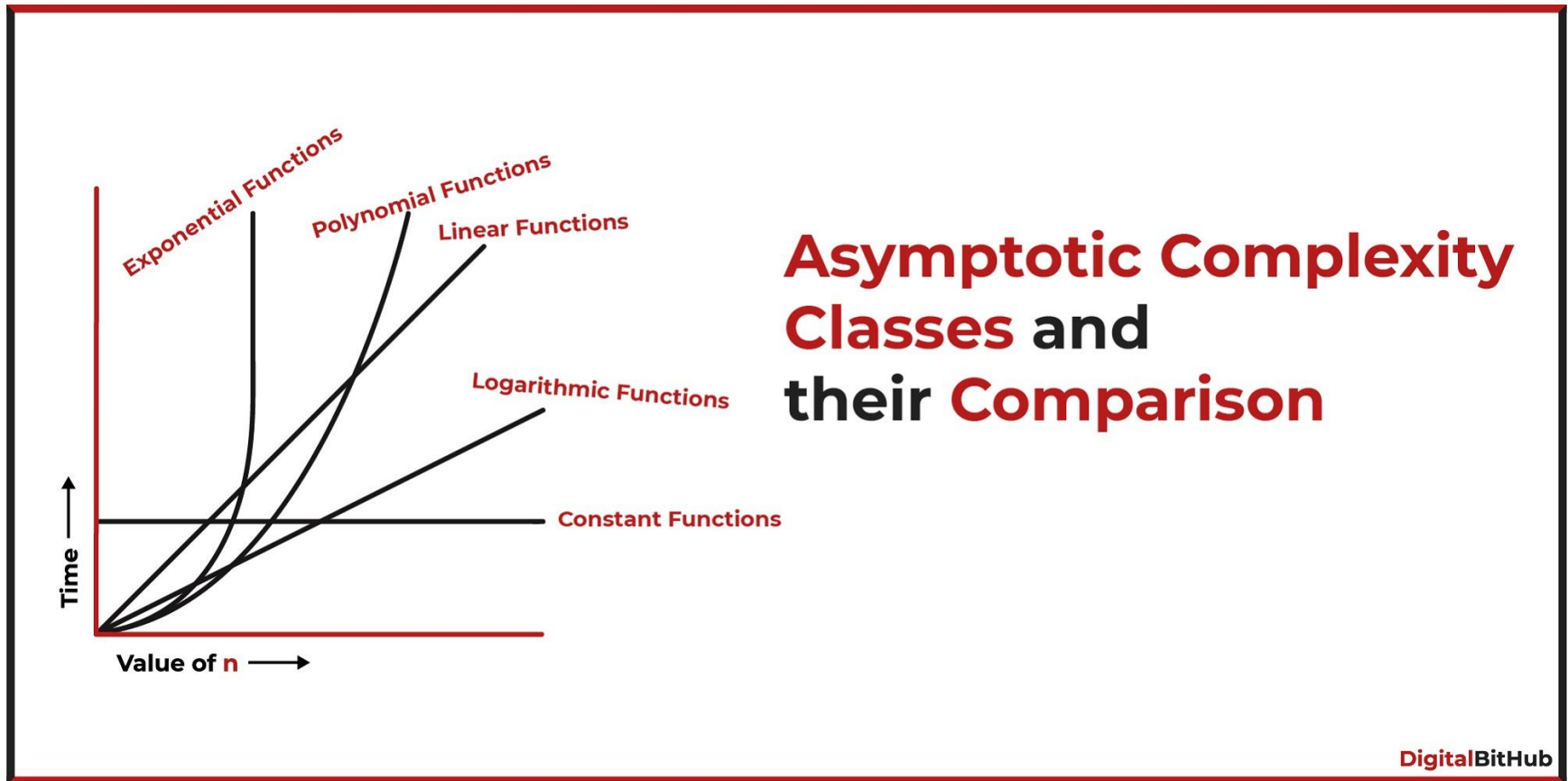


# Complexity classes



# Optimization & Decision Problems

---

- **Decision problems**

- Given an input and a question regarding a problem, determine if the answer is yes or no

- **Optimization problems**

- Find a solution with the “best” value

- Optimization problems can be cast as decision problems that are easier to study

- *E.g.:* Shortest path:  $G$  = unweighted directed graph

- Find a path between  $u$  and  $v$  that uses the fewest edges

- *Does a path exist from  $u$  to  $v$  consisting of at most  $k$  edges?*

# Algorithmic vs Problem Complexity

---

- The ***algorithmic complexity*** of a computation is some measure of how *difficult* is to perform the computation (i.e., specific to an algorithm)
- The **complexity of a computational *problem*** or *task* is the complexity of the algorithm with the **lowest** order of growth of complexity for solving that problem or performing that task.
  - e.g. the problem of searching an ordered list has *at most  $lgn$*  time complexity.
- **Computational Complexity:** deals with classifying problems by how hard they are.

# Class of “P” Problems

---

- **Class P** consists of (decision) problems that are solvable in polynomial time
- Polynomial-time algorithms
  - Worst-case running time is  $O(n^k)$ , for some constant  $k$
- Examples of polynomial time:
  - $O(n^2)$ ,  $O(n^3)$ ,  $O(1)$ ,  $O(n \lg n)$
- Examples of non-polynomial time:
  - $O(2^n)$ ,  $O(n^n)$ ,  $O(n!)$

# *Polynomial and Exponential*

---

Polynomial Time	Exponential Time
Linear Search	0/1 Knapsack
Binary Search	Travelling Salesman Problem
Matrix Multiplication	Hamiltonian Cycle
Sorting	Graph Colouring
Fractional Knapsack	Sum of Subset

# Tractable/Intractable Problems

---

- Problems in P are also called **tractable**
- Problems **not** in P are **intractable or unsolvable**
  - Can be solved in reasonable time only for small inputs
  - Or, can not be solved at all
- Are non-polynomial algorithms always worse than polynomial algorithms?
  - $n^{1,000,000}$  is *technically* tractable, but really impossible
  - $n^{\log \log \log n}$  is *technically* intractable, but easy

# Example of Unsolvable Problem

---

- Turing discovered in the 1930's that there are problems **unsolvable** by *any* algorithm.
- The most famous of them is the ***halting problem***
  - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an “*infinite loop?*”

# Examples of Intractable Problems

---

## Hamiltonian Paths

*Optimization Problem:* Given a graph, find a path that passes through every vertex exactly once

*Decision Problem:* Does a given graph have a Hamiltonian Path ?

## Traveling Salesman

*Optimization Problem:* Find a minimum weight Hamiltonian Path

*Decision Problem:* Given a graph and an integer  $k$ , is there a Hamiltonian Path with a total weight at most  $k$  ?

# Intractable Problems

---

- Can be classified in various categories based on their degree of difficulty, e.g.,
  - NP
  - NP-complete
  - NP-hard
- Let's define NP algorithms and NP problems ...

# Nondeterministic and NP Algorithms

---

**Nondeterministic algorithm** = two stage procedure:

1) Nondeterministic (“guessing”) stage:

generate randomly an arbitrary string that can be thought of as a candidate solution (“certificate”)

2) Deterministic (“verification”) stage:

take the certificate and the instance to the problem and returns YES if the certificate represents a solution

**NP algorithms (Nondeterministic polynomial)**

verification stage is polynomial

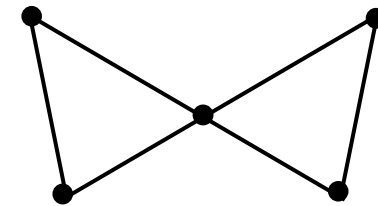
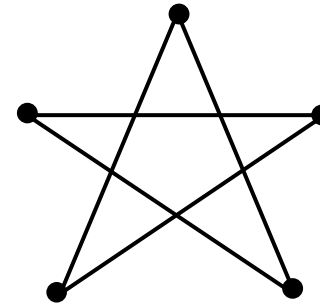
# Class of “NP” Problems

---

- **Class NP** consists of problems that could be solved by NP algorithms
  - i.e., verifiable in polynomial time
- If we were given a “certificate” of a solution, we could verify that the certificate is correct in time polynomial to the size of the input
- Warning: NP does **not** mean “non-polynomial”

# *E.g.:* Hamiltonian Cycle

- **Given:** a directed graph  $G = (V, E)$ , determine a simple cycle that contains each vertex in  $V$ 
  - Each vertex can only be visited once
- **Certificate:**
  - Sequence:  $\langle v_1, v_2, v_3, \dots, v_{|V|} \rangle$

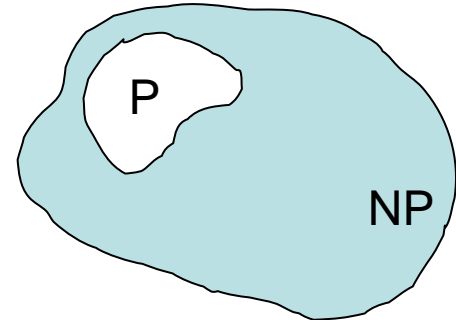


# Is $P = NP$ ?

---

- Any problem in  $P$  is also in  $NP$ :

$$P \subseteq NP$$

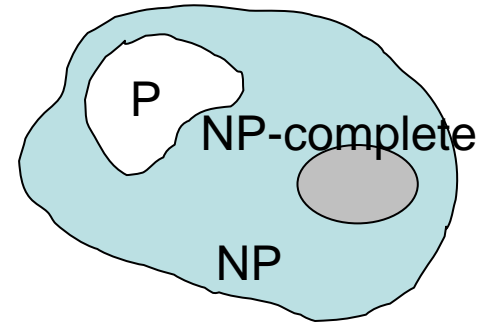


- The big (and **open question**) is whether  $NP \subseteq P$  or  $P = NP$ 
  - i.e., if it is always easy to check a solution, should it also be easy to find a solution?
- Most computer scientists believe that this is false but we do not have a proof ...

# NP-Completeness (informally)

---

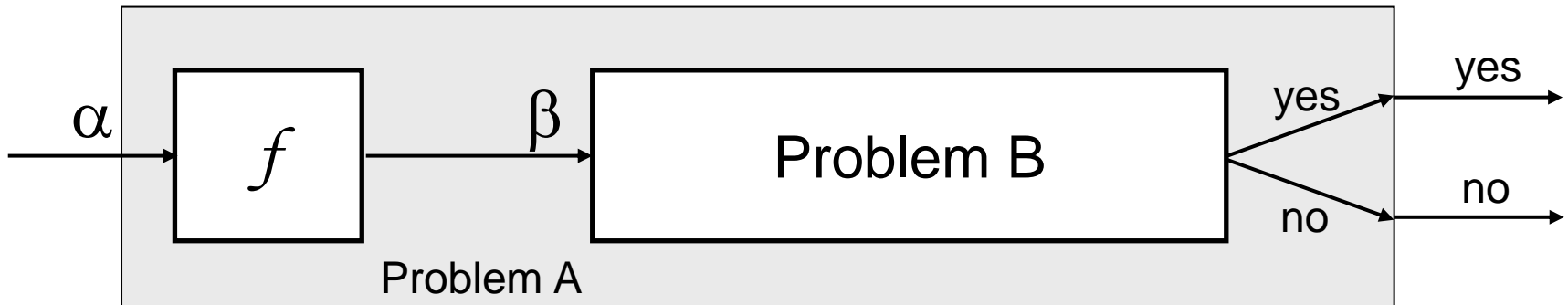
- **NP-complete** problems are defined as the hardest problems in NP



- Most practical problems turn out to be either P or NP-complete.
- Study NP-complete problems ...

# Reductions

- Reduction is a way of saying that one problem is “**easier**” than another.
- We say that problem A is easier than problem B, (i.e., we write “**A ≤ B**”) if we can solve A using the algorithm that solves B.
- **Idea:** transform the inputs of A to inputs of B



# Polynomial Reductions

---

- Given two problems  $A$ ,  $B$ , we say that  $A$  is polynomially **reducible** to  $B$  ( $A \leq_p B$ ) if:
  1. There exists a function  $f$  that converts the input of  $A$  to inputs of  $B$  in polynomial time
  2.  $A(i) = \text{YES} \Leftrightarrow B(f(i)) = \text{YES}$

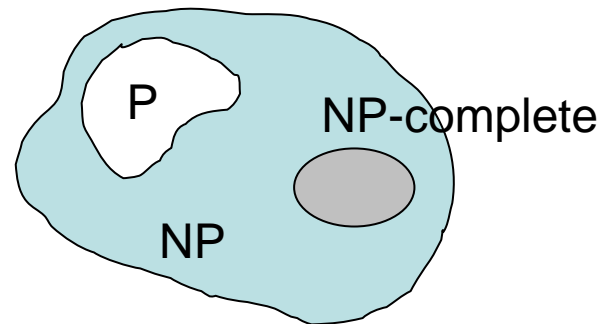
# NP-Completeness (formally)

---

- A problem B is **NP-complete** if:

(1)  $B \in \mathbf{NP}$

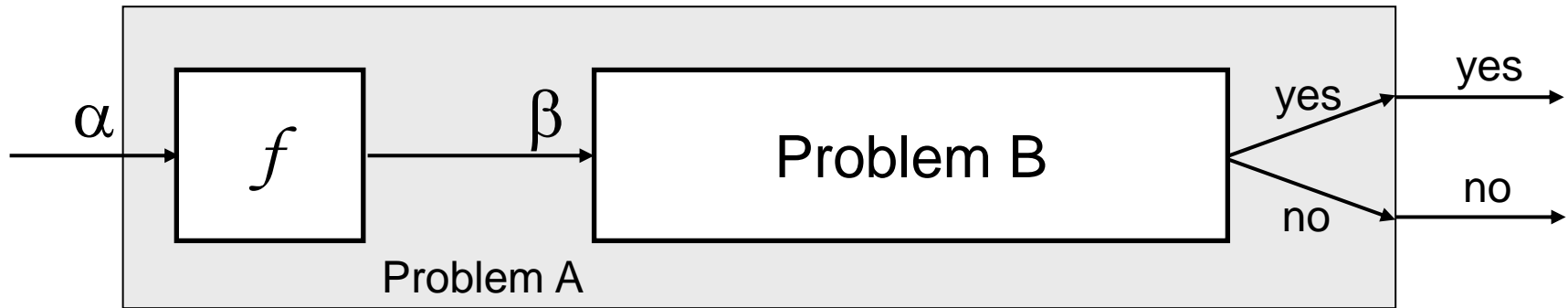
(2)  $A \leq_p B$  for all  $A \in \mathbf{NP}$



- If B satisfies only property (2) we say that B is **NP-hard**
- No polynomial time algorithm has been discovered for an **NP-Complete** problem
- No one has ever proven that no polynomial time algorithm can exist for any **NP-Complete** problem

# Implications of Reduction

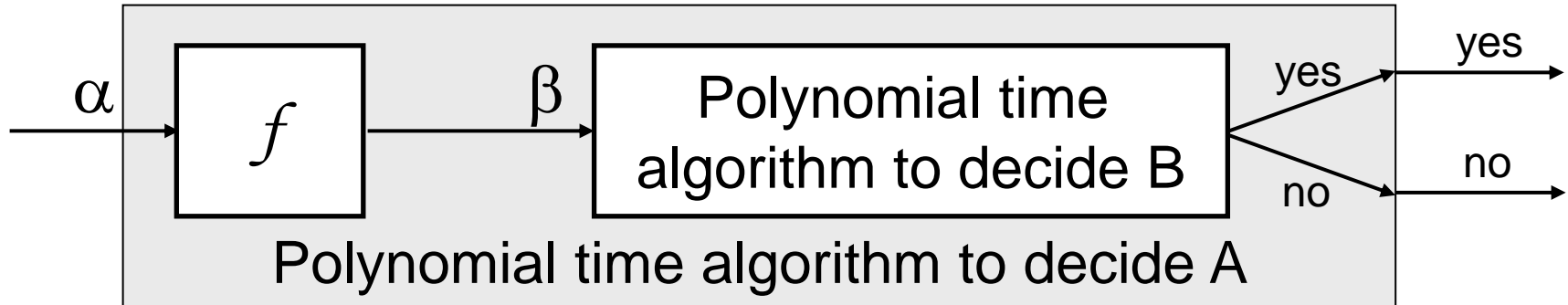
---



- If  $A \leq_p B$  and  $B \in P$ , then  $A \in P$
- if  $A \leq_p B$  and  $A \notin P$ , then  $B \notin P$

# Proving Polynomial Time

---



1. Use a **polynomial time** reduction algorithm to transform A into B
2. Run a known **polynomial time** algorithm for B
3. Use the answer for B as the answer for A

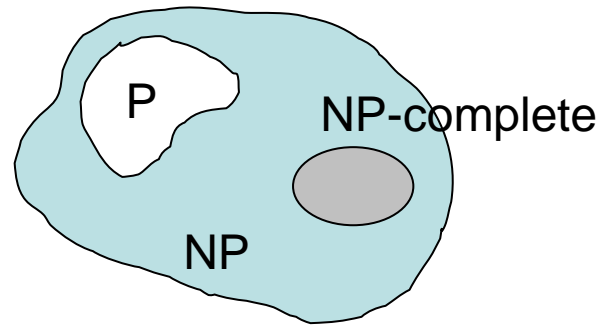
# Proving NP-Completeness In Practice

---

- Prove that the problem B is in NP
  - A randomly generated string can be checked in polynomial time to determine if it represents a solution
- Show that **one known NP-Complete** problem can be transformed to B in polynomial time
  - No need to check that **all NP-Complete** problems are reducible to B

# Revisit “Is $P = NP$ ?”

---



*Theorem:* If any NP-Complete problem can be solved in polynomial time  $\Rightarrow$  then  $P = NP$ .

# P & NP-Complete Problems

---

- **Shortest simple path**

- Given a graph  $G = (V, E)$  find a **shortest** path from a source to all other vertices
- Polynomial solution:  $O(VE)$

- **Longest simple path**

- Given a graph  $G = (V, E)$  find a **longest** path from a source to all other vertices
- NP-complete

# P & NP-Complete Problems

---

- **Euler tour**

- $G = (V, E)$  a connected, directed graph find a cycle that traverses each edge of  $G$  exactly once (may visit a vertex multiple times)
- Polynomial solution  $O(E)$

- **Hamiltonian cycle**

- $G = (V, E)$  a connected, directed graph find a cycle that visits each vertex of  $G$  exactly once
- NP-complete

# Satisfiability Problem (SAT)

---

- **Satisfiability problem:** given a logical expression  $\Phi$ , find an assignment of values (F, T) to variables  $x_i$  that causes  $\Phi$  to evaluate to T

$$\Phi = x_1 \vee \neg x_2 \wedge x_3 \vee \neg x_4$$

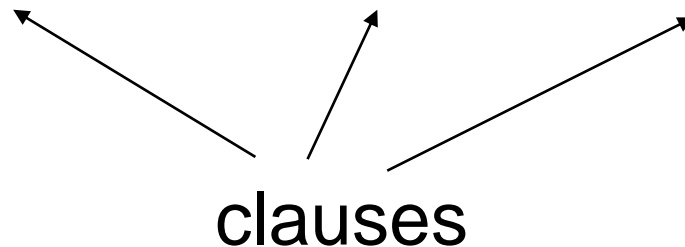
- SAT was the first problem shown to be NP-complete!

# CFN Satisfiability

---

- CFN is a special case of SAT
- $\Phi$  is in “Conjunctive Normal Form” (CNF)
  - “AND” of expressions (i.e., clauses)
  - Each clause contains only “OR”s of the variables and their complements

*E.g.:*  $\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$



# 3-CNF Satisfiability

---

## A subcase of CNF problem:

– Contains three clauses

• *E.g.:*

$$\Phi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

• **3-CNF** is NP-Complete

• Interestingly enough, **2-CNF** is in P!

# Clique

## Clique Problem:

- Undirected graph  $G = (V, E)$
- **Clique:** a subset of vertices in  $V$  all connected to each other by edges in  $E$  (i.e., forming a complete graph)
- **Size of a clique:** number of vertices it contains

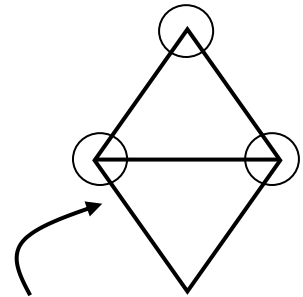
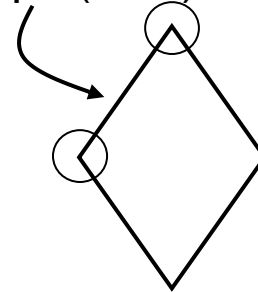
## Optimization problem:

- Find a clique of maximum size

## Decision problem:

- Does  $G$  have a clique of size  $k$ ?

Clique( $G, 2$ ) = YES  
Clique( $G, 3$ ) = NO

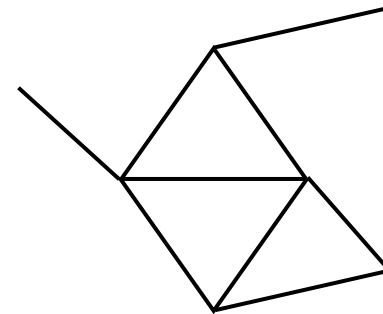


Clique( $G, 3$ ) = YES  
Clique( $G, 4$ ) = NO

# Clique Verifier

---

- **Given:** an undirected graph  $G = (V, E)$
- **Problem:** Does  $G$  have a clique of size  $k$ ?
- **Certificate:**
  - A set of  $k$  nodes
- **Verifier:**
  - Verify that for all pairs of vertices in this set there exists an edge in  $E$



# 3-CNF $\leq_p$ Clique

---

- **Idea:**

- Construct a graph  $G$  such that  $\Phi$  is satisfiable only if  $G$  has a clique of size  $k$

# NP-naming convention

---

- **NP-complete** - means problems that are 'complete' in NP, i.e. the most difficult to solve in NP
- **NP-hard** - stands for 'at least' as hard as NP (but not necessarily **in** NP);
- **NP-easy** - stands for 'at most' as hard as NP (but not necessarily **in** NP);
- **NP-equivalent** - means equally difficult as NP, (but not necessarily **in** NP);

# Examples NP-complete and NP-hard problems

---

## Hamiltonian Paths

NP-complete

*Optimization Problem:* Given a graph, find a path that passes through every vertex exactly once

*Decision Problem:* Does a given graph have a Hamiltonian Path ?

## Traveling Salesman

NP-hard

*Optimization Problem:* Find a minimum weight Hamiltonian Path

*Decision Problem:* Given a graph and an integer  $k$ , is there a Hamiltonian Path with a total weight at most  $k$  ?

# Difference

NP-hard	NP-Complete
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time	NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, it do not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Not all NP-hard problems are NP-complete.	All NP-complete problems are NP-hard
Example: Halting problem, Vertex cover problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, Circuit-satisfiability problem, etc.

# Examples NP-complete

---

[https://www.wikiwand.com/en/List\\_of\\_NP-complete\\_problems](https://www.wikiwand.com/en/List_of_NP-complete_problems)