Chapter 1

算法复杂性

回答“这样一个程序有多快”这个问题的明显方法是使用 UNIX 时间戳命令来直接查找。这个简单的答案可能有几种反对意见。程序所需的时间是输入函数，因此我们可能需要计算时间的几个实例命令并推断出结果。但是，有些程序对大多数人来说都很好输入，但有时需要很长时间；我们如何报出（事实上，我们怎么做一定要注意）这样的异常？我们如何处理我们的所有没有测量结果的输入。我们如何有效地应用在一台机器上收集的结果到另一台机器。

测量原始时间的麻烦在于信息是精确的，却是受限制的：此机器此配置的输入时间。在另一个机器的指令花费不同的绝对或相对时间，数字不一定适用。 实际上，假设我们比较两个不同的程序在相同的输入和同一台机器上做同样的事情。 程序 A 可能得到结果比程序 B 更快。但这并不意味着程序 A 会要比 B 快，当它们在其他输入上运行时，或者在相同的输入，不同的机器上运行。

在数学中，我们可以说原始时间是函数 $C_{\gamma}(I,P,M)$，在某些特定输入 I 下的值，某些程序 P 和某些“平台”M（这里的平台是一个机器，操作系统，编译器和运行库所支持）。 我在这里发明了函数 Cr，意思是“原始花费时间”，通过总结所有特的大小的输入，我们可以让这个数字更具信息性

$$\mathrm{C}_{\varpi}(N,P,M)=\max C_{r}(I,P,M),\left|I\right|=N$$

$|I|$ 表示输入I的“大小”。如何定义大小取决于问题：如果 I 是被排序的数组，例如，$|I|$ 可能表示 $I.length$。 我们说 Cw 衡量一个项目的最坏情况时间。 当然，给定大小输入的值可能非常大（5 个整数的数组，例如，$2^{160}>10^{48}$ 是），我们不能直接测量 Cw，但我们也许可以估计它借助于对 P 的一些分析。通过了解最坏情况的时间，我们可以做到关于程序运行时间的保守陈述：如果是最坏情况输入大小 N 的时间是 T，那么我们保证 P 不再消耗比任何输入大小为 N 的输入的时间 T。

但是，当然，我们的程序总是可以在大多数输入上正常工作，但是在一两个（不太可能的）输入上需要很长时间。 在这种情况下，我们可能会声称 Cw 是一个过于苛刻的总结措施，我们应该真的看一下平均时间。 假设输入的所有值 I 都与平均值相同时间是

$$C_{a}(N,P,M)=\frac{\sum_{|I|=N}C_{r}(I,P,M)}{N}$$

这可能是公平的，但通常很难计算。 因此，在这个课程中，我会对平均情况说很少，把它留给你的下一个算法课程。

我们通过考虑最坏情况时间总结了输入;现在让我们考虑一下我们如何总结机器。就像总结输入一样要求我们放弃一些信息 - 即特定的表现输入 - 如此总结机器需要我们放弃信息在特定机器上的精确性能。假设有两种不同的型号计算机正在运行（不同的翻译）相同

的程序，执行相同的步骤以相同的顺序。虽然它们以不同的速度运行，但可能执行不同数量的指令，执行任何指令的速度特定步骤往往因某些常数因素而不同。通过采取这些常数因素中最大和最小的一个，我们可以把它们的差异放在一边总体执行时间。 （这个论点不是很简单，而是出于我们的目的在这里，它就足够了。）也就是说，任何两程序都是同一个程序的时间在所有可能的情况下，平台的差异不会超过一些常数因素投入。如果我们可以在一个平台上确定程序的时间安排，我们就可以使用它，对于所有其他人而言，我们的结果“只会受到一个恒定因素的影响”。

但是，当然，1000 是一个常数因素，你通常不会麻木不仁，事实上，品牌 X 计划比品牌 Y 慢 1000 倍。然而，有一个重要的例子，这种表征是有用的：也就是说，当我们试图确定或比较算法的表现时—用于执行某项任务的理想化程序。算法和程序之间的区别（具体的，可执行的程序）有些模糊。最更高级的编程语言允许人们编写看起来非常好的程序很像他们应该实现的算法。区别在于细节水平。根据对“集合”的操作而施加的过程没有给出这些集合的具体实现，可能有资格作为算法。在谈论理想化的程序时，它并没有多大意义谈谈他们执行的秒数。相反，我们感兴趣在我可能称之为算法行为的形状中：诸如“如果我们输入的大小加倍，执行时间会发生什么？“鉴于那种情况问题，用于衡量绩效的特定时间单位（或空间）算法是不重要的 - 常数因素无关紧要。

如果我们只关心将算法的速度表征在一个内恒定因子，其他简化是可能的。我们不再需要担心算法中每个小语句的时间，但可以使用测量时间任何方便的“标记步骤”。例如，要进行十进制乘法运算标准方式，你将被乘数的每个数字乘以乘数的每个数字，并对这些一位数乘法中的每一个执行大约一位数的加法运算。因此，只计算一位数乘法给你恒定因子内的时间，这些乘法很容易计数（操作数中的位数的乘积）。

算法复杂性研究中的另一个特征假设（即，算法的时间或内存消耗量）是我们对典型感兴趣的理想化程序在整个可能输入集上的行为。理想化程序,当然，是理想的，可以在任何可能大小的输入上操作，并且大多数理想的数学世界中“可能的大小”非常大。因此，在这种分析，传统的不是对某个特定事实感兴趣算法非常适合小输入，而是在输入变得非常大时考虑其行为“取极限”。例如，假设有人想要分析用于将π计算到任何给定小数位数的算法。我可以通过简单存储，使任何算法看起来都适合输入，例如 1,000,000 数组中的前 1,000,000 个数字 π 并使用它来提供答案当请求 1,000,000 或更少的数字时。如果你注意我的方式为高达 1,000,000 的输入执行的程序，你可能会被严重误导我算法的聪明才智。因此，在研究算法时，我们来看看它们的渐近行为 - 它们在输入大小时的行为如何变为无穷大。

所有这些考虑的结果是考虑时间复杂性算法，我们可以选择任何特定的机器，并计算任何方便标记步骤，我们尝试找到真正渐近的特征到无穷远。 这意味着我们对算法的典型复杂度测量将会具有形式 Cw（N，A） - 意味着“在所有大小为 N 的输入上的最坏情况时间”算法 A（在某些单元中）。“因为算法将被理解为任何特别是讨论，我们通常只会写 Cw（N）或类似的东西。所以我们需要首先描述算法的复杂性，这是一种表征算法的方法函数的渐近行为。

1.1 渐近复杂度分析和顺序表示法

碰巧的是，有一种方便的符号工具 - 统称为命令“增序”的表示法 - 用于描述函数的渐近行为。它可以（并且）用于任何类型的整数或实值函数 - 不仅仅是复杂功能。 例如，您可能已经看到它在微积分课程中使用过。

我们写

$$f(n) \in O(g(n))$$

（注意，这是“f（n）在 g（n）中的大 O”）意味着函数 f 最终是什么。

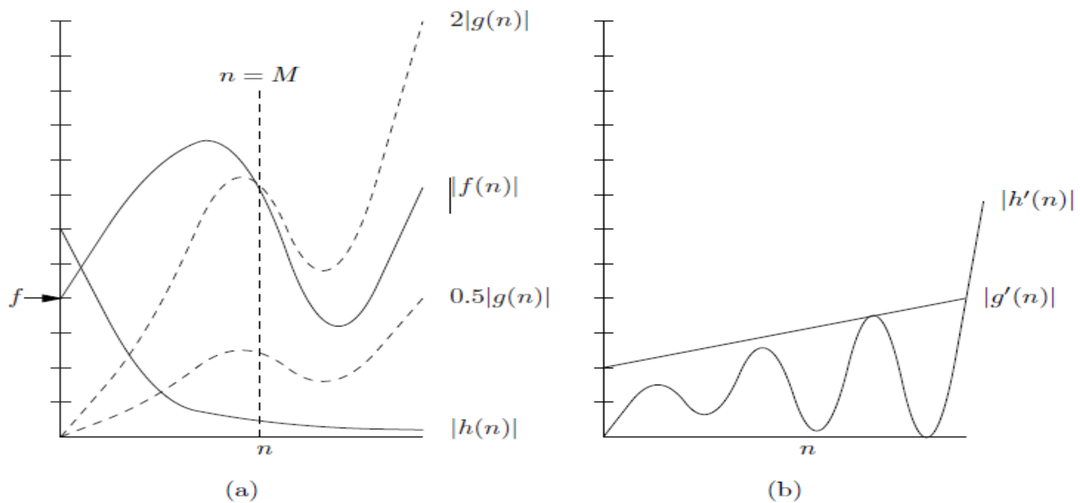

图 1.1：大写符号的图示。在图（a）中，我们看到|f(n)|≤2 | g(N)|，因为 n> M，所以 f（n）∈O(g(n))（K = 2）。同样，h(n)∈O(g(n))，说明 g 可能是一个非常收敛的界限。函数 f 也是有界的，下面两个 g（例如，K = 0.5，M 任何大于 0 的值）和 H。也就是说，f(n)Ω∈(g(n)) and f(n) ∈ Ω(h(n))。因为 f 高于和低于 g 的数倍，我们说 f(n)∈θ(g(n)).另一方面，h(n)∉ Ω (g(n)).事实上，假设 g 继续如图所示增长并且 h 收缩，h（n）∈o（g（n））。图（b）表明 o（·）不仅仅是集合的补集 Ω·(·); h'（n）6∈（G'（N）），但是 h'（n）6∉o(g'(n))。

由| g（n）|的一些倍数限制。更确切地说，f(n) ∈ O(g(n))

$$|f(n)| \leqslant K \cdot |g(n)|, n > M,$$

对于某些常数 K> 0 和 M。也就是说，O（g（n））是一组函数“增长不会比”| g（n）| 当 n 变得足够大时。 有些令人困惑的是，这里的 f（n）并不意味着“通常将应用于 f 的结果“确实。 相反，它被解释为参数为 n 的函数体。因此，我们经常把像 O（n2）这样的东西写成“所有增长函数的集合没有比他们的论点的平方更快 1。“图 1.1a 给出了一个直观的在 O（g（n））中意味着什么的想法。

说 f（n）∈O（g（n））只给出了 f 的行为的上界。例如，图 1.1a 中的函数 h - 就此而言，到处都是 0 的函数 - 都在 O（g（n））中，但肯定不会像 g 一样增长。因此，我们定义 f（n）∈（g（n））iff 对于所有 n> M，| f（n）| ≥K| g（n）| 对于 n> M，对于一些常数 K> 0 和 M.也就是说，（g（n））是所有函数的集合“增长速度至少和 g 超过某一点一样快。一个小代数足以显示出来 O（·）和。之间的关系·)：

$$|f(n)| \geqslant K|g(n)| \equiv |g(n)| \leqslant (1/K) \cdot |f(n)|$$

$$f(n) \in \Omega(g(n)) \Longleftrightarrow g(n) \in O(f(n))$$

由于我们对恒定因子的骑士处理，它可能具有一种功能 f（n）由另一个函数 g（n）在上下限定：f（n）∈O（g（n））和 f（n）∈Ω（G（N））。 为简洁起见，我们写 f（n）属于θ（g（n）），因此θ（g（n））=O（g（n））=Ω（g（N））。

仅仅因为我们知道 f（n）∈O（g（n）），我们不一定知道 f（n）比 g（n）小得多，或者甚至（如图 1.1a 所示）得到它比 g（n）小。 我们偶尔会想说“h（n）”之类的话与 g（n）相比，它变得可以忽略不计。“你有时会看到符号 h（n）«g（n），意思是“h（n）远小于 g（n）”，但这可能适用于这样的情况 h（n）= 0.001g（n）。 我们不需要对这样的恒定因素感兴趣更强的东西。 传统的符号是“小 o”，定义如下。

$$h(n) \in o(g(n)) \Longleftrightarrow \frac{lim}{n\to\infty} h(n)/g(n) = 0.$$

很容易看出，如果 h（n）∈o（g（n）），那么 h（n）6∉Ω（g（N））；没有常数 K 可以在定义中工作Ω（·）。 然而，事实并非如此在...之外 g（n））必须在 o（g（n））中，如图 1.1b 所示。

1.2 例子

您可能已经在微积分课程中看到过大 O 符号。 例如，泰勒定理告诉我们 2（在适当条件下）

$$f(x) = \underbrace{\frac{x^n}{n!} f^{[n]}(y)}_{\text{error term}} + \underbrace{\sum_{0 \le k < n} f^{[k]}(0) \frac{x^k}{k!}}_{\text{approximation}}$$

对于 0 和 x 之间的某些 y，其中 f [k]表示 f 的 n 阶导数。 因此，如果 g（x）表示 0 和 x 之间 f [n]的最大绝对值，那么我们也可以将错误术语写为

$$\begin{aligned} f(x) - \sum_{0 \le k < n} f^{[k]}(0) \frac{x^k}{k!} \\ \in \quad O\left(\frac{x^n}{n!} g(x)\right) = O(x^n g(x)) \end{aligned}$$

$f(n)$	*Is contained in*	*Is* not *contained in*
$1,\ 1+1/n$	$O(10000),\ O(\sqrt{n}),\ O(n),$ $O(n^2),\ O(\lg n),\ O(1-1/n)$ $\Omega(1),\ \Omega(1/n),\ \Omega(1-1/n)$ $\Theta(1),\ \Theta(1-1/n)$ $o(n),\ o(\sqrt{n}),\ o(n^2)$	$O(1/n),\ O(e^{-n})$ $\Omega(n),\ \Omega(\sqrt{n}),\ \Omega(\lg n),\ \Omega(n^2)$ $\Theta(n),\ \Theta(n^2),\ \Theta(\lg n),\ \Theta(\sqrt{n})$ $o(100+e^{-n}),\ o(1)$
$\log_k n,\ \lfloor \log_k n \rfloor,$ $\lceil \log_k n \rceil$	$O(n),\ O(n^\epsilon),\ O(\sqrt{n}),\ O(\log_{k'} n)$ $O(\lfloor \log_{k'} n \rfloor),\ O(n/\log_{k'} n)$ $\Omega(1), \Omega(\log_{k'} n), \Omega(\lfloor \log_{k'} n \rfloor)$ $\Theta(\log_{k'} n),\ \Theta(\lfloor \log_{k'} n \rfloor),$ $\Theta(\log_{k'} n + 1000)$ $o(n),\ o(n^\epsilon)$	$O(1)$ $\Omega(n^\epsilon), \Omega(\sqrt{n})$ $\Theta(\log^2_{k'} n), \Theta(\log_{k'} n + n)$
$n,\ 100n+15$	$O(.0005n-1000),\ O(n^2),$ $O(n \lg n)$ $\Omega(50n+1000),\ \Omega(\sqrt{n}),$ $\Omega(n+\lg n),\ \Omega(1/n)$ $\Theta(50n+100),\ \Theta(n+\lg n)$ $o(n^3),\ o(n \lg n)$	$O(10000),\ O(\lg n),$ $O(n-n^2/10000),\ O(\sqrt{n})$ $\Omega(n^2),\ \Omega(n \lg n)$ $\Theta(n^2),\ \Theta(1)$ $o(1000n),\ o(n^2 \sin n)$
$n^2,\ 10n^2+n$	$O(n^2+2n+12),\ O(n^3),$ $O(n^2+\sqrt{n})$ $\Omega(n^2+2n+12),\ \Omega(n),\ \Omega(1),$ $\Omega(n \lg n)$ $\Theta(n^2+2n+12),\ \Theta(n^2+\lg n)$	$O(n),\ O(n \lg n),\ O(1)$ $o(50n^2+1000)$ $\Omega(n^3),\ \Omega(n^2 \lg n)$ $\Theta(n),\ \Theta(n \cdot \sin n)$
n^p	$O(p^n),\ O(n^p+1000n^{p-1})$ $\Omega(n^{p-\epsilon}),$ $\Theta(n^p+n^{p-\epsilon})$ $o(p^n),\ o(n!),\ o(n^{p+\epsilon})$	$O(n^{p-1}),\ O(1)$ $\Omega(n^{p+\epsilon}),\ \Omega(p^n)$ $\Theta(n^{p+\epsilon}),\ \Theta(1)$ $o((n+k)^p)$
$2^n,\ 2^n+n^p$	$O(n!),\ O(2^n-n^p),\ O(3^n),\ O(2^{n+p})$ $\Omega(n^p),\ \Omega((2-\delta)^n),$ $\Theta(2^n+n^p)$ $o(n2^n),\ o(n!),\ o(2^{n+\epsilon}),\ o((2+\epsilon)^n)$	$O(n^p),\ O((2-\delta)^n)$ $\Omega((2+\epsilon)^n),\ \Omega(n!)$ $\Theta(2^{2n})$

表 1.1: 订单关系的一些例子。 在上面，除了 n 之外的名字，表示常数，ϱ>0,0≤δ≤1，p> 1， k， k'> 1。

1.3。算法分析应用

对于确定的 n 当然，这比原始声明要弱得多（它允许错误要比实际大得多）。

你会经常看到像这样的语句用一点代数操作：

$$f(x) \in \sum_{0 \le k < n} f^{[k]}(0)\frac{x^k}{k!} + O(x^n g(x)).$$

为了理解这种陈述，我们定义了加法（等等）在函数集（a，b 等）和函数集（A，B 等）之间：

$$\begin{aligned} a+b &= \lambda x.a(x)+b(x) \\ A+B &= \{a+b \mid a \in A,\ b \in B\} \\ A+b &= \{a+b \mid a \in A\} \\ a+B &= \{a+b \mid b \in B\} \end{aligned}$$

类似的定义适用于乘法，减法和除法。 所以如果 a 是√x 并且 b 是 lg x，那么 a + b 是一个函数，其值为 √x + lg x 为每个（可能的）x。 O（a（x））+ O（b（x））（或只是 O（a）+ O（b））是你可以得到的一组函数通过将 O（√x）的成员添加到 O（lg x）的成员中。 例如，O（a）包含 5√x+ 3 和 O（b）含有 0.01 lg x-16，因此 O（a）+ O（b）含有 5√x+ 0.01 lg k-13，在更多函数中中。

1.2.1 论证“大 O”

假设我们想要显示 5n2 +10√n∈O（n2）。 也就是说，我们需要找到 K.和 M，如下：

$$|5n^2 + 10\sqrt{n}| \le |Kn^2|, \quad \text{for } n > M.$$

我们意识到 n2 的增长速度比√n 快，所以它最终会大于 10√n。 所以也许我们可以取 K = 6 并找到 M> 0，如下：

$$5n^2 + 10\sqrt{n} \le 5n^2 + n^2 = 6n^2$$

为了得到 10√n<n2，我们需要 10 <n3 / 2，或 n> 102 /3≈4.7。 所以选择 M> 5 当然有效。

1.3 算法分析的应用

在本课程中，我们通常会处理由此产生的整数值函数衡量算法的复杂性。 表 1.1 给出了一些常见的例子我们处理的订单及其遏制关系，以及以下部分举例说明使用它们的简单算

法分析。

1.3.1 线性搜索

让我们将所有这些应用于特定程序。 这是一个尾递归线性搜索用于查看特定值是否在排序数组中：

```
/** True iff X is one of A[k]...A[A.length-1].
 *  Assumes A is increasing, k>= 0. */
static boolean isIn (int[] A, int k, int X) {
    if (k >= A.length)
        return false;
    else if (A[k] > X)
        return false;
    else if (A[k] == X)
        return true;
    else
        return isIn (A, k+1, X);
}
```

这本质上是一个循环。作为其复杂性的衡量标准，让我们定义 CisIn（N）作为 k = 0 和 0 的呼叫执行的最大指令数 A.length = N.通过检查，您可以看到这样的调用将执行第一个 if 测试高达 N + 1 次，第二次和第三次高达 N 次，并且尾部递归呼叫 isIn 最多 N 次。使用一个 compiler3，每次递归调用 isIn 都会执行在返回或尾递归调用 isIn 之前最多 14 条指令。最初的呼叫执行 18.总共提供最多 14N + 18 条指令。如果相反，我们计算比较次数 k> = A.length，我们得到最多 N + 1.如果算数对 X 的比较次数或 A [0]的提取次数，我们得到大多数 2N。因此，我们可以说功能最大处理大小为 N 的输入所需的时间是 O（14N + 18），O（N + 1），或 O（2N）。但是，这些都是相同的集合，实际上所有都等于 O（N）。因此，我们可能会抛弃所有那些混乱的整数并将 CisIn（N）描述为在 O（N）中，因此说明了忽略常数因子的简化能力。

这个界限是最坏的情况。 对于 X <= A [0]的所有参数，isIn 函数在恒定时间内运行。 那个时间界限 - 最好的情况 - 很少非常有用，特别是当它适用于非典型的输入时。

给予 CisIn（N）约束的 O（·）并不告诉我们 isIn 必须花时间即使在最坏的情况下，也与 N 成正比，只是它不再需要。 在这然而，特殊情况下，上面使用的论证表明最坏的情况，是在事实上，至少与 N 成正比，这样我们也可以说 CisIn（N）属于（N）。将两个结果放在一起，CisIn（N）∈θ（N）。

通常，然后，渐近分析给定的空间或时间算法涉及以下内容。

- 决定输入大小的适当度量（例如，长度为数组或列表）。
- 选择要衡量的代表性数量 - 与之成比例的数量所需的“真实”空间或时间。
- 通常在最坏的情况下，提出一个或多个约束我们决定的数量的函数测量。
- 通过给出 O（·），Ω（·），或者θ（·）他们的特征，可能总结这些函数。

1.3.2 二次例子

这里有一些用于排序整数的代码：

```
static void sort (int[] A) {
   for (int i = 1; i < A.length; i += 1) {
     int x = A[i];
     int j;
     for (j = i; j > 0 && x < A[j-1]; j -= 1)
       A[j] = A[j-1];
     A[j] = x;
   }
}
```

如果我们将 Csort（N）定义为最坏情况的次数，那么比较 x <A [j-1]对于 N = A.length 执行，我们看到 i 的每个值从 1 到 A.length-1，程序在内循环（在 j 上）最多执行 i 次比较。因此，

$$\begin{aligned} C_{\texttt{sort}}(N) &= 1 + 2 + \ldots + N - 1 \\ &= N(N-1)/2 \\ &\in \Theta(N^2) \end{aligned}$$

这是嵌套循环的常见模式。

1.3.3 爆炸的例子

考虑具有以下形式的函数。

```
static int boom (int M, int X) {
    if (M == 0)
        return H (X);
    return boom (M-1, Q(X))
         + boom (M-1, R(X));
}
```

并且假设我们想要计算 Cboom（M） - 要求 Q 的次数在最坏的情况下给定的 M. 如果 M = 0，则为 0.如果 M> 0，则执行 Q.一旦计算出第一次递归调用的参数，然后它被执行，然而很多时候，两个内部呼唤的繁荣与 M - 1 的争论执行它。换一种说法，

$$\begin{aligned} C_{\texttt{boom}}(0) &= 0 \\ C_{\texttt{boom}}(i) &= 2C_{\texttt{boom}}(i-1) + 1 \end{aligned}$$

简单数学推导：

$$
\begin{aligned}
C_{\texttt{boom}}(M) &= 2C_{\texttt{boom}}(M-1)+1, \\
&\quad \text{for } M \geq 1 \\
&= 2(2C_{\texttt{boom}}(M-2)+1)+1, \\
&\quad \text{for } M \geq 2 \\
&\vdots \\
&= \underbrace{2(\cdots(2}_{M} \cdot 0 + \underbrace{1)+1)\cdots+1}_{M} \\
&= \sum_{0 \leq j \leq M-1} 2^j \\
&= 2^M - 1
\end{aligned}
$$

所以 $C_{boom}(M) \in \theta(2^M)$。

1.3.4 分治法

当递归调用减小参数的大小时，事情变得更有趣，通过乘法而不是加法因子。 例如，考虑一下二分搜索。

```
/** Returns true iff X is one of
  * A[L]...A[U]. Assumes A increasing,
  * L>=0, U-L < A.length. */
static boolean isInB (int[] A, int L, int U, int X) {
  if (L > U)
    return false;
  else {
    int m = (L+U)/2;
    if (A[m] == X)
      return true;
    else if (A[m] > X)
      return isInB (A, L, m-1, X);
    else
      return isInB (A, m+1, U, X);
  }
}
```

这里的最坏情况时间取决于所考虑的 A 元素的数量，U - L + 1，我们称之为 N.让我们使用第一行的次数作为成本执行，因为如果执行了剩余的正文，则第一行也是如此必须被执行。如果 N> 1，则执行 isInB 的成本是 L 和 U 的 1 比较，后跟执行 isInB 的成本要么是⌊（N - 1）/2⌋，要么是⌈（N - 1）/2⌉为 N 的新值。 数量不超过⌈（N - 1）/2⌉。如果 N≤1，则在最坏的情况下对 N 进行两次比较。

因此，以下重复描述了此函数执行的成本 CisInB（i）当 U - L + 1 = i 时。

$$
\begin{aligned}
C_{\texttt{isInB}}(1) &= 2 \\
C_{\texttt{isInB}}(i) &= 1 + C_{\texttt{isInB}}(\lceil (i-1)/2 \rceil), \quad i > 1.
\end{aligned}
$$

这有点难以处理，所以让我们再次做出合理的假设成本函数的值，无论它是什么，必须随着 N 的增加而增加。 然后我们可以计算一个成本函数，C'isInB 略大于 CisInB，但是更容易计算。

$$
\begin{aligned}
C'_{\texttt{isInB}}(1) &= 2 \\
C'_{\texttt{isInB}}(i) &= 1 + C'_{\texttt{isInB}}(i/2), \quad i > 1 \text{ a power of } 2.
\end{aligned}
$$

这是对 CisInB 的轻微高估，但仍然允许我们计算上限界限。 此外，C'isInB 仅定义为 2 的幂，但是因为 isInB 的随着 N 增加，成本增加，我们仍可以保守地约束 CisInB（N）计算下一个更高效率的 C'isInB 2。再次推导:

$$
\begin{aligned}
C'_{\texttt{isInB}}(i) &= 1 + C'_{\texttt{isInB}}(i/2), \quad i > 1 \text{ a power of } 2. \\
&= 1 + 1 + C'_{\texttt{isInB}}(i/4), \quad i > 2 \text{ a power of } 2. \\
&\vdots \\
&= \underbrace{1 + \cdots + 1}_{\lg N} + 2
\end{aligned}
$$

数量 lgN 是以 2 为底 N 的对数，或者大约是 1 的次数在达到 1 之前可以将 N 除以 2。总之，我们可以说 CisIn（N）∈O（lgN）。同样，实际上可以推导出 CisIn（N）∈（lgN）。

1.3.5 二分到停滞

现在考虑一个包含两个递归调用的子程序。

```
static void mung (int[] A, L, U) {
    if (L < U) {
        int m = (L+U)/2;
        mung (A, L, m);
        mung (A, m+1, U);
    }
}
```

我们可以像以前一样用 N / 2 逼近两个内部调用的参数，结束以下近似，Cmung（N），调用 mung 的成本参数 N = U -L + 1（我们正在计算测试中的次数第一行执行）。

$$
\begin{aligned}
C_{\texttt{mung}}(1) &= 1 \\
C_{\texttt{mung}}(i) &= 1 + 2C_{\texttt{mung}}(i/2), \quad i > 1 \text{ a power of } 2.
\end{aligned}
$$

$$
\begin{aligned}
C_{\texttt{mung}}(N) &= 1 + 2(1 + 2C_{\texttt{mung}}(N/4)), \quad N > 2 \text{ a power of } 2. \\
&\vdots \\
&= 1 + 2 + 4 + \ldots + N/2 + N \cdot 3
\end{aligned}
$$

这是几何系列的总和（1 + r + r^2 +…+ r^m），稍加一点上。 几何系列的一般规则是

$$
\sum_{0 \le k \le m} r^k = (r^{m+1} - 1)/(r - 1)
$$

所以，让 r=2，

$$
C_{mung}(N) = 4N-1
$$

或者 $C_{mung}(N) \in \theta(N)$

1.4 回顾

到目前为止, 在确定的函数中, 我们已经考虑了单个操作或者单独调用的时间消耗。然而，有时候考虑整个调用序列的成本是富有成效的，特别是当每个调用影响以后的调用成本时。

例如，考虑一个简单的二进制计数器。 增加此计数器会导致它要通过这样的序列：

```
0 0 0 0 0
0 0 0 0 1
0 0 0 1 0
0 0 0 1 1
0 0 1 0 0
 ...
0 1 1 1 1
1 0 0 0 0
 ...
```

每个步骤包括翻转一定数量的位，将位 b 转换为 1-b。更准确地说，从一个步骤到另一个步骤的算法是

增量：将计数器的位从右向左、向上和包括遇到前 0 位（如果有）。

显然，如果要求我们对增量成本给出最坏情况的约束对于 N 位计数器（翻转次数）的操作，我们不得不说它是θ（N）：所有位都可以翻转。 只需使用那个界限，我们就得说执行 M 增量运算的成本是θ（M·N）。

但是连续增量操作的成本是相关的。 例如，如果一个增量翻转多个位，下一个增量将始终完全翻转一个（为什么？）。 事实上，如果你考虑位变化的模式，你会看到单位（最右边）位在每个增量上翻转，每秒增加 2 位，每四个增量上的 4 位，一般来说，每隔就2^k有2^k位增量。 因此，从 M 个连续增量的任何序列开始，从 0 开始，会有

$$
\begin{aligned}
& \underbrace{M}_{\text{unit's flips}} + \underbrace{\lfloor M/2 \rfloor}_{\text{2's flips}} + \underbrace{\lfloor M/4 \rfloor}_{\text{4's flips}} + \ldots + \underbrace{\lfloor M/2^n \rfloor}_{2^n\text{'s flips}}, \text{ where } n = \lfloor \lg M \rfloor \\
= \; & \underbrace{2^n + 2^{n-1} + 2^{n-2} + \ldots + 1}_{=2^{n+1}-1} + (M - 2^n) \\
= \; & 2^n - 1 + M \\
< \; & 2M \text{ flips}
\end{aligned}
$$

换句话说，如果我们执行 M 增量，这与我们得到的结果相同每个最坏情况下的成本为 2 翻，而不是 N.我们称之为 2 翻增量的摊余成本。在算法的上下文中分摊是为了按顺序处理每个单独操作的成本，就好像它是分散的一样在序列中的所有操作中。任何特定的增量都可能占用 N 翻转，但我们认为，因为 N / M 翻转记入每个增量操作序列（同样计算每个增量只需一次翻转为 1 / M 翻转对于每个增量操作）。结果是我们对如何获得更现实的想法整个计划需要花费很多时间;简单地乘以普通的最坏情况 M 的时间给了我们一个非常宽松和悲观的估计。摊销成本也不算高与平均成本相同;这是一个更强有力的措施。如果某个操作有给定的平均成本，留下了一些不太可能的序列的可能性输入会使它看起来很糟糕。受限于摊销的最坏情况成本另一方面，无论输入如何，都保证保持不变。

达到相同结果的另一种方法是使用所谓的潜在方法这里的想法是我们与我们的数据结构相关联（在这种情况下我们的位序列）一种非负面的潜力，代表着我们希望在几项行动中

分散的工作。如果c_i表示我们数据结构上第 i 个操作的实际成本，我们定义了第 i 个操作的摊余成本，a_i就是这样

$$a_i = c_i + \Phi_{i+1} - \Phi_i, \tag{1.1}$$

其中Φi 表示在第 i 次操作之前的储存潜力。 也就是说，我们给予我们自己选择在任何给定的操作上增加一点并对此充电增加Φ对抗 ai，导致 ai> ci 当Φ增加时。 或者，我们也可以通过减少Φ操作来减少 ai 低于 ci，实际上是先前使用了保存增加。 假设我们从Φ0 = 0 开始，n 次操作的总成本是

$$\begin{aligned} \sum_{0\le i<n} c_i &\le \sum_{0\le i<n} (a_i + \Phi_i - |\Phi_i + 1) \\ &= (\sum_{0\le i<n} a_i) + \Phi_0 - \Phi_n \\ &= (\sum_{0\le i<n} a_i) - \Phi_n \\ &\le \sum_{0\le i<n} a_i, \end{aligned} \tag{1.2}$$

因为我们要求Φi⩾0。 因此，这些 ai 提供了每个点的操作累积成本的保守估计。

例如，通过我们的位翻转示例，我们将Φi 定义为总数在第i^{th}次操作之前的 1 位。 第i^{th}个增量的成本总是一加翻转回 0 的 1 位，由于我们如何定义它，永远不会超过我（当然永远不会消极）。 所以定义 ai = 2 对于每个操作满足公式 1.1，再次证明我们可以约束以 2 比特翻转递增的摊销成本。

假设我们的位计数器始终从 0 开始，我在这里傻一下。如果它在 > 0 开始，我们在一次增量后停止，然后是总数成本（以翻转为单位）可能高达 1 +lg（N0 + 1）.因为我们要保险不等式 1.2 适用于任何 n，我们必须做一些调整来处理这个案例。 一个简单的技巧是重新定义Φi = 0，保持 i 的其他值相同（第 i 次操作前的 1 位数，最后定义 a0 = c0 + Φ1 效果，我们用计数序列的启动成本收取 a0。 当然，这个意味着 a0 可以任意大，但这仅仅反映了现实; 剩下的 ai 仍然不变。

1.5 问题的复杂性

到目前为止，我只讨论了算法复杂性的分析。 算法，然而，这只是解决某些问题的一种特殊方式。 因此我们可能会考虑要求对问题的复杂性进行复杂性限制。 也就是说，我们可以限制了最佳算法的复杂性？ 显然，如果我们有一个特别的算法及其时间复杂度为 O（f（n）），其中 n 是输入的大小，那么最好的算法的复杂性也必须是 O（f（n））。 我们称之为 f（n），因此，f（n）是最佳算法的（未知）复杂性的上限。 但这并没有告诉我们最佳算法是否是最佳算法比任何更快的速度都没有给出最佳算法所需时间的下限。例如，isln 的最坏情况时间是θ（N）。 但是，islnB 是快多了。 实际上，人们可以证明，如果算法能够获得唯一的知识是 X 与数组元素之间进行比较的结果，然后是 islnB 有最好的约束（它是最优的），以便找到一个完整的问题有序数组中的元素具有最坏情况时间θ（lgN）。

简单地说明执行某些问题所需的时间上限涉及找到问题的算法。 相比之下，放低下限所需的时间要困难得多。 我们基本上必须证明没有算法可以比我们的绑定有更好的执行时间，无论多少算法设计师比我们更聪明。 当然，琐碎的下界是容易：每个问题的最坏情况时间都是Ω（1），以及任何问题的最坏情况时间谁的答案取决于所有的数据Ω（N），假设一个人的理想化机器是完全现实的。 然而，比那些更好的下限需要相当一点工作。 让我们

的理论计算机科学家受雇更好。

1.6 对数的一些性质

对数在复杂性分析中经常出现，因此审查可能很有用关于他们的一些事实。 在大多数数学课程中，你会遇到自然对数，ln x = loge x，但计算机科学家倾向于使用以 2 为底对数，lg x = log2 x，一般来说，当我说“对数”时，这就是我的意思。当然，所有的对数通过一个常数因子相关：因为根据定义 aloga x = x = blogb x，它如下所示

$$\log_a x = \log_a b^{\log_b x} = (\log_a b)\log_b x.$$

它们与指数的联系决定了它们熟悉的特性：

$$\begin{aligned} \lg xy &= \lg x + \lg y \\ \lg x/y &= \lg x - \lg y \\ \lg x^p &= p \lg x \end{aligned}$$

在复杂性论证中，我们常常对不平等感兴趣。 对数是一个非常缓慢增长的功能：

$$\lim_{x\to\infty} \lg x/x^p = 0, \text{for all} p > 0.$$

它严格地增加并严格凹入，意味着它的值高于任何线段连接点（x，lg x）和（z，lg z）。用代数法来说，如果 0 <x <y <z，然后

$$lgy > \frac{y-x}{z-x}\lg x + \frac{z-y}{z-x}\lg z.$$

因此，如果 0 <x + y <k，则当 x = y = k / 2 时，lg x + lg y 的值最大化。

1.7 符号注记

其他作者使用诸如 f（n）= O（n2）而不是 f（n）∈O（n2）的符号。 我不因为我认为这是荒谬的。 为了证明使用'='，人们必须要思考 f（n）作为一组函数（它不是），或者将 O（n2）视为单个函数这与 O（n2）的每个单独外观不同（这是奇怪的）。 我看不到使用'∈'的缺点，这是完全有道理的，所以这就是我使用的。
练习：
1.1 展示以下内容，或在指明的地方给出反例。显示某个 O（·）公式为真意味着产生合适的 K 和 M.§1.1 开头的定义。 提示：有时候服用它是有用的您正在比较的两个函数的对数。

a. $O(\max(|f_0(n)|, |f_1(n)|\quad)) = O(f_0(n)) + O(f_1(n))$.

b. 如果 f(n)是关于 n 的多项式，然后 lgf(n)∈O(lgn).

c. O(f(n)+g(n))=O(f(n))+O(g(n)). 这是一个棘手的问题，真的，让你仔细看看这些定义。在什么条件下方程式是正确的？

d. 函数 f（x）> 0 使得 f（x）∉O（x）和 f（x）∉Ω（X）。

e. 有一个函数 f（x），使得 f（0）= 0，f（1）= 100，f（2）= 10000，f（3）=10^6，但

f（n）∈O（n）。

f. $n^3 lgn \in O(n^{3.0001})$.

g. 没有常数 k 使得n^3lgn∈θ（n^k）。

1.2 通过展示反例来显示以下每个错误。 假设 f 和 g 是任何实值函数。

a. $O(f(x) \cdot s(x)) = o(f(x))$, assuming $\lim_{x \to \infty} s(x) = 0$.

b. If $f(x) \in O(x^3)$ and $g(x) \in O(x)$ then $f(x)/g(x) \in O(x^2)$.

c. If $f(x) \in \Omega(x)$ and $g(x) \in \Omega(x)$ then $f(x) + g(x) \in \Omega(x)$.

d. If $f(100) = 1000$ and $f(1000) = 1000000$ then f cannot be $O(1)$.

e. If $f_1(x), f_2(x), \ldots$ are a bunch of functions that are all in $\Omega(1)$, then

$$F(N) = \sum_{1 \le i \le N} |f_i(x)| \in \Omega(N).$$