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=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=最小值 开始循环。
>
> dp时间复杂度：O(状态*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

```
#include <iostream>

using namespace std;

const int N = 510, INF = 1e9;;

int n;
int a[N][N];
int f[N][N];     //f[i][j]表示从起点走到(i, j)所有路径权值之和的最大值

int main(void) {
    scanf("%d", &n);

    for (int i=1; i<=n; i++)
        for (int j=1; j<=i; j++)
            scanf("%d", &a[i][j]);

    //初始化f[i][j], 以处理边界情况
    //注意每行左边界往左与右边界往右各应多初始化一个位置,下一层会用到这两个位置
    for (int i=0; i<=n; i++)
```

```
        for (int j=0; j<=i+1; j++)
            f[i][j] = -INF;

    //初始化dp初始边界
    f[1][1] = a[1][1];
    for (int i=2; i<=n; i++)
        for (int j=1; j<=i; j++)
            //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]);

    //在最后一层寻找最大值作为答案
    int res = -INF;
    for (int i=1; i<=n; i++) res = max(res, f[n][i]);

    cout << res;

    return 0;
}
```

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) >= ai，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

```
#include <iostream>

using namespace std;

const int N = 1010;

int n;
int a[N];
int f[N];    //f[i]表示以a[i]结尾的上升子序列的最大长度

int main(void) {
    scanf("%d", &n);

    for (int i=1; i<=n; i++) scanf("%d", &a[i]);

```

```
    for (int i=1; i<=n; i++) {
        f[i] = 1;   //当只有一个数的情况，最少为1
        //枚举倒数第二个数
        for (int j=1; j<i; j++)
            if (a[j] < a[i]) f[i] = max(f[i], f[j]+1);
    }

    int res = 0;
    //搜索f[i]最大值
    for (int i=1; i<=n; i++) res = max(res, f[i]);

    cout << res << endl;

    return 0;
}
```

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：

```
#include <iostream>

using namespace std;

const int N = 1010;

int n;
int a[N];
int f[N];   //f[i]表示以a[i]结尾的上升子序列的最大长度
int g[N];   //存储转移的过程

int main(void) {
    scanf("%d", &n);

    for (int i=1; i<=n; i++) scanf("%d", &a[i]);

    for (int i=1; i<=n; i++) {
        f[i] = 1;   //当只有一个数的情况，最少为1
        g[i] = 0;   //当只有一个数的情况，没有前驱
        //枚举倒数第二个数
        for (int j=1; j<i; j++)
            if (a[j] < a[i])
                if (f[i] < f[j]+1) {
                    f[i] = f[j] + 1;
                    g[i] = j;   //存储转移的前驱是谁，结束之后通过往前递推得出转移路径
                }
    }

    int k = 1;  //记录最优解下标
    //搜索f[i]最大值
    for (int i=1; i<=n; i++)
        if (f[k] < f[i]) k = i;

    cout << f[k] << endl;
```

```

    //倒着输出最长子序列
    for (int i=0, len=f[k]; i<len; i++) {
        printf("%d ", a[k]);
        k = g[k];
    }

    return 0;
}
```

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logn，一共有n个数，因此时间复杂度为O(nlogn)，满足题目数据要求

具体实现：

```
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 1000010;

int n;
int a[N];
int q[N], len;    //q[i]表示在当前数前面，上升子序列长度为i的结尾最小值
                  //len表示在当前最大上升子序列的长度，初始时长度为0
                  //也可以理解为当前q数组最后一个元素的位置

int main(void) {
    scanf("%d", &n);

    for (int i=0; i<n; i++) scanf("%d", &a[i]);

    q[0] = -2e9;     //设置哨兵，保证初始时q时单调序列，满足二分的性质

    //对每一个数分别考虑
    for (int i=0; i<n; i++) {
        int l = 0, r = len;     //二分左右边界

        while (l < r) {
            int mid = l+r+1>>1;
            if (q[mid] < a[i]) l = mid;
```

```
            else r = mid-1;
        }

        len = max(len, r+1);
        q[r+1] = a[i];  //由于二分性质，更新长度为r+1上升子序列结尾的最小值。
    }

    cout << len << endl;

    return 0;
}
```

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a[i]与b[j]不一定要求出现在最后一个位置。**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(状态数量*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。

```
#include <iostream>

using namespace std;

const int N = 1010;

int n, m;
char a[N], b[N];
```

```
int f[N][N];    //f[i][j]表示所有在a前i个字母中出现，且在b前j个字母中出现的公共子序列

int main(void) {
    scanf("%d%d", &n, &m);
    scanf("%s%s", a+1, b+1);    //需用到i-1与j-1，所以从第1个位置开始存储

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

    cout << f[n][m] << endl;

    return 0;
}
```

最短编辑距离

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 ~ i]变成b[1 ~ 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~ (i-1)] = b[1 ~ 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 ~ i] = b[1 ~ 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] != 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：

```
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 1010;

int n, m;
char a[N], b[N];
int f[N][N];    //f[i][j]表示将a[1~i]变成b[1~j]的操作方式中操作次数的最小值

int main(void) {
    scanf("%d%s", &n, a+1); //使用到了i-1
    scanf("%d%s", &m, b+1);

    //根据实际意义初始化边界情况
    for (int i=0; i<=m; i++) f[0][i] = i;   //在a中插入字符操作
```

```
    for (int i=0; i<=n; i++) f[i][0] = i;    //在a中删除字符操作

    //递推方程实现
    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++) {
            f[i][j] = min(f[i-1][j]+1, f[i][j-1]+1);
            if (a[i] == b[j]) f[i][j] = min(f[i][j], f[i-1][j-1]);
            else f[i][j] = min(f[i][j], f[i-1][j-1]+1);
        }

    cout << f[n][m] << endl;

    return 0;
}
```

编辑距离

最短编辑距离的简单应用

```
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 15, M = 1010;

int n, m;
char str[M][N];
int f[N][N];

int edit_dist(char *a, char *b) {
    int la = strlen(a+1), lb = strlen(b+1);

    //初始化边界
    for (int i=0; i<=lb; i++) f[0][i] = i;
    for (int i=0; i<=la; i++) f[i][0] = i;

    for (int i=1; i<=la; i++)
        for (int j=1; j<=lb; j++) {
            f[i][j] = min(f[i-1][j]+1, f[i][j-1]+1);
            f[i][j] = min(f[i][j], f[i-1][j-1]+(a[i] != b[j]));
        }

    return f[la][lb];
}

int main(void) {
    scanf("%d%d", &n, &m);

    for (int i=0; i<n; i++) scanf("%s", str[i]+1);

    while (m--) {
        char s[N];
        int limit;
```

```
        scanf("%s%d", s+1, &limit);

        int res = 0;     //记录有多少个字符串满足限制条件
        for (int i=0; i<n; i++)
            if (edit_dist(str[i], s) <= limit)
                res++;

        printf("%d\n", res);
    }

    return 0;
}

```

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] = Min(f[i, k] + f[k+1, j] + (s[j]-s[i-1]))，且k = i~(j-1)**

      时间复杂度：O(状态数量*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

```
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 310, INF=0x3f3f3f3f;

int n;
int s[N];    //前缀和数组
```

```
int f[N][N];    //f[i][j]表示将第i堆石子到第j堆石子合并成一堆石子的最小代价

int main(void) {
    scanf("%d", &n);

    for (int i=1; i<=n; i++) scanf("%d", &s[i]);

    //处理前缀和
    for (int i=1; i<=n; i++) s[i] += s[i-1];

    //首先枚举区间长度，且如果长度为1，则合并代价为0，故从len=2开始枚举
    for (int len=2; len<=n; len++)
        for (int i=1; i+len-1<=n; i++)  {   //枚举起点
            int l = i, r = i+len-1;     //计算区间左端点和右端点
            f[l][r] = INF;      //初始化f[l][r]为较大值

            //若区间长度为0，则合并代价为0，因此可以直接从len=2开始枚举
            //if (len == 1) f[l][r] = 0;

            for (int k=l; k<r; k++) //枚举分界线的位置l~(r-1)
                f[l][r] = min(f[l][r], f[l][k]+f[k+1][r]+s[r]-s[l-1]); //递推方程
        }

    cout << f[1][n] << endl;    //输出结果

    return 0;
}
```

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~i个位置分为j组的最小值），即DP套DP问题）