

## 《数据结构》课程实践报告

院、系	计算机学院	年级专业	21 计科	姓名	方浩楠	学号	2127405048
实验布置日期	2022.11.1		提交日期	2022.11.10		成绩	

### 课程实践实验 6：排序算法的实现及性能测试及比较

#### 一、问题描述及要求

1、实现顺序表下的插入排序、选择排序、归并排序、堆排序和快速排序算法并进行正确性测试。

2、通过实验比较顺序表下各排序算法性能，注意不同算法的比较应基于同样的初始数据。

(1) 排序算法包括：插入排序，选择排序，归并排序，快速排序，堆排序等；

(2) 测试指标包括：排序时的绝对运行时间、总的关键字比较次数和记录的移动次数；

(3) 针对初始为正序、逆序、随机序三种不同情况分别进行比较；

(4) 针对长度不同的原始数据进行排序比较。如  $n=100, 1000, 3000, 6000, 10000$  等，当  $n$  很大时，观察会发生什么情况？

(5) 设计表格，记录实验数据并分析实验与理论是否一致。

#### 二、概要设计

(1)内容理解:

运用不同的排序方式进行排序，并且比较不同排序方式的效率

## (2)功能列表:

```
1  Sort(int *a, int n);
2  ~Sort() = default;
3  void InsertionSort();           //插入排序
4  void ShellSort();              //希尔排序
5  void selectSort();             //选择排序
6  void BubbleSort();             //冒泡排序
7  void QuickSort(int first, int last); //快速排序
8  void HeapSort();               //堆排序
9  void MergeSort(int p, int r);  //归并排序(递归)
10 void MergeSort2();             //归并排序(非递归)
11 void Print();                  //输出数组
```

## (3) 界面设计

```
D:\Programming\C-CPP\Csteaching\experiment6-Sort\cmake-build-debug\experiment6_Sort.exe
1.Insertion Sort
2.Shell Sort
3.Select Sort
4.Bubble Sort
5.Quick Sort
6.Heap Sort
7.Merge Sort
选择一种排序方式
1
1.正序 2.逆序 3.随机
选择数据的类型
1
1.1000 2.3000 3.10000
选择数据的个数
3
文件名为:      D:\Programming\C-CPP\Csteaching\experiment6-Sort\data\negative-order\10000.txt
数据个数为:    10000
排序方式为:    InsertionSort
运行时间为:    88

进程已结束,退出代码0
```

## (4) 设计思路:

采用的类:class Sort

类内的函数:

```

1  class Sort
2  {
3  public:
4      Sort(int *a, int n);           //构造函数
5      ~Sort() = default;           //析构函数
6      void InsertionSort();         //插入排序
7      void ShellSort();             //希尔排序
8      void SelectSort();            //选择排序
9      void BubbleSort();            //冒泡排序
10     void QuickSort(int first, int last); //快速排序
11     void HeapSort();               //堆排序
12     void MergeSort(int p, int r);   //归并排序(递归)
13     void MergeSort2();              //归并排序(非递归)
14     void Print();                  //输出数组
15 private:
16     //快速排序的划分
17     int Partition(int first, int last);
18     //维护最大堆
19     void MaxHeapify(int i);
20     //构造最大堆
21     void BuildMaxHeap();
22     //堆的左节点
23     int Left(int i)
24     { return 2 * i + 1; }
25     //堆的右节点
26     int Right(int i)
27     { return 2 * i; }
28     //堆的父节点
29     int Parent(int i)
30     { return (i - 1) / 2; }
31     //归并排序
32     void Merge(int p, int q, int r);
33     void MergePass(int h);
34
35     int *data;    //int* data中存储数据
36     int length;   //length存储数组的长度

```

快速排序需要调用 private 中的 partition 过程，用于对一个数组进行划分。

归并排序需要调用 private 中的 merge 过程，用于两个有序数组的合并。

堆排序是需要调用 private 中的 BuildMaxHeap 过程构建最大堆，需要调用 MaxHeapify(int i) 来维护数组中第 i 号元素的最大堆的性质、

## 三、详细设计

### (1) 获取程序运行时间

需要#include<ctime>, 调用 ctime 中的 clock()函数, 在排序之前和之后分别使用两个 clock\_t 类型的变量, 变量名为 begin 和 end, 需要知道程序运行时间就将 end-begin

### (2) QuickSort(int first,int end)

快速排序的关键点在于对数组的划分, 因此 partition 过程是 QuickSort 的关键。在每次调用 QuickSort 时, 都需要调用 private 中的 Partition()函数来对数组进行一次划分。函数的两个形参分别是需要排序的部分数组的第一号元素和最后一号元素的位置

### (3) MergeSort()

MergeSort 的关键在于对于两个已经排序完成的数组的合并。合并时可以使用哨兵，也可以不适用。本函数中，private 中的 Merge()函数采用了不使用哨兵的方式来对两个排序好了的数组进行合并.Merge(int p,int q,int r)中的三个形参分别为第一个数组的头，第一个数组的尾部，第二个数组的尾部。

### (4) HeapSort()

在使用 HeapSort 之前，首先需要将数组构建成一个最大堆，构建最大堆由 private 中的 BuildMaxHeap()实现。BuildMaxHeap()中调用的 MaxHeapify(int i)是维护最大堆,方式为假定根节点为 Left(i)和 Right(i)的二叉树都是最大堆，此时让 data[i]逐步下沉，从而使得下标为 i 的根节点的字数仍然遵守最大堆的性质。堆排序时将数组的第零号元素(根据最大堆的性质，数组的第零号元素一定是整个数组最大的元素)与最后一号元素交换，使数组的长度减一，然后对数组的第零号元素调用 MaxHeapify()。

## 四、实验结果

### (1) 测试的数据:

data/positive-order/1000.txt 中存储着 1000 个递增的数字



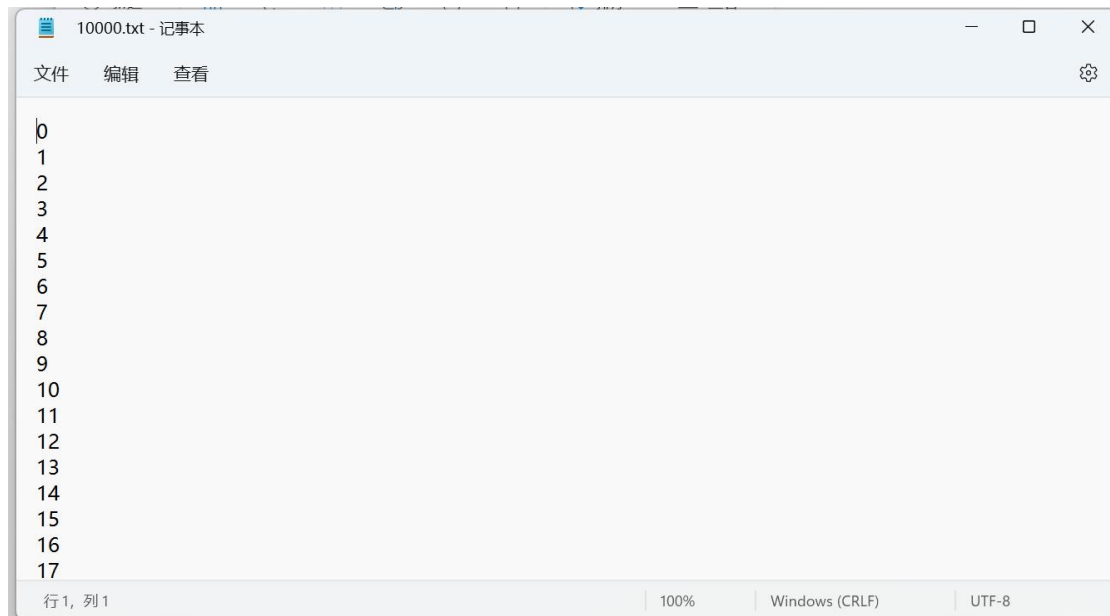
```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

data/positive-order/3000.txt 中存储着 3000 个递增的数字



```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

data/positive-order/10000.txt 中存储着 10000 个递增的数字



```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

data/negative-order/1000.txt 中存储着 1000 个递减的数字

```
1000
999
998
997
996
995
994
993
992
991
990
989
988
987
986
985
984
983
```

data/negative-order/3000.txt 中存储着 3000 个递减的数字

```
3000
2999
2998
2997
2996
2995
2994
2993
2992
2991
2990
2989
2988
2987
2986
2985
2984
2983
```

data/negative-order/10000.txt 中存储着 10000 个递减的数字



```
10000
9999
9998
9997
9996
9995
9994
9993
9992
9991
9990
9989
9988
9987
9986
9985
9984
9983
```

data/ramdom-order/1000.txt 中存储着 1000 个随机排序的数字



```
917
210
443
198
713
768
840
568
867
890
383
671
550
693
227
332
855
683
```

data/ramdom-order/3000.txt 中存储着 3000 个随机排序的数字



3000.txt - 记事本

文件 编辑 查看

```
2729
2018
513
1505
1289
142
2078
1615
133
2917
2626
697
895
1546
1198
1278
2643
804
```

行 1, 列 1 | 100% | Windows (CRLF) | UTF-8

data/random-order/10000.txt 中存储着 10000 个随机排序的数字



10000.txt - 记事本

文件 编辑 查看

```
817
8743
7949
3816
3229
6769
6109
8941
7168
6230
9831
969
1757
8726
421
8383
474
9931
```

行 1, 列 1 | 100% | Windows (CRLF) | UTF-8



(2) 运行结果截图(部分):

```
D:\Programming\C-CPP\Csteaching\experiment6-Sort\cmake-build-debug\experiment6_Sort.exe
1.Insertion Sort
2.Shell Sort
3.Select Sort
4.Bubble Sort
5.Quick Sort
6.Heap Sort
7.Merge Sort
选择一种排序方式
5
1.正序 2.逆序 3.随机
选择数据的类型
2
1.1000 2.3000 3.10000
选择数据的个数
3
文件名为:      D:\Programming\C-CPP\Csteaching\experiment6-Sort\data\negative-order\10000.txt
数据个数为:    10000
排序方式为:    QuickSort
运行时间为:    55
```

```
D:\Programming\C-CPP\Csteaching\experiment6-Sort\cmake-build-debug\experiment6_Sort.exe
1.Insertion Sort
2.Shell Sort
3.Select Sort
4.Bubble Sort
5.Quick Sort
6.Heap Sort
7.Merge Sort
选择一种排序方式
6
1.正序 2.逆序 3.随机
选择数据的类型
3
1.1000 2.3000 3.10000
选择数据的个数
3
文件名为:      D:\Programming\C-CPP\Csteaching\experiment6-Sort\data\random-order\10000.txt
数据个数为:    10000
排序方式为:    HeapSort
运行时间为:    6

进程已结束,退出代码0
```

## 五、实验分析与探讨

测试结果分析:

### 算法复杂度

	InsertionSort	ShellSort	SelectSort	BubbleSort	QuickSort	HeapSort	MergeSort
时间复杂度	$T(n) = O(n^2)$	$T(n) = O(n^{\frac{3}{2}})$	$T(n) = O(n^2)$	$T(n) = O(n^2)$	平均: $T(n) = O(n \log n)$ 最差: $T(n) = O(n^2)$	$T(n) = O(n \log n)$	$T(n) = O(n \log n)$
空间复杂度	$S(n) = O(1)$	$S(n) = O(1)$	$S(n) = O(1)$	$S(n) = O(1)$	$S(n) = O(1)$	$S(n) = O(1)$	$S(n) = O(n \log n)$

### 运行时间

正序:

	InsertionSort	ShellSort	SelectSort	BubbleSort	QuickSort	HeapSort	MergeSort
1000	2	1	3	1	2	1	1
3000	3	2	8	3	7	2	2
10000	3	3	69	3	51	3	4

逆序:

	InsertionSort	ShellSort	SelectSort	BubbleSort	QuickSort	HeapSort	MergeSort
1000	4	2	2	5	1	1	1
3000	12	2	8	28	8	3	2
10000	100	3	72	288	50	5	3

随机:

	InsertionSort	ShellSort	SelectSort	BubbleSort	QuickSort	HeapSort	MergeSort
1000	3	2	2	4	2	2	2
3000	5	1	8	18	1	2	1
10000	43	6	71	189	3	2	3

在随机时，各个排序方式遵循各自算法复杂度的平均值，其中，QuickSort,HeapSort 和 MergeSort 的时间复杂度均为 $T(n) = O(n \log n)$ ，希尔排序的时间复杂度为 $T(n) = O(n^{\frac{3}{2}})$ 。插入排序，选择排序和冒泡排序时间复杂度均为 $T(n) = O(n^2)$ 。由于

$$O(n \log n) < O\left(n^{\frac{3}{2}}\right) < O(n^2)$$

因此在大量随机数据时，快速排序，堆排序和归并排序的时间复杂度明显好于希尔排序，而希尔排序明显好于插入排序，选择排序和冒泡排序。

在数据为正序时，插入排序，冒泡排序的时间复杂度接近 $O(n)$ ，而快速排序由于每次选择的主元都为第一个（未随机选取主元），导致每次划分时两个数组中一个没有元素，而另一个有  $n-1$  个元素，递归表达式变为了 $T(n) = T(n-1) + T(0) + O(n)$ 因此为最坏情况，时间复杂度退化到了 $O(n^2)$ 。

在逆序时，插入排序和冒泡排序需要比较的次数显著增多，因此时间复杂度中的常量阶明显增大，排序的时间相较于正序和随机明显增大。而快速排序仍因为没有随机选取主元导致时间复杂度退化到了 $T(n) = O(n^2)$

## （2）对快速排序的优化

由上面的分析，我们了解到快速排序在正序和逆序时效率会退化，因此我们可以改进快速排序的代码，对快速排序进行随机化处理，从而使得算法对于所有的输入都能获得较好的期望性能。

代码如下：

```
1 int RandomizedPartition(int *a,int p,int r)
2 {
3     int i = (rand()%(r-p+1))+p; //选取[p,r]之间的随机数
4     swap(a[p],a[i]);
5     return Partition(a,p,r);
6 }
```

```
1 void QuickSort(int a[],int p,int r)
2 {
3     if(p<r)
4     {
5         int q = RandomizedPartition(a,p,r);
6         QuickSort(a,p,q-1);
7         QuickSort(a,q+1,r);
8     }
9 }
```

## 六、小结，

至此，排序的程序设计暂时告一段落。本次的测试也有不足之处，比如没有采用多次试验取平均值，导致函数运行的时间波动较大，可能会影响数据的统计。

## 附录：源代码

(1) 实验环境 Clion 2022.2.4 编译器:mingw

(2) 源代码:

```
(3) #include<iostream>
#include <ctime>
#include <fstream>

using namespace std;

class Sort
{
public:
    Sort(int *a, int n);

    ~Sort() = default;

    void InsertionSort();

    void ShellSort();

    void SelectSort();

    void BubbleSort();

    void QuickSort(int first, int last);

    void HeapSort();

    void MergeSort(int p, int r);

    void MergeSort2();

    void Print();

private:
    //快速排序的划分
    int Partition(int first, int last);

    //构建最大堆和维护最大堆
```

```

void MaxHeapify(int i);

void BuildMaxHeap();

int Left(int i)
{ return 2 * i + 1; }

int Right(int i)
{ return 2 * i; }

int Parent(int i)
{ return (i - 1) / 2; }

//归并排序
void Merge(int p, int q, int r);

void MergePass(int h);

int *data;
int length;
};

Sort::Sort(int *a, int n)
{
    data = new int[n];
    for (int i = 0; i < n; i++)
    {
        this->data[i] = a[i];
    }
    this->length = n;
}

void Sort::InsertionSort()
{
    for (int j = 1; j < this->length - 1; j++)
    {
        int key = this->data[j];
        int i = j - 1;
        while (i >= 0 && this->data[i] > key)
        {
            this->data[i + 1] = this->data[i];
            i--;
        }
        this->data[i + 1] = key;
    }
}

```

```

    }
}

void Sort::ShellSort()
{
    int d, i, j, temp;
    for (d = this->length / 2; d >= 1; d = d / 2)
    {
        for (i = d; i < length; i++)
        {
            temp = data[i];
            for (j = i - d; j >= 0 && temp < data[j]; j = j - d)
            {
                data[j + d] = data[j];
            }
            data[j + d] = temp;
        }
    }
}

```

```

void Sort::BubbleSort()
{
    int j, exchange, bound;
    exchange = length - 1;
    while (exchange != 0)
    {
        bound = exchange;
        exchange = 0;
        for (j = 0; j < bound; j++)
        {
            if (data[j] > data[j + 1])
            {
                swap(data[j], data[j + 1]);
                exchange = j;
            }
        }
    }
}

```

```

void Sort::QuickSort(int first, int last)
{
    if (first < last)
    {

```

```

        int p = Partition(first, last);
        QuickSort(first, p - 1);
        QuickSort(p + 1, last);
    }
}

```

```

int Sort::Partition(int first, int last)
{
    int p = data[first];
    int i = first;
    int j = last;
    while (i < j)
    {
        while (data[j] >= p && i < j)
        {
            j--;
        }
        while (data[i] <= p && i < j)
        {
            i++;
        }
        if (i < j)
        {
            swap(data[i], data[j]);
        }
    }
    swap(data[first], data[j]);
    return j;
}

```

```

void Sort::Print()
{
    for (int i = 0; i < length; i++)
    {
        cout << data[i] << " ";
    }
    cout << endl;
}

```

```

void Sort::HeapSort()
{
    int temp = length;
    for (int i = length - 1; i > -0; i--)
    {

```

```

        swap(data[0], data[i]);
        length--;
        MaxHeapify(0);
    }
    length = temp;
}

void Sort::MaxHeapify(int i)
{
    int l = Left(i);
    int r = Right(i);
    int largest;
    if (l < length && data[l] > data[i])
    {
        largest = l;
    } else
    {
        largest = i;
    }
    if (r < length && data[r] > data[largest])
    {
        largest = r;
    }
    if (largest != i)
    {
        swap(data[i], data[largest]);
        MaxHeapify(largest);
    }
}

void Sort::BuildMaxHeap()
{
    for (int i = length / 2 - 1; i >= 0; i--)
    {
        MaxHeapify(i);
    }
}

void Sort::MergeSort(int p, int r)
{
    if (p < r)
    {
        int q = (p + r) / 2;
        MergeSort(p, q);
    }
}

```



```

        MergeSort(q + 1, r);
        Merge(p, q, r);
    } else
    {
        return;
    }
}

```

```

void Sort::Merge(int p, int q, int r)
{
    int temp[length];
    int i = p;
    int j = q + 1;
    int k = p;
    while (i <= q && j <= r)
    {
        if (data[i] <= data[j])
        {
            temp[k++] = data[i++];
        }
        else
        {
            temp[k++] = data[j++];
        }
    }
    while (i <= q)
    {
        temp[k++] = data[i++];
    }
    while (j <= r)
    {
        temp[k++] = data[j++];
    }
    for (i = p; i <= r; i++)
    {
        data[i] = temp[i];
    }
}

```

```

void Sort::MergePass(int h)
{
    int i = 0;
    while (i + 2 * h <= length)
    {

```

```

        Merge(i, i+h-1, i+2*h-1);
        i = i + 2 * h;
    }
    if (i + h < length)
        Merge(i, i+h-1, length-1);
}

void Sort::MergeSort2( )
{
    int h = 1;
    while (h < length)
    {
        MergePass(h);
        h = 2 * h;
    }
}

void Sort::SelectSort( )
{
    int i, j, index, temp;
    for (i = 0; i < length - 1; i++)
    {
        index = i;
        for (j = i + 1; j < length; j++)
            if (data[j] < data[index]) index = j;
        if (index != i) {
            temp = data[i]; data[i] = data[index]; data[index] =
temp;
        }
    }
}

string fn(int type, int num)
{
    string filename;
    string type_filename;
    string num_filename;
    switch(type)
    {
        case 1:
            type_filename = "positive-order";
            break;
        case 2:
            type_filename = "negative-order";

```

```

        break;
    case 3:
        type_filename = "random-order";
        break;
    default:
        break;
}
switch(num)
{
    case 1:
        num_filename = "1000.txt";
        break;
    case 2:
        num_filename = "3000.txt";
        break;
    case 3:
        num_filename = "10000.txt";
        break;
    default:
        break;
}
filename = R"(D:\Programming\C-CPP\Csteaching\experiment6-Sort\data)";
return filename;
}

```

```

int GetNum(int num)
{
    switch (num)
    {
        case 1:
            return 1000;
        case 2:
            return 3000;
        case 3:
            return 10000;
        default:
            return -1;
    }
}

```

```

int main()
{

```

```

    cout<<"1.Insertion Sort\n2.Shell Sort\n3.Select Sort\n4.Bubble Sort\n5.Quick Sort\n6.Heap Sort\n7.Merge Sort\n";
    cout<<"选择一种排序方式\n";
    int select;
    cin>>select;

    cout<<"1.正序 2.逆序 3.随机\n";
    cout<<"选择数据的类型\n";
    int type;
    cin>>type;

    cout<<"1.1000 2.3000 3.10000\n";
    cout<<"选择数据的个数\n";
    int num;
    cin>>num;

    //auto filename = fn(type,num);
    string filename = R"(D:\Programming\C-CPP\Csteaching\experiment6-Sort\data\positive-order\10000.txt)";
    auto number = GetNum(num);
    auto begin = clock();
    const int length = number-1;

    int a[length];
    int temp;
    int count = 0;
    cout<<"文件名为:\t"<<filename<<endl;
    ifstream fin(filename);
    cout<<"数据个数为:\t"<<GetNum(num)<<endl;
    if(!fin.is_open())
    {
        cout<<"Open file wrong"<<endl;
        exit(0);
    }
    while(!fin.eof())
    {
        fin>>a[count];
        count++;
    }
    Sort(L)(a,length);
    switch(select)
    {
        case 1:
            cout<<"排序方式为:\tInsertionSort"<<endl;

```

```

        L.InsertionSort();
        break;
    case 2:
        cout<<"排序方式为:\tShellSort"<<endl;
        L.ShellSort();
        break;
    case 3:
        cout<<"排序方式为:\tSelectSort"<<endl;
        L.SelectSort();
        break;
    case 4:
        cout<<"排序方式为:\tBubbleSort"<<endl;
        L.BubbleSort();
        break;
    case 5:
        cout<<"排序方式为:\tQuickSort"<<endl;
        L.QuickSort(0,length-1);
        break;
    case 6:
        cout<<"排序方式为:\tHeapSort"<<endl;
        L.HeapSort();
        break;
    case 7:
        cout<<"排序方式为:\tMergeSort"<<endl;
        L.MergeSort(0,length-1);
        break;
    default:
        break;
}
//L.Print();
auto end = clock();
auto time = end-begin;
cout<<"运行时间为:\t"<<time<<endl;
}

```