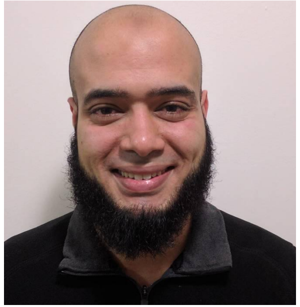
Cracking Coding Interviews

Maximum Product of 3 Numbers

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Leetcode 628 - Maximum Product of 3 Numbers

- Given an integer array, find 3 numbers whose product is maximum.
- Input ⇒ Output
  - [1,2,3,4] ⇒ 24   from 2x3x4
  - [-1,-2,-3] ⇒ -6
- Function
  - C++: int maximumProduct(vector<int>& nums)
  - Java: public int maximumProduct(int[] nums)
  - Python: def maximumProduct(self, nums: List[int]) -> int

Your turn

- Ask the right questions, if any, and state your assumptions
- Develop some test cases

Assumptions:

- You
  - I assume you want 3 different indices
  - I assume the multiplication of any 3 values won't overflow!
    - This is relevant for languages that will overflow, such as C++
    - You might be more specific (e.g. MAX_INT in c++ is the integer limit)
  - I assume the array will have at least 3 values
- Interviewer
  - That is ok. Go ahead!

Test cases

- We've already been given test cases with all positive or all negative numbers
- Let's develop a case with a mixture of positive and negative values
- [**-4,** 5, **-6**, 2, **7**] ⇒ -4 * -6 * 7 = -4 * -6 * 7
- [-2, -3, **5, 2, 7**] ⇒ 5 * 2 * 7 ⇒ 70

Your turn

- Can we approach it with brute-force? If so, how?

Brute-force it!

- As we need 3 indices, clearly we can try 3 nested loops, and compute the max among all triplets!
    - For different indices: i = 0, then j = i + 1, then k = j + 1
    - I use (i, j, k) to refer a lot to 3 nested loops
- Clearly this is $O(n^3)$.
- Obviously, this is not the intended
- What might be a potential order? O(n^2)? O(nlogn)? O(n)
- Can we find a way to optimize the brute-force solution?
    - We need to remove 1 or 2 loops!

Brute-force it!

- Can we find a way to optimize the brute-force solution?
- It may not look like there's a way to optimize this brute-force solution
  - Or, at least, it may not be clear how to do so!
- Let’s analyze more or utilize some thinking tools

Your turn

- Try to discover some observations / properties

Your turn

- Try to discover some observations / properties
- For sake of practice: think in different problem simplifications

Problem Simplification

- Simpler version 1: Maximum Product of 2 Numbers
  - Observe: multiplying 2 positive or 2 negative numbers gives a positive result
  - If available: the largest 2 positive values or the smallest 2 negative values are the answer
- Simpler version 2: Assume all are positive (added a constraint)
  - E.g. {2, 3, 4, 5, 6}: Clearly the answer is the largest 3 numbers: 4*5*6
- Simpler version 3: Assume all are negative (added a constraint)
  - E.g. {-6, -5, -4, -3, -2}: Hmm
  - Similar logic: we need the 3 largest values; all possible results are negative in any case ⇒ -2 * -3 * -4 = -24
- Observation: this problem is all about *several max/min values* and some choices based on that
- Now try to find some observation for mixed cases
  - Sorting the numbers helps us figure out the smallest/largest values

Observation

- Sorting the numbers helps us figure out the smallest/largest values
- **-20**, **-10**, -7, -6, -5, 2, **3, 4, 5**
  - A mix of multiple positive and negative values
  - What are the potential answers?
  - In this case, our answer is DEFINITELY positive
  - It could potentially be the product of the 3 largest values
  - However, if we take the 2 smallest negative values:: -20 x -10 = 200 positive value
  - And then we take the largest positive value: 5
  - Our total is 1000, which is larger than 3*4*5!

Verifying special cases

- General case: Either 3 positive values or 2 negative and 1 positive values
- What if we have a single positive value in the array?
  - Then our answer combines the 2 smallest negative values with positive value
- What if we have a single negative value in the array and 3 positive?
  - Clearly, this negative value is useless; we will use the largest 3 positive values
- Observation: The general rule works even for special test-cases
- So sort data in O(nlogn)
  - Get the largest 3 positive
  - Get the smallest 2 negatives and largest positive
  - Compute and get the maximum value
- O(1) memory

O(nlogn) time and O(1) memory

- The interviewer is happy, but wonders if it can be improved?
- Give a trial

```
int maximumProduct(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int n = nums.size();
    int a = nums[n - 1] * nums[n - 2] * nums[n - 3];
    int b = nums[0] * nums[1] * nums[n - 1];
    return max(a, b);
}
```

Optimized version

- Clearly, we need to:
    - Get the three largest values
    - And get the two smallest values
- In a single loop compute the 3 largest values and the 2 smallest values
    - It is just careful if-else code
- Now, it’s O(n) time
- There is a more elegant way than loops with a tricky if-else
    - Can u use a max-heap to get the max 3 values? And min-heap for the smallest 2?
    - And keep it O(1) memory and O(n) time?
- Your turn: try to code the 2 approaches

O(n)
loops style

```cpp
int maximumProduct(vector<int>& nums) {
    int max1 = INT_MIN, max2 = INT_MIN, max3 = INT_MIN;
    int min1 = INT_MAX, min2 = INT_MAX;

    for (int i = 0; i < (int) nums.size(); i++) {
        if (nums[i] <= min1)
            min2 = min1, min1 = nums[i];
        else if (nums[i] <= min2)
            min2 = nums[i];

        if (nums[i] >= max1)
            max3 = max2, max2 = max1, max1 = nums[i];
        else if (nums[i] >= max2)
            max3 = max2, max2 = nums[i];
        else if (nums[i] >= max3)
            max3 = nums[i];
    }
    return max(min1 * min2 * max1, max1 * max2 * max3);
}
```

O(n) using max & min heaps

```
int maximumProduct(vector<int>& nums) {
    priority_queue<int> mx_heap;    // for smallest 2 numbers
    priority_queue <int, vector<int>, greater<int>> mn_heap;

    for (int i = 0; i < (int) nums.size(); i++) {
        mx_heap.push(nums[i]);
        mn_heap.push(nums[i]);

        if(mx_heap.size() > 2)
            mx_heap.pop();

        if(mn_heap.size() > 3)
            mn_heap.pop();  // keep largest 3
    }
    int max1, max2, max3, min1, min2;
    max3 = mn_heap.top(), mn_heap.pop();
    max2 = mn_heap.top(), mn_heap.pop();
    max1 = mn_heap.top(), mn_heap.pop();

    min2 = mx_heap.top(), mx_heap.pop();
    min1 = mx_heap.top(), mx_heap.pop();

    return max(min1 * min2 * max1, max1 * max2 * max3);
}
```

O(n) using max & min heaps

- In Python, the code is way shorter and simpler with heapq!
    - nlargest is O(n log k) for selecting k elements. K here is 2 or 3
- Tip: if you're comfortable with multiple programming languages, pick the one that produces the shortest code

```python
import heapq

class Solution:
    def maximumProduct(self, array):
        largest = heapq.nlargest(3, array)
        smallest = heapq.nsmallest(2,array)

        return max(largest[0] * largest[1] * largest[2],
                   largest[0] * smallest[0] * smallest[1] )
```

Another O(n) optimized BF solution

- Let’s brute force one of the 3 values
  - E.g. iterate on every single value and consider it part of the solution
- The remaining 2 values?
  - Instead of 2 nested loops, we can figure out the best pair of values to add
  - To do this in O(1), we need to perform some *pre-computations*
- Give a trial

- Hint: We will use the left[idx] and right[idx] pre-computations style

Another O(n) optimized BF solution

- Let’s brute force one of the 3 values
  - E.g. iterate on every single value and consider it part of the solution
- The remaining 2 values?
  - Assume the BF value is iterating on the **middle** variable
  - The other two positions: one on the left, one on the right
  - Instead of 2 nested loops, we can figure out the best pair of values to add
  - To do this in O(1), we need to perform some *pre-computations*
- Hint
  - We will use the left[idx] and right[idx] pre-computations style
  - The 2 values will either be min or max values!
  - Construct four auxiliary arrays left_mx[], right_mx[], left_mn[] and right_mn[]
  - left_mx[idx]: the max in range {0, idx-1}
  - right_mx[idx]: the max in range {idx+1, size-1}

Another O(n) optimized BF solution

	5	**2**	**6**	**3**	**1**	**8**	**4**	**-2**	**12**
min_left[idx]	NA	5	2	2	2	1	1	1	-2
max_left[idx]	NA	5	5	6	6	6	8	8	8
min_right[idx]	-2	-2	-2	-2	-2	-2	-2	12	NA
max_right[idx]	12	12	12	12	12	12	12	12	NA

Another O(n) optimized BF solution

```
// mn[i]: min in range {0, i-1}
// mx[i]: max in range {0, i-1}
void left_min_max(vector<int>& nums, vector<int>& mn, vector<int>& mx) {
    mn = mx = nums;
    for (int i = 1; i < (int)nums.size(); ++i) {
        mn[i] = min(mn[i-1], nums[i-1]);
        mx[i] = max(mx[i-1], nums[i-1]);
    }
}

// mn[i]: min in range {i+1, size-1}
// mx[i]: max in range {i+1, size-1}
void right_min_max(vector<int>& nums, vector<int>& mn, vector<int>& mx) {
    mn = mx = nums;
    for (int i = (int)nums.size()-2; i >= 0; --i) {
        mn[i] = min(mn[i+1], nums[i+1]);
        mx[i] = max(mx[i+1], nums[i+1]);
    }
}
```

Another O(n) optimized BF solution

- Let’s brute force one of the 3 values
    - E.g. iterate on every single value and consider it part of the solution
- The remaining 2 values?
    - Now we need to analyze how to use the 4 arrays? No
    - Just go brute-force: there are 4 cases anyway. Don’t waste time on analysis
        - left_mn[idx], idx, right_mn[idx]
        - left_mn[idx], idx, right_mx[idx]
        - left_mx[idx], idx, right_mn[idx]
        - left_mx[idx], idx, right_mx[idx]

Another O(n) optimized BF solution

```
int maximumProduct(vector<int>& nums) {
    int max_product = INT_MIN;
    vector<int> mn_left, mx_left, mn_right, mx_right;

    left_min_max(nums, mn_left, mx_left);
    right_min_max(nums, mn_right, mx_right);

    for (int i = 1; i < (int)nums.size()-1; i++) {  // bf a position
        // bf the 4 possible cases
        max_product = max(max_product, mn_left[i] * nums[i] * mn_right[i]);
        max_product = max(max_product, mn_left[i] * nums[i] * mx_right[i]);
        max_product = max(max_product, mx_left[i] * nums[i] * mn_right[i]);
        max_product = max(max_product, mx_left[i] * nums[i] * mx_right[i]);
    }
    return max_product;
}
```

Another O(n) optimized BF solution

- Clearly, this is O(n) time, but also O(n) memory
- This solution involves 2 tricks
  - Don't automatically dismiss the brute force approach; it's not necessarily a completely mindless algorithm
  - Left[idx], right[idx] style is a key to some problems
- As you see, in an ad-hoc problem there is no specific pattern to solve!
  - However, keep this style of processing for problems that search for 3 values
  - Use the middle value (i.e brute-force it) and search for the value before and value after
  - This may reduce the order
  - Other examples: find all triplets that form an arithmetic/geometric progression

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."