Master Theorem

$$T(n) = aT(\frac{n}{b}) + f(n), \qquad f(n) = \Theta(n^k log^p n),\ where\ a \geq 1\ and\ b > 1$$

Case 1: if $log_b a > k$
- then, $\Theta(n^{\log_b a})$

Case 2: if $log_b a = k$
- if $p > -1$, then $\Theta(n^k log^{p+1} n)$
- if $p = -1$, then $\Theta(n^k loglogn)$
- if $p < -1$, then $\Theta(n^k)$

Case 3: if $log_b a < k$
- if $p \geq 0$, then $\Theta(n^k log^p n)$
- if $p < 0$, then $\Theta(n^k)$

Master Theorem for subtract and conquer

$$T(n) = aT(n-b) + f(n), \qquad n > 1\ \ for\ \ a > 0\ \ b > 0,\ \ f(n) = O(n^k), k \geq 0$$

Case 1: if $a < 1$, then
$T(n) = O(n^k)$

Case 2: if $a = 1$, then
$T(n) = O(n^{k+1})$

Case 3: if $a > 1$, then
$T(n) = O(n^k a^{\frac{n}{b}})$

Recurrence relation	big-Oh	Recurrence relation	big-Oh
T(n) = T(n/2) + O(1)	$O(logn)$	T(n) = 2T(n−1) – 1	$O(1)$
T(n) = T(n/2) + O(n)	$O(n)$	T(n) = T(√n) + O(1)	$O(loglogn)$
T(n) = T(n/2) + O(n^2)	$O(n^2)$	T(n) = 2T(√n) + O(1)	$O(logn)$
T(n) = T(n-1) + O(1)	$O(n)$	T(n) = T(√n) + O(n)	$\approx O(n)$
T(n) = 2T(n/2) + O(1)	$O(n)$	T(n) = 2T(√n) + O(n)	$O(n)$
T(n) = T(n-1) + O(n)	$O(n^2)$	T(n) = 2T(n-1) + 1	$O(2^n)$
T(n) = 2T(n/2) + O(n)	$O(nlogn)$	T(n) = 3T(n-1)	$O(3^n)$

Sorting Algorithms - Best | Average | Worst | Stability

Bubble: $\Omega(N)|\Theta(N^2)|O(N^2)|Stable$ → At the end of iteration **j**, the biggest **j** items are correctly sorted in the right most **j** position of array.

Selection: $\Omega(N^2)|\Theta(N^2)|O(N^2)|Not\ Stable$ → At the end of iteration **j**: the **smallest j** items are correctly sorted in the first **j** positions of the array. Always traverse entire array, always N^2.

Insertion: $\Omega(N)|\Theta(N^2)|O(N^2)|Stable$ → At the end of **j** iteration: the first **j** items in the array are in sorted order. (NOT necessarily the smallest). Best/Worst: Already sorted vs Reverse Sorted

Quick: $\Omega(N\ logN)|\Theta(N\ logN)|O(N^2)|Stable$ → At the end of every loop iteration: for all *i ≥ high, A[i] > pivot*, *for all 1 < j < low, A[i] < pivot*. Worst: Pivot smallest/biggest item.

Merge: $\Omega(N\ logN)|\Theta(N\ logN)|O(N\ logN)|Stable$ → At the end of **j** iteration, each half of the array that is split is already sorted. Stable if `merge()` is stable.

Heap: $\Omega(N\ logN)|\Theta(N\ logN)|O(N\ logN)$

Binary Search: $\Omega(1)|\Theta(logN)|O(logN)$ → At the end of iteration j, the active search region between begin and end is size at most $\frac{n}{2^j}$, and the item (if it exists) is in that region.

```java
static int genBinarySearch(int[] arr, int key, int k) {
  int low = 0; int high = arr.length - 1;
  while(low <= high) {
    int pivot = low + (high - low) / k;              //two halves split with fraction k
    if (arr[pivot]==key) return pivot;               //corner case: found target
    else if (arr[pivot] > key) { high = pivot - 1; } //decrease high (search LHS)
    else {  low = pivot + 1;  }                      //increase low (search RHS)
  } return -1 //not found anywhere;
} //realise that when k=2, this reduces to our vanila binary search
```

Data Structures

1. **Binary Trees** - Unbalanced, in-order, dynamic, traversable

   Time complexity of all BST Operations = $O(h)$, where h is the height of BST.
   Invariant: all in left sub-tree < key < all in right sub-tree.
   Best Case: BST is balanced, height of BST is $O(logn)$, hence complexity is $O(logn)$.
   Worst Case: BST is skewered, height of BST becomes n, time complexity is $O(n)$.

2. **Tries** - Unbalanced, string-based
   Best Case/Worst Case: For all operations of Insertion/Deletion/Searching $O(N)$ **except** if the word is 1 letter long, in which case search time is $O(1)$ (best case for searching).

3. **(a,b) trees** - Degree [a,b], balanced, uniform height
   Invariants: After every operation, every leaf has same depth. Every node has degree in the range [*a*,*b*]. Nodes satisfy tree-order property.

Node	Min Keys	Max Keys	Min Child	Max Child
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

   - An (a, b)-tree with n **nodes** has $O(log_a n)$ height, binary search for a key at every node takes $O(log_2 b)$ time.
   - Total cost = $O(log_2 b\ *\ log_a n)$
     (Cost of searching a node * Height)
     = $O(log_a n)$ * $(log_2 b$ const) = $O(logn)$

4. **AVL Trees** - Like binary trees, but height-balanced. Height-balanced tree has **at most** *height* $h < 2log(n)$, $\therefore$ **at least** $n > 2^{\frac{h}{2}}$ *nodes*. **at most** $2^{h+1} - 1$ *nodes*.
   Invariants: After every operation, every node is height-balanced.
   Best Case: $O(logn)$, for insertion/deletion/searching.
   Insertions take **at most** 2 rotations to balance as insertions DO NOT reduce heights. Deletion can take **more than** 2 rotations to balance. Rebalancing does not "undo" the change in height since rebalances usually reduces height as well.
   Worst Case: should tree be unbalanced from leaf-to-root upon deletions, it would take $logn$ rotations still, hence worst time complexity would be $O(logn)$.
   Updating AVL Trees:

   a. **Rotations:** Rotations only affect current rotating node and parents of rotating node, thus we update from **root-to-currNode** path.

   b. **Inserting:** Depend on how the variable affected is initialized, update while inserting/after.
      - If balanced, update variables from root-to-leaf path (if needed).
      - If not balanced, update first and conduct necessary rotations.

   c. **Deleting:** Three cases
      → Node is just a leaf → just delete and update parents.
      → 1 child → just delete and update from root to deleted node's variable, connect **parent** of deleted node to **child** of deleted node.
      → 2 children → Find successor (maximum of left-subtree of deleted node), delete node, replace with successor and update varibales. Have to update the parents of deleted nodes and the left sub-tree of deleted node (since a node is taken from there

5. **Interval Trees** Search(key) $\Rightarrow O(logn)$

   If value is in root, return. If value > max(left sub-tree) recurse right. else left (can't go right).

6. **Orthogonal Range Searching:** Binary trees; leaves store points internal nodes→left subtree max

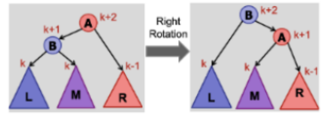
G.P $S_n = \dfrac{a(1-r^n)}{1-r}$

A.P $S_n = \dfrac{n}{2}[2a\ +\ (n-1)d]\ \ OR\ \ S_n = \dfrac{n}{2}[a+l]$

rebalancing

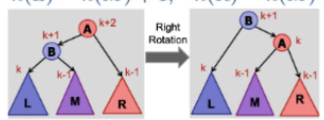
[case 1] B is **balanced: right-rotate**

$h(L) = h(M), \quad h(R) = h(M) - 1$



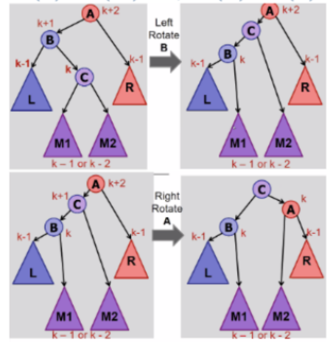
[case 2] B is **left-heavy: right-rotate**

$h(L) = h(M) + 1, \quad h(R) = h(M)$

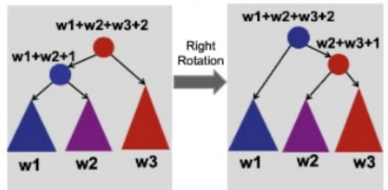


[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

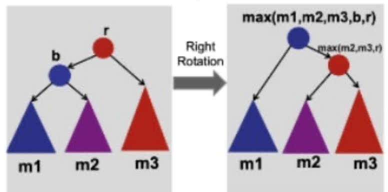
$h(L) = h(M) - 1, \quad h(R) = h(L)$



weights



max



Series	Asymptotic Bound
$n\ +\ \frac{n}{2}\ +\ \frac{n}{3} + ...\ +\ \frac{n}{n}$ $= n(1\ +\ \frac{1}{2}\ +\ ...\ +\ \frac{1}{n})$	$\approx n(logn)$ $O(nlogn)$
$1+2+3+4+\ldots+n = \frac{n(n+1)}{2}$	$= O(n^2)$
$n + \frac{n}{2} + ... + 1$ $= n + n(\frac{1}{2}) + n(\frac{1}{2})^2 + ... + n(\frac{1}{2})^n$	$\approx O(n)$
$1 + c + c^2 + ... + c^n$ if $\lvert c\rvert < 1$ = $O(1)$ OR if $\lvert c\rvert > 1$ = $O(c^n)$	
$1^2 + 2^2 + 3^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$	$O(n^3)$

7. Hash Tables

Let m be the table size and let n be the number of items; let $cost(h)$ be cost of hash function. $load(hash\ table)$, $\alpha = \frac{n}{m}$ → average/expected number of items per bucket.

- **Simple Uniform Hashing Assumption:** every key has an equal probablity of being mapped to every bucket, keys are mapped indenpendently.
- **Uniform Hashing Assumption:** every key is equally likely to be mapped to every permutation, independent of every other key, NOT fulfilled by linear probing.
- **Properties of a good hash function:** able to enumerate all possible buckets - $h : U \to \{1...m\}$ for every bucket j, $\exists i$ such that $h(key, i) = j$ && SUHA

Rules of `hashCode()` function: (rules for equals method → equivalence relation).

1. Always returns the same value if the object has not changed.
2. If two objects are equal, then they return the same hashCode.
3. **MUST redefine `.equals()` to be consistent with `hashCode()`.**

Chaining

`insert(key, value)` → $O(1 + cost(h)) \Rightarrow O(1)$,
for n items, expected max cost = $O(logn)$ → $O(\frac{logn}{log(logn)})$

`search(key)` → Worst case: $O(n + cost(h)) \Rightarrow O(n)$,
expected case, $O(\frac{n}{m} + cost(h)) \Rightarrow O(1)$ (Searching through a linked list).

Open addressing - linear probing

redefined hash function: $h(k, i) = h(k, 1) + i \mod m$

`delete(key)` : use a tombstone value - DON'T set to *null*

Performance

if the table is $\frac{1}{4}$ full, there will be clusters of size $\Theta(logn)$ expected cost of an operation, $E[\#probes] \leq \frac{1}{1-\alpha}$, (assume α < 1 and uniform hashing), degrades badly as α → 1

advantages

- saves space (use empty slots vs linked list)
- better cache performance (table is one place in memory)
- rarely allocate memory (no new list-node allocation)

disadvantages

- more sensitive to choice of hash function (clustering)
- more sensitive to load (as α → 1, performance degrades)

8. Table size

- If $(n == m)$, then $m = 2m$.
  If $(n < \frac{m}{4})$ then $m = \frac{m}{2}$.
- Everytime double a table of size m, at least $\frac{m}{2}$ new items were added (pay for doubling).
- Everything shrink a table of size m, at least $\frac{m}{4}$ items were deleted (pay for shrinking).
- Let m_1 be size of old hash table, m_2 be size of new hash table; n be number of elements. growing the table: $O(m_1 + m_2 + n) \approx O(n)$

table growth	resize	insert n items
increment by 1	$O(n)$	$O(n^2)$
double	$O(n)$	$O(n)$, average $O(1)$
square	$O(n^2)$	$O(n)$

9. **Amortized Analysis** - an operation has amortized cost $T(n)$ if for every integer k, the cost of k operations is ≤ $kT(n)$.

- binary counter ADT: increment ⇒ $O(1)$
- hash table resizing: $O(k)$for k insertions, search operation: expected $O(1)$ (not amortized)

$$1 < logn < \sqrt{n} < n < nlogn < n^2 < n^3 < 2^n < 2^{2n}$$
$$log_a n < n^a < a^n < n! < n^n$$

10. Graphs

- diameter: max. shortest path in the graph. **even cycles** are bipartite (every other node 1 grp).
- degree (graph): maximum degree of a node.
- degree (node): number of adjacent edges
- graph is **dense** if $|E| = \Theta(V^2)$, if **dense** use a Adjacency matrix instead.

adj	space	(cycle)	(clique)	use for
list	$O(V+E)$	$O(V)$	$O(V^2)$	sparse
matrix	$O(V^2)$	$O(V^2)$	$O(V^2)$	dense

- Searching

  BFS ⇒ $O(V + E)$, **parent edges** form a **tree** & shortest path from S, use queue.
  - $O(V)$ - every vertex is added exactly once to a frontier
  - $O(E)$ - every neighbourList is enumerated once

  DFS ⇒ $O(V + E)$, with adjacency matrix: $O(V)$per node ⇒ total $O(V^2)$, use stack.
  - $O(V)$ - DFSvisit is called exactly once per node
  - $O(E)$ - DFSvisit enumerates each neighbour
- Topological Ordering - Sequential total ordering of nodes + Edges only point foward.
  - Post-Order DFS ⇒ $O(V + E)$, prepend each node from a post-order traversal.
  - Khan's Algorithm ⇒ $O(ElogV)$, add nodes without incoming edges to topological order.
    - remove min-degree node from priority queue ⇒ $O(VlogV)$
    - `decreaseKey()` (in degree) of the child node ⇒ $O(ElogV)$
- Shortest Paths
  - **Bellman Ford** ⇒ $O(VE)$, $|V|$ iterations of relaxing, terminate when entire sequence of $|E|$ operations have no effect. Use to detect negative weight cycles, if same weight use BFS.
  - **Dijkstra's** ⇒ $O((E + V)logV) = O(E\ logV)$ → NO **Negative weight edges.**

    Dijkstra grows set by adding neighbouring node that is the **closest to source** (min-estimate

    Use a PQ to track the min-estimate to node, relax outgoing edges, add incoming to PQ.

    $|V|$ times of `insert()` / `deleteMin()` → $O(logV)$ each, (only inserted/removed once)

    $|E|$ times of `relax()` / `decreaseKey()` → $O(logE)$ each.

    Fibonacci heap ⇒ $O(E + VlogV)$, Array ⇒ $O(V^2)$
  - **DAG Shortest Path** ⇒ $O(V + E)$, Sort by topological order, then relax in this order

    Longest path: negate the edges/modify relax function
  - **Trees** ⇒ $O(V)$, relax each edge in BFS/DFS order.
- Union-Find, **Union** - connect 2 objects, **Find** - check if objects are connected.

	Find	Union
quick-find `int[] componentId` flat trees (connected if part of same componentId)	$O(1)$ - check if objects have the same `componentId`	$O(n)$ - enumerate all items in array to update `id`
quick-union `int[] parent` deeper trees (connected if same parent)	$O(n)$ - check for same root	$O(n)$ - add as a subtree of the root
weighted union `int[] parent, int[] size` (make shorter tree child of larger tree)	$O(logn)$ - check for same root	$O(logn)$ - add as **smaller** subtree of root
path compression (set parent of each traversed node to the root)	$O(logn)$	$O(logn)$
weighted union + path compression, for m union/find operations on n objects: $O(n + m\alpha(m,n))$,flat trees	$O(\alpha(m,n))$	$O(\alpha(m,n))$

11. Minimum Spanning Trees

Property 1: No cycles

Property 2: If you cut an MST, the two subtrees are both MSTs.

Property 3: Cycle property, For every cycle, the max weight edge is ***not*** in the MST.
False cycle property, For every cycle, the min weight edge may not be in the MST.

Property 4: Cut property, For every partition of nodes, min weight edge across cut is in MST.
→ for every vertex, minimum outgoing edge is in the MST.

Prim's Algorithm ⇒ $O(ElogV)$, grows set by adding node connected via lightest edge.

- add the minimum edge across the cut in the MST
- PQ to store nodes (priority: lowest incoming edge weight)
- each vertex: one `insert()` / `extractMin()` ⇒ $O(logV)$ each, hence total $O(V * logV)$
- each edge: one `decreaseKey()` ⇒ $O(ElogV)$

Kruskal's Algorithm ⇒ $O(ElogV)$, sort edges, add smallest edges that do not form a cycle.

- sorting ⇒ $O(ElogE) = O(ElogV)$
- each edge: `find()` / `union()` ⇒ $O(logV)$ using union-find DFS

12. Dynamic Programming

Optimal sub-structure - optimal solution constructed from solutions to smaller sub-problems

Overlapping sub-problems - can memoize solutions to sub problems

- optimal substructure but no overlapping sub-problems → divide-and-conquer
- **Longest Increasing sub-sequence**
  - DAG Solution ⇒ $O(n^3)$, find topological order $(O(V + E) = O(n^2))$, ran n times
  - S[i] = LIS(A[1...i]), ⇒ $O(n^2)$, find LIS up to each point in A[1...i] = $O(n)$, ran n times
- **Prize collecting** ⇒ $O(kE)$ → super source connects k graphs | $O(kV^2)$ for k steps (DP)
- **Vertex cover** ⇒ $O(V)$ or $O(V^2)$, set of nodes where every edge is adjacent to ≥ 1 node
- **All pairs shortest path** ⇒ $O(V * ElogV)$, dijkstra ran onto all nodes once (V times)
- **Diameter of a graph:** $O(V^2 logV)$, SSSP all nodes of a graph, (V times)
- **Floyd Warshall** ⇒ $O(V^3)$
  - $S[v, w, P_k]$ = shortest path from v to w only using nodes from set P
  - $S[v, w, P_8] = min(S[v, w, P_7]\ \ , S[v, 8, P_7] + S[8, w, P_7])$