QF: change id[], u in O(N), f in O(1), N^2 acc. **QF:** roots, u/f in O(N). **WQU:** move smaller subtree, u/f in O(lgN), space O(N).
WQUPC: connect node to root, wct O(N+Mlg*N) linear **TopM:** O(NlgM) M items, N input stream
MST: wct O(NlgN), space O(N), NlgN acc, stable
QS: wc cmps 0.5N^2, avg cmps Nlg, space O(N) for recursion, P[i and j cmp]=2NlgN+O(N)
HS: construct O(N), space O(1), cmps 2NlgN, delAllMax O(NlgN), put arr in heap order, delMax and move to end, O(.5N^2)
***any comparison-based sorting takes at least lg(N!) compares. **an alg is considered in place if uses <= clgN extra mem*
BST: wc sid O(N), avg sid rad(N), height O(rad(N)) w/ hibbard del, if pivot chosen is b/t i and j, they're never compared
Binary Heap: insert/delMax O(lgN), height lgN, perfectly balanced, insert lgN cmps, delete 2lgN cmps, parent >= child (heap order)
D-ary Heap: height log(b.d)N, faster swim O(log(b.d)N), slower sink O(dlog(b.d)N)
Treaps: wc hsid O(N), expected hsid logN, BST for keys, minheap for priorities, cmp to parent/child and **rotate!** history hidden
2-3 Trees: wc sid O(lgN) all 2 nodes, bc all 3 nodes 0.63lgN, deletion= no stepping into 2 node
LLRB: wc hsid O(2lgN), avg hsid O(lgN), insert with red edge, left rotation = red from right to left (turning car)
intIterator: O(N^2) time, O(N) space, copy into queue. **stackInterator:** O(N) time, O(H) space, push/pop continuously.
Hashing: every bin >= 1 ball after ~MlnM throws, after M throws, max in bin = logM / loglogM, load factor = N/M
Separate Chaining: linked list, wc sid N, avg sid 3-5, alpha>1, simple, one hash func
Linear Probing: open addressing, wc sid O(N), avg sid 3-5, avg probes search hit ~ 0.5(1+ 1/1-a), miss/insert ~ 0.5(1+ 1/(1-a)^2), a<0.5, faster bc consecutive in memory, simple, one hash func
2 Probe Hash: hash to 2, insert to shorter E[length longest chain] = loglogN. **Double Hash:** skip var amt, no cluster, hard deletion
Cuckoo Hash: wc insert infinite, O(1) search, could fail, 0.25 < load factor < 0.5, perfect, many funcs, less memory
Bloom Filter: is an element in the set? Array of M bits, k diff hash funcs maps set element to array slot. To add element, get k hash positions, change arr element from 0 to 1 O(k) time, P[false pos] =(1 – e^(KN/M)) ^k. as M↑ or N↑, K↑ and P↓

Persistent: duplicate in O(N) time/space per op, reuse nodes O(H) time/space per op, H=path traversed, 2 notes to rotate, no failfast iter
Adj list of edges: E+V space, find existance iterate in deg(V)
Path from v-w: DFS/BFS in E+V time. Shortest path v-w: BFS in E+V time. Digraph Search: DFS/BFS in O(V+E) time, O(V) space.
BFS: holds verts in queue. **DFS:** holds unreturned verts in an array, recursive. Multi-source SP: use BFS and enqueue all source verts
Topological sort: all DEdges point same way, reverse post order=opp recursion fin, if cycle exists, topo order impossible
Kosarsaju-Sharir: Kernal DAG (strong comp in 1 vert), finds strong comp in E+V time/space, DFSx2, makes copy of reversed digraph, run DFS in reverse topo order, start over=new comp
Greedy MST: any cut. {while G has a cut not crossed by T: e=min weight crossing edge bt C and V-C in MST, T+=e}
Prim: lazy: keep every frontier edge in PQ, insert and delMin in O(ElgV) time, PQ size = O(E). Eager: find lightest frontier edge, add to tree, if find better update item in indexed PQ, PQ size = O(V). {mark s, while not all vert marked: pick min e connecting marked to unmarked, T+=e, mark other endpt}

PQ type	insert	delMin	Δ entry	total	pros/cons
unordered arr	1	V	1	V^2	dense graphs
binary heap	logV	logV	logV	ElogV	fast for sparse
d-ary heap	log(b.d)V	dlog(b.d)V	log(b.d)V	Elog(b.E/V)V	best d = E/V
Fibonacci heap	1	logV	1	E+VlogV	best in theory

Kruskal's: add edges in ascending weight (PQ) unless cycle. Sort O(ElgV) time, O(E) space, check if cycle with UF in O(Elg*V) time, O(V) space, finds MST in O(ElogE) time. {for edge edge e of G: if T does not connect endpts of e, T+=e}
Euclidean: points in place, find all distances in 0.5N^2 time, use geometry = cNlogN
Boruvka's: each comp chooses lightest edge, k passes <= V/2^k verts, <= lgV phases, UF for comp, O(ElgV) time, O(V) space

Single source (1:all), single sink (all:1), source sink (1:1), all pairs
Relax: distTo[w] <= distTo[v] + e.weight()
Dijkstra's: (no neg edges) indexed PQ of verts by dist, relax all edges from that vert. space: lazy O(E), eager O(V), O(ElogV) time
Topo Sort/Acyclic: (no cycles) verts in topo order, relax edges from vert. O(E+V) time, V space
Longest pahts: negate weights, use Topo Sort (parallel job scheduling, critical path)
Bellman-Ford: (no neg cycles) relax each edge, repeat V times, after k times, found all paths using <= k edges, O(EV) time, V space
DP: compute value of distTo(v) after i iterations. Arbitrage detection: edge weight = -ln(exch rate), find neg cycle, arbitrage detect
Repeated Dijkstra's: run from evert vert, time/space = single Dij*V, good for sparse
Floyd-Warshall: allows neg edges, matrix describes graph, simple code, V^3 time, V^2 space, in D^(k), path of k edges (DP)
k+1 not used: O(1) time per subproblem

k+1 used:
Johnson's: good for sparse, run BF: get rid of neg edges: newHeight = oldHeight + h(u) – h(v), then Rep Dij, VE+V^3 time, V^2 space
**if neg cycle exists, no APSP (Johnson's detects neg cycles)

Reverse String: O(N^2) time. Compare Strings: O(shorter string) but often sublinear
Key-Indexed Counting: freq array size R+1, compute cumulative, traverse input increment location. O(N+R) time/space, stable
LSD Radix: same length, do key-indexed from right to left, 2W(N+R) time, N+R space, stable
MSD Radix: do key-indexed left to right, recursion, each func call needs own helper array. Slow for small subarrays, can be sublinear. wc time=2W(N+R), typical=Nlog(b.R)N, space=N+DR where D=length of longest prefix match/func call depth, stable, small R
3-Way Radix QS: in place, recursive, rand~2NlgN comps, wc time 1.39WNlgR, typical 1.39NlgN, space logN+W, unstable, big R
Suffix Arrays: sort in O(N^2logN) time
Trie: search tree, keys are strings, store chars in nodes, char-based ops, logN chars examined. Compressed=child w/o sib merge w parent

R-Way: (array) miss=log(b.R)N, hit/insert=L, space=N(R+1), fast, wastes space
Ternary: (BST) miss=lnN, hit/insert=L+lnN, space=4N, balanced=L+logN, naïve, left/right is BST, slower search, good space, balance
Choi: use hashtable for each node, char is in table, fast and low space, no ordered ops
Suffix tree: O(N) space/construction, suffix link=internal node-first char appears elsewhere, longest rep substring found in O(N) time
Brute Force: slow, little space, wc char cmps MN, need backup for every mismatch or need buffer
KMP: avoids backup, use DFS, if in state k, found match of first k chars, do what trailing state would do. Time O(MR+N), space O(MR)
NFA: correct char go right, wrong char go left, time O(N), space O(M), best wc
Boyer-Moore: scan pattern from right to left, skip up to M charps. Rand time sublinear O(N/M), wc O(MN), space O(M)
RabinKarp: hashmatching, rolling hash in O(1). If Q~MN^2, P[false collision]~1/N, wc time O(MN), typical O(N), space O(1), (((X%997) * 10) + 9) % 997 = newHash, MC assume correct, LV correct

Run-Length: exploits runs, use 4 bits to indicate how many zeros/ones alternating
Huffman: exploits frequencies, no code is prefix of another, make freq table, put ndoes in PQ, merge lightest. Left=0, right=1, expansion preorder traversal. Node=0, leaf=1, N+RlogR time. Dynamic model. ST for comp, trie for exp.
LZW: exploits repeated substrings, start at 81, trie with first layer given for compression, ST for exp. If use value immediately after creating it, last char is first char. Adaptive model. 81=BA, 83=BAB (82=AB)

BWT: og has repeated substring then trans has consecutively repeating chars, calculate cyclic shifts (suffix tree) in quadratic time, sort shifts, L[] of sorted is transform, compression exploits repeated substring, predictable (recently used letters), clumps. Invert: counting sort on L[] to get F[] in O(N), walk around cycle in O(N) time.
<u>Reverse</u>: sort transform, start at % in L[] and alternate pink $\rightarrow$ black, add letter from F[]
Manber's Algorithm: output sorted cyclic shifts in O(NlgN) time, find rank through<= lgN phase 1, 2, 4, 8,… until all distinct ranks
Min Cut: edge-weight digraph (source/target). Find cut of min capacity(sum of edges) b/t disjoint sets with s in one, t in other
Max Flow: ", assign edges so 0 <= flow <= capacity, inflow=outflow, value of flow = inflow at t, find flow w max value
Ford-Fulkerson: find/add augmenting paths w BFS (undirected path from s-t, ↑ forward if not full, ↓ back if not empty). #augmentations <= max flow (dumb in choosing paths), flow-value lemma: net flow across cut is flow from A-B minus B-A, always terminates (capacities are ints). Weak duality: flow value = net flow across cut <= capacity of cut.
Bipartite Matching: N students, N jobs, mult offers per student, perfect matching? Edge from s to each student and from each job to t (capacity 1), edge from student to job offered (capacity ∞), find max flow with FF, if no perfect matching, mincut explains why

Augmenting path	**numPaths**	**Implementation**
Random path	<= EU	Random queue
DFS path	<= EU	Stack (DFS)
Shortest path (BFS)	<= 0.5 EU	Queue (BFS)
Fattest path	<= Eln(EU)	PQ

Linear Programming: allocate scarce resources among competing activities, feasible region is convex (2pts inside, then line inside), optimal solution at extreme point, N vars, M eqs. Simplex: walk around corners, BFS the extreme points, step until not better, typically M+N steps, usually fast, wc exponential. Ellipsoid: wc polytime, slow, depends on #bits in input. Interior pt: wc polytime, can be better
Input: MxN matrix, v vectors b∈R^M and c∈R^N. Goal: find vector x∈R^N such that x >= 0, Ax <= b, c•x is maximized.

MAX = max{c•x: Ax <= b, x >= 0} x∈Rn. *MIN = min{b•y:* A^T*y >= c, y>= 0} y∈Rm.*
**weak duality : MAX <= MIN **strong duality: MAX == MIN
Duality: maxflow/mincut, brewmax/brewmin, shortest path (max distance at t, move far away. Mincost unit flow s-t)
*bipartite matching reduces to maxflow which reduces to LP **variable in one, constraint in the other
Dynamic Programming: store solutions of subproblems
<u>LCS</u>: springtime + pioneer = pine. **Brute force:** check every subsequence in x, see if in y O(N2^M) time. **DP:** find length of LCS of prefix of x and y of len i = c[i, j]. eventually want c[m, n]. O(1) time per subproblem. O(MN) to compute table. (longest path-diag). backtrack for realy string in O(N+M) time, diag tells sequence
if (i+1 and j+1 match) { c[i+1, j+1] = c[i, j] + 1 }, else { c[i+1, j+1] = max(c[i, j+1], c[i+1, j]) }
<u>TSP</u>: N cities, NxN distance matrix, find cyclic permutation of cities that minimizes total length. **Brute force:** consider all N! permutations. **DP:** Held-Karp Alg. Subset of cities S, compute d(S, j) = min d(S-{j}, k) + d(k, j) where k∈S-{1, j}. end: min d({1,…,N}, k) + d(N, 1). O(N) per subproblem, <= O(N2^N) probs. **Metric:** distances are metric space w/ triangle inequality, compute MST, find short cut w DFS, tour length <=2*MST tree, **Cristofides:** MST and matching, tour length <= 1.5*optimal. <u>2-opt improvement</u>: break tour in 2 places, reconnect, if it improves tour, keep move. Local improvement heuristics are fast, no guarantee they work. **Cutting Plane:** LP prob, keep adding constraints until optimal solution is an int, DFJ method, edge is in or out of tour, O(N^2) vars for in/out. Minimize c•x over all x∈S (exponential # of tours) such that Ax <= b, look for linear constraints of all tours, if have all constracts get optimal. Exact method. **Branch and Bound:** pruned exponential, find rules when to end early (good lower bounds), step at a time, good heuristics to pick early steps, exact method **PTAS: Polynomial Time Approx Scheme
<u>Knapsack</u>: max value of subset of items so total weight <= max weight of sack, compute max of subsets v(i, w) = max(v(i-1, w-w_i) + v_i, v(i-1, w)). O(1) time per subprblem, O(NW) probs, still exponentional bc W is big, recover items by backtracking in O(N) time
Amortized Time Bounds: AS IF each op were reasonal. <u>WQU w PC</u>: each op lg*N, O(N+Mlg*N). <u>Fib Heap</u>: delMin in O(lgN) amortized time, increase/decKey in O(1) amortized. <u>ArrayDoubling</u>: doubling is O(C) time and 2C acc but happens rarely. One doubling paid for by prvious C/2 pushes, avg 5 acc per push. <u>UF w PC</u>: Kruskal, O(N+MlgN) time, as if op = O(lgN) time, wc O(N) per op (long path). Potential Function=depends on state of data structure (bank acct). Φ = Σ lg(size of subtree rooted at i).
Amortized(o) = time(o) + c($\Phi_{\text{before o}}$ - $\Phi_{\text{after o}}$). Φ = 0, Φ >= 0, $\sum_{i=1}^{m} amortized(o) \geq \sum_{i=1}^{m} time(o_i)$ not true term by term, only in summation. o is union, amort = 3+3lgN = lgN. o is find, amort <= 1 + lgN
minVC: while edges not covered, put both endpts in Cover, remove edges touched by those endpts, polytime