8种排序算法的比较案例

作 者 姓 名： 李翠琪

学 号： 1751022

指 导 教 师： 张颖 \_\_

学院、 专业： 软件工程 \_\_

同济大学

Tongji University

目录

[1 分析 1](#_Toc23428942)

[1.1 项目简介 1](#_Toc23428943)

[1.2 功能分析 1](#_Toc23428944)

[2 设计 2](#_Toc23428945)

[2.1 排序算法设计 2](#_Toc23428946)

[2.2 辅助函数设计 2](#_Toc23428947)

[2.3 主程序设计 2](#_Toc23428948)

[3 实现 3](#_Toc23428949)

[3.1 八个排序方法的实现 3](#_Toc23428950)

[3.1.1 冒泡排序 3](#_Toc23428951)

[3.1.2 选择排序 4](#_Toc23428953)

[3.1.3 插入排序 4](#_Toc23428954)

[3.1.4 希尔排序 5](#_Toc23428955)

[3.1.5 快速排序 6](#_Toc23428956)

[3.1.6 堆排序 7](#_Toc23428957)

[3.1.7 归并排序 8](#_Toc23428958)

[3.1.8 基数排序 10](#_Toc23428959)

[3.2 辅助函数功能的实现 11](#_Toc23428960)

[3.2.1 exch函数 11](#_Toc23428961)

[3.2.2 isSorted函数 12](#_Toc23428962)

[3.3 总体系统的实现 12](#_Toc23428963)

[4 测试 13](#_Toc23428964)

# 1 分析

## 1.1 项目简介

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。

## 1.2 功能分析

首先,得在程序中实现了八种排序算法,其次,要有两个辅助函数,分别计算排序前后所花费的时间与排序的比较次数.

最后,在主程序中,给用户提供界面,选择某种排序算法后,展示排序所需要的时间和比较次数.用户可以不断选择排序算法直到退出.用户也可以任意选择想排序的规模.

# 2 设计

## 2.1 排序算法设计

八种排序算法之中,都会传入一个引用参数exchangeTime代表交换次数,当交换时让exchangeTime值+1.每个排序算法都会将原来的随机数组复制一遍,以免把原有的随机数组更改了.

同时为了规范,数组在进行交换数据时,都会调用同一个辅助函数exch进行交换,控制了变量.为了确保已经排序,也会调用isSorted确保已经排序.

## 2.2 辅助函数设计

首先,在SortCompare类中,构造函数将会为它的成员变量vec赋予N个随机数,然后在test类中,根据用户输入的参数,选择适当的算法进行排序.且在排序的前后计时,两者相减,得到了该排序算法的用时.

然后,exch()通过传入数组与要交换的两个位置把元素交换,isSorted()通过一次循环比较后方元素是否大于前方元素来确保排序,否则返回false.

## 2.3 主程序设计

首先,用一个数组将数字与排序算法的名字关联起来,便于查看.输出窗口打印出表格以给用户进行提示,让用户输入排序的规模N.随后进入循环,直到用户按9退出程序.循环中,根据用户输入,调用相应的函数,执行相应的操作, 完成排序,并打印排序的时间与交换次数.将交换次数清零,进行下一次选择,直到用户退出.

# 3 实现

## 3.1 八个排序方法的实现

### 3.1.1 冒泡排序

### 两遍循环,外层循环每次将最小的元素排定,直到整个数组有序。内层循环从数组最后一个元素分别与前一个元素对比，如果小于前一个元素，将两者交换，并且使交换次数+1，一直对比到排定的元素之前，因为排定的元素一定是小于后面的所有的元素的。

值得注意的是，每个排序都通过将原数组复制，并将复制后的数组temp进行排序来进行测试，并且每次排序完，都调用测试函数isSorted测试是否排序，所有排序都用了这个模式，所以之后不再赘述。

void bubbleSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
 //每次都把最小的排定  
 for (int i = 1; i < temp.size(); ++i)  
 {  
 for (int j = temp.size() - 1; j >= i; --j)  
 {  
 if (temp[j - 1] > temp[j])  
 {  
 exch(temp, j, j - 1);  
 ++exchangeTime;  
 }  
 }  
 }  
 if (!isSorted(temp))  
 {  
 cout << "没排序!" << endl;  
 }  
}

### 3.1.2 选择排序

同样是两层循环，外层循环先让i做为最小的元素，并且通过右层循环找出数组右边最小元素的索引，然后再交换i与内层循环找到的最小元素min，并且让交换次数+1。

void chooseSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
 for (int i = 0; i < temp.size() - 1; ++i)  
 {  
 int min = i;  
 for (int j = i + 1; j < temp.size(); ++j)  
 {  
 if (temp[min] > temp[j])  
 {  
 min = j;  
 }  
 }  
 exch(temp, min, i);  
 ++exchangeTime;  
 }  
 if (!isSorted(temp))  
 {  
 cout << "没排序!" << endl;  
 }  
}

### 3.1.3 插入排序

两层循环，外层循环从一开始到数组末尾。内层循环不断的将该元素j与它前面的一个元素j-1的值进行对比，如果小与索引j-1的值则交换，直到j-1<0了或者它的值大于索引j-1的值。算法本质就是左边一开始是一个元素，然后不断的让数组后面的元素插入到合适的位置，直到所有元素都插入，而它们也就是一个有序的数组了。

void insertSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
 for (int i = 1; i < temp.size(); ++i)  
 {  
 for (int j = i; j - 1 >= 0 && temp[j] < temp[j - 1]; --j)  
 {  
 exch(temp, j - 1, j);  
 ++exchangeTime;  
 }  
 }  
 if (!isSorted(temp))  
 {  
 cout << "没排序!" << endl;  
 }  
}

### 3.1.4 希尔排序

本质上希尔排序就是插入排序的改进。先用h不断\*3+1，直到到数组大小的三分之一，然后和插入排序的循环类似，只不过外层循环是从h开始，且内层循环每一次对比的都是j和j-h的元素，这样一次排序就形成了h有序数组，即每相隔h位置的元素形成了一个有序数组，然后最外面的循环负责不断的缩小h的值，使得形成的h越来越小，而当h=1时，即为插入排序，并且此次排序结束后循环终止。

之所以要这么麻烦，是为了刚刚开始时，h大概是数组的三分之一左右，然后每一次交换都能让相对很小的元素放在很前面，不必像是插入排序一样，如果碰到最小的元素，就要一个个的比较交换，直到它到最左边的位置。

void shellSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
  
  
 int h = 1;  
 while (h < temp.size() / 3)  
 {  
 h = 3 \* h + 1;  
 }  
  
 while (h >= 1)  
 {  
 for (int i = h; i < temp.size(); ++i)  
 {  
 for (int j = i; j - h >= 0 && temp[j] < temp[j - h]; j -= h)  
 {  
 exch(temp, j - h, j);  
 ++exchangeTime;  
 }  
 }  
 h /= 3;  
 }

### 3.1.5 快速排序

用了额外的两个辅助函数，每次排序，先调用辅助函数partition，它将返回一个索引j，且保证了j的左边元素都小于它，而它的右边元素都大于或等于它。然后继续递归地将左边的数组排序，将右边的数组排序，直到只有一个元素时返回。因为每次调用，j的位置都是排定的，所以通过归纳法可知最后数组一定排序好了。

Partition函数具体实现则是：先将传入的第一个元素（传入的元素不一定就是数组的首元素，而是要排序的子数组的第一个元素，用lo标定）定为v，用i，j指向该要排序的数组，然后i不断的向右扫描，j不断的向左扫描，如果i找到了大于等于v的元素，j找到了小于等于它的元素，则将他们两个交换。重复扫描直到i>=j或者i到了数组末尾，或者j到达数组开头。此时可知索引j的元素肯定是小于v的，而i一定是大于v的，此时只要交换索引j和首元素，再返回索引j，即实现了j左边都是小于它的而右边都是大于等于它的。

void quickSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
  
 //因为本来就是乱的,所以不用打乱了  
 quickSort(temp, 0, temp.size() - 1, exchangeTime);  
  
 if (!isSorted(temp))  
 {  
 cout << "没排序!" << endl;  
 }  
}

void quickSort(vector<int> &temp, int lo, int hi, long long &exchangeTime)  
{  
 if (hi <= lo)  
 {  
 return;  
 }  
 int j = partition(temp, lo, hi, exchangeTime);  
 quickSort(temp, lo, j - 1, exchangeTime);  
 quickSort(temp, j + 1, hi, exchangeTime);  
}

int partition(vector<int> &temp, int lo, int hi, long long &exchangeTime)  
{  
 int i = lo, j = hi + 1;  
 int v = temp[lo];  
 while (true)  
 {  
 while (temp[++i] < v) if (i == hi) break;  
 while (v < temp[--j]) if (j == lo) break;  
 if (i >= j)  
 break;  
 exch(temp, i, j);  
 ++exchangeTime;  
 }  
 exch(temp, lo, j);  
 ++exchangeTime;  
 return j;  
}

### 3.1.6 堆排序

首先构造一个堆，从数组长度/2的位置直到第一个元素进行构建，因为数组长度/2到数组末尾的可以直接作为堆底，不用构建了，所谓构建，就是对每一个索引i调用sink函数，直觉上来说就是把大的元素放上面，小的放下面。然后排序阶段，将数组最后一个元素与第一个交换，即将堆最大的元素放再数组末端，然后对数组首元素进行sink操作，来保证堆的有序性。N--，不断这样操作直到N==1，意味着数组已经排序。

Sink函数主要是将元素下沉，即通过对比该元素k与它的下一层2\*k与2\*k+1中较大的一个，如果k小于它的下一层，则交换，且让它等于2\*k或者2\*k+1，不断比较交换，直到到堆末尾或者它大于它下一层的较大的元素。

void heapSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
 int N = temp.size();  
 temp.insert(temp.begin(), -1);//temp[0]不用  
 //构造堆  
 for (int k = N/2; k >= 1 ; --k)  
 {  
 sink(temp, k, N,exchangeTime);  
 }  
 //排序  
 while(N>1)  
 {  
 exch(temp, 1, N--);  
 ++exchangeTime;  
 sink(temp, 1, N,exchangeTime);  
 }  
  
  
 if (!isSorted(temp))  
 {  
 cout << "没排序!" << endl;  
 }  
}

void sink(vector<int> &temp, int k, int N, long long &exchangeTime)  
{  
 while(2\*k<=N)  
 {  
 int j = 2 \* k;  
 if(j+1<=N && temp[j+1]>temp[j])  
 {  
 ++j;  
 }  
 if(temp[k]>temp[j])  
 {  
 break;  
 }  
 exch(temp, k, j);  
 k = j;  
 ++exchangeTime;  
 }  
}

### 3.1.7 归并排序

总的来说，是用了分治法的思路，即，首先将左边一半数组排序，右一半数组排序。然后将两个排序的数组进行归并。而将左右两边数组排序是采用递归的思路，例如将左边的一半数组排序，则为将左边的数组的左一半数组排序，右一半数组排序，再将之合并。归并到最后的结果就是，如果有一个子数组只有一个或者没有元素就返回，这时算是已经排序了的。然后将之不断合并。一直到整个数组合并为一个有序数组。

合并两个有序算法调用了一个辅助函数merge，思路很简单，即为用两个索引分别指向他们两个数组的开头，然后比较这两个索引的元素，存储结果的数组存储较小的那个元素，然后对于的索引+1，一直对比到其中的一个索引指向了那个数组的尽头。此时，结果数组直接全部存储剩余那个数组的剩余元素即可。

void mergeSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
  
 vector<int> aux(vec.size());//辅助数组  
  
 mergeSort(temp, aux, 0, temp.size() - 1, exchangeTime);  
  
 if (!isSorted(temp))  
 {  
 cout << "没排序!" << endl;  
 }  
}

void mergeSort(vector<int> &temp, vector<int> &aux, int lo, int hi, long long &exchangeTime)  
{  
 if (hi <= lo)  
 {  
 return;  
 }  
 int mid = lo + (hi - lo) / 2;  
 mergeSort(temp, aux, lo, mid, exchangeTime);  
 mergeSort(temp, aux, mid + 1, hi, exchangeTime);  
  
 merge(temp, aux, lo, mid, hi, exchangeTime);  
}

void merge(vector<int> &temp, vector<int> &aux, int lo, int mid, int hi, long long &exchangeTime)  
{  
 //mid左右两边已经拍好了序  
 int i = lo, j = mid + 1;  
 for (int k = lo; k <= hi; ++k)  
 {  
 aux[k] = temp[k];  
 }  
 for (int k = lo; k <= hi; ++k)  
 {  
 //上面两个不进行比较  
 if (i > mid) temp[k] = aux[j++];  
 else if (j > hi) temp[k] = aux[i++];  
 else if (aux[j] < aux[i])  
 {  
 temp[k] = aux[j++];  
 ++exchangeTime;  
 }  
 else  
 {  
 temp[k] = aux[i++];  
 ++exchangeTime;  
 }  
 }  
}

### 3.1.8 基数排序

本质上是低位优先的字符串排序，所以首先得到数组的元素中最大元素的位数。然后对每一位进行一次排序，因为如果前一位的不同，则它们肯定能排序，而前一位的相同，它是稳定的放入，而后一位已经排序好了，所以可以直到从低位往高位不断排序直到最后一定能排序好。

外层循环对每一位进行排序，而排序具体实现是：首先统计每个元素在这一位中0-9出现的个数，然后通过刚刚统计的元素个数，计算得到每一个元素放入辅助数组的开始的索引，然后对于每一个元素放入辅助数组相应的位置，就是通过查看元素这一位的值，在辅助数组找到相应的索引位置放入，并让该索引位置+1。直到把所有元素放入，即为将这一位的所有元素进行排序了。

void radixSort(long long &exchangeTime)  
{  
 vector<int> temp(vec);  
 vector<int> aux(temp);  
  
 int N = temp.size();  
 int R = 10;//一共只有0-9  
 int digit = getDigit(temp);  
 int base = 1;//目的是得到一个数的某一位的数,如得到123中的2  
 //0表示个位  
 for (int d = 0; d < digit; ++d)  
 {  
 vector<int> count(R + 1);//计算出现的频率  
 for (int i = 0; i < N; ++i)  
 {  
 //就算有一些的位数不够也没有关系,因为这样会是0,还是会排序  
 int index = temp[i] / base % 10;//第一次得到是个位,第二次是十位..  
 count[index + 1]++;  
 }  
 for (int r = 0; r < R; ++r)//将频率转换为索引  
 {  
 count[r + 1] += count[r];  
 }  
 for (int i = 0; i < N; ++i)//将元素分类  
 {  
 int index = temp[i] / base % 10;  
 aux[count[index]++] = temp[i];  
 }  
 for (int i = 0; i < N; ++i)//回写  
 {  
 temp[i] = aux[i];  
 }  
 base \*= 10;//不断处理高位  
 }  
 if (!isSorted(temp))  
 {  
 cout << "没排序!" << endl;  
 }  
}

int getDigit(const vector<int> & temp)  
{  
 int digit = 1;  
 int base = 10;  
 for (int i = 0; i < temp.size(); ++i)  
 {  
 while(base<=temp[i])  
 {  
 base \*= 10;  
 ++digit;  
 }  
 }  
 return digit;  
}

## 3.2 辅助函数功能的实现

### 3.2.1 exch函数

void exch(vector<int> &cur, int i, int j)  
{  
 int temp = cur[i];  
 cur[i] = cur[j];  
 cur[j] = temp;  
}

### 3.2.2 isSorted函数

bool isSorted(vector<int> &cur)  
{  
 for (int i = 0; i < cur.size() - 1; ++i)  
 if (cur[i + 1] < cur[i])  
 return false;  
 return true;  
}

## 3.3 总体系统的实现

进入一个循环,根据用户输入,调用不同的排序算法进行排序测试,并且打印测试时间与比较次数,随后重置排序时间并且进行下一次比较。

核心代码：

while(true)  
{  
 cout << "请选择排序算法:";  
 cin >> whichSort;  
 if(whichSort==9)  
 {  
 break;  
 }  
 sortCompare.test(whichSort, sortTime, exchangeTime);  
 cout<<changeIndex[whichSort]<<"所用时间:"<<sortTime<<"s"<<endl;  
 //归并排序不进行两两交换,所以看比较次数  
 if(whichSort==7)  
 cout<<changeIndex[whichSort]<<"比较次数："<<exchangeTime<<endl;  
 else  
 cout<<changeIndex[whichSort]<<"交换次数："<<exchangeTime<<endl;  
 //重置  
 sortTime = 0.0;  
 exchangeTime = 0;  
}

# 4 测试

