Hudson Cho, Ryan Wilson, Jesse Washburn, Colin Shuster, Samhith Patibandla
COSC 336
02/18/2025

Assignment 2

Exercise 1. For (b), (c), (d), the functions t_1, t_2, t_3, t_4 you pick must be selected from the common functions we have discussed, namely polynomials, logarithms, exponentials, factorial.

a. Find a Θ evaluation for the function $(4n+1)4^{\log(n)}$. (Hint: $4^{\log(n)}$ can be written in a simpler way.)

$(4n+1)4^{\log(n)} = \Theta(n2^n)$

b. Give an example of two functions $t_1(n)$ and $t_2(n)$ that satisfy the relations: $t_1(n) = \Theta(n^2)$, $t_2(n) = \Theta(n^2)$ and $t_1(n) - t_2(n) = o(n^2)$.

Let $t_1(n) = n^2 + n$ and $t_2(n) = n^2$ $t_1(n) - t_2(n) = n = \Theta(n)$

c. Give an example of a function $t_3(n)$ such that $t_3(n) = \Theta(t_3(2n))$.

$t_3(n) = n^2$
$\because t_3(n) = \Theta(n^2)$ and $t_3(2n) = \Theta(2n^2) \Rightarrow \Theta(n^2)$

d. Give an example of a function $t_4(n)$ such that $t_4(n) = o(t_4(2n))$.

$t_4(n) = 2^n$
$\because t_4(n) = 2^n = \Theta(2^n)$ but $t_4(2n) = 2^{2n} = \Theta(2^{2n})$

Exercise 2. Indicate, for each pair of expressions (A, B) in the table below, whether A is $O, o, \Omega, \omega,$ or Θ of B. Assume that $k \geq 1, \epsilon > 0$ and $c > 1$, are constants. Your answer should be "yes" or "no" for each cell. For example the entry on the first cell in the top row is "yes" because $\log^k n = O(n^\epsilon)$. (Note: in row c all the entries are "no", because $n^{\sin n}$ oscillates.)

	A	**B**	O	o	Ω	ω	Θ
a.	$\log^k(n)$	n^ϵ	yes	no	no	no	no
b.	n^k	c^n	yes	yes	no	no	no
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$	no	no	yes	yes	no
e.	$n^{\log c}$	$c^{\log n}$	yes	no	no	no	no
f.	$\log(n!)$	$\log(n^n)$	yes	yes	no	no	no

Exercise 3. For each of the following program fragments give a $\Theta(\cdot)$ estimation of the running time as a function of n.

(a)
```
sum = 0;
for (int i = 0; i< n * n; i++) {
      for(int j =0;  j < n/2; j++)
        sum++;
}
```

Given as $f(n) = n^2(1/2 * n) \Rightarrow \Theta(n^3)$

(b)
```
sum = 0;
for (int i = 0; i < n; i++) {
   sum++;
}
for(int j = 0;  j < n / 2; j++){
   sum++;
}
```

Given as $f(n) = n + (n/2) \Rightarrow \Theta(n)$

(c)
```
sum = 0;
for (int i = 0; i < n * n; i++) {
   for(int j = 0;  j < n * n; j++)
       sum++
}
```

Given as $f(n) = n^2 * n^2 \Rightarrow \Theta(n^4)$

(d)
```
sum = 0;
for (int i = 1; i < n; i = 2 * i)
   sum++;
```

Given as $f(n) = log_2(n) \Rightarrow \Theta(log_2(n))$

(e)
```
sum = 0;
for (int i = 0; i < n; i++) {
   for(int j = 1;  j < n * n; j = 2 * j)
       sum++;
}
```

Given as $f(n) = n * Log_2(n^2) \Rightarrow \Theta(nlog_2(n))$

Exercise 4.

a. Compute the sum $S_1 = 500 + 501 + 502 + 503 + \ldots + 999$ (the sum of all integers from 500 to 999). Do not use a program.

$$S_1 = 500 + 501 + 502 + 503 + \ldots + 999 = \sum_{i=0}^{499} i + \sum_{i=0}^{499} 500$$

$$\Rightarrow S_1 = \frac{(499+1) * 499}{2} + (500 * (499+1) = 374,750$$

b. Compute the sum $S_2 = 1+3+5+\ldots+999$ (the sum of all odd integers from 1 to 999). Do not use a program.

$S_2 = 1 + 3 + 5 + 7 + 9 + 11 + ... + 999 = ?$
For any range of integers $[1, x]$ the number of odd and even numbers can be calculated...

if x is odd there are $\frac{(x+1)}{2} = n$
if x is even $\frac{x}{2} = n$

seen at $n = 2$, $Odds_{n=2} = \{1, 3\} \Rightarrow \frac{3+1}{2} = 2$
seen at $n = 3$, $Odds_{n=3} = \{1, 3, 5\} \Rightarrow \frac{5+1}{2} = 3$
seen at $n = 4$, $Odds_{n=4} = \{1, 3, 5, 7\} \Rightarrow \frac{7+1}{2} = 4$

This rule can be easily seen with even numbers...
$Evens_n = \{(2*1), (2*2), (2*3), (2*4), ..., (2*x) = (999+1)\}$
$\Rightarrow x = 50$

The summation of odds can be calculated using $S_n = n^2$
seen at $n = 2$, $S_{n=2} = \{1+3\} = 4 \Rightarrow 2^2 = 4$
seen at $n = 3$, $S_{n=3} = \{1+3+5\} = 9 \Rightarrow 3^2 = 9$
seen at $n = 4$, $S_{n=4} = \{1+3+5+7\} = 16 \Rightarrow 4^2 = 16$

$S_2 = 50^2 = 250,000$

c. A group of 30 persons need to form a committee of 4 persons. How many such committees are possible?

$\binom{N}{k} = \binom{30}{4} = \frac{30!}{4!(30-4)!} = \frac{30*29*28*27*26*25*...*2*1}{4!*26!} = \frac{30*29*28*27*(1)}{4!*(1)} \Rightarrow$

$\binom{30}{4} = \frac{657720}{24} = 27405$

d. Let C_n be the number of committees of 4 persons selected from a group of n persons. Is the estimation $C_n = o(n^3)$ correct? Justify your answer. (Hint: using the formula $\binom{n}{k}$, you can express the number of committees as a function of n.)

Let $C_n = \frac{n!}{k!(n-k)!}$ used in above $C_{30} = 27405 \Leftarrow \binom{30}{4}$ Given $C_n = \frac{n!}{k!(n-k)!}$ and $g(n) = n^3$

We can use a limit test to determine validity of $C_n = o(n^3)$

$$\lim_{n\to\infty} \frac{\frac{n!}{4!(n-4)!}}{n^3}$$

$$\lim_{n\to\infty} \frac{n!}{4!(n-4)!(n^3)}$$

$$\frac{1}{4!} * \lim_{n\to\infty} \frac{n!}{(n-4)!(n^3)}$$

$$\frac{1}{4!} \lim_{n\to\infty} \frac{n*(n-1)(n-2)*(n-3)*(n-4)*...*(1)}{(n-4)*(n-5)*...(1)*n^3}$$

$$\frac{1}{4!} \lim_{n\to\infty} \frac{n*(n-1)*(n-2)*(n-3)}{n^3}$$

$$\frac{1}{4!}\lim_{n\to\infty}\frac{n^4-6n^3+11n^2-6n}{n^3}$$

$$\frac{1}{4!}\lim_{n\to\infty}[\frac{n^4}{n^3}*\frac{-6n^3}{n^3}*\frac{11n^2}{n^3}*\frac{-6n}{n^3}]$$

$$\lim_{n\to\infty}\frac{n!}{4!(n-4)!(n^3)}=\infty$$

$\Rightarrow$ No, $C_n=\omega(n^3)$

Exercise 5. Find a $\Theta(\cdot)$ evaluation for the sum

$$S=1^2\sqrt{1}+2^2\sqrt{2}+3^2\sqrt{3}+\ldots+n^2\sqrt{n}.$$

In other words, find a function $f(n)$ such that $S=\Theta(f(n))$. Show the work for both the upper bound and the lower bound. You can use the technique with integrals, or the method with bounding the terms of the sum.

$$S=\sum_{x=1}^{n}x^2\sqrt{x}\approx\int_1^n x^2\sqrt{x}$$

$$\int_1^n x^2\sqrt{x}=\int_1^n x^{4/2}x^{1/2}=\int_1^n x^{5/2}$$

$$\int_1^n x^{5/2}=\frac{2}{7}x^{7/2}\Big|_1^n$$

$$\frac{2}{7}x^{7/2}\Big|_1^n=\frac{2}{7}n^{3.5}-\frac{2}{7}$$

$$\therefore S=\Theta(n^{3.5})$$

Programming Task 1: This dynamic programming algorithm computes the longest increasing subsequence by using an array $d[i]$ to store the length of the longest increasing subsequence ending at index i (with the condition that for any valid subsequence, $a[j] < a[i]$ for $j < i$) and a companion array `prev`[] to remember the index of the previous element that contributed to that length. As the algorithm iterates over the input array $a[]$, it updates $d[i]$ by comparing $a[i]$ with every previous element $a[j]$ and, if $a[j] < a[i]$ and $d[j]+1$ is greater than the current $d[i]$, it updates $d[i]$ and records j in `prev`$[i]$. After filling the d array, the algorithm finds the index with the maximum value and rebuilds the longest subsequence by backtracking through `prev`[] from that index, collecting the elements in reverse order before printing the final sequence.

Input	**Output**
10, 9, 2, 5, 3, 101, 7, 18	4, (2, 5, 7, 18)
186, 359, 274, 927, 890, 520, 571, 310, 916, 798, 732, 23, 196, 579, 426,188, 524, 991, 91, 150, 117, 565, 993, 615, 48, 811, 594, 303, 191, 505, 724, 818, 536, 416, 179, 485, 334, 74, 998, 100, 197, 768, 421, 114, 739, 636, 356, 908, 477, 656	10, (186, 274, 310, 426, 524, 565, 615, 811, 818, 998)
318, 536, 390, 598, 602, 408, 254, 868, 379, 565, 206, 619, 936, 195, 123, 314, 729, 608, 148, 540, 256, 768, 404, 190, 559, 1000, 482, 141, 26, 230, 550, 881, 759, 122, 878, 350, 756, 82, 562, 897, 508, 853, 317, 380, 807, 23, 506, 98, 757, 247	9, (318, 536, 598, 602, 619, 729, 768, 881, 897)

Raw Code for Programming Task 1

```
import java.util.Arrays;

public class Assignment2Task1 {
    public static void main(String[] args) {
        // Define the sequences
        int[][] sequences = {
                {10, 9, 2, 5, 3, 101, 7, 18},
                {
                        186, 359, 274, 927, 890, 520, 571,
                           310, 916, 798, 732, 23, 196, 579,
                        426, 188, 524, 991, 91, 150, 117,
                           565, 993, 615, 48, 811, 594, 303,
                           191,
                        505, 724, 818, 536, 416, 179, 485,
                           334, 74, 998, 100, 197, 768, 421,
                        114, 739, 636, 356, 908, 477, 656
                },
                {
                        318, 536, 390, 598, 602, 408, 254,
                           868, 379, 565, 206, 619, 936, 195,
                        123, 314, 729, 608, 148, 540, 256,
                           768, 404, 190, 559, 1000, 482,
                           141, 26,
                        230, 550, 881, 759, 122, 878, 350,
                           756, 82, 562, 897, 508, 853, 317,
                        380, 807, 23, 506, 98, 757, 247
                }
        };

        // Process each sequence
        for (int i = 0; i < sequences.length; i++) {
            int[] lis = longestIncreasingSubsequence(
               sequences[i]);
            System.out.println("Sequence " + (i + 1) + ":");
            System.out.println("    Length: " + lis.length);
            System.out.println("    Subsequence: " + Arrays.
               toString(lis));
            System.out.println();
        }
    }

    /**
     * Computes the longest increasing subsequence of the
        input array
```

```
     * @param nums the input array of integers
     * @return an array representing the longest increasing
        subsequence
     */
    public static int[] longestIncreasingSubsequence(int[]
       nums) {
        int n = nums.length;
        if (n == 0) {
            return new int[0];
        }

        // d[i] stores the length of the longest increasing
           subsequence ending at index i.
        int[] d = new int[n];
        // prev[i] stores the index of the previous element
           in the subsequence ending at index i.
        int[] prev = new int[n];

        // Initialize: each element is an increasing
           subsequence of length 1, and no predecessor.
        for (int i = 0; i < n; i++) {
            d[i] = 1;
            prev[i] = -1;
        }

        // Build the d and prev arrays.
        // Process each element nums[i] to build the longest
           increasing subsequence ending at that index
        for (int i = 0; i < n; i++) {
            // For each element nums[i], examine all elements
                before it
            for (int j = 0; j < i; j++) {
                // Check two conditions:
                // 1. nums[j] < nums[i]: Determines if adding
                    nums[i] to the subsequence ending at j
                   maintains increasing order.
                // 2. d[j] + 1 > d[i]: Determines if
                   extending the subsequence ending at j by
                   nums[i] results in a longer subsequence
                   than the current known subsequence ending
                   at i
                if (nums[j] < nums[i] && d[j] + 1 > d[i]) {
                    d[i] = d[j] + 1; // Update d[i] to be the
                        length of the new longer subsequence
                       ending at i
```

```
                    prev[i] = j;     // Set prev[i] to j to
                        record that the best subsequence
                        ending at i comes from extending the
                        subsequence ending at j
                }
            }
        }

        // Find the index of the maximum value in d.
        int maxIndex = 0;
        for (int i = 0; i < n; i++) {
            if (d[i] > d[maxIndex]) {
                maxIndex = i;
            }
        }

        // Reconstruct the longest increasing subsequence
            into an array: lis []
        int len = d[maxIndex];
        int[] lis = new int[len];
        int index = maxIndex;
        for (int i = len - 1; i >= 0; i--) {
            lis[i] = nums[index]; // Place the current
                element in the correct position in the lis
                array
            index = prev[index]; // Move to the predecessor
                element
        }
        return lis;
    }
}
```

Programming Task 2: This dynamic programming solution for the longest decreasing subsequence is similar to Task 1, but with the update condition changed to $a[j] > a[i]$ to ensure a strictly decreasing order. It uses the same arrays $d[i]$ and `prev[]` to track the length of the longest decreasing subsequence ending at each index and the predecessor indices. After processing the input, the algorithm locates the maximum value in d, then reconstructs the subsequence by backtracking through `prev[]` in the same manner as in Task 1, and finally prints the resulting subsequence.

Input	**Output**
4, 9, 2, 5, 3, 101, 7, 18, 2, 1	5, (9, 5, 3, 2, 1)
186, 359, 274, 927, 890, 520, 571, 310, 916, 798, 732, 23, 196, 579, 426,188, 524, 991, 91, 150, 117, 565, 993, 615, 48, 811, 594, 303, 191, 505, 724, 818, 536, 416, 179, 485, 334, 74, 998, 100, 197, 768, 421, 114, 739, 636, 356, 908, 477, 656	11, (927, 890, 798, 732, 579, 524, 505, 416, 334, 197, 114)
318, 536, 390, 598, 602, 408, 254, 868, 379, 565, 206, 619, 936, 195, 123, 314, 729, 608, 148, 540, 256, 768, 404, 190, 559, 1000, 482, 141, 26, 230, 550, 881, 759, 122, 878, 350, 756, 82, 562, 897, 508, 853, 317, 380, 807, 23, 506, 98, 757, 247	10, (536, 390, 254, 206, 195, 148, 141, 122, 82, 23)

Raw Code for Programming Task 2

```
import java.util.Arrays;

public class Assignment2Task2 {
    public static void main(String[] args) {
        // Define the sequences
        int[][] sequences = {
                {4, 9, 2, 5, 3, 101, 7, 18, 2, 1},
                {
                        186, 359, 274, 927, 890, 520, 571,
                           310, 916, 798, 732, 23, 196, 579,
                        426, 188, 524, 991, 91, 150, 117,
                           565, 993, 615, 48, 811, 594, 303,
                           191,
                        505, 724, 818, 536, 416, 179, 485,
                           334, 74, 998, 100, 197, 768, 421,
                        114, 739, 636, 356, 908, 477, 656
                },
                {
                        318, 536, 390, 598, 602, 408, 254,
                           868, 379, 565, 206, 619, 936, 195,
                        123, 314, 729, 608, 148, 540, 256,
                           768, 404, 190, 559, 1000, 482,
                           141, 26,
                        230, 550, 881, 759, 122, 878, 350,
                           756, 82, 562, 897, 508, 853, 317,
                        380, 807, 23, 506, 98, 757, 247
                }
        };

        // Process each sequence
        for (int i = 0; i < sequences.length; i++) {
            int[] lds = longestDecreasingSubsequence(
               sequences[i]);
            System.out.println("Sequence " + (i + 1) + ":");
            System.out.println("    Length: " + lds.length);
            System.out.println("    Subsequence: " + Arrays.
               toString(lds));
            System.out.println();
        }
    }

    /**
     * Computes the longest decreasing subsequence of the
        input array.
```

```
     * @param nums the input array of integers
     * @return an array representing the longest decreasing
        subsequence
     */
    public static int[] longestDecreasingSubsequence(int[]
       nums) {
        int n = nums.length;
        if (n == 0) {
            return new int[0];
        }

        // d[i] stores the length of the longest decreasing
           subsequence ending at index i.
        int[] d = new int[n];
        // prev[i] stores the index of the previous element
           in the subsequence ending at index i.
        int[] prev = new int[n];

        // Initialize: each element is a decreasing
           subsequence of length 1, and no predecessor.
        for (int i = 0; i < n; i++) {
            d[i] = 1;
            prev[i] = -1;
        }

        // Build the d and prev arrays.
        // Process each element nums[i] to build the longest
           decreasing subsequence ending at that index.
        for (int i = 0; i < n; i++) {
            // For each element, examine all elements before
               it.
            for (int j = 0; j < i; j++) {
                // Check two conditions:
                // 1. nums[j] > nums[i]: so that adding nums[
                   i] maintains a decreasing order.
                // 2. d[j] + 1 > d[i]: that extending the
                   subsequence ending at j produces a longer
                   subsequence than the current known
                   subsequence ending at i
                if (nums[j] > nums[i] && d[j] + 1 > d[i]) {
                    d[i] = d[j] + 1; // Update d[i] to be the
                        length of the new longer subsequence
                       ending at i
                    prev[i] = j;       // Set prev[i] to j to
                       record that the best subsequence
```

```
                        ending at i comes from extending the
                        subsequence ending at j
                }
            }
        }

        // Find the index of the maximum value in d.
        int maxIndex = 0;
        for (int i = 0; i < n; i++) {
            if (d[i] > d[maxIndex]) {
                maxIndex = i;
            }
        }

        // Reconstruct the longest increasing subsequence
           into an array: lds []
        int len = d[maxIndex];
        int[] lds = new int[len];
        int index = maxIndex;
        for (int i = len - 1; i >= 0; i--) {
            lds[i] = nums[index];  // Place the current
               element in the correct position.
            index = prev[index];   // Move to the predecessor
                element.
        }
        return lds;
    }
}
```