**Marriage Problem** = Stable if no pair not matched together would prefer to be matched Gale-Shapley Propose-And-Reject Algorithm: finds **one** of possibly many stable matchings in O(n2) if efficient but could be O(n3) if naive. Man is a valid partner of woman if a stable matching exists where p + q matched. // maintain list of free men in a queue = O(1) naïve O(n) **C** // maintain array count[m] that counts - proposals made by m and use pointer to – point at last proposal = O(1) naïve O(n)// if else then O(1) if careful by using arrays that show women are engaged to what man, and what men are engaged to what women// maintain 2 arrays wife[m] and husband[w] initiated to 0 **– if** m matched to w then wife[m] = w and husband[w] = m// create inverse preference index for each woman (take linear time to build this preprocessed data structure using for i = 1 to n in Pref inverse[pref[i]] = i) = then we have constant time O(1) naïve O(n) // look at two values in inverse and compare

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Woman | 1st | 2nd | 3rd | 4th | | 5th | 6th | 7th | 8th | |
| Pref | 8 | 3 | 7 | 1 | | 4 | 5 | 6 | 2 | |
| Woman | 1 | 2 | 3 | | 4 | 5 | 6 | 7 | | 8 |
| Inverse | 4th | 8th | 2nd | | 5th | 6th | 7th | 3rd | | 1st |

In arbitrary matching max # of unstable pairs that can exists = n2 – n

Maximum number of matches possible = |men| \* |women| w/some unstable

**Matching Residents to Hospitals** – runs like “limited polygamy” Stable Matching but with multiple slots and potential for non-matches = Stable if there is no pair not matched together would prefer to be matched, Unstable if: 1) h and r like each other; + 2) either r is unmatched or r prefers h to her assigned h; and 3) either h does not have all its places filled, or h prefers r to an assigned r.

**Roommate Problem – Irving’s Algorithm:** UNSTABLE if any unmatched pair prefer each other as roommates over their current partner + no stable match if a roommate rejected by all other roommates: square = proposal ⃝ = accept X = reject

PHASE 1 O(n2): Run through proposals + acceptances and cross-cancelling rejections until each roommate ends up with both a proposal + acceptance PHASE 2 O(n): reject worst choices by crossing out all options to the right of roommates’ acceptance and removing them and any proposal and acceptance notation so all you have is potential matches let on diagram PHASE 3 O(n2): a) choose first choice with > 1 potential match and put above a line and put its 2nd best choice below the line; b) above the line place the worst choice of the option last placed below the line until a cycle is identified; c) cross cancel diagonally above and below line and symmetrically notate on diagram; d) continue this until single matches

Bipartite Matching – O(nk) = 2 partitioned sets w/no edges between sets (like matching test question): 1) Solution is no 2 edges share endpoint 1:1 matching; 2) Solved w/max flows reduction methodology

Indy Set = find max node subset w/no edges adjacent. NP-complete problem = exp

Competitive Facility Location = 2 competing players alternate in selecting nodes and cannot select a node if any of its neighbors have been selected – goal is to pick subset with maximum weight: 1) Input is a graph with a weight for each node – in a linear format; 2) PSPACE-complete unsolvable

**Interval Scheduling Problem** = greedy = O(n log n) - Weighted version requires dynamic programming O(n log n): 1) Input = set of jobs w/start/finish times; 2) Goal is to find max subset of mutually compatible jobs nonoverlapping in time w/|number of jobs| priority over utilization of resource (sometimes = same)

Earliest Start Time = consider jobs in ascending order of start time sj

Shortest Interval = consider jobs in ascending order of interval length

Fewest Conflicts = count # of conflicting jobs cj and schedule in ascending order

Earliest Finish Time = Only of these 4 algorithms that survives optimality: Sort jobs by finish time O(n log n) into queue or array A = jobs chosen set = > For j = 1 to n O(n) {if (job j compatible with A) => A <- A U { j } } return A. Proof is contra/substitut argue

**Interval Partitioning** = diff from Interval Scheduling - O(n log n) if you keep resources in priority queue = instead of trying to run the max # jobs, must find min resources to run all jobs. Depth of a set of open intervals is the max # that contain any given time, i.e. the max # of intervals that overlap at the same time. # of resources needed ≥ depth 1) Sort intervals by starting time so the s1 ≤ s2 ≤ s3 . . . , ≤ sn O(n log n) 2) d <- 0 for j = 1 to n { O(n) but + log n each iteration 3) if (lecture j is compatible with some classroom k) schedule lecture j in classroom k => Else => allocate a new classroom d + 1 => schedule lecture j in classroom d + 1 => d <- d + 1 } // O(1) if set of jobs in queue e.g. maintain finish time of last job added BUT you need to save earliest, latest finish time in additional PQ that is a min heap which takes log n to update each loop

**Scheduling to Minimize Lateness** = schedule all jobs to minimize lateness total of all jobs. 1) Shortest processing time first = consider jobs in ascending order of tj; 2) Smallest Slack = consider jobs in order of slack which = dj – tj; 3) Earliest deadline first = consider jobs in ascending order of deadline dj = O(n log n) 1) Sort n jobs by deadline so that d1 ≤ d2 ≤ . . . ≤ dn O(n log n) 2) timeline <- 0 3) for j = 1 to n O(n) \* O(1) { 4) assign job j to interval [t, t + tj] 5) sj <- t 6) fj <- t = tj 7) output intervals [sj, fj] } Inversions = in schedule S an inversion is a pair of jobs i and j such that the **deadline** of i < the deadline if j, but j is scheduled before i – An optimal schedule can have gaps, but here exists an optimal schedule w/no idle time as idle time cannot help reduce max lateness and a Greedy schedule has no inversions and no idle time

**Optimal Offline Caching** - Farthest-In-Future = Optimal **offline** caching per Bélády in the 1960s and serves as a guideline for optimal, BUT **only works in Offline case where you know the sequence of up-coming requests O(n)** – proof by induction showing that for any given request a special property holds in all cases

Reduced Eviction Schedules = a schedule that only inserts an item into the cache in a step in which that item is requested – given unreduced sched S, can transform it into a reduced S’ w/no more cache misses than were in S. PROOF: 1) d already in cache; 2) S and Sff evict same element; 3) S and Sff have same contents but evict different

LRU (Least Recently Used) = evict page whose most recent access was earliestis ff with direction of time reversed and is k-competitive

**Dijkstra’s** – **finds shortest path** to every node in graph CANNOT take negative edge weights (might work/might not, Bellman-Ford instead): Upper bound depends on implementation w/binary heap = O(m log n) and Fibonacci Heap = O(m + n log n)

1) assign every node a distance value d(u) from s to u; 2) Initialize S = { s }, d(s) = 0;

3) consider all unvisited neighbors to current node + calculate their distance from s and if < current distance replace w/new value; 4) when all neighbors considered mark current as visited and if u marked visited done 6) set unvisited node marked w/smallest tentative distance as next current node + go back to #3

**Minimum Spanning Tree** –Cayley’s Theorem = there are nn-2 spanning trees of Kn (no poly time algo to prove this/brute force not best approach) Direct Apps: network design, approx algos for NP-hard problems such as travelling salesperson + Steiner tree. Indirect apps: max bottleneck, Low Density Parity Check (LDPC) code for error correction, image restoration w/Renyi entropy, learning features for real-time face verification, reducing storage in sequencing amino acids, model interactions in turbulent fluid flows, autoconfig protocol for Ethernet bridging to avoid net cycles

Highest weight edge in a cycle is not included in any MST for graph. Definition of a “complete” or “connected“ graph is w/all possible edges connecting nodes.

**Cut Property:** if 2 subsets min cost edge that is in both subsets is in MST. **Cycle property:** max cost edge of a cycle is NOT in MST. **Cutset:** subset of edges that have exactly one endpoint in S, i.e. connect cutset to set and if all cut = 2 sets

**Kruskal’s** = O(E log E) w/binary heap O(E + V log V) w/Fibonacci if |V| < |E| CAN take neg edges: 1) Sort edges by weight 2) choose edge w/lowest weight 3) repeatedly add lowest edge weight that does not create a cycle

**Reverse-Delete Algorithm** = Reverse of Kruskal = start with ALL the edges in and cut edges due to cycle property and keep edges due to cut property

**Prim’s** (like Dijkstra’s and BFS) O(n2) with an array and O(m log n) with a binary heap – **CAN** take neg edge weights 1) start w/root node s and greedily grow a single tree T outward 2) assign s value 0 and other nodes ∞ 3) At each step apply cut property to set S of explored nodes and add the cheapest edge e to T 4) For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge from v to a node in S

**Clustering** = given a set U of n objects labeled p1, p2, . . . , pn, classify into coherent groups Distance Function = numeric value specifying “closeness” of two objects. Used in: routing in mobile ad hoc networks, IDing patterns in gene expression, document categorization for web or medical search, Skycat = cluster 109 sky objects into stars, quasars, galaxies K-clustering = divide objects into k non-empty groups. Distance function – satisfies several props: 1) ID of indiscernibles: d(pi, pj) = 0 iff pi = pj;

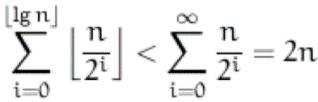
2) nonnegativity: d(pi, pj) ≥ 0; 3) symmetry: d(pi, pj) = d(pj, pi)

Spacing = min distance btwn any pair of points in diff clusters = found by choosing the pair that has the shortest distance (as part of k-clustering algorithm you do find all these pairs and their distances). Clustering of max spacing = given an integer k, find a k-clustering of max spacing

Single-link k-clustering algorithm: Kruskal’s but stop when n – k edges inserted into spanning tree, *i.e*. stop when n – k edges have been inserted into the set: k-cluster is formed by taking MST and deleting k – 1 weightiest edges. Similarly, if you have a k-clustering and you want to change it to a k – 1 clustering, simply add the min weight edge across the 3 components which turns 2 closest together clusters into one cluster

You can mix-match induction + exchange proofs

**Binary Counter**: INCREMENT function = Θ(n log n) in the worst case – BUT only happens when a lot of the bits are 1 – so Amortized Analysis = 2n using this formula:

i from 0-logn = number of binary digits for decimal value and 1/2i would be single increment, so n/2i = all, then take sum to ∞ use rule n\*Σ(1/2)i = 1/[1-(1/2)] = 2n get 2n (ternary sub in 3 for 2 = 3/2n)

Aggregate Method = average case for binary increment = 2n (from formula)/n = 2

Accounting Method = overpay for certain op to save for another kind of op. Amortize cost of binary INCREMENT is the total charge it incurs which is exactly $2 since single change of 0 to 1 = cost of 2 and all 1 to 0 changes already paid for via $2 charge

Potential Method = 0 potential energy then ops build or use energy as a function of the entire Data Structure – takes number of previous calls as a parameter , i.e. it needs φi-1 The amortized cost of the ith operation denoted ai is defined as the actual cost plus the change in potential: ai = ci + φi = φi-1 or the Amortized cost = actual cost + Δφ where Δφ = φ(after operation) – φ(before operation). Overall:Σamortized costs = Σactual costs + φ(end) – φ(beginning) NOTE You have to pay φ(beginning) so that gets moved to the left side so we end up with Σamortized costs + φ(beginning) = Σactual costs + φ(end). Stack example: ci = 1 and s = size of Stack so

Push => ai = 1 + (s + 1) – s = 2 because s cancels and 1 + 1 = 2

Pop => ai = 1 + (s – 1) – s = 0 because s cancels and 1 – 1 = 2

Multi-pop => ai = k’ + (s – k’) – s = 0 because both k’ and s cancels

k’ = cost of multipop (s – k’) – s = Δφ which is stack diff which = cost of multipop

Binary Count example: ci = #bits from 0 to 1 + #bits 1 to 0 = > φi = φi-1 = #bits 0 to 1 minus #bits 1 to 0, thus ai = ci + φi = φi-1 = 2 \* #bits changed from 0 to 1