项目说明文档

数据结构课程设计

——8种排序算法的比较

作 者 姓 名： 谢宇翔

学 号： 1951708

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 1](#_Toc11400)

[1.1 背景分析 1](#_Toc11634)

[1.2 功能分析 1](#_Toc21541)

[2 设计 1](#_Toc13220)

[2.1 数据结构设计 1](#_Toc27284)

[2.2 类结构设计 1](#_Toc5356)

[2.3 系统设计 1](#_Toc31260)

[3 实现 2](#_Toc26841)

[3.1生成随机数 2](#_Toc8423)

[3.1.1 功能流程图 2](#_Toc3675)

[3.1.2 核心代码 3](#_Toc24020)

[3.1.3程序截屏示例 7](#_Toc24028)

[3.2 冒泡排序 8](#_Toc12446)

[3.2.1 冒泡排序功能流程图 8](#_Toc9529)

[3.2.2 冒泡排序核心代码 8](#_Toc4214)

[3.2.3冒泡排序截屏示例 9](#_Toc1154)

[3.2.4冒泡排序算法分析 10](#_Toc5193)

[3.3 选择排序 11](#_Toc14752)

[3.3.1 选择排序功能流程图 11](#_Toc16961)

[3.3.2 选择排序核心代码 12](#_Toc6748)

[3.3.3 选择排序截图示例 13](#_Toc26692)

[3.3.4选择排序算法分析 13](#_Toc14163)

[3.4直接插入排序 14](#_Toc11648)

[3.4.1直接插入排序流程图 14](#_Toc14302)

[3.4.2直接插入排序核心代码 14](#_Toc19781)

[3.4.3直接插入排序截屏示例 16](#_Toc15870)

[3.4.4直接插入排序算法分析 16](#_Toc11705)

[3.5希尔排序 17](#_Toc23232)

[3.5.1希尔排序流程图 17](#_Toc22833)

[3.5.2希尔排序核心代码 18](#_Toc7871)

[3.5.3希尔排序截屏示例 19](#_Toc22312)

[3.5.4希尔排序算法分析 19](#_Toc28581)

[3.6快速排序 20](#_Toc2612)

[3.6.1快速排序流程图 20](#_Toc21220)

[3.6.2快速排序核心代码 21](#_Toc8811)

[3.6.3快速排序截屏示例 22](#_Toc7931)

[3.6.4快速排序算法分析 22](#_Toc28688)

[3.7堆排序 24](#_Toc20471)

[3.7.1堆排序流程图 24](#_Toc127)

[3.7.2堆排序核心代码 24](#_Toc24866)

[3.7.3堆排序截屏示例 26](#_Toc32693)

[3.7.4堆排序算法分析 26](#_Toc23800)

[3.8归并排序 28](#_Toc27371)

[3.8.1归并排序流程图 28](#_Toc7632)

[3.8.2归并排序核心代码 28](#_Toc17683)

[3.8.3归并排序截屏示例 30](#_Toc19689)

[3.8.4归并排序算法分析 30](#_Toc17467)

[3.9基数排序 31](#_Toc23220)

[3.9.1基数排序流程图 31](#_Toc26412)

[3.9.2基数排序核心代码 32](#_Toc17040)

[3.9.3基数排序截屏示例 34](#_Toc7344)

[3.9.4基数排序算法分析 34](#_Toc15705)

# 1 分析

## 1.1 背景分析

排序是经常出现在现实生活中的的问题，现在有着许多的排序算法，现在列举了8种算法来比较这些算法的优劣

## 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。

请在文档中记录上述数据量下，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点

# 2 设计

## 2.1 数据结构设计

本程序是需要一个存储不定个数的随机数的存储结构，所以只需要用一个数组存储即可。

## 2.2 类结构设计

本程序不需要使用类

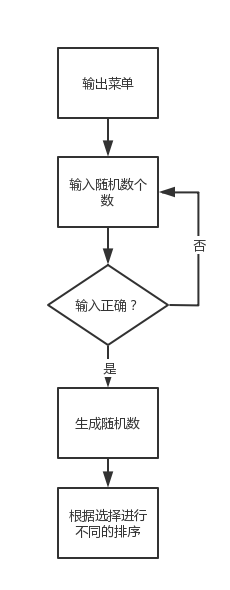
## 2.3 系统设计

首先输入要产生的随机数的个数，然后将随机数存储进数组，再通过关键码的选择进行不同的排序方式，输出排序所用时间，排序交换次数。

# 3 实现

## 3.1生成随机数

### 3.1.1 功能流程图



### 3.1.2 核心代码

int main()

{

srand((unsigned)time(NULL));

cout << "\*\* 排序算法比较 \*\*" << endl;

cout << "=============================================" << endl;

cout << "\*\* 1.冒泡排序 \*\*" << endl;

cout << "\*\* 2.选择排序 \*\*" << endl;

cout << "\*\* 3.直接插入排序 \*\*" << endl;

cout << "\*\* 4.希尔排序 \*\*" << endl;

cout << "\*\* 5.快速排序 \*\*" << endl;

cout << "\*\* 6.堆排序 \*\*" << endl;

cout << "\*\* 7.归并排序 \*\*" << endl;

cout << "\*\* 8.基数排序 \*\*" << endl;

cout << "\*\* 9.退出程序 \*\*" << endl;

cout << "=============================================" << endl;

int num;

cout << "请输入要产生的随机数:" << endl;

while (1)

{

cin >> num;

if (cin.fail() || num <= 0)

{

cout << "输入错误，请重新输入:" << endl;

cin.clear();

cin.ignore(1000, '\n');

}

else

break;

}

int\* true\_number = new int[num];

int\* number = new int[num];

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

{

true\_number[i] = rand();

}

while (1)

{

int choice;

int time1, time2;

cout << endl << "请选择排序的算法： ";

while (1)

{

cin >> choice;

if (cin.fail() || (choice <= 0 || choice > 9))

{

cout << "输入错误，请重新输入:" << endl;

cin.clear();

cin.ignore(1000, '\n');

}

else

break;

}

switch (choice)

{

case 1:

copy(true\_number, number, num);

BubbleSort(number, num);

break;

case 2:

copy(true\_number, number, num);

ChooseSort(number, num);

break;

case 3:

copy(true\_number, number, num);

insert(number, num);

break;

case 4:

copy(true\_number, number, num);

Shill(number, num);

break;

case 5:

compare\_step = 0;

exchange\_step = 0;

copy(true\_number, number, num);

time1 = clock();

QuickSort(number, 0, num - 1);

time2 = clock();

/\*for (int i = 0; i < num; i++)

cout << number[i] << " ";

cout << endl;\*/

cout << "快速排序所用时间: " << (time2 - time1) / CLOCKS\_PER\_SEC << "s" << endl;

cout << "快速排序比较次数: " << compare\_step << endl;

cout << "快速排序交换次数: " << exchange\_step << endl;

break;

case 6:

compare\_step = 0;

exchange\_step = 0;

copy(true\_number, number, num);

HeapSort(number, num);

break;

case 7:

compare\_step = 0;

exchange\_step = 0;

copy(true\_number, number, num);

Mergesort(number, num);

break;

case 8:

copy(true\_number, number, num);

radixsort(number, num);

break;

case 9:

return 0;

default:

cout << "错误输入！" << endl;

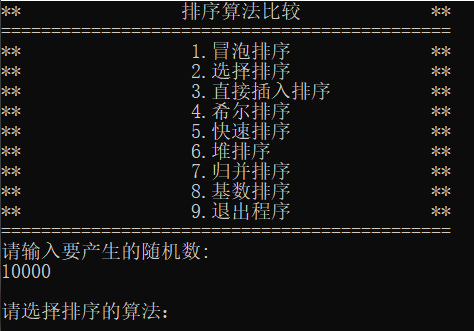
break;

}

}

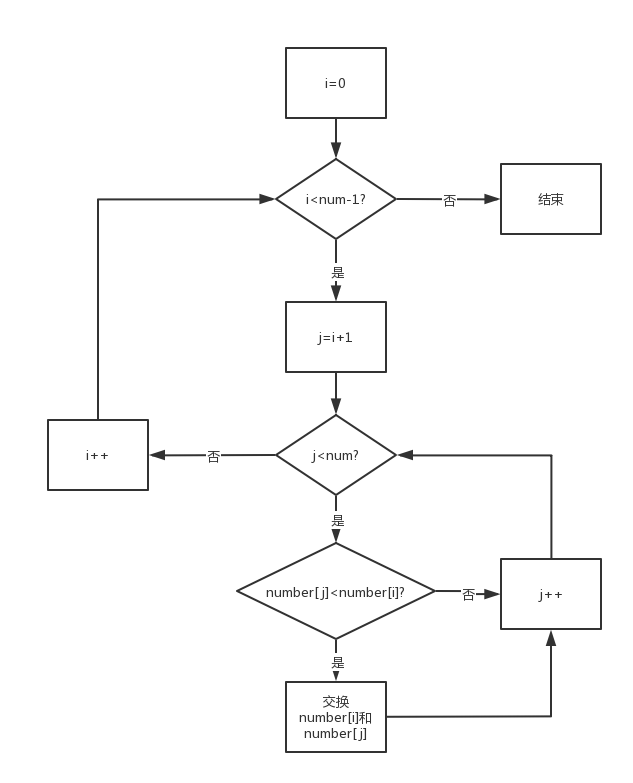
}

### 3.1.3程序截屏示例



## 3.2 冒泡排序

### 3.2.1 冒泡排序功能流程图



### 3.2.2 冒泡排序核心代码

void BubbleSort(int \*number,int num)

{

compare\_step = 0;

int time1 = clock();

for (int i = 0; i < num-1; i++)

{

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

{

compare\_step++;

if (number[i] > number[j])

{

int menory = number[j];

number[j] = number[i];

number[i] = menory;

}

}

}

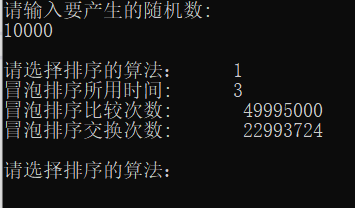
int time2 = clock();

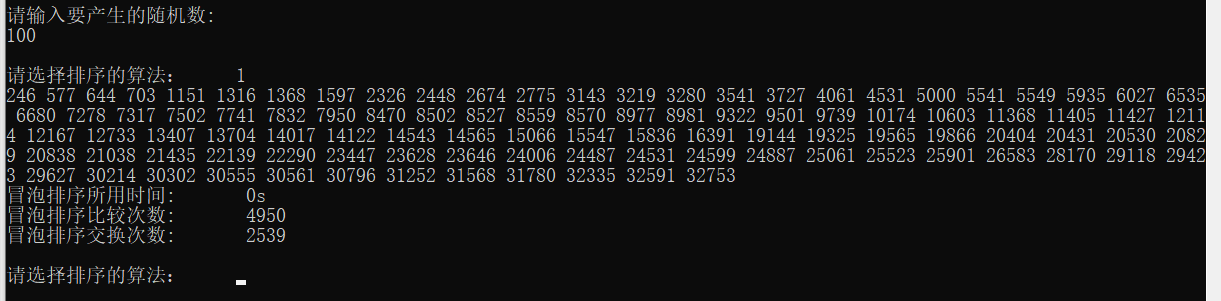
cout << "冒泡排序所用时间: " << (time2 - time1)/ CLOCKS\_PER\_SEC << endl;

cout << "冒泡排序交换次数: " << compare\_step << endl;

}

### 3.2.3冒泡排序截屏示例





### 3.2.4冒泡排序算法分析

#### 空间

空间复杂度为O(1).

#### 时间

最坏情况：序列为逆序状态，则每一轮遍历都需要 n-1 次交换位置，所以时间复杂度为 O(n^2)。

最好情况：序列为正序状态，每一轮遍历不需要交换位置，所以时间复杂度为 O(n)。

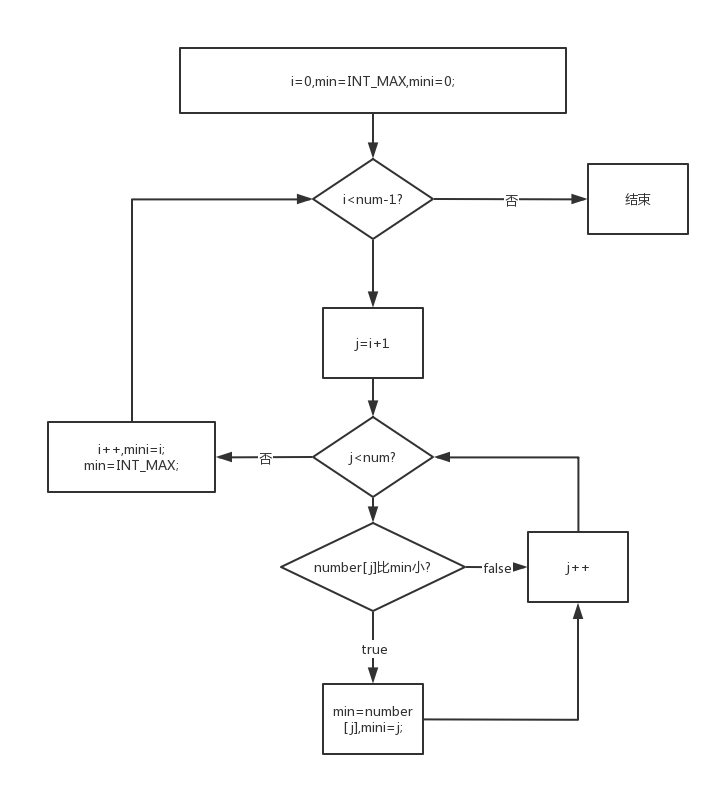
平均情况：每一轮遍历需要 n/2 次交换位置，所以时间复杂度依然为 O(n^2)。

#### 应用优势

在操作数较少时，对时间复杂度要求不高时，使用冒泡排序这个稳定的排序方式比较好，它逻辑简单，不容易出错，写代码速度快。

## 3.3 选择排序

### 3.3.1 选择排序功能流程图



### 3.3.2 选择排序核心代码

void ChooseSort(int number[], int num)

{

compare\_step = 0;

exchange\_step = 0;//置0

int time1 = clock();

int i = 0;

int min=INT\_MAX, mini=0;//min取最大值，mini取0；

for (i = 0; i < num-1 ; i++)//0到num-1

{

for (int j = i+1; j < num ; j++)//i+1到num

{

compare\_step++;

if (number[j] < min)//比min小，min更换，mini记录j

{

min = number[j];

mini = j;

}

}

int t = number[i];

number[i] = number[mini];

number[mini] = t;

//number[t]和最小的number[mini]更换

exchange\_step++;

min = INT\_MAX;

mini = i+1;

}

int time2 = clock();

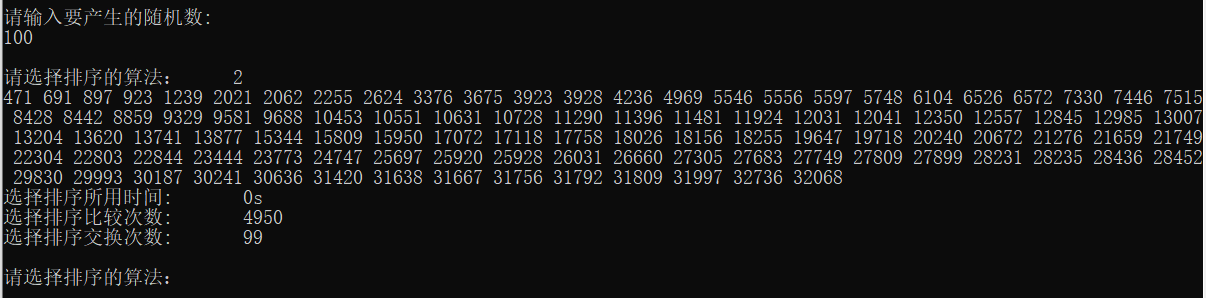
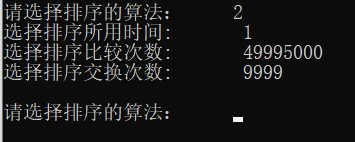
cout << "选择排序所用时间: " << (time2 - time1)/ CLOCKS\_PER\_SEC << endl;

cout << "选择排序比较次数: " << compare\_step << endl;

cout << "选择排序交换次数: " << exchange\_step << endl;

}

### 3.3.3 选择排序截图示例



### 3.3.4选择排序算法分析

#### 空间

空间复杂度O(1)

#### 时间

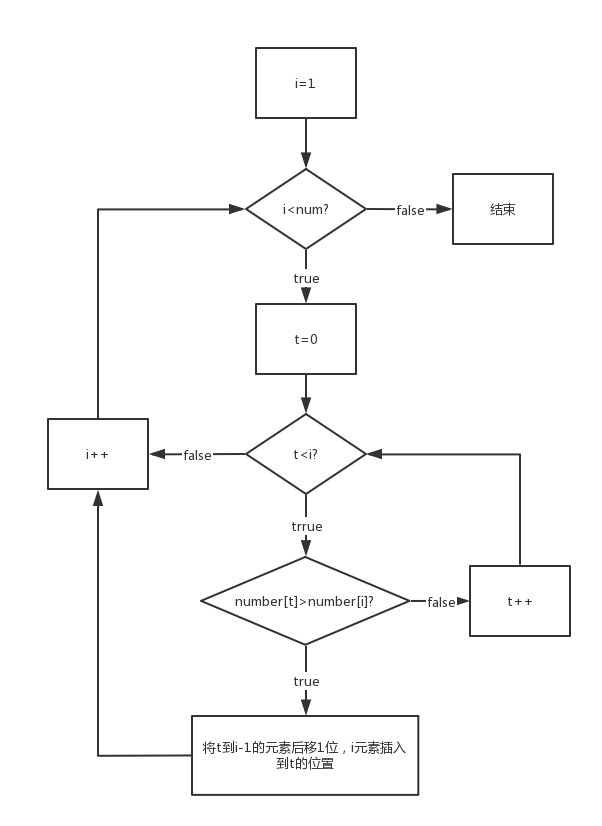
每次都遍历所有元素，时间复杂度为O(n^2).

#### 应用优势

当移动操作花费的时间比比较操作花费的时间大很多时，选择排序算法可以很好地进行移动操作，所以在此情况下可以试用选择排序。

## 3.4直接插入排序

### 3.4.1直接插入排序流程图



### 3.4.2直接插入排序核心代码

void NumInsert(int\* number, int begin, int end)//将begin到end-1的元素后移以为，end的元素插入到begin处

{

int num = number[end];

for (; end > begin; end--)

{

exchange\_step++;

number[end] = number[end - 1];

}

number[begin] = num;

}

void insert(int\* number, int num)

{

compare\_step = 0;

exchange\_step = 0;//置0

int time1 = clock();

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

{

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

{

compare\_step++;

if (number[t] > number[i])

{

NumInsert(number, t, i);//插入

break;

}

}

}

int time2 = clock();

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

cout << " " << number[i] << " ";

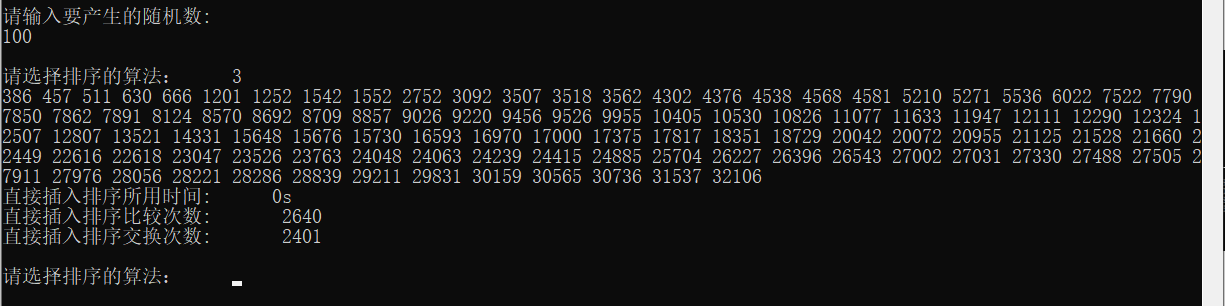
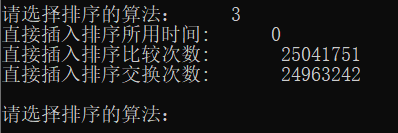
cout << "直接插入排序所用时间: " << (time2 - time1) / CLOCKS\_PER\_SEC << endl;

cout << "直接插入排序比较次数: " << compare\_step << endl;

cout << "直接插入排序交换次数: " << exchange\_step << endl;

}

### 3.4.3直接插入排序截屏示例



### 3.4.4直接插入排序算法分析

#### 空间

空间复杂度O(1)

#### 时间

最坏情况：当待排序序列正好为逆序状态，时间复杂度为 O(n^2)。

最好情况：当待排序序列正好为正序状态，时间复杂度为 O(n)。

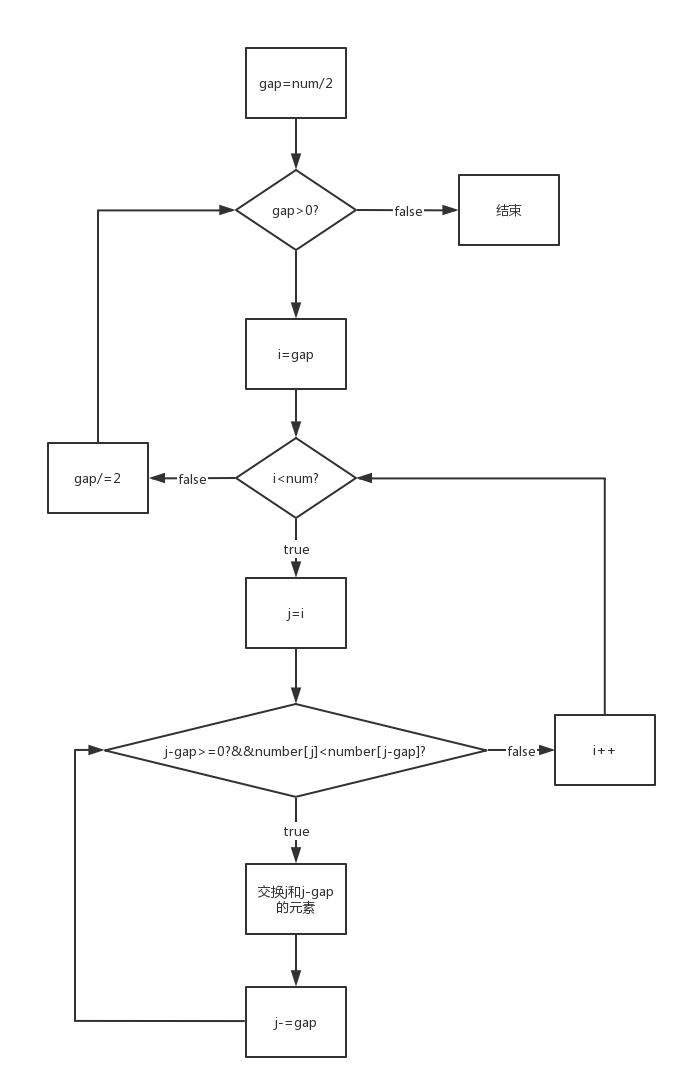
平均情况：当被插入的元素放在已排序的序列中间位置时，总的时间复杂度为 O(n^2)。

#### 应用优势

和冒泡排序一样在序列规模较小以及序列较好的情况下可以使用，此算法一样逻辑简单，空间复杂度小，也很稳定。

## 3.5希尔排序

### 3.5.1希尔排序流程图



### 3.5.2希尔排序核心代码

void Shill(int\* number, int num)

{

exchange\_step = 0;

compare\_step = 0;

int time1 = clock();

for (int gap = num / 2; gap > 0; gap /= 2)

{

//从第gap个元素，逐个对其所在组进行直接插入排序操作

for (int i = gap; i < num; i++)

{

int j = i;

while (j - gap >= 0 && number[j] < number[j - gap])

{

//插入排序采用交换法

swap(number, j, j - gap);

exchange\_step++;

j -= gap;

compare\_step++;

}

if (j - gap >= 0)

compare\_step++;

}

}

int time2 = clock();

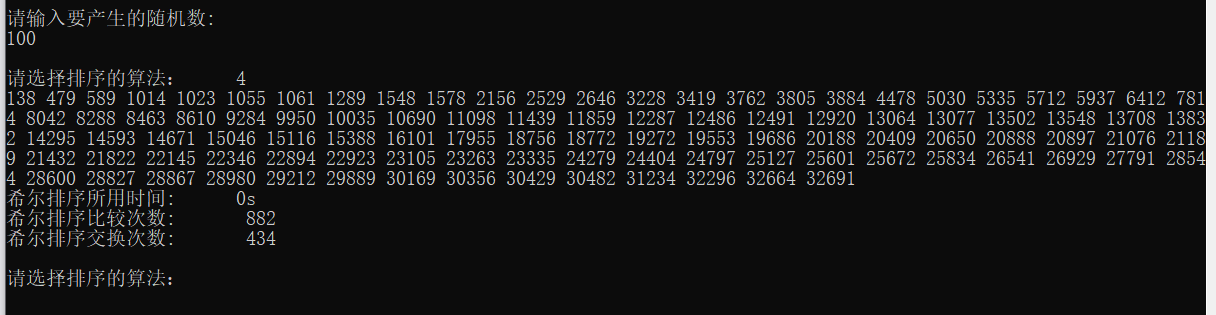
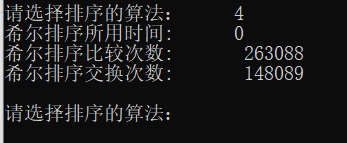
cout << "希尔排序所用时间: " << (time2 - time1) / CLOCKS\_PER\_SEC << endl;

cout << "希尔排序比较次数: " << compare\_step << endl;

cout << "希尔排序交换次数: " << exchange\_step << endl;

}

### 3.5.3希尔排序截屏示例



### 3.5.4希尔排序算法分析

#### 空间

空间复杂度为O(1)

#### 时间

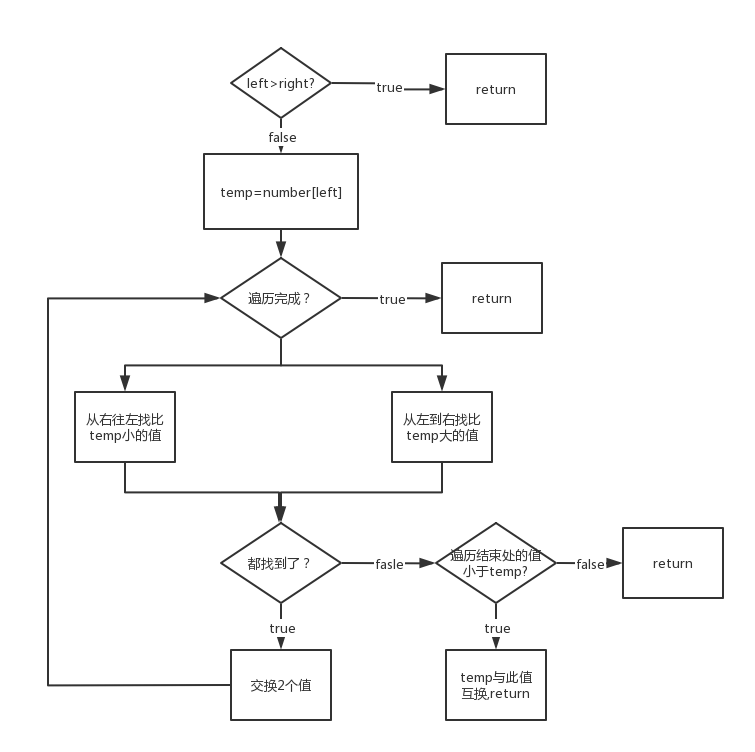
暂时不确定,约为n^1.3到n^2

#### 应用优势

在待排序规模较大、需要较快的排序算法且对稳定性没有要求的场景下可以使用希尔排序，它交换次数很少，所需的额外空间小；但唯一的不足是 gap 的取值和迭代 方法还没有确定的最优解，只能靠经验来设定。

## 3.6快速排序

### 3.6.1快速排序流程图



### 3.6.2快速排序核心代码

void QuickSort(int\* number,int left, int right)

{

if (left > right)

return;

int temp = number[left];//存基准值

int i = left;

int j = right;

while (i != j)

{

while (number[j] >= temp && i < j)//从右往左找

{

j--;

compare\_step++;

}

//找到了一个小于temp的number[j]

while (number[i] <= temp && i < j)

{

compare\_step++;

i++;

}

//找到了一个大于temp的number[i]

//交换位置

if (i < j)

{

exchange\_step++;

int t = number[i];

number[i] = number[j];

number[j] = t;

}

}

//i==j

number[left] = number[i];

number[i] = temp;//基准数放中间

exchange\_step ++;

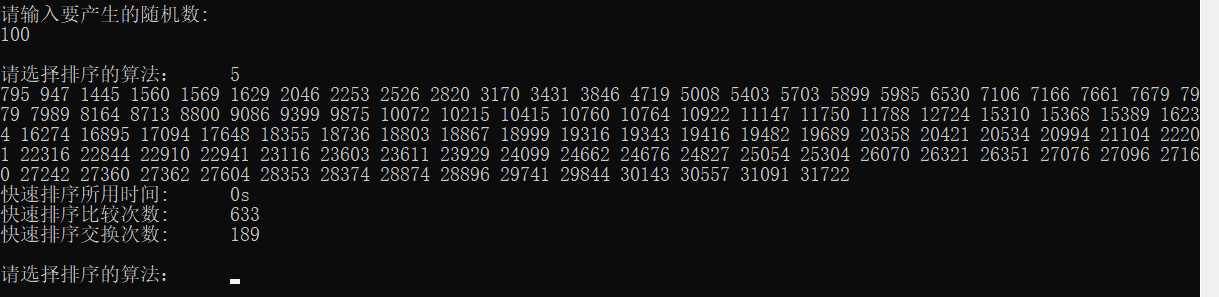
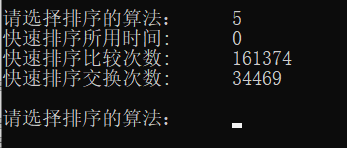
QuickSort(number, left, i - 1);

QuickSort(number, i + 1, right);

return;

}

### 3.6.3快速排序截屏示例



### 3.6.4快速排序算法分析

#### 空间

空间复杂度内部使用空间为O(1)，但递归调用时，递归工作栈要保留一些数据要占用空间，平均的空间复杂度是O(logn)

#### 时间

最坏情况：每一次选取的基准元素都是最大或最小的，时间复杂度为 O(n^2)

最好情况：每一次选取的基准元素都能平分整个序列，时间 复杂度为 O(nlog2n)。

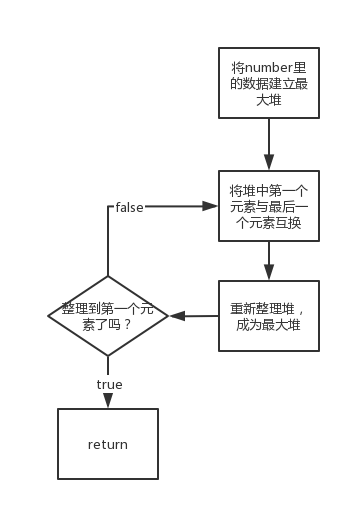
平均情况：平均情况下时间复杂度是 O(nlog2n)。

#### 应用优势

快速排序是目前速度最快的排序之一，在需要较快的排序速度且不要求稳定性的需求下可以使用快速排序。

## 3.7堆排序

### 3.7.1堆排序流程图



### 3.7.2堆排序核心代码

void HeapAdjust(int\* number, int i, int length)

{

int temp = number[i];//取出当前元素

for (int k = 2 \* i + 1; k < length; k = 2 \* k + 1)

{

compare\_step++;

if (number[k] < number[k + 1] && k + 1 < length)//右节点大

{

k++;

}

compare\_step++;

if (temp < number[k])//父节点小，子节点赋值给父节点

{

exchange\_step++;

number[i] = number[k];

i = k;

}

else

break;

}

number[i] = temp;

}

void HeapSwap(int\* number, int i, int j)

{

exchange\_step++;

int temp = number[j];

number[j] = number[i];

number[i] = temp;

}

void HeapSort(int\* number, int num)

{

int time1 = clock();

for (int i = num / 2 - 1; i >= 0; i--)

{

HeapAdjust(number, i, num);

}//建立堆

for (int j = num - 1; j >= 0; j--)

{

HeapSwap(number,0,j);

HeapAdjust(number, 0, j);

}

int time2 = clock();

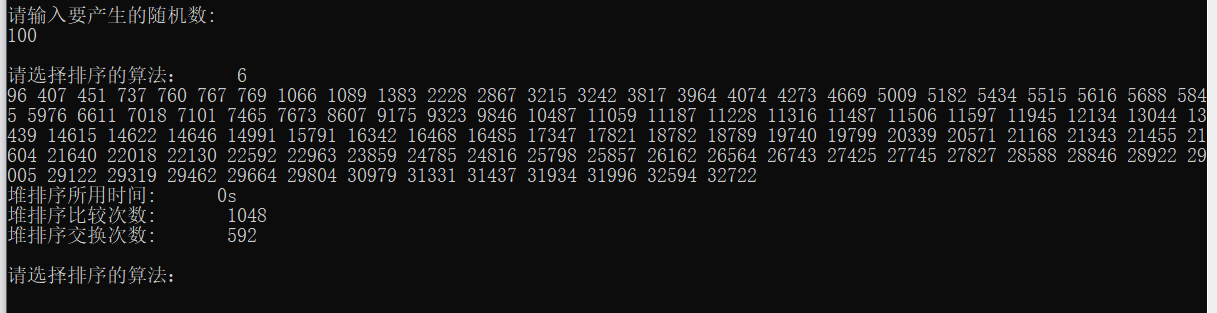
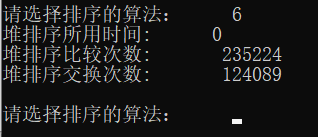
cout << "堆排序所用时间: " << (time2 - time1) / CLOCKS\_PER\_SEC << endl;

cout << "堆排序比较次数: " << compare\_step << endl;

cout << "堆排序交换次数: " << exchange\_step << endl;

}

### 3.7.3堆排序截屏示例



### 3.7.4堆排序算法分析

#### 空间

空间复杂度O(1)。

#### 时间

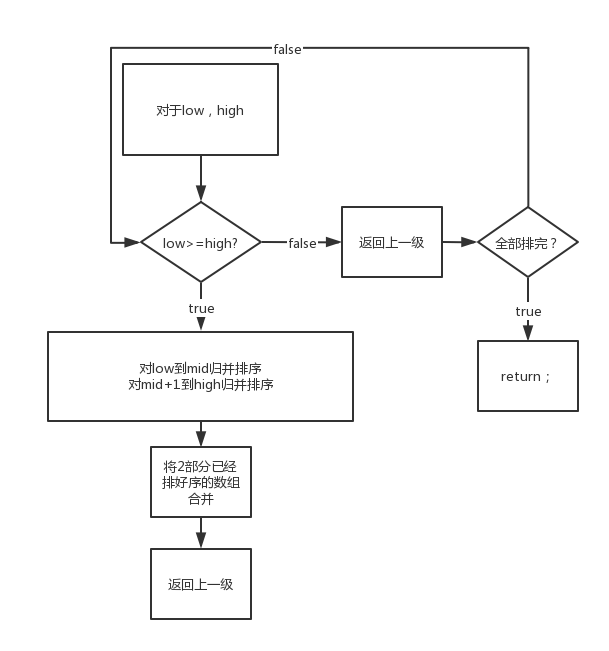
时间复杂度为O(nlog2n)。

#### 应用优势

较大的序列，时间要求较快，对稳定性要求不高时，可以使用堆排序。

## 3.8归并排序

### 3.8.1归并排序流程图



### 3.8.2归并排序核心代码

void Merge(int\* number, int\* list, int low, int high)

{

int s1 = low, s2 = high, t = low, k;

if (low >= high)

return;

int mid = (low + high) / 2;

Merge(number, list, low, mid);

Merge(number, list, mid + 1, high);

for (k = low; k <= mid; k++) {

list[k] = number[k];

}

for (k = mid + 1; k <= high; k++) {

list[high + mid - k + 1] = number[k];//list里是从low到mid的正序，mid+1到high的逆序

}

while (t <= high) {

exchange\_step++;

compare\_step++;

if (list[s1] < list[s2])

number[t++] = list[s1++];

else

number[t++] = list[s2--];

//将list里的数据存进number里

}

return;

}

void Mergesort(int\* number, int num)

{

int\* list = new int[100000];

int time1 = clock();

Merge(number, list, 0, num - 1);

int time2 = clock();

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

cout << number[i] << " ";

cout << endl;

cout << "归并排序所用时间: " << (time2 - time1) / CLOCKS\_PER\_SEC<< "s" << endl;

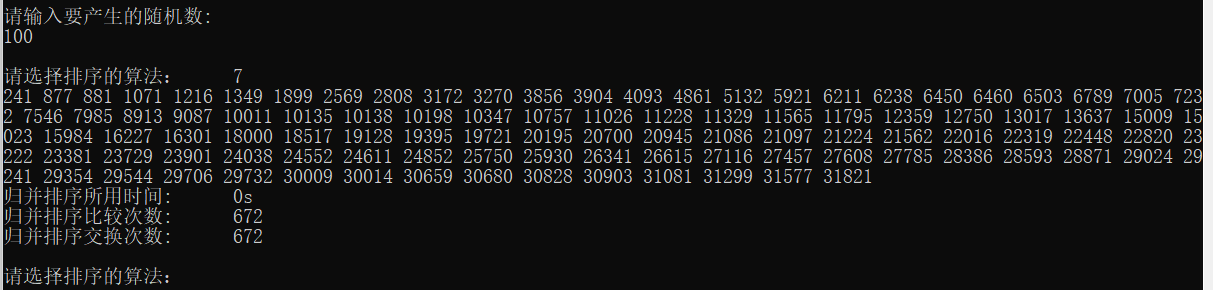
cout << "归并排序比较次数: " << compare\_step << endl;

cout << "归并排序交换次数: " << exchange\_step << endl;

delete[]list;

}

### 3.8.3归并排序截屏示例



### 3.8.4归并排序算法分析

#### 空间

空间使用了一个辅助数组，空间复杂度为O(1).

#### 时间

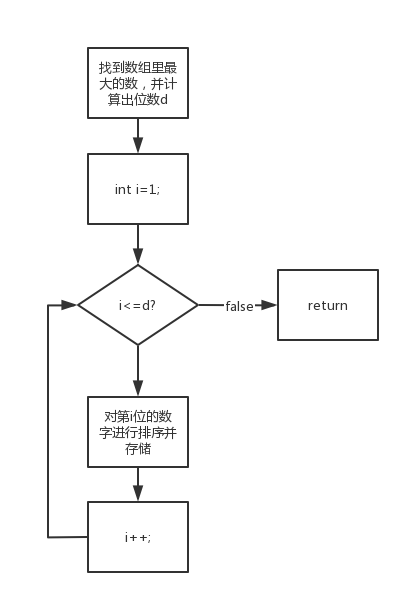
时间复杂度为O(nlogn).

#### 应用优势

归并排序速度很快，和快速排序希尔排序相当，相同的，在对大量数据，需要快速而不需要较高稳定性的情况下可以使用归并排序。

## 3.9基数排序

### 3.9.1基数排序流程图



### 3.9.2基数排序核心代码

int maxbit(int\* number, int num) //求数据的最大位数

{

int maxData = number[0]; ///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，稍微优化点。

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

{

if (maxData < number[i])

maxData = number[i];

}

int d = 1;

int p = 10;

while (maxData >= p)

{

maxData /= 10;

++d;

}

return d;

}

void radixsort(int number[], int num) //基数排序

{

int time1 = clock();

int d = maxbit(number, num);//求最高位数

int\* tmp = new int[num];

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 = 0; j < num; j++)

{

k = (number[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 = num - 1; j >= 0; j--) //将所有桶中记录依次收集到tmp中

{

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

tmp[count[k] - 1] = number[j];

count[k]--;

}

for (j = 0; j < num; j++) //将临时数组的内容复制到number中

number[j] = tmp[j];

radix = radix \* 10;

}

int time2 = clock();

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

cout << number[i] << " ";

cout << endl;

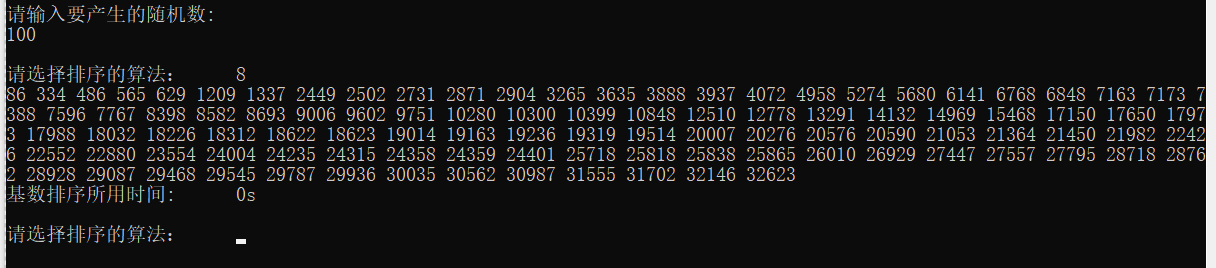
cout << "基数排序所用时间: " << (time2 - time1) / CLOCKS\_PER\_SEC << "s" << endl;

delete[]tmp;

delete[]count;

}

### 3.9.3基数排序截屏示例



### 3.9.4基数排序算法分析

#### 空间

空间复杂度O(n)

#### 时间

时间复杂度为O(n+d)，d为最大元素的位数

#### 应用优势

基数排序可以进行大量数据的快速排序，且具有稳定性。