# Asymptotic Notation **–––––––––––––––––––––––––––––––––––––––––––––––––––––––**

|  |  |  |
| --- | --- | --- |
| **Meaning** | **Rules** | |
|  | Irrelevance of factors  Transitivity  Sum Rule  Product Rule |  |
| Limit Rule |  |

# Sorting Algorithms **–****–––––––––––––––––––––––––––––––––––––––––––––––––––––––**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Insertion Sort** | | | | **Selection Sort** | | | | **Quick Sort**  A screenshot of a computer code  AI-generated content may be incorrect. |
| **Merge Sort** | | | | | | | | |
| A diagram of a diagram  AI-generated content may be incorrect. | | | | | | | | |
| **Sorting Algorithm Efficiency** | | | | | | | | |
| **Algorithm** | **Best Time** | **Average Time** | **Worst Time** | | **Is Stable** | **In Place** | **Memory Space** | |
| Insertion Sort | n | n2 | n2 | | Yes | Yes | constant | |
| Quick Sort | n log n | n log n | n2 | | No | Almost | log n → n | |
| Selection Sort | n2 | n2 | n2 | | No | Yes | constant | |
| Merge Sort | n log n | n log n | n log n | | Yes | No | n | |
| Heap Sort | n log n | n log n | n log n | | No | Yes | constant | |
| **Note that:**   * An algorithms stability refers to whether it preserves the relative order of equal elements in the sorted output as they appeared in the input * Whether an algorithm is in place means it sorts the data within the original array structure, requiring only a small, constant amount of auxiliary memory space, typically O(1), beyond the input array itself. | | | | | | | | |

# Solving Recurrences **–––––––––––––––––––––––––––––––––––––––––––––––––––––––**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bottom-up Method** | Example:   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | f(n) | 0 | 1 | 4 | 9 | 16 | 25 | 36 |   This looks like f(n) = n2 – Now prove by induction |
| 1. Start with base case 2. Compute first few values 3. Find Pattern |
| **Top-down Method** | Example:    … repeat a few times  \*3rd iteration  Hmm… we can simplify the pattern as …  \* observation we reach base case when n – k = 1  – Now prove with induction |
| 1. Start with definition 2. Expand definition a few times 3. Find Pattern |
| **Change of Variable Trick** | Example: |
|  |
| **Induction Proof** |  |
| 1. Start with proving base case 2. Inductive Hypothesis 3. Inductive Step | Does our new definition match the previous?  Assume our new definition holds true for all possible inputs “k”  Prove our statement holds true for “k + 1” |

# Data Structures **–****––––––––––––––––––––––––––––––––––––––––––––––––––––––––––**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Priority Queues** | | | | | | |
| [0, 27, 43, 48, 62, 53]  root\_index = 1  left\_child = 2 \* parent\_index  right\_child = 2 \* parent\_index + 1  parent = child // 2 | A close-up of a number  AI-generated content may be incorrect. | **Insert** | **Get min / max** | **Extract min / max** | **Update** | **Build** |
| log n | constant | log n | if (!i) {  log n  } else {  n  } | n |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Binary Search Trees, AVL Trees, and Red Black Trees** | | | | | | | | |
| **Binary Search Tree** Binary Search Tree - GeeksforGeeks | **Insert** | | **AVL Tree**  A binary tree where the height of the left subtree is no more than 1 different from the height of the right subtree | **Insert** | log n | **Red Black Trees**  Red-Black Trees | SpringerLink | **Insert** | log n |
| log n | n |
| **Delete** | | **Delete** | log n | **Delete** | log n |
| log n | n |
| **Search** | | **Search** | log n | **Search** | log n |
| log n | n |

# Hash Tables **––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––**

**A Hash Table** is a data structure which stores key-value pairs. It uses a **hash function** to map keys to array indices (slots) for efficient data retrieval.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Collisions: The Inevitable Challenge** | | **Collision Resolution Strategies** | | | |
| **Why deal with collisions?** Different keys can map to the same slot, leading to data overwriting or incorrect retrieval if not handled.  **Hash Function Properties:**  **Deterministic:** The same input key must always produce the same hash output.  **Random Distribution:** Ideally, the hash function should distribute keys uniformly across the table to minimize collisions and maintain performance. | | **Linear Probing**  **Method:** If a slot is occupied, check the next slot sequentially  **Problem:** Leads to clustering, where occupied slots group together, this degrades performance, and makes the table non-uniformly distributed. | | **Double Hashing**  **Method:** Uses to hash functions. The first function is used as the initial probe. And the second is used as a step  **Benefit:** reduces clustering, and maintains random distribution | **Chaining**  Each slot in the hash table points to a linked list (or other data structure) containing all elements that hash to that slot. |
| **Terminology** | | | | | |
| **Probe** | An access to an array element in the hash table during insertion or search | | | | |
| **n** | Total number of items stored in the hash table | | | | |
| **m** | Total number of slots (size) in the hash table | | | | |
| **λ = n/m: Load Factor** | Represents the average number of items per slot. A higher load factor generally means more collisions and degraded performance. | | | | |
| **Simple Uniform Hashing** | | | **Universal Hashing** | | |
| **Assumption:** Any given key is equally likely to hash into any of the m slots, independently of where other keys hash.  **Pigeonhole Principle & Collisions:** If n > m (more items than slots), at least one slot must contain more than one item, guaranteeing a collision. Even if n ≤ m, collisions are probable due to the birthday paradox.  **Average Chain Length:** In chaining, the average length of a chain is equal to the load factor, λ.  **Performance:** As long as the load factor (λ) is kept under control (e.g., by resizing the table), operations (insert, delete, search) can be performed in practically constant time, O(1). | | | **The Need:** If a hash function h is publicly known and deterministic, a malicious actor can choose input keys that all hash to the same slot. This creates a "worst-case" scenario (e.g., all items in one chain), leading to O(n) performance for operations, effectively crashing or significantly slowing down a server.  **Solution:** To prevent this, we choose a hash function randomly at the time of table initialization. This ensures that a malicious actor cannot predict the hash function and engineer worst-case inputs. | | |

# Graphs **–––––****–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Definitions** | | | | | | | |
| **Digraph (Directed Graph):**  A finite, non-empty set of vertices (V) together with a (possibly empty) set of ordered pairs of nodes (E) called arcs.  If :   * v is adjacent to u * u is an out-neighbour of v * v is an in-neighbour of u | | | | **Undirected Graph:** A finite, non-empty set of vertices (V) together with a (possibly empty) set of unordered pairs of vertices (E) called edges. | | | |
| **Order of a Digraph:** The number of nodes, | | | |
| **Size of a Digraph:** The number of arcs, | | | |
| **Sparse Digraphs:** | | | |
| (specifically for simple directed graphs without self-loops) | | | |
| **In-degree of a node (v):** The number of arcs entering v. | | | | From a node u to a node v is the minimum length of a path from u to v.   * If no path exists, the distance is undefined or . * For an undirected graph, for all vertices u and v. | | | |
| **Out-degree of a node (v):** The number of arcs leaving v. | | | |
| **Walk :** A sequence of nodes such that | | | |
| **Length of a Walk:** The number of arcs involved (n). | | | | The maximum distance between any two vertices. | | | |
| **Path:** A walk in which no node is repeated. | | | | .  (Minimum of the set of maximum distances from any single node to all other nodes). | | | |
| **Cycle:** A walk in which ​ and no other nodes are repeated. | | | | A graph is connected if it has finite radius and diameter. | | | |
| **Computer Representation of Graphs** | | | | | | | |
| **Adjacency Matrix**   * **An n \* n matric containing 0 or 1 to indicate an edge between nodes** * **Indexes indicate notes** * **Entry can contain an integer to represent weight of connection** | | | | **Adjacency Lists**   * **A list of lists** * **Index of list indicates source node** * **Items in inner list define out-nodes (destinations)** | | | |
| **Operation** | **Adjacency Matrix** | | | **Adjacency Lists** | | | |
| Are (i,j) exist? | is entry (i, j) 0 or 1 | | constant | find j in list i | | a | a notes size of list j |
| Out-degree of i | scan row and sum 1’s | | n | size of list i | | constant | |
| In-degree of i | scan column and sum 1’s | | n | for j != i,find i in list j | | n + m | |
| Add arc (i,j) | change entry (i,j) | | constant | insert j in list i | | constant | |
| Delete arc (i,j) | change entry (i,j) | | constant | delete j from list i | | a | |
| Add node | create new row / column | | n | add new list at end | | 1 | |
| Delete node | delete row / column I and shuffle other entries | | n2 | delete list i and for j != i, delete i from list j | | n + m | |
| **Tree Traversal** | | | | | | | |
| **Search Forest F:** a set of disjoint trees spanning a digraph G after its traversal  An arc is called a tree arc if it belongs to one of the trees of F  The arc , which is not a tree arc is called: | | | | | | | |
| A **forward arc** is u is an ancestor of v in F | | A **back arc** if u is an descendant of v in F | | | A **cross arc** if neither u nor v is an ancestor of the other F | | |
| **Depth First Search (DFS)**  - Start at node v  - Searching as far away from v as possible via neighbor  - Continue from the next neighbor unto no more new nodes | | | | **Breadth First Search (BFS)**  - Start at a node v  - Searching through all its neighbors, then the neighbors neighbors  - Continue unto no more new nodes | | | |
|  | | **Analysis of different Arcs Using DFS**  Notation: (seen[v] | done[v]), i.e. 9 | 10  If (v, w) is an arc, then a …   * Tree or forward arc is: seen[v] < seen[w] < done[w] < done[v] * Back arc is: seen[w] < seen[v] < done[v] < done[w] * Cross arc is: seen[w] < done[w] < seen[v] < done[v] | | | | | |
| **Priority First Search (PFS)**  A mix between depth first search and breadth first search. Which uses a priority queue structure to find the next node | | | | | | | |

# Graphs Traversal Algorithms **–––––––––––––––––––––––––––––––––––––––––––––––**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Topological Ordering** | | | | | |
| **Definition:** An ordering of nodes in a directed graph where every directed edge from node a to node B, node A appears before node B in the ordering  **Algorithm Time Complicity:** A topological sort algorithm can find a topological ordering in  **Uniqueness:** Topological orderings are not unique  **Graph Requirements:** Only Directed Acyclic Graphs (DAGs) have topological orderings  **Methods to Find:**   * In a tree, continuously pick off the “leaves”, then reverse the order * Reverse the output of a Depth First Search (Post-order) algorithm | | | | | |
| **Kosaraju's Algorithm (for Strongly Connected Components - SCCs)** | | | | | |
| **Goal:** Finds all Strongly Connected Components (SCCs) in a directed graph.  **SCC Definition:** A maximal subgraph where every pair of vertices (u,v) has a directed path from u to v AND from v to u.  **Core Idea:** Leverages two Depth-First Searches (DFS) on the original graph and its transpose.  **Steps:**   1. **First DFS (on Original Graph G):**  * Perform DFS on G. * Push vertices onto a stack in order of decreasing finishing times (i.e., when all descendants have been visited).  1. **Transpose Graph (GT):**  * Create GT by reversing the direction of all edges in G.  1. **Second DFS (on Transposed Graph GT):**  * Iterate through vertices in the order from the stack (from Step 1). * For each unvisited vertex, perform a DFS on GT starting from it. All nodes visited in this DFS form one SCC   **Time Complexity:**  **Space Complexity:** | | | | | |
| **Maximal and Maximum Matching** | | | | | |
| **Definition:** A matching M in a graph G=(V,E) is a set of edges such that no two edges in M share a common vertex. In other words, each vertex is incident to at most one edge in the matching. | | | | | |
| **Maximal Matching**  **A matching M is maximal if no more edges can be added to M** | | | **Maximum Matching**  **A matching is maximum matching if it contains the largest possible number of edges among all possible matchings in the graph** | | **Perfect Matching (or Complete Matching)**  **A matching M is perfect matching if every vertex in the graph is paired to exactly one edge in M** |
| **Single Source Shortest Path (SSSP)** | | Finds the shortest (minimum weight) path from a source node to every other node | | | |
| **Dijkstra’s Algorithm** | | | | **Bellman-Ford Algorithm** | |
| **Type:** Greedy algorithm  **Choice:** Each locally best choice is “globally” best  **Weights:** Works only if all weights are non negative  **Process:**   * Initial paths are arc paths of weight * Each step compares the shortest paths with and without each of the currently chosen new nodes as a predecessor   **Time Complexity for SSSP:**   * using binary heaps * using Fibonacci heaps   **Note:** Cannot work with weights of negative value | | | | **Type:** Dynamic Programming algorithm  **Weights:** Works with values < 0 (aswell)  **Negative Cycles:** Works as long there are no negative weighted cycles. (unless tweaked)  **Time Complexity for SSSP:** High time complexity  **Memory:** Uses extra memory to help  **Note:** Slower than Dijkstra’s | |
| **Shortest Path Retrieval** | |
| - When updating distance, store the predecessor next to it.  - When destination is reached, work backwards to reconstruct the path (like a reversed linked list) | |
| **All Pairs Shortest Path (APSP)** | Runs SSSP from every node | | | | |
| **Floyd’s Algorithm** | | | | | |
| **Goal:** Finds shortest path for each and every node. Works by comparing shortest path directly vs through another node  **Process:**   * Considers * Continues until all nodes are processed   **Time Complexity:** Runs in n3 time | | | | | |
| **Minimum Spanning Trees (MST)** | | | | | |
| **Definition:** A spanning tree is a subgraph with only  **Goal:** Connecting everything with the lowest total cost and without forming cycles  **Cost:** Total weight of edges  **Possible Spanning Trees:**  **Note that:** | | | | | |
| **Prim’s Algorithm** | | | | **Kruskal’s Algorithm** | |
| **Type:** Greedy Algorithm  **Selection:** Always selects the next smallest weighted edge  **Tree Maintaince:** Maintains a tree at each stage that grows to span the graph  **Implementation:** Implemented with a priority queue, similar to Dijkstra’s Algorithm  **Graph Requirement:** Graphs must be fully connected (one strong component) **Runtime:** | | | | **Type:** Greedy Algorithm  **Selection:** Always selects by the smallest edge but doesn’t form cycles  **Forest Maintenance:** Maintains a forest whose trees coalesce into one spanning tree  **Implementation:** Uses Disjoint Sets ADT  **Finding MST:** Will find for all components in disconnect graph  **Runtime:** | |