CS 170 HW 7

Due 2019-10-16, at 10:00 pm

1 Study Group

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

2 Money Changing Redux

During discussion section, we saw a simple greedy algorithm to try to find change that adds up to a given number. We saw that the greedy algorithm didn’t find the optimal solution in all cases. In this problem, we will use our newly-found powers of computer science to fix this. Recall that in the money-changing problem, we were given a fixed set of positive integers called denominations $x_1, x_2, \ldots, x_n$ (think of them as the integers 1, 5, 10, and 25). The problem you want to solve for these denominations is the following: Given an integer $A$, express it as

$$A = \sum_{i=1}^{n} a_i x_i$$

for some nonnegative integers $a_1, \ldots, a_n \geq 0$. Find the way to do this using the minimum number of coins, e.g. so that $\sum_{i=1}^{n} a_i$ is as small as possible.

(a) You might remember that we can represent any integer $k$ in unary form by repeating $k$ consecutive 1s (e.g., 3 is represented by 111 in unary). Assume you are given a positive integer $A$ and a set of denominations $x_1, x_2, \ldots, x_n$ in unary form. Give a fast algorithm to solve the money-changing problem.

(Please provide a 3-part solution)

(b) If you are given $A$ and $x_1, x_2, \ldots, x_n$ in binary, does your algorithm still run in polynomial time? Why or why not?

Solution:

(a) Main idea: We create a dynamic programming algorithm where, for each $n \leq A$, we will find the minimum number of coins needed to sum to $n$.

Pseudocode:

```python
Denominations = [ x1, x2, ..., xk ]
min_coins(0) = 0
for n = 0 to A:
    run & store min_coins(n)
return min_coins(A)
```

```python
min_coins(n){
    if n < 0:
        return \infty
    if n == 0:
        return 0
    min_coins = \infty
    for i = 1 to \min(k, \lfloor n/x_i \rfloor + 1):
        min_coins = \min(min_coins, min_coins(n - i * x_i) + i)
    return min_coins
}
```
Proof: We prove by strong induction on the size of \( A \) that the algorithm either returns the optimal solution (if it exists) or "infinity" (if it doesn’t exist). Base: If \( A == 0 \), the algorithm returns 0. If \( A < 0 \), the algorithm returns "infinity". Inductive: Assume the proof holds for all \( k < A \). If \( A \) doesn’t have a solution, then for all \( A - x_i \), the algorithm returns "infinity", by the induction hypothesis. If \( A \) has a solution, the optimal solution must contain some \( x_i \). By the induction hypothesis, the algorithm returns the optimal solution on \( A - x_i \) and thus the optimal solution on \( A \). Thus the algorithm works for all \( A \).

Runtime: This algorithm runs through each of \( k \) denominations once for all \( n < A \), so the runtime is \( O(kA) \).

(b) The algorithm becomes exponential. If \( A \) is given in binary form, then the value of \( A \) is exponential in the size of the binary representation of \( A \). Since the solution has to solve the money-changing problem for at most every number less than \( A \), this takes exponential time in the representation of \( A \).

3 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.

Please provide a 3-part solution. 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 \( M'(g) \) 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

$$M^1(g) = c_1g \quad \text{where } 0 \leq g \leq C.$$ 

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 $N$.

```plaintext
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.

4 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) Prove correctness of your algorithm by induction.

(i) Show that the base case is correct.

(ii) Assuming that $W(x, y, z)$ is computed correctly for previous subproblems, show that it is correct for the current subproblem.

(iii) Give the order where you can solve the subproblems. Show that for this order, evaluating the recurrence relation will use only subproblems that have already been computed.

(c) Find the runtime of your algorithm.

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 will 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 is still their turn, so they will win with probability $W(x, y, z + v)$. Since each value of the die has probability $1/6$, this means

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

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

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

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

(b) (i) The base case is when a player has a large enough running total to win immediately by passing, which is $W(x, y, z)$ where any of $x, y, z$ are at least $N$.

(ii) To compute $W(x, y, z)$, we assume that $W(x', y', z')$ are computed correctly for $x' > x$ or $y' > y$, and for $z' > z$. Then the proof in part (a) shows that $W(x, y, z)$ is correct when computed from the recurrence relation.
(iii) In the recursive formula, every recursive term either increases the number of points
one of the players has, or increases the running total. Hence we solve the subprob-
lems in decreasing order of $x$, $y$, $z$, starting from where $x$, $y$, $z$ are at least $N$, to the
subproblems where $x$, $y$, $z$ are 0.

(c) 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 $O(N^3)$.

5 Propositional Parentheses

You are given a propositional logic formula using only $\land$, $\lor$, $T$, 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 $A = (B \land C)$ or $A = (B \lor C)$
where $B$, $C$ are correctly parenthesized formulas. For example, the formula
$T \lor F \lor T \land F$
can be correctly parenthesized in 5 ways:

$$(T \lor (F \lor (T \land F))) \quad (T \lor ((F \lor T) \land F)) \quad ((T \lor F) \lor (T \land F))
\quad ((T \lor F) \lor (T \lor F)) \quad (((T \lor F) \lor T) \land F)$$

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

(a) Give a dynamic programming algorithm to solve this problem.

(Please provide a 3-part solution)

(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 $A = x_1 o_1 x_2 o_2 \ldots x_{n-1} o_{n-1} x_n$, where $x_i \in \{T, F\}$ and $o_i \in \{\land, \lor\}$. We first choose
an operator $o_r$ to be the root. This splits $A$ into subformulas $x_1 o_1 \ldots x_{r-1} o_{r-1} x_r$
and $x_{r+1} o_{r+1} \ldots x_{n-1} o_{n-1} x_n$. 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 $o \in \{\land, \lor\}$
with children labelled $x_i, x_r \in \{T, F\}$ and replace $v$’s label with the value of $x_i o x_r$. 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
$A_{i,j} = x_i o_1 \ldots x_j o_{j-1} x_j$. 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:

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

\[
t(i, j) = \sum_{i \leq k < j} t(i, k) \cdot t(k + 1, j) + f(i, k) \cdot t(k + 1, j) + t(i, k) \cdot f(k + 1, j)
\]

\[
+ \sum_{i \leq k < j} t(i, k) \cdot t(k + 1, j)
\]

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

\[
f(i, j) = \sum_{i \leq k < j} f(i, k) \cdot f(k + 1, j) + f(i, k) \cdot t(k + 1, j) + t(i, k) \cdot f(k + 1, j)
\]

\[
+ \sum_{i \leq k < j} f(i, k) \cdot f(k + 1, j)
\]

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 \( \alpha_k \) and then choosing some correct parenthesization of the left and right subformulas. We’ll look at the \( t \) recurrence; \( f \) is similar. If \( \alpha_k = \vee \), then such a parenthesization evaluates to \( T \) if and only if at least one of its children does. If \( \alpha_k = \wedge \), 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 \( t(1, n)/(t(1, n) + f(1, n)) \).

6 Knightmare

Give an algorithm to find the number of ways you can place knights on an \( N \) by \( M \) \((M < N)\) chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). Clearly describe your algorithm and prove its correctness. The runtime should be \( O(2^{3M} M \cdot N) \).
(Please provide a 3-part solution)

Solution:
We use length $M$ bit strings to represent the configuration of rows of the chessboard (1 means there is knight and otherwise 0).
The main idea of the algorithm is as follows: we solve the subproblem of the number of valid configurations of $(n-1) \times M$ chessboard and use it to solve the $n \times M$ case. Note that as we iteratively incrementing $n$, a knight in the $n$-th rows can only affect configurations of rows $n+1$ and $n+2$. So we can denote $K(n, u, v)$ as the number of possible configurations of the first $n$ rows with $u$ being the $(n-1)$-th row and $v$ being the $n$-th row, and then use dynamic programming to solve this problem.
Pseudocode: We say bit strings $u, v$ is valid no two knights can attack each other in the $2 \times M$ table represented by $u$ and $v$. Similarly we say bit strings $u, v, w$ is valid no two knights can attack each other in the $3 \times M$ table represented by $u, v$ and $w$.

Initialize $K(\cdot, \cdot, \cdot) := 0$
for all size $M$ bitstrings $v, w$
do
    Initialize $K(2, v, w) := 1$ if $v, w$ is valid else 0
for $n = 3$ to $N$
do
    for all size $M$ bitstrings $u, v, w$ if $u, v, w$ is valid
    do
        $K(n, v, w) += K(n-1, u, v)$
    return $\sum_{v,w} K(N, v, w)$

Proof of Correctness: By definition, $K(2, v, w) = 1$ if the $M$ by 2 chessboard configuration defined by $v$ and $w$ is legitimate. Otherwise, $K(2, v, w) = 0$.
For $n > 2$, we have

$$K(n, v, w) = \sum_u K(n-1, u, v),$$

where we are summing over all possible configurations $u$ for the third-last row of a chessboard whose last rows are specified by $v$ and $w$.
To bound the total runtime, first note that for each row, we iterate over all possible configurations of knights in the row and for each such configuration, we perform a sum over all possible valid configurations of the previous two rows. The time to check if a configuration is valid is $O(M)$. Therefore, the time taken to compute the sub-problems for a single row is $O(2^{3M}M)$ which gives us an overall runtime of $O(2^{3M}MN)$. 

7