Complexity

Asai Asaithambi

Fall 2016

Topic Overview

What does this topic cover?

Topic Overview

What does this topic cover?

- Terminology

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation
- Some Algorithm Analysis

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation
- Some Algorithm Analysis
  - Based on Pseudocode Notation

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation
- Some Algorithm Analysis
  - Based on Pseudocode Notation
  - Based on TM Descriptions

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation
- Some Algorithm Analysis
  - Based on Pseudocode Notation
  - Based on TM Descriptions
- Classes P and NP

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation
- Some Algorithm Analysis
  - Based on Pseudocode Notation
  - Based on TM Descriptions
- Classes P and NP
  - Example Problems in P

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation
- Some Algorithm Analysis
  - Based on Pseudocode Notation
  - Based on TM Descriptions
- Classes P and NP
  - Example Problems in P
  - Example Problems in NP

Topic Overview

What does this topic cover?

- Terminology
  - TM Recognizable/Decidable
  - Algorithms and Complexity
  - Big O and Small o Notation
- Some Algorithm Analysis
  - Based on Pseudocode Notation
  - Based on TM Descriptions
- Classes P and NP
  - Example Problems in P
  - Example Problems in NP
  - NP Completeness

Terminology: TM Recognizable/Decidable

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or
  - Never HALT

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or
  - Never HALT
    Never deciding whether to accept/reject.

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or
  - Never HALT
    Never deciding whether to accept/reject.
- A TM that halts on all inputs is called a *decider*.

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or
  - Never HALT
    Never deciding whether to accept/reject.
- A TM that halts on all inputs is called a *decider*.
- A decider recognizing a language:

Terminology: TM Recognizable/Decidable

- A language is "Turing-Recognizable":
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or
  - Never HALT
    Never deciding whether to accept/reject.
- A TM that halts on all inputs is called a *decider*.
- A decider recognizing a language:
  is also said to "decide" that language.

Terminology: TM Recognizable/Decidable

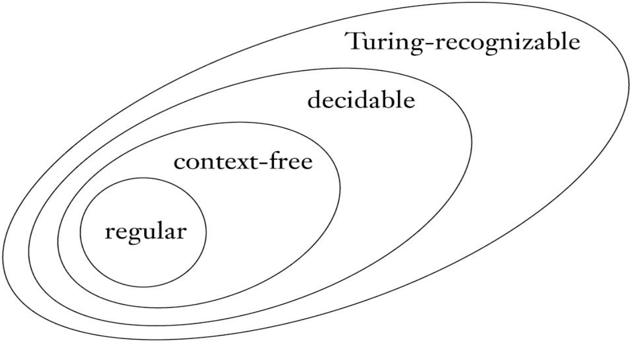
- A language is “Turing-Recognizable”:
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or
  - Never HALT
    Never deciding whether to accept/reject.
- A TM that halts on all inputs is called a *decider*.
- A decider recognizing a language:
  is also said to “decide” that language.
- A language is said to be “Decidable”

Terminology: TM Recognizable/Decidable

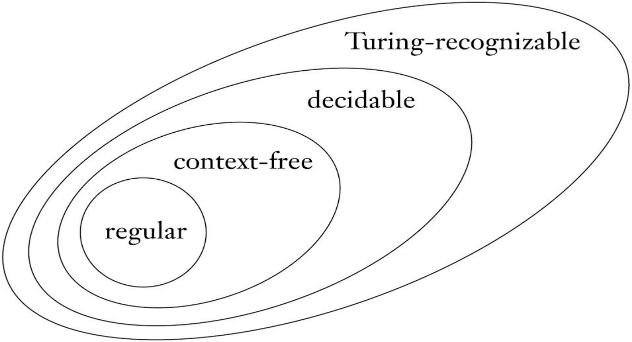
- A language is “Turing-Recognizable”:
  If some TM recognizes it.
- On processing an input string, a TM may:
  - Reach an ACCEPT state
    Accepting the string; or
  - Reach a REJECT state
    Rejecting the string; or
  - Never HALT
    Never deciding whether to accept/reject.
- A TM that halts on all inputs is called a *decider*.
- A decider recognizing a language:
  is also said to “decide” that language.
- A language is said to be “Decidable”
  if some TM decides it.

The Language Onion

The Language Onion

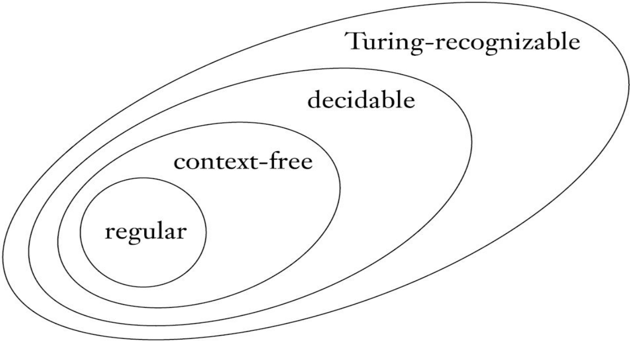


The Language Onion



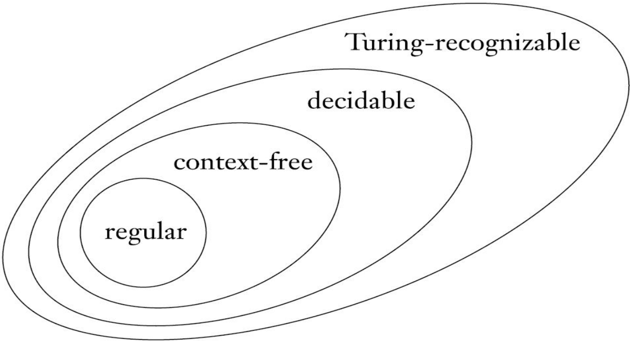
- Every decidable language is Turing-Recognizable,

The Language Onion



- Every decidable language is Turing-Recognizable, but not conversely.

The Language Onion



- Every decidable language is Turing-Recognizable,
  but not conversely.
- All examples of the languages we have seen are decidable.

Why Study Complexity?

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.
  - The *running time* or *time complexity* of M is:

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.
  - The *running time* or *time complexity* of M is:
  - the function $f : \mathbb{N} \mapsto \mathbb{N}$,

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.
  - The *running time* or *time complexity* of M is:
  - the function $f : \mathbb{N} \mapsto \mathbb{N}$,
  - where $f(n)$ is the maximum number of steps that

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.
  - The *running time* or *time complexity* of M is:
  - the function $f : \mathbb{N} \mapsto \mathbb{N}$,
  - where $f(n)$ is the maximum number of steps that M uses on any input of length n.

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.
  - The *running time* or *time complexity* of M is:
  - the function $f : \mathbb{N} \mapsto \mathbb{N}$,
  - where $f(n)$ is the maximum number of steps that M uses on any input of length n.
- If the running time of M is $f(n)$, then we say that

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.
  - The *running time* or *time complexity* of M is:
  - the function $f : \mathbb{N} \mapsto \mathbb{N}$,
  - where $f(n)$ is the maximum number of steps that M uses on any input of length n.
- If the running time of M is $f(n)$, then we say that
- M runs in time $f(n)$ and that M is an $f(n)$ time TM.

Why Study Complexity?

- Is a TM looping or simply taking a long time to decide?
- Knowing how much time a TM will need may be useful.
- How do we characterize the time that may be required?
  - Let M be a deterministic TM that halts on all inputs.
  - The *running time* or *time complexity* of M is:
  - the function $f : \mathbb{N} \mapsto \mathbb{N}$,
  - where $f(n)$ is the maximum number of steps that M uses on any input of length n.
- If the running time of M is $f(n)$, then we say that
- M runs in time $f(n)$ and that M is an $f(n)$ time TM.
- Customarily we use n to represent the length of the input.

Big O Notation

Big O Notation

- It is not necessary to know the *exact* amount of time.

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function,

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$,

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:
- for every integer $n \geq n_0$, it is true that $f(n) \leq c\, g(n)$.

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:
- for every integer $n \geq n_0$, it is true that $f(n) \leq c\, g(n)$.
- When $f(n) = O(g(n))$, we say that:

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:
- for every integer $n \geq n_0$, it is true that $f(n) \leq c\, g(n)$.
- When $f(n) = O(g(n))$, we say that:
  $g(n)$ is an

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:
- for every integer $n \geq n_0$, it is true that $f(n) \leq c\, g(n)$.
- When $f(n) = O(g(n))$, we say that:
  $g(n)$ is an *upper bound* for $f(n)$.

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:
- for every integer $n \geq n_0$, it is true that $f(n) \leq c\, g(n)$.
- When $f(n) = O(g(n))$, we say that:
  $g(n)$ is an *upper bound* for $f(n)$.
- More precisely:

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:
- for every integer $n \geq n_0$, it is true that $f(n) \leq c\, g(n)$.
- When $f(n) = O(g(n))$, we say that:
  $g(n)$ is an *upper bound* for $f(n)$.
- More precisely:
  $g(n)$ is an *asymptotic upper bound* for $f(n)$.

Big O Notation

- It is not necessary to know the *exact* amount of time.
- How fast does $f(n)$ grow as n grows?
- Compare $f(n)$ with another *standard* function, say, $g(n)$.
- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = O(g(n))$, (read $f(n)$ is Big O of $g(n)$), if:
- positive integers c and n_0 exist such that:
- for every integer $n \geq n_0$, it is true that $f(n) \leq c\, g(n)$.
- When $f(n) = O(g(n))$, we say that:
  $g(n)$ is an *upper bound* for $f(n)$.
- More precisely:
  $g(n)$ is an *asymptotic upper bound* for $f(n)$.
- We suppress constant factors.

Big O Notation: Example

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that
- $f(n) = 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for some $n \geq n_0$.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that
- $f(n) = 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for some $n \geq n_0$.
- This means that $n^3 - 2n^2 - 22n - 6 \geq 0$.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that
- $f(n) = 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for some $n \geq n_0$.
- This means that $n^3 - 2n^2 - 22n - 6 \geq 0$.
- The above inequality holds for $n \geq 6$.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that
- $f(n) = 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for some $n \geq n_0$.
- This means that $n^3 - 2n^2 - 22n - 6 \geq 0$.
- The above inequality holds for $n \geq 6$.
- Thus, we have $f(n) \leq 6n^3$ for $n \geq 6$.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that
- $f(n) = 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for some $n \geq n_0$.
- This means that $n^3 - 2n^2 - 22n - 6 \geq 0$.
- The above inequality holds for $n \geq 6$.
- Thus, we have $f(n) \leq 6n^3$ for $n \geq 6$.
- Therefore $f(n) = O(n^3)$.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that
- $f(n) = 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for some $n \geq n_0$.
- This means that $n^3 - 2n^2 - 22n - 6 \geq 0$.
- The above inequality holds for $n \geq 6$.
- Thus, we have $f(n) \leq 6n^3$ for $n \geq 6$.
- Therefore $f(n) = O(n^3)$.
- Note that $f(n) = O(n^k)$ for all $k \geq 3$.

Big O Notation: Example

- Let $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Then, $f(n) = O(n^3)$
- Note that $5n^3$ is the fastest growing term.
- All the other terms can be bounded by n^3.
- In other words, we could show that
- $f(n) = 5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for some $n \geq n_0$.
- This means that $n^3 - 2n^2 - 22n - 6 \geq 0$.
- The above inequality holds for $n \geq 6$.
- Thus, we have $f(n) \leq 6n^3$ for $n \geq 6$.
- Therefore $f(n) = O(n^3)$.
- Note that $f(n) = O(n^k)$ for all $k \geq 3$.
- Note also that $f(n) \neq O(n^k)$ for any $k < 3$.

Big O Notation (cont.)

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n \log_2 n + \log_2 \log_2 n + 2$.

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n \log_2 n + \log_2 \log_2 n + 2$.
- Then, $f(n) = O(n \log n)$.

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n \log_2 n + \log_2 \log_2 n + 2$.
- Then, $f(n) = O(n \log n)$.
- Other forms of growth functions:

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n \log_2 n + \log_2 \log_2 n + 2$.
- Then, $f(n) = O(n \log n)$.
- Other forms of growth functions:

$$f(n) = O(n^2) + O(n) \Rightarrow f(n) = O(n^2)$$

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n\log_2 n + \log_2 \log_2 n + 2$.
- Then, $f(n) = O(n \log n)$.
- Other forms of growth functions:

$$f(n) = O(n^2) + O(n) \Rightarrow f(n) = O(n^2)$$

$$f(n) = 2^{O(n)} \Rightarrow f(n) = 2^{cn} \text{ for some constant } c > 0.$$

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n \log_2 n + \log_2 \log_2 n + 2$.
- Then, $f(n) = O(n \log n)$.
- Other forms of growth functions:

$$f(n) = O(n^2) + O(n) \Rightarrow f(n) = O(n^2)$$

$$f(n) = 2^{O(n)} \Rightarrow f(n) = 2^{cn} \text{ for some constant } c > 0.$$

$$n = 2^{\log_2 n} \Rightarrow n^c = 2^{c \log_2 n} = 2^{O(\log n)}.$$

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n \log_2 n + \log_2 \log_2 n + 2$.
- Then, $f(n) = O(n \log n)$.
- Other forms of growth functions:

$$f(n) = O(n^2) + O(n) \Rightarrow f(n) = O(n^2)$$

$$f(n) = 2^{O(n)} \Rightarrow f(n) = 2^{cn} \text{ for some constant } c > 0.$$

$$n = 2^{\log_2 n} \Rightarrow n^c = 2^{c \log_2 n} = 2^{O(\log n)}.$$

- *Polynomial Bounds*: n^c for $c > 0$.

Big O Notation (cont.)

- Note $\log_a n = \log_b n / \log_b a$
- $\log_a n = O(\log n)$ as they differ by a constant factor.
- Suppose $f(n) = 3n \log_2 n + \log_2 \log_2 n + 2$.
- Then, $f(n) = O(n \log n)$.
- Other forms of growth functions:

$$f(n) = O(n^2) + O(n) \Rightarrow f(n) = O(n^2)$$

$$f(n) = 2^{O(n)} \Rightarrow f(n) = 2^{cn} \text{ for some constant } c > 0.$$

$$n = 2^{\log_2 n} \Rightarrow n^c = 2^{c \log_2 n} = 2^{O(\log n)}.$$

- *Polynomial Bounds*: n^c for $c > 0$.
- *Exponential Bounds*: 2^{n^δ} for $\delta > 0$.

Small o Notation

Small o Notation

- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.

Small o Notation

- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = o(g(n))$,

Small o Notation

- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = o(g(n))$, (read $f(n)$ is Small o of $g(n)$), if:

Small o Notation

- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = o(g(n))$, (read $f(n)$ is Small o of $g(n)$), if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Small o Notation

- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = o(g(n))$, (read $f(n)$ is Small o of $g(n)$), if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

- Equivalently, for any real number $c > 0$,

Small o Notation

- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = o(g(n))$, (read $f(n)$ is Small o of $g(n)$), if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

- Equivalently, for any real number $c > 0$,
  there exists an integer n_0 such that:

Small o Notation

- Let f and g be functions, $f, g : \mathbb{N} \mapsto \mathbb{R}^+$.
- We say $f(n) = o(g(n))$, (read $f(n)$ is Small o of $g(n)$), if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

- Equivalently, for any real number $c > 0$,
  there exists an integer n_0 such that:
- For every integer $n \geq n_0$, it is true that $f(n) < c\, g(n)$.

Analysis of Algorithms

Analysis of Algorithms

- Size or problem (n)

Analysis of Algorithms

- Size or problem (n)
- Basic operation (varies with algorithm)

Analysis of Algorithms

- Size or problem (n)
- Basic operation (varies with algorithm)
- Number of times the basic operation gets executed

Analysis of Algorithms

- Size or problem (n)
- Basic operation (varies with algorithm)
- Number of times the basic operation gets executed
- Express as a function of n

Analysis of Algorithms

- Size or problem (n)
- Basic operation (varies with algorithm)
- Number of times the basic operation gets executed
- Express as a function of n
- Analyze the growth of $f(n)$ as n is increased

Analysis of Algorithms

- Size or problem (n)
- Basic operation (varies with algorithm)
- Number of times the basic operation gets executed
- Express as a function of n
- Analyze the growth of $f(n)$ as n is increased
- Use Big O notation

Algorithm Analysis: Examples

Algorithm Analysis: Examples

How many times is `Hello` printed?

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Hello is printed `n` times.

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Hello is printed `n` times.
This is expressed as $O(n)$.

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Hello is printed `n` times.
This is expressed as $O(n)$.

Consider another example:

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Hello is printed `n` times.
This is expressed as $O(n)$.

Consider another example:

```
for (i = 1; i < n; i *= 2)
    print "Hello"
```

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Hello is printed `n` times.
This is expressed as $O(n)$.

Consider another example:

```
for (i = 1; i < n; i *= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} < n \leq 2^k$.

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Hello is printed `n` times.
This is expressed as $O(n)$.

Consider another example:

```
for (i = 1; i < n; i *= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} < n \leq 2^k$.
This amounts to $\lceil \log_2 \mathtt{n} \rceil$ times.

Algorithm Analysis: Examples

How many times is `Hello` printed?

```
for (i = 0; i < n; i++)
    print "Hello"
```

Hello is printed `n` times.
This is expressed as $O(n)$.

Consider another example:

```
for (i = 1; i < n; i *= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} < n \leq 2^k$.
This amounts to $\lceil \log_2 \mathtt{n} \rceil$ times.
This is expressed as $O(\log n)$.

Algorithm Analysis: Examples (cont.)

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.
This amounts to $\lfloor \log_2 \mathtt{n} \rfloor + 1$ times.

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.
This amounts to $\lfloor \log_2 \mathtt{n} \rfloor + 1$ times.
This is expressed as $O(\log n)$.

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.
This amounts to $\lfloor \log_2 \mathtt{n} \rfloor + 1$ times.
This is expressed as $O(\log n)$.

Consider another example:

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.
This amounts to $\lfloor \log_2 \mathtt{n} \rfloor + 1$ times.
This is expressed as $O(\log n)$.

Consider another example:

```
for (i = n; i > 0; i /= 3)
    print "Hello"
```

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.
This amounts to $\lfloor \log_2 \mathtt{n} \rfloor + 1$ times.
This is expressed as $O(\log n)$.

Consider another example:

```
for (i = n; i > 0; i /= 3)
    print "Hello"
```

Hello is printed k times, with $3^{k-1} \leq n < 3^k$.

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.
This amounts to $\lfloor \log_2 \mathtt{n} \rfloor + 1$ times.
This is expressed as $O(\log n)$.

Consider another example:

```
for (i = n; i > 0; i /= 3)
    print "Hello"
```

Hello is printed k times, with $3^{k-1} \leq n < 3^k$.
This amounts to $\lfloor \log_3 \mathtt{n} \rfloor + 1$ times.

Algorithm Analysis: Examples (cont.)

How many times is `Hello` printed?

```
for (i = n; i > 0; i /= 2)
    print "Hello"
```

Hello is printed k times, with $2^{k-1} \leq n < 2^k$.
This amounts to $\lfloor \log_2 \mathtt{n} \rfloor + 1$ times.
This is expressed as $O(\log n)$.

Consider another example:

```
for (i = n; i > 0; i /= 3)
    print "Hello"
```

Hello is printed k times, with $3^{k-1} \leq n < 3^k$.
This amounts to $\lfloor \log_3 \mathtt{n} \rfloor + 1$ times.
This is also expressed as $O(\log n)$.

Algorithm Analysis: More Examples

Algorithm Analysis: More Examples

How many times is `Hello` printed?

Algorithm Analysis: More Examples

How many times is `Hello` printed?

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
    print "Hello"
```

Algorithm Analysis: More Examples

How many times is `Hello` printed?

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
    print "Hello"
```

Hello is printed n^2 times.

Algorithm Analysis: More Examples

How many times is `Hello` printed?

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
    print "Hello"
```

Hello is printed n^2 times.
This is expressed as $O(n^2)$.

Algorithm Analysis: More Examples

How many times is `Hello` printed?

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
    print "Hello"
```

Hello is printed n^2 times.
This is expressed as $O(n^2)$.

Consider another example:

Algorithm Analysis: More Examples

How many times is `Hello` printed?

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
    print "Hello"
```

Hello is printed n^2 times.
This is expressed as $O(n^2)$.

Consider another example:

```
for (k = 1; k <= n; k++)
for (i = 1; i <= k; i++)
    print "Hello"
```

Algorithm Analysis: More Examples

How many times is `Hello` printed?

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
    print "Hello"
```

Hello is printed n^2 times.
This is expressed as $O(n^2)$.

Consider another example:

```
for (k = 1; k <= n; k++)
for (i = 1; i <= k; i++)
    print "Hello"
```

Hello is printed $(n^2 + n)/2$ times.

Algorithm Analysis: More Examples

How many times is `Hello` printed?

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
    print "Hello"
```

Hello is printed n^2 times.
This is expressed as $O(n^2)$.

Consider another example:

```
for (k = 1; k <= n; k++)
for (i = 1; i <= k; i++)
    print "Hello"
```

Hello is printed $(n^2 + n)/2$ times.
This is also expressed as $O(n^2)$.

Trace Table: Integer Multiplication

Trace Table: Integer Multiplication

Algorithm A:

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes			

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13		

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38

Trace Table: Integer Multiplication

<u>Algorithm A</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

<u>Algorithm B</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13		

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13		

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes			

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65		

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	

Trace Table: Integer Multiplication

<u>Algorithm A</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

<u>Algorithm B</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes			

Trace Table: Integer Multiplication

<u>Algorithm A</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

<u>Algorithm B</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169		

Trace Table: Integer Multiplication

<u>Algorithm A</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

<u>Algorithm B</u>:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
    p = p + m
    n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
    if (odd(n))
        {p = p + m}
    m = m * 2
    n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No			

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169		

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
   p = p + m
   n = n - 1
-----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
   if (odd(n))
      {p = p + m}
   m = m * 2
   n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No			

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169		

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes			

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
   p = p + m
   n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
   if (odd(n))
      {p = p + m}
   m = m * 2
   n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001		

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
   p = p + m
   n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
   if (odd(n))
      {p = p + m}
   m = m * 2
   n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	0

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	0
No				

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	0
No	–	–	–	–

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	0
No	–	–	–	–

`p=p+m` executed 77 times by A.

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
   p = p + m
   n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
   if (odd(n))
      {p = p + m}
   m = m * 2
   n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	0
No	–	–	–	–

p=p+m executed 77 times by A.

p=p+m executed 4 times by B.

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	0
No	–	–	–	–

p=p+m executed 77 times by A.
p=p+m executed 4 times by B.
A is of time complexity $O(n)$.

Trace Table: Integer Multiplication

Algorithm A:

```
m = 13; n = 77
p = 0
while (n > 0)
  p = p + m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 0
while (n > 0)
  if (odd(n))
    {p = p + m}
  m = m * 2
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	0	13	77
Yes	Yes	13	26	38
Yes	No	13	52	19
Yes	Yes	65	104	9
Yes	Yes	169	208	4
Yes	No	169	416	2
Yes	No	169	832	1
Yes	Yes	1001	1664	0
No	–	–	–	–

p=p+m executed 77 times by A.
p=p+m executed 4 times by B.
A is of time complexity $O(n)$.
B is of time complexity of $O(\log n)$.

Trace Table: Integer Exponentiation

Trace Table: Integer Exponentiation

Algorithm A:

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes				

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes			

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	

Trace Table: Integer Exponentiation

<u>Algorithm A</u>:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

<u>Algorithm B</u>:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes				

Trace Table: Integer Exponentiation

<u>Algorithm A</u>:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

<u>Algorithm B</u>:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes				

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes			

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes				

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes			

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes				

Trace Table: Integer Exponentiation

<u>Algorithm A</u>:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

<u>Algorithm B</u>:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No			

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes				

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No			

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes				

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes			

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}		

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	0

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	0
No				

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	0
No	–	–	–	–

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	0
No	–	–	–	–

`p=p*m` executed 77 times by A.

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	0
No	–	–	–	–

p=p*m executed 77 times by A.

p=p*m executed 4 times by B.

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
---------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	0
No	–	–	–	–

p=p*m executed 77 times by A.
p=p*m executed 4 times by B.
A is of time complexity $O(n)$.

Trace Table: Integer Exponentiation

Algorithm A:

```
m = 13; n = 77
p = 1
while (n > 0)
  p = p * m
  n = n - 1
----------------
```

Algorithm B:

```
m = 13; n = 77
p = 1
while (n > 0)
  if (odd(n))
    {p = p * m}
  m = m * m
  n = n / 2
```

n>0?	odd(n)	c	m	n
–	–	1	13	77
Yes	Yes	13^1	13^2	38
Yes	No	13^1	13^4	19
Yes	Yes	13^5	13^8	9
Yes	Yes	13^{13}	13^{16}	4
Yes	No	13^{13}	13^{32}	2
Yes	No	13^{13}	13^{64}	1
Yes	Yes	13^{77}	13^{128}	0
No	–	–	–	–

`p=p*m` executed 77 times by A.
`p=p*m` executed 4 times by B.
A is of time complexity $O(n)$.
B is of time complexity of $O(\log n)$.

The GCD Algorithm

The GCD Algorithm

- Computing the *greatest common divisor*

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m) =$ greatest integer d such that $d|n$ and $d|m$.

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m) =$ greatest integer d such that $d|n$ and $d|m$.
- List all the divisors n and all the divisors of m.

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m) =$ greatest integer d such that $d|n$ and $d|m$.
- List all the divisors n and all the divisors of m.
- There may be several common divisors.

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m) =$ greatest integer d such that $d|n$ and $d|m$.
- List all the divisors n and all the divisors of m.
- There may be several common divisors.
- Which is the greatest?

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m) =$ greatest integer d such that $d|n$ and $d|m$.
- List all the divisors n and all the divisors of m.
- There may be several common divisors.
- Which is the greatest?
- For example, $gcd(18, 12) = 6$.

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m) =$ greatest integer d such that $d|n$ and $d|m$.
- List all the divisors n and all the divisors of m.
- There may be several common divisors.
- Which is the greatest?
- For example, $gcd(18, 12) = 6$.
- The divisors of 18 are 1, 2, 3, 6, 9, and 18.

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m)$ = greatest integer d such that $d|n$ and $d|m$.
- List all the divisors n and all the divisors of m.
- There may be several common divisors.
- Which is the greatest?
- For example, $gcd(18, 12) = 6$.
- The divisors of 18 are 1, 2, 3, 6, 9, and 18.
- The divisors of 12 are 1, 2, 3, 6, and 12.

The GCD Algorithm

- Computing the *greatest common divisor*
- $gcd(n, m) =$ greatest integer d such that $d|n$ and $d|m$.
- List all the divisors n and all the divisors of m.
- There may be several common divisors.
- Which is the greatest?
- For example, $gcd(18, 12) = 6$.
- The divisors of 18 are 1, 2, 3, 6, 9, and 18.
- The divisors of 12 are 1, 2, 3, 6, and 12.
- Of the common divisors 1, 2, 3, and 6, 6 is the greatest.

The GCD Algorithm (cont.)

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) =$

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) =$

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18,12) = gcd(12, 18 \mod 12) = gcd(18,6)$.
- $gcd(6, 18 \mod 6) =$

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18,12) = gcd(12, 18 \mod 12) = gcd(18,6)$.
- $gcd(6, 18 \mod 6) = gcd(6,0) =$

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18,12) = gcd(12, 18 \mod 12) = gcd(18,6)$.
- $gcd(6, 18 \mod 6) = gcd(6,0) = 6$.

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242,

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18,12) = gcd(12, 18 \mod 12) = gcd(18,6)$.
- $gcd(6, 18 \mod 6) = gcd(6,0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242, 106,

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18,12) = gcd(12, 18 \mod 12) = gcd(18,6)$.
- $gcd(6, 18 \mod 6) = gcd(6,0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242, 106, 30,

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242, 106, 30, 16,

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n,m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242, 106, 30, 16, 14,

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242, 106, 30, 16, 14, 2,

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242, 106, 30, 16, 14, 2, 0,

The GCD Algorithm (cont.)

- If d divides n and m, d also divides $n \mod m$.

$$gcd(n, m) = \begin{cases} n, & \text{if } m = 0; \\ gcd(m, n \mod m), & \text{if } m \neq 0. \end{cases}$$

- $gcd(18, 12) = gcd(12, 18 \mod 12) = gcd(18, 6)$.
- $gcd(6, 18 \mod 6) = gcd(6, 0) = 6$.
- Remainder sequence: 18, 12, 6, 0.
- For $n = 7366$ and $m = 242$, the remainder sequence is:
- 7366, 242, 106, 30, 16, 14, 2, 0,
- so that $gcd(7366, 242) = 2$.

The GCD Algorithm (cont.)

The GCD Algorithm (cont.)

- The basic operation here is the division.

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1,

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1, 0

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1, 0
- Thus, $gcd(987, 610) = 1$

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1, 0
- Thus, $gcd(987, 610) = 1$
- The quotient is 1 at each step.

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1, 0
- Thus, $gcd(987, 610) = 1$
- The quotient is 1 at each step.
- The remainder sequence is the

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1, 0
- Thus, $gcd(987, 610) = 1$
- The quotient is 1 at each step.
- The remainder sequence is the
  Fibonacci sequence in reverse order.

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1, 0
- Thus, $gcd(987, 610) = 1$
- The quotient is 1 at each step.
- The remainder sequence is the
  Fibonacci sequence in reverse order.
- The number of division steps is k, such that $F_k \approx n$.

The GCD Algorithm (cont.)

- The basic operation here is the division.
- The maximum number of division steps will occur
- when the quotient for each division is no greater than 1.
- Consider $n = 987$, $m = 610$.
- The remainder sequence is:
- 987, 610, 377, 233, 144, 89, 55,
  34, 21, 13, 8, 5, 3, 2, 1, 0
- Thus, $gcd(987, 610) = 1$
- The quotient is 1 at each step.
- The remainder sequence is the
  Fibonacci sequence in reverse order.
- The number of division steps is k, such that $F_k \approx n$.
- Since $F_k \approx (\phi)^k$, where $\phi \approx 1.618$, $k \approx \log_\phi n = O(\log n)$.

Algorithm Analysis: TM for 0^k1^k

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.
Steps 2 and 3 repeatedly scan the tape:

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.
Steps 2 and 3 repeatedly scan the tape:
To cross out one 0 and one 1 $\Rightarrow O(n)$ time per scan.

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.
Steps 2 and 3 repeatedly scan the tape:
To cross out one 0 and one $1 \Rightarrow O(n)$ time per scan.
Each scan reduces the number of symbols in half $\Rightarrow n/2$ scans.

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.
Steps 2 and 3 repeatedly scan the tape:
To cross out one 0 and one 1 $\Rightarrow O(n)$ time per scan.
Each scan reduces the number of symbols in half $\Rightarrow n/2$ scans.
Steps 2 and 3 require $O(n^2)$ time together.

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.
Steps 2 and 3 repeatedly scan the tape:
To cross out one 0 and one 1 $\Rightarrow O(n)$ time per scan.
Each scan reduces the number of symbols in half $\Rightarrow n/2$ scans.
Steps 2 and 3 require $O(n^2)$ time together.
Step 4 decides to accept/reject $\Rightarrow O(n)$ time.

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.
Steps 2 and 3 repeatedly scan the tape:
To cross out one 0 and one 1 $\Rightarrow O(n)$ time per scan.
Each scan reduces the number of symbols in half $\Rightarrow n/2$ scans.
Steps 2 and 3 require $O(n^2)$ time together.
Step 4 decides to accept/reject $\Rightarrow O(n)$ time.
Total time:

Algorithm Analysis: TM for 0^k1^k

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

Step 1 check if input is of the form $0^*1^* \Rightarrow O(n)$ time.
Steps 2 and 3 repeatedly scan the tape:
To cross out one 0 and one 1 $\Rightarrow O(n)$ time per scan.
Each scan reduces the number of symbols in half $\Rightarrow n/2$ scans.
Steps 2 and 3 require $O(n^2)$ time together.
Step 4 decides to accept/reject $\Rightarrow O(n)$ time.
Total time: $O(n) + O(n^2) + O(n) = O(n^2)$.

Complexity Classes

Complexity Classes

- Choice of model affects time complexity.

Complexity Classes

- Choice of model affects time complexity.
- Deterministic vs. Nondeterministic models.

Complexity Classes

- Choice of model affects time complexity.
- Deterministic vs. Nondeterministic models.

Let $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Complexity Classes

- Choice of model affects time complexity.
- Deterministic vs. Nondeterministic models.

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

$$\mathbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

Class P

Class P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

Class P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

Analysis of Pseudocode vs. TM

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding of a corresponding string on a TM.

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding of a corresponding string on a TM.
  - encoding $\Rightarrow$ polynomial in the size of the string

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding of a corresponding string on a TM.
  - encoding $\Rightarrow$ polynomial in the size of the string
- However, unary encoding of integers is not appropriate

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding of a corresponding string on a TM.
  - encoding $\Rightarrow$ polynomial in the size of the string
- However, unary encoding of integers is not appropriate
  - because it is exponential in length

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding of a corresponding string on a TM.
  - encoding $\Rightarrow$ polynomial in the size of the string
- However, unary encoding of integers is not appropriate
  - because it is exponential in length compared to binary

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding of a corresponding string on a TM.
  - encoding $\Rightarrow$ polynomial in the size of the string
- However, unary encoding of integers is not appropriate
  - because it is exponential in length compared to binary
- Pose a computation problem as a

Analysis of Pseudocode vs. TM

- Algorithm analysis based on pseudocode
  - should be considered as analyzing an encoding of a corresponding string on a TM.
  - encoding $\Rightarrow$ polynomial in the size of the string
- However, unary encoding of integers is not appropriate
  - because it is exponential in length compared to binary
- Pose a computation problem as a string acceptance problem

Some Problems in P: $RELPRIME(x, y)$

Some Problems in P: *RELPRIME*(x, y)

- Given two integers x and y,

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.
- x and y are relatively prime

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.
- x and y are relatively prime
  if and only if $gcd(x, y) = 1$.

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.
- x and y are relatively prime
  if and only if $gcd(x, y) = 1$.
- From pevious analysis,

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.
- x and y are relatively prime
  if and only if $gcd(x, y) = 1$.
- From pevious analysis,
  gcd takes $O(\log x)$ time

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.
- x and y are relatively prime
  if and only if $gcd(x, y) = 1$.
- From pevious analysis,
  gcd takes $O(\log x)$ time (assume $x > y$).

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.
- x and y are relatively prime
  if and only if $gcd(x, y) = 1$.
- From pevious analysis,
  gcd takes $O(\log x)$ time (assume $x > y$).
- Binary encoding of x has length $n = \log_2 x$.

Some Problems in P: $RELPRIME(x, y)$

- Given two integers x and y,
  determine whether x and y are relatively prime.
- x and y are relatively prime
  if and only if $gcd(x, y) = 1$.
- From pevious analysis,
  gcd takes $O(\log x)$ time (assume $x > y$).
- Binary encoding of x has length $n = \log_2 x$.
- Thus, $RELPRIME$ takes $O(n)$ time, and thus in P.

Some Problems in P: $PATH(s, t)$

Some Problems in P: $PATH(s, t)$

- Given a directed graph G and two vertices s, t

Some Problems in P: $PATH(s,t)$

- Given a directed graph G and two vertices s,t
  Determine whether there is a directed path from s to t.

Some Problems in P: $PATH(s, t)$

- Given a directed graph G and two vertices s, t

  Determine whether there is a directed path from s to t.

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*.

Some Problems in P: $PATH(s, t)$

- Given a directed graph G and two vertices s, t
  Determine whether there is a directed path from s to t.

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*.

Takes polynomial time in the size of G

Some Problems in P: $PATH(s, t)$

- Given a directed graph G and two vertices s, t
  Determine whether there is a directed path from s to t.

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*.

Takes polynomial time in the size of G

size $= m$, number of edges

- Thus $PATH \in$ P

Some Problems in P: $CONNECTED(G)$

Some Problems in P: $CONNECTED(G)$

- Given an undirected graph G

Some Problems in P: $CONNECTED(G)$

- Given an undirected graph G
  Determine whether it is connected.

Some Problems in P: $CONNECTED(G)$

- Given an undirected graph G

  Determine whether it is connected.

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*.

Some Problems in P: $CONNECTED(G)$

- Given an undirected graph G

  Determine whether it is connected.

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*.

Takes polynomial time in the order of G,

Some Problems in P: $CONNECTED(G)$

- Given an undirected graph G

  Determine whether it is connected.

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*.

Takes polynomial time in the order of G, $O(n^2)$

Some Problems in P: $CONNECTED(G)$

- Given an undirected graph G

  Determine whether it is connected.

  1. Select the first node of G and mark it.
  2. Repeat the following stage until no new nodes are marked:
  3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
  4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*.

  Takes polynomial time in the order of G, $O(n^2)$

  order $= n$, number of vertices

- Thus $CONNECTED \in$ P

Towards Defining Class NP and NP-Complete

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.
- For such problems, it may be true that:

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.
- For such problems, it may be true that:
  - Polynomial time algorithms are yet to be discovered.

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.
- For such problems, it may be true that:
  - Polynomial time algorithms are yet to be discovered.
  - These problems *cannot* be solved in polynomial time.

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.
- For such problems, it may be true that:
  - Polynomial time algorithms are yet to be discovered.
  - These problems *cannot* be solved in polynomial time.
- Important: Complexity of many problems are linked.

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.
- For such problems, it may be true that:
  - Polynomial time algorithms are yet to be discovered.
  - These problems *cannot* be solved in polynomial time.
- Important: Complexity of many problems are linked.
- A class of problems exist such that

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.
- For such problems, it may be true that:
  - Polynomial time algorithms are yet to be discovered.
  - These problems *cannot* be solved in polynomial time.
- Important: Complexity of many problems are linked.
- A class of problems exist such that if a polynomial time algorithm may be devised for one of these problems,

Towards Defining Class NP and NP-Complete

- Polynomial time algorithms avoid brute-force searching.
- Can brute-force searching be avoided for all problems?
- Some problems exist for which:
  - Avoiding brute-force search has not succeeded.
  - No polynomial time algorithms are currently known.
- For such problems, it may be true that:
  - Polynomial time algorithms are yet to be discovered.
  - These problems *cannot* be solved in polynomial time.
- Important: Complexity of many problems are linked.
- A class of problems exist such that if a polynomial time algorithm may be devised for one of these problems, all such problems can be solved in polynomial time.

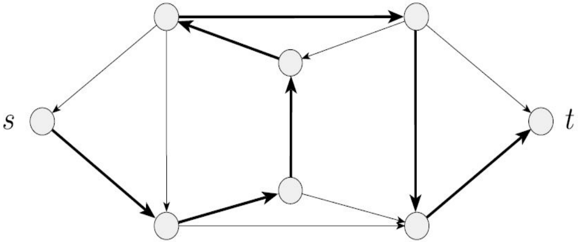
The Hamilton Path Problem

The Hamilton Path Problem

- A Hamilton path in a directed graph is a directed path that goes through each vertex exactly once.

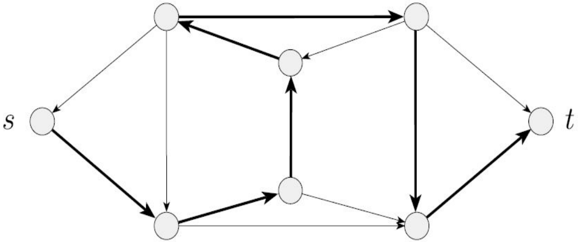
The Hamilton Path Problem

- A Hamilton path in a directed graph is a directed path that goes through each vertex exactly once.



The Hamilton Path Problem

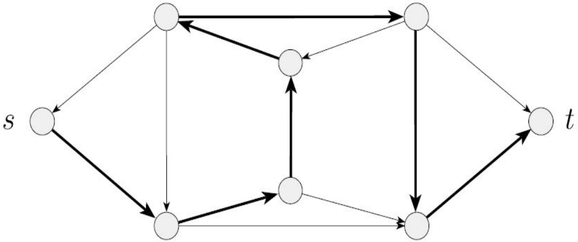
- A Hamilton path in a directed graph is a directed path that goes through each vertex exactly once.



- Given an directed graph G and vertices s and t, does there exist a Hamilton path from s to t?

The Hamilton Path Problem

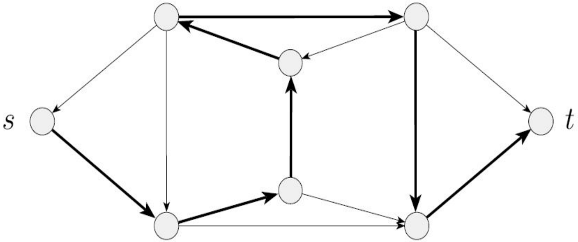
- A Hamilton path in a directed graph is a directed path that goes through each vertex exactly once.



- Given an directed graph G and vertices s and t, does there exist a Hamilton path from s to t?
- This problem is referred to as *HAMPATH*.

The Hamilton Path Problem

- A Hamilton path in a directed graph is a directed path that goes through each vertex exactly once.



- Given an directed graph G and vertices s and t, does there exist a Hamilton path from s to t?
- This problem is referred to as *HAMPATH*.
- It is not known whether a polynomial time algorithm exists for solving this problem.

Polynomial Verifiability: *HAMPATH*

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:

  Verify/Check if a potential path is Hamiltonian.

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:

  Verify/Check if a potential path is Hamiltonian.
- Note that such verification can be done in

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:
  Verify/Check if a potential path is Hamiltonian.
- Note that such verification can be done in
  polynomial time.

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:
  Verify/Check if a potential path is Hamiltonian.
- Note that such verification can be done in
  polynomial time.
- Verifying if a given path is Hamiltonian is much easier than

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:
  Verify/Check if a potential path is Hamiltonian.
- Note that such verification can be done in
  polynomial time.
- Verifying if a given path is Hamiltonian is much easier than
  Determining whether a Hamilton path exists.

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:
  Verify/Check if a potential path is Hamiltonian.
- Note that such verification can be done in
  polynomial time.
- Verifying if a given path is Hamiltonian is much easier than
  Determining whether a Hamilton path exists.
- Thus *HAMPATH* has the propery we will call

Polynomial Verifiability: *HAMPATH*

- *PATH* can be used to devise an algorithm for *HAMPATH*.
- Such an algorithm will take exponential time.
- Add a step that will:
  Verify/Check if a potential path is Hamiltonian.
- Note that such verification can be done in
  polynomial time.
- Verifying if a given path is Hamiltonian is much easier than
  Determining whether a Hamilton path exists.
- Thus *HAMPATH* has the propery we will call
  polynomial verifiability.

Polynomial Verifiability: *COMPOSITES*

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given
  natural number is composite.

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given
  natural number is composite.
- A polynomial time algorithm is known for this problem

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given
  natural number is composite.
- A polynomial time algorithm is known for this problem
  But it is somewhat more complicated than verification.

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given
  natural number is composite.
- A polynomial time algorithm is known for this problem
  But it is somewhat more complicated than verification.
- Given a number x, it is much easier to verify

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given
  natural number is composite.
- A polynomial time algorithm is known for this problem
  But it is somewhat more complicated than verification.
- Given a number x, it is much easier to verify
  if another number is a divisor of x

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given
  natural number is composite.
- A polynomial time algorithm is known for this problem
  But it is somewhat more complicated than verification.
- Given a number x, it is much easier to verify
  if another number is a divisor of x
  than it is to determine the divisors of x.

Polynomial Verifiability: *COMPOSITES*

- A natural number is said be composite
  if it is a product of two integers greater than 1.
- A composite number is not prime.
- *COMPOSITES*:
  Determine whether a given
  natural number is composite.
- A polynomial time algorithm is known for this problem
  But it is somewhat more complicated than verification.
- Given a number x, it is much easier to verify
  if another number is a divisor of x
  than it is to determine the divisors of x.
- *COMPOSITES* is polynomially verifiable.

Classes NP and coNP

Classes NP and coNP

- Not all problems can be verified in polynomial time.

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm
- Problems whose potential solutions can be

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm
- Problems whose potential solutions can be
  verified in polynomial time are said to in class NP.

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm
- Problems whose potential solutions can be
  verified in polynomial time are said to in class NP.
- Another way to state this:

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm
- Problems whose potential solutions can be
  verified in polynomial time are said to in class NP.
- Another way to state this:
  NP is the class of languages

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm
- Problems whose potential solutions can be
  verified in polynomial time are said to in class NP.
- Another way to state this:
  NP is the class of languages
  that have polynomial verifiers.

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm
- Problems whose potential solutions can be
  verified in polynomial time are said to in class NP.
- Another way to state this:
  NP is the class of languages
  that have polynomial verifiers.
- $HAMPATH \in$ NP,

Classes NP and coNP

- Not all problems can be verified in polynomial time.
- For example, consider $\overline{HAMPATH}$
- Even though we may determine
  that a graph is not Hamiltonian
- there is no known method to verify the
  nonexistence of a Hamiltonian path
  without using the exponential time algorithm
- Problems whose potential solutions can be
  verified in polynomial time are said to in class NP.
- Another way to state this:
  NP is the class of languages
  that have polynomial verifiers.
- $HAMPATH \in$ NP, and $\overline{HAMPATH} \in$ coNP.

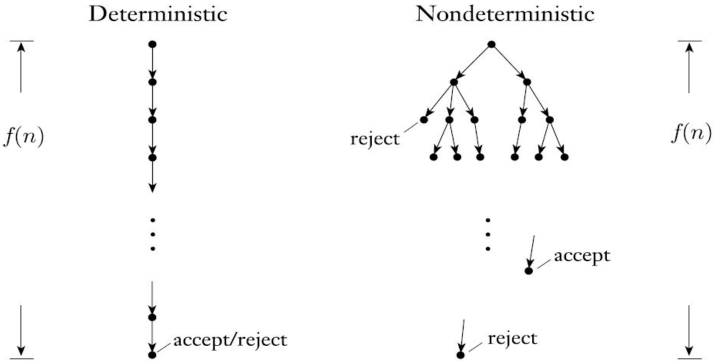
Running Time for Nondeterministic TMs

Running Time for Nondeterministic TMs

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

Running Time for Nondeterministic TMs

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.



Relationship of NP to TMs

Relationship of NP to TMs

$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

Relationship of NP to TMs

$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

$$\text{NP} = \bigcup_k \text{NTIME}(n^k).$$

Relationship of NP to TMs

NTIME(*t*(*n*)) $= \{L \mid L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine$\}$.

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k).$$

- NP comes from nondeterministic polynomial time

Relationship of NP to TMs

$$\textbf{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

$$\text{NP} = \bigcup_k \text{NTIME}(n^k).$$

- NP comes from nondeterministic polynomial time
- A language is in NP if and only if it is

Relationship of NP to TMs

$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

$$\text{NP} = \bigcup_k \text{NTIME}(n^k).$$

- NP comes from nondeterministic polynomial time
- A language is in NP if and only if it is decided by a nondeterministic polynomial time TM.

$HAMPATH \in \mathrm{NP}$

HAMPATH $\in$ NP

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, *reject*.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise, all tests have been passed, so *accept*.

$\mathit{CLIQUE} \in \mathrm{NP}$

$CLIQUE \in$ NP

- A *clique* in an undirected graph is a subgraph in which

CLIQUE $\in$ NP

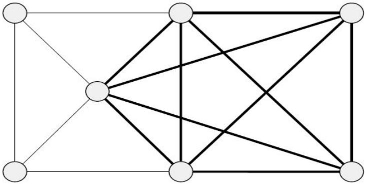
- A *clique* in an undirected graph is a subgraph in which every two nodes are connected by an edge.

CLIQUE $\in$ NP

- A *clique* in an undirected graph is a subgraph in which every two nodes are connected by an edge.
- A *k-clique* is a clique with k nodes.

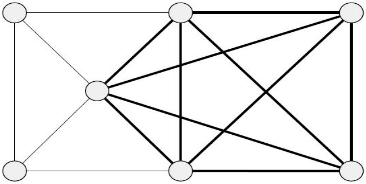
CLIQUE $\in$ NP

- A *clique* in an undirected graph is a subgraph in which every two nodes are connected by an edge.
- A *k-clique* is a clique with k nodes.



$CLIQUE \in$ NP

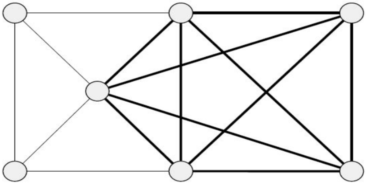
- A *clique* in an undirected graph is a subgraph in which every two nodes are connected by an edge.
- A *k-clique* is a clique with k nodes.



CLIQUE: Given an undirected graph G and an integer k,

CLIQUE $\in$ NP

- A *clique* in an undirected graph is a subgraph in which every two nodes are connected by an edge.
- A *k-clique* is a clique with k nodes.



CLIQUE: Given an undirected graph G and an integer k, Determine whether G contains a k-clique.

$CLIQUE \in$ NP (cont.)

$\mathit{CLIQUE} \in$ NP (cont.)

1. Nondeterministically select 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*.

CLIQUE $\in$ NP (cont.)

1. Nondeterministically select 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*.

- Step 1 is the nondeterministic part.

$CLIQUE \in$ NP (cont.)

1. Nondeterministically select 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*.

   - Step 1 is the nondeterministic part.
   - Step 2 is the polynomial part.

$\mathit{CLIQUE} \in$ NP (cont.)

1. Nondeterministically select 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*.

- Step 1 is the nondeterministic part.
- Step 2 is the polynomial part.

$\overline{\mathit{CLIQUE}} \in$ coNP.

$SUBSET\text{-}SUM \in$ NP

$\mathit{SUBSET\text{-}SUM} \in \mathrm{NP}$

- Given a collection of numbers $x_1, x_2, \cdots, x_k$

$\mathit{SUBSET\text{-}SUM} \in \mathrm{NP}$

- Given a collection of numbers $x_1, x_2, \cdots, x_k$ and a target number t

SUBSET-SUM $\in$ NP

- Given a collection of numbers $x_1, x_2, \cdots, x_k$
  and a target number t
- Determine whether the collection contains

$SUBSET\text{-}SUM \in \mathrm{NP}$

- Given a collection of numbers $x_1, x_2, \cdots, x_k$ and a target number t
- Determine whether the collection contains a sub-collection that adds up to t.

SUBSET-SUM $\in$ NP

- Given a collection of numbers $x_1, x_2, \cdots, x_k$ and a target number t
- Determine whether the collection contains a sub-collection that adds up to t.
- Example:

SUBSET-SUM $\in$ NP

- Given a collection of numbers $x_1, x_2, \cdots, x_k$
  and a target number t
- Determine whether the collection contains
  a sub-collection that adds up to t.
- Example: For $S = \{1, 4, 15, 20, 22, 27\}$ and $t = 41$,

$SUBSET\text{-}SUM \in$ NP

- Given a collection of numbers $x_1, x_2, \cdots, x_k$
  and a target number t
- Determine whether the collection contains
  a sub-collection that adds up to t.
- Example: For $S = \{1, 4, 15, 20, 22, 27\}$ and $t = 41$,
  the sub-collection $\{4, 15, 22\}$ adds up to 41.

$SUBSET\text{-}SUM \in \mathrm{NP}$

- Given a collection of numbers $x_1, x_2, \cdots, x_k$ and a target number t
- Determine whether the collection contains a sub-collection that adds up to t.
- Example: For $S = \{1, 4, 15, 20, 22, 27\}$ and $t = 41$, the sub-collection $\{4, 15, 22\}$ adds up to 41.

1. Nondeterministically 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 passes, *accept*; otherwise, *reject*.

$\textit{SUBSET-SUM} \in \mathrm{NP}$

- Given a collection of numbers $x_1, x_2, \cdots, x_k$
  and a target number t
- Determine whether the collection contains
  a sub-collection that adds up to t.
- Example: For $S = \{1, 4, 15, 20, 22, 27\}$ and $t = 41$,
  the sub-collection $\{4, 15, 22\}$ adds up to 41.

1. Nondeterministically 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 passes, *accept*; otherwise, *reject*.

$\overline{\textit{SUBSET-SUM}} \in \mathrm{coNP}$.

P versus NP and NP Completeness

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

- Cook-Levin discovered in the 1970s

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

- Cook-Levin discovered in the 1970s
- Some Problems in NP whose individual complexity

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

- Cook-Levin discovered in the 1970s
- Some Problems in NP whose individual complexity is related to the complexity of the entire class NP

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

- Cook-Levin discovered in the 1970s
- Some Problems in NP whose individual complexity is related to the complexity of the entire class NP
- These problems are called *NP-complete* problems.

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

- Cook-Levin discovered in the 1970s
- Some Problems in NP whose individual complexity
  is related to the complexity of the entire class NP
- These problems are called *NP-complete* problems.
- If a polynomial algorithm exists

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

- Cook-Levin discovered in the 1970s
- Some Problems in NP whose individual complexity
  is related to the complexity of the entire class NP
- These problems are called *NP-complete* problems.
- If a polynomial algorithm exists
  for any NP-complete problem, then all NP problems

P versus NP and NP Completeness

- P is the class of languages which can be *decided* quickly.
- NP is the class of languages which can be *verified* quickly.

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

- Cook-Levin discovered in the 1970s
- Some Problems in NP whose individual complexity
  is related to the complexity of the entire class NP
- These problems are called *NP-complete* problems.
- If a polynomial algorithm exists
  for any NP-complete problem, then all NP problems
  would be polynomial time solvable.

The 3-SAT Problem

The 3-SAT Problem

- A *literal* is a Boolean variable x,

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $$(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$$
- If every clause in a cnf formula contains

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$
- If every clause in a cnf formula contains
  exactly three literals, it is called a *3-cnf formula*.

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$
- If every clause in a cnf formula contains
  exactly three literals, it is called a *3-cnf formula*.
- A Boolean formula is *satisfiable*,

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$
- If every clause in a cnf formula contains
  exactly three literals, it is called a *3-cnf formula*.
- A Boolean formula is *satisfiable*, if there is a choice of

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$
- If every clause in a cnf formula contains
  exactly three literals, it is called a *3-cnf formula*.
- A Boolean formula is *satisfiable*, if there is a choice of
  values for the variables that make its value equal 1.

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$
- If every clause in a cnf formula contains
  exactly three literals, it is called a *3-cnf formula*.
- A Boolean formula is *satisfiable*, if there is a choice of
  values for the variables that make its value equal 1.
- The 3-SAT Problem: Decide whether

The 3-SAT Problem

- A *literal* is a Boolean variable x,
  or a negated Boolean variable, $\bar{x}$.
- A *clause* is several literals connected using $\vee$s, such as:
  $x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}$
- A Boolean formula in *conjunctive normal form (cnf)*
  contains several clauses connected by $\wedge$s:
  $(x_1 \vee x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee \bar{x_4})$
- If every clause in a cnf formula contains
  exactly three literals, it is called a *3-cnf formula*.
- A Boolean formula is *satisfiable*, if there is a choice of
  values for the variables that make its value equal 1.
- The 3-SAT Problem: Decide whether
  a given 3-cnf formula is satisfiable.

3-SAT is Polynomially Reducible to *CLIQUE*

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.
- Each triple corresponds to a clause

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.
- Each triple corresponds to a clause
- Each node in a triple corresponds to

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.
- Each triple corresponds to a clause
- Each node in a triple corresponds to
  a literal in the associated clause.

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.
- Each triple corresponds to a clause
- Each node in a triple corresponds to
  a literal in the associated clause.
- The edges of G connect all pairs of nodes, except:

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.
- Each triple corresponds to a clause
- Each node in a triple corresponds to
  a literal in the associated clause.
- The edges of G connect all pairs of nodes, except:
  - No edge exists between nodes of the same triple.

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.
- Each triple corresponds to a clause
- Each node in a triple corresponds to
  a literal in the associated clause.
- The edges of G connect all pairs of nodes, except:
  - No edge exists between nodes of the same triple.
  - No edge exists between nodes with

3-SAT is Polynomially Reducible to *CLIQUE*

- Given a 3-cnf formula with k clauses,
  an undirected graph G is constructed as follows:
- The nodes of G are organized into k groups of three nodes
  call them *triples*, $t_1, t_2, \cdots, t_k$.
- Each triple corresponds to a clause
- Each node in a triple corresponds to
  a literal in the associated clause.
- The edges of G connect all pairs of nodes, except:
  - No edge exists between nodes of the same triple.
  - No edge exists between nodes with
    contradictory labels, like x_3 and $\bar{x}_3$.

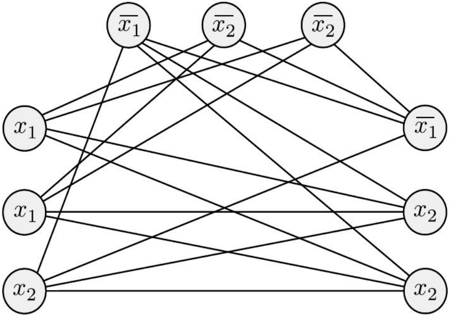
3-SAT to *CLIQUE* (cont.)

3-SAT to *CLIQUE* (cont.)

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

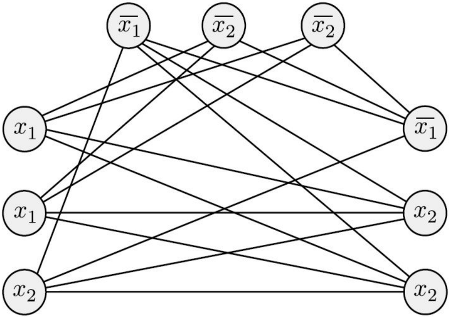
3-SAT to *CLIQUE* (cont.)

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



3-SAT to *CLIQUE* (cont.)

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$



HAMPATH and *SUBSET-SUM* are NP-complete.