CSC263: Assignment 1

Jingqi Zhang, Yulin WANG, Tianjing Ruan

Due Date: *05-26-2019*

Question	**Written By**	**Checked By**
1	Yulin WANG	Jingqi Zhang
2	Jingqi Zhang	Tianjing Ruan
3	Tianjing Ruan	Yulin WANG

Question 1

(a)

$T(n)$ is $O(n^2)$.
Justify: For every input, array A of size n that contains at least $n \geq 2$ arbitrary integers, the outer while loop (line $3-13$) takes at most $(n-1)$ iterations. And each such iteration can make the inner while loop (line $5-8$) execute at most $(n-2)$ times. Thus, the whole loop is executed at most $(n-1)(n-2)$ times, and since statements at all other lines execute constant time, so executing the procedure doStuff takes at most n^2 time. Then, $T(n)$ is $O(n^2)$.

(b)

$T(n)$ is $\Omega(n^2)$.
Justify: As a worst case, the outer while loop (line $3-13$) should not end early. So there exists an array A of size $n \geq 2$, which contains n 1's, namely $A[1..n] = \{1, 1, ..., 1, 1\}$. For this input array A, the total runtime of the loop (line $3-13$) is $1 + 2 + 3 + ... + (n-2) = \frac{[1+(n-2)](n-2)}{2} = \frac{(n-1)(n-2)}{2}$, and since statements at all other lines execute constant time, so there are at least proportional to n^2 steps, thus executing the procedure doStuff takes at least n^2 time. Then, $T(n)$ is $\Omega(n^2)$.

Question 2

(a)

The data structure I used is Max Heap.

My algorithm will be 3 parts:

A. Build-up an array S of ordered x position points of given rectangles.

Each element is a tuple of (s, h, e, ptToRectangle, ableToOutput) which:

- s: the x positions of given rectangles (including both left and right edges) in an increasing order (if duplicate s value occur, the order depends on l/r by the rules mentioned below)
- h: the height of these points in relative rectangles, (this h will be updated in part B as the exact height of required outline points).
- e: represent this point is either left edge x position (denote as L) or right edge (denote as R)
- ptToRectangle: a pointer points to the relative given rectangle.
- ableToOutput: represents if this point is our final output point, while set ableToOutput as TRUE now(this TRUE will be updated in part B as if it exactly can be output.)

We can first make an array S contains all the x position of given rectangles as the point tuple of (s, h, e, ptToRectangle, ableToOutput), that is each rectangle T0 (x1, x2, h) making 2 point tuples (x1, h, L, T0, TRUE) and (x2, h, R, T0, TRUE), and then use Heap-Sort to make these tuples as increasing x orders (the sorting KEY value is the first element value in the tuple). Notice that if duplicated KEY values occur, we sort the node according to the edge e in order to make part B efficient to reduce duplication:

a) left edge x position (e = L) is always prior to the right edge (e = R)
b) if both left edge (e = L), greater height h valued point has priority.
c) if both right edge (e = R), smaller height h valued point has priority.

Then we store these (s, h, e, ptToRectangle, ableToOutput) in the array S. And since the given array of rectangles is sized n, there are 2n x positions to be sorted, then the size of S should be 2n.

Now each point in S should be all our potential output points since the outline point's x position must be either one of the rectangle edge positions, then S is ready for being updated.

B. Build up Searching Max Heap H to get the required output x position 's height.

Our Searching Max Heap H is actually tracking the overlapping sets of given rectangles for each potential output point in S, that is to say, each point in S will be included in some rectangles, and we only need to get the highest height among such overlapping rectangles for each point. Each node of H represents a given rectangle and the KEY is the height of this rectangle. We update our Max Heap in the loop.

So for each point p in S:

-if p is the left edge of a rectangle (p.e = L), use the insert operation of Heap to insert this rectangle (p.ptToRectangle) as a node in H

-if p is the right edge of a rectangle (p.e = R), remove p.ptToRectangle from our Heap H and swap nodes relatively to always keep the Max-Heapify rule

Then use the getMax operation of Heap to get the value h0 from our current Heap H and set it as the p's height h. If our Heap H is empty, set p's height h as 0.

Now we reduce the duplication: 1), we only need each x position once; 2)if two adjacent x positions has the same height. So we also need a pointer to keep track the scanning point after the previous loop, denote as prePoint.

-if current p.s == prePoint.s, set prePoint.ableToOutput as FALSE (we only need the latest point of same x position points);

-if current p.h == prePoint.h, set p.ableToOutput as FALSE (we only need the first point of same height points);

Update the prePoint as p and end the loop.

Now our array S is updated and ready to output.

C. Output

Finally, we output all the x position s and height h of the point p in S as a tuple (s, h) if p.ableToOutput is TRUE, and the array of all satisfied points is what we need.

(b)

I will explain the worst-case time complexity related to my algorithm.

In part A we use heap sort on 2n keys, so we just do 2n times insert operation of Heap (which is O(log n) per insert that we learned from class) , notice that the updated key comparison rules cos constant time per insert, and 2n times extract-min operations (which is O(1) per extract that we learned from class), so the total cost of this part is O(n log n).

In part B, for the outer loop, we execute 2n times loops to update the array S. For each inner loop, we do either one insert operation of Heap or Delate operation of Heap while keeping the Binary Tree property and Max-Heapify property (which is both O(log n) per operation that we learned from class), then do getMax operation of Heap (which is O(1) per operation that we learned from class), and the duplication tracking costs constant time. So the total cost of this part is O(n log n).

In part C we visit all 2n elements in array S, each visit cost constant time to output the left end point, so the total cost of this part is O(n).

Consider all parts, the total cost of my algorithm is O(n log n).

Question 3

(a)

Let Binary(n) denote the binary representation of n. According to CLRS, for the binomial tree B_k, there are 2^k nodes, which means each binomial tree has a size which is a power of 2 and the binomial trees required to represent n nodes are uniquely determined. Since we can have at most one of each binomial trees and at most 1 as any digit of Binary(n), we can use Binary(n) to represent the number of binomial trees in a binomial heap —- include B_k if and only if the kth position of Binary(n) is 1.
Therefore, a binomial heap with n elements has $\alpha(n)$ binomial trees, where $\alpha(n)$ is the number of 1's in the binary representation of n. Let T_i denote the trees of the binomial heap, where $1 \leq i \leq \alpha(n)$. Since the number of edges in a tree is the number of vertices minus one, T_i with n_i vertices has $n_i - 1$ edges.

$\sum_{i=1}^{i=\alpha(n)} n_i - 1 = \sum_{i=1}^{i=\alpha(n)} n_i - \alpha(n) = n - \alpha(n)$

Therefore, the total number of edges in the binomial heap is exactly $n - \alpha(n)$.

(b)

Since we do pairwise comparisons when we union two separate binomial trees, the number of pairwise comparisons between the keys of the binomial heap that is required to do k consecutive insertions is equal to the number of new edges created during these insertions.

By part (a), the binomial heap H has $n - \alpha(n)$ edges before insertion. After k consecutive insertions, H with $(n+k)$ vertices has $(n+k) - \alpha(n+k)$ edges. So the number of new edges created equals to $(n+k) - \alpha(n+k) - (n - \alpha(n)) = k + \alpha(n) - \alpha(n+k)$.

Since $\alpha(n)$ is the number of 1's in the binary representation of n, $n \leq \lfloor \log_2 n \rfloor + 1$. Therefore, k consecutive insertions require at most $k + \alpha(n)$ comparisons, which is $k + \lfloor \log_2 n \rfloor + 1$ comparisons. When $k > \log_2 n$, k dominates the equation $k + \lfloor \log_2 n \rfloor + 1$ so that k consecutive insertions require O(k) pairwise comparisons.