

Lesson2

线性DP

递推方程有比较明显的线性特点

数字三角形

分析思路

- 状态表示: $f(i, j)$
 - 集合: 所有从起点, 走到 (i, j) 的路径
 - 属性: 所有路径权值之和的最大值 (Max)
 - 状态计算: 集合划分
 - 分为两类, 从左上方来和从右上方来
- 状态转移方程: $f[i, j] = \max(f[i-1, j-1] + a[i, j], f[i-1, j] + a[i, j])$

dp下标问题:

- 若涉及到 $f[i-1]$, 则下标一般从 $i = \text{最小值} + 1$ 开始, 并且为 $f[0]$ 设计边界值, 从而保证数组不越界与结果的正确性。
- 若没有涉及到 $i-1$ 下标, 则可以从 $i = \text{最小值}$ 开始循环。

dp时间复杂度: $O(\text{状态} * \text{转移的计算量})$

具体实现: 可以将以下代码进一步降维 (参考01背包), 另外也可以从下往上做

```
1 static int N = 510, INF = 0x3f3f3f3f;
2
3 static int n;
4 static int[][] a = new int[N][N];
5 static int[] f = new int[N];
6
7 public static void main(String[] args) throws Exception {
8     ins.nextToken(); n = (int)ins.nval;
9
10    //初始化f[j]
11    Arrays.fill(f, -INF);
12
13    for (int i=1; i<=n; i++)
14        for (int j=1; j<=i; j++) { ins.nextToken(); a[i][j] =
15            (int)ins.nval; }
16
17    f[1] = 0; //初始化f[1]
18    for (int i=1; i<=n; i++)
19        for (int j=i; j>=1; j--)
```

```

19         f[j] = Math.max(f[j], f[j-1])+a[i][j];
20
21         //找出底层最大的路径
22         int k = 1;
23         for (int i=1; i<=n; i++) {
24             if (f[i] > f[k]) k = i;
25         }
26
27         out.println(f[k]);
28
29         out.flush();
30     }

```

最长上升子序列

分析思路：

- 状态表示：f[i]
 状态维数确定原则：保证答案能依据状态推出来，且维数越少越好（从小往大考虑）
 - 集合：所有以第i个数a[i]结尾的上升子序列的集合
 - 属性：集合中每一个上升子序列长度的最大值
 - 状态计算：集合划分
 - 最后一个数是a[i]已经确定，我们可以用第i-1个数（倒数第二个数）进行分类，即：没有第i-1个数（序列只有一个数a[i]），第i-1个数分别是：a[1], a[2], ..., a[i-1]。其中每一类可能不存在，若a[i-k] >= a[i]，则该类不存在。若某一类是a[j]a[i] (j<i, a[j]<a[i])，则f[i]可以表示为 f[j]+1。
- 因此，状态转移方程为：f[i] = Max(f[j] + 1), 且满足a[j] < a[i], j=0, ..., i-1
- 时间复杂度：O(状态数量 * 每个状态需要的计算次数) = O(n*n) = O(n^2)

具体实现

```

1  static int N = 1010;
2
3  static int n;
4  static int[] a = new int[N], f = new int[N];
5
6  public static void main(String[] args) throws Exception {
7      ins.nextToken(); n = (int)ins.nval;
8
9      for (int i=1; i<=n; i++) { ins.nextToken(); a[i] = (int)ins.nval; }
10
11     for (int i=1; i<=n; i++) {
12         f[i] = 1;    //没有第i-1个数
13         for (int j=1; j<i; j++)
14             if (a[j] < a[i]) f[i] = Math.max(f[i], f[j]+1);
15     }
16
17     //找出答案
18     int ans = 0;
19     for (int i=1; i<=n; i++) ans = Math.max(f[i], ans);

```

```

20
21     out.println(ans);
22
23     out.flush();
24 }

```

如何保存最长序列？ 使用数组存储状态转移的过程

具体实现：

```

1  static int N = 1010;
2
3  static int n;
4  static int[] a = new int[N], f = new int[N], g = new int[N];    //g存储转
    移的过程
5
6  public static void main(String[] args) throws Exception {
7      ins.nextToken(); n = (int)ins.nval;
8
9      for (int i=1; i<=n; i++) { ins.nextToken(); a[i] = (int)ins.nval; }
10
11     for (int i=1; i<=n; i++) {
12         f[i] = 1;    //没有第i-1个数
13         g[i] = 0;    //前驱是0
14         for (int j=1; j<i; j++)
15             if (a[j] < a[i])
16                 if (f[i] < f[j]+1) {
17                     f[i] = f[j]+1;
18                     g[i] = j;
19                 }
20     }
21
22     //找出答案
23     int k = 1;
24     for (int i=1; i<=n; i++)
25         if (f[i] > f[k]) k = i;
26
27     out.println(f[k]);
28
29     //打印路径
30     for (int i=0, len=f[k]; i<len; i++) {
31         out.print(a[k]+" ");
32         k = g[k];
33     }
34
35     out.flush();
36 }

```

最长上升子序列2：如何优化（单调序列二分优化）

优化思路：

- 存储当前数前面每种长度的上升子序列（数组下标表示长度）的结尾值最小是多少

猜想：随着上升子序列长度的增加，结尾最小元素的值一定严格单调递增

证明：反证法，例如：如果长度是6的上升子序列的最小结尾只小于或等于长度是5的上升子序列的最小值结尾，则一定可以从长度6的上升子序列找出长度是5的上升子序列，且该子序列结尾小于上述长度5的上升子序列结尾，故存在矛盾。原结论正确。

- 因此求以 $a[i]$ 结尾的最长上升子序列长度可以转化为，将 $a[i]$ 接到上述最小值序列中最大的且小于 $a[i]$ 的数之后（由于序列单调，可以采用二分），接完之后再更新上述最小值序列。（贪心思想）

二分时间复杂度 $\log n$ ，一共有 n 个数，因此时间复杂度为 $O(n \log n)$ ，满足题目数据要求

具体实现：

```

1  static int N = 100010;
2
3  static int n;
4  static int[] q = new int[N];
5  static int len;
6
7
8  public static void main(String[] args) throws Exception {
9      ins.nextToken(); n = (int)ins.nval;
10
11     q[0] = (int)-2e9;    //设置哨兵
12     while (n-- > 0) {
13         ins.nextToken(); int a = (int)ins.nval;
14
15         //二分左边界
16         int l = 0, r = len;
17         while (l < r) {
18             int mid = l+r+1>>1;
19             if (q[mid] < a) l = mid;
20             else r = mid-1;
21         }
22
23         len = Math.max(len, r+1);
24         q[r+1] = a;
25     }
26
27     out.println(len);
28
29     out.flush();
30 }

```

最长公共子序列

分析思路

- 状态表示： $f[i, j]$
 - 集合：所有在第一个序列的前 i 个字母中出现，且在第二个的前 j 个字母中出现的公共子序列

(一般两个字符串问题求公共..., 可以用i, j分别表示两个字符串)

- 属性: 集合中每一个公共子序列长度的最大值 (Max)
- 状态计算: 集合划分
 - 设a, b为题意中两个字符串, 以a[i]和b[j]是否包含在子序列当中来划分子集合。考虑a[i], b[j], 则选或不选一共有4种情况(00, 01, 10, 11), 以此划分为4个子集 (不重不漏)。
 - 当00 (都不选), 为f[i-1, j-1]; 当11 (都选), 为f[i-1, j-1]+1, 且满足a[i] = b[j]。
 - 当01或者10时, 则并不为f[i-1, j]或者f[i, j-1], 因为f[i-1, j]或者f[i, j-1]所对应的集合中b[j]与a[i]不一定要求在最后一个位置一定出现。但f[i-1, j]一定严格包含01这种情况, 且f[i, j]的集合一定又严格包含f[i-1, j]这种情况。
 - 思路: 虽然f[i-1, j]一定严格包含01这种情况, 但我们仍可用f[i-1, j]来代替01这种情况。使用f[i-1, j]代替01情况的后果是, 可能导致其与f[i-1, j-1]所代表的集合存在重叠的情况。但由于我们只需求出所有子集中的最大值, 而发生重叠并不会影响最大值的计算, 只有遗漏才会导致最大值的变化, 因此这种弱化不重原则的划分方式是合理的。同理我们可以采用f[i, j-1]来代替10这种情况。
 - 优化: f[i-1, j-1]该类可以不用考虑, 因为f[i-1, j-1]对应的集合一定被包含于f[i-1, j]与f[i, j-1]的情况中。所以我们只考虑3种转移情况, 即f[i-1, j], f[i, j-1], f[i-1, j-1]+1
 - 时间复杂度: $O(\text{状态数量} * \text{状态转移的计算次数}) = O((n^2) * 3) = O(n^2)$

具体实现: 此题使用到了 f[i, j-1], 因此不能进行降维操作。

```
1 static int N = 1010;
2
3 static int n, m;
4 static char[] a = new char[N], b = new char[N];
5 static int[][] f = new int[N][N];
6
7 public static void main(String[] args) throws Exception {
8     String[] ss = inb.readLine().split(" ");
9     n = Integer.parseInt(ss[0]); m = Integer.parseInt(ss[1]);
10
11     String tmp = inb.readLine(); tmp = " "+tmp; a = tmp.toCharArray();
12     tmp = inb.readLine(); tmp = " "+tmp; b = tmp.toCharArray();
13
14     for (int i=1; i<=n; i++)
15         for (int j=1; j<=m; j++) {
16             f[i][j] = Math.max(f[i-1][j], f[i][j-1]);
17             if (a[i] == b[j]) f[i][j] = Math.max(f[i][j], f[i-1][j-1]+1);
18         }
19
20     out.println(f[n][m]);
21
22     out.flush();
23 }
```

DP分析 (对暴搜的优化, 用一个数表示一堆东西的某种属性)

- 状态表示: $f[i, j]$
 - 集合: 所有将 $a[1 \sim i]$ 变成 $b[1 \sim j]$ 的操作方式的集合
 - 属性: 所有操作方式的操作次数的最小值
 - 状态计算: 集合划分, 分类方式 (常常考虑最后一步, 倒数第二步...)
 - 考虑最后一步, 有三种操作方式。如果是删除 $a[i]$, 则需要满足 $a[1 \sim (i-1)] = b[1 \sim j]$ (**$f[i-1, j]$ 保证此条件成立**), 所以该类可以表示为 $f[i-1, j] + 1$; 如果是在 $a[i]$ 之后插入一个字符, 则满足插入字符是 $b[j]$, 且 $a[1 \sim i] = b[1 \sim j-1]$ (**$f[i, j-1]$ 保证此条件成立**), 则该类表示为 $f[i, j-1] + 1$; 如果是将 $a[i]$ 修改为 $b[j]$, 若未修改时 $a[i] \neq b[j]$, 则该类可以表示为 $f[i-1, j-1] + 1$, 若 $a[i] = b[j]$, 则该类可表示为 $f[i-1, j-1]$ 。
- 故最终, $f[i, j] = \min(f[i-1, j] + 1, f[i, j-1] + 1, f[i-1, j-1] + 1/0)$ 。
- 时间复杂度: $O(n * n * 3) = O(n^2)$

具体实现:

```
1 static int N = 1010;
2
3 static int n, m;
4 static char[] a = new char[N], b = new char[N];
5 static int[][] f = new int[N][N];
6
7 public static void main(String[] args) throws Exception {
8     n = Integer.parseInt(inb.readLine().split(" ")[0]);
9     String tmp = inb.readLine(); tmp = " "+tmp; a = tmp.toCharArray();
10    m = Integer.parseInt(inb.readLine().split(" ")[0]);
11    tmp = inb.readLine(); tmp = " "+tmp; b = tmp.toCharArray();
12
13    //初始化边界
14    for (int i=1; i<=m; i++) f[0][i] = i;
15    for (int i=1; i<=n; i++) f[i][0] = i;
16
17    for (int i=1; i<=n; i++)
18        for (int j=1; j<=m; j++) {
19            f[i][j] = Math.min(f[i-1][j]+1, f[i][j-1]+1);
20            if (a[i] == b[j]) f[i][j] = Math.min(f[i][j], f[i-1][j-1]);
21            else f[i][j] = Math.min(f[i][j], f[i-1][j-1]+1);
22        }
23
24    out.println(f[n][m]);
25
26    out.flush();
27 }
```

编辑距离

最短编辑距离的简单应用

```
1 static int N = 1010, M = 15;
```

```

2
3 static int n, m;
4 static char[][] str = new char[N][M];
5 static int[][] f = new int[N][N];
6
7 static int edit_dist(char[] a, char[] b) {
8     int la = a.length-1, lb = b.length-1;    //注意la, lb取值
9
10    for (int i=1; i<=la; i++) f[i][0] = i;
11    for (int i=1; i<=lb; i++) f[0][i] = i;
12
13    for (int i=1; i<=la; i++)
14        for (int j=1; j<=lb; j++) {
15            f[i][j] = Math.min(f[i-1][j], f[i][j-1])+1;
16            f[i][j] = Math.min(f[i][j], f[i-1][j-1]+(a[i] == b[j]? 0:
171));
18        }
19    return f[la][lb];
20 }
21
22 public static void main(String[] args) throws Exception{
23     String[] s = inb.readLine().split(" ");
24     n = Integer.parseInt(s[0]); m = Integer.parseInt(s[1]);
25
26     //注意下标从1开始
27     for (int i=0; i<n; i++) str[i] = (" "+inb.readLine()).toCharArray();
28
29     while (m-- > 0) {
30         s = inb.readLine().split(" ");
31         char[] ss = (" "+s[0]).toCharArray(); int limit =
Integer.parseInt(s[1]);
32
33         int res = 0;
34         for (int i=0; i<n; i++)
35             if (edit_dist(str[i], ss) <= limit) res++;
36
37         out.println(res);
38     }
39
40     out.flush();
41 }

```

区间DP

定义状态时，状态用区间表示

石子合并

分析思路：

- 状态表示：f[i, j] (第i堆石子到第j堆石子的闭区间)

- 集合：所有将第*i*堆石子到第*j*堆石子合并成一堆石子的合并方式集合，答案即为 $f[1, n]$
 - 属性：所有合并方式里面代价最小值
 - 状态计算：集合划分
 - 最后一步一定是将两堆石子合并为一堆石子，因此可以用最后一次合并石子的分界线来进行子集分类。
- 对于区间 $[i, j]$ ，一共有 k ($k=j-i+1$) 堆石子，以最后一次分界线作分类标准，则可分为如下类别：第一类是左边一个，右边为 $k-1$ 个，即 $(1, k-1)$ ；第二类为 $(2, k-2)$ 一直到第 $k-1$ 类 $(k-1, 1)$ 。
- 对于 $[i, k], [k+1, j]$ ，则该类的最小代价为 $f[i, k] + f[k+1, j] + (s[j] - s[i-1])$ ($s[i]$ 表示前*i*堆石子的重量之和，此处用前缀和求 $[i, j]$ 区间中所有石堆的重量之和)
- 因此，递推方程为： $f[i, j] = \text{Min}(f[i, k] + f[k+1, j] + (s[j] - s[i-1]))$ ，且 $k = i \sim (j-1)$
- 时间复杂度： $O(\text{状态数量} * \text{状态转移的计算次数}) = O((n^2) * n) = O(n^3)$

具体实现：区间DP注意枚举的顺序？需保证所有前驱状态在用到时都已经被计算出来

枚举顺序：枚举区间长度，按区间长度从小到大进行枚举

```

1  static int N = 310, INF = 0x3f3f3f3f;
2
3  static int n;
4  static int[] s = new int[N];
5  static int[][] f = new int[N][N];
6
7  public static void main(String[] args) throws Exception {
8      ins.nextToken(); n = (int)ins.nval;
9
10     for (int i=1; i<=n; i++) { ins.nextToken(); s[i] = (int)ins.nval; }
11     for (int i=1; i<=n; i++) s[i] += s[i-1];    //处理前缀和
12
13     for (int len=2; len<=n; len++) {          //枚举区间长度
14         for (int i=1; i+len-1<=n; i++) {      //枚举左端点
15             int l = i, r = i+len-1;          //区间左右端点
16             f[l][r] = INF;
17
18             for (int k=l; k<r; k++)
19                 f[l][r] = Math.min(f[l][r], f[l][k]+f[k+1][r]+s[r]-s[l-1]);
20         }
21     }
22
23     out.println(f[1][n]);
24
25     out.flush();
26 }

```

具体实现2：记忆化搜索

进一步思考：如果每次可以合并 n 堆石子怎么做（对分界线的位置再套一层dp（ $g[i, j]$ 表示在 $1 \sim i$ 个位置分为 j 组的最小值），即DP套DP问题）