Array Query

Binary Search

- **Assumption:** Sorted list.
- **Algorithm:** Find middle, compare to query. Repeat on one half.
- **Time complexity:** $O(\log n)$.
- **Pitfalls:** (Check for bad query on size 1 or 2)
  - Termination
  - Index out of bound
  - Loop invariant (strict/lax upper bound)

Quick Select

- **Algorithm**: Pick a random element and pivot around. Check index to determine which side to recurse.
- **Time Complexity**: Avf $O(n)$; Worst $O(n^2)$.

Array Sorting

Bubble Sort

- **Algorithm**: Swap adjacent elements if in wrong order. Stop if one iteration results in no swapping.
- **Invariant**: Last n elements are the largest and sorted.

Selection Sort

- **Algorithm**: Find the least element and swap to head.
- **Invariant**: First n elements are the smallest and sorted.

Insertion Sort

- **Algorithm**: Move the head element to the left.
- **Invariant**: First n elements are sorted.

Merge Sort

- **Algorithm**:
  1. (Top down) Divide in half, sort each and merge.
  2. (Bottom up) Merge the first 2^n elements.
- **Invariant**: Each segment is sorted.

Quick Sort

- **Algorithm**: Pick a random element and pivot around it (three-way for possible repeats).
- **Invariant**: The pivots are at their correct positions.

Heap Sort

- **Algorithm**: Heapify then repeatedly extract maximum.
- **Invariant**: Heap structure.

Name	Complexity	Stability	In-place
Bubble	$O(n)$ to $O(n^2)$	Yes	Yes
Selection	$O(n^2)$	No	Yes
Insertion	$O(n)$ to $O(n^2)$	Yes	Yes
Merge	$O(n \log n)$	Yes	$O(n)$
Quick	Avg $O(n \log n)$ Worst $O(n^2)$	No	Yes
Heap	$O(n \log n)$	No	Yes

Binary Search Trees (BST)

- **Invariant**: All left are smaller, all right are larger.
- **Modifying** methods $O(h)$: insert, delete.
  - **Delete**: If leaf just delete. If one child, attach child to parent. If two children, swap with successor and delete successor (who has at most one child).
- **Query** $O(h)$: search, predecessor/successor, findMin/Max.
  - **Successor**: If result > key, return result. Else if result has right child, return min of right child. Else return the first parent larger than result.
- **Range query** list = $O(h+k)$; get number = $O(h)$ with weights: Find the first split point. For left child, if in range, recurse on left and list all else. Otherwise recurse on right child (finds envelope of the range).
- **Traversal** $O(n)$: Pre/In/Post-order.
- **Balanced** iff height = $O(\log n)$.

AVL Tree

- **Invariant**: Height of children differ by at most 1.
- **Rotations**:
  - Left child balanced or left heavy: right rotate node. Left child right heavy: left rotate left child, right rotate node.
  - Rotation decreases height by 1 except with balanced grand-children.
  - Insertion: 2 rotations. Deletion: $O(\log n)$ rotations.

Trie

- Faster search $O(L)$ compared to $O(L \log n)$ for bBST.
- More overhead (more nodes).

Order Statistics Trees

- **Rank** queries $O(h)$:
  - **Select rank**: If rank = left_weight + 1, stop. If rank < left_weight + 1, go left. If rank > left_weight + 1, go to right with rank = rank - left_weight - 1.
  - **Compute rank**: Start with left_weight + 1 and go up. If went right, do nothing. If went left, add left_weight + 1.

Interval tree

- **Invariant**: BST with respect to left end points, store maximum right end point.
- **Interval search** $O(h)$: If value in interval, return. If value > maximum right end point of left child, go right. Else go left.
- **All overlaps** $O(kh)$: Interval search. Delete. Repeat. Add back.

2D Trees

Implementation 1: bBST of x. At each node is rooted a bBST of y.

- **Build tree**: $O(n \log n)$; **Space**: $O(n \log n)$; **Search**: $O(\log n)$;
- Modifying methods disable as very bad complexity.
- **Range query** $O\left(\log^2 n + k\right)$: Find x envelope. For each x, find y envelope.

Implementation 2: bBST. Alternate splitting wrt x and y.

- **Space**: $O(n)$; **Search**: $O(\sqrt{n})$ since $T(n) = 2T\left(\frac{n}{4}\right) + O(1)$.
- **Insert/Delete**: $O(\log n)$; **Range query**: $O(\sqrt{n} + k)$.

(a, b)-Tree

- **Invariants**: **Perfect** tree (has all leaves). Each (non-root, non-leaf) node has at least $a-1$ and at most b keys, thus dividing range into a to b segments.
- **Insert** $O(\log_a n)$: When node is full, **split** and send median up.
- **Delete** $O(\log_a n)$: Find element, swap with predecessor/successor (a leaf). Remove element. If underfull, **share** with sibling: Merge with a neighbouring sibling together withthe separating key from parent. If overfull, split it again. If parent is underfull, share with its siblings.
- **Query**: $O(\log_a n)$ but effective constant.

Heap

- Implements a (max) **priority queue**.
- **Invariants**: Parent > child. **Complete** binary tree.
- **Insert** $O(\log n)$: add to the end and bubble up.
- **ExtractMax** $O(\log n)$: swap with last, delete, bubble it down.
- **Increase/decrease key**: find and bubble up/down.
- **Array**: $\mathrm{left}(x) = 2x+1; \mathrm{right}(x) = 2x+2; \mathrm{paren}(x) = \left\lfloor \frac{x-1}{2} \right\rfloor$.
- **Heapify** $O(n)$: Last layer already ok. Add top layers one by one.

Union Find (Disjoint Set)

- Element store the set identifier: **Find**: $O(1)$; **Union**: $O(n)$.
- Element store parent:
  - No optimisation: **Both**: $O(n)$.
  - Weighted union (flat tree): **Both**: $O(\log n)$.
  - Weighted union with path compression:
    Amortised both: $\alpha(m, n)$.

Hashing

Chaining

- **Space**: $O(m+n)$; **Insert**: $O(1)$; **Search**: $O(1+\alpha)$ with SUHA.

Open addressing

- **Space**: $O(n)$; **Insert/Delete**: worst $O(n)$, amortised $O\left(\frac{1}{1-\alpha}\right)$.
- **Resizing**: double when full, half when a quarter full.

HashSet

- Probability of **no** false positive with SUHA ~ $e^{-\alpha}$.
- **Bloom filter**: Use multiple hash functions instead of one.

Graphs

- **Diameter**: Length of longest all pairs shortest path.

Searching

- **BFS**: Use a **queue** for outgoing edges. Linked list $O(V+E)$.
  - Outputs rooted **least hops tree**.
- **DFS**: Use a **stack** for outgoing edges. Linked list $O(V+E)$.

Single-Source Shortest Path (SSSP)

- **Bellman-Ford** $O(VE)$
  - **Algorithm**: Use every edge to relax distance. Repeat until an entire iteration changes nothing.
  - **Invariant**: After j-th iteration, vertices whose shortest path is $\leq j$ hops long have the correct estimate.
  - **Negative cycle**: If on $(V+1)$-th iteration there is still change, there is negative cycle.
- **Dijkstra** $O(E \log V)$, non-negative edges:
  - **Algorithm**: Use a min **priority queue** of vertices with priority being estimated distance **from root** (cf. Prim). Extract min, add to tree, and use its outgoing edges to update estimate.
  - **Invariant**: All nodes added to shortest path tree has correct estimate.
- **Directed Acyclic** $O(E)$:
  - **Algorithm**: Construct **topological order** with post-order DFS. Relax edges in that order.
- **Tree** $O(V)$: Path is unique, so just BFS/DFS.

Minimal Spanning Tree (MST)

Properties

- **Substructure**: The two components after a cut are both MSTs.
- **Cycle**: For every cycle, the largest is **not** in MST.
- **Cut**: For every cut, the smallest one is in **all** MSTs.

Algorithms

- **Prim** $O(E \log V)$:
  - **Algorithm**: Use a min **priority queue** of verices with priority being minimum distance to **current subtree** (cf. Dijkstra). Extract min, add to tree, and use its outgoing edges to update priorities.
  - Uses cut property between current subtree and remaining vertices.
- **Kruskal** $O(E \log V)$:
  - **Algorithm**: Use a **disjoint set** of vertices. Sort all edges according to weight. Starting from the east weight edge, for each edge, if two end points are not in same set, union, and add edge to tree.
  - Uses cycle property. End points in same set $\Rightarrow$ edge is maximum in a cycle.
- **Boruvka** $O(E \log V)$:
  - **Algorithm**: Use a **disjoint set** of vertices with quick find.
    - Initialise: For each vertex, add minimum outgoing edge.
    - Boruvka step $O(\log n)$ times: Run BFS/DFS. For each connected component, keep track of minimum outgoing edge. After BFS/DFS, union along all such edges.

Dynamic Programming

- **Optimal subproblem**: Optimal solution can be constructed from optimal solutions to smaller sub-problems.
  - E.g. Greedy: Dijkstra, MST; Divide-and-conquer: Mergesort

Longest Increasing Subsequence (LIS)

- Assume DAG in topological order. Total n subproblems.
- **Suffix LIS** $O(n^2)$: Define l_i = length of longest LIS from it. Compute $l_i = \max(l_j) + 1$ where $j > i$ and $i \to j$ is an edge.

Prize Collection

- Maximise total prize in k steps.
- **Idea 1** $O(kE)$: Create k copies. Use SSSP on DAG on super source.
- **Idea 2** $O(kE)$: Total kV subproblems in table $P[v, k]$. $P[v, k] = \max(P[w, k-1] + p(v, w))$, where $v \to w$ is edge.

Vertex Cover on Tree

- Output size of minimum vertex cover. Total $2V$ subproblems.
- **Algorithm** $O(V)$:
  - First choose a root. Define $S[v, 0 \,/\, 1]$ to be the subproblem at subtree rooted at v, given that v is/is not in vertex cover. Then $S[\texttt{leaf}, 0 \,/\, 1] = 0 \,/\, 1$.
  - $S[v, 0] = \sum S[w, 1]$, where w is child of v. $S[v, 1] = 1 + \sum \min(S[w, 0 \,/\, 1])$, where w is child of v.

All Pairs Shortest Path

- Output all pairs shortest path arranged in table $\operatorname{dist}[v, w]$.
- **Idea 1** $O(VE \log V)$: Run SSSP on every source.
- **Floyd-Warshall** $O\left(V^3\right)$: Total V^3 subproblems. Store one hop distances.
  - Define $S[v, w, P]$ as shortest path $v \to P \to w$. P is maximum index used.
  - Base case $S[v, w, \varnothing] = \operatorname{weight}(v \to w)$. $S[v, w, P_i] = \min(S[v, w, P_{i-1}], S[v, v_i, P_{i-1}]) + S[v_1, w, P_{i-1}]$
- **Variants**: Transitive closure , minimum bottleneck of path.