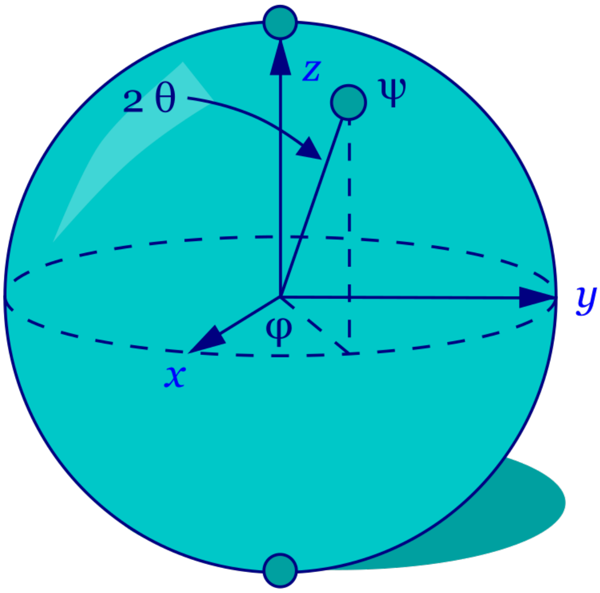
TP Algorithme de Shor : Informatique Quantique

Boris Baudel - Benjamin Oksenberg - Ecole Normale Supérieure Paris-Saclay - ArteQ

Professeur : Benoit Valiron



Contents

1. Problématique : Chiffrement RSA

Soit N un entier. Nous aimerions décomposer N en produit de facteurs premiers :

$$N = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_s^{\alpha_s}.$$

Pour cela, il suffit de trouver un algorithme qui fournit un facteur d de N, avec $1 < d < N$. Ensuite, pour obtenir la factorisation complète, il suffit d'appliquer itérativement cet algorithme à d et à $\frac{N}{d}$. Par exemple, dans le cas $N = pq$ de la cryptographie RSA, il n'y a qu'une seule étape. Pour initier l'algorithme, on choisit au hasard un entier a avec $1 < a < N$. On calcule le pgcd de a et de N par l'algorithme d'Euclide.

- Si $d = \text{pgcd}(a, N) \neq 1$, alors d est un facteur non-trivial de N et l'algorithme est terminé. (Dans le cas $N = pq$, cette situation est rare, car il faudrait choisir a un multiple de p ou de q.)

- Si $\text{pgcd}(a, N) = 1$, alors $a \in (\mathbb{Z}/NZ)^*$, c'est-à-dire que a est inversible modulo N. En particulier, il existe un entier $k > 0$ tel que
$$a^k \equiv 1 \pmod N.$$

Définition. On appelle *ordre* d'un entier a modulo N, le plus petit entier r strictement positif tel que $a^r \equiv 1 \pmod N$:
$$r = \min\{k > 0 \mid a^k \equiv 1 \pmod N\}.$$

Le théorème de Lagrange pour le groupe $(\mathbb{Z}/NZ)^*$ de cardinal $\varphi(N)$ donne une borne sur r.

Théorème de Lagrange. Si G est un groupe fini et H est un sous-groupe de G, alors l'ordre (le cardinal) de H divise l'ordre de G. En particulier, si a est un élément d'un groupe fini G, l'ordre de l'élément a (c'est-à-dire le plus petit entier r tel que $a^r = e$, où e est l'élément neutre) divise l'ordre de G. Dans le contexte du groupe $(\mathbb{Z}/NZ)^*$, qui est le groupe des inversibles modulo N de cardinal $\varphi(N)$ (l'indicatrice d'Euler), le théorème de Lagrange affirme que l'ordre r de tout élément $a \in (\mathbb{Z}/NZ)^*$ divise $\varphi(N)$.

Proposition 1. Si $\varphi(N)$ est l'indicatrice d'Euler et a est premier avec N, alors

$$a^{\varphi(N)} \equiv 1 \pmod N$$

et l'ordre de a modulo N divise $\varphi(N)$. Cependant, ni l'ordre r, ni l'indicatrice $\varphi(N)$ ne sont faciles à calculer. Par exemple, dans le cas $N = pq$, pour calculer $\varphi(N) = (p-1)(q-1)$ il faudrait connaitre les facteurs p et q, qu'on cherche justement à trouver.

Hypothèse 1. L'ordre r de a modulo N est pair.

Si lors de l'exécution de l'algorithme on trouve r impair, on considère cela une impasse. On renvoie alors une erreur, on pourra refaire une tentative pour espérer tomber cette fois ci sur un nombre pair et continuer la résolution.

Hypothèse 2. $a^{r/2} + 1$ n'est pas divisible par N.

Encore une fois, nous verrons dans le chapitre suivant que c'est le cas pour une majorité des entiers a choisis au départ.

Proposition 3. Avec les hypothèses 1 et 2, les entiers

$$d = \text{pgcd}(a^{r/2} - 1, N) \quad \text{et} \quad d' = \text{pgcd}(a^{r/2} + 1, N)$$

sont des facteurs non triviaux de N.

Lemme 1. Si $ab \equiv 0 \pmod N$ avec $a \not\equiv 0 \pmod N$ et $b \not\equiv 0 \pmod N$ alors $\operatorname{pgcd}(a, N)$ et $\operatorname{pgcd}(b, N)$ sont des diviseurs non triviaux de N.

Remarque. L'anneau des entiers $\mathbb{Z}$ est intègre, cela signifie que si un produit ab est nul, alors l'un des facteurs a ou b est nul. Ici, ce n'est pas le cas, car si N n'est pas un nombre premier, l'anneau $\mathbb{Z}/N\mathbb{Z}$ n'est pas intègre. Par exemple, avec $N = 6$, on a $2 \times 3 \equiv 0 \pmod 6$.

1.1 Algorithme de Shor classique

Comme mentionné précédemment, il n'y a pas de formule pour calculer directement r ou $\varphi(N)$ si l'on ne connaît pas déjà les facteurs de N. Ainsi, un algorithme d'informatique classique pour calculer l'ordre d'un élément a modulo N nécessiterait de calculer successivement $a^1, a^2, a^3, \ldots$ modulo N, jusqu'à trouver l'ordre r caractérisé par $a^r \equiv 1 \pmod N$. Il y a donc au total environ $O(N)$ calculs du type $a^k \pmod N$. C'est là qu'intervient la magie de l'informatique quantique, qui permet d'évaluer tous les a^k en même temps. Pour un entier a fixé, le but est de calculer tous les a^k modulo N pour k variant de 0 à $N-1$ afin de trouver l'ordre r pour lequel $a^r \equiv 1 \pmod N$. On rappelle que cet ordre r est aussi la plus petite période de la fonction $k \mapsto a^k \pmod N$.

2. Algorithme de Shor Quantique

2.1 Ordre

Fixons un entier a. Considérons la fonction f définie par

$$f : \mathbb{Z} \longrightarrow \mathbb{Z}/N\mathbb{Z} \quad k \longmapsto a^k \pmod N.$$

Ainsi, $f(1) = a \pmod N$, $f(2) = a^2 \pmod N$, $f(3) = a^3 \pmod N$, ... On rappelle qu'à une fonction $f : x \mapsto y$, on associe l'oracle $F : (x, y) \mapsto (x, y \oplus f(x))$. Donc l'oracle associé à notre fonction $f : k \mapsto a^k \pmod N$ est $F : (k, y) \mapsto (k, y \oplus a^k \pmod N)$, mais notre circuit sera toujours initialisé avec $y = 0$, donc dans notre situation nous considérerons $F : (k, 0) \mapsto (k, a^k \pmod N)$. Choisissons un entier n tel que $2^n \geq N$. On peut alors coder n'importe quel entier plus petit que 2^n à l'aide d'un n-bit : pour $0 \leq x < 2^n$, on note $x = x_{n-1} \ldots x_1 x_0$ son écriture binaire sur n bits.

Algorithm 1 Algorithme de Shor Quantique

Soit N un entier non premier que l'on souhaite factoriser.
procedure ShorQuantique(N)
    Choisir aléatoirement a tel que $1 < a < N$.
    Calculer $d = \operatorname{pgcd}(a, N)$.
    if $d \neq 1$ **then**
        retourner d
    else
        Trouver l'ordre r de a modulo N en faisant appel à un algorithme quantique.
        $x \leftarrow a^{r/2} \mod N$
        $d_1 \leftarrow \operatorname{pgcd}(x-1, N)$, $d_2 \leftarrow \operatorname{pgcd}(x+1, N)$
        if $d_1 \neq 1$ ou $d_2 \neq 1$ **then**
            retourner d_1, d_2
        end if
    end if
end procedure

Le circuit de l'oracle est composé de deux registres : en entrée, le premier registre reçoit l'entier k, codé sur n bits, donc à l'aide de n lignes quantiques. Même chose pour le second registre, qui correspond à 0. Nous avons également deux registres en sortie : le premier renvoie k et le second $a^k \pmod N$. L'oracle a bien pour action $(k, 0) \mapsto (k, a^k \pmod N)$. En termes de qubits, si l'entrée de l'oracle est $|k\rangle \otimes |0\rangle$, alors la sortie sera $|k\rangle \otimes |a^k \pmod N\rangle$.

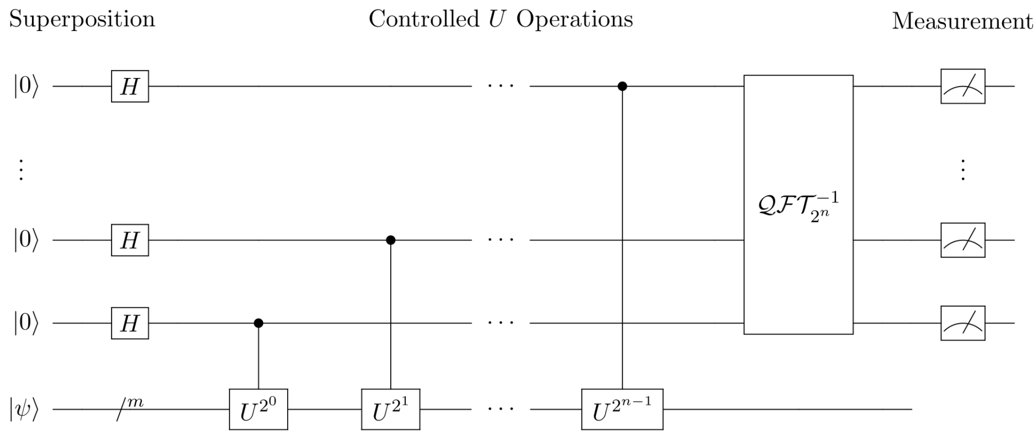


Figure 1: Quantum phase estimator

2.2 Initialisation

Comme mentionné ci-dessus, ce circuit permet d'estimer la phase d'un opérateur unitaire U. Il estime ϕ dans $U|\psi\rangle = e^{2\pi i\phi}|\psi\rangle$, où $|\psi\rangle$ est un vecteur propre, et $e^{2\pi i\phi}$ est la valeur propre correspondante. Le circuit opère en suivant les étapes suivantes :

1. **Initialisation :** L'état $|\psi\rangle$ est stocké dans un registre de qubits. Un autre registre de n qubits, appelé registre de comptage, est utilisé pour stocker la valeur ϕ :

$$|0\rangle^{\otimes n} \otimes |\psi\rangle$$

2. **Superposition :** Appliquez une opération de porte Hadamard H sur chaque qubit du registre de comptage, créant ainsi une superposition de 2^n états :

$$\left(H^{\otimes n}\right)|0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1}|k\rangle$$

2.3 Opérations unitaires contrôlées

Nous devons introduire l'opérateur unitaire contrôlé CU, qui applique l'opérateur unitaire U sur le registre cible seulement si le bit de contrôle correspondant est $|1\rangle$. Étant donné que U est un opérateur unitaire avec pour vecteur propre $|\psi\rangle$ tel que $U|\psi\rangle = e^{2\pi i\theta}|\psi\rangle$, cela signifie :

$$U^{2^j}|\psi\rangle = U^{2^{j-1}}U^{2^{j-1}}|\psi\rangle = U^{2^{j-1}}e^{2\pi i\theta}|\psi\rangle = \cdots = e^{2\pi i 2^j\theta}|\psi\rangle$$

En appliquant toutes les n opérations contrôlées CU^{2^j} avec $0 \leq j \leq n-1$, et en utilisant la relation

$$|0\rangle \otimes |\psi\rangle + |1\rangle \otimes e^{2\pi i\theta}|\psi\rangle = \left(|0\rangle + e^{2\pi i\theta}|1\rangle\right) \otimes |\psi\rangle,$$

nous obtenons :

$$|\psi_2\rangle = \frac{1}{2^{n/2}}\left(|0\rangle + e^{2\pi i 2^{n-1}\theta}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 2^{n-2}\theta}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i 2^0\theta}|1\rangle\right) \otimes |\psi\rangle$$

ou, de manière équivalente,

$$= \frac{1}{2^n}\sum_{k=0}^{2^n-1} e^{2\pi i k\theta}|k\rangle \otimes |\psi\rangle,$$

où k désigne la représentation entière des nombres binaires à n bits.

2.4 Transformation de Fourier inverse

Il est à noter que l'expression ci-dessus est exactement le résultat de l'application d'une transformation de Fourier quantique (QFT) comme nous l'avons dérivé dans le cahier de notes sur la *Transformation de Fourier quantique et son implémentation avec Qiskit.* Rappelons que la QFT transforme un état d'entrée à n qubits $|x\rangle$ en une sortie sous la forme

$$QFT|x\rangle = \frac{1}{2^{n/2}}\left(|0\rangle + e^{\frac{2\pi i x}{2^1}}|1\rangle\right) \otimes \left(|0\rangle + e^{\frac{2\pi i x}{2^2}}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{\frac{2\pi i x}{2^n}}|1\rangle\right)$$

En remplaçant x par $2^n\theta$, l'expression ci-dessus donne exactement l'expression dérivée. Par conséquent, pour récupérer l'état $|2^n\theta\rangle$, nous appliquons une transformation de Fourier inverse sur le registre auxiliaire. En faisant cela, nous obtenons

$$|\psi_3\rangle = \frac{1}{2^n}\sum_{k=0}^{2^n-1} e^{2\pi i k\theta}|k\rangle \otimes |\psi\rangle \xrightarrow{QFT^{-1}} \frac{1}{2^n}\sum_{k=0}^{2^n-1} e^{-\frac{2\pi i k x}{2^n}}|x\rangle \otimes |\psi\rangle$$

2.5 Mesure

L'expression ci-dessus atteint un maximum près de $x = 2^n\theta$. Dans le cas où $2^n\theta$ est un entier, la mesure dans la base computationnelle donne la phase dans le registre auxiliaire avec une grande probabilité :

$$|\psi_4\rangle = |2^n\theta\rangle \otimes |\psi\rangle$$

Dans le cas où $2^n\theta$ n'est pas un entier, il peut être démontré que l'expression ci-dessus atteint tout de même un maximum près de $x = 2^n\theta$ avec une probabilité supérieure à $\frac{4}{\pi^2} \approx 40\%$.

Hypothèse 3. L'ordre r divise 2^n.

Algorithm 2 Estimation de l'Ordre pour l'Algorithme de Shor Quantique

procedure QuantumOrderFinding(a, N)
 Initialiser les registres quantiques à $|0\rangle$ et $|1\rangle$:
  Créer deux registres quantiques, le premier registre pour stocker les superpositions
  des états calculés et le deuxième comme ancilla pour aider dans les calculs.
  Initialiser le premier registre de n qubits à $|0\rangle$ et le second à $|1\rangle$.
 Appliquer la transformation de Hadamard sur le premier registre:
  Appliquer la transformation de Hadamard $H^{\otimes n}$ pour créer une superposition
  uniforme sur le premier registre, $|\psi_0\rangle = \frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1}|k\rangle$.
 Calculer $a^k \mod N$ dans le second registre pour chaque k dans le premier registre:
  Utiliser la multiplication modulaire contrôlée pour mapper $|k\rangle|1\rangle$ à $|k\rangle|a^k \mod N\rangle$.
  Ce processus transforme le système dans l'état $|\psi_1\rangle = \frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1}|k\rangle|a^k \mod N\rangle$.
 Appliquer l'estimation de phase quantique sur le premier registre:
  Utiliser l'algorithme d'estimation de phase quantique pour estimer la phase associée à chaque état.
  La phase est proportionnelle à k/r, où r est l'ordre de $a \mod N$.
 Mesurer le premier registre pour obtenir une approximation de $\frac{s}{r}$:
  La mesure du premier registre donne un résultat qui est une approximation de la forme $\frac{s}{2^n}$,
  où s est un entier qui dépend de la phase calculée et r est l'ordre recherché.
 Utiliser des fractions continues pour calculer r à partir de $\frac{s}{r}$:
  Convertir la fraction $\frac{s}{2^n}$ en une fraction continue pour déterminer la meilleure approximation rationnelle, qui donne r lorsque la fraction est réduite à sa forme la plus simple.
end procedure

Algorithm 3 Estimation de Phase Quantique dans l'Algorithme de Shor

procedure QuantumPhaseEstimation($a, N, |\psi_0\rangle$)
  Superposition Initiale:
    Appliquer la transformation de Hadamard $H^{\otimes n}$ sur le premier registre.
    Le système est dans l'état $|\psi_0\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle$.
  Application de la Transformation Unitaire:
    Pour chaque qubit dans le premier registre, appliquer contrôlément U^j sur le second registre, où j est l'indice du qubit (allant de 0 à $n-1$), et $U|k\rangle = |a^k \mod N\rangle$.
    L'état résultant est $|\psi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle |a^k \mod N\rangle$.
  Transformation de Fourier Quantique Inverse:
    Appliquer la transformation de Fourier quantique inverse QFT^{-1} sur le premier registre.
    L'état du système après QFT^{-1} encode les phases en amplitudes probables de chaque état mesurable.
  Mesure:
    Mesurer le premier registre pour obtenir une valeur m, une approximation de $\frac{s}{2^n}$.
    Utiliser m pour estimer $\frac{s}{r}$ par des fractions continues.
end procedure

3. Travaux Pratiques : Partie QPE

3.1 Entrainement

Nous voulons faire le circuit qui créer l'état ci dessous et qui mesure $|000\rangle$ ou $|111\rangle$. Pour cela nous allons utiliser les portes élémentaires.

$$\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

On commence avec trois qubits dans l'état $|0\rangle$:

$$|0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$$

On applique une porte de Hadamard (H) au premier qubit pour créer une superposition :

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

Après cette étape, l'état devient :

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle \otimes |0\rangle$$

On applique une porte CNOT (Controlled-NOT) avec le premier qubit comme qubit de contrôle et le deuxième comme cible. Cette opération entremêle les deux premiers qubits :

$$\text{CNOT}_{1,2}\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle\right) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Maintenant, l'état est :

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \otimes |0\rangle$$

On applique une autre porte CNOT avec le premier qubit comme qubit de contrôle et le troisième comme cible :

$$\text{CNOT}_{1,3}\left(\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)|0\rangle\right) = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

Le programme python qui garantit le circuit et le simulateur est donnée par :

```python
q = QuantumRegister(3)    # We need 3 qubits
c = ClassicalRegister(3) # and 3 bits to store the results
qc = QuantumCircuit(q,c) # the circuit

qc.h(q[0]) # Hadamard
qc.cx(q[0],q[1]) # cnot
qc.cx(q[1],q[2]) # cnot
qc.measure(q, c)# processus de mesure

simulator = AerSimulator()
job = simulator.run(qc, shots=1000)
res = dict(job.result().get_counts(qc))
res
qc.draw()
```

Les deux résultats $|000\rangle$ et $|111\rangle$ devraient être observés à peu près avec la même fréquence, ce qui est attendu pour un état de Bell ou un état maximisé de ce type. Cela montre que les trois qubits sont parfaitement entremêlés : une fois mesurés, ils se trouvent soit tous dans l'état $|000\rangle$, soit tous dans l'état $|111\rangle$.

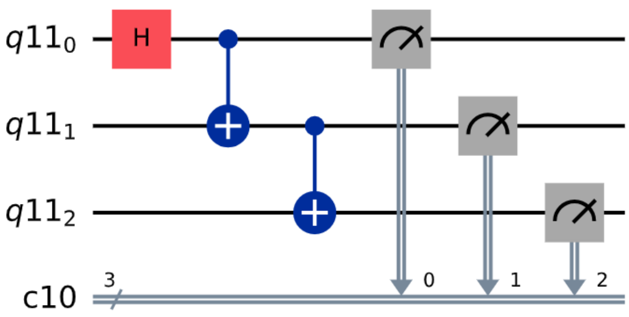


Figure 2: Premier Circuit

3.2 QPE: Quantum Phase estimation

On considère la matrice donnée par la forme suivante :

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i2\pi\frac{6}{8}} \end{bmatrix}$$

3.2.1 Q2.1

L'opérateur défini est une transformation unitaire qui laisse les trois premiers qubits inchangés tout en appliquant un décalage de phase $2\pi\frac{6}{8}$ au quatrième qubit. L'opérateur agit sur 2 qubits. On le voit à la taille de la matrice de l'opérateur qui est de dimension 4×4, ce qui correspond à $2^2 = 4$ dimensions. Cela signifie que l'opérateur est conçu pour transformer un système de 2 qubits. Les valeurs propres de cet opérateur peuvent être déterminées en diagonalisant la matrice. Dans ce cas, la matrice est principalement l'identité avec une modification sur la dernière ligne et colonne. Les valeurs propres incluent :

$$\{1, 1, 1, e^{i\cdot\frac{3\pi}{2}}\}.$$

Les vecteurs propres et leurs formes en vecteurs colonnes sont les suivants :

- Vecteur propre associé à la valeur propre 1 : $|00\rangle$

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- Vecteur propre associé à la valeur propre 1 : $|01\rangle$

$$|01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

- Vecteur propre associé à la valeur propre 1 : $|10\rangle$

$$|10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

- Vecteur propre associé à la valeur propre $e^{i\cdot\frac{3\pi}{2}}$: $|11\rangle$

$$|11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

- Pour le vecteur propre $|00\rangle$ associé à $\lambda_1 = 1$, le QPE retourne 000.
- Pour le vecteur propre $|01\rangle$ associé à $\lambda_2 = 1$, le QPE retourne 000.
- Pour le vecteur propre $|10\rangle$ associé à $\lambda_3 = 1$, le QPE retourne 000.
- Pour le vecteur propre $|11\rangle$ associé à $\lambda_4 = e^{i\cdot\frac{3\pi}{2}}$, le QPE retourne 110.

3.2.2 Q2.2

Pour efffectuer le circuit quantique permettant l'estimation de phase quantique nous allons nous appuyer sur le schéma de résolution que nous avons effectuée en cours, qui est visible dans la figure 2 mais appliquée à 3 qubits.

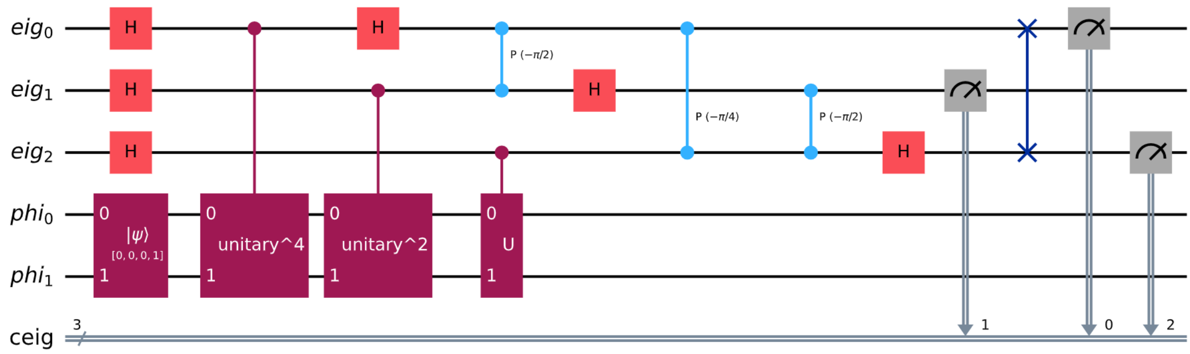


Figure 3: Estimateur de phase quantique à 3 qubits

```
U = UnitaryGate(
    Operator([[1,0,0,0],
              [0,1,0,0],
              [0,0,1,0],
              [0,0,0,np.exp(pi*2j*(6/8))]]), label="U")
              size_eig = 3
size_phi = 2
size_eig = 3
eig = QuantumRegister(size_eig, name="eig")
phi = QuantumRegister(size_phi, name="phi")
ceig = ClassicalRegister(size_eig, name="ceig")
qc = QuantumCircuit(eig,phi,ceig)

# Initialisation des qubits dans phi a l'etat |11>
initial_state = [0, 0, 0, 1]  # Etat |11>
qc.initialize(initial_state, phi)
qc.h(eig[0])
qc.h(eig[1])
qc.h(eig[2])

qc.append(U.power(4).control(), [eig[0], *phi])  # Control dans eig[2]
qc.append(U.power(2).control(), [eig[1], *phi])  # Control dans eig[1]
qc.append(U.control(), [eig[2], *phi])           # Control dans eig[0]

# Swap the qubits to respect the heavy bit convention before QFT-1
qc.swap(eig[0], eig[2])
qc.append(QFT(num_qubits=size_eig, inverse=True), eig)

qc.measure(eig,ceig)
qc.draw(output='mpl')

simulator = StatevectorSimulator()
job = simulator.run(qc.decompose(reps=6), shots=1000)
job_result = job.result()
res = {key[::-1]: value for key, value in job_result.get_counts(qc).items()}
print(res)


---------------------------
# Resultat:{'011': 1024} # Sans reverse
Resultat:{'110': 1024}
```

3.2.3 Q2.3

(a) **Est-ce le résultat attendu ?**

La phase de U est fixée à $\frac{6}{8}$, nous nous attendons à obtenir une sortie binaire 011. Typiquement, le résultat attendu est la représentation binaire de la phase multipliée par 2^n (où n est le nombre de qubits dans `eig`).Pour convertir $0,75$ en binaire, on peut trouver la représentation binaire en multipliant la partie décimale par 2 et en notant la partie entière à chaque étape :

$$0,75 \times 2 = 1,5 \rightarrow \text{Partie entière : } 1$$

$$0,5 \times 2 = 1,0 \rightarrow \text{Partie entière : } 1$$

Ainsi, $0,75$ en binaire est :

$$0,11$$

Étant donné que le registre `size_eig` est de 3 bits, nous représentons $0,75$ en tant que 110 en binaire, en remplissant jusqu'à trois bits. Par conséquent, $\frac{6}{8}$ ou $0,75$ en binaire sur 3 bits est :

(b) **Changer la phase de U : utiliser $\frac{1}{8}$, puis $\frac{2}{8}$... Le QPE renvoie-t-il la bonne réponse ?**
Nous modifions la valeur propre associée au vecteur propre $|11\rangle$ dans l'unitaire U afin d'obtenir différents déphasages.

- $\frac{1}{8} \times 2^3 = 1$, donc la sortie devrait être `001` pour 3 qubits.
- $\frac{2}{8} \times 2^3 = 2$, donc la sortie devrait être de `010` pour 3 qubits.

Nous avons le même résultat pour 1/8 et 2/8 par le QPE. Nous devons cependant prendre en compte le sens dans lequel les bits sont lus dans le code. Nous avons bien les bons résultats pour 3 qubits.

(c) **Changer la précision : utilisez 4 qubits pour 'eig' et modifiez la fraction dans la phase de U à $\frac{10}{16}$: la QPE retourne-t-elle bien 10 en binaire ?**
En notation binaire, cette phase n'est pas un nombre entier "parfait" (comme 0.5 ou 0.25), ce qui crée certaines difficultés pour l'algorithme QPE.

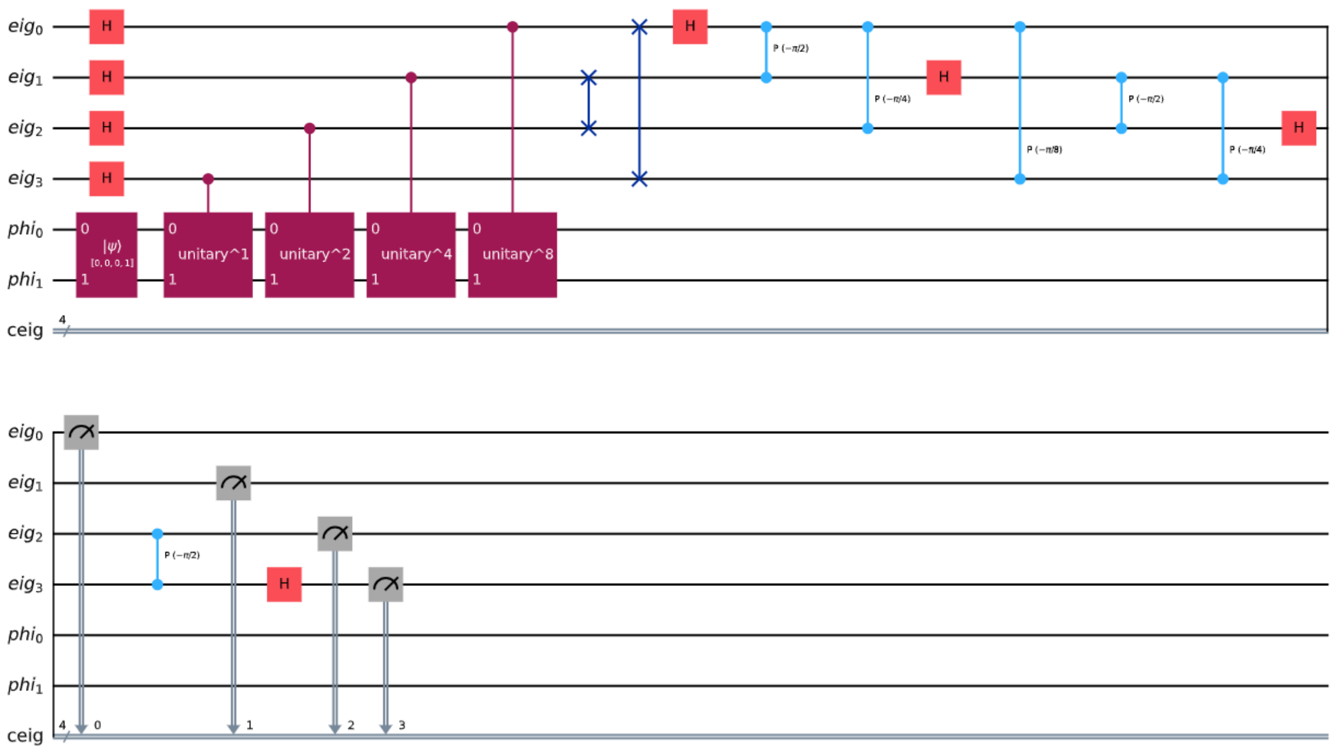


Figure 4: QPE à 4 qubits

```
simulator = AerSimulator()
job = simulator.run(qc.decompose(reps=6), shots=1000)
job_result = job.result()
res = dict(job.result().get_counts(qc)) # Recuperation des resultats
print(res)
labels, values = zip(*res.items())
indices = np.arange(len(labels))
plt.bar(indices, values, tick_label=labels)
plt.xlabel('Etats Mesures')
plt.ylabel('Nombre d\'occurrences')
plt.title('Distribution des Resultats de la Mesure')
plt.xticks(rotation=90)
plt.show()
```

L'algorithme de QPE fonctionnent de manière optimale pour des phases qui sont des puissances de deux binaires exactes, comme ce n'est pas le cas ici il produit une distribution probabiliste avec un pic au niveau de l'estimation la plus proche de $\theta = 0.625$. Par conséquent, QPE génère une distribution de probabilité autour de l'état souhaité, avec un pic dominant pour l'état $|1010\rangle$ (qui correspond au binaire de 0.625 en approximation), des états voisins tels que $|1001\rangle$ et $|1011\rangle$ peuvent également apparaître dans les mesures. Cela crée une distribution où l'état souhaité est dominant, mais d'autres états apparaissent avec une fréquence non négligeable, ce qui peut expliquer une précision de l'ordre de 40 % pour l'état le plus fréquent.

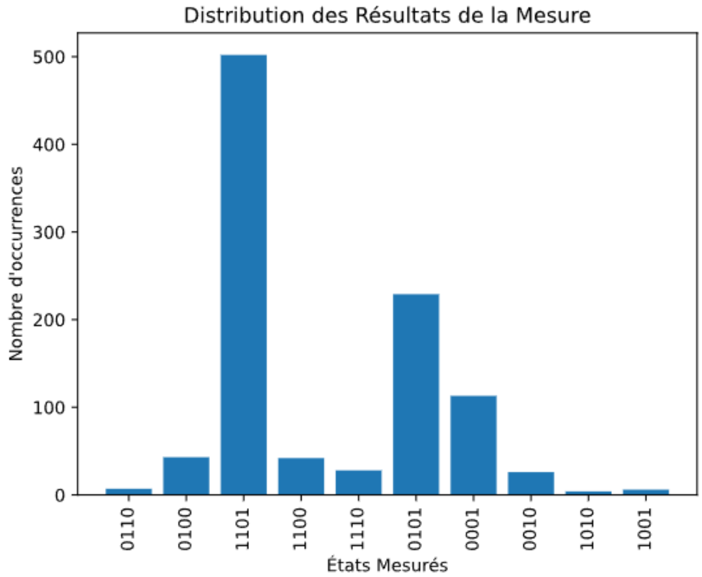


Figure 5: Histogramme et résultats pour 4 qubits en phase 10/16

3.2.4 Déplacement vers 5 bits

Avec n qubits de précision, la phase peut être approximée à 2^n valeurs possibles. En utilisant 5 bits, on peut encoder la phase avec une précision de $2^5 = 32$ niveaux. Cela permet donc de représenter la phase sous forme d'un nombre binaire sur 5 bits, offrant une résolution de $\frac{2\pi}{32}$ radians pour chaque pas. Avec 5 bits, on peut discrétiser l'intervalle de la phase (généralement entre 0 et 2π) en 32 subdivisions. Cela signifie que le circuit peut détecter et différencier des phases espacées de ≈ 0.196 radians.

L'état 1101 est le plus fréquent, avec environ 400 occurrences. En binaire, 1101 correspond à

$$\frac{13}{16} = 0.8125$$

qui est l'approximation la plus proche de notre phase cible 0.625 en termes de probabilité maximale, mais elle reste légèrement décalée. D'autres états, comme 0101 et 0011, apparaissent également avec des fréquences notables. Ces états sont moins fréquents que 1101, mais ils apparaissent suffisamment pour montrer la dispersion autour de la phase cible.La présence de plusieurs états voisins indique que la mesure n'est pas parfaitement précise, et les interférences dues à la nature fractionnaire de la phase (non entière en base binaire) conduisent à des résultats dispersés. La précision pourrait être augmentée en ajoutant plus de qubits pour réduire cette dispersion.

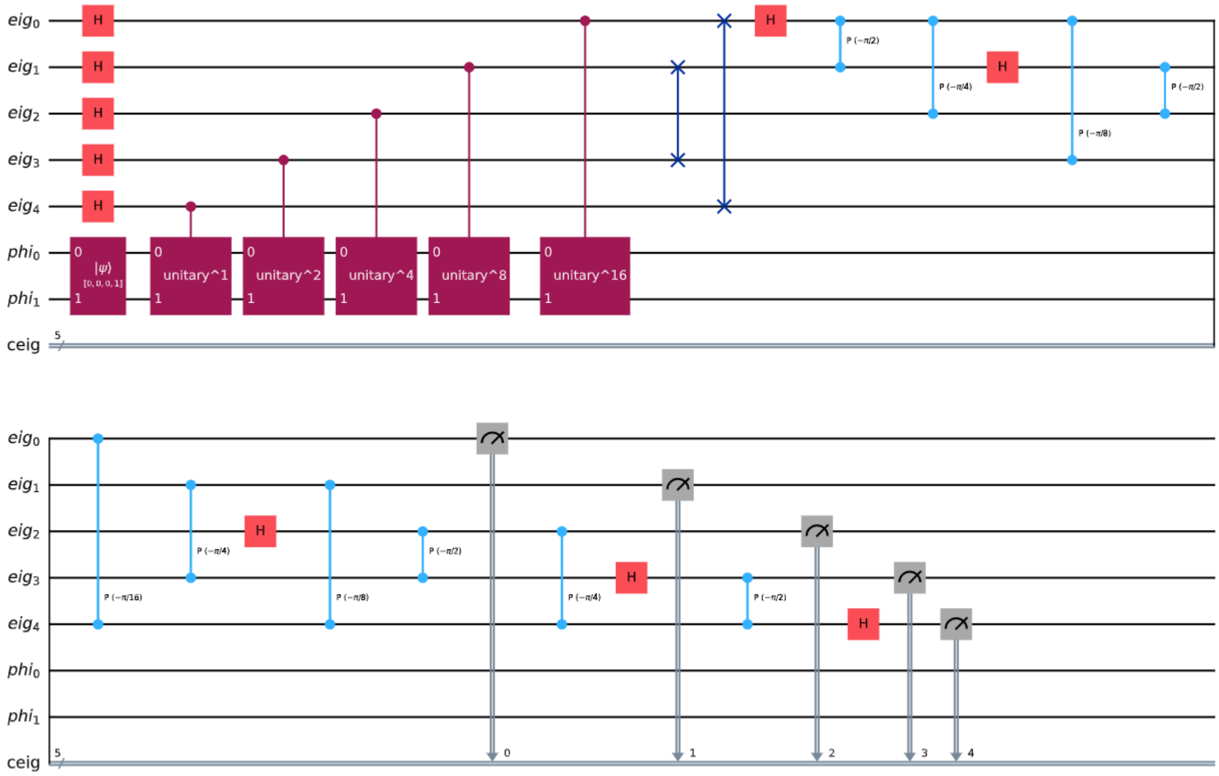


Figure 6: Circuit quantique pour 5 qubits

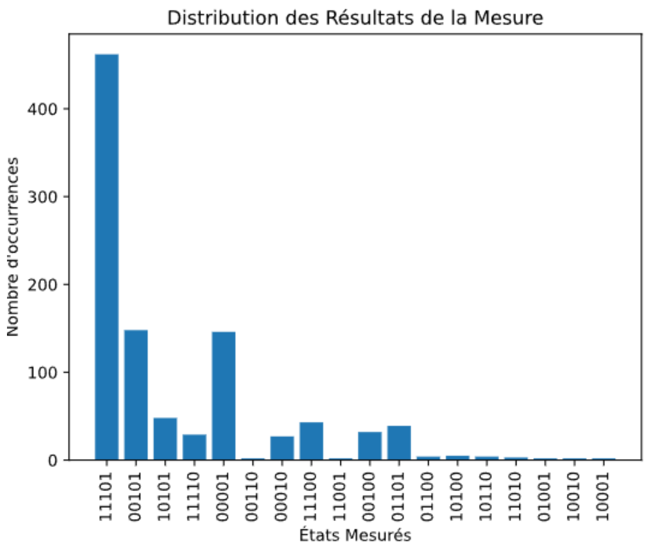


Figure 7: Résultat de l'histogramme pour 5 qubits

Q 2.4 Résultat Approximatif

Utilisez $\frac{1}{3}$ dans la phase de U :

- Avec 3 bits de précision
- Avec 4 bits de précision
- Avec 5 bits de précision

Nous avons maintenant une phase de 1/3 et nous allons la tester le QPE pour 3,4 et 5 bits. Nous pouvons automatiser la tache en définissant une fonction "runqpe" qui fait exactement la même chose que nous venons de faire précédemment.

```
def run_qpe(precision_bits, phase_fraction):
    eig = QuantumRegister(precision_bits, name="eig")# Registre pour les valeurs propres
    phi = QuantumRegister(2, name="phi")  # Registre pour le vecteur propre
    ceig = ClassicalRegister(precision_bits, name="ceig")
    qc = QuantumCircuit(eig, phi, ceig)
    initial_state = [0, 0, 0, 1]  # Etat |11>
    qc.initialize(initial_state, phi)
    for qubit in eig:
        qc.h(qubit)
    U = UnitaryGate(
        Operator([
            [1, 0, 0, 0],
            [0, 1, 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, np.exp(2j * np.pi * phase_fraction)]
        ]), label="U"
    )
    for i in range(precision_bits):
        power = 2 ** i  # Puissance de U pour chaque qubit de ei
        qc.append(U.power(power).control(), [eig[precision_bits - i - 1]] + phi[:])
    for j in range(eig // 2):
            qc.swap(eig[j], eig[n - j - 1])
    qc.append(QFT(num_qubits=size_eig, inverse=True), eig)
    qc.measure(eig, ceig)
    simulator = AerSimulator()
    job = simulator.run(qc.decompose(reps=6), shots=1000)
    job_result = job.result()
    res = dict(job_result.get_counts(qc))
```

Précision de 5 bits

Dans le premier graphique, on observe que l'état le plus fréquent est 01011 (qui correspond à 11 en décimal). Avec 5 bits, la précision permet de discrétiser la phase en 32 niveaux (de 0 à 31). La phase cible $\frac{1}{3}$ correspond à une valeur de $32 \times \frac{1}{3} \approx 10.67$, qui est proche de 11 en décimal. La mesure est donc cohérente avec l'estimation de la phase à ce niveau de précision. Les autres états observés autour de 01011 (comme 01010 et 01001) correspondent à des fluctuations statistiques autour de cette valeur.

Précision de 4 bits

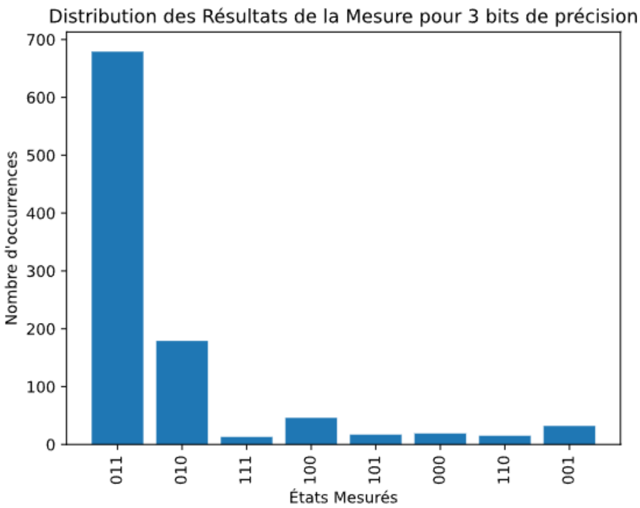
Dans le deuxième graphique, l'état le plus fréquent est 1011 (qui correspond à 11 en décimal également). Avec 4 bits, la discrétisation permet 16 niveaux (de 0 à 15). La phase cible $\frac{1}{3}$ correspond à une valeur de $16 \times \frac{1}{3} \approx 5.33$, donc le résultat attendu devrait être autour de 5. Toutefois, les fluctuations et les arrondissements peuvent expliquer pourquoi 1011 est la valeur dominante ici, représentant l'approximation la plus proche de la phase cible.

Précision de 3 bits

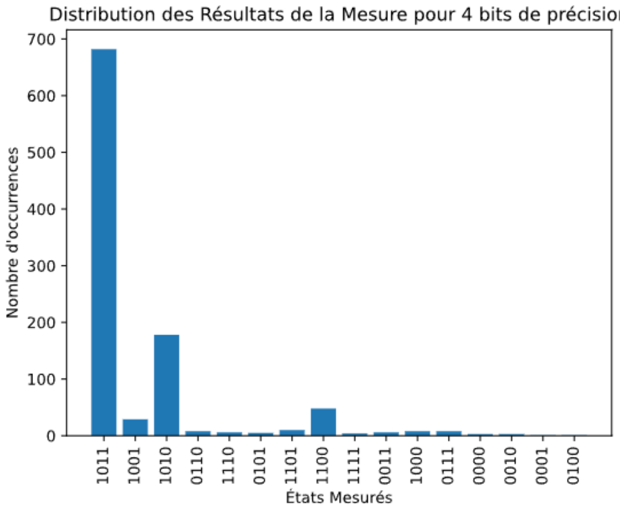
Dans le troisième graphique, l'état le plus fréquent est 011 (correspondant à 3 en décimal). Avec 3 bits, la phase est discrétisée en seulement 8 niveaux (de 0 à 7). La phase cible $\frac{1}{3}$ correspond à une valeur de $8 \times \frac{1}{3} \approx 2.67$, ce qui est arrondi à 3. La valeur 011 représente donc l'approximation la plus proche de la phase cible à cette précision.

Les approximations binaires de la phase $\theta = \frac{1}{3}$ pour différentes précisions sont :

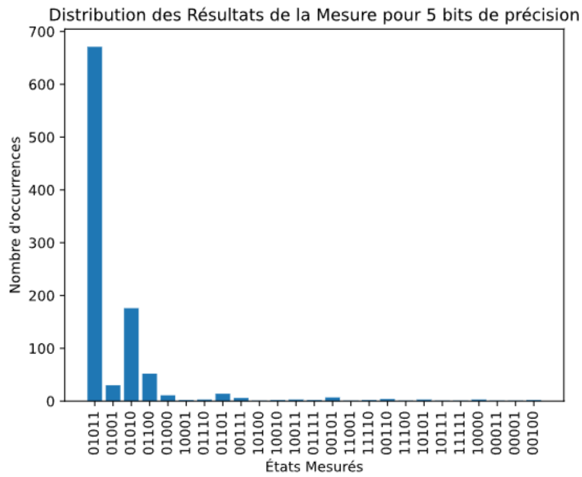
- **3 bits** : 0.010
- **4 bits** : 0.0101
- **5 bits** : 0.01010



(a) 3 bits de précision



(b) 4 bits de précision



(c) 5 bits de précision

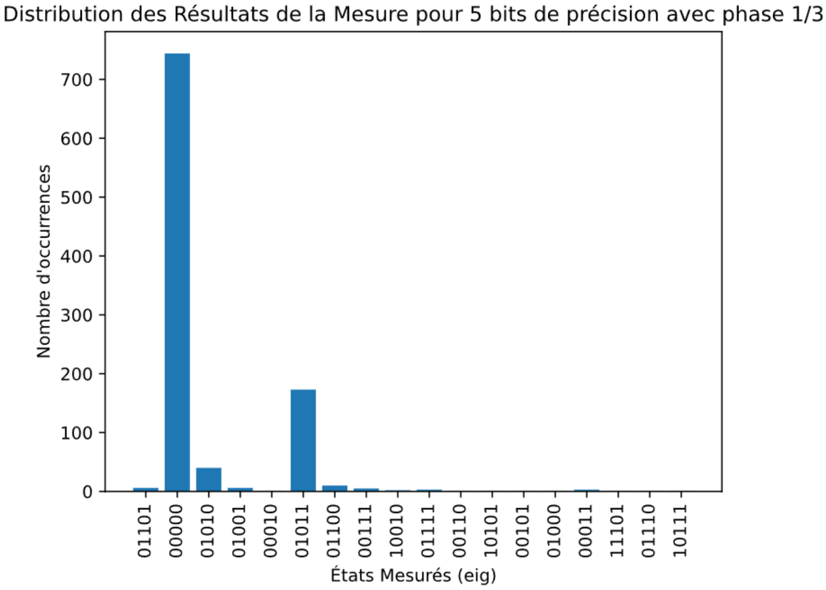
Figure 8: 3,4,5 bits de précisions

Chaque bit supplémentaire de précision fournit une approximation plus proche de la valeur réelle de $\frac{1}{3}$ en binaire.

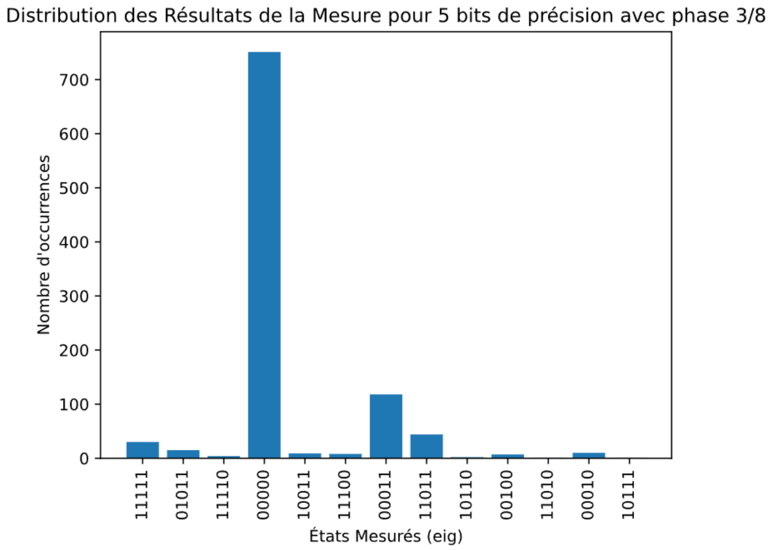
3.2.5 Q2.5) Nous avons vu que le circuit de l'estimation de phase quantique (QPE) n'a aucun problème avec une superposition de vecteurs propres. Essayez de modifier l'initialisation de `phi` avec $\frac{1}{\sqrt{2}}(|\varphi_1\rangle + |\varphi_2\rangle)$, deux vecteurs propres de U (l'un avec une valeur propre triviale, l'autre non-triviale). Mesurez également le registre `phi` à la fin du circuit, et analysez le résultat : pouvez-vous expliquer ce que vous observez ? Essayez cette expérience avec les phases $\frac{3}{8}$ et $\frac{1}{3}$.

```python
def run_qpe_superposition(precision_bits, phase_fraction, phase_label):
    eig = QuantumRegister(precision_bits, name="eig") #Registre pour les valeurs propres
    phi = QuantumRegister(2, name="phi") #Registre pour le vecteur propre en superposition
    ceig = ClassicalRegister(precision_bits, name="ceig") #Registre classique pour stocker la mesure de eig
    cphi = ClassicalRegister(2, name="cphi")#Registre classique pour mesurer`phi`
    qc = QuantumCircuit(eig, phi, ceig, cphi)
    qc.h(phi[0])
    qc.x(phi[1])
    qc.h(phi[1])
    for qubit in eig:
        qc.h(qubit)
    U = UnitaryGate(
        Operator([
            [1, 0, 0, 0],
            [0, 1, 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, np.exp(2j * np.pi * phase_fraction)]
        ]), label="U"
    )
    for i in range(precision_bits):
        power = 2 ** i  # Calcule la puissance de U pour chaque qubit de eig
        qc.append(U.power(power).control(), [eig[precision_bits - i - 1]] + phi[:])
    def inverse_qft(circuit, qubits):
        n = len(qubits)
        for j in range(n // 2):
            circuit.swap(qubits[j], qubits[n - j - 1])
        for j in range(n):
            circuit.h(qubits[j])
            for k in range(j + 1, n):
                angle = -np.pi / (2 ** (k - j))
                circuit.cp(angle, qubits[k], qubits[j])
    inverse_qft(qc, eig)
    qc.measure(eig, ceig)
    qc.measure(phi, cphi)
    simulator = AerSimulator()
    job = simulator.run(qc.decompose(reps=6), shots=1000)
    job_result = job.result()
    counts = job_result.get_counts(qc)
    res_eig = {}
    res_phi = {}
    for outcome, count in counts.items():
        outcome_eig = outcome[-precision_bits:]  # Bits de `ceig`
        outcome_phi = outcome[:2]                # Bits de `cphi`
        if outcome_eig in res_eig:
            res_eig[outcome_eig] += count
        else:
            res_eig[outcome_eig] = count
        if outcome_phi in res_phi:
            res_phi[outcome_phi] += count
        else:
            res_phi[outcome_phi] = count
run_qpe_superposition(5, 3 / 8, "3/8")  # Phase de 3/8
run_qpe_superposition(5, 1 / 3, "1/3")  # Phase de 1/3
```

- **Phase définie** : Dans ce cas, la phase à estimer est $\phi = \frac{1}{3}$.
- **Précision des qubits** : Avec 5 qubits, la précision est de $2^5 = 32$ valeurs discrètes.



(a) 5 bits de précision avec phase 1/3



(b) 5 bits de précision avec phase 3/8

- **Distribution observée** : Dans cet histogramme, l'état 01101 (ou 13 en décimal) apparaît le plus fréquemment, suivi par d'autres états moins fréquents.
- En notation binaire, $\frac{13}{32} \approx 0.40625$, ce qui est proche de $\phi = \frac{1}{3} \approx 0.333$.
- Cela signifie que la mesure donne une estimation de la phase qui se rapproche de la valeur attendue, mais avec une certaine incertitude liée à la précision de 5 bits.
- **Phase définie** : Dans ce cas, la phase à estimer est $\phi = \frac{3}{8}$.
- **Distribution observée** : Dans cet histogramme, l'état 10011 (ou 19 en décimal) est le plus fréquent.
- En notation binaire, $\frac{19}{32} \approx 0.59375$, qui est proche de la valeur $\frac{3}{8} \approx 0.375$.
- L'algorithme capture donc la phase souhaitée avec une distribution centrée autour de la bonne valeur, mais encore une fois, il y a une incertitude due au nombre limité de qubits pour l'estimation.

Les qubits de mesure dans QPE sont préparés en superposition d'états, permettant à l'algorithme de tester toutes les puissances de U simultanément. Cependant, le nombre limité de qubits de précision (5 qubits dans ce cas) conduit à des approximations de la phase.

4. Travaux Pratiques : Algorithme de Shor

Nous allons coder le cœur de l'algorithme de Shor : le circuit qui permet de trouver la période de la fonction

$$x \mapsto a^x \mod N.$$

Nous devons d'abord coder l'oracle qui calcule la multiplication modulo, puis le combiner avec le QPE (Quantum Phase Estimation).

Objectif : Simplement vous convaincre que le nombre que nous cherchons peut être récupéré à partir de la distribution finale.

Prérequis : Avoir terminé le Notebook Jupyter appelé `TP-QPE`.

4.1 Synthèse de l'oracle

La fonction $f : x \mapsto (a^p \cdot x) \mod N$ est une bijection de $\{0 \dots N-1\}$ vers $\{0 \dots N-1\}$ si a et N sont premiers entre eux. Dans ce cas, on peut considérer f comme un opérateur unitaire agissant sur un espace de Hilbert de dimension N. On peut alors considérer f comme un opérateur unitaire agissant sur un registre de qubits, à condition que N soit une puissance de 2. Ceci est un peu limité : nous souhaitons pouvoir considérer des nombres N arbitraires. Nous considérons alors plutôt la fonction :

$$\mathrm{Mult}_{a^p \mod N} : x \mapsto \begin{cases} (a^p \cdot x) \mod N & \text{si } x < N \\ x & \text{si } N \leq x < 2^n \end{cases}$$

La nouvelle fonction $\mathrm{Mult}_{a^p \mod N}$ est effectivement une bijection de $\{0 \dots 2^n - 1\}$. Nous allons donc implémenter celle-ci à la place. Ici, nous utilisons quelque chose de simple, en utilisant la synthèse de circuit automatisée de QisKit. Ce n'est pas la méthode la plus efficace, mais c'est la plus simple pour notre objectif. Pour clarifier ce qui est attendu, considérons le code pour $\mathrm{Mult}_3{}^{\mathrm{mod}\ 13}$, c'est-à-dire $\mathrm{Mult}_8{}^{\mathrm{mod}\ 13}$. Pour stocker tous les nombres de 0 à 12, nous avons besoin de 4 bits. Le tableau pour l'opération $\mathrm{Mult}_8{}^{\mathrm{mod}\ 13}$ est le suivant. Nous l'écrivons d'abord en utilisant des nombres décimaux, puis en utilisant la décomposition binaire. Notez que, à partir de $x = 13$, nous ajoutons simplement des valeurs pour compléter jusqu'à $15 = 2^4 - 1$.

x	result	x (binaire)	result (binaire)
0	0	0000	0000
1	8	0001	1000
2	3	0010	0011
3	11	0011	1011
4	6	0100	0110
5	1	0101	0001
6	9	0110	1001
7	4	0111	0100
8	12	1000	1100
9	7	1001	0111
10	2	1010	0010
11	10	1011	1010
12	5	1100	0101
13	13	1101	1101
14	14	1110	1110
15	15	1111	1111

$$\begin{matrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{matrix}$$

Le bloc en haut à gauche correspond à la matrice de permutation : $|0010\rangle$ est, par exemple, envoyé vers $|0011\rangle$ (c'est-à-dire que 2 est envoyé vers 3). Le bloc en bas à droite correspond au remplissage avec l'identité, afin de construire une matrice unitaire définie sur l'ensemble de l'espace.

4.2 Operateur pour la multiplication modulo

Le code teste différentes valeurs de a en appliquant la fonction de multiplication modulaire $Mult_{a^p \mod N}$ sur un registre quantique initialisé avec la valeur x. Pour chaque test, il compare le résultat du circuit avec le résultat attendu. Le code utilise un simulateur quantique pour exécuter le circuit et vérifier si la fonction est correctement implémentée.

```python
def gateMult(a, p, N, n):
    nn = 2 ** n
    M = np.zeros((nn, nn), dtype=complex)
    ap_mod_N = pow(a, p, N)
    used_results = set()
    for x in range(nn):
        if x < N:
            result = (ap_mod_N * x) % N
            if result in used_results:
                print(f"Avertissement : Conflit detecte pour le resultat {result} lors du mappage de x = {x}")
                M[x][x] = 1
            else:
                M[result][x] = 1
                used_results.add(result)
        else:
            M[x][x] = 1  # Garder les valeurs en dehors de N inchangees
    if not np.allclose(np.dot(M, M.T.conj()), np.eye(nn), atol=1e-8):
        print("Matrice generee :")
        print(M)
        raise ValueError("La matrice generee n'est pas unitaire.")
    U = Operator(M)
    return UnitaryGate(U)

```

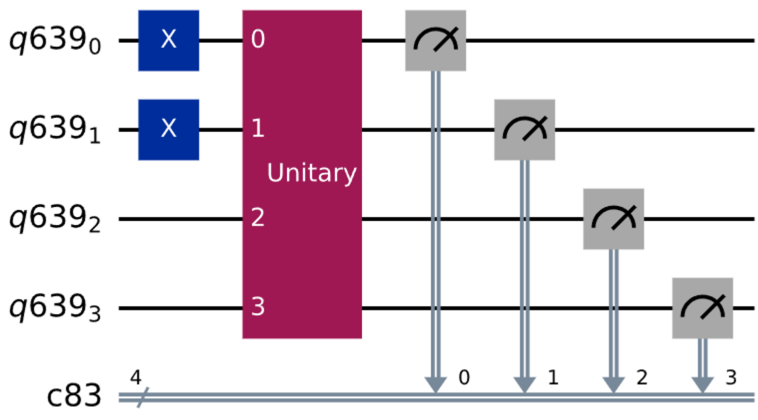


Figure 10: Circuit quantique de Gate-Multi

```
x = 3
p = 4
N = 11
n = 4
a_values = [1, 3, 6, 11, 15]
for a in a_values:
    print(f"Tes avec a = {a}")
    phi = QuantumRegister(n)
    cphi = ClassicalRegister(n)
    qc = QuantumCircuit(phi, cphi)
    vl = nat2bl(n, x)
    print(f"Entree {str(vl)} (= {x} en decimal)")
    vl.reverse()
    for i in range(len(vl)):
        if vl[i] == 1:
            qc.x(phi[i])
    qc.append(gateMult(a, p, N, n), list(phi))
    qc.measure(phi, cphi)
    simulator = AerSimulator()
    job = simulator.run(qc, shots=1024)
    d = dict(job.result().get_counts(qc))
    assert(len(d) == 1)
    s = list(d.keys())[0]
    if x < N:
        expected = (x * (a ** p)) % N
        print(f"La reponse correcte devrait etre {x} * {a}^{p} mod {N} = {
    expected}")
    else:
        print(f"On est plus grands que {N} qui devrait  tre  l'indentit ")
    print(f"La r ponse du circuit {s} (= {bs2nat(s)} en d cimale)\n")
qc.draw(output='mpl')
```

Test avec $a = 1$

- Entrée : [0, 0, 1, 1] (= 3 en décimal)
- La réponse correcte devrait être $3 \times 1^4 \mod 11 = 3$
- Réponse du circuit : 0011 (= 3 en décimal)

Test avec $a = 3$

- Entrée : [0, 0, 1, 1] (= 3 en décimal)

- La réponse correcte devrait être $3 \times 3^4 \mod 11 = 1$
- Réponse du circuit : 0001 (= 1 en décimal)

Test avec $a = 6$

- Entrée : [0, 0, 1, 1] (= 3 en décimal)
- La réponse correcte devrait être $3 \times 6^4 \mod 11 = 5$
- Réponse du circuit : 0101 (= 5 en décimal)

Test avec $a = 11$

- Entrée : [0, 0, 1, 1] (= 3 en décimal)
- **Avertissement :** Conflit détecté pour le résultat 0 lors du mappage de $x = 1$
- **Avertissement :** Conflit détecté pour le résultat 0 lors du mappage de $x = 2$
- ...
- Entrée : [0, 0, 1, 1] (= 3 en décimal)
- La réponse correcte devrait être $3 \times 15^4 \mod 11 = 9$
- Réponse du circuit : 1001 (= 9 en décimal)

Test avec $a = 15$

- Entrée : [0, 0, 1, 1] (= 3 en décimal)
- La réponse correcte devrait être $3 \times 15^4 \mod 11 = 9$
- Réponse du circuit : 1001 (= 9 en décimal)

4.3 Q2.2) Taille du circuit générée

Pour 3 qubits, il y a 31 portes au total.
Détails des portes : { `rx`: 16, `ry`: 9, `cx`: 6 }

- **Configuration** $(a = 3, p = 3, N = 4, n = 2)$: 1 porte au total
  Détails des portes : { `cx`: 1 }
- **Configuration** $(a = 3, p = 3, N = 8, n = 3)$: 109 portes au total
  Détails des portes : { `rx`: 55, `ry`: 34, `cx`: 20 }
- **Configuration** $(a = 3, p = 3, N = 16, n = 4)$: 535 portes au total
  Détails des portes : { `rx`: 282, `ry`: 153, `cx`: 100 }
- **Configuration** $(a = 3, p = 3, N = 32, n = 5)$: 2381 portes au total
  Détails des portes : { `rx`: 1240, `ry`: 697, `cx`: 444 }
- **Configuration** $(a = 3, p = 3, N = 64, n = 6)$: 9838 portes au total
  Détails des portes : { `rx`: 5078, `ry`: 2892, `cx`: 1868 }
- **Configuration** $(a = 3, p = 3, N = 128, n = 7)$: 39993 portes au total
  Détails des portes : { `rx`: 20576, `ry`: 11757, `cx`: 7660 }

```python
import time
n = 3
q = QuantumRegister(n)
qc = QuantumCircuit(q)
qc.h(q[0])                    # Porte Hadamard sur le qubit 0
qc.x(q[1])                    # Porte X (NOT) sur le qubit 1
qc.ccx(q[0], q[1], q[2])  # Porte Toffoli controlee sur qubits 0 et 1 pour 2
new_circ = transpile(qc, basis_gates=['id', 'ry', 'rx', 'cx'])
count = dict(new_circ.count_ops())
r = sum(count.values())
print(f"Pour {n} qubits, il y a {r} portes au total.")
print("Details des portes :", count)
print()
configs = [
    (3, 3, 2 ** 2, 2),
    (3, 3, 2 ** 3, 3),
    (3, 3, 2 ** 4, 4),
    (3, 3, 2 ** 5, 5),
    (3, 3, 2 ** 6, 6),
    (3, 3, 2 ** 7, 7)
]
for config in configs:
    a, p, N, n = config
    q = QuantumRegister(n)
    qc = QuantumCircuit(q)
    qc.append(gateMult(a, p, N, n), q)
    # Transpiler pour obtenir les portes elementaires
    start_time = time.time()
    transpiled_circ = transpile(qc, basis_gates=['id', 'ry', 'rx', 'cx'])
    if (time.time() - start_time) > 60:
        print(f"Temps d'execution trop long pour la configuration (a={a}, p={p}, N={N}, n={n})")
        break
    gate_count = dict(transpiled_circ.count_ops())
    # Calcul du nombre total de portes
    total_gates = sum(gate_count.values())
    print(f"Configuration (a={a}, p={p}, N={N}, n={n}): {total_gates} portes au total")
    print("Details des portes :", gate_count)
    print()
qc.draw(output='mpl')
```

4.4 Q2.3) Analyse:

1. **Quelles sont les tailles des circuits générés ?**
   La taille des circuits générés correspond au nombre total de portes élémentaires après transpilation. En utilisant différentes configurations de `gateMult` avec des valeurs croissantes de n, on observe que le nombre de portes augmente considérablement avec l'augmentation de n.

2. **Quelle est la complexité de la taille du circuit en fonction du nombre de qubits ?**
   La complexité de la taille des circuits générés est exponentielle en fonction du nombre de qubits n. Pour une transformation modulaire, chaque nouveau qubit double la taille de l'espace de Hilbert, ce qui entraîne une augmentation exponentielle du nombre de portes nécessaires pour représenter une transformation unitaire correcte.

3. **Pouvez-vous expliquer pourquoi ?**

   - **Opérations contrôlées** : Chaque qubit additionnel nécessite plus d'opérations contrôlées, ce qui augmente exponentiellement le nombre de portes.

- **Décomposition des portes** : Les portes quantiques de haut niveau comme la porte Toffoli nécessitent une décomposition en plusieurs portes élémentaires, ce qui ajoute de la complexité.
- **Complexité de l'arithmétique modulaire** : L'opération de multiplication modulaire doit être implémentée pour chaque état dans le registre. Plus le nombre de qubits est grand, plus l'opération devient complexe, car chaque qubit additionnel augmente les exigences computationnelles de la multiplication modulaire.

4. **Si c'est faisable pour un petit n, est-ce réaliste pour des tailles plus grandes ?**
Pour de petites valeurs de n, il est possible d'exécuter et de simuler les circuits générés en un temps raisonnable. Cependant, pour des valeurs de n plus grandes, la complexité exponentielle en termes de nombre de portes et de qubits rend ces circuits irréalistes à simuler ou à implémenter sur les ordinateurs quantiques actuels.

5. **Quelle méthode alternative pourriez-vous suggérer, et avec quels inconvénients potentiels ?**
Une méthode alternative pour gérer de grandes valeurs de n serait d'utiliser des algorithmes d'approximation qui réduisent la précision de l'opération modulaire, permettant ainsi de diminuer la taille du circuit en sacrifiant la précision.

   - **Inconvénients** :
     - L'approximation peut introduire des erreurs dans les résultats, ce qui pourrait affecter les applications nécessitant une précision élevée.
     - Les techniques d'optimisation de circuits ne garantissent pas toujours une réduction de la complexité de manière significative, surtout pour des opérations mathématiques complexes comme la multiplication modulaire.

5. Implémentation de l'Algorithme de Shor

5.1 Q3.1) Le circuit

Les quatre premiers qubits, nommés eig0 à eig3, constituent le registre de précision. Ils servent à stocker la phase estimée. Les portes Hadamard appliquées sur chaque qubit de eig placent ce registre dans un état de superposition, de sorte que chaque état binaire possible de eig soit exploré de manière égale. Le registre phi (composé de phi0 à phi4) représente l'état sur lequel on effectue l'estimation de phase. Dans l'algorithme de QPE, ce registre est initialisé dans un état propre de l'opérateur unitaire U dont on veut estimer la phase.

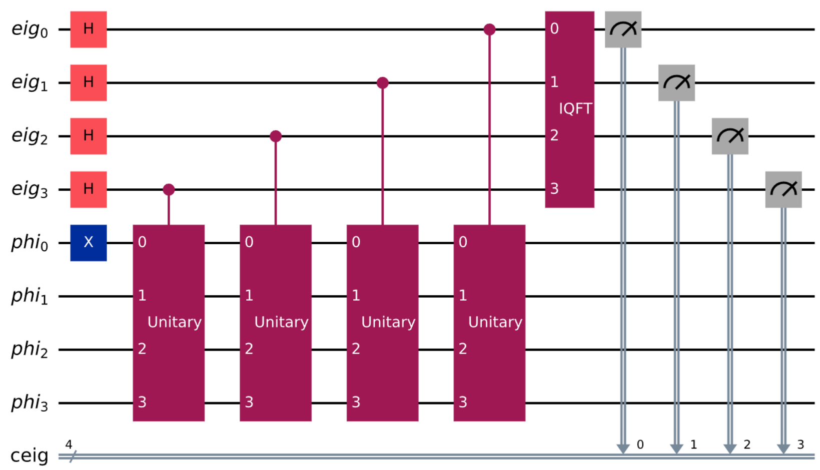


Figure 11: Circuit de 4 bits pour l'algorithme de Shor

```python
a = 7
p = 3
N = 15
n = 4
size_eig = 4
eig = QuantumRegister(size_eig, name="eig")
phi = QuantumRegister(n, name="phi")
ceig = ClassicalRegister(size_eig, name="ceig")
qc = QuantumCircuit(eig, phi, ceig)
qc.x(phi[0])  #  |0001>
qc.h(eig)
for i in range(size_eig):
    power = 2 ** i  # U^1, U^2, U^4, U^8
    controlled_mult_gate = gateMult(a, power, N, n).control(1)
    qc.append(controlled_mult_gate, [eig[size_eig - i - 1]] + list(phi))
qc.append(QFT(size_eig, inverse=True).to_gate(), eig)
qc.measure(eig, ceig)
qc.draw(output='mpl')
qc.draw(output='mpl')
simulator = AerSimulator()
job = simulator.run(transpile(qc, simulator))
job_result = job.result()
res = job_result.get_counts(qc)
print("Measurement results:", res)
qc.draw(output='mpl')

```

5.2 Q3-2) Résultats et Q3-3) Analyses

Le graphique montre un pic significatif autour de la valeur 9 sur l'axe des x, ce qui pourrait correspondre à une fraction de $\frac{s}{r}$. En utilisant 4 bits de précision, si nous avons un pic à la 9e position, cela pourrait être lié à la représentation binaire de $\frac{9}{16}$ pour une certaine approximation de $\frac{s}{r}$.

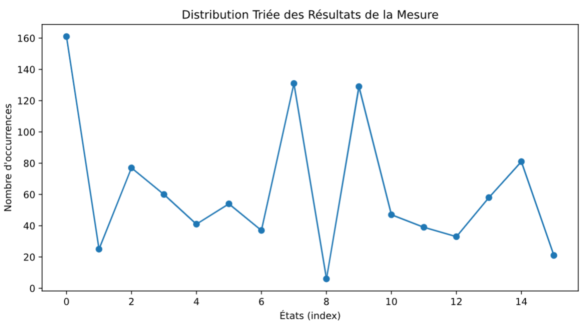


Figure 12: Premier résultat, r = 4-5

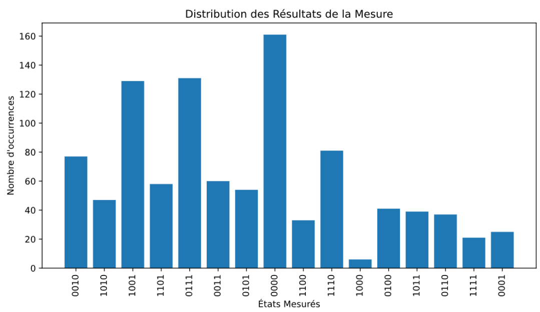


Figure 13: Histogramme

5.2.1 Quel est l'ordre r de $a \mod N$ (ici $7 \mod 30$)

Nous devonz utilser des données le circuit d'estimation de phase pour confirmer que $r = 4$ en observant les espacements et la répartition des pics dans les mesures. Ce calcul théorique de l'ordre est donc $r = 4$ pour $7 \mod 30$. Dans notre figure 13, on obtient bien cette ordre de grandeur 4-5.

5.2.2 Sur le graphique, où est-on censé voir les valeurs $\frac{s}{r}$? L'axe horizontal est gradué avec des entiers... À quels nombres réels compris entre 0 et 1 ces valeurs correspondent-elles ?

Les valeurs de $\frac{s}{r}$ correspondent aux valeurs fractionnaires de la phase, qui devraient apparaître sur l'axe horizontal du graphique. Cependant, l'axe horizontal est gradué en nombres entiers correspondant aux mesures binaires des qubits du registre de précision.

```
pts = []
for i in range(2**size_eig):
        k = nat2bs(size_eig,i)
        if k in d:
            pts.append((i,d[k]))
        else:
            pts.append((i,0))
def snd(a):
    (a1,a2) = a
    return a1
pts.sort(key = snd)
xs = []
ys = []
for i in range(len(pts)):
    (x,y) = pts[i]
    xs.append(x)
    ys.append(y)
plot(xs,ys)

```

Pour interpréter ces entiers en termes de valeurs réelles entre 0 et 1 :

- Considérons qu'avec n qubits de précision, nous avons 2^n niveaux de discrétisation pour l'axe horizontal.
- Chaque valeur entière k sur l'axe horizontal représente donc la fraction $\frac{k}{2^n}$.

Ainsi, chaque entier k entre 0 et $2^n - 1$ peut être mappé sur un nombre réel $\frac{k}{2^n}$ entre 0 et 1. Les pics dans le graphique à ces positions entières représentent les valeurs approximatives de $\frac{s}{r}$, où s et r sont des entiers liés à l'ordre de l'opération modulaire pour l'élément considéré.

5.2.3 Pouvez-vous déduire du graphique la valeur de r ? Où voyez-vous cette valeur sur le graphique ?

La valeur de r, qui représente l'ordre de a modulo N, peut être déduite en observant les positions des pics dans le graphique. Dans le contexte de l'estimation de phase, r est lié à l'espacement des pics significatifs. Si le graphe montre plusieurs pics espacés de manière régulière, la distance entre ces pics (en unités sur l'axe des x) correspond à $\frac{1}{r}$ en termes de la fraction de la phase. Par exemple, si nous observons des pics à des positions multiples de $k = \frac{2^n}{r}$, cela signifie que r est le nombre d'intervalles entre 0 et 2^n. La valeur r est donc visible sur le graphique en analysant la régularité et l'espacement des pics, ce qui donne une estimation de l'ordre de l'élément a modulo N.

Base (a)	Module (N)	Ordre de a mod N
2	3	2
3	4	2
2	5	4
3	5	4
4	5	2
5	6	2
2	7	3
3	7	6
4	7	3
5	7	6
6	7	2

Base (a)	Module (N)	Ordre de a mod N
3	8	2
5	8	2
7	8	2
2	9	6
4	9	3
5	9	6
7	9	3
8	9	2

5.2.4 Changez a et N respectivement à 20 et 29. Pouvez-vous lire la valeur de r ? Est-elle correcte ?

L'espace entre les pics sur l'axe des abscisses (indices) indique l'ordre r. Dans ce cas, si nous identifions un motif répétitif dans les résultats de mesure (par exemple, des pics à des intervalles de r indices), cet intervalle nous donne la valeur de r. Les principaux pics aux indices 0, 8, 16 et 24 nous suggèrent une périodicité de 8, indiquant que $r = 8$.

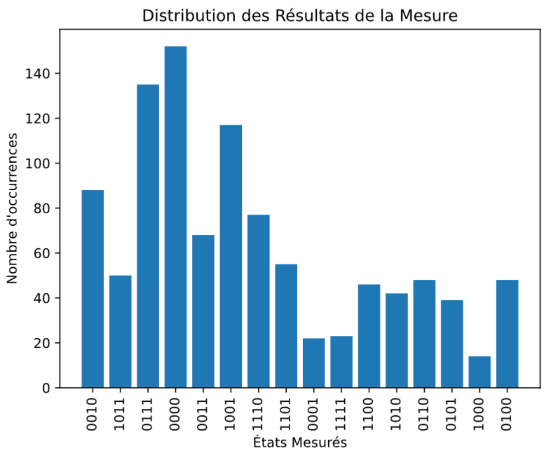


Figure 14: Histogramme pour a et N à 20 et 29

5.2.5 Le graphique n'est pas très précis... Comment l'améliorer ? Essayez !

Le registre de précision (`eig`) est défini avec 5 qubits. Ce changement augmente le nombre possible d'états pouvant être représentés, passant de $2^4 = 16$ états (si `size_eig` était 4) à $2^5 = 32$ états.

- En augmentant `size_eig`, l'algorithme QPE peut mieux résoudre de petites différences de phase. Chaque qubit supplémentaire dans le registre de précision double effectivement la résolution de phase.
- Une plus grande précision dans le registre `eig` permet au QPE de détecter et de représenter des détails plus fins dans la périodicité, ce qui conduit à une estimation plus précise de la phase et donc de l'ordre r.

La simulation est exécutée avec `shots=5000`, ce qui signifie que chaque mesure est répétée 5000 fois pour collecter davantage de données.

- Un plus grand nombre de tirs réduit le bruit statistique et augmente la fiabilité des résultats de mesure.

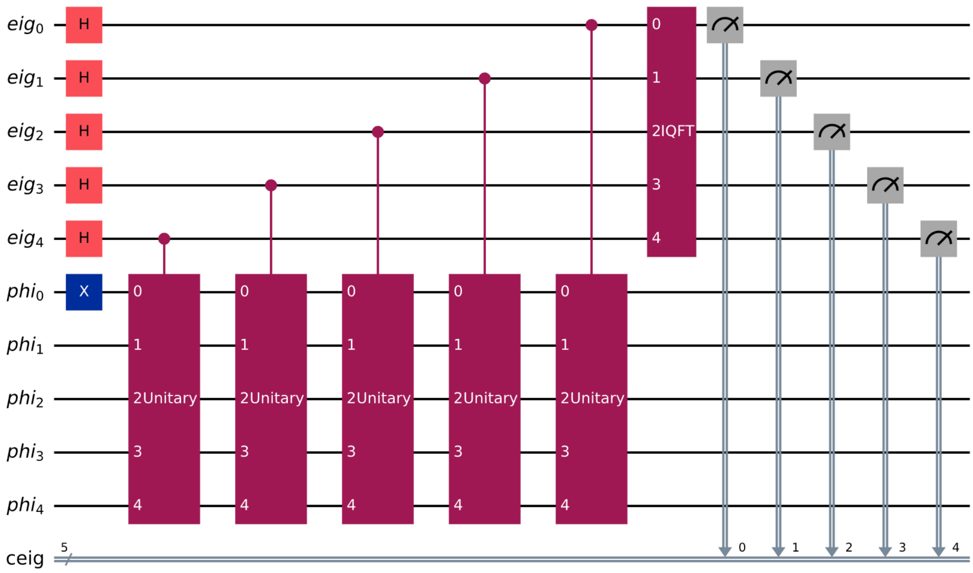


Figure 15: Circuit quantique 5 qubits

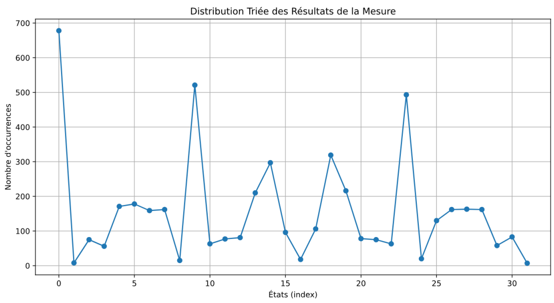


Figure 16: Résultat r = 8

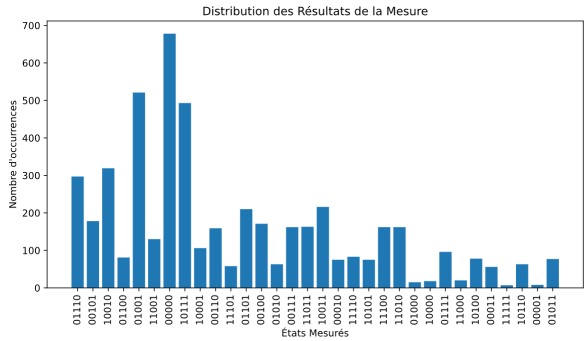


Figure 17: Histogramme pour a et N à 20 et 29

5.2.6 Est-ce que cela fonctionne toujours si vous changez la valeur de a et/ou de N pour d'autres valeurs ? Attention à ne pas utiliser des valeurs trop grandes pour N... Pour vous inspirer, ci-dessous se trouve la liste des possibilités jusqu'à 31

Le QPE peut être appliqué pour estimer la phase associée à une opération de multiplication modulaire, tant que l'ordre r de a modulo N est bien défini. Cela signifie que pour différentes valeurs de a et N, le code peut toujours être utilisé pour calculer l'ordre r à condition que N ne soit pas trop grand, car le nombre de qubits dans `size_eig` (précision) limite la capacité à détecter les périodes avec précision. Le choix de N doit être tel que $N \leq 2^n$, où n est le nombre de qubits dans le registre `phi` (ici défini à 5). Par exemple, pour $n = 5$, la valeur maximale de N est $2^5 - 1 = 31$. Si N dépasse 2^n, les calculs de multiplication modulaire risquent de ne pas être représentés correctement dans le registre `phi`, ce qui entraîne des erreurs. Pour que le QPE détecte un ordre périodique, a doit être un entier relativement premier à N (c'est-à-dire que le plus grand commun diviseur de a et N doit être 1). Si a et N ne sont pas copremiers, l'ordre peut ne pas être bien défini, ou bien les résultats du QPE peuvent ne pas correspondre à une période correcte.

References

[1] Qiskit. *Lab07_Scalable_Shor_Algorithm*. Available at: https://github.com/Qiskit/textbook/blob/main/notebooks/ch-labs/Lab07_Scalable_Shor_Algorithm.ipynb

[2] Benoît Valiron. *Introduction to Quantum Algorithms and Quantum Programming*, Course Notes v.2024.09.10.

[3] Qiskit. *Shor's Algorithm*. Available at: https://github.com/Qiskit/textbook/blob/main/notebooks/ch-algorithms/shor.ipynb

[4] Qiskit. *Quantum Phase Estimation*. Available at: https://github.com/Qiskit/textbook/blob/main/notebooks/ch-algorithms/quantum-phase-estimation.ipynb

[5] Quantum Exo7. *L'algorithme de Shor*. Available at: https://exo7math.github.io/quantum-exo7/shor/shor.pdf

```
def nat2bl(pad,n):
    if n == 0 :
        return [0 for i in range(pad)]
    elif n % 2 == 1:
        r = nat2bl(pad-1,(n-1)//2)
        r.append(1)
        return r
    else:
        r = nat2bl(pad-1,n//2)
        r.append(0)
        return r
for i in range(16):
    print(nat2bl(5,i))
def bl2nat(s):
    if len(s) == 0:
        return 0
    else:
        a = s.pop()
        return (a + 2*bl2nat(s))
def bl2bs(l):
    if len(l) == 0:
        return ""
    else:
        a = l.pop()
        return (bl2bs(l) + str(a))
def nat2bs(pad,i):
    return bl2bs(nat2bl(pad,i))
def bs2bl(s):
    l = []
    for i in range(len(s)):
        l.append(int(s[i]))
    return l
def bs2nat(s):
    return bl2nat(bs2bl(s))
print(nat2bs(10,17))
print(bs2nat("0011010"))
```

```python

a = 20
p = 3
N = 29
n = 5

size_eig = 5
eig = QuantumRegister(size_eig, name="eig")
phi = QuantumRegister(n, name="phi")
ceig = ClassicalRegister(size_eig, name="ceig")
qc = QuantumCircuit(eig, phi, ceig)

qc.x(phi[0])  #  |00001>
qc.h(eig)
for i in range(size_eig):
    power = 2 ** i  # U^1, U^2, U^4, U^8,
    controlled_mult_gate = gateMult(a, power, N, n).control(1)
    qc.append(controlled_mult_gate, [eig[size_eig - i - 1]] + list(phi))

qc.append(QFT(size_eig, inverse=True).to_gate(), eig)
qc.measure(eig, ceig)
qc.draw(output='mpl')

simulator = AerSimulator()
job = simulator.run(transpile(qc, simulator), shots=100000)  # Increased number of shots
job_result = job.result()
res = job_result.get_counts(qc)

print("Measurement results:", res)

labels, values = zip(*res.items())
indices = np.arange(len(labels))
plt.figure(figsize=(10, 5))
plt.bar(indices, values, tick_label=labels)
plt.xlabel(' tats  Mesur s')
plt.ylabel("Nombre d'occurrences")
plt.title('Distribution des R sultats de la Mesure')
plt.xticks(rotation=90)
plt.show()

pts = []
for i in range(2**size_eig):
    k = format(i, f'0{size_eig}b')
    if k in res:
        pts.append((i, res[k]))
    else:
        pts.append((i, 0))

pts.sort(key=lambda a: a[0])

xs = [x for x, y in pts]
ys = [y for x, y in pts]

plt.figure(figsize=(12, 6))
plt.plot(xs, ys, marker='o')
plt.xlabel(' tats  (index)')
plt.ylabel("Nombre d'occurrences")
plt.title('Distribution Tri e des R sultats de la Mesure')
plt.grid(True)
plt.show()
qc.draw(output='mpl')
```

```
size_eig = 4

simulator = AerSimulator()
job = simulator.run(transpile(qc.decompose(reps=6), simulator), shots=1000)
job_result = job.result()
res = dict(job_result.get_counts(qc))

print("R sultats de mesure (d composition 6):", res)

job = simulator.run(transpile(qc.decompose(reps=7), simulator), shots=1000)
job_result = job.result()
res = dict(job_result.get_counts(qc))

print("R sultats de mesure (d composition 7):", res)

labels, values = zip(*res.items())
indices = np.arange(len(labels))
plt.figure(figsize=(10, 5))
plt.bar(indices, values, tick_label=labels)
plt.xlabel(' tats  Mesur s')
plt.ylabel("Nombre d'occurrences")
plt.title('Distribution des R sultats de la Mesure')
plt.xticks(rotation=90)
plt.show()
pts = []
for i in range(2**size_eig):
    k = format(i, f'0{size_eig}b')
    if k in res:
        pts.append((i, res[k]))
    else:
        pts.append((i, 0))

pts.sort(key=lambda a: a[0])
xs = [x for x, y in pts]
ys = [y for x, y in pts]

plt.figure(figsize=(10, 5))
plt.plot(xs, ys, marker='o')
plt.xlabel(' tats  (index)')
plt.ylabel("Nombre d'occurrences")
plt.title('Distribution Tri e des R sultats de la Mesure')
plt.show()
```

```python
a = 20
p = 3
N = 29
n = 5

size_eig = 7
eig = QuantumRegister(size_eig, name="eig")
phi = QuantumRegister(n, name="phi")
ceig = ClassicalRegister(size_eig, name="ceig")
qc = QuantumCircuit(eig, phi, ceig)

qc.x(phi[0])
qc.h(eig)

for i in range(size_eig):
    power = 2 ** i  # U^1, U^2, U^4, U^8,
    controlled_mult_gate = gateMult(a, power, N, n).control(1)
    qc.append(controlled_mult_gate, [eig[size_eig - i - 1]] + list(phi))

qc.append(QFT(size_eig, inverse=True).to_gate(), eig)
qc.measure(eig, ceig)
qc.draw(output='mpl')

simulator = AerSimulator()
job = simulator.run(transpile(qc, simulator), shots=100000)
job_result = job.result()
res = job_result.get_counts(qc)

print("Measurement results:", res)

labels, values = zip(*res.items())
indices = np.arange(len(labels))
plt.figure(figsize=(10, 5))
plt.bar(indices, values, tick_label=labels)
plt.xlabel(' tats  Mesur s')
plt.ylabel("Nombre d'occurrences")
plt.title('Distribution des R sultats de la Mesure')
plt.xticks(rotation=90)
plt.show()

pts = []
for i in range(2**size_eig):
    k = format(i, f'0{size_eig}b')
    if k in res:
        pts.append((i, res[k]))
    else:
        pts.append((i, 0))

pts.sort(key=lambda a: a[0])
xs = [x for x, y in pts]
ys = [y for x, y in pts]

plt.figure(figsize=(12, 6))
plt.plot(xs, ys, marker='o')
plt.xlabel(' tats  (index)')
plt.ylabel("Nombre d'occurrences")
plt.title('Distribution Tri e des R sultats de la Mesure')
plt.grid(True)
plt.show()
qc.draw(output='mpl')
```