QUANTUM COMPUTERS AND FACTORING

PONTUS LEITZ

Abstract. We review the basic theoretical properties of a quantum computer together with a quantum algorithm for efficiently factoring composite numbers.

1. Introduction

The concept of the quantum computer dates back to 1982, when Richard Feynman envisioned it as a way to make more efficient computer simulations of quantum mechanics than a classical computer could [4]. It was then natural to take the question one step further and ask if the quantum computer could outperform the classical computer in other areas as well. Indeed, people have over the years constructed quantum algorithms for various problems that are fundementally different and more efficient than the known corresponding classical algorithms. Peter Shor's factoring algorithm from 1994 (revised 1996), which will be studied in this paper, is perhaps the most famous of these.

2. Quantum computers

A classical computer can be described as consisting of a collection of bits (subsystems that can take the value 0 or 1) that are prepared in a certain state, manipulated by a series of transformation gates and then measured to obtain a result of the computation [5]. A quantum computer, on the other hand, consists of a collection of *quantum bits* (or *qbits*). A qbit is also a two-state system but in the quantum mechanical sense, meaning that it can be in a state that is a superposition of 0 and 1:

$$|\Psi\rangle = a\,|0\rangle + b\,|1\rangle \tag{2.1}$$

Where a and b are complex numbers satisfying

$$|a|^2 + |b|^2 = 1 \tag{2.2}$$

In the general case we have a system of n qbits and their corresponding state is a nomalized vector in the 2^n-dimensional product space of the individual qbits, i. e. the space spanned by:

$$|0\rangle\otimes\ldots\otimes|0\rangle, |0\rangle\otimes\ldots\otimes|1\rangle, \ldots, |1\rangle\otimes\ldots\otimes|1\rangle = |0\ldots0\rangle, |0\ldots1\rangle, \ldots, |1\ldots1\rangle \tag{2.3}$$

The states $|0\ldots0\rangle, \ldots$ are called the *computational basis states* and correspond to the states of a classical computer. Therefore we can write the general n-qbit state $|\Psi\rangle$ like:

$$|\Psi\rangle = \sum_{0\leq x<2^n} a_x\,|x\rangle \tag{2.4}$$

$$\sum_{0\leq x<2^n} |a_x|^2 = 1 \tag{2.5}$$

Where x is the number given by the computational basis state i binary form. The type of quantum computer considered here is called a *quantum gate array*. [4] The qbits are prepared in a certain state, manipulated by unitary transformations (called *quantum gates*) and then a measurement is performed on all or a subset of the qbits, generating a result. (Here we will only consider measurements in the computational basis, but of course any base is possble.)

The advantage that the quantum computer has over the classical computer comes from the principle of superposition. While a classical computer needs to perform a computation once for each desired input value, the quantum computer can make this computation only *once*: on a superposition of these input values. Unfortunately the fact that we may have the results of all these "simultaneous" computations expressed in the coefficients of the final state does not mean that we can get all these values by simple measurement. This is because that, in general, when we have an unknown state we can never learn what it is by measuring; we only get a result with a certain probability dictated by the coefficients and thereby collapse the state to that particular result so that all the information contained in the coefficients is lost. Furthermore, the no-cloning theorem of quantum mechanics prevents us from making several copies of the state before measurement and getting further clues about the coefficients that way. There are, however, ways around this. The general idea is that after we have performed a computation on the superposition of all the input values we can apply another cleverly designed unitary transformation before measuring, that will change the state into something that will be useful for the problem at hand. In the case of factoring described here this unitary transformation is called the *Quantum Fourier Transform*.

The disadvantages of the quantum computer mainly lie in the difficulties of actually constructing one. We will not discuss any proposals of doing so here, only mention some thoughts on the general problem. Since the bits of a quantum computer are two-state systems they could be constructed out of, for example, electron spins. We must then have at least a few thousand (see the final section) of these and allow them to be entangled by precise manipulations while at the same time keeping them totally isolated from the environment at all times until the final measurement is done. It goes without saying that this poses a formidable task.

2.1. **Unitary Transformations.**

The laws of quantum mechanics permit only unitary transformations of states. We will here look at some unitary transformations that will be important in Shor's algorithm later.

Before continuing we define two binary operations on numbers represented by n bits, $\oplus$ and $\odot$. Let $x_{n-1}, \ldots, x_1, x_0$ be the digits of x. Then the individual digits of $z = x \oplus y$ satisfies $z_i = x_i + y_i \pmod 2$ for $0 \leq i \leq n-1$, and $x \odot y = x_{n-1}y_{n-1} + \ldots x_0 y_0 \pmod 2$.

A unitary tranformation is of course invertible, but not all functions we want to work with are. However, *any* function f with an argument of n bits that takes values of m bits can be realized as a unitary transformation on a state of $n+m$ bits, divided into an input register and an output register. It has the following action on the computational basis states and is extended linearly to the rest of the states:

$$U_f\left(|x\rangle_n |y\rangle_m\right) = |x\rangle_n |y \oplus f(x)\rangle_m \tag{2.6}$$

We see directly that U_f is invertible (it is in fact its own inverse), that it takes computational basis states into computational basis states and therefore is unitary.

Another important transformation is the *Hadamard transformation*, which acts on a single qbit as:

$$H|x\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + (-1)^x |1\rangle\right) = \frac{1}{\sqrt{2}}\sum_{y=0}^{1}(-1)^{xy}|y\rangle \tag{2.7}$$

The extension to n qbits, the *n-fold Hadamard transformation*, is simply the tensor product of n H:s. Its action on a state of n qbits can be compactly written as:

$$H^{\otimes n}|x\rangle_n = \frac{1}{2^{n/2}}\sum_{y=0}^{2^n-1}(-1)^{x\odot y}|y\rangle_n \tag{2.8}$$

We see that it takes the state of all zeros to an equal superposition of all computational basis states:

$$H^{\otimes n}|0\rangle_n = \frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}|x\rangle_n \tag{2.9}$$

3. The factoring problem

Shor's algorithm deals with a problem that is of great practical importance in cryptography: the factoring of a number into its prime factors. The fact that this is a hard problem to solve on a classical computer is the basis of the widespread RSA public key encryption method [5] [4]. To use this method one needs $N = p \cdot q$, the product of two (preferrably large) primes. To encode a message only N is needed, but to decode it one needs the individual factors p and q. Because factoring is such a hard problem the encryption number N can be made public without allowing anyone to know its factors p and q.

To be able to compare the effectiveness of the quantum algorithm with the classical ones we need to have some sort of quantification of the "hardness" of a problem using a certain algorithm. The common way to do this is to express the running time (or number of elementary operations required) as a function of the input number N and to look at the asymptotic behaivour when N grows large. The problems that can be solved with an algorithm that has an asymptotic running time that is at most polynomial in $\log N$ (that is, polynomial in the number of digits of N) can then be considered "easy", while the others are "hard".

The factoring problem belongs to the "hard" category, since the best known classical algorithm, called the *number field sieve*, has a running time asymptotic to $\exp\left(\left(64/9 + \epsilon(N)\right)^{1/3}(\log N)^{1/3}(\log\log N)^{2/3}\right)$ where $\epsilon(N) \to 0$ as $N \to \infty$, which grows faster than any polynomial in $\log N$. Another characteristic of the factoring problem is that while it is hard to find the solution it is easy to verify a solution (or to rule out a wrong one), since you just have to do a division.

3.1. Factoring as Period finding.

The problem of finding a factor of an odd number N can be reduced to finding the order r of an element b in the multiplicative group (mod N). The procedure goes as follows [4]: Take a random b (smaller than N) and compute its order (the smallest r so that $b^r \equiv 1 \pmod N$). Next, note that if r is even

$$0 \equiv b^r - 1 \equiv \left(b^{r/2} - 1\right)\left(b^{r/2} + 1\right) \pmod{N} \tag{3.1}$$

Now, two conditions must be fullfiled to proceed: as noted above, r must be even and $b^{r/2} \not\equiv -1 \pmod{N}$. (We already know that $\left(b^{r/2} - 1\right) \not\equiv 0$, since r is the order of b.) If this is the case, we can compute $\gcd\left(b^{r/2} - 1, N\right)$ with the Euclidean algorithm in polynomial time using a classical computer [4][2]. We also know from the second condition above that $\gcd\left(b^{r/2} - 1, N\right)$ must be a non-trivial divisor of N. Thus, N is factored. The probability for these conditions to be fulfilled can be estimated as follows. First we write down N in terms of its prime factors:

$$N = \prod_{i=1}^{k} p_i^{a_i} \tag{3.2}$$

Next we define r_i to be the order of $b \pmod{p_i^{a_i}}$. This means that r is the least common mutliple of these r_i. Now we look at the power of 2 in the prime factorization of each of these r_i. The two conditions are fullfilled if and only if all these powers are not equal. To see this we first note that if they are all 0, then r is odd so the first condition is violated. If this is not the case we can write:

$$\left(b^{r/2} - 1\right)\left(b^{r/2} + 1\right) \equiv 0 \pmod{p_i^{a_i}} \tag{3.3}$$

for each i. If the powers of 2 are all equal the first factor is $\not\equiv 0$ and therefore $b^{r/2} \equiv -1 \pmod{p_i^{a_i}}$ for all i. This then means that $b^{r/2} \equiv -1 \pmod{N}$. Conversely, if the powers of 2 are not equal there is a j so that $r/2$ is a multiple of r_j. This implies that $\left(b^{r/2} - 1\right) \equiv 0 \pmod{p_j^{a_j}}$, so $\left(b^{r/2} + 1\right) \not\equiv 0 \pmod{p_j^{a_j}}$ and therefore $b^{r/2} \not\equiv -1 \pmod{N}$. Now we want to estimate the probability that the highest power of 2 dividing r_i is not equal for all i. From the Chinese remainder theorem [2][3] we know that choosing a random b in the multiplicative group (mod N) is equivalent to choosing an b_i in the multipilcative group (mod $p_i^{a_i}$) for each i. (b is then the unique number (mod N) congruent to $b_i \pmod{p_i^{a_i}}$ for each i). We also know that the multiplicative group (mod $p_i^{a_i}$) is cyclic [2] for any odd prime power $p_i^{a_i}$, and its order is $p_i^{a_i - 1}(p_i - 1)$ (which is even). Let g be a generator of this group. We can then divide the group into two equally large parts, the even and the odd powers of g. But the highest power of 2 dividing the order of an element in the "even" part is necessarily lower than for an element in the "odd" part. Therefore the probability of a random b_i in this group to have a particular highest power of two dividing its order r_i cannot be greater than $1/2$. So, to conclude this part, the probability that a randomly chosen b in the multiplicative group (mod N) can be used in this method to get a non-trivial factor of N is at least:

$$1 - \frac{1}{2^{k-1}} \tag{3.4}$$

where k is the number of distinct prime factors of N.

The quantum computational part of Shors algorithm is actually an algorithm for finding the period of a function, of which a special case is finding the order of a random element $b \pmod{N}$. When the order is found, the procedure above is used to factor N. Note: there is a (very small) probability that b is not relatively prime to N and therefore not an element in the multiplicative group (mod N). In that case the algorithm will not work, but since b and N have a common factor

the Euclidean algorithm can be used directly to compute $\gcd(b, N)$ and therefore efficiently factor N without having to use the quantum computer at all.

4. Shor's algorithm

4.1. **Quantum Fourier Transform.** The main feature of Shors algorithm is a unitary transformation called the *Quantum Fourier transform*. It is defined by the following action on the computational basis states (and linearly extended to the whole space of states):

$$U_{FT} \left|x\right\rangle_n = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi i x y/2^n} \left|y\right\rangle_n . \tag{4.1}$$

That this is a unitary transformation can be verified with the following calculation:

$$\left\langle x_{U_{FT}} \left| x'_{U_{FT}} \right\rangle = \frac{1}{2^n} \sum_{y=0}^{2^n-1} \sum_{y'=0}^{2^n-1} e^{-2\pi i x y/2^n} e^{2\pi i x' y'/2^n} \left\langle y \left| y' \right\rangle = \frac{1}{2^n} \sum_{y=0}^{2^n-1} \left(e^{2\pi i \left(x'-x\right)/2^n} \right)^y \right. \right. \tag{4.2}$$

If $x = x'$, we see that each term in the sum equals 1 and since there are 2^n terms the whole expression$= 1$. If $x \neq x'$, we simply compute the geometric sum:

$$S = \frac{\left(e^{2\pi i \left(x'-x\right)/2^n} \right)^{2^n} - 1}{\left(e^{2\pi i (x'-x)/2^n} \right) - 1} = 0 \tag{4.3}$$

Thus, we know that U_{FT} takes orthonormal basis vectors into orthonormal basis vectors and can therefore conclude that it is unitary.

4.2. **Constructing the Quantum Fourier Transform.** Since the quantum fourier transform acts on all qbits at once it might be impractical to realize physically as a single quantum gate. It can, however, be constructed as a series of 1- and 2-qbit gates and furthermore, the number of gates (and therefore the running time) is only polynomial in $\log N$. First we define H_i to be the operator that acts as the Hadamard transformation on the i:th qbit and as the identity on the rest. Next, for $j < k$, let $S_{j,k}$ be the operator that acts on qbits j and k by multiplying with a phase factor $\exp\left(i\pi/2^{k-j}\right)$ if both bits are 1 and doing nothing otherwise. The quantum fourier transform for n-bit numbers is then:

$$H_{n-1} S_{n-2,n-1} H_{n-2} S_{n-3,n-1} S_{n-3,n-2} H_{n-3} \ldots H_1 S_{0,n-1} \ldots S_{0,2} S_{0,1} H_0 \tag{4.4}$$

with the exception that the bits in the states are reversed in order. We see that the total number of gates is $n(n-1)/2$. To verify that this actually is the quantum fourier transform, we first define the number y' to be the number y with the bits taken in reverse order. Next we consider the amplitude of going from $\left|x\right\rangle$ to $\left|y'\right\rangle$ via the array of transformations above and compare it to the coefficient of $\left|y\right\rangle$ in the definition of the quantum fourier transform. We note that the Hadamard transformations H_i take a computational basis state into a superposition of all computational basis states (with equal magnitude but different signs) and that the 2^n $1/\sqrt{2}$ factors together become the overall $\frac{1}{2^{n/2}}$ factor, while the $S_{i,j}$ transformations only change the phase of the states. Therefore we see that the absolute value of the amplitude of going from $\left|x\right\rangle$ to $\left|y'\right\rangle$ match the corresponding coefficient of $\left|y\right\rangle$ and it remains to check that their phases also do. The H_i matrices change the overall

sign for each i when x_i and y'_i (that is, the i:th bit of x and y' respectively) are both 1. A change of sign is he same as the addition of π to the phase, so the H_i:s contribution to the phase can be written as:

$$\sum_{j=0}^{n-1} \pi x_j y'_j \tag{4.5}$$

The contribution from the $S_{j,k}$ matrices can be written in a similar way. For each $j < k < n$ they add $\pi/2^{k-j}$ to the phase if the j:th and k:th bits of both x and y' are both equal to 1. Therefore the contribution to the phase can be written as

$$\sum_{0 \leq j < k < n} \frac{\pi}{2^{k-j}} x_j y'_k \tag{4.6}$$

These two expressions can be written under one summation sign by noting that the first expression is equal to:

$$\sum_{0 \leq j = k < n} \frac{\pi}{2^{k-j}} x_j y'_k \tag{4.7}$$

Therefore, they total to

$$\sum_{0 \leq j \leq k < n} \frac{\pi}{2^{k-j}} x_j y'_k \tag{4.8}$$

We want to compare this to the coefficient of y, so we rewrite it using $y'_k = y_{n-1-k}$.

$$\sum_{0 \leq j < k < n} \frac{\pi}{2^{k-j}} x_j y_{n-1-k} \tag{4.9}$$

Now we make the substitution $n - 1 - k = l$ and the inequality $j < k$ becomes $j + l < n$, so the sum is written as

$$\sum_{0 \leq j+l < n} 2\pi \frac{2^j 2^l}{2^n} x_j y_l \tag{4.10}$$

We see that if we would add terms where $j + l \geq n$ we would only add integer multiples of 2π, so we get the same phase if we sum over all j and l from 0 to $n-1$:

$$\sum_{j=0}^{n-1} \sum_{l=0}^{n-1} 2\pi \frac{2^j 2^l}{2^n} x_j y_l = \frac{2\pi}{2^n} \sum_{j=0}^{n-1} 2^j x_j \sum_{l=0}^{n-1} 2^l y_l = \frac{2\pi}{2^n} xy \tag{4.11}$$

Therefore we see that the amplitude of going from $|x\rangle$ to $|y'\rangle$ via the array of H_j and $S_{j,k}$ gates is

$$\frac{1}{2^{n/2}} e^{2\pi i xy/2^n} \tag{4.12}$$

which is the same as the coefficient of $|y\rangle$ in the quantum fourier transform of $|x\rangle$.

4.3. **Finding the Period.** Suppose now that we have an odd number N that we want to factor (in the application of breaking the RSA encryption $N = pq$ where p and q are two large primes). Let n_0 be the smallest number so that $2^{n_0} \geq N$. This means that in order to work with N our quantum computer must have n_0 bits in both the input and the output register. However, This particular algorithm even demands that the input register must have $n = 2n_0$ bits for reasons that will emerge below.

To start Shor's algorithm we first choose a random b in the multiplicative group mod N (that is, a b that is smaller than and relatively pime to N). Now, let $f(x) = b^x \pmod N$ and U_f be the corresponding unitary transformation. The problem of finding the order of b is then the same as finding the period of f.

The quantum computer is initially prepared in the $|0\rangle$ -state and then the input register is put into an equal superposition of all computational basis states with the Hadamard transformation. Then we apply the function U_f:

$$U_f \left(H^{\otimes n} \otimes 1^{\otimes n_0}\right) |0\rangle = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n |f(x)\rangle_{n_0} \tag{4.13}$$

Next comes the main trick of the algorithm: We apply the quantum fourier transform to the input register before measuring the device.

To simplify the expressions a bit, we can imagine that we first make the measurement on the output register, followed by the quantum fourier transform and then the measurement of the input register. The first measurement yields the result f_0 and puts the input register in the state:

$$|\Psi\rangle_n = \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} |x_0 + kr\rangle_n \tag{4.14}$$

where r is the period, x_0 the smallest number which satisfies $f(x_0) = f_0$ and m is the smallest integer so that $mr + x_0 \geq 2^n$. This means that m is the number of periods that fit into the size of the input register.

$$f_0 = f(x_0) = f(x_0 + kr) \tag{4.15}$$

Next, the quantum fourier transform is performed on the state:

$$\begin{aligned} U_{FT} \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} |x_0 + kr\rangle &= \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} \frac{1}{\sqrt{m}} \sum_{k=0}^{m-1} e^{2\pi i (x_0+kr) y/2^n} |y\rangle = \\ &= \sum_{y=0}^{2^n-1} e^{2\pi i x_0 y/2^n} \frac{1}{\sqrt{2^n m}} \left(\sum_{k=0}^{m-1} e^{2\pi i k r y/2^n} \right) |y\rangle \end{aligned} \tag{4.16}$$

Finally, the resulting state is measured and the result y is obtained with the probability $p(y)$:

$$p(y) = \frac{1}{2^n m} \left| \sum_{k=0}^{m-1} e^{2\pi i k r y/2^n} \right|^2 \tag{4.17}$$

Now the quantum computation is done and the process to obtain the period r from this y can be done on a classical computer, except for the fact (see below) that the process may have to be repeated a few times. The key property of the

function $p(y)$ is that it has maxima when y is close to $j \cdot \frac{2^n}{r}$, an integer multiple of $\frac{2^n}{r}$. To see this, consider such y:

$$y = y_j = j \cdot \frac{2^n}{r} + \delta_j \tag{4.18}$$

where

$$|\delta_j| \leq 1/2 \tag{4.19}$$

and insert in the expression for $p(y)$, which simplifies to:

$$p(y_j) = \frac{1}{2^n m} \frac{\sin^2(\pi \delta_j m r/2^n)}{\sin^2(\pi \delta_j r/2^n)} \tag{4.20}$$

This can be further simplified (approximately) by using the facts $r << 2^n$ and $mr/2^n \approx 1$:

$$p(y_j) \approx \frac{1}{2^n m} \left(\frac{\sin(\pi \delta_j)}{\pi \delta_j r/2^n} \right)^2 \approx \frac{1}{r} \left(\frac{\sin(\pi \delta_j)}{\pi \delta_j} \right)^2 \tag{4.21}$$

This expression can be bounded from below using the relations $\sin x \geq \frac{2}{\pi} x; 0 \leq x \leq \frac{\pi}{2}$ and $|\delta_j| \leq 1/2$:

$$p(y_j) \geq \frac{1}{r} \left(\frac{2\delta_j}{\pi \delta_j} \right)^2 = \frac{1}{r} \frac{4}{\pi^2} \tag{4.22}$$

To get probability of getting anyone of these y_j:s, note that there are at least $r-1$ different j:s for which $0 \leq y < 2^n$. Since r is large, an approximate lower bound of the probability is $4/\pi^2 \approx 0.4$. So there is at least a 40% chance that the measurement will yield on of these y:s. Therefore, if the process is repeated a few times there is a good chance that at least one of the results will be useful.

Suppose we have such a y:

$$\left| y - j \cdot \frac{2^n}{r} \right| = \delta_j \leq \frac{1}{2} \tag{4.23}$$

or

$$\left| \frac{y}{2^n} - \frac{j}{r} \right| \leq \frac{1}{2^{n+1}} \tag{4.24}$$

This relation must give a unique value of j/r, because $r < N$ and $2^n > N^2$ imply that if we suppose that there are two different such values this leads to a contradiction:

$$\frac{1}{N^2} < \frac{1}{r_1 \cdot r_2} \leq \left| \frac{j_1}{r_1} - \frac{j_2}{r_2} \right| \leq \left| \frac{y}{2^n} - \frac{j_1}{r_1} \right| + \left| \frac{y}{2^n} - \frac{j_2}{r_2} \right| \leq \frac{1}{2^n} < \frac{1}{N^2} \tag{4.25}$$

The value of j/r can be obtained by using theorem 184 on continued fractions in [1]: If

$$\left| \frac{p}{q} - x \right| < \frac{1}{2q^2}, \tag{4.26}$$

then p/q is a convergent ("partial sum" in the continued fraction expansion of x). Here we can write the relation above as:

$$\left|\frac{j}{r}-\frac{y}{2^n}\right|\leq\frac{1}{2^{n+1}}<\frac{1}{2r^2} \tag{4.27}$$

so j/r will be one of the convergents of $y/2^n$, reduced to lowest terms. The process of obtaining j/r in this way can be done in polynomial time on a classical computer [4]. If j and r have no common factors, this gives us r directly. To obtain the probability for this to happen, consider the function $\phi(r)$ whose value is the number of positive integers less than or equal to r that are relatively prime to r. This function is known as *Eulers totient function*. Therefore, there are $\phi(r)$ values of j that will work. We also know that each j occurs with a probability of at least $4/\pi^2 r$, so the probability of getting r is at least $\phi(r)\,4/\pi^2 r$. Now we can use the theorem 328 from [1], which states that $\phi(r)/r > a/\log\log r$ for some constant a, to note that the procedure needs to be repeated only $O(\log\log r)$ times to have a high probability of succeding.

4.4. **An Approximate Quantum Fourier Transform.** Even though it is a promising fact that the quantum fourier transform can be constructed out of 1- and 2-qbit gates one of its properties seems to be less so. The $S_{j,k}$ gates change the phase of the state, but when $k-j$ is large the phase change $\exp\left(i\pi/2^{k-j}\right)$ becomes very small and that might be difficult to control in practical applications. Actually, the same problem applies to all phase changes since there must always be a limited accuracy with which any physical manipulation can be done. Luckily however, it turns out that small inaccuricies in such phase changes only slightly change the *probability* of getting the desrired result and does not change the result itself. Furthermore, this result can be used to make an approximate quantum fourier transform, where all phase changes below a certain limit are put to 0 (that is, the corresponding $S_{j,k}$ matrices are simply identity matrices)[5][6]. This tranformation also has the additional advantage of using far fewer gates than the original.

To see this, we investigate what happens to the probability $p(y)$ when the phase of each term in the quantum fourier transform is disturbed by the amount $\xi(x,y)$ and that they are bounded by small value ξ:

$$|\xi(x,y)|\leq\xi<<1 \tag{4.28}$$

We call this disturbed probability $p_\xi(y)$ and it has the value

$$p_\xi(y)=\frac{1}{2^n m}\left|\sum_{k=0}^{m-1}e^{2\pi ikry/2^n}e^{i\xi(x_0+kr,y)}\right|^2 \tag{4.29}$$

This expression can be simplified (approximately) since the ξ:s are small in absolute value:

$$p_\xi(y)\approx\frac{1}{2^n m}\left|\sum_{k=0}^{m-1}e^{2\pi ikry/2^n}\left(1+i\xi(x_0+kr,y)\right)\right|^2 \tag{4.30}$$

Now we look at this probability for the values $y=y_j=j\cdot\frac{2^n}{r}+\delta_j$ considered above to see how much the ξ:s change the original value:

$$p_\xi(y_j)\approx\frac{1}{2^n m}\left|\sum_{k=0}^{m-1}e^{2\pi ikr\delta_j/2^n}\left(1+i\xi(x_0+kr,y_j)\right)\right|^2 \tag{4.31}$$

This can be further simplified by multiplying out the squared absolute value and only keeping the terms up to linear in the ξ:s. It then turns out that:

(4.32)
$$p_\xi(y_j) \approx p(y_j) + \frac{2}{2^n m}\Im\left(\left(\sum_{k=0}^{m-1} e^{2\pi i k r \delta_j/2^n}\xi(x_0+kr,y_j)\right)\left(\sum_{k'=0}^{m-1} e^{2\pi i k' r\delta_j/2^n}\right)\right)$$

where $\Im(z)$ is the imaginary part of z. Finally, we use this expression put an upper bound on the difference between the original and the disturbed probabilities:

$$|p(y_j) - p_\xi(y_j)| \leq \frac{2}{2^n m} m\xi m = \frac{2m}{2^n}\xi = \frac{2}{r}\xi \tag{4.33}$$

This inequality is valid for *one* of the y_j:s and, as noted before, there are r such y_j:s. Therefore the probability of getting any of these y_j:s is changed by at most 2ξ (plus some higher order terms in ξ which will be ignored since ξ is small). So if the phase is inaccurate by for example $1/100$, the probability of getting one of the y_j:s is changed by at most $1/50$, or 5% of the orginal probability of about 0.4.

The approximate quantum fourier transformation is then, as noted before, constructed as the original except that the $S_{j,k}$ gates are removed when $|j-k| < l$. This corresponds to an error of at most $\xi = n\pi/2^l$ for each term (since at most n $S_{j,k}$ gates contribute to each term), so the probability will change with at most $2\pi n/2^l$. If we for example put $n = 3000$ (which corresponds to factoring a ≈450 digit number since the input register must be doubled in size) and $l = 20$ the probability is changed by at most 0.018 or 4.5% of the original value.

The second advantage of the approximate quantum fourier transform is also clear: for large n, the number of gates required grows only as $l \cdot n$ instead of $n(n-1)/2$.

4.5. **Modular exponentiation.** Besides the quantum fourier transform we must consider how the actual evaluation of the function $f(x) = b^x \pmod N$ is done and what its running time is. Since all the numbers b, N and x may be hundreds of digits long it might at first glance seem like an enormous number of multiplications must be done. Luckily, the situation is not quite so bad but the evaluation of $f(x) = b^x \pmod N$ is still the most time-comsuming part of Shor's algorithm. The best method of doing the modular exponentiation is that of repeated squaring. First of all we assume that b, N and x are l-bit numbers. We start the process with b and then repeatedly square it $(\text{mod } N)$ l times to obtain $b^{2^i} \pmod N$ for all $i \leq l$. Next we just multiply together the powers $(\text{mod } N)$ for the values of i where the corresponding digit in the binary expansion of x is 1. This obviously gives us $b^x \pmod N$ using $O(l)$ multiplications (and squarings). For the multiplications themselves, the best algorithm asympothically is the Schnhage-Strassen algorithm [4], which uses $O(l \log l \log\log l)$ time to multiply two l-bit numbers. Therefore the evaluation of $f(x) = b^x \pmod N$ takes $O\left(l^2 \log l \log\log l\right)$ time.

This method also requires an additional work register besides the input and output register, where we keep the powers of b. It can be implemented as follows: We start with x in the input register, 1 in the output register and b in the work register. Then, for each bit x_i we multiply the output register with the work register if (and only if) $x_i = 1$. After this we square the work register and continue the process. Thus we need about n_0 extra bits for the evaluation of $f(x) = b^x \pmod N$.

4.6. **Summary.** Taken together, we see that in Shor's algorithm for factoring a number N, the evaluation of $f(x) = b^x \pmod N$ has

$$O\left((\log N)^2 (\log\log N)(\log\log\log N)\right) \tag{4.34}$$

running time while the quantum fourier transform part uses $\log N\left(\log N-1\right)/2$ time ($l\cdot\log N$ for the approximate quantum fourier transform). Therefore the quantum computational part of the algorithm has has $O\left(\left(\log N\right)^2\left(\log\log N\right)\left(\log\log\log N\right)\right)$ running time and this is followed by polynomial time computations on a classical computer, to get the result with an arbirtatily high probability.

We note that while the biggest obstacle to solve the factoring problem for a classical computer is the time required, it is the size that is the main issue to overcome for the quantum computer. As an example we make a rough estimate of how big a quantum computer would have to be to factor the biggest number plausible for today's best computers. The current fastest supercomputer is the IBM Sequoia which can perform 16 petaflops (flops = floating point operations per second) [7]. If we would let this machine operate for a year it would therefore perform $16\cdot10^{15}\cdot3600\cdot24\cdot365\approx5\cdot10^{23}$ operations. If we put this value as the running time for the number field sieve and solve for N we get $N\approx10^{247}$ (this value was obtained with the numerical equation solving function on a Texas Instruments TI-85 calculator). This corresponds to about 820 bits, so there must be a toal of 2460 qbits in the input (of double size) and output registers, plus the work register of about the same number of bits as N. (It may of course be questioned how "plausible" it is to occupy the one-of-a-kind worlds fastest supercomputer for a whole year just to factor a number, but such an estimate might as well be on the high end of things. If we would let the computer work for just a few days it would reduce the number of operations by a factor 100 and that would correspond to factoring a number $N\approx10^{202}$ instead). It is also worth to note that in 1994, when Shor's algorithm was first published, the most powerful supercomputer could perform only about 170 gigaflops [8], which is 5 orders of magnitude lower than today's standards. Using the same equation as before we estimate that this computer could factor a number $N\approx10^{144}$ if allowed to work for a year or $N\approx10^{112}$ in a few days.

4.6.1. *Final Conclusions.* The quantum computer turns out to be an interesting application of quantum mechanics that provides a perhaps unexpected example of how "strange" quantum behaviour can be used to do things that are quite unthinkable in the classical sense. The practical difficulties in constructing one are, however, still very large and the limit when a quantum computer would outperform the best classical computers is also being continously pushed forward by the rapid development of regular computers. It still gives a fascinating glimpse of what can possibly be achieved in the future.

References

[1] G. H. Hardy and E. M. Wright: *An Introdution to the Theory of Numbers*, Oxford University press, 6th edition, 2008.

[2] D. Knuth: *The art of computer programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley, 3rd edition, 1998.

[3] K. H. Fieseler: *Algebra MN3*, Kompendium, Uppsala universitet, 2004.

[4] P. W. Shor: *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, arXiv:quant-ph/9508027v2 25 Jan 1996.

[5] N. D. Mermin: *Lecture Notes on Quantum Computation*, Cornell University, 2006.

[6] D. Coppersmith: *An Approximate Fourier Transform Useful in Quantum Factoring*, arXiv:quant-ph/0201067v1 16 Jan 2002.

[7] Wikipedia article: http://en.wikipedia.org/wiki/IBM_Sequoia

[8] Wikipedia article: http://en.wikipedia.org/wiki/History_of_supercomputing