Algorithmic Analysis

$1 < \log(\log n) < \log n < \log^2 n \leq \sqrt{n} < n < n \log n < n^2 < 2^n < 3^n < n! < n^n$

T(n) = T(n-1) + O(1)	$O(n)$
T(n) = T(n-1) + O(logn)	$O(n \log n)$
T(n) = T(n-1) + O(n)	$O(n^2)$
T(n) = 2T(n-1) + O(1)	$O(2^n)$
T(n) = T(n/2) + O(1)	$O(\log n)$
T(n) = T(n/2) + O(n)	$O(n)$
T(n) = 2T(n/2) + O(1)	$O(n \log n)$
T(n) = 2T(n/2) + O(n)	$O(n)$
T(n) = 2T(n/2) + O(n^2)	$O(n^2)$
T(n) = T(sqrt(n)) + O(1)	$O(\log(\log n))$

Searching:

Binary Search

```
int search(A, key, n)
  begin = 0
  end = n-1
  while begin < end do:
    mid = begin + (end-begin)/2;
    if key <= A[mid] then
      end = mid
    else begin = mid+1
  return (A[begin]==key) ? begin : -1
```

Loop invariants:
Correctness - A[begin] <= key <= A[end]
Performance - (end-begin) <= n/2k in iteration k.

Peak Finding

FindPeak(A, n)

```
if A[n/2+1] > A[n/2]:
  FindPeak (A[n/2+1..n], n/2)
else if A[n/2–1] > A[n/2]:
  FindPeak (A[1..n/2-1], n/2)
else A[n/2] is a peak; return n/2
```

Sorting:

BubbleSort(A, n)

```
repeat (n times/until no swaps):
  for j in [1…n-1]:
    if A[j] > A[j+1] then swap(A[j], A[j+1])
```

SelectionSort(A, n)

```
for j in [1…n-1]:
  find minimum element A[k] in A[j..n]
    swap(A[j], A[k])
```

InsertionSort(A, n)

```
for j in [2…n]:
  key = A[j]
  i = j-1
  while (i > 0) and (A[i] >key):
    A[i+1] = A[i]
      i = i-1
  A[i+1] = key
```

MergeSort(A, n)

```
if (n=1) then return;
else:
  X ¬MergeSort(A[1..n/2], n/2);
  Y ¬MergeSort(A[n/2+1, n], n/2);
return Merge (X,Y, n/2);
```

Name	Best	Avg	Worst	+Space	Stable	Invariant
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Largest j items sorted in correct positions
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Smallest j items sorted in correct positions
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	First j items sorted, may not be correct
MergeSort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	Yes	Sort within buckets before merging
QuickSort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$ (non-random pivot)	$O(1)$	No	Pivot in correct position Left partition all smaller than pivot Right partition all larger than pivot
HeapSort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	No	-

master theorem

$$T(n) = aT(\tfrac{n}{b}) + f(n) \quad a \geq 0, b > 1$$

$$= \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

QuickSort(A[1..n], n)

```
if (n==1) then return;
else:
  Choose pIndex (random until >1/10 for paranoid)
  p = partition(A[1..n], n, pIndex)
  x = QuickSort(A[1..p-1], p-1)
  y = QuickSort(A[p+1..n], n-p)
partition(A[1..n], n, pIndex):        // Assume no duplicates, n>1
  pivot = A[pIndex];                  // pIndex is the index of pivot
  swap(A[1], A[pIndex]);              // store pivot in A[1]
  low = 2;                            // start after pivot in A[1]
  high = n+1;                         // Define: A[n+1] = infinity
  while (low < high)
    while (A[low] < pivot) and (low < high) do low++;
    while (A[high] > pivot) and (low < high) do high– –;
    if (low < high) then swap(A[low], A[high]);
  swap(A[1], A[low–1]);
  return low–1;
```

For duplicates:
3-way partitioning (2 pivots)
1-pass, maintain 4 array regions

< pivot	= pivot	In Progress	> pivot

>2 pivots still same time complexity

Order Statistics

Finds the kth smallest element in an array. O(n)

QuickSelect(A[1..n], n, k)

```
if (n == 1) then return A[1];
else:
  Choose random pivot index pIndex (paranoid)
  p = partition(A[1..n], n, pIndex)
  if (k == p) then return A[p];
  else if (k < p):
    return QuickSelect(A[1..p–1], k)
  else if (k > p):
    return QuickSelect(A[p+1], k – p)
```

Node type	#Keys Min	#Keys Max	#Children Min	#Children Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

Trees

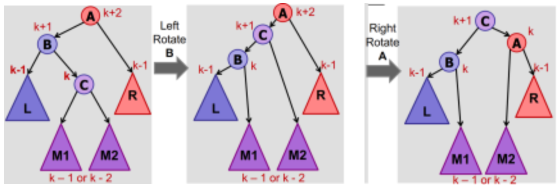
- For leaf node v: h(v) = 0, For root/internal node v: h(v) = max(h(v.left), h(v.right)) + 1
- Keep items in a hierarchical order to facilitate efficient search
- Keep items sorted in inorder traversal sequence

Binary Search Tree (BST)

- Not necessarily balanced unless h = O(log n)
- $\log n - 1 \leq h \leq n$
- On a balanced BST: all operations run in O(log n) time

AVL Tree

- Each node stores absolute height or relative height to parent
- On insert & delete update height
- A node v is height-balanced if: |v.left.height – v.right.height| <= 1
- $2^{\frac{h}{2}} \leq n < 2^h$
- Insert: O(logn)
  - Find location O(logn)
  - Walk up tree O(logn)
  - checkBalance O(1) -> unbalanced -> max 2 rotations -> O(1)



- Delete: O(logn)
  - Find location O(logn)
  - If v has 2 children, swap with successor
  - Delete v and reconnect children
  - For every ancestor of v: check height balance & rotate -> O(logn)

Trie

- Each node stores character & boolean endFlag
- Space: O(text size)
- Search, insert, delete: O(L)

(a,b)-Tree

- Each internal node: a <= children.length < b, where 2 <= a <= (b+1)/2
- Each root/internal node has children.length = keys.length + 1
- Child has key range (v_{i}–1, v_i]
- All leaf nodes have the same depth from root
- Maximum height: $O(\log_a n)$ + 1
- Minimum height: $O(\log_b n)$
- B-trees are simply (B, 2B)-trees
- Split: promote median of overfilled keys to parent key (proactive/passive)
- Delete: if leaf just delete, else swap with pred/succ then delete (proactive/passive)
- Pred: rightmost key in left subtree, Succ: leftmost key in right subtree
- Merge: Demote prev key of parent node to left sibling (propagate up)
- Share: Merge then Split (if overfilled)

Augmented Trees

Methodology:

1. Choose underlying data structure (tree, hash table, linked list, stack, etc.)
2. Determine additional info needed.
3. Modify data structure to maintain additional info when the structure changes. (subject to insert/delete/etc.)
4. Develop new operations

Dynamic Order Statistics

Support insert, delete, select(k): find the kth smallest node

1. AVL Tree
2. Weight of each node
3. Update weights
4. Select, Rank

Interval Queries

Support insert, delete, query(x): find an interval that contains x

1. AVL Tree
2. Upper bound of interval, upper bound of subtree
3. Update upper bound of subtree
4. Query: O(logn)

Orthogonal Range Searching

Support insert, delete, query(1Dinterval): find all points in 1Dinterval

1. BST
2. Store all points in leaves of the tree, internal node stores max of left subtree
3. Update max of left subtree
4. Query, FindSplit, LeftTraversal, RightTraversal, AllTraversal
   1. LeftTraversal: either AllTraversal right & recurse left, or recurse right -> O(logn)
   2. RightTraversal: either AllTraversal left & recurse right, or recurse left -> O(logn)
   3. AllTraversal: O(k), k = no. of items found
   4. Query: O(k + logn)
   5. BuildTree: O(nlogn)
   6. Space complexity: O(n)

If just need to count no. of points, store and use weight of each node instead of AllTraversal

Support insert, delete, query(2Dinterval): find all points in 2Dinterval

1. 1D Range Tree for x-coords
2. 1D Range Tree for y-coords in each node
3. Same as above
4. Same as above
   1. FindSplit: O(logn)
   2. Traversal on x: O(logn)
   3. Traversal on y: O(logn) (for each x traversal)
   4. Output: O(k)
   5. Query: O($\log^2$n + k)
   6. BuildTree: O(nlogn)
   7. Space complexity: O(nlogn)

kd-Trees

- Alternate levels in the tree
- Each level divides the points in the plane in half
- Supports more efficient updates
- Often better in practice
- Good for a variety of queries (e.g., nearest neighbor)

Type	Search	Insert	Delete	Pred/Succ	Traversal
BST	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
(a,b)	$O(log_a n)$	$O(blog_a n)$	$O(\log n)$		
Trie	$O(L)$	$O(L)$	$O(L)$		$O(n)$
HashTable	$O(1)$ *or* $O(n)$	$O(1)$	$O(1)$	--	--

Hashing
- h = hash function
- n = no. of keys
- m = no. of buckets

Hash Functions
- h: U -> {1…m} O(1)
- Store key k in bucket h(k)
- Collisions are inevitable (pigeonhole principle)

Chaining (Collision Resolution)
- Each bucket -> linked list
- Space: O(m + n)
- Insert: O(1)
    - Calculate h(key) -> O(1)
    - Get bucket h(key) -> O(1)
    - Add (key, value) to linked list -> O(1)
- Search: O(n) worst, O(1) best, O(1 + n/m) SUHA
    - Calculate h(key) -> O(1))
    - Get bucket h(key)) -> O(1)
    - Get items in linked list -> O(n)
- Delete: O(n) worst, O(1) best, O(1 + n/m) SUHA
- Inserting n elements: max cost O(logn/log(logn))

Simple Uniform Hashing Assumption (SUHA)
- Every key equally likely to map to every bucket
- Keys mapped independently
- load(HashTable) = n/m = avg # items per bucket
- Search: O(1 + n/m) = O(1) if load < 1

Open Addressing (Collision Resolution)
- On collision, probe sequence of buckets (all) till empty found
- h(key, **i**) : U -> {1..m} where **i** = no. of collisions (enumerate all buckets)
- Linear Probing: check next bucket
    - Problem: clusters of size O(logn) if table is 1/4 full (not O(1))
    - Reality: caching -> fast access nearby elems -> faster
- Search: check each bucket h(key, i) from 1 to m
- Delete: set removed bucket as tombstone value for search
- n items, m buckets, SUHA -> expected cost $\leq$ 1/(1-α)
- Sensitive to hash function choice & load; performance degrades as $\alpha \to 1$

Double Hashing
- h(k, i) = (f(k) + i*g(k)) mod m
- if g(k) relatively prime to m -> h(k, i) enumerate all buckets

Table Resizing
- If (n == m), then m = 2m; If (n < m/4), then m = m/2
- Only grow when at least m/2 new items added
- Only shrink when at least m/4 items deleted
- Every time you shrink a table of size m, at least m/4 items were deleted
- Growing: O(n + 2n + n) = **O(n)** assuming m < n < 2m (new table size)
    - Scanning old HT: O(m) = O(n), init new HT: O(2m) = O(n)
    - Inserting each element in new hash table: O(1)
- Amortized expected cost for operations: O(1) (most inserts take 1)
    - Amortized cost T(n): cost of (int) k operations is $\leq$ k*T(n)

Graphs
- Vertices + Edges (connect 2 nodes, unique)
- Multigraph: >=1 edge between 2 nodes
- Hypergraph: edge connects >=2 nodes
- Graph G <V,E>; $V > 0, E \subseteq \{(v,w): (v \in V), (w \in V)\}$

Terminology
- (Simple) Path: set of edges connecting 2 nodes
- (Dis)Connected: (not)every pair of nodes connected by path
- Cycle: form a loop
- Tree: connected acyclic graph
- Forest: acyclic graph
- Degree of node: no. of adjacent/ingoing/outgoing edges
- Degree of graph: max degree of nodes in graph
- Diameter: shortest path dist btw furthest nodes
- Sparse: E = O(V); Dense: E = O(V^2)

Special Graphs
- Star: 1 central node; all edges adjacent to center (deg n-1, diam 2)
- Clique (complete graph): all node pairs connected by edges (deg n-1, diam 1)
- Line/path: line of nodes (deg 2, diam n-1)
- Cycle: circle of nodes (deg 2, diam n/2 or n/2-1)
- Bipartite: 2 sets of nodes, no edges within same set (deg ?, diam ?)

Representations
- Adjacency List (Sparse): size V array of neighbour LinkedLists O(V+E)
- Adjacency Matrix (Dense): V*V matrix of edges (0/1) O(V^2)
    - A^n -> no. of n-length walks

Query	v&w nbrs?	Enumerate nbrs	Find any nbr
A-List	Slow	Fast	Fast
A-Matrix	Fast	Slow	Slow

BFS
1. Queue frontier, boolean[] visited, int[] parent; frontier.add(start)
2. While frontier not empty, v = frontier.poll()
3. Enum v.nbrs, if not visited then set parent & add to frontier
4. Take parents of end until start to get shortest path

- O(V+E), shortest path, adjacency list, form tree
- SSSP for weightless/same weight edges (min hops not distance)

DFS
1. Stack s, boolean[] visited, s.push(start)
2. While stack not empty, v = frontier.pop()
3. Enum v.nbrs, if not visited then push to s

- O(V+E), not shortest path, adjacency list, form tree

Directed Graphs
- Edges are directed (order matters, start & end nodes)
- In/Out-Degree: no. of incoming/outgoing edges
- BFS/DFS work, only consider outgoing edges as nbrs

Topological Sort
- Topo Order: sequential total order of all nodes, edges point fwd (DAG)
- Sort: post-order DFS -> process node on last visit (after children)
- Process node: prepend to T.O. (leaves processed first then parents)
- Time complexity: same as DFS O(V+E)
- Not unique
- Kahn's algo: add all roots to T.O., remove roots, repeat. O(V+E)

Connected Components
- Undirected: path from v to w
- Strongly CC (digraph): path from v to w & w to v
- Graph of Strongly CC is acyclic

Bellman-Ford
- SSSP on digraph
- relax(u, v): if (dist[v] > dist[u] + weight(u,v)) { reduce dist[v] }
- Relax all edges: O(E), V times: O(V) -> total O(VE)
- Early termination: entire iter of relax ops -> no change
- Invariants: after n iters, n hop estimate is correct. est[n] >= dist[n]
- Detect negative weight cycle: V+1 iters still changing, unsolvable

Djikstra
- SSSP on digraph, no negative weights, faster than Bellman-Ford
- Store ests in PQ, take min, add to tree & relax all outgoing
- O(ElogV) = insert/delete from PQ V times: O(VlogV) + relax/decreaseKey E times: O(ElogV)
- Invariant: every dequeued vertex has correct est
- Early term: destination done once dequeued

DAG TopoSort
- Relax edges in T.O., O(E) = O(V+E)
- Reverse post-order DFS
- Longest path: negate weights and find shortest path

Union-Find
- Union: connect 2 nodes; Find: path connecting the 2 exists?
- Quick-Find: Maintain array of componentIDs
    - U: update all CIDs of component -> O(n)
    - F: check matching CID -> O(1)
- Quick-Union: Maintain array of parents
    - U: set root1 parent of root2 -> O(n) (unbalanced tree)
    - F: check matching roots -> O(n) (unbalanced tree)
- Weighted Union: maintain size & parents
    - Balance tree by setting heavy root parent of light root
    - U: O(logn), F: O(logn)
    - Alts: union by rank = log(size) or height
- Path Compression:
    - When root found, set parent of traversed nodes to root
    - OR set parent to grandparent (alternate nodes)
    - U: O(logn), F: O(logn)
- WU + PC: m U/F ops on n objects -> O(n + m*α(m,n))
    - Inverse Ackermann function <= 5
    - U: O(α(m,n)), F: O(α(m,n))

MSTs
- Acyclic subset of edges + connects all nodes + minimum weight
- V-1 edges; Not unique; Not the same as shortest path
- Reweighting/negative weights are OK
- Properties:
    1. No cycles
    2. Cut MST -> 2 MSTs
    3. Max edge of a cycle not in MST (min edge may/not)
    4. Min edge across any cut is in MST
- Weightless/same weight graph: BFS/DFS, O(E)
- Directed MST: only if acyclic + 1 root -> add min incoming edge for all nodes except root O(E)
- MaxST: reverse Kruskal OR negate all edge weights + MST algo

Generic MST Algo (Greedy)
- Red rule: Cycle with no red -> max edge red
- Blue rule: Cut with no blue across -> min edge blue
- Apply any rule to any edge until all coloured
- Blue edges -> MST (red rule -> acyclic/tree, blue rule -> spanning)

Prim
- PQ (key: min edge) for nodes across cut of {S, V-S}, S = set of nodes in MST, parent HashMap (depending on MST representation)
- While PQ not empty, remove min node and add to S, update PQ
- Proof: each added edge is min on some cut
- O(ElogV) = O(VlogV) + O(ElogV); like Djikstra but weight not dist
- Variant: edges have fixed weight range (e.g. 1-10)
    - Fixed size array of LinkedLists w/ nodes of each weight
    - O(E) = V*O(1) add/remove PQ + E*O(1) decreaseKey

Kruskal
- Union-Find for blue tree components, sorted array of edge weights
- Edges in ascending weight order: if join 2 nodes in same blue tree, red (max weight in cycle); else blue (min weight across cut)
- O(ElogV) = O(ElogE) sorting + E * O(logV OR α) U/F ops
    - logE = O(logV) since E = O(V^2) -> logE = O(2logV)
- Variant: edges have fixed weight range (e.g. 1-10)
    - Fixed size array of LinkedLists w/ edges of each weight
    - O(αE) = O(E) fill array + E * O(α) U/F ops

Boruvka
- Create V connected components (1 per node) O(V)
- Repeat: O(logV) times
    - For each CC, add min outgoing edge (BFS/DFS) O(V+E)
    - Merge at least k/2 components where k = no. of CC O(V)
- O((E+V)logV) = O(ElogV)

Dynamic Programming
- Optimal sub-structure; subproblems overlap (memoise), no overlap (D&C)
- Top-down: recurse + memoise; bottom-up: solve small + combine
- Table view: rows = steps; columns = nodes; each cell = subproblem
- Longest Increasing Subsequence O(n^2):
    - S[i] = LIS[i..n] starting at A[i]
    - S[i] = (max connected node after) + 1; O(i) = O(n); for n nodes
- Lazy Prize Collecting (max points k steps) O(kE):
    - Transform G -> DAG w/ k copies per node, longest path per source
    - P[v,k] = max(P[w,k-1] + e(v,w) per v.nbrs)
- Min Vertex Cover on Tree (min set of nodes touching every edge) O(V):
    - S[v,0/1] = size of VC in v subtree if v not/covered (2V subproblems)
    - S[v,0] = sum of S[child,1]; S[v,1] = 1 + sum of S[child,0] O(1)
- All-Pairs Shortest Path with q queries O():
    - Djikstra per query: O(qElogV); w/ memoisation: O(VElogV)
    - Preprocessing: APSP + memoise all, then O(q) for queries
    - Weightless/same weights: BFS*V = O(VE)
    - Floyd-Warshall: S[v,w,P8] = min(S[v,w,P7], S[v,8,P7]+S[8,w,P7])
        - Consider only n+1 sets P in increasing size O(V)
        - Calculate SP from v-w: O(V^2) -> O(V^3)
        - Path reconstruction: store first hop/intermediate node
        - Transitive closure: return path adjacency matrix
        - Min bottleneck edge: return heaviest edge on path matrix

MaxHeap
- Unsorted binary tree: keys = priorities (p) -> can be used to implement PQ
- Invariant: u.p >= max(u.left.p, u.right.p)
- Complete tree: every level full except leaves, leaves as far left as possible
- bubbleUp: swap w/ parent; bubbleDown: swap w/ max child O(logn)
- Insert O(logn): ins @ end + bubbleU
- extractMax/delete O(logn)/O(h): swap(last, deleted) + bubbleD
- Represent: tree or compact array (seq: BFS order; variable-length)
- u.L = 2i, u.R = 2i+1, u.parent = ⌊i/2⌋ OR binary: append 0/1 & remove last
- Heapify: reverse traverse array & bubbleD if needed O(n)
    - Invar: all subtrees starting at [i..n] are heaps
- HeapSort: heapify O(n) + extract n times O(nlogn) = O(nlogn)