<https://leetcode.com/problems/kth-smallest-product-of-two-sorted-arrays/description/>

Given two sorted 0-indexed integer arrays nums1 and nums2 as well as an integer k, return the kth (1-based) smallest product of nums1[i] \* nums2[j] where 0 <= i < nums1.length and 0 <= j < nums2.length.

**Example 1:**

**Input:** nums1 = [2,5], nums2 = [3,4], k = 2

**Output:** 8

**Explanation:** The 2 smallest products are:

- nums1[0] \* nums2[0] = 2 \* 3 = 6

- nums1[0] \* nums2[1] = 2 \* 4 = 8

The 2nd smallest product is 8.

**Example 2:**

**Input:** nums1 = [-4,-2,0,3], nums2 = [2,4], k = 6

**Output:** 0

**Explanation:** The 6 smallest products are:

- nums1[0] \* nums2[1] = (-4) \* 4 = -16

- nums1[0] \* nums2[0] = (-4) \* 2 = -8

- nums1[1] \* nums2[1] = (-2) \* 4 = -8

- nums1[1] \* nums2[0] = (-2) \* 2 = -4

- nums1[2] \* nums2[0] = 0 \* 2 = 0

- nums1[2] \* nums2[1] = 0 \* 4 = 0

The 6th smallest product is 0.

**Example 3:**

**Input:** nums1 = [-2,-1,0,1,2], nums2 = [-3,-1,2,4,5], k = 3

**Output:** -6

**Explanation:** The 3 smallest products are:

- nums1[0] \* nums2[4] = (-2) \* 5 = -10

- nums1[0] \* nums2[3] = (-2) \* 4 = -8

- nums1[4] \* nums2[0] = 2 \* (-3) = -6

The 3rd smallest product is -6.

**Attempt 1: 2024-01-27**

**Solution 1: Binary Search (720 min)**

**A bit similar mechanism on L2824.Count Pairs Whose Sum is Less than Target Two Pointers solution**

class Solution {

    public long kthSmallestProduct(int[] nums1, int[] nums2, long k) {

        // 'neg1' store all negative numbers from 'nums1'

        List<Integer> neg1 = Arrays.stream(nums1)

                .filter(num -> num < 0)

                .boxed() // Convert int to Integer

                .collect(Collectors.toList());

        // 'pos1' store all non-negative numbers from 'nums1'

        List<Integer> pos1 = Arrays.stream(nums1)

                .filter(num -> num >= 0)

                .boxed()

                .collect(Collectors.toList());

        // 'neg2' store all negative numbers from 'nums2'

        List<Integer> neg2 = Arrays.stream(nums2)

                .filter(num -> num < 0)

                .boxed()

                .collect(Collectors.toList());

        // 'pos2' store all non-negative numbers from 'nums2'

        List<Integer> pos2 = Arrays.stream(nums2)

                .filter(num -> num >= 0)

                .boxed()

                .collect(Collectors.toList());

        // Convert 'neg1' and 'neg2' negative numbers to positive

        neg1.replaceAll(Math::abs);

        neg2.replaceAll(Math::abs);

        // Reverse order to keep increasing

        Collections.reverse(neg1);

        Collections.reverse(neg2);

        // Find total negative products count

        long negCount = neg1.size() \* pos2.size() + neg2.size() \* pos1.size();

        int sign = 1;

        // Target kth value > 0, search in the positive parts

        // e.g neg1 \* neg2 & pos1 \* pos2

        if(k > negCount) {

            k -= negCount;

        // Target kth value <= 0, search in the negative parts

        // e.g neg1 \* pos1 & neg2 \* pos2

        } else {

            // We search in neg1 \* pos1 & neg2 \* pos2, and search as positive,

            // so k should be reversed

            k = negCount - k + 1;

            // Also switch neg2 and pos2 to keep the following code consistent

            // the critical thing is 'keep the following code consistent',

            // because only after switch neg2 and pos2, we don't need to modify

            // a single word on binary search code section but completely switch

            // binary search from "count(neg1, neg2, mid) + count(pos1, pos2, mid)"

            // to "count(neg1, pos2, mid) + count(pos1, neg2, mid)"

            List<Integer> tmp = neg2;

            neg2 = pos2;

            pos2 = tmp;

            // The signal update from positive to negative

            sign = -1;

        }

        // Binary Search to find minimum 'mid' satisfy total pair count >= k

        // Find lower boundary + Auxiliary method

        long lo = 0;

        long hi = (long)1e10; // -10^5 <= nums1[i], nums2[j] <= 10^5

        while(lo <= hi) {

            long mid = lo + (hi - lo) / 2;

            if(count(neg1, neg2, mid) + count(pos1, pos2, mid) >= k) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo \* sign;

    }

    // Find how many list1[i] \* list2[j] product no more than 'val'

    public long count(List<Integer> list1, List<Integer> list2, long val) {

        long result = 0;

        int j = list2.size() - 1;

        for(int a : list1) {

            // Search for the max j satisfying the product <= val

            while(j >= 0 && (long)a \* (long)list2.get(j) > val) {

                j--;

            }

            result += j + 1;

        }

        return result;

    }

}

Time Complexity: O((M + N)log10^10)

Space Complexity: O(M + N)

**Step by Step**

Step 1: Create neg1 / pos1 / neg2 / pos2

nums1 = [ -2, -1, 0, 1, 2 ]

|--[neg1]--|--[pos1]--]

nums2 = [ -3, -1, 2, 4, 5 ]

|--[neg2]--|--[pos2]--|

Step 2: Inverse neg1 / neg2

Math::abs(neg1), Math::abs(neg2)

nums1 = [ 2, 1, 0, 1, 2 ]

|--[neg1]--|--[pos1]--]

nums2 = [ 3, 1, 2, 4, 5 ]

|--[neg2]--|--[pos2]--|

Step 3: Reverse neg1 / neg2

reverse(neg1), reverse(neg2)

nums1 = [ 1, 2, 0, 1, 2 ]

|--[neg1]--|--[pos1]--]

nums2 = [ 1, 3, 2, 4, 5 ]

|--[neg2]--|--[pos2]--|

Step 4: Check relation between k and negCount

negCount = neg1.size() \* pos2.size() + neg2.size() \* pos1.size() = 12

Since its in negative section, when we turn every negative number

to positive and reverse to increasing order, it will only create asending

positive product, instead of find kth smallest product, we need to find

(negCount - k + 1)th largest product(including sign)

k < negCount -> k = negCount - k + 1 = 12 - 3 + 1 = 10

e.g

In above if the original nums1[i] \* nums2[j] negative part will be:

nums1 = [ -2, -1, 0, 1, 2 ]

|--[neg1]--|--[pos1]--]

nums2 = [ -3, -1, 2, 4, 5 ]

|--[neg2]--|--[pos2]--|

neg1[i] \* pos2[j] = -4,-8,-10,-2,-4,-5

pos1[i] \* neg2[j] = 0,0,-3,-1,-6,-2

but ideally, if we merge two sets and sort together manually the combined

negative part will be:

-10,-8,-6,-5,-4,-4,-3,-2,-2,-1,0,0 -> the k = 3 smallest one is -6

And after we convert original input from Step 1 to Step 3, we have

nums1 = [ 1, 2, 0, 1, 2 ]

|--[neg1]--|--[pos1]--]

nums2 = [ 1, 3, 2, 4, 5 ]

|--[neg2]--|--[pos2]--|

neg1[i] \* pos2[j] = 2,4,5,4,8,10 (2,4,5 monopoly ascending, multiply sign -1 monopoly descending, same for 4,8,10)

pos1[i] \* neg2[j] = 0,0,1,3,2,6 (0,0 monopoly ascending, multiply sign -1 monopoly descending, same for 1,3 and 2,6)

the neg1[i] \* pos2[j] and pos1[i] \* neg2[j] monopoly ascending attribute will be used in count()

method to find range easily by increasing i in for loop while decreasing j in while loop

public long count(List<Integer> list1, List<Integer> list2, long val) {

long result = 0;

int j = list2.size() - 1;

for(int a : list1) {

// Search for the max j satisfying the product <= val

while(j >= 0 && (long)a \* (long)list2.get(j) > val) {

j--;

}

result += j + 1;

}

return result;

}

if we combine two sets together will be

0,0,1,2,2,3,4,4,5,6,8,10 -> the mapping one for k = 3 smallest one as -6

which getting previous from original input should be (negCount - k + 1)th

largest product(including sign) here, which is (12 - 3 + 1) = 10th one as

6 and multiply sign = -1 here, so 6 \* (-1) = -6 match the ideal result

**Test code**

import java.util.\*;

import java.util.stream.Collectors;

public class Solution {

public long kthSmallestProduct(int[] nums1, int[] nums2, long k) {

// 'neg1' store all negative numbers from 'nums1'

List<Integer> neg1 = Arrays.stream(nums1)

.filter(num -> num < 0)

.boxed() // Convert int to Integer

.collect(Collectors.toList());

// 'pos1' store all non-negative numbers from 'nums1'

List<Integer> pos1 = Arrays.stream(nums1)

.filter(num -> num >= 0)

.boxed()

.collect(Collectors.toList());

// 'neg2' store all negative numbers from 'nums2'

List<Integer> neg2 = Arrays.stream(nums2)

.filter(num -> num < 0)

.boxed()

.collect(Collectors.toList());

// 'pos2' store all non-negative numbers from 'nums2'

List<Integer> pos2 = Arrays.stream(nums2)

.filter(num -> num >= 0)

.boxed()

.collect(Collectors.toList());

// Convert 'neg1' and 'neg2' negative numbers to positive

neg1.replaceAll(Math::abs);

neg2.replaceAll(Math::abs);

// Reverse order to keep increasing

Collections.reverse(neg1);

Collections.reverse(neg2);

// Find total negative products count

long negCount = neg1.size() \* pos2.size() + neg2.size() \* pos1.size();

int sign = 1;

// Target kth value > 0, search in the positive parts

// e.g neg1 \* neg2 & pos1 \* pos2

if(k > negCount) {

k -= negCount;

// Target kth value <= 0, search in the negative parts

// e.g neg1 \* pos1 & neg2 \* pos2

} else {

// We search in neg1 \* pos1 & neg2 \* pos2, and search as positive,

// so k should be reversed

k = negCount - k + 1;

// Also switch neg2 and pos2 to keep the following code consistent

// the critical thing is 'keep the following code consistent',

// because only after switch neg2 and pos2, we don't need to modify

// a single word on binary search code section but completely switch

// binary search from "count(neg1, neg2, mid) + count(pos1, pos2, mid)"

// to "count(neg1, pos2, mid) + count(pos1, neg2, mid)"

List<Integer> tmp = neg2;

neg2 = pos2;

pos2 = tmp;

// The signal update from positive to negative

sign = -1;

}

// Binary Search to find

// Find lower boundary + Auxiliary method

long lo = 0;

//long hi = (long)1e10; // -10^5 <= nums1[i], nums2[j] <= 10^5

long hi = 100;

while(lo <= hi) {

long mid = lo + (hi - lo) / 2;

if(count(neg1, neg2, mid) + count(pos1, pos2, mid) >= k) {

hi = mid - 1;

} else {

lo = mid + 1;

}

}

return lo \* sign;

}

// Find how many list1[i] \* list2[j] product no more than 'val'

public long count(List<Integer> list1, List<Integer> list2, long val) {

long result = 0;

int j = list2.size() - 1;

for(int a : list1) {

// Search for the max j satisfying the product <= val

while(j >= 0 && (long)a \* (long)list2.get(j) > val) {

j--;

}

result += j + 1;

}

return result;

}

public static void main(String[] args) {

Solution so = new Solution();

int[] nums1 = new int[]{-2,-1,0,1,2};

int[] nums2 = new int[]{-3,-1,2,4,5};

int k = 3;

//int[] nums1 = new int[]{3};

//int[] nums2 = new int[]{-3};

//int k = 1;

long result = so.kthSmallestProduct(nums1, nums2, k);

System.out.println(result);

}

}

**Refer to**

<https://leetcode.com/problems/kth-smallest-product-of-two-sorted-arrays/solutions/1524312/python-binary-search-o-m-n-log10-10/comments/1138923>

<https://leetcode.com/problems/kth-smallest-product-of-two-sorted-arrays/solutions/1524312/python-binary-search-o-m-n-log10-10/comments/1117889>

Let me share my understanding

first, if we split the two arrays into four arrays

negative\_A, positive\_A,

negative\_B, positive\_B

then negative\_A[i] \* positive\_B[j]and negative\_B[i]\*positive\_A [j] are negative,

while negative\_A[i]\*negative\_B[j] and positive\_A[i]\*positive\_B[j] are positive

Now we have two cases:

Case 1: k <= n\_negative = length(negative\_A) \* length(positive\_B) + length(negative\_B) \* length(positive\_A),

Our answer will be k-th smallest in all the combinations of negative\_A[i] \* positive\_B [j] and negative\_B[i]\*positive\_A [j]

Case 2: k > n\_negative = length(negative\_A) \* length(positive\_B) + length(negative\_B) \* length(positive\_A)

Then the answer is k-n\_negative-th smallest in all the combinations of negative\_A[i]\*negative\_B[j] and positive\_A[i]\*positive\_B[j]

So for either case, we end up with two pairs of lists, say (A1, B1) and (A2, B2), and we want to find the K-th smallest product among these products. Note that here K may not be the given k in the problem as discussed above.

We can further make all the lists (A1, B1) and (A2, B2) positive and modify K accordingly

(it may become the problem of finding the smallest number\_of\_products-K+1-th product)

Anyway, in the end, the problem becomes: find this K-th smallest among the products of A1[i]\*B1[j] and A2[i]\*B2[j].

To solve this, you can use binary search to search the answer in the range [0, 10\*\*10]

So for a given number target, we need to find the position of target among all these products. This means to find how many pairs of A[i]\*B[j]<=target in (A1, B1) and (A2, B2)

The very first number target that has K pairs of products less than it is the answer.

This is the main idea of the solution in this post. I hope this helps.

class Solution {

public:

long long count(vector<long long>& arr1, vector<long long>& arr2, long long x) {

long long res = 0, j = arr2.size()-1;

for (long long i = 0; i < arr1.size(); ++i) { // for every i in arr1

while (j >= 0 && arr1[i]\*arr2[j] > x) // search for the max j satisfying the product <= x

--j;

res += j+1;

}

return res;

}

long long kthSmallestProduct(vector<int>& nums1, vector<int>& nums2, long long k) {

int m = nums1.size(), n = nums2.size();

auto it1 = lower\_bound(nums1.begin(), nums1.end(), 0);

auto it2 = lower\_bound(nums2.begin(), nums2.end(), 0);

vector<long long> neg1(nums1.begin(), it1), pos1(it1, nums1.end()); // pos includes 0

vector<long long> neg2(nums2.begin(), it2), pos2(it2, nums2.end());

for\_each(neg1.begin(), neg1.end(), [](long long& x){ x \*= -1; }); // make neg array positive

for\_each(neg2.begin(), neg2.end(), [](long long& x){ x \*= -1; });

reverse(neg1.begin(), neg1.end()); // reverse order to keep increasing

reverse(neg2.begin(), neg2.end());

long long negcnt = neg1.size()\*pos2.size() + neg2.size()\*pos1.size();

bool negtarget;

if (k > negcnt) { // target >= 0

k -= negcnt; // search in the positive parts, i.e. neg1\*neg2 & pos1\*pos2

negtarget = 0;

} else { // target < 0

k = negcnt - k + 1; // search in the negative parts

// we search in neg1\*pos1 & neg2\*pos2, and search as positive, so k should be reversed

// switch neg2 and pos2 to keep the following code consistent

vector<long long> tmp = neg2;

neg2 = pos2;

pos2 = tmp;

negtarget = 1;

}

long long left = 0, right = 1e10;

while (left < right) {

long long mid = left+(right-left)/2;

if (count(neg1, neg2, mid) + count(pos1, pos2, mid) >= k)

right = mid;

else

left = mid+1;

}

return negtarget ? -left : left;

}

};

**Refer to**

<https://leetcode.com/problems/kth-smallest-product-of-two-sorted-arrays/solutions/1524856/binary-search-with-detailed-explanation/>

**Note: credit goes to lee215 for the inverse/reverse trick**

**Step 1. k'th smallest product for lists with only positive numbers**

Imagine that there are only positive numbers, and we want to find the k'th smallest number.

Consider the two lists a=[1,2,3] and b=[1,5,8]. Their products form a matrix:

|  |  |  |  |
| --- | --- | --- | --- |
| - | 1 | 2 | 3 |
| **1** | 1 | 2 | 3 |
| **5** | 5 | 10 | 15 |
| **8** | 8 | 18 | 24 |

Assuming that we guess a number x, then it's possible to quickly determine how many numbers are smaller than x. Just iterate row by row, moving a right pointer left until the condition is satisfied and count the number of elements before the pointer:

func count(a, b []int, val int) int {

var count int

j := len(b) - 1

for i := range a {

for j >= 0 && a[i]\*b[j] > val {

j--

}

count += j + 1

}

return count

}

With this approach, it's easy to find the k'th positive number using binary search:

var lo, hi int = 0, 1e10

for lo < hi {

mid := (lo + hi) / 2

res := count(a, b, mid)

if res >= int(k) {

hi = mid

} else {

lo = mid + 1

}

}

return lo

**Step 2. Mixed numbers**

So far so good. The problem is with the negative numbers. When the lists contain negative numbers, then their products form a kind of criss-cross pattern. Positive numbers are the result of combining numbers diagonally across in the matrix, and negative numbers are the result of combining numbers vertically:

\ | - | + |

- | + | - |

+ | - | + |

In order to search numbers, input lists are split into two halves:

a = [ -5, -2, -1, 0, 1, 2 ]

|----[a1]---|---[a2]--]

b = [ -8, -5, -2, 3, 5 ]

|----[b1]---|-[b2]-|

The first thing to note is that the number of positive / negative numbers can be counted by multiplying the lengths of each section:

negCount := len(a1)\*len(b2) + len(b1)\*len(a2)

Knowing how many numbers exist, if a request is made for k <= negCount, then that product must be in one of the negative quadrants, and vice versa for k > negCount. This tells us which sections to combine when searching for an answer.

However, simply multiplying elements in order from different segments yield different signs and orderings:

|  |  |  |  |
| --- | --- | --- | --- |
| first | second | sign | order |
| a1 | b1 | + | desc |
| a1 | b2 | - | no order |
| a2 | b1 | - | no order |
| a2 | b2 | + | asc |

The first trick here is to reverse negative portions, which results in ordered outputs:

|  |  |  |  |
| --- | --- | --- | --- |
| first | second | sign | order |
| rev(a1) | rev(b1) | + | asc |
| rev(a1) | b2 | - | desc |
| a2 | rev(b1) | - | desc |
| a2 | b2 | + | asc |

At this point it would be possible to write two different binary search functions, one for searching from the right, and one from the left. But there is one final trick to make the code cleaner.

By inverting negative values (making them positive), it's possible to search in the same way for negative and positive numbers. However, the k'th result will no longer be the k'th smallest, but rather the k'th largest number:

|  |  |  |  |
| --- | --- | --- | --- |
| first | second | sign | order |
| inv(rev(a1)) | inv(rev(b1)) | + | asc |
| inv(rev(a1)) | b2 | + | desc |
| a2 | inv(rev(b1)) | + | desc |
| a2 | b2 | + | asc |

**Final solution:**

func kthSmallestProduct(nums1 []int, nums2 []int, k int64) int64 {

    aSplit := sort.SearchInts(nums1, 0)

    bSplit := sort.SearchInts(nums2, 0)

    a1, a2 := rev(inv(nums1[:aSplit])), nums1[aSplit:]

    b1, b2 := rev(inv(nums2[:bSplit])), nums2[bSplit:]

    negCount := len(a1)\*len(b2) + len(b1)\*len(a2)

    sign := 1

    if int(k) > negCount {

        k -= int64(negCount) // => kth smallest number from non-negative part

    } else {

        k = int64(negCount) - k + 1 // => kth largest number after transform

        b1, b2 = b2, b1             // swap to fix combination

        sign = -1

    }

    var lo, hi int = 0, 1e10

    for lo < hi {

        mid := (lo + hi) / 2

        if rcount(a1, b1, mid) + count(a2, b2, mid) >= int(k) {

            hi = mid

        } else {

            lo = mid + 1

        }

    }

    return int64(sign \* lo)

}

func count(a, b []int, val int) int {

    var count int

    j := len(b) - 1

    for i := range a {

        for j >= 0 && a[i]\*b[j] > val {

            j--

        }

        count += j + 1

    }

    return count

}

func inv(a []int) []int {

    for i := range a {

        a[i] = -a[i]

    }

    return a

}

func rev(a []int) []int {

    for l, r := 0, len(a)-1; l < r; l, r = l+1, r-1 {

        a[l], a[r] = a[r], a[l]

    }

    return a

}