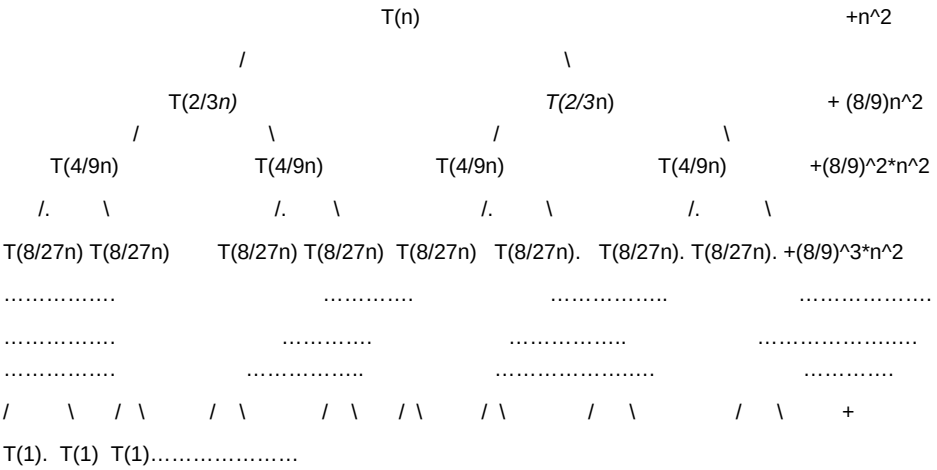
Algorithms Assignment 3

1. Solve the recurrence

$$T(n) = 2 * T(2/3 * n) + n^2$$

first by using a recursion tree and then using the Master theorem. Show work.

Using Recursion Tree Method:



the sequence will be in the form of T((2/3)^k*n) and after many recurrences we get T(1) and hence finding the kth term when we get T(1)

$$T((2/3)^k * n) = T(1)$$

$$(2/3)^k * n = 1$$

$$then\ it\ implies\ k = \log_{3/2} n\ levels$$

it is divided two times for every iteration so the number of computations is

$$2^{log_{3/2} n}$$

and next the f(n) terms we see the terms are in geometric progression

$$n^2 + (8/9)n^2 + (8/9)^2 * n^2 + +(8/9)^3 * n^2 \ldots\ldots\ldots\ldots$$

$$n^2[1 + 8/9 + (8/9)^2 + \ldots\ldots\ldots log_{3/2} n + 1\ terms]$$

hence these are in geometric progression and r<1

the sum of k terms in geometric progression and r<1 is

$$a * (1 - r^n / 1 - r)$$

$$n^2 * [1 - (8/9)^{log_{3/2} n + 1} / 1 - 8/9]$$

$$9 * n^2 * [1 - (8/9) n^{log_{3/2} 8/9}]$$

$$n^{log_{3/2}8/9} \to 0$$

$$therefore\, T(n) = O(n^2)$$

Recurrences level on the both left and right sides are the same :

$$T(n) = \Theta(n^2)$$

Using Masters theorem:

If the recurrence relation is of the form T(n) = aT(n/b) + f(n), where a ≥ 1, b > 1, and f(n) is an asymptotically smaller function than n^c for some constant c ≥ 0:

In this case, the time complexity can be determined as follows:

$$If\ f(n) = O(n^c)\ for\ some\ constant\ c < log_b(a),\ then\ the\ time\ complexity\ is\ T(n) = \Theta(n^{log_b(a)}).$$

$$If\ f(n) = \Theta(n^{log_b(a)} * log^k n)\ for\ some\ constant\ k \geq 0,\ then\ the\ time\ complexity\ is\ T(n)\ = \Theta(n^{log_b(a)} * log^{(k+1)} n$$

$$If\ f(n) = \Omega(n^c)\ for\ some\ constant\ c > log_b(a),\ and\ if\ a * f(n/b) \leq kf(n)\ for\ some\ constant\ k < 1\ and\ sufficie$$

here a=2,b=3/2 and f(n)=n^2 and hence c=2 then

$$2 > log_{3/2}2,\ c > log_b a$$

$$T(n) = \Theta(f(n))$$

$$hence\, T(n) = \Theta(n^2)$$

Thus proved using both the methods of recursion tree and Masters Theorem

2.

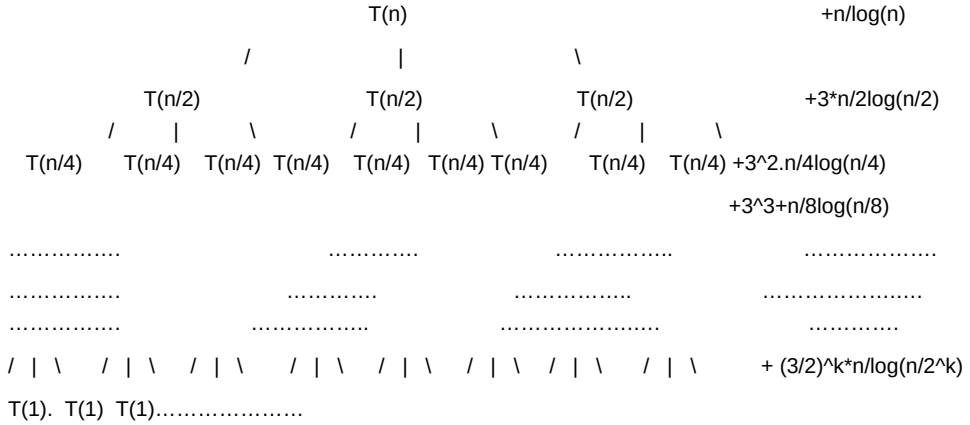
Give asymptotic upper and lower bounds for the recurrence

$$T(n) = 3 * T(n/2) + n/log(n)$$

Justify your answer.

using the recursion tree method

$$T(n) = 3 * T(n/2) + n/log(n)$$



After k iterations we get T(1) to find levels we equate

$$n/2^k = 1$$

$$we\ get\ log(n) = k$$

It is divided by 3 times every time so the number of computations is

$$3^{log(n)} \Rightarrow \Theta(n^{log_3 2})$$
$$\Theta(n^{1.58})$$

and then the expansion side of the n/log(n)

$$n * \sum_{i=1\ to\ k-1} (2/2)^i * 1/(log(n) - i) <= n * \sum_{i=1\ to\ k-1} (3/2)^i * 1/(log(n) - i) <= n * \sum_{i=1\ to\ k-1} (3/2)^i$$

this holds good until at k=k-1

$$1/log(n) - 1 = 1$$

$$n * \sum_{i=1\ to\ k-1} (2/2)^i * 1/(log(n) - i) <= n * \sum_{i=1\ to\ k-1} 1/k - i$$

this is harmonic progression and so we can express it as log(k) and hence this implies

$$n * \sum_{i=1\ to\ k-1} 1/k - i = \Omega(log(log(n))$$

similarly the other side we get the LHS

$$n * \sum_{i=1\ to\ k-1} (3/2)^i = 1 - (3/2)^{k-1}/1 - 3/2$$
$$2 * n * ((3/2^{log(n)-1} - 1)$$
$$we\ get\ O(n^{1.58})$$

from both equations, we can express T(n) as

$$\Omega(n^{1.58}) + \Omega(log(log(n))) = \Omega(n^{1.58})$$

from the above equations, we can also express the upper bound also to be

$$O(n^{1.58}) + O(n^{1.58}) = O(n^{1.58})$$

and hence we can say that

$$T(n) = \Theta(n^{1.58})$$

3a.

$$\sqrt{n} * T(\sqrt{n}) + n$$

using the Iterations method for the first iteration

$$T(\sqrt{n}) = n^{1/4}T(n^{1/4}) + n^{1/2}$$

T(n)=n^{1/2}*[n^{1/4}T(n^{1/4})+n^{1/2}] +n

T(n)=n^{3/4}*T(n^{1/4}+2*n

for the next iteration, you get

$$T(n) = n^{7/8} * T(n^{1/8}) + 3 * n$$

$$T(n) = n^{15/16} * T(n^{1/16}) + 4 * n$$

$$\dots$$

$$T(n) = n^{(2^k - 1/2^k)} * T(n^{(1/2^k)}) + k * n$$

the above equation resembles the kth iteration from the top and hence

the last term breaks down to be T(2) and hence comparing the coefficients

$$n^{(1/2^k)} = 2$$

applying logarithms on both the sides

$$1/2^k log(n) = log(2) = 1$$

log(n)=2^k

applying logarithms on both the sides

log(log(n))=klog2

$$log(log(n) = k$$

$$T(n) = n^{(2^{log(log(n))} - 1)/2^{log(log(n)}} * T(2) + log(log(n) * n$$

$$\Rightarrow n^{log(n) - 1/log(n)} * T(2) + log(log(n) * n$$

the first term becomes insignificant and the

$$T(n) = O(n * log(log(n))),\ T(n) = \Omega(n * log(log(n)))$$

therfore upper and lower bound will be n*log(log(n)).

3 b

$$3T(n-1)$$

using iteration method

T(n)=3*T(n-1)

T(n-1)=3*T(n-2)

After first iteration ⇒ 3*(3*T(n-2))

2nd Iteration ⇒ 3*(3*(3*T(n-3)))

3rd iteration 3*(3*(3*(3*T(n-4))))

..

3^k+1*(T(n-k-1))

It is following this sequence and the last term we end up with is and the base case will be T(0)

hence to get the k value equating to it T(0)=T(n-k-1)

therefore k=n-1

$$3^{n-1+1}T(0)$$

$$T(n) = 3^n$$

By this, we conclude the asymptotic bounds of this equation T(n)=3T(n-1) grows exponentially

and

$$T(n) = O(3^n),\ T(n) = \Omega(3^n)$$

4.

Maximum Product Subarray

Given an integer array `nums`, find a contiguous non-empty subarray within the array that has the largest product, and return *the product*.

The test cases are generated so that the answer will fit in a **32-bit** integer.

A **subarray** is a contiguous subsequence of the array.

1. The function `maxProduct` takes a list of integers `nums` as input and returns an integer, which represents the maximum product of a subarray.
2. Two lists `lmax` and `lmin` are initialized with the same length as `nums`. These lists are used to store the maximum and minimum product subarrays ending at each index `i` in `nums`.
3. The loop iterates over each index `i` in the range from 1 to the length of `nums`. For each index:

   a. The maximum product ending at index `i` is calculated by taking the maximum of three values:

   - `nums[i]`: the current element itself.
   - `lmax[i-1] * nums[i]`: the maximum product ending at the previous index `i-1`, multiplied by the current element.
   - `lmin[i-1] * nums[i]`: the minimum product ending at the previous index `i-1`, multiplied by the current element.
     The purpose of considering both positive and negative products is to handle cases where negative numbers may yield larger products when multiplied by other negative numbers.

   b. The minimum product ending at index `i` is calculated similarly, but by taking the minimum of the same three values.
4. Finally, the maximum value from the `lmax` list is returned as the result, representing the maximum product of any subarray in the given list of integers.

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        lmax=[nums[0]]*len(nums)
        lmin=[nums[0]]*len(nums)
        for i in range(1,len(nums)):
            lmax[i]=max(nums[i],lmax[i-1]*nums[i],lmin[i-1]*nums[i])
            lmin[i]=min(nums[i],lmax[i-1]*nums[i],lmin[i-1]*nums[i])
        return max(lmax)
```

Python3 • Auto

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        lmax=[nums[0]]*len(nums)
        lmin=[nums[0]]*len(nums)
        for i in range(1,len(nums)):
            lmax[i]=max(nums[i],lmax[i-1]*nums[i],lmin[i-1]*nums[i])
            lmin[i]=min(nums[i],lmax[i-1]*nums[i],lmin[i-1]*nums[i])
        return max(lmax)
```

Testcase | Result

Input

nums =

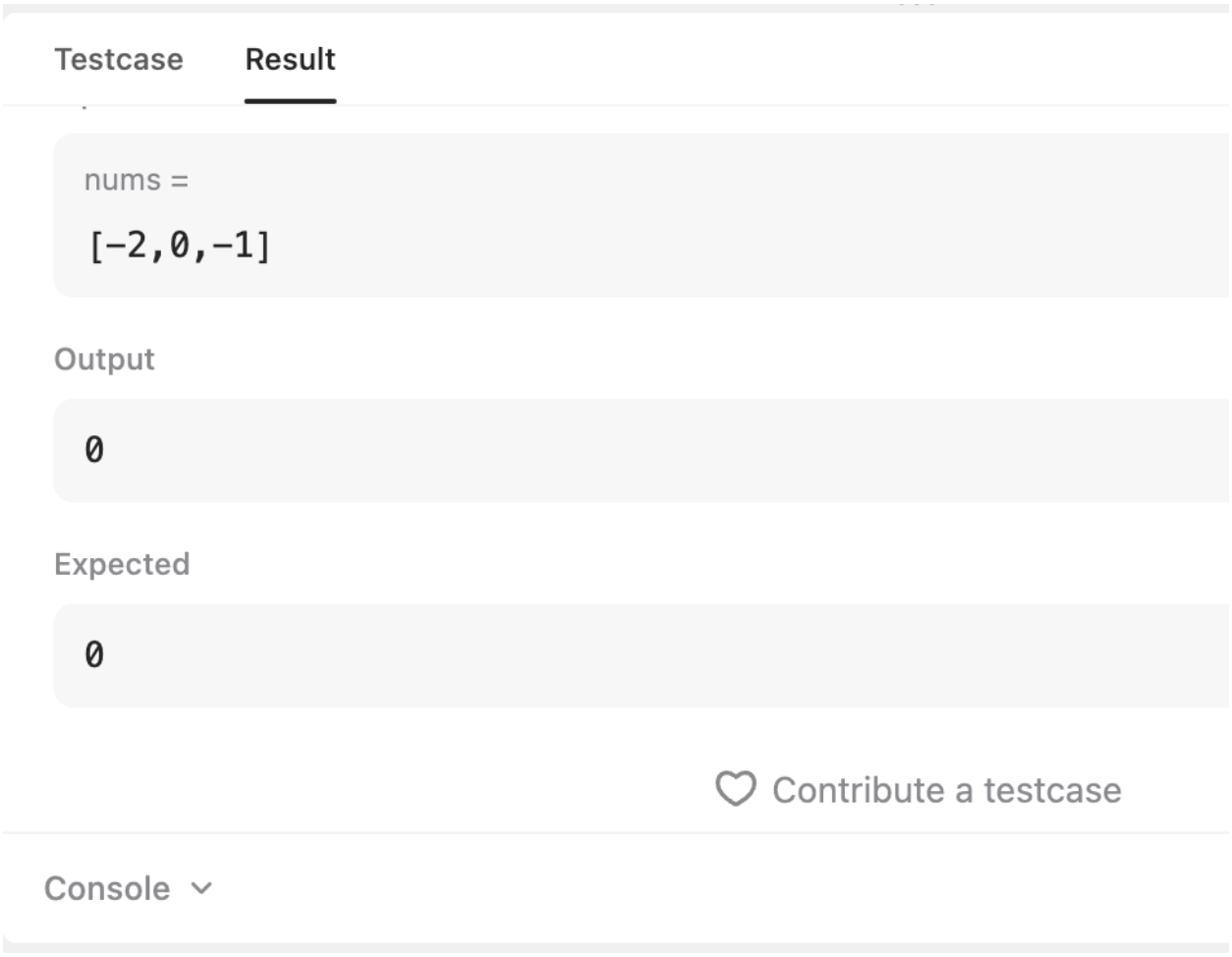
[2,3,-2,4]

Output

6

Expected

6


Testcase
Result
nums =
[-2,0,-1]
Output
0
Expected
0
Contribute a testcase
Console