**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. \( n^3 \in O(n^4) \). Yes, since \( n^3 \leq n^4 \) for \( n \geq 1 \). \( f(n) \in O(g(n)) \) means there is a constant \( c \) so that \( f(n) \leq cg(n) \). So a strictly smaller function is in \( O \) of a larger one.

2. \( n^3 \in \Theta(n^4) \). No, this is not true, since for two functions to be of each other, they have to grow at approximately the same rate. In particular, since \( n^3 < cn^4 \) whenever \( n > 1/c \), we can’t have \( n^3 \geq cn^4 \) for any \( c > 0 \) for sufficiently large \( n \).

3. \( 2^{2n} \in O(2^n) \). No, \( 2^{2n} = 2^n2^n > c2^n \) whenever \( n > \log c \). So \( 2^{2n} \) is not less than \( c2^n \) for any \( c > 0 \) and sufficiently large \( n \).

4. \( \sum_{i=1}^{\log n} 4^i \in \Theta(n^2) \). Yes, \( \sum_{i=1}^{\log n} 4^i = (4^{\log n+1} - 12)/3 = 4/3n^2 - 4 \in \Theta(n^2) \), using the formula for the sum of a geometric series.

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. No, let \( f(n) = n \). Then \( \sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} i = n(n-1)/2 \notin \Theta(n) \).

**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. \( A \leftarrow \max(O, \text{MaxWtSubtree}[r.leftchild]) \)
3. $B \leftarrow \max(O, \text{MaxWtSubtree}[r.\text{rightchild}])$

**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.)

In a complete binary tree of size $n$, both left and right subtrees have size $(n-1)/2 < n/2$. In line 2, we call the algorithm recursively on the left sub-tree, and in line 3 on the right subtree. The rest of the algorithm is constant time. Thus, it’s time on a complete binary tree is given by $T(n) \leq 2T(n/2) + O(1)$. This fits the main recurrence theorem with $a = 2, b = 2, k = 0$. Since $a = 2 > 1 = b^k$, this is a bottom-heavy case, given by $T(n) \in O(n \log_b a) = O(n \log_2 2) = O(n^1) = O(n)$.

**b** Prove that the same worst-case bound holds if $T$ is any tree of size $n$. (10 pts.)

In the general case, let $L$ represent the size of the left sub-tree, and $R$ the size of the right sub-tree. Then $n = L + R + 1$, so $R = n - L - 1$. Then the time for the algorithm on the entire tree is $T(n) = T(L) + T(R) + c$ for some constant $c$. Note that on trees of size 0, we immediately return so $T(0) = c'$ for some constant $c'$. Let $C = \max(c, c')$. We’ll prove by strong induction that $T(n) \leq 2cn + C$. First, this is true for $n = 0$, since $T(0) = c' \leq C(2*0 + 1)$. Second, assume that $T(k) \leq 2ck + C$ for all $0 \leq k \leq n - 1$. Then $T(n) \leq T(L) + T(n-L-1) + c$, where $L$ is the size of the left sub-tree. Since $0 \leq L \leq n - 1$, as is $n - L - 1$, we can apply the induction hypothesis to get $T(L) \leq 2cL + C$ and $T(n-L-1) \leq 2c(n-L-1) + C$. Thus, $T(n) \leq 2cL + C + 2c(n-L-1) + C + c = 2c(n-1) + 2c + c = 2cn - 2c + 2c + c = 2Cn + c \leq 2Cn + C$.

Therefore, by strong induction, $T(n) \leq 2cn + C$ for all $n$. Since $C$ is a constant, $T(n) \in O(n)$.

**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 $A$, $A[i_1], A[i_2], \ldots A[i_k]$ where $k \geq 0$ and $1 \leq i_1 < i_2 < i_3 < \ldots < i_k \leq n$. A subsequence is **increasing from $V$** if if $V < A[i_1] < A[i_2] < \ldots < A[i_k]$.

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

**LISV**($A[1..n]$; array of reals, $V$:real): integer
1. IF $n = 1$ and $V \geq A[1]$ return 0.

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

Since $8 > A[1] = 3$ we call recursively $LISV((9,7,10,11), 8)$. Then we branch two ways: The first recursive call is to $LISV((7,10,11), 9)$. Since $9 > 7$ this recursive call then calls $LISV((10,11), 9)$, which then calls first $LISV((11), 10)$ (returning 1), adds 1 to it, and compares it to $LISV((11), 9)$ which returns 1. Thus, $LISV((10,11), 9)$ returns 2 and we add 1 to that to get 3, for the first subbranch of $LISV((9,7,10,11), 8)$. The second sub-call for $LISV((9,7,10,11), 8)$ is to $LISV((7,10,11), 8)$ This skips to $LISV((10,11), 8)$, which then branches to first $LISV((11), 10) + 1 = 2$ then $LISV((11), 8) = 1$. So the second sub-call returns 2. Thus, the maximum, which is our final answer is 3. This represents the subsequence 9, 10, 11 of length 3.

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

In the worst case, we could make 2 recursive calls, both to arrays of size $n - 1$. This gives $T(n) = 2T(n - 1) + O(1)$, or $T(n) \in O(2^n)$ On an array that is increasing and a $V$ that is less than the smallest element, this will indeed be the number of recursive calls, so this bound is tight.

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

We note that the input to sub-calls have the following format: the array is $A[I..n]$ for some $1 \leq I \leq n$, and the value of $V$ is either the input value or $A[J]$ for some $1 \leq J \leq I - 1$. Thus, we’ll let $L[I][J]$ hold the value $LISV(A[I..n], A[J])$ for $J > 0$ and $L[I][0] = LISV(A[1..n], V)$.

In recursive calls, $I$ is incremented, and $J$ is either increased or stays the same. Thus, a safe bottom up order is in decreasing order of $I$ and any order of $J$.

The dynamic programming algorithm then becomes:

DPLISV[A[1..n]:array of reals, V:real]: integer
1. Initialize $L[1..n][0..n-1]$.
2. Set $A[0]$ to $V$.
3. FOR $J = 0$ TO $n - 1$ do:
      ELSE $L[n,J] \leftarrow 0$.
6. FOR $I = n - 1$ down to 1 do:
7. FOR $J = 0$ to $I - 1$ do:
9. \[ \text{THEN } L[I, J] \leftarrow L[I + 1, J] \]
10. \[ \text{ELSE } L[I, J] \leftarrow \max(L[I + 1, I] + 1, L[I + 1, J]) \]
11. \[ \text{Return } L[1, 0]. \]

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

The main cost is from the nested loops in lines 6 and 7, both of which go for up to \( n \) iterations. The inside of the loop is constant time. Thus, the total time is \( O(n^2) \).

**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.

Say we have 2 jobs. The first arrives at time 0 and has duration 4. The third arrives at time 1 and has duration 1. The above algorithm would schedule them in order of arrival, without preemption. This would complete the first job at time 4 and the second at time 5, for a total completion time of 9. A better schedule is to schedule
the first job from 0 to 1, schedule the second job from 1 to 2, and then complete the first from 2 to 5. The total finish time is then 5 + 2 = 7 < 9. Thus, on this example, the greedy strategy above does not produce an optimal solution.

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.

The optimal schedule for the previous example was obtained by the above algorithm. We ran the only job we had available in the first time slot. Then a second job arrived. The first job had 3 remaining duration, and the second only 1. So we scheduled the second in the second time slot. This completed it, so we scheduled the first in the remaining time slots until it finished.

Part 3: 10 points The following is a proof of a modify-the-solution lemma for the Smallest current duration strategy, with some phrases missing. Fill in the missing spaces in the proof, which have Roman numerals indexing them.

I’ll just fill in the gaps here. Lemma: Let \( t \) be the first time a job is available, and let \( J \) be a job available on \( t \) that has the smallest duration of any such job. Let \( S_2 \) be a schedule that does not schedule \( J \) at time \( t \). Then there is a schedule \( S_1 \) that schedules \( J \) at time \( t \), so that the total finish times for \( S_1 \) is at most that for \( S_2 \).

Proof: Either \( S_2 \) schedules no job at time \( t \), or schedules a job \( J' \) at time \( t \).

Case 1: If \( S_2 \) has no job scheduled at time \( t \), let \( S_1 \) be \( S_2 \) except that it schedules \( J \) at time \( t \) and schedules no job at the last time slot \( J \) was scheduled in \( S_2 \). Then in \( S_1 \), all jobs except \( J \) finish at the same time as in \( S_2 \), and \( J \) finishes earlier, so the total finish time for \( S_1 \) is less than that for \( S_2 \).

Case 2: If \( S_2 \) schedules \( J' \) at time \( t \), let \( d \) be the duration of \( J \) and \( d' \) that of \( J' \). By the definition of \( J \), \( d \leq d' \). Now, either \( J \) finishes before \( J' \) in \( S_2 \), or \( J \) finishes after \( J' \) in \( S_2 \).

Case 2a: If \( J' \) finishes before \( J \) in \( S_2 \), let \( S_1 \) be like \( S_2 \) except that the first \( d \) times \( J' \) is scheduled in \( S_2 \), \( S_1 \) instead schedules \( J \), and each time \( J \) is scheduled in \( S_2 \), \( S_1 \) schedules \( J' \). In \( S_1 \), we finish \( J' \) at the time \( J \) finishes in \( S_2 \), and we finish \( J \) at or before the time \( J' \) finishes in \( S_2 \). (And all other jobs don’t change) So the total finish time for \( S_1 \) is at most that of \( S_2 \).
Case 2b: If $J'$ finishes after $J$ in $S_2$, let $S_1$ be like $S_2$ except that in time $t$, $S_1$ schedules $J$, and the first time $S_2$ schedules $J$, $S_1$ schedules $J'$. Then since $J'$ finishes after $J$ in $S_2$, $J'$ finishes at the same time in $S_1$ and $S_2$. $J$ finishes no later in $S_1$ than it does in $S_2$, so the total finish time for $S_1$ is at most that of $S_2$ in this case, too. Thus, in all cases, the total finish time is no larger for $S_1$ than it is for $S_2$, as claimed.

**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 $J$”.

The idea is that we want to keep track of the available jobs, each with their remaining duration. For each time slot $t$, we need to add the set of jobs that arrived at time $t$, and then schedule the one of smallest duration. So we need to have insert and find min operations. This suggests using a min-heap, sorted by remaining duration. To make it easy to find the newly arrived jobs, we first sort by availability time, and keep a pointer to where we are in the list.

Preemptive $[\text{Jobs}[1..n]]$: List of pairs: $(\text{Job}, \text{interval})$, where each interval is a pair of time slots $t_1, t_2$, meaning schedule job in slots $t_1..t_2$.

1. Sort Jobs by arrival time. Add a fictional job $\text{Job}_{n+1}$ with arrival time infinite and duration 0.
2. Initialize an empty list of job, interval pairs Schedule
3. Initialize an empty min heap $H$ of pairs $(\text{job}, \text{remaining})$ keyed by remaining, of size $n$.
4. $I \leftarrow 1$, $t \leftarrow \text{Job}[1].\text{arrival}$.
5. While $I \leq n$ OR $H$ is non-empty do:
6. While $\text{Job}[I].\text{arrival} = t$ do: InsertH($I, \text{Job}[I].\text{duration}$); $I++$;
7. IF $t + \text{MinH.remaining} \leq \text{Job}[I].\text{arrival}$
8. THEN do:
9. Schedule $\leftarrow$ Schedule $\cup$($\text{MinH.job}, [t,t+\text{MinH.remaining}]$);
10. IF $H.\text{Size} = 1$ THEN $t \leftarrow \text{Job}[I].\text{arrival}$ ELSE $t \leftarrow t + \text{MinH.remaining}$.
11. DeleteMinH;
12. ELSE do:
13. Schedule $\leftarrow$ Schedule $\cup$($\text{MinH.job}, [t, \text{Job}[I].\text{arrival}]$);
14. Decrement $\text{MinH.remaining}$ by $\text{Job}[I].\text{arrival} - t$.
15. $t \leftarrow \text{Job}[I].\text{arrival}$.

We maintain the invariant that the heap contains all of the available jobs with the amount of times remaining for each job. \( I \) points to the next time jobs arrive, and \( t \) is the first unscheduled time when jobs are available. First we insert all newly arrived jobs. We will schedule the available job with the smallest duration starting at time \( t \). We look at the next possible point of time when we might switch jobs: either when the next jobs arrive, or when the current job finishes, whichever comes first. If no new job arrives before the current job finishes, we schedule it to completion, and then check whether we still have jobs to do. If there are no remaining jobs in our heap, we skip ahead to the next time jobs arrive. Otherwise, we schedule the current job until the next job arrives. We decrement the remaining time. The job with infinite arrival time ensures that even after all jobs have arrived, we do the jobs on the heap in order.

The sorting time is \( O(n \log n) \) using mergesort or heapsort. Each job gets inserted to the heap once. So the total time for inserts is \( O(n \log n) \). Each iteration, we either add something new to the heap, or finish the current job. So there will be at most \( 2n \) iterations of the while loop. Not counting the insertions (already counted above), we perform at most one heap operation of cost either \( O(\log n) \) or \( O(1) \) time (delete vs. decrement) and a number of constant-time operations per iteration. So the total time for the loop will be \( O(n \log n) \) as will the total time for the algorithm.