# CS 170 HW 6

## Due on 2018-03-04, at 11:59 pm

## 1 (★) Study Group

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

## 2 (★) Longest Increasing Subsequence

Given an array $A$ of $n$ integers, here is a linear time algorithm to find the length of the longest increasing subsequence, where $M[j]$ represents the length of the longest increasing subsequence of the first $j$ integers of $A$.

$$M[1] = 1$$

$$M[i] = \begin{cases} M[i-1] & \text{if } A[i-1] > A[i] \\ M[i-1]+1 & \text{otherwise} \end{cases}$$

Verify that this algorithm takes linear time. Is the algorithm correct? Prove it is or give a counter-example and explain.

**Solution:** The algorithm starts from the left of $A$ and works its way up, comparing each integer with the one before it and the one after it. Therefore, each integer is only touched twice, resulting in a linear run-time.

The algorithm is incorrect.

A counterexample is $A = [3, 4, 1, 2]$

The longest increasing subsequence length is 2, but the algorithm just counts (one plus) the number of indices $i$ such that $A[i] > A[i-1]$, and then it returns 3 (as if $A[1], A[2], A[4]$ were an increasing subsequence).

## 3 (★★) Copper Pipes

Bubbles has a copper pipe of length n inches and an array of 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. Give your answer in a four-part solution, but give only a brief justification of your algorithm in the proof section.

**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 $O(n^2)$.

## 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. Give only a brief justification of your algorithm. 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 $1 \le i \le n$ and $0 \le g \le C$.

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 (★★★★) Propositional Parentheses

Say you are given a propositional logic formula using $\wedge, \vee, T$, and $F$ that does not include parentheses. You want to find out how many different ways there are to draw parentheses so that the resulting formula evaluates to true. For example, the formula $T \vee F \vee T \wedge F$ has 3 solutions: $(T \vee F) \vee (T \wedge F)$, $T \vee ((F \vee T) \wedge F)$, and $T \vee (F \vee (T \wedge F))$.

(a) Give a four-part solution that solves this problem.

(b) Say you were to randomly draw parentheses so that the sentence is now syntactically correct. For example, the formula $T \vee T \wedge F$ might yield $(T \vee T) \wedge F$ and $T \vee (T \wedge F)$ with equal probability, but not $T(\vee T) \wedge F$ or $(T) \vee T)(\wedge F$, since they are not syntactically correct. Briefly explain how you could use your original algorithm to find the probability that randomly drawing syntactically-correct parentheses causes the sentence to become true.

**Solution:**

(a) Main Idea: Label the $T$ and $F$ elements of the formula $x_1, x_2, \ldots, x_n$, and let $OP(k)$ be the operator after $x_k$ for all $k$. For $i < j$, let $T(i, j)$ represent the number of ways to draw parenthesis so the subformula between $x_i$ and $x_j$ (inclusive) is true and let $F(i, j)$ represent the number of ways to draw parenthesis so the subformula is false. We can recursively construct $T(i, j)$ and $F(i, j)$ by splitting them into two subformulas over variables $x_i, \ldots, x_k$ and $x_{k+1}, \ldots, x_j$ (for all $k$ between $i$ and $j$) and computing $T(i, k)$, $F(i, k)$, $T(k+1, j)$ and $F(k+1, j)$. The recursive formula is given below:

```
T(i,i) = 1 if x_i = T
T(i,i) = 0 if x_i = F
F(i,i) = 1 if x_i = F
F(i,i) = 0 if x_i = T

T(i,j) =
sum() for all k between i and j

    //The number of ways both subformulas are true
     if OP(k) is ∧,  $T(i,k) \cdot T(k+1,j)$

    //The sum of the number of ways both subformulas are true,
           // the number of ways the right subformula is true,
                     // and the number of ways left subformula is true
     if OP(k) is ∨,  $T(i,k) \cdot T(k+1,j) + F(i,k) \cdot T(k+1,j) + T(i,k) \cdot F(k+1,j)$


And F(i,j) =
sum() for all k between i and j

    //The sum of the number of ways both subformulas are false,
           // the number of ways the right subformula is false,
                     //and the number of ways left subformula is false
     if OP(k) is ∧,  $F(i,k) \cdot F(k+1,j) + F(i,k) \cdot T(k+1,j) + T(i,k) \cdot F(k+1,j)$

     if OP(k) is ∨,  $F(i,k) \cdot F(k+1,j)$
```

We can fill in this recursive formula by starting with the smallest subformulas and working our way up to the largest subformulas, as given in the following pseudocode:

```
Count_True_Parenthesizations(OP, [x_1,\dots, x_n]):
    For i from 0 to n:
        If x_i == 0:
                        F(i,i) = 1
                T(i,i) = 0
        Else if x_i == 1:
                F(i,i) = 0
                T(i,i) = 1

    //gap is the size of the subformula being considered
    For gap = 1 to n:
        //considering all subformulas of size gap from left to right
        For i = 0, j = gap,  j < n, i++, j++:
            T(i,j) = F(i,j) = 0
            For k from i to j:
                If OP(k) == ∧:
                        T(i,j) += T(i,k)*T(k+1,j)
                        F(i,j) += F(i,k) * F(k+1,j) +
                                  F(i,k) * T(k+1,j) + T(i,k) * F(k+1,j)
                If OP(k) == ∨:
                        T(i,j) += T(i,k) * T(k+1,j) +
                                 F(i,k) * T(k+1,j) + T(i,k) * F(k+1,j)
                        F(i,j) += F(i,k)*F(k+1,j)
return T(1,n)
```

Proof: By strong induction on $m$ we will see that we produce the correct numbers of truth-yielding and false-yielding parethesizations for all subformulas of size $m$. The base-case, when a single variable is given, is easy to see. Assume we the correct numbers of truth-yielding and false-yielding parethesizations for all subformulas of size less than or equal to $m$. For any $k$ value used to split the subformula into $x_i, \dots, x_k$ and $x_{k+1}, \dots, x_{i+m}$, it is easy enough to check that the recursive formulas is correct. For example, if $OP(k) = \vee$, then the formula is true when both subformulas are true (which occurs $T(i,k) \cdot T(k+1,j)$ times), only the right subformula is true (which occurs $F(i,k) \cdot T(k+1,j)$ times) and only the left subformula is true (which occurs $T(i,k) \cdot F(k+1,j)$ times). Since these cases are mutually exclusive, this yields the expression seen in the recursive formula:

$$T(i,k) \cdot T(k+1,j) + F(i,k) \cdot T(k+1,j) + T(i,k) \cdot F(k+1,j)$$

Since different choices of $k$ values must yield different parenthesizations, we can sum this expression (or the expression when $OP(k) = \wedge$) for all $k$ between $i$ and $i+m$ to obtain $T(i, i+m)$. Thus, the algorithm always produces the correct values of $T(i,j)$ and $F(i,j)$.

Runtime: The algorithm contains three nested for-loops, each of which runs on at most $n$ values per call, resulting in a run-time of $O(n^3)$.

(b) Each syntactically correct parenthesization either evaluates to true or false. So the total number of syntactically 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 (★★) Jeweler

You're a jeweler. You can make necklaces or engagement rings. Necklace takes 4 oz of gold. Engagement ring takes 1 oz of gold + 1 diamond. You have a limited supply of gold at only 100 ounces, but unlimited diamonds. You make 30 profit per necklace and 100 per ring. There are only so many people getting married each month, so you can sell at most 40 engagement rings. You want to figure out how many of each type of jewelry you should make each month to maximize your profits. Formulate this problem as a linear programming problem and find the solution (state the cost-function, linear constraints, and vertices).

**Solution:**
$x$ = number of necklaces
$y$ = number of engagement rings

Maximize: $30x + 100y$

Linear Constraints:
$4x + y \leq 100$
$y \leq 40$
$x \geq 0$
$y \geq 0$

Draw picture in 2 dimensions
Vertices: $(x = 0, y = 40), (x = 15, y = 40), (x = 25, y = 0)$ Maximum at $(x = 15, y = 40)$ with utility of 4450

## 7 (★★★★★) Largest Substring Sum (Extra-Credit)

Let's say we are given a sequence of positive and negative integers $x_1, x_2, \ldots, x_n$. We use $sum(i,j)$ represent the sum of the *substring* $x_i, x_{i+1}, \ldots, x_j$. Give a four-part solution describing a linear-time algorithm to find the substring of $x_1, x_2, \ldots, x_n$ with the largest sum.

For example, the largest-summing substring of $[-1, 5, -2, -1, 4, -2, 1]$ is $[5, -2, -1, 4]$, which sums to 6.

**Solution:** Main idea: $x_i, x_{i+1}, \ldots, x_j$ has the maximum sum when the sum of the remaining elements $x_0, \ldots, x_{i-1}$ and $x_{j+1}, \ldots, x_n$ is minimized. We will create a dynamic programming algorithm that finds $sum(0,i)$ and $sum(i,n+1)$ for all $i$ (we treat $x_0$ and $x_{n+1}$ as equalling 0, just so we can say $sum(0,0) = sum(n+1,n+1) = 0$). We want to find $j$ and $k$, for $j < k$ such that $x_0 + \cdots + x_j + x_k + \cdots + x_n$ is as small as possible. To do this for each $i$, we let $Left(i) = min_{j<i}\{sum(0,j)\}$ and $Right(i) = min_{k\geq i}\{sum(k,n+1)\}$. We then return the $i$ such that $Left(i) + Right(i)$ is minimized.

Pseudocode:

```
LSS([$x_0, \ldots, x_n$]){

    //computes Left() values
    min_left = 0
    sum(0,0) = 0
    for i = 1 to n+1:
        sum(0,i) = x_i + sum(0, i-1)
        if sum(0,i) < min_left:
            min_left = sum(0,i)

    //computes Right() values
    sum(n+1,n+1) = 0
    min_right = 0
    for i = n to 1:
        sum(i,n+1) = x_i + sum(i+1, n+1)
        if sum(i,n+1) < min_right:
            min_right = sum(i,n+1)

    //computes min-summing endstrings
    min_ends = 0
    for i = 0 to n+1:
        if Left(i) + Right(i) < min_ends:
            min_ends = Left(i) + Right(i)

    return sum(0, n+1) - min_ends
```

Proof: Maximizing $x_i, x_{i+1}, \ldots, x_j$ is the same as minimizing $x_0, \ldots, x_{i-1}, x_{j+1}, \ldots, x_n$ since the sum of these two sequences is the same regardless of $i$ and $j$. The construction of Left() and Right() is fairly easy to see. We guarantee that $i$ is left of $j$ through the last for-loop.

Runtime: The algorithm uses three non-nested for-loops, leading to a run-time of O(n)