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,$\ldots$
  > **O($n^{1.5}$)** (最坏时间复杂度) Hibbard增量序列:1,3,7,$\ldots$,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
        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
{
    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<待排序元素的类型>()
- 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

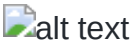
两数之和

 alt text

- 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



- 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
        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
    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
        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

 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

 alt text

- 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  alt text **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  alt text

```
#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
    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;
    }
```

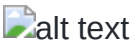
士兵

 alt text

- 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$$ \sum_{i = 1}^{n} {\vert y_i - Y \vert} $$
-  alt text
- x方向上的移动步数为$$ \sum_{i = 1}^{n} {\vert (x_i- i)- X \vert} $$
  - 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



- 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}$是序列1的最大值，则交换后可知$ C_{i + 1}^{'}$小于$ C_{i + 1}$，为了让序列2的最大值比序列1的最大值小，我们要让$ C_{i + 1}$大于$ C_{i}^{'}$，可以解得$A_{i} * B_i \geq A_{i+1} * V_{i+1} $，上述得到的式子是需要进行交换才能得到更优解的，因此我们可以得到$ A_{i} * B_i \leq A_{i+1} * V_{i+1} $的排序顺序 而若$C_i$是序列1最大值，交换后序列2还有比其更大的