Before jumpting to the full solution, let's take a step back and consider simpler versions of this problem (Please scroll to the bottom of this post for the full solution):

* If the array only consisted of positive integers, the solution would be trivial: product of the entire array. We can implement this by calculating a *running* product like so:

int runningProduct = 1;

for (int n: nums) {

runningProduct \*= n;

}

return runningProduct;

* If we add zeroes to the mix, now we have to get a bit smarter. As we loop over the array, if the current number n is zero, then runningProduct would be zero and be stuck there. So, we need to reset it. Intuitively, whenever runningProduct is less than the current number n, we start considering a new sub-array that starts from the current number. Since runningProduct will be reset, we will also need a variable to hold the max runningProduct that we have encountered. Enough said, let's see how we can code this:

int best = INT\_MIN;

int runningProduct = 1;

for (int n: nums) {

// Pick the larger of current number and the result of the multiplication

// Picking n means we start considering a new sub-array

runningProduct = max(runningProduct \* n, n);

// Keep track of the max runningProduct that we find

best = max(runningProduct, best);

}

return best;

* Above, we only cared about the maximum value we can achieve and started a new sub-array using max whenever we encountered a zero. Finally, let's also consider negative numbers to find the general solution to the given problem. Consider the array 1, -2, -3, 4. When we are at the second element, our runningProduct becomes -2. But at the third element it becomes 6. So, as we encounter negative numbers they change the sign of our product. Our product can become really small but then really big depending on the sign of the current number. To accomodate for this, we can have two running products: maxProd and minProd. maxProd will track the maximum product that we can achieve and will reset whenever the current number is greater, just as above. minProd, on the other hand, will track the minimum product that we can achieve and will reset whenever the current number is smaller. Finally, whenever we encounter a negative number, we will swap maxProd and minProd because if a <= b then -b <= -a. This makes sure that maxProd and minProd are still the max and min products after multiplied by a negative number. And, with that, we have the following solution:

class Solution {

public:

int maxProduct(vector<int>& nums) {

// Return early if input is empty

if (nums.empty()) return 0;

int best = INT\_MIN;

// Running products

int maxProd = 1;

int minProd = 1;

for (int n: nums) {

if (n < 0) {

// Swap max and min

swap(maxProd, minProd);

}

// Reset to current value if smaller or larger than it

// (intuitively means that we start considering a new sub-array)

maxProd = max(maxProd\*n, n);

minProd = min(minProd\*n, n);

// Update the best

best = max(best, maxProd);

}

return best;

}

};