CS 170 HW 7

Due on 2018-10-14, at 9:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★★) Copper Pipes

Bubbles has a copper pipe of length n inches and an array of nonnegative integers that contains prices of all pieces of size smaller than n. He wants to find the maximum value he can make by cutting up the pipe and selling the pieces. For example, if length of the pipe is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6).

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

Give a dynamic programming algorithm so Bubbles can find the maximum obtainable value given any pipe length and set of prices. Clearly describe your algorithm, prove its correctness and runtime.

Solution: Main idea: We create a recursive formula, where for each subproblem of length k we choose the cut-length i such that $Price(i) + Value(k-i)$ is maximized. Here $Price(i)$ is the price of selling the full pipe of length i and $Value(k-i)$ is the amount obtained after optimally cutting the pipe of length $k-i$.

```
def cutPipe(price, n):
    val = [0 for x in range(n+1)]
    val[0] = 0

    # Build the table val[] in bottom up manner and return
    # the last entry from the table
    for i from 1 to n:
        max_val = 0
        for j from 0 to i-1:
             max_val = max(max_val, price[j] + val[i-j-1])
        val[i] = max_val

    return val[n]
```

Proof: An inductive proof on the length of the pipe will show that our solution is correct. We let $cutPipe(n)$ represent the optimal solution for a pipe of length n. Base case: If the pipe is length 1, $cutPipe(1) = Price(1) = Val(1)$. Inductive: Assume the optimal price is found for all pipes of length less than or equal to k. If the first cut the algorithm makes x_1 is not optimal, then there is an x_1' such that $Val((k+1)-x_1)+Price(x_1) < Val((k+1)-x_1')+$

$Price(x_1')$. By the induction hypothesis, this implies that $cutPipe((k+1)-x_1)+Price(x_1) < cutPipe((k+1)-x_1')+Price(x_1')$. So the algorithm must have chosen x_1' instead of x_1, by construction (a contradiction). Therefore, by induction $cutPipe(n) = Val(n)$ for all $n > 0$.

Run-time: The algorithm contains two nested for-loops resulting in a run-time of $O(n^2)$.

3 (★★★) A Dice Game

Consider the following 2-player game played with a 6-sided die. On your turn, you can decide either to roll the die or to pass. If you roll the die and get a 1, your turn immediately ends and you get 1 point. If you instead get some other number, it gets added to a running total and your turn continues (i.e. you can again decide whether to roll or pass). If you pass, then you get either 1 point or the running total number of points, whichever is larger, and it becomes your opponent's turn. For example, if you roll 3, 4, 1 you get only 1 point, but if you roll 3, 4, 2 and then decide to pass you get 9 points. The first player to get to N points wins, for some positive N.

Alice and Bob are playing the above game. Let $W(x, y, z)$ be the probability that Alice wins given that it is currently Alice's turn, Alice's score (in the bank) is x, Bob's score is y and Alice's running total is z.

(a) Give a recursive formula for the winning probability $W(x, y, z)$.

(b) Based on the recursive formula you gave in the previous part, design an $O(N^3)$ dynamic programming algorithm to compute $W(x, y, z)$. Briefly describe your algorithm, prove its correctness and runtime.

Solution:

(a) *Hint if students struggle:* Work out the probabilities R and P that the current player will win if they decide to roll and pass respectively.

If the current player rolls and gets a 1, they'll win with probability $1 - W(y, x+1, 0)$. If they roll and get a different value v, then v gets added to the running total and it's still their turn, so they'll win with probability $W(x, y, z+v)$. Since each value of the die has probability $1/6$, this means

$$R = \frac{1}{6}\left(1 - W(y, x+1, 0)\right) + \frac{1}{6}\sum_{v=2}^{6} W(x, y, z+v).$$

If instead they pass, they get $\max(1, z)$ points and it becomes their opponent's turn. So we have

$$P = 1 - W(y, x + \max(1, z), 0).$$

Finally, $W(x, y, z) = \max(R, P)$, and substituting the expressions above gives a recursive formula for W.

(b) We convert the recursive formula above into a recursive algorithm. The base cases are as indicated in part (1): when a player has a large enough running total to win immediately by passing. Note that in the recursive formula, every recursive term either increases the

number of points one of the players has, or increases the running total. So the recursion is guaranteed to terminate. Using memoization to keep track of the values of W we have already computed gives a dynamic programming algorithm.

To work out the runtime of this algorithm when you need N points to win, notice that we never need to compute $W(x, y, z)$ where any of x, y, or z is $N + 6$ or larger. This is because if either player had that many points they would have already won on the previous turn, and if z was that large the current player could have won before the last die roll by passing. So the algorithm will compute W for at most $(N+6)^3$ different game positions, and therefore its runtime is $O(N^3)$.

4 (★★★) Road Trip

Suppose you want to drive from San Francisco to New York City on I-80. Your car holds C gallons of gas and gets m miles to the gallon. You are handed a list of the n gas stations that are on I-80 and the price that they sell gas. Let d_i be the distance of the i^{th} gas station from SF, and let c_i be the cost of gasoline at the i^{th} station. Furthermore, you can assume that for any two stations i and j, the distance $|d_i - d_j|$ between them is divisible by m. You start out with an empty tank at station 1. Your final destination is gas station n. You may not run out of gas between stations but you need not fill up when you stop at a station, for example, you might to decide to purchase only 1 gallon at a given station.

Find a polynomial-time dynamic programming algorithm to output the minimum gas bill to cross the country. Clearly describe your algorithm and prove its correctness. Analyze the running time of your algorithm in terms of n and C. You do not need to find the most efficient algorithm, as long as your solution's running time is polynomial in n and C.

Solution:

Main Idea: Let $M^i(g)$ be the minimum gas bill to reach gas station i with g gallons of gas in the tank (after potentially purchasing gas at station i). The range of the indices is $1 \leq i \leq n$ and $0 \leq g \leq C$.

The recursive equation will be written in terms of the number of gallons of gas in the car when leaving station $i-1$. Call this number h. Clearly $(d_i - d_{i-1})/m \leq h \leq C$ otherwise the car cannot reach station i. Also $h \leq (d_i - d_{i-1})/m + g$ because we cannot purchase a negative number of gallons at station i. The recursive equation is

$$M^i(g) = \min_h [M^{i-1}(h) + (g + (d_i - d_{i-1})/m - h)c_i]$$

where h runs from $(d_i - d_{i-1})/m$ to $\min(C, (d_i - d_{i-1})/m + g)$. The base case is

$$M^1(g) = c_1 g \quad \text{where } 0 \leq g \leq C.$$

The answer will be given by $\min_{g=0}^{C}(M^n(g))$. One can argue that the cheapest solution will involve arriving at gas station n with 0 gallons in the tank, so the answer is also simply the entry $M^n(0)$. We choose to evaluate the matrix in increasing order of i. Note that to compute M^i we only need M^{i-1}, so the space can be reused. This is demonstrated in the pseudo-code, which uses only two arrays M and N.

```
GasolineRefilling(n, d[], c[]) {
    for g from 0 to C
        M[g] = c[1]*g     // base case
    for i from 2 to n {
        for g from 0 to C {
            N[g] = infinity    // N is a temporary array
            for h from (d[i] - d[i-1])/m to min(C, (d[i] - d[i-1])/m + g) {
                cost = M[h] + (g + (d[i] - d[i-1])/m - h)*c[i]
                if (cost < N[g])
                    N[g] = cost;
            }
        }
        for g from 0 to C
            M[g] = N[g]        // copy entries from the temporary array
    }
    return M[0]
}
```

Proof: The algorithm considers all possible numbers of gallons we can purchase at each station along with all possible amounts of gas we can have when arriving at each station. By induction on n, we can see it finds the best possible amount to purchase at each station.

Runtime: There are 3 nested for-loops, one ranging over $n-1$ values, one ranging over $C+1$ values, and one ranging over at most C values. So the running time is $O(nC^2)$.

Aside: Because the distances between cities are divisible by m, it's possible to argue that one cannot achieve better by purchasing fractions of gallons, and so the integer solution found is really the best possible.

5 (★★★★) Non-Prefix Code

As we have learned in lecture, the Huffman code satisfies the *Prefix Property*, which states that the bit string representing each symbol is not a prefix of the bit string representing any other symbol. One nice property of such codes is that, given a bit string, there is at most one way to decode it back to a sequence of symbols. However, this is not true anymore once we are working with codes that do not satisfy the Prefix Property. For example, consider the code that maps A to 1, B to 01 and C to 101. A bit string 101 can be interpreted in two ways: as C or as AB.

Your task is to, given a bit string s, determine how many ways one can interpret s. The mapping from symbols to bit strings of the code will be given to you as a dictionary d (e.g., in the example, $d = \{A : 1, B : 01, C : 101\}$); you may assume that you can access each symbol in the dictionary in constant time. Your algorithm should run in time at most $O(nm\ell)$ where n is the length of the input bit string s, m is the number of symbols, and ℓ is an upper bound on the length of the bit strings representing symbols.

Clearly describe your algorithm, prove its correctness and runtime.

Solution:

Main Idea: We define our subproblems as follows: let $A[i]$ be the number of ways of interpreting the string $s[:i]$. We can then compute $A[i]$ using the values of $A[j], j < i$ via the following recurrence relation:

$$A[i] = \sum_{\substack{\text{symbol } a \text{ in } d \\ s[i-\text{length}(d[a])+1:i]=d[a]}} A[i - \text{length}(d[a])].$$

Note here that we set $A[0] = 1$. Our algorithm simply computes the above formula in a trivial manner.

Pseudocode:

procedure TRANSLATE(s):
    Create an array A of length $n+1$ and initialize all entries with zeros.
    Let $A[0] = 1$
    for $i := 1$ to n **do**
        for each symbol a in d **do**
            if $i \geq \text{length}(d[a])$ and $d[a] = s[i - \text{length}(d[a]) + 1 : i]$ **then**
                $A[i] += A[i - \text{length}(d[a])]$
    return $A[n]$

Proof of Correctness: We can show this via a simple induction argument.

Base Case. When $i = 0$, there is only one way to interpret $s[:0]$ (the empty string). Hence, $A[0] = 1$.

Inductive Step. Suppose that $A[0], \ldots, A[i-1]$ contains the right value. We will show that the above recurrence relation gives the right value for $A[i]$. To do this, we partition interpretations of $s[:i]$ as a sequence of symbols $a_1 \ldots a_k$ based on the ending symbol a_k. For $a_k = a$, if the suffix of $s[:i]$ coincides with $d[a]$, every interpretation $a_1 \ldots a_k$ has a one-to-one correspondence with an interpretation $a_1 \ldots a_{k-1}$ of $s[:i - \text{length}(d[a])]$. From our inductive hypothesis, there are exactly $A[i - \text{length}(d[a])]$ of the latter. On the other hand, if the suffix of $s[:i]$ differs from $d[a]$, then there is no interpretation of $s[:i]$ ending with symbol a. Summing this up over all symbols a's implies that our recurrence relation yields the right value for $A[i]$.

Finally, note that our program below implements this recurrence in a straightforward way, so the output of our program is indeed $A[n]$, the number of ways to interpret s.

Runtime Analysis: There are n iterations of the outer for loop and m iterations of the inner for loop. Inside each of these loops, checking that the two strings are equal takes $O(\text{length}(d[a])) \leq O(\ell)$ time. Hence, the total running time is $O(nm\ell)$.

Note that it is possible to speed up the algorithm running time to $O((n+m)\ell)$ using a trie instead of reconstructing the string every time, but this is not required to receive full credit for the problem.

6 (★) A HeLPful Introduction

Find necessary and sufficient conditions on real numbers a and b under which the linear program

$$\begin{aligned} \max\ & x + y \\ & ax + by \leq 1 \\ & x, y \geq 0 \end{aligned}$$

(a) Is infeasible.

(b) Is unbounded.

(c) Has a unique optimal solution.

Solution:

(a) Note that for any possible real valued a and b, the point $x = 0, y = 0$ is always feasible (since $a \cdot 0 + b \cdot 0 = 0 \leq 1$). Thus, the program is *not* infeasible for any choice of a and b.

(b) If the LP is unbounded, then the dual LP cannot be feasible (since, otherwise, by weak duality, the feasible solution for the dual will give an upper bound for the primal objective). Next, if an LP is bounded, then by the duality theorem, its (bounded) optimum is the same as that of its dual, which must therefore be feasible. Thus, this LP is unbounded if and only if its dual is infeasible.

We therefore write the dual of the original LP

$$\begin{aligned} \min\ & z \\ & az \geq 1 \\ & bz \geq 1 \\ & z \geq 0 \end{aligned}$$

Suppose $a > 0$ and $b > 0$. In this case, $z = \max(1/a, 1/b)$ is a feasible solution for the dual. Thus, at least one of a, b must be non-positive for the dual to be infeasible. Suppose $b \leq 0$. Then we have $bz \leq 0 < 1$ for $z \geq 0$, so that the dual is infeasible. Similarly when $a \leq 0$ the dual is infeasible. We conclude that the dual is infeasible if and only if $a \leq 0$ or $b \leq 0$. Thus, from the discussion above, the original LP is unbounded if and only if at least one of a and b is non-positive.

(c) From part (b), we see that the LP has a bounded optimum if and only if $a > 0$ and $b > 0$. In this case, the feasible region is bounded by the lines $x = 0$ and $y = 0$ and $ax + by = 1$, so that its vertices are $(0, 0), (1/a, 0)$ and $(0, 1/b)$. Since $a, b > 0$, the optimum is $\max(1/a, 1/b)$, and is attained at one of the last two vertices.

Since every point inside the feasible region of an LP can be written down as a convex combination of vertices, it follows that the optimum is non-unique if and only if it is attained at at least two vertices. Since the only two vertices where the optimum can be attained in our LP are $(0, 1/b)$ and $(1/a, 0)$, we will have a unique optimum as long as

the objective values at these points, $1/a$ and $1/b$, are different. We therefore get that the LP has a unique bounded optimum if and only if

$$a > 0, b > 0 \text{ and } a \neq b.$$

7 (★★) Spaceship

A spaceship uses some *oxidizer* units that produce oxygen for three different compartments. However, these units have some failure probabilities.

Because of differing requirements for the three compartments, the units needed for each have somewhat different characteristics.

A decision must now be made on just *how many* units to provide for each compartment, taking into account design limitations on the *total* amount of *space*, *weight* and *cost* that can be allocated to these units for the entire ship. Specifically, the total space for all units in the spaceship should not exceed 500 cubic inches, the total weight should not exceed 200 lbs and the total cost should not exceed $400,000$ dollars.

The following table summarizes the characteristics of units for each compartment and also the total limitation:

	Space (cu in.)	Weight (lb)	Cost ($)	Probability of failure
Units for compartment 1	40	15	30,000	0.30
Units for compartment 2	50	20	35,000	0.40
Units for compartment 3	30	10	25,000	0.20
Limitation	500	200	400,000	

The objective is to *minimize the probability* of all units failing in all three compartments, subject to the above limitations and the further restriction that each compartment have a probability of no more than 0.05 that all its units fail.

Formulate the *integer programming model* for this problem. An integer programming model is the same as a linear programming model with the added functionality that variables can be forced to be integers.

Side note: Integer programming is often intractable, so we use a linear program as a heuristic. We can take out the integrality constraints and change our model into a linear program. We then round the solution and make sure none of constraints have been violated (you should think about why this won't always give us the optimum)

Solution: Let x_1 be the number of oxidizer units provided for compartment 1, and define similarly x_2 for compartment 2 and x_3 for compartment 3. The probability that all units fail in compartment 1 is $(0.3)^{x_1}$, for compartment 2 is $(0.4)^{x_2}$ and for compartment 3 is $(0.2)^{x_3}$. The probability that all units fail in all compartments is $(0.3)^{x_1}(0.4)^{x_2}(0.2)^{x_3}$. The exponential function is non-linear, so we take logarithm (which preserves ordering of

numbers) to get the linear programme

$$\text{Minimize } \log(0.3)x_1 + \log(0.4)x_2 + \log(0.2)x_3$$

$$\text{subject to } \begin{cases} 40x_1 + 50x_2 + 30x_3 \leq 500 & \text{space constraint (cu in.)} \\ 15x_1 + 20x_2 + 10x_3 \leq 200 & \text{weight constraint (lb)} \\ 30x_1 + 35x_2 + 25x_3 \leq 400 & \text{cost constraint (\$1000)} \\ \log(0.3)x_1, \log(0.4)x_2, \log(0.2)x_3 \leq \log(0.05) & \text{reliability constraints (log scale)} \end{cases}.$$

(Note that the reliability constraints imply the non-negativity constraints that $x_1, x_2, x_3 \geq 0$.)