Sorting

Mustafa Hajij
MSDS program
University of San Francisco

Sorting

- Sorting algorithms are essential tools in computer science used to organize and arrange data efficiently.
- They play a crucial role in various applications, including databases, operating systems, and web development.
- Sorting algorithms are designed to rearrange elements in a specific order, such as ascending or descending, based on certain criteria.

*https://en.wikipedia.org/wiki/Triangle_inequality

Bubble sort

- Repeatedly compares adjacent elements and swaps them if they are in the wrong order.
- Smaller elements "bubble" to the top of the list with each iteration.

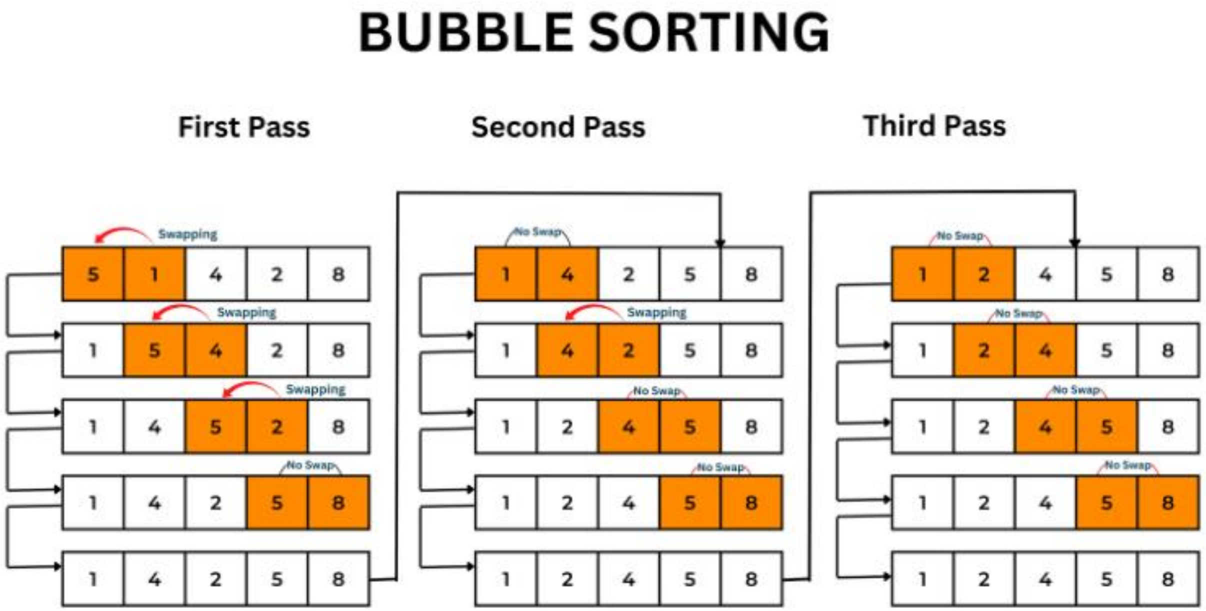


Image credit : https://favtutor.com/blogs/bubble-sort-python

Bubble sort Code

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array:", arr)
```

Question : why is this $O(n^2)$?

Merge sort

- Merge Sort divides array into halves recursively.
- Each sub-array is sorted individually.
- Sorted sub-arrays are then merged to form a single sorted array.
- Utilizes divide-and-conquer strategy.
- Merge sort is faster than bubble sort. It runs in O(n log(n)) time.

Merge sort

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)
```

```
def merge(left, right):
    result = []
    left_index = right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1

    result.extend(left[left_index:])
    result.extend(right[right_index:])

    return result
```

Merge sort complexity

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)
```

The merge_sort function recursively divides the input array into halves until each sub-array contains only one element. This function can be called at most O(log n) time.

Merge sort complexity

```
def merge(left, right):
    result = []
    left_index = right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1

    result.extend(left[left_index:])
    result.extend(right[right_index:])

    return result
```

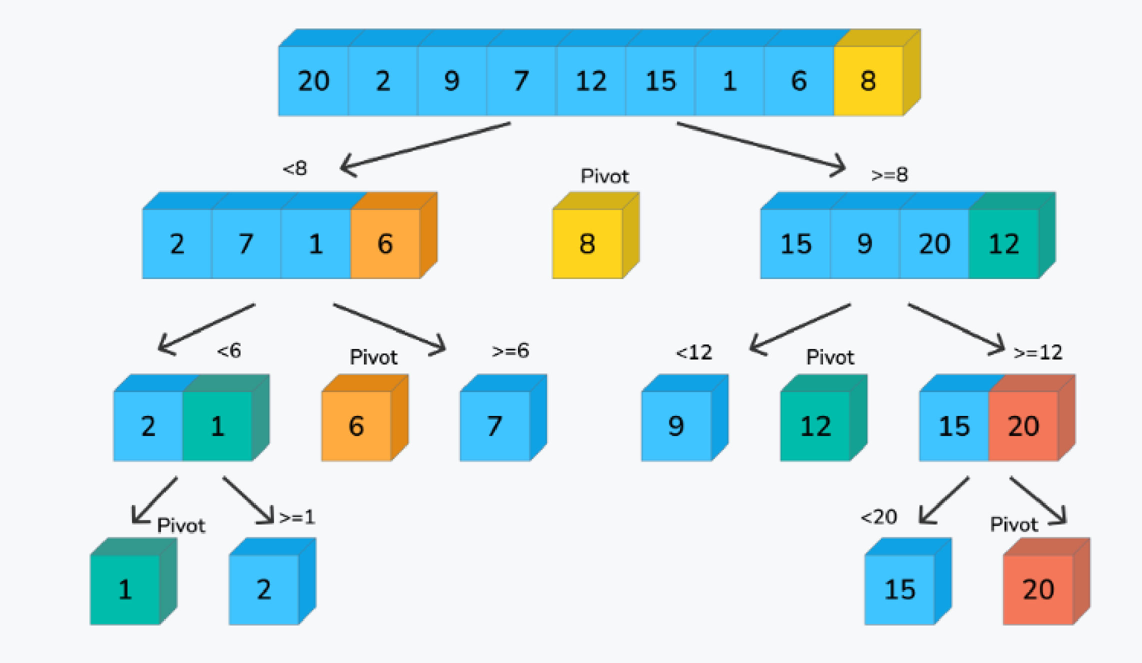
After the array is divided into single-element sub-arrays, the merge function combines these sub-arrays back together. Each merge operation takes O(n) time because every element needs to be compared and placed in the correct order in the merged array.

Quicksort algorithm

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr)-1]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)
```



Ref: https://workat.tech/problem-solving/tutorial/sorting-algorithms-quick-sort-merge-sort-dsa-tutorials-6j3h98lk6j2w

Quicksort algorithm

- **Worst-case complexity**: O(n^2)
- **Average complexity** : O(n log n)
- **Idea**: Pick a pivot, partition the array into elements smaller and larger than the pivot, then recursively partition until small enough to sort trivially.
- **Dynamic Pivot Selection**: Unlike mergesort's static splitting, Quicksort adapts by selecting a pivot element.
- **Efficiency**: Quicksort moves elements more efficiently than bubble sort, typically more than one position per iteration.
- **In-Place Sorting**: Quicksort can operate in-place, saving memory compared to mergesort's temporary arrays.

Summary

- Theoretically, sorting cannot be done faster than O(n log n)
- Merge and quicksort are primary algorithms. They are both examples of divide and conquer algorithms.
  - Mergesort recursively merges two sorted halves, requiring additional memory.
  - Quicksort, on the other hand, partitions the array instead of sorting halves, and typically operates in-place, offering improved efficiency.