Algorithm	Best Case	Average Case	Worst Case	Time Complexity	Space Complexity	Number of Swaps	Stability	In-Place	Typical Uses
Counting Sort	Ω(n + k)	θ(n + k)	O(n + k)	O(n + k)	O(k)	0	Stable	No	Large datasets with a limited range of integer values
Insertion Sort	Ω(n)	θ(n²)	O(n²)	O(n²)	O(1)	0 to O(n²)	Stable	Yes	Small or nearly sorted dataset, adaptive sorting
Bubble Sort	Ω(n)	θ(n²)	O(n²)	O(n²)	O(1)	0 to O(n²)	Stable (only if comparison is <, not ≤)	Yes	Educational purposes, small datasets (inefficient for large)
Selection Sort	Ω(n²)	θ(n²)	O(n²)	O(n²)	O(1)	O(n)	Unstable (stable only with extra space)	Yes	Small datasets, when minimizing swaps is a priority
Merge Sort	Ω(n log n)	θ(n log n)	O(n log n)	O(n log n)	O(n)	0	Stable	No	Sorting large data sets and linked lists
Quick Sort	Ω(n log n)	θ(n log n)	O(n²)	O(n²)	O(log n)	-	Unstable	Yes	General-purpose efficient sorting algorithm
Heap Sort	Ω(n log n)	θ(n log n)	O(n log n)	O(n log n)	O(1)	-	Unstable	Yes	Efficient for large data sets
Binary Search	Ω(1)	θ(log n)	O(log n)	O(log n)	O(1)	-	-	-	Array of known bounded size *n*
Exponential Search	Ω(1)	θ(log n)	O(log n)	O(log n)	O(1)	-	-	-	Very large or infinite arrays

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	θ(1)	θ(n)	θ(n)	θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	θ(1)	θ(1)	θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	θ(log(n))	θ(log(n))	θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	θ(log(n))	θ(log(n))	θ(log(n))	θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

BSP Trees

Binary space partitioning

Advantages

- Once the tree is computed, the tree can handle all viewpoints without reconstructing the tree, i.e. efficient
- Handle transparency
- it's a standard format (.BSP files) to store the environment for many games
- E.g. Quake, Half-life, Call of Duty, etc

Disadvantages

- Cannot handle moving/changing environments
- Preprocessing time for tree construction is long

Renders the polygon that is behind respective to the viewer

$for\ height\ h,$
$max\ nodes\ = 2^{h+1} - 1$
$min\ nodes\ (exact)\ = N(h-1) + N(h-2) + 1$
$N(0) = 1,\ N(1) = 2$
$min\ nodes\ (lower\ bound)\ > 2^{h/2}$

$for\ n\ nodes,$
$max\ height\ = 1.44log_2 n\ or\ 2log_2 n$
$min\ height\ = floor(log_2 n)$

Height 0: 1
Height 1: 2
Height 2: 4
Height 3: 7
Height 4: 12
Height 5: 20
Height 6: 33
Height 7: 54
Height 8: 88
Height 9: 143

AVL Tree time complexities

Function	Insertion	Deletion
Time complexity	O(log n)	O(log n)
Number of rotations	max 2 = O(1)	O(log n)
Method	1. Search 2. Insert 3. Rotate max 2 times if unbalanced	1. Search 2. Find successor to replace 3. Delete recursively to account for left child node of successor 4. Balance max 2 times per recursion and return root recursively

Hash Tables

Key features

- No duplicate keys
- No mutable keys
- If key is database, it cannot be changed

Performance of Open Addressing

Define:

- Load: α = n / m
- Assume α < 1.

Claim:

- For n items, in a table of size m, assuming uniform hashing, the expected cost of an operation is: 1/1-α

If there are n items in a hash table with size m, the expected number of probe for open addressing is m/(m-n)

Simple Uniform Hashing Assumption (chaining)

- Every key is equally likely to map to every bucket
- Keys are mapped independently

Define: load(hash table) = n/m = average # items / buckets
Expected search time = 1 + n/m

If m > n,
Expected search time = O(1)

Types of hashing	Open Addressing (closed hashing)	Closed Addressing (open hashing)
Collisions	• Collisions are dealt with by searching for another empty bucket within the hash table	• Key is always stored in the bucket it is hashed to • Collisions are dealt with using separate data structures on a per bucket basis (eg. linked list, BST)
Key to bucket ratio	At most one key per bucket	Arbitrary number of keys per bucket
Examples	• Linear probing • Quadratic probing • Double hashing • Hopscotch hashing • Cuckoo hashing • 2-choice hashing	• Separate chaining using linked lists • Separate chaining using dynamic arrays • Using AVL trees
Load factor α (number of keys / number of buckets)	Maximum of 1	No maximum
Size	Size of hash table must be at least as large as the number of keys in the hash table	Performance worsens as load factor grows
Advantages	• No size overhead apart from hash table • Better memory locality and cache performance • Performs better than closed addressing when number of keys is known in advance	• Easier removal - no need to mark buckets as deleted • Performs better with high load factor • No issues with clustering
Disadvantages	• More sensitive to choice of hash functions (eg. clustering in linear probing) • More sensitive to load	• Linked lists can wander all over the memory • Does not save space

Heaps (binary/min-max heap)

Properties

1. Heap Ordering
   - priority[parent] ≥ priority[child]
2. Complete Binary Tree
   - Every level is full, except possibly the last
   - All nodes are as far left as possible

Time complexities

- Insertion = O(log(n))
- Deletion = O(log(n))
- Increase/decrease key = O(log(n))

Heap vs AVL Tree

- same cost for operations
- slightly simpler: no rotations
- slightly better concurrency
- can store tree in an array

Heap Sort

Unsorted list → Heap: O(n)

Heap array → Sorted list: O(n log n)

Union Find

Type	Find	Union
Quick Find	O(1)	O(n)
Quick Union	O(n)	O(n)
Weighted Union	O(log n)	O(log n)
Weighed union + path compression	α(m, n)	α(m, n)

Pros and cons of Heap Sort

Advantages	Disadvantages
Consistent Time Complexity • O(n log n) in best, average, and worst cases	**Unstable Sorting** • Does not maintain the relative order of equal elements
In-Place Sorting • Requires only a constant amount of additional memory (O(1) space)	**More Complex Implementation** • More difficult to implement compared to simpler sorting algorithms like insertion and selection
Not Recursive • Can be implemented iteratively, avoiding stack overflow risks in recursive algorithms	**Less Cache-Friendly** • Access patterns lead to inefficient CPU cache usage as compared to quick sort
Efficient for Large Data Sets • Handles large arrays efficiently	**Not Adaptive** • Performance doesn't improve with partially sorted arrays

Graph terms

Degree of a graph: Maximum number of adjacent edges

Diameter: Maximum distance between two nodes, following the shortest path

Clique (Complete Graph) : Diameter = 1, degree = n-1

Star : Diameter = 2, degree = 1

Cycle : Diameter = n/2 or diameter = n/2-1, degree = 2

Adj list vs matrix

Adjacency List	Adjacency Matrix
Memory usage for graph G = (V, E): array of size |V| linked lists of size |E| Total: **O(V + E)** Cycle: E = **O(V)** Clique: O(V+E) = **O(V^2)**	Memory usage for graph G = (V, E): array of size |V|*|V| Total: **O(V^2)** Cycle: **O(V^2)** Clique: **O(V^2)**
Base rule: if graph is dense then use an adjacency matrix; else use an adjacency list.	

Graph

Bellman Ford
- **Best Case:** O(E), when distance array after 1st and 2nd relaxation are same, we can simply stop further processing.
- **Average Case:** O(V*E)
- **Worst Case:** O(V*E)

Dijkstra
- insert / deleteMin: |V| times each
  - Each node is added to the priority queue once.
- decreaseKey: |E| times
  - Each edge is relaxed once
- Priority queue operations: **O(log V)**
- Total: O((V+E)log V) = **O(E log V)**

BFS (Queue) / DFS (Stack)
- **O(V + E)**

PQ Implementation	insert	deleteMin	decreaseKey	Total
Array	1	V	1	**$O(V^2)$**
AVL Tree	log V	log V	log V	**O(E log V)**
d-way Heap	$d\log_d V$	$d\log_d V$	$\log_d V$	**$O(E\log_{E/V} V)$**
Fibonacci Heap	1	log V	1	**O(E + V log V)**

Min Spanning Tree

Algorithm	Analysis
Prim's	• Each vertex added/removed once from the priority queue: **O(V log V)** • Each edge => one decreaseKey: **O(E log V)**
Kruskal's	Sorting: O(E log E) = **O(E log V)** For E edges: Find: **O(α)** or **O(log V)** Union: **O(α)** or **O(log V)**

Boruvka's	Initially: Create n components, one for each node in the graph
	At each step:
	1. Assume k components initially
	2. For each connected component, search for the minimum weight outgoing edge
	  ○ DFS or BFS: **O(V + E)**
	  ○ Check if edge connects two components.
	  ○ Remember minimum cost edge connected to each component.
	3. At least k/2 edges added
	4. At least k/2 edges merged
	5. At most k/2 components remain
	Termination: 1 component
	Conclusion: At most **O(log V)** Boruvka steps.
	Total time: O((E+V)log V) = **O(E log V)**

MST edge cases

all the edges have the same weight:
- Depth-First-Search or Breadth-First-Search
- An MST contains exactly (V-1) edges
- Every spanning tree contains (V-1) edges
- Thus, any spanning tree you find with DFS/BFS is a minimum spanning

Prim: What if all the edges have weights from {1..10}?

Idea: Use an array of size 10
– Inserting/Removing nodes from PQ: O(V)
– decreaseKey: O(E)

Total: O(V + E) = O(E)

Kruskal: What if all the edges have weights from {1..10}?

Idea: Use an array of size 10
– Putting edges in array of linked lists: O(E)
– Iterating over all edges in ascending order: O(E)
– Checking whether to add an edge: O(α(V))
– Union two components: O(α(V))

Total: **O(α(V)E)**

Topological Sort

DFS: O(log(V+E))

Planar Graphs

$V - E + F = 1 + C$

The average degree of a node in a planar graph is less than 6

1D Range Search

Range List Query
- Given two numbers a < b, list out all the elements x such that a <= x <= b
- **O(log n + k)**
- k is the number of output elements

Range Count
- Given two numbers a < b, count all the elements x such that a <= x <= b
- rank (b) – rank(a) + 1
- If a or b are not in BST, use their successor or predecessor respectively

2D Range Search

Query time: **O(log^2 n + k)**

Time:
- O(log n) to find starting/ending node in x
- For each node in x, O(log n) y-tree searches of cost O(log n)
- O(k) enumerating output

Space: **O(n log n)**
Insertion, deletion, rotation may be O(n)

Convex Hull

Jarvis March	Graham Scan
Take the leftmost vertex Repeat • Search for the next vertex on the convex hull by choosing the one with the minimal turning angle Time complexity • O(hn) • h is number of points in hull • n is number of vertices	• Take the leftmost vertex • Sort the rest according to their angles • Connect them in that order and form a polygon • Starting from the leftmost vertex, go around the polygon • If it is a concave vertex, "make it convex" by "filling" it by connecting its two neighbors Time complexity • O(nlogn)

Properties
1. No cycles
2. If you cut an MST, the two pieces are both MSTs
3. Cycle property
   - For every cycle, the maximum weight edge is **not** in the MST
   - For every cycle, the minimum weight edge **may or may not** be in the MST
4. Cut property
   - For every partition of the nodes, the minimum weight edge across the cut is in the MST
   - For every vertex, the minimum outgoing edge is **always** part of the MST
   - For every vertex, the maximum outgoing edge **may or may not** be part of the MST

Applications

- Error correcting codes
- Face verification
- Cluster analysis
- Image registration

Advantages of Friend Functions
- A friend function is able to access members without the need of inheriting the class.
- The friend function acts as a bridge between two classes by accessing their private data.
- It can be used to increase the versatility of overloaded operators.
- It can be declared either in the public or private or protected part of the class.

Disadvantages of Friend Functions
- Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
- Friend functions cannot do any run-time polymorphism in their members.

Important Points About Friend Functions and Classes
1. Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data access lessens the value of encapsulation of separate classes in object-oriented programming.
2. Friendship is **not mutual**. If class A is a friend of B, then B doesn't become a friend of A automatically.
3. Friendship is not inherited

Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:
- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be the same in the base as well as the derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have a virtual destructor but it cannot have a virtual constructor.

Characteristics of Constructors
- The name of the constructor is the same as its class name.
- Constructors are mostly declared as public member of the class though they can be declared as private.
- Constructors do not return values, hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Multiple constructors can be declared in a single class (it is even recommended – Rule Of Three, The Rule of Five).
- In case of multiple constructors, the one with matching function signature will be called.

Characteristics of a Destructor
- A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence, destructor cannot be overloaded.
- It cannot be declared static or const.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

Polymorphism
- "Many forms" — same interface, different behavior
- Furniture can be a table or chair

Overloading (Compile-time Polymorphism)
- Same function/operator name, different parameters

Encapsulation
- Bundles data + methods; restricts direct access
- Achieved using **private**, **public**, **protected**

Inheritance
- Reuse code from base class
- Coffee table and dining table can both inherit properties of a table

Abstraction
- Shows essential features, hides internal details
- Achieved using **abstract classes (with pure virtual functions)**

Skip List

Each search takes **O(log n)** steps in expectation.

Advantages:
- The skip list is solid and trustworthy.
- To add a new node to it, it will be inserted extremely quickly.
- Easy to implement compared to the hash table and binary search tree
- The number of nodes in the skip list increases, and the possibility of the worst-case decreases
- Requires only **O(logn)** time in the average case for all operations.
- Finding a node in the list is relatively straightforward.

Disadvantages:
- It needs a greater amount of memory than the balanced tree.
- Reverse search is not permitted.
- Searching is slower than a linked list
- Skip lists are not cache-friendly because they don't optimize the locality of reference