CS 6150: HW2 – Dynamic Programming, Greedy Algorithms

HINTS / SOLUTIONS

1. (5 points) Suppose we have n different coins, and suppose the ith coin has a probability p_i of turning up heads, when tossed. Now, say we toss all the coins, and we are interested in the event that we obtain precisely k heads. Design an algorithm that computes the probability of this event in $O(n^2)$ time. [For this problem, you may assume that multiplying any two numbers in the interval $(0, 1)$ takes $O(1)$ time.] [Source: Dasgupta et al. textbook]

   Let $P[r, m]$ denote the probability that we get precisely r heads after tossing coins $1, 2, \ldots, m$. We can fill out $P[r, m]$ by proceeding in the order of increasing r, and for each r, in the order of increasing m.

   The key recurrence is: $P[r, m] = p_m P[r-1, m-1] + (1-p_m)P[r, m-1]$. The base cases are: $P[r, m] = 0$ for $r > m$, and $P[0, m] = (1-p_1)(1-p_2)\ldots(1-p_m)$ for all m.

   The running time is $O(nk)$, which is also the memory footprint.

2. (5 points) Suppose we are typing on an old keyboard with a broken space key. Given a string, there can now be several ways to break it up into component words, e.g., TENTENDING, could either have been TEN TEN DING, or TENT ENDING. Given a string of length N, and a collection of all the words in the language (say there are m of them, each of length at most L), design an algorithm that counts the number of possible ways to break the string into words. What is the run time of this algorithm?

   Suppose we have a look-up method for the words in the dictionary (this can be implemented in time $O(L)$, but we could even assume we have a trivial one, that takes time $O(mL)$).

   Let $s[1, 2, \ldots, N]$ denote the input string. Let $A[m]$ denote the number of possible ways to split the substring $s[m, m+1, \ldots, N]$. We are interested in computing $A[1]$.

   We can compute $A[m]$ values, starting from the right. To compute $A[m]$, we scan the string $s[m, m+1, \ldots]$, for lengths 1 through ℓ. Let χ_ℓ be an indicator that is 1 if $s[m, m+1, \ldots, m+\ell]$ is a valid word. Then, we have

   $$A[m] = \sum_{\ell=0}^{L} \chi_\ell A[m + \ell + 1].$$

   The base case is simply $A[N+1] = 1$. The naive implementation, with an $O(mL)$ dictionary lookup takes: $O(N \times L \times mL) = O(NmL^2)$.

3. Suppose a burglar breaks into a house, and realizes the house has far too many items to fit into his sack. He thus wishes to pick the items of most value that he can end up taking.

   (a) (2 points) Not being trained in algorithms, he starts stuffing items into his sack, the most valuable first. He stops when none of the items left can fit into his back. Give an example in which this strategy is not optimal.

   This is bad when the most valuable items take too much space. Here's a simple example of (value, size) pairs that gives a counter-eg: $(5, 5), (3, 2), (3, 2), (3, 1)$. If the sack capacity is 5, he would have got stuck after picking the first object (val 5), while he could have got a total value 9 by picking the last three.

(b) (2 points) After obvious disappointment at his performance, the robber decides to empty his sack and pick items in decreasing order of the *ratio* value/size. Give an example in which even this strategy is not optimal.

Intuitively, this strategy starts off well, but as the leftover space drops, we might be led to a suboptimal solution.
Here's a simple set of (val, size) pairs: $(4,4), (4,4), (6,5), (0,3)$, and capacity 8. The greedy choice picks the third object first, and then we're stuck with the zero-value object.

Lost in thought, the robber loses track of time and gets caught. To dissuade future (smarter) robbers, we now wish to take the most valuable items and move them to a bank locker. Unfortunately, we find ourselves in the same situation.

(c) (5 points) Suppose the locker has size S, and suppose the house has n items, with item i having value v_i and size s_i, both of which are positive integers. Design an algorithm with running time $O(nS)$ that finds the "most valuable" collection of items (i.e., max total value), and total size $\leq S$.

For each size $t \leq S$, let $V[t]$ denote the max value we can get for a locker of size $\leq t$. We have the recurrence:

$$V[t] = \max\{0, \max_{i:t \geq s_i} \{V[t - s_i] + v_i\}\}.$$

We can thus compute the $V[]$ values in the increasing order for t, with the base case $V[0] = 0$.

(d) (1 point) Note that the *input* for the problem consists of S, and the values v_i, s_i, for $1 \leq i \leq n$. Thus the space necessary to write down the input is $O(\log S + \sum_{i=1}^{n} \log v_i + \log s_i)$. Answer YES/NO, with justification: Is the dynamic programming algorithm above polynomial in the input size?

YES, the space bound is correct, because writing down any integer n in binary takes $\log_2 n$ space. Since algorithm takes time $O(nS)$, it is not poly in the input size, which only has a $\log S$ dependence in terms of S. Thus when, say $S = 2^{n^2}$, the run time is hopelessly exponential, but the input size is only polynomial in n.

4. (10 points) Let T be a rooted binary tree with n vertices, and let $1 \leq k \leq n$ be an integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. Formally, given a set S of marked vertices, we define $cost_S(v)$ for any $v \in T$ to be the distance (i.e., the number of hops) to the closest ancestor of v that is in S. (If no such ancestor exists, the cost is ∞, and if $v \in S$, the cost is zero.)

The "loss" of a set S is defined to be $\max_{v \in T} cost_S(v)$. Design an algorithm with $O(n^2k^2)$ running time that finds a set S of size k that minimizes the loss.
[Source: Jeff Erickson's Exercises]

The idea is to start with the question: suppose we are at the root of T and we want to deduce the answer by recursing on the left and right subtrees, what "parameters" should the sub-trees know? A bit of thought reveals that they only need to know how many marked nodes to pick in that subtree. However, to compute the objective value, the root of the subtree needs to know what its *cost*() value would be. This motivates the following definitions:

For a node v, let T_v denote the sub-tree rooted at v.

Let $C[v, m, d]$ denote the objective value in the sub-tree T_v, given that: (a) we are supposed to pick m marked nodes in T_v, and (b) the cost() value for the root v assuming it is *not picked* is d (we may choose to pick v, in which case its cost will become 0).

Then, the final answer we want is $C[\text{root}, k, \infty]$.

We can write a recurrence for $C[v, m, d]$ as follows: let v be a node with left and right children ℓ, r. There are two choices: we can either mark v or not.

Case 1: we mark v; then we have $m - 1$ marks left, which we have to split between the left and right subtrees. Say we split as x, $m - 1 - x$. Then the value of the objective is

$\max\{0, C[\ell, x, 1], C[r, m-1-x, 1]\}$. [The 0 is because we marked v.] The best choice in Case 1 is to pick the x that gives the smallest value for this quantity.

Case 2: we do not mark v; then we have m marks left, which we have to split. Say we split as $x,\ m-x$. Then the value of the objective is $\max\{d, C[\ell, x, d+1], C[r, m-x, d+1]\}$.

Overall, we pick the better of cases 1 and 2 (one with the smaller objecive value – unless $m = 0$, in which case we only have case 2). The base cases involve the leaves of the tree, for which we have $C[v, 0, d] = d$ and $C[v, m, d] = 0$ for all $m \geq 1$.

Running time: we compute $C[v, m, d]$, where $v \in T$, $m \in \{0, 1, \ldots, k\}$, and d ranges from 0 to the height of the tree (plus the single value infinity). Computing each such value takes time $O(k)$ (case 1 and case 2 both take k lookups). Thus the overall time is $O(n^2k^2)$ (using the trivial bound of n on the height of the tree).

5. In class, we discussed the Longest Increasing Subsequence (LIS) problem, and an algorithm based on dynamic programming with running time $O(n^2)$ (where n is the size of the input array). Let us now see how to reduce the running time to $O(n \log n)$. Let the input array be $A[0, 1, \ldots, n-1]$, and suppose the entries are *all distinct*.

   Now, let $L[i]$ denote the length of the longest increasing subsequence *starting at* position i. The idea was to start with $i = n-1$, move left, and compute the $L[i]$ values in that order. The simple algorithm we saw takes $O(n)$ time to compute the value of $L[i]$ given the values $L[j]$ for all $j > i$. The question is if we can do better.

   As we move from right to left, suppose we maintain additional array $B[]$ with the following property: the jth element in B is the value of the *largest* $A[i]$ in the array we have seen so far with the property that $L[i] \geq j$. I.e., it is the largest entry in the array seen so far that has an increasing subsequence of length j starting with itself.

   (a) (4 points) Prove that the entries of B are strictly decreasing.

   Consider any step of the algorithm (say we are computing $L[m]$). $B[j]$ is the largest $A[i]$, with $i \geq m$, that has an increasing sequence of length j starting with it. Let us claim that $B[j] > B[j+1]$ for all j for which the numbers are both well-defined. Look at the increasing sequence of length $j+1$ starting at $B[j+1]$; the next element in that sequence has a value $> B[j+1]$ (because all entries of A are distinct), and it has an increasing sequence of length j starting with it. $B[j]$ is at least this value, which proves the claim.

   (b) (6 points) As we move from right to left, show how we can update this array maintaining the property above, and complete the algorithm for computing the LIS. Prove that its running time is $O(n \log n)$ overall.

   The algorithm is as follows: set B to be the empty array. Now scan the elements of $A[]$ from right to left. Suppose we are at $A[m]$. Find the indices $j, j+1$ in B such that $B[j] < A[m] < B[j+1]$ (we don't have any equalities as A has distinct entries). Replace $B[j+1]$ with $A[m]$. If $A[m]$ is smaller than all the elements currently in $B[]$, add $A[m]$ to the end of B. The final output is the length of B.

   First, note that for each m, we can perform the update in $O(\log n)$ time as follows: we store B as a balanced binary tree that stores the *pair* $(B[j], j)$. The "sort key" is $B[j]$. Every step, we either change precisely one pair, or we add a new pair to collection. Hence, this can be done in time $O(\log n)$.

   Why is the algorithm correct? Consider how the B (as per the definition) changes as we scan $A[m]$. Suppose $B[j] < A[m] < B[j+1]$. None of the entries $B[i]$, $i \leq j$ can change (because those sequences already started with numbers bigger than j). $B[j+1]$ definitely changes to $A[m]$, as there is now a sequence of length $j+1$ that starts with $A[m]$ that is bigger than the current $B[j+1]$. The rest of the entries have to remain the same, because the only new sequences we can possibly consider are ones that start with $A[m]$, and we have already argued that the *longest* sequence we can have starting at $A[m]$ is of length $j+1$. (Thus it cannot affect the later entries of B.)

6. Consider Santa Claus' dilemma: there are n gifts, and n children, and each child has a non-negative value for each gift. Formally, the value of gift j to child i is given by $A_{i,j} (\geq 0)$. Santa's goal is to give one gift to each child, so as to maximize the *total value*. Formally, if child i gets the $\pi(i)$'th gift, then the total value is $\sum_{i=1}^{n} A_{i,\pi(i)}$.

   (a) (4 points) Suppose Santa is in a real hurry, and he assigns gifts greedily. He starts with the children in order $(1, 2, \dots)$, and for each child, gives him/her the most valuable gift among the ones remaining.

   Prove that this greedy strategy can be really bad. Concretely, give a setting in which the assignment produced by the greedy strategy is a factor 1000 worse than the *best* assignment for that setting.

   Consider two children $1, 2$, two gifts X, Y. Child-1's valuations for X, Y are $1, 2$ resp. Suppose Child-2's valuations are $1, 3000$. Because Santa started with Child-1, his assignment will have total value $2 + 1 = 3$. The optimal assignment has value $1 + 3000$, which is a 1000 factor larger.

   (b) (6 points) Suppose Santa realizes this, and decides to use local search: he starts with some assignment in his mind, and for every pair of children, he checks to see if swapping their gifts can improve the total value. If so, he swaps, and continues this process until no such swap improves the value.

   Prove that for any setting of the A_{ij}'s, the solution obtained in the end has a value at least $(1/2)$ the total value of the optimum for that setting.

   Consider some solution Santa ended up at. By re-numbering the gifts, we may assume that in this solution, child 1 was assigned gift 1, child 2 assigned gift 2, etc. Now consider any 'optimal' assignment, and let it be $(1, \pi(1)), (2, \pi(2)), \dots, (n, \pi(n))$, for some permutation π. Now in Santa's assignment, consider re-assigning gift $\pi(1)$ to child 1, and gift 1 to child $\pi(1)$. Since the assignment was optimal w.r.t. pairwise swaps, we have

   $$A_{11} + A_{\pi(1)\pi(1)} \geq A_{1\pi(1)} + A_{\pi(1)1} \geq A_{1\pi(1)}.$$

   The last ineq is becaue all A_{ij} are non-negative. Now, we can write this for all $2, 3, \dots,$ and sum up, to get:

   $$\sum_i A_{ii} + \sum_i A_{\pi(i)\pi(i)} \geq \sum_i A_{i\pi(i)}.$$

   Since π is a permutation, the two summations on the LHS are equal (and equal to the value of Santa's solution). The RHS is precisely the total val of the optimum assignment. This shows the desired guarantee.

Bonus [10]: Suppose Santa does a more careful local search, this time picking every *triple* of children, and seeing if there is a reassignment of gifts that can make them happier. Prove that the final solution has a value that is at least $(2/3)$ times the optimum. [This kind of a trade-off is typical in local search – each iteration is now more expensive $O(n^3)$ instead of $O(n^2)$, but the approximation ratio is better.]

Bit more tricky; See instructor/TAs for details.

Note. We will see in the coming lectures that this problem can indeed be solved (optimally) in polynomial time, using a rather different algorithm.