Linear Search

A simple search technique to find an element in a list (array).
It checks **each element one by one**, from the beginning, until the target is found or the list ends.

Simple but powerful!

What is Linear Search?

- Linear search is a **sequential search** algorithm.
- It starts from the **first element** and compares each element with the target value.
- If a match is found, it returns the index (position) of the element.
- If the end is reached and no match is found, it returns **-1** (or "not found").

When to Use?

- When the data is **not sorted.**
- For **small datasets** (simple and easy).
- When you need to search in a list/array (or any collection) without preprocessing.

It works on arrays, lists, strings, or any linear structure.

Example

Given an array and a target value, find the index of **7**.

	0	1	2	3	4
Array:	3	8	12	7	5

Target = **7**

Output: 3 (index of 7)

If not found, output: -1

Step-by-Step Working

1 Start from the first element.

3	8	12	7	5
0	1	2	3	4

Compare 3 with 7
(3 ≠ 7)

2 Move to the next element.

3	8	12	7	5
0	1	2	3	4

Compare 8 with 7
(8 ≠ 7)

3 Continue searching...

3	8	12	7	5
0	1	2	3	4

Compare 12 with 7
(12 ≠ 7)

4 Next element matches!

3	8	12	7	5
0	1	2	3	4

Compare 7 with 7
(7 = 7) → Found!

5 Return the index.

Index = 3

Algorithm (Pseudocode)

```
function linearSearch(arr, target):
    for i from 0 to length(arr) - 1:
        if arr[i] == target:
            return i      # index found
    return -1             # not found
```

The loop checks each element one by one until it finds the target or reaches the end.

Time & Space Complexity

Time Complexity

O(n)

(in worst case, we may check all n elements)

Space Complexity

O(1)

(uses only a few variables)

Advantages

- Simple to understand and implement.
- Works on **unsorted** data.
- Requires no extra memory.
- Good for small datasets or one-time search.

Disadvantages

- Slow for large datasets (inefficient).
- Worst case is O(n).
- Not suitable for sorted data (where **binary search** is better).

Key Takeaways

- Linear search checks each element **sequentially.**
- Time complexity: O(n)
- Space complexity: O(1)
- Returns the index if found, otherwise -1.

Simple logic...
Check one by one!

Binary Search

Binary Search is an efficient searching algorithm used to find a target value in a **sorted array** by repeatedly **dividing the search space in half.**

1. Definition & Key Points

- Works only on a sorted array (ascending or descending).
- Compares the middle element with the target value.
- If the target is smaller, search the left half.
- If the target is larger, search the right half.
- Repeats until the value is found or the search space is empty.

Time Complexity:
O(log n)
(much faster than linear search O(n) for large data)

Space Complexity:
O(1)
(iterative approach)

2. Example – Step by Step

Find 23 in the sorted array:

4	8	12	16	23	28	32	40
0	1	2	3	4	5	6	7

Step 1
low = 0, high = 7
mid = (0 + 7) / 2 = 3 → arr[3] = 16
16 < 23 → search right half

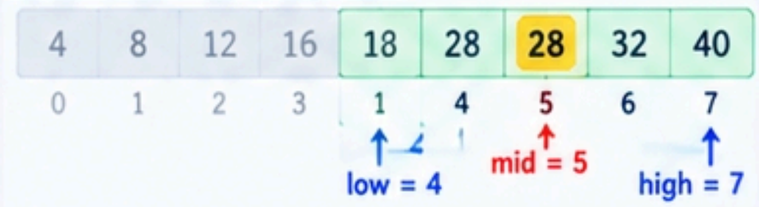
Step 2
low = 4, high = 7
mid = (4 + 7) / 2 = 5 → arr[5] = 28
28 > 23 → search left half

Step 3
low = 4, high = 4
mid = (4 + 4) / 2 = 4 → arr[4] = 23
Found!

3. Visual Representation

Step 1: Initial

Step 2: Search Right Half



Step 3: Found

4. Pseudocode

```
int binarySearch(int arr[], int n, int target) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;              // element found
        else if (arr[mid] < target)
            low = mid + 1;           // search right half
        else
            high = mid - 1;          // search left half
    }
    return -1;                       // element not found
}
```

5. Dry Run Example

Array: [3, 7, 11, 15, 19, 23, 27]
Target: 19

Iteration	low	high	mid	arr[mid]	Action
1	0	6	3	15	19 > 15 → left half
2	4	6	5	23	19 < 23 → left half
3	4	4	4	19	Found!

6. Important Characteristics

- ✅ Only works on sorted data.
- ✅ Uses divide and conquer approach.
- ✅ Reduces the search space by half each time.
- ✅ Iterative or recursive implementation.
- ✅ Efficient for large datasets.

Key Takeaway

Binary Search = Sorted Array + Divide by 2
Time Complexity = O(log n)
Space Complexity = O(1) (iterative)

7. When to Use / Not Use

✅ Use Binary Search When:

- The data is sorted.
- You need a fast search for a large dataset.
- Memory is limited (iterative version uses O(1) space).

❌ Do Not Use When:

- The data is unsorted (sort it first or use linear search).
- You are working with a linked list (random access is not available).
- The dataset is very small (linear search might be simpler).

Hashing & Hash Tables

Hashing is a technique to store and retrieve data efficiently using a **hash function**.
A **Hash Table** uses the **hash value** as an index to store the data in an array (or table).

1. What is Hashing?

- Hashing maps a large set of keys to a smaller set of values (indices) using a hash function.
- It allows fast access, insertion and deletion of data.
- It is widely used in databases, compilers, caches, and more.

Key Idea:
Use a hash function to convert a key into an integer index, called a hash value.

2. Hash Function

A hash function takes a key and produces a fixed-size integer (hash value).

Hash value = h(key) = key mod m

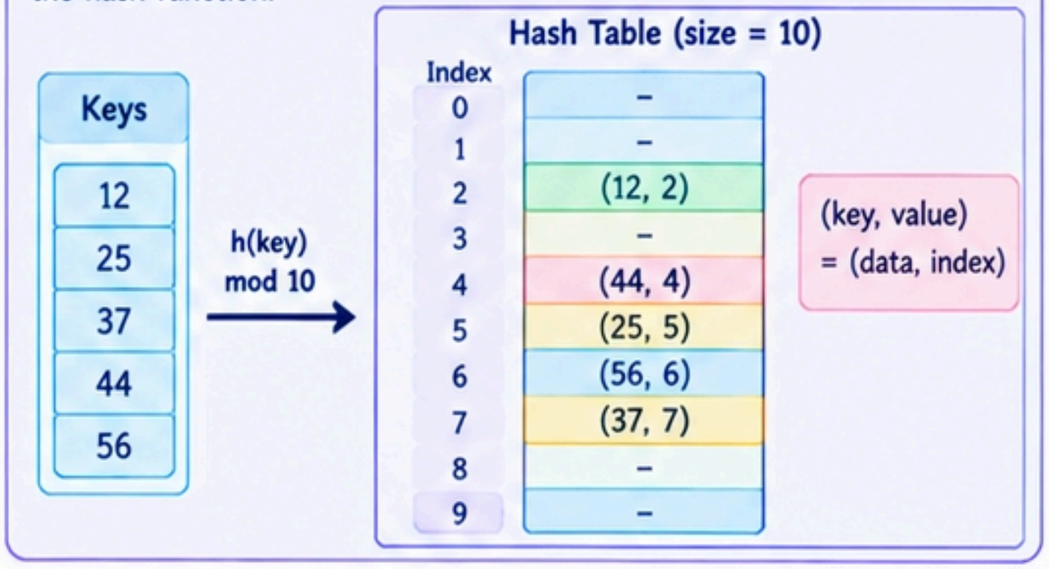
where m = size of the hash table.

Example:
Let m = 10 and h(key) = key mod 10

Key	Hash Value
12	2
25	5
37	7
44	4
56	6

3. Hash Table

A hash table is an array of buckets where each bucket stores the data (key-value pair) at the index given by the hash function.



4. Operations in Hash Table

1. Insertion

- Compute hash value.
- Go to the index.
- Store the key-value pair.
- Handle collision if needed.

Example: Insert (25, 5)
→ h(25) = 5
→ Store at index 5

2. Search

- Compute hash value.
- Go to the index.
- Check if key exists.
- If not, look in linked list (if collision).

Example: Search 37
→ h(37) = 7
→ Found at index 7

3. Deletion

- Compute hash value.
- Go to the index.
- Cind the key and remove it.
- Update (if using linked list).

Example: Delete 44
→ h(44) = 4
→ Remove from index 4

5. Collision Handling

Sometimes two different keys can have the same hash value. This is called a **collision**.

Common methods:

1. Separate Chaining

- Each index stores a linked list (or dynamic list) of key-value pairs.
- Easy to implement.

2. Open Addressing

- Find another empty slot in the table (e.g., linear or quadratic probing, double hashing).
- No extra memory needed.

6. Example with Collision

Using h(key) = key mod 7

Key	Hash Value	Index after Insertion
10	3	3
17	3	4 (collision)
24	3	5 (collision)
31	3	6 (collision)

Final Table (using linear probing)

Index	0	1	2	3	4	5	6
Value	–	–	–	10	17	24	31

7. Advantages & Disadvantages

Advantages

- Very fast search, insertion and deletion (average O(1)).
- Efficient for large datasets.
- Simple and easy to implement.
- Widely used in real-world applications.

Disadvantages

- Collisions can degrade performance (worst case O(n)).
- Requires extra memory for the table.
- Performance depends on a good hash function.
- Not suitable for ordered data.

8. Key Takeaways

- ✓ Hashing converts a key into a hash value using a hash function.
- ✓ A hash table stores data at the index given by the hash value.
- ✓ Collisions are handled using chaining or open addressing.
- ✓ Average time complexity for search, insertion and deletion is O(1).
- ✓ It is efficient, but depends on a good hash function and proper collision handling.

Hashing = Fast Access + Efficient Storage

Types of Hash Functions

A **hash function** is a function that converts a key into an **integer** (**hash value**) for storing and retrieving data in a hash table. Different types of hash functions use different techniques to compute the hash value.

1. Division (Modular) Hash Function

- Most commonly used and simplest method.
- Uses the remainder when the key is divided by a prime number (usually the size of the table).

$$h(key) = key \bmod m$$

(m = size of hash table, usually a prime number)

Example:

Table size (m) = 7

$h(25) = 25 \bmod 7 = 4$

$h(12) = 12 \bmod 7 = 5$

$h(17) = 17 \bmod 7 = 3$

Key	Index (h(key))
25	4
12	5
17	3

2. Multiplication Hash Function

- Uses multiplication with a constant value (usually $0 < A < 1$) and fractional part.
- Helps in better key distribution.

$$h(key) = \lfloor m \times (key \times A \bmod 1) \rfloor$$

(A is a constant, $0 < A < 1$, m = table size)

Example:

Let m = 10, A = 0.618

$h(25) = \lfloor 10 \times (25 \times 0.618 \bmod 1) \rfloor$

$= \lfloor 10 \times (0.5) \rfloor = 5$

$h(42) = \lfloor 10 \times (42 \times 0.618 \bmod 1) \rfloor$

$= \lfloor 10 \times (0.956) \rfloor = 9$

3. Folding Hash Function

- Divides the key into groups of equal size (folds them).
- Adds the groups (or uses XOR) to get the hash value.
- Useful for large keys (like strings or long numbers).

$$h(key) = (sum\ of\ parts) \bmod m$$

Example:

Key = 123456, Table size (m) = 10

Split into 2-digit parts: 12 + 34 + 56 = 102

$h(123456) = 102 \bmod 10 = 2$

4. Mid-Square Hash Function

- Squares the key and takes the middle digits of the result as the hash value.
- Works well when keys have patterns.

$$h(key) = middle\ digits\ of\ (key \times key)$$

Example:

Key = 52

$52^2 = 2704$

Middle 2 digits = 70

$h(52) = 70 \bmod 10 = 0$

52 → Square → 2704 → Middle digits → (70) → Hash value = 70

5. Universal Hash Function

- Uses a random function from a family of hash functions.
- Reduces the chance of collisions (especially for adversarial inputs).

$$h(key) = ((a \times key + b) \bmod p) \bmod m$$

a, b are random numbers, p is a large prime, m is table size.

Example:

Let a = 3, b = 5, p = 13, m = 7

$h(25) = ((3 \times 25 + 5) \bmod 13) \bmod 7$

$= (80 \bmod 13) \bmod 7 = 2$

6. String Hash Function

- Converts a string into an integer using character codes.
- Common methods: polynomial rolling, Horner's method, or simple summation.

$$h(key) = (\sum char[i] \times p^i) \bmod m$$

p = base (e.g., 31, 37), m = table size.

Example:

Key = "cat" (ASCII: c=99, a=97, t=116)

$h(cat) = (99 \times 31^2 + 97 \times 31 + 116) \bmod 101$

$= (95199 + 3007 + 116) \bmod 101 = 29$

7. Cryptographic Hash Function (Secure Hash)

- Uses complex algorithms (e.g., SHA-1, SHA-256).
- Produces a fixed-size hash value.
- Designed to be one-way (hard to reverse).

$$h(key) = SHA\text{-}256(key)$$ (example)

Example:

Input: "hello"

Output (SHA-256):

2cf24dba5f3390fbe3e2...

(Fixed length, 256 bits)

Uses:

- Data integrity
- Password storage
- Digital signatures

8. Application-Specific Hash Function

- Designed for specific use cases or data types.
- Examples: range hashing, consistent hashing, locale-sensitive hashing (for text), etc.

Example: Range Hashing

For keys in a range (e.g., 0–99), map to a smaller range.

$$h(key) = key \bmod 10$$ (for range 0–9)

Example:

Key = 23 → h(23) = 3

Key = 87 → h(87) = 7

Key Takeaways

- ✔ Hash functions convert a key into a hash value.
- ✔ Different types use different techniques (e.g., division, multiplication, folding, etc.).
- ✔ Good hash functions give uniform distribution and minimize collisions.
- ✔ The choice of hash function depends on the data, size, and application requirements.

Remember:

Better Hash Function = Fewer Collisions = Efficient Hash Table!

Collision, Collision Resolution

A **collision** occurs when two or more different keys produce the **same hash value** (same index).
Collision resolution techniques are used to handle this situation and store all the data correctly.

1. What is Collision?

- When two or more keys map to the same hash value (index) in a hash table, it is called a **collision**.
- It happens because the hash function cannot produce a unique index for every key.

Example:

Consider hash function: h(k) = k mod 10
Keys: 12, 22, 32

Key	h(k) = k mod 10
12	2
22	2 ← Collision
32	2 ← Collision

All three keys map to index 2, so collisions occur.

2. Collision Resolution Techniques

When a collision occurs, we use different methods to store the data. The two common methods are:

1. **Open Addressing** (store all elements in the table)
2. **Chaining** (use linked lists at each index)

Common Open Addressing Methods:

- Linear Probing
- Quadratic Probing
- Double Hashing

3. Open Addressing

- All elements are stored inside the hash table itself.
- If a collision occurs, find the next available slot using a probing technique.
- Common methods: Linear Probing, Quadratic Probing, Double Hashing.

Example: Linear Probing

Hash function: h(k) = k mod 10
Keys inserted: 12, 22, 32, 42

All these keys give index 2, so we use **linear probing** to find the next empty slot.

Index	0	1	2	3	4	5	6	7	8	9
Value	-	-	12	22	32	42	-	-	-	-

12 at 2 · 22 2 at 3 · 32 at 4 · 42 at 5

4. Open Addressing – Other Methods

1. Linear Probing

- Check next index (i+1).
- If full, continue to (i+2), (i+3), ...

Example:

h(k) = k mod 7, keys: 10, 17, 24

Index	0	1	2	3	4	5	6
Value	-	-	10	17	24	-	-

(collisions resolved by next index)

2. Quadratic Probing

- Check index using square term.
- i, $i+1^2$, $i+2^2$, $i+3^2$, ...

Example:

h(k) = k mod 7, keys: 10, 17, 24

Index	0	1	2	3	4	5	6
Value	-	-	10	17	24	-	-

Probes: 2, 3, 6, 4, 0, ...

3. Double Hashing

- Use a second hash function to find the next index.
  Helps reduce clustering.

Example:

h1(k) = k mod 7, h2(k) = 1 + (k mod 5)
Keys: 10, 17, 24

Index	0	1	2	3	4	5	6
Value	-	-	10	17	24	-	-

Next index = (h1 + i × h2) mod 7

5. Chaining

- Each index in the table stores a linked list (chain) of keys that hash to the same index.
- No need to find the next empty slot.
- Works well when the load factor is high (more collisions).

Example: Hash function: h(k) = k mod 7
Keys: 10, 17, 24, 31

Hash Table

Index	
0	
1	→ 31
2	
3	→ 10 → 17 → 24
4	
5	
6	

Here, 10, 17, 24 hash to index 3.

31 hashes to index 2.

Each index stores a linked list (chain).

6. Comparison: Open Addressing vs Chaining

Feature	Open Addressing	Chaining
Storage	All elements in table	Uses linked lists
Handling Collision	Probing (linear, quadratic, double)	Linked list at same index
Extra Memory	No extra memory	Needs pointer/linked list
Performance	Faster (good cache locality)	Slightly slower (pointer overhead)
Load Factor	Works well at low–medium	Works well even at high
Deletion	Tricky (needs reorganization)	Easy (remove from list)

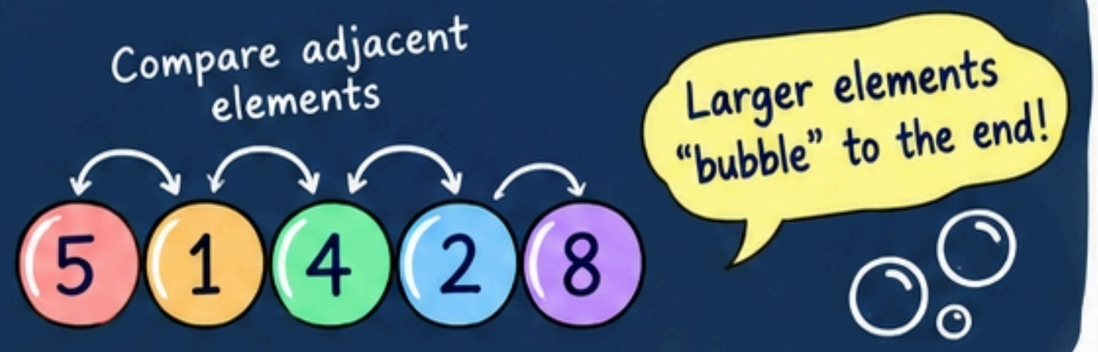
Key Takeaways

- ✅ Collision happens when two or more keys map to the same index.
- ✅ Open addressing stores all elements inside the table (uses probing).
- ✅ Chaining uses linked lists at each index.
- ✅ Linear probing, quadratic probing and double hashing are common open addressing methods.
- ✅ Chaining works well for high load factors.
- ✅ Choice depends on memory, expected load factor and performance needs.

Same Hash Value ≠ Same Location
Collision Resolution Makes It Possible !

Bubble Sort
in Data Structures

A simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order.



What is Bubble Sort?

- Bubble Sort is a comparison-based sorting algorithm.
- It works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order.
- After each pass, the largest element "bubbles" to the end of the array.
- It is simple to understand and implement, but not efficient for large datasets.

Pseudocode / C-like Code

```
void bubbleSort(int arr[], int n) {
   int i, j, temp;
   for (i = 0; i < n - 1; i++) {
      for (j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}
```

Time & Space Complexity

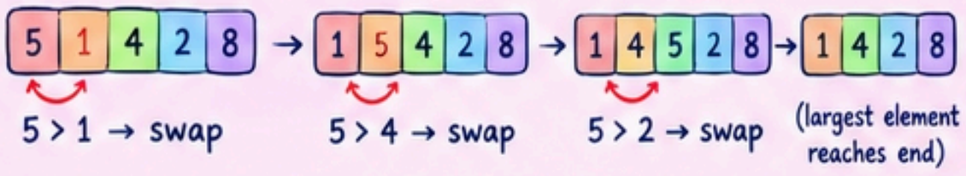
Time Complexity:
- Best Case : O(n) (already sorted)
- Average Case: $O(n^2)$
- Worst Case : $O(n^2)$

Space Complexity: O(1) (in-place)

How It Works (Step by Step)

Let's sort the array: [5, 1, 4, 2, 8]

Pass 1 (compare 0-1, 1-2, 2-3, 3-4)



Pass 2 (compare 0-1, 1-2, 2-3)

Pass 3 (compare 0-1, 1-2)

Pass 4 (compare 0-1)

After all passes, the array is sorted!

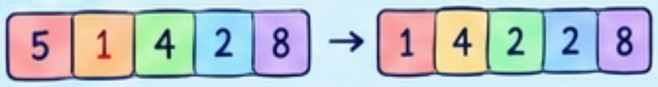
Key Characteristics

- ✔ Stable sorting algorithm (maintains order of equal elements).
- ✔ In-place (no extra memory required).
- ✔ Simple to implement.
- ✔ Not efficient for large datasets.

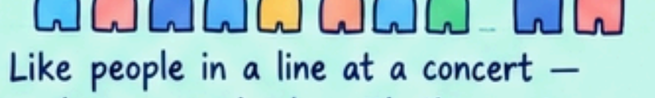
Example (Full Passes)

Array: [5, 1, 4, 2, 8]

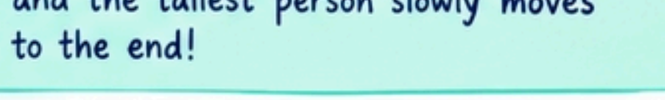
Pass 1:



Pass 2:



Pass 3:



Pass 4:

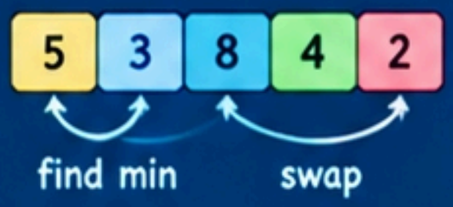
When to Use?

- ✔ Good for small or nearly sorted data.
- ✔ Useful for educational purposes and simple problems.
- ✔ Not recommended for large or performance-critical applications.

Real-Life Analogy

Like people in a line at a concert — each person checks with the next one, and the tallest person slowly moves to the end!

Bubble Sort = Simple Idea + Repeated Comparisons + Swaps = Sorted Array



Selection Sort

Selection Sort is a simple comparison-based sorting algorithm that works by finding the **minimum** (or **maximum**) **element** from the **unsorted** part and **swapping** it with the first element of the **unsorted part**.

- Simple to understand
- In-place (no extra memory)
- Not stable

1. Definition & Key Points

- Divides the array into two parts:
  - Sorted part (left side)
  - Unsorted part (right side)
- Repeatedly finds the smallest element from the unsorted part and swaps it with the first element of the unsorted part.
- Continues until the entire array is sorted.

Key Idea:

Select the minimum element and move it to the beginning of the unsorted part.

2. How It Works

1. Find the minimum element in the unsorted part (from current position to end).
2. Swap it with the first element of the unsorted part.
3. Move the boundary of the sorted part one position to the right.
4. Repeat steps 1–3 until the array is sorted.

3. Step-by-Step Example

Array: [64, 25, 12, 22, 11]

Pass 1: Find min (11) and swap with first element (64).

11	25	12	22	**64**	→ 11 is the minimum

Pass 2: Find min (12) and swap with second element (25).

11	**12**	25	22	64	→ 12 is the minimum

Pass 3: Find min (22) and swap with third element (25).

11	12	**22**	25	64	→ 22 is the minimum

Pass 4: Find min (64) (already in place).

11	12	22	25	**64**	– Sorted!

4. Pseudocode

```
for i = 0 to n - 2
    minIndex = i
    for j = i + 1 to n - 1
        if arr[j] < arr[minIndex]
            minIndex = j
    swap(arr[i], arr[minIndex])
```

n = size of array — minIndex = index of minimum
i = current position — j = scanning index

5. C Code Example

```c
void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

6. Trace Table (for example array)

Pass	Array After Pass	Min Element	Swap
Initial	64 25 12 22 11	–	–
1	[11 25 12 22 64]	11	64 ↔ 11
2	[11 12 25 22 64]	12	26 ↔ 12
3	[11 12 22 25 64]	22	25 ↔ 22
4	[11 12 22 25 64]	64	(no swap)

7. Complexity Analysis

Aspect	Time / Space
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$
Space Complexity	O(1) (in-place)

8. Advantages & Disadvantages

Advantages

- Simple and easy to implement.
- Requires only O(1) extra space.
- Works well for small datasets.
- In-place sorting (no extra memory).

Disadvantages

- Slow performance ($O(n^2)$).
- Not suitable for large datasets.
- Performs many swaps (even when not required).
- Not stable (equal elements may change order).

9. Key Takeaway

- ✔ Selection Sort = Find minimum + Swap.
- ✔ Time Complexity = $O(n^2)$.
- ✔ Space Complexity = O(1).
- ✔ Simple but inefficient for large data.
- ✔ Good for small datasets or learning basic sorting concepts.

Selection Sort is simple, but not the fastest!

Build a sorted portion from left to right

Insertion Sort

Insertion Sort is a simple comparison-based sorting algorithm that builds the sorted array one element at a time by inserting each element into its correct position in the already sorted portion.

- Simple to understand
- In-place (no extra memory)
- Stable (maintains relative order)
- Good for small datasets

1. Definition & Key Points

- Insertion Sort divides the array into two parts:
  - Sorted portion (left side)
  - Unsorted portion (right side)
- Picks the next element from the unsorted portion and inserts it into its correct position in the sorted portion.
- Repeats until the entire array is sorted.

Key Idea:
Build a sorted array from left to right by inserting each element into its correct position.

2. How It Works

1. Start with the first element as a sorted portion (size = 1).
2. Pick the next element from the unsorted portion.
3. Compare it with elements in the sorted portion (from right to left) and shift larger elements to the right.
4. Insert the element at its correct position.

3. Step-by-Step Example

Sort the array: [5, 2, 4, 6, 1]

Step 1: Start with first element (5) – already sorted.

Sorted: [5]
Unsorted: [2, 4, 6, 1]

Step 2: Insert 2 into the sorted part.

Sorted: [2, 5]
Unsorted: [4, 6, 1]

Step 3: Insert 4 into the sorted part.

Sorted: [2, 4, 5]
Unsorted: [6, 1]

Step 4: Insert 6 into the sorted part (already in place).

Sorted: [2, 4, 5, 6]
Unsorted: [1]

Step 5: Insert 1 into the sorted part.

Sorted: [1, 2, 4, 5, 6]
Unsorted: []

4. Pseudocode

```
void insertionSort(int arr[], int n) {
  for (int i = 1; i < n; i++) {
     int key = arr[i];          // current element
     int j = i - 1;             // last index of sorted part

     while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];    // shift element right
        j--;                    // move to previous element
     }
     arr[j + 1] = key;          // insert key at correct position
  }
}
```

i = current element index j = index for comparison
key = element to be inserted

5. Example with Array Tracing

Array: [9, 7, 5, 3, 1]

Pass 1: Insert 7

(7 < 9, insert at 0)

Pass 2: Insert 5

(5 < 7, 5 < 9, insert at 0)

Pass 3: Insert 3

(3 < 5, 3 < 7, 3 < 9, insert at 0)

Pass 4: Insert 1

(1 < 3, 1 < 5, 1 < 7, 1 < 9, insert at 0)

6. Complexity Analysis

Aspect	Time / Space
Best Case (Already sorted)	O(n) / O(1)
Average Case	$O(n^2)$ / O(1)
Worst Case (Reverse sorted)	$O(n^2)$ / O(1)
Space Complexity	O(1) (in-place)
Stability	Stable

7. Advantages & Disadvantages

Advantages

- Simple and easy to implement.
- Works well for small datasets.
- In-place sorting (no extra memory).
- Stable (maintains order of equal elements).

Disadvantages

- Slow for large datasets ($O(n^2)$).
- Requires more comparisons and shifts when the array is reverse sorted.
- Not efficient for large data.

Key Takeaways

- Insertion sort builds a sorted array from left to right.
- It is simple, in-place and stable.
- Time complexity: $O(n^2)$ (average & worst), O(n) (best).
- Best suited for small or nearly sorted datasets.

Simple Idea → Insert at Correct Position → Sorted Array

Quick Sort

Quick Sort is a divide and conquer sorting algorithm. It works by selecting a **pivot** element, partitioning the array into **smaller subarrays** (elements less than and greater than the pivot), and then **recursively** sorting the **subarrays**.

- In-place (no extra memory for arrays)
- Not stable
- Good for large datasets

1. What is Quick Sort?

- Quick Sort is a comparison-based sorting algorithm.
- It selects a pivot, partitions the array into two parts (less than and greater than pivot), and recursively sorts the parts.

Key Idea:

Choose a pivot, partition, and recursively sort the subarrays.

2. How It Works

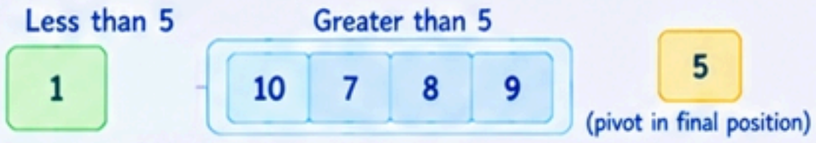
1. Choose a pivot (e.g., last element, first element, or random).
2. Partition the array so that elements less than pivot are on the left and greater than pivot are on the right.
3. Recursively sort the left and right subarrays.
4. Combine (implicit, since it's in-place).

3. Step-by-Step Example

Sort the array: [10, 7, 8, 9, 1, 5]

Step 1: Choose pivot (last element = 5).

Step 2: Partition the array.



Step 3: Recursively sort left and right parts.

Left: [1] (already sorted)

Right: [10 7 8 9] → sort → 9 8 7 10

4. Partition Process

Using last element as pivot (5):

Move elements < 5 to left, > 5 to right.

Result after partition:

- **i** = index for smaller element
- **j** = index for current element
- **pivot** = last element

5. Code Example (C/C++/Java style)

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;   // pivot index
}
  void quickSort(int arr[], int low, int high) {
      if (low < high) {
          int pi = partition(arr, low, high);
          quickSort(arr, low, pi - 1);
          quickSort(arr, pi + 1, high);
      }
  }
}
```

6. Full Example - Multiple Steps

Array: [10, 7, 8, 9, 1, 5]

1. Partition (pivot = 5)

   [1 | 10 8 9 7 5]

2. Sort right part [10 8 9 7] (pivot = 7)

   [1 | 7 | 8 9 10 | 5]

3. Sort [8 9 10] (pivot = 10)

   [1 | 7 | 8 9 10 | 5]

4. Sort [8 9] (pivot = 9)

   [1 | 7 | 8 9 | 10 | 5]

Sorted Array: 1, 5, 7, 8, 9, 10

7. Complexity Analysis

Case	Time Complexity	Space Complexity
Best Case	O(n log n)	O(log n)
Average Case	O(n log n)	O(log n)
Worst Case (When pivot is worst)	$O(n^2)$	O(log n)
Space (in-place)	-	O(log n)

8. Advantages & Disadvantages

Advantages

- Fast average performance (O(n log n).
- In-place (no extra memory).
- Good for large datasets.
- Cache friendly (good locality).

Disadvantages

- Worst case $O(n^2)$ (e.g., already sorted array with poor pivot).
- Not stable (equal elements may change order).
- Performance depends on pivot selection.

9. Key Takeaways

- ✔ Choose a good pivot (random or median-of-three) to avoid worst-case.
- ✔ Partition the array and recursively sort subarrays.
- ✔ Average time complexity is O(n log n).
- ✔ Space complexity is O(log n) (due to recursion stack).
- ✔ Quick Sort is fast, simple and widely used in practice.

Quick Sort = Partition + Recursion + Sort

Comparative Study of Various Searching and Sorting Algorithms in Data Structure

Different algorithms solve the same problem in different ways. Here's a side-by-side comparison of popular searching and sorting algorithms, based on their working principle, time & space complexity, and key characteristics.

Searching Algorithms

n = number of elements | k = key to be searched | log n = logarithmic time

1. Linear Search

- Works by checking each element one by one.
- Works on unsorted (and sorted) data.

Time Complexity	Space Complexity
Best: O(1) Average: O(n) Worst: O(n)	O(1) (only a few variables)

Key Use Small datasets or unsorted data.

2. Binary Search

- Works on **sorted** data.
- Compares the middle element with the key and halves the search space.

Time Complexity	Space Complexity
Best: O(1) Average: O(log n) Worst: O(log n)	O(1) (iterative) O(log n) (recursive)

Key Use Large sorted datasets.

3. Jump Search

- Works on sorted data.
- Skips blocks of size $\sqrt{n}$ and then does linear search in the block.

Time Complexity	Space Complexity
Best: O(1) Average: O(√n) Worst: O(√n)	O(1)

Key Use Sorted data, large datasets.

4. Interpolation Search

- Works on sorted and uniformly distributed data.
- Estimates the position of the key using interpolation.

Time Complexity	Space Complexity
Best: O(1) Average: O(log log n) Worst: O(n)	O(1)

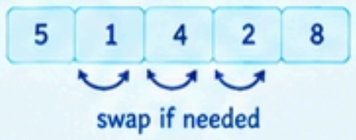
Key Use Uniformly distributed data.

Sorting Algorithms

n = number of elements | k = key to be sorted

1. Bubble Sort

- Repeatedly swaps adjacent elements if they are in the wrong order.
- Largest element "bubbles" to the end.

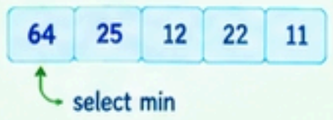


Time Complexity	Space Complexity
Best: O(1) Average: O(n²) Worst: O(n²)	O(1)

Key Use Small datasets, simple implementation.

2. Selection Sort

- Finds the minimum element and places it at the beginning.
- Repeats for the remaining elements.

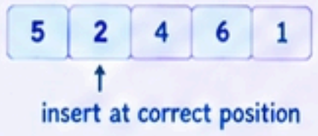


Time Complexity	Space Complexity
Best: O(n²) Average: O(n²) Worst: O(n²)	O(1)

Key Use Small datasets, in-place sort.

3. Insertion Sort

- Builds a sorted portion from left to right.
- Inserts each element into its correct position.



Time Complexity	Space Complexity
Best: O(n) Average: O(n²) Worst: O(n²)	O(1)

Key Use Small or nearly sorted data.

4. Merge Sort

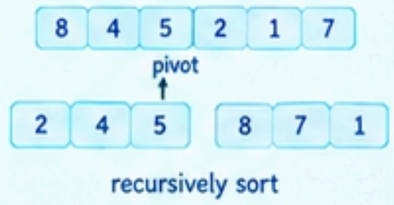
- Divides the array into halves, sorts them and merges.
- Uses divide and conquer strategy.

Time Complexity	Space Complexity
Best: O(n log n) Average: O(n log n) Worst: O(n log n)	O(n) (extra array)

Key Use Large datasets, stable sort.

5. Quick Sort

- Picks a pivot, partitions the array and sorts subarrays recursively.
- Uses divide and conquer.

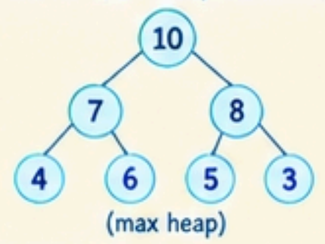


Time Complexity	Space Complexity
Best: O(n log n) Average: O(n log n) Worst: O(n²)	O(log n) (in-place)

Key Use Large datasets, general purpose.

6. Heap Sort

- Builds a heap from the array.
- Repeatedly extracts the max (or min) and heapifies.



Time Complexity	Space Complexity
Best: O(n log n) Average: O(n log n) Worst: O(n log n)	O(1) (in-place)

Key Use Large datasets, in-place sort.

Quick Comparison (at a glance)

Algorithm	Type	Time Complexity (Best / Avg / Worst)	Space Complexity	Stable?
Linear Search	Searching	O(1) / O(n) / O(n)	O(1)	Yes
Binary Search	Searching	O(1) / O(log n) / O(log n)	O(1)	Yes
Jump Search	Searching	O(1) / O(√n) / O(√n)	O(1)	Yes
Interpolation Search	Searching	O(1) / O(log log n) / O(n)	O(1)	Yes
Bubble Sort	Sorting	O(n³) / O(n²) / O(n²)	O(1)	Yes
Selection Sort	Sorting	O(n²) / O(n²) / O(n²)	O(1)	No
Insertion Sort	Sorting	O(n) / O(n²) / O(n²)	O(1)	Yes
Merge Sort	Sorting	O(n log n) / O(n log n) / O(n log n)	O(n)	Yes
Quick Sort	Sorting	O(n log n) / O(n log n) / O(n²)	O(log n)	No
Heap Sort	Sorting	O(n log n) / O(n log n) / O(n log n)	O(1)	No

Key Takeaways

- **Searching algorithms** help us find an element in a collection.
- **Sorting algorithms** arrange elements in a specific order (ascending or descending).
- Binary search is much faster than linear search on sorted data.
- **Merge sort** and **heap sort** are **stable** (merge sort) and in-place (heap sort) with good time complexity.
- Quick sort is generally the fastest in practice but has a worst-case of O(n²).
- Choice of algorithm depends on data size, order, memory and stability requirements.

Remember

Searching → Find an element
(Linear / Binary / Jump / Interpolation)

Sorting → Arrange elements
(Bubble / Selection / Insertion / Merge / Quick / Heap)

Right Algorithm + Right Data = Better Performance!