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：陈垲昕

学　　号：[redacted]

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

可通过按住 Ctrl 并单击访问说明文档各个模块：

1. 分析

背景与功能分析

排序算法是算法的重要基础，对不同的排序算法的比较以及使用，是了解算法分析算法的重要内容。排序算法的执行时间是衡量算法好坏的最重要的参数，其时间开销可以使用算法执行过程中**数据比较次数**与**数据移动次数**来衡量，这是本程序设计的原理

本程序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。

2.设计

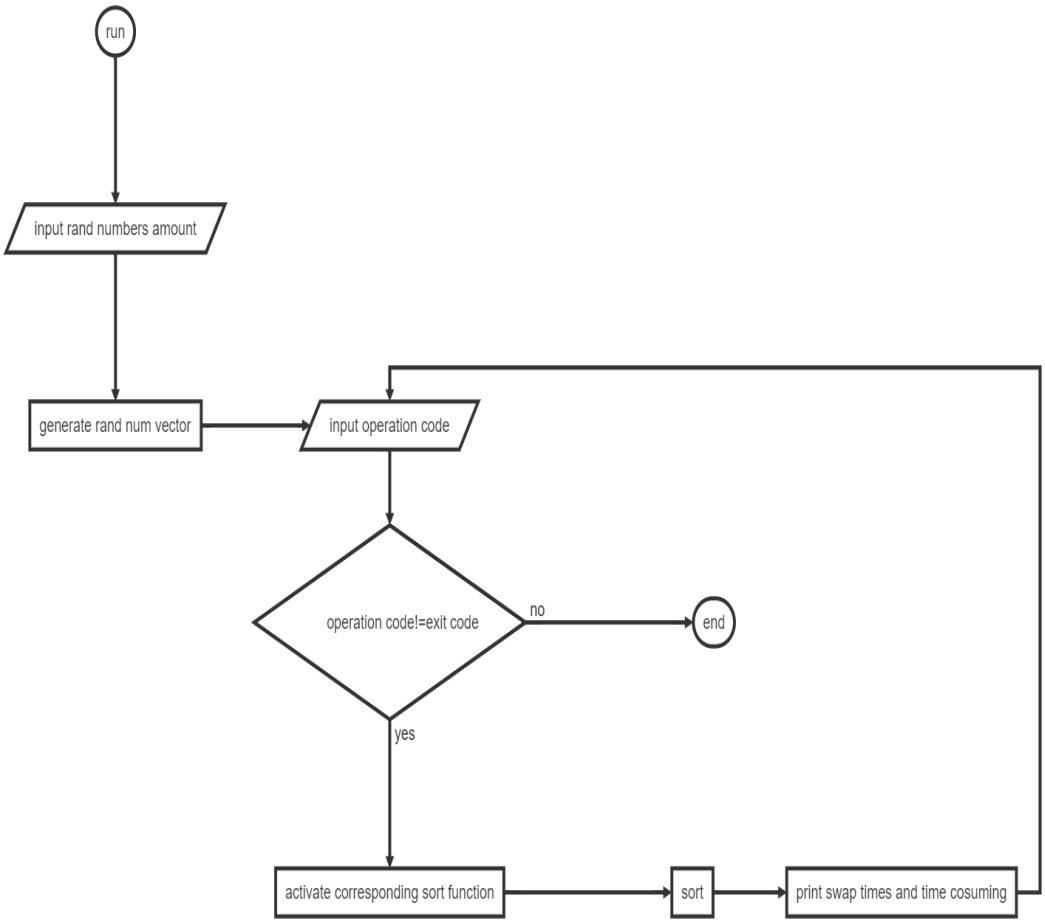
主要系统类设计

基本描述：**class SortCompare** 为总系统管理类，存储一个 **vector** 型容器 **_randArray** 用于存储随机数，选择相应的排序方式时，拷贝一份容器中存储的随机生成的特定数量的数据，进行排序并给出交换/比较的次数与所用的时间（单位：ms）

声明代码

```
class SortCompare {
public:
	//构造与析构函数
	SortCompare() = default;
	~SortCompare() { _randArray.clear(); }

	//运行程序
	void run();
private:
	//存放随机数的指针，动态内存分配
	vector<int> _randArray;
	int _arraySize;

	void _generateRandNumberArray(int randArraySize);

	void _sortAnalyse(int sortCode);

	void _bubbleSort(vector<int>& ivec,long long& counter);
	void _selectSort(vector<int>& ivec, long long& counter);
	void _directInsertSort(vector<int>& ivec, long long& counter);
	void _shellSort(vector<int>& ivec, long long& counter);

	void _quickSort(vector<int>& ivec, long long& counter);
	void _qSort(vector<int>& ivec, int low, int high, long long& coun
ter);
	int _qPartition(vector<int>& ivec, int low, int high, long long&
counter);

	void _heapSort(vector<int>& ivec, long long& counter);
	void _maintainHeap(vector<int>& ivec, int top, int end, long long
& counter);

	void _mergeSort(vector<int>& ivec, long long& counter);
	void _mSort(vector<int> &ivec, vector<int>& tmp,int lo,int hi, lo
ng long &counter);
	void _merge(vector<int>& ivec, vector<int>& tmp,int lo,int mid,in
t hi, long long& counter);

	void _radixSort(vector<int>& ivec, long long& counter);

};
```

流程图


run
input rand numbers amount
generate rand num vector
input operation code
operation code!=exit code
no
end
yes
activate corresponding sort function
sort
print swap times and time cosuming

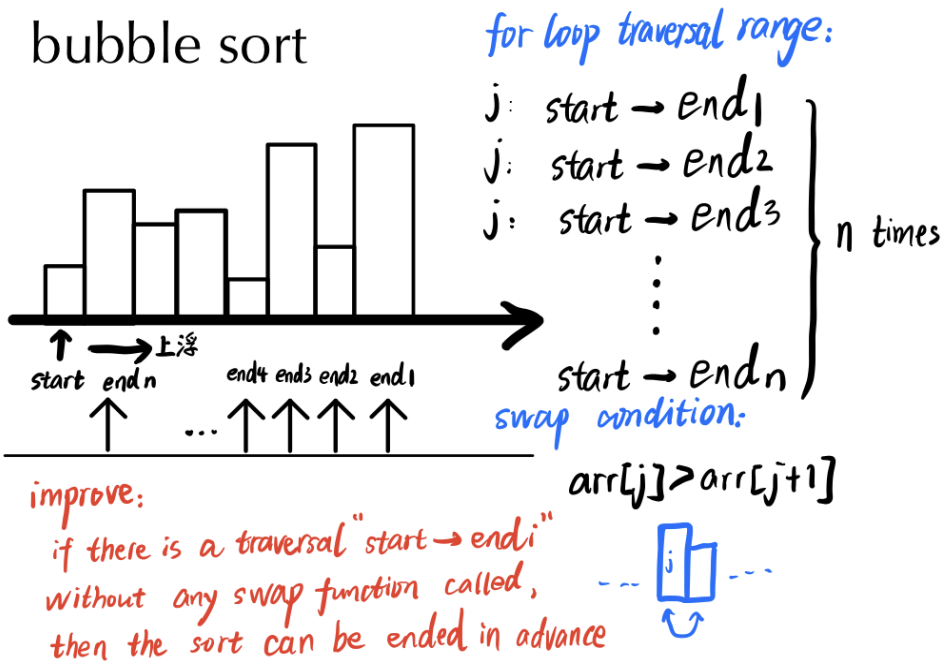
3.核心功能实现

3.1 冒泡排序分析

基本描述

算法平均时间复杂度	$O(n^2)$
算法最坏情况时间复杂度	$O(n^2)$
算法最好情况时间复杂度	$O(n)$
算法空间复杂度	$O(1)$
是否为原地(In-Place)算法	是
算法稳定性	稳定

图解



算法的分析

冒泡排序算法是最简单的排序算法之一，它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序不符合排序规则就把他们交换过来，如此重复地进行直到没有再需要交换的一对元素为止，此时排序完成。

冒泡排序过程中，需要进行 **n-1** 躺冒泡，其中第 **i** 次冒泡需要进行 **n-i** 次比较和交换操作，故总共的比较操作次数为

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

对冒泡排序有一种简易的改进方式：设置一个 **bool** 变量标记一次起泡过程中是否进行了交换，如果没有进行交换，则说明数组已经排序完成，之后的起泡已经不需要了，故可以通过检测 **bool** 变量值提前结束排序。这种方法对冒泡排序的效率优化有限。

排序算法代码段

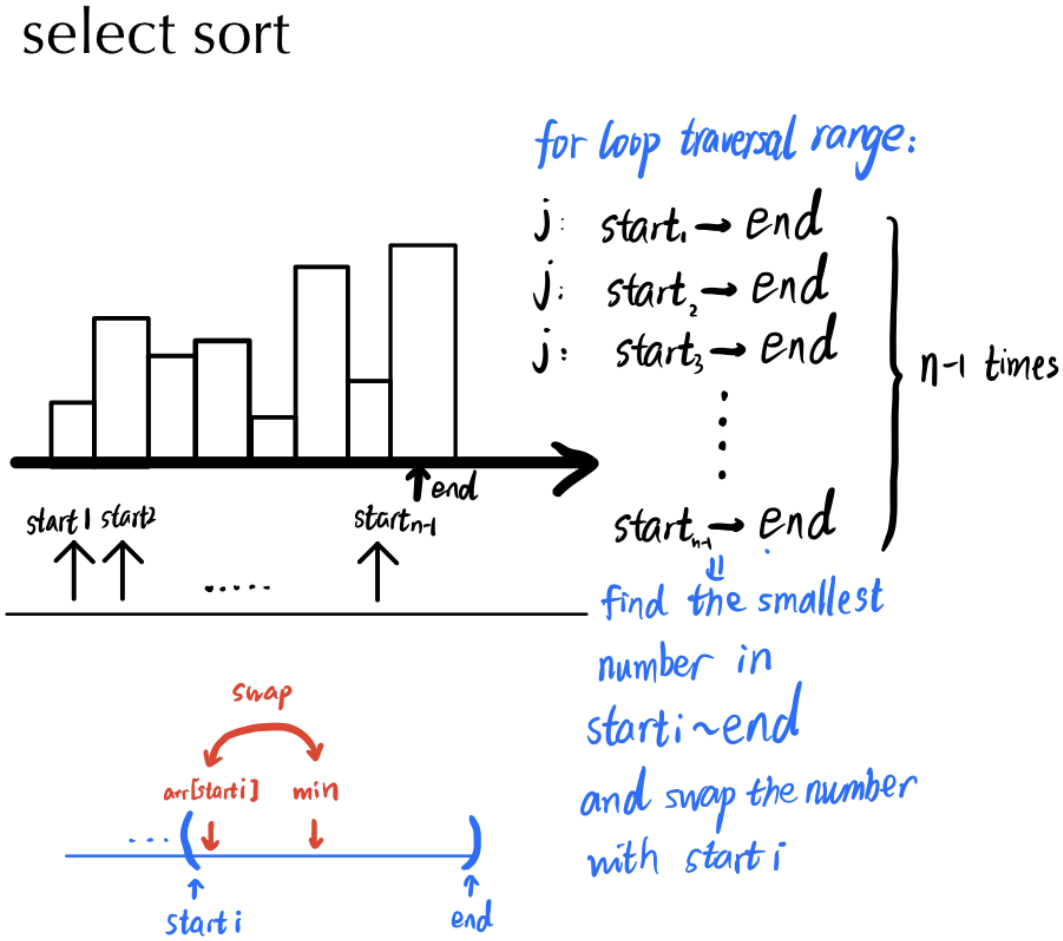
```
void SortCompare::_bubbleSort(int* ivec, long long& counter)
{
	bool hasExchange = false;
	for (int i=_arraySize-1; i >0; i--)
	{
		hasExchange = false;
		for (int j = 0; j < i; j++)
		{

			if (less(ivec[j + 1], ivec[j]))
			{

				std::swap(ivec[j], ivec[j + 1]);
				++counter;
				hasExchange = true;
			}
		}
		if (!hasExchange)
		{
			return;
		}
	}
}
```

3.2 选择排序分析

基本描述

算法平均时间复杂度	$O(n^2)$
算法最坏情况时间复杂度	$O(n^2)$
算法最好情况时间复杂度	$O(n^2)$
算法空间复杂度	$O(1)$
是否为原地(In-Place)算法	是
算法稳定性	不稳定

图解



算法的分析

选择排序将数组视为两个部分：未排序的数组与已排序的数组。程序通过 **n-1** 次遍历，在第 **i** 次遍历中寻找 **arr[i]与数组末位元素**之间的最小值，并将最小值与 **arr[i]**进行交换。可以说，选择排序一定会通过 n-1 次遍历找到每次对应的最小值，以此维护它的两个子数组。故时间复杂度也为：

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

且不论最坏情况还是最好的情况下，复杂度都为$O(n^2)$

排序算法代码段

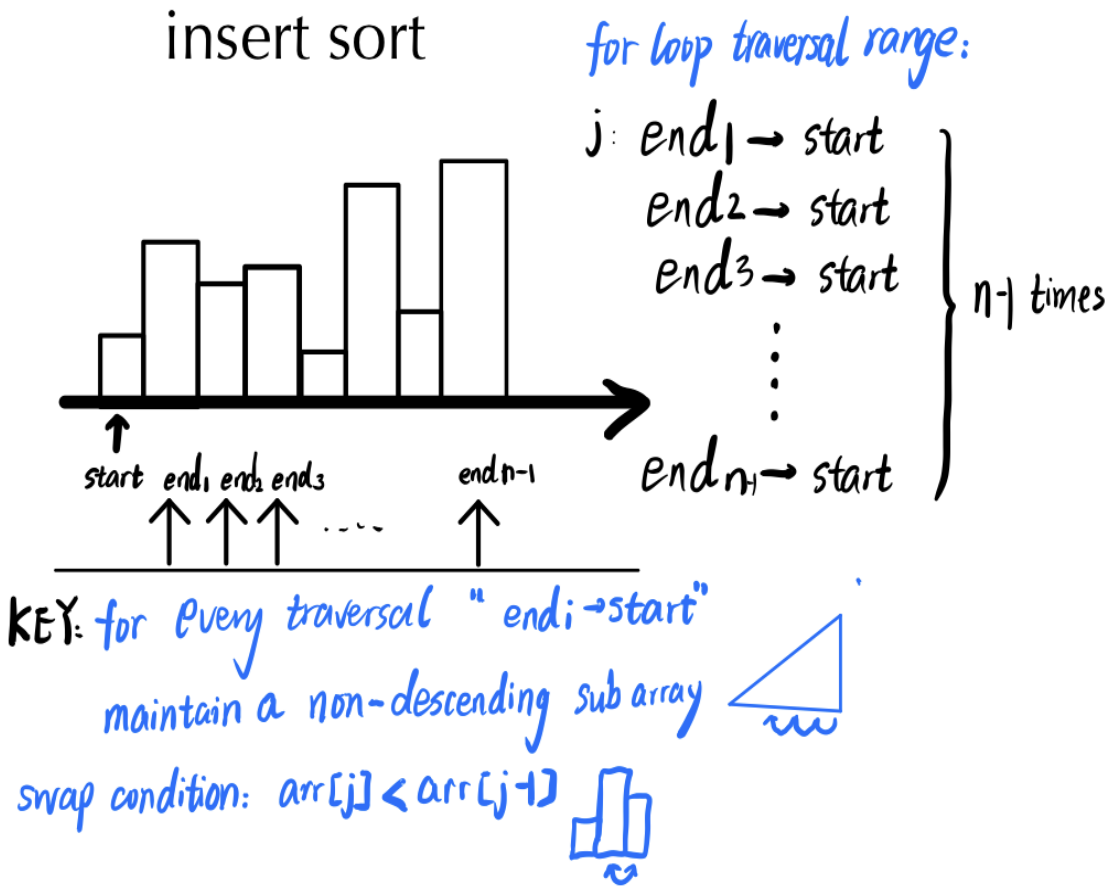
```
void SortCompare::_selectSort(int* ivec, long long& counter)
{
	int vecSize = _arraySize;
	for (int i = 0; i < vecSize; i++)
	{
		int minIndex = i;
		for (int j = i + 1; j < vecSize; j++)
		{

			if (less(ivec[j], ivec[minIndex]))
			{
				minIndex = j;
			}
		}
		std::swap(ivec[i], ivec[minIndex]);
		++counter;
	}
}
```

3.3 直接插入排序分析

基本描述

算法平均时间复杂度	$O(n^2)$
算法最坏情况时间复杂度	$O(n^2)$
算法最好情况时间复杂度	$O(n)$
算法空间复杂度	$O(1)$
是否为原地(In-Place)算法	是
算法稳定性	稳定

图解



算法的分析

直接插入排序是插入排序中较为简单与容易理解的。它可以类比在洗扑克牌并依次抓起扑克牌时对扑克牌进行排序的过程。直接插入排序的关键是：在每一次遍历中，都必须维护起始位置到遍历标记之间的数组的非降序性。在每次循环中，从数组后部向前遍历，发现有 **arr[j+1]>arr[j]**的情况便进行交换。这样便达到了直接插入排序的目的。

排序算法代码段

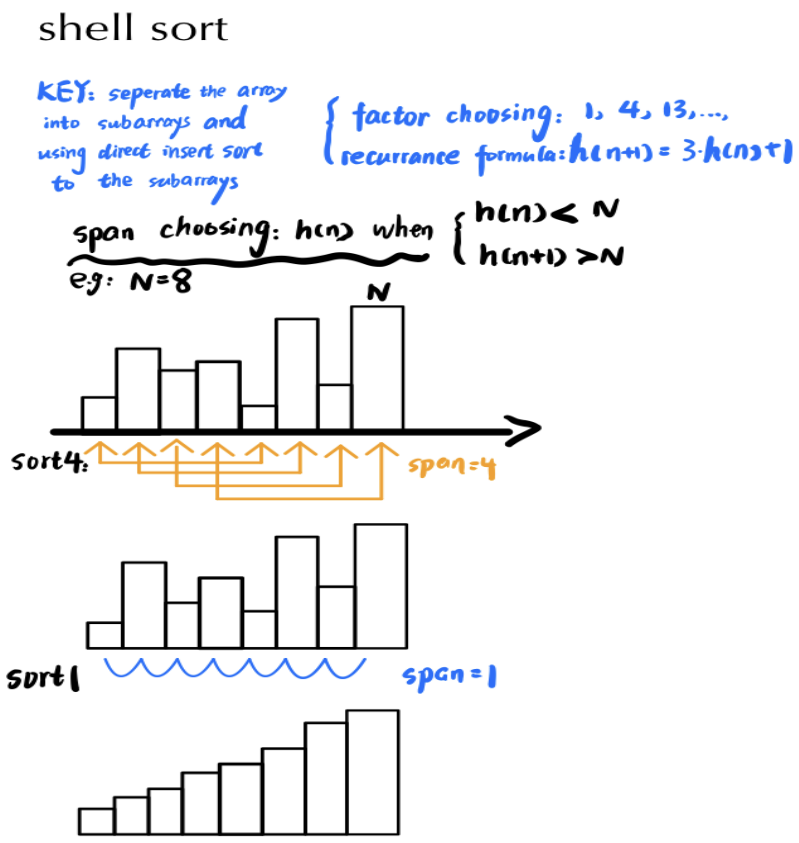
```
void SortCompare::_directInsertSort(int* ivec, long long& counter)
{
	int vecSize = _arraySize;
	for (int i = 1; i < vecSize; i++)
	{
		for (int j = i; j > 0 && less(ivec[j], ivec[j - 1]); j--)
		{

			std::swap(ivec[j], ivec[j - 1]);
			++counter;
		}
	}
}
```

3.4 希尔排序分析

基本描述

算法平均时间复杂度	取决于取跨步值的方法
算法最坏情况时间复杂度	$O(n^2)$
算法最好情况时间复杂度	$O(n)$
算法空间复杂度	$O(1)$
是否为原地(In-Place)算法	是
算法稳定性	不稳定

图解



算法的分析

希尔排序的核心思想是缩小增量排序。它取一个整数的跨度 gap 作为间隔，将全部元素分为 gap 个子序列，其中距离恰好为 gap 的元素放在同一个子序列中，并且在一次操作下，对这些子序列分别进行直接插入排序。接着，根据特定的规则缩小 gap，重复上述的子序列划分与排序工作指导最后 gap=1 时对整个数组进行最后一次直接插入排序。英文开始 gap 取值大，每个子序列中元素少，排序起来较快，而到了 gap 值逐渐减小时，子序列的有序性已经随着之前排序的进行越来越高，故后续的比较工作与元素移动次数已经远远小于未排序时的数列。

对 gap 缩小增量的选取没有固定的规则，不同的 gap 对不同序列的优化程度可能不同，这点还待数学上进一步的证明。

排序算法代码段

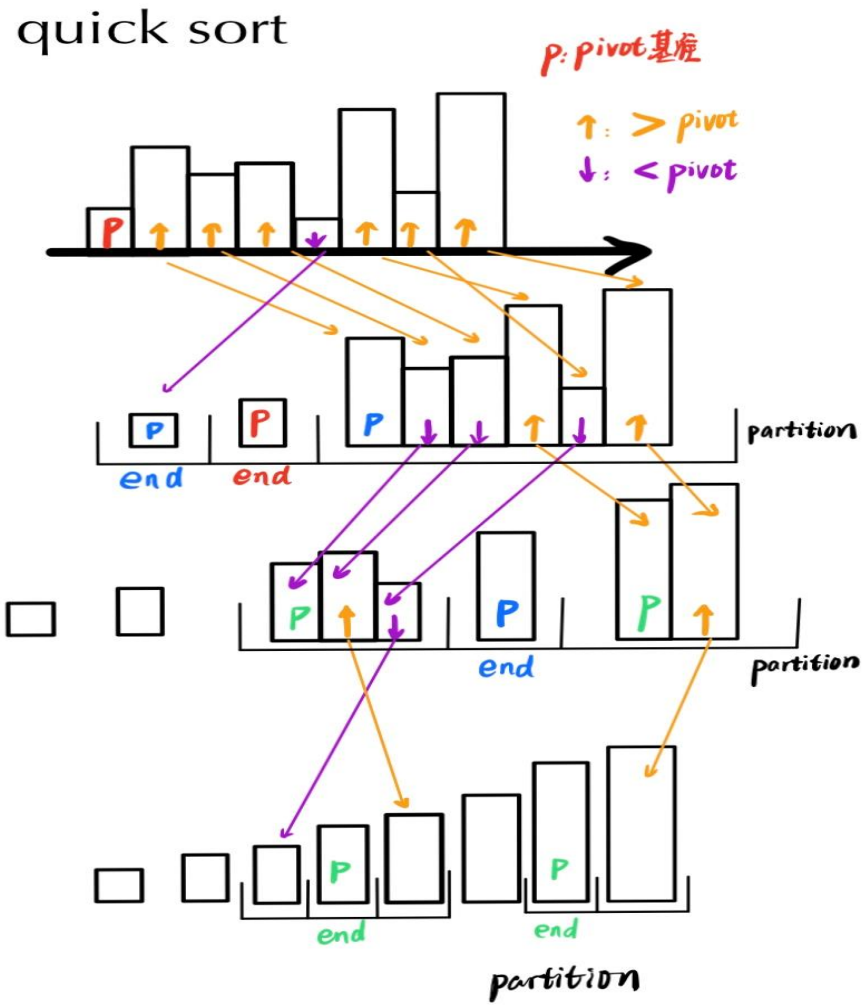
```
void SortCompare::_shellSort(int* ivec, long long& counter)
{
	int vecSize = _arraySize;
	int groupSize = 1;
	while (groupSize < vecSize / 3) groupSize = 3 * groupSize + 1;
	while (groupSize >= 1)
	{
		for (int i = groupSize; i < vecSize; i++)
		{
			for (int j = i; j >= groupSize && (less(ivec[j], ivec
[j - groupSize])); j -= groupSize)
				std::swap(ivec[j], ivec[j - groupSize]);
			++counter;
		}
		groupSize = groupSize / 3;
	}
}
```

3.5 快速排序分析

基本描述

算法平均时间复杂度	$O(nlog_2(n))$
算法最坏情况时间复杂度	$O(n^2)$
算法最好情况时间复杂度	$O(nlog_2(n))$
算法空间复杂度	$O(nlog_2(n))$
是否为原地(In-Place)算法	不是
算法稳定性	不稳定

图解



算法的分析

快速排序是应用最广泛的算法，其核心思想为分治法。快排的排序思路是对当前的序列，取序列中某个元素（本题使用第一个元素）为基准（pivot），将其他的数分成小于 pivot 与大于等于 pivot 的两个序列，分别置于 pivot 的左右形成两个子序列（两个序列中，所有元素大于或小于 pivot）。此后再对这两个子序列进行同样的操作，直到最后每个子序列都被划分为不可再划分的一个元素，此时排序完成。

c++标准库的 sort 算法是以快速排序为基本框架。不过由于快速排序为递归调用，为减小递归次数并提高效率，通常会在传入递归的子序列缩小到一定程度时使用诸如插入排序等算法以减小递归深度。

排序算法代码段

```
void SortCompare::_quickSort(int* ivec, long long& counter)
{
	_qSort(ivec, 0, _arraySize - 1, counter);
}

void SortCompare::_qSort(int* ivec, int low, int high, long long& count
er)
{
	if (low >= high) return;

	int cut = _qPartition(ivec, low, high, counter);
	_qSort(ivec, low, cut - 1, counter);
	_qSort(ivec, cut+ 1, high,counter);
}
int SortCompare::_qPartition(int* ivec, int low, int high, long long& c
ounter)
{
	//数组切分:ivec[low...i-1],ivec[i],ivec[i+1...high]

	int i = low, j = high + 1;//理解为begin()和end()对应索引

	int x = ivec[low];
	while (true)
	{
		//扫描左右，检查是否结束并交换元素
		while (ivec[++i] < x) if (i == high) break;
		while (x < ivec[--j]) if (j == low) break;
		if (i >= j) break;

		std::swap(ivec[i], ivec[j]);
		++counter;
```

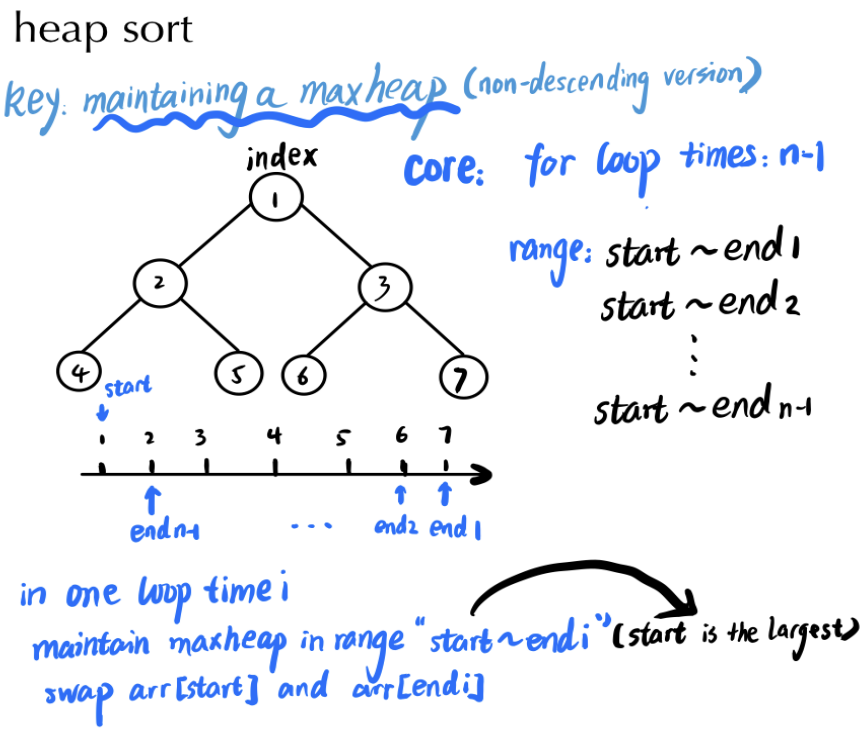
```
        }
        std::swap(ivec[low], ivec[j]);
        ++counter;
        return j;
}
```

3.6 堆排序分析

基本描述

算法平均时间复杂度	$O(nlog_2(n))$
算法最坏情况时间复杂度	$O(nlog_2(n))$
算法最好情况时间复杂度	$O(nlog_2(n))$
算法空间复杂度	$O(1)$
是否为原地(In-Place)算法	是
算法稳定性	不稳定

图解



算法的分析

堆排序是一种依赖于完全二叉树的数据结构的排序方式。通过完全二叉树，我们可以构建大根堆与小根堆。其中大根堆用于建立非降序数组，而小根堆用于建立非升序数组。我们以大根堆为例。在排序的过程中将数组视为两个子序列——待排序的子序列与未排序的子序列，将未排序的子序列视为大根堆，并且维护大根堆，即将最大的元素置于堆顶。在 n-1 次循环中重复以下核心操作:1.维护大根堆，将当前数组最大的元素置于堆顶；2.将堆顶元素与未排序的子序列的最后一个元素进行交换，该元素视为放入已排序的子序列中。

排序算法代码段

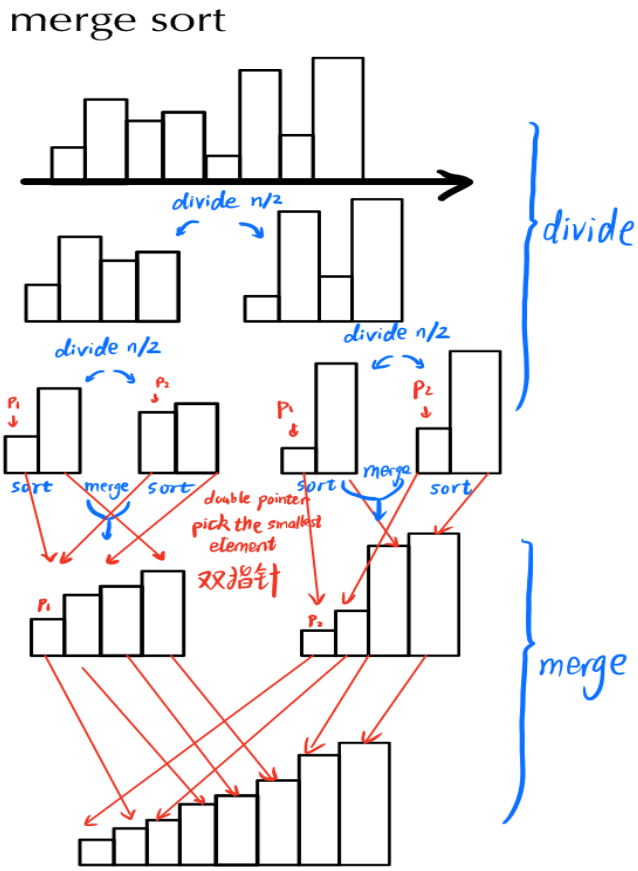
```
void SortCompare::_heapSort(int* ivec, long long& counter)
{
	int length = _arraySize;
	int lastIndex = length - 1;
	for (int cur = lastIndex/ 2; cur >= 0; --cur)
		_maintainHeap(ivec, cur, lastIndex,counter);
	for (int end = lastIndex; end >= 0; --end)
	{
		std::swap(ivec[0], ivec[end]);
		++counter;
		_maintainHeap(ivec, 0, end - 1, counter);
	}
}

void SortCompare::_maintainHeap(int* ivec, int top, int end, long long&
 counter)
{
	int topData = ivec[top];
	for (int i = top * 2 + 1; i <= end; i = i * 2 + 1)
	{
		if (i + 1 <= end && ivec[i] < ivec[i + 1])
		{
			++i;
		}

		if (ivec[i] < topData)
		{
			break;
		}
		std::swap(ivec[top], ivec[i]);
		top = i;
		++counter;

	}
}
```

3.7 归并排序分析

基本描述

算法平均时间复杂度	$O(nlog_2(n))$
算法最坏情况时间复杂度	$O(nlog_2(n))$
算法最好情况时间复杂度	$O(nlog_2(n))$
算法空间复杂度	$O(n)$
是否为原地(In-Place)算法	不是
算法稳定性	稳定

图解



算法的分析

归并排序是一种稳定的以分治思想为核心的排序算法。它将待排序元素序列分为两个长度相等的子序列，并对每个子序列执行相同部分，直到所有子序列不可再被划分。之后执行归并操作，即将分开的一对子序列合并成有序的子序列。在合并的过程中，两个子序列是非降序的，故可以使用双指针法，将当前两个子序列中最小的元素（即双指针中元素较小的指针所指向的元素），这样通过双指针遍历两个子序列的方式可以输出一个有序序列，此后对有序序列再进行相同的归并操作，最终可以得到完整的排序好的数组。在双指针遍历的过程中，需要另外开辟数组存储排序好的数据，对于整个参与归并排序的数组而言，需要开辟一个与原数组相同大小的数组存储结果，故归并排序并非原地排序算法。

排序算法代码段

```
void SortCompare::_mergeSort(int* ivec, long long& counter)
{
	int* tmpArr = new int[_arraySize];
	for (int i = 0; i < _arraySize; i++)
	{
		tmpArr[i] = ivec[i];
	}

	_mSort(ivec, tmpArr, 0, _arraySize - 1, counter);
}

void SortCompare::_mSort(int* ivec, int* tmp,int low, int high, long lo
ng& counter)
{
	if (high <= low) return;

	int mid = low + (high - low) / 2;
	_mSort(ivec,tmp, low, mid,counter);
	_mSort(ivec,tmp, mid + 1, high, counter);
	_merge(ivec,tmp, low, mid, high, counter);
}

void SortCompare::_merge(int* ivec, int* tmp,int low, int mid, int high,
 long long &counter)
{
	int i = low, j = mid + 1;
	int t= 0;
	while (i <= mid && j <= high)
	{
		if (ivec[i] < ivec[j])
		{
			tmp[t++] = ivec[i++];
			counter++;
```

```
            }
            else
            {
                  tmp[t++] = ivec[j++];
                  counter++;
            }
      }
      while (i <= mid)  {     tmp[t++] = ivec[i++]; counter++; }
      while (j <= high) { tmp[t++] = ivec[j++]; counter++; }
      t = 0;
      while (low <= high) {ivec[low++] = tmp[t++]; counter++;}
}
```

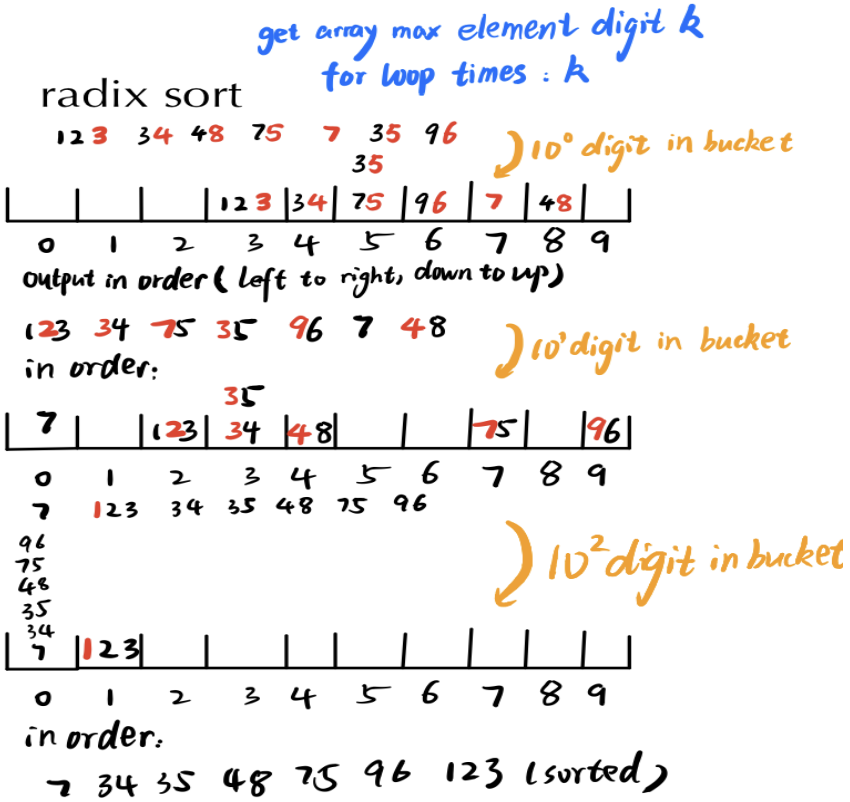
3.8 基数排序分析

基本描述

(注：d 为数组中最大元素的位数；r 是基数，在十进制数据存储数组中为 10；n 是比较的次数)

算法平均时间复杂度	$O(d*(n+r))$
算法最坏情况时间复杂度	$O(d*(n+r))$
算法最好情况时间复杂度	$O(d*(n+r))$
算法空间复杂度	$O(n+r*d)$
是否为原地(In-Place)算法	不是
算法稳定性	稳定

图解



算法的分析

基数排序是一种非比较排序算法。它通过根据元素的基数创建和分配元素到存储桶中来避免比较。对于有一个以上有效数字的元素，这个存储过程对每个数字都重复，同时保持前一个步骤的顺序，直到所有数字都被最终考虑。在算法运行时需要多分配一个大小为基数的桶数组（十进制运算中分别对应 0-9）之后从个位开始向高位执行排序，对待排序的数，根据对应位数的数字放入数组中。最高位为数组中最大一个数字的最高位，在位数小于最高位的数认为该位为 0。

排序算法代码段

```
void SortCompare::_radixSort(int* ivec, long long& counter)
{
	//找到最大数
	int length = _arraySize;
	int maxElement = ivec[0];
	for (int i = 1; i < length; i++)
	{
		if (ivec[i] > maxElement)
		{
			maxElement = ivec[i];
		}
	}
	int i = 0;
	for (int bit = 1; maxElement / bit > 0; bit *= 10)
	{
		int bucket[10] = { 0 };
		int* sortArr = new int[length];
		for (i = 0; i < length; i++)
		{
			int bitValue = (ivec[i] / bit) % 10;
			bucket[bitValue]++;
		}
		for ( i = 1; i < 10; i++)
		{
			bucket[i] += bucket[i - 1];
		}

		for (i = length - 1; i >= 0; i--)
		{
			int bitValue = (ivec[i] / bit) % 10;
			sortArr[bucket[bitValue] - 1] = ivec[i];
			bucket[bitValue]--;
		}

		for (int i = 0; i < length; i++)
		{
			ivec[i] = sortArr[i];
		}
```

```
            delete[]sortArr;
        }
}
```

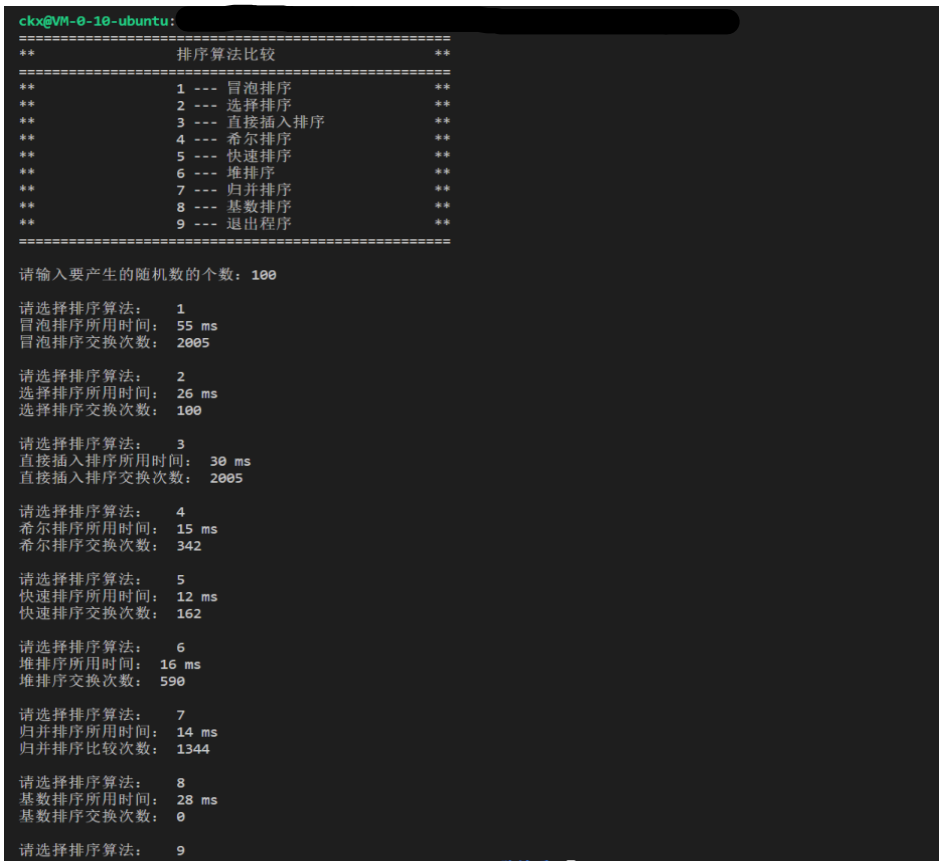
4.测试

4.1 常规结果测试

由于生成可执行文件时采用 linux 远程云服务器，最终在本地测评的时间运算结果可能与以下测算结果有所不同。

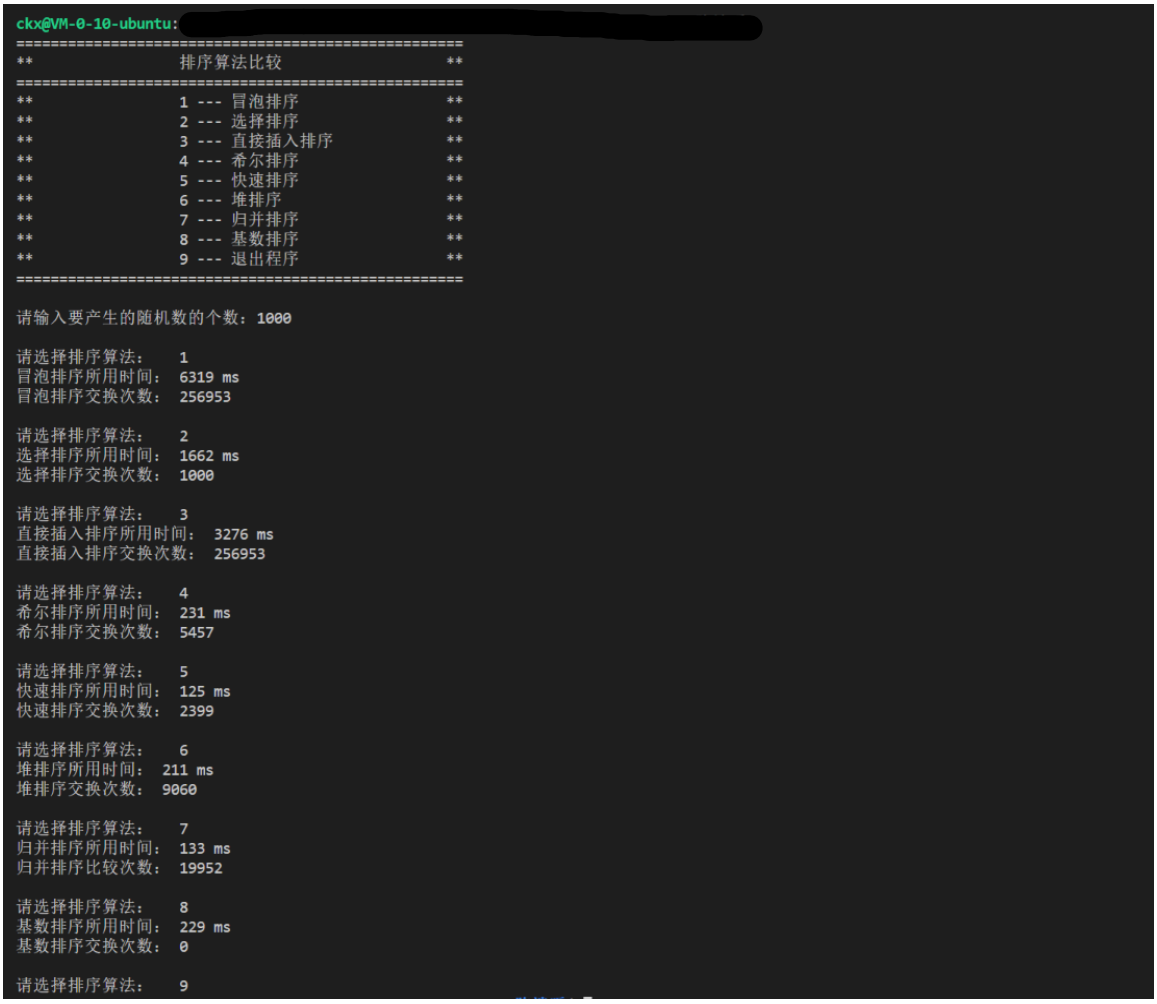
Task 1 生成 100 个随机数进行排序测试

实验结果：（耗时：快速排序<归并排序<希尔排序<堆排序<基数排序<选择排序<直接插入排序<冒泡排序）



Task 2 生成 1000 个随机数进行排序测试

实验结果：（耗时：快速排序<归并排序<堆排序<基数排序<希尔排序<选择排序<直接插入排序<冒泡排序）



Task 3 生成 10000 个随机数进行排序测试

实验结果：（耗时：快速排序<归并排序<基数排序<堆排序<希尔排序<选择排序<直接插入排序<冒泡排序）

```
ckx@VM-0-10-ubuntu:[redacted]
================================================
**                排序算法比较                **
================================================
**                1 --- 冒泡排序              **
**                2 --- 选择排序              **
**                3 --- 直接插入排序          **
**                4 --- 希尔排序              **
**                5 --- 快速排序              **
**                6 --- 堆排序                **
**                7 --- 归并排序              **
**                8 --- 基数排序              **
**                9 --- 退出程序              **
================================================

请输入要产生的随机数的个数：10000

请选择排序算法：    1
冒泡排序所用时间：  595534 ms
冒泡排序交换次数：  24776961

请选择排序算法：    2
选择排序所用时间：  158156 ms
选择排序交换次数：  10000

请选择排序算法：    3
直接插入排序所用时间：  312405 ms
直接插入排序交换次数：  24776961

请选择排序算法：    4
希尔排序所用时间：  3475 ms
希尔排序交换次数：  75243

请选择排序算法：    5
快速排序所用时间：  1512 ms
快速排序交换次数：  31568

请选择排序算法：    6
堆排序所用时间：  2660 ms
堆排序交换次数：  124152

请选择排序算法：    7
归并排序所用时间：  1667 ms
归并排序比较次数：  267232

请选择排序算法：    8
基数排序所用时间：  2215 ms
基数排序交换次数：  0

请选择排序算法：    9
```

Task 4 生成 100000 个随机数进行排序测试

实验结果：（耗时：快速排序<归并排序<基数排序<堆排序<希尔排序<选择排序<直接插入排序<冒泡排序）