Asymptotic Limit Rule:-

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} <\infty & O(g(n)) \\ = c & \Theta(g(n)) \\ >0 & \Omega(g(n)) \end{cases}$$

Master's Theorem

$T(n) = aT(n/b) + O(n^d)$

$a>0,\ b>1,\ d\geq 0$

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

- Branching Factor = a
- Depth of Tree = $\log_b n$
- Width of Tree = $a^{\log_b n} = n^{\log_b a}$

FFT ($O(n \log n)$):-

The Fast Fourier Transform in the scope of our knowledge thus far is used for Polynomial Multiplication. We want to know the coefficient of $(p \cdot q)(x)$ knowing only the coefficients of $p(x)$ and $q(x)$. The naive way is the one we all use in Algebra which runs in $O(n^2)$. Here is a faster algorithm:

1. Take the coefficients of $p(x)$ and $q(x)$ and plug them into the FFT with the Roots of Unity (We need $2n+1$ points). $O(n \log n)$
2. From the FFT we get the roots of unity evaluated at $p(x)$ and $q(x)$ and we multiply the respected values from each root together to get $2n+1$ respective pairs of values of $(p \cdot q)(x)$. $O(n)$
3. Finally we interpolate these points using the inverse FFT to get the coefficients of $(p \cdot q)(x)$. $O(n \log n)$

Evaluation: $< values > = \text{FFT}(< coeff >, w)$
Interpolation: $< coeff > = (1/n)\ \text{FFT}(< values >, w^{-1})$

```
function FFT(A, w)
Input: Coefficient representation of a polynomial A(x)
of degree less than or equal to n1,
where n is a power of 2w, an nth root of unity
Output: Value representation A(w_0),...,A(w_n1)

if w = 1: return A(1)
express A(x) in the form A_e(x^2) + xA_o(x^2)
call FFT(A_e, w^2) to evaluate A_e at even powers of w
call FFT(A_o, w^2) to evaluate A_o at even powers of w
for j = 0 to n  1:
   compute A(w_j) = A_e(w^2j) + w^j A_o(w^2j)
return A(w_0),...,A(w_n1)
```

This process happens recursively.

- undirected graph does not have cross edges

$a < \log n < n^a < n! < n^n$

$2^{\log_2 n} = n$

$$M_n(w) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{n-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & w^{n-1} & w^{2(n-1)} & \cdots & w^{(n-1)(n-1)} \end{bmatrix}$$

Search Algorithms

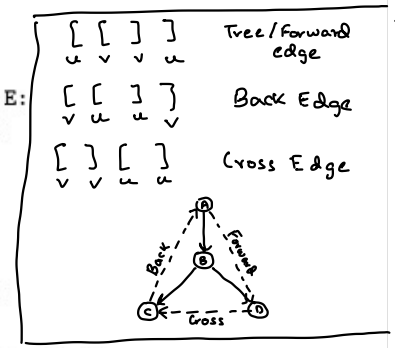
DFS ($O(V+E)$) [Stack] – LIFO

- Explores all the way down to a tree, then climbs back up and explores alt. paths. Used for Topological Sort and Finding Connected components.

($pre(u) < pre(v) < post(v) < post(u)$)

```
def explore(G,v): #Where G = (V,E) of a Graph
Input: G = (V,E) is a graph; v  V
Output: visited(u) is set to true for all nodes u
reachable from v
    visited(v) = true
    previsit(v)
    for each edge(v,u) in E:
        if not visited(u):
            explore(u)
    postvisit(v)

def dfs(G):
    for all v in V:
        if not visited(v):
            explore(v)
```



Previsit = count till node added to the queue
Postvisit = count till you leave the given node

- Directed graph has a cycle iff it has a back edge found during dfs.

Strongly Connected Components $O(n)$

Use for most edge removal problems, use this to check if still strongly connected.

Properties:

- If the explore subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.
- The node that receives the highest post number in a depth-first search must lie in a source strongly connected component.
- If C and C are strongly connected components, and there is an edge from a node in C to a node in C, then the highest post number in C is bigger than the highest post number in C.

- Big Picture: Topologically sort the graph; reverse the edge; Topologically sort again; if we reach a new source the resulting transversal is new SCC; continue till end of list.
- Algorithm: ① Run dfs on G. ② Run the undirected CC algorithm, during dfs process the vertices in dec order of post no. from ①.

Topological Sorting $O(V+E)$

Constraints: A directed graph G is acyclic if and only if a dfs of G yields no back edges. Formally we say a topologically sort of directed acyclic graph G is an ordering ther vertices of G such that every edge (v_i, v_j) of G we have $i<j$. If DAG is cyclic then no linear ordering is possible.

Topological Sort: Returns a list with all nodes pointing left such that basically all parents come before any any children (excluding sources). We order a graph from the highest post number in a decreasing order. Thus we create singly connected components from a DAG with several strongly connected components, each a unique source and unique sink in the DAG. There are multiple topological sorting possible. Used for Runtime compiling or Scheduling.

BFS $O(V+E)$ [Queue] – FIFO

```
Input: Graph G = (V, E), directed or undirected; vertex s  V
Output: For all vertices u reachable from s, dist(u) is set
to the distance from s to u.

def bfs(G,s):
    for all u in V:
        dist(u) = infinity
    dist(s) = 0
    Q = [s] (Queue containing just s)
    while Q is not empty:
        u = eject(u)
        for all edges (u,v) in E:
            if dist(v) = infinity:
                inject(Q,v)
                dist(v) = dist(u) + 1
```

Dijkstra's Algorithm $O((V+E)\log V)$ [Binary Heap]

Objective: To find the shortest path

```
def dijkstra(G,l,s):
    for all u in V:
        dist(u) = infinity
        prev(u) = nil
    dist(s) = 0
    H = makequeue(V) # using dist values as keys
    while H is not empty:
        u = deletemin(H)
        for all edges (u,v) in E:
            if dist(v) > dist(u)+l(u,v)
                dist(v) = dist(u)+l(u,v)
                prev(v) = u
                decreasekey(H,v)
```

n-V
m-E

deletemin(H) — $O(n)$ worst

decreasekey — $O(n)$ worst case

Dij on tree with -ve edge ✓
Dij on DAG, graph with -ve edge Sometimes
Dij on -ve cycle – never

Bellman Ford Algorithm - O(V·E)

Objective: Find shortest path allowing for -ve edges.

```
procedure shortest-paths(G, l, s)
Input: Directed graph G = (V, E);
edge lengths {l_e: e in E} with no negative cycles;
vertex s in V
Output: For all vertices u reachable from s,
dist(u) is set to the distance from s to u.
for all u in V:
    dist(u) = infinity
    prev(u) = nil

dist(s) = 0
repeat |V|-1 times:
    for all e in E:
        update(e)
```

4th root!

$M_4(w) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$

$M_2(w) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

$w = e^{\frac{2\pi i}{n}}$

Directed Acyclic Graph (DAG):-

- Every DAG has a source and sink
- A directed graph has a cycle if and only if its depth-first search reveals a back edge.
- In a DAG, every edge leads to a vertex with a lower post number.
- Every directed graph is a DAG of its strongly connected components.
- If the explore subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited
- The node that receives the highest post number in a depth-first search must lie in a source strongly connected component
- If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', the the highest post number in C is bigger than the highest post number in C'.
- In any path of a DAG, the vertices appear in an increasing linearized order allowing you to run Dijkstra's Algorithm in $O(n)$

Greedy Algorithm:-

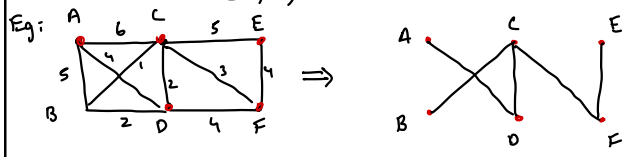
- Always takes the cheapest least weight edge for its next step.
- MST is tree (acyclic) with |V-1| edges
- Fringe is all the vertices exactly one hop from our current vertex

Krushal's MST Algorithm O(E log E):-

Sort edges using a sorting algorithm then repeatedly add the next lightest edge that doesn't produce a cycle. (i.e only visit new vertices)

```
for all u in V:
    makeset(u)
X = { }
Sort the edges E by weight
for all edges {u,v} in E, in inc order of weight:
    if find(u) != find(v):
        add edge {u,v} to X
        union(u,v)
```

Eg:



Cut Property

Suppose edges X are part of a MST of G=(V,E). Pick any subset of nodes S for which X doesn't cross between S and V-S, and let e be the lightest edge across the partition. Then X ∪ e is a part of some MST.

Prim's Algorithm O(E log E):-

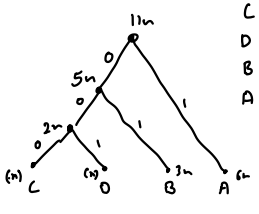
Objective: To find the MST Alt to Krushal's. Similar to Dijkstra's

```
procedure prim(G, w)
Input: A connected undirected graph G = (V, E) with weights
Output: A minimum spanning tree defined by the array prev
for all u in V :
    cost(u) = infinity
    prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0
H = makequeue (V) (priority queue with cost-values as keys)
while H is not empty:
    v = deletemin(H)
        for each {v, z} in E:
            if cost(z) > w(v, z):
                cost(z) = w(v, z)
                prev(z) = v
                decreasekey(H, z)
```

Huffman Encoding:- O(n log n)

A means to encode data using the optimal number of bits for each character given a distribution.

6n's A ; 3n's B ; n's C ; n's D



C = 000 (3)
D = 001 (3)
B = 01 (2)
A = 1 (1)

Total len = 1×6n + 2×3n + 3×n + 3×n
= 6n + 6n + 6n
= 18n

Disjoint Sets Data Structure:-

Contains a function "find" that returns the root a given set pi refers to the parent node. rank refers to the height subtree hanging from that node For any x, rank(x) < rank(π(x)). Any root node of rank K has at least 2^K nodes in its tree. If there are n elements overall, there can be atmost $n/2^K$ nodes of rank K. The max rank is log n.

```
def makeset(x): // O(1)
    pi(x) = x
    rank(x) = 0

def find(x): // O(E log V)
    while x != pi(x):
        x=pi(x)
    return x

def union(x,y): // O(E log V)
    if find(x) == find(y):
        return
    elif rank(find(x)) > rank(find(y)):
        pi(find(y)) = find(x)
    else:
        pi(find(x))=find(y)
        if rank(find(x)) == rank(find(y)):
            rank(find(y)) = rank(find(y)) + 1
```

Union Find:- (Uses disjoint sets data structure)

Runs in per operation O(log n) which is the number of times you can take a log of n before it becomes 1 or less. It is very slow and for all practical cases is const.

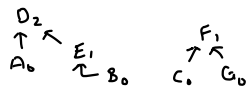
Basically if find(x) and find(y) return the same value they are in the same graph so do nothing, else add edge. Then union(x,y).

Union: Worst case is O(log n). Avg for all ops is O(n log n). (n is no. of elements in Data structure)

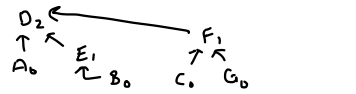
Eg: $A_0, B_0, C_0, D_0, E_0, F_0, G_0$

union(A,D); union(B,E); union(C,F) → D_1 E_1 F_1 / A_0 B_0 C_0 G_0

union(C,G); union(E,A) →



union(B,G) →



Horns Formula:-

Variable set to T by greedy cannot be changed to F. Vica-versa possible

- Start by all variables as F, then proceed to set some of them as T one by one only if we absolutely have to because an implication would otherwise be violated.

Once done with this and all implications are satisfied only then we go to negative clause and satisfy it.

Dynamic Programming:-

Brute force the solutions to a problem by turning it into smaller and smaller nested subproblem that remember useful info about its parent subproblem so that it can eventually reconstruct itself to solve it. 2 Methods of solving the problem: ① Top-down (memoization) ② Bottom-up (iteration).

- TD: Recursive idea of breaking the prob into smaller subprob and finding way to find max/min of every permutation.
- BU: Oppo approach of TD of breaking down to smallest SP and iteratively solve the parent SP. BU > TD: Space complexity

Steps to Solve DP

1. Define the subproblems. Know the # of subproblems.
2. Guess a solution for what is not the subproblem (max/min of brute force permutations). Know the # of guesses.
3. Relate the subproblems to the parent problem. This is the recursive/iterative step.
4. Do the recursion and memoize or iteratively build a table to keep track of previously used values. These can be used to form a DAG. Ensure the DAG for these are acyclic (i.e. have valid topological order or no dependences on parent problems)
5. Solve the original problem ($\theta(\#subproblems * time/subproblem)$)

Choosing the Subproblem:-

Input	Subproblem	
① Sequence a[1…n]	cost (a[1…i])	cost (a[i…j]) — no. of sp is linear
② Weighted graph G=(V,E,w)	cost (v) v∈V	no. of sp: O(mn)
③ 2 Seq a[1…n], b[1…m]	cost (a[1…i], b[1…j])	
④ Tree	cost (subtree)	

Shortest Path:- Runtime: Θ(VE)

- For DAG: dist(v) as length of shortest path from s to v. Order of subproblem: Topological order. Base case: dist(v)=0 dist(v) = ∞ if v ≠ s is a source.

$$dist(v) = \min_{(u,v)\in E} \{dist(u) + l(u,v)\}$$

- General: $S_k(s,v)$: weight of shortest path from s to v that uses ≤ k edges.

$$S_k(s,v) = \min_{(u,v)\in E}(S_{k-1}(s,u) + w(u,v))$$

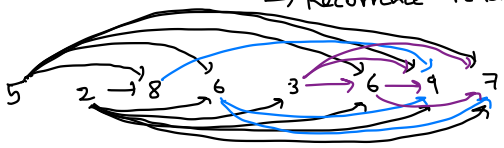
→ Bellman Ford

Longest Increasing Subsequence : O(n²)

Starts at one side of the list and find max length of seq ending at given node.

```
L = {}
for j = 1 to n:
    L[j] = 1 + max{L[i] : (i,j) in E}
return max(L)
```

→ Recurrence relation



Edit Distance :- Space: O(mn) Time: O(mn)

Works by basically choosing the min of the operations for every given letter. 3 options: adding gap '-' or matching the string and moving on.

Subproblem: E(i,j) → final object compute E(m,n)
↳ x[1…i], y[1…j]

Recurrence:-

$$E(i,j) = \min\{1 + E(i-1,j),\ 1 + E(i,j-1),\ diff(i,j) + E(i-1,j-1)\}$$

diff: 0 if x[i] = y[j], 1 otherwise

Knapsack : O(nW)

- With Repetition:- K(0) = 0

Subproblem: K(w) = max value achievable with Knapsack of w.

Recurrence: $K(w) = \max_{i:w_i \le w}\{K(w-w_i) + v_i\}$

- Without repetition:- K(0,j) = 0 and all K(w,0) = 0

SP: K(w,j): Max value with Knapsack w and items 1…j

$$R: K(w,j) = \max\{K(w-w_j, j-1) + v_j,\ K(w, j-1)\}$$

Parenthesization O(n³) → Matrix

$$C(i,j) = \min\{C(i,k) + C(k+1,j) + m_{i-1}\cdot m_k \cdot m_j\}$$

Shortest path : dist(v,i) : len of shortest from s to v that uses i edges

$$dist(v,i) = \min_{(u,v)\in E}\{dist(u,i-1) + l(u,v)\}$$

Floyd-Warshall : O(|V|³)

Used for finding shortest path in weighted graph with +ve or -ve but no negative cycles. len from i to j in which only nodes {1…k}

$$R: dist(i,j,k) = \min\{dist(i,k,k-1) + dist(k,j,k-1) + dist(i,j,k-1)\}$$

```
for i=1 to n:
    for j=1 to n:
        dist(i,j,0) = ∞
    for all (i,j) in E:
        dist(i,j,0) = l(i,j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
```

Fails when -ve cycles

TSP [O(n² 2ⁿ)]:-

Starting at 1 to j.

SP: C(S,j): len of shortest path visiting each node in S exactly 1

$$R: C(S,j) = \min_{i\in S, j\neq i} C(S-\{j\}, i) + d_{ij}$$

$C(\{1\},1) = 0$
for $s = 2$ to n:
  for all subsets $S \subseteq \{1,2,\ldots,n\}$ of size s and containing 1:
    $C(S,1) = \infty$
    for all $j \in S, j \neq 1$:
      $C(S,j) = \min\{C(S-\{j\},i) + d_{ij} : i \in S, i \neq j\}$
return $\min_j C(\{1,\ldots,n\},j) + d_{j1}$

Independent Sets in Trees O(V+E)

SP: I(u): size of the largest independent set of subtree hanging from u.

$$R: I(u) = \max\{1 + \sum_{\text{grandchild w of u}} I(w),\ \sum_{\text{child w of u}} I(w)\}$$

Linear Programming:-

- Problem as an objective func with set of linear constraints.
- If a LP has bounded optimum, then so does it's dual.
- If a LP has an unbounded value then its dual must be infeasible

Primal: max $c^T x$ s.t. $Ax \le b$, $x \ge 0$

Dual: min $y^T b$ s.t. $y^T A \ge c^T$, $y \ge 0$

Changing Objective	Inequality to Equality
max $c^T x$ = min $-c^T x$	$ax \le b \longrightarrow ax + s = b$, $s \ge 0$
min $c^T x$ = max $-c^T x$	
Equality to Inequality	**Unrestricted Variable**
$ax = b \longrightarrow ax \le b, ax \ge b$	$x \in R \longrightarrow x = x^+ - x^-$, $x^+, x^- \ge 0$

Simplex: Typically poly runtime, Worst case: exp runtime

If LP has min $c^T x$ s.t. $Ax \ge b$, $x \ge 0$ has a feasible solution with objective value M, then every feasible point of its dual has an obj value at most M.

→ Set of feasible flows is a convex set.

Max Flow

- Graph G with a source (s) and sink (t). Residual graph with forward edge for amt of capacity that is not being used (capacity - current used), back edge for what is currently being used.

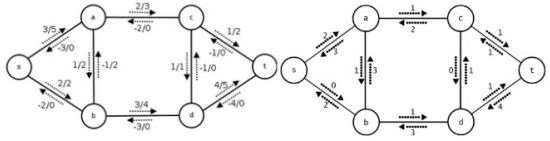
$r(u,v) = c(u,v) - f(u,v)$ if $(u,v)\in E$ & $f(u,v) < c(u,v)$
$r(u,v) = c(u,v) + f(v,u)$ if $(v,u)\in E$ & $f(v,u) > 0$

Runtime: O(|V|·|E|²) if used BFS

Ford-Fulkerson

Start with no flow on your graph. Find a path using DFS. Then create a residual graph. We can now use DFS for finding a path from s to t in the residual graph. If one exists we are not optimally using our flow. We then find the edge with the LEAST capacity edge - this is our bottleneck - and add flow onto all the edges in that path up to the capacity that is not being used by the bottleneck edge, hereby maximizing the flow of the path. Our new flow is guaranteed to be better. Create a new residual graph and repeat until no path in the residual graph can be found from s to t. This will happen when capacity = current use, as we lose our forward edge and only have a back edge.

This algorithm will sometimes decrease flow in one area to increase flow in another. Max flow is the total weights of incoming edges to the sink. Runtime would be $O(E * M)$, where M is the number of iterations and E is the time to find a path. This can also be stated as $O(\text{maxflow} * E)$. However this may not terminate. If you use BFS instead of DFS you are using the Edmonds-Karp Algorithm which will terminate and has complexity $O(V * (E)^2)$, which is better for large networks.



Max Flow Min Cut Thrm :-

The size of the max flow in network equals the capacity of the smallest (S,t)-cut, where and (S,t)-cut partitions the vertices into 2 disjoint groups L & R s.t. s is in L, t is in R.

Bipartite Matching :-

List of Bs, List of Gs, if a B likes G, a direct edge exists from B to G. Is there a perfect matching? Create s and t nodes, s has outgoing edges to all the boys and t is incoming edge from the girls. Every edge has capacity 1. Flow exists if there is a flow into t with size equal to no. of couples.

Zero Sum Games :-

Setup: Given a matrix where the rows are player A's moves, the column are player B's moves, and the values are A's reward for each move. Which ever player goes 1st has to announce their strategy 1st.

Stratergy: Vector prob of playing specific moves.

Main Idea: If A goes 1st, B will pick the move that min A's reward. So A's best stratergy will be max of min of B's moves. If B goes 1st, A will pick the move that max A's reward. So B's best is to min of max of A's reward.

Ex:
$$A \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$ (B)

A's stratergy: $[x_1, x_2]$

B's stratergy: $[y_1, y_2]$

A: picks $[x_1, x_2]$ to maximize value of $\min\{ax_1 + bx_2, cx_1 + dx_2\}$

B: picks $[y_1, y_2]$ to minimize value of $\max\{ay_1 + cy_2, by_1 + dy_2\}$

Min-Max Thm : $\max_x \min_y \sum_{i,j} G_{ij} x_i y_j = \min_y \max_x \sum_{i,j} G_{ij} x_i y_j$

Eg: (Row \ Col)
$$\begin{bmatrix} 4 & -3 \\ -2 & 1 \end{bmatrix}$$

Exp payoff if Col plays (x_1, x_2): $\max(4x_1 - 3x_2, -2x_1 + x_2)$

Exp payoff if R plays (y_1, y_2): $\min(4y_1 - 2y_2, -3y_1 + y_2)$

Eg: (S)
$$\begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 3 & 3 & 0 \end{bmatrix}$$

LP for S optimal :- Maximize z

subject to:
$$2b + 3c \geq z$$
$$2a + 3c \geq z$$
$$2a + 2b \geq z$$
$$a + b + c = 1$$
$$a, b, c \geq 0$$

Zero Sum can be deterministic as optimal stratergy.

P: Set of all search probs solvable in poly time (includes P)

NP: Set of all search prob whose soln can be checked in poly time.

NPH: Any prob in NP can be reduced in poly time to this. Most NPH are not in NP, but those are, they are NPC.

NPC. NP & NPH.
↳ if all other search prob reduces to it

$A \longrightarrow B$: A reduces to B
- diff flow of arrow
- efficiency opp flow

$A \to B$, $B \to C$ then $A \to C$

NPC Prob: Factoring, Rudrata path, Rudrata cycle, 3SAT, Independent Set,

Factoring $\longrightarrow$ Circuit SAT

3SAT $\longrightarrow$ Rudrata cycle $\longrightarrow$ 

Rudrata Path $\Longleftrightarrow$ Cycle: Add vertex x connecting s and t then run rudrata cycle, then remove x, we get the path

3SAT $\longrightarrow$ Independent Set: For each clause in 3SAT make a Δ edge connected, put an edge bet oppo literals in diff clauses.

$(\bar{x} \vee y \vee \bar{z})\ (x \vee \bar{y} \vee z) \longrightarrow$ 

goal = no. of clauses

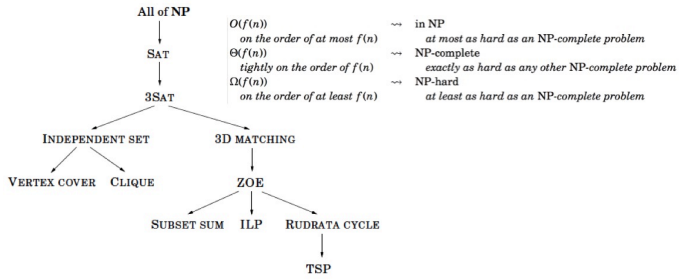
SAT $\longrightarrow$ 3SAT: For all the literals $(a_1 \vee a_2 \vee \dots a_k)$ make a 3 pair clause if mul of 3 else add oppo literals in clause like $(a_1 \vee a_2 \vee y_1)\ (\bar{y}_1 \vee a_3 \vee y_2)\ (\bar{y}_2 \vee a_4 \vee y_3) \dots$

Independ $\longrightarrow$ Vertex Cover: Look for vertex cover of G with $|V| - g$ nodes. If VC exists take all nodes not in it. If no VC, G has no IS.

3SAT $\longrightarrow$ 3D Matching: Creating a copy of preceding gadget for each variable x. Call the resulting nodes $p_{x_1}, b_{x_0}, g_{x_1} \dots$ b_{x_0} matches with g_{x_1} if $x = T$ and g_{x_0} if $x = F$. For each clause intro a new boy and a girl. They will involve in 3 triplets, one of each literals.

Rudrata Cycle $\longrightarrow$ TSP: TSP is cities with dist bet them 1 if $\{u, v\}$ edge in G and $1 + a$ otherwise. If G has a rudrata cycle then the same cycle is TSP.

All prob in NP $\longrightarrow$ SAT (Circuit SAT): Known input is bits of I, unknown input is bits of S, such that the output is T iff unknown inputs spell a soln S of I.



Common NP-Complete Problems

Hard problems (NP-complete)	Easy problems (in P)
3SAT	2SAT, HORN SAT
Traveling Salesman Problem	Minimum Spanning Tree
Longest Path	Shortest Path
3D Matching	Bipartite Matching
Knapsack	Unary Knapsack
Independent Set	Independent Set on trees
Integer Linear Programming	Linear Programming
Rudrata Path	Euler Path
Balanced Cut	Minimum Cut

ILP: Solve a prob using linear obj func and linear ineq, while constraining the values of variables to integers.

Approximation Ratio :- $OPT(I)$: Optimum soln to the instance. $A(I)$ is soln by our method.

For minimization: $\alpha_A = \max_{(I)} \text{of } \frac{A(I)}{OPT(I)} \to A(I) \leq \alpha \cdot OPT(I)$

For maximization: $\alpha_A = \max_{(I)} \text{of } \frac{OPT(I)}{A(I)} \to A(I) \geq OPT(I)/\alpha$

Vertex cover: $G = (V, E)$, gives a subset of vertices S contained in V that touches every edge. Goal is to minimize $|S|$. Special case works in $\log OPT(I)$

Proving NP-Comp: Show A is NP and Show A is NPH (Pick some NPC prob B and show B reduces to A.)

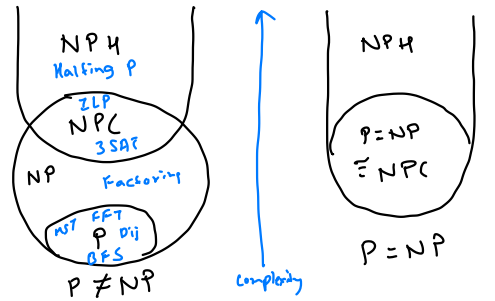
Vertex Cover 2-Approximation: Algo is to find a maximal matching M and return endpoints of all edges in M. Any vertex cover of a graph G must be atleast as large as the no. of edges in matching. Let S be a set containing both end points of each edge in a maximal matching M. Then S must be a vertex cover, if it isn't, i.e., if doesn't touch some edge e, then M could not possibly be maximal But our cover S has $2|M|$ vertices.. we know that any vertex cover must have size of atleast $|M|$.

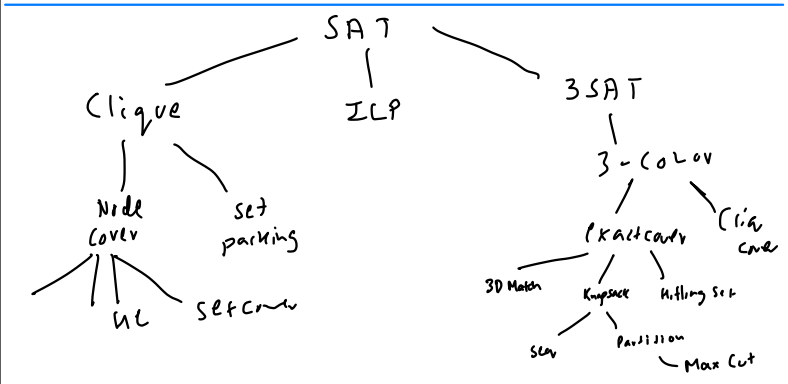
TSP approximation: If dist bet cities satisfy Δ^{le} inequality.

TSP cost $\geq$ cost of this path $\geq$ MST path

Remove edges from tour gets MST s.t.). If we use each edge twice, we get a tour that visits vertices more than once. Tour should skip any city about to revisit and move directly to next one in list.

$2 \cdot$ TSP cost $\geq 2 \cdot$ cost of path $\geq 2 \cdot$ MST cost





2-Point Pursuit:-

- A & B play a game, where there's some fixed point p^*. At each step of game, B chooses a point p, then A finds a line seperating p and p^* that is ϵ away from p. A loses when can't find this line. update $P_{t+1} = P_t + \epsilon V_t$
- In each iteration, the squared dist of B's point p to p^* dec by ϵ^2.
- Thm: If B plays by strategy the A fails to find a line sep within $\left\lceil \frac{\|p^{(0)} - p^*\|^2}{\epsilon} \right\rceil$ iterations.

- Inner Product $\langle x, y \rangle = \sum_i x_i \cdot y_i$ | Eucleng $\|x\|^2 = \sum x_i^2 = \langle x, x \rangle = 1$
- Hyper plane: Set of points that satisfy $\langle v, x \rangle - \theta = 0$
  - normal vector to hyperplane

Eg: Given $p^* = (0,0)$ and B's point is $(10, 5)$, how many iter it take for B to win the game? what points pass if $\epsilon = \sqrt{2}$ and slope is -2?

A) unit vector $= \left(\frac{-10}{\sqrt{10^2+5^2}}, \frac{-5}{\sqrt{10^2+5^2}} \right) = \left(\frac{-2}{\sqrt{5}}, \frac{-1}{\sqrt{5}} \right) \times \epsilon = (-2, -1)$

1st = $(10,5) + (-2,-1) = (8,4) + (-2,-1) \rightarrow 6,3 \rightarrow 2,1 \rightarrow 0,0$,

5 iterations

update rule = $P_{t+1} = P_t + (-2, -1)$

Eg: Provide eqn of a plane that is $\perp$ to $[5,1,2]$ that passes through $[1,4,3]$

A) plane $\perp^r$ to $[5,1,2]$ is $5x + 1y + 2z = K$. $K = 5(1) + 1(2) + 3(2)$

$\hookrightarrow 5x + y + 2z = 13$

Feasibility Prob: Find a point that satisfies a set of constraints. For an LP, these cons are linear of form $\langle a_x, m \rangle \geq b_w$.

↳ Applying Point pursuit: A proposes a soln point $x_{(t)}$, then B looks for a violated constraint $\langle a_i, x^{(t)} \rangle \leq b_i - \epsilon$

↳ Cons can be returned as hyperplane seperating x and x^*:

$l(x) = \langle a_i, x \rangle - b_i$

Algo:- $x^0 \leftarrow 1$

for t=0 to T

Find cons $\langle a_i, x \rangle \geq b_i$ violated by $x^{(t)}$ with error ϵ

$\langle a_i, x^t \rangle \leq b_i - \epsilon$

If no violation return

else → $x^{(t+1)} \leftarrow x^t + \eta \cdot a_i$

eg: For an LP with obj max $3x+4y$, how might use Alg 1 solving LP to find max?

↳ Formulate LP as $3x+4y \geq A \rightarrow$ Normalize $\rightarrow \frac{3}{5}x + \frac{4}{5}y \geq B$ then binary search to find largest value of B where LP is feasible.

We are gouranteed to find a LP feasible soln in D^2/ϵ^2 iterations if the dist from feasible region is $< D$.

- Halfspace: Set of pts on one side of a hyperplane. $H = \{x \in R^d \mid \langle w, x \rangle - \theta \geq 0\}$ ← eg: $x_1 + x_2 - 1 \geq 0$
- Convex Set: An intersection of a (possibly ∞) family of halfspaces.
  - ↳ A set S is convex if any pts $x, y \in S$, the line segment joining x, y is also contained in S.
- Convex Func: f obeys → $f\left(\frac{(x_1+x_2)}{2}\right) \leq \left(f(x_1) + f(x_2)\right)/2$
  - ↳ Pick any 2 points on a line and draw a straight line bet them. Now pick the mid of the x values and look at where the line is and where the func is.

  Convex ⟷ the func is below the line.

  ↳ can check of convexity by checking hold T for all x_1, x_2
- Max of 2 convex is also convex
- Convex func obey: $f(x^*) \geq f(x) + \frac{df}{dx}(x) \cdot (x^* - x)$
  - ↳ Pick any points on line and draw a tangent. The tangent should always stay below the func.

Eg: Show $f(x) = x^4 - 2x^2$ is not convex

$f(tx_1 + (1-t)x_2) \leq t f(x_1) + (1-t) f(x_2)$

Choose $x_1 = 1$, $x_2 = -1$

$f(x_1) = f(1) = -1$ $f(x_2) = -1$ $f\left(\frac{x_1+x_2}{2}\right) = f(0) = 0$

$f\left(\frac{x_1+x_2}{2}\right) \leq \frac{f(x_1)+f(x_2)}{2} \Rightarrow 0 \leq -1$ ← not True

- Complement of convex set is not convex
- Complement of convex set $S \in R^d$ is never a convex set.
- Sum of 2 convex is not convex eg: $f(x) = x^4$ $g(x) = x^2$
- Set of pts s.t $x^2 + y^2 \geq$ is a convex set → False

Seperation Oracle: That given a point x not in S, will find a halfspace seperating x from the set S.

↳ An ϵ-seperation oracle O for a convex set S is an algo that solves the following prob:-

Input: point x that is atleast ϵ away from convex set S

Output: halfspace H s.t $S \subseteq H$ but the point x is ϵ-away from H. The point x is ϵ-away from $|H|$ if $\|w\| = 1$ and $\langle w, x \rangle - \theta \leq -\epsilon$

Input: A separation oracle $\mathcal{O}$ for some convex set S
Output: A point x that is at most ϵ-away from the convex set S

for $t = 0$ to T **do**

- (Find a separating halfspace) Use the oracle $\mathcal{O}$ to find a halfspace that separates the current point $x^{(t)}$ from the convex set S, i.e., $\langle w, x^{(t)} \rangle - \theta \leqslant -\epsilon$ while, $\langle w, x \rangle - \theta \geqslant 0$ for all $x \in S$.
  If there is no separating halfspace then **return** $x^{(t)}$.
- Set
$$x^{(t+1)} \leftarrow x^{(t)} + \eta \cdot w$$

end for

Factoring via Circuit: Check n/p *p ==n. Imp Circuit that div and mul with n hard wired. n/p* p and n feed to final gate /XNOR =1 if factor 0 if not.

- Rudrata Cycle ⟶ Max Flow ⇒ P=NP
- ∀NP ⟶ Ind set prob ⇒ T
- ∀P ⟶ Max Flow ⇒ T
- 3SAT ⟷ Decision Ind set ⇒ T
- 3SAT ⟶ optimize Ind set ⇒ P=NP
- ∀NPH ⟶ 3SAT in poly ⇒ F
- Spanning tree T of cost w, then cost of Min TSP = 2w
- Cost of min TSP is w, MST = w.
- If there is poly time algo for one prob in NP, then it is for all ⇒ P=NP
- LIS is NPC ⇒ P=NP
- Matching ⟶ Ind Set
- If proved poly algo for NPC then P=NP.
- 3SAT ⟶ Palindrome Checking ⇒ P=NP
- All prob in NP can be solved in poly space.
- Ind Set ⟶ LIS ⇒ P=NP
- ∀P/NP ⟶ 3SAT ⇒ T
- ∀NP ⟶ IP ⇒ T
- Min Vertex Cover is solvable in Poly time.
- MST is NP
- ∀NP can be written in linear program.

Given an instance ϕ of prob [Red^n to Use] we will construct an instance Ψ of the prob [Q asked]