1. (6 points) Place the six functions below in the appropriate blanks to create a list of functions such that each function is big-O of the next function. No justification needed.

$$Verion\ A\ :\ n^{1.2} \quad (\log n)^2 \quad (n-2)! \quad 3^{2n} \quad n\sqrt{n} \quad \log(\log n)$$

$$Verion\ B\ :\ n^{1.7} \quad (\log n)^2 \quad (n-2)! \quad 2^{3n} \quad n\sqrt{n} \quad \log(\log n)$$

The answer below combines all items from bother versions.

$1,\ \log(\log n)\ ,\ (\log n)^2,\ n\ ,\ n^{1.2}\ ,\ n\sqrt{n}\ ,\ n^{1.7}\ ,\ n^2,\ 2^{3n}\ ,\ 3^{2n}\ ,\ (n-2)!\ ,\ n^n$

2. (5 points) For each following statement, answer with **TRUE** or **FALSE**. No justification is needed. Here, all logarithms are base 2. (Below are all the items from both versions combined)

Statement	True/False ?
$(2n^2+3)^3 \in \Theta((3n^3+2)^2)$	TRUE – both are $\Theta(n^6)$
If $f(n) \in \Theta(g(n))$, then $2^{f(n)} \in \Theta(2^{g(n)})$	FALSE – see HW2 #4.j
If $f(n) \in \Omega(n^3)$, then $f(n) \in \Omega(n^4)$	FALSE – consider $f(n) = n^{3.5}$
$\log(n) \in \Omega(\log(n)+n)$	FALSE – linear grows faster than logarithm
$n^3 \log_2 n \in O(n \log_2 n^3)$	FALSE – log-linear grows slower than poly-log
$\log(n)+n \in O(\log(n))$	FALSE – logarithm grows slower than linear
$(3n^3+2)^2 \in \Theta((2n^2+3)^3)$	TRUE – both are $\Theta(n^6)$
If $f(n) \in O(n^4)$, then $f(n) \in O(n^3)$	FALSE – consider $f(n) = n^{3.5}$

3. (4 points) **Use the limit argument** to prove that

$$\begin{aligned} Verion\ A\ &: \ln n \in O(n) \\ Verion\ B\ &: n \in \Omega(\ln n) \end{aligned}$$

Solution. Both versions can be proved by computing the limit $\lim_{n\to\infty} \frac{\ln n}{n}$. By L'Hoptial's rule:

$$\lim_{n\to\infty} \frac{\ln n}{n} = \lim_{n\to\infty} \frac{1/n}{1} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

This shows that n grows faster than $\ln n$, which implies the required identity.

4. (12 points) Suppose f is a function defined by the following recursive formula, where n is a positive integer,

$$\begin{aligned} Version\ A\ &: f(n) = \frac{3f(n-1)+6n}{3} \text{ and } f(0) = 1 \\ Version\ B\ &: g(n) = \frac{8n+4g(n-1)}{4} \text{ and } g(0) = 1 \end{aligned}$$

Find a closed-form formula for the given function. You may use any method discussed in the lecture.

Solution. Both versions give the same recurrence $f(n) = f(n-1) + 2n$, $f(0) = 1$. By unraveling

$$\begin{aligned} f(n) &= f(n-1) + 2n \\ &= f(n-2) + 2(n-1) + 2n \\ &= f(n-3) + 2(n-2) + 2(n-1) + 2n \\ &\vdots \\ &= f(n-k) + 2(n-k+1) + \cdots + 2(n-1) + 2n \\ &\vdots \\ &= f(0) + 2(1) + 2(2) + \cdots + 2(n-1) + 2n \qquad (\text{let } n = k) \\ &= 1 + 2(1 + 2 + \cdots + n) \\ &= 1 + 2 \cdot \frac{n(n+1)}{2} \\ &= n^2 + n + 1. \end{aligned}$$

Note: If you use guess-n-check, then you need to prove that your guess for the closed-formula satisfies the given recurrence.

5. Given two lists A of length m and B of length k, our goal is to construct a list of all elements in list A that are also in list B. (For version B, the input size of the lists are swapped, i.e. A of length k and B of length m)

   (a.) (6 points) Consider the following algorithm to solve this problem. Calculate the runtime of Search1 in Θ notation, in terms of m and k. Justify all your answers by referring specifically to the pseudocode.

   procedure Search1(List A of size m, List B of size k)

   1. Initialize an empty list L.
   2. **for** each item $a \in A$,
   3.   **if** LinearSearch$(a, B) \neq 0$ **then**
   4.   Append a to list L.
   5. **return** L

   Note: The LinearSearch algorithm used in line 3 is given below. This is the same algorithm discussed in lectures.

   procedure LinearSearch(x: integer, $a_1, a_2, \cdots, a_n$: distinct integers)

   $i := 1$
   while ($i \leq n$ and $x \neq a_i$)
       $i := i + 1$
   if $i \leq n$ **then** $location := i$
   else $location := 0$
   return $location$

   $\{location$ is the index of the term that equals x or is 0 if x is not found$\}$

   Solution: $\Theta(mk)$. This answer is the same for both versions.

   In Search1, line 1 is constant time, then we do a linear search in a list of size k (list B) a total of m times. Since linear search takes time proportional to k, and we do this in a loop that runs m times, the total runtime for the block of lines $2, 3$, and 4 is $\Theta(mk)$. Finally line 5 takes constant time.

   Thus, the runtime of this algorithm is $\Theta(mk)$.

(b.) (6 points) Here is another algorithm that solves the same problem. Calculate the runtime of Search2 in Θ notation, in terms of m and k. Justify all your answers by referring specifically to the pseudocode.

procedure Search2(List A of size m, List B of size k)

1. Initialize an empty list L.
2. SORT list B.
3. **for** each item $a \in A$,
4.   **if** BinarySearch$(a, B) \neq 0$ **then**
5.   Append a to list L.
6. **return** L

Note: Assume that the SORT algorithm used in line 2 takes time proportional to $n \log n$ on an input list of size n. The BinarySearch algorithm used in line 4 is given below. This is the same algorithm discussed in lectures.

procedure BinarySearch(x: integer, $a_1, a_2, \cdots, a_n$: increasing integers)

1. $i := 1$
2. $j := n$
3. **while** $i < j$
4. $m :=$ floor$((i + j)/2)$
5.   **if** $x > a_m$ **then** $i := m + 1$
6.   **else** $j := m$
7. **if** $x = a_i$ **then** $location := i$
8. **else** $location := 0$
9. **return** $location$

$\{location$ is the index of the term that equals x or is 0 if x is not found$\}$

Solution: $\Theta(k \log k + m \log k)$ for Version A and $\Theta(m \log m + k \log m)$ for Version B .

In Search2, line 1 is constant time, then in line 2, we sort list B, which takes times $k \log k$ since list B is a list of size k.

Then, for each element of A, we do a binary search in a list of size k (list B). Each such binary search takes times $\log k$ and we do m such searches, so the time of lines 3 through 5 is $m \log k$. Line 6 is also constant time.

Since the time of consecutive pieces of code comes from their sum, we know the whole algorithm takes time $\Theta(k \log k + m \log k)$.

Note that we can't drop either term because we don't know which is larger, m or k.

6. This problem is the same for both versions.

   Let n be a nonnegative integer. In this problem, we are given an array of integers $A[1, \ldots, n]$ and an integer x. We wish to compute the **successor** of x in A, which we define as **the smallest element in A which is greater than x.**

   For example, if $A = [8, 4, 2, -7, -5, 6, 2]$ and $x = 2$, then the successor of x in A is 4. Similarly, the successor of -6 in A is -5.

   We define the successor of x in A to be ∞ if there is no integer in A which is greater than x.

   Here is a recursive algorithm which takes as input $A[1, \ldots, n]$ and an integer x, and returns the successor of x in A, as defined above.

   procedure Successor$(A[1, \ldots, n], x)$

   1. **if** $n = 0$ **then return** ∞
   2. $s :=$ Successor$(A[1, \ldots, n-1], x)$
   3. **if** $(A[n] > x$ **and** $A[n] < s)$ **then** $s := A[n]$
   4. **return** s

   (a) (3 points) Let $T(n)$ be the running time of this algorithm. Write a recurrence relation that $T(n)$ satisfies. Remember to **include the base case(s)**. No justification needed.

   Solution. $T(n) = T(n-1) + c$ with the base case $T(0) = d$.

   (b) (8 points) Prove that this algorithm correctly returns the successor of x in A.

   Solution. We shall prove this by induction on n, the length of the array.

   For the base case, when $n = 0$, the algorithm returns ∞, which is correct because for any integer x, there is no integer in A which is greater than x, since there is nothing in A at all.

   Now, as our inductive hypothesis, suppose that for any input array A of length $n-1$ and any integer x, Successor(A, x) correctly returns the successor of x in A. We must show that for any input array A of length n and any integer x, Successor(A, x) correctly returns the successor of x in A.

   If $n > 0$, then in line 2, the algorithm recursively calls itself with an input array of length $n-1$, and by the inductive hypothesis, this recursive call returns the successor of x in $A[1, \ldots, n-1]$. That is, after line 2, s holds the value of the smallest element in $A[1, \ldots, n-1]$ which is greater than x. Then in line 3, s is updated to be $A[n]$ if and only if $A[n]$ is also greater than x and even smaller than the current value of s. Therefore, after line 3, s holds the value of the smallest element in $A[1, \ldots, n]$ which is greater than x. This is by definition the successor of x in A, so the algorithm is correct.

——— END ———