描述

给定一个整数序列 $S_1, S_2, \cdot, S_n$ ($1 \leqslant n \leqslant 1,000,000, -32768 \leqslant S_i \leqslant 32768$)，定义函数

$sum(i,j) = S_i + ...+ S_j$ ($1 \leqslant i \leqslant j \leqslant n$)。

现给定一个正整数 m，找出 m 对 i 和 j，使得 $sum(i_1, j_1) + sum(i_2, j_2) + ... + sum(i_m, j_m)$ 最大。这就是最大 M 子段和(maximum m segments sum)。

输入每个测试用例由两个正整数 m 和 n 开头，接着是 n 个整数。

输出

每行输出一个最大和。

样例输入

1 3 1 2 3

2 6 -1 4 -2 3 -2 3

样例输出

6

8

分析

设状态为 d[i,j]，表示前 j 项分为 i 段的最大和，且第 i 段必须包含 S[j]，则状态转移方程如下：

$d[i,j] = \max\{d[i,j-1] + S[j], \max\{d[i-1,t] + S[j]\}\}$，其中 $i \leqslant j \leqslant n, i-1 \leqslant t < j$

$target = \max\{d[m,j]\}$，其中 $m \leqslant j \leqslant n$

分为两种情况：

- 情况一，S[j] 包含在第 i 段之中，d[i,j − 1] + S[j]。
- 情况二，S[j]独立划分成为一段，max{d[i − 1,t] + S[j]}。

观察上述两种情况可知 d[i,j]的值只和 d[i,j-1] 和 d[i-1,t] 这两个值相关，因此不需要二维数组，

可以用滚动数组，只需要两个一维数组，用 d[j] 表示现阶段的最大值，即 d[i,j − 1] + S[j]，用 prev[j] 表示上一阶段的最大值，即 max{d[i − 1,t] + S[j]}。

代码

mmss.c

```
#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

/**

* @brief 最大 m 段子序列和

* @param[in] S 数组

* @param[in] n 数组长度

* @param[in] m m 段

* @return 最大 m 段子序列和

*/

int mmss(int S[], intn, int m) {

int max_sum, i, j;
```

```
/* d[i]表示现阶段最大值，prev[i] 表示上阶段最大值 */
/* d[0], prev[0] 未使用*/
int *d = (int*)calloc(n + 1, sizeof(int));
int *prev = (int*)calloc(n + 1, sizeof(int));
S--; // 因为 j 是从 1 开始，而 S 从 0 开始，这里要减一
for (i = 1; i <=m; ++i) {
    max_sum = INT_MIN;
    for (j = i; j <=n; ++j) {
        // 状态转移方程
        if (d[j - 1] <prev[j - 1])
            d[j] = prev[j - 1] +S[j];
        else
            d[j] = d[j - 1] +S[j];
        prev[j - 1] =max_sum; // 存放上阶段最大值
        if (max_sum <d[j])
            max_sum = d[j]; // 更新 max_sum
    }
    prev[j - 1] =max_sum;
}
free(d);
free(prev);
return max_sum;
```

```
}


int main() {

  int n, m, i, *S;
  while(scanf("%d%d", &m, &n) == 2) {
     S = (int*)malloc(sizeof(int) * n);
     for (i = 0; i < n;++i)
        scanf("%d",&S[i]);
     printf("%d\n",mmss(S, n, m));
     free(S);
  }
  return 0;
}
```

最大子段和问题(Maximum Interval Sum)

（有时也称 LIS）

经典的动态规划问题，几乎所有的算法教材都会提到.本文将分析最大子段和问题的几种不同效率的解法，以及最大子段和问题的扩展和运用.

一.问题描述

给定长度为 n 的整数序列，a[1...n]，求[1,n]某个子区间[i , j]使得 a[i]+…+a[j]和最大.或者求出最大的这个和.例如(-2,11,-4,13,-5,2)的最大子段和为 20,所求子区间为[2,4].

二. 问题分析

1.穷举法

穷举应当是每个人都要学会的一种方式，这里实际上是要穷举所有的[1,n]之间的区间，所以我们用两重循环，可以很轻易地做到遍历所有子区间，一个表示起始位置，一个表示终点位置.代码如下：

[cpp] view plain copy

print?

```cpp
int start = 0;//起始位置
int end = 0;    //结束位置
int max = 0;
for(int i = 1; i <= n; ++i)
{
        for(int j = i; j <= n;++j)
        {
                int sum = 0;
                for(int k = i; k <=j; ++k)
                        sum += a[k];
                if(sum > max)
                {
                        start = i;
                        end = j;
                        max = sum;
                }
        }
}
```

这个算法是几乎所有人都能想到的，它所需要的计算时间是 O(n^3).当然，这个代码还可以做点优化，实际上我们并不需要每次都重新从起始位置求和加到终点位置.可以充分利用之前的计算结果.

或者我们换一种穷举思路,对于起点 i，我们遍历所有长度为 1,2,…,n-i+1 的子区间和，以求得和最大的一个.这样也遍历了所有的起点的不同长度的子区间，同时，对于相同起点的不同长度的子区间，可以利用前面的计算结果来计算后面的.

比如，i 为起点长度为 2 的子区间和就等于长度为 1 的子区间的和+a[i+1]即可，这样就省掉了一个循环，计算时间复杂度减少到了 O(n^2).代码如下:

[cpp] view plain copy

print?

```cpp
int start = 0;//起始位置
int end = 0;//结束位置
int max = 0;
for(int i = 1; i <= n; ++i)
{
        int sum = 0;
        for(int j = i; j <= n;++j)
        {
                sum += a[j];
                if(sum > max)
                {
                        start = i;
                        end = j;
                        max = sum;
                }
        }
}
```

2.分治法

求子区间及最大和，从结构上是非常适合分治法的，因为所有子区间[start, end]只可能有以下三种可能性:

- 在[1, n/2]这个区域内
- 在[n/2+1, n]这个区域内
- 起点位于[1,n/2],终点位于[n/2+1,n]内

以上三种情形的最大者，即为所求．前两种情形符合子问题递归特性，所以递归可以求出．对于第三种情形，则需要单独处理．第三种情形必然包括了 n/2 和 n/2+1 两个位置，这样就可以利用第二种穷举的思路求出：

- 以 n/2 为终点，往左移动扩张，求出和最大的一个 left_max
- 以 n/2+1 为起点，往右移动扩张，求出和最大的一个 right_max
- left_max+right_max 是第三种情况可能的最大值

示例：

[cpp] view plain copy

print?

```cpp
int maxInterval(int *a, int left, int right)
 {
     if(right==left)
         return a[left]>0?a[left]:0;

     int center = (left+right)/2;
     //左边区间的最大子段和
     int leftMaxInterval = maxInterval(a,left,center);
     //右边区间的最大子段和
     int rightMaxInterval= maxInterval(a,center+1,right);

     //以下求端点分别位于不同部分的最大子段和

     //center 开始向左移动
     int sum = 0;
     int left_max = 0;
     for(int i = center; i >= left; --i)
     {
         sum += a[i];
         if(sum > left_max)
             left_max = sum;
     }
     //center+1 开始向右移动
     sum = 0;
     int right_max = 0;
     for(int i = center+1; i <= right; ++i)
     {
         sum += a[i];
         if(sum > right_max)
```

```
                right_max = sum;
        }
        int ret = left_max+right_max;
        if(ret < leftMaxInterval)
                ret = leftMaxInterval;
        if(ret < rightMaxInterval)
                ret = rightMaxInterval;
        return ret;
}
```

分治法的难点在于第三种情形的理解，这里应该抓住第三种情形的特点，也就是中间有两个定点，然后分别往两个方向扩张，以遍历所有属于第三种情形的子区间，求的最大的一个，如果要求得具体的区间，稍微对上述代码做点修改即可. 分治法的计算时间复杂度为 O(nlogn).

3.动态规划法

动态规划的基本原理这里不再赘述，主要讨论这个问题的建模过程和子问题结构.时刻记住一个前提，这里是连续的区间

- 令 b[j]表示以位置 j 为终点的所有子区间中和最大的一个
- 子问题:如 j 为终点的最大子区间包含了位置 j-1,则以 j-1 为终点的最大子区间必然包括在其中
- 如果 b[j-1] >0, 那么显然 b[j] = b[j-1] + a[j]，用之前最大的一个加上 a[j]即可，因为 a[j]必须包含
- 如果 b[j-1]<=0,那么 b[j] = a[j] ,因为既然最大，前面的负数必然不能使你更大

对于这种子问题结构和最优化问题的证明，可以参考算法导论上的“剪切法”，即如果不包括子问题的最优解，把你假设的解粘帖上去，会得出子问题的最优化矛盾.**证明如下：**

- 令 a[x,y]表示 a[x]+…+a[y] , y>=x
- 假设以 j 为终点的最大子区间 [s, j] 包含了 j-1 这个位置，以 j-1 为终点的最大子区间[r, j-1]并不包含其中
- 即假设[r,j-1]不是[s,j]的子区间
- 存在 s 使得 a[s, j-1]+a[j]为以 j 为终点的最大子段和,这里的 **r != s**
- 由于[r, j -1]是最优解, 所以 a[s,j-1]<a[r, j-1],所以 a[s,j-1]+a[j]<a[r, j-1]+a[j]
- 与[s,j]为最优解矛盾.

实例：

[cpp] view plain copy

print?

```cpp
int max = 0;
int b[n+1];
int start = 0;
int end = 0;
memset(b,0,n+1);
for(int i = 1; i <= n; ++i)
  {
      if(b[i-1]>0)
      {
          b[i] = b[i-1]+a[i];
      }else{
          b[i] = a[i];
      }
      if(b[i]>max)
          max = b[i];
  }
```