2018 KAIST RUN Spring Contest 풀이

For International Readers: English editorial starts on page 13.

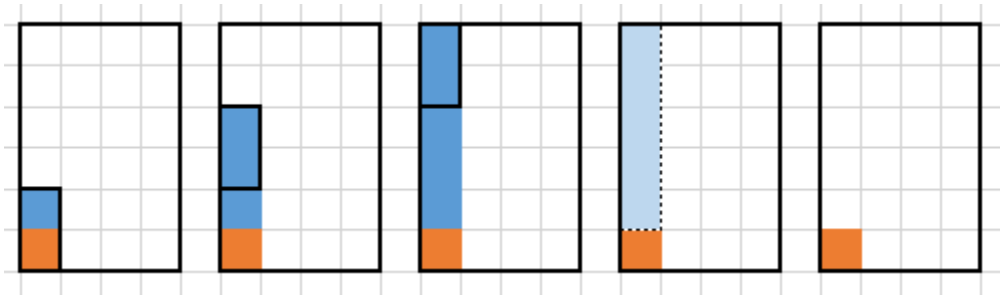
Credits

Problem	Author	Preparation	Editorial
P	강한필	강한필	강한필
Q	구재현	강한필	구재현
R	유재민	유재민	유재민
S	이태현	이태현	구재현
T	구재현	구재현	구재현
U	구재현	구재현	구재현
V	구재현	강한필	구재현
W	구재현	유재민	구재현
X	조민규	조민규	조민규
Y	강한필	강한필	구재현
Z	구재현	현지훈	구재현

PuyoPuyo

Subtask 1 (63 points)

제한이 작고, 어차피 최종 그리드에는 크기 4 이상의 컴포넌트가 없을 테니, 하나의 블록을 차근차근 쌓아 올리기만 해도 문제를 해결할 수 있습니다. 하나의 블록을 쌓는 것은 어떻게 해야 할 까요? 다음과 같은 그림을 살펴봅시다.



이 간단한 방법을 사용하면 3번의 연산으로 하나의 블록을 쌓을 수 있으니, 많아야 48번 안에 문제를 해결할 수 있습니다. 하나의 연산을 할 때 한 줄 한 줄 쌓아 올라가야 한다는 것만 유의하면 됩니다.

Subtask 1/2 (100 points)

이 방법을 그대로 사용하면 연산의 수가 너무 많지만, 이를 줄일 수 있습니다. 어떠한 열의 높이가 짝수일 때는 $\frac{B}{2}$ 번의 연산으로 그대로 그리드를 채울 수 있다는 것을 관찰합시다. 이제 문제가 되는 것은 어떠한 열의 높이가 홀수일 때 인데, 높이가 홀수인 그리드에 대해서 가장 아래에 있는 블록을 위의 방법으로 채운 후, 짝수 높이의 그리드를 채우는 방식으로 나머지 블록을 넣어주면 됩니다. 이 방법으로 최대 240번의 연산만을 사용하여 문제를 해결할 수 있습니다.

QueryreuQ

Subtask 1 (36 points)

Subtask 1은 비교적 단순한 알고리즘으로 해결할 수 있습니다. 일차적으로, 어떠한 문자열에서 맨 뒤에 있는 문자가 추가 / 삭제된다는 것은, 맨 마지막 문자를 포함하는 팰린드롬만이 사라지거나 추가된다는 것을 뜻합니다. 맨 마지막 문자를 포함하는 팰린드롬은 많아야 $O(Q)$개이니, 이들에 대해서 일일이 팰린드롬인지 아닌지를 단순하게 판별해 주면, 간단한 $O(Q^3)$ 알고리즘을 유도할 수 있습니다.

Subtask 1/2 (100 points)

Subtask 2에서도 위와 동일한 접근 방법을 사용하나, 동적 계획법 (Dynamic Programming)을 사용하여 각각의 부분 문자열이 팰린드롬인지 아닌지를 $O(1)$에 판별합니다. $DP[i,j]$를 $[i,j]$ 구간의 문자열이 팰린드롬인지 아닌지를 저장하는 불리언 배열이라고 생각합니다. $DP[i,i]$나 $DP[i,i+1]$은 기저 조건이고, 그 외의 경우 $DP[i,j] = (DP[i+1,j-1]$ & $S[i] == S[j])$ 와 같은 점화식이 성립합니다. 문자가 추가될 때에는 이 정의에 맞춰서 DP 배열을 갱신해 줄 수 있고, 삭제될 때는 단순히 한 줄을 무시해 주면 됩니다. 이렇게 되면 각 쿼리 당 DP 배열 관리에 $O(Q)$, 답 변화에 $O(Q)$ 가 들어서 $O(Q^2)$에 문제가 해결됩니다.

여담으로, 이 문제를 $O(QlgQ)$에 해결하는 알고리즘을 담은 2015년 논문이 존재합니다. `https://arxiv.org/abs/1506.04862`

Recipe

Subtask 1 (17 points)

i일에 식재료를 사서 j일에 요리한다면 이 과정을 구간 $[i,j]$로 생각할 수 있다. 이 구간은 가중치 $(F_i - j + i) \times C_j$를 가진다. 그러면 구간 $[1,N]$을 여러개의 연속된 구간들로 나누어 구간들의 가중치의 합을 최대화 하는 문제를 풀면 된다. 동적계획법을 사용하면 다음과 같은 점화식을 이용해 $O\left(N^2\right)$에 풀 수 있고 Subtask 1의 점수를 얻는다.

$$D_i = \min_{j=i}^{N}\{(F_i - j + i)\, C_j + D_{j+1}\} \quad if \quad L_j \leq F_i - j + i \tag{1}$$

Subtask 2 (20 points)

F_i가 음수로 떨어지는 경우는 절대 최적해가 될 수 없으므로 항상 음식의 신선도가 음이 아닌 정수일 때 요리를 해야 한다. 따라서, L_i가 항상 0이면 $L_j \le F_i - j + i$를 항상 만족해서 이 조건을 신경 쓸 필요가 없다.

Convex Hull Trick (CHT)을 이용하기 위해서 점화식을 바꾸면 다음과 같이 된다.

$$D_i = \min_{j=i}^{N} \{(F_i + i)\, C_j - jC_j + D_{j+1}\} \quad if \quad L_j + j \le F_i + i \tag{2}$$

직선들($f_j(x) = C_j x - jC_j + D_{j+1}$)의 기울기 C_j가 단조 감소하기 때문에 볼록다각형의 upper bound를 관리할 수 있다. (점화식을 N에서 1방향으로 돌려야 문제를 쉽게 풀 수 있다. 이 경우에는 C_j가 단조 감소함에 주의하자.) if 문을 생각하지 않고 CHT 최적화를 사용하면 Subtask 2의 점수를 얻는다.

Subtask 1/2/3 (100 points)

if 문을 관찰하면 $L_j + j \le F_i + i$를 만족하는 $f_j(x)$들만 D_i를 구하기 위해 사용할 수 있음을 알 수 있다. 이는 직선 $f_j(x)$이 사실 구간 $[L_j + j, \infty)$에서 정의된 반직선임을 의미한다. 반직선 $f_{i+1}, f_{i+2}, \cdots, f_N$들의 upper bound U_{i+i}는 증가함수이다. U_{i+1}에 f_i를 추가해서 U_i를 만들 때 두 가지 경우가 있다.

1. $U_{i+1} \ge f_i \quad for \quad all \quad x \in \left[\min_{j=i}^{N} (L_j + j), \infty \right)$

   $x = \min_{j=i}^{N} (L_j + j)$일 때, $f_i(x)$와 $U_{i+1}(x)$ 값을 비교하면 된다.

   이 경우에는 f_i가 U_{i+1}의 정의역에서 모두 U_{i+1}보다 크지 않기 때문에 U_{i+1}보다 왼쪽에 있는 f_i의 일부분만 U_{i+1}에 추가해서 U_i를 만들 수 있다.

2. $\exists x_0 \quad s.t \quad U_{i+1}(x) < f_i(x) \quad if \quad x < x_0 \quad and \quad U_{i+1}(x) > f_i(x) \quad if \quad x > x_0$

   위의 경우가 아니면, $[L_i + i, x_0)$에서 f_i가 U_{i+1}보다 큰 x_0가 존재한다. 이분탐색으로 $L_i + i$를 U_{i+1}에서 찾고 오른쪽으로 이동하면서 f_i가 U_{i+1}보다 클 때까지 U_{i+1}에 있는 직선들을 없애고 f_i를 추가하면 새로운 upper bound U_i를 만들 수 있다.

이렇게 하면 N번의 갱신을 $O(N \log N)$에 할 수 있다. 쿼리도 이분탐색으로 $x = F_i + i$일 때, $U_i(x)$ 값을 구하면 되므로 총 $O(N \log N)$에 문제를 해결할 수 있다.

이 외에 이 문제를 풀 수 있는 다양한 방법이 있다.

- Li Chao segment tree를 사용하면 C_i가 단조증가한다는 조건이 없어도 $O\left(N \log^2 N\right)$에 문제를 풀 수 있다.
- 분할 정복을 사용하면 반직선을 관리해 주는 자료구조 (Stack) 만을 사용해서 $O(N \log N)$ 에 해결할 수 있다. C_i가 단조증가한다는 조건이 없어도 $O\left(N \log^2 N\right)$에 문제를 풀 수 있으며, 복잡한 자료구조를 전혀 사용하지 않는 풀이이다.

- $L_i + i$의 범위가 $[1, 300, 000]$이므로 세그먼트 트리나 펜윅 트리에 CHT를 넣으면 $O(Nlog^2N)$에 풀 수 있다. 갱신할 때에는 반직선의 맨 왼쪽 점($L_i + i$)을 세그먼트 트리의 인덱스로 해서 $O(\log N)$개의 CHT에 추가하고, $x = (F_i + i)$에 대해 쿼리를 부를 때에는 $[1, x]$구간에 대해서 $O(\log N)$개의 CHT의 값 중 최댓값을 구하면 된다.

세 방법 모두 시간 제한 안에 문제를 해결할 수 있다.

Segmentation

이 문제는 "Map" 과 같은 기본 자료 구조들을 사용하면 쉽게 해결할 수 있는 문제입니다. 각각의 유저에 대해, 해당 유저의 (r, f) 정보를 아는 것이 이 문제의 핵심인데, 이는 각각의 유저의 이름을 Key로 하고, 해당 유저가 마지막으로 등장한 시간, 그리고 해당 유저가 등장한 횟수를 Value로 하는 Map을 각각 만들어 주면 해결할 수 있습니다. 이는 C++ `map`, Java `TreeMap` 등을 사용하면 충분합니다. Map을 사용하지 않고 단순 배열에 주어진 데이터들을 저장하거나, 최악 접근 시간 복잡도가 $O(n)$ 인 Hash Map을 사용하면 시간 초과가 날 수도 있습니다.

해당 유저의 (r, f) 정보를 알면, 문제를 어렵지 않게 해결할 수 있습니다. 각각의 케이스를 if-else로 처리하는 것보다는, 배열 등을 사용해서 주어진 (r, f) 쌍에 대한 출력 결과를 바로 찾을 수 있게끔 하는 것이 구현에 편리합니다.

Touch The Sky

Subtask 1 (22 points)

이 문제를 해결하기 위해서 가장 편리한 관찰은, 바로 이 문제가 실제로는 "스케줄링" 에 관한 문제라는 것을 깨닫는 것입니다. 일반적인 Deadline Scheduling 상황을 생각하고 이 문제를 변형해 봅시다. 풍선을 터트리는 것을 "작업을 처리함" 에 비유하고, 고도를 올리는 것을 "시간을 사용함" 으로 비유할 수 있습니다. 어떠한 작업을 수행하기 위한 데드라인은, 풍선을 불기 위한 고도(시간) 제한 + 풍선이 띄우는 높이(사용하는 시간) 이라고 생각할 수 있습니다. 고로 우리가 해결하는 문제는.

- 0시부터 작업을 시작하며,
- i번 작업을 처리하는 데 D_i 시간이 걸리며, 일을 처리하기까지의 데드라인은 $D_i + L_i$ 이다.
- 최대한 많은 작업을 처리해야 한다.

라는 형태의 최적화 문제로 환원할 수 있습니다.

최대한 많은 작업을 처리해야 하는 경우를 생각하지 않고, 그저 **모든 작업** 을 처리할 수 있는지 없는지를 Yes / No로 판별하는 문제를 생각해 봅시다. 이 때 모든 작업을 처리하는 가장 좋은 방법은, 일을 데드라인이 작은 순서대로 하는 것입니다 (Deadline first scheduling). 데드라인이 늦은 것을 이른 것보다 먼저 해서 얻을 수 있는 이득이 없다는 점을 보이면 Exchange argument를 사용하여 이 알고리즘의

optimality를 증명할 수 있습니다.

Subtask 1은 제한 조건이 작기 때문에 이 관찰만으로 문제를 바로 해결할 수 있습니다. 특정한 부분집합을 고정 시키고, 해당 부분집합 안에 있는 작업을 모두 처리할 수 있는 지 확인하면 됩니다. 모든 작업이 데드라인 $(D_i + L_i)$ 순으로 정렬되어 있음을 가정하면, 부분집합 안에 있는 작업을 모두 처리할 수 있는 지는 데드라인 순서대로 직접 해 보면 쉽게 $O(N)$에 판별할 수 있습니다. 고로 총 시간 복잡도 $O(2^N \times N)$ 에 문제가 해결됩니다.

Subtask 1/2 (55 points)

위 알고리즘을 동적 계획법을 사용해서 최적화하면 Subtask 2를 해결할 수 있습니다. 편의상 이제 모든 작업을 데드라인 순으로 정렬하고 생각합시다. 동적 계획법을 사용하는 데 있어 가장 중요한 관찰은, 어떠한 풍선을 터뜨리는 상태에서 중요한 것은 "몇 개의 풍선을 터뜨렸는지", 그리고 "현재 높이가 어디인지" 라는 두 가지 뿐이라는 것입니다. i번째 풍선까지를 보았을 때, 이 중 j개의 풍선을 터뜨릴 수 있는 여러 방법 중, 우리가 관심있는 것은 현재의 높이를 최소로 한 풍선 터뜨리기 방법 뿐입니다. 이제 $DP[i][j] = (i$ 번째 풍선까지 중에서 j개의 풍선을 터뜨렸을 때 최소 높이) 라는 점화식을 생각해 봅시다. 선택지는 두가지 뿐입니다.

- Case 1. $DP[i-1][j]$: i번째 풍선은 항상 무시할 수 있음.
- Case 2. $DP[i-1][j-1]$: 만약 $DP[i-1][j-1] + D[i] \leq D[i] + L[i]$ 라면 i번째 풍선을 터뜨릴 수 있음. 고로 이 때 $DP[i][j]$를 갱신해 줄 수 있음.

최종적으로 터뜨릴 수 있는 풍선의 개수는 $DP[n][*]$를 보면 알 수 있습니다. 이 알고리즘의 시간 복잡도는 $O(N^2)$입니다.

Subtask 1/2/3 (100 points)

직관적인 접근

i번째 풍선까지 중 최대로 터뜨릴 수 있는 풍선들의 부분 집합을 귀납적으로 계산했다고 합시다. 이 집합에 $i+1$번째 풍선을 넣을 수 있다면, 이 풍선을 넣지 않을 이유가 없습니다. 문제는 $i+1$번째 풍선을 넣었을 때 데드라인이 초과해 버리는 식의 문제가 발생하는 경우입니다. 이 때 우리는 아예 $i+1$번째 풍선을 넣지 않거나, 다른 풍선들을 몇 개 제거한 후 $i+1$번째 풍선을 넣는 등의 선택을 해야 합니다.

이제 조건을 조금 더 강하게 바꾸어 봅시다. i번째 풍선까지 중에서 최대로 터뜨릴 수 있고, 만약 그러한 것이 여러 개 있다면 그 중 **최종 고도를 최소화하는** 풍선들의 부분 집합을 귀납적으로 계산하였다고 합시다. 이 상황에서 $i+1$번째 풍선을 넣는 선택의 시기가 왔다고 생각해 봅시다. 풍선을 그냥 부분 집합에 넣을 수 있다면, 이것이 부분 집합의 크기를 최대화하는 유일한 방법이니 그대로 진행하면 됩니다. 문제는 부분 집합에 이 풍선을 그냥 넣을 수 없을 때입니다. 이러한 상황에서는 $i+1$번째 풍선을 넣고 다른 풍선을 제거함을 통해서 최종 고도를 최소화하는 시도를 해 봐야 합니다. 이 때 가장 제거하기 적절한 풍선은 고도를 최대화하는 풍선이 되기 때문에, (만약 풍선을 제거하는 게 이득을 준다면) 이

풍선을 제거해 주면 문제를 해결할 수 있습니다.

이 모든 알고리즘은 $O(nlgn)$ 에 Heap을 사용해서 아주 짧게 구현할 수 있습니다. 하지만, 최종 고도를 최소화하는 부분 집합을 처음부터 새로 만들 필요 없이, 위와 같은 방법으로만 구성해도 된다는 사실은 엄밀히 증명되지 않았습니다. 아래 단락에 이에 대한 엄밀한 증명을 제시합니다. 이 증명 내용은 Subtask 2의 동적 계획법 알고리즘과 거의 비슷하니, 이를 따라가기 위해서는 Subtask 2의 알고리즘을 완벽히 이해하는 것이 좋습니다.

엄밀한 증명

모든 $0 \le i \le N, 0 \le j \le i$ 에 대해서, $S_{i,j}$ 를 길이 j의 풍선 부분 집합으로 정의합니다. 이 부분 집합은 앞의 i개의 풍선 중 j개의 풍선을 담고 있으며, 모두 데드라인 순서대로 터뜨릴 수 있고, 그러한 부분 집합이 여러 개 있다면 그 중 고도 합을 최소로 하는 부분집합입니다. 만약 그러한 부분집합이 없다면 정의되지 않는다고 합시다.

Lemma 1. $S_{0,0} = \emptyset$ 이며, 모든 $1 \le i \le n, 0 \le j \le i$ 에 대해서, $S_{i,j}$가 될 수 있는 후보는 많아야 두 가지이다. 이 중 합이 최소인 후보를 선택하면 된다.

- $j > 0$일 때, $S_{i-1,j-1}$이 정의되고, $S_{i-1,j-1} \cup \{A_i\}$ 를 데드라인 순서대로 선택할 수 있다면 이것이 후보가 된다. (Choice 1)
- $S_{i-1,j}$가 정의된다면 이것이 후보가 된다. (Choice 2)

Proof. 쉬운 수학적 귀납법. □

Remark. 이 Lemma는 Subtask 2를 해결하는 동적 계획법을 그대로 옮겨 적은 것이다.

이 다음 Lemma는 모든 연산을 Heap을 사용해서 할 수 있는 이유를 그대로 보여주는, 가장 강력한 Lemma입니다.

Lemma 2. $S_{i,j+1}$이 정의된다면 $S_{i,j}$가 정의되며, $S_{i,j} \subset S_{i,j+1}$.

Proof. $S_{i,j+1}$이 정의될 경우 $S_{i,j}$가 정의되는 것은 자명합니다. 왜냐하면 $S_{i,j+1}$ 에서 어떤 원소를 제거하더라도 항상 데드라인 순으로 처리할 수 있기 때문입니다. 고로, $S_{i,j}$가 정의되는 최대 j를 L_i라 표기합시다. 이제 $S_{i,j} \subset S_{i,j+1}$ 임을 i에 대한 수학적 귀납법으로 증명합니다.

$i = 0$일 때는 $S_{0,0}$만 제대로 정의되니 자명합니다. $i > 0$일 때를 생각해 봅시다. 귀납 가정에 의해서 모든 $0 \le j < L_{i-1}$에 대해 $S_{i-1,j} \subset S_{i-1,j+1}$ 이 만족합니다. 한편, $S_{i-1,L_{i-1}}$은 데드라인 순서대로 터뜨릴 수 있기 때문에, 여기서 어떠한 원소들을 빼더라도 항상 데드라인 순서대로 터뜨릴 수 있습니다. 고로 $S_{i-1,j+1}$에서 $S_{i-1,j}$를 만들 때 고도를 최대화하는 원소를 항상 제거할 수 있습니다. 이 때문에, 만약 $F(S)$ 를 집합 S에 있는 원소들의 고도 합이라고 정의하면, $F(S_{i,j+1}) - F(S_{i,j}) \le F(S_{i,j+2}) - F(S_{i,j+1})$ 을 만족합니다.

이제 증명하게 될 것은, $S_{i,j}$ 를 귀납적으로 만들어 나가는 과정에서, 앞에 있는 원소들은 Choice 2을 택하다가, 어느 순간이 존재해서 이 순간부터 Choice 1을 택하게 됨을 증명합니다. 즉, $S_{i,*}$ 을 순서대로 나열하면 $S_{i-1,0}, S_{i-1,1}, \cdots, S_{i-1,j}, S_{i-1,j} \cup \{A_i\}, S_{i-1,j+1} \cup \{A_i\}, \cdots$ 와 같은 형태가 된다는 것입니다. 만약에 이 성질을 만족하게 되면, 모든 인접한 원소들에 대해서 $S_{i,j} \subset S_{i,j+1}$ 을 만족하니 증명이 끝납니다.

이를 증명하기 위해서는 $S_{i,j} = S_{i-1,j-1} \cup \{A_i\}, S_{i,j+1} = S_{i-1,j+1}$ 을 만족하는 j가 없음을 귀류법에 보입니다. 이를 보여주면 한번 A_i를 추가하게 된 이후 다시 원래대로 돌아갈 일이 없음을 알 수 있기 때문입니다. 이는 위 정보들을 조합하면 어렵지 않게 증명할 수 있습니다. 만약 Choice 1과 Choice 2가 같은 합을 가진다면 Choice 2를 고른다고 합시다. 그렇다면 :

$F(S_{i-1,j-1}) + F(\{A_i\}) < F(S_{i-1,j})$
$F(S_{i-1,j+1}) \leq F(S_{i-1,j}) + F(\{A_i\})$
$F(S_{i-1,j+1}) - F(S_{i-1,j}) \leq F(\{A_i\}) < F(S_{i-1,j}) - F(S_{i-1,j-1})$
$F(S_{i-1,j}) - F(S_{i-1,j-1}) \leq F(S_{i-1,j+1}) - F(S_{i-1,j})$

가정에 모순이니, 증명이 종료됩니다. □

Lemma 2를 사용하면, 위 알고리즘이 실제로는 S_{i,L_i} 를 귀납적으로 관리하는 알고리즘임을 깨달을 수 있습니다. 만약에 $L_{i-1} + 1 = L_i$를 만족하면, Heap에 A_i 원소를 넣고 넘어가게 됩니다. 그렇지 않다면, $S_{i-1,L_{i-1}}$ 를 고르거나 $S_{i,L_{i-1}-1} \cup \{A_i\}$ 을 고르는 두 가지 선택 중 하나를 하게 됩니다. 이 때 $S_{i-1,L_{i-1}-1}$은 $S_{i-1,L_{i-1}}$ 에서 최대 크기 원소를 골라 지워준 꼴이 되니, 쉽게 계산해 줄 수 있습니다. 이 둘 중 최적의 집합을 판단하면, S_{i,L_i}를 계산해 줄 수 있고, 이 알고리즘은 위에서 설명한 "직관적인 알고리즘"과 동일합니다.

United States of Eurasia

Subtask 1 (20 points)

Subtask 1은 동적 계획법으로 해결할 수 있습니다. 모든 점이 x좌표 순으로 정렬되어 있다고 가정합시다. $Cost[i, j]$ 를 $[i, j]$ 구간에 있는 점 쌍 중 가장 거리가 먼 점 쌍의 거리라 정의합시다. 편의상 $X_0 = -1$이라 두었을 때, DP 테이블을 다음과 같이 정의할 수 있습니다.

$DP[i, j] = min_{k<j, X_k \neq X_{k+1}}(max(DP[i-1, k], Cost[k+1, j]))$

시간 복잡도는 $O(N^3)$입니다. 시간 제한이 매우 매우 널널해서, $Cost[*, *]$를 $O(N^4)$에 채우더라도, 효율적으로만 채우면 시간 안에 통과가 됩니다. 하지만 이 접근으로는 시간 복잡도를 더 줄이기 쉽지 않습니다.

Subtask 1/2 (40 points)

"최댓값을 최소화" 하는 유형의 문제에서는, 문제를 결정 문제로 바꾸어서 답에 대한 이분 탐색을 하는 테크닉이 잘 알려져 있습니다. 예를 들어서, 지금까지 푸는 문제가 "최대 k개의 partition을 사용해서 분열도의 최댓값 M^2를 최소화하여라" 라는 최소화 문제였다면, 이를 변형해서 "각 partition의 분열도를 M^2 이하로 유지시키면서 k개 이하의 partitioning이 가능한가?" 와 같은 결정 문제로 변환하는 것입니다. M^2에 대한 이분 탐색이 가능하기 때문에, 이 결정 문제를 빠르게 해결 할 수 있다면, 이 결정 문제를 60번 반복해서 해결함으로써 문제를 풀 수 있습니다.

일단은 서브태스크 2만 풀면 되니, 이 결정 문제를 $O(N^2)$ 정도에 해결하기만 해도 됩니다. 이는 어렵지 않은데, 현재 그룹의 X_i가 가장 작은 원소에 대해서, 해당 원소가 가질 수 있는 가장 긴 구간을 찾아준 후, 이 구간을 전체 문제에서 제거하는 것을 k번 반복하면 됩니다. 다시 말해, 현재 문제에서 X_i가 가장 작은 원소를 고른 후, 크기 1의 partition을 만들어 주고, 거리가 M^2를 넘지 않을 때까지 이 partition의 크기를 최대한 늘려준 후, 이 partition을 제거해 주는 것입니다. 이를 $O(N^2)$에 구현하면 40점을 얻을 수 있습니다.

Subtask 1/2/3 (100 points)

전체 문제를 해결하기 위해서는, 먼저 문제의 기하적인 성질을 찾아낸 이후, 이 기하적인 성질을 사용해서 시간 복잡도를 줄이는 시도가 필요합니다. 첫번째로 할 수 있는 생각은, 어떠한 구간 $[S, E]$에 대한 분열도 - 즉, 가장 먼 두 점 쌍을 $O(E - S + 1)$에 찾을 수 있다는 점입니다. 이는 두 가지 방법을 조합하면 되는데,

- 1. 구간의 컨벡스 헐 (Convex Hull) 구하기 : 컨벡스 헐은 일반적으로 $O(N \log N)$ 의 시간 복잡도를 사용하는 Graham Scan을 사용하지만, 여기서는 구간에 있는 원소들이 X_i 순으로 정렬되어 있으니 $O(N)$ 시간 복잡도의 Andrew's Monotone Chain을 사용할 수 있습니다.
- 2. 컨벡스 헐에서 가장 먼 두 점 구하기 : 이는 Rotating Callipers 방법을 사용하여서 $O(N)$에 해결될 수 있는 잘 알려진 문제입니다. 이에 대해서는 `http://junis3.tistory.com/6`과 `http://codeforces.com/blog/entry/48868` 를 참고하십시오.

하지만, 이 방법을 알았다 하더라도 위에 있는 $O(N^2)$ 보다 빠른 방법을 찾기 쉽지 않아 보입니다. 각각의 시작점에 대해서 끝점을 이분 탐색으로 찾는 $O(KN \log N)$ 방법을 쓰고 싶어질 수 있으나, 시간 초과를 받게 됩니다. 만약에 해당 시작점에 대응되는 끝점이 충분히 멀다면 위 방법도 유용할 수 있으나, 해당 시작점에 대응되는 끝점이 너무나 가깝다면 위 알고리즘은 지나치게 넓은 구간을 탐색하는 낭비를 하기 때문입니다.

이러한 낭비를 막기 위해서 생각해 볼 수 있는 아이디어는, 어떠한 시작점에서 끝 점을 찾을 때, 답이 될 끝점과의 거리에 비례하는 변형된 이분 탐색을 해 보는 것입니다. 즉, 이분 탐색에서 소모된 시간이 전체 N이 아니라 끝점과의 거리 L에 비례하게 된다면, $(L_1 + L_2 + \cdots + L_k) = N$ 이 되므로 N에

비례하는 알고리즘을 찾을 수 있게 됩니다.

다행이 이러한 이분 탐색은 가능합니다. 어떠한 시작점 S에 대해서 $[S, S+2^{T-1}-1]$ 구간은 partition이 가능하지만, $[S, S+2^T-1]$는 partition이 불가능한 최소 T를 찾을 수 있습니다. 이제, 끝점의 구간을 $[S+2^{T-1}-1, S+2^T-1]$ 로 정해놓고 이 사이에서 이분 탐색을 해 봅시다. 이렇게 될 경우 이분 탐색은 총 $O(\log N)$ 번 구간의 가장 먼 두 점에 대한 질의를 날리는 데, 질의를 날릴 때 사용한 구간의 길이가 실제 끝점과의 거리의 2배를 넘지 않음을 알 수 있습니다. 이 방법을 사용하면, 각 이분 탐색에서 끝점과의 거리가 L일 경우 $O(L \log N)$ 시간 안에 끝점을 찾아낼 수 있고, 고로 각 이분 탐색에서 $O(N \log N)$ 시간에 문제를 해결할 수 있습니다. 결국, 총 $O(N \log N \log X)$에 전체 문제가 해결됩니다.

Voronoi Diagram

문제를 쪼개서, 각각의 에지에 대해서 답을 계산한 후 이를 합쳐준다고 생각하면 쉽게 접근할 수 있습니다. 각 에지의 양 끝 점 $e = \{u, v\}$에 대해서, 결국, 구역은 양분되지 않거나 (에지의 모든 위치가 하나의 점과만 가깝거나), 어떠한 점이 있어서 이 점을 기준으로 u쪽과 연결되는 위치와 v쪽과 연결되는 위치가 양분되는 두 가지 경우가 존재합니다. 이 모든 경우는 집합 안에 있는 v와 가장 가까운 점, 집합 안에 있는 w와 가장 가까운 점, 그리고 그 점들과의 거리를 알면 간단한 수식 계산으로 처리할 수 있습니다. 이 수식 계산 과정에서, 모든 답이 정수거나 정수 + 0.5의 꼴이라는 것도 간파할 수 있습니다 (고로 실수 연산이 필요하지 않습니다).

이제, 모든 정점 v에 대해서 해당 정점과 가장 가까운 집합 내 정점을 계산해 주는 것으로 문제를 환원할 수 있습니다. 이는 Floyd-Warshall을 통해서 $O(N^3)$에 간단하게 계산할 수 있으며, 이를 통해 27점을 받을 수 있습니다. 더 효율적인 방법은 Dijkstra Algorithm을 사용하는 것입니다. 일반적인 Dijkstra Algorithm은 고정된 시작점에 대해서만 문제를 해결하지만, 이 문제에서는 여러 개의 시작점이 있을 때 이들에서의 최단 거리를 묻습니다. 이는 이 시작점들을 모두 Heap에 넣어주는 변형을 통해서 일반적인 Dijkstra Algorithm과 거의 동일하게 해결할 수 있습니다. (가상의 시작점을 만든 후, 집합 내의 원소들과 가중치 0의 간선을 이었다고 생각할 수도 있습니다.) 이 방법의 시간 복잡도는 $O(M \log M)$ 입니다.

중복 간선, 셀프 루프가 주어질 수 있으며, 같은 거리일 때 가장 인덱스가 작은 정점을 찾아야 한다는 부분을 구현할 때 유의가 필요합니다. 제대로 구현하는 방법을 찾으면 크게 어렵지 않게 해결할 수 있습니다.

Remark. 이 알고리즘이 Voronoi Diagram보다 낫다는 디스크립션의 내용은 농담입니다. 그래프가 좌표 평면보다 일반화된 구조인 것은 사실이나, 좌표 평면을 시뮬레이션하기 위해서는 무한한 크기의 그래프가 필요하기 때문입니다. 하지만, Taxicab metric에서 주어진 좌표들이 작다면 이 알고리즘으로 Voronoi Diagram을 시뮬레이션 할 수 있습니다.

Winter Olympic Games

Subtask 1/2 (47 points)

$O(N^2)$에 문제를 해결하기 위한 가장 중요한 관찰은, 어떠한 문자열이 사전 순 최대이기 위해서는 이 문자열의 앞에 오는 연속된 1의 개수가 가장 많아야 한다는 것입니다. 앞에 오는 연속된 1의 개수가 최대가 아니면 무조건 사전순 최대가 아니기 때문에, 주어진 문자열의 앞에 k개의 연속한 1이 있고, 이후 다음 문자가 비어있거나 0이 되었다면, $S = k$로 두고 바로 이 뒤에 1을 붙여주는 것이 무조건 최선입니다. 이 관찰을 통해 대체할 문자열의 시작점 S를 쉽게 알 수 있습니다.

한편, S를 고정했다면, 이제 뒤에 1을 붙인 이후, S 뒤에 오는 suffix를 또 하나 붙이게 될 것입니다. 이 때는 $O(N)$ 개의 가능한 suffix 중 사전 순 최대인 것을 찾아서, 이를 붙여주면 됩니다. 사전 순 최대 suffix는 단순 구현으로 $O(N^2)$에 해결할 수 있습니다.

Subtask 1/2/3 (100 points)

위 $O(N^2)$ 알고리즘을 최적화 하는 세 가지 방법을 소개합니다.

- **접미사 배열 (Suffix Array)**. 의도된 풀이로, 문자열의 모든 접미사를 사전순으로 정렬한 형태의 배열입니다. 문자열의 일부 접미사 중 최댓값을 찾고 싶어하는 우리의 목적에 완전히 부합하는 자료구조로, $O(N \log^2 N)$ (느린 Manber-Myers) $O(N \log N)$ (Manber-Myers) $O(N)$ (DC3) 정도의 시간 복잡도에 구현할 수 있습니다. 시간 제한이 크기 때문에 이 중 어떠한 방법을 사용하여도 됩니다.
- **해싱 (Hashing)**. 해싱으로 문제를 해결하는 데 있어서 가장 중요한 관찰은, 두 suffix를 $O(\log N)$에 비교하는 것이 가능하다는 것입니다. 해싱은 두 부분문자열이 같은지를 $O(1)$에 판별할 수 있게 해 주니, 어떠한 두 부분 문자열의 LCP (Longest Common Prefix)를 이분 탐색을 통해 $O(\log N)$에 찾게 할 수 있습니다. 두 부분 문자열의 LCP를 찾은 이후에는 쉽게 두 문자열의 사전 순 비교를 진행할 수 있습니다. $O(N)$개의 원소 중 최댓값을 찾고, 한번의 비교 연산이 $O(\log N)$이니 시간 복잡도는 $O(N \log N)$ 입니다.
- **가장 빠른 알고리즘**. 복잡한 자료 구조를 사용하지 않고, 굉장히 적은 비교만으로 문제를 해결할 수 있는 알고리즘에 대한 논문들이 나와 있습니다. `https://link.springer.com/chapter/10.1007/978-3-642-13073-1_29` 을 참고하십시오. 물론 의도된 풀이는 아닙니다.

Xtreme NP-hard Problem

먼저 관찰해야 할 것은, $k > n$이거나 $k > m$인 경우 단순 경로를 만들기 위한 정점과 간선의 개수가 부족하여 답이 반드시 `-1`이라는 것입니다. 따라서 해당 경우를 제외하면 $min(n, m, k) = k$이 되므로 서브태스크 조건을 $k \leq x$ ($k \leq 3, k \leq 4, k \leq 5$)의 형태로 생각할 수 있습니다.

또한, $k = x$일 때의 풀이를 안다면, $k < x$일 때의 풀이로 확장할 수 있습니다. 어떤 x에 대해 $k = x$를 만족하는 경우의 풀이를 안다면, $x-k$개의 임시 정점을 만든 후 n번 정점과 첫 번째 임시 정점, 첫 번째 임시 정점과 두 번째 임시 정점... 처럼 마지막 임시 정점까지 가중치가 0인 간선을 이어주고, 마지막 임시 정점을 도착 정점으로 설정한 후 $k = x$일 때의 풀이를 적용함으로써 $k \le x$인 경우를 해결할 수 있습니다. 고로, 아래의 풀이들은 단순히 $k = \{3, 4, 5\}$ 일 때 해결하는 풀이가 아니라, $k \le 3, k \le 4, k \le 5$ 일 때 전체 문제를 해결하는 풀이로 사용될 수 있습니다.

Subtask 1 (19 points)

$k = 3$이므로 1번 정점과 n번 정점을 잇는 단순 경로는 $\{1, x, y, n\}$ 의 형태입니다. 이 때 $\{x, y\} \cap \{1, n\} = \emptyset$ 임에 유의해야 합니다. 이 서브태스크를 포함하여, 이 문제를 해결하는 핵심 아이디어는 경로의 가운데 지점을 잡고 모두 시도해보는 것입니다. 즉, 이 서브태스크의 경우 $\{x, y\}$ 간선을 모두 시도해보는 것입니다.

따라서 그 전에 모든 $\{1, x\}$ 간선에 대해 그 가중치를 x에 저장하고, 마찬가지로 모든 $\{y, n\}$ 간선에 대해 그 가중치를 y에 저장합니다. 이를 적절한 전처리 테이블에 저장한 후, $x \neq n, y \neq 1$을 만족하는 모든 $\{x, y\}$ 간선에 대해 $w(1, x) + w(x, y) + w(y, n)$ 의 최솟값을 찾으면 됩니다. ($w(i, j)$는 i와 j를 잇는 간선의 가중치를 뜻합니다.) 이 때의 시간 복잡도는 $O(n + m)$입니다.

Subtask 1/2 (46 points)

$k = 4$이므로 1번 정점과 n번 정점을 잇는 단순 경로는 $\{1, x, y, z, n\}$의 형태입니다. 이 때 $\{x, y, z\} \cap \{1, n\} = \emptyset$, $x \neq z$에 유의해야 합니다. 마찬가지로, 모든 $\{1, x, y\}$ 형태의 간선 쌍에 대해서 가중치 합의 최솟값을 y에 저장하고, 모든 $\{y, z, n\}$ 간선 쌍에 대해 가중치 합의 최솟값을 y에 저장한 후, 이를 모든 y 정점에 대해 계산해주면 됩니다. 길이 2의 경우를 전처리 할 때에는, 역시 한쪽 끝을 잡기 보다는 중심(이 경우 x, z)을 기준으로 뿌려주는 식의 구현이 편합니다. 서브태스크 1과의 차이점으로는, $x \neq z$ 조건 때문에 두 번째 최솟값과, 중간 정점 번호를 같이 저장해 주어야 한다는 점입니다. 최솟값의 중간 정점 번호가 같다면 둘 중 하나를 두 번째 최솟값으로 쓰면 됩니다. 이 때의 시간 복잡도는 $O(n + m)$입니다.

Subtask 1/2/3 (100 points)

$k = 5$이므로 1번 정점과 n번 정점을 잇는 단순 경로는 다음과 같은 형태입니다: $\{1, a, x, y, b, n\}$ 의 형태입니다. 이 때 $\{a, x, y, b\} \cap \{1, n\} = \emptyset, a \neq b, x \neq b, a \neq y$에 유의해야 합니다. $k = 4$와 마찬가지로 모든 $\{1, a, x\}$ 간선 쌍에 대해 가중치 합의 최솟값을 x에 저장하고, 모든 $\{y, b, n\}$ 간선 쌍에 대해 가중치 합의 최솟값을 y에 저장한 후, 모든 $\{x, y\}$ 간선에 대해 계산해주면 됩니다. 서브태스크 2와의 차이점으로는, $x \neq z, y \neq w, x \neq w$ 조건 때문에 세 번째 최솟값까지 저장해 주어야 한다는 점입니다. 이 때의 시간 복잡도는 $O(n + m)$입니다.

여담으로, 이 문제의 제목과 첫 문장에 적혀있듯이, 이 문제는 서브태스크 조건과 같은 k에 대한 조건이 없을 경우 NP-hard가 맞습니다. 한 번 증명을 시도해보세요! 힌트: `https://goo.gl/Es7jLs`

Yut Nori

윷놀이는 문제에 나와있는 내용을 단순히 구현하면 되는 문제입니다. 구현할 내용이 상당히 많고 복잡한데, 다음과 같은 것을 단계적으로 구현하는 것이 좋습니다.

- 각 위치를 번호로 구분한 후, 현재의 위치에서 x칸 움직인 후의 위치를 계산하는 모듈의 구현
- 각 윷 던지기마다, 업어가기와 잡기를 고려하여서 각 말들의 새 위치를 계산하는 모듈의 구현
- 각 말들의 최종 위치가 주어졌을 때 이를 출력하는 모듈의 구현

특히, 날밭에서의 움직임을 구현할 때 신중하게 생각하는 것이 좋습니다. 제대로만 처리하면 어렵지 않습니다.

Zigzag

제한이 작기 때문에 단순한 접근법으로 문제를 해결할 수 있습니다. 어떠한 구간의 시작점 i를 고정시키고, i를 시작점으로 하는 가장 긴 지그재그 연속 구간을 찾아주면 되기 때문입니다. 이 방법을 구현하면 $O(N^2)$ 에 문제를 해결할 수 있습니다.

혹시 $N = 3$인 서브태스크부터 틀렸습니다가 뜬다면, 다음과 같은 케이스에서 결과가 어떻게 나오는지 확인해 보시기 바랍니다. 올바른 정답은 2입니다. 길이 2인 수열은 원소의 배치와 상관없이 무조건 지그재그이기 때문입니다.

```
3
1 1 1
```

2018 KAIST RUN Spring Contest Editorial

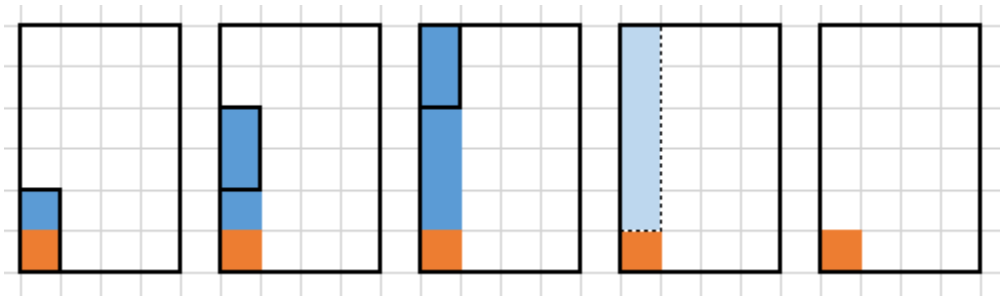
Credits

Problem	Author	Preparation	Editorial
P	Hanpil Kang	Hanpil Kang	Hanpil Kang
Q	Jaehyun Koo	Hanpil Kang	Jaehyun Koo
R	Jaemin Yu	Jaemin Yu	Jaemin Yu
S	Taehyun Lee	Taehyun Lee	Jaehyun Koo
T	Jaehyun Koo	Jaehyun Koo	Jaehyun Koo
U	Jaehyun Koo	Jaehyun Koo	Jaehyun Koo
V	Jaehyun Koo	Hanpil Kang	Jaehyun Koo
W	Jaehyun Koo	Jaemin Yu	Jaehyun Koo
X	Minkyu Jo	Minkyu Jo	Minkyu Jo
Y	Hanpil Kang	Hanpil Kang	Jaehyun Koo
Z	Jaehyun Koo	Jihoon Hyun	Jaehyun Koo

PuyoPuyo

Subtask 1 (63 points)

The limits are small, and the final grid doesn't contain component of size 4 or larger. Thus the problem can be solved by simply stacking up each block one by one. How can we stack up a single block? See the picture below :



This simple method can stack one block with 3 operation, thus solving Subtask 1 with 48 operations. Note that you should stack each block row by row.

Subtask 1/2 (100 points)

We can reduce the operation of above algorithm. Observe that columns with even height can be simply made with $\frac{R}{2}$ operation by simply stacking Puyo pairs. So, we have to handle odd height. To

handle this, we can put one block in each row using the above procedure, and stack the remaining even pairs with above algorithm. The largest possible number of Puyo pair used is $\frac{19+5}{2} \times 20 = 240$, which is sufficient for this problem.

QueryreuQ

Subtask 1 (36 points)

Subtask 1 can be solved using a relatively simple algorithm. Note that only palindromes containing last letter of string will be disappeared / appeared in each query. As there are at most $O(Q)$ palindromes containing the last letter, you can naively check whether they are palindromes or not - this yields an simple $O(Q^3)$ algorithm.

Subtask 1/2 (100 points)

The approach remains same in Subtask 2, but we use dynamic programming (DP) for checking whether each substring are palindrome. Let $DP[i, j]$ be a boolean matrix storing whether substring on $[i, j]$ is palindrome or not. $DP[i, i]$ and $DP[i, i + 1]$ are the base cases, and in other cases recurrences $DP[i, j] = (DP[i+1, j-1]$ & $S[i] == S[j])$ will hold. With this DP matrix, we can calculate whether each substrings are palindrome are not, in $O(1)$ time. When new characters are added we can modify DP array according to this definition, and when characters are deleted we can just ignore the last column. This makes each query $O(Q)$, and the total time complexity becomes $O(Q^2)$.

As a side note, there is a paper which describes an $O(QlgQ)$ solution to this problem. `https://arxiv.org/abs/1506.04862`

Recipe

Subtask 1 (17 points)

If the ingredients are bought on ith day and cooked on jth day, it can be regarded as the interval $[i, j]$. This interval has a weight $(F_i - j + i) \times C_j$. Then, the problem of dividing the interval $[1, N]$ into several consecutive subintervals to maximize the sum of the weights of them is to be solved.

Using dynamic programming with the following formula, you can solve it in $O\left(N^2\right)$, and get the score for Subtask 1.

$$D_i = \min_{j=i}^{N} \{ (F_i - j + i)\, C_j + D_{j+1} \} \quad if \quad L_j \leq F_i - j + i \tag{3}$$

Subtask 2 (20 points)

If F_i falls into a negtive number, this case cannot be the optimal solution. Therefore, cooking always occurs before the freshness of the food becomes negative. If all L_i is 0, $L_j \leq F_i - j + i$ always be held, so you can ignore it.

To use Convex Hull Trick (CHT), the formuula can be changed to the following.

$$D_i = \min_{j=i}^{N} \{(F_i + i)\, C_j - jC_j + D_{j+1}\} \quad if \quad L_j + j \leq F_i + i \tag{4}$$

The slopes C_j of the lines $f_j(x) = C_j x - jC_j + D_{j+1}$ is monotone decreasing, so that an upper bound of the convex hull can be managed. (It is easier to fill the dp table from N to 1. Note that the C_j is monotone decreasing in this direction.) If you do not consider about if statement and just use CHT optimization, you can get the score for Subtask 2.

Subtask 1/2/3 (100 points)

Only some $f_j(x)$ which satisfies $L_j + j \leq F_i + i$ is used to calculate D_i. It means that the line $f_j(x)$ is actually the half line defined on $[L_j + j, \infty)$.

Otherwise, you can manage half lines efficiently. The upper bound U_{i+i} of the half lines $f_{i+1}, f_{i+2}, \cdots, f_N$is increasing function. When we make U_i by adding f_i to U_{i+1}, there are two cases.

1. $U_{i+1} \geq f_i \quad for \quad all \quad x \in \left[\min_{j=i}^{N} (L_j + j), \infty\right)$

   It can be determined by comparing between $f_i(x)$ and $U_{i+1}(x)$ when $x = \min_{j=i}^{N} (L_j + j)$.

   In this case, f_i is always not greater than U_{i+1} for all x in the domain of U_{i+1}, so that U_i can be constructed by appending the f_i to the front of U_{i+1} if f_i is defined on the left side of the U_{i+1}.

2. $\exists x_0 \quad s.t \quad U_{i+1}(x) < f_i(x) \quad if \quad x < x_0 \quad and \quad U_{i+1}(x) > f_i(x) \quad if \quad x > x_0$

   Otherwise, some x_0 exists such that f_i is greater than U_{i+1} on $[L_i + i, x_0)$. Erasing half lines on $[L_i + i, x_0)$ and insert f_i allows to construct U_i.

Using balanced binary tree, all above tasks can be done in $O(N \log N)$.

There are other alternative solutions, which all fits in time limit :

- This problem also can be solved in $O\left(N \log^2 N\right)$ without monotone increasing condition of C_i, if you use Li Chao segment tree.

- You can use divide and conquer to solve the whole problem in $O(N \log N)$, without any complicated data structures. If C_i is not monotone, this solution can be adapted to $O\left(N \log^2 N\right)$ time complexity.
- As $L_i + i$ is in range $[1, 300000]$, you can maintain CHT in each node of fenwick tree, it can be solved in $O(Nlog^2N)$.

Segmentation

This problem can be easily solved with standard data structures such as "Map". The hardest part is to retrieve the (r, f) information for each users in query. This can be solved by making two maps, which takes each user's name as a "key", and each user's last apperance time, and the frequency as "value". `map` in C++ and `TreeMap` in Java can be used to solve this problem. If you replace maps as simple arrays, or Hash Maps which have worst case $O(n)$ time complexity, then you might get time limit exceeded.

With (r, f) information given for each query, it's straightforward to solve the given problem. For implementation, we recommend to use arrays to find the results for each (r, f) pair - rather than using lots of if-else cases.

Touch The Sky

Subtask 1 (22 points)

The most convenient observation is noticing that the problem is actually about task scheduling. Consider a standard "Deadline Scheduling" instance - Bursting the balloon corresponds to a "task completion", and altitude increment corresponds to "task processing time". Now, the deadline for a single task is the "time limit for starting a certain task" (L_i) + "processing time" (D_i). Now, we can model our problem as the following optimization problem :

- Task is started at time 0.
- It takes D_i time to process i-th task, and the deadline is $D_i + L_i$.
- You should process the maximum number of task.

Rather than thinking about the maximum number of task, let's just think whether we can process **all tasks** or not. The best way to process all the task is to start with the task with least deadline (Deadline first scheduling). The optimality of this method can be proven with exchange argument - there is no advantage for doing the task with longer deadline first.

This single observation yields a solution for Subtask 1. We enumerate all the subset of tasks, and check whether all the task in the subset can be processed. If we assume that tasks are sorted by deadline ($D_i + L_i$), we can find whether we can process all tasks or not, by simply processing it in order. The whole problem is solved in total time $O(2^N \times N)$.

Subtask 1/2 (55 points)

To solve subtask 2, we can optimize the above algorithm with Dynamic Programming. For convenience, let's sort all tasks by deadline from now on. To use dynamic programming, observe that only two parameters matter : we only care whether "how much balloons we burst", and "how much we ascended while bursting the balloon". For fixed number of bursted balloon, we want to minimize our height (as we want to minimize the total used time).

Now, we can come with a recurrence $DP[i][j] =$ (minimum height after bursting j balloons among i first balloons). We have two state transitions :

- Case 1. $DP[i-1][j]$: i-th balloon can be ignored.
- Case 2. $DP[i-1][j-1]$: If $DP[i-1][j-1] + D[i] \leq D[i] + L[i]$, we can burst ith balloon. Thus we can update $DP[i][j]$.

Maximum number of bursted balloon can be retrieved by scanning $DP[n][*]$. Total time complexity of this algorithm is $O(N^2)$.

Subtask 1/2/3 (100 points)

An intuitive approach

Suppose we inductively calculated the maximum burstable balloon subset among first i balloons. If we can insert $i+1$-th balloon in this set, we can just insert it into the set (why not). The problem arises when $i+1$-th balloon can't just be inserted. Now we should make a choice - we could not insert $i+1$-th balloon at all, or we could insert the balloons by removing some existing balloons.

Now we will strengthen the condition. Suppose we inductively calculated the maximum burstable balloon subset among first i balloons, and in case of tie we calculated the one which **minimizes the final altitude**. If we can insert $i+1$-th balloon without problem, we can just do it to increase subset size - suppose otherwise. Now we might attempt removing some balloons, and insert $i+1$-th balloons. Now, the single balloon which is best to remove is, the one that increases the altitude highest. If replacing it to $i+1$-th balloon reduces the final aititude, we can just do it.

Our above algorithm is indeed correct, and it can be implemented with heap in $O(N \log N)$ with very short code. However, the explanation given above is not a full proof - we dismissed many

possibilities which is not intuitive, but might give a better results. Now, I will try to give a rigorous proof for above algorithm, which starts from Subtask 2 solution. Don't start If you don't fully get the algorithm for Subtask 2!

A rigorous proof

For all $0 \le i \le N, 0 \le j \le i$, we define $S_{i,j}$ as size-j balloon subset, wihch is a subset of first i balloons, and fully burstable in deadline order. If there are more than one such subset, we pick the one which have minimum altitude sum. If there are no such subset, we call it "undefined".

Lemma 3. $S_{0,0} = \emptyset$, and for all $1 \le i \le n, 0 \le j \le i$, there is at most 2 candidates for $S_{i,j}$, and we can just pick the one with smaller altitude sum.

- If $j > 0$, and $S_{i-1,j-1}$ is defined, and $S_{i-1,j-1} \cup \{A_i\}$ is fully burstable in deadline order, this is a valid candidate. (Choice 1)
- If $S_{i-1,j}$ is defined, this is a valid candidate. (Choice 2)

Proof. Easy induction. □

Remark. This lemma is a translation of DP which solves subtask 2.

Our next lemma is the core of the problem, which shows why every operation is doable in heap :

Lemma 4. If $S_{i,j+1}$ is defined, then $S_{i,j}$ is defined, and $S_{i,j} \subset S_{i,j+1}$.

Proof. It's obvious to see that if $S_{i,j+1}$ is defined, then $S_{i,j}$ is deifned. Because removing any element in $S_{i,j+1}$ doesn't change that the set is fully burstable in deadline order. Let L_i the maximum j such that $S_{i,j}$ is defined. Now, we show $S_{i,j} \subset S_{i,j+1}$ by induction on i.

If $i = 0$, $S_{0,0}$ is only defined - trivial. Suppose $i > 0$. By inductive hypothesis, for all $0 \le j < L_{i-1}$, $S_{i-1,j} \subset S_{i-1,j+1}$ holds. However, $S_{i-1,L_{i-1}}$ can be bursted in deadline order, so removing any elements in $S_{i-1,L_{i-1}}$ doesn't change the fact that it can be bursted in deadline order. Thus we can always remove the element with maximum altitude difference, when we make $S_{i-1,j}$ from $S_{i-1,j+1}$. Let $F(S)$ a sum of altitude difference for elements in S. By previous observation, $F(S_{i,j+1}) - F(S_{i,j}) \le F(S_{i,j+2}) - F(S_{i,j+1})$ holds.

Now we prove that, when inductively building $S_{i,j}$ from smaller j to larger j, there exitst a single point x where for all $j \le x$ we pick Choice 2, and for all $j > x$ we pick Choice 1. Thus, if we write $S_{i,j}$ in increasing order of j, we get a form of $S_{i-1,0}, S_{i-1,1}, \cdots, S_{i-1,j}, S_{i-1,j} \cup \{A_i\}, S_{i-1,j+1} \cup \{A_i\}, \cdots$. If such property holds, for all adjacent subsets we have $S_{i,j} \subset S_{i,j+1}$, thus completing the proof.

To prove this, it's enough to prove that there exists no j such that $S_{i,j} = S_{i-1,j-1} \cup \{A_i\}, S_{i,j+1} = S_{i-1,j+1}$ holds. If we show this, then after adding A_i into the set (the point x), we will never go back before. Suppose that we pick Choice 2 in case of tie in sums. Then :
$F(S_{i-1,j-1}) + F(\{A_i\}) < F(S_{i-1,j})$
$F(S_{i-1,j+1}) \leq F(S_{i-1,j}) + F(\{A_i\})$
$F(S_{i-1,j+1}) - F(S_{i-1,j}) \leq F(\{A_i\}) < F(S_{i-1,j}) - F(S_{i-1,j-1})$
$F(S_{i-1,j}) - F(S_{i-1,j-1}) \leq F(S_{i-1,j+1}) - F(S_{i-1,j})$

This is a contradiction, and the proof is complete. □

By Lemma 4, we can notice that the "intuitive algorithm" actually maintains S_{i,L_i} inductively. If $L_{i-1} + 1 = L_i$, then we insert A_i on heap and continue. Otherwise, we have two choices - either picking $S_{i-1,L_{i-1}}$ or $S_{i,L_{i-1}-1} \cup \{A_i\}$. Now, we can see that $S_{i-1,L_{i-1}-1}$ is actually $S_{i-1,L_{i-1}}$ with its maximum removed, so it can be easily computed. If we determine the optimal set from both, we can compute S_{i,L_i}, which explains why the "intuitive algorithm" works.

United States of Eurasia

Subtask 1 (20 points)

Subtask 1 can be solved with dynamic programming. Assume that every points are sorted according to x coordinate. Let $Cost[i, j]$ be a maximum distance among point pairs in interval $[i, j]$. For convenience, assume $X_0 = -1$. We can now derive a following recurrence.

$DP[i, j] = min_{k<j, X_k \neq X_{k+1}}(max(DP[i-1, k], Cost[k+1, j]))$

Time complexity is $O(N^3)$, and slower solution which computes $Cost[*, *]$ in $O(N^4)$ will also pass if effeciently implemented. However, it doesn't seem easy to reduce time complexity further with this approach..

Subtask 1/2 (40 points)

For a problems which "minimizes the maximum", the technique of converting it to a **decision problem** is well known. For example, if the original problem is about "minimizing maximum splittedness with at most k partition", we can convert it to a problem of "partitioning with at most k partition while maintaining splittedness of each partition at most M". If we can solve such decision problem efficiently, we can use binary search on answer, to solve the original problem.

It turns out that it's actually not very hard to solve this decision problem in $O(N^2)$ time. For an element with minimum X_i, we find a longest interval containing this point, which have splittedness

at most M, and remove all numbers in that interval. With this algorithm, it performs $O(N)$ operation for each element, thus solving the whole decision problem in $O(N^2)$ time.

Subtask 1/2/3 (100 points)

For solving the whole problem, we first find the geometric properties of the problem, and exploit it in order to reduce the time complexity. First idea is that we can calculate the "splittedness" of interval $[S, E]$ in $O(E - S + 1)$, which is a combination of these two ideas :

- 1. Finding a **convex hull** of points in interval : In most cases we calculate the convex hull with Graham Scan algorithm, which have $O(N \log N)$ time complexity. However, the elements are sorted in order of x coordinate, so we can use faster approach - Andrew's monotone chain algorithm computes the convex hull in $O(N)$ time, assuming the points sorted in x coordinate.
- 2. Finding a farthest pair of points in convex polygon : This is a well known problem which can be solved in $O(N)$ with rotating callipers. You can learn about it here : `http://codeforces.com/blog/entry/48868`.

Still, this is not enough to reduce the time complexity from $O(N^2)$. However, now it might be tempting to use $O(KN \log N)$ algorithm, which finds a endpoint of interval for each corresponding startpoints. This algorithm indeed works fast if the intervals are enough large (about half of all), but it will waste too much time if the intervals are smaller than expected - the algorithm wastes most time doing too much optimistic guesses.

To reduce wastes that arises from too much "optimistic" gueses, we can try an alternative binary search, which takes time proportional to the answer - the length of a maximum interval. If the time taken in binary search is proportional to the length of interval, then the total time taken by an algorithm is proportional to N - thus, we can find an algorithm that is close to linear time!

For an "alternative binary search", think about this approach : for some startpoint S, we can find a minimum T which interval $[S, S + 2^{T-1} - 1]$ can be partitioned, but interval $[S, S + 2^T - 1]$ cannot be partitioned. Now, let's fix the endpoint of interval in range$[S + 2^{T-1} - 1, S + 2^T - 1]$ and try a binary search. Our binary search uses $O(\log N)$ queries for farthest pair of points in interval, and the intervals used in each queries is at most two times the maximum length of interval. Thus, if the maximum length of interval is L, each binary search takes $O(L \log N)$ time, and the total decision problem is solved in $O(N \log N)$ time. Thus, the whole problem is solved in time complexity $O(N \log N \log X)$.

Voronoi Diagram

To approach this problem, we solve the problem for each edge, and sum all the answers afterwards. For each edge $e = \{u, v\}$, the area is not demarcated at all (every location is closest with a single vertex), or there exists a point such that every point left to such passes through u, and every point right to such passes through v. This cases can be handled with simple calculation if we know the closest vertex in set S for a vertex u, v, and the distance with such. Also, you can now notice that all answers are either integers or half an integer (thus no need for floating point arithmetics).

Now the problem reduces to the following : for each vertex $v \in V(G)$, find a vertex in S which is closest to v. This can be easily computed with Floyd-Warshall, which gives $O(N^3)$ algorithm and receives 27 points. More efficient way is to use Dijkstra's algorithm. Standard Dijkstra's algorithm finds a shortest distance from single start vertex, but in this problem we are finding a shortest distance from multiple start vertex. If you put all the start vertex in heap in the beginning of algorithm, this can be solved in almost analogous way. (You can interpret this as making a dummy start vertex, which have distance 0 with all elements in S.) The time complexity is now $O(M \log M)$.

Note that multiple edges and self loops are given, and you should be careful when you have multiple vertex with smallest distance (you should break ties by vertex index). If well implemented, you don't need any case handling for these at all.

Remark. The statement claim that this is a better alternative to traditional Voronoi Diagram algorithm, but it's obviously a joke :) While the graph is indeed more generalized than Cartesian coordinates, you need infinite graph to simulate such. However, this kind of algorithm actually works if you are solving problems in Taxicab metric, with all points being small integers.

Winter Olympic Games

Subtask 1/2 (47 points)

For a string to be lexicographically maximum, it should have the most number of consecutive leading 1s in the beginning of string - otherwise, the string is obviously not maximum. This simple observation is crucial for reducing time complexity. If the given string has k consecutive leading 1s, then it's always optimal to set $S = k$. Thus, the startpoint of replacement can be easily computed. However, if we know the position of S, then we will append 1, and a suffix that comes after S. So, we should pick a lexicographically maximum suffix among those $O(N)$ candidates. This can be simply computed in $O(N^2)$.

Subtask 1/2/3 (100 points)

We present three methods that optimize above $O(N^2)$ algorithm.

- **Suffix Array**. This is an intended solution. Suffix array is an array which contains all suffixes lexicographically sorted - this is exactly what we want, as we are finding a maximum among certain suffixes. This array can be constructed in $O(N \log^2 N)$ (Poor man's Manber-Myers), $O(N \log N)$ (Manber-Myers), $O(N)$ (DC3). Any of these construction will work in time limit.
- **Hashing**. The most important observation for the hashing solution is that you can compare any suffixes lexicographically in $O(\log N)$ time. Hashing provides a way to find whether two substring are equal in $O(1)$ time. Thus, a logest common prefix (LCP) of two substring can be found with binary search in $O(\log N)$ time. After finding LCP of two substring, it's straightforward to compare two substrings. This algorithm works in $O(N \log N)$ time, as we are finding maximums among $O(N)$ element, and one comparison takes $O(\log N)$ time.
- **Fastest algorithm**. There are papers which doesn't use any data structures and uses a very small number of comparisons. See this : `https://link.springer.com/chapter/10.1007/978-3-642-13073-1_29` Obviously, this solution is not intended.

Xtreme NP-hard Problem

First, you could notice that in the case of $k > n$ and $k > m$, you lack the number of vertices and edges to construct the simple path. The answer is always `-1`, so without those cases, you will always have $min(n, m, k) = k$, and the subtask condition can be thought as $k \leq x$ ($k \leq 3, k \leq 4, k \leq 5$).

Also, if you have a solution for $k = x$, then you can extend this solution to $k < x$. For some x, if you know the solution for $k = x$, then you can make a $x - k$ virtual vertices and cost-0 edges connecting from vertex n, passing through first virtual vertex, second one... and finally, the real destination. Thus, the solutions presented below is not only a solution for $k = \{3, 4, 5\}$ - it's a solution for $k \leq 3, k \leq 4, k \leq 5$.

Subtask 1 (19 points)

As $k = 3$, any simple path that connects vertex 1 and vertex n looks as $\{1, x, y, n\}$. Be aware that $\{x, y\} \cap \{1, n\} = \emptyset$. The main theme of all the solutions below, is to solve by brute-forcing all "central" vertex and edges. For example, you can try all edges that lies in the center ($\{x, y\}$).

For all edges connecting 1 and x, you can store its weight at x, and vice versa for all edges connecting y and n. Now, for all edge $\{x, y\}$ with $x \neq n, y \neq 1$, we can find a minimum of $w(1, x) + w(x, y) + w(y, n)$. ($w(i, j)$ denotes the weight of edge connecting i and j.) The time complexity is $O(n + m)$.

Subtask 1/2 (46 points)

As $k = 4$, any simple path that connects vertex 1 and vertex n looks as $\{1, x, y, z, n\}$. Be aware that $\{x, y, z\} \cap \{1, n\} = \emptyset$, $x \neq z$. Analogously, for all length-2 path of form $\{1, x, y\}$, you can store the minimum weight sum in y, and for all length-2 path of form $\{y, z, n\}$, you can store the minimum weight sum in y, and merge the answer for all $y \in V(G)$. To preprocess the case of length 2, you should update the answer from a center (in this case x, z). Unlike subtask 1, you should store the second minimum and the vertex number due to condition $x \neq z$. If the vertex number of minimums are same, you should use one of them as second minimums. The time complexity is $O(n + m)$.

Subtask 1/2/3 (100 points)

As $k = 5$, any simple path that connects vertex 1 and vertex n looks as $\{1, a, x, y, b, n\}$. Be aware that $\{a, x, y, b\} \cap \{1, n\} = \emptyset, a \neq b, x \neq b, a \neq y$. Like case $k = 4$, for all length-2 path of form $\{1, a, x\}$ you can store the minimum weight sum in x, and for all length-2 path of form $\{y, b, n\}$ you can store the minimum weight sum in y, and iterate for all edges $\{x, y\}$. Note that you have more conditions ($x \neq z, y \neq w, x \neq w$), and you should store "three minimums". The time complexity is $O(n + m)$.

By the way, as written in problem statement, this problem is NP-hard if there is no restrictions given on k. `https://goo.gl/Es7jLs`

Yut Nori

This is a simple implementation problem which you can just code what problem says. There are lot of things to implement, so you should do it carefully step-by-step. This is a 3-step which author took :

- Identify each position with numbers, and implement a module which computes the x-th next position from current position.
- Implement a module which recalculates the position of each token, considering "token catch" and "move together".
- Implement a module that prints the game board, given location of tokens.

We recommend you to be especially careful when implementing the movement in "cham-meoki". If you do it in the right way, there's nothing really hard on that part.

Zigzag

As the limits are small, naive approach is enough to solve this problem. You can fix the startpoint of optimal endpoint i, and find the maximum length zigzag subsegment starting at i. Implementing this approach leads to an $O(N^2)$ solution.

If you get wrong answer on $N = 3$ (test 4 in CF), then you are missing the case below. The correct answer is 2 : Sequence of length 2 is always zigzag regardless of it's composition.

```
3
1 1 1
```