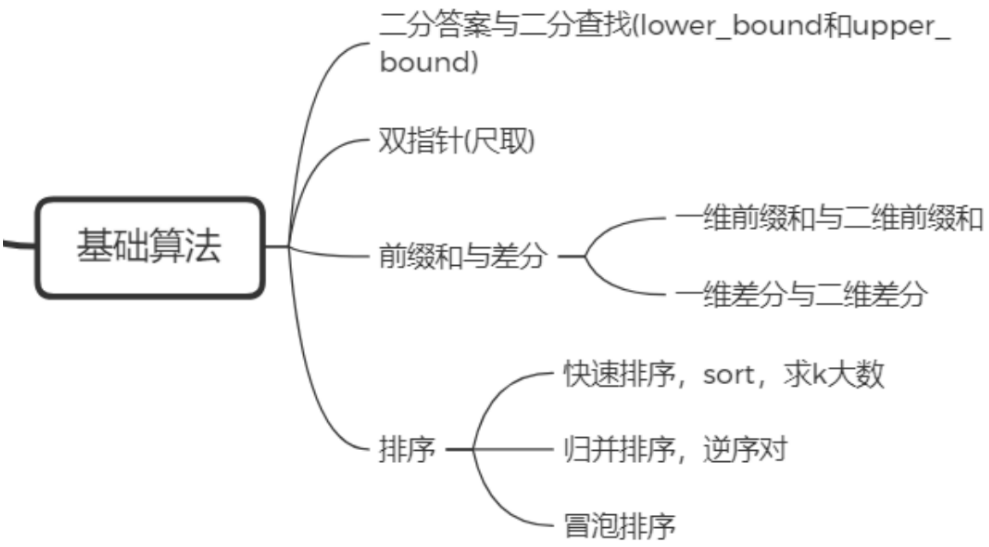
蓝桥杯十天冲刺省一


基础算法
二分答案与二分查找(lower_bound和upper_bound)
双指针(尺取)
前缀和与差分
一维前缀和与二维前缀和
一维差分与二维差分
排序
快速排序，sort，求k大数
归并排序，逆序对
冒泡排序

Day-3 基础算法

二分

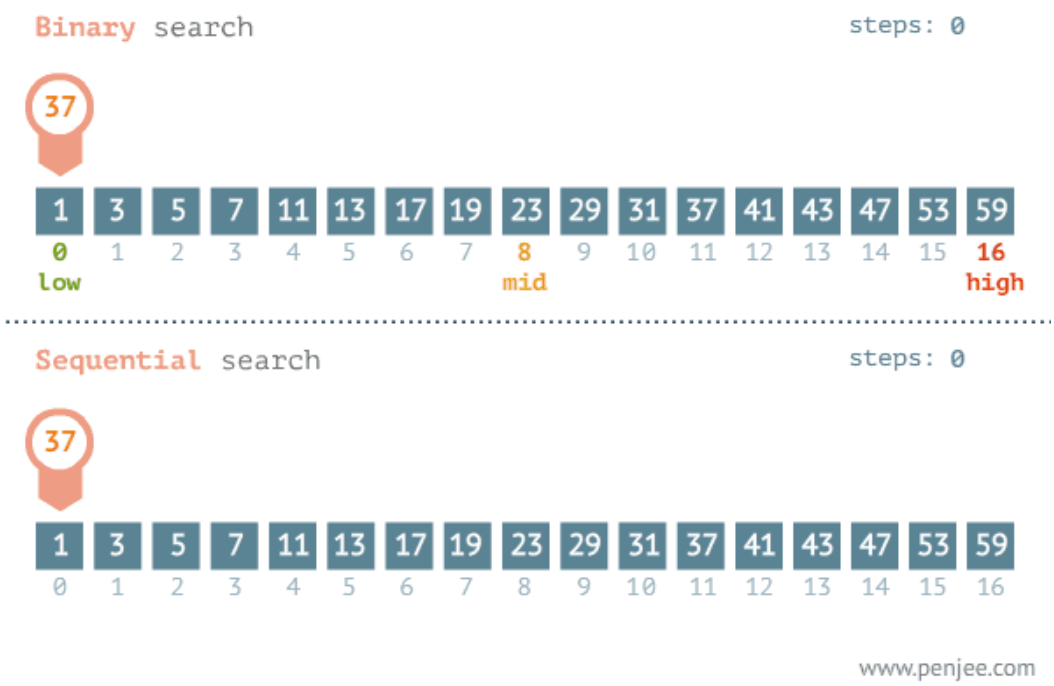
二分查找

二分查找算法思想：对于 n 个有序且没有重复的元素（假设为升序），从中查找特定的某个元素 x，我们可以将有序序列分为规模大致相等的两部分，然后取中间元素与要查找的元素 x 进行比较，如果 x 等于中间元素，则查找成功，算法终止；如果 x 小于中间元素，则在序列的前半部分继续查找，否则在序列的后半部分继续查找。这样就可以将查找的范围缩小一半，然后在剩余的一半中继续重复上面的方法进行查找。

这种每次都从中间元素开始比较，并且一次比较后就能把查找范围缩小一半的方法叫做二分查找。二分查找的时间复杂度是 $O(logn)$，是一种效率较高的查找算法。

一个比较形象的比喻就是大家翻书要翻到想看那一页，大家肯定都是先大概翻到中间

- 如果页码大了就直接往前翻，往后的页码都没必要翻了
- 如果页码小了就直接往后翻，往前的页码都没必要翻了



数的范围

```cpp
#include<bits/stdc++.h>  // 引入常用的头文件

using namespace std;

const int N = 1e5 + 10;  // 定义常量N，表示数组长度的上限

int a[N];  // 定义数组a，用于存储输入的数据
int n, q;  // 定义变量n表示数组长度，变量q表示查询次数

int main() {
    cin.tie(0);  // 解除cin与cout的绑定，提高输入输出效率
    cout.tie(0);

    cin >> n >> q;  // 输入数组长度和查询次数
    for (int i = 1; i <= n; i++) {
        cin >> a[i];  // 输入数组元素
    }

    while (q--) {
        int x;
        cin >> x;  // 输入查询的值x
```

```
        int l = 1, r = n;  // 初始化二分查找的左右边界
        int ans;  // 存储查询结果

        // 二分查找寻找第一个大于等于x的位置
        while (l <= r) {
            int mid = (l + r) / 2;
            if (a[mid] >= x) {
                ans = mid;
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }

        // 如果找不到等于x的元素，输出"-1 -1"并继续下一次查询
        if (a[ans] != x) {
            cout << "-1 -1" << endl;
            continue;
        }

        cout << ans - 1 << ' ';  // 输出第一个等于x的位置的左边界

        l = 1, r = n;  // 重新初始化左右边界
        ans = 1;  // 重新初始化查询结果

        // 二分查找寻找最后一个小于等于x的位置
        while (l <= r) {
            int mid = (l + r) / 2;
            if (a[mid] <= x) {
                ans = mid;
                l = mid + 1;
            } else {
                r = mid - 1;
            }
        }

        cout << ans - 1 << endl;  // 输出最后一个等于x的位置的右边界
    }
```

```
    return 0;
}
```

二分答案

二分思想不仅可以在有序序列中快速查找元素，还能高效地解决一些具有单调性判定的问题。

二分答案算法思想：某些问题不容易直接求解，但却很容易判断某个解是否可行，如果问题的答案具有单调性（即如果答案 x 不可行，那么大于 x 的解都不可行，而小于 x 的解都能可行），就像我们一开始玩的猜数字游戏一样，我们可以根据已知条件设定答案的上下界，然后用二分的方法枚举答案，再判断答案是否可行，根据判断的结果逐步缩小答案范围，直到符合题目条件的答案为止。

假设验证答案的时间复杂度为 $O(k)$，那么解决整个问题的时间复杂度为 $O(klogn)$。

推荐二分模板

推荐使用下面的二分模板，不用考虑烦人的左指针或者右指针的偏移问题。

```
bool check(int x)
{
    // 进行某些操作
}

// 二分查找函数
int binarySearch()
{
    int l = 1, r = n; // 初始化左右边界
    while (r - l > 1) // 当右边界与左边界相差大于1时
    {
        int mid = (l + r) >> 1; // 取中间位置
        if (check(mid)) // 如果满足条件
            r = mid; // 更新右边界为mid
        else
            l = mid; // 否则更新左边界为mid
    }
```

```
    if (check(l)) // 如果满足条件
        return l; // 返回左边界值
    else if (check(r)) // 如果满足条件
        return r; // 返回右边界值
    return -1; // 否则返回-1
}
```

> **Note**
>
> 注意上面的代码仅仅是更新右边界的，同样也有需要更新左边界的。后面的 check 函数还需要根据自身的情况来填写

砍树

- 样例分析：有 5 棵树，需要收集到 20 个单位的木材，每棵树的高度分别为 4、42、40、26、46，需要将锯子的高度调整为 36，这样可以分别锯下 0、6、4、0、10 高度的木材，刚好满足 20 个单位的木材。如果锯子再高，就不能满足要求了。
- 如果锯子非常低，可以收集的木材会相当多，以至于超过需要的数量。随着砍伐高度逐渐增大，获得的木材会逐渐减少。砍树高度增加到某个值之后，收集的木材就会不够用。因此需要找到最大的x，使得刚好满足要求：哪怕增加 1 米都无法满足需求。这时 x 就是答案。
- 可以从 1 开始往上枚举，每次枚举高度都需要计算收集到的木材数量，如果 x 满足要求且 $x+1$ 不满足要求，答案就是 x。这种方法虽然正确，但是时间复杂度 $O(n \times h)$，效率很低，因此需要考虑更好的方法。
- 先来调整一下题目：令条件=“当砍树的高度是x时可以获取到不少于 m 的木材”，那么就是要找最大的 x 使得条件成立。再来看一下是否满足单调性：当 x 超过某个数时，条件一定不成立，而不超过这个数时，条件一定成立。这里的答案完全符合单调性，所以可以使用二分答案来解决问题。
- 如果之前的二分查找的边界条件和 $+1$ 或 -1 不太明白，那么下面的写法可以大大降低理解的难度：只需要**想清楚答案是否需要更新**和**可能的答案在哪一侧**即可。

```
#include<bits/stdc++.h>  // 引入常用的头文件
using namespace std;

int a[1000010];  // 定义数组a，用于存储输入的数据
```

```cpp
int main() {
    cin.tie(0);  // 解除cin与cout的绑定，提高输入输出效率
    cout.tie(0);

    long long n, m, maxn = -1, u = 0, x = 0;  // 定义变量n、m，maxn用于存储最大值，u用于计算总和，x用于存储结果

    cin >> n >> m;  // 输入数据组数n和目标值m

    for(int i = 1; i <= n; i++) {
        cin >> a[i];  // 输入数组元素
        if(maxn < a[i]) {
            maxn = a[i];  // 更新最大值
        }
    }

    long long l = 0, r = maxn;  // 初始化二分查找的左右边界

    while(l <= r) {
        u = 0;  // 每次循环重新初始化变量u
        int mid = (l + r) / 2;  // 计算中间值

        // 计算大于中间值的差值之和
        for(int i = 1; i <= n; i++) {
            if(a[i] > mid) {
                u += a[i] - mid;
            }
        }

        // 判断是否满足条件，更新结果和边界
        if(u >= m) {
            x = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
```

```
    cout << x;  // 输出结果
    return 0;
}
```

分巧克力（蓝桥杯C/C++2017B组省赛）

```
#include<bits/stdc++.h>  // 引入常用的头文件
using namespace std;

int n, k;  // 定义变量n表示数据个数，k表示目标值
int a[100005], b[100005];  // 定义数组a和b，用于存储输入的数据
int l, r = 1e9, mid;  // 初始化二分查找的左右边界和中间值

// 定义函数f，用于判断是否满足条件
bool f(int mid) {
    int c = 0;
    for(int i = 1; i <= n; i++)
        c += (a[i] / mid) * (b[i] / mid);  // 计算并累加满足条件的值
    return c >= k;  // 返回判断结果
}

int main() {
    int res;
    cin >> n >> k;  // 输入数据组数n和目标值k

    // 输入数组a和b
    for(int i = 1; i <= n; i++)
        cin >> a[i] >> b[i];

    // 二分查找
    while(l <= r) {  // 忽略边界
        mid = (l + r) / 2;  // 计算中间值
        if(f(mid)) {// 判断是否满足条件
            res = mid;
            l = mid + 1;  // 更新左边界
        }
        else
            r = mid - 1;  // 更新右边界
```

```
    }

    cout << res;  // 输出结果
    return 0;
}
```

lower_bound()和upper_bound()

lower_bound():二分查找左边界

```
lower_bound(a+开始,a+结束+1,x,cmp);
```

函数含义：在a数组的下标区间内，按照cmp的排序规则，找元素x的左边界(第一个 >=元素x的位置)，返回位置指针；(指针(Pointer): 变量的地址，通过它能找到以它为地址的内存单元。)

注意点：
(1)基本注意点同binary_search；
(2)此处返回的不是下标的值，而是返回指针；如果找不到符合条件的元素位置，指向下标为第一个大于等于元素的位置

例子：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[6]={20,10,50,20,20,40};
    sort(a,a+5+1);//10 20 20 20 40 50
    int *p=lower_bound(a+0,a+5+1,20);
//  cout << p << " "<< *p << endl;
//  cout << p-a<< endl;
    cout << lower_bound(a,a+6,20)-a << endl;
    cout << lower_bound(a,a+6,15)-a << endl;
    cout << lower_bound(a,a+6,60)-a << endl;
    return 0;
}
```

upper_bound()：二分查找右边界

```
upper_bound(a+开始, a+结束+1,x,cmp);
```

函数含义：在a数组的下标内，按照cmp的排序规则，找元素x的 右边界(第一个 > 元素x的位置)，返回位置指针；
注意点：同 lower_bound；

例子：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[6]={20,10,50,20,20,40};
    sort(a,a+5+1);
    int *p=upper_bound(a,a+6,20);
    cout << p << " "<< *p << endl;
    cout << p-a<< endl;
    cout << upper_bound(a,a+6,20)-a << endl;
    cout << upper_bound(a,a+6,15)-a << endl;
    cout << upper_bound(a,a+6,60)-a << endl;
    return 0;
}
```

双指针(尺取)

简介

- 尺取法，通俗来讲，就是双指针法。
- 尺取法通常是对数组保存一对下标，即**所选区的区间的左右端**，然后根据实际情况不断推进 **区间左右端点**以得到答案
- 尺取法比暴力枚举要高效得多，一般情况下暴力枚举需要 $O(n^2)$ 的复杂度，而尺取法就用 $O(n)$ 的复杂度

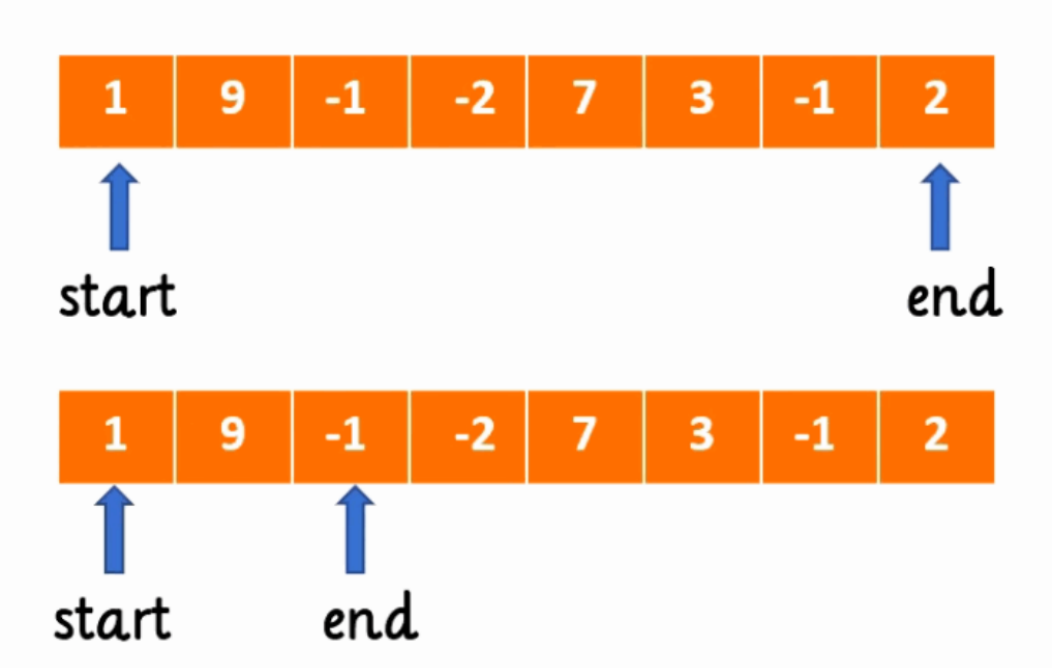
选用尺取法的情况

- 通常适用于所选的区间具有一定的规律性
- 其次，我们能知道对当前区间进行判断之后，指针如何进行改变
- 尺取法，在一般情况下需要对某些量进行预处理，以便更快速得到答案

尺取法的两种选择

- 左右指针上:两个指针反向扫描，一个在最左端，一个在最右端
- 快慢指针 :两个指针同向扫描，但是指针的移动快慢速度不同

如图所示:第一行是左右指针，第二行是快慢指针



尺取法的步骤

- 对于给定的一个序列，对其进行排序(根据题目再定)
- 预处理，比如:有的时候要进行前缀和操作3.设立双指针i和;再根据题意选定合适的尺取法
- 实例：尺取法实现数组去重

```
sort(num+1,num+n+1);//必须是有序的
for(int i=1,j=0;i<=n;i++)//i是快指针，j是慢指针
{
    if(num[i]!=num[i+1])
    {
        num[++j]=num[i];
    }
}
```

最长连续不重复子区间

```
#include<iostream>

using namespace std;

const int N = 100010;
int a[N], st[N]; // st:标记函数，可以使用哈希表代替
int n;

int main(){
    cin >> n;
    for(int i = 0; i < n; ++i){
        cin >> a[i];
    }

    int res = 1;
    for(int  i = 0, j = 0; i <n; ++i){
        st[a[i]] ++;
        while(j <= i && st[a[i]] > 1){ // 将慢指针往前移动
            st[a[j]] --;
            j ++;
        }
        res = max(res, i - j + 1);
    }
    cout << res << endl;
    return 0;
}
```

> **Note**
>
> 本题也可以使用上面的二分答案求解，留给同学们自行思考如何求解

子串简写（蓝桥杯C/C++2023B组省赛）

```
#include<bits/stdc++.h>  // 引入常用的头文件
using namespace std;

typedef long long ll;  // 定义别名 ll 表示 long long

int k, cnt;  // 定义变量 k 表示匹配长度，cnt 表示计数器
string s;  // 定义字符串 s，用于存储输入的数据
char a, b;  // 定义字符 a 和 b，用于存储输入的字符

int main() {
    cin >> k >> s >> a >> b;  // 输入数据 k、s、a、b
    int n = s.size();  // 获取字符串 s 的长度

    ll ans = 0;  // 初始化结果变量 ans

    for(int i = n - k, j = n - 1; i >= 0; i--, j--) {
        if(s[j] == b) cnt++;  // 如果当前字符为 b，则计数器加一
        if(s[i] == a) ans += cnt;  // 如果当前字符为 a，则将计数器的值累加到结果中
    }

    cout << ans;  // 输出结果
    return 0;
}
```

前缀和与差分

一维前缀和

1、什么是前缀和

对于一维数组来说，前缀和是这个数组的某个下标之前的（包括当前元素）所有元素之和。声明一个数组 a[],a的前缀和数组声明为sum[]。根据前缀和数组定义可知每一项的值：

$sum[1] = a[1]$

$sum[2] = a[1] + a[2]$

$sum[3] = a[1] + a[2] + a[3]$

$...$

$sum[i-1] = a[1] + a[2] + a[3] + ... + a[i-1]$

$sum[i] = a[i] + a[2] + a[3] + ... + a[i-1] + a[i]$

$sum[i+1] = a[i] + a[2] + a[3] + ... + a[i] + a[i+1]$

$...$

$sum[n] = a[1] + a[2] + a[3] + ... + a[n]$

通过上述递推式我们可以得到：

$sum[i-1] = a[1] + a[2] + a[3] + ... + a[i-1]$

$sum[i] = a[1] + a[2] + a[3] + ... + a[i-1] + a[i]$

那么前缀和的递推式就可以写成：

$sum[i] = sum[i-1] + a[i]$

2、前缀和的作用

利用前缀和，可以快速求得区间和。

例如：

数组a:存储元素

元素	1	3	6	5	4	2
下标	1	2	3	4	5	6

数组 sum：$sum[i] = sum[i-1] + a[i]$

元素	1	4	10	15	19	21
下标	1	2	3	4	5	6

有如下问题：

对于数组 a,从第 L 个数加到第 R 个数的和是多少？累加区间为 $[L,R]$。

如果我们用 `for` 循环来写的话，这个时间复杂度是 $O(n)$ 的

```
int s = 0;
for(int i = L;i <= R;i++) s = s + a[i];
```

但如果我们用前缀和的话，就可以把时间复杂度优化到 $O(1)$

```
int s = 0;

s = sum[R] - sum[L-1];
```

例如当 $L=2$，$R=6$ 时，$s = sum[6] - sum[1] = 20$。

一维前缀和

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

const int N = 1e5 + 10;  // 定义常量N，表示数组长度的上限
int sum[N];  // 定义数组sum，用于存储前缀和

int main() {
    int n, k, x;
    cin >> n;  // 输入数组长度

    // 计算前缀和
    for(int i = 1; i <= n; i++) {
```

```
        cin >> x;  // 输入数组元素
        sum[i] = sum[i - 1] + x;   // 计算前缀和并存储到数组sum中
    }

    cin >> k;  // 输入查询次数k
    while(k--) {
        int l, r;
        cin >> l >> r;  // 输入查询区间[l,r]

        // 输出区间和，利用前缀和数组sum进行快速计算
        cout << sum[r] - sum[l - 1] << endl;
    }
    return 0;
}
```

k倍区间（蓝桥杯C/C++2017B组省赛）

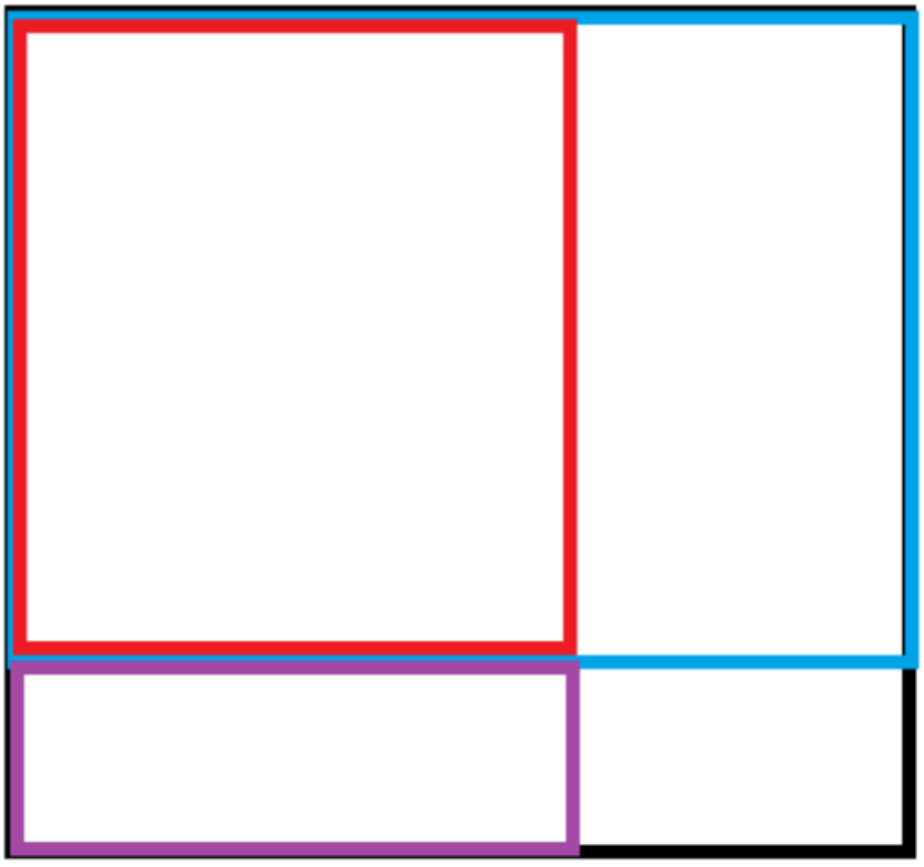
```
#include<bits/stdc++.h>
using namespace std;

int n;  // 表示n个数据
int k;  // 表示k倍区间
int a[100010];  // 存放n个数据
long long sum[100010];  // 存放前缀和的数组
long long cnt[100010];  // 存放余数的数组
long long res;  // 存放结果

int main() {
    // 输入数据
    cin >> n >> k;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        sum[i] = a[i] + sum[i - 1];  // 计算前缀和
    }

    // 循环枚举余数相同
    cnt[0] = 1;
    for (int i = 1; i <= n; i++) {
        res += cnt[sum[i] % k];
```

```
            cnt[sum[i] % k]++;
        }

        // 输出数据
        cout << res << endl;

        return 0;
    }
```

二维前缀和

构造数组

和一维前缀和一样，我们需要构造一个二维前缀和数组 $s[i][j]$，利用前缀和思想:我们需要通过前面已知的前缀和来推出后面未知的前缀和



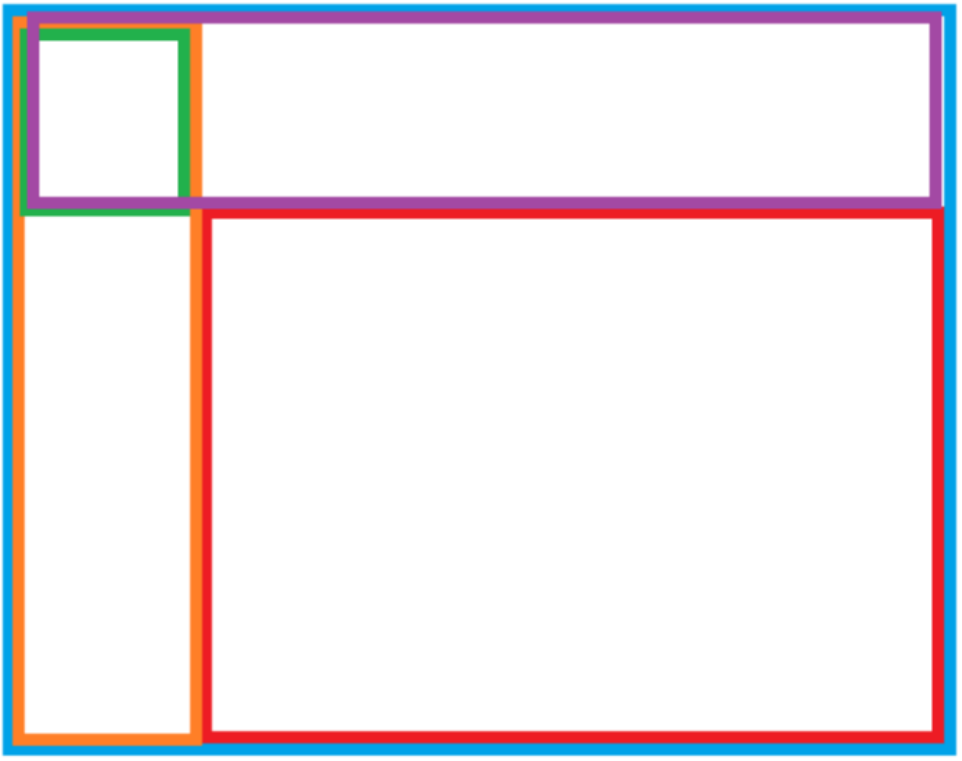
如图，一个二维矩阵(黑色部分)面积 = 蓝色部分面积 + 紫色部分面积 + 右下角小方块面积 - 红色部分面积(多加了一次)由此可以得出:

$$s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + a[i][j]$$

查询

我们想得到的是以 (x_1, y_1) 为左上角，以 (x_2, y_2) 为右下角的子矩阵的面积(即下图中红色矩型的面积)，我们知道的有绿色矩形、蓝色矩形、橙色矩形、紫色矩形的面积。

不难得出红色矩形面积 = 蓝色矩形面积一紫色矩形面积一橙色矩形面积+绿色矩形面积(多减了一次)



即红色矩形面积 = $s[x_1][y_1] - s[x_1 - 1][y_2] - s[x_2][y_1 - 1] + s[x_1 - 1][y_1 - 1]$

```
#include<iostream>
#include<vector>
using namespace std;
const int N = 1010;
int s[N][N];
int main(){
    int n,m,q; //n行m列,q次查询
    cin>>n>>m>>q;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
        {
```

```
            cin>>s[i][j];   // 此时 s[i][j] = a[i][j];  (a[i][j]就是原数组值，被省略掉了)
            s[i][j] += s[i-1][j]+s[i][j-1]-s[i-1][j-1]; // 构建前缀和矩阵
        }
    // 询问
    while(q--){
        int x1,y1,x2,y2;
        cin>>x1>>y1>>x2>>y2;
        // 因为多减了一次 s[x1-1][y1-1]，所以要加回去
        cout<<(s[x2][y2]-s[x1-1][y2]-s[x2][y1-1]+s[x1-1][y1-1])<<endl;
    }

    return 0;
}
```

统计子矩阵（蓝桥杯C/C++2022B组省赛）

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN=501;

int n,m,k,p[MAXN][MAXN];
long long ans;
inline void scan(){
    cin>>n>>m>>k;
    for(int i=1;i<=n;i++){//二维前缀和
        for(int j=1;j<=m;j++){
            cin>>p[i][j];
            p[i][j]+=p[i-1][j]+p[i][j-1]-p[i-1][j-1];
        }
    }
}
inline int getSum(int i,int j,int u,int v){
    return p[u][v]-p[u][j-1]-p[i-1][v]+p[i-1][j-1];
}
int main(){
    cin.tie(0),cout.tie(0);
```

```
    ios::sync_with_stdio(false);
    scan();
    //尺取法,限定上下边界,蠕动左右边界,减少判断次数
    for(int i=1;i<=n;i++){//上边界
        for(int j=i;j<=n;j++){//下边界
            for(int col_l=1,col_r=1;col_r<=m;col_r++){//左右蠕动
边界
                while(col_l<=col_r&&
getSum(i,col_l,j,col_r)>k)col_l++;
                if(col_l<=col_r)ans+=col_r-col_l+1;//如果区间合
法,那么说明此矩阵和必定小于等于k,所以子矩阵也合法
            }
        }
    }
    cout<<ans;
    return 0;
}
```

一维差分

差分的概念

差分就是记录某个数组相邻两项的差值。

声明原数组 a ,差分数组 d ，根据差分定义可知:$d[i]=a[i]-a[i-1]$

求数组 a 的差分数组 d:

```
for(int i=1;i<=n;i++)

 d[i]=a[i]-a[i-1];
```

从上述概念及代码不难看出差分数组主要是用来快速修改原数组区间值的。

例题分析：现在对数组 a（图标如下）的下标为 $[2,5]$ 的区间都加上元素 3 。

元素	1	3	6	5	7	10
下标	1	2	3	4	5	6

同时，我们也求出差分数组 d（如下图）

元素	1	2	3	-1	2	3
下标	1	2	3	4	5	6

当我们使用单循环的方式操作数组，那么数组 a 会发生如下变化：

元素	1	6	9	8	10	10
下标	1	2	3	4	5	6

我们还有另外一种解题思路，那就是对端点值进行赋值实现，即 $d1[2]=d[2]+3, d1[6]=d[6]-3$,其余位置元素不变。对区间 $[L,R]$ 分别加上 x，差分数组只有两个端点值发生变化：$d[L]=d[L]+x, d[R+1]=d[R+1]-x$。修改后的数组如下：

元素	1	5	3	-1	2	0
下标	1	2	3	4	5	6

大家看到这里可能会说了，我们这里修改了差分数组似乎没有看到任何变化啊，不要着急我们接着往下看。我们可以使用前缀和递推式：$sumd[i]=sumd[i-1]+d[i]$；求出 sum 数组的每一项（如下图）。

元素	1	6	9	8	10	10
下标	1	2	3	4	5	6

差分模板代码

```
#include<bits/stdc++.h>
using namespace std;
int  n,m,a[100005],d[100005],sumd[100005];
int main()
{
    cin>>n>>m;
    int l,r,c;
    for(int i=1;i<=n;i++)//存数据
    {
        cin>>a[i];
        d[i]=a[i]-a[i-1];//记录差分数组
    }
    for(int i=1;i<=m;i++)//m次区间操作
```

```
    {
        cin>>l>>r>>c;
        d[l]+=c;
        d[r+1]-=c;
    }
    for(int i=1;i<=n;i++)//求最终的前缀和，即修改收后的a
    {
        sumd[i]=sumd[i-1]+d[i];
        cout<<sumd[i]<<" ";
    }
    return 0;
}
```

一维差分

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

const int N = 1e5 + 10;

int a[N]; //原数组
int d[N]; //差分数组
int s[N]; //原数组（修改后的）

int main()
{
    int n, m;
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> a[i];

    //求差分数组
    for(int i = 1; i <= n; i++) d[i] = a[i] - a[i-1];

    //m次修改差分数组
    for(int i = 1; i <= m; i++)
    {
        int l, r, x;
        cin >> l >> r >> x;
        d[l] += x, d[r + 1] -= x;
```

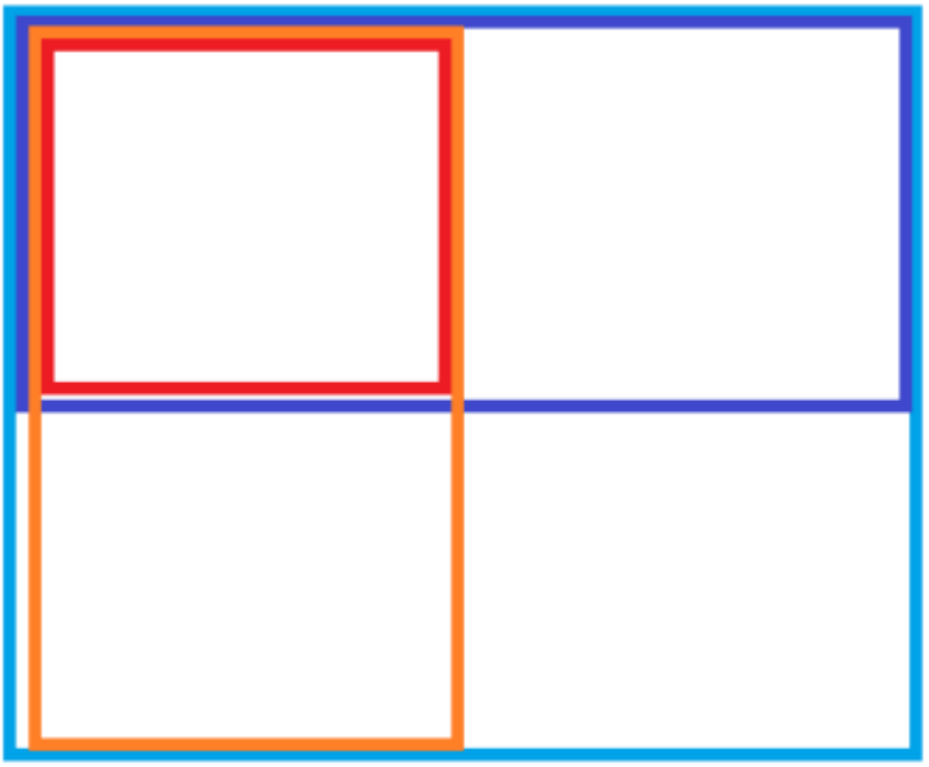
```
    }

    //对差分数组求前缀和，得到修改后的原数组
    for(int i = 1; i <= n; i++)
    {
        s[i] = s[i - 1] + d[i];
        cout << s[i] << " ";
    }
    return 0;
}
```

> **Tip**
>
> 还可以像下面所示例的代码一样，直接先把所有的差值算出来，然后相当于一个mask，再加上原先的数值

```
#include <bits/stdc++.h>
using namespace std;
int a[100005],d[100005];
int main()
{
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>a[i];
    while(m--)
    {
        int l,r,c;
        cin>>l>>r>>c;
        d[l]+=c;
        d[r+1]-=c;
    }
    for(int i=1;i<=n;i++) d[i]+=d[i-1]; //求出当前值更改了多少
    for(int i=1;i<=n;i++) cout<<a[i]+d[i]<<" "; //直接加上更新
}
```

二维差分

和一维差分一样，我们需要构造一个二维差分数组 $s[i][j]$，利用差分的思想:用当前位置原矩形的面积减去前面相邻的原矩形的面积

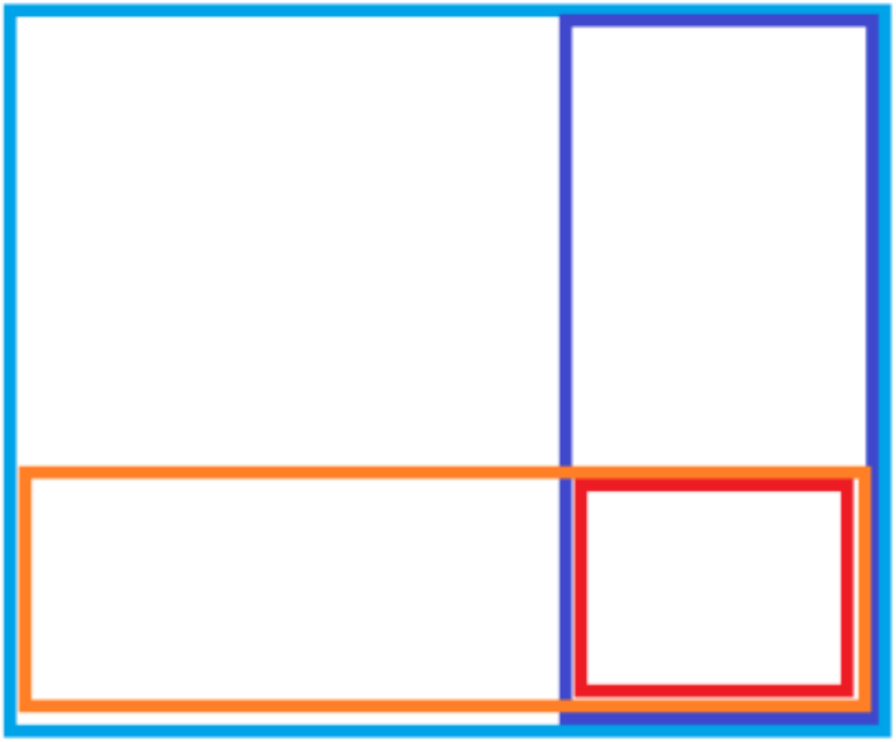


如图:差分面积=蓝色矩形面积一紫色矩形面积一橙色矩形面积+红色矩形面积(多减了一次)

即: $d[i][j] = a[i][j] - a[i-1][j] - a[i][j-1] + a[i-1][j-1]$

注意: $d[i][j]! = a[i][j] - a[i-1][j-1]$(边界)

给以 (x_1, y_1) 为左上角，(x_2, y_2) 为右下角的子矩阵中所有元素都加上 x 或者都减去 x 这个可以转换为:



1. $d[x_1][y_1] += x$ (蓝色区域面积都加上 x)
2. $d[x_1][y_2+1] -= x$ (紫色面积减 x，因为多加了一次)
3. $d[x_2+1][y_1] -= x$ (橙色矩形面积减 x，因为多加了一次)
4. $d[x_2+1][y_2+1] += x$ (红色矩形面积加 x，因为多减了一次)
5. 将差分数组求前缀和

二维差分

```
#include<iostream>
#include<cstdio>
using namespace std;
const int N = 1e3+5;
int a[N][N], s[N][N];

// (x1, y1) 到 (x2, y2) 增加 v
void insert(int x1,int y1,int x2,int y2,int v){
    a[x1][y1] += v;
    a[x2+1][y1] -= v;      // 不该加，减回去
    a[x1][y2+1] -= v;      // 不该加，减回去
    a[x2+1][y2+1] += v;    // 多减一次，加v进行抵消操作
}
```

```
int main(){
    int n,m,q;
    scanf("%d%d%d",&n,&m,&q);
    for(int i = 1; i <= n; i++){
        for(int j = 1;j <= m; j++){
            int t;
            scanf("%d",&t);
            insert(i,j,i,j,t);
        }
    }
    while(q--){
        int x1,y1,x2,y2,t;
        scanf("%d%d%d%d%d",&x1,&y1,&x2,&y2,&t);
        insert(x1,y1,x2,y2,t);
    }
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] +
a[i][j];
            printf("%d ",s[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

> **Tip**
>
> 同一维差分，二维差分也可以先求出所有的差分值类似于一个mask，然后再用原先的数加上

```
#include<iostream>
#include<cstdio>
using namespace std;
const int N = 1e3+5;
int a[N][N], s[N][N];
// (x1, y1) 到 (x2, y2) 增加 v
void insert(int x1,int y1,int x2,int y2,int v){
    a[x1][y1] += v;
    a[x2+1][y1] -= v; // 不该加，减回去
```

```
    a[x1][y2+1] -= v; // 不该加，减回去
    a[x2+1][y2+1] += v; // 多减一次，加v进行抵消操作
}
int main(){
    int n,m,q;
    scanf("%d%d%d",&n,&m,&q);
    for(int i = 1; i <= n; i++){
        for(int j = 1;j <= m; j++){
            int t;
            scanf("%d",&s[i][j]); //输入相应的值
        }
    }
    while(q--){
        int x1,y1,x2,y2,t;
        scanf("%d%d%d%d%d",&x1,&y1,&x2,&y2,&t);
        insert(x1,y1,x2,y2,t);
    }
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            a[i][j] += a[i-1][j] + a[i][j-1] - a[i-1][j-1]; //
求出当前值更改了多少
            printf("%d ",a[i][j]+s[i][j]); //直接加上更新
        }
        printf("\n");
    }
    return 0;
}
```

排序

快速排序

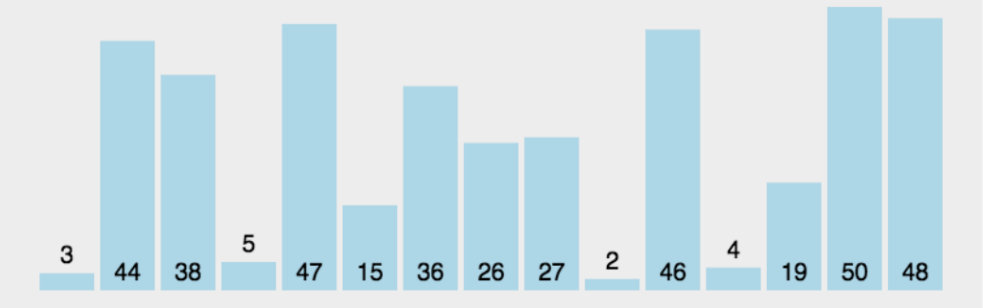
快速排序使用分治法来把一个串(list)分为两个子串(sub-lists)。具体算法描述如下:

- 从数列中挑出一个元素，称为“基准”(pivot);

- 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，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。

```
void quick_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

但更多的时候，我们为了方便都是用 `sort()`，快排的过程大家了解即可



> ⚠ **Warning**
>
> 手写的固定 `pivot` 的快速排序容易被卡超时，建议直接使用 `sort()`

分治

分治算法（Divide and Conquer Algorithm）是一种解决问题的基本算法范式，它将一个问题划分成若干个规模较小的子问题，然后递归地解决这些子问题，最后将这些子问题的解合并起来，得到原问题的解。分治算法通常包括三个步骤：分解（Divide）、解决（Conquer）、合并（Combine）。

1. **分解（Divide）**：将原问题划分成若干个规模较小、结构与原问题相似的子问题。这个过程可以递归地进行，直到子问题的规模足够小，可以直接求解。
2. **解决（Conquer）**：递归地解决每个子问题。当子问题的规模足够小时，不再继续递归，而是直接求解。这个步骤是分治算法的关键之一。
3. **合并（Combine）**：将子问题的解合并成原问题的解。这个步骤通常是简单且有效的。

分治算法的经典应用包括快速排序、归并排序、求解最大子数组、求解凸包等问题。

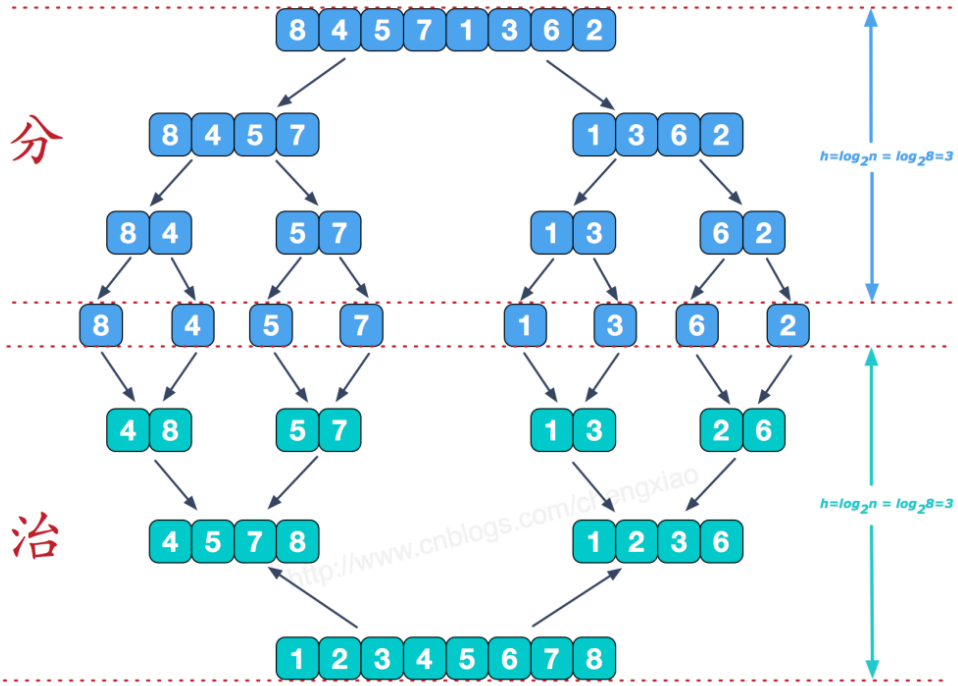
归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法(Divide and Conquer)的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列;即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

- 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;
费
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 直到某一指针达到序列尾;
5.将另一序列剩下的所有元素直接复制到合并序列尾。




```
const int N = 1e6+5;
int n, a[N], tmp[N];

void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    // 分（递归）
    int mid = l + r >> 1;          // 将数组平均分成两个部分
    merge_sort(q, l, mid);         // 对左半部分进行分治排序
    merge_sort(q, mid + 1, r);     // 对右半部分进行分治排序

    // 治（合并）
    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];

    while (i <= mid) tmp[k ++ ] = q[i ++ ];    // 可能的左半部分比右半部分多的
    while (j <= r) tmp[k ++ ] = q[j ++ ];      // 可能的右半部分比左半部分多的
```

```
    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```