同济大学

TONGJI UNIVERSITY

数据结构课程设计

项目名称	8 种排序算法的比较案例
学　　院	计算机科学与技术学院
专　　业	软件工程
学生姓名	杨瑞晨
学　　号	2351050
指导教师	张颖
日　　期	2024 年 12 月 20 日

目 录

1 项目分析

1.1 项目背景分析

本项目旨在通过多种排序算法对随机数数组进行排序，并对每种排序方法的时间复杂度和交换次数进行统计和比较。用户可以自定义随机数的个数，系统将根据用户输入生成指定数量的随机数。排序算法包括快速排序、直接插入排序、冒泡排序和选择排序等。项目通过 C++ 语言实现，运用了面向对象编程和数据结构的基础知识。

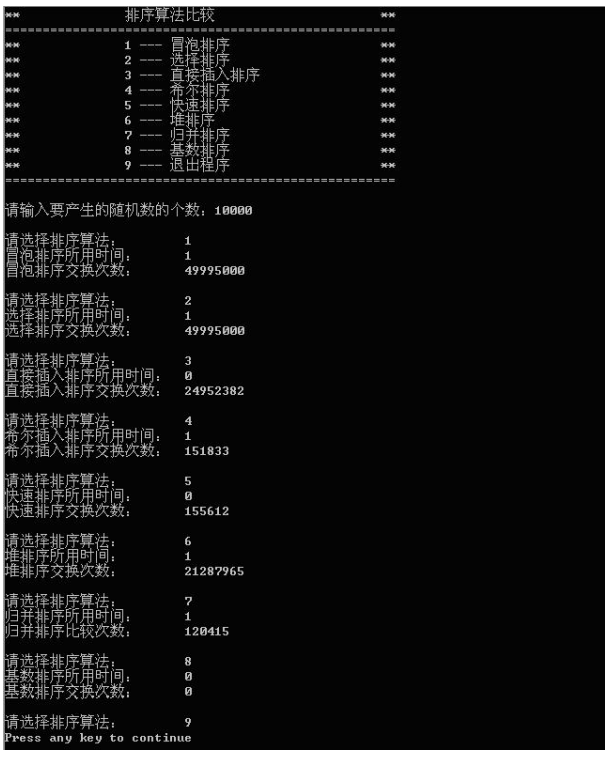
1.2 项目功能分析

1.2.1 功能要求

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。

1.2.2 项目实例



2 项目设计

2.1 类设计

Sort 类

Sort 类是本项目的核心，它封装了排序算法的实现，并提供了一个统一的接口来比较不同排序算法的性能。通过动态分配内存来存储用户定义大小的数据，并记录排序过程中的关键性能指标。

分析过程：

- 需要一个类来封装排序算法的公共行为，包括数据的存储、排序操作的执行以及结果的输出。
- 为了比较不同排序算法的性能，需要记录排序过程中的比较次数和执行时间。
- 用户需要能够自定义数据的大小，因此数据结构需要动态分配内存以存储随机生成的数据。

提供的接口：

- 构造函数和析构函数：初始化和释放动态分配的内存。
- swap：交换两个整数的值。
- reset：重置排序前的状态，将 currentData 恢复为 originalData 并清零计数器。
- print：打印数组。
- output：输出排序结果，包括所用时间和交换次数。
- 各种排序算法的实现函数：冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序和基数排序。

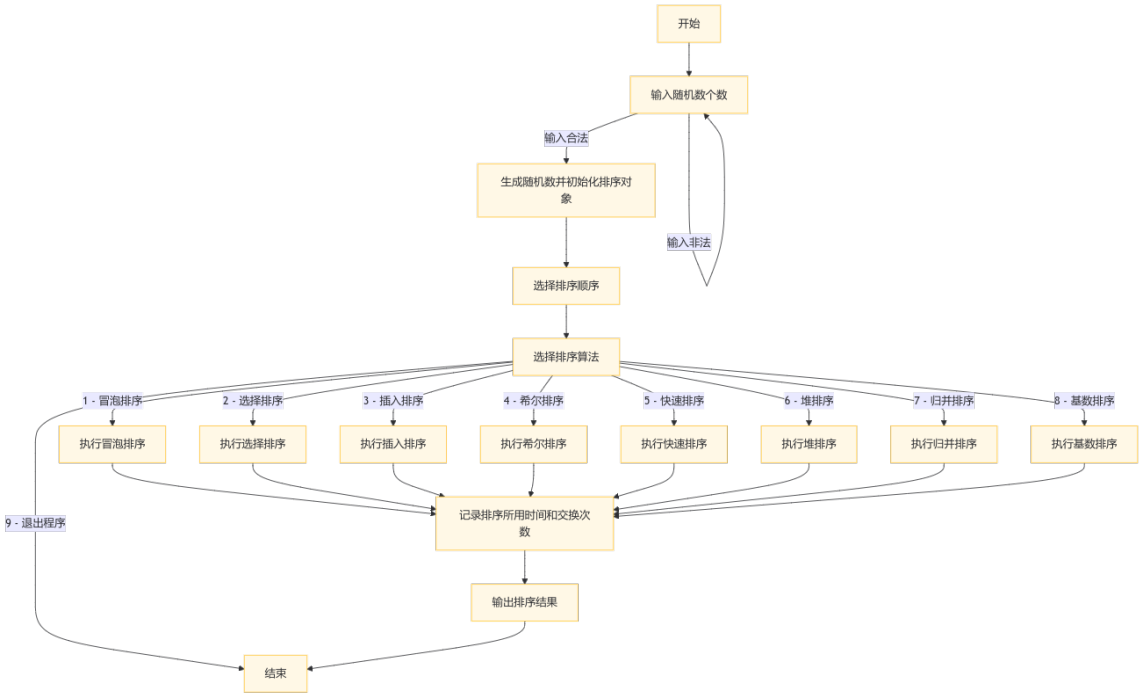
代码实现：

```
clock_t startTime, endTime;
class Sort
{
public:
    int size;          // 数据大小
    int count;         // 比较次数
    int *originalData; // 原始数据
    int *currentData;  // 用于排序的数据副本

    int getDigit();                                                          // 获取最大数字的位数
    void merge(bool (*comp)(int, int), int *arr, int left, int mid, int right); // 归并排序的合并操作
    int partition(int *arr, int left, int right, bool (*comp)(int, int));       // 快速排序的分区操作
    void quick_sort(bool (*comp)(int, int), int left, int right);               // 快速排序的递归操作
    void heapify(int *arr, int n, int i, bool (*comp)(int, int));               // 维护堆性质
public:
    Sort(int n) : size(n), count(0)
    {
        originalData = new int[size];
        currentData = new int[size];
        srand(static_cast<unsigned int>(time(nullptr))); // 生成随机数种子
        for (int i = 0; i < size; i++)
```

```
22              originalData[i] = currentData[i] = rand(); // 生成随机数
23      }
24
25      ~Sort()
26      {
27          delete[] originalData;
28          delete[] currentData;
29      }
30
31      void swap(int &a, int &b)
32      {
33          if (a == b)
34              return;
35          int temp = a;
36          a = b;
37          b = temp;
38      }
39
40      // 重置数据：将 `currentData` 恢复为 `originalData`，并清零计数器
41      void reset()
42      {
43          count = 0;
44          for (int i = 0; i < size; i++)
45              currentData[i] = originalData[i];
46      }
47
48      void print(int *arr)
49      {
50          for (int i = 0; i < size; i++)
51              cout << arr[i] << " ";
52          cout << endl;
53      }
54
55      void output(const char *prompt, ostream &os = cout)
56      {
57          os << left;
58          os << setw(20) << prompt << " 所用时间：" << static_cast<double>(endTime - startTime)
            ↪  / CLOCKS_PER_SEC * 1000 << " ms" << endl;
59          os << setw(20) << prompt << " 交换次数：" << count << endl;
60          os << endl;
61      }
62
63      void bubble_sort(bool (*comp)(int, int));                    // 冒泡排序
64      void select_sort(bool (*comp)(int, int));                    // 选择排序
65      void insert_sort(bool (*comp)(int, int));                    // 插入排序
66      void shell_sort(bool (*comp)(int, int));                     // 希尔排序
67      void merge_sort(bool (*comp)(int, int), int left, int right); // 归并排序
68      void quick_sort(bool (*comp)(int, int));                     // 快速排序
69      void heap_sort(bool (*comp)(int, int));                      // 堆排序
70      void radix_sort();                                           // 基数排序
71  };
72
```

2.2 系统流程设计


开始
输入随机数个数
输入合法
生成随机数并初始化排序对象
输入非法
选择排序顺序
选择排序算法
1 - 冒泡排序
2 - 选择排序
3 - 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
执行冒泡排序
执行选择排序
执行插入排序
执行希尔排序
执行快速排序
执行堆排序
执行归并排序
执行基数排序
9 - 退出程序
记录排序所用时间和交换次数
输出排序结果
结束

3 项目实现

所谓排序，就是使一串记录，按照其中的某个或某些关键字的大小，递增或递减的排列起来的操作。排序算法，就是如何使得记录按照要求排列的方法。排序算法在很多领域得到相当地重视，尤其是在大量数据的处理方面。一个优秀的算法可以节省大量的资源。在各个领域中考虑到数据的各种限制和规范，要得到一个符合实际的优秀算法，得经过大量的推理和分析。

3.1 简介

常见的内部排序算法有：插入排序、希尔排序、选择排序、冒泡排序、归并排序、快速排序、堆排序、基数排序等，本文只讲解内部排序算法。用一张表格概括：

表 3.1 常见排序算法的时间复杂度和空间复杂度

排序算法	时间复杂度（平均）	时间复杂度（最差）	时间复杂度（最好）	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(\log n)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n+k)$	$O(n+k)$	稳定
基数排序	$O(n\times k)$	$O(n\times k)$	$O(n\times k)$	$O(n+k)$	稳定

术语解释：

- n：数据规模，表示待排序的数据量大小。
- k：“桶”的个数，在某些特定的排序算法中（如基数排序、桶排序等），表示分割成的独立的排序区间或类别的数量。
- 内部排序：所有排序操作都在内存中完成，不需要额外的磁盘或其他存储设备的辅助。这适用于数据量小到足以完全加载到内存中的情况。
- 外部排序：当数据量过大，不可能全部加载到内存中时使用。外部排序通常涉及到数据的分区处理，部分数据被暂时存储在外部磁盘等存储设备上。
- 稳定：如果 A 原本在 B 前面，而 $A = B$，排序之后 A 仍然在 B 的前面。
- 不稳定：如果 A 原本在 B 的前面，而 $A = B$，排序之后 A 可能会出现在 B 的后面。
- 时间复杂度：定性描述一个算法执行所耗费的时间。
- 空间复杂度：定性描述一个算法执行所需内存的大小。

3.2 排序算法分类

十种常见排序算法可以分类两大类别：比较类排序和非比较类排序。

比较类排序：常见的快速排序、归并排序、堆排序以及冒泡排序等都属于比较类排序算法。比较类排序是通过比较来决定元素间的相对次序，由于其时间复杂度不能突破 $O(n \log n)$，因此也称为非线性时间比较类排序。在冒泡排序之类的排序中，问题规模为 n，又因为需要比较 n 次，所以平均时间复杂度为 $O(n^2)$。在归并排序、快速排序之类的排序中，问题规模通过分治法消减为 $\log n$ 次，所以时间复杂度平均 $O(n \log n)$。

比较类排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况。

非比较类排序：计数排序、基数排序、桶排序则属于非比较类排序算法。非比较排序不通过比较来决定元素间的相对次序，而是通过确定每个元素之前，应该有多少个元素来排序。由于它可以突破基于比较排序的时间下界，以线性时间运行，因此称为线性时间非比较类排序。非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。算法时间复杂度 $O(n)$。

非比较排序时间复杂度低，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。

3.3 排序算法实现

3.3.1 冒泡排序 (Bubble Sort)

冒泡排序是一种简单的排序算法。它重复地遍历要排序的序列，依次比较两个元素，如果它们的顺序错误就把它们交换过来。遍历序列的工作是重复地进行直到没有再需要交换为止，此时说明该序列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

算法步骤：

（1）比较相邻的元素。如果第一个比第二个大，就交换它们两个；

（2）对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

（3）针对所有的元素重复以上的步骤，除了最后一个；

（4）重复步骤 1 3，直到排序完成。

代码实现：

```
// 冒泡排序
void Sort::bubble_sort(bool (*comp)(int, int))
{
    reset();
    for (int i = 0; i < size - 1; ++i)
    {
        for (int j = 0; j < size - i - 1; ++j)
```

```
        {
            if (comp(currentData[j], currentData[j + 1]))
            {
                swap(currentData[j], currentData[j + 1]);
                count++;
            }
        }
    }
}
```

算法分析：

- 稳定性：稳定
- 时间复杂度：最佳：$O(n)$，最差：$O(n^2)$，平均：$O(n^2)$
- 空间复杂度：$O(1)$
- 排序方式：In-place

3.3.2 选择排序 (Selection Sort)

选择排序是一种简单直观的排序算法，无论什么数据进去都是 $O(n^2)$ 的时间复杂度。所以用到它的时候，数据规模越小越好。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

算法步骤：

（1）首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置；

（2）再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾；

（3）重复第 2 步，直到所有元素均排序完毕。

代码实现：

```
// 选择排序
void Sort::select_sort(bool (*comp)(int, int))
{
    reset();
    for (int i = 0; i < size - 1; ++i)
    {
        int min_index = i;
        for (int j = i + 1; j < size; ++j)
        {
            if (comp(currentData[min_index], currentData[j]))
                min_index = j;
            count++;
        }
        if (min_index != i)
        {
            swap(currentData[i], currentData[min_index]);
        }
    }
```

```
}
```

算法分析：

- 稳定性：不稳定
- 时间复杂度：最佳：$O(n^2)$，最差：$O(n^2)$，平均：$O(n^2)$
- 空间复杂度：$O(1)$
- 排序方式：In-place

3.3.3 **插入排序** (Insertion Sort)

插入排序是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用 in-place 排序（即只需用到 $O(1)$ 的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

插入排序是一种最简单直观的排序算法，它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

算法步骤：

（1）从第一个元素开始，该元素可以认为已经被排序；

（2）取出下一个元素，在已经排序的元素序列中从后向前扫描；

（3）如果该元素（已排序）大于新元素，将该元素移到下一位置；

（4）重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置；

（5）将新元素插入到该位置后；

（6）重复步骤 2-5。

代码实现：

```
// 插入排序
void Sort::insert_sort(bool (*comp)(int, int))
{
    reset();
    for (int i = 1; i < size; ++i)
    {
        int temp = currentData[i];
        int j = i - 1;
        while (j >= 0 && comp(currentData[j], temp))
        {
            currentData[j + 1] = currentData[j];
            count++;
            j--;
        }
        currentData[j + 1] = temp;
    }
}
```

算法分析：

- 稳定性：稳定
- 时间复杂度：最佳：$O(n)$，最差：$O(n^2)$，平均：$O(n^2)$
- 空间复杂度：$O(1)$
- 排序方式：In-place

3.3.4 希尔排序 (Shell Sort)

希尔排序是希尔 (Donald Shell) 于 1959 年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为递减增量排序算法，同时该算法是冲破 $O(n^2)$ 的第一批算法之一。

希尔排序的基本思想是：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

算法步骤：我们来看下希尔排序的基本步骤，在此我们选择增量 $gap = length/2$，缩小增量继续以 $gap = gap/2$ 的方式，这种增量选择我们可以用一个序列来表示，$\{\frac{n}{2}, \frac{(n/2)}{2}, \ldots, 1\}$，称为增量序列。希尔排序的增量序列的选择与证明是个数学难题，我们选择的这个增量序列是比较常用的，也是希尔建议的增量，称为希尔增量，但其实这个增量序列不是最优的。此处我们做示例使用希尔增量。

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

（1）选择一个增量序列 $\{t_1, t_2, \ldots, t_k\}$，其中 $t_i > t_j, i < j, t_k = 1$；

（2）按增量序列个数 k，对序列进行 k 趟排序；

（3）每趟排序，根据对应的增量 t，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

代码实现：

```
// 希尔排序
void Sort::shell_sort(bool (*comp)(int, int))
{
    reset();
    for (int gap = size / 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < size; ++i)
        {
            int temp = currentData[i];
            int j;
            for (j = i; j >= gap && comp(currentData[j - gap], temp); j -= gap)
            {
                currentData[j] = currentData[j - gap];
                count++;
            }
            currentData[j] = temp;
        }
    }
```

```
}
```

算法分析：

- 稳定性：不稳定
- 时间复杂度：最佳：$O(n\log n)$，最差：$O(n^2)$ 平均：$O(n\log n)$
- 空间复杂度：$O(1)$

3.3.5 归并排序 (Merge Sort)

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法 (Divide and Conquer) 的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为 2 - 路归并。

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是 $O(n\log n)$ 的时间复杂度。代价是需要额外的内存空间。

算法步骤：归并排序算法是一个递归过程，边界条件为当输入序列仅有一个元素时，直接返回，具体过程如下：

（1）如果输入内只有一个元素，则直接返回，否则将长度为 n 的输入序列分成两个长度为 $n/2$ 的子序列；

（2）分别对这两个子序列进行归并排序，使子序列变为有序状态；

（3）设定两个指针，分别指向两个已经排序子序列的起始位置；

（4）比较两个指针所指向的元素，选择相对小的元素放入到合并空间（用于存放排序结果），并移动指针到下一位置；

（5）重复步骤 3 - 4 直到某一指针达到序列尾；

（6）将另一序列剩下的所有元素直接复制到合并序列尾。

代码实现：

```
// 归并排序的合并操作
void Sort::merge(bool (*comp)(int, int), int *arr, int left, int mid, int right)
{
    int subArr1 = mid - left + 1;
    int subArr2 = right - mid;
    // 创建临时数组
    int *leftArr = new int[subArr1];
    int *rightArr = new int[subArr2];
    for (int i = 0; i < subArr1; i++)
        leftArr[i] = arr[left + i];
    for (int i = 0; i < subArr2; i++)
        rightArr[i] = arr[mid + 1 + i];

```

```
    // 合并两个子数组
    int i = 0, j = 0, k = left;
    while (i < subArr1 && j < subArr2)
    {
        if (!comp(leftArr[i], rightArr[j]))
        {
            arr[k++] = leftArr[i++];
        }
        else
        {
            arr[k++] = rightArr[j++];
        }
        count++;
    }
    // 将剩余元素复制到 arr
    while (i < subArr1)
    {
        arr[k++] = leftArr[i++];
    }
    while (j < subArr2)
    {
        arr[k++] = rightArr[j++];
    }

    delete[] leftArr;
    delete[] rightArr;
}

// 归并排序
void Sort::merge_sort(bool (*comp)(int, int), int left, int right)
{
    static bool firstCall = true;
    if (firstCall)
    {
        reset();
        firstCall = false;
    }

    if (left < right)
    {
        int mid = left + (right - left) / 2;
        merge_sort(comp, left, mid);
        merge_sort(comp, mid + 1, right);
        merge(comp, currentData, left, mid, right);
    }
}
```

算法分析：

- 稳定性：稳定
- 时间复杂度：最佳：$O(n \log n)$，最差：$O(n \log n)$，平均：$O(n \log n)$
- 空间复杂度：$O(n)$

3.3.6 **快速排序** (Quick Sort)

快速排序用到了分治思想，同样的还有归并排序。乍看起来快速排序和归并排序非常相似，都是将问题变小，先排序子串，最后合并。不同的是快速排序在划分子问题的时候经过多一步处理，将划分的两组数据划分为一大一小，这样在最后合并的时候就不必像归并排序那样再进行比较。但也正因为如此，划分的不定性使得快速排序的时间复杂度并不稳定。

快速排序的基本思想：通过一趟排序将待排序列分隔成独立的两部分，其中一部分记录的元素均比另一部分的元素小，则可分别对这两部分子序列继续进行排序，以达到整个序列有序。

算法步骤：快速排序使用分治法（Divide and conquer）策略来把一个序列分为较小和较大的 2 个子序列，然后递归地排序两个子序列。具体算法描述如下：

（1）从序列中随机挑出一个元素，做为“基准”（ pivot ）；

（2）重新排列序列，将所有比基准值小的元素摆放在基准前面，所有比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个操作结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

（3）递归地把小于基准值元素的子序列和大于基准值元素的子序列进行快速排序。

代码实现：

```
// 快速排序
void Sort::quick_sort(bool (*comp)(int, int))
{
    reset();
    quick_sort(comp, 0, size - 1);
}

// 快速排序的分区操作
int Sort::partition(int *arr, int left, int right, bool (*comp)(int, int))
{
    int pivot = arr[right];
    int i = left - 1;
    for (int j = left; j < right; j++)
    {
        if (comp(pivot, arr[j]))
        {
            i++;
            swap(arr[i], arr[j]);
            count++;
        }
    }
    swap(arr[i + 1], arr[right]);
    count++;
    return i + 1;
}

void Sort::quick_sort(bool (*comp)(int, int), int left, int right)
{
    if (left < right)
    {
        int pi = partition(currentData, left, right, comp);
```

```
        quick_sort(comp, left, pi - 1);
        quick_sort(comp, pi + 1, right);
    }
}
```

算法分析：

- 稳定性：不稳定
- 时间复杂度：最佳：$O(n\log n)$，最差：$O(n^2)$，平均：$O(n\log n)$
- 空间复杂度：$O(\log n)$

3.3.7 **堆排序** (Heap Sort)

堆排序是指利用堆这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构，并同时满足堆的性质：即子结点的值总是小于（或者大于）它的父节点。

算法步骤：

（1）将初始待排序列 $(R_1, R_2, \ldots, R_n)$ 构建成大顶堆，此堆为初始的无序区；

（2）将堆顶元素 R_1 与最后一个元素 R_n 交换，此时得到新的无序区 $(R_1, R_2, \ldots, R_{n-1})$ 和新的有序区 R_n，且满足 $R_i \leqslant R_n (i \in 1, 2, \ldots, n-1)$；

（3）由于交换后新的堆顶 R_1 可能违反堆的性质，因此需要对当前无序区 $(R_1, R_2, \ldots, R_{n-1})$ 调整为新堆，然后再次将 R_1 与无序区最后一个元素交换，得到新的无序区 $(R_1, R_2, \ldots, R_{n-2})$ 和新的有序区 (R_{n-1}, R_n)。不断重复此过程直到有序区的元素个数为 $n-1$，则整个排序过程完成。

代码实现：

```
// 维护堆性质
void Sort::heapify(int *arr, int n, int i, bool (*comp)(int, int))
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && comp(arr[left], arr[largest]))
        largest = left;
    if (right < n && comp(arr[right], arr[largest]))
        largest = right;
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        count++;
        heapify(arr, n, largest, comp);
    }
}

// 堆排序
void Sort::heap_sort(bool (*comp)(int, int))
{
    reset();
    for (int i = size / 2 - 1; i >= 0; --i)
```

```
        heapify(currentData, size, i, comp);
    for (int i = size - 1; i > 0; --i)
    {
        swap(currentData[0], currentData[i]);
        count++;
        heapify(currentData, i, 0, comp);
    }
}
```

算法分析：

- 稳定性：不稳定
- 时间复杂度：最佳：$O(n\log n)$，最差：$O(n\log n)$，平均：$O(n\log n)$
- 空间复杂度：$O(1)$

3.3.8 基数排序 (Radix Sort)

基数排序也是非比较的排序算法，对元素中的每一位数字进行排序，从最低位开始排序，复杂度为 $O(n\times k)$，n 为数组长度，k 为数组中元素的最大的位数。

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

算法步骤：

（1）取得数组中的最大数，并取得位数，即为迭代次数 N（例如：数组中最大数值为 1000，则 $N=4$）；

（2）A 为原始数组，从最低位开始取每个位组成 radix 数组；

（3）对 radix 进行计数排序（利用计数排序适用于小范围数的特点）；

（4）将 radix 依次赋值给原数组；

（5）重复 2 4 步骤 N 次。

代码实现：

```
// 获取最大数字的位数
int Sort::getDigit()
{
    int res = 1;
    int maxNum = originalData[0];
    for (int i = 1; i < size; i++)
    {
        if (originalData[i] > maxNum)
            maxNum = originalData[i];
    }

    while (maxNum /= 10)
        res++;
```

```cpp

    return res;
}

// 基数排序
void Sort::radix_sort()
{
    reset();

    int digit = getDigit();
    int radix = 1;

    int *countNum = new int[10]; // 计数数组
    int *output = new int[size]; // 输出数组

    for (int i = 0; i < digit; i++)
    {
        // 初始化计数数组
        for (int j = 0; j < 10; j++)
            countNum[j] = 0;
        // 统计当前位的数字频率
        for (int j = 0; j < size; j++)
        {
            countNum[(currentData[j] / radix) % 10]++;
            count++;
        }
        // 累计计数，确定结束位置
        for (int j = 1; j < 10; j++)
            countNum[j] += countNum[j - 1];
        // 根据当前位数字，将数据放到输出数组中
        for (int j = size - 1; j >= 0; j--)
            output[--countNum[(currentData[j] / radix) % 10]] = currentData[j];
        // 排序结果拷贝到原数组
        for (int j = 0; j < size; j++)
            currentData[j] = output[j];

        radix *= 10;
    }

    delete[] countNum;
    delete[] output;
}
```

算法分析：

- 稳定性：稳定
- 时间复杂度：最佳：$O(n \times k)$，最差：$O(n \times k)$，平均：$O(n \times k)$
- 空间复杂度：$O(n + k)$

3.4 排序算法适用场景

表 3.2 8 种排序算法的适用场景

排序算法	适用场景
冒泡排序	数据量小的情况下，或需要简单的实现
选择排序	数据量小的情况下，或需要简单实现的场合
插入排序	数据量小或基本有序的数组，适合实时插入
希尔排序	中等规模的数组，尤其是对部分有序的数据进行优化
归并排序	大规模数据的排序，尤其是在需要稳定排序的情况下
快速排序	大规模数据的排序，尤其是当内存空间有限时
堆排序	大规模数据的排序，尤其是内存限制较小的情况下
基数排序	大规模整数或字符串排序，特别是当数据范围较小且已知时

4 项目测试

在题目原有要求下，实现了自定义比较器，可以选择从小到大排序还是从大到小排序。测试结果如下：

```
**            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(只支持从小到大)  **
**        9---退出程序                  **
========================================
请输入要产生的随机数的个数：666
请选择从小到大排序还是从大到小排序(输入 L/l 代表从小到大排序，G/g 代表从大到小排序)：g
请选择排序算法：1
冒泡排序        所用时间：2.869 ms
冒泡排序        交换次数：109564

请选择排序算法：2
选择排序        所用时间：2.022 ms
选择排序        交换次数：221445

请选择排序算法：3
直接插入排序   所用时间：1.106 ms
直接插入排序   交换次数：109564

请选择排序算法：4
希尔排序        所用时间：0.179 ms
希尔排序        交换次数：4624

请选择排序算法：5
快速排序        所用时间：0.126 ms
快速排序        交换次数：3408

请选择排序算法：6
堆排序           所用时间：0.234 ms
堆排序           交换次数：5676

请选择排序算法：7
归并排序        所用时间：0.23 ms
归并排序        交换次数：5432

请选择排序算法：8
基数排序        所用时间：0.179 ms
基数排序        交换次数：6660

请选择排序算法：9
```

```
**            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(只支持从小到大)  **
**        9---退出程序                  **
============================================
请输入要产生的随机数的个数：10000
请选择从小到大排序还是从大到小排序(输入 L/l 代表从小到大排序，G/g 代表从大到小排序)：L
请选择排序算法：1
冒泡排序        所用时间：308.58 ms
冒泡排序        交换次数：24907457

请选择排序算法：2
选择排序        所用时间：155.936 ms
选择排序        交换次数：49995000

请选择排序算法：3
直接插入排序    所用时间：97.754 ms
直接插入排序    交换次数：24907457

请选择排序算法：4
希尔排序        所用时间：4.057 ms
希尔排序        交换次数：150723

请选择排序算法：5
快速排序        所用时间：2.488 ms
快速排序        交换次数：86866

请选择排序算法：6
堆排序          所用时间：4.825 ms
堆排序          交换次数：124213

请选择排序算法：7
归并排序        所用时间：3.089 ms
归并排序        交换次数：120489

请选择排序算法：8
基数排序        所用时间：1.558 ms
基数排序        交换次数：100000

请选择排序算法：9
```

图 4.1 8 种排序算法的比较案例

5 项目心得与体会

通过参与本项目的设计与实现，我对排序算法有了更深刻的理解和实践。以下是我的主要心得与总结:

通过对八种排序算法的实现和比较，我对每种算法的原理、时间复杂度、空间复杂度以及稳定性有了更全面的认识。我学习了如何根据不同的数据规模和数据特性选择合适的排序算法，以及如何评估算法在不同场景下的性能；在实现这些排序算法的过程中，我的编程能力得到了显著提升。我学会了如何编写清晰、高效且易于维护的代码，并且掌握了调试技巧和性能分析方法。这些技能对我的未来编程工作大有裨益。

总之，这个项目不仅让我掌握了数据结构和算法的具体应用，还提升了我的编程能力、问题解决能力和系统设计能力。这些经验和技能将为我未来的学习和工作奠定坚实的基础。