CS577 HW2

09/24/2015

Keith Funkhouser

Haruki Yamaguchi

Input: of positive integers, where.

Output: Maximum value of, where and .

**Procedure**

**return**

Input: is the same as above, and .

Output: Maximum value of, where and .

**Procedure**

**If** **then**

**return**

//Assume that returns a larger one of .

**While**  **and**

**If**  **then**

**While and**

**Else If**  **then**

**While**  **and**

**Else**  //

**While and**

**While and**

**While**

**While**  **and**

**While**

**While**  **and**

**Return**

**Proof of correctness**

**Claim:** The procedure returns a maximum value of, where and

**The base case**: It is invoked when has only one element and returns a value of the element. This is correct since is just a value of the element here, and the maximum value is .

**Find a maximum value between and** : “The maximum value of a rectangle under pictogram” is the largest area of:

* The largest rectangle in left hand side
* The largest rectangle in right hand side
* The largest rectangle lying between left hand side and right hand side

The first two values are given by the recursion calls.

Then, the procedure must determine a value of all rectangles that lies between both left and right side.

The examination starts with two indexes of elements that are next to a border; i.e., .

If , move to the right side until it finds such that or reaches to the end of right side. If such is found, then the area of rectangle with height is since for . In addition, rectangles with such don’t need to be determined since those rectangles can’t extend to the left side (if ) or are identical to the one with (if ). If , the same method is done in the left side.

If , move both and toward the end of each side until they find a smaller element or hit the end and determine the area.

If the area of determined rectangle is larger than the current maximum value, assign it as a new current max.

If one side is exhausted ( or ), the first loop is terminated. The area calculation still gives a correct value even if either or both are out of boundary by 1. At this point, **at least one half side of each rectangle whose value exists in that side is determined.**

Then either second or third loop determines the rectangles in the remaining side if they still exist. Assuming that after the first loop, the procedure gets into the second loop (and later skips the third loop). In the loop, move to the left side until it finds such that or reaches to the end. If such is found, then the area of rectangle with height is since for , and again, rectangles with don’t need to be determined. If the area is larger than current max, update it. At the time loop is terminated, **the other side of each rectangle whose value exists in that side is determined.**

At the end, the area of each possible rectangle is determined, and the maximum value is returned.

**Termination**

**Recursive call:** The termination of a recursive part of can be proven in the same way as a merge sort. The boundary of the procedure is defined by and , and the bound is halved in each recursive call since it can be either or , where. At last, as a base case, the recursive calls terminate when .

Since and is assigned by integer division, will never become larger than . For example, let and , so that for integer division. Then the next recursive calls are and .

**Iteration**: There are three types of while loops in the procedure.

The first type has two loop conditions: and . Since either or both is decremented by 1 and/or is Incremented by 1 in each iteration and since , the loop eventually terminates.

The second type has and the third type has as its loop condition, but they follow the same rule as the first type, as its variable is decremented/incremented for each iteration.

**Proof of runtime**

Since each recursive call halves its boundary, the depth of recursive calls is . In each call, an array is iterated through, but each element is visited at most one time. Since the area of rectangle is calculated in a constant time, its runtime is , where is some constant. Therefore, the complexity of this algorithm is .

**Optimization**

The first comparison to find a rectangle between left and right sides (when and ) is redundant since it actually calculates a rectangle that lies only in one side, unless . Therefore, following lines can be added right before the first while loop.

**If**  **then**

**While and**

**Else If**  **then**

**While and**