数据结构与算法I 作业14

2019201409 于倬浩

最大子矩阵和问题

给定一个m行n列的整数矩阵A，试求矩阵A的一个子矩阵，使其各元素之和为最大。

下面给出两种做法:

- $\Theta(n^2m^2)$: 朴素的暴力算法。
    - 维护$sum[i][j]=\sum_{a=1}^{i}\sum_{b=1}^{j}A[a][b]$，可以使用`sum[i][j]=sum[i][j-1]+sum[i-1][j]-sum[i-1][j-1]`，以$\Theta(nm)$的代价计算出来。
    - 接下来，将答案`ans`初始化为`-inf`，枚举矩形的左上角和右下角坐标`(x0,y0)`，`(x1,y1)`，对于每一组枚举的点对，当前的子矩阵答案即为`sum[x1][y1]-sum[x1][y0-1]-sum[x0-1][y1]+sum[x0-1][y0-1]`，每次比较当前子矩阵的答案以及`ans`的大小即可，该算法的时间复杂度即为$\Theta(nm*nm)$，正确性来源于枚举了全部有可能成为答案的子矩阵并计算了答案。
- $\Theta(min(n^2m,nm^2))$: 动态规划。
    - 上面的算法实际上还是在暴力枚举两个点对，因此需要枚举四次坐标，导致复杂度较高，因此我们考虑套用低维的动态规划算法，减少枚举的维数。
    - 首先假设$n\le m$，因为当$n>m$时，我们可以直接交换矩阵的行和列，保证$n'\le m'$。
    - 考虑一维的最大子段和算法：显然一维的最大字段和具有最优子结构，因为如果当前的最优解`[L,R]`的`[L,R-1]`不如`[L',R-1]`，那么我们可以直接将子问题的解更换成更优的，从而在当前问题中获得更优秀的解。因此，设`f[i]`表示序列A`[i]`上以`i`结尾的最大子段和，那么有$f[i]=max(0,f[i-1])+A[i]$，最终的答案$ans=max_{i=1}^{n}(f[i])$。
    - 考虑推广一维的算法到矩阵上：首先，我们可以枚举矩形的上下边缘`x`、`y`，即计算所有上边缘为x、下边缘为y的矩形中，矩形内数字之和最大的。那么，现在就可以直接套用一维的情况，设`f[i]`表示上边缘为x、下边缘为y的矩形中，以`i`为右边缘的最大矩阵和，那么有
    $$f[i]=max(0,f[i-1])+(sum[y][i]-sum[x-1][i])-(sum[y][i-1]-sum[x-1][i-1])$$
    其中`sum[i][j]`和第一种做法中的`sum`意义、计算方法相同。
    - 最终答案即为每次算出的`f[i]`中的最大值。因为这种算法实际上是通过枚举矩形的两条边，将问题转化为一维的情况，而一维情况的正确性已经证明，因此这种算法也具有正确性。
    - 时间复杂度$\Theta(min(n^2m,m^2n))$，空间复杂度$\Theta(nm)$。
    - 核心代码（仅展示求最大和的算法，如需输出方案，仅需多维护最优解转移来的下标数组即可）：

```
long long ans = -1e18;
for(int x = 1; x <= n; ++x) {
    for(int y = x; y <= n; ++y) {
        long long f = -1e18; //没有必要保存f[i]，因为每次只会用f[i-1]
        for(int j = 1; j <= m; ++j) {
            f = max(0, f) +
                (sum[y][j] - sum[x - 1][j]) - (sum[y][j - 1] - sum[x - 1][j - 1]);
            ans = max(ans, f);
        }
    }
}
```

最大m子段和问题

给定由n个整数（可能为负整数）组成的序列a1, a2, ..., an，以及一个正整数m，要求确定序列a1, a2, ..., an的m个不相交子段，使这m个子段的总和达到最大。

依旧和上一题一样，首先考虑从朴素的序列子段和推广到目前的问题。首先考虑最优子结构。假设我们目前有“只考虑序列的前i个元素，且分成了j段”的最优答案。如果第i-1个元素被选中，那么i-1和i在同一段内被选中且到i-1为止也分为了j段。那么假设子问题(i-1,j)存在更优秀的解，显然直接替换掉不会影响当前的答案。另一种情况，如果第i-1个元素没有被选中，那么第i个元素单独成一段，前i-1个元素分成了j-1段，如果(i-1,j-1)存在更优的解，依然可以更换局部最优解以改善全局最优解，且不影响分隔的段数，可以保证无后效性。因此，两种情况下，都可以保证最优子结构。

因此，写成状态转移方程，用`f[i][j]`表示以`i`结尾，选中的元素分成了`j`段的最优答案：$f[i][j] = max(f[i-1][j], max_{k=1}^{i-1}(f[k][j-1])) + A[i]$，答案$ans = max_{i=m}^{n}(f[i][m])$。

转移方程中，$max_{k=1}^{i-1}(f[k][j-1])$实际上考虑的是，第i个元素单独开了一段，那么还需枚举上一个被选中的元素k，取j-1段的最大值。即便如此，实际上并不需要真的通过枚举来计算最大值，只需要使用空间换时间的小技巧，在计算每个`f[i][j]`的同时维护`g[i][j]`表示$max_{x=1}^{i}(f[x][j])$，分成`j`段的最大值，这样改进的话，最终的答案$ans = g[n][m]$。

该算法的时间复杂度为$\Theta(nm)$，空间复杂度$\Theta(nm)$，实际上空间复杂度可以通过滚动数组的技巧优化到$\Theta(n)$。

核心代码（仅展示计算最大和的算法，如果需要输出方案，只需多记录`h[i][j]`保存上一个状态即可）：

```
for(int i = 1; i <= n; ++i) {
    for(int j = 1; j <= m; ++j) {
        f[i][j] = f[i - 1][j] + a[i];
        if(i > 2) f[i][j] = max(f[i][j], g[i - 2][j - 1] + a[i]);
        g[i][j] = max(g[i - 1][j], f[i][j]);
    }
}
```