# 引言

。。。-

# 排序算法的实验研究

## 测试数据生成

周

## 排序算法的实现、优化及测试

。。。（多文件结构，函数调用方便；针对不同类型使用template和auto；。。。）-卢

#### 冒泡排序

###### 基础冒泡排序实现

冒泡排序是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。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。

**实现步骤：**

Step1：比较相邻的元素。如果第一个比第二个大，就交换他们两个。

Step2：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。这步做完后，最后的元素会是最大的数。

Step3：针对所有的元素重复以上的步骤，除了最后一个。

Step4：持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

**代码展示：**

template <typename T>

void BubbleSort\_sorted(T begin, T end) //模仿sort()函数的输入参数

{

    int n = (end - begin); //获取数据长度, n为实际长度-1

    // std::cout << \*begin;

    bool sorted = false;                        //(sorted优化)判断是否已经有序

    for (int i = 0; i <= n - 1 && !sorted; i++) //从首数据开始遍历，直到倒数第二个数据为止，若有序则结束循环

    {

        for (int j = 0, sorted = true; j <= n - i - 1; j++) //内循环控制排序的数据交换，若有序则结束循环

        {

            if (\*(begin + j) > \*(begin + j + 1))

            {

                auto tmp = \*(begin + j + 1);

                \*(begin + j + 1) = \*(begin + j);

                \*(begin + j) = tmp;

                sorted = false; // (sorted优化)

            }

        }

    }

}

###### 优化1：鸡尾酒排序

鸡尾酒排序，即双向的冒泡排序，等于是冒泡排序的轻微变形。不同的地方在于从低到高然后从高到低（有先后顺序，并非同时；大循环下第一个循环是从开始扫到结束，将最大的归到最后；第二个循环是从倒数第二个位置往开始端扫，将最小的归到开始的位置），而冒泡排序则仅仅从低到高去比较序列里的每个元素。它可以得到比冒泡排序稍微好一点的效能，原因是冒泡排序只从一个方向进行比对（由低到高），每次只移动一个项目和。

**实现步骤：**

Step1：先对数组从左到右进行冒泡排序（升序），则最大的元素去到最右端

Step2：再对数组从右到左进行冒泡排序（降序），则最小的元素去到最左端

Step3：以此类推，依次改变冒泡的方向，并不断缩小未排序元素的范围，直到最后一个元素结束

**代码展示：**

template <typename T>

void BubbleSort\_Cocktail(T begin, T end) //模仿sort()函数的输入参数

{

    int n = end - begin; //获取数据长度, n为实际长度-1

    int left = 0, right = n;

    while (left < right)

    {

        bool sorted = true;                     //(sorted优化)

        for (int i = right - 1; i >= left; i--) //从右向左做冒泡排序

        {

            if (\*(begin + i) > \*(begin + i + 1))

            {

                exchange(begin + i, begin + i + 1);

                sorted = false; //(sorted优化)

            }

        }

        left++; //每从右向左排一次序，左侧有序数列+1，右移left

        for (int i = left + 1; i <= right; i++) //从左向右做冒泡排序

        {

            if (\*(begin + i) < \*(begin + i - 1))

            {

                exchange(begin + i, begin + i - 1);

                sorted = false; //(sorted优化)

            }

        }

        right--;    //每从左向右排一次序，右侧有序数列+1，左移right

        if (sorted) //(sorted优化)如果已经有序，则提前跳出循环

            break;

    }

}

###### 优化2：梳排序

梳排序改良自冒泡排序和快速排序，其要旨在于消除乌龟，亦即在数组尾部的小数值，这些数值是造成泡沫排序缓慢的主因。相对地，兔子，亦即在数组前端的大数值，不影响泡沫排序的性能。梳排序通过将间隔固定项(Gap)的数据做冒泡排序，来快速将乌龟移动到数组前端以实现消除乌龟。而后缩小Gap继续分组进行冒泡排序，其实现过程与插入排序的变体希尔排序相类似。

**实现步骤：**

Step1：定义一个步长(Gap)，该步长为数组的长度除以步长系数（1.3）

Step2：第0个元素开始，跟距离当前元素为步长大小的元素进行比较，若前者大于后者就进行交换，否则继续往后进行比较，直到比较到最后一个元素为止

Step3：将步长除以步长系数作为新步长，重复步骤2直至步长小于1

Step4：对数据做冒泡排序

**代码展示：**

template <typename T>

void BubbleSort\_Comb(T begin, T end) //模仿sort()函数的输入参数

{

    const double RATE = 1.3; //定义步长系数

    int n = end - begin;     //获取数据长度, n为实际长度-1

    int increment = n + 1;   //计算步长

    while (increment > 1)

    {

        increment = (int)(increment / RATE); //更新步长

        bool sorted = true;                  //(sorted优化)

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

        {

            for (int j = i + increment; j <= n; j += increment)

            {

                if (\*(begin + j) < \*(begin + j - increment)) //间隔increment项作比较，并作冒泡排序

                {

                    exchange(begin + j, begin + j - increment);

                    sorted = false; //(sorted优化)

                }

            }

        }

        if (sorted) //(sorted优化)

            break;

    }

    bool sorted = false;                    //(sorted优化)

    for (int i = 0; i <= n && !sorted; i++) //从首数据开始遍历，直到倒数第二个数据为止，若有序则结束循环

    {

        for (int j = n, sorted = true; j > i; j--) //内循环控制排序的数据交换，若有序则结束循环

        {

            if (\*(begin + j) < \*(begin + j - 1))

            {

                exchange(begin + j, begin + j - 1);

                sorted = false; //(sorted优化)

            }

        }

    }

}

###### 算法分析与测试

算法分析

1. **空间复杂度分析**

由于三种插入排序算法都是在地址上进行数据操作，因此算法本身并没有产生额外的内存空间，因此，空间复杂度为。

1. **时间复杂度分析**
   1. 基础冒泡排序

由代码片段：

for (int i = 0; i <= n - 1 && !sorted; i++) //从首数据开始遍历，直到倒

数第二个数据为止，若有序则结束循环

{

    for (int j = 0, sorted = true; j <= n - i - 1; j++) {……}//内循环控制排序的数据交换，若有序则结束循环

}

可知，不论在什么情况下，基础冒泡排序的时间复杂度都是，但在加入sorted优化的判断下，基础冒泡排序在数据已经有序时能自动跳出循环，使得在处理顺序或基本有序的数据时能达到的时间复杂度。

* 1. 鸡尾酒排序

由代码片段：

while (left < right)

{

    bool sorted = true;                     //(sorted优化)

    for (int i = right - 1; i >= left; i--) {……}//从右向左做冒泡排序

    left++; //每从右向左排一次序，左侧有序数列+1，右移left

   for (int i = left + 1; i <= right; i++) {……}//从左向右做冒泡排序

    right--;    //每从左向右排一次序，右侧有序数列+1，左移right

    if (sorted) //(sorted优化)如果已经有序，则提前跳出循环

        break;

}

可知，鸡尾酒排序的最坏时间复杂度都是，但在加入sorted优化的判断下，鸡尾酒排序在数据已经有序时能自动跳出循环，使得在处理顺序或基本有序的数据时能达到的时间复杂度。

* 1. 梳排序

由代码片段：

while (increment > 1)

{

……

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

    {

     for (int j = i + increment; j <= n; j += increment) {……}

    }

  if (sorted) //(sorted优化)

        break;

}

bool sorted = false;                    //(sorted优化)

for (int i = 0; i <= n && !sorted; i++) //从首数据开始遍历，直到倒数第二个数据为止，若有序则结束循环

{

    for (int j = n, sorted = true; j > i; j--) {……} //内循环控制排序的数据交换，若有序则结束循环

}

可知，梳排序的平均案例时间复杂度为Ω(N2/2p)，其中p为增量次数，在本案例中为1.3。该算法的最坏情况复杂度为O(n2)，最佳情况复杂度为O(nlogn)。

1. **稳定性分析**
   1. 基础冒泡排序

使用冒泡排序的过程中，较小的数据总是不断的被移动到前面，在排序过程中数据前部会形成有序数列，按照基础冒泡排序的设计，前部的有序数组不会被再次排序，因此基础冒泡排序是稳定排序。

* 1. 鸡尾酒排序

鸡尾酒排序是在冒泡排序的基础上改变了遍历的方向，从基础冒泡排序的单向排序改变为正向反向轮流遍历，在数据的前后同时形成有序部分，且已经有序的部分不会再进行排序，因此鸡尾酒排序是稳定排序。

* 1. 梳排序

梳排序虽然也是对基础冒泡排序的优化，但是由于对数据进行了间隔分组排序，所以在排序的过程中，虽然各组内都是稳定的，但是对于整体的数据而言，梳排序并不稳定。

1. **适用类型分析**
   1. 基础冒泡排序

基础冒泡排序适用于小规模顺序数据。

* 1. 鸡尾酒排序

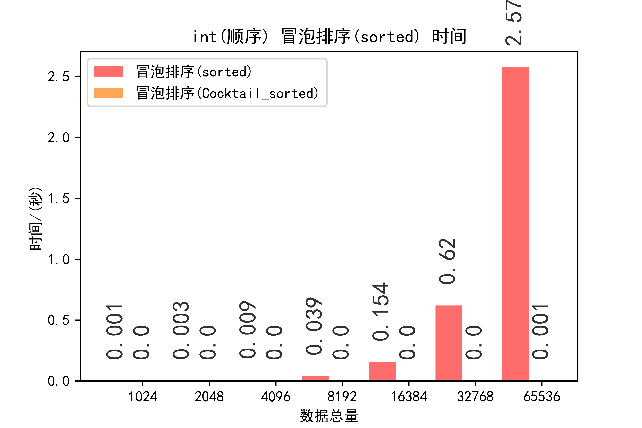
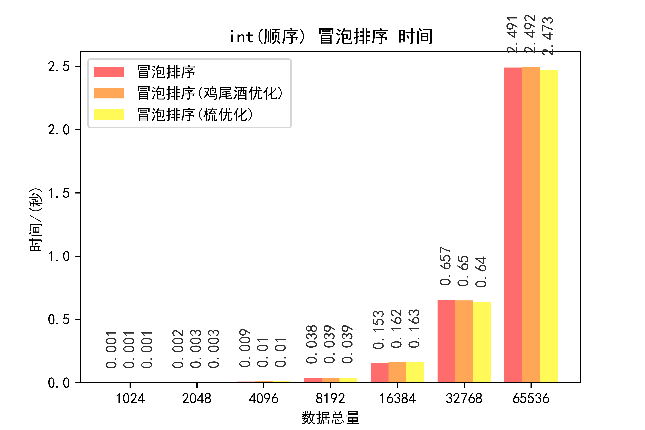
鸡尾酒排序在处理重复数据时将慢于基础冒泡排序，而在处理基本有序的数据时，鸡尾酒排序将远快于基础冒泡排序，达到线性时间。

* 1. 梳排序

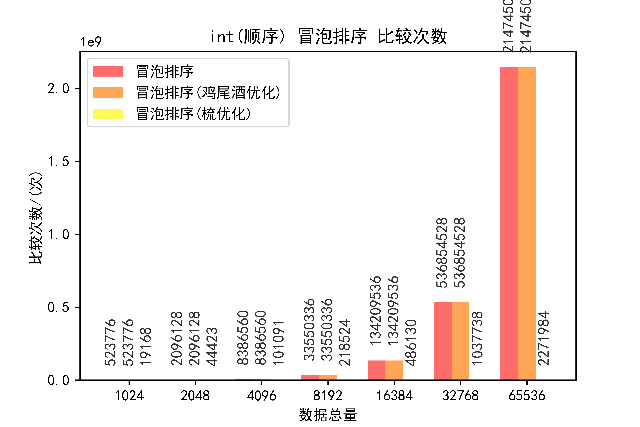
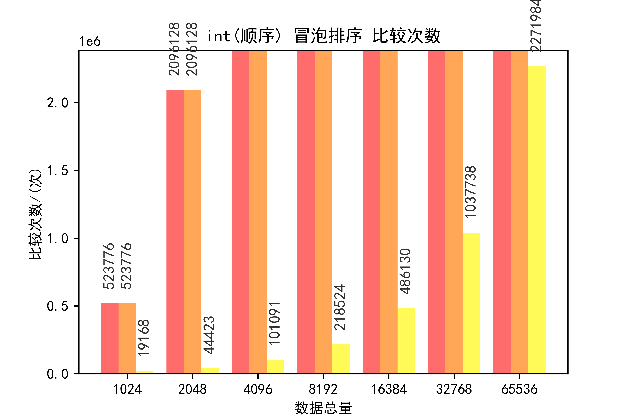
梳排序在处理大部分类型的数据时都将快于基础冒泡排序与鸡尾酒排序。

对不同数据构型排序时的表现

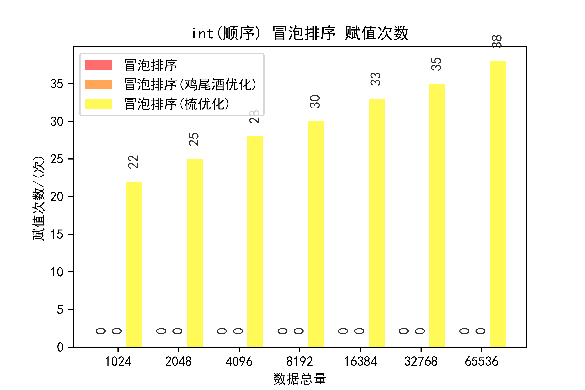
1. **对正序数据的测试**



**图2.2.1.1三种冒泡排序对顺序整型数据的排序时间 图2.2.3.2三种插入排序对顺序整型数据的赋值次数(sorted)**

****

**图2.2.1.3三种插入排序对顺序整型数据的比较次数 图2.2.1.4三种插入排序对顺序整型数据的比较次数(放大）**

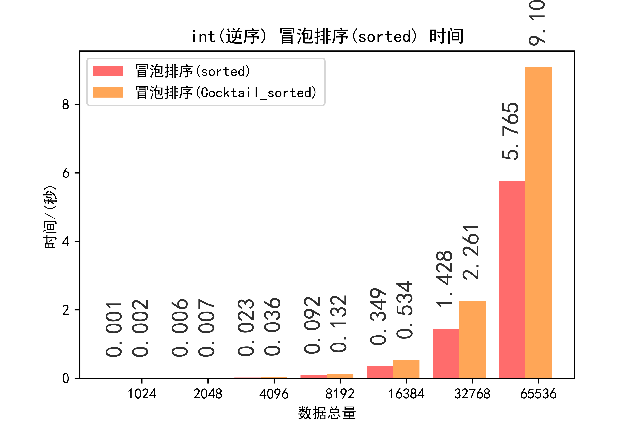
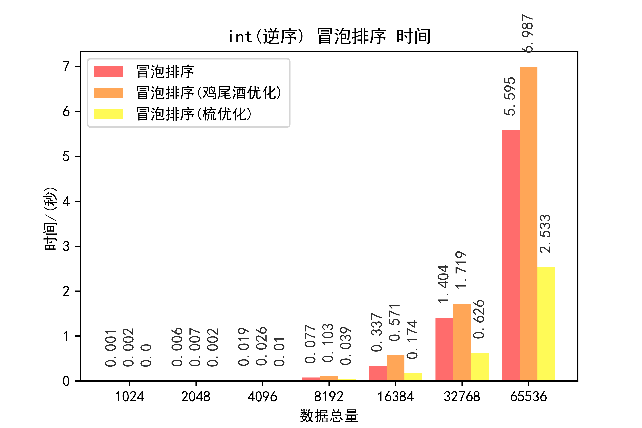
****

**图2.2.1.5三种插入排序对顺序整型数据的赋值次数**

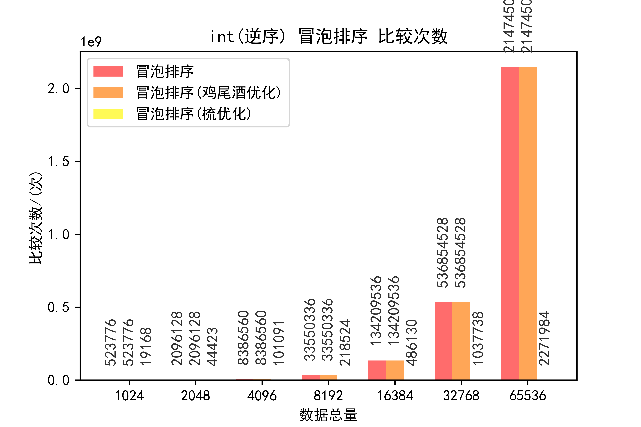
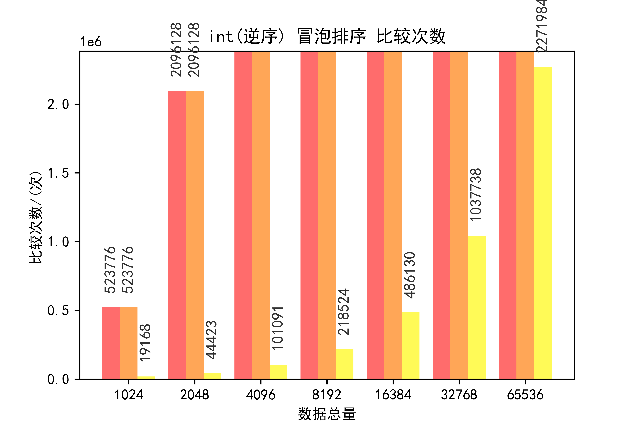
**分析：**

通过分析柱状图可以得知，在处理顺序数据时，基础冒泡排序与鸡尾酒排序的比较次数相同，都是对临时变量进行了赋值操作，而梳排序的由于将数据分组做冒泡排序，因此比较次数远小于前二者。在赋值次数方面，由于基础冒泡排序和鸡尾酒排序对于有序数据并不会进行赋值操作，所以二者都是零，而梳排序由于需要计算步长，因此有着少量的赋值操作。此外，通过加入sorted优化，即在排序时判断当前数据是否已有序，若有序则提前结束循环，可以极大程度提高对顺序数据的处理效率。因此可以得出结论：在处理顺序数据时，各个冒泡排序的变体在效率上并没有很大区别，加入sorted优化后，鸡尾酒排序明显快于其余两种排序方式。

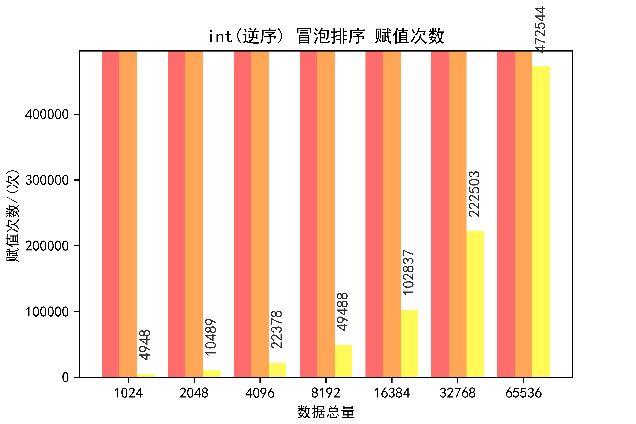
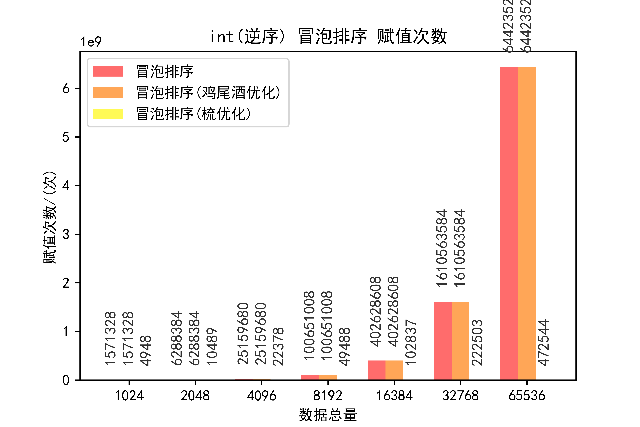
1. **对逆序数据的测试**

****

**图2.2.3.6三种冒泡排序对逆序整型数据的排序时间 图2.2.3.7三种冒泡排序对逆序整型数据的排序时间(sorted）**

****

**图2.2.3.8三种冒泡排序对逆序整型数据的比较次数 图2.2.3.9三种冒泡排序对逆序整型数据的比较次数(放大)**

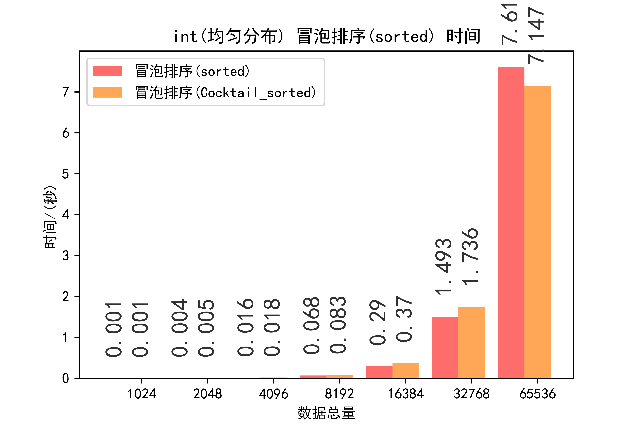
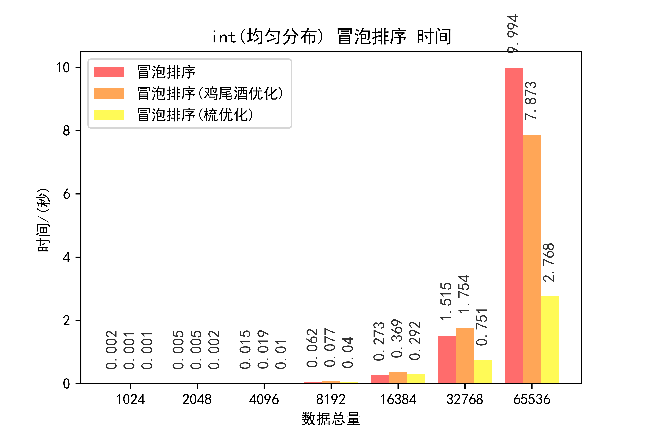
****

**图2.2.3.9三种冒泡排序对逆序整型数据的赋值次数 图2.2.3.9三种冒泡排序对逆序整型数据的赋值次数(放大)**

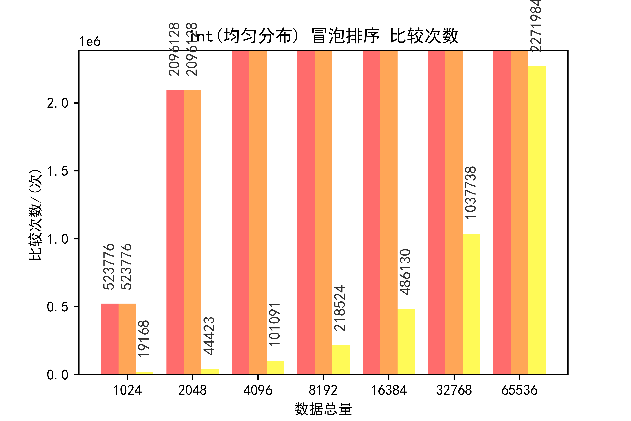
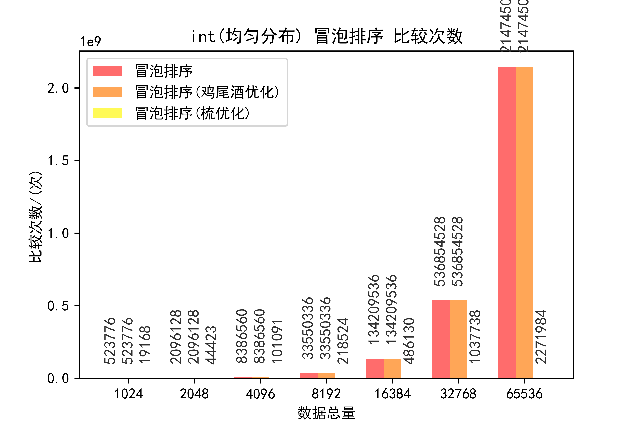
**分析：**

通过分析柱状图可以得知，在处理逆序数据时，基础冒泡排序与鸡尾酒排序的比较次数与赋值次数相同，而梳排序在这里两方面远少于前二者，因此在时间方面有着较大的提升，大约为前二者的三分之一至二分之一。由于鸡尾酒排序最适合的是顺序数据，因此在处理逆序数据时，由于内部变量更多的判断次数，导致其效率反而低于基础冒泡排序。在加入了sorted优化后，性能反而得到了负优化，这是因为在处理逆序数据的时候，基础冒泡排序与鸡尾酒排序只有在最后才将数据处理为有序，不存在提前跳出，反而由于更多的判断降低了效率。综合看来，在处理逆序数据时，算法优先级为：梳排序>基础冒泡排序>鸡尾酒排序。

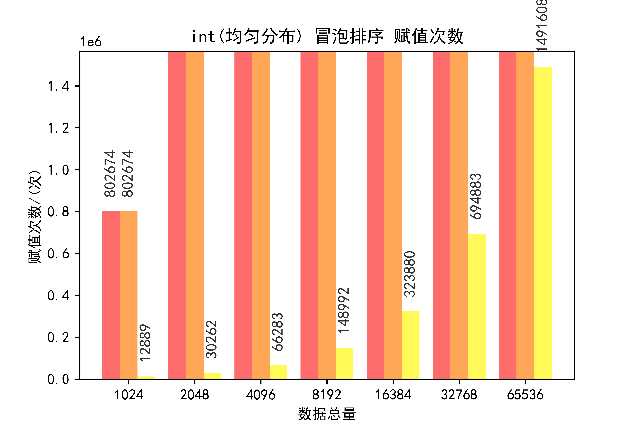
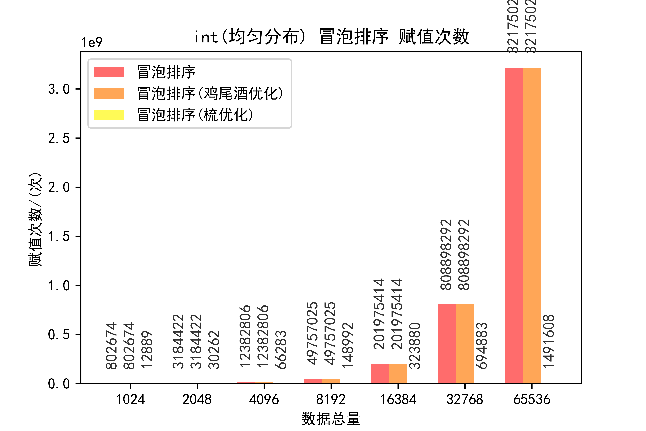
1. **对均匀分布数据的测试**



**图2.2.3.10三种冒泡排序对均匀分布整型数据的排序时间 图2.2.3.11三种冒泡排序对均匀分布整型数据的排序时间(sorted)**

****

**图2.2.3.12三种冒泡排序对均匀分布整型数据的比较次数 图2.2.3.13三种冒泡排序对均匀分布整型数据的比较次数(放大)**

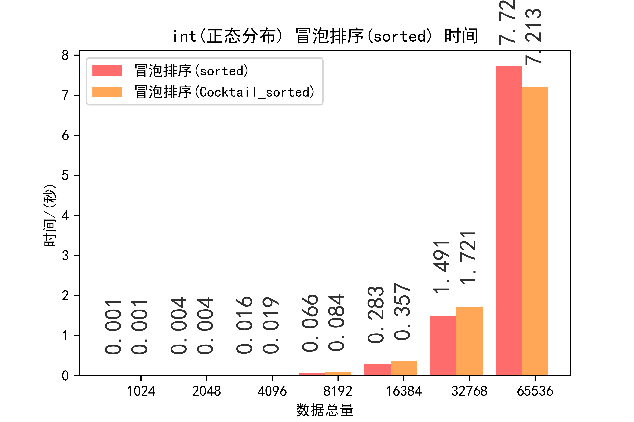
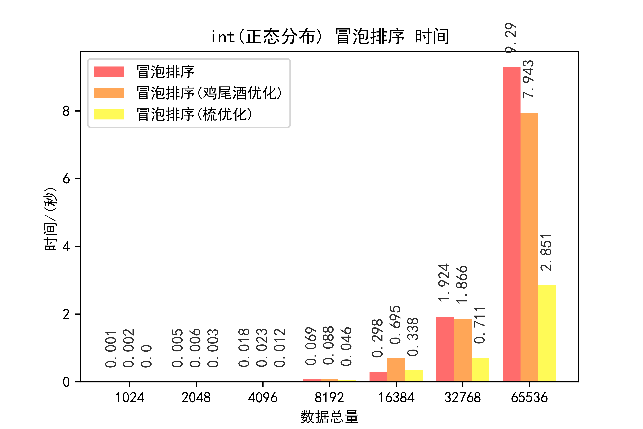
****

**图2.2.3.14三种冒泡排序对均匀分布整型数据的赋值次数 图2.2.3.15三种冒泡排序对均匀分布整型数据的赋值次数(放大)**

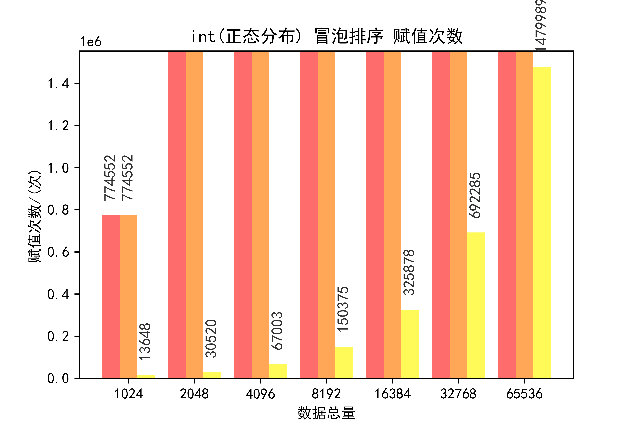
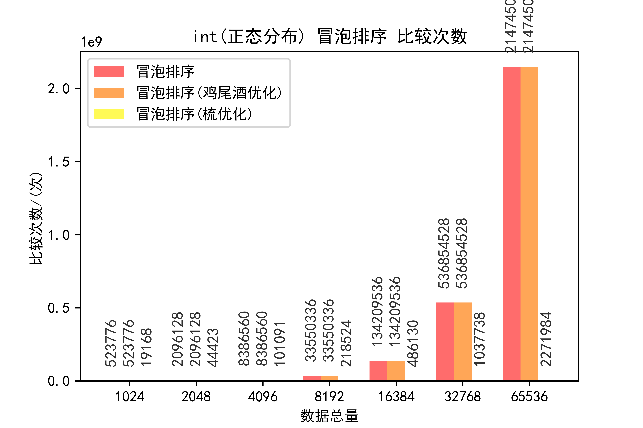
**分析：**

通过分析柱状图可以得知，在处理逆序数据时，基础冒泡排序与鸡尾酒排序的比较次数与赋值次数相同，而梳排序在这里两方面远少于前二者，因此在时间方面有着较大的提升，大约为前二者的三分之一至二分之一。由于均匀分布数据有序性高于逆序数据而低于顺序数据，因此鸡尾酒排序在处理均匀分布数据时表现优于基础冒泡排序。在加入了sorted优化后，基础冒泡排序性能有着显著的提升，这是因为在处理均匀分布数据的时候，基础冒泡排序将较快的把数据处理成有序，因此能够提前跳出循环；对于鸡尾酒排序也有一定的优化。综合看来，在处理均匀分布的数据时，算法优先程度：梳排序>鸡尾酒排序 >基础冒泡排序。

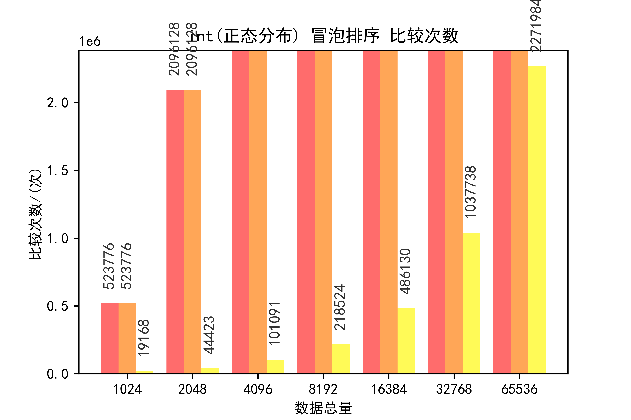
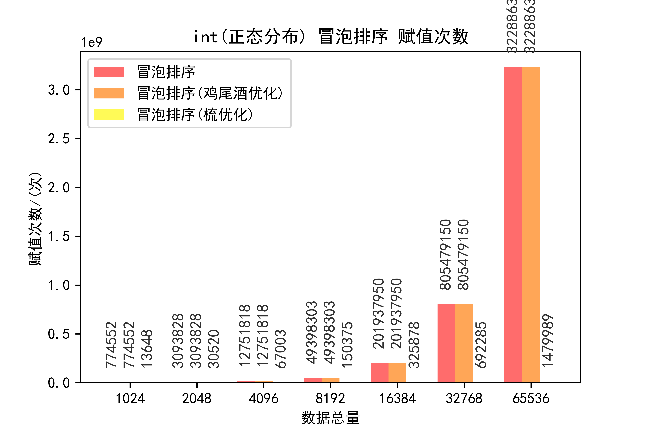
1. **对正态分布数据的测试**



**图2.2.3.16三种冒泡排序对正态分布整型数据的排序时间 图2.2.3.17三种冒泡排序对正态分布整型数据的排序时间(sorted)**

****

**图2.2.3.18三种冒泡排序对正态分布整型数据的比较次数 图2.2.3.19三种冒泡排序对正态分布整型数据的排序时间(放大)**

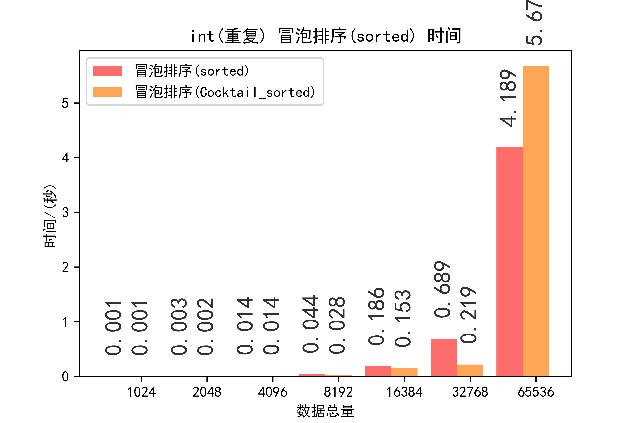
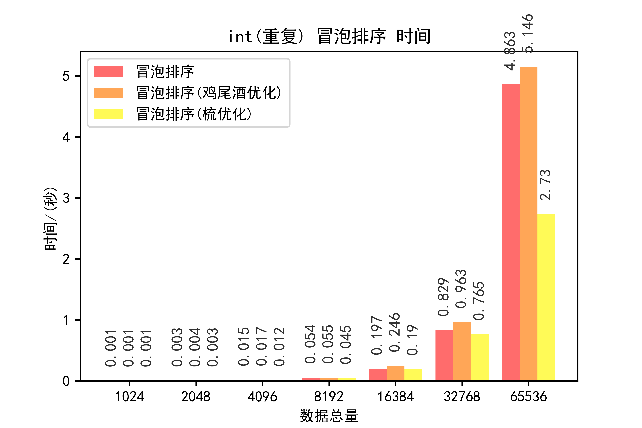
****

**图2.2.3.20三种冒泡排序对顺序整型数据的排序时间 图2.2.3.21三种冒泡排序对顺序整型数据的排序时间(放大)**

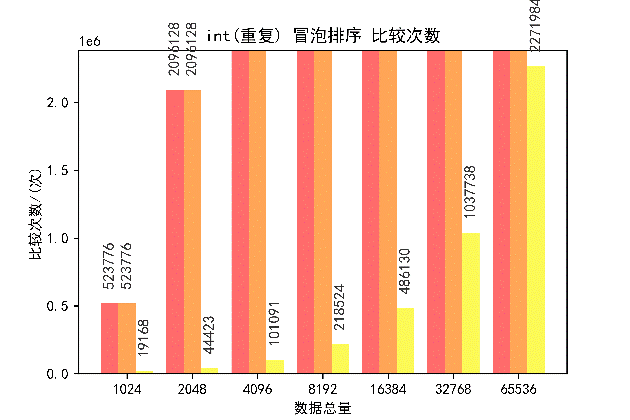
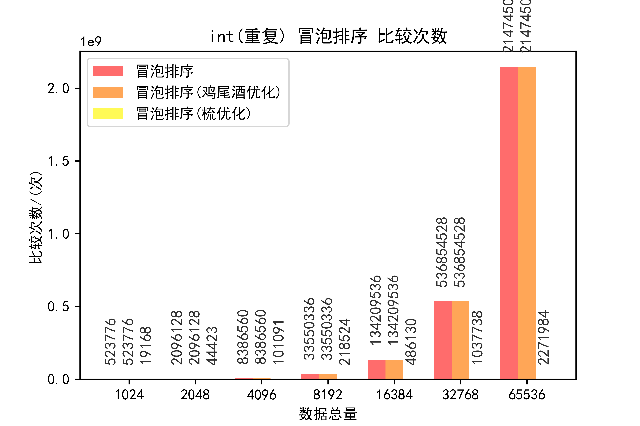
**分析：**

通过分析柱状图可以得知，在处理正态分布数据时，在赋值次数方面与比较次数方面三种排序方法几乎与处理均匀分布的数据一致。综合看来，在处理正态分布数据时，算法优先程度：梳排序>鸡尾酒排序>基础冒泡排序。

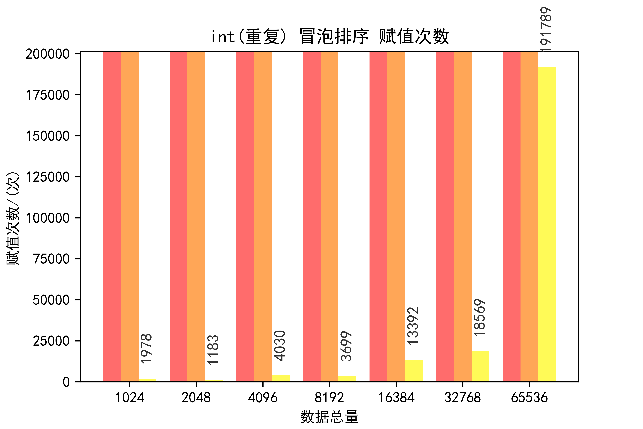
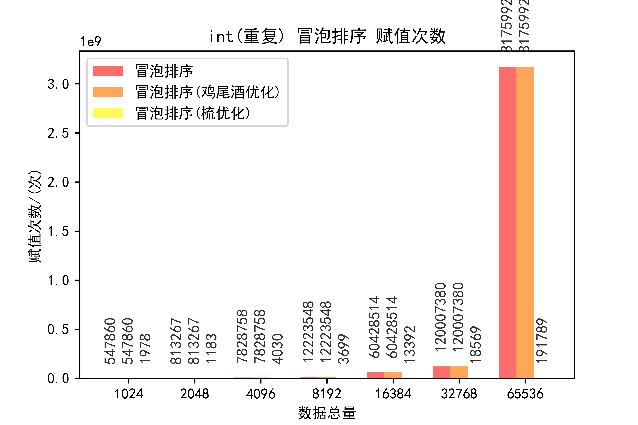
1. **对含有大量相同数字数据的测试**



**图2.2.3.22三种冒泡排序对重复整型数据的排序时间 图2.2.3.23三种冒泡排序对顺重复整型数据的排序时间(sorted)**

****

**图2.2.3.24三种冒泡排序对重复整型数据的比较次数 图2.2.3.25三种冒泡排序对重复整型数据的比较次数(放大)**

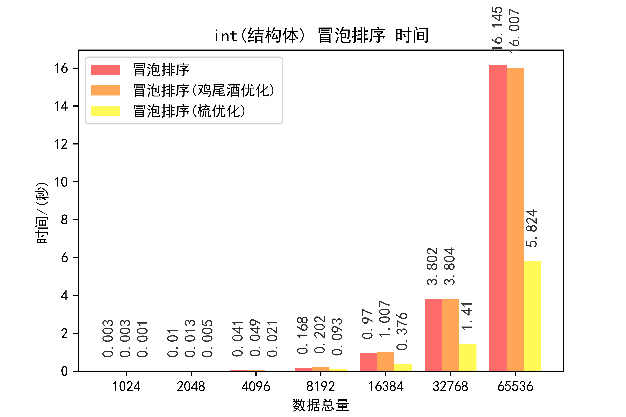
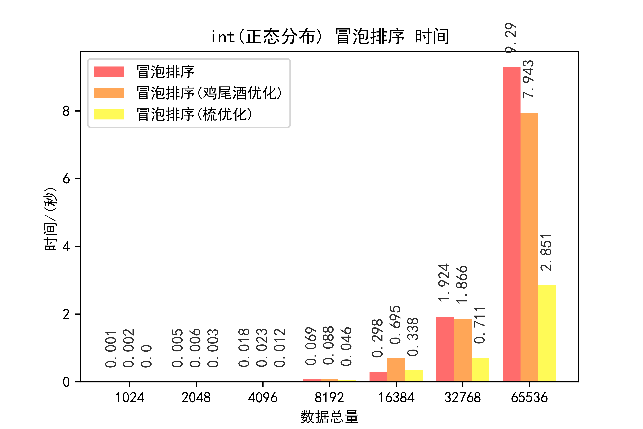
****

**图2.2.3.26三种冒泡排序对重复整型数据的赋值次数 图2.2.3.27三种冒泡排序对重复整型数据的赋值次数(放大)**

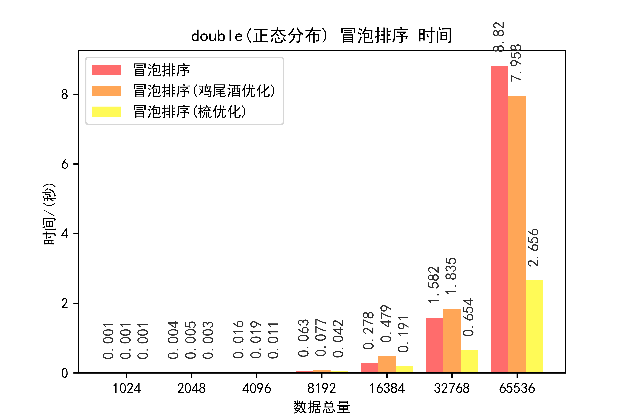
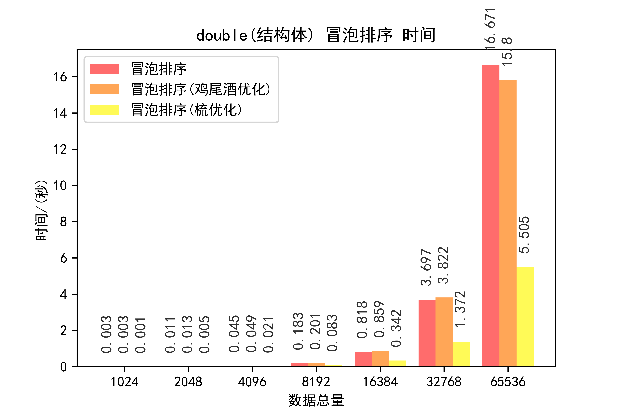
**分析：**

通过分析柱状图可以得知，在处理正态分布数据时，在赋值次数方面与比较次数方面三种排序方法几乎与处理均匀分布的数据一致。综合看来，在处理大量重复数据时，算法优先程度：梳排序>鸡尾酒排序>基础冒泡排序。

**不同数据类型对算法的影响**



**图2.2.3.28三种冒泡排序对正态分布整型数据排序时间 图2.2.3.29三种冒泡排序对正态分布整型结构体数据排序时间**

****

**图2.2.3.30三种冒泡排序对正态分布双精度数据排序时间 图2.2.3.31三种冒泡排序对正态分布双精度结构体数据排序时间**

**分析：**

用三种冒泡排序处理相同数据构型（均匀分布型）不同数据类型的数据时，纵向对比，普通整型与双精度浮点型，结构体类型整型与双精度浮点型消耗的时间基本一致；横向对比，结构体类型的数据消耗的时间基本在普通类型数据的一倍以上，其主要原因是访问结构体数据所花费时间大于直接访问普通数据。由此可以得出结论：对于任何类型的数据，在冒泡排序的变体之中，梳排序综合表现最佳，且处理不同数据类型的数据效率除了受到数据规模的影响外，还会受到访问数据的速度影响。

**小结**

在基于冒泡排序的三种优化之中，梳排序整体表现最佳，且由于基础冒泡排序的特性，我们特别测试了加入sorted优化来判断待排序数据是否已经完成排序，在大多数情况下，加入sorted优化的基础冒泡排序与鸡尾酒排序都相对表现更佳（除顺序外），因此为了提高排序效率与对不同数据处理所花费时间的稳定性，使用冒泡排序与鸡尾酒排序时建议加入sorted优化。若对算法稳定性无要求，则在这三者之中推荐使用梳排序。

#### 选择排序

-周

#### 插入排序

###### 直接插入排序

直接插入排序是一种最简单的排序方法，它的基本思想是将一个记录插入到已经排好序的有序表中，从而得到一个新的、记录数增1的有序表。在其实现过程使用双层循环，外层循环对除了第一个元素之外的所有元素，内层循环对当前元素前面有序表进行待插入位置查找，并进行移动。

**实现步骤：**

Step1：从第一个元素开始，记该元素已被排序。

Step2：取出下一个元素，在已排序的元素序列中从后向前扫描

Step3：如果该元素(已排序)大于新元素，则向前移动元素。

Step4：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并将元素插入该位置后。

Step5：重复步骤2-5

**代码展示：**

template <typename T>

void InsertionSort\_Origin(T begin, T end)

{

    int len = end - begin + 1; //通过首尾地址计算出数据长度

    for (int i = 1; i < len; i++) //外循环表里需要插入的数字

    {

        int key = \*(begin + i);

        int j = i - 1;

        while ((j >= 0) && (key < \*(begin + j))) //内循环控制查找数字的位置与交换元素位置

        {

            \*(begin + j + 1) = \*(begin + j);

            j--;

        }

        \*(begin + j + 1) = key;

    }

}

###### 优化1：折半插入排序

折半插入排序（Binary Insertion Sort）是在基础的插入排序上对查找算法进行了改进，通过二分法查找取代遍历查找，所以又被称为二分插入排序。

**实现步骤：**

Step1：从第一个元素开始，记该元素已被排序。

Step2：取出下一个元素，在已排序的元素序列中查找中间位置。

Step3：如果中位元素(已排序)大于新元素，则在前一半中寻找中间位置，反之亦然。

Step4：重复步骤3，直到找到唯一已排序元素并将元素插入该位置后。

SteP5：重复步骤2-5

**代码展示：**

template <class T>

void InsertionSort\_Binary(T begin, T end)

{

    int len = end - begin + 1; //通过首尾地址计算出数据长度

    int i, j, temp, m, low, high;

    for (i = 1; i < len; i++) //外循环表里需要插入的数字

    {

        temp = \*(begin + i);

        low = 0;

        high = i - 1;

        while (low <= high) //内循环1：二分法查找元素应插入的位置

        {

            m = (low + high) / 2;

            if (\*(begin + m) > temp)

                high = m - 1;

            else

                low = m + 1;

        }

        for (j = i - 1; j >= high + 1; j--) //内循环2：交换元素位置

            \*(begin + j + 1) = \*(begin + j);

        \*(begin + j + 1) = temp;

    }

}

###### 优化2：希尔排序

希尔排序又称为缩小增量排序，本质上是利用了直接插入排序对于小规模数据和顺序数据性能较为优异的特点而改良的一种算法。算法先将要排序的一组数按某个增量d（n／3 + 1，n为要排序数的个数）分成若干组，每组中记录的下标相差d.对每组中全部元素进行直接插入排序，然后再用一个较小的增量（d／3 + 1）对它进行分组，在每组中再进行直接插入排序。当增量减到1时，进行直接插入排序后，排序完成。

**实现思路：**

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

STEP1：选择一个增量序列 ，，……，，其中＞ ， ＝ 1；

STEP2：按增量序列个数k，对序列进行k趟排序；

STEP3：每趟排序，根据对应的增量 ，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。

**代码展示：**

template <typename T>

template <typename T>

void InsertionSort\_Shell(T begin, T end)

{

    int h = 1, len = end - begin + 1;

    while (h < len / 3) //选取增量(此处为非整除增量)

        h = 3 \* h + 1;

    while (h >= 1) //持续循环直到增量为1，即退化为直接插入排序后结束

    {

        for (int i = h; i < len; i++) //选取间隔h的数据进行排序

        {

            for (int j = i; j >= h && \*(begin + j) < \*(begin + j - h); j -= h) //判断元素位置并交换元素(直接插入排序)

                exchange(\*(begin + j), \*(begin + j - h));

        }

        h = h / 3;

    }

}

###### 算法分析与测试

算法分析

1. **空间复杂度分析**

由于三种插入排序算法都是在地址上进行数据操作，因此算法本身并没有产生额外的内存空间，因此，空间复杂度为。

1. **时间复杂度分析**
   1. 直接插入排序

由代码片段：

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

......

    while ((j >= 0) && memcmp(key, begin + j \* NUM, NUM \* sizeof(char)) < 0){......}

......

}

可知，直接插入排序使用了双重循环，因此，在最坏情况下，其时间复杂度为，平均情况下，时间复杂度同样为，只有当处理顺序数据时，内循环判断代码：

memcmp(key, begin + j \* NUM, NUM \* sizeof(char)) < 0

恒为FALSE，因此，双循环将退化为单循环，时间复杂度将退化为。

* 1. 折半插入排序

由代码片段：

for (i = 1; i < len; i++){

......

    while (low <= high){......}

    for (j = i - 1; j >= high + 1; j--) ......

    ......

}

可知，折半插入排序使用了双重循环，因此，在最坏情况下，其时间复杂度为，平均情况下，时间复杂度同样为， 只有当处理顺序数据时，第二个内循环退化，整体内循环为二分法时间复杂度，即，与外循环相结合，总时间复杂度为。

* 1. 希尔排序

由代码片段：

while (h < len / 3) ......

while (h >= 1){

    for (int i = h; i < len; i++){

    for (int j = i; j >= h && \*(begin + j) < \*(begin + j - h); j -= h) ......

    }

......

}

可知，希尔排序使用了三重循环，但由于其内外层循环相互影响，且时间复杂度受到增量序列选取的影响，其时间复杂度并不固定，对于处理不同数据构型时的时间复杂度在之间不等。本小组通过查找文献并比对的方式，选取d = n / 3 + 1为增量序列而非最常用的Hibbard序列，进一步优化时间复杂度，使其稳定在。[[1]](#endnote-1)[1]

1. **稳定性分析**
2. 直接插入排序

使用直接插入排序过程中，大小相同的元素，偶尔会由于交换而相离，但总体上会相互靠近，直到它们相互毗邻后，插入排序便不再改变它们的相互位置，所以插入排序是稳定的。

1. 折半插入排序

折半插入排序只在查询数据时对直接插入排序进行了改进，在交换的思路上并没有区别，因此，折半插入排序与直接插入排序一样，属于稳定的排序。

1. 希尔排序

希尔排序虽然也是对直接插入排序的优化，但是由于对数据进行了间隔分组，所以在排序的过程中，虽然各组内都是稳定的，但是对于整体的数据而言，希尔排序并不稳定。

1. **适用类型分析**
   1. 直接插入排序

直接插入排序适用于小规模顺序数据。

* 1. 折半插入排序

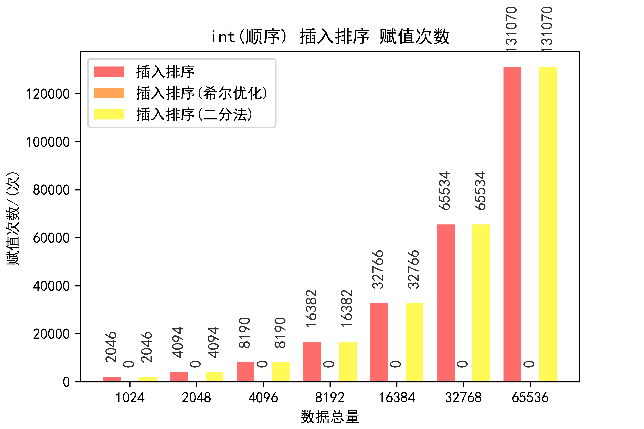
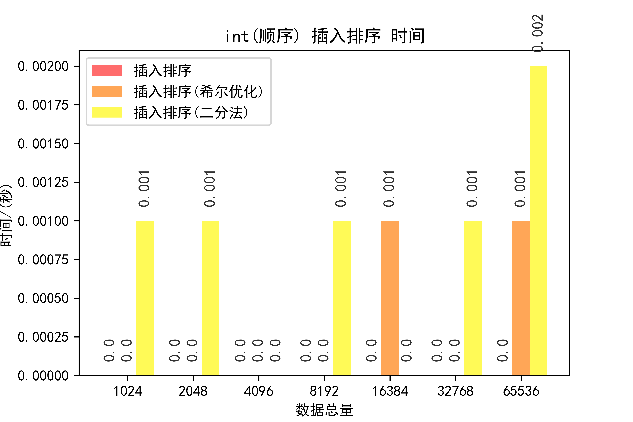
折半插入排序在处理极少量数据与顺序数据时慢于直接插入排序，而在处理大量数据时由于查询算法的改进，将快于直接插入排序。

* 1. 希尔排序

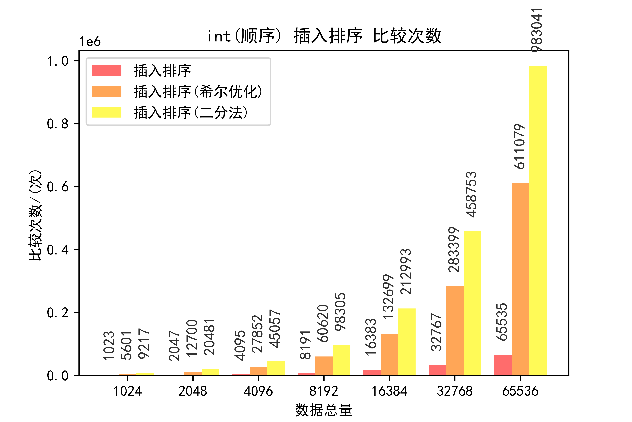
希尔排序在处理顺序数据时慢于直接插入排序，而在处理大量数据时，将远快于直接插入排序与折半插入排序。

对不同数据构型排序时的表现

1. **对正序数据的测试**



**图2.2.3.1三种插入排序对顺序整型数据的排序时间 图2.2.3.2三种插入排序对顺序整型数据的赋值次数**

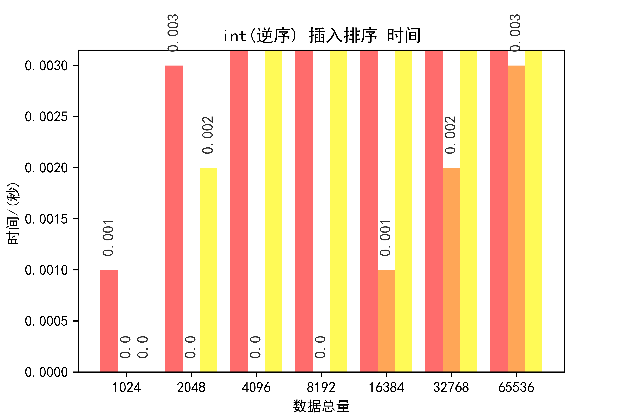
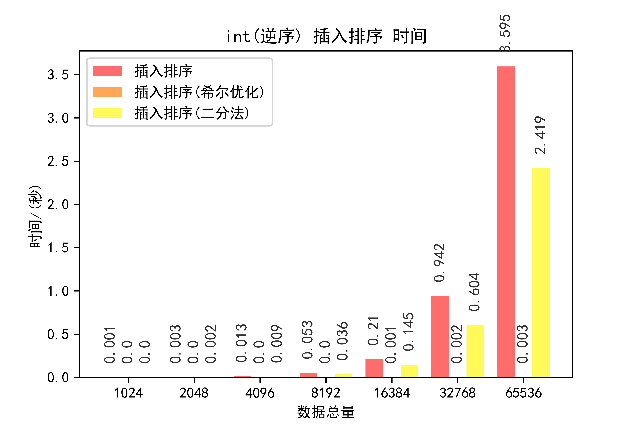


**图2.2.3.3三种插入排序对顺序整型数据的比较次数**

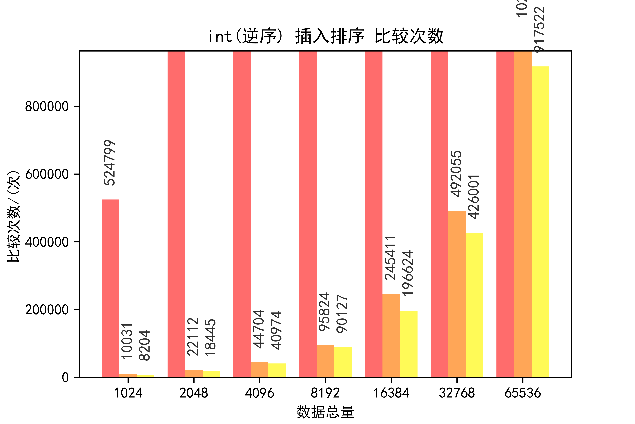
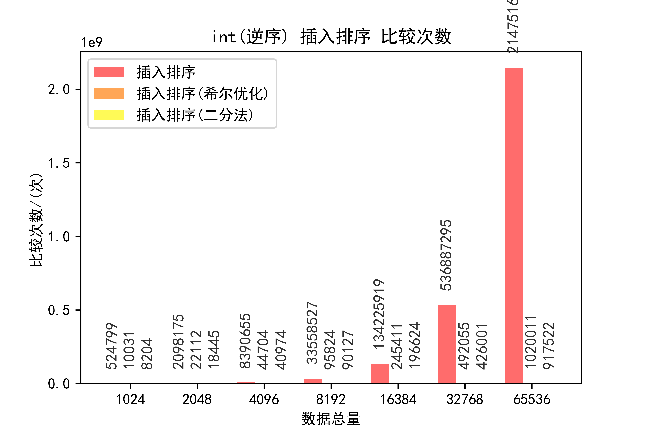
**分析：**

通过分析柱状图可以得知，在处理顺序数据时，直接插入排序与折半插入排序的赋值次数相同，都是对临时变量进行了赋值操作，而希尔排序不存在此操作。在比较次数方面，由于直接插入排序内循环只需要遍历第一个数据就能跳出循环，因此比折半插入排序的二分查找次数更少；希尔排序由于运用了三重循环嵌套，所以比较次数远大于直接插入排序，所以可以得出结论：在处理顺序数据时，各个插入排序的变体优先级为，直接插入排序>希尔排序>折半插入排序。

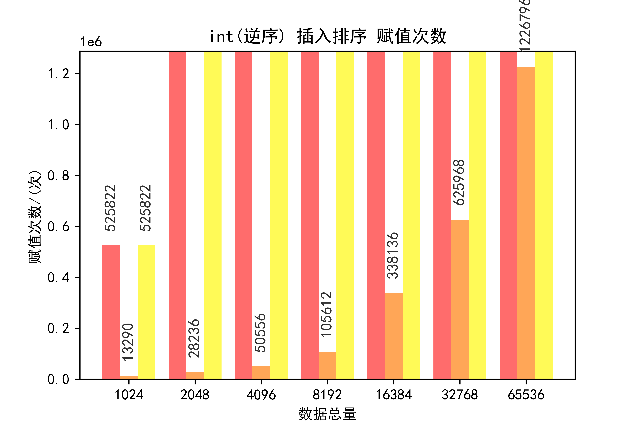
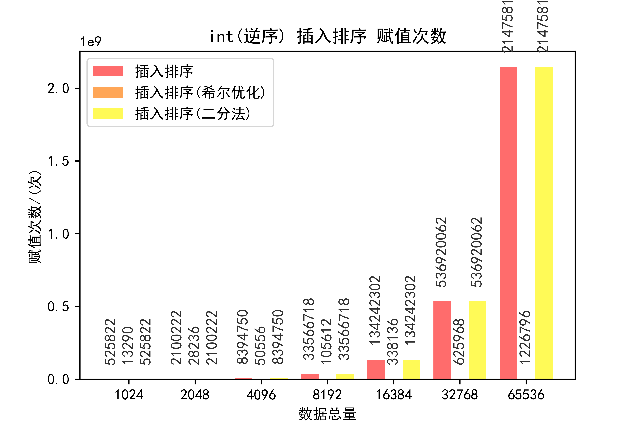
1. **对逆序数据的测试**

****

**图2.2.3.4三种插入排序对逆序整型数据的排序时间 图2.2.3.5三种插入排序对逆序整型数据的排序时间(放大）**



**图2.2.3.6三种插入排序对逆序整型数据的比较次数 图2.2.3.7三种插入排序对逆序整型数据的比较次数(放大）**

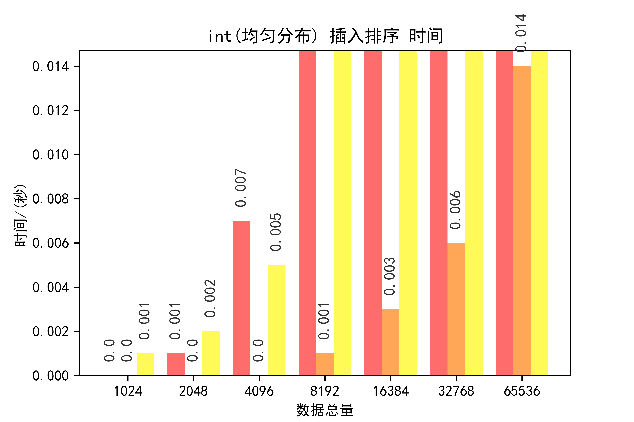
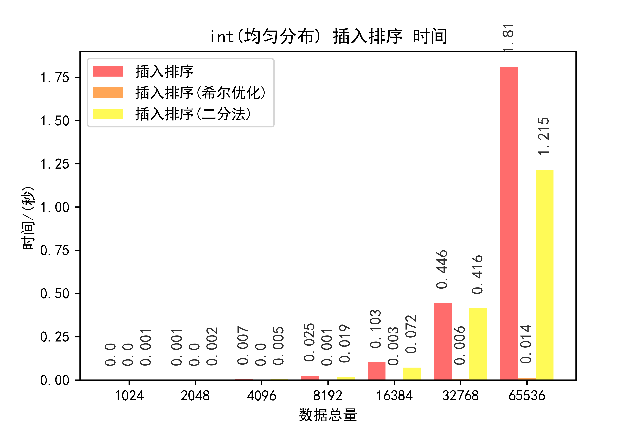


**图2.2.3.8三种插入排序对逆序整型数据的赋值次数 图2.2.3.9三种插入排序对逆序整型数据的比较次数(放大）**

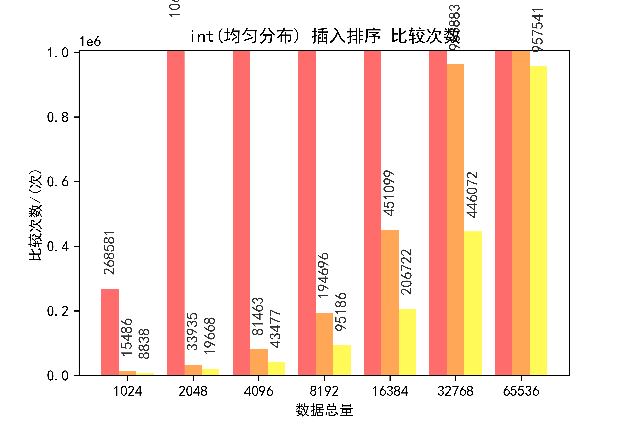
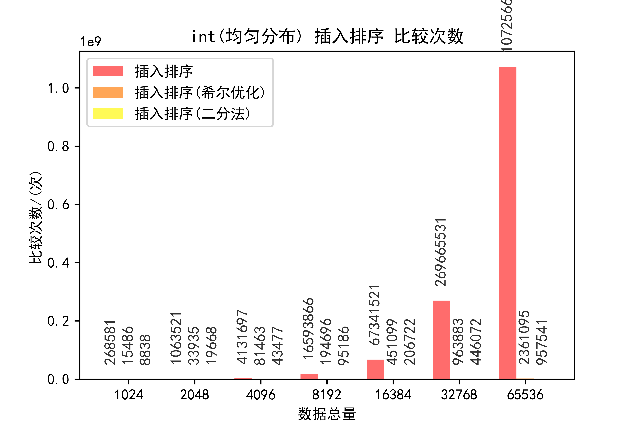
**分析：**

通过分析柱状图可以得知，在处理逆序数据时，直接插入排序与折半插入排序的赋值次数相同，都是对临时变量进行了赋值操作，而希尔排序通过将整体拆分成多个规模更小的数组进行插入排序，极大程度的减少了赋值次数。在比较次数方面，由于直接插入排序在处理逆序数据时，内循环需要遍历所有已排序数据才能跳出循环，因此直接插入排序的表现十分糟糕；折半插入排序由于使用了二分法，在比较次数上有着显著的提升；而希尔排序由于分组处理的思想，在比较次数上仅略逊于折半插入排序。综合看来，在处理逆序数据时，希尔排序>折半插入排序>直接插入排序。

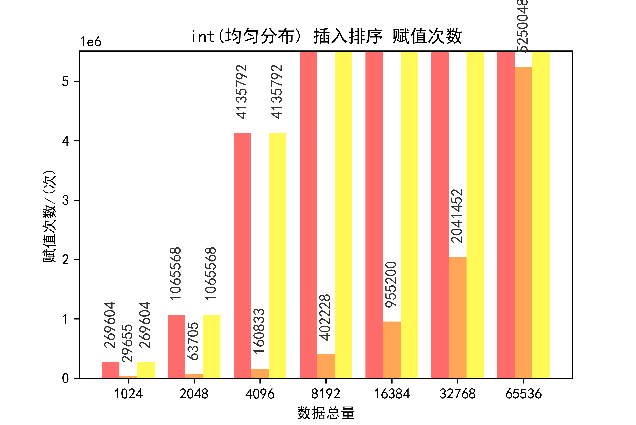
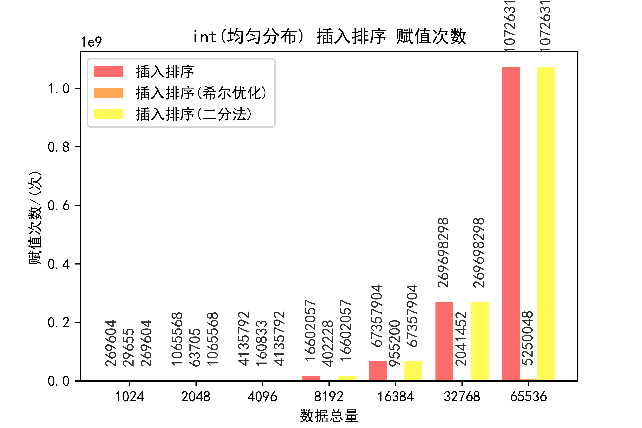
1. **对均匀分布数据的测试**



**图2.2.3.10三种插入排序对均匀分布整型数据的排序时间 图2.2.3.11三种插入排序对均匀分布整型数据的排序时间(放大)**

****

**图2.2.3.12三种插入排序对顺序整型数据的比较次数(次) 图2.2.3.13三种插入排序对顺序整型数据的比较次数(次）(放大)**

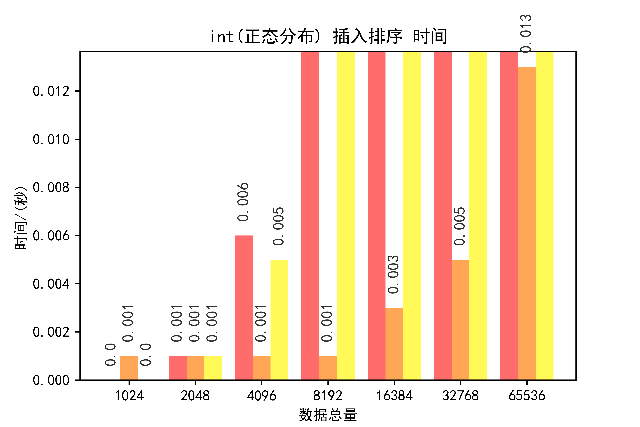
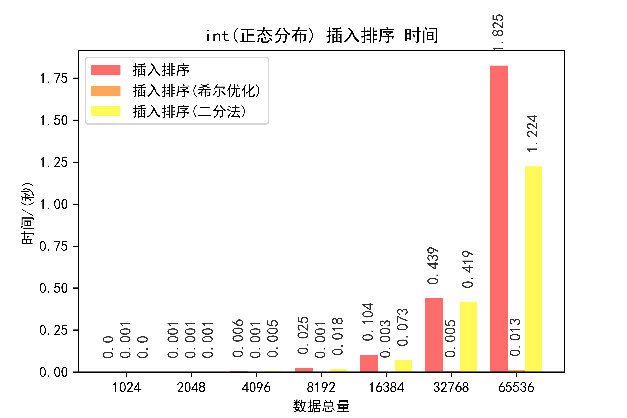
****

**图2.2.3.14三种插入排序对顺序整型数据的赋值次数(次) 图2.2.3.15三种插入排序对顺序整型数据的赋值次数(次）(放大)**

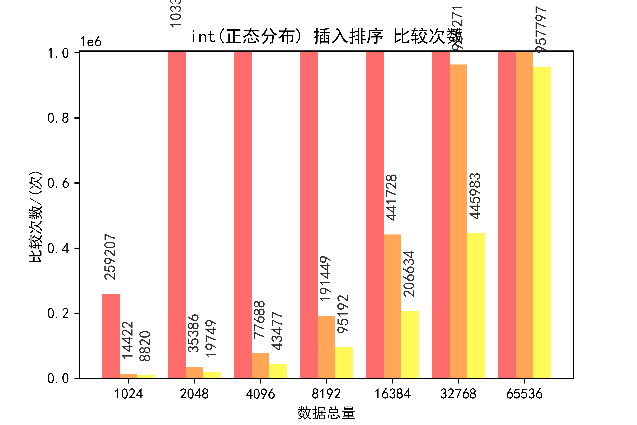
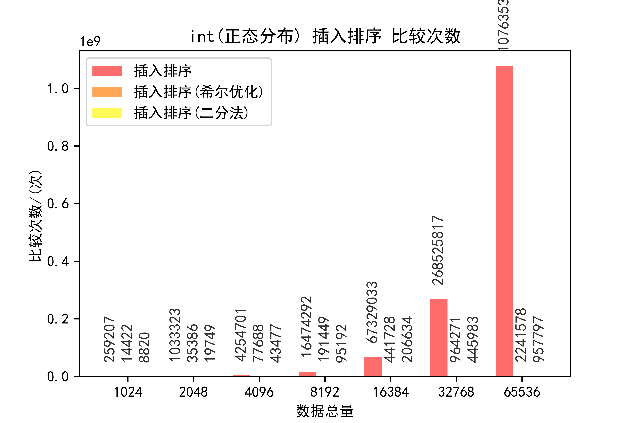
**分析：**

通过分析柱状图可以得知，在处理均匀分布数据时，直接插入排序与折半插入排序的赋值次数相同，都是对临时变量进行了赋值操作，而希尔排序通过将整体拆分成多个规模更小的数组进行插入排序，极大程度的减少了赋值次数。在比较次数方面，直接插入排序由于使用遍历的方式查找，与其他两组相比表现较糟糕；折半插入排序由于使用了二分法，在比较次数上优于希尔排序，为三者中最佳；而希尔排序通过分组处理，比较次数的表现也较为良好。综合看来，在处理均匀分布数据时，希尔排序>折半插入排序>直接插入排序。

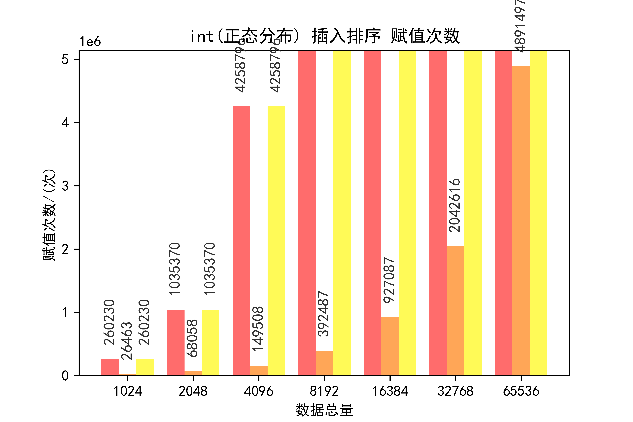
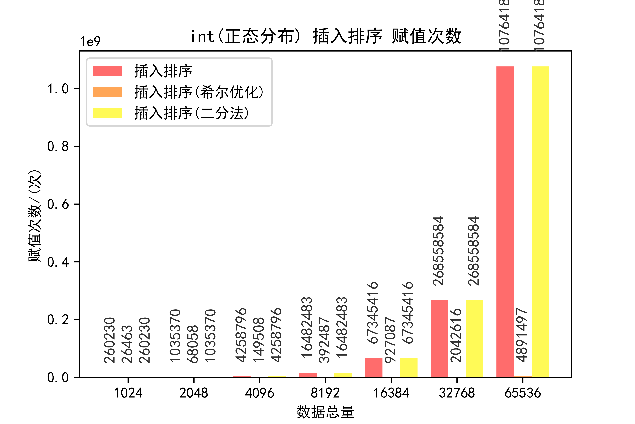
1. **对正态分布数据的测试**



**图2.2.3.16三种插入排序对正态分布整型数据的排序时间 图2.2.3.17三种插入排序对正态分布整型数据的排序时间(放大)**

****

**图2.2.3.18三种插入排序对正态分布整型数据的比较次数 图2.2.3.19三种插入排序对正态分布整型数据的排序时间(放大)**

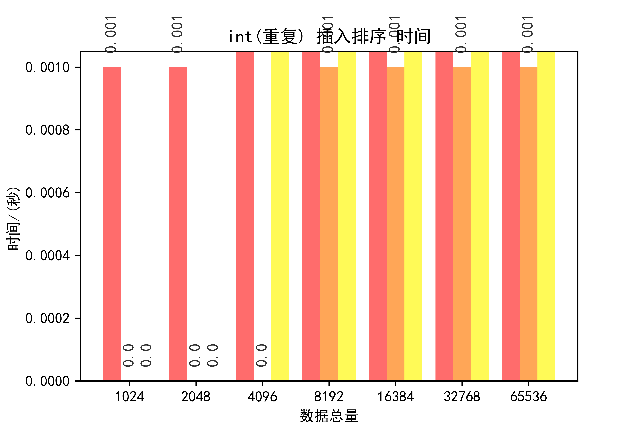
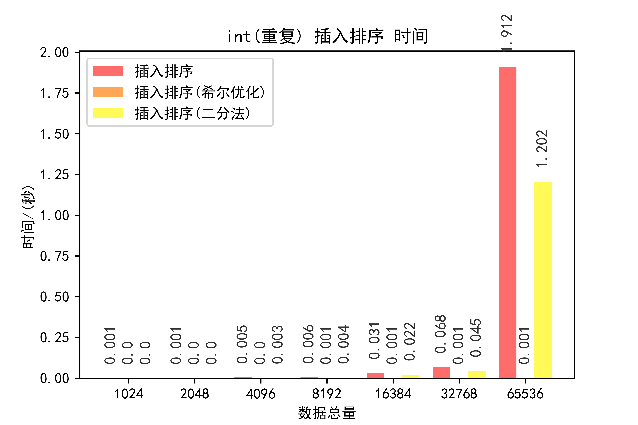
****

**图2.2.3.20三种插入排序对顺序整型数据的排序时间 图2.2.3.21三种插入排序对顺序整型数据的排序时间(放大)**

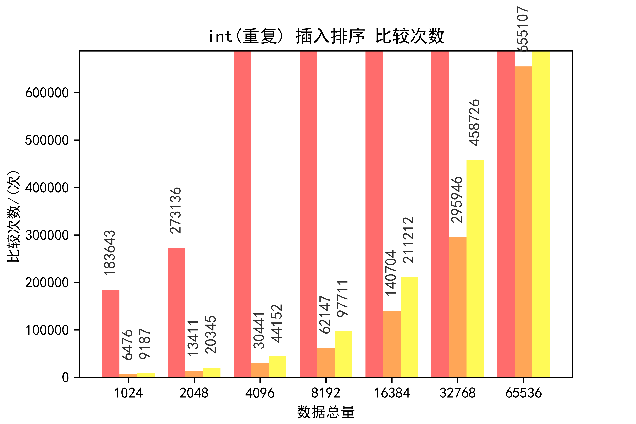
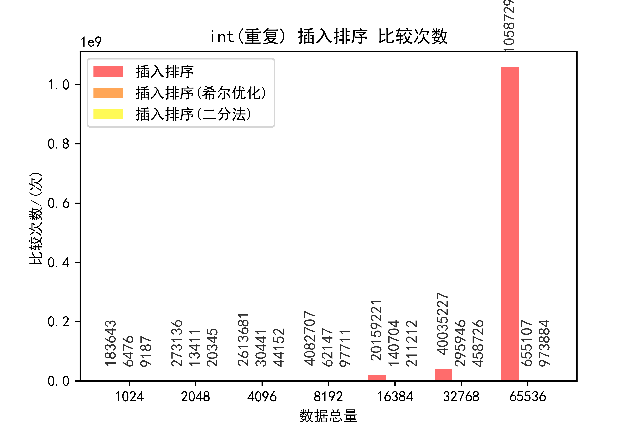
**分析：**

通过分析柱状图可以得知，在处理正态分布数据时，在赋值次数方面与比较次数方面三种排序方法几乎与处理均匀分布的数据一致。综合看来，在处理均匀分布数据时，希尔排序>折半插入排序>直接插入排序

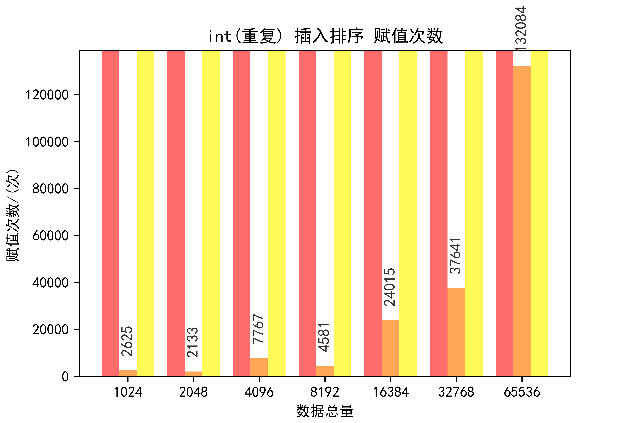
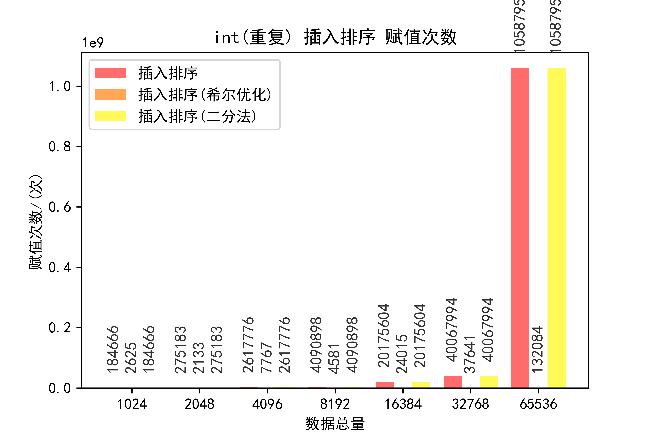
1. **对含有大量相同数字数据的测试**



**图2.2.3.22三种插入排序对重复整型数据的排序时间 图2.2.3.23三种插入排序对顺重复整型数据的排序时间(放大)**

****

**图2.2.3.24三种插入排序对重复整型数据的比较次数 图2.2.3.25三种插入排序对重复整型数据的比较次数(放大)**

****

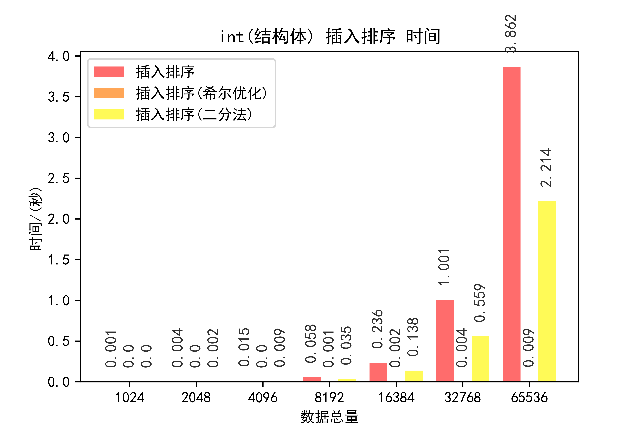
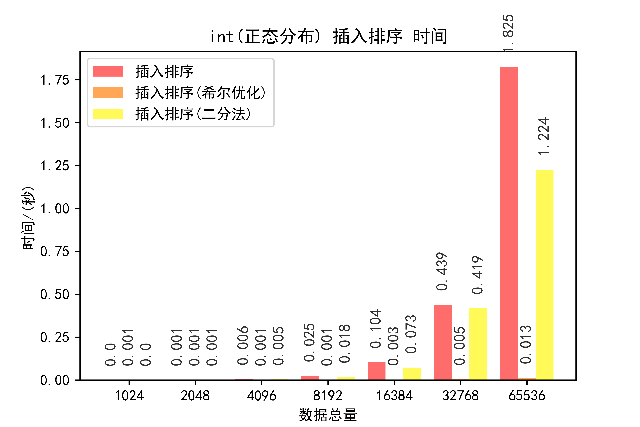
**图2.2.3.26三种插入排序对重复整型数据的赋值次数 图2.2.3.27三种插入排序对重复整型数据的赋值次数(放大)**

**分析：**

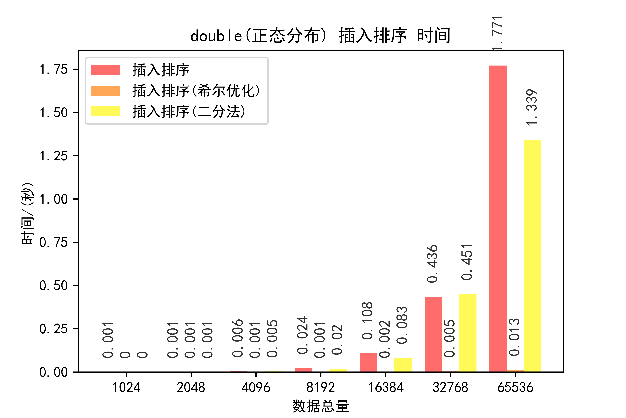
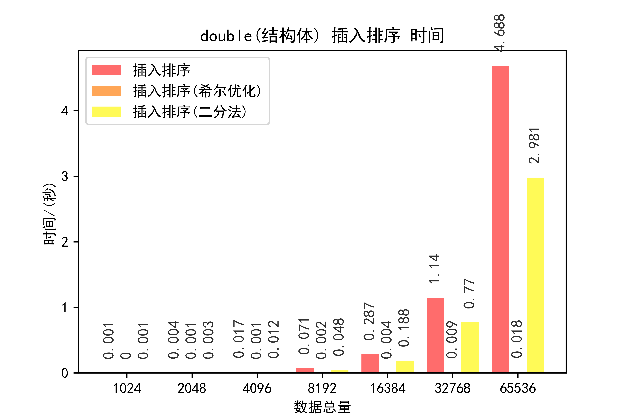
通过分析柱状图可以得知，在处理正态分布数据时，在赋值次数方面与比较次数方面三种排序方法几乎与处理均匀分布的数据一致。综合看来，在处理均匀分布数据时，希尔排序>折半插入排序>直接插入排序

不同数据类型对算法的影响

**不同数据类型对算法的影响**



**图2.2.3.28三种插入排序对正态分布整型数据排序时间 图2.2.3.29三种插入排序对正态分布整型结构体数据排序时间**

****

**图2.2.3.30三种插入排序对正态分布双精度数据排序时间 图2.2.3.31三种插入排序对正态分布双精度结构体数据排序时间**

**分析：**

用三种插入排序处理相同数据构型（均匀分布型）不同数据类型的数据时，纵向对比，普通整型与双精度浮点型，结构体类型整型与双精度浮点型消耗的时间基本一致；横向对比，结构体类型的数据消耗的时间基本在普通类型数据的一倍以上，其主要原因是访问结构体数据所花费时间大于直接访问普通数据。由此可以得出结论：对于任何类型的数据，在插入排序的变体之中，希尔排序综合表现最佳，且处理不同数据类型的数据效率除了受到数据规模的影响外，还会受到访问数据的速度影响。

**小结**

在基于插入排序的三种优化之中，希尔排序整体表现最佳。在大多数情况下，折半插入排序表现都优于直接插入排序。若对算法稳定性无要求，则在这三者之中推荐使用希尔排序排序，否则建议使用折半插入排序。

#### 快速排序

卢

## 对不同存储方式的C-字符串排序

为了能够更好的排序且符合函数的使用习惯，通过使用函数重载，使得同一个函数名能够处理包括字符串在内的各种不同类型的数据。此外，为了防止重载出错，处理字符类型的排序函数必须位于处理多种数字类型的函数前，使传入数组指针和指针数组时优先被字符排序函数接收。若将处理多种数字类型的函数放在前，由于使用了模板，在传参过程中并不会导致报错而进入重载，导致无法处理字符数据得问题。

**代码展示：**

//传入字符串数组时，不论是数组指针还是指针数组，传入的都应是数据的真实首地址位

置

void InsertionSort\_Shell(char \*begin[], char \*end[])

{

    int h = 1, len = end - begin + 1;

    while (h < len / 3)

        h = 3 \* h + 1;

    while (h >= 1)

    {

        for (int i = h; i < len; i++)

        {

            for (int j = i; j >= h && strcmp(begin[j], begin[j - h]) < 0; j -= h) // NUM为定义的数组长度

            {

                char tmp[NUM];

                strcpy(tmp, begin[j]);

                strcpy(begin[j], begin[j - h]);

                strcpy(begin[j - h], tmp);

            }

        }

        h = h / 3;

    }

}

void InsertionSort\_Shell(char \*begin, char \*end)

{

    int h = 1, len = end - begin + 1;

    while (h < len / 3)

        h = 3 \* h + 1;

    while (h >= 1)

    {

        for (int i = h; i < len; i++)

        {

            for (int j = i; j >= h && strcmp(begin + j \* NUM, begin + (j - h) \* NUM) < 0; j -= h)  // NUM为定义的数组长度

            {

                char tmp[NUM];

                strcpy(tmp, begin + j \* NUM);

                strcpy(begin + j \* NUM, begin + (j - h) \* NUM);

                strcpy(begin + (j - h) \* NUM, tmp);

            }

        }

        h = h / 3;

    }

}

template <typename T>

void InsertionSort\_Shell(T begin, T end) {……}

通过使用地址访问，只需要在传入参数时注意传入字符串的实际头指针，这一函数第一个重载可以处理指针数组类型的二维字符串，不论其中的指针指向的字符串内存地址是否连续，而第二个重载负责排序数组指针类型的二维字符串，由于数组指针的字符串地址连续，因而与指针数组有不同的处理方法。且希尔排序具有较良好的效率，能较快的完成字符串的排序。

## 多种排序算法的联合测试

周

# 乐器数字接口（MIDI）程序设计

## 自定义功能函数介绍

1. 将光标移动至屏幕指定位置

void gotoxy(int x, int y)

{

    COORD pos = {x,y};

    HANDLE hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

    SetConsoleCursorPosition(hOut, pos);

}

在C语言中conio.h头文件中有与该函数名称、功能完全相同的函数，但C++的库中取消了这个函数。由于本项目许多输出内容需要显示在屏幕指定位置，该函数非常重要，因此本小组自定义了该功能函数。

使用样例：

gotoxy(74, 15);

cout << "输入不合法！";

此语句段意为，在屏幕(74, 15)处输出“输入不合法！”

1. 改变屏幕输出字符的颜色

void COLOR(short x)

{

  if(x>=0 && x<=15)

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), x);

  else

    SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), x%15 + 1);

}

排序可视化项目中，可视化部分需要不同颜色的柱状色块表示数字；而音乐播放器项目需要不同颜色的色块代表不同声部。因此，本小组自定义该函数以实现改变字符颜色的功能。

使用样例：

COLOR(15);

cout << "<-";

此语句段意为：输出一个颜色为15(白色)的字符串“<-”

1. 播放声音

//            音量， 1为100%    音色，1为钢琴    通道         音高

void play(double volume\_percent, int timbre, int channel, int pitch, HMIDIOUT handle, int pause)

{

    int volume = volume\_percent \* 0x7f;//设置音量

    midiOutShortMsg(handle, timbre<<8 | 0xC<<4 | channel); //设置乐器

    int msg = (volume << 16 | (BASE + pitch) << 8 | 0x9<<4 | channel);

    midiOutShortMsg(handle, msg); //发声

    Sleep(pause);

}

虽然midiOutShortMsg()函数能实现播放声音，但其参数是一个4字节16进制数，调用很麻烦。因此本小组设计该函数用于发声，且能更容易调用。函数参数自左至右代表：音量百分比(double类型 0-1,0代表不发声，1代表最高音量)；乐器编号(0-127)；发声所用通道；音高(低/高于基础音高的值，如基础音高为36, pitch为2，则发声时音高为38)；句柄；发声后暂停时间（主要用于音乐播放器）

使用样例：

BASE = 36

play(0.9, 1, 1, 2, handle, 500);

此语句段意为：以90%的音量从1声道用明亮的钢琴声发出C大调re

## 快速排序及其优化方案的可视化

#### 程序运行效果及部分实现方法、代码

1. 部分函数介绍
2. 在规定位置显示色块

void show(num a, int location, int flag, Music m);

num、Music为自定义结构体，包含数字对应色块的颜色、数字代表的音高、此次显示需发出的声音的参数（乐器、声道、响度、暂停时间）。

location代表该数字在屏幕中需显示的位置。

flag用于确定显示样式。Flag为0时色块显示在屏幕左侧，为普通显示；为1时色块显示在右侧，用于表示基准值；为3时显示在左侧并在最后输出“<-”，用于表示被选中参与比较。

代码如下：

void show(num a, int location, int flag, Music m)

{

    int i;

    if(flag == 0 || flag == 2)i = 10;

    if(flag == 1) i = 20 + 2 \* MAX\_num;//根据flag确定显示位置的横坐标（左、右）

    gotoxy(i - 3, location + 3);//将光标移动至显示位置

    COLOR(15);

    cout << a.n;//输出白色数字

    //输出数字对应颜色的色块

    gotoxy(i, location + 3);

    COLOR(a.color);

    for(int j = 1; j <= 2 \* a.n; j++)

    {

        cout << "▋";

    }

    if(flag == 2)

    {

        gotoxy(i + 2 \* a.n + 1, location + 3);

        COLOR(15);

        cout << "<-";

    }

    //根据参数调用发声函数

    play(m.volume\_percent, m.timbre, m.channel, a.pitch, m.handle, m.pause);

    return;

}

1. 删除规定位置的色块

void del\_show(num a, int location, int flag)

参数意义同上，由于删除不发出声音，因此缺少Music型参数。该函数通过在指定位置输出空格覆盖原图像以达到删除目的。

代码与上述函数相近，不展示。

1. 开始界面

程序开始运行后，弹出输入数据规模界面。若输入超过45，则自动更改为45。数据规模设为45的原因是，该程序每个数字占据一行，若数据量超过45，可能会造成在部分设备上无法显示完全的情况。



图 3.2.1 排序可视化输入数据规模界面

该程序提供“完全随机序列”、“递减序列”、“大量重复元素序列”三个选项。为避免非法输入造成程序崩溃，选择界面被设置成只能通过键盘⬆、⬇两个按键切换选项，通过enter键确定选项。

该功能通过gotoxy函数实现选择箭头在指定位置显示，通过\_getch()与\_kbhit()函数获取键盘输入信息。

在箭头位置改变后，为使原位置箭头图标消失，本组通过在原位置输出两个空格覆盖实现。



图3.2.2 选择生成数据类型界面

代码如下：

int n = 1;//默认选项为第一个选项

gotoxy(76, 22);

cout << "->";//在第一个选项处输出箭头（初始化）

while(true)

{

    if(\_kbhit())//有按键被按下

    {

        int ch = \_getch();//获取被按下按键的键值

        gotoxy(76, 21 + n);

        cout << "  ";//在原位置输出空格覆盖旧箭头图标

        if(ch == 72)n--;//上箭头，待选项变为上一项

        if(ch == 80)n++;//下箭头

        if(n <= 0)n = 1;//防止选项越界，下同

        if(n >= 4)n = 3;

        if(ch == 13)break;//若按下enter键，结束循环

        gotoxy(76, 21 + n);

        cout << "->";//在新位置输出箭头

    }

}

return n;//返回选择的项

选择数据类型后，进入数据生成界面，同时生成待排序数据。

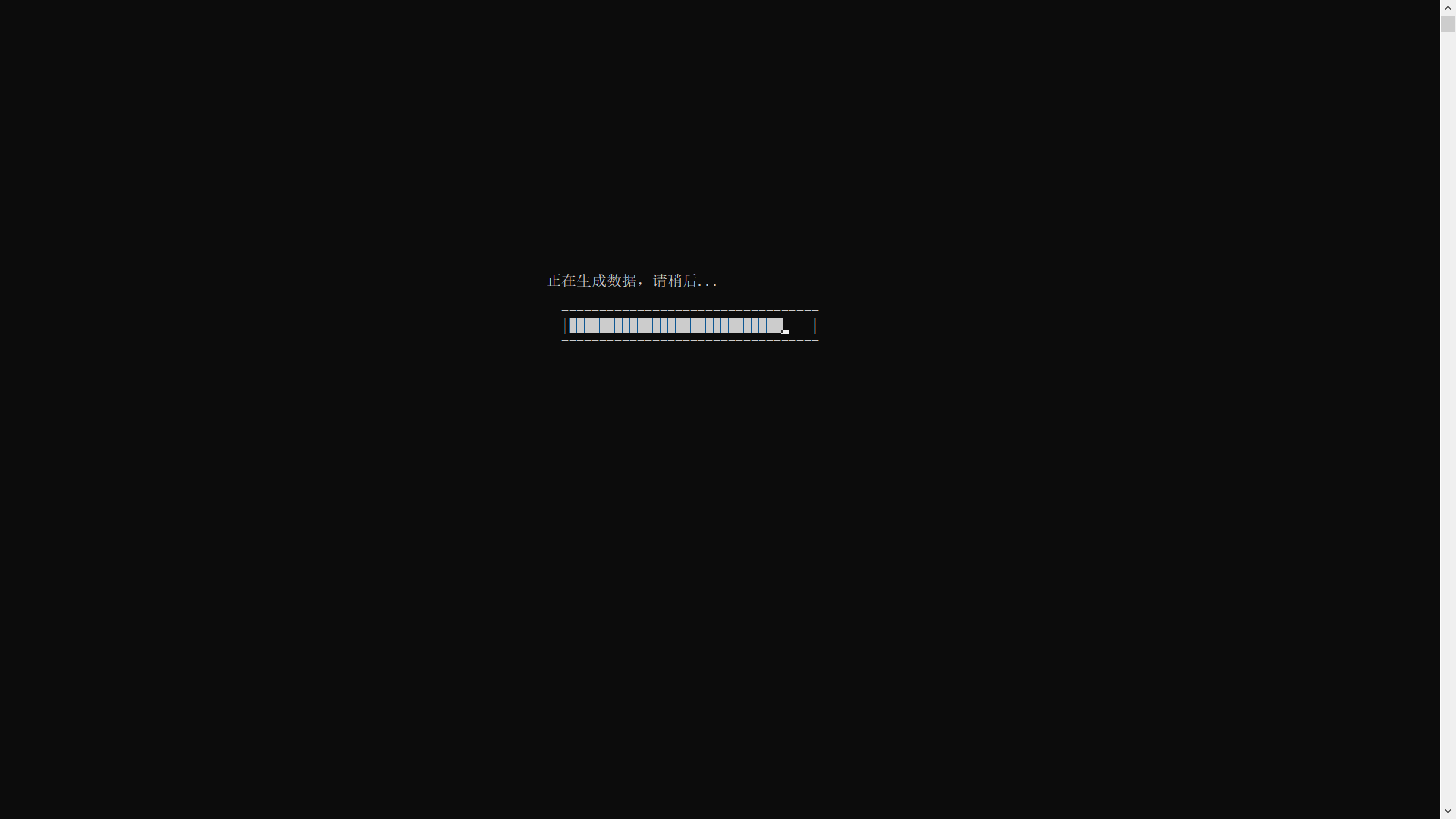


图3.2.3 数据生成界面

数据生成完成后，进入选择排序方法界面。本程序提供“普通快速排序”、“三数取中法快速排序”、“三切分快速排序”三种选项。选择界面具体实现同上。



图3.2.4 排序方式选择界面

1. 排序可视化界面

在选择好数据类型与排序方式后，程序会在屏幕上显示可视化界面。其中包括数据规模，原始数据（乱序）。数字后有一段不同颜色的条状色块表示该数字，数字大，色块越长。每个数字及其色块间隔100毫秒依次显示，同时发出明亮的钢琴声，数字越大，音高越高。

这部分的难点在于，在排序之前并不知道该数字在序列中的位次，无法知道显示时发出音乐的音高。解决方法是，在数据生成时用a\_base[]数组复制原数据，先进行排序，以此确定每个数字在序列中的位置。

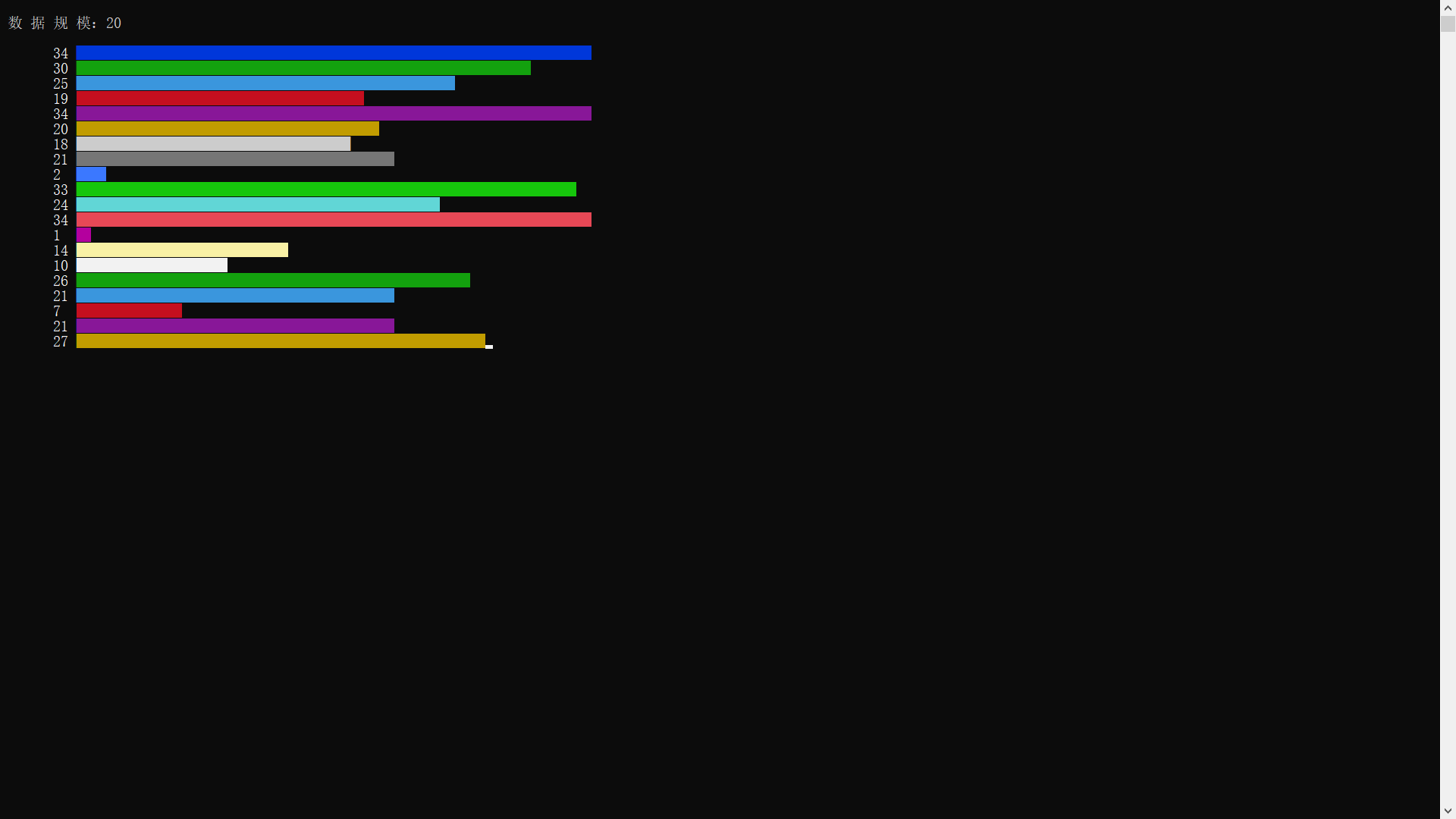


图3.2.5 原始数据的可视化

代码过于简单，不做展示。

初始化后，执行相应的排序算法。并在排序算法发生比较、交换时做出相应的可视化操作，并发出声音。以及在左侧表示出此时的左、右边界（三数取中法还有中间点的表示）。

声音包括：

数字参与比较时，发出明亮的钢琴声，音高为初始化时该数字对应的音高。

数字交换时，发出八音盒声，音高为其中一个数字对应的音高经运算后的值。这由于八音盒声在音高过低与过高时均不好听，因此对其对应音高做取余运算后再做加法，限定其在一个合适的范围内。

数字完成排序后（基准数字插入序列），发出尼龙弦吉他声，音高也为对应音高

当某个数字完成排序后，在其色块后输出字符串“-finish”。该数字不再参与排序。

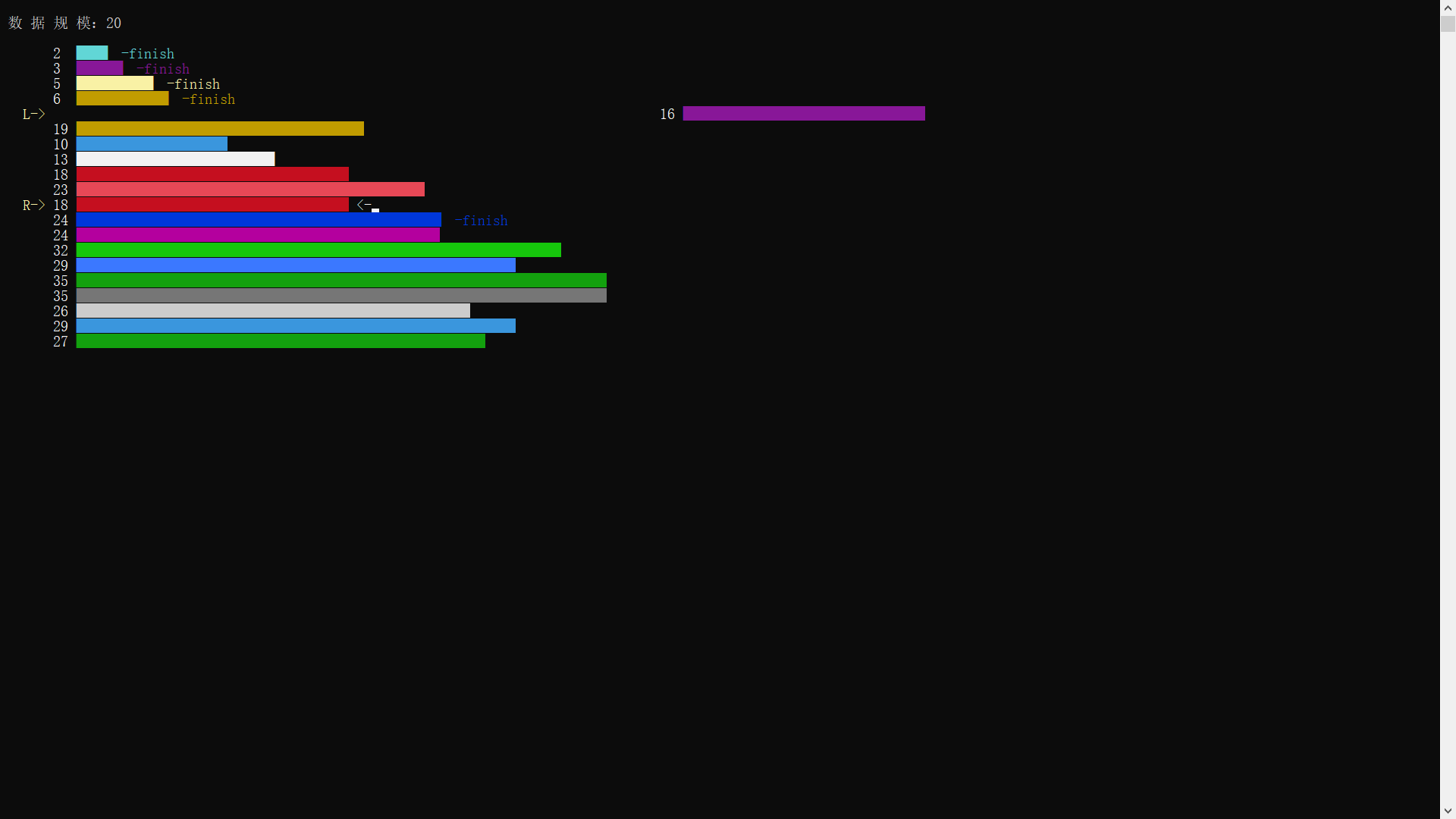


图3.2.6 普通快速排序在排序过程中的可视化

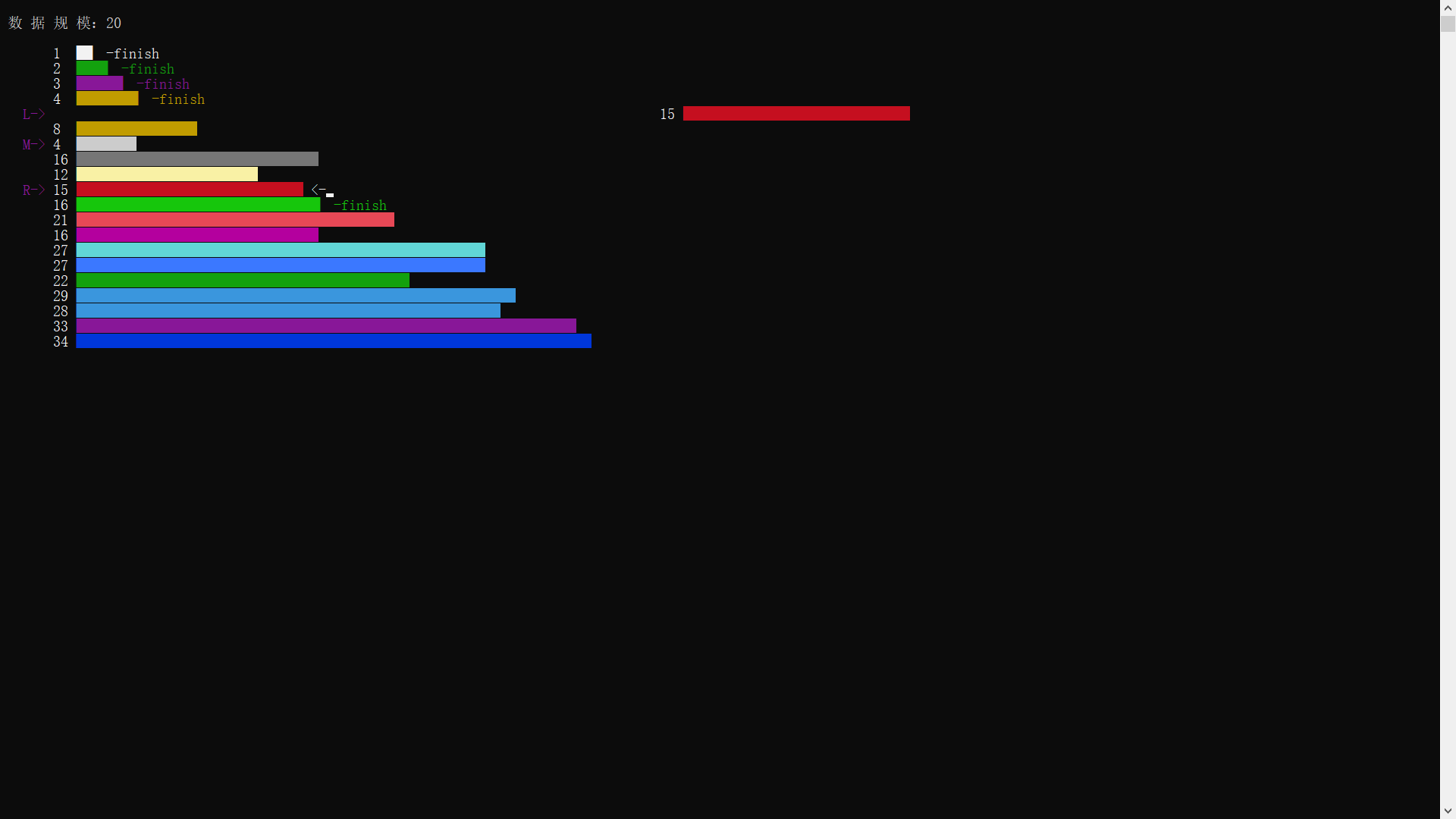


图3.2.7 三数取中法快速排序在排序过程中的可视化

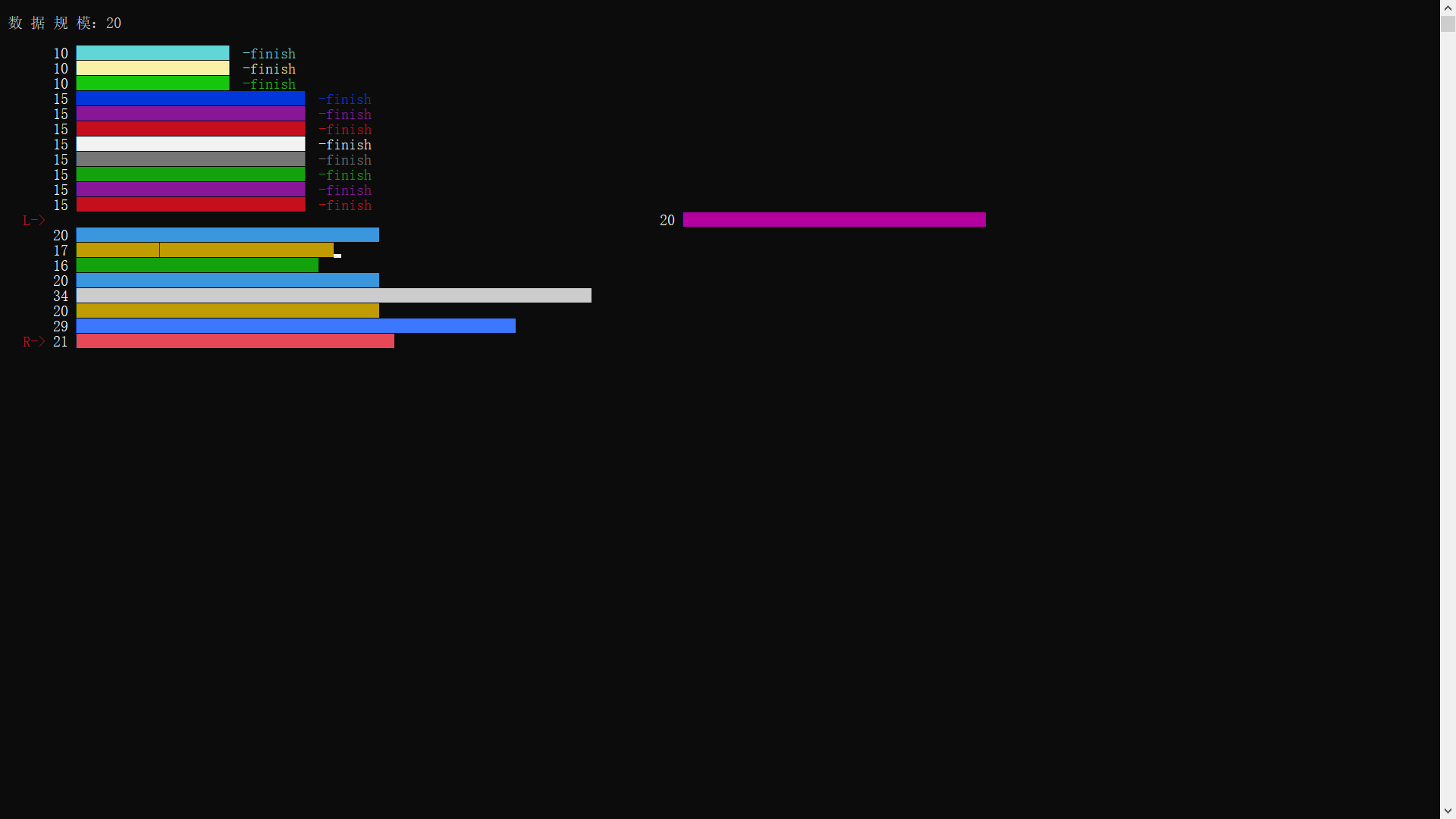


图3.2.8 三切分快速排序在排序过程中的可视化

代码过长，不做展示。

完成排序后，再依次按顺序发出数字对应的声音。此时由于数列顺序排列，声音为音阶。

最后，提供退出与重来选项，并执行对应操作。

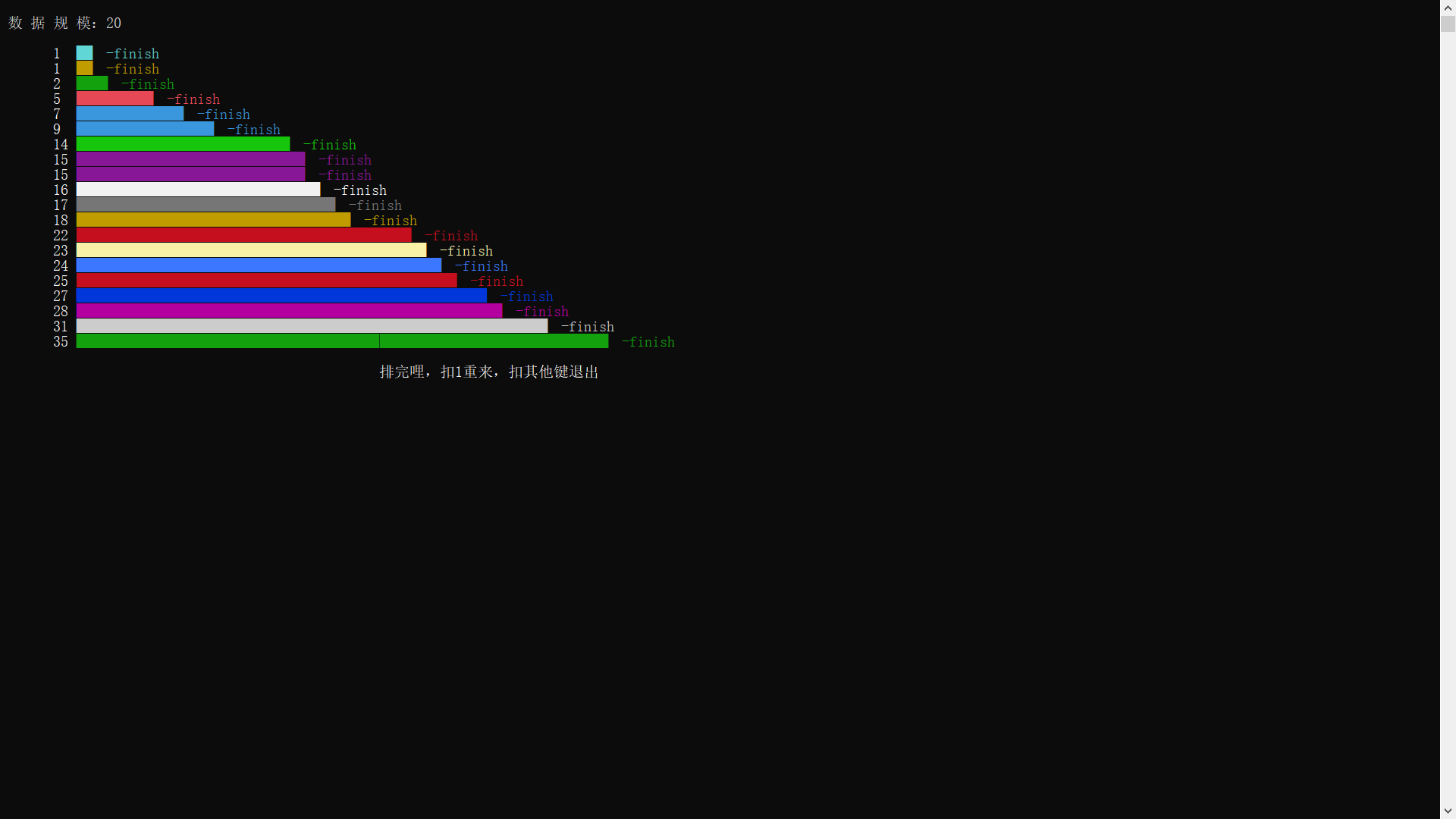


图3.2.9 排序完成时界面

#### 小结

实现排序可视化并不困难，根据原有的排序代码删除、显示色块，并发出对应声音即可。实现过程中仅在确定色块输出位置时出现过错误，并很快被修复。

## 音乐程序设计

#### 程序运行效果及部分实现方法、代码

1. 部分函数介绍

#### 难点及解决办法

# 结语

卢

**致谢**

**参考文献**

1. [1]谢婷.希尔排序算法设计思想研究[J].铜陵学院学报,2016,15(02):111-112+126.DOI:10.16394/j.cnki.34-1258/z.2016.02.027. [↑](#endnote-ref-1)