⊙ **Binary Search** $O(\log n)$

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

Quick Select $O(n)$
Bubble Sort: $\Omega(n)$ (already sorted), $O(n^2)$ (`n` iterations)
Selection Sort: $\Omega(n^2)$, $O(n^2)$
Insertion Sort: $\Omega(n)$ (sorted), $O(n^2)$ (inverse sorted)
Merge Sort: $O(n \log n)$
Quick Sort $\Omega(n \log n)$, $O(n^2)$
`3wayPartition`: two pass, pack duplicates // one pass, four regions

⊙ **Binary Search Trees:**
**Delete:** $O(h)$
Case 1: no children: Remove `v`
Case 2: 1 child: Remove `v`, Connect `v.child` to `v.parent`
Case 3: 2 children: `x = successor(v)`, `delete(x)`, subst `v` with `x`
**Successor:** $O(h)$
Case 1: has right child, search min in right
Case 2: no right child, go up to the first right

⊙ **AVL Trees:**
Store `v.height = h(v)` $|$`v.left.height` $-$ `v.right.height` $|\leq 1$
**Insert:** ($O(1)$ rotations)
left-balance: `right` left-left: `right` left-right: `left(left)`, `right`
**Delete:** ($O(\log n)$ rotations)
if `v` has two children, swap with successor, `delete(v)`
check ancestor balance and rotate if needed

⊙ $(a, b)$**-Trees:**
**Rule 1:** $(a, b)$**-child Policy**
Root: $[2, b]$ children, $[1, b-1]$ keys
Internal: $[a, b]$ children, $[a-1, b-1]$ keys Leaf: $[a-1, b-1]$ keys
**Rule 2: Key Ordering**
Sorted keys $v_1, v_2, \ldots, v_k$, subtrees $t_1, t_2, \ldots, t_{k+1}$
$t_1$: $(-\infty, v_1]$ $t_i$, $i \in [2, k]$: $(v_{i-1}, v_i]$ $t_{k+1}$: $(v_k, \infty)$
**Rule 3: Leaf Depth**
All leaf nodes have same depth, grow root upwards
Max height: $O(\log_a n) + 1$ Min height: $O(\log_b n)$

⊙ **Tries:**
Search: $O(L)$ Insert: $O(L * \text{overhead})$

⊙ **Order Statistics Trees:**
Store `v.weight = w(v)`, update weight ($O(1)$) on insert & delete

```
TreeNode select(k)
int order(TreeNode)
  int rank = node.left.weight + 1;
  while (node != null)
    if (node.isRightChild())
      rank += node.parent.left.weight + 1;
    node = node.parent;
```

⊙ **Interval Trees:**
Store `v.max = max(node.right, left.max, right.max)`

**Interval search:** $O(\log n)$

```
Pair intervalSearch(x)
  TreeNode c = root;
  while (c != null && !c.contains(x))
    if (c.left == null) c = c.right;
    else if (x > c.left.max) c = c.right;
    else c = c.left;
  return c.interval;
```

**All-overlaps search:** $O(k \log n)$

Repeat: Search for interval, add to list, delete interval
Repeat for all intervals: Add back to tree

⊙ **Orthogonal Range Searching:**
Store all points in the leaves, internal nodes store `left.max`
**Query:** $O(k + \log n)$

```
TreeNode findSplit(low, high)
  TreeNode v = root;
  while (true)
    if (high <= v. key) v = v.left;
    else if (low > v.key) v = v.right;
    else break;
```

```
void leftTraversal(v, low, high)
  if (low <= v.key) allTraversal(v.right);
                    leftTraversal(v.left, low, high);
  else leftTraversal(v.right, low, high);
```

```
void rightTraversal(v, low, high)
  if (v.key <= high) allTraversal(v.left);
                     rightTraversal(v.right, low, high);
  else rightTraversal(v.left, low, high);
```

⊙ **Chaining:**
Insertion: $O(1 + \text{cost}(h)) = O(1)$ Space: $O(m + n)$
Search: worst case $O(n + \text{cost}(h)) = O(n)$
expected $O(\alpha + cost(h)) = O(1)$
⊙ **Open Addressing:**
**Linear Probing**: $h(k, i) = h(k, 1) + i \bmod m$
**Double Hashing**: $h(k, i) = f(k) + i \cdot g(k) \bmod m$
Sensitive to choice of $h$ (clustering) and load $\alpha$
Remedy: grow and shrink table, $O(m_1 + m_2 + n)$
(scan, initialise, compute & copy)
$(n == m) \rightarrow m = 2m \mid (n < m/4) \rightarrow m = m/2$
Insertion: amortised $O(1)$ Search: expected $O(1)$

⊙ **Fingerprint:**
Store `0/1` bits, does not store key, no false -ve, have false +ve
⊙ **Bloom Filter:**
Use $k$ hash functions for fingerprint, no false -ve, have false +ve
⊙ **BFS (queue)** $O(V + E)$**:**

```
void BFS(Node[] nodeList, int startNode)
  frontier.add(startNode);
  while(!frontier.isEmpty())
    for (int v : frontier)
      for (int w : nodeList[v].nbrList)
        if (visited[w]) continue;
        visited[w] = true; parent[w] = v;
        nextFrontier.add(w);
    frontier = nextFrontier
```

⊙ **DFS (stack)** $O(V + E)$ **list,** $O\left(V^2\right)$ **matrix:**

```
void DFS_visit(Node[] nodeList, boolean[] visited, int
startNode)
  for (Integer v : nodeList[startNode].nbrList) {
    if (visited[v]) continue;
    visited[v] = true;  DFS_visit(nodeList, visited, v);
```

⊙ **Bellman-Ford** $O(EV)$**:**
After $k$ iteration, $k$-hop estimates are correct
(topo sorted DAG one-pass)

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

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

⊙ **Dijkstra's (pq)** $O(E \log V)$:
Each vertex added to PQ once $O(V \log V)$
Each edge relaxed once $O(E \log V)$
No negative edges

```
void searchPath(int start)
  pq.insert(start, 0);
  while(!pq.isEmpty())
    int w = pq.deleteMin();
    for (Edge e : G[w].nbrList) relax(e);

void relax(int u, int v, int weight)
  if (distTo[v] > distTo[u] + w)
    distTo[v] = distTo[u] + weight;  parent[v] = u;
    if (pq.contains(v)) pq.decreaseKey(v, distTo[v]);
    else pq.insert(v, distTo[v]);
```

⊙ **Post-Order DFS** $O(V + E)$:
⊙ **Kahn's Algorithm** $O(E \log V)$:
Nodes in pq, edges as keys
removal $O(\log V)$, decreaseKey $E \times O(\log V)$

```
Repeat:
  S = all nodes with no incoming edges
  add nodes in S to topo-order
  remove all edges adjacent to nodes in S
  remove nodes in S from graph
```

⊙ **Heap:**
`priority[parent] >= priority[child]`
Complete binary tree, $O(\log n)$ height
**Insertion / Deletion:**
Add to last, bubble up / Swap with last node, remove, bubble down
**increaseKey / decreaseKey:**
Update priority, bubble up / down

```
void bubbleUp(Node v)
  while (v != null)
    if (priority(v) <= priority(parent(v))) return;
    swap(v, parent(v)); v = parent(v);

void bubbleDown(Node v)
  while (!isLeaf(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;
```

⊙ **Heap Sort** $O(n \log n)$:

```
left(x) = 2x + 1;  right(x) = 2x + 2;
parent(x) = floor((x-1)/2);
```

**1. Unsorted** $\rightarrow$ **Heap** $O(n)$

```
for (int i = n-1; i >= 0; i++) bubbleDown(i, A);
```

**2. Heap** $\rightarrow$ **Sorted** $O(n \log n)$

```
for (int i = n-1; i >= 0; i--)  // in-place, unstable
  int value = extractMax(arr);  // O(log n)
  arr[i] = value
```

⊙ **Union Find:**
**1. Quick Find:** Check component ID, Find $O(1)$, Union $O(n)$
**2. Quick Union:** Check root ID, Find $O(n)$, Union $O(n)$
**3. Weighted Union:** Merge by size, Find $O(\log n)$, Union $O(\log n)$
**4. Path Compression:** Pointer to root, Find $O(\log n)$, Union $O(\log n)$

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

**5. WU with PC:** Find $\alpha(m, n)$, Union $\alpha(m, n)$

⊙ **MST:**
1 No cycles  2 Cut still MST
3 Max weight in cycle is **not in**  4 Min weight across cut is **in**
⊙ **Directed MST** $O(E)$:
Add min weight incoming edge (DAG with one root)

⊙ **Maximum Spanning Tree:**
Negate every edge + MST / Kruskal's in reverse

⊙ **Red-Blue:**
Red max weight edge in cycle  Blue min weight edge across cut

⊙ **Prim's** $O(E \log V)$:

```
while (!pq.isEmpty())
  Node v = pq.deleteMin();
  S.add(v);
  for (Node w : v.nbrList)
    if (!S.get(w))
      pq.decreaseKey(w, weight(v, w));
      parent.put(w, v);
```

⊙ **Kruskal's** $O(E \log V)$:
Sorting $O(E \log E)$ Union & Find $O(\log V)$

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

⊙ **Boruvka's** $O((V + E) \log V) = O(E \log V)$:
Initial $O(V)$ (store identifier):
$n$ connected component, one for each node
Boruvka $O(V + E)$:
For each connected component, add min weight outgoing edge
$O(V + E)$ (BFS/DFS)
Merge components $O(V)$ (update identifier)
Each step $k \rightarrow k/2 \implies O(\log V)$ steps

⊙ **2-opt Steiner Tree:**
For every pair of required nodes, calculate shortest path
Construct new graph on required nodes
Run MST on new graph
Map new edges back to original graph

optimal tree $\leq$ DFS without Steiner nodes $\leq$ 2 * OPT

⊙ **Dynamic Programming:**
Optimal sub-structure  Overlapping sub-problems

⊙ **Longest Increasing Subsequence:**
`S[i] = max(S[j]) + 1` where `j` $\in$ `E`
$n$ sub-problems, `S[i]` takes $O(i)$, $O(n^2)$

⊙ **Lazy Prize Collecting:**
`P[v,0] = 0`
`P[v,k] = max(P[w,k-1]` + `weight(v,w))` where `w` $\in$ `v.nbrList()`
$kV$ sub-problems, each takes $O(|$`v.nbrList`$|)$, $O(kV^2)$
$k$ rows, $E$ to solve each row, (examine each edge once), $O(kE)$

⊙ **Vertex Cover:**
`S[leaf,0] = 0`, `S[leaf,1]= 1`
`S[v,0]` = $\sum$ `S[w,1]` where `w` $\in$ `v.nbrList()`
`S[v,1]` = 1+$\sum$ `min(S[w,0], S[w,1])` where `w` $\in$ `v.nbrList()`
$2V$ sub-problems, each takes $O(V)$, $O(V^2)$
Each edge examined once, $O(V)$

⊙ **Floyd-Warshall** $O(V^3)$:
`S[v,w,P]`: shortest path `v` $\rightarrow$ `w` that only uses nodes in set `P`
`S[v,w,`$\varnothing$`] = weight(v,w)`
`S[v,w,`$P_n$`] = min(S[v,w,`$P_{n-1}$`],S[v,n,`$P_{n-1}$`] + S[n,w,`$P_{n-1}$`])`