CS 161 (Stanford, Winter 2023) Homework 1

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.
What we expect: Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Exercises. The following questions are exercises. We suggest you do these on your own. As with any homework question, though, you may ask the course staff for help.

1 Exercise: Complexity Bounds (6 pt.)

For each blank, indicate whether A_i is in O, Ω, or Θ of B_i. More than one space per row can be valid.

[We are expecting: All valid spaces in the table to be marked (checkmark, "x", etc.). No explanation is required.**]**

(The first two lines are already filled out for you as an example.)

A	**B**	O	Ω	Θ
$10n$	n	✓	✓	✓
10	n	✓		
n^2	$2n$			
n^{2021}	2^n			
$n^{\log 9}$	$9^{\log n}$			
$\log(n!)$	$\log(n^n)$			
$(3/2)^n$	$(2/3)^n$			
3^n	2^n			
$n^{1/\log n}$	1			
$\log^5 n$	$n^{0.5}$			
n^2	$4^{\log n}$			
$n^{0.2}$	$(0.2)^n$			
$\log \log n$	$\sqrt{\log n}$			
$\log(\sqrt{n})$	$\sqrt{\log n}$			

2 Exercise: Big-Oh Definitions

Prove the following:

2.1 (2 pt.)

$f(n) = 2n + 2$ is $O(n)$, but $f(n) = 2^{2n+2}$ is **not** $O(2^n)$.

2.2 (2 pt.)

$f(n) = O(g(n))$, where $f(n) = 6n$ and $g(n) = n^2 - 3n$.

[We are expecting: For both parts, a short but formal proof.**]**

Problems. The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

3 Worst-Case Analysis

A player has a square grid board of x rows and x columns. They place a piece on the board, which will explore it in search of a red star. The piece explores the board following a set pattern. The piece starts on the bottom left corner of the board, and moves right alongside the entire row, covering all its positions. Upon reaching the end of the row, it moves k positions up, thus reaching a new row. In this new row, the piece will move left alongside the entire row, covering all positions, and at the end of the row, it moves $2k$ positions up. The piece will sequentially apply this procedure, exploring the board in a zig-zag pattern, and always doubling the number of positions it goes up at the end of each row, from k to $2k$, to $4k$, to $8k$, and so on. The piece will stop exploring the board if it finds the red star or if it reaches the top-right corner of the board. We assume the board always has a number of rows and columns that allows the piece to end its path exactly on the top-right corner, without overshooting the last row. A sample exploration pattern is illustrated below.

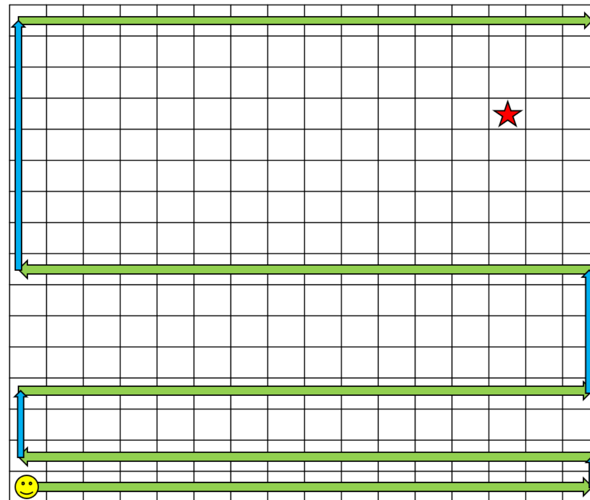

Figure 1: Exploration pattern on a board with $x = 16$ rows and columns for a piece using $k = 1$. The red star is placed in a random location.

3.1 First part (4 pt.)

For the case with $k = 1$, what is the worst-case number of steps the piece will take, in terms of the x rows and columns of the grid, before stopping?

[We are expecting: The worst-case complexity in Big-Oh notation, followed by a concise explanation**]**

3.2 Second part (2 pt.)

In the explained pattern the piece moves alongside the path continuously, covering the path one row at a time. Now instead, the piece has the same positions to cover as in the path explained in the first part but will explore the positions on that path one column at a time. The piece starts at the bottom-left corner then jumps k position up, then $2k$ positions up, then $4k$ positions up, and so on until the top of the first column. Then the piece jumps to the bottom position on the second column and does the same, repeating the pattern until reaching the top position on the last column of the board. Note that the visited positions by the piece are the same in both exploration patterns despite the non-continuous path in the second pattern. What would now be the worst-case number of steps? We still consider $k = 1$ and that the star is in an unknown position.

[We are expecting: The worst-case complexity in Big-Oh notation, followed by a concise explanation**]**

4 Gauging Complexity from Code

Refer to the algorithms below for solving these problems.

Algorithm 1: Search an element in an array of length n

```
Input: An array A of size n and an element e
Output: Index of element e
i ← 0
while i < n do
    if A[i] = e then
        return i
    i ← i + 1
return −1                              // If element e is not found
```

Algorithm 2: Replace an element in an array of length n with a given element

```
Input: An array A of size n, an element e present in the array and the new element E
Output: Updated array
i ← 0
while i < n do
    currElement ← A[i]
    if currElement = e then
        A[i] ← E
        return A
    else
        A[i] ← currElement
    i ← i + 1
return A        // If element e is not found, return the original array
```

Algorithm 3: Sort an array of length n

```
Input: An array A of size n
Output: Sorted array
for i ∈ [0, n − 1) do
    /* Declare i as the index having the current smallest element   */
    smallest ← i
    for j ∈ [i + 1, n) do
        /* Find the smallest element in the range [i, n)              */
        if A[j] ≤ A[smallest] then
            smallest ← j
    swap(A[i], A[smallest]) // Swap numbers present at index i and smallest
return A
```

4.1 (3 pt.)

Give the worst-case run-time complexity for each of the algorithms above.

[We are expecting: Worst Case runtime complexity in Big-Oh notation for each of the above algorithms**]**

4.2 (3 pt.)

Suppose that we run each algorithm by passing in the following inputs

- For Algorithm 1: An element e and an Array A of size 100 such that *e* is **not** present in A.
- For Algorithm 2: An element e and an Array A of size 100 such that *e* is present in A at the 5th index.
- For Algorithm 3: An array A of size 100 such that A is already sorted.

4.2.1

What will be the value of *i* when Algorithm 1 has finished running?

4.2.2

What will be the value of *i* when Algorithm 2 has finished running?

4.2.3

How many times will the command *swap(A[i],A[smallest])* execute by the time Algorithm 3 has finished running?

[We are expecting: A numerical quantity of each of the sub-question.**]**

5 Algorithm Design (10 pt.)

The following definitions will be important for this problem:

Degree of a polynomial: The degree of a polynomial is the highest degree of the individual terms which have a non-zero coefficient. For example the degree of $P(x) = 3x^4+2x^2+3x+1$ is 4 (because the coefficient of x^4 is non-zero)

Product of two polynomials: The product of two polynomials a, b gives a polynomial $c = a \times b$ with degree $\deg(c) = \deg(a) + \deg(b)$. Consider the following example:

Example product: For $a = x^4 + 3x - 5$, $b = x - 1$:

$$\begin{aligned} c = a \times b &= (x^4 + 3x - 5) \times (x - 1) \\ &= x^5 - x^4 + 3x^2 - 3x - 5x + 5 \\ &= x^5 - x^4 + 3x^2 - 8x + 5. \end{aligned}$$

Your objective: You will design an algorithm to efficiently multiply polynomials. Design an algorithm that takes two integers n and m, and also n polynomials, each of degree m, and returns the product of all of these polynomials. You are given a function MultiplyPoly(a, b) which multiplies two polynomials in $O(\deg(a) + \deg(b))$ time and can be used as a subroutine.[1].

Below are two examples of the desired behavior of the algorithm.

Example 1.

Input:

1. $n = 3$
2. $m = 2$
3. $P_1 = x^2 - 1$
4. $P_2 = -2x^2 + x - 3$
5. $P_3 = 6x^2 - 5x$

Output:

$-12x^6 + 16x^5 - 11x^4 - x^3 + 23x^2 - 15x$

Example 2.

Input:

1. $n = 2$

[1] This is a simplified assumption, as in reality the fastest known algorithm for multiplying two polynomials, the *Fast Fourier Transform (FFT)*, is a little bit slower with running time $O(\deg(a) + \deg(b) \log((\deg(a) + \deg(b)))$. You don't need to know anything about FFT for this problem.

2. $m = 4$
3. $P_1 = 2x^4 - x^2 + x - 1$
4. $P_2 = -x^4 + x^3 + x^2 - 3x - 3$

Output:

$-2x^8 + 2x^7 + 3x^6 - 8x^5 - 5x^4 + 3x^3 - x^2 + 3$

[We are expecting: A succinct but clear English description, pseudocode implementation, and running time analysis.**]**