

实验报告

--五种排序算法比较

1 实验目的

比较 Insertion Sort (插入排序), Shell Sort (希尔排序), Quick Sort (快速排序), Merge Sort (归并排序) 以及 Radix Sort (基数排序) 对 32 位无符号整数的排序效果。

2 数据描述

- (1) 输入数据随机产生, 数据范围为 $[0, 2^{32}-1]$;
- (2) 输入数据量分别为: $10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 2 \times 10^8, (10^9, \text{此数量级选做})$ 。

3 实验环境

- (1) 操作系统: windows10 专业版 (64 位)
- (2) 处理器: Inter(R) Core(TM) i5-4690K CPU @3.5GHz
- (3) 内存: 16G (15.9G 可用)
- (4) 编程环境: Visual Studio 2013
- (5) 编程语言: C++
- (6) 编译版本: Release

4 算法分析

插入排序在数据量比较小的时候效率较高, 空间开销很小, 但平均时间复杂度高, 不适用于数据量大的情况; 希尔排序是直接插入排序的一种改进算法, 减少了其复制的次数, 速度要快很多, 其时间复杂度与步长选取有关^[2]; 快速排序的空间开销也很小, 由于其 cache 命中率很高, 使其与现代计算机的体系结构有很好的契合性, 效率很高; 归并排序采用分治思想, 将待排序数据分为左右两部分, 递归排序, 再将排序结果合并到一个集合中, 其需要一个大小为 n 的临时存储空间用以保存合并序列; 基数排序与前面四种排序算法不同, 并不是基于比较的排序算法, 它对待排序数的每一位进行计数排序, 计数排序的运行时间为 $\theta(n)$, 在一定条件下, 基数排序的效率很高, 可达 $\theta(n)$ 。

五种算法的特性如表 4-1 所示。

表 4-1 五种算法特性表

算法	时间复杂度	空间复杂度	稳定性
----	-------	-------	-----

	平均情况	最坏情况	最好情况		
Insertion Sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n)$	$\theta(1)$	稳定
Shell Sort	$\theta(n \lg^2 n)$	$\theta(n^{1.5})$	$\theta(n)$	$\theta(1)$	不稳定
Quick Sort	$\theta(n \lg n)$	$\theta(n^2)$	$\theta(n \lg n)$	$\theta(1)$	不稳定
Merge Sort	$\theta(n \lg n)$	$\theta(n \lg n)$	$\theta(n \lg n)$	$\theta(n)$	稳定
Radix Sort	$\theta(\frac{b}{r}(n+2^r))$	$\theta(\frac{b}{r}(n+2^r))$	$\theta(n)$	$\theta(n+r)$	稳定

5 算法实现

5.1 随机数生成

本实验要求生成随机数的数据范围为 $[0, 2^{32}-1]$ ，而 C++ 的 rand() 函数只能产生 0~RAND_MAX(如在 Visual Studio 2013 上面最大为 0x7FFF, 即 32767)范围内的随机数，生成 $[0, 2^{32}-1]$ 范围内的随机数常用的有三种方法^[1]：

(1) Mersenne twister 生成算法：号称是目前最好的随机数生成算法，它是由 Takuji Nishimura 和 Makoto Matsumoto 于 1997 年开发的一种随机数生成方法，它基于有限二进制字段上的矩阵线性再生，可以快速产生高质量的伪随机数，该算法的循环周期为 $2^{19937}-1$ 。

(2) Windows API CryptGenRandom 方法：这个函数是 windows 用来提供给生成随机密码数字功能所用，一般应用在随机给 vector 赋值或者生成噪声点时使用。这个函数所提供的随机性要好于 rand() 函数。

(3) rand() 函数拼接法：使用 rand() 函数随机生成的三个数来拼接成一个 32 位数字的方法，这种方法实现起来也很简单，但是由于 rand() 的随机性并不是很好，所以不推荐这种方法。一个 32 位随机数由 $(\text{rand()} \ll 17) | (\text{rand()} \ll 2) | (\text{rand}())$ 方法拼合而成。

本实验采用方法 (1) Mersenne twister 生成算法的变种 Mersenne Twister MT19937 算法，保证测试数据集有较好的随机性。

5.2 Insertion Sort 实现

按照《算法导论》翻译版第 10 页的伪码实现，不再赘述。

5.3 Shell Sort 实现

希尔排序实现时选取不同步长序列，其算法效率也不同，如表 5-1 所示^[2]。

表 5-1 希尔排序性能与步长关系表

General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \dots, 1$	$\Theta(N^2)$ [when $N = 2^p$]	Shell, 1959 ^[4]
$2 \left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2 \left\lfloor \frac{N}{4} \right\rfloor + 1, \dots, 3, 1$	$\Theta(N^{3/2})$	Frank & Lazarus, 1960 ^[8]
$2^k - 1$	1, 3, 7, 15, 31, 63, ...	$\Theta(N^{3/2})$	Hibbard, 1963 ^[9]
$2^k + 1$, prefixed with 1	1, 3, 5, 9, 17, 33, 65, ...	$\Theta(N^{3/2})$	Papernov & Stasevich, 1965 ^[10]
Successive numbers of the form $2^p 3^q$	1, 2, 3, 4, 6, 8, 9, 12, ...	$\Theta(N \log^2 N)$	Pratt, 1971 ^[1]
$\frac{3^k - 1}{2}$, not greater than $\left\lfloor \frac{N}{3} \right\rfloor$	1, 4, 13, 40, 121, ...	$\Theta(N^{3/2})$	Pratt, 1971 ^[1]
$\prod_I a_q$, where $a_q = \min \left\{ n \in \mathbb{N}; n \geq \left(\frac{5}{2} \right)^{q+1}, \forall p: 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1 \right\}$ $I = \left\{ 0 \leq q < r \mid q \neq \frac{1}{2}(r^2 + r) - k \right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	1, 3, 7, 21, 48, 112, ...	$O\left(N^{1 + \sqrt{\frac{8 \ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgwick, 1985, ^[11] Knuth ^[3]
$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	1, 8, 23, 77, 281, ...	$O(N^{4/3})$	Sedgwick, 1986 ^[6]
$9(4^{k-1} - 2^{k/2}) + 1, 4^{k+1} - 6 \cdot 2^{(k+1)/2} + 1$	1, 5, 19, 41, 109, ...	$O(N^{4/3})$	Sedgwick, 1986 ^[12]
$h_k = \max \left\{ \left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1 \right\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11} \left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \dots, 1$	Unknown	Gonnet & Baeza-Yates, 1991 ^[13]
$\left\lfloor \frac{9^k - 4^k}{5 \cdot 4^{k-1}} \right\rfloor$	1, 4, 9, 20, 46, 103, ...	Unknown	Tokuda, 1992 ^[14]
Unknown (experimentally derived)	1, 4, 10, 23, 57, 132, 301, 701	Unknown	Ciura, 2001 ^[15]

本实验选取四种比较有代表性的步长序列进行算法性能比较。

- (1) $\left\lfloor \frac{N}{2^k} \right\rfloor$ ，最坏情况 $O(n^2)$ ，Shell, 1959
- (2) $2^k - 1$ ，最坏情况 $O(n^{1.5})$ ，Hibbard, 1963
- (3) $9(4^{k-1} - 2^{k/2}) + 1, 4^{k+1} - 6 \times 2^{(k+1)/2} + 1$ ，最坏情况 $O(n^{4/3})$ ，Sedgwick, 1986
- (4) Ciura, 2001（论文中只到 1750，前面的数据按 $h(k) = \text{INT}(2.25 * h(k-1))$ 求得）

5.4 Quick Sort 实现

按照《算法导论》翻译版第 95 页的伪码实现，不再赘述。

5.5 Merge Sort 实现

基本按照《算法导论》翻译版第 17 页的伪码实现，区别有两点：

- (1) 在 Merge 时以左右数组的长度为结束标志，而不是将最后一个值设为 ∞ ；
- (2) 若左右任意一侧元素全部被处理，则结束比较，将另一侧数据全部加入尾部。

5.6 Radix Sort 实现

按照《算法导论》翻译版第 109 页与 111 页的伪码实现，主要思路是将 32 位二进制数看作几个 r 位数拼接而成，然后分别对每一段 r 位二进制数进行计数排序。

根据《算法导论》的描述，理论上，基数排序中如果 $b = O(\lg n)$ ，我们选择 $r \approx \lg n$ ，则基数排序的运行时间可达 $\theta(n)$ ，这里 n 多达 2×10^8 ，直接使用理论值欠妥，首先需要对这个结论进行验证，确认基数排序中最佳的 r 取值。

这里在实现的时候，为了减少过程中数据复制的次数，使用目标数组 A 与临时数组 B 交替存放过程排序结果。

6 实验过程

6.1 数据采集

因为实现时采用的 `time(&t)` 作为种子，每次生成的随机数都是不同的。所以在取 r 值和排序的耗时数据之前，对于每一个数量级，都先生成对应数量级的随机数写入到文件中，如图 6-1 所示，目的是在每次排序后可以从文件中读取并恢复数组的初始状态，以保证初始条件的一致性。











 1_000.txt	周日 2016 10 16 ...	TXT 文件	12 KB
 1_000_000.txt	周日 2016 10 16 ...	TXT 文件	11,467 KB
 1_000_000_000.txt	周日 2016 10 16 ...	TXT 文件	11,466,12...
 10.txt	周日 2016 10 16 ...	TXT 文件	1 KB
 10_000.txt	周日 2016 10 16 ...	TXT 文件	115 KB
 10_000_000.txt	周日 2016 10 16 ...	TXT 文件	114,661 KB
 100.txt	周日 2016 10 16 ...	TXT 文件	2 KB
 100_000.txt	周日 2016 10 16 ...	TXT 文件	1,147 KB
 100_000_000.txt	周日 2016 10 16 ...	TXT 文件	1,146,612...
 200_000_000.txt	周日 2016 10 16 ...	TXT 文件	2,293,224...

图 6-1 生成测试数据样例示意图

每次针对一个数量级进行测试，生成数据后存入 excle 表格。

6.2 最终可执行程序

最终实现的可执行程序目的是便于展示，与采集数据时所用的程序有以下区别：

- (1) 最终可执行程序在调用每个排序算法前，随机数集都是新生成的；
- (2) 最终可执行程序没有文件读写操作。

最终实现效果如图 6-2 所示。

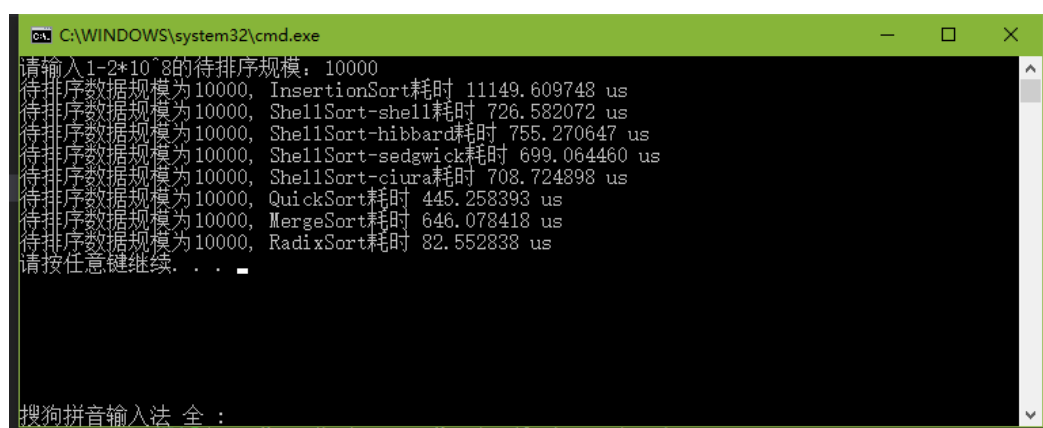


图 6-2 可执行程序效果图

7 实验结果及分析

7.1 Radix Sort 算法 r 值分析

取 $r=1,2,\dots,16$ ，分别测试其不同输入规模下计数排序的耗时，计算平均数据整理如表 7-1 所示。表中同一输入数量级下使用的数据样本相同，每个 r 值对应的耗时值均是三次实验数据的平均值（见附件数据分析.xlsx），单位为 us，具有较高的准确性与可对比性。

表 7-1 RadixSort 算法 r 值分析数据表

	10	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶	10 ⁷	10 ⁸	2*10 ⁸
1	4.489	14.052	110.851	1175.939	12043.737	119508.895	1207173.373	11962244.176	23794070.481
2	1.269	5.952	49.473	517.858	5222.004	53584.403	537420.928	5484698.061	10762550.129
3	0.683	3.903	31.323	329.431	3153.596	34097.640	347888.101	3623551.544	7213067.314
4	0.585	2.830	20.297	197.112	2339.875	22153.532	227493.081	2259392.435	4698455.755
5	0.585	2.537	28.493	182.768	2117.783	18789.943	202184.001	1973899.154	4150290.174
6	0.878	2.537	15.320	149.103	1503.613	17157.232	180311.597	1862543.377	3717975.988
7	1.561	2.830	20.785	125.586	1465.947	14162.984	157256.131	1577343.227	3080634.704
8	2.244	3.220	10.344	84.309	961.555	11294.321	121469.085	1246010.580	2486128.196
9	4.196	5.367	12.881	99.434	1286.497	13029.687	135713.060	1389939.502	2853176.655
10	9.660	11.319	21.858	168.619	1922.135	16413.280	179971.920	1891393.545	3775488.774
11	18.931	15.223	26.444	103.337	1493.855	16614.881	200528.455	2274823.277	4718878.214
12	26.932	30.835	45.668	117.291	1867.490	18141.523	226870.324	3206401.026	6050762.970
13	53.572	125.000	87.432	153.298	2165.304	22466.472	268039.112	3583422.570	7279371.685
14	107.924	118.072	139.052	222.093	2290.305	26200.768	301271.216	3920404.427	8331330.793
15	236.339	233.412	279.372	396.078	2550.649	29658.327	326339.176	4296719.491	9875400.054
16	321.917	372.951	406.129	436.086	2475.805	27792.106	314725.280	4137539.078	10161137.090

将表中数据绘制成折线图，如图 7-1 所示。

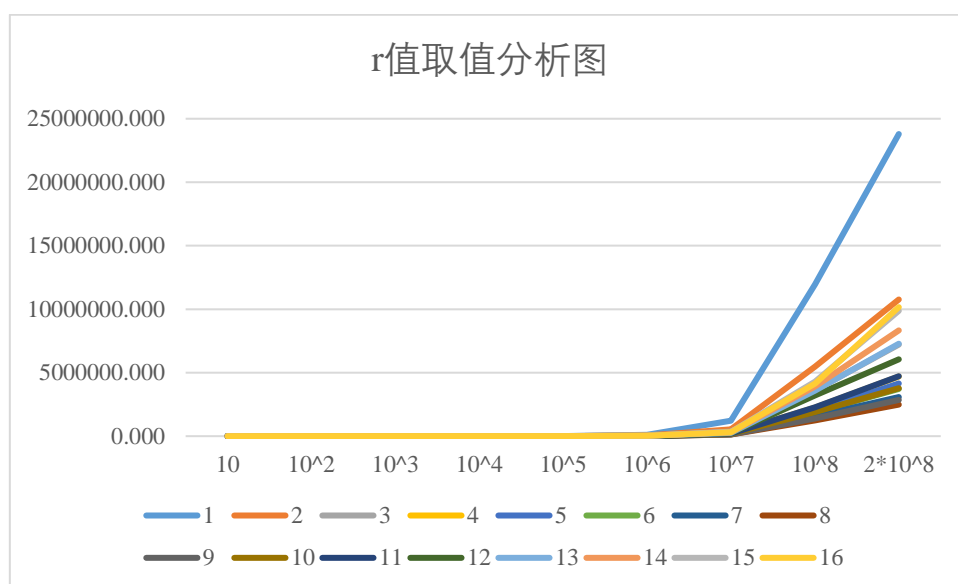


图 7-1 RadixSort 算法 r 取值分析图

从图表中可以看出，在 n 规模较小时， r 的最佳取值出现在 4-6 区间，但当输入规模达到 10^3 时， r 的最佳取值便收敛到 8，不再浮动，总体而言， r 最佳实验值为 8，此时基数排序的耗时最短，效率最高。表 7-2 是理论值与实验值的对比。

表 7-2 r 理论值实验值对比表

n	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8	2×10^8
lgn	3	6	9	13	16	19	23	26	27
best r	4	6	8	8	8	8	8	8	8

由表可知，当 n 规模很大时， r 的理论取值便不再适用于实际情况。其主要原因可能是受限于计算机高速 cache 的大小，当 r 值较大时，使得计数排序中 k 的取值增大，辅助数组的空间增大，一次对 r 位数的排列便需要占用较大的 cache 空间，当达到一定限度时，cache 的空间不能满足一次计数排序所需要的辅助空间大小，排序的效率便开始下降。

综上，这里选择 $r=8$ 作为最终算法使用值。

7.2 排序算法比较

分别测试共计 8 种排序算法在不同输入规模下的排序耗时，计算平均数据整理如表 7-3 所示。表中同一输入数量级下使用的数据样本相同，每个 r 值对应的耗时值均是三次实验数据的平均值（见附件数据分析.xlsx），单位为 us。这里插入排序只计算到 10^6 ， 10^7 运行时间过长，单方面拉高纵坐标轴，不利于折线图的观察，所以输入规模大于等于 10^7 之后不再考虑插入排序。

表 7-3 排序算法耗时记录表

	10	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶	10 ⁷	10 ⁸	2*10 ⁸
insertion sort	0.195	1.952	115.828	11067.545	1149289.880	117659473.055			
shell sort shell	0.293	3.513	48.595	774.884	10095.353	141801.674	2081008.241	30201235.424	67804893.783
shell sort hibbard	0.293	3.220	49.083	748.538	11212.256	158670.361	2360099.188	38300704.568	86239040.916
shell sort sedgwick	0.195	2.732	43.618	765.712	8967.034	115779.965	1468090.889	17888427.985	36862985.903
shell sort ciura	0.195	2.830	45.375	656.324	9044.122	116525.283	1458385.758	16890163.937	35079618.212
quick sort	0.293	2.732	34.251	482.534	5417.164	65046.367	780291.574	8776407.687	17896260.259
merge sort	2.440	7.221	64.793	649.591	7995.428	93934.786	1117682.876	12524497.506	25630856.885
radix sort	2.537	3.318	10.441	85.773	1054.061	10313.348	117333.052	1181455.139	2295801.359

将上表数据绘制成折线图，如图 7-2 所示。

分析表 7-3 和图 7-2 中的数据，可以总结出以下观点：

- (1) n 较小时 (10^2 规模以下)，插入排序性能最好，希尔排序 (sedgwick 步长序列与 ciura 步长序列) 也有很好的表现。
- (2) n 到达 10^3 规模以上时，插入排序的效果直线下降， 10^5 规模以上便不再适用，同时基数排序与快速排序的优势开始变得明显。

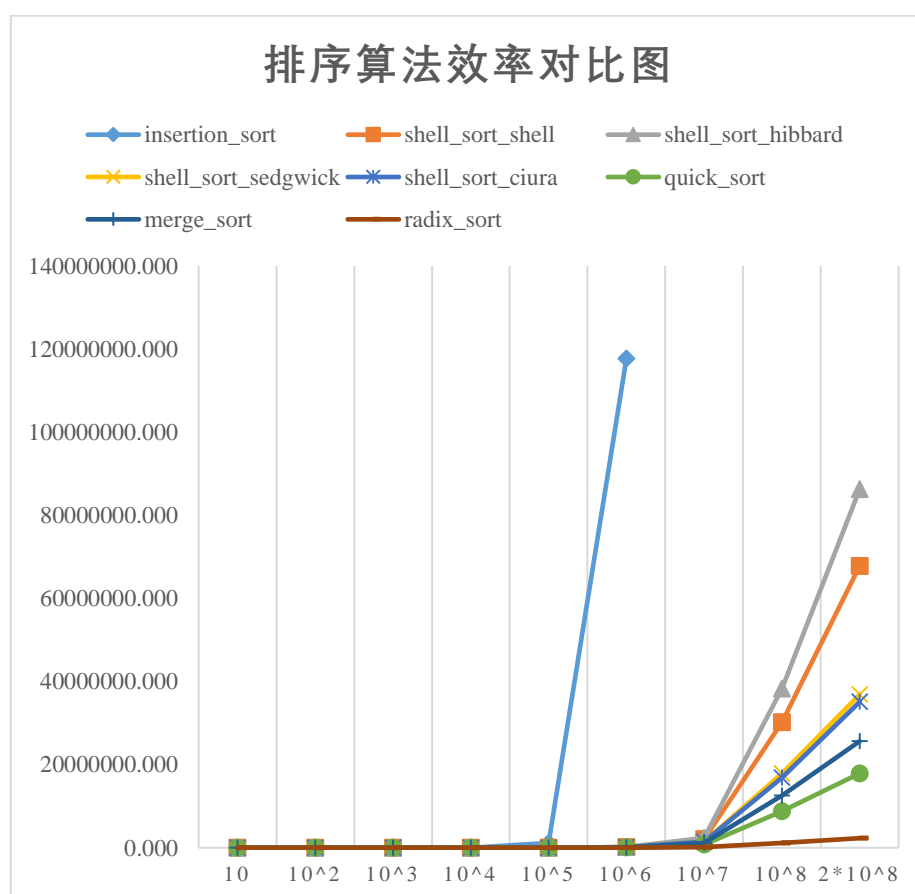


图 7-2 排序算法对比折线图

- (3) 希尔排序选择 4 种不同的步长，当数据输入规模足够大时，效率差距明显，使用 sedgwick 序列与 ciura 序列的希尔排序算法耗时大约只有使用普通步长

序列时的一半，实验证明 $2^k - 1$ 序列效率并没有比 $\left\lfloor \frac{N}{2^k} \right\rfloor$ 序列高多少，甚至在数据规模较大时，表现更差。

- (4) 快速排序和归并排序虽然理论上有着相同的时间复杂度，相同的平均效率，但是实际来看，快速排序的效果要比归并排序好很多，关键在于归并排序空间复杂度较高，有较多的赋值操作，其 cache 命中率不如快速排序。
- (5) 基数排序非比较的排序方式显示出很好的效果，即使在数据达到 2×10^8 规模时，依旧耗时很短，其所耗时间比快速排序小一个数量级。
- (6) n 数量级足够大时，按时间效率从高到低对算法进行排序，序列如下：基数排序 > 快速排序 > 归并排序 > 希尔排序 > 插入排序。

8 实验结论

- (1) 基数排序 r 值得选取要考虑到硬件的限制，理论值不一定是最佳选择，像本实验中的最佳值为 8，并不等于 $\lg n$ ；
- (2) 插入排序小规模数据排序效率很高，不适合于大规模数据的排序；
- (3) 希尔排序采用不同的步长序列，会有差异较为明显的表现，在数据规模适中时，采用合适序列的希尔排序效率可与快速排序媲美，但是数据规模较大时，依旧不如快速排序；
- (4) 快速排序很适合于大规模数据的排序，相比同样时间复杂度的归并排序有着更好的表现，但是如果在空间条件允许的情况下，基数排序的表现优于快速排序，因此条件允许的情况下基数排序也是很好的选择。

参考资料

- [1] 使用 C++ 如何产生 32 位随机数,
<http://www.cnblogs.com/thu539/archive/2011/11/14/2247717.html>
- [2] Shellsort, <https://en.wikipedia.org/wiki/Shellsort>