CS239 - UCLA

SPRING 2019

HOMEWORK 12: REPORT

Using Q# for Quantum Algorithms: Deutsch-Jozsa, Bernstein Vazirani, Simon's and Grover's

GROUP MEMBERS

Rishab Doshi - 305220397
Srishti Majumdar - 105225254
Vidhu Malik - 305225272

Part 1. Evaluation

1.1 Implementation Details

All the code used for this assignment was already available through official Microsoft Documentation for each of the algorithms. Our major contribution was to understand the programs and further add instrumentation statements to measure times for each of the different algorithm runs. The links to original source code files for each of the algorithms have been listed below:

- Deutsch Jozsa and Bernstein Vazirani - https://github.com/microsoft/Quantum/tree/master/Samples/src/SimpleAlgorithms

- Grover's Algorithm - https://github.com/microsoft/Quantum/tree/master/Samples/src/DatabaseSearch

- Simon's Algorithm - https://github.com/microsoft/QuantumKatas/tree/master/SimonsAlgorithm

Deutsch Jozsa Problem

The code for Deutsch Jozsa Problem was run by using the DeutschJozsaTestCase object. We have added different constant and balanced functions as shown in below code snippets. In addition to this we have also created balanced and constant functions for different number of Qubits, starting from n=1 to n=15.
The functions

- static void testConstantFunction
- static void testBalancedFunction

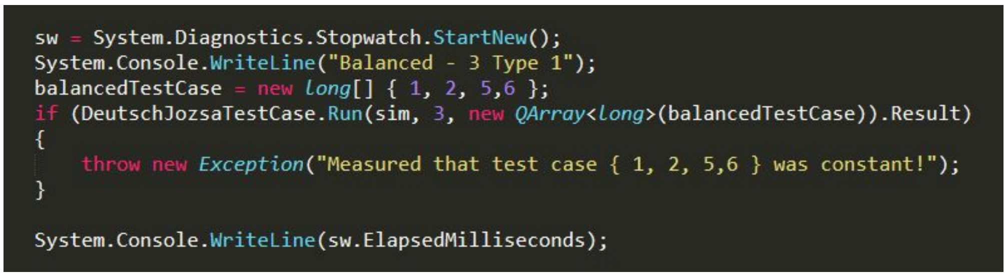
are used to perform the above mentioned operation. The results of running these have been discussed ahead.

Sample Functions - Deutsch Jozsa

To test the Deutsch Jozsa Problem, we have added profiling statements that measure the time taken for each run. To test the runtime variation for different Uf matrices, we test the code with four different balanced functions of nQubits = 3. The functions are as follows:

	000	001	010	011	100	101	110	111
f_1	0	1	1	0	0	1	1	0
f_2	0	0	1	1	1	1	0	0
f_3	0	1	0	1	0	1	1	0
f_4	0	1	1	1	1	0	0	0

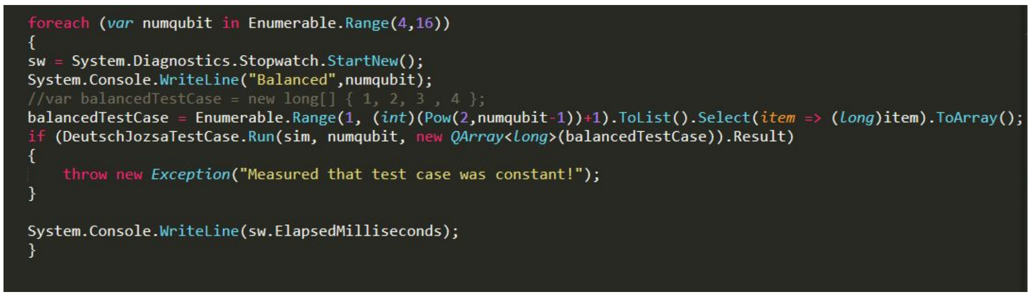
The code snippet describing the profiling operations for one of these functions is shown below:



```
sw = System.Diagnostics.Stopwatch.StartNew();
System.Console.WriteLine("Balanced - 3 Type 1");
balancedTestCase = new Long[] { 1, 2, 5,6 };
if (DeutschJozsaTestCase.Run(sim, 3, new QArray<Long>(balancedTestCase)).Result)
{
    throw new Exception("Measured that test case { 1, 2, 5,6 } was constant!");
}

System.Console.WriteLine(sw.ElapsedMilliseconds);
```

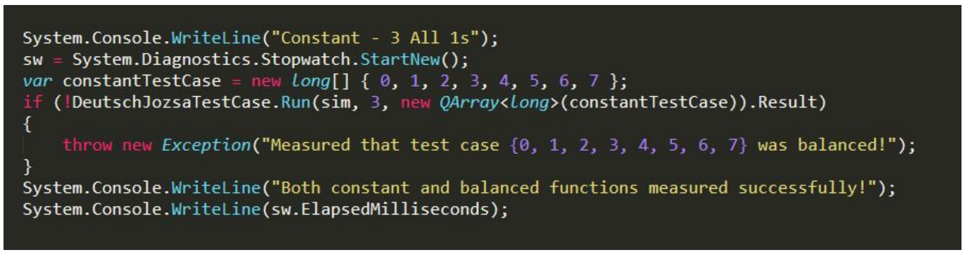
The input array balancedTestCase = {1,2,5,6} signifies that the input bits corresponding to 1,2,5,6 will have outputs 1 and the rest of the input bits will be related to outputs 0. Hence, the above code snippet corresponds to f_1.

Along with the above experiment we also evaluated the solution for different number of Qubits. The trend is observed for upto 16 qubits. For each different Qubit sie, the input function is designed such that the first n/2 Qubits are matched with an output of one and the second n/2 qubits correspond to an output 0. The code snippet is described below:



```
foreach (var numqubit in Enumerable.Range(4,16))
{
sw = System.Diagnostics.Stopwatch.StartNew();
System.Console.WriteLine("Balanced",numqubit);
//var balancedTestCase = new long[] { 1, 2, 3 , 4 };
balancedTestCase = Enumerable.Range(1, (int)(Pow(2,numqubit-1))+1).ToList().Select(item => (Long)item).ToArray();
if (DeutschJozsaTestCase.Run(sim, numqubit, new QArray<Long>(balancedTestCase)).Result)
{
    throw new Exception("Measured that test case was constant!");
}

System.Console.WriteLine(sw.ElapsedMilliseconds);
}
```

All the above experiments are pertaining to balanced functions. We have also tested the code for Constant functions with nQubits = 3 such that in one of the cases all qubits are 1 and in the other case, all qubits are 0. One of these snippets can be seen below:



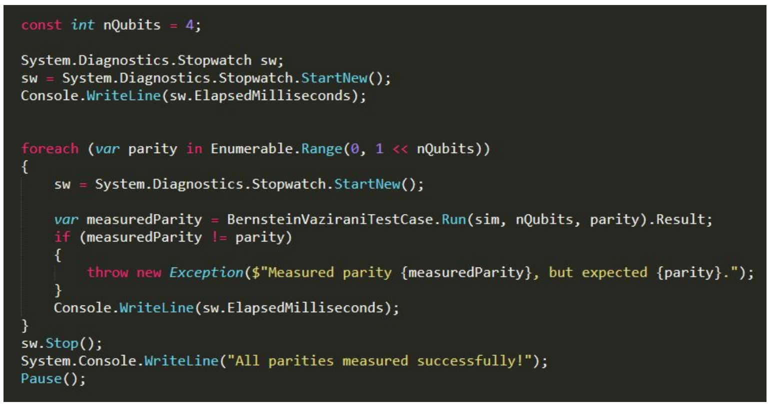
```
System.Console.WriteLine("Constant - 3 All 1s");
sw = System.Diagnostics.Stopwatch.StartNew();
var constantTestCase = new long[] { 0, 1, 2, 3, 4, 5, 6, 7 };
if (!DeutschJozsaTestCase.Run(sim, 3, new QArray<long>(constantTestCase)).Result)
{
    throw new Exception("Measured that test case {0, 1, 2, 3, 4, 5, 6, 7} was balanced!");
}
System.Console.WriteLine("Both constant and balanced functions measured successfully!");
System.Console.WriteLine(sw.ElapsedMilliseconds);
```

The results of all the experiments can be found in later sections. Additionally, logs of each of the runs are enclosed in the respective folder

Bernstein Vazirani Problem

The code for Bernstein Vazirani Problem was run by using the BernsteinVaziraniTestCase object. This code uses the concept of a function parity for every run. The parity represents the bitstring whose value for the function is 1 and rest of the bitstrings are all output 0 for function f(x).

To measure the performance of Bernstein Vazirani Problem, we have added profiling statements that measure the time taken for each run. The following image shows the update to the driver code that measures the time taken for every run.



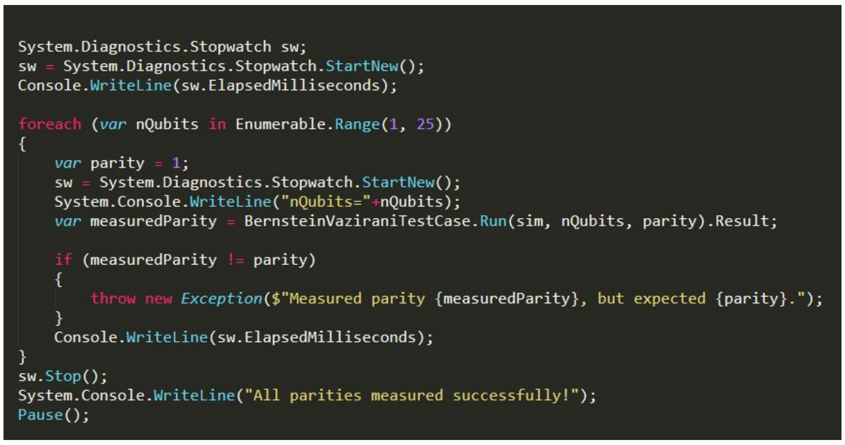
```
const int nQubits = 4;

System.Diagnostics.Stopwatch sw;
sw = System.Diagnostics.Stopwatch.StartNew();
Console.WriteLine(sw.ElapsedMilliseconds);


foreach (var parity in Enumerable.Range(0, 1 << nQubits))
{
    sw = System.Diagnostics.Stopwatch.StartNew();

    var measuredParity = BernsteinVaziraniTestCase.Run(sim, nQubits, parity).Result;
    if (measuredParity != parity)
    {
        throw new Exception($"Measured parity {measuredParity}, but expected {parity}.");
    }
    Console.WriteLine(sw.ElapsedMilliseconds);
}
sw.Stop();
System.Console.WriteLine("All parities measured successfully!");
Pause();
```

The piece of code shown above runs the algorithm for all functions with nQubits = 4. Since each function is different, the corresponding U_f matrices will also be different. Results of this experiment are discussed in later sections.

We have also recorded the results for different nQubits value applied to the same function. The function under evaluation is designed such that in a function $f(\vec{x})$ on bitstrings $\vec{x} = (x_0, \ldots, x_n)$ of the form

$$f(\vec{x}) \coloneqq \Sigma_i \, x_i \, r_i$$

where $\vec{r} = (r_0, \ldots, r_n)$ is an unknown bitstring that determines the parity of f, we set r_0 = 1 and the rest of $\vec{r}$ with 0's. This experiment shows us the change in runtime as the number of qubits are varied from 1 to 25. A code snippet showing the changes made for this experiment can be seen below:



```
System.Diagnostics.Stopwatch sw;
sw = System.Diagnostics.Stopwatch.StartNew();
Console.WriteLine(sw.ElapsedMilliseconds);

foreach (var nQubits in Enumerable.Range(1, 25))
{
    var parity = 1;
    sw = System.Diagnostics.Stopwatch.StartNew();
    System.Console.WriteLine("nQubits="+nQubits);
    var measuredParity = BernsteinVaziraniTestCase.Run(sim, nQubits, parity).Result;

    if (measuredParity != parity)
    {
        throw new Exception($"Measured parity {measuredParity}, but expected {parity}.");
    }
    Console.WriteLine(sw.ElapsedMilliseconds);
}
sw.Stop();
System.Console.WriteLine("All parities measured successfully!");
Pause();
```

Results of this experiment are discussed in later sections. In addition to this the logs of each of the runs have been reported in the specific folder(instructions in README).

Grover's Problem

The Grover's problem is used to search a database of bitstrings to identify those bitstrings that satisfy certain properties. The typical use-case is to identify which of the input bitstrings give an output of 1 when a function f(x) is applied to it. The implementation in Microsoft/Quantum for Grover's algorithm can be divided into the below categories.

1. Classical Random Database Search with Manual Oracle Definitions
   There is exactly one bitstring that satisfies the property of f(x) = 1, we don't use any quantum operations but rather search for the bitstring classically. The probability of finding the required bit-string in every iteration is 1/(DatabaseSize). The bitstring satisfying the property is set as $|11\ldots1_n>$ for a bit-string of size n.

2. Quantum Database Search with Manual Oracle Definitions
   There is exactly one bitstring that satisfies the property of f(x) = 1, we use the quantum circuit defined by Grover's algorithm and run iterations proportional to square-root(2^n) times, where n in no. of qubits. This quantum circuit is built by manual definition of an oracle circuit. The bitstring satisfying the property is set as $|11\ldots1_n>$ for a bit-string of size n.

3. Multiple Element Quantum Database Search with the Canon
   There are multiple bitstrings that satisfy the property of f(x) = 1. Also, the quantum circuit is built with canon.

The code was directly adapted from the ReferenceImplementation given. This uses an assertion statement to verify if the matrix created after GaussianElimination is the same as the input matrix.

In addition to this, we also added test-cases for canon quantum search with random initializations of the marked element(the element that satisfies the property). We observe that with the increase in no. of qubits, the quantum algorithm clearly has much higher success probability of finding the required bitstring. We discuss the results in the next section and also the logs from the run of the program have been submitted for inspection.

Simon’s Problem

This algorithm is used for identifying the bit-mask, also referred to as the secret bitstring that is used in a two-to-one function as below.

$$f(x) = f(y) \text{ implies that } x + y = \{0, s\}$$

The aim of the quantum algorithm is to identify the s value. This process of finding s gets an exponential speedup with the quantum algorithm. The quantum circuit can be used to produce n-1 linear equations in the n-bits that satisfy the property that dot-product with s value will be 0. These equations are then solved using Gaussian Elimination as a classical post-processing step. The program for this was adapted from the QuantumKatas , this checks using assert statements that the matrix input to create the oracle is same as the one output after gaussian elimination.

We removed unnecessary test functions and added extra functions to specifically profile the runs for different qubits. The code required for GaussianElimination has already been implemented. Also, different oracle functions used to build the oracle gate have been specified in the QuantumKatas. We discuss the results in the next section and also the logs from the run of the program have been submitted for inspection.

1.2 Execution time for different U_f

Deutsch Jozsa Problem

As per the details of the experiment that was listed earlier, we check if the Deutsch Jozsa Problem required different amount of time for different U_f matrices. The results can be seen below:

Table 1.2.1: Execution Time for each function over each Trial (Time in ms)

Trials	Function In Consideration			
	f1	f2	f3	f4
Trial 1	3	3	2	2

Trial 2	3	4	4	3
Trial 3	6	3	3	3
AVERAGE	4ms	3.33ms	3ms	2.66ms

Note: The definitions of the four functions have been provided in earlier sections. It can be seen that the time does not vary much with change in U_f. In fact the discrepancy between trials is more than the discrepancy between the resultsE of different U_f in a single trial.

Also given below are the results of the constant functions:

Bernstein Vazirani Problem

As per the details of the experiment that was listed earlier, we check if the Bernstein Vazirani Problem required a different amount of time for different U_f matrices. The results can be seen below:

Table 1.2.2: Execution Time for each function over each Trial (TIme in ms)

Function Parity	Trial 1 (Time in ms)	Trial 2 (Time in ms)	Trial 3 (Time in ms)	**Average**
0000	5	7	11	7.66 ms
0001	4	5	8	5.66 ms
0010	3	8	8	6.33 ms
0011	5	13	9	9 ms
0100	4	6	9	6.33 ms
0101	6	4	5	5 ms
0110	3	5	4	4 ms
0111	6	6	8	6.66 ms
1000	3	5	5	4.33 ms
1001	6	3	6	5 ms
1010	3	9	8	6.66 ms
1011	4	7	8	6.33 ms
100	4	3	8	5 ms
1101	3	3	8	4.66 ms
1110	3	6	4	4.33 ms

1111	5	4	8	5.66 ms

Note:The function definitions have been explained earlier. As stated before, each parity represents a different function and hence a different U_f matrix. nQubits = 4.It can be seen that the time does not vary much with change in U_f. In fact the discrepancy between trials is more than the discrepancy between the results of different U_f in a single trial.

Grover's Problem

As per the details of the experiment that was listed earlier, we discuss the details of running the experiment for Grover's problem.

Table 1.2.3: Details about Probability and Runs for three types of Quantum Runs done for Grover's algorithm (nIterations=3)

Run Type	Number of Qubits	Marked Element	Database Size	Classical Probability	Quantum Probability	SuccessCount/ TotalRuns	Average ElapsedTicks
Classical	3	111	8	0.125	-n/a-	144/1000	19130.808
Quantum	6	111111	64	0.015625	0.59	64/100	5727.45
Quantum Canon	8	0,39,59, 101,234	256	0.0196	0.69	6/10	390014

In Table 1.2.4, we discuss the results of running a quantum search on increasing number of qubits to identify a bit string that outputs f(x)=1 (initialized randomly). The probability function for classical and quantum circuits have been shown in figures 1.3.1 and 1.3.2.

Table 1.2.4: Grovers search for Random bitstring initialized to 1 (nIterations=3)

Number of Qubits	Marked Element	Database Size	Classical Probability	Quantum Probability	SuccessCount/ TotalRuns	Average ElapsedTicks
2	1	4	0.25	0.25	2/10	11973
3	7	8	0.125	0.33	2/10	18618
4	4	16	0.0625	0.96	10/10	29781
5	13	32	0.031	0.896	9/10	8361.2
6	30	64	0.015625	0.591	5/10	8533

As can be seen from the above table, the average elapsed ticks differs for increasing qubits. However, the time required to run the program remains roughly the same and the increase or decrease is negligible. Also, there is a discrepancy with variation in the values occurring between trials.

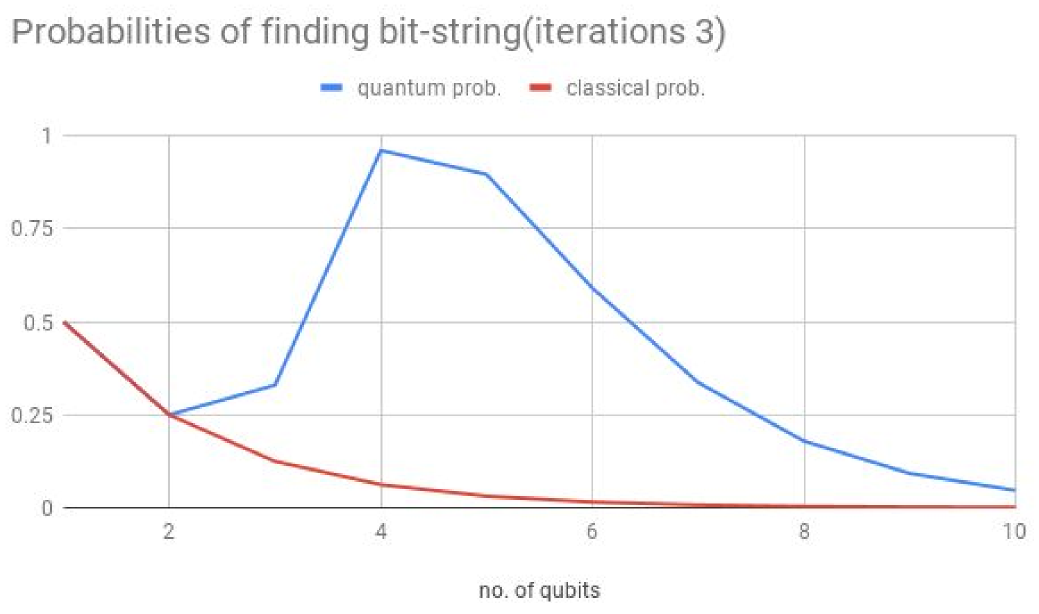


Figure 1.2.1: Classical and Quantum Probabilities of finding an n bit-string by Grovers(iterations 3)

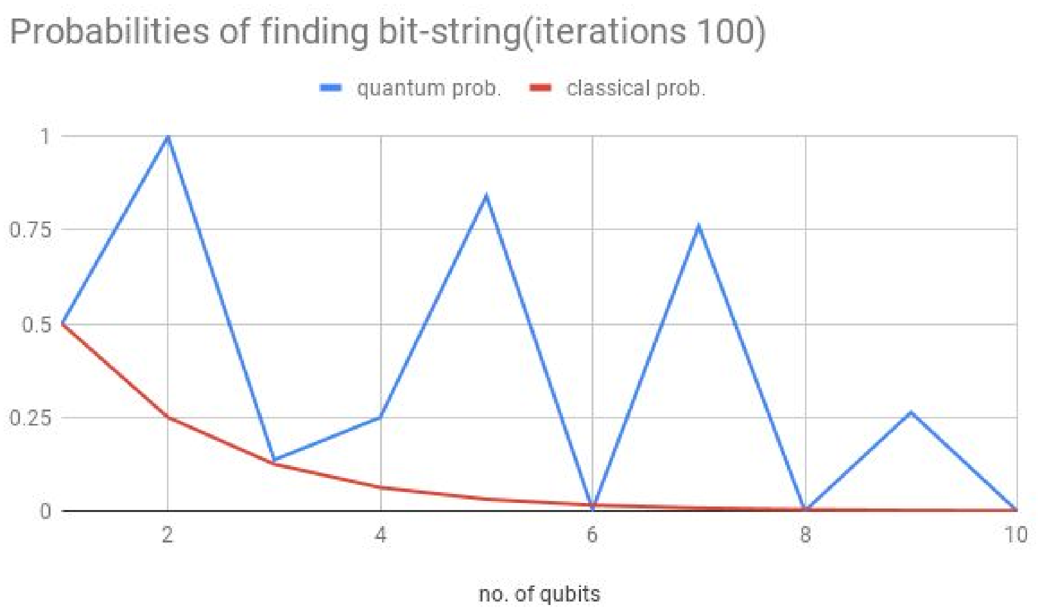


Figure 1.2.2: Classical and Quantum Probabilities of finding an n bit-string by Grovers(iterations 100)

Simon’s Problem

As defined in the earlier sections Simon’s problem attempts to find the secret value satisfying the below property of f(x) two-to-one function

$$f(x) = f(y) \text{ implies that } x + y = \{0, s\}$$

The aim of the quantum algorithm is to identify the s value.

In the implementation Simon’s algorithm in QuantumKatas, the Oracle matrices corresponding to s values are stored in an external file Instances.json. Our aim is to use a quantum circuit to gather vectors satisfying the property that v.s = 0. Using n-1 such vectors for an n qubit function, we can solve this set of equations using Gaussian Elimination. This has been implemented and the matrices inside Instances.json and the ones obtained by solving quantum circuit outputs are compared with an assertion statement to ensure equality.

Below we list a couple of matrices in Instances.json that are asserted to be equal to the matrix obtained by Gaussian elimination of equations produced by quantum circuit.

```
{
   "instance": 25,
   "transformation": [  //matrix created as an oracle gate
     [ 0, 1, 0, 1, 0, 1 ],
     [ 1, 1, 1, 1, 0, 1 ],
     [ 1, 0, 1, 1, 0, 1 ],
     [ 1, 1, 1, 1, 0, 0 ],
     [ 0, 0, 1, 0, 0, 1 ]
   ],
   "kernel": [ 0, 0, 0, 0, 1, 0 ] //
 }
Total Elapsed Ticks : 256048133
Total Elapsed Milliseconds : 256
```

1.3 Execution time as n grows

Deutsch Jozsa Problem

As described regarding the experiment previously, we have recorded runtime values in ms for different n. The results are summarised in the graph below:

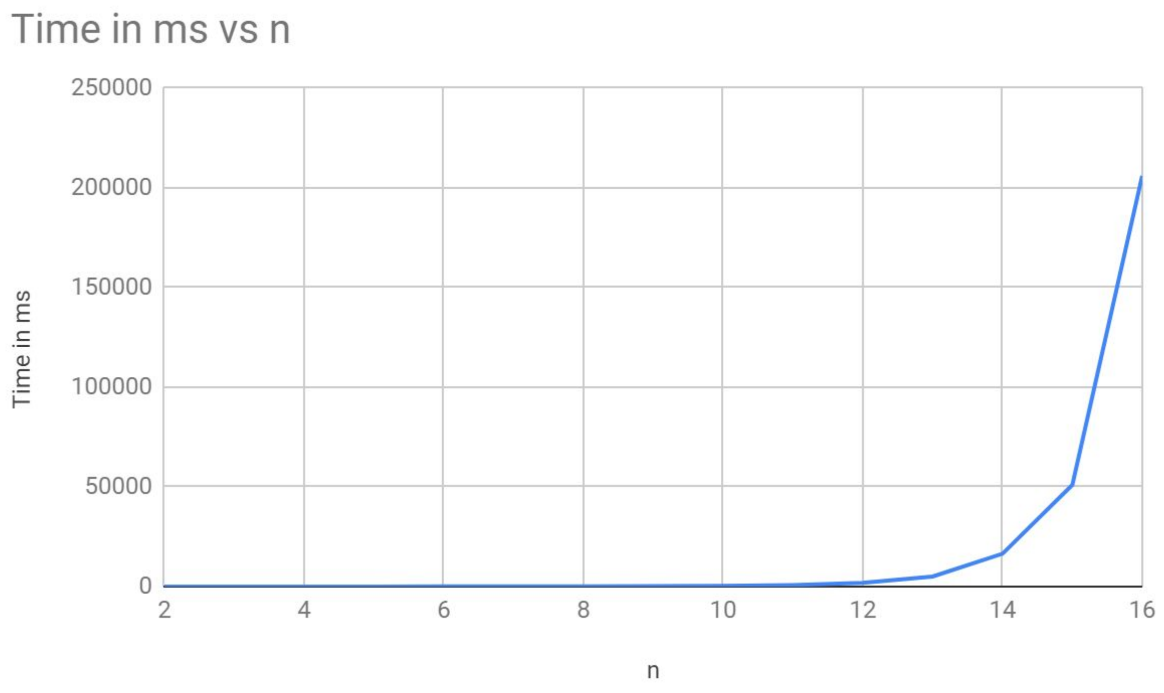


Figure 1.3.1: Execution time as a function of no. of qubits

Table 1.3.1: Data corresponding to Figure 1.3.1

n	Time in ms
2	2
3	4
4	11
5	12
6	41
7	53
8	78
9	166
10	353
11	767
12	1860
13	4990

14	16435
15	50979
16	206032

The growth is clearly exponential. An increase in even a single qubit can cause significant increase in runtime. The steepness of the growth significantly increases after n=12. Hence, runtime is a function of number of qubits. After 16 qubits, the program became non-responsive.

Bernstein Vazirani Problem

As described regarding the experiment previously, we have recorded runtime values in ms for different n. The results are summarised in the graph below:

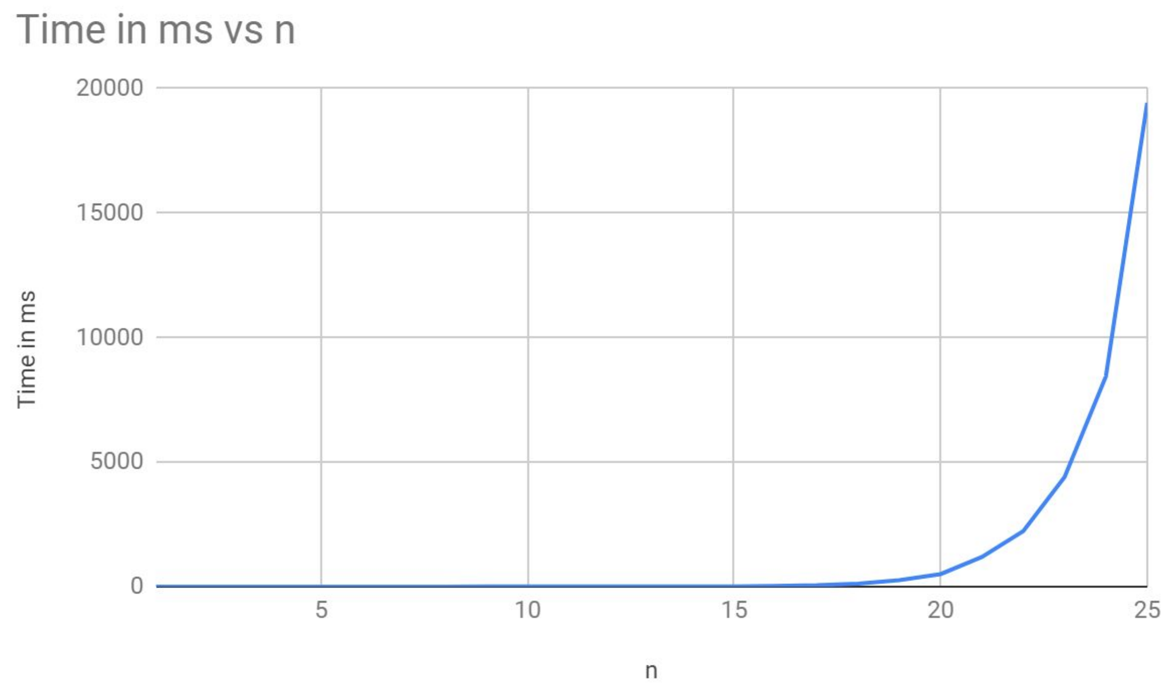


Figure 1.3.2: Execution time as a function of no. of qubits

Table 1.3.2: Data corresponding to Figure 1.3.2

n	Time in ms
1	0
2	1
3	0
4	0

5	0
6	1
7	0
8	1
9	2
10	2
11	2
12	3
13	4
14	7
15	11
16	23
17	52
18	119
19	255
20	503
21	1192
22	2231
23	4398
24	8443
25	19428

The growth is clearly exponential. An increase in even a single qubit can cause a significant increase in runtime. The steepness of the growth significantly increases after n=16. Hence, the runtime is a function of the number of qubits. Also, for the initial few runs when the number of qubits was lesser than 8, a lot of runs were recorded to have completed in 0ms which means that the order of runtime is lesser than milliseconds in those cases.

Grover's Problem

The difference in time for different qubit sizes for the same number of iterations and repeats was almost negligible(logs submitted). Instead we used the metric of elapsed ticks as that wasn't negligible. We notice that the number of elapsedTicks doesn't change significantly with respect to the number of qubits.

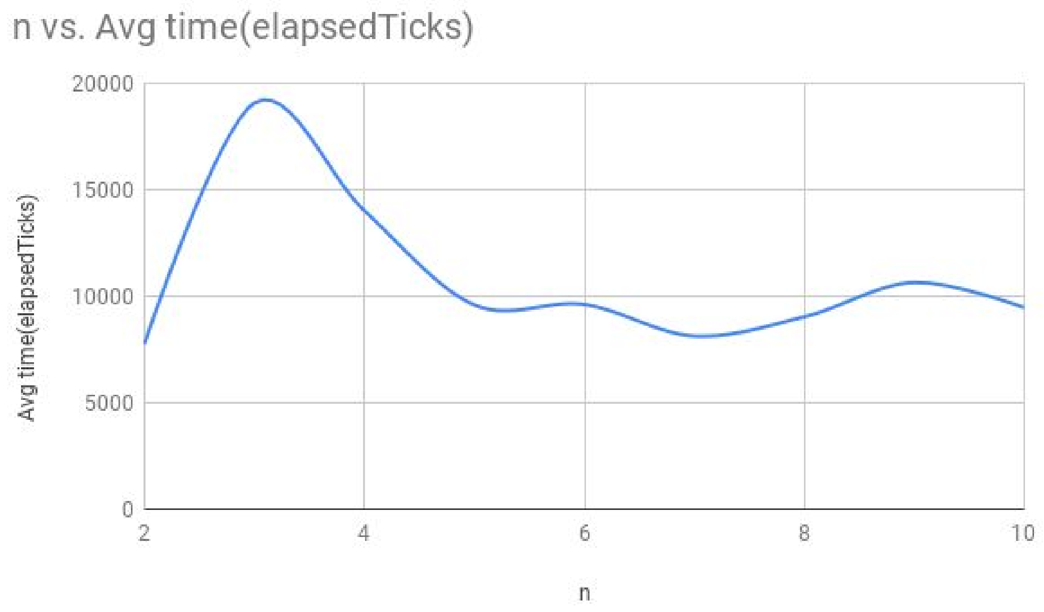


Figure 1.3.3: no. of qubits vs average elapsed ticks for iterations=10, repeats=10

Table 1.3.2: Data corresponding to Figure 1.3.3

n	Avg time(elapsedTicks)
2	7757
3	19073
4	14021
5	9591
6	9613
7	8132
8	9027
9	10648
10	9469

Simon's Problem

The test data provided in the Instances.json was used for the Simon's problem. The Kernel size refers to the number of qubits in the equations produced by the quantum circuit (dot product = 0) . As can be seen from the curves, there are minor variations in the time taken for the same kernel size. Also, these variations are not consistent between different various simulations. Generally speaking, with an increase in the number of qubits, the time taken by the quantum circuit also increases. For different U_f the results are different and dependent on the number of qubits.

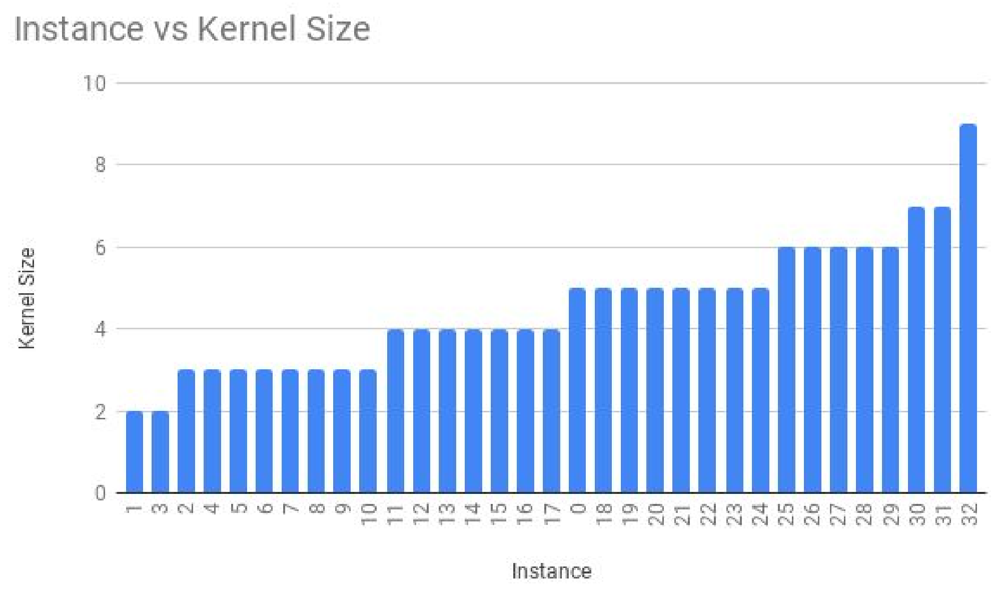


Figure 1.3.4: Instance vs Kernel Size for test data

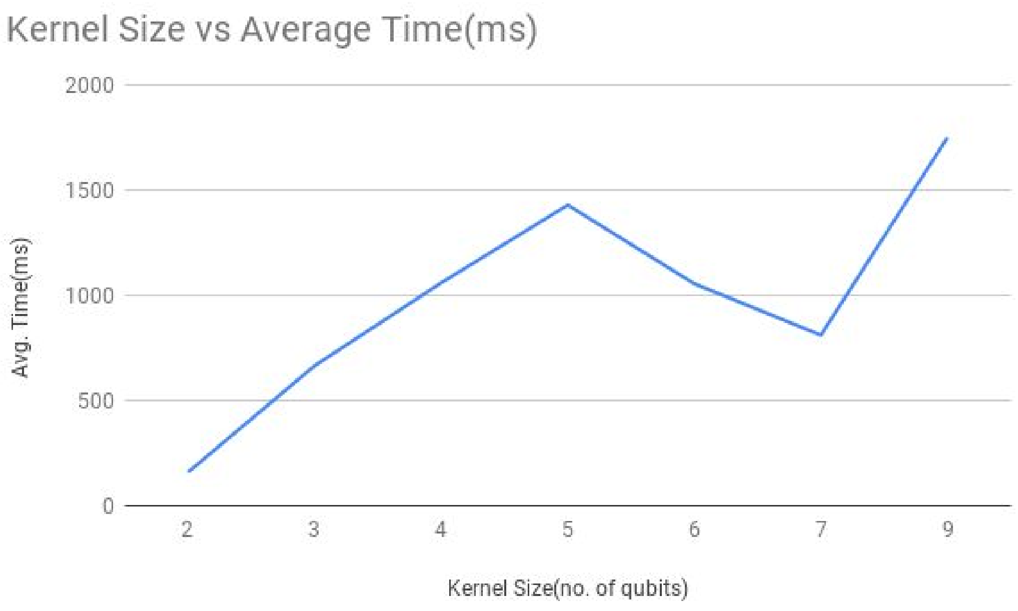


Figure 1.3.5: Kernel size vs Avg. Time taken by Quantum Circuit

Table 1.3.4: Data Corresponding to Figure 1.3.4 and 1.3.5

Instance no.	kernel size	time in ms
1	2	79
3	2	80
2	3	36
4	3	121
5	3	74
6	3	104
7	3	100
8	3	75
9	3	102
10	3	53
11	4	130
12	4	90
13	4	70
14	4	150
15	4	156
16	4	326
17	4	137
0	5	153
18	5	158
19	5	219
20	5	184
21	5	215
22	5	100
23	5	208
24	5	192
25	6	160
26	6	161
27	6	293
28	6	286
29	6	155
30	7	333
31	7	478
32	9	1750

Part 2. Instruction

The readme file has been attached with the submission files.

Part 3. Q#

3.1 List three aspects of quantum programming in Q# that turned out to be easy to learn and list three aspects that were difficult to learn

Easy

- The concept of creating quantum circuits through addition of gates and different operators turned out to be easy as there were ample examples and the flow was similar to other quantum programming libraries PyQuil and Cirq that we have seen so far.
- The concept of creating and manipulating Qubit instances was easy to learn as there are dedicated classes for these. Also there are many examples to illustrate this.
- Measurement of registers just means reading the values of the Qubit Arrays, this was much simpler than PyQuil and Cirq where these were explicit operations that returned separate objects. In Q#, the measurement is an operation, but it modifies the existing register.

Difficult

- Steep Learning Curve - Dependent on C#: The fact that Q# is very much dependent on C# for many of its features, adds a steep learning curve for most beginners. This is especially difficult for programmers not used to Object Oriented Programming.
- Steep Learning Curve - QS and CS files both are required for developing quantum programs on Q#. Initially it is not very intuitive for a programmer writing code from scratch as to how to organize the quantum program.
- Python notebooks provided along with the many sample algorithms are arcane and difficult to understand. A python programmer will have difficulty dissecting the program. Also, python notebooks are not yet present for many programmers.

3.2 List three aspects of quantum programming in Q# that the language supports well and list three aspects that Q# supports poorly

Supports Well:

- Mathematical Operations: Standard mathematical operations like Complex Number arithmetic functions, trigonometric functions are provided in the Quantum math library. This makes it easier to create quantum circuits.

- Unit tests: Q# supports creating unit tests for quantum programs, and which can be executed as tests within the xUnit testing framework. This makes testing and debugging of quantum circuits a reality, both cirq and pyquil don't provide this feature natively. Further, the tests can be run just by basic annotations and function name matching, making it a smooth experience for the programmer.

- Interfacing with Classical Languages: The interface to the classical languages of C-sharp has been implemented well, programmers are able to log messages from Quantum circuit runs in their classical programs. Additionally, the rich features of C# can also be used while developing Q# programs.

- Debugging: Since Q# has extensions that work on Visual studio code, programmers are able to actually debug the classical portions of the circuit. This is very helpful in narrowing down errors.

Supports Poorly:

- Diagnostics or Profiling of Quantum methods can't be run directly in the QS files, but the classical driver functions have to implement these profiling features. Since, our assignment required us to measure the time taken with increasing qubits, this would have been a useful feature. This can be a good addition to the Diagnostics package provided by Q#.
- Poor interface with Python - The reason programmers may prefer Python over Q# is due to its simplicity. With the arcane and complex code required to make Q# run on Python, the point of building a Python interface is defeated.
- Execution Time Inconsistency: Time taken by the first run of the quantum algorithm, for example in the Grovers and Simon's algorithm is much larger than other runs of the algorithm. This is possibly due to requirement of warming up of the Simulator and dry runs around it, however, even certain iterations after warm up report anomalous values.

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

Q# is heavily dependent on Visual Studio and dotnet right now. Q# also requires the splitting of code into a QS file and CS file. And lastly, it is based on F# and also loosely on C# that a lot of new coder's are not familiar with. Due to all of these reasons, Q# takes up a lot of time initially for amateur coders to understand. Ofcourse, after mastering the language well, Q# is seen to have a lot of prominent features compared to other Quantum languages. But the initial setup and learning of Q# can take a lot of time and hence make it difficult for programmers to get started with,

3.4 List three positives and three negatives of the documentation of Q#

Positives:

1. Introduction to Quantum Programming: The introduction to Quantum Computing contains all useful concepts like Dirac notation, Measurement of Qubits, Quantum gates, circuits and oracles. Every concept has been explained in detail and code snippets relating to these concepts have also been discussed. This section is better described than that of PyQuil and Cirq
2. Github comments and description: Along with the documentation, Q# has provided two github repositories: https://github.com/microsoft/Quantum and https://github.com/Microsoft/QuantumKatas#kata-as-project. Both of these repositories are very well documented and commented. Every function's description is very clear and detailed.
3. Quantum chemistry and Quantum Numerics library: The description of quantum chemistry is quite lacking in PyQuil. There are few examples available, and cirq has a seperate Open Fermion library. But both of these do not have any documentation for amateur users to explain its utilisation.
   Q# on the other hand has a very extensive documentation explaining algorithms for quantum chemistry and quantum numerics and their implementation in Q#.
4. Documentation for contributors: Unlike PyQuil and CirQ, Q# has a lot of information for contributors who would like to contribute to the framework. It has instructions regarding contributions, reporting bugs, style guides, opening pull requests, contributing documentations, and contributing code.

Negatives:

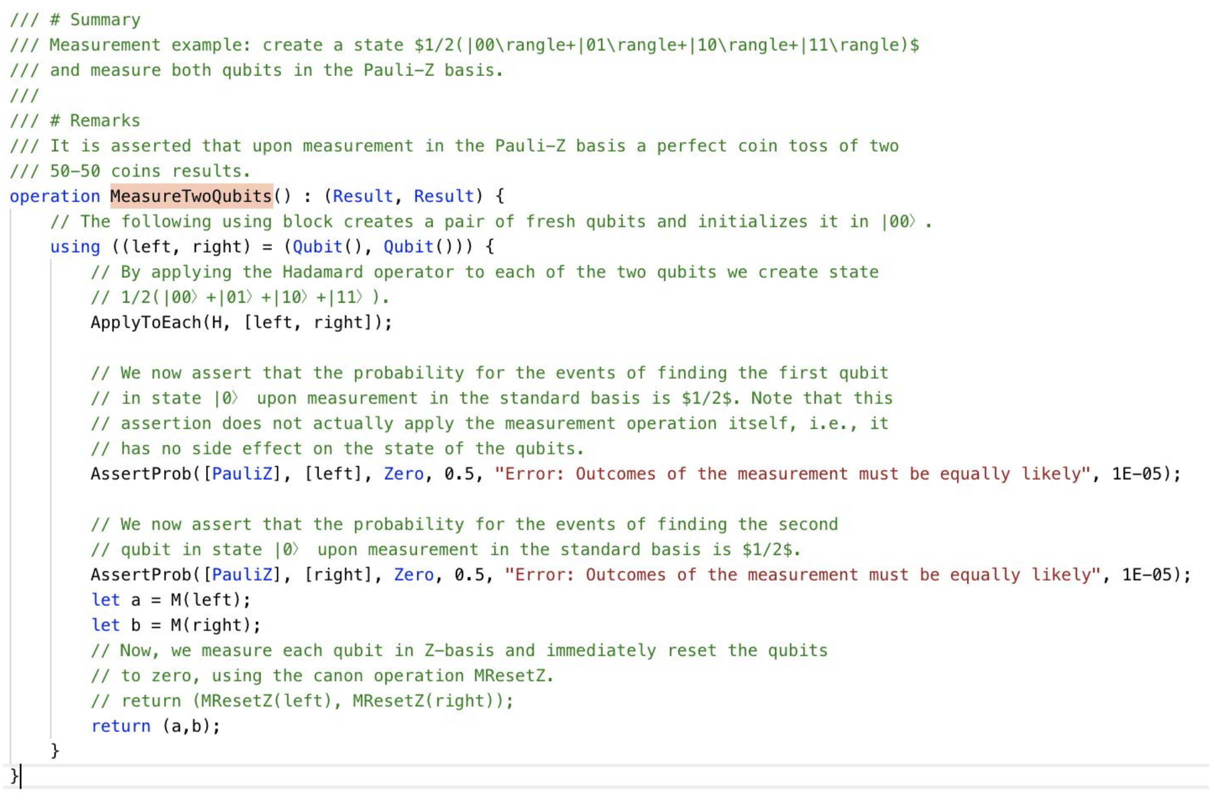
1. Scare Documentation: Unfortunately the documentation is very basic. It has very few details and does not have enough information about a lot of features. Unlike PyQuil and CirQ that have extensive documentation containing any quantum related feature/algorithm that one might require, Q# documentation is quite brief. Also, a lack of links across Q# documentation pages makes it harder to look for information
2. Dependency on Visual Code: Most of the Q# documentation including the github repository have instructions based on the usage through visual studio code. This feature makes it harder to run independent code and forces users to involve Visual studio in their coding process.
3. Basic API Reference: In macy cases, the API Reference for Q# does not have any examples. It only contains function prototypes. For a user that essentially is a heavy user of the API reference and is not starting by reading the whole of the documentation, coding in Q# can be a difficult task.

3.5 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#

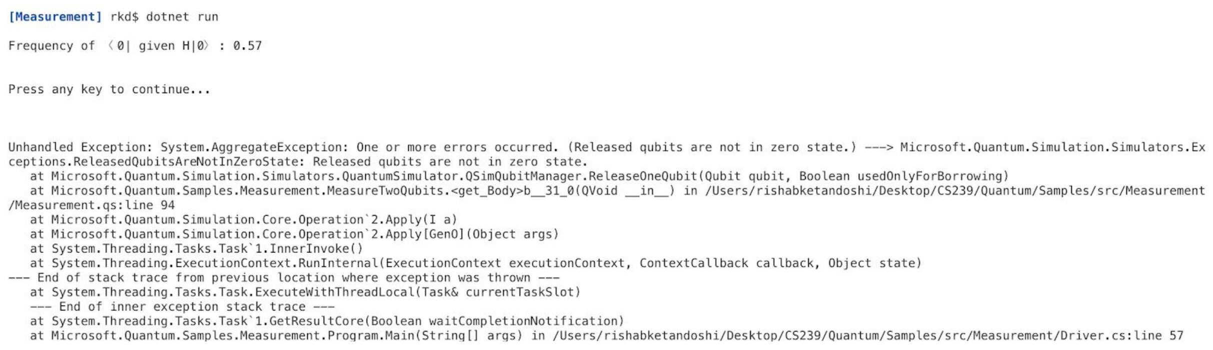
Quantum programming concepts	**Q# counterparts**
Bell State	EPR Pair
Complex Conjugate	Adjoint
X, Y, Z, H and S gates together	Clifford group
Probabilistic succession of an algorithm	Repeat-until-success
Adjoint and Controlled Operations	Adj + Ctl
Application of Quantum Operation	operation

3.6 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.

To verify that the using statement really enforces the constraint that the Qubits created must be in the original |0> state, we use the code in Measurement.qs. We change the return statement which returns a tuple of `(MResetZ(left), MresetZ(right))`. These functions reset the qubits left and right to the |0> state. We commented out the line `(MResetZ(left), MResetZ(right))` and instead sent a and b values which are measuring left and right values not resetting to 0. This leads to a runtime error as shown in the next figure. It throws `System.AggregateException`.



```
/// # Summary
/// Measurement example: create a state $1/2(|00\rangle+|01\rangle+|10\rangle+|11\rangle)$
/// and measure both qubits in the Pauli-Z basis.
///
/// # Remarks
/// It is asserted that upon measurement in the Pauli-Z basis a perfect coin toss of two
/// 50-50 coins results.
operation MeasureTwoQubits() : (Result, Result) {
    // The following using block creates a pair of fresh qubits and initializes it in |00〉 .
    using ((left, right) = (Qubit(), Qubit())) {
        // By applying the Hadamard operator to each of the two qubits we create state
        // 1/2(|00〉 +|01〉 +|10〉 +|11〉 ).
        ApplyToEach(H, [left, right]);

        // We now assert that the probability for the events of finding the first qubit
        // in state |0〉  upon measurement in the standard basis is $1/2$. Note that this
        // assertion does not actually apply the measurement operation itself, i.e., it
        // has no side effect on the state of the qubits.
        AssertProb([PauliZ], [left], Zero, 0.5, "Error: Outcomes of the measurement must be equally likely", 1E-05);

        // We now assert that the probability for the events of finding the second
        // qubit in state |0〉  upon measurement in the standard basis is $1/2$.
        AssertProb([PauliZ], [right], Zero, 0.5, "Error: Outcomes of the measurement must be equally likely", 1E-05);
        let a = M(left);
        let b = M(right);
        // Now, we measure each qubit in Z-basis and immediately reset the qubits
        // to zero, using the canon operation MResetZ.
        // return (MResetZ(left), MResetZ(right));
        return (a,b);
    }
}
```



```
[Measurement] rkd$ dotnet run

Frequency of 〈0| given H|0〉 : 0.57


Press any key to continue...



Unhandled Exception: System.AggregateException: One or more errors occurred. (Released qubits are not in zero state.) ---> Microsoft.Quantum.Simulation.Simulators.Ex
ceptions.ReleasedQubitsAreNotInZeroState: Released qubits are not in zero state.
   at Microsoft.Quantum.Simulation.Simulators.QuantumSimulator.QSimQubitManager.ReleaseOneQubit(Qubit qubit, Boolean usedOnlyForBorrowing)
   at Microsoft.Quantum.Samples.Measurement.MeasureTwoQubits.<get_Body>b__31_0(QVoid __in__) in /Users/rishabketandoshi/Desktop/CS239/Quantum/Samples/src/Measurement
/Measurement.qs:line 94
   at Microsoft.Quantum.Simulation.Core.Operation`2.Apply(I a)
   at Microsoft.Quantum.Simulation.Core.Operation`2.Apply[GenO](Object args)
   at System.Threading.Tasks.Task`1.InnerInvoke()
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object state)
--- End of stack trace from previous location where exception was thrown ---
   at System.Threading.Tasks.Task.ExecuteWithThreadLocal(Task& currentTaskSlot)
   --- End of inner exception stack trace ---
   at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)
   at Microsoft.Quantum.Samples.Measurement.Program.Main(String[] args) in /Users/rishabketandoshi/Desktop/CS239/Quantum/Samples/src/Measurement/Driver.cs:line 57
```

For borrowing, we couldn't run the same program as the program refused to build, but by looking at the documentation we realize that borrowing is akin to obtaining bits for random temporary use. The qubits are in unknow state and go out of scope at the end of the statement block, the borrower should leave the qubits in the same state as in when they were borrowed. They need not be in a classical state. So, to test this out we would try to NOT the Qubit using Paulis X gate. We anticipate that if we don't apply another Pauli's X, there will be a similar runtime exception.