Pockets
Proposed Solution

Marek Żylak

1 Introduction

The solution `h.cpp` consists of two phases:

1. paper folding simulation
2. pockets counting

2 Paper folding simulation

First, we can notice that the operation of folding the left part of the figure to it's right side can be substituted by folding the right part of the figure to the left. We should just remember that position of the surface, on which the figure was lying, has changed to opposite.

We will use rectangular array to represent unit square fields on the surface. Each field contains a stack of unit square pieces of paper. The stack is represented as a double linked list. We also keep vertical an horizontal orientation of each piece of paper. Paper folding is equivalent to taking all pieces from a group of stacks and putting them on the other group of stacks.

The number of folds is $O(N)$ and during every fold we have to move $O(N^2)$ pieces, so the overall simulation cost is $O(N^3)$.

It's easy to find a fold which moves some of the pieces to previously unused fields on the surface. Moreover, it's possible to move some of the pieces $\Theta(N^2)$ units from it's original position. That makes allocating sufficiently big array of fields impractical, because it would need $\Theta(N^4)$ elements. There are at least two solutions to this inconvenience:

- Thanks to the observation described in the beginning of this document we can avoid operations that require allocation of previously unused fields. In this method we must always keep track of the surface position and orientation. Moreover, when the surface is on the visible side of the figure, we need to fold paper to the back side of the figure, when normally we always fold to the front side. Despite all the difficulties it's possible to implement the algorithm elegantly, using conditional code in symmetrical

cases. We only have to use parametrization and arrays heavily (for example: instead of using two variables `next` and `prev` to store pointers to neighbour of the element on the list, it's better to use two element array `link[2]`. Described method was used in the proposed solution.

- The folding result can be put in a temporary array of surface fields. Author of the proposed solution considers this method easier to implement correctly. He didn't choose it only because he found and implemented the other method first.

3 Pockets counting

To count the number of pockets related to a particular edge of the final unit square we have to find have to find how paper covers itself on the edge. We can have two situations on the edge:

- two unit squares, connected on the edge cover everything between them
- the edge of the unit square is also the edge of the paper

The edge which connects two unit squares covers every pocket lying between the unit squares. We can represent such covering by the sequence of brackets. We build it by processing the unit square on the last stack from bottom to top. When we encounter the first unit square involved in connection on the edge, we put an opening bracket into the sequence. After encountering the second unit square relater to the connection we put a closing bracket. A unit square edge that is also the edge of the paper can be represented as two consecutive brackets — opening and closing bracket. In the end we must analyze created string of brackets to find the number of places where two expressions, that are not part of any other expressions, meet. Such place represents space between two consecutive unit squares that is not blocked from outside, by some paper.

4 Summary

Overall complexity of the proposed algorithm is $O(N^3)$.