Algorithm Analysis

Upper bound: $T(n) = \boldsymbol{O}(f(n))$ if $T(n) \leq cf(n)$ for all $n > n_0 > 0$ and $c > 0$

Lower bound: $T(n) = \boldsymbol{\Omega}(f(n))$ if $T(n) \geq cf(n)$ for all $n > n_0 > 0$ and $c > 0$

Exact: $T(n) = \boldsymbol{\Theta}(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

General rule: $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

Order of growth: 1 < log(log(n)) < log(n) < $\log^2$(n) < n < nlog(n) < n^2 < n^3 < n^3log(n) < n^4 < 2^n < 2^{2n} < n!

Rules:

1. If $T(n)$ has a polynomial degree of k then $T(n) = O(n^k)$
2. If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then $T(n) + S(n) = O(f(n) + g(n))$
3. If $T(n) = O(f(n))$ and $S(n) = O(g(n))$ then $T(n) \times S(n) = O(f(n) \times g(n))$
4. $\max(f(n), g(n)) = O(f(n) + g(n)) = O(n)$
5. $f(g(n)) \leq c_1 g(n)$
6. $f(n)^{g(n)}$: $O\left((n^{\log n})^2\right)$ constant matters in the exponent

String concatenation in loop: $O(n^2)$ performance

Common recurrence relations:

T(n) =T(n-1) + O(1) = O(n)	T(n) =T(n-1) + O(n) = O(n^2)
T(n) =T(n/2) + O(1) = O(logn) (Binary search)	T(n) =T(n/2) + O(n) = O(n) (Quickselect)
T(n) =kT(n/k) + O(1) = O(n)	T(n) =kT(n/k) + O(n) = O(nlogn), (Quicksort) (if O(log n) then $O(\log^2 n)$)
T(n) =T(n-1) + T(n-2) = O(2^n) (Fibonacci)	T(n) =T(n-1) + O(nk) = O(nk+1)
T(n) =T(n-1) + O(logn) = O(nlogn) (Stirling Approx) O(log(n!)) = O(nlogn)	T(n) =2T(n-1) + 1 = O(2^n)
T(n) =2T(n/4) + O(1) = O($n^{0.5}$)	n^b > (logn)^a always
n^b < a^m always	O($n^{9.9}$) < O(1.01^n)
$\sqrt{n} \log n = O(n)$	

Master theorem: given $T(n) = aT\left(\frac{n}{b}\right) + cn^k$ and $T(1) = c$, then if

$$T(n) \in \begin{cases} \Theta(n^k) \; if \; a < b^k \\ \Theta(n^k \log n) \; if \; a = b^k \\ \Theta\left(n^{\log_b a}\right) \; if \; a > b^k \end{cases}$$

AP: $S_n = \frac{n}{2}(2a + (n-1) \times d) = \frac{n}{2}(first\ term + last\ term)$

GP: $S_n = \frac{a(r^n - 1)}{r - 1} = \frac{a(1 - r^n)}{1 - r}$

Binary search O(log n)

Precondition: array is of size $\boldsymbol{n}$, array is sorted

Postcondition: if element is in array, A[begin] = key

Invariant: (end – begin) $\leq \frac{n}{2^k}$ in iteration k

Loop invariant: A[begin] ≤ key ≤ A[end]

Peak Finding O(log n)

Peak exists to the left/right where value is increasing; cut search space in half using this

Limitations:

- Cannot find global peaks efficiently (must use $O(n)$ solution)
- Cannot handle duplicate values well (don't know where to recurse)

Invariant: if we recurse in the right half, then there exists a peak in the right half

Correctness:

- There exists a peak in the range [begin, end]
- Every peak in [begin, end] is a peak in [1, n]

Quick Sort

Duplicates: running time $O(n^2)$ so use three way partitioning

Pivot choice: pivot is good if it divides the array into two parts (>= 1/10 size)

- First, last, middle: $O(n^2)$ if array designed poorly
- Median/Random(1:9 split on average): $T(n) = T(n-k) + T(k) + O(n)$

Paranoid quicksort: expected repetitions < 2 (10/8)

$$E[x] = (p)(1) + (1-p)(1 + E[x]) = 1$$
$$pE[x] = 1 \Leftrightarrow E[x] = \frac{1}{p}$$

Optimizations: recurse down to single element arrays and use insertion sort instead ($O(n)$ because elements move at most 1 position)

Stable partitioning: keep auxiliary array of indices (O(n)) which swaps will be performed on too used to disambiguate elements with equal keys and provide "total ordering" between every key

With k < n distinct keys: O(n log k)

Sorting Algorithms Overview

Descriptions:

Algorithm	Approach	Invariant
Bubble	For every element, if greater than right neighbor, swap positions, go back to start and repeat till no more swaps	At the end of iteration **j**, the biggest **j** items are correctly sorted in the final **j** position of the array
Selection	For every index, search the remaining array for the smallest element and swap to the current position	At the end of iteration **j**, the smallest **j** items are correctly sorted in the first **j** positions of the array
Insertion	For every element, keep swapping position with the previous elements before it until in the smallest possible position (for now)	At the end of iteration **j**, the first **j** items in the array are in sorted order; the remaining elements (after **j**) will remain untouched
Merge (top down)	Decompose the array into smaller unsorted arrays, once array is of size 1 or 2, sort and merge upwards	After k iterations, the first k elements in the final array are sorted, each pair of elements are correctly sorted
Merge (bottom up)	Start by sorting in groups of 2, multiply the factor by 2 and sort again, repeat till factor is greater than length of array	See above
Quick	While array length > 1, repeatedly partition array about pivot and recursively perform quick sort on both halves until everything sorted	**Partition invariants (new array):** • For every i < low, B[i] < pivot • For every j > high, B[j] > pivot • B[i] = pivot • Pivots are always sorted **Partition loop invariants (in-place):** • A[high] > pivot

Runtime:

Algorithm	Best	Average	Worst	S	Pros	Cons
Bubble	O(n) (sorted)	O(n^2)	O(n^2) (smallest at end; largest at front)	Y	Fast for sorted	Slow average case
Selection	O(n^2)	O(n^2)	O(n^2)	N		Slow for almost sorted
Insertion	O(n) (almost sorted/sorted)	O(n^2)	O(n^2) (reverse sorted)	Y	Fast for sorted	Slow average case
Merge	O(nlogn)	O(nlogn)	O(nlogn)	Y	Fast on average	Slow for small n and sorted; extra space required
Quick	O(nlogn) (random pivot)	O(nlogn) (random pivot)	O(n^2) (pivot is smallest/largest)	N	Optimisable	Slow worst case

Quick Select: $\boldsymbol{T(n) = T\left(\frac{n}{c}\right) + O(n) = O(n)}$

Finds the kth smallest element in an unsorted array (order statistics)

Choose a pivot, partition around pivot, if pivot position is greater/less than intended element, move left/right accordingly

Invariants: pivot element in exact position in sorted array; same as quicksort partition

Dictionary Implementations (key-value stores)

Behaviours: insert, search, successor, predecessor, delete, contains, size

	Sorted array	Unsorted array	Linked list
Insert	O(n) (find position)	O(1) (append)	O(1) (append)
Search	O(log n) (binary search)	O(n) (linear search)	O(n) (linear search)

Binary Search Trees (Cannot assume balanced)

Key property: all in left subtree < key < all in right subtree

$$Height = \begin{cases} 0 \; if \; leaf \\ -1 \; if \; null \\ \max(left.height, right.height) + 1 \; if \; neither \end{cases}$$

Operations: search/insert/successor/delete => O(h); traversal => O(n)

Successor: find node first

- Case 1 (right subtree exists): find the leftmost node on the right subtree
- Case 2 (no right subtree): traverse upwards (always left) until move to the right, then node at the right is the successor
- If target not found, just use successor of node

Deletion O(h):

1. Case 1 (leaf): delete
2. Case 2 (1 child): delete node and attach child to deleted node's parent
3. Case 3 (2 childs): find successor of node to delete, swap the successor with the node to delete, delete the node (now leaf)

Balanced Binary Search Trees

Balanced: $h = O(\log(n))$

Importance: ensures that $n \geq h \geq \log(n) - 1$ so that all operations are efficient

Time complexity: O(h) replaced with O(log n)

Obtaining a balanced tree:

1. Define a good property (invariant) of a tree
2. Show that if the good property holds, then it is balanced
3. After every insert/delete, make sure the good property holds, else fix it

AVL

Augment: add height to every node and adjust every insert/delete (or store balance factor with 2 bits = |left| - |right| = {-1, 0, 1})

Balance invariant: node if height-balanced if $|v.left.height - v.right.height| \leq 1$

- BST is height-balanced if every node is height-balanced

Property: height-balanced tree with n nodes has at most height $h < 2 \log n$ (tighter bound $\phi \log n$) and $n > 2^{\frac{h}{2}}$

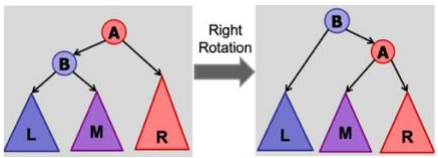
Maintain invariant: use rotations on the lowest unbalanced node

Maximally imbalanced: contains the minimum possible number of nodes given its height (start with height h and try constructing the trees) $S(h) = S(h-1) + S(h-2) + 1$

Tree rotations

Maintain BST property: A < B < C < D < E

Right rotation:



Heaviness: left/right heavy if left/right subtree has larger height than right/left subtree

Rotation order: v is -heavy

	Balanced	Left-heavy	Right-heavy
Left	Right v	Right v	Left v.left, right v
Right	Left v	Left v	Right v.right, left v

Number of rotations on insert: 2; lowest unbalanced

Number of rotations on delete: log n; all unbalanced up the walk

(a, b)-Trees

- $2 \leq a \leq \frac{b+1}{2}$
- Grows upwards

(a, b)-Trees vs binary trees:

Binary tree	(a, b)-tree
Each node has at most 2 children	Each node can have more than 2 children
Each node stores exactly one key	Each node can store multiple keys
Operations in O(log n)	Operations in O(log n)

- Use (a, b)-tree for large number of nodes as the height will decrease

Rules: must hold after every insertion/deletion

1. (a, b)-child policy: # of children always # of keys + 1

Node type	# of keys		# of 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

2. Key ranges: non-leaf node must have one more child than its number of keys
   - Ensures that all value ranges due to its keys are covered in its subtree
   - Keys are sorted
   - Key ranges: $(-\infty, v_1]$, $(v_1, v_2]$, $(v_2, v_3]$, …, $(v_{k-1}, v_k]$, (v_k, ∞)

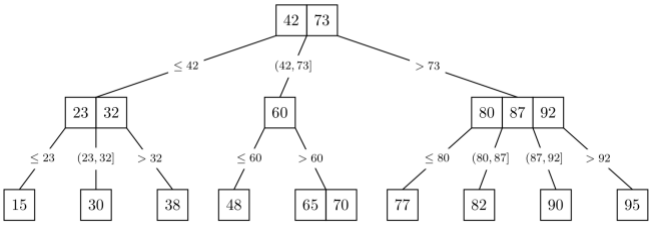
3. Leaf depth: all leaf nodes must be at the same depth from the root
   - Ensures balance (since height will never differ)

Height: minimum $O(\log_b n)$, maximum $O(\log_a n) + 1$ for sufficiently large n >> b
Search(x): $O(\log_2 b \times \log_a n) = O(\log n)$
Split: O(b) if the parent violates rule 1, split the parent too, till root; find median key and move to the parent, readjust all other nodes
Insert (Proactive): O(b × $\log_a$n) (traversal is O($\log_a$n) and splitting requires finding the median and performing the split which is O(b))
Insertion strategies:
1. Proactive (split before inserting): only applies to (a, b)-trees with b ≥ 2a
2. Passive (split after inserting while traversing up)

Merge(y, z): O(b) when y and z have < b – 1 keys together
Share(y, z): O(b) when y and z have ≥ b – 1 keys together
Delete (Passive): O(b × $\log_a$n) careful of orphaned nodes

B-trees

- (B, 2B)-tree (can use linked list to connect levels)



Cost of operations: can be optimised to O(1) if we fix the number of blocks present
1. Searching keylist: O(1)
2. Splitting/Merging/Sharing: O(1)

Storing text
AVL used: cost of tree operations is O(hL) assuming strings are of length L; assumed that strings take up space too
Tries: storing 1 letter per node/edge; root to leaf is string (end node have flag); search with DFS O(L); space O((size of text) * overhead); more node & overhead

Complexity	Tries	AVLs
Time	O(L) (search/insert)	O(hL)
Space	O(text size)	O(text size)

Augmenting Data Structures
1. Choose an underlying data structure (array, linked list, tree)
2. Determine additional info needed (weight, rank, height)
3. Modify data structure to maintain additional info when the structure changes
4. Develop new operations (select, rank)
   - Ensure that operation does not cost more than intended (e.g. inserting in a dynamic tree)

Augmented DS	Augmentation	Complexities
Dynamic order statistics (Select kth element in sorted array)	Store and update weight of each node in AVL tree; rank unsuitable (propagate update per insert) Rank = weight(node.left) + 1	**Select(k) O(log n):** If rank of node <= k, move left, else, move right and decrease k by rank of node, repeat till k found **Rank(k) O(log n):** Find v start with left.weight + 1, traverse up from the node, only if node is right child, add the rank of parent.left.weight + 1 to rank
Interval trees (Find an interval that overlaps x)	Each node is an interval of AVL tree; sort by min value; include max endpoint per node (max(n.high, left.max, right.max)) **Invariants:** • If search goes right, then no overlap in left subtree • If search goes left, then safe to go left • If search goes left and fails, then key < every interval in right subtree	Insert O(log n): key is left endpoint; update max on the way up Search O(log n): if key < left.max, move left, else move right FindOverlaps O(k log n): search for interval, add to list, delete until no more, add back all
Orthogonal range searching (Searching points in a n-d plane)	Store all points in leaves; internal nodes store max of any leaf in left subtree; rotation to change max values; each node stores a subtree of the other dimension within the interval **Further augmentation:** store count per node to avoid walking the entire subtree to find the number of points in a range • Traversal logic can replace "all-leaf-traversal" with "total += v.right.count"	**Space: O(n log^{d-1}n)** (O(log n) nodes per node of the parent for each nested dimension) **Constructing O(n log^{d-1}n)**: given a sorted list of points, sort, then choose the median point and split into left and right subtree; continue till all points of input appears as leaves; repeat per nested dimension **Query O(log^{d}n + k)** (k elements): 1. Find split node – O(log n) 2. Recurse on either sides – O(log n) 3. Search O(log n) y-trees – O(log n) each – O(log^{2}n) Enumerate output – O(k) **No insert/delete**
Merkle tree (comparing large amount of text)	Binary tree with leaves being the hashes of blocks of the text to be stored Adjacent neighbors are added together to form new hash, repeated till root	

kd-Trees
Alternate levels (horizontal, vertical) per level of tree; dividing the points in the plane in half; more efficient updates, better in practice, expanded set of queries; O(n log n) construction, O(h) queries, O(√n) to find min/max)

Priority Queue
Maintain set of prioritized objects
Operations: insert (with priority), extractMin, decreaseKey (lower priority), contains, isEmpty
Possible implementations: other operations supported with second dictionary indexed by key

	Sorted array	Unsorted array	AVL (index = priority)
Insert	O(n)	O(1)	O(log n)
Extract max	O(1)	O(n)	O(log n)

Heap

- Max priority queue implementation
- Largest item at root, smallest items at leaves

Properties: note that heaps are not BSTs
1. Heap ordering: priority[parent] >= priority[child]
2. Complete binary tree: every level is full except possibly last
   - Nodes are populated from the left first

Maximum height: $\lfloor \log n \rfloor$ or O(log n)
Insert O(log n):
1. Add new leaf with priority to the left most available spot
2. Bubble up till root (swapping with parent if new node priority larger)

increaseKey O(log n): change priority and bubble up
decreaseKey O(log n): change priority and bubble down till leaf (focus on moving to where the larger priority is)
delete O(log n): swap node with last priority (right most leaf on the last level), delete last, and bubble down the swapped node
extractMax O(log n): delete root

Heap vs AVL
1. Same asymptotic cost for operations
2. Faster real cost (no constant factors)
3. Simpler (no rotations)
4. Slightly better concurrency

Complete binary tree to array (Binary Heap)

- Same as queue from BFS (level-by-level)
- Array size same as $2^n - 1$ where n is the max level

Insertion O(log n): append to array end; bubble up by swapping elements in array
Left(x) O(1): $2x + 1$ where x is the index of the node
Right(x) O(1): $2x + 2$
Parent(x) O(1): $\left\lfloor \frac{x-1}{2} \right\rfloor$
Right rotation: not an O(1) operation since many nodes would be affected

Heapsort O(n log n)
1. Start with converting the array into complete tree (use operations above)
2. Recurse up from the leaves and bubble down parent to the leaves (start from the end)
   a. Treat each subtree as its own heap
3. Repeat till root
4. Result will be max heap

Cost of optimised heapify: $\sum_{h=0}^{\log n} \frac{n}{2^h} O(h) \leq O(n)$
Analysis: in-place, merge < heap < quick, deterministic, unstable

Union Find
Store the connectivity of objects using connected components; relies on transitivity property: if p <~> q and q <~> r, then p <~> r
Operations: find(p, q), union(p, q)
Quick find: union with arrays; forms flat trees
Quick union: union with trees; forms unbalanced trees; both operations reach root of each node and perform operation on roots
Weighted union: make the root with larger number of nodes the parent; max depth O(log n); weight/rank/size/height works; balanced tree

- Weight/rank/size/height of subtree does not change except at the root
- Weight/rank/size/height only increases when tree size doubles

Path compression: after finding the root, set the parent of each traversed node to the root; or make every other node point to grandparent; only occurs during finds
Weighted union with path compression: starting from empty, any sequence of m union/find operations on n objects takes O(n + ma(m, n)) time (almost linear time)

- a(m, n) is the Inverse Ackermann function (always <= 5)

Type	Find(p, q)	Union(p, q)
Quick find	O(1)	O(n)
Quick union	O(n)	O(n)
Weighted union	O(log n)	O(log n)
Path compression	O(log n)	O(log n)
WU + PC	a(m, n)	a(m, n)

Symbol table
Key-value mapping that implements inserts and searches in O(1) time
Operations: insert(key, value), delete(key), search(key), contains(key), size()

Hashing
h(k) : U -> {1..m} maps key k to an integer

- Time taken: time to compute h (assumed to be O(1)) + time to access bucket (O(1) if array)
- Maps key k to bucket h(k) in a direct access table
- If $k_1 = k_2$, then $h(k_1) = h(k_2)$

Collisions: two distinct keys produce the same hash function (unavoidable)
Hash function design: goal to make it look random
1. Division method: h(k) = k mod m
   - Don't use m = 2^x; use m = prime not too close to power of 2 or 10
     k and m common divisor d => k mod m divisible by d => 1/d table used
2. Multiplication method: h(k) = (Ak) mod 2^w >> (w – r)
   - Fix table size (m = 2^r), fix word size (w bits), and odd constant A
   - Faster than division method

Chaining
Each bucket contains a linked list of items for all items with same hash

- Total space: O(m + n) where m is the number of buckets and n is the number of entries
- Must store both the key and the value to identify when searching
- Operations: insert (O(1 + cost(h))), search (O(n + cost(h))

Simple Uniform Hashing Assumption (SUHA): every key is equally likely to map to every bucket and keys are mapped independently; reduced collisions

- load(hash table) = n/m (average number of items per bucket)
- Expected search time = O(1) + load(hash table)
- Probability that a key ends up in slot j: Pr(h(k) == j) = 1/m
- Expected value of key in slot j: 1/m
- Unsuccessful search: n/m
- Successful search: 1 + (n – 1)/m <= 1 + n/m
- Max cost to insert n elements: O(log n) or $\Theta\left(\frac{\log n}{\log \log n}\right)$

Table size: $m = \Theta(n)$

- m < 2n -> too many collisions; m > 10n -> too much wasted space
- n is not known in advance

Open addressing
Each bucket contains only one element and any collisions will probe a sequence of buckets until an empty one is found

- Insertions will fail if table is full

Deletion: replace deleted value with TOMBSTONE to signal to searching sequences to skip over value and continue searching; may fail to search without TOMBSTONE; overwritten on insert
Linear probing: h(k, 1) -> h(k, 2) = h(k, 1) + 1 -> h(k, 3) = h(k, 1) + 2… -> h(k, i) = h(k, 1) + i mod m (the probing loops the entire table)

- Problem: clustering (sequential buckets that are full); cluster => higher probability next hash hits a cluster and grows the cluster
- If table is ¼ full, then there will be clusters of size θ(log n) => average cluster size grows logarithmically
- Fast in practice as nearby memory access is fast (nearly O(0))

Properties:
1. H(key, i) enumerates all possible buckets
   - For every bucket j, there exists some i such that h(key, i) = j
   - Hash function is a permutation of {1..m}, otherwise table can be treated as full even if there's space left
2. Uniform Hashing Assumption: every key is equally likely to be mapped to every permutation, independent of every other key

- Probability of a key ends up in slot j on the i probe: Pr(h(k, i) == j) 1/m

Double hashing: h(k, i) = f(k) + i*g(k) mod m

- If g(k) is relatively prime to m, then h(k, i) hits all buckets
- If m = 2^r, then g(k) is odd

Expected cost per operation: depends on load $\alpha = \frac{n}{m}$ (only if α is smaller)

$$E[\#\ of\ probes] = 1 + \frac{n}{m}(E[\#\ of\ probes - 1]) = \frac{1}{1-\alpha}$$

Advantages: less space used; rarely allocating memory; better cache performance

Disadvantages: more sensitive to choice of hash function; clustering; more sensitive to load as performance degrades exponentially as $\alpha \to 1$

Table resizing

When n == m, m = 2m (O(n) resize cost); when n < m/4, m = m/2

1. Choose new table size m
2. Choose new hash function h
3. For every item in old hash table, compute a new hash and copy the item to the new hash table

- Time complexity: O(o + m + n) where o is the old table size

Amortized analysis

$$Avg = \frac{Total\ Cost\ of\ insert + resize}{Total\ Elements} = \frac{1 + 1 + 1 + \cdots + (resize\ cost)}{m}$$

- Total cost of insert and resize: resize time based on resize cost (above); insert time is O(1)
- Total elements is always |m – resize criteria limit|; 0 for insert, limit for delete

Set

Operations: insert(key), contains(key), delete(key), intersect(set), union(set)

Implementations:

1. Hash table: not space efficient
2. Fingerprint hash table: store m bits where 1 indicates that the hash is set; no false negatives; false positives if collisions; uses less space per bucket but requires more buckets to avoid collisions (p = false positive probability)

$$\mathrm{p(false\ positive)} = 1 - \left(1 - \frac{1}{m}\right)^n \approx 1 - \left(\frac{1}{e}\right)^{\frac{n}{m}}$$

$$\frac{n}{m} \le \log\left(\frac{1}{1-p}\right)$$

3. Bloom filter: fingerprint hash table + k hashing function; all buckets must contain 1 to consider the element present; only false positives when k collisions; slightly more space required

$$\mathrm{p(false\ positive)} = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

$$\mathrm{Optimal}\ k = \frac{m}{n}\ln 2$$

$$\mathrm{Error\ probability}: 2^{-k}$$

- Used when storing a set of data in a space-sensitive condition where false positives are ok

Graphs

Consists of set of nodes V where |V| > 0 and set of edges E where $E \subseteq \{(v, w) : (v \in V), (w \in V))\}$

Graph	Unique properties
Undirected	Each edge is unique (no repeats)
Directed	Each edge is unique; (v, w) != (w, v) as v -> w different from w -> v
Multigraph	Each edge may be repeated
Hypergraph	Each edge contains >= 2 nodes; each edge is unique
Star	Central node with all edges connected to the centre; degree of n – 1; diameter of 2
Clique (complete graph)	All pairs of connected by edges; degree of 2; diameter of n – 1; total edges is n(n-1)/2
Cycle	Diameter of n/2 or n/2 – 1; degree of 2
Bipartite graph	Nodes divided into two sets with no edges between nodes of the same edge; determined using two colour graph colouring (if two adjacent nodes have the same colour, then not bipartite); max diameter of n - 1
Planar	Graph without any edges that cross; area bound between 4 nodes is a face
Directed Acyclic Graph	Directed graph without any cycles

4-coloring theorem: any planar graph can be coloured using only four colours

Configuration graphs: represent every possible state as vertices and every edge is a possible move to be made

Graph representations:

	Adjacency list	Adjacency matrix
Description	Nodes stored in an array Edges stored as linked lists per node (List<List<Integer>>)	A[v][w] = 1 iff (v, w) in edges (boolean[][])
Memory	O(V + E) O(V) for cycles	O(V^2) regardless of graph type
Uses	Sparse graph representation Saves more space	Find if two vertices are n-hop neighbors (A^n) Dense graph representation Slightly faster
Fast queries	Find any neighbor Enumerate all neighbors	Are v and w neighbors?
Slow queries	Are v and w neighbors?	Find any neighbor Enumerate all neighbors

Graph searching

Given some start vertex, find a path to end vertex; or visit all nodes; expensive to compute if exponential paths

Frontier: list of vertices to visit

BFS	DFS
Explore nodes level-by-level	Explores path until no more nodes to visit
No backtracking	Backtracks to find unexplored nodes
Finds shortest path (not same as minimum distance); may have high degree and/or diameter	Does not find shortest path
Implemented with *queue*	Implemented with *stack*
Not all paths explored	
Nodes not revisited	
Path found is a tree	
Adjacency list: O(V + E) Adjacency matrix: O(V^2)	

1. Add start node to stack/queue
2. Repeat until stack/queue is empty
   a. Pop node v from stack/queue
   b. Visit v
   c. Explore all outgoing edges of v
   d. Push all unvisited neighbors of v to the stack/queue

Pre-order DFS: process each node when it is first visited (before recursive call)

Post-order DFS: process each node when it is last visited (after recursive call)

Shortest paths

Aδ(u, v) = distance from u to v

Triangle inequality: $\delta(S, C) \le \delta(S, A) + \delta(A, C)$

Estimate: every node has estimate of the distance it takes to reach it (initially inf except source 0)

- Goal to reduce estimate with invariant that estimate ≥ distance

Relaxing: update the estimate from u to v only if the new weight results in a shorter path

```
relax(Edge e) {
  int v = e.from();
  int w = e.to();
  double weight = e.weight();
  if (distTo[w] > distTo[v] + weight) {
    distTo[w] = distTo[v] + weight;
    parent[w] = v;
    if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
    else pq.insert(w, distTo[w]);
  }
```

Path relaxation property: if $p = v_0, v_1, \ldots, v_k$ is a shortest path from first and last vertex, and we relax the edges of p in order of zipped pairs, then $d[v_k] = \delta[v_k]$

- Holds regardless of any other relaxation steps that occur in between

Condition	Algorithm	Time Complexity
No negative weight cycles	Bellman-Ford Algorithm	O(VE)
On unweighted graphs (or equal weights)	BFS	O(V+E)
No negative weights	Dijkstra's Algorithm	O(E log V)
On tree	BFS/DFS (only 1 path)	O(V)
On DAG	Topological sort order & relax in order	O(V + E)
Longest path	Bellman-Ford with negated weights	O(VE)

Bellman-Ford

Relax every edge for every vertex in graph (or V – 1 works too); terminates early when entire sequence of relaxations have no effect

Time complexity: O(EV)

Invariant: after iteration j, if node u is j hops from s on the shortest path, then est[u] = dist(s, u); i.e. after iteration j, the nodes that are exactly j hops away will have the correct estimate

Negative cycles: negative weight cycles only further decrease the estimates; run BF one more iteration to check; set all nodes to constant needs |V| times

INFTY: choice of infinity must not go out of bounds since we will be relaxing all edges at once

Dijkstra

Relax edges in the "right" order where every edge is only relaxed once; edges must be non-negative; path monotonically increasing

Invariant: once a node is visited, it will hold the shortest possible path from source to itself

- Negative weights break this invariant as it may alter the shortest path after visiting a node
- Reweighting does not work as it alters the positive paths
- Stop once destination node is dequeued

```
public Dijkstra {
  private Graph G;
  private IPriorityQueue<Integer> pq = new PriorityQueue<>();
  private double[] dist;
  search(int start) {
    pq.insert(start, 0.0);
    distTo = new double[G.size()];
    Arrays.fill(distTo, INFTY);
    distTo[start] = 0;
    while (!pq.isEmpty()) { -- total O(V) times
      int w = pq.deleteMin(); -- O(log V) dependent on priority queue
      for (Edge e : G[w].nbrList) relax(e); -- total O(E) times; O(log V) at least once
    }
  }
}
```

PQ Impl	Insert	Delete min	Decrease	Total
Array	1	V	1	O(V^2)
AVL tree	log V	log V	log V	O(E log V)
d-way heap	$d \log_d V$	$d \log_d V$	$\log_d V$	O(E $\log_{E/V}$V)
Fibonacci Heap	1	log V	1	O(E + V log V)

Topological ordering

Properties: sequential total ordering of all nodes where dependencies are satisfied first; edges only point forward; only for DAGs; not unique

DFS: explore a node to its end before trying a different path; O(V + E)

- Pre-order: append to order
- Post-order: prepend to order (follow DFS order, then prepend when done with branches)

Khan's algorithm: repeat till no more nodes to add; O(V + E); hashmap to store incoming edges

1. S = all nodes in G that have no incoming edges
2. Add nodes in S to the topo-order
3. Remove all edges adjacent to nodes in S
4. Remove nodes in S from the graph

Spanning tree

Acyclic subset of edges that connect all nodes

Minimum Spanning Tree

Spanning tree with minimum weight (unique)

- One edge in a cycle can be removed to reduce weight
- Cannot be used to find shortest path
- Contains exactly V – 1 edges

Cut: partition of vertices V into two disjoint subsets

Crossing a cut: edge that have one vertex in each of the two sets of a cut

Properties:

1. No cycles
2. If an MST is cut, then two pieces are both MSTs
3. Cycle property: for every cycle, the maximum weight edge is not in the MST (no guarantee for minimum weight edge)
4. Cut property: for every partition of the nodes, the minimum weight edge across the cut is in the MST
   - For every vertex, the minimum outgoing edge is always part of the MST

Adding constant to all weights: no change in MST

Generic and greedy MST algorithm: repeat red or blue rule on every edge until none can be colored; blue edges are an MST

Red rule: C is a cycle with no red arcs => color the max-weight edge in C red

Blue rule: D is a cut with no blue arcs => color the min-weight edge in D blue

Variation	Approach
Constant weight edges	DFS/BFS O(E) since any DFS/BFS is a tree; total weight is n(V – 1) where n is the constant weight
Bounded integer edges (e.g. weight from {1..10})	**General modification:** store linked list of edges with given weight j in array of size b (b is the upper bound) **Kruskal's modification:** iterate over all edges in ascending order and for each edge, check if edge needs to be added (from UF) and union Time complexity: O(αE) where α is the time for UF

	Prim's modification: implement priority queue using size b with following operations: • Insert: put node in correct list O(V) • Remove: lookup node and remove from linked list • ExtractMin: remove from the minimum bucket • DecreaseKey: lookup node and move to correct linked list O(V) Time complexity: O(V + E) = O(E)
DAG with one "root"	For every node except root, add minimum weight incoming edge. Time complexity: O(E)
Maximum spanning tree	Multiply each edge weight by -1 and run the MST algorithm, MST with the most negative is the maximum Run Kruskal's algorithm in reverse (order by max edge weight)

Prim's Algorithm

Start with any node, add smallest adjacent edge and neighbor to MST, use subset of nodes to find next smallest adjacent edge to any node in MST and repeat till no more nodes to add; preferred if dense graph

Time complexity: O((V+E)logV)=O(E log V)

```
Create priority queue of all vertices and give them all priority of INF  -- O(V log V)
Set priority of starting vertex to 0

Create hash set of seen vertices
Add the starting vertex to hash set

Create hashmap of parent-child relationships where the key is the child, value is the parent
Add the starting vertex to the hash map with null value

While the priority queue is not empty: -- total V times at most; O((V+E)log V)
  Extract the minimum vertex to process  -- O(log V) (priority queue dependent)
  Add the current vertex to the hash set
  For every neighbor of the vertex: -- total E times at most; O(E log V)
    If neighbor has not been seen before:
      Lower the priority of the neighbor – O(log V) (priority queue dependent)
      If the priority of the neighbor was lowered, change the parent link of the neighbor to the
current vertex
```

Kruskal's Algorithm

Sort edges by weight from smallest to biggest and add edges that do not form cycles, otherwise discard (red rule); preferred if sparse graph

- Use union find to connect nodes if they are in the MST

Time complexity: O(E log V)

```
// Sort edges and initialize
Edge[] sortedEdges = sort(G.E()); -- O(E log E) = O(E log V²) = O(E log V)
UnionFind uf = new UnionFind(G.V());
ArrayList<Edge> mstEdges = new ArrayList<Edge>();

// Iterate through all the edges, in order
for (int i=0; i<sortedEdges.length; i++) { -- O(E)
  Edge e = sortedEdges[i]; // get edge
  Node v = e.one(); // get node endpoints
  Node w = e.two();
  if (!uf.find(v,w)) { // in the same tree? – O(log V)/O(a(m, n))
    mstEdges.add(e); // save edge
    uf.union(v,w); // combine trees – O(log V)/O(a(m, n))
  }
}
```

Boruvka's Algorithm

1. Initialize a forest with one-vertex trees (V connected components)
2. While connected components != 1:
   a. For each connected component:
      i. Begin with an empty set of edges
      ii. For each vertex in the connected component:
         1. Find the cheapest edge from vertex to outside of connected component
         2. Add cheapest edge to set of edges
      iii. Add cheapest edge of set of edges to current set of connected components
   b. Combine trees to form larger connected component

- Each step adds at least k/2 edges (k components), merges k/2 components, leaving at most k/2 components left (O(V + E))
  - Runs O(log V) times
  - Time complexity: O((E + V) log V) = O(E log V)

Dynamic programming

Key properties: optimal substructure and overlapping subproblem

Optimal substructure: optimal solution can be constructed from optimal solutions to smaller sub-problems

Overlapping sub-problems: same smaller problem is used to solve multiple different bigger problems

Divide and conquer: optimal substructure but no overlapping subproblems

Methods of solving:

1. Bottom-up: solve smallest problems, combine smaller problems, solve root problem
2. Top-down: start at root, recurse till base case, solve base case and memoize each solution
3. DAG: represent problem as DAG, topologically sort the DAG, and solve the problems in reverse order

Process:

1. Identify optimal substructure
2. Define subproblems
3. Solve problem using subproblems
4. Write code

Total cost: number of subproblems x cost to solve each subproblem

Longest Increase Subsequence

DAG $O(n^3)$: outgoing edge from node to every other node that is larger than it

- Longest paths per node (SSSP on DAG)

Subproblem $O(n^2)$: S[i] = LIS(A[i..n]) starting at A[i] (start from the back)

$$S[i] = \begin{cases} 0 \text{ if } i = n \\ \left(\max_{i,j \in E} S[j]\right) + 1 \text{ if } i \neq n \end{cases}$$

- Start from beginning: S[i] = LIS(A[1..i])

```
LIS(V):  // using DAG
  int[] S = new int[V.length]
  for i = 0 -> V.length - 1: S[i] = 0
  S[n-1] = 1
  For v = V.length – 2 -> 0:
    Max = 0
    For neighbor in v.neighbors:
      If S[w] > max: max = S[w]
    S[v] = max + 1
```

```
LIS(A):  // not using DAG
  S = [0] * len(A)
  S[0] = 1
  For i = 0 -> A.length – 1:  // start from the beginning and check backwards
    Max = 0
    For j = 0 -> i – 1:
      If A[j] < A[i] and S[j] > max: max = S[j]
    S[i] = max + 1
```

Lazy prize collecting

Find the highest prize given k edges to cross; includes negative and positive weight cycles

DAG O(kVE) using DAG and O(kE) using table: transform into DAG, make k copies of every node where edges between nodes denote a hop; solve using DAG SSSP for longest path

- Add super node to compute definitive shortest path in one go

Subproblems $O(kV^2) = O(kE)$ (fully connected graph): P[v, k] = maximum prize that can be collected starting at v and taking exactly k steps

$$P[v,k] = max\{P[w_1, k-1] + w(v,w_1), P[w_2, k-1] + w(v,w_2), P[w_3, k-1] + w(v,w_3), \dots\}$$

```
int LazyPrizeCollecting(V, E, kMax) {
  int[][] P = new int[V.length][kMax+1]; // create memo table P
  for (int i=0; i<V.length; i++) // initialize P to zero
    for (int j=0; j<kMax+1; j++) P[i][j] = 0;

  for (int k=1; k<kMax+1; k++) { // Solve for every value of k
    for (int v = 0; v<V.length; v++) { // For every node…
      int max = -INFTY; // …find max prize in next step
      for (int w : V[v].nbrList()) {
        if (P[w,k-1] + E[v,w] > max)
          max = P[w,k-1] + E[v,w];
      }
      P[v, k] = max;
  }
  return maxEntry(P); // returns largest entry in P
}
```

Minimum vertex cover in undirected tree

Set of nodes C where every edge is adjacent to at least one node in C

Subproblem 2V O(V): S[v, 0] = size of vertex cover in subtree rooted at node v if v is not covered; S[v, 1] = size of vertex cover in subtree rooted at node v if v is covered

$$S[v,0] = S[w_1,1] + S[w_2,1] + S[w_3,1] + \cdots$$
$$S[v,1] = 1 + \min(S[w_1,0], S[w_1,1]) + \min(S[w_2,0], S[w_2,1]) + \cdots$$

- Start at leaves where S[leaf, 0] = 0 and S[leaf, 1] = 1

```
int treeVertexCover(V){//Assume tree is ordered from root-to-leaf
  int[][] S = new int[V.length][2]; // create memo table S
  for (int v=V.length-1; v>=0; v--){//From the leaf to the root
    if (v.childList().size()==0) { // If v is a leaf…
      S[v][0] = 0;
      S[v][1] = 1;
    } else { // Calculate S from v's children.
      int S[v][0] = 0;
      int S[v][1] = 1;
      for (int w : V[v].childList()) {
        S[v][0] += S[w][1];
        S[v][1] += Math.min(S[w][0], S[w][1]);
      }
    }
  }
  return Math.min(S[0][0], S[0][1]); // returns min at root
}
```

All Pairs Shortest Path

Naïve: run SSSP once for every vertex in the graph

- Assuming non-negative weights: O(VE log V) (run Dijkstra V times)
  - Sparse graph (E = O(V)): O(V^2 log V)
- Assuming same weights: O(V(E + V)) = O(VE) (run BFS V times)
  - Sparse graph: O(V^2), dense graph: O(V^3)

Floyd-Warshall $O(V^3)$: optimal substructure: if P is the shortest path (u -> v -> w), then P contains the shortest path from u -> v and from v -> w

- **Subproblem:** S[v, w, P] = shortest path from v to w that only use intermediate nodes in the set P
  - P spans from ∅ to {1..n} adding 1 more vertex in each new set (n + 1 sets)
- **Base case:** S[v, w, ∅] = E[v, w]

$$S[v,w,P_8] = \min\left(S[v,w,P_7], S[v,w,P_7] + S[8,w,P_7]\right)$$

  - Shortest path either includes new node or does not
- Table representation: row (from) to column (to) with cells being total value

```
int[][] APSP(E){ // Adjacency matrix E
  int[][] S = new int[V.length][V.length];//create memo table S

  // Initialize every pair of nodes
  for (int v=0; v<V.length; v++)
    for (int w=0; w<V.length; w++)
      S[v][w] = E[v][w];

  // For sets P0, P1, P2, P3, …, for every pair (v,w)
  for (int k=0; k<V.length; k++)
    for (int v=0; v<V.length; v++)
      for (int w=0; w<V.length; w++)
        S[v][w] = min(S[v][w], S[v][k]+S[k][w]);

  return S;
}
```

FW Variations:

1. Path reconstruction O(V^2): store the first hop for each destination and reconstruct from v to w using all intermediate nodes
2. Transitive closure: return matrix where

$$M[v,w] = \begin{cases} 1 \text{ if } \exists \text{ path(v,w)} \\ 0 \text{ if } !\exists \text{path(v,w)} \end{cases}$$

3. Minimum bottleneck edge: (v, w) bottleneck is the heaviest edge on path between v and w; B[v,w] = weight of minimum bottleneck

Common Graph Techniques

1. Create supernode to find shortest path starting at multiple nodes to same destination (edge weight 0)
2. Reverse edge directions (shortest path does not change but might be able to simplify computation)
3. Connect source to mandatory vertices (with weight 0) to force traversals to that node
4. Duplicate graphs and add edges between duplicates to imply traversal
5. Shift goal post to duplicate graphs to handle cycles/duplicates
6. Solve from the end, rather than the start
7. Break up each node/edge to represent all possible state changes/transitions (e.g. direction)
8. Try applying graph coloring to figure out any relationship between data to exploit

Common CS2040S Tips

1. Work with smaller examples (array size 2/3, graph with few nodes)
2. Test for corner cases (duplicate elements, negative numbers)
3. Augment existing data structures
4. Figure out naïve solution and work from there
5. Reverse the problem, add to the problem, redefine the problem, apply the same algorithm multiple times