


이 문제를 풀기 위해 각 시각에 대해 위치 고수·위치 변경 두 가지로 나눠서 배열을 생성하거나 1번 나무·2번 나무 두 가지로 나눠서 배열을 생성하는 경우 올바른 점화식을 세우기 어렵다.

시각	0	1	2	3	4	5	6
자두가 떨어지는 나무	2	1	1	2	2	1	1
위치 고수시 받는 자두 개수의 최대치	0	1	2	2	2	3	4
위치 변경시 받는 자두 개수의 최대치	1	2	N/A	N/A	N/A	N/A	N/A

위치 고수·위치 변경 두 가지로 나누는 경우 1번 나무만 고수하는 경우 / 1번 2번 나무를 번갈아 받는 경우, 이 두 경우만 고려하게 된다.

시각	0	1	2	3	4	5	6
자두가 떨어지는 나무	2	1	1	2	2	1	1
1번 나무 선택시 받는 자두의 최대치, 변경 횟수	0, 0	2, 2	2, 2	2, 2	2, 2	3, 2	4, 2
2번 나무 선택시 받는 자두의 최대치, 변경 횟수	1, 1	1, 1	2, 2	3, 2	4, 2	4, 2	4, 2

시각	0	1	2	3	4	5	6
자두가 떨어지는 나무	2	1	1	2	2	1	1
1번 나무 선택시 받는 자두의 최대치, 변경 횟수	0, 0	2, 2	2, 2	2, 2	 이미 두 번 위치를 변경했으므로, 더 이상 위치를 변경할 수 없다.		
2번 나무 선택시 받는 자두의 최대치, 변경 횟수	1, 1	1, 1	2, 2				

1번 나무·2번 나무 두 가지로 나누는 경우 빠른 시각에 위치를 변경하는 안을 선택하게 되어서, (더 늦게 위치를 변경해야 선택할 수 있는) 최적의 안을 고려하지 못하게 된다.

그렇기 때문에 고려해야 하는 것은 시각과 변경 횟수 두 개의 축으로 이루어진 이차원 배열을 세우는 것이다. 이 경우에 자두가 선택할 수 있는 모든 선택지를 고려하여 최적의 안을 선별할 수 있다. (처음이 아닌 나중에 위치를 변경하는 안도 충분히 고려할 수 있다.)

시각	0	1	2	3	4	5	6
자두가 떨어지는 나무	2	1	1	2	2	1	1
0번 위치 변경 시 자두 개수, 현재 위치	0, 1	1, 1	2, 1	2, 1	2, 1	3, 1	4, 1
1번 위치 변경 시 자두 개수, 현재 위치	1, 2	1, 2	1, 2	3, 2	4, 2	4, 2	4, 2
2번 위치 변경 시 자두 개수, 현재 위치	N/A	2, 1	3, 1	3, 1	3, 1	5, 1	6, 1

시각	0	1	2	3	4	5	6
자두가 떨어지는 나무	2	1	1	2	2	1	1
0번 위치 변경 시 자두 개수, 현재 위치	0, 1	1, 1					
1번 위치 변경 시 자두 개수, 현재 위치	1, 2						
2번 위치 변경 시 자두 개수, 현재 위치	N/A						

바로 직전의 위치를 고수하는 경우
 바로 직전의 위치를 변경하는 경우
 두 화살표가 겹치는 곳은
 두 결과값 중 최적을 취한다.

위의 표를 채워나가는 방법

0번 위치를 변경한 경우의 칸은 맨 처음 자두가 있었던 1번 나무를 고수하는 것이므로 다음과 같이 점화식을 세울 수 있다.

```

struct 노드 {
    int 자두 개수;
    int 현재 위치;
};

노드 makeNode(int 자두 개수, int 현재 위치) {
    노드* result = new 노드();
    result->자두 개수 = 자두 개수;
    result->현재 위치 = 현재 위치;
    return result;
}

int 변경 후 위치(int 현재 위치) {
    return 3-현재 위치;
}

노드 max(노드 node0, 노드 node1) {
    return node0.자두 개수>node1.자두 개수 ? node0 : node1;
}
  
```

```
int 자두가 떨어지는 나무[1000];
노드 dpCache[1000][31];
```

와 같은 구조체 및 함수를 사용할 경우 0번 위치 변경 칸을 채우는 점화식:

```
for (int t=1; t<총 시간; t++)
```

```
dpCache[t][0] = makeNode(dpCache[t-1][0].자두 개수 + (자두가 떨어지는 나무[t]==1), 1);
```

비교 연산자는 산술 연산자보다 우선순위가 밀리므로 비교의 결과를 산술에 적용하려면 비교 연산을 괄호로 포장해 주어야 한다.

```
dpCache[t][0] = makeNode(dpCache[t-1][0].자두 개수 + (자두가 떨어지는 나무[t]==1), 1);
```

바로 직전까지 얻은
자두 개수

1번 나무를 고수하고 있으므로
이번에 1번 나무에서 자두가 떨어지는 경우
이번에 자두를 하나 얻게 된다.

시각	0	1	2	3	4	5	6
자두가 떨어지는 나무	2	1	1	2	2	1	1
0번 위치 변경 시 자두 개수, 현재 위치	0, 1	1, 1					
1번 위치 변경 시 자두 개수, 현재 위치	1, 2						
2번 위치 변경 시 자두 개수, 현재 위치	N/A						

바로 직전의 위치를 고수하는 경우 (Blue arrows)

바로 직전의 위치를 변경하는 경우 (Red arrows)

두 화살표가 겹치는 곳은 두 결과값 중 최적을 취한다. (Green circles)

0번 위치를 변경한 경우를 제외한 칸은 두 경우를 고려하여 그 중 최적을 구해야 한다. 점화식은 다음과 같이 세울 수 있다. (시각과 위치 변경이라는 두 개의 축이 있으므로 2중 반복문을 사용하게 된다.)

```
for (int t=1; t<총 시간; t++) {
    for (int c=1; c<=min(t+1, 30); c++) { //시각 t에 대해서 t+1 번을 초과하여 위치를 변경할 수 없다.
        //c번 위치를 변경하기 위해서는
        //바로 직전에 c-1번 위치를 변경한 상태에서 위치를 변경하거나- (node0)
        노드 node0 = makeNode(
            dpCache[t-1][c-1].자두 개수 + (자두가 떨어지는 나무[t]==변경 후 위치(dpCache[t-1][c-1].현재 위치)),
            변경 후 위치(dpCache[t-1][c-1].현재 위치)
        );
        //바로 직전에 c번 위치를 변경한 상태에서 위치를 고수해야 한다. (node1)
        노드 node1 = makeNode(
            dpCache[t-1][c].자두 개수 + (자두가 떨어지는 나무[t]==dpCache[t-1][c].현재 위치),
            dpCache[t-1][c].현재 위치
        );
        dpCache[t][c] = max(node0, node1); //node0과 node1 중 자두 개수가 더 많은 것을 취한다.
    } //c loop
} //t loop
```

이렇게 dpCache를 채웠다면 맨 오른쪽 칸 중 하나에서 최적의 해를 찾을 수 있다. 실제 문제에서는 자두가 위치를 변경할 수 있는 횟수의 상한이 주어져 있다. 따라서 dpCache의 오른쪽 변을 조회할 때 그 제한 조건을 지켜야 한다.

노드 optResult = dpCache[총 시간-1][0];

for (int c=1; c<=min(min(총 시간, 30), 위치 변경 횟수 상한); c++) {

 //총 시간이 T일 때 자두의 위치 변경은 T-1 번이 아닌 T 번 할 수 있다. 따라서 총 시간-1이 아닌 총 시간이

 //수식에 들어가야 한다.

 //위치 변경 횟수 상한 값 역시 적용되어야 함에 주의한다.

 optResult = max(optResult, dpCache[총 시간-1][c]);

}

std::cout << optResult.자두 개수;