Sorting

Binary Search	Bubble Sort	Insertion Sort	Selection Sort	Heap Sort
$A[begin] \le key \le A[end]$	At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array.	At the end of iteration j, the first j items in the array are in sorted order.	At the end of iteration j, the smallest j items are correctly sorted in the first j positions of the array.	At the start of the jth iteration, the jth largest node is at the root of the max heap

Name	Best	Ave	Worst	Space	Stable
Bubble	n	n^2	n^2	1	Yes
Selection	n^2	n^2	n^2	1	No
Insertion	n	n^2	n^2	1	Yes
Merge	$n\log(n)$	$n\log(n)$	$n\log(n)$	n	Yes
Quick	$n\log(n)$	$n\log(n)$	n^2	1	No
Heap	$n\log(n)$	$n\log(n)$	$n\log(n)$	1	No

Order Statistic (QuickSelect)

Key Idea. Partition the array but only recurse into the correct half.
Time Complexity: $O(n)$

Binary Tree Traversals

In-order	Pre-order	Post-order
Left, self, right	Self, left, right	Left, right, self
Dot on the bottom	Dot on the left	Dot on the right

Binary Tree Algorithms

```
successor()
  if (rightTree != null)
    return right.Tree.searchMin()
  TreeNode parent = parentTree
  TreeNode child = this
  while ((parent != null) && (child == parent.rightTree))
    child = parent
    parent = child.parentTree
  return parent
```

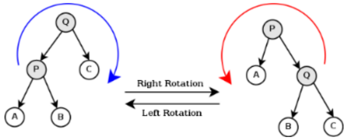
```
delete(v)
  if (v has no children)
    remove v
  else if (v has one child)
    remove v
    connect child(v) to parent(v)
  else if (v has two children)
    x = successor(v)
    delete(x)
    remove v
    connect x to left(v), right(v), parent(v)
```

AVL Trees

Height invariant: $|v.left.height - v.right.height| \le 1$

Height of n nodes: $h < 1.44\log(n) < 2\log(n)$ *or* $n > 2^{\frac{h}{2}}$

Tree Rotations:



Rotations during insert: Maximum 2
Rotations during delete: Maximum $\log(n)$

Augmented Search Trees

1. Order Statistic Tree

Augmentation: Size of the subtree in every node

Explanation: By knowing the size of the left subtree, we know the rank of the current node.

Maintenance: Weight is calculated locally taking $O(1)$ time

2. Interval Trees

Augmentation: Tree is sorted by the left endpoint. Each node stores an interval as well as the maximum right endpoint inside the subtree.

Explanation: By knowing the maximum right endpoint in the subtree, we can decide where to go left or right during the search

Maintenance: Maximum right endpoint is calculated locally taking $O(1)$ time

3. Orthogonal Range Searching

Augmentation: Points are only in the leaves. Internal nodes store the max of any leaf in the left subtree.

Explanation: By knowing the max in the left subtree, we know when to stop searching left/right.

Query Time: $O(k + log(n))$

(a,b)-Trees

(a, b)-Tree Rules

1. (a, b)-child Policy.

Node Type	#Keys		#Children	
	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

2. Key-ordering. A non-leaf node must have one more child than its number of keys.

3. Leaf Depth. All leaf nodes must all be at the same depth from the root.

(a, b)-Tree Max Height. $O(\log_a n) + 1$

(a, b)-Tree Min Height. $O(\log_b n)$

(a, b)-Tree Search. $O(\log b \times \log n) = O(\log n)$

(a, b)-Tree Insertion. $O(\log n + b\log n) = O(b\log n) = O(\log n)$

Hashing

Load. $\alpha = \frac{\text{number of items } (n)}{\text{number of buckets } (m)}$

Collisions. Collisions are unavoidable. Two distinct keys k_1 and k_2 collide if:

$$h(k_1) = h(k_2)$$

Chaining

Key Idea	Each bucket contains a linked list of items
Hash Function	A mapping from the universe of U to a small set of size m. $h: U \to \{1..m\}$
Simple Uniform Hashing Assumption	Every key is equally likely to map to every bucket. Keys are mapped independently
Space Complexity	$O(m+n)$
Insert	Worst-case: $O(1)$ Expected: $O(1)$
Search	Worst-case: $O(1+n)$ Expected: $O(1+\frac{n}{m})$
Delete	Worst-case: $O(1+n)$ Expected: $O(1+\frac{n}{m})$
(m == n)	We can still add new items to the hash table. Still able to search efficiently ($\approx O(1)$)

Open Addressing

Key Idea	If the bucket is full, try again in another bucket.
Double Hashing	Let f(k), g(k) be two ordinary hash functions. $h(k, i) = f(k) + i \cdot g(k) \bmod m$ If g(k) is relatively prime to m, then h(k, i) hits all buckets.
Uniform Hashing Assumption	Every key is equally likely to be mapped to every permutation, independent of every other key.
Space Complexity	$O(m)$
Insert	Worst-case: $O(n)$ Expected: $O\left(\frac{1}{1-\alpha}\right)$
Search: Algorithm	Worst-case: $O(n)$ Expected: $O\left(\frac{1}{1-\alpha}\right)$
Delete	Worst-case: $O(n)$ Expected: $O\left(\frac{1}{1-\alpha}\right)$
(m == n)	Table is full and we cannot insert any more items. Can no longer search efficiently. ($O(n)$)

Table Resizing

Resizing.

Cost of resize: $O(m_1 + m_2 + n)$

m_1: Size of old table

m_2: Size of new table

n: Number of items in the table

Rate of Growing.

Rate of resize	Cost of resize	Cost of inserting n items
$m_2 = m_1 + 1$	$O(n)$	$O(n^2)$
$m_2 = 2m_1$	$O(n)$	$O(n)$
$m_2 = m_1^2$	$O(n^2)$	$O(n^2)$

Rate of Shrinking. Optimally when $n = \frac{m}{4}$

Amortized Analysis

An operation has amortized cost $T(n)$ if for evert integer k, the cost of k operations is $\le k\,T(n)$

Hash Table Resizing Performance

Insert: Amortized $O(1)$

Search: Expected $O(1)$

For both chaining and open addressing since we can control the load factor.

Fingerprint Hashtable

Key idea: Do not store the key in the table. Just the boolean of whether the item is inside.

No False Negative: If the value is inside the table, it will always report true.

False Positives Possible: If the value is not inside the table, it will *sometimes* return true.

	Value inside the table	Value not inside table
Lookup returns true	Always	Sometimes
Lookup returns false	Never	Sometimes

Probability of a false positive:

$$1 - \left(\frac{1}{e}\right)^{\frac{n}{m}}$$

If you want probability of false positives to be less than p:

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

Bloom Filter

Key idea: Use k hash functions

Trade-off: Each item takes more space in the table, but the probability of a false positive is smaller.

Probability of false positive:

$$\left(1 - e^{-kn/m}\right)^k$$

If you want probability of false positives to be less than p:

$$\frac{n}{m} \le \frac{1}{k}\ln\left(\frac{1}{1-p^{\frac{1}{k}}}\right)$$

Optimal k:

$$k = \frac{m}{n}\ln(2)$$

Error Probability with Optimal k:

$$2^{-k}$$

Undirected Graphs

Connected. Every pair of nodes is connected by a path.

Connected Components. Connected subgraph which is connected to no additional vertices in the rest of the graph.

Degree of Node. The number of adjacent edges

Degree of Graph. The maximum degree of the graph is the maximum of its vertices' degree.

Diameter of a Graph. Maximum number of *edges* between two nodes, following the shortest path.

Special Graphs. Star, clique, line, cycle, and bipartite graph.

Representing Graphs

Adjacency list: Nodes are stored in an array. Edges are stored in a linked list per node.

Adjacency Matrix. $n \times n$ matrix where $a_{ij} = 1\ iff$ there is an edge from node i to node j

Adjacency List vs Adjacency Matrix

	Adjacency List	Adjacency Matrix
Space	$O(V+E)$ Suitable for sparse and normal graphs	$O(V^2)$ Suitable for dense graphs where $\lvert E\rvert = \Theta(V^2)$
Query	Good at enumerating neighbours	Good at determining relationship between two nodes, and determining k length paths

BFS vs DFS

	BFS	DFS
Time (AL)	$O(V+E)$	$O(V+E)$
Time (AM)	$O(V^2)$	$O(V^2)$
Key Idea	Explore level-by-level, never going backwards	Follow path until stuck, backtrack until you find a new edge, recursively explore it
Parent tree	Shortest path graph of the source node if the graph is unweight or have identical weights	Does not contain shortest paths
Underlying DS	Queue	Stack
Visits	Visits every vertex Visits every edge *Does not* visit every path	

Directed Graphs

Similar to directed graphs except each edge is now directed

In-degree: Number of incoming edges

Out-degree: Number of outgoing edges

SSSP

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

Shortest Subpath. Subpaths of shortest paths are shortest paths as well

Negative Weight Cycles. A connected graph with negative weight cycles do not have a shortest path.

Key ideas.

(i) Relax edges in the correct order.

(ii) Maintain an estimate for each node where estimate ≥ distance

Bellman-Ford

Algorithm:

```
n = V.length;
for (i = 0; i < n; i++)
  for (Edge e : graph)
    relax(e);
```

Invariant: After i iterations of the outer loop, for all paths from the source to u with i or fewer edges, the length of that path is no less than dist[u].

Early Termination: When an entire sequence of $|E|$ relax operations have no effect.

Negative Weights: Bellman-Ford works with negative weights as long as there is not negative weight cycles

Time (AL): $O(EV)$ **Time (AM):** $O(V^3)$

Detecting Negative Weight Cycle. If an estimate changes in after $|V|$ iterations, then the graph has a negative weight cycle.

Dijkstra

Basic Idea:

Maintain distance estimate for every node.

Begin with empty shortest-path-tree.

Repeat:

- Consider vertex with minimum estimate.
- Add vertex to shortest-path-tree.
- Relax all outgoing edges.

Data Structure: Use a AVL tree/binary heap as a priority queue to store the vertices.

Dijkstra's Algorithm:

```
searchPath(start)
  pq.insert(start, 0)
  distTo = new double[G.size()]
  distTo.fill(INFTY)
  distTo[start]=0
  while !pq.empty()
    int w = pq.deleteMin()
    for (Edge e : G[w].nbrList)
      relax(e)
relax(u, v)
  if (distTo[v] > distTo[u] + weight(u, v))
    distTo[v] = distTo[u] + weight(u, v)
```

Time Complexity:
V times of insert/deleteMin
E times of relax/decreaseKey
Time (AL): $O\big((V+E)\log(V)\big) = O(E\log(V))$
Time(AM): $O\big(V^2 + V\log(V) + E\log(V)\big) = O(V^2)$
Negative Weights: Will not work with negative weights
Early Termination: We can stop as soon as we dequeue the destination.
Functions on weights. Only strictly increasing or decreasing function can be used on the weights, given that they do not make the weights negative. (E.g. Multiplying the weights by positive constant c.)

Topological Order

Properties
1. Sequential total ordering of all nodes
2. Edges only point forward

Conditions. Only Directed Acyclic Graphs (DAG) have topological order.
Finding Topological Order: Algorithm.
Use a Post-Order *DFS*

```
for (start = i; start < |V|; start++)
  if (!visited[start])
    visited[start] = true;
    DFS(start)
    schedule.prepend(v)
```

Use Kahn's Algorithm

```
Repeat:
  S = nodes in G that have no incoming edge
  Add nodes in S to topo-order
  Remove all edges adjacent to nodes in S
  Remove nodes in S from the graph
```

We maintain an in-degree array as well as a queue of nodes with in-degree of 0. Dequeue and decrement the in-degree of neighbours. Add node to queue if their in-degree becomes 0.
Non-unique.
Time (AL): $O(V+E)$ **Time (AM):** $O(V^2)$

Topological Sort

Key Idea: Relax edges in topological order
Time (AL): $O(E)$ **Time (AM):** $O(V^2)$
Longest Path: Negate the edges or modify the relax function
Does not work on cyclic graphs.

Shortest Path on a Tree

Key Idea: Only one path, hence just use DFS/BFS starting from the source
Time (AL): $O(V)$ **Time (AM):** $O(V^2)$

SSSP Summary

Graph Type	Algorithm	Time
No negative weight cycles	Bellman-Ford	$O(VE)$
No negative edges	Dijkstra	$O(E\log(V))$
No directed cycles	TopoSort + Relax	$O(E)$
No cycles	DFS + Relax	$O(V)$

Heaps

Properties
1. **Heap ordering**

$$priority[parent] \geq priority[child]$$

2. **Complete binary tree.** Every level is full except possibly the last. All nodes are as far left as possible.

Maximum height of heap with n elements.
$floor(\log n)$
Time Complexity: $O(\log n)$ for all operations.
Storing the heap in an array. Fast.
Unsorted list to heap: $O(n)$
Algorithms:

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

Disjoint Set (Union Find)

Weighted Union. Only merge the smaller tree with the larger tree.
Height with Weight Union. $O(\log n)$ since when height increase, the number of nodes at least double.
Path Compression. Set the parent of each traversed node to the root.
Algorithms:

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

Time Complexity.

	With Weighted Union	Without Weighted Union
With Path Compression	n operations cost $O(n + m\alpha(m,n))$	$\Theta(n)$
Without Path Compression	Find: $O(\log n)$ Union: $O(\log n)$	Depends on implementation

Minimum Spanning Tree (MST)

Spanning Tree: A acyclic subset of the edges that connects all nodes.
MST. A spanning tree with minimum weight
Properties.
1. No cycles
2. If you cut an MST, the two pieces are both MSTs
3. **Cycle Property.** For every cycle, the max weight edge in not in the MST
4. **Cut Property.** For every partition of nodes, the minimum weight edge across the cut is in the MST.

Generic MST Algorithm
Red rule: If C is a cycle with no red arcs, then colour the max-weight edge in C red
Blue rule: If D is a cut with no blue arcs, then colour the min-weight edge in D blue.

```
Repeat:
  Apply red rule or blue rule to an
  arbitrary edge until no more edges
  can be coloured.
```

MST True or False

1. True. If we add one edge e to G and e is heavier than any edge in T, then T is still an MST of the new graph.
2. True. If we add one edge e to G and e is lighter than any edge in T, then the MST of the new graph can be constructed by removing exactly one edge from T and adding e to T.
3. True. If we increase the weight of an edge e in G and e is not in the MST, then T is still an MST of G.
4. True. If e is an edge in G that is not in T, then there is always a cycle in G where e is the heaviest edge on the cycle.
5. False. If we add one edge $e = (u, v)$ to G and e is heavier than any edge adjacent to u or v, then T is still an MST of the new graph
6. False. If $e = (u, v)$ is an edge in T, then there is a cycle in G where e is the lightest edge on the cycle
7. False. If $e = (u, v)$ is the heaviest edge in G then it is never in the MST

Prim's Algorithm

Basic Idea.
S: set of nodes connected by blue edges
Initially, $S = \{A\}$
Repeat:
- Identify cut: $\{S,\ V - S\}$
- Find min-weight across cut
- Add new node to S

Data structure: Use priority queue to find the lightest edge on a cut
Algorithm.

```
while (!pq.empty())
  Node v = pq.deleteMin()
  S.put(v)
  for (Edge e : v.edgeList())
    w = e.otherNode(v)
    if (!S.contain(w))
      pq.decreaseKey(w, e.getWeight())
      parent.put(w, v)
```

Time Complexity.
Each vertex is added and removed from the PQ once. $O(V\log V)$
Each edge causes one decrease key $O(E\log V)$
Time: $O(V\log V + E\log V) = O(E\log V)$

Kruskal's Algorithm

Basic Idea.
Sort edges by weight from smallest to biggest
Consider edges in ascending order
- If both endpoints are in the same blue tree, then colour red
- Otherwise, colour edge blue

Data structure: Use union-find to determine if two nodes are in the same blue tree.

Algorithm.

```
Edge[] sortedEdges = sort(G.E())
for (int i = 0; i < |E|; 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)
```

Time Complexity.
Sorting edges: $O(E\log E)$
Each edge causes 2 union find operation
Time: $O(E\log E + E \times \alpha(n)) = E\log E = E\log V$

Boruvka's Algorithm

Basic Idea.
Add all minimum adjacent edges. Repeat.
Advantages
Has good parallelism
Algorithm.
Initially:
- Create n connected components, one for each node in the graph

One "Boruvka" Step: $O(V+E)$
- For each connected component, search for the minimum weight outgoing edge using BFS/DFS $O(V+E)$
- Add selected edge
- Merge connected components by change components IDs O(V)

Time Complexity
$\log V$ Boruvka steps
Each $O(V+E)$
Time: $O((E+V)\log V) = O(E\log V)$

MST Variants

1. All edges same weights - $O(E)$ with BFS/DFS
2. Weights are $\{1..10\}$ - $O(E)$ with counting sort or modified PQ
3. DAG with only one root – $O(E)$ by adding minimum weight incoming edge per node
4. Max Spanning Tree – negate edges or Kruskal's/Prim's in reverse

Steiner Tree Algorithm (SteinerMST)

Goal. MST of a subset of vertices
Algorithm.
1. For every require vertex $(v,\ w)$, calculate the shortest path from $(v\ to\ w)$
2. Construct new graph on required nodes
3. Run MST on new graph
4. Map new edges back to original graph

Guarantees
Output of SteinerMST $<\ 2 *$ Optimal Solution

Dynamic Programming

Optimal sub-structure. Optimal solution can be constructed from optimal solutions to smaller sub-problems
Overlapping sub-problems. The same smaller problem is used to solve multiple bigger problems.

Longest Increasing Subsequence

Sub-problem.
$S[i] = LIS(A[1..n])$ starting from the back
Time complexity
n subproblems
Each subproblem takes $O(i)$ time
Time: $O(n^2)$

Lazy Prize Collecting

Sub-problem.
$P[v,\ k]$ = maximum prize you can collect starting at v and taking exactly k steps

$$P[v,\ k] = MAX\begin{Bmatrix} P[w_1, k-1] + w(v, w_1), \\ P[w_2, k-1] + w(v, w_2), \\ \vdots \end{Bmatrix}$$

where v.nbrList() = $\{w_1,\ w_2,\ w_3,\ \cdots\}$
Time complexity
k rows
Each row takes $O(E)$ time
Time: $O(kE)$

Vertex Cover on a Tree

Sub-problem.
$S[v,\ 0]$ = size of vertex cover in subtree if v *is not* covered
$S[v,\ 1]$ = size of vertex cover in subtree if v *is* covered

$$S[v,\ 0] = S[w_1, 1] + S[w_2, 1] + \cdots$$

$$S[v, 1] = 1 + \sum_{i=0}^{|nbrList|} \min\{S[w_{i,}, 0],\ S[w_i,\ 1]\}$$

where v.nbrList() = $\{w_1,\ w_2,\ w_3,\ \cdots\}$
Time complexity
Each edge explored once
Time: $O(V)$

All Pairs Shortest Path (Floyd-Warshall)

Sub-problem.
$S[v,\ w,\ P]$ = shortest path from v to w that only uses intermediate nodes in the set P.

$$S[v,\ w, P_8] = \min\begin{Bmatrix} S[v,\ w, P_7], \\ S[v,\ 8, P_7] + S[8,\ w, P_7] \end{Bmatrix}$$

where v.nbrList() = $\{w_1,\ w_2,\ w_3,\ \cdots\}$
Algorithm

```
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])
```

Time complexity
3 for loops
Time (AL & AM): $O(V^3)$
Augmentation.
This algorithm can be used to solve more problems as long as we change the 2 functions used in the last line
1. Matrix Multiplication

```
S[v][w] = plus(S[v][w], S[v][k]*S[k][w])
```

2. Transitive Closure

```
S[v][w] = OR(S[v][w], S[v][k] AND S[k][w])
```

3. Minimax

```
S[v][w] = MIN(S[v][w], MAX(S[v][k],S[k][w]))
```

Randomized vs Average-case

In a randomized algorithm, the algorithm makes random choices. Hence, for every input, there is a good probability of success. In average-case analysis, the environment chooses the random input.

Graph Augmentation Techniques.

Questions
What do the vertices represent?
What do the edges represent?
Are the edges directed?
Is the graph weighted?
What kind of graph representation will you use?
Common Techniques
1. Graph duplication to capture problem states
2. Graph duplication to force travel through certain edges
3. Dummy source node to capture different initial states
4. Reversing the edges and running SSSP from the destination instead
5. Applying a function on the edges (monotonic for SSSP, any for MST)
6. Modifying the relax function
7. For problems with limited "energy", you can just run Dijkstra and check if $distTo[v] <\ maxEnergy$

Recurrance Relation Helper

Given a recurrence relation that looks like

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^k)$$

a	b	k	i	Solution
1	2	0	0	$\log n$
x	x	0	0	n
x	x	1	0	$n\,log\,n$
2	4	0	0	$\sqrt{2}$

Useful Summations.

$$\sum_{i=1}^{n} i^c = O(n^{c+1})$$

$$\sum_{i=1}^{n} \frac{1}{i} = \mathrm{O}(\log n)$$

$$\sum_{i=1}^{n} c^i = O(c^n)$$

$$\sum_{i=1}^{n} \log i^c = O(n\log n^c)$$