Zach Harris, 105221897
Trevor Rollins, 105227465

DJ, BV, Simon, Grover in Q#

I. Evaluation

Deutsch-Jozsa

To test the DJ in Q#, the implementation found in the Quantum Katas was used. Rather than using the test harness however, we instead made a custom Driver.cs file with our own tests and timing. Four implementations of Uf were considered for testing, all implemented in the Kata. First is a balanced Uf which considers whether the number of "1" input bits is odd or even. Second is a balanced Uf with only considers the value for a reference bit (Kth bit). Third is a constant Uf, which always flips the helper (f(x)=1), and last is a constant Uf which is identity for the helper (f(x)=0).

Each of these cases were run 10,000 iterations. For each iteration, an assert statement verified that the balanced Uf's returned False, and the constant Uf's returned True. There were no failed asserts, and we are confident that the circuit works as expected.

The run time of each iteration was measured in clock cycles. To account for the inherent inconsistency of this measure, the average of all iterations was considered and plotted (Figure 1). As expected the balanced Uf's took longer to execute than the constant Uf's, however the margin was very small. This is a stark difference from both PyQuil and Cirq, in which balanced Uf's increased in runtime exponentially with n and constant Uf's retained constant execution time with n. Here all forms of Uf appear to increase linearly in execution time with regards to n.

The balanced Uf's considered here are special cases that require less time than a randomly generated balanced f. Considering number of odd bits, and a Kth bit should scale linearly rather than the n-bit controlled NOT approach we implemented in the other languages to account for non-predictable oracles. However it is unclear why the constant oracles are linear as well.

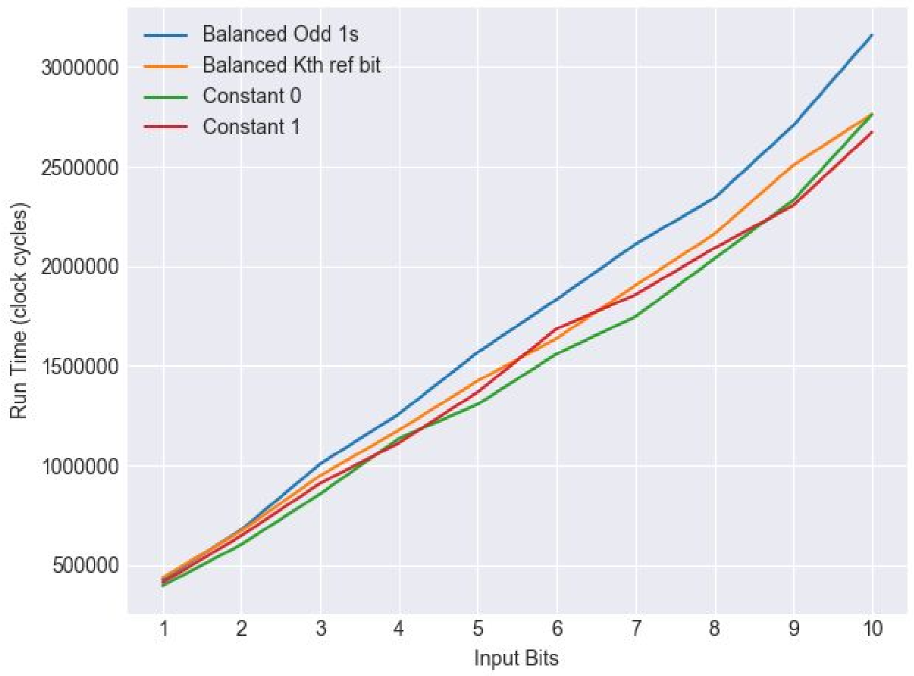

Figure 1: Execution time for Deutsch-Jozsa circuit

Bernstein-Vazirani

The value of $\overline{a}$ was verified for each value of $\overline{a}$ up to 2^5-1=31. This was verified by inputting every case of $\overline{a}$ and $\overline{b}$ into the system and ensuring the output was again $\overline{a}$. Since this problem scales with n, we can safely assume that it will work for any integer. Next, the quantum circuit was simulated for increasing values of n up to 10. The figure below shows the time, in computer ticks, spent by the simulator to perform each simulation.

The number of clock cycles per run are very few. We profiled a Ryzen CPU and found that the Q# Quantum Simulator takes up 100% of the CPU in order to achieve this performance. Any other programs running could require a portion of the CPU resources, and could therefore affect the timing of individual runs. To combat this effect, 100,000 samples were taken for each instance of n and averaged together to produce the final result. The blue line represents the most complex case for each n ($\overline{a} = 2^n - 1$ and $\overline{b} = even$), and the orange line represents the bare minimum ($\overline{a} = 0$ and $\overline{b} = odd$, no gates are actually added to U_f).

Much like the case in DJ, the execution time increases as the number of constituent gates in U_f increases. We were expecting a linear growth rate, since for BV we only need to add at most $n+1$ gates for a given n number of input bits. However, the increase in the number of input bits produces a polynomial-shaped curve. We are unsure of the reason why after 6 input bits we see a dip in run time. It may be that the simulator begins to spin up more threads once a certain number of input bits is reached. Furthermore, the orange line follows the trends of the blue line, suggesting that the dominant factor to run time in the Q# simulator is no longer how many gates are involved in the circuit, but rather how many inputs bits there are.

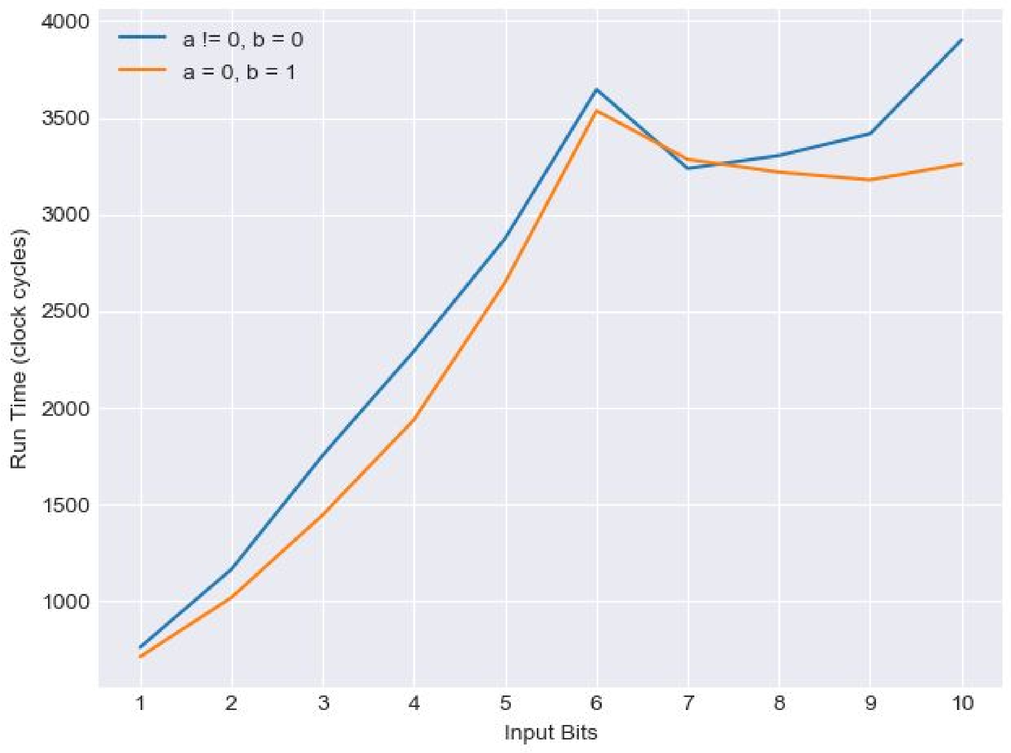

Figure 2: Execution time for Bernstein-Vazirani circuit

Simon

Each cycle of the quantum portion of Simon's algorithm produces a y such that y * s = 0. Many y's (in this case 4n) are collected to solve for s. Simon's algorithm is probabilistic in that there is at least a ¼ chance that all y's returned will be linearly independent. This means that there is a high chance over overlap between y's, and many y's will need to be collected to solve for s. We used the implementation of Simon's algorithm presented in the Quantum Katas.

Similar to DJ, we used a custom Driver.cs file for testing and timing of this algorithm. For timing, we compared two implementations of Uf: a right bitwise shift on the input, and a multi-dimensional operator on the input. The bit shift required slightly less time than the multi-dimensional operator which is unsurprising as it requires fewer gates. However both implementations grew exponentially in execution time with regards to n. This is exactly what we experienced in both PyQuil and Cirq, so is unsurprising here.

To test the correctness of the algorithm and the gaussian solver, we used the right bitwise shift Uf. The s from this oracle will always be $|0^{n-1}1\rangle$. So for n=2 it will be $|01\rangle$ and for n=3 it will be $|001\rangle$ which is exactly what was observed. We then used the `BooleanMatrix` object and its associated `GetKernel()` function to solve for s in the classical portion.

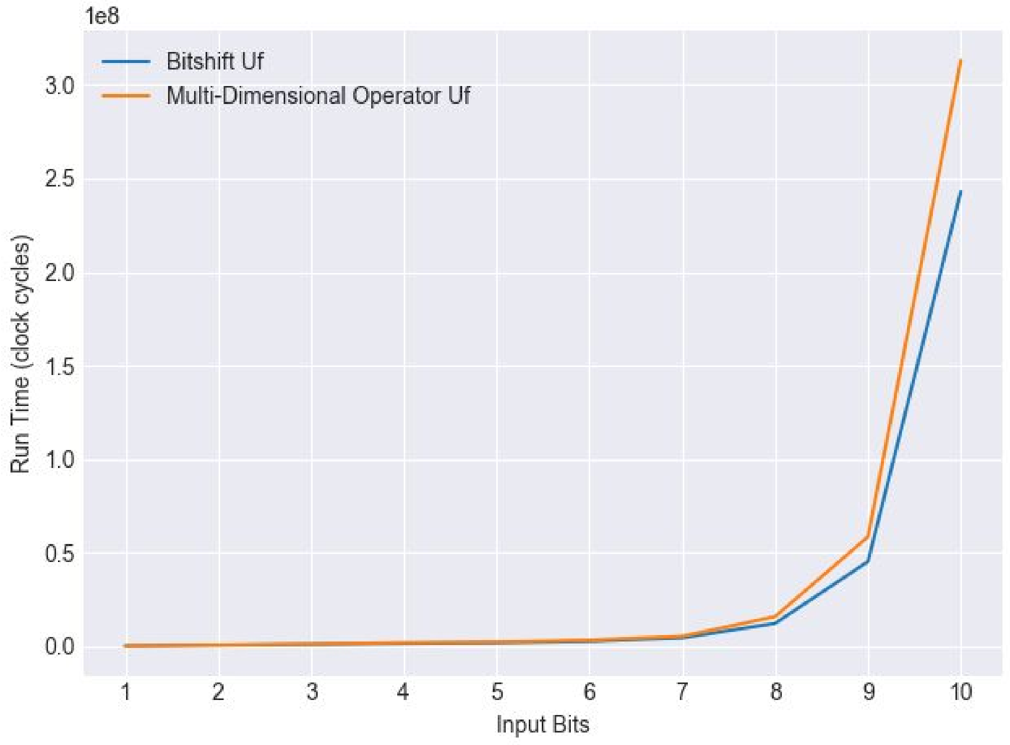

Figure 3: Execution time for Simon circuit

Grover

Because Grover is not guaranteed to give the correct answer with probability 1, it is necessary to test the outcome multiple times. In order to produce a more accurate measurement for the system, we simply run the simulation multiple times using the `ITER` parameter from `run_grover` and count the most common-occurring situation as the correct answer. As we mentioned previously, the Q# Quantum Simulator takes up as much space on the CPU as it can. In order to get accurate timing measurements, we performed 100,000 simulations for each n.

We found that this number of simulations was also high enough to ensure that the correct answer from the simulation was the most-occuring. For example, when n=4 and k=0, we find that the simulation chooses k=0 to be the output 47.10% of the time. This is extremely close to the probability predicted by the Quirk circuit simulator, which is 47.3% (proof). We performed the same tests with n up to 10 to verify our results were accurate.

The figure below shows the relationship between execution time of the Grover simulation and n. It is unclear why the simulation run time spikes at n=7. Declaring this point as an outlier, we see that the general trend of growth is linear in n. There is a jump between n=6 and n=8 because after n=7 the number of G sets that are used in the circuit doubles from 1 to 2. There is different behavior between the simplest circuit (orange line) and more complex one (blue line), however the orange line generally never surpasses the blue line. This suggests that the Q# simulator will take longer to perform on average if there are more gates present in the circuit.

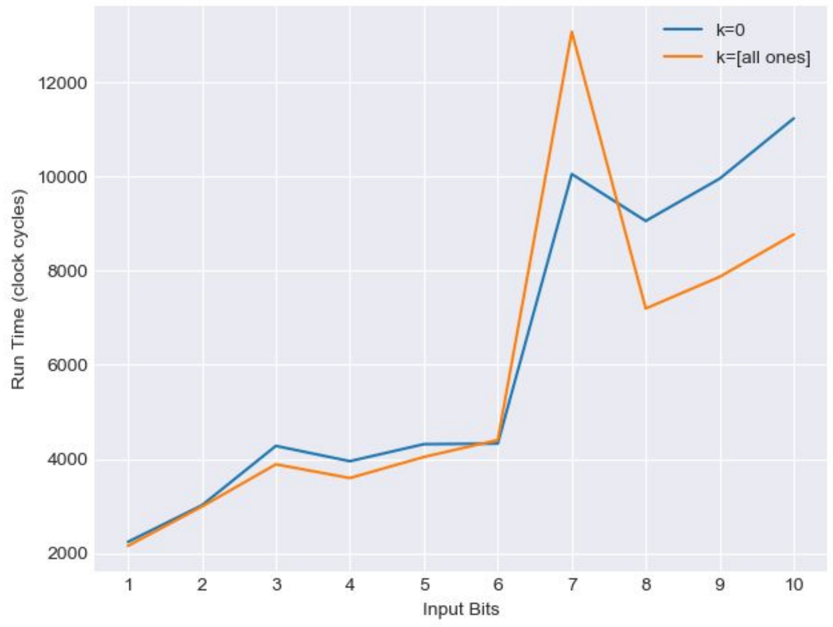

Figure 4: Execution time for Grover circuit

II. Instructions

The README.md file describes the packages necessary and steps involved to run the tests. Please read the descriptions below that summarize this file for each experiment. For all four coding problems, extract the .zip file pertaining to the problem of interest (DJ.zip, BV.zip, Simon.zip, or Grover.zip). This will create a folder that contains all code files necessary to build and run the problem.

In order to build the solution, `cd` into the target directory. Then, run `dotnet build` in a terminal to compile the program. Finally, run `dotnet run` to perform the steps found in the `*_Driver.cs` file.

Deutsch-Jozsa

`DJ/DJ_Driver.cs` is the file used to run the Deutsch-Jozsa algorithm. Currently there are 4 functions called within `Main()`: `DJ_bal_odd(), DJ_bal_Kth_ref(),` `DJ_const_zero()`, and `DJ_const_one()`. Each of these functions take the quantum simulator, a maximum n, and number of iterations as parameters. There are two `const int` declared, `ITER` and `MAX_N`, that are passed into all functions. They will run all values of n from `1...MAX_N`, `ITER` number of times each and then display the result and execution time for each n.

The balanced functions will have a result of false, while the constant functions will have a result of true. You can edit the values of `ITER/MAX_N` to change the parameters of all functions at once, or edit each function's parameters individually.

Bernstein-Vazirani

`BV/BV_Driver.cs` is the script used to run the Bernstein-Vazirani algorithm on a given function. To use this script, you will need to start a quantum simulator and define the integers n, ITER, a, and b.

The quantum simulator object is created by using the following: `var qsim = new QuantumSimulator()`. The variable n is used to represent the number of input bits. ITER is used to represent the number of times the simulation should be run for the given n. The variables a and b are used to represent the function, $f(x) = \overline{a} * x + \overline{b}$. Simply pass these values into `run_bv(qsim, n, ITER, a, b)` and it will create the circuit, insert the correct U_f, compile the program, and simulate it using the Q# simulator.

The averaged execution time is returned by `run_bv()`, and the results are printed at the end of the function. You may add your calls to `run_bv()` at the end of the `Main()` function. We suggest to perform a dry run of the simulator on a meaningless circuit in order to give the simulator time to warm up.

Simon

`Simon/Simon_Driver.cs` is the file used to run the Simon algorithm. Currently there are 4 functions called within `Main()`: `time_simon_bitshift()`, `time_simon_multi()`, `run_simon_bitshift()`, and `Test_Qubit_Reset()`. The first two functions are strictly for timing simon's with different Uf implementation. Their parameters are: the quantum simulator object, a maximum n, and number of iterations. There are two `const int` declared, `ITER` and `MAX_N`, that are passed into these functions. They will run all values of n from `1...MAX_N`, `ITER` number of times each and then display the execution time for each n.

After this there are two calls to `run_simon_bitshift()`. One with n=2, and one with n=3 as parameters. This will display the resulting s for a 2 or 3 bit bitshift. Additional calls can be added to run other n values, or you can edit the existing calls.

Lastly there is quantum operation call that tests the qubit reset functionality. This can be removed if you like, it will always fail with Q#'s current settings.

Grover

`Grover/Grover_Driver.cs` is the script used to run the Grover algorithm on a given function. To use this script, you will need to define the integers n, k, and grover_repeats.

The quantum simulator object is created by using the following: `var qsim = new QuantumSimulator()`. The variable n is used to represent the number of input bits. ITER is used to represent the number of times the simulation should be run for the given n. The variable k is used to represent the value of x where f(x)=1. It is assumed that in all other cases of f, f(x)=0. The variable grover_repeats is used to tell circuit builder how many times the G subcircuit will be placed into the circuit. Simply pass these values into `run_grover(qsim, n, ITER, k, grover_repeats)` and it will create the circuit, compile the program, and run it on the Q#simulator.

The averaged execution time is returned by `run_grover()`, and the results are printed at the end of the function. You may add your calls to `run_grover()` at the end of the `Main()` function. We suggest to perform a dry run of the simulator on a meaningless circuit in order to give the simulator time to warm up.

III. Q# Reflections

Three aspects of programming in Q# that turned out to be easy to learn

1. Dotnet turned out to be easier than expected to learn. I liked how it could initialize a directory with the template code, and handled all of the build/run under the covers of "dotnet run".
2. Qubit allocation did not have to be as explicit as PyQuil or even Cirq. The "using" directive handled most details, and a topology did not have to be declared.
3. The quantum simulator was equally easy to use. It also was enabled with the "using" directive, and could be used to simulate any quantum operation with the scope of that directive using the `.Run()` function.

Three aspects of programming in Q# that turned out to be difficult to learn

1. To program successfully, you must learn both Q# (quantum) and C# (classical/Driver). These two languages have similarities, but enough differences that they must be learned separately.
2. Q# is a strongly typed language, and has different types than C#. This means that passing values between quantum and classical sides requires explicit conversions, and is far less seamless than PyQuil/Cirq.
3. The representation of, and difference between, operations and functions in Q# was unclear in the beginning and took some getting used to.

Three aspects of programming in Q# that the language supports strongly

1. Q# does a better job of abstracting the quantum portion of the program than Cirq/PyQuil. Not having to explicitly declare qubits, topology, etc is nice.
2. It has a quantum library to simply certain mundane aspects. An example of this is the ApplyToEach() function, which allows you to easily apply a gate to all specified qubits.
3. Q#'s build system seems better set up to handle larger projects, and integrate the quantum with the classical. Q# and C# files that share a namespace, are automatically linked together.

Three aspects of programming in Q# that the language supports poorly

1. Though great for simplicity, not being able to specify exact topology/qubits gives the programmer less control.
2. There doesn't seem to be a way to define custom unitary gates.
3. Operations and timing using basic C# packages such as Math and Stopwatch are difficult to perform in Q#.

Which feature would you like Q# to support to make the quantum programming easier?

It would be outstanding for Q# to support automatic type conversion between Q# and C#, so we as developers don't have to spend much time thinking about all of the specific and unique variables types presented in Q#.

List three positives of the documentation of Q#

1. The Quantum Katas are a great set of examples to work off of.
2. Their documentation on github is an excellent way to manage issues that users stumble across.
3. If you can find them in the documentation, they have every programming element defined like they should.

List three negatives of the documentation of Q#

1. Does a poor job of going into the type conversion between C# and Q#. Nowhere in the documentation did I find that `Int[]` in Q# corresponds to `IQarray<long>` in C#. I had to look at compiler errors.
2. It briefly went over array types, but completely skipped how to work with 2D arrays, which were necessary for Uf in Simon's.
3. Q# is a very featureful language, and documentation goes deep into most of these features, but I don't think it does a good job of organizing it. Quantum theory is mixed in with Q# documentation, and it is often hard to find what I'm looking for.

In some cases, Q# has its own names for key concepts in quantum programming. Give a dictionary that maps key concepts in quantum programming to the names used in Q#

Q# jargon	**QP jargon**
Operation	Grouping of quantum gates applied to qubits
Zero	|0>
One	|1>
Adjoint U	U^{+}
M	Measure

The "using" and "borrowing" statements give access to some additional qubits. At the end of such a statement, each of those qubits must appear unchanged. Thus, if a qubit was in state Ψ at the beginning of such a statement, it can change during the statement, but must be in state ψ at the end. Show results from experiments that shed light on whether the simulator enforces this.

We added a check for this within simon.qs/Simon_Driver.cs. In the check qubits were “used” in an initial Zero state and then flipped to the One state. They were then released, and attempted to be re-aquired. However when the qubits are released without returning them to their original state, the simulator will throw an exception.

So the simulator does enforce the unchanged release of the qubits, however I’m not sure why it doesn’t just run `ResetAll()` rather than throwing an exception.