dynamic programming **May 7, 2025**

Homework 5

Spring 2025 *COSC31: Algorithms*

1. **Bigdonald's**
Bigdonald's is considering opening a series of restaurants along Deep Valley Highway (DVH). The n possible locations are along a straight line and their distances from the start of DVH, in miles and in increasing order, are $m[1], m[2], \dots, m[n]$. The constraints are as follows:

- At each location, Bigdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is $p[i]$, where $p[i] > 0$ and $i \in \{1, 2, \dots, n\}$.
- Any two restaurants should be at least k miles apart, where k is a positive integer.

Design a dynamic programming algorithm of $O(n)$ time complexity and $O(n)$ space complexity that, given $k, n, m[1 \dots n]$, and $p[1 \dots n]$, outputs the maximum expected total profit (subject to the given constraints) and the locations where to open the restaurants to realize this maximum expected total profit. Analyze the time and space complexity of your algorithm.

Answer

Observations: Let mp represent the maximum profit. We observe that:

- The maximum profit at index i is either the maximum profit at $i-1$ or the sum of $p[i]$ and the maximum profit at the latest valid index j before i such that $m[i] - m[j] \geq k$ for $1 \leq j < i \leq n$. The valid index j must be the largest index satisfying this condition, with no other index r for $j < r < i$ such that $m[i] - m[r] \geq k$.
- These valid indices can be precomputed efficiently using a linear two-pointer scan, which allows us to define a function *validIndex* giving the valid j for each i.

Notations: To implement the above idea, we define:

- $validIndex[i]$: the largest index j such that $j < i$ and $m[i] - m[j] \geq k$.
- $mp(i)$: the maximum profit achievable considering locations 1 through i.

Recurrence: The maximum profit at index i can be computed as:

$$mp[i] = \max(mp[i-1],\ p[i] + mp[validIndex[i]])$$

Algorithm Bigdonald's

Preconditions: $n \geq 1$, $k \geq 1$, $m[1 \ldots n]$ is an array of possible locations of restaurants from the start of DVH in miles, where $m[i] < m[i+1]$ for $1 \leq i \leq n$, k is a positive integer of the minimum distance between two restaurants, and $p[1 \ldots n]$ is an array of expected profit from opening a restaurant at location i, where $p[i] > 0$ for $1 \leq i \leq n$.
Postconditions: Terminates, and returns the maximum expected total profit and the locations to open restaurants for maximum expected total profit.

```
Function Bigdonald($m[1 \ldots n], p[1 \ldots n], k$):
    set validIndex[$i$] to $-1$ for $1 \leq i \leq n$
    set mp[$i$] to 0 for $s1 \leq i \leq n$
    define a list of $locations \leftarrow []$
    slowPtr $\leftarrow$ 1
    fastPtr $\leftarrow$ 2

    while $fastPtr < n$ do
        while $slowPtr < fastPtr$ and $m[fastPtr] - m[slowPtr] \geq k$ do
            slowPtr $\leftarrow$ slowPtr + 1
        end
        validIndex[fastPtr] $\leftarrow$ slowPtr - 1
        fastPtr $\leftarrow$ fastPtr + 1
    end

    mp[1] $\leftarrow p[1]$
    append 1 to $locations$
    for $i \leftarrow 2$ to $n$ do
        mp[$i$] = max($mp[i-1], p[i] + mp[validIndex[i]]$)
        if $mp[i] > mp[i-1]$ then
            append $i$ to $locations$
        end
    end
    return ($mp[n], locations$)
```

Time complexity: The total time complexity is $O(2n) = O(n)$. In the first loop, since the slow pointer only moves forward and is not reset, the inner while loop runs in constant time per iteration of the outer loop. The second loop takes exactly $O(n)$
Space complexity: The space complexity is $O(3n) = O(n)$, we are using 3 arrays for which each can have at most n elements.

2. **Printing**
Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $\ell_1, \ell_2, ..., \ell_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters ℓ_k , which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra Design a dynamic programming algorithm of $O(n^2)$ time complexity and $O(n)$ space complexity that, given $n \geq 1$, $\ell[1...n]$, and $M \geq 1$, prints neatly a paragraph of n words of lengths given by the ℓ array.

Answer

<u>Clever observation</u>: For words up to index i, we consider all possible break points at index j $(1 \leq j \leq i)$, where words j through i would be placed on the current line. For each possible j, we compute the penalty of placing words j to i on one line (or ∞ if they do not fit). The dynamic programming step chooses the j that minimizes the total cost up to i, which consists of the optimal cost up to $j-1$ plus the penalty for words j to i. The corresponding j is stored in $breakPoint(i)$ to allow reconstruction of the optimal paragraph.
<u>Notation</u>: We we have the following:

- $\text{extraSpaces}(j,i) = M - i + j - \sum_{r=j}^{k} \ell_r$
- $\text{breakPoint}(j)$ - position with minimum penalty for printing words through some indices j to i for $1 \leq j \leq i \leq n$
- $\text{penalty}(j,i)$

$$\text{penalty}(j,i) = \begin{cases} 0, & \text{if } i = n \text{ (last line has no penalty)} \\ \left(\text{extraSpaces}(j,i)\right)^3, & \text{if extraSpaces}(j,i) \geq 0 \\ \infty, & \text{if extraSpaces}(j,i) < 0 \end{cases}$$

- $\text{costs}(j,i)$ - minimizes the costs of printing a sequence of characters in a new line.

Algorithm PrintNeatly

Preconditions: $n \geq 1, M \leq 1$, $\ell[1 \dots n]$ is array of real numbers representing lengths of words.
Postcondition: Terminates, and prints the paragraph of n words neatly

```
Function PrintNeatly ($\ell[1 \dots n], M$):
    totalChars[1] ← $\ell_1$
    for $i \leftarrow 2$ to $n$ do
        totalChars[$i$] ← totalChars[$i-1$] + $\ell[i]$

    penalty ← 0
    breakPoints ← []
    Initialize all costs[$1 \dots n$] with $\infty$
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow i$ to 1 do
            charLength ← totalChars[$i$] - totalChars[$j-1$]
            spaces ← $i - j$
            extraSpaces ← $M$ - charLength - spaces

            if $extraSpaces < 0$ then
                penalty ← $\infty$
            if $i = n$ then
                penalty ← 0
            else
                penalty ← (extraSpaces)$^3$
            if $costs[j-1] + penalty < costs[i]$ then
                costs[$i$] ← costs[$j-1$]+ penalty
                breakPoints[$i$] ← $j$
    printParagraph($1, n$, breakPoints)
```

Preconditions: $\ell, n \in \mathbb{N}$, $\ell \leq n$, breakPoints[$\ell \dots n$] is array of real numbers, where ℓ is the print starting point
Postcondition: Terminates, and prints the paragraph neatly

```
Function PrintParagraph (start, n, breakPoints):
    if $start > n$ then
        return
    endIndex ← breakPoints[$start$]
    prints words $start$ through $endIndex$ on line, then newline
    PrintParagraph ($endIndex + 1, n, breakPoints$)
```

Time complexity: We have $O(n^2) + O(n)$, hence the overall time complexity is $O(n^2)$
Space complexity: We have $O(3n)$ because of the 3 arrays to used hence $O(n)$

3. **Submatrix**
Let $A[1 \dots n, 1 \dots n]$ be an nxn matrix. A submatrix of A is the matrix constituted by the elements of A at $\{(i,j) | r \leq i \leq r', c \leq j \leq c'\}$, for any r, r', c, c' such that $1 \leq r \leq r' \leq n$ and $1 \leq c \leq c' \leq n$. We say this submatrix starts at (r, c) and has a dimension of $r' - r + 1 \dots c' - c + 1$.

Design a $O(n^2)$-time dynamic programming algorithm that, given $n > 0, 0 < k \leq n$, and a matrix $A[1 \dots n, 1 \dots n]$ of numbers, finds out a submatrix of A of dimension kxk whose sum of elements is a maximum. Your algorithm should return the sum of the elements of the submatrix and where the submatrix starts.

Answer

Clever observation: The number of valid positions to place a $k \times k$ submatrix is $(n-k+1)$ for both rows and columns. To compute the sum of any $k \times k$ submatrix efficiently without recomputing overlapping elements, we precompute a 2D prefix sum matrix S, where $S[i][j]$ stores the sum of elements in the rectangle from $(1,1)$ to (i,j).
Using this prefix sum matrix, the sum of any $k \times k$ submatrix starting at (r,c) can be computed in $O(1)$ time using inclusion-exclusion, allowing us to check all possible positions in $O(n^2)$ time overall.
Notation:

- $P(i,j)$: Prefix sum of matrix A, defined as:

$$P(i,j) = \begin{cases} \sum_{1 \le x \le i, 1 \le y \le j} A[x,y], & \text{if } i \le n \text{ and } j \le n \\ 0, & \text{otherwise} \end{cases}$$

- $M(n,k)$: Maximum sum of any $k \times k$ submatrix, defined as:

$$M(n,k) = \max_{\substack{1 \le i \le n-k+1 \\ 1 \le j \le n-k+1}} \Big(P(i+k-1, j+k-1) - P(i-1, j+k-1) - P(i+k-1, j-1) + P(i-1, j-1) \Big)$$

- (i^*, j^*): The starting coordinates of the submatrix that achieves $M(n,k)$.

Algorithm SubmatrixMaxSum

Preconditions: $n, k \in \mathbb{N}$, $n \geq 1, 0 < k \leq n$, $A[1 \ldots n, 1 \ldots]$ is an nxn matrix with integers.
Postcondition: Terminates, and returns the sum elememts of submatrix with max sum, and where the submatrix starts

```
Function SubmatrixMaxSum ($A[1 \ldots n1 \ldots n], k$):
    Initialize matrix $S[1 \ldots n \ldots n]$ with all zeros

    // row prefix sum
    for $i \leftarrow n$ to 1 do
        for $j \leftarrow n$ to 1 do
            $S[i, j] = S[i, j+1] + A[i, j]$
    // column prefix sum
    for $j \leftarrow n$ to 1 do
        for $i \leftarrow n$ to 1 do
            $S[i, j] = S[i+1, j] + S[i, j]$
    maxSum $\leftarrow -\infty$
    maxRow $\leftarrow 0$, maxCol $\leftarrow 0$

    for $i \leftarrow 1$ to $n-k+1$ do
        for $j \leftarrow 1$ to $n-k+1$ do
            currSum $\leftarrow S[i+k-1, j+k-1] - S[i-1, j+k-1] - S[i+k-1, j-1] + S[i-1, j-1]$
            if currSum ¿ maxSum then
                maxSum $\leftarrow$ currSum
                maxRow $\leftarrow i$
                maxCol $\leftarrow j$
    return maxSum, maxRow, maxCol
```

Time complexity: $O(n^2)$ because of the nested loops
Space complexity: $O(n^2)$ because of 2D matrix s that prefixes the sum.

4. **Majority**
Given an array $A[1 \ldots n]$ $(n \geq 1)$ of numbers, a majority element is a number M such that more than half the entries of the array A are equal to M . For example, in the array $[2, 3, 2, 2, 4, -1, 2], M = 2$ is a majority element. Notice that some arrays do not have a majority element, e.g., $[1, 2, 3, 2]$. In this problem, you will give a linear time iterative algorithm, $Majority(A[1 \ldots n])$ that outputs a number m, which is guaranteed to be the majority element of A if A has a majority element.

In this problem, you will give a linear time iterative algorithm, $Majority(A[1 \ldots n])$ that outputs a number m, which is guaranteed to be the majority element of A if A has a majority element.

(a) Specify the formal precondition and postcondition for the $Majority(A[1 \ldots n])$ function.

Answer

Preconditions: $n \geq 1$, and $A[1 \ldots n]$ is an array of integers
Post condition: Terminates, and returns a number m if A has a majority element, otherwise returns $\perp$, $A[1 \ldots n]$remains unchanged within its bound 1 through n

(b) Give a linear time iterative algorithm for $Majority(A[1 \ldots n])$

Answer

Algorithm Majority Element

```
Function MajorityElement($A[1 \dots n]$):
    $m \leftarrow 0$
    $votes \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        if $votes = 0$ then
            $votes \leftarrow votes + 1$
            $m \leftarrow A[i]$
        end
        else if $m = A[i]$ then
            $votes \leftarrow votes + 1$
        end
        else
            $votes \leftarrow votes - 1$
        end
    end

$mCount \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    if $A[i] = m$ then
        $mCount \leftarrow mCount + 1$
    end
    if $mCount > \lfloor \frac{n}{2} \rfloor$ then
        return $m$
    end
end
return $\perp$
```

(c) Prove the correctness of your algorithm using a loop-invariant.

Answer

In both loops, we maintain the following invariants. Throughout, $i \in [1, n+1]$ and $A[1 \dots n]$ is unchanged.

Loop 1 Invariants (Majority Candidate Identification):

- If $votes = 0$, there is no current candidate for majority element in $A[1 \dots i-1]$.
- If $votes > 0$, let x be the true majority element (if one exists):
  - If $m = x$, then m is the candidate for majority element in $A[1 \dots i-1]$.
  - If $m \neq x$, then m becomes the candidate for majority element in $A[1 \dots i-1]$ because:

$$votes > \frac{i-1}{2}$$

    which means m appears more than half the time in $A[1 \dots i-1]$. Therefore, no other element, including x, can be majority in $A[1 \dots i-1]$.

Loop 1 Proof:

- **Base case** ($i = 1$): $votes = 0$, so there is no candidate m. Invariant holds.

- **Inductive step**: Assume invariant holds for i. For $i' = i + 1$, consider:
  - **Case 1**: $votes = 0$: Assign $m \leftarrow A[i']$ and $votes \leftarrow 1$. m becomes new candidate.
  - **Case 2**: $votes > 0$ and $A[i'] = m$: Increment $votes$. m remains candidate.
  - **Case 3**: If $votes > 0$ and $A[i'] \neq m$, decrement $votes$. If this makes $votes = 0$, the current candidate m is discarded and a new element is selected in the next iteration. This satisfies the invariant condition for $votes = 0$: no current candidate is held until the algorithm selects a new candidate when processing the next element.

Loop 2 Invariant (Verification):

- At any point during the second loop, $mCount \leq \lfloor \frac{n}{2} \rfloor$ until m is confirmed as majority.

Loop 2 Proof:

- **Base case** ($i = 1$): Initially, $mCount = 0 \leq \lfloor \frac{n}{2} \rfloor$.
- **Inductive step**: As i increases, $mCount$ increments only when $A[i] = m$. Once $mCount$ exceeds $\lfloor \frac{n}{2} \rfloor$, m is returned. Otherwise, if the loop finishes without this condition, $\perp$ is returned.