GP to infinity: $\frac{a}{1-r}$

$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

AP:

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$

$\sum_{i=1}^{n} a_i = a_1 + ... + a_n = \frac{n(a_n + a_1)}{2}$

nCr $= \frac{n!}{(n-r)!r!} = \frac{n(n-1)(n-2)...(n-r+1)}{r!}$

Sum of squares

$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$

Harmonic Series

$\sum_{i=1}^{n} \frac{1}{i} \approx ln(i + 1)$

$log(n!) = n(log(n))$

Binary Search - must be sorted first

```
binarySearch(A, key,n)
begin = 0;
end = n - 1;
while (begin < end) do:
  int mid = begin + (end - begin) /2;
  If (key <= A[mid]) then
end = mid;
  else
       begin = mid + 1;
return (A[begin] == key) ? begin : -1;
// returns the index of the key in the array
```

Invariant:

A[begin] <= key <= A[end]

Iteration k: (end - begin) = $\frac{n}{2^k}$

BubbleSort: last j elements in correct order

QuickSort: pivot will always be in the correct pos

SelectionSort: the smallest j elements are correctly sorted in the first j positions.

InsertionSort: At the end of iteration j, the first j items in the array are in sorted order (last (n-j) elements will stay in same position)

MergeSort: 2 halves don't mix before final step

HeapSort(max): elements at the back are in order, elements in front obey heap order property.

Sorting Algorithm	Best Case		Worst Case	
	Swaps	Comparisons	Swaps	Comparisons
Selection Sort Not stable	0 – already sorted	O(n^2) – Still need to check	O(n) – always n-1 swaps	O(n^2) – always n
Insertion Sort	0 – already sorted	O(n) – compare O(n) times and break	O(n^2) – reverse order hence constant shifting	O(n^2) – constant comparing
Bubble Sort w/o flag	0 – already sorted	O(n^2) – no flag hence need to check all	O(n^2) – reverse order	O(n^2) – check all
Bubble Sort w/ flag	0 – already sorted	O(n) – early termination with flag	O(n^2) – reverse order	O(n^2) – reverse order or smallest element at the back
Merge Sort Not in-place	0 – all copying, which takes O(n log n), but no swaps	≤O(n log n) – comparisons stop once one sublist is fully copied in	0 – all copying, which takes O(n log n), but no swaps	≤O(n log n) – comparisons stop once one sublist is fully copied in
Quick Sort Not stable	O(n log n) - ~log n levels of n/2 swaps	O(n log n) – log n levels of n comparisons	O(n^2) – reverse order with O(n) swaps for n levels O(n) – if we are talking about the worst case of an already sorted array, where we have n levels of 1 swap (with itself).	O(n^2) – n levels of comparing n elements
Radix Sort Not in-place	0 – O(kn) copying	0 – non-comparison based sort	0 – O(kn) copying	0 – non-comparison based sort

HeapSort|O(log n)|O(n log n)|O(log n)|O(n log n)

Unstable & in-place

QuickSelect(A[1...n],n,k)

```
if (n == 1) then return A[1];
else Choose random pivot index pIndex.
  p = partition(A[1..n], n, pIndex)
  if (k == p) then return A[p];
  else if (k < p) then
    return QuickSelect(A[1..p–1], k)
  else if (k > p) then
    return QuickSelect(A[p+1], k – p)
Time Complexity = O(n)
```

Trees

$N = 2^{h+1} - 1 < 2^{h+1}$

If v is out of balance and left-heavy:

1. `v.left` is balanced: `right-rotate(v)`
2. `V.left` is left-heavy: `right-rotate(v)`
3. `v.left` is right-heavy: `left-rotate(v.left)` → `right-rotate(v)`

- starting from the bottom-most, rotate in the reverse direction that is heavy
  - e.g. v is left-heavy, v.left is right-heavy: `left-rotate(v.left)` and then `right-rotate(v)`

`insert(v) [O(log n), max 2 rotations]`

`delete(v) [O(log n)]`

Case 1: no children → just delete v

Case 2: 1 child → delete v → connect with `child(v)` to `parent(v)`

Case 3: 2 children → delete the element → reconnect with the successor of the deleted element

- `x = successor(v)`
- `delete(x)`
- `delete(v)`
- connect x to `left(v)` , `right(v)` , `parent(v)`
- Max (log n) rotations

Interval Query

```
interval-search(x)
  c = root;
  while (c != null and x not in c.interval) do
    if (c.left == null) then c = c.right;
    else if (x > c.left.max) then c = c.right;
    else c = c.left;
  return c.interval;
```

Hashing

- **keys are immutable**

Good Hash Functions feature:

1. h(key, i) enumerates all possible buckets
2. Simple Uniform Hashing Assumption
   - Every key is equally likely to be mapped to every bucket (perm), independently

Chaining:

E(insert) = O(1)

E(search/delete) = O(1)

Worst-case(search/delete) = 1+n/m = O(n)

E(max cost for inserting n items) = O(log n)

Open Addressing:

h(key,i) where *i* is number of collisions

E(operation) ≤ $\frac{1}{1-a}$, where a is load factor, n/m

If table is ¼ full → cluster size = O(logn)

Double Hashing: $h(k, i) = (f(k) + i \times g(k)) \mod m$

Table Resizing: O(m1 + m2 + n) = O(n)

if n == m then m = 2m: Every time you double a table of size m, at least m/2 new items were added

if n < m/4 then m = m/2: Every time you shrink a table of size m, at least m/4 items were deleted

Amortized Analysis:

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

FHT & Bloom Filter

- Cannot store keys
- False positives only, no false negatives
- p is p(false +ve), $\frac{n}{m} \leq ln(\frac{1}{1-p})$

If k hash functions,

p(collisions at all k spots) = $(1 - e^{-kn/m})^k$

Graphs

BFS/DFS	Queue/Stack: O(V + E) • can be used to find the shortest path for unweighted/uniformly weighted. • find connected components
Dijkstra	O((V + E)logV) Add/Remove each node from PQ - V*logV Relax/decreaseKey each edge - E*logV • At the start of each iteration of the while loop, d[v] is the shortest path from s to v for each vertex v in S. Every visited node has the correct estimate. • no negative weight & cannot reweight
Bellman Ford	O(VE) - After i-th iteration, the i-th node of the shortest path must have its distance estimate set to the correct value. As the path is at most |V|−1 edges long, |V|−1 round suffices to find this shortest path. If the V-th iteration still relaxed some edges → negative cycle After k iterations of the outer loop, at least k nodes within k hops of the source has a correct estimate -can use for acyclic graphs with negative weights
Kahn's	O(V + E) • Replace the visited array with an index array that keeps track of the indegree of every vertex in the DAG - O(E) • Use an ArrayList to record the vertices - O(V)
Prim's	O((E + V)logV) - same as Dijkstra's BUT can take -ve weight and -ve weight cycles
Kruskal	Sort edge list by weight from smallest to biggest. Sorting: O(E log E) = O(E log V) For E edges: • Find/Union: O(α(n)) or O(log V)
	Use UFDS to find if to and from nodes of edge are in the same tree (prevent cycle), union if no cycle.
Boruvka	O((V+E) log V) *One "Boruvka" Step: O(V+E)* - for each connected component, search for the min weight outgoing edges. -DFS/BFS: check if edge connects two components → O(V+E) - add selected edge, merge CC O(V) - every iteration *k* → *k/2* CC: log(V) times in total
SP DAG	TopoSort - O(V + E) Relax edges in topo order - O(E) <u>Tree:</u> DFS + relax O(V)

ModifiedDijkstra

```
PQ.enqueue((0, s)) // store pair of (dist[u], u)
while PQ is not empty
(d, u) ← PQ.dequeue()
if (d == D[u]) // important check
  if D[v] > D[u] + w(u, v) // can relax
       D[v] = D[u] + w(u, v) // relax
       PQ.enqueue((D[v], v)) //(re)enqueue
```

MINiMAX: MST then DFS/BFS to find max edge.

MAXiMIN: MaxST

Heap

1.Heap Ordering priority[parent] ≥ priority[child]

2. Complete binary tree - fill left to right

Note: second largest element is always the child of the root

Max h = floor(log n)

Insert/Delete: [O(log n)]; Search: O(n); FindMin: O(1)

Delete

1.swap node with the last one in the right subtree

2.remove the last node (node to be deleted) in the right subtree

3.bubble down the new subtree root

```
bubbleDown(Node v)
while (!leaf(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; }
```

index of root of left subtree ⇒ left(x) = 2x+1

index of root of right subtree ⇒ right(x) = 2x + 2

parent(x) = floor((x-1)/2)

UFDS

Name (per operation)	find	union
Quick Find	O(1)	O(n)
Quick Union	O(n)	O(n)
Weighted Union	O(log n)	O(log n)
Weighted Union w Path Compression	α(m;n)	α(m;n)
Path Compression (avg)	O(log n)	O(log n)
Path Compression (worst)	O(n)	O(n)

Dynamic Programming

Floyd-Warshall → Time: O(V^3), Space: O(V^2)

```
int[][] APSP(E){ // Adjacency matrix E
int[][] S = new int[V.length][V.length]; //memo
// Initialize every pair of nodes
  for (int v=0; v<V.length; v++)
     for (int w=0; w<V.length; w++)
          S[v][w] = E[v][w]
// For sets P0, P1, …, for every pair (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; }
```

<u>Bottom-Up (Tabulation) - Recursive MergeSort</u>

1. Solve the smallest problems
2. Combine smaller problems
3. Solve root problem

<u>Top-down (Memoization) - Floyd-Warshall</u>

1. Start at root and recurse
2. Solve and memoize using hash table/arr
3. Only compute each sol once

<u>DAG + TopoSort</u>

1. Topo sort DAG
2. solve in topo (reverse) order