Interview Cake

Given a vector of integers, find the highest product you can get from three of the integers.

The input `vectorOfInts` will always have at least three integers.

Gotchas

Does your function work with negative numbers? If `vectorOfInts` is $[-10, -10, 1, 3, 2]$ we should return 300 (which we get by taking $-10 * -10 * 3$).

We can do this in $O(n)$ time and $O(1)$ space.

Breakdown

To brute force an answer we could iterate through `vectorOfInts` and multiply each integer by each *other* integer, and then multiply that product by each *other other* integer. This would probably involve nesting 3 loops. But that would be an $O(n^3)$ runtime! We can *definitely* do better than that.

Because any integer in the vector could potentially be part of the greatest product of three integers, we must at least *look* at *each integer*. So we're doomed to spend at least $O(n)$ time.

Sorting the vector would let us grab the highest numbers quickly, so it might be a good first step. Sorting takes $O(n \lg n)$ time. That's better than the $O(n^3)$ time our brute force approach required, but we can still do better.

Since we know we must spend *at least* $O(n)$ time, let's see if we can solve it in *exactly* $O(n)$ time.

A great way to get $O(n)$ runtime is to use a greedy approach. **How can we keep track of the `highestProductOf3` "so far" as we do one walk through the vector?**

Put differently, for each new `current` number during our iteration, how do we know if it gives us a new `highestProductOf3`?

We have a new `highestProductOf3` if the `current` number times two other numbers gives a product that's higher than our current `highestProductOf3`. **What must we keep track of at each step so that we know if the `current` number times two other numbers gives us a new `highestProductOf3`?**

Our first guess might be:

1. our current `highestProductOf3`
2. the `threeNumbersWhichGiveHighestProduct`

But consider this example:

```
vector<int> vectorOfInts({1, 10, -5, 1, -100});
```

C++

Right before we hit -100 (so, in our second-to-last iteration), our `highestProductOf3` was 10, and the `threeNumbersWhichGiveHighestProduct` were $[10, 1, 1]$. But once we hit -100, suddenly we can take $-100 * -5 * 10$ to get 5000. So we should have "held on to" that -5, even though it wasn't one of the `threeNumbersWhichGiveHighestProduct`.

We need something a little smarter than `threeNumbersWhichGiveHighestProduct`. **What should we keep track of to make sure we can handle a case like this?**

There are at least two great answers:

1. **Keep track of the `highest2` and `lowest2` (most negative) numbers**. If the `current` number times *some combination of those* is higher than the current `highestProductOf3`, we have a new `highestProductOf3`!
2. **Keep track of the `highestProductOf2` and `lowestProductOf2`** (could be a low negative number). If the `current` number times one of those is higher than the current `highestProductOf3`, we have a new `highestProductOf3`!

We'll go with (2). It ends up being *slightly* cleaner than (1), though they both work just fine.

How do we keep track of the `highestProductOf2` and `lowestProductOf2` at each iteration? (Hint: we may need to also keep track of *something else*.)

We also keep track of the `lowest` number and `highest` number. If the `current` number times the current `highest`—*or the current `lowest`*, if `current` is negative—is greater than the current `highestProductOf2`, we have a new `highestProductOf2`. Same for `lowestProductOf2`.

So at each iteration we're keeping track of and updating:

- `highestProductOf3`
- `highestProductOf2`
- `highest`
- `lowestProductOf2`
- `lowest`

Can you implement this in code? **Careful—make sure you update each of these variables in the right order**, otherwise you might end up e.g. multiplying the `current` number by itself to get a new `highestProductOf2`.

Solution

We use a greedy approach to solve the problem in one pass. At each iteration we keep track of:

- `highestProductOf3`
- `highestProductOf2`
- `highest`
- `lowestProductOf2`
- `lowest`

When we reach the end, the `highestProductOf3` is our answer. We maintain the others because they're necessary for keeping the `highestProductOf3` up to date as we walk through the vector. At each iteration, the `highestProductOf3` is the highest of:

1. the current `highestProductOf3`
2. `current * highestProductOf2`
3. `current * lowestProductOf2` (if `current` and `lowestProductOf2` are both low negative numbers, this product is a high positive number).

C++

```
int highestProductOf3(const vector<int>& vectorOfInts)
{
    if (vectorOfInts.size() < 3) {
        throw invalid_argument("Less than 3 items!");
    }

    // we're going to start at the 3rd item (at index 2)
    // so pre-populate highests and lowests based on the first 2 items.
    // we could also start these as null and check below if they're set
    // but this is arguably cleaner
    int highest = max(vectorOfInts[0], vectorOfInts[1]);
    int lowest  = min(vectorOfInts[0], vectorOfInts[1]);

    int highestProductOf2 = vectorOfInts[0] * vectorOfInts[1];
    int lowestProductOf2  = vectorOfInts[0] * vectorOfInts[1];

    // except this one--we pre-populate it for the first *3* items.
    // this means in our first pass it'll check against itself, which is fine.
    int highestProductOf3 = vectorOfInts[0] * vectorOfInts[1] * vectorOfInts[2];

    // walk through items, starting at index 2
    for (size_t i = 2; i < vectorOfInts.size(); ++i) {
        int current = vectorOfInts[i];

        // do we have a new highest product of 3?
        // it's either the current highest,
        // or the current times the highest product of two
        // or the current times the lowest product of two
        highestProductOf3 = max(max(
            highestProductOf3,
            current * highestProductOf2),
            current * lowestProductOf2);

        // do we have a new highest product of two?
        highestProductOf2 = max(max(
            highestProductOf2,
            current * highest),
            current * lowest);

        // do we have a new lowest product of two?
```

```
            lowestProductOf2 = min(min(
                lowestProductOf2,
                current * highest),
                current * lowest);

            // do we have a new highest?
            highest = max(highest, current);

            // do we have a new lowest?
            lowest = min(lowest, current);
        }

        return highestProductOf3;
    }
```

Complexity

$O(n)$ time and $O(1)$ additional space.

Bonus

1. What if we wanted the highest product of 4 items?
2. What if we wanted the highest product of k items?
3. If our highest product is really big, it could overflow. How should we protect against this?

What We Learned

Greedy algorithms in action again!

That's not a coincidence—to illustrate how one pattern can be used to break down several different questions, we're showing this one pattern in action on several different questions.

Usually it takes seeing an algorithmic idea from a few different angles for it to really make intuitive sense.

Our goal here is to teach you the right *way of thinking* to be able to break down problems you haven't seen before. Greedy algorithm design is a big part of that *way of thinking*.

For this one, we built up our greedy algorithm exactly the same way we did for the Apple stocks (/question/stock-price) question. By asking ourselves:

"Suppose we *could* come up with the answer in one pass through the input, by simply updating the 'best answer so far' as we went. What *additional values* would we need to keep updated as we looked at each item in our set, in order to be able to update the 'best answer so far' in constant time?"

For the Apple stocks question, the only "additional value" we needed was the min price so far.

For this one, we needed *four* things in order to calculate the new `highestProductOf3` at each step:

- `highestProductOf2`
- `highest`
- `lowestProductOf2`
- `lowest`