1. **内联函数**

在C++中我们通常定义以下函数来求两个整数的最大值：

代码如下:

```
int max(int a, int b)
{
 return a > b ? a : b;
}
```

为这么一个小的操作定义一个函数的好处有：

① 阅读和理解函数 max 的调用，要比读一条等价的条件表达式并解释它的含义要容易得多

② 如果需要做任何修改，修改函数要比找出并修改每一处等价表达式容易得多

③ 使用函数可以确保统一的行为，每个测试都保证以相同的方式实现

④ 函数可以重用，不必为其他应用程序重写代码

虽然有这么多好处，但是写成函数有一个潜在的缺点：**==调用函数比求解等价表达式要慢得多。==**在大多数的机器上，调用函数都要做很多工作：调用前要先保存寄存器，并在返回时恢复，复制实参，程序还必须转向一个新位置执行

C++中支持内联函数，其目的是为了提高函数的执行效率，用关键字 inline 放在函数定义(注意是定义而非声明，下文继续讲到)的前面即可将函数指定为内联函数，内联函数通常就是将它在程序中的每个调用点上"内联地"展开，假设我们将 max 定义为内联函数：

```
inline int max(int a, int b)
{
 return a > b ? a : b;
}
```

则调用： `cout<<max(a, b)<<endl;` 在编译时展开为： `cout<<(a > b ? a : b)<<endl;`

从而消除了把 max写成函数的额外执行开销。

2. **内联函数和宏**

无论是《Effective C++》中的 "Prefer consts，enums，and inlines to #defines" 条款，还是《高质量程序设计指南——C++/C语言》中的"用函数内联取代宏"，宏在C++中基本是被废了，在书《高质量程序设计指南——C++/C语言》中这样解释到：

3. **将内联函数放入头文件**

关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。

如下风格的函数 Foo 不能成为内联函数：

代码如下:

```
inline void Foo(int x, int y);  // inline 仅与函数声明放在一起
void Foo(int x, int y)
{
 ...
}
```

而如下风格的函数 Foo 则成为内联函数:

代码如下:

```
void Foo(int x, int y);
inline void Foo(int x, int y)  // inline 与函数定义体放在一起
{
 ...
}
```

所以说，C++ inline函数是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了 inline 关键字，但我认为 inline 不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量C++/C 程序设计风格的一个基本原则：**声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。**

定义在类声明之中的成员函数将自动地成为内联函数，例如:

复制代码 代码如下:

```
class A
{
public:
 void Foo(int x, int y) { ... }  // 自动地成为内联函数
}
```

但是编译器是否将它真正内联则要看 Foo函数如何定义

内联函数应该在头文件中定义，这一点不同于其他函数。编译器在调用点内联展开函数的代码时，必须能够找到 inline 函数的定义才能将调用函数替换为函数代码，而对于在头文件中仅有函数声明是不够的。

当然内联函数定义也可以放在源文件中，但此时只有定义的那个源文件可以用它，而且必须为每个源文件拷贝一份定义(即每个源文件里的定义必须是完全相同的)，当然即使是放在头文件中，也是对每个定义做一份拷贝，只不过是编译器替你完成这种拷贝罢了。但相比于放在源文件中，放在头文件中既能够确保调用函数是定义是相同的，又能够保证在调用点能够找到函数定义从而完成内联(替换)。

第一讲 基础算法

1.1 排序(总结一下：时间复杂度 稳定性 排序思想)

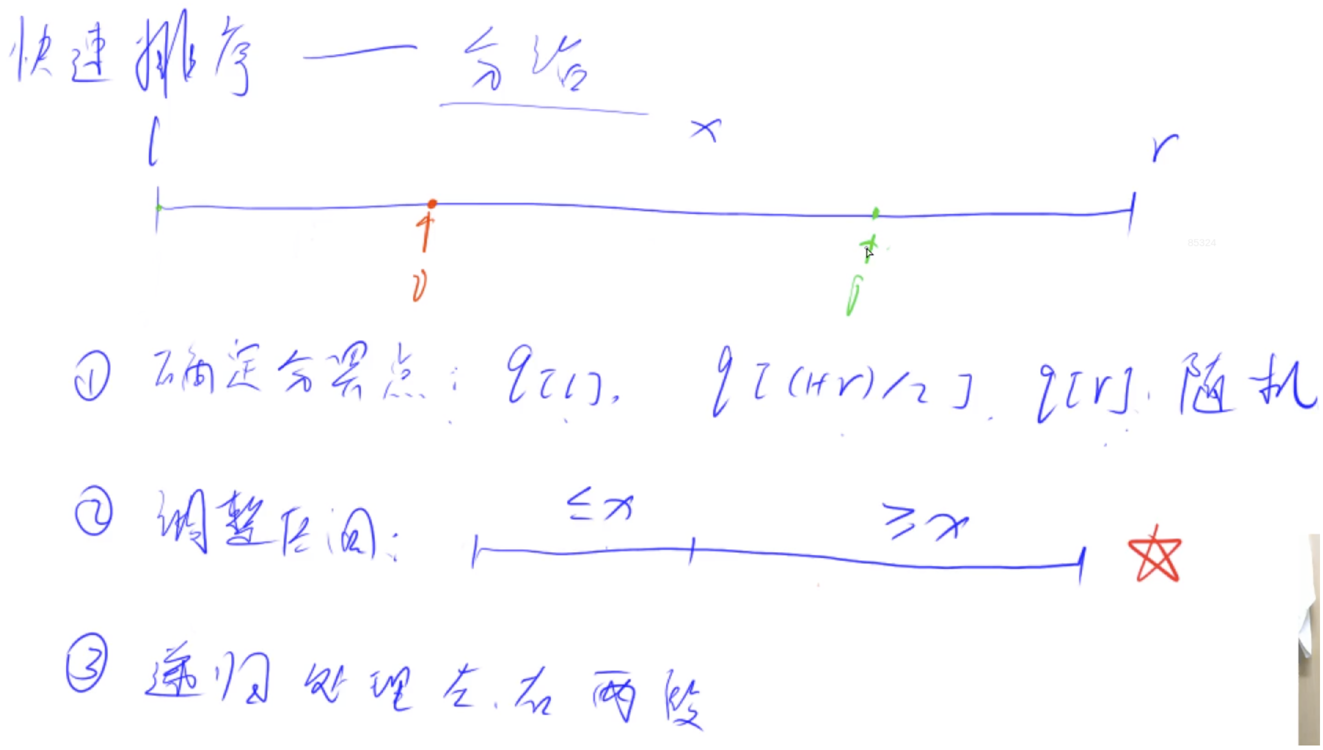
排序主要讲==快速排序==和==归并排序==。主要理解并背过代码模板，要达到能到快速把代码模板默写出来。这个工作很重要。然后用题目来检验模板是否记忆的很好，刷题大概可以重复3-5次。

1.1.1 快速排序

主要思想是基于分治。算法流程如下：

1. **确定分界点**
2. **调整区间**
3. **递归处理左右两段**

其中调整区间所用的思想是双指针：两端指针分别向中间移动，直到不满足`q[i]<x`,`q[j]>x`时停止移动，当双指针都停止移动时，交换此时它们所指向的值`swap`，然后继续移动，直到双指针相遇`i=j`。



==快排代码模板：==

```
//注意边界问题，分界点如果选取q[l]或q[r]，都有可能产生死循环，如果选中间则不会
#include <iostream>

using namespace std;

const int N = 100010;

int q[N];

void quick_sort(int q[], int l, int r)
{
    if (l >= r) return;//递归终止条件
```

```
    int i = l - 1, j = r + 1, x = q[l + r >> 1];//确定分界点 分界点为数组中具体的值
    while (i < j)//调整区间
    {
        do i ++ ; while (q[i] < x);//如果跳出循环，说明q[i]>=x
        do j -- ; while (q[j] > x);//如果跳出循环，说明q[j]<=x
        if (i < j) swap(q[i], q[j]);
    }

    quick_sort(q, l, j);//递归处理左右两段
    quick_sort(q, j + 1, r);
}

int main()
{
    int n;
    scanf("%d", &n);

    for (int i = 0; i < n; i ++ ) scanf("%d", &q[i]);

    quick_sort(q, 0, n - 1);

    for (int i = 0; i < n; i ++ ) printf("%d ", q[i]);

    return 0;
}
```

1.1.2 归并排序

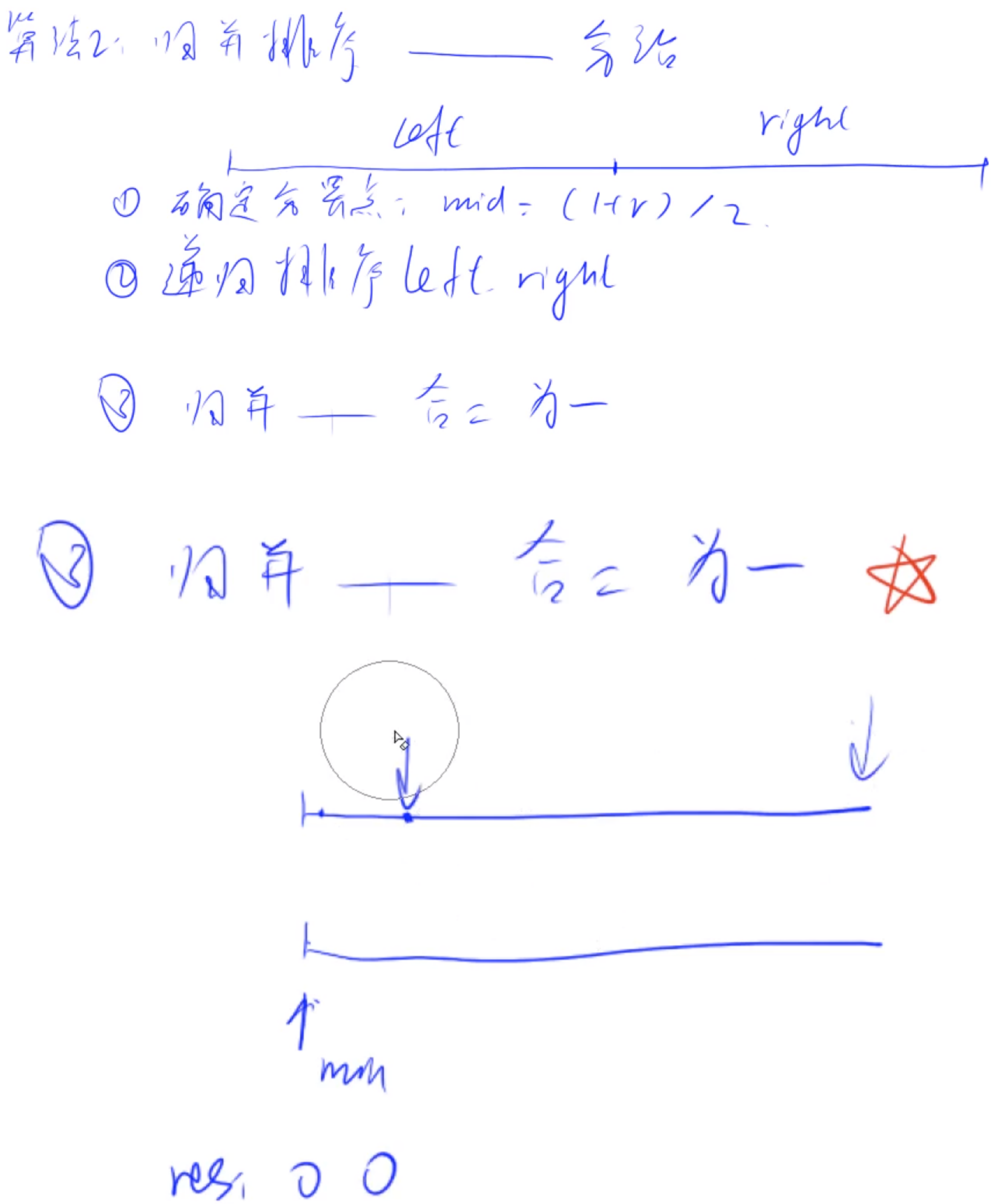
归并排序的思想也是分治，与快排不同之处是，快排选的是某个点的值，而归并是以数组中间点的位置来分。

算法流程如下：

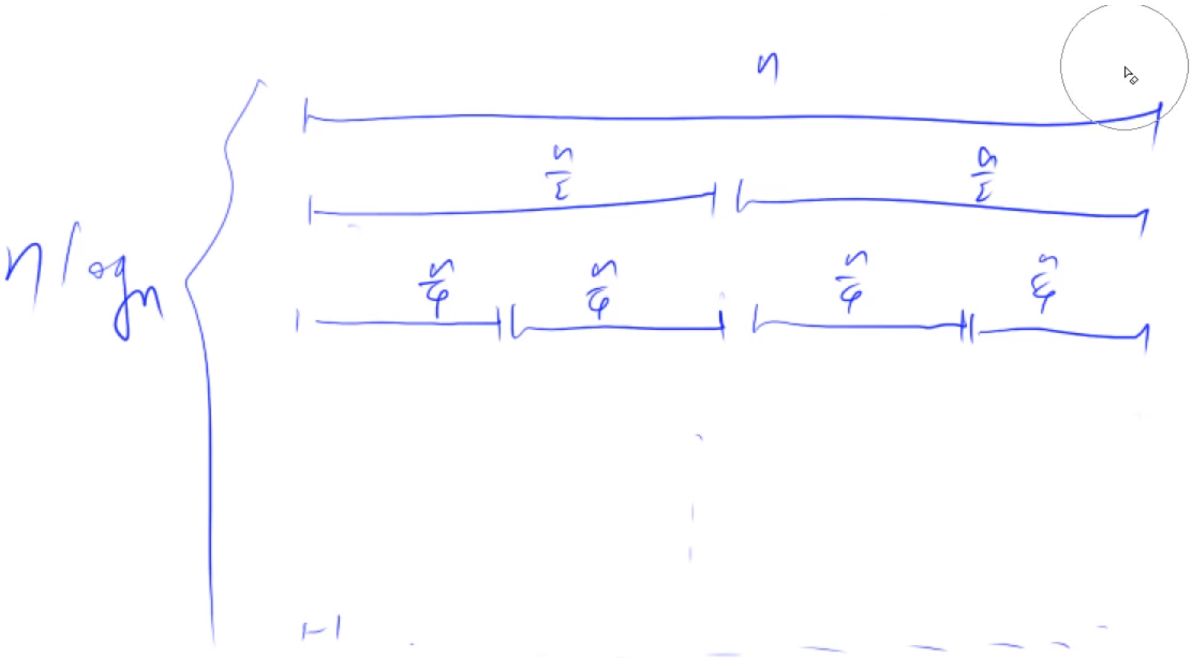
1.确定分界点`mid=(l+r)/2`，数组分为`left`,`right`；

2.对左右两边都进行递归排序,得到两个有序数组；

3.对两个有序数组进行合并，思路是双指针算法，用两个指针分别指向两个有序数组的最小值，然后对这两个最小值进行比较，最小的值放在`res`数组中，直到有一个指针到达尽头，另一个数组剩下的部分添加到队尾即可。



时间复杂度：每次遍历长度为n，通过递归一共有`log_n`层（因为要划分到长度为1的有序数组），所以总的时间复杂度是`O(nlogn)`。



==归并排序代码模板：==

```
#include<iostream>

using namespace std;

const int N = 100010;
int n;
int q[N],temp[N];

void merge_sort(int q[],int l,int r)
{
    if(l >= r) return; //递归终止条件
    int mid = l+r >> 1; //分界点
    merge_sort(q,l,mid),merge_sort(q,mid + 1,r); //递归左右数组

    int k = 0,i = l,j = mid + 1;  //合并两个有序数组到temp数组
    while(i <= mid && j <= r)
    {
        if(q[i] <= q[j]) temp[k++] = q[i++];
        else temp[k++] = q[j++];
    }
    while(i <= mid) temp[k++] = q[i++];//两个数组中还剩下的部分，加到temp数组后面
    while(j <= r) temp[k++] = q[j++];

    for(int i = l,j = 0;i <= r;i++,j++) q[i] = temp[j]; //从temp数组复制到原数组
}

int main()
{
    scanf("%d",&n);
    for(int i = 0;i < n;i++) scanf("%d",&q[i]);
```

```
    merge_sort(q,0,n-1);

    for(int i = 0;i < n;i++) printf("%d ",q[i]);
    return 0;
}
```

1.2 二分(考)

简单定义：在一个单调有序的集合中查找元素，每次将集合分为左右两部分，判断解在哪个部分中并调整集合上下界，重复直到找到目标元素。

时间复杂度：O (logn)，优于直接顺序查找O(n)

主要讲整数二分和浮点数二分。

1.2.1 整数二分

二分的本质：并不是单调性，如果有单调性就一定可以二分，可以二分的题目不一定非得有单调性，如果没有单调性，也有可能可以二分。**二分的本质是边界**，给定我们一个区间，**在这个区间上定义某种性质，这种性质在右半边满足，在左半边不满足**，如果可以找到这样一种性质，将区间一分为二的话，那二分就可以寻找这个性质的边界。（例如下图的两点）

==算法思路==：假设目标值在闭区间[l, r]中， 每次将区间长度缩小一半，当l = r时，我们就找到了目标值。当想要的target在左半区间时，就要更新右边界，即`r=mid`,所以相应地`l =mid + 1`,当target在右半区间时，需要更新左边界,即`l = mid` ,相应的`r = mid - 1`.

==二分模板==一共有两个，分别适用于不同情况。

版本1 当我们将区间[l, r]划分成[l, mid]和[mid + 1, r]时，其更新操作是r = mid或者l = mid + 1;，计算mid时不需要加1。

```
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;
        else l = mid + 1;
    }
    return l;
}
```

版本2

当我们将区间[l, r]划分成[l, mid - 1]和[mid, r]时，其更新操作是r = mid - 1或者l = mid;，==此时为了防止死循环，计算mid时需要加1。==

```
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

1.2.2 浮点数二分

浮点数二分就没有边界问题了，不用考虑+1的问题。思路：直接二分，选取想要的区间，更新边界值等于mid即可。

```
bool check(double x) {/* ... */} // 检查x是否满足某种性质

double bsearch_3(double l, double r)
{
    const double eps = 1e-6;   // eps 表示精度，取决于题目对精度的要求
    while (r - l > eps)
    {
        double mid = (l + r) / 2;
        if (check(mid)) r = mid;
        else l = mid;
    }
    return l;
}
```

1.3 高精度

一般考四种

1.两个比较大的整数相加，`A+B`，位数在10^6以内，即整数的长度在1000000以内。

2.两个比较大的整数相减`A-B`

3.一个大整数乘小整数`A*b`

4.`A/b`,求商和余

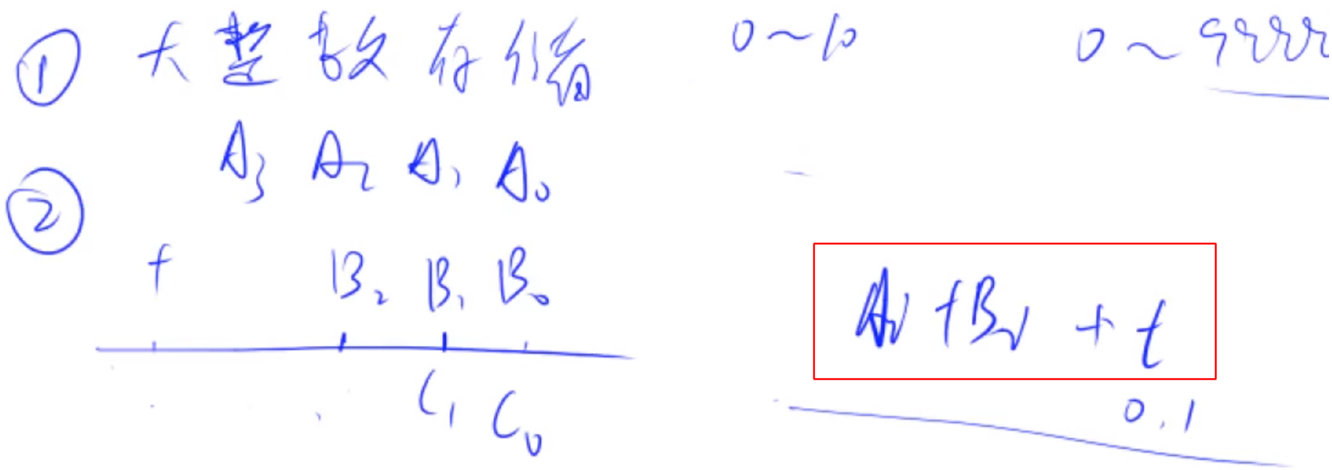
1.3.1 高精度加法

算法流程

1.首先是大整数的存储，==用int变量是存不下来的，其实是把大整数的每一位存到数组里面去==。存的时候就有问题了，是高位在前还是低位在前呢？

一般把个位（低位）放在数组首位，因为进位的时候，往数组尾端插入是最容易的。即==个位放首位，高位放队尾。==

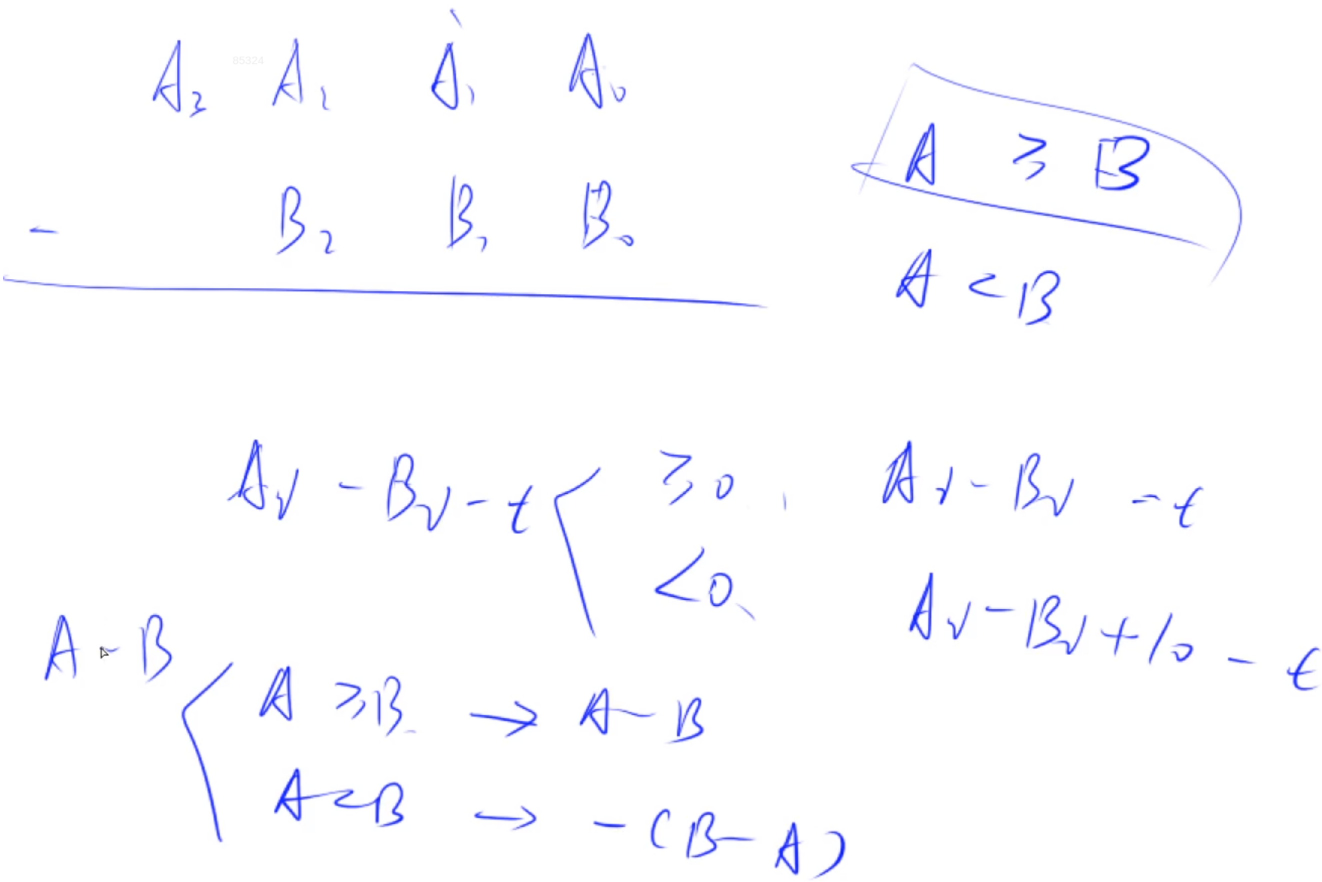
2.然后是用数组模拟加法。如果A还有，则加A，B还有，则加B



高精度加法-算法模板

```
// C = A + B, A >= 0, B >= 0
vector<int> add(vector<int> &A, vector<int> &B)
{
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )//从个位开始加
    {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;//进位
    }

    if (t) C.push_back(t);
    return C;
}
```

1.3.2 高精度减法



高精度减法-算法模板

```
// C = A - B, 满足A >= B, A >= 0, B >= 0
vector<int> sub(vector<int> &A, vector<int> &B)
{
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i ++ )//从个位开始减
    {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);//+10 是因为t可能减不到，是负的
        if (t < 0) t = 1;//意味着借位了，则计算高位的A[i]是要减去1
        else t = 0;//否则没有借位，正常减
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();//去掉前导0（即去掉数组高位-后面的0）
    return C;
}
```

1.3.3 高精度乘法

高精度乘低精度 —— 模板

```
// C = A * b, A >= 0, b >= 0
vector<int> mul(vector<int> &A, int b)
{
    vector<int> C;

    int t = 0;//进位
    for (int i = 0; i < A.size() || t; i ++ )
    {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);//取出个位
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();

    return C;
}
```

1.3.4 高精度除法

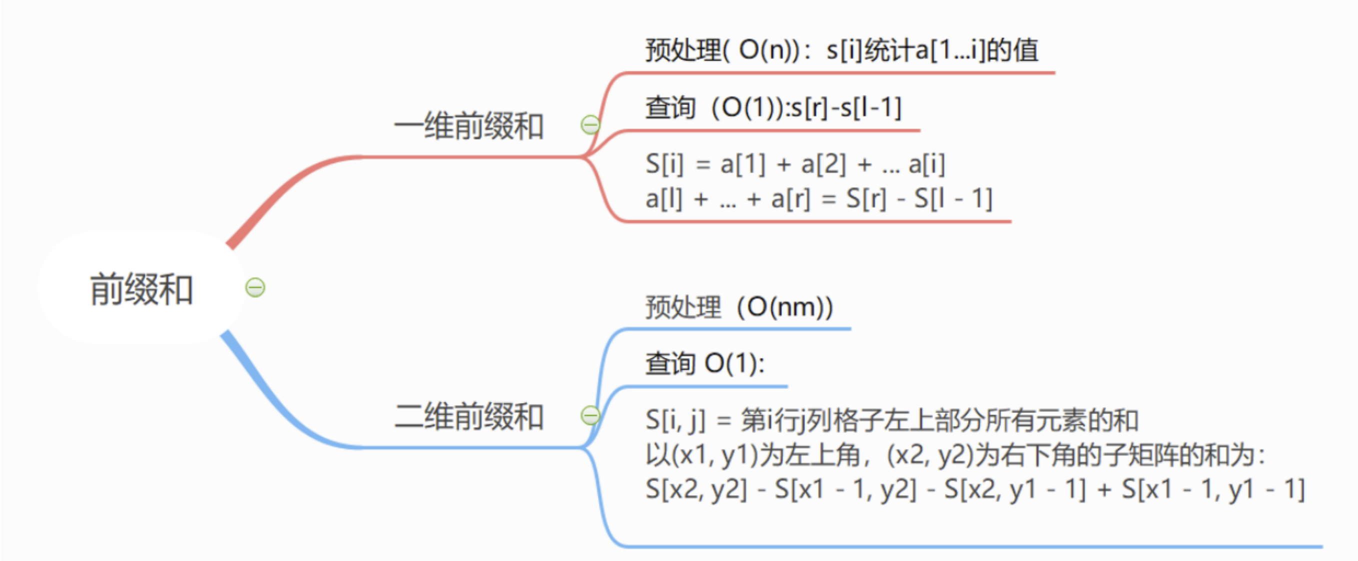
前面的加减乘都是从低位开始算，但除法是从高位开始算的。除法得到商和余

```
// A / b = C ... r, A >= 0, b > 0
vector<int> div(vector<int> &A, int b, int &r)
{
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i -- )
    {
        r = r * 10 + A[i];//中间量的余数，用于下一步的除法
        C.push_back(r / b);//把商加进去
        r %= b;//每一步剩下的余数
    }
    reverse(C.begin(), C.end());//因为前面的除法步骤是把高位数放在数组的低位的，这
与我们要求数组尾放高位是相反的，故需要逆转
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

1.4 前缀和差分

前缀和是指某序列的前n项和，可以把它理解为数学上的数列的前n项和，而差分可以看成前缀和的逆运算。合理的使用前缀和与差分，可以将某些复杂的问题简单化。

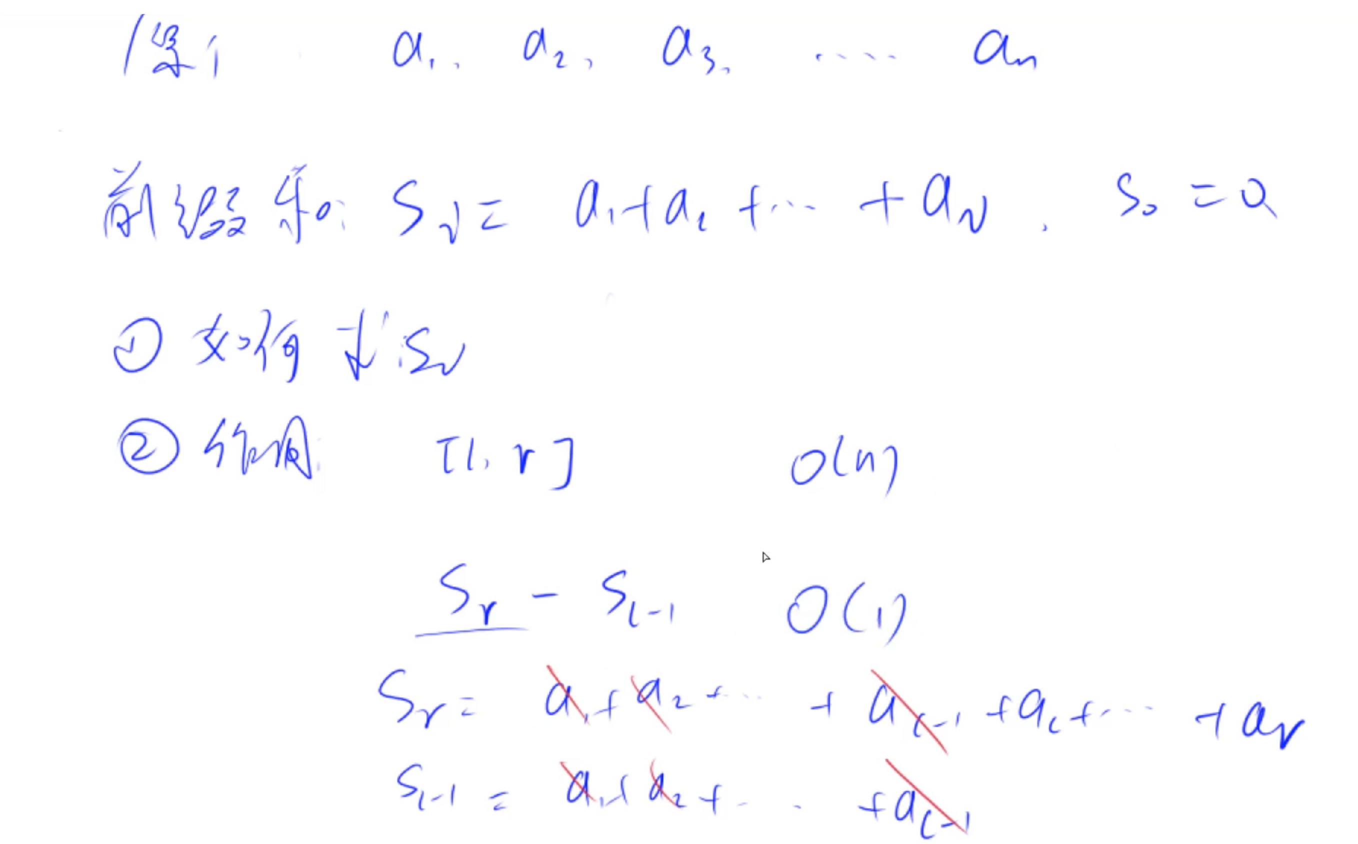
前缀和思维导图



1.4.1 一维前缀和

前缀和数组：$S_i=a_1+a_2+...+a_i$

==作用：快速求出原数组中一段数的和。可以用一次运算算出任意一段数的和。==



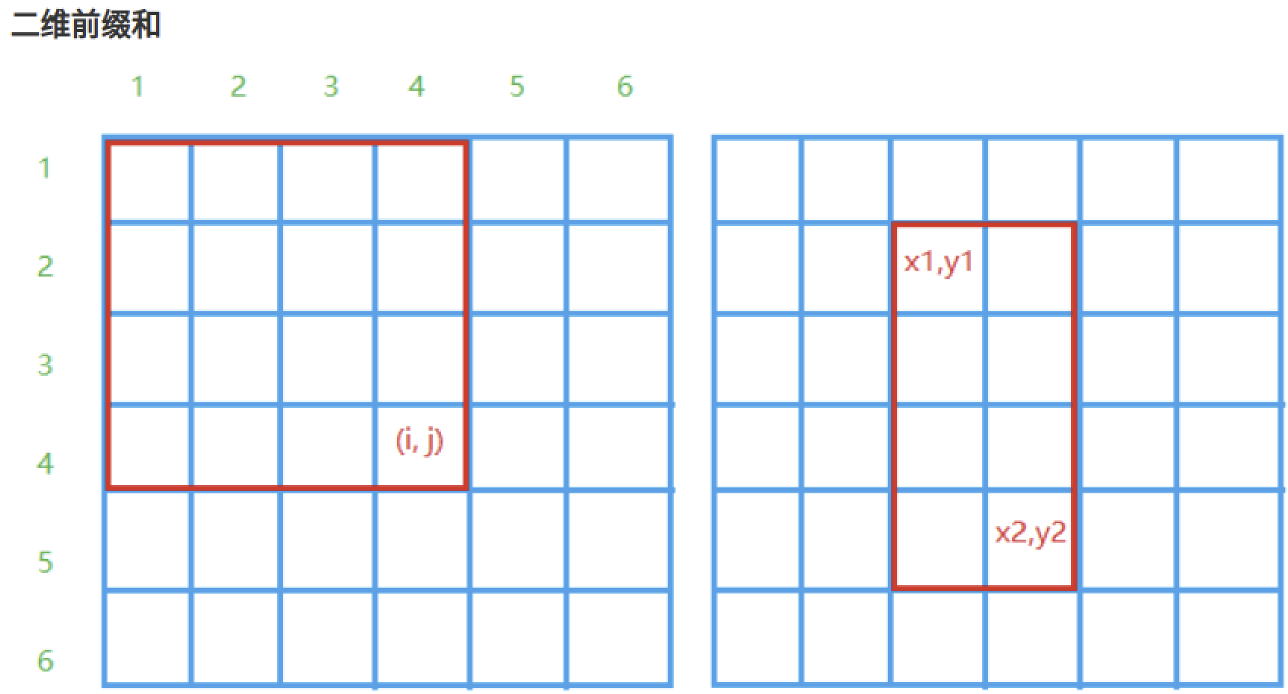
如何算S_i，$S_i=S_{i-1}+a_i$。

一维前缀和 —— 模板

```
S[i] = a[1] + a[2] + ... a[i]
S[i] = S[i-1]+a[i]
a[l] + ... + a[r] = S[r] - S[l - 1]
```

1.4.2 二维前缀和

S_{ij}表示左上角所有元素的和。



1. $S[i,j]$即为图1红框中所有数的的和为：

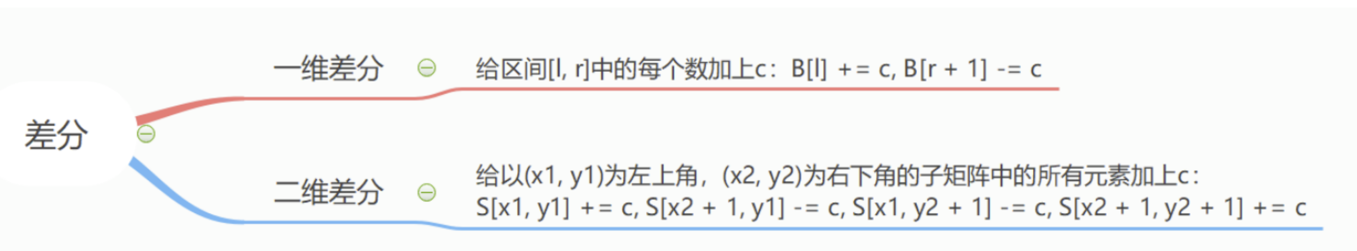
   $$S[i,j]=S[i,j-1]+S[i-1,j]-S[i-1,j-1]+a[i,j]$$

2. $(x_1,y_1),(x_2,y_2)$这一子矩阵中的所有数之和为：
   $S[x_2,y_2]-S[x_1-1,y_2]-S[x_2,y_1-1]+S[x_1-1,y_1-1]$

```
S[i, j] = 第i行j列格子左上部分所有元素的和
以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵的和为：
S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]
```

1.4.3 一维差分

差分思维导图：



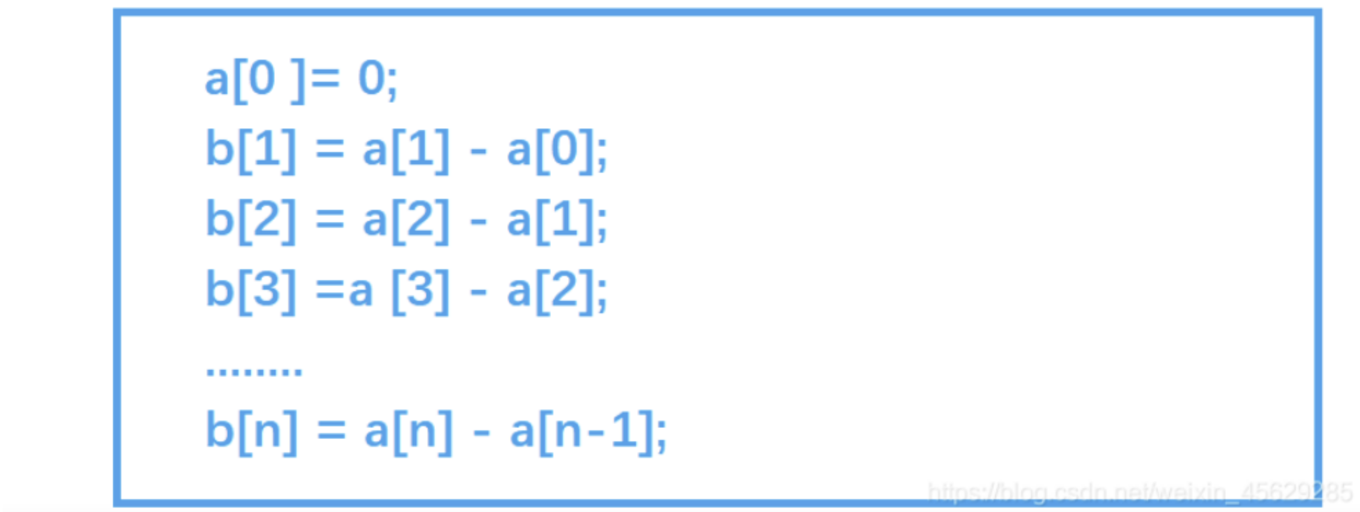
类似于数学中的求导和积分，**差分可以看成前缀和的逆运算**。

差分数组：首先给定一个原数组`a：a[1], a[2], a[3],,,,,, a[n]`；

然后我们构造一个数组`b : b[1],b[2],b[3],,,,,, b[i]`;使得 `a[i] = b[1] + b[2 ]+ b[3] +,,,,,, + b[i]`

也就是说，**a数组是b数组的前缀和数组，反过来我们把b数组叫做a数组的差分数组**。换句话说，每一个a[i]都是b数组中从头开始的一段区间和。考虑如何构造差分b数组？最为直接的方法:

> **如下:**
>
> `a[0 ]= 0;`
>
> `b[1] = a[1] - a[0]`;
>
> `b[2] = a[2] - a[1]`;
>
> `b[3] =a [3] - a[2]`;
>
> `........`
>
> `b[n] = a[n] - a[n-1]`;
>
> **图示:**


a[0]= 0;
b[1] = a[1] - a[0];
b[2] = a[2] - a[1];
b[3] =a [3] - a[2];
....
b[n] = a[n] - a[n-1];

我们只要有`b`数组，通过前缀和运算，就可以在`O(n)` 的时间内得到`a`数组 。

知道了差分数组有什么用呢？ 别着急，慢慢往下看。

话说有这么一个问题：

给定区间`[l ,r ]`，让我们把`a`数组中的`[ l, r]`区间中的每一个数都加上`c`,即 `a[l] + c , a[l+1] + c , a[l+2] + c ,,,,,, a[r] + c`;

暴力做法是`for`循环`l`到`r`区间，时间复杂度`O(n)`，如果我们需要对原数组执行`m`次这样的操作，时间复杂度就会变成`O(n*m)`。有没有更高效的做法吗? **考虑差分做法**，(差分数组派上用场了)。

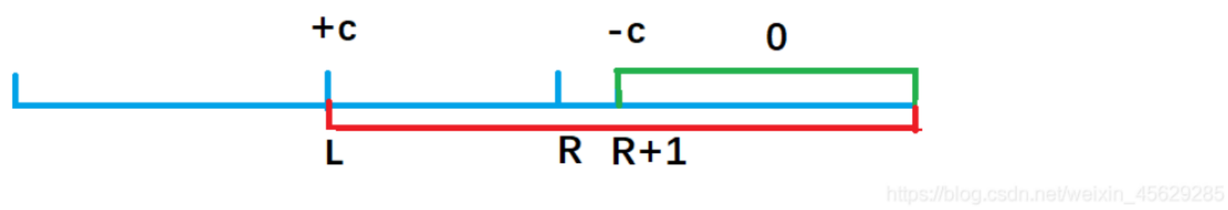
始终要记得，a数组是b数组的前缀和数组，比如对`b`数组的`b[i]`的修改，会影响到`a`数组中从`a[i]`及往后的每一个数。

首先让差分`b`数组中的 `b[l] + c` ,通过前缀和运算，`a`数组变成 `a[l] + c ,a[l+1] + c,,,,,, a[n] + c`;

然后我们打个补丁，`b[r+1] - c`, 通过前缀和运算，`a`数组变成 `a[r+1] - c,a[r+2] - c,,,,,,,a[n] - c`;

为啥还要打个补丁？

我们画个图理解一下这个公式的由来：



`b[l] + c`，效果使得`a`数组中 `a[l]`及以后的数都加上了`c`(红色部分)，但我们只要求`l`到`r`区间加上`c`, 因此还需要执行 `b[r+1] - c`,让`a`数组中`a[r+1]`及往后的区间再减去`c`(绿色部分)，这样对于`a[r]` 以后区间的数相当于没有发生改变。

因此我们得出**一维差分结论**：给`a`数组中的`[ l, r]`区间中的每一个数都加上`c`,只需对差分数组`b`做 `b[l] + = c`, `b[r+1] - = c`。时间复杂度为`O(1)`, 大大提高了效率。

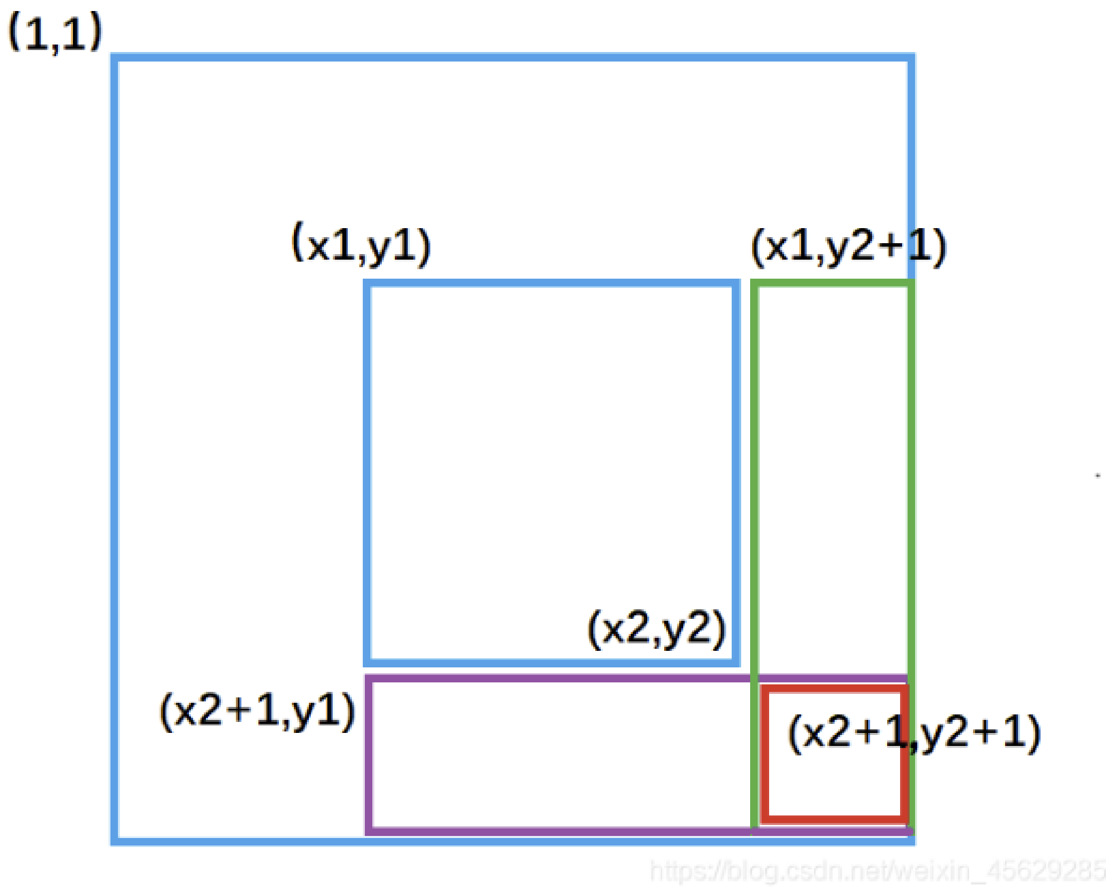
代码模板

```
#include<iostream>

using namespace std;

const int N = 1e6 + 10;

int n,m;
int a[N],b[N];

int main()
{
    cin >> n >> m;
    for(int i = 1;i <= n;i++)
    {
        scanf("%d",&a[i]);
        b[i] = a[i] - a[i-1];//构建差分数组
    }

    int l,r,c;
    while(m--)
```

```
    {
        cin >> l >> r >> c;
        b[l] += c;  //将序列中[l, r]之间的每个数都加上c
        b[r+1] -= c;
    }
    for(int i = 1;i <= n;i++)
    {
        a[i] = b[i] + a[i-1]; //前缀和运算,根据差分数组更新原数组
        printf("%d ",a[i]);
    }
    return 0;
}
```

1.4.4 二维差分

我们画个图去理解一下这个过程:



`b[x1][ y1 ] +=c` ; 对应图1 ,让整个 `a` 数组中**蓝色矩形面积**的元素都加上了 `c`。
`b[x1,][y2+1]-=c` ; 对应图2 ,让整个 `a` 数组中**绿色矩形面积**的元素再减去 `c`，使其内元素不发生改变。
`b[x2+1][y1]- =c` ; 对应图3 ,让整个 `a` 数组中**紫色矩形面积**的元素再减去 `c`，使其内元素不发生改变。
`b[x2+1][y2+1]+=c` ; 对应图4,,让整个 `a` 数组中**红色矩形面积**的元素再加上 `c`，红色内的相当于**被减了两次**，再加上一次 `c`，才能使其恢复。

```
#include<iostream>
using namespace std;
```

```
const int N = 1010;
int a[N][N],b[N][N];

void insert(int x1,int y1,int x2,int y2,int c)
{
    b[x1][y1] += c;
    b[x1][y2+1] -= c;
    b[x2+1][y1] -= c;
    b[x2+1][y2+1] += c;
}
int main()
{
    int n,m,q;
    cin >> n >> m >> q;
    for(int i = 1;i <= n;i++)
        for(int j = 1;j <= m;j++)
        {
            scanf("%d",&a[i][j]);
            insert(i,j,i,j,a[i][j]);//构建差分数组
        }
    while(q--)
    {
        int x1,y1,x2,y2,c;
        cin >> x1 >> y1 >> x2 >> y2 >> c;
        insert(x1,y1,x2,y2,c);
    }
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            a[i][j] = b[i][j]+a[i-1][j]+a[i][j-1]-a[i-1][j-1];//根据差分数组更新原数组
        }
    }
    for(int i = 1;i <= n;i++)
    {
        for(int j = 1;j <= m;j++)
        {
            printf("%d ",a[i][j]);
        }
        cout << endl;
    }
    return 0;
}
```

1.5 双指针算法(考)

双指针模板

```
for (int i = 0, j = 0; i < n; i ++ )
{
```

```
    while (j < i && check(i, j)) j ++ ;

    // 具体问题的逻辑
}
常见问题分类：
    (1) 对于一个序列，用两个指针维护一段区间
    (2) 对于两个序列，维护某种次序，比如归并排序中合并两个有序序列的操作
```

###最长连续不重复子序列 ####题目描述 给定一个长度为 n 的整数序列，请找出最长的不包含重复的数的连续区间，输出它的长度。 **输入格式** 第一行包含整数 n。 第二行包含 n 个整数（均在 0～105 范围内），表示整数序列。 **输出格式** 共一行，包含一个整数，表示最长的不包含重复的数的连续区间的长度。 数据范围 `1≤n≤105` **输入样例** 5 1 2 2 3 5 **输出样例** 3

思路: 首先是i枚举数组中的每一个位置，在i的某个位置中，j严格<=i,如果出现重复的元素，那么一定是枚举到的当前的a[i]，于是j指针可以开始移动了,只要这个重复的数不从这个"区间"中剔除，那么j就一直往右移动，最多是与i重合，那么while()停止。 抽象成理解的角度就为: 每次维护一个区间，一但出现重复的数，此时将该区间缩减(区间更新)，使区间的头为与ai重复的第一个数的后面,这样可以就有最优解。

1.6 位运算

模板

```
求n的第k位数字: n >> k & 1
返回n的最后一位1：lowbit(n) = n & -n
```

1.7 离散化

离散化模板

```
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素

// 二分求出x对应的离散化的值
int find(int x) // 找到第一个大于等于x的位置
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1; // 映射到1, 2, ...n
}
```

区间和 题目描述 假定有一个无限长的数轴，数轴上每个坐标上的数都是 0。 现在，我们首先进行 n 次操作，每次操作将某一位置 x 上的数加 c。 接下来，进行 m 次询问，每个询问包含两个整数 l 和 r，你需要求出在区间 [l,r] 之间的所有数的和。 **输入格式** 第一行包含两个整数 n 和 m。 接下来 n 行，每行包含两个整数 x 和 c。 再接下来 m 行，每行包含两个整数 l 和 r。 **输出格式** 共 m 行，每行输出一个询问中所求的区间内数字和。 **数据范围** −109≤x≤109, 1≤n,m≤105, −109≤l≤r≤109, −10000≤c≤10000 **输入样例** 3 3 1 2 3 6 7 5 1 3 4 6 7 8 输出样例 8 0 5

1.8 区间合并

流程：

1.按照左端点排序 2.

第二讲 数据结构

2.1 单链表（定义）

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int v) { val = v;}
    ListNode(int v = 0) { val = v;}
    ListNode(int v, ListNode* n) { val = v, next = n; }
};

ListNode* head = new ListNode(0);
ListNode* head = new ListNode();
// node1 是一个已经定义好的节点
ListNode* head = new ListNode(0, node1);   // head -> node1
```

2.2 双链表

2.3 栈

2.4 队列

2.5 单调栈

2.6 单调队列

2.7 KMP

2.8 Tire

2.9 并查集

2.10 堆

2.11 哈希表

第三讲 搜索与图论

3.1 DFS与BFS

3.2 树与图的遍历：拓扑排序

3.3 最短路

3.4 最小生成树

3.5 二分图：染色法、匈牙利算法

第四讲 数学知识（基本不考）

第五讲 动态规划(考)

5.1 背包问题

5.2 线性DP

5.3 区间DP

5.4 计数类DP

5.5 数位统计DP

5.6 状态压缩DP

5.7 树形DP

5.8 记忆化搜索

第六讲 贪心

6.1 区间问题

6.2 Huffman树

6.3 排序不等式

6.4 绝对值不等式

6.5 推公式

第七讲 时间复杂度分析