名前

- opcount.py - 暗号方式中の演算をカウントする

書式

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

説明

opcount.py は ペアリングに基づく暗号方式記述言語 (Python-like scheme description language, PSDL) で記述された暗号方式中の演算をカウントし, GP (PARI Calculator) 形式の代入文の列を出力するフィルタプログラムである. PSDL を標準入力から読みカウント結果を標準出力に出力する. {preamble} ... が指定されたときは, 結果を標準出力に出力する前にファイル {preamble} ... の内容を順次そのまま標準出力に出力する. -t {postamble} ... が指定された ときは, 結果を標準出力に出力した後にファイル{postamble} ... の内容を 順次そのまま出力する. 出力全体が一つの GP のプログラムになるような preamble および postamble のファイルを準備することが望ましい.

多重冪乗の演算コストの代数

$n \in \mathbb{N}_0$ および $x_0 \in \{0,1\}$ なる $\prod_{i=0}^{n} g_i^{x_i}$ 型の 多重冪乗を考える. $x_0, \ldots, x_n$ は単位的可換環 $\mathbb{L}$ の元を表現する変数 で冪と呼ばれる. $g_0, \ldots, g_n$ はある群 ($\mathbb{L}$-加群)$\mathbb{G}$ に属する元を表現する変数で基底と呼ばれる. 暗号方式を設計する際は, 個々の冪や基底が実際にはどの元に対応するかを考える必要は無く, 仮想的にそのような元があると考える (形式的多重冪乗). 例えば, $x_0 = 0$ のときは g_0 という元をそも そも考える必要は無いが, 仮想的にそのような元があると考える. 他の x_i と異なり x_0 だけ $\{0,1\}$ 上に制限される理由は $\mathbb{G}$ 上の乗算と冪乗をまとめて考えるのに便利だからである. 基底が全て $\mathbb{G}$ 型の単一の変数である 時, $\prod_{i=0}^{n} g_i^{x_i}$ を計算するのに必要な演算コストを (n, x_0) と書くとする. 例えば何の冪乗も乗算も無い $\mathbb{G}$ 型変数 g を計算する 演算コストは $(0,1)$ と書くことが出来る. 勿論

$$(0,1) = 0$$

である. 複数の形式的多重冪乗の代数的な演算コストを考察するため, 代数的表現 上の全ての多重冪乗を該当する演算コストに置き換えた表現 (演算コストの代数的表現) を考える. 例えば, 二つの $\mathbb{G}$ 型変数の積を計算する演算コストを mul と定義すると $g, h, k \in \mathbb{G}$ に対して次のような対応を考える事が出来る.

実際の演算	演算コストの代数的表現	実際の演算コスト
$h \times k \to g$	$(0,1) \times (0,1) \to (0,1)$	mul.

これらの対応により,

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

と考える事ができる. このような演算コストの代数的表現を一般の多重冪乗で 考える. 即ち

$$\text{実際の演算}\qquad\qquad\text{演算コストの代数的表現}$$
$$\left(h_0^{y_0}\prod_{i=1}^{m}h_i^{y_i}\right)\times\left(k_0^{z_0}\prod_{i=1}^{\ell}k_i^{z_i}\right)\to g_0^{x_0}\prod_{i=1}^{n}g_i^{x_i},\qquad (m,y_0)\times(\ell,z_0)\to(n,x_0),$$

但し, $n,m,\ell\in\mathbb{N}_0$, $x_0,y_0,z_0\in\{0,1\}$, を考察する.

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

と定義すると

$$n=m+\ell,$$
$$x_0=y_0\vee z_0$$

として良い. $y_0\wedge z_0=1$ なる時は新しい基底 $g_0=h_0\times k_0$ を 計算する為に基底の乗算が一回必要となる. 従って

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

とする事が出来る. また, 多重冪乗の冪乗も同様にして考える.

$$\text{実際の演算}\qquad\qquad\text{演算コストの代数的表現}$$
$$\left(h_0^{y_0}\prod_{i=1}^{m}h_i^{y_i}\right)^{z}\to g_0^{x_0}\prod_{i=1}^{n}g_i^{x_i},\qquad (m,y_0)^z\to(n,x_0),$$

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

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

但し h_{-1} は適当な変数, と定義すると

$$n=m+y_0,$$
$$x_0=0$$

として良い. 即ち

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

とする事が出来る. `opcount.py` はこの多重冪乗の演算コストをカウントして出力する.

実行環境

以下を準備せよ.

- UNIX-ライクなコマンド実行環境
- Python インタプリタ
- SymPy Python の記号計算用ライブラリ
- SageMath 数学ソフトウェア
- PARI/GP 計算代数システム
- `opcount.py` のソース一式

`opcount.py` は適切に設定された次の環境下で正しく実行できる事が確認されている.

環境	バージョン	アーキテクチャ	Note
Ubuntu	18.04.2 LTS	amd64	SageMath 8.1/Python 2.7/SymPy 1.1.1

オプション

- `-t` 以降指定されたファイルを postamble として出力する.

暗号方式記述言語

暗号方式記述言語 (Python-like Scheme Description Language, PSDL) はペアリングに基づく暗号方式のデータフロー解析を目的としたドメイン特化言語で, その文法は SageMath と概ね同じである.

実際のコマンド `opcount.py` が受理する言語の形式文法は厳密には Python 2.7 言語とほぼ同一であり, `**` を冪乗演算子と見なす為, 厳密な意味では `opcount.py` が処理する言語は SageMath の記法に基づく PSDL とは異なる. TeX に由来すると思しき `^` を冪乗演算子として使用する SageMath の記法で記述された `source.sage` なるファイルに対して コマンドラインより

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

という命令を与えると `source.sage.py` なるファイルが出力される. `opcount.py` は, この `source.sage.py` を入力とする事を想定し設計されている

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

以降は, ソースコードに対して `sage --preparse` が必ず行われる事を 想定して PSDL の説明を行う. SageMath のシステムの都合により PSDL の ソースファイルの拡張子は `.sage` とする.

PSDL のコメントは Python 言語のコメントと同じで, `#` で始まり次の改行で 終わる.

PSDL と Python 言語の文法上の最も大きな違いは `^` 演算子の解釈である. `^` 演算子は Python 言語では bitwise xor 演算子を表現し算術演算より弱い優先度 を持つ. 一方 PSDL では `^` 演算子は TeX のようにべき乗を表現し, どの算術二項演算より強い優先度を持ち右結合する. xor 演算を記述する場合は `^^` と 書く. (xor 代入演算の場合は `^^=`).

PSDL ではアルゴリズムは関数の中に記述する.

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

関数名は `main` でなくても良い. 同名でなければいくつ定義しても良い. 例えば:

```
def setup():
    ...
```

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

データ型

Python の組み込みデータ型の他に, PSDL は次のデータ型 (class) を持つ.

型名	意味
`Group`	ペアリングのソース群 $\mathbb{G}$
`Group1`	非対称ペアリングのソース群 $\mathbb{G}_0$
`Group2`	非対称ペアリングのソース群 $\mathbb{G}_1$
`Target`	ペアリングの標的群
`Integer`	整数 (上記群の自己準同型環 (の可換部分環))

Python 言語 (および SageMath 言語) は動的型付言語であり, あらゆる変数に関して, その型は原理的には実行時に参照されるまで決定 されない. 従って, 例えば次のような暗号アルゴリズムの 演算コストは Enc が実際に呼び出されるまで, 厳密には決定されない.

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

これでは暗号方式の演算コストの解析が著しく面倒になってしまう為, PSDL では関数の各引数に対して, デフォールト引数を与える事により引数の型を決定し, 変数の型は実行時に変化しない事を想定する.

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

(CSDL と違い) Python や SageMath 同様 PSDL では, 上記関数の `c1` や `c2` のように変数は宣言せずに, いきなり代入できる. 変数の型は `opcount.py` が 持つ型推論アルゴリズムに従い決定され, 決定出来ない場合は warning が出力 される.

リスト値とリスト変数

PSDL と Python 言語の間には形式的文法上の違いはほとんど無いが, Python 言語の list 及び tuple は PSDL では単なるリストとして 同一視される. リスト値は

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

のように幾つかの要素をカッコで括って．リスト値の要素が全て左辺値なら, そのリスト値は左辺値 となる．即ち代入出来る．

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

リストは入れ子にする事が出来る．

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

変数にはリスト値を代入することが出来る．

```
L = (a, b, c)
```

関数

関数は Python 言語っぽく定義する．

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

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

返り値の型は `return` 文から推論される．PSDL では一つの関数につきその戻り値の型は唯一に決まると想定される．一般の Python 言語の関数のように, 戻り値の型が実行時に変化するような関数はサポートしない．

特殊形式

以下の特殊形式が存在する．同じ名前の変数定義や関数宣言をしてしまうと, そのスコープでは使用出来なくなるので注意が必要である．ついうっかり変数 `e` を宣言してしまう事が良くある．

関数	想定機能
`e (g0, g1)`	pairing
`Repeat (n, g)`	`g` を `n` 回繰り返す
`product(n, t)`	`t` を `n` 回かけた積
`inline(s)`	文字列リテラル `s` をそのまま出力

それぞれオペレーションの数を表すと想定されている適当な関数などに変換されるので, pari/gp 側で該当する関数を定義して使用する.

文	変換	想定機能
`for x in y: ...`	`for_statement("x","y",{body},{orelse})`	繰り返し
`if x: ...`	`if_statement("x",{body},{orelse})`	分岐

`for` 文や `if` 文は, コードが複雑になると, 繰り返し回数の平均値や確率の代数的な表現をプログラムで自動的に解析する事は現実的ではないので `opcount.py` には, そうした繰り返し回数や確率をユーザが自分で設定できるインターフェイスが用意されている. `opcount.py` は

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

なる PSDL の文を

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

なる計算コストとして計上し,

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

なる PSDL の文を

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

なる計算コストとして計上する. ここで, `"x"` および `"y"` は それぞれ, 正規化された 式 `x` および 式 `y` を表現する文字列で, `{body}`, `{orelse}` はそれぞれ `body` 節および `orelse` 節の計算コストである. `for` 文または `if` 文に `else` 節がない場合は`{orelse}` は 0 となる. `if` 文に `elif` 節がある場合は `if_statement()` がネストされる. pari/gp 側に適切な `for_statement()` 関数および `if_statement()` 関数をユーザが定義する事により, 繰り返し回数や確率の適当な表現を得る事が出来る. `inline` 関数は後続 の文字列リテラルをそのまま pari/gp 側に渡す. 例えば, 以下のような記述を PSDL の先頭の方に記述しておけば `range` を使った単純なループや条件分岐は 処理できるであろう.

```
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 ;

""")
```

非互換性

opcount.py と opcount には現在次の非互換性がある.

- 入力言語が異なる
- for 文の展開の仕方は入力言語の言語仕様の違い為, 微妙に異なる.
- opcount.py には -v オプションと -p オプションが無い.
- opcount.py はべき乗などの主要な演算以外 (代入演算など) は処理しない.
- opcount.py は _ で始まる関数の評価は行わない.

例

例 1

hello_world.sage を次のような内容のテキストファイルとする.

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

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

hello_world.sage を sage --preparse で処理してから opcount.py に入力すると次のような出力を得る.
(但し先頭が$の行はコマンド入力行を表すとする.)

```
$ sage --preparse hello_world.sage
```

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

PSDL 中で関数が定義されると, 関数内の演算コストがカウントされ call_関数名という変数にその値を代入するスクリプトが出力される. 上記の場合は main() 関数が定義されているので, call_main への代入式が出力されている. ここで GROUP_batch(n,b) は Group 型 (class) の多重冪乗の演算コスト (n, b) を表現している. 計算代数システム gp がこのスクリプトをこのままで評価することは出来ないが, 例えば preamble に

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

等と記述しておけば gp が評価可能な出力を得る事ができる.

例 2

elgamal.sage を次のような内容のテキストファイルとする.

```
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
```

PSDL (Python) にはプロトタイプ宣言という概念は無いので, 使用する関数に型を定義したい場合は関数を定義する必要がある. この例では適当な関数 Random() が Integer 型を返す関数として 定義されている. elgamal.sage を sage と opcount.py で処理すると次のような出力を得る.

```
$ 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);
```

定義された関数を, 他の関数の定義の中で呼び出すと call_関数名としてカウントされる. 例えば上記の Random() 関数は call_Random としてカウントされている.

例 3

boneh_franklin_0.sage を次のような内容のテキストファイルとする.

```
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
```

この例では定義済み関数 e(·,·) (ペアリング) を方式の定義に 使用している. boneh_franklin.csdl を sage と opcount.py で処理すると次のような出力を得る. (但し行末の\\は行継続を表すとする.)

```
$ 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);
```

ペアリングはターゲットグループとは独立した演算コストとしてカウントされ, n 個のペアリングの積が pairing_batch(n) としてカウントされる.

例 4

PSDL に対して 適当な Group class や Target class 等を定義すると sage で実行可能な方式を記述することが出来る. 例えば boneh_franklin.sage を次のような内容のテキストファイルとする. この例では, 適当な超特異楕円曲線を使って一連のペアリング関連 class が定義されている. opcount.py で処理可能でかつ sage で実行可能な記述例となっている. (但し行末の\は行継続を表すとする.)

```
import sys
import hashlib

p =  8736985826870563606813062066048743770757502222365475231311720845029506\
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 =  4059226329414343323749407940138224762816729370502327545609581477013353\
03156374283945087077910463695635704719067129278651242991807090314845147890 9\
4287053963
Y =  4741563758530164509327879396707294542392348025877185149430608664058681\
22149192237765537321391701267519837177188987632012307188831810636403453189 2\
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)
```