

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. $n^3 \in O(n^4)$.
2. $n^3 \in \Theta(n^4)$.
3. $2^{2n} \in O(2^n)$
4. $\sum_{i=1}^{i=\log n} 4^i \in \Theta(n^2)$
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.

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. $m \leftarrow n \text{ div } 2$.
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: $PossColor[1] = \{blue, green\}$, $PossColor[2] = \{blue, green\}$, $PossColor[3] = \{red, green, yellow\}$, $PossColor[4] = \{yellow, orange\}$, $PossColor[5] = \{orange\}$, $PossColor[6] = \{orange, green\}$.

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.

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

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 .

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

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

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

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.

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 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.

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 job J at time t , so that the total finish times for S_1 is II that for S_2 .

Proof: Either S_2 schedules no job at time t , or schedules a job $J' \neq J$ 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 III finish at the same time as in S_2 , and IV finishes earlier, so the total finish time for S_1 is V that for S_2 .

Case 2: If J' is scheduled 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 the time J finishes in S_2 . (And all other jobs don't change) So the total finish time for S_1 is $XIII$ 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 XV in S_1 and S_2 . J finishes XVI in S_1 than it does in S_2 , so the total finish time for S_1 is $XVII$ that of S_2 in this case, too.

Thus, in all cases, the total finish time is no $XVIII$ 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 ".)