令 p 为 i+1 和 j 之间的任一下标。开始于下标 p 的任意子序列都不大于在下标 i 开始并包含从 a[i]到 a[p-1]的子序列的对应的子序列，因为后面这个子序列不是负的(j 是使得从下标 i 开始其值成为负值的序列的第一个下标)。因此，把 i 推进到 j+1 是没有风险的：我们一个最优解也不会错过。

```java
/**
 * Linear-time maximum contiguous subsequence sum algorithm.
 */
public static int maxSubSum4( int [ ] a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.length; j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }

    return maxSum;
}
```

图 2-8 算法 4

这个算法是许多聪明算法的典型：运行时间是明显的，但正确性则不那么容易看出来。对于这些算法，正式的正确性证明(比上面的分析更正式)几乎总是需要的；然而，即使到那时，许多人仍然还是不信服。此外，许多这类算法需要更有技巧的编程，这导致更长的开发过程。不过当这些算法正常工作时，它们运行得很快，而我们将它们和一个低效(但容易实现)的蛮力算法通过小规模的输入进行比较可以测试到大部分的程序原理。

该算法的一个附带的优点是，它只对数据进行一次扫描，一旦 a[i]被读入并被处理，它就不再需要被记忆。因此，如果数组在磁盘上或通过互联网传送，那么它就可以被按顺序读入，在主存中不必存储数组的任何部分。不仅如此，在任意时刻，算法都能对它已经读入的数据给出子序列问题的正确答案(其他算法不具有这个特性)。具有这种特性的算法叫做**联机算法**(on-line algorithm)。仅需要常量空间并以线性时间运行的联机算法几乎是完美的算法。

2.4.4 运行时间中的对数

分析算法最混乱的方面大概集中在对数上面。我们已经看到，某些分治算法将以 $O(N \log N)$ 时间运行。此外，对数最常出现的规律可概括为下列一般法则：*如果一个算法用常数时间($O(1)$)将问题的大小削减为其一部分(通常是 1/2)，那么该算法就是 $O(\log N)$。另一方面，如果使用常数时间只是把问题减少一个常数的数量(如将问题减少 1)，那么这种算法就是 $O(N)$的。*

显然，只有一些特殊种类的问题才能够呈 $O(\log N)$型。例如，若输入 N 个数，则算法只要把这些数读入就必须耗费 $\Omega(N)$的时间量。因此，当我们谈到这类问题的 $O(\log N)$算法时，通常都是假设输入数据已经提前读入。下面，我们提供具有对数特点的三个例子。

折半查找

第一个例子通常叫做折半查找(binary search)。