

CSE 101 Final Exam

Topics: Order, Recurrence Relations, Analyzing Loops, Invariants and induction, Divide-and-Conquer, Back-tracking, Dynamic Programming, Greedy Algorithms and Correctness Proofs, Using Data Structures in Algorithms

Time: 3 Hours

June 11, 2003

Some problems have multiple parts -do all parts.

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 Θ 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, $2^{2n} = 2^n 2^n > c2^n$ whenever $n > \log c$. So 2^{2n} is not less than $c2^n$ for any $c > 0$ and sufficiently large n .
4. $\sum_{i=1}^{i=\log n} 4^i \in \Theta(n^2)$ Yes, $\sum_{i=1}^{i=\log n} 4^i = (4^{\log n + 1} - 12)/3 = 4/3n^2 - 4 \in \Theta(n^2)$, using the formula for the sum of a geometric series.
5. $\sum_{i=1}^{i=n} f(i) = f(1) + f(2) + \dots + f(n) \in \Theta(f(n))$, if f is any non-negative increasing function. No, let $f(n) = n$. Then $\sum_{i=1}^{i=n} f(i) = \sum_{i=1}^{i=n} i = n(n-1)/2 \notin \Theta(n)$.

Divide and Conquer: Consider the following problem: We have n nodes along a line, and for each node we have a set of possible colors $PossColors[i] \subseteq \{purple, blue, green, yellow, orange, red\}$. We need to decide whether it is possible to give each node i a color $C[i] \in PossColors[i]$ so that no two neighboring nodes $i, i+1$ get the same color.

The following recursive algorithm solves this problem: Colorable(PossColors[1..n]): Boolean

1. IF $n = 0$ return True;
2. IF $n = 1$ THEN IF $PossColors[1] = \emptyset$ THEN return False;
3. ELSE return True;
4. $m \leftarrow ndiv2$.
5. IF $|PossColors[m]| \geq 3$ THEN Return (Colorable(PossColors[1..m-1])) AND (Colorable(PossColors[m+1..n])); {However we color the rest of the array, we only use two colors for the neighbors of m . So we can use a third color to color m }
6. For each $color \in PossColors[m]$ do:

7. begin{for}
8. IF $m > 1$ THEN $PossColors[m-1] \leftarrow PossColors[m-1] - \{color\}$
9. $PossColors[m+1] \leftarrow PossColors[m+1] - \{color\}$
10. IF (Colorable(PossColors[1..m-1])) AND (Colorable(PossColors[m+1..n]))
THEN Return True; {We can color m color, and successfully color
the rest of the array}
11. IF $m > 1$ THEN $PossColors[m-1] \leftarrow PossColors[m-1] \cup \{color\}$
12. $PossColors[m+1] \leftarrow PossColors[m+1] \cup \{color\}$
13. end{for};
14. Return False;

Part 1: 5 points Illustrate the above algorithm on the following example, as a tree of recursive calls: $PossColor[1] = \{blue, green\}$, $PossColor[2] = \{blue, green\}$, $PossColor[3] = \{red, green, yellow\}$, $PossColor[4] = \{yellow, orange\}$, $PossColor[5] = \{orange\}$, $PossColor[6] = \{orange, green\}$. First, $m = 3$, and there are three elements of $PossColor[3]$. So the algorithm first calls itself on the first two elements, and then on $PossColor[4..6]$.

Running on $PossColor[1..2]$, $m = 1$,

and it tries first setting color 1 to blue (deleting it from $PossColor[2]$. Running on just $PossColor[2]=red$, it returns True; which causes the algorithm to return true on $PossColor[1..2]$

Then running on $PossColor[4..6]$, it again sets $m = 1$, and first tries setting color4=yellow. It runs on $PossColor[5..6]$, and then colors 5 orange. Then it runs on a single element with $PossColor$ green, and returns True.

So then the main procedure returns True.

Part b: 15 points Give a recurrence relation for the worst-case time the above algorithm takes and solve it to get the order of the time.

Answer: If there are more than 2 colors in $Posscolor(n/2)$ then the algorithm calls itself recursively twice on arrays of size at most $n/2$. When there are 2 or fewer colors in $PossColors(n/2)$, for each such color it calls itself twice on arrays of size at most $n/2$. This means that the maximum number of recursive calls is 4, and all recursive calls are to arrays of size $n/2$. The rest of the algorithm is constant time. Thus, we have $T(n) \leq 4T(n/2) + O(1)$. Applying the main recurrence theorem with $a = 4, b = 2, k = 0$, we see that we are in the bottom-heavy case since $4 > 2^0$, and the time will be $T(n) \in O(n^{\log_2 4}) = O(n^2)$.

Back-tracking and Dynamic Programming In the maximum non-consecutive sum problem, we are given an array of real numbers $V[1..n]$. We wish to

find a subset of array positions, $S \subseteq [1..n]$ that maximizes $\sum_{i \in S} V[i]$ subject to no two consecutive array positions being in S . For example, say $V = [10, 14, 12, 6, 13, 4]$, the best solution is to take elements 1, 3, 5 to get a total of $10 + 12 + 13 = 35$. If instead, we try to take the 14 in position 2, we must exclude the 10 and 12 in positions 1 and 3, leaving us with the second best choice 2, 5 giving a total of $14 + 13 = 27$.

We showed an $O(n^2)$ time divide-and-conquer algorithm for this problem. Here, let's use the dynamic programming method. The recursive procedure is based on a case analysis, do we pick position 1 or not? The algorithm just finds the best sum, not the set of positions, but it would be easy to modify. BTMNCs stands for Back-Tracking Maximum Non-consecutive Sum.

BTMNCs[V[1..n]: array of positive reals]: real number;

1. IF $n = 0$ return 0;
2. IF $n = 1$ return $V[1]$;
3. $Sum1 \leftarrow BTMNCs[V[3..n]] + V[1]$ {Case 1: If we include $V[1]$, we cannot include $V[2]$ }
4. $Sum2 \leftarrow BTMNCs[V[2..n]]$. {Case 2: If we do not include $V[1]$, we can include any other positions.}
5. Return $max(Sum1, Sum2)$.

Part 1: 5 points Show the recursion tree of the above algorithm on the array 2, 4, 1, 6, 4

Since I can't draw trees easily, I'll just write it out.

It first calls itself on 1, 6, 4. On this array, it first calls itself on 4, returning 4. Adding 1 gives $Sum1 = 5$. Then to compute $Sum2$, it calls itself on 6, 4. $Sum1$ is then a recursive call on the empty array returning 0 + 6, compared to $Sum2$ is the call on the array 4, returning 4. Hence on 1, 6, 4, $Sum2 = max(6, 4) = 6$, and on 1, 6, 4, we return $max(5, 6) = 6$. Thus, the main procedure sets $Sum1 = 6 + 2 = 8$.

Then it calls itself on 4, 1, 6, 4. First, it calls itself on 6, 4, and as before will return 6, comparing a recursive call on the empty array + 6 to a recursive call on array 4. Thus, it sets $Sum1 = 4 + 6 = 10$. Then it calls itself on 1, 6, 4. Eventually, repeating the above paragraph, this will return 8, which gets saved to $Sum2$. The returned value is thus $max(10, 8) = 10$, which gets saved as $Sum2$ in the main procedure.

Thus, the main procedure returns $max(8, 10) = 10$. This corresponds to picking the first 4 and the 6.

Part 2: 5 points Give a bound on the worst-case number of recursive calls the algorithm could make on an array of size n .

The algorithm makes one call to an array of size $n-1$ and one to one of size $n-2$. Thus the time is given by $T(n) = T(n-1) + T(n-2) + O(1)$. This grows at the same rate as the Fibonacci sequence, $T(n) = O((1 + \sqrt{5})/2)^n$.

Part 3: 10 points Give a polynomial-time dynamic programming version of the recurrence.

We note that the sub-problems that occur are all of the form $BTNCS[V[I..n]]$, or to an empty array. Lets call the value for this call $Best[I]$, with $Best[n+1]$ representing the call to the empty array. The base cases are an empty array or $V[n]$, and the value of I increases when we make recursive calls (either by 1 or 2).

Hence, we get the following *DP* version:

DPMNCS[V[1..n]: array of positive reals]: real number;

1. Initialize $Best[1..n+1]$.
2. $Best[n+1] \leftarrow 0$.
3. $Best[n] \leftarrow V[n]$.
4. FOR $I = n-1$ downto 1 do:
5. $Sum1 \leftarrow Best[I+2] + V[I]$. {Case 1: If we include $V[I]$, we cannot include $V[I+1]$ }
6. $Sum2 \leftarrow Best[I+1]$. {Case 2: If we do not include $V[I]$, we can include any other positions.}
7. $Best[I] \leftarrow \max(Sum1, Sum2)$.
8. Return $Best[1]$.

Part 4: 5 points Give a time analysis of the dynamic programming algorithm.

The main loop executes n times, and the inside of the loop is constant time. Thus the algorithm takes $O(n)$ time.

Part 5: 5 points Show the solution array that your dp algorithm produces on the array 2, 4, 1, 6, 4.

Base cases: $Best[6] = 0, Best[5] = 4$.

Then $Best[4] = \max(6 + Best[6], Best[5]) = \max(6, 4) = 6$

$Best[3] = \max(1 + Best[5], Best[4]) = \max(5, 6) = 6$

$Best[2] = \max(4 + Best[4], Best[3]) = \max(10, 6) = 10$

$Best[1] = \max(2 + Best[3], Best[2]) = \max(8, 10) = 10$

The algorithm returns 10.

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, $Job_i = (a_i, d_i)$, 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 Job_i if $a_i \leq t$; and there must be at least d_i steps where we are working on Job_i . The *finish time* for Job_i is the last time when 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.

Schedule: 8-9: practice piano; 9-10 answer email; 10-12: practice piano; 12-16: do CSE homework.

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[Jobs[1..n]]: List of pairs: (Job, 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 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 (*job, remaining*) keyed by *remaining*, of size n .
4. $I \leftarrow 1, t \leftarrow Job[1].arrival$.
5. While $I \leq n$ OR H is non-empty do:
 6. While $Job[I].arrival = t$ do: InsertH($I, Job[I].duration$); $I++$;
 7. IF $t + MinH.remaining \leq Job[I].arrival$
 8. THEN do:
 9. $Schedule \leftarrow Schedule \cup (MinH.job, [t, t + MinH.remaining])$;
 10. IF $H.Size = 1$ THEN $t \leftarrow Job[I].arrival$ ELSE $t \leftarrow t + MinH.remaining$.
 11. DeleteMinH;
 12. ELSE do:
 13. $Schedule \leftarrow Schedule \cup (MinH.job, [t, Job[I].arrival])$;
 14. Decrement $MinH.remaining$ by $Job[I].arrival - t$.
 15. $t \leftarrow Job[I].arrival$.
16. Return *Schedule*.

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.