贪心

定义

贪心算法是用计算机来模拟一个**贪心**的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，**并不是所有的时候贪心法都能获得最优解**，所以一般使用贪心法的时候，都要确保自己能**证明其正确性**。

适用范围

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。

正确性证明

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 归纳法：先算得出边界情况（例如$n=1$）的最优解F_1，然后再证明：对于每个n，F_{n+1}都可以由F_n推导出结果。

[洛谷P2240] 部分背包问题

题目描述

阿里巴巴走进了装满宝藏的藏宝洞。藏宝洞里面有 $N(N \le 100)$ 堆金币，第 i 堆金币的总重量和总价值分别是 $m_i, v_i (1 \le m_i, v_i \le 100)$。阿里巴巴有一个承重量为 $T(T \le 1000)$ 的背包，但并不一定有办法将全部的金币都装进去。他想装走尽可能多价值的金币。所有金币都可以随意分割，分割完的金币重量价值比（也就是单位价格）不变。请问阿里巴巴最多可以拿走多少价值的金币？

输入格式

第一行两个整数 N, T。

接下来 N 行，每行两个整数 m_i, v_i。

输出格式

一个实数表示答案，输出两位小数

样例输入

```
4 50
10 60
20 100
30 120
15 45
```

样例输出

```
240.00
```

Solution

由于包的承重有限，因此，如果拿走相同重量的金币，一定是优先拿走**单价**越高的金币。所以正确的做法就是将金币的单价从高到低排序，按照顺序依次将金币装入背包，接下来我们将证明这一点：

1. 由于所有金币的价值都为**正数**且能够**在单价不变的情况下随意分割**，因此背包装的越满，价值一定越高
2. 假设最优解的背包中**没有**放入单价**相对较高**的金币，那么使用这些单价**相对较高**的金币替换单价**相对较低**的金币能使总价值更高，与这是**最优解**的假设矛盾

Code

```
void solve() {
    int n, t; cin >> n >> t;

    // 金币结构体
    struct coin {
        int w, v;
        // 自定义<号，由于我们需要从大到小，因此符号也相反
        bool operator<(const coin& oth) const {
            return v * oth.w > w * oth.v;
        }
    };
    vector<coin> a(n);
    for (auto& v: a) cin >> v.w >> v.v;
    sort(a.begin(), a.end());

    db ans = 0;
    for (int i = 0; (i < n) && t; i++) {
        db take = min(t, a[i].w);
        ans += take * a[i].v / a[i].w; t -= take;
    }

    cout << ans << endl;
}
```

思考：如果金币不可分，还能用贪心吗？如果不能，请举出反例

[洛谷P1223] 排队接水

题目描述

有 n 个人在一个水龙头前排队接水，假如每个人接水的时间为 T_i，请编程找出这 n 个人排队的一种顺序，使得 n 个人的**平均等待时间**最小。

输入格式

第一行为一个整数 $n(1 \leq n \leq 1000)$。

第二行 n 个整数，第 i 个整数 T_i 表示第 i 个人接水的时间 $T_i(1 \leq T_i \leq 10^6)$。

输出格式

输出文件有两行，第一行为一种平均时间最短的排队顺序；第二行为这种排列方案下的平均等待时间（输出结果精确到小数点后两位）。

样例输入

```
10
56 12 1 99 1000 234 33 55 99 812
```

样例输出

```
3 2 7 8 1 4 9 6 10 5
291.90
```

Solution

本题需要求得最小的**平均等待时间**，由于n×平均等待时间 = 等待时间之和，因此我们只需要最小化等待时间之和即可；

而所有人需要等待的总时长可以如下推导得到：

1. 第一个人不需要等待
2. 第二个人需要等待第一个人的接水时间t_1
3. 第三个人需要等待第一个人和第二个人的接水时间$t_1 + t_2$
4. 第四个人需要等待第一个人，第二个人和第三个人的接水时间$t_1 + t_2 + t_3$
5. $\cdots$
6. 第n个人需要等待前$n-1$个人的总接水时间$\sum_{i=1}^{n-1} t_i = t_1 + t_2 + \cdots + t_{n-1}$

因此，所有人的总等待时长s为：

$$s = \sum_{i=1}^{n-1} (n-i)t_i = (n-1) \times t_1 + (n-2) \times t_2 + \cdots + 2t_{n-2} + t_{n-1}$$

可以发现，t_1的系数比较大，t_n的系数比较小（虽然上式没有出现t_n，但我们可以视为出现了一个$0 \times t_n$）若想最小化总时长，我们可以将接水时长**由小到大**排序，接水时长越小的人我们放在越前面，接下来我们将对这个假设进行正确性证明：

假设我们的最优解存在一对$i, j(i < j)$，同时有$t_i > t_j$，如果此时交换i, j两个人的顺序，我们的总时长s会变为：

$$\begin{aligned} s &= s - t_i \times i - t_j \times j + t_i \times j + t_j \times i \\ &= s - i \times (t_j - t_i) - j \times (t_j - t_i) \end{aligned}$$

由于$i, j, t_j - t_i$均为**正**，因此总时长s减小，故原解不是最优解，所以贪心算法成立

Code

```
void solve(){
    int n; cin >> n;
    vector<pair<int, int>> a(n);

    for (int i = 0; i < n; i++) cin >> a[i].first, a[i].second = i + 1;

    sort(a.begin(), a.end());

    ll ans = 0;
    for (int i = 0; i < n; i++) {
        cout << a[i].second << " \n"[i == n - 1];
        ans += (n - 1 - i) * a[i].first;
    }

    cout << (db)ans / n << endl;
}
```

对于解题而言，我们可以进行策略的猜测，同时可以试图构造反例来推翻我们猜测的结论，如果无法构造反例，我们可以大胆地去将算法实现，但无论是解题过程中还是结束后，证明策略的正确性才是做题的重点

[洛谷P1803] 凌乱的yyy

题目背景

快 noip 了，yyy 很紧张！

题目描述

现在各大 oj 上有 n 个比赛，每个比赛的开始、结束的时间点是知道的。

yyy 认为，参加越多的比赛，noip 就能考的越好。

所以，他想知道他最多能参加几个比赛。

由于 yyy 是蒟蒻，如果要参加一个比赛必须善始善终，而且不能同时参加 2 个及以上的比赛。

输入格式

第一行是一个整数 $n(1 \le n \le 10^6)$，接下来 n 行每行是 2 个整数 a_i, b_i $(0 \le a_i < b_i \le 10^6)$，表示比赛开始、结束的时间。

输出格式

一个整数最多参加的比赛数目。

样例输入

```
3
0 2
2 4
1 3
```

样例输出

```
2
```

Solution

如果所有的比赛时间不冲突，那么就可以全部参加了，但实际情况是会有冲突的，**两场**比赛之间的冲突可以归结为以下两种情况：

1. 如图，两场比赛**相交**，此时我们选择**先结束**的那一场，这样我们对后续的比赛选择的余地就不会比另一场小

2. 如图，一场比赛被另一场包含，此时我们选择**被包含**的那一场，这样我们对后续的比赛选择的余地也不会比另一场小

综上所述，在有多场比赛可以同时选择时，我们总会选择**结束时间最早**的那一场比赛

Code

```
void solve(){
    int n; cin >> n;
    vector<pair<int, int>> a(n);
    for (auto& p: a) cin >> p.second >> p.first;

    sort(a.begin(), a.end());

    int now = 0, ans = 0;
    for (auto p: a) {
        auto bg = p.second, ed = p.first;
        if (now > bg) continue;
        ans++; now = ed;
    }

    cout << ans << endl;
}
```

[NOIP2018 提高组] 铺设道路

题目描述

春春是一名道路工程师，负责铺设一条长度为 n 的道路。

铺设道路的主要工作是填平下陷的地表。整段道路可以看作是 n 块首尾相连的区域，一开始，第 i 块区域下陷的深度为 d_i 。

春春每天可以选择一段连续区间 $[L, R]$ ，填充这段区间中的每块区域，让其下陷深度减少 1。在选择区间时，需要保证，区间内的每块区域在填充前下陷深度均不为 0 。

春春希望你能帮他设计一种方案，可以在最短的时间内将整段道路的下陷深度都变为 0 。

输入格式

输入文件包含两行，第一行包含一个整数 $n(1 \leq n \leq 10^5)$，表示道路的长度。 第二行包含 n 个整数，相邻两数间用一个空格隔开，第 i 个整数为 $d_i(0 \leq d \leq 10^4)$ 。

输出格式

输出文件仅包含一个整数，即最少需要多少天才能完成任务。

样例输入

```
6
4 3 2 5 3 5
```

样例输出

```
9
```

Solution

首先，对于某块区域$a[i]$，假设他的深度为m，此时我们至少需要m天才能将其填平

对于两块区域$a[i], a[i+1]$，假设他们的深度分别为x, y，此时我们至少需要$max(x, y)$天才能填平

对于**相邻**且**深度相等**的区域，我们可以将其视为一块区域计算

因此我们可以"贪心"地先填较深的区域，直到它和它**左边**齐平，随后再将它们视为**一块**区域进行后续的计算

实现上，我们发现，$max(a[i]-a[i-1]，0)$就可以代表将较深的区域填到和**左边**齐平需要花费的代价：

1. $a[i]-a[i-1] \leq 0$，代表$a[i]$不低于$a[i-1]$，此时只需要等到某次操作后$a[i-1]=a[i]$，后续操作**带上**$a[i]$即可
2. $a[i]-a[i-1] \geq 0$，代表$a[i]$低于$a[i-1]$，此时我们需要花费$a[i]-a[i-1]$天使得$a[i]$和$a[i-1]$齐平

因此答案可以用$\sum_{i=1}^{n} max(a[i]-a[i-1]，0)$计算得出

特别的，由于最后我们要让道路下陷深度为0，因此我们规定$a[0]=0$

Code

```
void solve(){
    int n; cin >> n;
    int lst = 0, ans = 0;
    for (int i = 0; i < n; i++) {
        int u; cin >> u;
        ans += max(0, u - lst); lst = u;
    }

    cout << ans << endl;
}
```

[洛谷P1106] 删数问题

题目描述

键盘输入一个高精度的正整数 N（不超过 250 位），去掉其中任意 k 个数字后剩下的数字按原左右次序将组成一个新的非负整数。编程对给定的 N 和 k，寻找一种方案使得剩下的数字组成的新数最小。

输入格式

输入两行正整数。

第一行输入一个高精度的正整数 n。

第二行输入一个正整数 k，表示需要删除的数字个数。

输出格式

输出一个整数，最后剩下的最小数。

样例输入

```
175438
4
```

样例输出

```
13
```

Solution

对于序列$a_1, a_2, \cdots, a_n$而言，假设我们删除的位置位于i，则序列会变为$a_1, a_2, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n$

和原序列相比，前i位分别为：

$$a_1, a_2, \cdots, a_{i-1}, a_i$$
$$a_1, a_2, \cdots, a_{i-1}, a_{i+1}$$

可以看出，前i位只有a_i变成了a_{i+1}，如果有$a_i > a_{i+1}$，整个序列的**高位**就会变小，如果$a_i < a_{i+1}$，则整个序列的**高位**就会变大

因为存在一种删除方法，即删除a_n，这样就可以保证每次删除后序列的**高位**都**不变**，因此，我们只会删除$a_i > a_{i+1}$的情况

如果存在多个点$i, j, k, \cdots$，都有$a_i > a_{i+1}, a_j > a_{j+1}, a_k > a_{k+1}, \cdots$，那么我们该选择哪一个进行删除呢（假设$i < j < k < \cdots$）

我们以三个点为例，写下删除这三个点后的前k位：

$$
\begin{aligned}
&a_1, a_2, \cdots, a_{i-1}, a_{i+1}, \cdots, a_j, a_{j+1}, \cdots, a_k, a_{k+1} \\
&a_1, a_2, \cdots, a_{i-1}, a_i, \cdots, a_{j-1}, a_{j+1}, \cdots, a_k, a_{k+1} \\
&a_1, a_2, \cdots, a_{i-1}, a_i, \cdots, a_{j-1}, a_j, \cdots, a_{k-1}, a_{k+1}
\end{aligned}
$$

由于在数的**长度相等**的情况下，高位越小，数越小，又有$a_i > a_{i+1}$，因此删除第i个数最优

接下来我们将贪心的结论推广到多次删除

先以$k = 2$为例，我们假设存在位置$x, y (x < y)$，使得删除a_x, a_y后比删除a_i, a_j后的数**更小**

先假设$x < i$，我们写下删除后的前i位

$$
\begin{aligned}
arr_x &= a_1, a_2, \cdots, a_{x-1}, a_{x+1}, \cdots a_i, a_{i+1} \\
arr_i &= a_1, a_2, \cdots, a_{x-1}, a_x, \cdots a_{i-1}, a_{i+1}
\end{aligned}
$$

由于i是第一位$a_i > a_{i+1}$，因此有$a_x \le a_{x+1}, a_{x+1} \le a_{x+2}, \cdots$，也就是说，$arr_i$的前$i-2$位不会大于$arr_x$，而对于第$i-1$位而言，$a_i < a_{i-1}$，因此我们得到$arr_x > arr_i$，故$i \le x$，而$i < x$的情景我们也证明了删除第$i$位会更小，因此得到$x == i$

由于$x == i$，此时问题从$k = 2$转换成了$k = 1$，以此类推即可

Code

```
void solve(){
    string str; cin >> str;
    int k; cin >> k;

    while (k--) {
        int fd = str.size() - 1;
        for (int i = 1; i < str.size(); i++) {
            if (str[i] < str[i - 1]) { fd = i - 1; break;}
        }
        str.erase(fd, 1);
    }

    while (str.size() && str[0] == '0') str.erase(str.begin());
    cout << (str.size() ? str : "0") << endl;
}
```

[NOIP2004 提高组] 合并果子

题目描述

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过 $n-1$ 次合并之后， 就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为 1 ，并且已知果子的种类 数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如有 3 种果子，数目依次为 1 ， 2 ， 9 。可以先将 1 、 2 堆合并，新堆数目为 3 ，耗费体力为 3 。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为 12 ，耗费体力为 12 。所以多多总共耗费体力 $=3+12=15$ 。可以证明 15 为最小的体力耗费值。

输入格式

共两行。
第一行是一个整数 $n(1 \le n \le 10000)$ ，表示果子的种类数。

第二行包含 n 个整数，用空格分隔，第 i 个整数 $a_i(1 \le a_i \le 20000)$ 是第 i 种果子的数目。

输出格式

一个整数，也就是最小的体力耗费值。输入数据保证这个值小于 2^{31} 。

样例输入

```
3
1 2 9
```

样例输出

```
15
```

Solution

①**最优方案可以表示成一个二叉树。**总代价$\sum_{i=1}^{n} a_i \times depth_i$。其中$depth$是深度，也就是这堆果子在整个过程中被有效合并了几次。

注意：a_i都是叶子结点。非叶子结点都是合并后的产物。

②**最小的两堆一定在最优方案树的最深层。**

这个用反证法。假设有一个最优方案树，其最深层中没有最小的两堆。那么把最小的堆与最深层的某堆互换位置形成新方案，保证新方案存在而且新方案的代价小于原方案。

注意：最深层总是一组（一组有两个）或多组叶子节点，来表示它们被**直接**合并。

③**同层叶子节点互换，对总代价无影响。**

根据①的总代价公式得。可见，最小的两堆，如果在最优方案树最深层中不是即将合并的一组，那么可以无损地替换为一组。

④根据上两步，我们已经明确：最优方案需要**直接**合并当前最小的两堆。现在我们就进行这个操作。**事实是：现在只剩$n-1$堆了。**我们将问题归结成了一个规模大小为$n-1$的子问题，而不需要关注操作之前是什么样的。我们现在只想对这个子问题进行求解，即又回到了①步骤。

Code

```
void solve() {
    int n; cin >> n;

    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i = 0; i < n; i++) {
        int v; cin >> v;
        pq.push(v);
    }

    int ans = 0;
    while (pq.size() >= 2) {
        auto u = pq.top(); pq.pop();
        auto v = pq.top(); pq.pop();

        ans += u + v; pq.push(u + v);
    }

    cout << ans << endl;
}
```

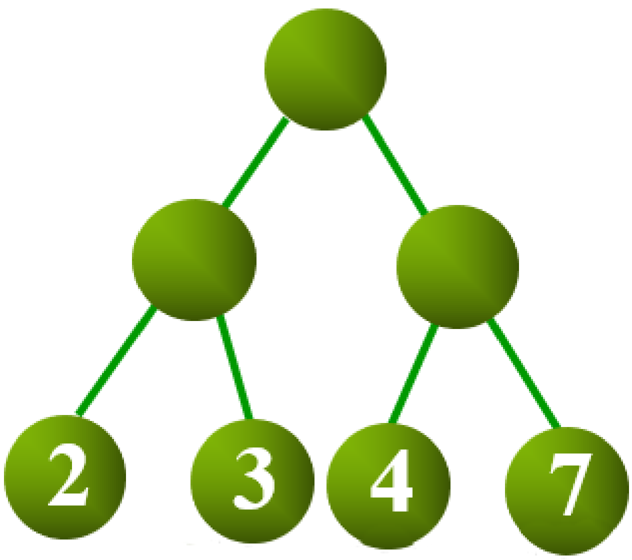
霍夫曼树

树的带权路径长度

设二叉树具有n个带权叶结点，从根结点到各叶结点的路径长度（即**深度**）与相应叶节点权值的乘积之和称为树的带权路径长度

$$\sum_{i=1}^{n} w[i] \times depth[i]$$

例如下面这棵二叉树:



其带权路径长度就为$2 \times 2 + 3 \times 2 + 4 \times 2 + 7 \times 2 = 32$

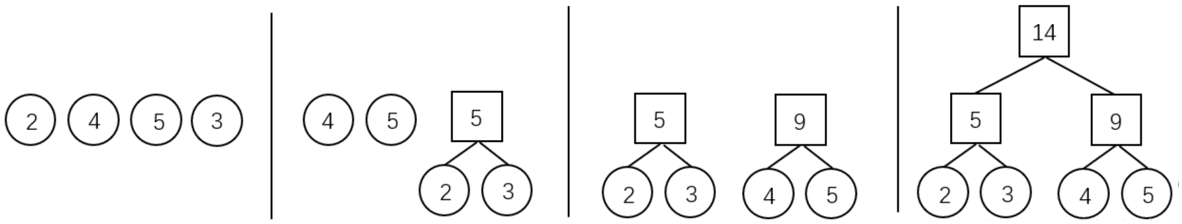
霍夫曼树

在**叶节点权值一定**的情况下，**带权路径长度最小**的二叉树，就被称为**霍夫曼树**

霍夫曼算法

霍夫曼算法用于构造一棵霍夫曼树，算法步骤如下：

1. **初始化**：由给定的n个权值构造n棵只有一个根节点的二叉树，得到一个二叉树集合F。
2. **选取与合并**：从二叉树集合F中选取根节点权值**最小的两棵**二叉树分别作为左右子树构造一棵新的二叉树，这棵新二叉树的根节点的权值为其左、右子树根结点的权值和。
3. **删除与加入**：从F中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到F中。
4. 重复 2、3 步，当集合中只剩下一棵二叉树时，这棵二叉树就是霍夫曼树。



霍夫曼编码

在进行程序设计时，通常给每一个字符标记一个单独的代码来表示一组字符，即**编码**，常见的编码有 `ASCII`，`UTF-8`，`UTF-16`等。

在进行二进制编码时，假设所有的代码都**等长**，那么表示n个不同的字符需要$\lceil log_2 n \rceil$位，称为**等长编码**。

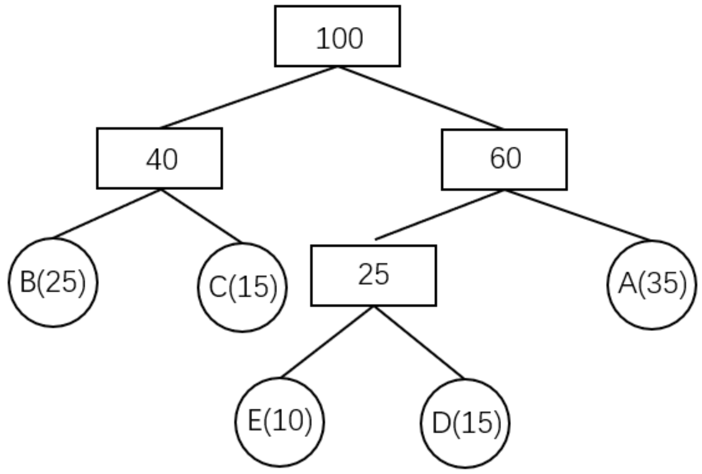
如果每个字符的**使用频率相等**，那么等长编码无疑是空间效率最高的编码方法，而如果字符出现的频率不同，则可以让频率高的字符采用尽可能短的编码，频率低的字符采用尽可能长的编码，来构造出一种**不等长编码**，从而获得更好的空间效率。

在设计不等长编码时，要考虑解码的**唯一性**，如果一组编码中任一编码都不是其他任何一个编码的**前缀**，那么称这组编码为**前缀编码**，其保证了编码被解码时的唯一性。

霍夫曼树可用于构造 **最短的前缀编码**，即 **霍夫曼编码（Huffman Code）**，其构造步骤如下：

1. 设需要编码的字符集为：$d_1, d_2, d_3, \cdots, d_n$，他们在字符串中出现的频率为：$w_1, w_2, w_3, \cdots, w_n$。
2. 以作$d_1, d_2, d_3, \cdots, d_n$为叶结点，$w_1, w_2, w_3, \cdots, w_n$作为叶结点的权值，构造一棵霍夫曼树。
3. 规定哈夫曼编码树的左分支代表0，右分支代表1，则从根结点到每个叶结点所经过的路径组成的01序列即为该叶结点对应字符的编码。

一棵构建好的霍夫曼树如下图所示：



字符	频率	编码
A	35	11
B	25	00
C	15	01
D	15	101
E	10	100

后悔贪心

后悔贪心的思路是无论当前的选项是否最优都**接受**，然后进行比较，如果选择之后不是最优了，则反悔，**舍弃**掉这个选项；否则，正式接受。如此往复。

[USACO09OPEN] Work Scheduling G

题目描述

约翰有太多的工作要做。为了让农场高效运转，他必须靠他的工作赚钱，每项工作花一个单位时间。 他的工作日从 0 时刻开始，有 10^9 个单位时间。在任一时刻，他都可以选择编号 1 到 N 的 $N(1 \leq N \leq 10^5)$ 项工作中的任意一项工作来完成。 因为他在每个单位时间里只能做一个工作，而每项工作又有一个截止日期，所以他很难有时间完成所有N个工作，虽然还是有可能。 对于第 i 个工作，有一个截止时间 $D_i(1 \leq D_i \leq 10^9)$，如果他可以完成这个工作，那么他可以获利 $P_i(1 \leq P_i \leq 10^9)$. 在给定的工作利润和截止时间下，约翰能够获得的利润最大为多少?

输入格式

第一行一个整数N

第二行到第$N+1$行每行两个整数，D_i, P_i

输出格式

一行一个整数，约翰能够获得最大利润

样例输入

```
3
2 10
1 5
1 7
```

样例输出

```
17
```

Solution

首先，我们将所有工作按照**截止时间**排序，并我们用一个**小顶堆**来维护我们已经干了的工作

随后我们依次遍历所有的工作，如果它的**截止时间**大于堆的大小，代表我们在它的截止时间前还有**空闲**，那么我们便无条件接受它，如果它的**截止时间**小于等于堆的大小，我们就要看是否存在一件比它利润低的工作，如果存在，则代表我们可以用当前这件工作去**替换**它

由于已经在堆里的工作截止时间是小于等于当前这件工作的，因此这种**替换**在时间上一定能够得到满足

Code

```
void solve(){
    int n; cin >> n;

    struct job{
        int d, p;
        bool operator<(const job& oth) const{
            return p > oth.p;
        }
    };

    priority_queue<job> pq;
    vector<job> a(n);
    for(auto& [d, p]: a) cin >> d >> p;

    sort(all(a), [](const job& a, const job& b){
        return a.d < b.d;
    });

    for(const auto&[d, p]: a){
        if(pq.size() < d) {pq.emplace(d, p); continue;}
        auto [topd, topp] = pq.top();
        if(topp < p) {pq.pop(); pq.emplace(d, p);}
    }

    ll ans = 0;
    while (!pq.empty()) {ans += pq.top().p; pq.pop();}
    cout << ans << endl;
}
```

[国家集训队] 种树

题目描述

A城市有一个巨大的圆形广场，为了绿化环境和净化空气，市政府决定沿圆形广场外圈种一圈树。

园林部门得到指令后，初步规划出 n 个种树的位置，顺时针编号 1 到 n。并且每个位置都有一个美观度 A_i，如果在这里种树就可以得到这 A_i 的美观度。但由于 A 城市土壤肥力欠佳，两棵树决不能种在相邻的位置（i 号位置和 $i+1$ 号位置叫相邻位置。值得注意的是 1 号和 n 号也算相邻位置）。

最终市政府给园林部门提供了 m 棵树苗并要求全部种上，请你帮忙设计种树方案使得美观度总和最大。如果无法将 m 棵树苗全部种上，给出无解信息。

输入格式

输入的第一行包含两个正整数 n，$m(1 \leq m \leq n \leq 2 \times 10^5)$。

第二行 n 个整数，第 i 个代表 $A_i(-1000 \leq A_i \leq 1000)$。

输出格式

输出一个整数，表示最佳植树方案可以得到的美观度。如果无解输出 `Error!`。

样例输入

```
7 3
1 2 3 4 5 6 7
```

样例输出

```
15
```

样例输入

```
7 4
1 2 3 4 5 6 7
```

样例输出

```
Error!
```