과제 #3 벽돌 담장- 201921438 조현태

1) 소스코드

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// 함수 선언
long long Start_Search();
long long Reverse_Search();

int n;
vector<int> blocks;
// 함수를 돌면서 반복횟수를 저장할 총 횟수
long long total_count = 0;

// 메인 함수
int main(void)
{
	cin >> n;
	for (int i = 0; i < n; i++)
	{
		int a;
		cin >> a;
		blocks.push_back(a);
	}

	// 함수 실행 후 출력.
	cout << Start_Search() << endl;

	return 0;
}

// 정방향 탐색 함수
long long Start_Search()
{
	for (int i = 0; i < n - 2; i++)
		// i번째 값 < i+1번째 값일때
		if (blocks[i] < blocks[i + 1])
		{
			// i+1번째 값이 i번째 값이 되도록 하는 값
```

```
                        int gap = blocks[i + 1] - blocks[i];
                        // i+2번째 값 - gap >= 0 일 경우
                        if (gap <= blocks[i + 2])
                        {
                                // gap을 total_count에 적용함.
                                total_count += gap;
                                // 그 값을 i+1번째, i+2번째 값에 빼준다.
                                blocks[i + 1] -= gap;
                                blocks[i + 2] -= gap;
                        }
                        // i+2번째 값 - gap < 0 일 경우
                        else
                        {
                                // gap으로 빼면 i+2의 값이 음수가 되기때문에 i+2번째 값
이 0이 되는 값까지만 빼준다.
                                total_count += blocks[i + 2];
                                blocks[i + 1] -= blocks[i + 2];
                                blocks[i + 2] -= blocks[i + 2];
                        }
                }

        // 전체 탐색
        for (int i = 0; i < n - 1; i++)
                // 모든 값이 같지 않을 경우
                if (blocks[i] != blocks[i + 1])
                {
                        // 역방향 탐색 함수 실행
                        return Reverse_Search();
                        break;
                }
        // 모든 값이 같아서 위의 for문이 무사히 종료되었다면 총 횟수 반환.
        return total_count;
}

// 역방향 탐색 함수 -> 정방향 탐색과 같은 방법으로 역순으로 실행.
long long Reverse_Search()
{
        for (int i = n - 1; i > 1; i--)
                if (blocks[i] < blocks[i - 1])
                {
                        int gap = blocks[i - 1] - blocks[i];
                        if (gap <= blocks[i - 2])
                        {
```

```
                        total_count += gap;
                        blocks[i - 1] -= gap;
                        blocks[i - 2] -= gap;
                }
                else
                {
                        total_count += blocks[i - 2];
                        blocks[i - 1] -= blocks[i - 2];
                        blocks[i - 2] -= blocks[i - 2];
                }
        }

    for (int i = 0; i < n - 1; i++)
        // 모든 값이 같지 않으면
        if (blocks[i] != blocks[i + 1])
        {
                // 문제에서 제시한대로 -1 출력.
                return -1;
                break;
        }
    // 모든 값이 같다면 총 횟수 반환.
    return total_count;
}
```

2) 문제 설명

이 문제는 n개의 배열 위에 k개의 벽돌(담장의 높이)이 쌓여있을 때,
인접한 두 벽돌을 하나씩 제거해서 모두 같은 높이가 될 때의 횟수를 구하는 문제입니다.

벽돌 담장의 높이를 일정하게 맞추기 위해서는 낮은 담장을 높일 수가 없으니
가장 높은 담장을 탐색 후 그 담장의 좌우를 비교해 그 중 더 큰 담장의 높이를 찾아
2개의 담장을 하나씩 낮추면서 횟수를 세는 방법을 사용하려 했습니다.
하지만 **담장의 높이**가 10^9까지여서 **하나씩 빼는 방법**으로는 시간이 오래 걸리고
매번 최댓값을 탐색해야 하므로 **시간 복잡도 매우 높게 나온다**는 것을 알게 되었습니다.

따라서 높은 담장을 찾아서 낮추는 방식이 아니라
정방향으로 **i의 값과 i + 1의 값이 같도록하는 x를 구해서 i + 1과 i + 2의 값을 빼고**
시작할 때 맨 처음 값을 기준으로 맞추었으므로 이후 역방향으로 i의 값과 i - 1의 값이
같도록하는 x를 구해서 i - 1과 i - 2의 값을 빼는 방법을 생각했습니다.

예를 들어 **arr = {5 10 8}**이라면 첫 번째 값이 5이고 i + 1의 값이 10이므로
10 - x1 = 5, **x1 = 5,** x1의 값이 i + 1, i + 2의 값보다 작기 때문에
10 - x1와 8 - x1를 합니다. 그러면 **arr = {5 5 3}**로 정방향 계산이 끝나게 됩니다.
이후 역방향으로 **끝 값**, 즉 **가장 작은 값인** 3을 기준이로 5 - x2 = 3, **x2 = 2**
x2보다 i - 1과 i - 2의 값이 더 크므로 5 - x2와 5 - x2를 하면
arr = {3 3 3}으로 담장의 높이가 모두 같아지게 됩니다.
이때, 변수를 따로 만들어서 **모든 x의 값을 더하면** 그 변수가 **총 횟수**입니다.
위의 예시에서는 x1 + x2 = 5 + 2 = 7, 총 횟수는 7번입니다.

안되는 예를 보여드리면 arr = {5 5 3 8 4}의 경우로 계산하면
정방향 계산
i = 0일 때, arr = {5 5 3 8 4} -> {5 5 3 8 4}
i = 1일 때, arr = {5 5 3 8 4} -> {5 5 3 8 4}
i = 2일 때, arr = {5 5 3 8 4} - > ~~arr = {5 5 3 3 -1}~~ => arr = {5 5 3 4 0}
역방향 계산
i = 4일 때, arr = {5 5 3 4 0} -> {5 5 0 1 0}
i = 3일 때, arr = {5 5 0 1 0} -> {5 5 0 1 0}
i = 2일 때, arr = {5 5 0 1 0} -> {0 0 0 1 0}
∴ 모든 값이 같은 값이 아니므로 -1을 출력한다.

정리하자면 **정방향**으로 위의 과정을 실행하여 **i번째 값을 기준으로 i - 1과 i - 2의 값을**
내리면서 진행하기 때문에 결과는 **내림차순**이고 **역방향**으로 위의 과정을 실행하면
마지막 값이 가장 작기 때문에 모든 값이 그 값을 기준으로 같아지게 됩니다.
산 모양에서 **내림차순 계단 모양**으로 깎이고 이후 **평평한 모양**으로 바뀐다고 볼 수 있습니다.

먼저 **blocks**와 **total_count, n**을 **전역변수**로 설정했습니다.
이후 정방향 탐색 함수인 **Start_Search**는 total_count를 반환하는 함수로
그 값이 int의 범위를 벗어날 가능성이 있으므로 **64비트 정수형인 long long**을 사용했습니다.

```
long long Start_Search();
long long Reverse_Search();
```

우선 첫 번째 조건은 내림차순형태로 만드는 것이 목표이므로 **blocks[i] < blocks[i+1]일 때,**
gap = blocks[i + 1] - blocks[i] <= blocks[i+2]이면
blocks[i+1]과 blocks[i+2]에서 gap을 빼도록 했습니다.
만약 **gap > blocks[i+2]**이면 위의 계산을 통해 blocks[i+2]가 **음수**가 되므로
gap = blocks[i + 1] - blocks[i] 대신 **blocks[i + 2]**를 빼도록 해서 **0**이 되도록 만들었습니다.
값을 빼는 연산을 할 때마다 total_count에 더합니다.

```
for (int i = 0; i < n - 2; i++)
                if (blocks[i] < blocks[i + 1])
                {
                        int gap = blocks[i + 1] - blocks[i];
                        if (gap <= blocks[i + 2])
                        {
                                total_count += gap;
                                blocks[i + 1] -= gap;
                                blocks[i + 2] -= gap;
                        }
                        else
                        {
                                total_count += blocks[i + 2];
                                blocks[i + 1] -= blocks[i + 2];
                                blocks[i + 2] -= blocks[i + 2];
                        }
                }
```

이후, 벽돌 담장을 모두 탐색해서 **모든 값이 같으면 total_count를 반환**하고
같지 않다면 내림차순의 형태일 것이므로 역방향 탐색 함수인
Reverse_Search의 값을 반환하게 했습니다.

```
for (int i = 0; i < n - 1; i++)
        if (blocks[i] != blocks[i + 1])
        {
                return Reverse_Search();
                break;
        }
return total_count;
```

Reverse_Search는 Start_Search와 같은 방식으로 구성했으며,
시작점만 처음 값에서 끝 값으로 설정하여 계산했습니다.

```
for (int i = n - 1; i > 1; i--)
        if (blocks[i] < blocks[i - 1])
        {
                int gap = blocks[i - 1] - blocks[i];
                if (gap <= blocks[i - 2])
                {
                        total_count += gap;
                        blocks[i - 1] -= gap;
                        blocks[i - 2] -= gap;
                }
                else
                {
                        total_count += blocks[i - 2];
                        blocks[i - 1] -= blocks[i - 2];
                        blocks[i - 2] -= blocks[i - 2];
                }
        }
```

2개의 함수를 모두 통과하고 나면, 계산이 가능한 배열이라면 모든 값이 같아지게 됩니다.
따라서 **모든 값이 같으면 total_count**를, **같지 않다면 -1**을 출력합니다.

```
for (int i = 0; i < n - 1; i++)
        if (blocks[i] != blocks[i + 1])
        {
                return -1;
                break;
        }
return total_count;
```

마지막으로 메인함수에서 Start_Search함수를 실행하면 위의 과정을 통과해서
계산이 가능하다면 total_count를, 아니라면 -1을 출력하게 됩니다.

```
cout << Start_Search() << endl;
return 0;
```