COMP2007 Notes

TRY BRUTE FORCE FIRST TO UNDERSTAND THE PROBLEM THEN TRY REDUCTION.

Definitions

- **Worst-case running time:** Obtain bound on largest possible running time of algorithm on input of a given size N.
- **Average case running time:** Obtain bound on running time of algorithm on random input as a function of input size N.
- Algorithms are **efficient** if its running time is polynomial.
- **Lower bound:** time complexity is not lower than this bound. Vice versa upper bound. **Tight bound** is it is both a lower and upper bound. Bounds have transitive and additive properties.
- **Polynomial time:** $O(n^d)\ for\ some\ constant\ d\ indepedent\ on\ input\ size\ n.$
- **Logs generally grow slower than all polynomials.**
- **Linear time:** Running time is at most a constant factor times the size of the input.

Graphs

- **Adjacency matrix:** n-by-n matrix with $A_{uv} = 1\ if\ (u, v)\ is\ an\ edge.$
  - Time: O(1) check edge.
  - Space: O(n^2), n by n matrix.
- **Adjacency list:** Node indexed array of lists.
  - That is, you have an array, and if u have a vertex 1 connected to 2 and 3. You have at index 1 in the array, a linked list of 2 and 3.
  - Saves space O(m+n) and identifying all edges is O(m+n); no wasted space. '0s' are gone.
- **Simple path:** sequence of paired nodes connected by edges (path) and all nodes are distinct (simple).
- **Connected graph:** All nodes have paths to each other (graph must be undirected).
- **Trees:** An undirected graph that is connected and acyclic.
  - **Property:** n-1 edges.
  - **Rooted:** Pick a root node and orient edges away from root; **hierarchical**.
- **s-t connectivity problem:** Given two nodes s and t, is there a path between s and t.
- **s-t shortest path problem:** given two nodes s and t, what is the shortest path between s and t.
- **Breath-first-search:** Layer by layer. Layers are determined (intuition): pick a root, 1 edge away from root is first layer, 2 edges is second and so on.
  - **Theorem:** For each i, L_i consists of all nodes at distance i from s, iff a path from s to t appears in some layer.
  - **Running time:** $O(V + E)$: Each vertex is en-queued and de-queued at most once O(V), and scanning all adjacent vertices takes O(E).
  - *Using either Adjacency-matrix or list. Space complexity will differ however.*
  - *Uses a queue.*
- **Connected component**: All nodes reachable from s.
  - **Theorem:** Upon termination, R is the connected component containing s.
- **Shortest PathS:** Compute the shortest path from a given node to all other nodes.
  - **BFS:** Computes the 'hop' distance from s to u.

        - Initialise all dist[u] as infinity in BFS. Dist[s] = 0.
- **Transitive closure:** Given a graph G, compute G' such that:
    - Has the same vertices
        - with an edge between all pairs of nodes that are connected by a path in G.
    - Use **BFS** but modify by adding an edge to (s,v) after we have '**seen**' it.
- **Depth first search**: Pick a starting vertex, s, and following outgoing edges that lead to 'undiscovered' vertices and backtrack (pop the stack) whenever 'stuck'.
    - Assume that G is connected.
    - Uses a stack.
    - **Running time:** $O(V + E)$
    - Grows a forest if the graph is not connected; each tree will be a connected component.
- **Bipartite Graphs:** An undirected graph G = (V,E) is bipartite if the nodes can be coloured red or blue such that every edge has at least one blue and red end.
    - Formally, we can divide V into two subsets where E is connected between the two subsets and not within themselves.
    - **Lemma:** If graph G is bipartite, it cannot contain an odd length cycle. **Proof:** Can be shown through a diagram.
        - Can be used to create a poly-time certifier that runs in $O(m + n)$.
    - **Lemma 2:** Let G be a connected graph and let $L_0, \dots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds:
        - **No edge** of G joins two nodes of the same layer
            - G is bipartite
        - **An edge joins two nodes** in the same layer such that (meaning that) G contains an odd length cycle.
            - G is not bipartite.
- **Cut edge:** In a connected graph, the removal of a 'cut edge' would disconnected the graph.
    - G = (V,E) is connected
    - G' = (V,E\{e}) is not connected.
    - Algorithm:
        - Run DFS on graph
        - For each edge in the DFS tree
            - Remove that edge from graph G.
            - Check if G is now disconnected (using DFS).
        - **Running time:** $O(mn)$.
    - Better algorithm: Test if edge (u,v), if there is a back edge from v, if there's not, then edge (u,v) is a cut edge. O(m+n).
- **Directed reachability:** In a DAG, find all nodes reachable from s.
    - **Directed s-**t shortest path problem
    - **Graph search:** BFS extends to directed graphs
    - **Web crawler:** start from web page s. Find all pages linked from s, either directly or indirectly.
- **Strongly connected:** Node u and v are mutually reachable if there is a path from u to v and also a path from v to u. i.e. every pair of nodes is mutually reachable. (Of a directed graph).
    - Lemma: G is strongly connected iff every node is reachable from s and s is reachable from every node. **Proof follows from definition.**
    - Running time: $O(m + n)$
    - Run BFS in G, and run BFS from s in $G_{reversed}$ return true iff true in both executions.

        - To consider disjoint SCC use DFS twice, once to compute finish[u], and then call in main loop in decreasing order. Output two forests and compare.
- **Topological order:** an ordering of nodes v, such that for every edge, (v_i, V_j) I < j.
    - **Lemma:** If G has a topological ordering then G is DAG.
        - **Contradiction:** we have I< j and j< I in a cycle. Cannot settle precedence.
    - Every DAG must have a topological ordering; there must be at least one node with no incoming edge; otherwise there would be a cycle.
    - Running time: $O(m+n)$. Maintain a counter for each edge. If it hits 0, remove v, and update all counts.
    -

<u>Greedy Algorithms</u>

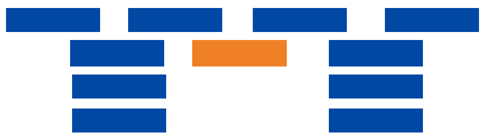
- **Definition:** A **greedy algorithm** is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

    1. **Define your solutions.** You will be comparing your greedy solution X to an optimal solution X_{opt}, so it's best to define these variables explicitly.

    2. **Compare solutions.** Next, show that if $X \neq X_{opt}$, then they must differ in some specific way. This could mean that there's a piece of X that's not in X_{opt}, or that two elements of X that are in a different order in X_{opt}, etc. You might want to give those pieces names.

    3. **Exchange Pieces.** Show how to transform X_{opt} by exchanging some piece of X_{opt} for some piece of X. You'll typically use the piece you described in the previous step. Then, prove that by doing so, you did not increase the cost of X_{opt} and you therefore have a different optimal solution.

    4. **Iterate.** Argue that you have decreased the number of differences between X and X_{opt} by performing the exchange, and that by iterating this process you can turn X_{opt} into X without impacting the quality of the solution. Therefore, X must be optimal. This last step might require a formal argument using an induction proof. However, in most cases this is not needed.
-
- **Interval scheduling**:
    - **Input**: Set of n jobs. $Each\ job\ i\ starts\ at\ s_i\ and\ finishes\ at\ f_i$.
    - Two jobs are compatible if they don't overlap in time.
    - **Goal**: find the maximum cardinality of a subset of compatible jobs.
    - **Templates:**
        - **Earliest start time:** breaks if there is one 'long' block that eclipses small intervals, where the first interval starts later.
        - **Shortest interval:** Consider two long intervals, and one short interval that is eclipsed by both intervals on either side. Optimal would be to ignore shortest.
        - **Fewest conflicts:**



    - **Earliest finish time greedy algorithm:** Consider jobs in increasing order of finish time, take each job provided it is compatible with the ones already taken.
        - **Running time:** $O(nlogn)$
        - **Compatible:** Job I is compatible if $s_i \geq f_i^*$ where * denotes that it was the last job added.
        - **Proof:** (by contradiction):

            - Assume greedy is not optimal.
            - Let $i_1, i_2 \ldots, i_k$ denote the set of jobs selected by greedy.
            - Let $j_1, j_2, \ldots, j_m$ denote the set of jobs in the optimal solution.
            - For this we only need to prove |A| = |O|, as proving A = O is too hard.
            - OPT would've just chosen a different interval in the same time frame however the overall intervals remains the same. Use induction.
                - Since $f(i_{r-1}) \leq f(j_r - 1) \leq s(j_r)$, Job j_r is available when the greedy algorithm makes its choice. Hence f(i_r) <= f(j_r).

- **Interval partitioning:** Given intervals (s_i, f_i) find the minimum number of 'bins' to schedule such that all the intervals in the bin are compatible.
    - Number of bins needed $\geq$ depth.
    - **Algorithm:** Sort the intervals by starting time, assign lectures to any compatible classroom or start a new classroom.
    - Maintain the finish time of the last job added.
        - Proof: d = number of classrooms that greedy allocates
        - Classroom d is opened because the job is incompatible with d-1 classrooms.
        - Since we sorted by start time, all incompatibilities caused by lectures start no later than s_i
        - Thus, we have d lectures overlapping at time $s_i + e$, where e a very small number.
        - Hence all schedules use $\geq$ d classrooms.
- **Schedule to minimise lateness:** Schedule all jobs to minimise maximum lateness. Lateness is caused by the fact that jobs have due times and only be processed linearly.
    - **Algorithm (Earliest deadline first):** sort by deadline and assign.
        - **Proof:** An inversion in a schedule is a pair (I,k) such that i<k (by deadline) but k is schedule before i. We can always swap inversions and it does not increase latness.
            - $l'_x = l_x \; for \; all \; x \neq i$
              $l'_k = f'_k - d_k$
              $= f_i - d_i$ (j finishes at time f_i).
              $\leq f_i - d_i \; (i < k) \leq l_i$

        - **Proof:** Define S* to be an optimal schedule that has the fewest number of inversions.
            - S* has no idle time.
            - If S* has no inversions, S = S* done.
            - If S* has an inversion, let i-k be an adjacent inversion
                - Swapping I and k does not increase maximum latness and strictly decreases the number of inversions
                - This contradicts definition of S*.
    - **Running time:** O(nlogn)
- **Minimum spanning tree (MST):** Given a connected Graph G=(V,E) with real-valued edge weights, an MST is a subset of the edges T, such that T is a spanning tree whose sum of edge weights are minimised.

    - **Cayley's theorem:** there are n^{n-2} *spanning trees of* K_n (*brute force nope*).
    - **Cut property:** Cut edges must be in the MST.
    - **Cycle property:** The maximum weighted edge in a cycle cannot be in the MST.
    - **Prim's algorithm (cut): (Grow a cloud from start node, Kruskal's is choosing from ordered weights).**
        - Initialise s = any node.
        - Apply cut property to S
        - Add min cost edge in cutset corresponding to S to T and add one new explored node u to S.
        - Keep adding the shortest length.
    - **Kruskal's algorithm (cycle):** Considering edges in ascending order of weight. If adding e creates a cycle, discard according to cycle property, otherwise insert e = (u,v) into T according to cut property where S = set of nodes in u's connected component.
    - **Tiebreaking:** add small perturbations to break tie-breakers, we can rid of the assumption that edge costs are distinct. Break ties according to index.
    - **Union-find data structure:** O(mlogn + m + nlogn).
- **Shortest Path (Djikstra's algorithm):**
    - Maintain a set of explored nodes for which we have determined the shortest path distance.
    - Repeatedly choose the unexplored node which minimises the path.
    - Proof via induction.

$$\ell(P) = \ell(P') + \ell(x,y) \underset{\substack{\uparrow \\ \text{inductive} \\ \text{hypothesis}}}{\geq} d(x) + \ell(x,y) \underset{\substack{\uparrow \\ \text{defn of } \pi(y)}}{\geq} \pi(y) \underset{\substack{\uparrow \\ \text{Dijkstra chose v} \\ \text{instead of y}}}{\geq} \pi(v)$$

    - **Invariant:** For each node, d(u) is the length of the shortest s-u path.
- **Clustering:** Given a set of U of n objects labelled $p_1, \dots, p_n$ classify into coherent groups.
- **Distance function:** Numeric value specifying "closeness" of two objects.
    - **Algorithm:** Form a graph on the vertex set V, corresponding to n clusters.
        - Find the closest pair of objects such that each item is in a different cluster, and add an edge between them.
        - Repeat n-k times until there are exactly k clustesrs.
        - Similar to finding an MST, and deleting k-1 expensive edges. Kruskal's.

Divide and conquer:

- Break up problems into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of the original problem.
2. Recursively solving these subproblems.
3. Appropriately combining (merging) their answers.

The real work is done in three different places: in the partitioning of problems into subproblems; when the subproblems are so small that they are solved outright; and in the gluing together of partial answers.

The standard way of proving correctness for a divide-and-conquer algorithm is by using induction as follows.

- Base case: Solve trivial instances directly, without recursing.
- Inductive step: Reduce the solution of a given instance to the solution of smaller instances, by recursing. For divide-and-conquer algorithms it usually requires a bit of work to prove that the step of merging two (or more) solutions to smaller problems into the solution for the larger problem.

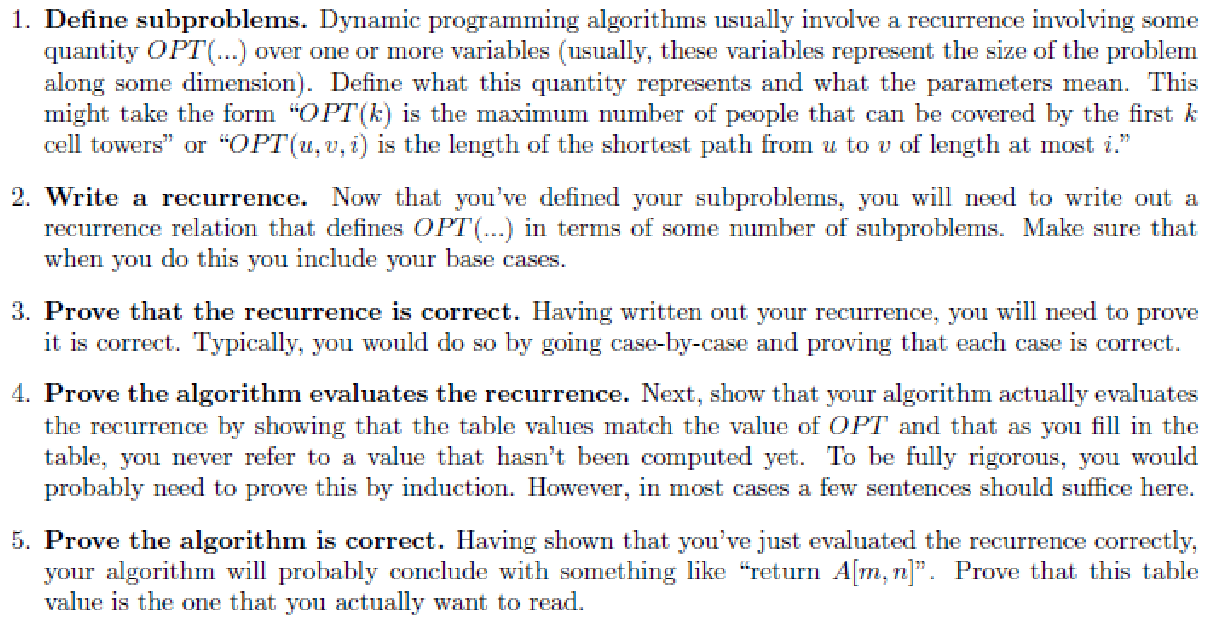
- **Binary search**
- **Mergesort**
- **Closest pair of points:**
  - Given n points in the plane, find a pair with smallest Euclidean distance between them.
  - Algorithm: (Assume no two points have same x coordinate).
    - **Divide**: Draw a vertical line L so that roughly $\frac{1}{2}n$ points on each side.
    - **Conquer:** find the closest pair in each side recursively.
    - **Combine:** find the closest pair with one point in each side.
    - Return best of 3 solutions.
    - We only need to consider the points that lie within a certain distance from the line.
      - Sort points in the 2*certain distance by their y-coordinate, check distances of those within the shorted list.
      - **Proof:** No two points lie in the same ½*certain distance by ½*certain distance box,
        - Two points at least 2 rows apart have distance >= 2*1/2(certain distance).
      - Running time: $O(nlog^2n)$
      - Can achieve $O(nlogn)$ if we have pre-sorted lists.
- **Master method:** Applies to recurrences that have the form below.
  - $T(n) = a * T\left(\frac{n}{b}\right) + f(n)$,
    where $a \geq 1, b \geq 1$ *and f is asymptotically positive*.
  - **Case 1:** If $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for any constant $\varepsilon > 0$ then $f(n)$ grows polynomially slower than $n^{\log_b a}$**.** Hence $T(n) = O(n^{\log_b a})$.
  - **Case 2:** If $f(n) = O\left(n^{\log_b a} \log^k n\right)$ for some constant $k \geq 0$ then $f(n)$ and $n^{\log_b a}$ grow at similar rates. Thus $T(n) = O(n^{\log_b a} \log^{k+1} n)$.
  - **Case 3:** If $f(n) = O\left(n^{\log_b a + \varepsilon}\right)$ *for some constant* $\varepsilon > 0$. In this case $f(n)$ dominates the recurrences and hence $T(n) = O\big(f(n)\big)$.
- **Quick and dirty: if n^logba = f(n), add a log(n), if less take f(n), if greater take n^logba.**
- **Sweepline technique (and computational geometry):** study of algorithms to solve problems stated in terms of geometry.

- **Depth of interval:** Given a set S of n intervals compute the depth of S is the maximum number of intervals passing over a point.
    - **Algorithm:** Sweepline sweeping from left to right while maintaining the current depth. **Data structure:** Binary search tree with event points.
    - **Event points:** Endpoints of the intervals.
    - Current depth is stored in the sweepline.
    - **Running time:** $O(nlogn)$.
- **Segment intersection:** Given randomly oriented intervals in space. Report all pairs of segments that intersect.
    - Simulate sweeping a vertical line from left to right across the plane.
    - **Events:** discrete points in "time" when sweep line status needs to be updated.
    - **Sweep line status:** store information along sweep line.
    - **Cleanliness property:** at any point in time, to the left of sweep line everything is clean, **i.e.** properly processed. (this is the invariant).
    - **Algorithm:**
        - Store segments in a blaanced binary search tree T.
        - Deleting a segment in T two segments become adjacent
        - When inserting a segment in T it becomes adjacement to two segments, we can find them in Ologn and check if they intersect
        - If we find an intersection, we are done.
        - O(nlogn). Decision version. O(nlogn + hlogn) for all segments.
- **Convex hulls:** a subset of the plane is convex if for every pair of points (p,q) in S the straight line segment pq is completely contained in S. The convex hull of a point set is the smallest convex set containing S.
    - **Divide and conquer approach:**
        - If S not empty then
        - Find farthest point C in S from AB
        - Add C to convex hull between A and B
        - S0 = {points inside ABC}
        - S1 = {points to the right of AC}
        - S2 = {points to right of CB}
        - FindHull(S1,A,C)
        - FindHull(S2, C,B)
    - **Sweepline approach:**
        - Maintain hull while adding the points one by one, from left to right ⇔ sweep the point from left to right.
        - O(nlogn)
- **Closest pair:** Use two parallel vertical sweep-lines: the front $L_f and\ back\ L_B$.
    - **Invariant**: The closest pair among the points to the left and the distance d between this pair.
    - **Data structure**: BST to store all points in S between the two sweeplines from top to bottom.
    - Once you reach an event point for both lines, calculate the distances.
    - Find the point s' closest to s inbetween $L_b\ and\ L_f$ within vertical distance d from s.
    - If |ss'| < d then
        - set d = |ss'|
        - CP = (s,s')
        - Sweep L_b and update T.

    - Insert s into T.
    - O(nlogn).
- **Visibility:** Let S be a set of n disjoint line segments in the plane, and let p be a point not on any line segment of S, determine all the line segments that p can see.
    - **Event points:** Endpoints of segments.
    - **Keep track of:** The segment that q sees in that direction, and the order of the segments along the ray.
    - **Invariant:** the segments seen so far, and the order of the segments intersecting the ray.
    - **Event handling:** Two cases: first endpoint, last endpoint.
        - **First endpoint:** insert new segment into D, if s is the first segment hit by ray, report s.
        - **Second endpoint:** Remove s from D, if s was the first segment in D, report new first segment in D.
    - **Complexity:** Number of endpoints: 2n, handle events log(n) (insert/delete from BST). = O(nlogn).
    - **Start sweep by sorting all segments intersecting starting ray.**
- **The median problem:** Given a sequence of n numbers, find the median in linear time.
- **The selection problem:** given an unsorted array A with n numbers, and a number k, find k-th smallest number in A.
    - **Algorithm to solving both:** Use divide and conquer; partition into 5 groups, recursively find the median. Running time: O(n).
    -

Dynamic programming

1) Define sub-problems: define what OPT(i) is, and what it does. And the invariant.
2) Find recurrences: name all the test cases and write the formula.
3) Solve the base cases
4) Transform recurrence into an efficient algorithm.



1. **Define subproblems.** Dynamic programming algorithms usually involve a recurrence involving some quantity $OPT(...)$ over one or more variables (usually, these variables represent the size of the problem along some dimension). Define what this quantity represents and what the parameters mean. This might take the form "$OPT(k)$ is the maximum number of people that can be covered by the first k cell towers" or "$OPT(u, v, i)$ is the length of the shortest path from u to v of length at most i."

2. **Write a recurrence.** Now that you've defined your subproblems, you will need to write out a recurrence relation that defines $OPT(...)$ in terms of some number of subproblems. Make sure that when you do this you include your base cases.

3. **Prove that the recurrence is correct.** Having written out your recurrence, you will need to prove it is correct. Typically, you would do so by going case-by-case and proving that each case is correct.

4. **Prove the algorithm evaluates the recurrence.** Next, show that your algorithm actually evaluates the recurrence by showing that the table values match the value of OPT and that as you fill in the table, you never refer to a value that hasn't been computed yet. To be fully rigorous, you would probably need to prove this by induction. However, in most cases a few sentences should suffice here.

5. **Prove the algorithm is correct.** Having shown that you've just evaluated the recurrence correctly, your algorithm will probably conclude with something like "return $A[m, n]$". Prove that this table value is the one that you actually want to read.

- **Definition:** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
- **Weighted Interval Scheduling:**
  - Goal: find maximum weight subset of mutually compatible jobs.
  - Greedy algorithm fails when weights are allowed. One with 999 weight and one with 1 weight.
  - Sort and then Instead define p(j) = largest index I < j such that I is compatible with j.
  - **Algorithm:**
    - OPT select job j, can't use incompatible jobs. Must include the optimal solution to problems consisting of remaining compatible jobs 1,2,…,p(j).
    - OPT does not selected job j. Must include optimal solution consisting remaining compatible jobs 1,2,…j-1.
    - $OPT(j) = 0 \; if \; j = 0$
    - $\mathrm{OPT(j)} = \max\{v_j + OPT(p(j)), OPT(j-1)\} \; otherwise$
  - **Dynamic programming solutions are bottom up.**
  - **Memoization:** Store the results of each sub-problem; lookup when needed.
  - **Running time:** $O(nlogn), or \; O(n) \; if \; jobs \; presorted.$
- **Invariant:** Everything to the left of i is optimal.
- **Maximum-sum contiguous subarray:**
  - OPT[i] = optimal ending at i
  - OPT[i] = max{OPT[i-1]+A[i], 0}.
- **Knapsack**
  - OPT(i,w) = max profit subset of items 1,…, I with weight limit w.
  - OPT(i,w) = 0 if i = 0, OPT(i-1,w) if w_i > w, max(OPT(i-1,w), v_i + OPT(i-1,w-w_i)) otherwise.
- **RNA**: string of B= b1,b2,…bn over alphabet {A,C,G,U}.
  - **Secondary structure:** RNA is single-stranded so it tends to loop back and forms base pairs with itself
  - **Free energy**: Usual hypothesis is that an RNA molecule will form the secondary structure with optimum total free energy.
  - **Goal**: Given an RNA molecule B = … and secondary structure S that maximizes the number of base pairs.
  - No crossing, no sharp turns, A connects to C, and G connects to C.
  - **Subproblems:** OPT(I,j) = maximum number of base pairs in a secondary structure of substring b_i, bi+1 to bj.
  - **Recurrence:**
    - Case 1: Base b_j is not involved in a pair. OPT(I,j) = OPT(I,j-1).
    - Case 2: Base b_j pairs with b_t for some $i \le t < j-4$.
      - Non-crossing constraint decouples resulting sub-problems
      - OPT(I,j) = 1+max{OPT(i,t-1)+OPT(t+1,j-1)} where $i \le t < j-4$
  - **Base case:**
  - If $i \ge j-4 \; then \; OPT(i,j) = 0 \; by \; no-sharp \; turns \; condition.$
  - Running time: $O(n^3)$
- **Shortest path problem:** Given a directed graph G=(V,E), with edge weights, find the shortest path from s to t.
  - **Motivation:** negative edge weights and cycles.
  - Adding a constant to every edge weight can fail.

- Negative edge cycles cannot have a shortest path; infinite loop.
- **Sub-problems:**
  - OPT(I,v) = length of shortest v-t path P using at most I edges.
- **Recurrences:**
  - Case 1: P uses at most i-1 edges. OPT(I,v) = OPT(i-1,v).
  - P uses exactly I edges.
    - If (v,w) is first edge, then OPT uses (v,w) and then selects best w-t path using at most i-1 edges.
  - $OPT(i,v) = \min\{OPT(i-1,v), min[OPT(i-1,w) + c_{vw}]\}\,(v,w)\epsilon\, E$
- **Solve base case:**
  - OPT(0,t) = 0, and OPT(0,v =/ t) = infinity.
- **Bellman ford:**
  - For reach node $v\ in\ V$, set M[v] = infinity, and successor[v] = ∅
  - M[t] = 0
  - For i =1 to n-1
    - For each node w in V, if M[w] has been updated in previous iteration, then for each node v such that (v,w) in E. If M[v] > M[w] + c_vw then update.
    - With successor[v] being w.
    - If no M[w] value has changed in iteration I, then stop.
  - Improvements:
    - Maintain only one array M[v] = shortest apth we have found so far.
    - No need to check edges of form (v,w) unless M[w] changed in previous iteration.
  - **Theorem:** Throughout the algorithm, M[v] is length of some v-t path, and after I rounds of updates, the value M[v] is no larger than the length of shortest v-t path using <= I edges.
  - **Memory:** O(m+n)
  - **Running time:** O(mn) worst case, but on average is faster.
- **Least squares**
  - Find a line or lines $y = ax + b$ that minimises the sum of the squared error.
  - **Sub-problems**:
    - OPT(j) = minimum cost for points $p_1, p_2, \dots, p_j$
  - **Recurrences**: OPT(j) =$\min(e(i,j) + c + OPT(i-1)\}\ 1 \le i \le j, e(i,j)$ = minimising sum of squares for points $p_1 \dots p_j$.
  - **Base case:** OPT(0) = 0.
  - **Running time:** $Running\ time: O(n^3), space\ (O(n^2))$.

Flow Networks

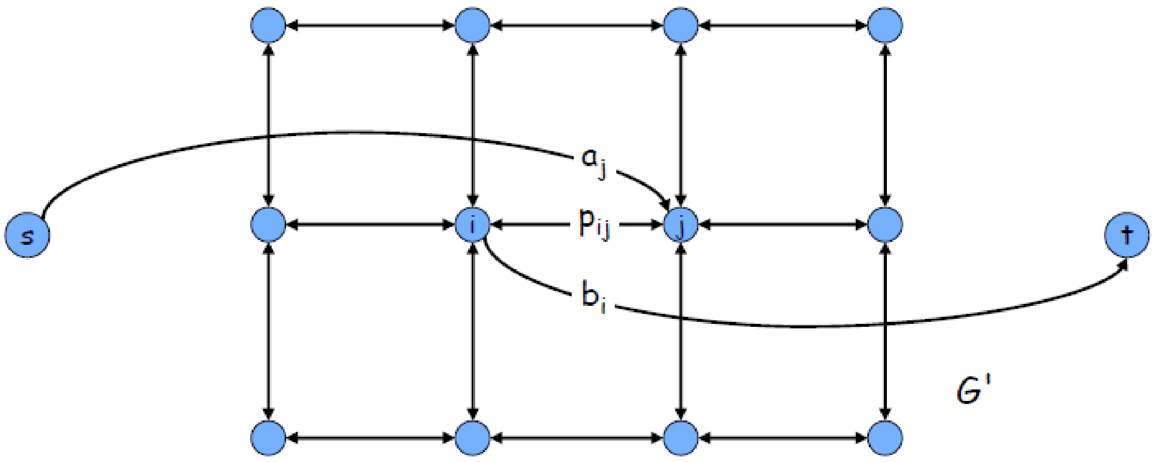
- An s-t flow is a function that satisfies
  - For each $e\ \epsilon\ E: 0 \le f(e) \le c(e)$ Capacity restriction.
  - For each $v\ \epsilon\ V - \{s,t\}: \sum f = \sum f(e)\ (e\ out\ of\ s)$ conservation.
  - $v(f) = \sum f(e)$, the total value of flow is the flow leaving s.
- **Max flow problem**
  - Find s-t flow of maximum value.
- **Cuts**
  - An s-t cut is a partition (A,B) of V with $s\ \epsilon\ A\ \&\ t\ \epsilon\ B$

- The capacity of a cut (A,B) is: $cap(A, b) = \sum c(e)\ \ (e\ out\ of\ A)$.
- The net flow sent across the cut is equal to the amount leaving s.
    - This is for flow across a cut not the capacity as described above.
    - $v(f) = f^{out}(a) - f^{in}(A)$
- **Weak duality**
    - The value of the flow is at most the capacity of the cut.
        - $v(f) \leq cap(A, B)$
- **Optimality**
    - If $v(f) = cap(A, B)\ then\ f\ is\ a \max flow\ and\ (A, B) is\ a \min cut$.

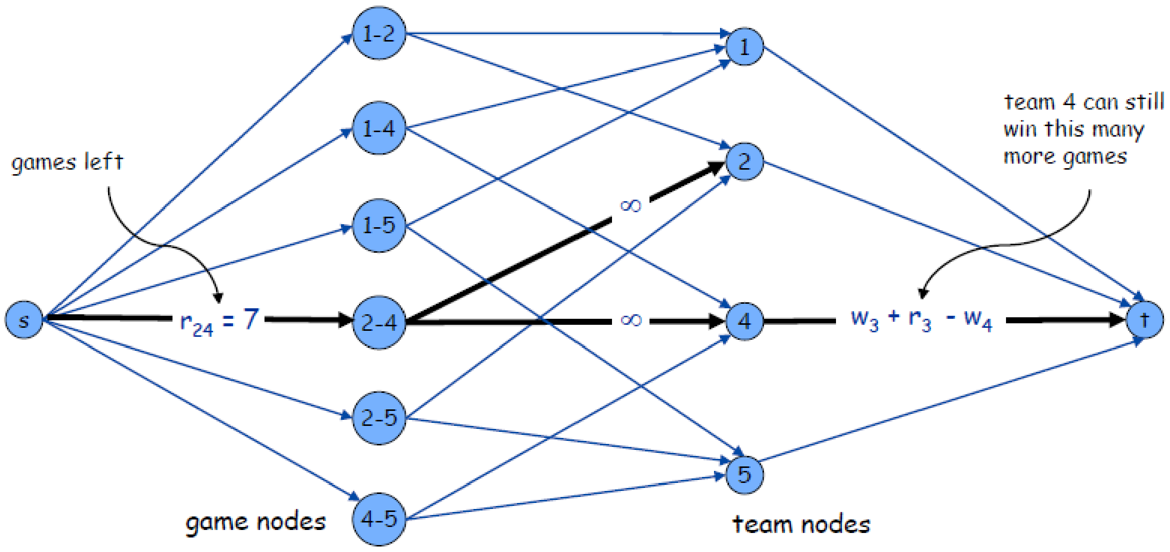
- **Greedy algorithm**
    - Start with f(e) = 0 for all edges $e \epsilon E$
    - Find an s-t path P where each edge has $f(e) < c(e)$
    - Augment flow along path P.
    - Repeat until stuck.
    - **Residual graph**
        - You put a backward edge with the amount of flow in graph G'
        - The capacity of the forward edge is $c(e) - f(e)$ and $f(e)$ for the backward edge.
        - "Undo" flow sent.
        - The residual capacity of an edge in G_f tells us how much flow we can send, given the current flow.
    - **Bottleneck**
        - The minimum residual capacity of any edge on P with respect to current flow f. This is the maximum flow we can send along this simple s-t path.
    - **Ford-Fulkerson**
        - Build residual graph, initialise all flows as 0.
        - While there exists an augmenting path P in G_f
            - f = Augment(f,P) **[Bottleneck]**
            - update G_f
            - return f.
        - **Proof**
            - Assume initial capacities are integers.
            - At every intermidate stage of Ford-Fulkerson algorithm the flow values and the residual graph capacities in G_f are integers
            - Induction
                - Base case: initially the statements are correct
                - Induction hyp: true after j iterations
                - Induction step: Since all residual capacities in Gf are integers, the bottleneck must also be an integer. Thus the flow will have integer values and hence also the capacities in the new residual graph
            - Integrality theorem: If all capacities are integers, then there exists a max flow f for which every flow value is an integer.
        - Augmenting path theorem: Flow f is a max flow if and only if there are no augmenting paths
        - Max-flow min-cut theorem: The value of the max flow is equal to the value of the min-cut.

        - Running time: O(C(m+n)).
        - **Scaling max-flow algorithm**
            - Use a delta scaling factor
            - Running time: $O(m^2 logC)$
    - **Bipartite matching**
        - A subset of E is a matching if each node appears in at most one edge in M.
        - **Max matching:** find a max cardinality matching.
            - **Algorithm**
            - Use max flow
                - Create digraph G'
                - Direct all edges from L to R, and assign unit capacity
                - Attached source s, and unit capacity edges from s to each node in L
                - Likewise R to t with unit capacities.
            - Proof
                - You can only have 1 unit of flow on each of k paths define by M,
                - F is a flow and it has a value k.
                - Cardinality is therefore at most k.
                - Integrality theorem k is integral so f(e) is 0 or 1.
                - Each node in L and R participate in at most one edge in M.
        - **Perfect matching:** All nodes appear in the matching.
            - **Marriage theorem:**
            - G has a prefect matching iff |N(S)| ≥ |S| for all subsets S in L. E.g. the neighbourhood of 3 nodes has only 2 nodes, hence no perfect matching.
- **Disjoint paths:** Two paths are edge-disjoint if they have no edge in common.
    - Assign unit capacity to every edge.
    - Max flow is the max edge disjoint paths.
    - **Network connectivity**
        - The maximum number of edge disjoint s-t paths is equal to the min number of edges whose removal disconnects t from s.
- **Circulation with demands**
    - Sum of supplies = sum of demands
    - Max flow
        - Attach a new source s and sink t,
        - For each v with d(v) < 0, add edge (s,v) with capacity –d(v).
        - " " " " d(v) > 0, add edge (v,t) with capacity d(v).
        - G has a circulation iff G' has max flow of value D.
- **Survey design**
    - Design survey asking n_1 consumers about n_2 products.
    - Can only survey consumer i about a product j if they own it
    - Ask consumer i between $c_i and\ c_i'$ questions
    - Ask between $p_j\ and\ p_j{}'$ consumers about product j
    - **Circulation problem**
        - Include edge (I,j) if consumer owns product I,

        - If the circulation problem is feasible then the survey problem is feasible and vice versa.
- **Image segmentation.**



- **Baseball elimination**
    - Team is eliminated iff max flow is less than the total number of games left.

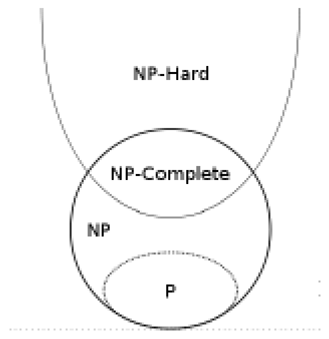


<u>**Polynomial-Time reduction**</u>

- Problem X polynomial reduces to problem Y, denote by $X \leq_p Y$ if arbitrary instances of problem X can be solved using polynomial reduction to Y.
- **Reduction strategies**
    - Reduction by simple equivalence
        - Vertex cover and independent set; they reduce to each other, S is an independent set iff V\s is a vertex cover.
    - Reduction from special case to general case
        - Vertex cover to set cover (basically the same problem).
    - Reduction by encoding with gadgets
        - Most used: change the input to match to solve the problem using another problem (like assignment 5).

<u>**Class NP-hard**</u>

- NP-Complete: A problem in NP such that every problem in NP polynomially reduces to it.
- NP-hard: A decision problem such that every problem in NP polynomially reduces to it.



$P \neq NP$

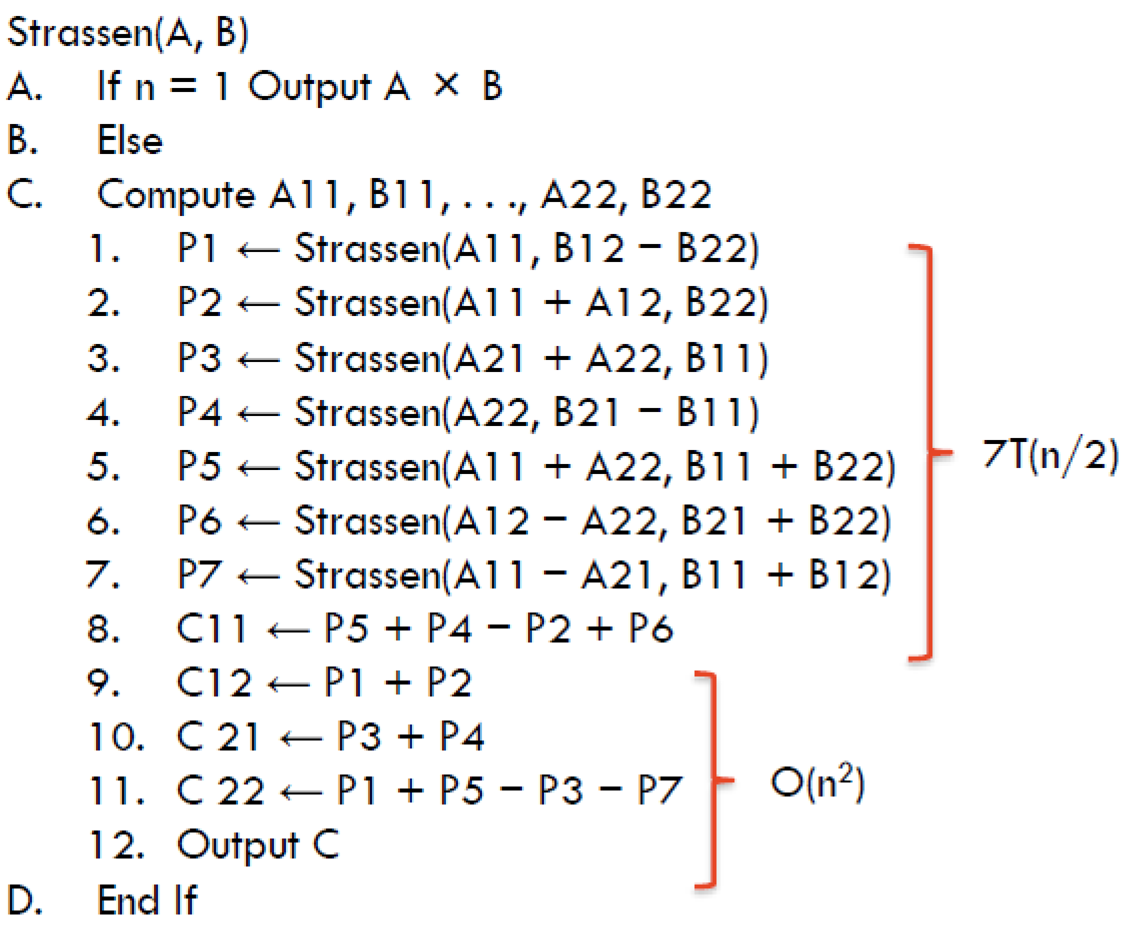
- P: decision problems for which there is a poly-time algorithm.
- NP: Decision problems for which there is a poly-time certifier.

Solving NP-complete problems

- Unlikely to find a poly-time algorithm.
    - Sacrifice one of the tree desired features.
    - Solve problem to optimality
        - Approximation
        - Randomization
    - Solve problem in polynomial time
        - Exponential algorithm
    - Solve arbitrary instances of the problem
        - Solve restricted classes of instances
        - Parameterized algorithms
        - **E.g.** independent set on trees can be solved in O(n) time.
- **Approximation ratio**
    - $Approximation\ ratio = \frac{Cost\ of\ apx\ solution}{Cost\ of\ optimal\ solution}$
    - An approximation algorithm for a minimization problem requires an approximation gurantee:
        - Approximation ratio $\leq c$
        - Approximation solution $\leq c * value\ of\ optimal\ solution$
- **3-Sat**
    - SAT where each clause contains 3 literals, is there a truth assignment? [satisfiable].
- **Clique**
    - A clique of a graph G is complete sub graph of G, is there a sub-graph of size k.
    - A G has a k-clique iff E is satisfiable.
- **Careful with direction of reduction:**
    - **In order to prove NP-hard, we reduce a known problem to it. E.g. independent set can be decided iff the instance is satisfiable.**
- **Hamiltonian cycle**
    - Does there exist a simple cycle C that visits every node.
- **Longest path**
- **TSP:** Given a set of n cities and a pairwise distance function d(u,v) is there a tour of length $\leq$ D.

COMP2907

- **MST based approximation algorithm TSP**
  - 2-approximation for metric TSP.
  - Proof
    - Assume H_{opt} is the optimal tour and that H_A is the tour returned by the approximation algorithm.
    - $cost(T) \leq cost(H_{opt})$
    - $cost(W) \leq 2 * cost(T) \leq 2 * cost(H_{opt})$
    - Will revisit at most each node twice.
- **Set Cover:** A finite set X and a family F of subsets of X such that
  - $x = U\ s\ (s\ in\ F)$
  - Find a subset C of minimal size which covers X.
  - That is find the minimum number of sets that covers the universe, X.
  - **Algorithm:**
    - **C <- empty set.**
    - **U < - X**
    - While U ≠ ∅ do
    - Select a subset in F that maximises |S intersection U|
    - C <- C union {S}
    - U <- U – S.
    - Return C.
- Loose ratio-bound
  - Claim: if set cover of size k, then after k iterations the algorithm covered at least ½ of the elements.
  - Proof
    - We can have the set O(logn) times.
    - Each time we perform k iterations.
    - Therefore after klogn iterations all the n elements must be covered.
- Fast matrix multiplication
  - **Divide**: partition A and B into ½ n by ½ n blocks.
  - **Compute**: 14 ½ n by ½ n matrices via 10 matrix additions.
  - **Multiply:** 7 ½ n by ½ n matrices recursively.
  - **Combine**: 7 products into 4 terms using 8 matrix additions.



```
Strassen(A, B)
A.  If n = 1 Output A × B
B.  Else
C.  Compute A11, B11, . . ., A22, B22
    1.  P1 ← Strassen(A11, B12 − B22)
    2.  P2 ← Strassen(A11 + A12, B22)
    3.  P3 ← Strassen(A21 + A22, B11)
    4.  P4 ← Strassen(A22, B21 − B11)
    5.  P5 ← Strassen(A11 + A22, B11 + B22)
    6.  P6 ← Strassen(A12 − A22, B21 + B22)
    7.  P7 ← Strassen(A11 − A21, B11 + B12)
    8.  C11 ← P5 + P4 − P2 + P6
    9.  C12 ← P1 + P2
    10. C 21 ← P3 + P4
    11. C 22 ← P1 + P5 − P3 − P7
    12. Output C
D.  End If
```

- **TSP by dynamic programming**
  - Regard a tour to be a simple path that starts and end at vertex 1.
  - Every tour consists of an edge (1,k) for some k in V – {1} and a path from k to vertex 1. The path from vertex k to vertex 1 goes through each vertex V – {1,k} exactly once.
  - Let OPT[U,t] be the length of the shortest path starting at vetex 1, going through all vertices in U and terminating at vertex t
    - Length of shortest ath starting at vertex 1 going through all vertices in U and terminating at vertex t.
    - OPT[V,1] is the length of an optimal salesperson tour

– $|U|=1 \Rightarrow \mathrm{OPT}[U=\{t\},t] = d(s,t)$

– $|U|>1 \Rightarrow \mathrm{OPT}[U,t] = \min \mathrm{OPT}[U\setminus\{t\},u] + d(u,t)$

  - Compute all solutions to sub-problems in order of increasing cardinality of U.
  - $Sets\ U = 2^n$
  - $vertices\ connected\ to\ t < n$
  - $Total\ time: O(n^2 2^n)$
- **Karger's algorithms**
  - Global minimum cut: find a cut (S,S') of minimum cardinality
  - While |V| > 2; contract an arbitrary edge (u,v) in G, return the cut S.
  - This algorithm is an approximation algorithm.

        - To amplify the probability of success, run the contraction algorithm many times
        - Repeat the contract algorithm r[n 2] times with independent random choices, the probability that all runs fail is at most n^(-c).
    - Running time:
        - N-2 iterations
        - Each iteration O(n)
        - O(n^2)
        - The algorithm is iterated O($n^2 \log(n)$) times.
        - Hence O($n^4 logn$).
- **PSPACE:** Decision problems solvable in polynomial space; can include exponential time algorithms.
- **PSPACE-complete**: Y is in Pspace and every problem, X, in Pspace can be reduced to y.
- **Quantified SAT:** Let there be a Boolean clause, is there a way that we can pick the first value, such that if the second is generated at random, can we have satisfiability, assume odd terms. IS IN PSPACE-complete.

QSAT: $\exists x_1 \; \forall x_2 \; \exists x_3 \; \forall x_4 \ldots \; \forall x_{n-1} \; \exists x_n \; \Phi(x_1, \ldots, x_n)$

SAT: $\exists x_1 \; \exists x_2 \; \exists x_3 \; \exists x_4 \ldots \; \exists x_{n-1} \; \exists x_n \; \Phi(x_1, \ldots, x_n)$

- Planning problems: Is it possible to apply a sequence of operations (legal) to get from initial configuration to goal configuration? IN EXPTime.

The art gallery problem:

- **How many guards are needed to guard an art gallery.**
- **Input**: A simple polygon with n line segments.
    - Divide the polygon into triangles.
    - #guards = #triangles.
    - Proof: every pair of points must see each other in triangle, hence a guard placed in the centre can see every point in the triangle.
    - Every simple polygon has a triangulation; because every polygon with vertices > 3 has a diagonal.
    - To solve; assign a 3-colour to each vertex such that no two adjacent vertices have the same colour; place all the guards on the colour with the least number.
    - A 3-colouring exists because every polygon n>3 vertices has at least two non-overlapping ears. Proof: connect all the mid-points of the ears; forms a tree. Each node is connected to two other nodes.

k-path

- Find a simple path in G on k vertices.
- Improved algorithm:
    - Colour the vertices of G with k colours uniformally at random.
    - Find a colourful k path in G if one exists.
    - Probability is e^k.
    - Use dynamic programming for all paths.

Iterate over i = the number of colours in C.

[i=1]: P[C,v] = 1 iff C={c(v)}

[i>1]: P[C,v] = 1 iff c(v)∈C and ∃(u,v)∈E s.t. P[C\{c(v)},u]=1.

Theorem: A k-colourful path in a graph G with k colours can be found in $O(2^k \cdot (|V|+|E|))$.

-
- **Parameterised problem:**
  - Given an instance of the problem and a parameter k, can we give a yes or no instance?

P: class of problems that can be solved in time $n^{O(1)}$
FPT: class of problems that can be solved in time $f(k) \cdot n^{O(1)}$
W[·]: parameterized intractability classes
XP: class of problems that can be solved in time $f(k) \cdot n^{g(k)}$

$$P \subseteq FPT \subseteq W[1] \subseteq W[2] \cdots \subseteq W[P] \subseteq XP$$

Known: If FPT $=$ W[1], then the Exponential Time Hypothesis fails, i.e. 3-SAT can be solved in time $2^{o(n)}$.

Kernilisation: A polynomial time transformation that maps an instance (I,k) to an instance (I’,k’) such that:

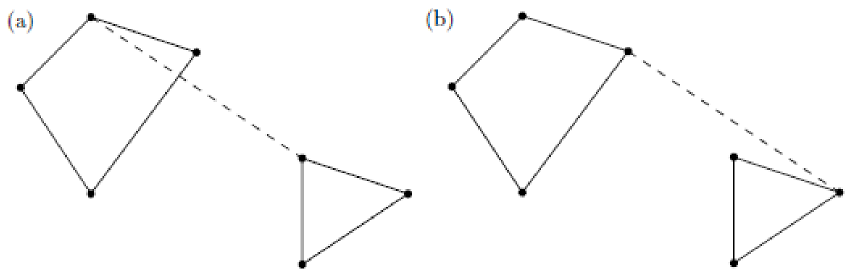
- The kernalised transform is mapped to yes if it is a yes.
- And k >= k’
- |I’| <= f(k) for some function f(k).
- i.e. for vertex cover; delete every vertex of degree > k and decrease k accordingly.

Tutorials

1) If we have a blackbox, we can only make assumptions about the overall lower bound. We can't make any assumptions about the upper bound.
2) Greedy algorithm, 2-approximation; pick the edge points of k-edges. If we pick at least k edges, we have at least 2k vertices.
3) If we add edge weights to shortest path; it is no longer the shortest path, or by squaring (weight < 1).
4) Optimal spanning tree does not change; order of edge weights doesn't change.
5) Greedy algorithm for sorting by time/weight; exchange argument.
6) S-t path with vertex weights; change vertex into two vertices with the vertex weight being the edge between them.

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with $\varepsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

7) Where a is the size of the sub-problem, and b is the number of subproblems.
8) Pigeon-hole principle: If n items are put into m containers such that n>m, then at least one container must contain more than one item.
9) Locally optimal index. Start at the centre and follow the lowest branch, if u hit an up branch, take the lower point.
10) K-th smallest element; find two pairs of points by comparing k to the rank. An element is I <= rank(i) <= I + m. can safely ignore items with rank greater than k in the larger array.
11) m line intersection problem; report every line on a segment S: Sort the end points of the intervals and the points from left to right. Set a counter c to zero. Sweep all the event points from left to right. If left endpoint then increment c, and if right endpoint decrement c, if a point in p, then report p if c >0. O((n+m)log(n+m)).
12) Square intersection, put in BST, reduces to intersection in 1-D IN THE Y direction. Maintain a BST, insert if point inside square in x-direction. If right side of square, remove it.
13) If I can see another interval; ray intersection; sweep r counterclockwise from a position, initialise a BST that contains all segments intersecting r, ordered using a distance function and the intersection point. Consider the events.
14) To merge two hulls in a hull; compute upper and lower tangents and discarding all points lying between the two tangents. Find the rightmost point and leftmost point, if it's not a tangent, go clockwise/anticlockwise.



15) Reverse-MST works. Sort by highest weights, and if graph remains connected after removing the edge, keep goin.
16) Another variation, for every cycle in T, remove the heaviest edge.

Proof for MSTs is via contradiction, we assume it's an MST first, and show that it contradicts with that MST definition.

17) Fibonacci recursive is equal to the Fibonacci number itself exponentially. An alternative is to use dynamic programming:
- Base cases: M[0] – 0, M[1] – 1, M[i] = M[i-1] + M[i-2].

18) For some cases like with coins, you must initialise your base case for the first 10 coins. And then use min(1+ j-1, j-7 and j-10).
19) Dynamic programming, make the table.

Algorithm 3 PLAYLIST

```
function PLAYLIST($n$)
    for $i = 1$ to $n$ do
        $C[i, i] = 0$
    end for
    for $|S| = 2$ to $n$ do
        for every $S$ with cardinality $|S|$ do
            for every $i \in S$ do
                $C[S, i] := \max_{k \in S \setminus \{i\}} \{C[S \setminus \{i\}, k] + c_{k,i}\}$
            end for
        end for
    end for
    return $\max_{1 \leq i \leq n} C[1..n, i]$
end function
```

20) To test for valid words; find 1,i.

21)
$$M[i] = \bigvee_{1 \leq j < i} \text{lookup}(s[j, i]) \wedge M[j-1]).$$

22) In the residual graph; any node reachable by s is in the min cut.
23) Vertex capacities can be accounted for like before.
24) Effective radius problem; precompute edge capacities if the pair is in range.
25) Flight problem: assign rigid unit capacities; add an extra edge, if the flight is reachable (pre-compute).
26) For an undirected graph; create two antiparallel edges; max flow is k for capture flag problem. Ford Fulkerson.
27) For even edges, the max flow must be even, (bottleneck must be even; use induction).
28) Every d-regular graph has pefect matching because marriage theorem holds.
29) To reduce vertex cover to set cover; U is the edge set of G, for each vertex, we make a set containing the edges incident on u, set t = k. Vertex cover is a special case of set cover.
30) D-interval scheduling; use independent set.
31) Degree at least delta, use base case n = 2, then input dummy nodes to increase degree.

32) Value of max flow always equal to the min cut.
33) Augment called at most C times; every iteration, capacity must increase by 1.