PHYM004 - Project 2

Frederick W Liardet

2020.03.06

Quantum computing allows the application of quantum mechanical effects in computational tasks. Theoretically, this can allow for calculations that can take impractically too long on a classical computer, to only take minutes to calculate on a quantum system. In this project an integer factorisation algorithm called Shor's algorithm is implemented via linear algebra to simulate a quantum system. A maximum integer of 493 was factorised into its prime factors of 17 and 29 in a simulated 27 qubit system. Shor's algorithm is only effective on a quantum system as this classical implementation is much slower than even a simple brute force method for all target integer sizes.

1 Introduction

A quantum computer uses quantum mechanical processes including superposition and entanglement to compute problems. The bas building block of a quantum computer is the qubit, this is analogous to the classical computer bit.

Due to the quantum mechanical nature of these computers, it is possible that they can theoretically solve some problems in much less time than a classical computer ever could. Recently, researchers at Google proved that a 54 qubit quantum computer could perform certain problems in minutes that a classical supercomputer would take thousands of years to complete. [1]

The majority of this project was based around simulating a version of Shor's Algorithm on a classical computer. On a quantum computer this algorithm will factorise an input integer in polynomial time. This is an especially interesting area that quantum computers excel at, as a lot of the modern cryptography methods rely on large integers being hard to factorise on a classical computer. A theoretical quantum computer with enough qubits would be able to find these factors quickly and therefore be able to bypass most sorts of modern encryption methods.

This project was based on the project outlined by D. Candela in his paper [2] on the classical implementation of Shor's algorithm. This report will focus mainly on the final 'programming project' in this paper as it is the culmination of the previous subprojects and it took the majority of the time to complete.

2 Theory

The base unit of a quantum computer is the qubit, this is analogous to a classical bit (either 0 or 1). However, a qubit can also be in a superposition of states with the probability amplitudes of either state being represented by a complex value [3]. The general quantum state of a qubit with states 0 and 1 and complex amplitudes a and b can be given by:

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

This system can be physically represented by a spin-1/2 particle with the spin up and spin down

states corresponding to the 0 and 1 values respectively.

A single qubit is not that useful for any meaningful calculations, so they are combined in a N-qubit register of these single qubits. This system can be described by a single quantum state $|\Psi\rangle$. A system of N qubits has 2^N basis states (this can be thought of as the total number of permutations of a N bit binary number). The state $|\Psi\rangle$ can be represented by a vector with 2^N elements with each element being the complex amplitude corresponding to that basis state. For example, in a two qubit system:

$$|\Psi\rangle = a\,|00\rangle + b\,|01\rangle + c\,|10\rangle + d\,|11\rangle = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \tag{2}$$

The probability of each state can be calculated from this equation, for example the probability of state $|01\rangle$ is:

$$P(|01\rangle) = |\,\langle 01|\,|\Psi\rangle\,|^2 = b^2 \tag{3}$$

Shor's algorithm take an input integer, C, and attempts to factorise (i.e. $a * b = C$) it into its prime factors. Most of the algorithm itself is fast to compute on a classical computer, however, one stage of it has much lower potential execution time if processed on a quantum computer. The stages of Shor's algorithm are as follows:

- Check that C is odd, otherwise the factors are 2 and $C/2$.
- Check that C is not a power of a smaller integer, otherwise that integer is a factor.
- Find the Greatest Common Divisor (GCD) of C and an integer in the range $0 < a < C$. If the GCD is greater than one, the GCD is a factor.
- Find the smallest p value that satisfies: $a^p = 1\%C$ (where % is the modulo operator). If p is odd or $a^p/2 = 1\%C$ then choose a different a value.
- Finally, the numbers $P_\pm = GCD(a^{p/2} \pm 1, C)$ are the factors of C.

A quantum computer allows a much quicker speedup finding the correct p value, this is known as the 'period finding' section of the algorithm. In our case the rest of the stages are classically calculated and the period finding stage is a simulation of how a quantum computer would calculate this value.

Again, similarly to a classical computer, a quantum computer's logic system relies on gates to alter the state of the qubits. Due to the quantum nature of the qubits, classical gates such as AND, OR and XOR are replaced by quantum gates that operate on their qubits. These quantum gates can be represented by a 2^N rank matrix. Applying one of these quantum gates is done by multiplying the gate matrix with the state vector [2].

The Hadamard gate, $\hat{H}$, acts on a single qubit and is represented by the matrix:

$$\hat{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4}$$

Applying this gate to a single basis state will transform it into an equal superposition of both basis states. Alternatively, if applied to a superposition of basis states, it can return a single basis state.

Another gate used throughout this project is the Phase Shift gate, this can be represented by the matrix:

$$\hat{R}_\theta = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix} \tag{5}$$

This matrix does not affect the magnitude of the basis states, however, it will change its complex phase, for this gate to have an effect on the final results its output must pass through other gates before it is measured. A version of this gate called the controlled phase-shift gate is used in Shor's algorithm.

The gates demonstrated so far will only work on one or two qubits, to generalise a gate for higher numbers of qubits the 'base' gate has to be operated on with an identity matrix in a tensor product. For a system of 3 qubits with the gate $\hat{A}$ applied on the 1st qubit is as follows:

$$\hat{A}^{(1)} = \hat{A} \otimes \hat{I} \otimes \hat{I} \tag{6}$$

In general, a gate operating on the i^{th} qubit in an N qubit system will be a series of tensor products and will have $N-1$ identity matrices with the base matrix in the i^{th} position.

In some algorithms the action taken by a gate must depend on the state of a particular qubit in the system. In this project the phase shift and modular multiplication gates depend on the state of another qubit in the system.

The modular multiplication gate is performed on the whole m register at once, and is controlled by one bit of the L register. If the value of the control bit is one, this gate will multiply the M register by $a^{2^l}\%C$, where a is the a value selected for Shor's algorithm, and l is index of the control qubit in the L register. Interestingly, the bits that this gates works on in Shor's algorithm are not measured for the final results, however, due to the quantum nature of the system the value of the control bit can be changed by this gate.

3 Methods

The most important parts of the algorithm are the state vector and gate matrices and how they interact. First the state vector of the N qubit system was produced by a creating an array of complex values of length 2^N. An array of rank $2^N \times 2^N$ is also needed to store each gate.

Equation 6 can be represented using the indices of the gate matrix in binary form and Kronecker delta functions, using the example from [2], for a 3 qubit system the element in the 7th row and 5th column of a Hadamard gate is given by:

$$\hat{H}^{(2)}_{6,4} = \hat{H}^{(2)}_{[110],[100]} = \delta_{1,1}\hat{H}_{1,0}\delta_{0,0} \tag{7}$$

For a controlled gate there is one less delta as the gate takes two bit indices inputs. For a gate with target bit 2 controlled by bit 1 the equation for element $(6,2)$ is given by:

$$\hat{C}^{(2,1)}_{6,2} = \hat{C}^{(2,1)}_{[110],[010]} = \hat{C}_{[11],[10]}\delta_{00} \tag{8}$$

Initially these equations were implemented by looping over every gate element and checking this equation for each one. However, this was very slow and does not take full advantage of the sparse matrices used in other parts of the project.

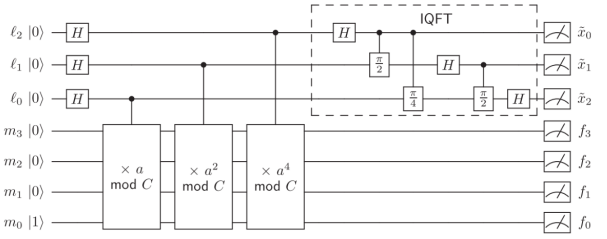

Figure 1: Diagram of Shor's algorithm for seven qubits, with inverse quantum fourier transform section outlined, taken from [2].

The layout of Shor's algorithm for a system of seven qubits is given by figure 1. The total N qubit system is split into two registers (L and M) in this case containing three and four bits respectively. In general, for a target value C, to guarantee the correct output the size of the L and M registers must satisfy:

$$L \geq \log_2(C^2), M \geq \log_2(C) \tag{9}$$

As seen from the diagram, the algorithm can be split into five separate parts:

- Initalise L and M registers
- Apply Hadamard gate to each bit of L register
- Apply modular multiplication gates controlled by L register to each bit of M register
- Perform an Inverse Quantum Fourier Transform (IQFT) on the L register
- Measure the value of the L register

The first stage is to initalise the registers, this is done by creating a 2^N vector with the zeroth element set to one (i.e. m_0 bit is 1). Next, a Hadamard gate is applied to each bit of the L register, these gates are produced in the method described earlier [2] and applied in turn to the L register.

Next the modular multiplication gates are applied to the M register. This gate is unique as it is not based off a smaller 2×2 gate, instead there is a single value set to one at some location in each row. To perform this operation on the M register first the N-bit row number must be written in binary, for example in the seven bit case $l_2 l_1 l_0 m_3 m_2 m_1 m_0$. If the control L bit is zero, then the location of the one value is on the diagonal. This value is also on the diagonal if the $m_3 m_2 m_1 m_0$ value created from the full binary value is greater than C. Otherwise the $m_3 m_2 m_1 m_0$ bits are replaced with the binary expression for $a^{2^l} \% C$, this is combined with the binary representation of the row index to find the value of the column index.

The inverse Fourier quantum transform is comprised of a series of Hadamard and controlled phase-shift gates, the arrangement of these gates is decided by a recursive pattern described by diagram 2. This arrangement of gates is easily implemented by a recursive algorithm with one stage having the right hand side of the diagram and the next recursion containing the QFT_{L-1} gates. This function continuously multiplies the state vector as it goes through the IQFT instead of returning the whole IQFT to save memory space.

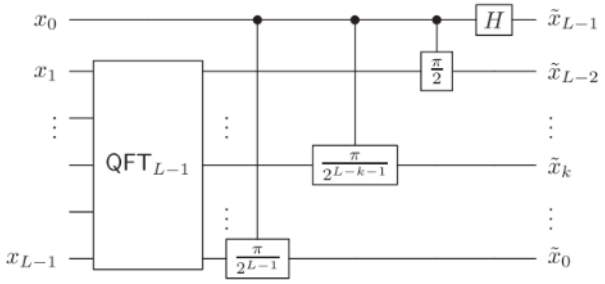

Figure 2: Diagram of the Quantum Fourier Transform on a system of x qubits, where the gates with H are the hadamard gates, and the gate with the π related values are controlled phase-shift gates with their value. Taken from [2].

The final part of the period finding section is to measure the final state of the system. Only the L register is needed for this section and the M register is discarded (these bits are known as ancillary bits). The output period, $\hat{x}$, is created from the L register bits in reverse order (as seen in diagram 1).

To take a measurement from a state vector first a random number between zero and one is created, then this value is compared with an increasing sum of the square of the elements in the vector. When this sum is greater than the random value, the last vector element added is the one that is measured, this relation is shown in equation 3.

If the factors of the target integer found are not prime numbers the program will begin to recursively try to calculate the factors of the non-prime factors. The method for checking if a number is prime is a relatively slow one, but for the range of target integers possible for this program it is easily fast enough.

4 Results

For the obtaining of these results simple python scripts were made to test each configuration of each stage.

As the total number of bits required to factorise a certain integer is related by equation 9, we can see that there will always be roughly twice as many L bits as M bits. This relation put a limit on the largest integer that can be factorised in this method, for my computer this was at 27 qubits which corresponds to a target value of around 512. The approximate amount of system memory used vs the number of bits can be seen in table 3.

The relation between the total number of bits in the system and the time taken for each stage of the algorithm to execute can be seen in figure 3.

The main code that created and multiplied the gates went through many optimisations before the final approach was settled on. The first iteration had the memory for each gate being allocated and deallocated after each gate was used. This became slow at higher numbers of bits, so the memory allocated for each gate was then reused so that there was only one gate allocated per stage of the algorithm.

Obtaining the correct output is dependent on the correct a value being selected for the algorithm. This value is selected randomly from 0 to C, if a invalid value is selected the algorithm must run again with another a value selected. The proportion of valid a

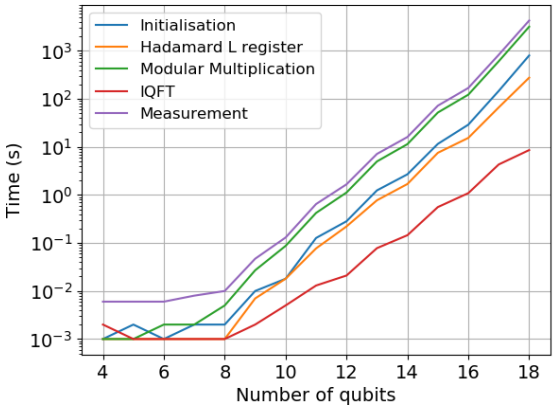

Figure 3: Diagram of execution time vs number of total bits for each section of Shor's algorithm. To keep these results consistent with each other, a target value of 15 and an a value of 7 were used throughout. The data for this plot can be found in table 1

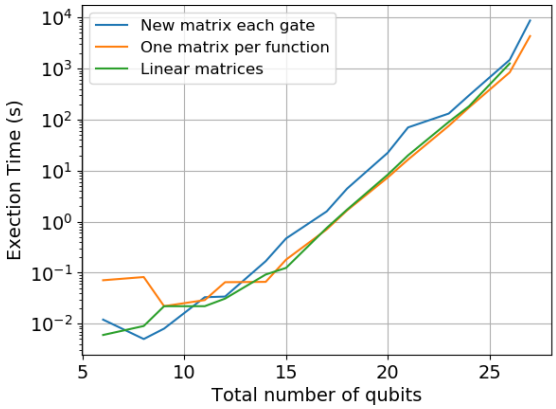

Figure 4: Diagram of execution time vs number of total bits for different optimisations that were added during the process of programming this project. To keep these results consistent with each other, a target value of 15 and an a value of 7 were used throughout. Data can be found in table 2

value compared to some of the lower C values are shown in figure 5.

5 Discussion

The initialisation and measuring of the system are the fastest sections to complete. This is due to the 2^N length array is only looped through once, giving a big O time complexity of $O(2^N)$. The measuring takes proportionally longer than the initialization as the vector has to be looped through a total of three times and other calculations are needed.

Applying the Hadamard gates to the L register and the modular multiplication stages take a similar amount of time due to both sections needing L number of gates to be generated. The Hadamard section takes slightly longer due to the gates having two values per row compared to a modular multiplication gate's one value. This stage has a time complexity of $O(N \times 2^N)$.

The IQFT is the stage that takes the longest time for all system sizes, the number of gates required is usually significantly larger than in the other stages, and is given by:

$$\sum_{i=0}^{L} i = \frac{L^2 + L}{2} \tag{10}$$

For example if the size of the L register is 8 qubits, then the number of gates in the IQFT is 36. Although each Hadamard gate takes longer than a phase-shift gate to generate and multiply, this difference is not noticeable as L increases. The overall time complexity of this stage is $O(N^2 \times 2^N)$, this is the dominant term for this algorithm which is shown in figure 3, in all but the lowest order cases the IQFT takes up over half of the total execution time. All but the lowest cases the time taken to prepare and output the data from Shor's algorithm is negligible, this accounts for the total execution time not being the sum of the indivdual stages of Shor's algorithm.

The slight 'zig-zag' pattern in the execution times in all section (above 8 bits) is likely caused by how the L and M registers increase in number of bits. Due to equation 9, the number of bits in the L register will be roughly twice the number in the M register. The size L register increases twice as often as the M register, the higher than usual increase in execution

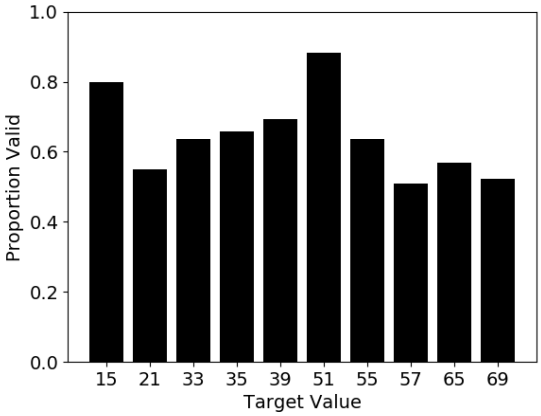

Figure 5: Bar chart of target integer vs proportion of a values that return the correct factors. The exact values can be seen in table 4.

time occurs when both the L and M registers increase at the same time, when only the L register increases there is a smaller increase in execution time.

The largest single performance improvement was to use the bitwise operators in C to allow the value and location of each non-zero element in the gates to be calculated directly, instead of by trial and error.

For example, with the Hadamard matrix, it always has a diagonal element and another element whose location in the row is found by inverting the target bit in the binary number formed from the row number. The value of the each element is found by taking the target bit from the row and column that the value is found and combining them into a coordinate for the original Hadamard matrix, as seen in equation 4.

A similar technique was used to calculate the phase-shift gate, however, this was quicker due to the fact that this gate only has one value per row on the diagonal.

Another speedup obtained was from allocating the memory for the gate in each section at the beginning of the function and deallocating it at the end, compared to allocating and deallocating each time a gate is generated. This created a significant execution time reduction which can be seen in figure 4. The median improvement in execution time for this was 42%.

Finally, the method for storing the matrices was changed from needing two pointer accesses per value (array containing pointers to arrays) to a linear array with the length of the rows stored and then through the use of a macro being able to access values in a two dimensional way. This created a median performance increase of 4% to the execution time.

A single complex double value in C uses 16 bytes of memory, a 14 qubit gate (i.e. rank 2^{14} square matrix) of this data type will take up around 4.5Gb and is the largest of this type of matrix that can be handled by my computer.

However, using the fact that the matrices produced are very sparse (only one or two values per row are non-zero), these matrices can be stored in much less space. With my implementation a 14 bit Hadamard gate takes up approximately 0.6Mb. By changing to a sparse matrix system the number of calulations are reduced from N to roughly $\sqrt{N}$, this improves the speed and memory usage of the program.

The maximum number of bits in the system is limited by the largest Hadamard gate that can be stored. This is due to the Hadamard gate being the only gate that has two values per row, and is such the largest. The memory of my home computer is exceeded when the number of bits in the system is 28, so the maximum number of bits possible is 27. This value of total bits corresponds to a L and M register sizes of 18 and 9 respectively, this size of the registers limits the maximum value of integers that can be factorised. Using equations 9, the highest factorisable integer is 512.

As the number of bits gets close to the 27 bit limit the time taken to complete Shor's algorithm increases more steeply than before, as seen in figure 3 the gradient between the 26th and 27 bit is much steeper than the rest of the plot. This increased time taken is likely due to the large size of the matrices exceeding the physical RAM on the computer, the excess is likely being stored in a paging file on the computer's hard drive, and as such, the time taken to access values from here takes much longer.

For a given target integer, a random value between 1 and the target has to be selected for the algorithm (refered to as a), only some of these values will out-

put correct factors and it is not possible to determine which will prior to running the algorithm. If the number of valid a values decreases with a higher target value it will increase the time to run the algorithm significantly, as each invalid a value found that algorithm has to run again. As seen in table 4, the proportion of valid values does not seem to change drastically with increasing target value. However, this is a small sample of integers as the time taken to check all a values increases dramatically with increasing system size.

If the proportion of valid a values decreases with increasing target number, this would lead to a proportional increase in execution time. When measuring the results for figure 3, a known valid a value was used to make sure that the results would only be affected by the 'true' difference in execution time of the algorithm, instead of having a random factor involved.

Interestingly, the time taken for the measurement section depends on which period will be found by it. Due to how the quantum measurement function loops linearly through the L register, a lower period value will take less time to find. This effect becomes significant at higher numbers of bits as the length of the L register becomes increasingly large.

Due to the nature of this algorithm the output factors will either be both correct, only one factor correct or both incorrect. In the case that only one correct factor is found, this can be divided with the target integer to obtain the correct second factor. The incorrect output seems to have two main forms, either the trivial factors are found (i.e. 1 and the target integer), or both factors found are equal to one.

Throughout all the testing of this algorithm there was never a case where the factors found were close (but slightly wrong) of the true factors. This is likely due to the way this algorithm deals in mostly integer values, and as such no error can be introduced.

Many integers were factorised up to a maximum value of 493, these values are are shown in table 5. To increase this maximum value some changes would have to be made to the program, for example in general the double data type could be replaced with the float type which although it has less precision, it would only take up half of the memory and may not affect the end output significantly. Another possible change would be to store the generated gates that will be reused later on in the program. This storage and reading from storage may be faster than generating the same gates again. In the same vein perhaps generating the value needed from the gate in situ would remove the need to have the gates stored in memory.

6 Conclusion

The future of quantum computing is an interesting one, if the issues of quantum decoherence and stability in large qubit systems can be fixed, there are many problems that cannot be solved with classical computers that will be solvable very easily on a quantum system.

On a classical computer Shor's algorithm is much slower than even a simple brute force method. The speedup in the period finding section for a quantum computer is significant and can lead to this part being computable in polynomial time, $O(n^a)$. However, on a classical computer this part of the algorithm is only computable in exponential time.

The limiting factor of this classical implementation of Shor's algorithm is the amount of memory on the system that it is run on, with 8GB of RAM, this leads to a system of 27 qubits. This corresponds to a theoretical maximum value of 512 that can be factorised.

Due to this limiting factor of the RAM and processor speed, this algorithm will only be truly useful when applied on a quantum computer, on a classical computer even a brute force method will be much faster than this implementation of Shor's algorithm. However, on a quantum computer with enough qubits this algorithm can factorise integers into is prime factors much faster than even the most advanced classical method could.

7 Data

Number of Bits	Init Time	Had Time	Mod Mult Time	IQFT Time	Measure Time	Total
6	0.001	0.001	0.001	0.001	0.002	0.006
8	0.001	0.002	0.001	0.001	0.001	0.006
9	0.001	0.001	0.001	0.002	0.001	0.006
11	0.002	0.002	0.001	0.002	0.001	0.008
12	0.001	0.002	0.001	0.005	0.001	0.01
14	0.001	0.01	0.007	0.027	0.002	0.047
15	0.002	0.018	0.018	0.087	0.005	0.13
17	0.007	0.127	0.077	0.422	0.013	0.646
18	0.015	0.283	0.221	1.123	0.021	1.663
20	0.063	1.244	0.774	4.988	0.078	7.147
21	0.125	2.688	1.688	11.421	0.145	16.067
23	0.509	11.556	7.539	52.009	0.559	72.172
24	1.002	29.041	15.405	122.287	1.09	168.825
26	4.044	146.492	66.494	607.621	4.313	828.964
27	8.281	802.838	276.776	3186.745	8.598	4283.238

Table 1: Execution time (s) vs number of bits for all sections of Shor's algorithm.

Total Bits	New Matrix each gate	One matrix per function	Linear matrices
6	0.012	0.006	0.071
8	0.005	0.009	0.082
9	0.008	0.022	0.022
11	0.033	0.022	0.029
12	0.034	0.031	0.065
14	0.168	0.092	0.066
15	0.468	0.124	0.18
17	1.588	0.753	0.704
18	4.433	1.717	1.66
20	22.363	8.323	7.324
21	70.24	19.909	16.458
23	131.264	89.884	75.282
24	298.656	182.803	173.008
26	1483.108	1260.626	843.132
27	8746.491	MEM ERROR	4332.762

Table 2: Execution time (s) vs total number of bits for various optimisations on the algorithm.

Qubits	Hadamard Matrix (MB)	Vector (MB)	Total (MB)
1	0.10	0.03	0.13
2	0.19	0.06	0.26
3	0.38	0.13	0.51
4	0.77	0.26	1.02
5	1.54	0.51	2.05
6	3.07	1.02	4.10
7	6.14	2.05	8.19
8	12.29	4.10	16.38
9	24.58	8.19	32.77
10	49.15	16.38	65.54
11	98.30	32.77	131.07
12	196.61	65.54	262.14
13	393.22	131.07	524.29
14	786.43	262.14	1,048.58
15	1,572.86	524.29	2,097.15
16	3,145.73	1,048.58	4,194.30
17	6,291.46	2,097.15	8,388.61
18	12,582.91	4,194.30	16,777.22
19	25,165.82	8,388.61	33,554.43
20	50,331.65	16,777.22	67,108.86
21	100,663.30	33,554.43	134,217.73
22	201,326.59	67,108.86	268,435.46
23	402,653.18	134,217.73	536,870.91
24	805,306.37	268,435.46	1,073,741.82
25	1,610,612.74	536,870.91	2,147,483.65
26	3,221,225.47	1,073,741.82	4,294,967.30
27	6,442,450.94	2,147,483.65	8,589,934.59
28	12,884,901.89	4,294,967.30	17,179,869.18

Table 3: Memory used to store a Hadamard gate and a state vector for increasing number of qubits.

Target Value	Proportion Valid
15	0.800
21	0.550
33	0.636
35	0.657
39	0.692
51	0.882
55	0.636
57	0.509
65	0.569
69	0.522

Table 4: Proportion of valid a values vs target integer to be factorised

C	L	M	a	Factors	Execution time (s)
15	8	4	7	3,5	0.041
21	9	5	15	3,7	0.14
57	12	6	2	3,19	7.262
65	13	7	48	5,13	25.687
87	13	7	59	3,29	10.789
143	15	8	111	11,13	562.341
493	18	9	4	17,29	4295.780

Table 5: Table of example target integers, size of L and M registers, valid a value selected, the factors produced and execution time

References

[1] Arute, F., Arya, K., Babbush, R. et al. Quantum supremacy using a programmable superconducting processor. Nature 574, 505510 (2019). https://doi.org/10.1038/s41586-019-1666-5

[2] Candela, D. Undergraduate computational physics projects on quantum computing. American Journal of Physics 83, 688 (2015). https://doi.org/10.1119/1.4922296

[3] Hagar A., Cuffaro M., Quantum Computing. Stanford Encyclopedia of Philosophy (2015) https://plato.stanford.edu/entries/qt-quantcomp/