Cheatsheet

Assumes worst case unless otherwise state:

Data Structure	Access	Search	Insert	Delete
Array	1	n	n	n
Stack (LIFO)	n	n	1	1
Queue (FIFO)	n	n	1	1
Singly-Linked List	n	n	1	1
Doubly-Linked List	n	n	1	1
Hash Table	NA	n	n	n
Heap	n	n	$\log n$	$\log n$
BST (Worst)	n	n	n	n
AVL Tree	$\log n$	$\log n$	$\log n$	$\log n$
Red-Black Tree	$\log n$	$\log n$	$\log n$	$\log n$
B-Tree	$\log n$	$\log n$	$\log n$	$\log n$

Algorithm	In-Place	Stable	Best	Average	Worst	Space (Worst)
Quicksort	1	0	$n \log n$	$n \log n$	n^2	$\log n$ (call stack)
Mergesort	0	1	$n \log n$	$n \log n$	$n \log n$	n (auxiliary array)
Heapsort	1	0	$n \log n$	$n \log n$	$n \log n$	1
Bubble Sort	1	1	n	n^2	n^2	1
Insertion Sort	1	1	n	n^2	n^2	1
Selection Sort	1	0	n^2	n^2	n^2	1

Graph Problem	Algorithm	Time	Space
SSSP No negative cycles	Bellman Ford	VE	V
SSSP No negative weight edge	Dijkstra's	$(V+E)\log V$	V
SSSP Unweighted/Equal weight edge	BFS (**not** DFS)	$V+E$	V
SSSP Directed acyclic (DAG)	Topological Sort	$V+E$	V
SSSP Tree	BFS/DFS	V	V
Path with fewest edges	BFS	$V+E$	V
APSP	Floyd-Warshall	V^3	V^2
SCC	Kosaraju's	$E+V$	V
MST	Kruskal's	$E\log E$	$E+V$
MST	Prim's	$E\log V$	V

Master Theorem

$T(n) = aT(\frac{n}{b}) + cn^k$
$T(1) = c$

Condition	Upper Bound
$a < b^k$	$O(n^k)$
$a = b^k$	$O(n^k \log n)$
$a > b^k$	$O(n^{\log_b a})$

Binary Search

Can only be used on a sorted list
Sorting = $n\log n$
Search = $\log n$

```
# Given sorted array A, want to find index of value V
L = 0
R = A.length() - 1
while L <= R
    m = floor((L + R) / 2)
    if A[m] < V
        L = m + 1
    elif A[m] > V
        R = m - 1
    else
        return m
return notFound
```

Bubble Sort

Continually swap current element with element at index + 1 if it is larger.

```
for i = A.length() to 0
    for j = 0 to i - 1
        if A[j] > A[j + 1]
            swap(A[j], A[j + 1])
```

- **In-place**
- **Stable** (line 3, only swaps if `>`, not `>=`)

Best case: $O(n)$ (already sorted list, if no swaps occur at all, stop algorithm)

Insertion Sort

Assume first element is sorted. Iterate through the unsorted section, constantly **inserting** each element at the correct position in the sorted section.

```
# mark A[0] as sorted
for i = 1 to A.length()
    j = i
    while j > 0 && A[j] < A[j - 1]
        swap(A[j], A[j - 1])
        j--
```

- **In-place**
- **Stable**

Best case: $O(n)$ (already sorted list, no swaps needed)

Selection Sort

Continually **select** the smallest element from unsorted section and swap that element with the starting element of the unsorted section (thereby creating a sorted section element by element).

```
for i = 0 to A.length() - 1
    minIndex = i
    for j = i + 1 to A.length()
        if A[j] < A[minIndex]
            minIndex = j

    swap(A[i], A[minIndex])
```

- **In-place**
- **Non-stable**: for example, given `[2a, 2b, 1]`, `1` and `2a` will swap to become `[1, 2b, 2a]` after first iteration.

Best case: $O(n^2)$ (still have to find smallest element for the unsorted section)

Heapsort

Procedure:

1. Heapify the array: $O(n)$
2. Poll the heap for min/max key: $O(\log n)$
3. Repeat step 3 till heap is empty: n times

Time: $O(n \log n)$

- Useful to find k largest element in sorted order (where $k < n$). Time: $O(n + k \log n)$
- **In-place**
- **Non-stable**

Mergesort

```
function merge(int low, int mid, int high)
    for i = low to high
        buffer[i] = A[i]
    i = low
    j = mid + 1
    k = low
    while i <= middle and j <= high
        if buffer[i] <= buffer[j]
            A[k] = buffer[i]
            i++
        else
            A[k] = buffer[j]
            j++
        k++
    while i <= middle
        A[k] = buffer[i]
        k++
        i++
```

Time complexity for `merge` is $O(n)$. Line 7 `while` loop will constantly be checked in the worst case.

Time complexity for merge sort is $O(n \log(n))$

```
function mergeSort(A, low, high)
    if low < high
        mid = (high + low) / 2
        mergeSort(A, low, mid)
        mergeSort(A, mid + 1, high)
        merge(low, mid, high)
```

If `mergeSort` takes $T(n)$ time,
line 2 = c_1
line 3 = c_2
line 4 = $T(\frac{n}{2})$
line 5 = $T(\frac{n}{2})$
line 6 = $c_3 n$

$T(n) = 2T(\frac{n}{2}) + cn$

- **Not in-place** (requires auxiliary buffer array)
- **Stable**

Quicksort

```
function quicksort(A, low, high)
    if low < high
        p = partition(A, low, high)
        quicksort(A, low, p - 1)
        quicksort(A, p + 1, high)

# Lumoto's Algorithm
function partition(A, low, high)
    pivot = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < pivot
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m

# Hoare's Algorithm
function partition(A, low, high)
    pivot = A[low]
    i = low + 1  # Initialize left index
    j = high     # Initialize right index

    while i < j
        while (A[i] < pivot) and (i <= high)
            i++
        while (A[j] > pivot) and (j >= low)
            j--
        if (i < j)
            swap(A[i], A[j])

    swap(A[j], A[low])
    return i
```

- **In-place**
- **Non-stable**

Binary Heap

- Represented by a contiguous array where root starts at index 1. For given node with index x,
    - Left child has index $2x$
    - Right child has index $2x + 1$
- Sink/Swim: choose larger/smaller of child to swap corresponding to max/min heap to keep heap property
- Assuming Max Heap, parent key will always be $\geq$ 2 direct child key

Insertion

1. Add to node at the end (first available leaf)
2. Swim it up
3. requires $O(\log n)$ time

ExtractMax

1. Remove root
2. Exchange last element with root
3. Sink it down
4. requires $O(\log n)$ time

Deletion

1. Make node infinitely large
2. Swim it up to root
3. Run ExtractMax

Heapify

```
function create(A)
    for i from A.length/2 to 1
        sink(i)
```

Given an **unsorted** array A of elements, heapify requires
Time: $O(n)$
Space: $O(1)$

Binary Search Tree (BST)

Properties

1. Left subtree only contains nodes with keys strictly less than node's key
2. Right subtree only contains nodes with keys strictly more than node's key
3. No duplicate keys

For BST with n nodes, worst case height is $O(n)$ (BST with only a right subtree/flat linked list)

Successor

- Next key immediately larger
- Go right child once and go left all the way (furthest left child of the right child)

Predecessor

- Next key immediately smaller
- Go left child once and go right all the way (furthest right child of the left child)

Search

```
function search(Key k)
    if k < m_key
        # search left subtree
        return m_leftTree.search(k)
    else if k > m_key
        # search right subtree
        return m_rightTree.search(k)
    return this
```

Time complexity of $O(h)$ where h is the height of the BST

Insert

Same algorithm as search
Time complexity of $O(h)$ where h is the height of the BST

Delete

1. Node not found: **do nothing**
2. Node is at a leaf: **just remove**
3. Node is an internal node with 1 child: **remove node and connect child to parent**
4. Node is an internal node with 2 children: **swap node with its successor, and recursively call delete on that node**

Time complexity of $O(h)$ where h is the height of the BST

AVL Trees

Properties

1. Height balanced
   - Difference in height of left and right subtree differs by **at most 1**
   - For n nodes, height, $h < 2\log n$. Thus, $h = O(\log n)$
2. Balance Factor, $b(o) = h(o.left) - h(o.right)$
   - By convention, $h(\text{empty tree}) = -1$

Balancing Algorithm

1. After inserting a new node, n_{new}, recurse up from n_{new} and find the first node, n_0, with $|b(0)| > 1$
2. If n_0 is imbalanced **without** an "elbow" shape,
   1. If n_0 is **right heavy**, perform **left rotation**
   2. Else, perform **right rotation**

3. If n_0 is imbalanced **with** an "elbow" shape,
    1. if n_0 is right heavy, do **right rotation** on **right subtree of** n_0, then **left rotation** on n_0
    2. if n_0 is left heavy, do **left rotation** on **left subtree of** n_0, then **right rotation** on n_0

Walk up the tree to the root from inserted/deleted node and update balance factors

```
if tree is right heavy
    if tree’s right subtree is left heavy
        right rotate, left rotate
    else
        left rotate
else if tree is left heavy
    if tree’s left subtree is right heavy
        left rotate, right rotate
    else
        right rotate
```

- Insertion: requires $O(1)$ rotations
- Deletion: requires $O(\log n)$ rotations

Red-Black Trees

Properties

0. BST property is maintained for all nodes
1. The root and leaves are always black (leaves meaning the sentinel leaves)
2. Nodes are either red or black
3. Parent of a red node must be black
4. For every node, number of black node on the path from it to any of its leaves must remain the same (red nodes are ignored). This is called black height

Rebalancing

1. If uncle is red, then recolour parents and uncle to black
2. If node is left child and uncle is black, then right rotate on grandparent, and colour parent black and grandparent red
3. If node is right child and uncle is black, then left rotate on parent (to get the same tree as case 2) and perform step 2

B-Trees

The B-tree is a generalisation of a BST in that a node can have more than two children

Properties

0. BST property is maintained for all nodes in the B-Tree
1. All leaf nodes are of same depth
2. All non-root nodes have between $b-1$ to $2b-1$ keys inclusive, where b is the branching factor
3. The root has at most $2b-1$ keys (no minimum)
4. An internal node with k keys must have $k+1$ subtrees/children

Hashing

Load Factor

$\alpha = \frac{n}{m}$, where
n = number of items in the table
m = size of table

Simple Uniform Hashing

Under SUHA, in a hash table where collisions are resolved by chaining, searching takes on average $\Theta(1+\alpha)$. If $n = O(m)$, $\alpha = O(1)$ and searching becomes $O(1)$.

Hash Functions

- Fast to compute, preferably $O(1)$
- Uniform, probability of hitting a bucket is same for all buckets
- Deterministic, hash of the same key should always give same answer

Division Method

- $h(k) = k \mod m$, given a key k and m slots to map to
- m should not be a power of 2
  - If $m = 2^p$, then $h(k)$ is just the p LSB of k
  - Unless we know all the p LSB of k follows a normal distribution, using another m is better
- m should be prime not near any power of 2
- k and m should be relatively prime

Multiplication Method

- $h(k) = \lfloor m(kA \mod 1) \rfloor$, given a key k, m slots to map to, and $0 < A < 1$
- $kA \mod 1 = kA - \lfloor kA \rfloor$, where kA is a float
- m is typically a power of 2
- Knuth recommends $A \approx \frac{\sqrt{5}-1}{2}$

Linear Probing

- index $i = (h(k) + \mathit{step} \times 1) \mod m$
- Suffers from primary clustering
- Initial probe determines the entire sequence, and so only m distinct probe sequences are used

Quadratic Probing

- If 2 keys have the same initial probe position, then their probe sequences are the same
- Suffers from secondary clustering
- Initial probe determines the entire sequence, and so only m distinct probe sequences are used

Double Hashing

- $\Theta(m^2)$ probe sequences are used

Chaining

- Different keys with the same hashes are stored in a linked list
- Java's implementation

Amortized Analysis

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

> Note: **every operation** must satisfy the above

Graph

Graph $G = \langle V, E \rangle$ ("A tuple of 2 sets")
V is a set of nodes/vertices
E is a set of edges/links

Definitions

1. Diameter - maximum distance between any two nodes
2. Distance - shortest path between two nodes
3. Degree - number of edges incident upon it (self loops are counted twice)
4. Connected - $\exists$ path between any two nodes
5. Component - a subgraph in which any two vertices are connected to each other by paths
6. Simple Path - a sequence of edges that do not contain cycles

Simple Graph (No Self Loops)

1. $e = (v, w)$ for $v \neq w$

Clique (Complete Graph)

All nodes are connected to one another
Diameter = 1

Star

One central node. All edges connect center to the other nodes
Diameter = 2

Line

Diameter = $n - 1$
Max Degree = 2

Cycle

Diameter = $\frac{n}{2}$
Max Degree = 2

Space Complexity

1. **Adjacency Matrix:** $O(V^2)$
    - Array of size $V \times V$
2. **Adjacency List:** $O(V)$
    - Array of size V

    - Linked list of size $E = 2$
- Generally: If graph is dense $[E = \Theta(V^2)]$, use adjacency matrix. Else, use adjacency list.

Time Complexity

Query	Adjacency Matrix	Adjacency List	Edge List
Check if v and w are neighbours	$O(1)$	$O(V)$	$O(E)$
Find any neighbour	$O(V)$	$O(1)$	$O(E)$
Enumerate all neigbours	$O(V)$	Possible $O(1)$	$O(E)$
Find V with max E	$O(V^2)$	$O(V + E)$	$O(E)$
Given V, find $E > N$	$O(V)$	$O(V)$	$O(E)$

- Convert adjacency list to planar graph: $O(V)$

Breadth-First Search (BFS)

```
BFS(G, s, f)
    visit(s)
    Queue.add(s) # FIFO
    while not Queue.empty()
        curr = Queue.dequeue()
        if curr == f
            return curr
        else
            for each neighbor u of curr
                if u is not visited
                    visit(u)
                    Queue.enqueue(u)
    return null
```

1. See if current node is correct element
2. If not, add all neighbours (if not visited yet) to a queue
3. Recursively do this algorithm for each element in the queue

Can fail for graphs with > 1 components

Each V must be visited once: $O(V)$
For every V, each E must be examined once: $O(E)$

Time: $O(V + E)$

Depth-First Search (DFS)

```
BFS(G, s, f)
    visit(s)
    Stack.add(s) # LIFO
    while not Stack.empty()
        curr = Stack.dequeue()
        if curr == f
            return curr
        else
            for each neighbor u of curr
                if u is not visited
                    # store path:
                    edgeTo[u] = curr
                    visit(u)
                    Stack.enqueue(u)
    return null
```

Time: $O(V+E)$

BFS & DFS

- To find shortest path, unweighted edges: **only BFS** works
- To find shortest path, weighted edges: **none** works
- Both visits **every node** and **every edge**, **but not every path**

Topological Sort

Topological Ordering:

1. Sequential total ordering of all nodes
2. Edges only point forward
3. Only possible $\iff$ Graph is a DAG

Kahn's Algorithm

1. Start at node v with no incoming edges, and add v to a list
2. Remove all outgoing edges from v, and go back to step 1

```
L = list()
S = list()
add all nodes with no incoming edge to S
while S is not empty:
    remove node v from S
    add v to tail of L
    for each of v's neighbors u
        remove edge e where source is v
        if u has no other incoming edges
            add u to S
```

Time: $O(V+E)$

> Topological ordering **are not unique**

Shortest Path for DAG, $O(V+E)$ time:

1. Let d be an array of the same length as V; this will hold the shortest-path distances from s. Set $d[s]=0$, all other $d[u]=\infty$
2. Let p be an array of the same length as V, with all elements initialised to `null`. Each $p[u]$ will hold the predecessor of u in the shortest path from s to u
3. Loop over the vertices u as ordered in V, starting from s:
    1. For each vertex v directly following u (ie. there exists an edge from u to v):
        1. Let w be the weight of the edge from u to v
        2. Relax the edge: if $d[v] > d[u] + w$, set
            1. $d[v] = d[u] + w$,
            2. $p[v] = u$

Relaxation Property

Let $\delta[u, v]$ denote the shortest distance from u to v, $d[u, v]$ denote the distance estimate from u to v, and $w[u, v]$ be the edge weight between u and v:

1. If $G+(V, E)$ contains **only positive weights**, then the shortest path p from source vertex s to a vertex v must be a **simple path**
2. If $G+(V, E)$ contains **no negative weight cycles**, then the shortest path p from source vertex s to a vertex v must be a **simple path**
3. $\delta[s, v] \leq \delta[s, u] + w[u, v]$
    - shortest path from s to v must be $\leq$ sum of (shortest path from s to u) $+$ (going from u to v)
4. $\forall v \in V, d[s, v] \geq \delta[s, v]$
    - once $d[s, v] = \delta[s, v]$, it never changes
5. If $\exists \delta[s, u]$, then relaxing edge $[u, v]$ will yield $\delta[s, v]$
    - holds true regardless of any other relaxation steps that occur before relaxing edge $[u, v]$ (due to the `if` part in the relaxation code)
6. Once $d[s, v] = \delta[s, v], \forall v \in V$, the predecessor subgraph is a shortest-path tree rooted at s

> A shortest path is a tree with at most $|V|-1$ edges
> Thus, to get shortest path, relax all $|E|$ edges, $|V|-1$ times

```
relax(int u, int v)
    if (dist[v] > dist[u] + weight(u, v))
        dist[v] = dist[u] + weight(u, v)
        prev[v] = u # store predecessor node
```

Bellman-Ford Algorithm (SSSP, Negative Edges)

```
BellmanFord(V, E)
    n = V.length
    for i = 1 to n - 1
        for Edge e in E
            relax(e)
```

- Outer for loop runs $V-1$ times

- Can be **terminated early** if an entire sequence of $|E|$ relax operations have no effect
- To find out if $\exists$ negative cycles, run one more time of relaxation
- To find out exactly which nodes are affected by a negative cycle, run $V - 1$ more times of relxation

Time: $O(VE)$

Dijkstra's Algorithm (SSSP, Non-Negative Edges)

> Only works for graphs with **non-negative weight edges**

1. Begin with empty shortest-path-tree
2. Repeat
    1. Consider vertex with **minimum estimate**
    2. Add vertex to shortest-path-tree
    3. Relax all outgoing edges

Step 2.1 (`insert` / `deleteMin`): done V times $\rightarrow$ each node is added to the PQ once
Step 2.3 (`relax` / `decreaseKey`): done E times $\rightarrow$ each edge is relaxed once
Each PQ operation is $O(\log V)$

Time: $O((V+E)\log V)$

Condition	Algorithm	Time Complexity
No negative weight cycles	Bellman-Ford	$O(VE)$
Unweighted/equal weight graphs	BFS	$O(V+E)$
No negative edge weights	Dijkstra's	$O((V+E)\log V)$
On Tree	BFS/DFS	$O(V)$
On DAG	Topological Sort	$O(V+E)$

Floyd Warshall's Algorithm (APSP)

- Subpaths of shortest paths are themselves shortest paths
- Only gives correct answer if graph **does not have negative cycles**
- To detect negative cycles, check if diagonals of result has negative value (ie. travelling from a node back to the same node is negative)

```
Function FloydWarshall(G)
    # memoization table S has |V| rows and |V| columns
    S = Array of size |V|x|V|
    # Initialize every pair of nodes
    for v = 0 to |V|-1
        for w = 0 to |V|-1
            S[v,w] = E[v,w]

    # For sets P0, P1, P2, P3, ..., for every pair (v,w)
    for k = 0 to |V|-1
        for v = 0 to |V|-1
            for w = 0 to |V|-1
                S[v,w] = min(S[v,w], S[v,k]+S[k,w])
    return S
```

Time: $O(V^3)$

Minimum Spanning Tree

Definitions:

1. Spanning Tree: an acyclic subset of the edges that connects all nodes
2. MST: A spanning tree with minimum weight
3. Cut: a partition of the vertices into 2 disjoint subsets

Assumptions:

- Edge weights are distinct
- Graph is connected

Properties of MSTs:

1. No cycles
2. If you cut an MST, the two pieces are both MSTs
3. $\forall$ cycle, the maximum edge is **not** in the MST
4. $\forall$ partition of nodes, the minimum weight edge across the cut is in the MST
5. $E = V - 1$

Warnings:

- MSTs **cannot** be used to find the shortest path
- $\forall$ cycle, the minimum weight edge **may or may not** be in the MST
- $\forall$ vertex, the minimum incident/outgoing edge **is always** part of the MST
- $\forall$ vertex, the maximum incident/outgoing edge **may or may not** be part of the MST
- Kruskal's and Prim's only work on undirected graphs

Prim's Algorithm

Utilises Cut Property

Idea:

1. Maintain set of visited nodes
2. Greedily grow the set by adding node connected via the lightest edges
3. Use min heap to order nodes by edge weight

Each vertex is added/removed once from heap: $O(V \log V)$
Each edge can incur one `decreaseKey`: $O(E \log V)$
Time: $O(E \log V)$

> Can only work on undirected graphs

Disjoint Set (Union-Find)

Any sequence of m union/find operations on n objects takes $O(n + m\alpha(m, n))$ time, where α is the inverse Ackermann function (always ≤ 5)

Kruskal's Algorithm

Basic idea: recursively choose the minimum weight edge to connect 2 disconnected trees

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

// 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
    }
}
```

Sorting (line 2): $O(E \log E)$
Union (line 11-13): $O(E\alpha)$
Time: $O(E \log E)$

> Can only work on undirected graphs