# P, NP, NP-Complete and NP-Hard Problems

# Polynomial Time

- Most (but not all) of the algorithms we have studied so far are easy, in that they can be solved in polynomial time, be it linear, quadratic, cubic, etc.
- Cubic may not sound very fast and isn't when compared to linear, but compared to exponential time we have seen that it has a much better asymptotic behavior.
- There are many algorithms which are not polynomial time. In general, if the space of possible solutions grows exponentially as the value of *n* increases, then we should not hope for a polynomial time algorithm. These are the exceptions rather than the rule.

# General Problems, Input Size and Time Complexity of Algorithms

- **Time Complexity of Algorithms:**

  **Polynomial Time Algorithm (“Efficient Algorithm")** vs.

  **Exponential Time Algorithm (“Inefficient Algorithm")**

| f(n) \ n | 10 | 30 | 50 |
|---|---|---|---|
| n | 0.00001 sec | 0.00003 sec | 0.00005 sec |
| $n^5$ | 0.1 sec | 24.3 sec | 5.2 mins |
| $2^n$ | 0.001 sec | 17.9 mins | 35.7 yrs |

# “Hard” and “Easy’ Problems

- Sometimes the dividing line between “easy” and “hard” problems is a fine one. For example:
  - Find the shortest path in a graph from node X to node Y (Easy)
  - Find the longest path in a graph from X to Y (with no cycles) (Hard)
- View another way – as “Yes/No” problems
  - Is there a simple path from X to Y with weight <= M? (Easy)
  - Is there a simple path from X to Y with weight >= M? (Hard)
  - First problem can be solved in polynomial time.
  - The second problem can be solved in exponential time.

# Hard Problems

- Problems with no known polynomial solution are called Hard Problems.
- Another class of problems (NP) seem like they should be easy. They have the following property:
  1. A Solution can be **Verified** in Polynomial Time.
- Consider the well known *Satisfiability* (*SAT*) Problem:
  – **Input:** A Boolean expression, *F*, over *n* variables (*x1*,. ..,*xn*)
  – **Output:** *1* if the variables of *F* can be fixed to values which make *F* equal **true**; *0* otherwise.

# Decision and Optimization Problems

- Decision Problem: It is a Computational Problem with an intended output of “Yes” or “No”, 1 or 0.
- Optimization Problem: It is a Problem where we try to maximize or minimize some value.
- Introduce a parameter k and let us ask whether the optimal value for the problem is at most or at least k. Turn optimization problem into decision problem.

- **Decision Problem:** The solution to the problem is "yes" or "no". Most optimization problems can be phrased (transformed) as decision problems (still we have the same time complexity).

Example: Let us assume that we have a decision algorithm X for 0/1 Knapsack Problem with capacity M, i.e. Algorithm X returns "Yes" or "No" to the following question.

*"Is there a solution with profit* $\geq P$

*subject to knapsack capacity* $\leq M$*?"*

We can repeatedly run algorithm X for various profits (P values) to find an optimal solution.

Example: Use binary search to get the optimal profit, maximum of $\log \sum p_i$ runs.

(where M is the capacity of the Knapsack optimization problem)

Min Bound　　　Optimal Profit　　　Max Bound

0　　　　　　　　　　　　　　　　　$\sum p_i$

Search for the optimal solution

|________________________|________________________|

# The Classes of P and NP Problems

- **P Class and Deterministic Turing Machine**
  - Given a decision problem X, if there is a Deterministic Turing Machine program that solves X in polynomial time, then X is belong to P Class.
  - ***Informally, there is a Polynomial Time Algorithm to solve the problem of P complexity class.***

# Complexity Class P

- Deterministic in nature
- Solved by conventional computers in polynomial time.
  - O(1) Constant
  - O(log n) Sub-linear
  - O(n) Linear
  - O(n log n) Nearly Linear
  - $O(n^2)$ Quadratic
- Polynomial upper and lower bounds

# NP Problems

- NP stands for non-deterministic polynomial time.
- The basic premise is we could guess at a solution and then test whether we guessed right or not easily. Guessing is of course non-deterministic!
- So, think of this as guessing in polynomial time. No guarantee that you will ever guess correctly though.

- **NP Class and Non-deterministic Turing Machine**
  - Let us consider a decision problem X. If there exists a Non-deterministic Turing machine program that solves a problem X in polynomial time, then X belongs to NP.
  - ***Given a decision problem X. For every instance I of X,***

    ***(a) Guess solution S for I, and***

    ***(b) Check (Verify) "is S a solution to I?"***

***If (a) and (b) can be done in polynomial time,***

***then X belongs to NP.***

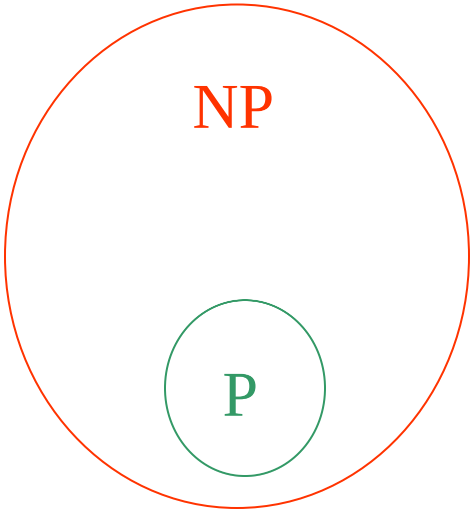
# Complexity Class NP

- Non-deterministic part as well
- Choose (b): Choose a bit in a non-deterministic way and assign to b (i.e. Check (Verify) “is S a solution to I?”).
- If someone tells us the solution to a problem, then we can verify the problem in polynomial time
- Two Properties: (i) Non-deterministic method to generate possible solutions, (ii) Deterministic method to verify in polynomial time that the solution is correct.

# Relation of P and NP

- P is a subset of NP
- “P = NP”?
- Language L is in NP, complement of L is in co-NP
- co-NP ≠ NP
- P ≠ co-NP

- Obvious : $P \subseteq NP$, i.e. A (decision) problem in P does not need "guess solution".
  The correct solution can be computed in polynomial time.




- Some problems which are in NP, but may not in P:
  - 0/1 Knapsack Problem
  - PARTITION Problem: Given a finite set of positive integers Z.
    Question: Is there a subset Z' of Z such that
    Sum of all numbers in Z' = Sum of all numbers in Z-Z' ?

$$\text{i.e. } \sum Z' = \sum (Z-Z')$$

- One of the most important open problem in theoretical compute science:

**Is P=NP ?**

Most likely “No”.

There are many known (decision) problems in NP, but there are no solutions to show anyone of them in P.

# Polynomial Time Reducibility

- There are many mappings (called **reductions**) that have been discovered that map one problem to the other, so if we solve one we can solve the other.

- These mappings are polynomial time mappings.

# Polynomial-Time Reducibility

- Language L is polynomial-time reducible to Language M if there is a function computable in polynomial time that takes an input x of L and transforms it to an input f(x) of M, such that x is a member of L if and only if f(x) is a member of M.

- In Shorthand, L $\xrightarrow{poly}$ M means L is polynomial-time reducible to M

# NP-Hard and NP-Complete

- Language M is NP-hard if every other language L in NP is polynomial-time reducible to M
- For every L that is a member of NP, $L \xrightarrow{poly} M$
- If Language M is NP-hard and also in the class of NP itself, then language M is NP-complete

# NP-Hard and NP-Complete

- Restriction: A known NP-complete problem M is actually just a special case of L
- Local Replacement: Reduce a known NP-Complete problem M to L by dividing instances of M and L into “basic units” then showing each unit of M can be converted to a unit of L
- Component Design: Reduce a known NP-Complete problem M to L by building components for an instance of L that enforce important structural functions for instances of M.

# NP-Complete Problems

- The set of NP problems that can be mapped to each other in polynomial time is called NP-Complete.
- It is difficult to prove that something can not be done.
- We know how to solve many of these algorithms in exponential time, whose to say that one of our bright students won't come up with a clever polynomial time algorithm!
- NP-Complete is useful from that standpoint, if any of them turns out to have a polynomial time algorithm, then they all do provide solutions (hence P=NP).
- These problems are looked at from every angle and no such solution will ever be found (hence, $P \neq NP$).

# NP-Complete Problems

Stephen Cook introduced the notion of NP-Complete Problems. This makes the problem "P = NP ?" more interesting to study with the following key things represented by Stephen Cook.

1. Polynomial Transformation (" $\propto$ ")

   - **L1 $\propto$ L2** :
     There is a polynomial time transformation that transforms arbitrary instance of L1 to some instance of L2.

   - **If L1 $\propto$ L2 then L2 is in P implies L1 is in P**
     (or L1 is not in P implies L2 is not in P)

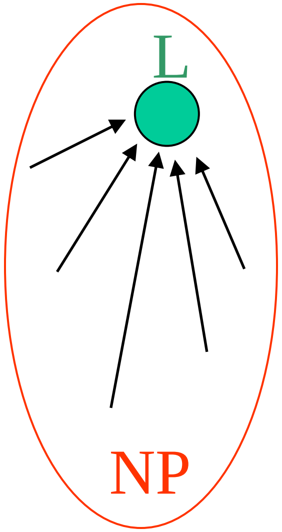
   - **If L1 $\propto$ L2 and L2 $\propto$ L3 then L1 $\propto$ L3**

2. Focus on the class of NP – **decision** problems only. Many intractable (unsolvable) problems, when phrased as decision problems, belong to this class.

3. L is **NP-Complete**

   ***if (#1) L $\in$NP & (#2) for all other L' $\in$ NP, L' $\propto$ L***

   - If an NP-complete problem can be solved in polynomial time then all problems in NP can be solved in polynomial time.
   - If a problem in NP cannot be solved in polynomial time then all problems in NP-complete cannot be solved in polynomial time.
   - NP-Complete problem is one of those hardest problems in NP.




4. L is **NP-Hard** if

   ***(#2 of NP-Complete) for all other L' $\in$ NP, L' $\propto$ L***

   - NP-Hard problem is a problem is as hard as an NP-Complete problem and it is not necessary a decision problem.
   - So, if an NP-Complete problem is in P then P=NP
   - if P != NP then all NP-Complete problems are in NP-P

# Question: How can we obtain the first NP-Complete problem L?

**Cook Theorem: SATISFIABILITY is NP-Complete.**
**(The first NP-Complete problem)**

***Instance : Given a set of variables, U, and a collection of clauses, C, over U.***

***Question : Is there a truth assignment for U that satisfies all clauses in C?***

# Example:

**$U = \{x_1, x_2\}$**

**$C_1 = \{(x_1, \neg x_2), (\neg x_1, x_2)\}$**

**$= (x_1 \text{ OR } \neg x_2) \text{ AND } (\neg x_1 \text{ OR } x_2)$**

**if $x_1 = x_2 = \text{True} \rightarrow C_1 = \text{True}$**

**$C_2 = (x_1, x_2)\ (x_1, \neg x_2)\ (\neg x_1) \rightarrow$ not satisfiable**

**"$\neg x_i$" = "not $x_i$"  "OR" = "logical or"  "AND" = "logical and"**

**This problem is also called "CNF-Satisfiability", since the expression is in CNF – Conjunctive Normal Form (the product of sums).**

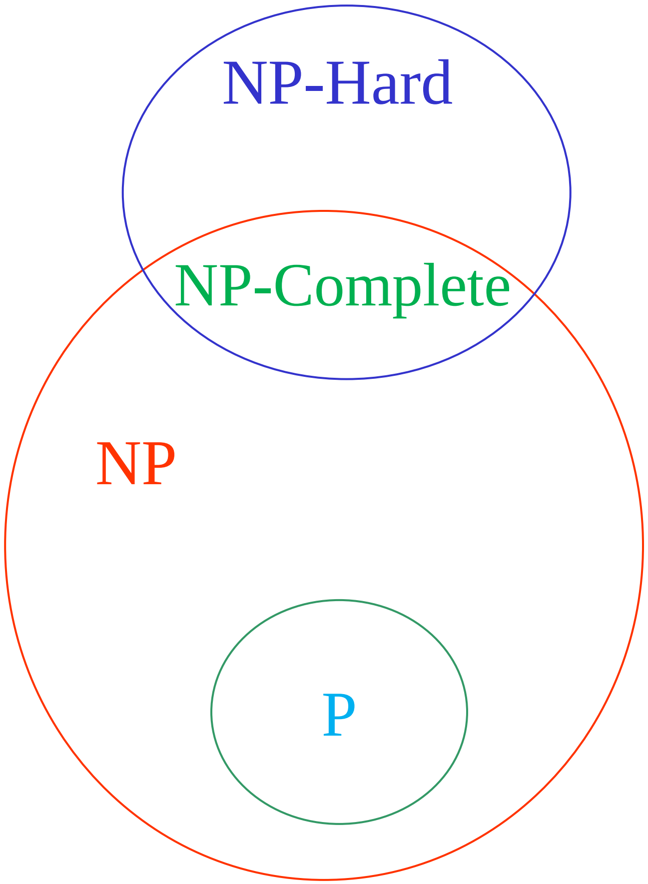
- With the Cook Theorem, we have the following property :

Lemma :
If L1 and L2 belong to NP,
L1 is NP-complete, and L1 $\propto$ L2
then L2 is NP-complete.

*i.e.* $L1, L2 \in NP$ *and for all other* $L' \in NP$, $L' \propto L1$ *and* $L1 \propto L2 \rightarrow L' \propto L2$

- So now, to prove a (decision) problem L to be NP-complete, we need to
  - Show L is in NP
  - Select a known NP-complete problem L’
  - Construct a polynomial time transformation f from L' to L
  - Prove the correctness of f (i.e. L’ has a solution if and only if L has a solution) and that f is a polynomial transformation

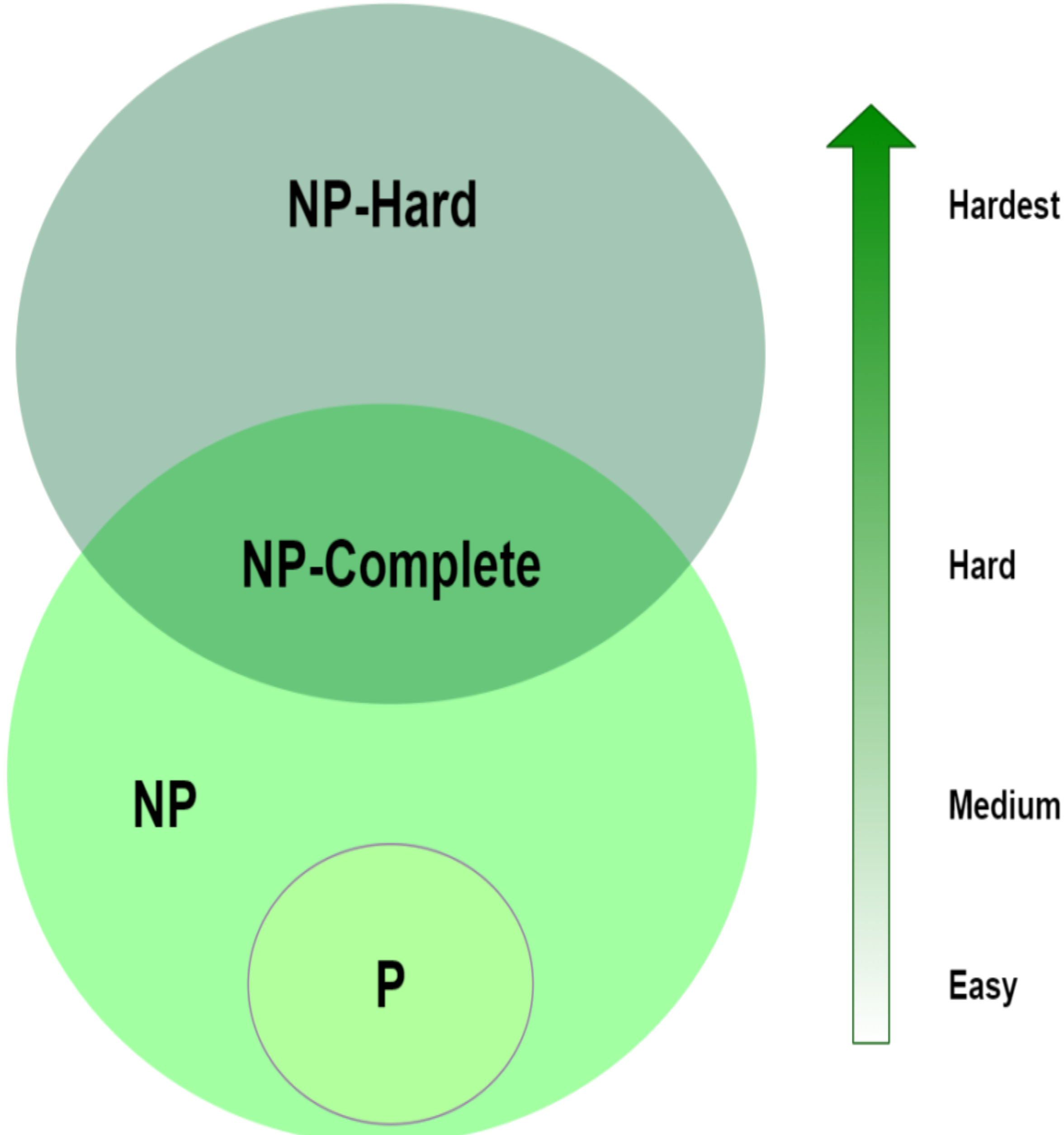
- P: (Decision) problems are solvable by the deterministic algorithms (Turing machine program) in polynomial time
- NP: (Decision) problems are solved by non-deterministic algorithms (non-deterministic Turing machine program) in polynomial time




- A group of (decision) problems, incl. Satisfiability, 0/1 Knapsack, Longest Path, Partition) have an additional important property:

If any of these can be solved in polynomial time, then they all can!

These problems are referred to as NP-Complete problems.


NP-Hard
NP-Complete
NP
P
Hardest
Hard
Medium
Easy

# NP-Complete Problems

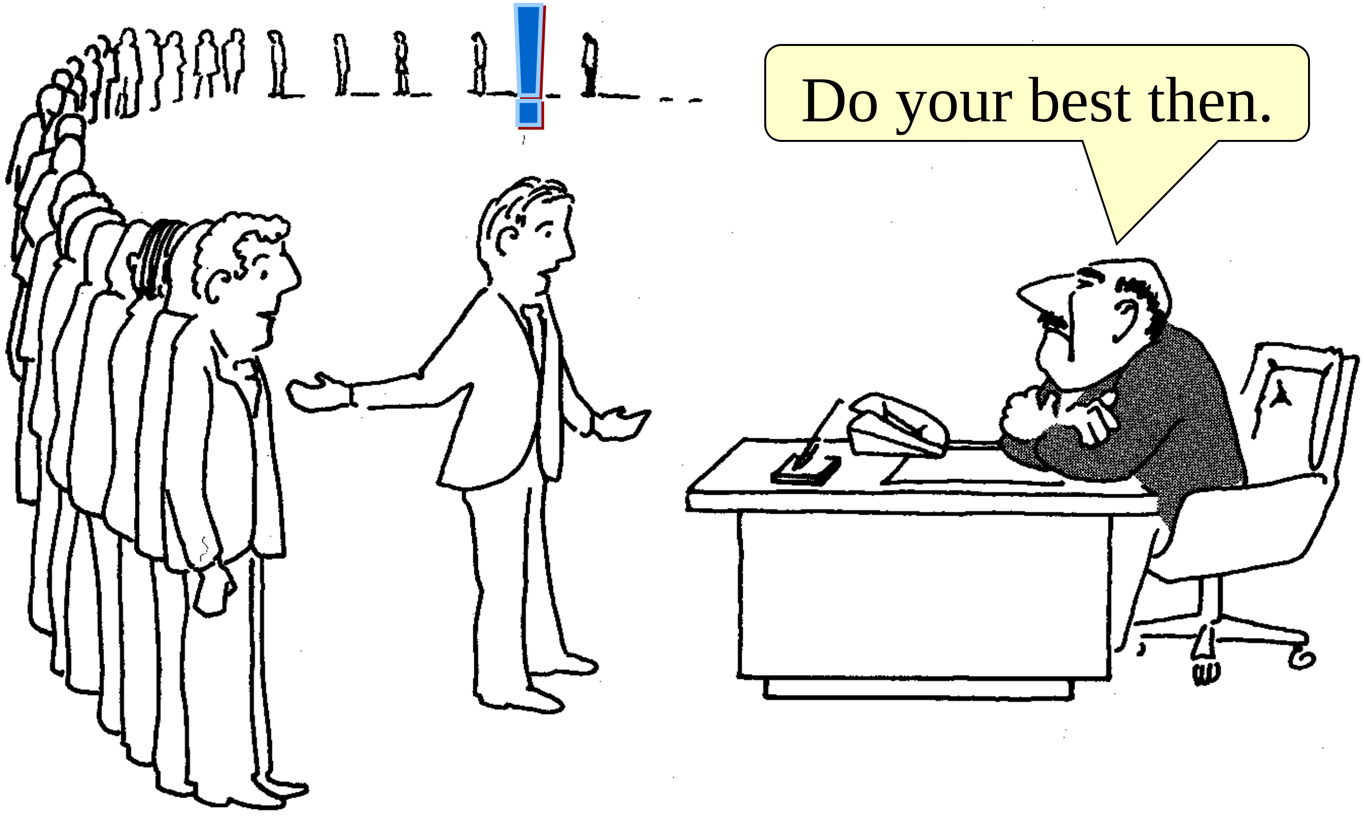
- It is also useful to know these algorithms, as they occur frequently in real applications and tackling them in a brute force fashion may be disastrous.
  - SAT Problem
  - Traveling Salesperson Problem
  - Knapsack Problem
  - Longest Path Problem
  - Graph Clique Problem

# NP-Complete

- It is also interesting to look at the relationship between some easy problems and hard ones:

| **Hard Problems (NP-Complete)** | **Easy Problems (in P)** |
|---|---|
| SAT, 3SAT | 2SAT |
| Traveling Salesman Problem | Minimum Spanning Tree |
| 3D Matching | Bipartite Matching |
| Knapsack | Fractional Knapsack |

# NP-Completeness




"I can't find an efficient algorithm, but neither can all these famous people."

# *Coping With NP-Hardness*

- **Brute-force Algorithms.**

  - Develop clever enumeration strategies.
  - Guaranteed to find optimal solution.
  - No guarantees on running time.

- **Meta-Heuristics Based Algorithms.**

  - Develop intuitive algorithms.
  - Guaranteed to run in polynomial time.
  - No guarantees on quality of solution.

- **Approximation Algorithms.**

  - Guaranteed to run in polynomial time.
  - Guaranteed to find "high quality" solution, say within 1% of optimum.

Challenge: We need to prove a solution's value is close to optimum solution, without even knowing what optimum value is!

# Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard optimization problem. What should I do?

A. Sacrifice one of three desired features.

i. Runs in polynomial time.
ii. Solves arbitrary instances of the problem.
iii. Finds optimal solution to problem.

ρ-approximation algorithm.

- Runs in polynomial time.
- Solves arbitrary instances of the problem
- Finds solution that is within ratio ρ of optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what is optimum value.