**Burrows-Wheeler Transform:** transform string into another string, if og has repeated substring, trans has consecutively repeating chars

* Calculate cyclic shifts (like suffix tree)
* Sort cyclic shift with end char first

BANANA% %BANANA

ANANA%B A%BANAN

NANA%BA ANA%BAN ANNB%AA (column L) is transform

ANA%BAN ANANA%B

NA%BANA BANANA%

A%BANAN NA%BANA

%BANANA NANA%BA

* Compute shifts in quadratic time
* Invertible, can recover original string
* Compression: exploits repeated substring, predictable (recently used letters), clumps
* To invert: counting sort on L[] to get F[] in O(N), find links to T[] “where did it go” walk around to get og string backward in O(N) time
* Manber’s Algorithm: O(NlgN) time
  + Output sorted cyclic shifts
  + d=1, rank = # of other substrings of length d that are less than it
  + while ranks not all distinct, double d and compute rank[i] = rank of S[i] for substrings of length d (use LSD sort). Sorted col based on char sort
  + one phase (LSD sort) in O(N) time, at most lgN phases

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | S | Rank d=1 |  | d=2 | Rank |  | d=4 | Rank | Sorted |
| 0 | B | 4 | BA | (4, 1) | 4 | BANA | (4, 5) | 4 | 6 |
| 1 | A | 1 | AN | (1, 5) | 2 | ANAN | (2, 2) | 3 | 5 |
| 2 | N | 5 | NA | (5, 1) | 5 | NANA | (5, 5) | 6 | 3 |
| 3 | A | 1 | AN | (1, 5) | 2 | ANA% | (2, 1) | 2 | 1 |
| 4 | N | 5 | NA | (5, 1) | 5 | NA%B | (5, 0) | 5 | 0 |
| 5 | A | 1 | A% | (1, 0) | 1 | A%BA | (1, 4) | 1 | 4 |
| 6 | $ | 0 | %B | (0, 4) | 0 | %BAN | (0, 2) | 0 | 2 |

**Min Cut:** edge-weighted digraph with source and target vert.

* st-cut: partition of verts into 2 disjoint sets; s in one, t in other
* capacity = sum of capacities of edges b/t sets
* goal: find cut of minimum capacity from s to t

**Max Flow:** edge-weighted digraph with source and target vert.

* st-flow: assign values to edges such that 0 <= flow <= capacity
* local equilibrium: inflow = outflow at every vert
* value of flow: inflow at t
* goal: find flow of max value

**Ford-Fulkerson:** start with 0 flow, keep looking for augmenting paths

* augmenting path: undirected path from s to t: can increase on forward edges if not full, can decrease on backward edge if not empty
* when all paths blocked by full forward edge or empty back edge, found max flow
* net flow across a cut is sum of flows from A to B minus flows from B to A
* flow-value lemma: net flow across any cut = any flow (conservation of flow)
* weak duality: value of flow = net flow across cut <= capacity of cut
* always terminates (edge capacities are ints), if terminates, always computes maxflow
* find augmenting path with BFS
* # augmentations <= value of max flow (could be dumb in choosing paths)

|  |  |  |
| --- | --- | --- |
| **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 |

**Bipartite Matching:** N students apply for N jobs, each student gets multiple offers, how to match all students to a job

* Vert per student, vert per job
* Edge from s to each student (capacity 1)
* Edge from each job to t (capacity 1)
* Edge from student to job = offer (capacity infinite)
* Find max flow with FF
* If no perfect matching exists, mincut explains why (too many students, too many jobs?)

**Linear Programming:** reducing bipartite matching to max flow

* Problem solving model to allocate scarce resources among competing activites
* Ex: resources: corn, hops, malt (limited amts of each)
  + Can make ale or beer (diff recipes, diff profit)
  + How much ale/beer should you make?
  + Corn: 5A + 15B <= 480 (only 480 corns)
* Graph lines to find feasible region (convex: if 2 pts in region then line in region)
* Farthest from origin diagonally = highest profit
* Optimal solution at extreme point (2 constraints intersect)
* N vars, M equations
* Algorithms:
  + Simplex: walk around corners, BFS the extreme points, step until it doesn’t get better, typically M+N steps needed, usually fast, w.c. is exponential
  + Ellipsoid: w.c. polytime, slow depends on #bits in input
  + Interior point: w.c. polytime but sometimes better
* Input: MxN matrix A, vectors b∈RM and c∈RN
* Goal: find vector x∈RN such that x >= 0, Ax <= b, c•x is maximized
* MAX = max{c•x: Ax <= b, x >= 0} x∈Rn
* MIN = min{b•y: ATy >= c, y>= 0} y∈Rm
* Variable in one corresponds to constraint in another
* Weak duality: MAX <= MIN
  + c•x <= b•y
* Strong duality: MAX == MIN
  + c•x == b•y

**Dynamic Programming:** store solutions of subproblems, each can be solved with prev solutions

* LCS: springtime and pioneer = pine
  + Brute force: check every subsequence in x and see if it y O(N2M) time
  + DP: for each prefix of x of length i and each prefix of y of length j, find LENGTH of LCS of 2 prefixes = c[i, j]… eventually want c[m, n]. O(1) time per subproblem, O(MN) to compute table

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]) }

* + - i.e. longest path in DAG where diag is 1 and all other sides are 0 (if match, use diag path)
    - backtrack to find actual string in O(N+M) time, diag tells you sequence

**Traveling Salesman:** NP-complete

* N cities, given NxN distance matrix; Dij = length of edge from i 🡪 j
* Goal: find cyclic permutation of cities {Π1, Π2, …, ΠN} that minimizes total length
* Brute force: consider all N! permutations
* DP: O(2^N • N^2) time i.e. Held-Karp algorithm
  + Pick subset of cities S where 1∈S, j∈S and 1!=j
  + Compute d(S, j) = shortest path from 1 to j visiting all verts in S
  + Base case: S = {1, j} d(S, j) = D1j
  + For S >= 3: d(S, j) = min d(S-{j}, k) + Dkj where k∈S-{1, j}
  + O(N) time per subproblem, at most O(N2^N) subproblems
  + Once compute all d(S, j), get length of shortest tour by min d({1, …, N}, k) + DN1 where k!=1 (backtrack in O(N^2) time)
* Approximations: find tour length 1.05 \* optimal
* Metric TSP: distances are a metric space
  + Triangle inequality: dij + djk  >= dik
  + Compute MST, find short cut with DFS traversal (add vertex to tour first time you see it)
  + Tour length <= 2\*(size of MST tree)
  + Cristofides method (MST and matching): tour length <= 1.5 \* optimal
  + Lower bound: factor of 113/112
  + 2-opt improvement: break tour in 2 places, reconnect in different ways, if it improves tour, keep move
  + Local improvement heuristics are fast but no guarantee that it works
* Cutting Plane method: LP problem, keep adding constraints until optimal solution is int
  + DFJ method: tour is a set of edges either in tour = 1 or not in tour = 0
  + O(N^2) vars for 1/0 inTour
  + S is a set of x’s that describes a tour and c{i, j} is cost of edge from i to j
  + Minimize c•x over all x∈S (exponential # of tours) such that Ax <= b
  + Look for linear constraints of all tours, add to A and solve LP
  + If have ALL constraints, get optimal tour
    - ex: each vert in a tour has 2 edges so get rid of tours where each vertex does not have 2 edges
  + exact method
* Branch and Bound: pruned exponential search
  + Find rules to know when to cut off search early
  + Pick optimal tour one step at a time
  + Need good heuristics to pick early steps
  + Good lower bounds on remaining tour (held-karper lower bound)
  + In practice: concorde solver
  + Exact method (committed to possibly doing exponential)
* Planar and Euclidean cases: polynomial time
  + EPTAS: exploit separations of nice metric spaces
  + For every epsilon > 0, there is a polytime algorithm that finds solution with cost <= (1 + epsilon)\*optimal

**Knapsack:** NP-complete

* N items with value vi and weight wi, find max value of a subset of the items so total weight <= W (max weight of sack)
* Compute value of max subset, backtrack to find items
* Subproblem: for each item in {i0, …, iN} and for each weight in {w0, …, w­N}, find max value of subset {1, …, i} with w <= W. max value of subset is v(i, w). want v(N, W)
  + Base case: i=0, no items allowed to v(i, w) = 0
  + i >= 1: v(i, w) = max( v(i-1, w-wi) + vi, v(i-1, w) )
    - use item i if wi <= w(remaining weight), find max of using i and not
  + O(1) time per subproblem, O(NW) subproblems to O(NW) total time/space
* Recover items by backtracking in O(N) time

**Amortized Time Bounds:** sequence of operations has reasonable time bound

* UF with PC and weights: O(N+Mlg\*N) nearly constant, AS IF each op were lg\*N
* Fibonacci heap: delMin in O(lgN) amortized time, insert/decreaseKey in (1) amortized
* Array Doubling/Halving: for stacks, queues, hashtables, etc
  + No advertised max size. Capacity C and public size S where S <= C
  + Doubling takes O(C) time and 2C array accesses but happens RARELY
  + One doubling op paid for by previous C/2 pushes that didn’t need doubling
  + Avg cost per push <= 5 array accesses
  + Array halving: if S < C/4, C=C/2 safer than S < C/2 bc can oscillate a lot
* UF with PC: fast enough for Kruskal’s
  + M ops on N items, O(N+MlgN) time, as if each op = O(lgN) time
  + w.c. is O(N) per op (path may be long, but only traverse once)
  + let o be a UF op. P(o) = # times that o modifies p array. Time for o is 1 + P(o)
  + Potential Function: depends on state of data structure, bank acct, can save time to be spent later. Φ = 0 initially bc all leaves, Φ >0 zero always
  + Φ = Σ lg(size of subtree rooted at i)
  + Amortized time for o = true time for o + change in Φ
  + Amortized(o) = time(o) + c(Φbefore o - Φafter o)
  + Φ initial is 0 so
    - Not true term by term, only in summation
* If o = find(x), amortized(o) <= 1 + lgN
  + Look at how size of subtree increases as go from x to root
  + Edge is gold if points to root, red if size doubles, blue otherwise
  + # red edges <= lgN bc doubling
  + # red + # blue = # edges that need to change
  + Compressing path, blue edges pay for themselves by decreasing Φ, only pay for red edges
* If o = union(p, q), amortized(o) <= lgN
  + Starts with 2 finds <= 2 + 2lgN
  + Make link between if roots not same, if make: time increases by 1 bc modify P
    - Size at new root increases by at most lgN

**Min Vertex Cover:** can approximate by factor of 2

* While there are edges not yet covered, put both endpoints (u, v) in Cover, remove edges touched by endpoints u and v
  + Polynomial time
* Selected edges are “maximal” matching, not maximum
* # edges <= minVC <= 2 \* numEdges