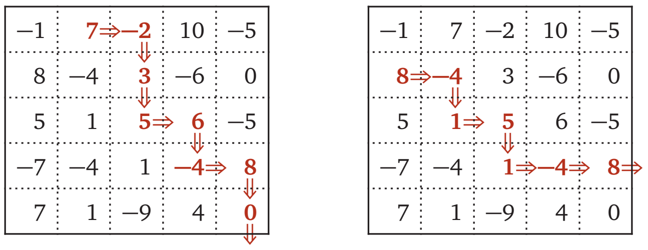
CS/ECE 374 A ✧ Fall 2021
Homework 6
Due Tuesday, October 12, 2021 at 8pm Central Time

---

1. *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 shown below, the player can score $7-2+3+5+6-4+8+0 = 23$ points by following the path on the left, or they can score $8-4+1+5+1-4+8 = 15$ points by following the path on the right.



   (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) A variant called *Vankin's Niknav* adds an additional constraint to Vankin's Mile: *The sequence of values that the token touches must be a* ***palindrome***. Thus, the example path on the right is valid, but the example path on the left is not. Describe and analyze an efficient algorithm to compute the maximum possible score for an instance of Vankin's Niknav, given the $n \times n$ array of values as input.

2. A ***snowball*** is a poem or sentence that starts with a one-letter word, where each later word is one letter longer than its predecessor. For example:

   I am the fire demon, moving castles: Calcifer!

   Snowballs, sometimes also known as ***chaterisms*** or ***rhopalisms***, are one of many styles of constrained writing practiced by OuLiPo, a loose gathering of writers and mathematicians, founded in France in 1960 but still active today.

   Describe and analyze an algorithm to extract the longest snowball hidden in a given string of text. You are given an array $T[1..n]$ of English letters as input. Your goal is to find the longest possible sequence of disjoint substrings of T, where the ith substring is an English word of length i. Your algorithm should return the number of words in this sequence.

   Your algorithm will call the library function IsWord, which takes a string w as input and returns True if and only if w is an English word. IsWord(w) runs in $O(|w|)$ time.

   For example, given the input string

   `EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGETABLECALCIFER`

   your algorithm should return the integer 8:

   $$\texttt{EVEN}\underline{\texttt{I}}\texttt{FYOU}\underline{\texttt{AM}}\underline{\texttt{THE}}\texttt{AREMY}\underline{\texttt{FIRE}}\texttt{LEAST}\underline{\texttt{DEMON}}\texttt{FAVORITE}\underline{\texttt{MOVING}}\underline{\texttt{CASTLES}}\texttt{VEGETABLE}\underline{\texttt{CALCIFER}}$$

Standard dynamic programming rubric. For problems worth 10 points:

- 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.)
  - 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.

  − 1 for naming the function "OPT" or "DP" or any single letter.

- 4 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.

  + 1 for base case(s). −½ 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.***
- Official solutions will provide target time bounds. Faster algorithms are worth more points, and slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n in either direction. Partial credit is scaled to the new maximum score, and all points above 10 (for algorithms that are faster than our target time bound) are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we do include them, significantly more students submit incorrect algorithms with the target running time (earning 0/10) instead of correct algorithms that are slower than the target (earning 7/10).
- 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).

Solved Problem

3. 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.

   BANANAANANAS   BANANAANANAS   BANANAANANAS

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

   PRODGYRNAMAMMIINCG   DYPRONGARMAMMICING

   (a) 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 $B[1..j]$. We need to compute $\mathit{Shuf}(m,n)$. The function *Shuf* satisfies the following recurrence:

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

We can memoize this function 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.

```
IsShuffle?($A[1..m]$, $B[1..n]$, $C[1..m+n]$):
    $Shuf[0,0] \leftarrow$ True
    for $j \leftarrow 1$ to $n$
        $Shuf[0,j] \leftarrow Shuf[0,j-1] \wedge (B[j] = C[j])$
    for $i \leftarrow 1$ to $n$
        $Shuf[i,0] \leftarrow Shuf[i-1,0] \wedge (A[i] = B[i])$
        for $j \leftarrow 1$ to $n$
            $Shuf[i,j] \leftarrow$ False
            if $A[i] = C[i+j]$
                $Shuf[i,j] \leftarrow Shuf[i-1,j]$
            if $B[i] = C[i+j]$
                $Shuf[i,j] \leftarrow Shuf[i,j] \vee Shuf[i,j-1]$
    return $Shuf[m,n]$
```

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

Rubric: 5 points, standard dynamic programming rubric. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

(b) Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine *the number of different ways* that A and B can be shuffled to obtain C.

Solution: Let $\#Shuf(i,j)$ denote the number of different ways that the prefixes $A[1..i]$ and $B[1..j]$ can be shuffled to obtain the prefix $C[1..i+j]$. We need to compute $\#Shuf(m,n)$.

The $\#Shuf$ function satisfies the following recurrence. Here I am using Iverson bracket notation to convert booleans to integers: For any proposition P, the expression $[P]$ is equal to 1 if P is true and 0 if P is false.

$$\#Shuf(i,j) = \begin{cases} 1 & \text{if } i=j=0 \\ \#Shuf(0,j-1)\cdot\big[B[j]=C[j]\big] & \text{if } i=0 \text{ and } j>0 \\ \#Shuf(i-1,0)\cdot\big[A[i]=C[i]\big] & \text{if } i>0 \text{ and } j=0 \\ \big(\#Shuf(i-1,j)\cdot\big[A[i]=C[i]\big]\big) & \\ \qquad +\big(\#Shuf(i,j-1)\cdot\big[B[j]=C[j]\big]\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $\#Shuf[0..m][0..n]$. As in part (a), we can fill the array in standard row-major order.

```
NumShuffles($A[1..m]$, $B[1..n]$, $C[1..m+n]$):
    $\#Shuf[0,0] \leftarrow 1$
    for $j \leftarrow 1$ to $n$
        $\#Shuf[0,j] \leftarrow 0$
        if $(B[j] = C[j])$
            $\#Shuf[0,j] \leftarrow \#Shuf[0,j-1]$
    for $i \leftarrow 1$ to $n$
        $\#Shuf[0,j] \leftarrow 0$
        if $(A[i] = B[i])$
            $\#Shuf[0,j] \leftarrow \#Shuf[i-1,0]$
    for $j \leftarrow 1$ to $n$
        $\#Shuf[i,j] \leftarrow 0$
        if $A[i] = C[i+j]$
            $\#Shuf[i,j] \leftarrow \#Shuf[i-1,j]$
        if $B[i] = C[i+j]$
            $\#Shuf[i,j] \leftarrow \#Shuf[i,j] + \#Shuf[i,j-1]$
    return $Shuf[m,n]$
```

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

Rubric: 5 points, standard dynamic programming rubric. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.