Computational Complexity
计算复杂性

张方国

中山大学计算机学院

isszhfg@mail.sysu.edu.cn

Lecture 1. Introduction and Preliminaries

- Introduce to Complexity Theory
- History of Complexity Theory
- About This Course
- Preliminaries
  - Notations and Representation
  - Computational Tasks

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

This is a major unsolved problem in computer science.

Informally, it asks whether every problem whose solution can be efficiently checked by a computer can also be efficiently solved by a computer(or If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem?).

It was introduced in 1971 by Stephen Cook in his paper "The complexity of theorem proving procedures".

It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute.

Clay Mathematics Institute Millennium Prize Problems(千禧年大奖难题):

- P versus NP
- The Hodge conjecture
- The Poincar é conjecture - solved, by Grigori Perelman
- The Riemann hypothesis
- Yang – Mills existence and mass gap
- Navier – Stokes existence and smoothness
- The Birch and Swinnerton-Dyer conjecture.

P问题是可以在多项式时间内求解的问题集合；

NP问题是指其解可以在多项式时间内被验证的问题集合。

如果P=NP，那就意味着能够有效验证就可以有效求解（所有的密码体制都容易破解!）。

2002，有70位数学家和计算机科学家被邀请参与P是否等于NP的投票，结果有61人认为P不等于NP。

惠普研究实验室的研究员Vinay Deolalikar声称证明了$P \neq NP$，并与2010年8月6日提交了关于论证该问题的论文草稿。

佐治亚理工学院Richard Lipton教授召集成立一个由国际顶级专家组成的group对Deolalikar的工作进行讨论，发现他的证明有错误。

This page collects links around papers that try to settle the "P versus NP" question.

http://www.win.tue.nl/∼ gwoegi/P-versus-NP.htm

2017.08.14，德国波恩大学的计算机科学家Nobert Blum在arXiv上传了一份38页的论文("A solution of the P versus NP Problem")，声称证明了$P \neq NP$，引发了这个领域的关注和讨论。

https://arxiv.org/abs/1708.03486

很快，加州大学伯克利分校的电子工程与计算机科学教授Luca Trevisan就发表意见称Nobert Blum的证明有问题。

Graph Isomorphism in Quasipolynomial Time

From 10 November to 1 December 2015, Laszlo Babai gave three lectures on ≪ Graph Isomorphism in Quasipolynomial Time ≫ in the “Combinatorics and Theoretical Computer Science” Seminar at the University of Chicago.

He outlined a proof to show that the Graph isomorphism problem can be solved in quasi-polynomial time. A preprint is at: http://arxiv.org/pdf/1512.03547v1.pdf

“As long as a branch of science offers an abundance of problems, so long it is alive; a lack of problems foreshadows extinction or the cessation of independent development.”

如果一个科学分支还能找到大量的研究问题，则这个分支还活着；研究问题的缺失则预示着这个科学分支的消亡或者独立发展的终结。

David Hilbert, 1900

Computational complexity theory is alive!!!

计算复杂性理论不是就一个个具体问题去研究它的计算复杂性，而是依据难度去研究各种计算问题之间的联系，按复杂性把问题分成不同的类。

计算复杂性理论不同于可计算性理论（解决哪些是能计算的、哪些是不能计算的）和算法复杂性（具体问题的解决方法）。

Why Complexity Theory?

- Computational resources:

  running time, storage space, parallelism, randomness, rounds of interaction, communication, others···
- The question of which tasks can be performed efficiently is central to the human experience

  Computer problems come in different varieties: some are easy and some are hard
- What is computationally feasible with limited resources ;
- Many applications:

  Logic, Graph theory, Algebra and number theory, Algorithmics, Cryptography, Coding and information theory, Date compression,

密码学中与计算复杂性理论相关的几个研究热点

- 零知识证明

  身份认证，区块链，匿名数字货币。。。

- 后量子密码

  格相关问题，纠错码相关问题，多变量方程组求解等*NP-hard*问题。。。

- 可证明安全性

  安全规约。。。

- 电路混淆

  黑盒混淆，*iO*。。。

计算复杂性的研究历史

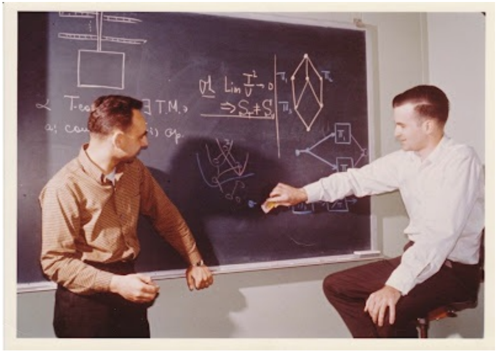
- 在20世纪60年代初，Trahtenbrot和Rabin的论文被认为是该领域最早的文献。

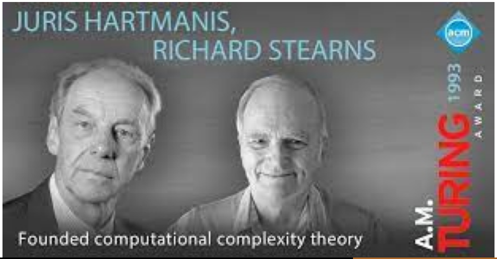
  B. Trakhtenbrot. Turing computations with logarithmic delay. Algebra i Logika, 3(4):33-48, 1964.

  M. Rabin. Real time computation. Israel Journal of Mathematics, 1:203-211, 1963.

- 而一般说来，被公认为奠定了计算复杂性领域基础的是Hartmanis和Stearns的1965的论文On the computational complexity of algorithms。

  在这篇论文中，作者引入了时间复杂性类TIME(f(n))的概念，并利用对角线法证明了时间层级定理（Time Hierarchy Theorem）。




JURIS HARTMANIS,
RICHARD STEARNS
acm
1993
A.M. TURING
AWARD
Founded computational complexity theory

- 20世纪70年代对NP完全问题的研究:
  Cook, Levin 和Karp 的工作证明一大类组合及逻辑问题
  有NP完全性. $\mathrm{P} = \mathrm{NP}$ 问题等价于任何这样问题的有效解.
- 20 世纪70 年代, 概率复杂性
  1977年，Gill定义了类BPP
- 20 世纪70 年代，交互式证明系统（Interactive proof system）理论和零知识证明（Zero-knowledge proof）
  交互证明系统最初由Goldwasser、Micali和Rackoff以及Babai
  分别提出
- 20 世纪80 年代，电路复杂性。
- 去随机化（Derandomization）的研究

- 20 世纪80 年代复杂性理论对近代密码学的影响

  Goldwasser, S. New Directions in Cryptography: Twenty Some Years Later (or Cryptography and Complexity: A Match Made in Heaven). Invited paper FOCS '97, Miami Beach, Florida, pages 314-324, October 1997
- 在20 世纪90 年代，新的计算模型之研究，类似量子计算机和命题证明系统。

  1994年，Peter Shor的量子算法：多项式时间解决整数分解(IF)与离散对数(DLP); 关键技术: 量子傅里叶变换

  1995年, Lov Kumar Grover的O(sqrt(n))时间的无序数据库（大小为n）的搜索算法;

  1997，Bernstein 和Vazirani 为量子计算机有效可计算的BQP类语言的一个形式化定义

- 复杂性理论与博弈论
- 几何复杂性理论纲领(GCT)：试图将代数几何，表示论联系起来！
- 量子计算机的研发与后量子密码体制。

主要内容

- Computational Models: Uniform models(Turing machines); Non-uniform models(circuits and advice);(2)
- P and NP (1)
- Reductions: Cook-reduction; Karp-reduction; Levin-reduction;(1)
- NP-Completeness(4)
- Variations on P and NP(1)
- Time Hierarchies Theorem(1)
- Space Complexity: PSPACE, NLC, Savitch’s Theorem(3)
- Randomness(1)
- Probabilistic Proof Systems: IPS, ZKP, PCP(3)

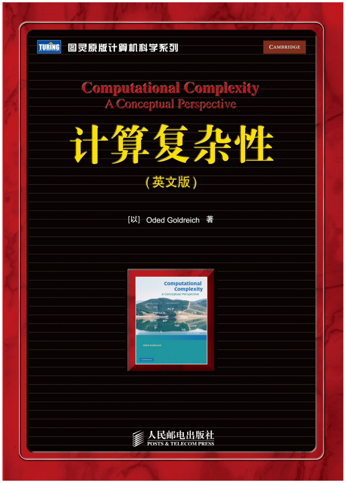
课程成绩

- 课程作业（类似期中考查）：30%
- 期末考试（开卷）：70%

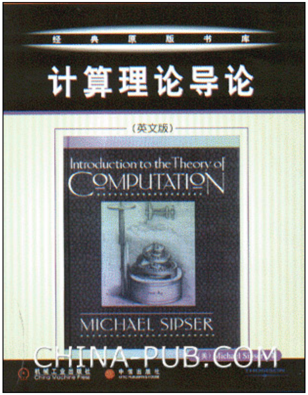
参考书目

Goldreich, O., Computational Complexity: A Conceptual Perspective. 2008: Cambridge University Press.

http://www.wisdom.weizmann.ac.il/∼ oded/

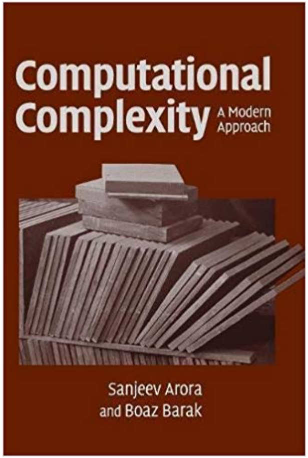

TURING 图灵原版计算机科学系列
CAMBRIDGE
Computational Complexity
A Conceptual Perspective
计算复杂性
（英文版）
[以] Oded Goldreich 著
人民邮电出版社
POSTS & TELECOM PRESS

Sipser, M., Introduction to the Theory of Computation. 2 ed. 2005.



Arora, S. and Barak, B., Computational Complexity: A Modern Approach,

http://www.cs.princeton.edu/theory/complexity/



Background need

- Discrete Mathematics:

  Logic; Set theory; Combinatorics; Graph theory
- Algorithms
- Algebra and number theory:

  group, ring, field, prime number,...
- Probability

  Random variables; Binomial, Bernoulli distributions,...
- Cryptography(不是必须的)

Notations

Big-OH notation;

$f, g : \mathbb{N} \to \mathbb{N}$ integral functions, then we say that $f = O(g)$ (resp. $f = \Omega(g)$, i.e.,$g = O(f)$): if $\exists\ c > 0$ s.t. $f(n) \leq c \cdot g(n)$(resp.$f(n) \geq c \cdot g(n)$) holds for all $n \in \mathbb{N}$.

we say that $f = \Theta(g)$ is $f = O(g)$ and $f = \Omega(g)$.

we say that $f = o(g)$ if for every $\epsilon > 0$, $f(n) \leq \epsilon \cdot g(n)$ for every sufficiently large n, and say that $f = \omega(g)$ if $g = o(f)$.

Here are some examples for use of big-Oh notation:

1. If $f(n) = 100n\log n$ and $g(n) = n^2$ then we have the relations $f = O(g)$, $g = \Omega(f)$, $f = o(g)$, $g = \omega(f)$.
2. If $f(n) = 100n^2 + 24n + 2\log n$ and $g(n) = n^2$ then $f = O(g)$. We will often write this relation as $f(n) = O(n^2)$. Note that we also have the relation $g = O(f)$ and hence $f = \Theta(g)$ and $g = \Theta(f)$.
3. If $f(n) = \min\{n, 10^6\}$ and $g(n) = 1$ for every n then $f = O(g)$. We use the notation $f = O(1)$ to denote this. Similarly, if h is a function that tends to infinity with n (i.e., for every c it holds that $h(n) > c$ for n sufficiently large) then we write $h = \omega(1)$.
4. If $f(n) = 2^n$ then for every number $c \in \mathbb{N}$, if $g(n) = n^c$ then $g = o(f)$. We sometimes write this as $2^n = n^{\omega(1)}$. Similarly, we also write $h(n) = n^{O(1)}$ to denote the fact that h is bounded from above by some polynomial. That is, there exist a number $c > 0$ such that for sufficiently large n, $h(n) \leq n^c$. We'll sometimes also also write $h(n) = \mathrm{poly}(n)$ in this case.

If $f(n)$ and $g(n)$ are two positive function for $n \geq n_0$, and if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = any\ constant,$$

then $f = O(g)$.

By the "length" of an integer we mean the number of bits it has.

We may assume that $\log_2 n$ is an integer rather than using a more cumbersome form as $\lfloor \log_2 n \rfloor$.

For any set S, we denote by 2^S the set of all subsets of S, i.e., $2^S = \{S' : S' \subseteq S\}$.

For a natural number $n \in \mathbb{N}$, we denote $[n] \overset{def}{=} \{1, \ldots, n\}$.

Representation

- Strings(finite binary sequence):

  $\{0,1\}^n$: the set of all strings of length n (n-bit(long) strings);

  $\{0,1\}^*$: the set of all strings, that is, $\{0,1\}^* = \bigcup_{n\in\mathbb{N}}\{0,1\}^n$;

  $|x|$: the length of x; x_i: the i^{th} bit of x;
- Numbers; Unless stated differently, natural numbers will be encoded by their binary expansion.
- Special symbols:

  λ: the empty string, i.e., $\lambda \in \{0,1\}^*$ and $|\lambda| = 0$;

  $\emptyset$: the empty set;

  $\perp$: denotes an indication by some algorithm that something is wrong.

Computational Tasks

A **computation** is a process that modifies an environment via repeated applications of a predetermined rule.

A computation rule is referred to as an algorithm.

Algorithm: An explicit step-by-step procedure for doing calculations (or is a finite set of rules or procedures that must be followed in solving some problem).

We will assume that when invoked on any finite initial environment, the computation halts after a finite number of steps. Typically, the initial environment to which the computation is applied encodes an input string, and the end environment encodes an output string. We consider the mapping from inputs to outputs induced by the computation.

The number of steps taken by the computation on each possible input is called the time complexity of the computational process (or algorithm) While time complexity is defined per input, we will often considers it per input length,taking the maximum over all inputs of the same length.

Complexity:

时间上分类有常数级，多项式，指数，亚指数；

还有空间复杂度，电路复杂度等。

An algorithm to perform a computation is said to be a polynomial time algorithm if there exists an integer d such that the number of bit operations required to perform the algorithm on integers of total length at most k is $O(k^d)$.

Let

$$L_x(\gamma;c) = O(e^{c((\ln x)^\gamma (\ln\ln x)^{1-\gamma})})$$

$L_x(1;c) = O(e^{c\ln x}) = O(x^c)$: exponential time;

$L_x(0;c) = O(e^{c\ln\ln x}) = O((\ln x)^c)$: polynomial time;

$L_x(\gamma;c), 0 < \gamma < 1$: subexponential time

Best, worst and average case complexity

- Best-case complexity: This is the complexity of solving the problem for the best input of size n.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size n.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size n.

An example: Euclidian Algorithm

The greatest common divisor of two numbers is the largest integer which divides both of the numbers.

The Euclidean algorithm uses repeated division to compute the greatest common divisor of two numbers.

INPUT: Two positive integers, a and $b (a > b)$.

OUTPUT: $GCD(a, b)$: the GCD d, of a and b.

```
1 if $b = 0$
2 return $a$
3 else
4 return $GCD(b, a \mod b)$
```

Input size and running time for GCD problem:

- Input size $= O(\log a + \log b)$.
- Time to read input $= O(\log a + \log b)$.
- No. of arithmetic operations $= O(\log a)$.
- Time for each operation $= O((\log a + \log b)^2)$.
- Running time $= O(\log a(\log b + \log a)^2)$.

Problems:

Search Problems; Given an instance, one is required to find a corresponding solution(or to determine that no such solution exists).

Solving a search problem: Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ and $R(x) \overset{\text{def}}{=} \{y : (x,y) \in R\}$ denote the set of solutions for the instance x. A function $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ *solves the search problem of R* if for every x the following holds: if $R(x) \neq \emptyset$ then $f(x) \in R(x)$ and otherwise $f(x) = \perp$.

Note: The solver f is required to find a solution; It is also required f never outputs a wrong solution.

A special case of interest is the case of search problems having a

Decision Problem(Languages);

A decision problem consists of a specification of a subset of the possible instances. Given an instance, one is required to determine whether the instance is in the specified set.

Solving a decision problem: Let $S \subseteq \{0,1\}^*$. A function $f : \{0,1\}^* \to \{0,1\}$ *solves the decision problem of S(or decides membership in S)* if for every x it holds that $f(x) = 1$ if and only if $x \in S$.

Note: We often identify the decision problem of S with S itself, and identify S with its characteristic function(i.e., with $\chi_S : \{0,1\}^* \to \{0,1\}$ defined such that $\chi_S(x) = 1$ if and only if $x \in S$).

Promise Problem;

The domain of possible instances is a subset of $\{0,1\}^*$ *rather than* $\{0,1\}^*$

Other Problems;

Optimization problems: Optimization problems ask for the best possible solution to a problem. A decision or search problem can have several optimization variants.

Counting problems: Counting problems ask for the number of solutions of a given instance.

An algorithm A computes the function $f_A : \{0,1\}^* \to \{0,1\}^*$ defined by $f_A(x) = y$ if, when invoked on input x, algorithm A halts with output y. However, algorithms can also serve as means of "solving search problems" or "making decisions". Specifically, we will say that algorithm A solves the search problem of R (resp., decides membership in S) if f_A solves the search problem of R (resp., decides membership in S).

Algorithms as problem solvers: We denote by $A(x)$ the output of algorithm A on input x. Algorithm A solves the search problem R (resp. the decision problem of S) if A, viewed as a function, solves R (resp. S).

In order to define computation(and computation time) rigorously, one needs to specify some model of computation, that is, provide a concrete definition of environments and a class of rules that may be applied to them.

计算复杂性理论的研究对象是算法在执行时所需的计算资源，而为了讨论这一点，我们必须假设算法是在某个计算模型上运行的。常讨论的计算模型包括图灵机（Turing machine）和电路（circuit），它们分别是一致性（uniform）和非一致性（non-uniform）计算模型的代表。

A computational task can be solved by a real life computer if and only if it can be solved by a Turing machine. In comparison to real life computers, the model of Turing machines is extremely over-simplified and abstract away many issues that are of great concern to computer practice

A. M. Turing: ：Father of the computer

Alan Mathison Turing，(1912.6.23-1954.6.7)，英国数学家、逻辑学家，他被视为计算机之父。1931年图灵进入剑桥大学国王学院，毕业后到美国普林斯顿大学攻读博士学位，二战爆发后回到剑桥，后曾协助军方破解德国的著名密码系统Enigma，帮助盟军取得了二战的胜利.

1936年，图灵向伦敦权威的数学杂志投了一篇论文，题为“论数字计算在决断难题中的应用”(On Computable Numbers, with an application to the Entscheidungsproblem)。在这篇开创性的论文中，图灵给“可计算性”下了一个严格的数学定义，并提出著名的“图灵机”(Turing Machine)的设想。“图灵机”不是一种具体的机器，而是一种思想模型，可制造一种十分简单但运算能力极强的计算装置，用来计算所有能想象得到的可计算函数。“图灵机”与“冯・诺伊曼机”齐名，被永远载入计算机的发展史中。1950年10月，图灵又发表了另一篇题为“机器能思考吗”(Can machines think?)的论文，成为划时代之作。也正是这篇文章，为图灵赢得了“人工智能之父”的桂冠。

二战中帮助英国破译ENIGMA密码机的最大功臣。按照图灵的计算理论，布莱切利公园得到十万镑的经费来研制这种机器，绰号叫“炸弹”(bombes)，并利用此机器破解了ENIGMA。

与ENIGMA密码机破译相关的4位关键人物：图灵，雷耶夫斯基，弗里德曼，施密特

苹果公司与被图灵咬了一口的苹果

在2009年9月10日，一份超过3万人的请愿签名，使英国首相戈登·布朗在《每日电讯报》撰文，因为英国政府当年以同性恋相关罪名起诉图灵并定罪，导致他自杀身亡，正式向艾伦·图灵公开道歉。

2013年12月24日，英国司法大臣宣布英国女王伊丽莎白二世赦免1952年因同性恋行为被定罪的艾伦·图灵。

为了纪念，图灵的事迹已被拍成影视剧，写成小说、诗歌等，以他名字命名的"图灵奖"也已成为计算机界的诺贝尔奖。

牛津大学著名数学家安德鲁·哈吉斯在为图灵写的一部脍炙人口的传记《谜一样的图灵》（Alan Turing: The Enigma）中这样描述到："图灵似乎是上天派来的一个使者，匆匆而来，匆匆而去，为人间留下了智慧，留下了深邃的思想，后人必须为之思索几十年、上百年甚至永远。"

Thank You Very Much!