**Equi**

**The problem description is very short:**

The equilibrium index of a sequence is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. For example, in a sequence A:

A[0]=-7 A[1]=1 A[2]=5 A[3]=2 A[4]=-4 A[5]=3 A[6]=0

3 is an equilibrium index, because:

A[0]+A[1]+A[2]=A[4]+A[5]+A[6]

6 is also an equilibrium index, because:

A[0]+A[1]+A[2]+A[3]+A[4]+A[5]=0

(The sum of zero elements is zero) 7 is not an equilibrium index - because it is not a valid index of sequence A. If you still have doubts, here is a precise definition: The integer k is an equilibrium index of a sequence A[0],A[1]..,A[n-1] if and only if *0<= k* and *sum(A[0..(k-1)])=sum(A[(k+1)..(n-1)])*. Assume the sum of zero elements is equal to zero.

Write a function

```
int equi(int A[], int n)
```

that, given a sequence, returns its equilibrium index (any) or -1 if no equilibrium index exists. Assume that the sequence may be very long.
The problem can be solved by using various approaches, the most common being simply to follow the equilibrium definition:

```
int equi ( int A[], int n ) {
   int k, m, lsum, rsum;
   for(k = 0; k < n; ++k) {
      lsum = 0; rsum = 0;
      for(m = 0; m < k; ++m) lsum += A[m];
      for(m = k + 1; m < n; ++m) rsum += A[m];
      if (lsum == rsum) return k;
   }
   return -1;
}
```

Unfortunately, this approach has two disadvantages:

- it fails on large input data sets, since the time complexity is $O(n^2)$
- it fails on large input values (for example if the input array contains values like MIN/MAX_INT) due to the arithmetic overflows

The solution analysis will detect such problems in submitted code:

**Analysis**

Detected time complexity:
**O(n^2)**

| test | time | result |
|---|---|---|
| example<br>Test from the task description | 0.001 s. | **OK** |
| extreme_empty<br>Empty array | 0.012 s. | **OK** |
| extreme_first | 0.012 s. | **OK** |
| extreme_large_numbers<br>Sequence with extremly large numbers testing arithmetic overflow. | 0.008 s. | **WRONG ANSWER**<br>got 2, but it is not equilibrium point, sum[0..1]=4294967294, sum[3..3]=-2 |
| extreme_last | 0.004 s. | **OK** |
| extreme_single_zero | 0.001 s. | **OK** |
| extreme_sum_0<br>sequence with sum=0 | 0.001 s. | **OK** |
| simple | 0.001 s. | **OK** |
| single_non_zero | 0.001 s. | **OK** |
| combinations_of_two<br>multiple runs, all combinations of {-1,0,1}^2 | 0.001 s. | **OK** |
| combinations_of_three<br>multiple runs, all combinations of {-1,0,1}^3 | 0.004 s. | **OK** |
| small_pyramid | 0.004 s. | **OK** |
| large_long_sequence_of_ones | 1.004 s. | **TIMEOUT ERROR**<br>running time: >1.00 sec., time limit: 0.10 sec. |
| large_long_sequence_of_minus_ones | 1.004 s. | **TIMEOUT ERROR**<br>running time: >1.00 sec., time limit: 0.10 sec. |
| medium_pyramid | 0.704 s. | **TIMEOUT ERROR**<br>running time: >0.70 sec., time limit: 0.10 sec. |
| large_pyramid<br>Large performance test, O(n^2) solutions should fail. | 1.008 s. | **TIMEOUT ERROR**<br>running time: >1.01 sec., time limit: 0.16 sec. |

We can fix the first problem with a better algorithm, and the second problem with a better data-type (for example, using long long type instead of int for sum computations). The key observation for better running time is to update the left/right sums in constant time instead of recomputing them from the scratch.

```
int equi(int arr[], int n) {
   if (n==0) return -1;
   long long sum = 0;
   int i;
   for(i=0;i<n;i++) sum+=(long long) arr[i];

   long long sum_left = 0;
   for(i=0;i<n;i++) {
      long long sum_right = sum - sum_left - (long long) arr[i];
      if (sum_left == sum_right) return i;
      sum_left += (long long) arr[i];
   }
   return -1;
}
```

Using this solution, you can obtain a perfect score:

**Analysis**

**Detected time complexity:**
**O(n)**

| test | time | result |
|---|---|---|
| example<br>Test from the task description | 0.001 s. | **OK** |
| extreme_empty<br>Empty array | 0.004 s. | **OK** |
| extreme_first | 0.001 s. | **OK** |
| extreme_large_numbers<br>Sequence with extremly large numbers testing arithmetic overflow. | 0.001 s. | **OK** |
| extreme_last | 0.001 s. | **OK** |
| extreme_single_zero | 0.001 s. | **OK** |
| extreme_sum_0<br>sequence with sum=0 | 0.001 s. | **OK** |
| simple | 0.004 s. | **OK** |
| single_non_zero | 0.004 s. | **OK** |
| combinations_of_two<br>multiple runs, all combinations of {-1,0,1}^2 | 0.001 s. | **OK** |
| combinations_of_three<br>multiple runs, all combinations of {-1,0,1}^3 | 0.004 s. | **OK** |
| small_pyramid | 0.004 s. | **OK** |
| large_long_sequence_of_ones | 0.012 s. | **OK** |
| large_long_sequence_of_minus_ones | 0.020 s. | **OK** |
| medium_pyramid | 0.012 s. | **OK** |
| large_pyramid<br>Large performance test, O(n^2) solutions should fail. | 0.048 s. | **OK** |