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
```

```
        int middle = median(array[first], array[first + (last - first) / 2],
    array[last - 1]);
        int cut = partition(array, first, last, middle);//分区
        introsort(array, cut, last, depth);//递归调用
        last = cut;
    }
}
```

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。

```
int partition(int* array, int first, int last, int pivot) {
    //选择是首尾中间位置三个值的中间值作为 pivot
    //因此一定会在超出此有效区域之前中止指针的移动
    while (true) {
        while (array[first] < pivot)
            first++;
        --last;
        while (pivot < array[last])
            last--;
        if (!(first < last))
            return first;
        int tmp = array[first];
        array[first] = array[last];
        array[last] = tmp;
        first++;
    }
}
```

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。

```
void unguardedlinearinsert(int* array, int last, int value) {
    //省略越界检查的插入排序
    int next = last;
    --next;
    while (value < array[next]) {
        array[last] = array[next];
        last = next;
        --next;
    }
    array[last] = value;
}

void linearinsert(int* array, int first, int last) {
    int value = array[last];
    if (value < array[first]) { //将当前值与最左边的值相比较
        copybackward(array, first, last); //将前面已经排列好的数据整体向后移动一位
        array[first] = value; //将最小值放在最左边
    }
    else //已经确保最小值在最左边
        unguardedlinearinsert(array, last, value); //调用省略越界检查的插入排序
}

void insertionsort(int* array, int first, int last) {
    if (first == last)
        return;
    for (int i = first + 1; i != last; ++i)
        linearinsert(array, first, i);
}
```

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。