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.

   (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.

Solution: Let $A[1..n, 1..n]$ denote the input array. Let $VMile(i, j)$ denote the maximum score that can be obtained starting at row i and column j.

We need the maximum of $VMile(i, j)$ over all indices $1 \le i, j \le n$.

The *VMile* function obeys the following recurrence:

$$VMile(i,j) = \begin{cases} 0 & \text{if } i > n \text{ or } j > n \\ A[i,j] + \max\{VMile(i+1,j),\ VMile(i,j+1)\} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $VMile[1..n+1, 1..n+1]$.

We can fill this array with two nested loops considering both i and j in decreasing order. (It doesn't matter which index we use in which loop.)

```
VankinsMile(A[1..n, 1..n]:
  maxScore ← −∞
  for i ← n + 1 down to 1
      VMile[i, n + 1] ← 0

  for j ← n down to 1
      VMile[n + 1, j] ← 0
      for i ← n down to 1
          VMile[i, j] ← A[i, j] + max{VMile[i + 1, j], VMile[i, j + 1]}
          maxScore ← max{maxScore, VMile[i, j]}
  return maxScore
```

The resulting algorithm runs in $O(n^2)$ *time*. ■

Rubric: 5 points: standard dynamic programming rubric

(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***. 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.

Solution: Let $A[1..n, 1..n]$ denote the input array. Let $VsV(i, j, i', j')$ denote the maximum score that can be obtained starting at row i and column j, ending at row i' and column j', and traversing a palindrome path.

We need the maximum of $VsV(i, j, i', j')$ over all indices $1 \le i, j, i', j' \le n$ such that $\max\{i', j'\} = n$.

The VsV function obeys the following recurrence:

$$VsV(i, j,\ i', j') = \begin{cases} -\infty & \text{if } A[i,j] \neq A[i',j'] \\ -\infty & \text{if } i > i' \text{ or } j > j' \\ A[i,j] & \text{if } i = i' \text{ and } j = j' \\ 2 \cdot A[i,j] & \text{if } i + 1 = i' \text{ and } j = j' \\ 2 \cdot A[i,j] & \text{if } i = i' \text{ and } j + 1 = j' \\ 2 \cdot A[i,j] + \max \begin{Bmatrix} VsV(i+1, j,\ i'-1, j') \\ VsV(i+1, j,\ i', j'-1) \\ VsV(i, j+1,\ i'-1, j') \\ VsV(i, j+1,\ i', j'-1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

(All cases except the first require $A[i, j] = A[i', j']$.)

We can memoize this function into a four-dimensional array $VsV[1..n, 1..n, 1..n, 1..n]$.

We can fill this array by decreasing i, decreasing j, increasing i', and increasing j'. (The nesting order of the loops doesn't matter.)

We can fill the array in $\boldsymbol{O(n^4)}$ ***time***, after which finding the largest entry of the form $VsV[i, j, n, j']$ or $VsV[i, j, i', n]$ takes $O(n^3)$ additional time. ■

Rubric: 5 points: standard dynamic programming rubric

2. 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 cal the library function IsWord(w), 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.

Solution: For any integer i, let $Snowball(i,\ell)$ denote the number of words in the longest snowball in the suffix $T[i..n]$, where the first word in the snowball has length ℓ. We need to compute $Snowball(1,1)$.

The *Snowball* function satisfies the following recurrence:

$$Snowball(i,\ell) = \begin{cases} 0 & \text{if } i+\ell-1 > n \\ Snowball(i+1,\ell) & \text{if } \neg\text{IsWord}(T[i..i+\ell-1]) \\ \max\left\{\begin{array}{c} Snowball(i+1,\ell) \\ 1+Snowball(i+\ell,\ell+1) \end{array}\right\} & \text{otherwise} \end{cases}$$

A snowball of length ℓ contains exactly $\sum_{i=1}^{\ell} i = \ell(\ell+1)/2$ letters. Thus, if a snowball of length ℓ appears inside the array $T[1..n]$, we must have $\ell(\ell+1)/2 \le n$, which implies $\ell < \sqrt{2n}$.

Thus, we can memoize the *Snowball* function into a two-dimensional array $Snowball[1..n, 1..\sqrt{2n}]$.

We can fill this array by decreasing both i and ℓ in two nested loops; the nesting order does not matter. For each array entry we call IsWord on a string of length at most $\sqrt{2n}$, which takes $O(\sqrt{n})$ time. Since there are $O(n^{3/2})$ array entries, the entire algorithm runs in ***$O(n^2)$ time***.

```
MaxSnowball(T[1..n]):
  for ℓ ← 2√n down to 1
      for i ← n down to 1
          if i + ℓ − 1 > n
              Snowball[i, ℓ] ← 0
          else if ¬IsWord(T[i..i + ℓ − 1])      ⟨⟨O(ℓ) = O(√n) time⟩⟩
              Snowball[i, ℓ] ← Snowball[i + 1, ℓ]
          else
              Snowball[i, ℓ] ← max{ Snowball[i + 1, ℓ]
                                  { 1 + Snowball[i + ℓ, ℓ + 1] }
  return Snowball[1, 1]
```

(Without the observation that $\ell = O(\sqrt{n})$, the memoization array would have size $O(n^2)$ and filling each entry would take $O(n)$ time, for a total running time of $O(n^3)$.) ■

Rubric: 10 points; standard dynamic programming rubric. Max 8 points for $O(n^3)$ time.

Solution (greedy(!)): I claim that the following greedy algorithm computes the length of the longest snowball in T.

```
GreedySnowball(T[1..n]):
  i ← 1
  ℓ ← 1
  while i + ℓ − 1 ≤ n
      if IsWord(T[i..i + ℓ − 1])        ⟨⟨O(ℓ) = O(√n) time⟩⟩
          ⟨⟨g_ℓ ← i⟩⟩
          i ← i + ℓ
          ℓ ← ℓ + 1
      else
          i ← i + 1
  return ℓ − 1        ⟨⟨We didn't find a word of length ℓ⟩⟩
```

We can prove this greedy algorithm is correct using an exchange argument. Intuitively, starting with *any* snowball, replacing its k-letter word with any earlier k-letter word in T that starts after the snowball's $(k-1)$-letter word gives a new snowball of the same length. For example, starting with the example 7-word snowball

EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGE...,

replacing the 5-letter word DEMON with the earlier 5-letter word LEAST gives us another valid 7-word snowball:

EVENIFYOUAMTHEAREMYFIRELEASTDEMONFAVORITEMOVINGCASTLESVEGE....

Thus, choosing earlier words can't result in shorter snowballs.

More formally, let $(g_1, g_2, \ldots, g_L)$ be the sequence of starting indices for the greedy snowball; for any index $1 \le \ell \le L$, the ℓth word in the greedy snowball is $T[g_\ell .. g_\ell + \ell - 1]$. The greedy algorithm defines

$$g_\ell = \min\{i \mid i \ge g_{\ell-1} + \ell - 1 \text{ and } T[i..i+\ell-1] \text{ is a word}\}.$$

Every snowball starts with some number (possibly zero) of the same words as the greedy snowball. Consider an ***arbitrary*** snowball with starting indices

$$(g_1, g_2, \ldots, g_{\ell-1}, i_\ell, i_{\ell+1}, \ldots, i_m)$$

for some word-index ℓ. Replacing the ℓ-letter word $T[i_\ell .. i_\ell + \ell - 1]$ with the earlier ℓ-letter word $T[g_\ell .. g_\ell + \ell - 1]$ gives us another valid snowball of the same length, with starting indices

$$(g_1, g_2, \ldots, g_{\ell-1}, g_\ell, i_{\ell+1}, \ldots, i_m).$$

By induction, we can apply this exchange to all later words to obtain a snowball $(g_1, g_2, \ldots, g_m)$ that contains the first m words of the greedy snowball. We conclude that $m \le L$; the length m of our *arbitrary* snowball is *at most* the length L of the greedy snowball. Equivalently, the greedy snowball is optimal.

This argument crucially depends on the precise formulation of the snowball problem; the greedy algorithm is no longer correct even for very minor variants. For example:

- If we require each word in a snowball to be longer than the previous word, but not necessarily by exactly one letter, the greedy algorithm does not compute the longest snowball.
- If we define the value of a snowball as some function of the contents of the words (for example, the total number of vowels, or the total number of Scrabble points), instead of just their number, the greedy algorithm does not compute the most valuable snowball.

Any snowball of length ℓ contains exactly $\sum_{i=1}^{\ell} i = \ell(\ell+1)/2$ letters. Thus, if the input array $T[1..n]$ contains a snowball of length ℓ, we must have $\ell(\ell+1)/2 \le n$, which implies $\ell < \sqrt{2n}$. it follows that each call to IsWord runs in $O(\ell) = O(\sqrt{n})$ time. We conclude that GreedySnowball runs in at most $\boldsymbol{O(n^{3/2})}$ ***time***. ■

Rubric: Max 13 points = 7 for algorithm + 3 for correctness argument + 3 for time analysis. The intuitive exchange argument is sufficient for full credit.