Sorting Algorithms

CPE202

Objectives

- Learn to sort a list of values efficiently.
- Be able to analyze several different sorting algorithms in terms of time and space complexity.
- Learn properties of sorting algorithms.
- Be able to select appropriate sorting algorithms depending on the nature of problems.
- Learn to corroborate with other programmers.

Properties of sorting algorithms

- Time Complexity
- Space Complexity
- Stability
  - Stable sort preserves the original order of duplicates

Sorting Problem

30	14	23	8	94	50	32	40	1	3

1	3	8	14	23	30	32	40	50	94

Approach 1: Compare neighbors and switch positions

30	14	23	8	94	50	32	40	1	3

14	30	23	8	94	50	32	40	1	3

Approach 1: Compare neighbors and switch positions

14	30	23	8	94	50	32	40	1	3

14	23	30	8	94	50	32	40	1	3

Approach 1: Compare neighbors and switch positions

14	23	30	8	94	50	32	40	1	3

14	23	8	30	94	50	32	40	1	3

We can keep doing this until the list is sorted. This algorithm is called bubble sort because larger values bubble up toward the end of th list but it takes N*N steps or O(N**2).

Select max value and move it to the end

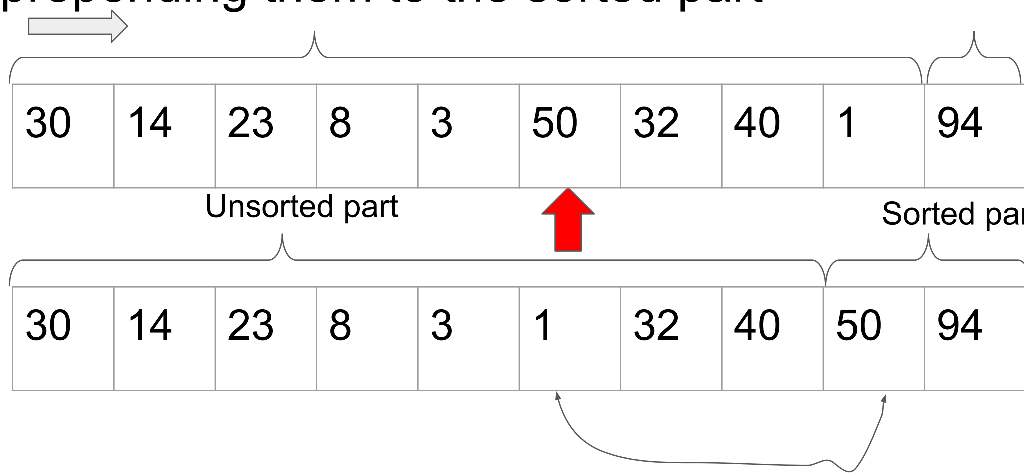
30	14	23	8	94	50	32	40	1	3

Unsorted part

Sorted part

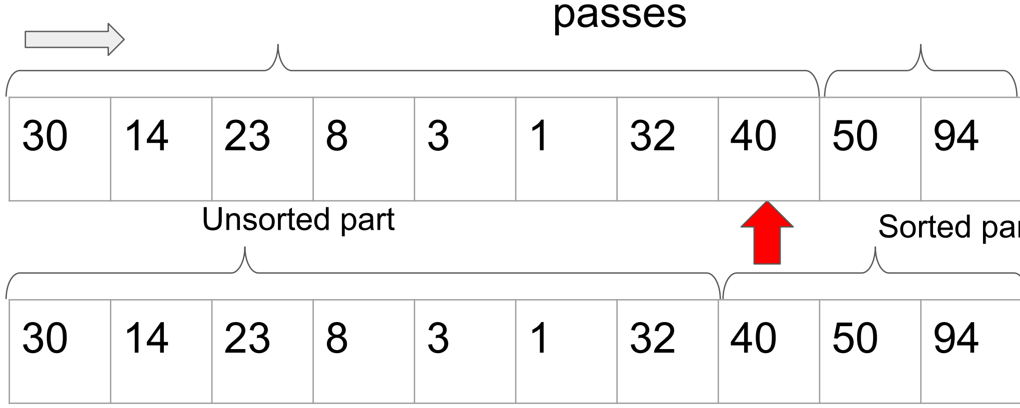
30	14	23	8	3	50	32	40	1	94

Keep selecting max values from the unsorted part and prepending them to the sorted part



Selection Sort

Repeat the process N - 1 passes



The time complexity of selection sort is also $O(N^2)$.
The space complexity is also $O(1)$.

```
def selection_sort(arr):
    size = len(arr)
    for i in reversed(range(1, size)):
        max_idx = 0
        for j in range(1, i):
            if arr[j] > arr[max_idx]:
                max_idx = j
        arr[max_idx], arr[i] = arr[i], arr[max_idx]
    return arr
```

swap

For each item, traversing from left to right

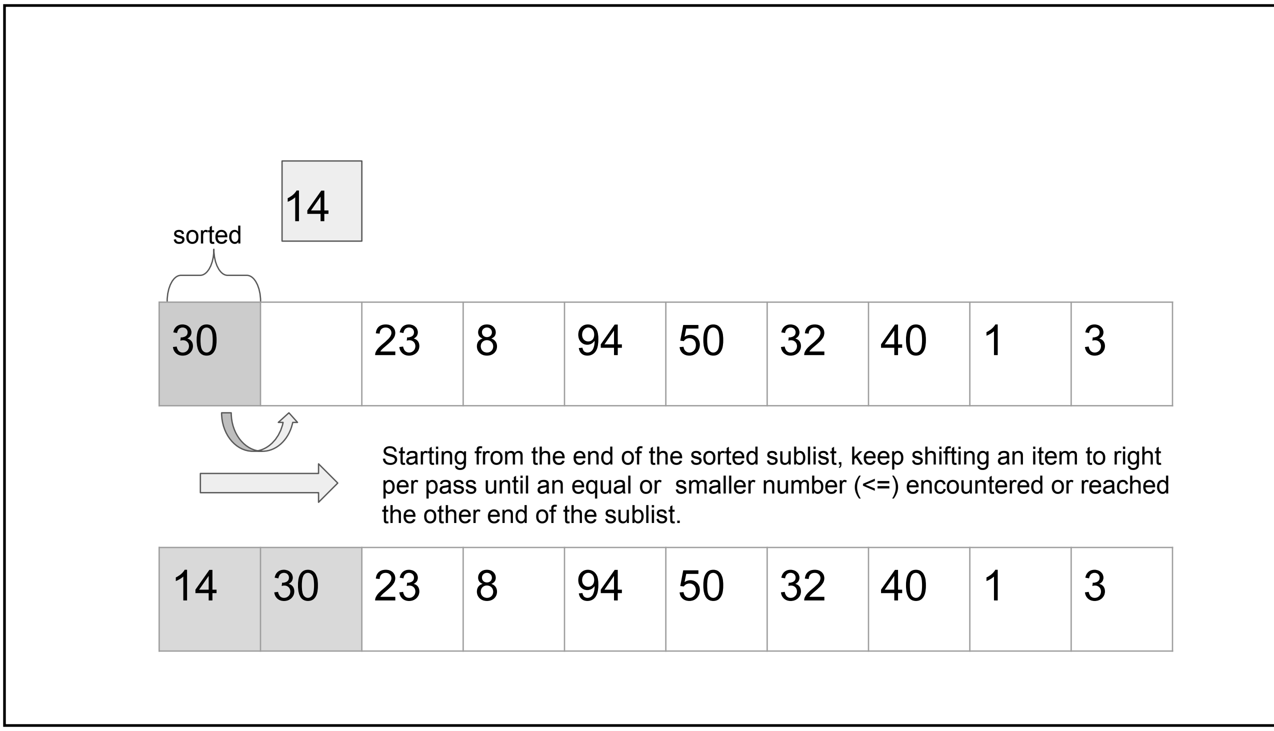
30	14	23	8	94	50	32	40	1	3

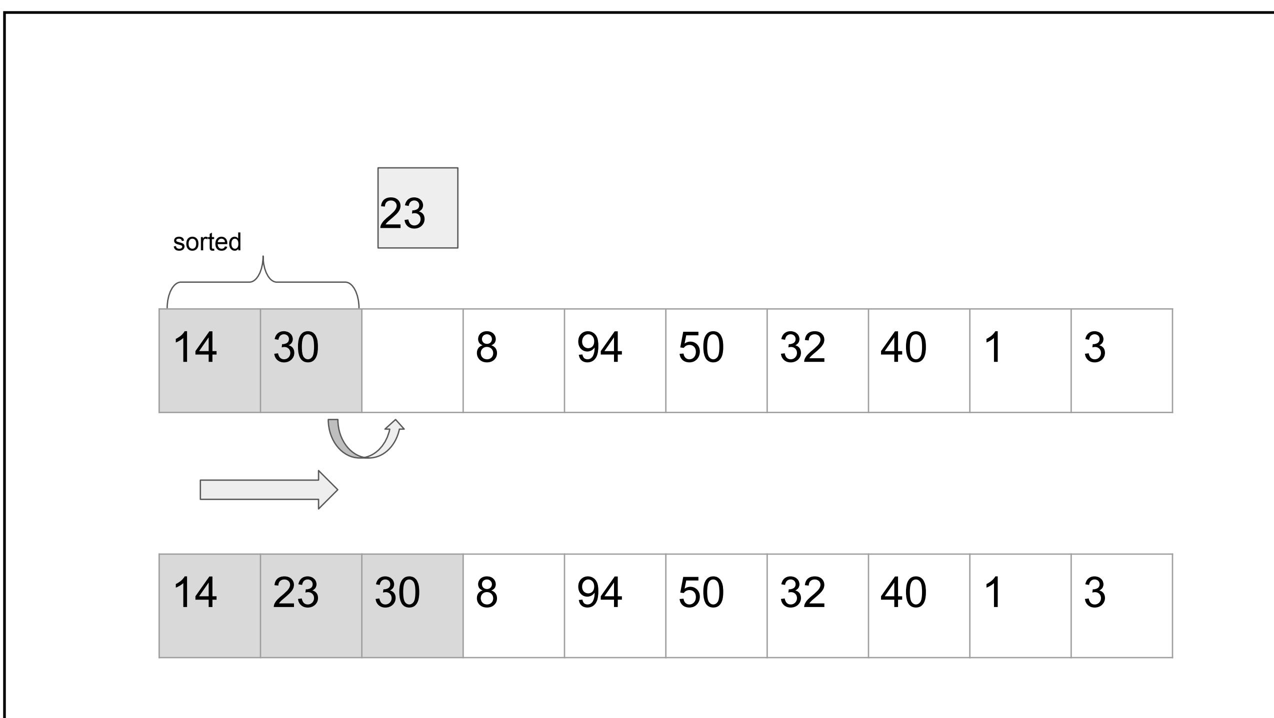
Insert it at an appropriate position in the sorted sublist.

sorted

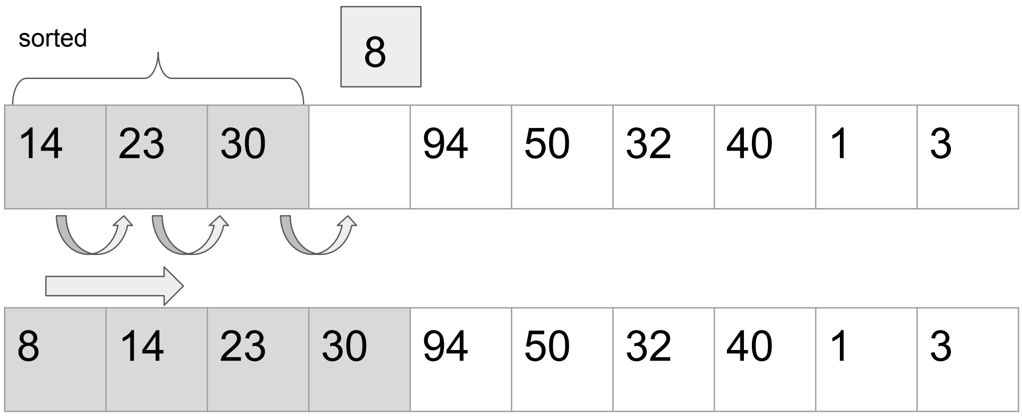
30	14	23	8	94	50	32	40	1	3

We consider that the first item to be already sorted.


14
sorted
30 | | 23 | 8 | 94 | 50 | 32 | 40 | 1 | 3
Starting from the end of the sorted sublist, keep shifting an item to right per pass until an equal or smaller number (<=) encountered or reached the other end of the sublist.
14 | 30 | 23 | 8 | 94 | 50 | 32 | 40 | 1 | 3


23
sorted
14 | 30 | | 8 | 94 | 50 | 32 | 40 | 1 | 3
14 | 23 | 30 | 8 | 94 | 50 | 32 | 40 | 1 | 3

Insertion Sort



The time complexity of the insertion sort is still $O(N^2)$. However, to add an item to a sorted list, the insertion sort performs in $O(N)$. In the best case, it takes only one comparison to sort an already sorted list. The space complexity is $O(1)$. Also, it is **stable**, meaning that original order can be preserved among items of an equal value.

```
def insertion_sort(arr):
    size = len(arr)
    for i in range(1, size):
        j = i
        while j > 0 and arr[j - 1] > arr[j]:
            #shift
            arr[j - 1], arr[j] = arr[j], arr[j - 1]
            j -= 1
    return arr
```

Divide and Conquer Approach

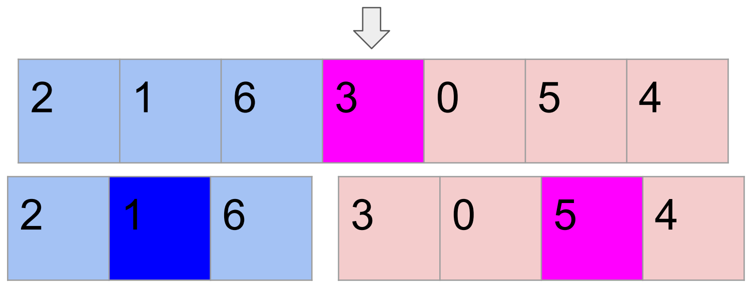
Merge Sort

Compute the midpoint and divide the list into 2 sublists

2	1	6	3	0	5	4

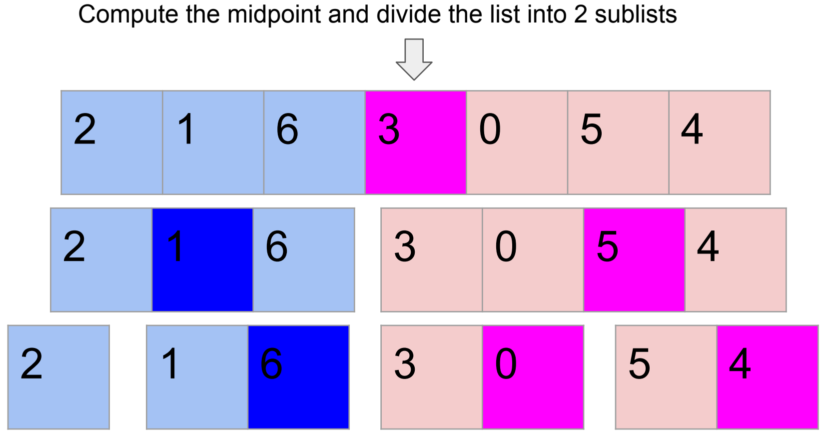
Merge Sort

Compute the midpoint and divide the list into 2 sublists



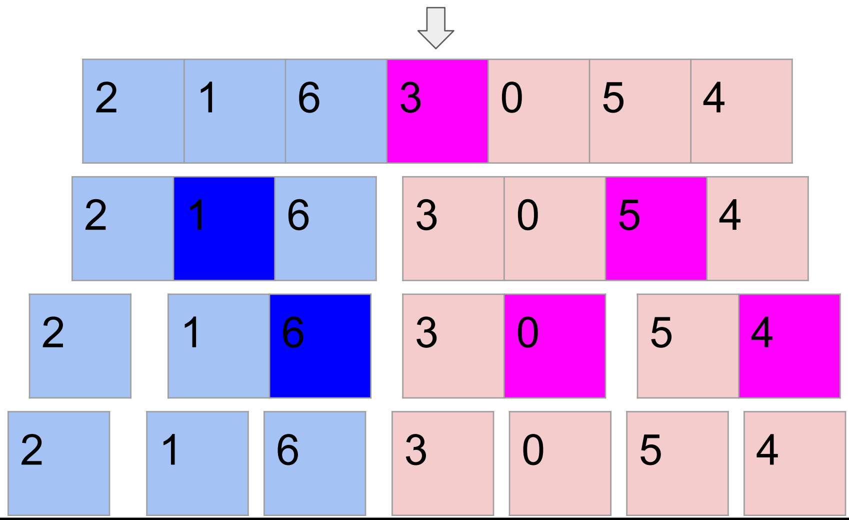
Merge Sort

Compute the midpoint and divide the list into 2 sublists



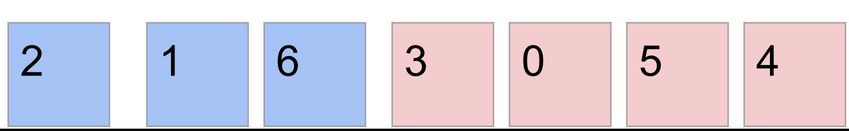
Merge Sort

Compute the midpoint and divide the list into 2 sublists



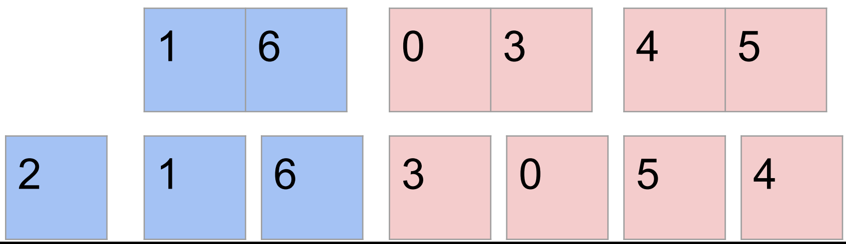
Merge Sort

items are sorted as being merged



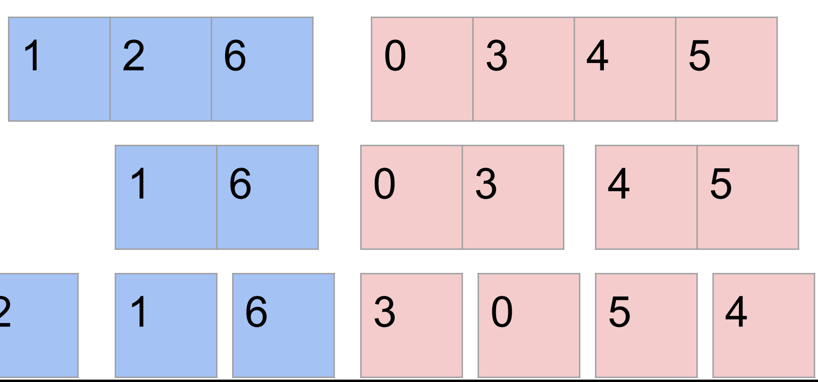
Merge Sort

items are sorted as being merged



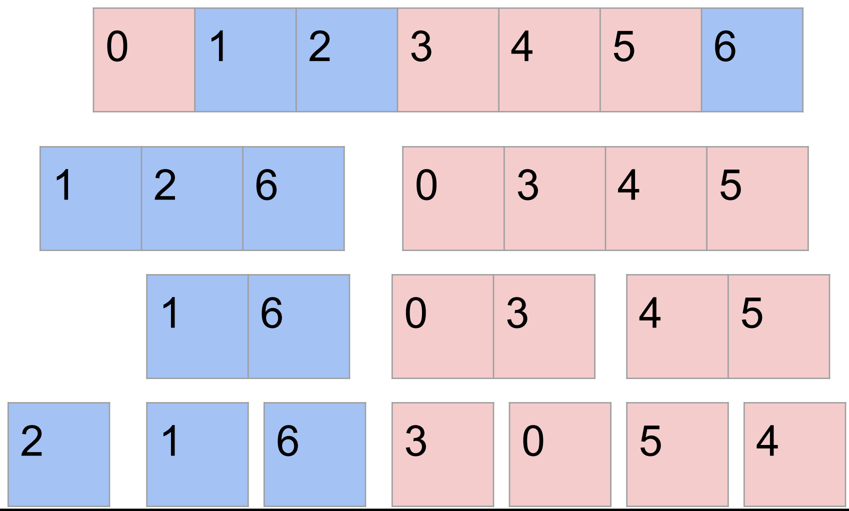
Merge Sort

items are sorted as being merged



Merge Sort

items are sorted as being merged




```
def merge_sort(items):
    """sort a list of items in ascending order using the merge sort algorithm.
    Note:
        This version of the merge sort produce many copies of the list and
        therefore not very efficient in terms of the space complexity.
    Args:
        items (list) : a list of integers or strings
    Returns:
        list : a copy of the original list with items sorted in acending order.
    """
    if the length of the items list <= 1, return the items list
    compute the midpoint
    left = merge_sort(items[:midpoint])
    right = merge_sort(items[midpoint:])
    return merge(left, right)
```

```
def merge(left, right):
    """merge two list into one as sorting items in ascending order.
    Args:
        left (list) : the left part of a list to be merged
        right (list) : the right part of a list to be merged
    Returns:
        list : a merged and sorted list
    """
    merged = []
    left_idx = right_idx = 0
    while left_idx < left.length and right_idx < right.length:
        if left[left_idx] <= right[right_idx]:
            merged.append(left[left_idx++])
        else:
            merged.append(right[right_idx++])
    if there are leftover in the left list:
        append all the leftover to the merged
    if there are leftover in the right list:
        append all the leftover to the merged
    return merged
```

Merge Sort

- Time Complexity
  - N comparisons in log N passes ~ O(N*logN)
  - O(NlogN) in the worst, best, and average cases.
- Space Complexity
  - O(N) because it makes a copy of the list.
- Stable
- Useful when you have data which is too big to fit in memory.

Quick Sort

Partition the list into 2 parts at a pivot value

1. lower half <= pivot
2. upper half >= pivot

<= pivot	pivot	pivot <=

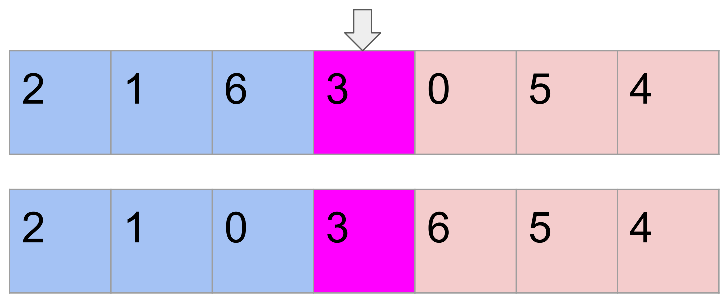
Quick Sort

Compute the midpoint and divide the list into 2 sublists

2	1	6	3	0	5	4

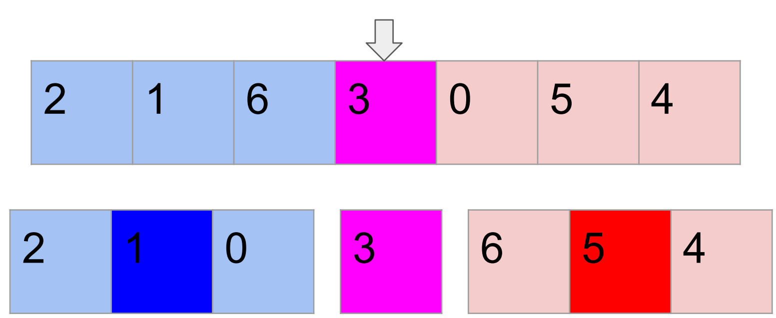
Quick Sort

Compute the midpoint and divide the list into 2 sublists



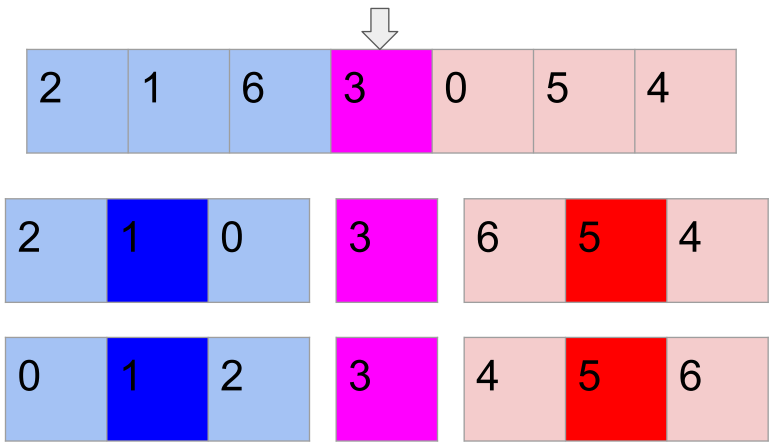
Quick Sort

Compute the midpoint and divide the list into 2 sublists

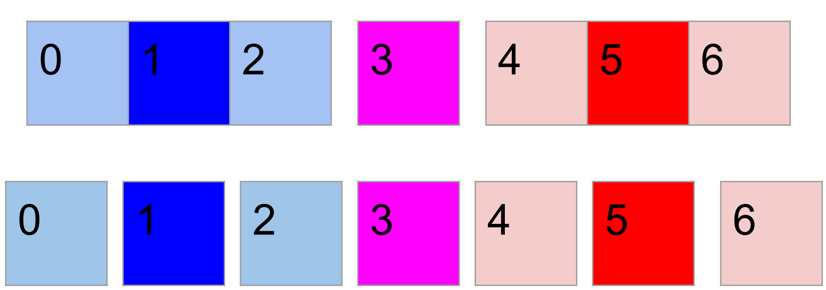


Quick Sort

Compute the midpoint and divide the list into 2 sublists



Quick Sort



Quick Sort

2	1	6	3	0	5	4

3	1	6	2	0	5	4

Pick the pivot.
Generally picking the median item is a good idea because you want the pivot value to be close to the mean value so that you can divide the list into 2 equal size sublists.

Move the pivot to the beginning of the list (swap).

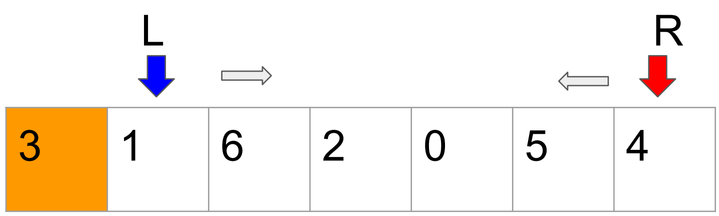
Quick Sort

1. Create a left pointer and point it to the second item from the left.
2. Create a right pointer and point it to the last item on the right.

	L					R
3	1	6	2	0	5	4

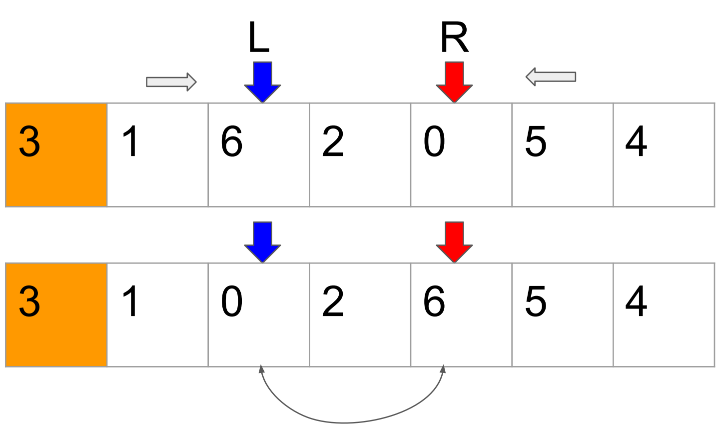
Quick Sort

1. Keep moving the left pointer to right until it points to an item whose value is equal to or larger than the pivot value.
2. Keep moving the right pointer to left until it points to an item whose value is equal to or smaller than the pivot value.



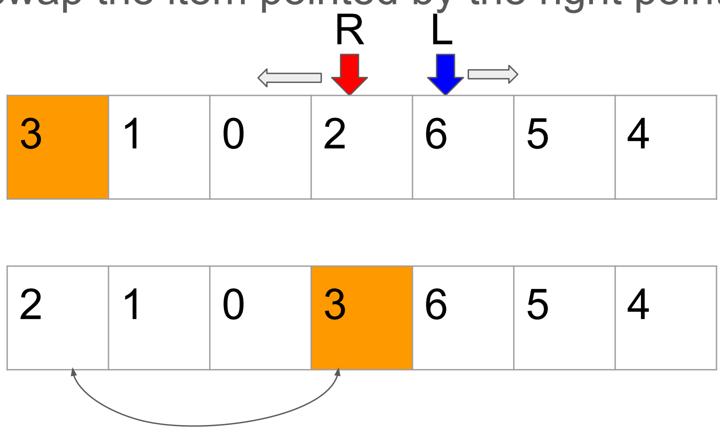
Quick Sort

3. Swap the item pointed by the left pointer with the item pointed by the right pointer.



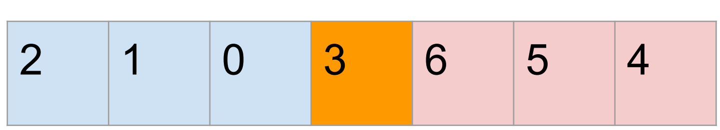
Quick Sort

4. Keep moving pointers and swapping items until the two pointers cross.
5. If they cross, swap the item pointed by the right pointer with the pivot.



Quick Sort

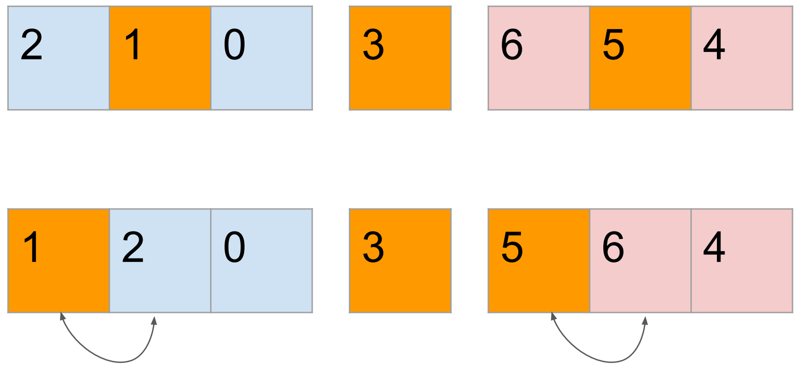
6. Divide the list into two sublists at where the right pointer was pointing.
7. Repeat the process within the left sublist and within the right sublist, util the size of the sublist becomes less than 2.



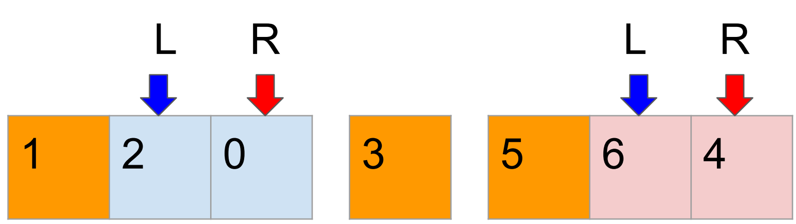
Quick Sort

Quick Sort

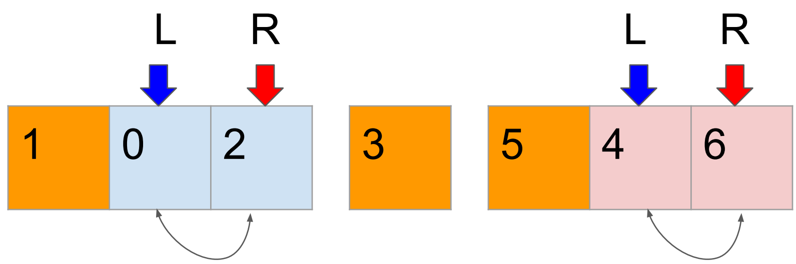
Quick Sort



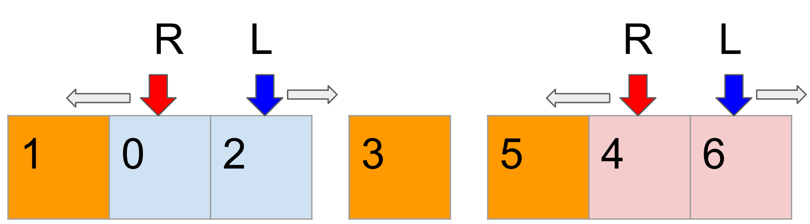
Quick Sort



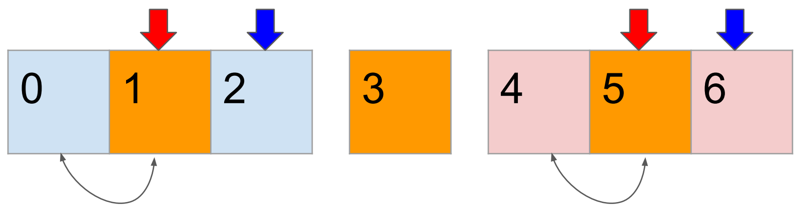
Quick Sort



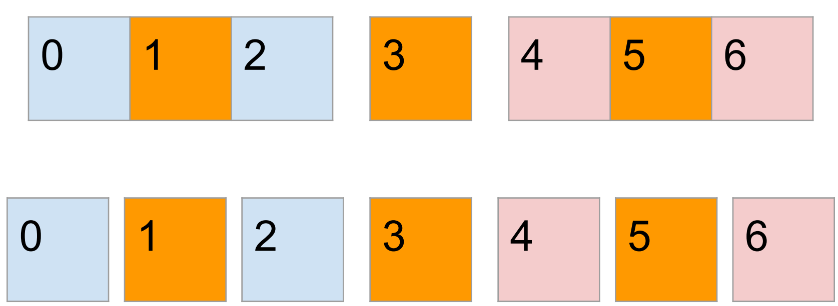
Quick Sort



Quick Sort



Quick Sort



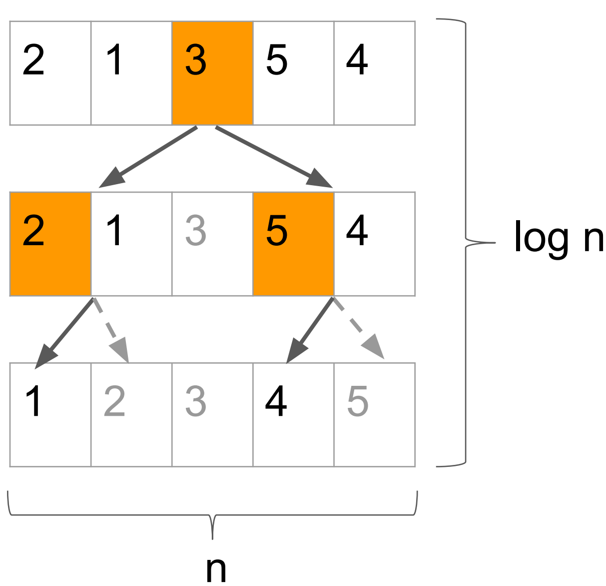
Pseudo code for a in-memory version

```
def quick_sort(items, lo, hi):
    if lo >= hi, return
    mid = (lo + hi) // 2 #compute the index of the pivot
    move the pivot to the head of the list for easiness
    set left pointer at the second item
    set the right index at the end
    while the 2 pointers do not pass each other:
        increment the left pointer while while left <= hi and items[left] <= pivot
        decrement the right pointer while right > lo and pivot <= items[right]
        swap values if the pointers have not passed each other.
    swap back the pivot with the item pointed by the right pointer
    quick_sort(items, lo, right - 1)
    quick_sort(items, right + 1, hi)
    return
```

Quicksort
Best & Avg Case

Time:
O(n * log n)

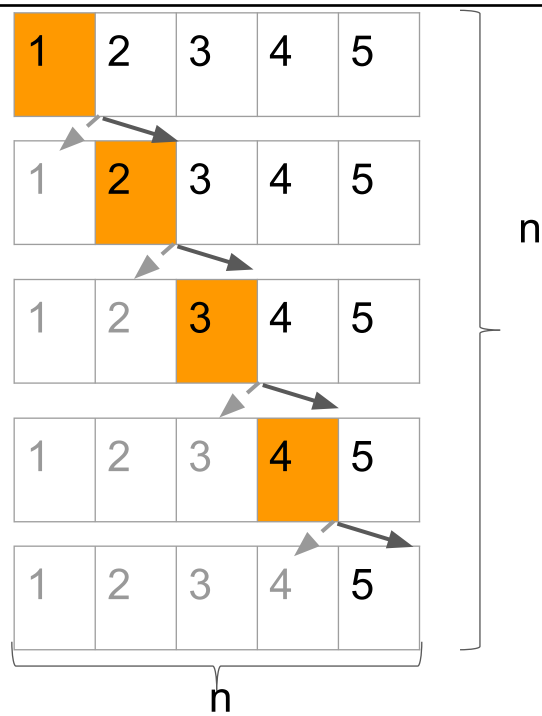
Space:
O(log n)



Quicksort
Worst Case

Time:
$O(n^2)$

Space:
$O(n)$



Quick Sort

- Time Complexity
  - Average and Best case: O(N*logN)
  - Worst case (when the choice of pivot is bad): O(N**2)
- Space Complexity
  - O(log N) because the recursive calls use at most log N call stack spaces.
  - O(N) in the worst case.
- The in-place version is unstable because items’ positions are swapped.

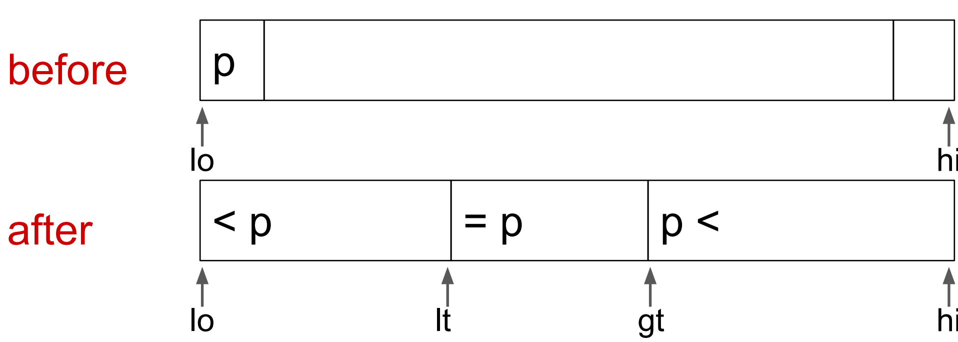
Dijkstra's Three-way Partitioning

Problem: Duplicate keys

Quicksort goes quadratic unless partitioning stops on equal keys!

3-way Partitioning

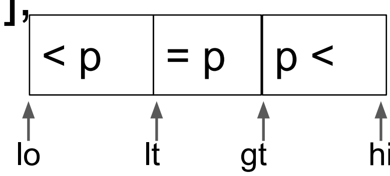
- Entries between lt and gt equal to partitioning item p (pivot).
- No larger entries to left of lt.
- No smaller entries to right of gt.



3-way Partitioning

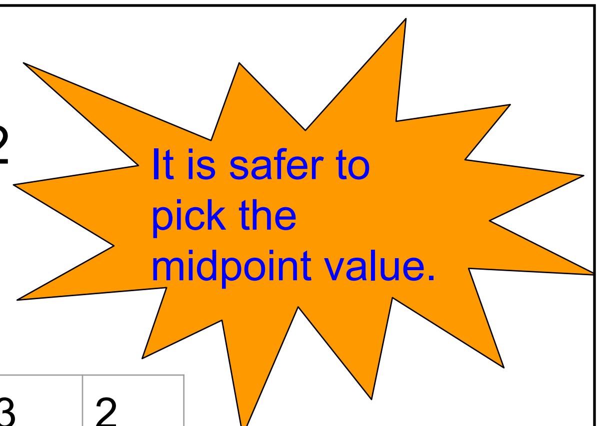
Let p be partitioning item (pivot) a[lo].
Let i and lt be lo, gt be hi
Scan i from left to right while i <= gt.

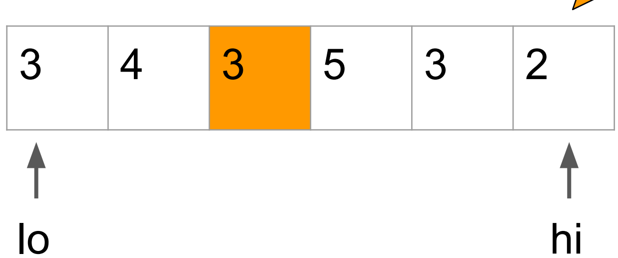
- (a[i] < p): exchange a[lt] with a[i];
  - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
  - decrement gt
- (a[i] == p): increment i

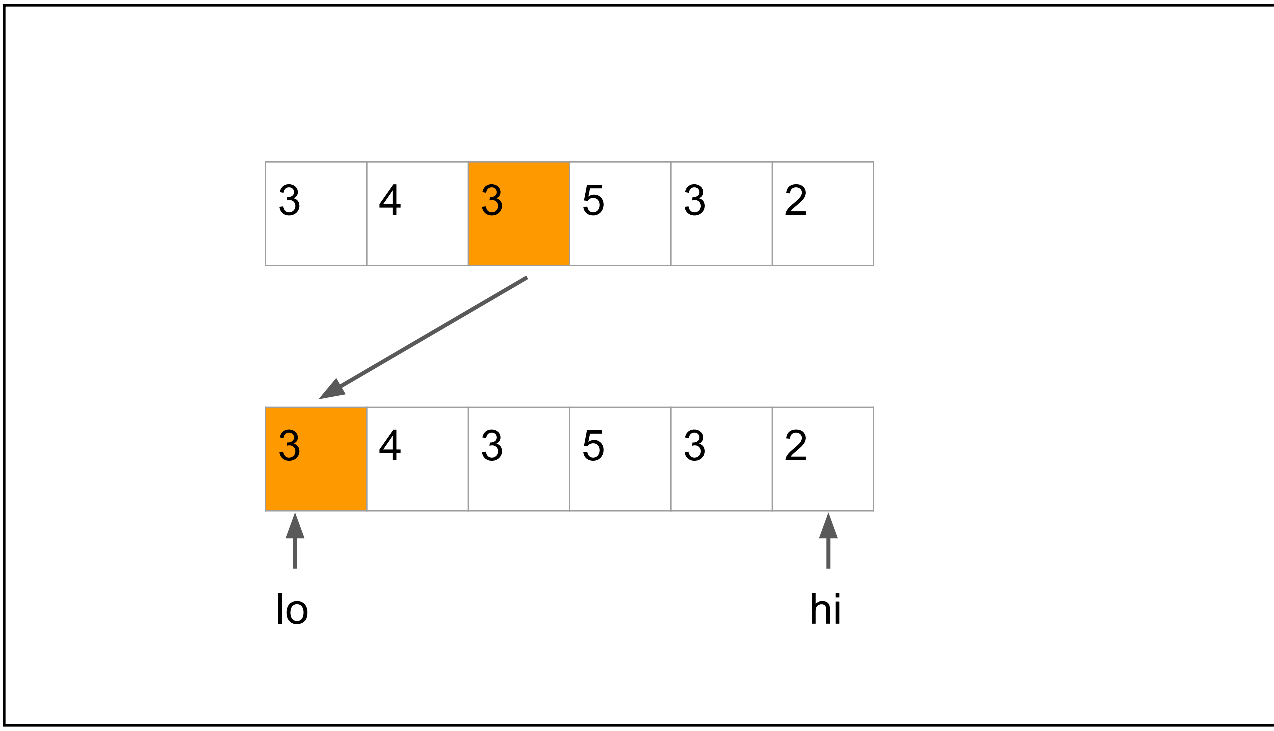


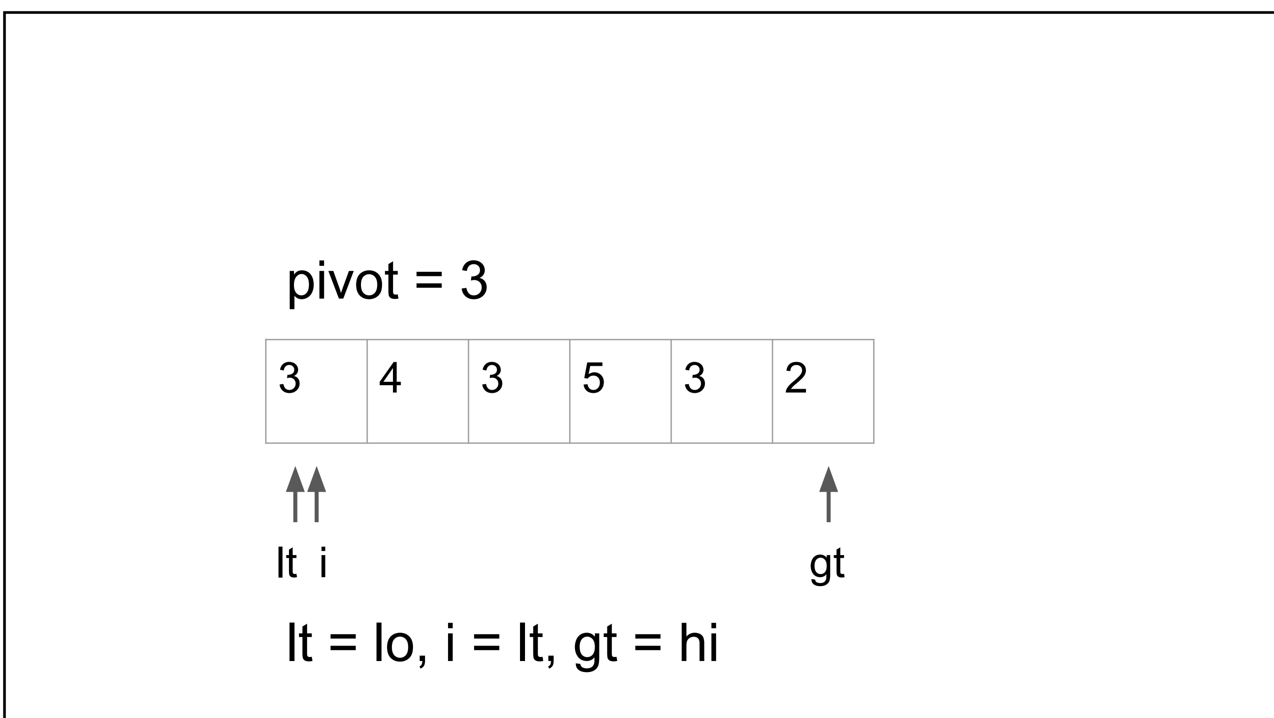
mid = (lo + hi) // 2

pivot = 3



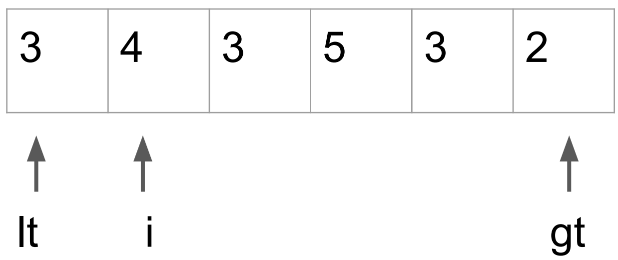



3 4 3 5 3 2
3 4 3 5 3 2
lo
hi


pivot = 3
3 4 3 5 3 2
lt i
gt
lt = lo, i = lt, gt = hi

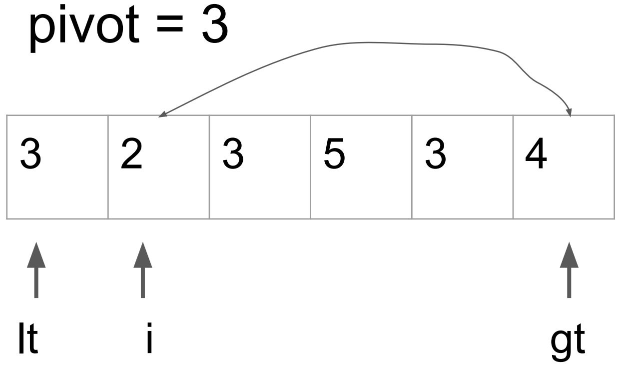
- (a[i] < p): exchange a[lt] with a[i];
  - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
  - decrement gt
- (a[i] == p): increment i

pivot = 3



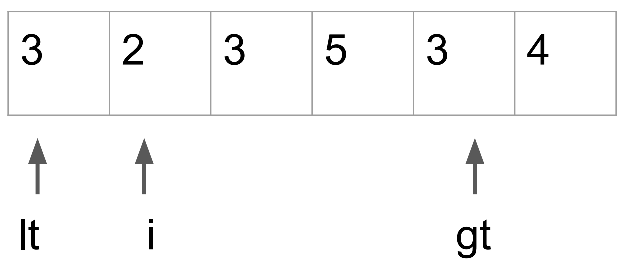
- (a[i] < p): exchange a[lt] with a[i];
  - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
  - decrement gt
- (a[i] == p): increment i

pivot = 3



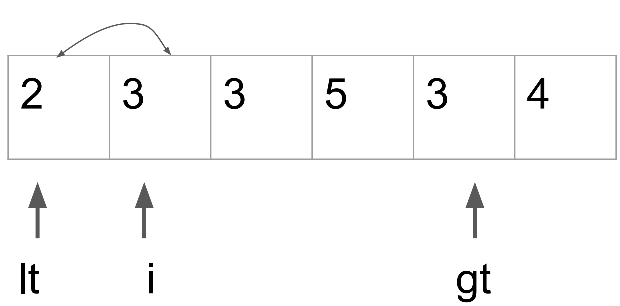
- (a[i] < p): exchange a[lt] with a[i];
  - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
  - decrement gt
- (a[i] == p): increment i

pivot = 3



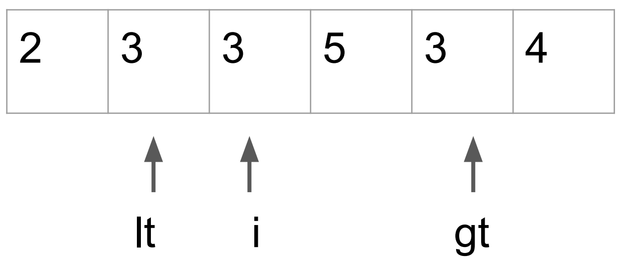
- (a[i] < p): exchange a[lt] with a[i];
  - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
  - decrement gt
- (a[i] == p): increment i

pivot = 3



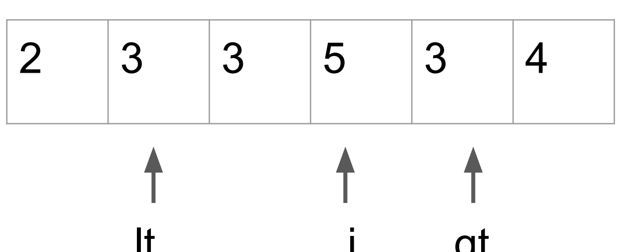
- (a[i] < p): exchange a[lt] with a[i];
    - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
    - decrement gt
- (a[i] == p): increment i

pivot = 3



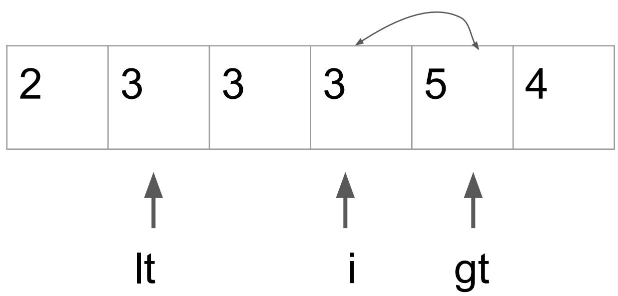
- (a[i] < p): exchange a[lt] with a[i];
    - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
    - decrement gt
- (a[i] == p): increment i

pivot = 3



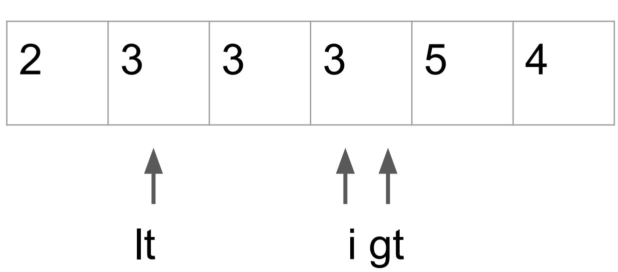
- (a[i] < p): exchange a[lt] with a[i];
  - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
  - decrement gt
- (a[i] == p): increment i

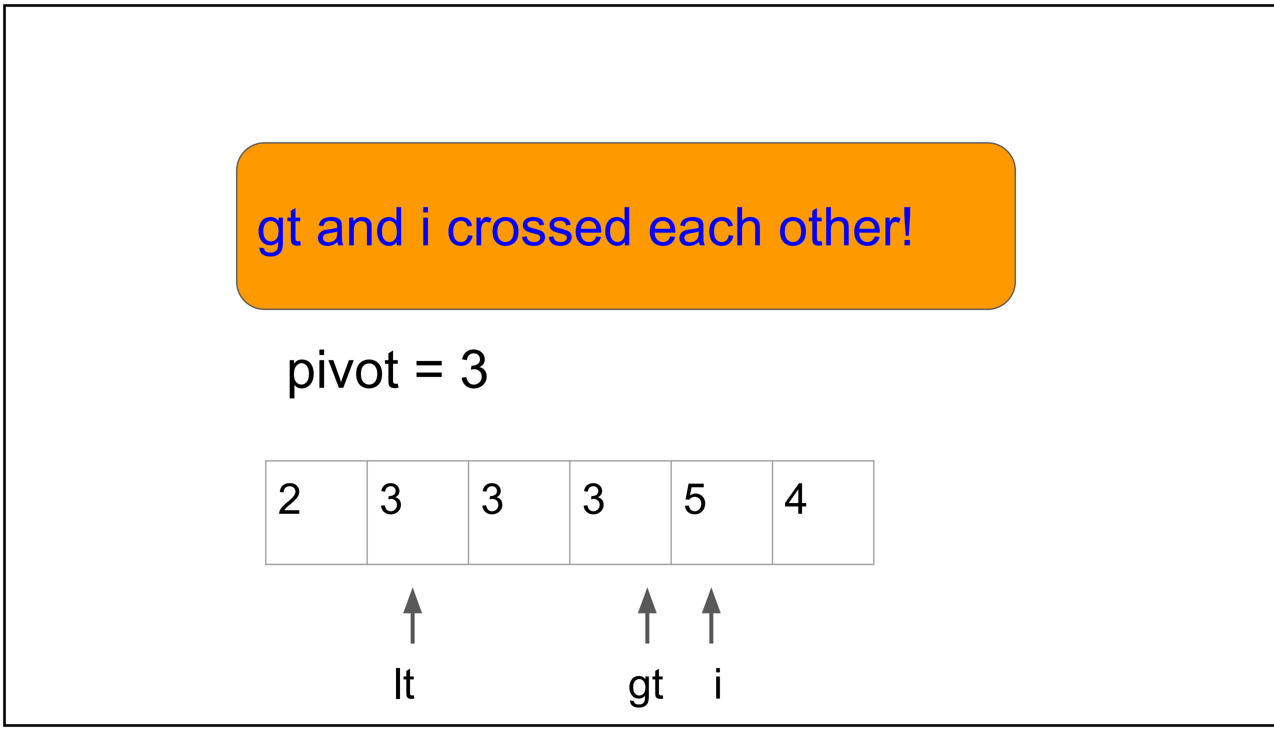
pivot = 3

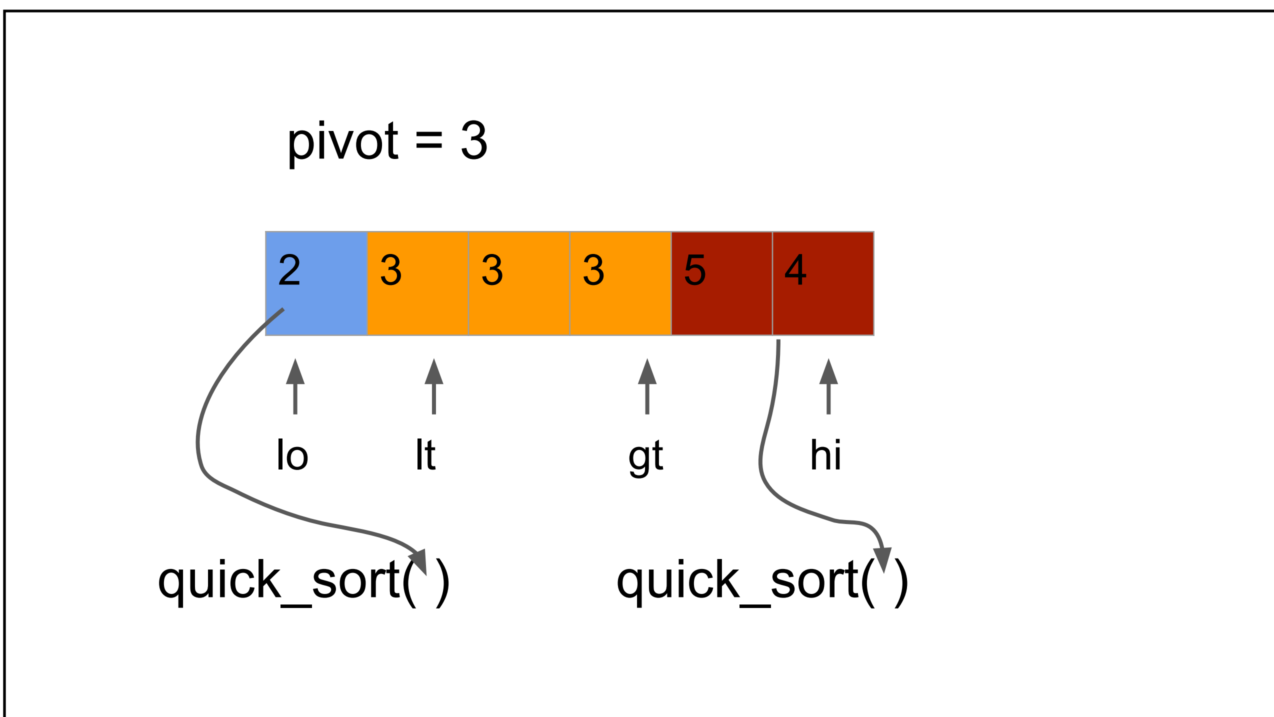


- (a[i] < p): exchange a[lt] with a[i];
  - increment both lt and i
- (a[i] > p): exchange a[gt] with a[i];
  - decrement gt
- (a[i] == p): increment i

pivot = 3




gt and i crossed each other!
pivot = 3
2
3
3
3
5
4
lt
gt
i


pivot = 3
2
3
3
3
5
4
lo
lt
gt
hi
quick_sort( )
quick_sort( )

```
def quick_sort(items, lo, hi):
    if lo >= hi: return
    mid = (lo + hi) // 2
    pivot = items[mid]
    lt, gt, i = lo, hi, lt
    while i <= gt:
        if smaller than the pivot:
            swap items[i] with items[lt]; i += 1, lt += 1
        elif larger than the pivot:
            swap items[i] with items[gt];  gt -= 1
        else:
            i += 1
    quick_sort(items, lo, lt - 1)
    quick_sort(items, gt + 1, hi)
```

Adaptive Approach

Adaptive Sorting Algorithms

- Adaptive sorting algorithms take advantage of existing order within the data.
- In the real world data, you can often find occurrences of sorted sequences.

Timsort

- An adaptive sort algorithm called Timsort, invented by software engineer Tim Peters, takes advantage of natural occurrences of sorted sequence in data by combining the insertion sort and the merge sort.
- It is used as the default sorting algorithm in Python, java, and other programming languages.
- Its time complexity is O(N log N) in average and worst cases, and O(N) in best case scenario.
- It is stable.

Comparison of algorithms

Name	Best(O)	Average(O)	Worst(O)	Space	Stable
Quicksort	N log N	N log N	N^2	**log N**	No
Merge sort	N log N	N log N	N log N	N	Yes
Heapsort	N log N	N log N	N log N	1	No
Insertion sort	N	N^2	N^2	1	Yes
Selection sort	N^2	N^2	N^2	1	No
Timsort	N	N log N	N log N	N	Yes
Shell sort	N log N	-	-	1	No
Bubble sort	N (with presorted list)	N^2	N^2	1	Yes