Quick Sort

Sorting Algorithm

Charles Carter

Auburn University

May 23, 2022

Table of Contents

Introduction

Why would we ever want to sort anything?

Sorting: arranging objects in some sort of order

- Integers: in numerical order, e.g., 1, 2, 3, 4, 5
- Characters: in alphabetical order, e.g. A, B, C, D, E
- Students: in class rank order
- Movies: in review order, highest to lowest
- Dates: ???

A motivational example

How do you finding a name in a large directory?
What if the names are not sorted?
Do you start at the beginning and search until you find the name you are looking for?
What if there are 1,000,000 names in the directory?
The complexity of bisection search is $O = \log_2 n$

- 4 guesses to find an item in a list of size 10
- 7 guesses in a list of size 100
- 10 guesses in a list of size 1000
- 20 guesses in a list of size 1000000
- 30 guesses in a list of size 1000000000
- 40 guesses in a list of size 1000000000000

Conceptual View of QuickSort

- A list of length 0 or 1 is "sorted." Why?
- A list of length greater than 1 can be split in the "middle" and each "half" can be sorted. Why?
- Sorted lists can be combined (in order) to form a sorted list. Why?

[] and [42] are both sorted lists.
[1, 2, 3] + [4, 6, 8] + [10, 20, 30] is a sorted list.

Brief Description of QuickSort

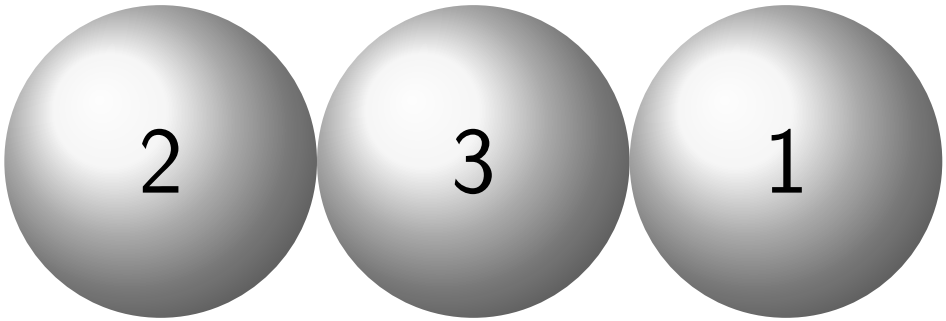
- QuickSort is a recursive algorithm based on the "divide and conquer" technique. It works by placing the "middle" element in the proper place, i.e., in the "middle", and then doing the same thing for the left and right halves. This "middle" element is known as the **PIVOT**.
- Think of sorting a stack of books by number of pages. Pick a book at random (the pivot, or "middle"), and place all the books with fewer pages on the left, and all the books with more pages on the right. Repeat for the left and right sides.
- Question: How would you do this for a deck of cards?

Description of QuickSort

1. Call the QuickSort() procedure, passing a perhaps unsorted list as the only parameter.
2. Start with the list parameter. Pick one element of the list to be the pivot. (First, last, middle, random — it doesn't matter.)
3. Iterate through the list placing all elements less than the pivot on the left and all elements greater than the pivot on the right. (Equal elements can go either on the left, right, or — in my demonstration — in the middle.)
4. Continue the same process for the left and right sides. Call the same procedure passing the left or right side as the parameter. This is the recursive step.
5. Stop the recursion when a list has one or zero elements and return the list. This stops the recursion.

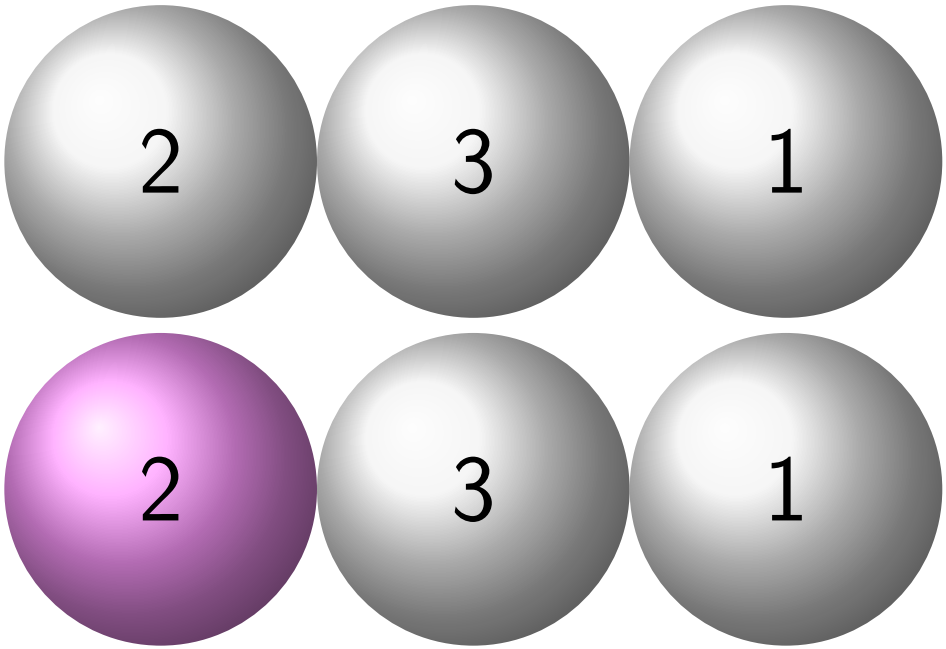
Quick Sort Examples

Example 1



Choose a pivot, use first element

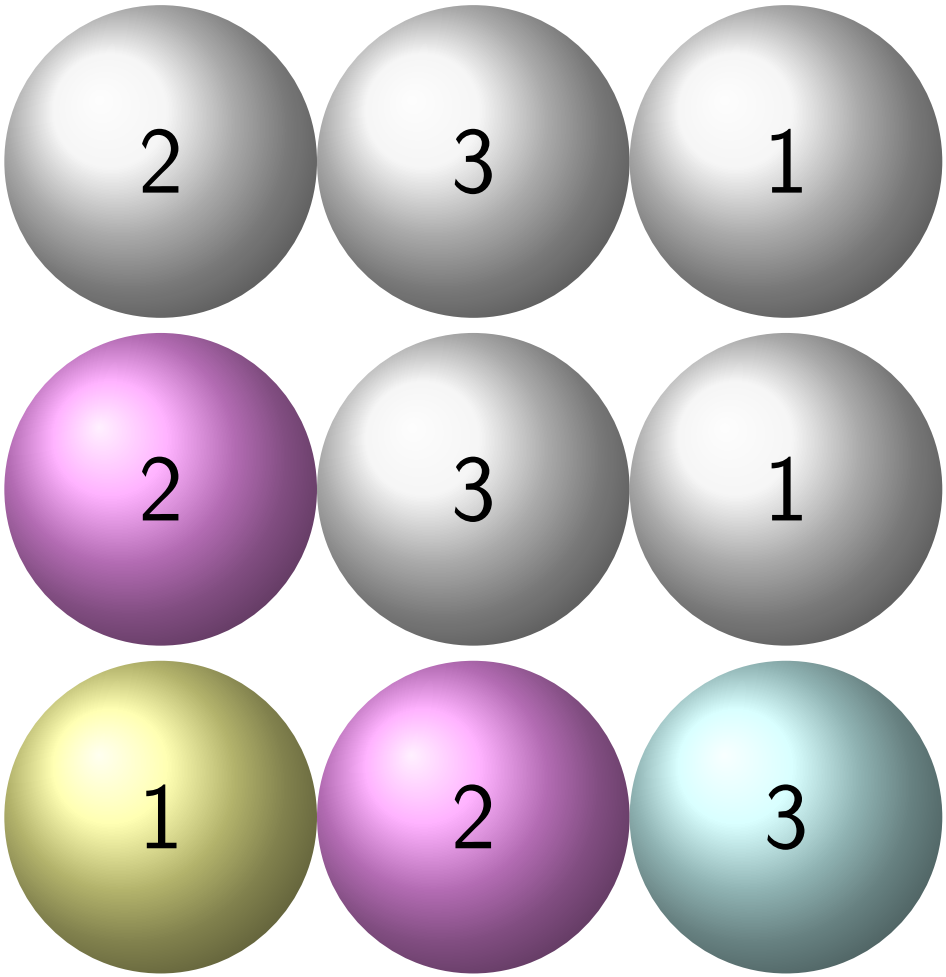
Example 1



Choose a pivot, use first element

Sort right and left lists

Example 1

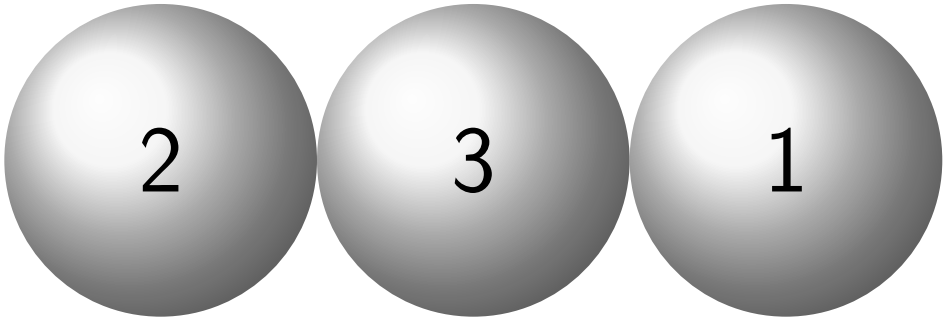


Choose a pivot, use first element

Sort right and left lists

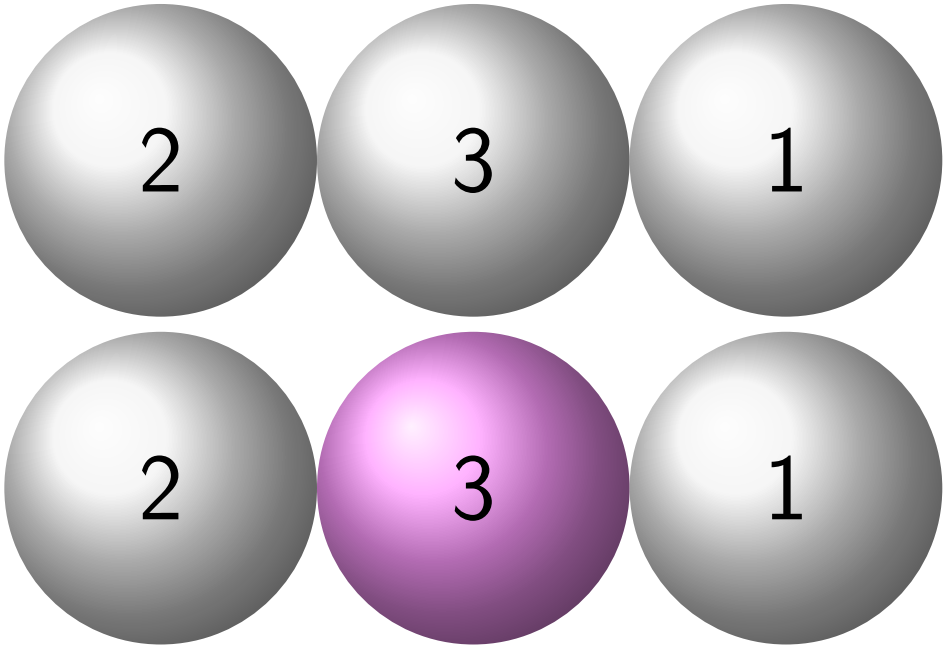
R & L stopped, List is sorted

Example 2



Choose a pivot, use middle

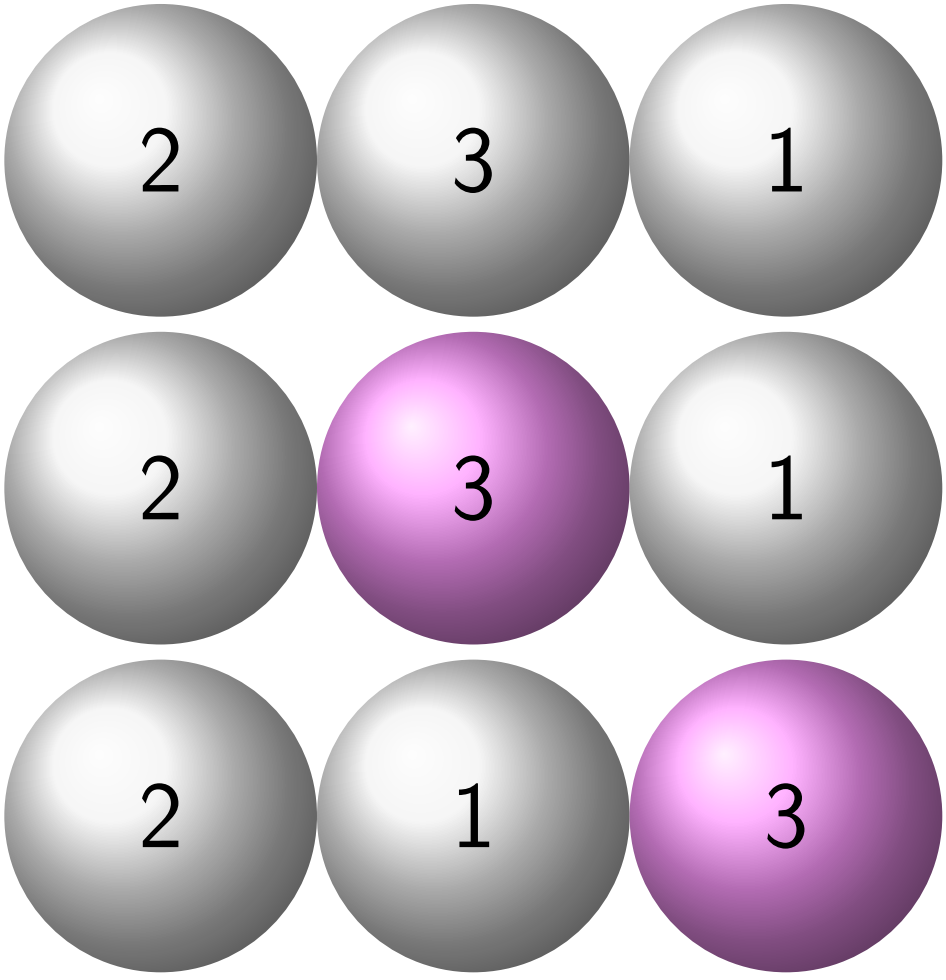
Example 2



Choose a pivot, use middle

Sort right and left

Example 2

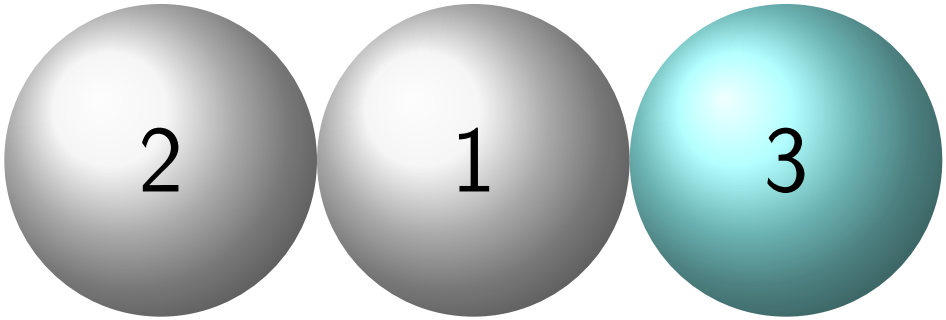


Choose a pivot, use middle

Sort right and left

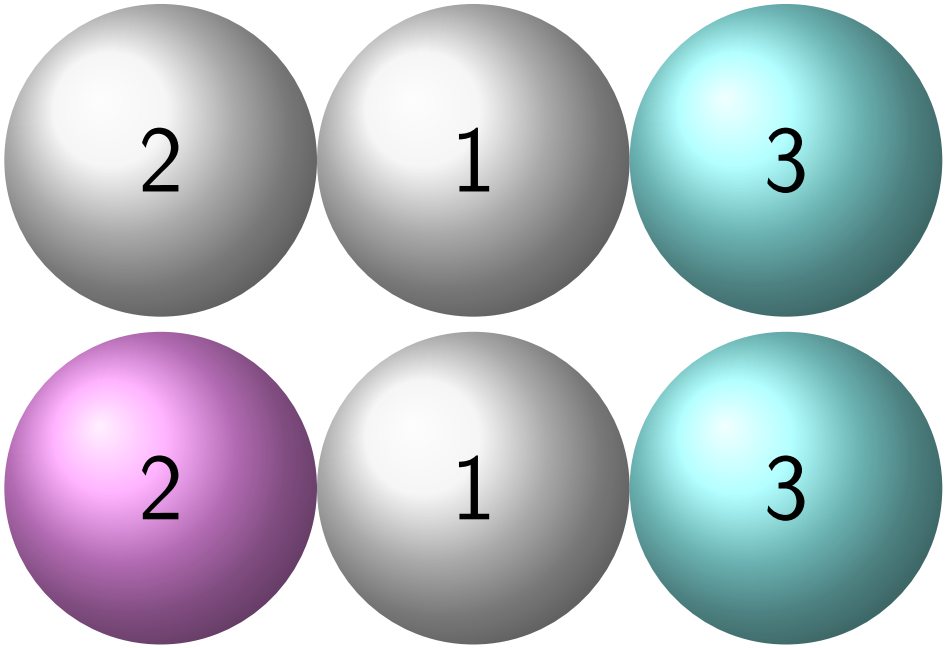
Sort left list

Example 2, continued



Choose a pivot, use middle

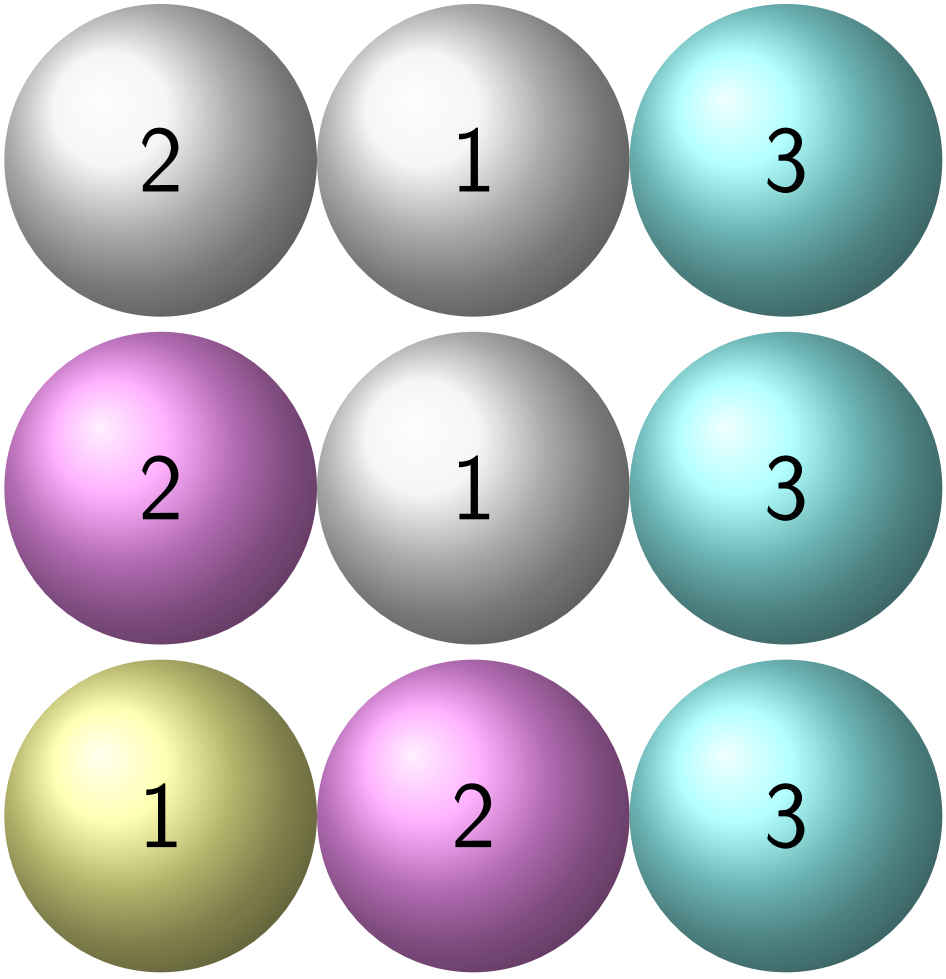
Example 2, continued



Choose a pivot, use middle

Sort right and left

Example 2, continued

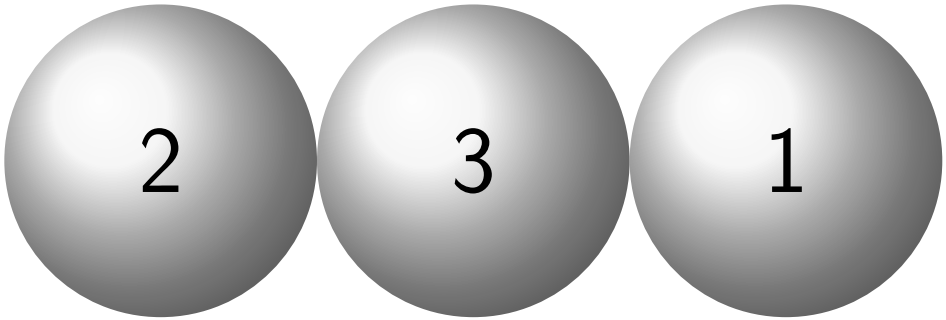


Choose a pivot, use middle

Sort right and left

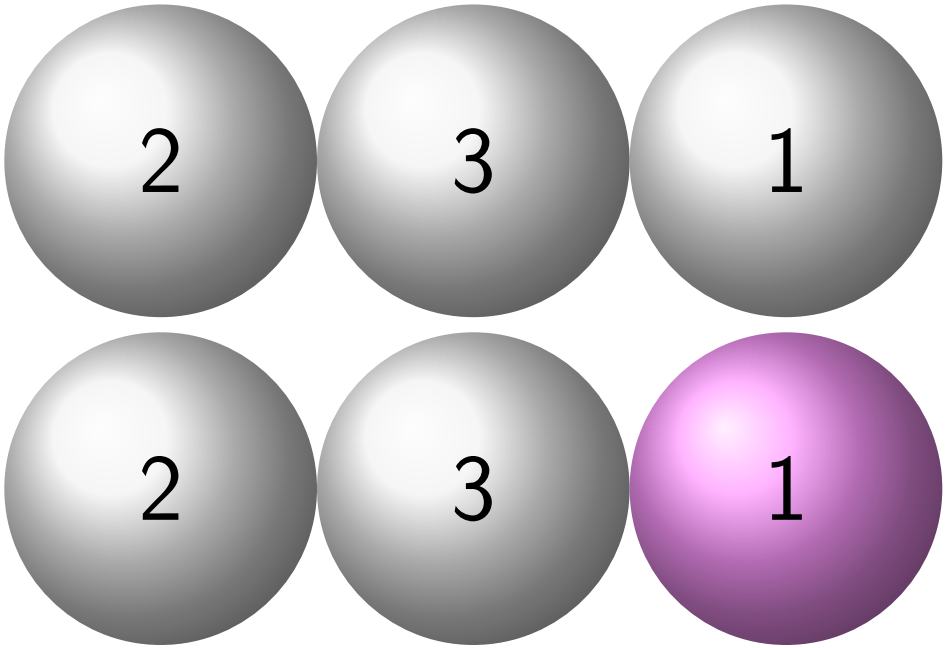
List is sorted

Example 3



Choose a pivot, use last

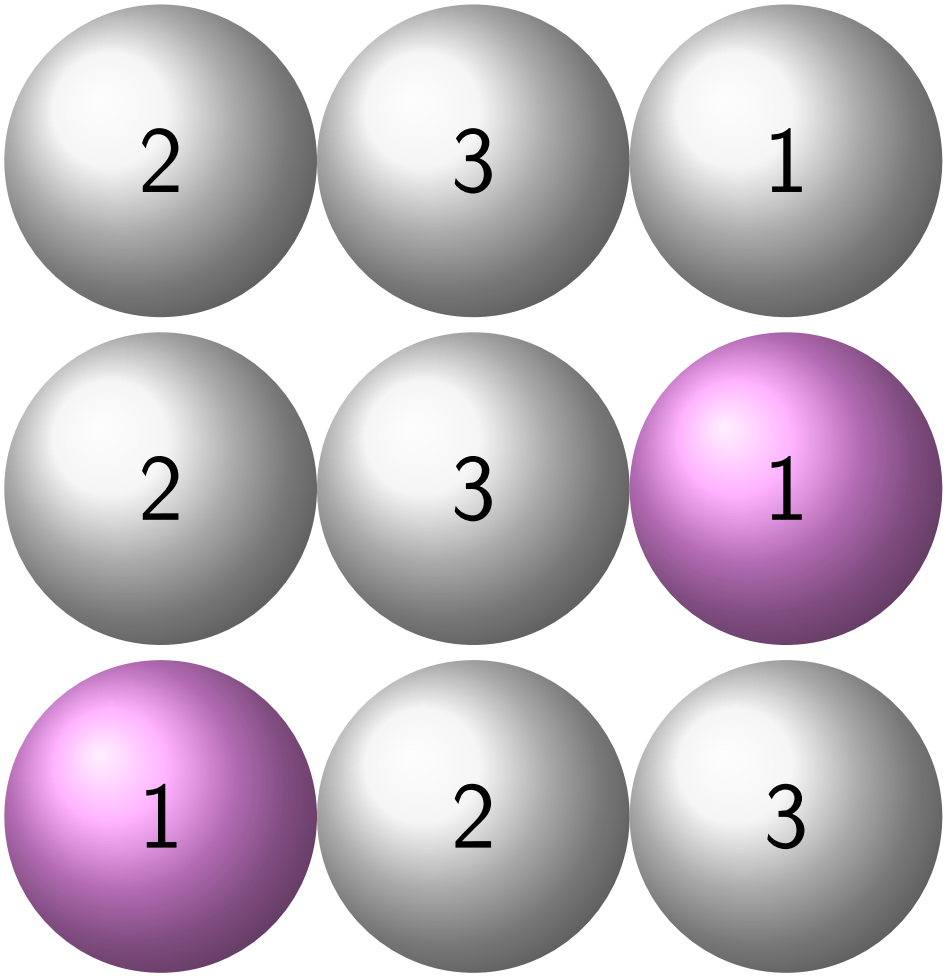
Example 3



Choose a pivot, use last

Sort right and left

Example 3

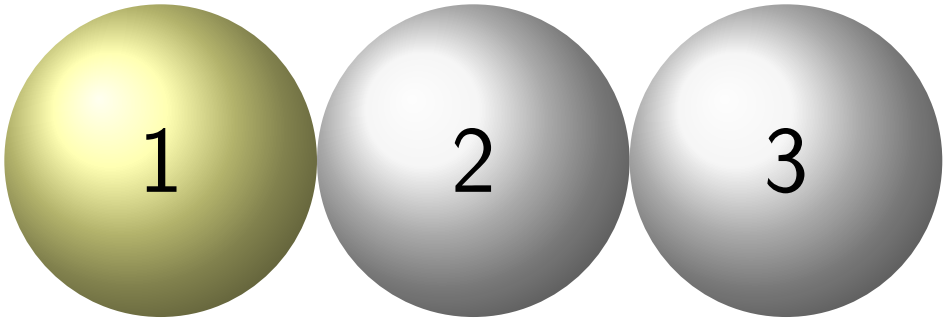


Choose a pivot, use last

Sort right and left

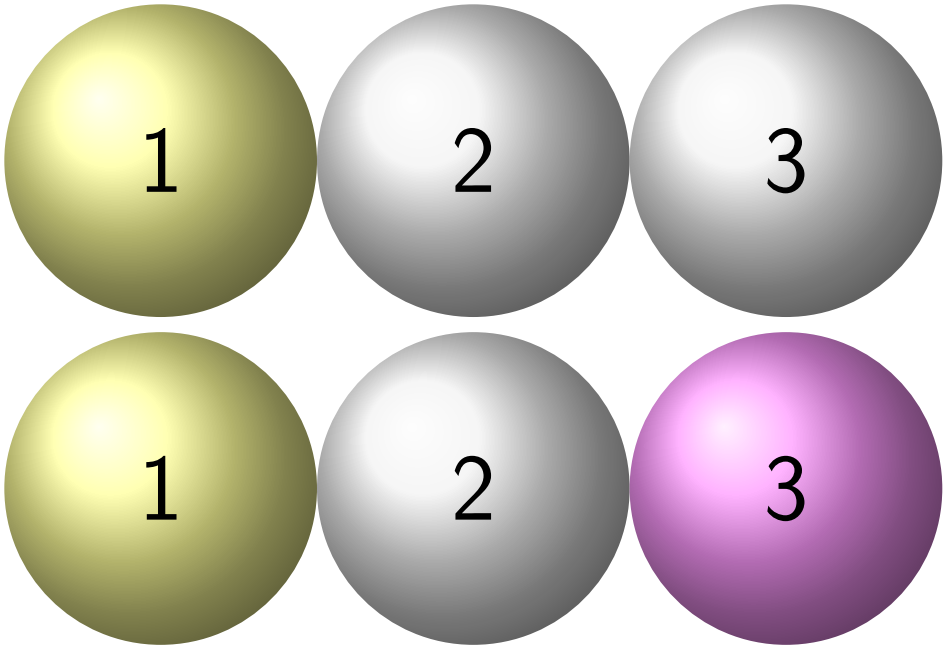
Sort not done, R $>$ 1

Example 3, continued



Choose a pivot, use last

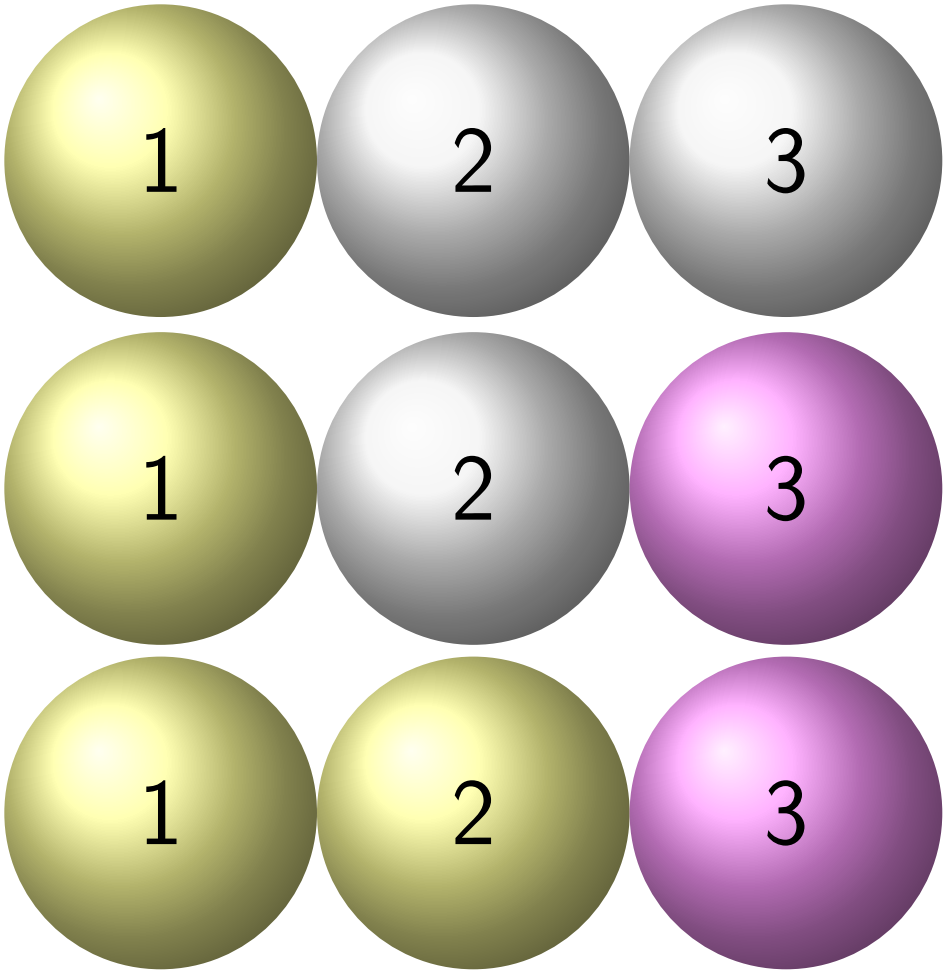
Example 3, continued



Choose a pivot, use last

Sorted, R == 1, L == 0

Example 3, continued

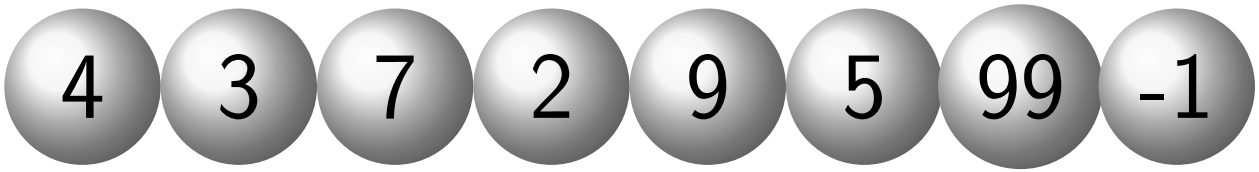


Choose a pivot, use last

Sorted, R == 1, L == 0

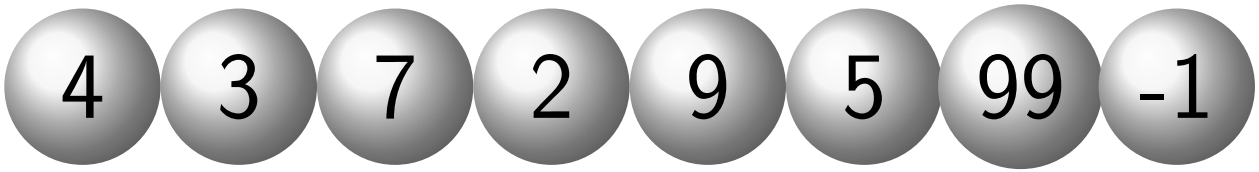
List is sorted

Extended example

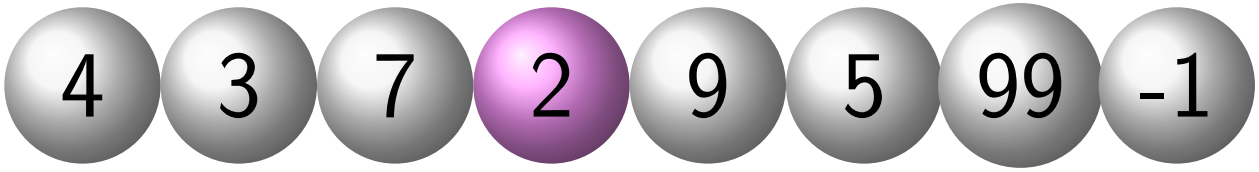


Choose pivot, use middle

Extended example

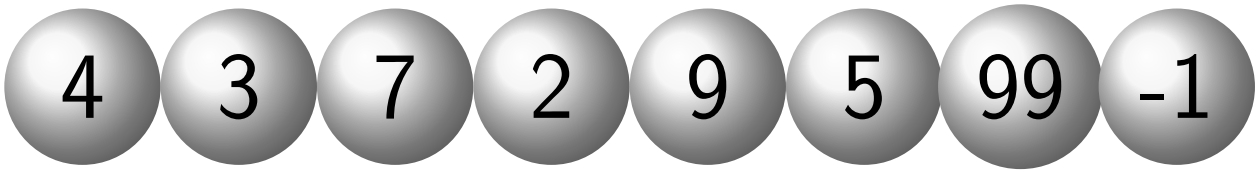


Choose pivot, use middle

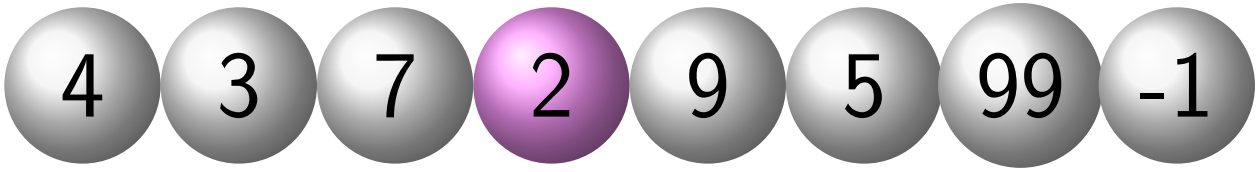


Place elements $<$ pivot on L, elements $>$ pivot on R

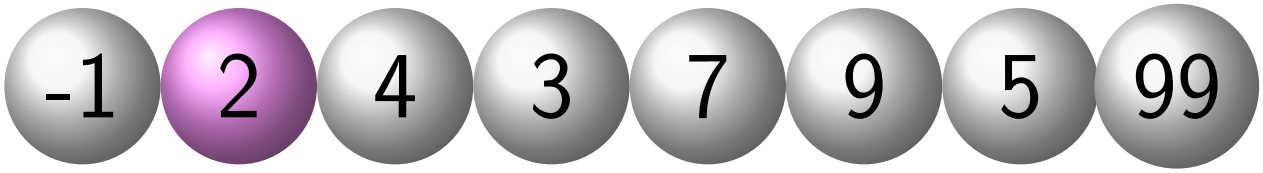
Extended example



Choose pivot, use middle

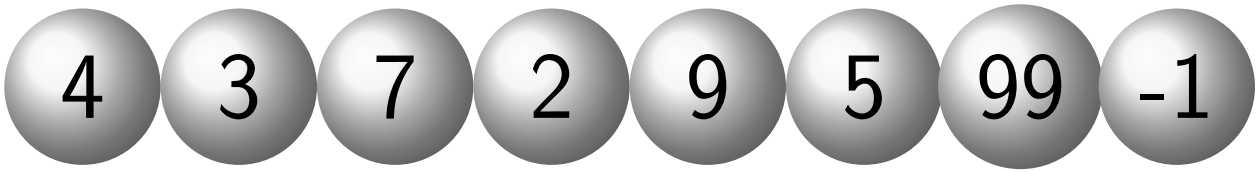


Place elements $<$ pivot on L, elements $>$ pivot on R

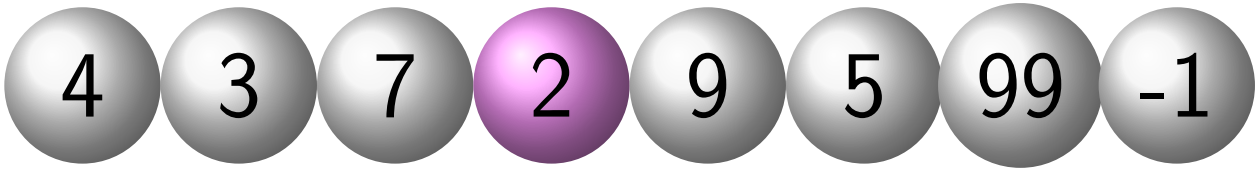


L is sorted, length $==$ 1

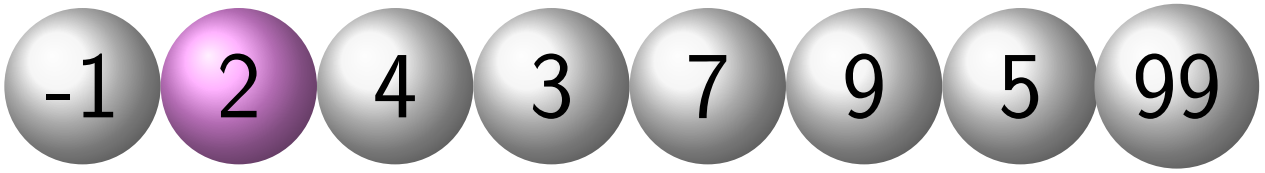
Extended example



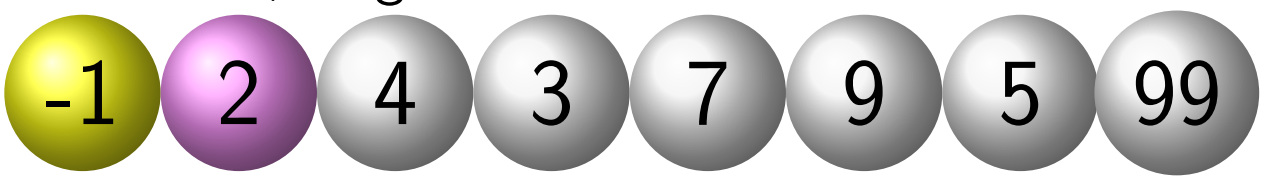
Choose pivot, use middle



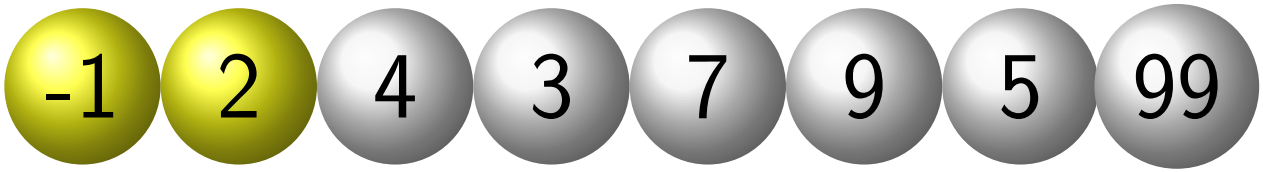
Place elements $<$ pivot on L, elements $>$ pivot on R



L is sorted, length $== 1$

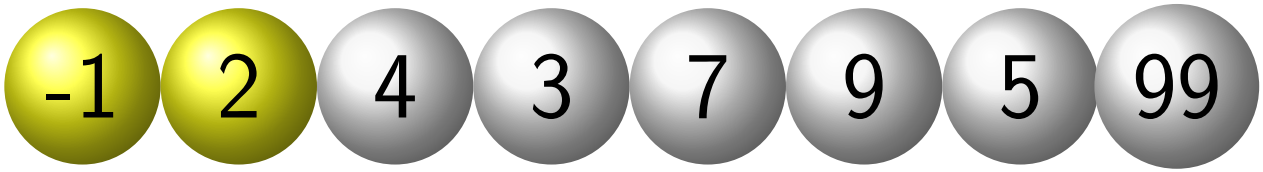


Extended example continued 1

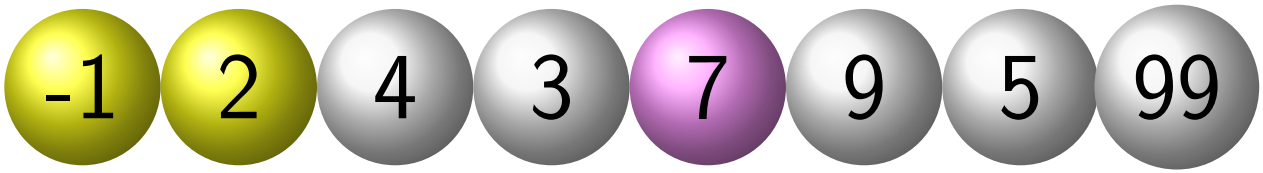


Choose pivot, use middle

Extended example continued 1

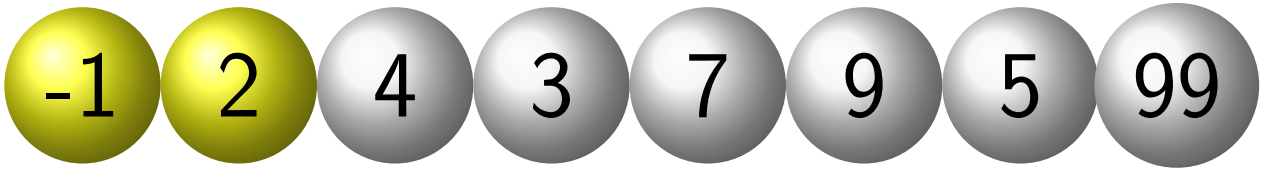


Choose pivot, use middle

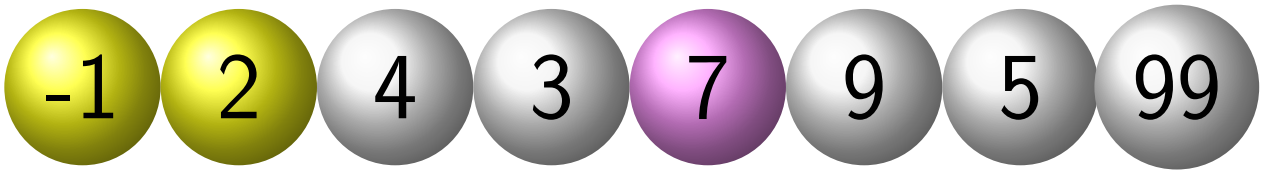


Place elements $<$ pivot on L, elements $>$ pivot on R

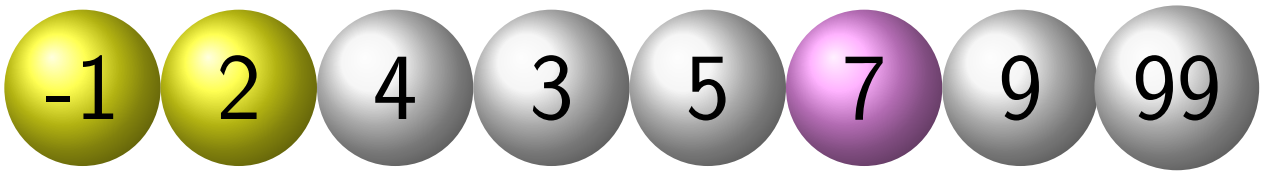
Extended example continued 1



Choose pivot, use middle

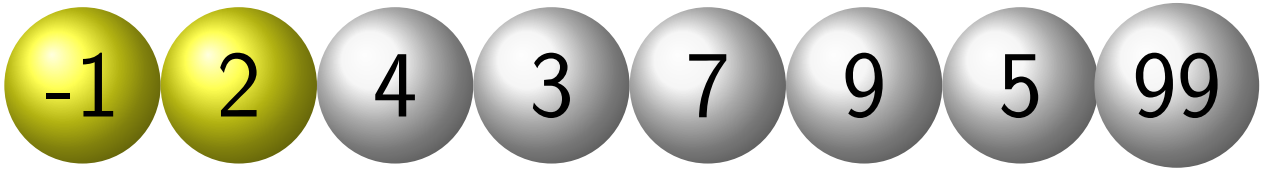


Place elements $<$ pivot on L, elements $>$ pivot on R

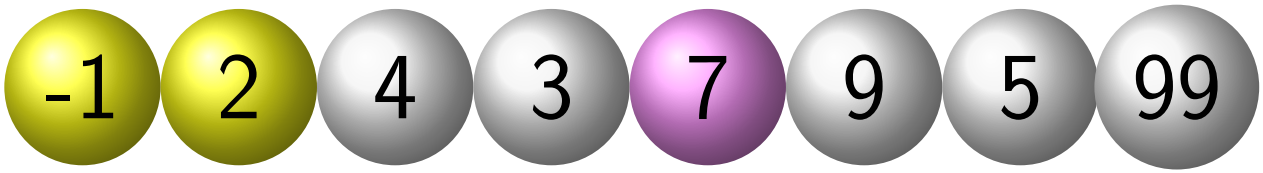


Pivot is in correct place (color is RED)

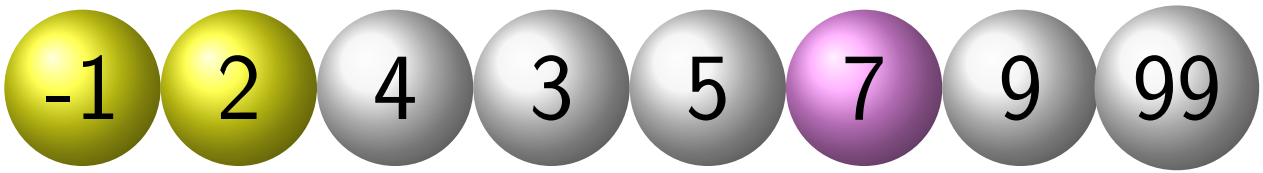
Extended example continued 1



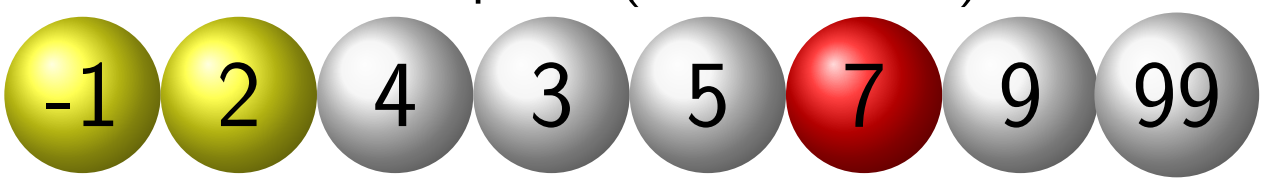
Choose pivot, use middle



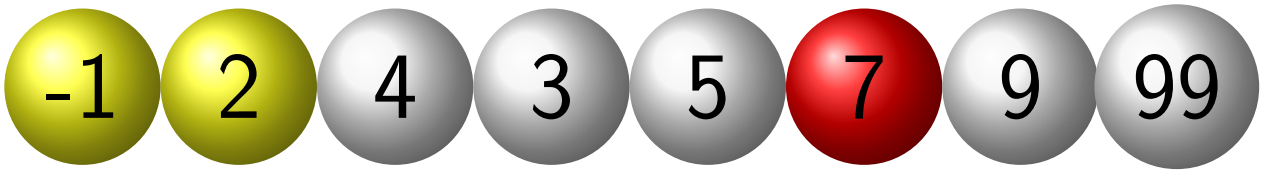
Place elements $<$ pivot on L, elements $>$ pivot on R



Pivot is in correct place (color is RED)

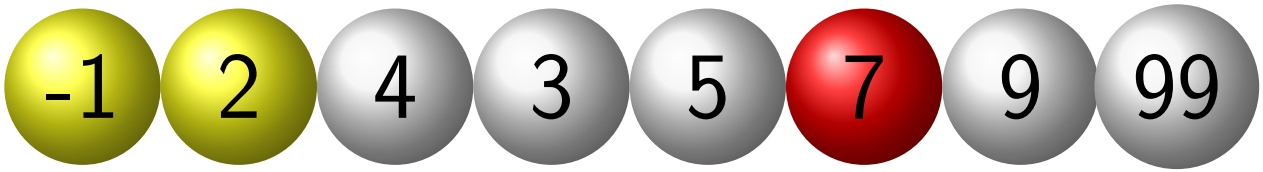


Extended example continued 2

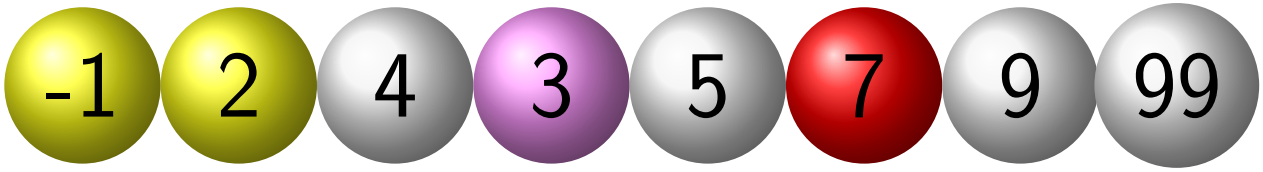


Sorting L, Choose pivot, use middle

Extended example continued 2

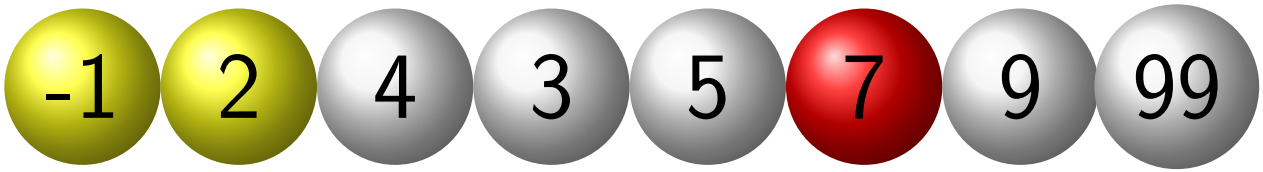


Sorting L, Choose pivot, use middle

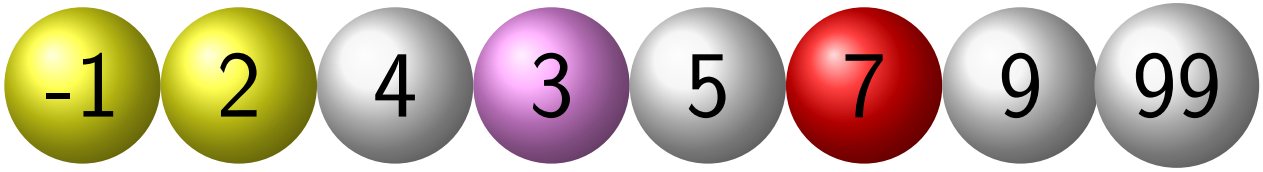


Place elements $<$ pivot on L, elements $>$ pivot on R

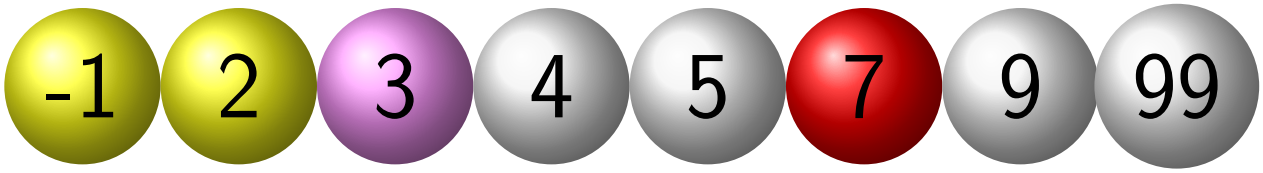
Extended example continued 2



Sorting L, Choose pivot, use middle

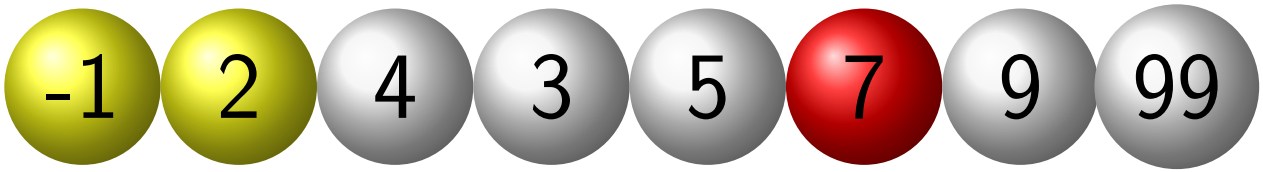


Place elements $<$ pivot on L, elements $>$ pivot on R

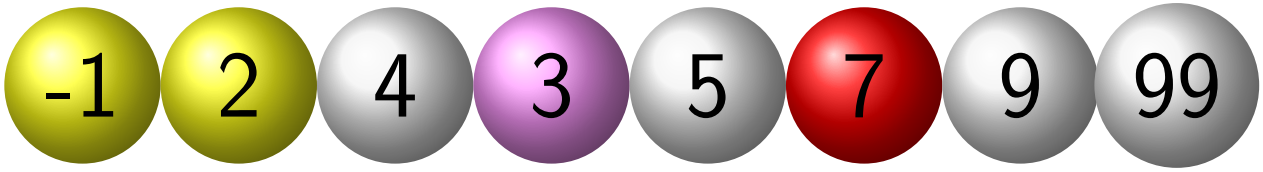


L is sorted, length $==$ 0, sort R

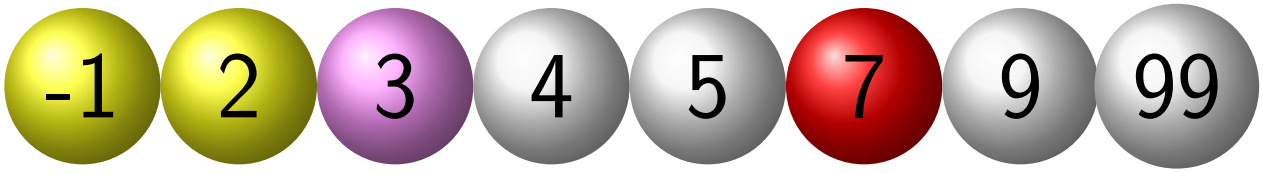
Extended example continued 2



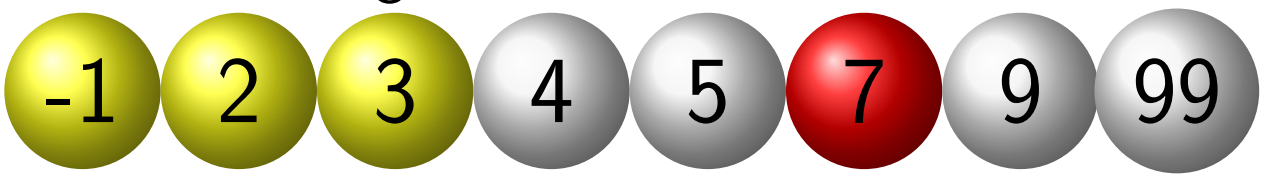
Sorting L, Choose pivot, use middle



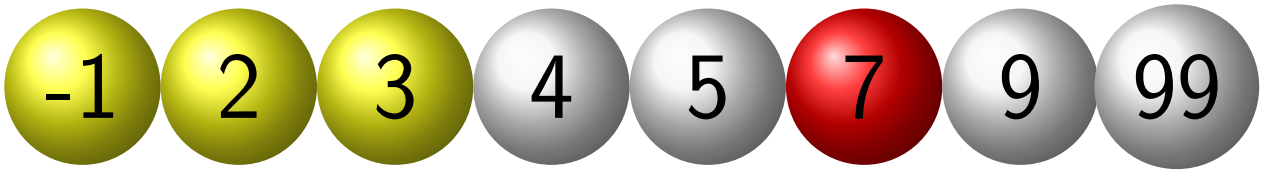
Place elements $<$ pivot on L, elements $>$ pivot on R



L is sorted, length $==$ 0, sort R

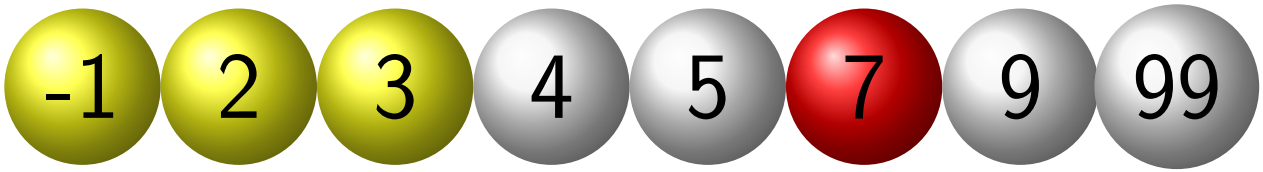


Extended example continued 3

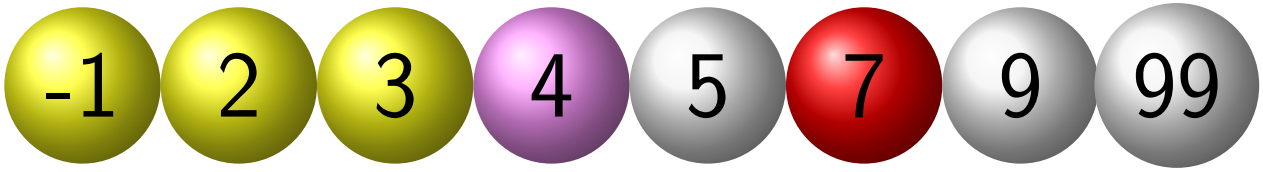


Choose pivot, use middle

Extended example continued 3

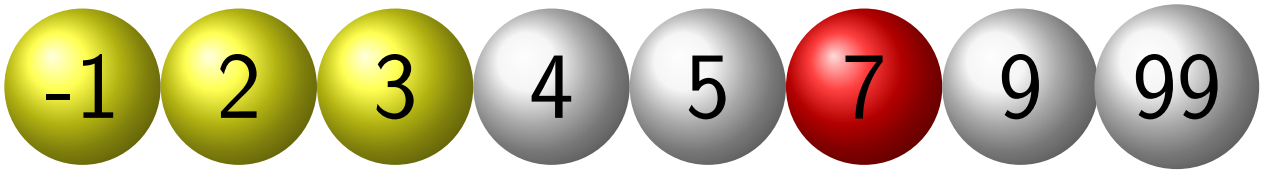


Choose pivot, use middle

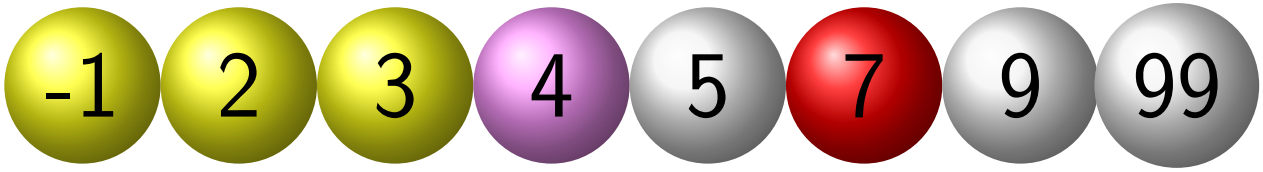


L is sorted, length $==$ 0

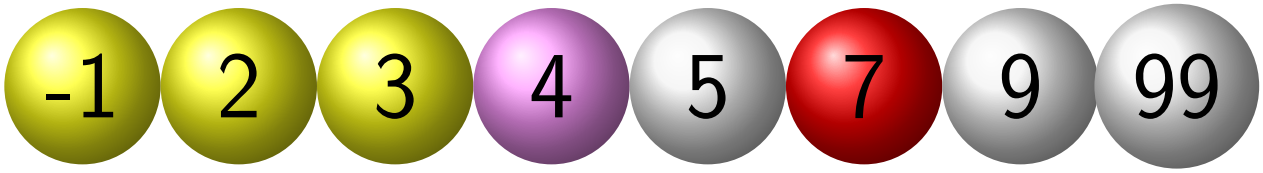
Extended example continued 3



Choose pivot, use middle

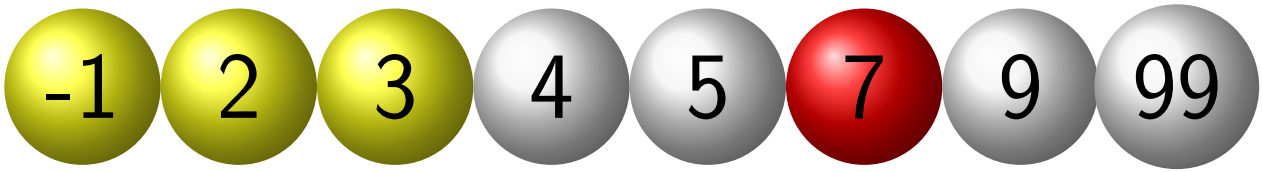


L is sorted, length $== 0$

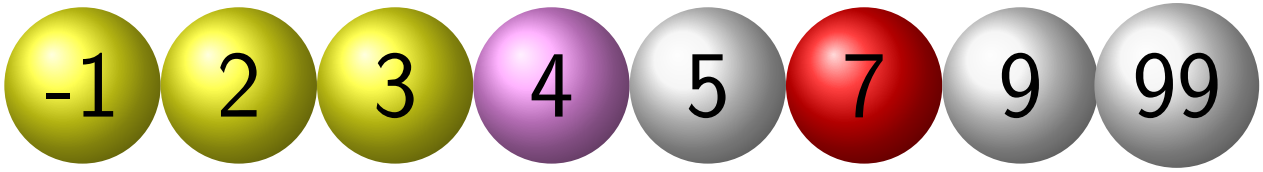


R is sorted, length $== 1$

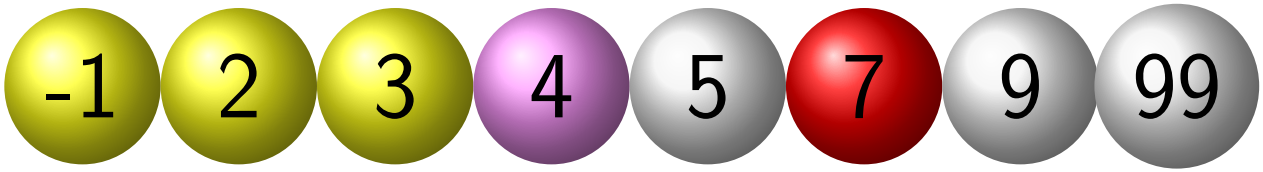
Extended example continued 3



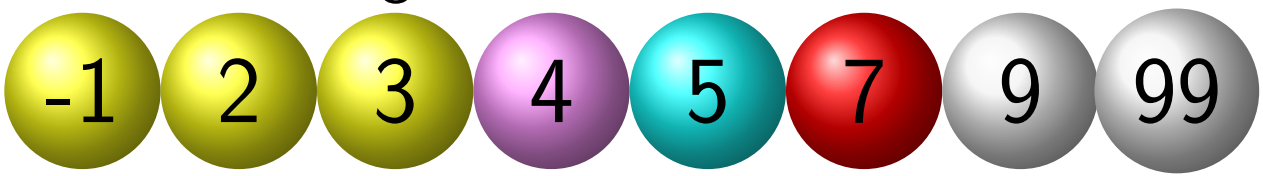
Choose pivot, use middle



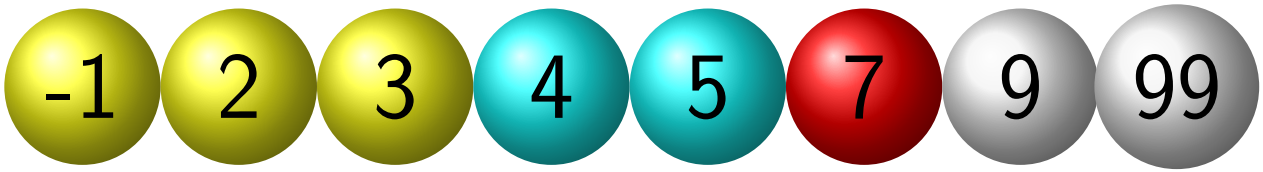
L is sorted, length $==$ 0



R is sorted, length $==$ 1

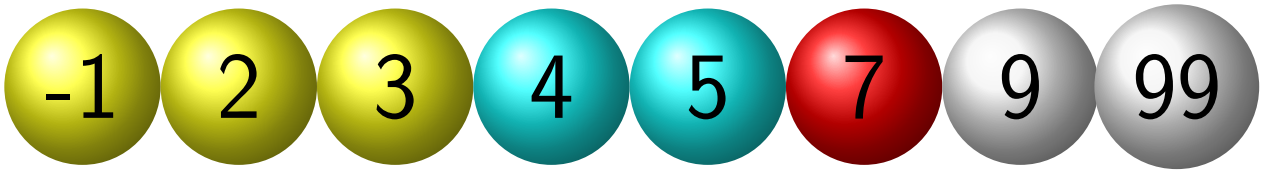


Extended example continued 4

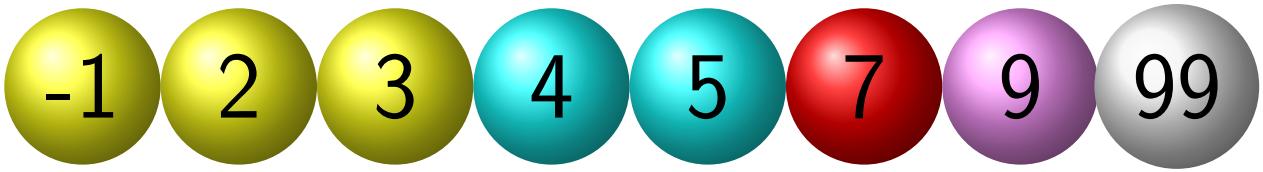


Choose pivot, use middle

Extended example continued 4

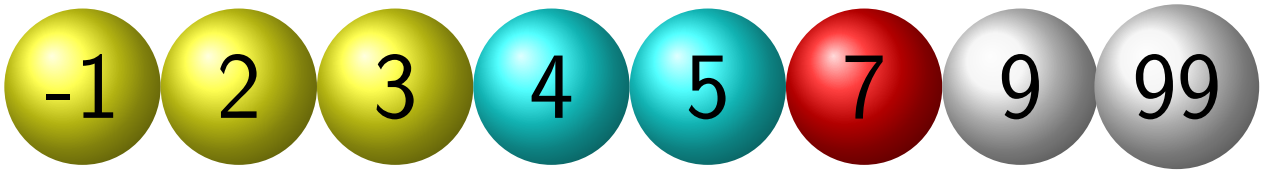


Choose pivot, use middle

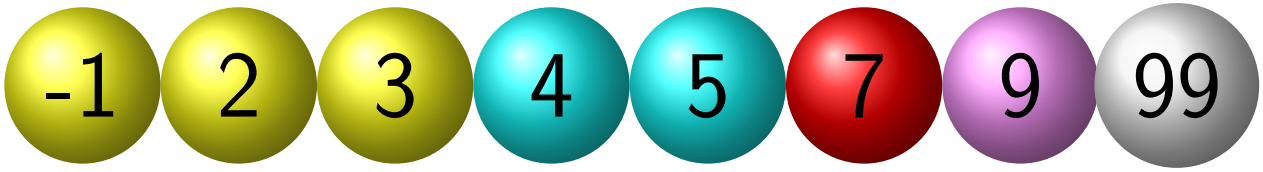


L is sorted, length == 0

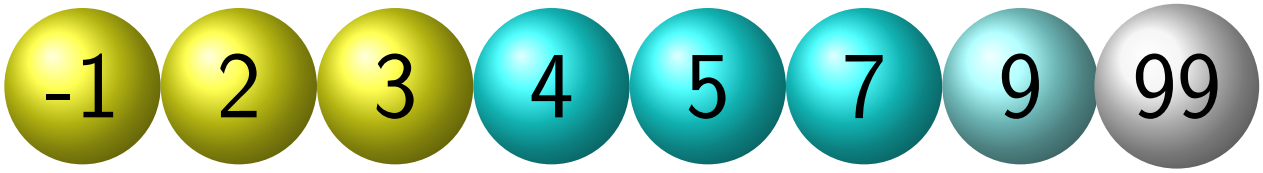
Extended example continued 4



Choose pivot, use middle

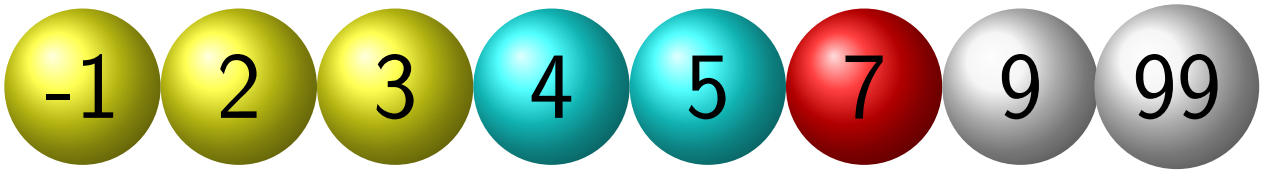


L is sorted, length $==$ 0

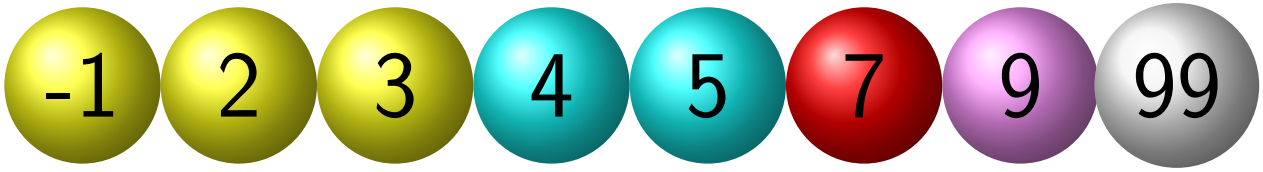


R is sorted, length $==$ 1

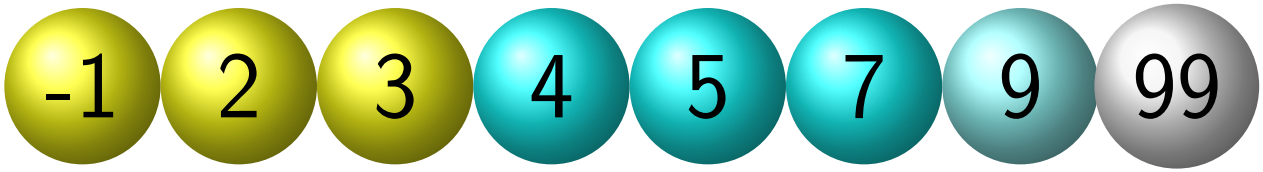
Extended example continued 4



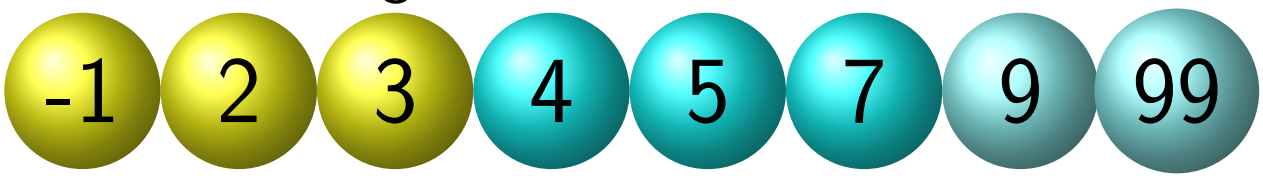
Choose pivot, use middle



L is sorted, length $==$ 0



R is sorted, length $==$ 1



Demonstration

Python code

```
def quick_sort(arr):
    less = []
    equal = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[(len(arr)-1) // 2]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                equal.append(i)
        less = quick_sort(less)
        more = quick_sort(more)
        return less + equal + more
```

Python code comments, declare variables

```
def quick_sort(arr):
    less = []
    equal = []
    more = []
```

Declare a function named *quick_sort* that takes one parameter, a list, which presumably is unsorted.
Declare a *less* list to hold the lesser-than items, a *more* list to hold the greater-than items, and an *equal* lost to hold the equal-to items.

Python code

```
if len(arr) <= 1:
    return arr
else:
```

If the length of the parameter list is 1 or 0, you are done. Stop the recursion.

Else, continue the recursion.

Python code

```
pivot = arr[(len(arr)-1) // 2] #select the middle element
for i in arr:
    if i < pivot:
        less.append(i)
    elif i > pivot:
        more.append(i)
    else:
        equal.append(i)
```

Select a pivot. It could be the first element (*arr[0]*), the last element (*arr[-1]*), the middle element (*arr[len(arr)-1 // 2]*), or a random element.

Iterate through the parameter list, placing all lesser-than elements on the left, and all greater than elements on the right.

Python code

```
#recursive calls for the right and left lists
less = quick_sort(less)
more = quick_sort(more)

#return the left, middle, and right lists glued together
return less + equal + more
```

Call the function recursively on the left and right sides, passing each as a parameter. Return the three lists glued together.

QuickSort output 1

```
initial list is [4, 3, 7, 2, 9, 5, 99, -1]
calling quick_sort([4, 3, 7, 2, 9, 5, 99, -1])
    else branch, list is [4, 3, 7, 2, 9, 5, 99, -1]
       for: i = 4 and pivot = 2, i > pivot, more = [4]
       for: i = 3 and pivot = 2, i > pivot, more = [4, 3]
       for: i = 7 and pivot = 2, i > pivot, more = [4, 3, 7]
       for: i = 2 and pivot = 2, i == pivot, equal = [2]
       for: i = 9 and pivot = 2, i > pivot, more = [4, 3, 7, 9]
       for: i = 5 and pivot = 2, i > pivot, more = [4, 3, 7, 9, 5]
       for: i = 99 and pivot = 2, i > pivot, more = [4, 3, 7, 9, 5, 99]
       for: i = -1 and pivot = 2, i < pivot, less = [-1]
```

QuickSort output 1

```
calling quick_sort([-1])
    DONE if list length <= 1, returning [-1]
calling quick_sort([4, 3, 7, 9, 5, 99])
    else branch, list is [4, 3, 7, 9, 5, 99]
       for: i = 4 and pivot = 7, i < pivot, less = [4]
       for: i = 3 and pivot = 7, i < pivot, less = [4, 3]
       for: i = 7 and pivot = 7, i == pivot, equal = [7]
       for: i = 9 and pivot = 7, i > pivot, more = [9]
       for: i = 5 and pivot = 7, i < pivot, less = [4, 3, 5]
       for: i = 99 and pivot = 7, i > pivot, more = [9, 99]
```

QuickSort output 1

```
calling quick_sort([4, 3, 5])
    else branch, list is [4, 3, 5]
       for: i = 4 and pivot = 3, i > pivot, more = [4]
       for: i = 3 and pivot = 3, i == pivot, equal = [3]
       for: i = 5 and pivot = 3, i > pivot, more = [4, 5]
calling quick_sort([])
    DONE if list length <= 1, returning []
calling quick_sort([4, 5])
    else branch, list is [4, 5]
       for: i = 4 and pivot = 4, i == pivot, equal = [4]
       for: i = 5 and pivot = 4, i > pivot, more = [5]
calling quick_sort([])
    DONE if list length <= 1, returning []
```

QuickSort output 1

```
calling quick_sort([5])
    DONE if list length <= 1, returning [5]
<<<returning [] + [4] + [5]
<<<returning [] + [3] + [4, 5]
calling quick_sort([9, 99])
    else branch, list is [9, 99]
       for: i = 9 and pivot = 9, i == pivot, equal = [9]
       for: i = 99 and pivot = 9, i > pivot, more = [99]
calling quick_sort([])
    DONE if list length <= 1, returning []
calling quick_sort([99])
    DONE if list length <= 1, returning [99]
<<<returning [] + [9] + [99]
<<<returning [3, 4, 5] + [7] + [9, 99]
<<<returning [-1] + [2] + [3, 4, 5, 7, 9, 99]
end list is [-1, 2, 3, 4, 5, 7, 9, 99]
```

Conclusion and Questions

Time complexity of QuickSort

Best case

$$O = n \log_2 n \tag{1}$$

Worst case

$$O = n^2 \tag{2}$$

Questions