# 排序算法

xbZhong

## Contents

本页 PDF

### 选择排序

- 分为待排序区和已排序区
- 顾名思义，选择一个最小值放到已排序区末尾

```
#include<bits/stdc++.h>
using namespace std;
int *getranddata(int n)
{
    int *arr = new int [n];
    for(int i = 0;i < n;i++)
    {
        arr[i] = rand()%1000;
    }
    return arr;
}

bool check(int *arr,int n)
{
    for(int i = 1;i < n;i++)
    {
        if(arr[i] < arr[i - 1])
            return false;
    }
    return true;
}

void print(int *arr ,int n)
{
    for(int i = 0;i < n;i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return;
```

```
}

void selection_sort(int *arr,int n)
{
    for(int i = 0;i < n - 1;i++)
    {
        int ind = i;
        for(int j = i + 1;j < n;j++)
        {
            if(arr[j] < arr[ind]) ind = j;
        }
        swap(arr[i],arr[ind]);
    }
    return;
}
int main()
{
    int n = 100;
    srand((unsigned)time(NULL));
    int *arr = getranddata(n);
    selection_sort(arr,n);
    print(arr,n);
    cout << check(arr,n) << endl;
    delete arr;
    return 0;
}
```

**插入排序**

- 分为待排序区和已排序区
- 每次将待排序区第一个元素插入到已排序区中某个位置去
- 从已排序区最后一个元素开始不断和已排序区元素比较，直到下一个元素比插入元素小

```
#include<bits/stdc++.h>
using namespace std;
int *getranddata(int n)
{
    int *arr = new int [n];
    for(int i = 0;i < n;i++)
    {
        arr[i] = rand()%1000;
    }
    return arr;
}

bool check(int *arr,int n)
{
    for(int i = 1;i < n;i++)
    {
```

```
            if(arr[i] < arr[i - 1])
                return false;
        }
        return true;
    }

    void print(int *arr ,int n)
    {
        for(int i = 0;i < n;i++)
        {
            cout << arr[i] << " ";
        }
        cout << endl;
        return;
    }

    void insert_sort(int *arr,int n)
    {
        for(int i = 1;i < n;i++)
        {
            int j = i;
            while(j > 0 && arr[j - 1] > arr[j])
            {
                swap(arr[j - 1],arr[j]);
                j -= 1;
            }
        }
        return;
    }
    int main()
    {
        int n = 100;
        srand((unsigned)time(NULL));
        int *arr = getranddata(n);
        insert_sort(arr,n);
        print(arr,n);
        cout << check(arr,n) << endl;
        delete arr;
        return 0;
    }
```

**选择排序执行次数是插入排序的两倍**

- 选择排序和插入排序 (计算执行次数的期望) 时间复杂度都为 O($n^2$)
- 底层 cpu 对于数组遍历有加速效果，对于数据交换没有优化
- 选择排序遍历次数多，数据交换少，插入排序数据交换多，因此他们执行时间差不多

**无监督的插入排序**

- 和插入排序区别就是找到全局最小值将其放到第一位，后续 while 判断条件就不用 j>0

- 执行 j>0 这个判断条件次数和交换数据的数量级一致，都是 O($n^2$)
- 增加了 O(n) 的查找最小值次数，减少了 O($n^2$) 的判断条件执行次数，优化了算法

```
void unsupervise_insert_sort(int *arr,int n) //无监督的插入排序
{
    int ind = 0;
    for(int i = 1;i < n;i++)
    {
        if(arr[ind] > arr[i]) ind = i;
    }
    while(ind > 0)
    {
        swap(arr[ind],arr[ind - 1]);
        ind -= 1;
    }
    for(int i = 1;i < n;i++)
    {
        int j = i;
        while( arr[j - 1] > arr[j])
        {
            swap(arr[j - 1],arr[j]);
            j -= 1;
        }
    }
    return;
}
```

**希尔排序**

- 设计一个步长序列
- 按照步长对序列进行分组，每组采用插入排序
- 直到执行到步长为 1 为止
- 希尔排序的效率和步长序列紧密相关 > **O($n^2$)** (最坏时间复杂度) 希尔增量序列：n/2，n/4，n/8，n/16,... > **O($n^{1.5}$)** (最坏时间复杂度) Hibbard 增量序列:1,3,7,...,$2^{k-1}$

**希尔增量**

```
void shell_sort(int *arr,int n)
{
int k = 2,step;
do
{
        step = n / k == 0 ? 1:n / k;
        for(int i = 0;i < step;i++) //执行 step 次是因为每隔 step，元素就是一组，要遍历完不同组
        unsupervise_insert_sort(arr,n,step);
        k *= 2;
}while(step != 1);
return;
}
```

**Hibbrad 增量**

```
void shell_sort_hibbard(int *arr,int n)
{
    int step = 1;
    while(step <= n / 2)  step = step * 2 + 1;
    do
    {
        step /= 2;
        for(int i = 0;i < step;i++)
        unsupervise_insert_sort(arr,n,step);
    }while(step > 1);
    return ;
}
```

**冒泡排序**

- 创建个变量 cnt 去判断第一次扫描数组时是否有序，有序则直接 break

```
#include<bits/stdc++.h>
using namespace std;
void bubble_sort(int *arr,int n)
{
    int cnt;
    for(int i = n;i > 0;i--)
    {
        cnt = 0;
        for(int j = 1;j < i;j++)
        {
            if(arr[j] >= arr[j - 1]) continue;
            swap(arr[j],arr[j - 1]);
            cnt += 1;
        }
        if(cnt == 0) break;
    }
    return ;
}

int *getranddata(int n)
{
    int *arr = new int [n];
    for(int i = 0;i < n;i++)
        arr[i] = rand() % 1000;
    return arr;
}

int  check(int *arr ,int n)
{
    for(int i = 1;i < n;i++)
        if(arr[i] < arr[i - 1]) return 0;
    return 1;
```

```
}

void print(int *arr,int n)
{
    for(int i = 0;i < n;i++)
        cout << arr[i] << " ";
    cout << endl;
    return ;
}
int main()
{
    srand((unsigned)time(NULL));
    int n = 100;
    int *arr = getranddata(n);
    bubble_sort(arr,n);
    print(arr,n);
    cout << check(arr,n) << endl;
    delete arr;
    return 0;
}
```

**快速排序**

- 找到一个基准值，利用其将数组分成两个区域
- 一区域的值比基准值小，另一区域的值比基准值大
  - 以数组第一个元素为基准值
  - 使用两个指针，一个指向头部，一个指向尾部
  - 尾指针移动找到比基准值小的值，将其移动到基准值所在位置
  - 头指针移动找到比基准值大的值，将其移动到尾指针所在位置
  - 头指针尾指针交替移动直到它们重叠，重叠位置就是基准值位置
- 后对两区域再执行上述操作 (递归)

```
#include<bits/stdc++.h>
using namespace std;

void quick_sort(int *arr ,int l,int r) //左闭右开
{
    if(r - l <= 2) //边界条件
    {
        if(r - l <= 1) return;
        if(arr[l] > arr[l + 1]) swap(arr[l],arr[l + 1]);
        return ;
    }
    //快排核心
    int x = l,y = r - 1,z = arr[l];
    while(x < y)
    {
        while(x < y && z <= arr[y]) y--;
        if(x < y) arr[x] = arr[y];
```

```
            while(x < y && z >= arr[x]) x++;
            if(x < y) arr[y] = arr[x];
        }
        arr[x] = z;
        quick_sort(arr,l,x);
        quick_sort(arr,x+1,r);
    }


    int *getranddata(int n)
    {
        int *arr = new int [n];
        for(int i = 0;i < n;i++)
            arr[i] = rand() % 1000;
        return arr;
    }

    int  check(int *arr ,int n)
    {
        for(int i = 1;i < n;i++)
            if(arr[i] < arr[i - 1]) return 0;
        return 1;
    }


    void print(int *arr,int n)
    {
        for(int i = 0;i < n;i++)
            cout << arr[i] << " ";
        cout << endl;
        return ;
    }

    int main()
    {
        int n = 100;
        int *arr = getranddata(n);
        quick_sort(arr,0,n);
        print(arr,n);
        cout << check(arr,n) << endl;
        delete arr;
        return 0;
    }
```

**快速排序优化**

- 用头尾指针找到一对元素，直接交换元素位置 (v1)
- 从基准值入手，找到适合的基准值 (三点取中法)(v2)
- 减少递归次数，对右半部分使用循环进行操作 (v3)

- 对于少量数据的区域采用无监督插入排序 (v4) **原则：后半区间的元素大小大于等于前半区间的元素大小** > 对于 while(x <= y) 需要加上等于号, 否则当 x=y 时，它们指向的可能不是基准值，此时左区间为 (l ~ y+1), 右区间为 (x ~ r) 才可得出正确排序结果。**但是这样效率降低，因为要对 x 和 y 同时指向的元素进行两次排序** > 对于 swap(arr[x++] , arr[y–]) x++ 与 y–是防止 x 和 y 同时指向基准值会导致死循环 > 对于 if(x <= y) 加上等于号是防止 x=y 时，x 与 y 同时指向基准值造成死循环 > 对于 while( z < arr[y]) 不加等于号是为了防止内存溢出，无限次调用系统栈，如数据 1，1，1，2，2，2，5，4，1，1，以 1 为基准值

```
#include<bits/stdc++.h>
using namespace std;
void unsupervise_insert_sort(int *arr,int l,int r)
{
    int ind = l;
    for(int i = l + 1;i < r;i++)
    {
        if(arr[ind] > arr[i]) ind = i;
    }
    while(ind > l)
    {
        swap(arr[ind],arr[ind - 1]);
        ind -= 1;
    }
    for(int i = l + 1;i < r;i++)
    {
        int j = i;
        while(arr[j - 1] > arr[j])
        {
            swap(arr[j - 1],arr[j]);
            j -= 1;
        }
    }
    return;
}

void print(int *arr,int n)
{
    for(int i = 0;i < n;i++)
        cout << arr[i] << " ";
    cout << endl;
    return ;
}
```

- 标准快排

```
void quick_sort(int *arr ,int l,int r) //左闭右开
{
    if(r - l <= 2) //边界条件
    {
        if(r - l <= 1) return;
        if(arr[l] > arr[l + 1]) swap(arr[l],arr[l + 1]);
```

```
        return ;
    }
    //快排核心
    int x = l,y = r - 1,z = arr[l];
    while(x < y)
    {
        while(x < y && z <= arr[y]) y--;
        if(x < y) arr[x] = arr[y];
        while(x < y && z >= arr[x]) x++;
        if(x < y) arr[y] = arr[x];
    }
    arr[y] = z;
    quick_sort(arr,l,x);
    quick_sort(arr,x+1,r);
}
```

- 直接交换左右指针指向的值

```
void quick_sort_v1(int *arr ,int l,int r) //直接交换左右指针指向的值
{
    if(r - l <= 2) //边界条件
    {
        if(r - l <= 1) return;
        if(arr[l] > arr[l + 1]) swap(arr[l],arr[l + 1]);
        return ;
    }
    //快排核心
    int x = l,y = r - 1,z = arr[l];
    while(x <= y)
    {
        while(z < arr[y]) y--;
        while(z > arr[x]) x++;
        if(x <= y)
        swap(arr[x++] , arr[y--]);
    }
    quick_sort_v1(arr,l,x);
    quick_sort_v1(arr,x,r);
}
```

- 优化基准值

```
int way(int a,int b,int c)
{
    if(a > b) swap(a,b);
    if(a > c) swap(a,c);
    if(b > c) swap(b,c);
    return b;
}

void quick_sort_v2(int *arr ,int l,int r) //优化基准值
```

```
{
    if(r - l <= 2) //边界条件
    {
        if(r - l <= 1) return;
        if(arr[l] > arr[l + 1]) swap(arr[l],arr[l + 1]);
        return ;
    }
    //快排核心
    int x = l,y = r - 1;
    int z = way(arr[l],arr[r - 1],arr[(l + r) / 2]);
    while(x < y)
    {
        while( z < arr[y]) y--;
        while( z > arr[x]) x++;
        if(x <= y)
        swap(arr[x++] , arr[y--]);//x++ 与 y--防止 x，y 指针遇到基准值
        造成死循环

    }
    quick_sort_v2(arr,l,y+1);
    quick_sort_v2(arr,x,r);
}
```

- 减少递归次数，采用循环方式

```
void quick_sort_v3(int *arr ,int l,int r) //减少递归次数，对右半区间采用循环方式
{
    while(l < r)
    {
    if(r - l <= 2) //边界条件
    {
        if(r - l <= 1) return;
        if(arr[l] > arr[l + 1]) swap(arr[l],arr[l + 1]);
        return ;
    }
    //快排核心
    int x = l,y = r - 1;
    int z = way(arr[l],arr[r - 1],arr[(l + r) / 2]);
    while(x <= y)
    {
        while(z < arr[y]) y--;
        while(z > arr[x]) x++;
        if(x <= y)//可能会出现头尾指针过度移动的情况
        swap(arr[x++] , arr[y--]);
    }
    quick_sort_v3(arr,l,x);
    l = x;
    }
```

```
        return ;
    }
```

- 对于少量数据采用无监督插入排序

```
    void __quick_sort_v4(int *arr ,int l,int r) //左闭右开
    {
        while(r - l > 16)
        {
        //快排核心
        int x = l,y = r - 1,z = arr[l];
        while(x <= y)
        {
            while(z < arr[y]) y--;
            while(z > arr[x]) x++;
            if(x <= y) //可能会出现头尾指针过度移动的情况
            swap(arr[x++] , arr[y--]);
        }
        __quick_sort_v4(arr,l,x);
        l = x;
        }
        return ;
    }


    void quick_sort_v4(int *arr ,int l,int r) //左闭右开
    {
        __quick_sort_v4(arr,l,r);
        unsupervise_insert_sort(arr,l,r);
    }
```

- 上述代码实现时 x 与 y 会错开，即 x 会在 y 的后面，y 会在 x 的前面

**归并排序**

- 分治：将数组拆分成小数组进行排序
- 归并：将排完序的小数组归并成为一个大的有序数组
- 采用的是递归的思想
- 分成两个小数组，将两个小数组的值按从小到大放进 temp 数组里面，再进行拷贝
- 时间复杂度稳定，为 O($ nlogn $)

```
    void merge_sort(int *arr,int l,int r)
    {
        if(r - l <= 1) return ;
        int mid = (l + r) / 2;
        merge_sort(arr,l,mid);
        merge_sort(arr,mid,r);
        int p1 = l , p2 = mid , k = 0;
        while(p1 < mid || p2 < r)
```

```
        {
            if(p2 == r || (p1 < mid && arr[p1] <= arr[p2]))
                temp[k++] = arr[p1++];
            else
                temp[k++] = arr[p2++];
        }
        for(int i = l;i < r;i++)
            arr[i] = temp[i - l];
        return ;
    }
```

**基数排序**

**相同数字相对位置不变** * 选择数字的某个位置进行排序 (个位，十位，等等) * 求得前缀和数列 * 从后向前扫描数组 * 时间复杂度为 O(n) > 对于一个整型数据，其最大数值为 $2^{32}$。因此我们选取 $2^{16}$ 为基数。这样无论对于任何一个整型数据，我们最多只需要排两轮。轮数为：$\lceil \log_{65536} m \rceil$ * 进行完一轮基数排序后，要将 temp 数据拷贝到 arr 再进行下一轮排序才能得到正确结果 *

```
#include<bits/stdc++.h>
using namespace std;

void print(int *arr,int l,int r)
{
    for(int i = l;i < r;i++)
    {
        if(i) cout << " ";
        cout << arr[i];
    }
    cout << endl;
    return;
}


int *getnewdata(int l ,int r)
{
    srand((unsigned)time(NULL));
    int *arr = new int[r - l];
    for(int i = l;i < r;i++)
    {
        arr[i] = rand() % 100;
    }
    return arr;
}
#define K 10
void radix_sort(int *arr,int l,int r)
{
    int *cnt = new int[K];
    int *temp = new int[r - l];
    memset(cnt,0,sizeof(int) * K);
```

```
    for(int i = l;i < r;i++) cnt[arr[i] % K] += 1;
    for(int i = 1;i < K;i++) cnt[i] += cnt[i - 1]; //求前缀和数组
    for(int i = r - 1;i >= l;i--) temp[--cnt[arr[i] % K]] = arr[i];
    memcpy(arr + l,temp,sizeof(int) * (r - l));
    memset(cnt,0,sizeof(int) * K);
    for(int i = l;i < r;i++) cnt[arr[i] / K] += 1;
    for(int i = 1;i < K;i++) cnt[i] += cnt[i - 1]; //求前缀和数组
    for(int i = r - 1;i >= l;i--) temp[--cnt[arr[i] / K]] = arr[i];
    memcpy(arr + l,temp,sizeof(int) * (r - l));
    delete cnt,temp;
    return ;
}


bool check(int *arr,int l,int r)
{
    for(int i = l + 1;i < r;i++)
    {
        if(arr[i] < arr[i - 1]) return false;
    }
    return true;
}
#define n 10000
int main()
{
    int *arr = getnewdata(0,n);
    radix_sort(arr,0,n);
    //print(arr,0,n);
    cout << check(arr,0,n);
    delete arr;
    return 0;
}
```

**sort 函数用法**

- 函数参数：第 1 个参数是排序区间的头部，第 2 个参数是排序区间的尾部 (**是左闭右开的**)
- 默认升序
- 若要实现从大到小排序，则在加上第 3 个参数，模板函数 greater()
- 对 vector 调用.end() 时，其返回的是最后一个元素的下一个位置 #### 自定义排序方式

1. 实现降序排序

```
bool cmp(int a,int b)
{
    return a > b; //前一个参数大于后一个参数返回true，也就是降序
}
sort(arr,arr + 10;cmp);
```

2. 实现多维度排序

```
bool cmp(struct a,struct b)
```

```
{
    if(a.x != b.x) return a.x > b.x;
    return a.y < b.y;
}
```

- 对结构体中的 x 进行降序排序，对 y 进行升序排序

**两数之和**

# 1. 两数之和

已解答

简单 相关标签 相关企业 提示

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 **和为目标值** *target* 的那 **两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

**示例 1：**

> **输入：** nums = [2,7,11,15], target = 9
> **输出：** [0,1]
> **解释：** 因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。

**示例 2：**

> **输入：** nums = [3,2,4], target = 6
> **输出：** [1,2]

* 题目要求的是返回原数组的元素下标，意味着我们不能对数组进行排序 * 因此我们创建一个下标数组，根据原数组元素大小对下标数组进行排序 * 随后使用双指针，一个指向原数组最小值，一个指向最大值 (最小值下标就是排序后的下标数组的第一位，最大值同理) * 比较两值和与 target 大小，移动 p1 或 p2 * 在最后压入 arr 即可

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> arr,ans;
        for(int i = 0;i < nums.size();i++)
        ans.push_back(i);
        sort(ans.begin(),ans.end(),[&](int i,int j)->bool
        {
            return nums[i] < nums[j];
        });
```

```
            int p1 = 0,p2 = nums.size() - 1;
            while(nums[ans[p1]] + nums[ans[p2]] != target)
            {
                if(nums[ans[p1]] + nums[ans[p2]] > target)
                p2--;
                if(nums[ans[p1]] + nums[ans[p2]] < target)
                p1++;
            }
            arr.push_back(ans[p1]);
            arr.push_back(ans[p2]);
            return arr;
        }
    };
```

**排序链表**

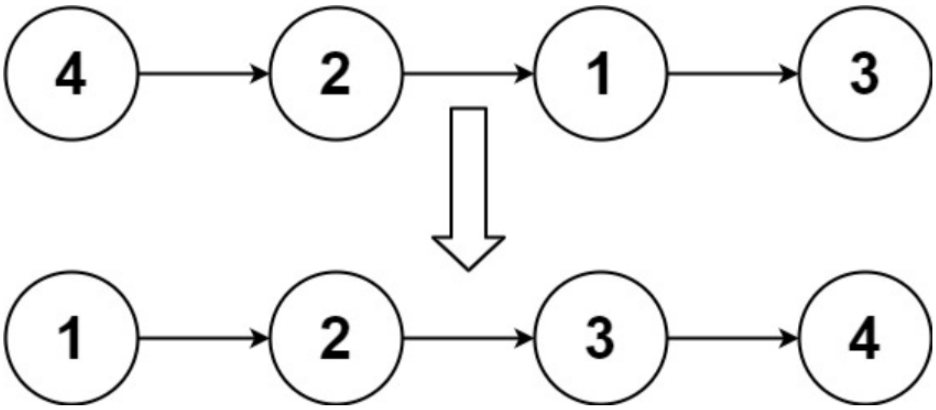
## 148. 排序链表

已解答

中等 相关标签 相关企业

给你链表的头结点 `head` ，请将其按 **升序** 排列并返回 **排序后的链表** 。

**示例 1：**




> **输入：** head = [4,2,1,3]
> **输出：** [1,2,3,4]

* 快速排序 (类似快排，其不是在原链表进行分区，而是将值拷贝到新链表，再进行连接) * z 的取值要注意是右移 (向下取整)，而不是除以 (向 0 取整)

```
    class Solution {
    public:
```

```
    ListNode* sortList(ListNode* head) {
        if(head == NULL || head->next == NULL) return head;
        int l = head->val,r = head->val, z;
        ListNode *p = head,*q,*h1 = NULL,*h2 = NULL;
        while(p) l = min(p->val,l), r = max(p->val,r),p = p->next;
        z = (l + r) >> 1;
        if(l == r) return head;
        p = head;
        while(p)
        {
            q = p->next;
            if(p->val <= z)
            {
                p->next = h1;
                h1 = p;
            }
            else
            {
                p->next = h2;
                h2 = p;
            }
            p = q;
        }
        h1 = sortList(h1);
        h2 = sortList(h2);
        p = h1;
        while(p->next) p = p->next;
        p->next = h2;
        return h1;
    }
};
```

- 归并排序

```
    class Solution {
    public:
        int getlength(ListNode *head)
        {
            int n = 0;
            while(head) n+=1 , head = head->next;
            return n;
        }
        ListNode *merge_sort(ListNode *head , int n)
        {
            if(n <= 1) return head;
            int l = n / 2, r = n - l;
            ListNode *p = head,*p1 = head,*p2,new_head;
            for(int i = 1;i < l;i++) p = p->next;
            p2 = p->next;
```

```
            p->next = NULL;
            p1 = merge_sort(p1,l);
            p2 = merge_sort(p2,r);
            p = &new_head,new_head.next = NULL;
            while(p1 || p2)
            {
                if(p2 == NULL || (p1 && p1->val < p2->val))
                {
                    p->next = p1;
                    p = p1;
                    p1 = p1->next;
                }
                else
                {
                    p->next = p2;
                    p = p2;
                    p2 = p->next;
                }
            }
            return new_head.next;
        }
        ListNode* sortList(ListNode* head) {
            int n = getlength(head);
            return merge_sort(head, n);
        }
    };
```

**合并两个有序数组**

- 使用 sort

```
    class Solution {
    public:
        int t = 65536;
        void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        if(n == 0) return;
        for(int i = 0;i < n;i++) nums1[m + i] = nums2[i];
            sort(nums1.begin(),nums1.end());
            return;
        }
    };
```

- 双指针

```
    class Solution {
    public:
        void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    vector<int> arr;
```

```
        arr.resize(nums1.size());
        int p1 = 0 , p2 = 0,k = 0;
        while(p1 != m || p2 != n )
        {
            if(p2 == n || (p1 != m && nums1[p1] < nums2[p2]))
            {
                arr[k] = nums1[p1];
                p1++,k++;
            }
            else
            {
                arr[k] = nums2[p2];
                p2++,k++;
            }
        }
        for(int i = 0;i <(m + n);i++)
        {
            nums1[i] = arr[i];
        }
        return ;
        }
    };
```

- 反向双指针
- 若想直接在 nums1 进行排序，则需要在 nums1 后部进行插入，在前面插入会使得数据被覆盖

```
    class Solution {
    public:
        void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int p1 = m - 1 , p2 = n - 1,k = m + n - 1;
        while(p1 >= 0 || p2 >= 0 )
        {
            if(p2 == -1 || (p1 != -1 && nums1[p1] > nums2[p2]))
            {
                nums1[k] = nums1[p1];
                p1--,k--;
            }
            else
            {
                nums1[k] = nums2[p2];
                p2--,k--;
            }
        }
            return ;
        }
    };
```

**合并两个有序链表**

- 运用到了归并排序的思想

  class Solution { public: ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) { ListNode *p , new_head; p = &new_head; while(list1 || list2) { if(list2 == NULL || (list1 != NULL && list1->val < list2->val)) { p->next = list1; list1 = list1->next; p = p->next; } else { p->next = list2; list2 = list2->next; p = p->next; } } return new_head.next; } };

**寻找两个正序数组的中位数**

# 4. 寻找两个正序数组的中位数

已解答

困难 相关标签 相关企业

给定两个大小分别为 `m` 和 `n` 的正序（从小到大）数组 `nums1` 和 `nums2`。请你找出并返回这两个正序数组的 **中位数** 。

算法的时间复杂度应该为 `O(log (m+n))` 。

**示例 1：**

> **输入：** nums1 = [1,3], nums2 = [2]
> **输出：** 2.00000
> **解释：** 合并数组 = [1,2,3] , 中位数 2

**示例 2：**

> **输入：** nums1 = [1,2], nums2 = [3,4]
> **输出：** 2.50000
> **解释：** 合并数组 = [1,2,3,4] , 中位数 (2 + 3) / 2 = 2.5

Figure 1: alt text

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int n = nums1.size(),m = nums2.size();
        vector<int> temp(n + m);
```

```
        int p1 = 0,p2 = 0,k = 0;
        while(p1 != n || p2 != m )
        {
            if(p2 == m ||(p1 != n && nums1[p1] < nums2[p2]))
            {
                temp[k] = nums1[p1];
                p1++,k++;
            }
            else
            {
                temp[k] = nums2[p2];
                p2++,k++;
            }
        }
        int t = temp.size() / 2;
        if(temp.size() % 2)
        return temp[t];
        return (temp[t] + temp[t - 1]) / 2.0;
    }
};
```

**存在重复元素**

- 个人

```
        class Solution {
        public:
            bool containsNearbyDuplicate(vector<int>& nums, int k) {
                int p1 = 0, p2 = 1, n = nums.size();
                while (p1 < n) {
                    while (p2 < n && (p2 - p1) <= k) {
                        if (nums[p1] == nums[p2])
                            return true;
                        p2++;
                    }
                    p1++;
                    p2 = p1 + 1;
                }
                return false;
            }
        };
```

* 船长

```
        class Solution {
        public:
            bool containsNearbyDuplicate(vector<int>& nums, int k) {
                int n = nums.size();
                vector<int> ind(n);
```

```
            for(int i = 0; i < n; i++) ind[i] = i;
            sort(ind.begin(),ind.end(),[&](int i ,int j)->bool
            {
                return nums[i] < nums[j];
            });
            for(int i = 0;i < n - 1;i++)
            {
                if(nums[ind[i]] != nums[ind[i + 1]]) continue;
                if(abs(ind[i] - ind[i + 1]) <= k) return true;
            }
            return false;
        }
    };
```

**逆序对个数**

---

# #248. 逆序对个数

描述 | 提交 | 自定义测试 | 题解视频 | 上一题 | 下一题 | 统计

## 题目描述

输入 $N$ 组数据，对每组数据输出逆序对个数。

## 输入

对于每组测试用例，第一行输入此组数据元素个数 $x$，接下来 $x$ 行，每行一个数，表示元素。（$1 \le x \le 500000$）

当读入的元素个数 $x$ 为零时，程序结束。

## 输出

对于每组测试用例，输出一个数，表示逆序对个数。

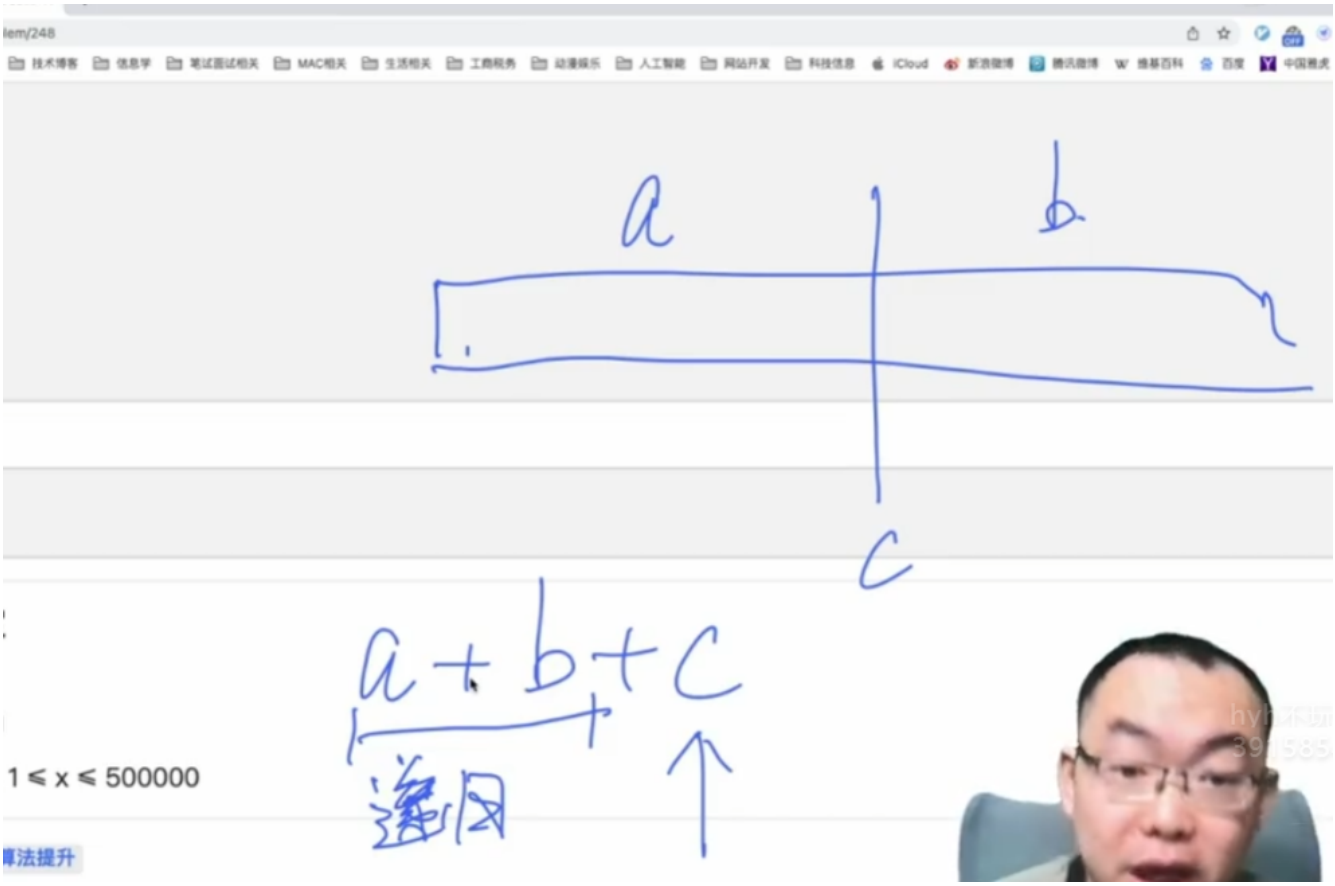
## 样例输入

```
5
9
1
0
5
4
3
1
2
3
0
```

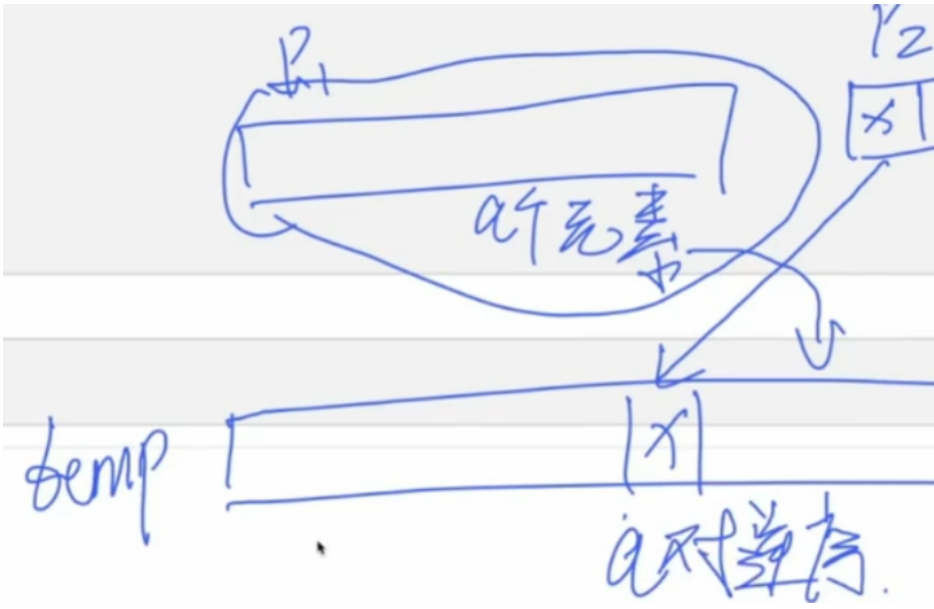
## 样例输出

```
6
0
```

* 可以分成 a(左半部分的逆序对)，b(右半部分的逆序对),c(横跨左右两区间的逆序对) * 对于 a 和 b，可以使用递归来进行求解 * 对于 a，b，其又可以分成三部分来求解逆序对，因此可使用递归来求解




**使用到了归并排序的思想** * 在 p1 和 p2 中选取较小的元素放入 temp，假设 p2 指向的元素被放入，则 p1 后面的 a 个元素与 p2 指向的元素为逆序对 * 要达到上述情况，则要保证左右两个区域的元素都是有序的，因此循环结束后要将




temp 里的元素赋给 arr,以保证区域内元素有序

```
#include<bits/stdc++.h>
using namespace std;
int arr[500005],temp[500005];

long long merge_sort(int *arr,int l,int r)
{
```

```
    if(r - l <= 1) return 0;
    int mid = (r + l) / 2;
    long long a = merge_sort(arr, l, mid);
    long long b = merge_sort(arr, mid, r);
    long long c = 0;
    int p1 = l, p2 = mid, k = 0;
    while(p1 != mid || p2 != r)
    {
        if(p2 == r || (p1 != mid && arr[p1] <= arr[p2]))
            temp[k++] = arr[p1++];
        else
        {
            temp[k++] = arr[p2++];
            c += (mid - p1);
        }
    }
    for(int i = l;i < r;i++) arr[i] = temp[i - l]; //是为了让统计过的区域变得有序
    return a + b + c;
}
void solve(int n)
{
    for(int i = 0;i < n;i++) cin >> arr[i];
    cout << merge_sort(arr,0,n) << endl;
    return;
}

int main()
{
    int n;
    while(1)
    {
        cin >> n;
        if(n == 0) break;
        solve(n);
    }
    return 0;
}
```

**士兵**

# #251. 士兵

描述 | 提交 | 自定义测试 | 题解视频 | 上一题 | 下一题 | 统计

## 题目描述

一些士兵站在矩阵的一些方格内，现要把他们移动到一横排，并连续地排成一队，士兵一次可以选择四个方向中的一个方向移动一格，求最少需要移动多少步才能完成要求。

**即所有士兵的y坐标相同并且x坐标相邻。**

## 输入

第一行输入一个正整数 $n$，表示士兵的数量。（$1 \le n \le 10000$）

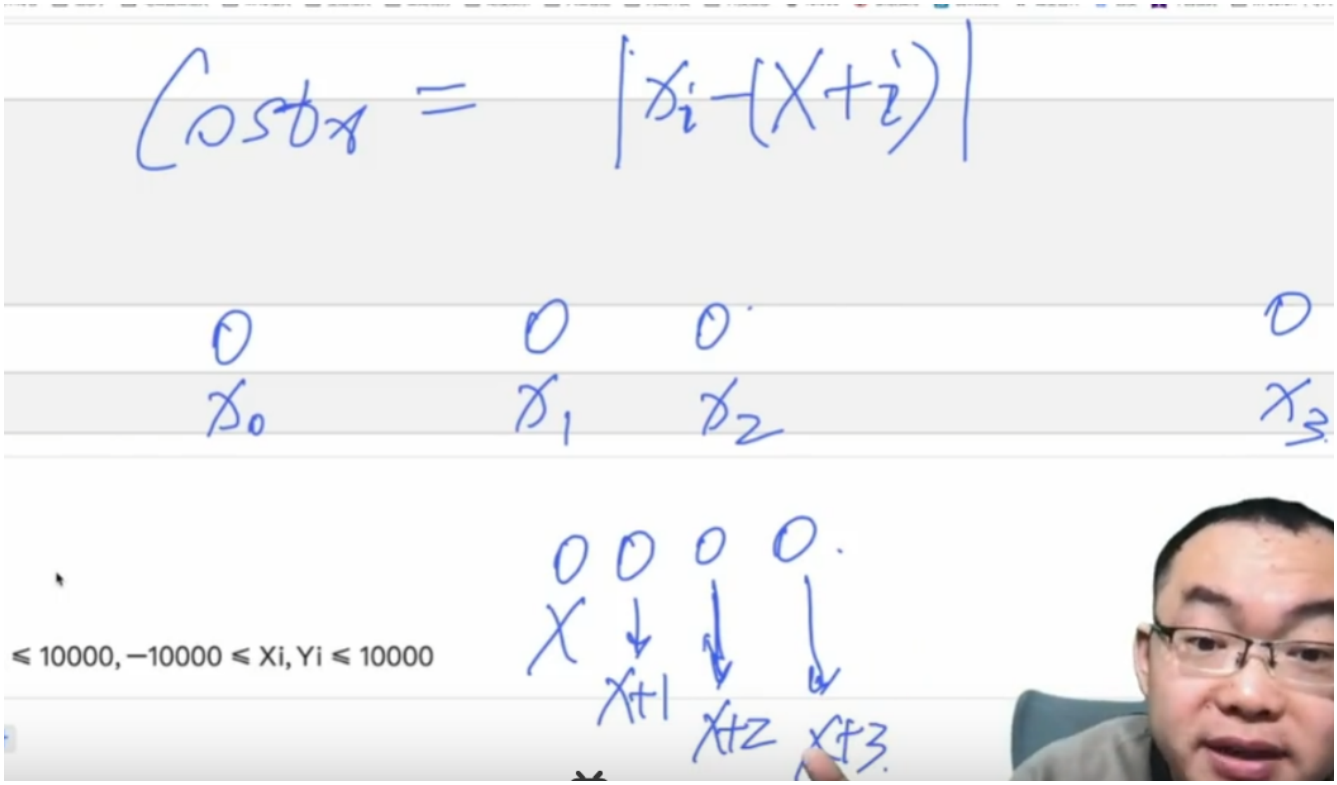
接下来 $n$ 行，每行两个数，代表第 $i$ 个士兵所处位置的横纵坐标 $Xi, Yi$。（$-10000 \le Xi, Yi \le 10000$）

## 输出

输出最少移动步数。

* 最少移动步数为 y 方向上的移动步数加上 x 方向上的移动步数 * y 方向上的移动步数为

$$\sum_{i=1}^{n} |y_i - Y|$$




*

* x 方向上的移动步数为

$$\sum_{i=1}^{n} |(x_i - i) - X|$$

* 要使得 x 方向上移动步数最少，则士兵相对位置是不变的 * 相当于是做两次货仓选址

```
#include<bits/stdc++.h>
using namespace std;
int main()
```

```
    {
        int n;
        cin >> n;
        vector<int> x(n),y(n);
        for(int i = 0;i < n; i++) cin >> x[i] >> y[i];
        int X,Y,costx = 0,costy = 0;
        sort(x.begin(),x.end());
        for(int i = 0; i < n; i++) x[i] = x[i] - i;
        sort(x.begin(),x.end());
        sort(y.begin(),y.end());
        X = x[n / 2];
        Y = y[n / 2];
        for(int i = 0;i < n; i++) costy += abs(y[i] - Y);
        for(int i = 0;i < n; i++) costx += abs(x[i] - X);
        cout << costx + costy << endl;
        return 0;
    }
```

**国王游戏**

时间限制:1 s 空间限制:256 MB

# #256. 国王游戏

描述 提交 自定义测试 题解视频 上一题 下一题 统计

## 题目描述

恰逢 $H$ 国国庆,国王邀请 $n$ 位大臣来玩一个有奖游戏。首先,他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数。然后，让这 $n$ 位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数,然后向下取整得到的结果。

国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能的少。注意，国王的位置始终在队伍的最前面。

## 输入

第一行包含一个整数 $n$，表示大臣的人数。

第二行包含两个整数 $a$ 和 $b$，之间用一个空格隔开,分别表示国王左手和右手上的整数。（均小于 10000）

接下来 $n$ 行，每行包含两个整数 $a$ 和 $b$，之间用一个空格隔开，分别表示每个大臣左手和右手上的整数。（均小于 10000）

## 输出

输出一个整数，表示重新排列后的队伍中获奖赏最多的大臣所获得的金币数。

* 对于序列 1，我们对其进行微扰 * 即交换 $C_i$ 与 $ C_{i + 1}$ 的位置得到序列 2, 其他位置的值不会变 * 为了满足题目所给的要求，我们希望得到的序列 2 的最大值小于序列 1 的最大值 > 假设 $ C_{i + 1} 1 C_{i+1}^{ ‘} C_{i+1} 2 1 C_{i+1} C_{i}^{’ } A_{i} * B_i ⩾A_{i+1} * V_{i+1} A_{i} * B_i ⩽A_{i+1} * V_{i+1} > C_i$ 是序列 1 最大值，交换后序列 2 还有比其更大

# HZOJ-256：国王游戏

对序列，施加『微扰』，调换 i 与 i+1 位置的大臣，观察序列前后的变化：

C1 C2 … $C_i$ $C_{i+1}$ … $C_n$

C1 C2 … $C_{i+1}$` $C_i$` … $C_n$

$$C_i = \prod_{j=0}^{i-1} A_j \Big/ B_i \qquad C_i` = \prod_{j=0}^{i-1} A_j \times A_{i+1} \Big/ B_i$$

$$C_{i+1} = \prod_{j=0}^{i-1} A_j \times A_i \Big/ B_{i+1} \qquad C_{i+1}` = \prod_{j=0}^{i-1} A_j \Big/ B_{i+1}$$

的