Q1.

a) In worst case how long does it take to push one more element onto a multistack containing n elements

after n elements first k stacks are full.

time to move these n elements recursively will take time $\propto 3^k \Rightarrow O(n)$

b) if user never pops amortized cost $= O(\log n)$ where n is the max number of elements in the multistack during lifetime.

S_i has 3^{i-1} element for each i

as described above after n elements if k stacks are full $\Rightarrow$ $k = O(\log k)$

there is no case of poping $\Rightarrow$ movement from higher stack to lower stack. is not there.

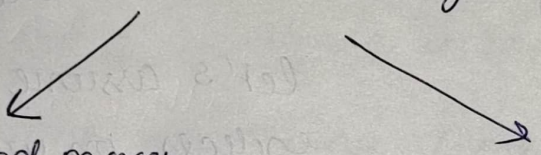
$\Rightarrow$ only movements from lower to higher will be present with each element movement will be $O(\log k)$

$\Rightarrow$ Since there are total n elements $\Rightarrow$ amortize cost will be

$$n * O(\log k)$$
$$= n * O(\log n$$
$$= O(n \log n).$$

c) Prove that in any intermixed seq. of pushes & pop each push & pop opsn take $O(\log n)$ amortized time where n is the max no of elements in stack during its life time.

max i for which S_i is filled is $O(\log n)$ as we know number of items in S_i are 3^{i-1} (described in last section) now take two cases for movement of an item



upward movement- (S_i to S_{i+1})

When it will happen.

S_1 to S_i are atleast one third filled (that's why overflow happens at ist place)

downward movement- (S_{i+1} to S_i).

S_1 to S_i are empty.

(as there are no element to pop in lower stacks.

Let's assume the total such movement for an items is j total cost for this element- $O(j\,3^i)$

but number of pushes & pop should be $O(j\,3^i)$

- we are considering all pairs of push & pop till $i = O(\log n)$

$\Rightarrow$ n push & pop will take $O(n \log n)$

$\Rightarrow$ 1 push opsn take $O(\frac{n}{n} \log n)$

similarly 1 pop opsn $O(\log n)$

Proved

Q2.

a). Suppose we call Nudge, n times -- show amortized complexity is O(1)

if we are calling the counter and x_k needs to be incremenent there will be 2 cases

$x_k \neq 2$
increment it
O(1)

$x_k = 2$
recursive calls should be done.

let's assume some neighbouring indices on which Nudge (i+1) & Nudge (i-1) is being called.
& they are x_i $\ell \leq i \leq m$
& $x_{\ell-1}$ & x_{m+1} are not 2.
recursive procedure Nudge (i+1) terminate at index m+1
& similarly Nudge (i-1) at $\ell-1$
Setting all x_ℓ --- x_m to 1

$\Rightarrow$ total number of oprn $2(m-\ell) + O(1)$

So our designed potential function will be
counter with value m-1

$T'(n) = T(n) + \phi_i$ → number of counters 2.

$T(1) = T(1) + 0$ → O(1) as discussed above.

$T(n)_{\text{amortized}} = O(1)$.

2/b) what is amortized complexity if $m = 2$ not 3

assume now $x_1 \ldots x_m$ are 1 instead of 2. ~~incre~~

⇒ increment will reset all of the counter in this interval. time taken will be.
$(m-1) + O(1)$

⇒ amortized time for one operation is $O(n)$ (after every alternate increment).

⇒ for n ~~insert~~ no of counting time taken will be $n * O(n)$
which will be $O(n^2)$.

③

a) Given : Tree T , each vertex has value val(v)

To find : local minima

constrain : O(n) time Algorithm.

Algorithm :

```
local_minima ( node n).
{
  if (n is leaf node)
      return n.
  else {
    • let c1 , c2 ... ck are k children of node n with
      values val(c1) , val(c2) -- val(ck).
    • value of n is val(n)

    if val(n) < val(Ci)  ∀  i ∈ 1 to k.
        return n
    else {
        for ci in  c1 , c2 -- ck.{
            if (val(ci) < val(n)){
                local_minima (ci)
                break;
            }
        }
    }
```

Complexity : From the above given algorithm we could able to deduce that we are traversing to only one branch of the tree with n nodes (see bcz of break statement it will go into just one path downwards.)

So the complexity of our solution is O(height of tree)
Since we don't have any control over structure of tree
⇒ height of tree = O(n) {worst case}

⇒ Complexity of our solution = O(n).

Proof of correctness :-

Our algorithm calls procedure local-minima from root of the given tree T and move downward only where the value of child is lower than parent.

In other words we can say that- path taken by algorithm is such that parent always have value which is greater than child.

⇒ if all values of children of node n are greater than parent then algorithm will stop and return the node n as local minima (as we know parent of n will also be greater than n is value) by above argument

⇒ in worst case algo will were till leaf and since there are no child to compare & parent of leaf is greater than leaf's value (again by above argument) it will return the leaf as minima.

& both of above cases will be correct solution.

3/(b).

Given: $n \times n$ grid with vertex v has value $val(v)$.

To find: local minima (in 4 neighbourhood),

constrain: $O(n)$ time complexity.

Algorithm

local_min (M : two dim square array)

{ M_{MC} : middle column of M ($n \times 1$ size).

M_{MR} : middle row of M ($1 \times n$ size).

$M[i,j] = \min (\min(M_{MC}), \min(M_{MR}))$

where i, j are endices are smallest element in M_{MC} & M_{MR}

if $|M_{MC}|$ & $|M_{MR}| == 1$.

return $M[i,j]$ -

if $(M[i,j] < M[i+1,j]$

& $< M[i,j+1]$

$< M[i-1,j]$

$< M[i,j-1])$.

return $M[i,j]$

else

~~min~~ $M[k,l] = \min(M[i+1,j], M[i,j+1], M[i-1,j]$

$M[i,j-1])$.

if k < middle_row index

if l < middle_column_index

return local_min (Top_left_submatrix)

else

return local_min (Top_Right_submatrix)

else

if l < middle_column index

return ~~bott~~ local_min (Bottom left submatrix)

else

return local_min (Bottom_right_submatrix)

Complexity: As we can see from the algorithm that we are partioning the problem into four parts and our solution lies guarentees to lie in one subpart (top left, top right, bottom-left, bottom-right).

While writing recurrence relation

$T(n) = T(n/4) + O(n)$ → iterating over middle col & row.

Solving it can give us

$n \quad O(n)$

$|$

$n/4 \quad O(n/4)$

$|$

$n/16 \quad O(n/16)$.

$$T(n) = O(n + \frac{n}{4} + \frac{n}{16} + \dots).$$
$$= O\left(n\left(\frac{1}{1-\frac{1}{4}}\right)\right)$$
$$= O(n)$$

Proof of correctness:

Claim: Subpart selected by our algorithm will always give the local minima.

Proof: we are selecting ~~middle~~ min element M[i,j] from the middle row & column it gives us two cases

CASE 1: min element M[i,j] is our minima (only if there is no neighboring element less than that)

CASE 2: there is any other neighboring element M[k,l] since it is surrounded by all elements greater than that in inner boundary of subproblem either min element or any other element in the same part will ~~be min~~ guarenteed to be min

(4)

a). Given: A [1 .. n+1], l bit string w/ one missing string

To find: missing string

constraints: A[i] th j bit can be accessed only w/ FB(i,j)
calls to FB[i,j] should be O(n).

Algorithm

```
map = [1, 1, 1, ... n times].
for j = 0 to l-1 {
    count_0 = 0
    count_1 = 0
    ones = [0,0,0, ... n times]
    zeros = [0,0,0 ... n times].
    for i in 0 to n-1 {
        if map[i] == 1
            y = FB[i, j)
            if y == 0
                count_0 ++;
                zeros[i] = 1.
            else
                count_1 ++;
                ones[i] = 1;
    }
    if count_0 < count_1.
        map = zeros ; s[j] = 0
    else
        map = ones. ; s[j] = 1
```

Complexity: number of calls to FB will be based on the number of 1's in map.

at each iteration (0 - l-1) the number of values is map in each iteration will be less than half of previous as # 1's & 0's will be ~~n/n~~ n/2.

Eg) - number of call = $\frac{n}{1} + \frac{n}{2} + \frac{n}{4} + \cdots$ O(n)

Proof of correctness:

For each position of str (missing string of size l) we are counting number of occurences at a position l is required str places in A.

Since for 2^{xe} elements # 1s = # 0s. (number of)

$\Rightarrow$ $2^{xe} - 1$ $\Rightarrow$ # 1s $\neq$ # 0s.

$\Rightarrow$ we will find the embalance and put the value 0/1 with less count value and put it in appropriate place of str (acc to outer loop).

recursively our procedure will set the value of str correctly.

Also problem size will getting smaller as we will are filtering out values that are not required strings.

4/b. **Given**: Given A[1, .. n], k missing strings

To find: all missing substring

constrain: A[i] in jth bit can be accessed in w/ FB[i,j]

calls to FB[i,j] should be O(nlogk)

Algorithm

i/p: l, ref_list: contain index of A from 0 to n-1,

k: number of s missing strings

o/p: X_{kxe} array of missing k strings of length l

```
L = l ; global variable
missing_string ( l, ref_list, k )
{ if k==0 return
  if ref_list == null return.

  c = 0
  Lt = null   // this will hold indices w/ bit-value 0
  Rt = null   //  "   "   "   "   "   "   "   "   "  1
  MSB = l-1
```

```
for i in ref_list-
    if FB(i, l-1) == 0
        lt.add(i).
    else
        rt.add(i).

k1 = 2^(l-1) - lt.length()
k2 = 2^(l-1) - rt.length().

if k1 > 0
    for j=0 to k1-1
        x[C][L-1-MSB] = 0
        C++
    missing_string(l-1, lt, k1).

if k2 > 0
    for j=0 to k2-1
        x[C][L-1-MSB] = 1
        C++
    missing_string(l-1, rt, k2).
```

Complexity:

we are dividing the reference_list into parts. and in each level of tree O(n) work is being done.

Also as division of ref_list is done according to k missing substring it will never be too unbalanced.

⇒ # number of levels in tree will always O(log k).

Summing up the whole work of bee it will give $T(n) = O(n) + O(\log k)$

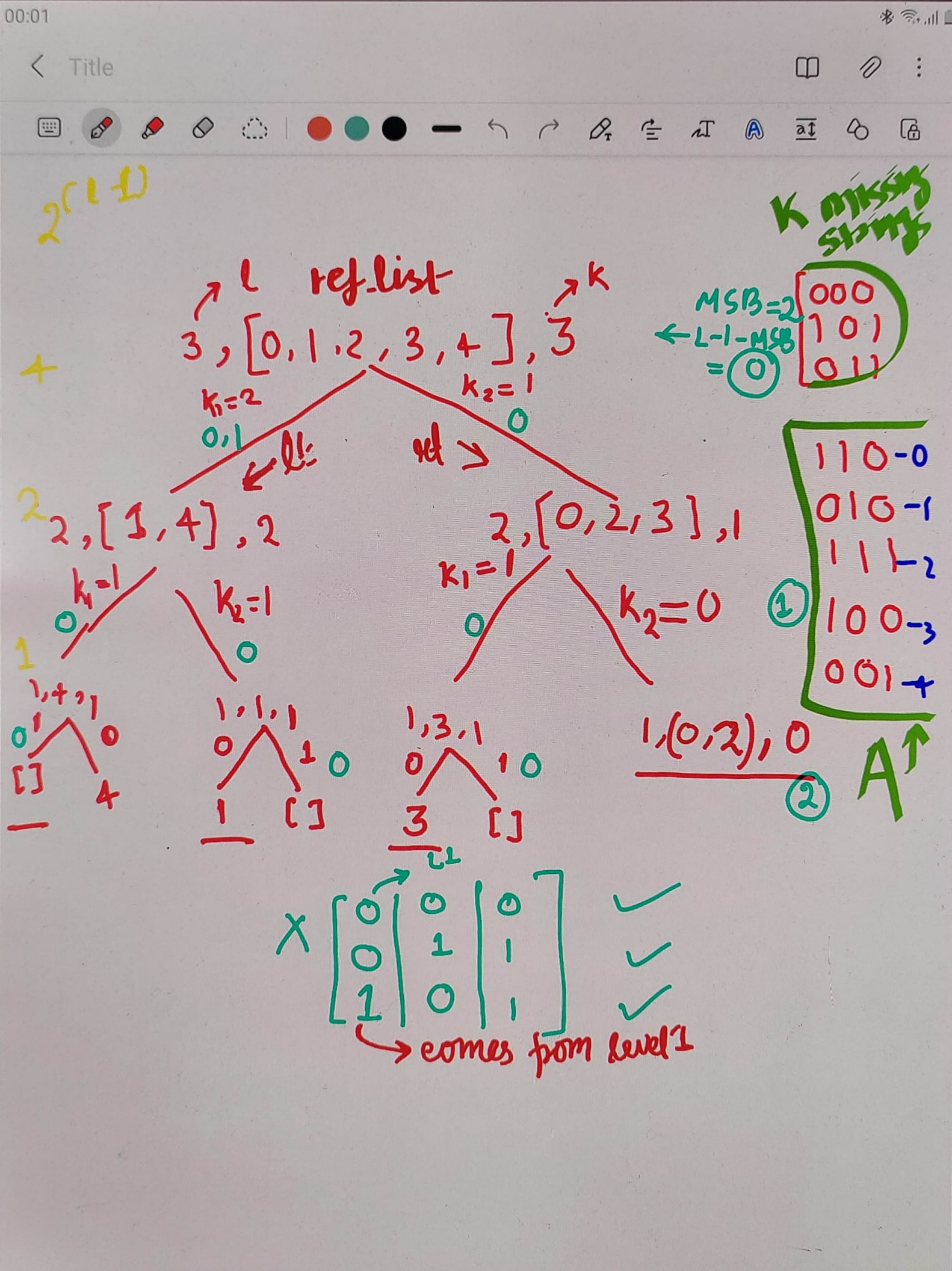
$= O(n \log k)$.

Proof of correctness :-

We are keeping track about counting of 0's & 1's (now is k_1 & k_2 instead of count_0 & count_1) value of imbalance of 0 & 1 at index i is balanced by for loops. (below two).

In this way we keep track of part of missing string in recursive manner.

at each level one positional value of each string is specified. by algorithm & loop will return when values are filled for all i bits in all levels.


2^(l+1)
K missing strings
l ref_list k
3, [0,1,2,3,4], 3
k1=2
0,1
k2=1
0
MSB=2
L-1-MSB
=0
000
101
011
2, [1,4], 2
2, [0,2,3], 1
k1=1
k2=0
k1=1
k2=1
1,4,1
1,1,1
1,3,1
1,(0,2),0
[]
4
1
[]
3
[]
110-0
010-1
111-2
100-3
001-4
A
X
comes from level 1

5) ⓐ Given: Array A [1, ... n] of real numbers.

To find: Maximum subarray sum. (cont).

Constrain: O(n) algorithm.

Algorithm:

```
max_sum = 0
running_sum = 0
for i = 0 to n-1
    running_sum = running_sum + A[i].
    if running_sum < 0
        running_sum = 0
    if running_sum > max_sum.
        max_sum = running_sum
return max_sum.
```

Complexity: There is only one pass of array and in each iteration O(1) amount of work is done

$\Rightarrow \quad T(n) = O(n)$

Proof of correctness: By induction let $S(x:i)$ = sum of all elements from index x to i

Base Case. $max\_sum(0) = max(S\,A[0]) \Rightarrow A[0]$.

Induction hypothesis

let IH is is true for max_sum (i).

$\Rightarrow max\_sum(i') = max(S(0:i), S(1,i), S(2,i) \ldots S(i,i))$

As we know.

$$S(x : i+1) = S(x : i) + A[i+1] \quad \text{—} \oplus$$

$$\text{max\_sum}(i+1) = \max(S(0 : i+1), S(1 : i+1), \ldots S(i+1 : i+1))$$

$$= \max(S(0;i) + A[i+1], S[1,i] + A[i+1], \ldots$$

$$= \max(S(0;i), S(1;i), \ldots S(i,i) + A[i+1]$$

$$\Rightarrow \text{Max}(\text{Max\_sum}(i) + A(i+1), 0]$$

↑ if $A[i+1] < 0$

$\Rightarrow \oplus$ proved by above expression.

5/b

Given: Array $A[1 \ldots n]$ of real numbers.

To find: maximum product of cont subarray

Constrain: $O(n)$ time Algorithm.

just like previous problem. we need to keep max_product & running product. unfortunately property of product is in diff cases.

1. +ve number : best case
2. -ve numbers : need to keep min_product too.
3. zero : reset running products to 1.
4. fraction w/absolute value less than 1 : modify the array to remove these numbers.

Algorithm.

1. get fractional value with minimum absolute value < 1. Let's take this value f.
2. modify Array A by multiplying each number of A to 1/f making A' with no absolute value less than 1.
3. call Algorithm A1 on A' and get the output as max_product as M
4. call Algorithm A2 on A' & get the length of maximum product subarray as L
5. return $M \cdot (f)^L$

Algorithm A1; max_product of cont. array.

```
result = max (A'(1..n)).
current_min = current_max = 1
for i in 1 to n:
    if A[i] == 0
        current_min = current_max = 1
    temp_max = current_max.
    current_max = max ( n * current_max ,
                        n * current_min , n )
    current_min = min ( n * temp_max ,
                        n * current_min , n ).
    result = max ( current_max , current_min ).
return result
```

Algorithm A2: // return maximum length of subarray product.

```
negatives = positives = result = 0
for i in 1 to n.
    if A[i] == 0
        positives = negatives = 0
    else if A[i] > 0
        positives = positives + 1.
        if Negatives != 0.
            Negatives = negatives + 1.
        result = max (result, positives).
    else.
        swap (positives, Negatives)
        Negatives = negatives + 1
        if positives != 0 <
            positives = positives + 1.
        result = max ( result, positives)
return result.
```

Complexity:

A2: O(n) only 1 loop & ~~const~~ O(1) work in each iteration.

A1: O(n) " " " " " " " "

Complexity of Algorithm

step	
1	O(n)
2	O(n)
3	O(n)
4	O(n)
5	O(1)
total =	O(n)

Proof of correctness:

Correctness of A1:

- We have used two variables for running product to deal with negative numbers. One for minimum product & one for maximum product. If the ith element of array is negative it means the minimum & maximum product will be swapped leaving the actual max product still in value current_max or current_min

Correctness of A2:

- ~~the~~ in this algorithm also there is a positive and negative count is being maintained to get indices values of largest subarray product. Counts of positive elements & negative elements are maintained if the product is positive.

Also we have handled fractional values by modifying A to A' by multiplying by 1/f where f is the absolute value of number b/w (-1, 1) - {0}

result will come ~~to~~ our scaled up which will be scaled down in step 5 according to length of interval.