Computational Thinking and Algorithms
159.171
Quick sort

Amjed Tahir

a.tahir@massey.ac.nz

Previous contributors: Catherine McCartin & Giovanni Moretti

Divide and conquer

Small problem: solve directly

Large problem: divide into subproblems,
solve each subproblem
combine solutions

The process is **recursive**, if a subproblem is still large then we invoke the divide and conquer process again on the subproblem.

Once we reach a situation where our subproblems are "small" then we solve directly and **recursively** combine solutions back up to a solution for the original problem.

Quicksort

Divides the list to be sorted into two parts, then sorts each part.
Division process is called **partition.**

Sizes of parts can range from nearly equal to highly unequal.
Division depends on the **partition element.**

Quicksort begins by partitioning the list to be sorted.

12	30	21	8	6	9	7

Quicksort

Divides the list to be sorted into two parts, then sorts each part.
Division process is called **partition.**

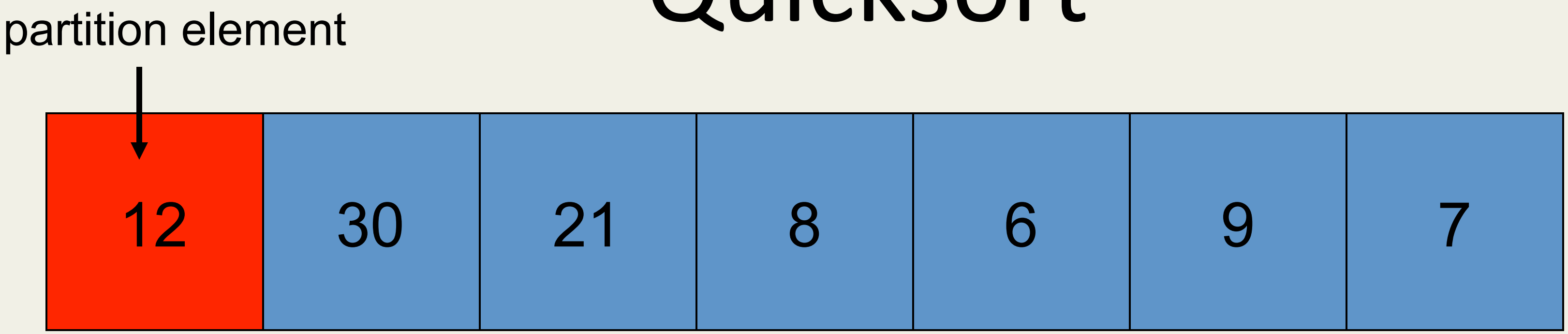
Sizes of parts can range from nearly equal to highly unequal.
Division depends on the **partition element.**

Quicksort begins by partitioning the list to be sorted.

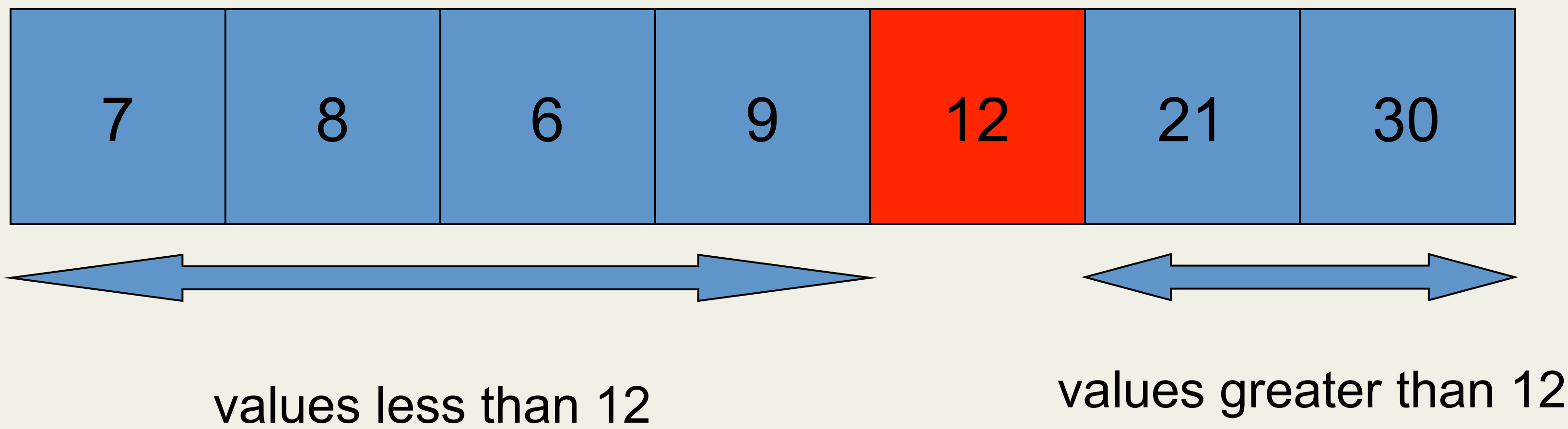
12	30	21	8	6	9	7

Suppose partition element is 12

Quicksort



We want to end up with the following situation after one partition process.



Quicksort

Divides the list to be sorted into two parts, then sorts each part.
Division process is called **partition.**

Sizes of parts can range from nearly equal to highly unequal.

The partitioning can be done

In-place

- data is moved but there's only a single list
- uses less storage
- harder to understand

Using multiple lists

- data is moved to multiple lists (with objects this is not a great penalty)
- easy to understand
- handles duplicate items

Quicksort – Multiple list version

sortedData = quicksort(L)   - returns sorted L. If L = [item] or [], return L

```
quicksort(L):
    if L contains a single item or is empty:
        return L                          # Base case – L is already sorted
    otherwise:
        - pick an item in L as the pivot
        - create three lists by scanning data
            same:    elements == pivot
            lower:   elements < pivot     # must be shorter than L, or [ ]
            greater: elements > pivot     # must be shorter than L, or [ ]

        return quicksort(lower) + same + quicksort(greater)
```

multi-list Quicksort [12,30,21,8,6,9,7]

Choose middle element as pivot: 12 30 21 **8** 6 9 7

quicksort [12,30,21, **8**, 6, 9, 7]
pivot = len(L)//2 = L[3] = 8
    lower **[6, 7]**
    same [**8**]
    greater [12, 30, 21, 9]

    quicksort [6, 7]
    pivot = len(L)// 2 ➔ L[1] = **7**
        lower [6]
        same [**7**]
        greater [ ]
    returns [6] + [7] + [ ]

return: [6, 7] + [8] + [9, 12 , 21, 30]

quicksort [12, 30, 21, 9]
pivot is L[2] = 21
    lower **[12, 9]**
    same [21]
    greater [30]

    quicksort [12, 9]
    pivot = len(L)// 2 ➔ L[1] = 9
        lower [ ]
        same [**9**]
        greater [12]

returns [9,12] + [21] + [30]

Multi-list Quicksort implementation

```
def quicksort(L):
    if len(L) <= 1:
        return L
    else:
        pivot = L[len(L)//2]

        lower = []
        same  = []
        greater = []
        for item in L:
            if item == pivot:
                same.append(item)
            elif item < pivot:
                lower.append(item)
            else:
                greater.append(item)

        return quicksort(lower) + same + quicksort(greater)
```

Analysis for Quicksort

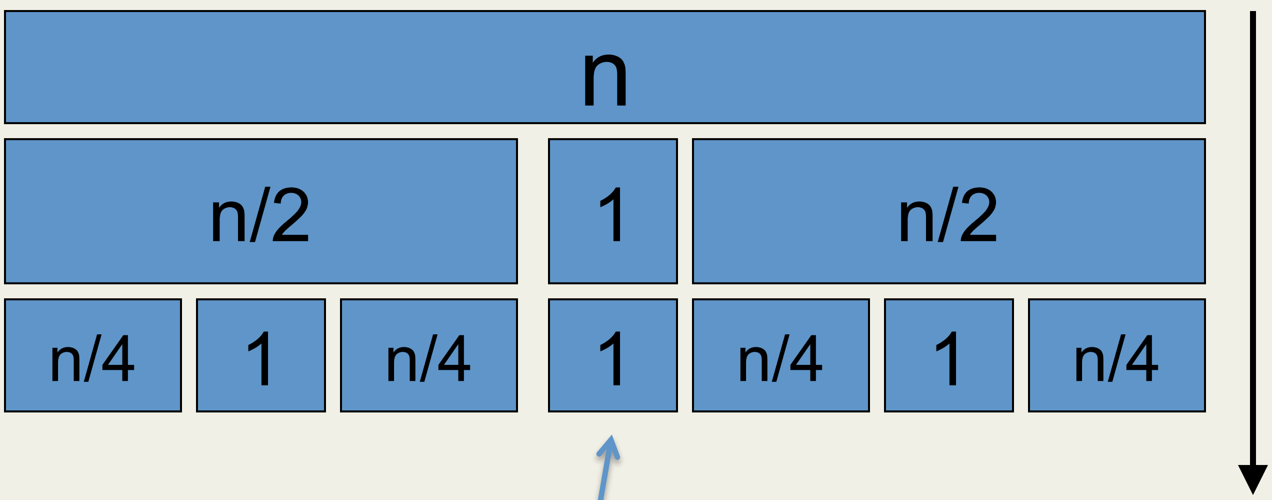
Quicksort for a list of length n

expected running time is $O(n \log n)$

worst-case running time is $O(n^2)$

what happens if our partition element is the largest element in the list?

what happens if the list is in reverse order to start with?



Expect to have **log(n)** levels

at each level do **O(n)** work

Assumes no duplicates

Quicksort – Pivot choice matters

Quicksort performance is generally O(n * log(n))

BUT: **a common choice is to use the first element as the pivot**

This is REALLY BAD if the data already sorted

Pivot will be the minimum value
lower → [ ]
same → [min-value]
greater → [N-1 elements, all still sorted]

The quicksort performance degenerates to an O(n**2) sort
& recursion depth is N, which often exceeds system limit

If recursive calls balanced (i.e. len(lower) ~= len(greater),
each pass has n/2 items

Stable Sorts

A sort is called *STABLE* if it does not alter the order of elements with the same key

This allows the results of an earlier sort to be preserved

"A sorting algorithm is stable if whenever there are two records R and S with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list"
from https://en.wikipedia.org/wiki/Sorting_algorithm#Stability

There are many sort algorithms

There are numerous variations of sorting algorithms. The sort algorithms discussed so far illustrate different approaches, and have different O(n) properties.

OPTIONAL: ways to alter/extend Quicksort

e.g. there are other implementation/partitioning schemes and ways to handle duplicates in the QuickSort. https://en.wikipedia.org/wiki/Quicksort

OPTIONAL: the Counting Sort. The sort on the following pages – the *Counting Sort* – shows a quite unusual and non-obvious type of sort.

The material in the followings slides

(an alternative Quicksort implementation & the Counting Sort)

is OPTIONAL

and

will NOT be in the exam

In-place Quicksort – common

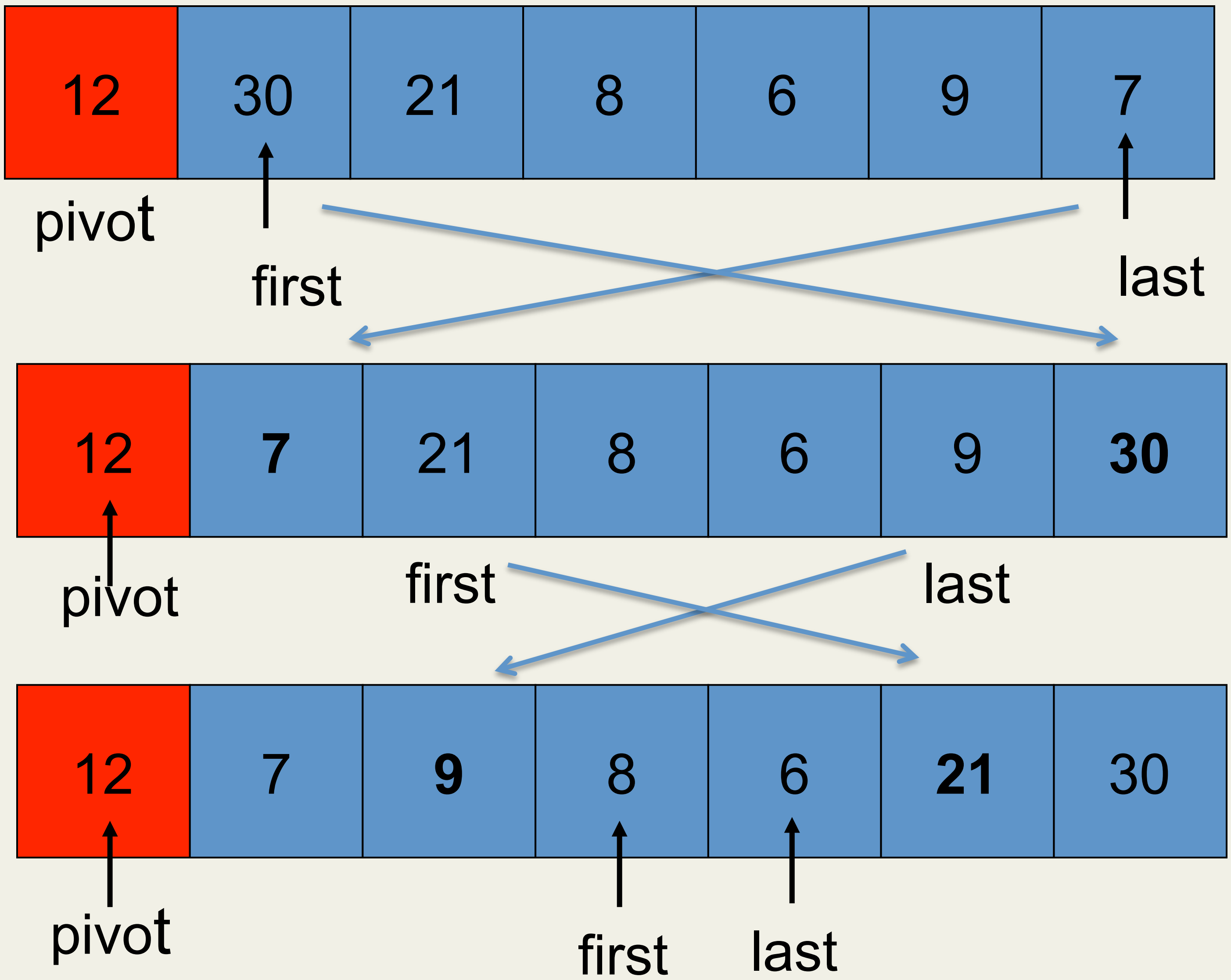
Divides the list to be sorted into two parts, then sorts each part.
Division process is called **partition.**

Sizes of parts can range from nearly equal to highly unequal.
Division depends on the **partition element.**

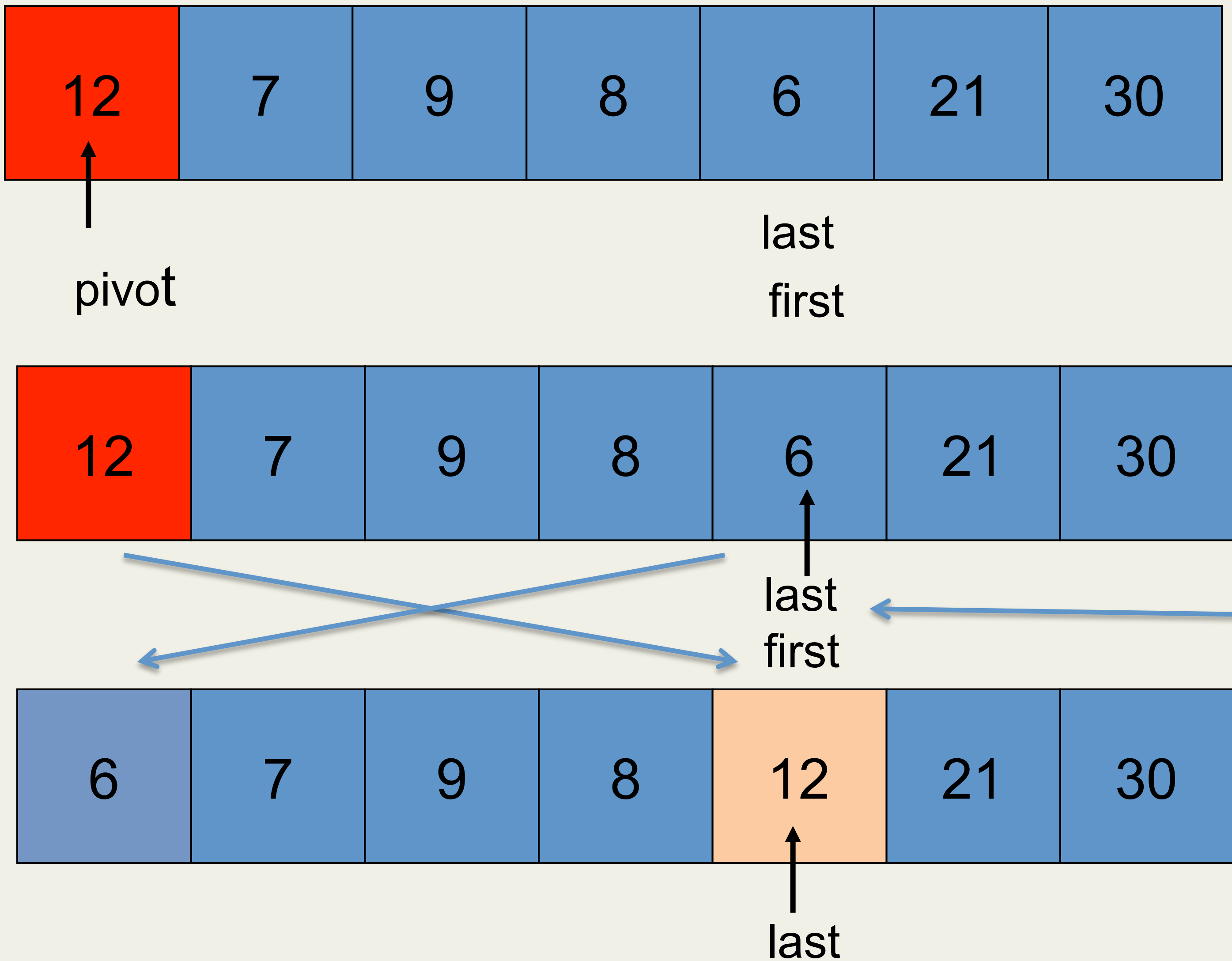
Quicksort begins by partitioning the list to be sorted.

12	30	21	8	6	9	7

Suppose partition element is 12

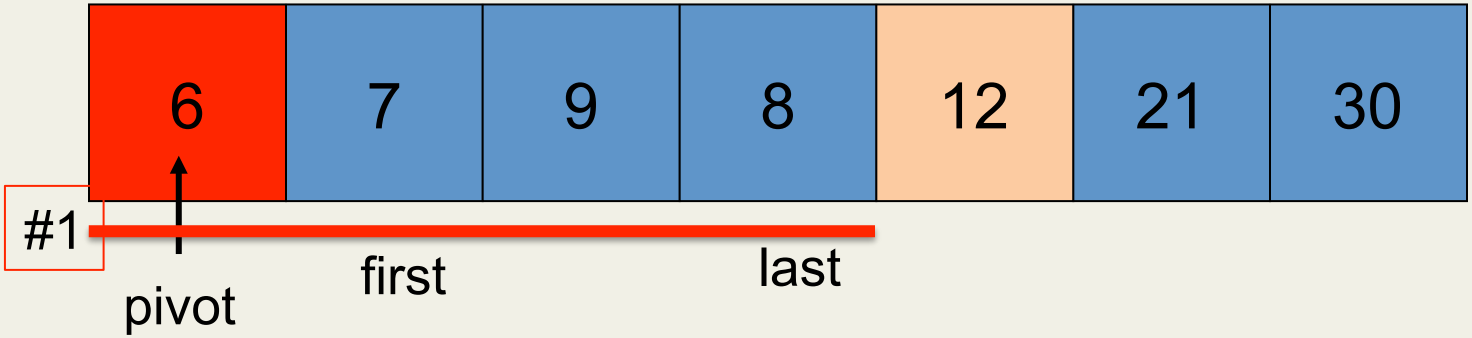


first & **last** move towards each other until a pair of element that can be exchanged is found or until they're the same

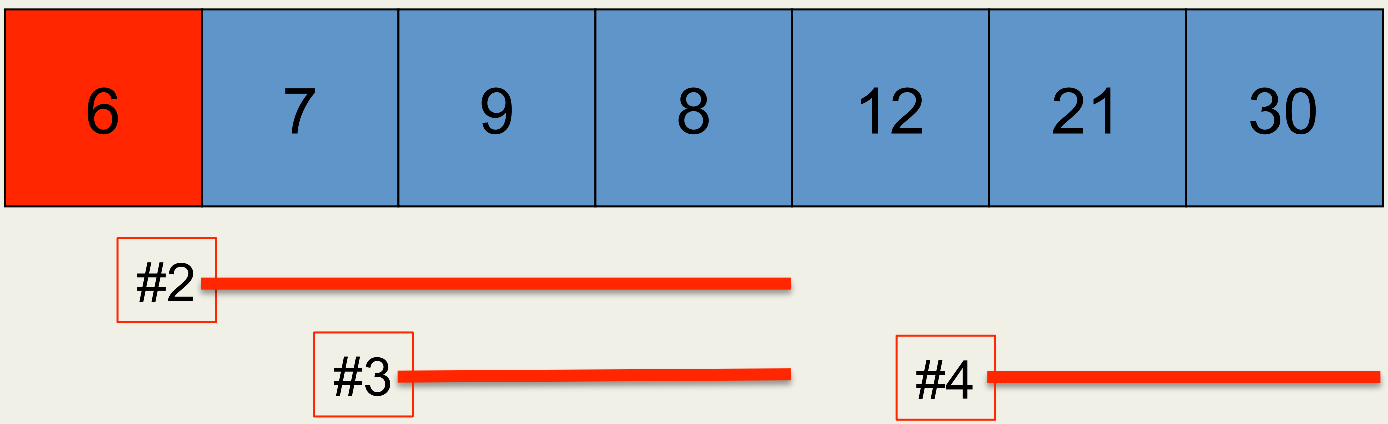


Stop when first == last

Then swap pivot and last values and exit



12 is in the right place
Now repeat process on left part(#1)



Then recursively on smaller sub-divisions of left (#2, #3) , then right part (#4)

Quicksort – a partition function

```
def partition(data, first, last):          # first & last specify a slice data list to work on
    pivot = data[first]                    # choose leftmost element as the pivot
    left, right = first + 1, last

    while left <= right:
       while left < right and data[left] < pivot:       # Find first key > pivot
           left += 1
       while right >= left and data[right] >= pivot:  # Find rightmost key < the pivot
           right -= 1

        if left < right:                          # Need to exchange left/right?
            data[left], data[right] = data[right], data[left]
        elif left == right:                       # if left-right meet, we've finished
            break

    if right != first:                            # finally, put pivot element in place
        data[first], data[right] = data[right], data[first]

    return right                                  # return the index of pivot
```

Quicksort implementation

```
def quicksort(list, bottom, top):
    # if there are two or more elements...
    if bottom < top:
        # ... partition the sublist...
        split = partition(list, bottom, top)
        # ... and sort both halves
        quicksort(list, bottom, split-1)
        quicksort(list, split+1, top)
    else:
        return
```

call this as:

```
  quicksort(the_list, 0, len(the_list)-1)
```

IMPORTANT
This Quicksort does NOT handle duplicate elements (repeated values) in data.

Counting sort

Uses specific knowledge about the input. Good for “small” ranges of items.

5	8	3	8	10	7

original list

we want count[k] = number of occurrences of value k in original list

start by initialising count list to be all zeros

0	0	0	0	0	0	0	0	0	0	0

Counting sort

Uses specific knowledge about the input. Good for “small” ranges of items.

5	8	3	8	10	7

count[k] = number of occurrences of value k in list

count[3]= 1, count[5]=1, count[7]=1, count[8]=2, count[10]=1

0	0	0	1	0	1	0	1	2	0	1

Counting sort

Uses specific knowledge about the input. Good for “small” ranges of items.

5	8	3	8	10	7

modify count so that count[k] is equal to number of elements
less than or equal to k in list

count[3]= 1, count[5]=2, count[7]=3, count[8]=5, count[10]=6

0	0	0	1	1	2	2	3	5	5	6

Counting sort

Uses specific knowledge about the input. Good for “small” ranges of items.

5	8	3	8	10	7

0	0	0	1	1	2	2	3	5	5	6

copy items into a new list b,
beginning with last element of list a

		7			

Counting sort

count[7]=3 so copy 7 to b[2], decrement count[7] to 2

		7			

0	0	0	1	1	2	2	2	5	5	6

Counting sort

predecessor of 7 is 10, count[10]=6 so copy 10 to b[5], decrement count[10] to 5

		7			10

0	0	0	1	1	2	2	2	5	5	5

Counting sort

predecessor of 10 is 8, count[8]=5 so copy last 8 to b[4], decrement count[8] to 4

		7		8	10

0	0	0	1	1	2	2	2	4	5	5

Counting sort

predecessor of last 8 is 3, count[3]=1 so copy
3 to b[0], decrement count[3] to 0

3		7		8	10

0	0	0	0	1	2	2	2	4	5	5

Counting sort

predecessor of 3 is 8, count[8]=4 so copy first 8 to b[3], decrement count[8] to 3

		7	8	8	10

0	0	0	0	1	2	2	2	3	5	5

Counting sort

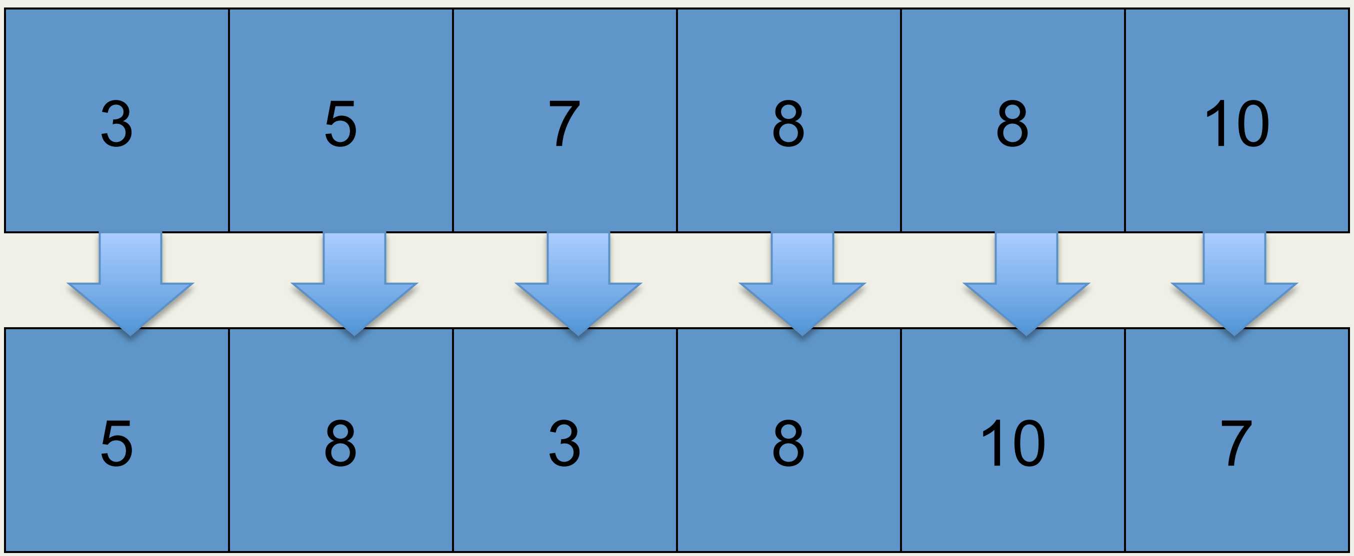
predecessor of first 8 is 5, count[5]=2 so copy
5 to b[1], decrement count[5] to 1

3	5	7	8	8	10

0	0	0	0	1	1	2	2	3	5	5

finally, copy b back to a

Counting sort



finally, copy b back to a

Counting sort

```
def countingSort(A, k):
        # A is the list to sort, k is the size of largest item in A
        # set up new list and counts list
        B = [0 for elem in A]
        C = [0 for elem in range(0, k+1)]
        # set counts to 0
        for i in range(0, k +1):
                C[i] = 0
        # add 1 to correct index in C for each item in A
        for j in range(0, len(A)):
                C[A[j]] += 1
        # modify C so that C[i] contains number of items <= i in A
        for i in range(1, k + 1):
                C[i] += C[i - 1]

        # start at end of A, insert items in B and update C
        for j in range(len(A)-1, 0-1, -1):
                list_item = A[j]
                position = C[list_item]-1
                B[position] = list_item
                C[list_item] -= 1
        # copy B to A
        for j in range(0, len(B)):
                A[i] = B[i]
```

Counting sort

time for each loop is either $\Theta(n)$ or $\Theta(m)$
where n = number of items in A
m = range of items in A

thus, algorithm runs in time $\Theta(m + n)$

if m is $O(n)$ then algorithm runs in linear time, $O(n)$

Note that counting sort is **stable**, two compound elements that are sorted on the same key will be sorted into the same order.

Quicksort is **not** stable.
Exercise: find an example to show this.