CS/ECE 374 ✧ Spring 2021

Homework 5

Due Thursday, March 18, 2021 at 10am

Groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

The following unnumbered problems are not for submission or grading. No solutions for them will be provided but you can discuss them on Piazza.

- Problems in Jeff's notes on dynamic programming. In particular, Probs 1, 2, 3, 5, 6.
- Problems in Dasgupta etal book Chapter 6. In particular Probs 1, 2
- Problems in Kleinberg-Tardos book Chapter 6. Problems 1, 2, 7.

0. **Not to submit** 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_2 measure as $\sum_{i=1}^{h} |u_i|^2$.

   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_2 measure if one exists.

1. Given a graph $G = (V, E)$ a vertex cover of G is a subset $S \subseteq V$ of vertices such that for every edge $(u, v) \in E$, u or v is in S. The goal in the minimum vertex cover problem is to find a vertex cover S of smallest size. In some cases vertices may have non-negative weights $w : V \to \mathbb{Z}_+$ and the goal is to find a vertex cover of minimum weight. You can find some examples and discussion at the following Wikipedia link https://en.wikipedia.org/wiki/Vertex_cover. Describe a *recursive* algorithm that given a graph $G = (V, E)$ and weights $w(v), v \in V$ outputs a vertex cover of G with minimum weight. Do not worry about the running time.

2. Let Σ be a finite alphabet and let L_1 and L_2 be two languages over Σ. Assume you have access to two routines IsStringIn$L_1(u)$ and IsStringIn$L_2(u)$. The former routine decides whether a given string u is in L_1 and the latter whether u is in L_2. Using these routines as black boxes describe an efficient algorithm that given an arbitrary string $w \in \Sigma^*$ decides whether $w \in (L_1 + L_2)^*$. To evaluate the running time of your solution you can assume that calls to IsStringInL_1() and IsStringInL_2() take constant time.

3. 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

   Since a given string w can always be decomposed to palindromes we may want to find a decomposition that optimizes some objective. Here are two objectives. The first objective is to decompose w into the smallest number of palindromes. A second objective is to find a decomposition such that each palindrome in the decomposition has length at least k where k is some given input parameter. Both of these can be cast as special cases of an abstract problem. Suppose we are given a function called $cost()$ that takes a positive integer h as input and outputs an integer cost(h). Given a decomposition of w into $u_1, u_2, \ldots, u_r$ (that is, $w = u_1 u_2 \ldots u_r$) we can define the cost of the decomposition as $\sum_{i=1}^{r} \text{cost}(|u_i|)$.

   For example if we define cost(h) $= 1$ for all $h \geq 1$ then finding a minimum cost palindromic decomposition of a given string w is the same as finding a decomposition of w with as few palindromes as possible. Suppose we define cost() as follows: cost(h) $= 1$ for $h < k$ and cost(h) $= 0$ for $h \geq k$. Then finding a minimum cost palindromic decomposition would enable us to decide whether there is a decomposition in which all palindromes are of length at least k; it is possible iff the minimum cost is 0.

   Describe an efficient algorithm that given black box access to a function cost(), and a string w, outputs the value of a minimum cost palindromic decomposition of w.

4. **Not to submit:** 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.

Solved Problem

5. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of

the strings BANANA and ANANAS in several different ways.

$$\texttt{BANANA}_{\texttt{ANANAS}} \qquad \texttt{BAN}_{\texttt{ANA}}\texttt{ANA}_{\texttt{NAS}} \qquad \texttt{B}_{\texttt{AN}}\texttt{AN}_{\texttt{A}}\texttt{A}_{\texttt{NA}}\texttt{NA}_{\texttt{S}}$$

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

$$\texttt{PRO}^{\texttt{D}}\texttt{G}^{\texttt{Y}}\texttt{R}^{\texttt{NAM}}\texttt{AMMI}^{\texttt{I}}\texttt{N}^{\texttt{C}}\texttt{G} \qquad \texttt{DY}^{\texttt{PRO}}\texttt{N}^{\texttt{G}}\texttt{A}^{\texttt{R}}\texttt{M}^{\texttt{AMM}}\texttt{I}^{\texttt{IC}}\texttt{ING}$$

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

Solution: We define a boolean function $Shuf(i, j)$, which is TRUE if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. This function satisfies the following recurrence:

$$Shuf(i,j) = \begin{cases} \text{TRUE} & \text{if } i = j = 0 \\ Shuf(0, j-1) \wedge (B[j] = C[j]) & \text{if } i = 0 \text{ and } j > 0 \\ Shuf(i-1, 0) \wedge (A[i] = C[i]) & \text{if } i > 0 \text{ and } j = 0 \\ \big(Shuf(i-1, j) \wedge (A[i] = C[i+j])\big) & \\ \quad \vee \big(Shuf(i, j-1) \wedge (B[j] = C[i+j])\big) & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

We need to compute $Shuf(m, n)$.

We can memoize all function values into a two-dimensional array $Shuf[0..m][0..n]$. Each array entry $Shuf[i, j]$ depends only on the entries immediately below and immediately to the right: $Shuf[i-1, j]$ and $Shuf[i, j-1]$. Thus, we can fill the array in standard row-major order. The original recurrence gives us the following pseudocode:

```
SHUFFLE?(A[1..m], B[1..n], C[1..m+n]):
  Shuf[0,0] ← TRUE
  for j ← 1 to n
      Shuf[0,j] ← Shuf[0,j-1] ∧ (B[j] = C[j])
  for i ← 1 to n
      Shuf[i,0] ← Shuf[i-1,0] ∧ (A[i] = B[i])
      for j ← 1 to n
          Shuf[i,j] ← FALSE
          if A[i] = C[i+j]
              Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i-1,j]
          if B[i] = C[i+j]
              Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i,j-1]
  return Shuf[m,n]
```

The algorithm runs in $O(mn)$ *time*. ■

Rubric: Max 10 points: Standard dynamic programming rubric. No proofs required. Max 7 points for a slower polynomial-time algorithm; scale partial credit accordingly.

Rubric: Standard dynamic programming rubric For problems worth 10 poins:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative psuedocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).