Acwing1262.鱼塘钓鱼

从贪心到动态规划到优先队列

南川，2021年02月10日

1262. 鱼塘钓鱼 - AcWing题库

有 N 个鱼塘排成一排，每个鱼塘中有一定数量的鱼，例如：N=5 时，如下表：

鱼塘编号	1	2	3	4	5
第1分钟能钓到的鱼的数量（1..1000）	10	14	20	16	9
每钓鱼1分钟钓鱼数的减少量（1..100)	2	4	6	5	3
当前鱼塘到下一个相邻鱼塘需要的时间（单位：分钟）	3	5	4	4	

即：在第 1 个鱼塘中钓鱼第 1 分钟内可钓到 10 条鱼，第 2 分钟内只能钓到 8 条鱼，……，第 5 分钟以后再也钓不到鱼了。

从第 1 个鱼塘到第 2 个鱼塘需要 3 分钟，从第 2 个鱼塘到第 3 个鱼塘需要 5 分钟，……

给出一个截止时间 T，设计一个钓鱼方案，从第 1 个鱼塘出发，希望能钓到最多的鱼。

假设能钓到鱼的数量仅和已钓鱼的次数有关，且每次钓鱼的时间都是整数分钟。

输入格式

共 5 行，分别表示：

第 1 行为 N；

第 2 行为第 1 分钟各个鱼塘能钓到的鱼的数量，每个数据之间用一空格隔开；

第 3 行为每过 1 分钟各个鱼塘钓鱼数的减少量，每个数据之间用一空格隔开；

第 4 行为当前鱼塘到下一个相邻鱼塘需要的时间；

第 5 行为截止时间 T。

输出格式

一个整数（不超过231−1），表示你的方案能钓到的最多的鱼。

数据范围

$1 \le N \le 100$,
$1 \le T \le 1000$

输入样例：

```
5
10 14 20 16 9
2 4 6 5 3
3 5 4 4
14
```

输出样例：

```
76
```

第一步：画出每个鱼塘的增量情况

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	10	8	6	4	2	0	0	0	0	0	0	0	0	0
2	5	14	10	6	2	0	0	0	0	0	0	0	0	0	0
3	4	20	14	8	2	0	0	0	0	0	0	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0
5	0	9	6	3	0	0	0	0	0	0	0	0	0	0	0

注：

1. 每个鱼塘都有三个信息，起始量、减量、下一个鱼塘距离。
2. 在这张图中，我们将起始量和减量展开成实际量，并用颜色标注其距离。
3. 也就是说，处于颜色标注的单元格内的信息都是相对“安全”的。

第二步：分析每个鱼塘的转移条件

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	10	8	6	4	2	0	0	0	0	0	0	0	0	0
2	5	14	10	6	2	0	0	0	0	0	0	0	0	0	0
3	4	20	14	8	2	0	0	0	0	0	0	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0
5	0	9	6	3	0	0	0	0	0	0	0	0	0	0	0

注：

1. 显然，考虑从一个鱼塘转移到下一个鱼塘的主要条件，就在于接下来 c[i] 分钟的时间代价是否能被下一个鱼塘的第一分钟的收益所弥补。
2. 但是，要想利用好这个条件，还得意识到，当且仅当该鱼塘走完有颜色的格子之后才能进入下一个鱼塘的第一个格子。
3. 并且，一旦进入下一个鱼塘的第一个格子后，原先积累的c[i]个格子，将要全部清除，以此类推。

第三步：转移鱼塘的关键截面（1→5）

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	**10**	**8**	**6**	**4**	**2**	0	0	0	0	0	0	0	0	0
2	5	14	10	6	2	0	0	0	0	0	0	0	0	0	0
3	4	20	14	8	2	0	0	0	0	0	0	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0
5	0	9	6	3	0	0	0	0	0	0	0	0	0	0	0

注：

1. 显然，前5分钟都应该呆在第一个鱼塘，这样能够钓到 10 + 8 + 6 + 4 + 2= 30 条鱼。
2. 因为，转移到下一个鱼塘，至少损失 8 + 6 + 4 + 2= 20 条鱼，超过其第二个鱼塘第一分钟内能提供的收益 14 条鱼。
3. **注意：成本要算 k + 1 分钟，即 当且仅当 F(i+1, 0) > F(i, t->t+k+1) 时才能转移（即 14 > 6 + 4 + 2 + 0）。**

第三步：转移鱼塘的关键截面（5→6）

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	**10**	8	6	4	2	0	0	0	0	0	0	0	0	0
2	5	**14**	**10**	6	2	0	0	0	0	0	0	0	0	0	0
3	4	20	14	8	2	0	0	0	0	0	0	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0
5	0	9	6	3	0	0	0	0	0	0	0	0	0	0	0

注：

1. 显然，当我们有六分钟时，一直呆在第一个鱼塘只能获得30条鱼，但如果我们抛弃其中的三分钟，对于剩下的三分钟，我们就有任意分配权。
2. 当我们拥有两个鱼塘的时间分配权后，无论是选择“1+2”（10+14+10=34）还是“2+1”（10+8+14=32）的方案，都比傻傻地呆在第一个鱼塘要好。
3. 这里面最关键的转移条件是：14 > 6 + 4 + 2 + 0，但更为值得注意的是，当我们触发转移后，我们便可以任意选择两个鱼塘里最高的几个数。
4. 这其中，隐约着有一丝丝并查集的思想。

第三步：转移鱼塘的关键截面（6→12）

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	**10**	**8** → **6** → **4** → **2**				0	0	0	0	0	0	0	0	0
2	5	**14**	**10** → **6**	**2**	0		0	0	0	0	0	0	0	0	0
3	4	20	14	8	2	0	0	0	0	0	0	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0
5	0	9	6	3	0	0	0	0	0	0	0	0	0	0	0

注：

1. 基于上一页的分析，我们意识到，当启动了两个鱼塘之后，我们需要对它们的信息合并（见图中前两行新的染色块）。
2. 此时，原先从第二个鱼塘转移到第三个鱼塘的问题，转化成从前两个鱼塘转移到第三个鱼塘的问题。
3. 显然，我们需要寻找新的五个最小的时间块，以触发前往第三个鱼塘的条件。
4. 但在此之前，我们每一分钟都应该选择最大的时间块，所以前12分钟我们能收获：14+10+10+8+6+6+4+2+2=62条鱼。

第三步：转移鱼塘的关键截面（12→14）

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	**10**	8	6	4	2	0	0	0	0	0	0	0	0	0
2	5	**14**	**10**	6	2	0	0	0	0	0	0	0	0	0	0
3	4	**20**	**14→8**		2	0	0	0	0	0	0	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0
5	0	9	6	3	0	0	0	0	0	0	0	0	0	0	0

Are We Really Right？

注：

1. 现在，从第12分钟到第13分钟，发生了翻天覆地的变化。
2. 最主要的问题在于，原先两个鱼塘最小的六个数（5+1）都给弃除了，用以换取第三个鱼塘第一分钟的收益。
3. 但无论如何，我们实现了我们的目标，即收获了最后 20+14+14+10+10+8=76条鱼，确实是正确答案。

上述贪心法的一个简单反例

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	10	8	6	4	2	0	0	0	0	0	0	0	0	0
2	1	4	3	2	1	0	0	0	0	0	0	0	0	0	0
3	4	100	90	80	70	60	50	40	30	20	10	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0

注：

1. 在该反例中，我们在两个较为“肥沃”的鱼塘之中，穿插了一个较为“贫瘠”的鱼塘。
2. 显然，第一分钟、第二分钟我们都应该留在第一个鱼塘，以获得18条鱼。
3. 但但凡我们有三分钟，我们都应该毫不犹豫地前往第三个鱼塘，因为只要过去，我们就能获得100条鱼！
4. 这个简单的例子说明，我们不能简单地通过“一步贪心法”获得正确答案（除非我们考虑了每一步可能到达的地方与其最大收益。
5. 然而，这就已经不是简单的贪心了，这是暴力搜索！

动态规划法第一步：先求每个鱼塘前缀和

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	10	8+10 =18	6+18 =24	4+24 =28	2+28 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30
2	5	14	10+14 =24	6+24 =30	2+30 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32
3	4	20	14+20 =34	8+34 =42	2+42 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44
4	4	16	11+16 =27	6+27 =33	1+33 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34
5	0	9	6+9 =15	3+15 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18

注：

1. 使用动态规划法，我们可以保证每一步的正确性，状态转移方程为 **F(i, T, t) = max(F(i, T-1, t+1) + w(i, t), F(i+1, T-c(i), 0)**。
2. 在该转移方程中，第一个子式是指继续留在池塘 i 再钓一分钟鱼能获得的总鱼数，第二个子式则是前往下一个鱼塘能获得的总鱼数。
3. 显然，最后一个鱼塘最容易分析，它没有下一个鱼塘，无论有多少时间，只能一直选择在该鱼塘钓鱼（即便已经没有鱼）。
4. 所以，我们可以得到，鱼塘五的收益之于时间的关系为：[0, 9, 15, 18, 18 … 18]，基于此，我们也可以直接先求出所有鱼塘的前缀和。

动态规划法第二步：状态转移过程（一）

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	10	8+10 =18	6+18 =24	4+24 =28	2+28 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30
2	5	14	10+14 =24	6+24 =30	2+30 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32
3	4	20	14+20 =34	8+34 =42	2+42 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44
4	4	16	11+16 =27	6+27 =33	1+33 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34
5	0	9	6+9 =15	3+15 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18

注：

1. 我们从第4个鱼塘往前推导。前四分钟内，必然不动为妙。但接下来的几分钟，则需要一一判定。
2. 首先是第五分钟，可以选择继续留在第四个鱼塘，值为F(4,5)=34，也可以是F(5,1)=9，但并不讨好。第六分钟，同理。
3. 但对于第七分钟，有7-4=3种选择，即F(4,7)=34、F(4,2)+F(5,1)=36、F(4,1)+F(5,2)=31，显然，应该选“2+1”的方案。
4. 如此，可遍历进行下去，对第i个池塘的第j分钟，它能获得的最大收益取决于它与下一个池塘 j-c[i]种时间分配方案的最大值。

动态规划法第二步：状态转移过程（二）

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	10	8+10 =18	6+18 =24	4+24 =28	2+28 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30	0+30 =30
2	5	14	10+14 =24	6+24 =30	2+30 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32	0+32 =32
3	4	20	14+20 =34	8+34 =42	2+42 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44	0+44 =44
新4	4	16	27	33	34	34	34	(2,1) 27+9 =36	(2,2) 27+15 =42	(3,2) 33+15 =48	(3,3) 33+18 =51	(4,3) 34+18 =52	(..) 52	(..) 52	(..) 52
4	4	16	11+16 =27	6+27 =33	1+33 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34	0+34 =34
5	0	9	6+9 =15	3+15 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18	0+18 =18

注：

1. 直接给出F(4,14)的状态。最下面两行确定了F'(4,14)这个位置的所有10种可能，图中颜色一一对应的十种相加取最大即是F(4,14)的值。
2. 倒数第三行记录了合并两个鱼塘后的信息，每个值代表着给定这个鱼塘多少分钟能获得的最大收益，从此不用考虑后续鱼塘。
3. 值得注意的是，由于两个（4、5）鱼塘的值是递增有序的，在更新时发现保持着一定的秩序，这是否是一定的，我觉得不一定。
4. 例如考虑：F(4) = [1,2, 20], F(5)=[7,8,9], c[4]=1，那么F'(4,1)=F(4,1)=1, F'(4,2)=F(5,1)=7, F'(4,3)=F(4,3)=20，可以发现出现了反复横跳。
5. 这并不是一个很好的发现，如果不会出现反复横跳，我们就可以一趟遍历完成全部的更新，现在我们需要平方复杂度了。

动态规划法第三步：更新所有转移，得出答案

Time / Id	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	10	18	24	28	30	30	30	30	30	30	30	30	30	30 76
2	5	14 14	24 24	30 30	32 32	32 32	32 32	32 34	32 48	32 58	32 66	32 72	32 74	32 76	32 85
3	4	20 20	34 34	42 42	44 44	44 44	44 50	44 61	44 69	44 75	44 77	44 78	44 78	44 78	44 86
4	4	16 16	11 27	6 33	1 34	0 34	0 34	0 36	0 42	0 48	0 51	0 52	0 52	0 52	0 52
5	0	9	15	18	18	18	18	18	18	18	18	18	18	18	18

注：

1. 从下更新的过程中，我们实际每次只操作三行数据，也就是基于当前的最后两行合并成一个新的行，并替换当前的倒数第二行。
2. 最后一行不用更新（但每行都要求前缀和），第一行只需要更新最后一个F(1,14)，其他每行每个单元格都需要更新。
3. 在每一行内，应该使用"从右往左"的更新方式，因为右边值的更新会基于左边的值。

动态规划法：代码实现与性能分析

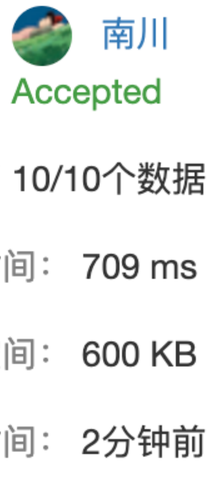
```
const int MAX_N  = 100 + 1, MAX_T = 1000 + 1;
int N, T, w[MAX_N], d[MAX_N], c[MAX_N], dp[MAX_N][MAX_T];

int main ()
{
    scanf("%d", &N);
    for(int i=1; i<=N; i++) scanf("%d", &w[i]);
    for(int i=1; i<=N; i++) scanf("%d", &d[i]);
    for(int i=1; i< N; i++) scanf("%d", &c[i]);
    scanf("%d", &T);

    for(int i=1; i<=N; i++)                 // dp每一行存储一个池塘拥有j分钟的最大收益（1<=j<=T)
        for(int j=1; j<=T; j++)
            dp[i][j] = dp[i][j-1] + w[i],
            w[i] = max(w[i] - d[i], 0);

    for(int i=N-1; i>0; i--)                // 从下往上更新第n-1到第1个池塘的最大收益
        for(int j=T; j>c[i]; j--) {         // 从右往左更新，防止可能的覆盖
            int mx = dp[i][j];              // 留在原池塘能获得的最大收益
            for(int k=1; k<=j-c[i]; k++)    // 分配k分钟给下一个池塘，j-c[i]-k就是原池塘的时间
                mx = max(mx, dp[i+1][k] + dp[i][j-c[i]-k]);
            dp[i][j] = mx;
        }
    printf("%d", dp[1][T]);
}
```

时间复杂度：$O(N * T^2)$
N个池塘
T分钟
每个池塘每一分钟的更新
都要基于前T种组合的最大值



动态规划法：反思与优化

- 基于刚刚的动态规划，我们已经能够实现AC，然而事实上，这版动态规划，几乎只是对暴力法的一个简单提升。
- 提升点在于，暴力法会从上到下把所有的可能都遍历出来，每一步都考虑停留或者向下，复杂度是指数级，盲猜$O(2^{N*T^2})$。
- 而动态规划从下往上，每次确定该鱼塘的所有可能，组织搜索链继续往下，所以消除了这个指数，达到了$O(N*T^2)$。
- 然而，毕竟还有个平方项，这是我们往往所不能接受的，如果能降到$O(N*TlogT)$甚至$O(N*T)$，我们才能松一口气。
- 方法是有的，但我们需要首先思考，这个T^2是怎么来的。
- 显然，是因为每个 $F'(i,t)$ 的产生都需要遍历 $1 \leq j \leq t$ 的时间组合，然而这里面很多组合都被重复使用，需要优化。
- 由于我们之前举了一个反例用于证明两个前缀和的"某种加法"，不具备连续递推性，因此我们需要换种思路。
- 其实，可以考虑放弃前缀和，直接利用原信息，也就是："差分"。
- 事实上，**当我们确定可以使用两个鱼塘的时候，我们每一分钟唯一要思考的就是两个鱼塘中还有谁的值是最大的**。
- 然后，将最大的拿掉，并继续选择下一个最大的，这不就是，"堆"吗？
- 所以，可以考虑将两个鱼塘视作两个堆，**一开始只在一个堆中选，接着可以两个堆一起选**。

双堆法：顺序思考

单堆初始化

1	2	3	4	5	6	7	8	9	10	11	12	13	14	cost	id
16	11	6	1	0	0	0	0	0	0	0	0	0	0	4	4
9	6	3	0	0	0	0	0	0	0	0	0	0	0	0	5

双堆遍历

1	2	3	4	5	6	7	8	9	10	11	12	13	14	cost	id
16	11	6	1	0	0	0	0	0	0	0	0	0	0	4	4
9	6	3	0	0	0	0	0	0	0	0	0	0	0	0	5

更新 i 鱼塘

1	2	3	4	5	6	7	8	9	10	11	12	13	14	cost	id
16	11	9-7=2	6	6	3	1	0	0	0	0	0	0	0		4

注：

1. 双堆要求重新理解每个单元格的意义：增量，比如对于第四个鱼塘，前四分钟分别能获得16、11、6、1条鱼，接下来就要思考新的增量。
2. 可以用一个函数判断，$\sum_{p=1}^{p=t} D(i,p) \le \sum_{p=1}^{p=t-c[i]-k} D(i,p) + \sum_{q=1}^{q=k} D(i+1,q)$，即留在原鱼塘与选择使用两个鱼塘哪个收益更高。
3. 但我们需要提防落入贪心法的陷阱，在双堆法中，我们依旧始终要检查各种组合的最大值，而不能仅仅只与第二堆的第一个元素比较。
4. 从单堆到双堆的检测与实现，是较为繁琐的，具体需要使用双指针的技术。
5. 虽然比原先的动态规划已经快了不少，但是我们依旧有更好的办法。

双堆法：前缀和 + 并行

单堆 →

1	2	3	4	5	6	7	8	9	10	11	12	13	14	cost	id
16	11	6	1	0	0	0	0	0	0	0	0	0	0	4	4
9	6	3	0	0	0	0	0	0	0	0	0	0	0	0	5

合并后的单堆 →

1	2	3	4	5	6	7	8	9	10	11	12	13	14	cost	id
0	0	0	0	16	11	9	6	6	3	1	0	0	0	4	4

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	cost	id
更新单堆前缀和	16	27	33	34	34	34	34	34	34	34	34	34	34	34	-	4
更新双堆前缀和	0	0	0	0	16	27	36	42	48	51	52	52	52	52	-	4 and 5
两者取最大	16	27	33	34	34	34	36	42	48	51	52	52	52	52	4	4′

注：

1. 在双堆前缀并行版之前，我们一直陷入了一个狭隘的思路：要求出T时间内每个时间点的最大收益。
2. 在这种思路下，我们就不得不去耗费心思比较到底留在原池塘收益高，还是下个池塘，如果是下个池塘，什么时候去，等等等等。
3. 逃离这个噩梦的关键在于：**当我们已经做出了需要开拓下一个池塘以获取最高收益的决定后，可以合并两个池塘**，但这里面到底发生了什么？
4. 其实只发生了一件事，那就是我们的时间少了C[i]，**所以时时刻刻思考要不要开辟下一个池塘的决定，完全可以在一开始就决定好。**
5. 也就是说，我们把求解第F(i, T)的问题分解成两个问题F′(i, T)和F′(i′, T-C[i])，其中F′表示不考虑开辟下一个池塘，i′表示合并下一个池塘后的新池塘。

优先队列法：差分+并行

	cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	10	8	6	4	2	0	0	0	0	0	0	0	0	0
2	5	14	10	6	2	0	0	0	0	0	0	0	0	0	0
3	4	20	14	8	2	0	0	0	0	0	0	0	0	0	0
4	4	16	11	6	1	0	0	0	0	0	0	0	0	0	0
5	0	9	6	3	0	0	0	0	0	0	0	0	0	0	0

	sum	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	30	10	8	6	4	2	0	0	0	0	0	0	0	0	0
2	62	14	10	10	8	6	6	4	2	2	0	0			
3	76	20	14	14	10	10	8								
4	36	20	16												
5	0														

注：

1. 经过前面那么长的铺垫，最后一种思路就呼之欲出了，那就是我们完全不考虑T时间内在什么鱼塘已经花了多少时间。
2. 我们只需要考虑要合并几个鱼塘，合并后剩余时间是多少，然后把这个"超级鱼塘"里的所有鱼都排序，求出这个超级鱼塘的最大收益。
3. 最后在所有超级鱼塘里取最大的那个值就可以了。而程序，用优先队列则十分容易实现。

优先队列法：代码实现与性能分析

```cpp
typedef pair<int, int> pi;
const int MAX_N  = 100 + 2;
int N, T, w[MAX_N], d[MAX_N], c[MAX_N];

int pq(int k, int t)
{
    priority_queue<pi> PQ;
    for(int i=1; i<=k; i++)
        PQ.push({w[i], d[i]});
    int ans = 0;
    while(t-- > 0) {
        const pi & f = PQ.top();
        ans += f.first;
        PQ.push({max(f.first - f.second, 0), f.second});
        PQ.pop();
    }
    return ans;
}

int main ()
{
    scanf("%d", &N);
    for(int i=1; i<=N; i++) scanf("%d", &w[i]);
    for(int i=1; i<=N; i++) scanf("%d", &d[i]);
    for(int i=1; i< N; i++) scanf("%d", &c[i]);
    scanf("%d", &T);

    int ans = 0, k = 1;
    while(T > 0 && k <= N)
        ans = max(ans, pq(k, T)), T -= c[k], k++;
    printf("%d", ans);
}
```

时间复杂度分析：
最坏每个鱼塘都要合并，
也就是最多会有N个超级鱼塘
每个超级鱼塘（优先队列）里，最多有T个数，
并且一趟遍历得出结果，所以复杂度为$O(TlogT)$
最终程序的时间复杂度为 $O(N*TlogT)$

空间复杂度分析：
本程序使用 pair 存储并实时更新队列值，
所以不用开 N * T 的空间，
最终空间复杂度为 (3 + 2) * N。

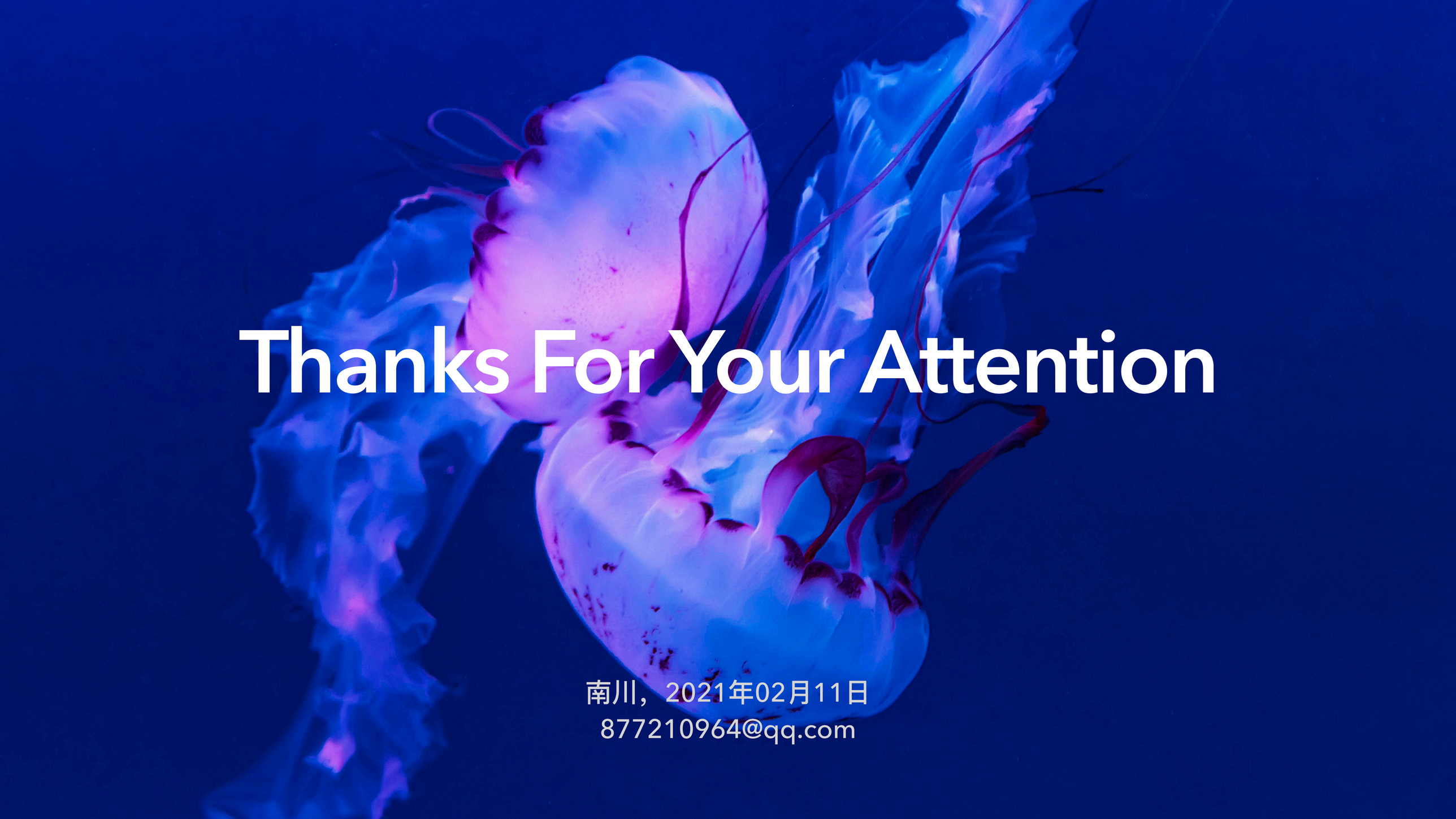
作者：  南川
结果：Accepted

通过了 10/10个数据

运行时间：71 ms

运行空间：220 KB

提交时间：3小时前



Thanks For Your Attention

南川，2021年02月11日

877210964@qq.com