Sorting Algorithm

🧠 MASTER NOTES ON SORTING ALGORITHMS

(Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort)

⚙️ What is Sorting?

Sorting means arranging a list of elements in a specific order — usually ascending or descending.

Example: `[4, 1, 3, 2] → [1, 2, 3, 4]`

Sorting is a foundational operation that:

- Speeds up searching (binary search only works on sorted data).
- Helps structure data for analysis and optimization.
- Simplifies complex algorithms (e.g., Kruskal's MST, median finding).

🔵 1. BUBBLE SORT

🧩 Definition

Bubble Sort is the simplest sorting algorithm.

It repeatedly **compares adjacent elements** and **swaps** them if they are in the wrong order.

After each pass, the largest element "bubbles" to its correct position at the end.

💡 Intuition

Suppose you are checking a list of numbers one by one, and every time two numbers are in the wrong order, you fix them immediately.

After you go through the list once, the biggest number is at the end.

Do this repeatedly, and you'll get a sorted list.

🌍 Real-life Analogy

Imagine you have soap bubbles of different sizes underwater.

If you let them go, the **largest bubble rises fastest and reaches the top first** — similarly, in Bubble Sort, the largest element "floats" to the top after each round.

-

⚙️ Step-by-Step Mechanism

1. Start from the beginning of the array.
2. Compare each adjacent pair of elements.
3. Swap if the left one is greater.
4. After the first pass, the largest element is at the end.
5. Repeat for remaining unsorted part.

If in any full pass **no swaps occur**, the array is already sorted (optimized bubble sort).

🧮 Detailed Dry Run

Input: `arr = [8, 7, 5, 4, 3]`

Pass 1 (`i=0`)

Step	Compare (arr[j], arr[j+1])	Action	Array State
j=0	(8,7)	8>7 → swap	[7,8,5,4,3]
j=1	(8,5)	8>5 → swap	[7,5,8,4,3]
j=2	(8,4)	8>4 → swap	[7,5,4,8,3]
j=3	(8,3)	8>3 → swap	[7,5,4,3,8]

✅ **Largest (8)** moved to the end.

Pass 2 (`i=1`)

Step	Compare	Action	Array
j=0	(7,5)	swap	[5,7,4,3,8]
j=1	(7,4)	swap	[5,4,7,3,8]
j=2	(7,3)	swap	[5,4,3,7,8]

✅ **7 settled at index 3.**

Pass 3 (`i=2`)

Step	Compare	Action	Array
j=0	(5,4)	swap	[4,5,3,7,8]
j=1	(5,3)	swap	[4,3,5,7,8]

✅ **5 settled at index 2.**

Pass 4 (`i=3`)

Step	Compare	Action	Array
j=0	(4,3)	swap	[3,4,5,7,8]

✅ Sorted.

✅ Final Output

`[3, 4, 5, 7, 8]`

🗒️ Python Code

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:  # optimization
            break
    return arr

arr = [8, 7, 5, 4, 3]
print("Bubble Sort:", bubbleSort(arr))
```

⏱️ Complexity Analysis

Case	Time	Reason
Best	O(n)	Already sorted
Average	O(n²)	Nested loops
Worst	O(n²)	Reverse sorted
Space	O(1)	In-place
Stable	✅	Equal elements stay in order

⚠️ Pitfalls

- Very slow for large arrays.
- Good only for small or almost-sorted datasets.

🟡 2. SELECTION SORT

🧩 Definition

Selection Sort **selects the smallest element** in the unsorted part of the array and places it in its correct position at the beginning.

💡 Intuition

Why do unnecessary swaps?

Just find the smallest item each time and put it where it belongs.

🌍 Analogy

If you're arranging books by thickness — you pick the thinnest book from the unsorted pile and place it in front. Repeat until done.

⚙️ Working Mechanism

1. Start at index `i=0` (first element).
2. Find the smallest element in `[i...end]`.
3. Swap it with `arr[i]`.
4. Move to next position (`i+1`).

After each pass, one element is fixed in correct place.

🧩 Detailed Dry Run

Input: `[4, 3, 5, 2, 1]`

i = 0

`min_index = 0`

j	arr[j]	arr[min_index]	Comparison	min_index
1	3	4	3<4 ✅	1
2	5	3	5<3 ❌	1
3	2	3	2<3 ✅	3
4	1	2	1<2 ✅	4

Swap arr[0]↔arr[4] → `[1,3,5,2,4]`

---

i = 1

`min_index = 1`

j	arr[j]	arr[min_index]	Comparison	min_index
2	5	3	❌	1
3	2	3	✅	3
4	4	2	❌	3

Swap arr[1]↔arr[3] → `[1,2,5,3,4]`

---

i = 2

`min_index = 2`

j	arr[j]	arr[min_index]	Comparison	min_index
3	3	5	✅	3
4	4	3	❌	3

Swap arr[2]↔arr[3] → [1,2,3,5,4]

---

i = 3

min_index = 3

j	arr[j]	arr[min_index]	Comparison	min_index
4	4	5	✅	4

Swap arr[3]↔arr[4] → [1,2,3,4,5]

✅ Final Array: [1,2,3,4,5]

🧾 Python Code

```
class SelectionSort:
    def selection(self, arr):
        for i in range(len(arr) - 1):
            min_index = i
            for j in range(i + 1, len(arr)):
                if arr[j] < arr[min_index]:
                    min_index = j
            arr[i], arr[min_index] = arr[min_index], arr[i]
        return arr

arr = [4, 3, 5, 2, 1]
print("Selection Sort:", SelectionSort().selection(arr))
```

⏱️ Complexity

Case	Time	Space	Stable
All	$O(n^2)$	O(1)	❌

⚠️ Pitfalls

- Not adaptive — even sorted data takes $O(n^2)$.
- Not stable (equal elements may change order).

🟢 3. INSERTION SORT

🧩 Definition

Insertion Sort **builds the sorted array one element at a time.**

Each new element is **inserted** into its proper position among already sorted ones.

💡 Intuition

If you already have a sorted list and one new number appears, it's faster to just put it where it belongs than re-sorting the whole list.

🌍 Analogy

Sorting playing cards: you pick each new card and slide it into the right position among the ones already arranged in your hand.

⚙️ Working

1. Treat `arr[0]` as sorted.
2. For each new element (`i` from 1 → n-1):
    - Save it in `temp`.
    - Move all larger elements to the right.
    - Insert `temp` in the correct position.

🧩 Detailed Dry Run

Input: `[5, 4, 3, 2, 1]`

i = 1 → key=4

Compare with arr[0]=5 → shift → insert

→ `[4,5,3,2,1]`

i = 2 → key=3

Compare with 5 → shift

Compare with 4 → shift

Insert 3

→ `[3,4,5,2,1]`

i = 3 → key=2

Compare with 5 → shift

Compare with 4 → shift

Compare with 3 → shift

Insert 2

→ `[2,3,4,5,1]`

`i = 4 → key=1`

`Compare with 5 → shift`

`Compare with 4 → shift`

`Compare with 3 → shift`

`Compare with 2 → shift`

`Insert 1`

→ `[1,2,3,4,5]`

✅ `Final:` `[1,2,3,4,5]`

🧾 Code

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        temp = arr[i]
        j = i - 1
        while j >= 0 and temp < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = temp
    return arr

arr = [5, 4, 3, 2, 1]
print("Insertion Sort:", insertionSort(arr))
```

⏱️ Complexity

Case	Time	Space	Stable
Best	O(n)	O(1)	✅
Avg	O(n²)	O(1)	✅
Worst	O(n²)	O(1)	✅

⚠️ Notes

- Works great for nearly sorted data.
- Used in hybrid algorithms (like Python's Timsort).

🔴 4. MERGE SORT

🧩 Definition

Merge Sort is a **Divide and Conquer** algorithm.

It divides the array into two halves, recursively sorts each half, and merges the sorted halves.

💡 Intuition

Sorting a large list directly is tough — so break it into smaller lists, sort those, and merge them efficiently.

🌍 Analogy

You and your friend each have sorted piles of cards.

You merge them by **repeatedly taking the smaller top card** from either pile — that's exactly what merge sort does.

⚙️ Working

1. Divide the array into two halves until each part has one element.
2. Merge two sorted halves into one sorted list.

🧩 Detailed Dry Run

Input: `[8, 7, 5, 4, 3]`

Step 1: Divide

Split into `[8, 7, 5]` and `[4, 3]`.

Step 2: Left half → `[8, 7, 5]`

Split again → `[8]`, `[7, 5]`

Split `[7,5]` → `[7]`, `[5]`

Merge `[7]` + `[5]` → `[5,7]`

Merge `[8]` + `[5,7]`:

- Compare 8>5 → pick 5
- Compare 8>7 → pick 7
- Pick remaining 8

  → `[5,7,8]`

Step 3: Right half → `[4,3]`

Split → `[4]`, `[3]` → merge → `[3,4]`

Step 4: Final Merge

Left `[5,7,8]`, Right `[3,4]`

Step	Left[i]	Right[j]	Pick	Merged
1	5	3	3	[3]

Step	Left[i]	Right[j]	Pick	Merged
2	5	4	4	[3,4]
3	5	–	5	[3,4,5]
4	7	–	7	[3,4,5,7]
5	8	–	8	[3,4,5,7,8]

✅ Final: [3,4,5,7,8]

🧾 Code

```
def merge(arr, low, mid, high):
    merged = []
    l, r = low, mid + 1
    while l <= mid and r <= high:
        if arr[l] <= arr[r]:
            merged.append(arr[l])
            l += 1
        else:
            merged.append(arr[r])
            r += 1
    while l <= mid:
        merged.append(arr[l])
        l += 1
    while r <= high:
        merged.append(arr[r])
        r += 1
    for i in range(len(merged)):
        arr[low + i] = merged[i]

def mergeSort(arr, low, high):
    if low >= high:
        return
    mid = (low + high) // 2
    mergeSort(arr, low, mid)
    mergeSort(arr, mid + 1, high)
    merge(arr, low, mid, high)

arr = [8, 7, 5, 4, 3]
mergeSort(arr, 0, len(arr) - 1)
print("Merge Sort:", arr)
```

⏱️ Complexity

Case	Time	Space	Stable
All	O(n log n)	O(n)	✅

⚠️ Notes

- Always predictable performance.
- Needs extra space (not in-place).
- Great for linked lists (easy merging).

🟣 5. QUICK SORT

🧩 Definition

Quick Sort also uses **Divide and Conquer**, but instead of merging, it partitions the array around a **pivot** so smaller elements go left and larger go right.

💡 Intuition

If you can place one element exactly where it belongs (pivot), the rest can be solved by recursion.

🌍 Analogy

Teacher picks one student as pivot; everyone shorter goes left, taller goes right — now sort both sides.

⚙️ Working

1. Pick a pivot (last element often).
2. Rearrange so smaller values move left, larger right.
3. Recursively sort both parts.

🧩 Detailed Dry Run

Input: `[8, 7, 5, 4, 3]`

Pivot = 3

Step	j	arr[j]	Action	i	Array
0	-	—	—	-1	[8,7,5,4,3]
1	0	8>3	no swap	-1	[8,7,5,4,3]
2	1	7>3	no swap	-1	[8,7,5,4,3]
3	2	5>3	no swap	-1	[8,7,5,4,3]
4	3	4>3	no swap	-1	[8,7,5,4,3]

Swap pivot (arr[4]) ↔ arr[i+1]=arr[0] → [3,7,5,4,8]

Left → [], Right → [7,5,4,8]

Repeat:

- Pivot=8 → all smaller → [3,7,5,4,8]
- Pivot=4 in [7,5,4] → [3,4,5,7,8]

✅ Final: [3,4,5,7,8]

🧾 Code

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quickSort(arr, low, high):
    if low < high:
        pIndex = partition(arr, low, high)
        quickSort(arr, low, pIndex - 1)
        quickSort(arr, pIndex + 1, high)

arr = [8, 7, 5, 4, 3]
quickSort(arr, 0, len(arr) - 1)
print("Quick Sort:", arr)
```

⏱️ Complexity

Case	Time	Space	Stable
Best	O(n log n)	O(log n)	❌
Avg	O(n log n)	O(log n)	❌
Worst	$O(n^2)$	O(log n)	❌

🔎 Understanding Quick Sort's Time Complexity

Quick Sort is a **divide-and-conquer** algorithm.

The total time depends on **how well the pivot divides the array** during each recursive call.

If the pivot divides the array **evenly**, we get many small, balanced subproblems.

If the pivot divides the array **unevenly**, we get a very deep recursion chain — and that's where the time blows up.

Let's understand both ends of this behavior.

🧠 Recap: How Quick Sort Works

1. Choose a **pivot** element.
2. **Partition** the array:
    - Elements smaller than pivot → left side.
    - Elements greater than pivot → right side.
3. Recursively sort both subarrays.

🟢 BEST CASE — O(n log n)

When it happens:

The pivot **perfectly divides** the array into two equal halves at each step.

That means:

> Each pivot → splits array into halves of size n/2.

Example:

Input = `[4, 2, 6, 1, 5, 3]`

Pivot = 4

Left → `[2, 1, 3]`, Right → `[6, 5]`

Both subarrays are roughly half of the original.

Step-by-step logic:

At each level of recursion, we do:

- **n comparisons** (to partition the array)
- And then recursively sort the two halves.

So the recurrence relation is:

$$T(n) = 2 * T(n/2) + O(n)$$

Here's what that means:

- `O(n)` for partitioning
- `2*T(n/2)` for sorting two halves

Now, if you solve that recurrence (using the Master Theorem):

$$T(n) = O(n \log n)$$

This is similar to **Merge Sort** — but with smaller constant factors because it's in-place (no extra arrays).

Intuitive View

Imagine cutting a pizza in half again and again:

If you perfectly halve it each time, you get **$\log_2(n)$** cuts.

Each level of cutting affects all slices (O(n) work per level).

Hence:

Total work = $\log_2(n)$ levels × O(n) per level = O(n log n)

🔴 WORST CASE — $O(n^2)$

When it happens:

The pivot divides the array **very unevenly**, such as:

- One side has **n-1 elements**
- The other side has **0 elements**

This happens if:

- The pivot is always the **smallest** or **largest** element.
- Input is **already sorted** (in ascending or descending order) and pivot selection is bad (like always picking the first or last element).

Example:

Let's say the array is already sorted: `[1, 2, 3, 4, 5]`

And your pivot choice is **always the last element**.

Trace:

- Pivot = 5 → left = `[1,2,3,4]`, right = `[]`

- Pivot = 4 → left = `[1,2,3]`, right = `[]`
- Pivot = 3 → left = `[1,2]`, right = `[]`
- Pivot = 2 → left = `[1]`, right = `[]`
- Pivot = 1 → done

What happens here:

Each partition only removes **one element** (the pivot) per recursion.

So your recursion tree is **a straight line**, not a balanced tree.

Visually:

```
[1,2,3,4,5]  (n elements)
 → [1,2,3,4] (n-1 elements)
  → [1,2,3] (n-2 elements)
   → [1,2] (n-3 elements)
    → [1] (n-4 elements)
```

Counting Comparisons

At each partition step, you compare each element once to the pivot.

So total comparisons:

$$(n-1) + (n-2) + (n-3) + \ldots + 1$$

That's an **arithmetic series**:

$$= n(n-1)/2 = O(n^2)$$

Intuitive View

If the pivot is always bad, Quick Sort becomes **a glorified bubble sort** —

no real "divide-and-conquer" happens, just linear scanning again and again.

⚖️ Comparing Both Scenarios

Case	Pivot Split	Recursion Tree Depth	Work per Level	Total Work
Best Case	Evenly balanced	log n	O(n)	**O(n log n)**
Worst Case	Extremely unbalanced	n	O(n)	**O(n²)**

🧩 Real-World Tip — How to Avoid the Worst Case

1. **Randomized Pivot Selection:**

   Choose a random element as pivot — avoids sorted-pattern traps.

2. **Median-of-Three Pivot:**

   Pick the median of the first, middle, and last elements as the pivot.

   (Statistically prevents extremely unbalanced partitions.)

3. **Hybrid Sorting:**

   Many libraries use Quick Sort + Insertion Sort hybrid (e.g., Python's Timsort uses Insertion for small partitions).

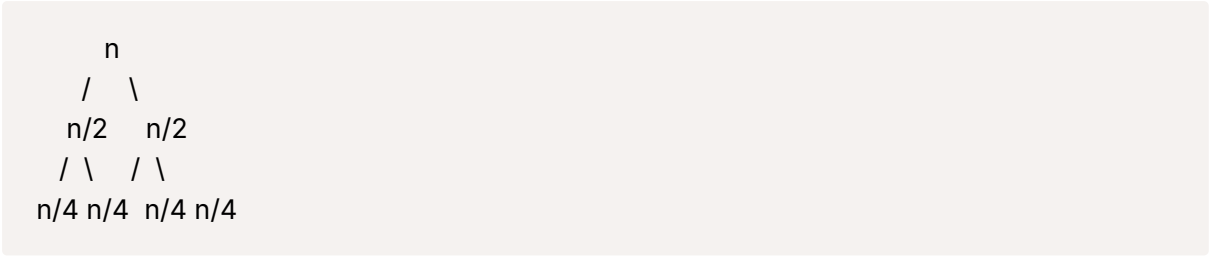
🧮 Mathematical Visualization (for Clarity)

Recurrence Relation Summary:

Case	Recurrence	Time
Best / Average	$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
Worst	$T(n) = T(n-1) + O(n)$	$O(n^2)$

📈 Recursion Tree Visualization

Best Case:



→ log n levels, total work per level = n

Total = n log n

Worst Case:

→ n levels, each with linear work → **O(n²)**

🧠 Key Takeaway

Quick Sort's efficiency **depends on pivot quality**.

If pivots divide data evenly → logarithmic depth (O(n log n))

If pivots divide badly → linear depth (O(n²))

That's why in practice, **randomized quicksort** or **median-of-three** is used to statistically guarantee near O(n log n) performance.

📌 Real-world Example

- **C / C++ std::sort()** uses a hybrid of **IntroSort** (Quick Sort + Heap Sort fallback).

  → If recursion depth exceeds a threshold (sign of worst case), it switches to Heap Sort to avoid O(n²).

✅ Final Summary

Scenario	Description	Time
Best Case	Pivot divides array evenly	O(n log n)
Average Case	Pivot divides unevenly but not worst	O(n log n)
Worst Case	Pivot always smallest/largest → one side empty	O(n²)

Would you like me to **draw recursion tree diagrams** for both the best and worst cases (balanced vs. skewed) to make this visual?

I can generate proper diagrams showing the partition depth and element splits.

⚖️ Comparison Summary

Algorithm	Best	Avg	Worst	Space	Stable	Nature	Best Use
Bubble	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅	Comparison	Teaching basics
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	❌	Selection	Low memory
Insertion	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅	Incremental	Nearly sorted
Merge	O(n log n)	O(n log n)	O(n log n)	O(n)	✅	Divide & Conquer	Predictable
Quick	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	❌	Divide & Conquer	Fastest avg