**数据结构与算法分析**

**课程设计报告**

**课题名称： 排序算法的实现与性能比较分析**

**基于哈夫曼编码的文本压缩与解压缩**

**班 级：**

**小组成员1：学号： 姓名：**

**小组成员2：学号： 姓名：**

**小组成员3：学号： 姓名：**

**小组成员4：学号： 姓名：**

**课题一**  **排序算法的实现与性能比较分析**

1. **任务要求：**

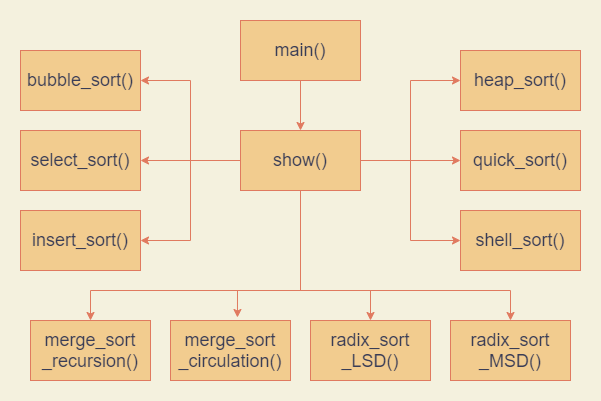
利用随机函数产生100000个随机整数，利用插入排序、起泡排序、选择排序、快速排序、堆排序、归并排序、基数排序、希尔排序排序方法进行排序，并统计每一种排序上机所花费的时间。

1. **系统功能：**
2. 利用数组建立线性表。
3. 利用rand()函数生成100000个随机数存储到数组中。
4. 利用函数指针结构体数组调用不同的函数，复制生成的随机数数组，对复制的数组进行排序。
5. 利用<windows.h>头文件中的**QueryPerformanceFrequency（）**函数和**QueryPerformanceCounter（）函数**，记录排序函数运行前的时间和排序后的时间，二者之差即为排序所花费的时间，实现精确的计时。
6. **系统方案：**

* 逻辑结构分析：逻辑结构为线性表：线性表是最基本、最简单、也是最常用的一种数据结构，数据元素之间的关系是一对一的关系。
* 存储结构设计：存储结构为顺序存储结构，具体为数组，顺序存储结构可以随机存取，

存储密度大

* 系统功能框图：



3-1 系统功能框图

1. **系统详细设计与实现**：

本程序共有8种排序，10个排序函数，函数定义如下：

void bubble\_sort(int \*arr, int left, int right);*// 冒泡排序*

void insert\_sort(int \*arr, int left, int right);*// 插入排序*

void select\_sort(int \*arr, int left, int right);*// 选择排序*

void shell\_sort(int \*arr, int left, int right);*// 基于sedgewick增量的希尔排序*

void quick\_sort(int \*arr, int left, int right);*// 快速排序*

void heap\_sort(int \*arr, int left, int right);*// 堆排序 递归实现*

void merge\_sort\_recursion(int \*arr, int left, int right);*// 归并排序 递归实现*

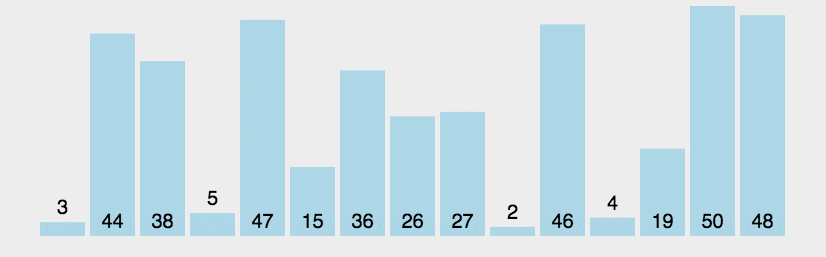
void merge\_sort\_circulation(int \*arr, int left, int right);*// 归并排序 循环实现*

void radix\_sort\_LSD(int \*arr, int left, int right);*// 基数排序 低位优先*

void radix\_sort\_MSD(int \*arr, int left, int right);*// 基数排序 高位优先*

（1）bubble\_sort冒泡排序

冒泡排序又称为泡式排序，是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他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。



4-1冒泡排序动画

*// 冒泡排序*

void bubble\_sort(int \*arr, int left, int right) {

  bool sorted;

*for* (int i = left; i < right; i++) {*// 第i+1趟*

    sorted = true;

*for* (int j = left; j < right - i;

         j++) {*// 从left到right - i 最大的数放在right-i位置上*

*if* (arr[j] > arr[j + 1]) {

        swap(arr[j], arr[j + 1]);

        sorted = false;

      }

    }

*if* (sorted == true) {

*return*;

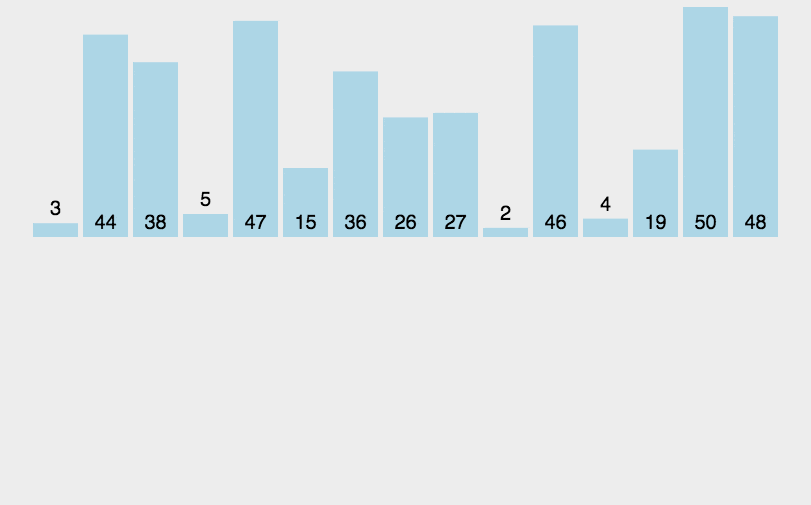
    }

  }

}

(2)insert\_sort插入排序

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，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。



4-1插入排序动画

void insert\_sort(int \*arr, int left, int right) {

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

    int temp = arr[i];*// 当前位置的值给temp*

    int j = i;

*for* (; j > 0 && arr[j - 1] > temp; j--) {

*// 假如arr[j-1]>temp 后移*

      arr[j] = arr[j - 1];

    }

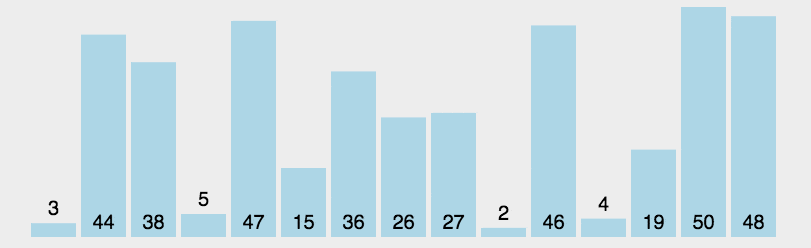
    arr[j] = temp;

  }

}

（3）select\_sort选择排序

选择排序是一种简单直观的排序算法。它的工作原理如下。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。



4-3选择排序动画

*// 选择排序*

void select\_sort(int \*arr, int left, int right) {

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

    int index = i;

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

*if* (arr[index] > arr[j]) {

        index = j;

      }

    }

*if* (index != i) {

      swap(arr[index], arr[i]);

    }

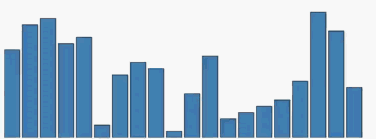
  }

}

（4）shell\_sort希尔排序

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

本程序的希尔排序是基于sedgewick增量序列的。



4-3希尔排序动画

*// 基于sedgewick增量的希尔排序*

void shell\_sort(int \*arr, int left, int right) {

*// 希尔增量序列*

  int sedgewick[15] = {146305, 64769, 36289, 16001, 8929, 3905, 929, 2161,

                       505,    209,   109,   41,    19,   5,    1};

  int h = 0, len = right - left + 1, temp;

*for* (h = 0; sedgewick[h] >= len; h++);

*for* (int i = sedgewick[h]; i > 0;

       i = sedgewick[++h]) {*// 不同的增量，增量递减*

*for* (int j = i; j < len; j++) {*// 间隔为i的插入排序*

      temp = arr[j];

      int k = j;

*for* (; k >= i && arr[k - i] > temp; k -= i) {

*// 假如arr[k-i]>temp 后移*

        arr[k] = arr[k - i];

      }

      arr[k] = temp;

    }

  }

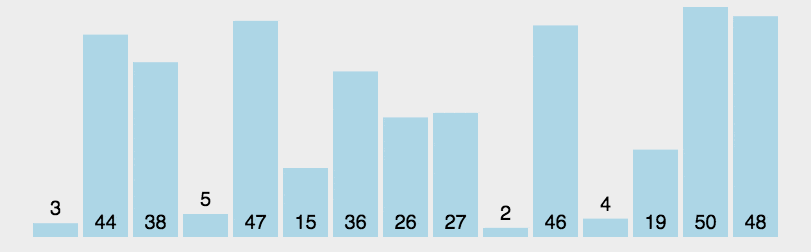
}

（5）quick\_sort快速排序

通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

算法步骤：

1. 从数列中挑出一个元素，称为 “基准”；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，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（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；



4-5快速排序动画

*// 快速排序*

void quick\_sort(int \*arr, int left, int right) {

*if* (left >= right) *return*;

  int i = left, j = right;*// i、j为两个“指针”*

  int base = arr[left];*// 基准*

*while* (i < j) {

*while* (i < j && arr[j] >= base) j--;*// 循环结束时 a[j] < base或i == j;*

*while* (i < j && arr[i] <= base) i++;*// 循环结束时 a[i] > base或i == j*

*if* (i < j) swap(arr[i], arr[j]);*// 交换*

  }

  swap(arr[left], arr[i]);*// 交换*

*// 递归对左右区间进行排序*

  quick\_sort(arr, left, i - 1);*// 还没有到右边界*

  quick\_sort(arr, i + 1, right);*// 还没有到左边界*

}

（6）归并排序（heap\_sort）

算法步骤：

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 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，并移动指针到下一位置
4. 重复步骤3直到某一指针超出序列尾
5. 将另一序列剩下的所有元素直接复制到合并序列尾

算法分析：

1. 对***n***个记录的文件进行归并排序，共需***log2n***趟
2. 每趟所需比较关键字的次数不超过***n***,共比较***O(nlog2n)***次
3. 每趟移动***n***个记录，共移动记录***O(nlog2n)***个
4. 归并排序需要 一个大小为***n***的辅助空间
5. 归并排序是稳定的

merge函数：

*// 将有序的arr[l]~arr[l-1]和arr[r]~arr[right\_end]归并成一个有序序列*

void merge(int \*arr, int \*temp\_arr, int l, int r, int right\_end) {

  int left\_end = r - 1;*// 左边终点位置*

  int temp = l, i = l;*// temp i有序序列的起始位置 i用于后期的复制*

*while* (l <= left\_end && r <= right\_end) {

*if* (arr[l] <= arr[r]) {

      temp\_arr[temp++] = arr[l++];*// 将左边元素复制到temp\_arr*

    } *else* {

      temp\_arr[temp++] = arr[r++];*// 将右边元素复制到temp\_arr*

    }

  }

*while* (l <= left\_end) {

    temp\_arr[temp++] = arr[l++];*// 直接复制左边剩下的*

  }

*while* (r <= right\_end) {

    temp\_arr[temp++] = arr[r++];*// 直接复制右边剩下的*

  }

*for* (; i <= right\_end; i++) {

    arr[i] = temp\_arr[i];

  }

}

递归实现：

void m\_sort(int \*arr, int \*temp\_arr, int l, int right\_end) {

*// 核心递归排序程序*

  int mid;

*if* (l < right\_end) {

    mid = (l + right\_end) / 2;

    m\_sort(arr, temp\_arr, l, mid);

    m\_sort(arr, temp\_arr, mid + 1, right\_end);

    merge(arr, temp\_arr, l, mid + 1, right\_end);

  }

}

*// 归并排序 递归实现*

void merge\_sort\_recursion(int \*arr, int left, int right) {

  int len = right - left + 1;

  int \*temp\_arr = new int[len];

*if* (temp\_arr != NULL) {

    m\_sort(arr, temp\_arr, left, right);

    delete temp\_arr;

  } *else* {

    cout << "Space is not enough!!!" << endl;

  }

}

循环实现：

*// 两两归并相邻有序子列*

void merge\_pass(int \*arr, int \*temp\_arr, int left, int right, int length) {

  int len = right - left + 1;*// 数组长度  length 合并的子序列的长度*

  int i, j;

*for* (i = left; i <= len - 2 \* length; i += 2 \* length) {

    merge(arr, temp\_arr, i, i + length, i + 2 \* length - 1);

  }

*if* (i + length <= right) {

    merge(arr, temp\_arr, i, i + length, right);

  } *else* {

*for* (j = i; j <= right; j++) {

      temp\_arr[j] = arr[j];

    }

  }

}

*// 归并排序 循环实现*

void merge\_sort\_circulation(int \*arr, int left, int right) {

  int len = right - left + 1;*// 归并的数组长度*

  int length = 1;*// 最开始的子序列长度*

  int \*temp\_arr = new int[len];

*if* (temp\_arr != NULL) {

*while* (length < len) {

      merge\_pass(arr, temp\_arr, left, right, length);

      length \*= 2;

      merge\_pass(temp\_arr, arr, left, right, length);

      length \*= 2;

    }

    delete temp\_arr;

  } *else* {

    cout << "Space is not enough!!!" << endl;

  }

}

（7）堆排序

算法思想：

1. 构建大顶堆（小顶堆）;
2. 输出大顶堆的根节点，交换堆顶和堆底元素；
3. 把剩余的i-1个元素整理成堆。

构建大根堆的基本思想：n表示元素的个数，从n/2位置开始遍历，直到遍历完所有根节点；

1. 若孩子节点都小于双亲节点，则调整结束。
2. 若存在孩子节点大于双亲节点，则将最大的孩子节点与双亲节点进行交换，并对孩子节点进行（1），（2）直到出现（1）或者叶节点为止。

算法分析：

1. 堆排序的时间，主要由建立初始堆和反复重建堆这两部分的时间开销构成
2. 堆排序的最坏时间复杂度为***O(nlog2n)***
3. 堆排序的平均性能较接近于最坏性能。由于建初始堆所需要的比较次数较多，所以堆排序不适宜于记录数较少的文件
4. 堆排序是就地排序，辅助空间为***O(1)***
5. 它是不稳定的排序方法

*// 最大堆的建立*

void perc\_down(int \*arr, int p, int len) {

  int parent, child, value = arr[p];*// 父节点 子节点 根节点存放的值*

*for* (parent = p; parent \* 2 + 1 < len; parent = child) {

    child = parent \* 2 + 1;

*if* (child != len - 1 && arr[child] < arr[child + 1]) {

      child++;*// child指向左右子节点的较大者*

    }

*if* (value >= arr[child]) {

*break*;

    } *else* {

      arr[parent] = arr[child];*// 下滤*

    }

  }

  arr[parent] = value;

}

*// 堆排序*

void heap\_sort(int \*arr, int left, int right) {

  int len = right - left + 1;

  int i;

*for* (i = len / 2 + 1; i >= 0; i--) {*// 建立最大堆*

    perc\_down(arr, i, len);

  }

*for* (i = len - 1; i >= 0; i--) {

*// 删除最大堆顶*

    swap(arr[0], arr[i]);

    perc\_down(arr, 0, i);

  }

}

（8）基数排序

基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。基数排序的发明可以追溯到1887年赫尔曼·何乐礼在打孔卡片制表机上的贡献。

*// 基数排序*

*// 桶元素节点*

const int MaxDigit = 5;*// 关键字*

const int Radix = 10;*// 基数*

typedef struct Node \*PtrToNode;

struct Node {

  int key;

  PtrToNode next;

};

*// 桶头节点*

struct HeadNode {

  PtrToNode head, tail;

};

typedef struct HeadNode Bucket[Radix];

*// 默认次位D=1, 主位D<=MaxDigit*

int get\_digit(int X, int D) {

  int d, i;

*for* (i = 1; i <= D; i++) {

    d = X % Radix;

    X /= Radix;

  }

*return* d;

}

次位优先（Least Significant Digit）

*// 基数排序 低位优先*

void radix\_sort\_LSD(int \*arr, int left, int right) {

  int d, di, i;

  Bucket B;

  PtrToNode temp, p, list = NULL;

*for* (i = 0; i < Radix; i++) {*// 初始化每个桶为空链表*

    B[i].head = B[i].tail = NULL;

  }

*for* (i = left; i <= right; i++) {*// 将原始序列逆序存入初始链表list*

    temp = new Node;

    temp->key = arr[i];

    temp->next = list;

    list = temp;

  }

*// 下面开始排序*

*for* (d = 1; d <= MaxDigit; d++) {*// 对数据得每一位循环处理*

*// 下面时分配的过程*

    p = list;

*while* (p) {

      di = get\_digit(p->key, d);

      temp = p;

      p = p->next;

      temp->next = NULL;

*if* (B[di].head == NULL) {

        B[di].head = B[di].tail = temp;

      } *else* {

        B[di].tail->next = temp;

        B[di].tail = temp;

      }

    }

*// 下面是收集的过程*

    list = NULL;

*for* (di = Radix - 1; di >= 0; di--) {

*if* (B[di].head) {

        B[di].tail->next = list;

        list = B[di].head;

        B[di].head = B[di].tail = NULL;

      }

    }

  }

*for* (i = left; i <= right; i++) {

    temp = list;

    list = list->next;

    arr[i] = temp->key;

    delete temp;

  }

}

主位优先（Most Significant Digit）

void MSD(int \*arr, int l, int r, int d) {

  int di, i, j;

  Bucket B;

  PtrToNode temp, list = NULL;

*if* (d == 0) {

*return*;

  }

*for* (i = 0; i < Radix; i++) {*// 初始化每个桶为空链表*

    B[i].head = B[i].tail = NULL;

  }

*for* (i = l; i <= r; i++) {*// 将原始序列逆序存入初始链表list*

    temp = new Node;

    temp->key = arr[i];

    temp->next = list;

    list = temp;

  }

*// 下面是分配的过程*

  PtrToNode p = list;

*while* (p) {

    di = get\_digit(p->key, d);

    temp = p;

    p = p->next;

*if* (B[di].head == NULL) {

      B[di].tail = temp;

    }

    temp->next = B[di].head;

    B[di].head = temp;

  }

*// 下面是收集的过程*

  i = j = l;

*for* (di = 0; di < Radix; di++) {

*if* (B[di].head) {

      p = B[di].head;

*while* (p) {

        temp = p;

        p = p->next;

        arr[j++] = temp->key;

        delete temp;

      }

      MSD(arr, i, j - 1, d - 1);

      i = j;

    }

  }

}

*// 基数排序 高位优先*

void radix\_sort\_MSD(int \*arr, int left, int right) {

  MSD(arr, left, right, MaxDigit);

}

1. **程序代码**

*#include* <ctime>

*#include* <fstream>

*#include* <iomanip>

*#include* <iostream>

*#include* <string>

*#include* <windows.h>

using namespace std;

void bubble\_sort(int \*arr, int left, int right);*// 冒泡排序*

void insert\_sort(int \*arr, int left, int right);*// 插入排序*

void select\_sort(int \*arr, int left, int right);*// 选择排序*

void shell\_sort(int \*arr, int left, int right);*// 基于sedgewick增量的希尔排序*

void quick\_sort(int \*arr, int left, int right);*// 快速排序*

void heap\_sort(int \*arr, int left, int right);*// 堆排序 递归实现*

void merge\_sort\_recursion(int \*arr, int left, int right);*// 归并排序 递归实现*

void merge\_sort\_circulation(int \*arr, int left, int right);*// 归并排序 循环实现*

void radix\_sort\_LSD(int \*arr, int left, int right);*// 基数排序 低位优先*

void radix\_sort\_MSD(int \*arr, int left, int right);*// 基数排序 高位优先*

void swap(int &a, int &b) {

  int temp = a;

  a = b;

  b = temp;

}

*// 冒泡排序*

void bubble\_sort(int \*arr, int left, int right) {

  bool sorted;

*for* (int i = left; i < right; i++) {*// 第i+1趟*

    sorted = true;

*for* (int j = left; j < right - i;

         j++) {*// 从left到right - i 最大的数放在right-i位置上*

*if* (arr[j] > arr[j + 1]) {

        swap(arr[j], arr[j + 1]);

        sorted = false;

      }

    }

*if* (sorted == true) {

*return*;

    }

  }

}

*// 插入排序*

void insert\_sort(int \*arr, int left, int right) {

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

    int temp = arr[i];*// 当前位置的值给temp*

    int j = i;

*for* (; j > 0 && arr[j - 1] > temp; j--) {

*// 假如arr[j-1]>temp 后移*

      arr[j] = arr[j - 1];

    }

    arr[j] = temp;

  }

}

*// 选择排序*

void select\_sort(int \*arr, int left, int right) {

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

    int index = i;

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

*if* (arr[index] > arr[j]) {

        index = j;

      }

    }

*if* (index != i) {

      swap(arr[index], arr[i]);

    }

  }

}

*// 基于sedgewick增量的希尔排序*

void shell\_sort(int \*arr, int left, int right) {

*// 希尔增量序列*

  int sedgewick[15] = {146305, 64769, 36289, 16001, 8929, 3905, 929, 2161,

                       505,    209,   109,   41,    19,   5,    1};

  int h = 0, len = right - left + 1, temp;

*for* (h = 0; sedgewick[h] >= len; h++)

    ;

*for* (int i = sedgewick[h]; i > 0;

       i = sedgewick[++h]) {*// 不同的增量，增量递减*

*for* (int j = i; j < len; j++) {*// 间隔为i的插入排序*

      temp = arr[j];

      int k = j;

*for* (; k >= i && arr[k - i] > temp; k -= i) {

*// 假如arr[k-i]>temp 后移*

        arr[k] = arr[k - i];

      }

      arr[k] = temp;

    }

  }

}

*// 快速排序*

void quick\_sort(int \*arr, int left, int right) {

*if* (left >= right) *return*;

  int i = left, j = right;*// i、j为两个“指针”*

  int base = arr[left];*// 基准*

*while* (i < j) {

*while* (i < j && arr[j] >= base) j--;*// 循环结束时 a[j] < base或i == j;*

*while* (i < j && arr[i] <= base) i++;*// 循环结束时 a[i] > base或i == j*

*if* (i < j) swap(arr[i], arr[j]);*// 交换*

  }

  swap(arr[left], arr[i]);*// 交换*

*// 递归对左右区间进行排序*

  quick\_sort(arr, left, i - 1);*// 还没有到右边界*

  quick\_sort(arr, i + 1, right);*// 还没有到左边界*

}

*// 最大堆的建立*

void perc\_down(int \*arr, int p, int len) {

  int parent, child, value = arr[p];*// 父节点 子节点 根节点存放的值*

*for* (parent = p; parent \* 2 + 1 < len; parent = child) {

    child = parent \* 2 + 1;

*if* (child != len - 1 && arr[child] < arr[child + 1]) {

      child++;*// child指向左右子节点的较大者*

    }

*if* (value >= arr[child]) {

*break*;

    } *else* {

      arr[parent] = arr[child];*// 下滤*

    }

  }

  arr[parent] = value;

}

*// 堆排序*

void heap\_sort(int \*arr, int left, int right) {

  int len = right - left + 1;

  int i;

*for* (i = len / 2 + 1; i >= 0; i--) {*// 建立最大堆*

    perc\_down(arr, i, len);

  }

*for* (i = len - 1; i >= 0; i--) {

*// 删除最大堆顶*

    swap(arr[0], arr[i]);

    perc\_down(arr, 0, i);

  }

}

*// 将有序的arr[l]~arr[l-1]和arr[r]~arr[right\_end]归并成一个有序序列*

void merge(int \*arr, int \*temp\_arr, int l, int r, int right\_end) {

  int left\_end = r - 1;*// 左边终点位置*

  int temp = l, i = l;*// temp i有序序列的起始位置 i用于后期的复制*

*while* (l <= left\_end && r <= right\_end) {

*if* (arr[l] <= arr[r]) {

      temp\_arr[temp++] = arr[l++];*// 将左边元素复制到temp\_arr*

    } *else* {

      temp\_arr[temp++] = arr[r++];*// 将右边元素复制到temp\_arr*

    }

  }

*while* (l <= left\_end) {

    temp\_arr[temp++] = arr[l++];*// 直接复制左边剩下的*

  }

*while* (r <= right\_end) {

    temp\_arr[temp++] = arr[r++];*// 直接复制右边剩下的*

  }

*for* (; i <= right\_end; i++) {

    arr[i] = temp\_arr[i];

  }

}

void m\_sort(int \*arr, int \*temp\_arr, int l, int right\_end) {

*// 核心递归排序程序*

  int mid;

*if* (l < right\_end) {

    mid = (l + right\_end) / 2;

    m\_sort(arr, temp\_arr, l, mid);

    m\_sort(arr, temp\_arr, mid + 1, right\_end);

    merge(arr, temp\_arr, l, mid + 1, right\_end);

  }

}

*// 归并排序 递归实现*

void merge\_sort\_recursion(int \*arr, int left, int right) {

  int len = right - left + 1;

  int \*temp\_arr = new int[len];

*if* (temp\_arr != NULL) {

    m\_sort(arr, temp\_arr, left, right);

    delete temp\_arr;

  } *else* {

    cout << "Space is not enough!!!" << endl;

  }

}

*// 两两归并相邻有序子列*

void merge\_pass(int \*arr, int \*temp\_arr, int left, int right, int length) {

  int len = right - left + 1;*// 数组长度  length 合并的子序列的长度*

  int i, j;

*for* (i = left; i <= len - 2 \* length; i += 2 \* length) {

    merge(arr, temp\_arr, i, i + length, i + 2 \* length - 1);

  }

*if* (i + length <= right) {

    merge(arr, temp\_arr, i, i + length, right);

  } *else* {

*for* (j = i; j <= right; j++) {

      temp\_arr[j] = arr[j];

    }

  }

}

*// 归并排序 循环实现*

void merge\_sort\_circulation(int \*arr, int left, int right) {

  int len = right - left + 1;*// 归并的数组长度*

  int length = 1;*// 最开始的子序列长度*

  int \*temp\_arr = new int[len];

*if* (temp\_arr != NULL) {

*while* (length < len) {

      merge\_pass(arr, temp\_arr, left, right, length);

      length \*= 2;

      merge\_pass(temp\_arr, arr, left, right, length);

      length \*= 2;

    }

    delete temp\_arr;

  } *else* {

    cout << "Space is not enough!!!" << endl;

  }

}

*// 基数排序*

*// 桶元素节点*

const int MaxDigit = 5;*// 关键字*

const int Radix = 10;*// 基数*

typedef struct Node \*PtrToNode;

struct Node {

  int key;

  PtrToNode next;

};

*// 桶头节点*

struct HeadNode {

  PtrToNode head, tail;

};

typedef struct HeadNode Bucket[Radix];

*// 默认次位D=1, 主位D<=MaxDigit*

int get\_digit(int X, int D) {

  int d, i;

*for* (i = 1; i <= D; i++) {

    d = X % Radix;

    X /= Radix;

  }

*return* d;

}

*// 基数排序 低位优先*

void radix\_sort\_LSD(int \*arr, int left, int right) {

  int d, di, i;

  Bucket B;

  PtrToNode temp, p, list = NULL;

*for* (i = 0; i < Radix; i++) {*// 初始化每个桶为空链表*

    B[i].head = B[i].tail = NULL;

  }

*for* (i = left; i <= right; i++) {*// 将原始序列逆序存入初始链表list*

    temp = new Node;

    temp->key = arr[i];

    temp->next = list;

    list = temp;

  }

*// 下面开始排序*

*for* (d = 1; d <= MaxDigit; d++) {*// 对数据得每一位循环处理*

*// 下面时分配的过程*

    p = list;

*while* (p) {

      di = get\_digit(p->key, d);

      temp = p;

      p = p->next;

      temp->next = NULL;

*if* (B[di].head == NULL) {

        B[di].head = B[di].tail = temp;

      } *else* {

        B[di].tail->next = temp;

        B[di].tail = temp;

      }

    }

*// 下面是收集的过程*

    list = NULL;

*for* (di = Radix - 1; di >= 0; di--) {

*if* (B[di].head) {

        B[di].tail->next = list;

        list = B[di].head;

        B[di].head = B[di].tail = NULL;

      }

    }

  }

*for* (i = left; i <= right; i++) {

    temp = list;

    list = list->next;

    arr[i] = temp->key;

    delete temp;

  }

}

void MSD(int \*arr, int l, int r, int d) {

  int di, i, j;

  Bucket B;

  PtrToNode temp, list = NULL;

*if* (d == 0) {

*return*;

  }

*for* (i = 0; i < Radix; i++) {*// 初始化每个桶为空链表*

    B[i].head = B[i].tail = NULL;

  }

*for* (i = l; i <= r; i++) {*// 将原始序列逆序存入初始链表list*

    temp = new Node;

    temp->key = arr[i];

    temp->next = list;

    list = temp;

  }

*// 下面是分配的过程*

  PtrToNode p = list;

*while* (p) {

    di = get\_digit(p->key, d);

    temp = p;

    p = p->next;

*if* (B[di].head == NULL) {

      B[di].tail = temp;

    }

    temp->next = B[di].head;

    B[di].head = temp;

  }

*// 下面是收集的过程*

  i = j = l;

*for* (di = 0; di < Radix; di++) {

*if* (B[di].head) {

      p = B[di].head;

*while* (p) {

        temp = p;

        p = p->next;

        arr[j++] = temp->key;

        delete temp;

      }

      MSD(arr, i, j - 1, d - 1);

      i = j;

    }

  }

}

*// 基数排序 高位优先*

void radix\_sort\_MSD(int \*arr, int left, int right) {

  MSD(arr, left, right, MaxDigit);

}

struct fun {

  void (\*fun\_point)(int \*, int, int);

  string fun\_name;

} fun\_arr[10] = {{bubble\_sort, "bubble\_sort"},

                 {insert\_sort, "insert\_sort"},

                 {select\_sort, "select\_sort"},

                 {shell\_sort, "shell\_sort"},

                 {quick\_sort, "quick\_sort"},

                 {heap\_sort, "heap\_sort"},

                 {merge\_sort\_recursion, "merge\_sort\_recursion"},

                 {merge\_sort\_circulation, "merge\_sort\_circulation"},

                 {radix\_sort\_LSD, "radix\_sort\_LSD"},

                 {radix\_sort\_MSD, "radix\_sort\_MSD"}};

*// 运行指定排序函数并将排好序的序列写入txt文件中*

void run(fun function, double &time, int arr[], int temp\_arr[], int left,

         int right) {

  int len = right - left + 1;

*for* (int i = 0; i < len; i++) {

    temp\_arr[i] = arr[i];

  }

*// 低精度计时 精确到1ms*

*/\**

*clock\_t start\_time, end\_time;*

*start\_time = clock();*

*function.fun\_point(temp\_arr, left, right);  // 执行相应的排序函数*

*end\_time = clock();*

*time = end\_time - start\_time;*

*cout << function.fun\_name << ":";*

*cout << time << "ms" << endl;*

*\*/*

  LARGE\_INTEGER start\_time, end\_time, time\_clock;*// time\_clock时钟频率*

  QueryPerformanceFrequency(&time\_clock);

  QueryPerformanceCounter(&start\_time);

  function.fun\_point(temp\_arr, left, right);*// 执行相应的排序函数*

  QueryPerformanceCounter(&end\_time);

  time = (double)((end\_time.QuadPart - start\_time.QuadPart) \* 1000) / (double)time\_clock.QuadPart;*// ms毫秒*

  cout << function.fun\_name << ":";

  cout << fixed << setprecision(2) << time << "ms" << endl;

  fstream out(function.fun\_name + ".txt", ios::out);

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

    out.setf(ios::left);

    out << setw(6) << temp\_arr[i] << " ";

  }

  out.close();

}

int main() {

  srand(time(NULL));

  int len = 100000;

  int arr[len];*// 随机数数组*

  int temp\_arr[len];*// 临时数组*

  int cnt = 1, fun\_size = 10;*// cnt:每个排序执行次数 fun\_size:函数个数*

  double time[fun\_size][cnt];*// 存储临时时间数组*

*// 每个排序的同一个随机数数组调用cnt次*

*for* (int k = 0; k < cnt; k++) {

*//cout << k + 1 << endl;*

*for* (int i = 0; i < len; i++) {

      arr[i] = rand();

    }

*for* (int i = 0; i < fun\_size; i++) {

      run(fun\_arr[i], time[i][k], arr, temp\_arr, 0, len - 1);

    }

  }

*// 写入时间*

  fstream out("time.txt", ios::out);

*for* (int i = 0; i < fun\_size; i++) {

    out << fun\_arr[i].fun\_name << ":";

*for* (int j = 0; j < cnt; j++) {

      out << fixed << setprecision(2) << time[i][j] << " ";

    }

    out << endl;

  }

*return* 0;

}

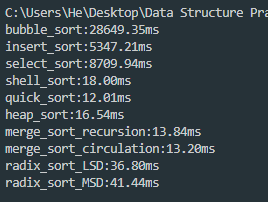
1. **实验结果及分析**

程序运行环境：处理器为i5-9300h，四核八线程，内存16GB

操作系统为win10，IDE为VSCODE

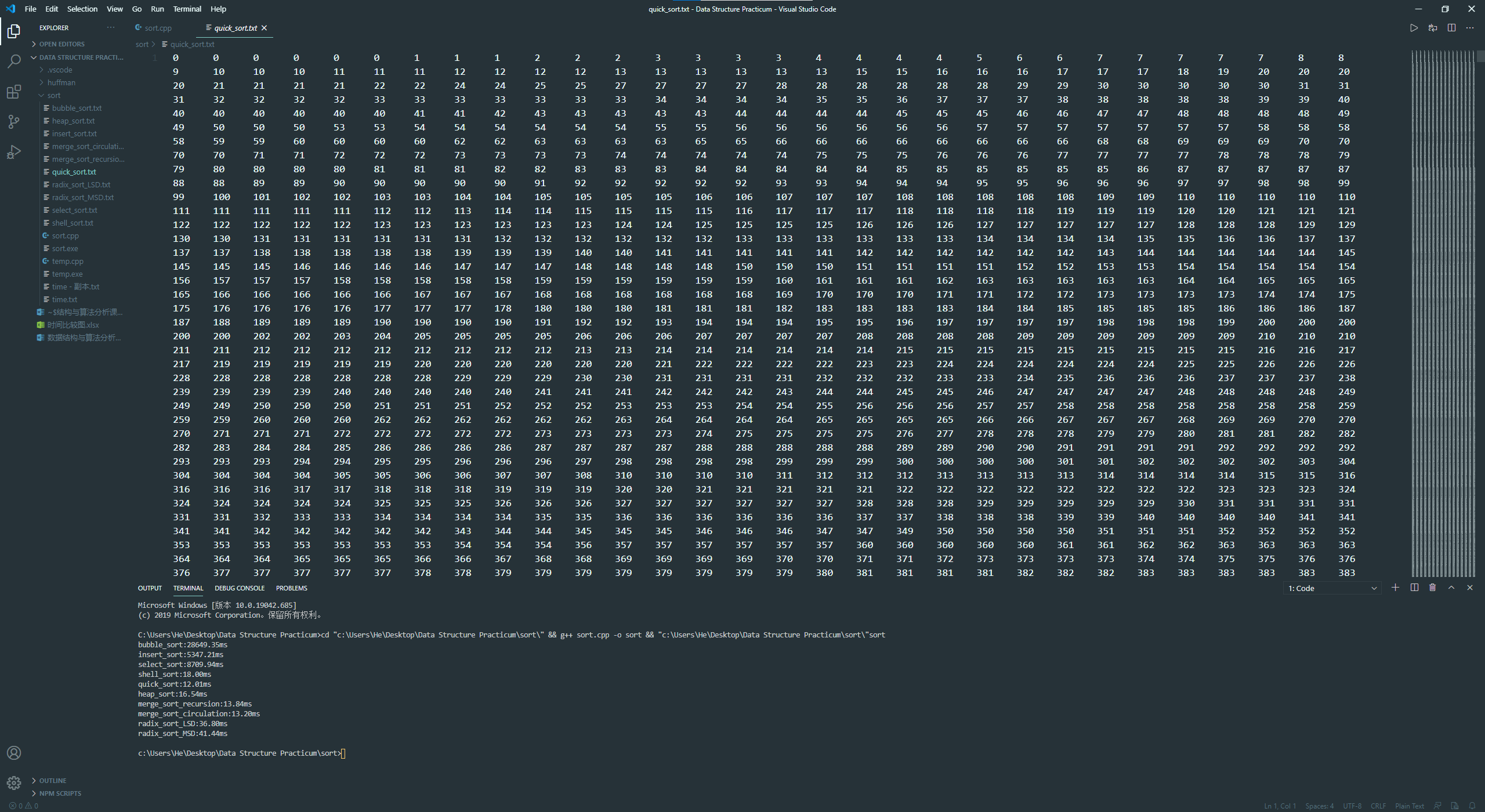
数据量：100000

运行时间：



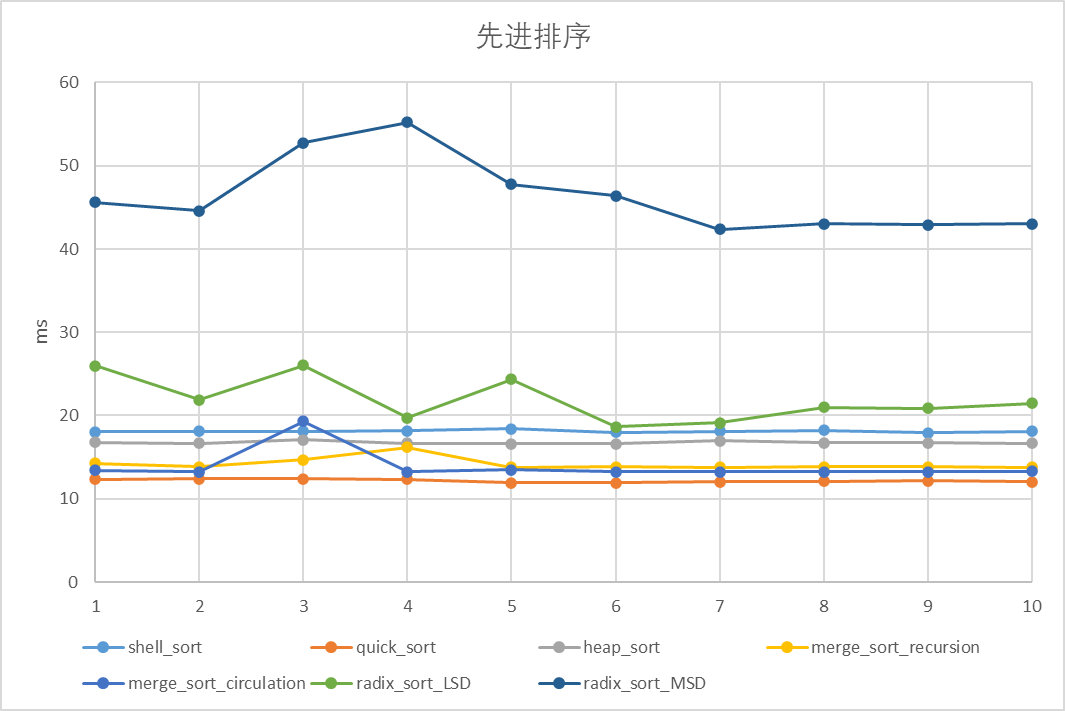
6-1运行时间图

排序结果：



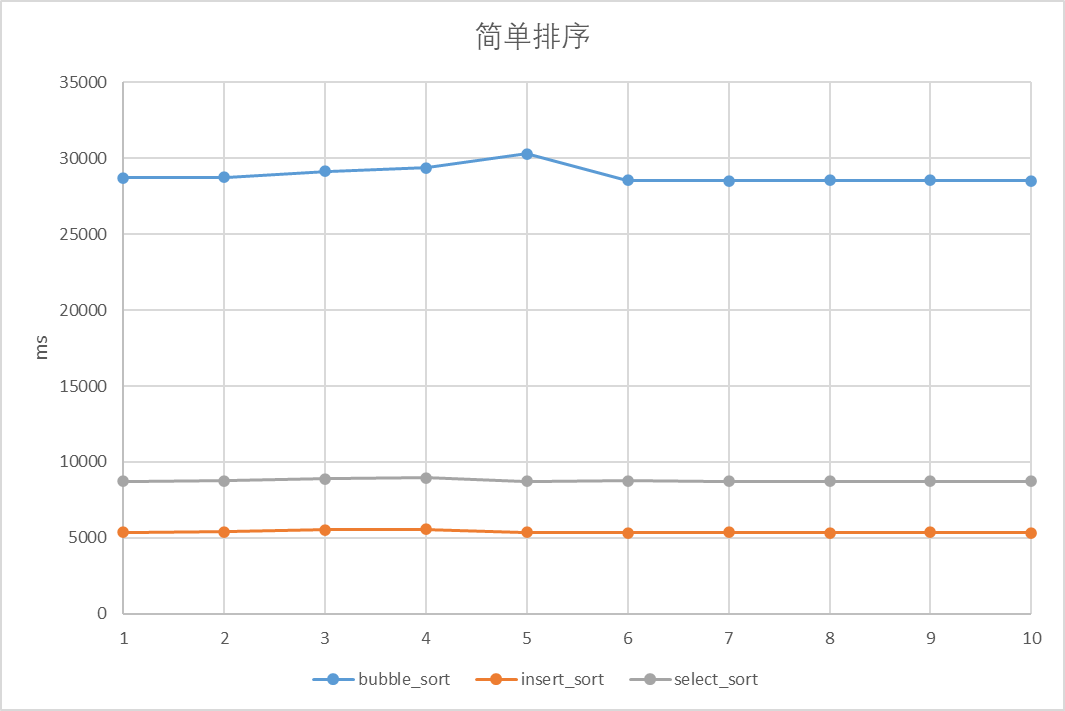
6-2排序结果图

先进排序时间对比：



6-3先进排序时间对比图

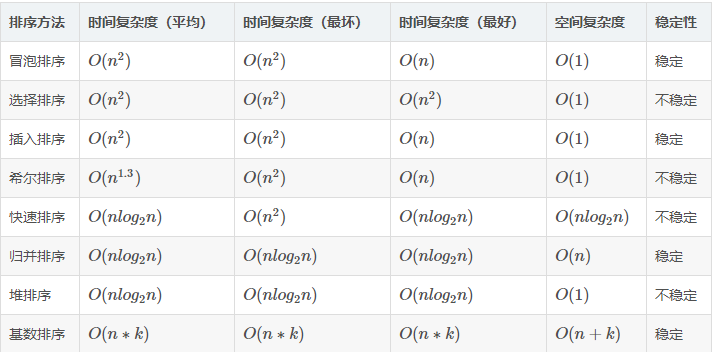
简单排序时间对比：



6-4简单排序时间对比图

时间复杂度及空间复杂度：

6-5算法对比表

****

在我们的排序过程中，快速排序始终是最快的，而平均时间复杂度和最坏时间复杂度为***O(log2n)***的堆排序和归并排序一直慢于快速排序。这是因为：

* 在快速排序中，每次数据移动都意味着该数据距离它正确的位置越来越近
* 在堆排序（大根堆）中，每次总是将最大的元素移除，然后将最后的元素放到堆顶，再让其自我调整。这样一来，想要维持大根堆，要进行很多次无效的比较
* 归并排序中有着大批量的数据复制

**课题二 基于哈夫曼编码的文本压缩与解压缩**

1. **任务要求：**

有data文件给出一段英语明文（不少于1000字母，含标点符号），设计程序将原文经过编码进行压缩，显示压缩都二进制代码（主要，是简单二进制数表示，应该bit位），再设计程序压缩后文件压缩。原理：统计字母及标点符号概率，定若干权值，建立哈夫曼树，并进行编码，将编码输出。

1. **系统功能：**

* 压缩

1. 对于一给定的文件，统计其不同值的字节出现的次数
2. 根据每个字节出现的次数，建立哈夫曼树
3. 根据哈夫曼树，得到哈夫曼编码
4. 根据得到的哈夫曼编码，对给定的文件进行压缩，并将huffman编码保存到哈夫曼编码文件中

* 解压

1. 读取压缩文件的哈夫曼编码
2. 根据哈夫曼编码，建立哈夫曼树
3. 根据压缩文件及其哈夫曼树，对文件进行解压
4. **系统方案：**

逻辑结构分析：哈夫曼树建立，编码和译码中字符权值采用的是逻辑结构为树形结构，具体为二叉树

存储结构设计：存储结构为链式存储结构，具体为二叉链表

压缩流程：

3-1压缩流程图

解压流程：

3-2解压流程图

1. **系统详细设计与实现：**

主要函数设计（类）：

class MyHuffman {

 private:

  string filepath;*// 要压缩的文件路径*

  long long byte\_value\_frequency[256] = {};*// 值为0-255的字节出现的次数*

  HuffmanNode \*root = NULL;*// 哈夫曼树根节点*

  string huffman\_code[256] = {};*// 值为0-255字节对应的哈夫曼编码*

  long long bit\_size = 0;*// 压缩文件bit总数*

 public:

  MyHuffman(string filepath);

  ~MyHuffman();

*// 压缩所用函数*

  void count\_byte\_frequency();*// 统计不同值得字节出现次数*

  void create\_huffman\_tree();*// 创建哈夫曼树*

*// 成员函数的默认参数只能在类内声明*

*// 根据哈夫曼树得到哈夫曼编码*

  void huffman\_tree\_to\_huffman\_code(HuffmanNode \*root = NULL, string code = "");

  void compress();*// 压缩*

*// 解压所用函数*

  void get\_huffman\_code();*// 读取huffman文件得到哈夫曼编码*

  void huffman\_code\_to\_huffman\_tree();*// 根据哈夫曼编码构建哈夫曼树*

  void decompress();*// 解压*

  void show\_huffman\_tree();*// 展示哈夫曼树*

  void show\_huffman\_code();*// 展示哈夫曼编码*

};

二叉链表节点设计：

struct HuffmanNode {

  unsigned char value = 0;

  int frequency = 0;

  struct HuffmanNode \*left = NULL;

  struct HuffmanNode \*right = NULL;

};

void count\_byte\_frequency();*// 统计不同值得字节出现次数*

该函数的功能是记录每个字节出现的次数

实现方法：遍历整个文件，碰到不同的字节便在数组相应的位置中+1

void create\_huffman\_tree();*// 创建哈夫曼树*

该函数的功能是根据字节出现的次数建立哈夫曼树

实现方法：根据字节出现的次数，利用 Huffman 编码思想创建 Huffman 树，将所记录的字节的频率作为权值来创建 Huffman 树，依次选择权值最小的两个字节作为左右孩子，其和作为父结点的权值，依次进行下去，直到所有的字节结点都成为叶子结点

*// 根据哈夫曼树得到哈夫曼编码*

  void huffman\_tree\_to\_huffman\_code();

该函数的功能是根据哈夫曼树得到哈夫曼编码

实现方法：根据创建的Huffman树来确定每个字节的哈夫曼编码，左孩子为 0，右孩子为1，类似于树的前序遍历，当达到叶子节点时，记录哈夫曼编码

void compress();*// 压缩*

该函数的功能是压缩文件

实现方法：读取文件的每个字节，在哈夫曼编码数组中找到对应的哈夫曼编码，用unsinged char存储哈夫曼编码，当unsinged char中记录了8bit时，写入unsinged char即一个字节，清空unsinged char，如此反复，直到文件被读完

void get\_huffman\_code();*// 读取huffman文件得到哈夫曼编码*

该函数的功能是读取保存在文件中的哈夫曼编码

实现方法：读取文件中的信息即可

void huffman\_code\_to\_huffman\_tree();*//**根据哈夫曼编码构建哈夫曼树*

该函数的功能是根据哈夫曼编码构建哈夫曼树

实现方法：先创建根节点，再读取每个字节的哈夫曼编码，‘0’代表右节点，‘1’代表左节点，若节点不存在就创建该节点，最后在叶子节点中写入字节的值

void decompress();*// 解压*

该函数的功能是解压

实现方法：先获取输入的路径中的后缀名，如”.txt”，再打开压缩文件和解压文件，从压缩文件中读入字节，在哈夫曼树寻找对应的叶子节点，没找到就继续读入下一个字节，找到了就将叶子节点对应的字节写入解压文件，继续寻找下一个叶子节点，如此反复，直到压缩文件被读完

**5程序代码**

MyHuffman.h：

*#ifndef* MYHUFFMAN\_H

*#define* MYHUFFMAN\_H

*#include* <iostream>

*#include* <string>

using namespace std;

struct HuffmanNode {

  unsigned char value = 0;

  int frequency = 0;

  struct HuffmanNode \*left = NULL;

  struct HuffmanNode \*right = NULL;

};

class MyHuffman {

 private:

  string filepath;*// 要压缩的文件路径*

  long long byte\_value\_frequency[256] = {};*// 值为0-255的字节出现的次数*

  HuffmanNode \*root = NULL;*// 哈夫曼树根节点*

  string huffman\_code[256] = {};*// 值为0-255字节对应的哈夫曼编码*

  long long bit\_size = 0;*// 压缩文件bit总数*

 public:

  MyHuffman(string filepath);

  ~MyHuffman();

*// 压缩所用函数*

  void count\_byte\_frequency();*// 统计不同值得字节出现次数*

  void create\_huffman\_tree();*// 创建哈夫曼树*

*// 成员函数的默认参数只能在类内声明*

*// 根据哈夫曼树得到哈夫曼编码*

  void huffman\_tree\_to\_huffman\_code(HuffmanNode \*root = NULL, string code = "");

  void compress();*// 压缩*

*// 解压所用函数*

  void get\_huffman\_code();*// 读取huffman文件得到哈夫曼编码*

  void huffman\_code\_to\_huffman\_tree();*// 根据哈夫曼编码构建哈夫曼树*

  void decompress();*// 解压*

  void show\_huffman\_tree();*// 展示哈夫曼树*

  void show\_huffman\_code();*// 展示哈夫曼编码*

};

*#endif  // MYHUFFMAN\_H*

MyHuffman.cpp：

*#include* "MyHuffman.h"

*#include* <algorithm>

*#include* <fstream>

*#include* <iomanip>

*#include* <iostream>

*#include* <queue>

*#include* <string>

*#include* <vector>

using namespace std;

MyHuffman::MyHuffman(string filepath) { *this*->filepath = filepath; }

MyHuffman::~MyHuffman() {}

void MyHuffman::count\_byte\_frequency() {

  ifstream in(filepath, ios::in | ios::binary);

*if* (!in) {

    cout << "File is not exist!!!" << endl;

  } *else* {

    unsigned char temp;*// 临时变量 读入每个字节*

*while* (in.peek() != EOF) {

      in.read((char \*)&temp, sizeof(unsigned char));

      byte\_value\_frequency[temp]++;

    }

*/\* for (int i = 0; i < 256; i++) {*

*if (byte\_value\_frequency[i] != 0) {*

*cout << i << ":" << byte\_value\_frequency[i] << endl;*

*}*

*} \*/*

  }

  in.close();

}

struct cmp {*// priority\_queue比较级是用结构体, 而不是函数*

  bool operator()(const HuffmanNode &a,

                  const HuffmanNode &b) {*// 重载操作, 记住就行*

*return* a.frequency > b.frequency;*// 小的优先在前面, 因为是用"堆"去实现的,*

  }

};

void MyHuffman::create\_huffman\_tree() {*// 建造哈夫曼树*

  priority\_queue<HuffmanNode, vector<HuffmanNode>, cmp> que;*// 优先队列*

*for* (int i = 0; i < 256; i++) {

*if* (byte\_value\_frequency[i] > 0) {

      HuffmanNode \*temp = new HuffmanNode;

      temp->value = i;

      temp->frequency = byte\_value\_frequency[i];

*// temp->left = temp->right = NULL;*

      que.push(\*temp);

    }

  }

  HuffmanNode \*temp\_left, \*temp\_right, \*temp\_root;

*while* (que.size() >= 2) {*// 不只有一棵树*

    temp\_left = new HuffmanNode;

    temp\_right = new HuffmanNode;

    temp\_root = new HuffmanNode;

    \*temp\_left = que.top();

    que.pop();

    \*temp\_right = que.top();

    que.pop();

*// 父节点的频率为左右孩子节点之和*

    temp\_root->frequency = temp\_left->frequency + temp\_right->frequency;

    temp\_root->left = temp\_left;

    temp\_root->right = temp\_right;

    que.push(\*temp\_root);

  }

  root = temp\_root;

}

void MyHuffman::huffman\_tree\_to\_huffman\_code(HuffmanNode \*root, string code) {

*// 初始化root 仅在第一层执行*

*if* (root == NULL && code == "") root = *this*->root;

*if* (root == NULL) *return*;

*if* (root->left == NULL && root->right == NULL) {

    huffman\_code[root->value] = code;

  }

  huffman\_tree\_to\_huffman\_code(root->left, code + '0');*// 左分支为'0'*

  huffman\_tree\_to\_huffman\_code(root->right, code + '1');*// 右分支为'1'*

}

void MyHuffman::compress() {

  ifstream in(filepath, ios::in | ios::binary);

  ofstream out(filepath + ".compress", ios::out | ios::binary);

*if* (!(in && out)) {

    cout << "File is not exist or can't create!!!" << endl;

  } *else* {

    unsigned char temp\_read = 0;*// 临时变量 读入每个字节*

    unsigned char temp\_write = 0;*// 临时变量 写入每个字节*

    string temp\_str;*// 临时字符串 存储哈夫曼编码*

    long long temp\_write\_bit\_size = 0;*// 临时变量temp\_write所写的二进制位数*

*while* (in.peek() != EOF) {

      in.read((char \*)&temp\_read, sizeof(unsigned char));

      temp\_str = huffman\_code[temp\_read];

*for* (int i = 0; i < temp\_str.size(); i++) {

*// 位运算的预先级低*

        temp\_write = (temp\_write << 1) + (temp\_str[i] - '0');

        temp\_write\_bit\_size++;

        bit\_size++;

*if* (temp\_write\_bit\_size == 8) {*// 8bit为1字节*

          out.write((char \*)&temp\_write, sizeof(unsigned char));

          temp\_write\_bit\_size = 0;

        }

      }

    }

*// 还有bit没写进去*

*if* (temp\_write\_bit\_size != 0) {

*while* (temp\_write\_bit\_size != 8) {

        temp\_write = temp\_write << 1;

        temp\_write\_bit\_size++;

      }

      out.write((char \*)&temp\_write, sizeof(unsigned char));

    }

    in.close();

    out.close();

*// 写入哈夫曼编码文件*

    out.open(filepath + ".huffman", ios::out);

    out << "HuffmanCode:" << endl;

    int types = 0;

*for* (int i = 0; i < 256; i++) {

*if* (byte\_value\_frequency[i] > 0) {

        types++;

      }

    }

    out << "TypesOfCharacter: " << types << endl;

    out.setf(ios::left);

*for* (int i = 0; i < 256; i++) {

*if* (byte\_value\_frequency[i] > 0) {

        out << setw(5) << i << huffman\_code[i] << endl;

      }

    }

    out << "WriteBitSize: " << bit\_size << endl;

  }

}

void MyHuffman::get\_huffman\_code() {

  ifstream in\_huffman(filepath + ".huffman", ios::in);*// 以文本方式打开*

*// 读取哈夫曼编码信息*

  string temp\_str, code\_str;*// 临时字符串 哈夫曼编码*

  int types\_character, value;*// 字节种类 字节值*

  in\_huffman >> temp\_str >> temp\_str;*// 跳过不必要的信息*

  in\_huffman >> types\_character;*// 读入字节种数*

*for* (int i = 0; i < types\_character; i++) {

    in\_huffman >> value >> code\_str;

    huffman\_code[value] = code\_str;

  }

  in\_huffman >> temp\_str >> bit\_size;

  in\_huffman.close();

}

void MyHuffman::huffman\_code\_to\_huffman\_tree() {

  root = new HuffmanNode;

  HuffmanNode \*temp\_parent, \*temp\_child;*// temp\_child用于开辟空间*

*for* (int i = 0; i < 256; i++) {

*// 没有该字节的哈夫曼编码就跳过*

*if* (huffman\_code[i].size() == 0) *continue*;

    temp\_parent = root;

*for* (int j = 0; j < huffman\_code[i].size(); j++) {

*if* (huffman\_code[i][j] == '0') {

*if* (temp\_parent->left == NULL) {

          temp\_child = new HuffmanNode;*// 该节点不存在就开辟空间*

          temp\_parent->left = temp\_child;

        }

        temp\_parent = temp\_parent->left;

      } *else* *if* (huffman\_code[i][j] == '1') {

*if* (temp\_parent->right == NULL) {

          temp\_child = new HuffmanNode;*// 该节点不存在就开辟空间*

          temp\_parent->right = temp\_child;

        }

        temp\_parent = temp\_parent->right;

      }

    }

    temp\_parent->value = i;

  }

}

void MyHuffman::decompress() {

  string exten;*// 后缀名*

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

    exten.push\_back(filepath[i]);

*if* (filepath[i] == '.') *break*;

  }

  reverse(exten.begin(), exten.end());

*// 以二进制的方式打开文件*

  ifstream in\_compress(filepath + ".compress", ios::in | ios::binary);

  ofstream out\_decompress(filepath + ".decompress" + exten,

                          ios::out | ios::binary);

*if* (!(in\_compress && out\_decompress)) {

    cout << "File is not exist or can't create!!!" << endl;

  } *else* {

*/\* // 哈夫曼树不存在 读取哈夫曼编码*

*if (root == NULL) {*

*get\_huffman\_code();              // 读取哈夫曼编码*

*huffman\_code\_to\_huffman\_tree();  // 构建哈夫曼树*

*} \*/*

    unsigned char temp\_read = 0;*// 临时变量 读入每个字节*

    unsigned char temp\_write = 0;*// 临时变量 写入每个字节*

    long long now\_bit\_size = 0;*// 现在读入的bit总数*

    HuffmanNode \*temp = root;

*// show\_huffman\_code();*

*while* (in\_compress.peek() != EOF) {

      in\_compress.read((char \*)&temp\_read, sizeof(unsigned char));

*for* (int i = 7; i >= 0; i--) {

*if* ((temp\_read >> i & 1) == 0) {*// 该bit为0*

          temp = temp->left;

        } *else* *if* ((temp\_read >> i & 1) == 1) {*// 该bit为1*

          temp = temp->right;

        }

*// 找到了对应的字节*

*if* (temp->left == NULL && temp->right == NULL) {

          temp\_write = temp->value;

          out\_decompress.write((char \*)&temp\_write, sizeof(unsigned char));

          temp = root;

        }

        now\_bit\_size++;

*if* (now\_bit\_size == bit\_size) *break*;

      }

    }

    in\_compress.close();

    out\_decompress.close();

  }

}

void MyHuffman::show\_huffman\_code() {

*for* (int i = 0; i < 256; i++) {

*if* (huffman\_code[i].size() > 0) {

      cout << i << " " << huffman\_code[i] << endl;

    }

  }

}

main.cpp

*#include* <iostream>

*#include* "MyHuffman.h"

using namespace std;

void show();*//开始界面*

void option\_1();*// 压缩*

void option\_2();*// 解压*

int main() {

*while* (1) {

    show();

    int option;

    cin >> option;

*if* (option == 1) {

      option\_1();

    } *else* *if* (option == 2) {

      option\_2();

    } *else* {

*break*;

    }

*//\_sleep(5000);*

    system("cls");

  }

*return* 0;

}

void show() {

  cout << "------------哈夫曼编码------------" << endl;

  cout << "1 压缩文件" << endl;

  cout << "2 解压文件" << endl;

  cout << "0 退出" << endl;

  cout << "------------哈夫曼编码------------" << endl;

  cout << "请输入选项：";

}

void option\_1() {

  string filepath;

  cout << "请输入要压缩的文件路径：";

  cin >> filepath;

  MyHuffman huffman(filepath);

  huffman.count\_byte\_frequency();

  huffman.create\_huffman\_tree();

  huffman.huffman\_tree\_to\_huffman\_code();

  huffman.compress();

  cout << "压缩成功！！！" << endl;

  cout << "是否解压该压缩文件" << endl;

  cout << "1 是" << endl;

  cout << "2 否" << endl;

  int is\_decompress;

  cin >> is\_decompress;

*if* (is\_decompress == 1) {

    huffman.decompress();

    cout << "解压成功！！！" << endl;

  }

}

void option\_2() {

  string filepath;

  cout << "请输入被压缩原文件的路径：";

  cin >> filepath;

  MyHuffman Huffman(filepath);

  Huffman.get\_huffman\_code();

  Huffman.huffman\_code\_to\_huffman\_tree();

  Huffman.decompress();

  cout << "解压成功！！！" << endl;

}

**6实验结果及分析：**

程序运行环境：处理器为i5-9300h，四核八线程，内存16GB

操作系统为win10，IDE为VSCODE

运行截图：



6-1压缩截图

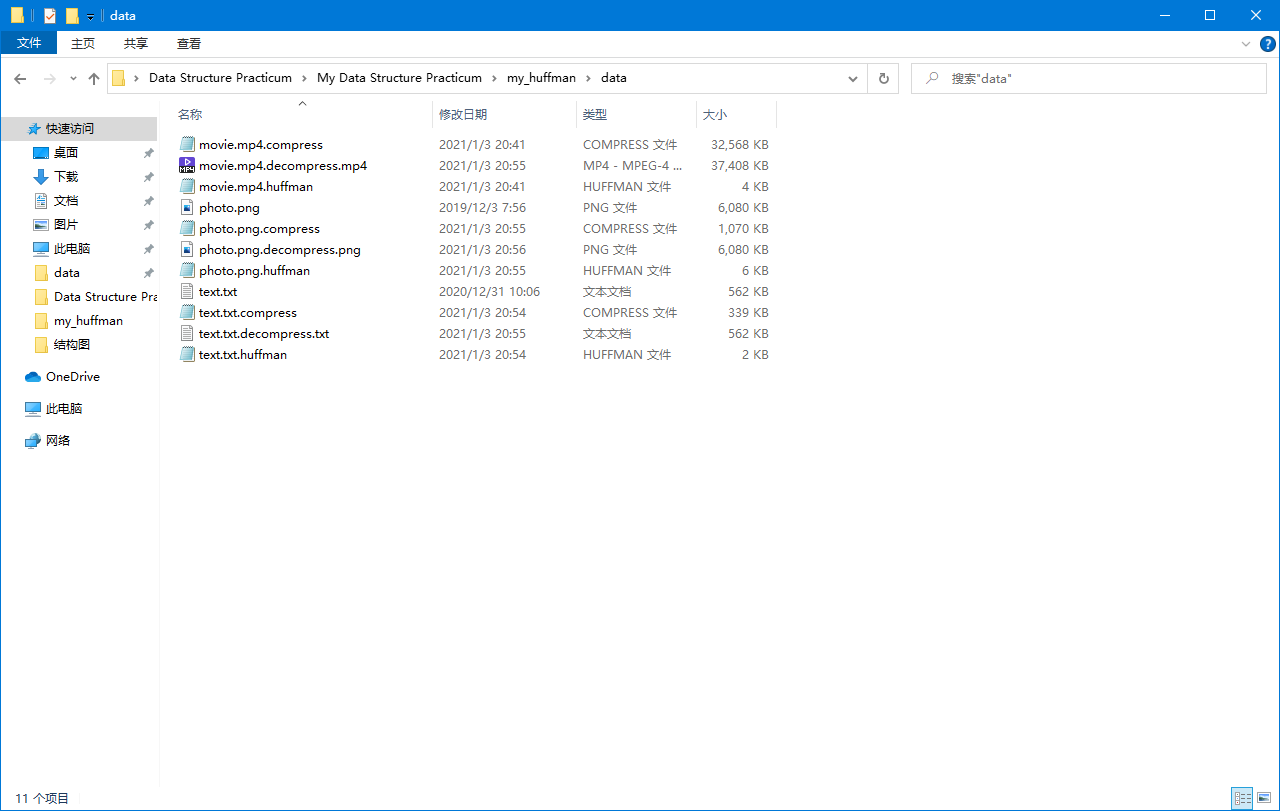
压缩结果：

**.xxx**为原文件

**.compress**为原文件

**.huffman**为对应的哈夫曼编码文件

**.decompress.xxx**为解压文件

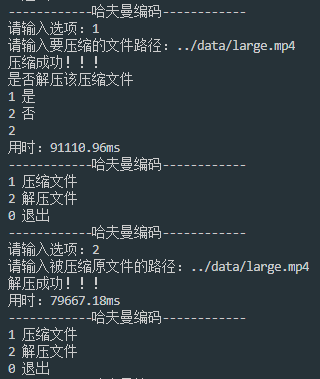


6-2压缩截图

运行时间：

压缩808 MB的视频用时91s

解压808 MB的压缩文件用时80s



6-3运行时间图