Institute for Software Technology
Oswin Aichholzer, Birgit Vogtenhuber

2022-11-24

Datastructures and Algorithms 2, WS 2022/23
Assignment 1

Surname: Freiberger *First name: Jakob* *Matr.No.:* 12109786

Remark: Please note that it is mandatory to write the exercise on your own, legibly and in logical order. The deadline for this exercise is **Tuesday, 2022-12-06 18:00**.
Your solution has to be handed in via the teach center in form of a pdf-file before this deadline. Please observe also the remarks in the course as well as on the homepage.
This homework assignment contributes 15% to the overall grading of the course (4% for task (a), 6% for task (b), and 5% for task (c)). Questions to this assignment will be answered in the discussion session on 2022-11-30 and in discord sessions with your TA.

Student's note: I would like to apologize for any ugly formatting, as I am not very good with LaTeX.

Maximum Product Given is an array $A[1,\ldots,n]$ of n numbers $a_1,\ldots,a_n$ with $a_k \geq 0$ for $k = 1,\ldots,n$. We are interested in an algorithm that finds a continuous subarray $A[i,\ldots,j]$ which maximizes the product $a_i * a_{i+1} * \ldots * a_j$.

(a) Design an algorithm that is simple to implement and easy to understand. This algorithm does not need to be the most efficient solution.

(b) Design an algorithm that is as efficient as possible.

For both algorithms you can use ideas from the lecture. Explain your approaches in detail, show that the algorithms are correct, and prove tight upper bounds for their time and space requirement.

(c) Which of your algorithms do also work if we release the restriction that $a_k \geq 0$ for $k = 1,\ldots,n$? In other words, entries of $A[1,\ldots,n]$ now can also be negative. Argue your answer for both algorithms.

Solution:

(a) This problem is reminiscent of the Maximum Subarray Sum problem (which I will abbreviate as MSS) treated in the lecture videos. For the MSS, there are array elements that increase the total sum (that is, *positive* elements) as well as elements that decrease the total sum (that is, *negative* elements). Similarly, in this Maximum Subarray Product (MSP), there are elements *greater than* 1 that increase the total product, and elements *less than* 1 that decrease the total product. Elements equal to one do not affect the product. Also, any subarray containing a zero (0) will reduce the total product to 0.

A simple algorithm to find the maximum product of a continuous subarray would be the following:
We initialize the maximum product and the wanted start/end indices with zero, because with all elements of the array ≥ 0, the maximum product cannot be less than zero. Then, for every element a_k, we would compute a_k, then the product of $a_k * a_{k+1}$, then $a_k * a_{k+1} * a_{k+2}$ and so on, until we reach the end of the original array; and for each of these products, we compare it to the previously greatest subarray product. If the product of the current subarray is greater than the global maximum product, we save the current product to the global maximum product, and also save the current subarray start and end index.
For each k, we check every subarray starting from that index; and k runs from 1 to n, so we can be sure that we do in fact compute the product of every single continuous subarray of A. That means that the actual subarray with the maximum product must be among those that we checked, and is saved as the global maximum. The loop ends when $k = n+1$, and the product and start/end indices are returned.

We see that we have n starting elements a_k, and **for each** of those, we need to check at most n subarrays, including those with only one element. Again, **for each** of those subarrays, we need to compute the product of at most n numbers. If we assume constant time for multiplication, we get three nested loops which all depend on the size of the input n. Other operations (writing to variables, comparisons) take constant time, so we have a running time of $T(n) \leq n * n * n * \mathcal{O}(1) = \mathcal{O}(n^3)$.
The algorithm only uses a fixed amount of variables, does not create any new arrays or call itself recursively. Therefore we can assume a space complexity of $\mathcal{O}(1)$.

(b) A more efficient approach would be to use a scanline algorithm, similar to the solution for the MSS problem presented in the lecture.
With this approach, we use a variable `current_max` containing the current maximum product of any subarray ending at index i and let that i run from 1 to n. Of course, we start with `current_max` initialized to 1, so that the actual array contents are not affected by it.
For each i, we multiply the new array element a_i to `current_max`. That way, we get the maximum product of any subarray ending at index i. If that local maximum product is greater than the global maximum product, we set the new values for the product and the start/end indices.
Then, if the `current_max` is smaller than 1, we reset it to 1 and set the new local starting index to the next position in the array. This is because multiplying with a number < 1 will always make a number smaller, so the next subarray will always be bigger if we do not multiply `current_max` (which is < 1) to it. The loop stops if i reaches $n+1$ and we have the global maximum product as well as the start/end indices saved.
For better comprehension, pseudocode for the algorithm is illustrated in Algorithm 1.

Due to the fact that we resetthe `current_max` when the product falls below 1, we can be sure that `current_max *` a_i is always the largest product of any subarray with end index i.
Thus, and because i runs from 1 to n, we also know that the subarray with the globally largest product has to be in our considered arrays, because it is the subarray with the largest product ending at index k, for one $k \in [1, n]$.
Therefore we can be sure that the final output `total_max`, `from` and `to` indeed belong to the maximum subarray product of the initial array A.

Algorithm 1 find_maximum_subarray_product(array a)

```
1:  total_max ← 0
2:  current_max ← 1
3:  from ← 0
4:  to ← 0
5:  k ← 1
6:  for i ← 1 to n do
7:      current_max ← current_max * a_i
8:      if current_max > total_max then
9:          total_max ← current_max
10:         from ← k
11:         to ← i
12:     end if
13:     if current_max < 1 then
14:         current_max ← 1
15:         k ← i + 1
16:     end if
17: end for
```

Note: Pseudocode formatting adopted from:
`https://www.overleaf.com/latex/examples/pseudocode-example/pbssqzhvktkj`
My Python implementation of the algorithm can be found under
`https://github.com/paseyy/DSA_2/blob/main/A1/main.py`

It can be seen that we have to go through the main loop exactly n times, and that all operations besides that only take constant time. Therefore, the total running time of the algorithm is linear, and accordingly $T(n) = \mathcal{O}(n)$.
Also, the algorithm does not create any new arrays or call itself recursively, and only uses a constant number of variables. Therefore, the total space complexity $S(n) = \mathcal{O}(1)$

(c) If we change the specification of the problem to include negative numbers in the array, the algorithm described in (a) would still work as expected. This is due to the fact that we consider every single continuous subarray and separately compute its product, no matter the previous result. That means we will always have the actual global maximum subarray product computed and saved.

In contrast, the algorithm given in (b) would not be correct anymore. Because we don't actually consider every single possible continuous subarray, there are inputs where we would operate incorrectly. In case that the element a_i is negative, `current_max` would fall below 1 (even below 0); and therefore we would reset it, as well as restart the computation starting from the next element. However, due to the fact that two negative numbers multiplied result in a positive number, this might not be the correct decision, because the product could become greater than 0 (even greater than 1) if we keep multiplying the next elements.
Take as an example the array $A = [3, 4, -6, 2, 7, -9]$. We can easily see that the subarray with the maximum product is just the whole array A, with a total product of 9072. If we apply the algorithm, however, it would return $B = [2, 7]$ with a product of 14. This is obviously incorrect, so the algorithm could not be used for this modified problem.