Quantum Factorization

Chapter 1

Einführung

1.1 Einführung ?

1.2 Mein Produkt - Die QInteger- und die QAlgorithm-Libraries und ein Faktorisierungs-Algorithmus

Das Produkt dieser Maturaarbeit sind zwei Libraries, die Algorithmen für Quanten-COmputer implementieren:

- Die QInteger-Library, welche Zahlen auf Quantencomputer implementiert, grundlegende arithmetische Operationen bereitstellt, und viele nützliche Operationen für jene Quantenzahlen implementiert.
- Die QAlgorithms-Library, welche Algorithmen für diese Quantenzahlen implementiert, aber vor allem auf die Implementierung von Shors Algorithmus zur Faktorisierung hinarbeitet, welcher ohne Frage einer der nennenswertesten Erfolge der Quantencomputer ist.

Diese beiden Libraries sind mit dem Ziel, generelle und nützliche Funktionen für Quantencomputer bereitzustellen, welche man auch für andere Projekte gebrauchen kann, implementiert. Diese Libraries sind in der Programmiersprache Q# von Microsoft geschrieben. Den Code zu diesen Libraries habe ich im Appendix angefügt und man kann ihn auch auf GitHub finden, wo er auch einige Qktualisierungen erhalten wird. Auf GitHub ist er unter folgendem Link verfügbar: https://github.com/johutha/QInteger-QAlgorithms

Ich habe Q# gewählt, da es momentan eine der populärsten Quantenprogrammiersprachen ist, eine gute Dokumentation hat, gut unterstützt wird und regelmässig aktualisiert wird. Gleichzeitig kann der Compiler automatische zu Quantenoperation deren Inverses oder deren kontrollierte Version generieren, was den Code gleich kürzer und übersichtlicher macht.

Neben diesen beiden Libraries finden sich im GitHub noch weitere Projekte. Eines davon implementiert den "Factorizer", welcher mit Hilfe einer Blackbox-Operation, die die Ordnung einer Zahl findet und einer Blackbox-Operation, die überprüft, ob eine Zahl eine Primzahl ist, den kompletten Faktorisierungs-Algorithmus. Dazu stehen verschiedene Primzahltester- und Ordnungs-Finder-Module zur Verfügung, so dass man die verschiedenen Algorithmen vergleichen kann. Natürlich ruft eines jener Module den Algorithmus die Quantenoperation auf, so dass der Faktorisierer zusammen mit jenem Modul eine komplette Implementierung von Shors Algorithmus ist. Weiterhin findet sich darunter auch ein Projekt, welches eine einfache Konsolen-Applikation

implementiert, welche diese Libraries benützt, als Proof-Of-Concept, und ein Projekt, welches die Zeit misst, die der Faktorisierer braucht.

Natürlich muss ich überprüfen, ob diese Implementierungen funktionieren, deshalb habe ich drei Unit-Test-Projekte, die die verschiedenen Komponenten einzeln testen. Das Konzept der Unit-Tests erlaubt ein einfaches Lokalisieren von Implementationsfehlern und Bugs und fangen fast alle Fehler ein.

Chapter 2

Grundlagen

2.1 Lineare Algebra

2.2 Quantensysteme

2.2.1 Qubits

Die Quantensysteme, die wir im Bereich des Quantum Coputing anschauen, sind rein mathemathematische Systeme, die auf keiner fixen physikalische Realisierung basieren. Dies bedeutet, dass es verschiedene Implementierungen gibt, die sich je nach Situation besser oder weniger gut eignen.

Definition 2.2.1: Ein **Qubit** ist das kleinste Quantensystem und damit die kleinste Informationseinheit in einem Quantencomputer. Das System hat die beiden Basiszustände $|0\rangle$ und $|1\rangle$ und kann somit alle Zustande $\alpha|0\rangle + \beta|1\rangle$ mit $|\alpha|^2 + |\beta|^2 = 1$ annehmen.

Falls α und β beide nicht 0 sind, dann ist das Qubit gleichzeitig $|0\rangle$ und $|1\rangle$. Dies nennt man eine Superposition.

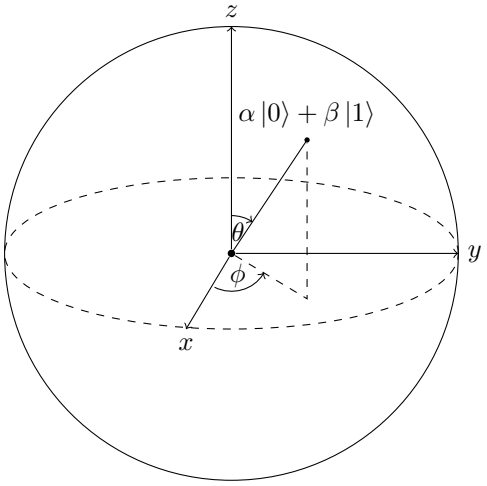
Den Zustand $\alpha|0\rangle + \beta|1\rangle$ kann man auch mit einem Vektor $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ darstellen. Die Bedingung, dass $|\alpha|^2 + |\beta|^2 = 1$ gelten muss, bedeutet, dass der Vektor normiert sein muss. Die Chance, den Zustand $|0\rangle$ zu messen, $|\alpha|^2$, und für den Zustand $|1\rangle$ gleich $|\beta|^2$ ist. Hier sehen wir wieder, dass der Vektor normiert sein muss, denn sonst würden sich die Wahrscheinlichkeiten nicht auf 1 summieren. Genereller lassen sich Zustände, die n Qubits involvieren, als Vektoren von Grösse 2^n darstellen. Sei a_j der j-te Eintrag in jenem Vektor. Die Chance, den Zustand j zu messen, wobei das k-te Qubit dann im Zustand $\lfloor \frac{j}{2^k} \rfloor \pmod 2$ ist (was einfach dem k-ten Bit in der Binärdarstellung von j entspricht), ist dann $|\alpha_j|^2$. Auch hier summieren sich die Wahrscheinlichkeiten auf 1, da der Vektor normiert ist. Man kann diese Zustände aber auch in Form eines mathematischen Ausdrucks $\alpha_0|0\rangle + ... + \alpha_j|j\rangle + ... + \alpha_{2^n-1}|2^n - 1\rangle$ oder $\alpha_{00...}|00\ldots\rangle + ... + \alpha_j|j\rangle + ... + \alpha_{11...}|11\ldots\rangle$ darstellen, wobei in der zweiten Schreibweise j ein Bitstring von Länge n ist.

Gehen wir zurück zum einzelnen Qubit im Zustand $\alpha|0\rangle + \beta|1\rangle$. Wir wissen schon, dass wir bei einer Messung mit einer Wahrscheinlichkeiten von $|\alpha|^2$ den Zustand $|0\rangle$ messen. Nach dieser Messung kollabiert das Quantensystem in den gemessenen Zustand. Das heisst, messen wir den Zustand $|0\rangle$, befindet sich das Quantensystem nachher im Zustand $|0\rangle$, egal, wie α und β vorher waren. Dies gilt auch für Multi-Qubit Systeme. Diese kollabieren dann in die

noch möglichen Quantenpositionen. Nehmen wir als Beispiel ein 2-Qubit-System im Zustand $\frac{1}{\sqrt{6}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{3}}|11\rangle$ und messen das erste Qubit. Die Chance, dass wir dieses Qubit im Zustand $|1\rangle$ messen, liegt bei $|\alpha_{10}|^2 + |\alpha_{11}|^2 = \frac{1}{3}$. Falls wir diesen Zustand messen, kollabiert unser Quantensystem sofort in den Zustand $\frac{\alpha_{10}|10\rangle + \alpha_{11}|11\rangle}{\sqrt{|\alpha_{10}|^2 + |\alpha_{11}|^2}} = |11\rangle$, wobei die Summe im Nenner dafür sorgt, dass der neue Quantenzustand wieder normalisiert ist. Die Chance eines $|0\rangle$ in der Messung des ersten Qubit hingegen liegt bei $|\alpha_{00}|^2 + |\alpha_{01}|^2 = \frac{2}{3}$. Der Zustand des Systems nach der Messung ist dann $\frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} = \sqrt{\frac{1}{4}}|00\rangle + \sqrt{\frac{3}{4}}|01\rangle$. Dass diese Zustände rein mathematisch durch Vektoren darstellen lassen, hat zur Folge, dass man Quanten verschränken kann. Dafür schauen wir uns den einfach realisierbaren Zustand $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Nun platzieren wir das eine Qubit auf die eine Seite des Universums, und das andere Qubit auf die andere Seite. Dann messen wir das erste Qubit. Die beiden Zustände $|0\rangle$ und $|1\rangle$ haben dabei die gleiche Wahrscheinlichkeit. Das andere Qubit auf der anderen Seite des Universums, kollabiert darauf sofort in den gleichen Zustand wie das erste. Ich habe dabei die beiden Seiten des Universums gewählt, um darauf aufmerksam zu machen, dass diese Qubits miteinander verknüpft sind, und dabei physikalische Distanz keine Rolle spielt.

2.2.2 Die Blochkugel

Die Blochkugel ist ein Instrument, um den Zustand eines einzelnen Qubits graphisch darzustellen. Dazu nehmen wir nochmals ein einzelnes Qubit $\alpha|0\rangle + \beta|1\rangle$ mit $|\alpha|^2 + |\beta|^2 = 1$. Diese Bedingung führt dazu, dass wir den Zustand als $e^{i\gamma}\left(cos\frac{\theta}{2}|0\rangle + e^{i\varphi}sin\frac{\theta}{2}|1\rangle\right)$. Der Faktor $e^{i\gamma}$ können wir nicht beobachten, da er auf alle Koeffizienten wirkt. Deshalb ist der Zustand durch die beiden Winkel θ und φ definiert. Diese beiden Winkel kann man graphish als einen Punkt auf der Einheitskugel darstellen.



2.2.3 Operationen auf Qubits

Die Operationen, die man auf Qubits implementiert, sind lineare Operationen und lassen sich deshalb als Matrizen darstellen. Dann kann man den Zustandsvektor mit der Matrix des Operators multiplizieren, um den Zustand nach der Operation zu bekommen. Weiterhin entwickeln sich Quantensysteme nach einem Postulat der Quantenmechanik die Schrödinger-Gleichung erfüllend, was dazu führt, dass die implementierten Operatoren unitär sein müssen. Dies hat die direkte

Konsequenz (für eine unitäre Matrix U gilt $UU^\dagger = I$), dass ein inverser Operator existieren muss, und deshalb alle Berechnungen reversibel sein müssen. Dies führt dazu, dass der Modulo Operator nicht auf Quanten implementiert werden kann, da man aus dem Ergebnis $x \equiv 2 \pmod 3$ die Eingabe $x \in \{2, 5, 8 \cdots\}$ nicht eindeutig wiederherstellen kann. Dies hat grosse Konsequenzen für die Berechnungen auf Quantencomputer. Gleichzeitig stellt sich heraus, dass sich alle unitäre Matrizen sich beliebig annähern lassen. Die Konstruktion dazu kann man in [QC], Seiten 188ff., nachlesen.

2.2.4 Wichtige Quantengatter

Hier sollen kurz die wichtigsten Quantengatter eingeführt werden, die wir benötigen werden. Zuerst schauen wir uns 5 grundlegende Gatter auf Qubits an, die 3 Pauli-Matrizen, das H-Gatter und das $CNOT$-Gatter.

- Das X-Gatter ist das Qubit-Equivalent zum NOT-Gatter. In Matrixform sieht der Operator so aus: $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Dieses Gatter dreht den Zustand des Qubits um π um die x-Achse in der Blochkugel.
- Das Y-Gatter implementiert die Operation der Matrix $Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$. Dieses Gatter entspricht einer Rotation von π um die y-Achse in der Blochkugel.
- Das Z-Gatter, als Matrix $Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$, dreht den Zustand um π um die z-Achse.
- Das H-Gatter ist der einfachste Weg, eine Superposition zu erzeugen. Mit der Matrix $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ kann man ganz die beiden Zustände $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ und $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ erzeugen. Diese beiden Zustände kommen so oft vor, dass man ihnen die Namen $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ und $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ gegeben hat.
- $CNOT$ steht als Abkürzung für "Controlled NOT". Dieses Gatter wirkt auf zwei Qubits und wendet ein NOT auf das zweite Qubit an, wenn das erste Qubit auf 1 ist. Als Matrix sieht die Operation so aus: $CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

Diese Gatter sind die wichtigsten Gatter im Bereich des Quantum Computing. Wir werden jedoch auf unserem Weg noch weitere Gatter antreffen. Eines davon, von dem wir noch mehr Gebrauch machen werden, möchte ich hier noch kurz definieren. Dieses Gatter nenne ich $Rot(k)$ und in Matrix-Form sieht es so aus: $Rot(k) = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2i\pi}{2^k}} \end{bmatrix}$. Dieses Gatter multipliziert den Koeffiztienten von $|1\rangle$ mit $e^{\frac{2i\pi}{2^k}}$ und wir werden es bei der Quanten-Fouriertransformation und dessen Anwendungen antreffen.

Zuletzt möchte ich noch ein Gatter auf zwei Qubits erwähnen, bekannt als das SWAP-Gate. Dieses Gate wechselt die Zustände der beiden Qubits. Dieses Gatter implementiert die Matrix:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Die rechte Seite zeigt uns, dass wir $SWAP(A, B)$ als $CNOT(A, B)$, $CNOT(B, A)$, und $CNOT(A, B)$ implementieren können.

Chapter 3

Arithmetische Operation auf Qubits ausführen - Die QInteger Library

3.1 Überblick

3.2 Zahlen in Qubits speichern - Der QInt-Typ

Da ich davon ausgehe, dass in nächster Zeit die Anzahl Qubits zwar wachsen wird, aber nicht so schnell ansteigen, dass man schon bald mehrere grössere Qubit-Einheiten speichern kann, habe ich mich entschieden, in meiner Implementation auf eine einheiltiche Grösse zu verzichten. Deshalb der QInt-Typ aus einer klassischen Zahl, die Anzahl Qubits, und einem Array von Qubits, welcher die eigentliche Zahl speichert. Ich habe mich auch dazu entschieden, die Quantenzahl im Little-Endian Format zu speichern, da so neue Qubits einfach angehängt werden können ohne den Wert der Zahl zu verändern.

```
// Definition of the QInt type with variable size. QInts
// are represented in little-endian.
newtype QInt = (Size : Int, Number : Qubit[]);
```

3.3 Die Quanten-Fouriertransformation und die Fourier-Basis

Die Quanten-Fouriertransformation ist eine Transformation, die eine Quantenzahl von der uns bekannten binären Basis in die Fourierbasis transformiert. Die Fouriertransformation, die dabei auf den Qubits implementiert ist, ist mathematisch definiert als eine Transformation, die zu einem Vektor $(x_0, x_1, ..., x_{n-1})$ zum Vektor $(y_0, y_1, ..., y_{n-1})$ transformiert, mit $y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j e^{2i\pi \frac{kj}{n}}$. Da dies ein linearer Operator ist, genügt es, wenn wir uns die Wirkung des Operators auf den Basiszuständen anschauen. Schauen wir also die Wirkung des Operators auf den Basiszustand

$|x\rangle$ an. Wir erhalten:

$$QFT\,|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} e^{2i\pi\frac{xj}{2^n}}\,|j\rangle$$

Gleichzeitig lässt dieser Zustand sich faktorisieren, nämlich zu:

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

Dies kann man durch ausmultiplizieren beweisen. Um die folgende Gleichung zu vereinfachen, sei hier $b_k(j) = 1$ falls das k-te Bit von j gesetzt ist, und $b_k(j) = 0$ falls nicht. Dazu sie B_j als das Set aller $k \in \mathbb{N}_0$ mit $b_k(j) = 1$. Dann bekommen wir:

$$\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} \left(|0\rangle + e^{2i\pi\frac{x}{2^{n-j}}}\,|1\rangle\right) = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} \left(\prod_{k\in B_j} e^{2i\pi\frac{x}{2^{n-k}}} \right) |j\rangle$$

$$= \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} e^{2i\pi\sum_{k=0}^{n-1}\left(\frac{x\cdot b_k(j)}{2^{n-k}}\right)}\,|j\rangle = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} e^{2i\pi\frac{x\sum_{k=0}^{n-1}\left(2^k\cdot b_k(j)\right)}{2^n}}\,|j\rangle$$

$$= \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} e^{2i\pi\frac{xj}{2^n}}\,|j\rangle$$

Was bringt uns diese Faktorisierung? Zuerst stellen wir fest, dass die Bits unabhängig und nicht verschränkt sind. Gleichzeitig schauen wir uns die einzelnen Qubits mit Hilfe der Blochkugel an. Wir stellen fest, der Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\theta}\,|1\rangle)$ in der Blochkugel einer Rotation von θ Grad um die Z-Achse in der XY-Ebene entspricht. Schauen wir uns die einzelnen Bits an, entspricht das j-te Bit einer Rotation von $\frac{x}{2^{n-j}}$ um die Z-Achse. Dies ist die sogenannte Fourier-Basis.

Nun kann man sich überlegen, was denn der Grosse Vorteil der Fourier-Basis ist. Die Fourierbasis hat verschiedene Vorteile. Zum Beispiel werden wir im Kapitel 4.3 sehen, dass wenn wir in einer Operation ein Qubit um θ um die Z-Achse drehen, was der Multiplikation des Koeffizienten von $|1\rangle$ mit dem Wert $e^{2i\pi\theta}$ entpricht, machen wir nichts anderes, als den Wert der Qubits in der Fourierbasis zu verändern. Später können wir dann die inverse QFT anwenden, um den Wert θ als Binärzahl auslesen zu können. Ein anderer grosser Vorteil der Fourierbasis ist, dass die einzelnen Qubits voneinander unabhängig sind. Dass dies ein grosser Vorteil ist, werden wir feststellen, wenn wir die Addition auf Qubits implementieren.

Die letzte Frage, die es nun noch zu klären gilt, ist, wie man diese Transformation nun implementiert. Wir werden sehen, wie man die Transformation mit $\mathcal{O}(n^2)$ Gatteroperationen implementieren kann, ohne zusätzliche Qubits. Schauen wir uns nochmals die Faktorisierung an. Wir stellen fest, dass das letzte Qubit in der binären Basis nur das erste Qubit in der Fourierbasis beeinfluss, da 2^{n-1} in x im Term $e^{2i\pi\frac{x}{2^a}}$ mit $a \leq n-1$ nur ganze Rotationen im Einheitskreis hinzufügt, was den Wert nicht beeinflusst. Weiterhin beeinflusst das zweitletzte Qubit in der binären Basis nur die beiden ersten Qubits in der Fourierbasis etc. Kehren wir die Reihenfolge der Qubits der Fourierbasis um. Wir bekommen $\bigotimes_{j=0}^{n-1}\left(|0\rangle + e^{2i\pi\frac{x}{2^{j+1}}}\,|1\rangle\right)$. Dann beeinflusst

jedes Qubit in der binären Basis nur das gleiche und alle nachfolgenden Qubits in der Fourierbasis. Dies können wir gebrauchen, um die Qubits von hinten nach vorbe in die Fourierbasis zu bringen, so dass wir sie dann einfach nur wieder in die richtige Reihenfolge bringen, was man ganze infach mit dem $SWAP$-Operator machen kann. Nun schauen wir das Qubit j an. Alle Qubits nach j sind schon in der Fourierbasis und alle vorher noch nicht. Zuerst wenden wir den H-Operator auf das Qubit an. Wir bekommen den Zustand $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ falls $b_j(x) = 0$ und $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ falls $b_j(x) = 1$. Dies ist nichts anderes als $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi \frac{b_j x 2^j}{2^{j+1}}} |1\rangle)$, was den Beitrag vom j-tzen Qubit in der binären Basis an das j-te Qubit in der umgekehrten Fourierbasis ist. Nun müssen wir nur noch den Beitrag aller Qubits vor dem j-ten Qubit dazurechnen. Das l-te Qubit mit $l < j$ soll $e^{2i\pi \frac{b_l(x)2^l}{2^{j+1}}}$ dazu beitragen. Das $b_l(x)$ lässt sich umsetzen so umsetzuen, dass wir das l-te Qubit in der binären Basis als Kontroll-Qubit für die Operation nehmen, die $e^{2i\pi \frac{2^l}{2^{j+1}}}$ zum Koeffizienten von $|1\rangle$ dazurechnet. Dies ist nichts anderes als das $Rot(k)$-Gatter, mit $k = (j+1)-l$. Damit haben wir eine Implementation für die Quanten-Fouriertransformation, die $\mathcal{O}(n^2)$ Gatteroperationen und keine zusätzlichen Qubits benötigt.

```
// Implements the Quantum Fourier Transform

internal operation PerformQFT(qn : QInt) : Unit is Adj+Ctl
{
        let n = qn::Size;
        for (i in n-1..-1..0)
        {
                H(qn::Number[i]);
                for (j in i - 1..-1..0)
                {
                    (Controlled RotQ)([qn::Number[j]],
                                      (i - j + 1, qn::Number[i]));
                }
        }
        ReverseBits(qn);
}
```

3.4 Addition

Die wohl grundlegendste arithmetische Operation ist die Addition. Die Subtraktion kann als Addition ausgedrückt werden, und auch die Multiplikation (somit auch die Division) sind abhängig von der Addition. Deshalb ist es die erste arithmetische Operation, die wir uns hier anschauen. Wir wollen dabei die Operation auf zwei QInts implementieren, welche zwei QInts im Zustand $(|x\rangle, |y\rangle)$ in den Zustand $(|x\rangle, |x+y\rangle)$ transformiert. Die Implementation anderer Additionsmethoden (Addition einer klassischen Zahl zu einem QInt, Addition zweier QInts in ein drittes QInt) funktionieren Analog. Zusätzlich kann man auch sehen, dass die Subtraktion nichts anderes als die inverse Operation zur Addition ist, somit hat man zur Addition gleich noch die Subtraktion mit-implementiert.

Heutzutage sind zwei verschiedene Additions-Techniken bekannt. Die eine benutzt zusätzliche Carry-Bits, und erreicht so eine Gatterzahl in $\mathcal{O}(n)$, braucht dafür aber $\mathcal{O}(n)$ zusätzliche Qubits, während die andere ohne zusätzliche Qubits auskommt, dafür aber $\mathcal{O}(n^2)$ zusätzliche Gatterop-

erationen benötigt. Ich habe mich entschieden, für den Moment die zweite Version in meiner QInteger-Library zu implementieren. Gründe dazu sind, dass in heutigen Systemen die Anzahl verfügbarer Qubits stark begrenzt sind und in Simulationen einzelne Qubits sehr viel Leistung kosten, während eine Laufzeit von $\mathcal{O}(n^2)$ in diesem Fall weniger ausmacht. Wenn dann aber mehr Qubits zur Verfügung stehen, wird es wahrscheinlich lohnenswerter, auf die andere Version zu wechseln, denn da Addition eine Operation auf einem sehr tiefen Level ist, kann die Zeit, welche die Addition benötigt, sehr grosse Auswirkungen auf die gesamte Laufzeit haben.

Schauen wir uns nun den in der QInteger-Library verwendete Additionsalgorithmus an. Der Algorithmus basiert auf der Fourierbasis (und damit auf der Faktorisierung der Fouriertransformation). Bei der Addition in der binären sind die einzelnen Bits voneinander abhängig. Deshalb werden sogenannte Carry-Bits verwendet, welche für jedes Bit abspeichern, ob wir beim nächsten Bit noch ein zusätzliches 1 addieren müssen. Dies ist bei der Fourierbasis nicht so: Die Bits sind voneinander unabhängig. Das heisst, wir können die einzelnen Bits voneinander unabhängig modifizieren, ohne dabei auf die anderen Bits achten zu müssen. Dies ist der grosse Vorteil der Fourier-Basis, welcher uns erlaubt, auf zusätzliche Qubits zu verzichten. Schauen wir uns nochmals die Faktorisierung an: Das j-te Qubit der Zahl y in der Fourierbasis ist im Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y}{2^{n-j}}}|1\rangle)$. Wir wollen es aber in den Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{x+y}{2^{n-j}}}|1\rangle)$ bringen, denn wenn wir alle Qubits in den entsprechenden Zustand bringen könnten, könnten wir mit der inversen QFT den Zustand $|x+y\rangle$ wiederherstellen. Dies ist aber nicht zu schwierig. Nehmen wir wieder das aus der Fouriertransformation bereits bekannte Gatter $Rot(k) = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0\\ 0 & e^{\frac{2i\pi}{2^k}}\end{bmatrix}$. Mit dem Gatter können wir den Wert $2i\pi\frac{1}{2^k}$ dem Exponenten von $|1\rangle$ hinzufügen. Das heisst, wenn wir das Gatter auf ein Qubit im Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y}{2^{n-j}}}|1\rangle)$ anwenden, wird es in den Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y+2^{n-j-k}}{2^{n-j}}}|1\rangle)$ versetzt. Wir können also mit Hilfe dieses Gatters Zweierpotenzen dem Qubit in der Fouriertransformation addieren. Wenn wir also das Qubit im Zustand $|x\rangle$ in der binären Basis lassen, können wir die Addition wie folgt mit $\mathcal{O}(n^2)$ Gatteroperationen implementieren:

1. Wende QFT auf den zweiten Summanden im Zustand $|y\rangle$ an. Das Register is nun im Zustand $\frac{1}{\sqrt{2^n}}\bigotimes_{j=0}^{n-1}\left(|0\rangle + e^{2i\pi\frac{y}{2^{n-j}}}\right)$.

2. Für das jedes j-te Bit im zweiten Register, wende für jedes k-te Bit im ersten (binären) Register mit $k < n-j$ ein kontrolliertes $Rot(n-i-j)$ an. Das j-te Bit befindet sich nachher im Zustand

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y+\sum_{k=0}^{n-j-1} b_k(x)\cdot 2^{n-j-(n-j-k)}}{2^{n-j}}}|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y+x}{2^{n-j}}}|1\rangle)$$

   Wobei alle Bits höher als 2^{n-j-1} uns nicht interessieren, da sie alle Vielfaches von 2^{n-j} sind, und somit nur gane Umrundungen zur Rotation hinzufügen.

3. Die Qubits im zweiten Register befinden sich nun in folgendem Zustand: $\frac{1}{\sqrt{2^n}}\bigotimes_{j=0}^{n-1}\left(|0\rangle + e^{2i\pi\frac{x+y}{2^{n-j}}}\right)$. Mit der inversen QFT kann man nun aus diesem Zustand den Zustand $|x+y\rangle$ wiederherstellen.

3.5 Modulare Addition

Bei Shor braucht man aber nicht nur Addition, sondern modulare Addition. Den uns bekannten Modulo-Operator kann man auf Qubits nicht implementieren, da er nicht reversibel ist (a und

$a + m$ haben das selbe Resultat Modulo m). Die modulare Addition ist jedoch reversibel, wenn die Summanden kleiner als das Modulo sind. Dafür benutzen wir Information über die beiden Summanden. Dafür haben wir wieder drei Register in den Zuständen $|x\rangle, |y\rangle$ und $|m\rangle$, und möchten sie in den Zustand $|x\rangle, |(x + y)\%m\rangle$ und $|m\rangle$. Theoretisch kann man das dritte Register durch ein klassisches Register ersetzten, wenn das m eine klassische Zahl ist. Für den Algorithmus von Shor werden wir auch nur die modulare Addition modulo eine klassische Zahl brauchen, aber um aufzuzeigen, dass man es auch mit QInt-Modulos implementieren kann, präsentiere ich hier diese Version. Die andere Implementation folgt analog.

Zuerst addieren wir x zum Register $|y\rangle$, um das Register in den Zustand $|x + y\rangle$ zu versetzen. Nun überprüfen wir, ob diese Summe grössergleich dem Modulo m ist.

Wie überprüfen wir ob eine Zahl grössergleich einer anderen Zahl ist? Sagen wir, ob $|A\rangle$ grössergleich $|B\rangle$ ist, wobei B auch eine normale Zahl sein könnte? In der QInteger-Library ist die Funktion *GreaterOrEqual* für $A >= B$ als $\neg$*LessThan* implementiert. Hier unterscheiden sich die Implementationen für de Fälle wenn B ein QInt oder eine klassische Zahl ist, sie machen jedoch ungefähr das Gleiche. Wir schauen uns die Implementation für den Fall an, wenn B eine klassische Zahl ist. Wir wissen, dass $A >= B$ gilt (für A und B ganze Zahlen), falls $A - B < 0$ oder $A - B == 0$ gilt. Da nicht beide Bedingungen gleichzeitig erfüllt sein können, können wir die Resultate der beiden Checks einfach auf das gleiche Qubit setzen. Ein QInt ist genau dann gleich 0, wenn alle seine Qubits auf 0 sind. Gleichzeitig, falls das Resultat der Subtraktion kleiner als 0 sein soll, gibt es einen Underflow, was so viel bedeuted, dass die Zahl $A-B$ zu $2^n-(B-A)$ wird, und somit das erste Qubit auf 1 gesetzt wird. Das heisst, wir können einfach das erste Qubit überprüfen, ob es auf eins gesetzt ist. Es kann aber passieren, dass $A-B >= 2^{n-1}$ gelten kann, deshalb verlängere ich in meiner Implementation das Register, welches A enthält, um 1.

```
// Checks if an Integer is less than a QInt

operation LessThanCQ(A : Int, B : QInt, res : Qubit) : Unit is Adj+Ctl
{
        using (an = Qubit())
        {
                // Extend the length of B by one to make sure that
                // the first Qubit is 0 if there is no overflow
                let b1 = GrowQIntBy(B, 1, [an]);

                within
                {
                        // Subtract A from B
                        (Adjoint AddCQ)(A, b1);
                }
                apply
                {
                        // Initialize the res to 1
                        X(res);
                        // Result smaller than 0 -> A !< B
                        CNOT(b1::Number[b1::Size - 1], res);
                        // Result equals to zero -> A !< B
                        IsZeroQInt(b1, res);
                }
        }
```

}

Nun können wir messen, ob $x + y >= m$ gilt, und diese Information in einem zusätzlichen Qubit speichern. Falls $x + y >= m$ gilt, subtrahieren wir m von der Zahl und bekommen den Zustand $|x + y - m\rangle$ im zweiten Register. Nun haben wir aber noch Problem, dass die Information, ob $x + y >= m$ gilt, noch in einem Qubit gespeichert wird, welches wir noch zurücksetzen müssen. Hier machen wir die Beobachtungen, dass $x + y >= m$ genau dann gilt, wenn das Resultat grössergleich dem Summanden x ist. Die Richtung $res < x \rightarrow x + y >= m$ ist nicht schwierig. Für die andere Richtung sehen wir, dass $x + y - m >= x$ Nur dann gelten kann, falls $y >= m$ gilt, was aber nach der Annahme $x, y < m$ nicht gelten kann. Somit können wir mit diesem Vergleich die Information in unserem Aushilfsqubit wieder löschen.

3.6 Modulare Multiplikation

Mit Hilfe der Modularen Addition können wir nun die modulare Multiplikation implementieren. Zuerst stellen wir fest, dass wir die Moudulare Multiplikation $|x\rangle \rightarrow |(ax) \pmod m\rangle$ nur dann implementieren können, wenn $ggT(a, m) = 1$ gilt, denn sonst wäre sie nicht reversiel.
Wir schauen uns die modulare Multiplikation auf QInts in zwei Schritten an.

Zuerst implementieren die Quantenoperation auf zwei Register, welche für gegebenes a und m folgende Operation implementiert:

$$U'_{a,m} |x\rangle |y\rangle = |x\rangle |(y + ax) \pmod m\rangle$$

. Dafür sehen wir, dass wenn wir $x = 2^0 x_0 + 2^1 x_1 ...$ in seine Zweierpotenzen aufteilen, dann können wir $ax = x_0(2^0 a \pmod m) + x_1(2^1 a \pmod m) + ...$ schreiben. Dieses $+ x_0(...)$ ist nichts anders als Addition, kontrolliert durch das x_0 Qubit. Dies heisst, wir können diese Operation relativ einfach durchführen:
Für jedes j, führe eine modulare Addition, kontrolliert durch das Qubit x_j, auf das Ausgaberegister mit dem Summanden $2^j a \pmod m$ durch, den wir klassisch berechnen können.
Diese Unteroperation ruft den modularen Addierer $\mathcal{O}(n)$ mal auf und jede dieser Additionen braucht $\mathcal{O}(n^2)$ Gatteroperationen. Damit kommen wir auf $\mathcal{O}(n^3)$ Gatteroperationen.

Mit Hilfe dieser Unteroperation können wir nun die Operation, welche

$$U_{a,m} |x\rangle = |(ax) \pmod m\rangle$$

bewirkt, implementieren:

1. Führe ein temporäres Register im Zustand $|0\rangle$ ein, und bringe es mit Hilfe der oberen Unteroperation den Zustand $|ax \pmod m\rangle$.

2. Berechne klassisch das Inverse von a Modulo m. Dieses Inverse existiert, da a und m teilerfremd sind.

3. Wende die inverse Operation der oben definierten Unteroperation mit a^{-1} mit dem temporären Register als Eingaberegister und dem ersten Register als Ausgaberegister an. Dies ergibt dann den Zustand

$$\left|x - a^{-1}(ax) \pmod m\right\rangle |ax \pmod m\rangle = |0\rangle |ax \pmod m\rangle$$

.

4. Wechsle den Wert der beiden Register mit Hilfe der Swap-Oeration, wir bekommen den Zustand $|ax \pmod m\rangle |0\rangle$. Das temporöre Register im Zustand $|0\rangle$ können wir wieder freigeben und das erste Register ist nun im Zustand $|ax \pmod m\rangle$.

Diese Multiplikation benötigt n extra Qubits für das temporäre Register. Sie ruft die Unteroperation $2 \in \mathcal{O}(1)$ mal auf, und benötigt somit $\mathcal{O}(n^3)$ Gatteroperationen. Wir erinnern uns daran, dass man die Addition mit $O(n)$ Gatteroperationen und dafür n zusätzlichen Qubits implementieren kann, was dazu führt, dass wir nur noch $O(n^2)$ Gatteroperationen benötigen, dafür aber $2n$ extra Qubits benötigen.

Chapter 4

Der Weg zu Shor

4.1 Überblick

In diesem Kapitel werden wir uns die notwendigen Konzepte und Ideen hinter dem quantenbasierten Teil von Shor's Algorithmus anschauen. Dabei starten wir beim simplen Konzept des "Phase Kickback"s, schauen uns dann die darauf basierende Phase Estimation an, bevor wir dann deren Anwendung in Period Finding anschauen. Zum Schluss werden wir uns dann die komplette Implementation vom quantenbasierten Teil von Shors Algorithmus anschauen und überprüfen.

4.2 Phase-Kickback

Beginnen wir den Abschnitt mit einer Frage: Wenn wir eie kontrollierte Operation ausführen, sollte sich das Control-Qubit eigentlich nicht ändern, oder? In diesem Abschnitt werden wir sehen, dass dies überraschender Weise nicht so ist. Dafür schauen wir uns das CNOT-Gate an. Was passirt, wenn wir CNOT auf zwei Qubits im State $|+-\rangle$ anwenden, mit dem ersten Qubit als Control-Qubit? Zuerst haben wir $|+-\rangle = |00\rangle - |01\rangle + |10\rangle - |11\rangle$, nachdem wir das CNOT anwenden bekommen wir den State $|00\rangle - |01\rangle - |10\rangle + |11\rangle = |--\rangle$. Überraschenderweise stellen wir fest, dass sich das Control-Qubit verändert hat, während das Ziel-Qubit gleich blieb. Was ist passiert? Nehmen wir das CNOT-Gate auseinander: Das CNOT-Gate ist eigentlich nichts anderes als eine kontrollierte Version vom X-Gate. Was passiert wenn wir das X-Gate auf den $|-\rangle$-State anwenden? $X|-\rangle = -|0\rangle + |1\rangle = -|-\rangle = (-1) * |-\rangle$. Hier können wir sehen, dass $|-\rangle$ ein Eigenvektor des X-Gates mit Eigenwert -1. Das heisst, der State des Qubits ändert sich nicht, es wird nur die Phase mit dem Eigenwert multipliziert. Da wir nur ein einzelnes Qubit anschauen, hat das keine Auswirkung, da die Phase global ist. Das heisst, alle Amplituden werden mit diesem Wert mit Betrag 1 multipliziert, und wir können dehalb keinen Unterschied feststellen. Wenn wir aber die Operation kontrolliert durchführen, wird diese Phase nur in den States sichtbar, in der die Operation durchgeführt wird, sprich in den States, wo das Control-Qubit im State $|1\rangle$ ist. Dies konnten wir vorher beim CNOT-Gate beobachten. Schauen wir uns nun mal ein generelleres Gate an. Sagen wir, wir nehmen das Gate U mit einem Eigenvektor $|\psi\rangle$ und dem Eigenwert λ. Nehmen wir jetzt ein Qubit q_c im State $\alpha|0\rangle + \beta|1\rangle$, n Qubits $q_0 ... q_{n-1}$ im State $|\psi\rangle$, und führen ein kontrolliertes U auf die Qubits $q_0 ... q_{n-1}$ mit Kontroll-Qubit q_c durch:

$$(\alpha|0\rangle + \beta|1\rangle)|\psi\rangle \xrightarrow{\text{C-U}} \alpha|0\psi\rangle + \beta|1\rangle * U|\psi\rangle = (\alpha|0\rangle + \lambda\beta|1\rangle)|\psi\rangle$$

Das Ziel-Qubit verändert sich nicht, es ist ja ein Eigenvektor, dafür sehen wir, dass der Eigenwert in die Phase des Kontroll-Qubit gekickt wird. Daher kommt der Name "Phase Kickback". Wir werden in der nächsten Sektion sehen, wie dieser Effekt ausgenutz werden kann, um den Eigenwert eines Operators abzuschätzen.

4.3 Phase Estimation

Verschiedene Quanten-Algorithmen basieren darauf, den Eigenwert eines Operators zu einem Eigenvektor abzuschätzen. Dazu benutzen wir Phase-Kickbacks, um den Eigenwert in ein Quantum-Register in der Fourier-Basis zu schreiben, welches wir dann mit der inversen Quanten-Fouriertransformation in die binäre Basis zurückrechnen. Dazu können wir die Anzahl Qubits variieren, um die Präzision der Approximation festlegen. Besser gesagt gibt der Algorithmus zum Eigenwert $\lambda = e^{2i\pi\theta}$ die Zahl $2^n\theta$ zurück, wobei n die Anzahl Qubits des Zählerregisters ist, die für bessere Präzision erhöht werden kann.
Um zu verstehen, wie dieser Algorithmus funktioniert, erinnern wir uns zuerst nochmals, wie eine Zahl in der Fourierbasis aussieht. Dafür benutzen wir nochmals die Bloch-Kugel. Wir erinnern uns, dass für die Zahl x in der Fourierbasis mit n Qubits das k-te Qubit um $\frac{2^k x}{2^n}$ um die Z-Achse gedreht wird. Das heisst, es befindet sich im Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{2^k x}{2^n}}|1\rangle)$. Wir machen jetzt die Beobachtung, dass wir mit Hilfe von Phase-Kickback das gesuchte θ in der Fourierbasis in die Kontrollqubits schreiben können, da der Phase-Kickback nichts anderes macht, als das Kontrollqubit auf die selbe Art und Weise zu rotieren. Schauen wir uns mal an, was passert, wenn wir das kontrollierte U 2^k mal anwenden:

$$\begin{aligned}
\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|\psi\rangle &\xrightarrow{(C-U)^{2^k}} \frac{1}{\sqrt{2}}(|0\psi\rangle + |1\rangle * U^{2^k}|\psi\rangle) \\
&= \frac{1}{\sqrt{2}}(|0\rangle + (e^{2i\pi\theta})^{2^k}|1\rangle)|\psi\rangle \\
&= \frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi 2^k\theta}|1\rangle)|\psi\rangle \\
&= \frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{2^k(2^n\theta)}{2^n}}|1\rangle)|\psi\rangle
\end{aligned}$$

Dies entspricht genau dem k-ten Qubit der Repräsentation von $2^n\theta$ in der Fourierbasis. Das heisst, wenn wir für jedes Qubit im Zählerregister mehrmals ein kontrolliertes U anwenden, können wir einen Zustand kreieren, welcher der Zahl $2^n\theta$ in der Fourierbasis entspricht. Wenden wir dann die inverse Fouriertransformation an, können wir die Zahl $2^n\theta$ im Zählerregister ablesen. Falls $2^n\theta$ keine ganze Zahl ist, dann bekommen wir im Zählerregister eine Superposition, wobei eine Zahl wahrscheinlicher ist, je näher sie am echten Wert ist.

Um die Phase abzuschätzen, müssen wir also den Operator mehrmals hintereinander anwenden, zuerst nur einmal, dann zweimal, im i-ten Mal 2^i mal. Dies führt dazu, dass wir die Operation 2^n mal anwenden müssen. Allerdings ist es oft möglich, dass wir die Operation U^{2^m} für einen beliebigen Parameter m implementieren können. Wenn dies möglich ist, dann brauchen wir nur n Anwendungen jener Operation.

Algorithmus

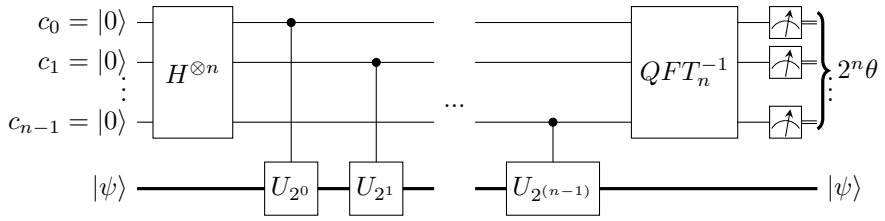
1. Initialisiere zwei Quantenregister, das Zählerregister und das Eigenstate-Register, und setze das Eigenstate-Register auf den gewünschten Eigenstate ψ.

2. Wende $H^{\otimes n}$ auf das Zähler-Register an, um es auf $|+\rangle^{\otimes n}$ zu setzen.

3. Für das i-te Bit im Zählerregister, wende ein kontrolliertes U^{2^i} mit c_i als Kontroll-Qubit an.

4. Wende die inverse Quantenfouriertransformation auf das Zählerregister an, um die Approximation in die binäre Basis umzurechnen.

5. Miss das Zählerregister, um die Abschätzung abzulesen.



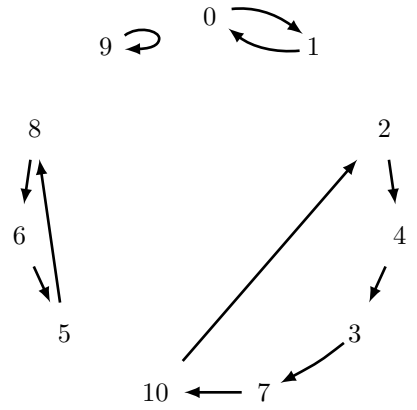
4.4 Period Finding

Gegeben sei eine Funktion $f : S \to S$ mit $S \subset \mathbb{Z}$, welche sich auf einem Quantencomputer implementieren lässt, und ein Wert $x \in S$. Wir versuchen nun, die kleinste Zahl $r \in \mathbb{N}$ zu berechnen, so dass $f^r(x) = x$ gilt. In anderen Worten: Sei $f_x(i) = f^i(x)$. Wir wollen nun die Periode von f_x zu berechnen.

Wir haben gesagt, unsere Funktion soll auf einem Quantencomputer implementierbar sein. Daraus folgt bereits, dass f bijektiv ist: Falls es ein a und ein b mit $f(a) = f(b) = c$ gibt, dann lässt sich $f^{-1}(c)$ nicht berechnen, was im Wiederspruch zur Reversibilität steht. Daraus folgt, dass f injektiv ist. Gleichzeitig müssen deshalb $|S|$ verschiedene Bilder von f existieren, damit jeder Wert ein eigenes Bild hat. Unsere Funktion permutiert die Elemente in S. Schaut man sich diese Permutation als Graph an, so hat jeder Knoten einen Eingangs- und einen Ausgangsgrad von 1. Dies ist jedoch nur möglich, wenn der Graph eine Vereinigung disjunkter Zyklen ist. Dies bedeutet auch, dass man S in verschiedene Teilmengen $S_0, S_1, ...$ aufteilen kann, so dass jede dieser Teilmengen ein einzelner Zyklus des Graphen bildet. Sei nun $x \in S_i$. Da S_i ein Zyklus bildet, gilt $f^{|S_i|}(x) = x$. Gleichzeitig kann kein $r \in \mathbb{N}$ mit $r < |S_i|$ existieren, so dass $f^r(x) = x$ gilt, denn sonst hätte unser Zyklus nur $r < |S_i|$ Elemente. Wir wollen nun also für ein $x \in S_i$ die Grösse $|S_i|$ finden.

Als Beispiel nehmen wir mal $g : A \to A$ mit $A = \mathbb{Z}/11\mathbb{Z}$, $g(x) = -x^3 + 1$. Man kann zeigen, dass $x^3 \pmod p$ bijektiv ist, falls $p \equiv 2 \pmod 3$. Somit ist auch f bijektiv. Wenn wir den Graphen anschauen, dann sehen wir die einzelnen Zyklen: $A_0 = \{0, 1\}$, $A_1 = \{2, 3, 4, 7, 10\}$, $A_2 = \{5, 6, 8\}$ und $A_3 = \{9\}$. Wir sehen nun, dass $f^1(9) = 9$, $f^3(8) = 8$, $f^5(2) = 2$ etc.

TODO Beispiel $f : \mathbb{Z}/11\mathbb{Z} \to \mathbb{Z}/11\mathbb{Z}$, $f(x) = -x^3 + 1$



Die Frage ist nun, wie können wir effizient die Grösse der Teilmenge finden, in der x sich befindet. Dafür müssen wir den Operator f genauer betrachten. Was passiert, wenn wir dem Operator eine Superposition der Zahlen in S_i übergeben? Seien $r = |S_i|$, $x_0, x_1, ..., x_{r-1}$ die Zahlen in S_i, so dass $f(x_j) = x_{(j+1)\%r}$, und U_f die Quantenoperation, die f implementiert. Schauen wir mal, was passiert, wenn wir U_f auf den Zustand $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle$ anwenden? Wir bekommen:

$$U_f(\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle) = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |f(x_j)\rangle = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \left|x_{(j+1)\%r}\right\rangle = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle)$$

Daraus schliessen wir, dass $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle$ ein Eigenstate von U_f mit Eigenwert 1 ist. Dieser Eigenwert ist nicht wirklich interessant. Wir können ihn aber interessanter machen, indem wir den einzelnen Summanden eine Phase mitgeben. Dazu konstruieren wir die Superposition $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)$ für ein $k < r$. Was passiert, wenn wie U_f darauf anwenden?

$$U_f(\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)) = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} \left|x_{(j+1)\%r}\right\rangle) =$$
$$\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{k(j-1)}{r}} |x_j\rangle) = e^{2i\pi \frac{k}{r}} (\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)))$$

Auch hier haben wir wieder einn Eigenvektor, aber mit einem interessanterem Eigenwert, nämlich $e^{2i\pi \frac{k}{r}}$, denn r ist im Eigenwert enthalten. Wir machen auch die Beobachtung, dass unser Eigenstate von vorher $(\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle)$ auch von der Form ist, die wir gerade analysiert haben, einfach mit $k = 0$. Falls wir jetzt irgendwie einen State von der Form $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)$ erzeugen können, könnten wir mit Hilfe der Phase Estimation den Quotienten $\frac{k}{r}$ abschätzen. Die Frage ist, wie können wir solch einen State generieren? Zuerst sagen wir, $|\psi_k\rangle$ sei $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)$. Dann stellen wir fest, dass $|\psi_0\rangle, |\psi_1\rangle, ..., |psi_{r-1}\rangle$ linear unabhängig und damit eine Basis des Untervektorraums über die Zahlen $x_0, x_1, ..., x_{r-1}$ sind. Was passiert nun, wenn wir alle diese Vektoren mit gleichem Gewicht aufsummieren?

$$\frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \frac{1}{\sqrt{r}} (\sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)) = \frac{1}{r} \sum_{j=0}^{r-1} \sum_{k=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle) = |x_0\rangle$$

Eine andere Art, dieses überraschende Resultat zu sehen, ist, dass man die Summe $\sum_{k=0}^{r-1}(e^{-2i\pi\frac{kj}{r}}\left|x_j\right\rangle)$ für $j = 0$ anzuschauen. Da $j = 0$ gilt, gilt $e^{-2i\pi\frac{kj}{r}} = e^0 = 1$ und somit $\sum_{k=0}^{r-1}(e^{-2i\pi\frac{kj}{r}}\left|x_j\right\rangle) = r\left|x_j\right\rangle$. Da $\frac{1}{r}(r\left|x_j\right\rangle)$ bereits einen Betrag von 1 hat, kann kein anderer Zustand mit positivem Betrag existieren, da die Beträge sich sonst zu etwas Grösserem als 1 aufsummieren.

Somit ist x_0 einfach eine Superposition jener Eigenvektoren. Da wir der Periodenabschätzungsfunktion einen Startwert mitgeben, sei jener Startwert WLOG x_0, haben wir eine Superposition dieser Eigenvektoren. Schätzen wir somit den Eigenwert dieser Superposition ab, kollabiert sie in eine der Eigenvektoren, und wir bekommen einen Quotienten $\frac{k}{r}$ zurück, wobei jedes k die gleiche Wahrscheinlichkeit hat. Genauer, bekommen wir die Zahl $2^n\frac{k}{r}$ zurück, wobei n die Präzision ist, die wir dem Phase Estimation-Algorithmus mitgeben. Wir können mit Hilfe von Kettenbrüchen den Quotienten $\frac{k}{r}$ vom Quotienten $\frac{2^n\frac{k}{r}}{2^n}$ abschätzen. Sobald wir den Bruch $\frac{k}{r}$ haben, wissen wir r, was die Zahl ist, die unsere Funktion zurückgeben soll. Nun kann es sein, dass $ggT(k, r) = g \neq 1$ ist, somit der Bruch mit g gekürzt wird und wir dann als Resultat $\frac{r}{g}$ bekommen. Wenn wir die Prozedur aber $2log(N)$ mal wiederholen, bekommen wir mit sehr hoher Wahrscheinlichkeit mindestens einmal die korrekte Periode. Den Beweis dazu kann man in [QC], Seiten 229ff., nachlesen.

4.5 Die Ordnung von Zahlen bestimmen

Der Algorithmus von Shor ist deshalb so schnell, da mit Hilfe des quantenbasierten Teils des Algorithmus die Ordnung einer Zahl schnell bestimmt werden kann. Sei a die Zahl deren Ordnung wir Modulo der Zahl n bestimmen wollen, so dass $ggT(a, n) = 1$. Wir rechnen nun in $\mathbb{Z}/n\mathbb{Z}$. Da $ord_n(a)$ nichts anderes ist als die Periode der Funktion $g(x) = a^x$. Somit können wir die Funktion $f(x) = ax$ implementieren, so dass $f_s(x) = f^x(s) = sa^x$. Mit $s = 1$ bekommen wir dann $f_1(x) = f^x(1) = a^x$. Sei U die Quantenoperation, die f_1 implementiert, dafür können wir einfach die Multiplikation aus der QInteger-Library verwenden. Gleichzeitig können wir auch U^{2^i} effizient implementieren: U^{2^i} ist nichts anderes als die Operation zu f^{2^i}. Da $f^{2^i}(x) = a^{2^i}x$, können wir ganz einfach a^{2^i} klassisch berechnen und dann wieder die gewöhnliche Multiplikation aus der QInteger-Library verwenden. Wir können nun den Algorithmus aus dem vorherigen Kapitel verwenden, um die Periode der Funktion $f_1(x) = a^x$ abzuschätzen. Wir brauchen dafür nur noch eine Funktion, die $f(x) = a^x$ klassisch berechnet, um das Resultat überprüfen zu können, dafür kann man fast direkt FastPowMod aus der QInteger-Library verwenden. Dies führt dazu, dass der Code dieser Funktion nur sehr kurz ist.

4.6 Das Ziel - Shors Algorithmus

Wie erlaubt uns das nun, Zahlen zu faktorisieren? Sei n die zu faktorisierende Zahl. Zuerst überprüfen wir, ob die Zahl durch 2 teilbar oder eine Primpotenz ist, und finden diese Faktoren entsprechend. Nun nehmen wir ein zufälliges $1 < a < n$. Falls $g = ggT(a, n) \neq 1$, dann haben wir bereits einen Teiler gefunden, nämlich g. Sonst sind a und n teilerfremd. Danach suchen wir die Ordnung von a modulo n. Falls diese Ordnung ungerade ist, beginnen wir nochmals von vorne, sonst ist sie gerade. Sei diese Ordnung r. Mit r können wir nun mit gewisser Wahrscheinlichkeit einen Teiler finden. Falls $a^{\frac{r}{2}} \not\equiv -1 \pmod n$ gilt, dann haben wir eine Wurzel von 1 (mod n) gefunden, sonst müssen wir es nochmals probieren. Da r die Ordnung von a ist, muss $a^{\frac{r}{2}} \not\equiv 1 \pmod n$ gelten. Nun gilt: $n|(a^{\frac{r}{2}} + 1)(a^{\frac{r}{2}} - 1)$, aber $n \not| a^{\frac{r}{2}} + 1$ und $n \not| a^{\frac{r}{2}} - 1$. Wir haben nun zwei Zahlen b und c, sodass $n|bc$, aber $n \not| b$ und $n \not| c$. Sei $n = p_0^{\alpha_0}p_1^{\alpha_1}....$ Schreibe nun $b = s_b p_0^{\beta_0}p_1^{\beta_1}...$ und $b = s_c p_0^{\gamma_0}p_1^{\gamma_1}....$ Wir wissen nun, dass $\beta_i + \gamma_i \geq \alpha_i$ gelten muss, da sonst

$n|bc$ nicht erfüllt wäre. Gleichzeitig müssen ein j_b und ein j_c existieren, so dass $\beta_{j_b} > 0$ und $\gamma_{j_c} > 0$ stimmt. Nehme an, dass ohne Beschrünkung der Allgemeinheit $\gamma_i = 0$ für alle i gelte. Dann müsste $\beta_i \geq \alpha_i$ für alle i gelten, und somit $n|b$ teilen, was ein Widerspruch zur Annahme $n \not| b$ wäre. Somit beinhalten beide Faktoren b und c Teiler von n, welche mit dem einfachen ggT-Algorithmus extrahiert werden können. Somit kennen wir nun den ganzen Algorithmus, um einen Teiler von n zu finden.

1. Falls n durch zwei teilbar ist, gib 2 zurück und terminiere.
2. Falls $n = p^a$ eine Primpotenz ist, gib die Primzahl p zurück und terminiere.
3. Bestimme eine zufällige Zahl $1 < a < n - 1$.
4. Finde $g = ggT(a, n)$. Falls $g \neq 1$ ist, gib g zurück und terminiere.
5. Bestimme die Ordnung von a modulo n mit Hilfe des Quantenteils des Algorithmus:
   (a) Schätze die Phase des Operators U_f ab, der $f(x) = ax$ implementiert mit einer Präzision von $m = 2\log_2(n)$ ab. Benutze dazu den in Kapitel 4.3 vorgestellten Algorithmus. Sei das Resultat $2^m\lambda$.
   (b) Schätze den Quotienten $\frac{k}{r}$ von $\frac{2^m\lambda}{2^m}$ ab. Falls r nicht die gesuchte Periode ist, gehe zurück zu (a), sonst gib die Periode r zurück.
6. Falls r ungerade ist, gehe zurück zu 2. Sonst berechne $a^{r/2} \pmod n$. Falls dies kongruent zu $-1 \pmod n$ ist, gehe zurück zu 2.
7. Berechne $b = (a^{\frac{r}{2}} + 1)$. Gib $ggT(b, n)$ zurück und terminiere.