Lecture 3 – Basic Algorithm Design and NP-Completeness

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

- Brute Force
- Enumeration
- Recursion and Backtracking
- NP-Completeness

Brute Force

A Problem Formulation

- **Naïve formulation for generic problems**: find/count/verify the solution that yields the best outcome satisfying certain constraints C

$$\arg_{\boldsymbol{x}} \max f(x)$$

$$\text{s.t. } \{\mathrm{c}(x) = True | c \in C\}$$

- Example
  - Select 5 students in a group that performs the best as a basketball team
  - Find the largest common divisor of any two number
  - Find the parameters that minimizes an MLP's empirical loss on a given dataset

Brute Force

- Definition: solving a problem by trying every possible solution
- Characteristics:
  - **Comprehensive**: Examines all possibilities.
  - **Simple**: Often the most straightforward approach to understand.
  - **Inefficient**: Can be slow or resource-heavy, especially for large input sizes

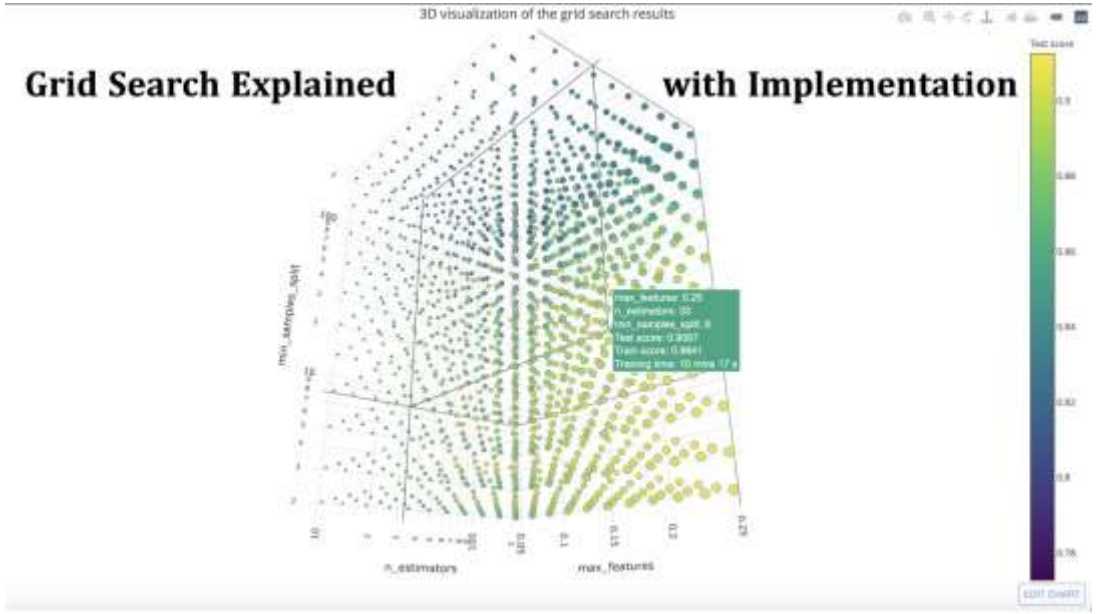
Unlock by trying all the combinations

```
3gA8bjqI m5MoKRJJ 0WZCBfwI zm9jqUiN glYqI53X eokUQTMr LDzoPv9e qXUCvXrT Enju6m2B
oiKqleah 9WtsbkvW W3gyedbV 12j9P5xy 8SxkUE9w nXXADMm5 GiB1Z6l0 jio5wIbp gozIsTIm
lzOWmgPt i9XnarBa yo0acST0 QlacFGJq IZkFcv0u 5ugNFvlw w2Gl7xIN w2bejXwY RvYxtMjU
nRlO6HW5 cHRSQrAb DoqQQrgd RTUevfn2 faFLIMRf MmlnJoZE JilIRGJ5 crEHftjc eDmS5WJp
RfVugGsl RgsGp0oU tHd4lyuV L0x6wZkG IdqZQtxK TIBeHAVa XYVA4UQI VXoncM5C uk3Eaauk
dB90qMas DJrU178v nCZrMpk8 Xaup9lw5 miufIFTB Z9p5K9Ut suAjQYep x5CosAtw bHTdQPkj
A1gr9Utx Pik7Il04 vaYlGHjc BtJ8ktAk au0greB1 P9FTnI7c a6UErOcr iEp0z3tC Hzg7iYWZ
6FFcHAoe Yfa3SY5I 351sV8w5 J3PxPYNz WGjLGuBW c2503M6c pDfsu2Q4 cdP5cwB5 9vFjEHQu
2MxkJa3i B4bLGWH4 UIJcx0ns IMT1fNa3 PASSWORD mfviEj5x EsKPneug GKJUOutG FK92JFQ3
rPlowpJr Yr30oFJ5 GHcDJqvx A3QA5Ye3 YbtwXwnn NGJLCNL8 2vJsptvH zCinx0EC UN3j3pXC
vmjRD4iO Q1kh5j6Y 5i6TSEaT lId407YG deYv9OSn 2nczWHh6 vFXjiFRI 4sDHxCZm Qpe5zL30
4eggPjtZ KRfuFRnU VtQhz1v9 XV9DkP4x S9mMEd5S bXyfJTGK NQxNSTOH qfSCnY1M WjJz8X2c
9rpYjpuU ZS69eKWL 7iMwKrlo mtCQSeYd mmam9dn9 5ha4ddzy o9KYUF5Y fJAzwIdn zzHoKGY1
DDGTFjZL Yt7Fm3lV zqJ8pdW1 7YcJfnB9 Soywq9fK 3sA0ewmA zHJyTRNe UupQOTkY XJpvqp2B
q3LWOtcJ 4Pm6aoip iE9NzJgc ntSxFnxl qW7eAqKE 1VmD6lf0 5uzTxti6 2gRsURBz yxseYGgg
beCDYzD2 dcrV4A54 jaT6KoQH MtEulhJ3 xt67N62g zKQIfxdi KbBFpDhY Qd5PGAPW csfwIRrf
UAidOMw8 ZDqQ2xZq QIJfG6Se prhomHT2 scLAHNAS NmIQ0UFQ 7TqPhSZP kp7MUZMk WSKd1TOU
```

Enumeration attack

Example: Grid Search

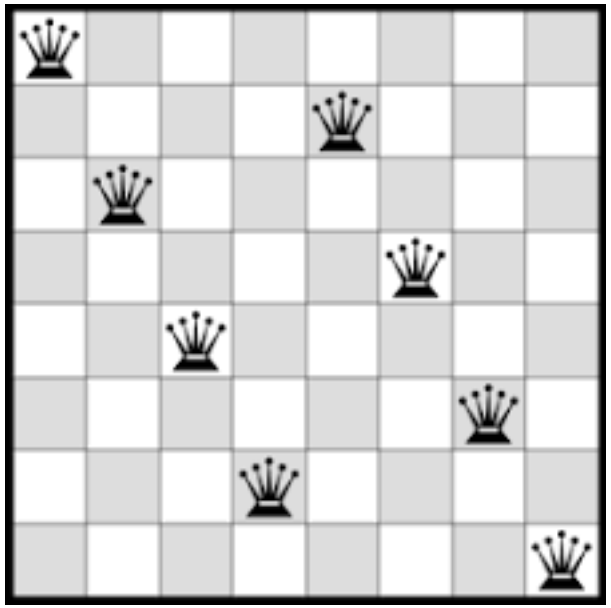
- Super-parameters largely influence the performance of model training
- Grid Search: a popular method for choosing super-parameters
  - Predefine the options for each parameter
  - Validate the model performance when using the combination of all the parameter options
  - Choose the best combination



Enumeration

Eight Queens Problem

- Find all the ways to place eight chess queens on an 8x8 chessboard so that no two queens threaten each other (cannot share the same row, column, or diagonal)
- First posed in the mid-19th century

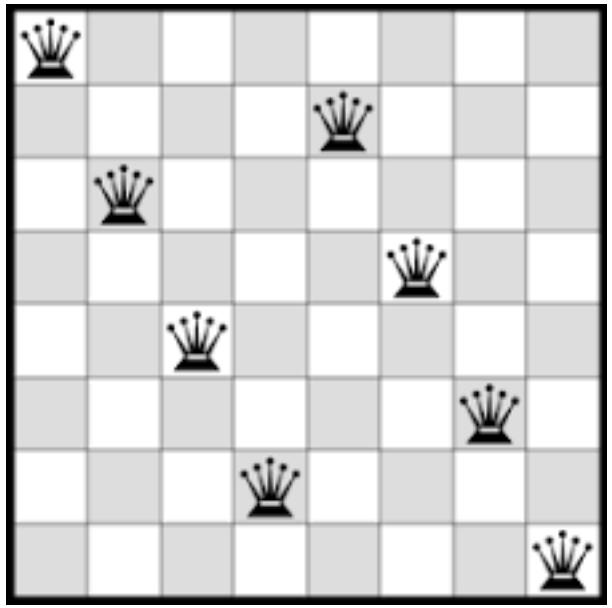


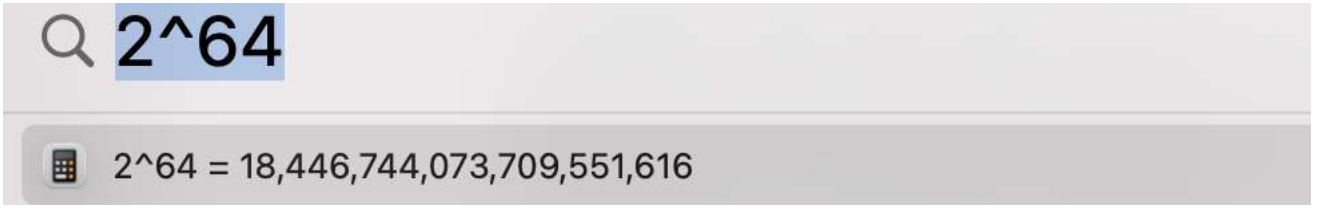
Enumeration for Brute Force

- List all possible solutions without repetition and traverse to find the best solution
- Useful when solutions can be easily listed
- Still different strategies for enumeration (especially when there are constraints)
    - Key point: make a concise possible list

Eight Queens Problem

- Listing all the possible solutions to a problem
- Checking if they satisfy the given requirements



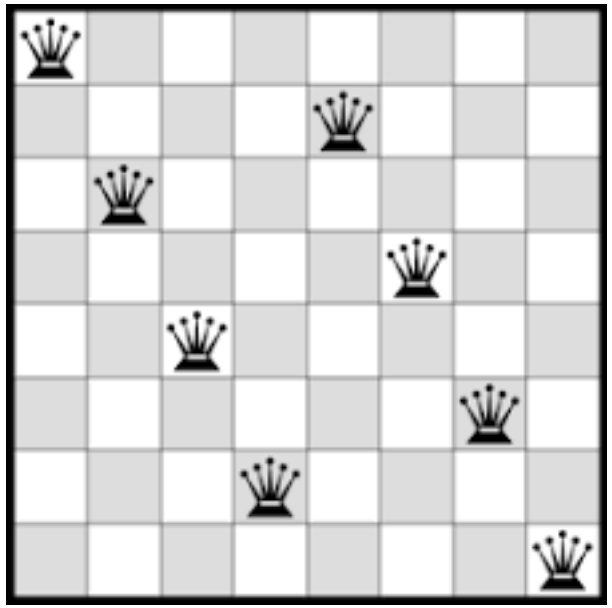


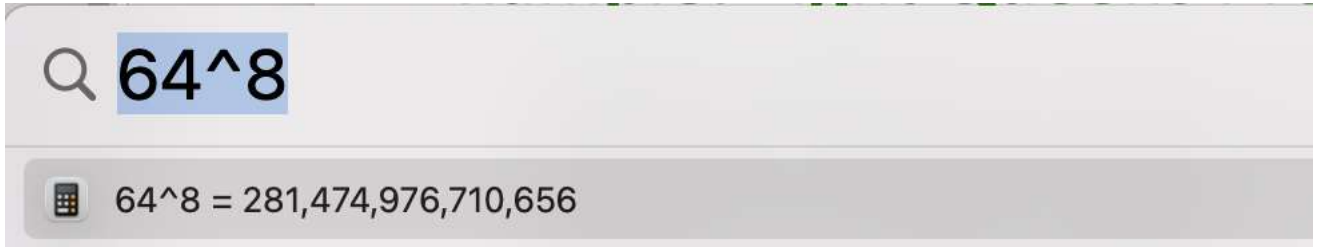
Enumeration 1:

- Enumerate all the possible $X \in R^{8\times 8}$, with $x_{ij} \in \{0, 1\}$
- 2^{64} enumerations $O(2^{n^2})$
- Infeasible solutions: X containing more than eight queens

Eight Queens Problem

- Listing all the possible solutions to a problem
- Checking if they satisfy the given requirements



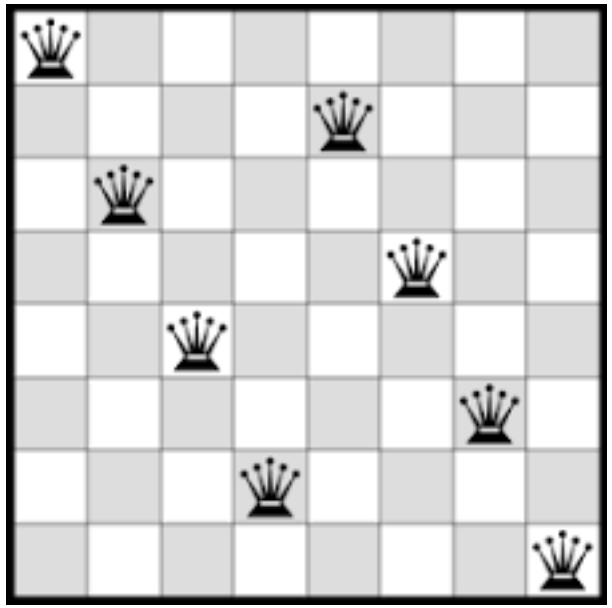


Enumeration 2:

- Enumerate the positions of the 8 queens
- 64^8 candidate solutions O(n^{2n})
- Infeasible solutions: two queens at the same position
- Redundant solutions: two queens switch positions (n! Redundant solutions for each solution)

Eight Queens Problem

- Listing all the possible solutions to a problem
- Checking if they satisfy the given requirements

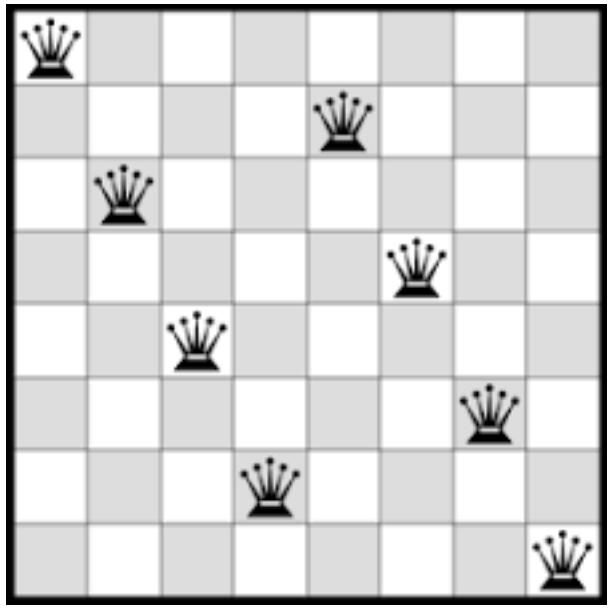


Enumeration 3:

- Pick 8 distinct grids from 64
- C(64, 8) candidate solutions 4,426,165,368
- $\lim_{n\to\infty} \frac{C(2n,n)}{4^n} = 0$, so $C(2n,n) = O(4^n)$
- Infeasible solutions: two queens at the same row/columns

Eight Queens Problem

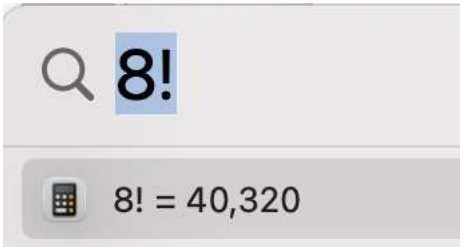
- Listing all the possible solutions to a problem
- Checking if they satisfy the given requirements



Enumeration 4:

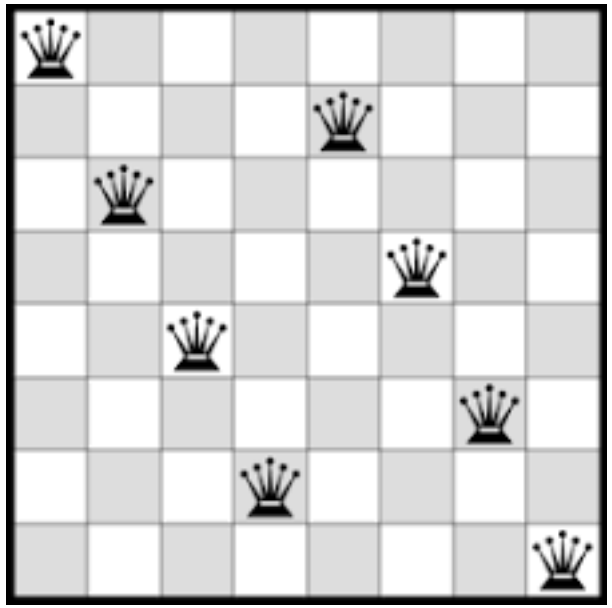
8 rows, 8 queens -> 1 queen for each row (pigeonhole principle)

- Decide the column for each row's queen, no duplication
- Enumerate permutation of numbers {1, 2, …, 8}
- 8! candidate solutions
- $O(n!)$



Eight Queens Problem

- Listing all the possible solutions to a problem
- Checking if they satisfy the given requirements



```python
import itertools

n = 3  # Replace with the value of n you're interested in
perm = itertools.permutations(range(1, n + 1))

for i, p in enumerate(perm):
    print(f"Permutation {i + 1}: {p}")
```

```yaml
Permutation 1: (1, 2, 3)
Permutation 2: (1, 3, 2)
Permutation 3: (2, 1, 3)
Permutation 4: (2, 3, 1)
Permutation 5: (3, 1, 2)
Permutation 6: (3, 2, 1)
```

Only checking whether two queens share the same diagnosis

Example: Equations

Given the equation $ax_1^2 + bx_2^2 + cx_3^2 + dx_4^2 = 0$, where a, b, c, d are non-zero integers in the range $[-50,50]$, $x_1,\ x_2,\ x_3,\ x_4$ are integers in the range $[-1000,1000]$. Find the total number of solutions $(x_1,\ x_2,\ x_3,\ x_4)$ that satisfy the equation.

Solution 1: Brute force all the combination of values

```
01. def count_solutions(a, b, c, d):
02.     count = 0
03.     for x1 in range(-1000, 1001):
04.         for x2 in range(-1000, 1001):
05.             for x3 in range(-1000, 1001):
06.                 for x4 in range(-1000, 1001):
07.                     if a*x1**2 + b*x2**2 + c*x3**2 + d*x4**2 == 0:
08.                         count += 1
09.     return count
```

Time Complexity: $O(n^4)$

http://acm.hdu.edu.cn/showproblem.php?pid=1496

Example: Equations

Given the equation $ax_1^2 + bx_2^2 + cx_3^2 + dx_4^2 = 0$, where a, b, c, d are non-zero integers in the range $[-50,50]$, $x_1,\ x_2,\ x_3,\ x_4$ are integers in the range $[-1000,1000]$. Find the total number of solutions $(x_1,\ x_2,\ x_3,\ x_4)$ that satisfy the equation.

Solution 2 (hint: combine with hash table)

- Transform the problem: $ax_1^2 + bx_2^2 = -(cx_3^2 + dx_4^2)$
- Brute force values of $x_1,\ x_2$, check how many $-(ax_1^2 + bx_2^2)$ can be obtained by combination of $x_3,\ x_4$
- Solution

  Total Time Complexity: $O(n^2)$ Hash table: O(1)

  - Brute force combinations of $x_3,\ x_4$, store all possible $cx_3^2 + dx_4^2$ in a hash table $O(n^2)$
  - Brute force combinations of $x_1,\ x_2$, search for $-(ax_1^2 + bx_2^2)$ in the hash table $O(n^2)$

http://acm.hdu.edu.cn/showproblem.php?pid=1496

Recursion and Backtracking

Backtracking

n! is still large.

Can we further reduce the time cost for eight queens problem?

- Motivative example

1 2 3 4 5 6 7 8

1 2 3 4 5 6 8 7

1 2 3 4 5 7 6 8

1 2 3 4 5 7 8 6

1 2 3 4 5 8 6 7

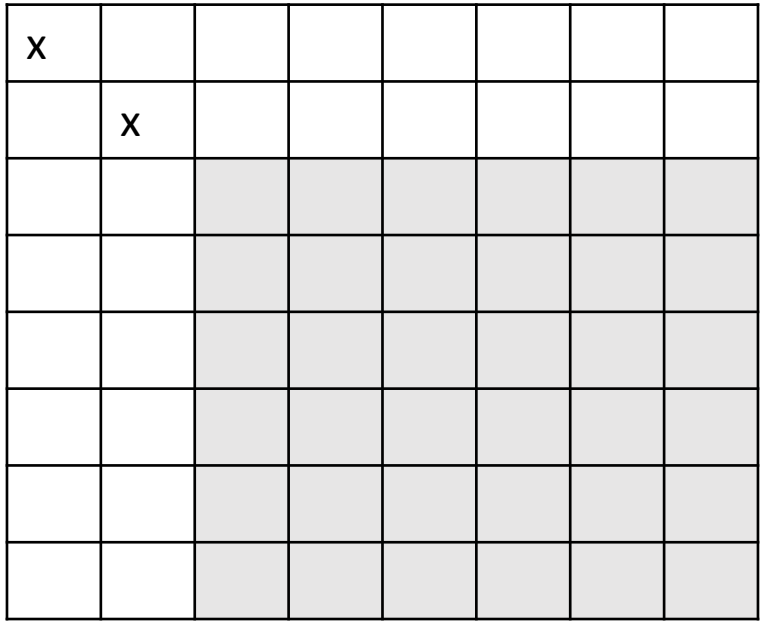
1 2 3 4 5 8 7 6

….

8 7 6 5 4 3 2 1

Share the prefix

Observation: 1 2 … are all infeasible!

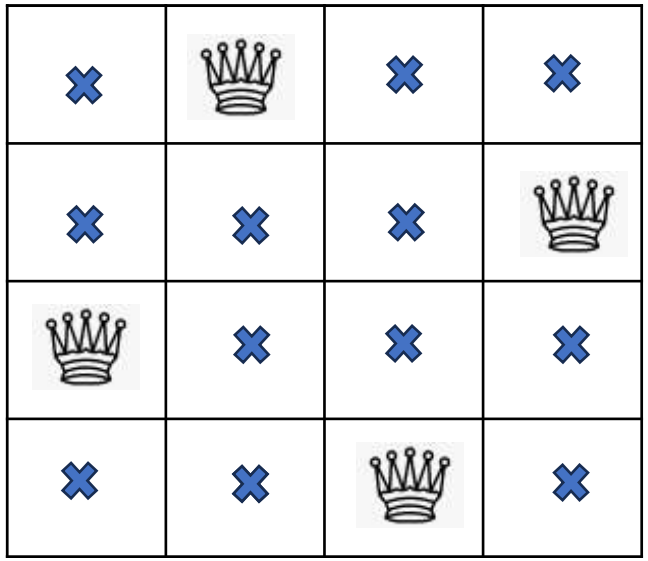


- Incrementally construct the solution

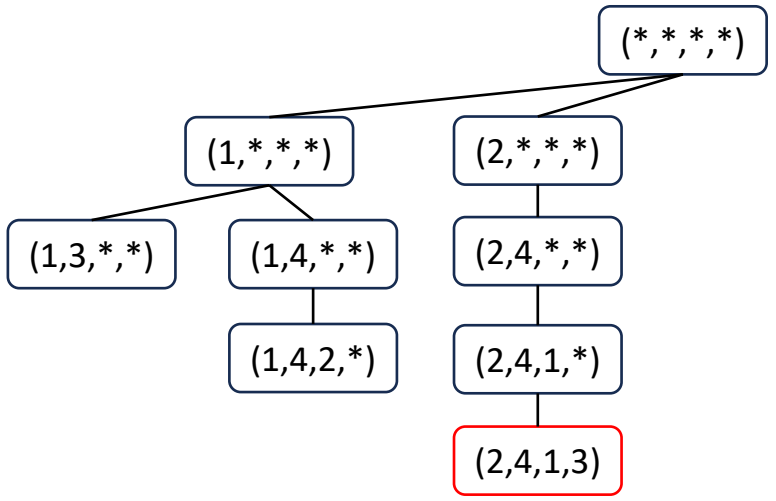
Backtracking

Explore all possible solutions incrementally and backing up when no feasible solutions available at the current stage

- Construct solutions step-by-step, explore states instead of the whole permutation
- Abandon a state when we find it lead to no valid solution



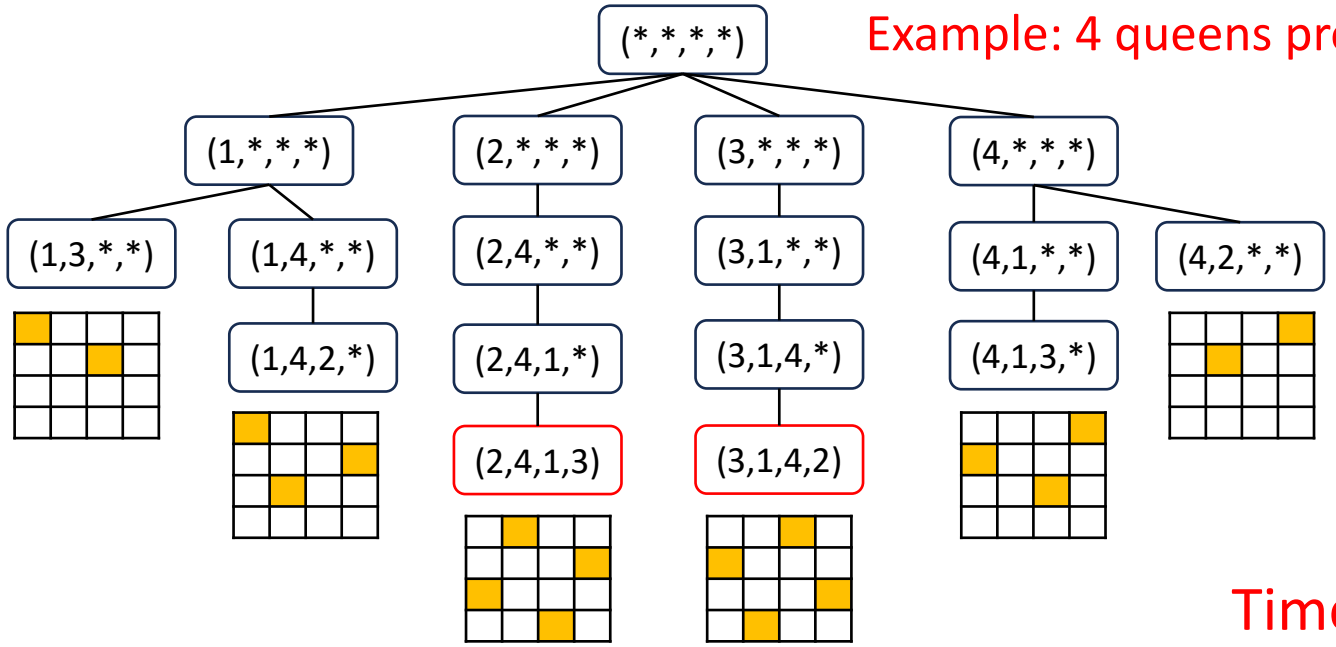
Example: 4 queens problem



Backtracking

Explore all possible solutions incrementally and backing up when no feasible solutions available at the current stage

- Construct solutions step-by-step, explore states instead of the whole permutation
- Abandon a state when we find it lead to no valid solution



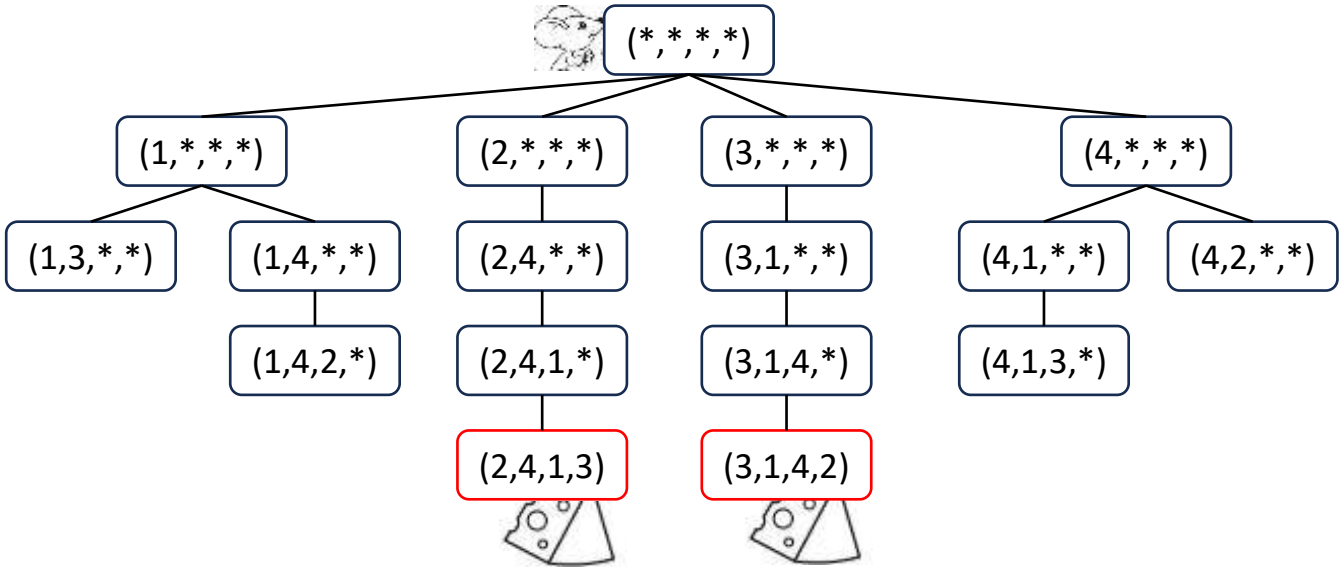
Example: 4 queens problem

Time complexity: the number of valid states

Backtracking

Intuition: a mouse seeking for cheese in a maze

- Move forward and search each possible way
- Move back to the previous crossroad when encountering a dead end



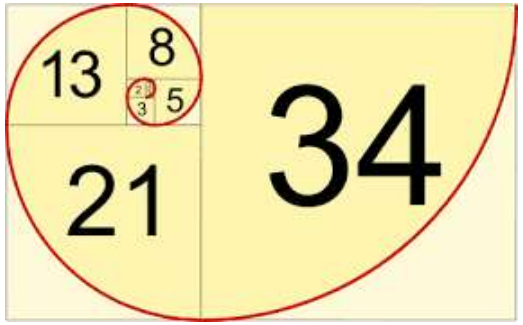
Implementation?

Recursion

- Recursion: the definition of a concept/process depends on a simpler version of itself
- Example: Fibonacci sequence

$$F_n = \begin{cases} 0 & if\ n = 0; \\ 1 & if\ n = 1; \\ F_{n-1} + F_{n-2} & if\ n \geq 2. \end{cases}$$

1 1 2 3 5 8 13 21 34 ...



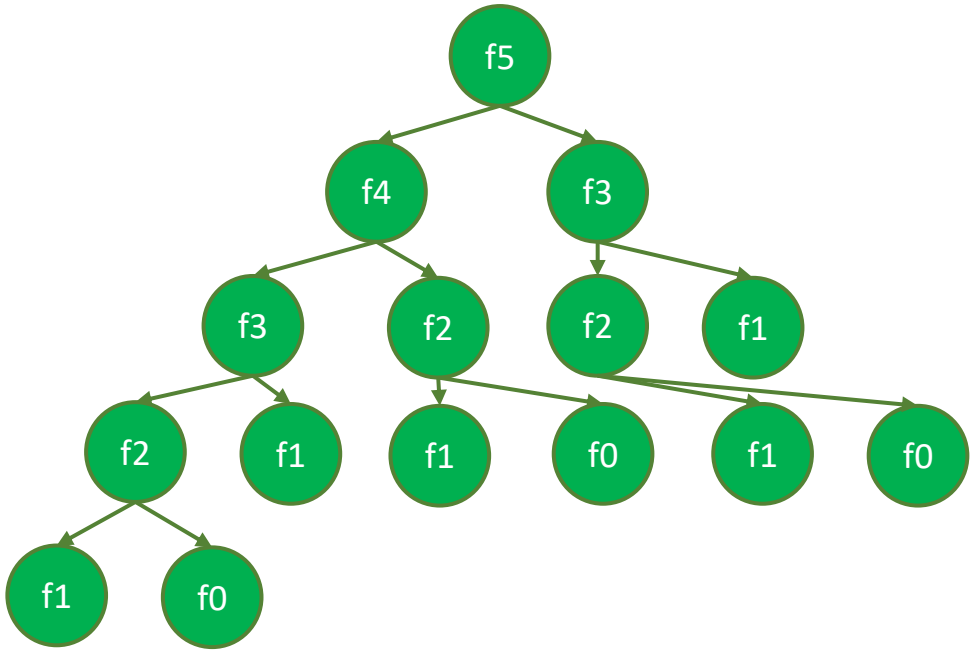
Fibonacci Spiral

- Recursion Algorithm: Solving a problem depending on solving smaller instances of the same problem
- **Recursive case**: instances to be solved recursively
- **Base case**: instances that directly obtain the result

Recursion

Example: calculate Fibonacci number fib(n)

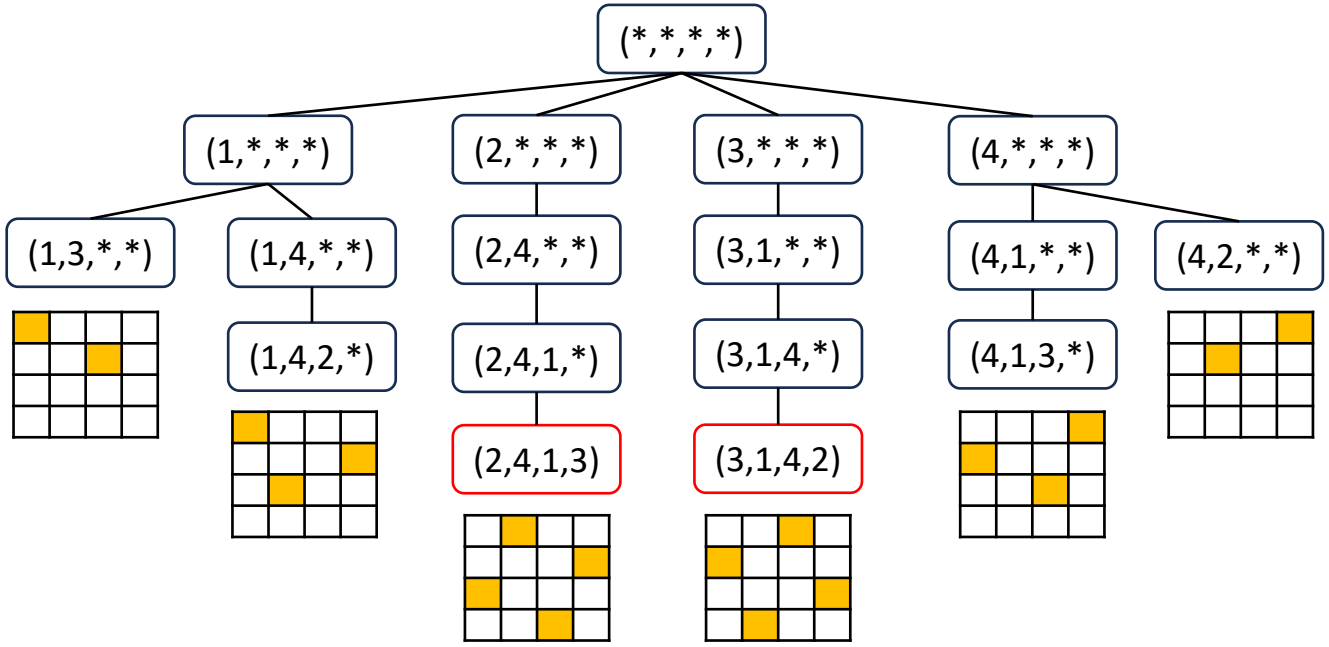
```
01. fib(n):
02.     if n<=1:
03.         return n
04.     else:
05.         return fib(n-1) + fib(n-2)
```



Backtracking: View from Recursion

Define the problem: given the current state S of permutation, find the solutions $f(S)$ that can be expanded from the current state

Solve the problem recursively: assuming $S_0, S_1, \ldots S_k$ are the states that can be expanded from S in one step, $f(S) = f(S_0) \cup f(S_1) \ldots f(S_k)$

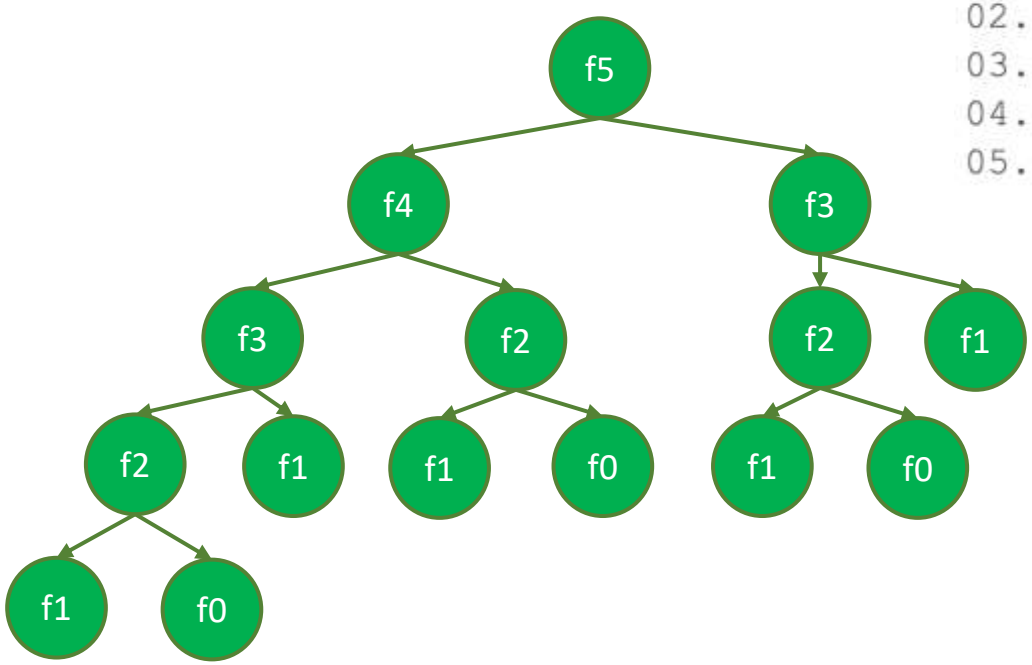



```
01. def explore(current_state):
02.     if len(current_state) == 4:  # current state
03.         return {current_state}
04.     result = {}
05.     for u in range(4):
06.         next_state = current_state.add(u)
07.         if is_valid_state(next_state):
08.             result.union(explore(next_state))  #
09.     return result
```

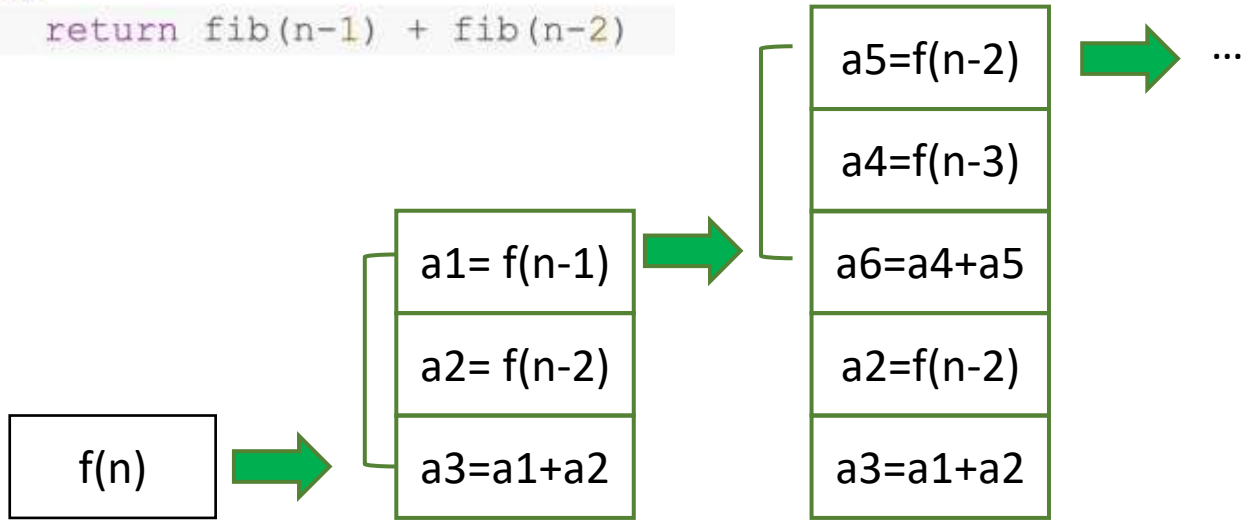
Recursion

How backtracking (return to previous states) happens during recursion?

- Operations dynamically organized in a stack
- Implicit backtracking: start the next branch after a branch has been popped



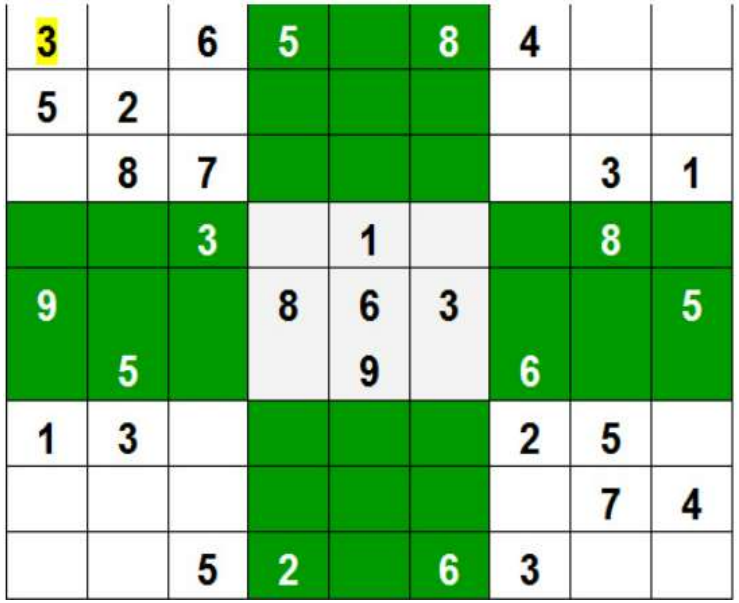

```
01. fib(n)
02.     if n <= 1
03.         return 1
04.     else
05.         return fib(n-1) + fib(n-2)
```



Example: Sudoku

Sudoku is a puzzle consisting of a 9x9 grid, divided into 3x3 subgrids (boxes)

- Input: a sudoku board with some cells are pre-filled with numbers 1-9
- Goal: Complete the grid by filling in numbers from 1 to 9 in such a way that each row, column, and 3x3 sub grid contains each number exactly once



- Simple backtracking: high complexity
- Backtracking + rules
  - e.g., middle box and 5

NP-Completeness

The hardness of a problem

Core aim in this section: discuss the hardness of solving a problem

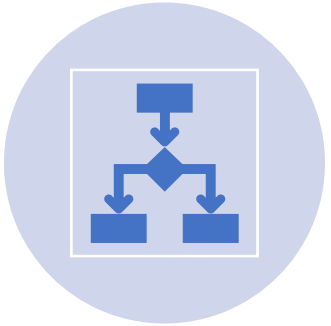
General categories:

- **P**: Problems that can be solved quickly
- **NP**: Problems for which solutions can be checked quickly
- **NP-complete**: The 'hardest' problems in NP
- **NP-hard**: Problems at least as hard as NP-complete problems, but not necessarily in NP

Type of Problems

Class P and NP are defined on decision problems

Optimization problems: Find a feasible solution leading to the best outcome



Decision problems: Problems that outputs simply “yes” or “no”

Optimization Problems

- **Optimization problem**: find a feasible solution with the best outcome

$$\arg_r \max f(r)$$
$$\text{s.t. } \{c(r) = True | c \in C\}$$

- **Searching problem** can be framed as an optimization problem: find a feasible solution

$$\arg_{\boldsymbol{r}} \max\ 1$$
$$\text{s.t. } \{c(r) = True | c \in C\}$$

  $f(r) = 1$ in the optimization problem

- Examples:
  - Find the largest common divisor of any two number
  - Find the shortest path on a given graph
  - Find a cycle in a direct graph

Optimization Problems

- **Optimization problem**: find a feasible solution with the best outcome

$$\arg_r \max f(r)$$
$$\text{s.t. } \{c(r) = True | c \in C\}$$

Optimization Algorithm A: an algorithm that solves an optimization problem

- Given: an instance (input) x of the problem
  - Example: in largest common divisor problem, the inputs are the two numbers
- $A(x)$ is the feasible solution with the best outcome

Decision Problem

Problems that simply requires the answer “yes” or “no”

Examples:

- Is string s a palindrome (i.e., read the same forward and backward)?
- Is a graph G connected (i.e., is there a path between any two vertices)?
- Does a given number have a square root that is an integer?

Decision Problem

Problems that simply requires the answer “yes” or “no”

Decision Algorithm A: an algorithm that solves a decision problem

- Given: an instance (input) x of the problem
  - Example: palindrome testing, the input is the string
- $A(x) = 1$ - algorithm A accepts x
- $A(x) = 0$ - algorithm A rejects x

Optimization & Decision Problem

Optimization problems has related decision problems

- Searching: ask for existence
- Optimization: ask for the bound of the outcome value

Given	Optimization	Decision
A directed graph G, vertices u and v	(SHORTEST-PATH) find the shortest path from u to v	(PATH) Does a path exists from u to v consisting of at most k edges?
Two numbers	Find the largest common divisor	Is there a common divisor >= k?
Direct graph	Find a cycle	Is there a cycle?

Optimization & Decision Problem

The corresponding decision problem cannot be more difficult than the optimization problem

- Solving a optimization problem leads to answering the decision problem
  - Answer "*given a graph, does a path exists from u to v consisting of at most k edges?*" (Decision) by comparing the number of edges in the *shortest path* (Optimization) to k
  - Answer "*given two numbers, is there a common divisor >= k?*"(Decision) by comparing the *largest common divisor* (optimization) with k
  - Answer "*given a graph, is there a cycle in a graph?*" (Decision) by *finding an specific cycle* (optimization).
- Answering the decision problem doesn't necessarily solve the optimization problem
  - Know the best outcome does not mean we can easily find the solution that leads to it
- A decision problem is difficult -> its optimization problem is difficult (NP-completeness talks about difficulty)

The Class P

The difficulty of a problem: whether can be solved in polynomial time

- P (Polynomial time): the collection of decision problems that can be solved in polynomial time, i.e., $O(n^k)$, where n is the size of the input and k is a constant
  - Approximates the ”practically solvable problems”
  - Lower-order like $O(\log n)$ is also $O(n^k)$
- Polynomial time: time cost does not grow rapidly, manageable for practical applications
- Other time complexity
  - **Factorial**: e.g., Eight Queens Problem - $O(n!)$, time cost multiplies by $n + 1$ when n increases by 1
  - **Exponential**: e.g., Subset Sum Problem - $O(2^n)$, time doubles when n increase by 1

Further classification for problems without a polynomial-time algorithm?

Verification Algorithms

Consider the difficulty of a simpler problem: verify the answer based on the evidence

- **Verification algorithm**
  - A two-argument algorithm A that takes an ordinary input x and a certificate y
  - A ***verifies*** x if there exists a certificate y such that $A(x, y) = 1$
- **Certificate**: evidence to support a “yes” answer

Verification Algorithms

Example: Boolean SATisfiability Problem (SAT)

- Determine if there exists an assignment of variables that makes a given Boolean formula true
- **Input instance** x: a bool formula such as $(b_1 \vee b_2 \vee b_3) \wedge (\neg b_2 \vee b_3 \vee b_4)$
- Decision algorithm: $A(x)$

Verification Algorithms

Example: Boolean SATisfiability Problem (SAT)

- Determine if there exists an assignment of variables that makes a given Boolean formula true
- **Input instance** x: a bool formula such as $(b_1 \vee b_2 \vee b_3) \wedge (\neg b_2 \vee b_3 \vee b_4)$
- Verification algorithm $A(x, y)$:
  - **Certificate** y: a True/False value assignment for variables that (claims to) make the formula true
    - E.g., b1=True, b2=False, b3=False, b4=False (like one potential r in the optimization formula)
  - Substitute the value into the formula and transform it to a simple boolean expression
    - e.g., $(True \vee False \vee False) \wedge (\neg False \vee False \vee False)$
  - Expression evaluation algorithm with stack
    - $O(N)$, where N is the length of the formula —— Polynomial time!

Non-deterministic polynomial (NP)

- $\boldsymbol{P}$ (Polynomial Time): the collection of decision problems that can be solved in polynomial time, i.e., $O(n^k)$, where n is the size of the input and k is a constant
- $\boldsymbol{NP}$ (Nondeterministic Polynomial Time): the collection of decision problems that can be verified in polynomial time, i.e., $O(n^k)$, where n is the size of the input and k is a constant
  - Given a set of value assignment, SAT problem can be verified in O(N) time complexity, so SAT is NP

Wonder about "non-deterministic"?

Original Definition:

- P: there exists a deterministic Turing machine (DTM) that can solves the problem in polynomial time
- NP: there exists a nondeterministic Turing machine (NDTM) that can solves the problem in polynomial time

(Proven to be equivalent)

P & NP

Original Definition:

- P: there exists a deterministic Turing machine (DTM) that can solves the problem in polynomial time
- NP: there exists a nondeterministic Turing machine (NDTM) that can solves the problem in polynomial time

(Proven to be equivalent)

- $P \subset NP$ is trivial because DTM $\subset$ NDTM
- $P = NP$?
  - One of the most prominent unsolved problems in computer science
  - Difficult to find certificate from exponential number of candidates
  - e.g., Unclear whether SAT is in P: exponential number assignments

Example: Subset Sum

Exercise: prove the following problem to be NP

Given a set of integers S and an integer k, is there a non-empty subset of S that sums to k?

sum = 15

5 + 8 + 2 = 15

Certificate

- A subset T of S

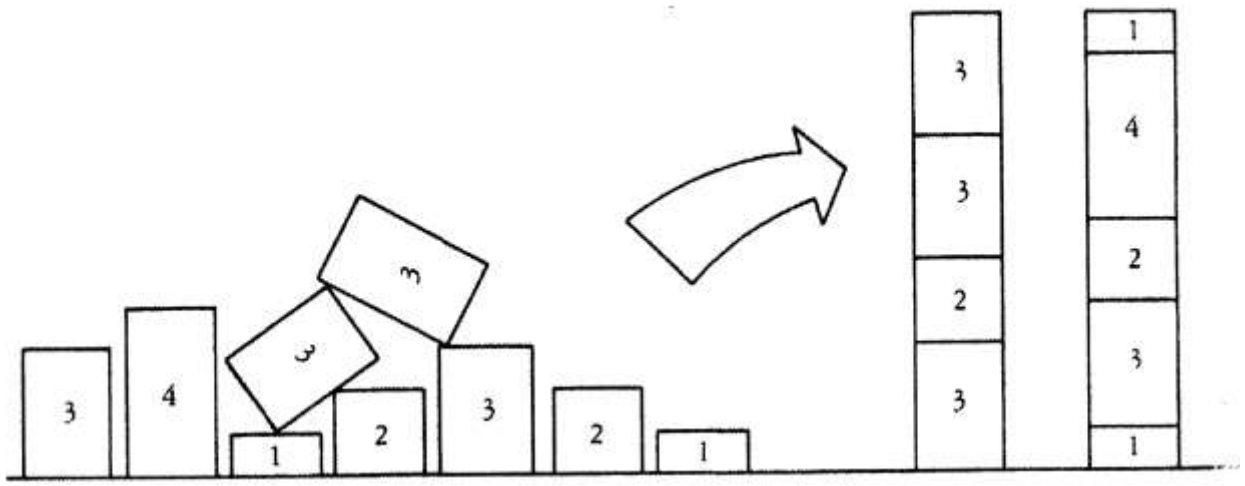
Verification Algorithm

- Iterate elements in T and get the sum $O(N)$
- Check whether the result is k $O(1)$

Example: PARTITION

Exercise: prove the following problem to be NP

Given a set of integers S, can it be partitioned into two subsets S_1 and S_2 such that the $\text{sum}(S_1) = \text{sum}(S_2)$?



Certificate

- Subsets S_1 and S_2

Verification Algorithm

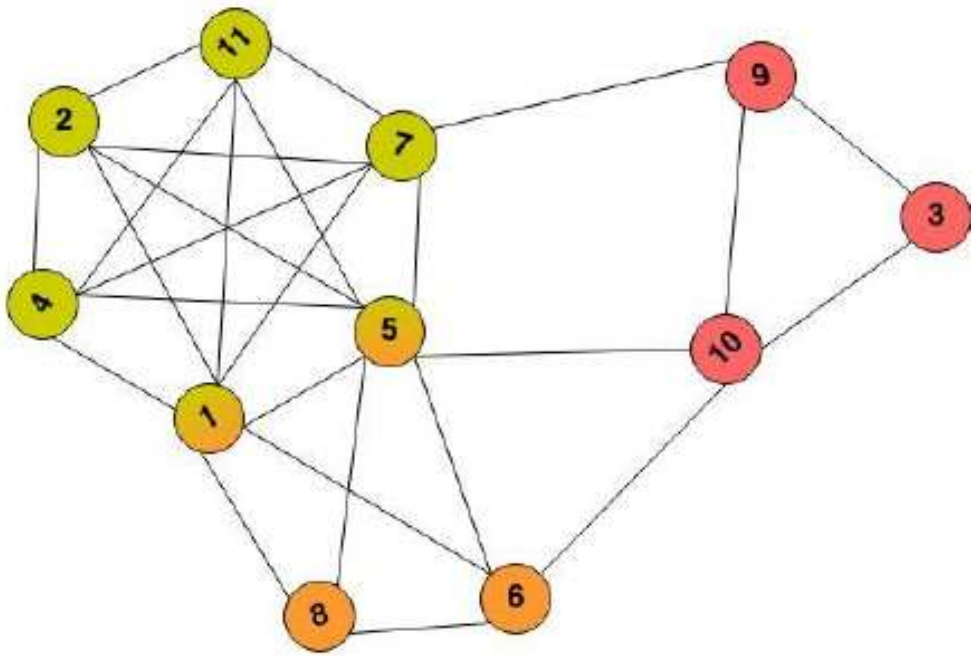
- Check $S = S_1 + S_2$, e.g.,
  - Concatenation $O(N)$
  - Insertion sort $O(N^2)$
  - Check by element $O(N)$
- Get sum of S_1, S_2 $O(N)$
- Check $sum(S_1) = sum(S_2)$

Example: Clique Problem

Exercise: prove the following problem to be NP

Given a graph *G*=(*V*,*E*) and a number *k*, is there a clique of size *k? (supposing |V|=N)*

- *Clique:* a set of vertices where every two are adjacent



Certificate

- A set of vertices S

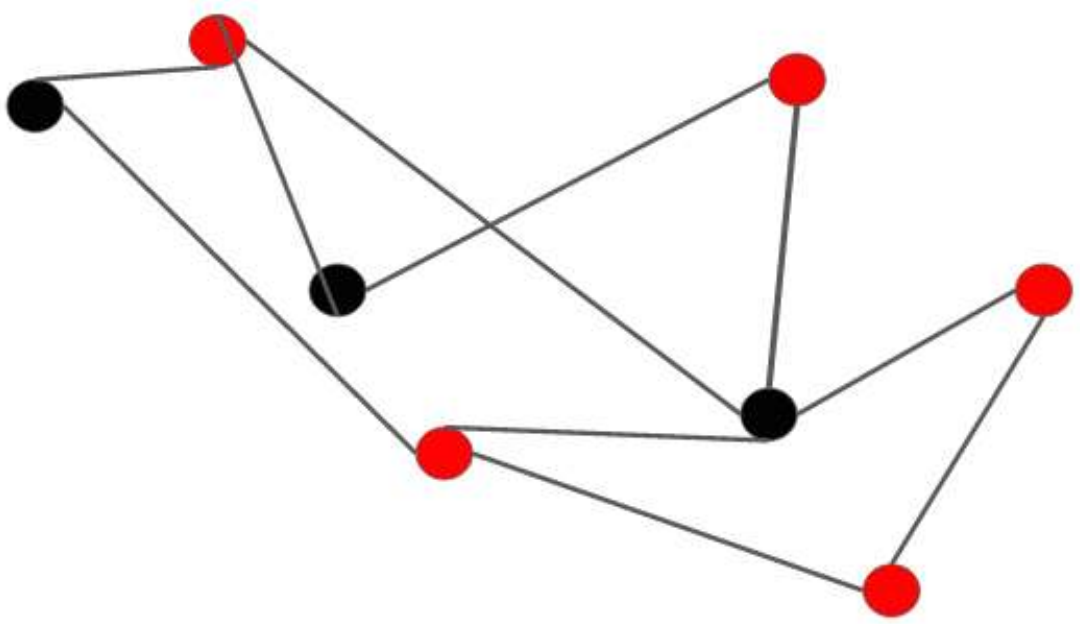
Verification Algorithm

- Check the vertices has size $= k$: $O(N)$
- Check whether there is an edge between any two vertices, e.g.,
  - Create an adjacency matrix M: $O(N^2)$
  - Traverse $i, j \in S$ and check $M_{ij} = 1$: $O(N^2)$

Example: Vertex Cover

Exercise: prove the following problem to be NP

Given a graph $G = (V, E)$ and a number k, is there a set of k vertices such that each edge in the graph is incident to at least one of these vertices?



Select a set of vertices to cover all edges in a graph

Certificate

- A set of vertices S

Verification Algorithm

- Check the vertices has size $= k$: $O(N)$
- Traverse all the edges and check whether there is an end in S
  - Create a direct addressing table for vertices
  - Add vertices in S to the table: $O(N)$
  - Search for each edges' ends in the table: $O(N^2)$

Polynomial-time Reduction

Definition (***polynomial-time reducible***): For decision problems A and B, A is said to be ***polynomial-time reducible*** to B (written $A \leq_p B$) if there is a polynomial-time computable function f such that

q is a Yes-instance of A iff $f(q)$ is a Yes-instance of B

Polynomial-time Reduction

Example: SUBSET-SUM problem is Polynomial-time reducible to PARTITION problem

- **SUBSET-SUM (A)**: Given integers set S and a integer k, is there a subset of S such that sums to k?
- **PARTITION (B)**: Given integers set S, can it be partitioned into S_1 and S_2 with $\text{sum}(S_1) = \text{sum}(S_2)$?
- **Transform Function** f:
  - Problem A: Instance S_A, k
  - Construct $S_B = f(S_A, k) = S_A \cup \{\text{sum}(S_A) - 2k\}$ $(s := \text{sum}(S_A) - 2k)$
- **If S_A, k is yes-instance of A**: partition S_A into S_1 and S_2, so that $\text{sum}(S_1) = k$, $\text{sum}(S_2) = \text{sum}(S_A) - k$. $S_1 \cup \{s\}$ and S_2 have the same sum, making S_B yes-instance for B.
- **If S_B is yes-instance of B**: partition S_B into S_1' and S_2' such that $\text{sum}(S_1') = \text{sum}(S_2')$. wlog, assume $s \in S_1'$. $S_1 = S_1' - \{s\}$ sums to k, thus making S_A, k a yes-instance of A.

Notice: there can be instances of B that can not be transformed from A, so polynomial-time reducible is not bidirectional

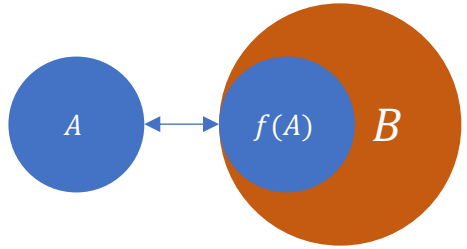
Polynomial-time Reduction

Definition (***polynomial-time reducible***): For decision problems A and B, A is said to be ***polynomial-time reducible*** to B (written $A \leq_p B$) if there is a polynomial-time computable function f such that

$$q \text{ is a Yes-instance of } A \text{ iff } f(q) \text{ is a Yes-instance of } B$$

B is at least as hard as A ($\leq$ is intuitive!)

- Find solution for A, then answer a subset of instances in B
- Find solution for B, then answer all instances in A



Informally: we construct a verification solution to problem A with an efficiency transformation to problem B and a verification solution to Problem B

Polynomial-time Reduction

Definition (***polynomial-time reducible***): For decision problems A and B, A is said to be ***polynomial-time reducible*** to B (written $A \leq_p B$) if there is a polynomial-time computable function f such that

q is a Yes-instance of A iff $f(q)$ is a Yes-instance of B

- Lemma: If $A \leq_p B$ and B in P, then A in P
- Proof: We construct a polynomial-time decider for A as follows
  - For any q:
    - Convert to instance of problem B - $f(q)$: polynomial time
    - Get answer for $f(q)$ in B: polynomial time

NP-completeness

NP-Complete problems: a class of NP problems that are at least “as hard as” any NP problem

Definition(***NP-complete***): A decision problem S is defined to be NP-complete if (1) S is in NP, (2) for all A in NP, it holds that $A \leq_p S$

- Key Implication: If an NP-complete problem is in P (i.e., can be solved in polynomial time), then all problems in NP can also be solved in polynomial time, then P=NP
- In research, we prove a problem to be NP-complete to show how hard it is to solve it (show the significance of you designing a good random/heuristic algorithm)

NP-completeness

NP-Complete problems: a class of NP problems that are “as hard as” any NP problem

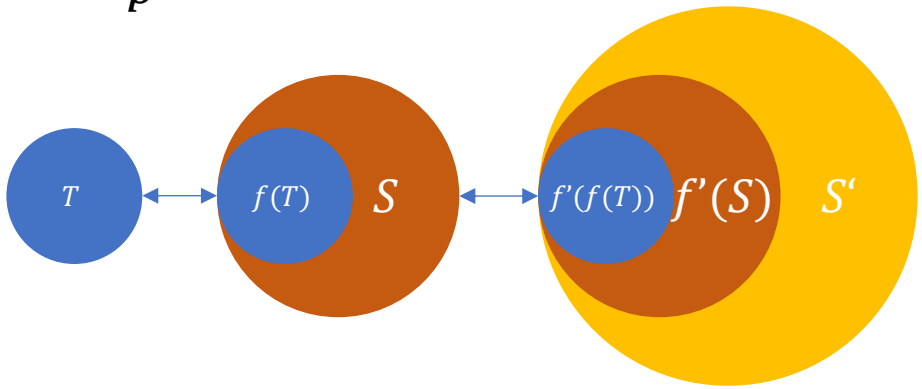
Definition(***NP-complete***): A decision problem S is defined to be NP-complete if (1) S is in NP, (2) for all A in NP, it holds that $A \leq_p S$

- **SAT Problem** is the first problem proved to be NP-Complete
  - Stephen Cook proved that any NP problem is polynomial-reducible to an SAT problem in 1970s, who won the Turing award for this
  - Encode the operation of any nondeterministic Turing machine by a boolean formula
- Since then, hundreds of NP-complete problems have been discovered
  - Much easier

Prove NP-completeness

Key idea: Given an NP-complete problem S, if S' is NP and $S \leq_p S'$, S' is NP-complete

- Since S is NP-complete, then any NP problem $T \leq_p S$
- Since $S \leq_p S'$, then any problem $T \leq_p S'$.
- S' is NP-complete



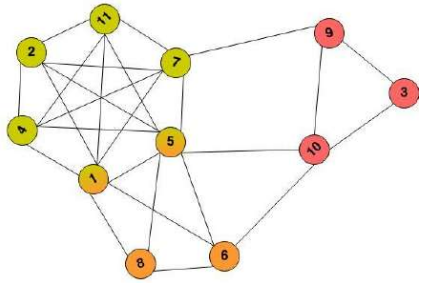
Procedure:

- Prove the problem S' to be NP
- Find an existing NP-complete problem S, so that $S \leq_p S'$

Example: The Clique Problem

Given a graph *G*=(*V*,*E*) and a number *k*, is there a clique of size *k? (supposing |V|=N)*

- *Clique:* a set of vertices where every two are adjacent



Prove 1: Clique problem is NP

Prove 2: SAT is polynomial-time reducible to the Clique problem

- Given an instance of SAT with a Boolean formula ϕ in CNF (Conjunctive Normal Form), ϕ is composed of m clauses $C_1, C_2, \dots, C_m$ over n variables $x_1,\ x_2,\ \dots,\ x_n$
  - Notice: every Boolean formula can be transformed to CNF in polynomial time: $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where $C_i = L_{i1} \vee L_{i2} \vee \cdots L_{ik}$, $L_{ij} \in \{x_1, x_2, \dots x_n, \neg x_1, \neg x_2, \dots \neg x_n\}$

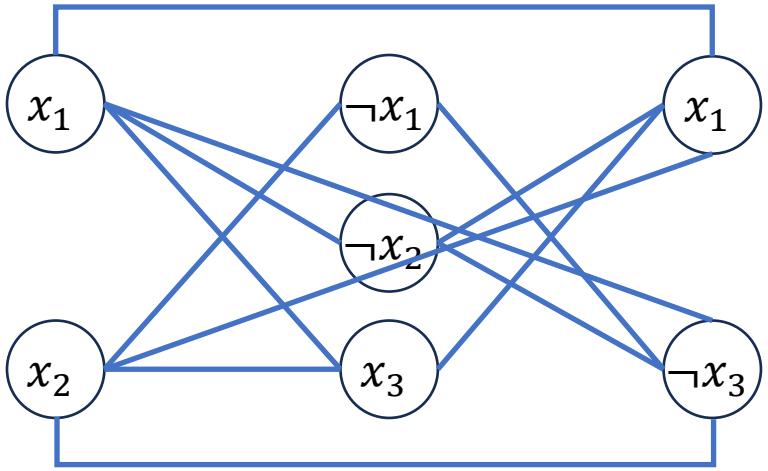
Example: The Clique Problem

$F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where $C_i = L_{i1} \vee L_{i2} \vee \cdots L_{ik}$, $L_{ij} \in \{x_1, x_2, \ldots x_n, \neg x_1, \neg x_2, \ldots \neg x_n\}$

Transformation f: Construct an instance of the Clique problem

- **Vertices**: Create a vertex $v_{i,l}$ for each literal L_{il} in each clause C_i
- **Edges**: Connect two vertices with an edge if they satisfy both the following conditions
  1. Come from different clauses
  2. Not conflicting literals (not x and $\neg x$)
- **Clique size**: m

- Reduction time complexity:
  - At most traverse all the literal pairs: $O(n^2m^2)$

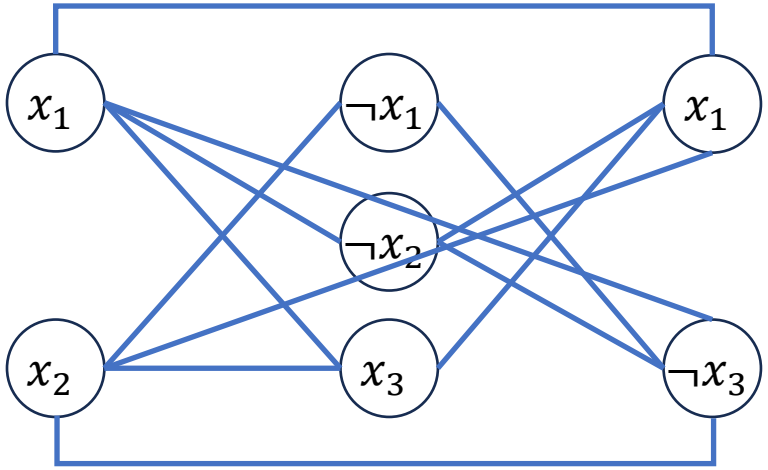


$$E = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$$

Example: The Clique Problem

Proof of correctness (ϕ is yes-instance of SAT iff <G, m> is yes-instance for clique problem)

- **When ϕ is yes-instance of SAT**: There is an assignment that makes ϕ to be true. Choose one literal from each clause that is set to 'true'. Since there are edges between non-conflicting literals from different clauses, ***these literals form a clique with size m***.
- **When <G, m> is yes-instance of clique problem**: given a clique with size m, the verticies are literals from different clauses and there is not conflict. ***Setting them to 'true' makes ϕ true.***



$$E = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$$

Examples for NP-complete problems

Graph Theory Problems:

- **Vertex Cover**: Given a graph G(V,E) and an integer k, is there a vertex cover of size k or less?
- **Independent Set**: Given a graph G(V,E) and an integer k, is there an independent set of size k or more?
- **Graph Coloring**: Given a graph G(V,E) and an integer k, can the vertices of G be colored using k or fewer colors such that no two adjacent vertices share the same color?
- **Hamiltonian Cycle**: Determine if a graph contains a cycle that visits every vertex exactly once

Set and Partition Problems:

- **Subset Sum**: Determine if a set of integers contains a non-empty subset whose sum is zero
- **Partition**: Determine if a given set of integers can be divided into two subsets with equal sum

Examples for NP-complete problems

Network Design:

- **Steiner Tree**: Given a graph G(V,E), a subset of vertices S⊆V, and an integer k, is there a tree of weight k or less that spans S?
- **Traveling Salesman Problem**: Given a weighted graph G(V,E) and an integer k, is there a tour of G with total weight k or less that visits each vertex once and returns to the starting vertex?
- **Others:**
- **Knapsack Problem**: Given a set of items, each with a weight and a value, a knapsack of capacity C, and an integer k, can you select items to place in the knapsack such that the total weight doesn't exceed C and the total value is at least k?
- **Tile Assembly**: Can a set of square tiles be assembled into a particular shape?

NP-hard

A class of problems that are at least as hard as the hardest problems in NP

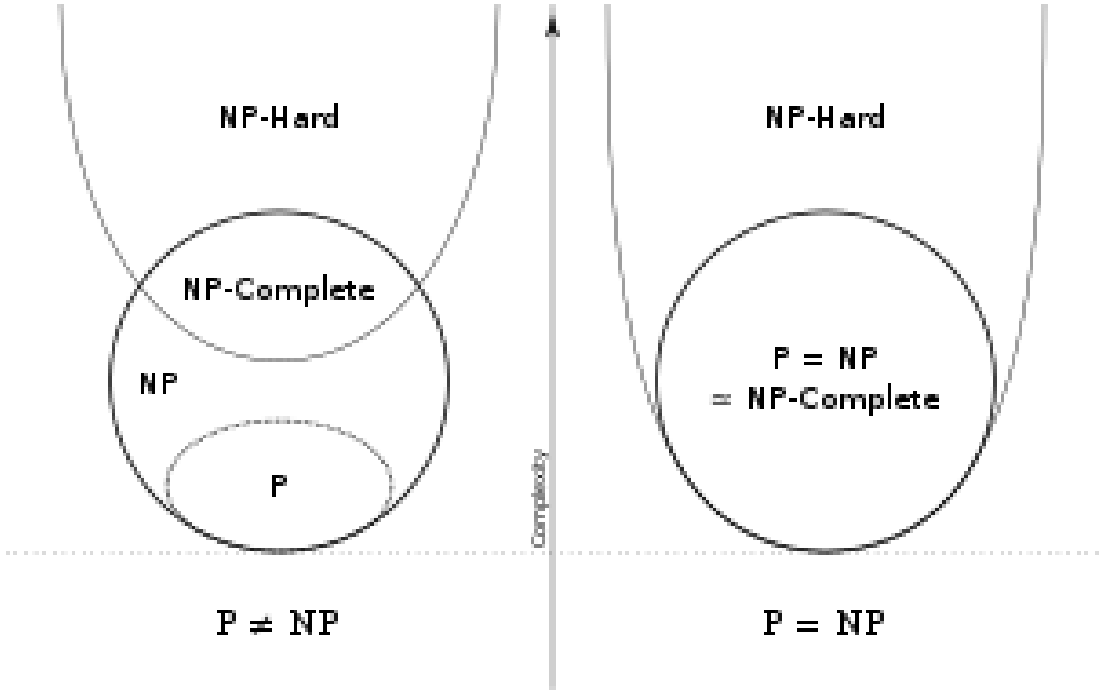
- An NP-complete problem is an NP-hard problem
- An NP-hard problem is not necessarily an NP-complete problem (not necessarily NP, not necessarily decision problems)

Example: Finding a specific value assignment for SAT is an NP-hard problem but not NP-complete problem.

- It is an optimization problem
- Optimization problem is at least as hard as the corresponding decision problem (SAT)

NP-hard

Summary of relationships between the classes.



Assignment

Prove that independence problem is NP-complete

- Given a graph G, an independent set is a set of nodes no pair of which linked by an edge

> INDEPENDENCE
> Input: Graph G and integer k
> Question: Does there exist an independent set with k nodes?

Summary

- Brute Force
- Enumeration
- Recursion and Backtracking
- NP-Completeness

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures