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

作 者 姓 名： 崔鑫宇

学 号： 1853444

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

# 1 分析

## 背景分析

排序算法是我们专业学习中的一种十分基础而又十分重要的算法，好好研究研究各种基础的算法对我们的学习是十分有必要的，尤其排序算法是我们日常生活中比较常用的算法之一，因此各种排序算法的比较尤其值得我们重视。

这次我们要研究的8种排序算法分别是冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序以及基数排序。

## 1.2 功能分析

由于是想要比较各种排序算法因此我们首先要写出上述的8种排序算法，并且统计他们每种的排序所花费的时间以及每次排序的元素交换次数，其中生成随机数的个数应该要用户自己去指定。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，该系统要求代码内执行8种排序算法其中的堆排序需要用到堆这一数据结构。

## 2.2 类结构设计

由于本程序中的堆排序需要用到堆这一数据结构，因此要设计堆类（Heap）

## 2.3 成员与操作设计

**堆类（Heap）**

**Public：**

Heap(vector<int>& A, int size, long long& swap\_times);

~Heap();

void OrderDown(int start, int end, long long& swap\_times);

int length();

bool isEmpty();

**Private:**

int\* heap;//堆的指针

int Size;//堆的大小

**主函数：(main)**

bool isPositive(string s)

void BubbleSort(vector<int>& B, int n, long long& compare\_times)

void SelectionSort(vector<int>& B, int n, long long& swap\_times)

void InsertionSort(vector<int>& B, int n, long long& swap\_times)

void ShellSort(vector<int>& B, int n, long long& swap\_times)

void QuickSort(vector<int>& B, const int left, const int right, long long& swap\_times)

void HeapSort(vector<int>& B, int size, long long& swap\_times)

void merge(vector<int>& L1, vector<int>& L2, const int left, const int mid, const int right, long long& compare\_times)

void MergeSort(vector<int>& L1, vector<int>& L2, int left, int right, long long& compare\_times)

void RadixSort(vector<int>& B, int size, long long& swap\_times) {

## 2.4 系统设计

首先先输出菜单，然后由用户指定生成随机数的个数，经检查如果用户输入随机数个数为正整数且小于1000000，则开始根据用户所输入的的操作数来执行某种相应的排序算法。

# 3 实现（以10000个随机数为例）

## 3.1 冒泡排序的实现

3.1.1 冒泡排序的核心代码

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

for (int j = 0; j < n - 1 - i; j++) {

compare\_times++;

if (vec[j] > vec[j + 1]) {

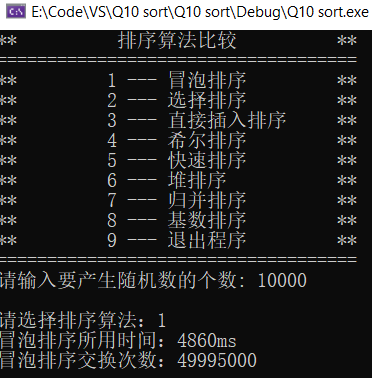
swap(vec, j, j + 1);

}

}

}

3.1.2 冒泡排序部分截屏示例



3.2 选择排序的实现

3.2.1 选择排序的核心代码

for (int i = 0; i < n; i++) {

int min = i;

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

swap\_times++;

if (vec[j] < vec[min]) {

min = j;

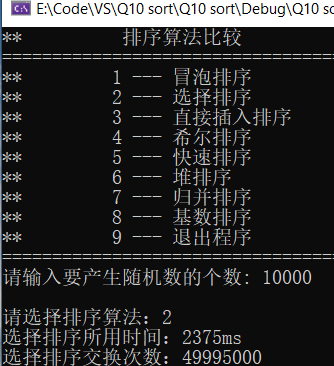
}

}

swap(vec, i, min);

}

3.2.2 选择排序部分截屏示例



## 

## 3.3 直接插入排序的实现

## 

## 3.3.1 直接插入排序的核心代码

int temp;

for (int i = 1; i < n; i++) {

temp = vec[i];

for (int j = i - 1; j >= 0; j--) {

if (j == 0) {

vec[j] = temp;

}

if (temp < vec[j]) {

vec[j + 1] = vec[j];

swap\_times++;

}

else {

vec[j + 1] = temp;

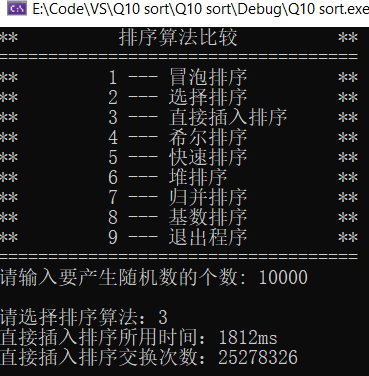
break;

}

}

}

## 3.3.2 直接插入排序截屏示例



## 3.4 希尔排序的实现

3.4.1 希尔排序的核心代码

int gap = (int)(n / 2);

while (gap) {

for (int i = gap; i < n; i++) {

for (int j = i; j >= gap && vec[j] < vec[j - gap]; j -= gap) {

swap(vec, j, j - gap);

swap\_times++;

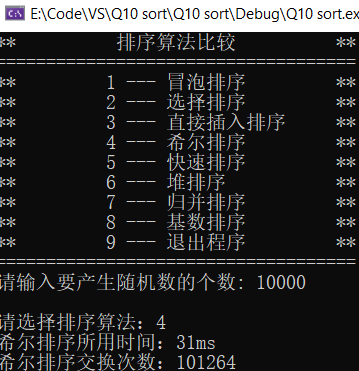
}

}

gap = gap == 2 ? 1 : (int)(gap / 2.2);

}

3.4.2 希尔排序截屏示例



## 

## 3.5 快速排序的实现

3.5.1 快速排序的核心代码

if (left < right) {

int pivotPos = left;

int pivot = vec[left];

for (int i = left + 1; i <= right; i++) {

swap\_times++;

if (vec[i] < pivot) {

pivotPos++;

if (pivotPos != i) {

swap(vec, pivotPos, i);

}

}

}

vec[left] = vec[pivotPos];

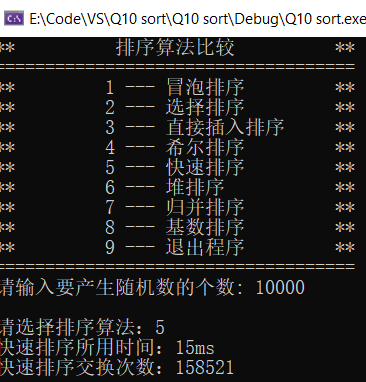
vec[pivotPos] = pivot;

QuickSort(vec, left, pivotPos - 1, swap\_times);

QuickSort(vec, pivotPos + 1, right, swap\_times);

}

3.5.2 快速排序截屏示例

****

## 3.6 堆排序的实现

3.6.1 堆排序的核心代码

Heap maxHeap = Heap(vec, size, swap\_times);

for (int i = size - 1; i > 0; i--) {

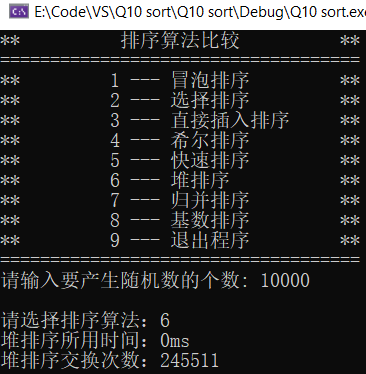
maxHeap.swap(0, i);

swap\_times++;

maxHeap.OrderDown(0, i - 1, swap\_times);

}

3.6.2 堆排序截屏示例



## 3.7 归并排序的实现

3.7.1 归并排序的核心代码

//归并操作

void merge(vector<int>& L1, vector<int>& L2, const int left, const int mid, const int right, long long& compare\_times) {

for (int i = left; i <= right; i++) {

L2[i] = L1[i];

}

int s1 = left;

int s2 = mid + 1;

int t = left;

while (s1 <= mid && s2 <= right) {

compare\_times++;

if (L2[s1] <= L2[s2]) {

L1[t++] = L2[s1++];

}

else {

L1[t++] = L2[s2++];

}

}

while (s1 <= mid) {

compare\_times++;

L1[t++] = L2[s1++];

}

while (s2 <= right) {

compare\_times++;

L1[t++] = L2[s2++];

}

}

//归并排序

void MergeSort(vector<int>& L1, vector<int>& L2, int left, int right, long long& compare\_times) {

if (left < right) {

int mid = (left + right) / 2;

MergeSort(L1, L2, left, mid, compare\_times);

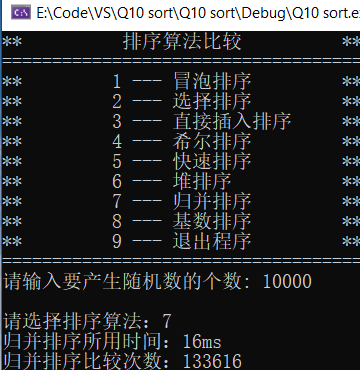
MergeSort(L1, L2, mid + 1, right, compare\_times);

merge(L1, L2, left, mid, right, compare\_times);

}

}

3.7.2 归并排序截屏示例



## 3.8 基数排序的实现

3.8.1 基数排序的核心代码

int d = MaxBit(vec, size);

int\* temp = new int[size];

int count[10];

int radix = 1;

for (int i = 1; i <= d; i++) {

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

count[j] = 0;

}

for (int j = 0; j < size; j++) {

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

count[k]++;

}

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

count[j] = count[j - 1] + count[j];

}

for (int j = size - 1; j >= 0; j--) {

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

temp[count[k] - 1] = vec[j];

count[k]--;

}

for (int j = 0; j < size; j++) {

vec[j] = temp[j];

}

radix \*= 10;

}

3.8.2 基数排序截屏示例



## 3.9 总体系统的实现

3.9.1 总体程序的核心代码

while (i != 9) {

cout << endl << "请选择排序算法：";

cin >> i;

unsigned long long time\_start, time\_end;

switch (i) {

case 1:

vec2 = copyVector(num);

swap\_times = 0;

time\_start = GetTickCount64();

BubbleSort(vec2, num, swap\_times);

time\_end = GetTickCount64();

cout << "冒泡排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

cout << "冒泡排序交换次数：" << swap\_times << endl;

break;

case 2:

vec2 = copyVector(num);

swap\_times = 0;

time\_start = GetTickCount64();

SelectionSort(vec2, num, swap\_times);

time\_end = GetTickCount64();

cout << "选择排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

cout << "选择排序交换次数：" << swap\_times << endl;

break;

case 3:

vec2 = copyVector(num);

swap\_times = 0;

time\_start = GetTickCount64();

InsertionSort(vec2, num, swap\_times);

time\_end = GetTickCount64();

cout << "直接插入排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

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

break;

case 4:

vec2 = copyVector(num);

swap\_times = 0;

time\_start = GetTickCount64();

ShellSort(vec2, num, swap\_times);

time\_end = GetTickCount64();

cout << "希尔排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

cout << "希尔排序交换次数：" << swap\_times << endl;

break;

case 5:

vec2 = copyVector(num);

swap\_times = 0;

time\_start = GetTickCount64();

QuickSort(vec2, 0, num - 1, swap\_times);

time\_end = GetTickCount64();

cout << "快速排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

cout << "快速排序交换次数：" << swap\_times << endl;

break;

case 6:

vec2 = copyVector(num);

swap\_times = 0;

time\_start = GetTickCount64();

HeapSort(vec2, num, swap\_times);

time\_end = GetTickCount64();

cout << "堆排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

cout << "堆排序交换次数：" << swap\_times << endl;

break;

case 7:

vec2 = copyVector(num);

vec3 = copyVector(num);

compare\_times = 0;

time\_start = GetTickCount64();

MergeSort(vec2, vec3, 0, num - 1, compare\_times);

time\_end = GetTickCount64();

cout << "归并排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

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

break;

case 8:

vec2 = copyVector(num);

swap\_times = 0;

time\_start = GetTickCount64();

RadixSort(vec2, num, compare\_times);

time\_end = GetTickCount64();

cout << "基数排序所用时间：" << (time\_end - time\_start) << "ms" << endl;

cout << "基数排序交换次数：" << swap\_times << endl;

break;

case 9:

break;

default:

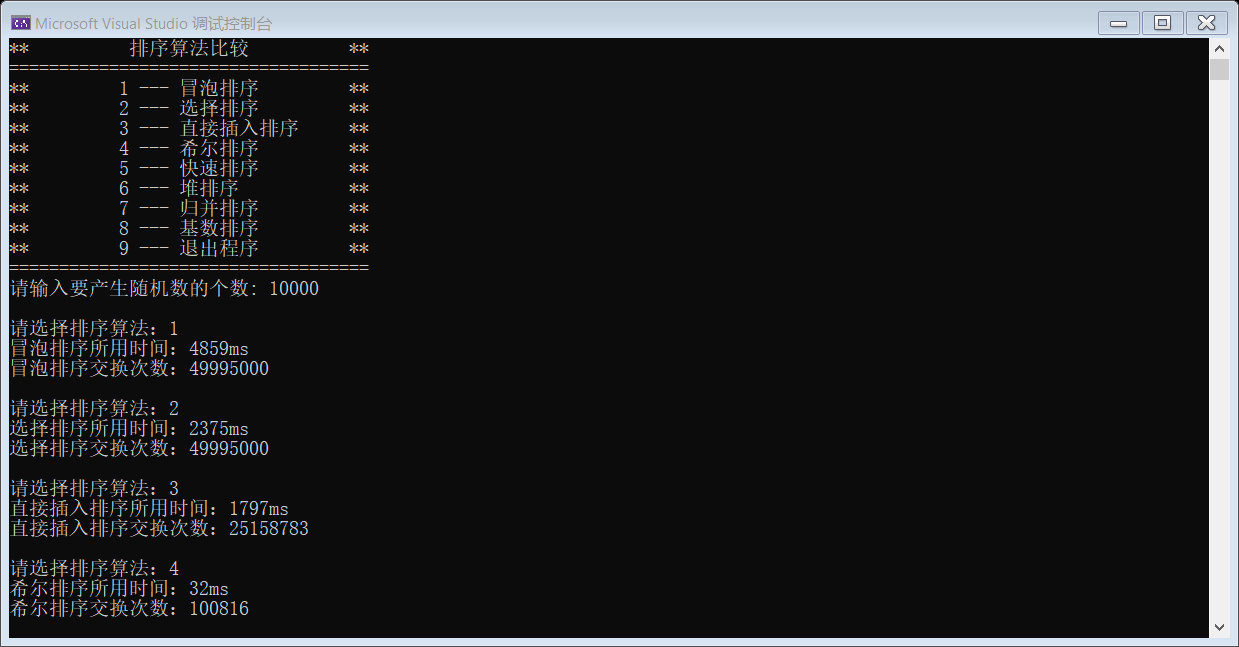
cerr << "请输入1到9间的一个正整数!" << endl;

break;

}

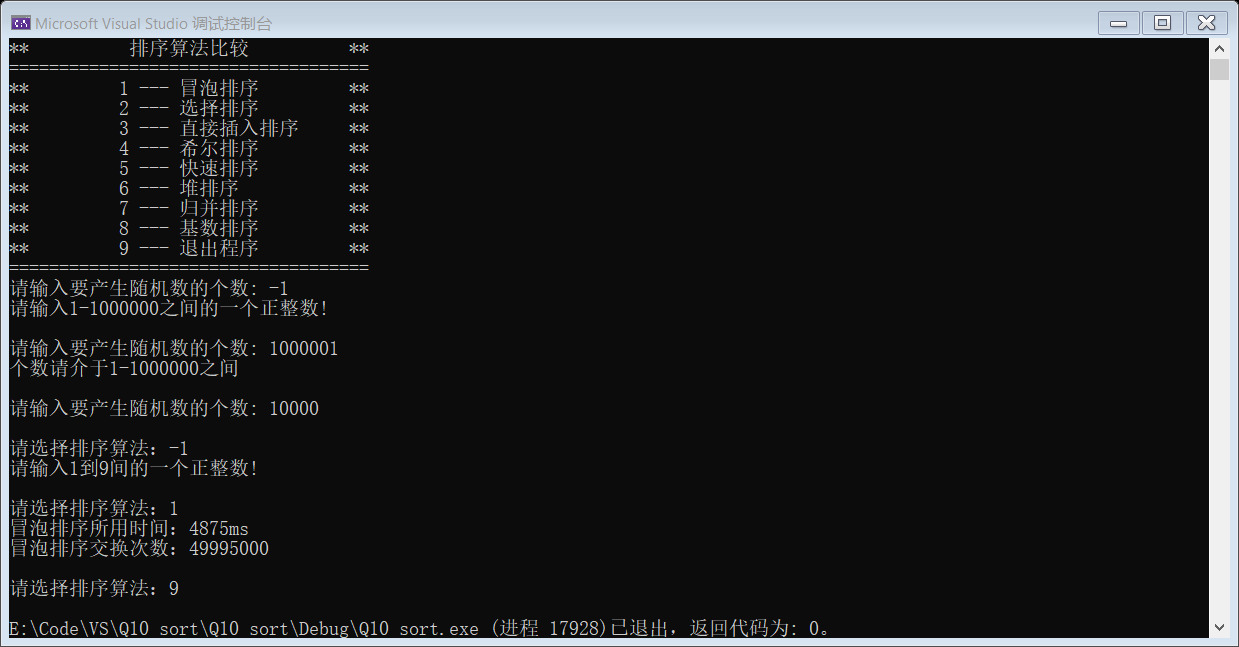
}

3.9.2 总体程序截屏示例

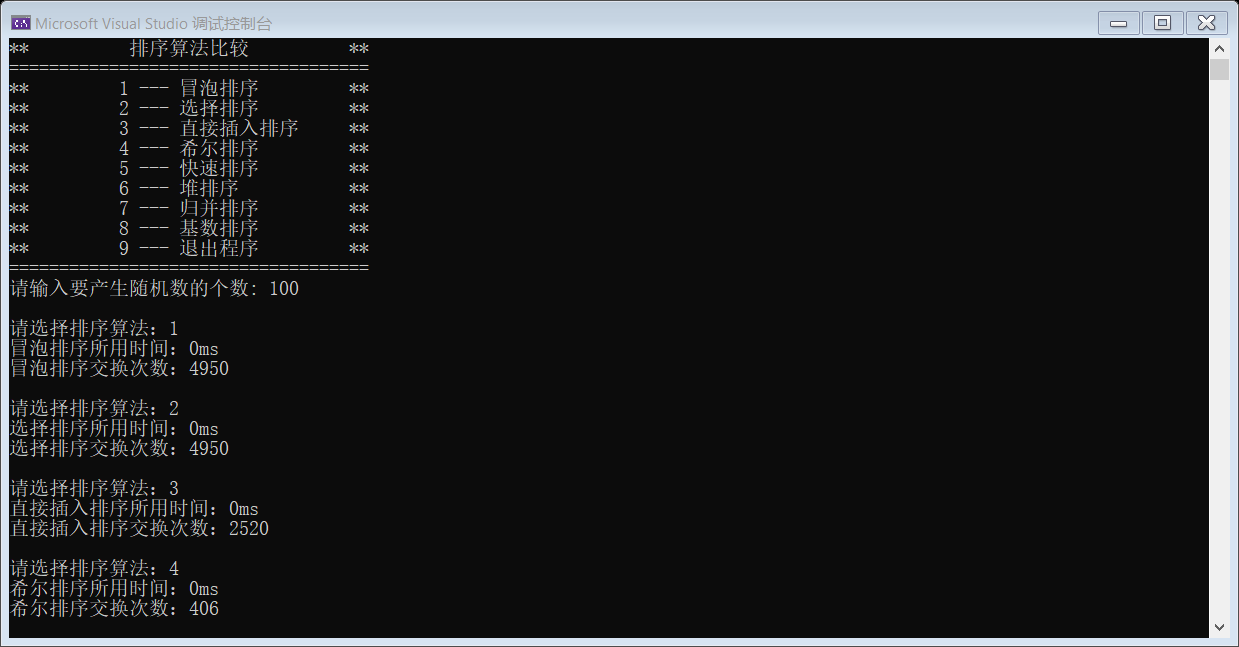


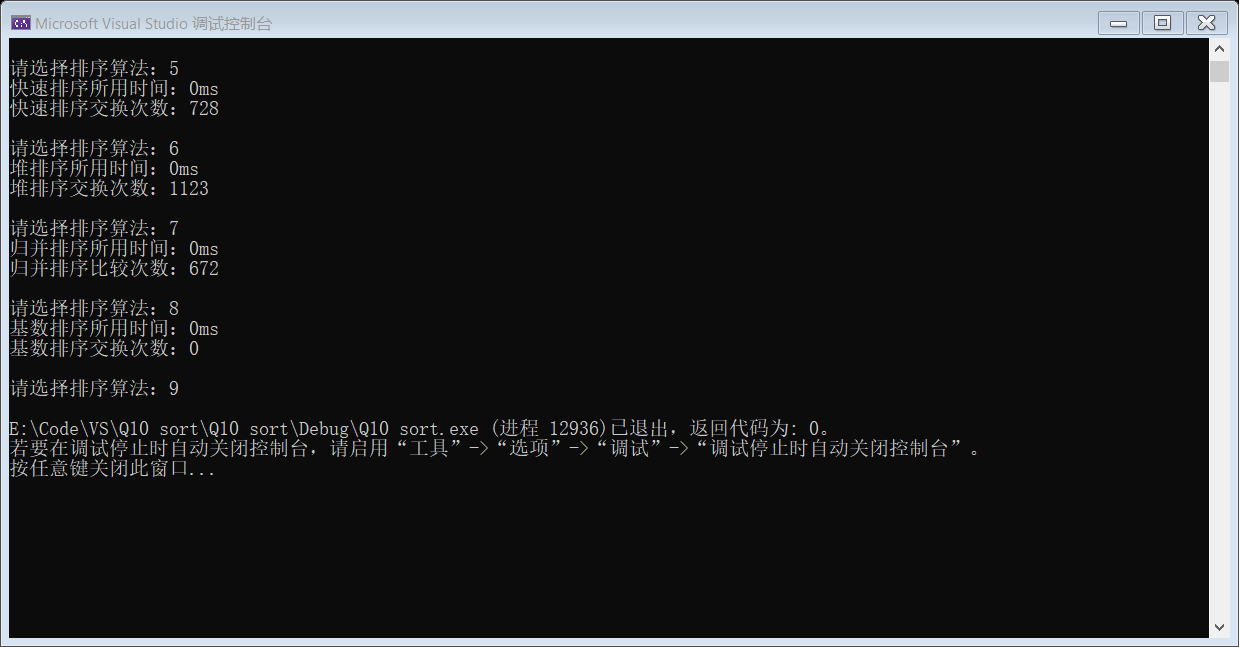
# 4 测试

## 4.1 不良输入测试

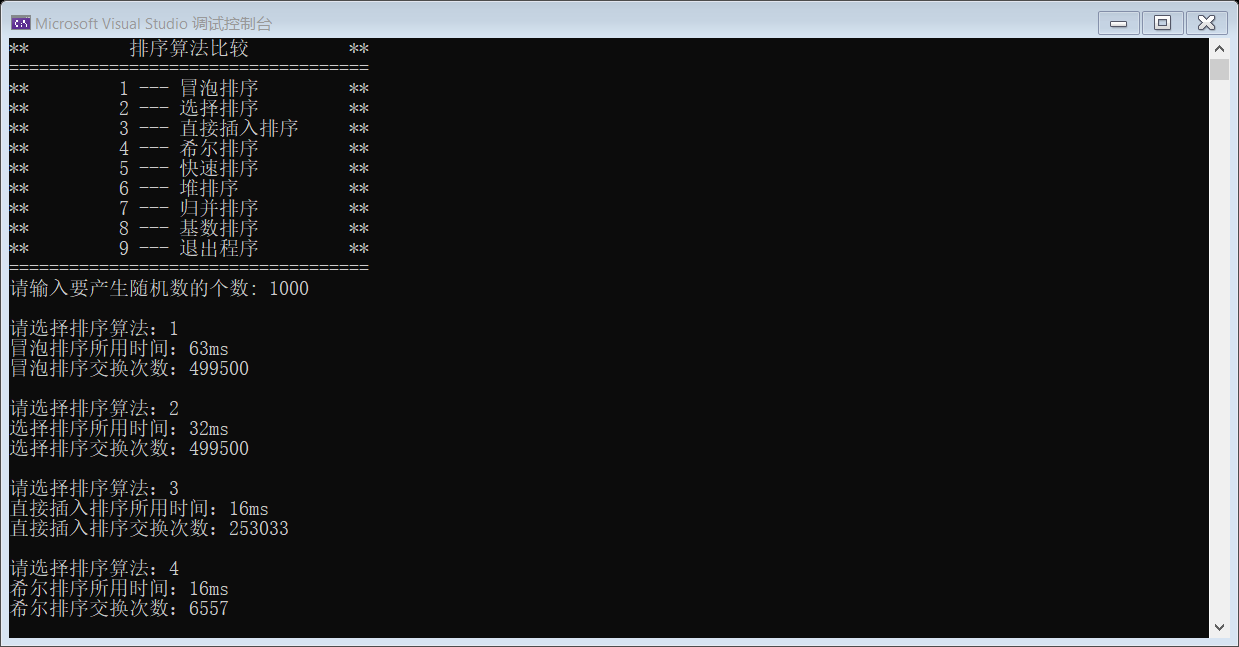


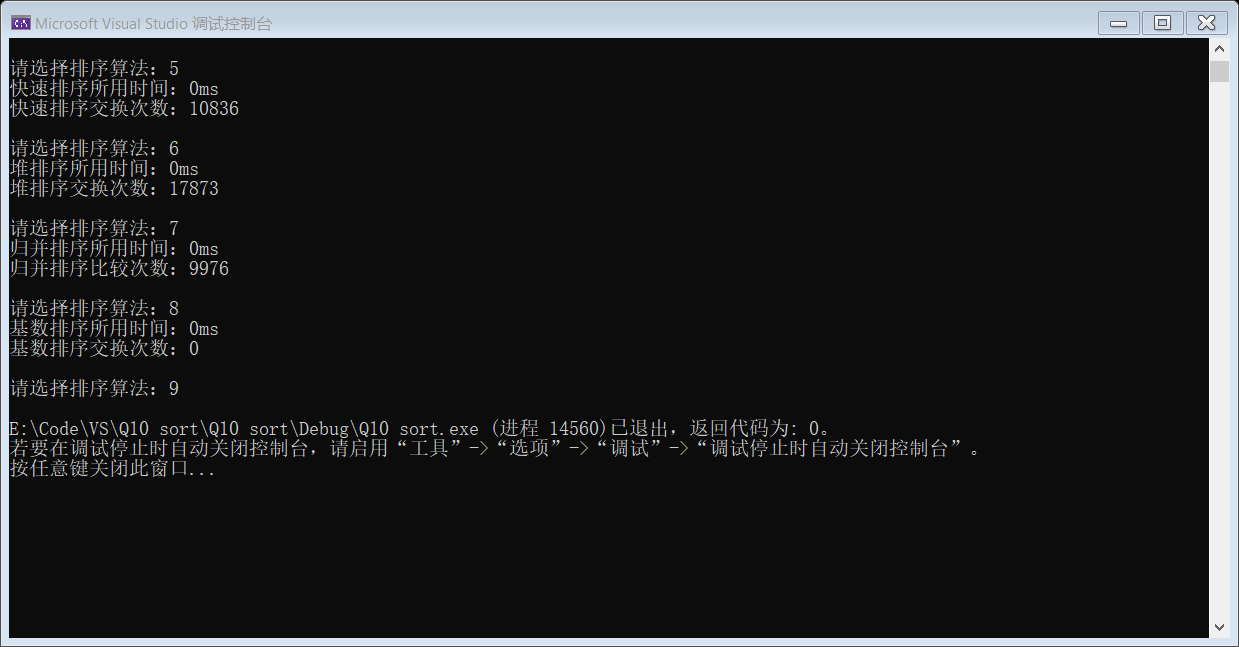
4.2 正常输入测试（100）





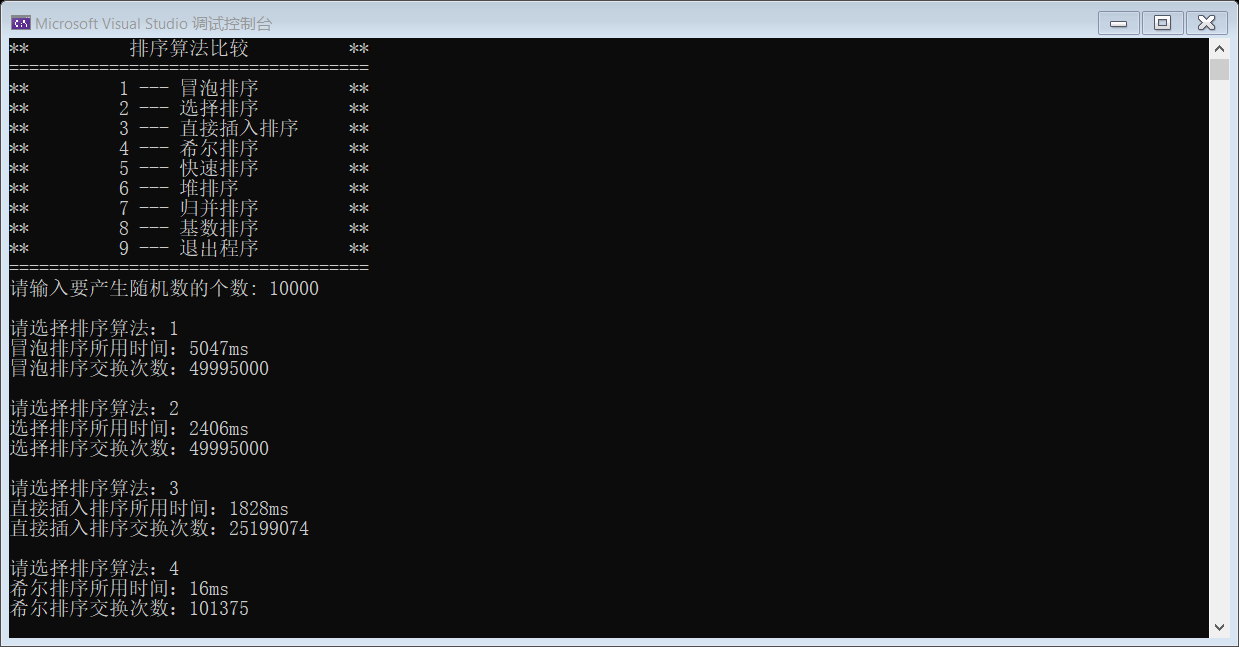
4.3 正常输入测试（1000）

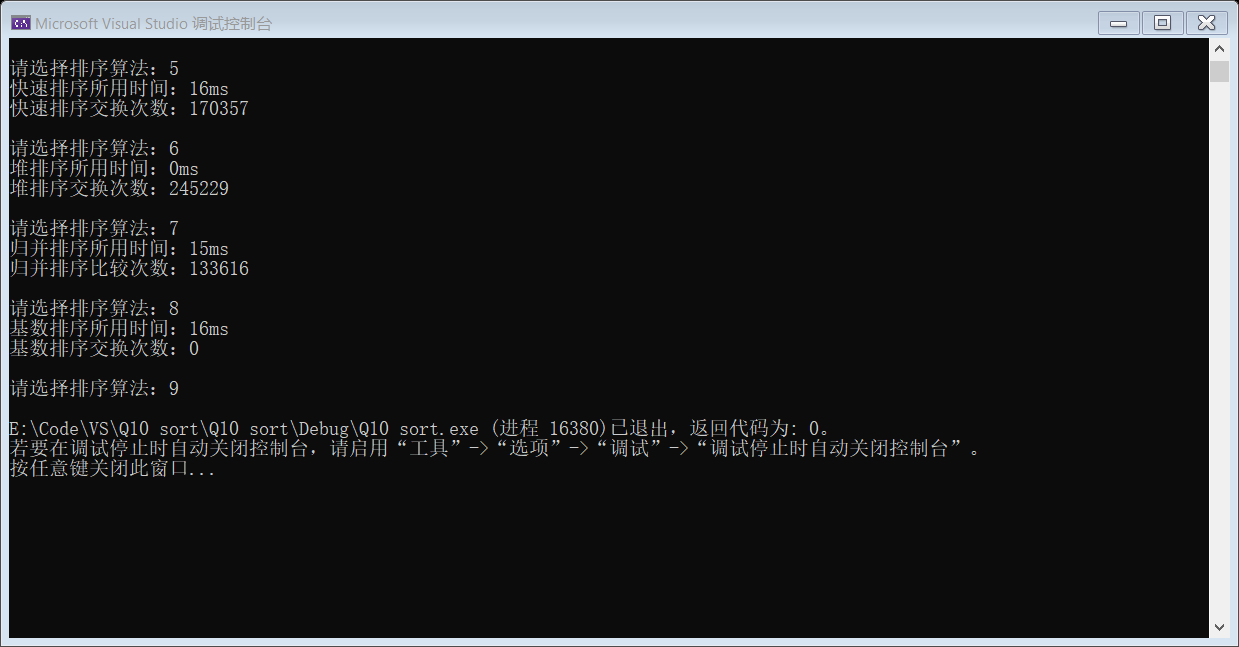




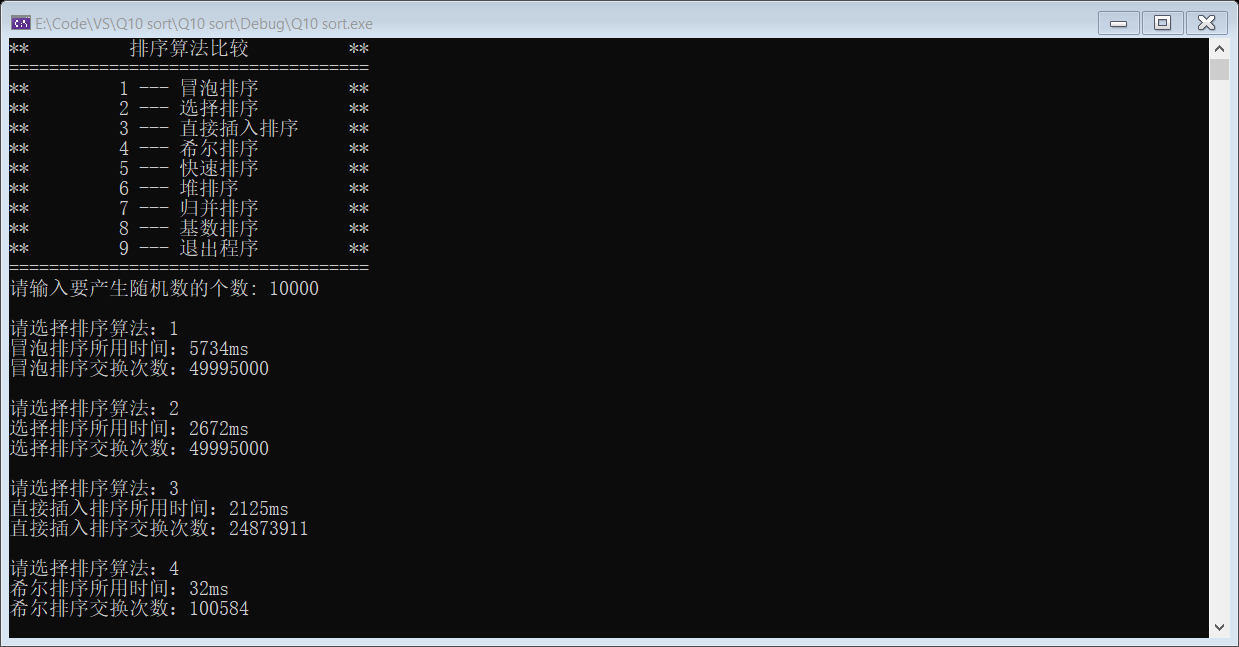
4.4 正常输入测试（10000）

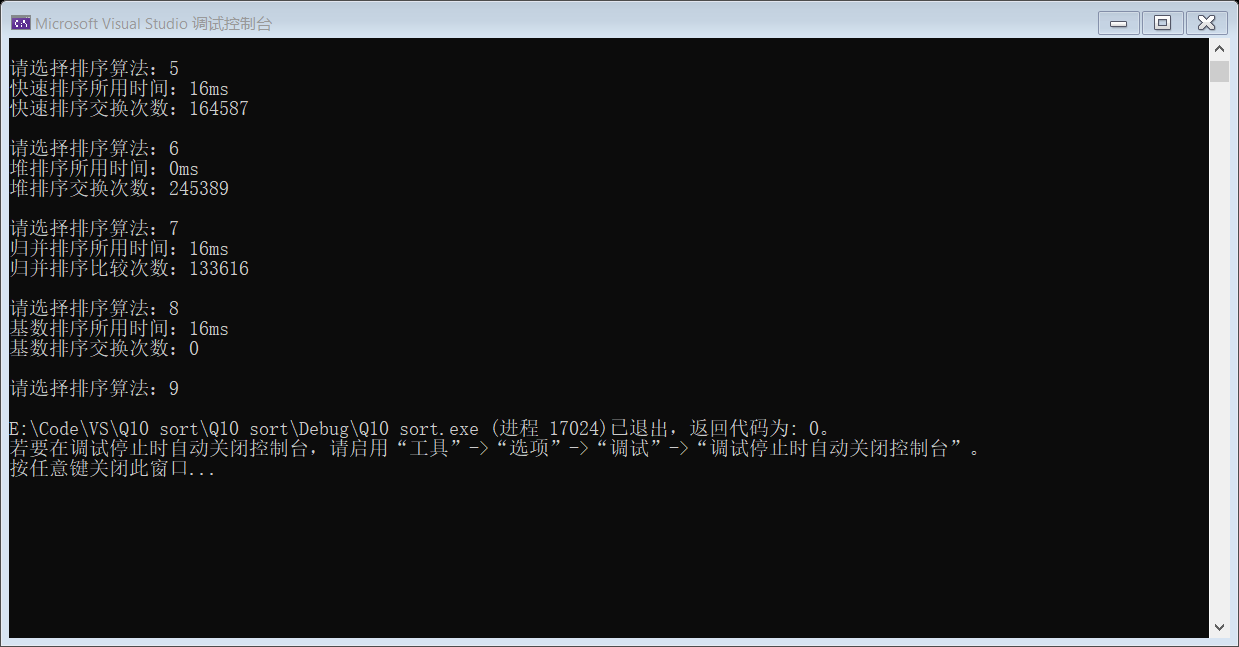
①



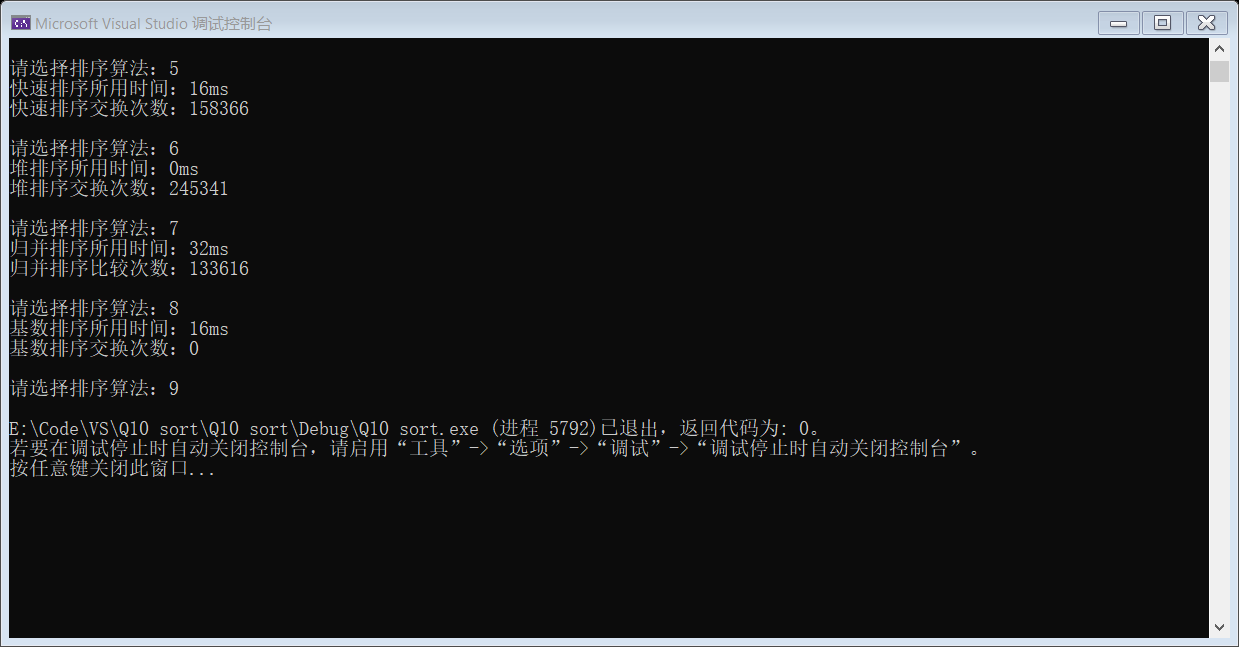
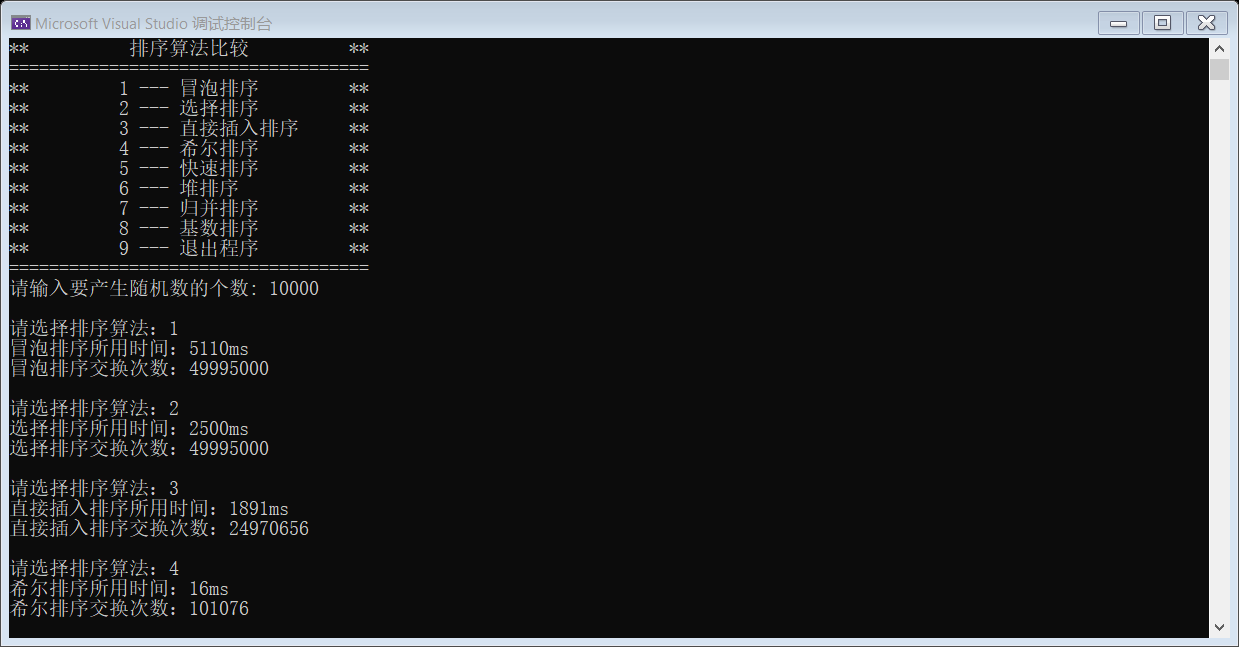


②

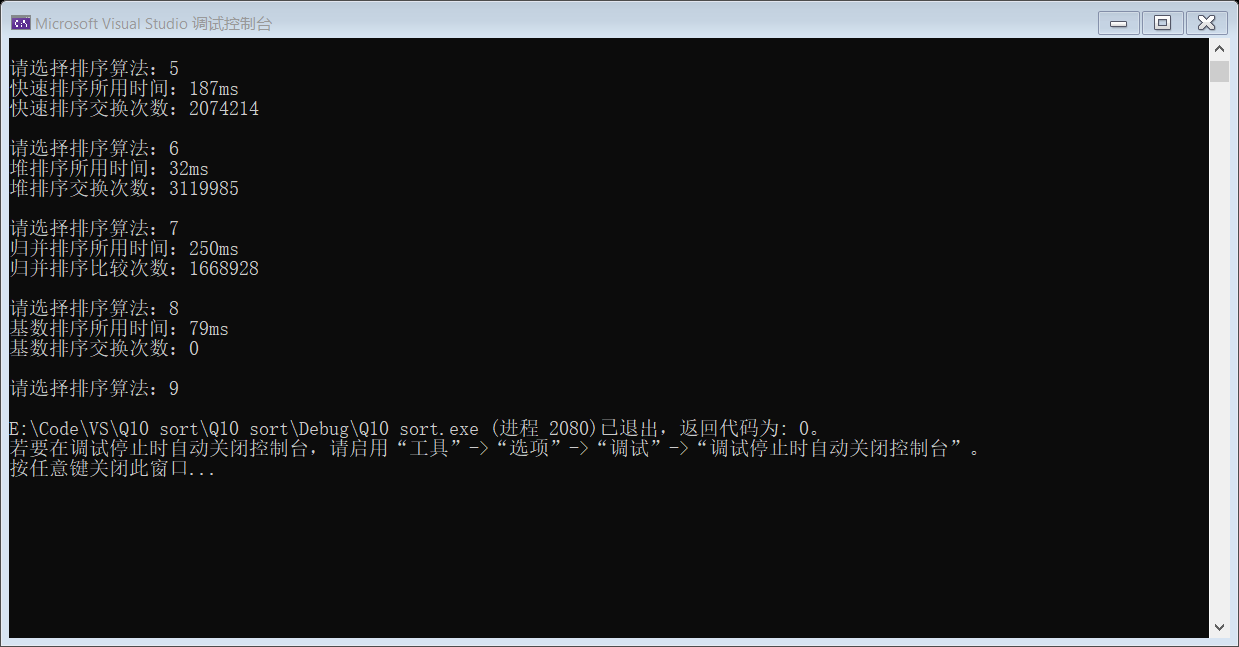
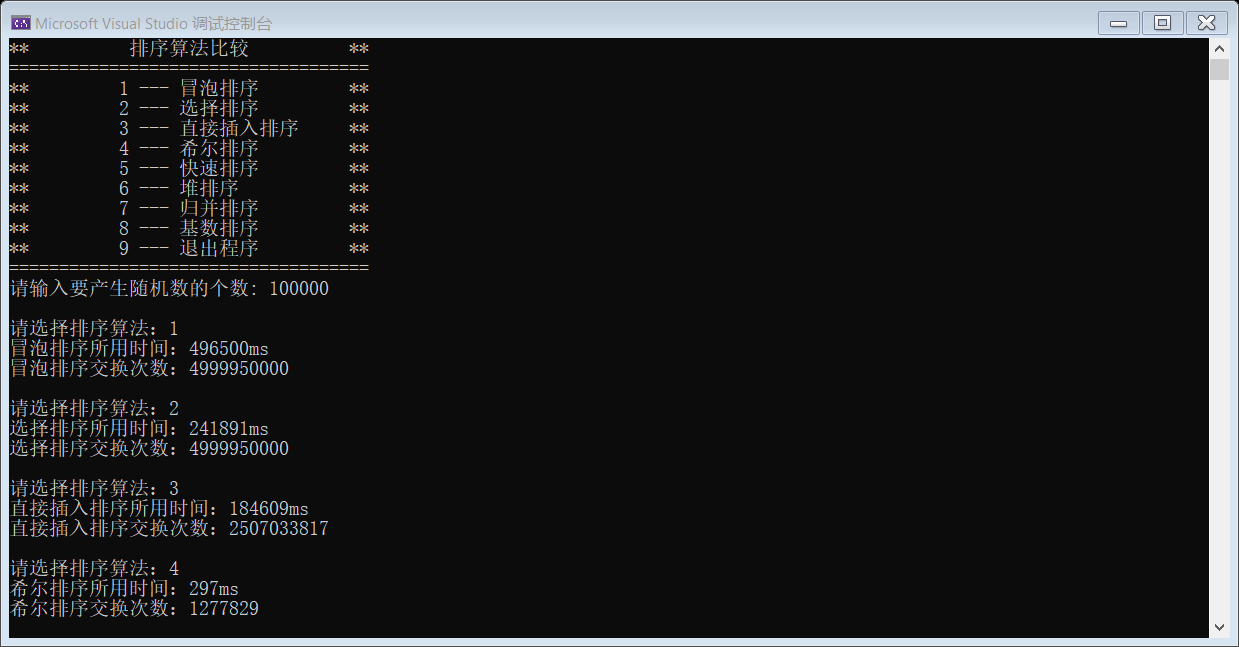




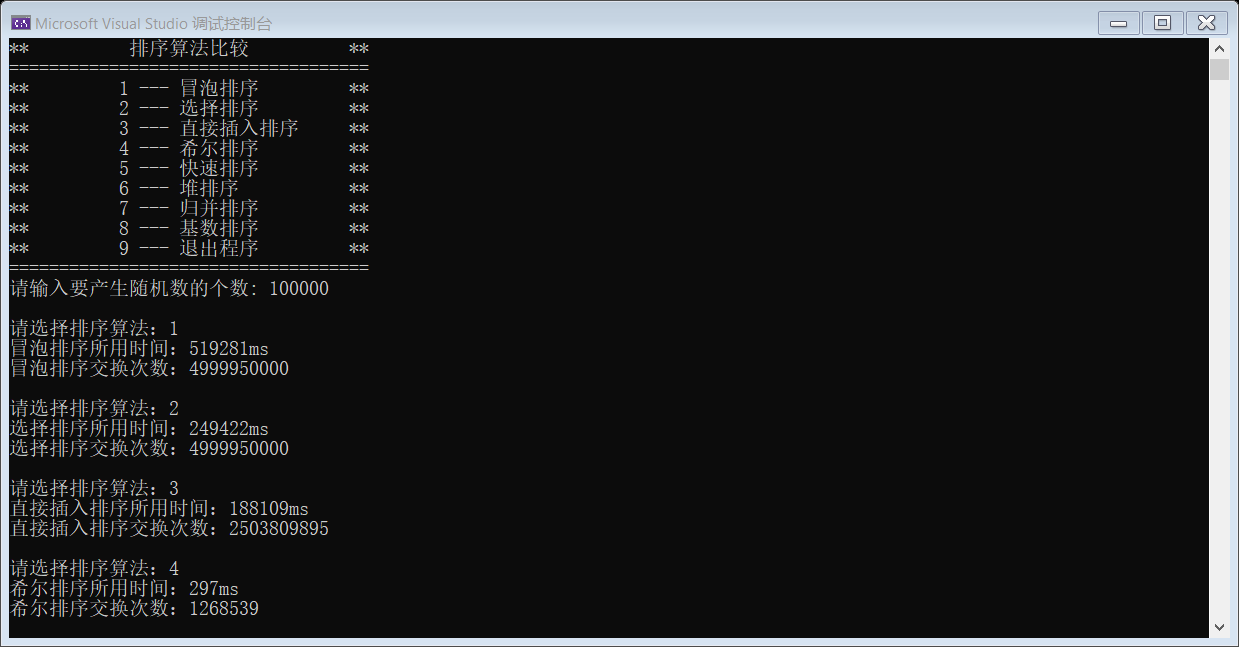
③

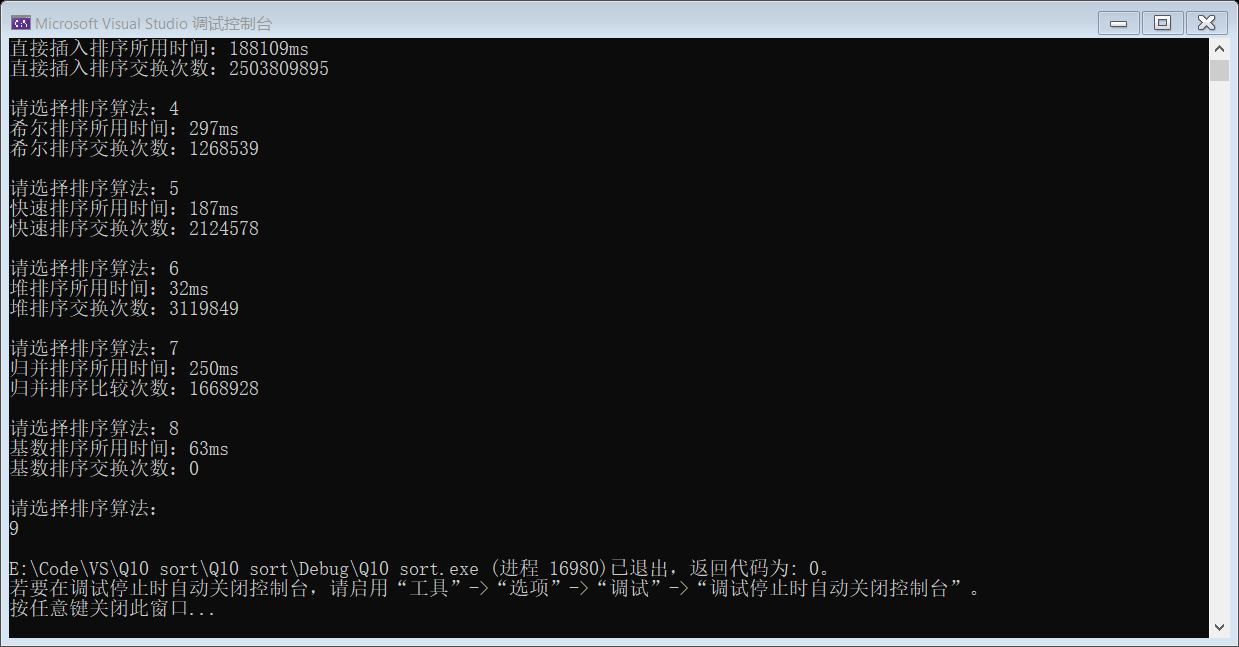


4.5 正常输入测试（100000）

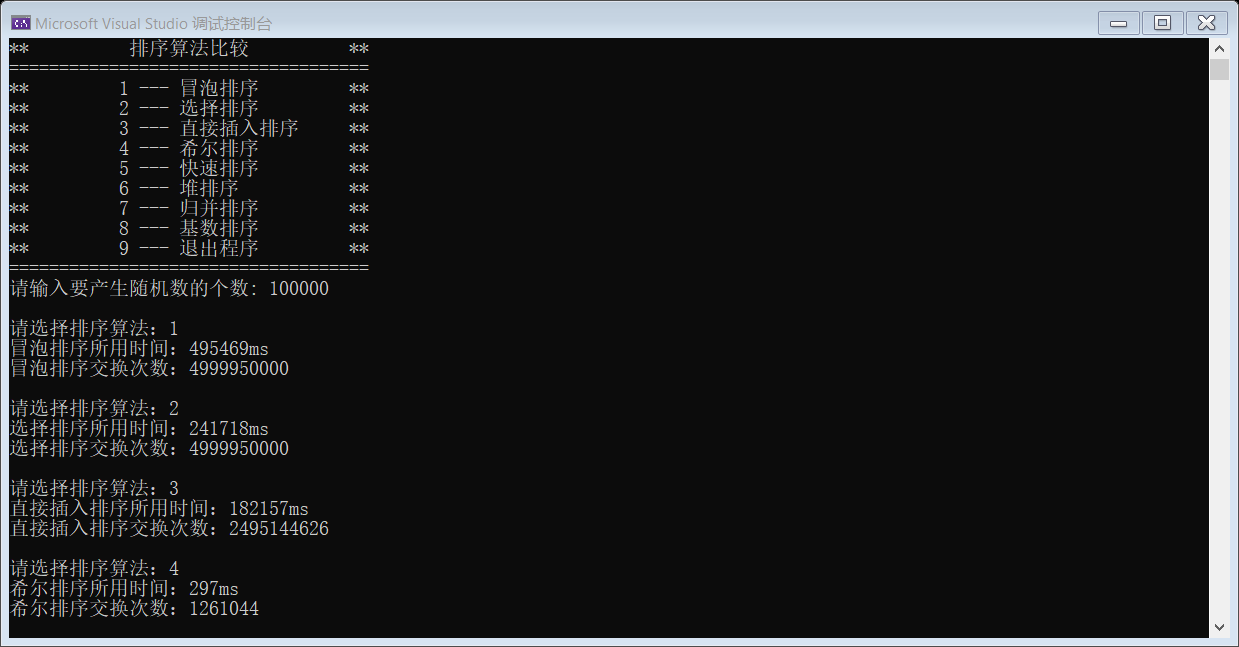
①

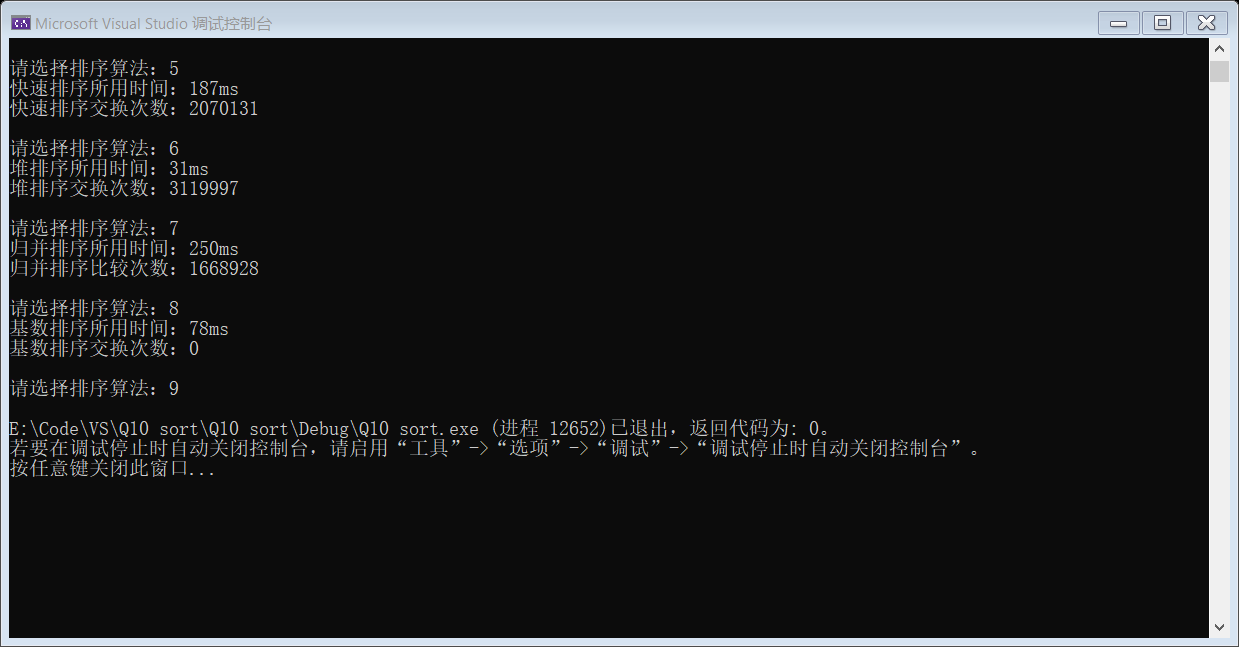
②





③





5 数据分析

在数据比较少（100个）的时候8种排序方式不分伯仲，排序花费的时间都在1ms以内。当数据增加到1000个时排序速度开始出现差距，其中冒泡排序耗时最长（63ms）其次是选择排序（32ms）然后是直接插入排序和希尔排序，他们的耗时大致相等为（16ms），剩下4种排序还是小于1ms无法看出孰优孰劣。当数据量继续增长至超过10000个时，由于每次随机生成数据时会出现较大的偏差，因此每组分别实验三次，从实验结果中可以看出，冒泡排序的耗时最长大概为5400ms，接下来依然是选择排序耗时大概在2500ms左右，再往下是直接插入排序时长大概为1950ms，然后是希尔排序、快速排序、基数排序和堆排序耗时大概都在16ms左右，只有堆排序的耗时仍在1ms以内。到了数据个数达到100000是冒泡排序、选择排序和插入排序十分耗时的特点变得特别明显。分别大概是500000ms、245000ms和185000ms，希尔排序三次实验结果均为297ms，快速排序为187ms，堆排序约为32ms，归并排序约为250ms，基数排序约为73ms。

总结：

插入排序、选择排序、冒泡排序

优点：稳定（只有冒泡排序和插入排序）、结构简单

缺点：慢、很慢和特别慢

希尔排序

优点：快，数据移动少

缺点：不稳定

快速排序

优点：极快且数据移动较少

缺点：不稳定

堆排序

优点：对序列要求相对快速排序更小，且辅助存储空间使用较少

缺点：不稳定，序列较短时比较麻烦

归并排序

优点：很快

缺点：要求提供大量的空间来辅助存储

基数排序

优点：不用比较，排序时间复杂度低

缺点：需要辅助存储空间，较麻烦