CMPUT 175
Introduction to Foundations of Computing

Sorting

Objectives

- Introduce the problem of sorting collections
- Learn how to sort using different strategies
- Evaluate the complexity of some sorting algorithms

Outline of Lecture

- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

The Sort Problem

- Given a container, with elements that can be compared, put it in increasing or decreasing order.

0	1	2	3	4	5	6	7	8	9
25	50	10	95	75	30	70	55	60	80

0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	60	70	75	80	95

Sorting Problem (con't)

- Given a container of n elements A[$0..n-1$] such that any elements x and y in the container A can be compared directly, either x<y, or x=y, or x>y.
- We want to permute the elements of A so that at the end $A[0] \leq A[1] \leq \ldots \leq A[n-1]$ (monotone non-decreasing),
  or $A[0] \geq A[1] \geq \ldots \geq A[n-1]$
  (monotone decreasing)

The Order of Things

- Numbers
  - -99 < -34< -6< 0 < 1 < 9 < 23 < 999
- Characters
  - A < B < C < D < E < F < …< X < Y < Z
  - a < b < c < d < e < f < …< x < y < z
  - A < Z < a < z
- Strings
  - Abacus < Alpha < Hello < Memorization < Memorize < Memory < Zebra

Sorting

- There is often a need to put data in order.
- Sorting is among the most basic and universal of computational problems.
- There are hundreds of algorithms and variations on algorithms.
- Variety of sorting methods: internal vs. external, sorting in place vs. sorting with auxiliary structures, etc.

Operations

- Given a collection, with elements that can be compared, put the elements in increasing or decreasing order.
- We must perform two operations to sort a collection:
  - compare elements
  - move elements
- The time to perform each of these two operations, and the number of times we perform each operation, is critical to the time it takes to sort a collection.

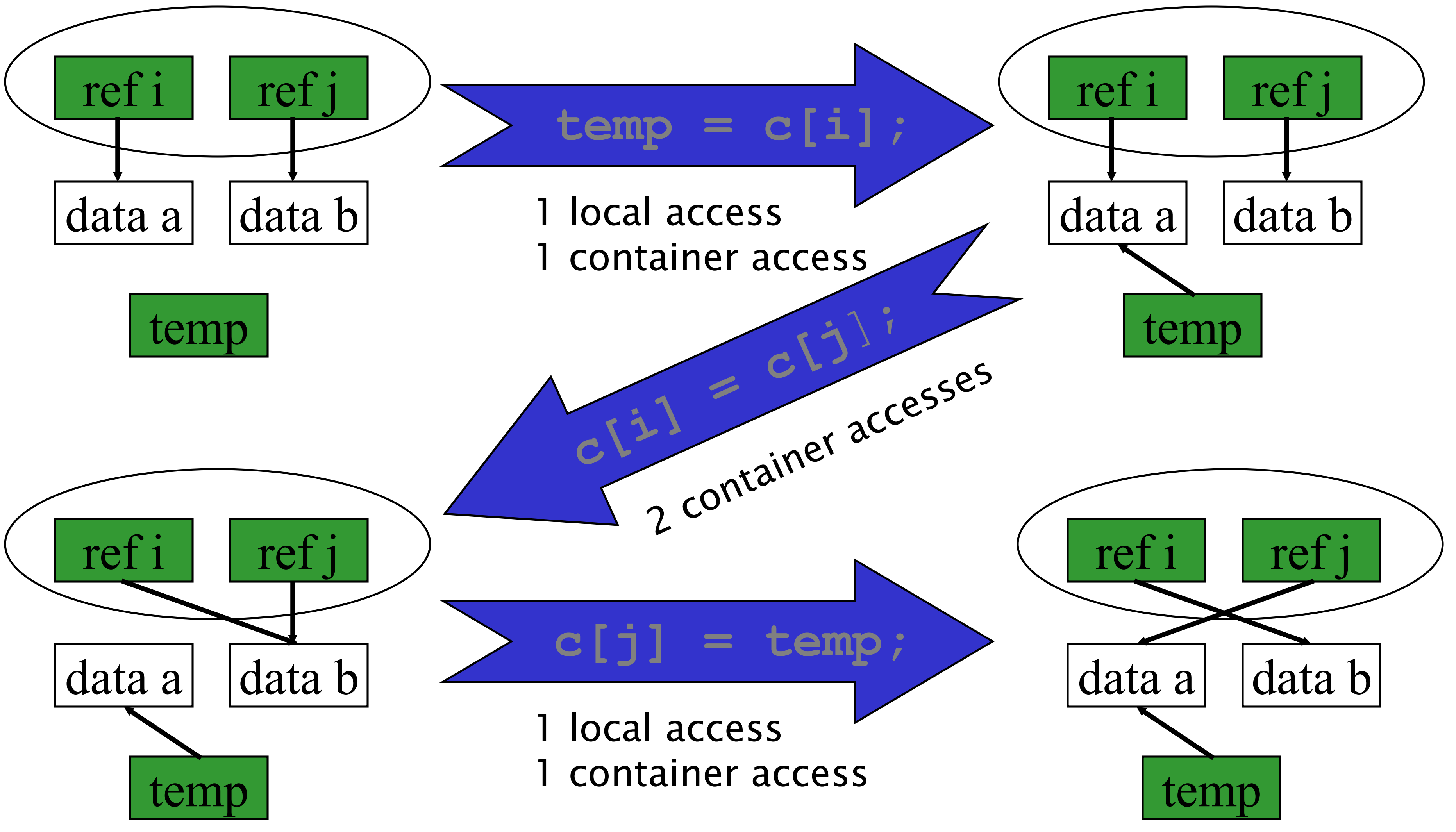
Comparing Primitive Values and Objects

- The sorting algorithms we will consider are based on comparing individual elements.
- If the elements are primitive values, we can use the < operator to compare them.
- If the elements are objects, we cannot use <
- To compare objects we need to add a method that compares these objects, instances of the same class, that returns for example
  - 0 if both objects are equal
  - A negative number of the first is smaller than the second
  - A positive number if the first is larger than the second.

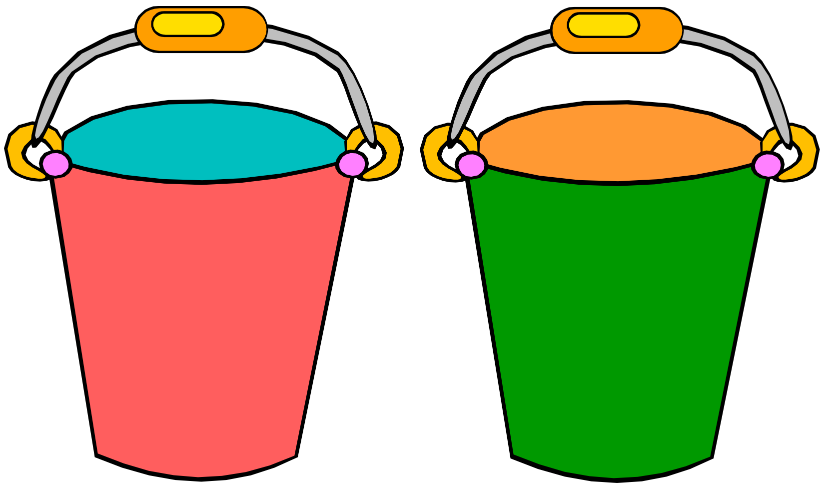
Moving Elements

- Besides comparing elements, the only other operation that is essential to sorting is moving elements.
- The exact code for moving elements depends on the type of collection and the pattern of element movement, but it consists of a series of data accesses.
- One common form of element movement is an exchange which is done using a single temporary variable and three assignments.
- This process usually involves four container accesses and two local variable accesses.
- Since the local variable accesses often get mapped to registers or cache memory we won't count them.

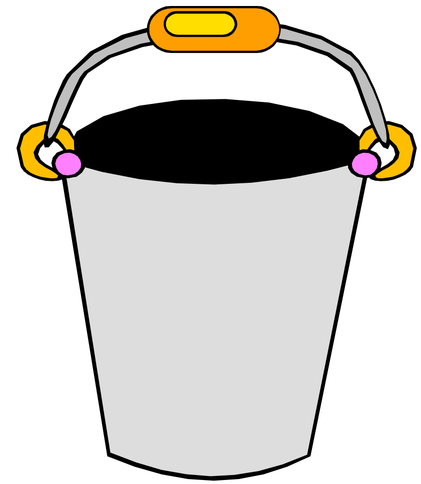
Exchange Algorithm

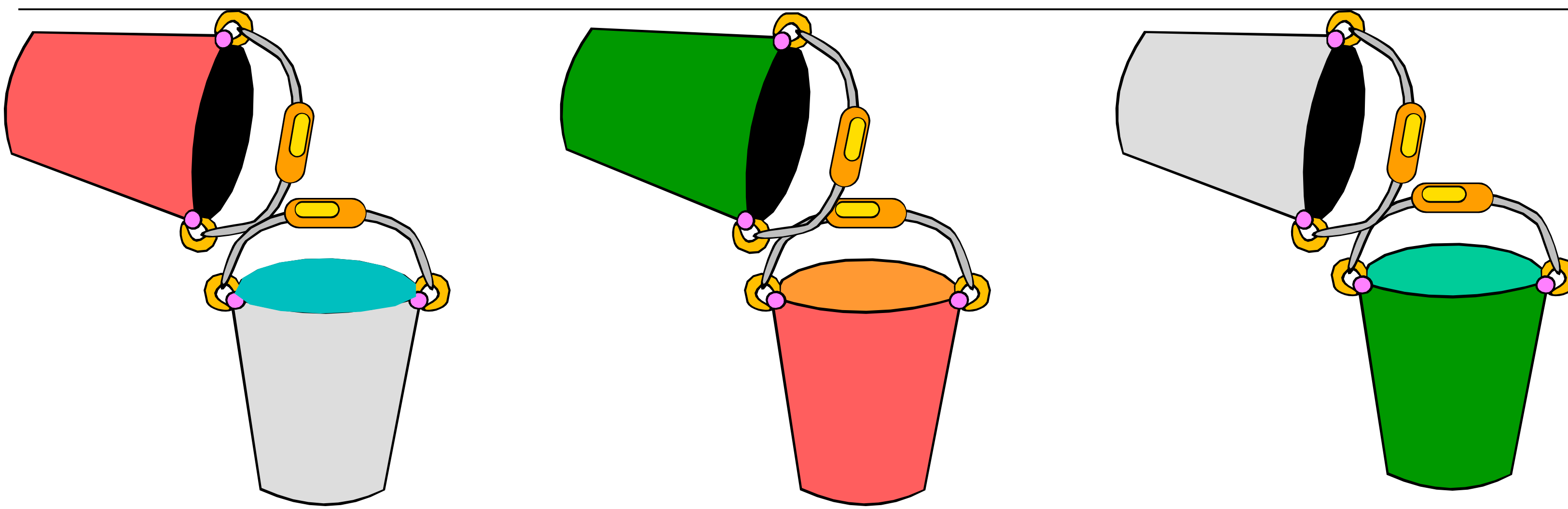


Problem: Transferring liquid to a second bucket that is also full



We need to transfer the blue paint of the red bucket to the green bucket but it already is full with orange paint. We need a third bucket that is empty





Outline of Lecture

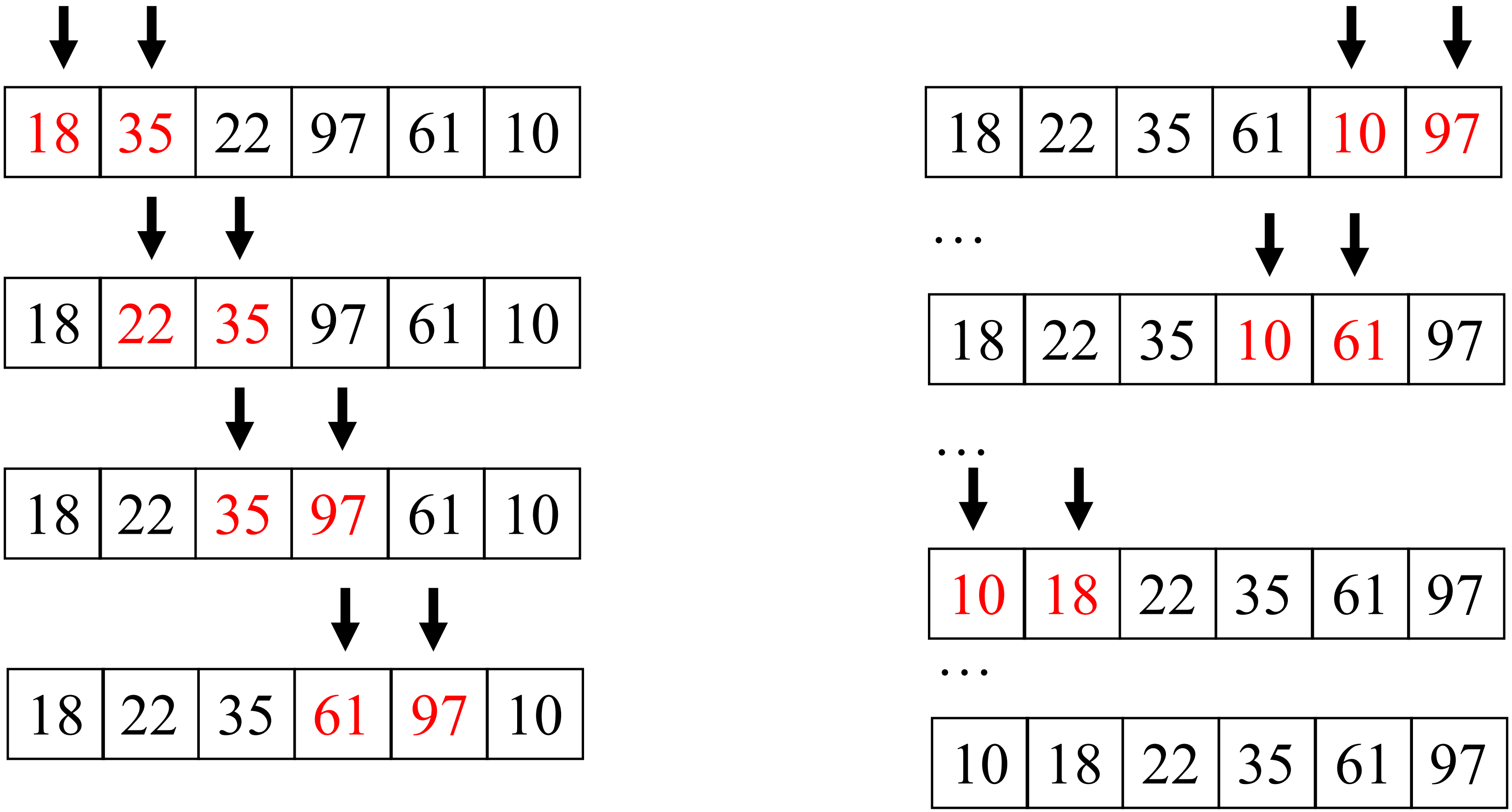
- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

One simple sorting method

Given a list:

35	18	22	97	61	10

Iterate over the collection and permute neighbours if necessary repeat iteration until no permutation possible.



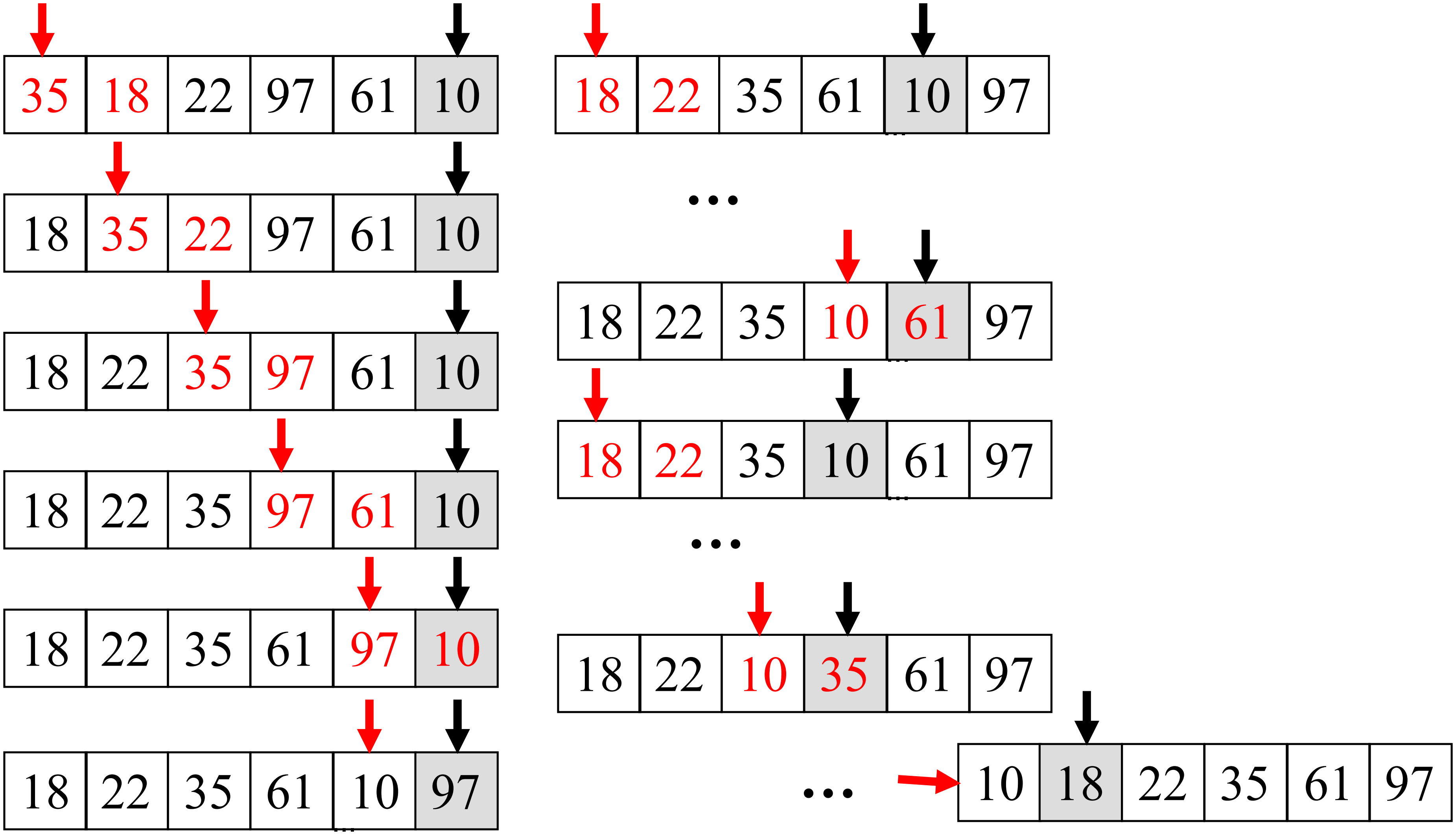
...

The Bubble Sort

Given a list:

35	18	22	97	61	10

First pass of (compare and exchange) will send the largest to the last position.



```
def bubbleSort(data) :
  # Sort the given Array with bubble sort method
  # (Ascending order)

   for  last  in range (len(data)-1,0,-1):
      for current in range (last):
         if ( data[current] > data[current+1] ):
            temp = data[current]
            data[current]=data[current+1]
            data[current+1]=temp
```

Another variation of BubbleSort that stops iterating when sorted

```
def bubbleSortTrackingExchange(data) :
 # Sort the given Array with Bubble sort method  (Ascending order)
 # We keep track if exchanges were made and
 # stop if no exchanges are made in a given pass

   exchange = True
   last = len(data)-1
   while exchange and last>=0:
       exchange = False
       for current in range (last):
           if ( data[current] > data[current+1] ):
               temp = data[current]
               data[current]=data[current+1]
               data[current+1]=temp
               exchange = True
       last-=1
```

```
def bubbleSort(data) :
   # Sort the given Array with bubble sort method
   # (Ascending order)

   for  last  in range (len(data)-1,0,-1):
      for current in range (last):
         if ( data[current] > data[current+1] ):
            temp = data[current]
            data[current]=data[current+1]
            data[current+1]=temp
```

Outline of Lecture

- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

Selection Sort

- Look for the smallest element and exchange it with the element whose index is 0.

0	1	2	3	4	5	6	7	8	9
25	50	10	95	75	30	70	55	60	80

0	1	2	3	4	5	6	7	8	9
10	50	25	95	75	30	70	55	60	80

Selection Sort (con't)

- Look for the smallest element whose index is greater than or equal to 1 and exchange it with the element whose index is 1.

0	1	2	3	4	5	6	7	8	9
10	50	25	95	75	30	70	55	60	80

0	1	2	3	4	5	6	7	8	9
10	25	50	95	75	30	70	55	60	80

Selection Sort (con't)

- Look for the smallest element whose index is greater than or equal to 2 and exchange it with the element whose index is 2.

0	1	2	3	4	5	6	7	8	9
10	25	50	95	75	30	70	55	60	80

0	1	2	3	4	5	6	7	8	9
10	25	30	95	75	50	70	55	60	80

Selection Sort (con't)

- Look for the smallest element whose index is greater than or equal to k and exchange it with the element whose index is k (for k = 3, 4, …, n-1)

0	1	2	3	4	5	6	7	8	9
10	25	30	95	75	50	70	55	60	80

0	1	2	3	4	5	6	7	8	9
10	25	30	50	75	95	70	55	60	80

Selection Sort (con't)

0	1	2	3	4	5	6	7	8	9
10	25	30	50	75	95	70	55	60	80

0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	95	70	75	60	80

0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	60	70	75	95	80

Selection Sort (con't)

0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	60	70	75	95	80

0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	60	70	75	95	80

0	1	2	3	4	5	6	7	8	9
10	25	30	50	55	60	70	75	80	95

Outline of Lecture

- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

Selection Sort Algorithm

INPUT : data: an array of integers
OUTPUT: data: sorted in ascending order

Method:

```
first = 1
While (first < data.length - 1) do {
    find Smallest such that data[Smallest] is the
    smallest between data[first] and data[length-1];
    permute data[first] and data[Smallest];
    first ++
}
```

Selection Sort Code in Python

```
def selectionSort(data):
    # Sort the given Array with selection sort method
    #(Ascending order)

    for  index in range(len(data)):
        smallIndex = index
        for i in range(index,len(data)):     # finding smallest
            if (data[i]<data[smallIndex]):
                smallIndex=i

        temp=data[index]                     # swapping
        data[index]=data[smallIndex]
        data[smallIndex]=temp
```

Outline of Lecture

- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

Complexity of Selection Sort

- How many comparison operations are required for a selection sort of an n-element container?
- The sort method searches for the smallest for the indexes: $0, 1, \ldots n-2$.
- Each time we search for the smallest for an index, we do: (n - index) comparisons.
- The total number of comparisons is:

$$(n-0) + (n-1) + \ldots + (n-(n-2)) = (1 + 2 + \ldots + n) - 1 =$$

$$\frac{n(n+1)}{2} - 1 \approx n^2 \text{ for large } n.$$

O(n^2) ➔ Quadratic time complexity

Outline of Lecture

- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

Insertion Sort Algorithm

- The lower part of the collection is sorted and the higher part is unsorted.

0	1	2	3	4	5	6	7	8
60	30	10	20	40	90	70	80	50

- Insert the first element of the unsorted part into the correct place in the sorted part.

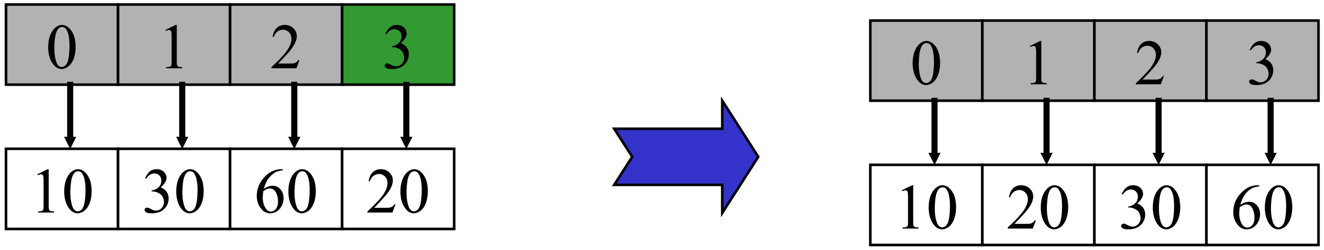
0	1	2	3	4	5	6	7	8
30	60	10	20	40	90	70	80	50

Insertion Sort Algorithm Animation

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90

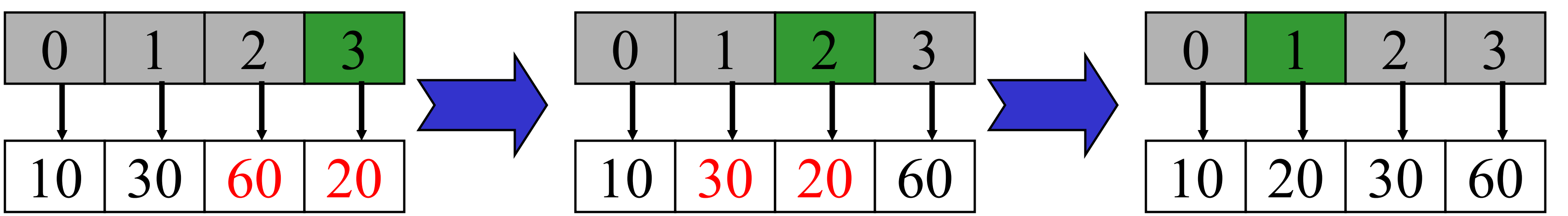
Moving Elements in Insertion Sort

- The Insertion Sort does not use an exchange operation.
- When an element is inserted into the ordered part of the collection, it is not just exchanged with another element.
- Several elements must be “moved”.



Multiple Element Exchanges

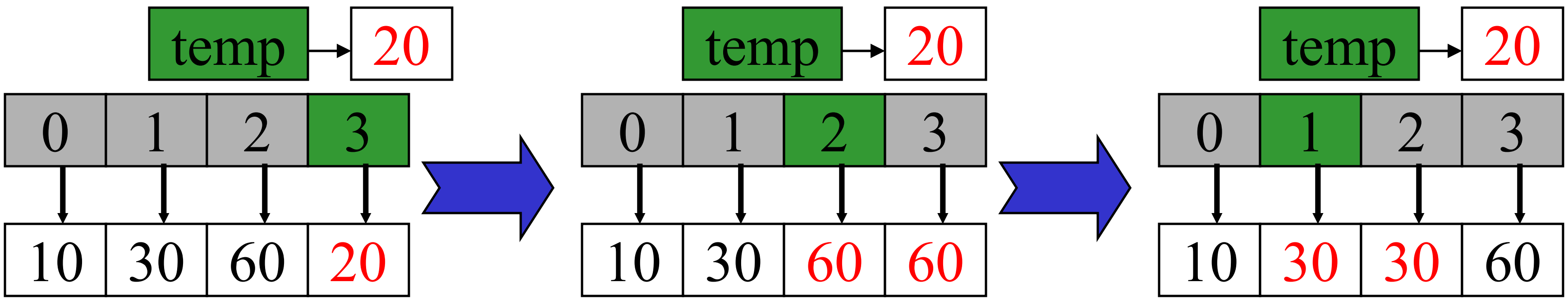
- The naïve approach is to just keep exchanging the new element with its left neighbour until it is in the right location.

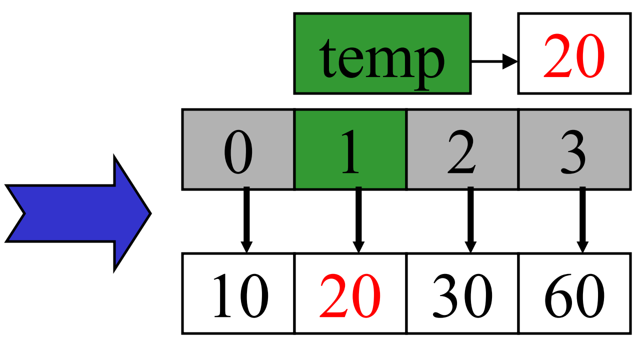


- Every exchange costs four access operations.
- If we move the new element two spaces to the left, this costs 2*4 = 8 access operations.

Shifting Elements

- A better approach is to copy the current then keep shifting to the right elements that are greater than the current then copy back the current in its right location.





- If we shift elements to the left, this costs 2+2 = 4 access operations.

```
1.  temp = data[3]
2.  data[3] = data[2]
3.  data[2] = data[1]
4.  data[1] = temp
```

Insertion Sort Code in Python

```
def insertionSort(data):
    # Sort the given Array with insertion sort method
    #(Ascending order)

    for  index in range(1,len(data)):
        temp = data[index]
        position=index
        while position>0 and data[position-1]>temp:  # shifting
            data[position]=data[position-1]
            position=position-1

        data[position]=temp                           # inserting
```

Counting Comparisons

- How many comparison operations are required for an insertion sort of an n-element collection?
- The sort method makes a decision for insertion in a loop for the indexes: index = 1, 2, … n - 1.

```
for  index in range(1,len(data)):
```

- Each time, we do a comparison in a loop to shift for some of the indexes: position, position-1, … 1.

```
while position>0 and data[position-1]>temp:
```

Time Complexity of Insertion Sort

- Best case $O(n)$ accesses.
  - Container is already sorted
- Worst case $O(n^2)$ accesses.
  - Container in reverse order
- Average case $O(n^2)$ accesses.
- Note: this means that for nearly sorted collections, insertion sort is better than selection sort even though in average and worst cases, they are the same: $O(n^2)$.

Space Complexity of Insertion Sort

- Besides the collection itself, the only extra storage for this sort is the single temp reference used in the move element method.
- Therefore, the space complexity of Insertion Sort is O(n).

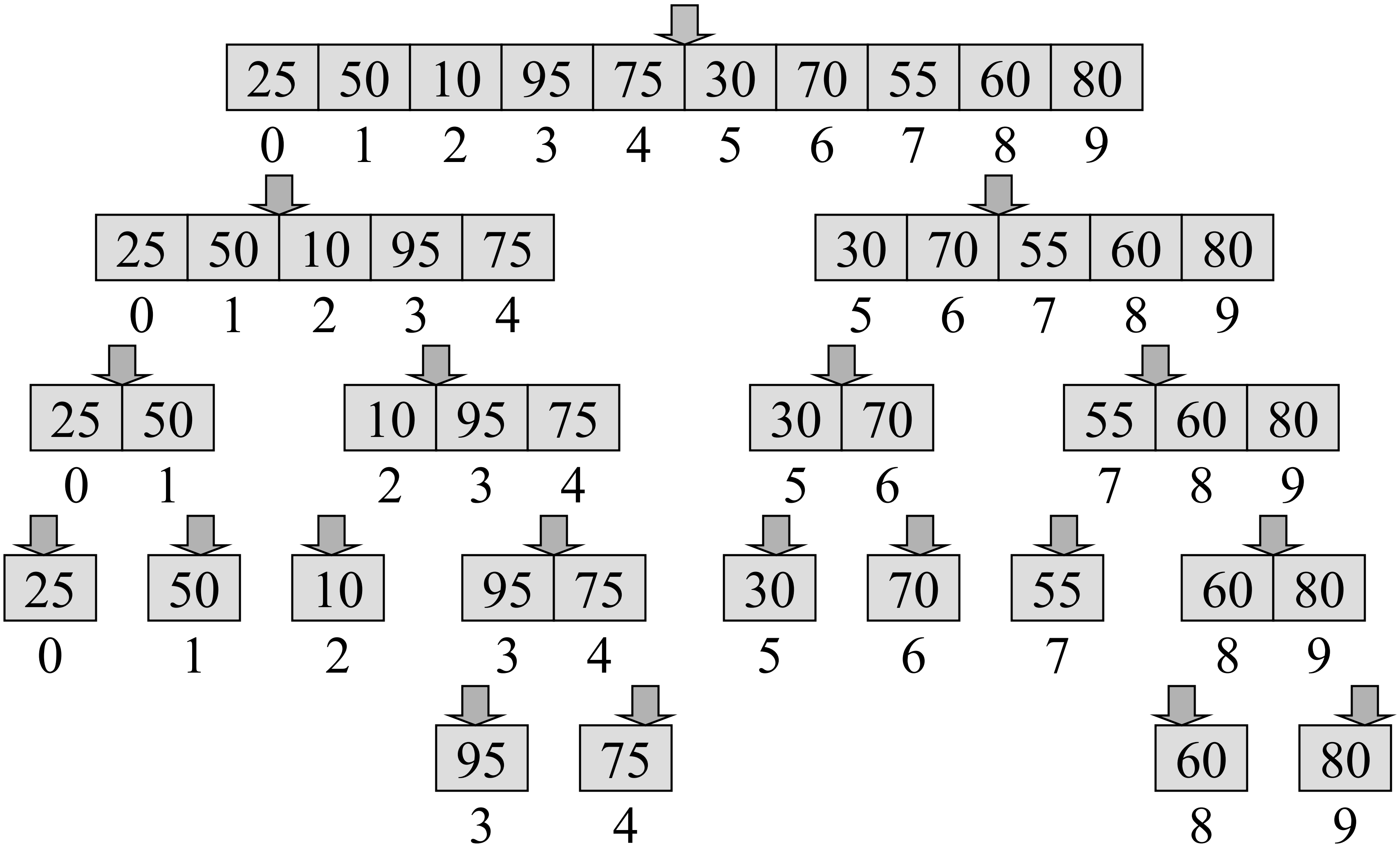
Outline of Lecture

- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

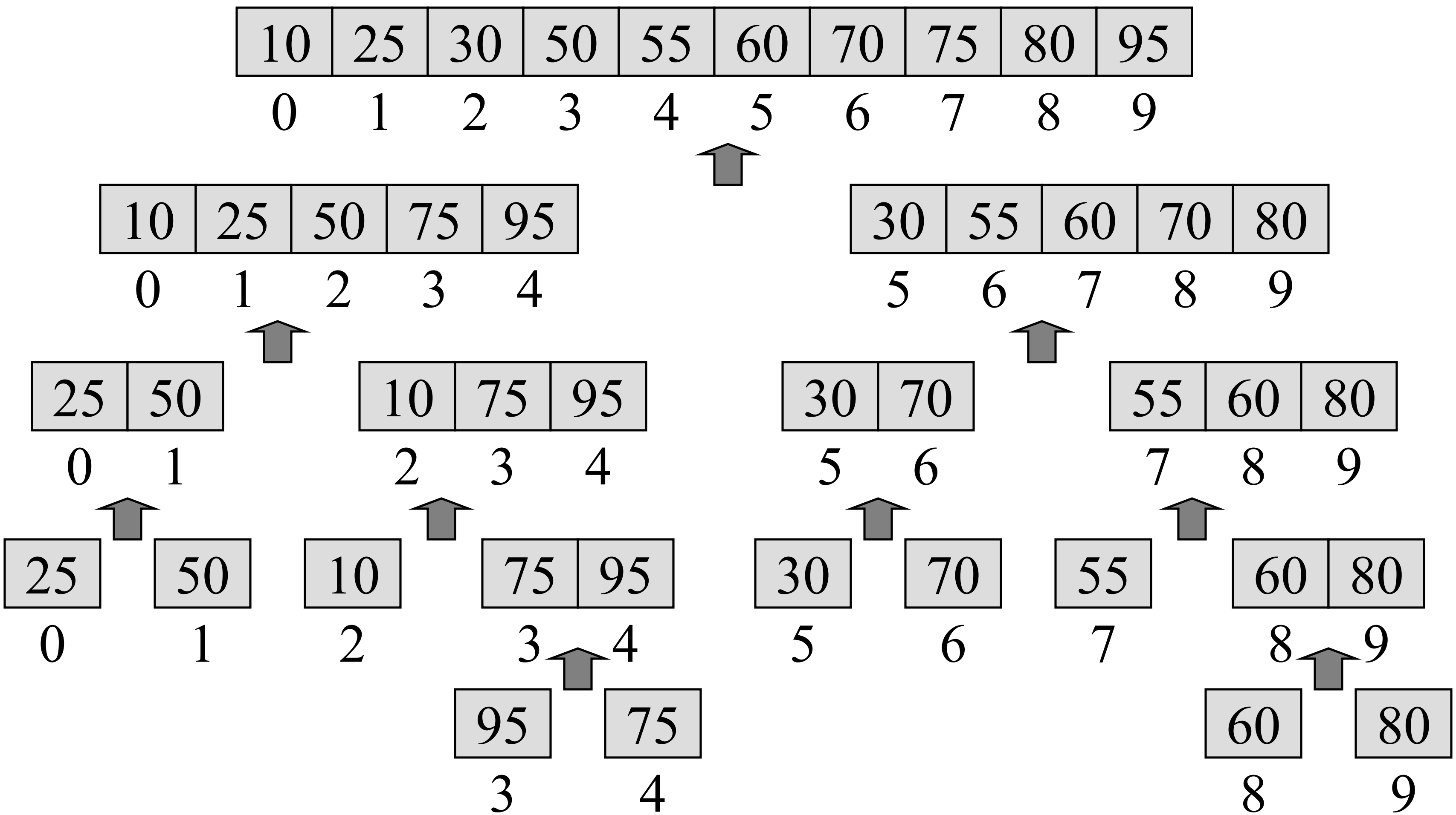
Recursive MergeSort Concept

- So far the sorting methods we saw are all in the $O(n^2)$. Can we do better?
- We can build a recursive sort, called mergeSort:
  - split the list into two equal sub-lists
  - sort each sub-list using a recursive call
  - merge the two sorted sub-lists

MergeSort Example - split

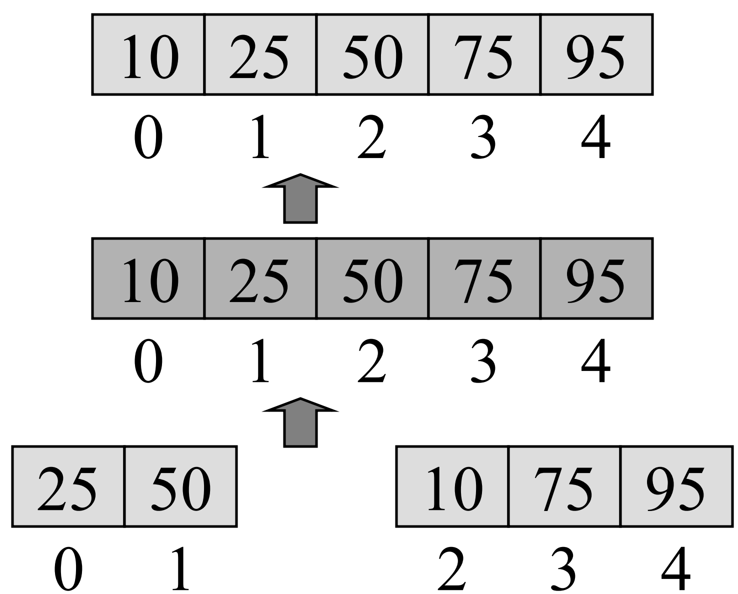


MergeSort Example - join



MergeSort Needs Extra Storage

- Unlike selection sort or insertion sort, merge sort does not work “in place”.
- A temporary collection is needed so twice as much memory is required.

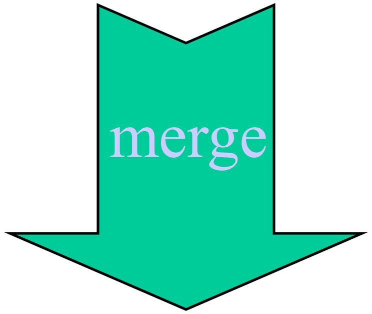


Merging Two Sorted Lists

- Merge is an operation that combines two sorted lists together into one.

0	1	2
10	40	60

0	1	2	3
50	70	80	90



0	1	2	3	4	5	6
10	40	50	60	70	80	90

Merge Algorithm

- For now, assume the result is to be placed in a separate array called `result`, which has already been allocated.
- The two given arrays are called `left` and `right`
- `Left` and `right` are in increasing order.
- For the complexity analysis, the size of the input, n, is the sum $n_{left} + n_{right}$

Merge Algorithm

- For each array keep track of the current position (initially 0).

- REPEAT until all the elements of <u>one</u> of the given arrays have been copied into `result` :
  - Compare the current elements of **`left`** and **`right`**
  - Copy the smaller into the current position of **`result`** (break ties however you like)
  - Increment the current position of **`result`** and the array that was copied from

- Copy all the remaining elements of the other given array into `result`.

Merge Example (1)

Current positions indicated in red

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6

Compare current elements; copy smaller; update current

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10						

Compare current elements; copy smaller; update current

Merge Example (2)

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40					

Compare current elements; copy smaller; update current

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40	50				

Compare current elements; copy smaller; update current

Merge Example (3)

All elements copied

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40	50	60			

Copy the rest of the elements from the other array

0	1	2
10	40	60

0	1	2	3
50	70	80	90

0	1	2	3	4	5	6
10	40	50	60	70	80	90

Code for merge()

```
def merge(left,right):
    result=[]
    i,j=0,0
    while i<len(left) and j<len(right):
        if left[i]<=right[j]:
            result.append(left[i])
            i+=1
        else:
            result.append(right[j])
            j+=1

    result += left[i:]
    result += right[j:]
    return result
```

Why appending both?

Code for mergeSort

```
def mergeSort(data):
    # Sort myself using a merge sort.

    if len(data) <=1:
        return data

    middle = len(data)//2

    left=mergeSort(data[:middle]
    right=mergeSort(data[middle:]

    return merge(left,right)
```

Time comparison of MergeSort and selection sort

- Sample times for our program

	n = 20,000	n = 100,000
merge sort	< 1 second	1 second
selection sort	16 seconds	400 seconds

- Why is the MergeSort way faster?

Complexity of MergeSort

- The complexity of the MergeSort algorithm is the complexity of the split and the complexity of the merge.
- The complexity of the split is the depth of the tree of execution – the number of times we need to divide n by 2 to get 1. That is *log(n)*
- The complexity of the merge is O(n) since every single element is copied to `result` once
- We have a merge involving all elements at each level of the recursion tree
- In all there are *log (n)* levels of recursive calls & at each level the total cost of merging all the lists at that level is at most *n*
- The time complexity of the merge sort is O(n log(n))

Space Complexity of MergeSort

- The MergeSort algorithm requires extra space to do the merge (`result`). This doubles the space needed.
- The space is in the order of 2n
- This is because is sorting is not done "in place"
- Can we do better? Can we have the same O(nlog(n)) time complexity and keeping the space requirement to n?
- Can we do in place sorting with the divide and conquer strategy?

Outline of Lecture

- The sorting problem
- Simple methods like bubble sort
- Selection sort example
- Selection sort code
- Complexity of selection sort
- Insertion Sort
- MergeSort
- Quicksort

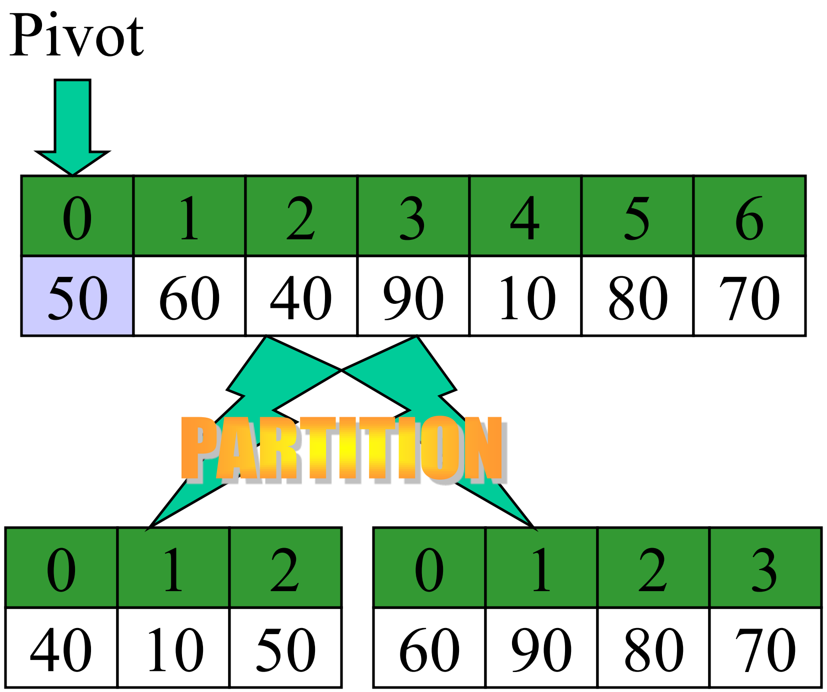
Merge Sort Algorithm - reminder

- Merge Sort sorts a given array (`anArray`) into increasing order as follows:
- Split `anArray` into two non-empty parts any way you like. For example
  `left` = the first n/2 elements in `anArray`
  `right` = the remaining elements in `anArray`
- Sort `left` and `right` by recursively calling MergeSort with each one.
- Now you have two sorted arrays containing all the elements from the original array. Use `merge` to combine them, put the result in `anArray`.

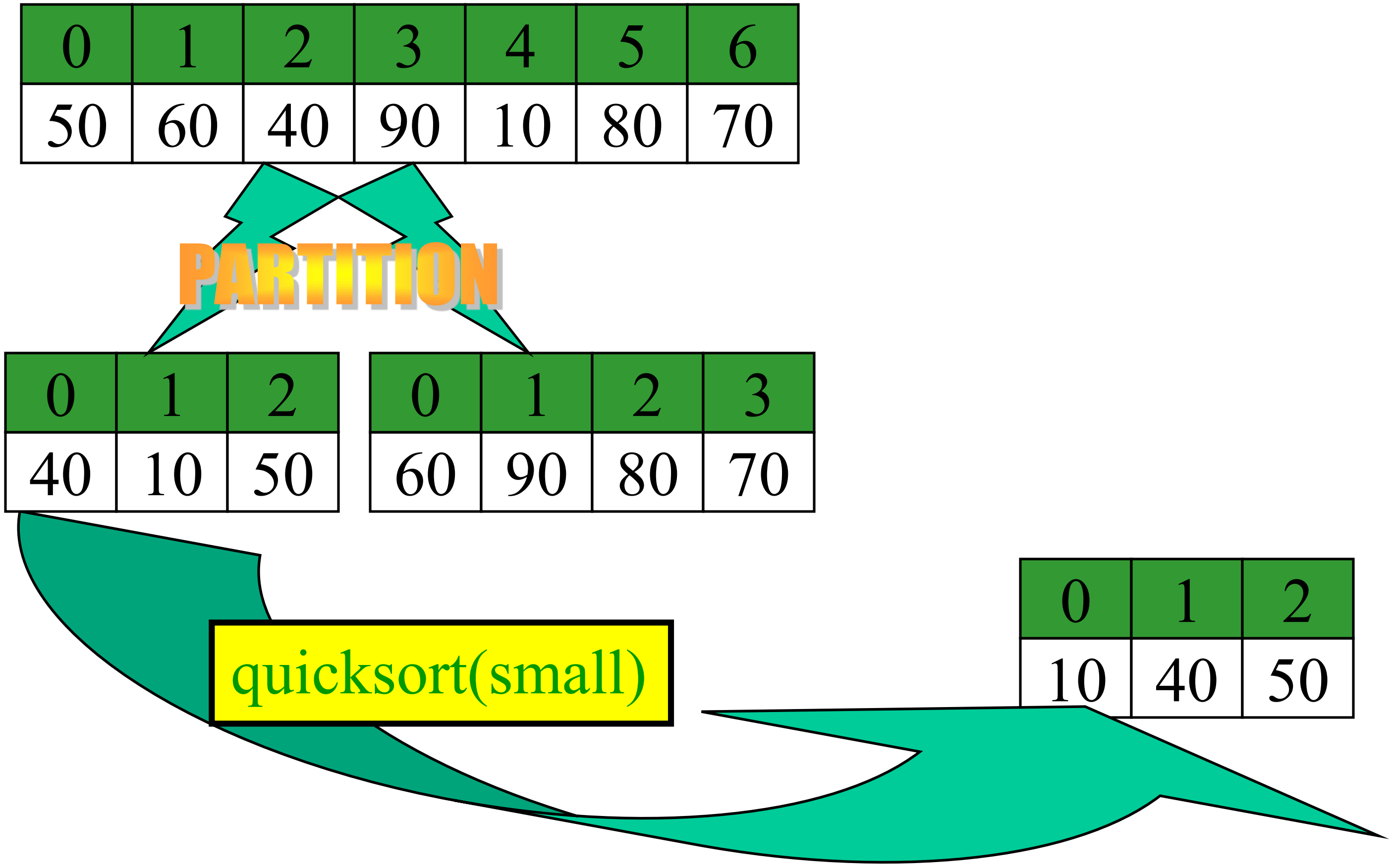
Quicksort Algorithm

- Partition `anArray` into two non-empty parts.
  - Pick any value in the array, `pivot`.
  - `small` = the elements in `anArray` < `pivot`
  - `large` = the elements in `anArray` > `pivot`
  - Place `pivot` in either part, so as to make sure neither part is empty.
- Sort `small` and `large` by recursively calling Quicksort with each one.
- You could use `merge` to combine them, but because you know the elements in `small` are smaller than the elements in `large` you can simply <u>concatenate</u> `small` and `large`, and put the result into `anArray`.

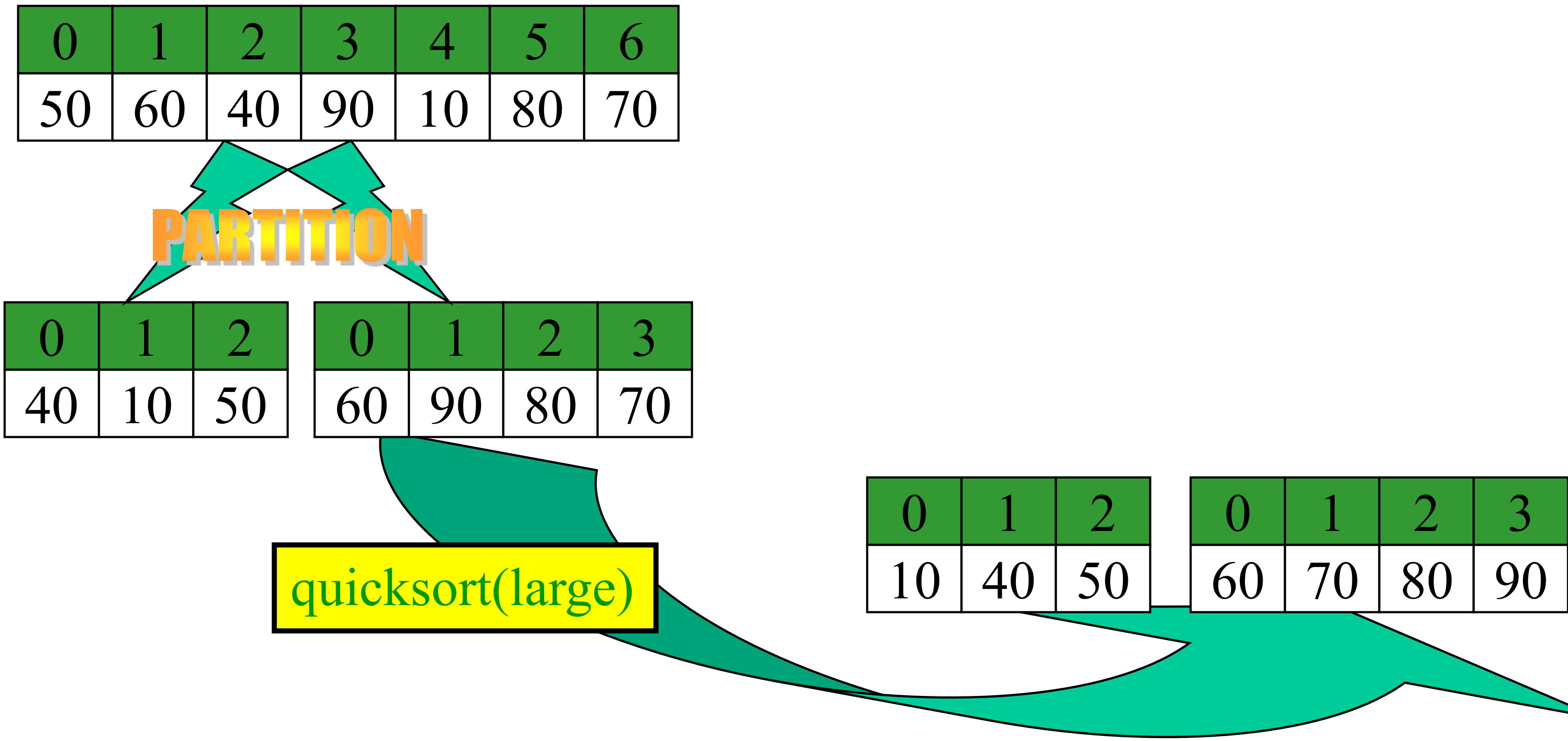
Quicksort – (1) Partition



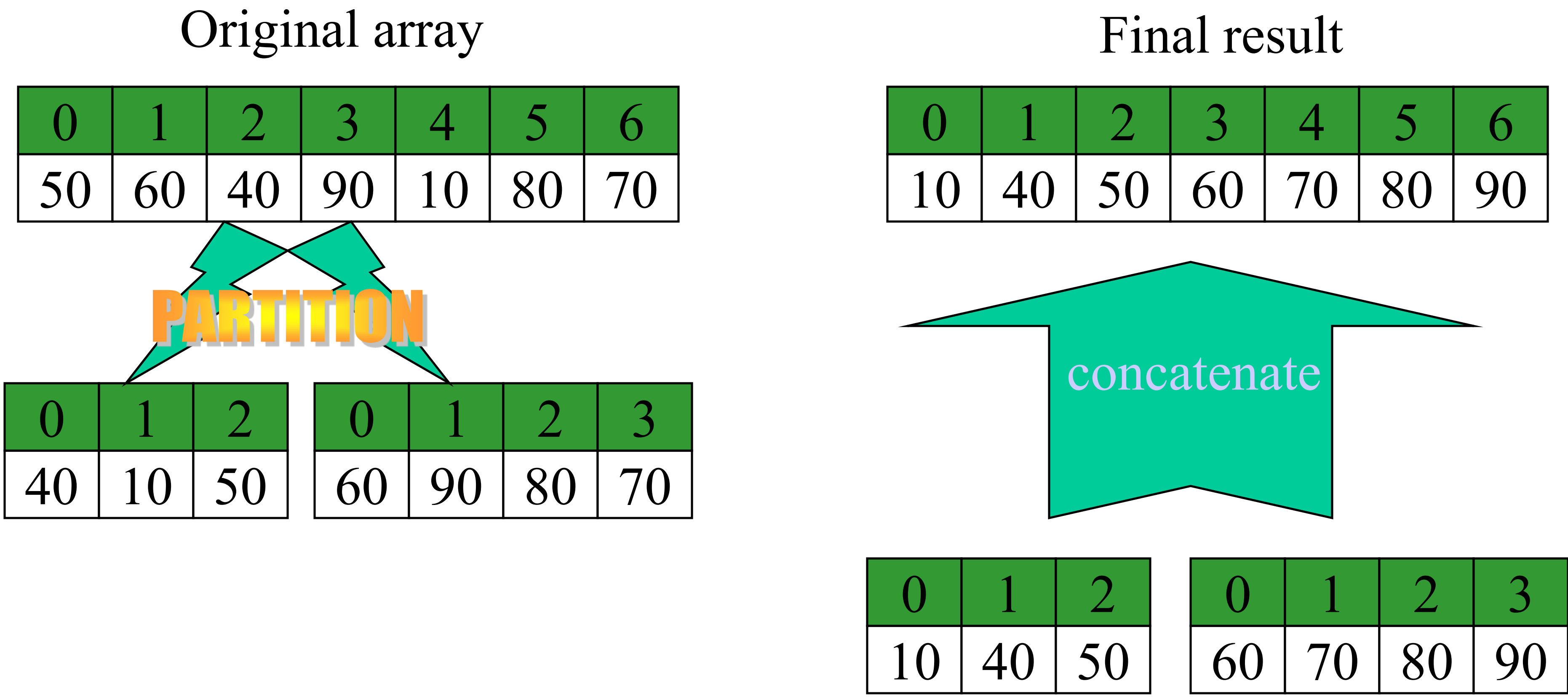
Quicksort – (2) recursively sort small



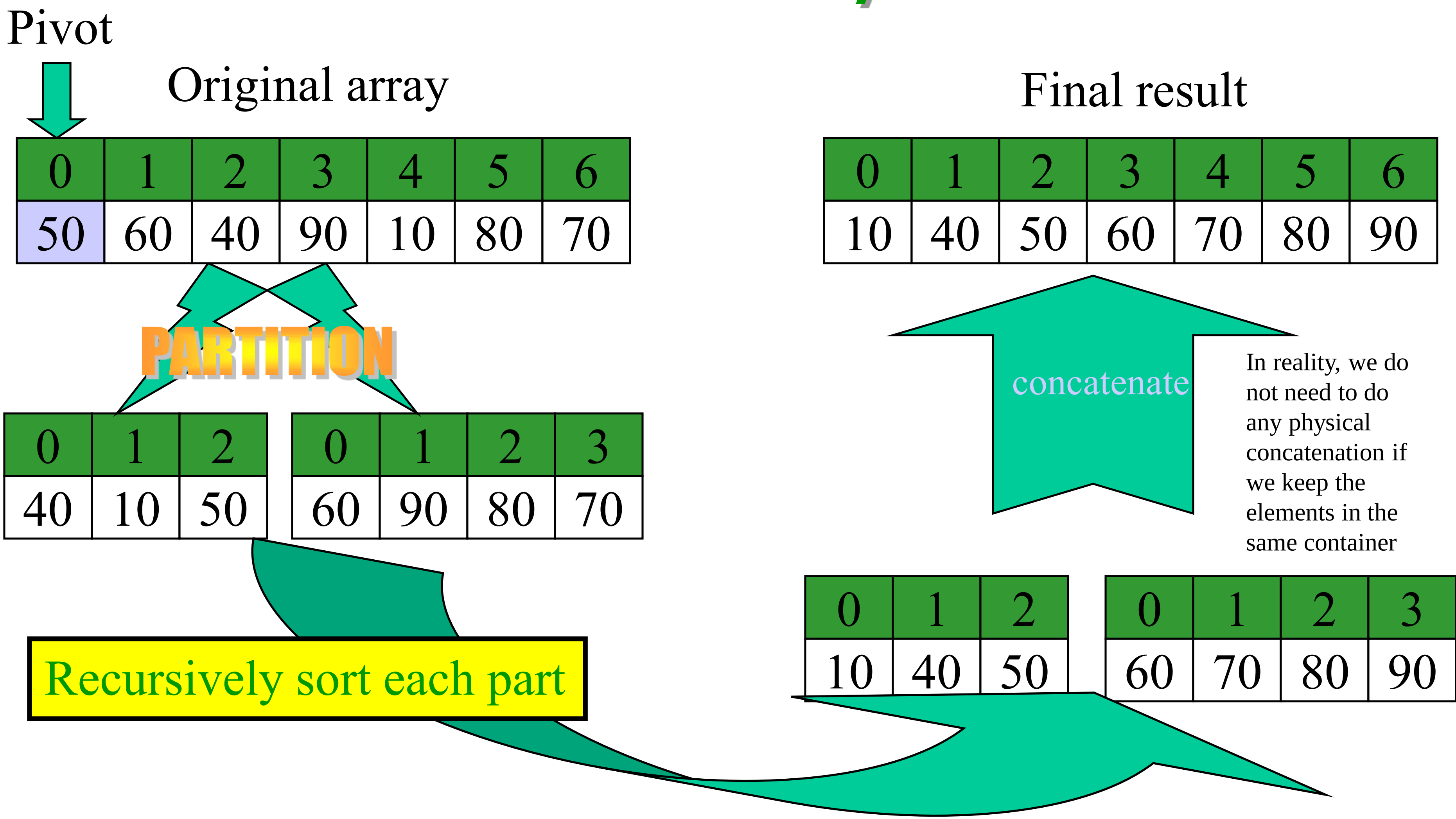
Quicksort – (3) recursively sort large



Quicksort – (4) concatenate



Quicksort Algorithm – summary



Quicksort – in place sorting

0	1	2	3	4	5	6
10	40	50	60	90	80	70

0	1	2	3	4	5	6
10	40	50	60	70	80	90

quicksort front part

quicksort back part

0	1	2	3	4	5	6
40	10	50	60	90	80	70

0	1	2	3	4	5	6
40	10	50	60	90	80	70

Eliminating copying of elements in temporary storage

- It is possible to re-arrange the values in **`anArray`** so that:
  - **`pivot`** is in its final position (**`pivotIndex`**)
  - All values in positions < **`pivotIndex`** are smaller than **`pivot`**
  - All values in positions > **`pivotIndex`** are greater than **`pivot`**

- using only <u>one</u> temporary variable.

Quicksort Algorithm

- Partition `anArray` in-place so that the pivot is in its correct final position, `pivotIndex`, all smaller values are to its left, and all larger values are to its right.

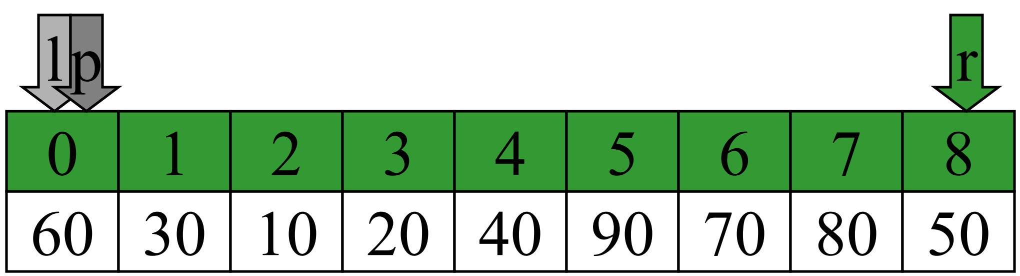
- quicksortP(`anArray,first,pivotIndex-1`)
- quicksortP(`anArray,pivotIndex+1,n-1`)

Concept behind the Partition Algorithm

- Usually we Partition an array in Quicksort using the First (Leftmost) Element as the Pivot
- We work with 2 Indices Left (L) and Right (R) in addition to the Pivot (P)
- L & P are initially the Leftmost element, while R is the index of the Rightmost element; following this at each step we either increment L or decrement R & exchange with element at P depending on certain conditions
- Details follow in the next few slides

In-place Partition Algorithm (1)

- Our goal is to move one element, the pivot, to its correct final position so that all elements to the left of it are smaller than it and all elements to the right of it are larger than it.
- We will call this operation partition().
- We select the left element as the pivot.



In-place Partition Algorithm (2)

- Find the rightmost element that is smaller than the pivot element.

l p								r r
0	1	2	3	4	5	6	7	8
60	30	10	20	40	90	70	80	50

- Exchange the elements and increment the left.

	l							p r
0	1	2	3	4	5	6	7	8
50	30	10	20	40	90	70	80	60

In-place Partition Algorithm (3)

- Find the leftmost element that is larger than the pivot element.

0	1	2	3	4	5	6	7	8
50	30	10	20	40	90	70	80	60

- Exchange the elements and decrement the right.

0	1	2	3	4	5	6	7	8
50	30	10	20	40	60	70	80	90

In-place Partition Algorithm (4)

- Find the rightmost element that is smaller than the pivot element.

0	1	2	3	4	5	6	7	8
50	30	10	20	40	60	70	80	90

- Since the right passes the left, there is no element and the pivot is the final location.

Code for quickSort

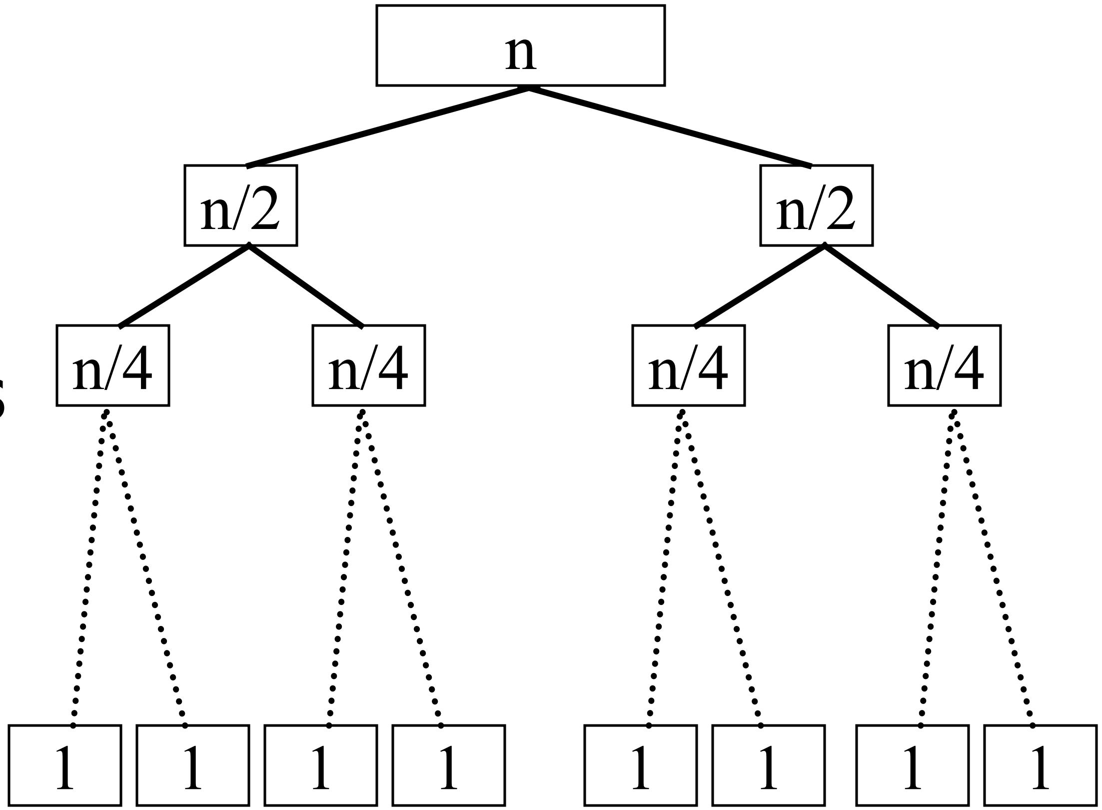
```
def quickSort(data):
    # Sort myself using a quick sort.
    quickSort_helper(data,0,len(data)-1)

def quickSort_helper(data,first,last):
    if first<last:
        pivot=partition(data,first,last)   # partition around a pivot
        quickSort_helper(data,first,pivot-1)   #sort 1st half
        quickSort_helper(data,pivot+1,last)   # sort 2nd half
```

In-place Partition

```
def partition(data,first,last):
    pivotValue=data[first]      # choosing the pivot as the first element in the list
    leftMark=first+1            # leftMark indicates the end of the first partition (+1)
    rightMark=last              # rightMark indicates the beginning of the second partition
    done = False

    while not done:
        while leftMark<= rightMark and data[leftMark] <= pivotValue:
            leftMark = leftMark + 1                     # shifting the pointer to the right

        while rightMark >= leftMark and data[rightMark] >= pivotValue:
            rightMark = rightMark – 1                   # shifting the pointer to the left

        if rightmark < leftMark:        # the partitioning is done
            done = True
        else:                           # elements blocking the partitioning need to be swaped around pivot
            temp= data[leftMark]
            data[leftMark] = data[rightMark]
            data[rightMark]=temp

    temp= data[first]                   # putting pivot in place
    data[first] = data[rightMark]
    data[rightMark]=temp

    return rightMark
```

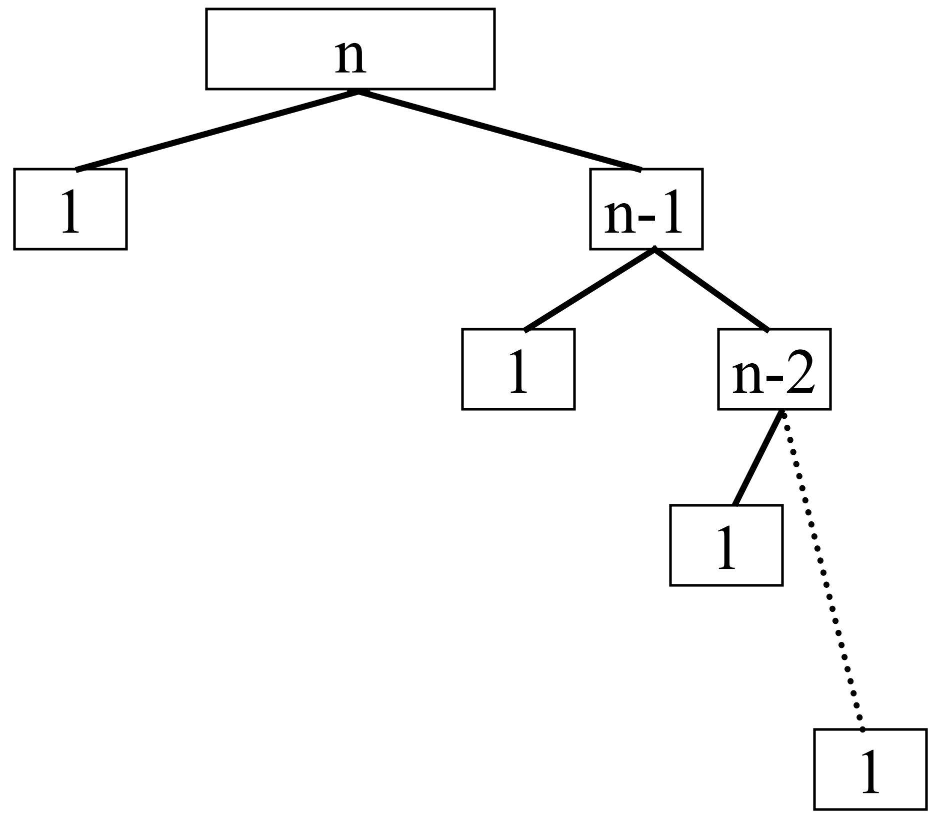
Quicksort - Time Complexity (best)

- Best case: every pivot chosen by quicksort partitions the array into equal-sized parts. In this case quicksort is the same big-O complexity as mergesort – O(n*log(n))



Quicksort - Worst Case

- Worst case: the pivot chosen is the largest or smallest value in the array. Partition creates one part of size 1 (containing only the pivot), the other of size n-1.



Quicksort Time Complexity - Worst Case

- There are n-1 invocations of Quicksort (not counting base cases) with arrays of size

  $p = n, (n-1), \ldots 2$

- Since each of these does $O(p)$ accesses the total number of accesses is

  $O(n) + O(n-1) + \ldots + O(1) = O(n^2)$

- Ironically, the worst case occurs when the list is sorted (or near sorted)!

Comparisons and Accesses - Average Case

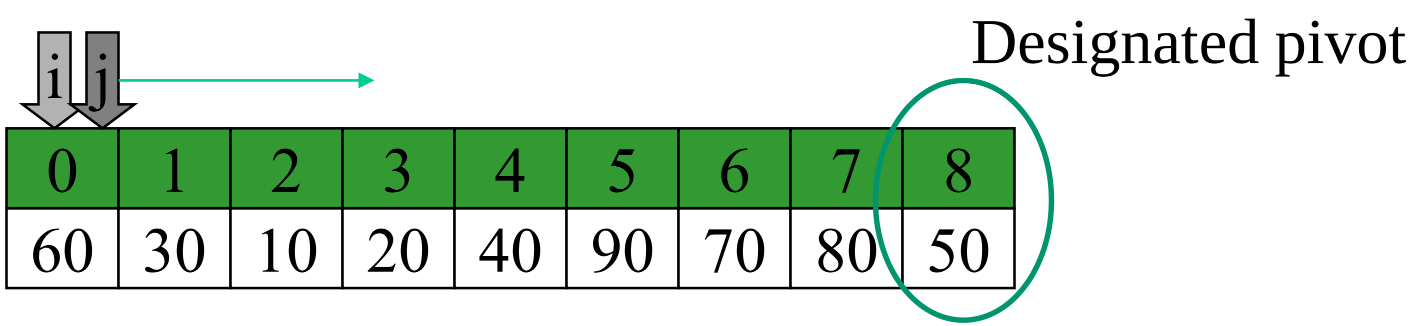
- The average case must be between the best case and the worst case, but since the best case is O(n log(n)) and the worst case is is $O(n^2)$, some analysis is necessary to find the answer.
- Analysis yields a complex recurrence relation.
- On average, the elements are in random order after each partition so about half should be smaller than the pivot and about half should be larger, so the average case is more like the best case.
- The average case number of comparisons turns out to be approximately: 1.386*n*log(n) - 2.846*n
- Therefore, the average case time complexity is: O(n log(n)).

Time Complexity of Quick Sort

- Best case $O(n \log(n))$
- Worst case $O(n^2)$
- Average case $O(n \log(n))$
- Note that the quick sort is inferior to insertion sort and merge sort if the list is sorted, nearly sorted, or reverse sorted.
- However, on the average, i.e., if you need to sort many times and different arrangements are more or less equally likely, Quick Sort is even faster than Merge Sort because it does not need to copy values from one list onto another as an intermediate step.

Variations on the Partitioning

```
algorithm Lomuto-partition(A, first, last)
      pivot := A[last]  // last element volunteered as pivot
      i := first // place for swapping
      for j := first to last – 1 do
            if A[j] ≤ pivot then
                  swap A[i] with A[j]
                  i := i + 1
      swap A[i] with A[last]
      return i
```



Last is pivot; traverse container with j and swap any smaller than pivot to position and increment i; at the end I is position for pivot.

Summary of Lomuto partitioning

- The position i of the pivot is guessed from left to right
- The index j scans the whole array and whenever we find an element A[j] smaller than the pivot, we do a swap with position i.
- When a swap is done i is incremented
- At the end the pivot is put in its position i

Variations on the Partitioning

```
algorithm Hoare-partition(A, first, last)
        pivot := A[first] // first element volunteered as pivot
        l := first - 1    // uses 2 approaching indices i and j
        r := last + 1
        while True do
                repeat
                        r := r – 1
                until A[r] ≤ pivot
                repeat
                        l := l + 1
                until A[l] ≥ pivot
                if l < r then
                        swap A[l] with A[r]
                else
                        return r
```

Summary of Hoare partitioning

- The indices l and r run towards each other until they cross, which indicates the final position of the pivot
- This effectively divides the array into two parts: A left part (small) which is scanned by l and a right part (large) scanned by r.
- a swap is done for every pair of “misplaced” elements, i.e. a large element (larger than pivot, thus belonging in the right partition) which is currently located in the left part and a small element located in the right part.