01 Divide and Conquer

1. T(n) = O/Ω($f(n)$): There exists a constant c such that, for all sufficiently large n, T(n) ≤/≥ $cf(n)$.
2. Two colors are always sufficient to color the regions formed by any n lines: For each region separated by new line L, left side retain, right side reverse.
3. Master Theorem:
   $T(n) = aT(n/b) + f(n)$. Assume f is non-decreasing. Let $k = \log_b a$.
   (a) $f(n) = O(n^{k'})$ where $k' < k$ ⇒ $T(n) = O(n^k)$
   (b) $f(n) = O(n^k \log^c n)$ for some c ≥ 0 ⇒ $T(n) = O(n^k \log^{c+1} n)$
   (c) $f(n) = O(n^{k''})$ where $k'' > k$ ⇒ $T(n) = O(f(n))$
4. Integer Multiplication: $x = 2^{n/2}a+b$, $y = 2^{n/2}c+d$, $xy = 2^n ac + 2^{n/2}(ad + bc) + bd$. $T(n) \le 3T(n/2) + k'n = O(n^{1.585})$, where $\log_2 3 = 1.585$.
5. Finding the closest pair of points:
   Divide: $n/2$ points on each side.
   Conquer: Merge the sorted points, and find the closest pair as before.
   Combine: Sort the points inside the 2-d strip with respect to y-axis. Inside the strip, find the closet pair with one point in each side.
   $T(n) \le 2T(n/2) + cn = O(n\log n)$.
6. Strassen's algorithm
7. Median/k-th smallest element for any given k:
   1. SELECT (S, k) // |S| = n, and $1 \le k \le n$
   2. Divide the n elements into $n' = n/5$ groups of 5.
   3. Find the median of each 5-element group.
   4. x = SELECT ({ medians of all 5-element group }, $\lceil n'/2 \rceil$)
   5. Partition the n numbers into 3 subsets A, B and C :
      – A contains numbers < x,
      – B for those = x, and
      – C for those > x.
   6. If |A| < k ≤ |A|+|B| then return x
      If k ≤ |A| then SELECT (A, k)
      If k > |A|+|B| then SELECT (C, k - |A| - |B|).
   $T(n) = T(n/5) + T(3n/4) + cn$. Since n/5 + 3n/4 < n, T(n) = 20 cn.

02 Graph Basics

1. u and v are said to be connected if there is a path from u to v.
   G is said to be connected if every pair of vertices are connected.
2. DFS(v):
   Visited[v] = true;
   For each vertex w in the adjacent list of v
     if Visited[w] = false
       then DFS(w)
3. BFS:
   While Q is not empty
     Remove the first vertex v from Q;
     For each vertex w in the adjacent list of v,
       if Visit[w] = false
         then Visit[v] = true; add w into Q;
4. Find connected components:
   cc = 0;
   Visited [u] = 0 for all vertices u;
   For u = 1 to n;
     if Visited[u] = 0 then
       cc = cc + 1;
       DFS(u, cc); ["visited[v] = true"→"visited[v] = cc"]
5. Construct a spanning tree:
   in BFS or DFS, add "Parent(v) = u" before iteration
6. Detect cycles:
   DFS(u)
     Visited[u] = true;
     For each vertex v in the adjacent list of u,
       if Visited[v] = false
         then Parent(v) = u; DFS(v)
       else if Parent(u) ≠ v then report a cycle exist; exit

03 MST

1. Prim's algorithm: Start with an arbitrary node s and greedily grow a tree T from s outward. Repeatedly add the "lightest" (min-weight) edge e that is between a vertex in T and a vertex outside T.
2. Implementation of Prim's algorithm: O(m log n) time.
   Initialization:
   1. Pick an arbitrary vertex u in V. Visited[u] = true.
   2. Visited[v] = false for all vertices v in V – {u}.
   3. For every v in V,
        if v is adjacent ot u,
          then min-weight[v] = w({u, v}); min-neighbor[v] = u;
        else min-weight[v] = ∞
   4. Build a heap Q: [v, min-weight(v)] for all v in V – {u}, with min-weight(v) as the key.
   5. While (Q is not empty) {
        extract and delete min key from Q: [v, min-weight(v)]
        Visited[v] = true
        for each edge e' = {v, w}
          if Visited[w]=false and $w(e')$ < min-weight[w]
            min-weight[w] = $w(e')$; decrease-key for w in Q
            min-neighbor[w] = v
      }
3. Kruskal's algorithm: Start with an empty tree T. Repeatedly add the next lightest edge e that does not create a cycle.
4. Implementation of Kruskal's algorithm:
   1. Sort the edges in ascending order.
   2. Create n sets each containing a vertex of the graph.
   3. At any time, vertices that are connected together by the edges found by the Kruskal Algorithm are merged into one set.
   4. Inserting an edge $e = (u, v)$ means merging two sets containing u and v, respectively. O(1) time.
   5. To check cycle for an edge $e = (u, v)$: see if the set containing u is the same set as that of v. O(log n) time.
   Overall: O(n + m log$n$) time.
5. Union and Find
   FIND (x): follow the chain of pointers to an element y that points to itself; then return y; O(log n) time.
   UNION(x, y): if the set of x is smaller, then x points to y; otherwise y points to x; O(1) time.

04 Digraphs

1. Find all vertices w such that there is a path from s to w:
   If s has a path to w, w must be inside the DFS tree rooted at s.
2. Find all vertices w which have a path to s: construct G^R
3. Check strongly connected: A simple O(n+m)-time algorithm: Pick any vertex s, check if s can reach all vertexes, and all vertexes can reach s. If yes, then G is strongly connected.
4. Edges in DFS of digraphs:
   (a) Tree edges: edges in a DFS tree (forest).
   (b) Back edges: from a node to an ancestor.
   (c) Forward edges: from a node to a non-child descendant.
5. Cycles in directed graph:
   A directed graph G has a cycle if and only if DFS (no matter where it starts) finds a back edge .
6. Find Back/Tree/Cross edge:
   Back edge: start(v) < start(u) < finish(u) < finish(v);
   Forward/tree edge: start(u) < start(v) < finish(v) < finish(u);
   Cross edge: start(v) < finish(v) < start(u) < finish(u).
   Therefore (u, v) is a back edge iff start(v) < start(u) < finish (v).
7. Detect cycles:
   For u = 1 to n { Visited[u] = false; start[u] = finish[u] = ∞ }
   clock = 1;
   For u = 1 to n;
     if Visited[u] = false then DFS(u);

   DFS(u)
   1. Visited[u] = true; start[u] = clock; clock = clock + 1;
   2. For each edge (u, v),
        if Visited[v] = false
          then DFS(v)
        else if (start(v) < start(u) < finish(v)) reports a cycle; exit.
   3. finish[u] = clock; clock = clock + 1
8. DAG: A directed graph without a cycle (directed acyclic graph).
9. Lemma: In a DAG $G = (V, E)$, there is a vertex with no incoming edges (source), and there is a vertex with no outgoing edges (sink).
10. Topological Sort (G)
    1. DFS (G);
    2. Output the nodes in reverse order of their "finish" times.
11. SCC (Strongly connected components): every vertex is reachable from every other vertex.
12. The SCC Algorithm:
    1. DFS the entire G^R, and record finish(v) for all vertices.
    2. Find the vertex v with largest finish(v) - sink in G.
    3. DFS (G, v); all vertices visited are in the same SCC.
    4. Remove the vertices just visited.
    5. Repeat Step 2 until no vertex is left.

05 Shortest paths

1. Dijkstra's Algorithm (form s to u, greedy): O((m + n)log$n$)
   S = { all nodes v for which SD(v) is known }. Initialize S = { s }.
   1. For each other node u,
        let d[u] = w(e) if e = (s, u) exists, and ∞ otherwise
   2. let v outside S have the smallest d[v];
   3. Insert v into S.
   4. For each edge e = (v, u) and u not in S,
        if d[v] + w(e) < d[u] then update d[u] = d[v] + w(e)
   5. Repeat Step 2 to Step 4 until |S| = n
2. Bellman-Ford Algorithm (support negative edges, dp): O(mn)
   D[v] = shortest s-v path we have found so far
   1. For each node v in V
        D[v] = ∞, D[s] = 0.
   2. For i = 1 to n-1
        For each edge (u, v) in E
          D[v] = min{ D[v], D[u] + w(u, v)}
   3. Method 1: By Wikipedia
        For each edge (u, v) with weight w
        if D[u] + w < D[v] then report a cycle and exit
      Method 2: By Prof.
        Add a new vertex s_0 to G, with zero-weight edges from s_0 to all v in V. Compute Opt(n+1, v) and Opt(n, v) for the (n+1) nodes. If Opt(n+1, v) < Opt(n, v) for some v, then report a cycle and exit.

06 Greedy Algorithms

1. Minimum spanning trees: Prim's and Kruskal's algorithm
2. Shortest path: Dijkstra's Algorithm
3. Interval scheduling:
   Input: n jobs; job j starts at s_j and finishes at f_j.
     Two jobs are compatible if they don't overlap.
   Goal: find the largest subset of mutually compatible jobs.
   Greedy: earliest finish time.
4. Huffman codes: Build the tree bottom up. The two characters x, y with the lowest frequencies are put under an internal node.

07 Dynamic programming

1. Shortest paths: Bellman-Ford Algorithm
2. Weighted compatible intervals:
   Job j starts at s_j, finishes at f_j and has value (weight) v_j. Let p(j) = largest i < j such that job i is compatible with job j.
   DP-Select {
     W[0] = 0
     For j = 1 to n
       W[j] = max (v_j + W[$p(j)$], W[j-1]).
   }
   What jobs are selected: backward tracing: call Find-set (n)
   Find-set (j) {
     if j = 0 return;
     if (v_j+ W[p(j)] > W[j-1]) then print j; Find-set (p(j))
     else Find-set(j-1).
   }
3. Longest increasing subsequence: O(n^2) Algorithm
   For i = 1, 2, …, n
     OPT(i) = 1
     For k = 1, …, i-1
       if $a_k < a_i$ then OPT(i) = max{ OPT(i), OPT(k)+1 }
4. Longest increasing subsequence: O($n$log$n$) from book
   ```
   int dp[MAX_N];
   void solve(){
     fill (dp, dp+n, INF);
     for (int i = 0; i < n; ++i)
       *lower_bound(dp, dp+n, a[i]) = a[i];
     printf("%d\n", lower_bound(dp, dp+n, INF) - dp);
   }
   ```
5. Knapsack problem: Item i weighs w_i, values v_i. Capacity = W.
   For i = 1 to n
     For x = 1 to W
       M[i, x] = max { M[i-1,x], v_i + M[i-1, x-w_i] } (if x-w_i > 0)
   Return M[n, W]
6. Matrix-Chain Multiplication: O(n^3)
   For s = 1 to n-1
     For i = 1 to n [precisely, for i = 1 to n-s]
       $j = i + s$
       $C[i, j] = \min_{i \le k < j} \{ C[i, k] + C[k+1, j] + d_{i-1}d_k d_j \}$
7. All-pairs shortest paths: Warshall's Algorithm O(n^3)
   Initialization: dist[x, y, 0] = e(x, y) is in E, otherwise ∞.
   Prof.'s: For k = 1 to n
       For all x, y ∈ [1, n]
         dist[x, y, k] = min {dist[x, y, k-1],
                   dist[x, k, k-1] + dist[k, y, k-1] }.
   Wiki's: For k from 1 to n
       For i from 1 to n
         For j from 1 to n
           if dist[i][j] > dist[i][k] + dist[k][j]
             then dist[i][j] = dist[i][k] + dist[k][j]
8. An alignment: pairing up two strings character by character, possibly with space inserted. A similarity function (score) δ: specifies how much each match/mismatch/space contributes to the overall similarity.
   With respect to a similarity function, an optimal alignment is an alignment with the maximum score. The alignment problem is to find the optimal alignment (also known as the global alignment problem) and its score.
9. Needleman-Wunsch algorithm:
   Consider two strings S[1..n] and T[1..m], V(i, j) = the score of the optimal alignment between the substrings S[1..i] and T[1..j].
   V(0, 0) = 0, V(0, j) = j δ(_, T[j]), V(i, 0) = i δ(S[i], _).
   For i > 0, j > 0
     V(i, j) = max {V(i-1, j-1)+δ(S[i], T[j]),
            V(i-1, j)+δ(S[i], _),
            V(i, j-1)+δ(_, T[j])}
   Time = O(nm), Space = O(nm). If we only need the score, we can just store the two current rows and Space => O(m).
10. Find the mid-point alignment (where S[n/2] is aligned to)
    Fact: $V(S[1..n], T[1..m]) = \max_{0 \le j \le m}\{V(S[1..n/2], T[1..j]) + V(S[(n/2)+1..n], T[j+1..m])\}$
    1. Recompute V(S[1..n/2], T[1..j]) for all j:
       Using Needleman-Wunsch algorithm.
    2. Recompute V(S[(n/2)+1..n], T[j+1..m]) for all j:
       First reverse S and T then do the same as step 1.
    3. Determine which j maximizes the above sum.
    Time = O(n/2 m) + O(n/2 m) + O(m) = O(nm).
11. Recover the alignment:
    1. Dind the mid-point of the alignment.
    2. Divide the problem into two halves.
    3. Recursively deduce the alignments for the two halves.
    Time Analysis: T(n, m) ≤ cnm + cmn/2 + cmn/4... = 2cnm
    Space Analysis: O(n+m)
12. Edit distance (S, T) = the fewest number of insert/delete/modify operations to transform S to T. => a special case of alignment
13. X is a suffix of S[1..n] if X=S[k..n] for some k ≥ 1
    X is a prefix of S[1..n] if X=S[1..k] for some k ≤ n
14. Local Alignment:
    Input: two strings (DNA) S and T.
    Compute: the most similar substrings of S & T (i.e., substrings A and B whose alignment score is maximized over all possible pairs of substrings)
    Method: Define V(i, j) to be the maximum score of the (global) alignment of A and B over all suffixes A of S[1..i] and all suffixes B of T[1..j].
      Then, the score of local alignment is $\max_{i,j}$ V(i ,j)

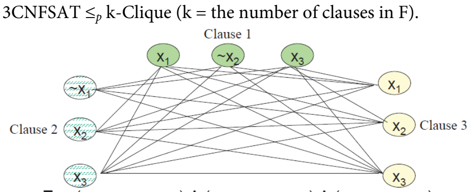
15. Smith-Waterman algorithm: for local alignment.
Its main difference to the Needleman–Wunsch algorithm is that negative scoring matrix cells are set to zero, which renders the (thus positively scoring) local alignments visible.
$V(i, 0) = V(0, j) = 0$
For $i > 0, j > 0$
$$V(i, j) = \max \{ 0,\ V(i\text{-}1, j\text{-}1)+\delta(S[i], T[j]),\ V(i\text{-}1, j)+\delta(S[i], \_),\ V(i, j\text{-}1)+\delta(\_, T[j])\}$$
Time = $O(nm)$, Space = $O(nm)$.

08 Network Flow

1. A flow network or simply network G = (V, E, c) is a directed graph in which every edge (u, v) has a capacity $c(u, v) \geq 0$.
   G has two distinguished vertices: a source s and a sink t.
2. A flow f on G assigns a real (non-negative) value to every edge in G that satisfies two constraints:
   Capacity constraint: For every edge $(u, v) \in E$, $f(u, v) \leq c(u, v)$.
   For every vertex $v \in V - \{s, t\}$, total inflow of v = total outflow of v.
3. Find maximum flow: Ford-Fulkerson method.
   1. Construct Residual network G_f
      Step 1: for every edge $(u, v) \in E$, add (u, v) (forward edge) to G_f with residual capacity = $c(u, v) - f(u, v)$.
      Step 2: for every edge $(u, v) \in E$, add (v, u) (backward edge) to G_f with residual capacity = $f(u, v)$.
      Step 3: remove all edges with 0 residual capacity
   2. Augmenting path
      Step 1: Find a path p in G_f, and compute the residual capacity rc of this path. If no such path exists, return f as the maximum flow.
      Step 2. Augment G along p as follows:
      if $(u, v) \in$ p and is a forward edge, $f'(u, v) = f(u, v) + rc$
      else if $(u, v) \in$ p and is a backward edge, $f'(u, v) = f(u, v) - rc$
      otherwise $f'(u, v) = f(u, v)$.
      Go to Step 1.
   Running time: $O(nmC)$
4. A s-t cut of G is a partition (A, B) of the vertices such that s in A and t in B. The capacity of cut (A, B) is $\text{cap}(A, B) = \Sigma_{e \text{ out of } A} c(e)$.
5. value(f) = total flow out of s = the net flow across the cut = flow from A to B minus flow from B to A = $\Sigma_{\text{in } A}$\{ total outflow of v – total inflow of v \} = $\Sigma_{e \text{ out of } A} c(e) - \Sigma_{e \text{ into } A} c(e)$.
6. $\Sigma_{v \text{ in } A}$\{ total outflow of v \} = $\Sigma_{e \text{ inside } A} f(e) + \Sigma_{e \text{ out of } A} f(e)$.
   $\Sigma_{v \text{ in } A}$\{ total inflow of v \} = $\Sigma_{e \text{ inside } A} f(e) + \Sigma_{e \text{ into } A} f(e)$.
7. Flow value lemma: value(f) = $\Sigma_{e \text{ out of } A} c(e) - \Sigma_{e \text{ into } A} c(e)$.
   Corollary 1: value(f) ≤ cap (A, B).
   Corollary 2: If value(f^*) = cap(A, B), then f^* is a max flow.
8. Let f be a flow such that G_f has no augmentation paths. Then there exists an s-t cut (A, B) such that value(f) = cap(A, B).
9. by (6. Corollary 1), maximum flow ≤ minimum cut (capacity).
   by (7), maximum flow ≥ minimum cut.
   Hence max flow = value(f) = cap(A, B) ≥ min cut.
10. Faster: Scaling algorithm:
    Let C = max capacity
    1. Δ = biggest power of 2 less than C, f = empty flow
    2. Compute $G_f(\Delta)$ = residual network with edge capacity ≥ Δ.
    3. While an augmenting path from s to t exists in $G_f(\Delta)$
       find any augmenting path (whose $rc \geq \Delta$);
       update the flow f and then $G_f(\Delta)$
    4. $\Delta = \Delta / 2$; if $\Delta \geq 1$ then repeat step 2.
    In total $(2m\log C)$ augmentations and $O(m^2 \log C)$ time.
11. Faster: Edmond & Karp Algorithm:
    Edge capacity can be arbitrarily large real number.
    1. Start with zero flow f (i.e., $f(u, v)$=0 for all $(u, v) \in E$).
    2. Repeat
       Construct the residual network G_f: forward and backward edges with residue capacity > 0.
       Find a path p with residual capacity δ from s to t in G_f
       If no such path exists, return f as the maximum flow.
       else augment the flow f with respect to p and δ.
    Note: Edmond-Karp Algorithm use BFS: the path found contains the fewest edges among all paths from s to t in G_f.
12. For Edmond & Karp Algorithm:
    Distance Lemma: Consider any f_i and f_{i+1}. For any vertex v, $\text{distance}_{f_{i+1}}(s, v) \geq \text{distance}_{f_i}(s, v)$.
    Critical Edge Lemma: For every edge (u, v) in G, (u, v) can be critical at most n/2 times; (v, u) can be critical at most n/2 times.
13. Application of max flow: Bipartite matching:
    Input: undirected, bipartite graph $G = (L, R, E)$. $M \subseteq E$ is a s matching if each node (vertex) appears in at most one edge in M.
    Max matching: find a matching with max number of edges.
    Method: Direct all edges from L to R, and assign infinite (or unit) capacity. Add a source s, and unit capacity edges from s to each node in L. Add sink t, and unit capacity edges from each node in R to t.
14. Application of max flow: k Edge-disjoint Paths
    Given directed graph G, and two nodes s and t, find k paths from s to t such that no two paths share an edge.
    Method: construct network with unit capacity, find if f is at least k.

09 NP-completeness

1. X is in NP (X is an NP problem) if there is a polynomial time algorithm to check a solution to X.
   An NP problem is said to be in P if there is a polynomial time algorithm to solve the problem.
2. Example of NP:
   Subset-Sum Problem (knapsack): Given a set A of n integers and a target integer t, find a subset of A whose sum equals t.
   Long(est) path: Find a path from s to t with total weight ≥ h.
   Note: Subset-Sum Problem is NP-complete, a dp solution is:
   A = sum of negative ints, B = sum of positive ints;
   For s = A to B
   $Q(1, s) = (x_1 == s)$
   For i = 2 to n
   $Q(i, s) = Q(i-1, s)$ or $(x_i == s)$ or $Q(i-1, s-x_i)$
   with additional condition that Q is false if s < A or s > B.
3. Polynomial-time reduction ($\leq_p$, →)
   X_1 is said to be polynomial-time reducible to X_2 ($X_1 \leq_p X_2$), if there is a polynomial time algorithm f to transform any input I_{X1} of X_1 to an input I_{X2} of X_2 such that
   -- I_{X1} has a solution <-> I_{X2} has a solution.
   Furthermore, a solution to I_{X1} can be constructed from any solution to I_{X2} in polynomial time.
4. NP-completeness:
   A search problem X is said to be NP-complete if X is in NP; and for all problems Y in NP, Y $\leq_p$ X.
   Fact: For any NP-complete problem X, if we can show that X is in P, then all problems in NP are in P (i.e., NP = P).
5. Examples of NP-complete problems: formula satisfiability, clique, vertex cover, travelling salesman problem, longest path problem, minimum cardinality key…
6. NP-completeness Lemma:
   Let A, B be two problems in NP. Suppose that A $\leq_p$ B and A is NP-complete. Then B is NP-complete.
7. The roadmap:
   SAT -> CNFSAT -> 3CNFSAT -> Clique -> Vertex Cover
   -> Subset Sum…
8. Formula Satisfiability (SAT): Given a formula F, find an assignment to satisfy F, or report false if none exists.
   CNFSAT: A formula F in conjunctive normal form. i.e. F comprises several clauses connected with ∧s, where a clause comprises literals (i.e., variables or their negations) connected with ∨s.
   The Clique Problem: Given G and an integer k, find a clique with k vertices (or else report none). A clique of G is a subset V' of V such that every pair of vertices in V' is connected with respect to E. i.e. V' is a compete subgraph.
   Vertex Cover Problem: Given a graph G and an integer h, find a vertex cover with h vertices. A vertex cover U is a subset of V where every edge in E connects to at least one of the vertices of U.
9. SAT $\leq_p$ CNFSAT:
   Step 1: Transform F to an equivalent formula F1 such that all negation operators are applied to variables only.
   By $\sim(a \wedge b) = \sim a \vee \sim b$, $\sim(a \vee b) = \sim a \wedge \sim b$, $\sim\sim a = a$.
   Step 2: Transform F1 to a CNF-formula F2 recursively.
   E.g. $[C_1 \wedge C_2 \wedge \ldots \wedge C_m] \vee [D_1 \wedge D_2 \wedge \ldots \wedge D_n]$.
   $\Rightarrow [y \vee C_1] \wedge [y \vee C_2] \wedge \ldots \wedge [y \vee C_m] \wedge [\sim y \vee D_1] \wedge [\sim y \vee D_2] \wedge \ldots \wedge [\sim y \vee D_n]$
   Example: $((x_1 \wedge \sim x_3) \vee x_2) \vee (x_1 \wedge \sim x_3) \Rightarrow$
   $(y' \vee y \vee x_1) \wedge (y' \vee y \vee \sim x_3) \wedge (y' \vee \sim y \vee x_2) \wedge (\sim y' \vee x_1) \wedge (\sim y' \vee \sim x_3)$
10. CNFSAT $\leq_p$ 3CNFSAT
    Example: $(\sim x_1 \vee x_5 \vee x_6 \vee x_8) \wedge (x_2 \vee x_8 \vee x_4 \vee \sim x_1 \vee \sim x_{11}) \Rightarrow$
    $(\sim x_1 \vee x_5 \vee y) \wedge (\sim y \vee x_6 \vee x_8) \wedge$
    $(x_2 \vee x_8 \vee y') \wedge (\sim y' \vee x_4 \vee \sim x_1 \vee \sim x_{11})$
    Note: Both 1SAT and 2SAT are in P.
11. 3CNFSAT $\leq_p$ k-Clique (k = the number of clauses in F).



F = $(x_1 \vee \sim x_2 \vee x_3) \wedge (\sim x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

12. k-Clique $\leq_p$ (n-k)-Vertex Cover.
    $G = (V, E)$, $n = |V|$. Construct G' = Complete Graph - G.
    If G' has a vertex cover U with n - k vertices, then $V - U$ is a k-clique of G.
13. 3SAT $\leq_p$ Subset Sum: t is the constraint of number of picks

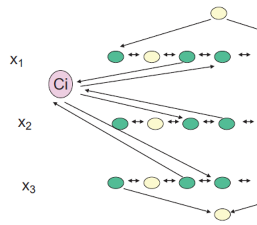
(x1 or x2 or ~x3) and (~x1 or x2 or x3)

	3 variables			Clause 1	Clause 2	
a_1 (x_1)	1	0	0	1	0	
a_2 ($\sim x_1$)	1	0	0	0	1	Each column contains 2 or 5 ones.
a_3 (x_2)	0	1	0	1	1	
a_4 ($\sim x_2$)	0	1	0	0	0	
a_5 (x_3)	0	0	1	0	1	
a_6 ($\sim x_3$)	0	0	1	1	0	
a_7	0	0	0	1	0	
a_8	0	0	0	1	0	这四个用来弥补最少只选一个Ture的情况
a_9	0	0	0	0	1	
a_{10}	0	0	0	0	1	
t	1	1	1	3	3	

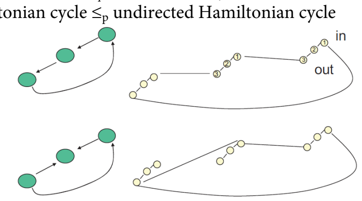
14. Subset Sum $\leq_p$ Equal-sum Partition
    Let sum = $a_1+a_2+\ldots+a_n$.
    If t = sum/2, then a equal-sum partition of $a_1, a_2 \ldots a_n$ is a solution to knapsack of $a_1+a_2+\ldots+a_n$;
    If t < sum/2, then A = C ∪ \{ a_{n+1} = sum - 2t\};
    If t > sum/2, then A = C ∪ \{ a_{n+1} = 2t - sum\}.
15. Subset Sum $\leq_p$ General Knapsack
    General knapsack: Given n items with (integer) weights w_1 to w_n and (integer) values v_1 to v_n, and a knapsack of capacity W, and a goal g, find a collection of items such that their total weight is at most W and their total values is at least g.
    The idea: $w_i = v_i = a_i$. $W = g = t$.
16. 3SAT $\leq_p$ Hamiltonian Path
    Hamiltonian (Rudrata) path: a simple path visiting every vertex of G exactly once.
    Let F be a 3CNF formula with n variables and m clauses.
    Basic structure of the graph G: one row for each variable
    - 2^n different assignment to x_i's ⇔ 2^n different Hamiltonian paths.
      True = left to right; false = right to left.

    Each row contains 3m+3 vertices.

Each clause defines an extra node. E.g., Ci = $(x_1 \vee \sim x_2 \vee x_3)$



17. Hamiltonian path $\leq_p$ Hamiltonian cycle. Connect s and t.
18. Hamiltonian cycle $\leq_p$ undirected Hamiltonian cycle



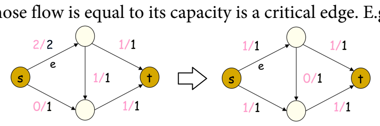
19. Euler path/cycle: a path/cycle visiting every edge exactly once
20. (directed) Hamiltonian path $\leq_p$ Longest path
    Edge weight = 1, threshold = |V|-1. A Hamiltonian path exists if and only if a path with total weight at least |V|-1 exists.
21. (undirected) Hamiltonian cycle $\leq_p$ travelling salesman problem
    TSP: Given an integer k and a complete undirected graph G that has a non-negative cost $c(u, v)$ associated with each edge, find a Hamiltonian cycle with total cost at most k.
22. Optimization problem: Find the best solution.
    An approximation algorithm refers to an algorithm that may not find the optimal solution (smallest vertex cover), but can find a solution with a certain guarantee even in the worst case.
    Question F is solved instead of F*.
    Approximation ratio = (worst-case ratio of F)/F*.
23. Vertex Cover Approximation:
    Algorithm:
    $G = (V, E)$, VC = empty set, $E' = E$.
    While E' is not empty
      pick any edge (u, v) in E'
      Add vertexes u and v into VC
      Remove every edge incident on u or v from E'
    Return VC
    Analysis: AR = 2. Let VC* be the smallest vertex cover. To cover the edge (u, v), VC* must include at least u or v. Thus VC* includes at least half of the vertices in VC.
24. TSP Approximation:
    TSP: For an integer k and a complete undirected graph G that has a non-negative cost $c(u, v)$ associated with each edge, find a Hamiltonian cycl with total cost at most k.
    Optimization problem: a Hamiltonian cycle with minimum cost.
    Method: Use triangle inequality. Compute the MST T. Pick a node in T as the root and perform a preorder traversal of T. Let L be the list of vertices visited. Skip duplicate vertices on L and the resulting list is a Hamiltonian cycle. Let C be the Hamiltonian cycle with the minimum cost. Removing an edge from C defines a spanning tree of G. cost(C) ≥ cost(T), where T is the minimum spanning tree. A preorder traversal visits every edge of T exactly twice, therefore cost (L) = 2 cost(T).
    To obtain a Hamiltonian cycle, we skip the duplicate vertexes in L, but this can't increase the total cost (triangle inequality).
    Conclusion: The resulting Hamiltonian cycle has a total cost at most 2 cost(C).
25. HC $\leq_p$ TSP (AR = 2)
    Construct an instance of the TSP as follows. G' is a complete graph over vertex set V. For any vertices u, v, define $c(u, v)$ = 1 if (u, v) is in E, otherwise $c(u, v)$ = 2|V| + 1. If G has a Hamiltonian cycle, then G' contains a tour with total cost = |V|.
26. Approximation on Knapsack problem.
27. NP-hard: Let X be an NP search problem, and let Y be the corresponding optimization problem. Assume that X is NP-complete. We call Y NP-hard. (If a polynomial time algorithm for Y exists, we can solve X in polynomial time, and P = NP.)

画flow的时候，中性笔画正反全部边，铅笔写值。
不用每次都更新原图f，最后更新即可。

Graphs, Network flows, NPs

usually in graphs, $n = |V|$ and $m = |E|$

1. bipartite: use DFS to set color for each node. If visited, check if color is the same, if same then exit. If not visited, set the other color and continue DFS.
2. To prove NP-completeness, we need (a) A and B are NP and A is NP -complete (b) A is polynomial-time reducible to B (c) solution to A <=> solution to B.
3. Prove that a graph with n vertices and n edges must has a cycle => The number of edges in a tree on n vertices is $n-1$. We can prove this by MI.
4. Given an undirected connected graph, show how to find a path that goes through each edge exactly once in each direction: build a DFS.
5. Maximum spanning tree: assign negative weight => Minumun ST.
6. For Dijkstra's Algorithm, time complexity is $O(ma + nb)$, where a is complexities of the decrease-key and b is complexities of the extract-minimum. With a self-balancing binary search tree or binary heap, $a = b = \log n$ in the worst case. The Fibonacci heap improves this to $a = 1$, $b = \log n$, and if the graph is dense, time complexity is approximately $O(n^2)$.
7. For Prim's algorithm, build heap cost $O(n)$, we need at most m decrease key operations, using $O(m \log n)$ time. Hence total time is $O(m \log n)$ suppose $n \leq m$. (without this assumption $O((m+n)\log n)$). For Kruskal's algorithm, merge cost O(n), insert cost O(1), check cycle cost $O(\log n)$. As there are m query, in total $O(n+m\log n)$.
8. Largest number of edges in a graph with n vertices and at least one cut-vertex: Suppose removing of a vertex u cut the graph into 2 connected components, one with k vertices and another with n-1-k vertices. Then the maximum number of original edges (respect to some k) is: $F(k) = \frac{1}{2}(k-1)(k-2) + \frac{1}{2}(n-k-1)(n-k-2) + n-1$. When $k = 1$ or n-2, $F(k)$ reaches maximum value.
9. Find shortest cycle in directed graph $O(n^3)$:
   Algorithm 1: For each vertex u, use Dijkstra's algorithm to find the cost of shorted path between u and each other vertex v, stored as $s[u][v]$ ($s[u][v]$ left $+\infty$ if no path from u to v). Then the cost of the shortest cycle that contains (u, v) is $s[u][v] + s[v][u]$. Iterate on each pair (u, v) to get the shortest cycle of the graph, and return no cycle if all $s[u][v] + s[v][u]$ are zeroes.
   Algorithm 2: run Floyd-Warshall. For all pairs, find minium of $s[u][v] + s[v][u]$. $O(n^3) + O(n^2) = O(n^3)$.
10. MST is unique if all edges are distinct: prove by assume two MSTs, let e in A not in B, e separate A into two conected components C_1 and C_2, let such edge that separate C_1 and C_2 in B be e'. If e is the lightest edge across C_1 and C_2, then swap (e, e') will generate a ST lighter than B; else swao (e, lightest edge across C_1 and C_2) will generate a ST lighter than A. Contradiction.
11. Second-best MST is not unique.
    If a SB-MST exists, it differs from the MST by one edge.
    Proof: assume differs by two edges e_1 and e_2, by removing e_1 we obtain two conected components C_1 and C_2. By swap (lightest edge acrooss C_1 and C_2, e_1) we obtain a tree with less weight than the second best MST but more weight than MST because of e_2. Therefore it is not the SB-MST, contradiction.
    Algorithm to find SB-MST:
    For each edge e in (E - MST)
      find the cycle created by adding e to T
      find edge e_{max} with max edge on this cycle
      record min(w(e) - w(e_{max})) among all edges e, may not be unique
    return the tree MST - e_{max} + e for pair found above.
12. find an odd-length cycle in a strongly connected directed graph:
    use -1, 0, 1 flag: -1 means not visited, 0 and 1 mark odd or even.
    Algorithm: Apply find cycle method in directed graph, but
    (a) in the first line of dfs, apart form set start and finish, set
        visited[current] = visited [previous] == 0 ? 1: 0
    (b) dfs has two arguments (current, previous).
        with in the dfs, it should further search dfs (next, current).
13. An undirected graph G has a cycle containing $e = (u, v)$ if and only if there is a path from u to v that does not contain edge e.
14. Blood type: source to supplies to demands to sink. supplies to demands are assigned with infinite weight if biologically possible.
15. An edge of a flow network G is called critical if decreasing the capacity of this edge results in a decrease in the maximum flow.
    (a) It is false that with respect to a maximum flow of G, any edge whose flow is equal to its capacity is a critical edge. E.g.



    (b) Find a critical edge: Let e be a saturated (full capacity) edge, G be the original nextwork and G_f be the residual network. If there is a cycle that passes through reverse e in G_f, we can reduce flow in the cycle while maintaining the maximum flow not decreased.
    Algorithm: for each saturated edge (u, v) in G, run DFS to check whether there is a path from u to v in G_f. If there is a path, (u, v) is not critical, otherwise it is.
16. Let G be a flow network with m edges. Suppose a maximum flow of G is given, of which the value is x, then:
    (a) this maximum flow can be decomposed into ≤ m paths.
        proof: Let e be an edge in a path P from s to t, $f(e) > 0$. Find the edge with the minimum flow in P, denoted by $f_{min}(P)$. Subtract $f_{min}(P)$ from every edge in P. At least one edge will have zero flow after subtraction. So this process can be repeated no more than m times. Therefore the max flow can be decomposed to ≤ m paths.
    (b) there exists such a path carrying a flow with value at least x=m.
        proof: Pigeonhole principle

18. If there are multiple sinks/sources, combine them. For example, if there are two sinks with demand at least d_1 and d_2, we construct a new sink and set $cap(t_i, t) = d_i$. Note that if we want a strict demand larger than d_1+d_2, we can first set such capacity and run algorithm, then consider if we can still find a flow in s-t_1 or s-t_2.
19. Network with lower bound:
    Create G' as follows:
    1. Create 2 new vertices s' and t'
    2. for each edge (u, v) with lower bound l and capacity c in G:
       i. create edge (s', v) with capacity l to represent a required flow into v
       ii. create edge (u, t') with capacity l to represent a requited flow from u
       iii. remore the lower bound from (u, v) and replace the capacity with $(c\text{-}l)$
       If any of above edge already exists, the capacity is then added to the existing edge.
    3. Create edge (t, s) with infinite capacity.
    Find max flow from s' to t' in G'.
    Check whether max flow is equal to sum of required flow which are all lower bounds in this case. All edges with flow f in G' are moved back to G with flow $(f + l)$.
20. Cut-vertex:
    (1) the vertex is a root in a DSF, with more than one tree edge (children), or
    (2) the vertex is not root in a DSF, but all its descendents have no back edges towards its ancestors.
    Cut edge:
    a tree edge (u, v) that v and all descendant of v do not have back edge to u or ancestors of u
21. A matching or independent edge set in a graph is a set of edges without common vertices.

Do not forget about the base cases! ==> Divide and Conquers, DPs, Greedys

1. Solution to $T(n = \sqrt{n}T(\sqrt{n} + n$ is $O(n\mathrm{loglog}n$
2. Prove if an $n \times n$ matrix can be squared in time $O(n^c$, then any two $n \times n$ matrices can be multiplied $O(n^c$ time: Consider M = {{B, 0}, {A, 0}} and square M.
3. Definition of divide and conquer:
    1. divide the problem into subproblems;
    2. solve the subproblems recursively;
    3. combine the solutions of the subproblems into a solution of the original problem.
4. Huffman endoding; longest possible encoding (codeword) of a character: set $f_{n-i} = 2^{-i}$ for i from 1 to (n-1), the longest codeword will have length (n−1).
5. limitted supply knapsack:
   $dp[i+1][j] = \max(dp[i][j], dp[i][j-w[i]]+v[i])$; (if $j \geq w[i]$)
   unlimited supply knapsack:
   $dp[i+1][j] = \max(dp[i][j], dp[i+1][j-w[i]]+v[i])$; (if $j \geq w[i]$)
6. 3 subset sum:
   Algorithm: Set bool sum[n][U/3][U/3] = {false}, each entry [i][s_1][s_2] denotes whether a_1 to a_i can form 2 disjoint subsets with sum equal to s_1 and s_2.

```
For i = 1 to n
    For m = a_i to U/3
        For n = a_i to U/3
            sum[i][m][n] = sum[i-1][m][n]
                || (if m-a_i > 0 then sum[i-1][m-a_i][n])
                || (if n-a_i > 0 then sum[i-1][m][n-a_i])
return sum[n][U/3][U/3].
```