Sorting

Bubble Sort

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

✓ In-place ✓ Stable

At the end of iteration j of the outer loop, the biggest j items correctly stored in the final j positions of the array.

Worst / average $O(n^2)$, best $O(n)$ when array is already sorted

Insertion Sort

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

✓ In-place ✓ Stable

A[1..j-1] is sorted → the first few elements are *relatively* sorted.

Worst / average $O(n^2)$, best $O(n)$ when array is already sorted

Selection Sort

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

✓ In-place ✗ Not stable

Always $O(n^2)$.

At the end of iteration j, the smallest j items are correctly stored in the first j positions of the array.

Minimizes no. of times data is written (only $2n$ writes). Good if you have very large data to sort as comparing is cheap.

Merge Sort

```
mergeSort(A, n)
  if (n == 1) 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);
```

✗ Not in place → space complexity $O(n)$ for arrays, $O(\log n)$ for linked lists

✓ Stable

$O(n \log n)$ but slow when there are small number of items to sort → at base case, use InsertionSort

Quick Sort

```
paranoidQuickSort(A[1..n], n) // T(n)
  if (n == 1) then return;
  else
    repeat // expected: executed max 2 times
      pIndex = random(1, n);
    until p > (1/10)n and p < (9/10)n
  p = partition(A[1..n], n, partitionIndex);
  x = quickSort(A[1..p-1], p-1); // T(k)
  y = quickSort(A[p+1..n], n-p); // T(n-k)
```

$O(n \log n)$ ✓ In-place ✓ Stable

Choice of Pivot: A pivot is good if it divides the array into two pieces, each of which is size at least n/10.
Improvement: Halt recursion early, leaving small arrays unsorted. Perform InsertionSort on entire array.

Partition

```
partition(A[1..n], n, pIndex)
  pivot = A[pIndex];
  swap(A[1], A[pIndex]);
  low = 2; high = n+1;
  while (low < high)
    while ((A[low] < pivot) and (low<high))
      low++;
    while ((A[high] > pivot) and (low < high)):
      high−−;
    if (low < high)
      swap(A[low], A[high]);
  swap(A[1], A[low−1]);
  return low−1;
```

$O(n)$

Assumes no duplicates, and pIndex is the index of pivot. If there are duplicate pivots, pack them.

Quick Select

```
quickSelect(A[1..n], n, k)
  if (n == 1) return A[1];
  else
    choose random pivot index pIndex;
    p = partition(A[1..n], n, pIndex);
    if (k == p) return A[p];
    else if (k < p)
      return quickSelect(A[1..p-1], k);
    else
      return quickSelect(A[p+1..n], k-p);
```

$O(n)$

From an array A of n elements, returns the kth smallest element.

Only recurses once. Take note that recursing left and right are not exactly the same. But dominated by the $O(n)$ operation that is partition.

Knuth Shuffle

```
KnuthShuffle(A[1..n])
  for i = 2 to n do
    r = random(1, i)
    swap(A, i, r)
```

Generates a random permutation of an array. It does so by iterating through an array, creating a random prefix.

Searching

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

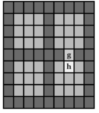
Loop Invariant: A[begin] ≤ key ≤ A[end]

Precondition: Monotonicity in the search sequence.

```
findPeak1D(A, n)
  if (A[n/2] is a peak)
    return n/2
  else if (A[n/2 + 1] > A[n/2])
    findPeak1D(A[n/2+1..n], n/2)
  else if (A[n/2 - 1] > A[n/2])
    findPeak1D(A[1..n/2-1], n/2)
```

Returns a local maximum element / peak from some array A[1..n]. A[i] is a local maximum iff A[i-1] ≤ A[i] and A[i+1] ≤ A[i]. $O(\log n)$

```
findPeak2D() // T(n,m)
  find max element in the
  border and cross; // O(n + m)
  if (that element is a peak)
    return that element;
  else // T(n/2, m/2)
    recurse on the quadrant
    containing the element bigger than max
```



Given a 2D array A[1..n,1..m], returns a peak that is not smaller than the (at most) 4 neighbours.
$T(n,m) = T(n/2, m/2) + cn + cm = O(n + m)$

Hashing

```
insert(key, value) // Average≤1/(1-α)
  int num_collisions = 1;
  while (num_collisions <= m)
    int bucket = h(key, num_collisions);
    if (T[bucket] == null){
      T[bucket] = {key, value};
      return success;
    num_collisions++;
  throw new TableFullException();

delete(key)
  find key to delete;
  remove it from the table;
  set bucket to DELETED;
```

Open Addressing

When you insert items, on collision, probe a sequence of buckets in the hash table until you find an empty one. **Load** α refers to the average number of items per bucket, ie. $\alpha = \frac{n}{m}$.

Separate Chaining

Given n items, a hash table of these entries would have m (prime) buckets, each bucket containing a linked list of items to store multiple keys. **Space Complexity:** $O(m + n)$
Insert: $O(1)$
Search: Worst $O(n)$, exp. $O(1 + n/m)$
Delete: $O(1 + n/m)$

Hash Functions

Simple Uniform Hashing Assumption: Every key is equally likely to map to every bucket, and keys are mapped independently.

Linear Probing:
$h(key, i) = (h(key) + i)\%m$

Double Hashing:
$h(key, i) = (f(key) + i * g(key))\%m$

Resizing

growTable: if (n == m) then m = 2m;
shrinkTable: if (n < m/4) then m = m/2;
Inserting k elements takes time $O(k)$ so the insert operation has amortized cost $O(1)$

Recurrences

If $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$, then $T(n) = \begin{cases} O(n^d \log n), & if\ a = b^d \\ O(n^d), & if\ a < b^d \\ O(n^{\log_b a}), & if\ a > b^d \end{cases}$

$\sum_{i=1}^{n} i^c$	$\sum_{i=1}^{n} \frac{1}{i}$	$\sum_{i=1}^{n} c^i$	$\sum_{i=1}^{n} \log(i)^c$	$\sum_{i=1}^{n} \log(i)^c \cdot i^d$	$\sum_{i=1}^{n} \log(i)^c \cdot i^d \cdot b^i$
$\theta(n^{c+1})$	$\theta(\log n)$	$\theta(c^n)$	$\theta(n \log(n)^c)$	$\theta(n^{d+1} \cdot \log(n)^c)$	$\theta(n^d \cdot \log(n)^c \cdot b^n)$

Amortized Analysis

Operation has amortized cost T(n) if **for every integer k**, the cost of k operations is ≤ k T(n).

Skip Lists

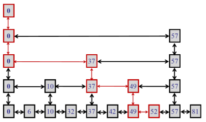
In a **skip list** with two lists, the express list skips √n elements per stop, and search takes O(√n). A proper skip list has log n levels.

```
search(key) // O(log n)
  node = root
  while (node.key < key)
  and (node.level > 1):
    while (node.next.key < key)
      node = node.next
    node = node.down
  while (node.next.key < key)
    node = node.next
  return node.key == key
    ? node : null
```

Invariant: The bottom list contains every element, and about half of the elements in level j gets promoted to level j + 1.

Every search and insert operation completes in $O(\log n)$ time with high probability of at least $1 - 1/n$.

```
insert(key) // O(log n)
  k = 0
  while (!done):
    flip a fair coin:
      with probability ½ :
        done = true
      with probability ½ : k++
```



Fingerprint Hash Tables

Implements a Set ADT which maintains a vector of 0/1 bits.

```
insert(key):m_table[hash(key)] = 1;
lookup(key):return (m_table[hash(key)] == 1);
```

Even if the key is not in the set, it may sometimes report true. For a lookup,
$P(no\ false\ positive) = P(no\ collision) = (1 - \frac{1}{m})^n \approx (\frac{1}{e})^{\frac{n}{m}}$

Bloom Filter

A fingerprint hash table that makes use of more than one hash functions.

```
insert(URL):
  k1 = hash1(URL); k2 = hash2(URL); T[k1] = 1; T[k2] = 1;
query(URL): // requires 2 collisions for false +ve
  k1 = hash1(URL); k2 = hash2(URL);
  if (T[k1] && T[k2]) return true;
  else return false
```

For a bloom filter with k hash functions, assuming (bogus) fact that each table bit is independent,
$P(a\ given\ bit\ is\ 0) = (1 - \frac{1}{m})^{kn} \approx e^{(-\frac{kn}{m})}$
$P(false\ positive) = (1 - (\frac{1}{e})^{\frac{kn}{m}})^k$
Optimal k is $(\frac{m}{n}) \ln 2$.

BSTs

```
height()
  leftHeight = -1;
  rightHeight = -1;
  if (m_leftTree != null)
    leftHeight = m_leftTree.height();
  if (m_rightTree != null)
    rightHeight = m_rightTree.height();
  return max(leftHeight, rightHeight)+1;
```

Height: Number of edges on the longest path from the root to the leaf.

Largest possible height: $\theta(n)$. A tree is **balanced** if $h = O(\log n)$. **Duplicate values:** Treat each node as a bucket of keys, store a value corresponding to bucket size.

Search

```
search(queryKey) // O(h)
  if (queryKey.compareTo(key) < 0)
    if (leftTree != null)
      return leftTree.search(queryKey);
    else return null;
  else if (queryKey.compareTo(key) > 0)
    if (rightTree != null)
      return rightTree.search(queryKey);
    else return null;
  else return this;

searchMax()
  if (rightTree != null)
    return rightTree.searchMax();
  else return this;
searchMin()
  if (leftTree != null) {
    return leftTree.searchMin();
  else return this;

successor()
  if (this.right != null)
    return this.right.searchMin();
  else
    parent = this.parent;
    child = this;
    while ((parent != null)
    && (child == parent.right))
      child = parent;
      parent = child.parent;
    return parent;
```

Traversal

```
inOrderTraversal() // O(n)
  if (leftTree != null)
    leftTree.inOrderTraversal();
  visit(this);
  if (rightTree != null)
    rightTree.inOrderTraversal();
```

IOT: Operates on each vertex in **sorted order**.

```
preOrderTraversal()
  visit(this);
  if (leftTree != null)
    leftTree.preOrderTraversal();
  if (rightTree != null)
    rightTree.preOrderTraversal();

postOrderTraversal()
  if (leftTree != null)
    leftTree.postOrderTraversal();
  if (rightTree != null)
    rightTree.postOrderTraversal();
  visit(this);
```

Useful property: Any binary tree can be uniquely identified by its pre-order and in-order traversal or its post-order and in-order traversal.

Rebalance Node

```
rebalance(node)
  if (node is right-heavy):
    if (node.right is balanced
    || node.right is right-heavy)
      leftRotate(node);
    else if (node.right is left-heavy):
      rightRotate(node.right);
      leftRotate(node);
  else if (node is left-heavy):
    if (node.left is balanced
    || node.left is left-heavy):
      rightRotate(node);
    else if (node.left is right-heavy):
      leftRotate(node.left);
      rightRotate(node);
```

Balance Factor: In an AVL tree, the balance factor for every node is within [−1,1]. Rotation is necessary otherwise.

$balanceFactor = height(left\ subtree) - height(right\ subtree)$

Tries

One letter is stored in each node. Root-to-leaf paths represent strings.

search(string) runs in $O(L)$

Space: $O(size\ of\ text \times overhead)$

Order Statistics Trees

In an **Order Statistics Tree**, we also store the **size** of the subtree in every node. The **weight** of a node is the size of the tree rooted at that node. This weight attribute must be maintained whenever a rotation occurs.
$w(leaf) = 1$,
$w(v) = w(v.left) + w(v.right) + 1$

Rank

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

Select

Finds the node of rank k within the tree.

```
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);
```

AVL Trees

Insertion

```
insert(x)
  if (x < key) left.insert(x);
  else right.insert(x);
  height = max(left.height,
               right.height) + 1;
  walk up tree and for every step
    check for balance;
    if out of balance:
      use rotations to rebalance;
```

After every insertion, you will have to update the height and rebalance the tree. **Worst case**: only **two** rotations are required.

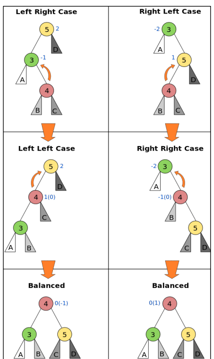
Deletion

Deletion can take up to $O(\log n)$ rotations.

```
delete(v)
  if (v has no children):
    remove(v);
  else if (v has only one child):
    remove (v);
    connect(child(v), parent(v));
  else if (v has two children):
    x = successor(v);
    swap(v, x);
    delete v (which is in the
    original pos of the successor);
  update heights up to the root;
  for every ancestor of deleted node:
    if it is not height-balanced:
      perform a rotation;
```

Rotation

```
rightRotate(node) // left != null
  oldParent = node.parent;
  newParent = node.left;
  grandChild = newParent.right;
  connect(newParent, node);
  connect(team, grandChild);
  connect(oldParent, newParent);
leftRotate(node) // right != null
  oldParent = node.parent;
  newParent = node.right;
  grandchild = newParent.left;
  connect(newParent, team);
  connect(team, grandchild);
  connect(oldParent, newParent);
```



Interval Trees

Each node is an interval (low, high) and contains the maximum endpoint **max** in its subtree. It is sorted by left endpoint. For every insertion, the **max** has to be updated every node along the path from the inserted node to the root.

Interval Search

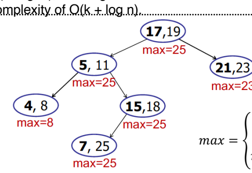
```
intervalSearch(x) // O(log n)
  c = root;
  while (c != null and
  x is not in c.interval)
    if (c.left == null)
      c = c.right;
    else if (x > c.left.max)
      c = c.right;
    else
      c = c.left;
  return c.interval;
```

Finds an interval that contains x.
Correctness: If the search goes right, then there is no overlap in the left subtree → either search finds key in the right subtree or it is not in the tree. If the search goes left, and there is no overlap in the left subtree → there is no overlap in the right subtree either.

All Overlaps

```
allOverlaps(x) // O(k log n)
  repeat until no more intervals:
    search for an interval containing x;
    add that interval to list;
    delete that interval;
  repeat for all intervals in list:
    add the interval back in tree;
```

Lists all the intervals that overlap with point x. The time complexity if there are k overlapping intervals is O(k log n), although the best known solution has a time complexity of O(k + log n).



$max = \begin{cases} high, \\ left.max, \\ right.max \end{cases}$

Range Trees

Store all points in the leaves, whereas internal nodes only store copies. Each internal node v stores the maximum of any leaf in its left subtree.

1D Range Query

```
rangeQuery(low, high)
  v = findSplit(low, high); // O(log n)
  LeftTraversal(v.left, low, high);
  RightTraversal(v.right, low, high);
```

A one-dimensional range query; will return all the points in the tree that fall between the range of low and high.

Space Complexity: $O(n)$
Preprocessing Time Complexity: $O(n \log n)$
Time Complexity: $O(k + \log n)$, where k is the number of points found.

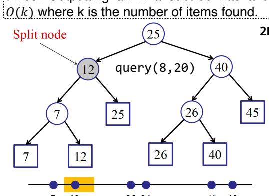
```
findSplit(low, high) // O(log n)
  v = root;
  done = false;
  while (!done) {
    if (high <= v.key) then v=v.left;
    else if (low > v.key) then v=v.right;
    else done = true;
  return v;
```

A **split node** is the first node from the root in which its point value is between the range query.

1D Range Traversal

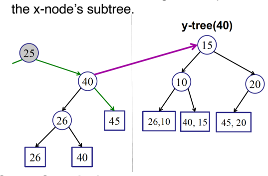
```
LeftTraversal(v, low, high)
  if (low <= v.key)
    all-leaf-traversal(v.right);
    LeftTraversal(v.left, low, high);
  else
    LeftTraversal(v.right, low, high);

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

Traversal algorithm recurses at most $O(\log n)$ times. Outputting all in a subtree has a cost of $O(k)$ where k is the number of items found.



2D Range Trees

1. Create a 1D-range tree (using only x-coordinates) of the points (ie. the x-nodes.
2. Store a y-tree (using only y-coordinates) at each x-node containing all the points in the x-node's subtree.



Space Complexity: $O(n \log n)$
Each point in the x-tree appears at most $O(\log n)$ y-trees (ie. at most one y-tree per level). The rest of the x-tree takes $O(n)$ space.
BuildTree Time Complexity: $O(n \log n)$
Inserting / Deleting Nodes: $O(n)$. On every insertion / deletion, you will have to do a rotation, and every rotation requires us to entirely rebuild the y-trees for the rotated nodes.

2D Range Query

```
2DRangeQuery(low, high)
  v = findSplit(low, high); // O(log n)
  2DLeftTraversal(v, low, high);
  2DRightTraversal(v, low, high);
```

Given a 2D array of points, searches for all the points that fall within a rectangular box.
$O(\log n)$ to find split node, $O(\log n)$ $O(\log n)$ y-tree searches of $O(\log n)$, $O(k)$ enumerating output

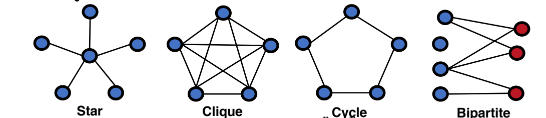
```
2DLeftTraversal(v, low, high)
  if (v.key.x >= low.x)
    ytree.search(low.y, high.y); // O(log n)
    2DLeftTraversal(v.left, low, high);
  else
    2DLeftTraversal(v.right, low, high);
```

For d-dimensional range trees, recursively construct a (d-1)-dimensional range tree in each node of a 1D-range tree.
Query cost: $O(\log^d n + k)$
BuildTree cost: $O(n \log^{d-1} n)$
Space complexity: $O(n \log^{d-1} n)$

Graphs:

A graph is represented as G = <V, E>
V is the set of **nodes**, where $|V| > 0$
E is the set of unique **edges**

A **tree** has $E = O(V)$, and a **complete graph** has $E = O(V^2)$.



	Star	Clique	Cycle	Bipartite
Diameter:	2	1	$\frac{n}{2}$ or $\frac{n}{2} - 1$	(maximum) $n - 1$
Degree:	$n - 1$	$n - 1$	2	

(**Degree** refers to the maximum number of adjacent edges) (**Diameter** refers to the maximum distance between two nodes, following the shortest path)

Representations

	Adjacency List *for sparse graphs*	**Adjacency Matrix** *for dense graphs* $\lvert E\rvert = \theta(V^2)$
Are v and w neighbors?	Slow	Fast
Find me any neighbor of v.	Fast	Slow
Enumerate all neighbors.	Fast	Slow
Memory Usage	$O(V+E)$, or $O(V)$ for a cycle	$O(V^2)$
	Store nodes in an array of size $\lvert V\rvert$. Each node contains a linked list of edges (along with weights if any), with a total size $\lvert E\rvert$.	A two-dimensional array (of size $\lvert V\rvert \times \lvert V\rvert$) where A[v][w] = 1 iff $(v, w) \in E$. **Finding n-hop neighbors:** Let $B = A^n$, and B[v][w] = 1 iff v and w are n-hop neighbors.

Breadth-First Search

```
BFS(Node[] nodeList)
  boolean[] visited
    = new boolean[num_nodes];
  Arrays.fill(visited, false);

  int[] parent = new int[num_nodes];
  Arrays.fill(parent, -1);

  for (int start = 0 to num_nodes)
    if (!visited[start])
      Bag<Integer> frontier
        = new Bag<Integer>;
      frontier.add(startId);

      while (!frontier.isEmpty())
        Collection<Integer> next
          = new Collection<Integer>;
        for (Integer v : frontier)
          for (Integer w : nodeList[v].nbrList)
            if (!visited[w])
              visited[w] = true;
              parent[w] = v;
              next.add(w);
        frontier = next;
```

For **unweighted / all-same-weight** graphs (can also use *dummy vertices* to transform weighted graphs in order to use BFS for SSSP problems). Parent pointers will store the path with a minimum number of hops.

$O(V+E)$ for adjacency list
Each vertex v is added to the next frontier only once, and after visited, it is never re-added → $O(V)$. Each v.nbrlist is enumerated once → $O(E)$.

Fails to visit all nodes if the graph is **disconnected**.

1. Explore the graph level by level, the current level being the frontier.
2. Save the nodes for the next level in next, skipping already-visited nodes.
3. Advance the frontier.

Iterative Version: Add start-node to a queue. Repeat until queue is empty: Remove node v from front of the queue, visit v and explore all outgoing edges of v, add all unvisited neighbors of v to the queue.

Depth-First Search

```
DFS(Node[] nodeList){
  boolean[] visited = new boolean[num_nodes];
  Arrays.fill(visited, false);
  for (start = 1; start < num_nodes; start++) {
    if (!visited[start]){
      visited[start] = true;
      ProcessNode(v) → only for pre-order DFS!
      visit(nodeList, visited, start);
      ProcessNode(v) → only for post-order DFS!

visit(Node[] nodeList, boolean[] visited, int startId){
  for (Integer v : nodeList[startId].nbrList) {
    if (!visited[v]){
      ProcessNode(v) → only for pre-order DFS!
      visited[v] = true;
      ProcessNode(v) → only for post-order DFS!
      visit(nodeList, visited, v);
```

$O(V+E)$ for adjacency list, $O(V^2)$ for adjacency matrix
Visit method is called only once per node → $O(V)$. In visit, each neighbor is enumerated → $O(E)$ for adj. list, $O(V)$ for adj. matrix.
Space complexity: $O(V)$ as it requires a recursion stack and visitation table.

Basic Idea: Follow path until you get stuck. Backtrack until you find a new edge. Recursively explore it. Don't repeat a vertex.

Iterative Version: Add start-node to a stack. Repeat until stack is empty: Pop node v from front of the stack, visit v and explore all outgoing edges of v, push all unvisited neighbors of v on the front of the stack.

Topological Sort

A topological sort (not unique) is done only on a **Directed Acyclic Graph (DAG)** and it results in a total ordering of the nodes, where all edges point forwards.

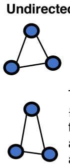
Approach 1:
Post-order DFS, where ProcessNode(v) is replaced by schedule.prepend(v) → $O(V+E)$

Approach 2: $O(V+E)$
Repeat the following: Let S = all nodes in G that have no incoming edges. Add nodes in S to the topological-order. Remove all edges adjacent to nodes in S. Remove nodes in S from the graph.

For a DAG, solving the SSSP problem simply means relaxing the nodes in the DFS post-order (topological order) → $O(E)$ as we just need to relax edges in a single pass.

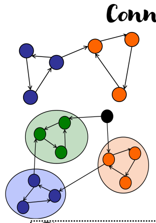
Connected Components

Undirected Graphs:



Vertex v and w are in the same **connected component** if there is a path from v to w.

There is a set $\{v_1, v_2, \cdots, v_k\}$ where there is no path from any v_i to v_j ⇔ there are only k connected components.



Directed Graphs:
In a **strongly connected component**, for every vertex v and w, there is a path from v to w and there is a path from w to v.

The graph of strongly connected components is acyclic.

Bellman-Ford

```
int[] dist = new int[num_nodes];
Arrays.fill(dist, INFTY);
dist[start] = 0;
for (i = 0; i < num_nodes; i++)
  for (Edge e : graph)
    relax(e)

relax(int u, int v){
  if (dist[v] > dist[u] + weight(u,v))
    dist[v] = dist[u] + weight(u,v);
```

Terminate early when an entire sequence of $|E|$ relax operations have no effect.

Fails when graph has **negative weight cycles** (ie. a cycle with weights that sum to a negative number) $O(EV)$

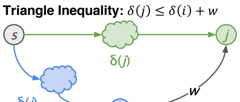
Invariant: At every relax operation, we attempt to reduce the estimate of the distance. Hence, *estimate* $\geq$ *distance*. Uses Edge List.

Special Case: If graph is a weighted **tree**, perform DFS or BFS and relax each edge the first time you see it → $O(V)$ time as there are only $O(V)$ edges in a tree.

To detect negative weight cycles, run this algorithm for $|V| + 1$ iterations. If an estimate changes in the last iteration, such a cycle exists in the graph.

Maximizing x = Maximizing $\log x$ = Minimizing $-\log x$.

Triangle Inequality: $\delta(j) \leq \delta(i) + w$



Every node in a **tree** has a unique path to every other node. The **Lowest Common Ancestor** of two nodes u and v is the deepest node in the tree with both u and v as its descendants.

Tips: You can modify the "relax" operation to suit the needs of the question.

Dijkstra's Algorithm

```
Dijkstra
  Graph G;
  PriorityQueue pq = new PQQueue();
  double[] distTo;
  searchPath(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]);
```

Basic Idea: Maintain the distance estimate for every node. Begin from an empty shortest-path tree. Repeat: Find unfinished vertex with the minimum estimate, add vertex to the shortest-path-tree, and relax all outgoing edges of that tree. Mark vertex as finished.

$O((V+E)\log V)$ using an AVL tree Priority Queue
Each node is added to the priority queue once → insert / deleteMin called $|V|$ times. Each edge is relaxed once → relax / decreaseKey called $|E|$ times. Priority Queue operations take $O(\log V)$. Total time complexity will hence be $O((V+E)\log V) = O(E \log V)$

You can stop once you dequeue the destination, as a vertex is "finished" once it is dequeued.

Key Property: If P is the shortest path from S to D, and if P goes through X, then P is also the shortest path from S to X (and from X to D)

Fails if graph has negative weight edges. **Requirement for correctness:** Extending a path with an edge should only *increase* the path's "badness".

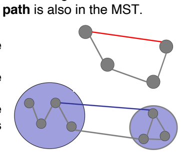
Longest Paths: For a Directed Acyclic Graph, you can negate the edges. Shortest path in negated = longest path in regular. Does not apply for general cyclic graphs.

Minimum Spanning Trees

A spanning tree is an acyclic subset of the edges that connects all nodes. A **minimum spanning tree** is a spanning tree with minimum weight.

Properties: Every edge in the graph's **minimax path** is also in the MST.

1. No cycles
2. If you cut an MST, the two pieces are MSTs
3. **Cycle Property:** For every cycle, the maximum weight edge is not in the MST.
4. **Cut Property:** For every partition of the nodes, the minimum weight edge across the cut is in the MST.
5. Contains $|V| - 1$ edges



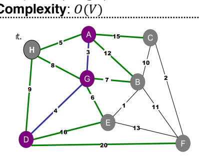
If all the edges in the graph have the **same weight**, finding its MST simply requires a DFS / BFS → $O(E)$. Otherwise, for a directed acyclic graph (DAG) with only one root, for every node except the root, just add the minimum weight incoming edge → $O(E)$.

Prim's Algorithm

```
PriorityQueue pq = new PriorityQueue();
for (Node v : G.V())
  pq.insert(v, INFTY);
pq.decreaseKey(start, 0);

HashSet<Node> S = new HashSet<Node>();
S.put(start);

HashMap<Node,Node> parent = new
HashMap<Node,Node>();
parent.put(start, null);

while (!pq.isEmpty())
  Node v = pq.deleteMin();
  S.put(v);
  for each (Edge e : v.edgeList())
    Node w = e.otherNode(v);
    if (!S.get(w))
      pq.decreaseKey(w, e.getWeight());
      if (weight_decreased)
        parent.put(w, v);
```

Basic Idea: Let S be the set of nodes connected by blue edges. Initially, S = {A}. Then repeat: Identify cut {S, V-S}, find minimum weight edge on cut, add new node to S.

$O(E \log V)$ using a binary heap Priority Queue. Each vertex added / removed once from priority queue → $O(V \log V)$. Each edge => one decreaseKey → $O(E \log V)$
Space Complexity: $O(V)$



Variant: If all edges have weights from $\{1..10\}$, use an array (with size 10) as a PriorityQueue, where slot A[j] holds a linked list of nodes of weight j. Inserting / removing nodes from such a queue will be $O(V)$, and a decreaseKey operation just involves looking up the node from a HashTable and moving it to the correct linked list and is hence $O(E)$. Total cost → $O(V+E) = O(E)$

Kruskal's Algorithm

```
Edge[] sortedEdges = sort(G.E());
ArrayList<Edge> mstEdges = new
ArrayList<Edge>();
UnionFind uf = new UnionFind(G.V());

for (int i=0; i<sortedEdges.length; i++)
  Edge e = sortedEdges[i];
  Node v = e.one();
  Node w = e.two();
  if (!uf.find(v,w))
    mstEdges.add(e);
    uf.union(v,w);
```

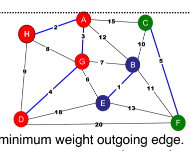
Basic Idea: Sort edges by weight. Then consider edges in ascending order: If both endpoints are in the same blue tree, then color the edge red. Otherwise, color the edge blue.

$O(E \log V)$. Sorting takes $O(E \log V)$, and for E edges, find takes $O(\log V)$, and union takes $O(\log V)$.
Space Complexity: $O(V)$ for union-find with path compression.

Variant: If all edges have weights from $\{1..10\}$, use an array (with size 10) where slot A[j] holds a linked list of edges of weight j. Putting the edges into this array and iterating over them will be $O(E)$ and checking whether to add an edge and union operations are $O(\alpha)$. Total cost $O(\alpha E)$.

Boruvka's Algorithm

Initially: Create n connected components, one for each node in the graph. For each node, store a component identifier → $O(V)$



One 'Boruvka' Step → $O(V+E)$
1. For each connected component, search for the minimum weight outgoing edge.
   - Use DFS or BFS to check if edge connects two components, and remember minimum cost edge connected to each component → $O(V+E)$
2. Add selected edges.

3. Merge selected components.
   - Scan every node: Compute new component IDs, update component IDs, mark added edges → $O(V)$

$O(E \log V)$. We initially have $|V|$ components. In each 'Boruvka' step, assuming k components initially, at least $k/2$ edges are added, at least $k/2$ components merge, and at least $k/2$ components remain. Algorithm terminates when only 1 components is left, so there are at most $O(\log V)$ 'Boruvka' steps.

Comparison of Algorithms

Maintain a set of explored vertices. Add vertices to explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

Algorithm	Data Structure	Takes edge from vertex
BFS	Queue	Discovered least recently
DFS	Stack	Discovered most recently
Djikstra	Priority Queue (ordered by distance)	Closest to source
Prim	Priority Queue (ordered by edge weight)	Connected via lightest edge

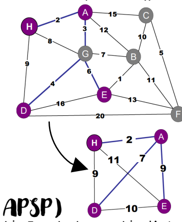
Maximum Spanning Trees

Multiply each edge weight in the graph by -1, and run the MST algorithm. The MST that is the "most negative" is the maximum spanning tree.

Steiner Tree Problem

To find the MST for a **subset of the vertices** in a graph, let the nodes in the subset be "required nodes" and the rest of the nodes be "Steiner nodes". The following SteinerMST algorithm guarantees that its output spanning tree will have a weight that is less than 2 × the weight of the Optimal MST. Its best approximation is 1.55 × the optimal weight.

1. For every required vertex (v, w), calculate the shortest path from v to w.
2. Construct new graph on required nodes.
3. Run MST algorithm on the new graph.
4. Map new edges back to the original graph.



Floyd-Warshall (APSP)

Given an input adjacency matrix E, outputs a matrix dist where dist[v,w] gives the shortest distance from v to w, for all pairs of vertices (v, w).

Basic Idea: Let $S[v, w, P_k]$ be the shortest path from v to w that only uses intermediate nodes from the set $P_n = \{1, 2, \ldots, n\}$. If we known $S[v, w, P_k]$, then $S[v, w, P_{k+1}] = min(S[v, w, P_k], S[v, k+1, P_k] + S[k+1, w, P_k])$.

$O(V^3)$

```
APSP(E):
  int[][] S = new int[V.length][V.length];
  for (int v=0; v<V.length; v++)
    for (int w=0; w<V.length; w++)
      S[v][w] = E[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;
```

In comparison, APSP via exhaustive Djikstra's is $O(V(V+E)\log V)$ which is preferred for tree-like graphs.

Heaps

A binary heap or MaxHeap implements a Max Priority Queue.

1. priority[parent] >= priority[child]
2. **Complete binary tree:** Every level is full except possibly the last, and all nodes are as far left as possible.
3. **Maximum height for n elements:** $floor(\log n)$

To implement, map each node in complete binary tree into a slot in an array, where left(x)=2x+1, right(x)=2x+2, parent(x)=floor((x-1)/2).

Insert, extractMax, increaseKey, decreaseKey, delete → all $O(\log n)$
Successor queries → $O(n)$.

```
bubbleUp(Node v)
  while (v != null)
    if (priority(v)
      > priority(parent(v)))
      swap(v, parent(v));
    else return;
    v = parent(v);
bubbleDown(Node v)
  while (!leaf(v))
    leftP = priority(left(v));
    rightP = priority(right(v));
    maxP = max(leftP, rightP,
               priority(v));
    if (leftP == max)
      swap(v, left(v));
      v = left(v);
    else if (rightP == max)
      swap(v, right(v));
      v = right(v);
    else return;
```

insert(25): Add a new leaf with priority v, and then bubble up.
decreaseKey(28 → 4): Update the priority and then bubbleDown(4).
delete(5): swap(5, last), remove(last), bubbleDown the node that you swapped.
extractMax(): Delete and return the root.

```
HeapSort:
// A is a heap
for (i=(n-1) to 0):
  A[i] = extractMax(A);
```

Heapify: $O(n)$

```
// A is unsorted array
for (i=(n-1) to 0)
  bubbleDown(i, A);
```

HeapSort is in-place, faster than MergeSort but a little slower than QuickSort, unstable, and always $O(n \log n)$

Union-Find

In a Union-Find Disjoint Set of N objects, `boolean find(Key p, Key q)` answers if p and q are in the same set, and `void union(Key p, Key q)` replaces sets containing p and q with their union.

	Find	Union
quick-find	$O(1)$	$O(n)$
quick-union	$O(n)$	$O(n)$
weighted-union	$O(\log n)$	$O(\log n)$
path compression	$O(\log n)$	$O(\log n)$
weighted-union (with path-compression)	$\alpha(m, n)$	$\alpha(m, n)$

In **QuickFind**, we have an `int[] componentID`, and two objects (mapped to integers via Hashing with Open Addressing) are connected if they have the same component identifier. In **QuickUnion**, we have an `int[] parent`, and two objects are connected if they are part of the same tree. It is slow because the trees are linear and unbalanced.

```
union(int p, int q)
  while (parent[p] !=p) p = parent[p];
  while (parent[q] !=q) q = parent[q];
  if (size[p] > size[q]
    parent[q] = p;
    size[p] = size[p] + size[q];
  else
    parent[p] = q;
    size[q] = size[p] + size[q];
```

Weighted Union: Heavier tree will be made the root. Resultant trees will have a maximum depth of $O(\log n)$.

```
findRoot(int p)
  root = p;
  while (parent[root] != root)
    parent[root]
      = parent[parent[root]];
    root = parent[root];
  return root;
```

← Alternatively…

Path Compression: After finding the root, set the parent of each traversed node to the root.

For weighted union with path compression, starting with empty, any sequence of m union/find operations on n objects takes $O(n + m\alpha(m, n))$ time.

Optimization

Newton's Method

```
findRoot(float x, float error)
  diff = error + 1;
  while (diff > error)
    float f = function(x);
    float df = derivative(x);
    float newX = x - (f/df);
    diff = abs(x - newX);
    x = newX;
  return x;
```

Finds a root of $f(x) = 0$. After every iteration i, we update our value of root x to be $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$.

If quadratic convergence (ie. at every iteration, no. of correct digits doubles) → for d digits of accuracy, require $O(\log d)$ iterations, but has more expensive computation.

Conditions for Quadratic Convergence: f must be continuously differentiable, have a non-zero derivative at root, and have a second derivative at root.

Gradient Descent

```
findMin(float x, float step, float error)
  diff = error + 1;
  while (diff > error)
    float df = derivative(x);
    float newX = x – (step * df);
    diff = |x – newX|;
    x = newX;
  return x
```

Finds the minimum value of f(x) via successive approximation.

Has more iterations than Newton's method because it has linear convergence, but it has faster computation.

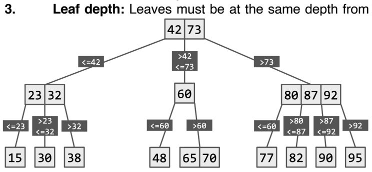
Conditions for Convergence: Algorithm converges if f is convex and differentiable, and its gradient is L-Lipschitz, and if step size $\gamma \le 1/L$.

Recitations

(a,b)-trees

In an (a,b)-tree, where $2 \le a \le \frac{b+1}{2}$,

1. **(a,b)-child policy:** The root node and internal nodes have $[2, b]$ and $[a, b]$ number of children respectively. Leaf nodes have $[a-1, b-1]$ number of keys.
2. **Key ordering:** Internal nodes have one more child than its number of keys.
3. **Leaf depth:** Leaves must be at the same depth from



Height: Maximum $O(\log_a n) + 1$, minimum $O(\log_b n)$

Search Cost: $O(\log n)$ because the tree has $O(\log_a n)$ height and binary search for a key at every node takes $O(\log_2 b)$ time.

B-trees are a subcategory of (a,b)-trees where $a = B$ and $b = 2B$.

2Sum Problem

Given a sorted array of numbers, find all valid pairs of numbers (whose sum would fall within a certain range).

$1	$8	$15	$18	$20	$23	$25	$30

$8 + $25 = 33 < 35

Converging Pointers Solution:
Use a low pointer and a high pointer. If sum is too small, advance low pointer. If sum is too large, retreat high pointer. When a valid pair is found, increment low pointer and decrement high pointer, and continue on to the next iteration.

3Sum Problem: $O(\log n)$ space, $O(n^2)$ time
Sort the array. Go through each item a_i in the array, and check if we can find a pair using $2SUM(x - a_i)$ in the subarray after a_i using converging pointers.

$N - SUM(x, arr) = (N-1) - SUM(x - a_i, arr \backslash a_i)$ for all a_i in the array arr.

Identifying Deleted Photos

Alice initially had n photos, and all were backed up remotely as photos $r_1, \cdots, r_n$. Alice's local computer had a virus which deleted some photos, so it is left with photos $l_1, \cdots, l_m$. Which are the deleted photos?

Let k be some integer (which may depend on n and m)
Repeat:
  Randomly pick a hash function *h* that maps a photograph to an integer in $[1, k]$
  For each photo l_i on Alice's local computer, compute its hash value $h(l_i)$
  Save all locally hashed values to a file $H_l = \{h(l_i): i \in [1, m]\}$ on Alice's local computer
  For each photo r_i on the remote server, compute its hash value $h(r_i)$
  Save all remotely hashed values to a file $H_r = \{h(r_i): i \in [1, n]\}$ on the remote server
  Download H_r to Alice's local computer
    If $|H_r| - |H_l| = n - m$,
      Download the photos r_i whose hash value $h(r_i)$ is in H_r but not in H_l
      Terminate the repeat loop
    Else, continue the loop to look for a better hash function

When a remote signature is found locally, it could be a **false positive** due to collisions.
When a remote signature is **not found** locally, it is always a **true negative**.

Longest Common Substring (LCS)

```
hasCommonSubstring(String A, String B, int L)
Initialize hash table H which uses hash function h₁
for each L-length substring a in A:
  address ← h₁(a)
  fingerprint ← h₂(a)
  bucket ← H[address]
  Insert (fingerprint, a) into bucket
for each L-length substring b in B:
  address' ← h₁(b)
  fingerprint' ← h₂(b)
  bucket' ← H[address']
  for each (fingerprint, a) in bucket':
    if (fingerprint == fingerprint'):
      if (a == b) return true
return false
```

h_1 is the main hash function mapping a string to a bucket in the range $[1, n]$

h_2 is the fingerprint hash function mapping a string to an integer in the range $[1, n^2]$

$O(n^2)$

Rolling Hash Functions

A **rolling hash function** is an algorithm that avoids having to re-hash an entire substring at every step of the way. Instead, it exploits the *incremental* changes in sequential substrings. Whenever a string is updated, it computes the new hash in an efficient and incremental manner. When a hash is requested, it just returns the pre-computed hash.

`Initialize(S)`	Take a string *S* of length *L*.	$O(L)$
`DeleteFirst`	Removes first letter from the string.	$O(1)$
`AddLast(c)`	Adds character *c* to the end of the string.	$O(1)$
`Hash`	Returns the hash of the current string.	$O(1)$