CS201 DISCRETE MATHEMATICS FOR COMPUTER SCIENCE

Dr. Qi Wang

Department of Computer Science and Engineering
Office: Room413, CoE South Tower
Email: wangqi@sustech.edu.cn

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.

- Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.

- Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

- Too often, programmers try to slove problems using brute force techniques and end up with slow complicated code!

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.

- Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

- Too often, programmers try to slove problems using brute force techniques and end up with slow complicated code!

- A few hours of abstract thought devoted to algorithm design could have speeded up the solution substantially and simplified it!

Dealing with Hard Problems

- What happens if you can't find an efficient algorithm for a given problem?

Dealing with Hard Problems

- What happens if you can't find an efficient algorithm for a given problem?

Blame yourself.

I couldn't find a polynomial-time algorithm.
I guess I am too dumb.

Dealing with Hard Problems

- What happens if you can't find an efficient algorithm for a given problem?

Show that no-efficient algorithm exists.

I couldn't find a polynomial-time algorithm, because no such algorithm exists.

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

- Proving that no efficient algorithm exists for a particular problem is difficult:

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

- Proving that no efficient algorithm exists for a particular problem is difficult:

  How can we prove the non-existence of something?

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

- Proving that no efficient algorithm exists for a particular problem is difficult:

  How can we prove the non-existence of something?

  We will now learn about NP-Complete problems, which provide us with a way to approach this question.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.

- Researchers have spent innumberable man-years trying to find efficient solutions to these problems but failed.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.

- Researchers have spent innumberable man-years trying to find efficient solutions to these problems but failed.

- So, NP-Complete problems are very likely to be hard.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

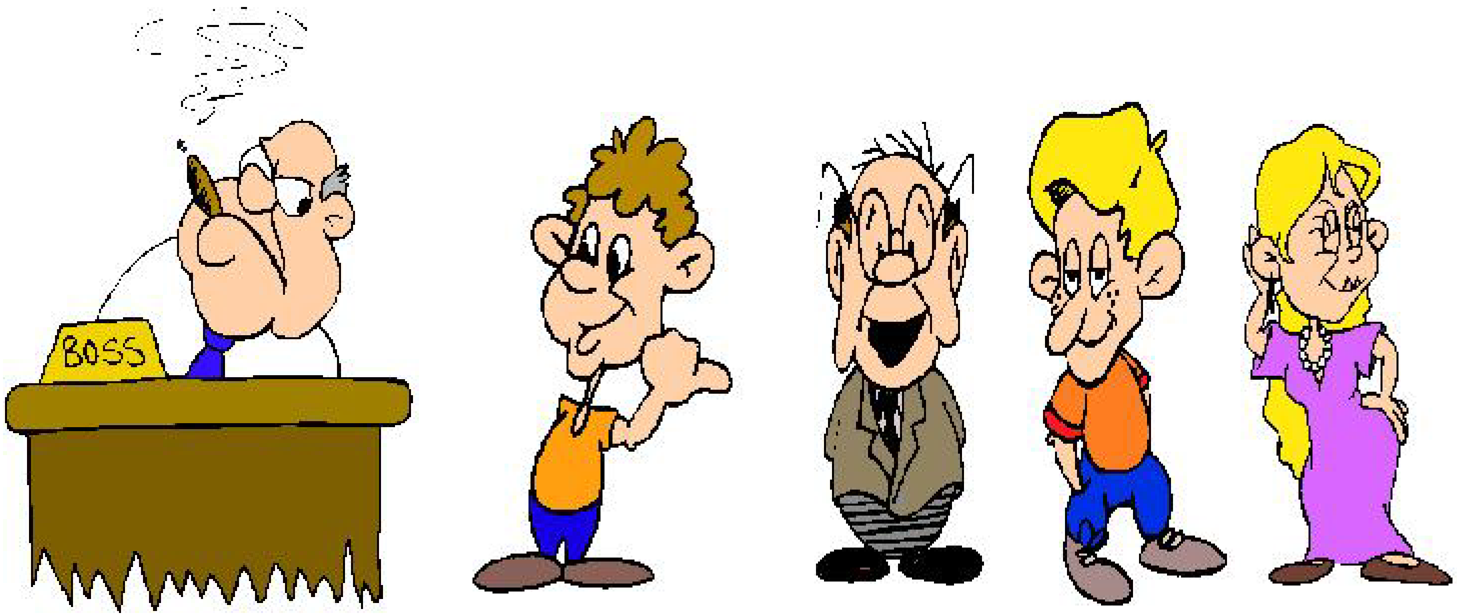
- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.

- Researchers have spent innumberable man-years trying to find efficient solutions to these problems but failed.

- So, NP-Complete problems are very likely to be hard.

- What do you do: prove that your problem is NP-Complete.

What do you actually do:



I couldn't find a polynomial-time algorithm,
but neither could all these other smart people!

Encoding the Inputs of Problems

- Complexity of a problem is measure w.r.t the size of input.

Encoding the Inputs of Problems

- Complexity of a problem is measure w.r.t the size of input.

- In order to formally discuss how hard a problem is, we need to be much more formal than before about the input size of a problem.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

  Definition The *input size* of a problem is the minimum number of bits ($\{0,1\}$) needed to encode the input of the problem.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

  Definition The *input size* of a problem is the minimum number of bits ($\{0,1\}$) needed to encode the input of the problem.

- The exact input size s, determined by an optimal encoding method, is hard to compute in most cases.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

  Definition The *input size* of a problem is the minimum number of bits ($\{0,1\}$) needed to encode the input of the problem.

- The exact input size s, determined by an optimal encoding method, is hard to compute in most cases.

  However, we do not need to determine s exactly.

  For most problems, it is sufficient to choose some natural, and (usually) simple, encoding and use the size s of this encoding.

Input Size Example: Composite

- **Example:**

  Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

Input Size Example: Composite

- **Example:**

  Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

  Question:

  What is the input size of this problem?

- **Example:**

  Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

  Question:

  What is the input size of this problem?

  Any integer $n > 0$ can be represented in the binary number system as a string $a_0 a_1 \cdots a_k$ of length $\lceil \log_2(n+1) \rceil$.

  Thus, a natural measure of input size is $\lceil \log_2(n+1) \rceil$ (or just $\log_2 n$)

Input Size Example: Sorting

- **Example:**

  Sort n integers $a_1, \ldots, a_n$

Input Size Example: Sorting

- **Example:**

  Sort n integers $a_1, \ldots, a_n$

  Question:

  What is the input size of this problem?

Input Size Example: Sorting

- **Example:**

  Sort n integers $a_1, \ldots, a_n$

  Question:

  What is the input size of this problem?

  Using fixed length encoding, we write a_i as a binary string of length $m = \lceil \log_2 \max(|a_i| + 1) \rceil$.

  This coding gives an input size nm.

Complexity in terms of Input Size

- **Example:** (Composite)

  The naive algorithm for determining whether n is composite compares n with the first $n-1$ numbers to see if any of them divides n

Complexity in terms of Input Size

- **Example:** (Composite)

  The naive algorithm for determining whether n is composite compares n with the first $n-1$ numbers to see if any of them divides n

  This makes $\Theta(n)$ comparisons, so it might seem linear and very efficient.

Complexity in terms of Input Size

- **Example:** (Composite)

  The naive algorithm for determining whether n is composite compares n with the first $n-1$ numbers to see if any of them divides n

  This makes $\Theta(n)$ comparisons, so it might seem linear and very efficient.

  But, note that the input size of this problem is $size(n) = \log_2 n$, so the number of comparisons performed is actually $\Theta(n) = \Theta(2^{size(n)})$, which is exponential.

Input Size of Problems

- **Definition** Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large n, where $a_1, b_1, c_1, a_2, b_2, c_2$ are some positive constants.

Input Size of Problems

- **Definition** Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \le f(n) \le c_2 g(n^{a_2})^{b_2}$$

for all large n, where $a_1, b_1, c_1, a_2, b_2, c_2$ are some positive constants.

Example:

All polynomials are of the same type, but *polynomials* and *exponentials* are of different types.

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

  Compute $a \times b$.

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

  Compute $a \times b$.

  Question:

  What is the input size of this problem?

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

  Compute $a \times b$.

  Question:

  What is the input size of this problem?

  The minimum inpute size is

  $$s = \lceil \log_2(a+1) \rceil + \lceil \log_2(b+1) \rceil.$$

  A natural choice is to use $t = \log_2 \max(a, b)$ since $\frac{s}{2} \le t \le s$.

Decision Problems

- **Definition** A *decision problem* is a question that has two possible answers: yes and no.

Decision Problems

- **Definition** A *decision problem* is a question that has two possible answers: yes and no.

  If L is the problem, and x is the input, we will oftern write $x \in L$ to denote a yes answer and $x \notin L$ to denote a no answer.

- **Definition** An *optimization problem* requires an answer that is an optimal configuration.

- **Definition** An *optimization problem* requires an answer that is an optimal configuration.

  An optimization problem usually has a corresponding decision problem.

Optimization Problems

- **Definition** An *optimization problem* requires an answer that is an optimal configuration.

  An optimization problem usually has a corresponding decision problem.

  Examples:

  Knapsack vs. Decision Knapsack (DKnapsack)

Knapsack vs. DKnapsack

- We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, where v_i and w_i are positive integers.

Knapsack vs. DKnapsack

- We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, where v_i and w_i are positive integers.

  Optimization problem: (Knapsack)

  Find the largest value $\sum_{i \in T} v_i$ of any subset T that fits in the knapsack, i.e., $\sum_{i \in T} w_i \leq W$.

  Decision problem: (DKnapsack)

  Given k, is there a subset of the objects that fits in the knapsack and has total value at least k?

Optimization and Decision Problems

- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

Optimization and Decision Problems

- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

  First solve the optimization problem, then check the decision problem. If it does, answer yes, otherwise no.

Optimization and Decision Problems

- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

  First solve the optimization problem, then check the decision problem. If it does, answer yes, otherwise no.

  Thus, if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

Complexity Classes

- The Theory of Complexity deals with

  ◇ the classification of certain "decision problems" into several classes:

    ◇ the class of "easy" problems

    ◇ the class of "hard" problems

    ◇ the class of "hardest" problems

Complexity Classes

- The Theory of Complexity deals with

  ◇ the classification of certain "decision problems" into several classes:

    ◇ the class of "easy" problems

    ◇ the class of "hard" problems

    ◇ the class of "hardest" problems

  ◇ relations among the three classes

Complexity Classes

- The Theory of Complexity deals with
  - ◇ the classification of certain "decision problems" into several classes:
    - ◇ the class of "easy" problems
    - ◇ the class of "hard" problems
    - ◇ the class of "hardest" problems
  - ◇ relations among the three classes
  - ◇ properties of problems in the three classes

Complexity Classes

- The Theory of Complexity deals with

  ◇ the classification of certain "decision problems" into several classes:

    ◇ the class of "easy" problems

    ◇ the class of "hard" problems

    ◇ the class of "hardest" problems

  ◇ relations among the three classes

  ◇ properties of problems in the three classes

Question:
How to classify decision problems?

Complexity Classes

- The Theory of Complexity deals with

    ◇ the classification of certain "decision problems" into several classes:

    - ◇ the class of "easy" problems
    - ◇ the class of "hard" problems
    - ◇ the class of "hardest" problems

    ◇ relations among the three classes

    ◇ properties of problems in the three classes

Question:
How to classify decision problems?

A. Use polynomial-time algorithms.

Polynomial-Time Algorithms

- **Definition** An algorithm is *polynomial-time* if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

Polynomial-Time Algorithms

- **Definition** An algorithm is *polynomial-time* if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

  Whether we use n or n^a (for a fixed $a > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial-time.

Polynomial-Time Algorithms

- **Definition** An algorithm is *polynomial-time* if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

  Whether we use n or n^a (for a fixed $a > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial-time.

  Example:

  The standard multiplication algorithm has time $O(m_1 m_2)$, where m_1, m_2 denote the number of digits in the two integers, respectively.

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

  Let's return to the Composite problem.

  ◇ it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N-1$.

  ◇ The complete algorithm therefore uses $\Theta(N(\log N)^2)$ time.

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

  Let's return to the Composite problem.

  ◇ it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N - 1$.

  ◇ The complete algorithm therefore uses $\Theta(N(\log N)^2)$ time.

  Conclusion: The algorithm is nonpolynomial!

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

  Let's return to the Composite problem.

  ⋄ it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N - 1$.

  ⋄ The complete algorithm therefore uses $\Theta(N(\log N)^2)$ time.

  Conclusion: The algorithm is nonpolynomial!

  Question:
  Why?

Nonpolynomial-Time Algorithms

- **Definition** An algorithm is *nonpolynomial-time* if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Let's return to the Composite problem.

$\diamond$ it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N-1$.

$\diamond$ The complete algorithm therefore uses $\Theta(N(\log N)^2)$ time.

Conclusion: The algorithm is nonpolynomial!

Question:
Why?

In terms of the input size, the complexity is $\Theta(2^n n^2)$.

Polynomial- vs. Nonpolynomial-Time

- Nonpolynomial-time algorithms are impractical.

  2^n for $n = 100$: it takes billions of years!!!

Polynomial- vs. Nonpolynomial-Time

- Nonpolynomial-time algorithms are impractical.

  2^n for $n = 100$: it takes billions of years!!!

  In reality, an $O(n^{20})$ algorithm is not really practical.

Polynomial-Time Solvable Problems

- **Definition** A problem is *solvable in polynomial time* (or more simply, the problem is *in polynomial time*) if there exists an algorithm which solves the problem in polynomial time (a.k.a. *tractable*).

Polynomial-Time Solvable Problems

- **Definition** A problem is *solvable in polynomial time* (or more simply, the problem is *in polynomial time*) if there exists an algorithm which solves the problem in polynomial time (a.k.a. *tractable*).

  Definition (The Class P) The class P consists of all decision problems that are solvable in polynomial time. That is, there exists an algorithm that will decide in polynomial time if any given input is a yes-input or a no-input.

The Class P

- **Question:**
  How to prove that a decision problem is in P?

The Class P

- **Question:**
  How to prove that a decision problem is in P?

  A. Find a polynomial-time algorithm.

The Class P

- **Question:**
  How to prove that a decision problem is in P?

  A. Find a polynomial-time algorithm.

  Question:
  How to prove that a decision problem is not in P?

- **Question:**
  How to prove that a decision problem is in P?

  A. Find a polynomial-time algorithm.

  Question:
  How to prove that a decision problem is not in P?

  A. You need to prove that there is no polynomial-time algorithm for this problem. (much much harder)

Certificates and Verifying Certificates

- **Observation:** A decision problem is usually formulated as:

  Is there an object satisfying some conditions?

Certificates and Verifying Certificates

- **Observation:** A decision problem is usually formulated as:

  Is there an object satisfying some conditions?

Certificates and Verifying Certificates

- A *certificate* is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

Certificates and Verifying Certificates

- A *certificate* is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

  Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a yes-input.

The Class NP

- **Definition** The class NP consists of all decision problems such that, for each yes-input, there exists a *certificate* which allows one to verify in polynomial time that the input is indeed a yes-input.

The Class NP

- **Definition** The class NP consists of all decision problems such that, for each yes-input, there exists a *certificate* which allows one to verify in polynomial time that the input is indeed a yes-input.

  NP – “nondeterministic polynomial-time”

Composite $\in$ NP

- For Composite, a yes-input is just the integer n that is composite.

Composite $\in$ NP

- For Composite, a yes-input is just the integer n that is composite.

  Question: (Certificate)
  What is needed to show n is actually a yes-input?

Composite $\in$ NP

- For Composite, a yes-input is just the integer n that is composite.

  Question: (Certificate)
  What is needed to show n is actually a yes-input?

  A. An integer a $(1 < a < n)$ with the property that $a|n$.

Composite $\in$ NP

- For Composite, a yes-input is just the integer n that is composite.

  Question: (Certificate)
  What is needed to show n is actually a yes-input?

  A. An integer a $(1 < a < n)$ with the property that $a|n$.

  - ◇ Given a certificate a, check whether a divides n.
  - ◇ This can be done in $O((\log n)^2)$.
  - ◇ Composite $\in$ NP

Composite $\in$ NP

- For Composite, a yes-input is just the integer n that is composite.

  Question: (Certificate)
  What is needed to show n is actually a yes-input?

  A. An integer a $(1 < a < n)$ with the property that $a|n$.

  - Given a certificate a, check whether a divides n.
  - This can be done in $O((\log n)^2)$.
  - Composite $\in$ NP

  DKnapsack $\in$ NP

$\text{P} = \text{NP}$?

- One of the most important problems in CS is whether $\text{P} = \text{NP}$ or $\text{P} \neq \text{NP}$?

$\mathrm{P} = \mathrm{NP}$?

- One of the most important problems in CS is whether $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$?
- Observe that $\mathrm{P} \subseteq \mathrm{NP}$.

$\mathsf{P} = \mathsf{NP}$?

- One of the most important problems in CS is whether $\mathsf{P} = \mathsf{NP}$ or $\mathsf{P} \neq \mathsf{NP}$?
- Observe that $\mathsf{P} \subseteq \mathsf{NP}$.
- Intuitively, $\mathsf{NP} \subseteq \mathsf{P}$ is doubtful.

P = NP?

- One of the most important problems in CS is whether $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$?
- Observe that $\mathrm{P} \subseteq \mathrm{NP}$.
- Intuitively, $\mathrm{NP} \subseteq \mathrm{P}$ is doubtful.

  Just being able to verify a certificate in polynomial time does not necessarily mean we can tell whether an input is a yes-input or a no-input in polynomial time.

$\mathsf{P} = \mathsf{NP}$?

- One of the most important problems in CS is whether $\mathsf{P} = \mathsf{NP}$ or $\mathsf{P} \neq \mathsf{NP}$?
- Observe that $\mathsf{P} \subseteq \mathsf{NP}$.
- Intuitively, $\mathsf{NP} \subseteq \mathsf{P}$ is doubtful.

  Just being able to verify a certificate in polynomial time does not necessarily mean we can tell whether an input is a yes-input or a no-input in polynomial time.

  However, we are still no closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into what distinguishes an "easy" problem from a "hard" one.

Application of Number Theory

- G. H. Hardy (1877 - 1947)

  In his 1940 autobiography *A Mathematician's Apology*, Hardy wrote

  "The great modern achievements of applied mathematics have been in relativity and quantum mechanics, and these subjects are, at present, **almost as 'useless' as the theory of numbers**."

Application of Number Theory

- G. H. Hardy (1877 - 1947)

  In his 1940 autobiography *A Mathematician's Apology*, Hardy wrote

  "The great modern achievements of applied mathematics have been in relativity and quantum mechanics, and these subjects are, at present, **almost as 'useless' as the theory of numbers**."

If he could see the world now, Hardy would be spinning in his grave.

Number Theory

- *Number theory* is a branch of mathematics that explores integers and their properties, is the basis of **cryptography**, **coding theory**, **computer security**, **e-commerce**, etc.

Number Theory

- *Number theory* is a branch of mathematics that explores integers and their properties, is the basis of **cryptography**, **coding theory**, **computer security**, **e-commerce**, etc.

- At one point, the largest employer of mathematicians in the United States, and probably the world, was the National Security Agency (NSA). The NSA is the largest spy agency in the US (bigger than CIA, Central Intelligence Agency), and has the responsibility for code design and breaking.

Division

- If a and b are integers with $a \neq 0$, we say that a *divides* b if there is an integer c such that $b = ac$, or equivalently b/a is an integer. In this case, we say that a is a *factor* or *divisor* of b, and b is a *multiple* of a. (We use the notations $a|b$, $a \nmid b$)

Division

- If a and b are integers with $a \neq 0$, we say that a *divides* b if there is an integer c such that $b = ac$, or equivalently b/a is an integer. In this case, we say that a is a *factor* or *divisor* of b, and b is a *multiple* of a. (We use the notations $a|b$, $a \nmid b$)

Example

- $4 \mid 24$
- $3 \nmid 7$

Divisibility

- **All integers divisible by** $d > 0$ can be enumerated as:

$$\ldots, -kd, \ldots, -2d, -d, 0, d, 2d, \ldots, kd, \ldots$$

Divisibility

- **All integers divisible by** $d > 0$ can be enumerated as:

$$\ldots, -kd, \ldots, -2d, -d, 0, d, 2d, \ldots, kd, \ldots$$

- **Question:** Let n and d be two positive integers. How many positive integers not exceeding n are divisible by d?

Divisibility

- **All integers divisible by** $d > 0$ can be enumerated as:

$$\ldots, -kd, \ldots, -2d, -d, 0, d, 2d, \ldots, kd, \ldots$$

- **Question:** Let n and d be two positive integers. How many positive integers not exceeding n are divisible by d?

Answer: Count the number of integers such that $0 < kd \leq n$. Therefore, there are $\lfloor n/d \rfloor$ such positive integers.

Divisibility

- **Properties**

Let a, b, c be integers. Then the following hold:

(i) if $a|b$ and $a|c$, then $a|(b+c)$

(ii) if $a|b$ then $a|bc$ for all integers c

iii) if $a|b$ and $b|c$, then $a|c$

Divisibility

- **Properties**

Let a, b, c be integers. Then the following hold:

(i) if $a|b$ and $a|c$, then $a|(b+c)$

(ii) if $a|b$ then $a|bc$ for all integers c

(iii) if $a|b$ and $b|c$, then $a|c$

Proof.

- **Corollary** If a, b, c are integers, where $a \neq 0$, such that $a|b$ and $a|c$, then $a|(mb + nc)$ whenever m and n are integers.

Divisibility

- **Corollary** If a, b, c are integers, where $a \neq 0$, such that $a|b$ and $a|c$, then $a|(mb+nc)$ whenever m and n are integers.

  Proof. By part (ii) and part (i) of Properties.

The Division Algorithm

- If a is an integer and d a positive integer, then there are **unique** integers q and r, with $0 \leq r < d$, such that $a = dq + r$. In this case, d is called the *divisor*, a is called the *dividend*, q is called the *quotient*, and r is called the *remainder*.

The Division Algorithm

- If a is an integer and d a positive integer, then there are **unique** integers q and r, with $0 \leq r < d$, such that $a = dq + r$. In this case, d is called the *divisor*, a is called the *dividend*, q is called the *quotient*, and r is called the *remainder*.

In this case, we use the notations $q = a\ div\ d$ and $r = a \bmod d$.

Congruence Relation

- If a and b are integers and m is a positive integer, then a is *congruent to b modulo m if m divides $a - b$*, denoted by $a \equiv b \pmod{m}$. This is called *congruence* and m is its *modulus*.

Congruence Relation

- If a and b are integers and m is a positive integer, then a is *congruent to b modulo m if m divides $a - b$*, denoted by $a \equiv b \pmod{m}$. This is called *congruence* and m is its *modulus*.

Example

- ◇ $15 \equiv 3 \pmod{6}$
- ◇ $-1 \equiv 11 \pmod{6}$

More on Congruences

- Let m be a positive integer. The integers a and b are congruent modulo m if and only if there is an integer k such that $a = b + km$.

More on Congruences

- Let m be a positive integer. The integers a and b are congruent modulo m if and only if there is an integer k such that $a = b + km$.

Proof.

"only if" part

"if" part

$\pmod{m}$ and $\bmod m$ Notations

- $a \equiv b \pmod{m}$ and $a \bmod m = b$ are different.
  - $a \equiv b \pmod{m}$ is a relation on the set of integers
  - In $a \bmod m = b$, the notation mod denotes a function

$(\mathrm{mod}\ m)$ and $\mathrm{mod}\, m$ Notations

- $a \equiv b \pmod{m}$ and $a \bmod m = b$ are different.
  - $a \equiv b \pmod{m}$ is a relation on the set of integers
  - In $a \bmod m = b$, the notation mod denotes a function

- Let a and b be integers, and let m be a positive integer. Then $a \equiv b \bmod m$ if and only if $a \bmod m = b \bmod m$

$\pmod{m}$ and $\bmod m$ Notations

- $a \equiv b \pmod{m}$ and $a \bmod m = b$ are different.
  - ◇ $a \equiv b \pmod{m}$ is a relation on the set of integers
  - ◇ In $a \bmod m = b$, the notation mod denotes a function

- Let a and b be integers, and let m be a positive integer. Then $a \equiv b \bmod m$ if and only if $a \bmod m = b \bmod m$

  Proof.

Congruences of Sums and Products

- Let m be a positive integer. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a + c \equiv b + d \pmod{m}$ and $ac \equiv bd \pmod{m}$

Congruences of Sums and Products

- Let m be a positive integer. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a + c \equiv b + d \pmod{m}$ and $ac \equiv bd \pmod{m}$

  Proof.

Algebraic Manipulation of Congruences

- If $a \equiv b \bmod m$, then
  - $c \cdot a \equiv c \cdot b \pmod{m}$?
  - $c + a \equiv c + b \pmod{m}$?
  - $a/c \equiv b/c \pmod{m}$?

Algebraic Manipulation of Congruences

- If $a \equiv b \bmod m$, then
  - $c \cdot a \equiv c \cdot b \pmod{m}$?
  - $c + a \equiv c + b \pmod{m}$?
  - $a/c \equiv b/c \pmod{m}$?

$14 \equiv 8 \pmod{6}$ but $7 \not\equiv 4 \pmod{6}$

Computing the mod Function

- **Corollary** Let m be a positive integer and let a and b be integers. Then

$$(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$
$$ab \bmod m = ((a \bmod m)(b \bmod m)) \bmod m$$

Computing the mod Function

- **Corollary** Let m be a positive integer and let a and b be integers. Then

$$(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$
$$ab \bmod m = ((a \bmod m)(b \bmod m)) \bmod m$$

Proof.

Arithmetic Modulo m

- Let $\mathbf{Z}_m$ be the set of nonnegative integers less than m: $\{0, 1, \ldots, m-1\}$.

Arithmetic Modulo m

- Let $\mathbf{Z}_m$ be the set of nonnegative integers less than m: $\{0, 1, \ldots, m-1\}$.

$$+_m : \; a +_m b = (a+b) \bmod m$$

$$\cdot_m : \; a \cdot_m b = ab \bmod m$$

Arithmetic Modulo m

- Let $\mathbf{Z}_m$ be the set of nonnegative integers less than m: $\{0, 1, \ldots, m-1\}$.

$$+_m : \; a +_m b = (a+b) \bmod m$$

$$\cdot_m : \; a \cdot_m b = ab \bmod m$$

Example

◇ $7 +_{11} 9 = ?$

◇ $7 \cdot_{11} 9 = ?$

Arithmetic Modulo m

- **Closure**: if $a, b \in \mathbf{Z}_m$, then $a +_m b,\ a \cdot_m b \in \mathbf{Z}_m$

Arithmetic Modulo m

- **Closure**: if $a, b \in \mathbf{Z}_m$, then $a +_m b,\ a \cdot_m b \in \mathbf{Z}_m$
- **Associativity**: if $a, b, c \in \mathbf{Z}_m$, then $(a +_m b) +_m c = a +_m (b +_m c)$ and $(a \cdot_m b) \cdot_m c = a \cdot_m (b \cdot_m c)$

Arithmetic Modulo m

- **Closure**: if $a, b \in \mathbf{Z}_m$, then $a +_m b,\ a \cdot_m b \in \mathbf{Z}_m$
- **Associativity**: if $a, b, c \in \mathbf{Z}_m$, then
  $(a +_m b) +_m c = a +_m (b +_m c)$ and
  $(a \cdot_m b) \cdot_m c = a \cdot_m (b \cdot_m c)$
- **Identity elements**: $a +_m 0 = a$ and $a \cdot_m 1 = a$

Arithmetic Modulo m

- **Closure**: if $a, b \in \mathbf{Z}_m$, then $a +_m b,\ a \cdot_m b \in \mathbf{Z}_m$
- **Associativity**: if $a, b, c \in \mathbf{Z}_m$, then $(a +_m b) +_m c = a +_m (b +_m c)$ and $(a \cdot_m b) \cdot_m c = a \cdot_m (b \cdot_m c)$
- **Identity elements**: $a +_m 0 = a$ and $a \cdot_m 1 = a$
- **Additive inverses**: if $a \neq 0$ and $a \in \mathbf{Z}_m$, then $m - a$ is an additive inverse of a modulo m

Arithmetic Modulo m

- **Closure**: if $a, b \in \mathbf{Z}_m$, then $a +_m b,\ a \cdot_m b \in \mathbf{Z}_m$
- **Associativity**: if $a, b, c \in \mathbf{Z}_m$, then $(a +_m b) +_m c = a +_m (b +_m c)$ and $(a \cdot_m b) \cdot_m c = a \cdot_m (b \cdot_m c)$
- **Identity elements**: $a +_m 0 = a$ and $a \cdot_m 1 = a$
- **Additive inverses**: if $a \neq 0$ and $a \in \mathbf{Z}_m$, then $m - a$ is an additive inverse of a modulo m
- **Commutativity**: if $a, b \in \mathbf{Z}_m$, then $a +_m b = b +_m a$

Arithmetic Modulo m

- **Closure**: if $a, b \in \mathbf{Z}_m$, then $a +_m b,\ a \cdot_m b \in \mathbf{Z}_m$
- **Associativity**: if $a, b, c \in \mathbf{Z}_m$, then
  $(a +_m b) +_m c = a +_m (b +_m c)$ and
  $(a \cdot_m b) \cdot_m c = a \cdot_m (b \cdot_m c)$
- **Identity elements**: $a +_m 0 = a$ and $a \cdot_m 1 = a$
- **Additive inverses**: if $a \neq 0$ and $a \in \mathbf{Z}_m$, then $m - a$ is an additive inverse of a modulo m
- **Commutativity**: if $a, b \in \mathbf{Z}_m$, then $a +_m b = b +_m a$
- **Distributivity**: if $a, b, c \in \mathbf{Z}_m$, then
  $a \cdot_m (b +_m c) = (a \cdot_m b) +_m (a \cdot_m c)$ and
  $(a +_m b) \cdot_m c = (a \cdot_m c) +_m (b \cdot_m c)$

Representations of Integers

- We may use *decimal* (*base* 10) or *binary* or *octal* or *hexadecimal* or other notations to represent integers.

Representations of Integers

- We may use *decimal* (*base* 10) or *binary* or *octal* or *hexadecimal* or other notations to represent integers.

- Let $b > 1$ be an integer. Then if n is a positive integer, it can be expressed uniquely in the form $n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b + a_0$, where k is nonnegative, a_i's are nonnegative integers less than b. The representation of n is called *the base-b expansion of n* and is denoted by $(a_k a_{k-1} \ldots a_1 a_0)_b$.

Base-b Expansions

- To get the decimal expansion is easy.

- To get the decimal expansion is easy.

Example

- $\diamond$ $(101011111)_2 = 2^8 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 351$
- $\diamond$ $(7016)_8 = 7 \cdot 8^3 + 1 \cdot 8 + 6 = 3598$

- To get the decimal expansion is easy.

Example

◇ $(101011111)_2 = 2^8 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 351$

◇ $(7016)_8 = 7 \cdot 8^3 + 1 \cdot 8 + 6 = 3598$

- Conversions between binary, octal, hexadecimal expansions are easy.

Base-b Expansions

- To get the decimal expansion is easy.

Example

◇ $(101011111)_2 = 2^8 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 351$

◇ $(7016)_8 = 7 \cdot 8^3 + 1 \cdot 8 + 6 = 3598$

- Conversions between binary, octal, hexadecimal expansions are easy.

Example

◇ $(101011111)_2 = (\underline{101}\overline{011}\underline{111}) = (537)_8$

◇ $(7016)_8 = (\underline{111}\overline{000}\underline{001}\overline{110})_2$
$= (\underline{1110}\overline{0000}\underline{1110})_2 = (E0E)_{16}$

Base-b Expansions

$$\begin{aligned} n &= a_k b^k + a_{k-1} b^{k-1} + a_{k-2} b^{k-2} + \cdots + a_2 b^2 + a_1 b + a_0 \\ &= b(a_k b^{k-1} + a_{k-1} b^{k-2} + a_{k-2} b^{k-3} + \cdots + a_2 b + a_1) + a_0 \\ &= b(b(a_k b^{k-2} + a_{k-1} b^{k-3} + a_{k-2} b^{k-4} + \cdots + a_2) + a_1) + a_0 \\ &= \cdots \end{aligned}$$

Base-b Expansions

$$\begin{aligned} n &= a_k b^k + a_{k-1} b^{k-1} + a_{k-2} b^{k-2} + \cdots + a_2 b^2 + a_1 b + a_0 \\ &= b(a_k b^{k-1} + a_{k-1} b^{k-2} + a_{k-2} b^{k-3} + \cdots + a_2 b + a_1) + a_0 \\ &= b(b(a_k b^{k-2} + a_{k-1} b^{k-3} + a_{k-2} b^{k-4} + \cdots + a_2) + a_1) + a_0 \\ &= \cdots \end{aligned}$$

To construct the base-b expansion of an integer n,

- Divide n by b to obtain $n = bq_0 + a_0$, with $0 \le a_0 < b$
- The remainder a_0 is the rightmost digit in the base-b expansion of n. Then divide q_0 by b to get $q_0 = bq_1 + a_1$ with $0 \le a_1 < b$
- a_1 is the second digit from the right. Continue by successively dividing the quotients by b until **the quotient is 0**

Algorithm: Constructing Base-b Expansions

procedure *base b expansion*(n, b: positive integers with $b > 1$)
$q := n$
$k := 0$
while $(q \neq 0)$
    $a_k := q$ **mod** b
    $q := q$ **div** b
    $k := k + 1$
return$(a_{k-1}, \ldots, a_1, a_0)$ $\{(a_{k-1} \ldots a_1 a_0)_b$ is base b expansion of $n\}$

Example

- $(12345)_{10} = (30071)_8$

Example

- $(12345)_{10} = (30071)_8$

$$12345 = 8 \cdot 1543 + 1$$

$$1543 = 8 \cdot 192 + 7$$

$$192 = 8 \cdot 24 + 0$$

$$24 = 8 \cdot 3 + 0$$

$$3 = 8 \cdot 0 + 3$$

Binary Addition of Integers

$a = (a_{n-1}a_{n-2}\dots a_1a_0)$, $b = (b_{n-1}b_{n-2}\dots b_1b_0)$

procedure *add*(a, b: positive integers)
{the binary expansions of a and b are $(a_{n-1},a_{n-2},\dots,a_0)_2$ and $(b_{n-1},b_{n-2},\dots,b_0)_2$, respectively}
$c := 0$
for $j := 0$ to $n-1$
    $d := \lfloor (a_j + b_j + c)/2 \rfloor$
    $s_j := a_j + b_j + c - 2d$
    $c := d$
$s_n := c$
return$(s_0, s_1, \dots, s_n)$ {the binary expansion of the sum is $(s_n, s_{n-1}, \dots, s_0)_2$}

Binary Addition of Integers

$a = (a_{n-1}a_{n-2}\ldots a_1a_0)$, $b = (b_{n-1}b_{n-2}\ldots b_1b_0)$

procedure *add*(a, b: positive integers)
{the binary expansions of a and b are $(a_{n-1},a_{n-2},\ldots,a_0)_2$ and $(b_{n-1},b_{n-2},\ldots,b_0)_2$, respectively}
$c := 0$
for $j := 0$ to $n-1$
    $d := \lfloor (a_j + b_j + c)/2 \rfloor$
    $s_j := a_j + b_j + c - 2d$
    $c := d$
$s_n := c$
return$(s_0,s_1,\ldots, s_n)$ {the binary expansion of the sum is $(s_n,s_{n-1},\ldots,s_0)_2$}

$O(n)$ bit additions

Algorithm: Binary Multiplication of Integers

$$a = (a_{n-1}a_{n-2}\ldots a_1 a_0)_2,\ b = (b_{n-1}b_{n-2}\ldots b_1 b_0)_2$$

$$\begin{aligned} ab &= a(b_0 2^0 + b_1 2^1 + \cdots + b_{n-1}2^{n-1}) \\ &= a(b_0 2^0) + a(b_1 2^1) + \cdots + a(b_{n-1}2^{n-1}) \end{aligned}$$

procedure *multiply*(a, b: positive integers)
{the binary expansions of a and b are $(a_{n-1}, a_{n-2}, \ldots, a_0)_2$ and $(b_{n-1}, b_{n-2}, \ldots, b_0)_2$, respectively}
for $j := 0$ to $n - 1$
    if $b_j = 1$ **then** $c_j = a$ shifted j places
    else $c_j := 0$
$\{c_0, c_1, \ldots, c_{n-1}$ are the partial products$\}$
$p := 0$
for $j := 0$ to $n - 1$
  $p := p + c_j$
return p {p is the value of ab}

Algorithm: Binary Multiplication of Integers

$a = (a_{n-1} a_{n-2} \dots a_1 a_0)_2, \; b = (b_{n-1} b_{n-2} \dots b_1 b_0)_2$

$$\begin{aligned} ab &= a(b_0 2^0 + b_1 2^1 + \cdots + b_{n-1} 2^{n-1}) \\ &= a(b_0 2^0) + a(b_1 2^1) + \cdots + a(b_{n-1} 2^{n-1}) \end{aligned}$$

procedure *multiply*(a, b: positive integers)
{the binary expansions of a and b are $(a_{n-1}, a_{n-2}, \dots, a_0)_2$ and $(b_{n-1}, b_{n-2}, \dots, b_0)_2$, respectively}
for $j := 0$ to $n - 1$
    if $b_j = 1$ **then** $c_j = a$ shifted j places
    else $c_j := 0$
{$c_0, c_1, \dots, c_{n-1}$ are the partial products}
$p := 0$
for $j := 0$ to $n - 1$
  $p := p + c_j$
return p {p is the value of ab}

$O(n^2)$ shifts and $O(n^2)$ bit additions

Algorithm: Computing div and mod

procedure *division algorithm* (a: integer, d: positive integer)
$q := 0$
$r := |a|$
while $r \geq d$
    $r := r - d$
    $q := q + 1$
if $a < 0$ and $r > 0$ **then**
    $r := d - r$
    $q := -(q+1)$
return (q, r) {$q = a$ **div** d *is the quotient*, $r = a$ **mod** d *is the remainder* }

Algorithm: Computing div and mod

```
procedure division algorithm (a: integer, d: positive integer)
 q := 0
 r := |a|
while r ≥ d
      r := r − d
      q := q + 1
if  a < 0 and r > 0 then
      r := d - r
      q := -(q+1)
return (q, r) {q = a div d is the quotient, r = a mod d is the
remainder }
```

$O(q \log a)$ bit operations. But there exist more efficient algorithms with complextiy $O(n^2)$, where $n = \max(\log a, \log d)$

Algorithm: Binary Modular Exponentiation

$$b^n = b^{a_{k-1}\cdot 2^{k-1}+\cdots+a_1\cdot 2+a_0} = b^{a_{k-1}\cdot 2^{k-1}} \cdots b^{a_1\cdot 2} \cdot b^{a_0}$$

Successively finds $b \bmod m$, $b^2 \bmod m$, $b^4 \bmod m$, $\ldots$, $b^{2^{k-1}} \bmod m$, and multiplies together the terms b^{2^j} where $a_j = 1$.

procedure *modular exponentiation*(b: integer, $n = (a_{k-1}a_{k-2}\ldots a_1a_0)_2$, m: positive integers)
$x := 1$
$power := b$ **mod** m
for $i := 0$ to $k - 1$
  if $a_i = 1$ **then** $x := (x \cdot power)$ **mod** m
  $power := (power \cdot power)$ **mod** m
return x {x equals b^n **mod** m }

Algorithm: Binary Modular Exponentiation

$$b^n = b^{a_{k-1}\cdot 2^{k-1}+\cdots+a_1\cdot 2+a_0} = b^{a_{k-1}\cdot 2^{k-1}} \cdots b^{a_1\cdot 2} \cdot b^{a_0}$$

Successively finds $b \bmod m$, $b^2 \bmod m$, $b^4 \bmod m$, $\ldots$, $b^{2^{k-1}} \bmod m$, and multiplies together the terms b^{2^j} where $a_j = 1$.

procedure *modular exponentiation*(b: integer, $n = (a_{k-1}a_{k-2}\ldots a_1a_0)_2$, m: positive integers)
$x := 1$
$power := b$ **mod** m
for $i := 0$ to $k - 1$
  if $a_i = 1$ **then** $x := (x \cdot power)$ **mod** m
  $power := (power \cdot power)$ **mod** m
return x {x equals b^n **mod** m }

$O((\log m)^2 \log n)$ bit operations

Next Lecture

- number theory, cryptography ...