Divide & Conquer
Problem Set 1 – CS 6515/4540 (Fall 2025)

Answers to problem set 1, question 2.

1 Divide & Conquer Algorithm

a) For ease of time complexity analysis, I assume that n is even. Then define the following algorithm to compute the maximum possible product of a contiguous sub-array:

(a) Base case: if A contains only 1 element, then the max product of any sub-array of A = A[0].

(b) Divide the array A into 2 halves $A[0, ..., \frac{n}{2} - 1]$ is the left half, called L, and $A[\frac{n}{2}, ..., n - 1]$ is the right half, called R.

(c) Solve the problem for the L and R sub-arrays. We do some extra processing here to improve our combination step, where we need to look at possible sub-arrays that span across L and R. To improve the merge step, instead of returning only the maximum product, each recursive call returns the following information: (`totalProduct`, `prefixProduct`, `suffixProduct`, `maxProduct`) where

- `totalProduct` = product of all elements in the input array,
- `prefixProduct` = maximum product of a prefix of the input array,
- `suffixProduct` = maximum product of a suffix of the input array,
- `maxProduct` = maximum product of any sub-array in the input array.

(d) Given the results from the left and right sub-arrays, we now perform the combination step:

- $\texttt{totalProduct} = \texttt{totalProduct}_L * \texttt{totalProduct}_R$,
- $\texttt{prefixProduct} = max(\texttt{prefixProduct}_L, \texttt{totalProduct}_L * \texttt{prefixProduct}_R)$,
- $\texttt{suffixProduct} = max(\texttt{suffixProduct}_R, \texttt{totalProduct}_R * \texttt{suffixProduct}_L)$,
- $\texttt{maxProduct} = max(\texttt{maxProduct}_L, \texttt{maxProduct}_R, \texttt{suffixProduct}_L * \texttt{prefixProduct}_R)$

The pseudo-code:

Algorithm 1: Max product of a contiguous sub-array

```
Function MaxProduct(A[0, ...., (n − 1)])
    if n == 1 then
        return (tp = A[0], pp = A[0], sp = A[0], mp = A[0])
    else
        l = 0;
        r = n − 1;
        mid = (l + r)/2;
        # Solve sub-problems
        LM = MaxProduct(A[0, ..., mid]);
        RM = MaxProduct(A[(mid + 1), ..., (n − 1)]);
        # Combination Step
        tp = LM.tp · RM.tp
        pp = max(LM.pp, LM.tp · RM.pp)
        sp = max(RM.sp, RM.tp · LM.sp)
        mp = max(LM.mp, RM.mp, LM.sp · RM.pp)
        return (tp, pp, sp, mp);
```

b) (a) When A has length 1, then the array contains only 1 element, and only 1 sub-array is possible containing that element. Therefore, the max product, prefix product, suffix product and total product of any sub-array would be equal to that 1 element in the array. Hence, the base case trivially holds.

(b) Assuming that the algorithm returns the correct output for the left and right half of the array, LM and RM respectively, for A, any maximum product contiguous sub-array falls into one of the following categories:

- It lies completely in the left half array, $A[0, ..., (n/2 - 1)]$ -> $LM.tp$, or
- It lies completely in the right half array $A[(n/2), ..., (n - 1)]$ -> $RM.tp$, or
- It includes the middle element, at index = $n/2$, and spans into the left and right sub-array's. Any maximum product subarray that crosses the midpoint must be the product of the maximum suffix in the left half and the maximum prefix in the right half -> $LM.sp * RM.pp$. The extra information we compute to reduce processing at each step of the recursion follow from their definitions.

Taking the max of the above 3 scenarios ensures that we have properly handled all cases and are returning the correct result. An important point to note here is that if there were negative elements in the array, then this solution would not work and we'd also need to keep track of the min product with the number of negative elements in each sub-array.

c) We need to solve 2 subproblems of size half the original array and then compute a prefix/suffix product spanning across the middle element. Because of the extra information that we return at each step of the recursion, computing the product for a sub-array crossing the middle element is O(1). Therefore, we can formulate the recursion as:

$$T(1) = O(1)$$

$$T(n) = 2T(\frac{n}{2}) + O(1)$$

Using the tree method or using Masters' Theorem we can find a bound for the time complexity.

- **Tree Method**: At level h of our tree, we have 2^h nodes and the operation we perform at each node takes O(1) time. Expanding this we get the total time complexity $= \sum_{hi=0}^{log_2 n} 2^{hi} = 2^{log_2 n+1} - 1 = 2n - 1$. Hence, the time complexity is O(n). We can also reason that this algorithm doesn't have a difference in time complexity from worst case to best case, as at each step we'll need to solve 2 subproblems of half the size. Therefore, $T(n) = \Omega(n)$, which implies $T(n) = \Theta(n)$
- **Masters' Theorem**: states that for a recursion of the form

  $$T(n) = aT(\frac{n}{b}) + O(n^d)$$

  if $d < log_b a$, which is true in this case, then the recursion is of the form $\Theta(n^{log_b a}) = \Theta(n)$