CS/ECE 374 Sec A ✧ Spring 2023

Homework 6

Due Wednesday, March 8, 2023 at 10am

---

1. We consider two problems on strings that connect dynamic programming to some of the previous material on languages and regular expressions.

   (a) 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 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. Given $w \in \Sigma^*$ we would like to find an L-valid split of minimum *cost* if one exists. We define the cost of a split $u_1, u_2, \ldots, u_h$ as $\sum_{i=1}^{h} cost(|u_i|)$ where $cost : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ is a function that is specified as part of the problem and you can assume that $cost(0) = 0$. You should assume black box access to a subroutine $cost$ that takes as input a non-negative integer p as input and outputs $cost(p)$ which we also assume is non-negative. One can model different objectives via different functions. For instance, if we define cost(p) $= 1$ for all $p \geq 1$ then finding a minimum cost split is the same as finding the split with fewest strings in the split. Suppose we define cost() as follows: cost(p) $= 0$ if $p \leq k$ and cost(p) $= \infty$ for $p > k$. Then finding a minimum cost split would enable us to decide whether there is a split in which each string in the split is of length at most k. A final example is a cost function where we set $cost(p) = 0$ if $p \geq k$ and $cost(p) = \infty$ for $1 \leq p < k$. This models the situation where we want to find a split in which each string in the split is of length at least k.

   Describe an efficient algorithm that given black box access to the function cost() and IsStringInL(), and a string w, outputs the value of a minimum cost L-split. If $w \notin L^*$ your algorithm should report this. To evaluate the running time of your solution you can assume that each call to IsStringInL(x) takes $|x|$ time and that $cost()$ takes $O(1)$ time.

   (b) Let r be a regular expression and w be a string over a finite alphabet, say $\{0, 1\}$. We would like an algorithm that decides whether $w \in L(r)$. Describe a *recursive* algorithm for this problem. The input to your algorithm is the string w and a recursive description of r (note that r is either a base case, $s + t$ or st or $(s)^*$ for smaller regular expressions). For example, $w = 10101010111111000000111111$ and $r = (0 + 111)^*(11 + 1000) + (11100 + 01)(10 + 11)^*$. For sake of consistency use the template IsStringInRegExp(w, r) for your function. In your pseudocode you can use conditionals such as "If $r = s + t$ then" to avoid the low-level details of how the recursive definition of r is formally specified. You do not need to worry about the running time of the algorithm or analyze it but the work in your function should be efficient (polynomial-time) assuming recursive calls take constant time — in particular, this precludes brute force solutions that do exponential work by enumeration or other type of ideas. You do not have to answer this part but would your algorithm run in polynomial time with automatic memoization?

2. Problem 16 in Jeff's chapter on DP. https://jeffe.cs.illinois.edu/teaching/algorithms/book/03-dynprog.pdf. A clarification on the problem: Mr. Fox has to go straight through the obstacle course and visit each booth exactly once on their way.

3. **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.

BANANA**ANANAS** BAN**ANA**ANA**NAS** B**AN**AN**A**A**NA**NA**S**

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

PRO**D**G**Y**R**NAM**AMMI**I**N**C**G DY**PRO**N**G**A**R**M**AMM**IC**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 $\mathit{Shuf}(i, j)$, 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:

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

We need to compute $\mathit{Shuf}(m, n)$.

We can memoize all function values into a two-dimensional array $\mathit{Shuf}[0..m][0..n]$. Each array entry $\mathit{Shuf}[i, j]$ depends only on the entries immediately below and immediately to the right: $\mathit{Shuf}[i-1, j]$ and $\mathit{Shuf}[i, j-1]$. 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.

Standard dynamic programming rubric. 10 points divided as follows

- 3 points for a clear and correct English description of the recursive function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.)
  - -1 for naming the function "OPT" or "DP" or any single letter.
  - – No credit if the description is inconsistent with the recurrence.
  - – No credit if the description does not explicitly describe how the function value depends on the named input parameters.
  - – No credit if the description refers to internal states of the eventual dynamic programming algorithm, like "the current index" or "the best score so far". The function must have a well-defined value that depends *only* on its input parameters (and constant global variables).
  - – An English explanation of the *recurrence* or *algorithm* does not qualify. We want a description of *what* your function returns, not (here) an explanation of *how* that value is computed.
- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - 1 for base case(s). $-\frac{1}{2}$ for one *minor* bug, like a typo or an off-by-one error.
  - 3 for recursive case(s). -1 for each *minor* bug, like a typo or an off-by-one error.
  - -2 for greedy optimizations without proof, even if they are correct.
  - – **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 3 points for iterative details
  - + 1 for describing an appropriate memoization data structure
  - + 1 for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested for loops, be sure to specify the nesting order.
  - + 1 for correct 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 specifically says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative pseudocode 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 ***do*** still need and English description of the underlying recursive function (or equivalently, the contents of the memoization structure). ***Perfectly correct iterative pseudocode, with no explanation or time analysis, is worth at most 6 points out of 10.***
- Partial credit for incomplete solutions depends on the running time of the ***best possible*** completion (up to the target running time). For example, consider a solution that contains *only* a clear English description of a function, with no recurrence or iterative details. If the described function *can* be developed into an algorithm with the target running time, the solution is worth 3 points; however, if the function leads

to an algorithm that is slower than the target time by a factor of n, the solution could be worth only 2 points ($=$ 70% of 3, rounded).