NAME

- `opcount.py` - count operations in a cryptographic scheme

SYNOPSIS

```
opcount.py [{preamble} ...] [-t {postamble} ...] < {filename} | gp -q
```

DESCRIPTION

The `opcount.py` is a filter program that counts group operations in a cryptographic scheme written in Python-like scheme description language (PSDL), and outputs some sequence of assignment statements of formulae for GP (PARI Calculator). It reads a PSDL code from the standard input, then writes the counting results to the standard output. If `{preamble} ...` are specified, it outputs the contents of the files `{preamble} ...` as it is in the specified order before the counting results. In the same way, if `-t {postamble} ...` are specified, it outputs the contents of the files after the counting results. It is preferable to prepare files of `preamble` and `postamble` so that the total output of the `opcount` constitutes a complete script for the GP calculator.

ALGEBRA OF THE OPERATION COSTS OF MULTI-BASE EXPONENTIATION

Consider the multi-base exponentiation $\prod_{i=0}^{n} g_i^{x_i}$ where $n \in \mathbb{N}_0$ and $x_0 \in \{0,1\}$. $x_0, \ldots, x_n$ are variables represent elements in a commutative ring $\mathbb{L}$ with a unit, and called exponents. $g_0, \ldots, g_n$ are variables represent elements in some group ($\mathbb{L}$-module), and called bases. When we design a cryptographic scheme, we do not need to care about the actual elements stored in the exponents or bases. Instead we assume that there exists such an element virtually (formal multi-base exponentiation).

For example, when $x_0 = 0$, we don't have to consider the element g_0, but we assume that there exists such an element virtually. The reason why x_0 is restricted in $\{0,1\}$ unlike the other x_i's is that this expression make it easy to handle both the multiplication and the exponentiation on $\mathbb{G}$.

If all bases are single variables of type $\mathbb{G}$, we write (n, x_0) to express the operation cost to calculate $\prod_{i=0}^{n} g_i^{x_i}$.

For example, the operation cost to calculate a variable g of type $\mathbb{G}$ without any exponentiation or multiplication can be written as $(0,1)$, because $g = \prod_{i=0}^{0} g_i^{x_i}$ where $g_0 = g$ and $x_0 = 1$. Of course,

$$(0,1) = 0.$$

To derive algebraic operations of costs of formal multi-base exponentiation, we consider a representation s.t. all multi-base exponentiation in an algebraic representation are replaced into corresponding operation costs.

For example, Let `mul` be the cost of multiplication of two variables of type $\mathbb{G}$. For $g, h, k \in \mathbb{G}$, we can consider the following correspondence.

actual operation	algebraic representation of cost	actual cost
$h \times k \to g$	$(0,1) \times (0,1) \to (0,1)$	`mul`.

Considering this correspondence, we can regard

$$(0,1) \times (0,1) = (0,1) + \texttt{mul}.$$

Similarly, we can consider such a correspondence for general multi-base exponentiation. That is

actual operation	algebraic representation of cost
$\left(h_0^{y_0} \prod_{i=1}^{m} h_i^{y_i}\right) \times \left(k_0^{z_0} \prod_{i=1}^{\ell} k_i^{z_i}\right) \to g_0^{x_0} \prod_{i=1}^{n} g_i^{x_i},$	$(m, y_0) \times (\ell, z_0) \to (n, x_0),$

where $n, m, \ell \in \mathbb{N}_0$, $x_0, y_0, z_0 \in \{0, 1\}$. Let

$$g_i = \begin{cases} h_0^{y_0} \times k_0^{z_0} & i = 0, \\ h_i & i \in \{1, \ldots, m\}, \\ k_{i-m} & i \in m + \{1, \ldots, \ell\}. \end{cases}$$

We can assume that

$$\begin{aligned} n &= m + \ell, \\ x_0 &= y_0 \vee z_0. \end{aligned}$$

When $y_0 \wedge z_0 = 1$, we need one multiplication of bases to calculate the new base $g_0 = h_0 \times k_0$. Therefore we can derive

$$(m, y_0) \times (\ell, z_0) = (m + \ell, y_0 \vee z_0) + (y_0 \wedge z_0) \cdot \texttt{mul}.$$

In the same manner, we can derive the cost of an exponentiation of multi-base exponentiation.

actual operation	algebraic representation of cost
$\left(h_0^{y_0} \prod_{i=1}^{m} h_i^{y_i} \right)^z \to g_0^{x_0} \prod_{i=1}^{n} g_i^{x_i},$	$(m, y_0)^z \to (n, x_0),$

where, $n, m \in \mathbb{N}_0$, $x_0, y_0 \in \{0, 1\}$. Let

$$g_i = \begin{cases} h_i & y_0 = 0,\ i \in \{0, \ldots, m\}, \\ h_{i-1} & y_0 = 1,\ i \in \{0, \ldots, m+1\}, \end{cases}$$

where h_{-1} is an appropriate variable. we can assume that

$$\begin{aligned} n &= m + y_0, \\ x_0 &= 0 \end{aligned}$$

That is

$$(m, y_0)^z = (m + y_0, 0).$$

`opcount.py` counts this costs of multi-base exponentiation, then outputs them.

EXECUTION ENVIRONMENT

First of all, prepare the follows.

- UNIX-like command line environment
- Python Interpreter
- SymPy Python library for symbolic mathematics
- SageMath mathematics software system
- PARI/GP computer algebra system
- complete set of source code of `opcount.py`

`opcount.py` is confirmed to run correctly under the following environments with an appropriate configuration.

Environment	Version	Architecture	Note
Ubuntu	18.04.2 LTS	amd64	SageMath 8.1/Python 2.7/SymPy 1.1.1

OPTIONS

- `-t` outputs the contents of the files specified after this option as a postamble.

PYTHON-LIKE SCHEME DESCRIPTION LANGUAGE

Python-like Scheme Description Language (PSDL) is a domain specific language to analyze data flow of pairing-based cryptographic schemes. Its syntax is almost the same as that of SageMath language.

The formal syntax of the language accepted by the actual command `opcount.py` is almost the same as that of Python 2.7. Therefore the language of `opcount.py` is different from PSDL based on the language of SageMath in the strict sense, because `opcount.py` considered `**` as exponential operator as in the python language. For a file `source.sage` written in the SageMath notation, if the command

```
sage --preparse source.sage
```

is executed, then it outputs a file `source.sage.py` written in the python notation. `opcount.py` is designed to input this `source.sage.py`.

```
./opcount.py < source.sage.py
```

In the following, we assume that for any souce files in PSDL, the command `sage --preparse` is applied to them. Due to convenience for SageMath system, we choose `.sage` as the filename extension of PSDL.

Comment of PSDL is the same as python language, starts with `#` and extending to the next newline character.

The most syntactically significant difference between PSDL and python is interpretation of the `^` operator.

In python language, the `^` operator represents the bitwise xor operation, and has lower precedence than any arithmetic operator. While in PSDL, it represents the exponent operation like TeX, has higher precedence than any binary arithmetic operator and right-associativity. Use `^^` for xor operator (`^^=` for xor assignment) instead.

In PSDL, a description of an algorithm must be placed inside a function such as

```
def main():
    ...
```

It is no matter that function named other than `main` is defined. Multiple functions can be defined if all functions have unique names in a file. For example:

```
def setup():
    ...
def sign():
    ...
def verify():
    ...
```

DATA TYPE

Other than the standard python data types, PSDL support the following data types.

Type name	Explanation
`Group`	source group $\mathbb{G}$ of pairing
`Group1`	source group $\mathbb{G}_1$ of asymmetric pairing
`Group2`	source group $\mathbb{G}_2$ of asymmetric pairing
`Target`	target group of pairing
`Integer`	integer ((commutative subring of) the endomorphisms of the above groups)

Since python language is a dynamically typed language, in principle, for any variables in python code, their type is not determined until they are refered in execution. Therefore the operation cost of the following encryption algorithm is not determined strictly until `Enc` is really called at runtime.

```
def Enc(m, g, y, r):
    c1 = g^r
    c2 = m * y^r
    return (c1,c2)
```

To avoid complication of analysis for such schemes, PSDL assumes that all arguments of functions have default values which indicate types of the arguments.

```
def Enc(m = Group(), g = Group(), y = Group(), r = Integer()):
    c1 = g^r
    c2 = m * y^r
    return (c1,c2)
```

(Unlike in CSDL) in PSDL, variables like `c1` or `c2` in the above function can be assigned without any declaration as in python or SageMath language. The types of variables are determined according to type inference algorithm in `opcount.py`, or output warnings when it fails to infer.

LIST VALUE AND LIST VARIABLE

Although there are very few syntactic differences between PSDL and python, list and tuple in python are simply identified and have no difference in PSDL. A list value is described as some elements enclosed in parentheses like

```
(a, b, c)
```

If all elements in a list value are left values, then the list becomes a left value, i.e. assignable.

```
integer a, b, c, d, e, f ;
(a,b,c) = (d^2, e^3*f, e*d/f) ;
```

List can be nested.

```
integer a, b, c, d, f ;
((a,b),c) = ((d^2, d^3*f), d/f) ;
```

A list value can be assigned to a variable of type `list` (list variable).

```
integer a, b, c ;
list L = (a, b, c) ;
```

FUNCTION

Functions must be defined as in the python language.

```
def Random():
    return Integer()

def setup():
  g   = Group()
  msk = Random()
  y   = g^msk
  return  ((g,y),msk)
```

The type of return value is inferred from `return` statement. In PSDL, each function is assumed to have unique return type. Functions who alters its return type in runtime like in python language are not supported in PSDL.

SPECIAL FORM

The following special forms are available. Notice that if there is a variable or a function of the same name of a special form, the form is not available in the scope. It frequently happens that a variable `e` is declared carelessly.

Special form	Assumed function
`e (g1, g2)`	pairing
`Repeat (n, g)`	repeat `g` `n` times
`product(n, t)`	multiply `t` `n` times
`inline("s")`	Output string literal `s` as it is

Some special forms are converted into some functions assumed to represent the number of some operations. In such a case, the corresponding functions must be defined in the pari/gp script.

CSDL Statement	Translation to pari/gp	Assumed function
`for x in y: ...`	`for_statement("x","y","z",{body})`	loop
`if x: ...`	`if_statement("x",{body},{orelse})`	branch

Regarding `for` and `if` statement: since it is not realistic to analyse average number of loops or probability of branches automatically when the code is complicated, `opcount` have an interface with which the user can set such number or probability to a preferred representation. `opcount` interprets the PSDL statement

```
for x in y:
    body
else:
    orelse
```

as an operation cost

```
for_statement("x","y",{body},{orelse})
```

and the PSDL statement

```
if x:
    body
else:
    orelse
```

as an operation cost

```
if_statement("x",{body},{orelse})
```

Here `"x"` and `"y"` are strings to represent normalized equations `x` and `y` respectively, and `{body}` and `{orelse}` are the operation costs of `body` and `orelse` clauses respectively. When `for` or `if` statement has no `else` clause, `{orelse}` becomes 0. When `if` statement has `elif` clauses, `if_statement()`'s are nested in pari/gp side. By defineing appropriate `for_statement()` or `if_statement()` function in pari/gp side, user can obtain proper representation of number of loops or probability of branches. `inline` function hands over the succeeding string literal to the pari/gp side as it is. For example, by inserting the following code to the head of CSDL file, pari/gp can handle relatively simple loops with `range` function or branches.

```
inline("""

  range(A_, B_ = 0, C_ = 1) = {
    return (floor((A_ - B_)/C_)) ;
  }

  for_statement(A_,B_,C_,D_) = {
    A_ = eval(B_) ;
    return (A_ * C_ + D_) ;
  }

  if_statement(A_,B_,C_) = {
    return (if_prob * B_ + (1 - if_prob) * C_) ;
  }

  ni = 10;
  if_prob = 1/2 ;

""")
```

INCOMPATIBILITY

Current implementation of `opcount.py` is incompatible with `opcount` in terms of the followings.

- different input language
- expansion of for statement
- `opcount.py` does not have `-v` and `-p` options.
- `opcount.py` ignores minor operations like assignment
- `opcount.py` does not evaluate functions whose name begin with `_`

EXAMPLES

EXAMPLE 1

Suppose that `hello_world.sage` is a text file which contains the following code.

```
hello = Group()
world = Integer()

def main():
    return hello^world
```

After processing `sage --preparse hello_world.sage`, by inputting `hello_world.sage.py` to `opcount.py`, the following output is obtained. (Note that the line followed by `$` represents the input of the command line.)

```
$ sage --preparse hello_world.sage
$ ./opcount.py < hello_world.sage.py
call_main = GROUP_batch(1,0);
```

When a function is defined in PSDL, `opcount.py` counts the operation cost of the function, then output a script to assign the value to the variable `call_`{function name}. In the above case, it outputs an assignment to `call_main`, since `main()` function is defined in the source. Here `GROUP_batch(`n`,`b`)` represents the operation cost of multi-base exponentiation of type `group`, i.e. (n, b). Although the computer algebra system PARI/GP cannot evaluate this script just as is, it can be handled with appropriate definitions in preamble e.g.

```
GROUP0_batch(n,b) = n * GROUP0_pow + (n-1+b) * GROUP0_mul ;
```

EXAMPLE 2

Suppose that `elgamal.sage` is a text file which contains the following code.

```
g = Group()

def Random():
    return Integer()

def Setup():
    x = Random()
    y = g^x
    return (x,y)

def Enc(y = Group(), m = Group()):
    r  = Random()
    return (g^r, m * y^r)

def Dec(x = Integer(), c = (Group(), Group())):
    (c1,c2) = c
    return c2/c1^x
```

Since PSDL (Python) has no concept of prototype declaration, to assign return types to functions, the functions must be defined. In this example, `random()` is defined as a function which returns `Integer`. By processing `elgamal.sage` with `sage` and `opcount.py`, the following output is obtained.

```
$ sage --preparse elgamal.sage
$ ./opcount.py < elgamal.sage.py
call_Random = 0;
call_Setup = GROUP_batch(1,0) + call_Random;
call_Enc = GROUP_batch(1,0) + GROUP_batch(1,1) + call_Random;
call_Dec = GROUP_batch(1,1);
```

If a function is called in the definition of other function, it is counted as a `call_{function name}`. For example, the above `random()` function is counted as a `call_Random`.

EXAMPLE 3

Suppose that `boneh_franklin_0.sage` is a text file which contains the following code.

```
def Random():
    return Integer()

def Hash2point(ID = str()):
    return Group()

g1 = Group()
g2 = Group()

def Setup():
    x = Random()
    y = g2^x
    return (x,y)

def Keygen(ID = str(), x = Integer()):
    return Hash2point(ID)^x

def Enc(m = Target(), ID = str(), y = Group() ):
    r  = Random()
    c1 = g1^r
    c2 = m * e(Hash2point(ID),y^r)
    return (c1,c2)

def Dec(c = (Group(), Target()), d_ID = Group() ):
    (c1,c2) = c
    m = c2/e(c1,d_ID)
    return m
```

In this example, a cryptographic scheme is defined by using a special form `e(·,·)` (pairing). By processing `boneh_franklin_0.sage` with `sage` and `opcount.py`, the following output is obtained. (Note that \\ at the end of a line represents continuation to the next line.)

```
$ sage --preparse boneh_franklin_0.sage
$ ./opcount.py < boneh_franklin.sage.py
call_Random = 0;
call_Hash2point = 0;
call_Setup = GROUP_batch(1,0) + call_Random;
call_Keygen = GROUP_batch(1,0) + call_Hash2point;
call_Enc = 2*GROUP_batch(1,0) + TARGET_mul + call_Hash2point\\
 + call_Random + pairing_batch(1);
call_Dec = TARGET_div + pairing_batch(1);
```

Pairing is counted as an independent operation cost from that of the target group. A product of n pairings is counted as a `pairing_batch(`n`)`.

EXAMPLE 4

For an appropriate PSDL code, if some appropriate `Group`, `Target`, ... classes are given, such a PSDL code actually runs with SageMath environment. Suppose that `boneh_franklin.sage` is a text file which contains the following code. In thie example, a complete set of pairing associated classes are defined by using an appropriate super singular elliptic curve. Thie code can be processed by `opcount.py`, and actually runs with SageMath. (Note that \ at the end of a line represents continuation to the next line.)

```
import sys
import hashlib

p =  873698582687056360681306206604874377075750222236547523131172084502950 6\
97829899007119845869346722117702060619846374890663598182852707400096370075 7\
8657541483
K.<w> = GF(p^2, modulus=(x^2 + x + 1))
E = EllipticCurve(K,[0,0,0,0,1])
q = 2^160+7
c = Integer((p+1)/q)
X =  405922632941434332374940794013822476281672937050232754560958147701335 3\
031563742839450870779104636956357047190671292786512429918070903148451478909 \
4287053963
Y =  474156375853016450932787939670729454239234802587718514943060866405868 1\
221491922377655373213917012675198371771889876320123071888318106364034531892 \
2577502350
P = E([X, Y])

class Group:
    def __init__(self, P=[X,Y,1]):
        if type(P) == Group:
            self.point = P.point
        else:
            self.point = E(P)

    def __mul__(self,Q):
        return Group(self.point + Q.point)

    def __div__(self,Q):
        return Group(self.point - Q.point)

    def __pow__(self,x):
        return Group(x * self.point)

    def __eq__(self,Q):
        return self.point.__eq__(Q.point)

    def __ne__(self,Q):
        return self.point.__ne__(Q.point)

    def __delitem__(self, key):
        self.point.__delitem__(key)

    def __getitem__(self, key):
        return self.point.__getitem__(key)

    def __setitem__(self, key, value):
```

```
        self.point.__setitem__(key, value)

    def __repr__(self):
        return str(self.point)

    def __str__(self):
        return str(self.point)

    def _distortion(self):
        return Group([self[0]*w, self[1], self[2]])

gT = Group().point.tate_pairing(Group()._distortion().point,q,2)

class Target:

    def __init__(self, value = gT):
        if type(value) == Target:
            self.value = value.value
        else:
            self.value = value

    def __mul__(self,y):
        return Target(self.value * y.value)

    def __div__(self,y):
        return Target(self.value / y.value)

    def __pow__(self,y):
        return Target(self.value ^ y)

    def __eq__(self,Q):
        return self.value.__eq__(Q.value)

    def __ne__(self,Q):
        return self.value.__ne__(Q.value)

    def __delitem__(self, key):
        self.value.__delitem__(key)

    def __getitem__(self, key):
        return self.value.__getitem__(key)

    def __setitem__(self, key, value):
        self.value.__setitem__(key, value)

    def __str__(self):
        return str(self.value)

    def __repr__(self):
        return str(self.value)

def e(P,Q):
    return Target(P.point.tate_pairing(Q._distortion().point,q,2))

def Random():
```

```
    return Integer(randrange(0,q-1))

def Hash2point(ID = str()):
    Y = Mod(Integer('0x' + hashlib.sha256(ID).hexdigest()),p)
    X = (Y^2 - 1)^((2*p-1)/3)
    P = E([X, Y])
    Q = c * P
    return Group(Q)

g1 = Group()
g2 = Group()

def Setup():
    x = Random()
    y = g2^x
    return [x,y]

def Keygen(ID = str(), x = Integer()):
    return Hash2point(ID)^x

def Enc(m = Target(), ID = str(), y = Group() ):
    r  = Random()
    c1 = g1^r
    c2 = m * e(Hash2point(ID),y^r)
    return [c1,c2]

def Dec(c = [Group(), Target()], d_ID = Group() ):
    c1 = c[0]
    c2 = c[1]
    m = c2/e(c1,d_ID)
    return m

m  = Target()^Random()
(x,y) = Setup()
d_ID = Keygen('Alice',x)
c = Enc(m,'Alice',y)
md = Dec(c,d_ID)
print(m == md)
```