

# Divide & Conquer

## Problem Set 1 – CS 6515/4540 (Fall 2025)

This problem set is due on **Thursday, August 28th**. Submission is via Gradescope. Your solution must be a typed pdf (e.g. via LaTeX) – no handwritten solutions.

### 1 O-Notation

#### 1.1 LLM Question

Use the following prompt in an LLM (like chatGPT/CoPilot/Gemini) and upload the transcript of your session.

“Act like an undergrad with a CS major who wants to understand Big O notation. Now interactively ask me 5 questions (i.e., one by one while waiting for my answers) to improve your (and my) understanding of Big O. In the end, give me all the correct answers and constructive feedback.”

#### 1.2 Question:

We are interested in finding *all* constants for which the following  $O(\cdot)$  bounds hold.

1. For which constants  $c, d > 0$  do we have  $n^c = O(d^n)$ ? Prove your answer.
2. For which constants  $c, d > 0$  do we have  $(\log(n))^c = O(n^d)$ ? Prove your answer.

Remark:  $c, d$  do not need to be integers. In particular, they could be smaller than 1.

**1:** This is true for all  $c > 0$  and  $d > 1$ . Proof:

First, assume that  $c$  is an integer. If  $c$  is not an integer, we have  $n^c \leq n^{\lceil c \rceil}$  and it would suffice to just show  $n^{\lceil c \rceil} = O(d^n)$ . So we can focus on the integer case.

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{n^c}{d^n} \\ (\text{L'Hopital: derivate numerator and denominator}) &= \lim_{n \rightarrow \infty} \frac{cn^{c-1}}{\ln(d) \cdot d^n} \\ (\text{Repeatedly apply L'Hopital}) &= \lim_{n \rightarrow \infty} \frac{c!}{\ln(d)^c \cdot d^n} \\ (c! \text{ is a constant}) &= 0 \end{aligned}$$

Since the limit is a value  $< \infty$ , we have  $n^c = O(d^n)$ .

Remark: For  $d \leq 1$  the denominator is  $\leq 1$ , that's why for  $d \leq 1$  we have  $n^c \neq O(d^n)$ .

**2:** This is true for all  $c, d > 0$ . Proof:

First, assume that  $c$  is an integer. If  $c$  is not an integer, we have  $(\log(n))^c \leq (\log(n))^{\lceil c \rceil}$  and it would suffice to just show  $(\log(n))^{\lceil c \rceil} = O(n^d)$ . So we can focus on the integer case.

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{(\log(n))^c}{n^d} \\ (\text{L'Hopital: derivate numerator and denominator}) &= \lim_{n \rightarrow \infty} \frac{c(\log(n))^{c-1} \frac{1}{n}}{d \cdot n^{d-1}} \\ &= \lim_{n \rightarrow \infty} \frac{c(\log(n))^{c-1}}{d \cdot n^d} \\ (\text{Repeatedly apply L'Hopital}) &= \lim_{n \rightarrow \infty} \frac{c!}{d^c n^d} \\ (c! \text{ is a constant}) &= 0 \end{aligned}$$

Since the limit is a value  $< \infty$ , we have  $(\log(n))^c = O(n^d)$ .

## 2 Divide & Conquer Algorithm

Given a length  $n$  array  $A$  (you may assume there are only non-negative entries), construct an algorithm that returns the maximum possible product of *contiguous* entries. For example when  $A = [0.5, 2, 0.25, 2, 0.75, 2, 0.5]$ , the maximum product is 3 from the sub-array  $[2, 0.75, 2]$ .

**Problem:** Construct a divide & conquer algorithm  $\text{MAXPRODUCT}(A[1\dots n])$  that solves the above problem. Your answer must provide the following:

- (a) A pseudo-code description of your algorithm.
- (b) A correctness argument. You may provide a proof by induction, but answering the following questions suffices:
  - (a,i) Why is the base case correct (when  $A$  has length 1)?
  - (a,ii) Why does the algorithm return a correct answer, if it has the solution for the left and right half of the array, i.e.,  $\text{MAXPRODUCT}(A[1\dots n/2])$  and  $\text{MAXPRODUCT}(A[n/2 + 1, \dots, n])$ .
- (c) Complexity analysis (e.g., via Tree Method, induction, or reusing a bound from class).

There exist iterative algorithms for this problem, but the purpose of this exercise is to practice divide & conquer. Your solution must be a divide & conquer algorithm, i.e., recursively solving the problem by splitting array  $A[1\dots n]$  into  $A[1\dots n/2]$  and  $A[n/2 + 1\dots n]$ .

For full points, your solution should be as fast as possible.

The following is very long, that's because we provide 2 correct solutions. The first solution is simpler, but slightly slower (and thus wouldn't get full points). The second solution is more complicated but faster and would receive full points. The more complicated solution may be easier to understand if you first read the simpler solution.

---

**Algorithm 1:** Simple MAXPROD

---

```
1 procedure MAXPROD(A)
2   if length of A = 1 then
3     if A[1] ≥ 1 then
4       return A[1]
5     else
6       return 1
7   else
8     // compute max product of form A[i...j] with i ≤ j ≤ n/2
9     leftMaxProd = MAXPROD(A[1,...,n/2])
10    // compute max product of form A[i...j] with n/2 < i ≤ j
11    rightMaxProd = MAXPROD(A[n/2 + 1,...,n])
12    // compute max product of form A[i...n/2]
13    maxProdL = 1, tempProd = A[n/2]
14    for i = n/2 - 1...1 do
15      tempProd* = A[i]
16      if tempProd > maxProdL then maxProdL = tempProd ;
17      // compute max product of form A[n/2 + 1...j]
18      maxProdR = 1, tempProd = A[n/2 + 1]
19      for j = n/2 + 2...n do
20        tempProd *= A[j]
21        if tempProd > maxProd then maxProd = tempProd ;
22      // maxProdL * maxProdR is max product of form A[i...j] for i ≤ n/2 < j
23      return max(maxProdL · maxProdR, leftMaxProd, rightMaxProd)
```

---

**(b, i) Base Case** For  $n = 1$ , if the entry is  $< 1$ , the best product is 1 by having an empty product. Otherwise, if the entry is  $> 1$ , the best product is just that one entry. (It's fine if your solution does not accept empty products.)

**(b,ii) Inductive Step** Assume the algorithm returns the correct result for all arrays of length up to  $n - 1$ .

Then by  $A[1\dots n/2]$  and  $A[n/2 + 1\dots n]$  having size  $n/2$ , we know

$$\text{leftMaxProd} = \max_{i < j \leq n/2} \prod_{k=i}^j A[k] \quad \text{rightMaxProd} = \max_{n/2 < i < j} \prod_{k=i}^j A[k]$$

so to get the largest possible product, the only part that might be larger than leftMaxProd or rightMaxProd are products that cross the center. Here we have

$$\max_{i \leq n/2 < j} \prod_{k=i}^j A[k] = (\max_{i \leq n/2} \prod_{k=i}^{n/2} A[k]) \cdot (\max_{n/2 < j} \prod_{k=n/2+1}^j A[k])$$

where the two products on the right-hand side are precisely the products maxProdL and maxProdR computed by our algorithm. Thus the maximum product is

$$\begin{aligned} \max_{i \leq j} \prod_{k=i}^j A[k] &= \max(\max_{i \leq j \leq n/2} \prod_{k=i}^j A[k], \max_{n/2 < i \leq j} \prod_{k=i}^j A[k], \max_{i \leq n/2 < j} \prod_{k=i}^j A[k]) \\ &= \max(\text{leftMaxProd}, \text{rightMaxProd}, \text{maxProdL} \cdot \text{maxProdR}) \end{aligned}$$

which is precisely the value returned by our algorithm.

**(c)** We have  $T(n) = 2T(n/2) + O(n)$  since we have two recursive calls on arrays that are half the size. Further, we spend  $O(n)$  time to combine the solutions of the subproblem.

By Master theorem this is  $O(n \log n)$  because  $a = b = 2, c = 1$  and  $\log_2(2) = 1$  which is case 2 of the Master Theorem.

---

**Algorithm 2:** Advanced MAXPROD

---

```
1 procedure MAXPROD( $A$ )
2   // Idea is the same as the previous algorithm, but instead of computing max
   // product of form  $A[i...n/2]$  and  $A[n/2+1...j]$  via a for-loop, we use recursion.
   // Recursion returns the (i) maximum product, (ii) max product of form  $A[1...i]$ ,
   // (iii) max product of form  $A[j...n]$ , (iv) total product of all entries
3   if length of  $A = 1$  then
4     if  $A[1] \geq 1$  then
5       | return  $A[1], A[1], A[1], A[1]$ 
6     else
7       | return  $1, 1, 1, A[1]$ 
8   else
9     leftMaxProd, leftPrefixMaxProd, leftSuffixMaxProd, leftTotalProd =
10    MAXPROD( $A[1, \dots, n/2]$ )
11    rightMaxProd, rightPrefixMaxProd, rightSuffixMaxProd, rightTotalProd =
12    MAXPROD( $A[n/2 + 1, \dots, n]$ )
13    totalProd = leftTotalProd · rightTotalProd
14    // the max product of form  $A[1...i]$  either has  $i \leq n/2$  (so the value is
      // leftPrefixMaxProd) or  $i > n/2$  (then the value is
      // leftTotalProd · rightPrefixMaxProd)
15    prefixMaxProd = max( leftPrefixMaxProd, leftTotalProd · rightPrefixMaxProd )
16    // the max product of form  $A[j...n]$  either has  $j > n/2$  (so the value is
      // rightSuffixMaxProd) or  $j \leq n/2$  (then the value is
      // rightTotalProd · leftSuffixMaxProd)
17    suffixMaxProd = max( rightSuffixMaxProd, rightTotalProd · leftSuffixMaxProd )
18    // the max product of  $A[i...j]$  either has  $i \leq j \leq n/2$  (so the value is
      // leftMaxProd) or  $n/2 < i \leq j$  (then the value is rightMaxProd), or  $i \leq n/2 < j$ 
      // (then the value is leftSuffixMaxProd · rightPrefixMaxProd)
19    maxProd = max( leftMaxProd, rightMaxProd, leftSuffixMaxProd · rightPrefixMaxProd )
20
21 return maxProd, prefixMaxProd, suffixMaxProd, totalProd
```

---

**Faster variant** The algorithm not only returns the maximum product, but also the 3 additional values. We claim their values are as follows and prove it by induction.

$$totalProd = \prod_{k=1}^n A[k], \quad prefixMaxProd = \max_i \prod_{k=1}^i A[k], \quad suffixMaxProd = \max_i \prod_{k=i}^n A[k],$$

**(b, i) Base Case** For  $n = 1$ , if the entry is  $< 1$ , the best product is 1 by having an empty product. Otherwise, if the entry is  $\geq 1$ , the best product is just that one entry. Respectively the algorithm may also return 1 or  $A[1]$  for the maximum prefix and suffix products.

**(b,ii) Inductive Step** Assume the algorithm returns the correct result for all arrays of length up to  $n - 1$ .

Then by  $A[1\dots n/2]$  and  $A[n/2+1\dots n]$  having size  $n/2 \leq n - 1$ , we know the output from the recursive calls are correct, i.e., we have

$$\begin{aligned} leftMaxProd &= \max_{i \leq j \leq n/2} \prod_{k=i}^j A[k], & leftTotalProd &= \prod_{k=1}^{n/2} A[k], \\ leftPrefixMaxProd &= \max_{i \leq n/2} \prod_{k=1}^i A[k], & leftSuffixMaxProd &= \max_{i \leq n/2} \prod_{k=i}^{n/2} A[k] \\ rightMaxProd &= \max_{n/2 < i \leq j} \prod_{k=i}^j A[k], & rightTotalProd &= \prod_{k=n/2+1}^n A[k] \\ rightPrefixMaxProd &= \max_{n/2 < i} \prod_{k=n/2+1}^i A[k] & rightSuffixMaxProd &= \max_{n/2 < i} \prod_{k=i}^n A[k] \end{aligned}$$

So to get the largest possible product, the only part that might be larger than  $leftMaxProd$  or  $rightMaxProd$  are products that cross the center. Here we have

$$\max_{i \leq n/2 < j} \prod_{k=i}^j A[k] = (\max_{i \leq n/2} \prod_{k=i}^{n/2} A[k]) \cdot (\max_{n/2 < j} \prod_{k=n/2+1}^j A[k]) = leftSuffixMaxProd \cdot rightPrefixMaxProd$$

Thus the maximum product is

$$\begin{aligned} \max_{i \leq j} \prod_{k=i}^j A[k] &= \max(\max_{i \leq j \leq n/2} \prod_{k=i}^j A[k], \max_{n/2 < i \leq j} \prod_{k=i}^j A[k], \max_{i \leq n/2 < j} \prod_{k=i}^j A[k]) \\ &= \max(leftMaxProd, rightMaxProd, leftSuffixMaxProd \cdot rightPrefixMaxProd) \end{aligned}$$

which is precisely the first value returned by our algorithm.

We further have

$$\begin{aligned} prefixMaxProd &= \max(leftPrefixMaxProd, leftTotalProd \cdot rightPrefixMaxProd) \\ &= \max(\max_{i \leq n/2} \prod_{k=1}^i A[k], \max_{n/2 < i} \prod_{k=1}^i A[k]) = \max_i \prod_{k=1}^i A[k] \\ suffixMaxProd &= \max(rightSuffixMaxProd, leftSuffixMaxProd \cdot rightTotalProd) \\ &= \max(\max_{n/2 < i} \prod_{k=i}^n A[k], \max_{i < n/2} \prod_{k=i}^n A[k]) = \max_i \prod_{k=i}^n A[k] \\ totalProd &= leftTotalProd \cdot rightTotalProd = \prod_{k=1}^n A[k] \end{aligned}$$

Thus all our return values are correct.

(c) We have  $T(n) = 2T(n/2) + O(1)$  since we have two recursive calls on arrays that are half the size. Further, we spend  $O(1)$  time to combine the solutions of the subproblem.

By Master theorem this is  $O(n)$  because  $a = b = 2, c = 0$  and  $\log_2(2) = 1 > 0$  which is case 1 of the Master Theorem.

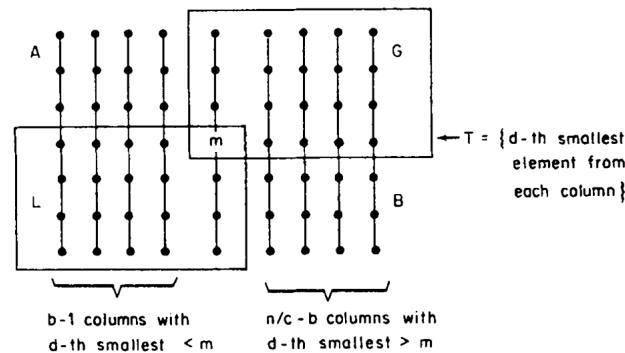
### 3 Median of Medians

The standard median of the median algorithm uses groups of five. Instead, consider a median of the median algorithm, with groups of  $M \geq 3$ .

1. Give the recurrence for this algorithm (with respect to  $M$ ) to calculate the worst-case runtime.
  2. Suppose  $M = 7$ , now solve the recurrence by induction. Make sure to clearly write the induction hypothesis, base case, and induction step.
  3. Which of the following is true about the running time of the algorithm in part (b.) for  $M = 7$  (as compared to  $M = 5$  from class): (i) running time improves to  $o(n)$ , (ii) running time stays  $\Theta(n)$ , or (iii) running time worsens to  $\omega(n)$ ? Briefly explain your choice.
- (a.) (10 points) Give the recurrence for this algorithm (with respect to  $M$ ) to calculate the worst-case runtime?

$$T(n) = T(\lceil n/M \rceil) + T(n - \lceil \frac{M}{2} \rceil \cdot \lceil \frac{n}{2M} \rceil) + \Theta(n)$$

Explanation (not graded): We need a sub-problem for finding the median of the middle row. We need to recurse on the sub-problem after removing one of the quadrants.



- (b.) (15 points) Suppose  $M = 7$ , now solve the recurrence by induction. Make sure to clearly write the induction hypothesis.

When separating the array into groups of 7, finding the median is still  $\Theta(n)$ . Using the equation above, we see that our sub-problems will be of size  $n/7$  and  $5/7n$ . The lower bound is trivially linear, we need to split into groups and perform the median finding before we test, so we only need to worry about establishing the upper bound. This means that the  $T(n) \leq T(5/7n) + T(n/7) + \Theta(n)$ .

*Inductive hypothesis:* Given that  $T(i) \leq T(5/7i) + T(i/7) + ci$ ,  $\forall c, \exists k$  s.t.  $T(n) \leq kn$ . We need to prove it for  $i + 1^{th}$  case.

*Base Case:* For median with list of length 1, there will be one step, so the induction hypothesis is true for  $k > 1$ .

*Inductive Step:* Assume the induction hypothesis  $\forall i \leq n$ , then

$$\begin{aligned} T(n+1) &\leq T(5(n+1)/7) + T((n+1)/7) + cn \\ &\leq k(5(n+1)/7) + k((n+1)/7) + c(n+1) = (6k/7 + c)(n+1) \\ &\leq k(n+1), \text{ if } k \geq 7c. \end{aligned}$$

From this we can conclude that the induction hypothesis is true for a sufficiently large  $k$ , showing that our assumption was correct and  $T(n) = \mathcal{O}(n)$ .

- (c.) (5 points) Which of the following is true about the running time of the algorithm when  $M = 7$ : (i) running time improves to  $o(n)$ , (ii) running time stays  $\Theta(n)$ , or (iii) running time worsens to  $\omega(n)$ ? Briefly explain your choice.

Since  $T(n) = \mathcal{O}(n)$  and  $T(n) = \Omega(n)$ ,  $T(n) = \Theta(n)$ .