CSCC73 A5

Saad Makrod

October 2022

Q1 DP Algo to Find Maximum Subarray Product in Array

SUBPROBLEMS TO SOLVE: For each i, $1 \leq i \leq n$, let

$$P(i) = \text{the maximum product between all subarrays in } A[1..i] \text{ that contain } A[i] \qquad (*)$$

For example, let $\{A_1, A_2, \dots, A_i\}$ be the set of all subarrays in $A[1..i]$ that contain $A[i]$ (note that there are exactly i of them) and $\{p_1, p_2, \dots, p_i\}$ be the set of all possible products (note p_j = product of all elements in A_j). Then $P(i) = \max(p_1, p_2, \dots, p_i)$.

SOLVING THE ORIGINAL PROBLEM: The answer to the original problem is $\max\{P(i) : 1 \leq i \leq n\}$

RECURSIVE FORMULA TO COMPUTE THE SUBPROBLEMS:

Before defining the formula I will define a new term. Let $M(i)$ = the minimum product between all the subarrays in $A[1..i]$ that contain $A[i]$. Then the recursive formula for $P(i)$ is,

$$P(i) = \begin{cases} A[i] & i = 1 \\ max(A[i], A[i] * P(i-1), A[i] * M(i-1)) & otherwise \end{cases} \qquad (\dagger)$$

JUSTIFICATION WHY (†) CORRECTLY COMPUTES (*): There are two cases

CASE 1. If $i = 1$ then there is no element at $A[i-1]$ since indices start at 1. In this case there is only one possible solution which is $A[i]$ itself. Thus (†) computes the correct result in this case.

CASE 2. If $1 < i \leq n$, let P be the maximum product between all subarrays in $A[1..i]$ that contain $A[i]$. Since the product P must contain $A[i]$ I know that $P = A[i] * P'$. From (†) P' can be either 1, $P(i-1)$, or $M(i-1)$. For contradiction, assume this is not the case and P'' is a different number which maximizes P such that $A[i] * P' < A[i] * P''$. We have 3 cases $A[i] > 0$, $A[i] = 0$, and $A[i] < 0$.

In the case where $A[i] > 0$ we want $A[i]$ to be multiplied by a positive number greater than or equal to 1. This is because any other value would make the product smaller than $A[i]$. Thus we know that $P'' = max\{1, x\}$ where x is the largest possible product of a subarray in $A[1..i-1]$ that contains $A[i-1]$. By definition $P(i-1)$ is this number since it is the maximum product between all subarrays in $A[1..i-1]$ that contain $A[i-1]$. Clearly in this case we have $P' = P''$ However we assumed that P'' was some other number such that $A[i] * P' < A[i] * P''$, thus we have come to a contradiction.

The case where $A[i] = 0$ is trivial. This is because there is no P'' such that $A[i] * P' < A[i] * P''$ since any number multiplied by 0 is 0. Thus in this case we have a contradiction since $A[i] * P' = A[i] * P''$.

In the last case where $A[i] < 0$, we want $A[i]$ to be multiplied by the smallest possible number less than or equal to 1. This is because any other number will result in a product which is smaller than $A[i]$. Thus we know that $P'' = min\{1, x\}$ where x is the smallest possible product of a subarray in $A[1..i-1]$ that contains $A[i-1]$. By definition $M(i-1)$ is this number since it is the minimum product between all subarrays in $A[1..i-1]$ that contain $A[i-1]$. Clearly in this case we have $P' = P''$ However we assumed that P'' was some other number such that $A[i] * P' < A[i] * P''$, thus we have come to a contradiction.

Thus we have come to a contradiction in all cases so it is clear that (†) computes the correct result in this case.

PSEUDOCODE:

Note that the proof for computing $M(i)$ is very similar to the justification of the computation of $P(i)$.

MAX_SUBARRAY_PRODUCT(A)

$P(1) = A[1]$

$M(1) = A[1]$

for $i = 2$ to n

$M(i) = \min(A[i], A[i] * P(i-1), A[i] * M(i-1))$

$P(i) = \max(A[i], A[i] * P(i-1), A[i] * M(i-1))$

return $\max\{L(i) : 1 \leq i \leq n\}$

RUNNING TIME ANALYSIS: The loop clearly takes $\Theta(n)$ time as calculating the max and min between three numbers is an $O(1)$ operation. Returning the max of the $P(i)$s can be done in $\Theta(n)$ time as well. Thus overall this algorithm is $\Theta(n)$.

Q2 DP Algo to Determine if a can be Generated From Operations on Sequence

<u>SUBPROBLEMS TO SOLVE:</u> Let S be the given string. Then for each $1 \leq i \leq n$ where $n = |S|$ let,

$$A(i,j) = \text{all characters that can be generated from a substring of } S \text{ of length } j \text{ starting at position } i \qquad (*)$$

Note by generated I mean a character that is produced as a result of a choice of parenthesizing of S.

<u>SOLVING THE ORIGINAL PROBLEM:</u> The answer to the original problem is checking if $a \in A(1,n)$

<u>RECURSIVE FORMULA TO COMPUTE THE SUBPROBLEMS:</u> Note $|S| = n$,

$$A(i,j) = \begin{cases} \emptyset & i+j > n+1 \\ \{S[i]\} & j = 1, i \leq n \\ \{(xy) : \forall x, y, k \; where \, x \in A(i,k), y \in A(i+k+1, j-k), 1 \leq k < j\} & otherwise \end{cases} \qquad (\dagger)$$

<u>JUSTIFICATION WHY (†) CORRECTLY COMPUTES (*):</u> There are three cases:

CASE 1. If we have $i + j > n + 1$, clearly this is not a valid substring since if $i + j > n + 1$ then we go start at position i and go beyond the length of the string. Thus (†) returns an empty set in this case as wanted.

CASE 2. In the case where $j = 1$ and $i \leq n$ the substring starts at position i and is only 1 in length. In this case it is clear that the only possible character that can be generated from a substring of length 1 is the substring itself. Thus in this case (†) returns $\{S[i]\}$ which is the correct response.

CASE 3. In the last case where $j > 1$ and $i \leq n$ we need to compute all the characters that can be generated from the substring $S[i..i+j]$. This means I have to consider all possible choices of parenthesizing. This is done exactly by $\{(xy) : \forall x, y, k \; where \, x \in A(i,k), y \in A(i+k+1, j-k), 1 \leq k < j\}$. (†) in this case works from left to right, starting with $k = 1$ and splits $S[i..j]$ into two parts being $S[i..i+k]$ and $S[i+k+1..j]$. Then (†) looks at all possible characters generated by parenthesizing $S[i..i+k]$ and $S[i+k+1..j]$ then takes all possible combinations of the two sets. As k goes from 1 to $j-1$ all choices of parenthesizing are covered. A proof by contradiction covers this below.

For the sake of contradiction, assume that there is some parenthesizing of $S[i..j]$ not considered by (†). Without loss of generality we will assume that the parenthesizing also parenthesizes clear left to right operations. For example evaluating abc is the same as evaluating (((a)(b))(c)). Then ignoring the outer most pair of parenthesis will result in two clear pairs of parenthesis. For example ignoring the outer parenthesis of (((a)(b))(c)) leads to the two clear pairs ((a)(b)) and (c). We will refer to the substring contained by the first set of parenthesis to be A and the substring contained by the second set of paranthesis to be B. Let $k = |A|$, then clearly in this case $A = S[i..i+k]$ and $B = S[i+k+1..j]$. So in this case we have a parenthesizing sequence of $S[i..i+k]$ combined with a parenthesizing sequence of $S[i+k+1..j]$. Clearly these sequences are contained by $A(i,k)$ and $A(i+k+1, j-k)$ respectively, by defintion. Thus we have a contradiction since this must be contained in the set evaluated by (†) but we assumed that it was not.

Thus clearly (†) computes the correct result in this case.

PSEUDOCODE:

```
POSSIBLE_TO_GENERATE_A(S)
    for i = 1 to n
        A(i, 1) = S[i]
    for i = 1 to n
        for j = 2 to n
            if j + i > n + 1
                A(i, j) = ∅
            else
                for k = 1 to j − 1
                    for x in A(i, k)
                        for y in A(i + k + 1, j − k)
                            A(i, j) = A(i, j) ∪ {(xy)}
    return true if a ∈ A(1, n) else false
```

RUNNING TIME ANALYSIS: Clearly the first for loop which initializes $A(i, 1)$ runs in $\Theta(n)$. Clearly the next for loop runs in $\Theta(n^3)$ time since we have 3 nested for loops that go from 1 to n (note in worst case $j = n$ so the third for loop goes from $k = 1$ to $n - 1$). It is important to note that even though we have two additional nested for loops the algorithm is not n^5. This is because $A(i, j)$ indicates a set and since the only possible members of the set is a, b, or c, in the worst case those two nested for loops run $3 * 3 = 9$ times. Thus we can count it as a constant operation. Similarly setting $A(i, j) = \emptyset$ is a constant operation. Lastly returning $a \in A(1, n)$ is a constant operation since $A(1, n)$ has at most three members. Thus we have $\Theta(n) + \Theta(n^3) + \Theta(1) \in \Theta(n^3)$. So this algorithm runs in $\Theta(n^3)$ time.

Q3 DP Algo to Return Minimum Cost Vertical Seam of P

SUBPROBLEMS TO SOLVE: For each i, j where $1 \leq i \leq m$ and $1 \leq j \leq n$, let

$$S(i,j) = \text{the minimum cost of a vertical seam ending at pixel } (i,j) \qquad (*)$$

SOLVING THE ORIGINAL PROBLEM: The answer to the original problem is $\min\{S(m,j) : 1 \leq j \leq n\}$.

Note that as we are determining the minimum cost vertical seam we can keep track of the pixels that construct it.

RECURSIVE FORMULA TO COMPUTE THE SUBPROBLEMS:

$$S(i,j) = \begin{cases} C(i,j) & i = 1 \\ C(i,j) + min(S(i-1,j), S(i-1,j+1)) & j = 1,\, i \geq 2 \\ C(i,j) + min(S(i-1,j-1), S(i-1,j)) & j = n,\, i \geq 2 \\ C(i,j) + min(S(i-1,j-1), S(i-1,j), S(i-1,j+1)) & otherwise \end{cases} \qquad (\dagger)$$

JUSTIFICATION WHY (†) CORRECTLY COMPUTES (*): There are two cases

CASE 1. If $i = 1$ then there is no row in P before i since indices start at 1. Thus in this case there is only one possible solution which is $C(i,j)$ itself. Thus (†) computes the correct result in this case.

CASE 2. In this case we have $2 \leq i \leq m$ and $1 \leq j \leq n$. Let V be the the minimum cost of a vertical seam ending at pixel (i,j). I know that $V = V' + C(i,j)$. Note that V' is a vertical seam that ends at either $(i-1,j-1), (i-1,j)$, or $(i-1,j+1)$ (note the same argument proposed below holds for when we consider only $(i-1,j), (i-1,j+1)$ or only $(i-1,j-1), (i-1,j)$). This is because it was not then V could not be a vertical seam since the pixels in the seam can differ by at most one pixel to the left and right.

Furthermore, note that V' is a minimum cost vertical seam. If V' was not and there was a V'' that ends at vertical seam that ends at either $(i-1,j-1), (i-1,j)$, or $(i-1,j+1)$ such that the cost of $V'' <$ cost of V'. However, then V would not be a minimal cost vertical seam. This is a contradiction since V is a minimal cost vertical seam to (i,j). Therefore it is indeed the case that we can compute V with $C(i,j) + min(S(i-1,j-1), S(i-1,j), S(i-1,j+1))$. Thus the formula (†) is correct in this case.

PSEUDOCODE:

Note that in the pseudocode we will let $V(i,j)$ to represent the pixel before (i,j) in the vertical seam.

MINIMUM_COST_VERTICAL_SEAM(P)

    for $i = 1$ to m

        for $j = 1$ to n

            if $i = 1$

                $S(i,j) = C(i,j)$

                $V(i,j) =$ NULL

            elif $i \geq 2$ and $j = 1$

                $S(i,j) = C(i,j) + min(S(i-1,j), S(i-1,j+1))$

                $V(i,j) = (x,y)$ with min S value where $(x,y) \in \{(i-1,j),(i-1,j+1)\}$

            elif $i \geq 2$ and $j = n$

                $S(i,j) = C(i,j) + min(S(i-1,j-1), S(i-1,j))$

                $V(i,j) = (x,y)$ with min S value where $(x,y) \in \{(i-1,j-1),(i-1,j)\}$

            else

                $S(i,j) = C(i,j) + min(S(i-1,j-1), S(i-1,j), S(i-1,j+1))$

                $V(i,j) = (x,y)$ with min S value where $(x,y) \in \{(i-1,j-1),(i-1,j),(i-1,j+1)\}$

    $(i,j) = \min\{S(i,j)\colon\ i = m,\ 1 \leq j \leq n\}$

    $V =$ the vertical seam constructed by following $V(i,j)$ until reaching NULL

    return V

RUNNING TIME ANALYSIS: Clearly the double for loop runs in $\Theta(mn)$ time as all operations in the loop such as calculating the min are $O(1)$ operations. It takes $\Theta(n)$ to determine the minimum $S(i,j)$ where $i = m$ and $1 \leq j \leq n$. Furthermore, it takes $\Theta(m)$ to construct V as we have to follow $V(i,j)$ m times to get all pixels in the vertical seam. Thus we have $\Theta(mn) + \Theta(n) + \Theta(m) \in \Theta(mn)$. Thus this algorithm is $\Theta(mn)$.