Teoria degli Algoritmi

Corso di Laurea Magistrale in Matematica Applicata
a.a. 2020-21

Gabriele Tolomei

Dipartimento di Informatica
Sapienza Università di Roma
tolomei@di.uniroma1.it

Lecture 6: Complexity Classes

Table of Contents

Table of Contents

Complexity Relationships

- The last two theorems illustrate an important distinction

Complexity Relationships

- The last two theorems illustrate an important distinction
- On the one hand, we proved that at most a square (i.e., **polynomial**) difference between the time complexity of problems decided by single- vs. multi-tape TMs

Complexity Relationships

- The last two theorems illustrate an important distinction
- On the one hand, we proved that at most a square (i.e., **polynomial**) difference between the time complexity of problems decided by single- vs. multi-tape TMs
- On the other hand, we showed that at most an **exponential** difference exists between the time complexity of problems decided by deterministic vs. non-deterministic TMs

Complexity Relationships

- The last two theorems illustrate an important distinction
- On the one hand, we proved that at most a square (i.e., **polynomial**) difference between the time complexity of problems decided by single- vs. multi-tape TMs
- On the other hand, we showed that at most an **exponential** difference exists between the time complexity of problems decided by deterministic vs. non-deterministic TMs
- Let's now try to classify (decision) problems on top of this distinction

Table of Contents

Polynomial Time

- Polynomial differences in running time are considered to be "small", whereas exponential ones are deemed to be "large"

Polynomial Time

- Polynomial differences in running time are considered to be "small", whereas exponential ones are deemed to be "large"

Example

Suppose we have given with two algorithms, i.e., one whose running time is n^3 and the other is 2^n.

Polynomial Time

- Polynomial differences in running time are considered to be "small", whereas exponential ones are deemed to be "large"

Example

Suppose we have given with two algorithms, i.e., one whose running time is n^3 and the other is 2^n.
Let's see how running time actually grows using the two algorithms above on different input size (i.e., for several values of n).

Polynomial Time

- Polynomial differences in running time are considered to be "small", whereas exponential ones are deemed to be "large"

Example

Suppose we have given with two algorithms, i.e., one whose running time is n^3 and the other is 2^n.
Let's see how running time actually grows using the two algorithms above on different input size (i.e., for several values of n).

- $n = 10$; $n^3 = 1,000; 2^n = 1,024$ (for small n, differences are small as well)

SAPIENZA UNIVERSITÀ DI ROMA

Polynomial Time

- Polynomial differences in running time are considered to be "small", whereas exponential ones are deemed to be "large"

Example

Suppose we have given with two algorithms, i.e., one whose running time is n^3 and the other is 2^n.
Let's see how running time actually grows using the two algorithms above on different input size (i.e., for several values of n).

- $n = 10$; $n^3 = 1,000; 2^n = 1,024$ (for small n, differences are small as well)
- $n = 100$; $n^3 = 1,000,000; 2^n \approx 1.26 \times 10^{30}$ (for larger n, huge difference!)

Polynomial Time

- Polynomial differences in running time are considered to be "small", whereas exponential ones are deemed to be "large"

Example

Suppose we have given with two algorithms, i.e., one whose running time is n^3 and the other is 2^n.
Let's see how running time actually grows using the two algorithms above on different input size (i.e., for several values of n).

- $n = 10$; $n^3 = 1,000; 2^n = 1,024$ (for small n, differences are small as well)
- $n = 100$; $n^3 = 1,000,000; 2^n \approx 1.26 \times 10^{30}$ (for larger n, huge difference!)

- Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms are rarely useful in practice

Brute-Force Search

- Exponential time algorithms typically arise when we solve problems by **exhaustively searching** through the whole space of solutions

Brute-Force Search

- Exponential time algorithms typically arise when we solve problems by **exhaustively searching** through the whole space of solutions
- This technique is known as **brute-force search**

Brute-Force Search

- Exponential time algorithms typically arise when we solve problems by **exhaustively searching** through the whole space of solutions
- This technique is known as **brute-force search**

Example

An example of brute-force search algorithm is when we try to factor a number into its constituent primes by searching through all potential divisors.

Brute-Force Search

- Exponential time algorithms typically arise when we solve problems by **exhaustively searching** through the whole space of solutions
- This technique is known as **brute-force search**

Example

An example of brute-force search algorithm is when we try to factor a number into its constituent primes by searching through all potential divisors.
Given a number x such that $n = \log x$ (bits), a trivial algorithm to find all prime factors of x would require to loop from 2 to $x - 1$ (or, more cleverly, up to $\sqrt{x}$).

Brute-Force Search

- Exponential time algorithms typically arise when we solve problems by **exhaustively searching** through the whole space of solutions
- This technique is known as **brute-force search**

Example

An example of brute-force search algorithm is when we try to factor a number into its constituent primes by searching through all potential divisors.
Given a number x such that $n = \log x$ (bits), a trivial algorithm to find all prime factors of x would require to loop from 2 to $x - 1$ (or, more cleverly, up to $\sqrt{x}$).
Anyway, it requires a number of $O(x) = 2^{O(n)}$ iterations!

Polynomial Equivalence

- All reasonable deterministic computational models are **polynomially equivalent**

Polynomial Equivalence

- All reasonable deterministic computational models are **polynomially equivalent**
- That is, any one of them can simulate another with at most a polynomial increase in running time

Polynomial Equivalence

- All reasonable deterministic computational models are **polynomially equivalent**
- That is, any one of them can simulate another with at most a polynomial increase in running time
- When we say *reasonable* deterministic models, we are not actually defining the term “reasonable”

Polynomial Equivalence

- All reasonable deterministic computational models are **polynomially equivalent**
- That is, any one of them can simulate another with at most a polynomial increase in running time
- When we say *reasonable* deterministic models, we are not actually defining the term "reasonable"
- Indeed, we refer to a broader notion that includes models that closely approximate running times on actual computers

Polynomial Equivalence

- All reasonable deterministic computational models are **polynomially equivalent**
- That is, any one of them can simulate another with at most a polynomial increase in running time
- When we say *reasonable* deterministic models, we are not actually defining the term "reasonable"
- Indeed, we refer to a broader notion that includes models that closely approximate running times on actual computers
- For example, we showed that single-tape and multi-tape TMs are polynomially equivalent

Polynomial Equivalence

- From here on we focus on aspects of time complexity theory that are not affected by polynomial time differences

Polynomial Equivalence

- From here on we focus on aspects of time complexity theory that are not affected by polynomial time differences
- In other words, we consider polynomial time differences negligible

Polynomial Equivalence

- From here on we focus on aspects of time complexity theory that are not affected by polynomial time differences
- In other words, we consider polynomial time differences negligible
- By doing so we can develop the theory in a way that is independent on the specific model of computation

Polynomial Equivalence

- From here on we focus on aspects of time complexity theory that are not affected by polynomial time differences
- In other words, we consider polynomial time differences negligible
- By doing so we can develop the theory in a way that is independent on the specific model of computation

Note

Our goal is to present the fundamental properties of **computation**, rather than properties of Turing machines or any other specific model

Disregarding Polynomial Differences

- Disregarding polynomial differences in running time may sound odd

Disregarding Polynomial Differences

- Disregarding polynomial differences in running time may sound odd
- After all, real programmers spend a lot of time trying to speedup their programs by that polynomial factors (or even less, for that matters!)

Disregarding Polynomial Differences

- Disregarding polynomial differences in running time may sound odd
- After all, real programmers spend a lot of time trying to speedup their programs by that polynomial factors (or even less, for that matters!)
- We already disregarded constant factors when we introduced asymptotic notation

Disregarding Polynomial Differences

- Disregarding polynomial differences in running time may sound odd
- After all, real programmers spend a lot of time trying to speedup their programs by that polynomial factors (or even less, for that matters!)
- We already disregarded constant factors when we introduced asymptotic notation
- Now, we propose to disregard the much grater polynomial differences, e.g., between n and n^3

Disregarding Polynomial Differences

- That does **not** mean we are considering those differences unimportant!

Disregarding Polynomial Differences

- That does **not** mean we are considering those differences unimportant!
- On the contrary, the difference between an algorithm whose running time is n and another one n^3 is significant

Disregarding Polynomial Differences

- That does **not** mean we are considering those differences unimportant!
- On the contrary, the difference between an algorithm whose running time is n and another one n^3 is significant
- However, "core" questions like the polynomial vs. non-polynomial solvability of a problem is somewhat more important

The Class P: Definition

Definition (The Class P)

P is the class of languages that are decidable in polynomial time by a deterministic single-tape Turing machine. More formally:

$$P = \bigcup_k TIME(n^k)$$

The Class P

The class P plays a central role in computational complexity theory:

The Class P

The class P plays a central role in computational complexity theory:

1. P is **invariant** for all models of computation that are polynomially-equivalent to a single-tape TM

The Class P

The class P plays a central role in computational complexity theory:

1. P is **invariant** for all models of computation that are polynomially-equivalent to a single-tape TM
2. P roughly corresponds to the class of problems that are realistically solvable on a computer (i.e., "easy to solve")

The Class P

The class P plays a central role in computational complexity theory:

1. P is **invariant** for all models of computation that are polynomially-equivalent to a single-tape TM
2. P roughly corresponds to the class of problems that are realistically solvable on a computer (i.e., "easy to solve")

- Item 1. indicates that P is mathematically robust, as it is not affected by specific peculiarities or nuances of the model of computation used

The Class P

The class P plays a central role in computational complexity theory:

1. P is **invariant** for all models of computation that are polynomially-equivalent to a single-tape TM
2. P roughly corresponds to the class of problems that are realistically solvable on a computer (i.e., "easy to solve")

- Item 1. indicates that P is mathematically robust, as it is not affected by specific peculiarities or nuances of the model of computation used
- Item 2. says that P is relevant from a practical perspective

The Class P

- When a problem is known to be in P, we have an algorithm that solves it running in time n^k (for some constant k)

The Class P

- When a problem is known to be in P, we have an algorithm that solves it running in time n^k (for some constant k)
- Of course, whether this running time is practical or not depends on the constant k

The Class P

- When a problem is known to be in P, we have an algorithm that solves it running in time n^k (for some constant k)
- Of course, whether this running time is practical or not depends on the constant k
- For example, a running time of n^{100} is rare to be of any practical use!

The Class P

- When a problem is known to be in P, we have an algorithm that solves it running in time n^k (for some constant k)
- Of course, whether this running time is practical or not depends on the constant k
- For example, a running time of n^{100} is rare to be of any practical use!
- Still, setting the "threshold of practical solvability" to the class of polynomials has proven useful

The Class P

- When a problem is known to be in P, we have an algorithm that solves it running in time n^k (for some constant k)
- Of course, whether this running time is practical or not depends on the constant k
- For example, a running time of n^{100} is rare to be of any practical use!
- Still, setting the "threshold of practical solvability" to the class of polynomials has proven useful
- Once a polynomial time algorithm is found for a problem that formerly appeared to be solvable only in exponential time, we can get insights on the complexity of other problems as well (through reductions...)

Examples of Problems in P

- When we present a polynomial time algorithm, we give a high-level description of it without any reference to the model of computation

Examples of Problems in P

- When we present a polynomial time algorithm, we give a high-level description of it without any reference to the model of computation
- In this way, we can focus on the important aspects and disregard tedious details like head movements or tape contents

Examples of Problems in P

- When we present a polynomial time algorithm, we give a high-level description of it without any reference to the model of computation
- In this way, we can focus on the important aspects and disregard tedious details like head movements or tape contents
- To analyze the polynomiality of an algorithm we describe it in terms of **number of stages**

Examples of Problems in P

- When we present a polynomial time algorithm, we give a high-level description of it without any reference to the model of computation
- In this way, we can focus on the important aspects and disregard tedious details like head movements or tape contents
- To analyze the polynomiality of an algorithm we describe it in terms of **number of stages**
- The notion of **stage** of an algorithm is similar to that of a **step** of a TM

Examples of Problems in P

- When we present a polynomial time algorithm, we give a high-level description of it without any reference to the model of computation
- In this way, we can focus on the important aspects and disregard tedious details like head movements or tape contents
- To analyze the polynomiality of an algorithm we describe it in terms of **number of stages**
- The notion of **stage** of an algorithm is similar to that of a **step** of a TM
- In general, though, implementing one stage of an algorithm will require many steps of a TM

Examples of Problems in P

Analyzing an algorithm to show that it runs in polynomial time requires to do 2 things

Examples of Problems in P

Analyzing an algorithm to show that it runs in polynomial time requires to do 2 things

1. We have to give a polynomial upper bound (using big-O) on the number of stages required by the algorithm to run on an input of length n

Examples of Problems in P

Analyzing an algorithm to show that it runs in polynomial time requires to do 2 things

1. We have to give a polynomial upper bound (using big-O) on the number of stages required by the algorithm to run on an input of length n
2. We have to make sure that individual stages of the algorithm can be implemented in polynomial time on a reasonable deterministic model

Examples of Problems in P

Analyzing an algorithm to show that it runs in polynomial time requires to do 2 things

1. We have to give a polynomial upper bound (using big-O) on the number of stages required by the algorithm to run on an input of length n
2. We have to make sure that individual stages of the algorithm can be implemented in polynomial time on a reasonable deterministic model

When both tasks are done, we can conclude that the algorithm runs in polynomial time because it runs for a polynomial number of stages, each one running in polynomial time

Examples of Problems in P: Encoding

- One point that requires attention is the encoding method used

Examples of Problems in P: Encoding

- One point that requires attention is the encoding method used
- We stick to the usual notation $\langle \cdot \rangle$ to denote a reasonable encoding of one or more objects into a string

Examples of Problems in P: Encoding

- One point that requires attention is the encoding method used
- We stick to the usual notation $\langle \cdot \rangle$ to denote a reasonable encoding of one or more objects into a string
- A reasonable encoding is one that allows for polynomial time encoding and decoding of objects into natural internal representations

Examples of Problems in P: Graph Encoding

- Many computational problems operate on encoding of graphs

Examples of Problems in P: Graph Encoding

- Many computational problems operate on encoding of graphs
- There are two reasonable ways of encoding a graph: **adjacency list** and **adjacency matrix**

Examples of Problems in P: Graph Encoding

- Many computational problems operate on encoding of graphs
- There are two reasonable ways of encoding a graph: **adjacency list** and **adjacency matrix**
- The former is a list of nodes along with the list of edges

Examples of Problems in P: Graph Encoding

- Many computational problems operate on encoding of graphs
- There are two reasonable ways of encoding a graph: **adjacency list** and **adjacency matrix**
- The former is a list of nodes along with the list of edges
- The latter uses a matrix, where the entry $(i, j) = 1$ if there is an edge connecting node i with node j, or $(i, j) = 0$ otherwise

Examples of Problems in P: Graph Encoding

- When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation

Examples of Problems in P: Graph Encoding

- When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation
- This is because the size of reasonable graph representations is a polynomial in the number of nodes

Examples of Problems in P: Graph Encoding

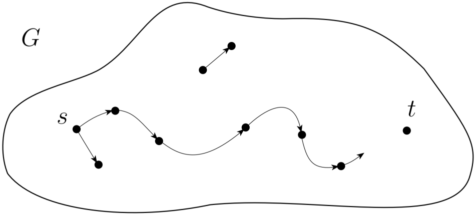
- When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation
- This is because the size of reasonable graph representations is a polynomial in the number of nodes
- Thus, if we find that the running time of an algorithm on a graph is polynomial (exponential) in the number of nodes, we know that it is also polynomial (exponential) in the size of the graph representation

Examples of Problems in P: $PATH$

Definition (The $PATH$ problem)

Let $G = (V, E)$ be a **directed graph** containing two nodes $s, t \in V$ as shown below.
The $PATH$ problem is to determine whether a directed path exists from s to t:

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is s directed graph with a directed path from } s \text{ to } t\}$$



Examples of Problems in P: $PATH$

Theorem ($PATH \in P$)

The PATH problem is in the class P

Examples of Problems in P: *PATH*

Theorem ($PATH \in P$)

The PATH problem is in the class P

Sketch.

We must design a polynomial time algorithm that decides *PATH*.

Before sketching the proof, let's see that:

- a **brute-force** solution exists (i.e., *PATH* is actually decidable);

Examples of Problems in P: $PATH$

Theorem ($PATH \in P$)

The PATH problem is in the class P

Sketch.

We must design a polynomial time algorithm that decides $PATH$.
Before sketching the proof, let's see that:

- a **brute-force** solution exists (i.e., $PATH$ is actually decidable);
- the brute-force solution is not fast enough.

□

PATH: Brute-Force Solution

- A brute-force solution for *PATH* examines **all possible** paths in G and checks whether there exists at least one from s to t or not

PATH: Brute-Force Solution

- A brute-force solution for *PATH* examines **all possible** paths in G and checks whether there exists at least one from s to t or not
- *PATH* is indeed decidable! But what is the running time of the trivial brute-force solution?

PATH: Brute-Force Solution

- A brute-force solution for *PATH* examines **all possible** paths in G and checks whether there exists at least one from s to t or not
- *PATH* is indeed decidable! But what is the running time of the trivial brute-force solution?
- A path in G is a sequence of nodes whose length is at most the total number of nodes in G, i.e., $n = |V|$

PATH: Brute-Force Solution

- A brute-force solution for *PATH* examines **all possible** paths in G and checks whether there exists at least one from s to t or not
- *PATH* is indeed decidable! But what is the running time of the trivial brute-force solution?
- A path in G is a sequence of nodes whose length is at most the total number of nodes in G, i.e., $n = |V|$
- This would happen if the path from s to t touches every node in G, as there is no point of passing through the same node more than once

PATH: Brute-Force Solution

- A brute-force solution for *PATH* examines **all possible** paths in G and checks whether there exists at least one from s to t or not
- *PATH* is indeed decidable! But what is the running time of the trivial brute-force solution?
- A path in G is a sequence of nodes whose length is at most the total number of nodes in G, i.e., $n = |V|$
- This would happen if the path from s to t touches every node in G, as there is no point of passing through the same node more than once
- The number of paths is, roughly, n^n, which is exponential in the number of nodes (i.e., each node is connected to any other node)

PATH: Brute-Force Solution

- A brute-force solution for *PATH* examines **all possible** paths in G and checks whether there exists at least one from s to t or not
- *PATH* is indeed decidable! But what is the running time of the trivial brute-force solution?
- A path in G is a sequence of nodes whose length is at most the total number of nodes in G, i.e., $n = |V|$
- This would happen if the path from s to t touches every node in G, as there is no point of passing through the same node more than once
- The number of paths is, roughly, n^n, which is exponential in the number of nodes (i.e., each node is connected to any other node)
- The brute-force solution has **exponential** running time

PATH: Polynomial-Time Solution

- To get a polynomial time solution for *PATH* we must avoid doing any brute-force

PATH: Polynomial-Time Solution

- To get a polynomial time solution for *PATH* we must avoid doing any brute-force
- One way to do so is to apply a well-known graph-searching technique like **breadth-first search**

PATH: Polynomial-Time Solution

- To get a polynomial time solution for *PATH* we must avoid doing any brute-force
- One way to do so is to apply a well-known graph-searching technique like **breadth-first search**
- Here, we successively mark all nodes in G that have been visited so far starting from s and going to the nodes directly reachable from it

PATH: Polynomial-Time Solution

- To get a polynomial time solution for *PATH* we must avoid doing any brute-force
- One way to do so is to apply a well-known graph-searching technique like **breadth-first search**
- Here, we successively mark all nodes in G that have been visited so far starting from s and going to the nodes directly reachable from it
- We can do so for length-1, length-2, up to length-n paths

PATH: Polynomial-Time Solution

A polynomial time algorithm for *PATH*.

$M =$ "On input $\langle G, s, t \rangle$:

1. Mark node s as **visited**;

PATH: Polynomial-Time Solution

A polynomial time algorithm for *PATH*.

$M =$ "On input $\langle G, s, t \rangle$:

1. Mark node s as **visited**;
2. Repeat the following until no additional nodes can be marked (i.e., visited):

PATH: Polynomial-Time Solution

A polynomial time algorithm for *PATH*.

$M =$ "On input $\langle G, s, t \rangle$:

1. Mark node s as **visited**;
2. Repeat the following until no additional nodes can be marked (i.e., visited):
   a. Scan all the edges of G, and if an edge (i, j) is found going from a marked node i to an unmarked node j, mark node j;

PATH: Polynomial-Time Solution

A polynomial time algorithm for *PATH*.

$M =$ "On input $\langle G, s, t \rangle$:

1. Mark node s as **visited**;
2. Repeat the following until no additional nodes can be marked (i.e., visited):
   a. Scan all the edges of G, and if an edge (i, j) is found going from a marked node i to an unmarked node j, mark node j;
3. If t is marked, **accept**; otherwise **reject**."

$\square$

$PATH$: Polynomial-Time Solution

- Let's analyze the algorithm M above

PATH: Polynomial-Time Solution

- Let's analyze the algorithm M above
- Obviously, stage 1 and 3 are executed only once

PATH: Polynomial-Time Solution

- Let's analyze the algorithm M above
- Obviously, stage 1 and 3 are executed only once
- Stage 2.a runs at most n times because at each iteration (except for the last one) at most one node is marked as visited

PATH: Polynomial-Time Solution

- Let's analyze the algorithm M above
- Obviously, stage 1 and 3 are executed only once
- Stage $2.a$ runs at most n times because at each iteration (except for the last one) at most one node is marked as visited
- Thus, the total number of stages used by M is at most $1 + 1 + n$, namely polynomial in the number of nodes of G and therefore on its size

PATH: Polynomial-Time Solution

- Let's analyze the algorithm M above
- Obviously, stage 1 and 3 are executed only once
- Stage 2.a runs at most n times because at each iteration (except for the last one) at most one node is marked as visited
- Thus, the total number of stages used by M is at most $1 + 1 + n$, namely polynomial in the number of nodes of G and therefore on its size

Note

Stages 1 and 3 are easily implemented in polynomial time on any reasonable deterministic model. Stage 2.a involves scanning the input and test for marked nodes, which again can be implemented in polynomial time. Hence M is a polynomial time algorithm for *PATH*

Table of Contents

The Class NP

- As we observed in the case of $PATH$, we can sometimes avoid brute-force search to obtain polynomial time solutions

The Class NP

- As we observed in the case of *PATH*, we can sometimes avoid brute-force search to obtain polynomial time solutions
- However, for some other (interesting) problems, any attempt to avoid brute-force solutions has been unsuccessful

The Class NP

- As we observed in the case of *PATH*, we can sometimes avoid brute-force search to obtain polynomial time solutions
- However, for some other (interesting) problems, any attempt to avoid brute-force solutions has been unsuccessful
- For those problems, no polynomial time algorithms have been found (so far)

The Class NP

- We don't know *exactly* why for some problems we were unsuccessful to find any polynomial time algorithm

The Class NP

- We don't know *exactly* why for some problems we were unsuccessful to find any polynomial time algorithm
- Perhaps, these problems have in fact polynomial time algorithms that solve them, yet they are still unknown

The Class NP

- We don't know *exactly* why for some problems we were unsuccessful to find any polynomial time algorithm
- Perhaps, these problems have in fact polynomial time algorithms that solve them, yet they are still unknown
- Or, maybe, some of these problems simply cannot be solved in polynomial time as they are *intrinsically* difficult

The Class NP

- A remarkable result, though, shows that the complexity of many problems are linked together

The Class NP

- A remarkable result, though, shows that the complexity of many problems are linked together
- A polynomial time algorithm for one such problem can be used to solve an entire class of problems

The Class NP

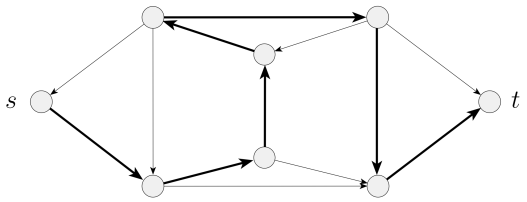
- A remarkable result, though, shows that the complexity of many problems are linked together
- A polynomial time algorithm for one such problem can be used to solve an entire class of problems
- Let's see this through an example, called *HAMPATH*

The *HAMPATH* Problem

Definition (*HAMPATH*)

Let $G = (V, E)$ be a **directed graph**.
We define a so-called **Hamiltonian path** a directed path that goes through each and every node of G **exactly once**.
The *HAMPATH* problem asks to find whether G contains a Hamiltonian path connecting two specific nodes, s and t, as shown below.

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ contains a Hamiltonian path from } s \text{ to } t\}$$



The *HAMPATH* Problem

- We can easily obtain an exponential time algorithm for the *HAMPATH* problem

The *HAMPATH* Problem

- We can easily obtain an exponential time algorithm for the *HAMPATH* problem
- This is just a slight modification of the brute-force algorithm given for *PATH*

The *HAMPATH* Problem

- We can easily obtain an exponential time algorithm for the *HAMPATH* problem
- This is just a slight modification of the brute-force algorithm given for *PATH*
- We enumerate all directed paths of G, check if there exists a path from s to t, and - if it does - test if this is a Hamiltonian path

The *HAMPATH* Problem

- We can easily obtain an exponential time algorithm for the *HAMPATH* problem
- This is just a slight modification of the brute-force algorithm given for *PATH*
- We enumerate all directed paths of G, check if there exists a path from s to t, and - if it does - test if this is a Hamiltonian path
- No one knows whether *HAMPATH* is solvable in polynomial time

The *HAMPATH* Problem: Polynomial Verifiability

- The *HAMPATH* problem has, though, a property called **polynomial verifiability**

The *HAMPATH* Problem: Polynomial Verifiability

- The *HAMPATH* problem has, though, a property called **polynomial verifiability**
- We don't know of a "fast" (i.e., polynomial time) algorithm to determine whether a directed graph contains a Hamiltonian path

The *HAMPATH* Problem: Polynomial Verifiability

- The *HAMPATH* problem has, though, a property called **polynomial verifiability**
- We don't know of a "fast" (i.e., polynomial time) algorithm to determine whether a directed graph contains a Hamiltonian path
- Still, if someone claims that a Hamiltonian path exists and gives it to us, we can "easily" check if that is true

The *HAMPATH* Problem: Polynomial Verifiability

- The *HAMPATH* problem has, though, a property called **polynomial verifiability**
- We don't know of a "fast" (i.e., polynomial time) algorithm to determine whether a directed graph contains a Hamiltonian path
- Still, if someone claims that a Hamiltonian path exists and gives it to us, we can "easily" check if that is true
- In other words, **veryfing** the existence of a Hamiltonian path may be much easier than **finding** if it exists

The $\overline{HAMPATH}$ Problem

- Some problems may not be even polynomially verifiable

The $\overline{HAMPATH}$ Problem

- Some problems may not be even polynomially verifiable
- For example, the complement of the $HAMPATH$ problem, i.e., $\overline{HAMPATH}$, is not polynomially verifiable

The $\overline{HAMPATH}$ Problem

- Some problems may not be even polynomially verifiable
- For example, the complement of the $HAMPATH$ problem, i.e., $\overline{HAMPATH}$, is not polynomially verifiable
- Even if we could determine (somehow) that a graph does not contain a Hamiltonian path, we don't know how to give a "proof" that someone else can use to verify its non-existence

The $\overline{HAMPATH}$ Problem

- Some problems may not be even polynomially verifiable
- For example, the complement of the *HAMPATH* problem, i.e., $\overline{HAMPATH}$, is not polynomially verifiable
- Even if we could determine (somehow) that a graph does not contain a Hamiltonian path, we don't know how to give a "proof" that someone else can use to verify its non-existence
- The only (known) way to verify the non-existence would be to use the same exponential-time algorithm used for making the claim in the first place

Polynomial Verifiability: Definition

Definition (Polynomial Verifiability)

A **verifier** for a language A is an algorithm V, where:

$$A = \{x \mid V \text{ accepts } \langle x, c \rangle \text{ for some string } c\}$$

Polynomial Verifiability: Definition

Definition (Polynomial Verifiability)

A **verifier** for a language A is an algorithm V, where:

$$A = \{x \mid V \text{ accepts } \langle x, c \rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of x.

Polynomial Verifiability: Definition

Definition (Polynomial Verifiability)

A **verifier** for a language A is an algorithm V, where:

$$A = \{x \mid V \text{ accepts } \langle x, c\rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of x.
A **polynomial time verifier** runs in polynomial time in the size of x.

Polynomial Verifiability: Definition

Definition (Polynomial Verifiability)

A **verifier** for a language A is an algorithm V, where:

$$A = \{x \mid V \text{ accepts } \langle x, c\rangle \text{ for some string } c\}$$

We measure the time of a verifier only in terms of the length of x.
A **polynomial time verifier** runs in polynomial time in the size of x.
A language A is **polynomially verifiable** if it has a polynomial time verifier.

The Verifier

- A verifier uses additional information – i.e., the string c in the Definition above – to check that $x \in A$

The Verifier

- A verifier uses additional information – i.e., the string c in the Definition above – to check that $x \in A$
- This information is also called a **certificate** (or **proof**) of membership in A

The Verifier

- A verifier uses additional information – i.e., the string c in the Definition above – to check that $x \in A$
- This information is also called a **certificate** (or **proof**) of membership in A
- Note that, for polynomial verifiers, the certificate c has polynomial length (in the size of x)

The Verifier

- A verifier uses additional information – i.e., the string c in the Definition above – to check that $x \in A$
- This information is also called a **certificate** (or **proof**) of membership in A
- Note that, for polynomial verifiers, the certificate c has polynomial length (in the size of x)

Example

For the *HAMPATH* problem, a certificate for the string $\langle G, s, t \rangle \in HAMPATH$ is just the Hamiltonian path from s to t.
The verifier can check in polynomial time that $\langle G, s, t \rangle \in HAMPATH$, given such certificate.

A Polynomial Verifier for *HAMPATH*

- Let's consider $\langle G, s, t\rangle \in HAMPATH$, where $G = (V, E)$, $|V| = n$, and $|E| \leq n^2 = O(n^2)$

A Polynomial Verifier for *HAMPATH*

- Let's consider $\langle G, s, t \rangle \in HAMPATH$, where $G = (V, E)$, $|V| = n$, and $|E| \leq n^2 = O(n^2)$
- Suppose we are given with a certificate c, i.e., a list of nodes $p_1, \ldots, p_n$ that is claimed to be a Hamiltonian path in G from s to t

A Polynomial Verifier for *HAMPATH*

- Let's consider $\langle G, s, t \rangle \in HAMPATH$, where $G = (V, E)$, $|V| = n$, and $|E| \leq n^2 = O(n^2)$
- Suppose we are given with a certificate c, i.e., a list of nodes $p_1, \ldots, p_n$ that is claimed to be a Hamiltonian path in G from s to t
- We can verify the certificate by checking:
  1. If each node in G appears exactly once in claimed path, which takes $O(n^2)$ time

A Polynomial Verifier for *HAMPATH*

- Let's consider $\langle G, s, t \rangle \in HAMPATH$, where $G = (V, E)$, $|V| = n$, and $|E| \leq n^2 = O(n^2)$
- Suppose we are given with a certificate c, i.e., a list of nodes $p_1, \ldots, p_n$ that is claimed to be a Hamiltonian path in G from s to t
- We can verify the certificate by checking:
  1. If each node in G appears exactly once in claimed path, which takes $O(n^2)$ time
  2. If each pair (p_i, p_{i+1}) is an edge in G, which also may take $O(n^2)$ time

A Polynomial Verifier for *HAMPATH*

- Let's consider $\langle G, s, t \rangle \in HAMPATH$, where $G = (V, E)$, $|V| = n$, and $|E| \leq n^2 = O(n^2)$
- Suppose we are given with a certificate c, i.e., a list of nodes $p_1, \ldots, p_n$ that is claimed to be a Hamiltonian path in G from s to t
- We can verify the certificate by checking:
  1. If each node in G appears exactly once in claimed path, which takes $O(n^2)$ time
  2. If each pair (p_i, p_{i+1}) is an edge in G, which also may take $O(n^2)$ time
- Overall, verification takes $O(n^2)$ steps, which is clearly polynomial in n

The Class NP: Definition

Definition (The Class NP)

NP is the class of languages/problems that have polynomial time verifiers.

- The class NP is crucial because it contains many problems of practical interest

The Class NP: Definition

Definition (The Class NP)

NP is the class of languages/problems that have polynomial time verifiers.

- The class NP is crucial because it contains many problems of practical interest
- For example, we have shown that $HAMPATH \in NP$

The Class NP: Definition

Definition (The Class NP)

NP is the class of languages/problems that have polynomial time verifiers.

- The class NP is crucial because it contains many problems of practical interest
- For example, we have shown that $HAMPATH \in NP$
- The term "NP" comes from **non-deterministic polynomial time**, and is derived from an alternative definition that makes use of non-deterministic Turing machines

A Non-Deterministic TM Solving *HAMPATH*

We can design a non-deterministic TM $N_{HAMPATH}$ to decide *HAMPATH*

Example (A non-deterministic decider for *HAMPATH*)

$N_{HAMPATH}$ = "On input string $\langle G, s, t\rangle$:

1. Write a list of n numbers: $p_1, \ldots, p_n$, where n is the number of nodes in G, i.e., $n = |V|$. Each number in the list is non-deterministically selected to be between 1 and n.

SAPIENZA UNIVERSITÀ DI ROMA

A Non-Deterministic TM Solving *HAMPATH*

We can design a non-deterministic TM $N_{HAMPATH}$ to decide *HAMPATH*

Example (A non-deterministic decider for *HAMPATH*)

$N_{HAMPATH}$ = "On input string $\langle G, s, t \rangle$:

1. Write a list of n numbers: $p_1, \ldots, p_n$, where n is the number of nodes in G, i.e., $n = |V|$. Each number in the list is non-deterministically selected to be between 1 and n.
2. Check for repetitions in the list; if any, **reject**.

A Non-Deterministic TM Solving *HAMPATH*

We can design a non-deterministic TM $N_{HAMPATH}$ to decide *HAMPATH*

Example (A non-deterministic decider for *HAMPATH*)

$N_{HAMPATH}$ = "On input string $\langle G, s, t \rangle$:

1. Write a list of n numbers: $p_1, \ldots, p_n$, where n is the number of nodes in G, i.e., $n = |V|$. Each number in the list is non-deterministically selected to be between 1 and n.
2. Check for repetitions in the list; if any, **reject**.
3. Check if $s = p_1$ and $t = p_n$; if either fails, **reject**.

A Non-Deterministic TM Solving *HAMPATH*

We can design a non-deterministic TM $N_{HAMPATH}$ to decide *HAMPATH*

Example (A non-deterministic decider for *HAMPATH*)

$N_{HAMPATH}$ = "On input string $\langle G, s, t \rangle$:

1. Write a list of n numbers: $p_1, \ldots, p_n$, where n is the number of nodes in G, i.e., $n = |V|$. Each number in the list is non-deterministically selected to be between 1 and n.
2. Check for repetitions in the list; if any, **reject**.
3. Check if $s = p_1$ and $t = p_n$; if either fails, **reject**.
4. For each $1 \leq i < n$, check whether (p_i, p_{i+1}) is an edge of G; if any is not, **reject**, otherwise **accept**."

Complexity Analysis of $N_{HAMPATH}$

- To analyze this algorithm and check it runs in non-deterministic polynomial time, we examine each of its stages (assuming G is represented as adjacency list):

Complexity Analysis of $N_{HAMPATH}$

- To analyze this algorithm and check it runs in non-deterministic polynomial time, we examine each of its stages (assuming G is represented as adjacency list):
- Stage 1 runs trivially in $O(n)$ time, therefore in polynomial time;

Complexity Analysis of $N_{HAMPATH}$

- To analyze this algorithm and check it runs in non-deterministic polynomial time, we examine each of its stages (assuming G is represented as adjacency list):
- Stage 1 runs trivially in $O(n)$ time, therefore in polynomial time;
- Stages 2 and 3 are simple polynomial-time checks, i.e., $O(n^2) + O(1) = O(n^2)$ time

Complexity Analysis of $N_{HAMPATH}$

- To analyze this algorithm and check it runs in non-deterministic polynomial time, we examine each of its stages (assuming G is represented as adjacency list):
- Stage 1 runs trivially in $O(n)$ time, therefore in polynomial time;
- Stages 2 and 3 are simple polynomial-time checks, i.e., $O(n^2) + O(1) = O(n^2)$ time
- Finally, also stage 4 runs in polynomial time, as we must check if each of the n pairs is an actual edge, thereby needing $O(n^2)$ time

NP and NTM

Theorem (NP and NTM)

A language is NP iff it is decided by some non-deterministic polynomial Turing machine.

NP and NTM

Theorem (NP and NTM)

A language is NP iff it is decided by some non-deterministic polynomial Turing machine.

Proof.

The idea of the proof is based on converting a polynomial time verifier to an equivalent polynomial time NTM, and vice versa.
The NTM simulates the verifier by guessing the certificate.
The verifier simulates the NTM by using the accepting branch as the certificate. $\square$

NP and NTM: Proof

- ($\Rightarrow$) Let $A \in NP$, we must show that A is decided by a polynomial time NTM N

NP and NTM: Proof

- ($\Rightarrow$) Let $A \in NP$, we must show that A is decided by a polynomial time NTM N
- Let V be the polynomial time verifier for A that exists by the definition of NP

NP and NTM: Proof

- ($\Rightarrow$) Let $A \in NP$, we must show that A is decided by a polynomial time NTM N
- Let V be the polynomial time verifier for A that exists by the definition of NP
- Assume V is a TM that runs in n^k steps, then we can construct a NTM N as follows:
  $N =$ "On input string x of length n:
  1. Non-deterministically select the string c of length at most n^k.

NP and NTM: Proof

- ($\Rightarrow$) Let $A \in NP$, we must show that A is decided by a polynomial time NTM N
- Let V be the polynomial time verifier for A that exists by the definition of NP
- Assume V is a TM that runs in n^k steps, then we can construct a NTM N as follows:
  $N =$ "On input string x of length n:
  1. Non-deterministically select the string c of length at most n^k.
  2. Run V on the input $\langle x, c \rangle$.

NP and NTM: Proof

- ($\Rightarrow$) Let $A \in NP$, we must show that A is decided by a polynomial time NTM N
- Let V be the polynomial time verifier for A that exists by the definition of NP
- Assume V is a TM that runs in n^k steps, then we can construct a NTM N as follows:

  $N =$ "On input string x of length n:

  1. Non-deterministically select the string c of length at most n^k.
  2. Run V on the input $\langle x, c \rangle$.
  3. If V accepts, **accept**; otherwise, **reject**."

NP and NTM: Proof

- ($\Leftarrow$) Assume A is decided by a NTM N, we can construct a polynomial time verifier V as follows:
  $V =$ "On input $\langle x, c \rangle$:
  1. Simulate N on input x, treating each symbol of c as the encoding of the non-deterministic choice to make at each step (Remember: N decides A!).

NP and NTM: Proof

- ($\Leftarrow$) Assume A is decided by a NTM N, we can construct a polynomial time verifier V as follows:
  V = "On input $\langle x, c \rangle$:
  1. Simulate N on input x, treating each symbol of c as the encoding of the non-deterministic choice to make at each step (Remember: N decides A!).
  2. If this branch of N's computation accepts, **accept**; otherwise, **reject**."

NP: Two Definitions

- So far, we have given **two definitions** of the class NP:
  1. The class of problems whose solution can be verified in polynomial time by a polynomial time verifier;

NP: Two Definitions

- So far, we have given **two definitions** of the class NP:
  1. The class of problems whose solution can be verified in polynomial time by a polynomial time verifier;
  2. The class of problems that can be decided by a polynomial time non-deterministic TM.

NP: Two Definitions

- So far, we have given **two definitions** of the class *NP*:
  1. The class of problems whose solution can be verified in polynomial time by a polynomial time verifier;
  2. The class of problems that can be decided by a polynomial time non-deterministic TM.
- Also, we showed that the two definitions above are **equivalent**

$NTIME(t(n))$

Analogously to the deterministic time complexity class $TIME(t(n))$, we can define the non-deterministic time complexity class $NTIME(t(n))$ as follows:

Definition ($NTIME(t(n))$)

$NTIME(t(n)) = \{L \mid L \text{ is decided by a NTM in } O(t(n))\}$

$NTIME(t(n))$

Analogously to the deterministic time complexity class $TIME(t(n))$, we can define the non-deterministic time complexity class $NTIME(t(n))$ as follows:

Definition ($NTIME(t(n))$)

$NTIME(t(n)) = \{L \mid L \text{ is decided by a NTM in } O(t(n))\}$

Corollary ($NP = \bigcup_k NTIME(n^k)$)

The class NP is insensitive to the choice of "reasonable" non-deterministic computational model, as all such models are polynomially equivalent.

Examples of Problems in NP: $CLIQUE$

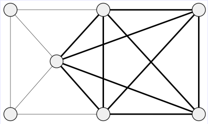
Definition (The $CLIQUE$ Problem)

A **clique** in an undirected graph is a subgraph, where every two nodes are connected by an edge.

Examples of Problems in NP: *CLIQUE*

Definition (The *CLIQUE* Problem)

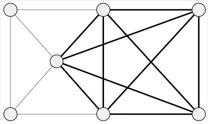
A **clique** in an undirected graph is a subgraph, where every two nodes are connected by an edge.
A k-clique is a clique that contains k nodes (e.g., a 5-clique is shown in the picture below).



Examples of Problems in NP: $CLIQUE$

Definition (The $CLIQUE$ Problem)

A **clique** in an undirected graph is a subgraph, where every two nodes are connected by an edge.
A k-clique is a clique that contains k nodes (e.g., a 5-clique is shown in the picture below).



The $CLIQUE$ problem is to determine whether a graph contains a clique of a specified size:

$$CLIQUE = \{\langle G, k \rangle \mid G \text{ is undirected graph with a } k\text{-clique}\}$$

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The CLIQUE problem is in the class NP

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The CLIQUE problem is in the class NP

The clique is the certificate.

We use the first definition of NP, thereby describing a polynomial time verifier V for $CLIQUE$:

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G.

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The $CLIQUE$ problem is in the class NP

The clique is the certificate.

We use the first definition of NP, thereby describing a polynomial time verifier V for $CLIQUE$:

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G.
2. Test whether G contains all edges connecting nodes in c.

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The CLIQUE problem is in the class NP

The clique is the certificate.

We use the first definition of NP, thereby describing a polynomial time verifier V for $CLIQUE$:

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G.
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, **accept**; otherwise, **reject**."

□

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The $CLIQUE$ problem is in the class NP

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The CLIQUE problem is in the class NP

A polynomial-time NTM decides $CLIQUE$.

We use the second definition of NP, thereby describing a polynomial time NTM N that decides $CLIQUE$:

$V =$ "On input $\langle G, k \rangle$:

1. Non-deterministically guess a subset c of k nodes of G.

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The $CLIQUE$ problem is in the class NP

A polynomial-time NTM decides $CLIQUE$.

We use the second definition of NP, thereby describing a polynomial time NTM N that decides $CLIQUE$:

$V =$ "On input $\langle G, k\rangle$:

1. Non-deterministically guess a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.

Examples of Problems in NP: $CLIQUE$

Theorem ($CLIQUE \in NP$)

The CLIQUE problem is in the class NP

A polynomial-time NTM decides $CLIQUE$.

We use the second definition of NP, thereby describing a polynomial time NTM N that decides $CLIQUE$:

$V =$ "On input $\langle G, k \rangle$:

1. Non-deterministically guess a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, **accept**; otherwise, **reject**."

$\square$

Examples of Problems in NP: $SUBSET\_SUM$

Definition (The $SUBSET\_SUM$ Problem)

Let $x_1, \ldots, x_k$ be a collection of integers, i.e., $x_i \in \mathbb{Z}\ \forall i \in \{1, \ldots, k\}$, and $t \in \mathbb{Z}$ a **target**.

Examples of Problems in NP: $SUBSET\_SUM$

Definition (The $SUBSET\_SUM$ Problem)

Let $x_1, \ldots, x_k$ be a collection of integers, i.e., $x_i \in \mathbb{Z}\ \forall i \in \{1, \ldots, k\}$, and $t \in \mathbb{Z}$ a **target**.
The $SUBSET\_SUM$ problem is to determine if the collection contains a subcollection whose sum is exactly t.

$$SUBSET\_SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \subseteq \mathbb{Z}, \exists S' \subseteq S, \sum_{x \in S'} = t\}$$

Examples of Problems in NP: $SUBSET\_SUM$

Definition (The $SUBSET\_SUM$ Problem)

Let $x_1, \ldots, x_k$ be a collection of integers, i.e., $x_i \in \mathbb{Z}\ \forall i \in \{1, \ldots, k\}$, and $t \in \mathbb{Z}$ a **target**.
The $SUBSET\_SUM$ problem is to determine if the collection contains a subcollection whose sum is exactly t.

$$SUBSET\_SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \subseteq \mathbb{Z}, \exists S' \subseteq S, \sum_{x \in S'} = t\}$$

Example

Let $S = \{4, 11, 16, 21, 27\}$ and $t = 25$. Thus, $\langle S, t\rangle \in SUBSET\_SUM$ because $S' = \{4, 21\}$ and $=4 + 21 = 25$.

Examples of Problems in NP: $SUBSET\_SUM$

Definition (The $SUBSET\_SUM$ Problem)

Let $x_1, \ldots, x_k$ be a collection of integers, i.e., $x_i \in \mathbb{Z}\ \forall i \in \{1, \ldots, k\}$, and $t \in \mathbb{Z}$ a **target**.

The $SUBSET\_SUM$ problem is to determine if the collection contains a subcollection whose sum is exactly t.

$$SUBSET\_SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \subseteq \mathbb{Z}, \exists S' \subseteq S, \sum_{x \in S'} = t\}$$

Example

Let $S = \{4, 11, 16, 21, 27\}$ and $t = 25$. Thus, $\langle S, t\rangle \in SUBSET\_SUM$ because $S' = \{4, 21\}$ and $=4 + 21 = 25$.

Note that S and S' are, in fact, considered **multisets** and so repetitions are allowed.

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

The subset is the certificate.

We use the first definition of NP, thereby describing a polynomial time verifier V for $SUBSET\_SUM$:

$V =$ "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of numbers that sum to t.

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

The subset is the certificate.

We use the first definition of NP, thereby describing a polynomial time verifier V for $SUBSET\_SUM$:

$V =$ "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

The subset is the certificate.

We use the first definition of NP, thereby describing a polynomial time verifier V for $SUBSET\_SUM$:

$V =$ "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.
3. If both pass, **accept**; otherwise, **reject**."

$\square$

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

A polynomial-time NTM decides $CLIQUE$.

We use the second definition of NP, thereby describing a polynomial time NTM N that decides $SUBSET\_SUM$:

$N =$ "On input $\langle S, t \rangle$:

1. Non-deterministically select a subset c of the numbers in S

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

A polynomial-time NTM decides $CLIQUE$.

We use the second definition of NP, thereby describing a polynomial time NTM N that decides $SUBSET\_SUM$:

$N =$ "On input $\langle S, t \rangle$:

1. Non-deterministically select a subset c of the numbers in S
2. Test whether c is a collection of numbers that sum to t.

Examples of Problems in NP: $SUBSET\_SUM$

Theorem ($SUBSET\_SUM \in NP$)

The $SUBSET\_SUM$ problem is in the class NP

A polynomial-time NTM decides $CLIQUE$.

We use the second definition of NP, thereby describing a polynomial time NTM N that decides $SUBSET\_SUM$:

$N =$ "On input $\langle S, t \rangle$:

1. Non-deterministically select a subset c of the numbers in S
2. Test whether c is a collection of numbers that sum to t.
3. If the test above passes, **accept**; otherwise, **reject**."

□

Table of Contents

P vs. NP

- P is the class of languages for which membership can be **decided** in polynomial time

P vs. NP

- P is the class of languages for which membership can be **decided** in polynomial time
- NP is the class of languages for which membership can be **verified** in polynomial time

P vs. NP

- P is the class of languages for which membership can be **decided** in polynomial time
- NP is the class of languages for which membership can be **verified** in polynomial time
  (Equivalently, NP is the class of languages that are decidable in polynomial time by an NTM)

P vs. NP

- P is the class of languages for which membership can be **decided** in polynomial time
- NP is the class of languages for which membership can be **verified** in polynomial time
  (Equivalently, NP is the class of languages that are decidable in polynomial time by an NTM)
- We loosely refer to "polynomial time" as "quick"

P vs. NP

- We have presented examples of languages like *HAMPATH* or *CLIQUE* that are members of NP yet we don't know if they are in P

P vs. NP

- We have presented examples of languages like *HAMPATH* or *CLIQUE* that are members of NP yet we don't know if they are in P
- As hard as it may be to imagine, P and NP could in fact be equal

P vs. NP

- We have presented examples of languages like *HAMPATH* or *CLIQUE* that are members of NP yet we don't know if they are in P
- As hard as it may be to imagine, P and NP could in fact be equal
- So far, we haven't been able to **prove** the existence of a single language that is in NP but not in P

P vs. NP

- The question of whether $P = NP$ is a major unsolved problem in theoretical computer science and contemporary mathematics

P vs. NP

- The question of whether $P = NP$ is a major unsolved problem in theoretical computer science and contemporary mathematics
- It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute, each of which carries a US$1,000,000 prize for the first correct solution

P vs. NP

- The question of whether $P = NP$ is a major unsolved problem in theoretical computer science and contemporary mathematics
- It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute, each of which carries a US$1,000,000 prize for the first correct solution
- At its core, it asks whether every problem whose solution can be quickly verified can **also** be solved quickly

P vs. NP

- The question of whether $P = NP$ is a major unsolved problem in theoretical computer science and contemporary mathematics
- It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute, each of which carries a US$1,000,000 prize for the first correct solution
- At its core, it asks whether every problem whose solution can be quickly verified can **also** be solved quickly
- If $P = NP$, any polynomially verifiable problem would be also polynomially decidable

P vs. NP

- Most researchers tend to believe that, in fact, $P \neq NP$

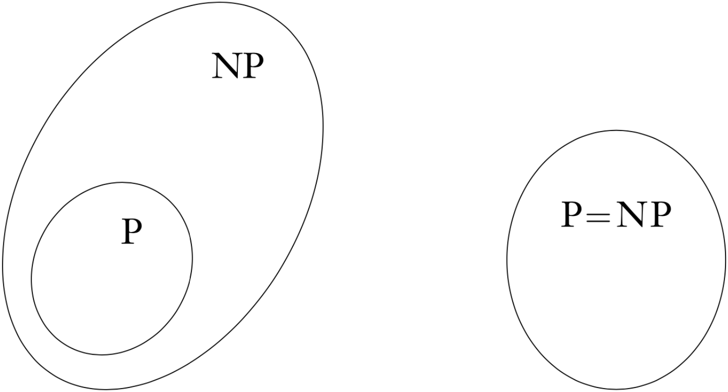
P vs. NP

- Most researchers tend to believe that, in fact, $P \neq NP$
- To check for a proof of $P = NP$, many people have tried to design a polynomial time decider for some well-known problems in NP, without success

P vs. NP

- Most researchers tend to believe that, in fact, $P \neq NP$
- To check for a proof of $P = NP$, many people have tried to design a polynomial time decider for some well-known problems in NP, without success
- On the other hand, proving that $P \neq NP$ would need to show that no polynomial time algorithm exists to replace brute-force search deciders

P vs. NP



P vs. NP

- The best method known for deciding languages in NP deterministically requires **exponential time**

P vs. NP

- The best method known for deciding languages in NP deterministically requires **exponential time**
- In other words:

$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$$

P vs. NP

- The best method known for deciding languages in NP deterministically requires **exponential time**
- In other words:

$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k})$$

- Unfortunately, we don't know whether NP is contained in a "smaller" deterministic time complexity class

Table of Contents

Final Remarks

- P is the class of languages for which membership can be **decided** in polynomial time

Final Remarks

- P is the class of languages for which membership can be **decided** in polynomial time
- NP is the class of languages for which membership can be **verified** in polynomial time

Final Remarks

- P is the class of languages for which membership can be **decided** in polynomial time
- NP is the class of languages for which membership can be **verified** in polynomial time
  (Equivalently, NP is the class of languages that are decidable in polynomial time by an NTM)

Final Remarks

- P is the class of languages for which membership can be **decided** in polynomial time
- NP is the class of languages for which membership can be **verified** in polynomial time
(Equivalently, NP is the class of languages that are decidable in polynomial time by an NTM)
- We loosely refer to "polynomial time" as "quick" and "exponential time" as "intractable"

Final Remarks

- P is the class of languages for which membership can be **decided** in polynomial time
- NP is the class of languages for which membership can be **verified** in polynomial time
  (Equivalently, NP is the class of languages that are decidable in polynomial time by an NTM)
- We loosely refer to "polynomial time" as "quick" and "exponential time" as "intractable"
- The P vs. NP problem is still open and a proof of $P = NP$ (resp., $P \neq NP$) would have a tremendous impact on the computational complexity hierarchy