1 Time Complexity

Master's theorem for $T(n) = aT(\frac{n}{b}) + f(n)$ where $a \geq 1$ and $b > 1$
Let $c_{crit} = log_b(a)$ and if $f(n) = \theta(n^c)$

1. If $c < c_{crit}$ then $T(n) = \theta(n^{c_{crit}})$
2. If $c = c_{crit}$ then $T(n) = \theta(n^c log(n))$
3. If $c > c_{crit}$ then $T(n) = \theta(f(n))$
4. If $f(n) = \theta(n^{c_{crit}} log^k(n))$, then $T(n) = \theta(n^{c_{crit}} log^{k+1}(n))$

2 Sorting

2.1 Bubblesort

Time complexity: $\Omega(n)\ \theta(n^2)\ O(n^2)$
Invariant: At the end of iteration `j`, the biggest `j` items are correctly sorted in the final `j` positions of the array.

2.2 SelectionSort

Time complexity: $\Omega(n^2)\ \theta(n^2)\ O(n^2)$
Invariant: At the end of iteration `j`: the smallest `j` items are correctly sorted in the first `j` positions of the array.

2.3 InsertionSort

Time complexity: $\Omega(n)\ \theta(n^2)\ O(n^2)$
Invariant: At the end of iteration `j`: the first `j` items of the array are sorted in order.

2.4 QuickSort

Time complexity: $\Omega(nlog(n))\ \theta(nlog(n))\ O(n^2)$ $O(nlog(n))$ **(for paranoid QuickSort)**
Invariant: For every `i < low: B[i] < pivot` and for every `j > high: B[j] > pivot`. Duplicates: Use three-way partition to store duplicates.

2.5 MergeSort

Time complexity: $\Omega(nlog(n))\ \theta(nlog(n))\ O(nlog(n))$
Invariant: At the end of each loop, the subarrays are sorted.

2.6 QuickSelect

Time complexity: $\Omega(n)\ \theta(n)\ O(n^2)$
Invariant: For every `i < low: B[i] < pivot` and for every `j > high: B[j] > pivot`.

3 Trees

3.1 Binary Search Tree (BST)

Only has 2 children per node. Left child is smaller than parent. Right child is larger than parent.
Operations: $O(log(n))$ / $O(n)$ (balanced)

3.2 AVL Tree

Maximum height between two children tree is 1.
Operations: $O(log(n))$
Insertion Max Rotations: 2
Deletion Max Rotations: $log(n)$
Invariant: `|height(u) - height(v)| < 2` if `v` and `u` are sibling nodes. `|height(u) - height(v)| > 0` if `u` is the parent node of `v`.

3.3 Order Statistics (Rank finding)

AVL Tree augmented with a `weight` property in each node.
Updating weights on rotate: $O(1)$

3.4 Interval Trees

Store the maximum value of entire subtree under a node as a parameter. Update `max` during rotations by taking `Math.max` of all children under the two nodes being swapped.
Time complexity: $O(log(n))$

3.5 1D Range Finding

All leaves hold the value, each internal node `v` stores the `MAX` of any leaf in the left sub-tree.
Operations:

`findSplit`	Finds the node to start searching.
`traverseLeft`	Traverse the left subtree after findSplit.
`traverseRight`	Traverse the right subtree after findSplit.

Complexity:

`Query`	$O(k + log(n))$
`Build`	$O(nlog(n))$
`Space`	$O(n)$

3.6 (a, b) trees

An (a, b) tree has a bound of $2 \leq a \leq \frac{b+1}{2}$.

Rule 1:

Node Type	#Keys		#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

Rule 2: A **non-leaf node** must have one more child than its number of keys.
Rule 3: All **leaf nodes** must be at the same depth (from the root).

`search`	Traverse from the root node and binary search the node array. ($O(log(n))$).
`insert`	Traverse from the root to find point of insertion. Split if needed ($O(log(n))$).
`delete`	Traverse from the root to find node and delete. Sacrifice/merge/split if needed ($O(log(n))$).
`split`	Find the median element in the node array and sacrifice to the parent. The split halves are now the child of the promoted node.
`merge`	Merges two siblings. Delete the parent and add the parent to the keylist of one child. Merge the two children.
`share`	Merge and split combined.

4 Hashing

4.1 Hashing with Chaining

If a current bucket is occupied, add to the linked list in the bucket.

- **Insert:** $O(1 + cost(f))$
- **Search/Delete:** $O(n + cost(f))$ where n is the number of nodes.
- Expected Number of items per bucket = $\frac{n}{m}$.
- Under Simple Uniform Hashing Assumption (every key is likedly to be mapped to every permutation), $E(\text{search}) = O(1)$.
- Can still add items when $m == n$ and search efficiently.

4.2 Hashing with Open Addressing

If the current bucket is full, find the next available bucket (Linear Probing).

- **Insert/Delete/Search:** $O(1)$ if $\alpha < 1$ where $\alpha = \frac{n}{m}$ where n is the #items and m is #buckets.
- When deleting a key, replace the bucket with `DELETED`. Functions will probe past `DELETED`.
- Clusters of size $\Theta(log(n))$ form when table is $\frac{1}{4}$ full. Caching of arrays make this fast.
- Unable to insert and search efficiently when $m == n$.
- Expected cost of operation is $\leq \frac{1}{1-a}$.

4.3 Double Hashing

Two ordinary hash functions $f(k)$ and $g(k)$ with new hash function:

$$h(k,i) = f(k) + i \cdot g(k)\ mod m$$

$g(k)$ must be relatively prime to m for $h(k,i)$ to hit all buckets.

4.4 Hashcode

`Integer`	The int value itself.
`Long`	Split the long into 32 bits, XOR.
`String`	Iterate through each char, sum the value with an offset.

4.5 Resizing

If $(n == m)$, then $m = 2m$ and if $(n < \frac{m}{4})$, then $m = \frac{m}{2}$.

Definition 4.1 *Operation has amortized cost $T(n)$ if for every integer k, the cost of k operations is $\leq kT(N)$.*

4.6 HashSet

- Stores a bit instead of key.
- $P(\text{false positive}) = 1 - (1 - \frac{1}{n})^n \approx 1 - (\frac{1}{e})^{\frac{n}{m}}$.

4.7 Bloom Filter

- Use 2 hash functions $f(k)$ and $g(k)$. Both buckets must return `True` for a positive match.
- $P(\text{false positive}) = (1 - \frac{1}{e}^{\frac{2n}{n}})^2$.
- $P(\text{bit is 0}) = (1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$.
- $P(\text{collision at 1 spot}) = 1 - e^{-kn/m}$.
- $P(\text{collision at 1 spot}) = (1 - e^{-kn/m})^k$

5 Graphs

Definition 5.1 *A cycle is a connected graph with a loop. It has a diameter $\frac{n}{2}$ or $\frac{n}{2} - 1$ and degree 2.*

Definition 5.2 *A bipartite graph is a graph with nodes divided into two sets. There are no edges between nodes of the same set.*

Definition 5.3 ***Cayley's Formula:*** *The number of possible MSTs of a complete graph is V^{V-2}.*

5.1 Representation

- Adjacency List. Each node has a linked list of edges to other nodes. ($O(V + E)$ memory).
- Adjacency Matrix. 2D array that stores path from a node to another. ($O(V^2)$ memory).

5.2 Traversal

- BFS. Use a queue to explore neighbours level by level.
- DFS. Use a stack to explore the maximum depth, then neighbours.

5.3 Directed Acyclic Graph

A topological sorted graph can find shortest paths in $O(E)$ by walking through the graph once.

5.3.1 Post-order DFS

- Use post-order DFS to add nodes to the set.
- Time Complexity: $O(V + E)$.

5.3.2 Kahn's Algorithm

- Maintain a set of nodes with no incoming edges. Remove edges and add them to the set when a node has no incoming edge.
- Time Complexity: $O(Elog(V))$ or $O(E+V)$.

5.4 Shortest Paths

5.4.1 Bellman Ford

- Relax all edges V times. Can detect negative weight cycles.
- Time Complexity: $O(EV)$ (Can optimise to terminate early when no weight changes detected).
- Negative weight cycles can be detected with $|V|+1$ iterations. To label all negative weight cycles, run for $2|V|$.

5.4.2 Dijkstra's

- Relax shortest edge using a priority queue.
- Time Complexity: $O(Elog(V))$ (Assumes PQ operations are $log(v)$).
- Each edge is relaxed once and each node is added to the priority queue once.

6 Heaps

Binary heap stores inserts nodes from left to right and ensures the maximum height is $O(floor(log(n)))$.

`insert`	$O(log(n))$
`delete`	Swap with min element and `bubbleDown` min element. $O(log(n))$
`decreaseKey`	$O(log(n))$
`extractMax`	$O(log(n))$

For a lookup array (swap positions when we bubble):

`leftChild`	$2x+1$
`rightChild`	$2x+2$
`parent`	$floor((x-1)/2)$

6.1 Heapify

Bubble from leaf upwards, recursively ensured that subtrees are heaps.

```
for (int i = n - 1; i >=0; i--):
    bubbleDown(i, A) // O(logn)
```

Time Complexity:

$$\sum_{h=0}^{h=log(n)} \frac{n}{2^h} O(h) \leq 2 \cdot O(n)$$

6.2 HeapSort

Extract the max and put it at the back of the HeapArray.

```
for (int i = n -1; i >= 0; i--):
    A[i] = extractMax(A);
```

- **Time Complexity:** $O(nlog(n))$.
- Unstable, inplace, faster than MS, slower than QS, deterministic (always $O(nlog(n))$).

7 UFDS

7.1 QuickFind

Time Complexity: Find: $O(1)$ Union: $O(n)$

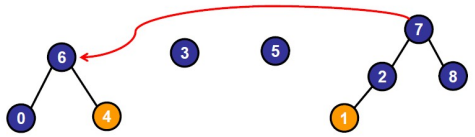
`find`	Check the int array if two items have the same id.
`union`	`union(p, q)` change all items with id of p to id of q.

7.2 QuickUnion

Time Complexity: Find: $O(n)$ Union: (n)

`find`	Check the ADT if two items have the same parent id.
`union`	`union(p, q)` change parent of q to id of `parent(p)`.

QuickUnion(1, 4):



7.2.1 Weighted Union

Time Complexity: Find: $O(log(n))$ Union: $(log(n))$
Connect the parent in order of size to ensure the height of tree is $O(log(n))$.

7.2.2 Path Compression

Time Complexity: $O(n+m\alpha(m,n))$.
When `find` is called, compress each traversed parent to the found root.

8 MST

Properties:

1. No cycles
2. If you cut an MST, both pieces are MSTs.
3. For every cycle, the max weight edge is NOT in.
4. For every cut, the min weight is in the MST.

8.1 Prim's Algorithm

Time Complexity: $O(Elog(V))$.
Queue all nodes in a PQ. Decrease key for every adjacent edge of dequeued node.

8.2 Kruskal's Algorithm

Time Complexity: $O(Elog(V))$.
Sort all edges. Use UFDS, add edges if the two nodes are not in the same set.

8.3 Boruvka's Algorithm

Time Complexity: $O(Elog(V))$.
Step 1: Add all minimum adjacent edges.
Step 2: Add minimum edge out of each connected component. Repeat till 1 connected component left.

8.4 Variants

For all edges having weights from **{1..10}**.

8.4.1 Kruskal

Time Complexity: $O(\alpha E)$.

1. Put all edges array of linked lists.
2. Iterate through all edges and check to union or not.

8.4.2 Prim's

Time Complexity: $O(V+E) \approx O(E)$.

1. Use an array of size 10 as PQ, each slot holding a linked list.
2. `decreaseKey` moves node from 1 linked list to another. $(O(E))$.

8.4.3 Directed MST

Only works for a **DAG with one "root"**.
Walk through the graph prioritising weights, maintaining `visited[]`.

8.4.4 Steiner (2-approx)

1. For every required vertex, calculate the shortest path.
2. Construct new graph with the required nodes.
3. Run MST and map new edges to original graph.

9 Dynamic Programming

9.1 Longest Increasing Subsequence

Time Complexity: $O(n^2)$.

1. Start from the last item.
2. Find the maximum outgoing edge and add 1 and store.
3. Recurse until the array is iterated through

9.2 Lazy Prize Collection

Time Complexity: $O(kE)$.

1. Subproblem: `P[v,k]` = maximum prize you can collect starting at v, taking k steps.
2. `P[v,k] = MAX{ P[`w_1`, k-1] + w(v,` w_1`), P[`w_2`, k-1] + w(v,` w_2`), ... }` where `v.nbrlist() = {`w_1`,` w_2`,` w_3`, ... }`

9.3 Vertex Cover

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

1. `S[v,0]` = size of vertex cover in subtree at `v` if `v` is NOT covered.

   `S[v,0] = S[`w_1`, 1] + S[`w_2`, 1] + ...`

2. `S[v,1]` = size of vertex cover in subtree at `v` if `v` is covered.

   `S[v,1] = 1 + min(S[`w_1`, 0], S[`w_1`, 1] + min(S[`w_2`, 0], S[`w_2`, 1]) + ...`

9.4 APSP (Floyd Warshall)

Time Complexity: $O(V^3)$.

1. `S[v,w,P]` = shortest path from `v` to `w` that only uses intermediate nodes in set P.
2. Use a 2D array and solve for S as we add more nodes into set P.
3. `S[v,w,`P_i`] = min (S[v,w,`P_{i-1}`], S[v,i,`P_{i-1}`] + S[i,w,`P_{i-1}`])`

9.4.1 Path Reconstruction

Reconstruct the shortest path by storing a `predecessor[]` and update whenever we update a node's shortest path.

9.4.2 Transitive Closure

Instead of finding the distance, we want to know if there exist a path.
Change to: `S[v,w,`P_i`] = OR (S[v,w,`P_{i-1}`], S[v,i,`P_{i-1}`] AND S[i,w,`P_{i-1}`])`

10 Recitation BS

10.1 MET

Iterate through 3 dimensions (ins, del, rep) to achieve the destination string. Since it is a DAG, topo-sort and run SSSP.

10.2 Leftist Heaps

Property: For every node that has two children L, R, `rightRank(L) >= righRank(R)`.

`insert` $O(log(n))$	Insert down the right spine. If `Pr(curr) > Pr(orig)`, swap the two nodes. Insert when a node with no right child is found. Update `rightRank` as the recursion unrolls.
`merge` $O(log(n))$	Walk down the right spine and compare the root. If the merging root is of higher priority, swap the subtree and recursively merge on the subtree. Walk down and check if any swaps are required for the LEFTIST property.
`extractMax` $O(log(n))$	Remove the root and call `merge` on the left and right subtree.
`delete` $O(log(n))$	Delete the node and swap if LEFTIST property is violated. Walk up and update `rightRank`.
`updateKey` $O(log(n))$	Call delete and insert.

10.3 Indirection

Group continuous sequence in an array to a node, and have a pointer to the next element.

10.4 Minimax

1. Use Binary Search.
2. Use Floyd Warshall/APSP.
3. Use MST then BFS/DFS.
4. Use Lowest Common Ancestor search tree.

Solution	Time Complexity		Space Complexity		
	Preprocess	Query	Preprocess	Query	DS
Binary search on query-time trimmed graph	0	$O((V+E)\log k)$	0	$O(V)$	N/A
Binary search on preprocessed trimmed graphs	$O(k(V+E))$	$O(\log k)$	$O(kV)$	$O(1)$	$O(kV)$
Modified relax SSSP with Dijkstra's	0	$O((V+E)\log V)$	0	$O(V)$	N/A
Modified relax APSP with Dijkstra's	$O(V(V+E)\log V)$	$O(1)$	$O(V^2)$	$O(1)$	$O(V^2)$
Modified relax APSP with Floyd-Warshall's	$O(V^3)$	$O(1)$	$O(V^2)$	$O(1)$	$O(V^2)$
Query-time MST†	0	$O(E\log V)$	0	$O(V)$	N/A
Preprocessed MST†	$O(E\log V)$	$O(V)$	$O(V)$	$O(V)$	$O(V)$
Preprocessed MST† + Binary search + LCA	$O(E\log V+V^2)$	$O(\log^2 V)$	$O(V\log V)$	$O(1)$	$O(V\log V)$