论随机化算法的原理与设计

上海市控江中学 周咏基

[关键字]

随机化算法，稳定性

[摘要]

本文提出了一种新的解决信息学问题的算法——随机化算法，并讨论了其原理与设计方法。论文首先给出随机化算法的定义，说明了由于“运气”的影响，必须对随机化算法的稳定性进行分析。然后分“随机不影响算法的执行结果”，“随机影响执行结果的正确性”，“随机影响执行结果的优劣”三种情况，以从基本算法到竞赛试题中用随机化算法有效解决问题的例子，详细分析了三种情况的随机化算法的原理与设计方法。最后总结出随机化算法的基本原理和共同性质，提出设计随机化算法的一般方法，并指出随机化算法的适用范围和一个有效的随机化算法应具备的特点。

[正文]

1．引论

在这篇论文中，我们将研究一种新概念的算法——随机化算法。顾名思义，随机化就是指使用了随机函数。这里的随机函数不妨是 Borland Pascal(或 Turbo Pascal)中的 RANDOM(N)，其返回值为[0,N-1]中的某个整数，且返回每个整数

都是等概率的[1]。

一个含有随机函数的算法很可能[2]受到不确定因素的支配。人们通常认为，一个受到不确定因素支配的算法肯定不是一个有效的算法——正是在这种思维方式的支配下，随机化算法一直被冷落——但是，在接下来的讨论中，我们将看到完全相反的事情发生：对于一些特定的问题，随机化算法恰恰成了十分有效的解题工具，有时甚至比一般的非随机化算法做得更好。

随机化算法的定义

随机化算法是这样一种算法，在算法中使用了随机函数，且随机函数的返回值直接或间接地影响了算法的执行流程或执行结果。

根据这个定义，并不是所有的用了随机函数的算法都可称为随机化算法。例如，某个算法包含

i←RANDOM(N)，

但变量 i 除了在这里被赋予一个随机值之外，在其它地方从未出现过。显然，如果这个算法没有在其它地方用过随机函数，上面这条语句就无法影响执行的流程或结果，这个算法就不能称为随机化算法。

另一方面，若一个算法是随机化算法，则它执行的流程或结果就会受其中使用的随机函数的影响。我们按影响的性质和程度分三种情况：

1．随机不影响执行结果。这时，随机必然影响了执行的流程，其效应多表现为算法的时间效率的波动。

2．随机影响执行结果的正确性。在这种情况中，原问题要求我们求出某个可行解，或者原问题为判定性问题[3]，随机的效应表现为执行得到正确解的概率。

3．随机影响执行结果的优劣。这时，随机的效应表现为实际执行结果与理论上的最优解或期望结果的差异。

第 2，3 种情况中，随机的影响还可能伴随有对执行流程的影响。

我们后面的讨论就分这三种情况进行。在讨论之前，我们还要澄清一个问题。

随机化和“运气”

由于随机化算法的执行情况受到不确定因素的支配，因此即使同一个算法在多次执行中用同样的输入，其执行情况也会不同，至少略有差异。差异表现为出解速度快慢，解正确与否，解的优劣等等。例如：一个随机化算法可能在两次执行中，前一次得到的解较优，后一次的较劣。现在的问题是：在大多情况中，尤其是竞赛时，对于同样的输入，只允许程序运行一次，根据运行结果判定算法的好坏。如此一来，我们就会把出劣解的一次运行归咎于运气不佳，反之亦然然而，比赛比的是谁的算法更有效，而不是谁的运气更好。

既然我们使用了随机函数，我们就无法摆脱运气的影响，所以我们的目标是尽量将运气的影响降到最低。也就是说，我们必须使算法的执行情况较为稳定因此，在接下来的对算法的分析中，我们将从以下四方面分析算法的性能。

1．时间效率；

2．解的正确性；

3．解的优劣程度(解与最优解的接近程度)；

4．稳定性，即算法对同样的输入的执行情况的变化。变化越小则越稳定。

非随机化算法的稳定性为 100%，随机化算法的稳定性属于区间(0%,100%)。

通常，只要算法的程序实现所用的空间不超过内存限制，我们就不必刻意提高算法的空间效率，所以我们省去了空间效率这项分析。上面第 4 项的“稳定性”可以是算法的平均时间复杂度，也可以是执行算法得到正确解的概率，还可以是实际解达到某一优劣程度的概率。“稳定性”这一项是评判随机化算法好坏的一个重要指标。

2．执行结果确定的随机化算法

在这一节中，我们以快速排序和它的随机化版本为例，讨论执行结果确定的随机化算法。根据引言中的分析，一个随机化算法的执行结果确定，则它的执行流程必会受随机的影响，影响多表现在算法的时间效率上。所以在下面的讨论中，我们省去了对算法执行结果正确性和优劣的分析。

快速排序算法

快速排序是一种我们常用的排序方法，它的基本思想是递归式的：将待排序的一组数划分为两部分，前一部分的每个数不大于后一部分的每个数，然后继续分别对这两部分作划分，直到待划分的那部分数只含一个数为止。算法可由以下伪代码描述。

```
QUICKSORT(A,lo,hi)
1  if lo < hi
2    p←PARTITION(A,lo,hi)
3    QUICKSORT(A,lo,p)
4    QUICKSORT(A,p+1,hi)
```

如果待排序的 n 个数存入了数组 A，则调用 QUICKSORT(A,1,n)就可获得升序排列的 n 个数。以上的快速排序的算法依赖于 PARTITION(A,lo,hi)划分过程。该过程在$\Theta(n)$的时间内，把 A[lo..hi]划分成不大于 x=A[lo]，和不小于 x=A[lo]的两部分。这两部分分别存入 A[lo..p]和 A[p+1,hi]。而在 QUICKSORT(A,lo,hi)过程中递归调用 QUICKSORT()，对 A[lo..p]和 A[p+1..hi]继续划分。

可以证明[4]，快速排序在最坏情况下(如每次划分都使 p=lo)的时间复杂度为$\Theta(n^2)$，在最坏情况下的时间复杂度为$\Theta(n\log_2 n)$。如果假设输入中出现各种排列都是等概率的(但实际情况往往不是这样)，则算法的平均时间复杂度为$O(n\log_2 n)$。

随机化的快速排序

经分析我们看到，快速排序是十分有效的排序法，其平均时间复杂度为$O(n\log_2 n)$。但是在最坏情况下，它的时间复杂度为$\Theta(n^2)$，当 n 较大时，速度就很慢(见本节后部的算法性能对照表)。其实，如果照前面的假设，输入中出现各种排列都是等概率的，那么出现最坏情况的概率小到只有$\Theta(1/n!)$，且在$\Theta()$中隐含的常数是很小的。这样看来，快速排序还是相当有价值的。

但是实际情况往往不符合该假设，可能对某个问题来说，我们遇到的输入大部分都是最坏情况或次坏情况。一种解决的办法是不用 x=A[lo]划分 A[lo..hi]，

而用 x=A[hi]或 x=A[(lo+hi) div 2]或其它的 A[lo..hi]中的数来划分 A[lo..hi]，这要看具体情况而定。但这并没有解决问题，因为我们可能遇到的这样的输入：有三类，每一类出现的概率为 1/3，且每一类分别对于 x=A[lo]，x=A[hi]，x=A[(lo+hi) div 2]为它们的最坏情况，这时快速排序就会十分低效。

我们将快速排序随机化后可克服这类问题。随机化快速排序的思想是：每次划分时从 A[lo..hi]中随机地选一个数作为 x 对 A[lo..hi]划分。只需对原算法稍作修改就行了。我们只是增加了 PARTITION_R 函数，它调用原来的 PARTITION()过程。QUICKSORT_R()中斜体部分为我们对 QUICKSORT 的修改。

```
PARTITION_R(A,lo,hi)

1  r←RANDOM(hi-lo+1)+lo

2  交换 A[lo]和 A[r]
3  return PARTITION(A,lo,hi)


QUICKSORT_R(A,lo,hi)
1  if lo < hi

2    p←PARTITION_R(A,lo,hi)

3    QUICKSORT_R(A,lo,p)
4    QUICKSORT_R(A,p+1,hi)
```

分析随机化快速排序算法

随机化没有改动原来快速排序的划分过程，故随机化快速排序的时间效率依然依赖于每次划分选取的数在排好序的数组中的位置，其最坏，平均，最佳时间复杂度依然分别为$\Theta(n^2)$，$O(n\log_2 n)$，$\Theta(n\log_2 n)$，只不过最坏情况，最佳情况变了。最坏，最佳情况不再由输入所决定，而是由随机函数所决定。也就是说，我们无法通过给出一个最坏的输入来使执行时出现最坏情况(除非我们运气不佳)。

正如引论中所提到的，我们现在来分析随机化快速排序的稳定性。按各种排列的出现等概率的假设(该假设不一定成立)，快速排序遇到最坏情况的可能性为$\Theta(1/n!)$。假设 RANDOM(n)产生 n 个数的概率都相同(该假设几乎一定成立)，则随机化快速排序遇到最坏情况的可能性也为$\Theta(1/n!)$。如果 n 足够大，我们就有多于 99%的可能性会“交好运”。也就是说，随机化的快速排序算法有很高的稳定性。

下面是原来的快速排序和随机化后的快速排序的性能对照表。

分析项目		原算法	随机化后的算法
理论时间效率	最坏情况	$\Theta(n^2)$	$\Theta(n^2)$
	最佳情况	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$
	平均情况	$O(n\log_2 n)$	$O(n\log_2 n)$
	稳定性	$\Theta(1)$	$\Theta(1-1/n!)$
实际	随机输入(n=30000)	0.22s	0.27s

运行情况	最坏输入(n=30000)	66s	0.22s
	稳定性(n 足够大)	100%	>99%
结论	最坏情况的起因	最坏输入	随机函数返回值不佳
	时间效率对输入的依赖	完全依赖	完全不依赖

对以上表格有几点说明：

1．程序运行环境为 Pentium 100MHz，BP7.0 编译。

2．随机化算法的相应程序的运行时间均为 1000 次运行的平均值。

3．测试随机化算法的稳定性时，相应程序对不同输入各运行了 1000 次。

4．程序代码见 QSORT.PAS。

小结

从以上分析看出，执行结果确定的随机化算法原理是：用随机函数全部或部分地抵消最坏输入的作用，使算法的时间效率不完全依赖于输入的好坏。

通过对输入的适当控制，使得执行结果相对稳定，这是设计这一类随机化算法的常用方法。例如，在随机化快速排序算法中，我们每次随机地选取 x 来划分 A[lo..hi]。这一方法的效应等价于在排序前先随机地将 A 中的数打乱。又如在建立查找二叉树时，可先随机地将待插入的关键字的顺序打乱，然后依次插入树中，以获得较平衡的查找二叉树，提高以后查找关键字的效率。

3．执行结果可能偏离正确解的随机化算法

在这一节中我们讨论第 2 种情况的随机化算法。这种随机化算法甚至会输出错误的结果，但它依然是很有效的。我们以判定素数的算法为例。

朴素的素数判定算法

对于较小的 n，我们可以用“筛数法”判定 n 是否为素数。对于稍大一点的 n，我们可以先求出[2,⌊sqrt(n)⌋]内的所有素数，再用这些素数试除 n。这两种方法都要借助于大数组，如果 n 足够大，就不再适用了。这时，我们只能用 2,3,...,⌊sqrt(n)⌋试除 n，一旦除尽，n 必然是合数，否则为素数。算法描述如下：

```
ISPRIME_NAIVE(n)
1  for a←2 to⌊sqrt(n)⌋
2    if a|n
3      return FALSE
4  return TRUE
```

实现时，我们可以先判断 n 是否为偶数，然后用 3,5,7,9，...试除 n，以加快程序运行速度。经管如此，当遇到较大的素数 n 时，这一算法还是会显得十分慢的。其最坏情况时间复杂度为$\Theta(n^{1/2})$。

随机化的素数判定算法

换一个角度，由 Fermat 定理我们知道：若 n 是素数，a 不能整除 n，则

$$a^{n-1}\equiv 1 \pmod n$$

必然成立。我们将它改成：若 n 是素数，对于 a=1,2,...,n-1，有 $a^{n-1}\equiv 1 \pmod n$。所以，若存在整数 $a\in[1,n-1]$，使得 $a^{n-1}\neq 1 \pmod n$，则 a 必为合数。我们考虑以下算法：

```
ISPRIME_R(n, s)
1  for i←1 to s
2    a ←RANDOM(n-1)+1
3    if a^(n-1)≠1 (mod n)
4      return FALSE
5  return TRUE
```

该算法随机地选取 s 个 a 值，检查 $a^{n-1}\equiv 1 \pmod n$是否成立。若发现某个 a，使得该式不成立，则算法肯定地判决“n 是合数”；若选取的 a 都使 $a^{n-1}\equiv 1 \pmod n$成立，则算法提出假设“n 是素数”。

这个算法只产生一种错误，即选取的 s 个 a 值均满足 $a^{n-1}\equiv 1 \pmod n$，而 n 是合数时，算法会认为 n 是合数的证据不足，判其为素数。

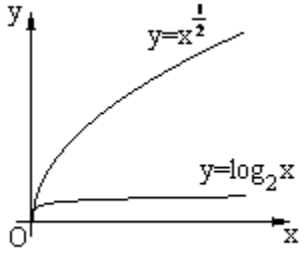
该算法伪代码第 3 行中得求 $a^{n-1} \bmod n$。我们只要$\lfloor \log_2(n-1) \rfloor$次循环就可以求出该值。设 $n-1=b_k2^k+b_{k-1}2^{k-1}+\ldots\ldots+b_02^0$，$b_kb_{k-1}\ldots b_0$ 为 n-1 的二进制表示，则 $k=\lfloor \log_2(n-1) \rfloor$。这样

$$a^{n-1} \bmod n=a^{(\ldots(((bk)\cdot 2+bk-1)\cdot 2+bk-2)\cdot 2+\ldots\ldots b1)\cdot 2+b0} \bmod n$$

记<a>=a mod n，则上式可改写为

$$<<<\ldots<<<<<<a^{bk}>^2><a^{bk-1}>>^2><a^{bk-2}>>^2>\ldots\ldots>^2><a^{b0}>>$$

用 k 次循环即可求出该值。因此随机化的素数判定算法的最坏情况时间复杂度为$\Theta(s\log_2 n)$。如果视 s 为常数，则其时间复杂度为$\Theta(\log_2 n)$。



比较两种算法

随机化素数判定算法的稳定性取决于，对合数 n，在[1,n-1]内满足 $a^{n-1}\equiv 1 \pmod n$的 a 值的个数。记该个数为 H(n)，则算法判定 n 的稳定性为

$(1-H(n)/n)^s$。事实上，大多数 n 的 H(n)/n 都可达到 98%，几乎所有 n 的 H(n)/n 都不小于 50%，在 10000 内，H(n)/n 小于 50%的不超过 10 个。我们有理由相信，只要 s 取适当的值，就能使算法有很高的稳定性。如取 s=50，则算法对几乎所有 n 的稳定性至少为 $1-2^{-50}>99\%$。即使取较小的 s(如 s=5)，算法也未必会得到错误的结果。

随机化素数判定算法的时间效率比朴素的素数判定算法高许多，这可从它们各自的时间复杂度看出。实际运用中，取 s=50，n=761838257287(是素数)，将两种算法对应程序各运行 100 次，前者需 20 秒，后者需 102 秒[5]。其实素数判定通常只是作为一个子程序被调用，其实际使用次数可能还不止 100 次。可见这两种算法的差距是明显的。

小结

执行结果可能偏离正确解的随机化算法基于这样的原理：一个近似正确的算法的近似程度受某个参数的影响。当该参数取某个特点值时，算法能得到正确解。因此，只要将算法执行多次，每次参数取指定范围内的随机值，算法就可能得到正确结果。

通常我们可以这样设计此类随机化算法：先选一个近似算法(这里用了以 Fermat 定理为基础的判定算法)，然后在算法中加入随机化控制(这里用了 a)，最后加外循环控制，多次执行近似算法(这里用了 s)，如此构成随机化算法。可通过重复执行近似算法来控制算法的稳定性，使之达到期望的水平。

4．执行结果有优劣变化的随机化算法

最后我们讨论第 3 种情况的随机化算法。这种随机化算法大多针对要求较优解的问题，例如 NOI’98 中的问题《并行计算》。

若干贪心算法

《并行计算》问题的描述大意为：输入一个由+，-，□，□，(，)，变量(大写字母)组成的四则运算表达式，输出一段指令，控制有两个运算器的并行计算机在尽量短的时间内正确计算表达式的值。程序的得分将取决于其所能找到的最优解与标准答案(未必是最优解)相比较的优劣程度。

如果用搜索算法来解决这个问题，则每次扩展搜索树必须确定当前的操作数、运算符和运算器，搜索量大得惊人。由于问题并未要求我们求最优解，我们可以用贪心算法求较优解。贪心标准有多种选择，如每次先选取耗时长的运算符又如每次选取最早空闲的运算器。我们目前尚未找到一种普遍适用的贪心标准，而且找到这种标准的可能性不大。另一方面，现在的每种标准都只可在一定程度上对某些输入取得较好的结果，我们完全可以对各种标准分别设计出不符合其贪心方式的输入，使它输出不理想的结果。囿于上述限制，用纯粹的贪心算法无法有效地解决《并行计算》问题。

一种解决的方法是：由于贪心算法有很高的时间效率，我们可在同一个程序中将各种贪心标准全都试一次，但这样无疑极大地增加了“编程复杂度”。下面的随机化贪心算法给出了一个较好的解决方式。

随机化的贪心算法

随机化贪心算法的基本思想是：设置贪心程度 rate%(rate∈[0,100])，选一种较好的贪心标准为基础，每次求局部最优解的过程改为每次求在该贪心标准下贪心程度不小于 rate%的某个局部较优解。这一修改可由以下伪代码描述：

```
PARTOPTIMIZE(rate)

1  for A←局部最优解 to 局部最差解

2    if RANDOM(101) <= rate
3      return A
4  return 局部最差解
```

对于目前任一种贪心标准，都存在不符合它的输入，也就是说，存在输入使算法在不是每次都选局部最优解的情况下，得到的解比每次都选局部最优解所得到的解更优，而上述随机化贪心算法能覆盖这种情况。同时上述随机化贪心算法在 rate=100 时得到的结果就是原来未加修改的贪心算法的结果，所以上述随机化的算法至少不比非随机化的差。

上述思想较简单，所以实现起来不困难。而且贪心算法都有很高的时间效率，多次贪心消耗的时间也不会很长。程序代码见 PARALLEL.PAS。在实现中，我们还加了些其它的优化，如对重复项只计算一次。

随机化贪心算法的性能

由于问题对解的约束不多，解的种类和个数就可能很多，因此要从理论上分析随机化贪心算法的性能较为困难。不过我们可以用当时比赛的测试数据对算法进行测试。这样做还是有一定说服力的。下表为各种贪心算法对应程序的运行结果与标准答案的对照表。

输入文件	一般贪心算法	随机化贪心算法	标准答案	随机化贪心算法与标准答案的差距
INPUT.001	14	14	14	O
INPUT.002	50	50	50	O
INPUT.003	55	55	55	O
INPUT.004	22	21	21	O
INPUT.005	53	47(95%) 46(5%)	46	+1
INPUT.006	42	23	22	+1
INPUT.007	130	100	100	O
INPUT.008	39	33	33	O
INPUT.009	3800	3700	3700	O
INPUT.010	230	210	210	O
INPUT.011	10	12	10	+2
INPUT.012	6300	6300	6300	O
INPUT.013	234	234(99%) 235(1%)	234	O
INPUT.014	3597	3474(42%) 3486(30%)	3498	-24

		3462(24%) 3459(4%)		
INPUT.015	412	353(52%) 254(22%) 356(16%) 357(6%) 359(3%) 360(1%)	370	-17
INPUT.016	1250	1150(98%) 1250(2%)	1150	O
INPUT.017	580	559(96%) 580(4%)	559	O
INPUT.018	3108	3108	3108	O
INPUT.019	0	0	0	O
INPUT.020	192	192	171	+21

对以上表格有几点说明:

1．程序运行环境为 Pentium 100MHz，BP7.0。
2．“一般贪心算法”一列中的数为各种贪心标准下的算法所得到的较优解。
3．随机化算法的相应程序运行 100 次后确定其运行结果。上表“随机化贪心算法”这一列中，数旁的括号内有得到该结果的概率。若无括号，则表示 100 次运行均为此结果。
4．随机化贪心算法与标准答案的差距 = 随机化贪心算法得到的较优解 } 标准答案。
5．程序代码见 PARALLEL.PAS。

我们从上表看出，随机化贪心算法对大部分输入都能得到与标准答案同样优的结果，甚至对某几个输入(INPUT.014，INPUT.015)能得到比标准答案更优的结果。同时，该算法也有较高的稳定性(注意，这里的稳定性是指获得某一范围内的解的概率，如得到与标准答案同样优的结果的概率，又如得到比标准答案稍好一点的结果的概率)。另外，其时间效率自然不会差。可见，这里的随机化贪心算法是解决《并行计算》问题的一个有效算法。

小结

执行结果有优劣变化的随机化算法的原理与上节中的基本相同：一个近似算法的近似程度受某个参数的影响，当参数变化时，算法的执行结果会有优劣变化。只要将该算法执行多次，每次参数取指定范围内的随机值，并在执行后及时更新当前最优解，当前解就能不断逼近理论最优解。

设计这样的算法通常以贪心算法为基础，进行像 PARTOPTIMIZE()函数这样的改造。其实 PARTOPTIMIZE()完全可以作为算法框架来套用，该过程外部的算法可以用以下伪代码表述:

```
GREEDY_R()
1  bA←最差解
2  for rate←0 to 100
3    A←{}
4    while 未得到全局解 do
```

```
5    a←PARTOPTIMIZE(rate)
6    A←A∪{a}
7   if A 优于 bA
8    bA←A
9  return bA
```

我们从《并行算法》问题的例子中看出，在解决只需求较优解的问题中(如IOI'97 中的《有害的千足虫》、《在地图上标地名》等问题，CTSC'98 中的《设置站牌》)，执行结果有优劣变化的随机化算法不失为一种有效算法。

5．总结

经过以上对三种情况的随机化算法的分析，我们作如下总结。

随机化算法的原理与设计

随机化算法的基本原理是：当某个决策中有多个选择，但又无法确定哪个是好的选择，或确定好的选择需要付出较大的代价时，如果大多数选择是好的，那么随机地选一个往往是一种有效的策略。通常当一个算法需要作出多个决策，或需要多次执行一个算法时，这一点表现得尤为明显。

这个原理使得随机化算法有一个共同的性质：没有一个特别的输入会使算法执行出现最坏情况。最坏情况可以表现在执行流程中(主要是时间效率)，也可以表现在执行结果中。

根据这个原理，我们设计随机化算法时，通常一某个算法为母板，加入随机因素，使得算法在难以作出决策时随机地选择。在设计执行结果受随机影响的算法时，我们还可以多次执行算法，使算法不断逼近正确解或最优解[6]。

以上只是设计随机化算法的基本方式。实践中，随机化算法的设计没有公式可套。我们必须具体问题具体分析，深入研究，设计出好的随机化算法。同时在设计中，我们还需特别注意一个问题——

随机化算法的适用范围和有效性

最后我们考虑对于随机化算法的一个至关重要的问题——随机化算法的适用范围和有效性。

一个问题对算法的所有要求除了问题本身的描述之外，还有诸如内存空间限制，时间限制，稳定性限制等要求。如果一个问题对算法不强求 100%的稳定，即对于同样的输入，不必每次运行的情况都相同(当然这种不稳定性不能太大)，同时作为补偿的，又要求算法在其它某些方面有较好的性能(如出解迅速)，而这些性能是一般非随机化算法无法达到的，那么此时，随机化算法可能就是能有效解决问题的候选算法之一。否则，随机化算法便不适用。

当一个随机化算法适用于解决某个问题，且该算法有较高的稳定性，同时它在其它某些方面有突出表现(如速度快，代码短等)，能比一般非随机化算法做得更出色，那么这个随机化算法就是一个行之有效的算法。

[附录]

[1] 严格地说，RANDOM(N)返回的数是混沌数，而不是随机数。

[2] 这里不能说“一定”。《随机算法的定义》一目中解释了这一点。

[3] 判定性问题是这样的问题，它要求判定输入数据(问题无输入的话，可把问题中的已知条件看作输入)是否满足某一特定的条件。

[4] 证明过程如下：

记 T(n)为 QUICKSORT(A,lo,hi=lo+n-1)的时间复杂度。显然 T(n)依赖于划分时用的 x=A[lo]在排序后的 A[lo..hi]中的位置。若 PARTITION(A,lo,hi)的返回值为 p，则 T(n)=T(p-lo+1)+T(hi-p)+Θ(n)，其中Θ(n)是 PARTITION(A,lo,hi)的时间复杂度。

在最坏情况下，

$$T(n) = \max_{q=1,2,\ldots,n-1}\{T(q)+T(n-q)\}+\Theta(n)$$

可以证明 $T(n)<=cn^2$，其中 c 为常数，而且当每次划分都使 q=1 时，$T(n)=\Theta(n^2)$，所以在最坏情况下，快速排序的时间复杂度为Θ(n^2)。

证明 $T(n)<=cn^2$ 可以用数学归纳法。以下为简要过程。

简证：$T(n) = \max_{q=1,2,\ldots,n-1}\{T(q)+T(n-q)\}+\Theta(n)$

由归纳假设，

$$T(n) <= \max_{q=1,2,\ldots,n-1}\{cq^2+c(n-q)^2\}+\Theta(n)$$

$$= c\cdot\max_{q=1,2,\ldots,n-1}\{q^2+(n-q)^2\}+\Theta(n)$$

而 max{}中的关于 q 的函数在 q=1,n-1 时取得最大值，故

$T(n)<=cn^2-2c(n-1)+\Theta(n)<=cn^2$

其中选择适当的 c，可消去-2c(n-1)+Θ(n)。

证毕。

在最佳情况下，每一次划分都将 A 分成相同大小的两部分。这时，

$T(n)=2T(n/2)+\Theta(n)$

由于排序过程中形成的递归树有Θ($\log_2 n$)层，每层的时间代价均为Θ(n)，因此有 $T(n)=\Theta(n\log_2 n)$。上面的分析中忽略了 n/2 不是整数的情况，但这对分析结果没有影响。

我们已经假设输入中出现各种排列都是等概率的。对于平均情况，

$$T(n)=\frac{1}{n}\sum_{q=1}^{n-1}(T(q)+T(n-q))+\Theta(n)$$
$$=\frac{2}{n}\sum_{q=1}^{n-1}T(q)+\Theta(n)$$

可以证明 T(n)<=anlog₂n+b，其中 a,b 为常数，所以快速排序的平均时间复杂度为 O(nlog₂n)。

证明 T(n)<=anlog₂n+b 可以用数学归纳法。以下为简要过程。

简证：$T(n)=\frac{2}{n}\sum_{q=1}^{n-1}T(q)+\Theta(n)$

由归纳假设，

$$T(n)<=\frac{2}{n}\sum_{q=1}^{n-1}(aq\log_2 q+b)+\Theta(n)$$
$$=\frac{2a}{n}\sum_{q=1}^{n-1}q\log_2 q+\frac{2b(n-1)}{n}+\Theta(n)$$

第一项中的

$$\sum_{q=1}^{n-1}q\log_2 q=\sum_{q=1}^{\lceil n/2\rceil-1}q\log_2 q+\sum_{q=\lceil n/2\rceil}^{n-1}q\log_2 q$$

等号右边的第一个和式中的 log₂q<=log₂(n/2)=log₂n-1，第二个和式中的 log₂q<=log₂n。这样，

$$\sum_{q=1}^{n-1}q\log_2 q<=(\log_2 n-1)\sum_{q=1}^{\lceil n/2\rceil-1}q+\log_2 n\sum_{q=\lceil n/2\rceil}^{n-1}q$$
$$=\log_2 n\sum_{q=1}^{n-1}q-\sum_{q=1}^{\lceil n/2\rceil-1}q$$
$$<=\frac{n(n-1)}{2}\log_2 n-\frac{1}{2}\cdot\frac{n}{2}(\frac{n}{2}-1)$$
$$<=\frac{n^2}{2}\log_2 n-\frac{n^2}{8}$$

由此可得，

T(n)<=anlog₂n+b+(Θ(n)+b-an/4)

<=anlog₂n+b

其中选择适当的 a,b，可消去Θ(n)+b-an/4。

证毕。

[5] 运行环境为 Pentium 100MHz，BP7.0 编译。

[6] 如果将执行结果确定的随机化算法执行多次，时间效率可能很低，所以这类随机化算法一般只执行一次。

[参考书目]

1．《实用算法与程序设计》 吴文虎等著 电子工业出版社

2．《计算机数据结构和实用算法大全》 北京希望电脑公司 谋仁主编

3．《组合数学》 吴文虎等著 电子工业出版社

4．《组合数学》 卢开澄著 清华大学出版社

5．《数据结构》 施伯乐等编 复旦大学出版社

6．《中学数学竞赛导引》 上海教育出版社