Ex1

May 27, 2024

1 Trabalho Prático 4

André Freitas PG54707

Bruna Macieira PG54467

1.1 Exercício 1

Implemente um protótipo do esquema descrito no "draft" FIPS 204 que deriva do algoritmo Dilithium.

[ ]:
```
import hashlib
import sys
import time
from sage.all import *
import os
from pickle import load, dumps
from math import *
import json
from sage.modules.free_module import FreeModule
```

[ ]:
```
class dilithium:
    def __init__(self, nivel):
        self.q = 8380417 #2^23 - 2^13 + 1
        self.d = 13

        if nivel == 2:
            self.weigth = 39
            self.entropy = 192
            self.y1 = 2^17# 2**17
            self.y2 = (self.q-1)/88# (self.q-1)//88
            self.k = 4
            self.l = 4
            self.n = 2
            self.b = 78
            self.w = 80
            self.repetitions = 4.25
        elif nivel == 3:
```

```
            self.weigth = 49
            self.entropy = 225
            self.y1 = 2^19# 2**19
            self.y2 = (self.q-1)/32# (self.q-1)//32
            self.k = 6
            self.l = 5
            self.n = 4
            self.b = 196
            self.w = 55
            self.repetitions = 5.1
        else:
            self.weigth = 60
            self.entropy = 257
            self.y1 = 2^19# 2**19
            self.y2 = (self.q-1)/32# (self.q-1)//32
            self.k = 8
            self.l = 7
            self.n = 2
            self.b = 120
            self.w = 75
            self.repetitions = 3.85

        Z.<x> = ZZ[]
        R.<x> = QuotientRing(Z,Z.ideal(x^self.n+1))

        self.R = R

        Zq.<x> = GF(self.q)[]
        fi = x^self.n + 1
        Rq.<x> = QuotientRing(Zq,Zq.ideal(fi))
        self.Rq = Rq

    def H(self, zeta, tam):
        m = hashlib.shake_256()
        m.update(zeta)
        return m.digest(int(tam))

    def H_128(self, coef, tam):
        m = hashlib.shake_128()
        m.update(coef)
        return m.digest(int(tam))

    def highestbittozero(self, h128):
        ret = []
        array = bytearray(h128)
        # ate 255 pois se fosse 256 nao iria encontrar o array[i] nem array[i+2]
        for i in range(0, len(array)-1, 3):
```

```
            transform = array[i+2]
            if transform >= 128:
                transform -= 128
            byte = bytearray([array[i], array[i+1], transform])
            ret.append(int.from_bytes(byte, "little"))
        return ret

    def ExpandA(self, p):
        #    {0, 1}^256
        #  maps a uniform seed    {0, 1}^256 to a matrix A   (Rq)^ k x l
        A = []
        # It computes each polynomial a(i,j)   Rq of Â separately
        for i in range(self.k):
            # criar os t elementos de acordo com o p
            line = []
            for j in range(self.l):
                # 0   256 * i + j < 2^16 in little-endian byte order into␣
↪SHAKE-128
                twobytes = int((256 * i) + j).to_bytes(2, "little")
                # For the coefficient a(i,j) it absorbs the 32 bytes of p
                # immediately followed by two bytes 0   256 * i + j < 2^16
                hash128 = self.H_128(p + twobytes, 256)
                # setting the highest bit of every third byte to zero and␣
↪interpreting
                # blocks of 3 consecutive bytes in little endian byte order
                # â(i,j)   Rq
                line.append(self.Rq(self.highestbittozero(hash128)))
            # faz append do vetor criado em cima
            A.append(line)
        return Matrix(A)

    def number_to_bits_array(self, number):
        # Convert the number to binary representation
        binary_string = bin(number)[2:]
        # Create an array of bits
        bits_array = [int(bit) for bit in binary_string]
        return bits_array

    def bits_to_number(self, bits):
        bits = bits[::-1]
        ret = 0
        for i in range(len(bits)):
            if bits[i] == 1:
                ret += 2**i
        return ret

    def lowerupper(self, num):
```

```
        # 1    1  1  1   1 1 1 1
        # 128 64 32 16   8 4 2 1
        while num > 256:
            num -= 256
        array = self.number_to_bits_array(num)
        array = array[::-1]
        l = array[:4][::-1]
        u = array[4:][::-1]
        upper = self.bits_to_number(u)
        lower = self.bits_to_number(l)
        return lower, upper

    def sequencepositive(self, h):
        coef = []
        array = bytearray(h)
        for i in range(0, len(array)-1):
            lowerbits, upperbits = self.loweruppper(array[i])
            if self.n == 2:
                if lowerbits < 15:
                    lowerbits = lowerbits % 5
                else :
                    while lowerbits > 15:
                        lowerbits -= 15
                    lowerbits = lowerbits % 5
                if upperbits < 15:
                    upperbits = upperbits % 5
                else :
                    while upperbits > 15:
                        upperbits -= 15
                    upperbits = upperbits % 5
            else:
                if lowerbits < self.n + 1:
                    pass
                else :
                    while lowerbits > self.n + 1:
                        lowerbits -= self.n
                if upperbits < self.n + 1:
                    pass
                else :
                    while upperbits > self.n + 1:
                        upperbits -= self.n
            coef.append(lowerbits)
            coef.append(upperbits)
        return coef

    def ExpandS(self, p):
```

```
        # The function ExpandS, used for generating the secret vectors in key␣
↪generation,
        # maps a seed p0 to (s1, s2)   (S )^l × (S )^k
        s1 = []
        for i in range(self.l):
            # 2 bytes representing i in little endian byte order into SHAKE-256
            twobytes = int((256 * i)).to_bytes(2, "little")
            # it absorbs the 64 bytes of  0 concatenated with 2 bytes␣
↪representing i
            hash256 = self.H(p + twobytes, 256)
            #
            s1.append(self.R(self.sequencepositive(hash256)))
        s2 = []
        for i in range(self.k):
            new_i = i + self.l
            # 2 bytes representing i in little endian byte order into SHAKE-256
            twobytes = int((256 * new_i)).to_bytes(2, "little")
            # it absorbs the 64 bytes of  0 concatenated with 2 bytes␣
↪representing i
            hash256 = self.H(p + twobytes, 256)
            #
            s2.append(self.R(self.sequencepositive(hash256)))
        s1 = vector(s1)
        s2 = vector(s2)
        return s1, s2

    def modmais(self, r, q):
        ret = []
        for elem in r:
            arr = []
            for i in elem:
                arr.append(mod(i, q))
            ret.append(self.R(arr))
        return vector(ret)

    def modmaismenos(self, r, a):
        # Check if r is of type FreeModuleElement_generic_dense
        if isinstance(r, sage.modules.free_module_element.
↪FreeModuleElement_generic_dense):
            # Convert r to a list
            r = list(r)
        else:
            r = [int(r)]
        a = int(a)
        ret = []
        for elem in r:
            arr = []
```

```
            for i in elem:
                if a % 2 == 1:
                    a = a-1
                r0 = mod(i, a)
                r0 = mod(r0, a)
                # -/2 < r0   /2 sendo a é par
                if a % 2 == 0:
                    if not ( -int(a/2) < int(r0) <= int(a/2) ):
                        n = abs(int(r0) // a) + 1
                        if r0 < a:
                            n = 2  # or some other factor
                        if r0 > int((a-1)/2):
                            r0 -= a * n
                        else: r0 += a * n
                # -( -1)/2   r0   ( -1)/2 sendo a é impar
                if a % 2 == 1:
                    if not ( -int((a-1)/2) <= int(r0) and int(r0) <= int((a-1)/
↪2) ):
                        n = abs(int(r0) // a) + 1
                        if r0 < a:
                            n = 2  # or some other factor
                        if r0 > int((a-1)/2):
                            r0 -= a * n
                        else: r0 += a * n
                arr.append(r0)
            ret.append(self.R(arr))
        return vector(ret)

    def Power2Round(self, r, d):
        # r := r mod q
        r = self.modmais(r, self.q)
        #r = self.modmais(r, self.q)
        #r0 := r mod 2^d
        r0 = self.modmaismenos(r, 2^d)
        #r0 = self.modmaismenos(r, 2^d)
        #return ((r - r0)/2^d, r0)
        return ((r - r0) / 2^d), r0

    def genChaves(self):
         #   {0, 1}^256
        zeta = os.urandom(32)
        # ( ,  ' , K)   {0, 1}^256 × {0, 1}^512 × {0, 1}^256 := H( )
        h = self.H(zeta, 128)
        p = h[:32]
        p0 = h[32:96]
        K = h[96:]
        # A   (Rq)^k×l := ExpandA( )
```

```
        A = self.ExpandA(p)
        # (s1, s2)   (S )^l × (S )^k := ExpandS( 0)
        s1, s2 = self.ExpandS(p0)
        # t := As1 + s2
        t = A*s1 + s2
        # (t1, t0) := Power2Roundq(t, d)
        t1, t0 = self.Power2Round(t,self.d)
        # tr   {0, 1}^256 := H(  || t1)
        tr = self.H(p + dumps(t1), 32)
        # pk =  ( , t1)
        pk = {'p': p, 't1': t1}
        # sk = ( , K, tr, s1, s2, t0)
        sk = {'p': p, 'K': K, 'tr': tr, 's1': s1, 's2': s2, 't0': t0}
        return pk, sk

    def positivenumber(self, h):
        coef = []
        array = bytearray(h)
        for i in range(self.n):
            #print(self.y1 - array[i])
            coef.append(self.y1 - array[i])
        return coef

    def ExpandMask(self, p0, k):
        # The function ExpandMask, used for deterministically generating the␣
↪randomness
        # of the signature scheme, maps a seed  0 and a nonce k to y   (S y1)^l
        y = []
        # It computes each of the l coefficients of y, which are polynomials in␣
↪S( 1), independently.
        for i in range(self.l):
            # 2 bytes representing   + i in little endian byte order
            twobytes = int(k + i).to_bytes(3, "little")
            # For the i-th polynomial, 0   i < l, it absorbs the 64 bytes of  0␣
↪concatenated
            # with the 2 bytes representing   + i in little endian byte order␣
↪into SHAKE-256
            hash256 = self.H(p0 + twobytes, 256)
            # Then the output bytes are used to create a positive number in the␣
↪range
            # {0, ... , (2* 1) - 1}
            y.append(self.R(self.positivenumber(hash256)))
        return vector(y)

    def HighBits(self, r, a):
        # (r1, r0) := Decompose(r, a)
        r1, _ = self.Decompose(r, a)
```

```
        # return r0
        return r1

    def LowBits(self, r, a):
        # (r1, r0) := Decompose(r, a)
        _, r0 = self.Decompose(r, a)
        # return r0
        return r0

    def SampleInBall(self, p):
        # Initialize c = c0c1 . . . c255 = 00 . . . 0
        c = [0 for i in range(256)] #256x
        h = self.H(p,256)
        array = bytearray(h)
        digest = int.from_bytes(h[:8], 'little')
        bitssign = [int(digit) for digit in list(ZZ(digest).binary())][:self.
↪weigth]
        # for i := 256 -   to 255
        ind = 0
        for i in range(256 - self.weigth, 255):
            # j ← {0, 1, . . . , i}
            j = array[i]
            # s ← {0, 1}
            # ERRO
            if len(bitssign) > ind:
                s = bitssign[ind]
            else:
                s = 0
            ind += 1
            # ci := cj
            c[i] = c[j]
            # cj := (-1)
            c[j] = (-1)^s # será ou 1 ou -1
        return self.R(c)

    def MakeHint(self, z, r, a):
        # r1 := HighBits(r, a)
        r1 = self.HighBits(r, a)
        #
        v1 = self.HighBits(r + z, a)
        vec = []
        for r1i, v1i in zip(r1, v1):
            for r1ij, v1ij in zip(r1i, v1i):
                vec.append(r1ij != v1ij)
        # return [r1 != v1]
        return vec
```

```
    def hammingweigth(self, a):
        soma = 0
        for elem in a:
            if elem == 1:
                soma+=1
        return soma

    def sizeelems(self, w):
        ret = []
        for elem in w:
            array = []
            try:
                iter(elem)
            except TypeError:
                # If elem is not iterable, make it a list so we can iterate␣
↪over it
                elem = [elem]
            for i in elem:
                rangee = self.q
                r0 = int(mod(i, self.q))
                # -/2 < r0   /2
                if self.q % 2 == 0:
                    lower_bound = -int(rangee/2)
                    upper_bound = int(rangee/2)
                    while not (lower_bound <= r0 and r0 <= upper_bound):
                        r0 -= self.q
                # -( -1)/2   r0   ( -1)/2
                if self.q % 2 == 1:
                    lower_bound = -int((rangee-1)/2)
                    upper_bound = int((rangee-1)/2)
                    while not (lower_bound <= r0 and r0 <= upper_bound):
                        if r0 < lower_bound:
                            r0 += self.q
                        else:
                            r0 -= self.q
                array.append(r0)
            ret.append(max(array))
        return max(ret)

    def signMessage(self, sk, mensagem):
        # A   (Rq)^k x l := ExpandA(p)
        A = self.ExpandA(sk['p'])
        # μ   {0, 1}^512 := H(tr || M)
        # the first 64 output bytes are used as the resulting hash -> pagina 20
        u = self.H(sk['tr'] + dumps(mensagem), 64)
        #   := 0, (z, h) :=
        k = 0
```

```
        z = None #[]
        h = None #[]
        # 0   {0, 1} 512 := H(K || μ)
        # the first 64 output bytes are used as the resulting hash -> pagina 20
        p0 = self.H(dumps(self.k) + u, 64)
        # or  0 ← {0, 1}^512
        # p0 = os.urandom(32)

        c0 = None

        # while (z, h) =   do
        while z == None and h == None:
            # y   S^` 1 := ExpandMask( 0 ,  )
            y = self.ExpandMask(p0, k)
            # w := A * y
            w = A * y
            # w1 := HighBits(w, 2* 2)
            w1 = self.HighBits(w, 2*self.y2)
            # c0   {0, 1}^256 := H(μ || w1)
            c0 = self.H(u + dumps(w1), 64)
            # c   B  := SampleInBall(c0)
            c = self.SampleInBall(c0)
            # z := y + cs1
            z = min(y + c * sk['s1'])
            # r0 := LowBitsq(w - cs2, 2 2)
            r0 = min(self.LowBits(w-c * sk['s2'], 2*self.y2))
            # if kzkω   1 -   or kr0kω   2 - ,

            if (self.sizeelems(z) >= int(self.y1 - self.b)) or (self.
↪sizeelems(r0) >= int(self.y2 - self.b)):
                # then (z, h) :=
                z = None #[]
                h = None #[]
            else:
                # h := MakeHintq(-ct0, w - cs2 + ct0, 2 2)
                h = self.MakeHint((-c) * sk['t0'], w - (c * sk['s2']) + (c *␣
↪sk['t0']), 2 * self.y2)
                # if kct0kω   2 or the # of 1's in h is greater than  , then␣
↪(z, h) :=
                if ((self.sizeelems(c * sk['t0'])) >= int(self.y2)) or (h.
↪count(True) > self.w):
                    z = None #[]
                    h = None #[]
            #   :=   + l
            k = k + self.l
        #   = (c0, z, h)
        o = {'c0': c0, 'z': z, 'h': h}
```

```
        return o

    def Decompose(self, r, a):
        # r := r mod q
        r = self.modmais(r, self.q) #r mod self.q
        # r0 := r mod a
        r0 = self.modmaismenos(r, a)
        # if r - r0 == q - 1
        if r-r0 == self.q-1:
            # then r1 := 0; r0 := r0 - 1
            r1 = 0
            r0 = r0 - 1
        #  else r1 := (r - r0)/a
        else:
            r1 = (r - r0) / a
        # return (r1, r0)
        return r1, r0

    def UseHint(self, h, r, a):
        # m := (q - 1)/a
        m = (self.q - 1) / a
        # (r1, r0) := Decompose(r, a)
        r1, r0 = self.Decompose(r, a)
        # if h = 1 and r0 > 0 return (r1 + 1) mod+ m
        if (h == 1) and (r0 > 0):
            return (self.modmais(r1 + 1, m))
        # if h = 1 and r0   0 return (r1 - 1) mod+ m
        if (h == 1) and (r0 <= 0):
            return (self.modmais(r1 - 1, m))
        # return r1
        return r1

    @staticmethod
    def polynomial_matrix_to_number_matrix(matrix_of_polynomials):
        number_matrix = []
        for row in matrix_of_polynomials:
            number_row = []
            for p in row:
                lifted_p = p.lift()
                coefficients = lifted_p.coefficients(sparse=False)
                if coefficients:
                    # Only keep the coefficient of the highest degree term
                    number_coefficient = coefficients[-1].n()
                else:
                    # The polynomial is a constant
                    number_coefficient = lifted_p.n()
                number_row.append(number_coefficient)
```

```
            number_matrix.append(number_row)

        return matrix(number_matrix)

    def verifySign(self, pk, sign, mensagem):
        # Convert the PolynomialQuotientRing_domain_with_category.element_class␣
↪object to a polynomial over the integers
        z_as_poly = sign['z'].lift()

        # Convert the polynomial to a list of coefficients
        sign['z'] = list(z_as_poly.coefficients(sparse=False))

        A = self.ExpandA(pk['p'])

        # Get the base ring of the matrix A
        ring = A.base_ring()

        # Check the length of sign['z']
        if len(sign['z']) != A.ncols():
            # If sign['z'] does not have the same number of elements as columns␣
↪of A, resize it
            sign['z'] = sign['z'] + [0] * (A.ncols() - len(sign['z']))

        # Now create the vector z
        z = vector(sign['z'])

        z = z.change_ring(ring)

        # μ   {0, 1}^512 := H(H(  || t1) || M)
        u = self.H(self.H(pk['p'] + dumps(pk['t1']), 64) + dumps(mensagem), 64)

        # c := SampleInBall(c0)
        c = self.SampleInBall(sign['c0'])

        c_pk_t1 = matrix([list(c * pk['t1'])] * 4)
        c_pk_t1_d = (c_pk_t1 * 2**self.d)

        # Get the number of rows and columns of c_pk_t1_d
        num_rows2 = c_pk_t1_d.nrows()
        num_cols2 = c_pk_t1_d.ncols()

        # Ensure A*z and c_pk_t1_d are in the same ring
        Az = A * z

        # Get the parent of the first element of Az
        parent1 = Az[0].parent()
```

```
        # Convert Az to a matrix
        Az_matrix = sage.matrix.matrix_space.MatrixSpace(parent1, len(Az),␣
↪1)(Az)

        # Now you can get the number of columns
        num_cols1 = Az_matrix.ncols()
        num_rows1 = Az_matrix.nrows()

        # Convert Az_matrix to a Matrix_generic_dense object
        Az_matrix_dense = sage.matrix.matrix_space.MatrixSpace(parent1,␣
↪num_rows1, 1)(Az)

        # Assuming Az_matrix is a 4x1 matrix and c_pk_t1_d is a 4x4 matrix
        Az_matrix_expanded = matrix(Az_matrix_dense.base_ring(), num_rows2,␣
↪num_cols2)
        for i in range(Az_matrix_expanded.nrows()):
            for j in range(Az_matrix_expanded.ncols()):
                Az_matrix_expanded[i, j] = Az_matrix[i, 0]

        # Convert the coefficients of the polynomials in Az_matrix_expanded to␣
↪numbers
        Az_number_matrix = self.
↪polynomial_matrix_to_number_matrix(Az_matrix_expanded)

        # Get the parent of the first element of the first row of c_pk_t1_d
        parent2 = c_pk_t1_d[0, 0].parent()

        # Create c_pk_t1_d_matrix over the parent ring of c_pk_t1_d
        c_pk_t1_d_matrix = sage.matrix.matrix_space.MatrixSpace(parent2,␣
↪num_rows2, num_cols2)(c_pk_t1_d)

        # Now do the same for c_pk_t1_d_matrix
        c_pk_t1_d_number_matrix = self.
↪polynomial_matrix_to_number_matrix(c_pk_t1_d_matrix)

        # Now you can perform the operation
        result = Az_number_matrix - c_pk_t1_d_number_matrix

        # w1 := UseHint(self.h, A*z - ct1 · 2^d, 2 2)
        w1 = self.UseHint(sign['h'], result * 2**self.d, 2 * self.y2)

        # Verify conditions
        condition1 = self.sizeelems(sign['z']) < self.y1 - self.b
        condition2 = sign['c0'] == self.H(u + dumps(w1), 64)  # Ensure `H` is␣
↪called with the required size
```

```
        condition3 = sum(sign['h']) <= self.w  # Count the number of `True` values in `sign['h']`

        return condition1, condition2, condition3
```

[ ]:
```
# Exemplo de uso
dilithium = dilithium(nivel=5)
```

[ ]:
```
# Start the timer
start_time = time.time()

# Gerar chaves
chavepub, chavepriv = dilithium.genChaves()

# End the timer
end_time = time.time()

# Calculate the elapsed time in milliseconds
elapsed_time = (end_time - start_time) * 1000


# Calculate the size of the keys
size_chavepub = sys.getsizeof(chavepub)
size_chavepriv = sys.getsizeof(chavepriv)

print("Chave Pública:", chavepub)
print("Tamanho da Chave Pública:", size_chavepub, "bytes")
print("Chave Privada:", chavepriv)
print("Tamanho da Chave Privada:", size_chavepriv, "bytes")

print(f"Tempo para gerar as chaves: {elapsed_time:.3f} ms")
```

```
Chave Pública: {'p': b'\xc5a\x84\xa8\xc1\xbcCJ\x8d\xb3n\xf33}r2@\xb3\xca;\xa4\xd
95\x03B\xfe=)\xadvz\xbd', 't1': (1008*x + 838, 972*x + 445, 294*x + 109, 479*x +
669, 746*x + 247, 584*x + 250, 290*x + 282, 413*x + 907)}
Tamanho da Chave Pública: 232 bytes
Chave Privada: {'p': b'\xc5a\x84\xa8\xc1\xbcCJ\x8d\xb3n\xf33}r2@\xb3\xca;\xa4\xd
95\x03B\xfe=)\xadvz\xbd', 'K': b"\x18'C%0\xc3{\xf5R)\x91\xb7I+\xf9N\x89\x1b,\x0b
\xb8\xadA=\xd6\xa5\x98kf\x86\r\xb6", 'tr': b'\x7f \x8a\xcd2\xd4\xc6\xff\xa8\x03\
xec=u~28\xc9\xb7_\x1fg\x99g\xf9\xf8\x9cx(\x824\xaa\xc4', 's1': (-7*x + 55, -9*x
+ 28, -7*x + 17, 34*x - 9, -6*x - 5, 10*x + 16, 5*x + 10), 's2': (-35*x - 17,
-13*x - 18, 17*x - 23, 34*x + 10, -2*x + 32, -30*x - 22, -50*x - 1, -40*x + 6),
't0': (3394*x + 4809, 308*x + 4984, 2419*x + 5872, 6594*x + 1165, 7052*x + 7014,
6317*x + 7396, 5270*x + 5320, 198*x + 4448)}
Tamanho da Chave Privada: 360 bytes
Tempo para gerar as chaves: 50.935 ms
```

[ ]:
```
# Start the timer
start_time = time.time()

# Assinar mensagem
mensagem = b"Mensagem de exemplo"
assinatura = dilithium.signMessage(chavepriv, mensagem)

# End the timer
end_time = time.time()

# Calculate the elapsed time in milliseconds
elapsed_time = (end_time - start_time) * 1000

size_assinatura = sys.getsizeof(assinatura)

print("Assinatura:", assinatura)
print("Tamanho da Assinatura:", size_assinatura, "bytes")

print(f"Tempo para assinar a mensagem: {elapsed_time:.3f} ms")
```

```
Assinatura: {'c0': b"\x02\x08Ac\x93\x934>^\x0b\x99\x0cRh\\\xe6\xcb\x1c\x9a.\xe8M
\xa6\x05\x96\xb7\x86 aA\xa1}@w\xbc(d3QYd\x07t\xc2\xe5\xe9\x0b\xd4\x94\x9b!'\xf2\
xc8\xbd\x18\x8c\x80\x8c\x8bd$I\xe6", 'z': 523861*x + 523927, 'h': [False, False,
False, False, False, False, False, False, False, False, False, False, False,
False, False, False]}
Tamanho da Assinatura: 232 bytes
Tempo para assinar a mensagem: 61.655 ms
```

[ ]:
```
# Start the timer
start_time = time.time()

# Verificar assinatura
verificacao = dilithium.verifySign(chavepub, assinatura, mensagem)

# End the timer
end_time = time.time()

# Calculate the elapsed time in milliseconds
elapsed_time = (end_time - start_time) * 1000

print("Verificação da Assinatura:", verificacao)

print(f"Tempo para assinar a mensagem: {elapsed_time:.3f} ms")
```

```
Verificação da Assinatura: (True, False, True)
Tempo para assinar a mensagem: 15.317 ms
```