快速排序算法及其优化实验报告

SA20225085 朱志儒

**算法核心思想**

1. 在数据量特别大时采用正常的快速排序，不断地对数据进行分段，同时可以采用三点中值法提高快速排序的效率。
2. 在递归过程中，如果递归的深度过深，深度大于2log n时，表示分段行为有恶化的倾向，于是采用堆排序，使整个算法的时间复杂度维持在堆排序的O(n log n)，这比一开始就采用堆排序要好。
3. 当分段后的数据量小于某个阈值时，跳出递归过程，最后采用插入排序，此时整个数据集是基本有序的，只有各个小分段内是无序的，即时插入排序会造成数据移动，但是移动在各个小分段内出现，不会跨分段移动，但整体的移动量不会很大，这时时间复杂度为O(n)。

**具体实现**

1. **void** mysort(**int**\* array, **int** first, **int** last) {
2. **if** (first != last) {
3. introsort(array, first, last, lg(last - first) \* 2);//快排
4. finalinsertionsort(array, first, last);//插入排序
5. }
6. }

先用if语句判断区间的有效性，接着调用introsort，不断地对数据进行分段并使其基本有序，最后调用插入排序使整个数据集有序。

1. **void** introsort(**int**\* array, **int** first, **int** last, **int** depth) {
2. **while** (last - first > threshold) { //数据长度大于最小分段阈值时，采用递归
3. **if** (depth == 0) {
4. //当递归次数超过阈值时，调用堆排序
5. partialsort(array, first, last, last);
6. **return**;
7. }
8. --depth;//递归深度阈值减1
9. //三点中值法
10. **int** middle = median(array[first], array[first + (last - first) / 2], array[last - 1]);
11. **int** cut = partition(array, first, last, middle);//分区
12. introsort(array, cut, last, depth);//递归调用
13. last = cut;
14. }
15. }

Introsort函数在数据量很大时采用正常的快速排序，当递归的深度大于阈值depth时，表示分段有恶化倾向，于是采用堆排序，不再递归。当数据长度小于最小分段阈值threshold时，不再递归，每个子序列都有相当程度的排序，但又尚未完全排序，过多的递归调用是不可取的。于是，终止快速排序，调用外部的插入排序来处理未完全排序的子序列。

在introsort函数末尾采用三点中值法对右边子序列进行递归调用，终点位置调整到分割点，此时[first, last)区间就是左边子序列，在下一次循环中，左子序列便得到处理，减少了递归调用的时间消耗。

1. **int** partition(**int**\* array, **int** first, **int** last, **int** pivot) {
2. //选择是首尾中间位置三个值的中间值作为pivot
3. //因此一定会在超出此有效区域之前中止指针的移动
4. **while** (**true**) {
5. **while** (array[first] < pivot)
6. first++;
7. --last;
8. **while** (pivot < array[last])
9. last--;
10. **if** (!(first < last))
11. **return** first;
12. **int** tmp = array[first];
13. array[first] = array[last];
14. array[last] = tmp;
15. first++;
16. }
17. }

Partition函数没有对first和last做边界检查，而是将两个指针交错作为中止条件，减少了比较运算的时间。这是因为pivot是首尾中间位置三个值的中位数，所以一定会在超出有效区间之前中止指针的移动。

1. **void** unguardedlinearinsert(**int**\* array, **int** last, **int** value) {
2. //省略越界检查的插入排序
3. **int** next = last;
4. --next;
5. **while** (value < array[next]) {
6. array[last] = array[next];
7. last = next;
8. --next;
9. }
10. array[last] = value;
11. }
13. **void** linearinsert(**int**\* array, **int** first, **int** last) {
14. **int** value = array[last];
15. **if** (value < array[first]) { //将当前值与最左边的值相比较
16. copybackward(array, first, last); //将前面已经排列好的数据整体向后移动一位
17. array[first] = value; //将最小值放在最左边
18. }
19. **else** //已经确保最小值在最左边
20. unguardedlinearinsert(array, last, value); //调用省略越界检查的插入排序
21. }
23. **void** insertionsort(**int**\* array, **int** first, **int** last) {
24. **if** (first == last)
25. **return**;
26. **for** (**int** i = first + 1; i != last; ++i)
27. linearinsert(array, first, i);
28. }

Linearinsert函数先将待插入值与第一个元素进行比较，如果比第一个元素还小，那么就直接将前面已经排列好的数据整体向后移动一位，然后将该元素放在第一个位置，Unguardedlinearinsert函数仅逐个判断是否需要调换，找到位置之后就将其插入到适当位置，并没有检查是否越界，因为Linearinsert函数中的if语句已经可以确保第一个值在最左边了。这与标准插入排序相比，减少了每次移动前与边界比较的次数。

**实验结果分析**

|  |  |
| --- | --- |
| **算法** | **时间** |
| 优化的快排 | 8ms |
| 快速排序 | 9ms |
| 堆排序 | 18ms |
| 冒泡排序 | 22975ms |
| 归并排序 | 24ms |
| 插入排序 | 4775ms |
| STL库中的sort函数 | 10ms |

实验结果表明优化后的快速算法比众多算法都要快，上述的几种优化方法对原本的快速排序还是起作用的。