我们可以通过撤除一个 for 循环来避免三次的运行时间。不过这不总是可能的，在这种情况下算法中出现大量不必要的计算。纠正这种低效率的改进算法可以通过观察 $\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$ 而看出，因此算法 1 中第 13 行和第 14 行上的计算过分地耗费了。图 2-6 给出了一种改进的算法。算法 2 显然是 $O(N^2)$；对它的分析甚至比前面的分析还简单。

```java
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 */
public static int maxSubSum2( int [ ] a )
{
    int maxSum = 0;

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

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

图 2-6 算法 2

对这个问题有一个递归和相对复杂的 $O(N \log N)$ 解法，我们现在就来描述它。要是真的没出现 $O(N)$(线性的)解法，这个算法就会是体现递归威力的极好的范例了。该方法采用一种“分治(divide-and-conquer)”策略。其想法是把问题分成两个大致相等的子问题，然后递归地对它们求解，这是“分”的部分。“治”阶段将两个子问题的解修补到一起并可能再做些少量的附加工作，最后得到整个问题的解。

在我们的例子中，最大子序列和可能在三处出现。或者整个出现在输入数据的左半部，或者整个出现在右半部，或者跨越输入数据的中部从而位于左右两半部分之中。前两种情况可以递归求解。第三种情况的最大和可以通过求出前半部分(包含前半部分最后一个元素)的最大和以及后半部分(包含后半部分第一个元素)的最大和而得到。此时将这两个和相加。作为一个例子，考虑下列输入：

前半部分				后半部分			
4	-3	5	-2	-1	2	6	-2

其中前半部分的最大子序列和为 6(从元素 A_1 到 A_3)而后半部分的最大子序列和为 8(从元素 A_6 到 A_7)。

前半部分包含其最后一个元素的最大和是 4(从元素 A_1 到 A_4)，而后半部分包含其第一个元素的最大和是 7(从元素 A_5 到 A_7)。因此，横跨这两部分且通过中间的最大和为 4 + 7 = 11(从元