算法分析与实践 排序专题

PB20061210 赖永凡

实验一

实验描述

随机生成一个包括n个整数的数组（元素取值范围是1~1000），利用插入排序、归并排序、快速排序、堆排序、基数排序、桶排序等算法对数组进行非降序排序，记录不同算法的运行时间。

实验内容

使用如下代码生成1000个取值范围为1~1000的随机数，运用插入排序、归并排序、快速排序、堆排序、基数排序、桶排序六种算法的排序时间如下

```
// Generate random sequence
std::uniform_int_distribution<int> u(MINNUM, MAXNUM);
std::default_random_engine e(0);

for (int i = 0; i < n; i++) {
    Seq[i] = u(e);
    if (print_seq) std::cout << Seq[i] << " ";
}
if (print_seq) std::cout << std::endl;
```

n = 1000	插入	归并	快速	堆	基数	桶
时间（微秒）	495	248	69	130	73	108

实验二

实验描述

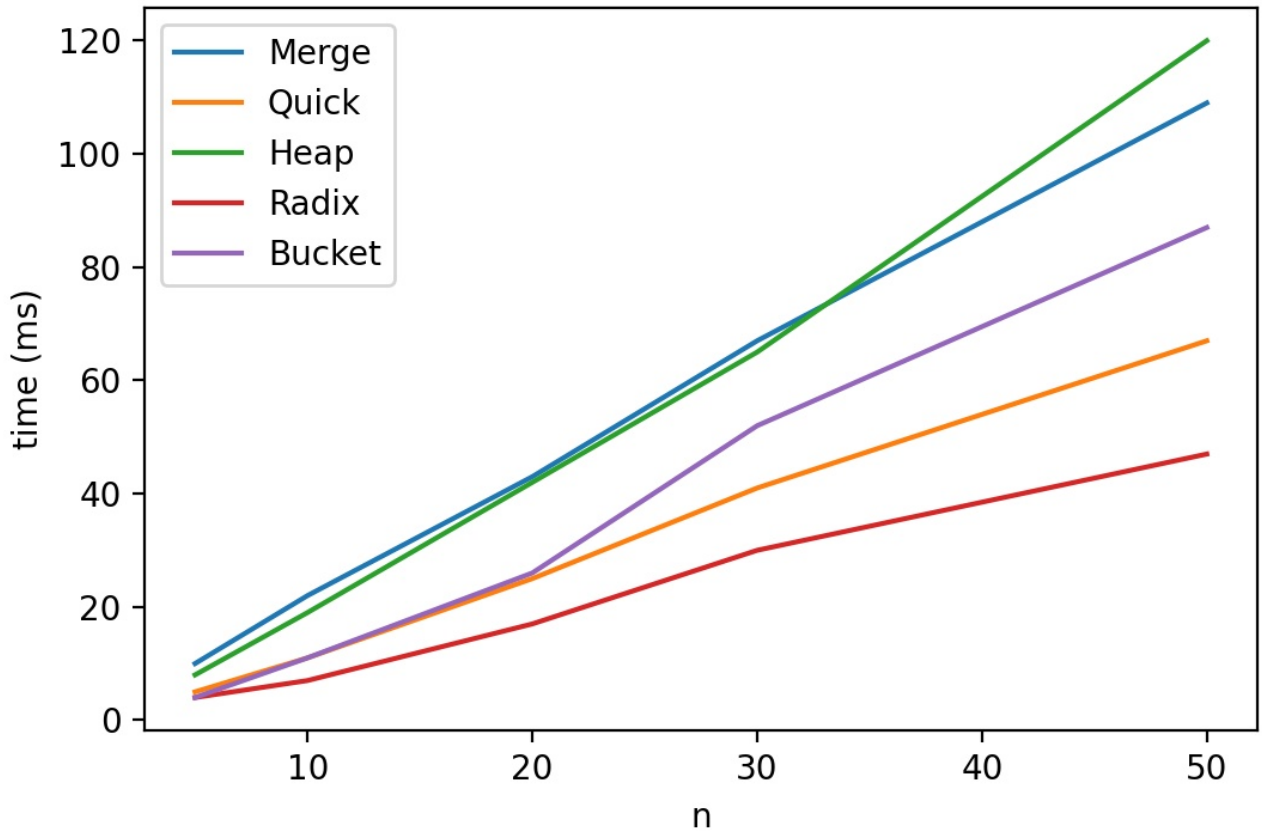
改变数组规模 n = 5万、10万、20万、30万、50万，记录不同规模下各个算法的排序时间。

实验内容

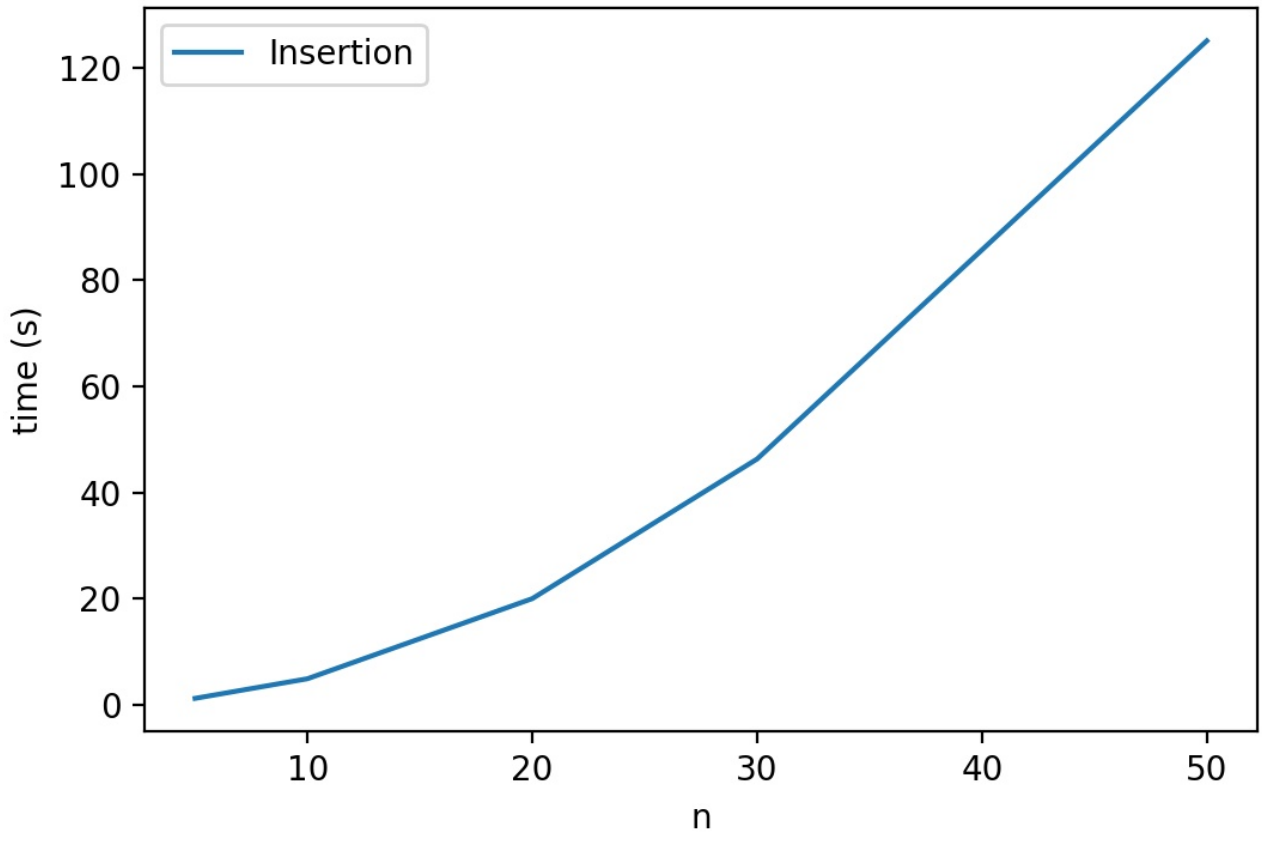
按照实验描述中的说明改变n的大小，记录的算法运行时间如下（单位 ms）：

n（万）	插入	归并	快速	堆	基数	桶
5	1227	10	5	8	4	4
10	4927	22	11	19	7	11
20	20027	43	25	42	17	26
30	46342	67	41	65	30	52
50	125136	109	67	120	47	87

绘制出 n-耗费时间 折线图如下：



其中插入排序的单独列出：



实验三

实验描述

对固定规模 （n = 10万）的数组进行随机扰乱，对扰乱后的数组进行排序并记录各个算法的排序时间。本实验要求重复5次，观察输入数据分布和运行时间的关系。

实验内容

采用如下方法扰乱数组，若degree为1，则随机扰乱数组，即在遍历时让当前元素与随机选择的一个其之后的元素进行交换；若degree为-1，则输出逆序的数组（非增序列）；若degree为0，则输入已经按要求排好序的序列。

```
int n = 100000;
int *Seq_ordered = new int[n];
for(int i=0; i < n; i++) {
    Seq_ordered[i] = i + 1;
}
std::default_random_engine e(2);
int *Seq_spoiled = new int[n];
memcpy(Seq_spoiled, Seq_ordered, n * sizeof(int));
if(degree > 0)
    for(int i=0; i < n; i++) {
        std::uniform_int_distribution<int> u(i, n - 1);
        swap(&Seq_spoiled[i], &Seq_spoiled[u(e)]);
    }
if(degree < 0)
    for(int i=0; i < n / 2; i++) {
        swap(&Seq_spoiled[i], &Seq_spoiled[n - i - 1]);
    }
```

分别测试各算法的运行时间，得到如下的表格（单位 ms）：

n = 10万	插入	归并	快速	堆	基数	桶
顺序	0.36	17	7932	15	7	8
扰乱	4979	17	12	22	7	11
逆序	9806	16	8331	14	7	6

从表中可以看出，插入排序、快速排序对数据的分布比较敏感，具体的分析将在下一部分中给出。

实验四

实验描述

这一部分对前面实验中用到的排序算法进行时间复杂度分析。

实验内容

一、插入排序。
插入排序运行时从第二个元素开始，始终维护遍历元素及其之前的序列是有序的，因此对于位置i上的元素就需要至多i次比较操作和交换。当输入数据为顺序的序列时，插入排序在遍历期间只需一次比较且无需进行交换，因此时间复杂度为 $O(n)$ 。相反地，当输入逆序序列时，插入排序需要完整地执行全部的 n(n - 1)/2 次交换操作，达到了 $O(n^2)$ 的量级。实验三的结果和上述分析契合。对于一般情况，对于数据i，其期望的比较和交换次数为其之前元素的一半，也即 i / 2 次，因此平均的时间复杂度也在 $O(n^2)$ 量级，实验二中绘制的插入排序的折线图很好地反应了这一二次函数关系。

二、归并排序
归并排序的本质是分治策略，其中使用 `MERGE()` 函数来对两个有序的序列进行合并。对于合并操作，我们需要额外使用两个辅助队列来分别储存有序的子列，其中小端位于队列头。之后每次将两个队列头中更小的元素出列，并放到原序列中的相应位置，这一操作本质遍历上两个子序列，因此复杂度为 $O(n)$ 。对于归并排序的时间复杂度，可以用主方法求解：

$$T(n) = 2T(\frac{n}{2}) + O(n)$$
$$T(n) = O(n \log n)$$

三、快速排序
原始的快速排序可以看做归并排序的进一步，其首先取序列中最后一个元素作为主元，之后从左往右进行遍历，通过交换将数组划分成小于主元和大于

主元的两个子序列。之后对两个子序列分别调用快速排序函数。对于一个随机扰乱的数组，快速排序将会产生一个一定比例的划分，同样可以证明这将产生深度为 $O(\log n)$ 的递归树，而每一层中所有内部节点的划分代价和都为 $O(n)$ ，因此算法的平均时间复杂度为 $O(n \log n)$。
然而，对于输入顺序或逆序序列时，算法的递归式将变为 $T(n) = T(n-1) + \Theta(n)$ ，这是一个 $O(n^2)$ 量级的复杂度，从实验三的数据来看也确实如此。此外，遇到最差情况时原始的快速排序由于递归栈过深将会导致堆溢出，这里我采用了优化后的快速排序，其核心思想便在于用迭代来代替其中的一次递归。

四、堆排序
堆排序可以分为建堆和排序两部分，但两者的核心都是维护大根堆的操作 `MAX-HEAPIFY()` 。假设给定的堆中，仅有节点i不满足大根堆的性质，于是我们对i和其两个孩子节点进行比较，让i与更大的孩子节点进行交换，从而节点i满足大根堆的要求。之后我们再递归地对被交换的子节点使用这一操作，直到其已经满足大根堆操作或达到了叶子节点。不难证明，此操作的时间复杂度与堆的高度相关，因此为 $O(\log n)$ 。考虑到递归操作的效率较低，在实现时我采用了迭代的版本。
在建堆的过程中，我们从第一个内节点开始从后往前遍历，依次对每个节点调用维护大根堆的操作，这样一来就将随机扰乱的数组转化成了一个大根堆。对于建堆的时间复杂度，直观上来看需要遍历 n / 2 个节点，每次维护最大堆的操作为 $O(\log n)$，因此总复杂度为 $O(n \log n)$。其实，还可以通过精确的证明，将时间复杂度转化为 $O(n)$ 的量级。这也可以理解，对于在 n / 2 个内节点中占了多数的最后两层内节点，实际上维护最大堆的操作分别仅需迭代1次和2次，因此在整个建堆操作中占主导的还是遍历数组花费的时间。此外，从实验三中可以发现，当输入数据为逆序时，数组天然是大根堆，因此省去了在建堆上花费的时间。
根据大根堆的性质，其首元素一定是数组中的最大元素。因此我们每次将首元素和堆中最后一个元素进行交换，之后将堆的长度减一，再对新的首元素调用一次维护最大堆的操作，重复n - 1次即达成了排序的要求。排序需要遍历数组，每次维护大根堆的操作均为 $O(\log n)$，因此总时间复杂度为 $O(n \log n)$。
值得注意的是，堆这一数据结构一个重要的性质便是可以通过下标的操作来实现对节点前驱和后继的访问，但这也就要求数组下标从1开始，为了在C中实现堆排序的算法，在逻辑上我们令数组下标从1开始，但每次访问时都会在逻辑下标上减1来实现兼容。

五、基数排序
讨论基数排序，还得从计数排序开始说起。计数排序的思想本质便是抛弃比较，直接把元素放到其对应的位置上。假设序列的取值范围是 $[0, k-1]$，就新增一个长度为k 的数组C，并将其所有元素初始化为0，之后遍历序列，在C中统计出每个取值下元素的出现次数。之后从左往右遍历数组C，计算 `C[i] = C[i] + C[i - 1]` ，这样一来，C[i]对应的值便是元素i在原序列中最后的位置。我们最后从后往前遍历原数组，在数组C的指示下依次将数据放入辅助数组B中，并将C中对应的值减一，表示下一次遇到同样的元素时就放到其前一个位置上，从而维持排序的稳定性。计数排序一共需要遍历两次原序列和依次数组C，因此时间复杂度为 $O(n + k)$。
对于基数排序，我们每次使用计数排序的方法，根据序列中元素的第d位来对序列排序，其中d从个位起一直到最高位。由于计数排序是稳定的，因此确保了基数排序的正确性。由于数据的分布为 $[1, 10^d]$ ，在计数排序中先将序列中的元素减一，再获得对应位的元素，这样实际上是将数据映射到 $[0, 10^d - 1]$,从而将计数排序的调用次数从d + 1减少为d次。基数排序需要调用d次计数排序，且后者对应的k为10，故总时间复杂度为 $O(d * (n + 10)) = O(n)$。
但是，在实验二里序列元素的最大值与n相关，即 $d = \lceil \lg n \rceil$，因此时间复杂度实际上仍为 $O(n \log n)$。

六、桶排序
桶排序利用了哈希的思想，其核心便在于让每个桶中的元素个数为$O(1)$量级。由于我们的数据是均匀分布的，因此可以根据区间来将数据散落到桶中。假设希望每个桶中储存元素的期望是d，那么每个桶储存的元素的区间就是$[kd, (k+1)d)$，且一共有 $\lceil n/d \rceil$ 个桶。将元素散落到桶中后，调用插入排序对每个桶进行排序，之后再根据桶的标号从小到大依次将每个桶中的元素放回原序列中即可。
由于桶中元素的个数为$O(1)$量级，因此对桶进行插入排序的复杂度也可以认为是$O(1)$。因此，考虑元素散列到桶中和从桶中取出元素的时间后，算法总的复杂度为 $O(2n + c\frac{n}{d}) = O(n)$

七、总结
以上部分分析了相关算法的时间复杂度，其中插入排序的平方关系在实验二中得到了较好的验证。至于其余算法的运行时间，就实验二的结论来看似乎 $O(n)$ 和 $O(n \log n)$ 的差别不是特别显著，个人认为可能存在以下的原因：

1. 由于数据的最大值随着数据的个数增长而增长，因此一些算法（如基数排序）需要的时间会发生相应的变化。
2. 输入数据的规模还比较小，不足以反映出函数的量级关系。log函数在n较小的情况下值差别不大，如 $\log 50000 = 15.6$，而 $\log 500000 = 18.9$ ，因此无法和线性的关系带来较大的差异。此外，虽然是同样的待排数据，对于同一算法每次运行的时间都不同，且存在一定的方差。为了确保获得平均性能，实验中的数据都是多次运行后求平均的结果，虽然如此，也不能确保最终反映的是算法本身确切的性能。