Sorting Algorithms

Objectives

- Examine different sorting algorithms that can be implemented in-place (without the use of auxiliary collections) and using auxiliary collections.

Sorting Problem

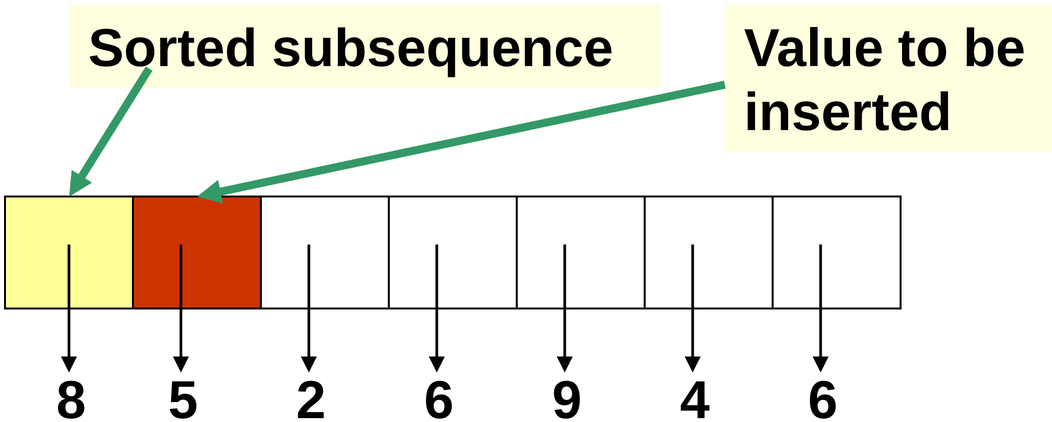
- Consider an unordered list of *n* objects that we wish to have sorted into ascending order
- We will study the following sorting algorithms:
  - *Insertion sort* using stacks and in-place
  - *Selection sort* using queues and in-place
  - *Quick Sort*

Insertion Sort

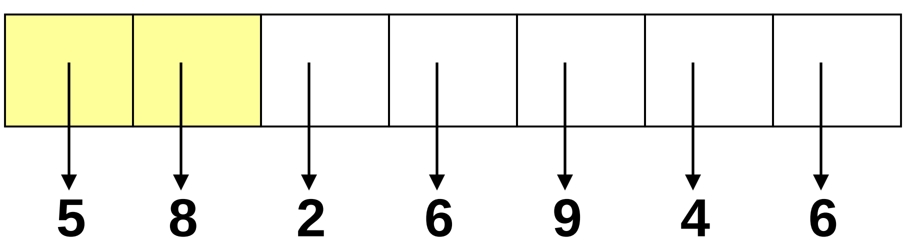
- ***Insertion Sort*** orders a sequence of values by repeatedly taking each value and inserting it in its proper position within a *sorted subset* of the sequence.
- More specifically:
  - Consider the first item to be a *sorted subsequence* of length **1**
  - Insert the second item into the *sorted 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

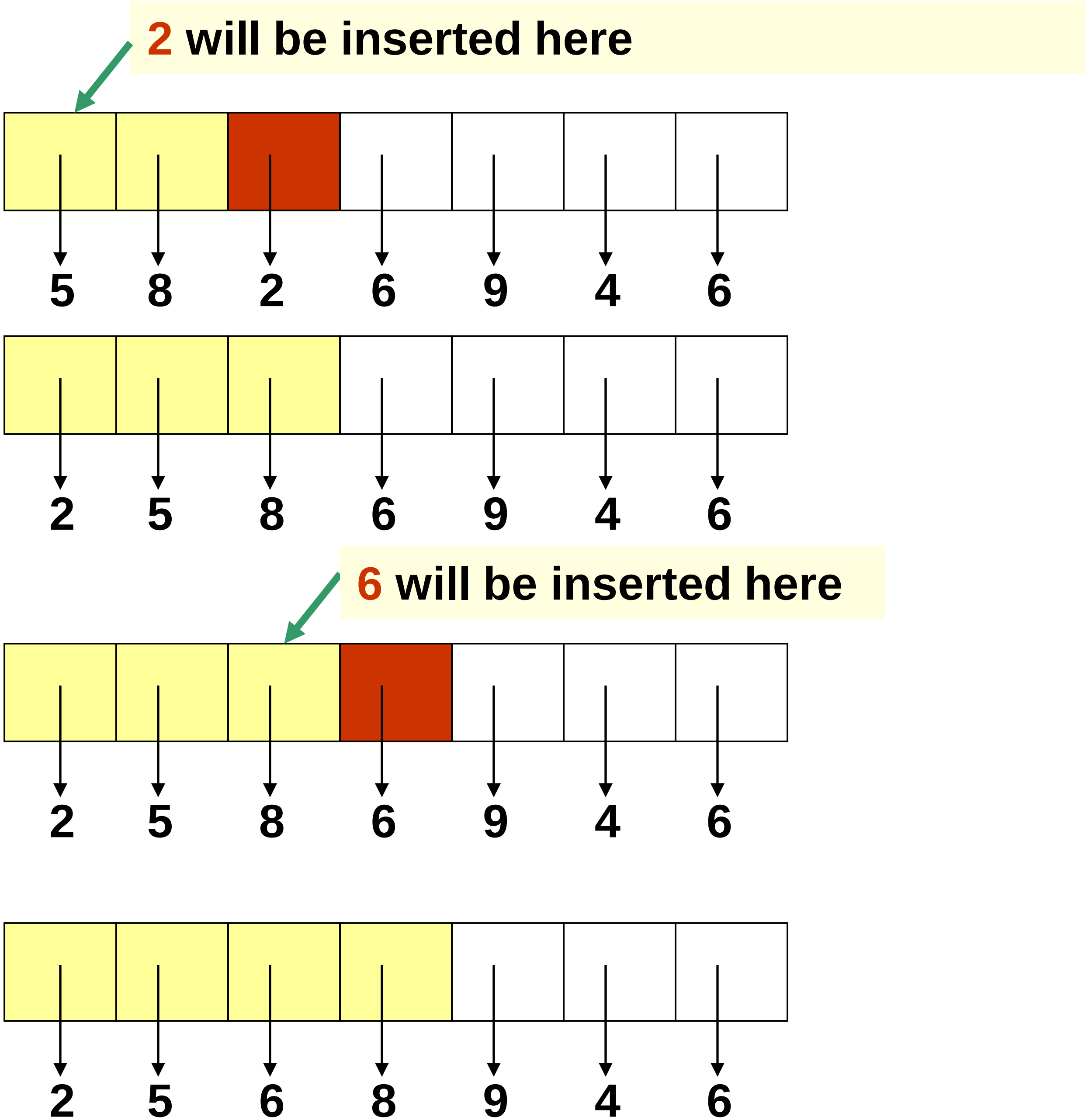
Insertion Sort Algorithm

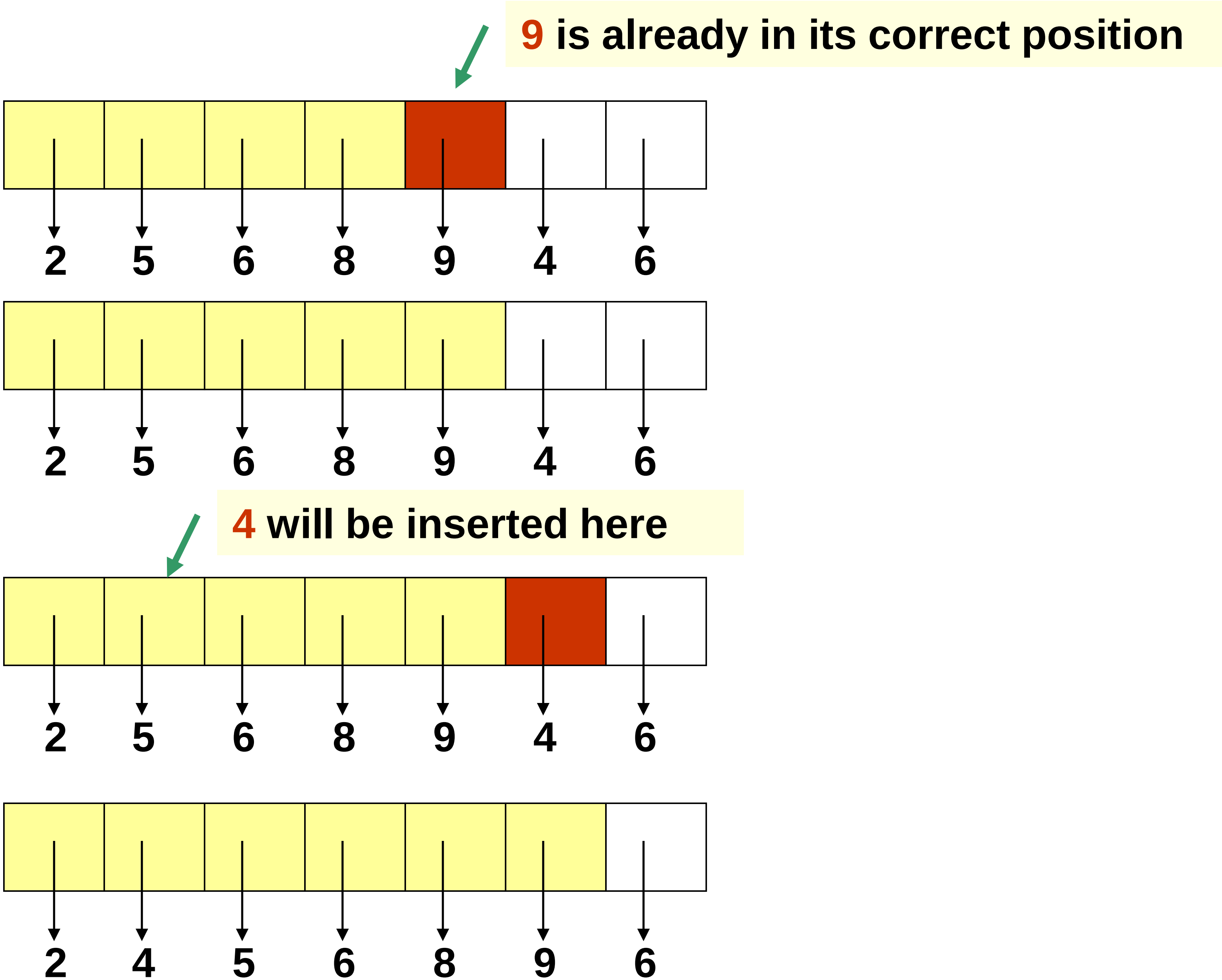
Example: **sorting a sequence of Integer objects**



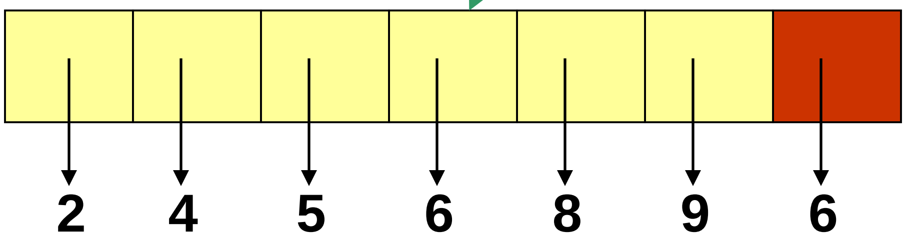
Value 5 is to be inserted in the sorted sequence to its left. Since 5 is smaller than 8, then 8 needs to be shifted one position to the right and then 5 can be inserted on the first position of the array.



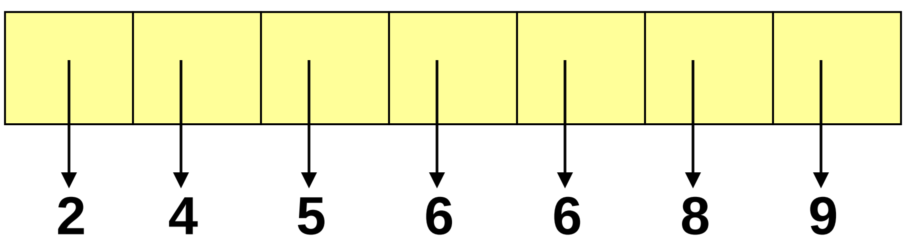

2 will be inserted here
5 8 2 6 9 4 6
2 5 8 6 9 4 6
6 will be inserted here
2 5 8 6 9 4 6
2 5 6 8 9 4 6


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**

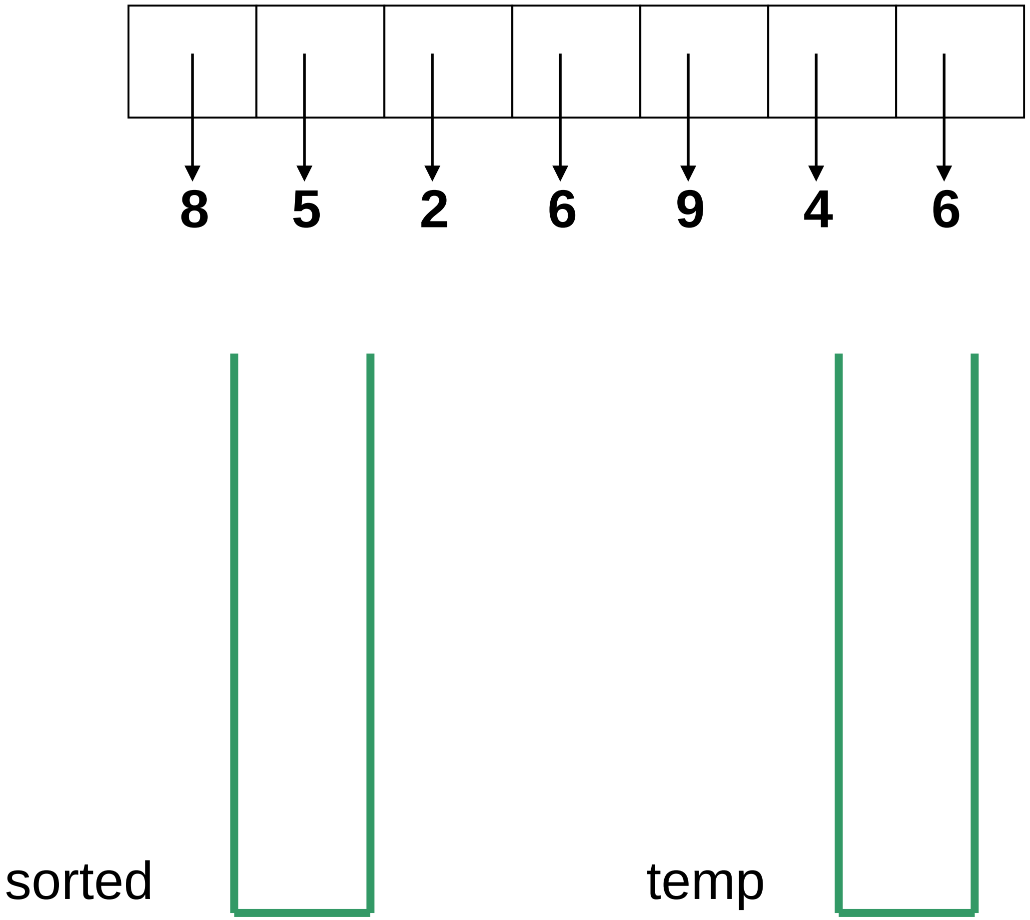
Algorithm insertionSort (A,n)

In: Array A storing n elements

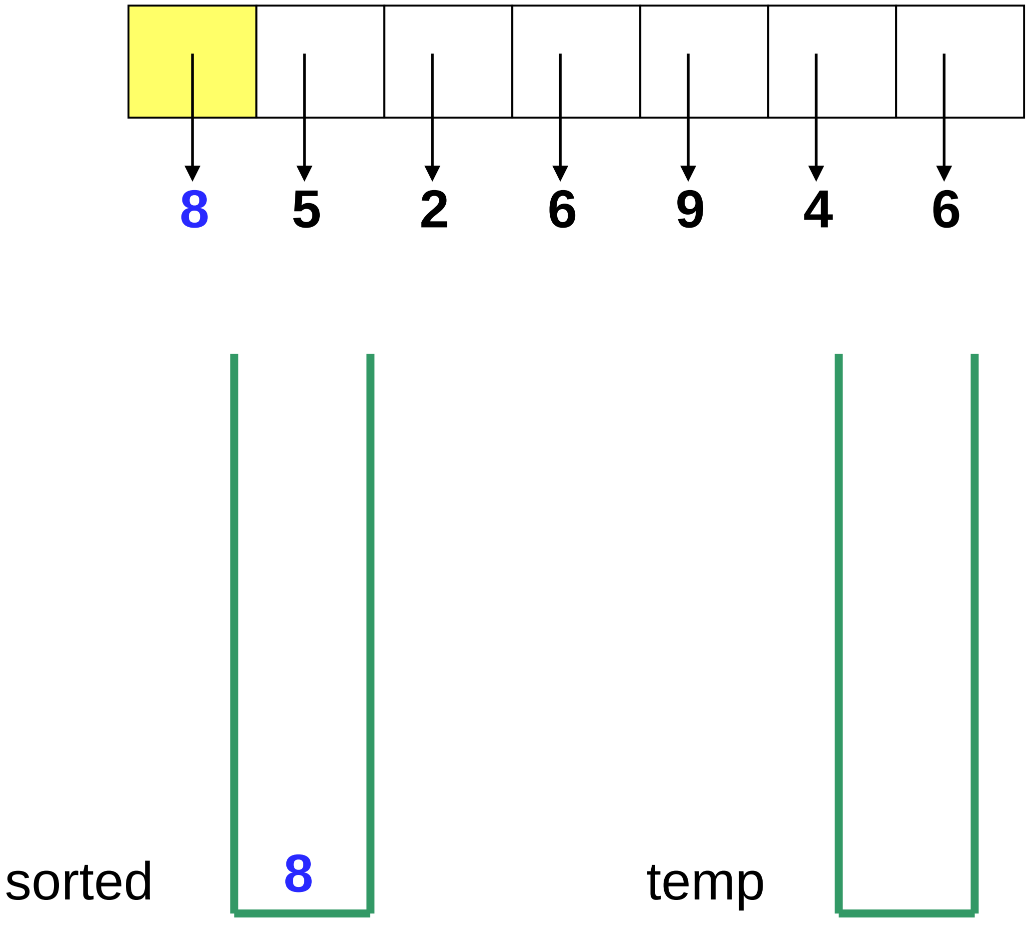
Out: Sorted array

```
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
```

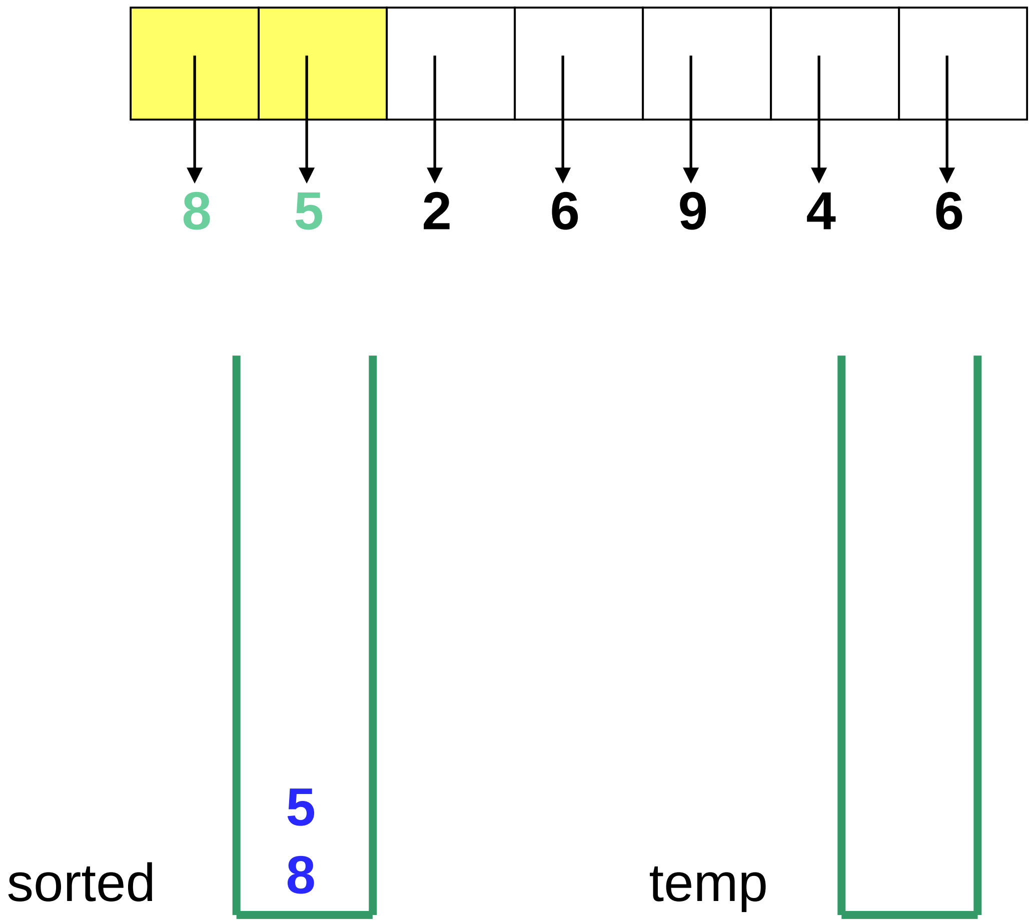
Insertion Sort



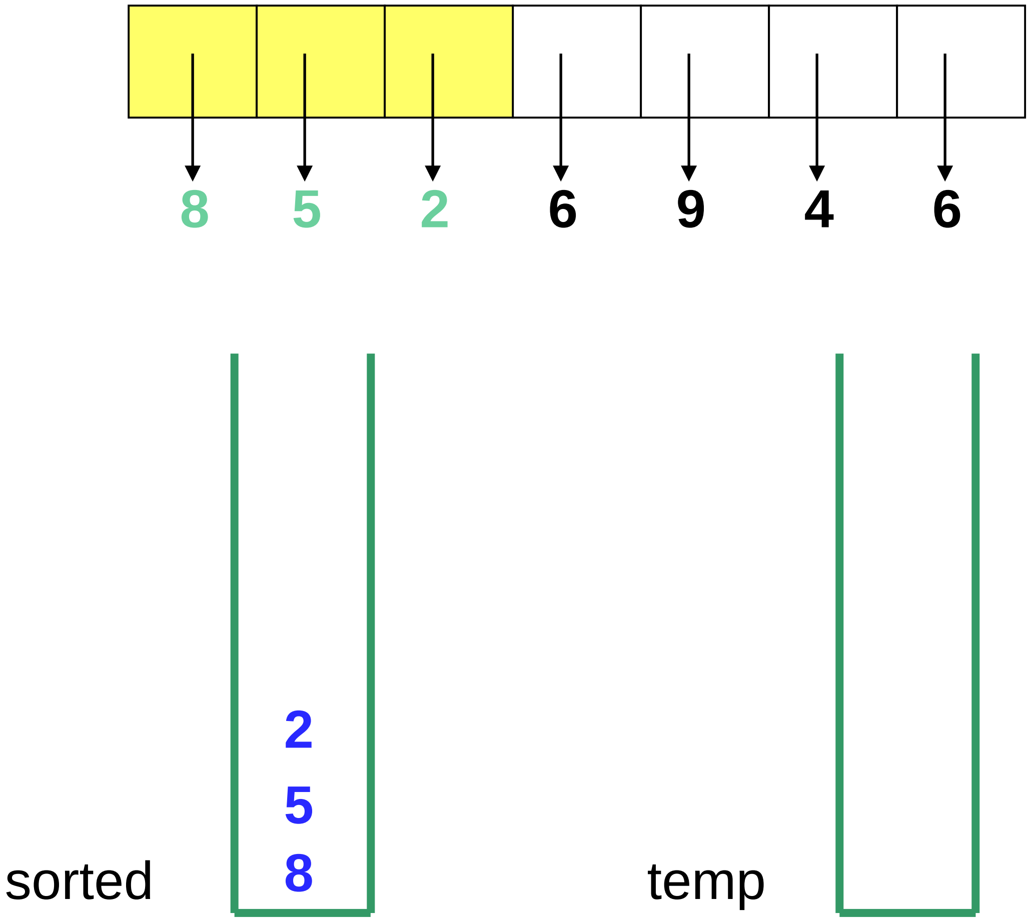
Insertion Sort



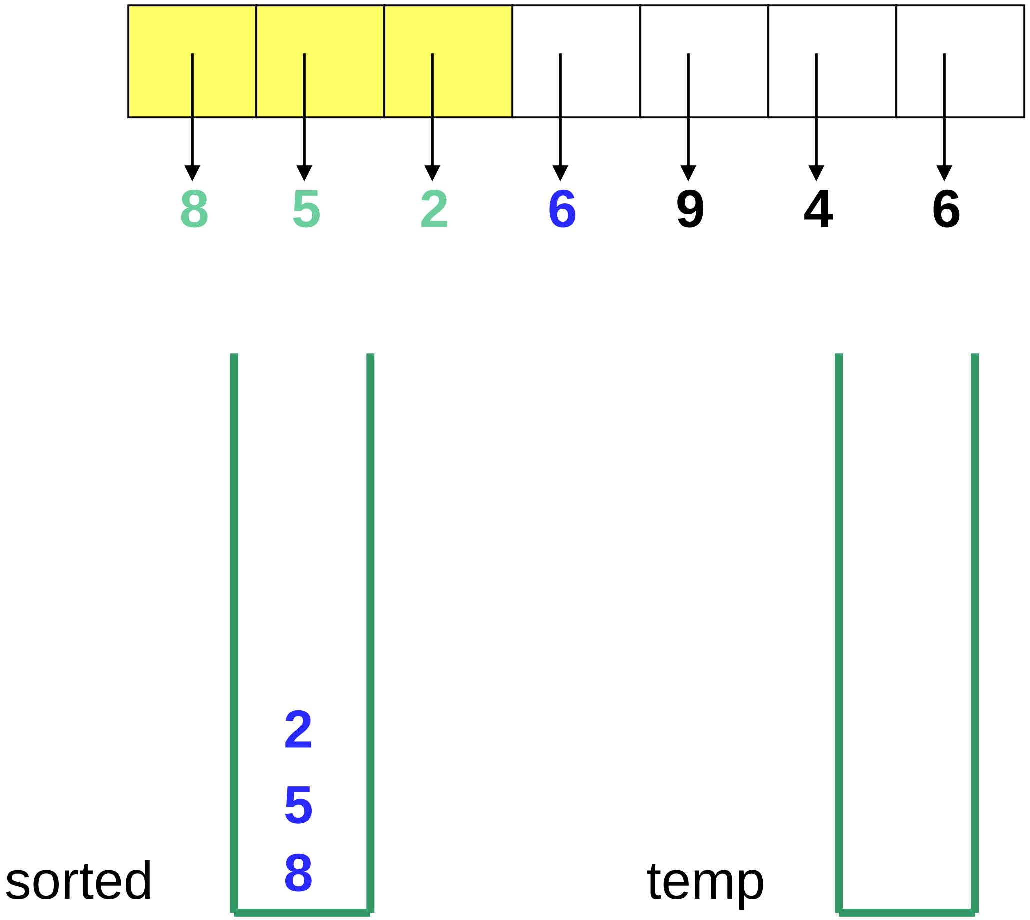
Insertion Sort



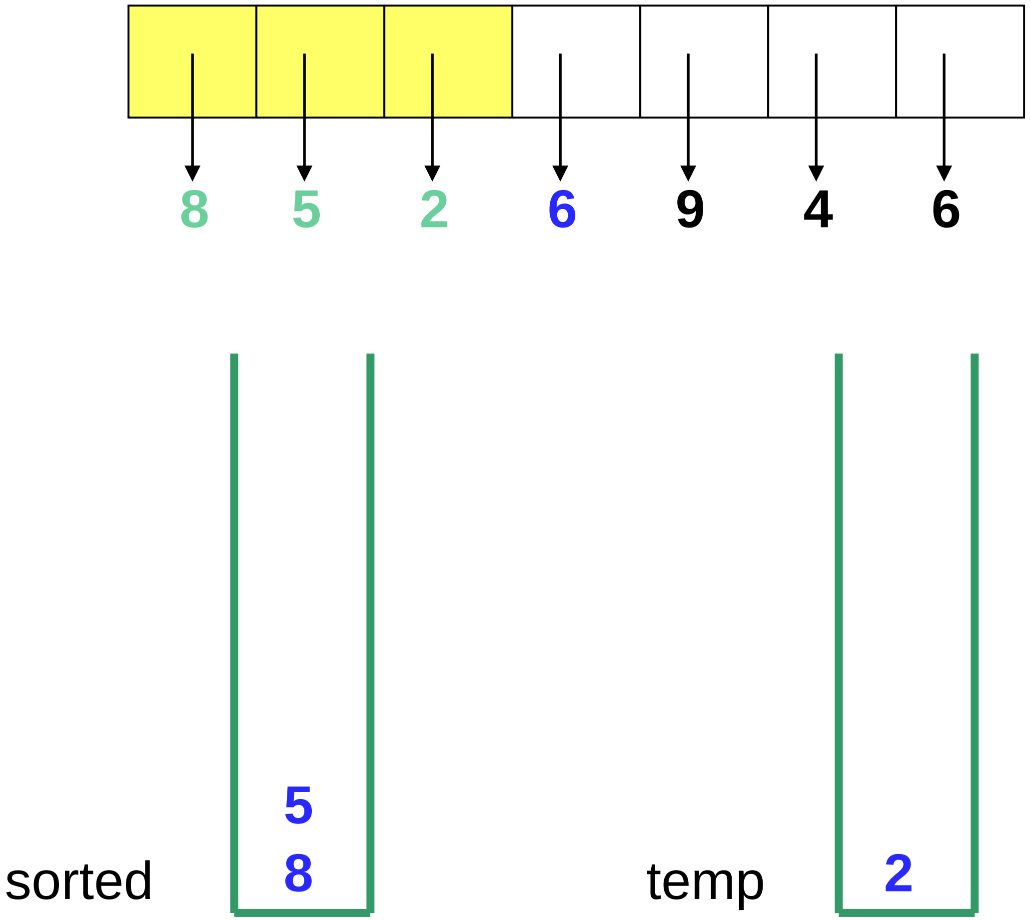
Insertion Sort



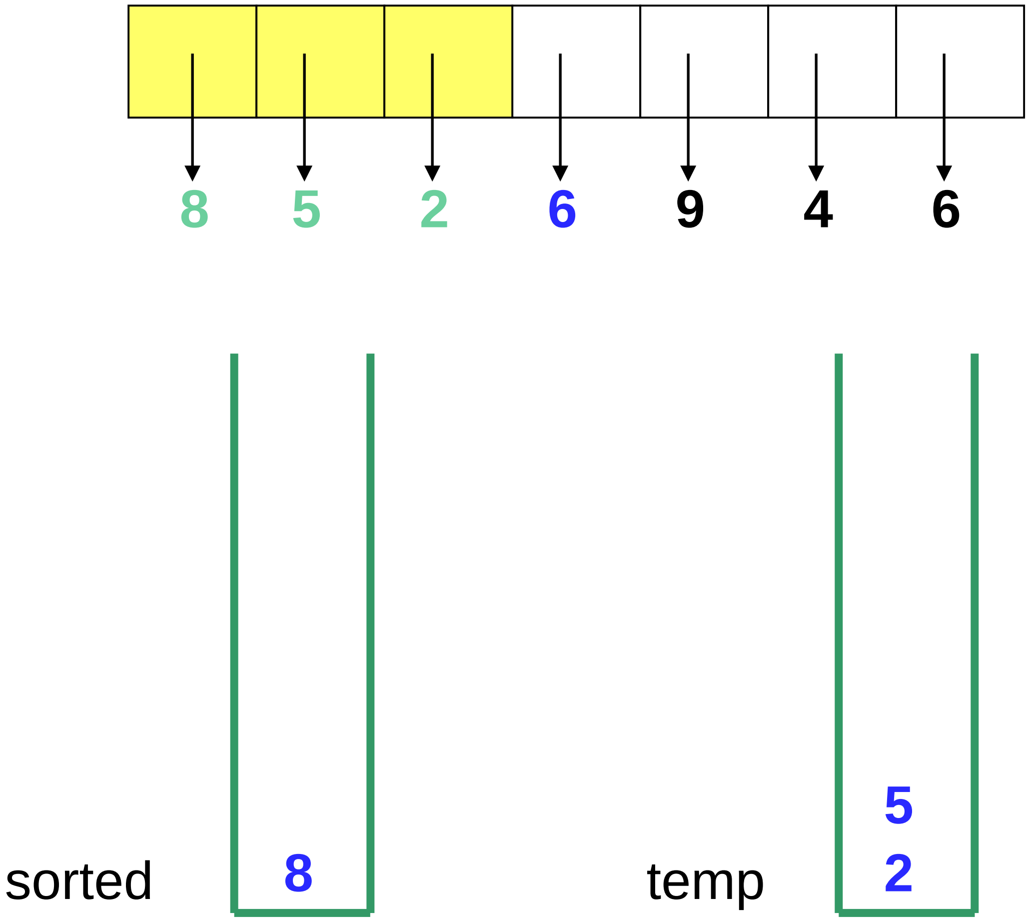
Insertion Sort



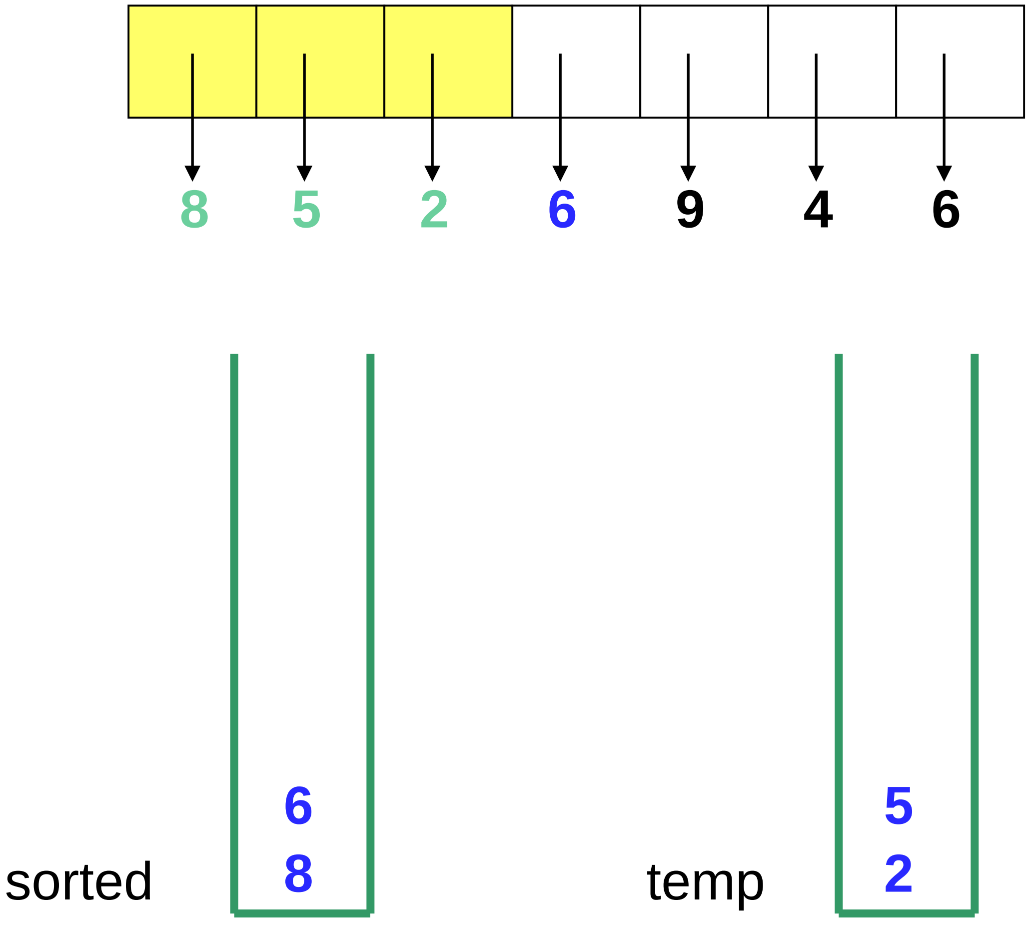
Insertion Sort



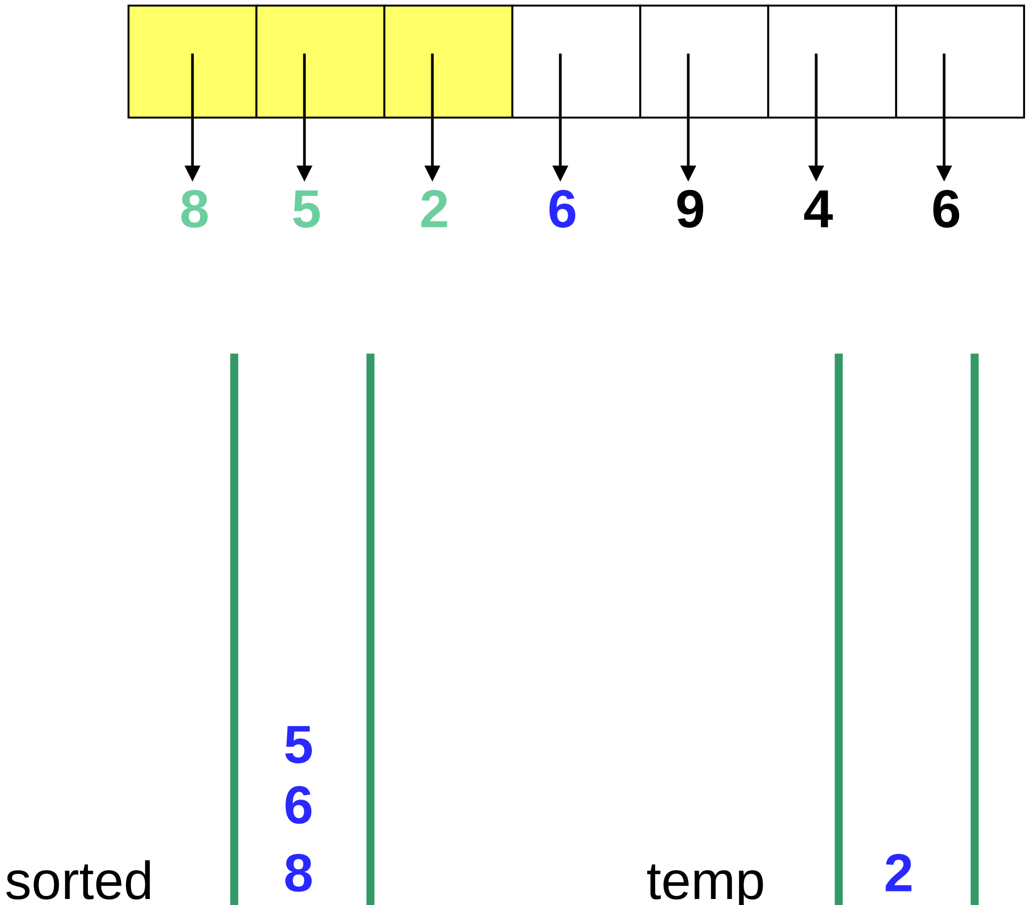
Insertion Sort



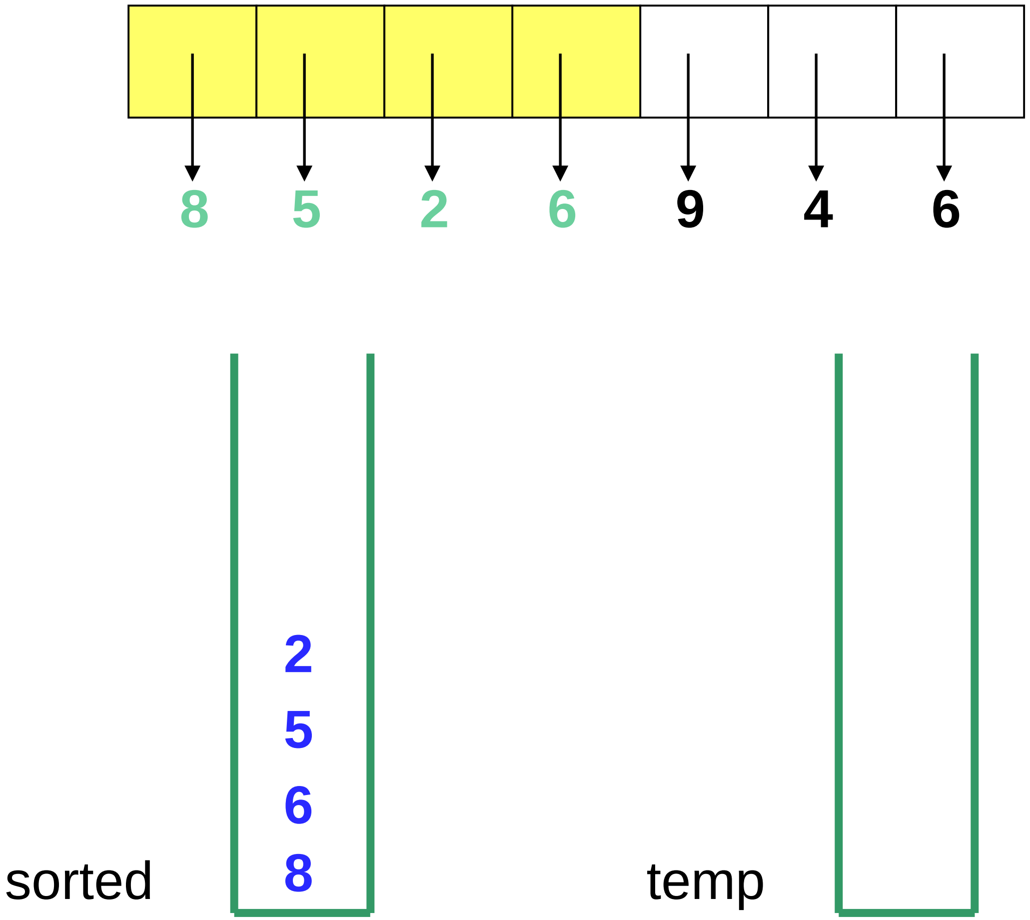
Insertion Sort



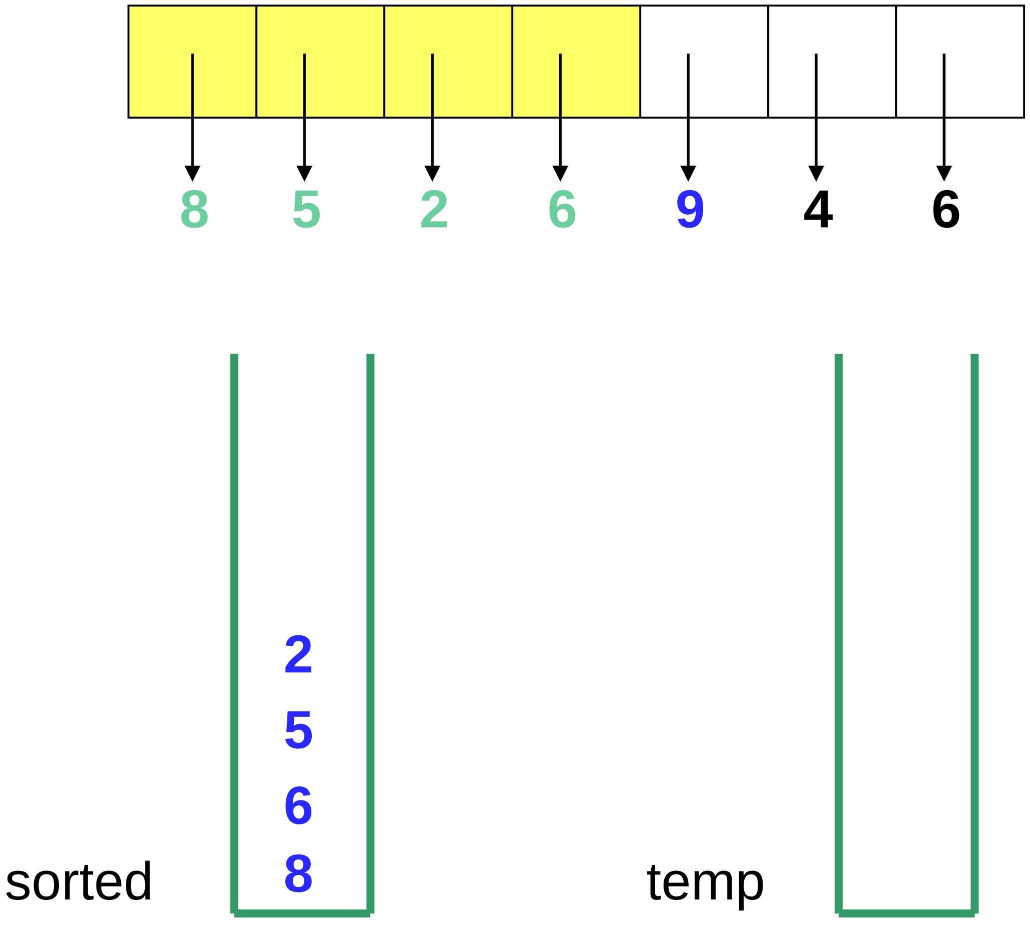
Insertion Sort



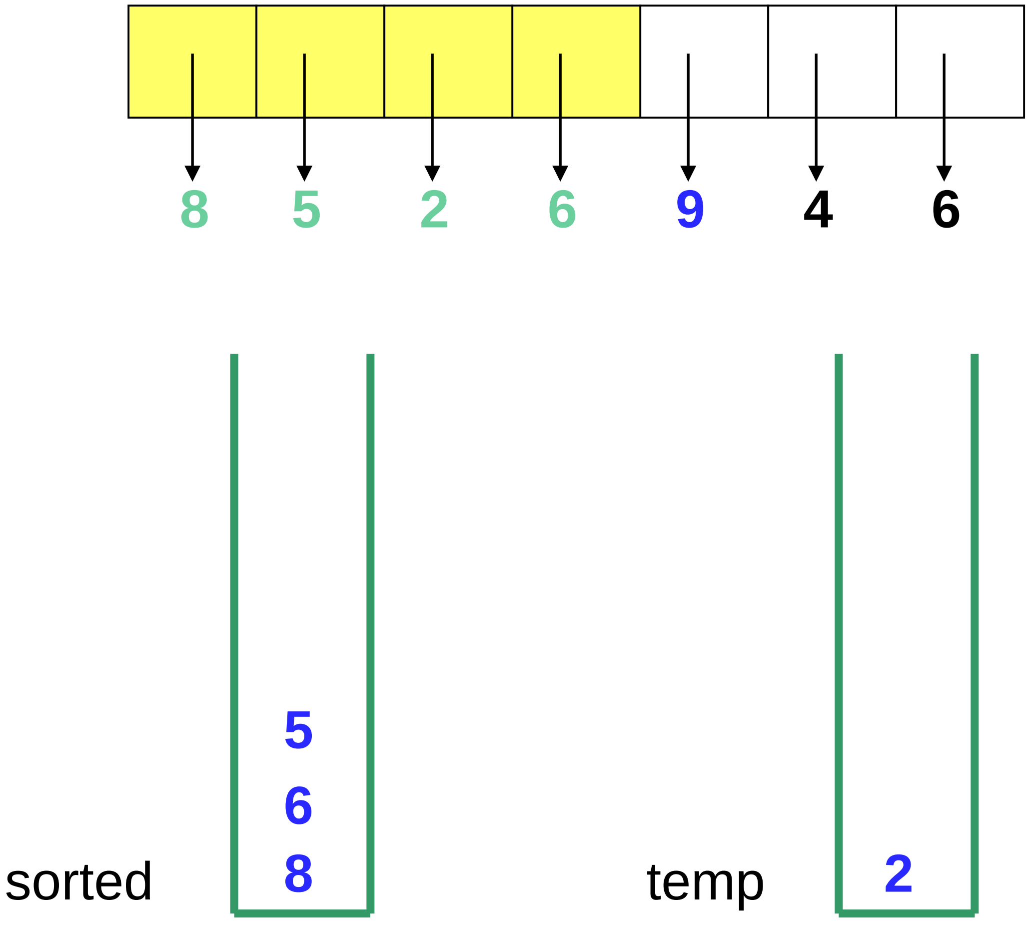
Insertion Sort



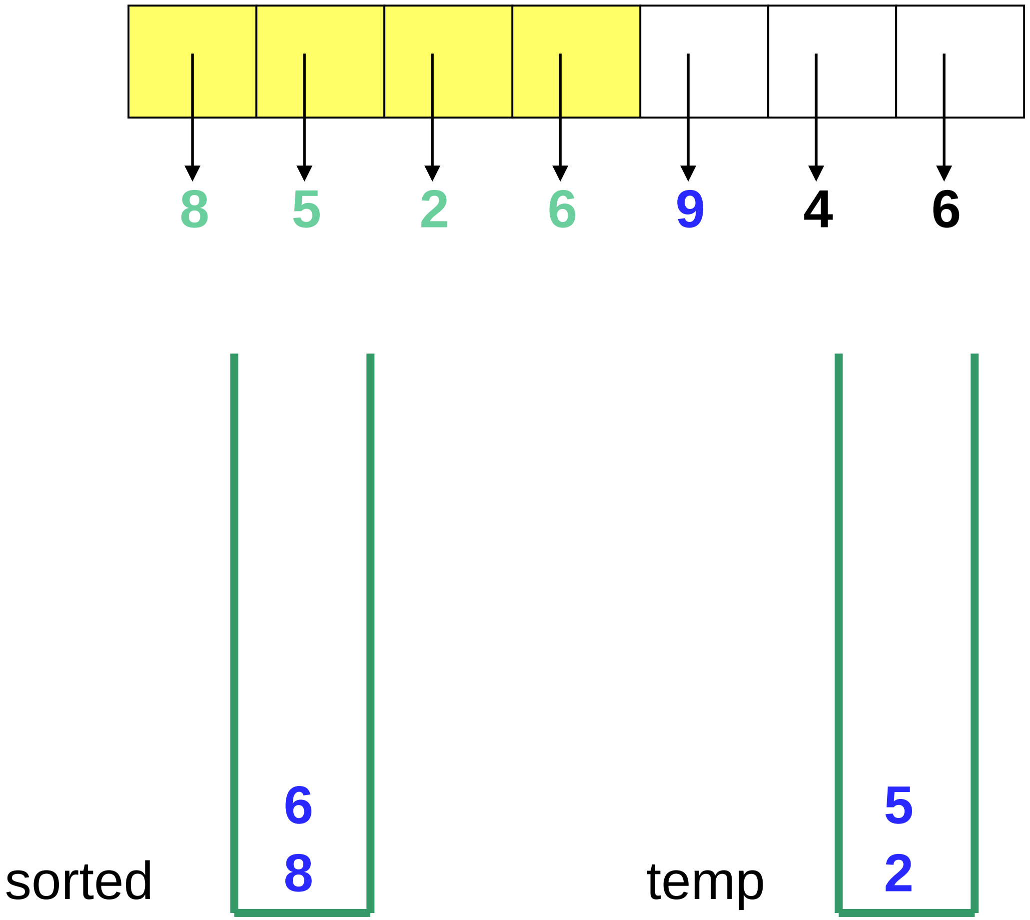
Insertion Sort



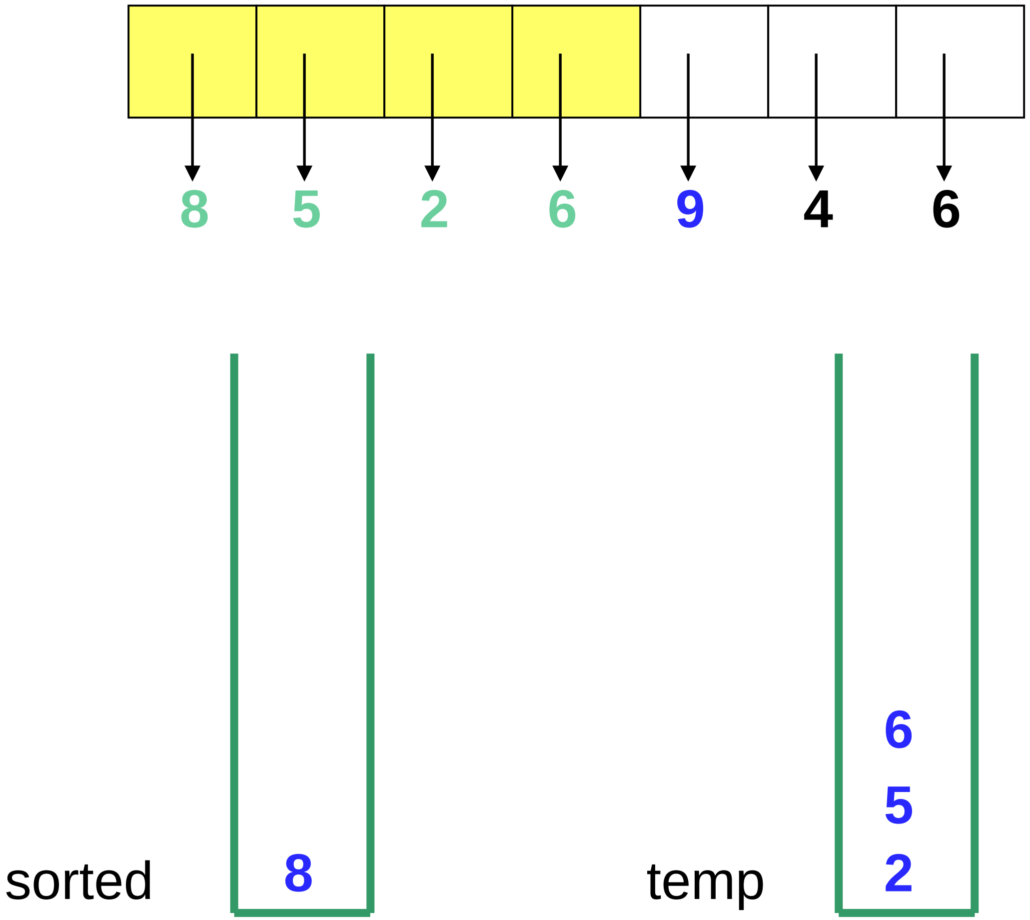
Insertion Sort



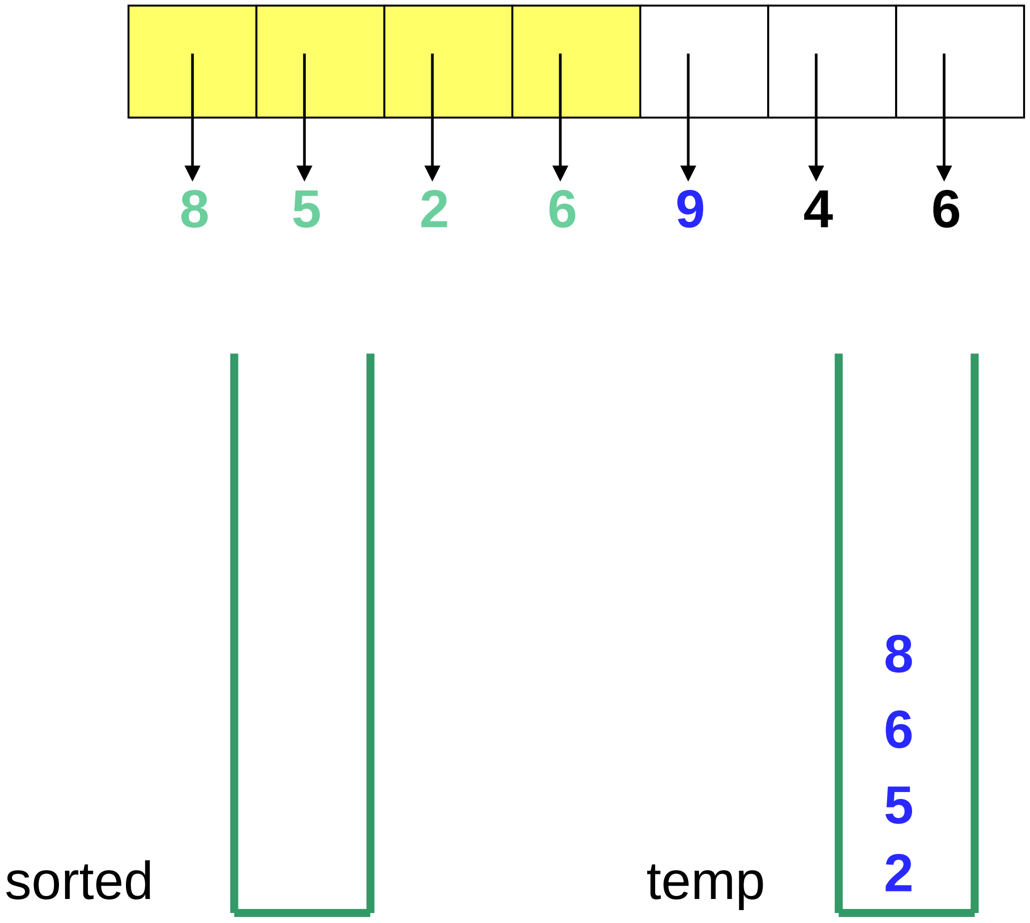
Insertion Sort



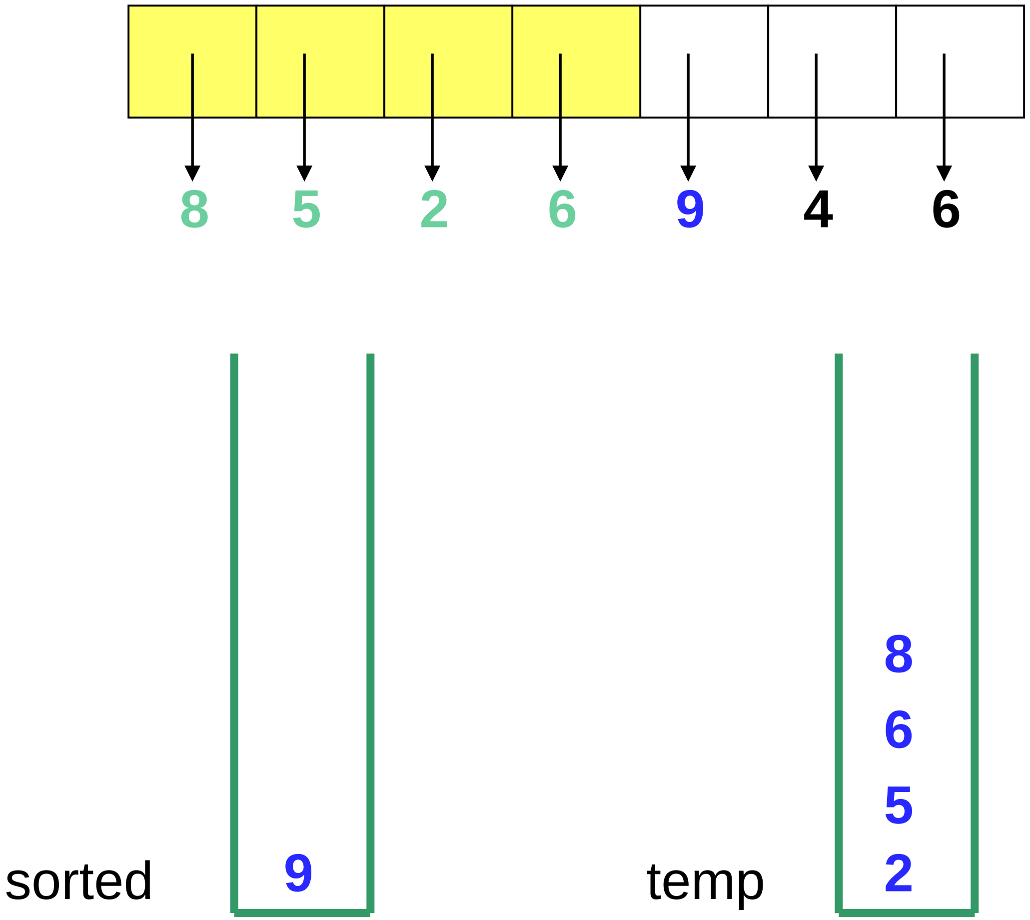
Insertion Sort



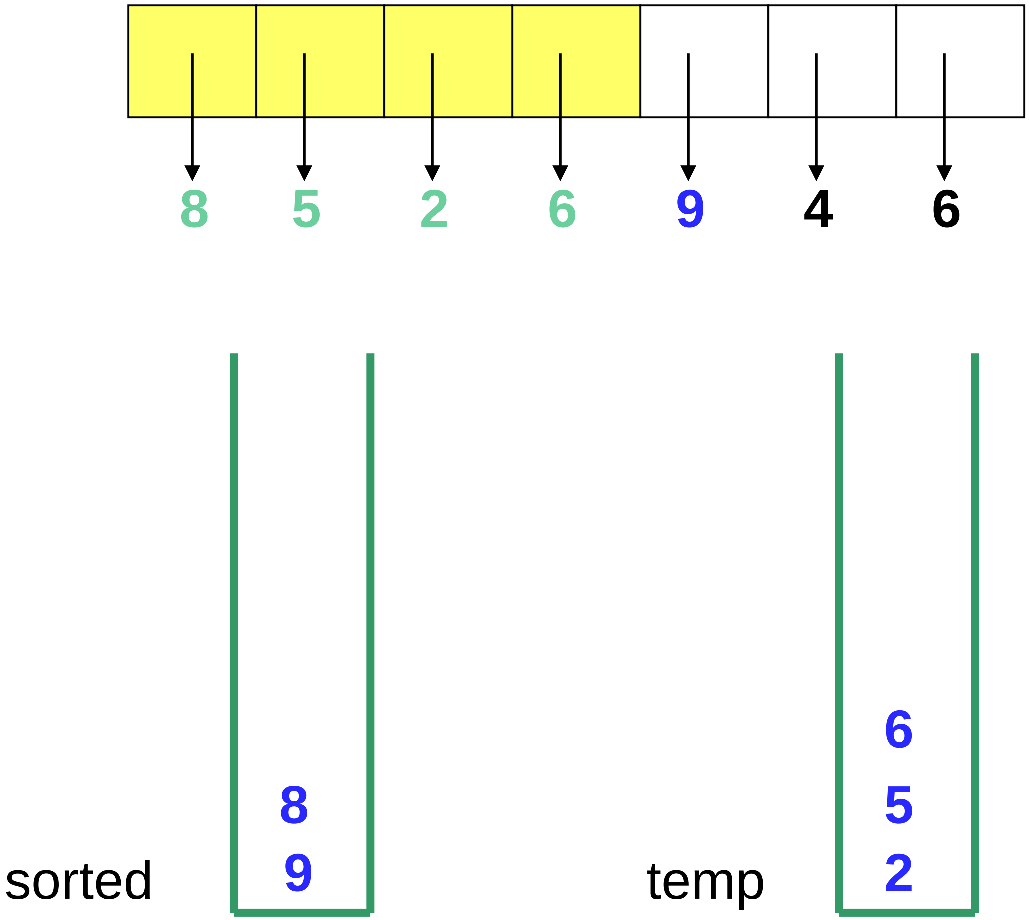
Insertion Sort



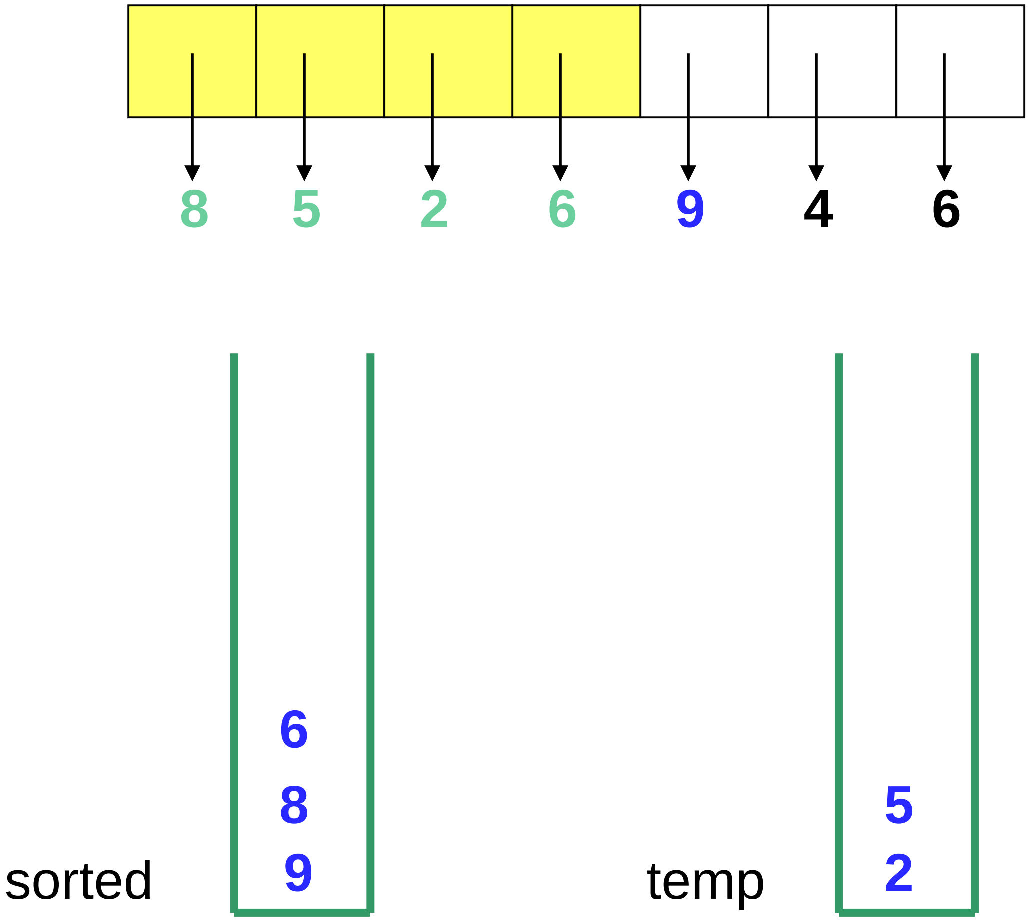
Insertion Sort



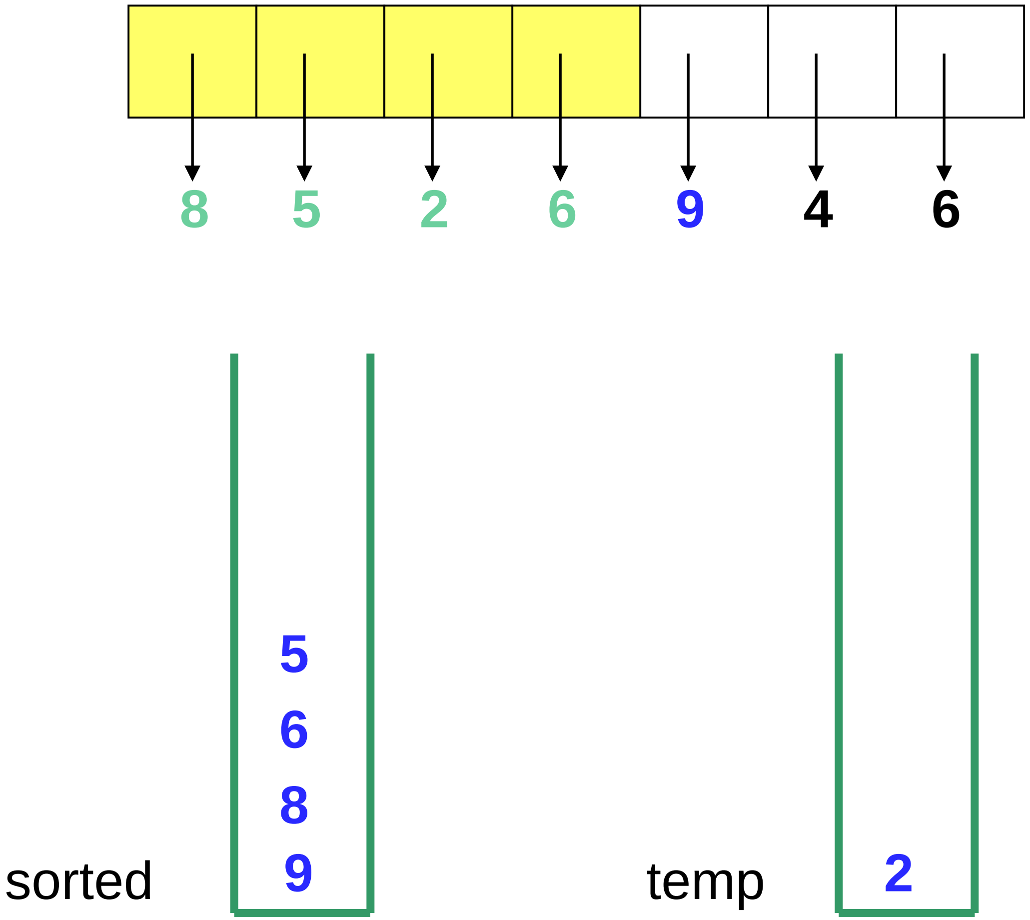
Insertion Sort



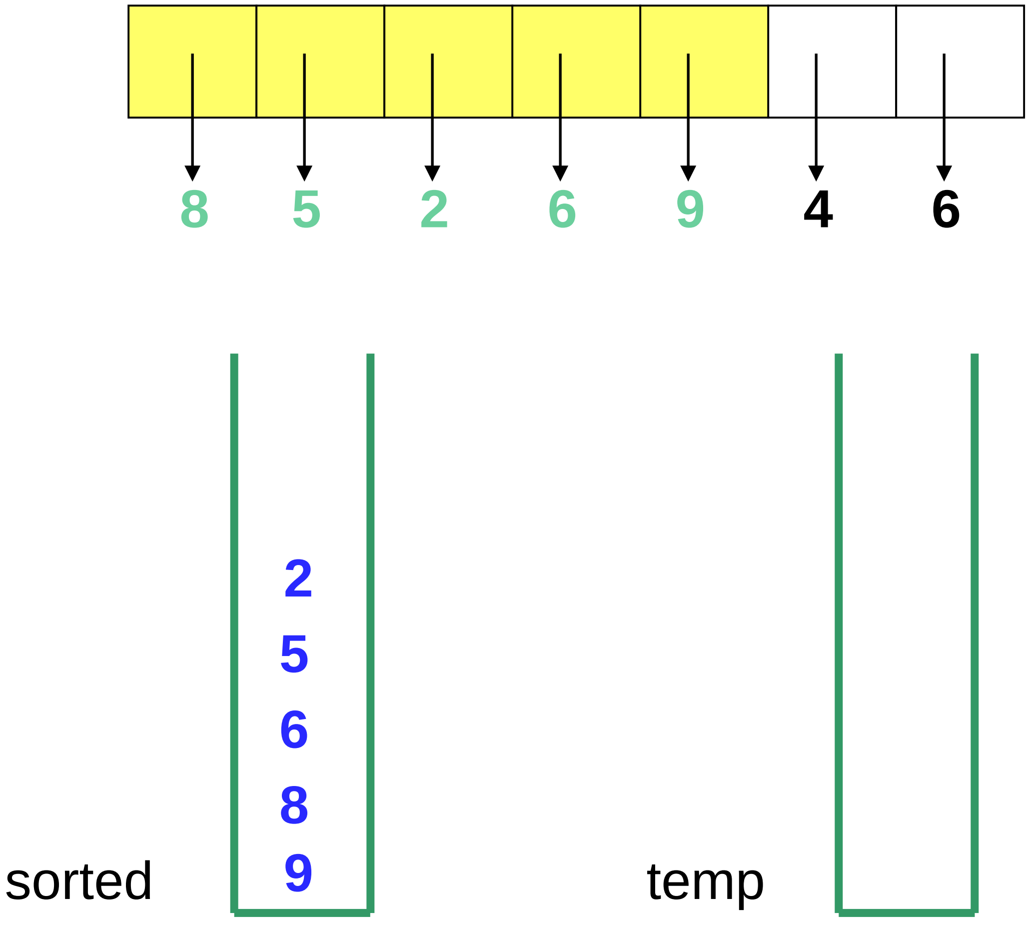
Insertion Sort



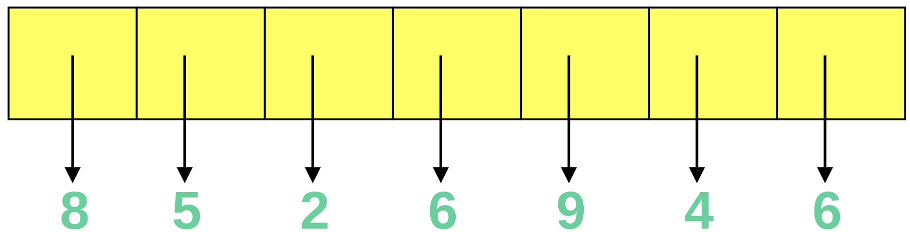
Insertion Sort



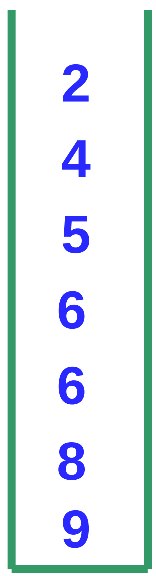
Insertion Sort



Insertion Sort

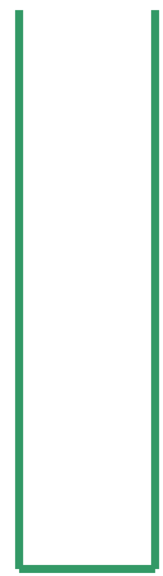


sorted



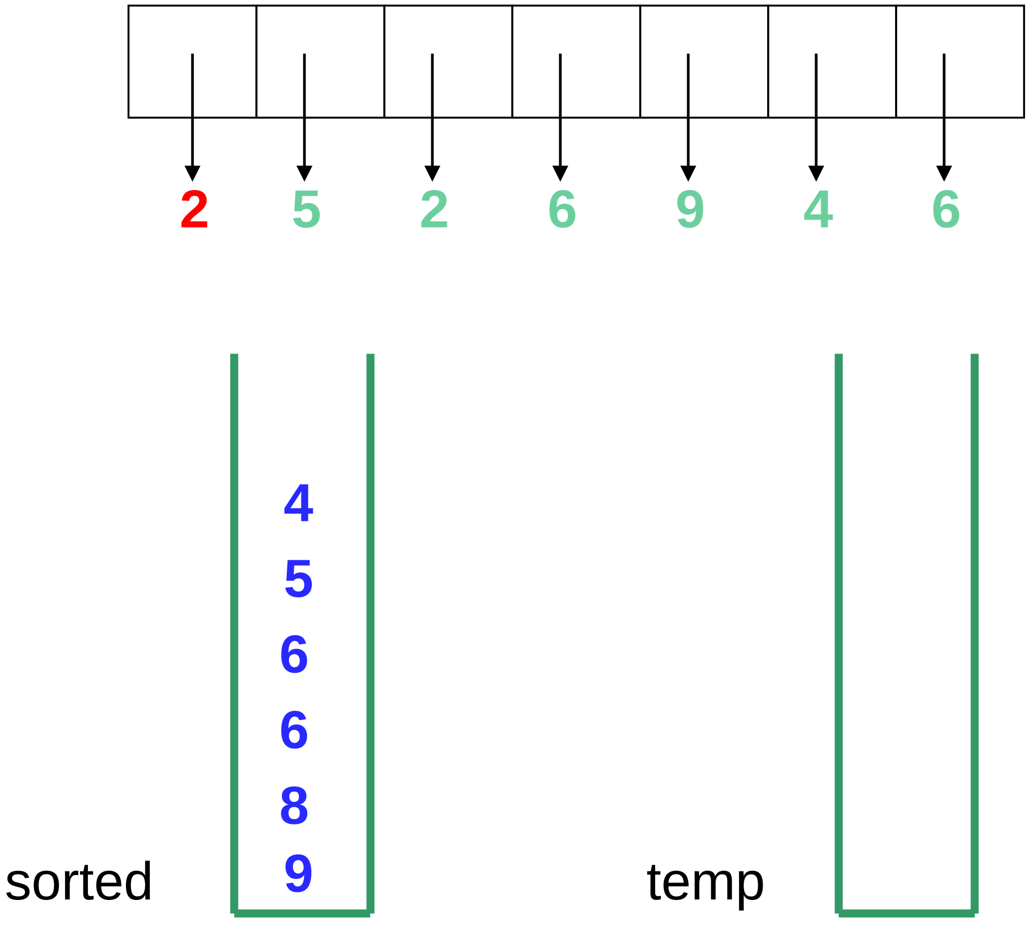
… and so on until all values are stored and ordered in stack sorted

temp



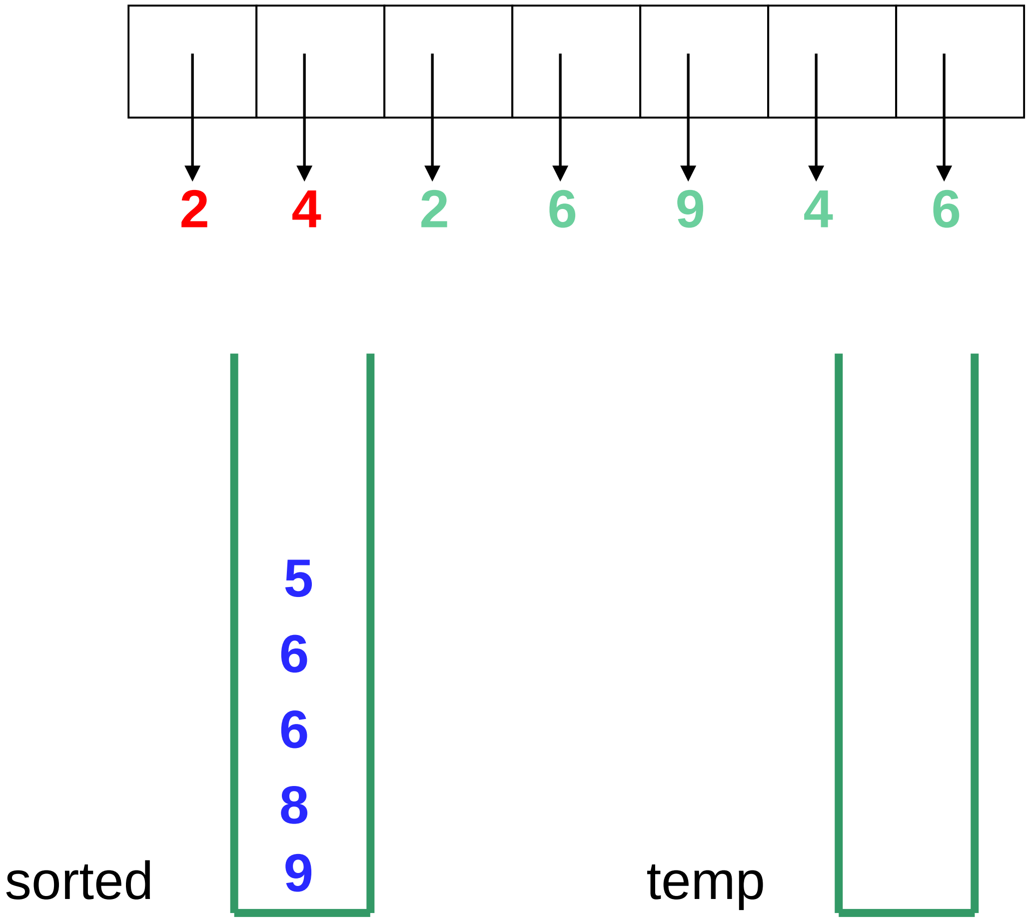
Insertion Sort

Now, copy the values back into the array…



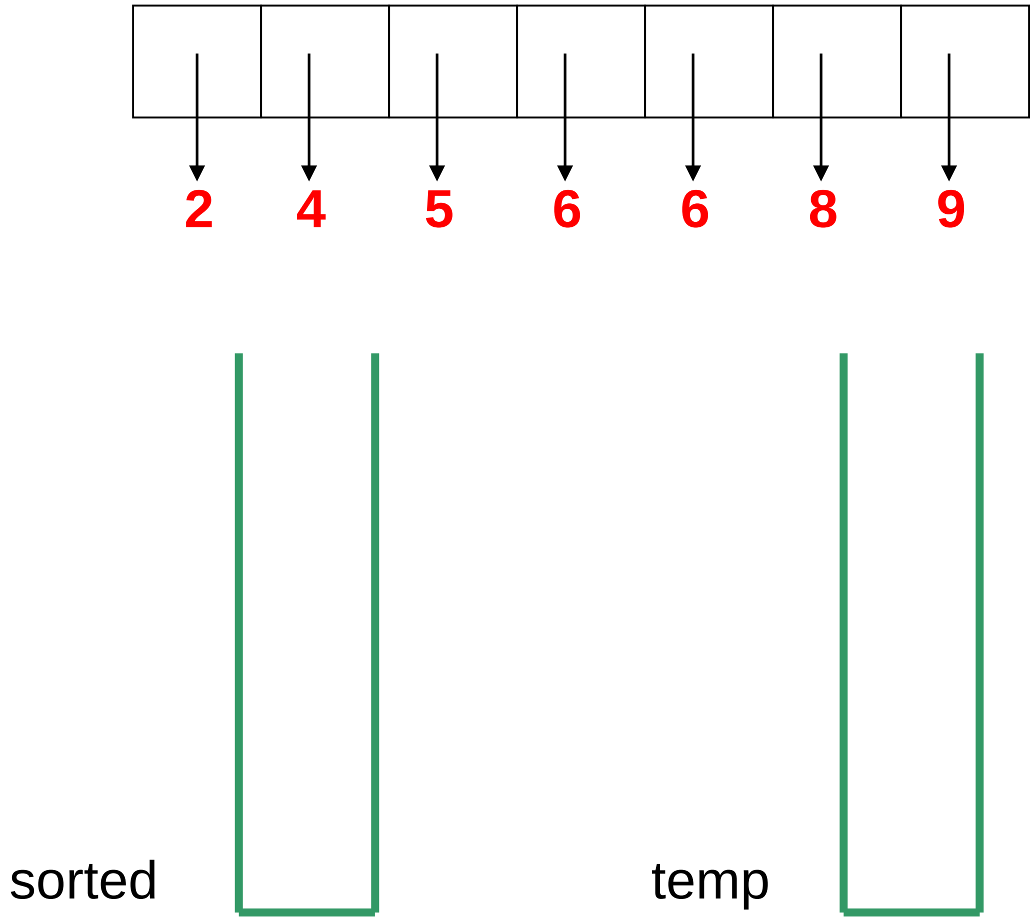
Insertion Sort

Now, copy the values back into the array…



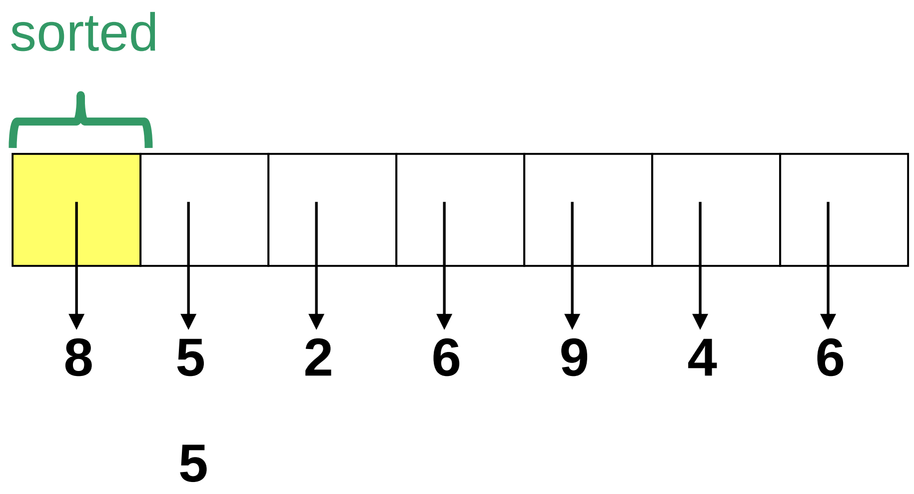
Insertion Sort

Now, copy the values back into the array…



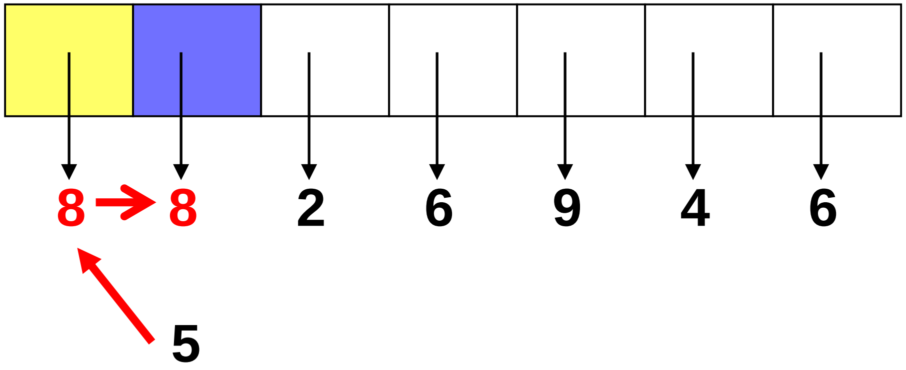
In-Place Insertion Sort

In-Place: the algorithm does not use any auxiliary data structures.



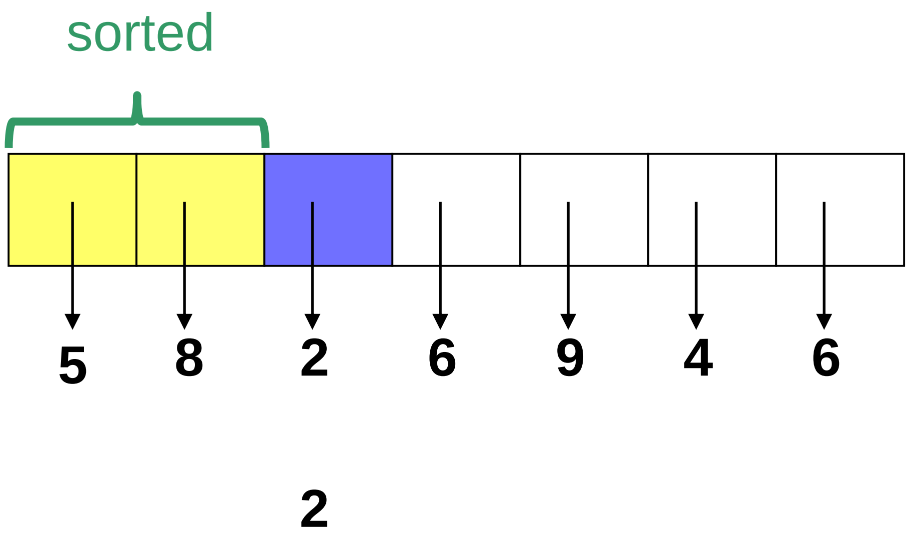
Consider the next value: 5

In-Place Insertion Sort



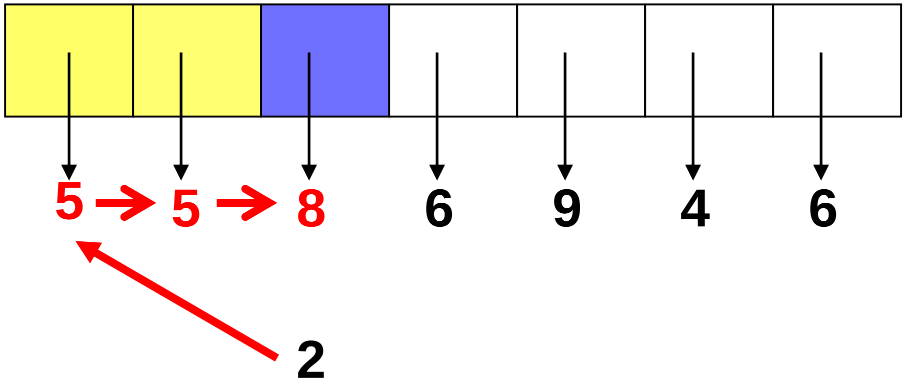
Shift 8 to make room for 5

In-Place Insertion Sort



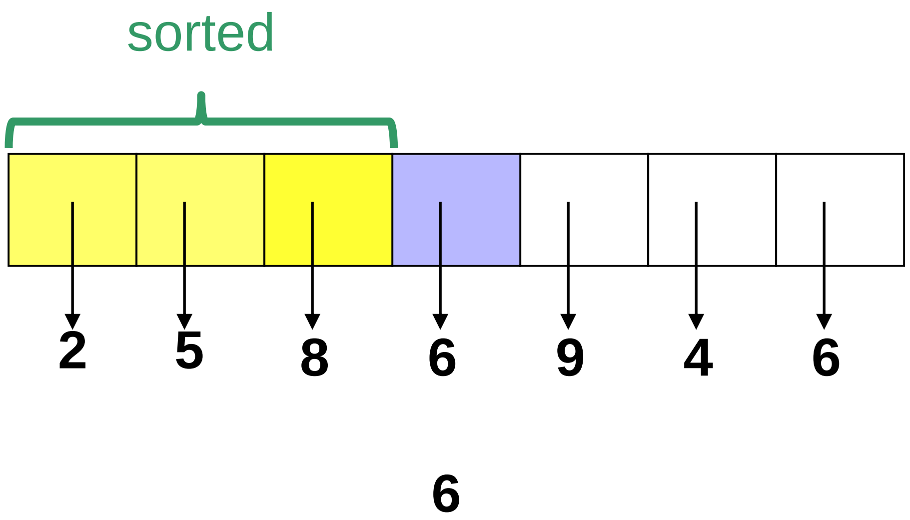
Consider the next value: 2

In-Place Insertion Sort



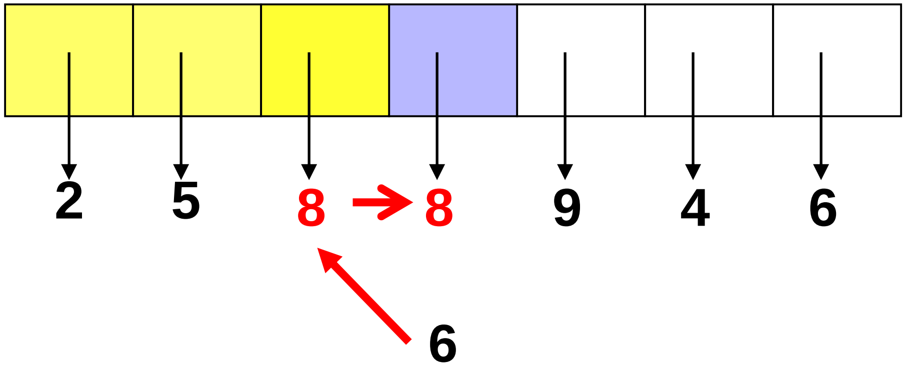
Shift 8 and 5 to the right

In-Place Insertion Sort



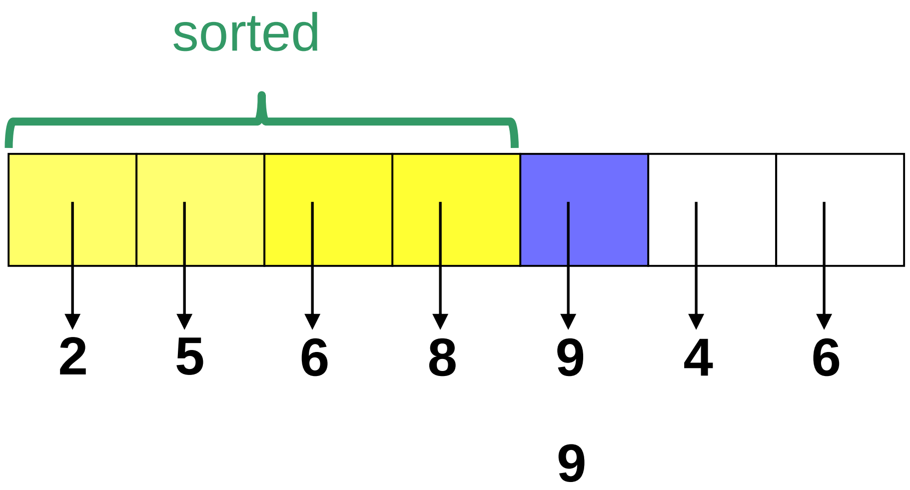
Consider the next value: 6

In-Place Insertion Sort



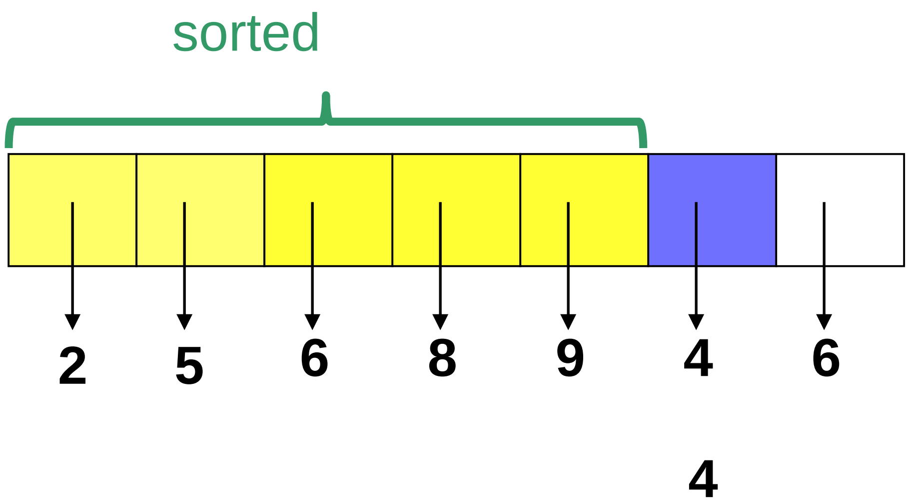
Shift 8 to the right

In-Place Insertion Sort



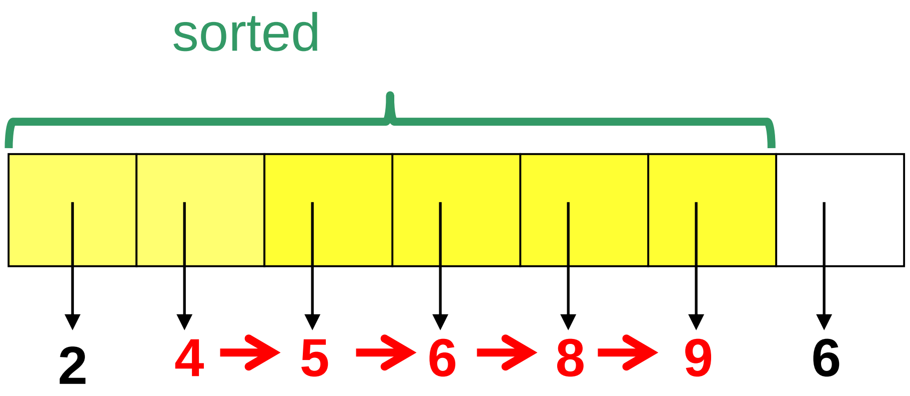
9 is already in its correct position

In-Place Insertion Sort



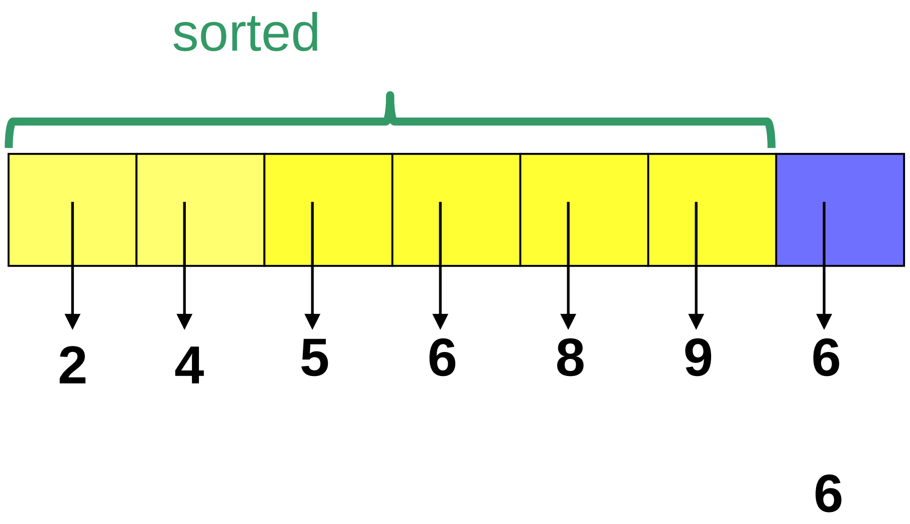
Consider the next value: 4

In-Place Insertion Sort



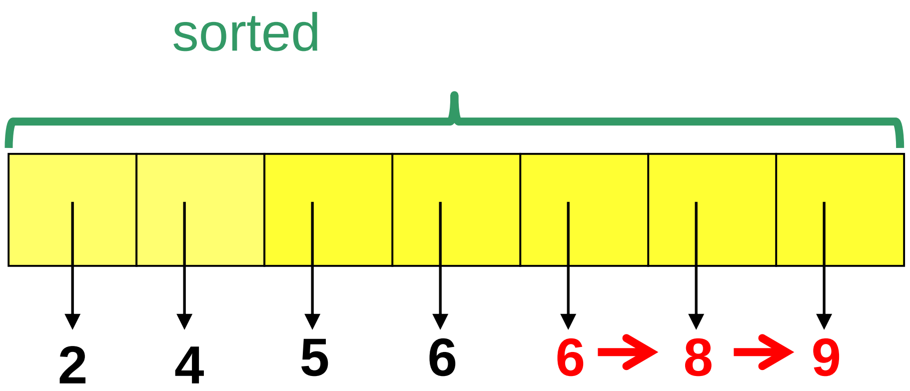
Shift 5, 6, 8, 9 to the right and
insert 4 in the second position

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!

```
Algorithm insertionSort (A,n)
In: Array A storing n values
Out: {Sort A in increasing order}
for i = 1 to n-1 do {
   // Insert A[i] in the sorted sub-array A[0..i-1]
    temp = A[i]
    j = i – 1
    while (j >= 0) and (A[j] > temp) do {
        A[j+1] = A[j]
        j = j – 1
    }
    A[j+1] = temp
}
```

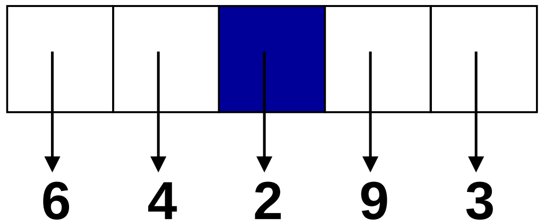
Selection Sort

- ***Selection Sort*** orders a sequence of values by repetitively putting a particular value into its ***final*** position
- 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

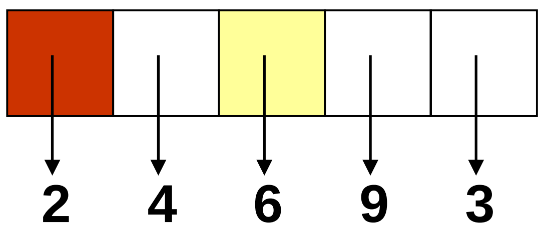
Selection Sort Algorithm

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

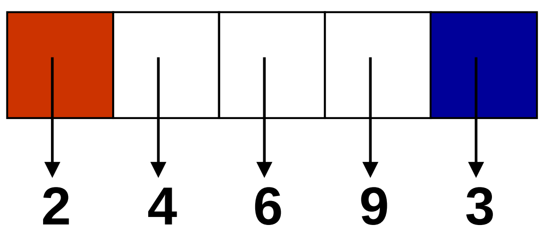
The sorted portion is in red.



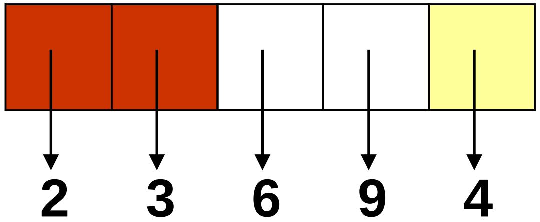
Find the smallest element in the unsorted portion of the array



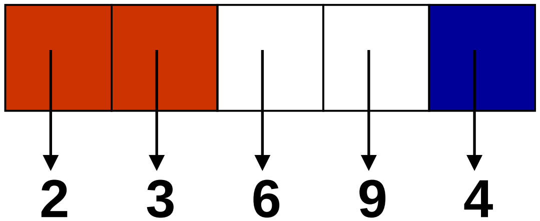
Interchange the smallest element with the one at the first position of the array



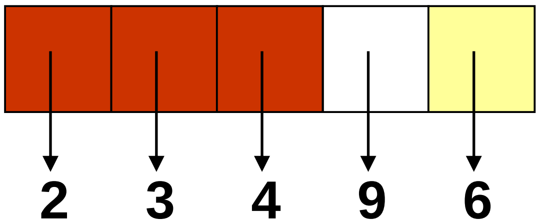
Find the smallest element in the unsorted portion of the array



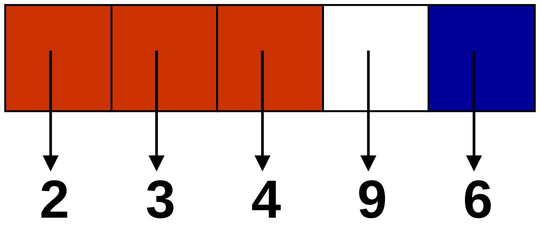
Interchange the smallest element with the one at the second position



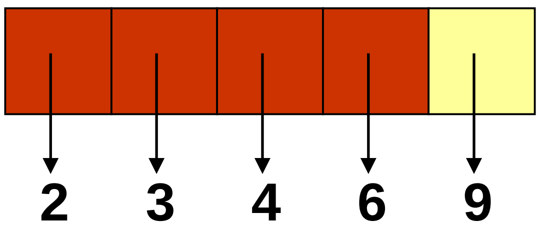
Find the smallest element in the unsorted portion



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

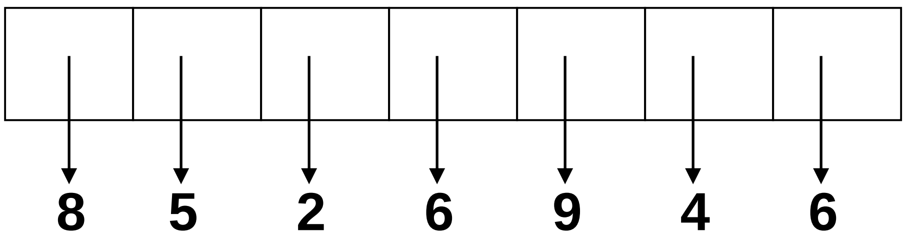
Selection Sort Using Queue Algorithm

- 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

```
Algorithm selectionSort(list)
In: Unsorted list
Out: Sorted list
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 {
        item = get item in the i-th position of list
        if item < smallestSoFar then smallestSoFar = item
    }
    sorted.enqueue(smallestSoFar)
    remove smallestSoFar from list
    n = n - 1
}
for i = 0 to n – 1 do
    insert sorted.dequeue() in the i-th position of list
return list
```

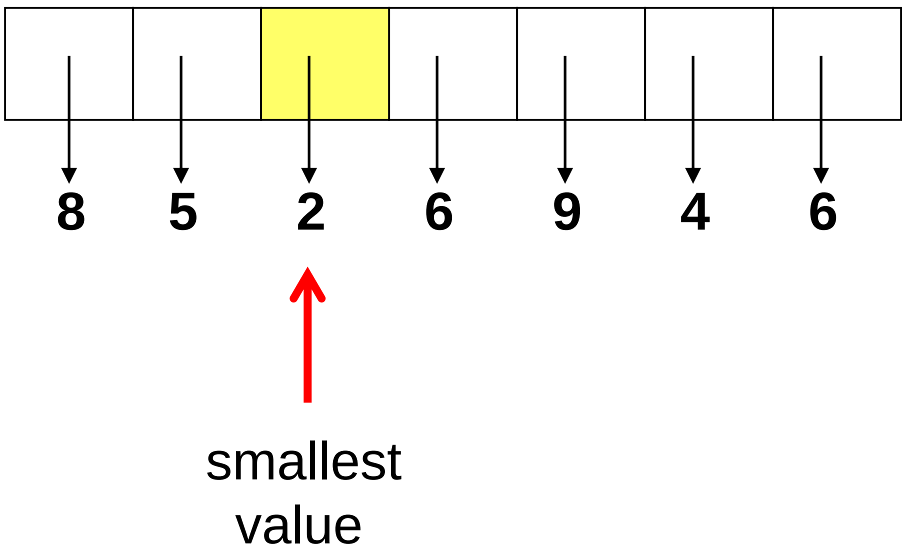
In-Place SelectionSort

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



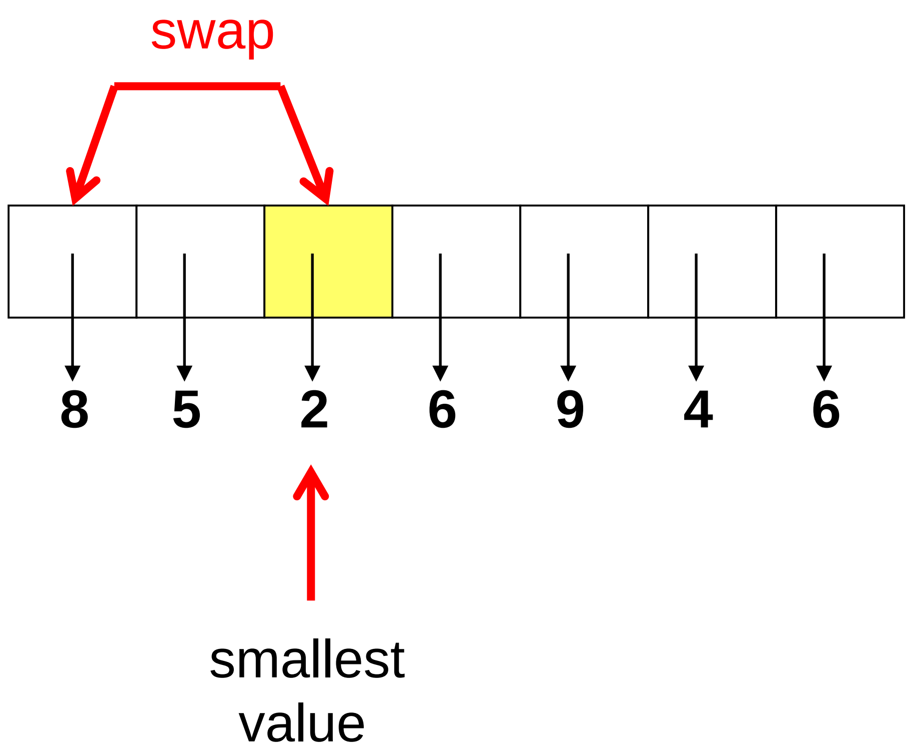
In-Place SelectionSort

First, find the smallest value



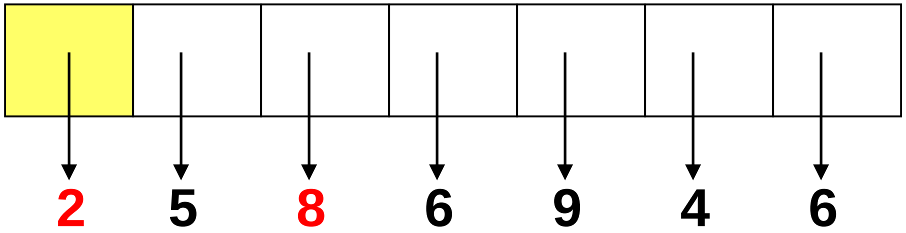
In-Place SelectionSort

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

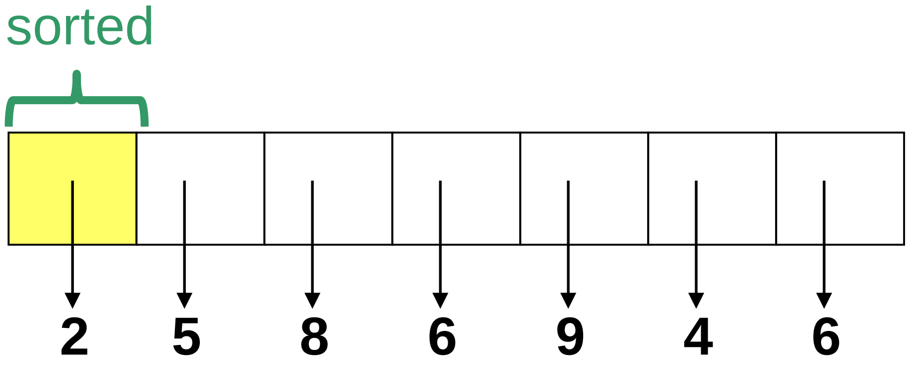


In-Place SelectionSort

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

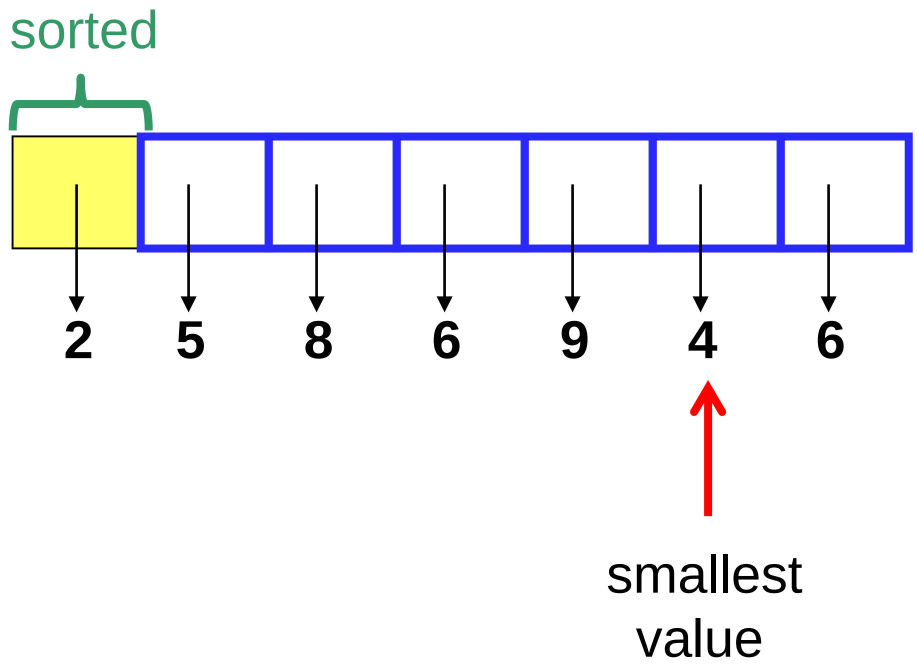


In-Place SelectionSort



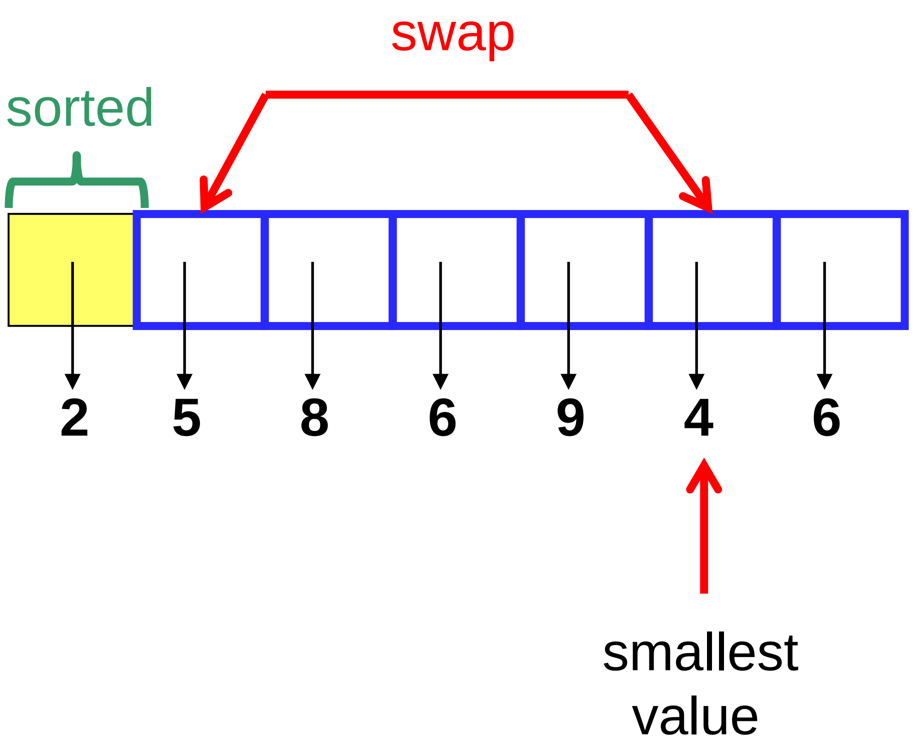
In-Place SelectionSort

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

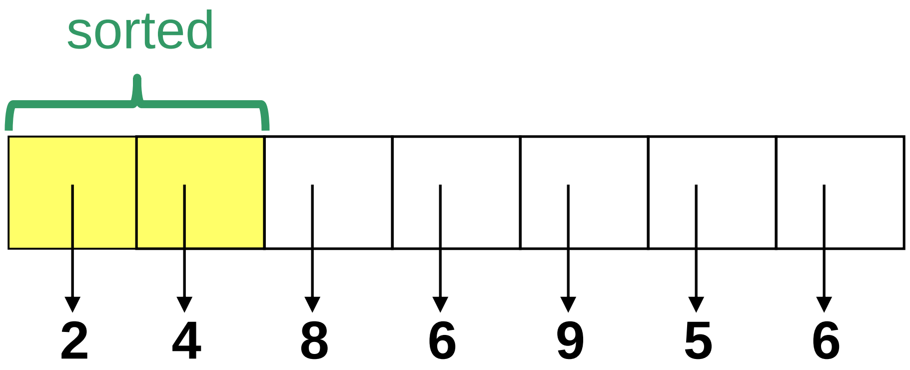


In-Place SelectionSort

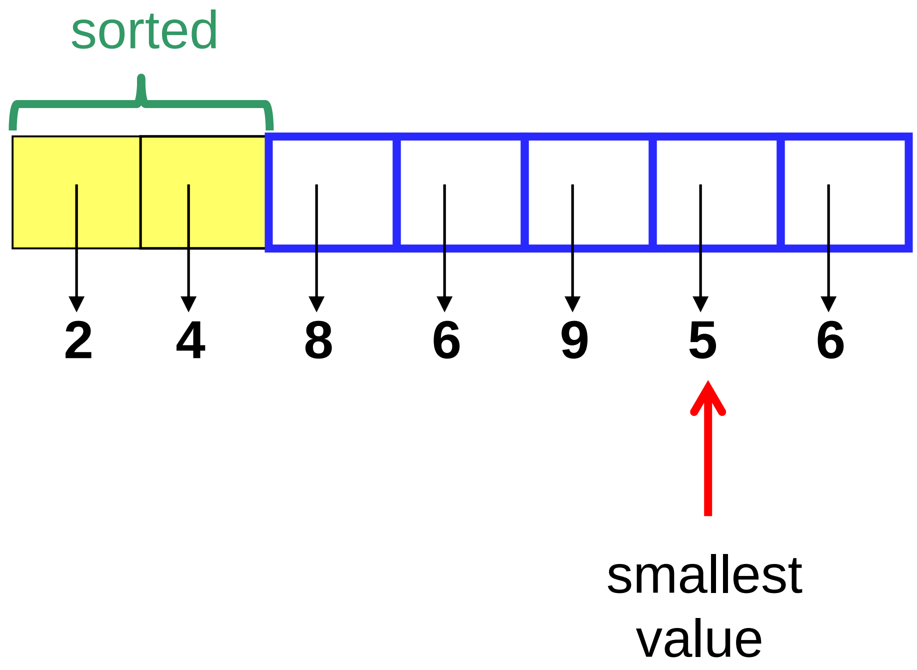
Swap it with the element in the second position of the array, and so on.



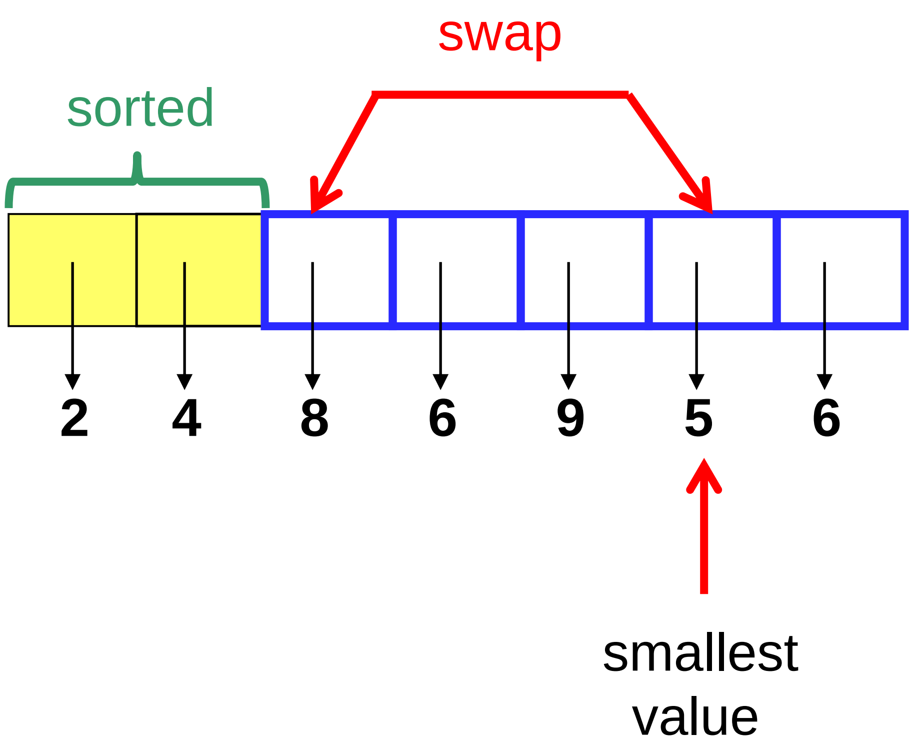
In-Place SelectionSort



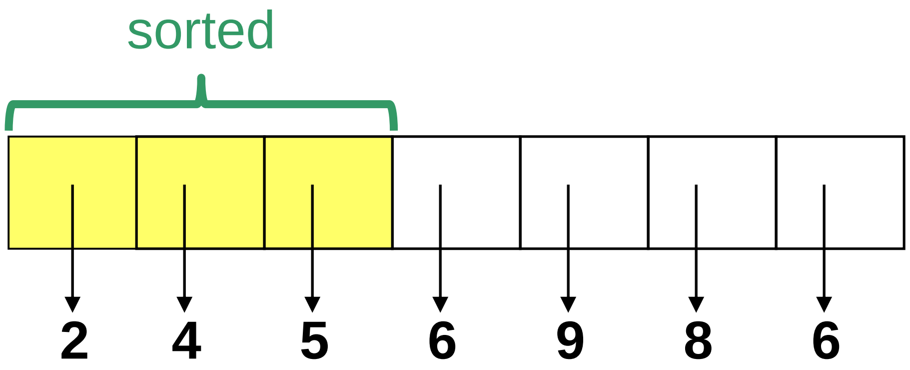
In-Place SelectionSort



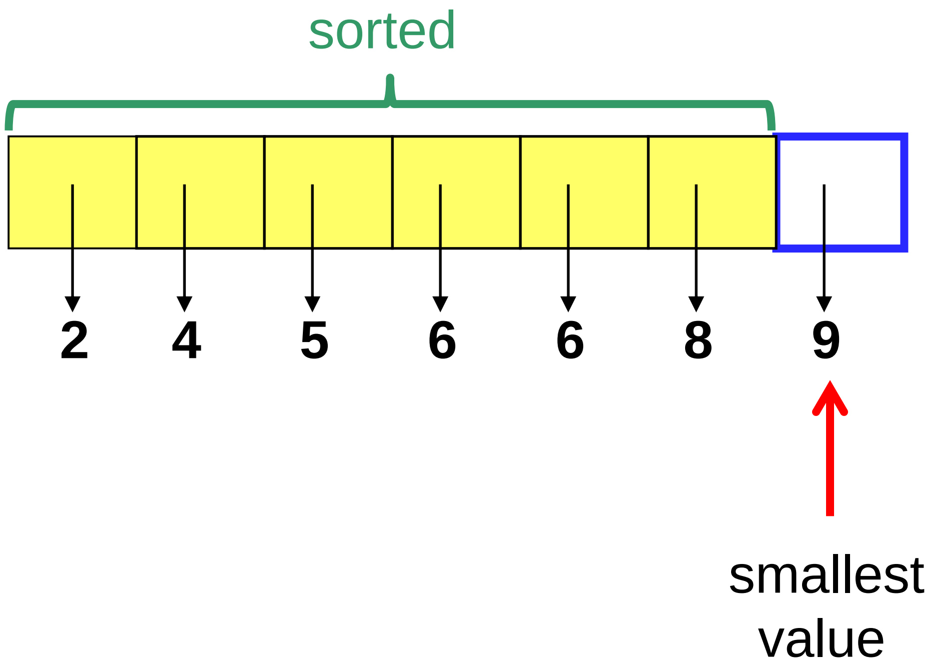
In-Place SelectionSort



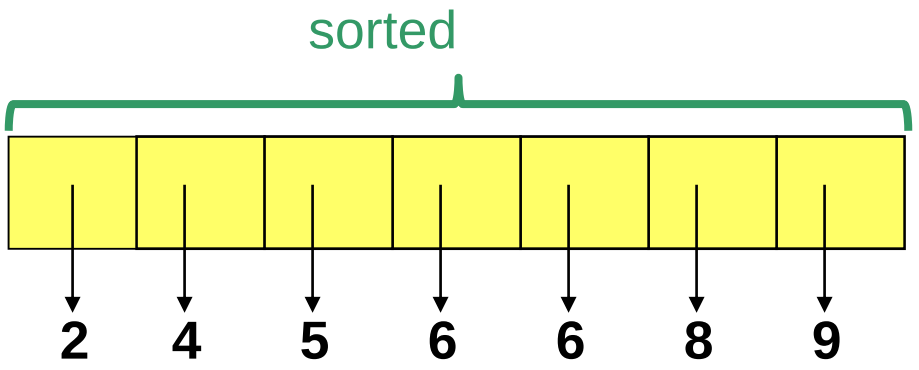
In-Place SelectionSort



In-Place SelectionSort



In-Place SelectionSort



```
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 - 1do {
        if A[j] < A[smallest] then
            smallest = j
    }
    // Swap A[smallest] and A[i]
    temp = A[smallest]
    A[smallest] = A[i]
    A[i] = temp
}
```

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

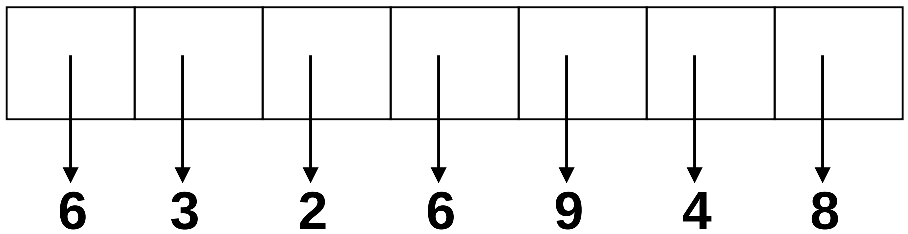
Partition element or ***pivot***:

- 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

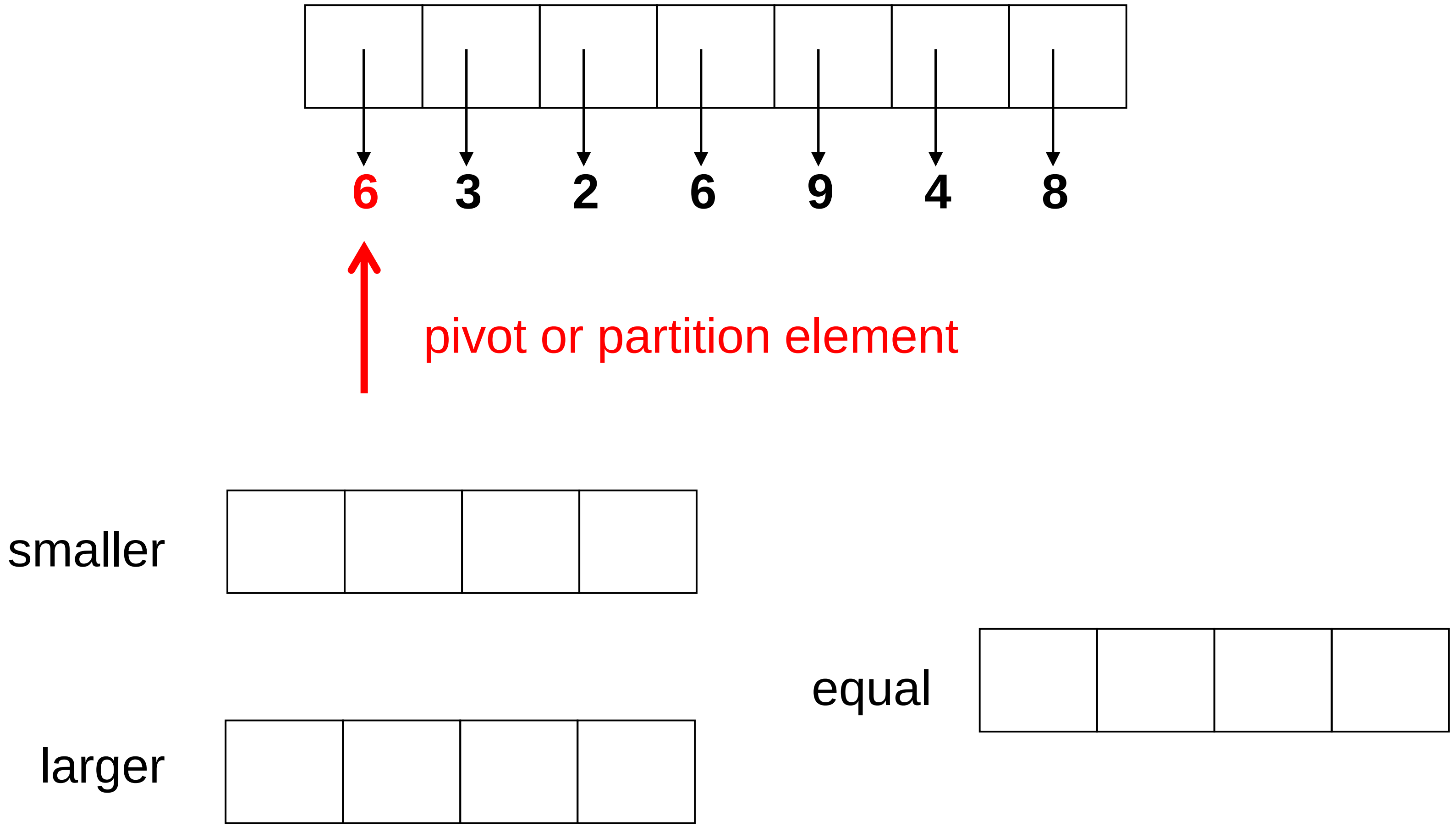
Quick Sort

- We put all the items to be sorted into a container (e.g. an array)
- We choose the pivot (partition element) as the first element from the container
- We 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

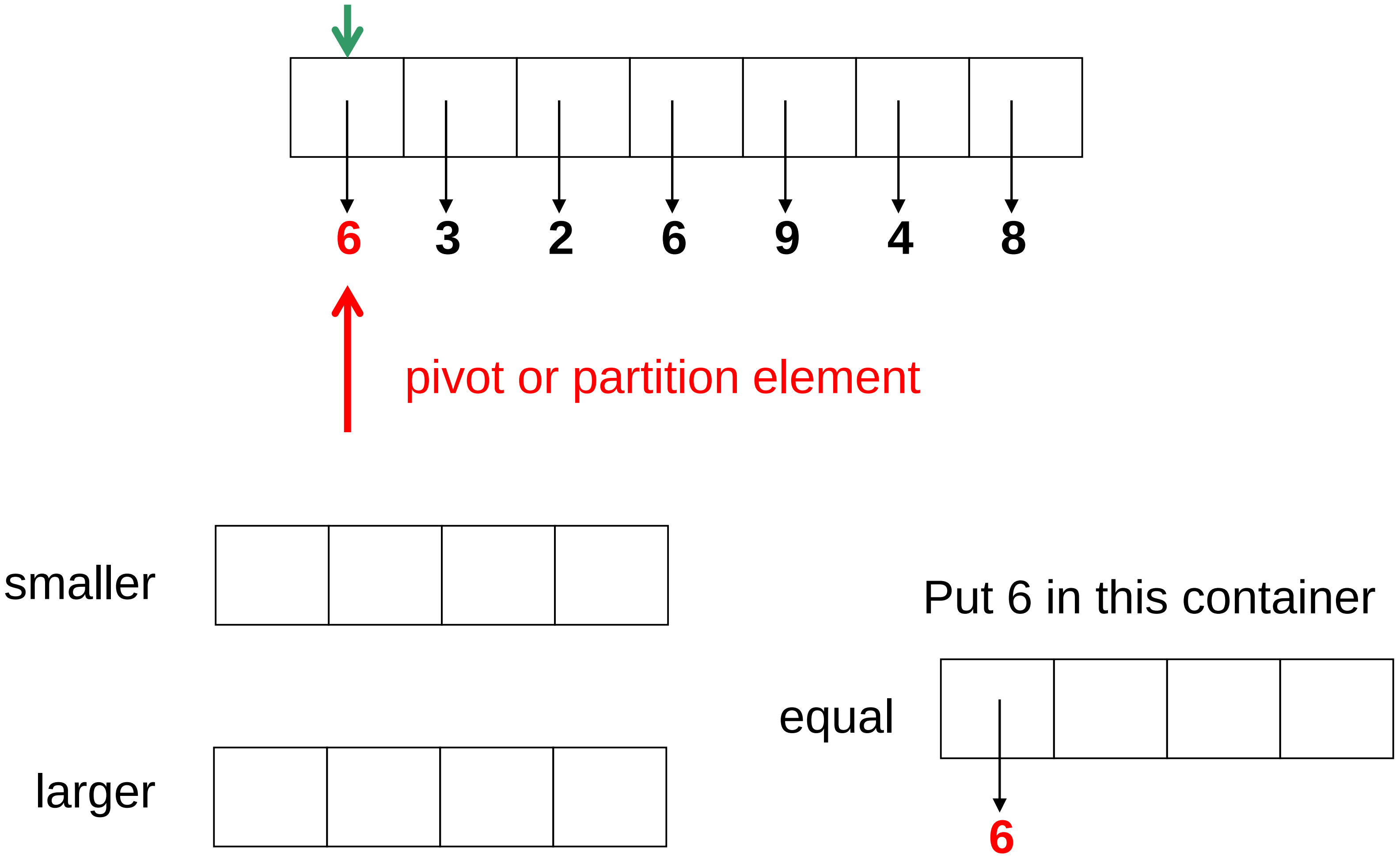
QuickSort



QuickSort



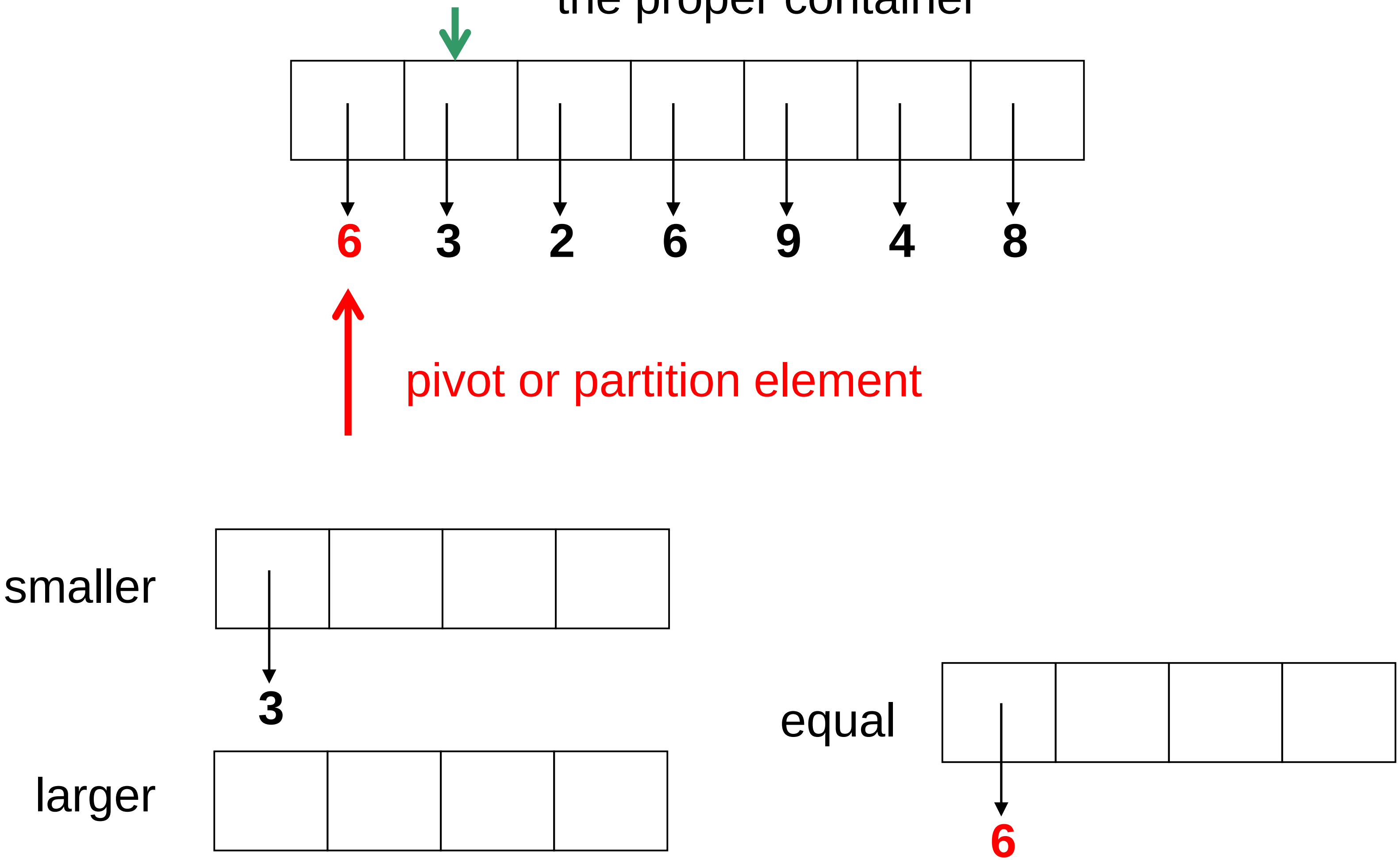
QuickSort



6
3
2
6
9
4
8
pivot or partition element
smaller
larger
Put 6 in this container
equal
6

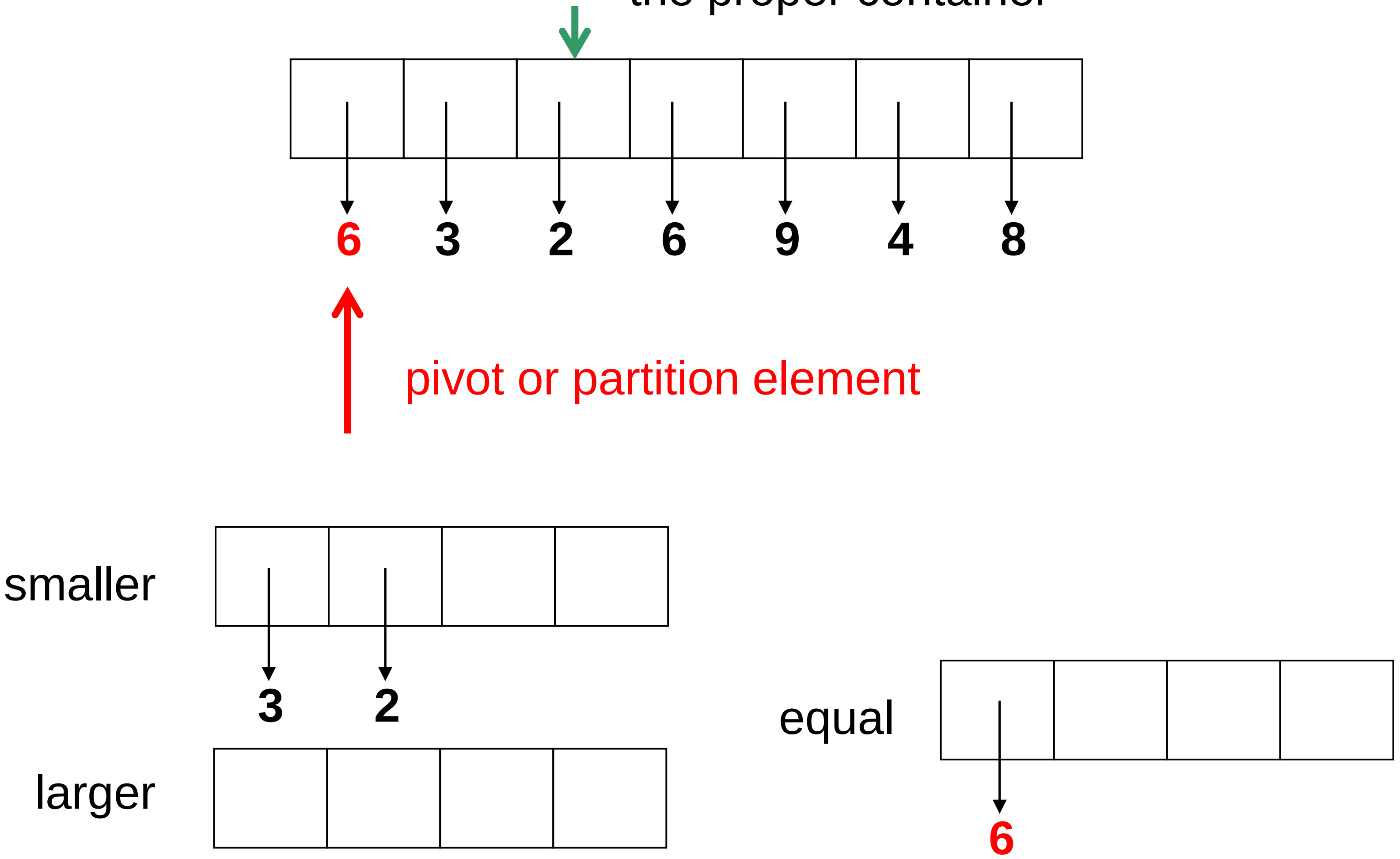
QuickSort

scan the array and place the values in the proper container



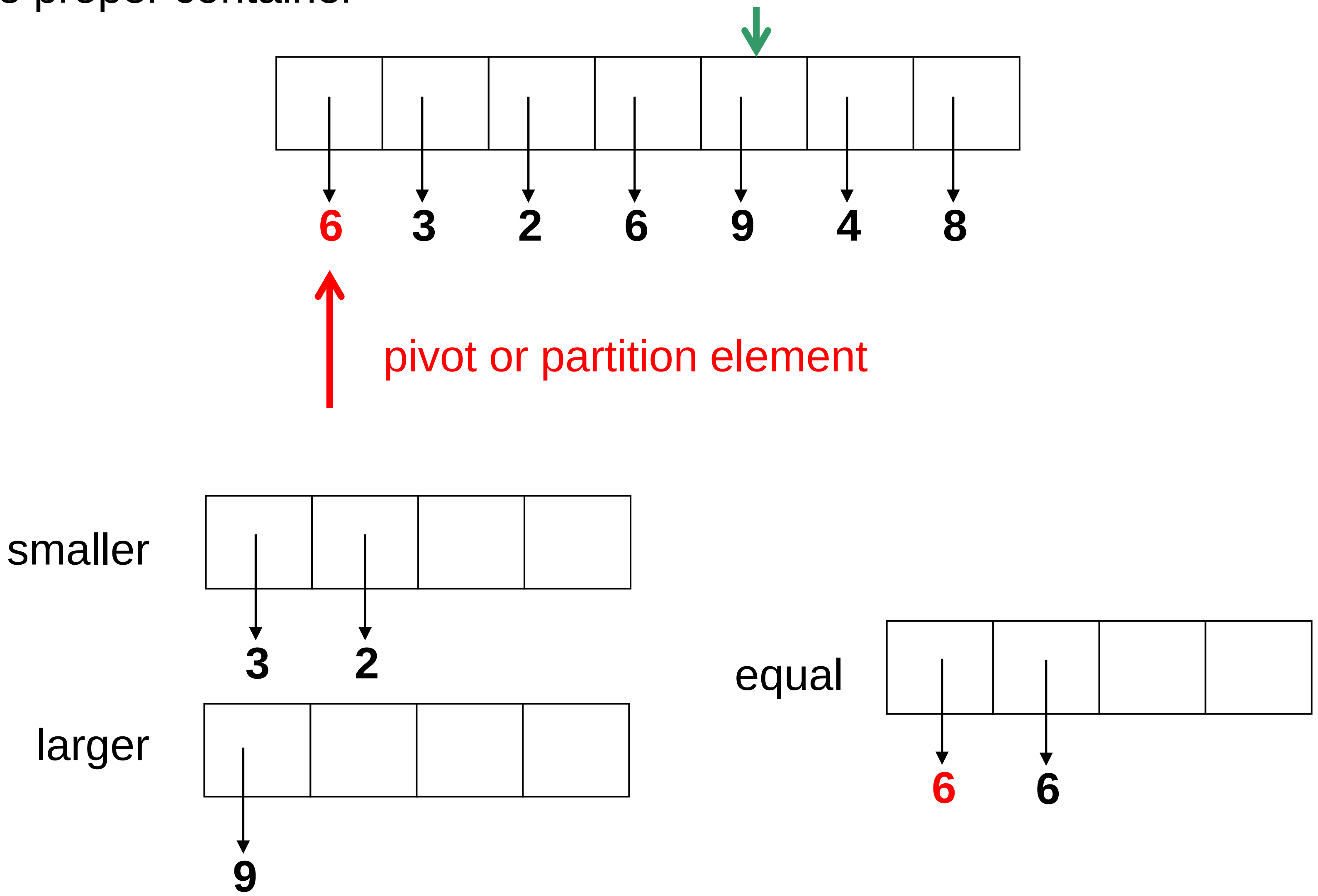
QuickSort

scan the array and place the values in the proper container



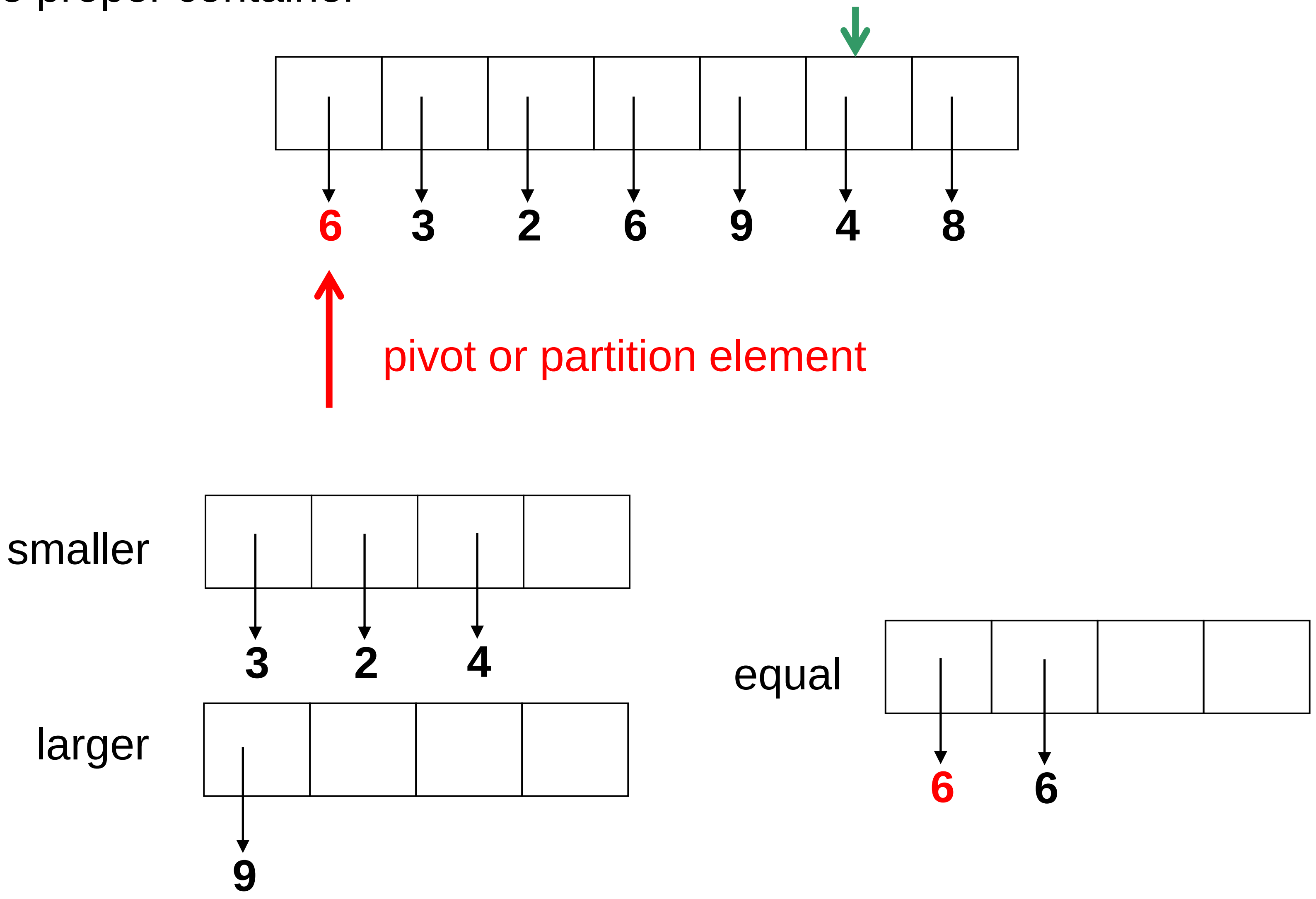
QuickSort

scan the array and place the values in the proper container



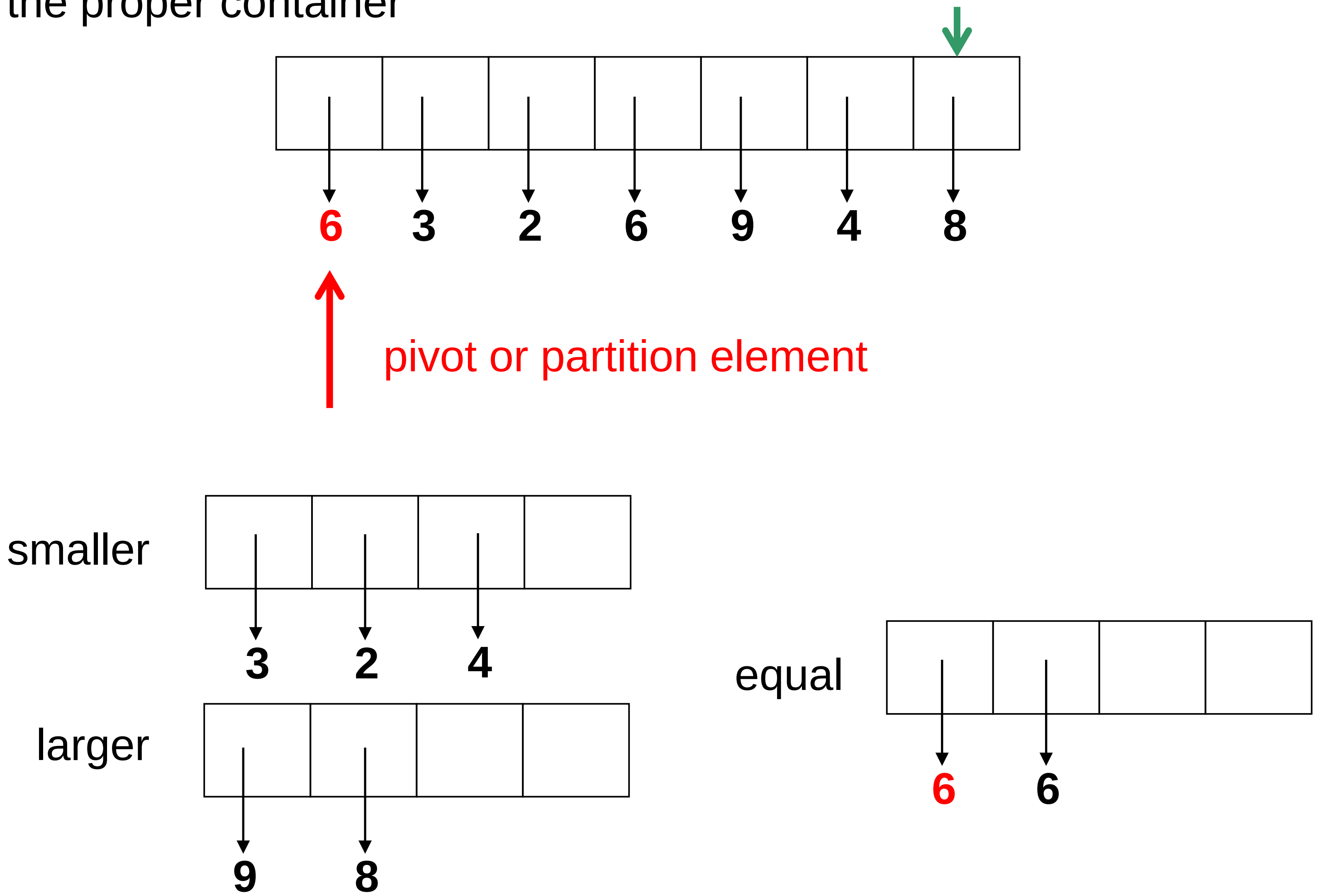
QuickSort

scan the array and place the values in the proper container

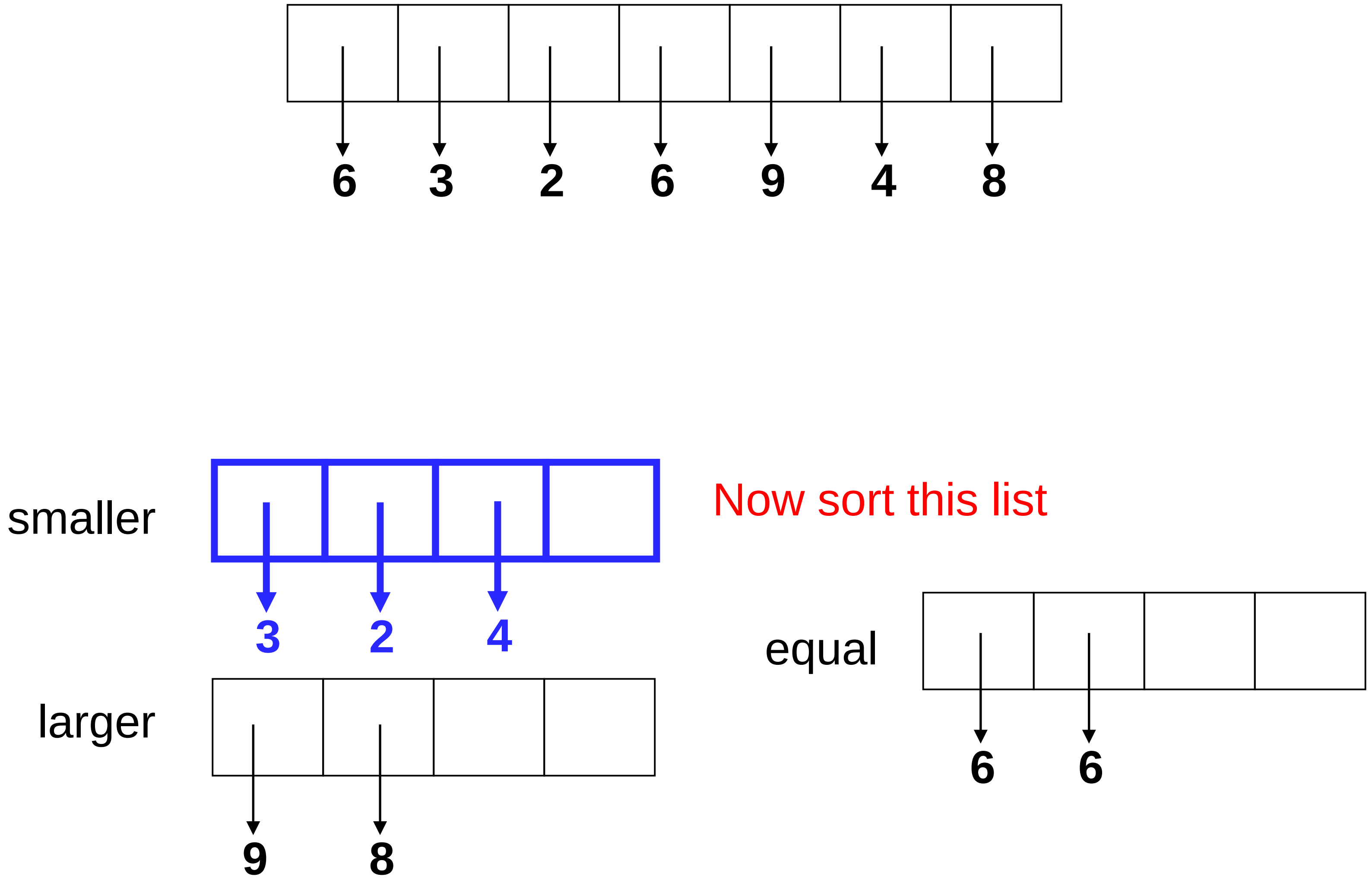


QuickSort

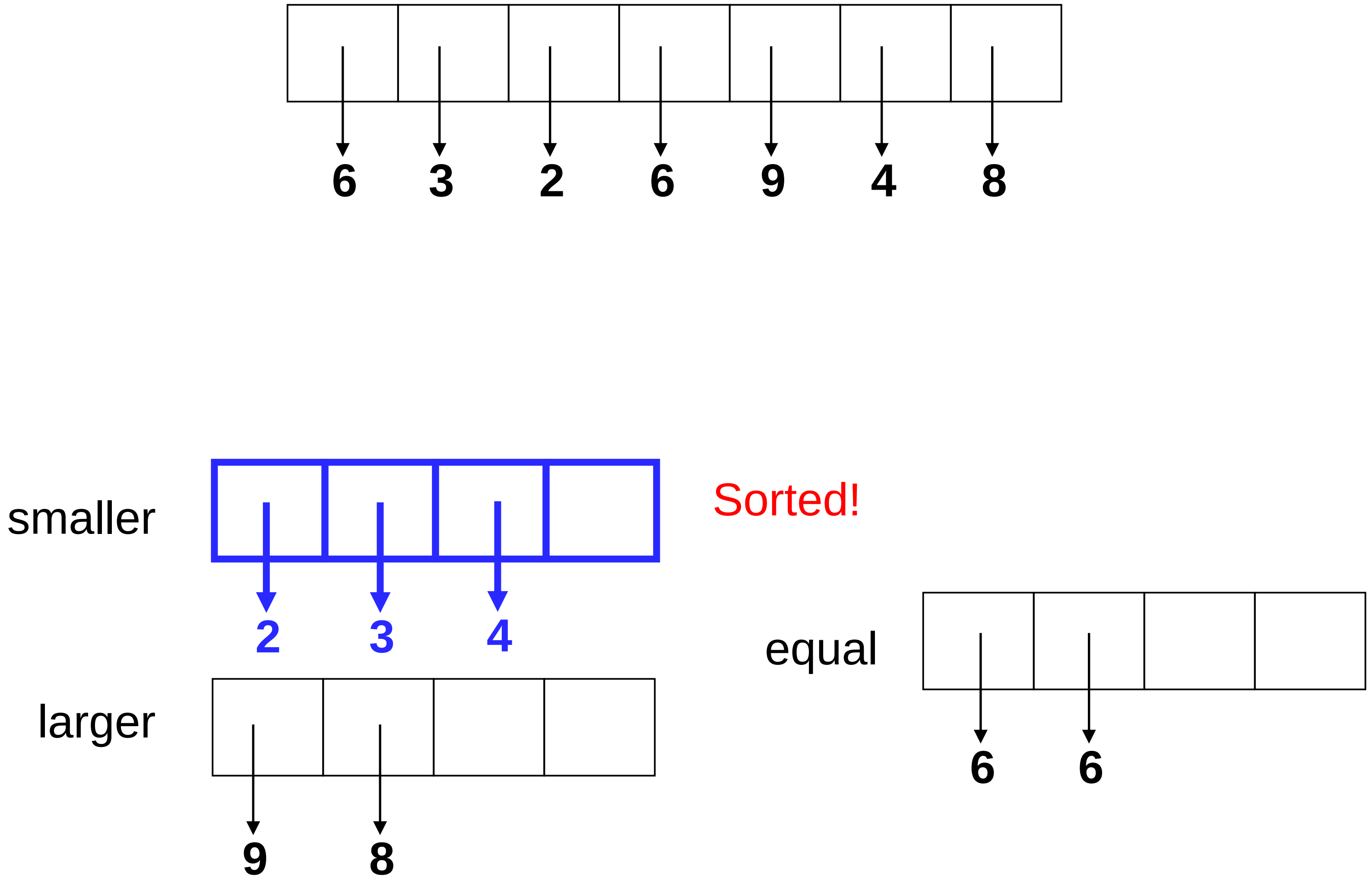
scan the array and place the values in the proper container



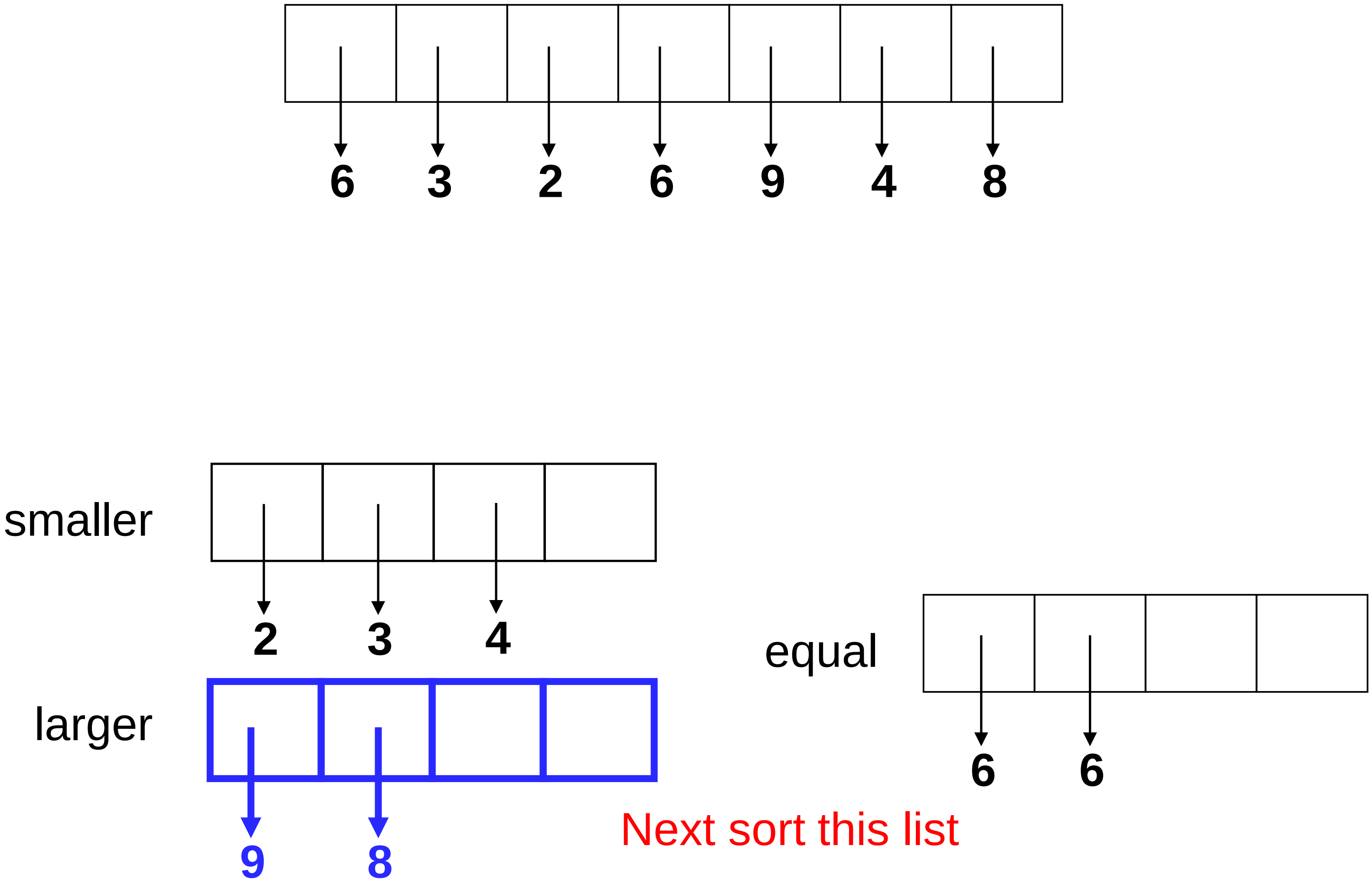
QuickSort



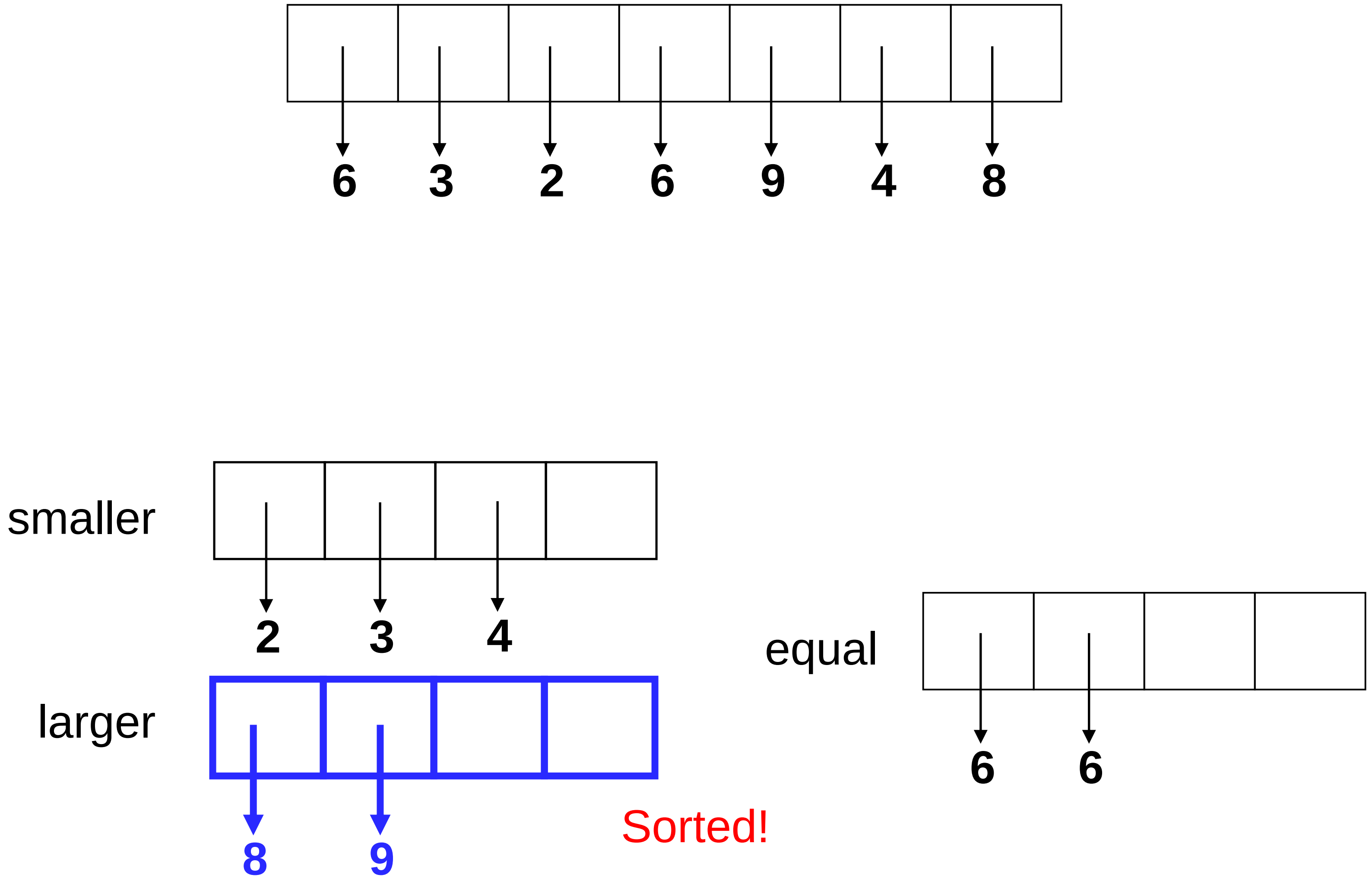
QuickSort



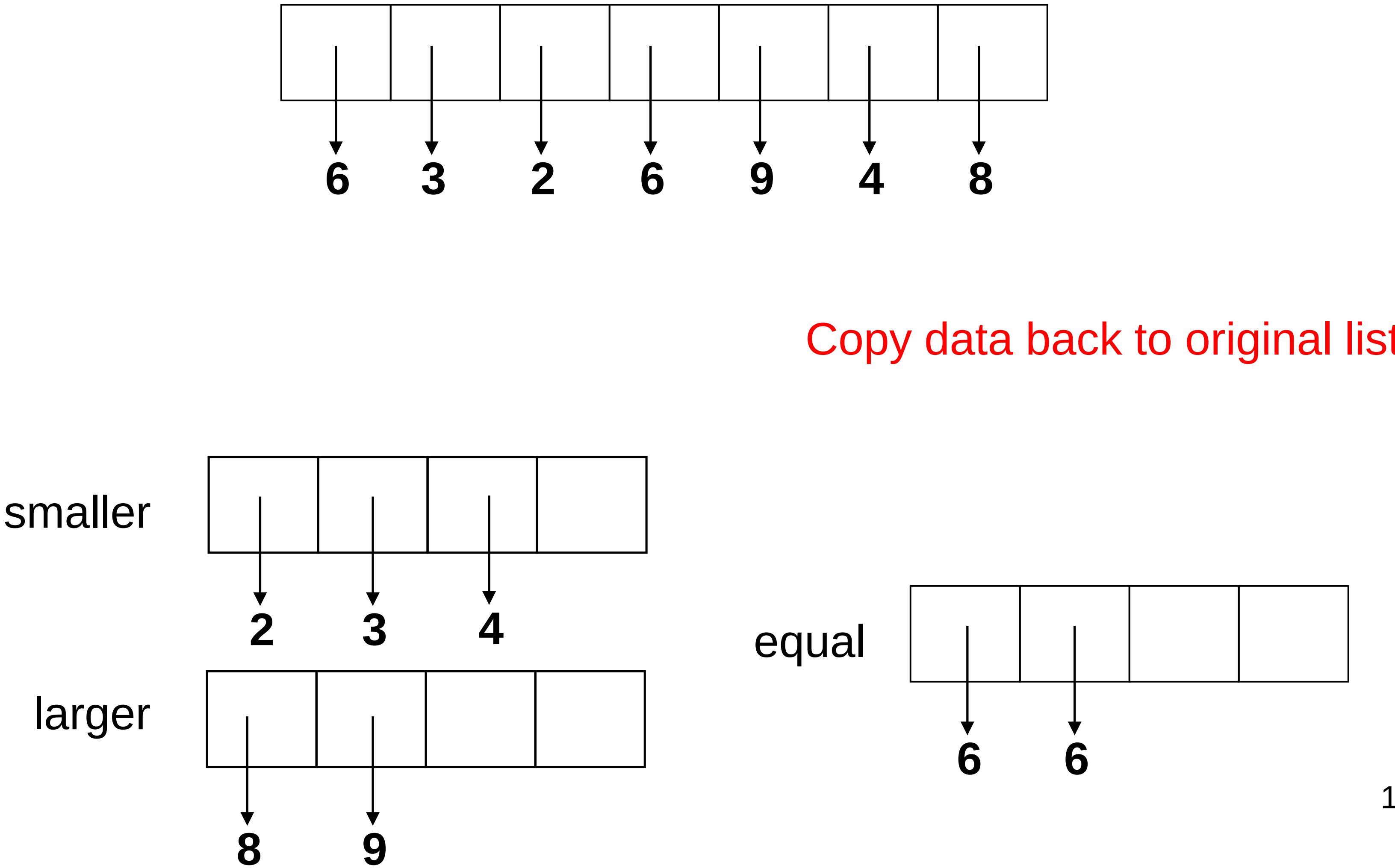
QuickSort



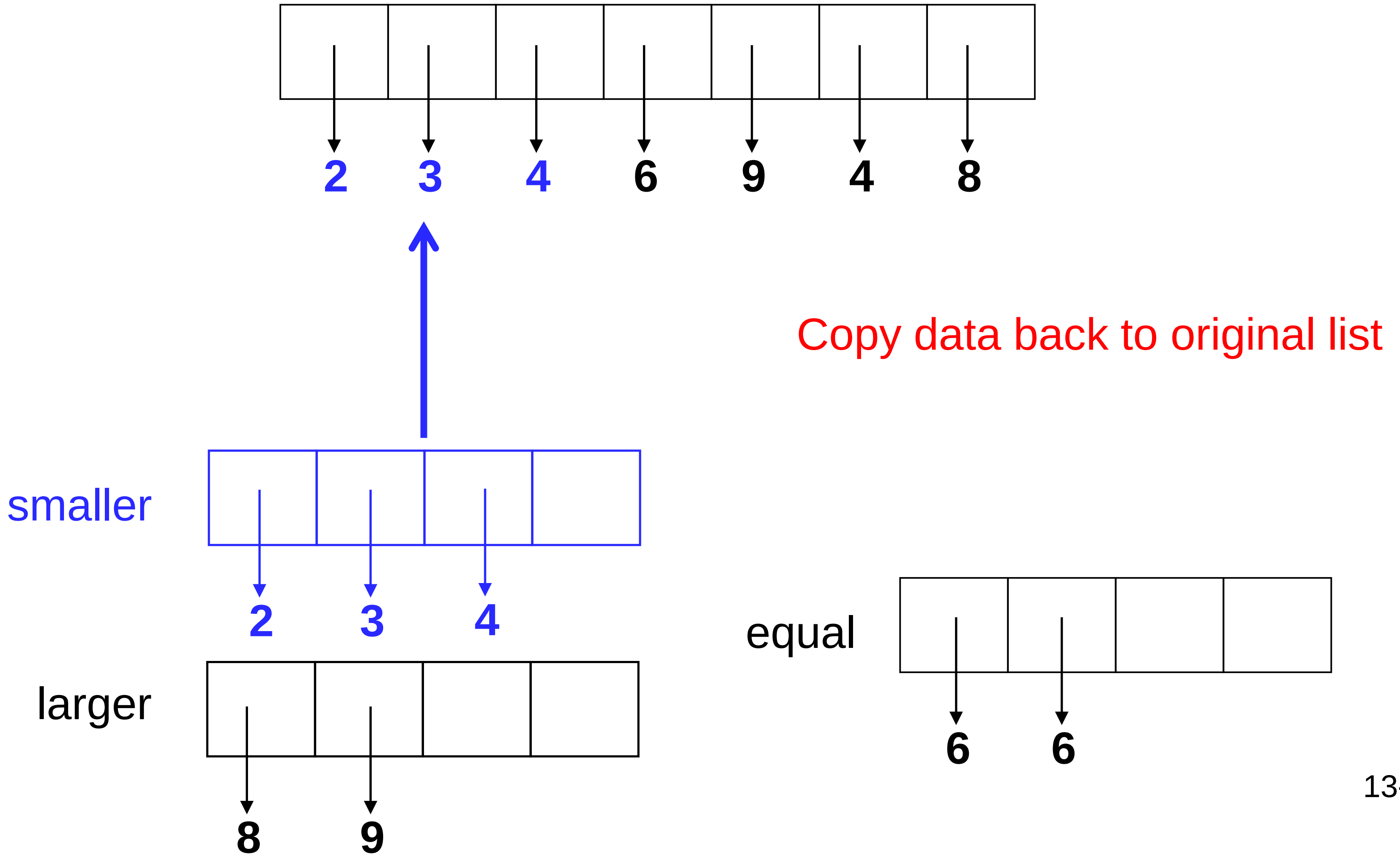
QuickSort



QuickSort

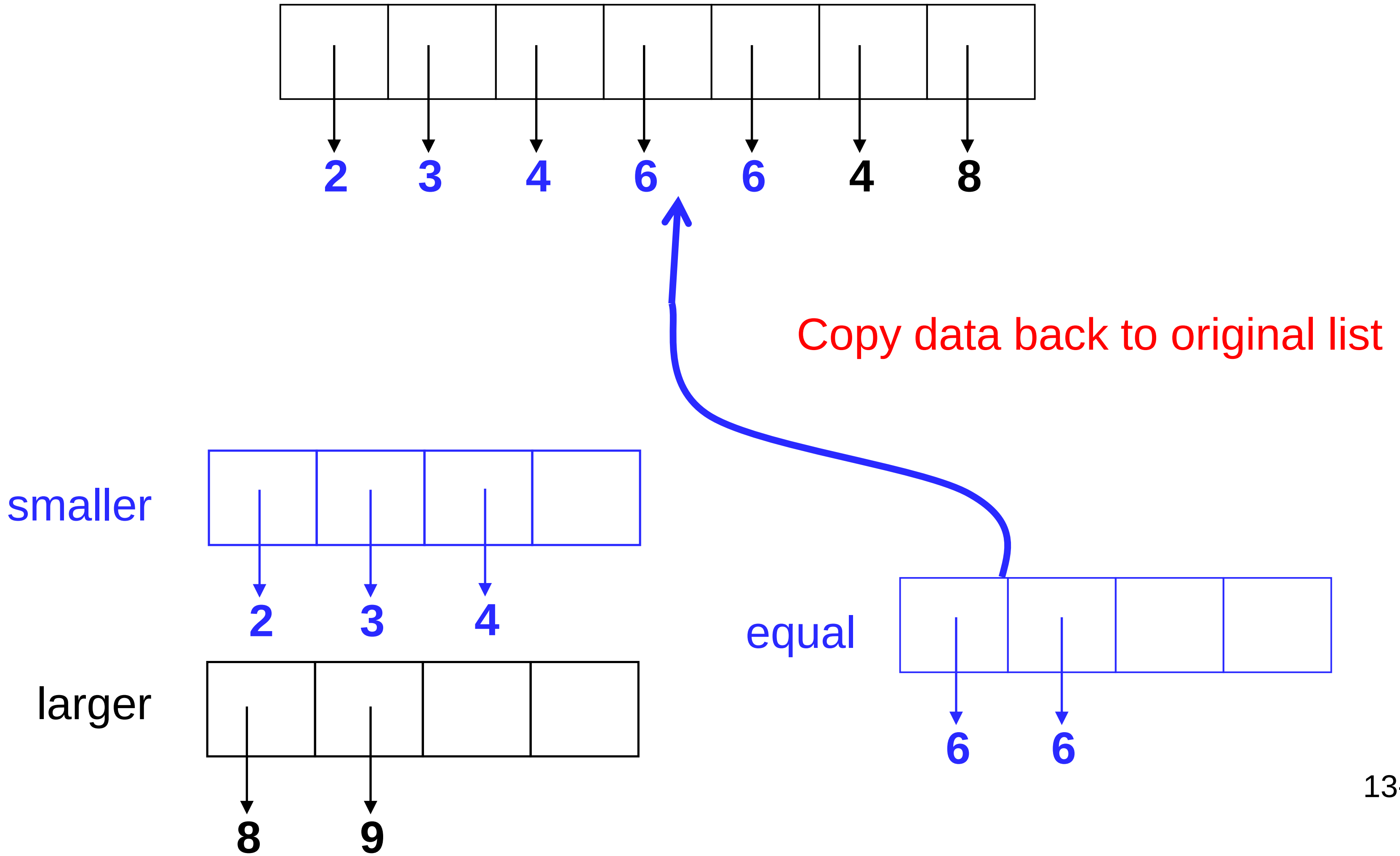

6 3 2 6 9 4 8
Copy data back to original list
smaller
2 3 4
larger
8 9
equal
6 6

QuickSort



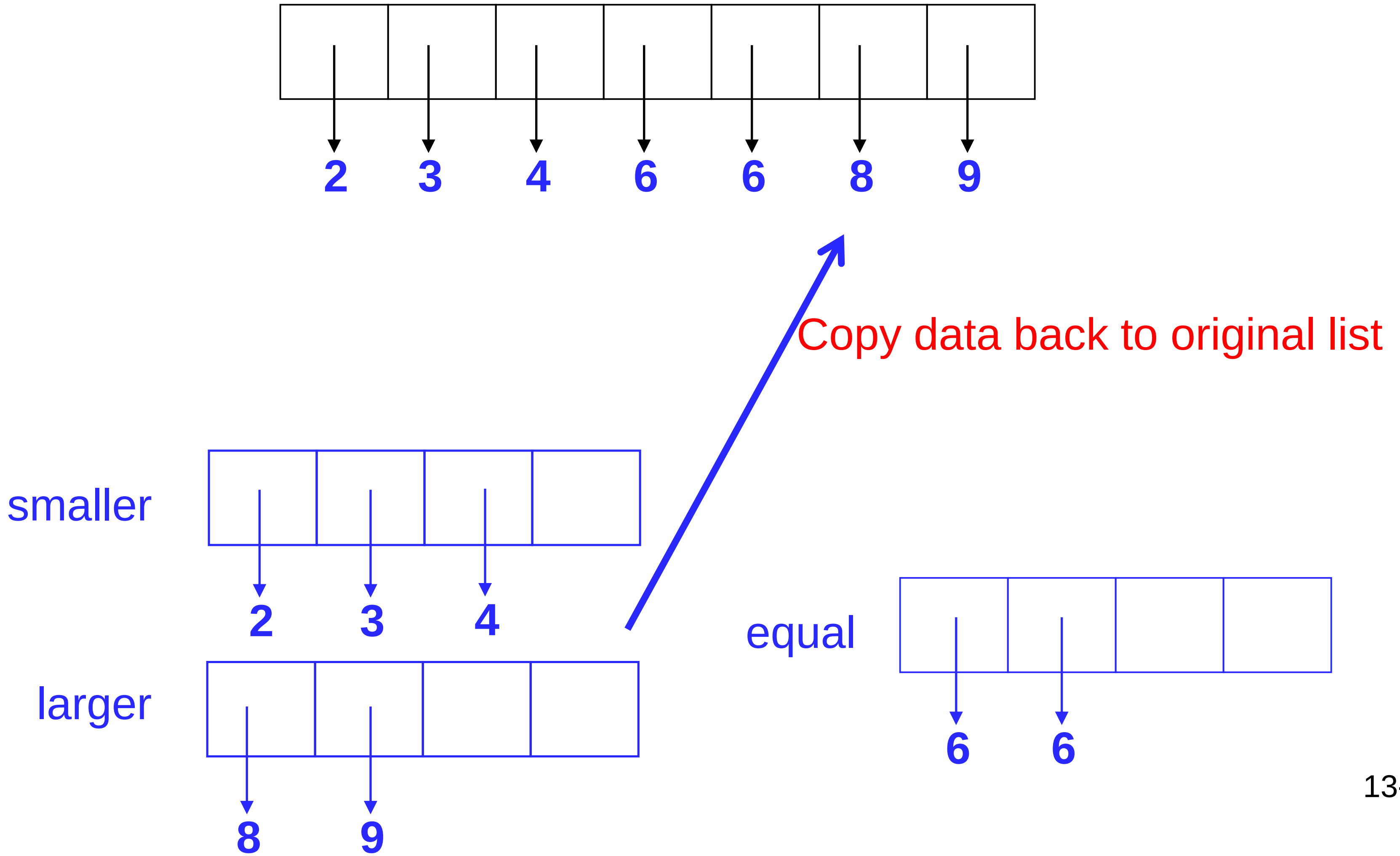
2
3
4
6
9
4
8
Copy data back to original list
smaller
2
3
4
equal
larger
6
6
8
9

QuickSort



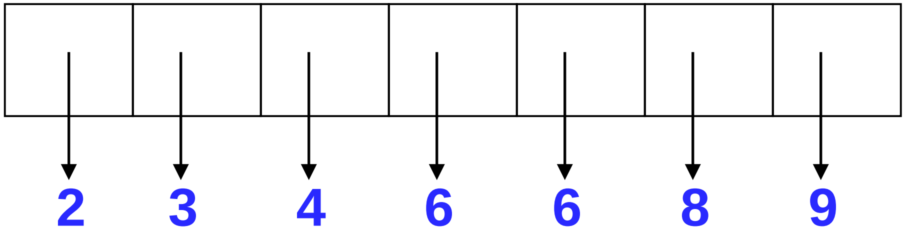
2
3
4
6
6
4
8
Copy data back to original list
smaller
2
3
4
larger
8
9
equal
6
6

QuickSort

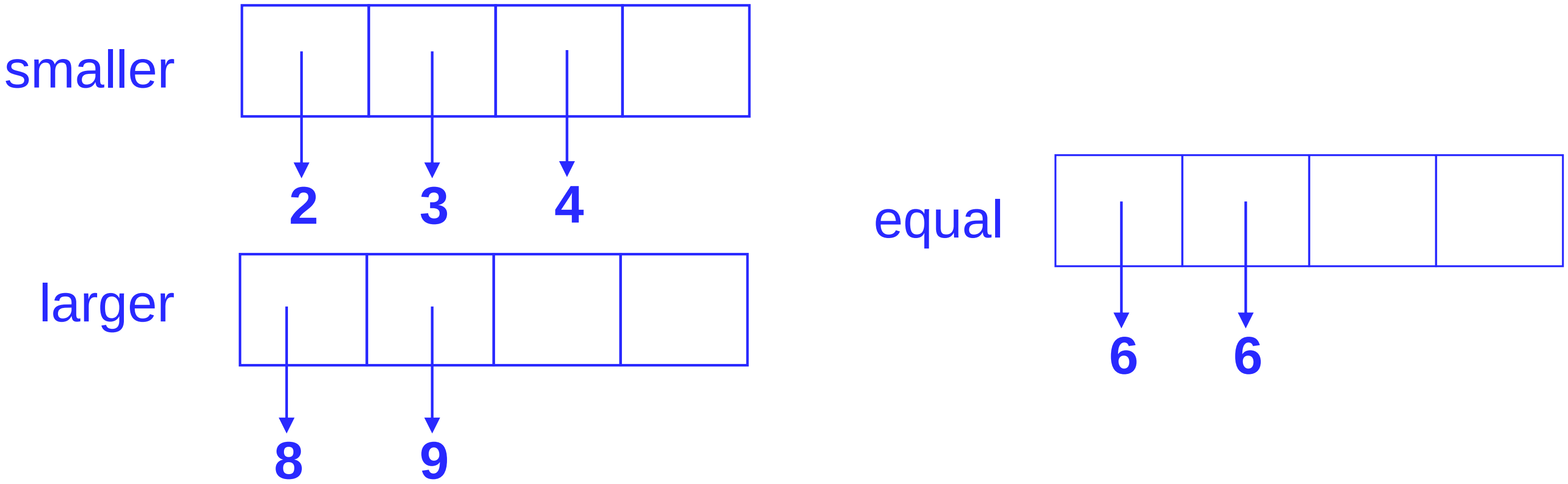


2 3 4 6 6 8 9
Copy data back to original list
smaller
2 3 4
larger
8 9
equal
6 6

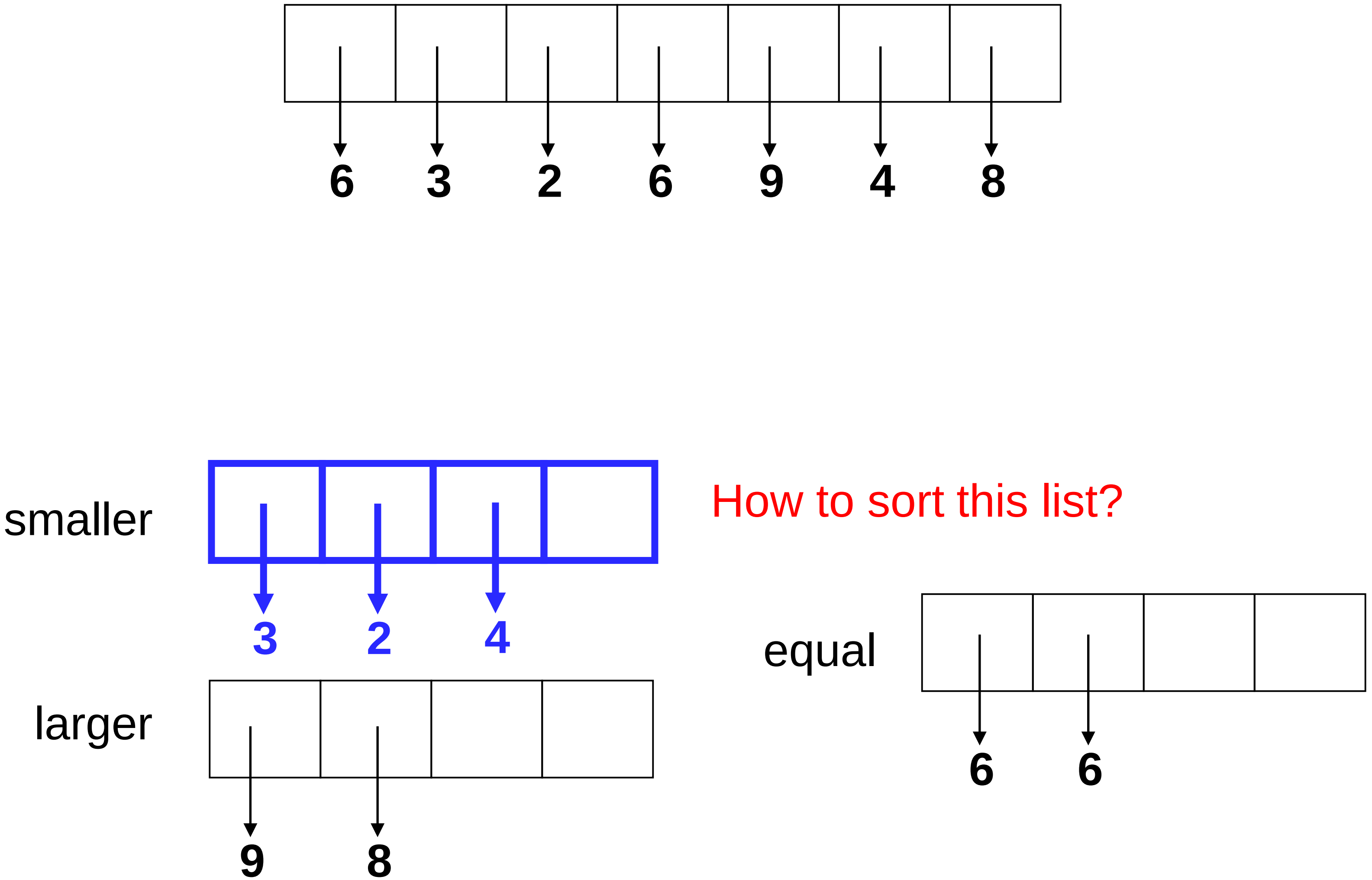
QuickSort



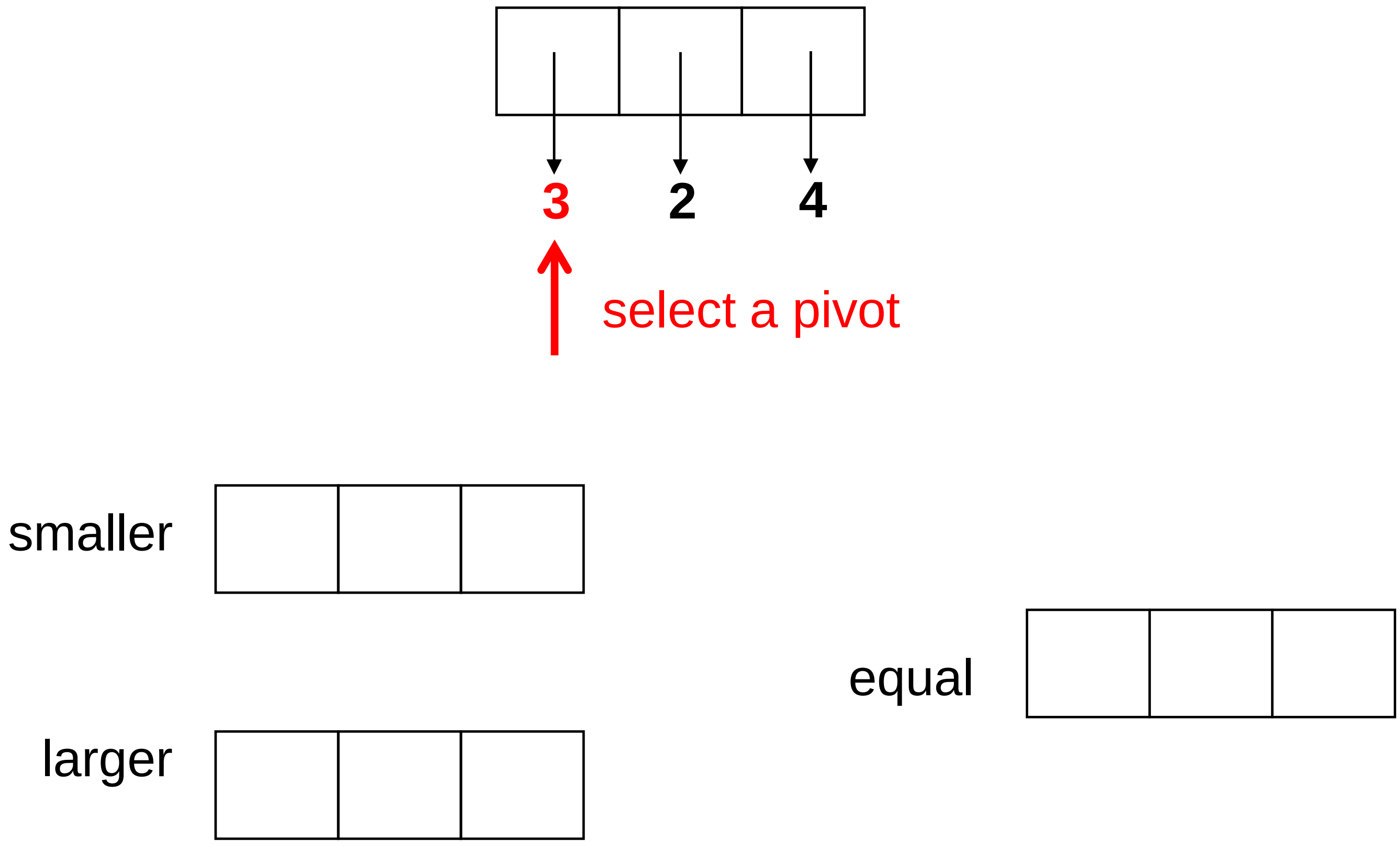
Sorted!



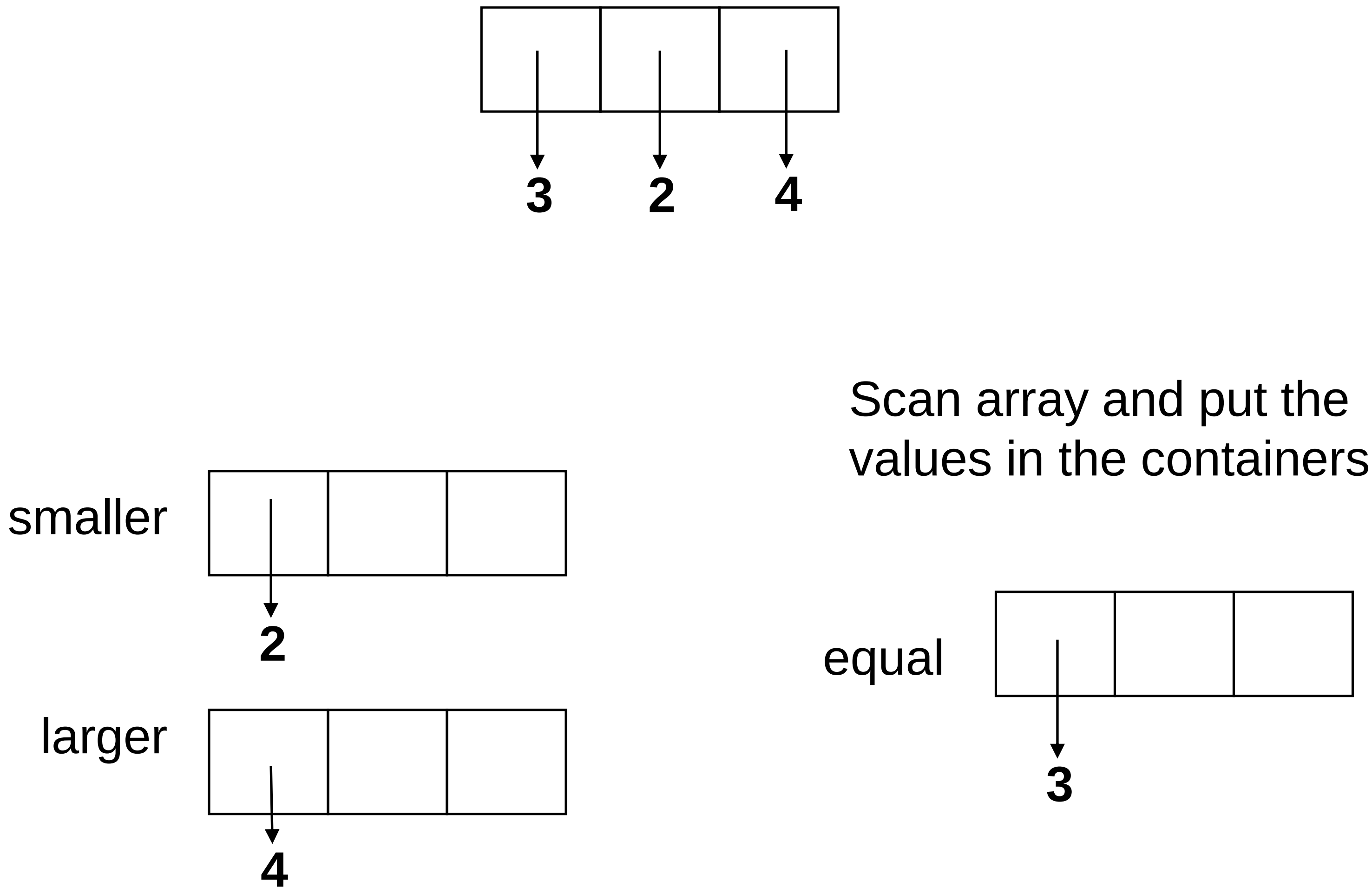
QuickSort



QuickSort

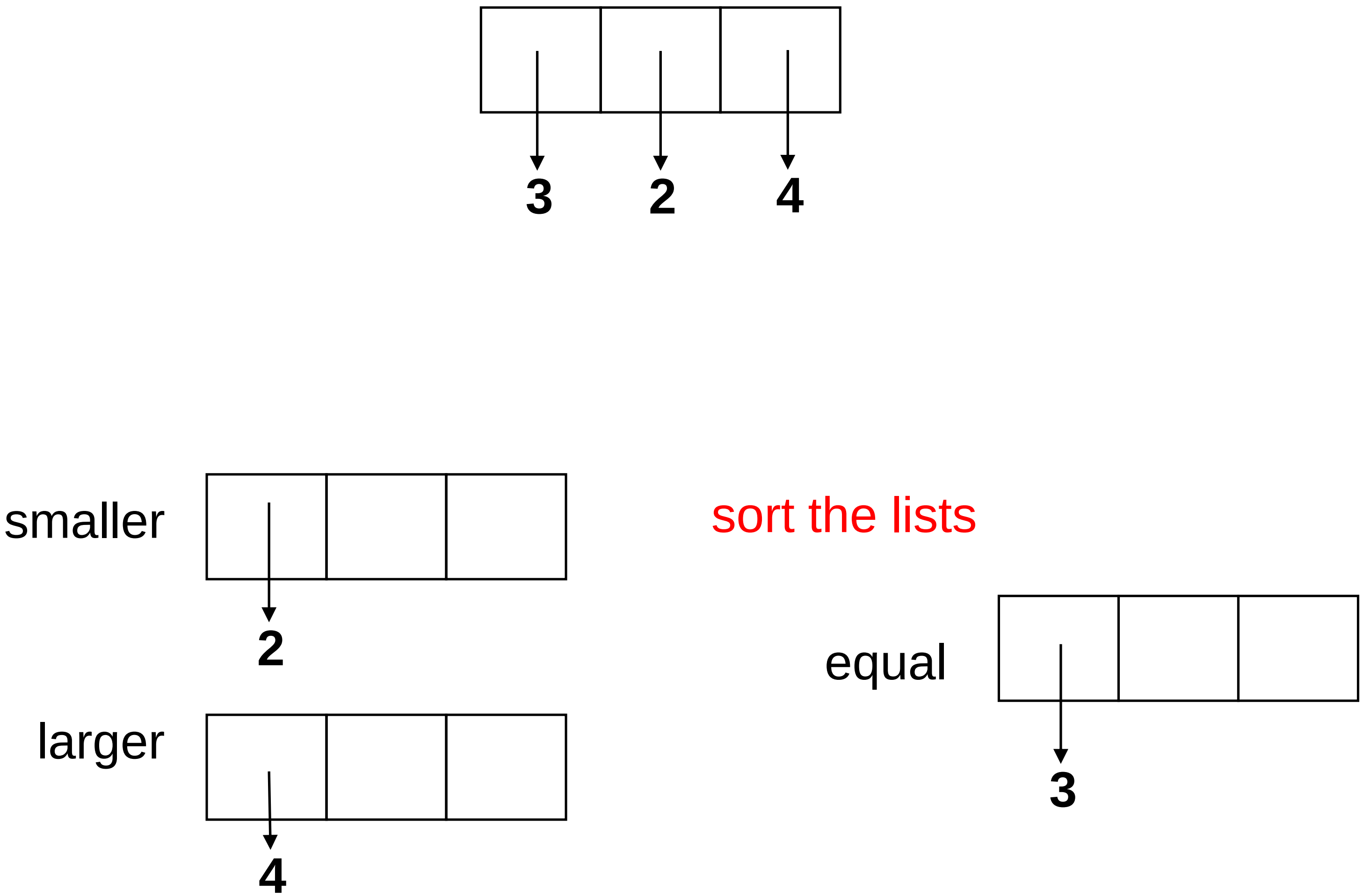


QuickSort

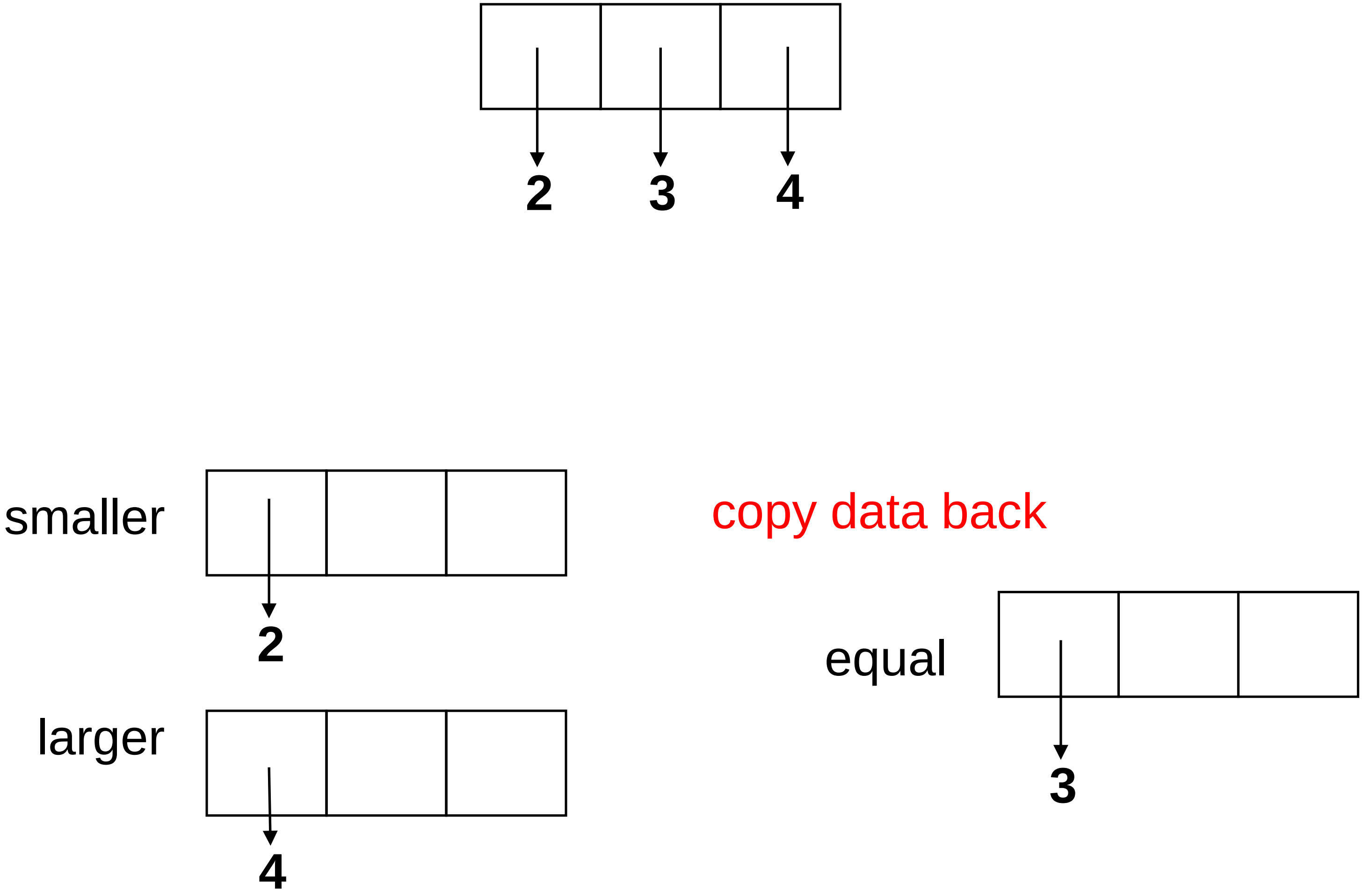


Scan array and put the values in the containers

QuickSort



QuickSort



copy data back

Algorithm quicksort(A,n)

In: Array A storing n values

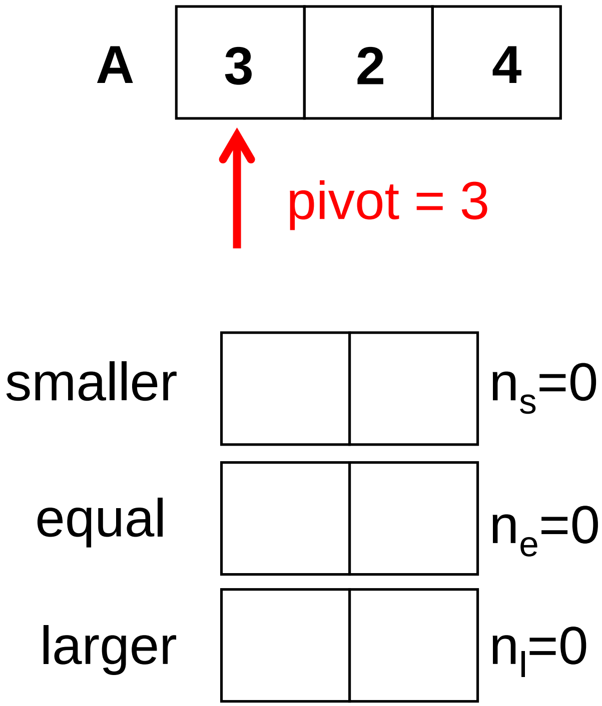
Out: {Sort A in increasing order}

If n > 1 **then** {

smaller, equal, larger = new arrays of size n

$n_s = n_e = n_l = 0$

pivot = A[0]



}

Algorithm quicksort(A,n)

In: Array A storing n values

Out: {Sort A in increasing order}

If n > 1 **then** {

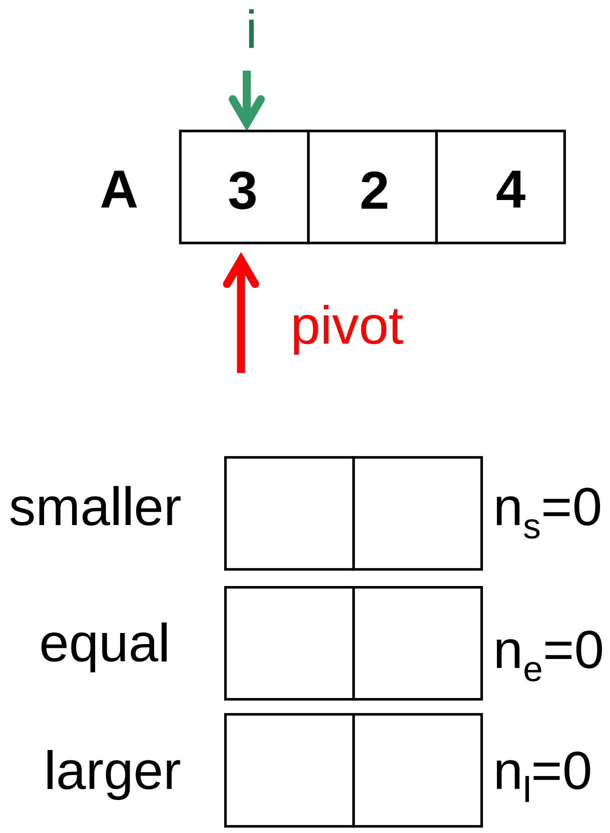
smaller, equal, larger = new arrays of size n

$n_s = n_e = n_l = 0$

pivot = A[0]

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

}



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

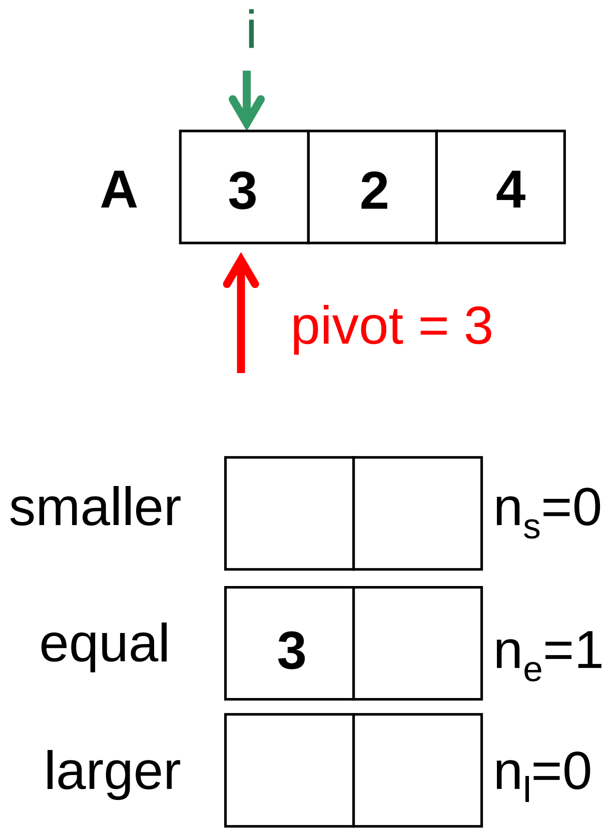
$n_s = n_e = n_l = 0$

pivot = A[0]

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

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

}



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

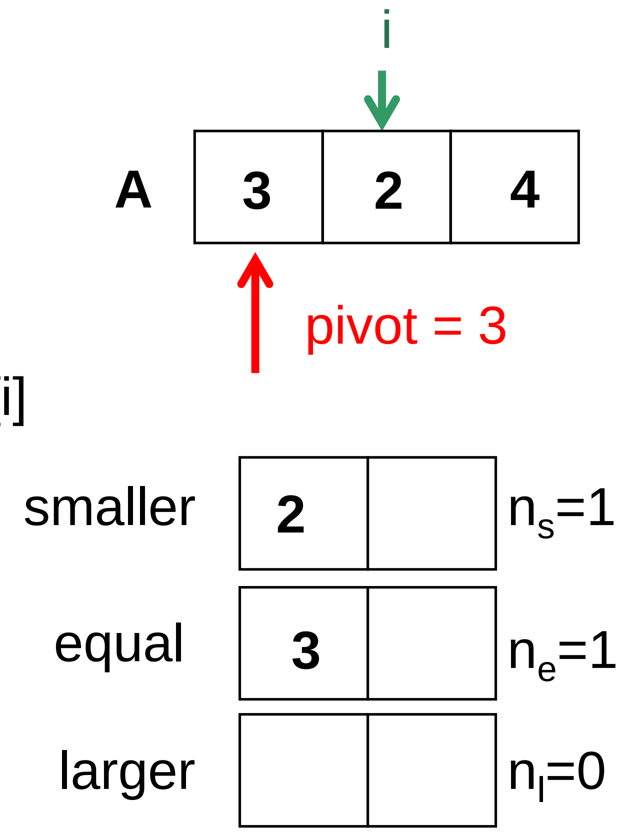
$n_s = n_e = n_l = 0$

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]



}

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

$n_s = n_e = n_l = 0$

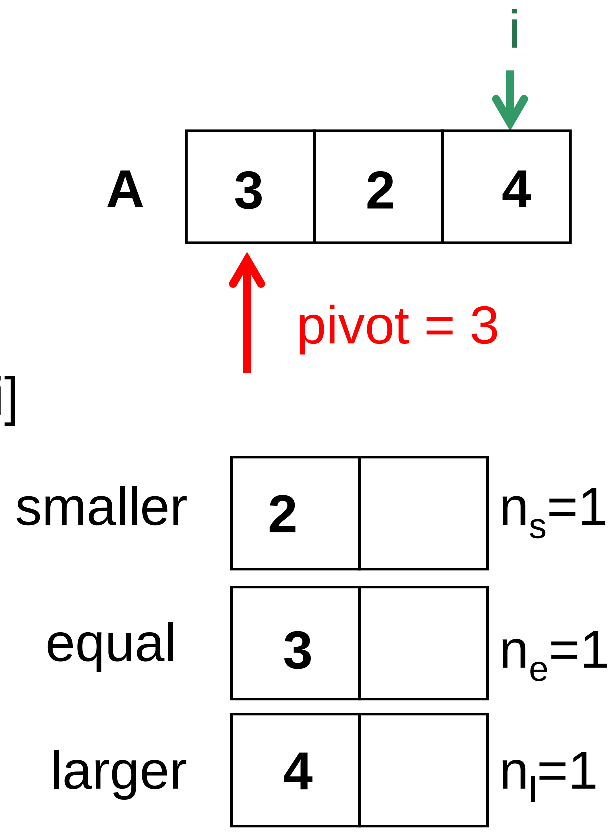
pivot = A[0]

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]



}

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

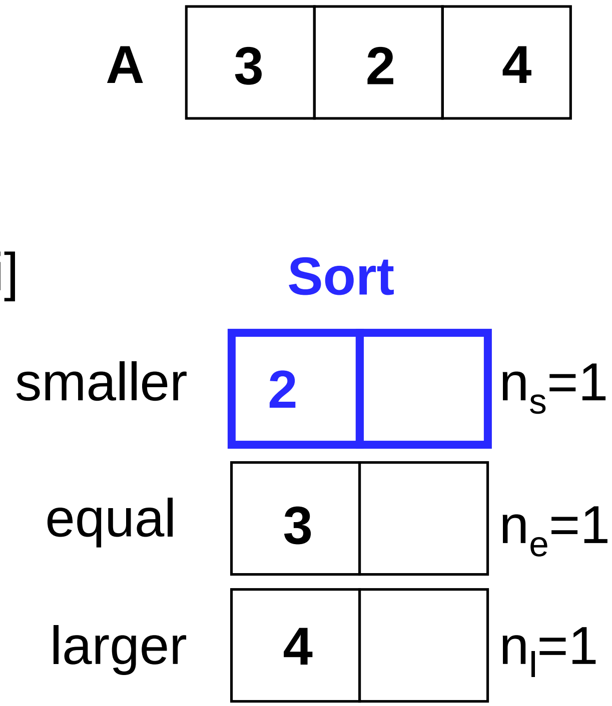
if A[i] = pivot **then** equal[n_e++] = A[i]

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

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

quicksort(smaller,n_s)

}

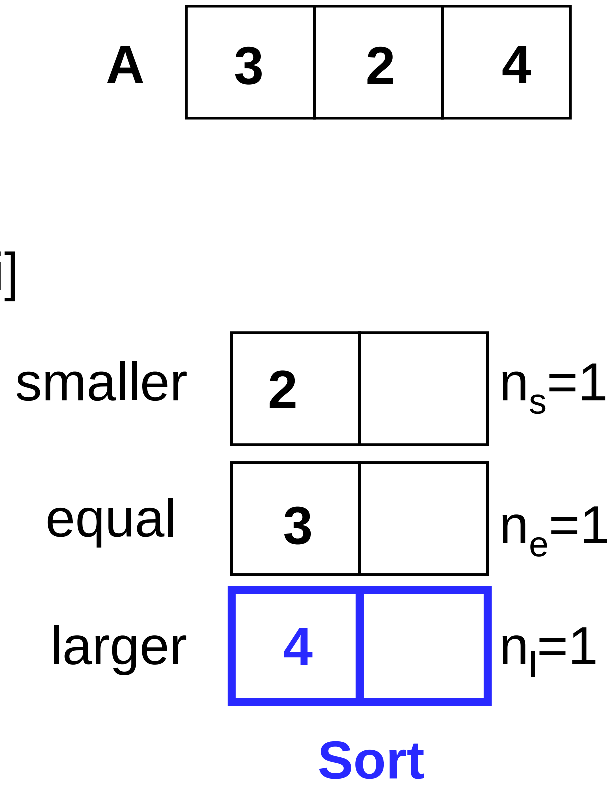


Algorithm quicksort(A,n)

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)
}
```



```
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]
}
```