CS/ECE 374 ✧ Spring 2017

Homework 5 Solutions

Due Wednesday, March 8, 2017 at 10am

---

1. Let $w \in \Sigma^*$ be a string. We say that $u_1, u_2, \ldots, u_h$ where each $u_i \in \Sigma^*$ is a valid split of w iff $w = u_1 u_2 \ldots u_h$ (the concatenation of $u_1, u_2, \ldots, u_h$). Given a valid split $u_1, u_2, \ldots, u_h$ of w we define its ℓ_3 measure as $\sum_{i=1}^{h} |u_i|^3$.

   Given a language $L \subseteq \Sigma^*$ a string $w \in L^*$ iff there is a valid split $u_1, u_2, \ldots, u_h$ of w such that each $u_i \in L$; we call such a split an L-valid split of w. Assume you have access to a subroutine IsStringInL(x) which outputs whether the input string x is in L or not. To evaluate the running time of your solution you can assume that each call to IsStringInL() takes constant time.

   Describe an efficient algorithm that given a string w and access to a language L via IsStringInL(x) outputs an L-valid split of w with minimum ℓ_3 measure if one exists.

   Solution: For a valid split $w_1, \ldots, w_k$ of w, let $\text{cost}(w_1, \ldots, w_k)$ be the l_3 measure of this split and let $\text{Min}l_3(w)$ be the measure of an optimal split of a w, or ∞ if no such split exists. In order to describe this problem using a dynamic program, we seek a suitable recursive definition for this function. Towards that, we prove the following:

   Lemma 1.1. *If $w = w_1 x$ where $w_1 \in L$ and $x \in \Sigma^*$, then $\text{Min}l_3(w) \leq |w_1|^3 + \text{Min}l_3(x)$.*

   Proof: Let $w_2, \ldots, w_k$ be an optimal valid split of x. Then $w_1, \ldots, w_k$ is a valid split of w. Thus, by definition, $|w_1|^3 + \text{Min}l_3(x) = \text{cost}(w_2, \ldots, w_k) = \text{cost}(w_1, \ldots, w_k) \geq \text{Min}l_3(w)$. □

   Lemma 1.2. *If $w_1, w_2, \ldots, w_k$, where $w_1, \ldots, w_k \in L$, is an optimal valid split for w, then $w_2, \ldots, w_k$ is an optimal valid split for $w_2 w_3 \cdots w_k$.*

   Proof: Let $x = w_2 w_3 \cdots w_k$. If $w_2, \ldots, w_k$ is not an optimal valid split of x, then $\text{Min}l_3(x) < \text{cost}(w_2, \ldots, w_k)$. Therefore, $\text{Min}l_3(w) = \text{cost}(w_1, \ldots, w_k) = |w_1|^3 + \text{cost}(w_2, \ldots, w_k) > |w_1|^3 + \text{Min}l_3(x)$. But by Lemma 1.1, $\text{Min}l_3(w) \leq |w_1|^3 + \text{Min}l_3(x)$. This contradiction proves that $w_2, \ldots, w_k$ has to be an optimal valid split of x. □

   Corrolary 1.3. $\text{Min}l_3(w) = \min\left\{|w_1|^3 + \text{Min}l_3(x) \mid w_1 \in L, x \in \Sigma^*, w = w_1 x\right\}$.

   Using Corallary 1.3, we give the following recursive definition of $\text{Min}l_3$:

$$\text{Min}l_3(\epsilon) = 0$$
$$\text{Min}l_3(w) = \min\left\{|w_1|^3 + \text{Min}l_3(x) \mid w_1 \in L, x \in \Sigma^*, w = w_1 x\right\}$$

   We implement this recursive definition using the standard dynamic programming Algorithm techniques, as follows. Let w be the input string, of length n. Our Algorithm maintains an array of length $n+1$, named $M[1..(n+1)]$. The index i of this array is going to hold the value of $\text{Min}l_3(w[i..n])$, with the understanding that $w[(n+1)..n] = \epsilon$. Using Corollary 1.3, we can compute the value $M[i]$ using only values of $M[j]$ for $j > i$. When M is computed, the value of $M[1]$ is by definition the cost of an optimal valid split of w.

```
MinimumL3(w[1..n]):
  M[n+1] ← 0
  for i ← n down to 1
      M[i] ← ∞
      for k ← i to n
          if IsStringInL(w[i..k]) and (k−i+1)^3 + M[k+1] < M[i]
              M[i] ← (k−i+1)^3 + M[k+1]
  return M[1]
```

The outer loop in this Algorithm iterates n times. In each iteration of the outer loop, the inner loop iterates at most n times. Every line of this Algorithm can be executed in constant time. Thus, the total running of this Algorithm is bounded by $O(n^2)$.

Rubric: Standard dynamic programming rubric.

2. Recall that a *palindrome* is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

   Any string can be decomposed into a sequence of palindrome substrings. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (among many others):

   BUB • BASEESAB • ANANA

   B • U • BB • A • SEES • ABA • NAN • A

   B • U • BB • A • SEES • A • B • ANANA

   B • U • B • B • A • S • E • E • S • A • B • ANA • N • A

   Describe and analyze an efficient algorithm that given a string w and an integer k decides whether w can be split into palindromes each of which is of length at least k. For example, given the input string BUBBASEESABANANA and 3 your algorithm would answer yes because one can find a split BUB • BASEESAB • ANANA. The answer should be no if we set $k = 4$. Note that the answer is always yes for $k = 1$.

   Solution: Let $A[1..n]$ be the input string. We define two functions:

   - $IsPal?(i, j)$ is $True$ if the substring $A[i..j]$ is a palindrome and $False$ otherwise.
   - $KPals(m)$ returns $True$ if the array $A[m..n]$ can be decomposed into palindromes of length atleast k each.

   The problem asks for an algorithm to compute $KPals(1)$.

   Every string with length at most 1 is a palindrome. A string of length 2 or more is a palindrome if and only if its first and last characters are equal and the rest of the string is a palindrome. Thus:

$$IsPal?(i,j) = \begin{cases} \text{True} & \text{if } i \le j \\ (A[i] = A[j]) \wedge IsPal?(i+1, j-1) & \text{otherwise} \end{cases}$$

   The empty string can be partitioned into zero(valid) palindromes. Otherwise, the "optimal" palindrome decomposition has at least one palindrome. If the first palindrome in the optimal decomposition of $A[1..n]$ ends at index l, where $l \ge k$, the remainder must be the "optimal" decomposition for the remaining characters $A[l+1..n]$. The following recurrence considers all possible values of ℓ.

$$KPals(m) = \begin{cases} True & \text{if } m > n \\ True & \text{if } \exists\ \ell,\ KPal?(\ell) \wedge IsPal?(m, l-1) = true,\ n \ge l \ge m+k-1 \\ False & \text{otherwise} \end{cases}$$

   We can memoize the $IsPal?$ function into a two-dimensional array IsPal?$[1..n, 0..n]$. Each entry IsPal?$[i, j]$ depends only on IsPal?$[i+1, j-1]$. Thus, we can fill this array row-by-row, from row n to row 1.

```
COMPUTEISPAL?(A[1..n]):
  for i ← n down to 1
      IsPal?[i, i−1] ← TRUE
      IsPal?[i, i] ← TRUE
      for j ← i+1 to n
          if A[i] = A[j]
              IsPal?[i, j] ← IsPal?[i+1, j−1]
          else
              IsPal?[i, j] ← False
```

This algorithm clearly runs in $O(n^2)$ time. Notice that we aren't returning anything; we're just memoizing the function for use by the main algorithm.

We can memoize the KPALS function into a one-dimensional array KPALS$[1..n]$. Each entry KPALS[m] depends only on entries KPALS$[l+1]$ with $l \geq m$. Thus, we can fill this array from index n down to 1.

```
KPALS(A[1..n]):
  COMPUTEISPAL?(A[1..n])
  KPals[n+1] ← True
  for m ← n down to 1
      KPals[m] ← False
      for l ← (m+k−1) to n
          if IsPal?[m, l−1] and KVal[l]
              KPals[m] ← True
return KPals[1]
```

The subroutine COMPUTEISPAL? runs in $O(n^2)$ time, and the rest of the algorithm also clearly runs in $O(n^2)$ time. So the total running time is $O(n^2)$.

If we had used the obvious iterative algorithm to test whether each substring is a palindrome, instead of precomputing the array ISPAL?, our algorithm would have run in $O(n^3)$ time.

■

Rubric: Standard dynamic programming rubric. This is more detail than necessary for full credit. Max 8 points for an $O(n^3)$-time algorithm; scale partial credit.

3. The McKing chain wants to open several restaurants along Red street in Shampoo-Banana. The possible locations are at $L_1, L_2, \ldots, L_n$ where L_i is at distance m_i meters from the start of Red street. Assume that the street is a straight line and the locations are in increasing order of distance from the starting point (thus $0 \leq m_1 < m_2 < \ldots < m_n$). McKing has collected some data indicating that opening a restaurant at location L_i will yield a profit of p_i independent of where the other restaurants are located. However, the city of Shampoo-Banana has a zoning law which requires that any two McKing locations should be D or more meters apart. Describe an algorithm that McKing can use to figure out the maximum profit it can obtain by opening restaurants while satisfying the city's zoning law.

Solution: Let $F(i)$ be the maximum profit we can make considering only the locations $L_i \ldots L_n$. For each $F(i)$, we can decide to: (1) select the current restaurant L_i, adding up its profit p_i and the maximum profit we can make from those restaurants starting at the next available location that is at least D distance away from L_i, or (2) skip L_i and consider the maximum profit we can make from those restaurants starting at location L_{i+1}. Then, we take the maximum of the two cases:

$$F(i) = \begin{cases} 0 & i > n \\ \max\{p_i + F(next(i)), F(i+1)\} & \text{otherwise} \end{cases}$$

where $next(i) = \min\{j : m_j \geq m_i + D\}$ is the next available location that is at least D distance away from the current location at L_i. The pseudocode is given as follows.

```
McKing(m[1..n], p[1..n]):
  next[1..n] ← ComputeNext(m[1..n])
  F[n+1] ← 0
  for i ← n downto 1
      F[i] ← max(F[i+1], p[i] + F[next[i]])
  return F[1]

ComputeNext(m[1..n]):
  i ← 1, j ← 1
  while i ≤ n
      while j ≤ n and m_j < m_i + D
          j ← j + 1.
      next[i] ← j.
      i ← i + 1
  return next[1..n]
```

There are $O(n)$ distinct subproblems. If *next* was preprocessed, each subproblem takes $O(1)$ time, so the running time is $O(n)$. Without the preprocessing step, there will be $O(n)$ work for each sub-problem and the running time will be $O(n^2)$. ■

Rubric: Standard dynamic programming rubric.

Consider a scale of 10 points. The English description requirement was relaxed and −2 points if the description is missing from a fairly obvious correct solution. No partial credit given if no English description is provided as it is impossible to gauge what the approach was trying to accomplish.

We did not penalize for not preprocessing *next*. However, if without preprocessing *next* the running time is incorrectly reported as $O(n)$, then the students lose 1 point for reporting incorrect running time.