Romerico David, Haley Elliott, Julian Halsey, Nkechiyem Molokwu
COSC 336
02/15/2024

Assignment 1

Exercise 1:

1. $T_a(n) = \Theta(n)$

$T_b(n) = \Theta(n)$

$T_c(n) = \Theta(log_2(n))$

2. Yes, the running time for (b) is $O(n)$ and the running time for (a) is $O(n)$. Therefore $T_b(n) \leq T_a(n)$ and $T_b(n) = O(T_a(n))$

3. No, the running time for (c) is $O(log_2(n))$ and the running time for (a) is $O(n)$. Therefore $T_c(n) \neq T_a(n)$ and $T_c(n) \neq \Theta(T_a(n))$.

Exercise 2:

An example of a function $f(n)$ with the property that $f(n)$ s $\omega(n^2)$ and also $f(n)$ is $o(n^3)$ is: $n^2 \log(n)$.

Exercise 3:

$(3n + 7)(2n + 3) = 6n^2 + 23n + 21$
$Max[\Theta(6n^2), \Theta(23n), \Theta(21))] = Max[\Theta(n^2), \Theta(n), \Theta(1)] = \Theta(n^2)$
The running time of this program is: $\Theta(n^2)$

Programming Task 1:

Description of Algorithm:

Our algorithm iterates over the nums array using dynamic programming. The thought process was that if we want to compute the longest increasing subsequence at any given ith index, we will determine the longest increasing subsequence at each element up to nums[i] and continuously update d[i] by storing the largest of those longest increasing subsequences. Additionally, this led us to assume that at any given element, the smallest possible increasing subsequence is the sequence formed by itself. This means the elements

of d[i] are initialized to 1 but since there are no elements prior to the first element, d[0] will always remain equal to 1.

We had an outer loop traverse from i = 1 to n = the length of nums and an inner loop from j = 0 to i. If nums[i] is greater than nums[j], it means nums[i] is part of an increasing subsequence that includes nums[j] (at index j). Hence, we can update d[i] to be the maximum between its current value and d[j] + 1. This will be done over all the elements of the nums array. The running time for our program is $O(n^2)$ since we are iterating over each element and comparing it with all the elements up to it.

Code:

```
public static int longestIncreasingSub(int[] nums) {
    /*
     * Base cases for an early exit if size of nums is
     * less than or equal to 1
     */
    if (nums.length == 0)
        return 0;
    else if (nums.length == 1)
        return 1;
    int n = nums.length;
    int[] d = new int[n];
    /*
     * Each element is at least a subsequence of itself
     */
    for (int i = 0; i < n; i++)
        d[i] = 1;
    /*
     * At any given ith index, we will determine the longest
     * increasing subsequence at each element up to nums[i]
     * and continuously update d[i] by storing the largest
     * of those longest increasing subsequences
     */
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j])
                d[i] = Math.max(d[i], d[j] + 1);
        }
```

```
        }
        /*
         * Return the largest increasing subsequence found
         * in the d array
         */
        int max = d[0];
        for (int i = 1; i < n; i++)
            if (d[i] > max)
                max = d[i];

        return max;
    }
```

Results Table:

Data Set #	LIS
1	4
2	10
3	9

Table 1: Programming Task 1

Programming Task 2:

Description of Algorithm:

Our algorithm iterates over the nums array using dynamic programming. The thought process was that if we want to compute the longest decreasing subsequence at any given ith index, we will determine the longest decreasing subsequence at each element up to nums[i] and continuously update d[i] by storing the largest of those longest decreasing subsequences. Additionally, this led us to assume that at any given element, the smallest possible decreasing subsequence is the sequence formed by itself. This means the elements of d[i] are initialized to 1 but since there are no elements prior to the first element, d[0] will always remain equal to 1.

We had an outer loop traverse from i = 1 to n = the length of nums and an inner loop from j = 0 to i. If nums[i] is less than nums[j], it means nums[i]

is part of a decreasing subsequence that includes nums[j] (at index j). Hence, we can update d[i] to be the maximum between its current value and d[j] + 1. This will be done over all the elements of the nums array. The running time for our program is $O(n^2)$ since we are iterating over each element and comparing it with all the elements up to it.

Code:

```
public static int longestDecreasingSub(int[] nums) {
    /*
     * Base cases for an early exit if size of nums is
     * less than or equal to 1
     */
    if (nums.length == 0)
        return 0;
    else if (nums.length == 1)
        return 1;

    int n = nums.length;
    int[] d = new int[n];

    /*
     * Each element is at least a subsequence of itself
     */
    for (int i = 0; i < n; i++)
        d[i] = 1;

    /*
     * At any given ith index, we will determine the
     * longest decreasing subsequence at each element
     * up to nums[i] and continuously update d[i] by
     * storing the largest of those longest decreasing
     * subsequences
     *
    */
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] < nums[j])
                d[i] = Math.max(d[i], d[j] + 1);
```

```
            }
        }

        /*
         * Return the largest decreasing subsequence in
         * the d array
         */
        int max = d[0];
        for (int i = 1; i < n; i++)
            if (d[i] > max)
                max = d[i];

        return max;
    }
```

Results Table:

Data Set #	LDS
1	5
2	11
3	10

Table 2: Programming Task 2