제 3 회 소프트콘

풀이

A. 점프 숨바꼭질

Subtask1

- n번 전부 +방향으로 점프했다고 합시다. 이 경우 2^0 + 2^1 + 2^2 + ...2^(n-1)을 모두 더한 위치인 2^n - 1 번 위치로 이동하게 됩니다.
- 여기서, 임의의 몇 번의 점프를 반대 방향(-방향)으로 바꿔 뛸 수 있습니다.
- 어떤 k번째 점프를 반대로 뛸 경우, +2^k가 -2^k로 바뀌게 됩니다. 즉, 최종 위치는 2 * 2^k만큼 감소하게 됩니다.

A. 점프 숨바꼭질

Subtask1

- 따라서, 어떤 k번째 점프를 선택해서 방향을 바꿔도 최종 위치는 2의 배수만큼 감소하게 되고, 2^n - 1은 홀수이므로 n번 점프를 뛰어서 도착 가능한 위치는 홀수번 위치 밖에 없습니다(맨 처음 시작점인 0은 제외)
- 따라서, 0이 입력으로 주어지면 0을 출력, 그 외의 경우 -1을 출력하면 답이 됩니다.

A. 점프 숨바꼭질

Subtask2

- Subtask1에서 이야기한 것처럼 n번을 모두 + 방향으로 점프한 후 임의의 몇 가지 점프를 골라 반대방향으로 바꿀 경우 최종 위치에서 2 * x 만큼 감소한 위치로 이동 가능합니다.
- 이 때, x는 0이상 (2^n - 1) 이하의 모든 수가 될 수 있습니다. 각 점프를 이진법에서의 한 비트에 대응된다고 생각하면 임의의 n비트 수를 만들 수 있기 때문입니다. 따라서, n번 점프할 경우 -(2^n - 1) 이상 (2^n - 1)이하의 모든 홀수를 만들 수 있습니다.
- 이제 최종 위치 k에 도달 가능한 최소 크기의 n을 위의 공식으로부터 찾아서 출력하면 답이 됩니다. 시간복잡도는 O(logK)입니다.

B. 2D 큐브

- Subtask1
  - 입력으로 가능한 경우의 수가 많지 않습니다.
  - 손으로 직접 가능한 모든 경우에 대한 답을 구해서 넣거나, 백트래킹을 통해 가능한 모든 경우를 확인해보는 등의 방법으로 문제를 해결할 수 있습니다.

B. 2D 큐브

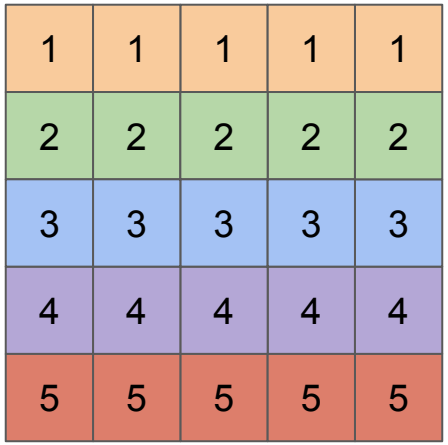
- Subtask2
  - 왼쪽에서 i번째 줄에 i가 적혀 있는 데이터만 주어집니다.

B. 2D 큐브

- Subtask2
  - 왼쪽에서 i번째 줄에 i가 적혀 있는 데이터만 주어집니다.
  - 위에서 i번째 줄을 왼쪽으로 (i-1)칸만큼 밉니다. 이렇게 밀 경우 아래와 같은 형태가 됩니다.

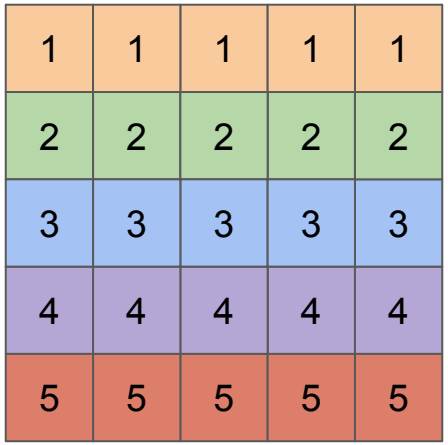
B. 2D 큐브

- Subtask2
  - 여기서, 다시 각 줄에 대해 1이 맨 위로 오게끔 위로 밀어주면 전체 큐브가 맞춰집니다.



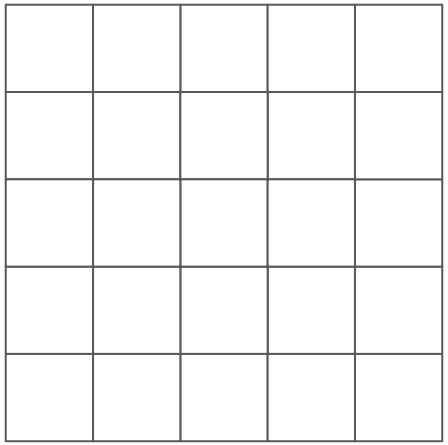
B. 2D 큐브

- Subtask2
  - 여기서, 다시 각 줄에 대해 1이 맨 위로 오게끔 위로 밀어주면 전체 큐브가 맞춰집니다.
  - 즉, (n-1) * 2 번의 동작으로 항상 큐브를 맞출 수 있습니다.



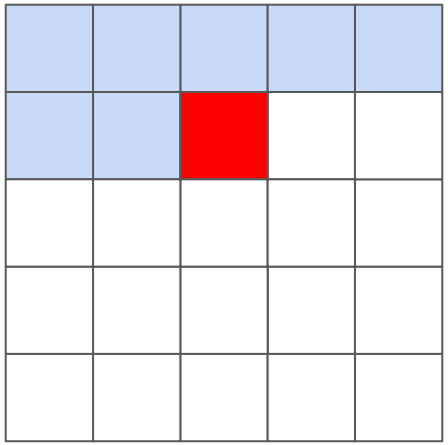
B. 2D 큐브

- Subtask3
  - 맨 왼쪽위부터 순서대로 모든 칸을 맞춘다고 생각해봅시다.



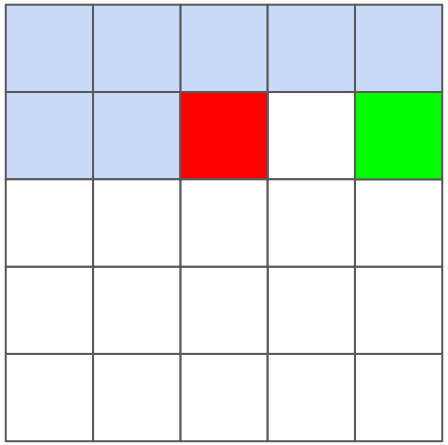
B. 2D 큐브

- Subtask3
  - 맨 왼쪽위부터 순서대로 모든 칸을 맞춘다고 생각해봅시다.
  - 파란색으로 칠해진 칸은 해당 위치에 올바른 수가 채워져 있고, 빨간색 위치를 맞출 차례라고 가정해봅시다.



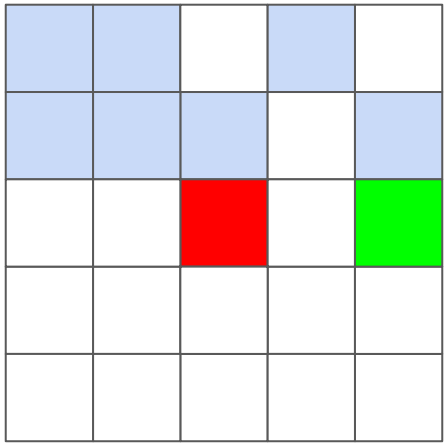
B. 2D 큐브

- Subtask3
  - Case 1 : 해당 위치에 들어가야 하는 수가 같은 줄의 오른쪽에 있는 경우



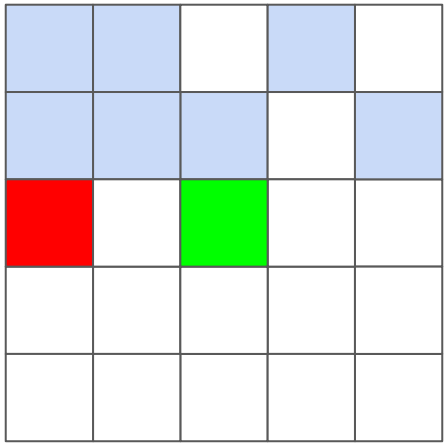
B. 2D 큐브

- Subtask3
  - 빨간색이 있는 라인과 초록색이 있는 라인을 모두 한칸씩 내립니다.



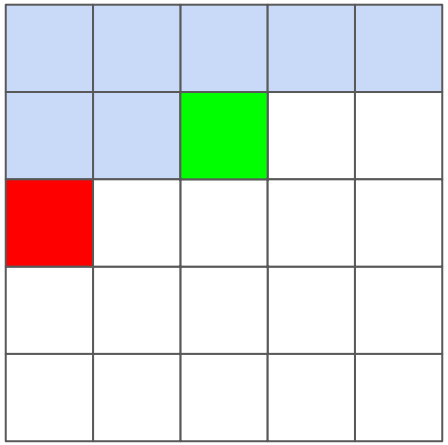
B. 2D 큐브

- Subtask3
  - 초록색을 빨간색 위치에 오게 왼쪽으로 밉니다.



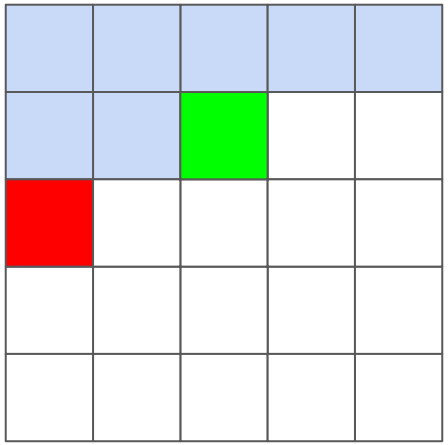
B. 2D 큐브

- Subtask3
  - 다시 양 라인을 위로 올립니다.



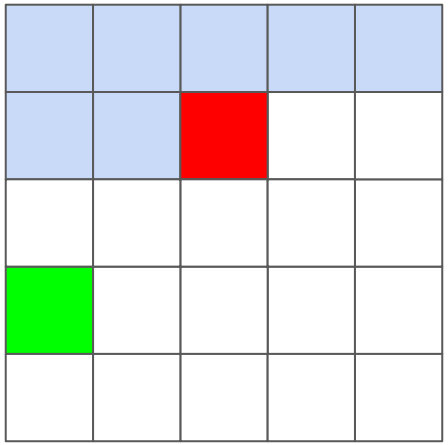
B. 2D 큐브

- Subtask3
  - 다시 양 라인을 위로 올립니다. 5번의 동작으로 이미 맞춰진 칸들을 건드리지 않은 채 원하는 값을 해당 위치로 옮길 수 있습니다.



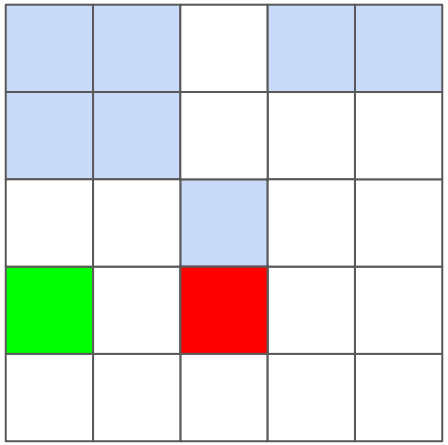
B. 2D 큐브

- Subtask3
  - Case 2 : 해당 위치에 들어가야 하는 수가 아래쪽 줄에 있는 경우



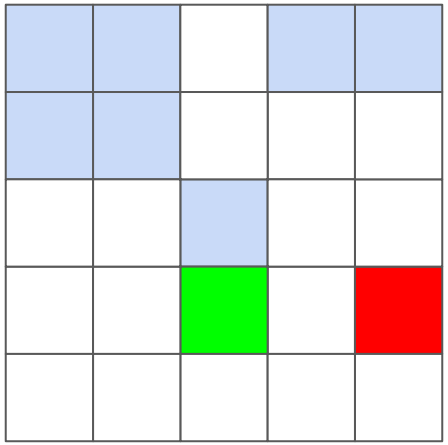
B. 2D 큐브

- Subtask3
  - 맞춰야 하는 칸이 있는 줄을 초록색이 있는 칸과 같은 위치에 가게 내립니다.



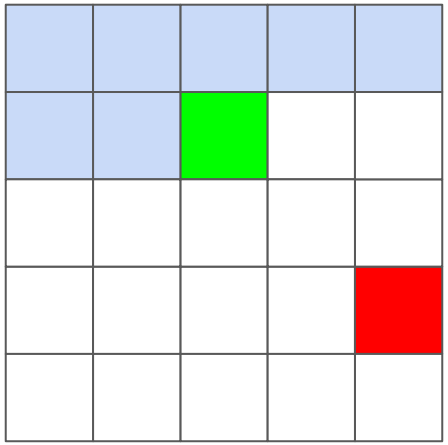
B. 2D 큐브

- Subtask3
  - 초록색 칸이 빨간색 칸의 위치에 가게 오른쪽으로 밀어줍니다.



B. 2D 큐브

- Subtask3
  - 다시 해당 줄을 위로 밀어올립니다. 3번의 동작으로 필요한 값을 원하는 위치에 옮길 수 있습니다.



B. 2D 큐브

- Subtask3
  - 이 방법으로 N-1번째 줄까지 전부 맞추고 나면, N번째 줄은 자연스럽게 맞춰집니다. N-1번째 줄까지는 항상 위의 방법을 적용해서 필요한 값을 원하는 위치에 옮길 수 있습니다.
  - 따라서 이 방법을 사용할 경우 많아야 5번의 동작으로 각 칸을 원하는 위치로 옮길 수 있으며, N-1번째 줄까지만 맞추면 되므로 필요한 횟수의 상한은 5(N-1)*N - 5 = 5N^2-5N-5 임을 알 수 있습니다. N제한이 50이므로 여유롭게 수행가능함을 알 수 있습니다.

C. 도시 계획

Subtask1

- N개의 빌딩 중 어떤 빌딩을 파괴할지를 고릅니다. 총 O(2^N)가지 경우를 고를 수 있습니다.
- 고른 빌딩을 파괴하고 난 후 남은 빌딩들이 주어진 조건을 만족하는지 확인합니다. 임의의 두 빌딩 쌍에 대해 그 사이에 있는 빌딩이 전부 두 빌딩을 잇는 직선을 넘지 않는지 확인해보면 됩니다. O(N^3) 시간에 해당 과정을 끝마칠 수 있습니다.
- 따라서, 전체 O(2^N * N^3) 시간 복잡도에 해결 가능합니다.

C. 도시 계획

Subtask2

- 주어진 조건을 만족하려면 인접한 빌딩들 사이에 기울기가 항상 증가하는 형태여야 합니다. 즉, 연속한 임의의 i번째, i+1번째 빌딩을 이은 직선의 기울기가 항상 i+1번째, i+2번째 빌딩을 이은 직선의 기울기 이하여야 합니다.
- 즉, 마지막으로 선택된 마지막 두 빌딩이 무엇인지 안다면 이번 빌딩을 선택할 수 있는지 아닌지 여부도 파악할 수 있습니다.
- 따라서, DP(idx, a, b) = 0..idx번째 빌딩까지 고려했을 때 마지막으로 선택된 두 빌딩이 a번째, b번째 빌딩일 때, 고를 수 있는 최대 빌딩 수(a <= b <= idx)
- 라고 정의하면 N - DP(N, a, b)중 최솟값이 답이 됩니다.
- 따라서 O(N^3)에 주어진 문제를 해결할 수 있습니다.

C. 도시 계획

Subtask3

- 임의의 두 빌딩 쌍을 이은 것을 기울기 순서대로 정렬합시다. 기울기가 서로 같은 두 직선에 대해서는 오른쪽 끝 점 좌표가 작은 것이 먼저 오게 정렬합니다.
- 이렇게 정렬하고 나면, 기울기가 항상 증가하는 순서대로 직선을 확인하게 되므로, 어떤 l번째 빌딩과 r 번째 빌딩을 잇는 직선에 대해 DP[r] = max(DP[r], DP[l] + 1) 을 반복해주면 조건을 만족하는 최댓값을 구할 수 있습니다.
- 따라서, O(N^2logN) 시간에 문제를 해결할 수 있습니다.

D. 계산 최적화

Subtask 1

- 연산이 바뀔 때마다 전체 결과값을 시뮬레이션해서 다시 구합니다.
- O(QN) 시간에 문제를 해결할 수 있습니다.

D. 계산 최적화

Subtask 2

- 항상 모든 덧셈이 끝나고 나야 곱셈이 일어납니다. 연속한 덧셈 영역에 있는 연산들이 더하는 수치의 합을 p, 곱셈 영역에 있는 연산들이 곱하는 수치의 곱을 m이라고 합시다. 전체 연산의 결과는 p * m 이 됩니다.
- 연산의 변경이 일어날 때마다, 덧셈은 변경된 합 만큼을 더하거나 빼주고, 곱셈은 변경된 곱 만큼을 곱하거나 나눠주면 p와 m을 바로 갱신해줄 수 있습니다. 나눗셈 과정에서 modular inverse가 필요하므로 최대 O(logK) 시간이 필요합니다.
- 따라서, 전체 시간복잡도 O(N+QlogK)에 문제를 해결할 수 있습니다.

D. 계산 최적화

Subtask 4

- 덧셈과 곱셈을 묶어서, ax + b 형태로 생각해봅시다.
- 임의의 두 연산 ax + b와 cx + d를 연속해서 수행하면, 그 결과는 c(ax + b) + d = acx + bc + d가 됩니다.
- 따라서, 연산을 ax + b 형태로 표현할 경우 두 연산을 연속해서 수행한 결과도 마찬가지로 ax + b 형태로 표현할 수 있습니다.
- 또, 처음 시작할 때 x = 0이므로 결과값은 전체 연산을 하나로 합쳤을 때 마지막 b값과 같아집니다.

D. 계산 최적화

Subtask 4

- 이러한 형태의 쿼리와 업데이트는 각 위치의 연산을 ax + b 꼴로 나타내면 세그먼트 트리를 이용해 O(logN)시간에 계산이 가능합니다.
- 따라서, 전체 시간복잡도 O(N + QlogN) 시간에 문제를 해결할 수 있습니다.

E. 달팽이는 그늘에서 쉬고 싶다

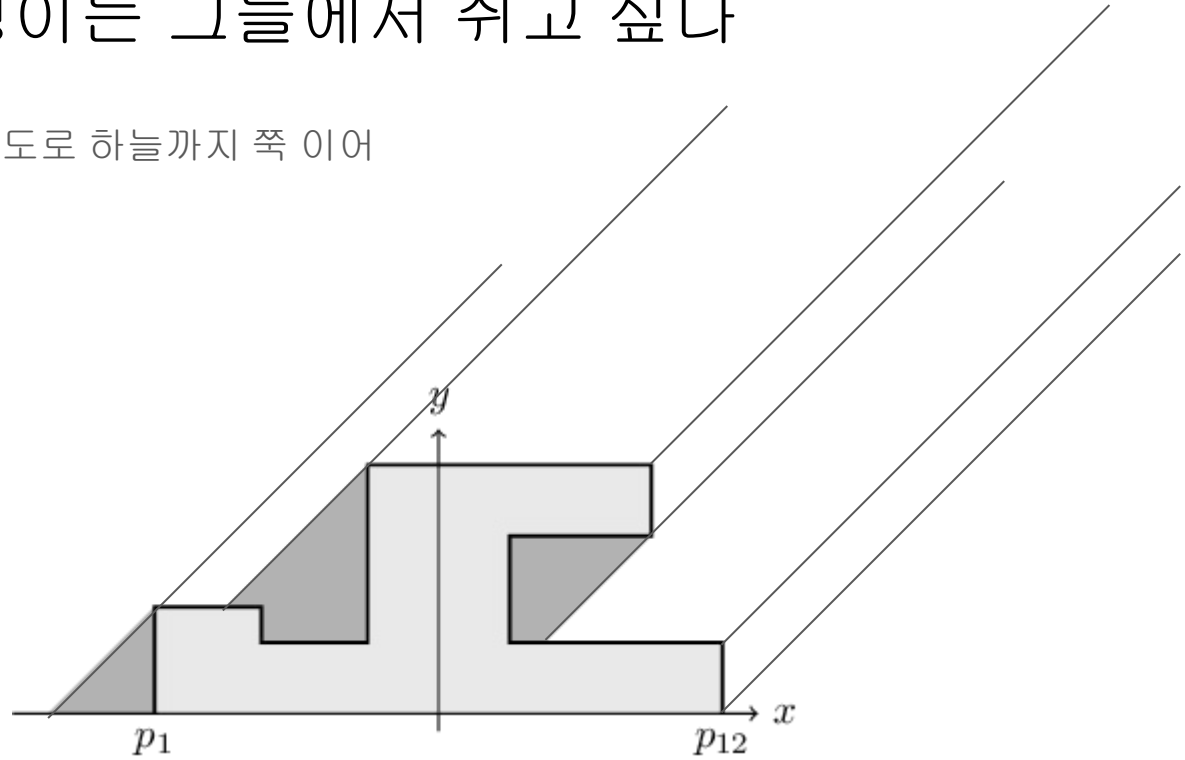
전체 길이 = |그림자길이| + |햇빛길이|

라서 햇빛길이를 알면 그림자길이를 알 수 있습니다.

햇빛 길이를 구하는게 더 쉽지 않을까요?

E. 달팽이는 그늘에서 쉬고 싶다

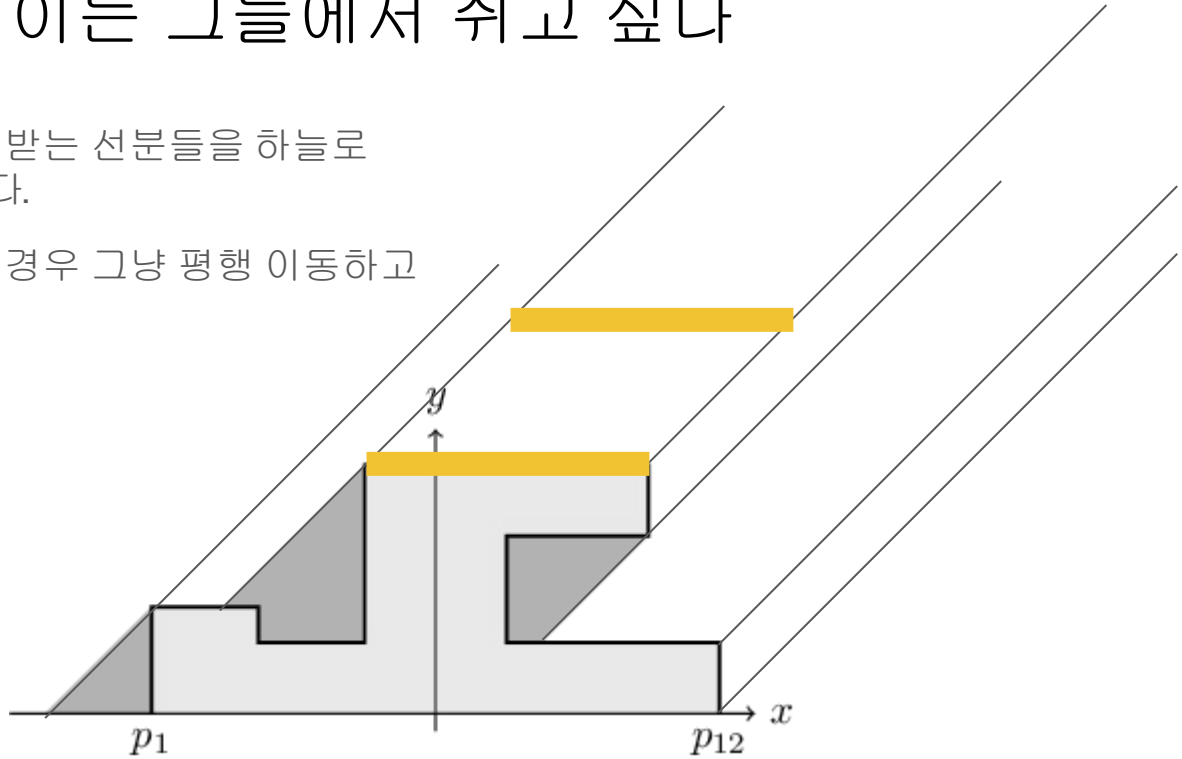
기준선을 45도로 하늘까지 쭉 이어 봅시다.



E. 달팽이는 그늘에서 쉬고 싶다

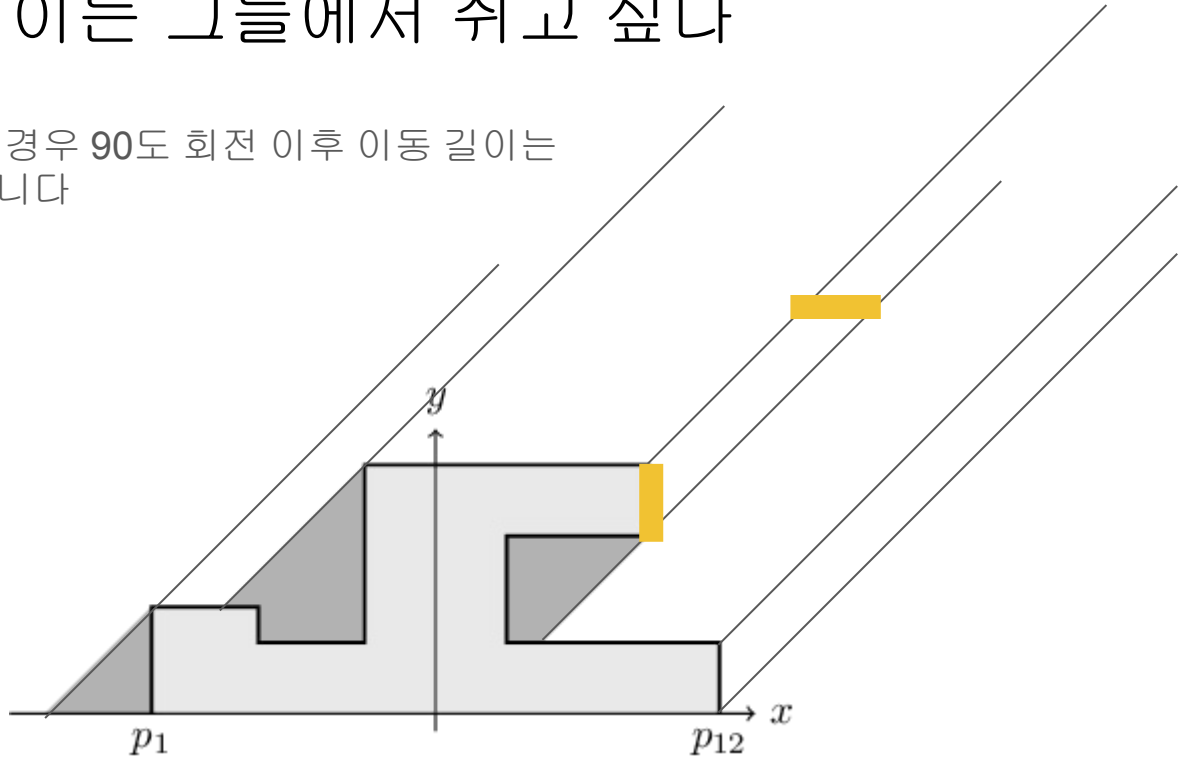
이제 햇빛을 받는 선분들을 하늘로 옮겨줄겁니다.

가로 선분의 경우 그냥 평행 이동하고



E. 달팽이는 그늘에서 쉬고 싶다

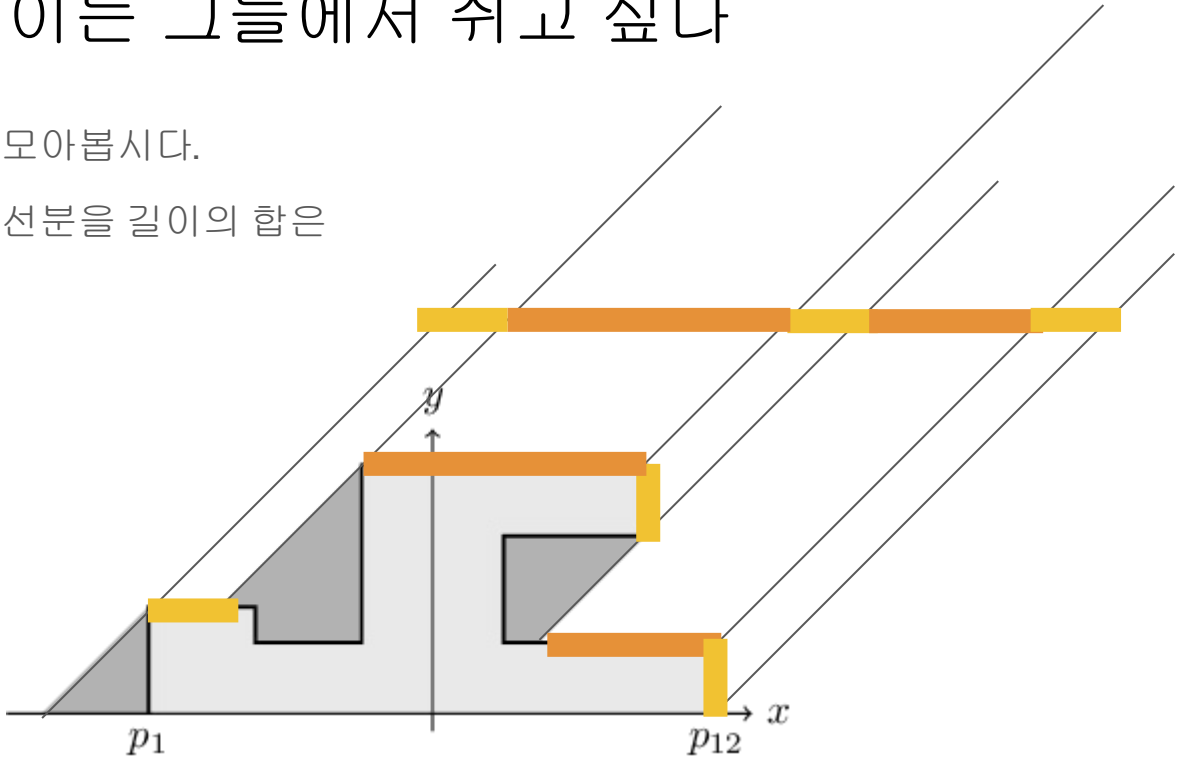
세로 선분의 경우 90도 회전 이후 이동 길이는
변하지 않습니다



E. 달팽이는 그늘에서 쉬고 싶다

이제 한줄로 모아봅시다.

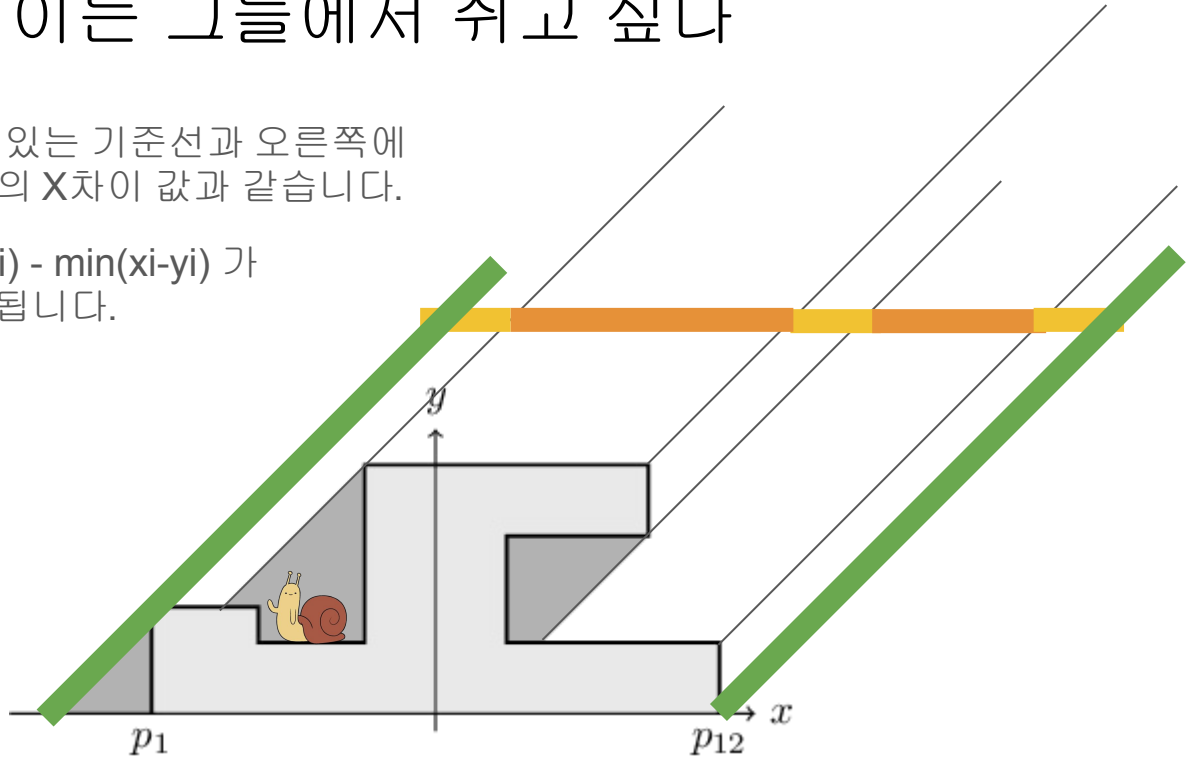
하늘에 있는 선분을 길이의 합은



E. 달팽이는 그늘에서 쉬고 싶다

제일 왼쪽에 있는 기준선과 오른쪽에 있는 기준선의 X차이 값과 같습니다.

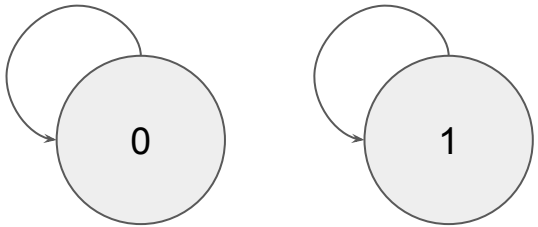
즉, max(xi-yi) - min(xi-yi) 가 햇빛길이가 됩니다.



F. 압축 프로그램

Subtask1

- 포인터를 이용해 참조하는 셀을 그래프처럼 생각해봅시다.
- 0번 메모리셀에 0, 1번 메모리셀에 1을 기록할 경우, 각 셀은 자기 자신을 참조합니다. 즉, 각 셀은 포인터를 몇 개를 붙이든 항상 0, 1을 출력합니다.



F. 압축 프로그램

Subtask1

- 데이터가 항상 000...0111...1 꼴로 나타나므로, 0이 a개, 1이 b개 있을 때 각각을 해당 개수만큼 순서대로 출력해주면 됩니다.
- 각각의 0과 1을 99개씩 잘라 한 그룹으로 생각해봅시다. 메모리셀에 2번 셀부터 순서대로 각 그룹이 0이면 0, 1이면 1을 기록합니다. 예를 들어 a가 198개, b가 297개가 있다면 2번셀부터 순서대로 00111이 기록됩니다.
- 이제 현재 출력할 그룹의 번호 g를 저장하는 메모리 셀을 하나 두고, 이 셀을 2로 초기화합니다.

F. 압축 프로그램

Subtask1

- *g, **g, ***g, … 를 순서대로 모두 출력해주면 해당 그룹에 해당하는 수를 99번 출력해줄 수 있습니다. 이렇게 출력한 후, g의 값을 1 올리면 다음 그룹으로 넘어가서 똑같은 작업을 반복해줄 수 있습니다.
- 정확히 99의 배수가 되지 않는 맨 앞 0은 0의 개수를 99로 나눈 나머지만큼 먼저 출력해주고 시작하면 됩니다.
- 따라서 전처리 과정에서 99 + a + (10000/99) 번 정도에 해당하는 명령어, 각 루프 단계에서 102개(start,end,출력,g 1 증가)의 명령어가 필요하므로 많아야 1000몇백개의 명령어면 충분히 답을 출력할 수 있습니다.

F. 압축 프로그램

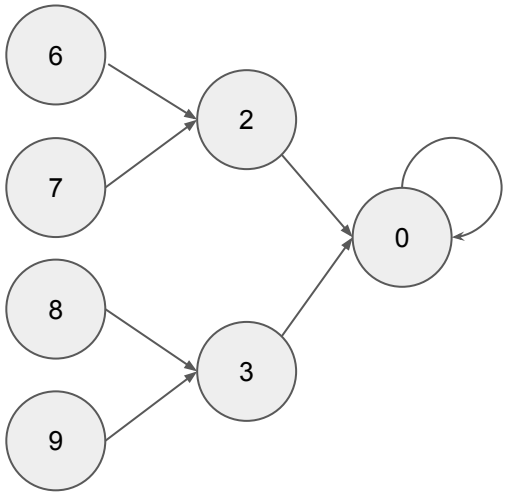
Subtask2

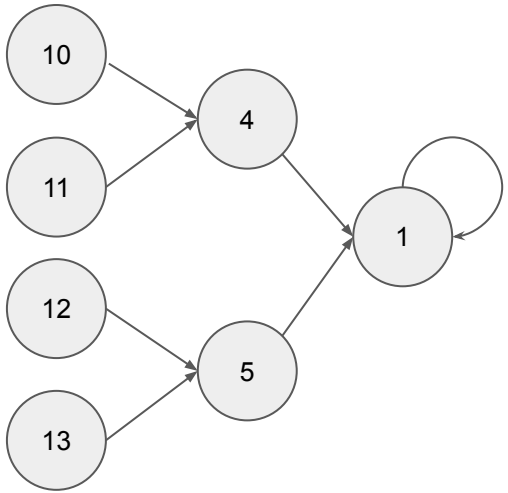
- Subtask1과 동일한 논리를 사용해봅시다. 명령어를 길이가 8인 구간으로 나누고, 길이가 8인 구간에서 나올 수 있는 모든 패턴을 저장해둔 후 그룹별로 어느 패턴에 속하는지 여부를 기록해서 출력합니다.

F. 압축 프로그램

Subtask2

- 길이가 k일 때 나올 수 있는 모든 패턴은 아래와 같은 방법으로 기록하면 효율적으로 저장 및 탐색을 할 수 있습니다.

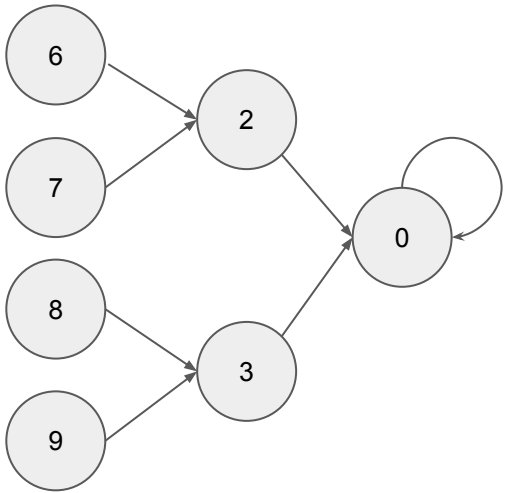


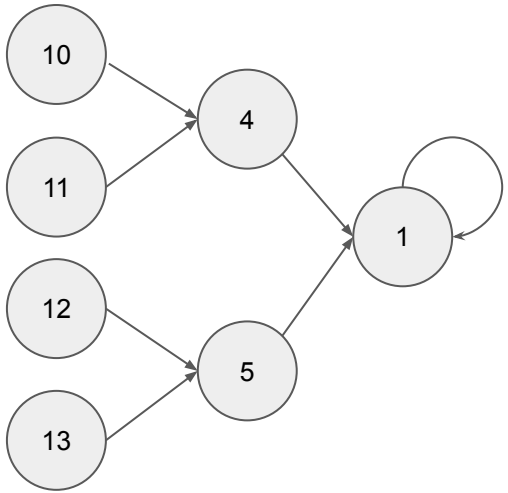


F. 압축 프로그램

Subtask2

- 아래 번호를 타고 가면서 2로 나눈 나머지를 계산해서 늘어놓으면 2^3개 패턴이 전부 나옴을 확인할 수 있습니다.





F. 압축 프로그램

Subtask2

- 따라서, 길이가 8인 패턴을 전부 기록하기 위해 512개의 명령어면 충분합니다.
- 여기에 나올 수 있는 그룹의 개수가 10000/8 = 1250개
- 출력 과정에서 필요한 명령어가 11개이므로
- 약 1800개 이내의 명령어로 모든 종류의 문자열을 표현할 수 있습니다.

G. 색종이 붙이기

Subtask1

- O(NM)가지 모든 색종이에 대해 해당 색종이를 이용해 빈칸을 전부 채울 수 있는지 확인해봅시다.
- 확인 과정에서 O(NM)개 모든 칸에 대해 그 위치에 색종이를 붙일 수 있으면 색종이가 덮는 모든 칸에 색칠을 해줍니다. 따라서 색칠 과정에 O(N^2M^2)의 시간이 필요합니다.
- 색칠이 안 된 칸이 없다면 해당 색종이로 모든 칸을 덮을 수 있습니다.
- 전체 시간 복잡도 O(N^3M^3)에 문제를 해결할 수 있습니다.

G. 색종이 붙이기

Subtask2

- 부분합 개념을 이용해 색종이를 붙일 수 있는지 확인하는 과정을 최적화할 수 있습니다.
- O(NM) 전처리를 거치고 나면 임의의 사각형 영역에 이미 색칠된 칸이 존재하는지 아닌지 O(1)에 판별할 수 있습니다.
- 따라서 각 색종이를 (x, y) 위치에 붙일 수 있는지 여부를 O(1)에 판별할 수 있고, 색종이를 붙였을 때 (x, y) ~ (x+w - 1, y+h - 1) 영역을 색칠하는 것 역시 부분합 개념을 이용해 O(NM)에 해결할 수 있습니다.
- 따라서, 색종이 크기가 정해졌을 때 전체를 색칠할 수 있는지 확인하는 과정을 O(N^2M^2)에서 O(NM)으로 최적화 가능합니다.

G. 색종이 붙이기

Subtask2

- 이렇게 최적화할 경우 전체 시간복잡도는 O(N^2M^2)이 됩니다.
- 여기서 어떤 (w,h) 크기의 사각형으로 전체를 덮는게 가능할 경우 (w,h-1) 크기의 사각형으로도 전체를 덮을 수 있다는 점에 착안해봅시다.
- w가 고정됐을 때 최대한 큰 h값을 찾으면 1..h까지의 모든 사각형으로 빈 칸을 전부 덮을 수 있습니다.
- 이러한 h는 이분탐색으로 최댓값을 O(logM) 시간에 찾을 수 있으므로, 전체 시간복잡도 O(N^2MlogM) 시간에 문제를 해결할 수 있습니다.

G. 색종이 붙이기

Subtask3

- 색종이 너비가 w일 때 가능한 최대 높이 값을 x라고 합시다.
- 너비가 w + 1일 때 가능한 최대 높이 값 y에 대해, x >= y가 항상 성립합니다.
- 즉, 너비가 늘어나면 최대 높이는 항상 감소합니다.
- w = 1일 때 가능한 최대 높이를 구하고, w를 늘려나가면서 최대 높이값을 이전 w에서의 높이부터 시작해서 줄여나가며 계산합시다.
- w가 증가하면 h는 감소하므로, 체크하는 최대 횟수는 O(N+M)이 됩니다.
- 따라서, 전체 시간복잡도 O(NM(N+M))에 문제를 해결할 수 있습니다.

G. 색종이 붙이기

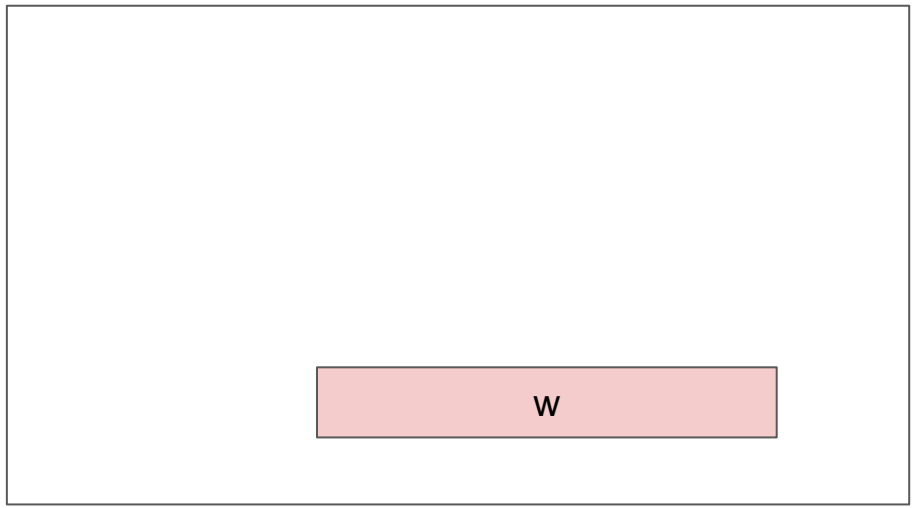
Subtask4

- maxH[k] = 너비가 k인 색종이를 이용해서 빈칸을 모두 덮을 수 있는 색종이의 최대 높이 라고 정의해봅시다.
- 이 때 정답은 maxH[1] + maxH[2] + … + maxH[W] 가 됩니다. 따라서, 이 배열의 값을 빠르게 계산할 수 있다면 정답도 빠르게 구할 수 있습니다.
- 그러니 maxH 배열의 값을 빠르게 계산하는 방법을 생각해봅시다.

G. 색종이 붙이기

Subtask4

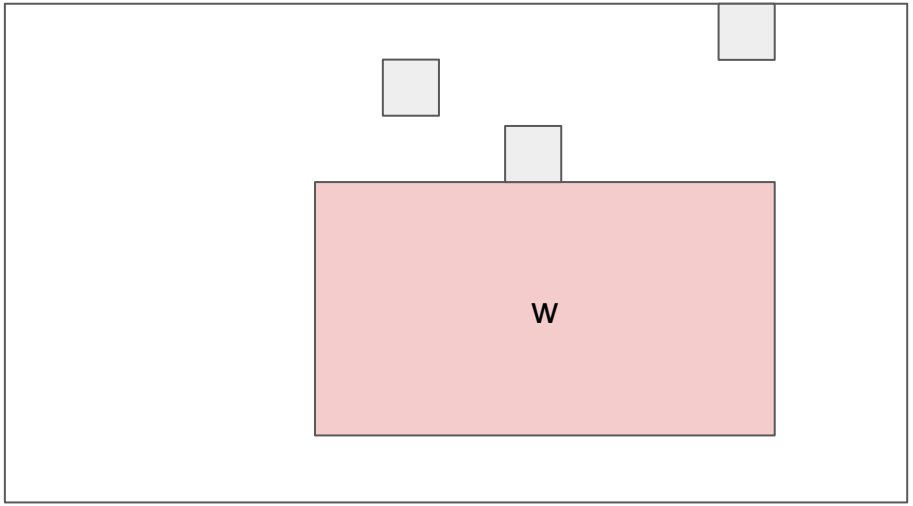
- 너비가 W인 색종이를 어딘가에 붙였다고 생각해봅시다.



G. 색종이 붙이기

Subtask4

- 이미 색칠된 칸들에 대해, 해당 색종이는 이미 색칠된 칸 중 w범위 내에서 위쪽에서 제일 가까운 위치까지만 위로 뻗을 수 있습니다.



G. 색종이 붙이기

Subtask4

- 따라서, up[x][y] = (x, y) 위치를 기준으로 위쪽에서 가장 가까운 색칠된 칸까지의 거리 라고 정의하면,
- 어떤 (x, y) 에서 (x+w -1,y)까지 좌우로 w길이의 색종이를 놓을 경우, 이 색종이가 가질 수 있는 최대 높이는 min(up[x][y], up[x+1][y], …, up[x+w-1][y]) 가 됩니다.
- 이 때 모든 (x, y)칸에 대한 up[x][y] 값은 O(NM) 시간에 부분합 개념을 이용해서 전처리해둘 수 있습니다.

G. 색종이 붙이기

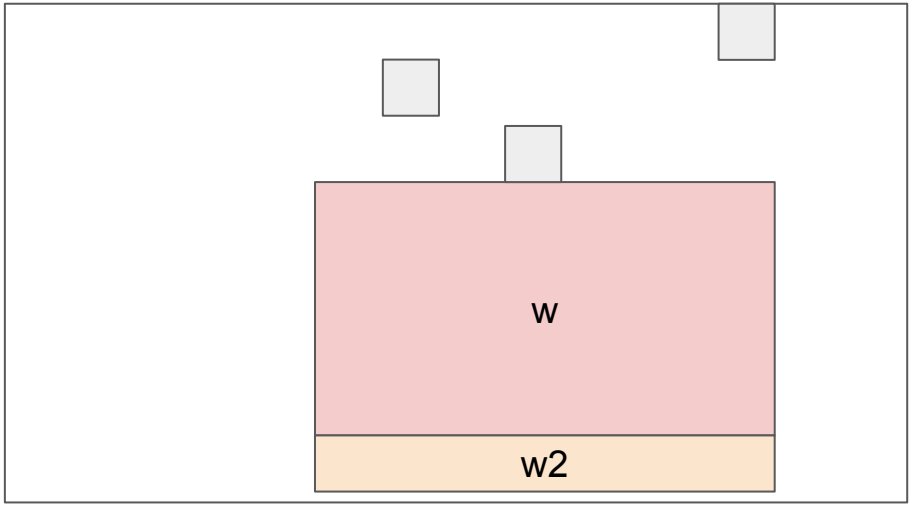
Subtask4

- 색종이를 아랫변이 (x, y) 에서 (x+w-1, y)까지 오게 배치하고, 여기서 최대한 뻗을 수 있는 높이가 h였다고 합시다.
- 이 때 색종이를 한칸 더 아래쪽에도 배치할 수 있다고 합시다. 즉, (x,y-1)에서 (x+w-1,y-1) 영역에도 배치가 가능합니다.
- 이 경우, 해당 영역을 모두 덮을 수 있는 사각형의 최대 높이는 항상 h+1로 확장이 가능합니다.

G. 색종이 붙이기

Subtask4

- 오른쪽 그림과 같이 w2 위치에 색종이가 오게 배치하고, 위로 h+1만큼 뻗을 경우 동일한 영역을 항상 덮을 수 있기 때문입니다.



G. 색종이 붙이기

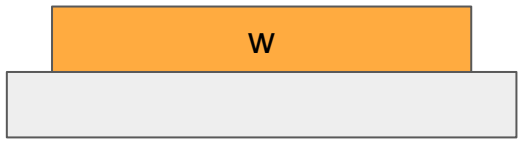
Subtask4

- 따라서, 해당 위치보다 더 아래쪽에 w 너비의 색종이를 붙일 수 있는 경우는 답에 영향을 주지 않는다는 것을 알 수 있습니다.
- 즉, 답에 영향을 주는 위치는 항상 하나 이상의 색칠된 칸과 접하는 위치여야 합니다.
- 이제 더 아래쪽에 w 너비의 색종이를 붙일 수 없는 경우들에 대해 생각해 봅시다.

G. 색종이 붙이기

Subtask4

case1 : w 너비의 색종이가 w보다 더 긴 색칠된 칸에 좌우로 포함되게 위치하는 경우

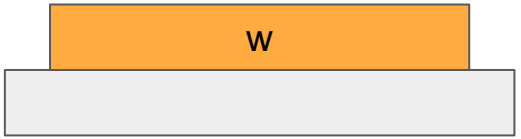


G. 색종이 붙이기

Subtask4

case1 : w 너비의 색종이가 w보다 더 긴 색칠된 칸에 좌우로 포함되게 위치하는 경우

- 이러한 경우는 항상 전부 색종이를 붙여야만 모든 위치를 덮을 수 있습니다.
- 따라서 색칠된 칸 바로 위에 해당하는 위치들은 반드시 모두 고려해주어야 합니다.



G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

모든 칸을 덮을 수 있으려면, 색칠된 칸들 사이에도 항상 좌우로 빈 칸 간격이 w칸 이상 존재해야 합니다.

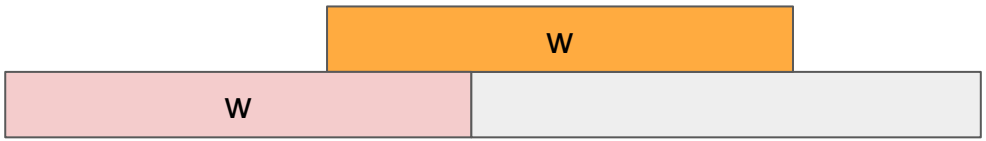
G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

모든 칸을 덮을 수 있으려면, 색칠된 칸들 사이에도 항상 좌우로 빈 칸 간격이 w칸 이상 존재해야 합니다.

즉, 아래 빨간 영역의 w도 항상 배치가 가능합니다.

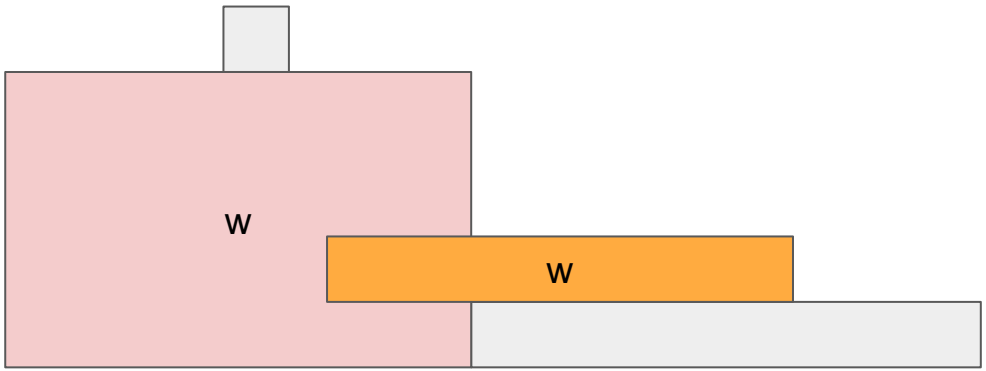


G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

여기서 빈칸 영역에서 위쪽으로 붙일 수 있는 최대 높이가 얼마인지는 계산 가능합니다.

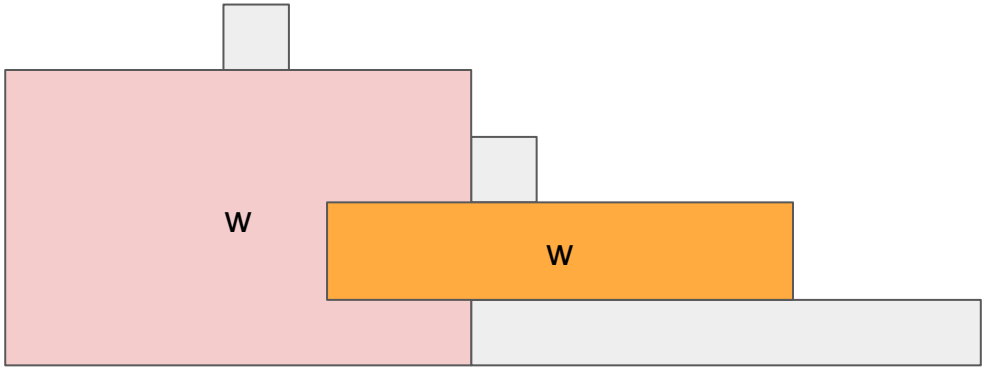


G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

만약 가운데 높이가 이것보다 낮다면, 삐져나오게 붙이는 경우는 신경 쓸 필요가 없습니다

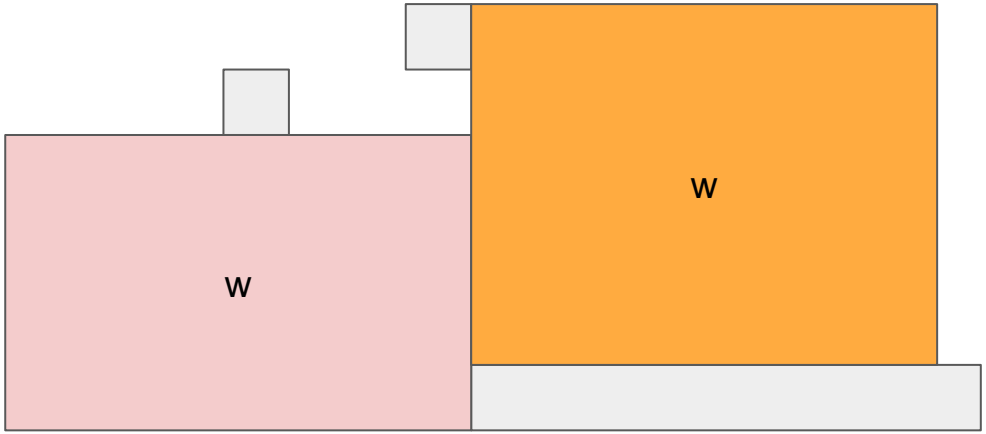


G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

더 높은 영역이 있는 경우, 해당 중간 영역을 걸쳐서 채워주지 않으면 안 됩니다.

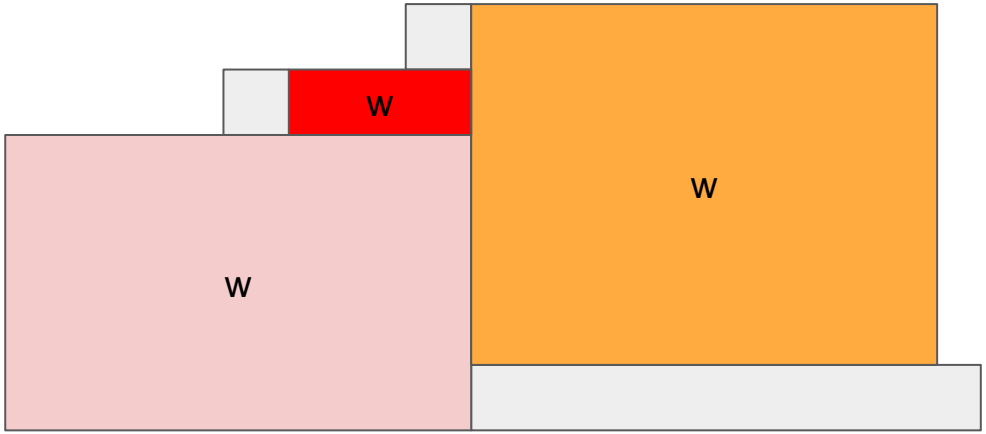


G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

더 높은 영역이 있는 경우, 해당 중간 영역을 걸쳐서 채워주지 않으면 안 됩니다.

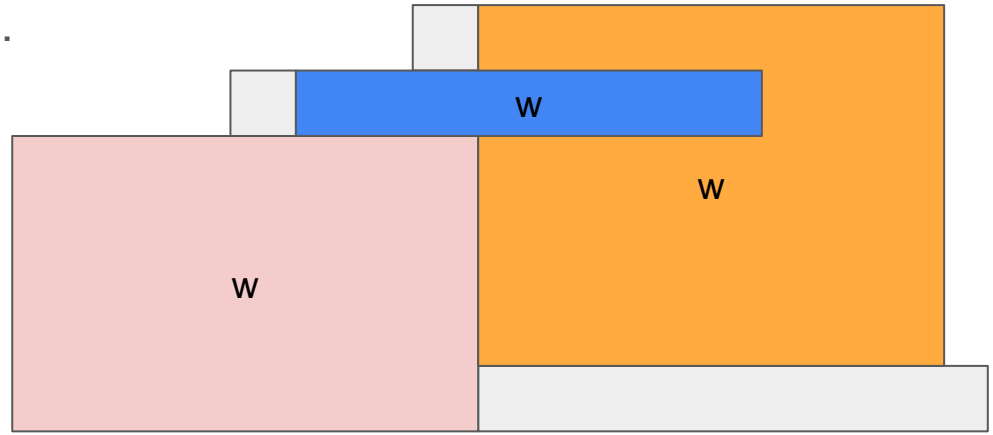


G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

빨간 영역을 채우기 위해서는 해당 위치를 포함하게 w 너비의 색종이를 놓아야만 합니다.

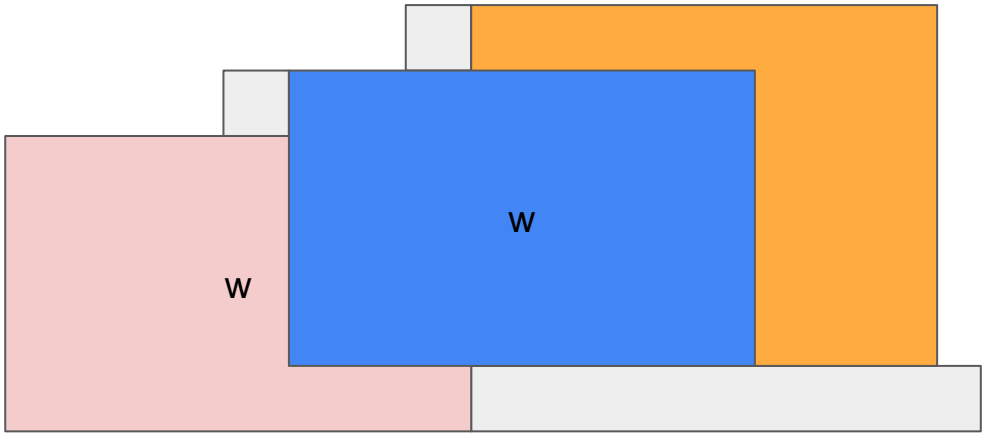


G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

최대한 아래로 내려서 보면,

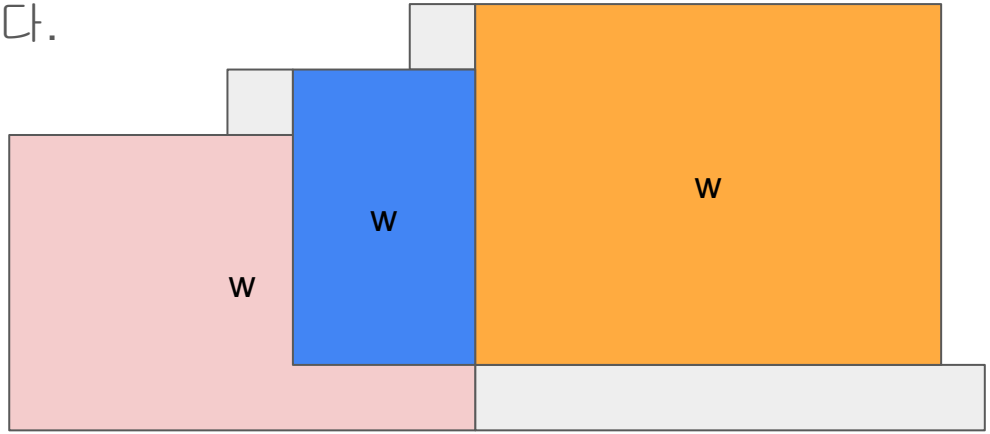


G. 색종이 붙이기

Subtask4

case2 : 색칠된 칸에서 한쪽으로 걸치게 붙이는 경우

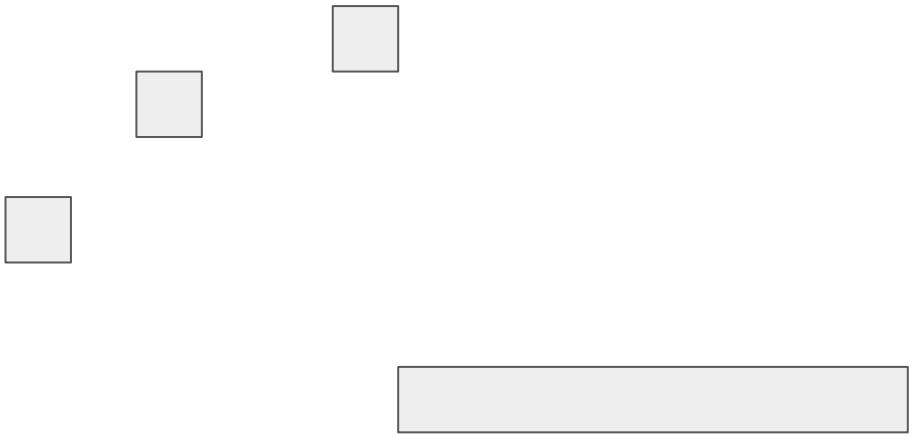
최대한 아래로 내려서 보면, 걸쳤을 때 고려해야하는 영역은 파란색 영역과 같습니다.



G. 색종이 붙이기

Subtask4

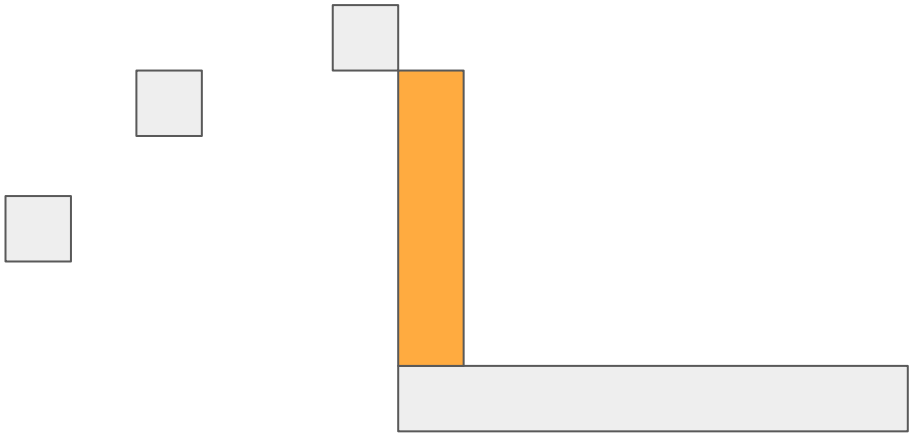
색종이의 너비에 따라 이렇게 걸치는 영역이 포함되는 경우를 생각해봅시다.



G. 색종이 붙이기

Subtask4

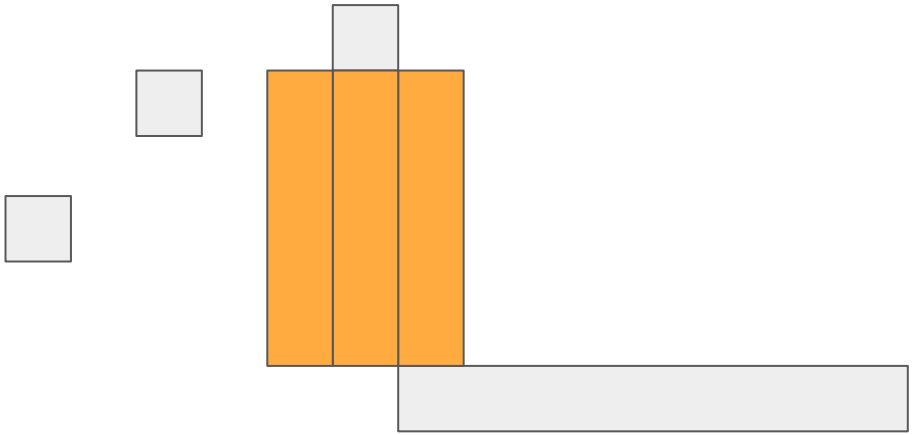
w = 1인 경우 걸치게 붙일 수 없으므로 걸치게 붙이는 경우를 고려할 필요가 없습니다



G. 색종이 붙이기

Subtask4

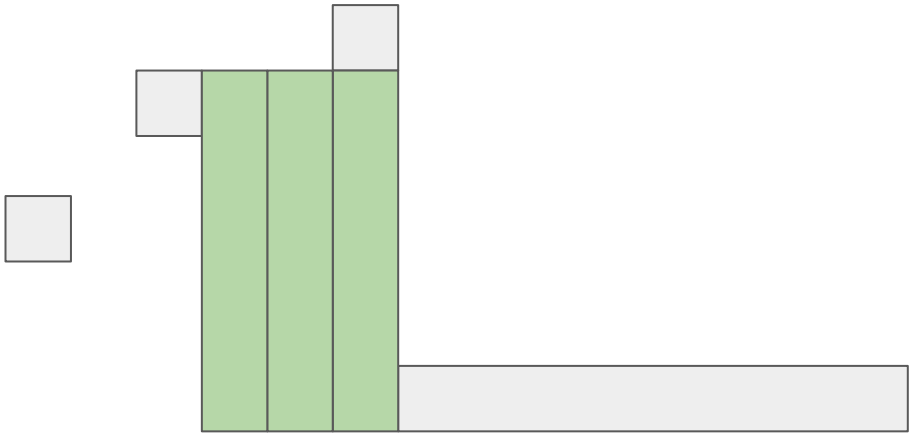
w =2, w = 3일 때에도 여기서 걸치게 붙일 필요는 없습니다. 아래와 같이 걸쳐서 놓는 경우를 생각해봅시다.



G. 색종이 붙이기

Subtask4

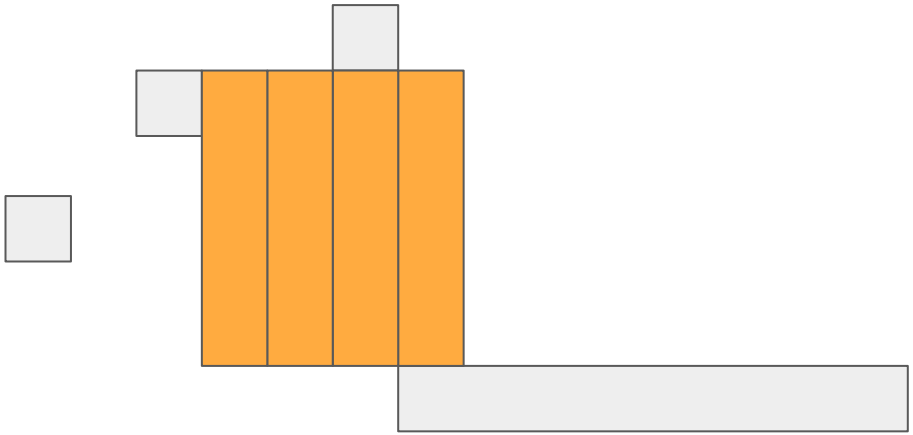
이 경우는 빈공간 쪽에서 붙이는 것으로 걸쳐서 붙이는 영역을 모두 색칠할 수 있으므로, 여기서 고려해줄 필요가 없습니다.



G. 색종이 붙이기

Subtask4

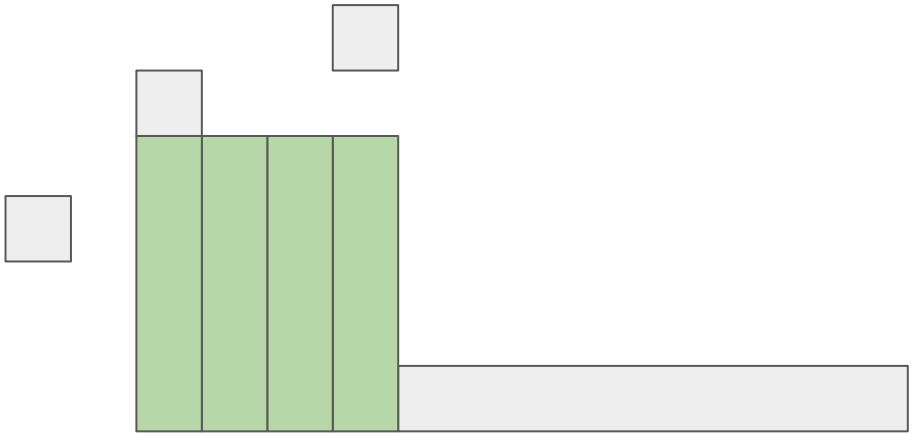
w = 4일 때를 생각해봅시다.



G. 색종이 붙이기

Subtask4

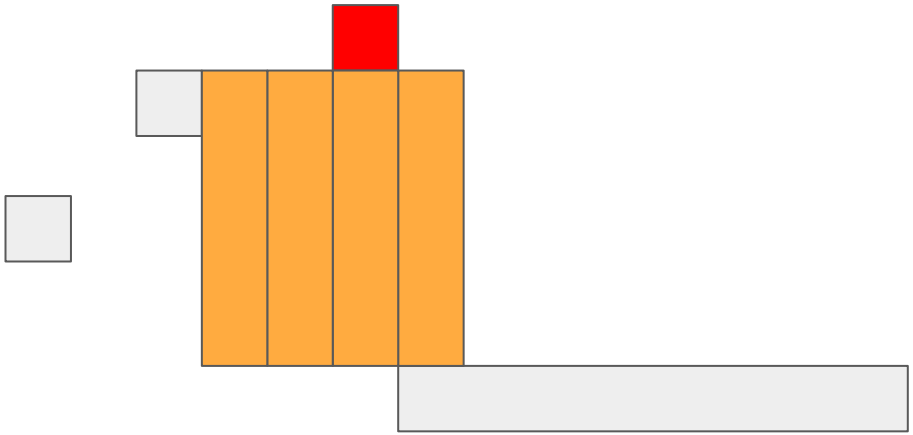
여기서는 앞서 언급했던 것과 같이 아래쪽 영역에 붙이는 걸로는 색칠할 수 없는 공간이 생깁니다.



G. 색종이 붙이기

Subtask4

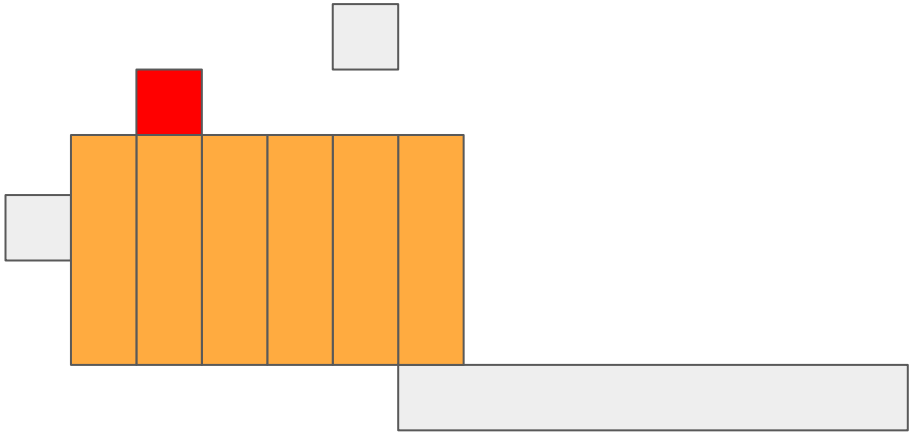
따라서, w >= 4 인 경우 빨갛게 칠한 벽까지의 높이를 고려해주어야만 합니다
(maxH값이 이 벽까지의 높이보다 커질 수 없습니다)



G. 색종이 붙이기

Subtask4

마찬가지 논리로, w >= 6일 때 maxH는 아래 빨간 벽까지의 높이보다 높아질 수 없습니다.



G. 색종이 붙이기

Subtask4

왼쪽으로 빈 공간에 k칸만큼 걸치게 색종이를 놓았다고 가정합시다. 이렇게 걸쳐서 놓을 수 있는 색종이의 최소 너비는 k+1입니다. 색종이의 오른쪽 끝지점이 (x,y) 라고 하면, 색종이를 놓을 수 있는 높이는

min(up[x][y], up[x-1][y], … , up[x-k][y])

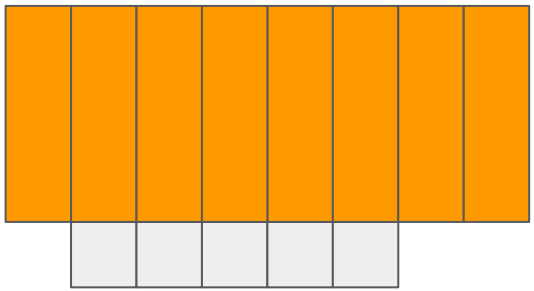
가 됩니다. 이 때 걸쳐서 놓지 않고 아래쪽 빈 공간에서 붙일 수 있는 색종이의 높이는 min(up[x-1][y], up[x-2][y], … , up[x-k-1][y]) 까지의 값에 의해 결정됩니다. 이 값보다 앞의 값이 더 크다면, 너비가 k+1 이상일 때는 이렇게 걸쳐서 붙이는 경우를 고려해주어야만 합니다(앞의 그림 케이스 참조).

G. 색종이 붙이기

Subtask4

case3 : 색칠된 칸에서 양쪽으로 튀어나오게 붙이는 경우

w가 충분히 큰 경우(w가 아래쪽 색칠된 칸의 너비보다 큰 경우 한쪽으로만 걸치는게 아니라 양쪽으로 튀어나오는 경우가 생깁니다.

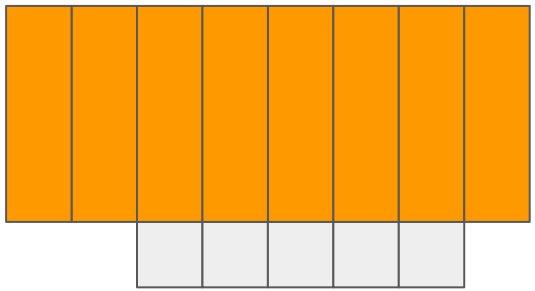


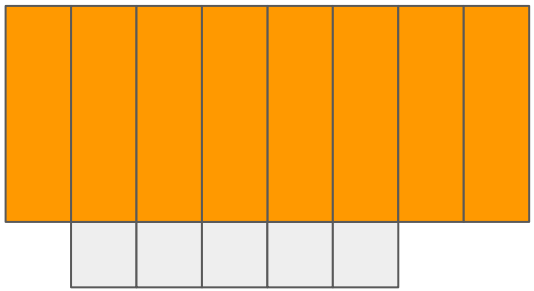
G. 색종이 붙이기

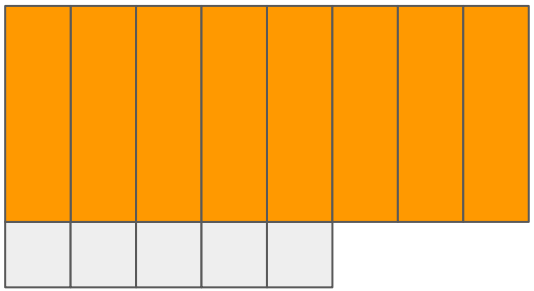
Subtask4

case3 : 색칠된 칸에서 양쪽으로 튀어나오게 붙이는 경우

이 경우, 색종이를 어떻게 붙여도 (w - (아래쪽 색칠된 칸의 너비)) 칸 만큼은 좌우 빈 공간에 걸칠 수 밖에 없습니다.







G. 색종이 붙이기

Subtask4

case3 : 색칠된 칸에서 양쪽으로 튀어나오게 붙이는 경우

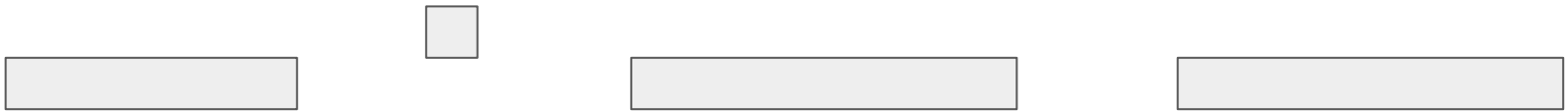
따라서, 아래쪽 색칠된 칸의 너비를 k라고 하고 w > k 인 경우, 좌우 영역에서 (w-k)개의 칸 만큼은 반드시 고려를 해야합니다. 즉, 좌우에서 인접한 (w-k)개의 칸을 뽑는 경우 중 가장 높이가 높은 경우 보다 더 높이가 높아질 수는 없습니다.

G. 색종이 붙이기

Subtask4

이 성질들을 이용해서 답을 빠르게 계산해봅시다.

우선, 같은 y 라인에 해당하는 좌표들에 대해 아래쪽에 색칠된 칸이 있는 좌표들을 기준으로 구간을 나눕니다.

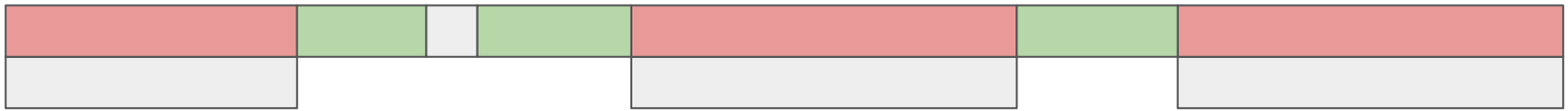


G. 색종이 붙이기

Subtask4

이 성질들을 이용해서 답을 빠르게 계산해봅시다.

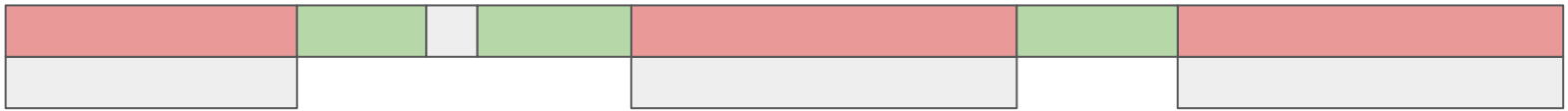
우선, 같은 y 라인에 해당하는 좌표들에 대해 아래쪽에 색칠된 칸이 있는 좌표들을 기준으로 구간을 나눕니다.



G. 색종이 붙이기

Subtask4

이렇게 구간을 나눈 경우, maxH는 빨간 칸에 해당하는 모든 up[x][y]보다 작거나 같은 값이어야 합니다.

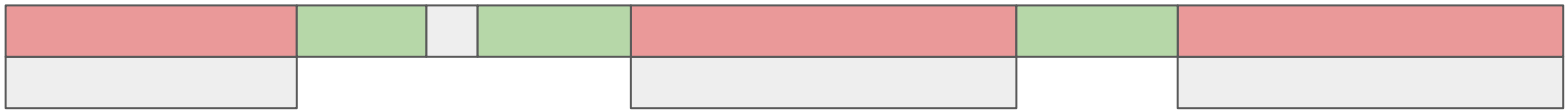


G. 색종이 붙이기

Subtask4

이렇게 구간을 나눈 경우, maxH는 빨간 칸에 해당하는 모든 up[x][y]보다 작거나 같은 값이어야 합니다.

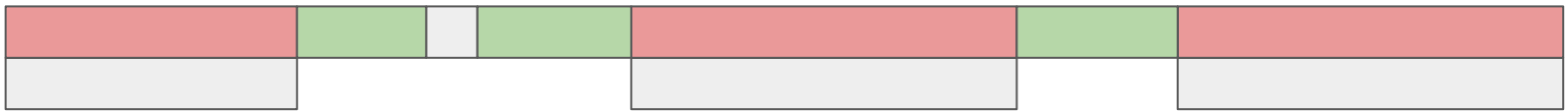
w 값과 무관하게 항상 해당 칸 위에는 사각형을 배치해야하고, 이 경우의 최대 높이는 up[x][y]중 최솟값보다 커질 수는 없기 때문입니다.



G. 색종이 붙이기

Subtask4

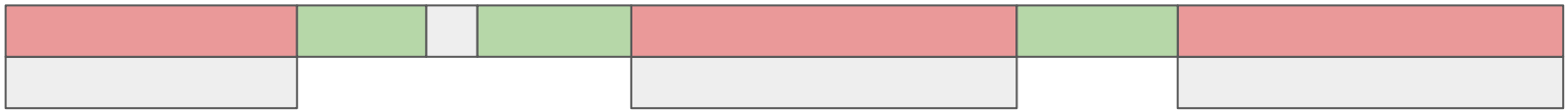
이제 초록색 영역에서 답에 포함되는 경우만 계산해주면 되는데, 이 경우는 Case2 또는 Case 3에 해당합니다.



G. 색종이 붙이기

Subtask4

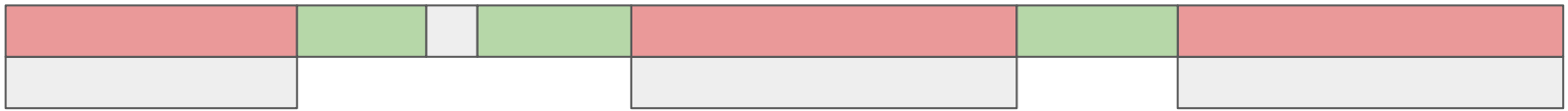
Case2의 경우, 한쪽 방향으로 w=2부터 순서대로 늘려나가며 앞서 설명한 케이스처럼 가능한 높이가 감소해서 무조건 해당 위치에 색종이를 걸쳐서 붙여야 하는 경우가 있는지 확인해줍니다.



G. 색종이 붙이기

Subtask4

Case3의 경우도 비슷하게, 양쪽으로 튀어나올 수 밖에 없는 너비에서부터 시작해서 높이가 최대한 덜 감소하게 하는 칸을 좌우에서 하나씩 선택하면서 너비를 늘려나가면 가능한 최대 높이를 계산해줄 수 있습니다.



G. 색종이 붙이기

Subtask4

각각의 영역을 기준으로 생각해보면, 빨간색 영역은 한 번씩만 처리합니다.

초록색 영역의 경우 빨간색 영역을 기준으로 걸쳐서 놓아야하는지 여부를 두 번씩 (Case 2, Case 3) 판단해봐야하기 때문에, 좌우 빨간색 영역에서 각각 두 번씩 처리가 필요합니다.

따라서 빨간색 영역은 최대 한 번, 초록색 영역은 최대 네 번만 방문하면 maxH값을 정확히 계산할 수 있으므로 O(NM)에 문제를 해결할 수 있습니다.

H. 테스트 케이스 만들기

Subtask1

경우의 수가 충분히 크고 MOD가 소수이기 때문에 랜덤으로 찾을 수 있지 않을까 라는 느낌이 듭니다.

실제로 50 x 50 그리드에서 가장자리를 제외하고 벽 확률을 20% 정도로 설정하면

0부터 1008 사이에 모든 테케를 0.5초 안에 찾을 수 있습니다.

H. 테스트 케이스 만들기

Subtask2

P = 1,000,000,009면 Subtask1처럼 다 해보기엔 시간이 너무 오래 걸립니다.

그냥 원하는 경우의 수가 딱 나오도록 할 수는 없을까요?

경우의 수를 컨트롤하는 방법을 찾아봅시다.

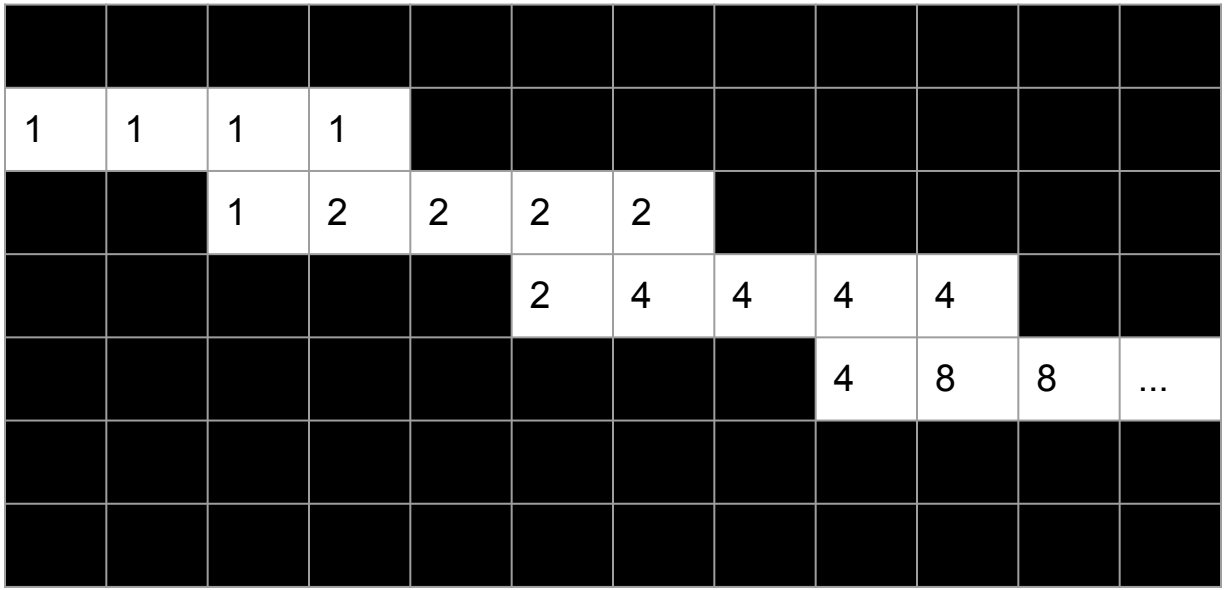
H. 테스트 케이스 만들기

경우의 수 2배로 불리기

X	X	X	
	X	2X	2X

H. 테스트 케이스 만들기

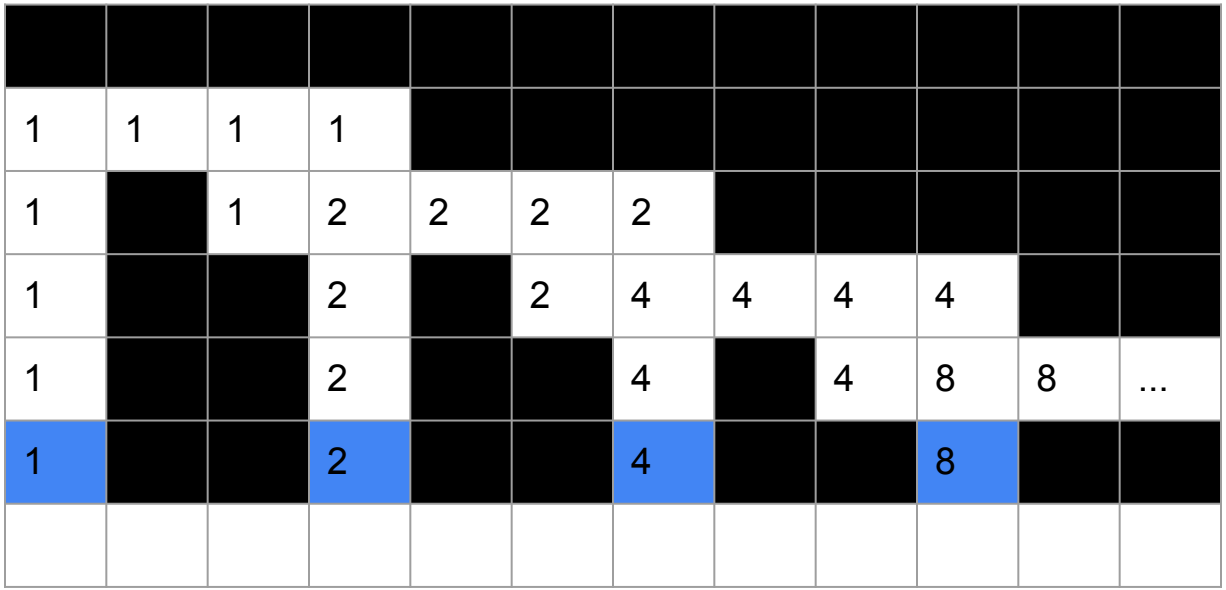
여러개 붙여 봅시다.



H. 테스트 케이스 만들기

이제 2^k 칸들을 조합하여 임의의 경우의 수를 만들 수 있습니다.

아래로 구멍을 뚫어서 우측하단까지 도달하도록 해줍니다.



H. 테스트 케이스 만들기

조금 더 컴팩트하게 만들 수 있습니다.

아래와 같이 만들면 비트 하나당 가로 2칸 세로 1칸이 늘어나니 n=32 m=64 정도가 되겠고 32+64 = 96 <= 100으로 해결이 가능합니다.