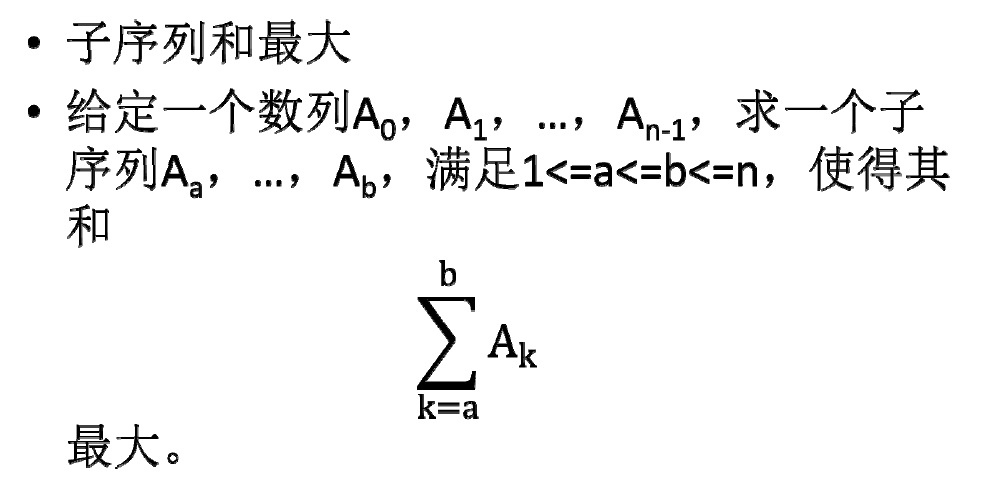
1. **子序列和最大问题：**



注意，有一点题目没有说到，就是本问题还要求a到b是连续的序列，否则将序列中所有正数相加就能得到答案。

朴素解法：维护一个历史最优解Smax和当前和S，然后从头遍历序列，初始S=Smax=0。当发现S比Smax大时，更新Smax。另外，当S小于0时，S赋值为0。否则S=S+ai。

要输出下标时，可再维护两个下标，记录Smax对应的起始下标和终止下标。

比如一个序列为10、5、-7、10、3、-30、25

则S依次为10、15、8、18、21、0、25

所以最终结果为25。

本方法的思路就是，如果当前和大于0，那么就比从ai重新开始序列要好，即ai应该连接到之前的序列中。

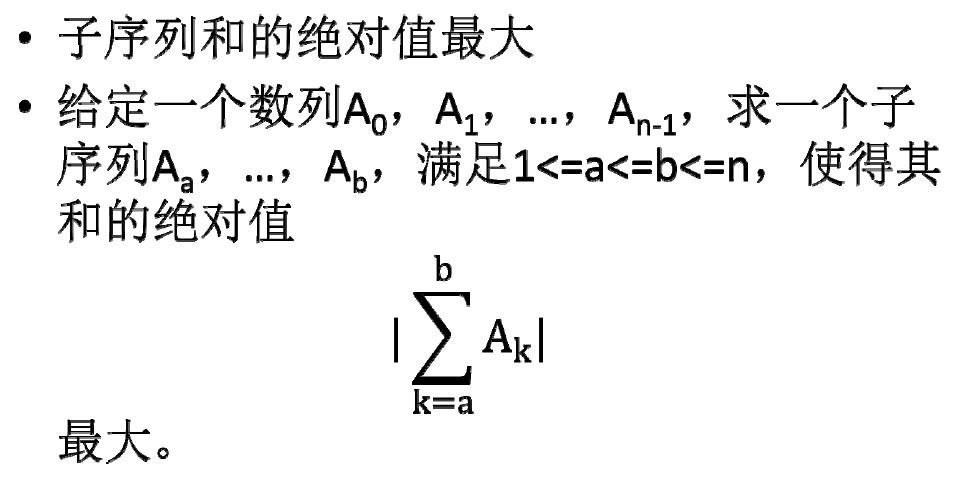
本题时间复杂度O(n)、空间复杂度为O(1)。

将上述解法写成动归形式：

f(i)=max{ai, f(i-1)+ai}=ai+max{0, f(i-1)}

f(i)表示，以元素i结尾的子序列，和的最大值。

1. **子序列和的绝对值最大：**



注意不是子序列绝对值的和，仍要求a到b连续。

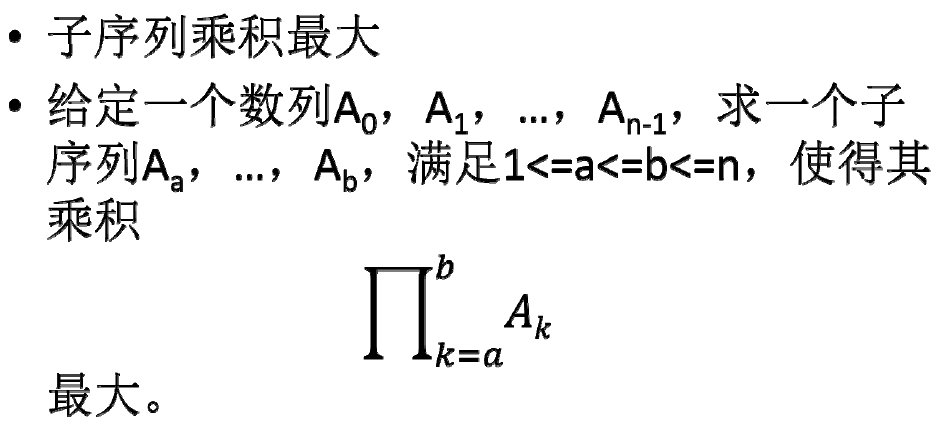
解法1：计算一次最大和，一次最小和，比较两个和的绝对值，取绝对值大的作为结果。计算最小的转移方程：

f(i)=ai+min{0,f(i-1)}

解法2：对原序列取反后，再做一次求最大和，这样可以少写一个函数。

证明正确性：假设|ai+ai+1+…+aj|是最优解，S= ai+ai+1+…+aj, 则一定有Smin<=S<=Smax，所以如果S为正，则|S|<=|Smax|，如果S为负，则-|Smin|>=-S，即|Smin|>=|S|。所以最优解一定是Smin或Smax中的一个。

1. **子序列乘积最大：**



我们先考虑一个简单的问题，如果该数组的数都是正数。

显然，如果数都大于1，则所有数相乘即可。但如果有的数小于1，就需要选择了：

f(j)=ai\*max{1, f(j-1)}=max{aj, aj\*f(j-1)}

注意不是f(j)=max{1,aj\*f(j-1)}，因为当序列所有数都小于1时，该函数最终输出1，是错误的。

当然，如果都是正数还存在一种解法，即取log，这样小于1的数会变负，转换成了求最大和问题。

如果有负数呢？

显然，由于受到符号的影响，当前最大的积在乘一个负数之后就变得最小，同样，当前最小的积，在乘一个负数之后，可能变得最大。所以直观上，我们需要维护两个数组，一个是当前最大积，一个是当前最小积。这样就跟问题2对应起来了。

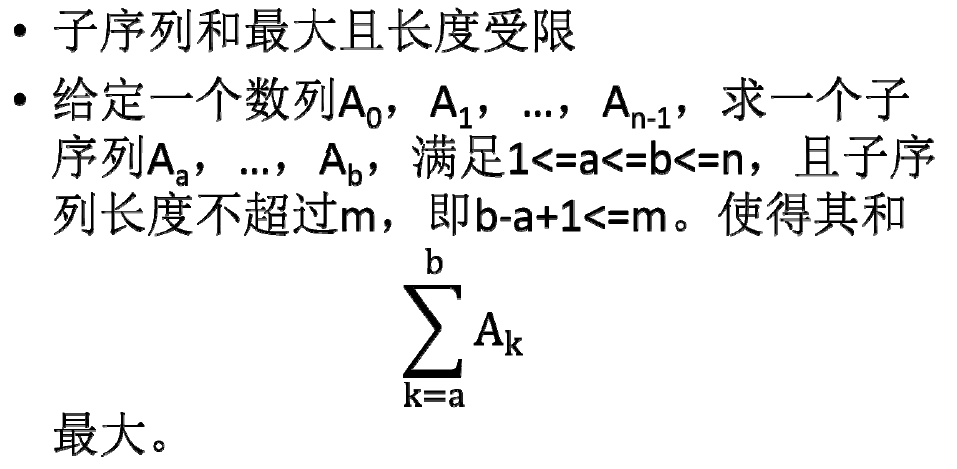
状态转移方程为:

fmin(j)=min{aj, aj\*fmin(j-1), aj\*fmax(j-1)}，

fmax(j)=max{aj, aj\*fmin(j-1), aj\*fmax(j-1)}

即花括号中的内容完全一致，返回时选择fmax中的最大值即可。

1. **子序列和最大但长度受限**



**解法1：**单向暴搜，从ai开始，往后扫描到最多ai+m，记录最大值，即包含ai的长度不超过m的子序列的最大值。对每个ai进行遍历，复杂度为O(n\*m)

**解法2：**计算一个辅助数组S，Si的定义为原序列前i个元素的累加和。之后我们在S上进行扫描，ai对应的包含ai最优的结果为：f(i)=Si-min{Sj-m-1, …, Sj-1}，即想象一个滑动窗口，长度为m，能够自动计算窗口中的最小值，当计算Si对应的最优解时，窗口框住Si前面的m个元素，并找到最小值，然后我们计算Si和这个最小值的差即可。

根据方程也可以看出，该问题本质上已经不能算动归问题了，而是简单的遍历问题，因为下一个状态与上一个状态之间并没有联系。

那么如何高效地实现解法2中的窗口呢？

很自然地，我们想到了堆，因为堆可以在logn时间内找到其中的最小值。但现在的问题是，怎么删除从窗口左侧出去的元素，因为普通的堆仅支持删除堆顶元素，而不能任意删除。

**方法1：**使用支持index的堆，比较复杂而且不在STL库中。

**方法2：**如果我们使用STL库，那么在建堆时，不要仅添加序列元素的值，而是要添加一个结构体，结构体内部是元素值和下标，排序时按照值排序即可。对于超出窗口的元素，我们让它暂留在队列中。在比较时，我们看堆顶元素的下标是否比j-m-1更小，如果是则说明它要出队，则我们现在让它出队，并重新计算最小值和下标是否符合要求。这样暂缓删除的代价仅仅是将复杂度从logm提升到了logn，因为最坏情况就是所有元素最终都在队列中，每次排序的复杂度就是logn，但并不会影响很多。

**方法3：**单调队列。即我们在【Note】10\_课堂补充提到的，双向都可以出队的一种队列。当时该队列被用于快速寻找当前元素aj之前第一个比aj小的元素，现在我们可以用它来更快地找到最小的元素。

我们将方法2中的堆改为一个长度不超过m的队列，如果超过则踢出队首元素。在遍历S数组到Sj并计算f(j)时，我们从队首中取出一个元素Si，则f(j)=Sj-Si，之后将Si与队尾元素比较，如果比队尾元素小，则踢出队尾元素并继续比较，直到遇到比Sj更小的元素或队列为空停止，最后将Sj入队，入队时同样要检查是否超出m，如果超出，将队首元素出队，再将Sj入队。

这种方法的好处在于，不需要存储下标或结构体，同时可以在常数时间内找到最小值。所以，相比之前找第一个比Sj小的元素，直接找最小的元素反而更快。

该算法的复杂度为O(n)，跟m无关，且只需要两次遍历，一次求S数组，一次求f。

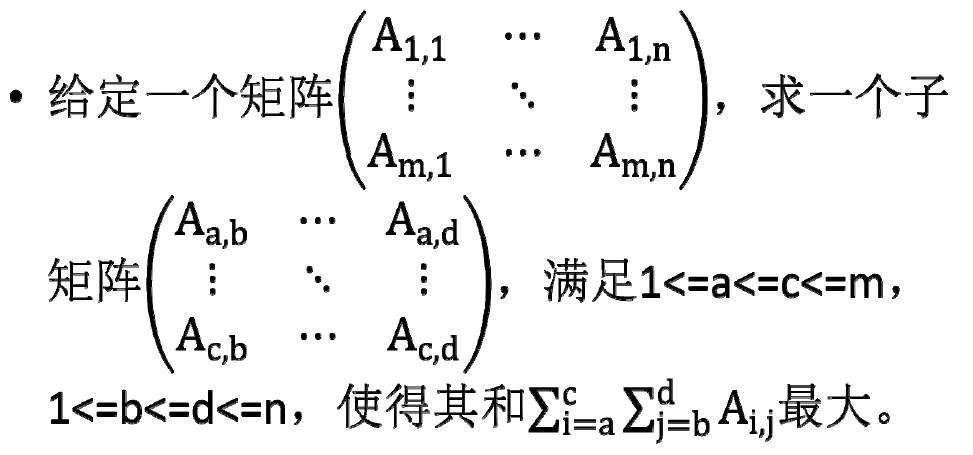
同时该算法思想也可以用于问题1，只不过问题1的解本身已经是O(n)了。

**最后再说一个错误的解法：**那就是仍按照问题1类似的方法计算结果，但同时维护一个序列长度。当序列长度超过m时，选择最前面的一个元素删除。

比如序列为20，-10，18，21，m=3，则一开始当前和为20、10、28，到21时，比较{21,21+28-20}，所以最大和为29，此时删去了第一个元素20。但这里仅删去第一个元素是错误的，因为我们保留-10的原因就是希望能够得到前面的20，如果删去了20，则-10也不应该继续留在数组中，所以有更大的最大和为39。

（不知道可不可以同时将第一个数之后所有的负数删去，同时维护一个序列长度。但似乎比较复杂）

1. **子矩阵最大和问题：**



暴搜复杂度为O(n4)。

与子序列最大和相比，该问题是一个2维动归问题，目标是找一个pair，对应矩阵左上角和右下角，似乎动归的复杂度是O(n2)，但复杂度其实是O(n3)，如果要找的矩阵不是方阵，而是n\*m的，则复杂度根据算法设计不同，可能为O(n\*m2)或O(m\*n2)，所以我们要选择m和n中较小的那条边进行枚举，使总复杂度降低一些。

我们这样定义转移函数：f(i, j, k)表示以第i行第j列到第k列为底，所能得到的子矩阵中的最大值。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

之后，类似地，我们计算一个辅助数组S(i,j,k)，表示第i行，第k列到第k列的累加和。于是我们将一个二维问题转化为了一维问题，即将第j列到第k列之间求和，捏成一个1维数组，再计算一维序列最大和。所以状态函数转移定义为：

f(i, j, k)= S(i, j, k)+ max{0, f(i-1, j, k)}

初始化时，假设第一行上面还有一行，全为0，即所有f(0, j, k)=0。最后返回f(i,j,k)的最大值。

如果矩阵为m\*n，且m<n，则我们将f(i, j, k)定义为以第i列，第j行到第k行为右侧边，子矩阵的最大值。也就是说，我们将维数更小的作为f的后两维枚举，总复杂度更小。

上面的算法空间复杂度为O(n3)，时间复杂度也为O(n3)。

但我们可以将数组S的空间复杂度优化为O(n)，因为在计算f(i, j, k)时，我们仅需要当前行的S值，之前行的S值可以忽略，而之后行的S值也可以到时候再算。而每行的S值可以通过累加和相减来得到，所以我们只需定义一个一行n列数组sum，S(j,k)=sum[k]-sum[j-1]即可。然后计算本行时，先计算本行的S(j, k)，再计算f(i,j,k)。