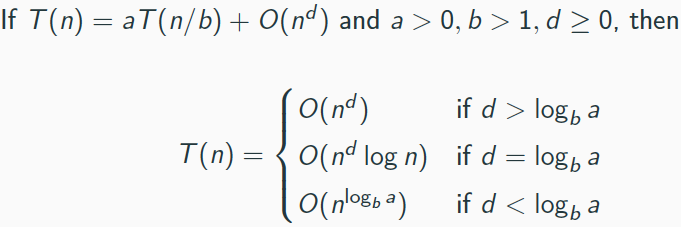
|  |  |
| --- | --- |
| **Stable** | **Unstable** |
| Insertion Sort | Selection Sort |
| MergeSort | QuickSort |
| Counting Sort |  |
| Radix Sort |  |



Counting Sort:

For each x, find how many elements < x

A[1..n], B[1..n], C[1..k]

Medians:

Pick v and split list into three (less than, equal to, greater)

Best: T(n)=T(n/2)+O(n) (even split)

Worst cast: v is min/max value

in array

Avg: T(n)=T(3n/4)+O(n)

Sorting Lower Bound:

Comparison based sort has lower bound: Ω(nlogn)

n-bit #’s -> Gauss: T(n) = 4T(n/2)+O(n) to T(n) = 3T(n/2)+O(n) 🡪 O(n^2) to O(n^1.59)

Matrix Multiplication:

Divide into 8 (n/2 x n/2) then recursively multiply

Add resulting matrices

T(n) = 8T(n/2) + O(n^2) -> O(n^3)

Strassen’s Method:

T(n) = 7T(n/2) + O(n^2) -> O(n^2.81)

Radix Sort:

N numbers, each d digits long

Sort right to left 1 col at a time

Sort 1 col: O(n+k), Sort d cols: O(d(n+k))

Have to use stable sort to sort columns

O(nlogn)

Merge Sort:

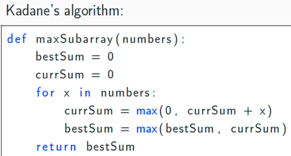
Split into subproblems of size n/2 until units of 1, then recursively merge back

T(n) = 2T(n/2) + O(n)

Master theorem:

Kadane’s Algorithm:

Iterate through array only once, calc. sum at each # and record local max and global max- O(n)



Max. Circ Subarray: O(n) Use kidane’s with min to find min subarray and Rest is max subarray.. O(n^2) naïve and O(nlogn) div and c

Minimum Spanning Tree:

Tree (no cycles), Connects

all vertices w/ min cost and (spanning)

Entropy: w/ huffman encoding 🡪 smallest bits are the most predictable

Graph Properties:

Undirected graph is connected if there is a path from any vertex to any other

In a disconnected graph, each connected subgraph = a connected component

Two nodes (u,v) are strongly connected if there is u->v path and v->u path

Tree Properties:

Tree with n nodes has n-1 edges

Connected undirected graph with |E|=|V|-1 is a tree

Undirected graph is tree IFF unique path btw any pair of nodes

MST-> Kruskal’s Algorithm:

Start with empty set

Add next lightest edge that

does not create a cycle O(|E|log|V|)

DFS:

🡪 mark nodes previsit, postvisit O(1) and O(|V|) to mark all nodes

O(|E|) work to scan edges. O(|V||E|)

MST-> Prim’s Algorithm:

Edge set x is always subtree of G

X grows by lightest edge each time

(value of node = lightest incoming edge)

BFS:

Proceed layer by later. Find layer d+1 from neighbors of node at layer d. Each vertex is placed onto queue 🡪 O(|V|+|E|)

Tower of Hanoi:

T(n)=2T(n-1)+O(1)

🡺 O(2^n)

DAG: (Directed Acyclic Graph)

Good for modeling hierarchies and dependencies

Source 🡪 node w/ no incoming edges

Sink 🡪 node w/ no outgoing edges

Cycle in DAG:

IFF DFS reveals a backedge

Dijkstra’s:

Given graph G, start S, find shortest paths to all reachable vertices. O((|V|+|E|) log|V|)

Bellman-Ford:

Computes shortest paths from single source v, to

all other vertices in graph 🡪 (Dijkstra’s, but..)

🡪 Simply update all edges |V|-1 times (to get shortest path)..Can handle negative edge weights O(|V||E|)

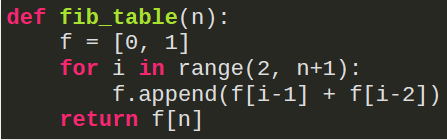
Linearize DAG:

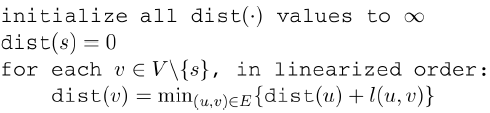
List nodes in decreasing order of post number (start with source end with sink)

Sparsity:

Sparse:|E|~ |V|

Dense:|E|~|V^2|





Asymptotics:

ω(…): “greater-than” limit f(n)/g(n) = 0, f(n) = o(g(n))

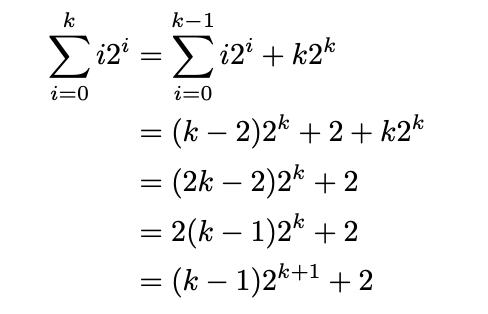
Ω(…): “greater than or equal” limit f(n)/g(n) = ∞, f(n) = ω(g(n))

θ(…): “equal” limit f(n)/g(n) ∈ R, f(n) = θ(g(n))

O(…): “less than or equal”

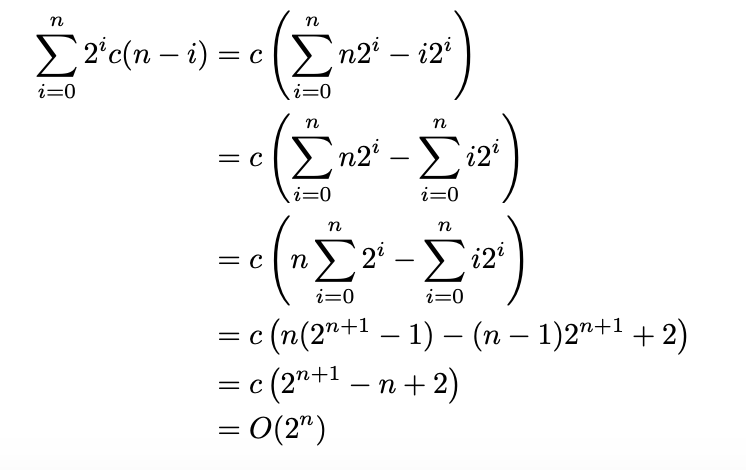
o(…): “less than”

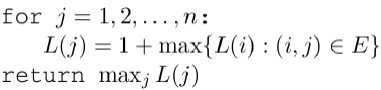
Proof by Induction:



Longest Increasing Subsequence

Recursion Trees: T(n) = 2T(n-1) + O(n)





Shortest Reliable path:

Shortest path from start s and t w/ at most k edges.

Dist(v,i) = min{dist(u, i-1)+l(u,v)} for all u,v

All Pairs Shortest Paths:

Could use bellman-ford on every vertex, but this takes O(|V|^2|E|)

Instead, use the Floyd-Warshall algorithm which takes O(|V|^3)

Floyd-Warshall:

Shortest path from u🡪 v w/

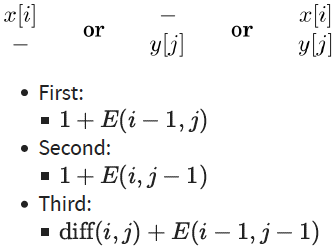
no nodes in btw = u-v path.

Min(dist(I,k,k-1) + dist(k,j,k-1),

dist(I,j,k-1) 🡪 O(|V|^3) < O(|V|^2|E|)

Edit Distance:

Minimum number of edits to transform s1 into s2

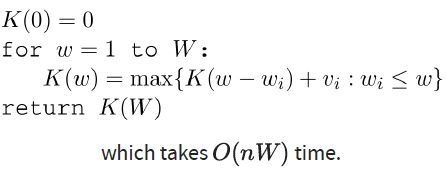


Independent Sets:

Subset that has no edges between its nodes

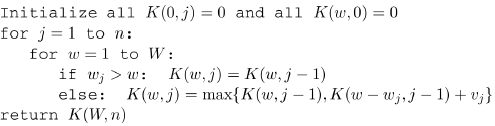
I(u) = max{1 + E(grandchildren w of u) I(w), E(children w of u) I(w)}

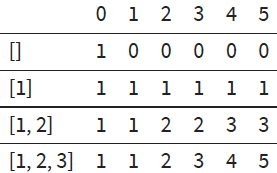
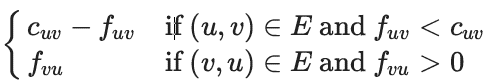
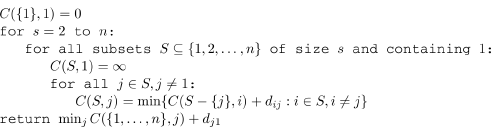
Knapsack with repetition:



O(nW) -> W can be 2^b

Knapsack (0/1)





Bipartite Graph:

Vertices can be partitioned into two sets V1,V2 such that there are no edges between vertices in the same group

Max Flow Min Cut Theorem:

Size of the maximum flow in a network equals the capacity of the smallest (s,t) cut

Duality Theorem:

If a linear program has a bounded optimum, then so does its dual, and they’re equal

Longest Path In DAGS Reduces…

To the shortest path (negate edges and rum shortest path

Linear Programming:

Optimization problems with an objective function and constraints that are linear functions

Dynamic Programming:

Break problem into collection of subproblems

Solve smallest subproblem

Move on to bigger subproblems

Recursive Algos:

Top down and Often optimal

Divide problem into subproblems of the same type

Combine subproblems

Coin Change:

Build table w/ 0,1,2,…value as columns, different sets with coins as rows

Cell Ai,j = Ai-1,j + Ai,j-v

Ford-Fulkerson Algorithm:

Start with zero flow

Choose s🡪t path and increase flow

as much as possible O(|V||E|^2)

Traveling Salesman Problem:

Brute force every tour

takes O(n!)

DP Algo: O(n^2 \* 2^n)

Circuit Value:

given a boolean circuit, what t/f values does it evaluate to true?

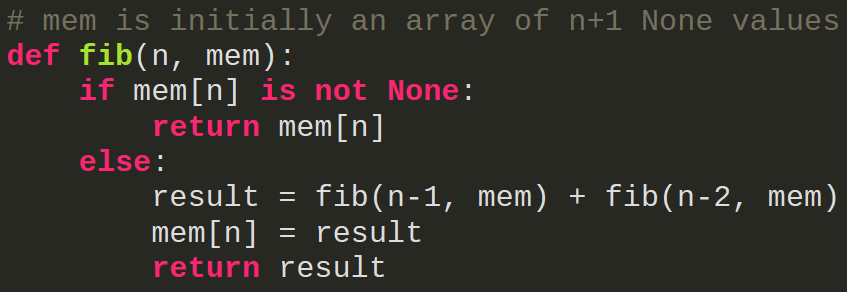
Any poly time algo can be reduced to a circuit of poly size 🡪 circuit reduced to a LP 🡪 LP can be done in poly time 🡪 any algo with poly time reduces to Linear Programming

Standard Form:

Variables all nonnegative

Constraints all equalities

Objective function to be minimized



Adjacency list:

Each vertex has list of neighbors

Check edge: O(n)

Space: O(|E|)

Adjacency matrix:

|V|=v, nxn matrix

Aij=1 if e(I,j), else 0

Check edge: O(1)

Space: O(n^2)

Simplex Method:

Start at a vertex

Walk to adjacent vertex with higher value

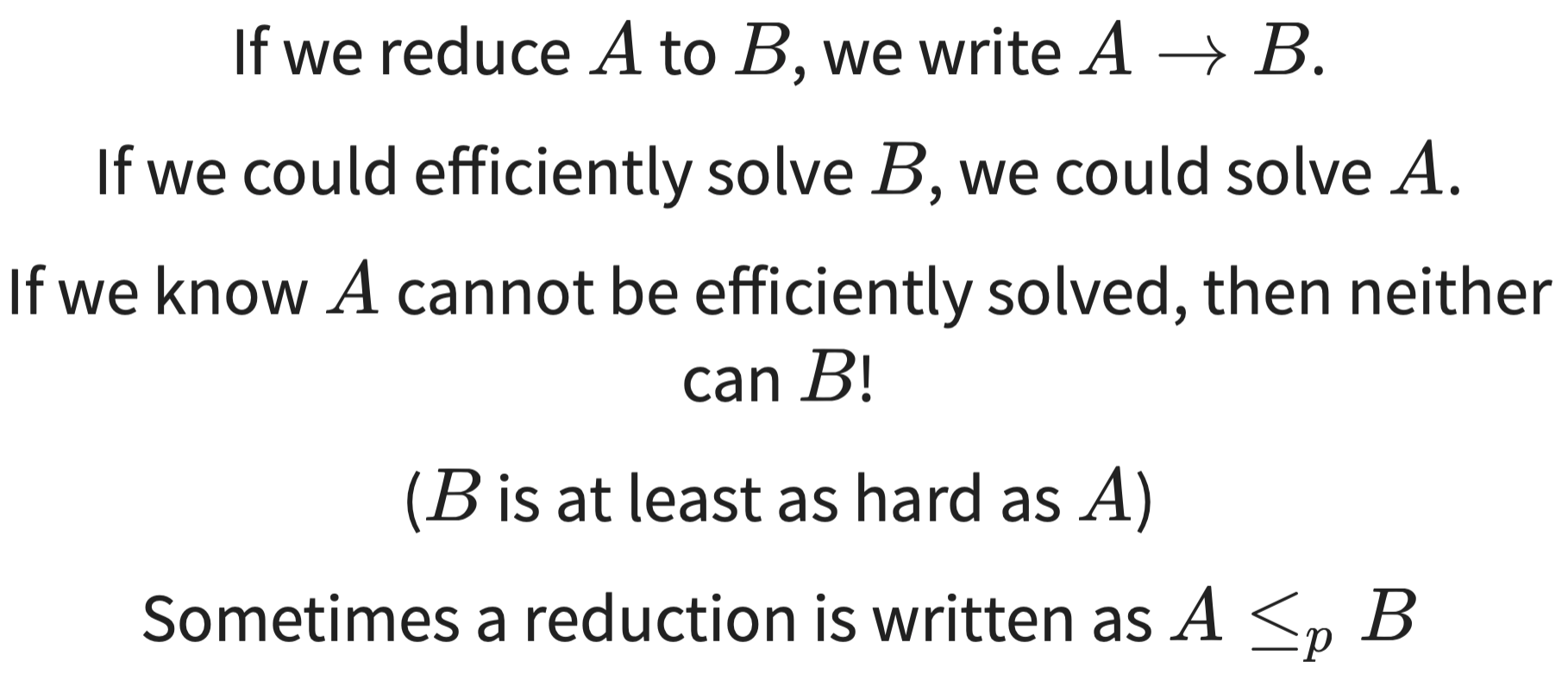
Stop when neighbors all have lower value

Clique (and max Clique):

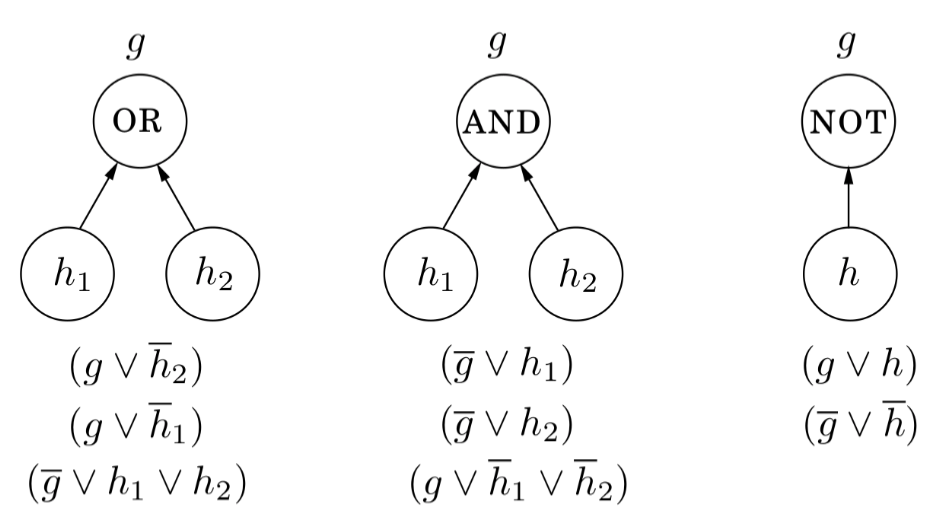
A clique is a: subset of vertices c in V where every 2 distinct vertices are adjacent.

Additionally, a max clique will be the inverse graph of the max independent set graph, if one can be solved in P then both can be.

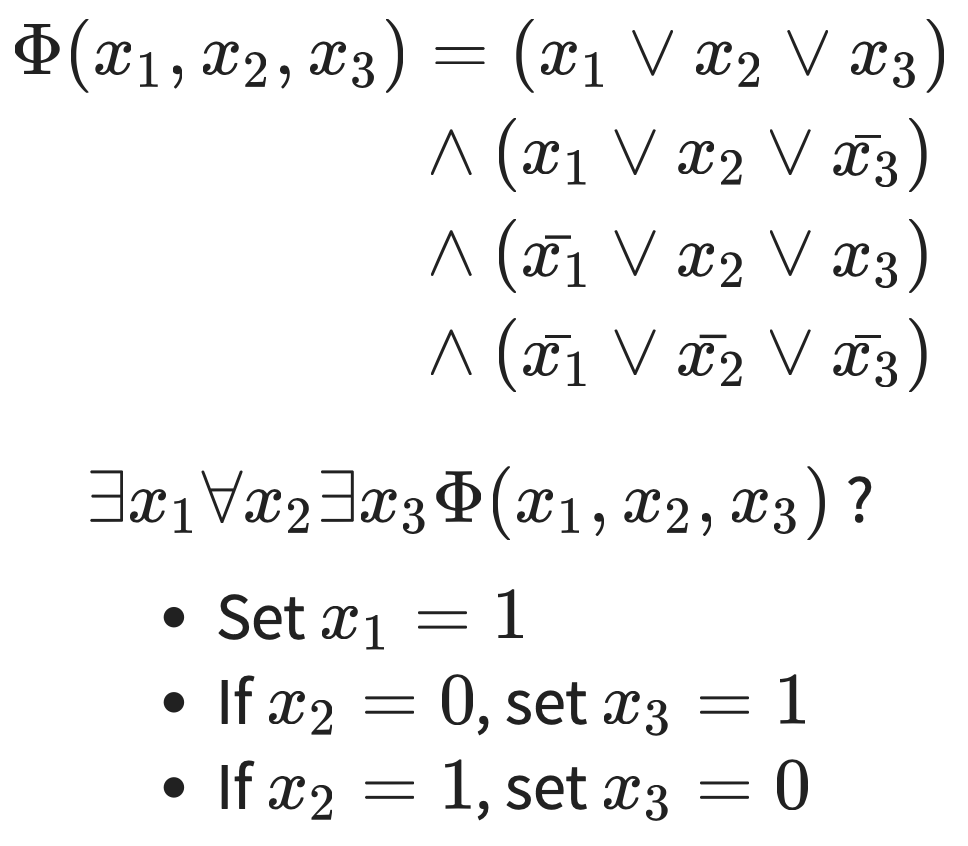
Reducing A to B:



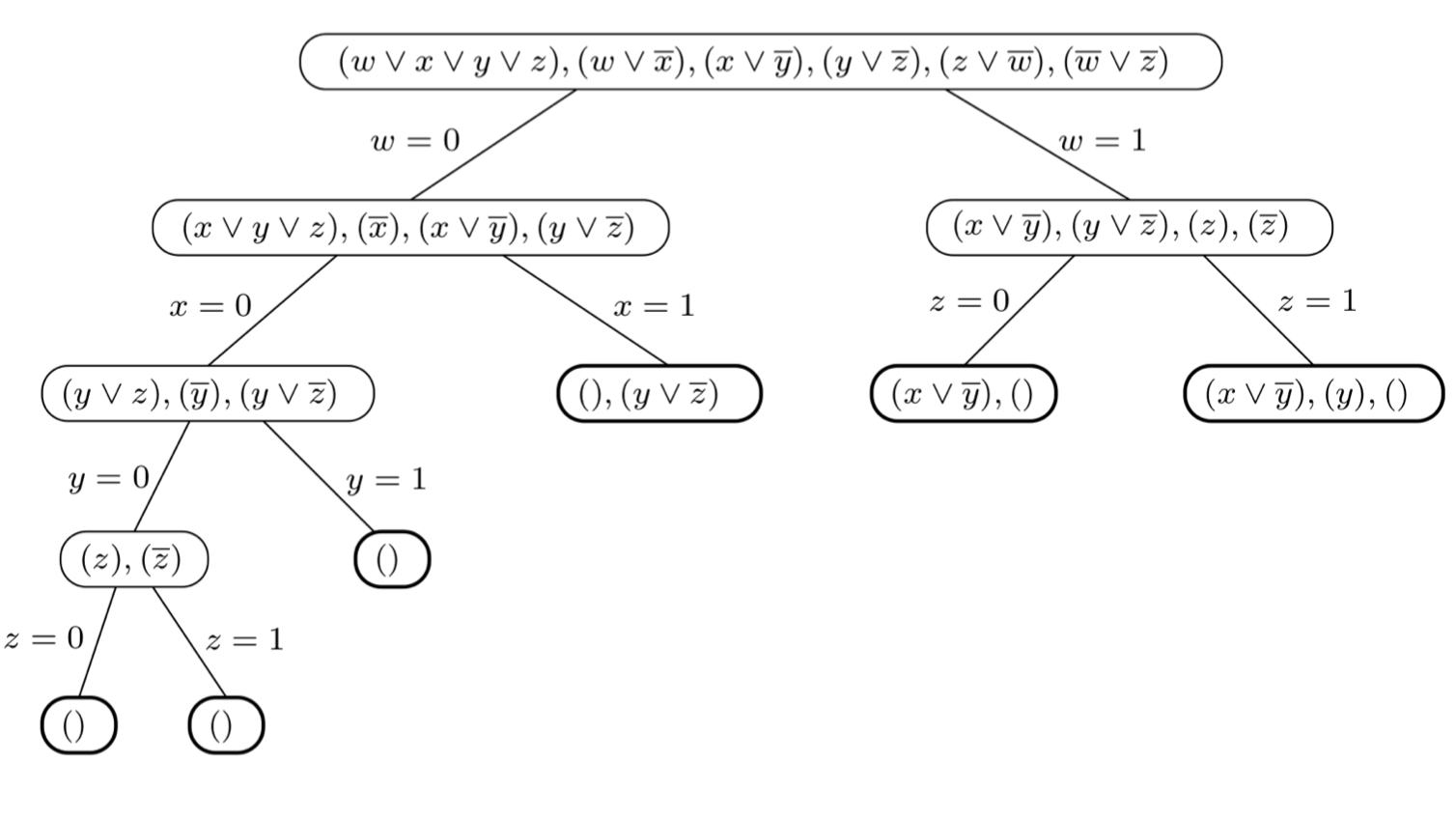
Circuit SAT 🡪 SAT



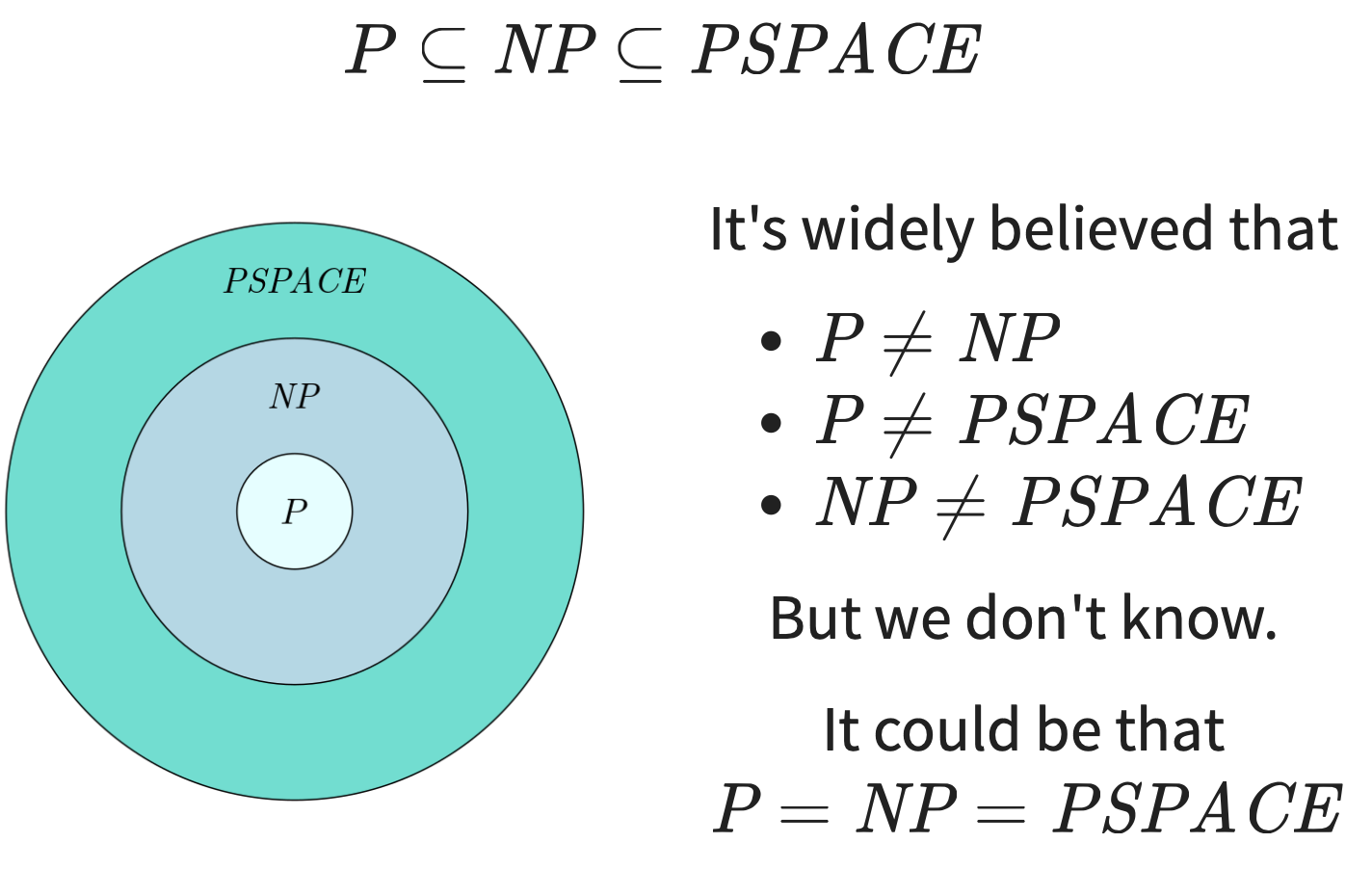
QSAT:



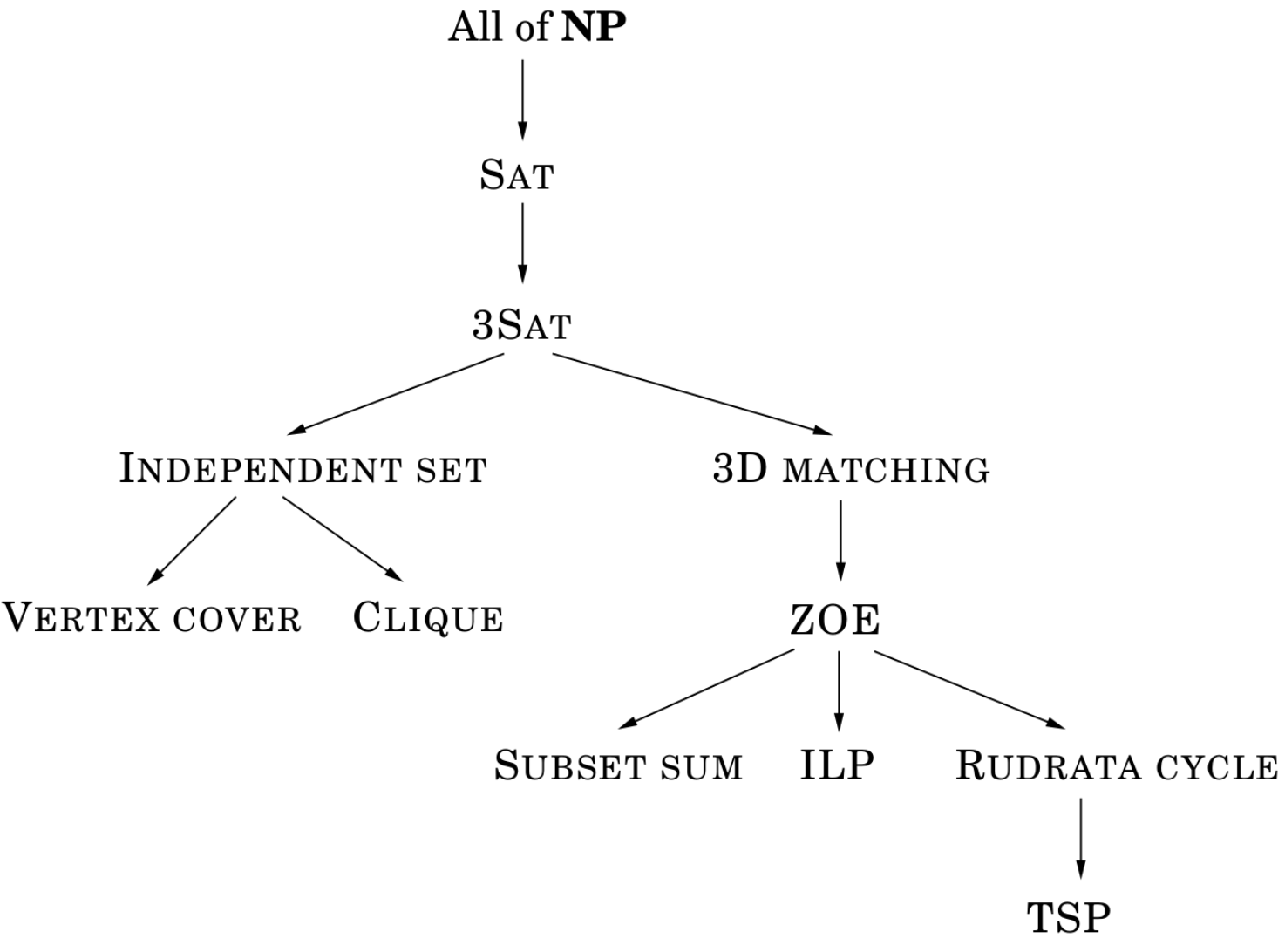
Backtracking:



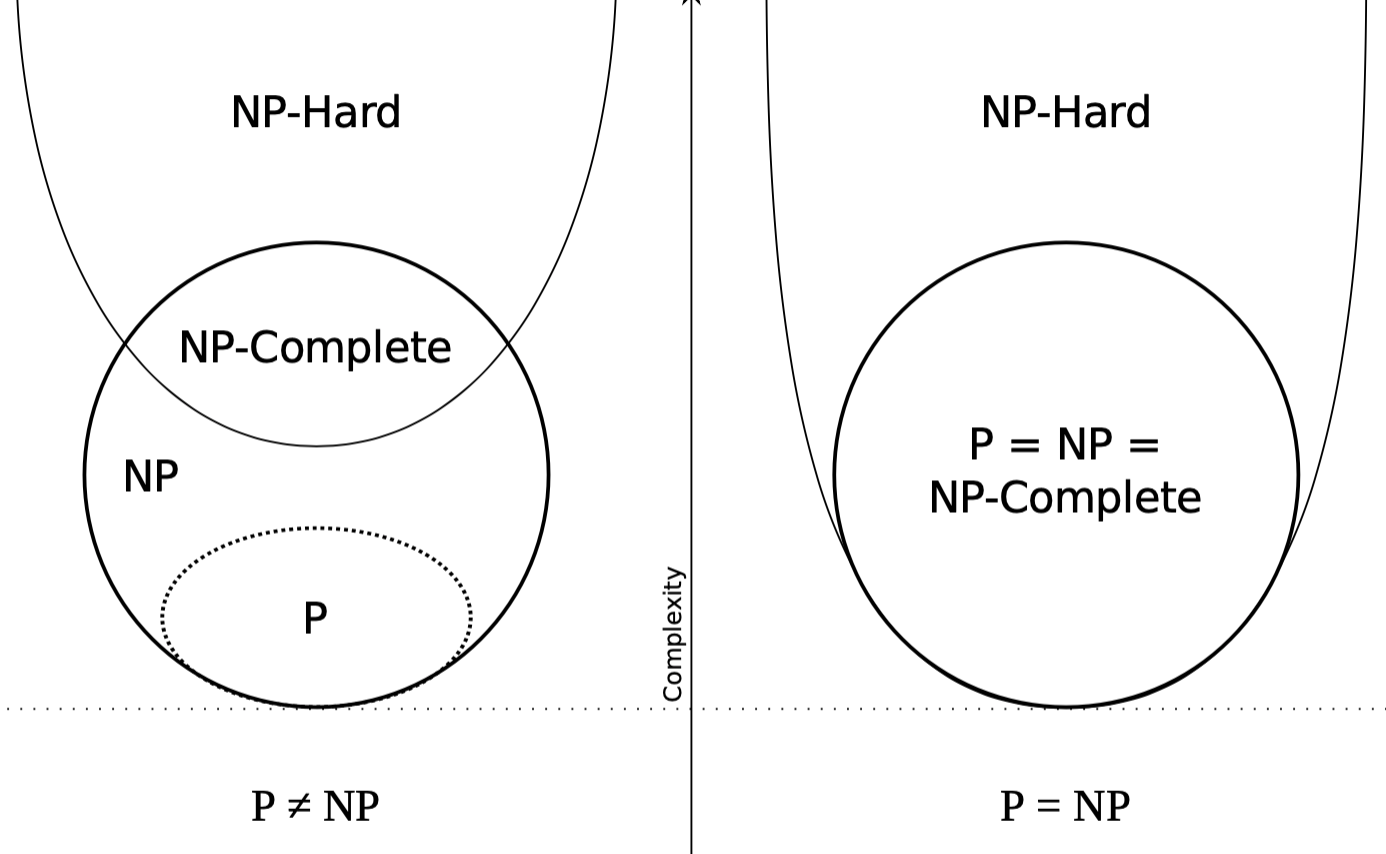
P, NP and P Space:



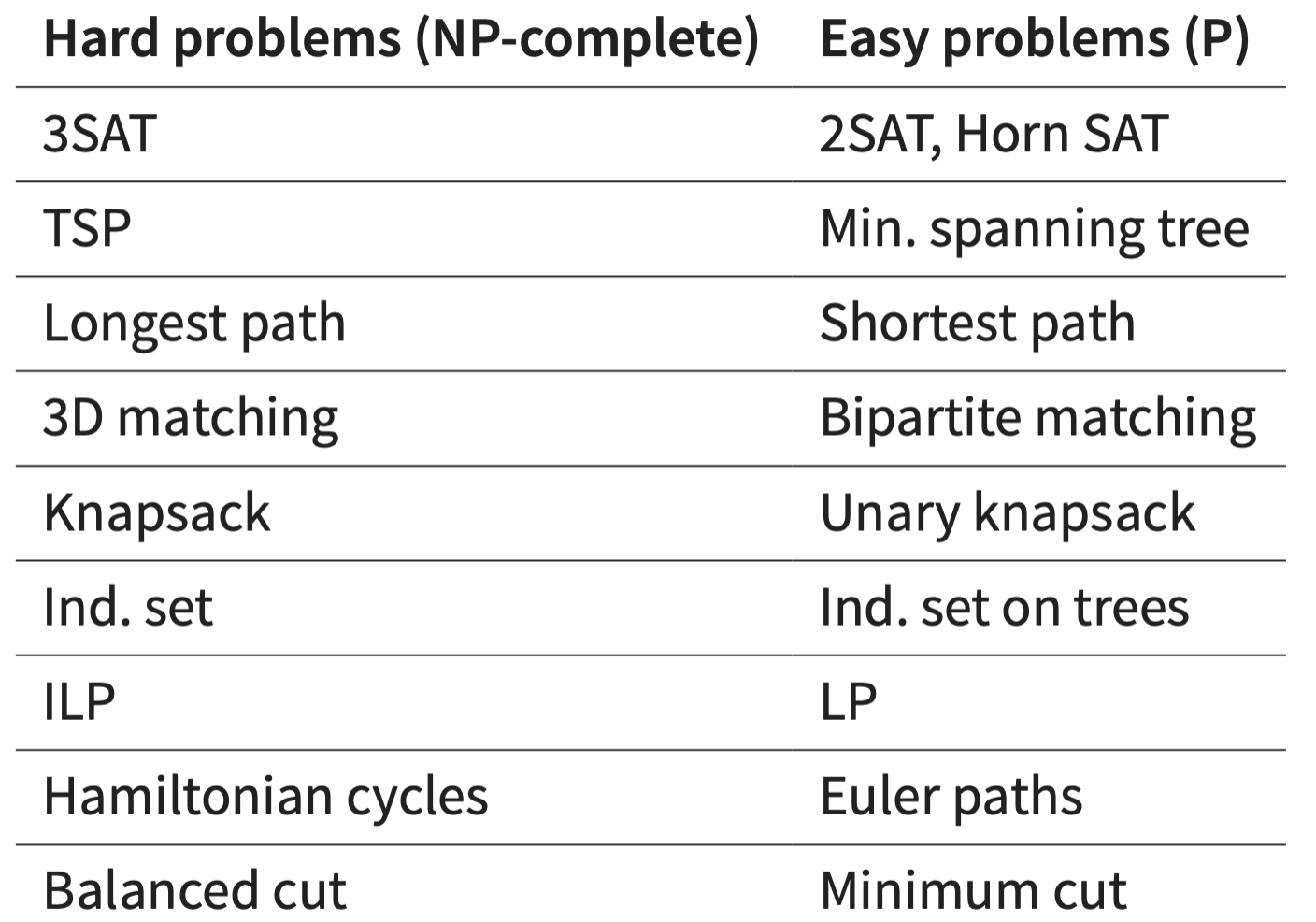
Reductions:



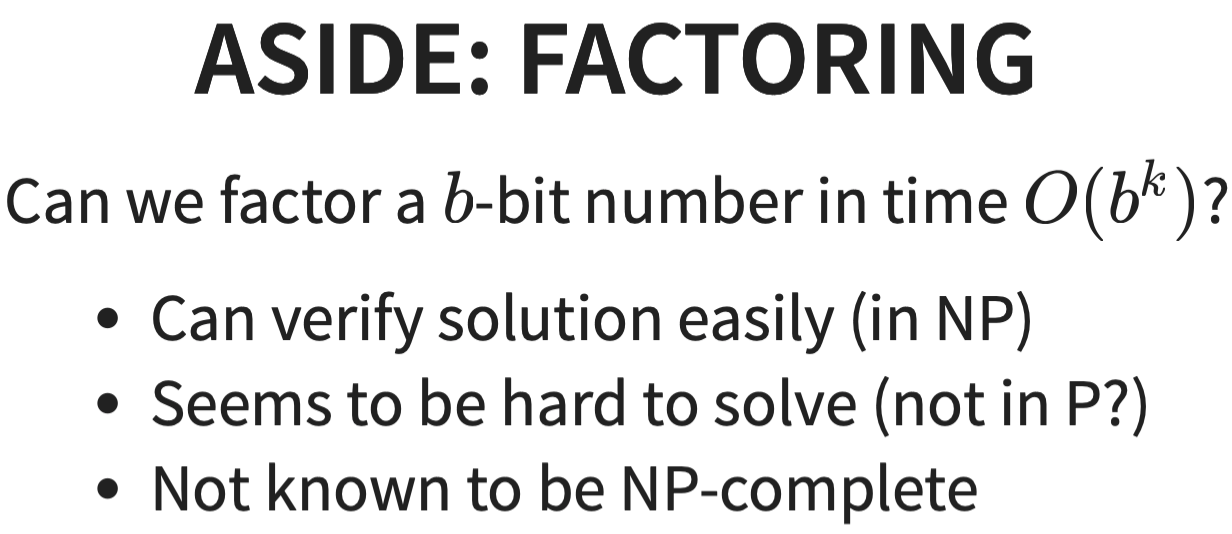
P to NP relationships:



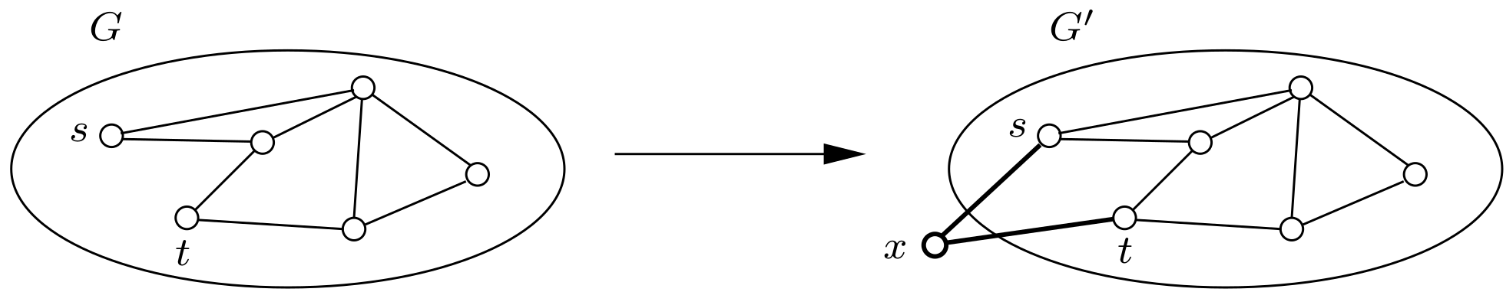
Problem Classification Levels:



Factoring:



Hamiltonian Cycle:



Subset sum:

Item’s value = item’s weight

Goal = capacity W

Find subset of ints that add to W

Longest Path:

Find path from s to t with weight at least g

Zero-One Equation:

Ax=1, A is mxn of 0’s,1’s

X is n-vector of 0-1 values

Integer Linear Programming:

Restrict variables to be integers

Balanced Cut:

Given graph and b, find cut with at most b edges that split into S,T where |S|,|T| >= n/3

Minimum Cuts:

Given graph and b, find cut with at most b edges

QSAT:

Brute force, try every possible x1 value, recursivel solve rest, takes S(n)<=2S(n-1)+p(n) which is exponential

Modified: only keep one bit results, becomes S(n)<=S(n-1)+p(n) <= n\*p(n)

QSAT in PSPACE

Planning:

Model interaction with environment to achieve goals

Given conditions, operators

Operators have preconditions, add, delete list

Apply operations to go from C0 to C\*

Hamiltonian Cycle Problem:

Given graph, find cycle hat visits every vertex exactly once

Euler Path:

Contains each edge exactly once

At most 2 nodes in graph can have odd degree

PSPACE Complete:

Problem in PSPACE

All PSPACE problems reduce to it

PSPACE:

Polynomial Space

P ⊆ PSPACE

NP ⊆ PSPACE

ANY NP PROB -> CIRCUIT SAT:

Given problem A in NP, has C(I,S)

Bits of S become unknowns

Satisfying assignment to unknown = solutions of I

Efficient Checking:

There exists polynomial time algo that takes I,S and reports where S is a solution for I

Satisfiability versions:

General: exponential

Horn Form, 2SAT: linear

3SAT: exponential

Satisfiability As Search:

Given instance I, find solution S that satisfies some specification

SAT -> 3SAT:

Split clause with > 3 literals into series of clauses with Ais and Yis

If RHS satisfied, at least 1 Ai true

If LHS satisfied, some Ai is true, set y1 to Yi-2 true, other Yj false

3SAT -> IND. Set:

Turn clauses into graph

If ind. Set found, those vals true

If not, no satisfying agreement

Local Search Heuristics:

Let s be initial solution, while solution s’ exists in neighborhood of s and less costly, swap s,s’

Horn Formulae:

Boolean variables, horn clauses of variables and negations

Facts take form: 🡪 X

Find assignment so clauses all true

Ham. Path -> Ham. Cycle:

Add e(x,s), e(t,x)

If cycle, delete edges, theres path

If not, there is no path

TSP As Search:

Given budget, is there tour within budget?

TSP As Optimization:

Given graph, find length of minimal tour

TSP Cost:

TSP cost >= path cost >= MST cost

C<=2\*MST cost<= 2\*TSP cost

Approx.. ratio at most 2

Approximation Ratio:

αA = max(i) A(I) / OPT(I)

P and NP:

P: solveable in poly. Time

NP: verifiable in poly. Time

P ⊆ NP

Proof NP Completeness:

Prove B, A is NP complete

Show A -> B

NP Completeness:

Problem is NP complete if:

1. It’s in NP

2. All problems in NP reduce to it

(“hardest” problems in NP)

3RD half of Year stuff