Injurious Company

First of all, note that the `H` and `V` moves act totally independently of each another. Therefore, we can solve the x and y coordinates separately and then combine our answers. For the rest of the editorial, we only consider the 1D case. We'll give some remarks on implementation towards the end.

We are given a sequence $K_1, K_2, K_3, \ldots, K_n$. A value x is called *constructible* if there exists,

$$\pm k_1 \pm k_2 \pm k_3 \pm \cdots \pm k_n = x,$$

where $1 \leq k_i \leq K_i$ for all i, and each one can be (independently) added or subtracted from the sum. Given some x, is it constructible?

We will make the following claim, and then prove it constructively (so the details of the proof can be almost-directly adapted into code and submitted as our solution). Let $S = \sum K_i$.

- It is necessary that $-S \leq x \leq S$, otherwise it is not constructible (this part is hopefully clear).
- If there exists at least one index t such that $K_t \geq 2$, then all integers in this range are constructible.
- Otherwise, if $K_i = 1$ for all i (and so $S = n$), then *only* the integers with the same parity (even or odd) as n are constructible.

The outline of our proof is as follows:

- We first show that all integers from n to S are constructible; by flipping all the signs, all integers from $-S$ to $-n$ are also constructible.
- We then determine which integers from $-(n-1)$ to $(n-1)$ are constructible.

n to S

So, let's start by showing that all values from n to S are constructible.

- Begin with the configuration $+1 + 1 + 1 \cdots + 1 = n$
- Then, repeat the following step as many times as possible:
  - Select a non-maximal k_i (i.e. one whose value is not yet K_i) and increment its value by $+1$.
- This process terminates on $K_1 + K_2 + K_3 + \cdots + K_n = S$, when all terms have been "saturated"
- Since the value of the expression increases by $+1$ at each step, we have shown that all integer values from n to S are attainable since we "passed by" all of them along the way[1].

By flipping all the signs from `+` to `-`, this tells us that $-S$ to $-n$ are also all constructible as well.

[1] This is a sort of "discrete intermediate value theorem."

$-(n-1)$ to $(n-1)$

What about the values from $-(n-1)$ to $n-1$? Let's split into cases.

All elements are 1

Consider the expression:

$$\pm 1 \pm 1 \pm 1 \cdots \pm 1 = x,$$

where there are n ones. By choosing each $\pm$ sign correctly, we can satisfy this equation **if and only if** $-n \leq x \leq n$ and x has the same parity as n.

Proof: The proof uses a similar "discrete IVT" technique as earlier.

- Begin with the configuration $-1-1-1-\cdots-1 = -n$.
- Then, repeat the following step as many times as possible:
  - Select a 1 whose sign is - and flips its sign to +
- This process terminates on $+1+1+1+\cdots+1 = n$, i.e. when we have selected + for all terms.
- Since the value of the expression increases by $+2$ at each step, we have shown that all values from $-n$ to n with the same parity as n are attainable, since we "passed by" all of them along the way.

However, also using a parity argument, any numbers in this range with a *different* parity from n cannot be constructed—changing any of the $\pm$ signs in the expression will not change the parity of the sum, so it is impossible for us to attain any parity different from the one we started with (with is that of n).

There exists a > 1 element

If we have at least one K_t such that $K_t \geq 2$, then **all** integers from $-(n-1)$ to $n-1$ are constructible. Just use the construction from earlier, slightly tweaked:

- Suppose there exists an index t such that $K_t \geq 2$
- Set $k_i = 1$ for all $i \neq t$.
- Using the construction in the previous case, choose the ± 1s appropriately in order to construct a value "neat x", and then use k_t to exactly attain x. More precisely:
  - If $-(n-3) \leq x \leq n-1$...
    - $\oplus$ If $x-1$ has the same parity as $n-1$, then construct that, and then add $k_t = 1$
    - $\oplus$ Else, $x-2$ has the same parity as $n-1$, so construct that, and then add $k_t = 2$
  - Else, $x = -(n-1)$ or $x = -(n-2)$.
    - $\oplus$ If $x = -(n-1)$, construct $-(n-3)$ using the ones, and then *subtract* $k_t = 2$.
    - $\oplus$ Else, $x = -(n-2)$. Similarly, construct $-(n-3)$ using the ones, but then subtract $k_t = 1$.

Implementation Notes

I will end on some remarks for how I coded the model solution for this problem. Implementation choices and styles may differ, but I am reasonably happy with the "neatness" of my approach.

This section will probably make more sense once you've actually skimmed through my source code.

First of all, I made a function `construct(x: int, moves: list[Move])` that solves the 1D case of the problem. Then, I filtered the `H` and `V` moves into separate lists, and then solved each case independently.

More precisely, I implemented a class `Move` that contains each move's K_i and dir_i information, **and** also contains the k_i and chosen sign for each move (I convert from sign to `NSEW` only at the end). What the `construct` function does is *mutate* the `Move`s given to it, i.e. the information about the chosen k_i and sign is attached to the `Move` itself.

This is nice because the `Move` objects in the filtered lists passed to the `construct` function are references to the *same* objects in the original list. So, when I must output my construction, it's easy for me to still do so in the correct order, since I just refer to the original version of the list (which still has all the `Move`s in the correct order).

Finally, you'll notice that I use a lot of helper classes—in particular, a fondness for `Enum` over naked string literals. You don't have to do this in comp prog, but I do it in work for a reason (it helps prevent errors from stupid typos). It also prevents natural places to put methods like "flip this sign" or "convert a sign to a compass direction".