Georgia Institute of Technology
College of Computing
Apr 1, 2016 Bhakta

CS 3510 **Name:** ________________________________

Test 3 **Id:** ________________________________

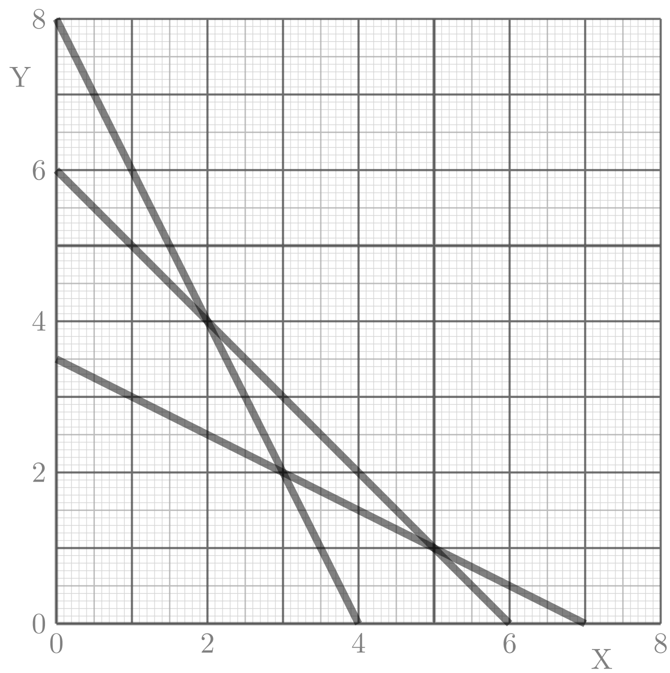
Directions: Do problem 1, and any two of problems 2-4. Mark an X below through the problem that you do not want graded. If you mark nothing, we will assume that you do not want problem 4 graded. Ask questions if you are not sure what a problem is asking.

GOOD LUCK!!

Problem	Points	Score
1	20	
2	40	
3	40	
4	40	
Bonus	10	
Total	100	

1. (20 points) Here's a simple LP in two variables, with a handy graph.

$$\begin{aligned} \max \quad & x - y \\ x, y &\geq 0 \\ y + 2x &\leq 8 \\ y + x &\leq 6 \\ 2y + x &\geq 7 \end{aligned}$$



(a) (10 points) Sketch/shade the feasible region for this LP above, and find the optimum value of the objective and the value of the variables at that optimum.

Solution: The feasible region is the quadrilateral bounded by the points (0,6), (0,3.5), (3,2), (2,4). Of these, the optimum point is $(3, 2)$ with an optimum value of 1.

(b) (10 points) Write down, but do not solve, the dual LP. (Watch the direction of that last inequality!)

Solution: To put the last equation in the right form, we multiply both sides of the inequality by -1 to get $-2y - x \leq -7$. Then taking the dual, we get

$$\begin{aligned} \min \quad & 8a + 6b - 7c \\ a, b, c &\geq 0 \\ 2a + b - c &\geq 1 \\ a + b - 2c &\geq -1 \end{aligned}$$

2. (Longest Common SubSTRING) In the longest common subSTRING (not subsequence) problem, you are given two strings, X of length n and Y of length m. You goal is to find the longest **contiguous** substring that is common to both X and Y. For example, the longest common substring of "CALORIE" and "ALOE" is "ALO" (not "ALOE"). An obvious algorithm based on ordinary search would take $O(nm \cdot \min(n, m))$ time. Design a more efficient dynamic program to find the longest common substring.

   Use the subproblem: $L(i, j)$ is the length of the longest common substring of $X[1, \cdots, i]$ and $Y[1, \cdots, j]$ that *ends at i in X and j in Y.*

   (a) (5 points) What are your base cases?

   Solution: If either i or j is 0, it makes sense that $L(i, j)$ should be 0.

   (b) (15 points) What is the recursive equation for $L(i, j)$? Feel free to use shorthand if describing minimums/maximums/sums. Give brief justification for your ideas. Don't forget to include the base cases!

   Solution:

   - If $i = 0$ or $j = 0$, $L(i, j) = 0$.
   - If $X[i] = Y[j]$, then we have a match, and we should add 1 to the longest common substring that ends at $i - 1$ in X and $J - 1$ in Y. In this case, $L(i, j) = L(i - 1, j - 1) + 1$.
   - If $X[i] \neq Y[j]$, then there can be no positive length substring in common that ends at $X[i]$ and $Y[j$. In this case, $L(i, j) = 0$.

(c) (15 points) Write an efficient dynamic program to calculate the longest common substring. You **may use either an iterative or memoized approach**. After computing all the $L(i, j)$, your algorithm should return the length of the longest common substring of X and Y, and also return the substring itself. (note: this last step is much easier than usual).

> **Solution:**
>
> **function** L(i,j)
>   Array $Soln[mxn]$
>   **for** $i = 0$ to n **do**
> $Soln[i, 0] \leftarrow 0$
>   **for** $j = 1$ to n **do**
> $Soln[0, j] \leftarrow 0$
> $BestLength \leftarrow -1$ ▷ Store the best length across all i, j
> $Besti \leftarrow null$
>   **for** $i = 1$ to n **do** ▷ Main Recursive Part
>   **for** $j = 1$ to n **do**
>   **if** $X[i] = Y[j]$ **then**
> $Soln[i, j] \leftarrow Soln[i - 1, j - 1] + 1$
>   **else**
> $Soln[i, j] \leftarrow 0$
>   **if** $Soln[i, j] > BestLength$ **then**
> $BestLength \leftarrow Soln[i, j]$
> $Besti \leftarrow i$
>   **return** $BestLength$, $X[Besti - BestLength + 1 : Besti + 1]$ ▷ This last bit might vary depending on how you are indexing your strings

(d) (5 points) Analyze the final runtime of your algorithm.

> **Solution:** There are $O(mn)$ sub problems, and we do $O(1)$ work per subproblem, for a total of $O(mn)$ work.

3. (Resource Acquisition) You need T tons of avocados for your guacamole festival. There are n farms, and each farm i sells avocados in w_i-ton crates, for c_i dollars per crate.

   Give a dynamic program to find the cheapest way to buy *at least* T tons of avocados. You can buy more than one crate from each farm, but you must buy whole crates. Your final algorithm should be polynomial in n and T.

   (a) (10 points) Clearly define the meaning of your subproblems. What are your base cases?

   > **Solution:** Let $Cost(x, i)$ represent the least cost way to buy x tons from the first i farms. Then, if $x = 0$, the least cost should be 0. If $i = 0$ and $x > 0$, it's unclear what this should mean. We'll put ∞, since this represents a situation that we don't want to be in (where we still have avocados to buy, but no farm to buy from).

   (b) (20 points) What is the recursive solution? Feel free to use shorthand if describing minimums/maximums/sums. Give some justification for your ideas. Don't forget to include the base cases!

   > **Solution:** Think about whether or not to buy a crate from farm i.
   >
   > - If we don't buy a crate from farm i, we have $Cost(x, i) = Cost(x, i - 1)$.
   > - If we do buy a crate from farm i, we spend c_i dollars, and have w_i tons less to buy. We may still want to buy more from farm i. If $x \leq w_i$, then we don't need to buy any more avocados (which can also be done by setting the tons to buy to 0. Thus, $Cost(x, i) = c_i + Cost(\max(x - w_i, 0), i)$.

   (c) (10 points) Analyze the final runtime of your algorithm, if it were implemented as an efficient dynamic program.

   > **Solution:** We have nT total subproblems, and do $O(1)$ work per problem, for a total nT runtime.

4. You are flipping n independent coins, and each coin i has it's own probability $0 \leq p_i \leq 1$ of coming up heads. Design an $O(nk)$ dynamic programming algorithm to compute the probability that **exactly** k of these coins land heads. Assume all arithmetic operations are $O(1)$.

   Use the subproblem: $EXACT(i, h)$ is the probability that exactly h of the first i coins are heads.

   (a) (5 points) What are your base cases for $EXACT$?

   > **Solution:** It makes sense that $EXACT(0, 0) = 1$. If $h < 0$ or if $i < h$, then $EXACT(i, h) = 0$, as these are impossible. Note, depending on how you express the recursion, you may have a different set of base cases.

   (b) (15 points) What is the recurrence relation for $EXACT(i, h)$? Feel free to use shorthand if describing minimums/maximums/sums. Give some justification for your ideas, and explain the final running time. Hint: Think about coin i. Don't forget to include the base cases! Keep in mind that we want the final runtime to be $O(nk)$.

   > **Solution:** If $i = h = 0$, $EXACT(i, h) = 0$. If $i < h$, then $EXACT(i, h) = 0$. Otherwise, look at the ith coin. There are two disjoint ways for h of the first i coins to be heads, those where the ith coin is heads, and those where it is tails. If the ith coin is heads, then $h - 1$ of the first $i - 1$ coins must be heads, and if the ith coins is tails, then h of the first $i - 1$ coins must be heads. Thus the total probability is $EXACT(i, h) = p_i \cdot EXACT(i - 1, h - 1) + (1 - p_i) \cdot EXACT(i - 1, h)$.

   Now instead consider the different problem of finding the probability that *at least* k of the n coins land heads.

   Use the subproblem: $ATLEAST(i, h)$ is the probability that at least h of the first i coins are heads. We still want an $O(nk)$ algorithm for this problem.

   (c) (5 points) What are the new values of the base cases?

   > **Solution:** It still makes sense that if $i < h$, then $ATLEAST(i, h) = 0$. But now, if $h < 0$, it makes sense to return 1. In fact, we can extend this base case over to the next line, and say that if $h \leq 0$, $ATLEAST(i, h) = 1$.

   (d) (15 points) What is the recurrence relation for $ATLEAST(i, j)$? What changed between the solution for $ATLEAST$ and $EXACT$? Don't forget to include the base cases!

   > **Solution:** If $h = 0$, $ATLEAST(i, h) = 1$. If $i < h$, then $ATLEAST(i, h) = 0$. Otherwise, look at the ith coin. There are two disjoint ways for at least h of the first i coins to be heads, those where the ith coin is heads, and those where it is tails. If the ith coin is heads, then at least $h-1$ of the first $i-1$ coins must be heads, and if the ith coins is tails, then at least h of the first $i - 1$ coins must be heads. Thus the total probability is $ATLEAST(i, h) = p_i \cdot ATLEAST(i - 1, h - 1) + (1 - p_i) \cdot ATLEAST(i - 1, h)$.
   >
   > But wait, this is the same recurrence as in part b! What changed? The only difference is one thing - the values of the base cases.

5. (10 points) (bonus) Recall that an *independent set* of a graph $G = (V, E)$ is a subset of the vertices $S \subset V$ such that no two vertices in the set share an edge. In class, you saw a dynamic program to find an independent set with maximum total weight in a vertex-weighted tree.

   Given a vertex-weighted *binary* tree T (a tree where every vertex v has an associated weight w_v and at most 2 children), and an integer k, give an algorithm that will return the max weight of an independent set in T that has *exactly* k vertices, among all independent sets in T with exactly k vertices, or *null* if no such set exists.

   (a) Clearly define the meaning of your subproblems. What are your base cases?

   Solution: Let $MIS(v, i, b)$ be the weight of the maximum independent set of size i in the subtree rooted at v. If $b = 1$ then we are including v, else we are not including v. For any v, $MIS(v, 0, 0) = 0$. For a leaf node $MIS(v, 1, 1) = w_v$.
   Depending on your implementation, it may or may not be convenient for you to set $MIS(v, i, b) = -\infty$ for any invalid conditions, like if v is a leaf node and $i \neq b$, or if v is any vertex and $b = 1 > i = 0$.

   (b) What is the recursive solution? Feel free to use shorthand if describing minimums/maximums/sums. Give some justification for your ideas. Don't forget to include the base cases!

   Solution: To find $MIS(v, i, 1)$, the children of v must not be included, and the sum of the vertices used by the children must sum to $i - 1$.

   - If v is a leaf, then $MIS(v, i, 1) = w_v$ if $i = 1$, and $-\infty$ if $i \neq 1$.
   - If v has one child x, then $MIS(v, i, 1) = w_v + MIS(x, i - 1, 0)$.
   - If v has two children x_1, x_2, then

   $$MIS(v, i, 1) = w_v + \max_{k:0 \leq k \leq i-1} [MIS(x_1, k, 0) + MIS(x_2, i - 1 - k, 0)]$$

   To find $MIS(v, i, 0)$, the children of v may or may not be included, and the sum of the vertices used by the children must sum to i. So similarly,

   - If v is a leaf, then $MIS(v, i, 0) = 0$ if $i = 0$, and $-\infty$ if $i \neq 0$.
   - If v has one child x, then $MIS(v, i, 0) = \max(MIS(x, i, 0), MIS(x, i, 1))$.
   - If v has two children x_1, x_2, then

   $$MIS(v, i, 0) = \max_{k:0 \leq k \leq i-1} [\max(MIS(x_1, k, 0), MIS(x_1, k, 1)) + \max(MIS(x_2, i-k, 0), MIS(x_2, i-k$$

   Like before, we will choose the larger of these two.

   (c) Analyze the runtime of your final algorithm, if it were implemented as an efficient dynamic program. A faster and more correct algorithm is worth more points.

Solution: We have $O(kn)$ subproblems, and do $O(k)$ work per step, for a total $O(k^2n)$ algorithm.