

数据结构与算法 课程实验报告

学号：202000130198	姓名：隋春雨	班级：20.4																								
实验题目：排序算法																										
实验学时：2	实验日期：2021-10-04																									
实验目的： 掌握各种简单排序算法。如：冒泡、选择、插入等排序，并掌握及时终止的排序。																										
软件开发环境： CLION2020																										
1. 实验内容 ①题目描述： 用任意一种排序方式给出 n 个整数按升序排序后的结果，满足以下要求： <ol style="list-style-type: none"> 不得使用与实验相关的 STL； 需使用类模版 (template<class T>)； 需定义排序类，封装各排序方法； 排序数据需使用动态数组存储； 排序类需提供以下操作：名次排序、及时终止的选择排序、及时终止的冒泡排序、插入排序。 输入输出格式： 输入：输入的第一行是一个整数 n ($1 \leq n \leq 1000$)，表示需排序的数的个数。接下来一行是 n 个整数，数的范围是 0 到 1000，每两个相邻数据间用一个空格分隔。 输出：一行排好序的序列。																										
2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法） (1) 选择排序思路：选择排序的思路比较容易理解，对于一个数组，我们每一次选出它的一个子序列的最大值或者最小值放在它的最后边/最前面，进行 n 次就能排序好。例子如下： <div style="text-align: center; margin-top: 20px;"> <table border="0"> <tr> <td>4</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>3</td> <td>4</td> <td>2</td> <td>2</td> </tr> <tr> <td>1</td> <td>3</td> <td>4</td> <td>3</td> </tr> <tr> <td>2</td> <td>2</td> <td>3</td> <td>4</td> </tr> <tr> <td>5</td> <td>5</td> <td>5</td> <td>5</td> </tr> <tr> <td>初始状态</td> <td>第一遍排序后</td> <td>第二遍排序后</td> <td>第三遍排序后</td> </tr> </table> </div>			4	1	1	1	3	4	2	2	1	3	4	3	2	2	3	4	5	5	5	5	初始状态	第一遍排序后	第二遍排序后	第三遍排序后
4	1	1	1																							
3	4	2	2																							
1	3	4	3																							
2	2	3	4																							
5	5	5	5																							
初始状态	第一遍排序后	第二遍排序后	第三遍排序后																							

其中，我们可以进行及时终止的选择排序，用一个 bool 变量记录是否应该终止，如果扫过的子序列是一个单调不减的序列的话，就应该及时终止。

(2) 冒泡排序：冒泡排序的思路其实与选择排序差不多，每一次都把最大值冒到最后边/最前边。最终得到有序序列。

(3) 冒泡排序/选择排序时间复杂度分析：

可以看出，冒泡排序与选择排序都是一个从 N 到 1 的一个序列，故时间复杂度均为 $O(N^2)$

排序类别	排序方法	时间复杂度			空间复杂度	稳定性	复杂性
		平均情况	最坏情况	最好情况			
交换排序	冒泡排序	$O(N^2)$	$O(N^2)$	$O(N)$	$O(1)$	稳定	简单

考虑到 bool 值判定，如果冒泡排序/选择排序是一个接近有序的序列，那么排序时间复杂度会很好，能够达到 $O(N)$ ，但是快排/归并就不行，都是 $O(N\log N)$ 。

(4) 插入排序：插入排序的思想其实与堆的插入差不多，都是在面对一个已经有序/已经是堆的情况下，将新的值插入到合适的位置。值得注意的点就是要确定循环终止的条件，要么是已经找到合理的插入位置，要么是扫描完全部序列，发现都没找到，那么我们就应该插在最前面。经过思考，我们发现这两种情况其实就是一种情况，代码如下：

```
for (; j >= 0 && pointer[j] > pointer[i]; j--); //j+1是我们即将插入的值的位
```

(5) 按照名次排序：因为一个数字在一个固定的数组中的大小关系是一定的，故给出一个数组，我们一定可以给出它排序后的情况。因此我们额外开辟一个 rank 数组，记录它的大小关系，如果它前边的数字 \geq 它，那么它 rank 数组对应的位置便+1，否则那个数字对应的位置的数组元素便+1。得到一个全部的 rank 数组后，我们使用一个 for 循环进行调整位置，如果没在对应的位置，那么我们便调用 swap，具体如下：


```
for (int i = 0; i < size; i++)
{
    while (rank[i] != i) {
        std::swap(pointer[i], pointer[rank[i]]);
        std::swap(&rank[i], &rank[rank[i]]);
    }
}
```

3. 测试结果（测试输入，测试输出）

(1) 用系统时间为种子生成随机数 1000 个

```
srand((unsigned)time(NULL));
int n;
cin >> n;
Sort_class<int>s(n);
for (int i = 0; i < n; i++)
{
    s[i]=rand();
}
bool flag = 1;
s.sort_by_selection();//分别调用，发现都排序成功
for (int i = 0; i < n; i++) {
    cout << s[i] << endl;
}
for (int i = 1; i < n; i++) {
    if (s[i] < s[i - 1]) {
        flag = false;
        break;
    }
}
if (flag) {
    cout << "数组是有序的" << endl;
}
```

结果：

 Microsoft Visual Studio 调试控制台

```
31515
31591
31616
31653
31667
31676
31713
31755
31756
31817
31834
31836
31876
31924
31925
31951
32017
32060
32062
32236
32255
32263
32276
32331
32333
32376
32392
32407
32407
32478
32534
32543
32572
32602
32604
32610
32622
32629
32635
32674
32685
32750
数组是有序的
```

Contest	数据结构 (3/4班) 实验2 排序算法 >
Problem Code	A >
Problem Title	排序算法
Submission ID	33f6f5e7d40642e
Create Time	2021-10-07 19:04:21
Judge Time	2021-10-07 19:04:21
Username	202000130198
Judge Result	✓ Accepted
Score	100
Judge Template	C++11
Total Time	1 ms
Total Memory	3472 KiB

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

(1) 冒泡排序最好的排序效果与最坏的排序效果分别是多少？

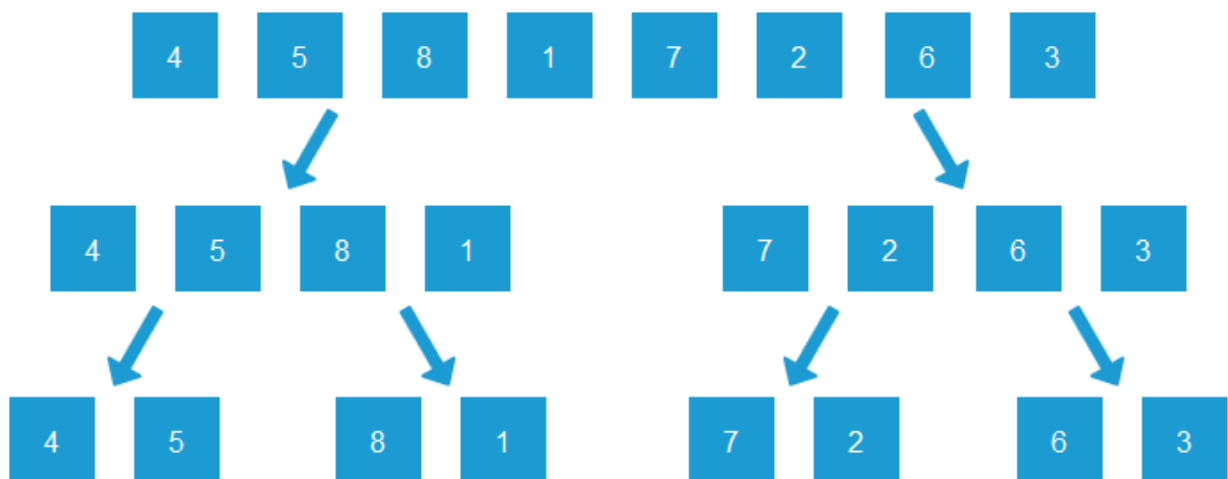
$$C_{\max} = N(N-1)/2 = O(N^2)$$

$$M_{\max} = 3N(N-1)/2 = O(N^2)$$

冒泡排序的最坏时间复杂度为 $O(N^2)$ 。

(2) 除了实验要求的冒泡、选择、插入排序时间复杂度都是 $O(N^2)$ ，有没有别的更通用的排序算法了？

答：有，比如快排和归并。归并排序的原理就是：我们先排左边子列，使其有序，再排右边子列，使其有序。最后合并到一起。例子如下：



其中，递归终止条件为：只有一个元素。一个元素必然有序。

(3) 插入排序为什么不能用及时终止？

答：因为插入排序有后效性，就像中缀表达式不能及时马上求值一样，需要转化为后缀表达式才能读到一个数字就马上求值。冒泡排序与选择排序都是因为无后效性才能马上判断是否应该终止。

(4) 递归排序除了较好的时间复杂度之外有什么作用？我们可以看下面的这个求逆序对的例子

输入格式

第一行，一个数 n ，表示序列中有 n 个数。

第二行 n 个数，表示给定的序列。序列中每个数字不超过 10^9 。

输出格式

输出序列中逆序对的数目。

输入输出样例

输入 #1

复制

输出 #1

复制

```
6
5 4 2 6 3 1
```

```
11
```

按照平常的思路，我们肯定是两个 for 循环搜索一遍，时间复杂度为 $O(N^2)$ ，很慢，如果我们结合归并排序的思想，在排序的过程中顺便求出逆序对的数量，可以达到 $O(N\log N)$ 的时间复杂度。很显然，如果我们要求逆序对，必须要求两个子列降序。考虑到极端情况给出了一个严格单调递减的序列，我们的结果可能达到 $1e^{10}$ 的级别，我们需要开一个 long long 变量进行存储。

最终代码：

```
1. #include <bits/stdc++.h>
2. using namespace std;
```

```
3.  const int N=5e5+50;
4.  int a[N];
5.  int temp[N];
6.  long long res=0;
7.  void merge_sort(int l,int r){
8.      if(l==r){//递归终止条件
9.          return ;
10.     }
11.
12.     int mid=l+r>>1;//移位
13.     merge_sort(l,mid);//归并左半部分
14.     merge_sort(mid+1,r);//归并右半部分
15.     int i=l;
16.     int j=mid+1;//双指针
17.     int pos=l;
18.     while(i<=mid&& j<=r){
19.         if(a[i]>a[j]){
20.             res+=r-j+1;
21.             temp[pos++]=a[i++];
22.         }
23.         else{
24.             temp[pos++]=a[j++];
25.         }
26.     }
27.     while(i<=mid)
28.     {
29.         temp[pos++]=a[i++];
30.     }
31.     while(j<=r)
32.     {
33.         temp[pos++]=a[j++];
34.     }
35.     for(int i=l;i<=r;i++)
36.     {
37.         a[i]=temp[i];//回写
38.     }
39. }
40. int main()
41. {
42.     int n;
43.     scanf("%d",&n);
44.     for(int i=0;i<n;i++){
45.         scanf("%d",&a[i]);
46.     }
47.     merge_sort(0,n-1);//调用归并排序
48.     printf("%lld",res);
```

```
49.     return 0;
50. }
```

最终结果：

测试点信息

#1 AC 3ms/2.49MB	#2 AC 3ms/2.61MB	#3 AC 3ms/2.56MB	#4 AC 2ms/2.53MB	#5 AC 3ms/2.61MB	#6 AC 10ms/2.75MB	#7 AC 11ms/2.73MB
#8 AC 10ms/2.61MB	#9 AC 11ms/2.73MB	#10 AC 10ms/2.63MB	#11 AC 112ms/4.25MB	#12 AC 111ms/4.38MB	#13 AC 112ms/4.28MB	#14 AC 112ms/4.36MB
#15 AC 112ms/4.35MB	#16 AC 112ms/4.25MB	#17 AC 112ms/4.28MB	#18 AC 111ms/4.37MB	#19 AC 112ms/4.36MB	#20 AC 111ms/4.21MB	

(5) 快速排序的思路是怎样的？

答：选择一个基准数，通过一趟排序将要排序的数据分割成独立的两部分；其中一个序列的所有数据都比另外一个序列的数据要小。然后，再按此方法对这两部分数据分别递归进行快速排序，终止条件为只有一个元素，以此达到整个数据变成有序序列。

(6) 快速排序时间复杂度是多少？

答：平均时间复杂度是 $O(N\log N)$ 。快速排序是采用分治法进行遍历的，不妨将其看为树的数据结构，它需要遍历的次数就是二叉树的层数，它的深度上界是 $\lg(N+1)$ 。因此，快速排序的遍历次数最少是 $\log(N+1)$ 次。

(7) 基数排序的稳定性是否为必须的？

答：是必须的，因为我们在每次从低位到高位排序的时候，首先需要保证之前的结果不被破坏。即为，如果高位相同，那么按照低位排序的结果来排序。

(8) 基数排序的优点和缺点？

优点：时间复杂度优秀， $O(n)$

缺点：需要额外的数组空间，同样也是 $O(n)$ ，如果在我们排序的数组很长的时候，这个消耗是很大的。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
1.  #include<iostream>
2.  using namespace std;
3.
4.  template<class T>
```



```

5.  class Sort_class {
6.      int size;//数组元素的数量
7.      int capacity;//数组的最大容量
8.      T* pointer = nullptr;
9.  public:
10.     Sort_class():capacity(1), pointer(new T[capacity]) {}
11.     ~Sort_class()
12.     {
13.         delete[]pointer;
14.         pointer = nullptr;
15.     }
16.     Sort_class(int to_size):pointer(new T[to_size]), size(to_size), capacity(to_size==0?1:2*to_size) {}//单独考虑为 0
    的情况
17.     T& operator[](int id)
18.     {
19.         if (id < 0 || id >= size) { //异常处理
20.             throw "error";
21.         }
22.         return pointer[id];
23.     }
24.     void push_back(T val)
25.     {
26.         if (size == capacity) //如果不能再插入了就 capacity 加倍
27.         {
28.             T* new_p = new T[2 * capacity];
29.             for (int i = 0; i < size; i++)
30.             {
31.                 new_p[i] = pointer[i];
32.             }
33.             T* temp = pointer;
34.             pointer = new_p;
35.             delete[]temp;
36.             temp = nullptr;
37.             capacity *= 2;
38.         }
39.         pointer[size++] = val; //读入新的值
40.     }
41.
42.     void sort_by_rank()
43.     {
44.         int* rank = new int[size](); //初始化为 0
45.         for (int i = 0; i < size; i++)
46.             { //可以证明这是一个从 0-n-1 的一个连续数列，否则我们需要对其进行离散化
47.                 for (int j = i + 1; j < size; j++)
48.                 {
49.                     pointer[i] <= pointer[j] ? rank[j]++ : rank[i]++;

```

```

50.     }
51. }
52. for (int i = 0; i < size; i++)
53. {
54.     while (rank[i] != i) {
55.         std::swap(pointer[i], pointer[rank[i]]); //考虑到未来有可能加入 swap 函数，这里使用 std::swap
56.         std::swap(rank[i], rank[rank[i]]);
57.     }
58. }
59. delete[] rank; //删除动态分配的内存
60. }
61.
62. void sort_by_selection() //及时终止的插入排序
63. {
64.     bool sorted = false;
65.     for (int i = size - 1; !sorted && i >= 1; i--) //如果还没有有序就继续进行
66.     {
67.         sorted = true;
68.         int index_of_max = 0;
69.         for (int j = 0; j <= i; j++)
70.         {
71.
72.             pointer[j] >= pointer[index_of_max] ? index_of_max = j : sorted = false; //简洁书写
73.
74.         }
75.         std::swap(pointer[i], pointer[index_of_max]);
76.     }
77. }
78.
79. void sort_by_bubble() //及时终止的冒泡排序
80. {
81.     bool sorted = false;
82.     for (int i = size - 1; !sorted && i >= 1; i--) //如果没有排序成功就继续处理
83.     {
84.         sorted = true;
85.         for (int j = 0; j < i; j++)
86.         {
87.             if (pointer[j] > pointer[j + 1])
88.             {
89.                 std::swap(pointer[j], pointer[j + 1]);
90.                 sorted = false;
91.             }
92.             //发现每一个都小于等于后一个就终止
93.         }
94.     }
95. }

```

```
96.
97. void sort_by_insert()
98. {
99.     //插入排序有后效性，不能用及时终止
100.    //插入排序有后效性
101.    for (int i = 1; i < size; i++)
102.    {
103.        int t = pointer[i];
104.        int j = i - 1;
105.        for (; j >= 0 && pointer[j] > pointer[i]; j--); //j+1 是我们即将插入的值的位罝
106.        for (int k = i; k > j; k--)
107.        {
108.            pointer[k] = pointer[k - 1]; //必须从后往前遍历，否则会被覆盖
109.        }
110.        pointer[j] = t;
111.    }
112. }
113.
114. friend ostream& operator<<(ostream& os, const Sort_class<T>& s)
115. {
116.     for (int i = 0; i < s.size; i++)
117.     {
118.         os << s.pointer[i] << " ";
119.     }
120.     return os;
121. }
122.
123. friend istream& operator>>(istream& is, Sort_class<T>& s)
124. {
125.     for (int i = 0; i < s.size; i++)
126.     {
127.         is >> s.pointer[i];
128.     }
129.     return is;
130. }
131.
132.
133. };
134.
135. int main()
136. {
137.     int n;
138.     cin >> n;
139.     Sort_class<int>s(n);
140.     for (int i = 0; i < n; i++)
141.     {
```

```
142.     cin >> s[i];
143. }
144. // s.sort_by_bubble();
145. //s.sort_by_insert();
146. // s.sort_by_rank();
147. s.sort_by_selection();//分别调用，发现都排序成功
148. cout << s;
149. return 0;
150. }
```