项目说明文档

数据结构课程设计

——8种排序算法的比较案例

同济大学

Tongji University

目 录

[1 分析 1](#_Toc59930235)

[1.1 背景分析 1](#_Toc59930236)

[1.2 功能分析 1](#_Toc59930237)

[2 设计 1](#_Toc59930238)

[2.1 数据结构设计 1](#_Toc59930239)

[2.2 类结构设计 2](#_Toc59930240)

[2.3 成员与操作设计 2](#_Toc59930241)

[2.4 系统设计 3](#_Toc59930242)

[3 实现 4](#_Toc59930243)

[3.1 冒泡的实现 4](#_Toc59930244)

[3.1.1 冒泡排序流程图 4](#_Toc59930245)

[3.1.2 冒泡排序核心代码 6](#_Toc59930246)

[3.1.3 冒泡排序截屏示例 7](#_Toc59930247)

[3.2 选择排序的实现 8](#_Toc59930248)

[3.2.1 选择排序流程图 8](#_Toc59930249)

[3.2.2 选择排序核心代码 9](#_Toc59930250)

[3.2.3 选择排序截屏示例 9](#_Toc59930251)

[3.3 直接插入排序的实现 10](#_Toc59930252)

[3.3.1 直接插入流程图 10](#_Toc59930253)

[3.3.2 直接插入排序核心代码 11](#_Toc59930254)

[3.3.3 直接插入排序截图示例 11](#_Toc59930255)

[3.4 希尔排序的实现 12](#_Toc59930256)

[3.4.1 希尔排序的思想 12](#_Toc59930257)

[3.4.2 希尔排序核心代码 13](#_Toc59930258)

[3.4.3 希尔排序截屏示例 14](#_Toc59930259)

[3.5 堆排序的实现 14](#_Toc59930260)

[3.5.1 堆排序思想 14](#_Toc59930261)

[3.5.2 堆排序核心代码 14](#_Toc59930262)

[3.5.3堆排序截屏示例 15](#_Toc59930263)

[3.6 快速排序的实现 16](#_Toc59930264)

[3.6.1 快速排序流程图 16](#_Toc59930265)

[3.6.2 快速排序核心代码 18](#_Toc59930266)

[3.6.3 快速排序截屏示例 19](#_Toc59930267)

[3.7 归并排序的实现 19](#_Toc59930268)

[3.7.1 归并排序的思想 19](#_Toc59930269)

[3.7.2 归并排序核心代码 20](#_Toc59930270)

[3.7.3 归并排序截屏示例 21](#_Toc59930271)

[3.8 基数排序的实现 21](#_Toc59930272)

[3.8.1 基数排序的思想 21](#_Toc59930273)

[3.8.2 基数排序核心代码 21](#_Toc59930274)

[3.8.3 基数排序截屏示例 23](#_Toc59930275)

[4 各种排序方法优缺点分析 24](#_Toc59930276)

[4.1 实验结果 24](#_Toc59930277)

[4.1.1 数据量：1000 24](#_Toc59930278)

[4.1.2 测试数据：10000 25](#_Toc59930279)

[4.1.3 测试数据：100000 26](#_Toc59930280)

[4.1.4 测试数据：1000，000（仅测试后五种排序方法） 28](#_Toc59930281)

[4.1.5 测试数据：5，000，000（仅测试后五种排序方法） 29](#_Toc59930282)

[4.2 结果分析 30](#_Toc59930283)

[4.2.1 简单排序性能对比 30](#_Toc59930284)

[4.2.2 高性能排序效率比较： 32](#_Toc59930285)

[4.2.3 其他比较 34](#_Toc59930286)

# 1 分析

## 1.1 背景分析

排序是计算机程序设计中的一种重要操作，它的功能是将一个数据元素(或记录)的任意序列，重新排列成一个按关键字有序的序列。

为了查找方便,通常希望计算机中的表是按关键字有序的。因为有序的顺序表可以采用查找效率较高的折半查找法，其平均查找长度为log(n)，而无序的顺序表只能进行顺序查找，其平均查找长度为(n+1)/2。又如建造树表(无论是二叉排序树或B-树)的过程本身就是一个排序的过程。因此,学习和研究各种排序方法是计算机工作者的重要课题之一。

## 1.2 功能分析

随机函数产生一百，一千，一万和十万个随机数，用快速排序，直接插入排序，冒泡排序，选择排序的排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机数。并且显示他们的比较次数。

由功能可知，需要产生随机数，之后要能实现八种排序方法，对数组进行排序，并且输出排序时间、交换时间和比较时间。

# 2 设计

## 2.1 数据结构设计

如上背景分析所述，排序应用广泛，因此考虑使用模板类，从而使得程序能够泛用。同时，在存储结构上，使用静态数组，该数组由用户提供（不作为类中的元素），在测试程序中，由Array类提供测试数组。为了实现简易，保留数组的首元素（arr[0]），用以存储中间量，使得程序简洁。

## 2.2 类结构设计

排序类本身是一个函数的集合，没有相应的私有成员。但为了化解递归，使用了栈这种数据结构。

## 2.3 成员与操作设计

**栈类（STACK）**

**为了使栈能够泛用，使用类模板T**

**私有成员：**

T\* \_top; //表示栈顶的上一个元素

T\* \_base; //表示栈底元素

int stackSize; //表示栈在内存中申请的大小

const int STACK\_INIT\_SIZE = 100; //规定栈初始化时申请的内存

**公有成员：**

Stack(); //构造函数

Status popStack(T&); //弹出栈，不会报错

T pop(); //弹出栈，会报错

Status push(T); //将T加入栈

Status get\_top(T&); //取栈首元素，不会报错

T top(); //取栈首元素，会报错

int size() //取栈的大小

bool empty() //判断栈是否为空

**排序函数（sorting functions）**

void BubbleSort(T \*arr,int \_num) //冒泡排序

void SelectSort(T \*arr,int \_num) //选择排序

void InsertSort(T \*arr,int \_num) //直接插入排序

void ShellInsert(T\*arr,int \_num, int dk,long long &exchangeTime)

//希尔插入

void ShellSort(T \*arr,int \_num) //希尔排序

int Partition(T \*arr,int low, int high,long long &exchangeTime,long long &compareTime)

//划分

void QuickSort(T \*arr,int \_num) //快排

void HeapAdjust(T\* arr, int s, int m,long long &exchangeTime,long long &compareTime) //堆筛选

void HeapSort(T\*arr,int \_num) //堆排序

void merge(T \*arr,int low, int high,int mid,long long &exchangeTime)

//归并

void MergeSort(T\* arr, int \_num) //归并排序

int maxbit(T\* data, int n) //辅助函数，求数据的最大位数

void radixsort(T\* data, int \_num) //基数排序

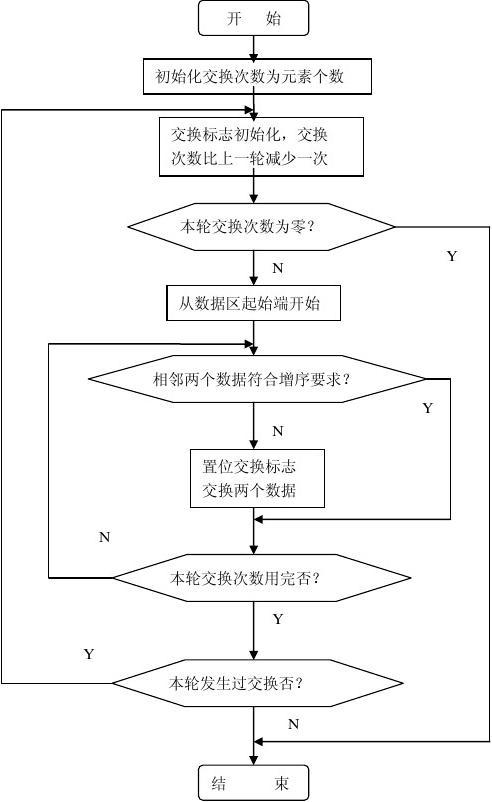
## 2.4 系统设计

系统首先调用printHead()函数实现对屏幕的初始化，之后调用Array中的函数来提供无序数组（先产生一份无序数组，之后每一次要排序的时候再复制一份），然后根据用户所输入的操作码执行相应的排序操作。

# 3 实现

## 3.1 冒泡的实现

### 3.1.1 冒泡排序流程图



优化说明：

每一次遍历时检查是否有发生数据的交换，如果没有数据的交换则说明数组已经有序，不必再排。在此优化下，最优时间复杂度为O(n)，最优情况为有序的数组，只需要遍历一次。

### 3.1.2 冒泡排序核心代码

void BubbleSort(T \*arr,int \_num)

{

long long exchangeTime = 0, compareTime = 0;

int runTime;

bool change = 1;//记录是否有发生交换

clock\_t startTime, endTime;

startTime = clock();

for (int i = \_num; change && i > 0; --i)

{

change = 0;

//一轮起泡过程

for (int j = 1; j < i; ++j)

if (arr[j] > arr[j + 1])

{

swap(arr[j], arr[j + 1]);

change = 1;

++exchangeTime;

}

compareTime += i - 1;

}

endTime = clock();

runTime = double(endTime - startTime) \* 1000 / CLOCKS\_PER\_SEC;

cout << "冒泡排序所用时间：\t" << runTime << "ms\n";

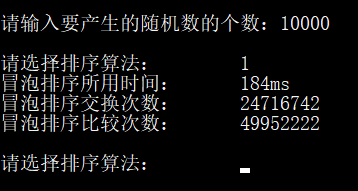
cout << "冒泡排序交换次数：\t" << exchangeTime << endl;

cout << "冒泡排序比较次数：\t" << compareTime << endl;

}

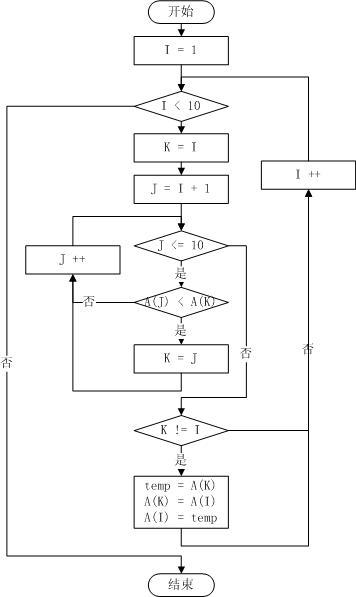
### 3.1.3 冒泡排序截屏示例

（在最后数组有序的情况下，冒泡排序停止，比较次数为49952222<4999500）



## 3.2 选择排序的实现

### 3.2.1 选择排序流程图



优化说明：在插入的过程中，只要发现待插入元素大于当前元素的前一个元素，即可停止搜索，因为在该元素之前的元素均已有序，故不存在大于待插入元素的值，可以将待插入元素直接插入在该处。

### 3.2.2 选择排序核心代码

for (int i = 1; i <= \_num; ++i)

{

int k = i;

for (int j = i + 1; j <= \_num; ++j)

if (arr[j] < arr[k]) { k = j; ++exchangeTime; }

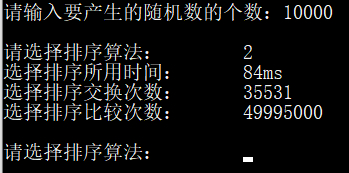
//将i与k进行交换

if (k != i) { swap(arr[i], arr[k]); exchangeTime += 3; }//一次赋值等效于1/3次交换

}

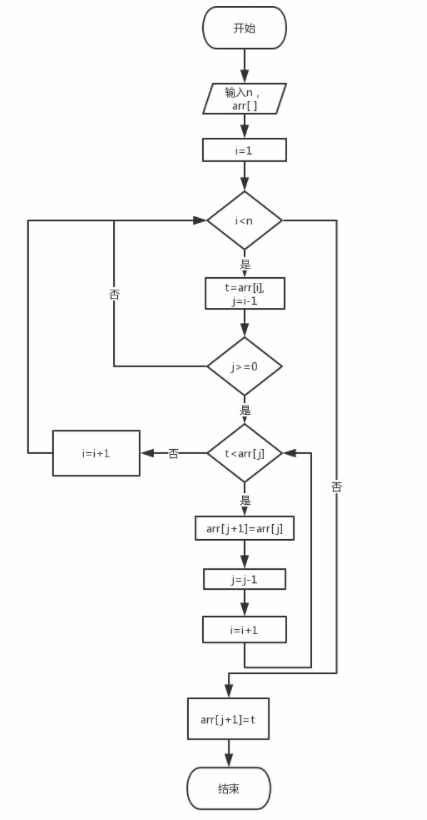
### 3.2.3 选择排序截屏示例

（由于采取从数组中择取最小者，再与最后一个无序者相交换的策略，所以交换次数很少）



## 3.3 直接插入排序的实现

### 3.3.1 直接插入流程图



### 3.3.2 直接插入排序核心代码

for (int i = 2; i <= \_num; ++i)

{

if (arr[i] < arr[i - 1])

{

arr[0] = arr[i];

arr[i] = arr[i - 1];

int j = 0;

//寻找小于待插入元素的元素

for (j = i - 2; arr[0] < arr[j]; --j)

{

arr[j + 1] = arr[j];

++compareTime;

}

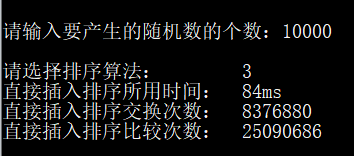
arr[j + 1] = arr[0];

exchangeTime += i - j + 2;

}

}

### 3.3.3 直接插入排序截图示例

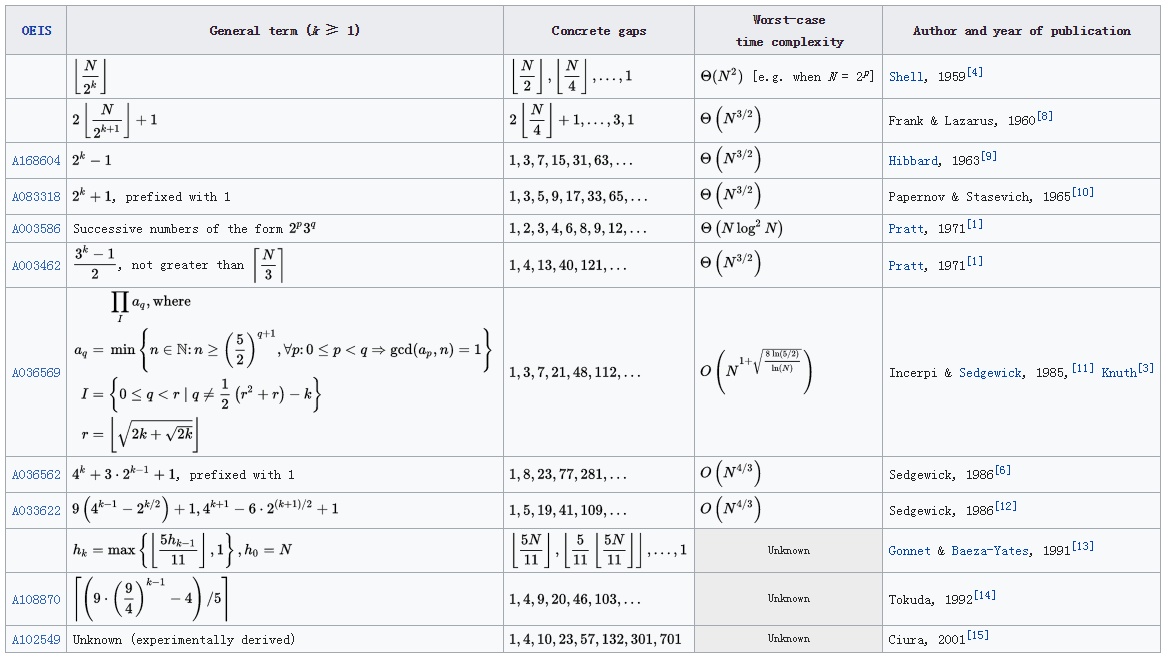


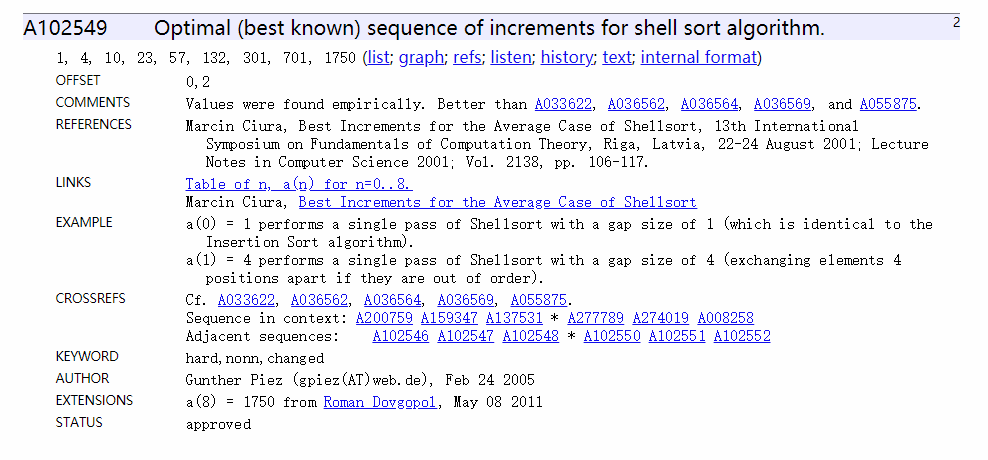
## 3.4 希尔排序的实现

### 3.4.1 希尔排序的思想

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

在本程序中，使用Marcin Ciura间距值。查阅资料得，使用不同得间距值对于时间复杂度具有不小得影响，下表是一些情况下的间距值以及其最坏时间复杂度：





### 3.4.2 希尔排序核心代码

shellInsert函数中核心部分：

//arr[0]暂存元素

for(int i=dk+1;i<=\_num;++i)

if (arr[i] < arr[i - dk])

{//需要将arr[i]插入有序增量子表

arr[0] = arr[i];

int j;

//记录后移，寻找插入位置

for (j = i - dk; j > 0 && arr[0] < arr[j]; j -= dk)

arr[j + dk] = arr[j];

arr[j + dk] = arr[0];//插入

exchangeTime += 2 + (i - j ) / dk;

}

shellSort函数中核心部分：

int dlta[9] = { 1750, 701, 301, 132, 57, 23, 10, 4, 1 };//Marcin Ciura间距值

for (int i = 0; i < 9; ++i)

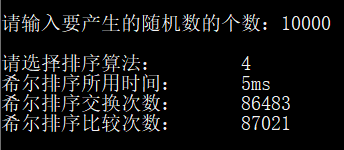
{

ShellInsert(arr, \_num, dlta[i], exchangeTime);

compareTime += \_num - dlta[i];

}

### 3.4.3 希尔排序截屏示例



## 3.5 堆排序的实现

### 3.5.1 堆排序思想

将待排序的序列构造成一个大顶堆。此时，整个序列的最大值就是堆顶的根节点。将它移走(其实是将其与堆数组的末尾元素交换，此时末尾元素就是最大值)，然后将剩余的n-1个序列重新构造成一个堆，这样就会得到n个元素中的次最大值。如此反复执行，就能得到一个有序序列了。这个过程其实就是先构建一个最大/最小二叉堆，然后不停地取出最大/最小元素（头结点），插入到新的队列中，以此达到排序的目的。

### 3.5.2 堆排序核心代码

HeapAdjust部分代码：

int rc = arr[s];

for (int j = (s << 1); j <= m; j = (j << 1))

{

//沿值较大的孩子向下筛选

if (j < m && arr[j] < arr[j + 1])++j;

if (arr[j] <= rc)break;

arr[s] = arr[j]; s = j;

exchangeTime += 2;

compareTime += 2;

}

arr[s] = rc;

exchangeTime += 2;

HeapSort部分核心代码：

for (int i =( \_num >> 1); i > 0; --i)

HeapAdjust(arr, i, \_num,exchangeTime,compareTime);

for (int i = \_num; i > 1; --i)

{

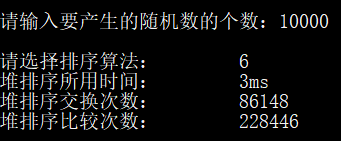
swap(arr[1], arr[i]);

//将arr重新调整为大顶堆

HeapAdjust(arr, 1, i - 1, exchangeTime,compareTime);

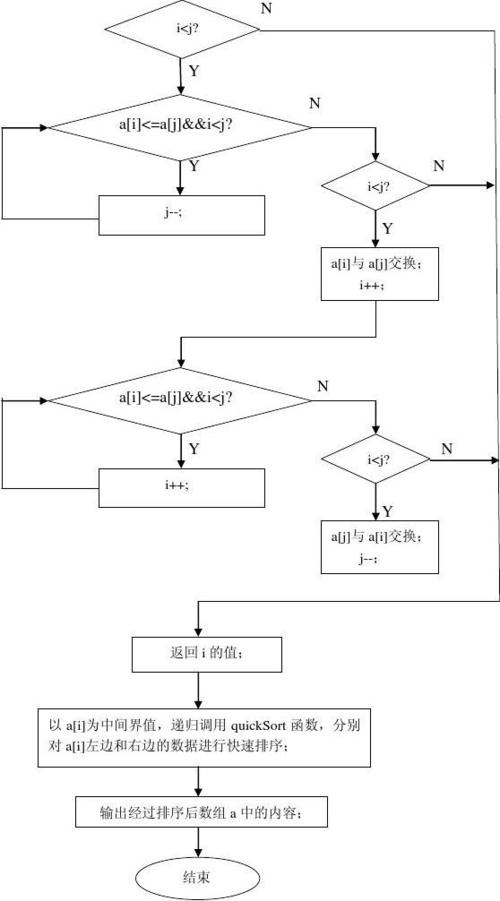
}

### 3.5.3堆排序截屏示例



## 3.6 快速排序的实现

### 3.6.1 快速排序流程图



算法优化：

若初始记录序列按关键字有序或基本有序时，快速排序将蜕化为起泡排序，其时间复杂度为。为改进之，本程序依“三者取中”的法则来选取枢轴记录，即比较arr[low],arr[mid],arr[high]，取三者中其关键字取中值的记录为枢轴，之后将该记录和arr[low]互换即可。

同时，本程序采用非递归算法，防止在最坏情况下枢轴位置偏向子序列的一端，达到O(n)的深度，导致栈溢出。

### 3.6.2 快速排序核心代码

Partition部分核心代码：

int mid = (low + high) >> 1;

if (arr[low] < arr[mid])swap(arr[mid], arr[low]);

if (arr[high] < arr[low])swap(arr[high], arr[low]);

if (arr[low] < arr[mid])swap(arr[mid], arr[low]);

arr[0] = arr[low];//取子表的第一个作为枢轴记录

int pivotKey = arr[low];

compareTime += high - low;

exchangeTime += high - low;

while (low < high)

{

while (low < high && pivotKey <= arr[high])--high;

arr[low] = arr[high];//将比枢轴小的记录移动到低端

while (low < high && arr[low] <= pivotKey)++low;

arr[high] = arr[low];//将比枢轴大的记录移动到高端

exchangeTime += 2;

++compareTime;

}

arr[low] = arr[0];

exchangeTime += 11;

return low;//返回枢轴位置

QuickSort部分核心代码：

//采用非递归算法，防止在最坏情况下栈的深度为O(n)导致栈溢出。

if (low < high)

{

int mid = Partition(arr, low, high, exchangeTime,compareTime);

if (low < mid - 1) { S.push(low); S.push(mid - 1); }

if (mid + 1 < high) { S.push(mid + 1); S.push(high); }

}

while (S.size())

{

int q = S.pop(), p = S.pop();

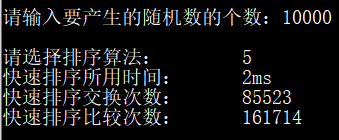
int mid = Partition(arr, p, q, exchangeTime, compareTime);

if (p < mid - 1) { S.push(p); S.push(mid - 1); }

if (mid + 1 < q) { S.push(mid + 1); S.push(q); }

}

### 3.6.3 快速排序截屏示例



## 3.7 归并排序的实现

### 3.7.1 归并排序的思想

本程序使用自底向上的方法来实现递归排序。

自底向上的归并排序算法的思想就是数组中先一个一个归并成两两有序的序列，两两有序的序列归并成四个四个有序的序列，然后四个四个有序的序列归并八个八个有序的序列，以此类推，直到，归并的长度大于整个数组的长度，此时整个数组有序。需要注意的是数组按照归并长度划分，最后一个子数组可能不满足长度要求，这个情况需要特殊处理。这样，就可以把常规的递归算法改成迭代算法。

本程序使用了一些优化手段：比如只有在arr[mid-1]>arr[mid]时才进行归并；并且通过实验验证，发现gap在32及以下时使用插入排序比使用辅助数组归并更快。最后优化后的速度大约是优化前的2.83倍（在数组大小为10000时的测试结果）。

### 3.7.2 归并排序核心代码

Merge部分代码：

//gap小于等于32时使用插入排序

if (high - low <= 33)

{

for (int i = mid; i < high; ++i)

{

if (arr[i] < arr[i - 1])

{

arr[0] = arr[i];

arr[i] = arr[i - 1];

int j;

for (j = i - 2; j >= low && arr[0] < arr[j]; --j) { arr[j + 1] = arr[j]; ++exchangeTime; }

arr[j + 1] = arr[0];

exchangeTime += 3;

}

}

return;

}

//否则使用递归排序

T\* tempArr = new T[mid - low];

//辅助数组只需要复制一半

for (int i = low; i < mid; ++i)tempArr[i - low] = arr[i];

int p = low, q = mid, cur = p;

while (p < mid || q < high)

{

if (q >= high || (p < mid && tempArr[p - low] < arr[q]))

arr[cur++] = tempArr[p++ - low];

else arr[cur++] = arr[q++];

}

exchangeTime += (high + mid - 2 \* low);

delete[]tempArr;

mergeSort部分代码

int n = \_num;

int s = 2, i;

while (s <= n) {

i = 1;

while (i + s <= n + 1) {

//只有在arr[mid - 1] > arr[mid]时才进行归并

if (arr[i + s / 2 - 1] > arr[i + s / 2])

{

merge(arr, i, i + s, i + s / 2, exchangeTime);

compareTime += s;

}

i += s;

}

//cout << i<<' '<<i + s / 2 << endl;

//处理末尾残余部分

if (i + s / 2 < n + 1) merge(arr, i, n + 1, i + s / 2, exchangeTime);

compareTime += n + 1 - i;

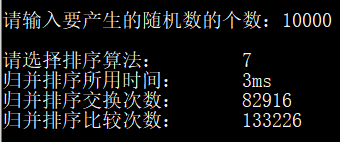
s = (s << 1);

}

//最后再从头到尾处理一遍

merge(arr, 1, n + 1, s / 2 + 1, exchangeTime);

### 3.7.3 归并排序截屏示例



## 3.8 基数排序的实现

### 3.8.1 基数排序的思想

本程序使用LSD(Least Significant Digit first)法，从最次位关键字起进行排序。然后再对高一位的关键字进行排序,依次重复,直至对进行排序后便成为一个有序序列。

### 3.8.2 基数排序核心代码

maxibit部分：

T maxData = data[1]; ///< 最大数

/// 先求出最大数，再求其位数，这样有原先依次每个数判断其位数

for (int i = 2; i <= n; ++i)

{

if (maxData < data[i])

maxData = data[i];

}

int d = 1;

int p = 10;

while (maxData >= p)

{

maxData /= 10;

++d;

}

return d;

radixsort部分：

int n = \_num;

int d = maxbit(data, n);

T\* tmp = new T[n + 1];

int\* count = new int[10]; //计数器

int i, j, k;

int radix = 1;

for (i = 1; i <= d; i++) //进行d次排序

{

for (j = 0; j < 10; j++)

count[j] = 0; //每次分配前清空计数器

for (j = 1; j <= n; j++)

{

k = (data[j] / radix) % 10; //统计每个桶中的记录数

count[k]++;

}

for (j = 1; j < 10; j++)

count[j] = count[j - 1] + count[j]; //将tmp中的位置依次分配给每个桶

for (j = n; j > 0; j--) //将所有桶中记录依次收集到tmp中

{

k = (data[j] / radix) % 10;

tmp[count[k]] = data[j];

count[k]--;

}

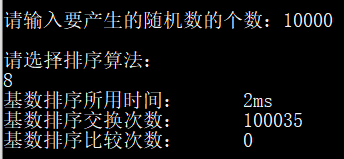
for (j = 1; j <= n; j++) //将临时数组的内容复制到data中

data[j] = tmp[j];

radix = radix \* 10;

}

### 3.8.3 基数排序截屏示例



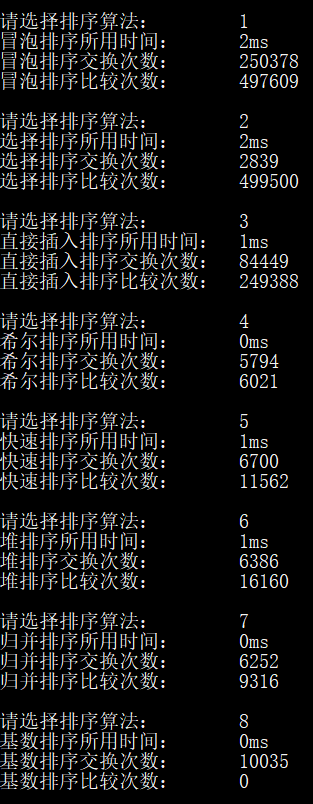
# 4 各种排序方法优缺点分析

## 4.1 实验结果

测试环境：visual studio community 2019 release

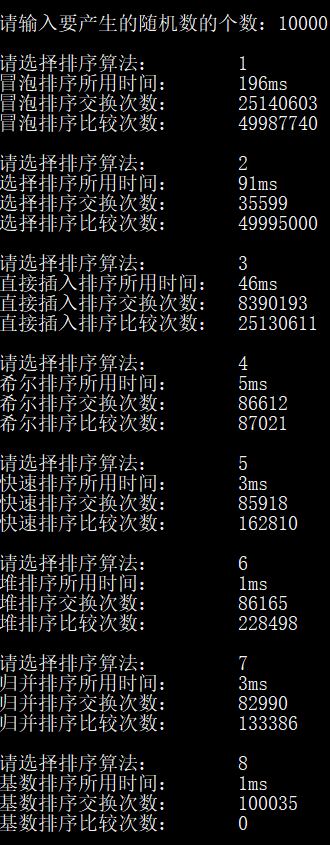
### 4.1.1 数据量：1000

**实验结果**



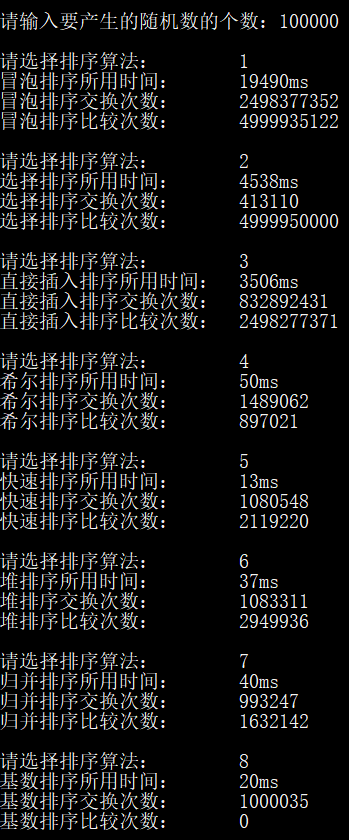
### 4.1.2 测试数据：10000

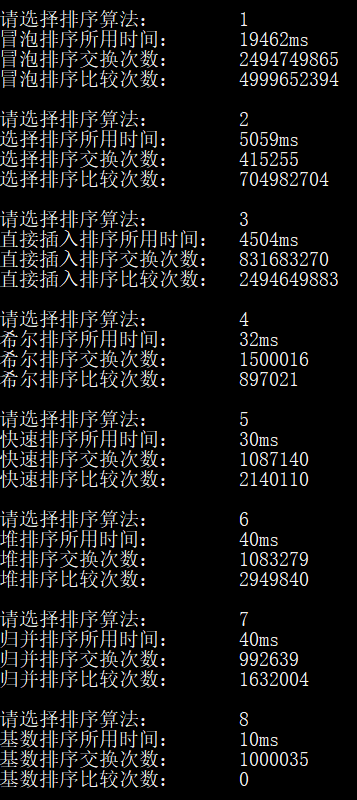
**实验结果：**



### 4.1.3 测试数据：100000

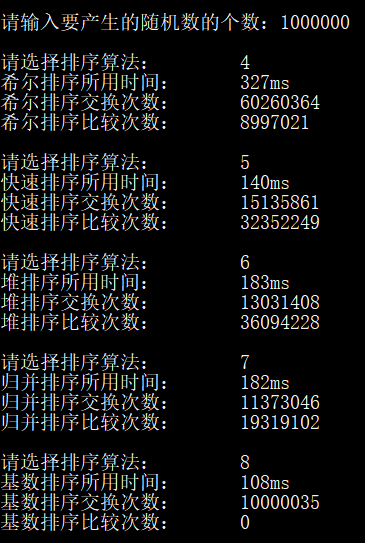
**实验结果：**

****



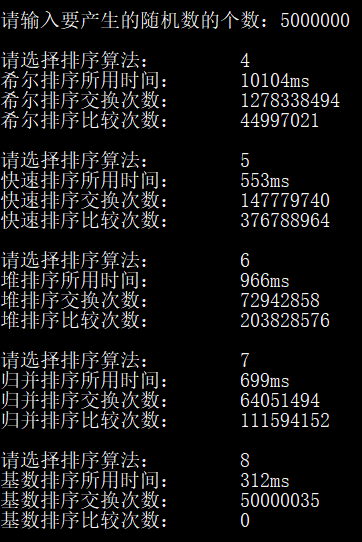
### 4.1.4 测试数据：1000，000（仅测试后五种排序方法）

**实验结果：**



### 4.1.5 测试数据：5，000，000（仅测试后五种排序方法）

**实验结果：**



## 4.2 结果分析

### 4.2.1 简单排序性能对比

时间对比：



归一化处理后：



可以发现，三者的时间消耗比大约是3.535：1.821：1

比较三者的交换次数：



归一化处理：



发现三者大约是233.2：1：78.0这样的比例，其中冒泡排序和直接插入排序交换次数比是2.99：1，接近两者的时间消耗比。

比较三者的比较次数：



归一化处理



可以发现，三者比值接近2：2：1，其中选择排序和直接插入排序的比较次数比例2：1接近它们的时间比。

综上可以发现，这三者虽然时间复杂度均为，但是效率上直接插入排序>选择排序>冒泡排序。其中，冒泡排序交换次数和比较次数都是最多的，而选择排序的时间消耗主要在比较上，而直接插入排序的时间消耗主要在交换上。

### 4.2.2 高性能排序效率比较：

时间对比：



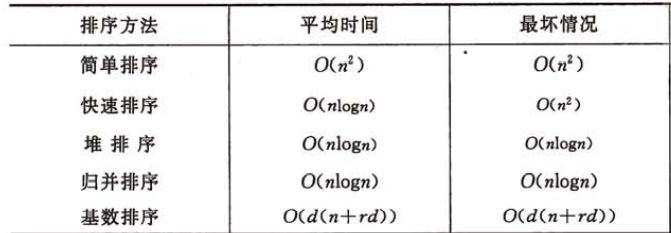
假设这些排序的时间复杂度均为。将时间除以之后（再乘以1000，000），再进行归一化处理，得到它们除以的系数：



画出折线图：

仅仅凭借这些实验结果，发现快速排序、堆排序以及归并排序效率接近，而基数排序效率最高，使用Marcin ciura间距值的希尔排序随着n的增大，效率逐渐变低。

而根据课本上学习的理论知识：



当关键字数小时，基数排序的效率确实可能会高于 时间复杂度的算法，在该程序中，rand()的最大值是，最多包含5个关键字，所以会在这些算法中表现得最好；而使用Marcin ciura间距值的希尔排序所消耗的平均时间尚未得到证明；而有过证明的希尔排序最优最坏时间复杂度为，在n很高的时候，时间复杂度不如的高效排序算法。

### 4.2.3 其他比较

从方法的稳定性来比较，基数排序是稳定的内排方法,所有简单排序法也是稳定的，然而，快速排序、堆排序和希尔排序等时间性能较好的排序方法都是不稳定的。

从占用存储来比较：

|  |  |
| --- | --- |
| 排序方法 | 辅助存储 |
| 简单排序 | O(1) |
| 快速排序 | O(logn) |
| 堆排序 | O(1) |
| 归并排序 | O(n) |
| 基数排序 | O(rd) |
| 希尔排序 | O(1) |

从空间来看，简单排序/希尔排序/堆排序>快速排序>归并排序/基数排序。