Task 5: `Flooding`

Authored by: Yaw Chur Zhe

Prepared by: Yaw Chur Zhe

Editorial written by: Yaw Chur Zhe

In this editorial, let $s((r, c))$ be the set of cells that would be flooded if (r, c) were the initially flooded cell. In particular, $f((r, c)) = |s((r, c))|$.

Assume for convenience that the grid is surrounded by buildings (if not, pad the grid with buildings).

Subtask 1

Additional constraints: $h = 1$

It is easy to see that $s((1, c))$ forms a maximal range that includes $(1, c)$. By iterating through the grid from left to right and maintaining the length of the maximal range, it is possible to solve this subtask in linear time.

Time complexity: $\mathcal{O}(w)$

Subtask 2

Additional constraints: $h, w \leq 80$

It is possible to simulate the flooding process with a Breadth-First Search (BFS). To compute $f((r, c))$:

1. Push cell (r, c) into a queue.
2. Repeatedly, pop the front cell (a, b) from the queue, and if there exists a cell adjacent to (a, b) that becomes flooded. If so, add that cell to the queue.

Since this BFS runs in $\mathcal{O}(hw)$, performing this BFS for all empty cells (r, c) yields a $\mathcal{O}(h^2w^2)$ solution.

Time complexity: $\mathcal{O}(h^2w^2)$

Subtask 3

Additional constraints: $h, w \leq 500$

A crucial observation is the following:

Theorem 5.1. *For all (r, c), $s((r, c))$ forms a subrectangle of the grid.*

Proof. One of the many ways to prove this theorem is via a constructive algorithm:

1. Initially, let $s((r, c)) := \{(r, c)\}$.
2. Repeat the following steps:
    - If $s((r, c))$ is surrounded by buildings on all sides, then $s((r, c))$ cannot possibly expand further. Break out of this loop.
    - Otherwise, there exists an empty cell adjacent to $s((r, c))$, which becomes flooded. Assume without loss of generality that this cell is on the top edge of $s((r, c))$. Water flows into this cell, and then flows into the row of that cell. The overall effect is that $s((r, c))$ expands upwards by one row.

Since $s((r, c))$ is a subrectangle of the grid at all times in this algorithm, the theorem is proven. □

The constructive algorithm can be used to compute $f((r, c))$. With appropriate preprocessing, it is possible to perform the check in step 2 in $\mathcal{O}(1)$ time (using a prefix sum, for instance). In this manner, computing $f((r, c))$ can be done in $\mathcal{O}(h + w)$, since $s((r, c))$ can only expand h times up or down and w times left or right.

By repeating this for all empty cells, we obtain a $\mathcal{O}(hw(h + w))$ solution.

Time complexity: $\mathcal{O}(hw(h + w))$

Subtask 4

Additional constraints: $h, w \leq 2000$

From this point onwards, we require further observations concerning the structure of $s((r, c))$.

Lemma 5.2. *For all rows and columns of $s((r, c))$, there must exist at least one empty cell.*

Proof. Assume otherwise, i.e. there exists some row or column that consists completely of buildings. It is not possible for water to flow past this row or column, hence contradicting the initial assumption. □

Corollary 5.3. *$s((r, c))$ is surrounded completely by buildings.*

Proof. This follows directly from the constructive algorithm provided in the proof of Theorem 5.1. □

Call a subrectangle *good* if it satisfies the properties in Lemma 5.2 and Corollary 5.3 (even if the subrectangle is not equal to $s((r, c))$ for any (r, c)). It turns out that it is possible to generate all such good subrectangles quickly (**Phase 1**). Then, all that remains is to determine which good subrectangles correspond to which cells (**Phase 2**).

Phase 1

Fix the top row t of the good subrectangle. Let $i_1, i_2, \ldots, i_k$ be the indices of columns where (t, i_j) has a building. Consider a good subrectangle with left column $l = i_j + 1$ and right column $r = i_{j+1} - 1$ (if $l > r$, ignore the following). We can uniquely determine its bottom row b (if it exists) due to Lemma 5.2. Specifically, b is equal to the minimum row where cells $(b + 1, l), (b + 1, l + 1), \ldots, (b + 1, r)$ all have buildings, provided $b \geq t$. Computing b can be done efficiently with a binary search. You must additionally ensure that the top, left, and right borders of this good subrectangle are surrounded by buildings, as per Corollary 5.3.

Now, we have identified all good subrectangles with left and right columns adjacent in the sequence $i_1, i_2, \ldots, i_k$. Consider the index i_j where the number of contiguous buildings directly below (t, i_j) is minimised. In other words, the distance from (t, i_j) to the nearest empty cell below it is minimised. Observe that there are no other good rectangles where i_j is the left or right column (under the fixed top row t), due to Lemma 5.2. Hence, we can safely remove i_j and try to identify a good subrectangle with $l = i_{j-1}$ and $r = i_{j+1}$. By repeatedly removing the index with the minimum number of contiguous buildings directly below it, we can identify all good subrectangles for a fixed top row t. It is also obvious that $\mathcal{O}(w)$ good subrectangles can be identified in this manner, since $\mathcal{O}(w)$ good subrectangles are identified initially, and additional good subrectangles will only be identified if an element from $i_1, i_2, \ldots, i_k$ is removed.

By repeating the above process for all top rows t, we are able to identify all good subrectangles. Crucially (and surprisingly), there are $\mathcal{O}(hw)$ of these good subrectangles.

Phase 2

To match good subrectangles to their corresponding cells, we observe the following fact:

Lemma 5.4. *$s((r, c))$ is equal to the smallest good subrectangle containing (r, c).*[2]

Proof. Consider all the good subrectangles that contain cell (r, c). Let X be the intersection of all of these subrectangles (X cannot be empty since it must contain the cell (r, c) itself). One can verify that X is also a good subrectangle:

1. The intersection of many subrectangles that are surrounded by buildings will also yield a subrectangle surrounded by buildings.
2. It is not possible for X to contain a row or column filled with buildings, since it would imply the existence of a smaller good subrectangle that is a subrectangle of X (by "cutting" the rectangle along the filled row or column).

It is not possible for water to flow out of X if (r, c) were initially flooded (since X is surrounded by buildings). Hence $s((r, c)) \subseteq X$. However, since X is the intersection of all good subrectangles containing (r, c), $X \subseteq s((r, c))$. Hence $s((r, c)) = X$. Further observe that X is exactly equal to the smallest good subrectangle containing (r, c), hence proving the lemma. □

With the knowledge of Lemma 5.4, a straightforward way to complete the solution would be to use standard sweep-line techniques and a segment tree to simulate several 2D range-set operations. This yields a time complexity of $\mathcal{O}(hw \log h \log w)$, which is sufficient to pass this subtask.

Time complexity: $\mathcal{O}(hw \log h \log w)$

Subtask 5

Additional constraints: None

To obtain the full solution, **Phase 2** of the algorithm must be sped up. At least two distinct ways to achieve this have been identified.

Method 1

This method relies on the following observation:

Lemma 5.5. *When identifying good subrectangles in Phase 1 in **decreasing** t, the smallest good subrectangle containing (r, c) is equal to the first good subrectangle identified containing (r, c).*

[2] Note that there must exist at least one good subrectangle containing (r, c), since (r, c) is empty.

Proof. Omitted. □

In other words, we may perform **Phase 1** and **Phase 2** in parallel: once a good subrectangle is identified, we can immediately determine the answer for all cells in that subrectangle.

Sweep t from h to 1. Maintain a segment tree of size w indexed by columns, where the value of the segment tree at a particular column is equal to the minimum row of the cell in that column whose answer has not been determined yet, with the additional constraint that the row must be at least t. When a good subrectangle t, l, r, b has been identified, repeatedly do a range minimum query on columns l to r in this segment tree. If it yields a row $\leq b$, set the answer of that cell to $(b-t+1)(r-l+1)$ and delete it from the segment tree (then updating the value of the segment tree at that column). When transitioning from t to $t-1$, update all values in the segment tree to $t-1$. This is clearly amortised $\mathcal{O}(hw)$, since the answer for each cell can only be determined once.

Time complexity: $\mathcal{O}(hw(\log h + \log w))$

Method 2

The 2D range-set problem from **Phase 2** can be reframed as multiple independent 1D range-set problems (one for each row), which can then be solved with Union-Find Disjoint Sets. However, naively splitting each good subrectangle into its constituent rows may not be efficient enough. One possible optimisation is to "ignore" rows that have already been filled; maintain the maximum fully filled row from previously identified subrectangles, and only fill rows after that. There exists a proof showing that the number of 1D range-set operations generated by this process is $\mathcal{O}(hw)$.

Time complexity: $\mathcal{O}(hw(\log h + \log w))$

Author's Comments

I am aware of other solutions with unproven time complexities:

- Optimising the solution for Subtask 3 using a Union-Find Disjoint Set, eagerly "merging" subrectangles whenever possible. This solution appears to be empirically efficient, and is likely able to score 60 or 100 points with enough constant-time optimisation.
- Naively performing **Method 1**. In other words, instead of using a segment tree to determine if there exists a cell whose answer has yet to be determined, simply iterate from l to r one-by-one to clear all such cells. Such a solution comfortably scores 100 points. I

am not sure how to bound the complexity of this solution, but I conjecture that it is fast because the sum of widths (or heights) of good subrectangles is somehow bounded.

Proving that these solutions are efficient remains an open problem. I welcome further discussion on this topic.