Bubble Sort

What is it:

- Bubble Sort is a simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- The pass through the list is repeated until the list is sorted.

Where we need it:

- Bubble Sort is often used for educational purposes to teach sorting concepts. It's not suitable for large datasets due to its inefficiency.

How it works:

- Start at the beginning of the list.
- Compare adjacent elements.
- Swap them if the first element is greater than the second.
- Move to the next pair and repeat.
- Continue until no more swaps are needed.

Use cases:

- Educational demonstrations
- Small datasets where simplicity is preferred over efficiency

Big O:

- Best case: O(n) (when the list is already sorted)
- Average case: O(n 2)
- Worst case: O(n2)

Quick Sort

What is it:

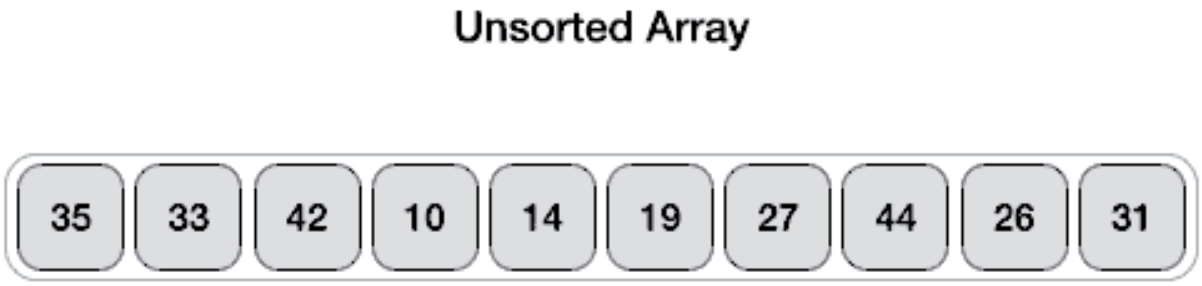
- Quick Sort is a divide-and-conquer algorithm that works by selecting a 'pivot' element and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

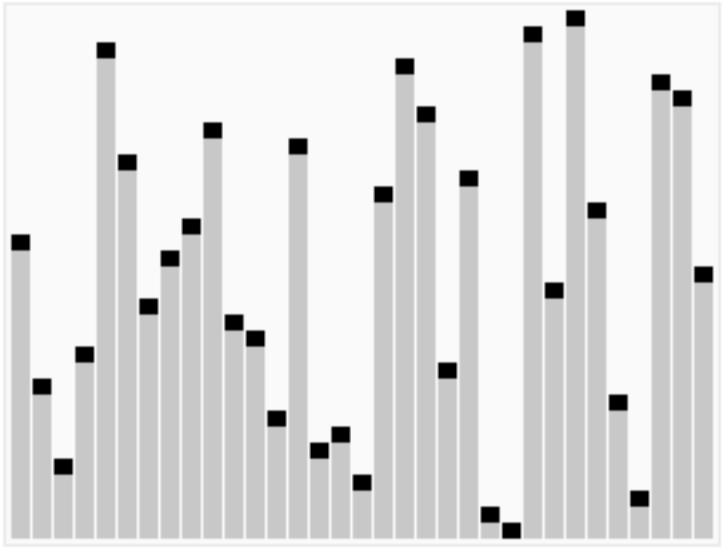
Where we need it:

- Quick Sort is used for efficient sorting of large datasets and is often used in practice due to its average-case performance.

How it works:

- Choose a pivot element from the array.
- Partition the array into two sub-arrays: one with elements less than the pivot and one with elements greater than the pivot.
- Recursively apply the same process to the sub-arrays.





Use cases:

- General-purpose sorting
- Applications requiring efficient sorting for large datasets

Big O:

- Best case: O(nlogn)

- Average case: O(nlogn)
- Worst case: O(n2) (when the pivot selection is poor, e.g., always the smallest or largest element)

Merge Sort

What is it:

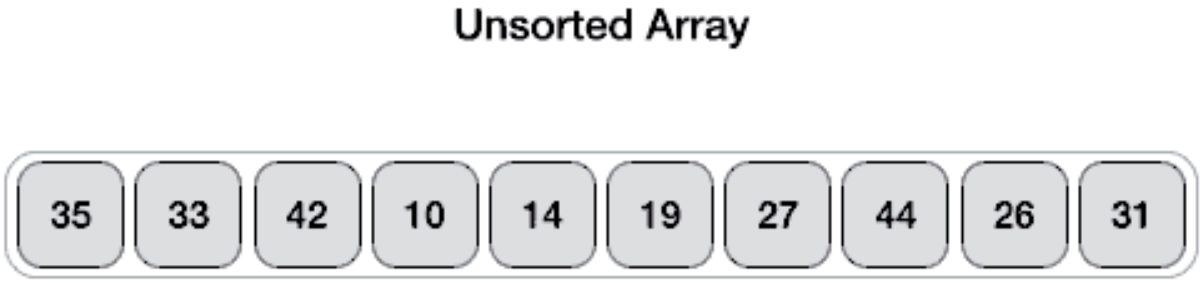
- Merge Sort is a divide-and-conquer algorithm that divides the array into halves, sorts each half, and then merges the sorted halves to -0 - produce a single sorted array.

Where we need it:

- Merge Sort is used for its stable sorting and predictable performance. It is particularly useful when working with linked lists or large datasets.

How it works:

- Divide the unsorted list into n sub-lists, each containing one element.
- Repeatedly merge sub-lists to produce new sorted sub-lists until there is only one sub-list remaining.



Use cases:

- Stable sorting required
- Large datasets
- Sorting linked lists

Big O:

- Best case: O(nlogn)
- Average case: O(nlogn)
- Worst case: O(nlogn)

Binary Search

What is Binary Search?

- an efficient algorithm for finding a target value within a sorted array.

- It works by repeatedly dividing the search interval in half, which allows it to quickly narrow down the possible locations of the target value.

Where We Need It:

- Sorted Data: Binary search can only be applied to sorted arrays or lists. If the data is not sorted, you need to sort it first, which might not always be efficient.
- Efficient Lookup: Binary search is useful when you need to perform repeated searches in a sorted dataset and want to optimize for fast lookup times.

How It Works:

- Initialize Pointers:
- Set two pointers: low (start of the array) and high (end of the array).
- Find the Middle:
  - Compute the middle index mid using mid = (low + high) // 2.
  - Compare and Narrow Down:
  - Compare the target value with the value at mid.

    - If the target is equal to the value at mid, return mid (the index of the target).
    - If the target is less than the value at mid, adjust the high pointer to mid - 1 (search in the left half).
    - If the target is greater than the value at mid, adjust the low pointer to mid + 1 (search in the right half).
- Repeat:
    - Continue the process until the low pointer exceeds the high pointer. If the target is not found, return -1 or some indication that the target is not present.

Use Cases:

- Searching in Sorted Arrays: Binary search is often used to find elements in sorted arrays or lists.
- Database Indexing: Many database systems use binary search or its variations for indexing.
- Problem Solving: Binary search can be applied to problems involving ranges or finding the boundary in a sorted dataset.

Big O Notation:

Operation	Best Case	Average Case	Worst Case
Binary Search	(O(1))	(O(\log n))	(O(\log n))

Python Implementation

```
def binarySearch(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1  # Target not found
```

Problem Solving : Sliding Window

- a powerful approach for solving problems related to arrays or lists, where you need to find a subarray or subsequence that satisfies certain conditions.
- This technique is often used to optimize solutions that would otherwise involve nested loops.

How it Works :

Example : Maximum Sum of a Subarray of Size k

- Given an array of integers and a positive integer k, find the maximum sum of any contiguous subarray of size k.
- Input: arr = [1, 2, 3, 4, 5, 6, 7, 8, 9], k = 3
- Output: 24 (The subarray with the maximum sum is [7, 8, 9])

```
def max_sum_subarray(arr, k):
    if not arr or k <= 0:
        return 0

    # Initialize the window sum with the sum of the first k elements
    window_sum = sum(arr[:k])
    max_sum = window_sum

    # Slide the window from the start to the end of the array
    for i in range(len(arr) - k):
        window_sum = window_sum - arr[i] + arr[i + k]
        max_sum = max(max_sum, window_sum)

    return max_sum

# Example usage
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
k = 3
print(max_sum_subarray(arr, k))  # Output: 24
```

```
24
```

Leetcode problemes

- 2255 : https://leetcode.com/problems/count-prefixes-of-a-given-string/description/
- 2446 : https://leetcode.com/problems/determine-if-two-events-have-conflict/description/
- 2586 : https://leetcode.com/problems/count-the-number-of-vowel-strings-in-range/description/

```
# 2255 : https://leetcode.com/problems/count-prefixes-of-a-given-string/description/

def count_prefixes(words, s):
    count = 0

    for word in words:
        if s.startswith(word):
            count += 1

    return count

# Example usage
words = ["a", "b", "c", "ab", "bc", "abc"]
s = "abc"
print(count_prefixes(words, s))  # Output: 3
```

```
3
```

```
# 2446 : https://leetcode.com/problems/determine-if-two-events-have-conflict/description/
def have_conflict(events):
    # Sort intervals by start time
    events.sort()

    # Iterate through the sorted intervals
    for i in range(len(events) - 1):
        # Check if the end time of the current interval is greater than or equal to the start time of the next interval
        if events[i][1] >= events[i + 1][0]:
            return True

    return False

# Example usage
events = [[1, 2], [2, 3], [3, 4]]
print(have_conflict(events))  # Output: False
```

```
True
```

```
# 2586 : https://leetcode.com/problems/count-the-number-of-vowel-strings-in-range/description/
def count_vowel_strings(words, left, right):
```

```
    vowels = {'a', 'e', 'i', 'o', 'u'}
    count = 0

    for i in range(left, right + 1):
        word = words[i]
        if word[0] in vowels and word[-1] in vowels:
            count += 1

    return count

# Example usage
words = ["are", "amy", "u"]
left = 0
right = 2
print(count_vowel_strings(words, left, right))  # Output: 2
```

```
2
```