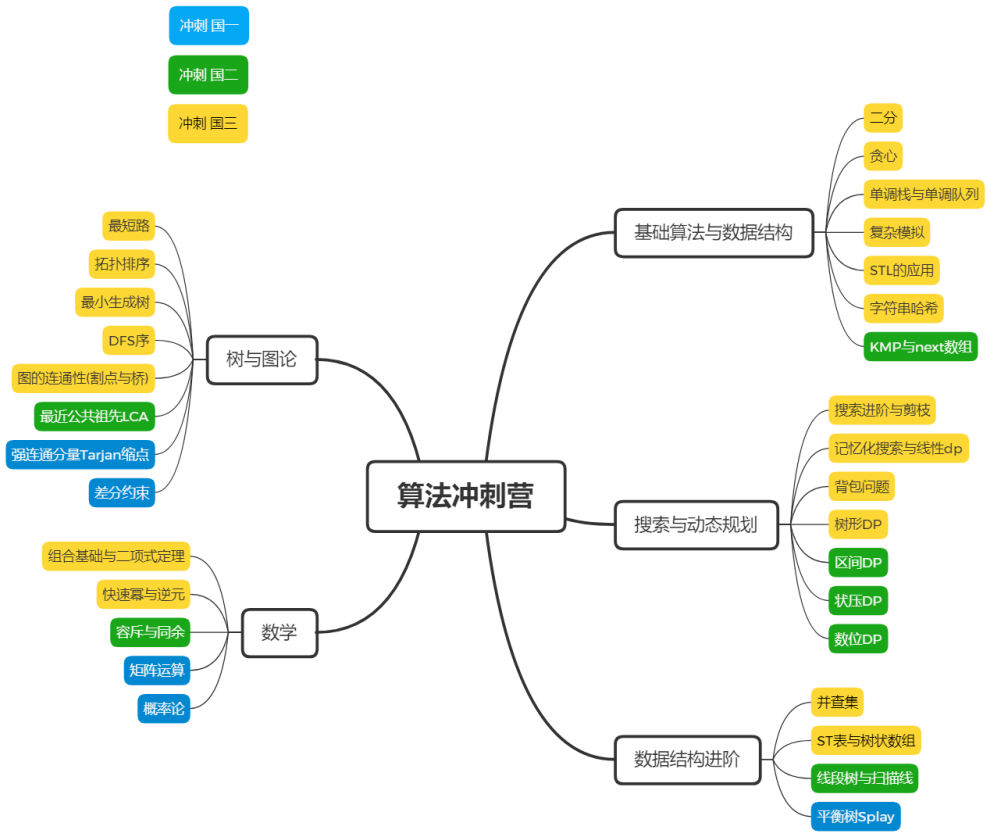
蓝桥杯30天算法冲刺集训



Day-10 区间DP

能量项链

什么是区间dp呢，指在一段区间上的 dp，通过枚举左右子区间来求出解。

本题属于区间动态规划，一圈珠子，每个珠子的尾标记都是后一个珠子的头标记。题目中输入的是每个珠子的头标记，那么每个珠子的尾标记则为 $num[(i+1)\%n]$，这样就知道了每个珠子的头标记和尾标记。要求最后释放的总能量最大，长 n 的区间最后合并成一个点，区间动态规划，先从小的开始，即以区间长度为 1 开始合并，一直到区间长度为 $n-1$ 的状态，每个状态再考虑分割合并，求局部最优解。

```
# 区间dp模板:
for length in range(1, n + 1):
    for l in range(1, n - length + 2):
        r = l + length - 1
        # do something
        for k in range(l, r):
            # update dp array, such as 'min(dp[l][r], dp[l][i] + dp[i + 1][r])'
            pass  # placeholder for the actual code
```

$dp[i][j]$ 是以 $tou[i]$ 为头标记到以 $wei[j]$（即 $tou[j+1]$）为尾标记合并释放的最大能量。

状态转移方程：

$$dp[i][j] = max(dp[i][j], dp[i][k] + dp[(k+1)\%n][j] + num[i] * num[(k+1)\%n] * num[(j+1)\%$$

所有的状态求完后，最后只要在区间程度为 $n-1$ 的 $dp[][]$ 里面找最大值就是所求。

```
import sys

maxn = -1
dp = [[0]*300 for _ in range(300)]

def main():
    global maxn
    n = int(input())
    e_input = input().split()
    e = [0] * 601
    for i in range(1, n + 1):
        e[i] = int(e_input[i - 1])
        e[i + n] = e[i]
    e[2 * n + 1] = e[1]  # 将尾链接到头
    # 珠子由环拆分为链，重复存储一遍
    for length in range(2, n + 1):
        for l in range(1, 2 * n - length + 2):
            r = l + length - 1
            dp[l][r] = 0
            for k in range(l, r):
                # k是项链的左右区间的划分点
                dp[l][r] = max(dp[l][r], dp[l][k] + dp[k + 1][r] + e[l] * e[k + 1] *
e[r + 1])
            # 状态转移方程：max(原来能量，左区间能量+右区间能量+合并后生成能量)
            if length == n and dp[l][r] > maxn:
                maxn = dp[l][r]  # 求最大值
    print(maxn)

if __name__ == "__main__":
    main()
```

蓝桥杯真题

搭积木（蓝桥杯C/C++2018B组国赛）

> *[!NOTE]*
>
> *本题要拿到满分需要额外在区间DP上做一个优化，因此只要能写出朴素的区间DP即可拿到40%的分数，比单纯的暴力要多30%的分数，本题应该属于压轴题难度，往年一个题算25分的话，那么也就能拉开7.5分的差距，相当于一个半填空题了。*

首先我们讲一下朴素区间DP的做法，也就是能拿到40%分数的做法。

首先我们需要看如何去转移，根据题目里面的条件：

1. 每块积木必须紧挨着放置在某一块积木的正上方，与其下一层的积木对齐；
2. 同一层中的积木必须连续摆放，中间不能留有空隙；
3. 小明不喜欢的位置不能放置积木。

我们可以设置$f[i][j][k]$**代表第**i**层在区间**$[j,k]$**内放置积木所有可能的情况**

我们可以从下到上枚举每一层的可以放上积木的区间，然后我们最终的答案就是这些可以放上积木的区间的加和，也就是说我们可得答案如下：

$$ans = 1 + \sum f[i][j][k]$$

其中1代表的是题目里面的 地基上什么都不放，也算作是方案之一种

那么怎么样才算可以放上积木的区间呢，是不是只要我们当前这一行的这个要选定的区间内没有 X 并且我们下面有拖着我们的积木即可。

那么我们怎么去找任意的给定一个区间类似$[j,k]$这个区间内有没有 X 呢？实际上我们只要对每一层做一个前缀和处理即可，然后直接就能$O(1)$得出当前的区间是否会有 X

我们还需要初始值设置第n层也就是最下面一层的$f[n][j][k]$满足下面的初始值条件：

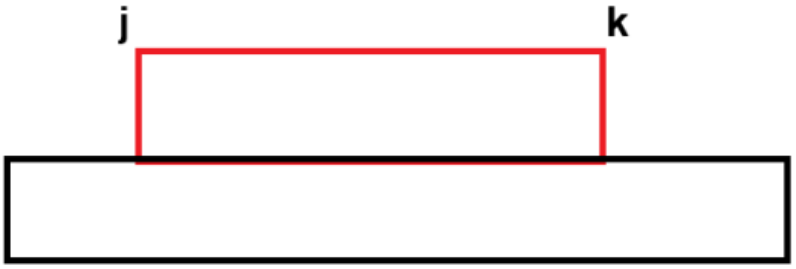
$$\sum_{j=1}^{m}\sum_{k=j}^{m}[\text{满足在区间}[j,k]\text{中没有}X]\ f[n][j][k] = 1$$

也就是只要能放积木的区间，我们都先给他设置一个初始值1，因为我们后面都是从$n-1$层开始枚举转移了。

然后我们是不是对于每一层只要枚举区间即可这样的话复杂度已经到达了$O(nm^2)$

我们再考虑转移，转移的话是不是我们要枚举我们下面的积木的长度，我们知道我们当前这一层如果能放上积木的话，我们下面这一层肯定至少是有$k-j+1$个积木的，但是这不一定就是只有这些我们还需要左右两边一块枚举再把之前的算出的答案加上即可。

大概如下图所示：



我们现在知道红色部分能够放置积木，那么我们至少包含红色积木下面的积木，然后再两边拓展边界，组合成不同的情况。

注意如果当前可以放置红色部分积木但是红色部分积木正下面包含 X 的话，也是可以的，因为转移的值都会是0

也就是说我们需要设置

$$f[i][j][k]=\sum_{x=1}^{j}\sum_{y=k}^{m}f[i+1][x][y]$$

那么我们再开两个for循环枚举即可，最终复杂度达到了$O(nm^4)$，具体转移如下，期望最少通过40%的数据：

$$f[i][j][k]=\sum_{i=n-1}^{1}\sum_{j=1}^{m}\sum_{k=j}^{m}\sum_{x=1}^{j}\sum_{y=k}^{m}[\text{满足在区间}[j,k]\text{中没有}X]\ f[i+1][x][y]$$

对于满分的做法我们需要优化上面给出的式子，那么如何优化呢？我们可以考虑用二维前缀和的思想来优化。

我们都知道二维前缀和是用来求解矩阵内的任意一个左上角和右下角类似$(x1,y1)$，$(x2,y2)$所包含在内的子矩阵的矩阵值之和的。但是二位前缀和不止如此，实际上我们还可以广义上的去使用他，也就是说对于任意一个二重for循环求解值的式子可以把两个for循环起始的节点当作左上角的$(x1,y1)$终点当作右下角的$(x2,y2)$，那么我们最终实际上可以看作在一个矩阵上面求二维前缀和，所以我们对于之前的式子。

$$f[i][j][k]=\sum_{x=1}^{j}\sum_{y=k}^{m}f[i+1][x][y]$$

实际上可以看作是从左上角$(1,k)$到右下角(j,m)做的一个二维矩阵的求矩阵值之和的运算，只是我们的矩阵值都是做区间DP转移的DP值，那么我们直接把这个用二维前缀和优化就好了。这样我们只需要$O(1)$的时间就把这个式子优化掉了。

具体怎么优化呢？实际上只要在每次遍历当前层的时候先把上一层的结果拿出来，做一个二维前缀和优化，然后再计算当前层的结果即可。最终时间复杂度为$O(nm^2)$可以满分通过本题，具体转移如下：

$$getSum(i+1,j,k)=\sum_{x=1}^{j}\sum_{y=k}^{m}f[i+1][x][y]$$

$$f[i][j][k]=\sum_{i=n-1}^{1}\sum_{j=1}^{m}\sum_{k=j}^{m}[\text{满足在区间}[j,k]\text{中没有}X]\ getSum(i+1,j,k)$$

示例代码如下：

```
mod = 10**9 + 7
N = 105

def main():
    n, m = map(int, input().split())
    num = [[0] * (m + 1) for _ in range(n + 1)]
    dp = [[[0] * (m + 1) for _ in range(m + 1)] for _ in range(n + 1)]
    sum_dp = [[0] * (m + 1) for _ in range(m + 1)]

    for i in range(1, n + 1):
        s = input()
        for j in range(1, m + 1):
            num[i][j] = num[i][j - 1] + (s[j - 1] == 'X')

    ans = 1
    for l in range(1, m + 1):
        for r in range(l, m + 1):
            dp[n][l][r] = int(num[n][r] - num[n][l - 1] == 0)
            ans = (ans + dp[n][l][r]) % mod

    for i in range(n - 1, 0, -1):
        for l in range(1, m + 1):
            for r in range(1, m + 1):
                sum_dp[l][r] = (dp[i + 1][l][r] + sum_dp[l][r - 1] + sum_dp[l - 1][r]
- sum_dp[l - 1][r - 1]) % mod
        for l in range(1, m + 1):
            for r in range(l, m + 1):
                if num[i][r] - num[i][l - 1] == 0:
                    dp[i][l][r] = (sum_dp[l][m] - sum_dp[0][m] - sum_dp[l][r - 1] +
sum_dp[0][r - 1]) % mod
                    ans = (ans + dp[i][l][r]) % mod

    print((ans + mod) % mod)

if __name__ == "__main__":
    main()
```

习题

小蓝与狼

首先，基本攻击力造成的伤害在整个过程中一定是 n，因为会消灭 n 匹狼，消灭每匹狼受到的基本攻击力为 1，所以这 n 点伤害我们单独加上去，在求解的时候只考虑攻击力加成造成的伤害。

其次，我们消灭狼到只剩两匹，假设这两匹狼分别是狼 x 和狼 y，那么在狼圈中给 x 和 y 连一条线，两侧的狼之间互不影响（互相不会给攻击力加成），那么我们可以把两侧的狼单独考虑，分别考虑如何消灭这些狼能使得收到的攻击最少。

那么，对于一条直线上的 m 头狼，消灭除了端点外的其他狼需要付出的代价我们可以通过区间 DP 求解。

设 $dp[i][j]$ 表示消灭区间 $[i,j]$ 中除了端点外的其他狼需要付出的最小代价。

$$dp[i][j] = min\{dp[i][k] + dp[k+1][j] \mid i < k < j-1\}$$

但如果每次确定下来狼 x 和狼 y 之后才做区间 DP，时间复杂度会高达 $O(n^5)$，因此，我们可以把狼圈破环为链，变成 $1,2,3,\ldots,n,1,2,3,\ldots,n$ 这个长度为 $2n$ 的狼群序列。对这个序列做区间 DP，我们会发现，确定下来狼 x 和狼 y 之后，对应的两半总能在上面那个序列中找到对应的区间与之匹配，从而直接在预先算出的 dp 数组中获取消灭这些狼要付出的最小代价。

```
import sys

def main():
    inf = 10**9
    n = int(input())
    a = [0] * 1010
    dp = [[0] * 1010 for _ in range(1010)]

    # 读取空格分隔的数字序列，并转换为整数列表
    numbers = list(map(int, input().split()))
    for i in range(1, n + 1):
        a[i] = numbers[i - 1]
        a[i + n] = a[i]

    if n == 1:
        print(1)
        return

    for i in range(1, (n << 1) + 1):
        for j in range(1, (n << 1) + 1):
            dp[i][j] = inf
        dp[i][i] = dp[i][i + 1] = 0

    for l in range(2, n):
        for i in range(1, (n << 1) + 1):
            j = i + l
            if j > (n << 1):
                continue
            for k in range(i + 1, j):
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + a[i] + a[j])

    ans = inf
    for i in range(1, n + 1):
        for j in range(i + 1, n + 1):
            ans = min(ans, dp[i][j] + dp[j][i + n] + min(a[i], a[j]))

    print(ans + n)

if __name__ == "__main__":
    main()
```

小蓝与招聘会

设 $f[i][j]$ 表示学生综合能力为 $[i,j]$ 的子集最大权值。

显然，$f[i][j]=\max(f[i][k-1]+f[k][j])$，$k\in[i,j]$

再考虑待合并的两个状态 $f[i][k-1]$ 和 $f[k+1][j]$，此时第 k 个学生还没有被公司招收。

我们设 $g[k][i][j]$ 表示所有满足 $i\le l\le k\le r\le j$ 的公司能给出的最高的薪水，那么 k 号学生一定会去 $g[k][i][j]$ 对应的公司。

那么转移方程：

$$f[i][j]=\max(f[i][k-1]+g[k][i][j]+f[k+1][j]),k\in(i,j)$$

因为计算 f 数组需要用到 g 数组，因此代码实现时，需要先 $O(n^3)$ 预处理出 g 数组。

```
N = 305
n, m = map(int, input().split())
f = [[0] * N for _ in range(N)]
g = [[[0] * N for _ in range(N)] for _ in range(N)]

for _ in range(m):
    w, l, r = map(int, input().split())
    l -= 1
    r -= 1
    for j in range(l, r + 1):
        g[l][j][r] = max(g[l][j][r], w)

for length in range(1, n):
    for i in range(n - length):
        j = i + length
        for k in range(i, j + 1):
            if j > 0:
                g[i][k][j] = max(g[i][k][j], g[i][k][j - 1])
            if i < n - 1:
                g[i][k][j] = max(g[i][k][j], g[i + 1][k][j])

A, B = 0, 0
for length in range(n):
    for i in range(n - length):
        j = i + length
        for k in range(i, j + 1):
            if k < n - 1:
                f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j])
            A = f[i][k - 1] if k > 0 else 0
            B = f[k + 1][j] if k < n - 1 else 0
            f[i][j] = max(f[i][j], A + B + g[i][k][j])

print(f[0][n - 1])
```