Romerico David, Haley Elliott, Julian Halsey, Nkechiyem Molokwu
COSC 336
03/12/2024

Assignment 4

Exercise 1:

1. $\Theta(n^2)$

2. $\Theta(\log(n))$

3. $\Theta(n\log(n))$

4. $\Theta(n^2)$

5. $\Theta(n^4)$

6. $\Theta(\log(n))$

7. $\Theta(n\log(n))$

8. $\Theta(n\log(n))$

9. $\Theta(n^2)$

Exercise 2:

1. -2

2. $2\ 3 + 5 * 15-$

Exercise 3:

1. $T_A(n) = T(n/2) + 1$
Using Master Theorem, $n^{log_2 1}$ vs $1 = n^0$ vs 1
$\therefore T_A(n) = \Theta(logn)$

2. $T_B(n) = 2T(n/2) + 1$
Using Master Theorem, $n^{log_2 2}$ vs. $1 = n$ vs. 1

$n/1$ is a polynomial
$\therefore T_B(n) = \Theta(n)$

3. Algorithm A is the faster of the two.

Exercise 4: The runtime of the code is $\Theta(\log^2(n))$.

Programming Task:

Description of Algorithm: Our algorithm calculates the maximum area under a histogram given an array of (x, y) coordinates. We implemented a stack to keep track of the indices of our rectangles and arrays to store the area, height, and length of each rectangle. To compute the height, we determined a common function based on the index 'i' and the index of the y-value for the ith rectangle. We did a similar approach for determining the areas and lengths of each rectangle. Lastly, we calculated the area of the top bar by using a common function to compute the area from the rightmost x-value and leftmost x-value (beginning x-value of the ith rectangle).

Code:

```
public static int getMaxArea(int hist[], int total, int n)
    {
     // Stack to keep track of indices, 'total' represents
        the number of elements, and 'n' is the number of
        rectangles
     Stack<Integer> s = new Stack<>();

     // Arrays to store areas, heights, and lengths of each
        rectangle
     int[] areas = new int[n];
     int[] heights = new int[n];
     int[] lengths = new int[n];

     int max_area = 0;
     int tp; // To store the index of the top of the stack
     int area_with_top; // To store the area with top bar as
         the smallest bar

     // Calculating the area, height, and length of each
        rectangle then storing them in the respective arrays
```

```
for (int i = 0; i < n; i++) {
    areas[i] = ((hist[4 + (4 * i)] - hist[2 + (4 * i)])
        * (hist[3 + (4 * i)]));
    heights[i] = (hist[3 + (4 * i)]);
    lengths[i] = (hist[4 + (4 * i)] - hist[2 + (4 * i)
       ]);
}

int i = 0;
while (i < n) {
    // If the rectangle is taller or equal to the top
       bar, then push its index onto the stack
    if (s.empty() || heights[s.peek()] <= heights[i])
        s.push(i++);
    /*
        * If the rectangle is shorter than the top bar,
        * then calculate the area of the rectangle with
            the top as the smallest rectangle
        * 'i' is the right index for the top and the
           element before the top is the left index
        */
    else {
        tp = s.peek();
        s.pop();

        // Calculate the area with the rectangle at
           index 'tp' as the smallest rectangle
        // If the stack is empty, multiply height by
           the x-value of the rightmost vertex
        // Otherwise, subtract the x-values of the
           rightmost and leftmost vertices
        // (rightmost vertex - leftmost vertex)

        area_with_top = heights[tp]
                * (s.empty() ? hist[2 + (4 * i)] : hist
                   [2 + (4 * i)] - hist[4 + (4 * s.peek
                   ())]);

        if (max_area < area_with_top)
            max_area = area_with_top;
```

```
        }
    }

    // Pop the remaining rectangles from the stack
    // and calculate the area with every popped rectangle
       as the smallest rectangle
    while (s.empty() == false) {
        tp = s.peek();
        s.pop();

        // Same idea and calculation from previous loop
        area_with_top = heights[tp]
                * (s.empty() ? (hist[2 + (4 * i)] - hist
                   [0]) : hist[2 + (4 * i)] - hist[4 + (4 *
                    s.peek())]);
        if (max_area < area_with_top)
            max_area = area_with_top;
    }
    return max_area;
}
```

Results Table:

Data Set #	Largest Area
4.3	16
4.4	1,475,958

Table 1: Programming Task