Algorithm Analysis

Upper bound: $T(n) = \boldsymbol{O}(f(n))$ if $T(n) \le cf(n)$ for all $n > n_0 > 0$ and $c > 0$
Lower bound: $T(n) = \boldsymbol{\Omega}(f(n))$ if $T(n) \ge cf(n)$ 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. $f(g(n)) \le c_1 g(n)$
6. $f(n)^{g(n)}$: $O\left(\left(n^{\log n}\right)^2\right)$ 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 > $(\log n)^a$ 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
- $T(n) \in \Theta(n^k)$ *if* $a < b^k$
- $T(n) \in \Theta(n^k \log n)$ *if* $a = b^k$
- $T(n) \in \Theta(n^{\log_b a})$ *if* $a > b^k$

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

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end - begin) / 2
        if key <= A[mid] then
            end = mid
        else begin = mid + 1
    return (A[begin]==key) ? begin : -1
```

Time complexity: $O(\log n)$
Precondition (true before function): array is of size $\boldsymbol{n}$, array is sorted
Postcondition (true after function): if element is in array, A[begin] = key
Invariant (r/s between variables that is always true):
- (end – begin) ≤ $\frac{n}{2^k}$ in iteration k

Loop invariant (invariant that is true after every iteration):
- A[begin] ≤ key ≤ A[end]

Peak Finding

```
FindPeak(A, n)
    if A[n/2+1] > A[n/2] then
        FindPeak (A[n/2+1..n], n/2)
    else if A[n/2–1] > A[n/2] then
        FindPeak (A[1..n/2-1], n/2)
    else A[n/2] is a peak
    return n/2
```

Time complexity: $T(n) = T\left(\frac{n}{2}\right) + O(1) = O(\log n)$
Limitations:
- Cannot find global peaks efficiently (must use $O(n)$ solution)
- Cannot handle duplicate values well (don't know where to recurse)

Steep peak: precisely greater than (not equals to)
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]

Bubble Sort

```
BubbleSort(A, n)
    repeat (until no more swaps):
        for j ←1 to n-1
            if A[j] > A[j+1] then swap(A[j], A[j+1])
```

Loop invariant: at the end of iteration **j**, the biggest **j** items are correctly sorted in the final **j** position of the array

Selection Sort

```
SelectionSort(A, n)
    for j ← 1 to n-1:
        find minimum element A[j] in A[j..n]
        swap(A[j], A[k])
```

Loop invariant: at the end of iteration **j**, the smallest **j** items are correctly sorted in the first **j** positions of the array

Insertion Sort

```
InsertionSort(A, n)
    for j ← 2 to n
        key ← A[j]
        i ← j-1
        while (i > 0) and (A[i] >key)
            A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

Loop invariant: 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 Sort

```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ← MergeSort(A[1..n/2], n/2);
        Y ← MergeSort(A[n/2+1, n], n/2);
    return Merge (X,Y, n/2);
```

Iterative:
1. Start by sorting in groups of 2
2. Then multiply the factor by 2 and sort again
3. Keep repeating till the factor is greater than the length of the array

Invariant: After k iterations, the first k elements in the final array are sorted, each pair of elements are correctly sorted.

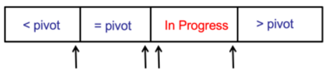
Quick Sort

```
QuickSort(A[1..n], n)
    if (n==1) then return;
    else
        p = partition(A[1..n], n)
        x = QuickSort(A[1..p-1], p-1)
        y = QuickSort(A[p+1..n], n-p)

partition(A[1..n], n, pIndex)     // Assume no duplicates, n > 1
    pivot = A[pIndex];            // pIndex is the index of the pivot
    swap(A[1], A[pIndex]);        // Store pivot in A[1]
    low = 2;                      // Start after pivot in A[1]
    high = n+1;                   // Define A[n + 1] = ∞
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]);
    return low–1;
```

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

Duplicates: running time $O(n^2)$ so use three way partitioning
- Option 1 (two pass partitioning):
  - Partition normally
  - Pack duplicates
- Option 2 (one pass partitioning):



Pivot choice:
- Pivot is good if it divides the array into two parts, each at least 1/10 the size of the array
- 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: keep repeating the partition algorithm with randomly chosen pivots until the partition is at least 1:9 ($T(n) = T(n-k) + T(k) + \frac{10}{8}cn$)
- Repetitions should take < 2 in expectation (p is the probability of picking a good pivot which is 8/10)
- $E[x] = (p)(1) + (1-p)(1 + E[x]) = 1$
  $pE[x] = 1$ $E[x] = \frac{1}{p}$
- Therefore, the expected probability is **10/8**

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

Algorithm	Best	Average	Worst	S	Pros	Cons
Bubble	O(n) (sorted)	O(n^2)	O(n^2) (smallest at end;	Y	Fast for sorted	Slow average case

			largest at front)			
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
- Finds the kth smallest element in an unsorted array (order statistics)

```
Select(A[1..n], n, k)
  if (n == 1) then return A[1];
  else Choose random pivot index pIndex.
    p = partition(A[1..n], n, pIndex)
    if (k == p) then return A[p];
    else if (k < p) then
      return Select(A[1..p–1], k)
    else if (k > p) then
    return Select(A[p+1], k – p)
```

Invariants:
- The pivot element will be in its exact position in the sorted array
- After every partition, the elements that are less than the pivot are found on the left (unsorted) and the elements that are more than the pivot are found to the right (unsorted)

Time complexity: $T(n) = T\left(\frac{n}{c}\right) + O(n) = O(n)$
Paranoid quick select: similar to paranoid quick sort

Dictionary Implementations (key-value stores)
Behaviours: insert, search, successor, predecessor, delete, contains, size
Types of data structure:
1. Sorted array
   a. Insert: O(n) (finding the right position)
   b. Search: O(log n) (binary search)
2. Unsorted array
   a. Insert: O(1) (append to the end)
   b. Search: O(n) (linear search)
3. Linked list
   a. Insert: O(1) (append to the start)
   b. Search: O(n) (linear search)

Binary Search Trees (Cannot assume balanced)
Key property: all in left subtree < key < all in right subtree
Height: number of edges on longest path from root to leaf
- If node is leaf, height = 0
  else if node is null, height = -1
  else, height = max(left.height, right.height) + 1

Depth: number of edges from the root to the current node
Minimum key: traverse to the left most node on the left subtree (from root)
Maximum key: traverse to the right most node on the right subtree (from root)
Searching O(h): given a target, check current node's key against target
- If target <= key, traverse left and repeat search
  else, traverse right and repeat search

Inserting O(h): given a target, search for the immediate parent and add to the left/right depending on the target value (only add when corresponding child does not exist yet)
Successor O(h): 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)

Tree shape: dependent on order of insertion (n! ways to insert and ~4^n shapes)
Tree traversals O(n):
1. Pre-order: self, left, right
2. In-order: left, self, right
3. Post-order: left, right, self
4. Level-order: level by level

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

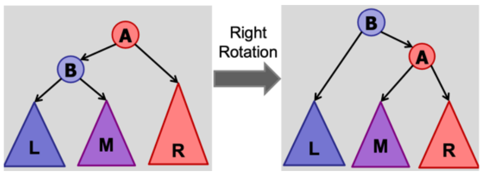
AVL
Augment: add height to every node and adjust every insert/delete (or store balance factor with 2 bits = |left| - |right| = {-1, 0, 1})

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

Claim: 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:

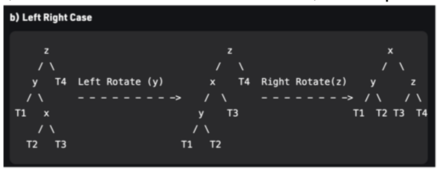
```
right-rotate(v)   // assume v.left != null
  w = v.left
  w.parent = v.parent
  v.parent = w
  v.left = w.right
  w.right = v
```



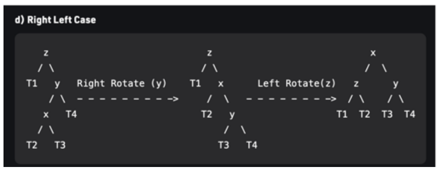
Heaviness: left/right heavy if left/right subtree has larger height than right/left subtree
Rotation order:
- **v is left-heavy**
  - v.left is balanced: right rotation on v
  - v.left is left-heavy: right rotation on v
  - v.left is right-heavy: left rotation on v.left, right rotation on v
- **v is right-heavy**
  - v.right is balanced: left rotation on v
  - v.right is right-heavy: left rotation on v
  - v.right is left-heavy: right rotation on v.right, left rotation on v

Number of rotations on insert: 2
- Insert, walk up, find first unbalanced (lowest), rebalance, return

Number of rotations on delete: log n
- Insert, walk up, for each unbalanced rebalance, walk up till root


b) Left Right Case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```


d) Right Left Case

```
     z                            z                            x
    / \                          / \                          /  \
  T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
      / \  - - - - - - - - ->      /  \   - - - - - - - -> / \    / \
     x   T4                      T2    y                  T1  T2  T3  T4
    / \                               /  \
  T2   T3                           T3    T4
```

(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: $(-\infty, v_1],\ (v_1, v_2],\ (v_2, v_3], \ldots, (v_{k-1}, v_k],\ (v_k, \infty)$
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): $O(\log_2 b \times \log_a n) = O(\log n)$
1. Find the appropriate key range using binary search on keylist
2. If not found, traverse to appropriate bucket where x in key range
3. Continue till found

Split: O(b) if the parent violates rule 1, split the parent too, till root
1. Find median key v_m in violating keylist
2. Split violating keylist into LHS and RHS using v_m

3. Add v_m to the keylist of the parent – O(b)
4. Adjust the children of the new keylist accordingly (make sure to unlink all subtrees of v_m and re-add later)
5. If node to split is root, let v_m be the new root

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))

```
w ← BTree.root
while true do
    if w contains b – 1 keys then
        v_m, y, z ← Split(w, v_m)
        if x ≤ v_m then
            w ← y
        else
            w ← z
    if w is a leaf then
        break
    else
        w ← GetSubtree(w, x) // subtree with keyrange containing x
InsertKey(w, x)
```

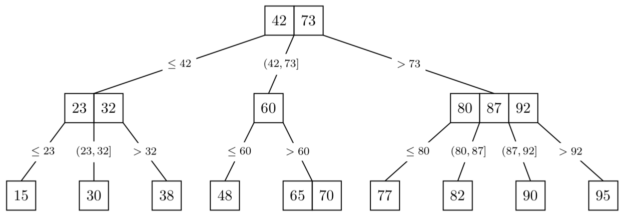
Insertion strategies:
1. Proactive (split before inserting): only applies to (a, b)-trees with b ≥ 2a
2. Passive (split after inserting while traversing up)

Merge(y, z): O(b) when y and z have < b – 1 keys together
1. In parent, delete key v that separates the siblings
2. Add to the keylist of y
3. In y, merge in z's keylist and treelist
4. Remove z

Share(y, z): O(b) when y and z have ≥ b – 1 keys together
1. Merge(y, z)
2. Split newly combined node

Delete (Passive): O(b × $\log_a$n) careful of orphaned nodes
1. Swap key to delete with successor/predecessor (same as BST)
2. Delete predecessor/successor node instead
3. Merge/Share depending on number of keys

__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

Tries: storing 1 letter per node/edge (does not depend on size of total text/number of strings)
- Root to leaf path represent strings (where the end nodes have flags)
- Searching is a DFS along the possible path (O(L) time)
- Space complexity: O((size of text) * overhead)

Complexity	**Tries**	**AVLs**
__Time__	O(L) (search/insert)	O(hL)
__Space__	O(text size)	O(text size)

- Tries has more nodes and more overhead
- AVLs assume strings take up space as well

__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)

__Dynamic Order Statistics__
- Select kth element in sorted array

Possible solutions:
1. Sorted array: O(1) search, O(n) insert
2. Unsorted array: O(n) search, O(1) insert

Augmenting: use AVL as underlying data structure
- Store __weight__ of each node (__weight__: size of the tree rooted at that node)
  - w(leaf) = 1
    w(v) = w(v.left) + w(v.right) + 1
  - Weight updated while inserting
- __Rank__ cannot be used since inserting a new number requires updating every rank which is too expensive (__rank:__ index in sorted array)

Derive rank from weights: rank of a node is the weight of the left subtree + 1

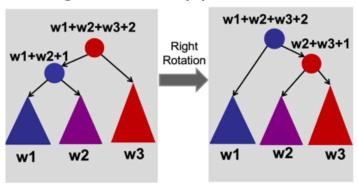
Select(k) O(log n): find node with rank k
- If rank of node <= k, move left
  else, move right and decrease k by rank of node
  repeat till k found

```
select(k)
    rank = m_left.weight + 1;
    if (k == rank) then
        return v;
    else if (k < rank) then
        return m_left.select(k);
    else if (k > rank) then
        return m_right.select(k–rank);
```

Rank(v) O(log n): computes rank of node v
- 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

```
rank(node)
    rank = node.left.weight + 1;
    while (node != null) do
        if node is left child then
            do nothing
        else if node is right child then
            rank += node.parent.left.weight + 1;
        node = node.parent;
    return rank;
```

Maintaining weights during rotation O(1):



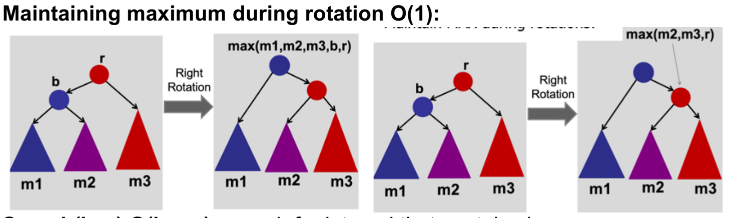
__Interval Trees__
- Find an interval that overlaps x

Possible solutions:
1. Sorted array (by minimum value of interval): O(n) search, O(n) insert
2. Running dictionary where keys are coverage and values are the indices: O(1) search, high space cost

Augmenting: uses AVL as underlying data structure
- Each node is an interval
- Sorted by minimum value
- Include maximum endpoint at each node of the subtree
  - max(n.high, left.max, right.max)

Insertion O(log n): same as AVL insertion where key is left endpoint; update max on the way up

Maintaining maximum during rotation O(1):



Search(key) O(log n): search for interval that contains key
- If key < left.max, move to the left
  else, move to the right
- **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

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

FindOverlaps(key) O(k log n): return all intervals that contain key (optimal is O(k + log n)

```
All-Overlaps Algorithm: O(k log n)
    Repeat until no more intervals:
        – Search for interval.
        – Add to list.
        – Delete interval.
    Repeat for all intervals on list:
        – Add interval back to tree.
```

__Orthogonal Range Searching (Dynamic Trees)__
- Searching points in a n-d plane

Space complexity: O(n $\log^{d-1}$n) (O(log n) nodes per node of the parent for each nested dimension)

Augmenting: use AVL as underlying data structure
- Store all points in the leaves of the tree
- Each internal node stores the max of any leaf in the left subtree
- Rotation to change max values

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 appear as leaves
- Repeat for each nested dimension

Split node: highest node where search includes both left and right subtrees
Query time complexity: O($\log^d$n + k)
1D range queries O(k + log n): note that halting condition is when node is a leaf; if leaf and leaf.key in [low, high] then add to result

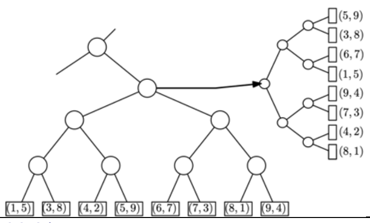
1. Find split node (node whose key is between [low, high]) – O(log n)
2. Perform left traversal (keep moving left if v.key > low)
   - Output all right subtree and recurse left – O(k) where k is the number of elements found
   - Recurse right – O(log n)
3. Perform right traversal (keep moving right if v.key > high)
   - Output all left subtree and recurse right – O(k) where k is the number of elements found
   - Recurse left – O(log n)

```
FindSplit(low, high)
    v = root;
    done = false;
    while !done {
        if (high <= v.key) then v=v.left;
        else if (low > v.key) then v=v.right;
        else (done = true);
    }
    return v;

LeftTraversal(v, low, high)
    if (low <= v.key) {
        all-leaf-traversal(v.right);
        LeftTraversal(v.left, low, high);
    } else {
        LeftTraversal(v.right, low, high);
    }

RightTraversal(v, low, high)
    if (v.key <= high) {
        all-leaf-traverasal(v.left);
        RightTraversal(v.right, low, high);
    } else {
        RightTraversal(v.left, low, high);
    }
```

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"

2D range queries O($\log^2$n + k): augment every node in the initial x-tree to store the y-tree containing all the points in the subtree within the interval (sorted by y values)



```
LeftTraversal(v, low, high)
    if (v.key.x >= low.x) {
        ytree.search(low.y, high.y);
        LeftTraversal(v.left, low, high);
    } else {
        LeftTraversal(v.right, low, high);
    }
```

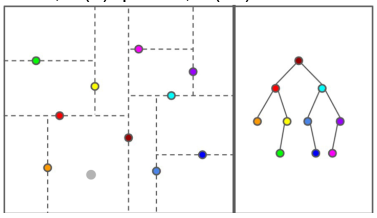
1. Find split node – O(log n)
2. Recurse on either sides – O(log n)
3. Search O(log n) y-trees – O(log n) each – O($\log^2$n)
4. Enumerate output – O(k)

Insertion/deletion: not supported as y-tree must be re-built entirely when x is rotated (O(n) operation)

- Thus, only use static 2D range trees with no insertions or deletions

__kd-Trees__

- Alternate levels (horizontal, vertical) per level of tree
- Each level divides 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

1. Sorted array:
   a. insert: O(n)
   b. extractMax: O(1)
2. Unsorted array
   a. insert: O(1)
   b. extractMax: O(n)
3. AVL tree (indexed by priority)
   a. insert: O(log n)
   b. extractMax: 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 (swapping with parent if new node priority larger)

```
insert(Priority p, Key k) {
    Node v = completeTree.insert(p,k);
    bubbleUp(v);
}

bubbleUp(Node v) {
    while (v != null) {
        if (priority(v) > priority(parent(v)))
            swap(v, parent(v));
        else return;
        v = parent(v);
    }
}
```

increaseKey O(log n): change priority and bubble up
decreaseKey O(log n): change priority and bubble down (focus on moving to where the larger priority is)

```
bubbleDown(Node v)
    while (!leaf(v)) {
        leftP = priority(left(v));
        rightP = priority(right(v));
        maxP = max(leftP, rightP, priority(v));
        if (leftP == maxP) {
            swap(v, left(v));
            v = left(v);
        } else if (rightP == maxP) {
            swap(v, right(v));
            v = right(v);
        } else return;
    }
```

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__

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

Insertion O(1): same as appending to end of the array the bubbling up is the same as swapping elements in the array with parent (see below)
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__

Inefficient:

1. Unsorted list to heap (heapify) – O(n log n) (insert for every element in array)
2. Heap to sorted list – O(n log n) (extract max until no more elements in heap)

Optimised heapify:

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

```
// int[] A = array of unsorted integers
for (int i=(n-1); i>=0; i--) {
    bubbleDown(i, A); // O(log n)
}
```

Cost of optimised heapify:

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

Analysis: in-place, faster than merge sort, a little slower than quick sort, deterministic (O(n log n) no matter the case), unstable

Union Find

Data structure to store the connectivity of objects using connected components

- Relies on transitivity property: if p <~> q and q <~> r, then p <~> r

Applications: checking connected locations in mazes; possibility of transition between game states; checking if two network locations are connected; checking which node in a tree network is the closest

Operations: find(p, q) – if p and q are in same set, union(p, q) – set p and q to be in same set

Quick find: use integer array; each element of array holds the component that the corresponding object belongs to

- Two objects are connected if they share the same component identifier
- Forms flat trees
- Expensive union
- **Find(p, q) O(1)**: component id of p == component id of q
- **Union(p, q) O(n):** set all components in q to component id of p

```
union(int p, int q):
  updateComponent = componentId[q]
  for (int i=0; i<componentId.length; i++)
    if (componentId[i] == updateComponent)
      componentId[i] = componentId[p];
```

Quick union: use integer array to represent parent of object; parents have the same index as the component identifier

- Two objects are connected if part of same tree
- Unbalanced trees
- Expensive union and find
- **Find(p, q) O(n):** reach root of both p and q and check if equal
- **Union(p, q) O(n):** reach the root of both p and q and set the parent of p to the parent of p to the parent of q

Weighted union: make the root with larger number of nodes the parent

- Maximum depth of tree is O(log n)
- Can use height to generate balanced trees as well
  - Weight/rank/size/height of subtree does not change except at the root
  - Weight/rank/size/height only increases when tree size doubles
- Tree of height k has size at least 2^k nodes
  - Height of tree of size n is at most log(n)

```
union(int p, int q):
  // go to roots of p and q
  if size[p] > size[q]:
    parent[q] = p
    size[p] += size[q]
  else:
    parent[p] = q
    size[q] += size[p]
```

Path compression: after finding the root, set the parent of each traversed node to the root

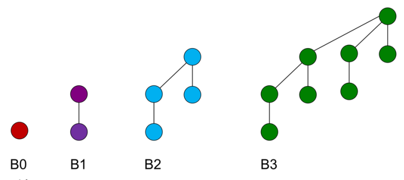
- Alternative: make every other node in the path point to its grandparent

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

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

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)

Binomial trees



B(n) = B(n-1) + B(n-1)

- Size(Bk) = O(2^k); height(Bk) = k -1
- Built using union find:
  1. Build initial binomial tree with union operations
  2. Union by creating new root
  3. Find the deepest leaf and perform path compression
  4. Repeat step 2

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
  - But too much space taken to fit all k keys
- If $k_1 = k_2$, then $h(k_1) = h(k_2)$

Collisions: when two distinct keys produce the same hash function

- Unavoidable as table size is smaller than universe size => pigeonhole principle states that there must exist two keys that map to the same bucket
- Solutions: a) new and better hash function (time consuming, hard to find, eventually find another collision); b) chaining (put same element into the same bucket); c) open addressing (find another bucket for the item)

Hash function design: goal to make it look random

1. Division method: h(k) = k mod m
   - Collision occurs when k = g mod m
   - Don't use m = 2^x (k mod 2^x = k – ((k >> x) << x) as if all input keys are even, then h(k) is even
     - k mod m + i*m = k
     - If k and m have a common divisor d, then k mod m is also divisible by d => only 1/d of the table is used
   - Use m = prime number where not too close to either a power of 2 or 10
   - Slow
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
   - A is an odd integer > 2^{w-1} because using even numbers would lose at least one bits worth and bit enough to use all bits in A
3. Tabulation hashing (fast and uniform)
4. Zobrist hashing (for board games)

Chaining

Each bucket contains a linked list of items where the all items have the same hash

- Total space: O(m + n) where m is the number of buckets and n is the number of entries (total size of all linked lists)
- Must store both the key and the value to identify when searching
- Operations:
  - Insert (O(1 + cost(h))): calculate hash and lookup hash, adding value to linked list
  - Search (O(n + cost(h)): calculate hash and search for value in linked list
    - Expected search time becomes O(1 + n/m) under SUHA
- If hash table is full, add more items

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

- Assume that this property holds for all hashing functions
- Better than inserting randomly as searching would be slow (O(n))
- 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 collisions: 1/m
- Unsuccessful search: n/m
- Successful search: 1 + (n – 1)/m <= 1 + n/m
- Expected maximum chain length: 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

- Potential to fail a search if no TOMBSTONE used
- TOMBSTONE overwritten if insert called

Types of hashing functions:

1. Linear probing
2. Weird probing: randomly select another bucket to probe after first fails
3. Double hashing
4. Quadratic probing

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)
  - If there is a cluster, then there is a higher probability that the next h(k) hits the cluster; also grows the cluster
- If table is ¼ full, then there will be clusters of size θ(log n) => average cluster size grows logarithmically
- In practice, linear probing is fast 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

- f(k, 1) is almost random and g(k) is almost random
- If g(k) is relatively prime to m, then h(k, i) hits all buckets
- If m = 2^r, then g(k) is odd

Performance: depends on load $\alpha = \frac{n}{m}$ which represents the total amount of space used

- For n items and m table size, expected cost of an operation (i.e. number of probes) is $\frac{1}{1-\alpha}$ (assuming uniform hashing)
  - Only when α is not too close to 1, closer values must re-calculate this expected cost

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

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 $\alpha \rightarrow 1$

Table resizing

Table growing:

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

Growth size	Resize cost	Insert cost	Total resize cost
m + 1	O(n)	-	O(n^2)
2m	O(n)	-	O(n) (average cost O(1))
m^2	O(n^2)	O(n)	O(n^2)

Table shrinking:
Half table size when less than ¼ elements

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

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

given p = probability of false positive

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

$$\text{p(false positive)} = \left(1 - e^{-\frac{kn}{m}}\right)^k$$
$$\text{Optimal } k = \frac{m}{n}\ln 2$$
$$\text{Error probability:} 2^{-k}$$

   - 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

(Simple) path: set of edges that connect two nodes, intersecting each node at most once
Connected: every pair of nodes is connected by a path
Disconnected: some pair of nodes is not connected by a path
Cycle: "path" where first and last node are the same (not exactly a path since first node visited twice)
(Unrooted) tree: connected graph with no cycles
Forest: graph with no cycles (can be disconnected)
Degree of node: number of adjacent edges to a node

- In-degree: number of incoming edges
- Out-degree: number of outgoing edges

Degree of graph: maximum degree of across all nodes
Diameter: maximum distance between two nodes following the shortest path (i.e. length of BFS from one node to another)
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: $\delta(S,C) \leq \delta(S,A) + \delta(A,C)$
Estimate: every node has an 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; dist[v] > dist[u] + weight(u, v)
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

- Can relax vertex of V – 1 times instead since the further node is at most V – 1 from source (imagine chain)
- 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

- Check: run Bellman-Ford for 1 more iteration
- Set nodes in cycle to constant: run Bellman-Ford another |V| times (does not work for negative cycle, only weights)

Same weight: use BFS
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

- Right order exists if all edges are non-negative so extending a path does not shorten it

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

1. Maintain distance estimate for every node
2. Begin with empty shortest path tree
3. Repeat
   a. Consider node with minimum estimate
   b. Add node to shortest path tree
   c. Relax all outgoing edges
   d. Add all unvisited nodes
4. 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()) {
      int w = pq.deleteMin();
      for (Edge e : G[w].nbrList) relax(e);
    }
  }

  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]);
    }
  }
}
```

Time complexity: O((V + E) log V) = O(E log V) if using AVL priority queue

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

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

- Pre-order DFS: append to order
- Post-order DFS: prepend to order

Using Khan's algorithm: repeat algorithm till no more nodes to add; O(V + E)

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

- Use hashmap to store the number of incoming edges instead of

Spanning tree

Acyclic subset of edges that connect all nodes

- Harder with directed graphs since the topological ordering must be preserved

Minimum Spanning Tree

Spanning tree with minimum weight

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

Applications: telephone/electrical/computer networks, ethernet autoconfig, road network, bottleneck paths, error correcting codes, face vertification, cluster analysis, image registration

Assumption: all edge weights are distinct (otherwise tie breakers)

  - Result: unique MSTs

Cut: partition of vertices V into two disjoint subsets

Crossing a cut: edge that ha 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: if C is a cycle with no red arcs, then color the max-weight edge in C red

Blue rule: if D is a cut with no blue arcs, then 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})	**Kruskal's modification:** Store linked list of edges with given weight j in array of size b (b is the upper bound); iterate over all edges in ascending order and for each edge, check if edge needs to be added (from UF) and union; total O(αE) where α is the time for UF **Prim's modification:** Implement priority queue using size b (similar array as above) 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"	Cut and cycle property does not hold, generic MST algorithm does not work For every node except root, add minimum weight incoming edge. Since every node has to have at least one incoming edge in the MST, the resulting minimum weight is the MST 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)

Same weight graphs: use DFS/BFS to find MST (O(log E))

- Total cost is n(V-1) where weight is n

Prim's Algorithm

Starting from an arbitrarily chosen start node, identify the cut {S, V-S} and find the minimum weight edge on the cut and add the connected node to the MST. Repeat till no more edges can be added

- Each added edge is the lightest on some cut so each edge is in the MST

Time complexity: O(E log V)

```
Create priority queue of all vertices and give them all priority of INF
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:
        Extract the minimum vertex to process
        Add the current vertex to the hash set
        For every neighbor of the vertex:
                If neighbor has not been seen before:
                        Lower the priority of the neighbor
                        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)

- Use union find to connect nodes if they are in the MST
- Each added edge crosses a cut that is lightest cut; all other lighter edges across the cut have already been considered

Time complexity: O(E log V)

```
// Sort edges and initialize
Edge[] sortedEdges = sort(G.E());
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++) {
  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?
    mstEdges.add(e); // save edge
    uf.union(v,w); // combine trees
  }
}
```

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
  - Takes O(V + E) time to run
  - 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)): each element in the array is mapped to a node with outgoing edges to the elements strictly larger than itself

- Calculate the longest paths per node (SSSP on DAG), relaxing edges in topological order

Subproblem: 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
```

- Solution above is O(n^2)

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 graph into DAG and make k copies of every node where edges between nodes denote a traversal; solve using DAG SSSP for longest path

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

DP O(kV^2) = O(kE) (fully connected graph): given optimal solution of k – 1, optimal solution of k is trivially solved

- Subproblems: 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: 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 (never true but used for base case) 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
}
```

Time complexity: O(V)

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

Longest simple path

Longest path of simple path without any weights is NP-hard

Important time complexities

Algorithms

BFS:

```
BFS(Node[] nodeList, int startId) {
  boolean[] visited = new boolean[nodeList.length];
  Arrays.fill(visited, false);
  int[] parent = new int[nodelist.length];
  Arrays.fill(parent, -1);
  Collection<Integer> frontier = new Collection<Integer>;
  frontier.add(startId);
  // Main code goes here!
  visited[startId] = true;
  while (!frontier.isEmpty()){
    Collection<Integer> nextFrontier = new Collection<Integer>;
    for (Integer v : frontier) {
        for (Integer w : nodeList[v].nbrList) {
            if (!visited[w]) {
              visited[w] = true;
              parent[w] = v;
              nextFrontier.add(w);
            }
       }
       frontier = nextFrontier;
    }
  }
}
```

```
// For covering all connected components in disconnected graph
for (int start = 0; start < nodeList.length; start++) {
  if (!visited[start]) {
    Collection<Integer> frontier = new Collection<>();
    frontier.add(start);
    // Main code goes here!
  }
}
```

DFS:

```
DFS-visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
    if (!visited[v]){
    visited[v] = true;
    DFS-visit(nodeList, visited, v);
    }
  }
}

DFS(Node[] nodeList) {
  boolean[] visited = new boolean[nodeList.length];
  Arrays.fill(visited, false);
  for (start = i; start<nodeList.length; start++) {
    if (!visited[start]) {
      visited[start] = true;
      DFS-visit(nodeList, visited, start);
    }
  }
}
```