Algorithm: deterministic (same input = same output), effective (implementable with current hardware) and finite (has an end).
Array: typically stored in contiguous memory, can be statically or dynamically (linked lists) stored.
Bag: can add stuff but can't remove.

Tilde approximation

$\sim f(n)$ or $T(n)$ is an approximation of $f(n)$ where you drop everything but the highest order term since

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = 1$$

Order of growth approximation (Big O)

$O(f(n))$ is an approximation of $T(n)$ where you drop the constants. $O(f(n))$ is a set of functions that forms an **upper bound** of $T(n)$ past some n_0.

$$\exists c > 0 \quad \exists n_0 \geq 0 \quad \forall n \geq n_0 \quad 0 \leq T(n) \leq c \cdot f(n)$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \neq \infty$$

Big Omega

$\Omega(f(n))$ is a set of functions that forms a **lower bound** of $T(n)$ past some n_0.

$$\exists c > 0 \quad \exists n_0 \geq 0 \quad \forall n \geq n_0 \quad T(n) \geq c \cdot f(n)$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \neq 0$$

Big Theta

A combination of Big O and Big Omega where $\Theta(f(n))$ is a set of functions that forms an **upper and lower bound** of $T(n)$.

$$\exists c_1, c_2 > 0 \quad \exists n_0 \geq 0 \quad \forall n \geq n_0 \quad c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \notin \{0, \infty\}$$

A $T(n)$ with Big Theta must have the same Big O and Big Omega as Big Theta

Union-Find

Given N nodes, we want functions:
`connected`: are two nodes in the same group
`union`: connect two nodes
`find`: what group is a node in
`count`: how many groups are there (N minus number of successful unions)

Algorithm	Quick-Find	Quick-Union	Weighted QU
`Initialization`	$O(n)$	$O(n)$	$O(n)$
`Find`	$O(1)$	$O(n)$	$O(\lg n)$
`Connected`	$O(1)$	$O(n)$	$O(\lg n)$
`Union`	$O(n)$	$O(n)$	$O(\lg n)$

Quick find: let nodes be indexes for an array where their value is the group they're in.
Quick union: let nodes be indexes for an array where their value is their parent node's index.
Weighted QU: when performing `union` with quick union, connect the root node of the smaller tree to the root of the larger tree.
Path compression: when performing `find`, have every traversed node either point to the found root node or their own grandparent.

Shellsort

Have a sequence of h's where the last one is 1. For i in set of h, create subarrays with every ith element and perform insertion sort and put them back. After each iteration, the array is i-sorted. h is usually a $3x + 1$ sequence where best case $O(n \log n)$ and worst case $O(n^{1.5})$

Priority Queue

A queue with implicitly ordered items where you can pop off the smallest/largest item.
Stored in a binary heap structure (complete binary tree that maintains the heap property: where the children are always $\leq$ for max or $\geq$ for min).
Stored in an array where the children of a node k are in $2k$ and $2k + 1$ and for a child k, its parent is in $\lfloor k/2 \rfloor$ and indices start at 1.
Insertion and deletion have worst case $O(\lg n)$.
Nodes are placed according to swim and sink.

New nodes are inserted into the end of the binary heap and swam up.

Root node is popped off by swapping with the end node and sinking down the new root node.

Heapsort

1. Construct a binary heap with the input data
2. Sort by popping off the root node until there's no elements in the heap

Inner loop is longer than quicksort's and makes poor use of cache since array entry comparisons are rarely with nearby array entries.

Sorting

Stable: preserves the relative order of equal keys in the array (i.e., sorted by other keys if the given key is equal).

Algorithm	Stable?	In-place?	Running time	Extra space
selection sort	no	yes	N^2	1
insertion sort	yes	yes	between N and N^2	1
shellsort	no	yes	$N \lg N$ or $N^{6/5}$	1
quicksort	no	yes	$N \lg N$	$\lg N$
mergesort	yes	no	$N \lg N$	N
heapsort	no	yes	$N \lg N$	1

Insertion sort's running time depends on the order of items; it can outperform mergesort and quicksort when $N < 30$.
Quicksort's worst-case is quadratic when the pivot is the smallest or greatest element.

BST

Floor: the largest key in the BST less than or equal to the key we're flooring.
Ceiling: the smallest key in the BST greater than or equal to the key we're ceiling-ing.
Predecessor: the node with the largest key smaller than the current node's key (rightmost leaf on the left branch).
Successor: the node with the smallest key larger than the current node's key (leftmode leaf on the right branch).

Deletion

Lazy deletion: for a node, set its key to `null` to create a tombstone.
Hibbard deletion: to delete a node with key k, search for node t with key k:
0 children: delete t by setting its parent link to `null`.
1 children: delete t and connect its single child to t's parent.
2 children: t is replaced by x, the minimum key in t's right subtree and x's right child is x's replacement.

2-3

Worst case of $O(\lg n)$ and direct implementation is complicated since you have to maintain multiple node types.
Every path from the root to a leaf node has the same length.

Red-black

Always insert with red links and perform rotation and colour flip as necessary to maintain left-leaning property.

B-tree

Nodes are arrays that allow up to $M - 1$ key-link pairs per node, where a link is the address of a page, rather than a symbol table value. M can be very large, but it must be even. We maintain at least 2 key-link pairs at root and at least $M/2$ key-link pairs in other nodes. External nodes contain client keys, and have references to actual data. Internal nodes contain copies of keys to guide search. A special key $*$ known as a *sentinel* helps implement the B-tree and acts as a wild card entry that's defined to be less than all other keys.
Search: start at the root, find the interval for the search key and take the corresponding link. Search terminates in an external node.
Insertion: search for the new key, insert at the bottom, and split nodes with M key-link pairs on the way up the tree.

Hash tables

Works best when the number of keys stored is much less than all the possible keys. The array index for each element is computed using a hash function. *k hashes to $h(k)$*. $h(k)$ is the *hash value* of k. A *collision* occurs where more than one key hashes to the same slot. Can be solved with *chaining* where each slot in the array is a linked list or red-black tree. Can also use open addressing if you only want one value per slot. A good hash function should have $h(k_1) = h(k_2)$ for $k_1 \neq k_2$ unlikely and have *simple uniform hashing* where each key is equally likely to hash into any slot, independent of other elements.

Open addressing

Linear probing: $h(k, i) = (h'(k) + i) \bmod m$ with $i = 0, 1, \ldots, m - 1$ where h is the linear probing hash function and h' is a good hash function. i is called the *iterative index*. Suffers from the *primary clustering* problem since over several insertions, it takes more time to insert and longer search times.
Quadratic probing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ with $c_1, c_2 \neq 0$ and $i = 0, 1, \ldots, m - 1$. Works better than linear probing, but if two keys have the same initial probe position,

then their probe sequences are the same. Leads to a milder form of clustering called *secondary clustering*.
Double hashing: $h(k,i) = (h'(k) + ih''(k)) \bmod m$ where h' and h'' are good hash functions. Works better when $h''(k)$ is relative prime to the hash-table size m. Set m to prime and design h'' to always return a positive integer less than m.

Graphs

Path: sequence of vertices connected by edges.
Cycle: path whose first and last vertices are the same.
Connected: two vertices are connected when in the same path.
Adjacent: a vertex A is adjacent to another B if there's an edge from A to B.
Incident: an edge is incident to two vertices if it exists between them.
Degree of a vertex: the number of edges incident to it.
Indegree: the number of directed edges that point to that vertex.
Outdegree: the number of directed edges that come out of that vertex.
Self-loop: an edge that connects a vertex to itself.
Parallel: two edges that connect the same pair of vertices.

DFS

Used to check if two vertices are connected. Generates a tree from a graph where the tree depends on the order the graph's connections are stored.

```
DFS(x):
    mark x as marked
    for vertices v connected to x:
        if v isn't marked:
            edgeTo[v] = x
            DFS(v)
```

`edgeTo[]` is the *output tree* where its root is the initial vertex. Preprocessing space and time proportional to $O(|V| + |E|)$. A non-recursive DFS uses a stack in BFS instead of a queue. In theory, DFS is faster than union find since it provides a constant time guarantee, but in practice, the difference is negligible and union find is faster since it doesn't have to build a full representation of the graph and union find is online while DFS has to preprocess the graph.

BFS

Used to find the shortest path in an unweighted graph. Takes time proportional to $O(|V| + |E|)$. Only different from DFS is that BFS uses a queue instead of a stack.

```
put initial vertex into a queue
while queue isn't empty:
    take the next vertex from the queue
    if that vertex hasn't been marked:
        mark it as marked
        add its connected vertices to the queue
```

Connected components

An equivalence relation that satisfies reflexivity (v is connected to v), symmetry (if v is connected to w, then w connected to v), and transitivity (if v connected to w and w connected to x, then v connected to x). A connected component is a maximal set of connected vertices. DFS and BFS can be used to find a connected component given an initial vertex. For checking correctness, union find is better since it doesn't have to preprocess the graph every time.

Multiple-source shortest path

Use BFS, but initialize by enqueueing all source vertices.

Precedence-constrained scheduling

Put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order (or report that doing so is not possible). The order is not unique. A directed acyclic graph is required and solved by implementing a topological sort.
Topological sort: use DFS with a reverse postorder.
Preorder: put the vertex on a queue before the recursive calls.
Postorder: put the vertex on a queue after the recursive calls.
Reverse postorder: put the vertex on a stack after the recursive calls.

Strongly-connected components

Vertices v and w are strongly connected if there's a directed path from v to w and from w to v (i.e. a cycle containing v and w).
Kosaraju-Sharir: computes strongly connected components in a digraph G. First run DFS on G^R (G with edges reversed) and compute the reverse postorder. Run DFS on G, considering vertices in the order of the reverse postorder. Runtime is proportional to $O(|V| + |E|)$.

Minimum spanning tree

Spanning tree: a subgraph that's connected, acyclic, and includes all of the vertices.
Minimum spanning tree: a spanning tree whose weight (sum of the weights of its edges) is smaller than the weight of any other spanning trees of the same graph.
Cut: a partition of its vertices into two non-empty disjoint sets (for a graph G, split it into S and $G - S$).
Crossing edge: connects a vertex in one set with a vertex in the other.
Cut property: given any cut, the crossing edge of minimum weight is in the MST.
Cycle property: let C be any cycle and let f be the maximum weighted edge belonging to C, then the MST doesn't contain f.

Kruskal's algorithm

Create a MST T by considering edges in ascending order of weight. Add next edge to tree unless doing so would create a cycle. Proof is that adding an edge that creates a cycle wouldn't be in the MST according to cycle property and if it doesn't create an edge, it's in the MST by the cut property.

Cycle checking

Checking if adding an edge (v, w) creates a cycle with DFS is bad since it takes $O(|E||V|)$ (at most $O(|V|)$ dfs calls $|E|$ times); instead, use union find, maintaining a set for each connected component. If v and w are in the same component, then the edge creates a cycle. To add the edge to the tree, union components containing v and w.

Implementation

Use a minimum priority queue to store all edges. Perform delete minimum operation on the queue. For each edge e, check for a cycle. If e doesn't create a cycle, then add e to T. Continue until $|V| - 1$ edges are added to T. Worst case $O(|E| \lg |E|)$ since $O(|E|)$ for queue, all delete-min is $O(|E| \lg |E|)$, and cycle checking with union find is $O(|E| + |V|)$.

Prim's algorithm

Creates a MST T by starting with vertex v_0 and until there's $|V| - 1$ edges, consider edges incident to vertices in T, but disregard any edges with both endpoints in T, then add to T the edge with the minimum weight. Proof is that if S is a subset of vertices in current tree T, Prim adds the cheapest edge with exactly one endpoint in S and that edge is in the MST by the cut property.

Cheapest edge with only one endpoint in T

Checking all edges leads to $O(|E||V|)$ so instead maintain a priority queue. Lazy implementation is the queue stores edges incident to the vertices in S, taking $O(|E| \lg |E|)$. Eager implementation is the queue stores vertices adjacent to the vertices in S, taking $O(|E| \lg |V|)$.

Lazy implementation

Maintain a queue of edges with at least one endpoint in T, where priority of edge e is its weight. Delete minimum weight edge $e = (v, w)$ from the queue. Disregard if both endpoints v and w are marked (i.e. both in T). Otherwise, let w be the unmarked vertex: add to queue any edge incident to w (assuming other endpoint isn't in T) and add e to T and mark w.

Eager implementation

Maintain a priority queue of vertices connected by an edge to T where priority of vertex v is the minimum weighted edge connecting v to T. Delete the minimum vertex v and mark v to be in T. Update the priority queue by considering all edges $e = (v, w)$ incident to v. Ignore if w is already in T. Add w to queue if not already in T. If already on queue, then reduce priority of w if e becomes the more minimal weighted edge connecting w to T.

Kruskal vs. Prim

Kruskal adds edges, but Prim adds nodes until it forms an MST.