제 2회 소프트콘

풀이

A. 보물 찾기

정답자 수 : 55명(73.3%)

가장 처음 푼 사람 : ainta, 3분

출제 및 풀이 : cnhs2205

A. 보물 찾기

격자점에서, (a,b)위치에서 (x,y) 위치가 보이려면 (x-a)와 (y-b)가 서로소여야 한다. 이 규칙을 이용하여 생각하면, 몇 가지로 경우를 나누어 문제를 풀 수 있다.

subtask1

두 수가 같은 경우, 두 값이 모두 0이면 답은 0이고 둘 다 1 혹은 -1이면 답이 1인 것은 자명하다. 그 외의 경우는 항상 두 번이면 목적지에 도달할 수 있는데, 두 수 중 하나를 1로 고정하면 나머지 수와는 항상 서로소가 되고, 따라서 (0,0) -> (1,y-1) -> (x,y) 로 이동하면 항상 두 번만에 목적지에 도착할 수 있기 때문이다.

A. 보물 찾기

subtask2

이제 x != y 인 경우도 생각해보자. 우선 부호는 중요하지 않기 때문에(둘 다 양수인 경우를 해결하면 마찬가지 방식으로 나머지 경우도 해결 가능하다), 두 수가 모두 양수라고 가정하고 풀어 보자. 이 경우도 x == y 인 경우랑 유사한 방식으로 풀 수 있는데, 두 가지 경우로 나눠서 생각할 수 있다.

1. gcd(x,y) == 1인 경우 : 두 값이 서로소인 위치이므로, 한 번만에 도달 가능하다.
2. 그 외의 경우 : x==y에서 다뤘던 경우와 마찬 가지로, (0,0) -> (1,y-1) -> (x,y) 로 이동하면 항상 두 번만에 도달 가능하다.

B. 명상 방해꾼

정답자 수 : 38명(50.6%)

가장 처음 푼 사람 : august14, 8분

출제 및 풀이 : djm03178

B. 명상 방해꾼

subtask1

브루트포스로 풀 수 있다. 각 새를 제외하고 나머지 새들에 대해 매 초마다 영향을 합한 절댓값이 현재까지 찾은 답보다 더 크면 갱신해주는 식으로 하면 된다. 각 새에 대해 찾은 최댓값들 중 그 값이 가장 작은 새를 택하면 된다. O(N^2*M)이기 때문에 조금은 효율적인 구현이 필요하다. (N을 더 줄였어야 했는데 못 뚫으신 분들 죄송합니다 ㅠㅠ)

B. 명상 방해꾼

subtask2

각 새가 미치는 영향의 prefix sum을 구하면서 진행하면, 시간 순으로 한 번만 탐색해도 답을 구할 수 있다. 즉, 각 초에 대해 그 때까지 모든 새가 미치는 영향의 합을 구한 뒤, 거기서 각 새의 prefix sum을 제외시킨 값이 해당 새를 잡았을 때의 영향의 최대를 갱신하는지 판별하고, 탐색을 끝낸 뒤 모든 새들 중에 그 값이 가장 작은 새를 택하면 된다. 시간복잡도가 O(NM)으로 줄어들게 된다.

C. 상자 열기

정답자 수 : 26명(34.6%)

가장 처음 푼 사람 : ainta, 13분

출제 및 풀이 : jwvg0425

C. 상자 열기

모든 버튼 집합을 순서대로 눌렀을 때 나온 소리에 대해서, 올바른 버튼이 서로 다른데 같은 소리가 나면 그 두 버튼에 대해 답이 무엇인지 구분할 수 없게 된다. 따라서, 내가 버튼들을 누른 횟수(띵동/삐삐 소리를 들은 횟수)가 k번이라면 최대 2^k개의 버튼들을 서로 구분할 수 있다.

따라서, n<=2^k인 가장 작은 k개의 버튼 집합으로 구분하는 경우가 가능한 최솟값이고, 항상 이 경우를 만족하는 답을 만들 수 있다.

C. 상자 열기

각각 눌러볼 버튼 집합에 대해서, 띵동 소리를 내면 T, 삐삐 소리를 내면 F라고 해보자. 예를 들어 눌러볼 버튼 집합이 2가지이고 총 버튼이 4개인 경우 아래와 같은 표로 나타낼 수 있다.

	버튼 쌍 1	버튼 쌍 2
1번 버튼이 올바른 경우	T	T
2번 버튼이 올바른 경우	T	F
3번 버튼이 올바른 경우	F	T
4번 버튼이 올바른 경우	F	F

C. 상자 열기

여기서, 어떤 버튼 x가 올바른 버튼일 때 버튼 집합 y에서 띵롱 소리가 났다는 건 버튼 집합 y가 버튼 x를 포함하고 있다는 이야기이다(올바른 버튼이 누른 버튼 집합에 포함되어야 띵동 소리가 나므로)

	버튼 쌍 1	버튼 쌍 2
1번 버튼이 올바른 경우	T	T
2번 버튼이 올바른 경우	T	F
3번 버튼이 올바른 경우	F	T
4번 버튼이 올바른 경우	F	F

C. 상자 열기

따라서, 아래와 같은 방식으로 표를 만든 후 표를 위에서 아래로 읽으며 묶어주면 눌러야 하는 버튼 집합을 구할 수 있다.

1,2번 버튼 1,3번 버튼

	버튼 쌍 1	버튼 쌍 2
1번 버튼이 올바른 경우	T	T
2번 버튼이 올바른 경우	T	F
3번 버튼이 올바른 경우	F	T
4번 버튼이 올바른 경우	F	F

C. 상자 열기

N개의 버튼을 구분하기 위해 logN개의 버튼 집합이 필요하므로, 전체 시간복잡도 O(NlogN) 시간에 답을 구할 수 있다.

subtask1의 경우, 손으로 직접 n=6까지의 케이스에 대해 답을 모두 구해서 출력하는 방식으로 점수를 얻을 수 있도록 의도했다.

D. 국제 소 줄서기 사진 콘테스트

정답자 수 : 13명(17.3%)

가장 처음 푼 사람 : ainta, 21분

출제 및 풀이 : jwvg0425

D. 국제 소 줄서기 사진 콘테스트

subtask1

M개의 자리 바꿈 각각에 대해, 모든 연속한 부분 구간들을 다 확인하면서 암소와 수소의 숫자가 같은 가장 긴 부분 구간이 무엇인지를 탐색하면 O(MN^2)의 시간복잡도로 답을 구할 수 있다.

D. 국제 소 줄서기 사진 콘테스트

subtask2

수소는 1, 암소는 -1로 수열의 값을 변경해서 생각해보자. 이 경우 합이 정확히 0이 되는 가장 긴 연속한 부분 구간의 길이를 구해야 하는 문제로 생각할 수 있다.

이 때, sum[x] = 1~x까지 수열의 합

으로 두면, x번째에서 y번째 위치까지의 연속한 부분 구간의 합은

sum[y] - sum[x-1]

로 표현할 수 있다.

D. 국제 소 줄서기 사진 콘테스트

subtask2

주어진 문제는 sum[y] - sum[x-1] = 0 이 되는 가장 긴 길이를 구하는 것인데, 이 식을 변형하면 sum[y] = sum[x-1]이 되므로, sum 배열의 값이 서로 같은 위치들 중에 서로 가장 멀리 떨어져 있는 위치 간의 거리가 곧 답이 된다.

sum 배열은 N번의 순회로 구할 수 있고,

left[x] = sum[k] 가 x인 가장 왼쪽 위치, right[x] = sum[k]가 x인 가장 오른쪽 위치

라고 정의하면 각각의 sum 배열 값에 대해 left[x], right[x]값도 모두 구할 수 있다.

D. 국제 소 줄서기 사진 콘테스트

subtask2

이제 모든 x에 대해 right[x] - left[x]의 값 중 가장 큰 값이 곧 답이 되는데, sum 배열은 부분합 배열이므로 항상 -n ~ n 사이 범위를 가지게 된다. 따라서 모든 sum 배열의 값을 다 순회하면서 right[x] - left[x]의 값중 최댓값을 구해주면 이게 답이 된다.

소들이 자리를 바꿀 때마다 이 과정을 수행해주면 각각에 대해 O(N) 시간에 답을 구할 수 있으므로, 총 시간복잡도는 O(MN)이 된다.

D. 국제 소 줄서기 사진 콘테스트

subtask3

p번째 소와 p+1번째 소가 서로 위치를 바꾼다고 했을 때, sum 배열에서 실제로 값이 변하는 것은 p번째 위치의 값 뿐이다.

먼저 배열을 순회하며 아래의 값들을 계산 해두자.

pos[s] = 1..k까지 더한 합이 s가 되는 모든 지점들

length = 임의의 합 s에 대해 max(pos[s])-min(pos[s])를 모아둔 집합

이 경우 length에 저장된 수 중에서 가장 큰 수가 답이 된다.

D. 국제 소 줄서기 사진 콘테스트

subtask3

pos 값과 length 값을 전부 map, set 등의 참조 삽입 삭제가 O(logn)인 자료구조를 이용해 관리하면, 앞서 말한 것처럼 p번째 소와 p+1번째 소가 자리를 바꿀 때 변화가 생기는 지점만 계산해 줄 수 있다. 즉,

x = 자리를 바꾸기전 1..p번째까지의 합, y = 자리를 바꾼 후 1..p번째 까지의 합

이라고 할 경우 pos[x]에서 p를 제거하고, pos[y]에서 p를 추가해주면 pos 값을 올바르게 갱신해 줄 수 있다. length 집합 역시 마찬가지로 pos 배열 값을 수정할 때 같은 방식으로 갱신해줄 수 있다.

D. 국제 소 줄서기 사진 콘테스트

subtask3

이러한 갱신 과정은 매 자리 바꿈마다 앞서 언급한 map, set 등의 적절한 자료구조를 사용할 경우 O(logN)에 수행할 수 있고, 맨 처음 초기화 과정에서도 O(NlogN)의 시간이 필요하므로 전체 시간 복잡도 O((M+N)logN)에 문제를 해결할 수 있다.

E. 순간이동 발판

정답자 수 : 6명(8%)

가장 처음 푼 사람 : ainta, 43분

출제 및 풀이 : djm03178

E. 순간이동 발판

subtask1

모든 발판의 주기가 같으므로, 두 발판이 방문하는 모든 구역들은 항상 두 발판이 동시에 나타나거나 항상 따로따로 나타난다. 따라서 [발판 번호][시간 % 주기]를 통한 BFS가 가능하다.

E. 순간이동 발판

subtask2

크게 두 가지 솔루션으로 나뉜다. 첫째로, subtask1과 마찬가지로 BFS를 통해 풀 수 있다. 단, subtask2에서는 각 발판에 대해 한 바퀴를 도는 것만으로는 다른 발판으로 건너가는 것이 가능한지 여부를 알 수 없으니, [발판 번호][시간]으로 BFS를 해야 한다.

E. 순간이동 발판

subtask2

하지만 무작정 BFS를 계속 돌리면 영원히 끝에 도달이 불가능한 경우에도 BFS를 빠져나가지 못할 수 있다. 이 때는 답이 최대 몇까지 도달이 가능한지를 계산해 상한선을 정해서, 그 이상을 넘어가게 될 경우 답이 없는 것으로 간주할 수 있다.

만일 성의 출구까지 도달하는 것이 가능할 경우, 최대 N-1번 발판을 갈아타면 최단 시간에 도달할 수 있다. 한 발판에서 다른 발판으로 건너가는 것이 가능할 경우, 두 발판이 겹치는 주기는 두 발판의 주기의 최소공배수인 것을 쉽게 알 수 있으며, 이 값은 두 주기의 곱 이하이다.

E. 순간이동 발판

subtask2

따라서 주기가 최대 K인 발판 N개에 대해 성의 출구에 도달하는 시간은 NK^2보다 작음을 알 수 있으며, 대략 이 정도를 상한선으로 놓고 BFS를 하면 탈출이 가능한 경우와 불가능한 경우를 구분할 수 있다.

E. 순간이동 발판

subtask3

subtask2의 두 번째 방법은 정해에 매우 근접하므로 여기서 같이 설명한다. 앞에서 살펴보았듯이 성의 출구에 도달하는 시간은 최대 NK^2에 가까우므로, K가 최대 4000인 subtask3에서는 답이 거의 16억이 될 수 있기 때문에 초 단위로 BFS를 하는 것이 불가능하다. (출제자가 찾은 최대의 답은 1,583,607,900이다.)

E. 순간이동 발판

subtask3

생각해보면, 두 발판이 동시에 나타나지 않는 경우는 굳이 고려할 필요 없이, 각 발판에 처음 도착한 시점으로부터 다른 발판들로 갈아탈 수 있는 가장 빠른 시간만 구해도 문제를 해결할 수 있다. 즉, 각 발판을 하나의 정점으로 놓고, 두 발판이 같은 구역에 동시에 나타나는 시간들마다 두 발판 사이에 간선이 있는 것으로 보고, 각 발판에 가장 빠르게 도달하는 시간이 언제인지를 각각 구해주면 된다. 쉽게 말해서, 다익스트라로 풀 수 있다.

E. 순간이동 발판

subtask3

이제 각 발판에 처음 도착한 시간 이후로 다른 발판들과 가장 처음으로 만나는 시간이 언제인지를 각각 구해야 한다. 현재 발판에 처음 도착한 시간부터 시작하여, 한 바퀴를 돌면서 다른 발판과 접점이 생기는지 차례대로 확인하면 된다.

예를 들어 현재 발판이 A이고, A와 B가 좌표 X에 모두 나타난다고 하자. 현재 시간 이후로 A가 X에 처음 도착하는 시간이 T라고 하면, A와 B는 X에 [T, T + LCM((A의 주기), (B의 주기))] 에서 현재 시간 이후로 처음으로 동시에 나타나게 됨을 알 수 있다. 만일 이 시간 내에 동시에 나타나는 때가 없다면 A와 B는 절대로 만날 수 없는 것이므로 간선이 없다고 보면 된다.

E. 순간이동 발판

subtask3

subtask2에서는 주기의 범위가 작기 때문에 T초부터 T+LCM초까지를 1초 단위로 확인해도 통과할 수 있다. 하지만 여기까지 생각해 놓고 6점밖에 가져가지 못한다면 너무 아쉽다! 조금만 생각해보면, T초부터 A의 주기만큼씩을 더한 시간들만 고려해도 된다는 것을 알 수 있다. 그러면 A와 B가 만나는 경우가 생기는지를 B의 주기번만 확인하면 알 수 있다.

이러한 관계는 서로 다른 (A, B) 쌍의 수만큼 존재할 수 있으므로 최대 완전 그래프의 형태가 되고, 따라서 O(N^2*K) 시간에 답을 구할 수 있다.

E. 순간이동 발판

subtask3

더 빠르게 답을 찾고자 한다면 확장 유클리드 알고리즘을 이용하여 두 발판이 만나는 시기를 구할 수 있는 것으로 보인다. 이러한 방법이 될 것 같다고 생각은 했으나, 아쉽게도 출제진/검수진 누구도 정확한 방법은 알지 못했고, 너무 수학적인 내용이라 굳이 생각하지 않기로 결정했다. 정답을 받은 몇몇 코드를 통해 그게 가능하다는 것은 입증(?)되었다.

F. 유물 도둑

정답자 수 : 5명(6.6%)

가장 처음 푼 사람 : koosaga, 60분

출제 : smu201111192

풀이 : yukariko

F. 유물 도둑

subtask1

Q=0인 경우, 정확히 K시간에 도달할 수 있는 정점 집합을 구하는 문제로 바뀐다.
이는 각 정점에 도달 가능하기 위해 걸리는 홀수 최단거리 값과 짝수 최단거리 값을 각각 구해두면 쉽게 구할 수 있다.

어떤 정점에 도달한 시간 K가 홀수인 경우 홀수 최단거리로 도착한 다음 인접한 정점에서 왔다갔다를 반복하면 반드시 K 시간에 해당 정점에 도달 가능하고,
짝수인 경우도 마찬가지이기 때문이다.

따라서 이와 같은 방법으로 O(N+M) 시간에 답을 구할 수 있다.

F. 유물 도둑

subtask2

이제 Q가 0이 아닌 경우를 보자. Q개의 이동 불가능한 정점에 대해, 이를 시간 순으로 정렬하면 i번째와 i+1번째 사이의 시간은 모든 정점들이 방문 가능한 상태이다.

따라서, i번째 시점에서 도달가능한 정점 집합을 알고 있다면 i+1번째 시점에 도달가능한 정점 집합은 Q=0일 때의 풀이를 이용해 구할 수 있다.

이와 같은 방법을 이용하여 답을 구할 수 있으므로, 전체 시간복잡도 O(Q(N+M+logQ)) 시간에 답을 구할 수 있다.

G. 모든 결말을 보고 싶어

정답자 수 : 4명(5.3%)

가장 처음 푼 사람 : ainta, 88분

출제 : djm03178

풀이 : jwvg0425

G. 모든 결말을 보고 싶어

subtask1

DP를 이용해 문제를 해결할 수 있다. 우선 아래와 같은 식을 생각해보자.

DP(n,k) = 내가 쓸 수 있는 돈이 k원이고 현재 n번 장면에 있을 때 가능한 최소 시간

이렇게 정의할 경우, DP(n, k)는 아래와 같은 2가지 경우를 고려하면 답을 구할 수 있다.

1. 세이브 포인트를 n에 두는 경우
2. 세이브 포인트를 n에 두지 않는 경우

G. 모든 결말을 보고 싶어

subtask1

1. 세이브 포인트를 n에 두는 경우

이 경우, 일단 n의 모든 자식에 해당하는 결말은 볼 수 있다. 하지만 비용이 좀 더 남아서 n의 왼쪽 자식 혹은 오른쪽 자식을 커버할 수 있는 경우, 해당 자식에 세이브포인트를 더 두는게 이득일 수도 있다. 따라서 cost[i] = i번째 위치에 저장하는 비용, time[i] = i번째 위치에 저장했을 때 모든 결말을 보기 위해 필요한 시간, n의 왼쪽 오른쪽 자식을 각각 l,r 이라고 정의하면 식은 다음과 같이 된다.

G. 모든 결말을 보고 싶어

subtask1

1. 세이브 포인트를 n에 두는 경우

min(time[i], dp(l, k - cost[i]) + time[i] - time[l], dp(r, k-cost[i]) + time[i] - time[r])

세이브 포인트를 n에만 두고 l,r에는 안 두는 경우가 있고, 코스트가 남아서 l 혹은 r에 두는 경우가 있을 수 있다(l,r에 모두 두는 경우 n에는 둘 필요가 없으므로 이 경우는 고려하지 않아도 된다). 따라서 이 세 가지 경우 각각의 값을 고려해주면 n에 세이브포인트를 두는 경우의 답을 구할 수 있다.

G. 모든 결말을 보고 싶어

subtask1

2. 세이브 포인트를 n에 두지 않는 경우

n에 두지 않는 경우는 l과 r 각각의 서브트리의 결말을 모두 봐야하기 때문에, l 서브트리에 할당할 비용, r 서브트리의 할당할 비용을 나누는 모든 경우를 확인해주면 된다. 즉, 0~k까지의 모든 i에 대해

min(solve(i,l) + solve(k-i,r)) 을 계산해주면 된다.

G. 모든 결말을 보고 싶어

subtask1

따라서 전체 부분 문제가 O(NK)에, 각 부분 문제를 해결할 때 n에 세이브 포인트를 두지 않는 경우를 해결하기 위해 O(K)의 시간이 걸린다. 즉, 전체 시간복잡도 O(NK^2)에 문제를 해결할 수 있다.

G. 모든 결말을 보고 싶어

subtask2

시간 복잡도를 개선하기 위해, 트리에서 어떤 장면 X가 Y의 부모일 경우 항상 X에 저장하는 비용 Xc >= Yc라는 특성을 이용해보자.

이 특성으로 인해, 어떤 장면 X에 저장을 했다면 그 장면의 왼쪽 혹은 오른쪽 자식에 저장하는 경우는 고려하지 않아도 된다는 것을 증명할 수 있다.

G. 모든 결말을 보고 싶어

subtask2

X의 왼쪽 자식을 L, 오른쪽 자식을 R이라고 하고 각각에 저장하는 비용을 Xc, Lc, Rc라고 하자. 그러면 Xc >= Lc와 Xc >= Rc를 항상 만족한다. 이 때, 저장하는데 필요한 비용을 계산해보면,

1. X와 L에 각각 세이브 포인트를 두는 경우 : Xc + Lc의 비용이 필요하다. 하지만 Xc >= Rc이므로 Xc + Lc >= Lc + Rc이기 때문에 이 경우 항상 X에 둔 세이브 포인트를 R로 옮길 수 있고, 세이브 포인트가 결말 지점에 더 가까워졌으므로 항상 이 경우가 더 답이 작다.
2. X와 R에 각각 세이브 포인트를 두는 경우 : 마찬가지 논리에 의해 항상 L,R에 세이브포인트를 둘 수 있고 이 경우가 더 답이 작다.

G. 모든 결말을 보고 싶어

subtask2

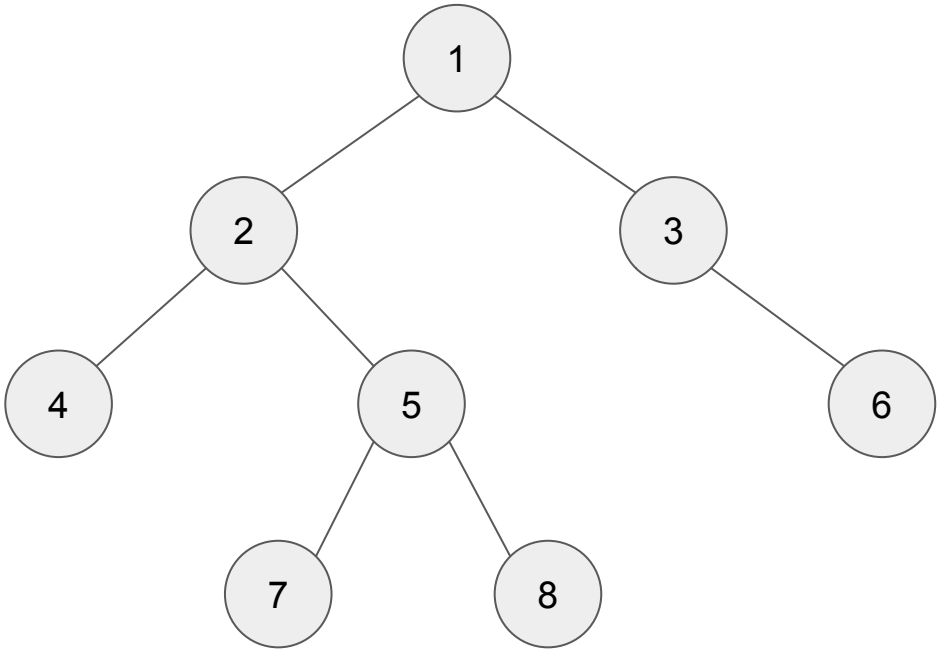
따라서, X에 세이브 포인트를 두었다면 X의 자식에는 세이브 포인트를 둘 이유가 없고, X에 두지 않았다면 반드시 그 노드의 두 자식 L,R의 모든 결말을 보기 위한 세이브 포인트를 각 서브 트리에 두어야만 한다.

이 성질을 활용하기 위해, DP의 정의를 바꿔보자.

DP(n,k) = n번째 노드를 기준으로, 내 자식이 아닌 내 왼쪽의 모든 결말을 볼 수 있을 때, 내 서브트리를 포함해서 내 오른쪽의 모든 결말을 보기 위해 필요한 최소 비용

G. 모든 결말을 보고 싶어

subtask2



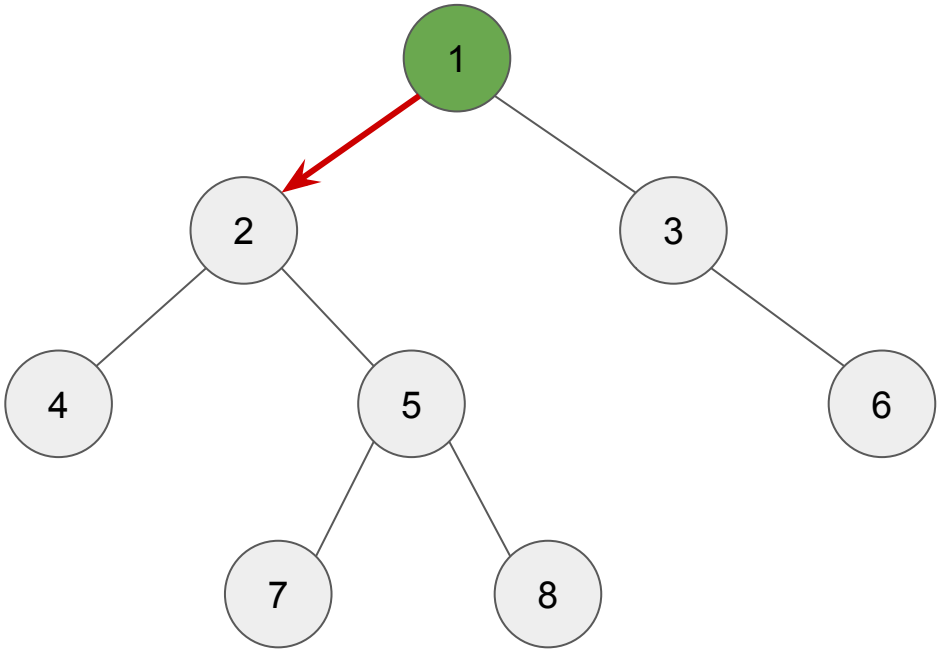
DP 정의가 복잡하니 왼쪽 트리를 예시로 살펴보자. 우선, 이 DP 정의에서 우리는

1. 내 왼쪽 자식으로 세이브를 미루거나
2. 현재 위치에 세이브하고 내 자식도 부모도 아닌 가장 가까운 오른쪽 노드로 이동하거나

둘 중 하나의 선택을 할 수 있다.

G. 모든 결말을 보고 싶어

subtask2



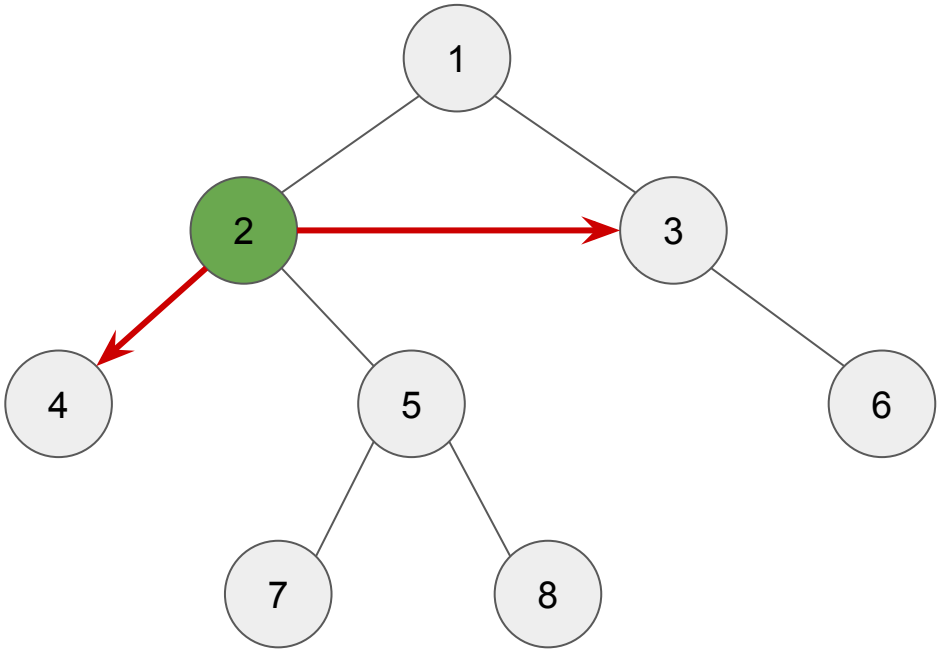
1번 노드에서 부터 시작해 보자.

여기서 세이브하는 경우 모든 결말에 도달 가능하다.

세이브하지 않는 경우, 앞서 말한 것 처럼 왼쪽 자식(2)으로 세이브를 미룰 수 있다.

G. 모든 결말을 보고 싶어

subtask2



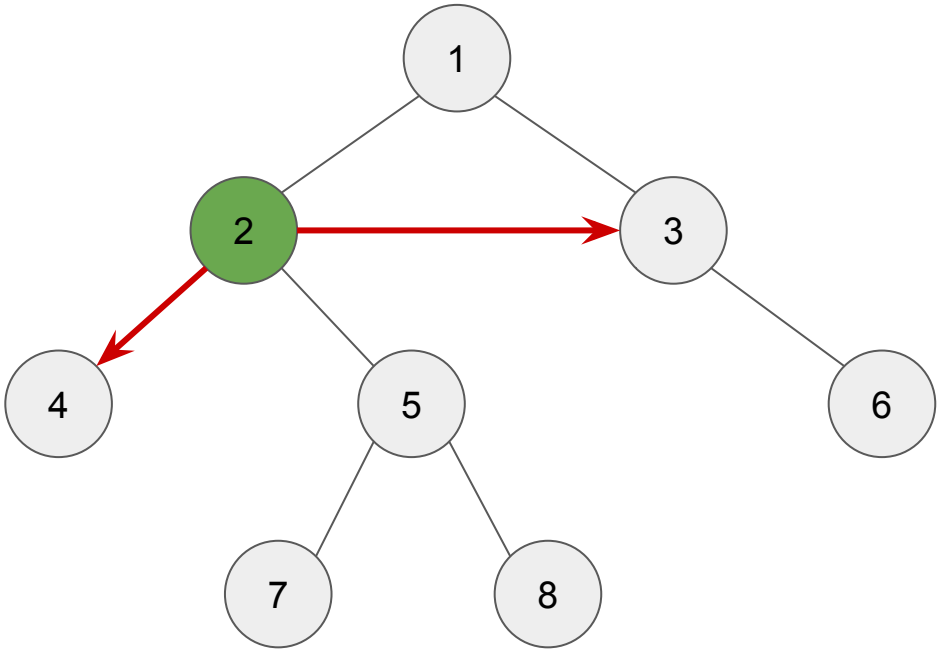
1에 세이브하지 않고 2로 세이브를 미뤘다고 생각해보자.
여기서 우리는,

1. 왼쪽 자식(4)로 세이브를 미루거나
2. 여기(2) 세이브하고 내 자식도 부모도 아닌 가장 가까운 오른쪽 노드(3)으로 이동하거나

둘 중 하나의 선택을 할 수 있다.

G. 모든 결말을 보고 싶어

subtask2

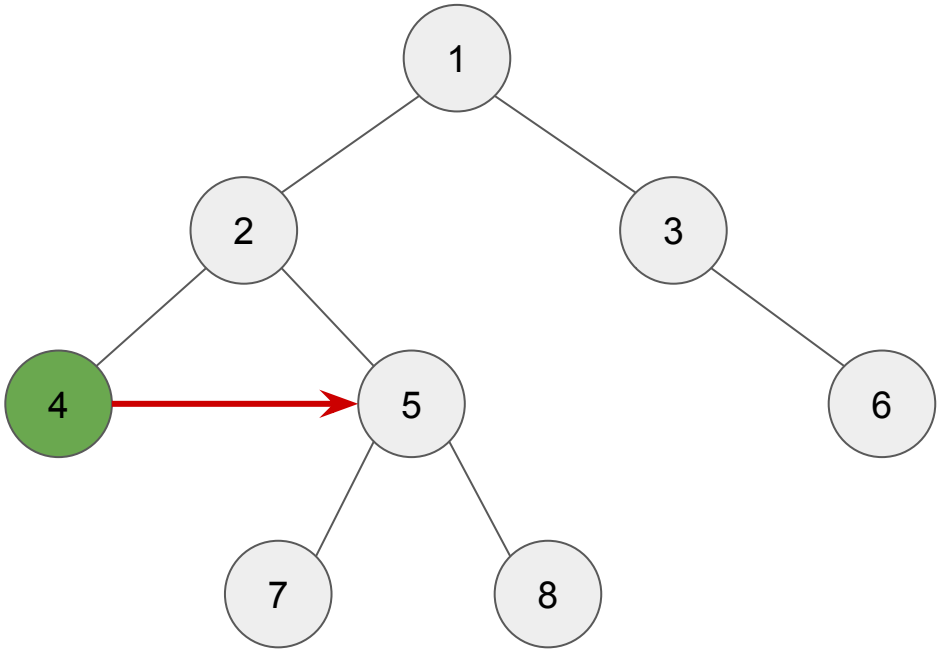


왜냐하면, 맨 처음의 증명으로 인해 만약 2에 세이브 포인트를 두었다면 그 자식(4,5,7,8)에는 세이브 포인트를 둘 이유가 없고,

또, 그 부모(1)에도 마찬 가지로 세이브 포인트를 둘 이유가 없기 때문이다. 따라서 이 지점들을 후보에서 제외하고 나면 다음에 봐야 하는 위치는 3이 된다.

G. 모든 결말을 보고 싶어

subtask2

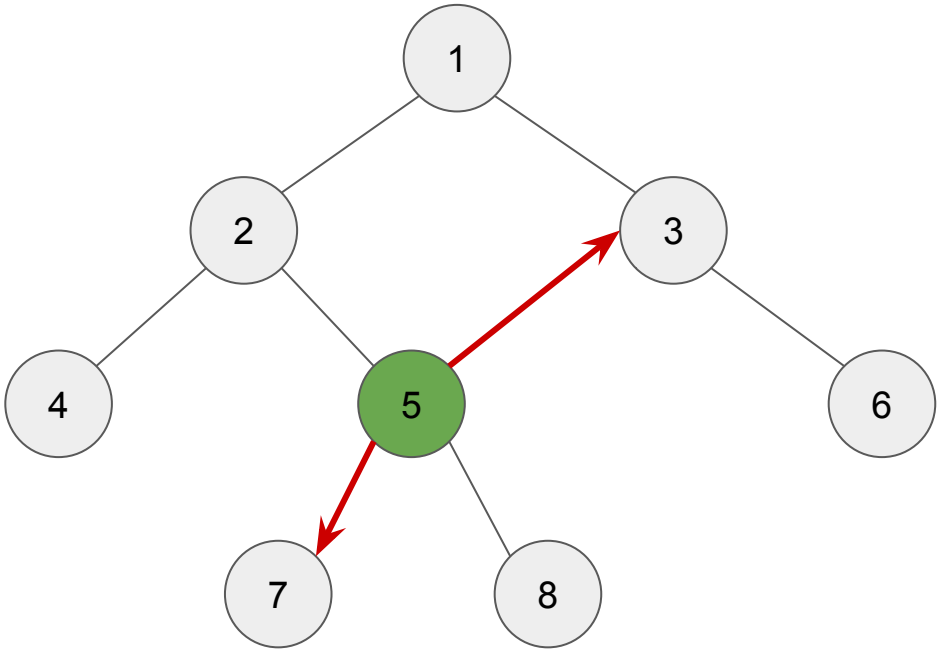


여기서 다시 2에 세이브하지 않고 4로 세이브를 미뤘다고 생각해 보자. 이 경우 여기서 더 이상 세이브를 미룰 수는 없다.

따라서 할 수 있는 선택은 4에 세이브하고 5로 이동하는 것 뿐이다.

G. 모든 결말을 보고 싶어

subtask2

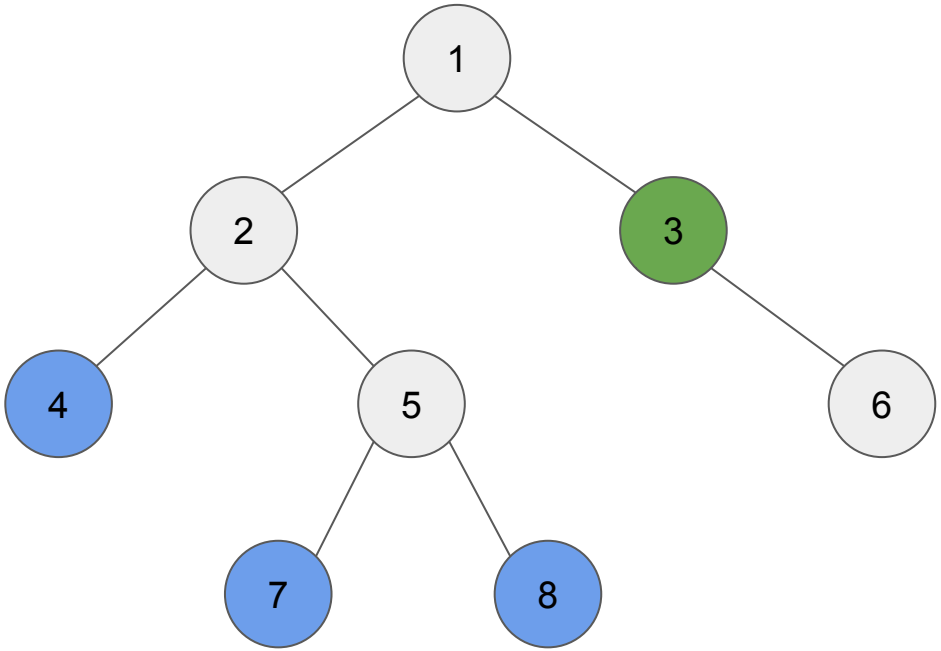


5로 이동한 경우를 생각해보자.
마찬가지 방식으로, 할 수 있는 선택은 두 가지다.

1. 왼쪽 자식(7)으로 세이브를 미룬다.
2. 이 위치에 저장하고 가장 가까운 부모도 자식도 아닌 오른쪽 노드(3)으로 이동한다.

G. 모든 결말을 보고 싶어

subtask2



3으로 이동해온 경우를 생각해보자.
이 경우는 DP의 정의에 의해 내 자식이 아닌 왼쪽의 결말은 모두 볼 수 있는 상황이어야 한다.

왼쪽으로 세이브를 미루는 경우 더 이상 미룰 수 없으면 세이브하고 오른쪽으로 이동하므로, 이 조건 역시 만족한다는 것을 알 수 있다.

G. 모든 결말을 보고 싶어

subtask2

따라서, 이와 같은 방식으로 DP를 정의할 경우 하나의 값이 더 필요해진다.

next[i] = i번째 노드의 자식도 부모도 아니면서 i에 가장 가까운 오른쪽 노드의 번호

라고 정의할 경우, n번째 노드의 왼쪽 자식을 l이라고 할때 DP(n,k)는 다음과 같이 정의할 수 있다.

DP(n,k) = min(DP(l,k), DP(next[n], k - cost[n])

G. 모든 결말을 보고 싶어

subtask2

이러한 DP 정의로 문제를 풀 경우, 마찬가지로 O(NK)개의 부분 문제가 존재하지만 각각의 부분 문제를 풀기 위해 필요한 시간복잡도가 O(1)로 줄어들게 된다. next[i] 배열을 계산하기 위해선 O(N)의 시간복잡도가 필요한 트리 탐색 한 번이면 충분하므로, 전체 시간 복잡도 O(NK)에 문제를 해결할 수 있게 된다.

출제자의 한 마디: 원래 이 문제는 subtask1에 해당하는 풀이를 생각하고 만들어졌으나, :god: jwvg0425님에 의해 마개조되어 이런 난이도가 되었습니다 :(

H. 마법 목걸이

정답자 수 : 10명(13.3%)

가장 처음 푼 사람 : ainta, 75분

출제 : pinch3773, cnhs2205

풀이 : jwvg0425

H. 마법 목걸이

subtask1

주어진 문제는 주어진 배열을 1칸씩 회전시킨 n개의 각 배열에 대해 해당 배열을 gcd가 1인 연속한 그룹들로 나누되 그 개수가 가장 많게 하는 문제로 생각할 수 있다.

이 경우 연속한 그룹의 gcd가 1이 되는 가능한 한 가장 빠른 위치에서 그룹을 나눠주는 방식으로 구성하면 그 경우가 가장 그룹의 개수가 많아지는 경우가 된다.

H. 마법 목걸이

subtask1

만약 이런 방식으로 구하지 않고 구한 답이 있다고 가정하자. 이 경우 gcd가 1이 되는 가능한 한 빠른 위치에서 자르지 않은 어떤 그룹에 대해, 그 그룹을 gcd가 1이 되는 가능한 한 빠른 위치에서 잘라주고, 나머지 값들을 그 뒤쪽 그룹에 포함시켜도 그룹의 개수는 줄어들지 않는다.

따라서 답이 되는 모든 경우를 앞서 말한 방식으로 구한 답으로 변환시켜 줄 수 있으므로, 이 경우가 최댓값이 됨을 알 수 있다.

H. 마법 목걸이

subtask1

매 배열에 대해 앞서 언급한 방식으로 답을 구하면 각각의 시작 위치 i에 대한 답을 O(NlogX) (X는 배열에서 등장할 수 있는 가장 큰 값) 시간에 구할 수 있다.

따라서, 전체 시간복잡도 O(N^2logX) 시간에 답을 구할 수 있다.

H. 마법 목걸이

subtask2

주어진 문제에서, 우리가 관심이 있는 것은 gcd가 1이 되는 가장 가까운 위치이다.

이를 near[i] = i번째 위치에서 연속한 부분 배열의 gcd가 1이 되는 가장 가까운 위치

로 정의할 경우, 모든 위치에 대해 near 배열의 값을 구해두면 각각의 선형 배열에 대해 O(NlogX)의 시간복잡도를 O(N)으로 개선시킬 수 있다.

이러한 near배열은 구간에 대한 gcd 값을 빠르게 구할 수 있게 해주는 세그먼트 트리 등의 자료구조를 적절히 활용하면 O(NlogNlogX) 시간에 계산 가능하고, 따라서 전체 시간 복잡도 O(NlogNlogX + N^2) 시간에 답을 구할 수 있다.

H. 마법 목걸이

subtask3

near 배열을 조금 더 나아가서,

near[i][x] = i번째 위치에서 gcd가 1인 그룹 개수가 2^x개가 되는 가장 가까운 위치

로 정의하면, binary lifting을 이용하여 각각의 시작 위치에 대해 O(logN) 시간에 답을 구할 수 있게 된다. 따라서 이 경우 전체 시간복잡도 O(NlogNlogX) 시간에 답을 구할 수 있다.

H. 마법 목걸이

subtask3

또다른 방법으로, union-find와 유사한 방법을 이용해 near 배열을 빠르게 건너뛸 수 있다. near 배열을 이용해 순회하는 과정에서 아래의 두 값을 초기화 해준다

warp[i] = 0 ~ i+n까지 위치에 대해 gcd가 1이 되는 그룹이 가장 많은 위치

count[i] = i ~ warp[i]까지 위치 사이에 존재하는 gcd가 1이 되는 그룹의 개수

이 두가지 값은 near값을 이용해 탐색한 후 방문한 좌표들에 대해 갱신해줄 수 있고, 이미 warp / count 값을 알고 있는 위치에서는 해당하는 좌표로 한 번에 이동할 수 있다.

H. 마법 목걸이

subtask3

이렇게 이동한 후 내가 이동하는 과정에서 방문한 좌표에 대해서만 전부 가장 오른쪽 위치로 위치 갱신 및 개수를 갱신해주면 이는 union-find의 경로 압축과 동일한 효과를 가지게 된다. 따라서 union-find의 시간 복잡도를 O(1)로 생각하면 모든 좌표에 대해 탐색해서 최댓값을 찾는데 걸리는 시간이 대략 O(N)에 비례하는 것으로 볼 수 있다.

따라서 이 방법 역시도 전체 시간복잡도 O(NlogNlogX)에 문제를 해결할 수 있다.

I. 팀 빌딩

정답자 수 : 8명(10.6%)

가장 처음 푼 사람 : tncks0121, 88분

출제 : jwvg0425

I. 팀 빌딩

subtask1

매 쿼리에 대해 N명의 팀원들을 모두 순회하면서 merge의 경우 서로 다른 팀들을 모두 같은 팀으로, split의 경우 나머지 조건에 맞춰 서로 다른 팀으로, query의 경우 주어진 팀원이 속한 팀의 사람 수를 계산해서 출력할 수 있다. 이 경우 각 쿼리마다 O(N)의 시간이 걸리므로, 전체 시간 복잡도는 O(QN)이 된다.

I. 팀 빌딩

subtask2

주어진 연산에서 split을 제외한 나머지는 union-find로 쉽게 구현할 수 있다. 이제 split의 구현이 문제인데, P=2인 경우 split을 수행하면 항상 주어진 집합이 짝수 집합과 홀수 집합으로 나뉘어 진다는 것을 알 수 있다.

따라서, merge 과정에서 짝수 집합과 홀수 집합을 merge할 때 실제로 합치지 않고 나랑 합쳐진 반대편 집합이 무엇인지만 추적하면 split은 단순히 나랑 합쳐진 집합 번호를 제거해주는(반대편 집합으로의 연결을 끊어주는) 식으로 쉽게 구현할 수 있다. 이 경우 union-find의 시간 복잡도를 O(1)이라고 가정하면 O(N+Q)로 풀 수 있다.

I. 팀 빌딩

subtask3

팀원 X를 P개의 노드로 분할해서 생각해보자. 이 때 분할된 각각의 Xi는 P로 나눈 나머지가 i인 집합을 의미한다. 이 때, 실제로 X mod P = k라면 Xk만 사이즈가 1인 집합이고, 나머지는 사이즈가 0인 집합으로 생각한다.

이렇게 하나의 숫자를 P개로 분할하면 N명의 팀원을 총 P*N개의 노드로 구분해서 처리하는게 된다. 이 때, 실제 X가 포함된 집합은 Xk가 포함된 집합일 것이다.

I. 팀 빌딩

subtask3

이제 문제에서 주어진 세 가지 쿼리를 이렇게 각 팀원을 P개로 나눈 전체 P*N개 팀원에 대해서 구현해보자.

1. split(분할) 쿼리
2. merge(병합) 쿼리
3. find(탐색) 쿼리

I. 팀 빌딩

subtask3

1. split(분할) 쿼리

핵심인 split 쿼리부터 생각해보자. 이 쿼리를 구현할 때, 매 split 연산마다 새로운 가상의 팀원(나머지에 따라 P개로 분할 된) T를 추가하면 상대적으로 편해진다. 어떤 팀원 X에 대해서 P로 나눈 나머지가 k에 대응되는 노드를 Xk라고 하자. split의 경우, 새로운 가상의 팀원 T(모든 사이즈가 0인)를 만든 다음, Xk와 Tk만 merge해주고 나머진 내버려 두면 나머지가 k인 노드들만 다른 나머지와 부모가 달라진다. 즉, k가 아닌 i에 대해 find(Xi) != find(Xk)가 된다.

I. 팀 빌딩

subtask3

2. merge(병합) 쿼리

병합의 경우, 주어진 X,Y에 대해 split으로 분리되지 않은 집합만 모아서 합쳐줘야 한다. 이 때 X % P = k라면 실제로 X가 속한 집합은 find(Xk)일 것이다. 따라서 find(Xk)와 부모가 동일한 팀원 숫자를 지닌 나머지에 대해서만 병합을 해주고, 나머지는 병합하지 않으면 merge 쿼리 역시 구현할 수 있다.

I. 팀 빌딩

subtask3

3. find(탐색) 쿼리

이 쿼리도 마찬가지로, 어떤 팀원 X에 대해 X % P = k인 경우 다른 나머지 값에 대해 Xk와 부모가 같은 팀원에 속한 경우에 대해서만 크기를 계산해서 합쳐주면 된다.

따라서, union-find 연산의 시간 복잡도를 O(1)이라고 가정할 경우 모든 연산에 대해 O(P)의 시간이 필요하므로, 전체 시간복잡도 O((N+Q)P)에 문제를 해결할 수 있다.