香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

T26G1 - Digit Puzzle II

Nicholas Ko {__jk__}
2026-03-28

Background

Problem Idea by `mtyeung1`

Preparation by `__jk__` and `happypotato`

Problem Statement

- Input gives you N **even** integers A[1], A[2], ..., A[N].
- You put either a +, - or × between every two neighbouring integers.
- Find the maximum value of the mathematical expression formed by it.

Statistics

T26G1 - Digit Puzzle II	16	100	79.687	21.647	8: 16	10: 16	22: 13	13: 8	17: 15	24: 14	6: 1

Subtask 1 (8%): N = 2, A[1] = 0

- If A[2] = 0, the answer is 0, no matter which operator we put.
- If A[2] > 0, the answer is A[2] by putting a +.
- If A[2] < 0, the answer is -A[2] by putting a -.

- In short, the answer is abs(A[2]).
- Expected Score: 8

Subtask 2 (10%): A[1], A[2], …, A[N] are all positive

- **Observation**: for all integers a, b ≥ 2, we have a × b ≥ a + b ≥ 2.

- This means putting a * between two positive integers is never worse than putting a + between them.
- Therefore, by the observation, it is never worse to use * than + or - in this subtask.

- So the answer is A[1] × A[2] × … × A[N].
- Expected Score: 10

Subtask 3 (22%): A[1], A[2], …, A[N] are all non-negative

- Note that if at least one term in a multiplication is 0, then the final product must be 0.
- Therefore, we cannot get any non-zero value for a chain of multiplication involving a 0.

- Because all numbers are non-negative here, isolating the 0s by changing the operators to the left and right of it to be + or -) is never worse.

Subtask 3 (22%): A[1], A[2], …, A[N] are all non-negative

- We thus first partition the 0s out first, so that each subarray either is either one 0 only, or some positive integers.

- Compute the product of each of those maximal length subarrays without a 0, and sum them together to get the answer.

- Expected Score: 10 + 22 = 32

The “Big Picture” of the task

- Because × is calculated before + and -, we can interpret the + and -s as a partitioning of the array A.
- In other words, the + and -s divide A into multiple subarrays, where within each subarray there are only multiplications (and no + or -s).

- For example, if the input is A = [2, 4, 0, 6, -8], and you decide to form the expression 2 * 4 + 0 - 6 * (-8), then you get

2	*	4	+	0	-	6	*	-8	=	56

- So the first subarray has product 2 * 4 = 8, the second subarray is 0, and the third subarray is 6 * (-8) = -48.

The “Big Picture” of the task

- Why are we motivating ourselves to think about partitioning into subarrays of products, then taking their sums and differences?

- Because × is calculated before + and -, so this gives us a more organised and structured way to evaluate the expression!

- Now that we know how to handle ×, how can we handle and determine when to use + or -?

The “Big Picture” of the task

- Let’s look at our previous example again and look for more properties:

2	*	4	+	0	-	6	*	-8	=	56

- Why did we choose to put a - instead of a + in front of 6 * (-8)?

- Because the product 6 * (-8) = -48 is negative!
- To maximise the final value of the expression, we should negate the negative product so that it contributes more to the final answer.

The “Big Picture” of the task

- So from the example above, we have realised that:
  - If a subarray has product < 0, we should put a - in front of it.
  - If a subarray has product > 0, we should put a + in front of it.
  - If a subarray has product = 0, it doesn’t matter whether we put + or -.
- In other words, it contributes the absolute value of the product of the integers within a subarray towards the final answer.

- However, note that we are forced to (effectively) put a + in front of the first subarray.
- So mathematically, if we denote P[i] to be the product of the i^{th} subarray, then the final answer is P[1] + abs(P[2]) + abs(P[3]) + … + abs(P[k]) + …

Subtask 4 (13%): A[1] = 0

- For a fixed partition, the final answer is P[1] + abs(P[2]) + abs(P[3]) + ... + abs(P[k]) + ...

- But because A[1] = 0 here, we know that P[1] = 0.
- So, we are trying to maximise abs(P[2]) + abs(P[3]) + ... + abs(P[k]) + ...

- Recall that if at least one term in a multiplication is 0, then the product must be 0 (which has a small absolute value ⇒ bad for us!)

Subtask 4 (13%): A[1] = 0

- We can essentially reuse the same spirit of the solution from Subtask 3.

- Compute the product of each of those maximal length subarrays without a 0, and sum **their absolute values** together to get the answer.

- Expected Score: 10 + 22 + 13 = 45

Subtasks 5 & 6 (17% + 24%): A[1], A[2], …, A[N] are all negative

- In the solution of Subtask 2, we justified that there is no need to partition into two or more subarrays. We can do something similar here:

-2	+	-4	*	-2	*	-6	*	-2	=	94

- **Observation:** there is no need to partition into three or more subarrays (so at most two is sufficient).

- Why? This is because for every subarray except the first, we can apply the argument that for all integers $a, b \geq 2$, we have $a \times b \geq a + b \geq 2$.

Subtasks 5 & 6 (17% + 24%): A[1], A[2], …, A[N] are all negative

- If we splitted the maximal subarray which is not the first one somewhere in between, say for example we did

-2	-	-4	-	-2	*	-6	*	-2	=	26

  then doing so would never be more optimal than keeping the nonzero elements in the same subarray to obtain the absolute value of its product.
- So, we can exhaust where the cut between the two subarray is (if it exists), and for each scenario, use the formula P[1] + abs(P[2]) to calculate the answer.
- Expected Score: 17 + 24 = 41

Subtask 7 (6%): No additional constraints

- Recall our formula P[1] + abs(P[2]) + abs(P[3]) + ... + abs(P[k]) + ...
- Can we rewrite this formula in a way that is more convenient for us?

-2	+	-4	*	-2	*	-6	*	-2	=	94

- If the i-th subarray is formed by the elements A[l], A[l + 1], A[l + 2], ..., A[r], then by definition P[i] = A[l] × A[l + 1] × A[l + 2] × ... × A[r].
- Observe that

  abs(P[i]) = abs(A[l]) × abs(A[l + 1]) × abs(A[l + 2]) × ... × abs(A[r])
- Why (and how) is this form useful to us?

Subtask 7 (6%): No additional constraints

- If we fix where the first subarray ends at, then we obviously know the value of P[1].
- Can we know the value of abs(P[2]) + abs(P[3]) + ... + abs(P[k]) + ... too?

- Because we noted that abs(P[i]) = abs(A[l]) × abs(A[l + 1]) × abs(A[l + 2]) × ... × abs(A[r]), we can simply take the absolute value of each element for the remaining suffix, then apply the solution of Subtask 3, since all elements are non-negative.

- Expected Score: 100

Remarks

- Implementation can be very light if done properly, the model solution in C++ is shorter than 500 bytes :)

- Challenge: solve the problem when the integers given are not necessarily even. Can you find an efficient solution?