Folding Cloth

Editorial

Task Author: Hanks Chong

---

1. Problem recap

We are given a strip of cloth made of N unit cells labelled $0, 1, ..., N-1$ from left to right. The cloth occupies a number of **columns**; each column is a vertical **stack** of one or more cells. Initially there are N columns, the i-th holding only cell i.

A **folding step** takes a contiguous block of x columns from the left end or the right end, rotates the whole block by 180°, and lays it **above** or **under** the remaining columns, aligned at the appropriate end. After placing, any vertical gap that appears inside a column is closed by pressing. There are four step types:

- `LA x` — take the leftmost x columns, place them **above** the rest;
- `LU x` — take the leftmost x columns, place them **under** the rest;
- `RA x` — take the rightmost x columns, place them **above** the rest;
- `RU x` — take the rightmost x columns, place them **under** the rest.

The 180° rotation is a genuine rotation: it reverses the left-to-right order of the columns in the block **and** the top-to-bottom order of the cells inside each of those columns.

We are given M steps as arrays T (the type, $0, 1, 2, 3$ for `LA` / `LU` / `RA` / `RU`) and X (the length x). The steps are guaranteed to end with all N cells in a single column. We must output that column, top to bottom. Writing L_i for the number of columns after i steps, the limits are $1 \le M < N \le 500,000$ and $1 \le X_i < L_i$.

The intended solution runs in $O(N)$ time. Below we build up to it through the subtasks.

2. Subtask 1 — direct simulation ($N \le 2000$)

Simulate the process literally. Keep one stack of cell labels per column (a `vector<int>` per column, or a single array of stacks). Each step rotates the chosen block — reversing the column order and reversing each individual stack — and concatenates each rotated column onto the column it lands on.

A single step touches $O(N)$ cells and there are $O(N)$ steps, so the running time is $O(N^2)$, comfortable for $N \le 2000$. This also makes a convenient reference brute force for stress-testing the faster solutions.

```
// One LA/LU step over the leftmost x columns (block lives in [L, L+x)).
for (int j = 0; j < x; j++) {
    // src is folded onto dst, the column it lands on after the rotation
    int src = L + j;
    int dst = L + 2 * x - 1 - j;
    // rotate the folded column 180 degrees, then stack it onto column dst:
    // above for an LA step, under for an LU step
    reverse(stk[src].begin(), stk[src].end());
```

```
    if (type == LA)
        stk[dst].insert(stk[dst].begin(), stk[src]...);
    else
        stk[dst].insert(stk[dst].end(), stk[src]...);
}
L += x;
```

The snippet above only covers the **non-overflowing** case $2x \le L_i$, where the folded block lands entirely within the existing columns. A long fold, $2x > L_i$, overhangs the far end and spills into **new** columns there, so `dst = L + 2x - 1 - j` would run past the current rightmost column. Subtask 1 has no short-fold guarantee, so it does contain such folds. The simplest way to cover them — and what the reference brute force does — is to first apply the long-fold rewrite of Subtask 8. That rewrite is more than a local change of `x` and `t`: it also maintains a global `reversed` flag, reinterprets every later instruction accordingly, and reverses the final answer once at the end (see Subtask 8 for the details). With it in place, every fold satisfies $2x \le L_i$ and the loop above applies unchanged.

3. Subtasks 2–5 — the case $X_i = 1$

When every fold has $x = 1$ the cloth keeps a very simple shape, and we can maintain the answer with deques instead of moving cells around.

Subtask 2 (`LA` only)

With only `LA 1` folds, the configuration is always: the leftmost column holds the prefix of cells $0, 1, ..., a$ (as a set — their vertical order is scrambled by the repeated rotations, e.g. two folds give $[1, 0, 2]$, not $[0, 1, 2]$), and every other column holds a single cell. Each `LA 1` rotates that growing left column 180° and lays it on top of the next single cell (column $L + 1$), merging them into the new left column. Concretely, store the order of the stacked column in a deque; in those terms each fold:

- reverses the current stack (the 180° rotation), then
- appends the newly absorbed single cell.

Reversing then appending at the back is the same as **alternating** between pushing the new cell to the front and to the back of the deque, so we never pay for the reversal: keep a boolean "is the deque logically reversed" and push to the matching end. This is $O(1)$ per step, $O(N)$ overall.

Since the final order depends only on N, one may instead derive a closed-form pattern and print it directly.

Subtask 3 (`LA` / `LU`)

Same shape as above, but whether the new cell is glued to the **top** or the **bottom** of the stack now depends on the type: `LA` places the block above, `LU` below. With the "logically reversed" flag this is still a single front/back push per step.

Subtask 4 (`LA` / `RA`)

Now folds come from both ends, so the configuration is: a stacked column on the left holding a prefix of cells $0, ..., a$, a stacked column on the right holding a suffix $b, ..., N - 1$ (again each only as a set, in scrambled vertical order), and single cells in between. Maintain **two** deques, one per end, growing them inward. When the two ends become adjacent ($L + 1 = R$), the final fold joins them; concatenate the two deques (one of them reversed) in the order dictated by the last step.

Subtask 5 (all four types, $X_i = 1$)

Combine the two previous ideas: two deques, each with its own reversed flag, and the front/back choice determined by the fold type. Merge them when the two ends meet. Everything stays $O(N)$.

4. Subtasks 6–7 — short folds ($2X_i \le L_i$)

For larger x the deque trick breaks down, but a cleaner structural idea takes over. We describe two equivalent $O(N)$ solutions.

The adjacency invariant

> **Lemma.** If at any moment a column holds cells $A_0, A_1, ..., A_{K-1}$ from top to bottom, then for every $0 \le i \le K-2$ the cells A_i and A_{i+1} are **adjacent in the final ordering**.

This holds because folding never separates two cells that are currently stacked on top of each other: a rotation may flip a column upside down, and pressing only removes empty gaps, but vertically-touching cells stay touching forever. Hence the final single column is a path that visits every cell once, and the whole problem reduces to recovering that path.

Solution 1 — adjacency graph

Build an undirected graph on the N cells. We only ever need, for each currently active column, its **top** cell and its **bottom** cell — the interior of a stack is already pinned down by earlier edges.

When two columns are merged by a step, the rotation lines them up so that:

- for `LA` / `RA` (placed **above**) the two columns meet **top to top**, so we add an edge between the two top cells;
- for `LU` / `RU` (placed **under**) they meet **bottom to bottom**, so we add an edge between the two bottom cells.

After adding the edge we update the merged column's exposed top/bottom endpoints. Keeping the two exposed cells of each column in `pos[c]` makes every merge $O(1)$. In the reference code `pos[c].second` is the **top** cell and `pos[c].first` the **bottom** cell, so an **above** fold joins the two `.second` endpoints:

```
// LA: for each folded column j, find the column k it lands on, join tops.
for (int j = L; j < L + x; j++) {
    int k = 2 * L + 2 * x - j - 1;        // mirror position after rotation
    createEdge(pos[j].second, pos[k].second);
    pos[k].second = pos[j].first;         // new exposed endpoint of column k
}
L += x;
```

When all steps are done the graph is a simple path. Walk it from one endpoint of the final column (the cell with degree 1, i.e. the current `top` of the surviving column) to the other, emitting cells in order. That walk is the answer.

Solution 2 — tree of positions

A second view builds a rooted tree whose nodes are the **spatial column positions**. Whenever a column A is stacked onto a column B, make B the parent of A and label the child **type-1** if A went **above** and **type-2** if it went **under**. Children are kept in chronological order.

Let S_A be the cell order of A's subtree, written bottom to top, once all folds touching the subtree are done. If the type-1 children in chronological order are $B_0, ..., B_{K-1}$ and the type-2 children are $C_0, ..., C_{P-1}$, then S_A is the concatenation

$$\text{reverse}\left(S_{C_{P-1}}\right), ..., \text{reverse}\left(S_{C_0}\right),\ A,\quad \text{reverse}\left(S_{B_0}\right), ..., \text{reverse}\left(S_{B_{K-1}}\right).$$

A single DFS recovers the answer directly in top-to-bottom order. The nested reversals are handled by a parity bit. The root is visited at even parity; at a node whose depth has the **same** parity as the root, recurse into its **type-1** (above) children in reverse chronological order, emit the node, then recurse into its **type-2** (below) children in chronological order. At nodes of the **opposite** parity, swap the two groups: type-2 children in reverse chronological order, the node, then type-1 children in chronological order. (Reading the code below, the `else` branch is the same-parity-as-root case.)

```
void dfs(int u, int parity) {
    if (parity & 1) {
        for (int i = (int)child2[u].size() - 1; i >= 0; --i)
            dfs(child2[u][i], parity ^ 1);
        ans.push_back(u);
        for (int v : child1[u])
            dfs(v, parity ^ 1);
    } else {
        for (int i = (int)child1[u].size() - 1; i >= 0; --i)
            dfs(child1[u][i], parity ^ 1);
        ans.push_back(u);
        for (int v : child2[u])
            dfs(v, parity ^ 1);
    }
}
```

The tree can degenerate into a chain of depth $\Theta(N)$ — for instance when every fold uses $x = 1$ — so a naive recursive DFS overflows the default 8 MiB stack. Either enlarge the stack explicitly or convert the DFS to an explicit-stack loop.

Why this is $O(N)$

If $2X_i \le L_i$ then the x folded columns each land on a **distinct** remaining column without sticking out past the far end, so $L_{i+1} = L_i - X_i$. Summing over all steps, from $L_0 = N$ down to $L_M = 1$,

$$X_0 + X_1 + ... + X_{M-1} = N - 1.$$

The total number of column merges (edges, or tree links) is therefore $N - 1$, and both solutions run in $O(N)$ time and memory.

5. Subtask 8 — long folds (no extra constraints)

The only thing left is a step with $2X_i > L_i$: a fold that grabs **more than half** of the columns. Such a fold can wrap past the opposite end, so $L_{i+1} \neq L_i - X_i$ and the clean accounting above fails.

The fix is a symmetry observation:

> Folding x columns over from one side (with $2x > L$) creates **exactly the same column merges** as the short fold of length $L - x$ from the **opposite** side applied to the **current**

cloth (note $2(L-x) < L$). The two differ only by a global mirror of the cloth (left↔right and top↔bottom) — which we do **not** perform now, but record in a flag and undo at the very end.

So we keep a global boolean `reversed` meaning "the cloth is currently stored mirrored on both axes". For each incoming step:

1. if `reversed`, the user's instruction must be reinterpreted in mirrored storage, which flips both the side bit and the above/under bit: `t ^= 3`;
2. if the (possibly reinterpreted) fold is long, $2x > L$, rewrite it to the equivalent short fold from the opposite side — `t ^= 2`, `x = L - x` — and toggle `reversed`.

After the rewrite every fold satisfies $2x \le L$, so the subtask-7 machinery applies verbatim. Mirroring never changes **which** pairs of columns get merged, so the graph/tree is built correctly; we only need to undo the global mirror at the very end, reversing the produced ordering when `reversed` is set.

```
int t = T[i], x = X[i];
if (reversed) t ^= 3;          // reinterpret in mirrored storage
int len = R - L + 1;
if (x * 2 > len) {             // long fold -> equivalent short fold
    reversed ^= 1;
    t ^= 2;
    x = len - x;
}
// ... now 2x <= len: run the short-fold merge for type t ...
```

This completes a clean $O(N)$ solution for the full problem.

6. Reference implementation

The model solution combines the adjacency graph (Solution 1) with the long-fold rewrite. It runs in $O(N)$ time and uses $O(N)$ memory.

```
#include <algorithm>
#include <utility>
#include <vector>
using namespace std;

static const int MAXN = 500005;
static vector<int> adj[MAXN];

static inline void createEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

vector<int> fold(int N, vector<int> T, vector<int> X) {
    for (int i = 0; i < N; i++) adj[i].clear();

    vector<pair<int, int>> pos(N);     // pos[c] = (one endpoint, other endpoint)
    for (int i = 0; i < N; i++) pos[i] = {i, i};
```

```
    int L = 0, R = N - 1, reversed = 0;
    int M = (int)T.size();
    for (int i = 0; i < M; i++) {
        int t = T[i], x = X[i];
        if (reversed) t ^= 3;
        int len = R - L + 1;
        if (x * 2 > len) { reversed ^= 1; t ^= 2; x = len - x; }

        if (t == 0) {                   // LA: join tops, fold from the left
            for (int j = L; j < L + x; j++) {
                int k = L * 2 + x * 2 - j - 1;
                createEdge(pos[j].second, pos[k].second);
                pos[k].second = pos[j].first;
            }
            L += x;
        } else if (t == 1) {            // LU: join bottoms, fold from the left
            for (int j = L; j < L + x; j++) {
                int k = L * 2 + x * 2 - j - 1;
                createEdge(pos[j].first, pos[k].first);
                pos[k].first = pos[j].second;
            }
            L += x;
        } else if (t == 2) {            // RA: join tops, fold from the right
            for (int j = R; j > R - x; j--) {
                int k = R * 2 - x * 2 - j + 1;
                createEdge(pos[j].second, pos[k].second);
                pos[k].second = pos[j].first;
            }
            R -= x;
        } else {                        // RU: join bottoms, fold from the right
            for (int j = R; j > R - x; j--) {
                int k = R * 2 - x * 2 - j + 1;
                createEdge(pos[j].first, pos[k].first);
                pos[k].first = pos[j].second;
            }
            R -= x;
        }
    }

    // The graph is a path; walk it from one endpoint of the final column.
    vector<char> visited(N, 0);
    vector<int> ans;
    ans.reserve(N);
    int cur = pos[L].second;
    while (cur != pos[L].first) {
        ans.push_back(cur);
        visited[cur] = 1;
        cur = !visited[adj[cur][0]] ? adj[cur][0] : adj[cur][1];
    }
    ans.push_back(cur);
    if (reversed) reverse(ans.begin(), ans.end());
    return ans;
}
```

7. Summary of the progression

Subtasks	Complexity	Key idea
1 — small	$O(N^2)$	Simulate the folds directly.
2–5 — $X_i = 1$	$O(N)$	One or two deques with a "reversed" flag.
6–7 — $2X_i \le L_i$	$O(N)$	Adjacency invariant; recover a path via a graph or a tree.
8 — full	$O(N)$	Rewrite long folds as mirrored short folds with a global flag.