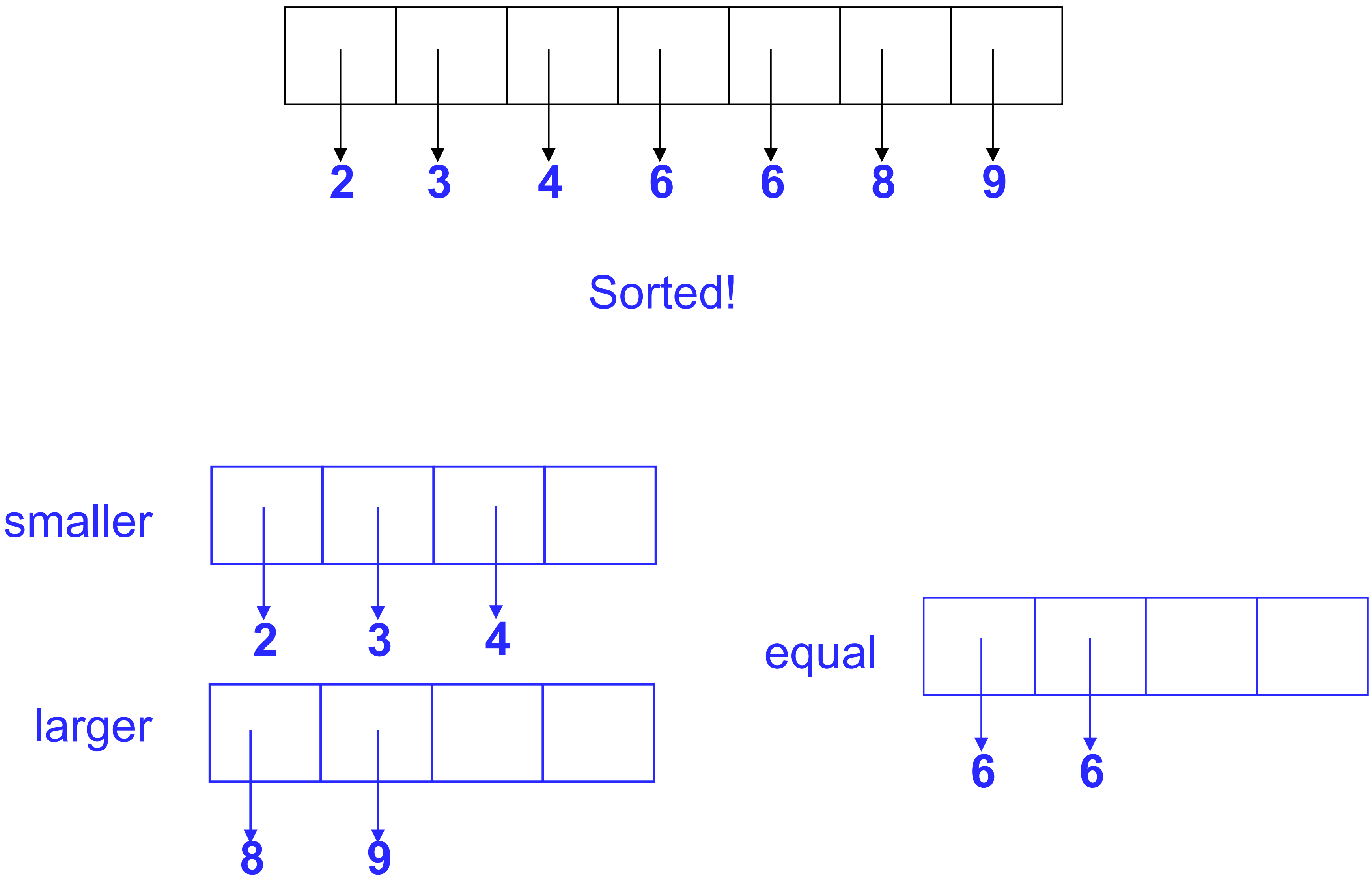
QuickSort


2
3
4
6
6
4
8
Copy data back to original list
smaller
2
3
4
larger
8
9
equal
6
6

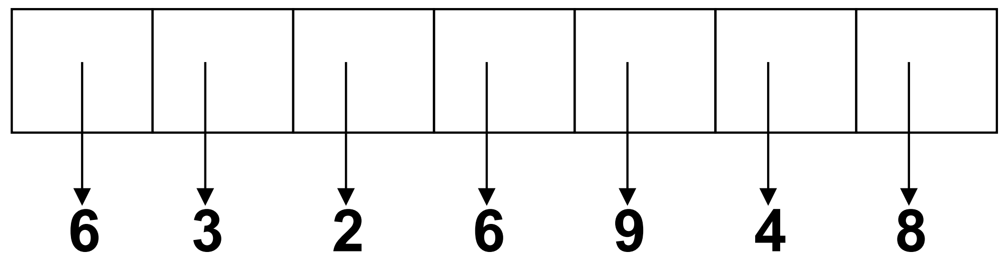
QuickSort

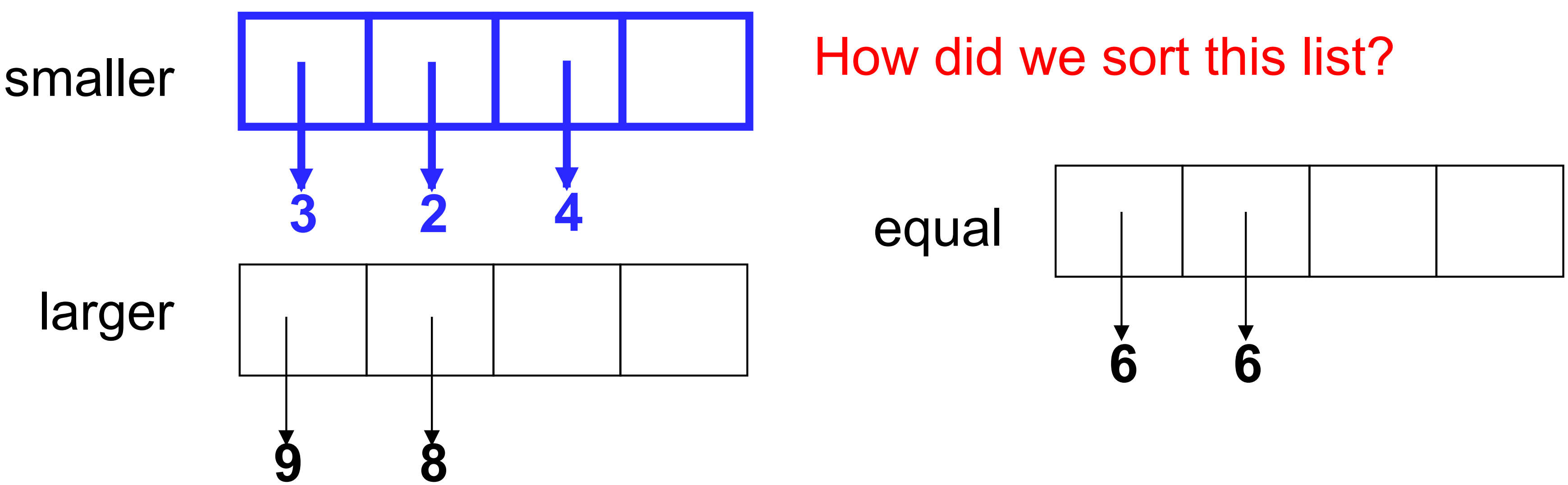

2
3
4
6
6
8
9
Copy data back to original list
smaller
2
3
4
larger
8
9
equal
6
6

QuickSort



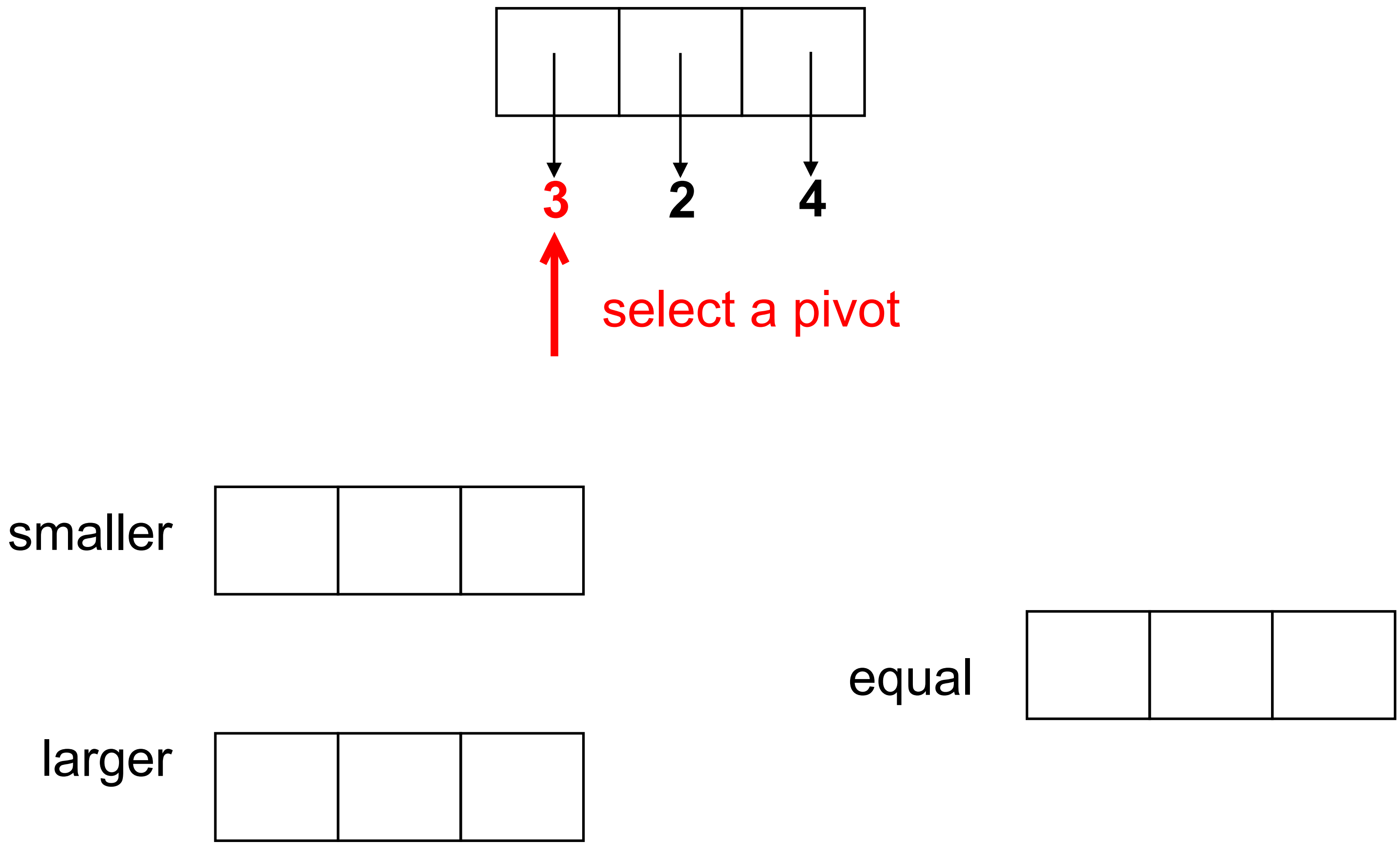
QuickSort



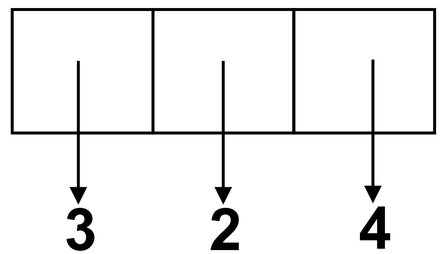


How did we sort this list?

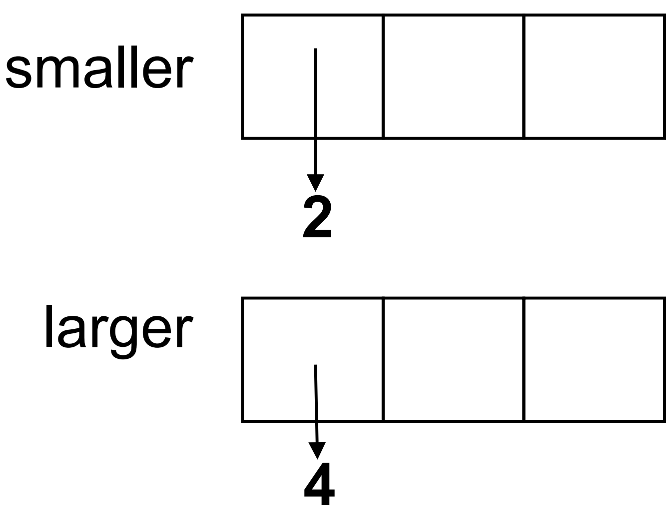
QuickSort

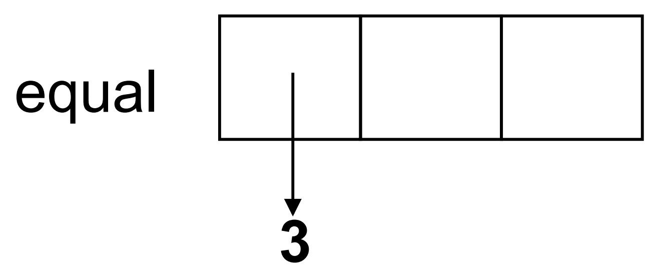


QuickSort

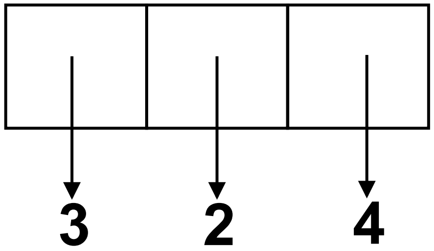


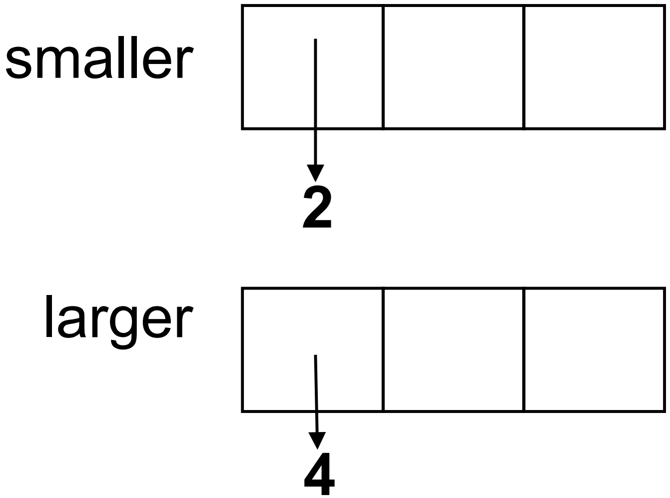
Scan array and put the values in the containers



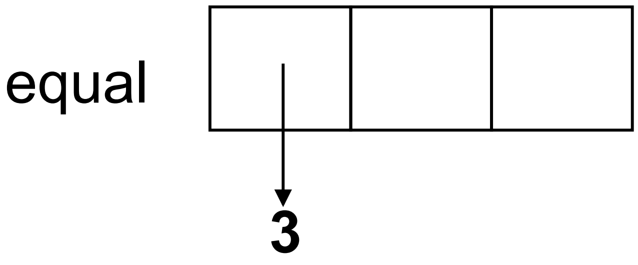


QuickSort

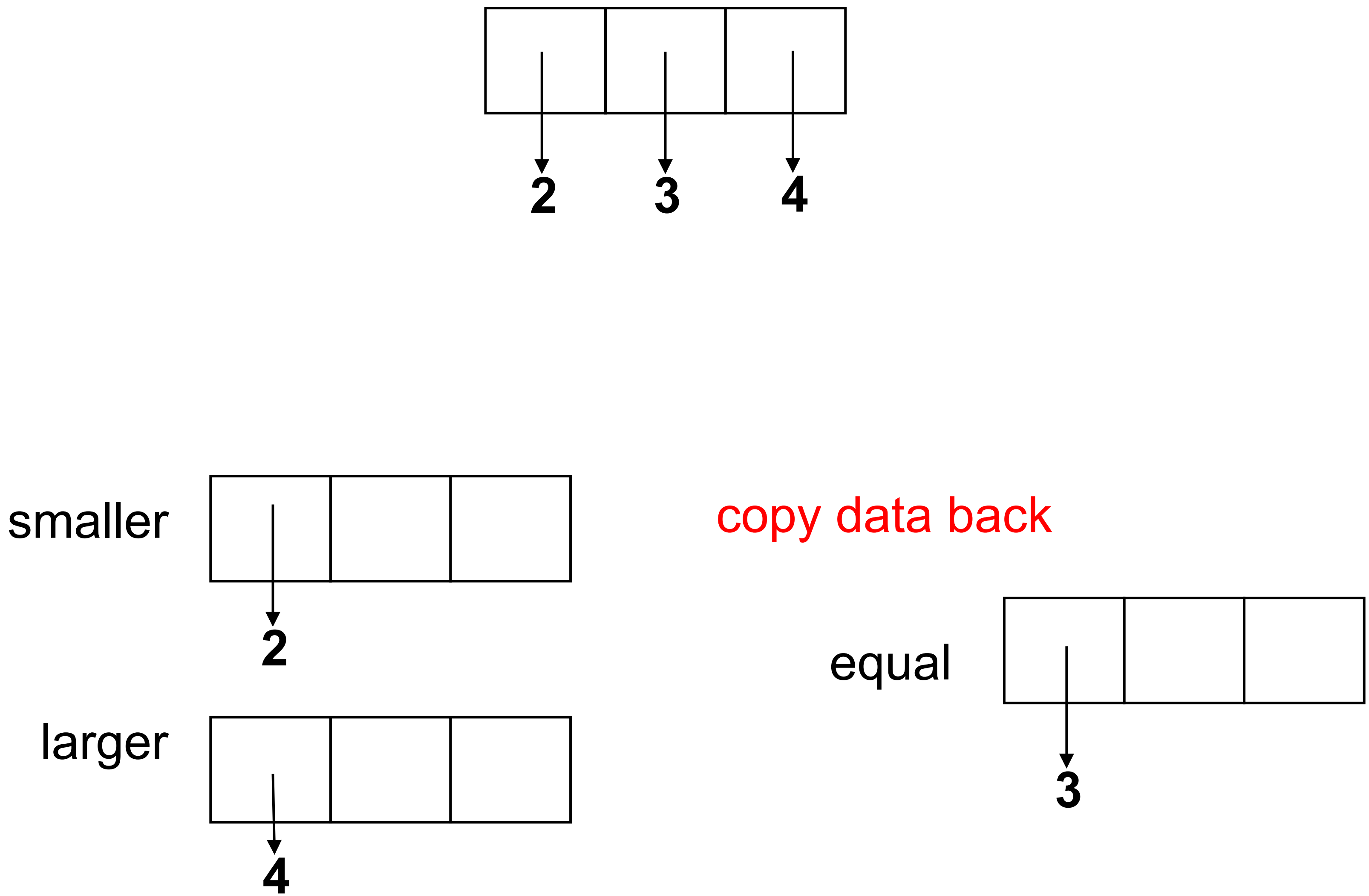




sort the lists



QuickSort



Algorithm quicksort(A,n)

In: Array A storing n values

Out: {Sort A in increasing order}

If n > 1 **then** {

smaller, equal, larger = new arrays of size n

$n_s = n_e = n_l = 0$

pivot = A[0]

for i = 0 **to** n-1 **do** // Partition the values

if A[i] = pivot **then** equal[n_e++] = A[i]

else if A[i] < pivot **then** smaller[n_s++] = A[i]

else larger[n_l++] = A[i]

quicksort(smaller,n_s)

quicksort(larger,n_l)

i = 0

for j = 0 **to** n_s **do** A[i++] = smaller[j]

for j = 0 **to** n_e **do** A[i++] = equal[j]

for j = 0 **to** n_l **do** A[i++] = larger[j]

}

after adding the element, increment the counter

n

n

n

A	2	3	4

smaller	2		n_s=1
equal	3		n_e=1
larger	4		n_l=1

Analysis of Quick Sort

- **We will look at two cases for Quick Sort:**
  - ***Worst case***
    - **When the pivot element is the *largest* or *smallest* item in the container (why is this the worst case?)**
  - ***Best case***
    - **When the pivot element is the *middle* item (why is this the best case?)**

Analysis of Quick Sort