Order Notation For each of the following answer “True” or “False” and give a brief explanation (1 or 2 lines or sentences.) (4 points each)

1. \( n^3 \in O(n^4) \).
2. \( n^3 \in \Theta(n^4) \).
3. \( 2^{2n} \in O(2^n) \).
4. \( \sum_{i=1}^{\log n} 4^i \in \Theta(n^2) \).
5. \( \sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots f(n) \in \Theta(f(n)), \) if \( f \) is any non-negative increasing function.

Divide and Conquer The maximum weight sub-tree problem is as follows.
You are given a balanced binary tree \( T \) of size \( n \), where each node \( i \in T \) has a (not necessarily positive) weight \( w(i) \) for each node \( i \in T \). (Every node in \( T \) has pointers to its left-child, right-child, and parent, and you are given a pointer to the root of the tree. A NIL field for the children means the node is a leaf, and for the parent, means the node is the root. You are given a pointer to the root \( r \) of \( T \).) A rooted sub-tree of \( T \) is a connected sub-graph of \( T \) containing the root \( r \). (So a sub-tree is not necessarily the entire sub-tree rooted at a node. However, it cannot contain the children of a node without containing the node.) You wish to find the maximum possible value of the sum of weights of nodes in a rooted sub-tree \( S \) of \( T \), \( \sum_{i \in S} w(i) \).

Here is a recursive algorithm that solves this problem, given a pointer to the root of \( T \):
\[ \text{MaxWtSubtree}[r] \]
1. IF \( r = \text{NIL} \) return 0.
2. \( A \leftarrow \max(O, \text{MaxWtSubtree}[r.\text{leftchild}]) \)
3. \( B \leftarrow \max(O, \text{MaxWtSubtree}[r.\text{rightchild}]) \)
4. Return \( w[r] + A + B \).

a Give a recurrence and a worst-case time analysis for this algorithm in the case when \( T \) is a complete binary tree of height \( h \) and size \( n = 2^h - 1 \) (10 pts.)
b Prove that the same worst-case bound holds if \( T \) is any tree of size \( n \). (10 pts.)

**Back-tracking and Dynamic Programming** Consider the following problem.

Let \( A[1..n] \) be an array of integers, and \( V \) an integer. A *subsequence* of \( A \) is a not necessarily consecutive list of elements of \( A \) in the same order as they are in \( A \), \( A[i_1], A[i_2], ..., A[i_k] \) where \( k \geq 0 \) and \( 1 \leq i_1 < i_2 < i_3 < ... < i_k \leq n \). A subsequence is *increasing from \( V \)` if if \( V < A[i_1] < A[i_2] < ... < A[i_k] \).

The following backtracking algorithm, given \( A[1..n] \) and \( V \), finds the length \( k \) of the largest subsequence increasing from \( V \) in \( A \).

\[
\text{LISV}[A[1...n] : \text{array of reals, } V : \text{real}] : \text{integer}
\]

1. IF \( n = 1 \) and \( V \geq A[1] \) return 0.
2. IF \( n = 1 \) and \( V < A[1] \) return 1.
4. Return \( \text{Max}(\text{LISV}[A[2..n], A[1]] + 1, \text{LISV}[A[2..n], V]) \)

**Part 1: 5 points** Show the recursion tree of the above algorithm on the following input: \( A[1..5]=(3,9,7,10,11) \), \( V=8 \).

**Part 2: 5 points** Give a bound on the worst-case number of recursive calls the above algorithm could make in terms of \( n \).

**Part 3: 10 points** Give a dynamic programming version of the recurrence.

**Part 4: 5 points** Give a time analysis of this dynamic programming algorithm, in terms of \( n \).

**Part 5: 5 points** Show the array that your dynamic programming algorithm produces on the above example.

**Greedy Algorithms and use of data structures in algorithms** Consider the following *preemptive scheduling problem*. You are trying to schedule jobs on a machine that are arriving at different times, and require different numbers of steps to finish. Your schedule can be preemptive, in that you can start one job, then switch to another, then finish the first job. You are trying to minimize the sum over all jobs of the time they finish.

More precisely, the input is a sequence of \( n \) jobs, \( \text{Job}_i = (a_i, d_i) \), where \( a_i \) is an integer giving the *arrival time* of the job (first time step when we could start the job), and \( d_i \) is a positive integer giving the *duration* of the job, the number of steps required to finish the job. A *schedule* specifies for each time step, which job we are working on. At time step, \( t \), we can only work on \( \text{Job}_i \) if \( a_i \leq t \); and there must be at least \( d_i \) steps where we are working on \( \text{Job}_i \). The *finish time* for \( \text{Job}_i \) is the last time when \( \text{Job}_i \) is scheduled. The objective is to find a schedule that minimizes the sum of all the finish times.
Example: Job 1: Arrives at 8 AM: Practice piano. Duration: 3 hours.
Job 2: Arrives at 9 AM. Answer morning email. Duration 1 hour.
Job 3: Arrives at 11 AM. Do CSE homework. Duration 4 hours.
Finish times: email: 10; piano: 12; homework: 16. Total: 38.

**Part 1: 5 points** The following is an incorrect greedy strategy for this problem. Give a counter-example that shows that this strategy fails to produce optimal schedules.
Earliest Arrival: Sort the jobs from first to last arrival time, breaking ties arbitrarily. Perform each job until it finishes.

**Part 2: 5 points** The following is a correct greedy strategy for this problem. Show the steps of the algorithm on the counter-example you gave above. (If you couldn’t find a counter-example, use the example above.)
Smallest current duration: For each time slot (in order, from first availability time until all jobs are complete), if at least one uncompleted job is available, schedule the uncompleted job with the smallest (current) duration. Then decrement that job’s (current) duration, and repeat.

**Part 3: 10 points** State and prove a modify-the-solution lemma for the smallest current duration strategy.

**Part 4: 10 points** Give an efficient algorithm that carries out the Smallest current duration strategy. Your description should mention which data structures you use, and any pre-processing steps. Give a time analysis. If possible, don’t have the algorithm’s time depend on the durations of jobs, just the number of jobs. (When does the strategy switch from scheduling one job to another? Output the schedule as a set of time intervals when the same job is scheduled, not time steps. For example, instead of J being scheduled in time step 2 and 3 and 4 and 5 and 6, output, “Time 2-6: job J”.)