HW6实验报告

1.涉及数据结构和相关背景

- 排序相关及排序的应用
- 基本排序 $O(n^2)$
  - 选择排序
  - 冒泡排序
  - 插入排序
- 优化排序 $O(nlogn)$
  - 归并排序
  - 堆排序
  - 快速排序
- 特殊排序
  - 希尔排序

2. 实验内容

2.1 求逆序对数

2.1.1 问题描述

对于一个长度为N的整数序列A，满足$i < j$且 $A_i > A_j$的数对 (i, j) 称为整数序列A的一个逆序,请求出整数序列A的所有逆序对个数

2.1.2 基本要求

- 输入
  - 输入包含多组测试数据，每组测试数据有两行
  - 第一行为整数N(1 <= N <= 20000)，当输入0时结束
  - 第二行为N个整数，表示长为N的整数序列
- 输出
  - 每组数据对应一行，输出逆序对的个数

2.1.3 数据结构设计

- 考虑到排序的背景，可以考虑到在进行归并排序时，每次在合并两个有序数组时，每当后半部数组中有一个元素提前加入数组，就说明当前前半部还没有加入数组的所有元素都比当前这个后半部的元素要大，说明这些元素都与当前元素组成逆序对，将数量计入总数即可
- 临时排序数组

```
long long tmp[100009];
```

- 总体数组

```
int q[100005];
```

2.1.4 功能说明

- 进行修改版的归并排序

```
void ms(int q[], int l, int r)
{
    /*归并排序*/
    if (l >= r) return;
    int mid = l + r >> 1;
    ms(q, l, mid), ms(q, mid + 1, r);                //将左右两边分别排序

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)                       //遍历当前两个数组并按位进行比较
    {
        if (q[i] <= q[j])
        {
            tmp[k++] = q[i++];
        }

        else tmp[k++] = q[j++], ans += (mid - i + 1);    //当有后边的数字提前出去则前边数组剩下的部分都是逆序的
    }

    while (i <= mid) tmp[k++] = q[i++];
    while (j <= r) tmp[k++] = q[j++];

    for (int i = l, j = 0; i <= r; i++, j++) q[i] = tmp[j];//最后将临时数组写到原数组当中
}
```

2.1.5 调试分析

- 主要利用了归并排序的特点，将归并排序中合并过程的分治思想应用到了求解逆序对的数量上
- 主体是归并函数，同样因为应用到了归并的思想，整体的复杂度尽在 $O(nlogn)$ 的数量级

2.1.6 总结和体会

- 维护逆序对的数量，本题用分治的方法较为经典，时间复杂度控制同样可以满足要求
- 另一种方法是利用树状数组维护逆序对的数量

2.2 最大数

2.2.1 问题描述

- 给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。
- 注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数

2.2.2 基本要求

- 输入
  - 输入包含两行 第一行包含一个整数n，表示组数nums的长度
  - 第二行包含n个整数nums[i]对于100%的数据，$1 <= nums.size() <= 100$，$0 <= nums[i] <= 10^9$
- 输出
  - 输出包含一行，为重新排列后得到的数字

2.2.3 数据结构设计

- 最朴素想法是将每一个数字的最高位进行比较，，将数组中的数字进行排序
- 此方法仅仅局限于当所有位置的数字的首位均不一样的情况
- 正确方法是每次仅比较两个数字，选择两个数字的前后组合中更大的一组，在数组中更改成这个顺序进行排序

2.2.4 功能说明

- 进行相应排序

```
std::string largestNumber(std::vector<int>& nums)
    {
        sort(nums.begin(), nums.end(), [](const int &a, const int &b)
            {
                long long q1 = 10, q2 = 10;
                while (q1 <= a) q1 *= 10;
                while (q2 <= b) q2 *= 10;
                return q2 * a + b > q1 * b + a;
            }
        );
        if (nums[0] == 0)
            return "0";
        string rt;
        for (int i = 0; i < nums.size(); i++)
        {
            rt += to_string(nums[i]);
        }
        return rt;
    }
```

> 注意此处的sort函数中最后的谓词可以写成当前形式，代替伪函数的形式进行书写。
>
> 即
>
> ```
> sort(nums.begin(), nums.end(), [](type arg*){})
> ```

> 其中函数部分直接给出相应实现

2.2.5 调试分析

- 在调试过程中首先将首位直接进行排序，没有考虑到首位相同的情况以及不同位数的情况

2.2.6 总结和体会

- 数组的排序总是可以从数组中每两个元素之间的排序考虑起，如本题可以考虑到相邻两个元素之间如何进行排序的问题，将问题规模最小化

2.3 排序

2.3.1 问题描述

- 排序算法分为简单排序（时间复杂度为O(n^2))和高效排序（时间复杂度为O(nlog n)）。
- 本题给定N个整数，要求输出从小到大排序后的结果。请用不同的排序算法测试，注意有些算法无法拿到满分，由此同学们可以猜测一下10个测试用例的数据特征。
- 请同学们自己随机生成不同规模的数据（例如10，100，1K，10K，100K，1M，10K正序，10K逆序），用不同的排序算法（快速排序，归并排序，堆排序，选择排序，冒泡排序，直接插入排序，希尔排序）分别对这些数据进行测试，输出运行时间，将结果写在实验报告中，并总结各种排序算法的特点、时间复杂度，以及是否是稳定排序。

2.3.2 基本要求

- 输入
- 第1行一个正整数n，表示元素个数，第2行n个整数，用空格分割
- 输出
- 从小到大排序后的结果，以空格分割

2.3.3 数据结构设计

- 本题用 `python` 实现
- 设计时间计算装饰器 `cla_time`，对每一个排序函数进行计时
- 分别实现排序算法进行时间比较

2.3.4 功能说明

- 时间计算装饰器 `cal_time`

```
import time
from functools import wraps

def cal_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        func(*args, **kwargs)
        end_time = time.time()
        t = end_time - start_time
        print('{:<10}\t{:>40}:{} {}'.format(func.__name__, 'Running time', t
* 1000, 'ms'))

    return wrapper
```

> 装饰器形式较为简单，易于调用

- 初始化所有待排序列表(此处用深拷贝)

```
# Initialize a disorganized list
list_test = [random.randint(0, 1000) for _ in range(10000)]
print('Original List:\n', list_test)

list1 = copy.deepcopy(list_test)
list2 = copy.deepcopy(list_test)
list3 = copy.deepcopy(list_test)
list4 = copy.deepcopy(list_test)
list5 = copy.deepcopy(list_test)
list6 = copy.deepcopy(list_test)
list7 = copy.deepcopy(list_test)
list8 = copy.deepcopy(list_test)
list9 = copy.deepcopy(list_test)
list10 = copy.deepcopy(list_test)
```

> 保证每次随机列表且每一个排序的列表相同

- 冒泡排序

```
@cal_time
def bubble_sort(li):
    for i in range(len(li) - 1):
        exchange = False

        for j in range(len(li) - 1 - i):
            if li[j] > li[j + 1]:
                li[j], li[j + 1] = li[j + 1], li[j]
                exchange = True

        if not exchange:
            return
```

> 优化后的冒泡排序，当后边的数字不再交换时说明已经排好序了，提前退出

- 选择排序

```
@cal_time
def select_sort(li):
    for i in range(len(li) - 1):
        min_loc = i

        for j in range(i + 1, len(li)):
            if li[j] < li[min_loc]:
                min_loc = j

        li[i], li[min_loc] = li[min_loc], li[i]
```

- 插入排序

```
@cal_time
def insert_sort(li):
    for i in range(1, len(li)):
        tmp = li[i]
        j = i - 1

        while j >= 0 and tmp < li[j]:
            li[j + 1] = li[j]
            j -= 1

        li[j + 1] = tmp
```

- 快速排序

```
def partition(li, left, right):
    tmp = li[left]

    while left < right:
        if left < right and tmp <= li[right]:
            right -= 1
        li[left] = li[right]

        if left < right and tmp > li[left]:
            left += 1
        li[right] = li[left]

    li[left] = tmp
    return left
```

```
def _quick_sort(li, left, right):
    if left < right:
        mid = partition(li, left, right)
        _quick_sort(li, left, mid - 1)
        _quick_sort(li, mid + 1, right)


@cal_time
def quick_sort(li):
    _quick_sort(li, 0, len(li) - 1)
```

> 此处为保证调用接口的一致性多封装了几次

- 堆排序

```
def sift(li, low, high):
    i = low
    j = 2 * i + 1
    tmp = li[low]

    while j <= high:
        if j + 1 <= high and li[j] < li[j + 1]:
            j = j + 1

        if li[j] > tmp:
            li[i] = li[j]
            i = j
            j = 2 * i + 1

        else:
            break

    li[i] = tmp


@cal_time
def heap_sort(li):
    n = len(li)

    for i in range((n - 2) // 2, -1, -1):
        sift(li, i, n - 1)

    for i in range(n - 1, -1, -1):
        li[0], li[i] = li[i], li[0]
        sift(li, 0, i - 1)
```

- 归并排序

```
def merge(li, left, mid, right):
    i = left
    j = mid + 1
    ltmp = []

    while i <= mid and j <= right:
        if li[i] < li[j]:
            ltmp.append(li[i])
            i += 1

        else:
            ltmp.append(li[j])
            j += 1

    while i <= mid:
        ltmp.append(li[i])
        i += 1

    while j <= right:
        ltmp.append(li[j])
        j += 1

    li[left: right + 1] = ltmp


def _merge_sort(li, left, right):
    if left < right:
        mid = (left + right) // 2
        _merge_sort(li, left, mid)
        _merge_sort(li, mid + 1, right)
        merge(li, left, mid, right)


@cal_time
def merge_sort(li):
    _merge_sort(li, 0, len(li) - 1)
```

- 希尔排序

```
def per_shell(li, gap):
    for i in range(1, len(li)):
        tmp = li[i]
        j = i - gap

        while j >= 0 and tmp < li[j]:
            li[j + gap] = li[j]
            j -= gap

        li[j + gap] = tmp


@cal_time
def shell_sort(li):
```

```
    d = len(li) // 2

    while d >= 1:
        per_shell(li, d)
        d //= 2
```

- 计数排序

```
@cal_time
def count_sort(li, max_num=1000):
    count_table = [0 for _ in range(max_num + 1)]
    for var in li:
        count_table[var] += 1

    li.clear()
    for ind, var in enumerate(count_table):
        for _ in range(var):
            li.append(ind)
```

> 散列方式计数排序，需要知道列表中数的上限

- 桶排序

```
@cal_time
def bucket_sort(li, n=100, max_num=1000):
    buckets = [[] for _ in range(n)]
    for var in li:
        target_buc = min(var // (max_num // n), n - 1)
        buckets[target_buc].append(var)

        for i in range((len(buckets[target_buc])) - 1, 0, -1):
            if buckets[target_buc][i] < buckets[target_buc][i - 1]:
                buckets[target_buc][i], buckets[target_buc][i - 1] =
buckets[target_buc][i - 1], buckets[target_buc][i]
            else:
                break
    li.clear()
    for buc in buckets:
        li.extend(buc)
```

> 分块进行排序，每一块排序好后进行合并即可

- 基数排序

```
@cal_time
def radix_sort(li):
    max_num = max(li)
```

```
    ti = 0

    while 10 ** ti <= max_num:
        buckets = [[] for _ in range(10)]
        for var in li:
            digit = (var // 10 ** ti) % 10
            buckets[digit].append(var)

        li.clear()
        for buc in buckets:
            li.extend(buc)
        ti += 1
```

2.3.5 调试分析

- 进行小规模测试:0-50随机排列测试
- 结果:

```
main
Original List:
 [18, 30, 40, 25, 35, 18, 30, 32, 50, 49, 5, 20, 36, 50, 9, 40, 25, 41, 37, 5, 21, 2
bubble_sort                              Running time:9710.35885810852 ms
select_sort                              Running time:3665.8263206481934 ms
insert_sort                              Running time:4674.12543296814 ms
quick_sort                               Running time:297.0142364501953 ms
heap_sort                                Running time:43.50018501281738 ms
merge_sort                               Running time:37.291526794433594 ms
shell_sort                               Running time:40.53902626037598 ms
count_sort                               Running time:2.0301342010498047 ms
bucket_sort                              Running time:1630.4116249084473 ms
radix_sort                               Running time:6.976842880249023 ms

Process finished with exit code 0
```

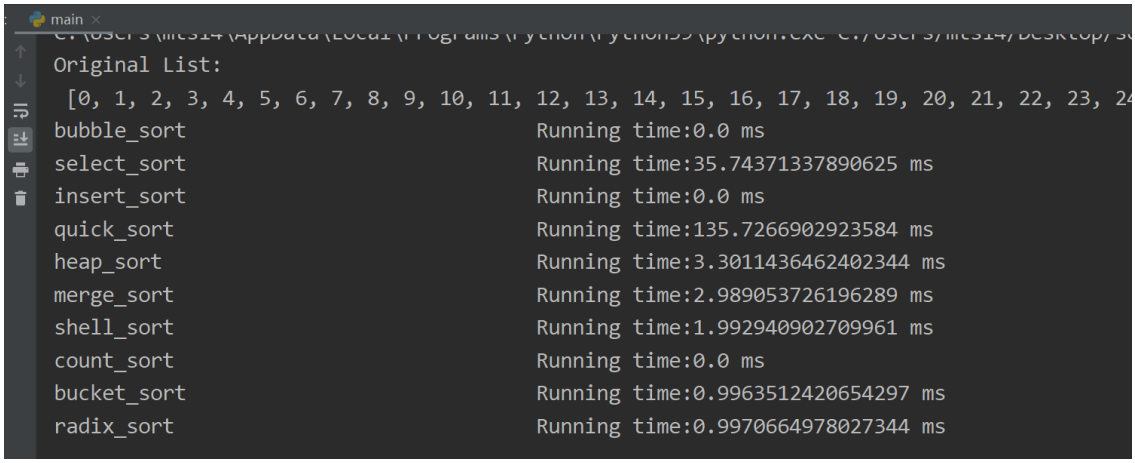
- 进行较大规模0-1000随机排列测试
- 结果:

```
main
Original List:
 [956, 418, 427, 130, 84, 670, 312, 300, 584, 641, 175, 407, 959, 749, 887, 402
bubble_sort                              Running time:10054.833889007568 ms
select_sort                              Running time:3712.5394344329834 ms
insert_sort                              Running time:4772.658109664917 ms
quick_sort                               Running time:49.86119270324707 ms
heap_sort                                Running time:43.46275329589844 ms
merge_sort                               Running time:40.20500183105469 ms
shell_sort                               Running time:54.69918251037598 ms
count_sort                               Running time:2.194643020629883 ms
bucket_sort                              Running time:85.37626266479492 ms
radix_sort                               Running time:15.323400497436523 ms

Process finished with exit code 0
```

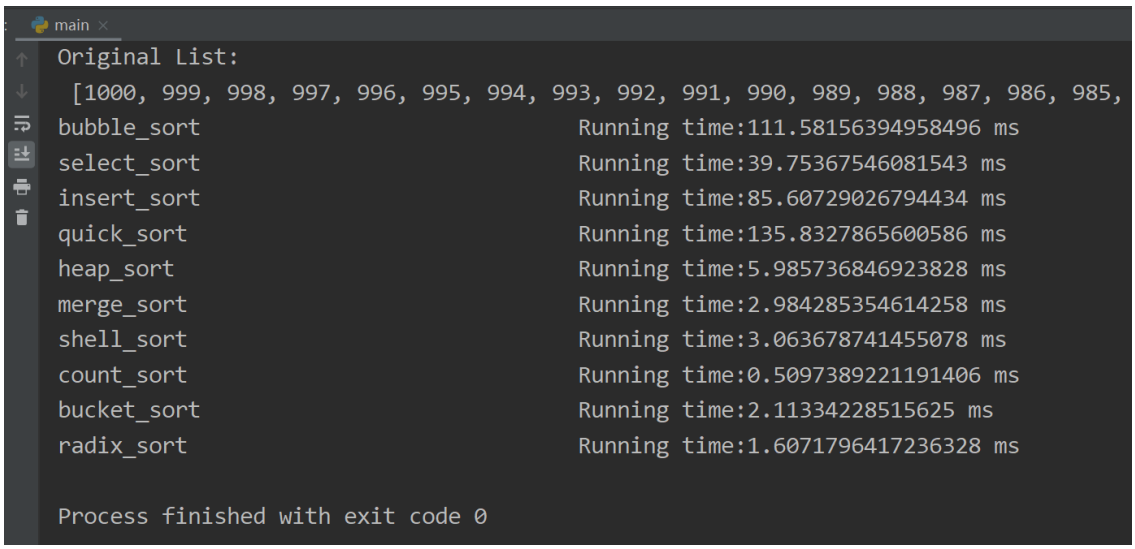
- 1000正序测试

- ```
  #set recursion depth
  sys.setrecursionlimit(10000)
  list_test = [i for i in range(1000)]
  ```

- 

- 1000逆序测试

- ```
  #set recursion depth
  sys.setrecursionlimit(10000)
  list_test = [i for i in range(1000)]
  ```

- 

- 可以看到前三种简单排序的时间远远大于后几种
- 其中后几种中前四种快速排序方法时间复杂度再 $Olog(n)$ 的水平
- 而后边 计数排序、桶排序、基数排序的时间复杂度并不稳定，其中计数排序的时间最少
- 在顺逆序测试中 插入排序和冒泡排序的差异最大，其与数列本身的特征有关，而其他排序的时间差异没有超过同一数量级

- **排序算法特性比较**

排序方法	平均时间复杂度	空间复杂度	稳定性
直接插入排序	$O(n^2)$	$O(1)$	稳定

排序方法	平均时间复杂度	空间复杂度	稳定性
希尔排序	$O(n^{1.5})$	$O(1)$	不稳定
直接选择排序	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlogn)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(1)$	稳定
快速排序	$O(nlogn)$	$O(nlogn)$	不稳定
归并排序	$O(nlogn)$	$O(1)$	稳定
基数排序	$O(d(r+n))$	$O(rd+n)$	稳定

> 基数排序:
>
> - r : 关键字基数
> - d : 长度
> - n : 代表关键字的个数

2.4 最大频率栈

2.4.1 问题描述

- 设计一个类似堆栈的数据结构，将元素推入堆栈，并从堆栈中弹出出现频率最高的元素。
- 实现 FreqStack 类:
  - `FreqStack()` 构造一个空的堆栈。

    `void push(int val)` 将一个整数 `val` 压入栈顶。

    `int pop()` 删除并返回堆栈中出现频率最高的元素。如果出现频率最高的元素不只一个，则移除并返回最接近栈顶的元素。

2.4.2 基本要求

- 输入

  第一行包含一个整数n

  接下来n行每行包含一个字符串（push或pop）表示一个操作，若操作为push，则该行额外包含一个整数val，表示压入堆栈的元素

  对于100%的测试数据，1<=n<=20000，0<=val<=10^9，且当堆栈为空时不会输入pop操作

- 输出

  输出包含若干行，每有一个pop操作对应一行，为弹出堆栈的元素

2.4.3 数据结构设计

- 将每一个元素与其频率绑定在一起入栈，将所有频率相同的元素放在一个栈中，形成频率栈堆，出栈时从最高频率的栈中将元素出栈即可
- 这样在入栈时不用再因寻找频率排序位置而有时间的损耗

2.4.4 功能说明

- push操作

```
void push(int val) {
    if (cnt[val] == stacks.size())  // 这个元素的频率已经是目前最多的，现在又出现了一次
        stacks.push_back({});       // 那么必须创建一个新栈
    stacks[cnt[val]].push(val);
    ++cnt[val];                     // 更新频率
}
```

> 涉及到判断是否需要创建新栈的操作，同时要进行频率的更新

- pop操作

```
 int pop() {
     int val = stacks.back().top();     // 弹出最右侧栈的栈顶
     stacks.back().pop();
     if (stacks.back().empty())         // 栈为空
         stacks.pop_back();             // 删除
     --cnt[val];                        // 更新频率
     return val;
 }
```

> 从最大频率的栈中弹出元素，注意栈空时进行栈堆的更新，同时注意栈频率的更新

2.4.5 调试分析

- 在尝试时首先尝试了一个栈是否能够解决，并仅用栈的结构。
- 在进行入栈时，如果栈顶的频率要大于当前入栈元素的频率则入栈元素下沉直到栈中元素的频率与入栈元素的频率相等为止

```
FreqStack() {
    for (int i = 0; i < 20010; i++)
    {
        this->frequncy[i] = 0;
    }
}

void push(int val) {
    this->frequncy[val]++;
    while (!this->freqStack.empty() && this->freqStack.top().second > this->frequncy[val])
    {
        this->temp.push(this->freqStack.top());
```

```
            this->freqStack.pop();
        }
        this->freqStack.push({val, this->frequncy[val]});
        while (!this->temp.empty())
        {
            this->freqStack.push(this->temp.top());
            temp.pop();
        }
    }

    int pop() {
        int res = this->freqStack.top().first;
        this->frequncy[res]--;
        this->freqStack.pop();
        return res;
    }

    stack<pair<int, int>> temp;
    stack<pair<int, int>> freqStack;
    long long* frequncy = new long long[20010];
```

> 然而这样在进行"位置下沉"的过程中会产生较大的时间损耗，导致`TLE`

- 进而我尝试用`vector`存储栈的结构，利用二分查找的方式寻找插入位置下标

```
int findpos(int l, int r, vector<pair<int, int>> ele, int val)
{
    while (l < r)                           //二分查找优化插入位置
    {
        int mid = (l + r + 1) >> 1;
        if (ele[mid].second > val)
            r = mid - 1;
        else
            l = mid;
    }
    return l;
}

void push(int val) {
    int feq = 1;
    if (mp.empty() || mp.count(val) == 0)
    {
        mp.insert({ val, 1 });
    }
    else
    {
        feq = ++mp[val];
    }
    if (freqStack.empty())
    {
        freqStack.push_back({val, feq});
        return;
    }
    freqStack.insert(freqStack.begin() + findpos(0, freqStack.size() - 1,
freqStack, feq) + 1, {val, feq});
```

```
}
```

> 但是同样的，insert方法同样会消耗很多时间，其中也涉及到了元素的向后移动串位，可能还会涉及到vector底层自动开辟空间的问题，同样是时间消耗巨大，所以没有从根本上解决问题

- 其实以上的想法均是将频率栈堆叠在一起的做法，导致时间复杂度很高，在实际中可以采用分块的方式，将不同频率的元素分在一个区块中，便于查找

2.4.6 总结和体会

- 主要运用的是分块的思想，将频率相同的元素放在一个栈中，最后形成栈堆，用区块定位提高了插入效率

2.5 序列

2.5.1 问题描述

- 给定m个数字序列，每个序列包含n个非负整数。我们从每一个序列中选取一个数字组成一个新的序列，显然一共可以构造出 n^m 个新序列。接下来我们对每一个新的序列中的数字进行求和，一共会得到 n^m个和，请找出最小的n个和

2.5.2 基本要求

- 输入：

- 输入的第一行是一个整数T，表示测试用例的数量，接下来是T个测试用例的输入
- 每个测试用例输入的第一行是两个个正整数m（0 < m <= 100）和n(0 < n <= 2000)，然后有m行，每行有n个数，数字之间用空格分开，表示这m个序列,序列中的数字不会大于10000
- 输出：
- 对每组测试用例，输出一行用空格隔开的数，表示最小的n个和

2.5.3 数据结构设计

- 多路归并排序

- 因为所求的是前n小的和，而这个长度与每一行的数字组数量相等，可以很自然的想到每次就是将两个数组进行合并
- 进行合并时候的数组就是当前已经遍历过的数组的前n小的和
- 合并的过程就是首先将要合并的两个数组的$n \times n$ 的和划分成n组，第i组的元素分别是 $a[0]+b[i], a[1]+b[i], \cdots a[n]+b[i]$,其中a数组代表已经遍历过的数组的前n小的和的数组(有序)， 这样划分每一行都是有序的，因为a[i]都是有序的。
- 这样首先将每一组的第一个元素都拿出来放入小根堆中(小根堆根据)，从小根堆中pop出最小的元素，将pop出来的这个元素所在的那一组的下一个元素加入到小根堆中进行排序，再重复以上过程

2.5.4 功能说明

- 将两个数组合并的函数

```
void merge()
{
```

```
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> heap;
    for (int i = 0; i < n; i++)
    {
        heap.push({a[0] + b[i], 0});
    }

    for (int i = 0; i < n; i++)
    {
        pair<int, int> temp = heap.top();
        heap.pop();
        c[i] = temp.first;
        heap.push({ temp.first - a[temp.second] + a[temp.second + 1],
temp.second + 1 });
    }
    for (int i = 0; i < n; i++)
    {
        a[i] = c[i];
    }
}
```

> 其本质是进行多路归并排序，但是每次选出来最小值用小根堆进行维护进行数组的更新

2.5.5 调试分析

- 进行调试的过程中首先进行了朴素方法，也就是堆 n^2 的和矩阵进行遍历求解，时间复杂度上并不允许通过
- 后来进行了错误的分组，每一组均需要排序之后才可以进行最小值的取出，进行排序的时间复杂度同样是比较高

2.5.6 总结和体会

- 进行排序时进行分组可以更加快速进行
- 进行分组时可以最好进行有序的分组，利用好原有的有序数组进行分组最好

3. 实验总结

- 排序相关及排序的应用
- 基本排序 $O(n^2)$
  - 选择排序
  - 冒泡排序
  - 插入排序
- 优化排序 $O(nlogn)$
  - 归并排序
  - 堆排序
  - 快速排序
- 特殊排序
  - 希尔排序
- 分块进行排序，分块进行数据的取用