CS 170 HW 7

Due on 2019-03-11, at 10:00 pm

1 Study Group

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

2 Steel Beams

You're a construction engineer tasked with building a new transit center for a large city. The design for the center calls for a T-foot-long steel beam for integer $T > 0$. Your supplier can provide you with an *unlimited* number of steel beams of integer lengths $0 < c_1 < \ldots < c_k$ feet. You can weld as many beams as you like together; if you weld together an a-foot beam and a b-foot beam you'll have an $(a + b)$-foot beam. Unfortunately, every weld increases the chance that the beam might break, so you want as few as possible.

Your task is to design an algorithm which outputs how many beams of each length you need to obtain a T-foot beam with the minimum number of welds, or 'not possible' if there's no way to make a T-foot beam from the lengths you're given. (If there are multiple optimal solutions, your algorithm may return any of them.)

(a) Consider the following greedy strategy. Start with zero beams of each type. While the total length of all the beams you have is less than T, add the longest beam you can without the total length going over T.

(i) Suppose that we have 1-foot, 2-foot and 5-foot beams. Show that the greedy strategy always finds the optimum.

(ii) Find a (short) list of beam sizes $c_1, \ldots, c_k$ and target T such that the greedy strategy fails to find the optimum. Briefly justify your choice.

(b) Give a dynamic programming algorithm which always finds the optimum. Describe your algorithm, including a clear statement of your recurrence, show that it is correct and prove its running time. How much space does your algorithm use?

Solution: Formal statement of problem: Given a list of integers $C = (c_1, \ldots, c_k)$ with $0 < c_1 < \ldots < c_k$ and a target $T > 0$, the algorithm should output **nonnegative** integers $(a_1, \ldots, a_k)$ such that $\sum_{i=1}^{k} a_i c_i = T$ where $\sum_{i=1}^{k} a_i$ is as small as possible, or return 'not possible' if no such integers exist.

(a) (i) Let a_1, a_2, a_3 be some optimum solution. We know that $a_1 < 2$ since if $a_1 \geq 2$ we can improve the solution by taking $a_1 - 2, a_2 + 1, a_3$. If $a_1 = 1$ then $a_2 < 2$ because otherwise we could improve by taking $0, a_2 - 2, a_3 + 1$. So the possible values of the optimum are $0, 1, j$, $0, 2, j$, $1, 1, j$ for some $j \geq 0$. In each case the greedy algorithm would give the same answer.

(ii) $C = (4, 5)$, $T = 8$ is one possibility.

(b) Description: We create a dynamic programming algorithm where, for each $n \leq A$, we will find the minimum integer combination that sums to n. The recurrence is $f(n) = \min_{i=1}^{k} f(n-c_i)+1$, with $f(0) = 0$. Maintain pointers so we can trace back the solution. We initialize the array to ∞. If $f(T) = \infty$ then return 'not possible', otherwise return the solution.

Proof of correctness: By induction. Base case is easy, so fix $n > 0$. Suppose that $f(n')$ is optimal for all $n < n'$. Firstly note that if $a'_1, \ldots, a'_k$ is a minimum integer combination summing to $n-c_i$, then $a'_1, \ldots, a'_i+1, \ldots, a'_k$ is an integer combination summing to n (not necessarily minimum). Let $a_1, \ldots, a_k$ be a minimum integer combination summing to n. Then for every i with $a_i > 0$, $a_1, \ldots, a_i - 1, \ldots, a_k$ is a minimum integer combination summing to $n-c_i$ (otherwise $a_1, \ldots, a_k$ wouldn't be optimal). We know that some $a_i > 0$ (we don't know which), so we take the minimum over all i.

Running time: We compute T subproblems, each one being a minimum of k values, so running time is $O(Tk)$. The space requirement is $O(T)$.

3 Egg Drop

You are given two identical eggs and a hundred story building. You need to figure out the maximum floor $\ell \in \{1, \ldots, 100\}$ that you can drop them from without breaking them. Each egg will break if dropped from a floor greater than or equal to ℓ, will never break when dropped from a floor less then ℓ, and once an egg breaks, you cannot use it any more.

(a) What is a strategy that takes the fewest number of drops to figure out what ℓ is, no matter what value ℓ is? What is the maximum number of drops in this strategy?

(b) What if you had an n story building? What if you had k eggs?

Solution:

(a) The strategy for the basic problem is to start by dropping the first egg on the 14^{th} floor. If it breaks, we need to do a linear search from $1 \ldots 13$ to find ℓ. Else, the next drop of the first egg will be at the $14 + 13 = 27^{th}$ floor. If it breaks there, we must do a linear search from $15 \ldots 26$ to find ℓ. Else, the next drop is on the $14+13+12 = 39^{th}$ floor, etc. The maximum number of drops with this strategy on a hundred story building is 14.

More in depth

An egg could break at any time. When it does, well be down to the one-egg problem. So we need to solve the one-egg problem first. How many floors can we explore with a single egg? Since we cant afford to lose our only egg, were forced to start at the ground floor. If the egg breaks, then weve found the right floor. Otherwise, we get to keep our egg, and we can try again with the next floor. Continuing on like this, the distance we can explore with 1 egg and t tries is:

$$d_1(t) = t$$

Now lets use this to solve the two-egg problem, where we start with 2 eggs and t tries: After we have dropped our first egg, well have explored a single floor. If the egg broke, then well have to explore the lower floors with 1 egg and $t-1$ tries. If the egg survived, then were free to explore the upper floors with 2 eggs and $t-1$ tries. Consequently, the overall distance we can explore with 2 eggs and t tries is:

$$d_2(t) = 1 + d_1(t-1) + d_2(t-1)$$
$$d_2(t) = 1 + (t-1) + d_2(t-1)$$
$$d_2(t) = t + d_2(t-1)$$

We can expand this recurrence relation to get:

$$d_2(t) = t + (t-1) + ... + d_2(1)$$

When were down to our last try, we can only use one egg. So $d_2(1) = d_1(1) = 1$. In general, extra eggs arent useful when we dont have enough turns left to use them. This gives us the closed-form solution:

$$d_2(t) = t + (t-1) + ... + 1$$
$$d_2(t) = t(t+1)/2$$

In particular:

$$d_2(13) = 91$$
$$d_2(14) = 105$$

So, given two eggs, were guaranteed to find the right floor within fourteen tries.

(b) Let $f(n,k)$ be the minimum number of drops to find ℓ if there are k eggs and n possible values that ℓ can be. For arbitrary n and k, we wish to compute $f(n,k)$. We have that $f(1,k) = 1$, $f(0,k) = 0$, $f(n,1) = n$ since we must do a linear search to find ℓ if we only have one egg, and $f(n,0) = \infty$ for $n > 0$. We then get the dynamic program:

$$f(n,k) = 1 + \min_{x \in \{1...n\}} \max\{f(x-1,k-1), f(n-x,k)\}.$$

We can interpret the recurrence relation as minimizing the number of drops we need in the worst case. At each step, we use one drop and choose the best floor out of the n floors (hence the $1 + \min_{x\in\{1...n\}} ...$). To determine which floor is the best, we consider that either the egg breaks, or it doesn't, and we choose the floor whose worst case is the best.

4 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.

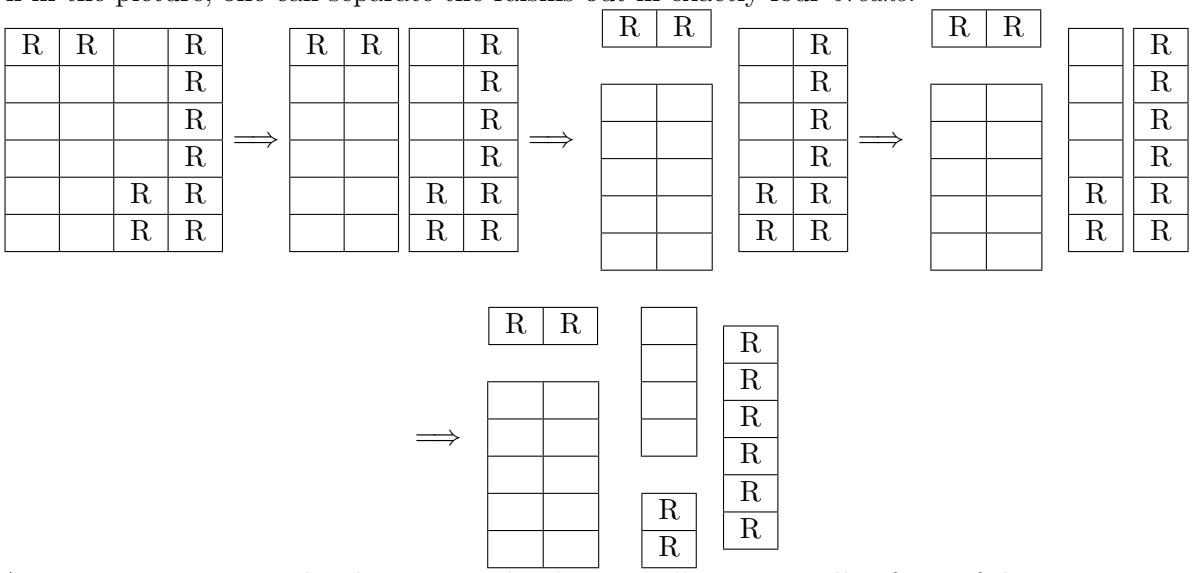
Runtime Analysis: There are n iterations of the outer for loop and m iterations of the inner for loop. Inside each of these loops, checking that the two strings are equal takes $O(\text{length}(d[a])) \leq O(\ell)$ time. Hence, the total running time is $O(nm\ell)$.

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.

5 Breaking Chocolate

There is a chocolate bar consisting of an $m \times n$ rectangular grid of squares. Some of the squares have raisins in them, and you hate raisins. You would like to *break* the chocolate bar into pieces so as to separate all the squares with raisins, from all the squares with no raisins.

For example, shown below is a 6×4 chocolate bar with raisins in squares marked R. As shown in the picture, one can separate the raisins out in exactly four *breaks*.



(At any point in time, a *break* is a cut either horizontally or vertically of one of the pieces at the time)

Design a DP based algorithm to find the smallest number of breaks needed to separate all the raisins out. Formally, the problem is as follows:

Input: Dimensions of the chocolate bar $m \times n$ and an array $A[i,j]$ such that $A[i,j] = 1$ if and only if the ij^{th} square has a raisin.

Goal: Find the minimum number of breaks needed to separate the raisins out.

(a) Define your subproblem.

(b) Write down the recurrence relation for your subproblems.

(c) What is the time complexity of solving the above mentioned recurrence? Provide a justification for the same.

Solution:

(a) We define $B[i_1, j_1, i_2, j_2]$ to be the minimum number of breaks needed to separate the sub-matrix $A[i_1 \leq i_2, j_1 \leq j_2]$ into pieces consisting either entirely of raisin pieces or entirely of non-raisin pieces.

(b)

$$B[i_1, j_1, i_2, j_2] = \min \begin{cases} 0, & \text{if all entries of } A[i_1 \dots i_2, j_1 \dots j_2] \text{ are equal} \\ 1 + B[i_1, j_1, i_1 + k, j_2] + B[i_1 + k + 1, j_1, i_2, j_2] & \text{for any } k \in \{1, \dots, i_2 - i_1\} \\ 1 + B[i_1, j_1, i_2, j_1 + k] + B[i_1, j_1 + k + 1, i_2, j_2] & \text{for any } k \in \{1, \dots, j_2 - j_1\} \end{cases} \tag{1}$$

Alternatively, you could have also encapsulated the 0 base case in all single-square pieces, and determined if a piece was pure via the merging, see below.

(c) Two answers are acceptable: $O((m+n)m^2n^2)$ and $O(m^3n^3)$

We have $O(m^2n^2)$ total subproblems: $O(mn)$ possibilities for (i_1, j_1), and $O(mn)$ possibilities for (i_2, j_2). For each subproblem, we examine up to m possible choices for horizontal splits, and n possible choices for vertical splits. A single split consideration will result in two smaller subproblems, which we can assume have already been solved, so we just need to find the best split, which takes $O(n + m)$ time.

In addition, for a subproblem, we also want to check the base case for if the piece is "pure" (contains only raisins, or contains only non-raisins). Brute force checking this takes $O(mn)$ time, for a total subproblem time of $O(mn + (m + n)) \to O(mn)$.

However, this $O(mn)$ factor can be reduced to $O(m + n)$, and here is how (didn't need to do it to receive full points). We can precompute the purities of every single possible subrectangle and store it in a table. Technically, for us to pre-compute faster than $O(m^3n^3)$, we'll need to compute the purities more intelligently than by brute-force. It is possible to do this in as fast as $O(\max\{m, n\}^2)$ time, although the details of this are complicated and won't be explained here (feel free to ask). So to solve our recurrence relation, if we can determine purity/impurity in $O(1)$ time, then we can reach an overall time of $O((m + n)m^2n^2)$.

Alternatively, we can initialize all min-break values of single square pieces to be 0. Then, if it is possible to have some break such that both resulting pieces have min-break values of 0, and both resulting pieces are of the same type (raisin-only or non-raisin-only, and we can take any sample of either and compare them), then we ourself are a pure piece. This would allow you to avoid the entire pre-computation business as mentioned before, and still achieve a runtime of $O((m + n)m^2n^2)$.

6 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.

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.

7 Propositional Parentheses

You are given a propositional logic formula using only $\wedge$, $\vee$, T, and F that does not have parentheses. You want to find out how many different ways there are to *correctly parenthesize* the formula so that the resulting formula evaluates to true.

A formula A is correctly parenthesized if $A = T$, $A = F$, or $A = (B \wedge C)$ or $A = (B \vee C)$ where B, C are correctly parenthesized formulas. For example, the formula $T \vee F \vee T \wedge F$ can be correctly parenthesized in 5 ways:

$$(T \vee (F \vee (T \wedge F))) \quad (T \vee ((F \vee T) \wedge F)) \quad ((T \vee F) \vee (T \wedge F))$$
$$(((T \vee F) \vee T) \wedge F) \quad ((T \vee (F \vee T)) \wedge F)$$

of which 3 evaluate to true: $((T \vee F) \vee (T \wedge F))$, $(T \vee ((F \vee T) \wedge F))$, and $(T \vee (F \vee (T \wedge F)))$.

(a) Give a dynamic programming algorithm to solve this problem. Describe your algorithm, including a clear statement of your recurrence, show that it is correct, and prove its running time.

(b) Briefly explain how you could use your algorithm to find the probability that, under a *uniformly randomly chosen* correct parenthesization, the formula evaluates to true.

Solution:

(a) Choosing parentheses amounts to deciding how to represent the formula as a binary tree. Let $A = x_1 o_1 x_2 o_2 \dots x_{n-1} o_{n-1} x_n$, where $x_i \in \{T, F\}$ and $o_i \in \{\wedge, \vee\}$. We first choose an operator o_r to be the root. This splits A into subformulas $x_1 o_1 \dots x_{r-1} o_{r-1} x_r$ and $x_{r+1} o_{r+1} \dots x_{n-1} o_{n-1} x_n$. We then recurse on each side, choosing operators to be the roots of the left and right subtrees, until we reach a formula which is just T or F. Every correct parenthesization corresponds to exactly one such binary tree.

Given such a tree it is clear how to evaluate it: take any node v labelled with $o \in \{\wedge, \vee\}$ with children labelled $x_l, x_r \in \{T, F\}$ and replace v's label with the value of $x_l \, o \, x_r$. Keep doing this until the root is labelled with T or F. This is the same as evaluating the corresponding parenthesized formula.

The subproblems in our dynamic program will be defined by pairs (i, j) with $i \leq j$. Let $A_{i,j} = x_i o_i \dots x_j o_{j-1} x_j$. Let $t(i, j)$ be the number of ways to parenthesize $A_{i,j}$ which evaluate to T; let $f(i, j)$ be the number of ways to parenthesize $A_{i,j}$ which evaluate to F. We obtain the following recurrences:

$$t(i,i) = \begin{cases} 1 & \text{if } x_i = T \\ 0 & \text{if } x_i = F \end{cases}$$

$$t(i,j) = \sum_{\substack{i \leq k < j \\ o_k = \vee}} t(i,k) \cdot t(k+1,j) + f(i,k) \cdot t(k+1,j) + t(i,k) \cdot f(k+1,j) + \sum_{\substack{i \leq k < j \\ o_k = \wedge}} t(i,k) \cdot t(k+1,j)$$

$$f(i,i) = \begin{cases} 0 & \text{if } x_i = T \\ 1 & \text{if } x_i = F \end{cases}$$

$$f(i,j) = \sum_{\substack{i \leq k < j \\ o_k = \wedge}} f(i,k) \cdot f(k+1,j) + f(i,k) \cdot t(k+1,j) + t(i,k) \cdot f(k+1,j) + \sum_{\substack{i \leq k < j \\ o_k = \vee}} f(i,k) \cdot f(k+1,j)$$

The proof is by induction on the size of an interval. Every interval of size 1 is correct since there's only one way to parenthesize T or F (i.e. by not giving them parentheses). Now let $m := j - i + 1$ and suppose that t and f are correct for every interval of size less than m. Every correct parenthesization is given by choosing a root o_k and then choosing some correct parenthesization of the left and right subformulas. We'll look at the t recurrence; f is similar. If $o_k = \vee$, then such a parenthesization evaluates to T if and only if at least one of its children does. If $o_k = \wedge$, then we need both children to evaluate to T. Since we can choose the left and right parenthesizations independently, we obtain the expression above.

Our algorithm must compute the $t(i, j)$ and $f(i, j)$ in order of interval size: we start with all intervals of size 1, then size 2, etc. The running time is $O(n^3)$ because there are $O(n^2)$ intervals and it takes $O(n)$ time to compute $t(i, j)$ and $f(i, j)$. The result is found in $T(1, n)$.

(b) Each correct parenthesization either evaluates to true or false. So the total number of correct parenthesizations of the given formula is $t(1, n) + f(1, n)$. Thus the probability that a randomly drawn correct parenthesization evaluates to true is $t(1, n)/(t(1, n) + f(1, n))$.