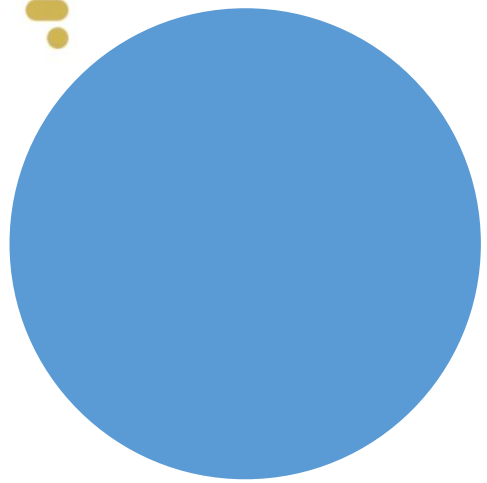


马上开始

35tang-C++竞赛系列四阶课程





《 35tang-C++竞赛系列四阶课程 》

几个数据结构的默认排序顺序

`priority_queue` : 从大到小排列

- 修改为从小到大:
- `priority_queue<int, vector<int>, greater<int> > myq;`

`map<key,value>`: 按照key从小到大排序

- 修改为从大到小
- `map<string, int, greater<string> >`

`set`: 从小到大

- 修改为从大到小
- `set <int, greater<int> >`

如果上述结构里面放的是pair类型?

如果放的是结构体类型, 自定义排序

前缀和

- 就是序列中一个数和前面所有数的和
- 一个序列，计算前n项和的话，不需要每一次都从第一项加到第n项，如果前面已经加了，只要加第n项就好。
- 例如：
- 给定整数数组有n个元素： {1,20,3,41,5,61,7,81,9,10}
- 如果要计算前i（i在1到n之间）项的和依次输出？

```
for (int i=1;i<=n;i++){  
    sum=0;  
    for (int j=1;j<=i;j++) sum+=a[j];  
    cout<<sum<<" ";  
}
```

时间复杂度？ $O(n*n)$, 有没有更快的方法？

如果数组元素从0开始，可以这么做么？

```
for (int i=0;i<n;i++) sum[i]=sum[i-1]+a[i];
```

```
cin>>n;
```

```
for (int i=1;i<=n;i++) cin>>a[i];
```

```
for (int i=1;i<=n;i++) sum[i]=sum[i-1]+a[i]; //sum[0]初始化为0
```

```
for (int i=1;i<=n;i++) cout<<sum[i]<<" ";
```

```
cout<<endl;
```

i	1	2	3	4	5	6	7	8	9	10
a[i]	1	20	3	41	5	61	7	81	9	10
sum[i]	1	21	24	65	70	131	138	219	228	238

a数组第5,6,7三个元素的和: $\text{sum}[7]-\text{sum}[5]+a[5]=73$



后缀和?

最大的作用把时间复杂度从 $O(n)$ 减为 $O(1)$,
或者说减少一次循环

进一步：也可以用来计算任意两项之间的和？

```
cout<<sum[y]-sum[x]+a[x]<<endl;  
cout<<sum[y]-sum[x-1] <<endl;
```



思考：如果没有前缀和，让你计算任意两项（第 x 项和第 y 项）之间的数据和，每一次是不是都要遍历一次？现在有了前缀和，无论 y 和 x 是多少，求每一个 x ， y 之间的数的和都是 $O(1)$ 的时间复杂度，而不是 $O(n)$

例：连续子序列和

给出一个整数序列和一个整数 m ，求其中和为 m 的连续子序列和的个数。

输入格式：

第一行2个整数 n 和 m ， n 表示整数序列长度， $1 \leq n \leq 10^5$ ， m 表示要找的和。 $-10^5 \leq m \leq 10^5$

第二行空格分离的 n 个整数，每一个整数 $\geq -2^{31}$ 并且 $\leq 2^{31} - 1$

输出格式：

一个整数，表示有多少个连续子序列的和为 m 。

示例输入：

5 8

-2 10 -5 3 0

示例输出：

3

初步分析

- 首先对数据类型做判断，显然超过 10^9 ，和需要用long long。
- 有点类似求最大子序列和的题目，但是这里不是找最大，而是找所有和等于m的连续子序列。但是方法上应该有类似的地方。
- 核心思路应该是枚举，因为问的连续子序列和有多少等于m，无论用什么方法，对于每一个区间的和总是要算出来的。不同于前面求最大，我们可以用 $O(n)$ 解决，因为求最大只要维持一个当前区间和就可以（小了扔掉），在这里 $O(n)$ 的算法估计不行，因为小也不能扔掉。

方法1： 时间复杂度 $O(n^3)$

- 最容易想到的算法，实在不行得一些分数
- 暴力枚举所有可能的区间并计算区间和
 - 双循环，分别枚举开始位置和结束位置。
 - 里面嵌套一层循环，从开始位置到结束位置累加求和，然后判断这个区间的和是否等于m
 - $O(n^3)$

```
for (int i=1;i<=n;i++)//区间左
for (int j=i;j<=n;j++) //区间右
//上面双循环枚举所有区间
{
    //累加区间内所有元素
    long long suma=0;
    for(int k=i;k<=j;k++)
        suma+=a[k];
    if (suma==m) cnt++;
}
cout<<cnt<<endl;
```

方法2：时间复杂度 $O(n^2)$

- 枚举开始位置，移动结束位置，累加
- 有没有发现方法一中的k循环有一些重复计算。如果我们枚举每一个区间的开始位置，那么由于是连续子序列和，是不是可以让移动位置++，逐步扩大当前区间的右端点同时累加这个新的右端点的值到区间内，这样在枚举右端点的同时就可以计算出来当前区间的和了。

```
for (int i=1;i<=n;i++)//枚举区间开始位置
{
    long long suma=0;
    for(int j=i;j<=n;j++)
        //移动区间结束位置，累加
        {
            suma+=a[j];
            if (suma==m) cnt++;
        }
}
cout<<cnt<<endl;
```

方法3：前缀和 $O(n^2)$

- 方法1中双循环枚举所有区间，再来一次循环累加这个区间内的所有数。里面的这个循环要求和，不正好是前缀和的概念？
- 如果对a数组（第一个数放在a[1]）计算出来一个前缀和数组presum[]，那么第j项和第i项之间的区间和就是：
 $\text{presum}[j] - \text{presum}[i] + a[i]$ ，双循环枚举所有的区间开始i和结束j，每一次计算就好了。相比方法1省去了一次循环。效率和方法2类似。
- $\text{presum}[j] - \text{presum}[i] + a[i]$ 也可以修改为 $\text{presum}[j] - \text{presum}[i-1]$ ，但是这时候需要保证presum[0]是0，因为当i是1的时候会访问presum[0]。

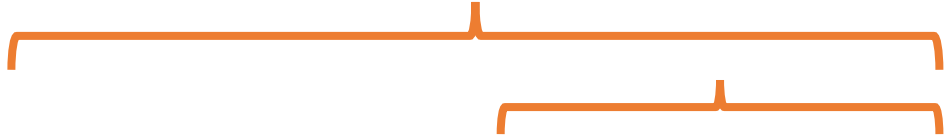
```
for (int i=1;i<=n;i++)//区间左
for (int j=i;j<=n;j++) //区间右
//上面双循环枚举所有区间
{
    //用前缀和直接计算区间[i,j]的和

    if (presum[j]-presum[i]+a[i]==m)
        cnt ++;
}

cout<<cnt<<endl;
```

方法4：方法3的优化 $O(n\log n)$

根据方法2的思路来思考一下方法3。下面序列找 $m=8$ 的区间和。



位置	0	1	2	3	4	5
a[]	0	-2	2	10	-5	3
presum[]	0	-2	0	10	5	8

换个角度，以a[5]为结束的区间为什么有两个和为8的区间？因为前面presum[0]和presum[2]都是0，而presum[5]是8。
 $8-0==8$

我们知道区间[3,5]和是8.因为 $a[3]+a[4]+a[5]==8$, 或者用presum[],
 $presume[5]-presume[2]==8$

我们还知道区间[1,5]和也是8.因为 $a[1]+a[2]+a[3]+a[4]+a[5]==8$, 或者用presum[],
 $presume[5]-presume[0]==8$

算法

- 如果我们对于每一个区间结束位置 j 来看：就是要去找 j 前面的区间开始位置 i 有多少个满足 $\text{presum}[j] - \text{presum}[i-1] == m$ ，也就是满足 $\text{presum}[i-1] == \text{presum}[j] - m$ ； i 最大是 j ，最小是1，那么就是要找 $\text{presum}[0]$ 到 $\text{presum}[j-1]$ 出现了多少个值 $== \text{presum}[j] - m$ 。
- 如果我们 j 从1到 n 开始循环，累加 presum （不需要数组，累加器变量就可以）为当前从第一项到第 j 项的和。把出现过的所有不同的和以及他们的个数保存起来（flag计数），当找到第 j 项的时候，我们只需要去flag数组查找 $m - \text{presum}$ 出现了多少个不就ok了？这样只需要一次循环，时间复杂度 $O(n)$ 。
- 但是，遗憾的是flag计数数组对数的大小限制太大，这里可能的出现的和很大，放不下，怎么办？替代数据结构map，时间复杂度 $O(\log n)$ 。
- 当然`unordered_map`更好，和flag计数一样是 $O(n)$ ，但是需要C++ 11的支持，一般来说map就够了。
- 实际上数组都不需要读一个处理一个。

```
cin>>n>>m;
int a;
int cnt=0;
long long presum=0;
mym[0]++;//注意最前面要有一个0，因为我们前面枚举实际上是 区间开始位置，是 i-1，不是 i
for (int j=1;j<=n;j++)
{
    cin>>a;
    presum+=a;
    mym[presum]++;
    cnt+=mym[presum-m];
}

cout<<cnt<<endl;
```

方法5： 区间动态规划 $O(n^2)$

- 区间动态规划
- 每一个区间 $[i,j]$ 的和就是区间 $[i-1,j]$ 的和+第 i 个元素。区间动态规划的思路，读入数组到 $a[]$ ，那么 $dp[i][j]$ 表示区间 $[i,j]$ 的和，可以得到 $dp[i][j]=dp[i+1][j]+a[i]$.

```
int cnt=0;
for(int len=1;len<=n;len++)//枚举长度
for(int i=1; i+len-1<=n;i++)//枚举区间左端点
{
    int j=i+len-1;//右端点
    dp[i][j]=dp[i+1][j]+a[i];
    if (dp[i][j]==m) cnt++;
}
cout<<cnt<<endl;
```


方法5： 优化

- 上述dp数组太大了，实际上放不下。
- 减维度，dp修改为一维数组： $dp[j]=dp[j]+a[i]$.
这里 $dp[i+1][j]$ 缩到了 $dp[j]$ ，因为我们在计算当前i的所有 $dp[j]$ 的时候，旧的 $dp[j]$ 实际上放的是上一次i+1循环时候算出来的 $dp[j]$ 。为了实现这个目的，我们需要先循环右端点j，然后循环左端点i，而且i一定是从大到小倒过来循环，这样才能保证 $dp[i+1][j]$ 比 $dp[i][j]$ 先计算出来的。

```
int cnt=0;
for(int j=1;j<=n;j++)//枚举长度
for(int i=j; i>=1;i--)//枚举区间左端点
{
    dp[j]=dp[j]+a[i];
    if (dp[j]==m) cnt++;
}
cout<<cnt<<endl;
```



最大子矩阵

输入一个 $N \times N$ ($N \leq 100$) 的矩阵，找出在矩阵中，所有元素加起来之和最大的子矩阵。

输入：

4

0 -2 -7 0

9 2 -6 2

-4 1 -4 1

-1 8 0 -2

输出：15，因为最大子矩阵是

9 2

-4 1

-1 8

要求使用动态规划的方法，时间复杂度 $O(n^3)$

先来看看简单的情况

- 如果题目修改成所有的子矩阵都是第1行第1列为左上角。就是说不会去比较下图的黄色部分类似子矩阵。

grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2

一开始的想法

- 右面遍历行不行？maxsum是最终最大的结果。

grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2

```
for (int row=1;row<=n;row++)  
for (int col=1;col<=n;col++)  
{  
    sum+=grid[row][col] ;  
    maxsum=max(maxsum,sum);  
}
```

不行！ 比如row=2; col=3; 的时候计算的sum累加的是左面黄色部分， 不是个矩阵。

所以，要找的是右下的行列确定的子矩阵所有元素的和的最大

grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2

```
for (int row=1;row<=n;row++)
for (int col=1;col<=n;col++)
{
    int sum=0;
    for (int i=1;i<=row;i++)
    for (int j=1;j<=col;j++)
        sum+=grid[i][j]    ;
    maxsum=max(maxsum,sum);
}
```

可以，row和col相当于遍历了所有的可能的子矩阵的右下角，内部的i和j循环是遍历并且累加该子矩阵的所有元素。

问题是4重循环， $O(n^4)$ ，怎么办？看的出来，计算sum的时候肯定重复计算了很多，能不能动态规划？找一下状态之间的关系。

找规律

- 如果遍历的顺序是row和col从小到大，那么如果要找图中绿色部分的子矩阵，他和前面计算过的子矩阵有什么关系？

```
maxsum=grid[1][1];//打擂法初始化，取最小矩阵初始化
for (int row=1;row<=n;row++)
for (int col=1;col<=n;col++){
    dp[row][col]=dp[row][col-1];
    for (int k=1;k<=row;k++)
        dp[row][col]+=grid[k][col];
    maxsum=max(maxsum,dp[row][col]);
}
```

grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2



grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2

黄色部分的子矩阵的和加上蓝色的值就等于左面绿色的。这就是动态规划，如果我们假设 $dp[i][j]$ 表示右下角行是 i 列是 j 的子矩阵的所有元素的和。就有：

$dp[i][j]=dp[i][j-1]+sum(grid[k][j])$ (k 从1到 i)

也就是 (i,j) 为右下角的子矩阵的元素和= $(i,j-1)$ 为右下角的子矩阵的元素和加上第 j 列从1到 i 行的所有元素的值。



前缀和

前缀和后缀和，就是提前做数据准备，使得后面计算的时候少走一次循环

再优化

```
for (int k=1;k<=row;k++)
```

```
    dp[row][col]+=grid[k][col];
```

计算蓝色这一列的值，只看这一列，前缀和，如果把这一列的值提前计算出来？

grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2

```
maxsum=grid[1][1];//打擂法初始化，取最小矩阵初始化
```

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

```
for (int col=1;col<=n;col++){
```

```
    dp[row][col]=dp[row][col-1];
```

```
    /*for (int k=1;k<=row;k++)
```

```
        dp[row][col]+=grid[k][col]; */
```

```
    dp[row][col]+= presum[row][col];
```

```
    maxsum=max(maxsum,dp[row][col]);
```

```
}
```

由于是2维的，降维。把每一列从第1行道第n行看作一个序列，这一列用前缀和。也就是提前计算出presum[row][col] 对每一个col遍历计算每一行的累加。presum[row][col] 表示当前第col列0到row行的所有数字的和。

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

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

```
    presum[row][col]=presum[row-1][col]+grid[row][col]
```


再再优化：压缩

看下图如果只对row做循环，比如row=3的时候，颜色不同的四列的和都在presum[3][]里面，把这4个和看成一个序列，那么现在是不是很像最大子序列和的求和问题，如右图，现在求出前2项和最大为6，其实就是说前4行组成的子矩阵里面前4行2列的和最大。

grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2

```
maxsum=grid[1][1];//打擂法初始化，取最小矩阵初始化
for (int row=1;row<=n;row++)
{
    int sum=0;
    for (int col=1;col<=n;col++)
    {
        if(sum<0) sum==0;
        sum+= presum[row][col];
        maxsum=max(maxsum,sum);
    }
}
```

col	1	2	3	4
presum[3][]	5	1	-7	3
sum	5	6	0	3

回到原题

回到原题，子矩阵的左上角位置不固定，看下面有颜色的部分，如果我们把开始行和结束的行定了。能不能算出在这两个行之间所有子矩阵的最大值？看下面图中的四个有颜色列值，每一列的和根据前缀和都可以计算出来。和前面唯一的区别就是要多一个循环去枚举子矩阵的开始行而已。

grid	1	2	3	4
1	0	-2	7	0
2	9	2	-10	2
3	-4	1	-4	1
4	-1	8	0	-2

比如黄色部分第2列的和就是 $\text{presum}[3][2] - \text{presume}[1][2]$ ，也就是说如果子矩阵的开始行和终止行是 startrow 和 endrow ，第 col 列的和就是 $\text{presum}[\text{endrow}][\text{col}] - \text{presum}[\text{startrow} - 1][\text{col}]$ ；

有了这个，这是不是就变成了连续最大子序列和，只不过这一次不固定从第1个位置开始？

记住：开始和结束行已经通过外面的双循环固定了。

这时候4列的和分别是5，3，-14，3.按照最大子序列算法，累加 sum ，如果 sum 是负的设置0，比如第1列的时候是5，第2列 sum 是 $5 + 3 = 8$ ，第3列是 $-13 + 8 = -5$ ，第4列就是 $0 + 3 = 3$. 这四个 sum 的最大就是第2行和第3行组成的所有子矩阵的和里面最大的。

col	1	2	3	4
$\text{presum}[3][\text{col}] - \text{presume}[2][\text{col}]$	5	3	-14	3
sum	5	8	0	3

```

cin>>n;
for (int i=1;i<=n;i++)
for (int j=1;j<=n;j++)
    cin>>grid[i][j];
//得到presum--前缀和--前面所有行的当前列的值相加
for (int row=1;row<=n;row++) //行
for (int col=1;col<=n;col++) //列
    presum[row][col]=presum[row-1][col]+grid[row][col];
int maxsum=grid[1][1];//打擂法初始化，取左上角一个元素的矩阵作为初始值
for (int startrow=1;startrow<=n;startrow++) //子矩阵开始行
for (int endrow=startrow;endrow<=n;endrow++) //子矩阵结束行
{
    //最大子序列和的方式找列从1到n的连续最大和
    int sum=0;
    for (int col=1;col<=n;col++) {
        if (sum<0) sum=0;
        sum+=presum[endrow][col]-presum[startrow-1][col] ;
        if (sum>maxsum) maxsum=sum;
    }
}

```

grid	1	2	3	4	
1	0	-2	7	0	
2	9	2	-10	2	← startrow
3	-4	1	-4	1	← endrow
4	-1	8	0	-2	

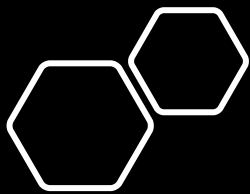
通过presum可以直接计算出某一列startrow到endrow之间的和，就相当于把startrow到endrow之间的每一列压缩为一个数字，如下，对于startrow到endrow之间的所有子矩阵的计算就转换成了对这些和组成的数字序列求最大子序列和的问题

grid	1	2	3	4
1	0	-2	7	0
2	5	5	-14	2
3				
4	-1	8	0	-2

下节课预习

- 不需要做，争取理解题目，思考一下可能的算法
- USACO 2013 November Contest, Silver Crowded Cows
 - <https://www.luogu.com.cn/problem/P3088>





作业1

水壶

- <https://www.luogu.com.cn/problem/P6568>

挑战选做作业2

数字和

number.cpp

- 给出 $n(1 \leq n \leq 1000000)$ 个自然数。从第一个数开始分组，且每组必须是连续的一段数，要求每组和相等，问每组和最小可以是多少。（当然这些数一定可以被分组，大不了直接分成一组。）
- 输入:第一行为一个数 n ,第二行为 n 个整数（每个数均小于等于1000大于0），两个数间用空格隔开。
- 输出:一行，最小的和
- 样例输入

6


2 5 1 3 3 7

- 样例输出

7

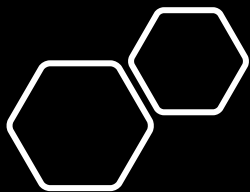
样例说明:

$2+5=7$ $1+3+3=7$ $7=7$ 分成3段连续的数的和，都是7



作业2提示

- 暴力枚举。
- 枚举哪些情况？ 分段的和分别为前1个， 前2个， ...前n个和？
- 是否可以用前缀和提前把前i个和都计算出来 ($1 \leq i \leq n$)



超强挑战选 做作业3

NOIP2015推销员

<https://www.luogu.com.cn/problem/P2672>

作业3提示

- 模拟。
- 找规律，从示例数据不难发现，这是一个贪心算法。当我们第 x 次找到一个最大疲劳值 $value$ 以及当前最大的位置，下一次 $x+1$ ，找的一定是在 x 次的结果上添加一个可以增加疲劳值最大的（因为左右两面的计算疲劳值增加量的算法不同）。找的时候不是去左面走过的找就是去右面找没有找过的找，二选一。
- 问题是每一次怎么去找下一个？遍历会爆掉。对于左面的，我们每一次都是找一个最大疲劳值，用过了就删除（走过的人家不能再走），而且这个队列是变化的，可以用优先队列。对于右面的，我们可以用后缀和的方法，因为每一次我们都是找最当前右面的后面最大的一个（不用删除，也不用变化）。

由易到难，思维体系训练
实战结合，创新协作培养
兴趣导向，未来职业引领

<https://www.35tang.com>



扫码关注公众号

<https://www.三五堂.com>



添加辅导老师