Pre-empts	
Restrictions for CS2040S	Avoid Java “advanced” features like Lambda expressions, Type inferences (var), Default, static private methods in an interface… Do not use libraries if they make the problem set easier Do not use libraries unless the problem set specifically says you can
Considerations	Only makes code shorter Little extra functionality Often hide what is really happening May or may not make code harder / easier to read
Goal of writing code in CS2040S	Correct / Bug-Free Easy to read / understand Efficient Submitted by deadline Short
Advice	Make your code intentional (Avoid default / non-explicit behaviours)

OOP Paradigm (Same concept as CS2030S, different applications)	
Abstraction	User-centric, information on a need-to-know basis, hide implementations where possible
Encapsulation	Group methods and data in a class meant to represent something (noun), Hiding implementation and only interface publicly visible.
Inheritance	Build new classes by extending existing classes (Sharing and Adding functionality)
Polymorphism	Same interface, but different behaviour based on context

Application of OOP Paradigm (for Algorithmic Design)	
Application	Divide Problem into Components Define interface between components Solve each problem separately Repeat, then combine solutions
Abstraction	Interface: how you manipulate the object Implementation: details hidden inside the object
Encapsulation	Class as a template for producing an object Grouping functionalities to solve a subset of problems

Principles of Java	
First Principle	Everything is an Object
Second Principle	Everything has a Type

Classes & Objects, Regular & Static	
Classes	Template for how to make an object
Objects	An instance of the class
Constructors	Creates and instantiate the object and its fields
Parts of an Object	State (data), Behaviour (methods for modifying the state)

Regular vs Static	Regular Variables/Functions are PER OBJECT Static Variables/Functions are PER CLASS

Access Control	
(none specified)	Within the same package
public	Everywhere
private	Only in the same class
protected	Within the same package, and by subclasses
Advice: Always specify the access you intend (even if the default is okay)	

Java Operators	
=	Assignment
+, -, *, /	Plus, minus, multiplication, division
%	Remainder, Modulo
++, --	Increment, decrement
<, >	Less than, greater than
<=, >=	Less-than-or-equal, greater-than-or-equal
&&, ||	Logical and, logical or
~, &, ^, |	Bitwise operations: complement, and, xor, or

Primitive Data Types			
Byte	8 bit	-2^7	2^7 - 1
Short	16 bit	-2^15	2^15 – 1
Int	32 bit	-2^31	2^31 - 1
Long	64 bit	-2^63	2^63 - 1
Float	32 bit (IEEE 754)	(2 – 2^23) *	
Double	64 bit (IEEE 754)	+-(2 ^ -1074)	+-((2 - 2^-52) * 2^1023)
Boolean	1 bit	False	True
Char	16 bit (Unicode)	\u0000 (0)	\uffff(65535)

Algorithm Analysis (Big O notation)	
Pre-empt	Take Logs to be Base 2, Loga(n) = Log2(n)/log2(a) Always think big inputs
Big-O Notation $T(n) = O(f(n))$	$\exists c > 0 \land \exists n_0 > 0 \rightarrow \left(\forall n > n_0 (T(n) \leq cf(n))\right)$
Big-Ω Notation $T(n) = \Omega(f(n))$	$\exists c > 0 \land \exists n_0 > 0 \rightarrow \left(\forall n > n_0 (T(n) \geq cf(n))\right)$
Big-Θ Notation $T(n) = \Theta(f(n))$	$T(n) = O(f(n)) \land T(n) = \Omega(f(n))$
Order of Size:	(see table below)
Summation	$T(n) = O(f(n)) \land S(n) = O(f(n))$ $\rightarrow T(n) + S(n) = O(f(n) + g(n))$
Product	$T(n) = O(f(n)) \land S(n) = O(f(n))$ $\rightarrow T(n) * S(n) = O(f(n) * g(n))$
Sterling’s Approximation	$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
Sequential Statements	$cost = cost_{first} + cost_{second}$
If / else statements	$cost = max(cost_{first}, cost_{second}) \leq cost_{first} + cost_{second}$
Recursion (Summations)	Geometric Sum: $\sum_{k=0}^{n} ar^k = \begin{cases} a(n+1) & a = 1 \\ a\left(\frac{1-r^{n+1}}{1-r}\right) & otherwise \end{cases}$ Arithmetic Sum: $\sum_{i=1}^{n}(a + d * i) = \frac{n(2*a+(n+1)d)}{2}$ Master Theorem: where, $T(n) = aT\left(\frac{n}{b}\right) + f(n), a \geq 1, b > 1$ $T(n) = \begin{cases} \theta(n^{\log_b a}) & f(n) = O(n^{\log_b a - \epsilon}) \\ \theta(n^{\log_b a} \times \log n) & f(n) = \theta(n^{\log_b a}) \\ \theta(f(n)) & f(n) = \Omega(n^{\log_b a + \epsilon}) \end{cases}, \epsilon > 0$

Order of Size:

Function	Name	Function	Name
5	Constant	n^3	Polynomial
loglog(n)	Double Log	$n^3\log(n)$	
log(n)	Logarithmic	n^4	Polynomial
$\log^2(n)$	Polylogarithmic	2^n	Exponential
n	Linear	2^{2n}	
nlog(n)	Log-linear	n!	Factorial

Searching Algorithms	
Characteristics	Runtime
Search Algorithm	Linear Search: O(n); Binary Search: O(log(n)) Quick Search: O(log(n))
Linear Search	Check all elements
Binary Search	Check mid element, compare with required value, Must be searching an ordered array
Quick Search	Check Relative Positioned element, compare with required value, Must be searching an ordered array
Precondition	Fact that is true when the function begins Something important for it to work correctly Useful to validate when possible
Postcondition	Fact that is true when the function ends Something useful to show that the computation was done correctly
Invariant	Relationship between variables that is always true
Loop Invariant	Relationship between variables that is true at the beginning (or end) of each iteration of a loop
Peak Finding	Find Local Max, Binary Search, Check slope direction Invariants: There is a peak in the range [begin, end] Every peak in [begin, end] is a peak in [0, n-1]
Steep Peaks	Steep peaks are strictly larger than its neighbours
2D Peak	2D Peak: Larger than or equal to its neighbours Find Max of All Col, then find peak (O(mn + log(m))) Find Local of All Col, then find peak (incorrect) Find Max of Mid Col, then recurse (O(nlog(m)) Find Max of Border + Cross, then recurse in quadrant where (the neighbour of Max) > Max (O(n+m))

Sorting Algorithms	
Characteristics	Runtime, Space, Stability, Worst Cases
Runtime	Best Case (Ω), Average Case (Θ), Worst Case (O, Impt)
Space	Total Space ever allocated

	(Alt, Realistic: Max Space allocated at one time)
Stability	Preserves order of equal elements
Bogo Sort	Randomly permutate array, check if sorted, O(n*n!), Unstable, All cases are worst cases
Quantum Bogo Sort	Generate permutation and check the array if sorted, destroy universe if not, If many-worlds interpretation holds, there exists a surviving universe where array is sorted, O(n)
Bubble Sort	Iterate through the array, swap if greater than next element, loop first n-1 element Invariant: Largest k element sorted at k loops $\Omega(n)/O(n^2)$, In-Place, Stable, Reversed / Circular Left Shift
Selection Sort	Iterate through the array, swap minimal element to the front, loop last n – 1 element Invariant: smallest k element sorted at k loops $\Omega(n)/O(n^2)$, In-Place, Unstable, All cases are worst cases
Insertion Sort	Take first element, swap insert into sorted array at the front, loop to next unsorted element Invariant: smallest k element sorted at k loops $\Omega(n)/O(n^2)$, In-Place, Stable, Reversed
Merge Sort	Split array in half, recurse halves, merge in order Invariant: Subarrays are sorted at end of loop Θ(nlog(n)), Space: $\Omega(n)/O(n^2)$ by implementation, Stable (check merge), All cases are worst case
Ingrassia-Kurtz Sort	Generate all permutations, sort permutations, return first element in the sorted list of permutations
Quick Sort	Partition the array on pivot by swapping bigger elements on the left with smaller elements on the right, then recurse Invariant (Partition): for every i < low, A[i] < pivot, for every j > high, A[j] > pivot Runtime dependent on pivot selection, In-Place, Unstable, All cases are worst case Runtime: 1st elem = $\Omega(n^2)$, Median elem = O(nlog(n)), 1/10+9/10 = O(nlog(n))
Quick Sort (Duplicate)	3-Way Partitioning: 1) Two Pass: Regular Partition then Pack Duplicates 2) One Pass: More Complex, Maintain four regions of array <pivot, =pivot, in-progress, >pivot (4 pointers)
Paranoid Quick Sort	Randomise pivot index selection Θ(nlog(n)) Runtime

Data Structure Design	
Data Structure	A way of storing and organizing data efficiently, such that required operations can be performed efficiently with respect to time as well as memory Considerations: Maintenance, Modification, Query Upgrades: Augmentations, New Properties
Static Data Structure	Size of Structure is fixed; Content can be modified but without changing memory space allocated to it Eg. Array, Stack, Queue, Fixed Size Tree
Dynamic Data Structure	Size of Structure is not fixed and can be modified during the operations performed on it Eg. Lists, Trees, Tries, Hash Tables

Augmenting Data Structures	1) Choose underlying data structure 2) Determine additional info needed 3) Modify data structure to maintain additional info when structure changes 4) Develop new operations
Order Statistics	Preprocessing, Accessing, Modifying, Postprocessing

Tree Data Structure	
Idea	Given a dictionary, storing key-value pairs Possible Implementations Sorted Array } insert: O(n), (binary) search: O(log(n)) Unsorted Array } insert: O(1), search = O(n) Linked List } insert: O(1), search = O(n) Balanced BS Trees } insert: O(log(n)), search = O(log(n))
Trees	Components: Nodes (1 Root), Edges, No Cycles
Binary Trees	Empty or A node pointing to two binary trees
BST	Keys in left sub-tree < key < Keys in right sub-tree
Root	The base node, all search/insert start here
Leaf	No children, Height = 0, Weight = 1
Siblings	Nodes that share a parent
Height	-1 if null, 0 if leaf, else max(h(v.left), h(v.right)) + 1
Weight	0 if null, 1 if leaf, else w(v.left) + w(v.right) + 1
Rank	r(leftparent) + r(v.left)
Shape	Same keys != Same Shape, affects performance, determined by order of insertion of nodes # orders: n!; # shapes: ~4^n (Catalan)
Tree Traversal	Pre-Order, In-Order, Post-Order, Level-Order Order of visited nodes

Binary Search Tree (BST)	
Description	Keys in left sub-tree < key < Keys in right sub-tree
Applications	Max/min, rank/select, successor/predecessor operations
Search	At each node, compare node key, go to key direction
Insert	Search, then add at null
Delete	No child: remove v 1 child: remove v, connect child(v) to parent(v) 2 child: x = successor (v), delete(x), remove v, connect x to left(v), right(v), parent(v)
Successor / Predecessor	Successor: Get right child left most node, else left parent Predecessor: Get left child right most node, else right parent
Runtime Summary	Insert, delete, search, predecessor, successor, findMax, findMin: O(h); in-order-traversal: O(n)
Balanced	h = O(log(n)), for Balanced BST: all operations are O(log(n))
Getting a Balanced Tree	1) Define good property of a tree 2) Show that if the good property holds, then the tree is balanced 3, Invariant) After every insert/delete, make sure the good property still holds, If not, fix it
AVL Tree	Adelson-Velskii & Landis 1962 Tree

	Step 0, Augment: every node v, store height Update on insert/delete operations Step 1, Define Height Balance: node v is height-balanced if $\lvert v.left.height - v.right.height \rvert \le 1$ Binary Search Tree is height balanced if every node in the tree is height balanced / # keys in heavier sub-tree at most twice of # keys in lighter sub-tree Step 2, Maintain Height Balance: Tree Rotation
Claim	A height-balanced tree with n nodes has at most height h = O(log(n))
Tree Rebalancing: Tree Rotation	Right Rotation on Node v: v.left = vLeft.right, vLeft.right = v Left Rotation on Node v: v.right = vRight.left, vRight.left = v Maintains ordering of keys => Maintains BST Property
LR/RL-Heavy	Left-Right-Heavy: Left Rotate Left Child, then Right Rotate Right-Left-Heavy: Right Rotate Right Child, then Left Rotate
Insert in AVL	Insert key in BST, then walk up tree and check for balance Only need to fix lowest out-of-balance node Only at most 2 rotations to fix
Delete in AVL	If v has 2 children, swap with successor Delete node v and reconnect children Check every ancestor of deleted node for height-balance At most O(log(n)) rotations to fix

Trie / Dictionary	
Description	Trees where nodes can have many children Used for storing address and words (ie Dictionary)
Root-to-Leaf Path	Represent Strings ie Keys
Terminating Character	Marks the end of String ie Keys
Space Required	O((size of text)*overhead)
Search	O(L) where L is length of string
VS Trees	Shorter Runtime, Bigger Space, No Ordering
Trie Nodes	Many Children, for Strings: Fixed degree (ASCII: 256)

Hash Table / Symbol Table	
Description	Store Key-Value Pairs by putting them into the key's hashcode mapped with the table's unique hashing function
Keys	Should have no duplicate and be immutable
Duplicate Key Handling	Replace existing key / Add new value (ie key has 2 values) / throw error
Empty / Null Value Handling	Delete existing (key, value) pair / Create a null value / throw error
Insert / Put	Insert (Key, Value) into table
Search	Get value paired with key
Delete	Remove key and value by key
Contains / Get	Check if there is a value for key
Size	Number of (Key, Value)
(Suc/Prede)cessor	Does not Exist

Hash Function	Random mapping a small number n of keys out of a huge universe U of possible keys into m ≈ n buckets $h: U \to 1 \dots m$, store key k in bucket $h(k)$ Time to compute h and access bucket $\approx O(1)$
hashCode()	Every object supports the method int hashCode() $A = B \to A.\text{hashCode}() = B.\text{hashCode}()$
hashCode Truncation	Usually, table size is 2^n, so we just get the first n least significant bits, code: hashcode & (length-1)
hash in HashMap	Further differentiate keys with the same hashCode truncation
equals(Object o)	Should match behaviour of hashcode Equivalence relationship, o = null gives false
String hash	$\sum_{i=0}^{n-1} s[i] \times 31(n - i + 1)$, 31 is prime, $2^5 - 1$ easy to compute
SUHA	Simple Uniform Hashing Assumption Every key is equally likely to map to every bucket & keys are mapped independently Assume n items, $m = \Omega(n)$ buckets, e.g., $m = 2n$ Search Time: $O(n) + n/m = O(1)$
Key Collisions	Inevitable by Pigeon-Hole Principle
Hashing with Chaining	Use linked list to store multiple keys in one bucket Searching: $\theta(1), O(n)$ Inserting: $O(1)$, Inserting n items: $O(\log n / \log \log n)$
Hashing with Open Addressing	Collided items are still inserted into the table directly
Linear Probe	Probe next location if current is filled, put in next available slot, *deleted items hold tombstone value Better than Chaining in practise due to: Caching & Prefetching
Deletion in Linear Probing	1) Just probe entire table during search: expensive 2) Tombstoning: need handle too many items deleted 3) Replace with another element further down
Re-hashing	Regenerate the whole table, happens when: Too many tombstones ($n/4$ items in n size $\to 2n$ size) No more space in table (n items in n size $\to 2n$ size) Time: $O(m_1 + m_2 + n) = O(n)$, as $m_1 = an, m_2 = bn$
Amortised Cost	If an expensive operation (re-hashing: O(n)) is reliant on and relative to cheaper operations (insert/delete: O(1)), we may spread the cost of the expensive operation to the cheaper operation, ie make expensive free
Amortised Analysis	Single Request (Risks Tail Latency), no spike in runtime / no single expensive operations: Tree Batch of n Requests, prefer higher throughput: Hashtable

Heap & Priority Queues	
Considerations	Better insert / extractMax function than trees, allow implementing new operations like merge / split Can be converted from an array in O(n) time, heapify

Binary Heaps Invariant	Priority at each node < parent priority: help find max Complete Binary Tree, filled from left to right: maintain O(log n) height
Binary Heap Components	"Tree" in an array: represents the heap, a[0] = size $childIdx = 2 \times parentIdx + 0/1$ Hashtable: map ID to node indices Array: map indices to IDs
Bubble Up	Recursively swap with parent if priority is bigger than parent
Bubble Down	Recursively swap with larger child if priority is smaller than larger child
Insert	Put in next free spot, then bubble up
Delete	Swap with last element, last element bubble down
Decrease Key	Simply bubble down
Heapify	Heapify from right to left, bottom to top Amortised O(n) time
Heapsort	Convert heap to sorted array by recursively swap root with last element then bubble down root node

Graphs	
Components	Nodes & Edges (connecting 2 nodes)
Simple Graphs	Unique ie no 2 edges share the same start & end nodes No self-loops
Multigraph	Simple graph with non-unique edges
Hypergraph	Edges contain more than 2 nodes, Unique Edges
Undirected Graph	Edges are bidirectional,$(i, j) \in E \leftrightarrow (j, i) \in E)$
Directed Graph	Edges are directional,$\exists i, j \in V\big((i, j) \in E \land (j, i) \notin E\big)$
Sparse Graph	$E \approx O(V)$
Dense Graph	$E \approx O(V^2)$
Path	Set of edges connecting 2 nodes, no repeated node
Connected	Every pair of nodes is connected by a path
Cycle	"Path" where first and last node are the same
Tree	Connected Graph with no cycle
Forest	Graph of Tree components
Degree of Node	Number of adjacent edges
Degree of Graph	Maximum degree of all nodes in graph
Diameter	Distance of maximum shortest path between 2 nodes
Star	One central node, all edges connect centre to edge nodes
Clique	Complete graph, all pairs connected by edges
Self-Explanatory	Line, Cycle
Bipartite Graph	Nodes divided into two sets with no edges between nodes in the same set
Application of Graphs	Implemented in connecting different states or representing networks
Rubik's Cube	Diameter of (n x n x n) cube = $\theta(n^2 / \log n)$
Representing Graphs	Adjacency List: Array (node) of linked list (edges) Adjacency Matrix: ij = edge from i node to j node Edge List: List of node pairs (edges)

Adjacency List	Get All Neighbours: O(deg(v)) Get if node x & y are neighbours: O(min(deg(x), deg(y))) Space $= O(\|V\| + \|E\|)$
Adjacency Matrix	$A[v][w] = 1 \leftrightarrow (v, w) \in E$ Symmetric for undirect graph $A^n = $ # of length n paths Get all neighbours: O(V) Get if node x and y are neighbours: O(1) Space $= O(V^2)$
Pagerank Vector	Vector that describes the distribution of nodes Eigenvector of matrix with eigenvalue 1 $\approx A^{\infty}$

Searching Graph	
Description	Start at some vertex, end at some other vertex
Methods	BFS/Breadth-First Search & DFS/Depth-First Search
BFS	Search level[i] from level[i-1] Pseudocode: 1) Set queue to contain only source node 2) while queue is not empty a) Take next node out of queue b) Go through all neighbours of node c) If updated, skip. Else, update info and enqueue 3) Finish when queue is empty, review info Run Time: O(V + E) with adjacency list Gets shortest path graph (a Tree)
DFS	BFS but with a stack instead of queue Run Time: O(V + E) with adjacency list, O(V^2) with matrix Cannot get shortest path
Handling Disconnected Graph	When queue/stack is empty, check if all nodes are visited, then, if necessary, continue search on an unvisited node
Topological Ordering	Ordering where $(u, v) \in E \to u$ appears after v Only for Directed Acyclic Graph (DAG), not unique
Topological Sort	DFS, add node to the end if it has no unvisited child
Pre-Order DFS	Process each node when it is first visited
Post-Order DFS	Process node when it is last visited (Toposort)
Strongly Connected Component	Forms a cycle in a graph, two nodes v, w are reachable to/from each other Graph of SSC is acyclic
Cycle Finding	Found with DFS, at each node u visit: 1) update u time 2) for all v neighbour to u, update u low time a) if v has time, no low time, consider time b) if v has no time, recurse on v, consider low time 3) if u low time = u time, u belongs to acyclic graph of SSC / if u low time <= u time, u belongs in a cycle
Articulation Point	Removing this node disconnects the graph If u low time < u time, it is not an articulation point Repeat cycle finding on SSC Graph until no changes
Bridge Edge	Connects two articulation points

Single Source Shortest Paths (SSSP)	
Cases	Unweighted Graphs: BFS Weighted Non-negative Acyclic Graphs: Dijkstra Weighted Acyclic Graphs: Bellman Ford
BFS for SSSP	Graph level = SSSP distance, connect next node to node, ignore edges pointing to visited next node
Dijkstra	BFS but visit lightest unvisited node each time Pseudocode 1) Initialise minimum priority queue and graph with start node and all other nodes with priority as 0 and Max priority respectively 2) While the PQ is not empty a) get minimum node from PQ b) relax adjacent nodes and if needed, parentNode 3) Recursively trace parentNode from endpoint for SSSP Run Time: $V * O(log\, V) + E * O(log(V)) = O(E\, log\, V)$
Dijkstra Invariant	Remove each node from PQ at most once Decrease priority of node v in PQ at most in-deg(v) Visited nodes have smallest distance at the end Unvisited nodes have estimated smallest distance > their final smallest distance
Relax node	Lower distance estimate if new distance is lower
Triangle Inequality	For Dijkstra, $\delta(S, C) \leq \delta(S, A) + \delta(A, C)$
Bellman-Ford on general	Spam relax with every edge for the number of nodes, works since most number rounds of relaxation is the diameter of graph < number of nodes Pseudocode 1) Initialize array for distance, 0 for start, max for all others 2) for |V|-1 iterations: a) for edge (u, v) in the graph: i) relax(arr, u, v) b) stop if no changes / relaxation made Run Time: O(VE)
Negative Cycle Detection	If distance array still changes at |V| number of relaxations, there is a negative cycle
Bellman-Ford with toposort on DAG	Pseudocode: 1) Set up distance estimate array 2) Get toposorted list of nodes topo_list 3) for u in topo_list: a) for neighbour v in u.neighbour_list: i) relax(dist, u, v) Run Time: O(V + E)
Multiple Sources	Add super node with directed 0 weight edge to sources
Creating State Space	Creating layers / connections between layers to force SSSP to go certain directions

	ie Encode many things about your traversal through the graph
Shortest Path at exactly k edges	K + 1 copies / layers of graph, edges points to next node in the next layer

Union Find	
Description	Simplifies the isConnected query after graph search
Pre-processing	Set up another data structure or graph augmentation
Union	Connect two objects
Find	Gets if any path connecting the two objects
Version 1	Component identity array, union by updating all objects in the child component to have the new parent component identity Expensive union: O(n), Cheap find: O(1)
Version 2	Parent pointer array, union by updating root node's parent to the other root node Expensive union: O(n), Expensive Find: O(n)
Weighted Union	Same as Ver 2, but specify union to update lighter tree's root node's parent to heavier root node Cheap union: O(1), Cheap? Find: O(log(n))
Path Compression	After finding root, set parent of each traversed node to the root
Weight Union with Path Compression	Any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$

Minimum Spanning Tree	
Definition	Spanning tree with minimum weight Property 1: No cycle Property 2: If an MST is cut, the two pieces are MST Property 3 (Cycle Property): For every cycle, the maximum weight edge is not in MST Property 4 (Cut Property): For every partition of the nodes, the minimum weight edge across the cut is in the MST
Spanning Tree	Acyclic subset of edges that connects all nodes
Caution	Cannot find shortest path
Generic MST Algorithm	Red Rule: If C is a cycle with no red edges, then colour the max-weight edge in C red Blue Rule: If D is a cut with no blue edges, then colour the min-weight edge in D blue Greedy Algorithm: Repeat apply red rule / blue rule to an arbitrary edge until all edges are either blue or red
Wrong MST Algorithm	Divide-and-Conquer 1) If the number of vertices is 1, then return 2) Divide the nodes into two sets 3) Recursively calculate the MST of each set 4) Find the lightest edge that connects the two sets and add it to the MST
Kruskal's Algorithm	Add edges to MST Pseudocode:

	1) Initialise UFDS for n nodes, all initially disjoint 2) sort edges by weights in ascending order 3) for each edge e = (u, v) a) skip if u and v are in the same component b) add edge in union u and v component Run Time: O(E log(E)) =O(E log(V)) for sorting edges O(E a(E)) for find and union vertexes
Prim's Algorithm	Add nodes to MST Pseudocode: 1) Set min-pq to contain only source node 2) while min-pq is not empty a) take next node out of min-pq b) if node has not been added before, include the edge used into the MST, otherwise skip c) go through all neighbours n of node d) if edge has weight w, insert n into min-pq with priority w Run Time: O(V log(V)) for extracting every vertex once + O(E log(V)) for decreasing key = O(E log(v))
Prim's Algorithm Variant	Start min-pq with all nodes with priority infinity, and source node with priority 0. When we process a node's neighbours, decrease key the neighbour if the new edge weight is smaller than its current priority
Kruskal's & Prim's Algorithm Variant	If edges have weights from {a..b}, 0 < a < b Linked list array of size b-a as a "Priority Queue" Kruskal's: Put & iterate all edges: O(E), union-find each edge: O(aE), Total = O(aE) Prim's: Insert/Remove: O(V), decreaseKey: O(E) Total: O(V + E) = O(E) Variant fails in Dijkstra since Dijkstra holds total distance and not smallest edge, variant only work if we know the maximum distance
Directed MST	For every node except root, add minimum weight incoming edge Runtime: O(E)
Maximum ST	Negate the weights

Dynamic Programming	
Description	Used for problems with overlapping subproblems
Optimal Substructure	Optimal solution can be constructed from optimal solutions to smaller sub-problems
Overlapping Subproblem	The same smaller problem is used to solve multiple different bigger problems in an optimal substructure
Dynamic Programming Recipe	1) Identify Optimal Substructure 2) Define subproblems 3) Solve problem using subproblems 4) Write pseudocode

Dynamic Programming Analysis	1) Count subproblems 2) Figure out total time taken to solve all subproblems Hint: Often times, it is just # subproblems x time per subproblem
Basic Strategy 1: Bottom Up Dynamic Programming	1) Solve smallest problem (ie base case) 2) combine smaller problems to bigger problems 3) solve bigger problems 4) recursively solve upwards to the root problem
Basic Strategy 2: DAG + topological sort	1) Topologically sort DAG 2) Solve problems in reverse order
Basic Strategy 3: Top down Dynamic Programming	1) Start at root and recurse 2) Recurse down until base case 3) Solve & memorize, compute each solution once
Longest Increasing Subsequence	Strat 1: Subproblem: S[i] = LIS(A[i...n]) start at A[i] Solve: S[n] = 0, S[i] = $\left(max_{(i,j)\in E} S[j]\right) + 1$ Run Time: O(n^2) Strat 2: Toposort (alr done), find longest path Run Time: V(O(V + E)) = O(n^3)
Prize Collecting	Check for positive weight cycle first, else... Strat 1: subproblem: $P[v,k] =$ max prize starting at v at k steps Solve: $P[v,0] = 0$, $P[v,k] = max_{i=1}^{n}(P[w_i + w(v,w_i)])$, $v.nbrList() = \{w_1 \dots w_n\}$ Run Time: O(kV^2) Strat 2: Transform G into DAG by making k copies Solve for longest path with DAG_SSSP Run Time: O(kV + kE)
Vertex Cover on a Tree	Strat 1: Subproblem: $S[v,k] =$ size of vertex covers in subtree rooted at node v, if v is covered , $k = 1$ or not, $k = 0$ Solve: $S[v,0] = \sum(\forall n \in \text{v's neighbours } S[n,1])$ $S[v,1] = \sum\left(\forall n \in \text{v's nbrs } \min(S[n,0], S[n,1])\right)$ Run Time: O(V)
All Pairs Shortest Path	Simple Strat 1: no preprocessing Preprocessing: 0, q queries: O(qE log(V)) Simple Strat 2: For every node v, run SSSP, then store distance to every other node Preprocessing: O(VE log(V)), q queries: O(q) Floyd-Warshall: if P is shortest path (u to v to w), then P contains shortest path (u to v) and (v to w) Subproblem: $S[v,w,P]$ be shortest path (v,w) using only intermediate nodes in set P Solve: $S[v,w,\emptyset] = E[v,w], S[v,w,P_i] = min(S[v,w,P_{i-1}], S[v,i,P_{i-1}] + S[i,w,P_{i-1}])$ Run Time: O(V^3) Space: O(V^2) with routing table, where M(v,w) is weight of minimum bottleneck
Knapsack (max value without exceeding limit)	Subproblem: Value(S, L): max attainable value using items from set S not exceeding L Solve: Value(S, 0) = 0, $Value(S,L) = max(Value(S\backslash\{S_n\},L),$
	$Value(S\backslash\{S_n\}, L - w) + v)$ If $S_n > L, Value(S,L) Value = S\backslash\{S_n\}$ Run Time: O(nL)