CS2040S AY22/23 SEM 2 FINALS CHEATSHEET

Storing String in AVL tree:
- AVL tree are selfbalancing → O(log n)
- Comparison between strings → O(L)
- ∴ Insert/Delete/Search → O(L log n)

Node Type	# Keys (Slots)		# Children	
	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

- When #Child > #Keys, choose median key & split keylist into 2 halves.
- Left half goes into a new node
- Move median to parent (i.e shift median to a key in parent node)

$\log_a(n) < n^a < a^n < n! < n^n$

Orders of Growth:

$k < \log\log(n) < \log(n) < \sqrt{n} < \log^2(n) < n < n\log(n) < n^2 < n^3 < n^3\log(n) < n^4 < 2^n < 2^{2n} < n!$

In general:

Recursion	Master Theorem: $a > 0, b > 1, d \geq 0$ $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$; $T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$

Important:

- $\sqrt{n}\log n = O(n)$
- $O(2^{2n}) = O((2^n)^2) \neq O(2^n)$ **Exp power is significant
- $\log(n!) = \Theta(n \log n)$ (Sterling's Approximation)

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right) \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

- GP: $n + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^n} = 2n = O(n)$

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1}-1}{x-1} \quad | \quad \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \text{ when } |x| < 1$$

- AP: $T(n-1) + T(n-2) + \cdots + T(1) = 2T(n-1)$

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n = \frac{1}{2}n(n+1) = \Theta(n^2)$$

- Harmonic Series

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n}\frac{1}{k} = \ln(n) + O(1)$$

- $f(n) = \frac{1}{n} \rightarrow f(n) = O(1)$

** Note for strings: Concatenation takes O(n) time, append $O(n^2)$;

Space Complexity:

- Max space used during computation | $\Theta(f(n))$ time $\rightarrow \Theta(f(n))$ space

Logarithmic Rules: (Example: $2^{4\log n} = (2^{\log n})^4 = n^4$)

$a^{mn} = (a^m)^n$	$a^{\log_a b} = b$	$\log_b(a) = \frac{\log_x(a)}{\log_x(b)}$
$\log_b\left(\frac{1}{a}\right) = -\log_b(a)$	$\log_b(a)^n = n\log_b(a)$	$\log_b(a) = \frac{1}{\log_a b}$

PeakFinding: (Key idea: Binary Search) ** if there are multiple peaks, algo fails

log n	`FindPeak(A, n)`
	`  if A[n/2+1] > A[n/2] // right`
	`    FindPeak (A[n/2+1..n], n/2)`
	`  else if A[n/2-1] > A[n/2] // left`
	`    FindPeak (A[1..n/2-1], n/2)`
	`  else A[n/2] is a peak; return n/2`
		✅ Only 1 peak.
		❌ multiple/ no peaks

n	`FindPeak(A, n)       //STEEP PEAKS`
	`  if A[n/2-1] == A[n/2] == A[n/2+1] // L&R`
	`    FindPeak (A[n/2+1..n], n/2)`
	`    FindPeak (A[1..n/2-1], n/2)`
	`  else if A[n/2+1] > A[n/2] // R`
	`    FindPeak (A[n/2+1..n], n/2)`
	`  else if A[n/2-1] > A[n/2] // L`
	`    FindPeak (A[1..n/2-1], n/2)`
	`  else A[n/2] is a peak; return n/2`
		✅ Homogeneous arrays.
		❌ multiple/ no peaks

Sorting

✅ **Bubble**	After i^{th} iteration, last i elements are sorted **$O(n^2)$**
✅ **Insertion**	After i^{th} iteration, 1^{st} i elements are relatively sorted **$O(n)$, $O(n^2)$**
❌ **Selection**	After i^{th} iteration, 1^{st} i elements are sorted **$O(n^2)$**
✅ **Merge**	For every call to merge, both arguments are alw sorted. **O(nlogn)**
❌ **Quick**	After partitioning, all elements > pivot occur before the pivot, and all elements > pivot occur after the pivot **O(nlog n), $O(n^2)$**

Binary Search (Loop Invariant: A[begin] ≤ key ≤ A[end])

- Time: O(log n); Space: O(1) ** Works with duplicated values
- Pre-Condition: Array A is sorted; Post-Condition: A[begin] = key

```
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
```

ways to order insertions: n!
shapte of a binary tree → 4^n (Catalan No.)

- Use when input is sorted/ can manipulate input to be sorted
- By checking a value, eliminate all that are smaller/larger
- To find local minimums or maximums

Trees (Height of Tree = max(left.height, right.height) + 1)

Binary Search Trees (BST):

A BST is a binary tree what satisfies the BST invariant: left subtree has smaller elements & right subtree has larger elements

Function	Average	Worst
Insert, Delete, Search	O(h) = O(log n)	O(n)
Successor/Pred	O(h) = O(log n)	O(n)
findMax/ findMin	O(h) = O(log n)	O(n)
In-order Traversal	O(n)	

- A node v is height balanced if |v.left.height – v.right.height| ≤ 1
  - If all nodes in a BST is height balanced → height balanced BST
- The weight of a node u is equal to the no. of nodes in the subtree rooted at u → u.weight
  - The tree T is ¾ weight balanced if ever node in T is ¾ weight balanced
  - Every ¾ weight balanced tree is balanced ** **balanced if h = O(log n)**

$$n\left(\frac{3}{4}\right)^h = 1 \rightarrow h = \log_{\frac{4}{3}} n + 1$$

- Balanced Tree: Insert/Delete/Search → O(log n)
- Normal BST: Might degrade into linear tree, Worst O(n).
- Traversal for both is still O(n). As you need to visit every node
- The In-Order of a BST can give us the whole tree structure.

Balanced Binary Search Tree (BBSTs)

- Balanced if h = log n OR O(h)

Function	Average	Worst
Insertion, Delete, Search	Θ(log n)	O(log n)

AVL Tree

- Allows for O(log n) insert, delete & search
- Balanced Factor (BF): H(node.R) – H(node.L)
- Height Balanced: |left.height – right.height| ≤ 1
  - Right rotate → root of the subtree moves R
  - Left rotate → root of the subtree moves L
- **Left rotation requires Right child (vice versa)
- A height-bal tree w height h has at least n > $2^{h/2}$ nodes; (at most h < 2log(n))
- IF BF ∉ { -1, 0, +1 } → rebalance tree

LeftLeft → Rotate Right	**LeftRight → Rotate Left then Right**
RightRight → Rotate Left	**RightLeft → Rotate Right then Left**

- Space Complexity: O(LN),

Tree Rotation: (Tree rotations can create every possible tree shape)

After Insertion	Max 2 rotations to balance
After Deletion	O(log n) rotations to balance

Tries:

- 1 path down the Trie can represent multiple words (depending on flag)

Search (String length L)	O(L)
Insert (String length L)	O(L)
Space	O(size of text) * OverHead

Tries Trade Offs:

Time	Tries tend to be faster: O(L) vs Tree: O(hL) • Does not depend on size of text • Does not depend on no. of string
Space	Tries tend to use more space • ACSII chat set: 256 (OverHead) • A lot of children; but wasted space

Probability Theory:

- If an event occurs with probability p, the expected number of iterations needed for this event to occur is 1/p .
- For random variables: expectation is always equal to the probability
- linearity of expectation: E[A + B] = E[A] + E[B]

Uniform Random Permutation:

- if we have n elements, there are n! possible orderings, and each ordering has a probability of 1/n! of being chosen as the random permutation.
- E.g Fisher-Yates shuffle Algo: Start with an ordered sequence of elements, and iteratively swap each elem with another randomly chosen element from the remaining sequence. → O(n)

1. Start with an ordered sequence of n elements.
2. For i from n-1 down to 1, do the following:
   - Pick a random int j from 0 to i inclusive.
   - Swap the i-th elem with the j-th elem.

$\frac{\#Outcome}{\#Permutations} \in \mathbb{N}$ P(item remaining in initial pos) = 1/n

Dynamic Order Statistics on Balanced Tree:

Store size of sub-tree in every node

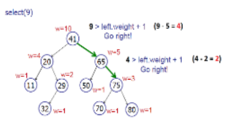
Define weight:

- w(leaf) = 1
- w(v) = w(v.left) + w(v.right) + 1
- "rank in subtree" = left.weight + 1

Let n in select(n) be the rank of node

Let L be left.weight +1 (i.e weight of left child node)

- If n > L → Go right, then select(n-L) on child node
- If n < L →Go left, then select(n) on child node
- Rank(v) → Computes the rank of a node, v.
- Select(k) → Finds the node with rank k.



Key Invariant:

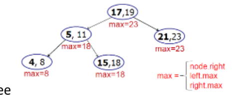
- After every iteration, the rank is equal to the its rank in the subtree rooted at node.
- At the end of the program, the rank would be equal to the rank of the subtree rooted at the root (correctness)
- In every node, the rank is either the same (if no nodes have come before it), or the rank is increased by the number of nodes that came before it)

Note: Update Weights & Rotate to maintain D.S when insert/delete.

Interval Queries/Searching

Interval Trees:

- Each node is an interval.
- Tree is sorted by the left endpoint.
- Augment: Maximum endpoints in subtree
- Stores the max endpoint (right) in the subtree at the node too
- Rotate tree & update intervals to maintain D.S

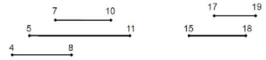


Searching: (If there are k leaves → total nodes in tree ≤ 2k)

- interval-search(x): find interval containing x. → O(log n) time
- **If search goes right**: then no interval in left subtree. → Either search finds key in right subtree or it is not in the tree.
- **If search goes left**: if there is no interval in left subtree, then there is no interval in right subtree either. → Either search finds key in left subtree or it is not in the tree.
- **Conclusion**: search finds an overlapping interval, if it exists.
- If search goes left & no overlap → key < every interval in right subtree.

Listing all intervals that overlap a point: **(All-Overlaps Algorithm)**

- **Repeat** until no more intervals:
  - Search for interval
  - Add to list
  - Delete interval
- **Repeat** for all intervals on list:
  - Add interval back to tree.



- Running Time: O(k log n) (lecture eg); irl, best time: O(k + log n)

Orthogonal Range Searching

1D Range Queries

1. Use a binary search tree. (Need to maintain BST invariant)
2. Store all points in the leaves of the tree. (Internal nodes store only copies; i.e internal nodes are guide posts.)
3. Each internal node v stores the MAX of any leaf in the left sub-tree.
   - Leaf nodes → Data nodes; Internal Nodes → Guide Posts

- Find "split" node → v = FindSplit(low, high); O(log n)
- Do left traversal → LeftTraversal(v, low, high); O(2k) = O(k)
  At every step, we either:
  1. Output all right sub-tree and recurse left. OR
  2. Recurse right.
- Do right traversal → RightTraversal(v, low, high); O(2k) = O(k)
  At every step, we either:
  1. Output all left sub-tree and recurse right. OR
  2. Recurse left.

Invariant:

The search interval for a left-traversal at node v includes the maximum item in the subtree rooted at v

Note:

- Binary Search is good for finding 1 item in the tree.
- However, now that we want to find a range of items in a tree, we have to augment the tree, such that we can find range with an efficient algorithm

2D Range Tree: (Use Augmented Trees)

- Build an x-tree using only x-coordinates.
- Create a 1d-range-tree on the x-coords.
- For every node in the x-tree, build a y-tree out of nodes in subtree using only y-coordinates. (I.e Range-trees inside range trees)

d-dimensional Range Queries:

- Store d–1 dimensional range-tree in each node of a 1D range-tree.
- Construct the d–1-dimensional range-tree recursively.

Query time Q_d (not including point reporting) given by the recurrence:

$Q_d(n) = O(\log n) + O(\log n) \cdot Q_{d-1}(n)$

Priority Queue:

Maintain a set of prioritized objects: (Can be either min/max)

– insert: add a new object with a specified priority

– extractMax: remove and return the object with max valued priority

Sorted Array	Unsorted Array
insert: O(n) • Find insertion location in array. • Move everything over. **extractMax: O(1)** • Return largest element in array	**insert: O(1)** • Add object to end of list **extractMax: O(n)** • Search for largest elem in array. • Remove&move everything over.
AVL Tree (indexed by priority) (Sorted Array_)	
insert: O(log n) • Insert object in tree	**extractMax: O(log n)** • Find maximum item. Delete it from tree

d-Dimensional Range Queries

Query cost:	$O(\log^d n + k)$
buildTree cost:	$O(n \log^{d-1} n)$
Space:	$O(n \log^{d-1} n)$

Max Height of heap = floor(log n) = O(log n)

Heap (Binary Heap/ Max Heap)

- Implements a Max Priority Queue
- Maintain a set of prioritized objs.
- Store items in a tree.
  - Biggest items at root.
  - Smallest items at leaves.

2 properties of a Heap:

Heap Ordering	Complete Binary Tree
Priority[parent] ≥ priority[child]	• Every level is full, (except possibly last lvl) • All nodes are as far left as possible.
Insert	**delete**
Steps: (e.g insert(25)) 1. Add a new leaf w priority 25 2. BubbleUp	1. Swap(n, last()); 2. Remove(last()); 3. BubbleDown
decreaseKey	**extractMax**
1. Update the priority 2. BubbleDown ** bubble down the LEFT side	1. Node v = root 2. Delete(root)
increaseKey	Just bubble up

Heap vs. AVL Tree

- Same asymptotic cost for operations
- Faster real cost (no constant factors!)
- Simpler: no rotations
- Slightly better concurrency

How to store a Tree (Heap) in an Array?

- Map each node in the complete binary tree into a slot in an array
- Level-Order (BFS)

What about inserting?

- Insert at the next available index (fill the leaf nodes from left)
- Bubble up priority, s.t the heap invariant is preserved

Check for Child node	left(x) = 2x + 1, right(x) = 2x + 2
Check for Parent node	parent(x) = floor((x-1) / 2)

* NOTE, we cannot store AVL-trees as an array:

- Many "holes" in array → waste space
- Need to rotate to maintain invariant, but it is not O(1) → costly
- Storing trees in an array is not efficient (except complete balanced tree)

Advantage of heap:

- Heaps can be stored in an array instead of nodes
- Everything will be stored in the same memory area → cache locality will be faster

HeapSort	
Invariant	Step 1 Heapify → Maintain Heap Invariant Step 2 ExtractMax → Maintain Heap Invariant
Time	Always complete in O(n log n)
Extra Space	O(1); in-place, only uses O(n) space

How to perform HeapSort?

1. Build a heap (from unsorted list)
2. HeapSort (By finding the max node over and over again)

Cost of Building a Heap → O(n)

- Initial: Start with a Complete Tree (recursion)
- Base Case: Each leaf is a Heap
- Recurse: Left + Right are Heaps
- cost(bubbleDown) = height
- More than n/2 nodes are leaves (h = 0)
- Most nodes have small height

Fib Heap:
Insert, Union, decreaseKey → O(1)
extractMin → amortized O(log n)

Heapify → O(n) ; HeapSort → O(n log n)

More Interval Trees:

- Process intervals into a tree, sorted by their lower bound (this.low)
- Each node also stores this.max = max(left.max, right.max, this.high). Recursively bubbled up from bottom to top.

FindInterval: Always go to the left subtree if possible

FindAllIntervals: Find & remove all intervals, then add all intervals back

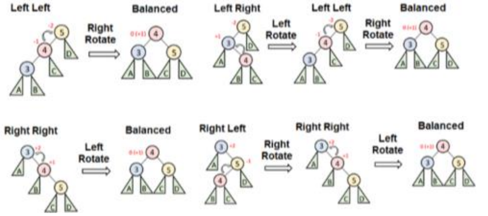
```
FindInterval(x) // or interval-search in lecture
  c = root;
  while (c != null and x is not in c.interval) do
    if (c.left == null) then c = c.right;
    else if (x > c.left.max) then c = c.right;
    else c = c.left;
  return c.interval;
```

Stacks & Queue:

Stacks	Queue
Push, Pop, Peek → O(1) • Diff permutations	Enqueue, Dequeue, Peek → O(1) • Process items in a sequence

Prims, Boruvka, Kruskal → MST
Dijkstra, BellmanFord → SSSP
Kahn's, Post-order DFS + relax → DAG_SSSP

Tree Rotation Diagram



UFDS (Connected Components)

Quick Find	O(1) Find: check if objects have the same componentId		
	O(n) Union: enumerate all items in arr to update id		
Quick Union	O(n) Find: check for same root		
	O(n) Union: add as a subtree of the root		
Weighted Union	O(log n) Find: check for same root		
	O(log n) Union: Add as a smaller tree as subtree of root		
Path Comp.	F: O(log n)	U: O(log n)	Size (Bk) = $\Theta(2^k)$
Weighted+PC	F: O(α(m, n))	U: O(α(m, n))	Height(Bk) = k – 1

- Quick Find/Union: Slow, expensive, not balanced
- Weighted Union: Fast, Balanced
- Weighted + PC: UFDS on n objects: O(n + mα(m, n))
- Path Compression: link node to root/ grandparent, both works equally

Weighted + PC: Worst:O(log n) → Cannot compress, as we keep unifying the roots + find lowest child

Important property: (Weight is easier to balance vs rank/size/height)
- weight/rank/size/height of subtree does not change except at root (so only update root on union).
- weight/rank/size/height only increases when tree size doubles.

Hashing
- No successor/predecessor operations
- Let m → table size; n → no. of items, cost(h) → cost of hashing
- $load, \alpha = \frac{n}{m}$

SUHA:
1) Every key has equal probability of being mapped to every bucket
2) Keys are mapped independently

Uniform Hashing Assumption
- Every key is equally likely to be mapped to every permutation, independent of every other key
- NOT fulfilled by linear probing

Properties of a good hash function:
1) Able to enum all possible buckets $h: U \to \{1 \dots m\}$
   - For every bucket j, $\exists i$ such that $h(key, i) = j$ (linear probing: true)
2) SUHA + Deterministic

Open Addressing: **replace deleted value with T.S value, not NULL

Division	$h(k) = k \bmod m, (m \text{ is prime})$ • DON'T choose $m = 2^x$ or 10^x • If k & m have common divisor d, only $\frac{1}{d}$ table used
Multiply	$h(k) = (Ak) \bmod 2^w \gg (w - r)$ for odd constant A • $m = 2^r, w = size\ of\ key\ in\ bits$

Issues with Open Hashing:
- Chaos with cycles (Cyclic Probing): Due to: High load factor, bad hash fn
  - Re-size table (n = m → mx2; n < n/4 → m/2), use good hash fn.

Linear Probing	$h(k, i) = h(k, 1) + i \bmod m$ • table ¼ full → clusters of size θ(log n) • Expected **cost of operation**, $E[\#probes] \le \frac{1}{1-\alpha}$
Double Hashing	$h(k, i) = f(k) + i \cdot g(k) \bmod m$ • If $g(k)$ is relatively prime to m→$h(k, i)$ hits all buckets

Closed Addressing – Chaining

Insert(k,v)	$O(1 + cost(h)) = O(1)$ Expected Max Cost: $O(\log n) = \theta\left(\frac{\log n}{\log(\log n)}\right)$	
Searching	Worst: $O(n + cost(h)) = O(n)$ Expected: $O\left(\frac{n}{m} + cost(h)\right) = O(1)$	
Space	O(m + n), m = size of HT, n = sum(linked lists.length())	
	LinkedList	**BBST**
Search	O(n)	O(log n)
Insert	Insert @ head = O(1) Insert @ tail = O(n)	O(log n)
Delete	Delete @ head = O(1) Delete anywhere = O(n)	O(log n)

Expected cost of Searching(Open): $\le \frac{1}{1-\alpha}$

$\frac{n-i}{m-i} \le \frac{n}{m} \le \alpha < 1$

$O\left(\frac{1}{1-\alpha}\right) = O\left(\frac{1}{1-\left(\frac{n}{m}\right)}\right)$

- To avoid primary clustering, Resolution: (H(key)+i*d)%m
- i → id; d → some const int; m → m's coprime
- Since d & m are coprime, this sequence covers the entire hash table
- Any insertion into the same starting position will have the same probe

- Linear Probing: If the table is ¼ full, then there will be clusters of size θ(log n)
  - Cache may hold entire
  - Expected cost of operation $\le \frac{1}{1-\alpha}$

Bellman-Ford & Dijkstras Common Invariant: ** PQ → overestimate
- Each node v stores an estimate est[v]. During the execution (after initialization), if est[v] is equal to an integer, then it is equal to the distance from s to v along some path.
- est[v] >= shortest path distance from s to v.

Table Size:

Table Size	Resize	Insert n items
Increment by 1	O(n)	O(n²)
Double	O(n)	O(n), avg O(1)
square	O(n²)	O(n)

Set

Fingerprint/ Cuckoo Hashing Size m, n elements	• A FHT does not store the key in the table • Only store 0/1 vector | P(false negative) = 0	
	P(NO false positive)	**P(false positive)**
	$\left(1 - \frac{1}{m}\right)^n \approx \left(\frac{1}{e}\right)^{n/m}$	$1 - \left(\frac{1}{e}\right)^{n/m}$
Bloom filter	• A Bloom Filter DON'T HAVE False Negatives - Use more than one hash function. - Redundancy reduces collisions. • Trade offs of using 2 hash functions: - Each item takes more "space" in the table - Requires 2 collisions for a false positive	
	P(NO false positive)	**P(false positive)**
	$\left(1 - \frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$	$\left(1 - \left(\frac{1}{e}\right)^{2n/m}\right)^2$

Implementation of Set ADT:

Insert, delete query	O(k)
Intersection	Bitwise AND of two Bloom filters: O(m)
Union	Bitwise OR of two Bloom filters: O(m)

Graphs

Undirected Graphs — Sparse → A.list; Dense → A.matrix

	Adjacency List	Adjacency Matrix
How	Graph consists of: Nodes: Stored in an array Edges: Linked List per node	Nodes Edges: Pairs of nodes Graph Represented as: A[v][w] = 1 iff (v, w) ∈ E
For a Cycle	• Memory usage - array of size |V| - linkedlists of size |E| • Total: O(V + E) • **For a cycle: O(V)**	• Memory usage - array of size|V|*|V| • Total: O(V²) • **For a cycle: O(V²)**
Clique	O(V + E) = **O(V²)**	**O(V²)**

Comparison:

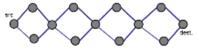
	Adjacency List	Adjacency Matrix
Are v & w n.brs?	Slower Query: O(n)	Fast Query: O(1)
Find any n.brs of v	Fast Query: O(1)	Slow Query: O(n)
Enum all n.brs	Fast Query: O(# n.brs)	Slow Query: O(n)

Directed Graphs:

Adjacency List	Adjacency Matrix
• Nodes: Stored in an array • **OUTGOING** Edges → Linked List per node	• Nodes • Edges: Pairs of nodes Graph Represented as: A[v][w] = 1 iff (v, w) ∈ E
• Array of nodes • Each node maintains a list of nbrs • Space: O(V + E)	• Matrix A[v,w] → edge (v, w) • Space: O(V²)

Searching

Breadth -First Search (BFS) **Uses Queue**	• Explore level by level • When does BFS fail to visit every node? → In graphs with 2 or more components (disconnected) • **Running time (For Adjacency list): O(V + E)**, has to look at every edge & node • Vertex v = "start" once. ← O(V) • Vertex v added to nextFrontier once. ← O(V) - After visited, never re-added. • Each v.nbrlist is enumerated once. ← O(E) - When v is removed from frontier. • **Running time (For Adjacency Matrix): O(V²)**
Depth-First Search (DFS) **Uses Stack**	• **Running time (For Adjacency list): O(V + E)** • DFS-visit called only once per node. ← O(V) - After visited, never call DFS-visit again. • In DFS-visit, each neighbor is enumerated. ← O(E) • **Running time (For Adjacency Matrix): O(V²)** • DFS-visit called only once per node. ← O(V) - After visited, never call DFS-visit again. • In DFS-visit, each n.brs is enumed. ← O(V) per node



O(2ⁿ) Note: Priority Queue is an overestimate of the distance of the path

Bellman-Ford Invariant:
After iteration k of the outer loop, every node whose shortest path from the source is ≤ k hops has a correct estimate

Dijkstra's:
- Maintain distance estimate for every node.
- Begin with empty shortest-path-tree.
- Repeat:
  - Consider vertex with minimum estimate. ** Keeps track of visited nodes
  - (We will show that this node has a good estimate 😊)
  - Add vertex to shortest-path-tree.
  - Relax all outgoing edges. ** Each edge is relaxed at most once

Relaxation: ** we may want to relax based on previous iterations

```
relax(int u, int v){
  if(dist[v] > dist[u] + weight(u,v))
    dist[v] = dist[u] + weight(u,v);
}
```

*Relax can be altered to suit our needs (create invariants of known algo)

- If $p = (v_0, v_1, \dots, v_k)$ is the shortest path from $s = v_0$ to v_k and we relax the edges of p in the order. Then $d[v_k] = \delta[v_k]$. $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- This property holds regardless of any other relaxation steps that occur (even intermixed) E.g $(v_0, v_1), (v_i, v_j), (v_1, v_2), \dots, (v_{k-1}, v_k)$ will result in $d[v_k] = \delta[v_k]$

Shortest Path (SSSP)

Bellman	$O(EV)$ • On the iᵗʰ iteration, nodes that are iᵗʰ hops away on the shortest paths have their estimates determined • |V| iterations of relaxing every age, terminate when an entire sequence of |E| operations have NO effect 1 Pass-Bellman Ford (Only for **DAGs**) → $O(E + V)$ • Topological Sorting: $O(E + V)$ • Iterating through all nodes and all edges: $O(E + V)$
Dijkstra no –'ve weights (depds on context)	$O((V + E) \log V) = O(E \log V)$ • Extending a path DOES NOT make it shorter • Uses a PQ to track min-estimate node, relaxes outgoing edges & add incoming nodes to PQ • |V| times to insert/deleteMin(logV each) → $O(V \log V)$ • |E| times of relax/decreaseKey (logV each) → $O(E \log V)$
DAGs	$O(E) \to toposort\ \&\ relax\ in\ this\ order$ • Longest Path: Negate weights/modify relax (then run SSSP)
Trees	$O(V)$, relax each edge in BFS/DFS order ** No cycles(undir)

PQ D.S	insert	delMin	↓key	Total
Array	1	V	1	$O(V^2)$
AVL Tree	$\log V$	$\log V$	$\log V$	$O(E \log V)$
dway Heap	$d\log_d V$	$d\log_d V$	$\log_d V$	$O(E \log_{E/V} V)$
Fib Heap	1	$\log V$	1	$O(E + V\log V)$

Topological Ordering (Start from in-degree = 0) **partial ordering CS1231S

The linear ordering of the vertices of a DAG s.t for every directed edge (u, v), vertex u comes before vertex v in the ordering.
*A graph that cannot be topo ordered → contains a cycle, not a DAG

Post-Order DFS O(V + E)	• Prepend each node from the post-order traversal • Output is in reverse order
Kahns O(E log V)	• Add nodes wo incoming edges to the topo order • Remove min-degree node from PQ → O(V log V) • decreaseKey (in-degree) of its children → O(E log V)
Kahns O(E + V)	• Uses Queue/Stack • Add nodes with in-degree = 0 to queue/stack • ↓ in-degree of its adjacent nodes, dequeue + repeat

Spanning Tree (any connected graphs with n vertices will have n-1 diff STs)

- Every edge is either red/blue. CANNOT BE BOTH

MST Algorithms
- Min edge in a **CYCLE** may/may not in MST
- Min edge in **CUT** MUST be in MST

1. MST has NO cycles
2. Any 2 subtrees of the MST are also MST (cut MST, both are MST)
3. For every cycle, MAX weight is NOT in MST (MIN may/may not be in MST)
4. For every partition of nodes, the MIN weight edge across the cut is IN MST

Prims O(E log V)	• Add the MIN edge across the cut to MST • Use PQ to store nodes (Prio lowest incoming edge w) • Insert,extractMin, decreaseKey → O(log V) • Each vertex: one added/removed→ O(V log V) • Each Edge: one decreaseKey → O(E log V)
Kruskals O(E log V)	• Sort edges by weight, add if unconnected • Sorting → O(E log E) = O(E log V) • For each edge: [O(E)] a) Get the two nodes connected by the edge b) Check if they are in the same tree/set [O(α(V))] i) If no, union the two [O(α(V))]
Boruvka's O(E log V)	• Each node: Store a componentID → O(V) • 1 Boruvka Step: for each cc, add min weight outgoing edge to merge cc's → O(V + E) dfs/bfs.

How to find MaxST?

Negative Weights O(ElogV)	1. Convert all weights to negative weights (i.e multiply by -1, **O(E)**) 2. Perform normal MST algorithms (e.g Prims/ Kruskal). **O(ElogE)** 3. MST of –'ve weighted edges → MaxST +'ve weighted edges
Reverse Kruskal O(ElogV)	• Modify Kruskal's algo to sort edges in descending order **O(ElogV)** • Then, add the edges to the tree in order of decreasing weight until all vertices are included. **O(ElogV)**
Max Prim O(ElogV)	• Modify Prim's algorithm to select the MAX weight edge to add to the tree at each step **O(ElogV)** • Use a PQ that sorts the edges in descending order of weight, and selecting the edge with the MAX weight at each step. **O(ElogV)**

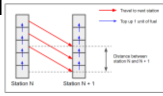
Note: Finding the LONGEST path (i.e opposite of SSSP) is NP hard (e.g travelling salesman O(n²log n)
However, if graph is a DAG → LSSP can be found via modified topo sort/ negate edges → O(V + E)

Bounded Integer Weights

Kruskal Variant	• Use an **array** of size n to **sort** • Slot A[j] holds a linked list of **edges** of weight j • Putting edges in array of linked lists: O(E) • Iterating over all edges in ascending order: O(E) • For each edge: - Checking whether to add an edge: O(α) - Union two components: O(α) • Total Time Complexity: O(αE)
Prim's Variant	• Use an **array** of size n as **Priority Queue** • Slot A[j] holds a linked list of **nodes** of weight j • decreaseKey: O(E) • Insert/remove nodes from PQ: O(V) • Total Time Complexity: O(E)

How to find patient zero?
Indegree = 0

SP of mod x length:
Matri Multi + BFS
O(V³) + O(V²) = O(V³)
OR
Make x copies of graph
For every edge (a, b) in the original G, draw an edge from node a in G_i to node b in $G_{(i+1)\%x}$, then BFS
O(V + E)



Graph Hacks: **Checking for cycles is SLOW → BF → O(VE)
- Add dummy node if you need to find min/max across all paths
- Find longest paths & MST in DAG: O(V + E)
- If there is a monotonically ↑/↓ property, can use binary search
- To find min path across diff start pt, but same end pt → reverse edges
- Graph duplication → Forces edges to follow a certain path
- Each layer in the graph → 1 graph state; Each Edge → transition state
- IF edge weights are bounded integer → Use buckets (e.g Kruskal/prims var.)

Dynamic Programming

Optimal sub-structure	Overlapping sub-problems
• Optimal solution derived from smaller subproblems	• Memoization • Divide & Conquer

Longest Increasing Subseq	• Greedy subproblems: S[i] = LIS(A[1..i]) - n subproblems - SubProb i takes O(i) time • Total time: O(n²)	Patience Sort (via B.Search) • Best Case: O(n) • Worst Case: O(n log n)
Longest Common Subseq	$IF\ A[n] == B[n] \to LCS(A(n), B(n) = LCS(A(n-1), B(n-1)) + 1$ $Else \to \max\{LCS(A(n), B(n-1)), LCS(A(n-1), B(n))\}$ If Length A = m, Length B = n → O(mn)	
(Lazy) Prize Collecting	• No. of rows: k • Cost to solve each row: E Total: O(kE)	• No. of subproblems: kV • Cost to solve each subproblem: |v.nbrList| Total: O(kV²)
	$P[v, k] = \max\{P[w, k-1] + Weight\ of\ edge(v, w) \mid w\ is\ n.br\ of\ v\}$	
Lazy Prize kV subProb	• Topo-sort/ Longest Path → O(kV + kE) = O(kE) • Once per source: repeat V times → **O(kVE)**	• Topo-sort/ Longest Path → O(kV + kE) = O(kE) • Create dummy node → only ONE SP → **O(kE)**
Vertex Cover on a Tree (Min) augment for Max	• 2V sub-problems • O(V) time per sub-problem → O(V²) time - Each edge explored once. - Each sub-problem involves exploring children edges. ** Similar to DFS	
APSP Non DP	Running time of running Single Source Shortest Path(SSSP) algorithm for every vertex in V: Bellman Ford: O(V²E) Dijkstra's: O(VE log V)	
Diameter of Graph	SSSP all → O(V² log V) Longest shortest path between any two vertices in the graph	
APSP DP	• In a weighted sparse graph where E = O(V): O(V²log V) • In an unweighted graph, use BFS: O(V(E+V)) - dense graph: O(V³); sparse graph: O(V²) Floyd-Warshall: O(V³) **V³ total subproblems • Subproblem: find shortest path btwn every pair of nodes in the graph that goes through a specific intermediate node $S[k+1][v][w] = \min\{S[k][v][w], S[k][v][k+1], S[k][k+1][w]\}$ • $S[v, w, P_k]$= shortest path from v to w only with nodes in set P.	
Transitive Closure	• Return a matrix M where: - M[v, w] = 1 if there exists a path from v to w; - M[v, w] = 0, otherwise. $M[k+1][v][w] = OR(M[k][v][w], AND(M[k][v][w], M[k][k][w]))$	

1. DAG + toposort (Cyclic → dependencies, cannot solve with DP)
2. Top-Down(Recursion)
3. Bottom-Up

- **Connected** is usually associated with undirected graphs (two-way edges): there is an **edge** between every two nodes.
- **Strongly connected** is usually associated with directed graphs (one-way edges): there is a **path** between every two nodes. (cycles)
- **Complete** graphs are undirected graphs where there is an **edge** between every pair of nodes.

Prims	• Initially: S = {A} ** Keeps track of the set of edges • Repeat: I) Identify cut: {S, V–S} II) Find MIN weight edge on cut. • Add new node to S.
Kruskal	• **Sort edges** by weight from smallest to biggest. • Consider edges in ascending order: - If both endpoints are in same blue tree, color the edge red. - Else, color the edge blue.
Boruvka	• Add all blue edges, for each CC, search for MIN weight outgoing edge • Merge the newly connected components into a single CC • Repeat (At most O(log V) Boruvka steps.)