算法设计与分析

复杂度分析

目录

问题与解

问题描述

- **一组参数**：用于刻画问题（集合、变量、函数、序列等），包括定义域和参数间关系的描述
- **解的定义**：由优化目标或约束条件确定

问题与解

问题描述

- **一组参数**：用于刻画问题（集合、变量、函数、序列等），包括定义域和参数间关系的描述
- **解的定义**：由优化目标或约束条件确定

实例：对参数进行一次赋值 $\rightarrow$ 问题的一个实例

算法

定义 1（算法）

算法 A 是一个有限的、定义明确的、可由计算机实现的指令序列，用于解决一类问题。

- 算法总是无歧义的
- 是执行计算、数据处理、自动推理及其他任务的规范

问题 P 的算法 A

- 以 P 的任意实例作为 A 的输入，每一步计算都是确定性的
- A 在有限步内终止
- 总是输出正确的解

基本计算步骤与输入规模

深刻的分析建立在正确的简化之上。

基本计算步骤：抽象的原子操作

- 例如：比较、加法、乘法、交换、赋值……

这是第一个重要的简化！

输入规模：刻画实例的规模，与实例编码串的长度成正比

- 例如：数组元素个数、调度任务数、顶点和边的数目

输入规模与基本计算步骤的示例（1/2）

排序：数组 $a[n]$

- n：数组中元素的个数
- 元素比较与移动

查找：在数组 $a[n]$ 中查找 x

- n：数组中元素的个数
- x 与 $a[i]$ 之间的元素比较

整数乘法：$a \times b$

- a 和 b 的二进制长度，即 $m = \log_2 a$，$n = \log_2 b$
- 按位乘法——$a \times b$ 需要 mn 次按位乘法

输入规模与基本计算步骤的示例（2/2）

矩阵乘法：$A_{n_1 \times n_2} \cdot B_{n_2 \times n_3}$

- A 和 B 的维度，即 n_1, n_2, n_3
- 逐点乘法——$A \cdot B$ 需要 $n_1 n_2 n_3$ 次逐点乘法
- 当 $n_1 = n_2 = n_3 = n$ 时，需要 n^3 次

图遍历：$G = (V, E)$

- 顶点数和边数
- 标记变量的赋值

算法效率的度量

通过将基本计算步骤的数目表示为输入规模的函数，来刻画运行时间。

简洁、与机器无关的刻画方式

算法效率的度量

通过将基本计算步骤的数目表示为输入规模的函数，来刻画运行时间。

简洁、与机器无关的刻画方式

对于相同规模的不同输入，基本计算步骤的数目可能不同 $\Rightarrow$ 函数可能不同

选择哪一个?

三种分析类型

“做最坏的打算，抱最好的希望。”

—Benjamin Disraeli

三种分析类型

“做最坏的打算，抱最好的希望。”

—Benjamin Disraeli

最坏情况：规模为 n 的所有输入中的最大运行时间
例：快速排序对 n 个元素排序最多需要 n^2 次比较

三种分析类型

“做最坏的打算，抱最好的希望。”

—Benjamin Disraeli

最坏情况：规模为 n 的所有输入中的最大运行时间
例：快速排序对 n 个元素排序最多需要 n^2 次比较

最好情况：规模为 n 的所有输入中的最小运行时间
例：当输入已排序时，插入排序只需要 n 次比较

三种分析类型

“做最坏的打算，抱最好的希望。”

—Benjamin Disraeli

最坏情况：规模为 n 的所有输入中的最大运行时间
例：快速排序对 n 个元素排序最多需要 n^2 次比较

最好情况：规模为 n 的所有输入中的最小运行时间
例：当输入已排序时，插入排序只需要 n 次比较

平均情况：规模为 n 的随机输入的期望运行时间
例：快速排序的期望元素比较次数约为 $n \log n$

关于最坏情况

算法：一些指数时间算法在实践中被广泛使用，因为最坏情况的实例似乎很少见。

- Linux 的 grep 命令

关于最坏情况

算法：一些指数时间算法在实践中被广泛使用，因为最坏情况的实例似乎很少见。

- Linux 的 grep 命令

密码学：要求困难实例能够被高效采样——仅具有高最坏情况复杂度的问题可能不适合作为困难性假设

关于最坏情况

算法：一些指数时间算法在实践中被广泛使用，因为最坏情况的实例似乎很少见。

- Linux 的 grep 命令

密码学：要求困难实例能够被高效采样——仅具有高最坏情况复杂度的问题可能不适合作为困难性假设

对算法的好消息 = 对密码学的坏消息

双赢的意味

$A(n)$ 的公式

$A(n)$：平均情况复杂度

- 设 X 为规模为 n 的所有输入的集合，$\Pr[x \in X] = p(x)$
- $t(x)$：算法 A 在输入 x 上执行的基本操作次数

$$A(n) = \sum_{x \in X} p(x)t(x)$$

在很多情况下，我们假设输入服从均匀分布。

查找问题示例

查找问题

输入：升序数组 $a[n]$，待查找元素 x

输出：$j \in [0, \ldots, n]$

- 若 $x \in a[n]$，则 j 是满足 $a[j] = x$ 的第一个下标
- 否则，$j = 0$

基本操作：x 与 $a[i]$ 之间的元素比较

顺序查找算法

Require: 数组 $a[n]$，待查找元素 x
Ensure: 下标 j

1: $flag \leftarrow 0$
2: **for** $j = 1$ **to** n **do**
3: **if** $a[j] = x$ **then**
4: $flag \leftarrow 1$
5: **break**
6: **end if**
7: **end for**
8: **if** $flag = 0$ **then**
9: $j \leftarrow 0$
10: **end if**
11: **return** j

示例：$1, 2, 3, 4, 5$

- $x = 4$：4 次比较
- $x = 2.5$：5 次比较

最坏情况复杂度

共有 $2n+1$ 种不同类型的输入：

- **在数组内**：$x = a[1], x = a[2], \ldots, x = a[n]$
- **在数组外**：$x < a[1], a[1] < x < a[2], \ldots, a[n] < x$

最坏情况输入：$x \notin A$ 或 $x = A[n]$，需要 n 次比较

最坏情况复杂度：$T(n) = n$

平均情况复杂度

假设 $\Pr[x \in A] = p$，且在每个位置上的分布概率相等。

$$T(n) = \sum_{i=1}^{n} i \cdot \frac{p}{n} + (1-p)n \quad //\text{等差数列求和}$$

$$= \frac{p(n+1)}{2} + (1-p)n$$

当 $p = 1/2$ 时：

$$T(n) = \frac{n+1}{4} + \frac{n}{2} \approx \frac{3n}{4}$$

目录

算法的伪代码

定义 2（伪代码）

对算法操作原理的非正式高层描述：使用程序设计语言的结构约定，但面向人类阅读而非机器阅读。

指令	符号
赋值	$\leftarrow$ 或 :=
分支语句	if...then...[else...]
循环结构	while, for, repeat until
转移语句	goto
返回语句	return
函数调用	Func()
注释	// 或 /* */

示例：求最大公约数的欧几里得算法

Require: $n, m \in \mathbb{Z}^+, \ n \geq m$
Ensure: $\gcd(n, m)$
1: **while** $m > 0$ **do**
2: $\quad r \leftarrow n \mod m$
3: $\quad n \leftarrow m$
4: $\quad m \leftarrow r$
5: **end while**
6: **return** n

演示：$n = 36, m = 15$

循环	n	m	r
第 1 次	36	15	6
第 2 次	15	6	3
第 3 次	6	3	0
	3	0	0

输出：3

插入排序示例

Require: 数组 $A[n]$
Ensure: 升序排列的 $A[n]$

```
1: for j ← 2 to n do
2:     x ← A[j]
3:     i ← j − 1 {将 A[j] 插入 A[1 . . . j − 1]}
4:     while i > 0 and x < A[i] do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← x
9: end for
```

i 是最终插入位置的左邻下标

插入排序演示

二路归并排序示例

Require: 数组 $A[l, r]$
Ensure: 升序排列的 $A[l, r]$
1: **if** $l < r$ **then**
2: $m \leftarrow \lfloor (l + r)/2 \rfloor$
3: MergeSort(A, l, m)
4: MergeSort$(A, m + 1, r)$
5: Merge(A, l, m, r)
6: **end if**

MergeSort 是一个递归算法

- 在其代码内部调用自身

算法 A 的伪代码

Require: 数组 $P[0, \ldots, n] \in \mathbb{R}^{n+1}$，$x \in \mathbb{R}$
Ensure: y
1: $y \leftarrow P[0]$; $power \leftarrow 1$
2: **for** $i \leftarrow 1$ **to** n **do**
3: $\quad power \leftarrow power \times x$
4: $\quad y \leftarrow y + P[i] \times power$
5: **end for**
6: **return** y

第 3–4 行计算了什么?

算法 A 的解释

循环	$power$	y
0	1	$P[0]$
1	x	$P[0] + P[1] \times x$
2	x^2	$P[0] + P[1] \times x + P[2] \times x^2$
3	x^3	$P[0] + P[1] \times x + P[2] \times x^2 + P[3] \times x^3$
		$\vdots$

输入 $P[0, \ldots, n]$ 是 n 次多项式 $P(x)$ 的系数
算法 A 计算 $P(x) = \sum_{i=0}^{n} P[i]x^i$

目录

动机

我们使用定义在 $\mathbb{N}$ 上的函数来刻画算法的运行时间或空间需求如何随输入规模增长。

动机

我们使用定义在 $\mathbb{N}$ 上的函数来刻画算法的运行时间或空间需求如何随输入规模增长。

如何比较它们？如何分类它们？

动机

我们使用定义在 $\mathbb{N}$ 上的函数来刻画算法的运行时间或空间需求如何随输入规模增长。

如何比较它们？如何分类它们？

第一个简化引出了另一个简化。现在，第二个简化登场了：考虑函数的阶而非其具体形式。

大 O 记号

Paul Bachmann 和 Edmund Landau 发明了一族记号，称为大 O 记号，用于描述当输入趋向无穷大时函数的极限行为。

[Paul Bachmann 与 Edmund Landau 的图片]

- 也称为 Bachmann-Landau 记号或渐近记号
- 数学记号 $\rightarrow$ 描述运行时间

大 O 记号

定义 3（大 O）

$\exists c > 0$，$\exists n_0$，使得 $\forall n \geq n_0$：

$$f(n) \leq c \cdot g(n)$$

f 渐近地被 g 从上界定（相差常数因子）

$$f(n) = O(g(n))$$

极限定义

$$\limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

一些说明

大 O 记号根据增长率对函数进行分类：具有相同增长率的不同函数可以用相同的 O 记号表示。

- 使用字母 O 是因为函数的增长率也被称为函数的**阶**
- 存在很多 (c, n_0) 对，找到一对即可
- 对于有限值 $n \leq n_0$，不等式可能不成立
- 常数函数可以写成 $O(1)$

关于大 O 的更多内容

$f(n) = O(g(n))$：$f(n)$ 的阶小于等于 $g(n)$ 的阶
典型用法：给出上界

- 插入排序对 n 个元素排序需要 $O(n^2)$ 次比较

例 1：$f(n) = n^2 + n$

- $f(n) = O(n^2)$ ← 取 $c = 2$，$n_0 = 1$
- $f(n) = O(n^3)$ ← 取 $c = 1$，$n_0 = 2$

例 2：$f(n) = 32n^2 + 17n + 1$

- $f(n) = O(n^2)$ ← 取 $c = 50$，$n_0 = 1$
- $f(n)$ 也是 $O(n^3)$
- $f(n)$ 既不是 $O(n)$ 也不是 $O(n \log n)$

大 O 的局限性

大 O 记号只提供函数增长率的**上界**。

与大 O 记号相关的还有几个记号，使用符号 o、Ω、ω 和 Θ，用于描述渐近增长率的其他类型的界。

大 Ω 记号

定义 4（大 Ω）

$\exists c > 0$，$\exists n_0$，$\forall n \geq n_0$：

$$f(n) \geq c \cdot g(n)$$

f 渐近地被 g 从下界定

$$f(n) = \Omega(g(n))$$

极限定义

$$\liminf_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

大 Ω 示例

$f(n) = \Omega(g(n))$：$f(n)$ 的阶大于等于 $g(n)$ 的阶

典型用法：给出下界

- 任何基于比较的排序算法在最坏情况下都需要 $\Omega(n \log n)$ 次比较

无意义的陈述：任何基于比较的排序算法在最坏情况下至少需要 $O(n \log n)$ 次比较。

- $O(\cdot)$ 不能给出下界

例：$f(n) = n^2 + n$

- $f(n) = \Omega(n^2) \leftarrow c = 1$，$n_0 = 1$
- $f(n) = \Omega(100n) \leftarrow c = 1/100$，$n_0 = 1$

紧界

大 O 和 Ω 记号最初用作算法代价增长的紧上界（或紧下界）。
但是，根据定义：

$g(n)$ 可能是一个松的上界（或下界）。

紧界

大 O 和 Ω 记号最初用作算法代价增长的紧上界（或紧下界）。
但是，根据定义：

$g(n)$ 可能是一个松的上界（或下界）。

为了更清楚地表明紧上界的作用，使用小 o 和 ω 记号来描述不可能是紧的上界/下界。

小 o 记号

定义 5（小 o）

$\forall c > 0$，$\exists n_0$，使得 $\forall n \geq n_0$：

$$f(n) < c \cdot g(n)$$

f 渐近地被 g 支配：

$$f(n) = o(g(n))$$

极限定义

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

关于小 o 的更多内容

$f(n) = o(g(n))$：$f(n)$ 的阶**严格小于** $g(n)$ 的阶
典型用法：$\log n = o(n)$

例：$f(n) = n^2 + n$，$f(n) = o(n^3)$

- $c \geq 1$：显然成立，取 $n_0 = 2 \Rightarrow n^2 + n < cn^3$

$$cn^3 \geq n^3 = n^2((n-1)+1) \geq n^2 + n, \quad \text{当} n \geq n_0$$

- $0 < c < 1$：取 $n_0 > \lceil 2/c \rceil$，因为

$$cn \geq cn_0 \geq 2 \Rightarrow n^2 + n < 2n^2 \leq cn \cdot n^2 < cn^3$$

小 ω 记号

定义 6（小 ω）

$\forall c > 0$，$\exists n_0$，$\forall n \geq n_0$：

$$f(n) > c \cdot g(n)$$

f 渐近地支配 g：

$$f(n) = \omega(g(n))$$

极限定义

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

小 ω 示例

$f(n) = \omega(g(n))$：$f(n)$ 的阶**严格大于** $g(n)$ 的阶
典型用法：$n = \omega(\log n)$

例：$f(n) = n^2 + n$，$f(n) = \omega(n)$

- $\lim_{n\to\infty} \frac{f(n)}{n} = \infty$
- $f(n) \neq \omega(n^2)$：取 $c = 2$，不存在 n_0 使得 $\forall n \geq n_0$

$$cn^2 = 2n^2 < n^2 + n$$

这些记号之间关系的可视化

比较

记号	$?c > 0$	$?n_0$	$f(n)\ ?\ c \cdot g(n)$	含义
O	$\exists$	$\exists$	$\leq$	上界
o	$\forall$	$\exists$	$<$	非紧上界
Ω	$\exists$	$\exists$	$\geq$	下界
ω	$\forall$	$\exists$	$>$	非紧下界

虽然 o 和 ω 不常用于描述算法，但我们定义 O 和 Ω 的组合：Θ，表示 $g(n)$ 既是紧上界也是紧下界。

大 Θ 记号：旨在给出紧界

定义 7（大 Θ）

$\exists c_1 > 0$，$\exists c_2 > 0$，$\exists n_0$，使得 $\forall n > n_0$：

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

f 渐近地被 g 从上下两方界定

$$f(n) = \Theta(g(n))$$

极限定义

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \quad (c > 0)$$

等价性证明

证明：由极限的定义 $\Rightarrow \forall \varepsilon > 0$，$\exists n_0$，$\forall n \geq n_0$：

$$|f(n)/g(n) - c| < \varepsilon$$

$$c - \varepsilon < f(n)/g(n) < c + \varepsilon$$

等价性证明

证明：由极限的定义 $\Rightarrow \forall \varepsilon > 0$，$\exists n_0$，$\forall n \geq n_0$：

$$|f(n)/g(n) - c| < \varepsilon$$

$$c - \varepsilon < f(n)/g(n) < c + \varepsilon$$

取 $\varepsilon = c/2 \Rightarrow c/2 < f(n)/g(n) < 3c/2$

- $\forall n \geq n_0$，$f(n) \leq (3c/2)g(n) \Rightarrow f(n) = O(g(n))$
- $\forall n \geq n_0$，$f(n) \geq (c/2)g(n) \Rightarrow f(n) = \Omega(g(n))$

等价性证明

证明：由极限的定义 $\Rightarrow \forall \varepsilon > 0$，$\exists n_0$，$\forall n \geq n_0$：

$$|f(n)/g(n) - c| < \varepsilon$$

$$c - \varepsilon < f(n)/g(n) < c + \varepsilon$$

取 $\varepsilon = c/2 \Rightarrow c/2 < f(n)/g(n) < 3c/2$

- $\forall n \geq n_0$，$f(n) \leq (3c/2)g(n) \Rightarrow f(n) = O(g(n))$
- $\forall n \geq n_0$，$f(n) \geq (c/2)g(n) \Rightarrow f(n) = \Omega(g(n))$

这证明了 $f(n) = \Theta(g(n))$

关于大 Θ 的更多内容

$f(n) = \Theta(g(n))$：$f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$，$f(n)$ 和 $g(n)$ 具有相同的阶

典型用法：

- 归并排序对 n 个元素排序需要 $\Theta(n \log n)$ 次比较

例 1：$f(n) = n^2 + n$，$g(n) = 100n^2$

- $f(n) = \Theta(g(n))$

例 2：$f(n) = 32n^2 + 17n + 1$

- $f(n)$ 是 $\Theta(n^2)$ ← 取 $c_1 = 32$，$c_2 = 50$，$n_0 = 1$
- $f(n)$ 既不是 $\Theta(n)$ 也不是 $\Theta(n^3)$

素性测试示例

Require: 奇整数 $n > 2$
Ensure: true 或 false
1: $s \leftarrow \lfloor n^{1/2} \rfloor$
2: **for** $j \leftarrow 2$ **to** s **do**
3: **if** j 整除 n **then**
4: **return** false
5: **end if**
6: **end for**
7: **return** true

素性测试示例

Require: 奇整数 $n > 2$
Ensure: true 或 false
1: $s \leftarrow \lfloor n^{1/2} \rfloor$
2: **for** $j \leftarrow 2$ **to** s **do**
3: **if** j 整除 n **then**
4: **return** false
5: **end if**
6: **end for**
7: **return** true

如果 $n^{1/2}$ 可在 $O(1)$ 时间内计算，基本操作是除法

素性测试示例

Require: 奇整数 $n > 2$
Ensure: true 或 false
1: $s \leftarrow \lfloor n^{1/2} \rfloor$
2: **for** $j \leftarrow 2$ **to** s **do**
3: **if** j 整除 n **then**
4: **return** false
5: **end if**
6: **end for**
7: **return** true

如果 $n^{1/2}$ 可在 $O(1)$ 时间内计算，基本操作是除法

朴素素性测试的最坏情况复杂度是多少?

输入规模很重要：素性测试的情形

以 n 作为 $W(\cdot)$ 的输入规模：

$$W(n) = O(n^{1/2})\checkmark \quad W(n) = \Omega(n^{1/2})\times$$

- 考虑形如 $3m$ 的输入，则 $n^{1/2}$ 不是下界

以 λ（n 的二进制表示长度）作为 $W(\cdot)$ 的输入规模：

$$W(n) = O(2^{\lambda/2})\checkmark \quad W(n) = \Omega(2^{\lambda/2})\checkmark$$

即 $W(\lambda) = \Theta(2^{\lambda/2})$

输入规模旨在刻画一类实例的规模。这使得该概念有意义。对于第一种情况，一类实例退化为单个实例，从而使最坏情况复杂度失去意义。

多变量的大 O 记号

上界：$f(m,n)=O(g(m,n))$，若 $\exists c>0$，$m_0\geq 0$，$n_0\geq 0$，使得 $\forall n\geq n_0$，$m\geq m_0$，$f(m,n)\leq c\cdot g(m,n)$

例：$f(m,n)=32mn^2+17mn+32n^3$

- $f(m,n)$ 既是 $O(mn^2+n^3)$ 也是 $O(mn^3)$
- $f(m,n)$ 既不是 $O(n^3)$ 也不是 $O(mn^2)$

典型用法：广度优先搜索在有向图中寻找从 s 到 t 的最短路径需要 $O(m+n)$ 时间

大 O 记号的性质（1/2）

传递性：函数的阶具有传递性

- $f = O(g) \land g = O(h) \Rightarrow f = O(h)$
- $f = \Omega(g) \land g = \Omega(h) \Rightarrow f = \Omega(h)$
- $f = \Theta(g) \land g = \Theta(h) \Rightarrow f = \Theta(h)$
- $f = o(g) \land g = o(h) \Rightarrow f = o(h)$
- $f = \omega(g) \land g = \omega(h) \Rightarrow f = \omega(h)$

大 O 记号的性质（2/2）

乘积

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

大 O 记号的性质（2/2）

乘积

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

求和

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$
- 这意味着 $f_1 = O(g) \wedge f_2 = O(g) \Rightarrow f_1 + f_2 \in O(g)$，即 $O(g)$ 是凸锥

大 O 记号的性质（2/2）

乘积

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

求和

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$
- 这意味着 $f_1 = O(g) \wedge f_2 = O(g) \Rightarrow f_1 + f_2 \in O(g)$，即 $O(g)$ 是凸锥

此性质可推广到 f_i 的有限复合

- **应用**：对于一个算法，如果其每一步的运行时间都被 $h(n)$ 上界，且算法只包含常数步，则总体复杂度为 $O(h(n))$

大 O 记号的性质（2/2）

乘积

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

求和

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$
- 这意味着 $f_1 = O(g) \wedge f_2 = O(g) \Rightarrow f_1 + f_2 \in O(g)$，即 $O(g)$ 是凸锥

此性质可推广到 f_i 的有限复合

- **应用**：对于一个算法，如果其每一步的运行时间都被 $h(n)$ 上界，且算法只包含常数步，则总体复杂度为 $O(h(n))$

常数乘法：设 $k > 0$ 为常数，则：

- $O(kg) = O(g)$，若 $k \neq 0$
- $f = O(g) \Rightarrow kf = O(g)$（乘法常数可省略）

目录

重要的函数类（按增长速度递增）

- 常数：$O(1)$
- 双对数：$\log\log n$
- 对数：$\log n$
- 多对数：$(\log n)^c$，$c > 1$
- 分数幂：n^c，$0 < c < 1$
- 线性：$O(n)$
- 对数线性或拟线性：$n \log n$
- 多项式：n^c，$c > 1$（二次：n^2，三次：n^3）
- 指数：c^n，$c > 1$
- 阶乘：$n!$

常见函数的渐近界（1/3）

技术工具：O、Ω、Θ、o、ω 的极限定义

多项式：设 $f(n) = a_0 + a_1 n + \cdots + a_d n^d$，则 $f(n) = \Theta(n^d)$

证明：

$$\lim_{n\to\infty} \frac{a_0 + a_1 n + \cdots + a_d n^d}{n^d} = a_d > 0$$

例：设 $f(n) = n^2/2 - 3n$，则 $f(n) = \Theta(n^2)$

常见函数的渐近界（2/3）

对数：对于任意常数 $a, b > 0$，$\Theta(\log_a n) \sim \Theta(\log_b n)$

- 无需指定底数（假设底数为常数）

常见函数的渐近界（2/3）

对数：对于任意常数 $a, b > 0$，$\Theta(\log_a n) \sim \Theta(\log_b n)$

- 无需指定底数（假设底数为常数）

对数 vs. 多项式：$\forall d > 0$，$\log n = o(n^d)$

常见函数的渐近界（2/3）

对数：对于任意常数 $a, b > 0$，$\Theta(\log_a n) \sim \Theta(\log_b n)$

- 无需指定底数（假设底数为常数）

对数 vs. 多项式：$\forall d > 0$，$\log n = o(n^d)$

证明：$\lim_{n\to\infty} \ln n = \infty$ 且 $\lim_{n\to\infty} n^d = \infty$，且均可微分。应用洛必达（Bernoulli）法则：

$$\lim_{n\to\infty} \frac{\ln n}{n^d} = \lim_{n\to\infty} \frac{1/n}{dn^{d-1}} = \lim_{n\to\infty} \frac{1}{dn^d} = 0$$

常见函数的渐近界（3/3）

指数 vs. 多项式：$\forall c > 1$，$\forall d > 0$，$n^d = o(c^n)$

常见函数的渐近界（3/3）

指数 vs. 多项式：$\forall c > 1$，$\forall d > 0$，$n^d = o(c^n)$

证明：不失一般性，设 d 为正整数。$\lim_{n\to\infty} n^d = \infty$ 且 $\lim_{n\to\infty} c^n = \infty$，且均可微分。反复应用洛必达法则直到分子为常数：

$$\lim_{n\to\infty} \frac{n^d}{c^n} = \lim_{n\to\infty} \frac{dn^{d-1}}{c^n \ln c} = \lim_{n\to\infty} \frac{d(d-1)n^{d-2}}{c^n (\ln c)^2} = \cdots = \lim_{n\to\infty} \frac{d!}{c^n (\ln c)^d} = 0$$

阶乘函数

Stirling 公式

（以 James Stirling 命名，但最早由 Abraham de Moivre 提出）

阶乘函数

Stirling 公式

（以 James Stirling 命名，但最早由 Abraham de Moivre 提出）

精确形式:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

阶乘函数

Stirling 公式

（以 James Stirling 命名，但最早由 Abraham de Moivre 提出）

精确形式：

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

简化形式：

$$\ln n! = n \ln n - n + O(\ln n)$$

阶乘函数

Stirling 公式

（以 James Stirling 命名，但最早由 Abraham de Moivre 提出）

精确形式：

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

简化形式：

$$\ln n! = n\ln n - n + O(\ln n)$$

- $n! = o(n^n)$
- $n! = \omega(2^n)$
- $\ln n! = \Theta(n\ln n)$（通过积分法证明）

上界的证明

$$\ln n! = \sum_{k=1}^{n} \ln k \leq \int_{2}^{n+1} \ln x \, dx$$

$$= (x \ln x - x)\Big|_{2}^{n+1} = O(n \ln n)$$

下界的证明

$$\ln n! = \sum_{k=1}^{n} \ln k \geq \int_{1}^{n} \ln x \, dx$$

$$= (x \ln x - x)\Big|_{1}^{n} = n \ln n - n + 1 = \Omega(n \ln n)$$

应用：估计搜索空间的大小

回顾 ROI 优化问题：将 m 枚硬币分配到 n 个项目的不同投资方案数

$$C_{m+n-1}^{m} = \frac{(m+n-1)!}{m!(n-1)!}$$

$$= \frac{\sqrt{2\pi(m+n-1)}(m+n-1)^{m+n-1}\left(1+\Theta\left(\frac{1}{m+n-1}\right)\right)}{\sqrt{2\pi m}m^{m}\left(1+\Theta\left(\frac{1}{m}\right)\right)\sqrt{2\pi(n-1)}(n-1)^{n-1}\left(1+\Theta\left(\frac{1}{n-1}\right)\right)}$$

$$= \Theta((1+\varepsilon)^{m+n-1})$$

取整函数

取整是指用一个近似相等但表示更短、更简单的数来替换原数

- **向下取整**（floor）：$y = \lfloor x \rfloor$：y 是不超过 x 的最大整数
- **向上取整**（ceiling）：$y = \lceil x \rceil$：y 是不小于 x 的最小整数

取整函数

取整是指用一个近似相等但表示更短、更简单的数来替换原数

- **向下取整**（floor）：$y = \lfloor x \rfloor$：y 是不超过 x 的最大整数
- **向上取整**（ceiling）：$y = \lceil x \rceil$：y 是不小于 x 的最小整数

例：$\lfloor 2.6 \rfloor = 2$，$\lceil 2.6 \rceil = 3$，$\lfloor 2 \rfloor = \lceil 2 \rceil = 2$

取整函数

取整是指用一个近似相等但表示更短、更简单的数来替换原数

- **向下取整**（floor）：$y = \lfloor x \rfloor$：y 是不超过 x 的最大整数
- **向上取整**（ceiling）：$y = \lceil x \rceil$：y 是不小于 x 的最小整数

例：$\lfloor 2.6 \rfloor = 2$，$\lceil 2.6 \rceil = 3$，$\lfloor 2 \rfloor = \lceil 2 \rceil = 2$

应用：在 $A[n]$ 中进行二分查找时，中位数的下标是 $\lfloor n/2 \rfloor$，子问题的规模是 $\lfloor n/2 \rfloor$

取整函数的性质

命题 1：$x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1$

证明：分两种情况考虑：

1. x 是整数：显然成立
2. $\exists n\in\mathbb{Z}$ 使得 $n<x<n+1$，由取整函数的定义 $\Rightarrow\lfloor x\rfloor=n$，$\lceil x\rceil=n+1$

命题 2：设 $n,a,b\in\mathbb{Z}$，有：

$$\lfloor x+n\rfloor=\lfloor x\rfloor+n,\quad \lceil x+n\rceil=\lceil x\rceil+n$$

$$\left\lceil\frac{n}{2}\right\rceil+\left\lfloor\frac{n}{2}\right\rfloor=n$$

$$\left\lceil\frac{\lceil n/a\rceil}{b}\right\rceil=\left\lceil\frac{n}{ab}\right\rceil,\quad \left\lfloor\frac{\lfloor n/a\rfloor}{b}\right\rfloor=\left\lfloor\frac{n}{ab}\right\rfloor$$

运行时间

目录

常见运行时间（1/4）

常数时间：$T(n) = O(1)$

- 判断一个二进制数是奇数还是偶数
- 数组 $A[i]$ 的随机访问或哈希表（键值对）访问

常见运行时间（1/4）

常数时间：$T(n) = O(1)$

- 判断一个二进制数是奇数还是偶数
- 数组 $A[i]$ 的随机访问或哈希表（键值对）访问

对数时间：$T(n) = O(\log n)$

- 在大小为 n 的有序数组中查找：二分查找

常见运行时间（1/4）

常数时间：$T(n) = O(1)$

- 判断一个二进制数是奇数还是偶数
- 数组 $A[i]$ 的随机访问或哈希表（键值对）访问

对数时间：$T(n) = O(\log n)$

- 在大小为 n 的有序数组中查找：二分查找

分数幂时间：$T(n) = n^{1/2}$

- 素性测试

常见运行时间（2/4）

线性时间：$T(n) = O(n)$：运行时间与输入规模成正比

- 归并：将两个有序列表 $A = a_1, \ldots, a_n$ 与 $B = b_1, \ldots, b_n$ 合并成一个有序整体

常见运行时间（2/4）

线性时间：$T(n) = O(n)$：运行时间与输入规模成正比

- 归并：将两个有序列表 $A = a_1, \ldots, a_n$ 与 $B = b_1, \ldots, b_n$ 合并成一个有序整体

每次比较后，输出列表的长度至少增加 1。当一个列表为空时，另一个列表的剩余部分直接合并到结果列表。

- 上界：$2n - 1$ vs. 下界：n

常见运行时间（3/4）

对数线性时间：$T(n) = O(n \log n)$（出现在分治算法中）

- 归并排序和堆排序是执行 $O(n \log n)$ 次比较的排序算法
- 快速傅里叶变换（FFT）

常见运行时间（3/4）

对数线性时间：$T(n) = O(n \log n)$（出现在分治算法中）

- 归并排序和堆排序是执行 $O(n \log n)$ 次比较的排序算法
- 快速傅里叶变换（FFT）

二次时间：$T(n) = O(n^2)$

- **最近点对**：给定平面上 n 个点 $(x_1, y_1), \ldots, (x_n, y_n)$，找出距离最近的一对。$O(n^2)$ 解法：尝试所有点对

注：$\Omega(n^2)$ 似乎不可避免，但这只是错觉。

常见运行时间（3/4）

对数线性时间：$T(n) = O(n \log n)$（出现在分治算法中）

- 归并排序和堆排序是执行 $O(n \log n)$ 次比较的排序算法
- 快速傅里叶变换（FFT）

二次时间：$T(n) = O(n^2)$

- **最近点对**：给定平面上 n 个点 $(x_1, y_1), \ldots, (x_n, y_n)$，找出距离最近的一对。$O(n^2)$ 解法：尝试所有点对

注：$\Omega(n^2)$ 似乎不可避免，但这只是错觉。

三次时间：枚举所有三元组

- 朴素矩阵乘法：$A_{n \times n} \times B_{n \times n}$：每个 $c_{i,j}$ 需要 $O(n)$ 次乘法，$C_{n \times n}$ 共有 n^2 个元素

常见运行时间（4/4）

多项式时间：$T(n) = O(n^k)$

- **大小为 k 的独立集**：给定一个 n 个节点的图，是否存在 k 个节点使得任意两个都不相邻？
- 枚举所有 k 个节点的子集然后检验
  - 检验 S_k 是否为独立集需要 $O(k^2)$ 时间
  - $\#(S_k) = C_n^k \leq n^k/k!$
- $O(k^2 n^k/k!) = O(n^k)$（当 $k = 17$ 时为多项式时间，但不实用）

常见运行时间（4/4）

多项式时间：$T(n) = O(n^k)$

- **大小为 k 的独立集**：给定一个 n 个节点的图，是否存在 k 个节点使得任意两个都不相邻?
- 枚举所有 k 个节点的子集然后检验
  - 检验 S_k 是否为独立集需要 $O(k^2)$ 时间
  - $\#(S_k) = C_n^k \leq n^k/k!$
- $O(k^2 n^k/k!) = O(n^k)$（当 $k = 17$ 时为多项式时间，但不实用）

指数时间：$T(n) = O(c^n)$

- **独立集**：给定一个图，最大独立集的基数是多少?
- 枚举所有子集并检验：$O(n^2 2^n)$

关于多项式运行时间

理想的缩放性质：当输入规模翻倍时，算法只应减慢某个常数因子 c。

关于多项式运行时间

理想的缩放性质：当输入规模翻倍时，算法只应减慢某个常数因子 c。

多项式运行时间满足上述缩放性质

- $T(n) = O(n^d) \leftarrow$ 取 $c = 2^d$

关于多项式运行时间

理想的缩放性质：当输入规模翻倍时，算法只应减慢某个常数因子 c。

多项式运行时间满足上述缩放性质

- $T(n) = O(n^d) \leftarrow$ 取 $c = 2^d$

我们称一个算法是**高效的**，如果它具有多项式运行时间。

关于多项式运行时间

理想的缩放性质：当输入规模翻倍时，算法只应减慢某个常数因子 c。

多项式运行时间满足上述缩放性质

- $T(n) = O(n^d) \leftarrow$ 取 $c = 2^d$

我们称一个算法是**高效的**，如果它具有多项式运行时间。

例外：一些多项式时间算法确实有很高的常数和/或指数，在实践中无用。
问题：你更喜欢哪个？ $20n^{100}$ vs. $n^{1+0.02 \ln n}$

本讲小结（1/2）

- 介绍算法的抽象定义
- 如何刻画算法的复杂度?
  - 第一个简化：用函数表示基本计算步骤数与输入规模的关系
- 如何比较函数?
  - 第二个简化：大 O 记号（五种标准渐近记号）刻画函数的阶。我们研究了定义、典型用法、示例、性质

大 O 记号让我们关注全局。

- 有用的类比：$O(\leq)$，$\Omega(\geq)$，$\Theta(=)$，$o(\ll)$，$\omega(\gg)$

本讲小结（2/2）

- 研究重要的运行时间函数和经典算法示例

记号滥用：$O(g(n))$ 是一个函数集合，但计算机科学家经常写 $f(n) = O(g(n))$ 而不是 $f(n) \in O(g(n))$。
底线：可以滥用记号，但不可误用。

不要误解这种对常数的随意态度。程序员非常关心常数，愿意熬夜只为获得 5% 的效率提升。

常数改进很重要（勿以善小而不为）

理论突破太～～～～难了！