CS/ECE 374 ✧ Fall 2016

Homework 5

Due Tuesday, October 11, 2016 at 8pm

---

1. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \le n$.

   (a) Describe and analyze an algorithm to determine whether X occurs as two ***disjoint*** subsequences of Y, where "disjoint" means the two subsequences have no indices in common. For example, the string PPAP appears as two disjoint subsequences in the string $\underline{\text{P}}\text{EN}\underline{\text{P}}\text{INE}\underline{\text{A}}\overline{\text{P}}\overline{\text{P}}\text{LE}\overline{\text{A}}\underline{\text{P}}\text{PLE}\overline{\text{P}}\text{EN}$, but the string PEEPLE does not.

   (b) Describe and analyze an algorithm to compute the number of occurrences of X as a subsequence of Y. For example, the string PPAP appears exactly 23 times as a subsequence of the string PENPINEAPPLEAPPLEPEN. If all characters in X and Y are equal, your algorithm should return $\binom{n}{k}$. For purposes of analysis, assume that each arithmetic operation takes $O(1)$ time.

2. You are driving a bus along a long straight highway, full of rowdy, hyper, thirsty students and an endless supply of soda. Each minute that each student is on your bus, that student drinks one ounce of soda. Your goal is to drive all students home, so that the total volume of soda consumed by the students is as small as possible.

   Your bus begins at an exit (probably not at either end) with all students on board and moves at a constant speed of 37.4 miles per hour. Each student needs to be dropped off at a highway exit. You may reverse directions as often as you like; for example, you are allowed to drive forward to the next exit, let some students off, then turn around and drive back to the previous exit, drop more students off, then turn around again and drive to further exits. (Assume that at each exit, you can stop the bus, drop off students, and if necessary turn around, all instantaneously.)

   Describe an efficient algorithm to take the students home so that they drink as little soda as possible. Your algorithm will be given the following input:

   - A sorted array $L[1..n]$, where $L[i]$ is the Location of the ith exit, measured in miles from the first exit; in particular, $L[1] = 0$.
   - An array $N[1..n]$, where $N[i]$ is the Number of students you need to drop off at the ith exit
   - An integer *start* equal to the index of the starting exit.

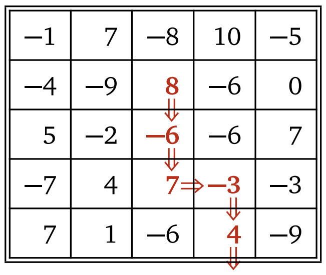
   Your algorithm should return the total volume of soda consumed by the students when you drive the optimal route.[1]

---

[1]Non-US students are welcome to assume kilometers and liters instead of miles and ounces. Late 18th-century French students are welcome to use decimal minutes.

3. *Vankin's Mile* is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

   For example, given the grid below, the player can score $8-6+7-3+4=10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)



−1	7	−8	10	−5
−4	−9	**8** ⇓	−6	0
5	−2	**−6** ⇓	−6	7
−7	4	**7** ⇒	**−3** ⇓	−3
7	1	−6	**4** ⇓	−9

   (a) Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

   (b) In the Canadian version of this game, appropriately called *Vankin's Kilometer*, the player can move the token either one square down, one square right, *or one square left* in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Kilometer, given the $n \times n$ array of values as input.[2]

[2]If we also allowed upward movement, the resulting game (Vankin's Fathom?) would be NP-hard.

Solved Problem

4. 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.

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

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

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

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:

$$\mathit{Shuf}(i,j) = \begin{cases} \textsc{True} & \text{if } i=j=0 \\ \mathit{Shuf}(0,j-1) \land (B[j]=C[j]) & \text{if } i=0 \text{ and } j>0 \\ \mathit{Shuf}(i-1,0) \land (A[i]=C[i]) & \text{if } i>0 \text{ and } j=0 \\ \big(\mathit{Shuf}(i-1,j) \land (A[i]=C[i+j])\big) & \\ \qquad \lor \big(\mathit{Shuf}(i,j-1) \land (B[j]=C[i+j])\big) & \text{if } i>0 \text{ and } j>0 \end{cases}$$

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?($A[1..m]$, $B[1..n]$, $C[1..m+n]$):
  $Shuf[0,0] \gets$ TRUE
  for $j \gets 1$ to $n$
      $Shuf[0,j] \gets Shuf[0,j-1] \land (B[j] = C[j])$
  for $i \gets 1$ to $n$
      $Shuf[i,0] \gets Shuf[i-1,0] \land (A[i] = B[i])$
      for $j \gets 1$ to $n$
          $Shuf[i,j] \gets$ FALSE
          if $A[i] = C[i+j]$
              $Shuf[i,j] \gets Shuf[i,j] \lor Shuf[i-1,j]$
          if $B[i] = C[i+j]$
              $Shuf[i,j] \gets Shuf[i,j] \lor Shuf[i,j-1]$
  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. 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).