《数据结构》上机报告

2020年12月31日

姓名: 林日中 学号: 1951112 班级: 10072602 得分:

<table>
<tr><td>实验题目</td><td colspan="2">排序算法实验报告</td></tr>
<tr><td>问题描述</td><td colspan="2">排序是计算机内经常进行的一种操作, 其目的是将一组“无序”的记录序列调整为“有序”的记录序列。分内部排序和外部排序, 若整个排序过程不需要访问外存便能完成, 则称此类排序问题为内部排序。反之, 若参加排序的记录数量很大, 整个序列的排序过程不可能在内存中完成, 则称此类排序问题为外部排序。内部排序的过程是一个逐步扩大记录的有序序列长度的过程。

假定在待排序的记录序列中, 存在多个具有相同的关键字的记录, 若经过排序, 这些记录的相对次序保持不变, 即在原序列中, r[i]=r[j], 且 r[i] 在 r[j] 之前, 而在排序后的序列中, r[i] 仍在 r[j] 之前, 则称这种排序算法是稳定的; 否则称为不稳定的。快速排序, 希尔排序, 堆排序, 直接选择排序不是稳定的排序算法, 而基数排序, 冒泡排序, 直接插入排序, 折半插入排序, 归并排序是稳定的排序算法。

实验目的:
1. 掌握基于比较的各种排序算法的实现;
2. 掌握各种排序算法的特点, 时间复杂度, 稳定性。

实验内容:
(1) 随机生成不同规模的数据 (10, 100, 1K, 10K, 100K, 1M, 10K 正序, 10K 逆序) , 并保存到文件中, 以 input1.txt, input2.txt, ..., input8.txt 命名。
(2) 用上面产生的数据, 对不同的排序方法 (插入排序, 选择排序, 冒泡排序, 希尔排序, 堆排序, 快速排序, 归并排序) 进行排序测试, 给出实验时间。
(3) 分析各种排序算法的特点, 给出时间复杂度, 以及是否是稳定排序。</td></tr>
<tr><td rowspan="2">基本要求</td><td colspan="2">(1) 程序要添加适当的注释, 程序的书写要采用缩进格式。
(2) 程序要具在一定的健壮性, 即当输入数据非法时, 程序也能适当地做出反应, 如插入删除时指定的位置不对等等。
(3) 程序要做到界面友好, 在程序运行时用户可以根据相应的提示信息进行操作。
(4) 根据实验报告模板详细书写实验报告, 在实验报告中给出主要算法的复杂度分析。
(5) 实验结果以表格形式列出, 给出测试环境 (硬件环境和软件环境) , 并对结果进行分析, 说明各种排序算法的优缺点。
(6) 将实验报告和输入文件打包成 .zip 文件, 上传。</td></tr>
<tr><td>已完成基本内容 (序号) :</td><td>(1) (2) (3) (4) (5) (6)</td></tr>
<tr><td rowspan="2">选做要求</td><td colspan="2"></td></tr>
<tr><td>已完成选做内容 (序号) :</td><td>无</td></tr>
</table>

数据结构设计

在本次上机作业中,我设计的数据结构和数据结构的成员变量如下。

- `struct Element` : 表示顺序表元素的结构体
  - `KeyType key` : 本 `Element` 对象参与顺序表排序依据的关键字
- `class SqList` : 表示顺序表的类
  - `RecordType *original` : 按照原始顺序存储的元素数组
  - `RecordType *r` : 各算法函数作用的元素数组
  - `RecordType *correct` : 按照正序存储的元素数组
  - `int length` : 记录本 `SqList` 对象的元素长度

功能(函数)说明

HW9_4_generate_data.cpp

```
int generate_random_data() ···

int main()
{
    srand((unsigned)time(nullptr));
    generate_random_data();
}
```

- `int generate_random_data()` : 包装了分别生成元素数量为 10, 100, 1k, 10k, 100k, 1M, 1k(正序), 1k(逆序) 数据文件的操作
- `int main()` : 初始化随机数发生器, 并提供进入上述函数的入口

HW9_4.cpp

```
struct Element
{
    KeyType key;
    Element(const KeyType &k = NULL_KEY) : key(k) {}
    Element(const Element &e) : key(e.key) {}
    Element &operator=(const Element &e) ···
}; // struct Element

bool comp(const Element &l, const Element &r) ···

void swap(Element &l, Element &r) ···
```

- `Element::Element()` : 构造函数, 负责本 `Element` 对象的初始化, 给成员变量赋初值
- `Element &Element::operator=()` : 对运算符 `=` 的重载函数, 包装了对成员变量的赋值操作, 令对象的赋值操作更方便

```
class SqList
{
protected:
    RecordType *original = nullptr;
    RecordType *r = nullptr;
    RecordType *correct = nullptr;
    int length = 0;
    void recover() ···
    void output_to_file(ofstream &out) ···
    bool check_accuracy() ···
    void shell_insert(const int dk) ···
    void shell_sort(const int delta[], const int t) ···
    int partition(int low, int high) ···
    void qsort(int low, int high) ···
    void heap_adjust(int s, const int m) ···
    // merge [L..M], [M+1..H] the two ordered sequences into one
    void merge(RecordType *SR, RecordType *&TR,
               const int L, const int M, const int H) ···
    void merge_pass(RecordType *SR, RecordType *&TR,
                    const int len, const int n) ···
    void merge_sort(RecordType *R, const int n) ···
```

- `void SqList::recover()` : 将元素数组 `r` 恢复为原顺序
- `void SqList::output_to_file()` : 将元素依次输出到文件流中

- bool SqList::check_accuracy() : 检查 r 和 correct 两个元素数组的一致性
- void SqList::shell_insert() : 类似插入排序的操作过程
- void SqList::shell_sort() : 利用传入的 delta 数组进行希尔排序过程
- int SqList::partition() : 一趟快速排序，返回 pivot
- void SqList::qsort() : 递归函数实现快速排序
- void SqList::heap_adjust() : 堆排序中使用的筛选算法
- void SqList::merge() : 2-路归并排序中的核心操作，将一维数组中前后相邻的有序序列归并为一个有序序列
- void SqList::merge_pass() : 一趟归并排序
- void SqList::merge_sort() : 归并排序

```
public:
    SqList(const int l)…
    ~SqList()…
    void Display()…
    /*************************************…
    void InsertSort()…
    /*****************************************…
    void BinaryInsertSort()…
    /*****************************************…
    void ShellSort()…
    /*********************************…
    void BubbleSort()…
    /*********************************…
    void QuickSort()…
    /*************************************…
    void SelectSort()…
    /*********************************…
    void HeapSort()…
    /*********************************…
    void MergeSort()…
    void TestSort(ofstream &out,
                  const string &sort_name,
                  void (SqList::*func)())…
    void Test(ofstream &out)…
}; // class SqList

void SqListDebug(const string &in_file_name,
                 const string &out_file_name,
                 ofstream &out_file)…

int main()…
```

- SqList::SqList() : 构造函数，根据传入参数 l 进行空间的动态申请，从 stdin 中依次读入关键字进行顺序表的构建，并利用 std::sort 进行 correct 数组的构建，便于后续的检查操作
- SqList::~SqList() : 析构函数，对 original，r，correct 三个指针进行 delete 操作
- void SqList::Display() : 向 stdout 中输出所有元素
- void SqList::InsertSort() : 给用户调用的插入排序
- void SqList::BinaryInsertSort() : 给用户调用的二分插入排序
- void SqList::ShellSort() : 给用户调用的希尔排序
- void SqList::BubbleSort() : 给用户调用的冒泡排序
- void SqList::QuickSort() : 给用户调用的快速排序
- void SqList::SelectSort() : 给用户调用的选择排序
- void SqList::HeapSort() : 给用户调用的堆排序
- void SqList::MergeSort() : 给用户调用的归并排序
- void SqList::TestSort() : 包装了测试某一排序函数的操作
- void SqList::Test() : 包装了测试全部排序函数的操作
- void SqListDebug() : 测试 class SqList 的所有功能
- int main() : 主函数，SqListDebug()函数入口

开发环境

Hardware: Intel® Core™ i5-10210U, 8GB RAM, 1TB HDD
Software: Windows 10 Pro, C++ language, VSCode with MinGW g++ version 9.2.0, Debug-x86

调试分析

HW9_4_generate_data.cpp

本程序成功按要求生成了 input1.txt, input2.txt, ..., input8.txt 等 8 个测试数据文件, 运行正常, 功能完好。

HW9_4.cpp

本程序依照题目要求进行了 8 组不同数量的测试数据的不同排序, 功能完好, 运行正常。在每次排序结束后, 都会调用 check_accuracy() 函数检查排序的正确性, 在本程序调试过程中, 所有排序结果均为正确。

根据程序运行结果, 整理出数据表格如下。

	插入排序	折半插入排序	希尔排序	冒泡排序	快速排序	选择排序	堆排序	归并排序
时间复杂度	$O(n^2)$	$O(n^2)$	$O(n^{1.3}\sim n^{1.5})$	$O(n^2)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$
稳定性	稳定	稳定	不稳定	稳定	不稳定	不稳定	不稳定	稳定
input1 运行时间(s)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
input2 运行时间(s)	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
input3 运行时间(s)	0.003	0.000	0.000	0.004	0.001	0.002	0.000	0.000
input4 运行时间(s)	0.114	0.102	0.023	0.487	0.001	0.117	0.001	0.002
input5 运行时间(s)	23.681	16.407	2.750	63.243	0.014	9.442	0.022	0.017
input6 运行时间(s)	1225.431	983.927	244.656	4878.841	0.155	947.107	0.278	0.181
input7 运行时间(s)	0.000	0.000	0.000	0.004	0.089	0.104	0.001	0.001
input8 运行时间(s)	0.214	0.192	0.045	0.561	0.080	0.107	0.001	0.001

综上, 本程序设计的数据结构很好地完成了题目对排序顺序表的要求, 功能完好, 使用方便, 并且能够较好地处理非法数据的输入并反馈相关错误提示; 程序界面友好, 具有比较恰当的人性化提醒, 具有一定的鲁棒性。

心得体会

一、 实验总结

在本次上机实验中, 我复习了理论课上学到的顺序表的各种排序, 并封装成了 class SqList, 实现了插入排序, 折半插入排序, 选择排序, 冒泡排序, 希尔排序, 堆排序, 快速排序, 归并排序等 8 种排序, 结果正确, 功能完好。

从以上表格中, 可以分析得出以下结论:

1. 快速排序, 希尔排序, 堆排序, 直接选择排序不是稳定的排序算法, 而基数排序, 冒泡排序, 直接插入排序, 折半插入排序, 归并排序是稳定的排序算法。

2. 传统简单排序在数据量很小的时候也表现不错, 但当数据量增大, 其耗时也增大得十分明显; 冒泡, 插入, 选择三种时间复杂度为 $O(n^2)$ 的排序中, 当数据量大时, 选择排序性能会更好。

3. 数据量小, 或数据元素有某种特殊顺序时, 归并排序和堆排序, 甚至其他时间复杂度为$O(n^2)$的排序算法效率会比快速排序高, 但随着数据元素数量增大, 快速排序体现出的优越性越强。

4. 某些算法可以通过减少元素交换次数缩短运行时间, 但实际比较次数仍不变, 实质上的时间复杂度不变 (如插入排序优化为折半插入排序, 后者的交换次数比前者少, 但两者时

间复杂度都为 $O(n^2)$)。希尔排序使用的 delta 数组对顺序表数据元素的“适配程度”对该算法复杂度影响很大, 但总体上小于等于 $O(n^2)$。

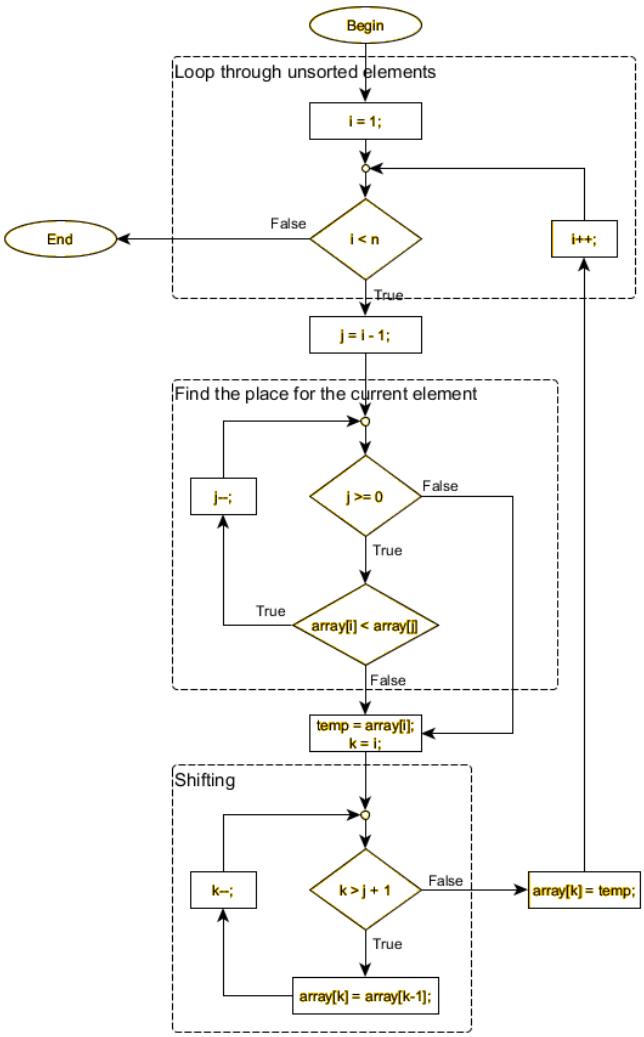
5. 在本程序测试的排序算法中, 平均情况下, 快速排序速度最快。

二、 关键算法性能分析

（1） 插入排序 (Insertion Sort)

时间复杂度: $O(n^2)$

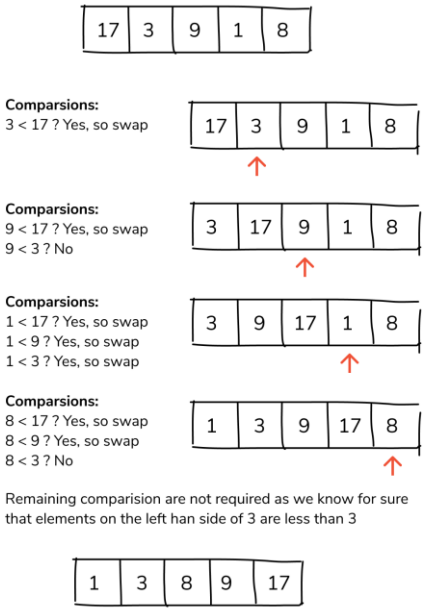
稳定性: 稳定



（2） 希尔排序 (Shell Sort)

时间复杂度: $O(n^{1.3}\sim n^{1.5})$

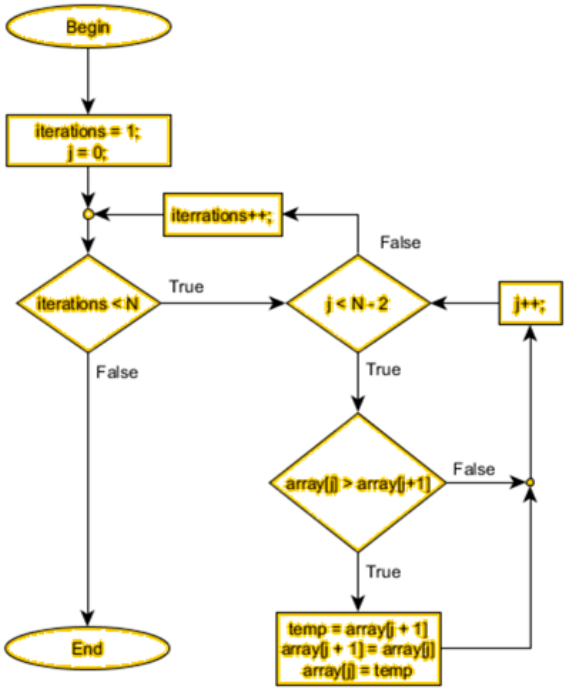
稳定性: 不稳定



（3） 冒泡排序 (Bubble Sort)

时间复杂度: $O(n^2)$

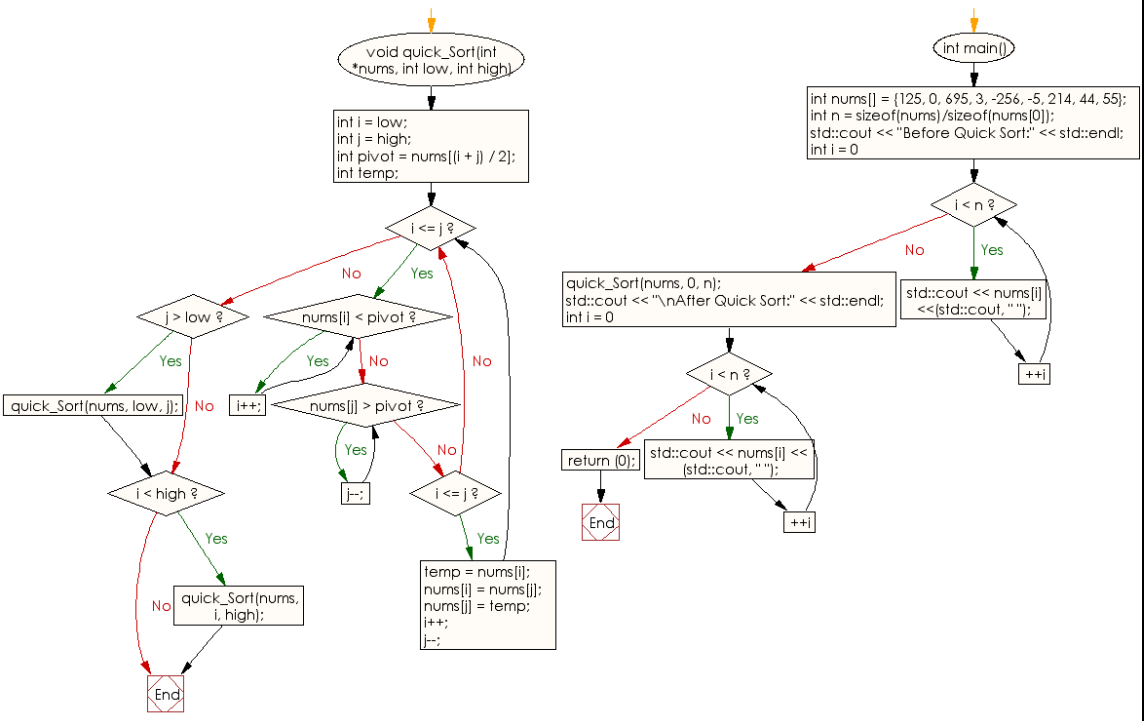
稳定性: 稳定



（4） 快速排序 (Quick Sort)

时间复杂度: $O(n \log n)$

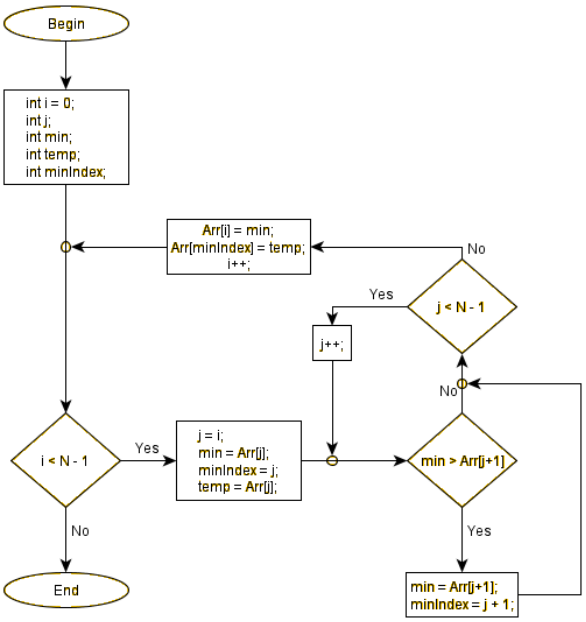
稳定性: 不稳定



（5） 选择排序 (Selection Sort)

时间复杂度: $O(n^2)$

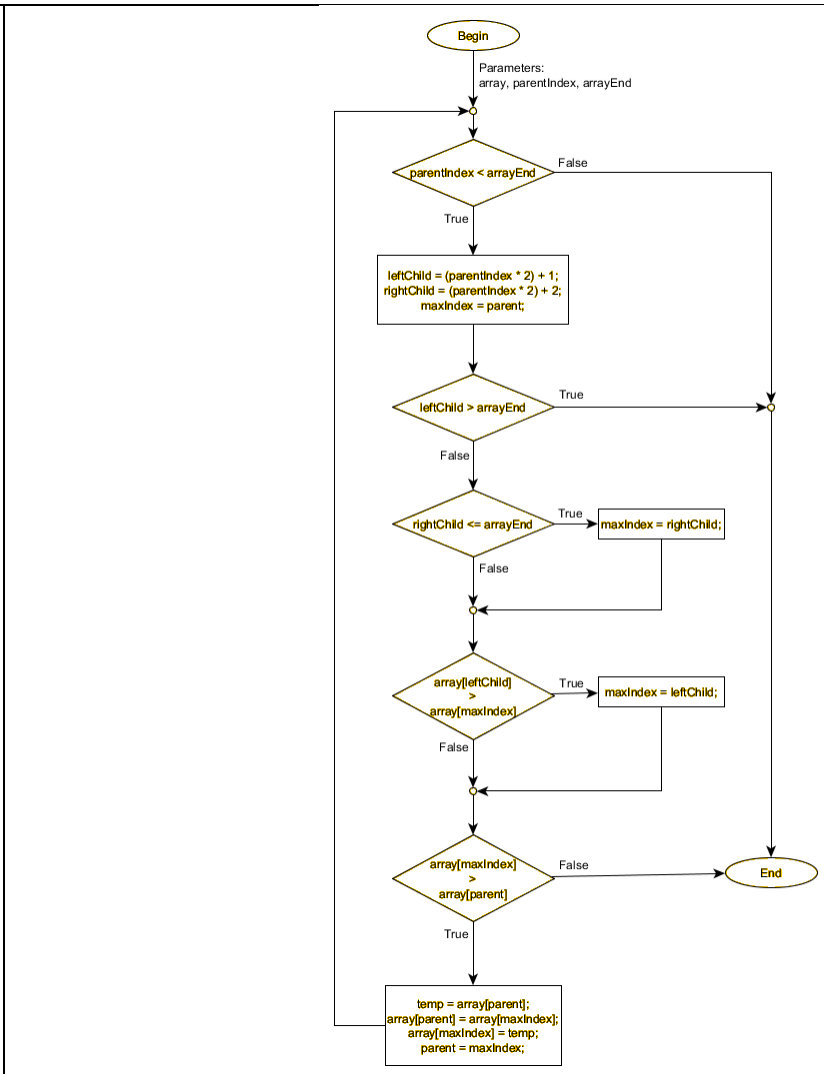
稳定性: 不稳定



（6） 堆排序 (Heap Sort)

时间复杂度: $O(n \log n)$

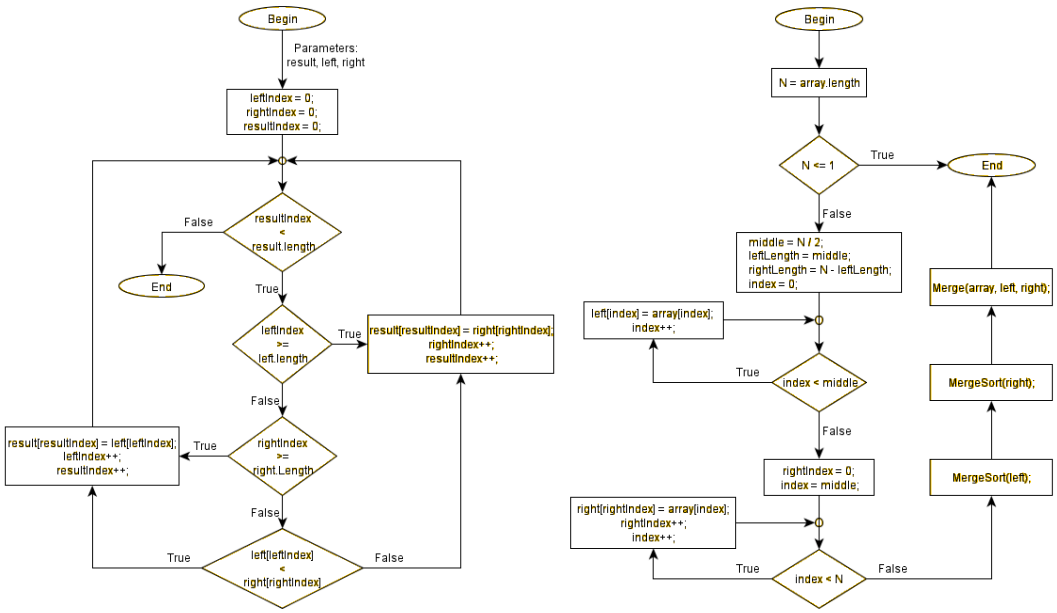
稳定性: 不稳定



（7） 归并排序 (Merge Sort)

时间复杂度: $O(n \log n)$

稳定性: 稳定



源代码	**HW9_4_generate_data.cpp**
	
	1. /**********************************
	2. * @date 12/30/2020
	3. * @author 林日中 (1951112)
	4. * @file HW9_4_generate_data.cpp
	5. **********************************/
	6. #if defined(__GNUC__)
	7. #include <bits/stdc++.h>
	8. #elif defined(_MSC_VER)
	9. #define _CRT_SECURE_NO_WARNINGS
	10. #include <iostream>
	11. #include <fstream>
	12. #include <cstdio>
	13. #include <cstdlib>
	14. #include <cstring>
	15. #include <climits>
	16. #include <ctime>
	17. #include <iomanip>
	18. #include <queue>
	19. #endif
	20.
	21. using namespace std;
	22.
	23. int generate_random_data()
	24. {
	25. int count = 0;
	26. string address;
	27. fstream out;
	28.
	29. {
	30. count = 10;
	31. address = "input1.txt";
	32.
	33. out.open(address, ios::out | ios::binary);
	34. out << count << endl;
	35.
	36. while (count--)
	37. {
	38. out << rand() << ' ';
	39. }
	40. out << endl;
	41.
	42. out.close();

```
    }

    {
        count = 100;
        address = "input2.txt";

        out.open(address, ios::out | ios::binary);
        out << count << endl;

        while (count--)
        {
            out << rand() << ' ';
        }
        out << endl;

        out.close();
    }

    {
        count = 1000;
        address = "input3.txt";

        out.open(address, ios::out | ios::binary);
        out << count << endl;

        while (count--)
        {
            out << rand() << ' ';
        }
        out << endl;

        out.close();
    }

    {
        count = 10000;
        address = "input4.txt";

        out.open(address, ios::out | ios::binary);
        out << count << endl;

        while (count--)
        {
            out << rand() << ' ';
```

```
87.         }
88.         out << endl;
89.
90.         out.close();
91.     }
92.
93.     {
94.         count = 100000;
95.         address = "input5.txt";
96.
97.         out.open(address, ios::out | ios::binary);
98.         out << count << endl;
99.
100.         while (count--)
101.         {
102.             out << rand() << ' ';
103.         }
104.         out << endl;
105.
106.         out.close();
107.     }
108.
109.     {
110.         count = 1000000;
111.         address = "input6.txt";
112.
113.         out.open(address, ios::out | ios::binary);
114.         out << count << endl;
115.
116.         while (count--)
117.         {
118.             out << rand() << ' ';
119.         }
120.         out << endl;
121.
122.         out.close();
123.     }
124.
125.     {
126.         count = 10000;
127.         address = "input7.txt";
128.
129.         out.open(address, ios::out | ios::binary);
130.         out << count << endl;
```

```
131.
132.         int *arr = new int[count];
133.         for (int i = 0; i < count; i++)
134.         {
135.             arr[i] = rand();
136.         }
137.         sort(arr, arr + count);
138.
139.         for (int i = 0; i < count; i++)
140.         {
141.             out << arr[i] << ' ';
142.         }
143.         out << endl;
144.
145.         delete[] arr;
146.         out.close();
147.     }
148.
149.     {
150.         count = 10000;
151.         address = "input8.txt";
152.
153.         out.open(address, ios::out | ios::binary);
154.         out << count << endl;
155.
156.         int *arr = new int[count];
157.         for (int i = 0; i < count; i++)
158.         {
159.             arr[i] = rand();
160.         }
161.         sort(arr, arr + count);
162.
163.         for (int i = count - 1; i >= 0; i--)
164.         {
165.             out << arr[i] << ' ';
166.         }
167.         out << endl;
168.
169.         delete[] arr;
170.         out.close();
171.     }
172.
173.     return 0;
174. }
```

```
175.
176. int main()
177. {
178.     srand((unsigned)time(nullptr));
179.     generate_random_data();
180. }
```

HW9_4.cpp

```
1.  /*************************
2.   * @date   12/30/2020
3.   * @author 林日中 (1951112)
4.   * @file   HW9_4.cpp
5.   *************************/
6.  #if defined(__GNUC__)
7.  #include <bits/stdc++.h>
8.  #elif defined(_MSC_VER)
9.  #define _CRT_SECURE_NO_WARNINGS
10. #include <iostream>
11. #include <fstream>
12. #include <cstdio>
13. #include <cstdlib>
14. #include <cstring>
15. #include <climits>
16. #include <ctime>
17. #include <iomanip>
18. #include <queue>
19. #endif
20.
21. using namespace std;
22.
23. #define OK 1
24. #define ERROR 0
25. #define TRUE 1
26. #define FALSE 0
27. #define LOVERFLOW -1
28. #define MAX_SIZE 1000
29. #define NULL_RECORD nullptr
30. #define NULL_KEY 0
31. #define EQ(a, b) (a == b)
32. #define LT(a, b) (a < b)
33. #define LE(a, b) (a <= b)
34. #define EVL(a, b) (a = b)
35.
```

```
36. typedef int KeyType;
37. typedef struct Element RecordType;
38. 
39. struct Element
40. {
41.     KeyType key;
42.     Element(const KeyType &k = NULL_KEY) : key(k) {}
43.     Element(const Element &e) : key(e.key) {}
44.     Element &operator=(const Element &e)
45.     {
46.         EVL(key, e.key);
47.         return *this;
48.     }
49. }; // struct Element
50. 
51. bool comp(const Element &l, const Element &r)
52. {
53.     return LT(l.key, r.key);
54. }
55. 
56. void swap(Element &l, Element &r)
57. {
58.     Element temp = r;
59.     r = l;
60.     l = temp;
61. }
62. 
63. class SqList
64. {
65. protected:
66.     RecordType *original = nullptr;
67.     RecordType *r = nullptr;
68.     RecordType *correct = nullptr;
69.     int length = 0;
70. 
71.     void recover()
72.     {
73.         copy(original, original + length + 1, r);
74.     }
75. 
76.     void output_to_file(ofstream &out)
77.     {
78.         for (int i = 1; i <= length; i++)
79.         {
```

```
            out << r[i].key << " ";
        }
        out << endl;
    }

    bool check_accuracy()
    {
        RecordType *pr = r + 1, *pc = correct + 1;
        while (pr - r <= length && EQ((pr++)->key, (pc++)->key))
            ;
        return (pr - r == length + 1);
    }

    void shell_insert(const int dk)
    {
        for (int i = dk + 1, j; i <= length; i++)
        {
            if (LT(r[i].key, r[i - dk].key))
            {
                r[0] = r[i];
                 // j = i - dk : equivalently in each group, j = i - 1
                 // j -= dk    : equivalently in each group, j--
                 // j > 0      : if j <= 0, the insert pos is found
                 for (j = i - dk; j > 0 && LT(r[0].key, r[j].key); j -= dk)
                 { // elements move backwards
                     r[j + dk] = r[j];
                 }
                 r[j + dk] = r[0];
             }
         }
    }

    void shell_sort(const int delta[], const int t)
    {
        for (int k = 0; k < t; k++)
        {
            shell_insert(delta[k]);
        }
    }

    int partition(int low, int high)
    {
        r[0] = r[low];
        KeyType pivot_key = r[low].key;
```

```
124.         while (low < high)
125.         {
126.             while (low < high && r[high].key >= pivot_key)
127.             {
128.                 high--;
129.             }
130.             r[low] = r[high];
131.             while (low < high && r[low].key <= pivot_key)
132.             {
133.                 low++;
134.             }
135.             r[high] = r[low];
136.         }
137.         r[low] = r[0];
138.         return low;
139.     }
140.
141.     void qsort(int low, int high)
142.     {
143.         if (low < high)
144.         {
145.             auto pivot_loc = partition(low, high);
146.             qsort(low, pivot_loc - 1);
147.             qsort(pivot_loc + 1, high);
148.         }
149.     }
150.
151.     void heap_adjust(int s, const int m)
152.     {
153.         RecordType rc = r[s]; // save r[s]
154.         // traverse root to leaves in s
155.         // j = 2 * s : j = s->left_child
156.         // j *= 2    : j = j->left_child
157.
158.         int father = s;
159.         int child = father * 2;
160.
161.         for (int j = 2 * s; j <= m; j *= 2)
162.         {
163.             if (j < m && LT(r[j].key, r[j + 1].key))
164.             { // if right child is bigger, choose it (max heap)
165.                 j++;
166.             }
167.             if (!LT(rc.key, r[j].key))
```

```
168.             {
169.                 break;
170.             }
171.             r[s] = r[j]; // root = bigger one between two children
172.             s = j;       // s becomes root of the child
173.         }
174.         r[s] = rc;
175.     }
176.
177.     // merge [L..M], [M+1..H] the two ordered sequences into one
178.     void merge(RecordType *SR, RecordType *&TR,
179.                const int L, const int M, const int H)
180.     {
181.         int i = L, j = M + 1, k = L;
182.         while (i <= M && j <= H) // while two sequences are both not traversed c
    ompletely
183.         {
184.             if (LT(SR[i].key, SR[j].key))
185.             {
186.                 TR[k++] = SR[i++]; // smaller one into TR, move forward
187.             }
188.             else
189.             {
190.                 TR[k++] = SR[j++]; // smaller one into TR, move forward
191.             }
192.         }
193.
194.         while (i <= M) // merge remain of the sequence into result
195.         {
196.             TR[k++] = SR[i++];
197.         }
198.         while (j <= H) // merge remain of the sequence into result
199.         {
200.             TR[k++] = SR[j++];
201.         }
202.     }
203.
204.     void merge_pass(RecordType *SR, RecordType *&TR,
205.                     const int len, const int n)
206.     {
207.         int i;
208.
209.         // loop when lengths of two subsequences are both n
210.         for (i = 1; i + 2 * len - 1 <= n; i += 2 * len)
```

```
211.         {
212.             merge(SR, TR, i, i + len - 1, i + 2 * len - 1);
213.         }
214.
215.         if (i + len - 1 < n) // two subsequences, length of latter one < len
216.         {
217.             merge(SR, TR, i, i + len - 1, n);
218.         }
219.         else // only one subsequence
220.         {
221.             while (i <= n)
222.             {
223.                 TR[i] = SR[i];
224.                 i++;
225.             }
226.         }
227.     }
228.
229.     void merge_sort(RecordType *R, const int n)
230.     {
231.         int len = 1;
232.         RecordType *R1 = new RecordType[n + 1];
233.
234.         while (len < n)
235.         {
236.             merge_pass(R, R1, len, n);
237.             len *= 2;
238.             merge_pass(R1, R, len, n);
239.             len *= 2;
240.         }
241.
242.         delete[] R1;
243.     }
244.
245. public:
246.     SqList(const int l)
247.         : original(new RecordType[l + 1]),
248.           r(new RecordType[l + 1]),
249.           correct(new RecordType[l + 1]),
250.           length(l)
251.     {
252.         for (int i = 1; i <= l; i++)
253.         {
254.             cin >> original[i].key;
```

```
        }
        copy(original, original + length + 1, correct);
        sort(correct + 1, correct + length + 1, comp);
        recover();
    }

    ~SqList()
    {
        delete[] original;
        delete[] r;
        delete[] correct;
    }

    void Display()
    {
        for (int i = 1; i <= length; i++)
        {
            cout << r[i].key << " ";
        }
        cout << "\n";
    }

    /**********************************
     * Sort Name        : Insertion Sort
     * Space Complexity : O(1)
     * Time Complexity  : O(n^2)
     **********************************/
    void InsertSort()
    {
        for (int i = 2, j; i <= length; i++)
        {
            if (LT(r[i].key, r[i - 1].key))
            {
                r[0] = r[i]; // set a sentinel, avoid overflow
                r[i] = r[i - 1];
                for (j = i - 2; LT(r[0].key, r[j].key); j--)
                {
                    r[j + 1] = r[j];
                }
                r[j + 1] = r[0]; // r[i] is already covered
            }
        }
    }

```

```
299.     /*****************************************
300.      * Sort Name         : Binary Insertion Sort
301.      * Space Complexity : O(1)
302.      * Time Complexity   : O(n^2)
303.      *****************************************/
304.     void BinaryInsertSort()
305.     {
306.         for (int i = 2; i <= length; i++)
307.         {
308.             r[0] = r[i];
309.             int low = 1;
310.             int high = i - 1;
311.             while (low <= high)
312.             {
313.                 int mid = (low + high) >> 1;
314.                 if (LT(r[0].key, r[mid].key))
315.                 {
316.                     high = mid - 1;
317.                 }
318.                 else
319.                 {
320.                     low = mid + 1;
321.                 }
322.             }
323.             // until now, we have found the insert pos: [high + 1] or [low]
324.
325.             for (int j = i - 1; j >= high + 1; j--)
326.             { // elements move backwards
327.                 r[j + 1] = r[j];
328.             }
329.             r[high + 1] = r[0]; // r[i] has been covered
330.         }
331.     }
332.
333.     /*****************************************
334.      * Sort Name         : Shell Sort
335.      * Space Complexity : O(1)
336.      * Time Complexity   : O(n^1.3 ~ n^1.5)
337.      * Parameters        :
338.      *       - delta[]   : increment sequence,
339.      *                     last element must be 1
340.      *       - t         : length of delta
341.      *****************************************/
342.     void ShellSort()
```

```
343.     {
344.         int delta[] = {5, 3, 1};
345.         shell_sort(delta, sizeof(delta) / sizeof(int));
346.     }
347.
348.     /*********************************
349.      * Sort Name         : Bubble Sort
350.      * Space Complexity : O(1)
351.      * Time Complexity   : O(n^2)
352.      *********************************/
353.     void BubbleSort()
354.     {
355.         int change_flag = 1;
356.         for (int i = length; change_flag && i >= 2; i--)
357.         {
358.             change_flag = 0;
359.             for (int j = 1; j < i; j++)
360.             {
361.                 if (LT(r[j + 1].key, r[j].key))
362.                 {
363.                     swap(r[j + 1], r[j]);
364.                     change_flag = 1;
365.                 }
366.             }
367.         }
368.     }
369.
370.     /*********************************
371.      * Sort Name         : Quick Sort
372.      * Space Complexity : O(1)
373.      * Time Complexity   : O(nlog_2n)
374.      *********************************/
375.     void QuickSort()
376.     {
377.         qsort(1, length);
378.     }
379.
380.     /*********************************
381.      * Sort Name         : Selection Sort
382.      * Space Complexity : O(1)
383.      * Time Complexity   : O(n^2)
384.      *********************************/
385.     void SelectSort()
386.     {
```

```
387.         for (int i = 1; i < length; i++)
388.         {
389.             int m = i;
390.             for (int j = i + 1; j <= length; j++)
391.             { // find minimum pos in [i..n]
392.                 if (LT(r[j].key, r[m].key))
393.                 {
394.                     m = j;
395.                 }
396.             }
397.             if (m != i)
398.             {
399.                 swap(r[i], r[m]);
400.             }
401.         }
402.     }
403.
404.     /*********************************
405.      * Sort Name         : Heap Sort
406.      * Space Complexity : O(1)
407.      * Time Complexity  : O(nlog2_n)
408.      *********************************/
409.     void HeapSort()
410.     {
411.         for (int i = length / 2; i > 0; i--)
412.         { // loop from last non-leaf node to root, we get initial heap
413.             heap_adjust(i, length);
414.         }
415.         for (int i = length; i > 1; i--) // i == 1 is root
416.         {
417.             swap(r[1], r[i]);
418.             heap_adjust(1, i - 1); // first (i - 1) elements adjusted to heap
419.         }
420.     }
421.
422.     /*********************************
423.      * Sort Name         : Merge Sort
424.      * Space Complexity : O(n)
425.      * Time Complexity  : O(nlog2_n)
426.      *********************************/
427.     void MergeSort()
428.     {
429.         merge_sort(r, length);
430.     }
```

```
431.
432.     void TestSort(ofstream &out,
433.                   const string &sort_name,
434.                   void (SqList::*func)())
435.     {
436.         clock_t begin_time = clock();
437.         (this->*func)();
438.         clock_t end_time = clock();
439.         out << "Sort name : " << sort_name << endl
440.             << "Time cost : " << fixed << setprecision(3) << (double)(end_time -
    begin_time) / CLOCKS_PER_SEC << "s" << endl
441.             << "Accuracy  : " << (check_accuracy() ? "Y" : "N") << endl
442.             << "Status    : Complete" << endl
443.             << endl;
444.         recover();
445.     }
446.
447.     void Test(ofstream &out)
448.     {
449.         TestSort(out, "Insertion Sort", &SqList::InsertSort);
450.         TestSort(out, "Binary Insertion Sort", &SqList::BinaryInsertSort);
451.         TestSort(out, "Shell Sort", &SqList::ShellSort);
452.         TestSort(out, "Bubble Sort", &SqList::BubbleSort);
453.         TestSort(out, "Quick Sort", &SqList::QuickSort);
454.         TestSort(out, "Selection Sort", &SqList::SelectSort);
455.         TestSort(out, "Heap Sort", &SqList::HeapSort);
456.         TestSort(out, "Merge Sort", &SqList::MergeSort);
457.     }
458.
459. }; // class SqList
460.
461. void SqListDebug(const string &in_file_name,
462.                  const string &out_file_name,
463.                  ofstream &out_file)
464. {
465.     int count_of_elements;
466.
467.     if (out_file.is_open())
468.     {
469.         out_file.close();
470.     }
471.
472.     freopen(in_file_name.c_str(), "rb", stdin);
473.     out_file.open(out_file_name, ios::out | ios::binary);
```

```
474.     cin >> count_of_elements;
475.     SqList sqlist(count_of_elements);
476.
477.     out_file << "*************" << endl
478.              << "PROGRAM BEGIN" << endl
479.              << "*************" << endl
480.              << endl;
481.
482.     out_file << "INPUTFILE : " << in_file_name << endl
483.              << "OUTPUTFILE: " << out_file_name << endl
484.              << "COUNT     : " << count_of_elements << endl
485.              << endl;
486.
487.     sqlist.Test(out_file);
488.
489.     out_file << "*****************" << endl
490.              << "PROGRAM COMPLETE" << endl
491.              << "*****************" << endl;
492.
493.     out_file.close();
494. }
495.
496. int main()
497. {
498.     ofstream out;
499.     SqListDebug("input1.txt", "output1.txt", out);
500.     SqListDebug("input2.txt", "output2.txt", out);
501.     SqListDebug("input3.txt", "output3.txt", out);
502.     SqListDebug("input4.txt", "output4.txt", out);
503.     SqListDebug("input5.txt", "output5.txt", out);
504.     SqListDebug("input6.txt", "output6.txt", out);
505.     SqListDebug("input7.txt", "output7.txt", out);
506.     SqListDebug("input8.txt", "output8.txt", out);
507.
508.     return 0;
509. }
```