CSE 101 Final Exam
Topics: Order, Recurrence Relations, Analyzing Programs, Divide-and-Conquer, Back-tracking, Dynammic Programming, Greedy Algorithms and Correctness Proofs, Data Structures (Heap, Binary Search Tree, B-Tree, Lists), Using Data Structures in Algorithms
Time: 3 Hours
Answer Key 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.) Each is worth 4 points.

1. $(n+1)^2 \in \Theta(n^2)$.
2. $4^n \in O(2^n)$
3. $4^n \in \Omega(2^n)$.
4. $n\lceil \log n \rceil \in \Theta(n \log n)$
5. If $f(n) \in O(g(n))$ then $f(n^2) \in O(g(n^2))$.

1. True; For $n \geq 1$, $n^2 \leq (n+1)^2 \leq (2n)^2 = 4n^2$, so for $n_0 = 1$, $c_0 = 1$ and $c_1 = 4$, it meets the definition of $O(n^2)$ and $\Theta(n^2)$.
2. False. Assume it were true. Then for some $c, n_0 \geq 0$ and all $n \geq n_0$, $4^n \leq c2^n$. But then $4^n/2^n \leq c$ for all such c, so $2^n \leq c$ for all n. In particular, we can pick $n \geq max(n_0, \log c+1)$ and get a contradiction, since then $c \geq 2^n \geq 2c$.
3. True. For all $n \geq 1$, $4^n \geq 2^n$, so it meets the definition of Ω with $n_0 = 1, c = 1$.
4. True. $n \log n \leq n\lceil \log n \rceil \leq n(logn + 1)$. As long as $n \geq 2$, $1 \leq \log n$, and $n(\log n + 1) \leq 2n \log n$. Thus, choosing $n_0 = 2, c_0 = 1$ and $c_1 = 2$, it meets the definitions of O and Ω.
5. True. If $f(n) \in O(g(n))$, then $f(n) \leq cg(n)$ for all $n \geq n_0$. In particular, if $n \geq \sqrt{(}n_0)$, then $n^2 \geq n_0$ and $f(n^2) \leq cg(n^2)$ by substitution. Thus, picking $n_0' = \sqrt{(}n_0)$ and $c_0 = c$, we satisfy the definition of O.

Divide and Conquer: 20 pts Consider the following recursive algorithm: We are given a binary tree T. In addition, at each node x except the root, we are given a positive real value $d(x)$ called the distance between x and its parent, We want to find the distances between every two nodes x, y in the tree and store it in an array $D[x, y]$.

Let L_x represent the left sub-tree rooted at x and R_x the right sub-tree. The algorithm to do so is as follows:

1. Distances(root){compute all distances between pairs of nodes in the sub-tree rooted at *root*. }

2. If $root = NIL$ then halt.
3. ELSE do:
4. begin;{else}
5. Distances(lc(root));
6. Distances(lc(root));
7. For each $I \in L_{root}$, $D[I, root] := D[I, lc(root)] + d(lc(root))$.
8. For each $J \in R_{root}$, $D[J, root] := D[J, rc(root)] + d(rc(root))$.
9. For each $I \in L_{root}$ do:
10. For each $J \in R_{root}$ do:
11. $D[I, J] = D[I, root] + D[J, root]$.
12. end;{else}

Consider the case when the tree is almost perfectly balanced, i.e, for every x, $|L(x)| - 1 \leq |R(x)| \leq |L(x)| + 1$. Give a recurrence relation for the time the algorithm takes in this case, and solve it to get the order of the time.

In the general case, if $l = |L(x)|$ and n is the total size of the tree, then $|R(x)| = n - l - 1$. The algorithm calls itself recursively on $L(x)$, and $R(x)$, and then does three loops the most expensive of which is time $O(|L(x)||R(x)|) = O(l(n - l - 1))$. Thus we have $T(n) \leq T(l) + T(n - l - 1) + O(l(n - l - 1))$. for some $1 \leq l \leq n - 1$. In the case above, both $|L(x)| \leq n/2$ and $|R(x)| \leq n/2$. Thus, in this case we have $T(n) \leq 2T(n/2) + O(n^2/4) = 2T(n/2) + O(n^2)$. Applying Theorem B.5 with a=2, b=2, k=2, since $a = 2 < 4 = b^k$, we are in the top-heavy case, and $T(n) \in O(n^2)$.

A sub-sequence of a word is a word that can be obtained by deleting some letters in the word. A palindrome is a word that is the same backwards as forwards. Let $w \circ u$ represent the concatenation of two words w and u. The following recursive algorithm finds the longest palindrome that is a subsequence of the word $w[1]w[2]...w[n]$:

1. MaxPal[w[1]...w[n]];
2. If $n = 0$ return NIL ELSE:
3. If $n = 1$ return $w[1]$ ELSE :
4. If $w[1] = w[n]$ return $w[1] \circ MaxPal[w[2]...w[n-1]] \circ w[n]$ ELSE: return $Longer(MaxPal[w[2], ..w[n]], MaxPal[w[1], ..w[n-1]])$

Here *Longer* is a routine that compares two words and returns the longer word, breaking ties arbitrarily.

1. Show the recursion tree of the above algorithm on the word $EDGED$.

   On EDGED, the algorithm would call itself on $EDGE$ and $DGED$.

   On EDGE it would call itself on DG.

   On DG it calls itself on D and on G, both cases returning the one letter.

   So then backing up to EDGE it would return (if Longer picks the first in case of ties) EDE.

   On DGED it would call itself on GE, again branching to G and E so returning say DGD.

   On EDGED the algorithm finally picks either of EDE and DGD and returns it.

2. Give a bound on the worst-case number of recursive calls the algorithm could make on a word with n symbols.

   Each time the algorithm calls itself, the word is reduced by at least one symbol. Thus, the maximum depth of the tree of recursions is n. The maximum number of calls at any one step is 2, if the symbols don't match. Thus, the recursion tree is a binary tree of depth at most n, and so has at most $O(2^n)$ nodes. If all the symbols are different, this will be the actual number of calls, so no sharper bound is possible.

3. Give a polynomial-time dynammic programming version of the recurrence.

   The recursion only eliminates characters from the beginning and end of the word. If it eliminates $I-1$ characters from the beginning, and $N-J$ from the end, we get $w[I]..w[J]$ as the word that's left. So we only need to solve this for the $O(n^2)$ words of the form $w[I]..w[J]$ for $J \geq I$. Either I increases or J decreases in the recursion, so in the bottom-up version, I would decrease and J increase. We use the same recursive definition, but replace returns with writes and recursive calls with reads to an array $Pals[I,J]$, storing the best palindromic subsequence in $W[I]..w[J]$.

   $DPPal[w[1]...w[n]]$

   (a) Initialize $Pals[I:1..n; J:0..n] : array of words$

   (b) FOR $I = 1 to n$ do: $Pals[I, I-1] \leftarrow \lambda, Pals[I,I] \leftarrow w[I]$, {If no more letters are left, the empty word is only possible. If one letter left, that letter is the best possible.}

   (c) FOR $I = n-1$ to 1 do:

   (d) FOR $J = I+1$ to n do:

   (e) beginfor;

   (f) If $w[I] = w[J]$ THEN $Pals[I,J] \leftarrow w[I] \circ Pals[I+1, J-1] \circ w[J]$

(g) ELSE: $Pals[I, J] \leftarrow Longer(Pals[I+1, J], Pals[I, J-1])$

(h) end;{for}

(i) Return Pals(1,n)

4. Give a time analysis of the dynammic programming algorithm.

   The time is dominated by the second nested loop, which is constant time and executed $O(n^2)$ times. Thus, the total time is $O(n^2)$.

5. Show the array that your algorithm produces on the word $EDGED$.

   First it sets $Pals[1,1] = E, Pals[2,2] = D, Pals[3,3] = G, Pals[4,4] = E, Pals[5,5] = D$, and all the cases $J < I$ to the empty word. Then it goes through $I = 4$ and sets $Pals[4,5] = Longer(Pals(5,5), Pals(4,4)) = D$. Then for $I = 3$, it sets $Pals[3,4] = E, Pals[3,5] = D$. For $I = 2$, it sets $Pals[2,3] = G, Pals[2,4] = E$ and $Pals[2,5] = Dj \circ Pals(3,4) \circ D = DED$. Finally, it sets $Pals[1,2] = D, Pals[1,3] = G, Pals(1,4) = EDE, Pals(1,5) = Longer(DED, EDE) = DED$. This last is returned.

Greedy Algorithms and use of data structures in algorithms Consider the following problem. You are trying to schedule your course work for the next n day quarter. You have a list of m asssignments for all your courses, where for each assignment a_i, you have a day $start_i$ when the assignment will be given to you, and a day end_i, the last day you could do it before the due date. (For example, if the assignment were given out the morning of the 13th and due early the next morning, then both $start_i$ and end_i would equal 13.) You know that each assignment will take one uninterrupted day of work.

You wish to perform as many of the assignments before their due dates as possible, scheduling each assignment you will complete for a different day. To be completed, a_i must be scheduled at a day d_i with $start_i \leq d_i \leq end_i$. No two assignments can be scheduled on the same day.

Parts 1 and 2: 10 points for the proof, 5 points for counter-example Below are two greedy strategies for this problem. One produces optimal solutions, the other does not. Decide which one produces optimal solutions. Give a correctness proof for this strategy, and a counter-example for the other.

Candidate Strategy one: Schedule the assignment in order of start days, except for those moved beyond their due dates by earlier assignments. Break ties by performing the first assignment due, first. Peform each assignment on the first possible free day, given the order.

This strategy schedules the assignments by start day, even if previous assignments have made the start day irrelevant, because if we've filled up our schedule until Wednesday, whether an assignment was handed

out Monday or Tuesday doesn't matter. We'll use this flaw to give a counter-example. Say that there are three assignments $A1-3$ given out Monday and due Wednesday. Then one assignment $A4$ is given out Tuesday and due Friday. One assignment $A5$ is given out Wednesday and due Thursday. This strategy would do them in the order A1:Monday,A2: Tuesday,A3:Wednesday, A4:Thursday, A5: missed. A better schedule is to switch A5 to Thursday and do A4 on Friday, getting both done in the nick of time.

Candidate Strategy two: Schedule assignments from the first day of the quarter to the last. Each day, look at the set of assignments that are available, but not already scheduled, and where we have not missed the due date already. Schedule the one with the first due date.

This strategy produces optimal schedules. We'll prove it with a Modify-the-solution argument. The greedy decision is to do, on the first unscheduled day, the available assignment that is due first. So we show how to modify a solution that does otherwise to get one that does this without decreasing the number of completed assignments.

Lemma: Let d be the first unscheduled day an assignment is available. Let a be an assignment that is available on d and not due before d, and has the earliest due date of any such assignment. Let S' be a schedule that does not perform a on day d. Then there is a schedule S that performs a on day d and performs at least as many assignments as S' does.

Proof: Case 1: If S' does no assignment on day d, then let S be the same as S', except that it does a on d. Then S either does the same number of assignments as S' (if S' did a on a different day) or one more if S' doesn't do a.

Case 2: If S' does an assignment a' on d, and it does not perform a at all, then let S be the schedule that replaces a' on d with a on d. Then S does the same number of assignments as S'.

Case 3: If S' does an assignment a' on d and does a on day d', let S be the schedule that switches a to d and a' to d'. Then S does the same number of assignments as S'. We claim S never does an assignment after its due date or before it is available. This is true for all assignments except a and a', since it is true in S', and is true for a since a is available but not already due on d. Since S' did a' on d, a' is available on d, so by definition of a, a' is due no sooner than a. Therefore, a' is due on or after d', so doing it on d' meets the due date. Since a' was handed out before dm and d' is after d, since d is the first available day, a' is available on d'. Therefore, S is a legal schedule that does the same number of assignments as S'.

Part 3: 5 points Use the strategy you chose to show how to compute the optimal order for the following example:

A1: start day 1, due day 6; A2: start day 1, due day 3; A3: start day 2, due day 6; A4: start day 3, due day 4; A5: start day 3, due day 4; A6: start day 4, due day 6;

On day 1, A1 and A2 are available. We do A2 on day 1, since it is due first. On day 2, A1 and A3 are available, and we do either, say A1. On day 3, A3,A4,A5 are available. We do either A4 or A5, say A4. On day 4, A3 and A5 are available, and we do A5. On day 5, A3 and A6 are available, we do either, say A3. On day 6, we do A6.

Part 4: 10 points For the strategy you choose, describe an efficient algorithm that carries out the strategy. Your description should mention which data structures you use, and any pre-processing steps. Give a time analysis.

We need to keep track of the set of available assignments, and pick the one with the smallest due date. This suggests using a min-heap of assignments, ordered by due date. We want to go in order of availability time, to find the earliest day an assignment is available and insert all the assignments available that day. This suggests sorting by availablity first. This gives:

ScheduleAssignments[a[1..n]];

1. MergeSort $a[1..n]$ by $start[I]$.
2. Initialize a min-heap H of assignments ordered by $due[I]$.
3. Initialize an empty list $Schedule$ of pairs, assignment, day
4. $I \leftarrow 1$; {pointer to the first to start assignment not scheduled or already available}
5. Until $I > n$ do:
6. begin;
7. IF H is empty, THEN $CurrentDay \leftarrow start[I]$ ELSE $CurrentDay++$. {Get next day when possible to do work}
8. Until $I > n$ OR $start[I] > CurrentDay$ do: Insert(a[I]); I++ {Insert all the jobs available on current day.}
9. Until $due[J] \geq CurrentDay$ do :$(due[J], J) \leftarrow Find{-}Min; DeleteMin.$ {Get job due after CurrentDay that is first due}
10. $Schedule \leftarrow Schedule \cup \{(J, CurrentDay)\}$ {Schedule J on $CurrentDay$ }
11. end;
12. Return Schedule

We can sort in $O(n \log n)$ time. While the loops inside the main loop take variable amounts of time, we eventually insert each assignment into the heap and eventually delete each assignment from the heap. Since Inserting and Deleting take $O(\log n)$ time each, FindMin is constant time, and we do each operation n times, the total time for

all such operations is $O(n \log n)$. Thus, the algorithm takes time $O(n \log n)$ total.