Report for Programming Problem 2 - ARChitecture

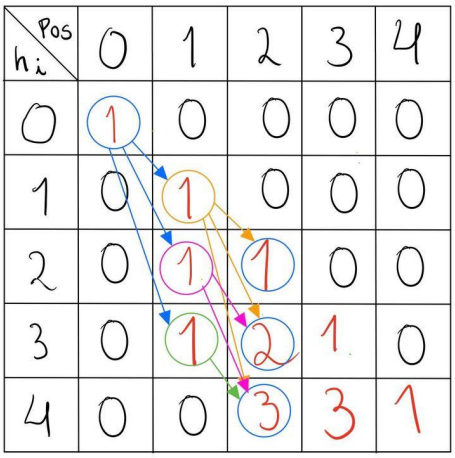
Team:

Student ID: 2018293728 Name: Miguel Rabuge
Student ID: 2018283166 Name: Pedro Rodrigues

1. Algorithm description

Note: The subsequent values are not calculated as shown in the pictures. We take advantage of values already calculated.

1. **Ascending Step**
   In the first part of the algorithm the ascending step is performed. It calculates all possible ascending combinations.



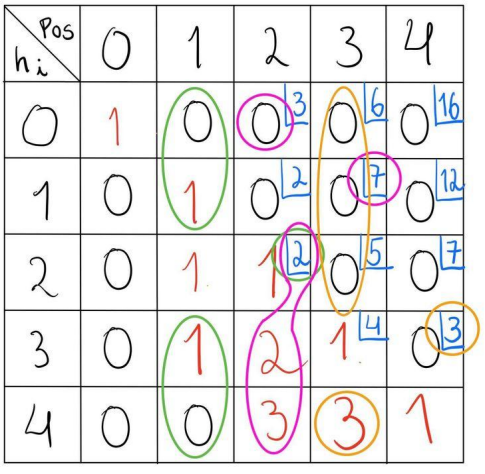
It starts with the The following picture helps clarifying the concept for n = 5, h = 4, H = 8: M[0][0][ascending] = 1.

The subsequent values are built on top of this one, following the shown logic.

It is possible to see that the ascending matrix corresponds to a lower triangular matrix and never more than that, has the minimum incrementing step is 1, corresponding to the diagonal.

2. **Descending Step**

   Once the ascending matrix has been built, it is time to calculate the descending matrix. The following picture helps clarifying the concept for n = 5, h = 4, H = 8:



An element (7) in the descending matrix corresponds to the sum of all the values that can monotonically ascend to it (0) plus the ones that have monotonically ascended to a point before the current and are monotonically descending since that. (2 + 2 + 3)

The values in **blue** are the ones != 0 on the descending matrix.

3. **Sub-Solutions Sum**
   The solution to this problem can be decomposed like shown below for a given n:

$$S(< n) = S(n) + S(< n - 1) =$$
$$= S(n) + S(n - 1) + S(< n - 2) =$$
$$= \dots =$$
$$= \sum_{i=3}^{n} S(i) \quad \Rightarrow \quad \sum_{j=2}^{n-1} M[0][j][descending]$$

   Hence, the solution (S(< n)) is the sum of the first row of the descending matrix, using the values of the columns j = 2, …, n - 1.

2. Data structures

In this problem, a 60000x500x2 integer hypermatrix is used. It is built through the use of nested std::array structures, i.e. std::array<std::array<std::array<int, 2>, 500>, 60000>. The 60000 and 500 values correspond to the maximum **H** and maximum **n** allowed by the problem description. The 2 depth dimensions are used as an “ascending” matrix and “descending” matrix, due to ascending/descending conflits, which prevents recalculations. Also, each element of the matrix corresponds to all possible valid combinations to have a block in a certain position (3,.., n), with a certain height (0,1,2, … , H - h), keeping in mind that the elements of the “ascending” matrix only store combinations that monotonically ascend, while the elements of the “descending” matrix store combinations that monotonically ascend to that point plus combinations that monotonically ascend to a point before the current and monotonically descended since that. This structure is cleared with std::fill in order to save memory space and time.

3. Correctness

1. **Rule 1 - $3 \leq k \leq n$**

   Solutions with n lower than 3 are not taken into account.

2. **Rule 2 - First and Last Blocks must be placed on the floor**

   The solutions are the sum of the first row of the descending matrix, which implies that the ending block is at height 0. Also, every possibility starts with the first block at height 0, hence the first and last blocks are placed on the floor.

3. **Rule 3 - A block must share at least 1 height with his neighbors**

   In the ascending step, the value is propagated at maximum to the height of h - 1 above, on the next column. This is due to the fact that, at maximum, 1 block has to share the height with the previous one. In the descending step, a given element is the sum of the h - 1 elements

"ascending" above and the "ascending"/"descending" h - 1 elements below, on the previous column, following the similar ascending logic.

4. **Rule 4 - Monotonically increase to a block and then monotonically decrease from that same block**

   The algorithm has 2 steps because of this rule. The ascending one guarantees us that all the resulting values are ascending-only into the ascending matrix. The next step, the descending one, will calculate all descending-only combinations starting from each and every resulting value of the ascending step, into the descending matrix.

4. Algorithm Analysis

We can observe that the solution that was elaborated for this problem follows a bottom-up strategy typically characterized for the use of a tabulation method and an iterative process being quite fast and as memory efficient as possible. When compared to the top-down approach, this one tends to be more conceptually harder to design but it avoids having a heavy impact in memory usage and temporal overhead introduced by the recursive nature of a top-down approach. Our solution being a bottom-up approach follows a tabulation structure instead of a memoization one. This mean solutions to the subproblems are stored in a tabular data structure which is accessed according to a given formula that was defined by us in order to solve both ascending and descending steps of the algorithm, therefore taking advantage of the previous solutions to the subproblems and avoiding unnecessary computations.

In terms of spatial complexity the algorithm requires a table in order to save the subproblem results. That table will be a hypermatrix of size "H" by "n" by "2", where H is the maximum height of the room n is the number of blocks and 2 is two value cell storing the results of ascending and descending steps. That being said the spatial complexity will be given by the following expression.

Spatial Complexity: $O\ (2\ *\ n\ *\ H),\ n\ \leq 500\ ,\ H \leq 60000$

In terms of temporal complexity the algorithm requires in a worst case scenario to loop through (H - h) * n elements in both the ascending and ascending steps. In reality it is not exactly value being this only an upper bound, but asymptotically we can say that it tends to this value.

Temporal Complexity: $O((H\ -\ h)\ *\ n)$

5. References

[1] cppreference.com
[2] T. H. Cormen, et a.l Introduction to Algorithms, 3rd ed., 2009.
[3] J. Erikson, Algorithms, 2019