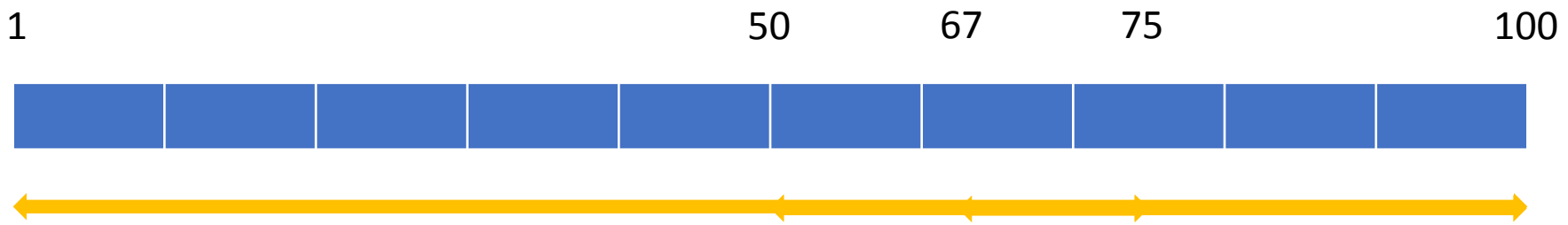
C++编程

目录

二分查找

- 任给一个大小不超过100的正整数，每次根据情况给出猜大/猜小/猜中的提示，最坏的情况下需要多少次一定可以猜中？
- 答案我们都知道是7次，因为$\log_2 100 \approx 6.643856 < 7$



二分查找

- 长度为n的某整数序列，要找出某数m是否在序列中，若在则返回其位置序号（如有多个则任一皆可），否则返回-1

Sample input	Sample output
7 12 //n和m 2 4 5 7 10 12 15	5

分析

- 我们需要设立L、R和mid三个变量，来记录不断变化的区间（实际通过记录数组下标来标识区间最方便）
- 收缩区间这件事需要循环做，有两个结束标记：

1. L、R碰头了（区间已经无可再缩了）
2. 找到m了（区间无需再缩）

- 这个算法里谁在找 m？
- 是 mid

分析

```
L 和 R 赋初值;
while (L <= R)
{
    mid 赋初值;
    if (找到m) 返回位置;
    else if (m在左半区间) 收缩 R;
    else 收缩 L;
}
return -1;  // L、R碰头都没能返回位置，必然是没找到 m
```

参考代码

特别注意：如果输入数据不能保证有序，那么在查找前需要排序，否则结果是错的

```
#include<bits/stdc++.h>
using namespace std;
int a[10010], n, m;
int find(int x)
{
    int L = 0, R = n - 1;
    while (L <= R)
    {
        int mid = (L + R) / 2;
        if (x == a[mid]) return mid;
            else if (x < a[mid]) R = mid - 1;
                else L = mid + 1;
    }
    return -1;
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i ++) scanf("%d", &a[i]);
    printf("%d", find(m));
    return 0;
}
```

换一种问法

- 长度为n的某有序整数序列，找出m应该插入的位置
- 区别在于：不存在无解的情况

Sample input	Sample output
7 9 //n和m 2 4 5 7 10 12 15	4

参考代码

```
#include<bits/stdc++.h>
using namespace std;
int a[10010], n, m;
int find(int x)
{
    int L = 0, R = n - 1, mid;
    while (L <= R)
    {
        mid = (L + R) / 2;
        if (x < a[mid]) R = mid - 1;
            else L = mid + 1;
    }
    return mid;
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i ++) scanf("%d", &a[i]);
    printf("%d", find(m));
    return 0;
}
```

lower_bound 函数

```
7 9
2 4 5 7 10 12 15
4
--------------------------------

Process exited after 7.086 seconds wi
th return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[10010], n, m;
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i ++) scanf("%d", &a[i]);
    printf("%d", lower_bound(a, a + n, m) - a);
    //在数组a的[0,n)范围内，寻找 m比数组元素小的第一个位置
    return 0;
}
```

lower_bound 函数

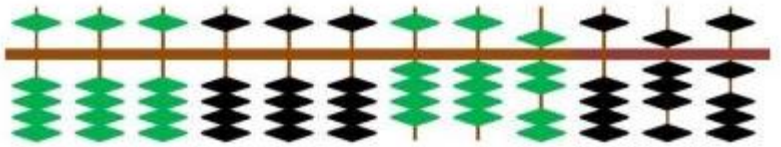
- lower_bound函数实际就是STL中自带的二分查找

```
int find(int m)
{
    int L = 0, R = n - 1, mid;
    while (L <= R)
    {
        mid = (L + R) / 2;
        if (m < a[mid]) R = mid - 1;
            else L = mid + 1;
    }
    return mid;
}
```

```
lower_bound(a, a + n, m) - a
```

- 对应的还有upper_bound函数

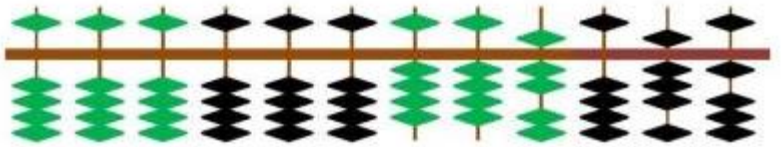
珠心算测验



- 某学校的珠心算老师采用一种快速考察珠心算加法能力的测验方法。他随机生成一个正整数集合，集合中的n（n≤10000）个数各不相同，然后要求学生回答：其中有多少个数，恰好等于集合中另外两个（不同的）数之和？

Sample input	Sample output
4 //n 1 2 3 4	2
样例解释：1+2=3，1+3=4，共2组答案	

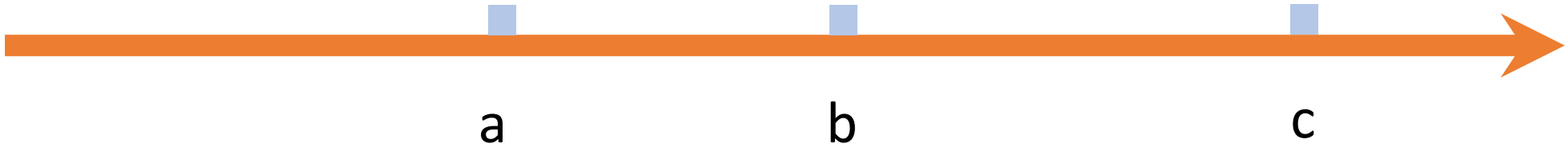
分析



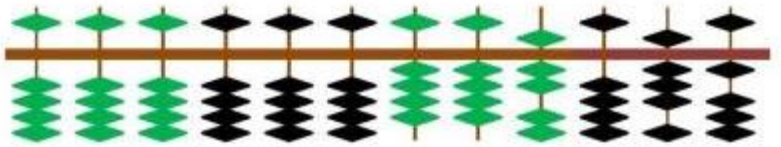
- 我们假设这三个数构成的加法等式为：num[a]+num[b]=num[c]
- 大致的思路就是穷举法，分别穷举符合条件的a、b、c

```
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        for (c = b + 1; c < n; c ++)
        {

        }
```



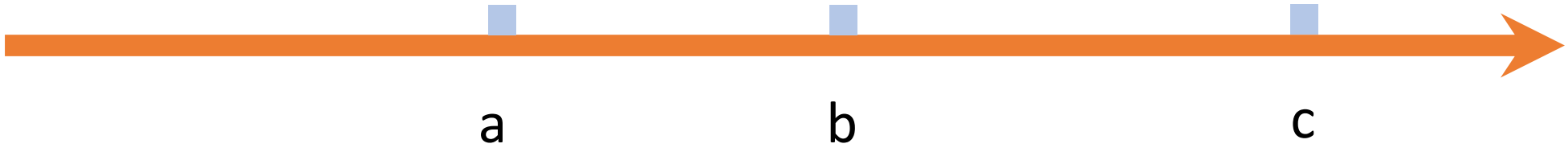
分析



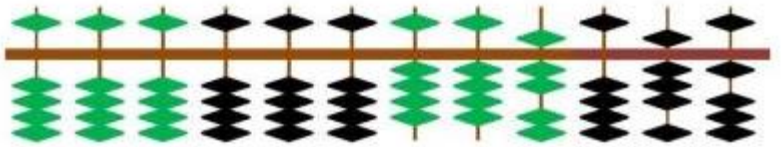
- 为了提高穷举的效率，我们还要先把数组排序
- **这是经常用的数据预先处理环节**

```
数组先排序;
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        for (c = b + 1; c < n; c ++)
        {

        }
```



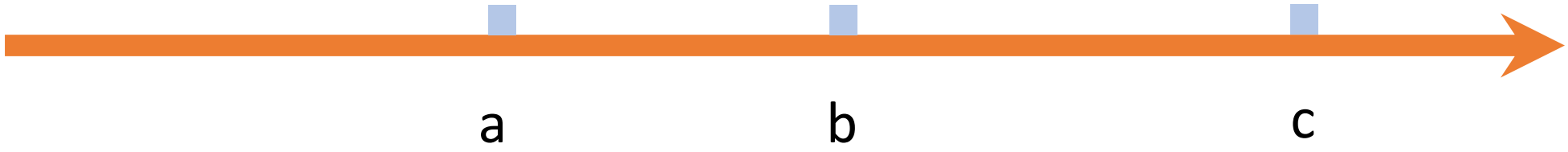
分析



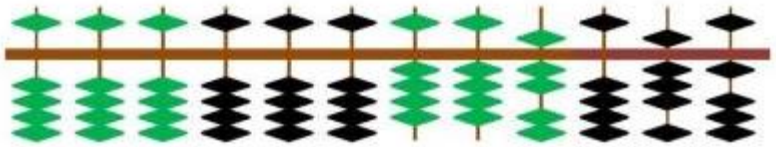
- 等等，这样的想法有漏洞吗?
- 2+3=5，1+4=5这样的组合方式，是不合题意的，属于重复计数

```
数组先排序;
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        for (c = b + 1; c < n; c ++)
        {

        }
```



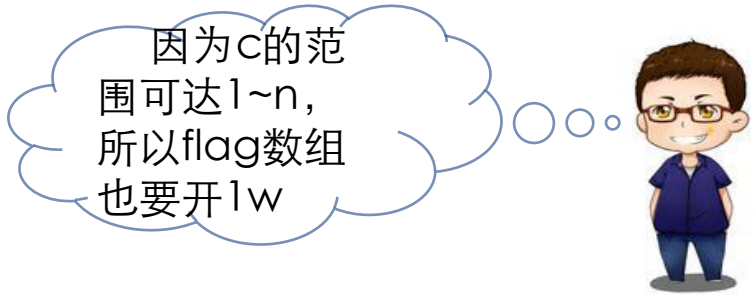
分析

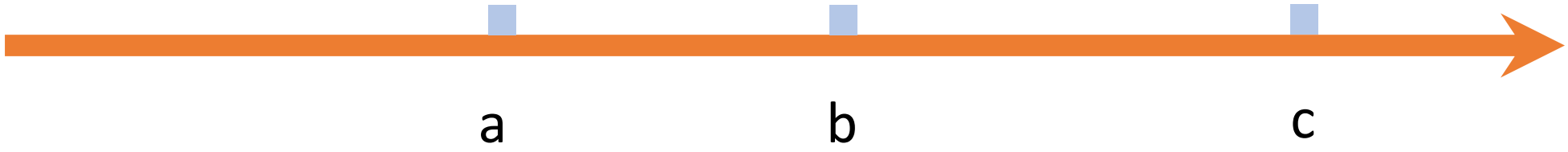


- 所以我们有必要为c设一个标记数组，标记这个c是否被组合过
- 1+4=5被先组合出来，2+3=5这样的组合就不被计入了

```
int flag[10010];

三层循环穷举a、b、c
if (等式成立 && flag[c] == 0)
    ans ++, flag[c] = 1;
```

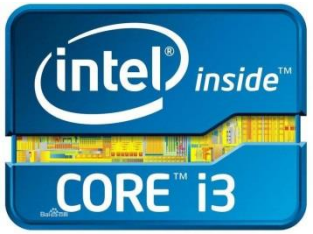




参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
int num[10010], flag[10010], n, i, a, b, c, ans;
int main()
{
    scanf("%d", &n);
    for (i = 0; i < n; i ++) scanf("%d", &num[i]);
    sort(num, num + n);
    for (a = 0; a < n; a ++)
        for (b = a + 1; b < n; b ++)
            for (c = b + 1; c < n; c ++)
                if (num[a] + num[b] == num[c] && !flag[c])
                    {ans ++; flag[c] = 1;}
    printf("%d", ans);
    return 0;
}
```

然而



- 这个测试点的规模是多大呢？n=5000

选手：n^3算法

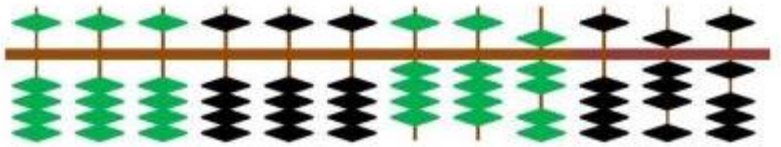
试题 count （重新测试）

源程序：count.cpp

测试点	输入文件	测试结果	运行用时	内存消耗	得分
#1	count5.in	答案正确	15.984 s	2.262 MB	100

- 为什么这么慢？因为我们这个程序的复杂度是$O(n^3)$的
- 10000的数据规模就没必要测试了，因为n扩大一倍，n^3将扩大8倍，那么差不多要2分钟，你几乎以为它死循环了。。

效率分析

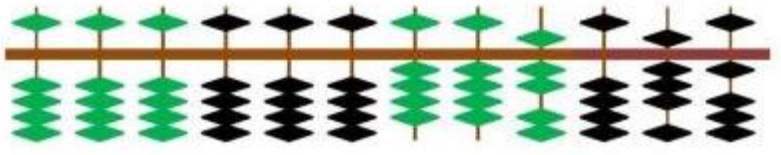


- 我们这个程序的复杂度是O(n^3)的，其复杂度的主要来源，或者说程序运行的主要耗时，在下面这个三层循环：

```
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        for (c = b + 1; c < n; c ++)
            if (num[a] + num[b] == num[c] && !flag[c])
                {ans ++; flag[c] = 1;}
```

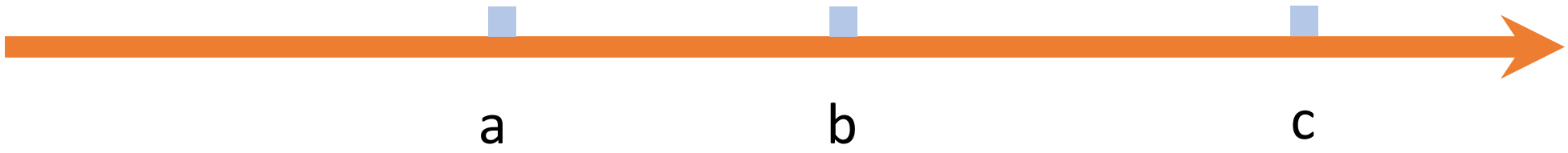
- 优化它，势在必行！

优化

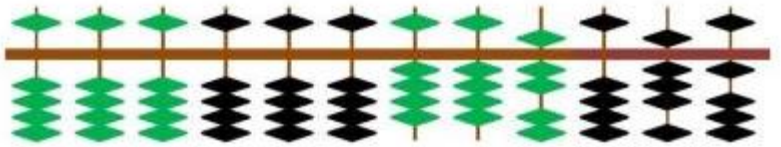


- 根据我们之前穷举优化的经验，要么缩小范围，要么降维
- 三个待穷举的未知量a、b、c，完全可以只要穷举其中的两个！

```
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        for (c = b + 1; c < n; c ++)
            if (num[a] + num[b] == num[c] && !flag[c])
                {ans ++; flag[c] = 1;}
```



优化

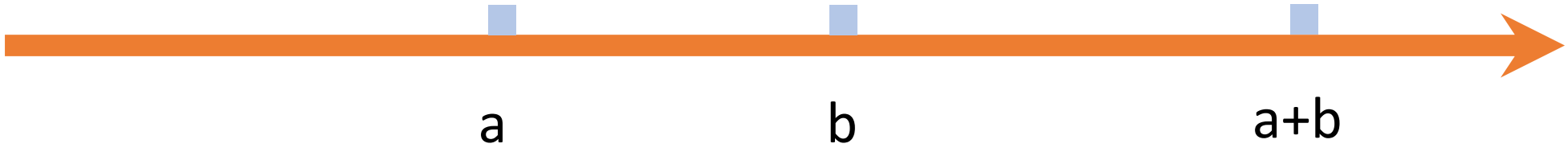


- 我们枚举两个加数a和b，则：

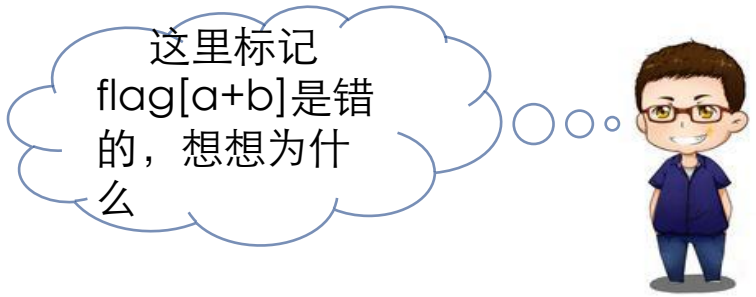
num[c]=num[a]+num[b]

```
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
    {

     }
```

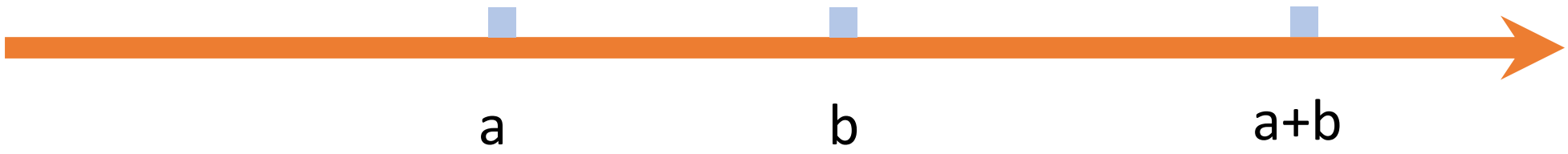


优化

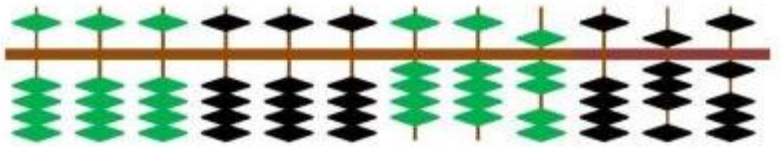


- 那我们只需要确定num[a]+num[b]在数列中，就找到了c的一种组合方式
- 注意：因为我们丢掉了下标c，所以标记数组只能标记数字本身

```
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        if (num[a]+num[b]在数列中 && !flag[num[a]+num[b]])
            {ans ++; flag[num[a]+num[b]] = 1;}
```

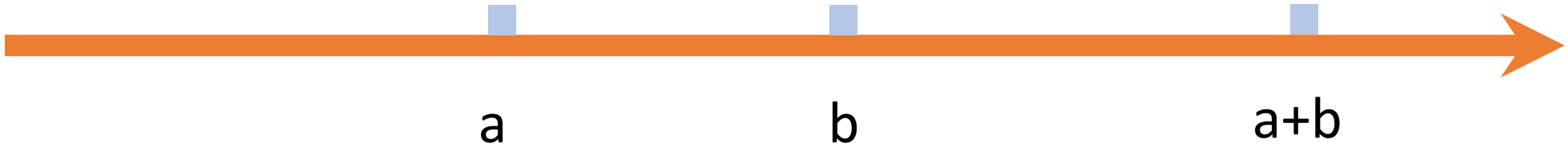


优化

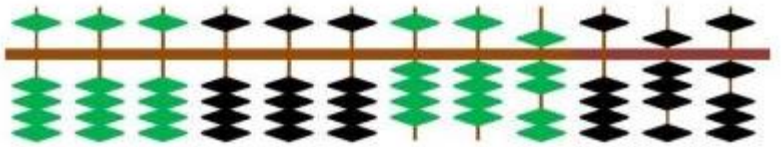


- 反之num[a]+num[b]就无法拆分成num[a]和num[b] 的和（因为num[a]+num[b]不存在）

```
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        if (num[a]+num[b]在数列中 && !flag[num[a]+num[b]])
            {ans ++; flag[num[a]+num[b]] = 1;}
```

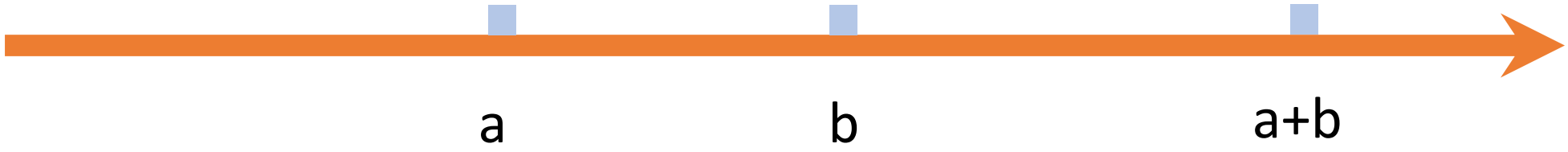


优化

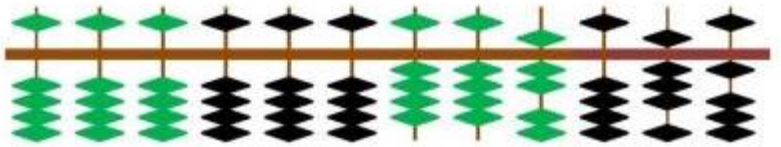


- 那怎样快速查找num[a]+num[b]是否在数列中？
- 如果还是从1~n扫一遍，那相当于又退回到三层循环
- 二分查找！二分查找的复杂度是O(logn)的，于是总复杂度$O(n^2 logn)$

```
for (a = 0; a < n; a ++)
    for (b = a + 1; b < n; b ++)
        if (find(num[a]+num[b]) && !flag[num[a]+num[b]])
        //此处调用二分查找
            {ans ++; flag[num[a]+num[b]] = 1;}
```

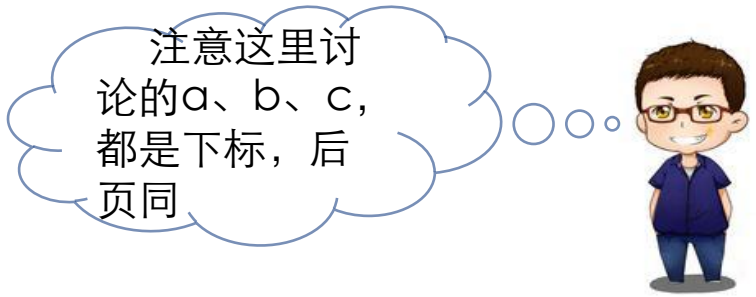


继续优化



- 复杂度从$O(n^3)$降到$O(n^2 logn)$，能不能更进一步？
- 琢磨这个式子：a+b=c
- 再借鉴二分查找的思路：两头向中间收缩
- 这里的a和b也不一定非得要从头开始扫，可以从两头向中间扫

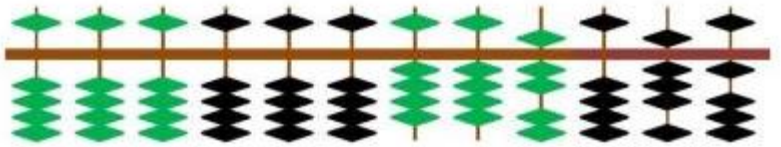
继续优化



- 如果这样做可以的话，相当于a和b合起来扫过一个n的范围
- 需要解决几个问题：

1. a、b的循环上下限
2. a+b≠c后，怎么继续

继续优化

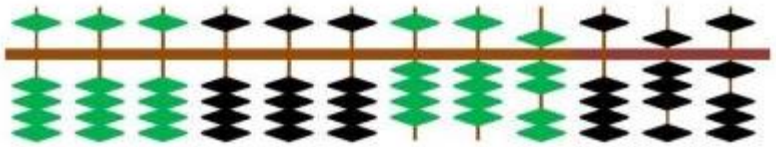


1. a、b的循环上下限

a：从0号位置出发向右走

b：从c-1号位置出发向左走

继续优化



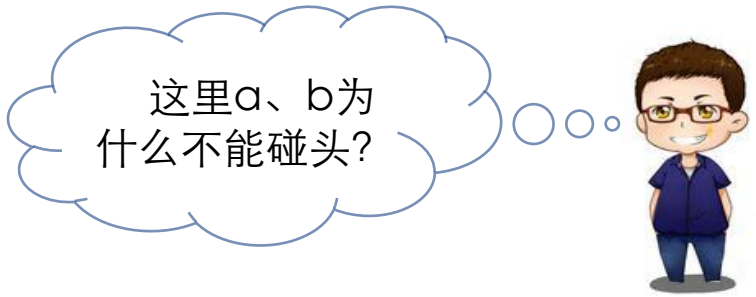
2. a+b≠c后，怎么继续

要分两种情况讨论：

a+b<c，说明a过小，a++

a+b>c，说明b过大，b--

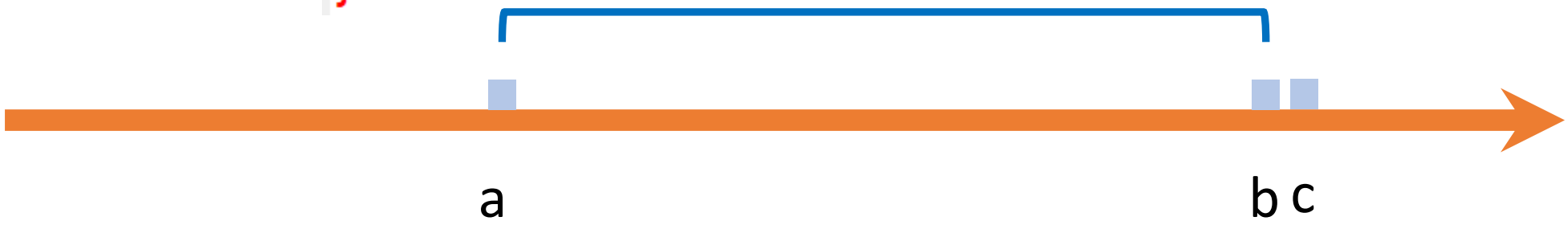
继续优化



- 而且我们意外地发现：我们不再需要标记数组：只需要在a+b=c时，ans++之后，break掉即可进入对下一个c的判断

```
int a = 0, b = c - 1;
while (a < b)  //这里a、b不能碰头
{
    if (a + b < c) a ++;
        else if (a + b > c) b --;
            else {ans ++; break;}

}
```



参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
int n, ans, num[10010];
int main()
{
    scanf("%d", &n);
    for (int i = 0; i < n; i ++) scanf("%d", &num[i]);
    sort(num, num + n);  // 先排序
    for (int c = n - 1; c >= 2; c --)    //逆向枚举和 c
    {
        int a = 0, b = c - 1;    //两个加数a、b分别从左右向中间逼近
        while (a < b)  // 注意 a ≠ b
        {
            if (num[a] + num[b] < num[c]) a ++;
                else if (num[a] + num[b] > num[c]) b --;
                    else {ans ++; break;}
        }
    }
    printf("%d", ans);
    return 0;
}
```

小结

- 我们最终不仅仅是直接调用二分查找，而且更进一步把二分查找的思想运用进来解题
- 最终的复杂度O(n^2)（因为break的存在，实际效率更高）

在线算法和离线算法

- 在算法设计的过程中，某些时候并不需要输入全部的数据，就可以得出（到输入当前数据为止的）答案
- 就是我们看到的“输入一组数据，得到一个答案，再输入一组数据，再得到一个答案”，类似于在线交互的过程，这种类型的算法被称为在线算法

- 离线算法则刚好相反，只有当全部数据都输入完成后，才可能算出答案，称为离线算法

在线算法和离线算法

- 以我们之前写的“猜数游戏”为例：

1. 如果我们每输入一个数，就返回一个结果“猜大”、“猜小”、“猜中”

- 它当然是在线算法

- 这种情况下，就没法写成离线算法
- 除非我们猜了一个数，程序也不给反应，直到我们用完5次机会，然后程序反馈回来：我们每次是猜大了，还是猜小了，或者在哪一次猜中了。这是离线了，但猜数游戏没法这么玩

在线算法和离线算法

- 再以我们之前写的排序为例：

2. 我们调用sort函数对a数组排序

- 它当然是离线算法（若数据没有全部输入，sort的结果无法保证正确）

- 这种情况下，就没法写成在线算法
- 除非我们每新输入一个数，程序就把所有的数据重新洗牌一次，但这样效率损失非常大

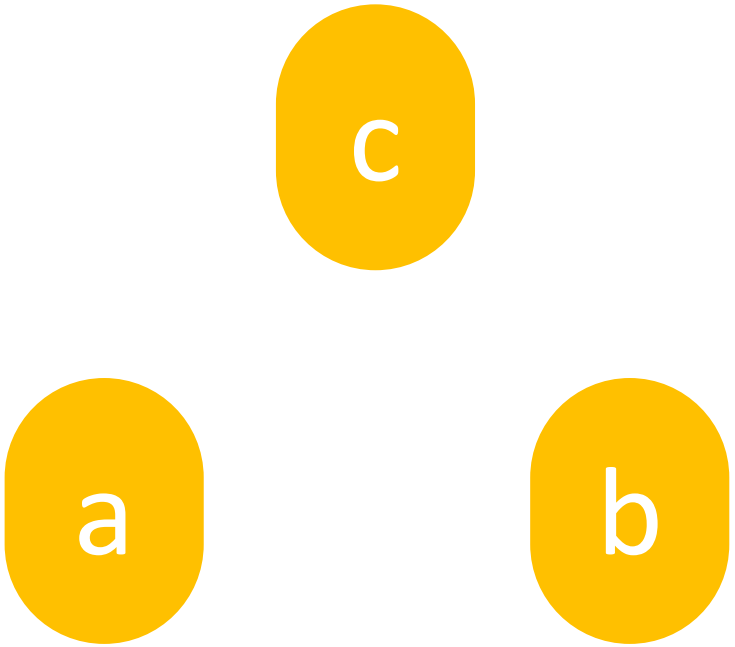
在线算法和离线算法

- 等等，继续刚才的例子：数据是动态变化的，不断有新数据加入（这种应用场景在现实生活中简直太常见），要求排好序

- 也就是：要排序，但不断有新数加入，怎么处理？

插入排序

- 还记得这个动图吗？



插入排序

- 把这个思想延伸，就是插入排序算法：

1、4、2、8、5、7

参考代码（在线版本）

1、for j 循环为何要逆向?
2、离线版本大家可以自己写
3、这里所谓在线版本，考虑输出的问题，并非完全在线

```
#include<bits/stdc++.h>
using namespace std;
int a[100010], i, j, n, m;
void insert(int x)
{
    if (x < a[i-1])
    {
        for (j = i - 1; j >= 0 && a[j] > x; j --)
            a[j+1] = a[j];     //平移
        a[j+1] = x;            //插入
    }
    else a[i] = x;
 }

int main()
{
    scanf("%d%d", &n, &a[0]);
    for (i = 1; i < n; i ++)  //从第 2个数开始
    {
        scanf("%d", &m);
        insert(m);
     }
    for (i = 0; i < n; i ++) printf("%d ", a[i]);
    return 0;
}
```

二维数组

- 二维数组的输入、输出、访问，需要借助二层循环

```
int a[10][2];

for (int i = 0; i < 10; i ++)
    for (int j = 0; j < 2; j ++)
        scanf("%d", &a[i][j]);
```

a[0][0]	a[1][0]	a[2][0]	a[3][0]	a[4][0]	a[5][0]	a[6][0]	a[7][0]	a[8][0]	a[9][0]
a[0][1]	a[1][1]	a[2][1]	a[3][1]	a[4][1]	a[5][1]	a[6][1]	a[7][1]	a[8][1]	a[9][1]

二维数组

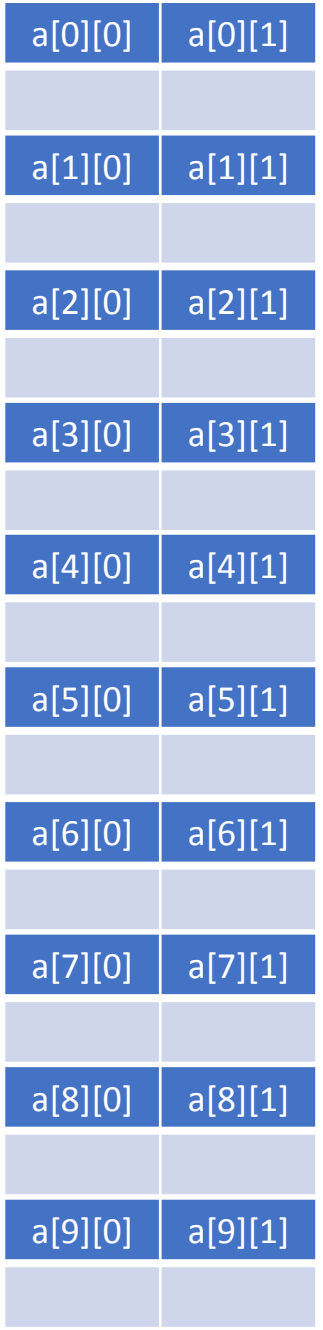
- 二维数组的输入、输出、访问，需要借助二层循环

```
int a[10][2];

for (int i = 0; i < 10; i ++)
    for (int j = 0; j < 2; j ++)
        scanf("%d", &a[i][j]);
```

- 更推荐定义为a[10][2]，而不是a[2][10]
- （外层）i 为横行，（内层） j 为纵列

√



二维数组

- 如果要把二维数组输出成“矩阵”状，还需要活用printf的输出格式

```
for (i = 0; i < 10; i ++)
{
    for (j = 0; j < 2; j ++)
        printf("%d ", a[i][j]);
    printf("\n");
}
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

a[0][0] : 1 a[0][1] : 2
a[1][0] : 3 a[1][1] : 4
a[2][0] : 5 a[2][1] : 6
a[3][0] : 7 a[3][1] : 8
a[4][0] : 9 a[4][1] : 10
a[5][0] : 11 a[5][1] : 12
a[6][0] : 13 a[6][1] : 14
a[7][0] : 15 a[7][1] : 16
a[8][0] : 17 a[8][1] : 18
a[9][0] : 19 a[9][1] : 20
```

挑战

- 把二维数组中每行的最小值输出（n,m≤100）

Sample input	Sample output
3 4 //n行m列 1 2 9 5 3 7 4 6 8 0 1 2	1 3 0

应战


```cpp
#include<bits/stdc++.h>
using namespace std;
int a[110][110], n, m;
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i ++)
    {
        int ans = 0x3f3f3f3f;
        for (int j = 0; j < m; j ++)
        {
            scanf("%d", &a[i][j]);
            ans = min(a[i][j], ans);
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

继续挑战

- 把二维数组中同时满足行上最小、列上最大的数输出，如果没有则输出“-1”（n,m≤100）

Sample input	Sample output
3 4 //n行m列 7 1 9 5 3 2 4 6 8 0 1 2	2

分析

Sample input	Sample output
3 4 //n行m列 7 1 9 5 3 2 4 6 8 0 1 2	2

- 我们首先在行上扫一遍，找出最小的，并记录其列标
- 在该列上扫一遍，若该数是最大的，就输出
- 若不是，则换到下一行
- 直到最后一行也没找到，则输出-1

参考代码

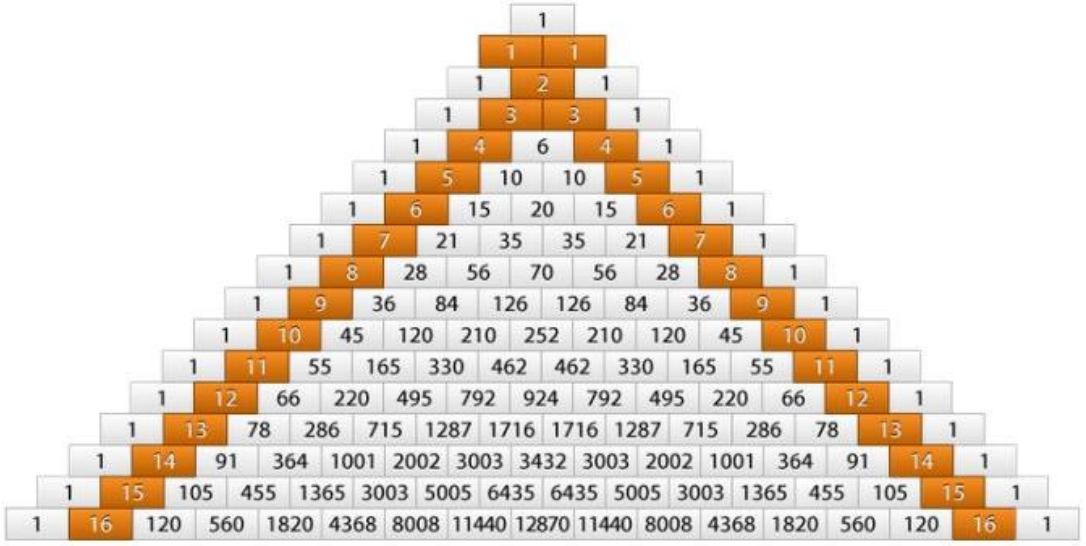

```
#include<bits/stdc++.h>
using namespace std;
int a[110][110], n, m, i, j, k;
int main()
{
    scanf("%d%d", &n, &m);
    for (i = 0; i < n; i ++)
        for (j = 0; j < m; j ++)
            scanf("%d", &a[i][j]);
    for (i = 0; i < n; i ++)
    {
        int minx = 0x3f3f3f3f, maxx = 0, pos;
        for (j = 0; j < m; j ++)
            if (a[i][j] < minx)
                minx = a[i][j], pos = j;  //在该行上找最小值并记录列标
        for (k = 0; k < n; k ++)   //在该列上扫一遍找最大值
            maxx = max(a[k][pos], maxx);
        if (minx == maxx)
            {printf("%d", minx); return 0;}
    }
    printf("-1");
    return 0;
}
```

杨辉三角

- 最早由南宋数学家杨辉在十三世纪发现。近400年后法国数学家帕斯卡也独立发现，西方数学界亦强行称之为帕斯卡三角
- 杨辉三角和二项展开式系数密切相关，比如：

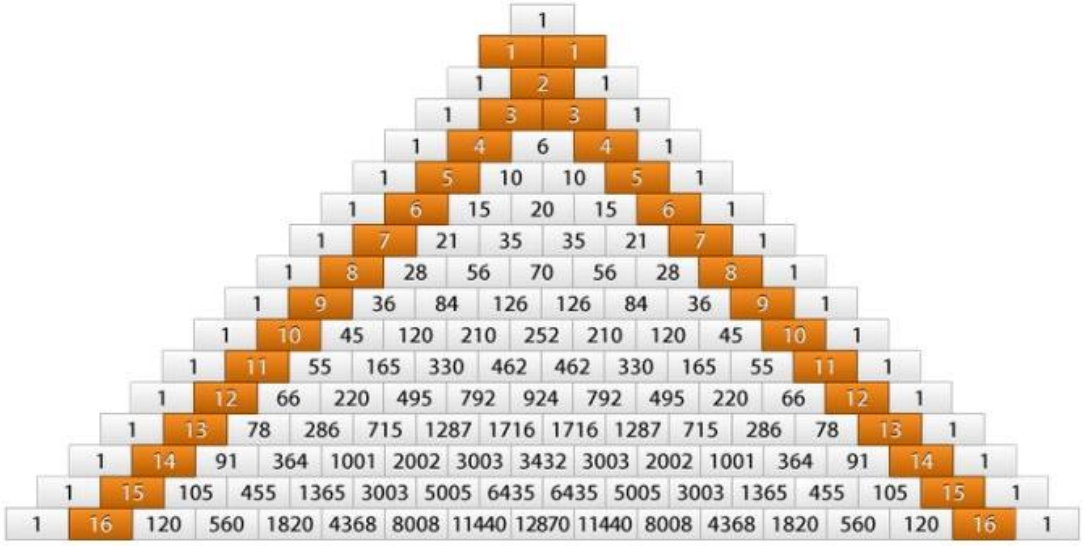
$(a+b)^2=a^2+2ab+b^2$

$(a+b)^3=a^3+3a^2b+3ab^2+b^3$



杨辉三角

- 现在要求输出杨辉三角前n行（n≤50）



分析

- 杨辉三角的经典递推关系：

f[x][1]=1

f[x][y]=f[x-1][y-1]+f[x-1][y]（x代表行、y代表列）

1	4	6
	5	10

```
10
  1
  1   1
  1   2   1
  1   3   3   1
  1   4   6   4   1
  1   5  10  10   5   1
  1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
  1   8  28  56  70  56  28   8   1
  1   9  36  84 126 126  84  36   9   1

--------------------------------
```

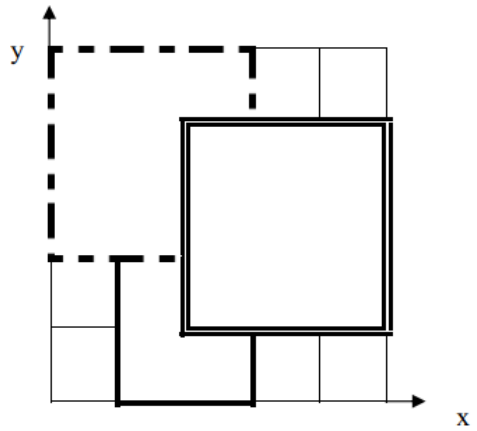
参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
long long a[100][100];
int n, i, j;
int main()
{
    scanf("%d", &n);
    for (i = 1; i <= n; i ++) a[i][1] = 1;      //初始化第一列
    printf("1\n");   //第一行无法递推出来，单独输出
    for (i = 2; i <= n; i ++)
    {
        for (j = 1; j <= i; j ++)
        {
            a[i][j] = a[i-1][j] + a[i-1][j-1];
            printf("%lld ", a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

铺地毯

- n 张编号为1~n（n≤10,000） 的地毯按照编号顺序铺在一块矩形区域上，后铺的地毯会覆盖在已经铺好的地毯之上
- 现在想问覆盖某个点最上面那张地毯的编号（注意地毯边沿也算被覆盖），如果该点恰好没有地毯覆盖则输出 -1

Sample input	Sample output
3 //n 1 0 2 3 //地毯左下角坐标及长度和宽度,下同 0 2 3 3 2 1 3 3 2 2 //被询问点的坐标	3



分析

- 我们首先遇到的第一个问题，就是这些地毯的数据以何种形式保存下来？

- 每张地毯有4个数据，我们可以开一个10000×4的二维数组

```
int c[10010][4];
```

分析

- 接下来的事情，就是判断被询问的点，有没有被地毯覆盖：如果有，直接输出并结束；如果没有，就一直循环下去，直到所有地毯的数据被扫一遍

- 很明显要从后往前逆序，因为问的是最上面那张地毯，也就是靠后被读入的地毯

课外加练

- luogu 2141　　珠心算测验
- luogu 1003　　铺地毯
- luogu 2239　　螺旋矩阵
- luogu 2615　　神奇的幻方
- luogu 2670　　扫雷游戏
- luogu 2038　　无线网络发射器选址