# 第3章

# 算法分析方法

前面已经分析了如何计算一种算法的时间复杂度，可以看出其计算是非常困难的，渐近符号的引入简化了时间复杂度的分析和计算。本章主要介绍一些常用的算法分析方法。

## 3.1 概率分析

第 1 章已经介绍过算法最好情形和最坏情形的分析，本节将介绍如何分析算法的平均时间复杂度。

平均时间复杂度可以通过计算所有问题实例的运算时间，然后取平均值，从而得到求解某个问题实例所需要的平均基本运算次数。然而，由于问题实例太多，想要通过这种方法计算出每个问题实例的时间显然不现实。这时候借助概率分析(Probabilistic Analysis)，可以方便地分析算法的平均时间复杂度。但是，这种分析方法需要一个假设，即每个问题实例都有同样的概率作为算法的输入而被算法计算。由此可见，要分析算法的平均时间复杂度，必须掌握概率知识，特别是需要了解一些关于求解问题实例输入分布的先验知识。

下面给出一些利用概率知识来分析算法平均时间复杂度的例子。

第 1 章习题 1-9 已经介绍过线性搜索问题，即给定具有 $n$ 个数的数组 $A$，回答是否存在一个指定的数 $x$，如果存在，则给出该数在数组中的位置；如果不存在，则回答不能找到。该问题的求解算法 LinearSearch($A$，$x$)的伪代码具体如下。

```
LinearSearch(A, x)
1    k ← 1
2    while k≤n and x≠A[k] do
3        k← k + 1
4    if k >n then
         return 0
5    else
         return k
```

LinearSearch($A$，$x$)从数组 $A$ 的第一个位置开始寻找，如果找到 $x$ 或者 $k>n$，则退出 **while** 循环。对于这个算法，其最好情形时间复杂度为 $\Omega(1)$，最坏情形时间复杂度为 $O(n)$。下面分析该算法的平均情形时间复杂度。为了便于分析，假设数组 $A$ 的元素为 1～

$n$ 的整数，而且两两互不相同。事实上，这种假设不失一般性，这是由于数组的下标是两两不同的。即使某些元素相同，但它们仍然处在数组的不同位置。下面，考虑数组 $A$ 含有元素 $x$，即能成功搜索到 $x$ 的情形。

数组 $A$ 共有 $n$ 个位置，要搜索的元素 $x$ 有可能在任何一个位置出现。假设元素 $x$ 在 $n$ 个位置中任意一个位置出现的概率相同，那么在任意一个位置 $k$ 上，元素 $x$ 被找到的概率为 $\frac{1}{n}$，如果元素在位置 $k$ 被找到，则算法执行比较运算的次数为 $k$。由此可见，LinearSearch($A$，$x$)的时间复杂度由比较的次数决定，找到元素 $x$ 的平均比较次数为

$$T(n)=\sum_{k=1}^{n}k\ \frac{1}{n}=\frac{1}{n}\sum_{k=1}^{n}k=\frac{1}{n}\ \frac{n(n+1)}{2}=\frac{n+1}{2}$$

整个算法的平均时间复杂度由平均比较次数决定，因此 LinearSearch($A$，$x$)的平均情形时间复杂度为 $\Theta(n)$。

第 1 章介绍了插入排序算法，并分析了其最好情形时间复杂度和最坏情形时间复杂度，下面分析插入排序算法的平均情形时间复杂度。为了便于分析，仍然假设数组 $A$ 中的元素为 1～$n$ 的整数，而且两两互不相同。

任意给定数组 $A$，其排列的数目为 $n!$。在 $n!$ 种排列中，任意一种排列具有同样的概率作为算法的输入。考虑当前迭代要插入的元素为 $A[j]$，插入排序算法的目的是将 $A[j]$ 插入 $A[1]$～$A[j]$ 共 $j$ 个位置中的任何一个位置，因此 $A[j]$ 插入任何一个位置的概率为 $\frac{1}{j}$。插入排序算法只涉及比较和插入操作，因而算法的时间复杂度主要由比较次数决定。假设 $A[j]$ 插入 $A[k]$，那么算法需要比较的次数如下。如果 $k=1$，则算法需要比较的次数为 $j-1$；如果 $1<k\leqslant j$，则算法需要比较的次数为 $j-k+1$，因此 $A[j]$ 插入 $A[1]$～$A[j]$ 任何一个位置的平均比较次数为

$$\frac{j-1}{j}+\sum_{k=2}^{j}\frac{j-k+1}{j}=\frac{j-1}{j}+\sum_{k=1}^{j-1}\frac{k}{j}=\frac{j}{2}-\frac{1}{j}+\frac{1}{2}$$

由于共迭代 $n-1$ 次，因此，整个算法需要的平均比较次数为

$$\sum_{j=2}^{n}\left(\frac{j}{2}-\frac{1}{j}+\frac{1}{2}\right)=\frac{n(n+1)}{4}-\frac{1}{2}-\sum_{j=2}^{n}\frac{1}{j}+\frac{n-1}{2}=\frac{n^2}{4}+\frac{3n}{4}-\sum_{j=1}^{n}\frac{1}{j}$$

故插入排序算法的平均情形时间复杂度为 $\Theta(n^2)$。

考虑一个更实际的问题：假设你对目前聘用的秘书不满意，需要重新聘请一位秘书，并一直试图寻找更合适的人选。对当前的求职者进行面试之后，你做出决定：如果求职者的能力比现任秘书强，则解雇现任秘书，并聘用该求职者。当然前提是面试的成本很低，而聘用的成本是昂贵的。你愿意支付这种策略所产生的费用，但还是想预估一下全部费用。

为了解决这个问题，这种策略可以形式化为算法 HireAssistant($n$)，其伪代码具体如下。

```
HireAssistant(n)
1    best ← 0
2    for i ← 1 to n do
3        interview candidate i
4        if candidate i is better than candidate best then
```

```
5            best ← i
6            hire candidate i
```

其中，现任秘书从 0 开始，即原来没有秘书；best 表示当前最好的秘书。

令 $C_i$ 表示面试成本，$C_h$ 表示聘用成本，$m$ 表示被聘用的人数，则 HireAssistant($n$)的总费用为 $O(nC_i+mC_h)$。当第一个求职者就是最好的秘书时，算法的最好情形时间复杂度为 $\Omega(nC_i+C_h)$；当应聘的求职者一个比一个好，即后面来的求职者比前面的求职者好时，算法的最坏情形时间复杂度为 $O(nC_i+nC_h)$。

对于聘用问题，它的平均费用有多大？为了进行概率分析，假设求职者以随机顺序来应聘，并且两个求职者能够比较出哪一个更合格。对于当前求职者而言，如果能够获得其被聘用的概率，则可以估出整个聘用的费用。事实上，对于 $i$ 个求职者而言，每一个人都有同样的概率被聘用，因此，第 $i$ 个求职者被聘用的概率为 $\frac{1}{i}$。平均聘用的费用为

$$\sum_{i=1}^{n}\frac{C_h 1}{i}=C_h \ln n$$

对于聘任问题，算法平均情形复杂度为 $\Theta(nC_i+C_h\ln n)$。

从上面几个例子的分析可以看出，通过对输入的分布进行假设，便可以利用概率分析算法来分析平均情形时间复杂度，而且平均情形时间复杂度确实接近最坏情形时间复杂度。本书第 5 章还将介绍概率分析的应用，这里就不再赘述。

## 3.2 分摊分析

前面介绍了如何分析一种算法的平均情形时间复杂度，但在一些实际应用中，一种算法平均时间复杂度的分析比较复杂；或者按照前面的最坏情形分析方法，可能得到比较高的时间复杂度。此时，可以借助分摊分析(Amortized Analysis)，得出合理的时间复杂度。分摊分析的思想来源于会计学，它为算法分析提供了一种直观的比喻。分摊分析通过研究执行一系列数据结构操作所需要的费用，来研究各个操作之间的关系。一个操作序列可能存在一两个费用比较高的操作，如果割裂了各个操作之间的相关性或忽视问题的具体条件，那么操作序列所需费用的分析结果就比较离谱。分摊分析能够证明：如果一系列操作的总费用是小的，则其中一个操作的分摊费用也是小的，即使一系列操作中某个操作的费用很高。分摊分析与平均情形分析的不同之处在于分摊分析不涉及复杂的概率分析，保证了在最坏情形下每个操作具有的平均性能。

此外，对于一系列操作而言，最坏情形分析是找出其中运行时间最长的操作，然后对每个操作都按照这个最坏情形来估计，从而得到一系列操作的最坏情形时间复杂度。这种估计可能导致比较高的时间复杂度。事实上，最坏情形并不总是频繁发生。比如，读大学的时候，父母每个月会给孩子一定金额的生活费。如果按照花钱最多的一个月去估算，那么父母一年里会给孩子很多钱。事实上，每个月的消费会有多有少，前面几个月如果有剩下的钱，那么在接下来的几个月会继续使用，也就是说，某个月的消费取决于前面几个月的钱是否有剩余。

利用分摊分析的思想，父母可以在第一个月多给一些钱。如果第一个月的实际花费少于父母给的生活费，则剩余的钱可以在第二个月继续使用。这也表明，一系列数据结构的操作之间是相互关联的，从而才能利用分摊分析的思想。

下面介绍分摊分析的3种方法。

### 3.2.1　合计方法

合计方法(Aggregate Method)分析由 $n$ 个操作构成的序列在最坏情形下的运行时间总和 $T(n)$，因而在最坏情形下，每个操作的平均费用，或者分摊费用可定义为 $\frac{T(n)}{n}$。请注意：分摊费用的计算方法对每个操作都是适用的，即使当序列中存在几种类型的操作时也一样成立。

考虑进栈和出栈操作的例子，进栈操作(Push($S$,$x$))和出栈操作(Pop($S$))分别如下。

Push($S$,$x$)：将元素 $x$ 压入栈 $S$。

Pop($S$)：弹出并返回 $S$ 的顶端元素。

因为这两个操作的运行时间复杂度都为 $O(1)$，所以可以把每个操作的费用看作1。这样的话，一个包含 $n$ 个进栈和出栈操作的序列的总费用为 $n$，因而这 $n$ 个操作的实际运行时间复杂度为 $O(n)$。

现在增加一个栈操作 MultiPop($S$,$k$)，其功能是弹出栈 $S$ 顶部的 $k$ 个元素；但如果栈 $S$ 中元素的个数小于 $k$，则此操作将把栈清空。其伪代码如下。

```
MultiPop(S,k)
1    while StackEmpty(S)≠∅ and k≠0 do
2        Pop(S)
3        k←k-1
```

其中，行1表示当 StackEmpty($S$)不为空(∅)且 $k\neq0$ 时，元素出栈。现在分析一下元素个数为 $s$ 的栈 $S$ 执行 MultiPop($S$,$k$)所需要的运行时间。如果 $s$ 小于 $k$，则此操作将把栈清空，否则弹出栈 $S$ 顶部的 $k$ 个元素，因此其费用取决于 $s$ 和 $k$，即 MultiPop($S$,$k$)的费用为 $\min\{s,k\}$。

下面给出一个例子 MultiPop($S$,$k$)，如图3.1所示。

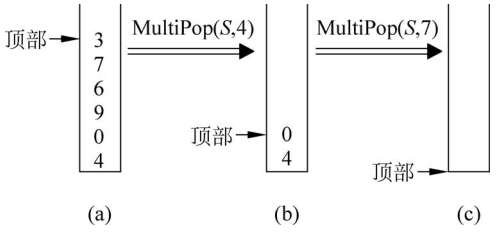


图3.1　MultiPop($S$,$k$)

图3.1(a)所示的栈中有6个元素，执行 MultiPop($S$,4)操作后，得到图3.1(b)。此时栈中有2个元素，执行 MultiPop($S$,7)操作后，得到空栈，如图3.1(c)所示。

例如有一个初始为空的栈，执行一个由 $n$ 个 Push($S$,$x$)、Pop($S$)和 MultiPop($S$,$k$)构成的序列，现对其所花费的时间进行分析。

在 $n$ 个操作组成的序列中，MultiPop($S$,$k$)可能有 $O(n)$个，而堆栈的大小至多为 $n$。由于每个操作的费用为 $O(n)$，包含 $n$ 个操作的序列最坏情形的总费用为 $O(n^2)$，则每个操作的平均费用为 $\frac{O(n^2)}{n}=O(n)$。虽然这个分析是正确的，但是上界太大。事实上，用分摊分析可以得到更好的上界。虽然某一次的 MultiPop($S$,$k$)的费用可能较高，但是作用于本例的费用至多为 $O(n)$，这是因为一个元素在每次被压入栈后至多被弹出一次，即调用 Pop($S$)（包括 MultiPop($S$,$k$)中的 Pop($S$)）的总次数至多等于调用 Push($S$,$x$)的次数 $n$。因此，本例序列的最坏运行时间为 $O(n)$，每个操作的分摊费用为 $\frac{O(n)}{n}=O(1)$。

考虑实现一个由 0 开始向上计数的 $k$ 位二元计数器。令大小为 $k$ 的数组 $A[0..(k-1)]$ 表示一个 $k$ 位的计数器，存储二进制数 $x$，其低位在 $A[0]$，高位在 $A[k-1]$，因此有

$$x=\sum_{i=0}^{k-1}A[i]2^i$$

对于初始状态为 $x=0$，$A[i]=0(i=0,1,\cdots,k-1)$。为了实现计数器的加法运算，定义加操作 Increment($A$)，其伪代码具体如下。

```
Increment(A)
1    i ← 0
2    while i < k and A[i] = 1 do
3        A[i] ← 0
4        i ← i + 1
5    if i < k then
6        A[i] ← 1
```

Increment($A$)总是从低位开始执行加操作。行 2 中当 $i<k$ 且 $A[i]=1$，$A[i]\leftarrow 0$，然后前进一位。行 5 中如果 $i<k$，表明 $A[i]=0$，则 $A[i]\leftarrow 1$。

对于 Increment($A$)而言，其时间复杂度主要取决于位改变的次数。如果数组 $A$ 的元素均为 1，则最坏情形时间复杂度为 $O(k)$，因此，对于初始为 0 的计数器而言，一个具有 $n$ 个加操作的序列最坏情形时间复杂度为 $O(nk)$，平均每个加操作的分摊费用为 $\frac{O(nk)}{n}=O(k)$。这个上界太大，可以进一步进行缩小。如果分析得更精确些，则可得到执行 $n$ 次 Increment($A$)的序列的最坏情形时间复杂度为 $O(n)$，事实上每次调用 Increment($A$)时，并不是所有的位都发生了变化。

一个 8 位二进制计算器从 0 开始执行 Increment($A$)，连续执行 16 次操作的结果如表 3.1 所示，其中最后一列为位发生变化的次数。从表 3.1 可以看出，位 $A[0]$每次执行 Increment($A$)翻转一次；位 $A[1]$每两次执行 Increment($A$)翻转一次，即 $A[1]$翻转 $\left\lfloor\frac{n}{2}\right\rfloor$ 次；位 $A[2]$每四次执行 Increment($A$)翻转一次，即 $A[2]$翻转 $\left\lfloor\frac{n}{2^2}\right\rfloor$ 次。一般地，对于初始为 0 的计数器而言，执行 $n$ 次 Increment($A$)，其位 $A[i]$翻转 $\left\lfloor\frac{n}{2^i}\right\rfloor$ 次，其中，$i=0,1,\cdots,\lfloor\lg n\rfloor$。

对于 $i>\lfloor \lg n \rfloor$，位 $A[i]$不翻转，因而执行 $n$ 次 Increment($A$)，位翻转的总次数为

$$\sum_{i=0}^{\lfloor \lg n \rfloor}\left\lfloor \frac{n}{2^i} \right\rfloor < n\sum_{i=0}^{\infty}\frac{1}{2^i} \approx 2n$$

因此，对一个初始为 0 的计数器，执行 $n$ 个 Increment($A$)的最坏情形时间复杂度为 $O(n)$，每个操作的分摊费用为$\frac{O(n)}{n}=O(1)$。

**表 3.1　8 位二进制计数器连续执行 16 次 Increment($A$)的结果**

| 计数器值 | $A[7]$ | $A[6]$ | $A[5]$ | $A[4]$ | $A[3]$ | $A[2]$ | $A[1]$ | $A[0]$ | 位翻转次数/次 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 3 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 4 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 7 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 8 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 10 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 11 |
| 8 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 15 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 16 |
| 10 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 18 |
| 11 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 19 |
| 12 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 22 |
| 13 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 23 |
| 14 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 25 |
| 15 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 26 |
| 16 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 31 |

### 3.2.2　记账方法

记账方法(Accounting Method)的思想是对不同的操作赋予不同的费用。某些操作赋予的费用可能比其实际费用(实际所需的费用)多，也有可能少。对某一操作赋予的费用，就记为该操作的分摊费用。当一个操作的分摊费用比其实际费用多时，多出来的费用(即余款)作为存款保存在数据结构的一些特定对象中。这笔存款用来支付给分摊费用比实际费用少的操作。这样，一个操作的分摊费用可以分成两部分：一部分用来支付实际费用；另一部分作为存款或者透支(分摊费用不够支付该操作的实际费用)保存在数据结构的某个特定对象中，以备后用。

在选择每个操作的分摊费用时，需要非常小心。如果利用分摊费用来证明在最坏情形下每个操作的平均费用是小的，那么操作序列的总分摊费用必须为操作序列总实际费用的上界，从而保证与该数据结构的总存款始终为非负值。令操作序列中操作 $i$ 的实际费用为 $c_i$，分摊费用为 $\hat{c}_i$，则需满足

$$\sum_{i=1}^{n} c_i \leqslant \sum_{i=1}^{n} \hat{c}_i$$

从而使数据结构的总存款 $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$ 始终为非负值。

为了描述记账方法的应用，再次考虑栈操作的例子。各个栈操作的实际费用如表 3.2 所示。

**表 3.2 各个栈操作的实际费用**

| 栈 操 作 | 实 际 费 用 |
|---|---|
| Push($S$,$x$) | 1 |
| Pop($S$) | 1 |
| MultiPop($S$,$k$) | min{$s$,$k$} |

各个栈操作的分摊费用如表 3.3 所示。

**表 3.3 各个栈操作的分摊费用**

| 栈 操 作 | 分 摊 费 用 |
|---|---|
| Push($S$,$x$) | 2 |
| Pop($S$) | 0 |
| MultiPop($S$,$k$) | 0 |

假设用 1 元表示单位费用，开始时栈 $S$ 为空，一个元素 $x$ 压进栈的 Push($S$,$x$)的分摊费用为 2 元，其中，1 元支付实际费用，1 元作为存款保存在该元素中，因此，在任何时候，栈 $S$ 的元素均有 1 元存款。当需要一个元素出栈时，出栈的实际费用就用该元素的存款来支付。这样，通过赋予 Push($S$,$x$)多一点的分摊费用，就不必赋予 Pop($S$)分摊费用。同样地，对于 MultiPop($S$,$k$)也一样，不管该操作弹出几个元素，其实际费用，均可以用弹出元素的存款来支付。由于栈 $S$ 的元素均有 1 元存款，这就可以保证栈中存款总数为非负值，因而，对于任意包含 $n$ 个 Push($S$,$x$)、Pop($S$)和 MultiPop($S$,$k$)的序列，实际费用的和不会超出分摊费用的和，也就说总分摊费用就是总实际费用的上界。又因为总分摊费用为 $O(n)$，故总实际费用也不会超过 $O(n)$。

下面给出另一个例子。

前面已经分析了，二元计数器加操作 Increment($A$)的运行时间取决于位翻转的次数，而位翻转的次数的分析比较复杂。本例用 1 元表示单位费用。为进行分摊分析，规定为某一位设置为 1 的操作支付 2 元的分摊费用，其中，1 元用于支付将该位设置为 1 的实际费用，1 元作为存款存在该位上。若将该位复位为 0，则其实际费用用该位的存款来支付。

在 Increment($A$)伪代码的 **while** 循环中，复位的费用由该位的存款支付，行 6 至多有一位被置为 1，因此，Increment($A$)的分摊费用至多为 2 元。二元计数器中“1”的数目总是非负数，因此计数器中存款的数目总是非负数。因此，对于包含 $n$ 个 Increment($A$)的操作序列，其总分摊费用为 $O(n)$。而总分摊费用是实际费用的上界，故总实际费用也为 $O(n)$。

### 3.2.3　势能方法

前面已经介绍了记账方法把余款作为存款存储在数据结构的某个特定对象中，而势能方法(Potential Method)把每个操作的余款表示成一种“势能”，存储在整个数据结构中。这种势能在需要时可以用来支付后面操作所需要的费用。

视频讲解

势能方法的思想是从一个初始的数据结构 $D_0$ 出发，执行 $n$ 个操作序列。设 $c_i$ 为操作 $i$ 的实际费用，$D_i$ 为对数据结构 $D_{i-1}$ 执行操作 $i$ 的结果，其中，$i=1,2,\cdots,n$。势能函数 $\Phi$ 将每个数据结构 $D_i$ 映射为实数 $\Phi(D_i)$，即与数据结构 $D_i$ 相关的势能。

定义操作 $i$ 的分摊费用为

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

则 $n$ 个操作的总分摊费用为

$$\sum_{i=1}^{n}\hat{c}_i = \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n}c_i + \Phi(D_n) - \Phi(D_0)$$

如果势能函数 $\Phi$ 能使 $\Phi(D_n)\geqslant\Phi(D_0)$，则总分摊费用就是总实际费用的上界。如果能保证对所有操作，有 $\Phi(D_i)\geqslant\Phi(D_0)$，则如同记账方法一样，可以预先支付费用。通常定义 $\Phi(D_0)=0$，然后证明对所有操作，有 $\Phi(D_i)\geqslant 0$。直观地说，如果操作 $i$ 的势能差 $\Phi(D_i)-\Phi(D_{i-1})>0$，意味着付给操作 $i$ 的分摊费用除了支付掉实际费用外，还有剩余，则数据结构的总势能将增加。如果势能差 $\Phi(D_i)-\Phi(D_{i-1})\leqslant 0$，则表示付给操作 $i$ 的分摊费用过少，需要从总势能中提取一笔费用来支付实际费用，因而数据结构中的总势能将减少。值得注意的是，上述定义的分摊费用依赖于所选择的势能函数 $\Phi$。不同的势能函数可能会产生不同的分摊费用，但这些分摊费用都是实际费用的上界。因此，在选择势能函数时，常要做一些权衡，最佳势能函数的选择取决于想要得到的上界。下面给出势能方法的应用。

对于前面介绍的栈操作，定义栈上的势能函数 $\Phi$ 为栈中元素的个数。开始时要处理的是空栈 $D_0$，因此 $\Phi(D_0)=0$。因为栈中的元素个数始终是非负的，所以在操作 $i$ 之后的栈 $D_i$ 就具有非负的势能，即 $\Phi(D_i)\geqslant\Phi(D_0)=0$，又因为 $\sum_{i=1}^{h}\hat{c}_i = \sum_{i=1}^{h}c_i + \Phi(D_n) - \Phi(D_0) \geqslant \sum_{i=1}^{h}c_i$，因此势能函数为 $n$ 个操作的总实际费用的上界为分摊费用的总和。

现在计算各个栈操作的分摊费用。如果作用于一个包含 $s$ 个元素的栈上的操作 $i$ 是 Push$(S,x)$操作，则其势能差为

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$$

分摊费用为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1})\\ &= 1 + 1\\ &= 2\end{aligned}$$

假设操作 $i$ 是 MultiPop$(S,k)$，且弹出了 $k'=\min(S,k)$个元素，该操作的实际费用为 $k'$，其势能差为

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'$$

因此，MultiPop(S，$k$)的分摊费用为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' \\ &= 0\end{aligned}$$

类似地，Pop(S)的分摊费用为0。上述计算表示每个操作的分摊费用均为$O(1)$。对于任意包含$n$个Push(S，$x$)、Pop(S)和MultiPop(S，$k$)的操作序列，其总实际费用的上界为分摊费用的总和，因此该序列的最坏情形的费用为$O(n)$。

对于前面介绍的二元计数器加操作Increment($A$)的例子，定义在第$i$次操作后计数器的势能为$b_i$，其中，$b_i$为第$i$次操作后二元计数器中“1”的个数。

现在计算Increment($A$)的分摊费用。假设第$i$次操作对$t_i$位进行了复位，即置0，那么该操作的实际费用至多为$t_i+1$，这是因为除了将$t_i$位复位外，Increment($A$)至多将一位设置为1。若$b_i=0$，则第$i$个操作将所有$k$位复位，因此有$b_{i-1}=t_i=k$。若$b_i>0$，则有$b_i=b_{i-1}-t_i+1$。对上述任一情形，均有$b_i \leqslant b_{i-1}-t_i+1$。

第$i$次操作后，Increment($A$)的势能差为

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= b_i - b_{i-1} \leqslant (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i\end{aligned}$$

分摊费用为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leqslant (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

二元计数器从0开始，因而$\Phi(D_0)=0$。因为二元计数器中“1”的个数始终是非负的，所以第$i$个操作之后，二元计数器的势能非负，即$\Phi(D_i) \geqslant \Phi(D_0)=0$。这意味着$n$个操作的总实际费用的上界为分摊费用的总和，故该序列的最坏情形复杂度为$O(n)$。

势能方法提供了一种方便分析二元计数器加操作序列的时间复杂度的方法。即使二元计数器不从0开始，也可以通过这种方法类似地分析，这里就不再详述，具体练习见习题3-10。

前面介绍的3种分摊分析方法是分析算法的不同的、新的工具。分摊分析在许多算法的分析及优化设计中都得到了直接地应用，例如，图算法时间复杂度的分析（见第8章）。近似算法的应用，有兴趣的读者可参考文献[1]。

## 3.3 实验分析

前面介绍的几种分析都是基于渐近符号来分析算法的渐近复杂性，然而渐近复杂性也有不合理之处。渐近复杂性考虑的是随着问题规模的无限增大时，算法的效率表现，而实际问题的规模总有一个范围，因而渐近复杂度低的算法在求解规模较小的问题时，并不一定优于一个渐近复杂度高的算法。还有一些问题非常复杂，无法对输入做某种假设，因而很难对算法进行理论分析。此外，一些非常难的问题很难找到最优解，其算法的分析不仅要考虑计算速度，而且要考虑解的质量，即算法的性能（Performance），这时就需要借助实验分析。

实验分析（Empirical Analysis）的目的是借助计算机对算法进行大量的实验测试，通过

分析实验结果，来比较不同算法的性能，具有简单、实用的优点。其基本过程是：首先选择一些具体的问题实例，然后利用算法对这些问题实例进行测试，最后记录解的质量及计算所需要的时间。通过对计算结果进行统计分析，来分析比较算法的性能。通过分析算法的实际运行结果，可以知道在现有的计算资源下算法可以求解多大的问题，解的质量如何，所需要的计算时间等。实验分析可以在完全真实的条件下进行，不需要像概率分析那样做各种假设，因此，实验分析倍受工程应用领域的青睐，成为算法分析的有力工具之一。

实验分析的缺点是容易受所选问题实例、计算模型、算法参数等因素的影响。例如，对于一些随机算法，即使是同一问题实例，如果多次运行算法，其运行结果可能也不一样。实验分析的主要步骤有数据的选择与生成、实现算法、计算结果分析，具体如下。

(1) 数据的选择与生成

由于实验分析容易受所选问题实例的影响，因此实验数据的选择必须保证具有足够多。这些数据既要包括规模小的问题实例，又要包括规模大的问题实例；既有容易的问题实例，又有困难的问题实例。只有这样，选择的数据才会具有代表性，相应的实验分析才有说服力，也才能得到统计意义上的重要结果。

对于一些困难的问题，网上都有一些公开的测试实例(Benchmark)，例如，文献[19]给出了一些经典 NP 问题的测试实例，因此，通常测试这些公开的实例，就可以比较算法的性能，满足研究的需要。如果没有公开的测试实例，也可以随机生成一些测试数据，然后通过测试这些数据，进行算法比较分析，来验证算法的可行性和有效性。

(2) 实现算法

由于算法实现与具体的程序设计技巧有关，当用实验分析比较不同算法的性能时，必须假设实现不同算法的程序设计技巧相同，代码优化方法也相似。算法的实现还要注意尽量保证使用相同的计算机(计算模型)，如果使用的计算机不同，则需要考虑 CPU 频率、内存等情况。只有这样，不同的算法才能进行公平比较，所得到的计算结果也才更有说服力。

(3) 计算结果分析

计算结果通常借助图表进行可视化，这样可以非常直观地比较不同算法的性能，发现算法的未知特性。此外，计算结果分析也可以帮助分析算法的时间复杂度。例如，比值测试，分析方法是先估计算法的时间复杂度函数 $t(n)$，然后对问题规模 $n$ 从小到大的各种实例进行测试，计算算法所需要的时间与 $t(n)$的比值。如果该比值趋向一个常数，那么可以认为 $t(n)$与实际运行时间相吻合，是一个好的估计。这种估计建立在实验的基础上，因而比纯粹的理论分析给人一种更为真实的感觉。

此外，实验分析也有助于研究算法的最好情形、平均情形和最坏情形的性能，其方法是选择各种问题实例，例如，容易的问题实例、特别困难的问题实例及随机的问题实例。在这些实例上测试算法的性能，可以估计出算法最好情形、平均情形和最坏情形的性能，以及分析算法对问题实例的敏感性。对于随机算法，针对 Benchmark 数据或者随机生成的测试例子，进行若干次计算，然后统计出最好的性能，平均性能，以及最坏的性能，以分析不同算法的表现。

综上所述，实验分析不仅可以按理论分析的一些模式来分析算法，而且可以发现理论分析无法发现的结果和现象。特别是有些算法，如模拟退火算法、遗传算法，常常依赖一些参数的设计，利用实验分析不仅可以测试算法性能，还可以确定较好的参数设置，从而设计出

更有效的算法。

## 3.4 小结

本章的内容主要取材于《算法导论》[2]，其他内容参考了文献[5,6]。主要介绍如何利用概率分析来研究算法的平均时间复杂度，然后介绍了目前算法分析里比较新的分析工具，即分摊分析。关于分摊分析的一些应用，感兴趣的读者可以阅读《算法导论》[2]。最后，本章介绍了目前流行的实验分析方法。由于目前元启发式(Meta-heuristics)算法，如遗传算法、模拟退火、蚁群算法，以及一些机器学习方法，很难从理论上分析算法的时间复杂度，因此经常采用实验分析的方法。本书后面介绍算法的时候，也会涉及实验分析的应用。

## 习题

3-1 在 HireAssistant($n$)中，假设应聘者以随机的顺序出现，正好被雇用一次的概率是多少？正好被雇用两次的概率是多少？正好被雇用 $n$ 次的概率又是多少？

3-2 请分析习题 1-5，找出 $n$ 个数中最大值的算法平均时间复杂度。

3-3 如果一组栈操作中包括一次 MultiPush($S$,$k$)，一次性地把 $k$ 个元素压入栈内，那么栈操作分摊费用的界 $O(1)$是否还能保持？

3-4 对某个数据结构执行一个具有 $n$ 个操作的序列，如果 $i$ 为 2 的整数幂，则第 $i$ 个操作的费用为 $i$，否则为 1。请利用合计方法确定每次操作的分摊费用。

3-5 对一个元素数量从不超过 $k$ 的栈执行一系列栈操作，并在每 $k$ 个操作后，复制整个栈的内容以作备份。证明对各种栈操作分配合适的分摊费用后，$n$ 个栈操作(包括复制栈的操作)的费用为 $O(n)$。

3-6 利用记账方法分析习题 3-4。

3-7 假设存在势函数 $\Phi$，对所有 $i$ 有 $\Phi(D_i)\geqslant\Phi(D_0)$，但是 $\Phi(D_0)\neq 0$。证明存在一个势函数 $\Phi'$，对所有 $i\geqslant 1$，$\Phi'(D_0)=0$，$\Phi'(D_i)\geqslant 0$，且用 $\Phi'$ 表示的分摊费用与用 $\Phi$ 表示的分摊费用相同。

3-8 用势能方法分析习题 3-4。

3-9 假设某个栈在执行 $n$ 个栈操作 Push($S$,$x$)、Pop($S$)和 MultiPop($S$,$k$)之前包含 $S_0$ 个对象，结束后包含 $S_n$ 个对象，试求 $n$ 个栈操作的总费用。

3-10 对于一个二元计数器执行加操作序列，如果计数器不从 0 开始，用势能方法分析该序列的时间复杂度。