Python Programming

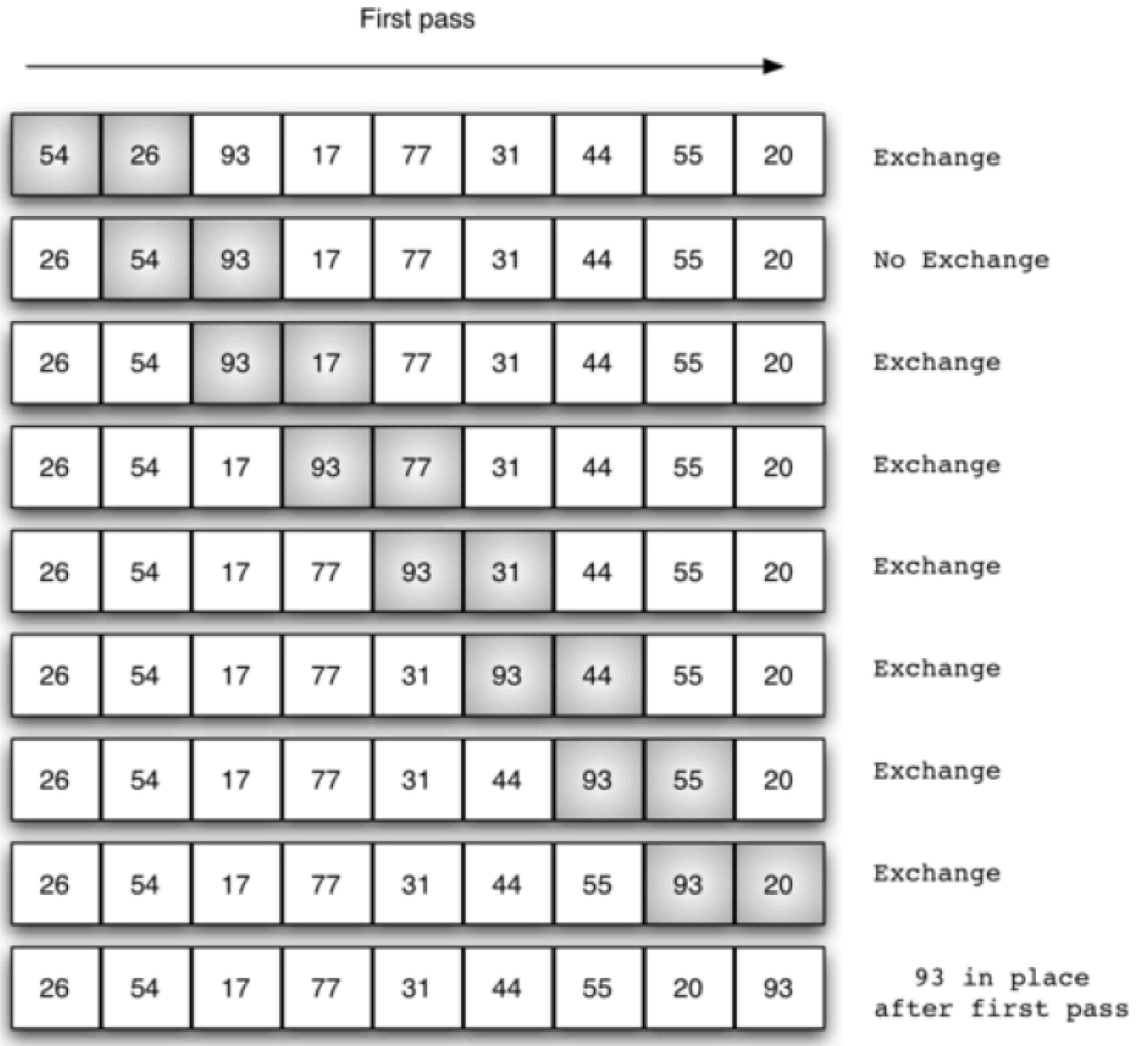
Lecture 15 Sorting Algorithms, Greedy Algorithm

15.1 Sorting Algorithms

- Sorting is the process of placing elements from a collection in some kind of order. There are many, many sorting algorithms that have been developed and analyzed. This suggests that sorting is an important area of study in computer science.
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
    - Merge Sort
    - Quick Sort

1. Bubble sort (冒泡排序)

- The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.
- If there are n items in the list, then there are $n - 1$ pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.
- At the start of the second pass, the largest value is now in place. There are $n - 1$ items left to sort, meaning that there will be $n - 2$ pairs. Since each pass places the next largest value in place, the total number of passes necessary will be $n - 1$. After completing the $n - 1$ passes, the smallest item must be in the correct position with no further processing required.


First pass
54 26 93 17 77 31 44 55 20
Exchange
26 54 93 17 77 31 44 55 20
No Exchange
26 54 93 17 77 31 44 55 20
Exchange
26 54 17 93 77 31 44 55 20
Exchange
26 54 17 77 93 31 44 55 20
Exchange
26 54 17 77 31 93 44 55 20
Exchange
26 54 17 77 31 44 93 55 20
Exchange
26 54 17 77 31 44 55 93 20
Exchange
26 54 17 77 31 44 55 20 93
93 in place after first pass

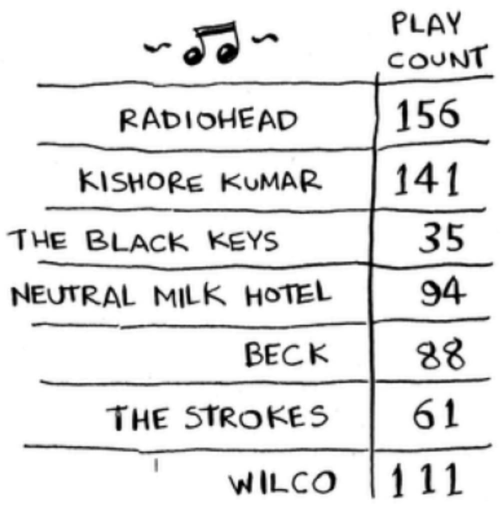
```python
def bubble_sort(a_list):
    for pass_num in range(len(a_list) - 1, 0, -1):
        for i in range(pass_num):
            if a_list[i] > a_list[i+1]:
                a_list[i+1], a_list[i] = a_list[i], a_list[i+1]
```

```python
a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
bubble_sort(a_list)
print(a_list)
```

2. Selection sort (选择排序)

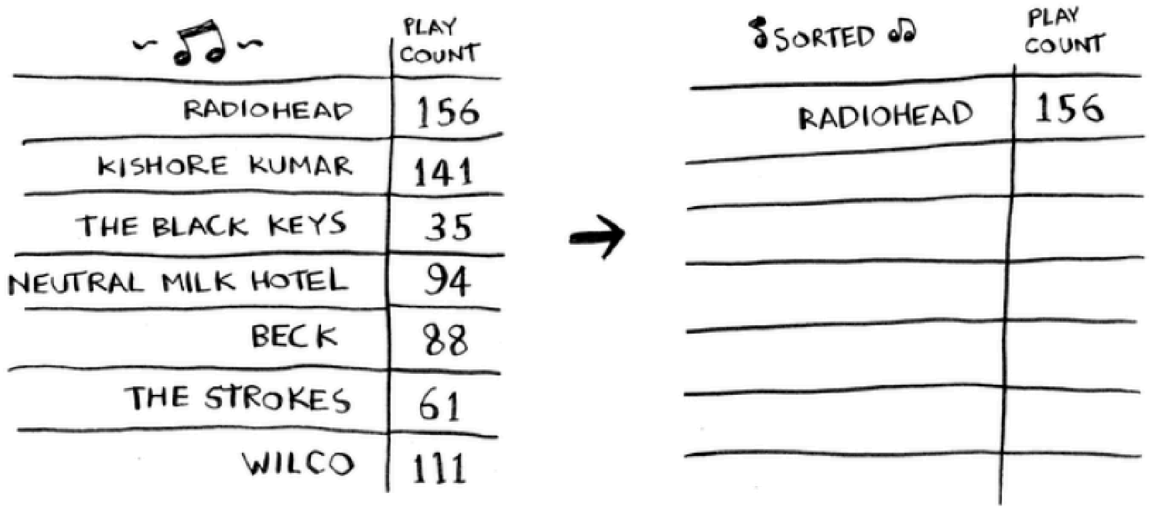
- The selection sort improves on the bubble sort by making only one exchange for every pass through the list.
- As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n – 1 passes to sort n items, since the final item must be in place after the (n – 1)st pass.

- Suppose you have a bunch of music on your computer. For each artist, you have a play count.



♫	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

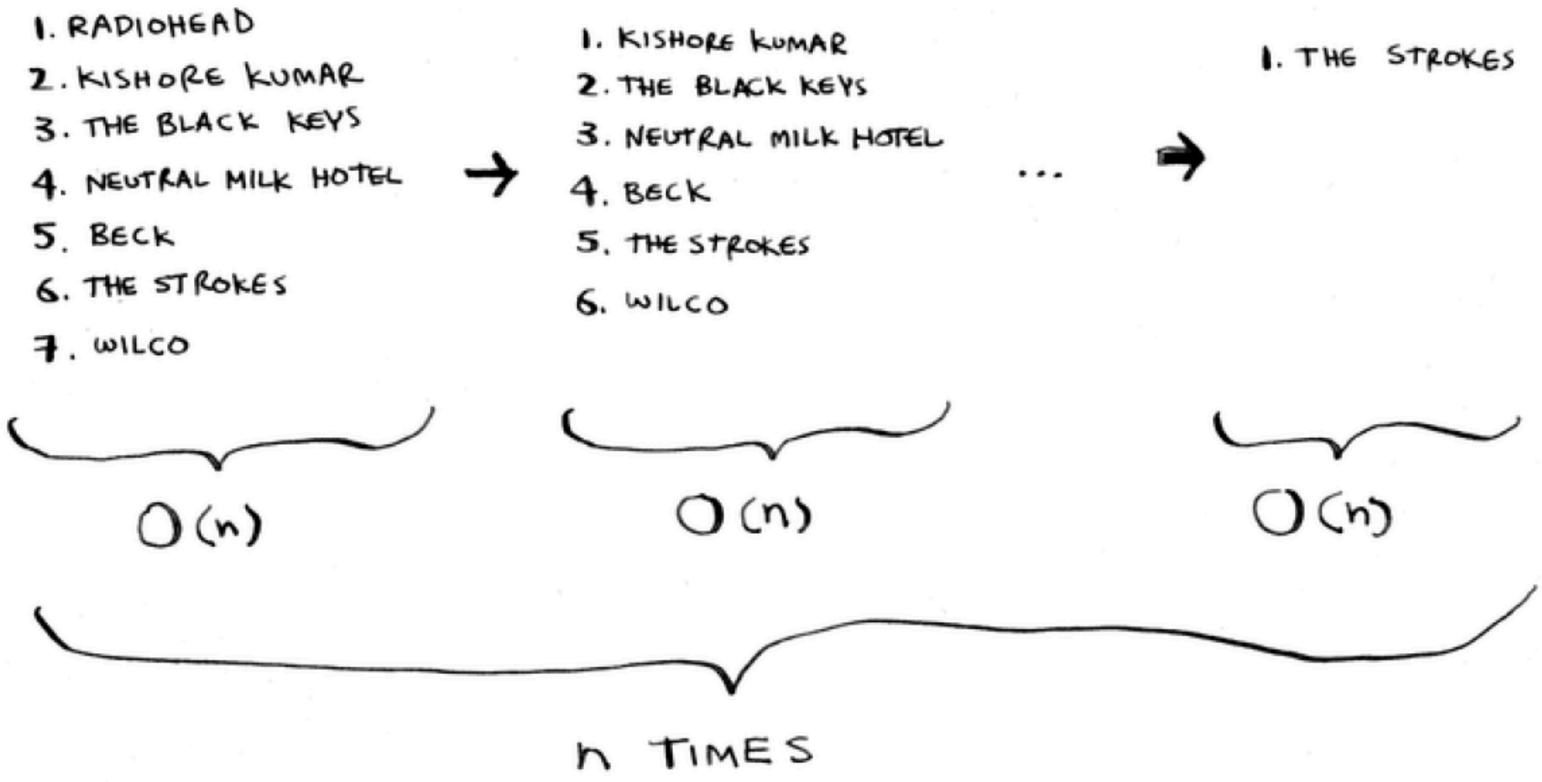
- One way is to go through the list and find the most-played artist. Add that artist to a new list.



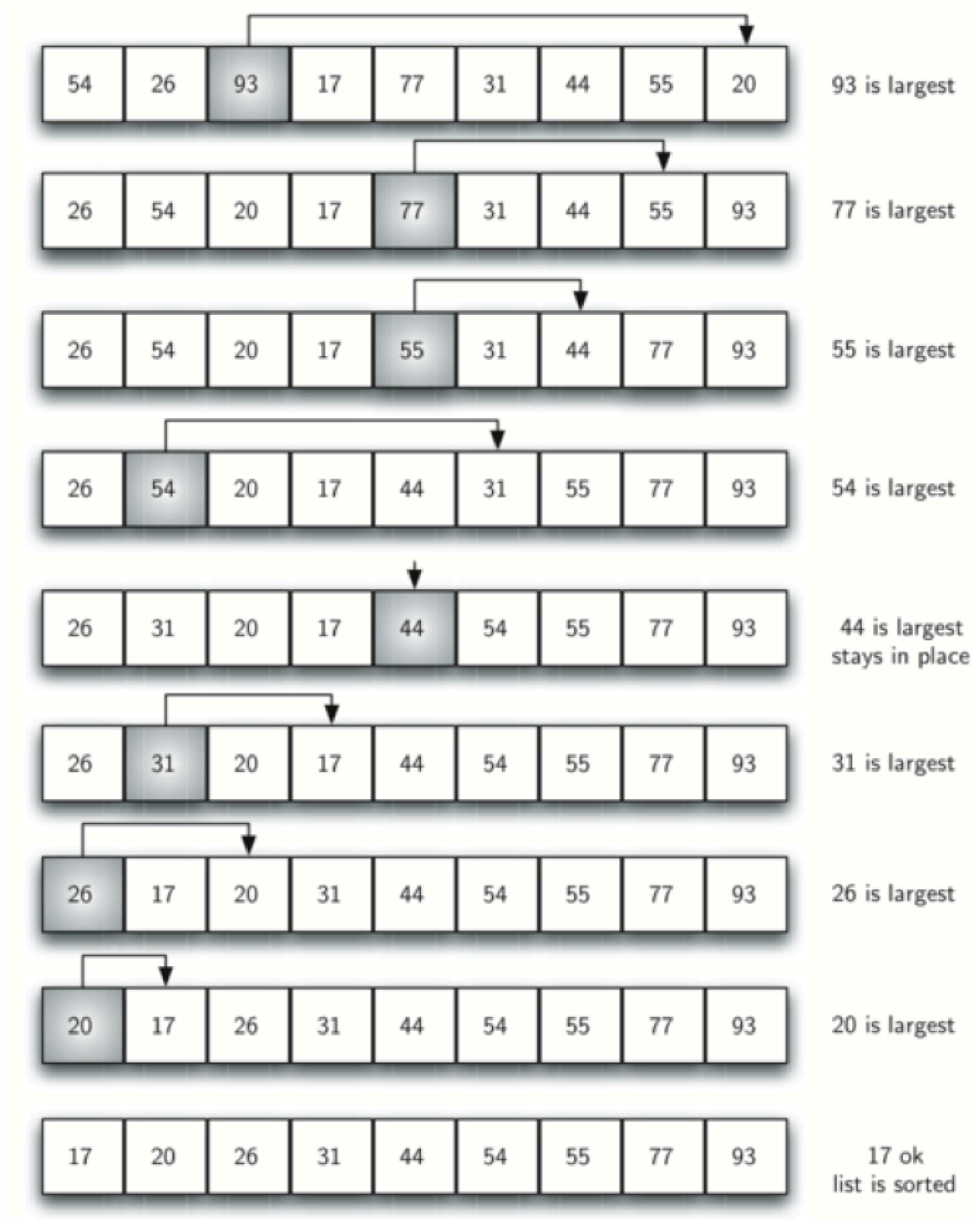
	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

SORTED	PLAY COUNT
RADIOHEAD	156

- To find the artist with the highest play count, you have to check each item in the list. This takes $O(n)$ time, as you just saw. So you have an operation that takes $O(n)$ time, and you have to do that n times:

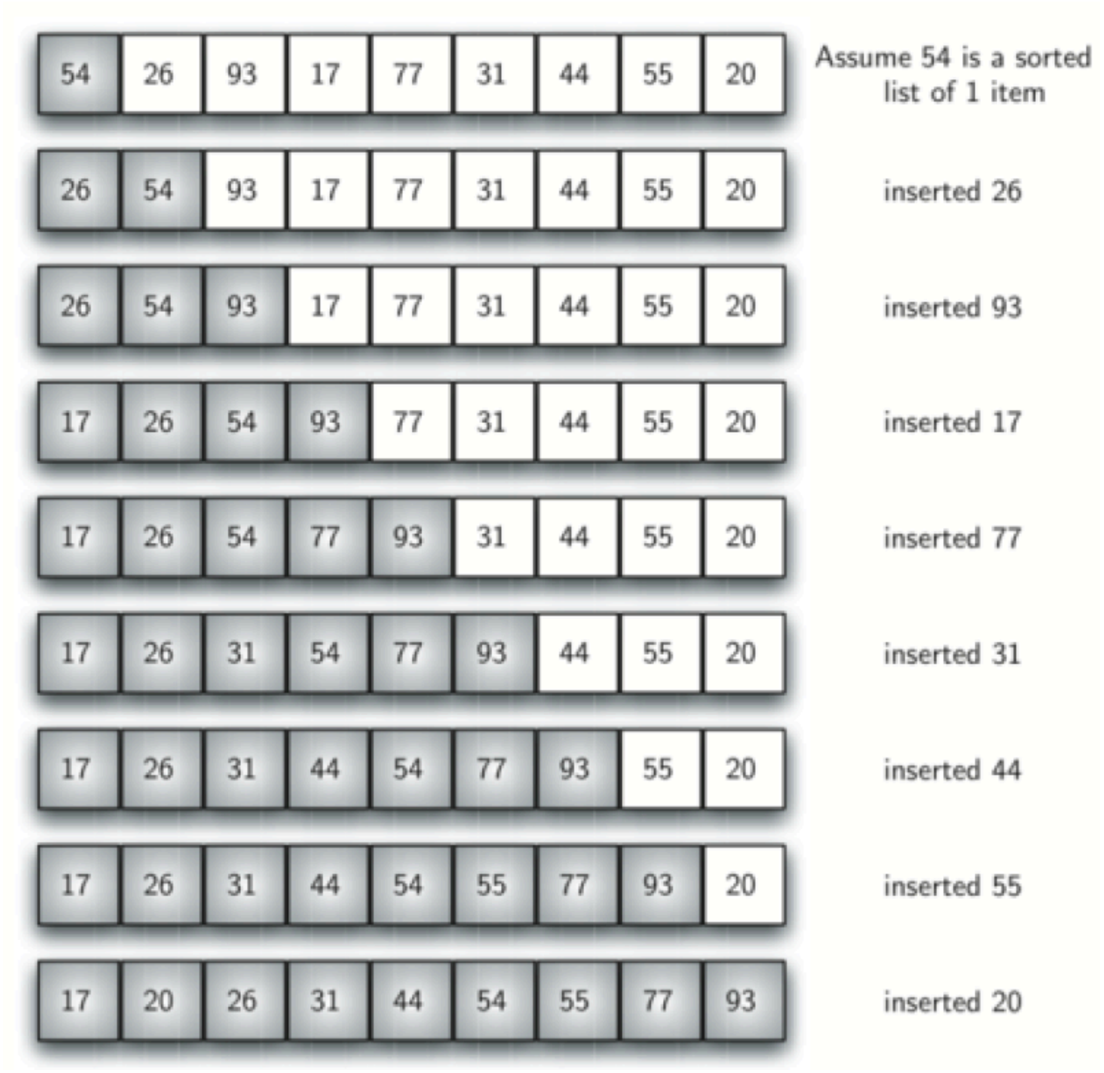


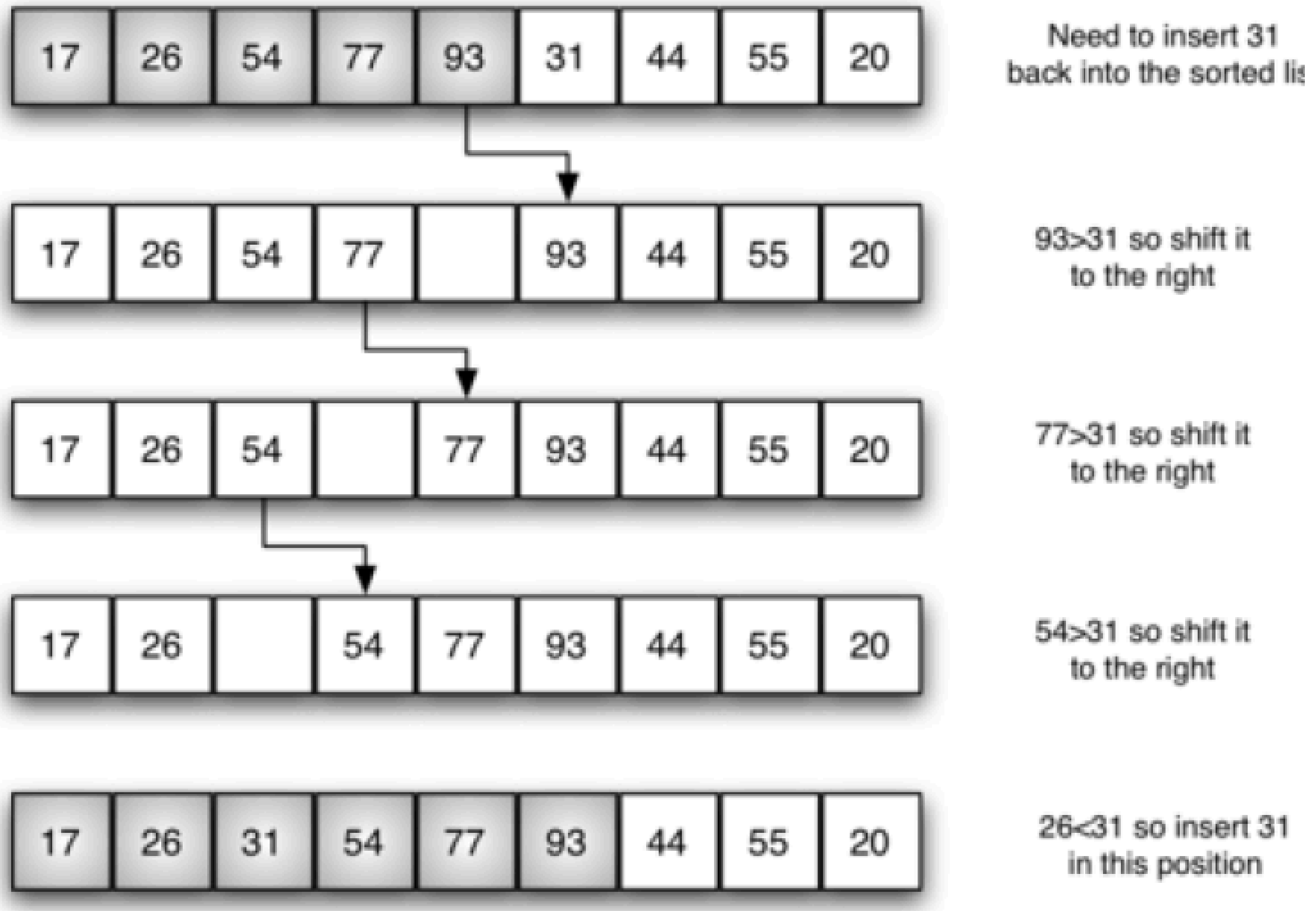
- This takes $O(n \times n)$ time or $O(n^2)$ time.


54 26 93 17 77 31 44 55 20
93 is largest
26 54 20 17 77 31 44 55 93
77 is largest
26 54 20 17 55 31 44 77 93
55 is largest
26 54 20 17 44 31 55 77 93
54 is largest
26 31 20 17 44 54 55 77 93
44 is largest
stays in place
26 31 20 17 44 54 55 77 93
31 is largest
26 17 20 31 44 54 55 77 93
26 is largest
20 17 26 31 44 54 55 77 93
20 is largest
17 20 26 31 44 54 55 77 93
17 ok
list is sorted

```python
def selection_sort(a_list):
    for fill in range(len(a_list) - 1, 0, -1):
        pos_max = 0
        for location in range(1, fill + 1):
            if a_list[location] > a_list[pos_max]:
                pos_max = location
        a_list[pos_max], a_list[fill] = a_list[fill], a_list[pos_max]
```

3. Insertion Sort (插入排序)



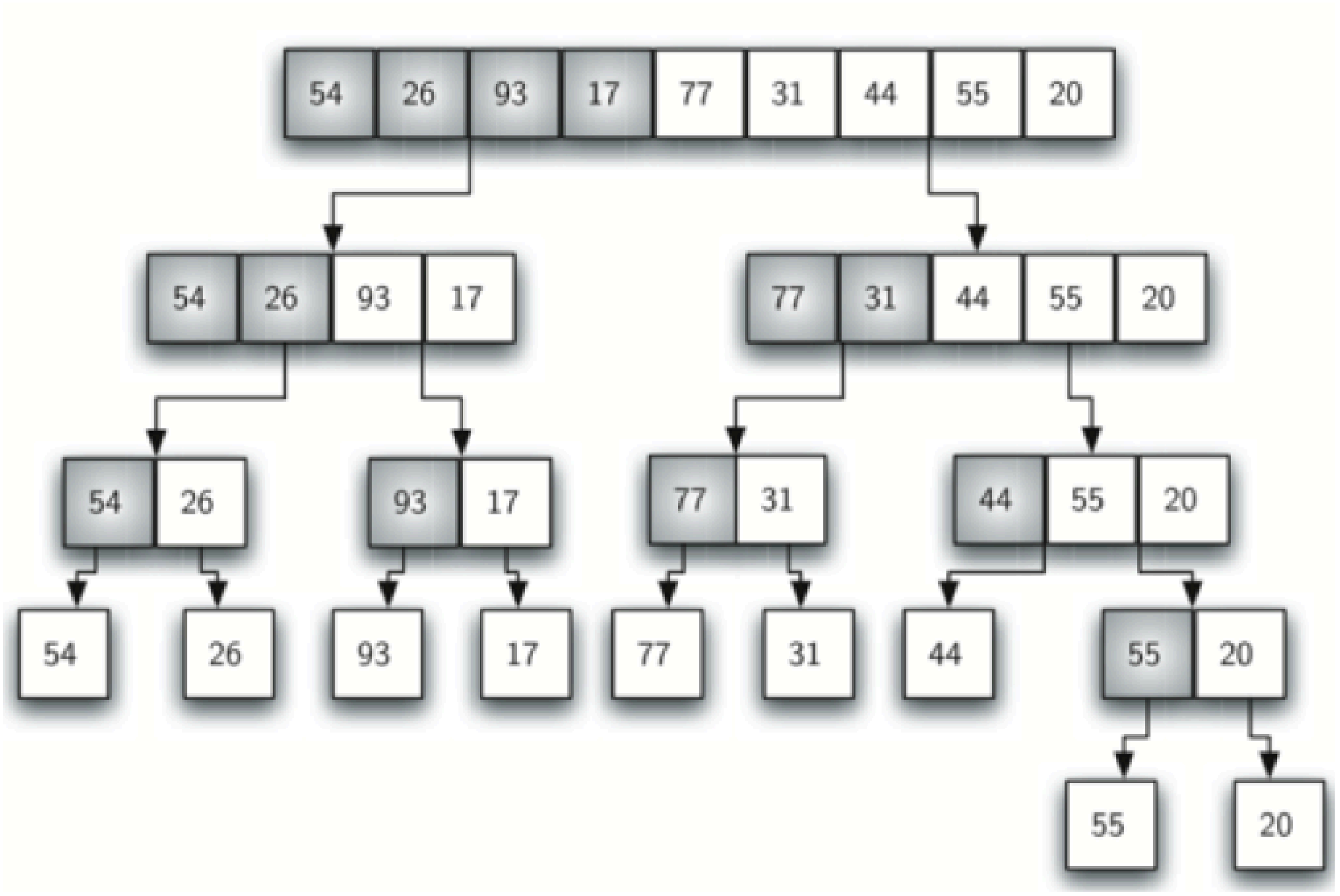

17 26 54 77 93 31 44 55 20
Need to insert 31
back into the sorted lis
17 26 54 77 93 44 55 20
93>31 so shift it
to the right
17 26 54 77 93 44 55 20
77>31 so shift it
to the right
17 26 54 77 93 44 55 20
54>31 so shift it
to the right
17 26 31 54 77 93 44 55 20
26<31 so insert 31
in this position

```python
def insertion_sort(a_list):
    for index in range(1, len(a_list)):
        current_value = a_list[index]
        position = index
        while position > 0 and a_list[position - 1] > current_value:
            a_list[position] = a_list[position - 1]
            position = position - 1
        a_list[position] = current_value
```

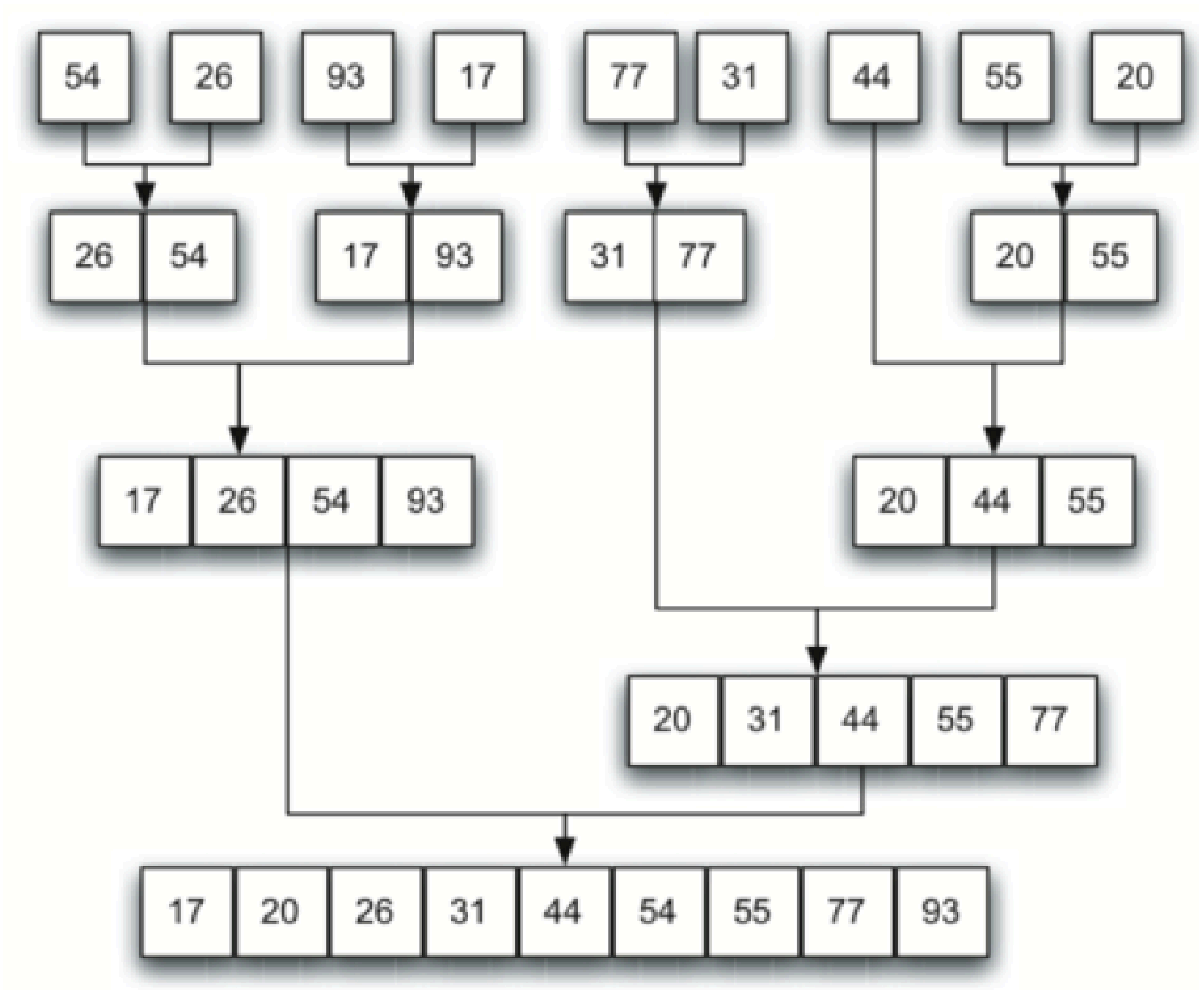
15.2 Sorting with Recursion

4. Merge Sort (合并排序)

- Splitting the List in a Merge Sort



- Merge Together



- Splitting the List in a Merge Sort

```
def merge_sort(a_list):
    print("Splitting ", a_list)
    if len(a_list) > 1:
        mid = len(a_list) // 2
        left_half = a_list[:mid]
        right_half = a_list[mid:]

        merge_sort(left_half)
        merge_sort(right_half)
        i = 0
        j = 0
        k = 0
```

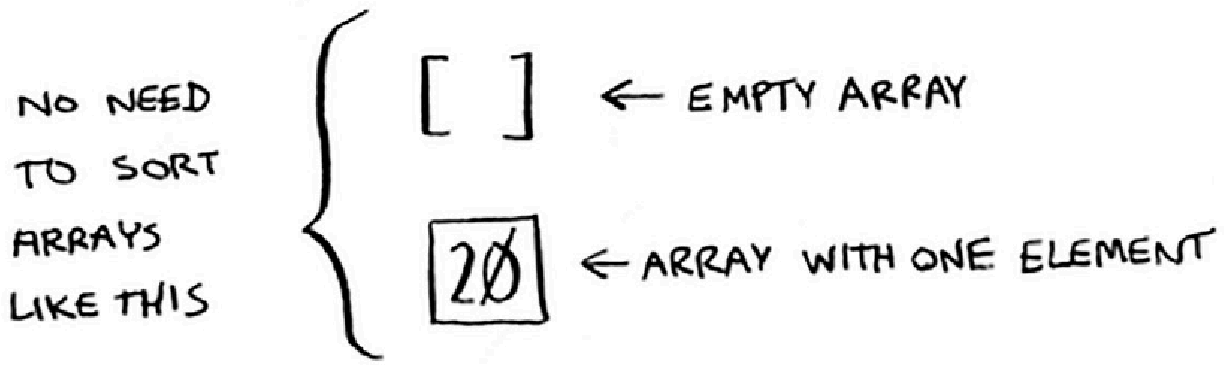
- Merge Together

```
#continue
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                a_list[k] = left_half[i]
                i = i + 1
            else:
                a_list[k] = right_half[j]
                j = j + 1
            k = k + 1

        while i < len(left_half):
            a_list[k] = left_half[i]
            i = i + 1
            k = k + 1

        while j < len(right_half):
            a_list[k] = right_half[j]
            j = j + 1
            k = k + 1
    print("Merging ", a_list)
```

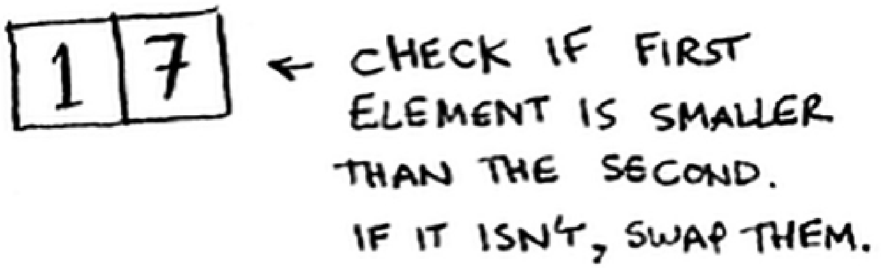
- In order to analyze the merge_sort function, we need to consider the two distinct processes that make up its implementation.
- The result of this analysis is that log n splits, each of which costs n for a total of n log n operations. A merge sort is an $O(nlogn)$ algorithm.

5. Quick Sort (快速排序)

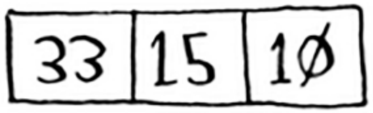
- Base case



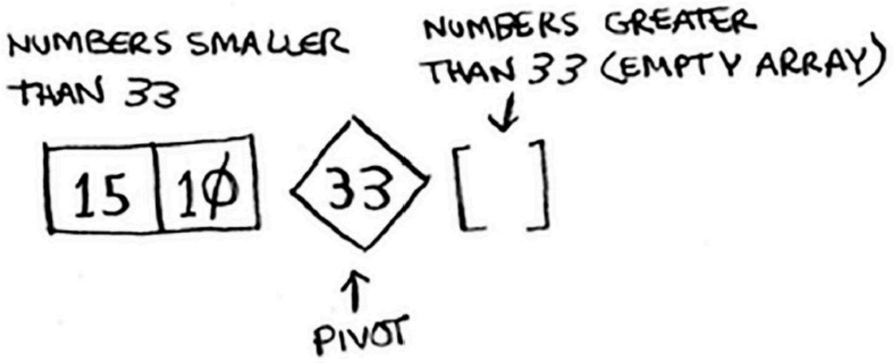
- An array with two elements is pretty easy to sort, too.


1
7
← CHECK IF FIRST ELEMENT IS SMALLER THAN THE SECOND. IF IT ISN'T, SWAP THEM.

- What about an array of three elements?



- We use D&C to solve this problem. Let's pick a pivot at first, say, 33.



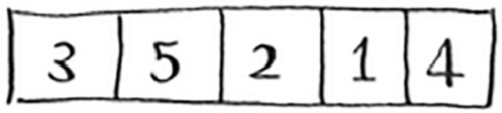
  - This is called partitioning. Now you have:

    A sub-array of all the numbers less than the pivot

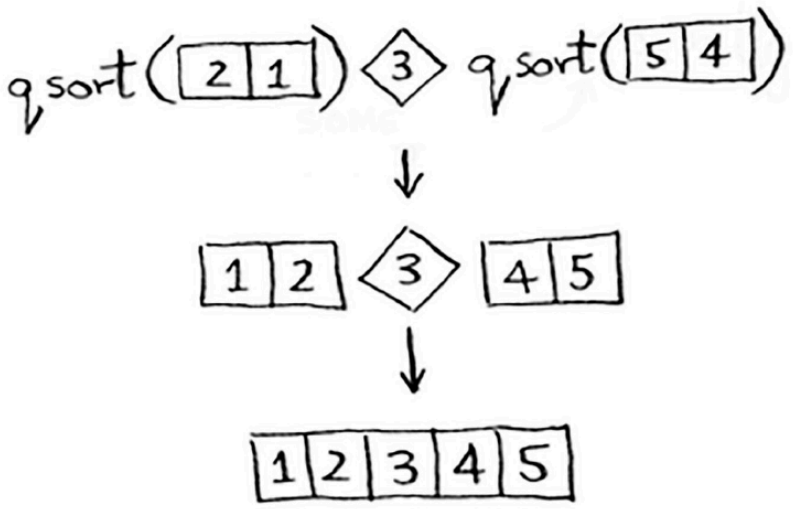
    The pivot

    A sub-array of all the numbers greater than the pivot

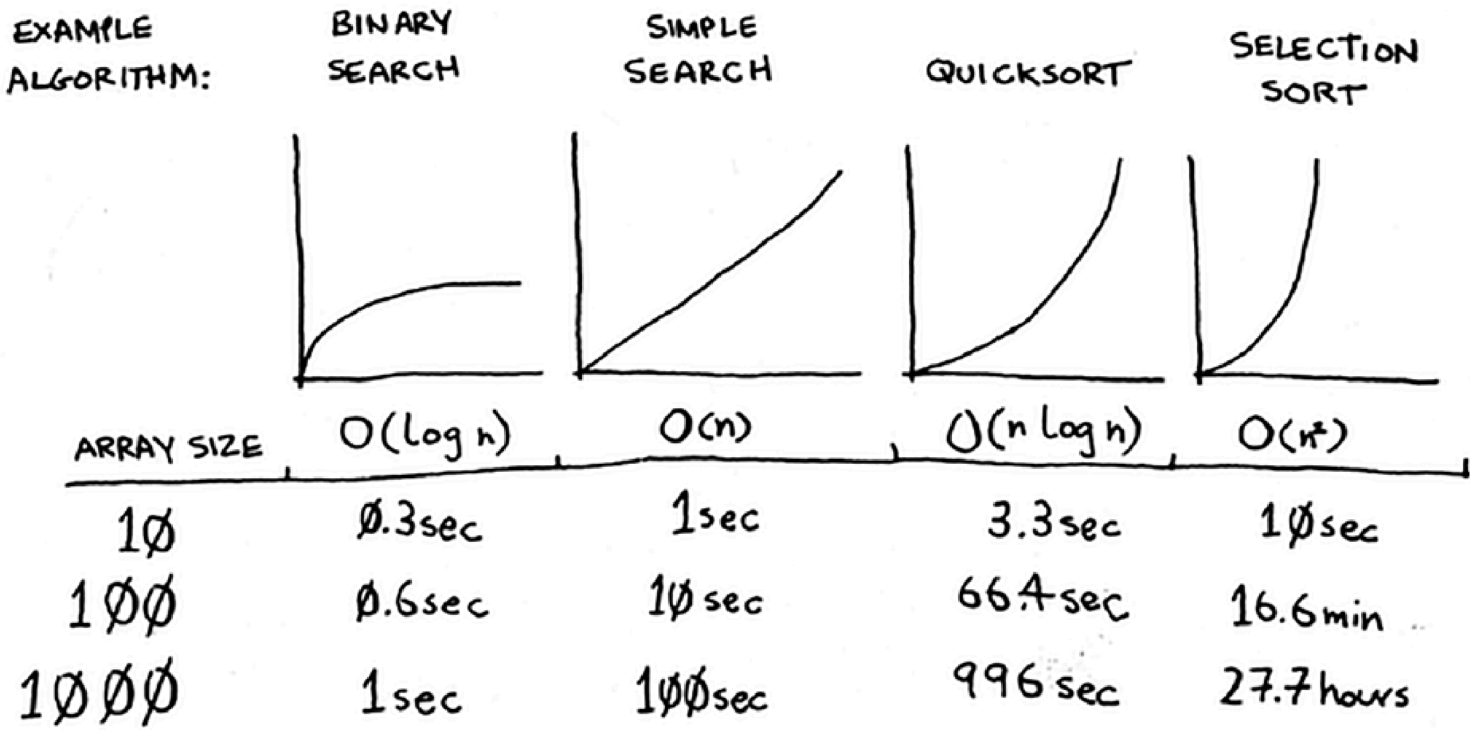
- If the sub-arrays are sorted, then you can combine the whole thing like this—left array + pivot + right array—and you get a sorted array.
- Suppose you have this array of five elements.



- For example, suppose you pick 3 as the pivot. You call quicksort on the sub-arrays.



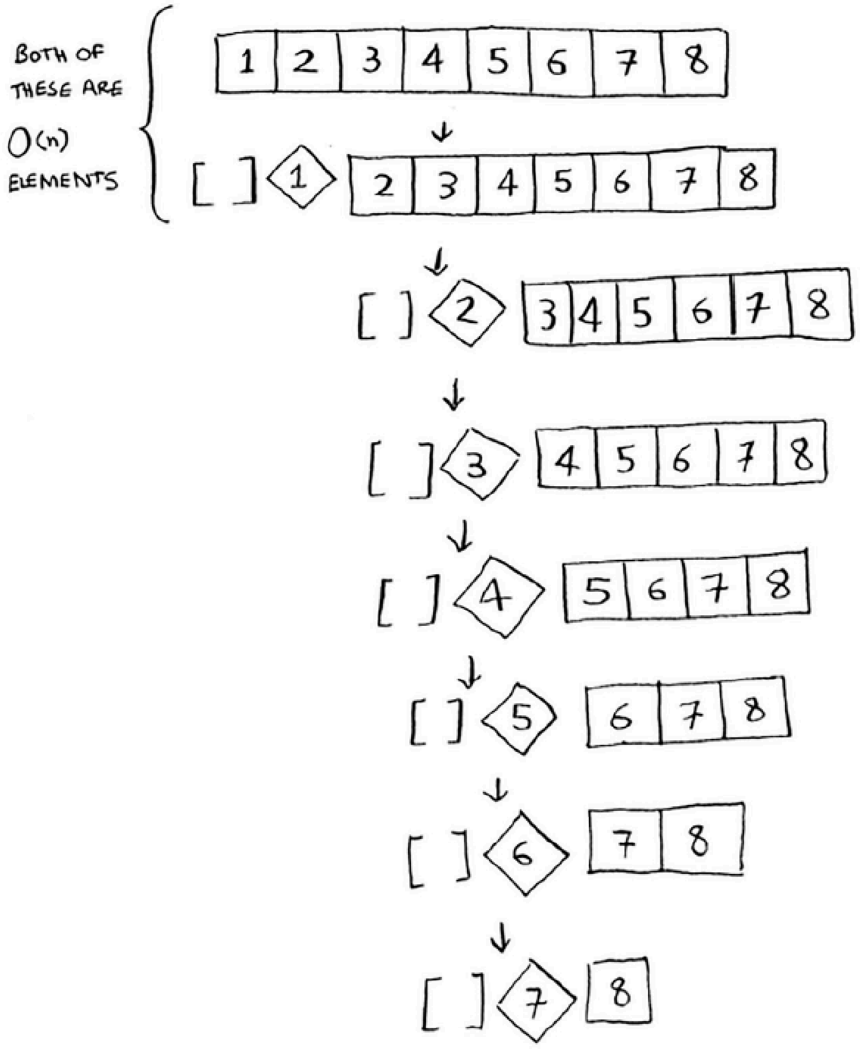
- Quicksort is unique because its speed depends on the pivot you choose.



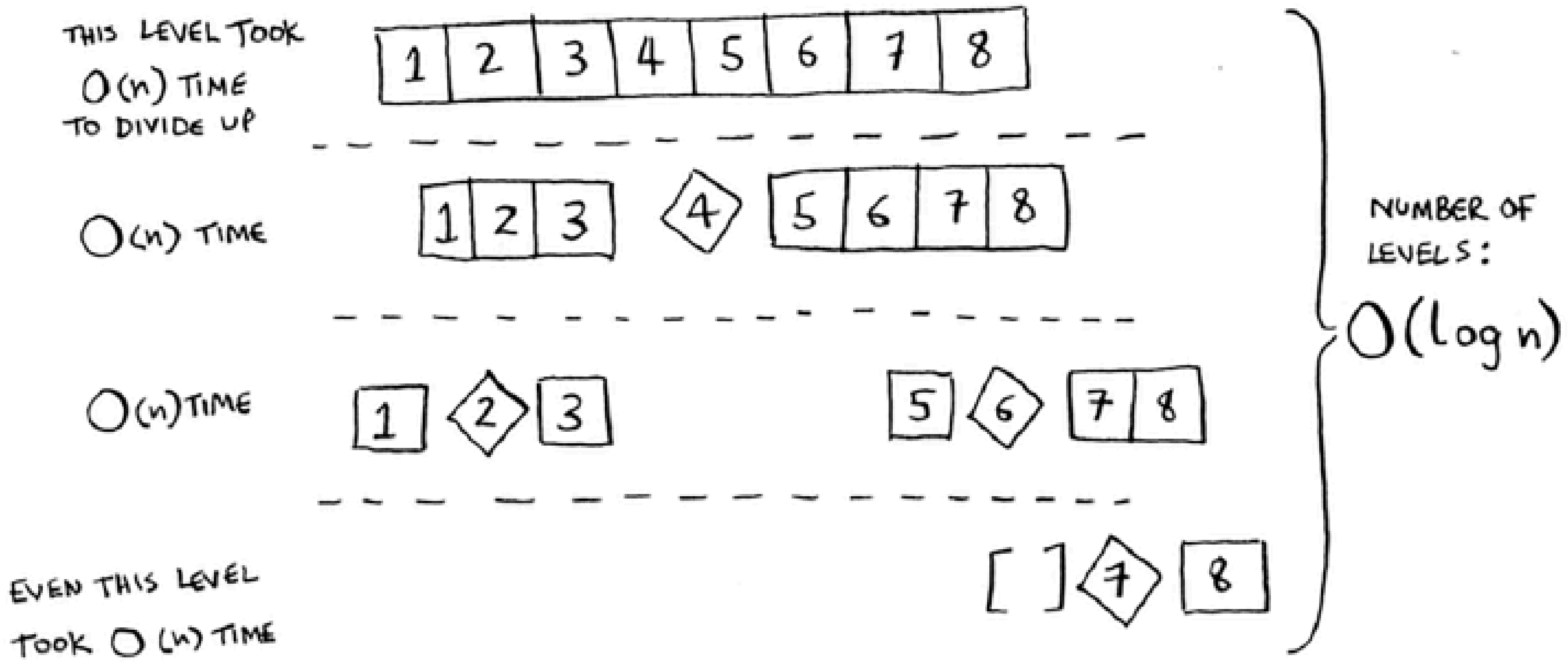
EXAMPLE ALGORITHM:	BINARY SEARCH	SIMPLE SEARCH	QUICKSORT	SELECTION SORT
ARRAY SIZE	O(log n)	O(n)	O(n log n)	O(n^2)
1ø	ø.3sec	1sec	3.3sec	1øsec
1øø	ø.6sec	1ø sec	66.4sec	16.6min
1øøø	1sec	1øøsec	996 sec	27.7hours

- Actually, the big O of the quick sort algorithm depends on the pivot you pick.
- In the best case, the big O of quick sort is $O(nlogn)$. However, in the worst case, the big O of it turns to be $O(n^2)$.
- Why?

- Worst Case


BOTH OF THESE ARE O(n) ELEMENTS
1 2 3 4 5 6 7 8
[ ] 1 2 3 4 5 6 7 8
[ ] 2 3 4 5 6 7 8
[ ] 3 4 5 6 7 8
[ ] 4 5 6 7 8
[ ] 5 6 7 8
[ ] 6 7 8
[ ] 7 8

- Best Case



- The average case is the best case, if you pick pivot randomly.

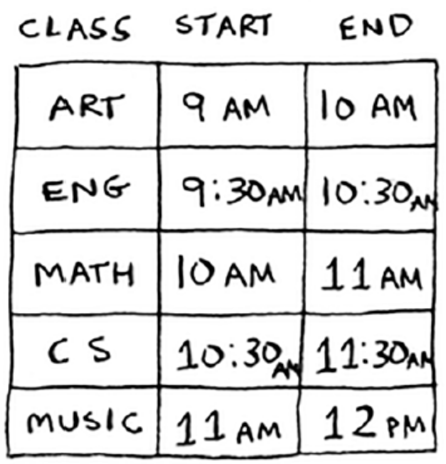
- A variant of the Insertion Sort: Shell Sort
- The sorting algorithm in Python: Timsort
- Timsort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data.

15.3 Greedy Algorithm

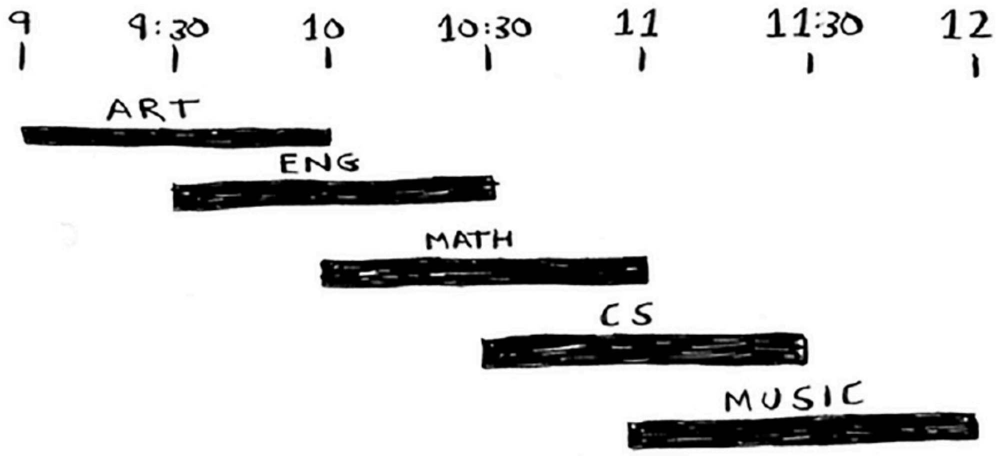
Greedy Algorithm（贪心算法）

- A simple and intuitive problem-solving strategy.
- Greedy algorithms are often described as short-sighted because they make each decision based only on the information available at that moment, aiming for immediate benefit. (贪心算法通常被形容为“短视”的策略，因为它们每一步都只基于当前可见的信息做出选择，目标是立即获得最好的结果。）
- While greedy algorithms can sometimes serve as heuristics , they are a distinct class with well-defined properties and correctness criteria. (虽然贪心算法在某些场景中可以被视为一种启发式方法（Heuristic），但它们属于一个具有明确定义和正确性判据的独立算法类别。）
- Example: Classroom Scheduling Problem

- Suppose you have a classroom and want to hold as many classes here as possible. You get a list of classes.

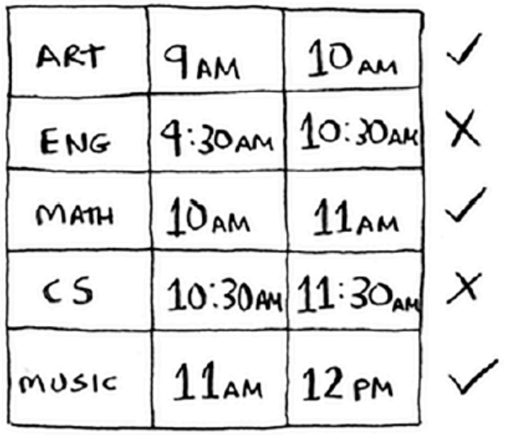


CLASS	START	END
ART	9 AM	10 AM
ENG	9:30 AM	10:30 AM
MATH	10 AM	11 AM
CS	10:30 AM	11:30 AM
MUSIC	11 AM	12 PM



- You want to hold as many classes as possible in this classroom. How do you pick what set of classes to hold, so that you get the biggest set of classes possible?

- Here's how the greedy algorithm works
- Pick the class that ends the soonest. This is the first class you'll hold in this classroom.
- Now, you have to pick a class that starts after the first class. Again, pick the class that ends the soonest. This is the second class you'll hold.



ART	9AM	10AM	✓
ENG	9:30AM	10:30AM	✗
MATH	10AM	11AM	✓
CS	10:30AM	11:30AM	✗
MUSIC	11AM	12PM	✓

Python Solution by Greedy Algorithm

```python
# 原始课程数据，每个元组包含：(课程名, 开始时间, 结束时间)
courses = [("数学", 9, 10),("物理", 9.5, 10.5),("化学", 10, 11),
           ("生物", 10.5, 11.5),("英语", 11, 12),("历史", 8, 9),
           ("地理", 12, 13),("政治", 13, 14),("语文", 12.5, 13.5)]

# 按结束时间排序
sorted_courses = sorted(courses, key=lambda x: x[2])  # x[2] 是结束时间

# 贪婪选择不冲突的课程
selected = []
earliest_end = 0
for name, start, end in sorted_courses:
    if start >= earliest_end:
        selected.append((name, start, end))
        earliest_end = end

# 输出选中的课程
for course in selected:
    print(course)
```

找零问题：用贪婪算法解决

给定一组硬币面额和一个金额，使用最少数量的硬币来找出该金额。

```python
def greedy_coin_change(coins, amount):
    coins.sort(reverse=True)  # 从大到小排序
    result = []
    for coin in coins:
        while amount >= coin:
            amount -= coin
            result.append(coin)
    return result if amount == 0 else None

# 示例：标准货币系统
coins = [1, 5, 10, 25]  # 美分
amount = 63
print(greedy_coin_change(coins, amount))
# 输出： [25, 25, 10, 1, 1, 1]
```

```python
# 示例：标准货币系统
coins = [1, 5, 10, 21, 25]  # 如果有21美分这个选项
amount = 63
print(greedy_coin_change(coins, amount))
# 输出： [25, 25, 10, 1, 1, 1] #贪婪算法失败，最优是21*3
```

Summary

- Sorting Algorithms
- Greedy Algorithm