CMPSCI 311: Introduction to Algorithms **Fall 2016**

Homework 4

Name: Solutions

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Instructions. You make work in groups, but you must individually write your solutions yourself. List your collaborators on your submission.

If you are asked to design an algorithm as part of a homework problem, please provide: (a) the pseudocode for the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

Submission instructions. This assignment is due by 8:00pm on 11/4/2016 in Moodle. Please submit a pdf file. You may submit a scanned handwritten document, but a typed submission is preferred.

1. **Dynamic Programming Short Answer**

   (a) **Scheduling.** The solution here is based on a reduction to subset sum, which we already saw how to solve with dynamic programming. Let $T = \sum_{i=1}^{n} t_i$ and observe that for any set $S \subset \{1, \ldots, n\}$ we have $F_1 = \sum_{i \in S} t_i$ and $F_2 = T - F_1$. We are trying to minimize $\max\{F_1, T - F_1\}$, which is equivalent to trying to maximize F_1, provided that we do not exceed $T/2$ (since if we stay below $T/2$, the term $T - F_1$ will be the maximum).

   Thus, we want to find a subset $S \subset \{1, \ldots, n\}$ that maximizes $\sum_{i \in S} t_i$ subject to $\sum_{i \in S} t_i \leq T/2$. This is exactly the subset sum problem, and we can run the standard dynamic program to solve this in $O(nT/2)$ time. As a reminder, the recursion is

   $$\text{val}(j, w) = \max\{\text{val}(j-1, w), \text{val}(j-1, w - t_j) + t_j\}.$$

   where $j \in \{1, \ldots, n\}$ and $w \in \{1, \ldots, T/2\}$.

   As for the running time, if the numbers t_i are represented as k-bit integers, then we know that $T \leq n2^k$ and so the total running time is

   $$O(nT/2) = O(n^2 2^{k-1}) = O(n^2 2^k).$$

   (b) **Longest Common Substring.** The idea here is to run a dynamic program where the subproblems correspond to prefixes of both strings. Let $\text{val}(i, j)$ denote the longest common *suffix* of $x_1 \ldots x_i$ and $y_i \ldots y_j$. Then the recurrence is

   $$\text{val}(i, j) = (1 + \text{val}(i-1, j-1))\mathbf{1}\{x_i = y_j\}.$$

   Here $\mathbf{1}\{\cdot\}$ denotes the indicator function, which is 1 if the quantity inside is true, and zero otherwise. The pseudocode is in Algorithm 1 The algorithm runs the forward dynamic program (there is also an analogous "backward" program), then finds the largest element in the dynamic programming table. The longest common substring is the substring of length v leading up to the indices of the largest element in the array.

   The running time is clearly $O(nm)$.

Algorithm 1 $LCS(X, Y)$

```
m = length(X), n = length(Y)
initialize M[0 : m, 0 : n] = 0 (m + 1 × n + 1 dimensional matrix)
M[:, 0] = 0, M[0, :] = 0
for i = 1, ..., m do
  for j = 1, ..., n do
    if x_i == y_j then
      M[i, j] = 1 + M[i − 1, j − 1]
    else
      M[i, j] = 0
    end if
  end for
end for
(i*, j*) = argmax_{i,j} M[i, j], v = max_{i,j} M[i, j]
return (x_{i*−v+1} ... x_{i*}, y_{j*−v+1} ... y_{j*})
```

2. **A Strategy Game.**

   The idea here is that Russell will compute something for every game state, where the game states are parameterized by the interval (i, j) of bills remaining and whose turn it is. When it is Russell's turn, he will choose the bill that maximizes his value, assuming that Jesse plays as maliciously as possible. When it is not his turn, he will simulate Jesse playing as maliciously as possible, which amounts to choosing the bill that will minimize Russell's future potential. The recurrences are given by:

   $$\text{val}(i, j, 1) = \max\{S[i] + \text{val}(i + 1, j, 2), S[j] + \text{val}(i, j - 1, 2)\}$$
   $$\text{val}(i, j, 2) = \min\{\text{val}(i + 1, j, 1), \text{val}(i, j - 1, 1)\}$$

   The first recurrence is straightforward. If it is Russell's turn, we want to maximize the instantaneous payoff ($S[i]$ or $S[j]$) plus what we can get from the rest of the game. For the second recurrence, since we are only interested in the amount Russell can achieve and since we are simulating Jesse being as malicious as possible, Jesse should choose an action to minimize the future value that Russell can get. It is also helpful to unroll the recurrence and eliminate the need for the second definition

   $$\text{val}(i, j) = \max\{S[i] + \min\{\text{val}(i + 2, j), \text{val}(i + 1, j - 1)\}, S[j] + \min\{\text{val}(i + 1, j - 1), \text{val}(i, j - 2)\}\}.$$

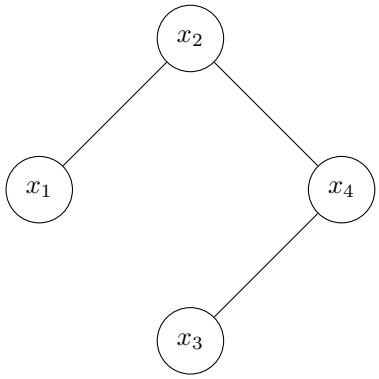
   This definition is equivalent to $\text{val}(i, j, 1)$ above.

   Using a standard dynamic programming approach these values can be computed for all pairs (i, j) in $O(n^2)$ time (solve the smaller length intervals first). As usual we should also store the argmax action.

   When we are playing the game, whenever it is our turn and we are at some state (i, j) we play the action the argmax action in the calculation of $\text{val}(i, j)$.

3. **Optimal BST.**

   (a) The optimal BST would be:



The total sum is $= 4 \times 1 + 2 \times 2 + 6 \times 2 + 3 \times 3 = 29$. There are also many other BSTs with this cost.

(b) Proof by contradiction: If we use x_i at the root, and have BSTs T_L, T_R below, then the cost will be:

$$c_i + \text{cost}(T_L) + \sum_{j=1}^{i-1} c_j + \text{cost}(T_R) + \sum_{j=i+1}^{n} c_j.$$

Here c_i comes from accessing x_i at the root, the two cost terms come from accessing elements in the left and right subtrees, but not counting the comparison with x_i at the root. We account for this comparison in the two summations. Rewriting, we get:

$$\sum_{i=1}^{n} c_i + \text{cost}(T_L) + \text{cost}(T_R).$$

From this it is clear that if we should use the optimal BSTs T_L, T_R. Or more formally, if T_L were a suboptimal BST on $x_1, \ldots, x_{i-1}$, we could improve the overall cost by instead using an optimal BST, which improves the cost(T_L) term.

(c) The decomposition above motivates a dynamic programming algorithm. The subproblems involve intervals (i, j) and the recurrence is

$$\text{val}(i, j) = \min_{i \le k \le j} \left\{ \sum_{t=i}^{j} c_t + \text{val}(i, k-1) + \text{val}(k+1, j) \right\}.$$

where we set $\text{val}(i, j) = 0$ for $i > j$.

The ordering you should use is to solve the smaller intervals first. The tree you produce repeatedly uses the argmax element as the root of the current subtree.

As for the running time, there are $O(n^2)$ subproblems, and solving each one requires taking a minimum of $O(n)$ terms. Apart from the sum (which can be precomputed before solving the current subproblem) evaluating each of the terms in the minimization takes $O(1)$ time, so the total running time is $O(n^3)$.

(d) **(0 points).** refer to the paper by Knuth [1].

4. **Piecewise Regression.**

(a) It is easy to see that no matter the segmentation S, we have $\sum_{s \in S} |s| = n$, which is a constant independent of S. Therefore, when performing the minimization, we simply need to worry about the "fit" term $\sum_{i=1}^{n} (y_i - x_i)^2$. This is always non-negative, and it is minimized when $y_i = x_i$ for all $i \in \{1, \ldots, n\}$. Thus, the segmentation is $\{\{1\}, \ldots, \{n\}\}$ (i.e. put everything in its own segment) and the solution is to simply return the vector x.

(b) Let

$$\text{val}(i) = \min_{y_1, \ldots, y_i} \sum_{j=1}^{i} (y_j - x_j)^2 - \sum_{s \in S(y)} |s|^2,$$

denote the cost of the optimal segmentation on just the prefix $y_1, \ldots, y_i$. Here we are using the notation $S(y)$ to denote the segments of the vector y. We will derive a recursive form for val(i).

The intuition is that we will "guess" at the length of the last segment, and then recursively solve the problem on the prefix. If we are trying to compute val(n) and we guess that the the last segment is length i, then since the cost decomposes additively, our cost will be

$$\text{val}(n - i) + \sum_{j=n-i+1}^{n} (\bar{y}_{n-i+1:n} - x_j)^2 - |i|^2,$$

where $\bar{y}_{n-i+1:n} = \frac{1}{i}\sum_{k=n-i+1}^{n} x_k$ is the correct value for $y_{n-i+1}, \ldots, y_n$ once we fix them to all be in the same segment. Since we know that the last segment is of some length, we can minimize over all choices for i

$$\text{val}(n) = \min_{1 \le i \le n} \left\{ \text{val}(n-i) + \sum_{j=n-i+1}^{n} (\bar{y}_{n-i+1:n} - x_j)^2 - |i|^2 \right\}.$$

The important fact here is that the "objective function" decomposes additively across the segments, so once we take a guess at the length of the last segment, we can recursively solve a smaller subproblem. At the same time, once we have a guess of the length of the last segment, we can easily compute how much this segment contributes to the overall cost.

For the running time, there are $O(n)$ subproblems, and each one requires minimizing over $O(n)$ terms. At first glance, computing each one of these terms seems to take $O(n)$ time since we have to compute $\bar{y}_{n-i+1:n}$ which requires taking an average of i values. This argument leads to $O(n^3)$ running time.

An few amortization tricks can bring this down to $O(n^2)$ overall running time. First notice that by expanding the square

$$\sum_{j=1}^{n} (y_j - x_j)^2 = \sum_{j=1}^{n} y_j^2 + x_j^2 - 2y_j x_j$$

Before worrying about the minimization, in one pass over the list x, we can compute all of the partial sums $\{\sum_{j=n-i+1}^{n} x_j\}_{i=1}^{n}, \{\sum_{j=n-i+1}^{n} x_j^2\}_{i=1}^{n}$ and we can also actually compute all of the averages $\bar{y}_{1:n}, \bar{y}_{2:n}, \ldots, \bar{y}_{n:n}$ and similarly $\{\sum_{j=n-i+1}^{n} \bar{y}_{n-i+1:n}^2\}_{i=1}^{n}$. The only difficult one here is computing all of the $\bar{y}$s but this can be done by just dividing the partial sums $\sum_{j=n-i+1}^{n} x_j$ by i each time, so it is just as hard as computing the partial sums.

Once you have precomputed all of these quantities, which takes $O(n)$ time, evaluating each term in the minimum now takes $O(1)$ time, since it simply requires a constant number of lookups into precomputed information. This solving each subproblem now takes $O(n)$ time and we have brought the overall complexity down to $O(n^2)$.

5. **(0 points).**

References

[1] Donald E. Knuth. Optimum Binary Search Trees. *Acta informatica*. 1971.