Implementing algorithms

Tutorial

Code folder: max_product_python.zip

Steps

1. Understand the problem, play with toy examples
2. Formalize the problem: input → desired output
3. Sketch possible solution in pseudocode/block/text
4. Translate an idea into a particular language taking into account language constraints
5. Test your implementation

Step 1. Understand the problem

Find the maximum product of two distinct numbers drawn from a sequence of non-negative integers.

Step 1. Understand the problem

Find the maximum product of two distinct numbers drawn from a sequence of non-negative integers.

My understanding:

Given: A sequence of non-negative integers (each number is either 0 or positive).

Need to find: The maximum value that can be obtained by multiplying two different elements from the sequence.

Step 1. Understand the problem

My understanding:

Given: A sequence of non-negative integers (each number is either 0 or positive).

Need to find: The maximum value that can be obtained by multiplying **two different elements** from the sequence.

Ask and clarify!

What do you mean by different elements?

The numbers are not necessarily distinct - but they are **at different positions** in the sequence

Step 1. Understand the problem

Given: A sequence of non-negative integers (each number is either 0 or positive).

Need to find: The maximum value that can be obtained by multiplying two different elements from the sequence.

Sample input:								
7	5	14	2	8	8	10	1	2
Sample output: 140								

Sample input:					
7	5	8	8	1	3
Sample output: 64 and not 56					

Step 2. Formalize the problem

Maximum pairwise product problem

Input: a sequence of n integers $a_1, \ldots, a_n \mid a_i \geq 0$, $\forall i$ in $[1 \ldots n]$

Output: max $(a_i * a_j)$, $1 \leq i \neq j \leq n$

Step 3. Sketch solution

Maximum pairwise product problem

Input: a sequence of n integers $a_1, \ldots, a_n$ | $a_i \geq 0$, $\forall i$ in $[1 \ldots n]$

Output: max $(a_i * a_j)$, $1 \leq i \neq j \leq n$

The naive solution is in the problem definition:

we need to check all pairs of integers in a sequence and find which pair produces the largest product

Step 3. Sketch solution

```
Algorithm max_pairwise_product_naive(A[1 . . . n]):
    product ← 0
    for i from 1 to n:
      for j from 1 to n:
         if i≠j:
            if product < A[i] · A[j]:
               product ← A[i] · A[j]
    return product
```

Step 3. Sketch solution

Algorithm max_pairwise_product_naive($A[1 \ldots n]$):

```
product ← 0
for i from 1 to n:
  for j from i + 1 to n:
    product ← max(product, A[i] · A[j])
return product
```

```
#define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))
```

Step 4. Implement solution

Language constraints:

Python:

We can find the size of list A using *len*(A)

C:

There is no way of finding the length of array A

(pointer decay)

Zero-based arrays/lists:

First position is 0, last position is n-1

Positive integer constraints:

Number of elements in an array: $2 \le n \le 2*10^9$

Values of elements: $0 \le a_1, . . . , a_n \le \sqrt{(2 \cdot 10^9)}=4.4*10^4$

so the product does not exceed the largest positive int

max_product_naive.py

Step 5. Test

Test implementation:

lst = [5, 6, 2, 7, 4] → 42

lst = [1,2] → 2

lst = [2,1] → 2

max_product_naive.py

Step ... Think!

Algorithm max_pairwise_product_naive($A[1\ldots n]$):

$product \leftarrow 0$
for i from 1 to n:
  for j from $i + 1$ to n:
    $product \leftarrow \max(product, A[i] \cdot A[j])$
return $product$

How many steps in total?

Step ... Think!

Algorithm max_pairwise_product_naive($A[1 \ldots n]$):

$product \leftarrow 0$
for i from 1 to n:
  for j from $i + 1$ to n:
    $product \leftarrow \max(product, A[i] \cdot A[j])$
return $product$

How many steps in total?
This is an $O(n^2)$ algorithm
For max input size $2*10^9$ it will perform $4*10^{18}$ steps!

Can we do better?

Step … Think!

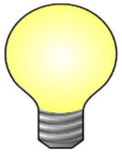
Sample input:				
5	6	2	7	4
Sample output: ?				

Do you see a faster solution?

Step … Think!

Sample input:				
5	6	2	7	4
Sample output: ?				

Do you see a faster solution?



Step 3A. Sketch faster solution

Algorithm max_pairwise_product_fast($A[1 \ldots n]$):

$index_1 \leftarrow 1$
for i from 2 to n:
  if $A[i] > A[index_1]$:
    $index_1 \leftarrow i$
$index_2 \leftarrow 1$
for i from 2 to n:
  if $i \neq index_1$ and $A[i] > A[index_2]$:
    $index_2 \leftarrow i$
return $A[index_1] \cdot A[index_2]$

In total about $2n$ steps: $O(n)$ algorithm!

Step 4A. Implement faster solution

Algorithm max_pairwise_product_fast($A[1 \ldots n]$):

```
index₁ ← 1
for i from 2 to n:
   if A[i] > A[index₁]:
      index₁ ← i
index₂ ← 1
for i from 2 to n:
   if i≠index₁ and A[i] > A[index₂]:
      index₂ ← i
return A[index₁] · A[index₂]
```

max_product_fast1.py

Step 5A. Test

Test implementation:

lst = [5, 6, 2, 7, 4] → 42

lst = [1,2] → 2

lst = [2,1] → 2 (outputs 4!!!!!)

Look at the code to find a bug or debug

Real correctness test: stress test

Algorithm stress_test(N, M):

```
while true:
    n ← random integer between 2 and N
    allocate array A[1 . . . n]
    for i from 1 to n:
        A[i] ← random integer between 0 and M
    print(A[1 . . . n])

    result_1 ← max_pairwise_product_naive(A)
    result_2 ← max_pairwise_product_fast(A)
    if result_1 = result_2:
        print("OK")
    else:
        print("Wrong answer: ", result_1, result_2)
        return
```

stress_test.py

Correct algorithm

Algorithm max_pairwise_product_fast($A[1\ldots n]$):

```
index ← 1
for i from 2 to n:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n]

index ← 1
for i from 2 to n − 1:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n − 1]

return A[n − 1] · A[n]
```

```
#define SWAP(a,b) {int temp; temp=a; a=b; b=temp;}
```

Correct algorithm: implementation

Algorithm max_pairwise_product_fast($A[1 \ldots n]$):

$index \leftarrow 1$
for i from 2 to n:
  if $A[i] > A[index]$:
    $index \leftarrow i$
swap $A[index]$ and $A[n]$

$index \leftarrow 1$
for i from 2 to $n - 1$:
  if $A[i] > A[index]$:
    $index \leftarrow i$
swap $A[index]$ and $A[n - 1]$

return $A[n - 1] \cdot A[n]$

max_product_fast.py

Implementing algorithms

1. Understand the problem, play with toy examples
2. Formalize the problem: input → desired output
3. Sketch a naive solution in pseudocode
4. Implement naive solution
5. Improve your solution
6. Test your improved solution using stress test until all the bugs are fixed

Measuring real-life performance

- How do we compare algorithms which belong to the same big-Oh class?
- Some of them may contain a very large constant: but we already got rid of all constants in our analysis
- Some of the algorithms may use a faulty data structure:

  an example would be an ancient version of the Sieve of Eratosthenes, where we removed an element from the middle of the array (expensive operation)
- The implementation quality and the programming language also matter:

  good implementation can make an algorithm run for up to 1000 times faster for the same input
- For these reasons, we run comparative performance tests

performance_test.py