Corrigé

Cryptographie à clef secrète

1. Chiffrement de Vigenère

Question 1.

```
def chiffre(clef, message):
    s = ''
    i = 0
    for c in message:
        s = s + alph[(alph.index(c)+alph.index(clef[i])) % 26]
        i = (i+1) % len(clef)
    return s

def dechiffre(clef, message):
    s = ''
    i = 0
    for c in message:
        s = s + alph[(alph.index(c)-alph.index(clef[i])) % 26]
        i = (i+1) % len(clef)
    return s
```

Le premier message une fois déchiffré se révèle être la *Chanson d'automne* de Paul Verlaine.

Question 2. On utilise le script suivant :

```
s = open('hugo.txt', 'r')
francais = [0] * 26
n = 0
for l in s:
    for c in l:
        if c in alph:
            n += 1
            francais[alph.index(c)] += 1
for i in range(26):
    francais[i] /= n
s.close()
```

Question 3.

```
def frequence(texte):
    f = [0] * 26
    for c in texte:
        if c not in alph:
            print(c)
            continue
        f[alph.index(c)] += 1
    for i in range(26):
        f[i] /= len(texte)
    return f
```

Question 4. On observe que $\sum_i (x_i - \overline{x})(y_i - \overline{y}) = \sum_i x_i y_i - n\overline{x}\,\overline{y}$, $\sum_i (x_i - \overline{x})^2 = \sum_i x_i^2 - n\overline{x}^2$ et $\sum_i (y_i - \overline{y})^2 = \sum_i y_i^2 - n\overline{y}^2$, ce qui permet un calcul en un seul parcours des deux listes X et Y :

```
def correlation(X, Y):
    n = len(X)
    sx = sy = sx2 = sy2 = sxy = 0
    for x, y in zip(X, Y):
        sx += x
        sy += y
        sx2 += x * x
        sy2 += y * y
        sxy += x * y
    mx, my = sx / n, sy / n
    return (sxy - n * mx * my) / sqrt(sx2 - n * mx * mx) / sqrt(sy2 - n * my * my)
```

Question 5. Commençons par rédiger une fonction qui applique un certain décalage à un texte :

```
def decale(d, texte):
    s = ''
    for c in texte:
        s = s + alph[(alph.index(c) - d) % 26]
    return s
```

La fonction demandée s'écrit alors ainsi :

```
def clefL(message, l):
    d = [0] * l
    for k in range(l):
        m = -1
        for i in range(26):
            c = correlation(francais, frequence(decale(i, message[k::l])))
            if c > m:
                m, d[k] = c, i
    s = ''
    for k in range(l):
        s = s + alph[d[k]]
    return s
```

Cette fonction permet de trouver la clef du second message : NERVAL, et le message une fois déchiffré se révèle être le poème *El Desdichado* de Gérard de Nerval.

Question 6.

```
def clef(message):
    sol = (0, [])
    for l in range(4, 21):
        d = [0] * l
        c = [-1] * l
        for k in range(l):
            for i in range(26):
                s = correlation(francais, frequence(decale(i, message[k::l])))
                if s > c[k]:
                    c[k], d[k] = s, i
        moy = sum(c) / l
        if moy > sol[0]:
            s = ''
            for k in range(l):
                s = s + alph[d[k]]
            sol = (moy, s)
    return sol[1]
```

Le troisième message est chiffré à l'aide de la clef : APOLINAIRE et se révèle être le poème *Le Pont Mirabeau*, de Guillaume Apollinaire.

2. Échange de clefs Diffie-Hellman

Courbes elliptiques

Question 7. Le calcul du pgcd se réalise récursivement à partir de la formule d'Euclide :

```
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a % b)
```

Pour obtenir les coefficients de Bézout, faisons l'observation suivante : si $a = bq + r$ et si $ub + vr = \text{pgcd}(a,b)$ alors $ub + v(a - bq) = \text{pgcd}(a,b)$, soit encore : $va + (u - qv)b = \text{pgcd}(a,b)$. On en déduit la fonction :

```
def bezout(a, b):
    if b == 0:
        return (a, 1, 0)
    q, r = divmod(a, b)
    (d, u, v) = bezout(b, r)
    return (d, v, u - q * v)
```

Tout ceci conduit à une fonction efficace pour calculer l'inverse modulo p :

```
def inverse(p, x):
    (d, _, v) = bezout(p, x)
    if d != 1:
        raise ValueError("{} n'est pas inversible modulo {}".format(x, p))
    return v % p
```

Question 8. Il peut être utile de définir deux variables globales :

```
inf = float('inf')
O = (inf, inf)
```

Puis la fonction :

```
def addition(P, Q, a=0, b=5, p=17252297107):
    if Q == O:
        return P
    if P == O:
        return Q
    (xp, yp) = P
    (xq, yq) = Q
    if xp != xq:
        r = (yp - yq) * inverse(p, xp - xq) % p
        s = (yq * xp - yp * xq) * inverse(p, xp - xq) % p
        x = (r * r - xp - xq) % p
        y = (-r * x - s) % p
        return (x, y)
    elif P == Q and yp != 0:
        r = (3 * xp * xp + a) * inverse(p, 2 * yp) % p
        s = (yp - xp * r) % p
        x = (r * r - xp - xq) % p
        y = (-r * x - s) % p
        return (x, y)
    else:
        return O
```

Question 9.

```
def mult(k, P):
    if k == 0:
        return O
    elif k == 1:
        return P
    elif k % 2 == 0:
        return mult(k // 2, addition(P, P))
    else:
        return addition(P, mult(k // 2, addition(P, P)))
```

Question 10. L'algorithme d'exponentiation rapide est la version multiplicative de l'algorithme précédent :

```
def exp(x, n, p=17252297107):
    if n == 0:
        return 1
    elif n == 1:
        return x
    elif n % 2 == 0:
        return exp(x * x % p, n // 2)
    else:
        return x * exp(x * x % p, n // 2) % p
```

Si $p = 4m+3$ et s'il existe $z \in \mathbb{K}$ tel que $x = z^2$ alors d'après le petit théorème de Fermat on a $z^{4m+2} = 1$, autrement dit $x^{2m+1} = 1$. Mais alors $y^2 = x^{2m+2} = x$.

Puisque l'entier p que nous utilisons est congru à 3 modulo 4, la remarque ci-dessus conduit à une méthode pour tirer au hasard un point de G : on prend au hasard un entier x de l'intervalle $[\![0, p-1]\!]$; si $x^3 + ax + b$ possède une racine carrée y alors le point (x, y) appartient à G.

On définit d'abord une fonction qui calcule la racine carrée :

```
def psqrt(x, p=17252297107):
    m = (p - 3) // 4
    y = exp(x, m+1)
    if y * y % p == x:
        return y
    return None
```

puis la fonction génératrice :

```
def genere_point(a=0, b=5, p=17252297107):
    while True:
        x = randint(p)
        y = psqrt((x * x * x + a * x + b) % p)
        if y is None:
            continue
        return (x, y)
```

Question 11. Imaginons qu'Alice choisisse le nombre secret $a = 12345678$ et Bob le nombre $b = 87654321$. Alice calcule puis envoie à Bob $a\mathrm{P} = (4007575112, 1815875655)$. Ce dernier calcule alors $\mathrm{K} = b(a\mathrm{P}) = (6928751830, 15047157867)$.

De son côté, Bob calcule puis envoie à Alice $b\mathrm{P} = (17111385050, 2423289360)$.

Cette dernière calcule $a(b\mathrm{P}) = (6928751830, 15047157867) = \mathrm{K}$ et obtient bien la même clef secrète que son partenaire.

L'attaque de Shanks

Question 12. On définit la fonction :

```
def shanks(P, Q, r=4313008603):
    m = int(sqrt(r)) + 1
    S = O
    L = []
    for i in range(m):
        L.append(S)
        S = addition(S, P)
    (x, y) = S
    S = (x, -y)  # S = -mP
    for j in range(m):
        for i, R in enumerate(L):
            if R == Q:
                return (i + m * j) % r
        Q = addition(Q, S)
    return None
```

À partir de la valeur de P et du point (17111385050, 2423289360) transmis par Bob, Alice peut retrouver le secret de Bob grâce à la fonction ci-dessus :

```
>>> shanks(P, (17111385050,2423289360))
87654321
```

Il en est de même de Bob, à partir du message envoyé par Alice :

```
>>> shanks(P, (4007575112, 1815875655))
12345678
```