De la quantique en cryptographie

Élie Besnard, Malo Leroy,
Yun Marcola–da-Cunha Macedo

26 juin 2022

Table des matières

1 Ordinateur quantique **1**
1.1 Qubits 1
1.2 Portes et oracles 5
1.3 Calcul formel 8
1.4 Fonctions utiles 25

2 Ordinateur quantique : tests **26**
2.1 Qubits 26
2.2 Portes et oracles 29
2.3 Calcul formel 32
2.4 Fonctions utiles 44
2.5 Analyses d'efficacité 45

3 Algorithmes **48**
3.1 Deutsch-Jozsa et Bernstein-Vazirani 48
3.2 Grover 48
3.3 Shor 49
3.4 Parité 51
3.5 Tests d'algorithmes 52

4 Polarisation **53**

5 Interface graphique **54**

1 Ordinateur quantique

1.1 Qubits

```
from random import choices
from calcul import Matrice, un, zero, int_log2, int_vers_strbin, bin_vers_int, int_vers_bin


class EtatPropre:
```

```python
    def __init__(self, nom):
        # nom est un int ou un str a l'entree (il est converti en str)
        self.nom =str(nom)

    def __str__(self):
        return '|' +self.nom +'   '

    def __and__(self, autre):
        assert isinstance(autre, EtatPropre)
        return EtatPropre(self.nom +autre.nom)

    def __eq__(self, autre):
        # On ne teste pas l'egalite des noms
        return isinstance(autre, EtatPropre) and autre.valeur ==self.valeur

    # Change le nom d'un etat propre
    def renomme(self, nom):
        self.nom =str(nom)


# L'equivalent d'un Qubit, mais avec dim etats propres differents
class Qudit:
    # Un Qudit possede dim etats possibles
    def __init__(self, mat: Matrice):
        assert mat.q ==1
        self.dim =mat.p
        nom = lambda i: int_vers_strbin(i, taille =int_log2(self.dim)-1)
        self.base =[EtatPropre(nom(i)) for i in range(self.dim)] # vecteurs propres
        self.matrice =mat

    @staticmethod
    def zero(dim: int):
        return Qudit(Matrice.colonne(*([un] +[zero] *(dim-1))))

    def __eq__(self, autre):
        return self.matrice ==autre.matrice

    def __getitem__(self, bras):
        if isinstance(bras, int):
            return self.matrice[bras]
        assert all([(i ==0 or i ==1) for i in bras])
        return self.matrice[bin_vers_int(*bras)]

    def __setitem__(self, bras, valeur):
        self |= bras, valeur

    # Notation plus commode pour les circuits
    def __rshift__(self, autre):
        return autre *self

    # Stockage d'une valeur dans une composante
    # La condition de normalisation peut ne plus tre verifiee apres
    def __ior__(self, autre):
        bra, val =autre
        if isinstance(bra, int):
            self.matrice[bra] =val
        else:
            assert isinstance(bra, Bra)
```

```python
            self |= (bin_vers_int(*bra.composante), val)
        return self

    def __mul__(self, bra):
        assert isinstance(bra, Bra)
        return self.matrice *bra.matrice

    def __matmul__(self, autre):
        return Qudit(self.matrice @ autre.matrice)

    def mesure(self):
        probs =[float(abs(self[i])*abs(self[i])) for i in range(self.dim)]
        choix =choices(list(range(self.dim)), weights =probs)[0]
        for i in range(self.dim):
            self |= (i, zero)
        self |= (choix, un)
        return self

    def __neg__(self):
        return Qudit(Matrice([
            [- self[i]] for i in range(self.dim)
        ]))

    def __str__(self):
        return ' + '.join([
            str(self[i]) +str(self.base[i])
            for i in range(self.dim)
            if self[i] !=zero
        ])

    def __repr__(self):
        return str(self)


class Qubit(Qudit):
    def __init__(self, c0 =None, c1 =None):
        super().__init__(Matrice.colonne(un, zero))
        if c0 is not None and c1 is not None:
            assert abs(c0) *abs(c0) +abs(c1) *abs(c1) ==un
            self.matrice[0] =c0
            self.matrice[1] =c1

    @staticmethod
    def propre(n: int):
        assert n ==0 or n ==1
        if n ==0:
            return Qubit.zero()
        return Qubit.un()

    @staticmethod
    def zero():
        return Qubit()

    @staticmethod
    def un():
        return Qubit(zero, un)

    def _puissance_rapide(self, n :int):
```

```
        if self ==ket(0):
            return Qudit.zero(2 **n)
        return Qudit(Matrice.colonne(*([zero] *(2**n-1) +[un])))

    def __pow__(self, n: int):
        if self ==ket(0) or self ==ket(1):
            return self._puissance_rapide(n)
        if n ==1:
            return self
        a = self **(n//2)
        if n % 2 ==1:
            return self @ (a @ a)
        return a @ a


def ket(*arg, taille =None):
    assert taille is None or isinstance(taille, int)
    d = []
    for i in arg:
        d += int_vers_bin(int(i))
    assert all([i ==0 or i ==1 for i in d])
    if taille is not None:
        d = (taille -len(d)) *[0] +d
    q = Qubit.propre(d[0])
    for i in d[1:]:
        q = q @ Qubit.propre(i)
    return q


class Bra:
    def __init__(self, *composante):
        self.composante =()
        for i in composante:
            self.composante +=tuple(int_vers_bin(int(i)))
        self.matrice =Matrice.zeros(1, 2 **len(self.composante))
        self.matrice[bin_vers_int(*self.composante)] =un

    def __eq__(self, autre):
        return (isinstance(autre, Bra)
            and self.composante ==autre.composante)

    def __or__(self, autre):
        if isinstance(autre, Qudit):
            return autre[self.composante]
        # Il s'agit d'une assignation
        return self, autre

    def __matmul__(self, autre):
        assert isinstance(autre, Bra)
        return Bra(*(self.composante +autre.composante))

    def __pow__(self, n):
        n = int(n)
        if n ==1:
            return self
        a = self **(n//2)
        if n % 2:
            return self @ (a @ a)
```

```
        return (a @ a)

    def __str__(self):
        if isinstance(self.composante, int):
            return '   ' + str(self.composante) +'|'
        return '   ' + ''.join([str(i) for i in self.composante]) +'|'


def bra(*composante):
    return Bra(*composante)
```

1.2 Portes et oracles

```
from calcul import un, i, sqrt, Matrice, expi, Expi, pi
from qubit import Qudit, bra, ket

class Porte:
    # Une `Porte` s'utilise comme une matrice, en multipliant
    def __init__(self, matrice):
        assert isinstance(matrice, Matrice)
        assert matrice.p ==matrice.q
        assert (matrice.p ==1) or matrice.p % 2 ==0
        self.matrice =matrice
        self.taille =matrice.p //2 # 0 si c'est la porte neutre

    @staticmethod
    def neutre():
        return _neutre

    def __eq__(self, autre):
        return isinstance(autre, Porte) and self.matrice ==autre.matrice

    def __mul__(self, autre):
        if self ==Porte.neutre():
            return autre
        if autre ==Porte.neutre():
            return self
        if isinstance(autre, Qudit):
            return Qudit(self.matrice *autre.matrice)
        elif isinstance(autre, Porte):
            return Porte(self.matrice *autre.matrice)
        raise TypeError(f'{autre} n\'est ni une porte chanable ni un qudit')

    def __rshift__(self, autre):
        return autre *self

    def dague(self):
        return Porte(self.matrice.transposee().conjuguee())

    def __matmul__(self, autre):
        assert isinstance(autre, Porte)
        return Porte(self.matrice @ autre.matrice)

    def __pow__(self, n: int):
        if self ==H and n ==7:
            return __import__('hadamarapide').H7
```

```
        return Porte(self.matrice **n)

    def __neg__(self):
        return Porte(- self.matrice)

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

    def __repr__(self):
        return str(self)


_neutre =Porte(Matrice.identite(1))

I = Identite =Porte(Matrice([[1, 0], [0, 1]]))

H = Hadamard =Porte(sqrt(un /2) *Matrice([[1, 1], [1, -1]]))

X = PauliX =Porte(Matrice([[0, 1], [1, 0]]))

Y = PauliY =Porte(Matrice([[0, -i], [i, 0]]))

iY = iPauliY =Porte(Matrice([[0, 1], [-1, 0]]))

Z = PauliZ =Porte(Matrice([[1, 0], [0, -1]]))

R = lambda phi: Porte(Matrice([[1, 0], [0, expi(phi)]]))

PhaseCond =lambda n: Porte((ket(0) **n) *(bra(0) **n) *2 -(Matrice.identite(2**n)))

S = SWAP =Porte(Matrice([
    [1, 0, 0, 0],
    [0, 0, 1, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1]
]))

cX = CNOT =Porte(Matrice([
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1],
    [0, 0, 1, 0]
]))

def QFT(N: int):
    omega =Expi(pi *2 / N)
    t = [[None] *N for i in range(N)]
    for k in range(N):
        for j in range(N):
            t[k][j] =(omega **(k * j)).sous()
    return Porte(sqrt(un /N) *Matrice(t))
```

```
from qubit import bra, ket, Qudit, Qubit
from calcul import zero, un, F2, int_vers_bin, sqrt, int_log2, Matrice, F2Uplet, bin_vers_int

class Oracle:
    @staticmethod
    def phase(f):
```

```
        return OracleDePhase(f)

    @staticmethod
    def somme(f, *, m =1):
        return OracleDeSomme(f, m =m)

    @staticmethod
    def brut(f):
        return OracleBrut(f)


class OracleBrut:
    def __init__(self, f):
        self.f =f

    def __mul__(self, qudit):
        assert isinstance(qudit, Qudit)
        return ket(self.f(qudit))


class OracleDePhase:
    # f est une fonction de (F2)^n dans F2
    def __init__(self, f):
        self.f =f

    def __mul__(self, qudit):
        n = int_log2(qudit.dim) -1
        r = Qudit.zero(qudit.dim)
        for i in range(qudit.dim):
            r |= bra(i) | (bra(i) | qudit) *(-un) **self.f(
                *[F2(i) for i in int_vers_bin(i, taille=n)])
        return r


class OracleDeSomme:
    # f est une fonction de (F2)^n dans (F2)^m
    # par defaut, la taille de y est 1 (f va de (F2)^n dans F2)
    def __init__(self, f, *, m =1):
        self.f =f
        self.m =m

    def _trouve_i0(self, psi):
        for i in range(psi.dim):
            if bra(i) | psi !=zero:
                return i

    def _coord_y(self, c_non_nul, c_autre):
        a = (c_autre /c_non_nul)
        return sqrt((a*a +un).inverse())

    def _trouve_alpha_beta(self, psi):
        i0 = self._trouve_i0(psi)
        i1 = i0+1 if i0 % 2 ==0 else i0-1
        c = self._coord_y(psi[i0], psi[i1])
        if i0 % 2 ==0:
            alpha =c
            beta =sqrt(un -alpha*alpha)
        else:
```

```
            beta = c
            alpha = sqrt(un - beta*beta)
        return alpha, beta

    def _trouve_coord_x(self, alpha, beta, psi):
        n = psi.dim // 2
        if alpha != zero:
            a_inv = alpha.inverse()
            return [a_inv * psi[2*i] for i in range(n)]
        b_inv = beta.inverse()
        return [b_inv * psi[2*i+1] for i in range(n)]

    def _trouve_x_y(self, psi):
        x, y = psi, None
        for i in range(self.m):
            alpha, beta = self._trouve_alpha_beta(x)
            coord_x = self._trouve_coord_x(alpha, beta, x)
            x = Qudit(Matrice.colonne(coord_x))
            if y is None:
                y = Qubit(alpha, beta)
            else:
                y = Qubit(alpha, beta) @ y
        return x, y

    def __mul__(self, qudit):
        psi = Qudit(
            Matrice([[abs(qudit[i])] for i in range(qudit.dim)]))
        n = int_log2(psi.dim // 2) - self.m
        x, y = self._trouve_x_y(psi)
        res = Qudit.zero(psi.dim)
        res[0] = zero
        for i in range(2**n):
            fx = self.f(*[F2(k) for k in int_vers_bin(i, taille=n)])
            if isinstance(fx, F2): fx = (fx,)
            u = F2Uplet(*fx)
            for j in range(2**self.m):
                v = F2Uplet(*int_vers_bin(j, taille=self.m))
                c = tuple(int_vers_bin(i, taille=n)) + tuple(u + v)
                res[bin_vers_int(*c)] = x[i] * y[j]
        return res
```

1.3 Calcul formel

```
class Nombre:
    @staticmethod
    def ou_int(x):
        if isinstance(x, int):
            return Relatif(x).sous()
        return x

    def sur(self, E: type):
        if E == type(self):
            return self
        if not self.peut_sur(E):
            return None
        if hasattr(self, '_sur'):
```

```
            return self._sur(E)

    def appartient(self, E):
        return not (self.sur(E) is None)

    def peut_sur(self, E):
        T = type(self)
        d = {
            F2: [Naturel, Relatif, Rationnel, Puissance, Complexe],
            Naturel: [F2, Relatif, Rationnel, Puissance, Complexe],
            Relatif: [F2, Rationnel, Puissance, Complexe],
            Rationnel: [Puissance, Complexe],
            Puissance: [Complexe],
            Complexe: [Expi],
            Expi: [Complexe]
        }
        return E in d[T] if T in d else False

    def __add__(self, autre):
        T = type(self)
        autre =Nombre.ou_int(autre)
        if isinstance(autre, Somme):
            return autre +self
        if self ==zero:
            return autre
        if autre ==zero:
            return self
        b = autre.sur(T)
        if b is None:
            return (autre +self).sous()
        return self.plus(b).sous()

    def __mul__(self, autre):
        autre =Nombre.ou_int(autre)
        if self ==zero or autre ==zero:
            return zero
        if isinstance(autre, Matrice) or isinstance(autre, Somme):
            return autre *self
        T = type(self)
        b = autre.sur(T)
        if b is None:
            return (autre *self).sous()
        return self.fois(b).sous()

    def __sub__(self, autre):
        return self +(- autre)

    def signe(self):
        return 1 if abs(self) ==self.sous() else -1

    def __float__(self):
        if isinstance(self, Complexe):
            return float(self.x) +1j *float(self.y)
        if isinstance(self, Expi):
            return float(abs(self)) **(1j *float(self.arg()))
        a = self.sur(Puissance)
        if a is not None:
            return a.sigma *((a.x.num /a.x.denom) **(a.p.num /a.p.denom))
```

```
        raise NotImplementedError

    def __truediv__(self, autre):
        autre =Nombre.ou_int(autre)
        return self *autre.inverse()

    def __repr__(self):
        return str(self)

    def arg(self):
        if self.signe() ==1:
            return zero
        return pi

    def conjugue(self):
        return self

    def abs_carre(self):
        return abs(self) **2

class MultiplicationErreur(ArithmeticError):
    def __init__(self, n1, n2, message =None):
        self.message =message or f'Multiplication impossible: {n1} * {n2}'
        self.message +=' (types: ' +str(type(n1))[8:-2] +' et ' +str(type(n2))[8:-2] +')'
        super().__init__(self.message)

class AdditionErreur(ArithmeticError):
    def __init__(self, n1, n2, message =None):
        self.message =message or f'Addition impossible: {n1} + {n2}'
        self.message +=' (types: ' +str(type(n1))[8:-2] +' et ' +str(type(n2))[8:-2] +')'
        super().__init__(self.message)


class Naturel(Nombre):
    def __init__(self, n: int):
        assert n >=0
        self.n =n

    def __eq__(self, autre):
        return isinstance(autre, Naturel) and autre.n ==self.n

    def __int__(self):
        return self.n

    def sous(self):
        return self

    def _sur(self, E: type):
        if E ==F2:
            return F2(self.n)
        return Relatif(self.n).sur(E)

    def plus(self, autre):
        return Naturel(self.n +autre.n)

    def fois(self, autre):
        return Naturel(self.n *autre.n)
```

```python
    def inverse(self):
        return Rationnel(un, self.n)

    def __mod__(self, autre):
        return Naturel(int(self) % int(autre)).sous()

    def __floordiv__(self, autre):
        return Relatif(self.n //int(autre)).sous()

    def __pow__(self, exposant):
        return self.sur(Relatif) **exposant

    def __neg__(self):
        return Relatif(- self.n).sous()

    def __abs__(self):
        return self

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


class Relatif(Nombre):
    def __init__(self, z: int):
        self.z =z

    def __eq__(self, autre):
        return isinstance(autre, Relatif) and autre.z ==self.z

    def __int__(self):
        return self.z

    def sous(self):
        if self.z >=0:
            return Naturel(self.z)
        return self

    def _sur(self, E):
        return Rationnel(self.z, 1).sur(E)

    def plus(self, autre):
        return Relatif(self.z +autre.z)

    def fois(self, autre):
        return Relatif(self.z *autre.z)

    def inverse(self):
        return Rationnel(self.signe(), abs(self.z)).sous()

    def __mod__(self, autre):
        return Relatif(int(self) % int(autre)).sous()

    def __floordiv__(self, autre):
        return Relatif(self.z //int(autre)).sous()

    def __neg__(self):
        return Relatif(- self.z).sous()
```

```
    def __abs__(self):
        return Naturel(abs(self.z))

    def __pow__(self, exposant):
        exposant =Nombre.ou_int(exposant)
        if exposant.appartient(Naturel):
            return Relatif(self.z **int(exposant)).sous()
        if exposant.appartient(Relatif):
            return Rationnel(
                un,
                self.z **abs(int(exposant))
            ).sous()
        return self.sur(Rationnel) **exposant

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


zero =Naturel(0)
un = Naturel(1)


def pgcd(a, b) ->int:
    a, b =abs(int(a)), abs(int(b))
    if b ==0:
        return a
    return pgcd(b, a % b)


class Rationnel(Nombre):
    def __init__(self, num: int, denom: int):
        assert denom !=0
        num, denom =int(num), int(denom)
        d = pgcd(abs(num), abs(denom))
        signe =1 if num *denom >=0 else -1
        self.num =signe *abs(num) //d
        self.denom =abs(denom) //d

    def __eq__(self, autre):
        return (isinstance(autre, Rationnel)
                and self.num ==autre.num
                and self.denom ==autre.denom)

    def sous(self):
        if self.denom ==1:
            return Relatif(self.num).sous()
        return self

    def _sur(self, E):
        sigma =1 if self.num >=0 else -1
        return Puissance(
            Rationnel(abs(self.num), self.denom),
            un,
            sigma
        ).sur(E)

    def fois(self, autre):
        return Rationnel(self.num *autre.num,
```

```
                    self.denom *autre.denom)

    def plus(self, autre):
        return Rationnel(self.num *autre.denom +autre.num *self.denom,
                         self.denom *autre.denom).sous()

    def __neg__(self):
        return Rationnel(- self.num, self.denom)

    def __abs__(self):
        return Rationnel(abs(self.num), self.denom)

    def signe(self):
        return 1 if self.num >=0 else -1

    def __pow__(self, exposant):
        exposant =Nombre.ou_int(exposant)
        if exposant ==zero:
            return un
        if self.sous() ==zero:
            return zero
        if exposant.appartient(Naturel):
            return Rationnel(Relatif(self.num **exposant.n).z,
                             Relatif(self.denom **exposant.n).z).sous()
        if exposant.appartient(Relatif):
            return self.inverse() **(- exposant)
        raise ArithmeticError(f'Exponentiation impossible: {self}**{exposant}')

    def inverse(self):
        assert self.num !=0
        return Rationnel(self.denom, self.num).sous()

    def __str__(self):
        return f'{self.num}/{self.denom}'


class Puissance(Nombre):
    def __init__(self, x, p, sigma: int =1):
        assert sigma ==1 or sigma ==-1
        x, p =Nombre.ou_int(x), Nombre.ou_int(p)
        self.x =x.sur(Rationnel)
        assert self.x.num >=0
        self.p =p.sur(Rationnel)
        self.sigma =sigma

    def __eq__(self, autre):
        autre =Nombre.ou_int(autre).sur(Puissance)
        return (autre is not None
                and self.sigma ==autre.sigma
                and self.x **Relatif(self.p.num) ==autre.x **Relatif(autre.p.num)
                and self.p.denom ==autre.p.denom)

    def sous(self):
        if self.p.appartient(Relatif):
            return self.sigma *(self.x **self.p)
        r = self._sous_racine()
        if r is not None:
            return Relatif(self.sigma) *(r **Relatif(self.p.num).sous())
```

```
        return self

    def _sous_racine(self):
        dp = self.p.denom
        for dr in range(1, self.x.denom +1):
            eta = self.x *Naturel(dr **dp)
            if eta.appartient(Naturel):
                nr = 0
                while nr **dp < eta.n:
                    nr += 1
                if nr **dp == eta.n:
                    return Rationnel(nr, dr)
        return None

    def _sur(self, E: type):
        if E == Complexe:
            return Complexe(self.sous(), zero)

    def fois(self, autre):
        s = self.sigma *autre.sigma
        if self.x == autre.x:
            return Puissance(self.x, self.p +autre.p, s).sous()
        if self.x == autre.x.inverse():
            return Puissance(self.x, self.p -autre.p, s).sous()
        if self.p == autre.p:
            return Puissance(self.x *autre.x, self.p, s).sous()
        if self.p ==- autre.p:
            return Puissance(self.x /autre.x, self.p, s).sous()
        if autre.sous().sur(Rationnel) is not None:
            a = Puissance(abs(autre.sous().sur(Rationnel)), self.p.inverse()).sous().sur(Rationnel)
            if a is not None:
                return Puissance(self.x *a, self.p, self.sigma *autre.signe()).sous()
        raise MultiplicationErreur(self, autre)

    def inverse(self):
        assert self.x != zero
        return Puissance(self.x, -self.p, self.sigma)

    def __pow__(self, autre):
        s = Nombre.ou_int(autre).sous()
        if isinstance(s, Naturel):
            return Puissance(self.x, self.p *s,
                             sigma=1 if s.n % 2 ==0 else self.sigma).sous()
        if isinstance(s, Relatif):
            return self.inverse() **(-s)
        raise NotImplementedError

    def plus(self, autre):
        if autre ==-self:
            return zero
        if autre ==self:
            return self *2
        return Somme(self.sous(), autre.sous(), feuille=True)

    def __neg__(self):
        return Puissance(self.x, self.p, -self.sigma)

    def __abs__(self):
```

```
        return Puissance(self.x, self.p)

    def __str__(self):
        s = '' if self.sigma ==1 else '-'
        if self.p ==Rationnel(1, 2):
            return f'{s}sqrt({str(self.x.sous())})'
        x, p =self.x.sous(), self.p.sous()
        s += str(x) if x.appartient(Naturel) else f'({str(x)})'
        if p.sous() !=un:
            s += '^'
            s += str(p) if p.appartient(Naturel) else f'({str(p)})'
        return s


def sqrt(r):
    return Puissance(r, Rationnel(1, 2)).sous()


class Complexe(Nombre):
    def __init__(self, x, y =0):
        x, y =Nombre.ou_int(x), Nombre.ou_int(y)
        self.x =x.sous()
        self.y =y.sous()

    def __eq__(self, autre):
        if isinstance(autre, Expi):
            s = self.sur(Expi)
            if s is not None:
                return s ==autre
        return (isinstance(autre, Complexe)
                and self.x ==autre.x
                and self.y ==autre.y)

    def sous(self):
        if self.y ==zero:
            return self.x
        return self

    def _sur(self, E):
        try:
            a = self.arg()
        except ArithmeticError:
            return
        return Expi(a, module=abs(self))

    def plus(self, autre):
        return Complexe(
            self.x.sous() +autre.x.sous(),
            self.y.sous() +autre.y.sous()
        )

    def fois(self, autre):
        return Complexe(self.x *autre.x -self.y *autre.y,
                        self.x *autre.y +self.y *autre.x)

    def __neg__(self):
        return Complexe(- self.x, -self.y)
```

```python
    def conjugue(self):
        return Complexe(self.x, -self.y).sous()

    # Le module d'un nombre complexe
    def __abs__(self):
        return sqrt(self.x *self.x +self.y *self.y)

    def arg(self):
        if self.y ==zero: return self.x.arg()
        if self.x ==zero:
            if self.y.signe() ==1: return pi /2
            return -pi /2
        if self.y.signe() ==1:
            if self.x ==self.y: return pi /4
            if self.x ==-self.y: return (pi /4) *3
        if self.x ==-self.y: return -pi/4
        if self.x ==self.y: return (-pi /4) *3
        raise ArithmeticError(f'Extraction d\'argument impossible sur {self}')

    def __str__(self):
        sx = str(self.x)
        sy = str(self.y) if self.y.signe() ==1 else '-' +str(abs(self.y))
        sy += 'i'
        if self.y ==un: sy ='i'
        if self.y ==-un: sy ='-i'
        if self.x ==self.y ==zero: return '0'
        if self.x ==zero: return sy
        if self.y ==zero: return sx
        return sx +' + ' +sy


i = Complexe(zero, un)


def int_vers_bin(n: int, *, taille=None):
    assert n >=0
    b = [int(i) for i in bin(n)[2:]]
    return b if taille is None else [0] *(taille -len(b)) +b


def bin_vers_int(*valeurs):
    return int(''.join([str(i) for i in valeurs]) or '0', base=2)


def int_log2(n: int): # partie entiere superieure de log2(n)
    assert n >0
    return len(bin(n)[2:])


def strbin_vers_int(s):
    return int(s, base=2)


def int_vers_strbin(n: int, *, taille=None):
    assert n >=0
    s = bin(n)[2:]
    return s if taille is None else '0' *(taille -len(s)) +s
```

```python
class F2(Nombre):
    def __init__(self, n):
        if isinstance(n, int):
            self.n =n % 2
        else:
            self.n =n.sous().sur(Relatif).z % 2

    def __eq__(self, autre):
        return isinstance(autre, F2) and self.n ==autre.n

    def __int__(self):
        return self.n

    def sous(self):
        return self

    def plus(self, autre):
        return F2((self.n +autre.n) % 2)

    def fois(self, autre):
        return F2(self.n *autre.n)

    def _sur(self, E):
        return Naturel(self.n).sur(E)

    @staticmethod
    def uplet(*valeurs):
        return F2Uplet(*valeurs)

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


class F2Uplet:
    def __init__(self, *valeurs):
        self.taille =len(valeurs)
        self._valeurs =[F2(i) for i in valeurs]

    def __eq__(self, autre):
        return (isinstance(autre, F2Uplet)
                and self.taille ==autre.taille
                and all([self[i] ==autre[i] for i in range(self.taille)]))

    def __add__(self, autre):
        assert isinstance(autre, F2Uplet) and self.taille ==autre.taille
        return F2Uplet(*[self[i] +autre[i] for i in range(self.taille)])

    def __getitem__(self, indice):
        return self._valeurs[indice]

    def __setitem__(self, indice, valeur):
        assert isinstance(valeur, F2)
        self._valeurs[indice] =valeur

    def __int__(self):
        return bin_vers_int(*self._valeurs)
```

```python
    def __str__(self):
        return '(' +', '.join(str(i) for i in self._valeurs) +')'


class Matrice(Nombre):
    def __init__(self, t):
        assert all([len(i) ==len(t[0]) for i in t])
        self.p =len(t)
        self.q =len(t[0])
        self.forme =(self.p, self.q)
        self._c =[
            [Nombre.ou_int(t[i][j]) for j in range(self.q)]
            for i in range(self.p)
        ]

    def __eq__(self, autre):
        if (not isinstance(autre, Matrice)) or self.p !=autre.p or self.q !=autre.q:
            return False
        for i in range(self.p):
            for j in range(self.q):
                if self[i, j] !=autre[i, j]:
                    return False
        return True

    def sous(self):
        return self

    # Renvoie une matrice remplie de zeros
    # p est la longueur de la matrice (nombre de lignes)
    # q est la largeur de la matrice (nombre de colonnes)
    @staticmethod
    def zeros(p, q =None):
        if q is None:
            q = p
        assert isinstance(p, int) and isinstance(q, int)
        return Matrice([[zero for _ in range(q)] for _ in range(p)])

    def est_carree(self):
        return self.p ==self.q

    def __getitem__(self, item):
        if isinstance(item, tuple):
            i, j =item
            assert isinstance(i, int) and 0 <=i <self.p
            assert isinstance(j, int) and 0 <=j <self.q
            return self._c[i][j]
        assert isinstance(item, int)
        if self.p ==1:
            return self._c[0][item]
        assert self.q ==1
        return self._c[item][0]

    def __setitem__(self, cle, valeur):
        valeur =Nombre.ou_int(valeur)
        assert isinstance(valeur, Nombre), f'Pas un nombre : {valeur}'
        if isinstance(cle, tuple):
            i, j =cle
            assert isinstance(i, int) and 0 <=i <self.p
```

```
            assert isinstance(j, int) and 0 <=j <self.q
            self._c[i][j] =valeur
        else:
            assert isinstance(cle, int)
            if self.p ==1:
                self._c[0][cle] =valeur
            else:
                assert self.q ==1
                self._c[cle][0] =valeur

    def __add__(self, autre):
        assert self.p ==autre.p
        assert self.q ==autre.q
        return Matrice([
            [self[i, j] +autre[i, j] for j in range(self.q)]
            for i in range(self.p)
        ])

    def __mul__(self, autre):
        if isinstance(autre, Matrice):
            assert self.q ==autre.p
            m = Matrice.zeros(self.p, autre.q)
            for i in range(self.p):
                for j in range(autre.q):
                    m[i, j] =zero
                    for k in range(self.q):
                        m[i, j] +=self[i, k] *autre[k, j]
            return m
        autre =Nombre.ou_int(autre)
        if autre ==zero: return Matrice.zeros(self.p, self.q)
        if autre ==un: return self
        if autre ==-un: return -self
        return Matrice([
            [autre *self[i, j] for j in range(self.q)]
            for i in range(self.p)
        ])

    def transposee(self):
        return Matrice([
            [self[j, i] for i in range(self.p)] for j in range(self.q)
        ])

    def conjuguee(self):
        return Matrice([
            [self[i, j].conjugue() for j in range(self.q)]
            for i in range(self.p)
        ])

    @staticmethod
    def scalaire(k, n):
        return Matrice([
            [k if i ==j else zero for j in range(n)] for i in range(n)
        ])

    @staticmethod
    def identite(n):
        return Matrice.scalaire(un, n)
```

```
    @staticmethod
    def ligne(*args):
        l = []
        for i in args:
            if isinstance(i, list):
                l += i
            else:
                l.append(i)
        return Matrice([l])

    @staticmethod
    def colonne(*args):
        c = []
        for i in args:
            if isinstance(i, list):
                c += i
            else:
                c.append(i)
        return Matrice([[i] for i in c])

    # Correspond au produit tensoriel pour deux matrices (prod. de Kronecker)
    # S'utilise pour deux matrices A et B en ecrivant A @ B.
    # se fait en complexite O(self.p * self.q * autre.p * autre.q)
    def __matmul__(self, autre):
        assert isinstance(autre, Matrice)
        c = lambda i, j: self[i //autre.p, j //autre.q] *autre[i % autre.p, j % autre.q]
        return Matrice([[c(i, j) for j in range(self.q *autre.q)] for i in range(self.p *autre.p)])

    def __pow__(self, exposant):
        exposant =int(exposant)
        if self ==Matrice.identite(self.p) or exposant ==0:
            return Matrice.identite(self.p **exposant)
        if self._c ==[[un]]:
            return self
        if exposant ==1:
            return self
        a = self **(exposant //2)
        if exposant % 2 ==1:
            return self @ (a @ a)
        return a @ a

    def __str__(self):
        n = max([len(str(self[i, j])) for i in range(self.p) for j in range(self.q)])
        return '\n'.join([
            '( ' + ' '.join([
                str(self[i, j]).ljust(n) for j in range(self.q)
            ]) + ' )'
            for i in range(self.p)
        ])

    def __neg__(self):
        return Matrice([
            [-self[i, j] for j in range(self.q)] for i in range(self.p)
        ])


class VectPi(Nombre): # pi * t avec t.appartient(Puissance)
    _float_pi =3.141592653589793
```

```
def __init__(self, t):
    t = Nombre.ou_int(t)
    assert t.appartient(Puissance)
    self.t =t

def __float__(self):
    return VectPi._float_pi *float(self.t)

def mod2pi(self):
    t = self.t.sur(Rationnel)
    if t is None:
        return self
    if t.num <0:
        s = (self +(pi * 2))
        if s ==zero:
            return zero
        return s.mod2pi()
    if t.num >=2 * t.denom:
        s = (self -(pi * 2))
        if s ==zero:
            return zero
        return s.mod2pi()
    return self

def __eq__(self, autre):
    return (isinstance(autre, VectPi) and self.t ==autre.t)

def sous(self):
    if self.t ==zero:
        return zero
    return self

def __add__(self, autre):
    autre =Nombre.ou_int(autre)
    if autre ==zero:
        return self
    if isinstance(autre, VectPi):
        return VectPi(self.t +autre.t).sous()
    raise Somme(self, autre, feuille=True)

def __neg__(self):
    return VectPi(- self.t)

def __mul__(self, autre):
    autre =Nombre.ou_int(autre)
    if autre.appartient(Puissance):
        return VectPi(self.t *autre).sous()
    raise MultiplicationErreur(self, autre)

def __str__(self):
    if self.t ==un: return ''
    if self.t ==-un: return '-'
    if self.t.appartient(Relatif):
        return f'{self.t}'
    r = self.t.sur(Rationnel)
    if r is not None:
        return f'{r.num if r.num != 1 else ""}/{r.denom}'
```

```
        return f'({self.t})*'


class Expi(Nombre): # module * exp(i * arg)
    def __init__(self, arg, *, module=un):
        arg, module =Nombre.ou_int(arg), Nombre.ou_int(module)
        assert arg.appartient(Puissance) or arg.appartient(VectPi)
        assert module.signe() ==1
        a = arg.sur(VectPi)
        if a is not None:
            self._arg =a.mod2pi()
        else:
            self._arg =arg
        self._module =module

    def __eq__(self, autre):
        return (isinstance(autre, Expi)
                and self.arg() ==autre.arg()
                and abs(self) ==abs(autre))

    def arg(self):
        return self._arg

    def sous_leger(self):
        a = self.arg()
        if a ==zero: return un
        if abs(self) ==zero: return zero
        if a.appartient(VectPi):
            if a.t.appartient(Relatif):
                return abs(self) *((-un) **self.arg().t)
            t = a.t.sur(Rationnel)
            if t is not None and t.denom ==2:
                if t.num ==1: return i
                if t.num ==3: return -i
        return self

    def sous(self):
        s = self.sous_leger()
        return s

    def _sur(self, E):
        sl = self.sous_leger()
        if not isinstance(sl, Expi):
            return sl.sur(E)
        sa = self.arg()
        if sa.appartient(VectPi):
            t = sa.t.sur(Rationnel)
            if t is not None and t.denom ==4:
                re = 1 if t.num in (1, 7) else -1
                im = 1 if t.num in (1, 3) else -1
                m = abs(self) *sqrt(un /2)
                return Complexe(m *re, m *im).sur(E)

    def __abs__(self):
        return self._module

    def __mul__(self, autre):
        autre =Nombre.ou_int(autre)
```

```
        if autre.appartient(Puissance):
            if autre.signe() ==1:
                return Expi(self.arg(), module=abs(self) *autre).sous_leger()
            return Expi(self.arg() +pi, module=abs(self) *(-autre)).sous_leger()
        if isinstance(autre, Expi):
            return Expi(self.arg() +autre.arg(),
                        module=abs(self) *abs(autre)).sous_leger()
        if isinstance(autre, Complexe):
            s = autre.sur(Expi)
            if s is not None: return self *s
        raise MultiplicationErreur(self, autre)

    def __pow__(self, n):
        n = Nombre.ou_int(n)
        return Expi(self.arg() *n, module =abs(self) **n).sous()

    def __add__(self, autre):
        autre =Nombre.ou_int(autre).sous()
        if autre ==zero:
            return self
        if isinstance(autre, Expi):
            if self.arg() ==autre.arg():
                return Expi(self.arg(),
                            module =abs(self) +abs(autre))
            if self.arg() +pi ==autre.arg():
                r = abs(self) -abs(autre)
                if r.signe() ==1:
                    return Expi(self.arg(), module=r).sous()
                return Expi(self.arg() +pi, module=-r).sous()
            if self.arg() ==autre.arg() +pi:
                return autre +self
        s = self._sur(Complexe)
        if s is not None and self !=s:
            return s +autre
        return Somme(self, autre, feuille=True)

    def __neg__(self):
        return Expi(self.arg() +pi, module=abs(self))

    def inverse(self):
        return Expi(-self.arg(), module=abs(self).inverse())

    def conjugue(self):
        return Expi(-self.arg(), module=abs(self))

    def __str__(self):
        if abs(self) ==un:
            return f'exp(i*({self.arg()}))'
        return f'{abs(self)}*exp(i*({self.arg()}))'


pi = VectPi(1)
def expi(theta): return Expi(theta).sous()

def somme(*termes):
    return Somme(*termes).sous()

class Somme(Nombre):
```

```python
    # feuille: True si pas de simplifications a faire
    def __init__(self, *termes, feuille=False):
        assert isinstance(feuille, bool)
        self._pos =0
        if feuille:
            self._termes =[Nombre.ou_int(i) for i in termes]
        else:
            self._termes =[]
            for i in termes:
                self._ajoute_nombre(Nombre.ou_int(i))

    def __eq__(self, autre):
        if not isinstance(autre, Somme) or len(self) !=len(autre):
            return False
        for i in self:
            if i not in autre._termes:
                return False
        return True

    def __len__(self):
        return len(self._termes)

    def sous(self):
        if len(self) ==1:
            return self._termes[0]
        return self

    def __add__(self, autre):
        if isinstance(autre, Somme):
            return self._plus_somme(autre)
        return self._plus_nombre(autre)

    def __mul__(self, autre):
        if isinstance(autre, Somme):
            return Somme(*[
                self._termes[i] *autre._termes[j]
                for i in range(len(self))
                for j in range(len(autre))
            ]).sous()
        return Somme(*[i *autre for i in self]).sous()

    def _plus_somme(self, autre):
        s = Somme(*[i for i in self])
        for j in autre:
            s += j
        return s.sous()

    def _ajoute_nombre(self, autre):
        for i in range(len(self)):
            k = self._termes[i] +autre
            if not isinstance(k, Somme):
                self._termes.pop(i)
                self._ajoute_nombre(k)
                break
        else:
            self._termes.append(autre)

    def _plus_nombre(self, autre):
```

```
        r = Somme(*[i for i in self])
        r._ajoute_nombre(autre)
        return r.sous()

    def __iter__(self):
        self._pos =0
        return self

    def __next__(self):
        if self._pos <len(self):
            self._pos +=1
            return self._termes[self._pos-1]
        raise StopIteration

    def __neg__(self):
        return Somme(*[-i for i in self])

    def signe(self):
        return 1

    def __abs__(self):
        if all(hasattr(i, 'signe') and i.signe() ==1 for i in self):
            return self.sous()
        if all(hasattr(i, 'signe') and i.signe() ==-1 for i in self):
            return (-self).sous()
        return sqrt(self.abs_carre())

    def abs_carre(self): # renvoie le module au carre de la somme
        return self *self.conjugue()

    def conjugue(self):
        return Somme(*[i.conjugue() for i in self]).sous()

    def __float__(self):
        return sum(float(i) for i in self)

    def __sub__(self, autre):
        return self +(- autre)

    def __str__(self):
        return ' + '.join(str(i) for i in self)

    def __repr__(self):
        return '  [' +' + '.join(str(i) for i in self) +']'
```

1.4 Fonctions utiles

```
from portes import Porte, H, I
from qubit import bra, ket
from calcul import un, zero, int_vers_bin, int_log2, Matrice

# Teste si une liste d'entiers correspond a un qubit
# (ne fonctionne evidemment que pour les etats propres)
def sequence_egale(sequence_attendue, qubit_obtenu):
    val = bra(*sequence_attendue)
    resultat =val | qubit_obtenu
```

```
    return (2 **len(sequence_attendue) ==qubit_obtenu.dim
            and resultat ==un)

def etat_de_base(n_principal, n_auxiliaire, val_auxiliaire =0):
    return (ket(0) **n_principal) @ (ket(val_auxiliaire) **n_auxiliaire)

# ne fonctionne que pour les etats propres
def ket_vers_liste(q):
    n = q.dim
    for i in range(n):
        if bra(i) | q !=zero:
            return int_vers_bin(i, taille=int_log2(n)-1)

# Fonctionne comme range, la fin est exclue.
# Si ‘fin‘ est negatif on part de la fin.
def H_option(total, *, debut, fin):
    assert isinstance(total, int) and isinstance(fin, int)
    assert isinstance(debut, int) and debut >=0
    if fin <0:
        fin = total +fin
    A = (I ** debut)
    B = (H ** (fin - debut))
    return kron_id(A @ B, total -fin)

# Cree des etats propres et les fait tous passer dans une porte de Hadamard.
def qubits_intriques(n, valeur =0):
    assert isinstance(n, int) and isinstance(valeur, int)
    return (ket(valeur)**n) >>(H**n)

# On fait le calcul ‘m @ I(n)‘, avec I(n) l’identite de taille n,
# et m une matrice quelconque.
# Les analyses montrent que c’est en moyenne 30 fois plus rapide.
def kron_id_mat(m, n):
    r = Matrice.zeros(m.p *n, m.q *n)
    for i in range(m.p):
        for j in range(m.q):
            for k in range(n):
                r[i*n +k, j*n +k] = m[i, j]
    return r

# On calcule la *porte* ‘P @ (I ** 2)‘, avec I l’identite de taille 2.
def kron_id(P, n):
    return Porte(kron_id_mat(P.matrice, 2**n))
```

2 Ordinateur quantique : tests

2.1 Qubits

```
from unittest import TestCase, main
from qubit import Qubit, Qudit, ket, bra, Bra, EtatPropre
from calcul import un, zero, Matrice
from portes import H

class TestQubit:
    def test_ket_zero(self):
```

```
        q = Qubit()
        p = Qubit.zero()
        assert q[0] ==q[(0,)] ==un
        assert q[1] ==q[(1,)] ==zero
        assert p ==q

    def test_un(self):
        ket_un =Qubit.un()
        assert ket_un[0] ==zero
        assert ket_un[1] ==un

    def test_mesure(self):
        a, b =Qubit.un(), Qubit.zero()
        l = [(H * Qubit()).mesure() for _ in range(100)]
        assert a.mesure() ==Qubit.un()
        assert b.mesure() ==Qubit.zero()
        assert all([i in [a, b] for i in l])
        assert 2/3 <=l.count(a) /l.count(b) <=3/2

    def test_multiples_qubits(self):
        q = ket(0) @ ket(1)
        assert q[0, 0] ==zero
        assert q[0, 1] ==un
        assert q[1, 0] ==zero
        assert q[1, 1] ==zero
        assert str(q) =='1| 01 '

    def test_pow(self):
        e = ket(0) **3
        assert e[0, 0, 0] ==un
        for i in range(2):
            for j in range(2):
                for k in range(2):
                    if (i, j, k) !=(0, 0, 0):
                        assert e[i, j, k] ==zero

    def test_ket(self):
        assert ket(0) ==Qubit.zero() ==Qubit()
        assert ket(1) ==Qubit.un()
        assert ket(0, 0) ==Qubit.zero() @ Qubit.zero()
        assert ket(0, 1) ==Qubit.zero() @ Qubit.un()
        assert ket(1, 1, 1) ==Qubit.un() **3

    def test_ket_int(self):
        assert ket(6) ==ket(1, 1, 0)
        assert ket(2, 3) ==ket(1, 0, 1, 1)

    def test_ket_taille(self):
        assert ket(1, taille=3) ==ket(0, 0, 1)
        assert ket(1, 0, 1, taille=1) ==ket(1, 0, 1)

    def test_str_ket(self):
        assert str(ket(0, 0)) =='1| 00 '
        assert str(ket(1, 1, 0)) =='1|110'
        assert str(ket(1, 1, 1, 0)) =='1|1110'

    def test_change_composante(self):
        e0 = ket(1, 0)
```

```
        e0 |= bra(0, 0) | un
        e0 |= bra(1, 0) | zero
        assert e0 ==ket(0, 0)

    def test_neg(self):
        e0 = ket(1, 0)
        e1 = Qudit(Matrice.colonne(0, 0, -1, 0))
        assert e1 ==-e0
        assert e0 ==-e1

    def test_dim(self):
        e1 = Qudit(Matrice.colonne(0, 0, -1, 0))
        assert e1.dim ==4


class TestEtatPropre:
    def test_nom(self):
        ket_a =EtatPropre('a')
        ket_0 =EtatPropre(0)
        assert ket_a.nom =='a'
        assert ket_0.nom =='0'

    def test_renomme(self):
        e = EtatPropre('a')
        assert e.nom =='a'
        e.renomme('b')
        assert e.nom =='b'

    def test_str(self):
        e0 = EtatPropre(0)
        e1 = EtatPropre(1)
        assert str(e0) =='| 0 '
        assert str(e1) =='| 1 '

    def test_and(self):
        e0, e1 =EtatPropre(0), EtatPropre(1)
        e01 = e0 & e1
        e101 = e1 & e01
        assert str(e01) =='| 01 '
        assert str(e101) =='| 101 '


class TestBra(TestCase):
    def test_raccourci(self):
        assert Bra(0) ==bra(0)
        assert Bra(1) ==bra(1)
        assert Bra(1, 0, 1) ==bra(1, 0, 1)

    def test_bra(self):
        assert bra(0) | ket(0) ==un
        assert bra(1) | ket(0) ==zero
        assert bra(1, 0) | (ket(1) @ ket(0)) ==un
        assert bra(1, 1) | (ket(1) @ ket(0)) ==zero

    def test_eq(self):
        assert bra(0, 1) ==bra(zero, un)

    def test_produit(self):
```

```
        mx, my =Matrice.zeros(8, 1), Matrice.zeros(4, 1)
        mx[7] =un
        my[3] =un
        x, y =Qudit(mx), Qudit(my)
        z = x @ y
        for i in range(8):
            for j in range(4):
                if (i, j) !=(7, 3):
                    assert bra(i, j) | z ==zero
        assert bra(7, 3) | z ==un

    def test_ket_bra(self):
        k = ket(0, 1)
        b = bra(1, 1)
        m = Matrice.zeros(4)
        m[1, 3] =un
        assert k *b ==m

    def test_bra_vectoriel_bra(self):
        b1 = bra(1, 0)
        b2 = bra(1, 1, 0)
        b3 = bra(1, 0, 1, 1, 0)
        assert b1 @ b2 ==b3

    def test_pow(self):
        b0 = bra(0, 0)
        b1 = bra(0, 0, 0, 0, 0, 0)
        b2 = bra(1, 0, 1)
        b3 = bra(1, 0, 1, 1, 0, 1)
        assert b0 **3 ==b1
        assert b2 **2 ==b3


if __name__ =='__main__':
    main()
```

2.2 Portes et oracles

```
from unittest import TestCase, main
from calcul import Matrice, sqrt, un, zero, i
from portes import H, I, X, Y, Z, CNOT, S, Porte, PhaseCond, QFT
from qubit import Qubit, ket

class TestPortes(TestCase):
    def test_eq(self):
        assert H ==H
        assert I ==Porte(Matrice([[1, 0], [0, 1]]))
        assert I !=X

    def test_produit_tensoriel(self):
        M = Porte(un /2 * Matrice([
            [1,  1,  1,  1],
            [1, -1,  1, -1],
            [1,  1, -1, -1],
            [1, -1, -1,  1]
        ]))
```

```
        assert H @ H ==M

    def test_pow(self):
        assert H **3 ==H @ H @ H
        assert (I **2).matrice ==Matrice.identite(4)

    def test_dague(self):
        assert H *H.dague() ==I
        assert CNOT *CNOT.dague() ==I @ I

    def test_petit_circuit(self):
        C = S * (X @ I) *S * (I @ X)
        assert C.matrice ==Matrice.identite(4)
        assert C ==I @ I

    def test_droite_a_gauche(self):
        C1 = S * (X @ I) *S * (I @ X)
        C2 = (I @ X) >>S >> (X @ I) >>S
        assert C1 ==C2


class TestPortesRemarquables(TestCase):
    def test_identite(self):
        q = Qubit(sqrt(un /2), sqrt(un /2))
        assert I *ket(0) ==ket(0)
        assert I *ket(1) ==ket(1)
        assert I *q == q

    def test_pauli_x(self):
        ket_zero =ket(0)
        ket_un =ket(1)
        assert X *ket_zero ==ket_un
        assert X *ket_un ==ket_zero

    def test_pauli_y(self):
        p1 = Qubit(zero, i)
        p2 = Qubit(-i, zero)
        assert Y *ket(0) ==p1
        assert Y *ket(1) ==p2

    def test_pauli_z(self):
        moins_ket_un =Qubit(zero, -un)
        assert Z *ket(0) ==ket(0)
        assert Z *ket(1) ==moins_ket_un

    def test_hadamard(self):
        z = ket(0)
        u = ket(1)
        q1 = Qubit(sqrt(un /2), sqrt(un /2))
        q2 = Qubit(sqrt(un /2), -sqrt(un /2))
        assert H *z == q1
        assert H *u == q2

    def test_swap(self):
        q1 = ket(0) >>H
        q2 = ket(1) >>H
        assert (q1 @ q2) >>S ==q2 @ q1
        assert (q2 @ q1) >>S ==q1 @ q2
```

```
    def test_neutre(self):
        # la porte neutre n'est applicable sur aucun qubit
        n = Porte.neutre()
        ph, pi =H ** 0, I **0
        assert ph ==pi ==n
        assert n *n ==n == n ** 7
        assert n *(H **2) == H ** 2
        assert (H **2) *n == H ** 2

    def test_phase_conditionelle(self):
        pc = PhaseCond(3)
        assert ket(0, 0, 0) >>pc ==ket(0, 0, 0)
        assert ket(1, 1, 0) >>pc ==(- ket(1, 1, 0))

    def test_qft(self):
        m = un / 2 * Matrice([
            [1, 1, 1, 1],
            [1, i, -1, -i],
```

```
from unittest import TestCase, main
from calcul import un, F2, Matrice
from qubit import bra, ket, Qudit
from portes import H
from oracle import OracleDePhase, OracleDeSomme


class TestOracles(TestCase):
    def f(self, x, y):
        assert isinstance(x, F2) and isinstance(y, F2)
        return x +y

    def g(self, a, b):
        assert isinstance(a, F2) and isinstance(a, F2)
        return a +b, b

    def test_phase_propre(self):
        Uf = OracleDePhase(self.f)
        e1, e2 =ket(1, 0), ket(1, 0)
        e2 |= bra(1, 0) | (-un)
        assert e1 >>Uf ==e2

    def test_phase_superposition(self):
        Uf = OracleDePhase(self.f)
        e1 = ket(1, 0) >>(H**2)
        e2 = Qudit(
            (un / 2) * Matrice([[1], [-1], [1], [-1]])
        )
        assert e1 >>Uf ==e2

    def test_somme_propre_y1(self):
        Uf = OracleDeSomme(self.f)
        e1 = ket(1, 1, 0)
        e2 = ket(1, 0, 1)
        e3 = ket(1, 0, 0)
        assert e1 >>Uf ==e1
        assert e2 >>Uf ==e3
        assert e3 >>Uf ==e2
```

```
    def test_somme_superposition_y1(self):
        Uf = OracleDeSomme(self.f)
        e0 = ket(0)**3 >>H**3
        assert e0 >>Uf ==e0

    def test_somme_propre_y2(self):
        Uf = OracleDeSomme(self.g, m =2)
        x = ket(1, 1)
        y = ket(1, 0)
        e1 = ket(1, 1, 1, 1)
        e2 = ket(1, 0, 0, 1)
        assert (x @ y) >>Uf ==e1
        assert (y @ x) >>Uf ==e2

    def test_somme_superposition_y2(self):
        # le mme exemple que dans le PDF
        Uf = OracleDeSomme(self.g, m =2)
        x = ket(0, 0) >>H **2
        y = ket(1, 1)
        e0 = x @ y
        e1 = e0 >> Uf
        d = un / 2
        e2 = Qudit(
            Matrice.colonne([0, 0, 0, d, d, 0, 0, 0, 0, d, 0, 0, 0, 0, d, 0]))
        assert e1 ==e2


if __name__ =='__main__':
    main()
```

2.3 Calcul formel

```
from unittest import TestCase, main
from calcul import zero, un, Naturel, Relatif, Rationnel, Puissance, \
    sqrt, Complexe, i, F2, F2Uplet, int_log2, int_vers_bin, bin_vers_int, \
    strbin_vers_int, int_vers_strbin, Matrice, VectPi, pi, Expi, expi, Somme, \
    somme

class TestZero(TestCase):
    def test_sous(self):
        assert zero.sous() ==zero

    def test_sur(self):
        assert zero.sur(Naturel) ==Naturel(0)
        assert zero.sur(Relatif) ==Relatif(0)
        assert zero.sur(Rationnel) ==Rationnel(0, 1)

    def test_plus(self):
        trois =Naturel(3)
        assert zero +zero ==zero
        assert trois +zero ==trois
        assert zero +trois ==trois

    def test_fois(self):
        trois =Naturel(3)
```

```
        moins_trois =Relatif(-3)
        un_tiers =Rationnel(1, 3)
        assert zero *zero ==zero
        assert zero *trois ==zero
        assert trois *zero ==zero
        assert zero *moins_trois ==zero
        assert moins_trois *zero ==zero
        assert zero *un_tiers ==zero
        assert un_tiers *zero ==zero

    def test_signe(self):
        assert zero.signe() ==1

    def test_pow(self):
        assert zero **3 ==zero **un == zero
        assert zero **zero ==un

    def test_arg(self):
        assert zero.arg() ==zero

    def test_conjugue(self):
        assert zero.conjugue() ==zero


class TestNaturels(TestCase):
    def test_sous(self):
        trois =Naturel(3)
        z = Naturel(0)
        assert trois.sous() ==trois
        assert z.sous() ==zero

    def test_sur(self):
        deux =Naturel(2)
        assert deux.sur(Relatif) ==Relatif(2)
        assert deux.sur(Rationnel) ==Rationnel(2, 1)
        assert deux.sur(Puissance) ==Puissance(2, 1)
        assert deux.sur(Complexe) ==Complexe(deux, zero)

    def test_plus(self):
        deux =Naturel(2)
        trois =Naturel(3)
        cinq =Naturel(5)
        assert deux +trois ==cinq
        assert trois +deux ==cinq

    def test_fois(self):
        deux =Naturel(2)
        trois =Naturel(3)
        six =Naturel(6)
        assert deux *trois ==six
        assert trois *deux ==six
        assert deux *trois ==deux *3

    def test_neg(self):
        assert (- un) ==Relatif(-1)
        assert (- Naturel(3)) ==Relatif(-3)

    def test_abs(self):
```

```
        assert abs(Naturel(2)) ==Naturel(2)

    def test_div(self):
        assert Naturel(6) /Naturel(2) ==Naturel(3)
        assert (un*6) /2 ==(un*3)

    def test_mod(self):
        quinze =Naturel(15)
        assert quinze % Naturel(3) ==quinze % 3 ==zero
        assert quinze % Naturel(6) ==quinze % 6 ==Naturel(3)

    def test_floordiv(self):
        quinze =Naturel(15)
        assert quinze //Naturel(3) ==quinze //3 ==Naturel(5)
        assert quinze //Naturel(6) ==quinze //6 ==Naturel(2)

    def test_signe(self):
        assert Naturel(3).signe() ==1

    def test_pow(self):
        deux, trois =Naturel(2), Naturel(3)
        assert deux **3 ==deux **trois ==Naturel(8)
        assert deux **(-3) ==deux **(-trois) ==Rationnel(1, 8)

    def test_arg(self):
        assert Naturel(2).arg() ==zero

    def test_conjugue(self):
        assert Naturel(3).conjugue() ==Naturel(3)


class TestRelatifs(TestCase):
    def test_sous(self):
        moins_trois =Relatif(-3)
        trois =Relatif(3)
        z = Relatif(0)
        assert z.sous() ==zero
        assert trois.sous() ==Naturel(3)
        assert moins_trois.sous() ==moins_trois

    def test_plus(self):
        deux =Relatif(2)
        trois =Relatif(-3)
        moins_un =Relatif(-1)
        assert deux +trois ==moins_un
        assert trois +deux ==moins_un

    def test_fois(self):
        deux =Relatif(2)
        moins_trois =Relatif(-3)
        moins_six =Relatif(-6)
        assert deux *moins_trois ==moins_six
        assert moins_trois *deux ==moins_six
        assert un *(-3) ==moins_trois

    def test_neg(self):
        assert (- Relatif(-2)) ==Naturel(2)
        assert (- Relatif(2)) ==Relatif(-2)
```

```python
    def test_abs(self):
        assert abs(Relatif(-2)) ==Naturel(2)

    def test_div(self):
        assert Relatif(-2) /Relatif(3) ==Rationnel(-2, 3)
        assert Relatif(-2) /Relatif(-3) ==Rationnel(2, 3)

    def test_mod(self):
        moins_quinze =Relatif(-15)
        assert moins_quinze % Relatif(-3) ==moins_quinze % (-3) ==zero
        assert moins_quinze % Relatif(-6) ==moins_quinze % (-6) ==Relatif(-3)

    def test_floordiv(self):
        moins_quinze =Relatif(-15)
        assert moins_quinze //Relatif(-3) ==moins_quinze //(-3) ==Naturel(5)
        assert moins_quinze //Relatif(-6) ==moins_quinze //(-6) ==Naturel(2)

    def test_signe(self):
        assert Relatif(3).signe() ==1
        assert Relatif(-3).signe() ==-1

    def test_pow(self):
        moins_deux, trois =Relatif(-2), Naturel(3)
        assert moins_deux **3 ==moins_deux **trois ==Relatif(-8)
        assert moins_deux **(-3) ==moins_deux **(-trois) ==Rationnel(-1, 8)

    def test_arg(self):
        assert Relatif(2).arg() ==zero
        assert Relatif(-2).arg() ==pi

    def test_conjugue(self):
        assert Relatif(-2).conjugue() ==Relatif(-2)


class TestRationnels(TestCase):
    def test_sous(self):
        assert Rationnel(0, 3).sous() ==zero
        assert Rationnel(6, 2).sous() ==Naturel(3)
        assert Rationnel(-6, 2).sous() ==Relatif(-3)

    def test_eq(self):
        assert Rationnel(2, 6) ==Rationnel(1, 3)
        assert Rationnel(1, -3) ==Rationnel(-1, 3)
        assert Rationnel(-1, -3) ==Rationnel(1, 3)

    def test_neg(self):
        assert (- Rationnel(1, 3)) ==Rationnel(-1, 3)
        assert (- (- Rationnel(1, 3))) ==Rationnel(1, 3)

    def test_plus(self):
        un_demi =Rationnel(1, 2)
        moins_un_tiers =Rationnel(-1, 3)
        un_sixieme =Rationnel(1, 6)
        assert un_demi +un_demi ==un
        assert un_demi +(-un_demi) ==zero
        assert un_demi +moins_un_tiers ==un_sixieme
```

```
    def test_fois(self):
        deux_tiers =Rationnel(2, 3)
        trois_quarts =Rationnel(3, 4)
        un_demi =Rationnel(1, 2)
        assert deux_tiers *trois_quarts ==un_demi
        assert trois_quarts *deux_tiers ==un_demi
        assert Relatif(-2) *un_demi ==Relatif(-1)
        assert un_demi *Relatif(-2) ==Relatif(-1)

    def test_div(self):
        assert un /5 ==Rationnel(1, 5)
        assert Rationnel(1, 3) /Rationnel(4, 3) ==un /4

    def test_abs(self):
        assert abs(Rationnel(1, 2)) ==Rationnel(1, 2)
        assert abs(Rationnel(-1, 2)) ==Rationnel(1, 2)

    def test_signe(self):
        assert (un /2).signe() ==1
        assert Rationnel(-1, 2).signe() ==-1
        assert Rationnel(2, 1).signe() ==1

    def test_arg(self):
        assert (un /2).arg() ==zero
        assert (-un /2).arg() ==pi

    def test_conjugue(self):
        assert (un /2).conjugue() ==(un /2)


class TestPuissance(TestCase):
    def test_sous(self):
        assert Puissance(4, Rationnel(1, 2)).sous() ==Naturel(2)
        assert Puissance(Rationnel(1, 1), Rationnel(1, 1)) ==un

    def test_eq(self):
        x = Puissance(2, Rationnel(2, 3))
        y = Puissance(4, Rationnel(1, 3))
        z = Puissance(un /2, -un /2)
        assert x ==y
        assert z ==sqrt(2)

    def test_sqrt(self):
        assert sqrt(2) ==Puissance(2, Rationnel(1, 2))
        assert sqrt(5) ==Puissance(5, Rationnel(1, 2))
        assert sqrt(4) ==Naturel(2)

    def test_fois(self):
        d = un / 2
        x = Puissance(d, d, -1)
        y = Puissance(d, Rationnel(3, 2))
        z = Puissance(d, -d)
        assert sqrt(2) *sqrt(2) ==Naturel(2)
        assert sqrt(2) *sqrt(d) ==sqrt(d) *sqrt(2) ==un
        assert sqrt(d) *sqrt(d) ==d
        assert un *sqrt(2) ==sqrt(2) *un ==sqrt(2)
        assert Relatif(-1) *sqrt(d) ==sqrt(d) *Relatif(-1) ==x
        assert y *(-2) ==-sqrt(d)
```

```
        print(z *sqrt(8))
        assert z *sqrt(8) ==un *4
        assert sqrt(8) *z ==un *4


    def test_pow(self):
        assert sqrt(2) **2 ==Naturel(2)
        assert sqrt(un /2) **(-2) ==Naturel(2)

    def test_neg(self):
        assert -Puissance(5, Rationnel(1, 2)) ==Puissance(5, Rationnel(1, 2), -1)
        assert -Puissance(5, Rationnel(1, 2), -1) ==Puissance(5, Rationnel(1, 2))

    def test_abs(self):
        assert abs(- sqrt(7)) ==sqrt(7)

    def test_inverse(self):
        assert sqrt(2).inverse() ==sqrt(Rationnel(1, 2))

    def test_signe(self):
        assert sqrt(2).signe() ==1
        assert (-sqrt(2)).signe() ==-1

    def test_arg(self):
        assert sqrt(2).arg() ==zero
        assert (-sqrt(2)).arg() ==pi

    def test_conjugue(self):
        assert sqrt(3).conjugue() ==sqrt(3)


class TestComplexe(TestCase):
    def test_plus(self):
        z1 = Complexe(1, -2)
        z2 = Complexe(-2, 1)
        z3 = Complexe(-1, -1)
        assert z1 +z2 ==z3

    def test_fois(self):
        z1 = Complexe(un, Relatif(-2))
        z2 = Complexe(Relatif(-2), un)
        z3 = Complexe(zero, Naturel(5))
        assert z1 *z2 ==z3
        assert i *(-i) ==un

    def test_neg(self):
        z = Complexe(un, Relatif(-2))
        moins_z =Complexe(Relatif(-1), Naturel(2))
        assert (- z) ==moins_z

    def test_abs(self):
        z = Complexe(un, Relatif(-2))
        assert abs(z) ==sqrt(5)
        assert abs(i) ==un

    def test_sous(self):
        assert Complexe(2).sous() ==Naturel(2)
        assert Complexe(-6).sous() ==Relatif(-6)
```

```
        assert Complexe(un /2).sous() ==(un /2)
        assert Complexe(sqrt(2)).sous() ==sqrt(2)

    def test_sur(self):
        assert (i +1).sur(Expi) ==Expi(pi /4, module=sqrt(2))
        assert ((-i +1) /sqrt(2)).sur(Expi) ==Expi(- pi /4)

    def test_arg(self):
        assert (i *5).arg() ==pi /2
        assert (-i *2).arg() ==-pi /2
        assert (i -1).arg() ==(pi *3) / 4
        assert (-i +1).arg() ==-pi /4
        assert Complexe(-6, 0).arg() ==pi
        assert Complexe(6, 0).arg() ==zero

    def test_conjugue(self):
        z = Complexe(un, Relatif(-2))
        z_barre =Complexe(un, Naturel(2))
        assert z.conjugue() ==z_barre
        assert sqrt(2).sur(Complexe).conjugue() ==sqrt(2)


class TestF2(TestCase):
    def test_sous(self):
        z = F2(0)
        u = F2(1)
        assert z.sous() ==z
        assert u.sous() ==u

    def test_int(self):
        assert int(F2(1)) ==int(F2(5)) ==1
        assert int(F2(0)) ==int(F2(8)) ==0

    def test_calcul(self):
        assert F2(F2(1)) ==F2(1)
        assert F2(0) ==F2(zero) ==F2(Rationnel(8, 4))
        assert F2(1) ==F2(un) ==F2(Rationnel(9, 3))

    def test_sur(self):
        z = F2(0)
        u = F2(1)
        assert z.sur(Naturel) ==Naturel(0)
        assert u.sur(Naturel) ==un

    def test_plus(self):
        z = F2(0)
        u = F2(1)
        assert z +z == u + u == z
        assert z +u == u + z == u
        assert z +0 == u + 1 == z
        assert z +1 == u + 0 == u

    def test_fois(self):
        z = F2(0)
        u = F2(1)
        assert z *z == z
        assert z *u == z
        assert u *z == z
```

```
        assert u *u == u

    def test_int_vers_bin(self):
        assert int_vers_bin(11) ==[1, 0, 1, 1]
        assert int_vers_bin(0) ==[0]
        assert int_vers_bin(2, taille =4) ==[0, 0, 1, 0]

    def test_bin_vers_int(self):
        assert bin_vers_int(1, 0, 1, 1) ==11
        assert bin_vers_int() ==bin_vers_int(0) ==0

    def test_int_log2(self):
        assert int_log2(3) ==2
        assert int_log2(4) ==int_log2(5) ==3

    def test_int_vers_strbin(self):
        assert int_vers_strbin(11) ==’1011’
        assert int_vers_strbin(0) ==’0’
        assert int_vers_strbin(2, taille =4) ==’0010’

    def strbin_vers_int(self):
        assert strbin_vers_int(’1011’) ==11
        assert strbin_vers_int(’0010’) ==2

    def test_conjugue(self):
        assert F2(0).conjugue() ==F2(0)
        assert F2(1).conjugue() ==F2(1)


class TestF2Uplet(TestCase):
    def test_creation(self):
        assert F2.uplet(1) ==F2Uplet(1)

    def test_eq(self):
        t1 = F2Uplet(un, un, zero)
        t2 = F2Uplet(1, 1, 0)
        assert t1 ==t2

    def test_plus(self):
        t1 = F2Uplet(1, 0, 0)
        t2 = F2Uplet(1, 1, 0)
        t3 = F2Uplet(0, 1, 0)
        assert t1 +t2 ==t3


class TestMatrice(TestCase):
    def test_zero(self):
        m = Matrice.zeros(2, 3)
        for k in range(2):
            for j in range(3):
                assert m[k, j] ==zero

    def test_setitem(self):
        m = Matrice.zeros(2)
        assert m[0, 1] ==zero
        m[0, 1] =un
        assert m[0, 1] ==un
        m[0, 1] =2
```

```
        assert m[0, 1] ==Naturel(2)

    def test_tableau(self):
        m1 = Matrice([[un, zero], [un, zero]])
        m2 = Matrice([[1, zero], [1, 0]])
        assert m1[0, 0] ==un
        assert m1[0, 1] ==zero
        assert m1[1, 0] ==un
        assert m1[1, 1] ==zero
        assert m1 ==m2

    def test_eq(self):
        m1 = Matrice([[1, 0], [0, 1]])
        m2 = Matrice([[1, 0], [0, 1]])
        assert m1 ==m2
        assert m1 !=Matrice.zeros(2)
        assert m1 !=Matrice.zeros(2, 3)

    def test_fois(self):
        m1 = Matrice([[1, 2], [3, 4]])
        m2 = Matrice([[4, 5], [6, 7]])
        m3 = Matrice([[16, 19], [36, 43]])
        m4 = Matrice([[19, 28], [27, 40]])
        assert m1 *m2 ==m3
        assert m2 *m1 == m4
        m5 = Matrice([[5], [6]])
        m6 = Matrice([[17], [39]])
        assert m1 *m5 ==m6

    def test_fois_scalaire(self):
        m1 = Matrice([[2, 4], [6, 8]])
        m2 = Matrice([[1, 2], [3, 4]])
        assert Rationnel(1, 2) *m1 ==m1 *Rationnel(1, 2) ==m2
        assert m2 *Naturel(2) ==m2 *2 == m1

    def test_acces_rapide(self):
        l = Matrice([[1, 2]])
        c = Matrice([[1], [2]])
        assert l[1] ==Naturel(2)
        assert c[1] ==Naturel(2)
        l[1] =zero
        c[1] =un
        assert l[1] ==zero
        assert c[1] ==un

    def test_produit_tensoriel(self):
        m1 = Matrice([[1, 2], [3, 4]])
        m2 = Matrice([[5, 6], [7, 8]])
        m3 = Matrice([
            [5, 6, 10, 12],
            [7, 8, 14, 16],
            [15, 18, 20, 24],
            [21, 24, 28, 32]
        ])
        assert m1 @ m2 ==m3
        m4 = sqrt(Rationnel(1, 2)) *Matrice([[1, 1], [1, -1]])
        m5 = Rationnel(1, 2) *Matrice([
            [1,  1,  1,  1],
```

```
            [1, -1, 1, -1],
            [1,  1, -1, -1],
            [1, -1, -1, 1]
        ])
        assert m4 @ m4 ==m5

    def test_identite(self):
        m = Matrice.identite(3)
        for k in range(3):
            for j in range(3):
                if k ==j:
                    assert m[k, j] ==un
                else:
                    assert m[k, j] ==zero

    def test_scalaire(self):
        assert Matrice.scalaire(un, 3) ==Matrice.identite(3)
        assert Matrice.scalaire(-2, 3) ==Matrice.identite(3) *(-2)

    def test_ligne(self):
        m = Matrice([[1, 2, 3]])
        assert Matrice.ligne(1, 2, 3) ==m
        assert Matrice.ligne([1, 2, 3]) ==m

    def test_colonne(self):
        m = Matrice([[1], [2], [3]])
        assert Matrice.colonne(1, 2, 3) ==m
        assert Matrice.colonne([1, 2, 3]) ==m


class TestVectPi(TestCase):
    def test_pi(self):
        assert pi ==VectPi(1)
        assert pi ==VectPi(un)

    def test_sous(self):
        assert VectPi(0).sous() ==zero
        assert pi.sous() ==pi
        assert VectPi(sqrt(2)) ==VectPi(sqrt(2))

    def test_add(self):
        pi_sur_trois =VectPi(Rationnel(1, 3))
        pi_sur_quatre =VectPi(Rationnel(1, 4))
        pi_sur_six =VectPi(Rationnel(1, 12))
        assert pi -pi ==zero
        assert pi +pi ==VectPi(2)
        assert pi_sur_trois -pi_sur_quatre ==pi_sur_six
        assert pi +zero ==zero +pi == pi

    def test_mul(self):
        deux_pi =VectPi(2)
        assert zero *pi ==pi * zero ==zero
        assert un *pi ==pi * un == pi
        assert pi *2 ==pi * Naturel(2) ==deux_pi

    def test_div(self):
        assert pi /2 ==VectPi(Rationnel(1, 2))
        assert pi *3 / 2 == VectPi(Rationnel(3, 2))
```

```
    def test_neg(self):
        assert (- VectPi(2)) ==VectPi(-2)

    def test_mod2pi(self):
        assert (pi *3).mod2pi() ==(pi *-3).mod2pi() ==pi
        assert (pi *2).mod2pi() ==(pi *-4).mod2pi() ==zero


class TestExpi(TestCase):
    def test_eq(self):
        assert expi(-pi /2) ==expi(pi *3 /2)
        assert expi(-pi *2 /3) ==expi(pi *4 / 3)
        assert expi(pi /2) ==i

    def test_type(self):
        assert expi(pi) ==(-un)
        assert expi(pi /2) ==i
        assert expi(pi /3) ==Expi(pi /3)
        assert expi(pi /4) ==Expi(pi /4)

    def test_exp0(self):
        assert expi(zero) ==expi(0) ==un
        assert Expi(zero) ==Expi(0) !=un

    def test_arg(self):
        e3i = expi(3)
        pi_sur_4 =VectPi(Rationnel(1, 4))
        eipi_sur_4 =expi(pi_sur_4)
        assert e3i.arg() ==Naturel(3)
        assert eipi_sur_4.arg() ==pi_sur_4

    def test_conjugue(self):
        assert expi(3).conjugue() ==expi(-3)
        assert expi(pi *-3).conjugue() ==expi(pi *3)

    def test_abs(self): # le module
        z1 = Expi(1, module =4)
        z2 = expi(3)
        assert abs(z1) ==Naturel(4)
        assert abs(z2) ==un

    def test_neg(self):
        assert -expi(pi /3) ==expi(pi *4 /3)

    def test_inverse(self):
        z1 = Expi(pi /5, module=(un /2))
        z2 = Expi(-pi /5, module=2)
        assert z1.inverse() ==z2
        assert z2.inverse() ==z1
        assert un /expi(pi /3) ==expi(-pi /3)

    def test_mul(self):
        z1 = expi(pi /2)
        assert z1 *z1 ==(-un)
        assert z1 *3 ==i * 3
        assert expi(pi /3) *i ==expi(pi *5 / 6)
        z2 = expi(pi /4) / 16
```

```
        z3 = i / 16
        assert z2 *z3 ==expi(pi *3 / 4) / 256
        assert Expi(pi /4, module=sqrt(2)) ==expi(pi /4) *sqrt(2)

    def test_add(self):
        assert expi(pi /3) +zero ==zero + expi(pi /3) == expi(pi /3)
        assert expi(pi /3) +expi(pi /3) ==expi(pi /3) * 2
        assert expi(pi /3) -expi(pi /3) ==zero
        assert -expi(pi /3) +expi(pi /3) ==zero
        assert (expi(-pi /4) *sqrt(2) +i) ==un

    def test_pow(self):
        z1 = expi(pi /6)
        z2 = expi(pi /2)
        assert z1 **3 ==z1 ** Naturel(3) ==z2

    def test_sous(self):
        assert Expi(pi *2).sous() ==Expi(0).sous() ==un
        assert Expi(pi).sous() ==Expi(pi *3).sous() ==-un
        assert Expi(pi /2).sous() ==Expi(pi *(-3) /2).sous() ==i
        assert Expi(- pi /2).sous() ==Expi(pi *3 /2).sous() ==-i

    def test_sur(self):
        assert Expi(pi /4).sur(Complexe) ==sqrt(un /2) *(i +1)


class TestSomme:
    def test_somme(self):
        assert somme(1, 2, 3) ==Naturel(6)
        assert Somme(1, 2, 3).sous() ==Naturel(6)
        assert Somme(1, 2, 3) !=Naturel(6)

    def test_eq(self):
        assert Somme(1, 2, 3) ==Somme(2, 3, 1)
        assert Somme(1, 2, 3) ==Somme(6)
        assert Somme(1, 2, 3) !=Naturel(6)
        assert Somme(1, sqrt(2)) ==Somme(sqrt(2), 1)

    def test_sous(self):
        assert Somme(sqrt(2)).sous() ==sqrt(2)
        assert Somme(sqrt(2), zero).sous() ==sqrt(2)
        assert Somme(1, 2).sous() ==Naturel(3)

    def test_plus_nombre(self):
        x = somme(1, sqrt(2))
        y = somme(Rationnel(3, 2), sqrt(2))
        assert x +Rationnel(1, 2) ==y

    def test_plus_somme(self):
        x = somme(1, sqrt(2))
        y = somme(sqrt(3), 4)
        z = somme(sqrt(2), sqrt(3), 5)
        assert x +y == z

    def test_creation(self):
        assert sqrt(2) +1 ==Somme(sqrt(2), 1)
        assert sqrt(2) +1 +2 == Somme(sqrt(2), 3)
```

```
    def test_mul(self):
        assert sqrt(2) *(sqrt(2) +1) ==sqrt(2) +2
        a = sqrt(2) +1
        b = sqrt(2) -1
        assert a *b == un
        assert a *a == sqrt(8) +3

    def test_zeros(self):
        s = sqrt(2) +1
        assert s -1 == sqrt(2)

    def test_conjugue(self):
        s1 = i * sqrt(2) +1
        s2 = -i * sqrt(2) +1
        assert s1.conjugue() ==s2
        assert s2.conjugue() ==s1

    def test_abs(self):
        s1 = sqrt(2) +1
        assert abs(s1) ==s1
        assert abs(-s1) ==s1
        s2 = Somme(expi(pi /4), expi(-pi /4))
        print(abs(s2), type(abs(s2)))
        assert abs(s2) ==sqrt(2)


if __name__ ==’__main__’:
    main()
```

2.4 Fonctions utiles

```
from unittest import TestCase, main

from calcul import un, sqrt, Matrice
from qubit import ket
from portes import H, I
from fonctions_utiles import H_option, ket_vers_liste, etat_de_base, \
    sequence_egale, qubits_intriques, kron_id_mat, kron_id


class TestFonctions(TestCase):
    def test_sequence_egale(self):
        e0 = ket(1, 1, 0)
        e1 = ket(1, 0, 0)
        e2 = ket(0, 1, 1, 0)
        assert sequence_egale([1, 1, 0], e0)
        assert not sequence_egale([1, 1, 0], e1)
        assert not sequence_egale([1, 1, 0], e2)

    def test_etat_de_base(self):
        e0 = etat_de_base(2, 1)
        e1 = etat_de_base(2, 1, 0)
        e2 = etat_de_base(2, 1, 1)
        assert e0 ==e1 ==ket(0, 0, 0)
        assert e2 ==ket(0, 0, 1)
```

```
    def test_ket_vers_liste(self):
        l1, l2 =[1, 0, 1, 1], [0, 1]
        assert ket_vers_liste(ket(*l1)) ==l1
        assert ket_vers_liste(ket(*l2)) ==l2

    def test_H_option(self):
        B = I @ H @ (I **2)
        assert H_option(4, debut=1, fin=-2) ==B
        assert H_option(4, debut=1, fin=2) ==B
        assert H_option(3 *2, debut=0, fin=3) ==kron_id(H**3, 3)
        assert H_option(8, debut=0, fin=4) ==kron_id(H **4, 4)

    def test_qubits_intriques(self):
        assert qubits_intriques(2) ==ket(0, 0) >>H**2
        assert qubits_intriques(2, valeur=1) ==ket(1, 1) >>H**2

    def test_kron_id_mat(self):
        H = sqrt(un /2) * Matrice([[1, 1], [1, -1]])
        M = H ** 2
        assert M @ (Matrice.identite(2**2)) ==kron_id_mat(M, 2**2)

    def test_kron_id(self):
        M = H ** 2
        assert M @ (I **2) ==kron_id(M, 2)


if __name__ ==’__main__’:
    main()
```

2.5 Analyses d’efficacité

```
import sys
from timeit import timeit
import matplotlib.pyplot as plt

from portes import H, I, QFT
from fonctions_utiles import H_option, kron_id

def temps(entiers, fonction, N =12):
    t = []
    resultats =[]
    for i in entiers:
        rep = 100 if not i else (N //i)
        print(f’Calcul {rep} fois pour n = {i} ...’)
        fait =False
        def f():
            nonlocal fait
            if not fait: resultats.append(fonction(i))
            else: fonction(i)
            fait =True
        t.append(timeit(f, number=rep) /rep)
    return t, resultats

def memoire(resultats):
    return [taille(i) for i in resultats]
```

```
def perf_pow(M):
    n = [1, 2, 3, 4, 5, 6, 7, 8]
    t, r =temps(n, lambda i: M **i, N=24)
    return n, t, memoire(r)

def perf_H_option_moins_un():
    n = [1, 2, 3, 4, 5, 6, 7, 9]
    t, r =temps(n, lambda i: H_option(i, debut=0, fin=-1))
    return n, t, memoire(r)

def perf_H_option_moitie():
    n = [1, 2, 3, 4]
    t, r =temps(n, lambda i: H_option(2*i, debut=0, fin=i))
    return n, t, memoire(r)

def perf_qft():
    n = [2, 4, 8, 16]
    t, r =temps(n, lambda i: QFT(i), N=24)
    return n, t, memoire(r)

def perf_shor():
    m = [1, 2, 3, 4, 5]
    t, r =temps(m, lambda i: QFT(2**i) @ (I **i), N=24)
    return m, t, memoire(r)

def compare_kron_id():
    m = [1, 2, 3, 4, 5]
    M = QFT(16)
    print('Test 1 (classique)')
    t1, _ =temps(m, lambda i: M @ (I **i))
    print('Test 2 (kron_id)')
    t2, _ =temps(m, lambda i: kron_id(M, i))
    return m, t1, t2

def aff_temps_memoire(n, t, m):
    fig, ax1 =plt.subplots()
    ax1.set_xlabel('Entier')

    ax1.plot(n, t, '+', label='Temps (s)', color='orange')
    ax2 = ax1.twinx()
    ax2.plot(n, m, '+', label='Memoire', color='green')

    fig.legend(loc='upper left', bbox_to_anchor=(0,1), bbox_transform=ax1.transAxes)
    plt.show()

def aff_temps(n, t):
    fig = plt.figure()
    fig.plot(n, t, '+', label='Temps', color='purple')
    fig.xlabel('Entier')
    fig.ylabel('Temps (s)')
    fig.legend()
    plt.show()

def aff_temps1_temps2(n, t1, t2):
    plt.plot(n, t1, '+', label='Temps 1', color='red')
    plt.plot(n, t2, '+', label='Temps 2', color='blue')
    plt.xlabel('Entier')
    plt.ylabel('Temps (s)')
```

```
    plt.legend()
    plt.show()

def temps_fmt(t):
    if 1e-6 <t < 1e-3:
        return f'{(t * 1e6):.3g} s'
    if 1e-3 <t < 1:
        return f'{(t * 1e3):.3g} ms'
    return f'{t:.3g} s'

def nb_fmt(x):
    if isinstance(x, float):
        return f'{x:.2e}'
    return str(x)

def print_donnees(abscisse, *ordonnees):
    ligne ='{:>12}' *len(abscisse)
    print(ligne.format(*abscisse))
    l = zip(*ordonnees)
    for row in l:
        print(ligne.format(*[nb_fmt(i) for i in row]))

def print_ntm(n, t, m):
    print_donnees(
        ['Entier', 'Temps', 'Memoire'], n,
        [temps_fmt(i) for i in t], m)

def print_nt(n, t):
    print_donnees(['Entier', 'Temps'], n, [temps_fmt(i) for i in t])

def print_nt1t2(n, t1, t2):
    N = len(t1)
    print_donnees(
        ['Entier', 'Temps 1', 'Temps 2', 'Rapport'], n,
        [temps_fmt(i) for i in t1], [temps_fmt(i) for i in t2],
        [f'{(t1[i]/t2[i]):.3g}' for i in range(N)])
    r_moy =sum(t1[i] /t2[i] for i in range(N)) /N
    print(f'Rapport moyen : {r_moy:.3g}')

def taille(obj):
    if isinstance(obj, list) or isinstance(obj, tuple):
        return sys.getsizeof([]) +sum([taille(i) for i in obj])
    if isinstance(obj, int) or isinstance(obj, bool):
        return sys.getsizeof(obj)
    vus = []
    def aux(o):
        if o in vus: return 0
        vus.append(o)
        s = sys.getsizeof(o)
        return s +sum([taille(i) for i in obj.__dict__.values()])
    return aux(obj)

if __name__ =='__main__':
    n, t1, t2 =compare_kron_id()
    print_nt1t2(n, t1, t2)
    aff_temps1_temps2(n, t1, t2)
```

3 Algorithmes

3.1 Deutsch-Jozsa et Bernstein-Vazirani

```
from calcul import un, zero
from fonctions_utiles import qubits_intriques
from portes import H
from oracle import Oracle
from qubit import bra

def dj(f, n):
    q = qubits_intriques(n)
    U = Oracle.phase(f)
    C = q >> U >> (H**n)
    return C

def est_constante(f, n):
    q = dj(f, n)
    test =bra(*([0]*n)) | q
    return (test ==un or test ==-un)

def point(x_list, a_list):
    n = len(x_list)
    d = zero
    for i in range(n):
        d += (x_list[i]*a_list[i])
    return d

def bv(a):
    return dj(lambda *args: point(args, a), len(a))
```

3.2 Grover

```
from math import asin, pi, sqrt
import matplotlib.pyplot as plt

from calcul import F2, int_log2, int_vers_bin
from fonctions_utiles import etat_de_base, H_option
from portes import H, I, PhaseCond
from oracle import Oracle

def main():
    ef = grover(*indicatrice(0, 1, 0, 1, 1, 0))
    print('L\'etat de sortie est :', ef)
    print('Une solution est', solution(ef))
    affiche_amplitudes(ef)

def grover(f, n, M =1):
    theta0 =asin(sqrt(M /(2 **n)))
    rep = int(pi /(4 * theta0))
    H_op =H_option(n, debut=0, fin=-1)
    Uf = Oracle.phase(f)
    e = etat_de_base(n-1, 1, 1) >>(H**n)
    B = H_op >>(PhaseCond(n-1) @ I) >>H_op
    for i in range(rep):
```

```
        e = e >> B >> Uf
    return e

def solution(ef):
    s, fm =None, None
    for i in range(ef.dim):
        f = float(ef[i].abs_carre())
        if s is None or fm <f:
            s, fm =i, f
    return tuple(int_vers_bin(s, taille=int_log2(ef.dim)-1))

def affiche_amplitudes(ef):
    l = [float(ef[i].abs_carre()) for i in range(ef.dim)]
    x = list(range(ef.dim))
    plt.bar(x, l)
    plt.xlabel('etat propre')
    plt.ylabel('Probabilite (module au carre de l\'amplitude)')
    plt.show()

def indicatrice(*solution):
    def f(*valeurs):
        if valeurs ==tuple([F2(i) for i in solution]):
            return F2(1)
        return F2(0)
    return f, len(solution)


if __name__ =='__main__':
    main()
```

3.3 Shor

```
from random import randint
from time import time
import matplotlib.pyplot as plt

from calcul import zero, pgcd, bin_vers_int, int_vers_bin, Naturel
from portes import QFT
from oracle import Oracle
from fonctions_utiles import etat_de_base, H_option, kron_id

m = 4

def main():
    N = int(input('Entrez un nombre : '))
    t0 = time()
    f = facteurs(N)
    print('-' *20)
    print(f"{N} = {'  '.join([str(i) for i in f])}")
    print(f'(calcul effectue en {(time() - t0):.2f} s)')

def facteurs(N):
    n, p =deux_facteurs_shor(N)
    if n ==1:
        print(f'| {N} est premier')
        return [N]
```

```
    return facteurs(n) +facteurs(p)

def deux_facteurs_shor(N):
    if N ==2:
        return 1, 2
    print('Decomposition en deux facteurs de', N)
    deja_vus =[]
    while True:
        a = randint(2, N-1)
        if a in deja_vus:
            continue
        else:
            deja_vus.append(a)
        d = pgcd(a, N)
        if d !=1:
            print(f'Coup de bol ! {d} divise le NPA {a} et {N}')
            n, p =d, N //d
            return min(n, p), max(n, p)
        print(f'Recherche de periode [NPA={a}, N={N}] ...')
        r = periode(a, N)
        print(f'\tLa periode obtenue avec le NPA {a} est {r}')
        if r % 2 ==0 and a**(r//2) % N !=N-1:
            n = pgcd(a **(r // 2) + 1, N)
            p = pgcd(a **(r // 2) - 1, N)
            if n *p == N:
                print(f'C\'est une periode valide, {N} = {n}  {p}')
                return min(n, p), max(n, p)
        print('Periode invalide, on recommence')

def periode(a, N):
    ef = recherche_periode(a, N)
    demander_affichage(ef)
    for i in range(ef.dim):
        x = ef[i].abs_carre()
        if x !=zero:
            inv =x.inverse()
            if not inv.appartient(Naturel):
                raise ValueError(f'la periode {inv}, de type {type(inv)} n\'est pas un naturel')
            return int(inv)
    raise ValueError('etat final nul')

def cree_f(a, N):
    def f(*bits):
        x = bin_vers_int(*bits)
        return tuple(int_vers_bin((a **x) % N, taille=m))
    return f

def recherche_periode(a, N):
    U = Oracle.somme(cree_f(a, N), m =m)
    print('\tCreation de l\'etat initial ...')
    e0 = etat_de_base(m, m , 0)
    print('\tIntrication des etats ...')
    e1 = e0 >>H_option(2*m, debut=0, fin=m)
    print('\tPassage dans l\'oracle ...')
    e2 = e1 >>U
    print('\tCreation du circuit QFT ...')
    Q = QFT(2**m).dague()
    P = kron_id(Q, m)
```

```
    print('\tPassage dans le circuit QFT ...')
    e3 = e2 >> P
    return e3

def demander_affichage(ef):
    if input('\tAfficher l\'etat final ? [y/n] ') =='y':
        print('\tL\'etat final est :', ef)
    if input('\tAfficher les amplitudes ? [y/n] ') =='y':
        affiche_amplitudes(ef)

def affiche_amplitudes(ef):
    l = [float(ef[i].abs_carre()) for i in range(ef.dim)]
    x = list(range(ef.dim))
    plt.bar(x, l)
    plt.xlabel('etat propre')
    plt.ylabel('Amplitude (module au carre)')
    plt.show()

if __name__ =='__main__':
    main()
```

3.4 Parité

```
from portes import R, X
from oracle import Oracle
from qubit import bra, ket


def f(q):
    return (bra(1) | q).arg() % 2

Uf = Oracle.brut(f)

def etat_sortie(n):
    return ket(0) >>X >>R(n) >>Uf

def est_pair(n):
    return etat_sortie(n) ==ket(0)

def main():
    n = int(input('Entrez un entier : '))
    S = etat_sortie(n)

    print('L\'etat de sortie est ' +str(S) +'.')
    parite ='pair' if S ==ket(0) else 'impair'
    print(f'Donc {n} est {parite} !')

if __name__ =='__main__':
    main()
```

```
from portes import cX
from qubit import ket
from calcul import int_vers_bin
from fonctions_utiles import sequence_egale
```

```
def est_pair(n):
    psi =ket(int_vers_bin(n)[-1])
    s = (psi @ ket(1)) >>cX
    return sequence_egale([0, 1], s)


def affiche_parite(n):
    if est_pair(n):
        print("Le nombre pris en entree est pair.")
    else:
        print("Le nombre pris en entree est impair.")

if __name__ =='__main__':
    n = int(input("Entrez un nombre entier : "))
    affiche_parite(n)
```

3.5 Tests d'algorithmes

```
from unittest import TestCase, main
from calcul import zero, un

from dj import est_constante, bv
from fonctions_utiles import ket_vers_liste
import pair_impair, parite
from grover import indicatrice, grover, solution


class TestDJ(TestCase):
    def test_constantes(self):
        f = lambda a, b, c, d: zero
        g = lambda a, b, c, d: un
        assert est_constante(f, 4)
        assert est_constante(g, 4)

    def test_equilibrees(self):
        f = lambda a, b, c, d: a
        g = lambda a, b, c, d: a +b
        assert not est_constante(f, 4)
        assert not est_constante(g, 4)


class TestBV(TestCase):
    def test_bv(self):
        a = [zero, un, un]
        b = [zero, zero, un, un]
        assert ket_vers_liste(bv(a)) ==[int(i) for i in a]
        assert ket_vers_liste(bv(b)) ==[int(i) for i in b]


class TestPairImpair(TestCase):
    def test_pair_impair(self):
        for n in list(range(15)):
            assert pair_impair.est_pair(n) ==(n % 2 ==0)

    def test_parite(self):
```

```
        for n in list(range(15)):
            assert parite.est_pair(n) ==(n % 2 ==0)


class TestGrover:
    def test_grover(self):
        l = ((0, 1, 1), (0, 0))
        for i in l:
            a = grover(*indicatrice(*i))
            assert solution(a) ==i


if __name__ =='__main__':
    main()
```

4 Polarisation

```
import random
import matplotlib.animation as animation
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import numpy as np

fig, ax =plt.subplots()
plt.axis('square')

N_item =5 #nombre de photons a animer

photons ={}
for i in range(N_item):
    photons[i], =ax.plot([], [], ls ='none', marker ='o', color ='purple')

plt.xlim(-10,10)
plt.ylim(-10.5,10.5)

N = 300 #resulution de l'echantillonage de (Ox)
disp =20 #coeff d'ecartement
base =np.linspace(-10, 10, N)
x_pol_1 =-2
x_pol_2 =4
X = {}
for i in range(N_item):
    X[i] =np.concatenate((np.full(disp*i, -10), np.linspace(-10, 10, N), np.full(N, 10)), axis =None)
color =['blue', 'green']

plt.plot([x_pol_1,x_pol_1], [10.5,-10.5], color ='black')
plt.plot([x_pol_2,x_pol_2], [10.5,-10.5], color ='black')

ax.set_xticklabels([])
ax.set_yticklabels([])

plt.tick_params(axis ='x', length =0)
plt.tick_params(axis ='y', length =0)

key = ['0'] *N_item
```

```python
def list_to_str(l):
    ret = ""
    for e in l:
        ret = ret + e
    return ret

text = {}
text[0] =ax.text(4, 4, list_to_str(key), fontsize =15, fontweight ='bold', color ='black')
ax.add_patch(Rectangle((8, -1), 2, 2))

def ret_tuple(dico):
    return tuple(dico[c] for c in dico)

def animate(i):
    for c in photons:
        photons[c].set_data(X[c][i], 0)

        if abs(photons[c].get_xdata() +10) <=10**-1:
                photons[c].set_color('purple')

        if abs(photons[c].get_xdata() -x_pol_1) <=10**-1:
                photons[c].set_color(random.choice(color))

        if abs(photons[c].get_xdata() -x_pol_2) <=10**-1 and photons[c].get_color() =='green':
            photons[c].set_alpha(0.0)

        if abs(photons[c].get_xdata() -8) <=10**-1 and photons[c].get_alpha() !=0.0:
            key[c] ='1'
            text[0].set_text(list_to_str(key))

    return ret_tuple(photons) +tuple(text[c] for c in text)

ani = animation.FuncAnimation(fig, animate, frames=range(2*N), blit =True, interval =5, repeat =False
    )
plt.show()
```

5 Interface graphique

```html
<!DOCTYPE html>
<html>
  <head>
    <!-- <script defer src="https://pyscript.net/alpha/pyscript.js"></script> -->
    <title>Ordinateur Quantique</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="stylesheet.css">
    <script src="script.js"></script>
    <script src="https://polyfill.io/v3/polyfill.min.js?features=es6"></script>
    <script id="MathJax-script" async src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-
        mml-chtml.js"></script>
  </head>
  <body>
    <div class="title_wrapper">
```

```
  <h1 class="title">Ordinateur Quantique</h1>
  <button type="button" class="run_button" onclick="runCircuit()"><b>Excuter</b></
      button>
</div>
<br>
<div class="side_pane">
  <div class="dimensions_form">
    <h3>Dimensions du circuit</h3>
    <div>
      Nombre de qubits :
      <input type="number" id="qubits_input" min="0" step="1" onclick="updateGridSize()"
           placeholder="ex. 3">
    </div>
    <br/>
    <div>
      Nombre d'tapes :
      <input type="number" id="steps_input" min="0" step="1" onclick="updateGridSize()"
          placeholder="ex. 2">
    </div>
  </div>
  <h3>Portes</h3>
  <div id="container_portes" class="container_portes">
    <div id="H" class="porte" draggable="true" ondragstart="drag(event)"> H </div>
    <div id="U" class="porte" draggable="true" ondragstart="drag(event)"> U </div>
    <div id="cX" class="porte" draggable="true" ondragstart="drag(event)"> cX </div>
    <div id="I" class="porte" draggable="true" ondragstart="drag(event)"> I </div>
    <div id="S" class="porte" draggable="true" ondragstart="drag(event)"> S </div>
    <div id="X" class="porte" draggable="true" ondragstart="drag(event)"> X </div>
  </div>
  <br>
  <div class="oracle_form">
    <fieldset>
      <legend>Oracle</legend>
      <div>
        <input type="radio" name="oracle_type" value="sum" checked>
        <label for="sum">Somme $$U_f \ket{x, y} = \ket{x, y \oplus f(y)}$$</label>
      </div>
      <div>
        <input type="radio" name="oracle_type" value="phase">
        <label for="phase">Phase $$U_f \ket{x} = (-1)^{f(x)} \ket{x}$$</label>
      </div>
      <div class="function_editor">
        <u> diteur  de fonction</u><br>
        \(f(x) = \) <input type="text" id="function_input">
      </div>
  </fieldset>
  </div>
</div>

<div id="main" class="main_container">
```

```
        <div id="container_circuit" class="container_circuit">
          <div id="notthisone" class="sub_container_portes"></div>
        </div>
    </div>
  </body>
</html>

let circuit = [];

let gatesQubits = {
    ’’: 1,
    ’|0  ’: 1,
    ’|1  ’: 1,
    ’H’: 1,
    ’U’: NaN,
    ’cX’: 2,
    ’I’: 1,
    ’S’: 2,
    ’X’: 1
};

function drag(ev) {
    ev.dataTransfer.setData(’text’, ev.target.id);
}

function allowDrop(ev) {
    ev.preventDefault();
}

function drop(ev) {
    ev.preventDefault();
    let dest = ev.target;
    let name = ev.dataTransfer.getData(’text’, ev.innerText);
    let x = cellRow(dest);
    let y = cellColumn(dest);
    console.log(’Dropping’, name, ’at’, x, y);
    if (isNaN(gatesQubits[name])) {
        circuit[0][x] = name;
        for (let i = 1; i < circuitQubits(); i++) {
            circuit[i][x] = null;
        }
    } else {
        circuit[y][x] = name;
        if (gatesQubits[name] == 2) {
            circuit[y+1][x] = null;
        }
    }
    updateGrid();
}
```

```
function clearGrid() {
    let grid = document.getElementById(’container_circuit’);
    var child = grid.lastElementChild;

    while (child) {
        grid.removeChild(child);
        child = grid.lastElementChild;
    }
}

function ketToString(n) {
    if (n == 0) {
        return ’|0  ’;
    }
    return ’|1  ’;
}

function setStates(qubits) {
    for (let i = 0; i < qubits; i++) {
        let cell = document.getElementById(idRowColumn(0, i));
        cell.innerText = ketToString(0);
        circuit[i][0] = ketToString(0);
        cell.setAttribute(’onclick’, ’switchState(event.target)’);
        cell.setAttribute(’ondrop’, ’’);
        cell.setAttribute(’ondragover’, ’’);
        // cell.setAttribute(’’)
    }
}

function switchState(cell) {
    let x = cellRow(cell);
    let y = cellColumn(cell);
    let s = (cell.innerText == ketToString(1)) ? ketToString(0) : ketToString(1);
    cell.innerText = s;
    circuit[y][x] = s;
}


function idRowColumn(i, j) {
    return ‘r${i}_${j}‘
}

function circuitQubits() {
    return circuit.length;
}

function circuitSteps() {
    return circuit[0].length;
}
```

```
function gateAspectRatio(gateName) {
    let q = gatesQubits[gateName];
    if (q == 2) { return '0.5'; }
    if (isNaN(q)) { return '0.25'; }
}

function gateHeight(gateName) {
    let q = gatesQubits[gateName];
    if (q == 1) { return '50%'; }
    if (q == 2) { return '75%'; }
    if (isNaN(q)) { return '85%'; }
}

function gateSpan(gateName) {
    let q = gatesQubits[gateName];
    if (isNaN(q)) { return circuitQubits().toString() }
    return q.toString();
}

function gateInnerHTML(gateName) {
    if (gateName == 'U') {
        return '<i>U<sub>f</sub></i>'
    }
    return `<i>${gateName}</i>`
}

function cellRow(element) {
    return parseInt(element.id.slice(1).split('_')[0], 10);
}

function cellColumn(element) {
    return parseInt(element.id.slice(1).split('_')[1], 10);
}

function updateGridSize() {
    let qubits = parseInt(document.getElementById('qubits_input').value);
    let steps = 1 + parseInt(document.getElementById('steps_input').value);
    if (isNaN(qubits) || isNaN(steps)) {
        return;
    }
    console.log(`Side updated: ${steps} steps, ${qubits} qubits`);

    circuit = Array(qubits);
    for (let i = 0; i < qubits; i++) {
        circuit[i] = Array(steps).fill('');
    }
    updateGrid();
}

function createCell(i, j, name = '') {
```

```
    let el = document.createElement('div');
    el.style.gridColumn = `${i+1} / ${i+2}`;
    el.style.gridRow = `${j+1} / span ${gateSpan(name)}`;
    el.style.aspectRatio = gateAspectRatio(name);
    el.style.height = gateHeight(name);
    el.setAttribute('class', 'sub_container_portes');
    el.setAttribute('id', idRowColumn(i, j));
    el.setAttribute('ondrop', 'drop(event)');
    el.setAttribute('ondragover', 'allowDrop(event)');
    el.style.zIndex = '10';
    return el;
}

function updateGrid() {
    let rows = circuitSteps();
    let columns = circuitQubits();
    let grid = document.getElementById('container_circuit');
    grid.style.gridTemplateColumns = 'auto '.repeat(rows);
    grid.style.gridTemplateRows = 'auto '.repeat(columns);
    clearGrid();

    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < columns; j++) {
            let name = circuit[j][i];
            if (name != null) {
                let cell = createCell(i, j, name);
                cell.innerHTML = gateInnerHTML(name);
                grid.appendChild(cell);
            }
        }
    }
    setStates(columns);
}

function runCircuit() {
    console.log('Running ...')
}

body, input, button {
    font-family: CMU Serif, Verdana, Geneva, Tahoma, sans-serif;
}

.title_wrapper {
    float: left;
    margin-left: 20px;
}

.title {
    float: left;
}
```

```
.run_button {
   margin-top: 30px;
   margin-left: 30px;
   width: 100px;
   height: 30px;
}

.main_container {
   display: block;
   margin-left: 20px;
}

.dimensions_form input {
   width: 70px;
   font-size: 16px;
}

.main_container > div {
   margin-bottom: 30px;
}

.container_circuit {
   border-style: solid;
   border-width: 1px;
   width: 70%;
   height: 400px;
   display: grid;
   grid-template-columns: repeat(2, 50%);
   grid-template-rows: repeat(2, 50%);
   align-items: center;
   justify-items: center;
   place-self: center;
}

.sub_container_portes {
   grid-column: 1/2;
   grid-row: 1/2;
   aspect-ratio: 1;
   height: 50%;
   border: solid;
   border-width: 1px;
   display: flex;
   align-items: center;
   justify-content: center;
   font-weight: bold;
   background-color: white;
   user-select: none;
   -webkit-user-select: none;
   -moz-user-select: none;
```

```
}

[id^="r0"] {
   border: none;
   font-size: 1.2em;
}


/* #notthisone {
   grid-column: 1/2;
   grid-row: 1/2;
   border-color: blueviolet;
   background-color:blueviolet;
} */

.side_pane {
   position: fixed;
   top: 0;
   right: 0;
   padding-right: 10px;
   width: 25%;
}

.container_portes {
   /* border-color: rgb(3, 42, 126); */
   /* border-radius: 5%; */
   /* border-style: solid; */
   /* width: 200px; */
   height: 200px;
   display: grid;
   grid-template-columns: auto auto auto;
   /* grid-template-columns: repeat(0, 100px [col-start]);
   grid-template-rows: repeat(0, 100px [col-start]); */
   align-items: center;
   justify-items: center;
}


.porte {
   background-color: rgb(3, 42, 126);
   border-radius: 20%;
   border: none;
   color: white;
   font-size: 1.5em;
   text-align: center;
   font-weight: bold;
   padding: 0.5em;
   cursor: move;
   display: flex;
   align-items: center;
```

```
    justify-content: center;
    height: 70%;
    width: 70%;
    aspect-ratio: 1;
}
```