Computational Complexity

P:

- Set of all problems you can solve in polynomial time;

EXP:

- Set of all problems you can solve in exponential (2^n^const) time;

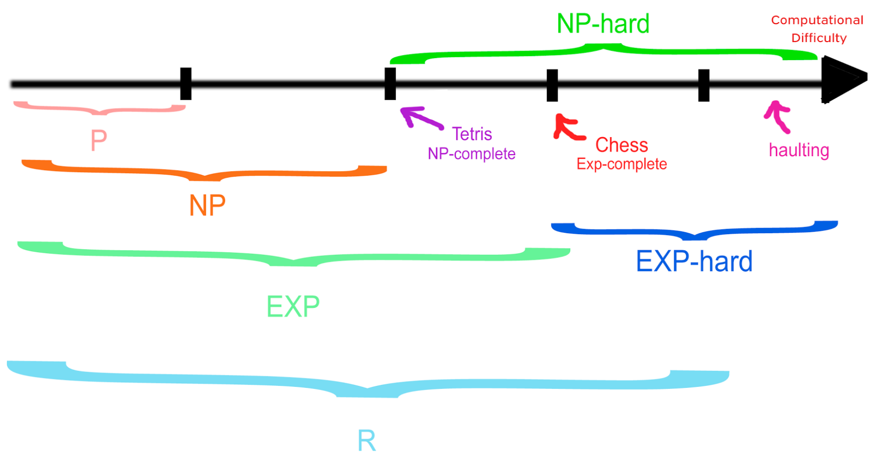
R:

- Set of all problems you can solve in finite time;

NP:

- Set of decision problems in polynomial time via a “lucky” algorithm;
- Nondeterministic polynomial;
- Lucky algorithm: makes guesses (only one and it gets the right choice);
- Nondeterministic model;
- “by magic”;
- The output of the algorithm is a YES or NO answer;
- The guesses are guaranteed to lead to a YES answer if possible;
- {decision problems with “solutions” that can be “checked” in polynomial time};
- When answer is YES you can prove it & check the proof in polynomial time.

- Does P = NP?
- Most people think that P ≠ NP: big conjecture;
- “can’t engineer luck”;
- Generating (proofs of) solutions can be harder than checking them).



Tetris:

- Given a set of pieces, determine if there's a way to get a yes answer (if I can win the game by setting the pieces in a certain way);
- You guess and place a piece and keep doing it;
- "Can I survive?" is in NP;
- "Can I die?" we don't know;
- Tetris is in NP;
- Proof = sequence of moves to make;
- Claim: if $P \neq NP$ then Tetris $\in NP - P \Rightarrow \notin P$;
- Why? Tetris is NP-hard: as hard as every problem in NP;
- In fact, Tetris is NP-complete = NP-hard $\cap$ NP;
- 3-Partition $\rightarrow$ Tetris.

Examples:

- Negative-weight cycle detection $\in P$;
- n x n Chess $\in$ Exp & $\notin P$;
- Tetris $\in$ Exp & can't know whether $\in P$;
- Haulting problem: given a computer program, does it ever hault/stop? $\notin R$.

- Most decision problems are uncomputable ($\notin R$):
    - Decision problems: yes or no answer; = function: input $\rightarrow$ {YES, NO};
    - Program ~ binary string ~ natural number $\in \mathbb{N}$;
    - Program: finite string of bits;
    - Decision problem: infinite string of bits; $\in \mathbb{R}$;
    - $|\mathbb{R}| >> |\mathbb{N}|$;
    - Almost every problem is unsolvable by any program,

REDUCTIONS:

- Way to design algorithms;
- Problem A, convert to B (you already know how to solve);
- $A \rightarrow B$;
- B is at least as hard as A;
- All of NP problems you can reduce it to each other;
- To prove your problem is a NP problem: find some known NP problem and reduce it to your problem;
- Convert problem A into problem B;
- Unweighted shortest paths $\rightarrow$ weighted shortest paths (add weights of 1);
- Minimize-product path $\rightarrow$ shortest path: logs can convert products into subtractions;
- Longest path $\rightarrow$ shortest path: just negate all the weights.

COMPLEXITY: P, NP, NP-completeness

P: Problems solvable in polynomial time

- (n^const) -> n is the size of the problem;

NP: Decision problems solvable in nondeterministic polynomial time

- answer yes or no;
- Nondeterministic means you can guess one of out polynomially many options in O(1) time;
- Guess is guaranteed to be a good guess;
- If any guess leads to YES then we get such a guess;
- Decision problems with polynomial size certificates and polynomial time verifiers for YES inputs;
- X $\notin$ P unless P = NP (which would mean that every problem would be solvable in polynomial time -> sadly this is not true, but you can't prove);
- NP-complete:
    - X is NP-complete if X $\in$ NP & X is NP-hard;
    - X is NP-hard if every problem Y $\in$ NP reduces to X;
    - X is at least as hard as everything else in NP (no harder, no easier);
    - How to prove X is NP-complete:
        - X $\in$ NP;
        - Reduce from known NP-complete problem Y to X;
        - Nondeterministic algorithm or some certificate + verifier.
- Reduction:
    - From problem A -> problem B = Polynomial time algorithm converting A inputs -> equivalent B inputs
    - Same yes or no answer;
    - If B $\in$ P then A $\in$ P;
    - If B $\in$ NP then A $\in$ NP;
    - B is at least as hard as A.

3SAT:

- Given Boolean formula of the form (X1 v X2 v X6) ^ (X2 v X3 v X7) …
- Can you set the variables X1, X2, … -> {T, F} such that formula = T?
- It's a NP problem: guess X1 = T or F
guess X2 = T or F
…
check formula: if T return YES, else if F return NO;
- Polynomial time verification algorithm to check the formula;
- If you say that this problem is not satisfiable, you can't prove that;
- If you can guess the X's and find an YES answer, you are proving that it is indeed satisfiable.

SUPER MARIO BROS.:

- Prove that it is NP-hard;
- Generalize to arbitrary screen size n x n;
- Reduce from 3SAT to Super Mario Bros;
- Build a level that implements the formula;
- The variables have each one a True choice and a False choice.

3 - DIMENSIONAL MATCHING (3DM):

- Given disjoint sets X, Y, Z each size n;
- Given triples ($T \subseteq X \times Y \times Z$);
- Your goal is to choose among those subsets of Triples ($S \subseteq T$) such that every element $\in X \cup Y \cup Z$ is in exactly one $s \in S$;
- You can guess which elements of T is in S and you check if this is True or False;
- So it is NP-complete;
- To prove that it is NP-hard, you can reduce from 3SAT to 3DM;
- So you need to convert the formula into an equivalent 3DM input.

SUBSET SUM:

- Given n integers, A = { a1, a2, ..., a3} & target sum t;
- Is there a subset of the integers that ads up to that target?;
- $\sum S = \sum_{ai \in S} ai = t$;
- Weakly NP-hard.

PARTITION:

- Given A = {a1, ..., an};
- Is there a subset $S \subseteq A$ such that $\sum S = \sum(A - S) = (\sum A)/2$?;
- Reduction from Subset Sum;
- Let a = $\sum A$;
- Add a(n+1) = $\sigma + t$ & add a(n+2) = $2\sigma - t$;
- + σ - t = + t.

RECTANGLE PACKING:

- Reduction from Partition.

NP-Completude: CIRCUIT-SAT e outros problemas

- Mostrar a existência de um problema NP-Completo;

CIRCUIT-SAT:

- Existe uma atribuição de x1, x2 e x3 para que a saída um circuito combinacional dessas entradas seja 1?;
- Seja n o nome de entradas do circuito, existem 2^n possibilidades;
- O tamanho de um circuito é dado pelo número de elementos combinacionais (entradas) + número de fios do circuito;
- Devemos provar que CIRCUIT-SAT $\in$ NP e CIRCUIT-SAT é NP-hard:
    - Suponha um algoritmo A(x,y) onde x é uma codificação do circuito e y um certificado (atribuição de valores aos fios do circuito);
    - A: Para cada porta lógica, verifica se o valor fornecido no fio de saída é calculado corretamente em função dos fios de entrada, se não for retorna 0. Se a saída do circuito inteiro é 1, retorne 1, senão retorne 0;
    - Ou seja, A é capaz de verificar uma resposta para CIRCUIT-SAT em tempo polinomial, portanto, CIRCUIT-SAT é NP.
    - Para provar que esse problema é NP-hard, deve-se mostrar que todo problema em NP é redutível em tempo polinomial a CIRCUIT-SAT:

  NP $\rightarrow$ CIRCUIT-SAT (ou seja, CIRCUIT-SAT is at least as hard as any problem in NP);
    - Qualquer programa de computador que pertence à NP $\rightarrow$ CIRCUIT-SAT, pois um programa de computador no nível mais baixo é implementado por portas lógicas, ou seja, circuitos combinacionais;
    - Então, qualquer que seja L (um problema em NP), vamos conseguir reduzir para CIRCUIT-SAT, pois vai existir um algoritmo F que reduz L para CIRCUIT-SAT;
    - Portanto, CIRCUIT-SAT é NP e é NP-hard, logo, CIRCUIT-SAT é NP-completo.

MAIS PROBLEMAS NP-COMPLETOS:

- SAT:
    - Suponha n variáveis booleanas e m conectivos booleanos (qualquer função booleana com uma ou duas entradas e uma saída), SAT é satisfeito quando existe uma atribuição verdadeira que faz com que a fórmula seja avaliada como 1;
    - SAT $\in$ NP: verificação se um certificado é igual a 1 em tempo polinomial;
    - SAT é NP-hard: devemos mostrar que CIRCUIT-SAT reduz (em tempo polinomial) para SAT. SAT is at least as hard as CIRCUIT-SAT;

- Podemos expressar qualquer CIRCUIT-SAT como um SAT, assim SAT é no mínimo tão difícil quanto CIRCUIT-SAT. Mas CIRCUIT-SAT é NP-completo, logo SAT será NP-hard;
- Como já foi mostrado que SAT pertence à NP, logo SAT também é NP-completo.

- 3-SAT:
    - Conjunto de disjunções;
    - 3 literais distintos em cada cláusula;
    - 3-SAT $\in$ NP: a verificação do certificado em tempo polinomial é simplesmente a avaliação da fórmula;
    - 3-SAT $\in$ NP-hard: Reduzindo SAT para 3-SAT, temos que SAT <= 3-SAT;
    - Redução:
        1. Construir uma árvore binária de análise;
        2. Para cada nó, colocar na forma de conjunções;
        3. Converter as cláusulas para a forma normal conjuntiva;
        4. Quando houver cláusulas que não possuem exatamente 3 elementos, deve-se adicionar literais auxiliares sem impactar na avaliação da fórmula;
    - Sabendo que conseguimos reduzir de SAT para 3-SAT, podemos concluir que 3-SAT é NP-completo (pois SAT já foi provado ser NP-completo) e 3-SAT é tão difícil quanto SAT.