QISKIT - THE ENGINEERING GROUP

QUANTUM PROGRAMMING

Attiano Purpura-Pontoniere attiano@cs.ucla.edu **Nishant Sabharwal** nsabharwal@ucla.edu

Pratik Sathe psathe@physics.ucla.edu

May 29, 2020

ABSTRACT

We use Qiskit (version = 0.14.1) to simulate four popular quantum algorithms (Deutsch-Jozsa, Bernstein-Vazirani, Simon's and Grover's search algorithms) on a classical computer. For each of the algorithms, we benchmark the performance of our simulations, measuring the scalability of compilation and execution with the number of qubits and dependence of the execution time on the oracle U_f. We then discuss our code organization and implementation of quantum circuits. Finally, we provide some comments about Qiskit and its documentation, and provide some suggestions for improving the user experience.

1 Design and Evaluation

1.1 Deutsch-Josza algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}$ *which is either balanced or constant, determine whether it is balanced or constant.*

While a classical algorithm requires $2^{n-1}+1$ calls to the function f, the Deutsch-Jozsa algorithm finds a solution with just one call to the oracle U_f, which is defined by its action on the computational basis as follows:

$$U_f \left|x\right\rangle \otimes \left|b\right\rangle = \left|x\right\rangle \otimes \left|b \oplus f(x)\right\rangle, \tag{1}$$

for all $x \in \{0,1\}^n$ and $y \in \{0,1\}$. The Deutsch-Jozsa circuit is shown in Fig. 1. If the measured state is $\left|0\right\rangle^{\otimes n}$, then the algorithm concludes that f is constant, and balanced otherwise.

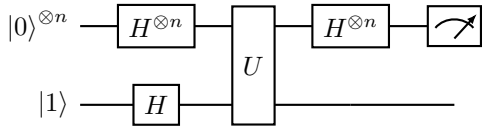


Figure 1: Deutsch-Jozsa algorithm

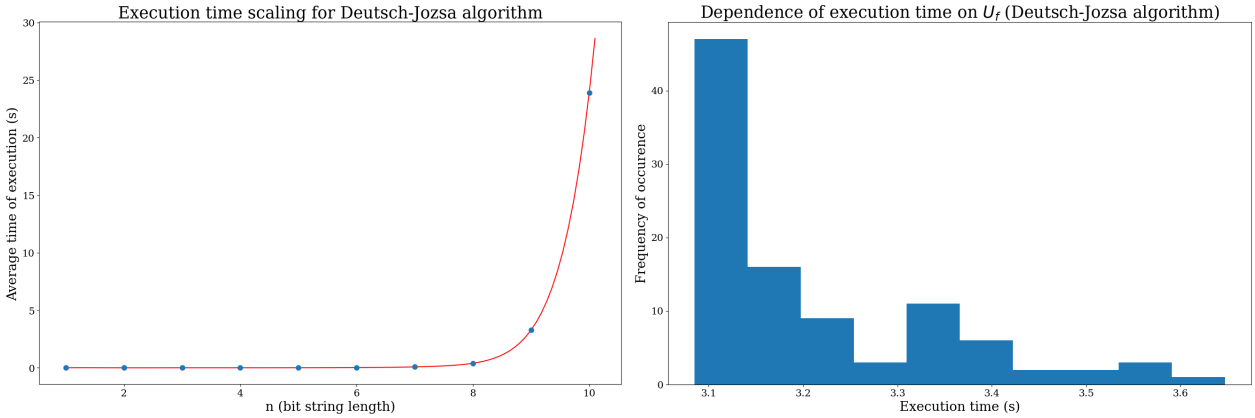
Now we discuss various aspects of our code design:

- **Implementation of** U_f: Before implementing the unitary matrix U_f, it is first necessary to obtain the matrix representation in the computational basis. Care must be taken here since Qiskit is little endian. To that end, we implement the following pseudo-code:
  1. Initialize U_f to be a $2^{n+1} \times 2^{n+1}$ zero matrix.

2. For every possible bit string a of length n, and bit b, representing the helper bit, obtain $x = a + b$ and $y = a + (b \oplus f(a))$, with '+' denoting string concatenation. We let i be x bitwise reversed and j be y bitwise reversed. We then convert i and j to their decimal representations and set the row-i, column-j entry of U_f to 1.

This procedure ensures that U_f maps all the computational basis vectors according to the definition (1). After that, we can simply pass the matrix in to `Operator` to generate a gate. This is abstracted away in a function `create_uf`, which calls a subroutine `build_matrix` to generate the matrix from f. The code is neat and modular. A single for loops creates the matrix and the user can simply pass in f and get a gate without worrying about the implementation.

- **Code readability:** To improve readability, we implemented a class called `Program` that organizes all the functions. The class has modular methods that build and run the circuit so the user can call a high level function to build and run the circuit or lower level functions to use it as they please. We will briefly describe the role of each function in our code below:
  1. `run_dj(f, num_shots)`: High level function which generates the oracle U_f, compiles the deutsch-josza circuit, runs it and returns whether it is constant or balanced. It takes an optional parameter `num_shots` for how many times to run it.
  2. `build_circuit(n, uf)`: Assembles the deutsch-jozsa circuit and returns a `QuantumCircuit` object. It creates a circuit of $n + 1$ wires, applies H to all wires, applies U_f, then applies H and measures the n qubits.
  3. `measure(circuit, backend, num_shots)`: This takes a circuit runs it and returns a counts dictionary. Optional parameters include the backend to use and the number of times to run it.
  4. `evaluate(counts, n)`: This evaluates the counts dictionary and returns whether the input function was constant or balanced. If 0^n is ever measured, it reports constant.
  5. `create_uf(n, f)`: This takes the input functions, generates a unitary matrix, and returns the Operator object.
  6. `build_matrix(n, f)`: Takes the input function and builds the matrix as described above.
  7. `to_int(bit_string)`: Converts an array of 0s and 1s to it's numeric representation.
  8. `bit_string(i, n)`: Converts an int to a bit string of length n.
  9. `generate_functions(n, fully_random, random_trials)`: Generates test functions for a given arity. Since the number of test functions is large, there are parameters for controlling the number. It returns a list of tuples of whether it's constant or balanced and function, represented by a 1 x 2^n array representing f(x) for each input.
- **Preventing access to U_f implementation:** We discuss this in detail at the end of the document.
- **Parametrizing the solution in n:** The circuit is different for every value of n. Two parts of the circuit depend on n. Creating the U_f gate is dependent on n. We have a function which builds the matrix dynamically depending on n, which is then passed to the gate operator. The `build_circuit` function takes n as a parameter and then creates circuit with $n + 1$ wires. It then uses n in a for loop to apply H to the correct wires and to plug the correct wires into U_f. The high level function interpolates n from the length of the input function so the user does not need to worry about the different sizes.
- **Code testing:** We took an exhaustive approach to testing. For any given function f, our function `run_dj()` returns 1 if it determines that f is constant, and 0 if the function is balanced. For each n, $N = 2^n$, there are exactly 2 constant functions and $\binom{N}{\frac{N}{2}}$ balanced functions. For n up to 3, we test all of them. After this we select 100 functions at random, always including the 2 constant. We then ran it with the high level function `run_dj` which returns the output. We used `generate_functions` which returned a list of tuples of the expected answer and the function. We then compared the output against the expected answer. We repeated this from $n = 1$ to $n = 10$.
- **Dependence of execution time on U_f:** To view the impact U_f had on the execution time, we ran the circuit 100 times for U_fs selected at random. It did have an effect. Most took 3 seconds but some took up to 3.6 seconds. Qiskit has a much tighter range than PyQuil. Most runs had the same execution time except for a few outliers on the longer side. U_f has a much smaller effect on execution time in Qiskit. A histogram of the recorded execution times can be seen in figure 2b.
- **Scalability:** We observed that the execution time for the circuit simulation grows exponentially with the number of qubits involved, i.e. $n + 1$. For each n from 1 to 10, we executed the Deutsch-Jozsa algorithm for five randomly chosen U_f's, and obtained the average execution time for each value of n. As can be seen in figure 2a, the execution time grows exponentially, with $n = 10$ requiring about 30 seconds of execution time. This is much faster than PyQuil which reached into the minutes at $n = 6$.



(a) Execution time scaling.

(b) Execution time histogram ($n = 9$).

Figure 2: Deutsch-Jozsa algorithm: Benchmarking.

1.2 Bernstein-Vazirani algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \rightarrow \{0,1\}$ *of the form* $f(x) = a.x + b$*, with* $a \in \{0,1\}^n$ *and* $b \in \{0,1\}$ *are unknown bit strings, obtain the value of* a*.*

While a classical algorithm requires $\mathcal{O}(n)$ calls to the function f, the Bernstein-Vazirani (BV) algorithm obtains the bit string a with just one call to the oracle U_f, which satisfies equation (1). The quantum circuit for the BV algorithm is the same as that of the DJ algorithm (Fig. 1). The BV algorithm determines a to be the state of the n qubits measured at the end of the circuit.

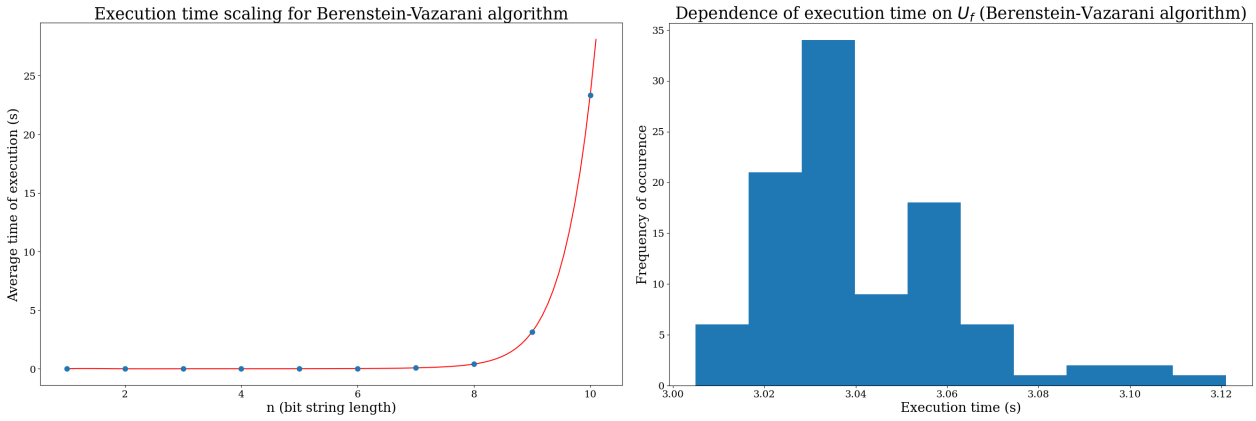
Now we discuss various aspects of our code design:

- **Implementation of** U_f: Given a function f which satisfies the required criteria, our procedure for obtaining U_f for the BV algorithm is identical to that of the DJ algorithm. Once the matrix is defined, we Qiskit's `operator` function which we pass the matrix into. The resulting operator can be appended to each wire in Qiskit.
- **Code readability:** Similar to DJ, we define a class `Program` to organize all the code. It executes small methods to build and run the circuit. Again a high level function is defined so the user can simply call. This creates the oracle U_f, assembles the circuit, runs it, and returns a and b.
  In our implementation of the BV algorithm, we reuse the functions `build_circuit`, `measure`, `create_uf`, `build_matrix`, `to_int`, and `bit_string`. The only new functions used are:
  1. `run_bv(f, num_shots)`: High level function which generates the oracle U_f, compiles the berenstein-vazarani circuit, runs it and returns a and b. It takes an optional parameter `num_shots` for how many times to run it. It measures a by running the circuit. It finds b by calling f(0).
  2. `evaluate(counts, n)`: This evaluates the counts dictionary and returns a. Note it reverses a to account for endian-ness. Since this is a simulator with no noise, the circuit measures a with 100% probability. It throws an error if there is more than one measurement.
  3. `generate_functions(n, fully_random, random_trials)`: Generates test functions for a given arity. Since the number of test functions is large, there are parameters for controlling the number. It returns a list of tuples, the first item being the tuple a and b and the second being the function, represented by a 1 x 2^n array representing f(x) for each input.

  By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code readable.
- **Preventing access to** U_f **implementation:** We discuss this aspect at the end of the report.
- **Parametrizing the solution in** n: The circuit is different for every value of n. Similar to DJ, we parametrize n by building the matrix dynamically. The `build_circuit` takes n as an argument so it can create $n + 1$

wires, apply H to all, plug the wires into U_f, and then apply H and measure to the first n. H is applied by a for loop and the wires are plugged into U_f using Python's `range` function to generate a list from 0 to n. The user does not need to be aware of this if they use the high level function. if they build the circuit with the low level function, they must simply pass n as a parameter.

- **Code testing:** Again, we used an exhaustive method to generate functions. For a given function f, there are 2^n possible values of a and 2 possible values of b. There are 2^{n+1} possible functions. For $n = 1$ to $n = 3$, we test all. For $n > 3$, we select 100 at random. We repeated this for $n = 1$ until $n = 10$. `generate_functions` would return a list of test functions containing the expected value for a and b. We then called the high level function on each f and made sure the calculated values of a and b were correct.
- **Similarities in our codes for the BV and DJ implementations**: We estimate about 90% code similarity between our BV and DJ algorithms. In particular, the functions `build_circuit`, `measure`, `create_uf`, `build_matrix`, `to_int`, and `bit_string` were identical. To comply with submission, we copied and pasted but these functions could be shared from an imported library. The main run function was nearly identical, except with BV we need to call f(0) to calculate b. `evaluate` was different since BV measures the value of a while DJ is looking for 0^n. Most of the difference came from `generate_functions`. The similarity in the codes is due to the fact that the circuits and oracle definition in both these algorithms are identical.
- **Dependence of execution time on U_f**: Similar to the DJ algorithm, we observe a variance in the execution time with U_f. For $n = 9$, we simulated 100 randomly chosen U_f's, and plotted the histogram in figure 3b. Again this forms a tight spread around 3 seconds with a range of 0.11 seconds. This is a much more reliable time than n PyQuil. Also, it forms closer to a normal curve than the DJ profile.



(a) Execution time scaling.

(b) Execution time histogram ($n = 9$).

Figure 3: Bernstein-Vazirani algorithm: Benchmarking.

- **Scalability:** Similar to the DJ algorithm, we observed an exponential increase in the execution time with n. For $n = 1$ to $n = 10$, we obtained the average execution time for five randomly chosen U_f's. As can be seen in figure 2a, the execution time grows exponentially, with $n = 10$ requiring about 25 seconds of execution time. Again, this was much faster than the implementation on PyQuil which would take 15 minutes for $n = 6$.

1.3 Simon's algorithm

First, we summarize the problem statement:

Given a function $f : \{0,1\}^n \to \{0,1\}^n$ *such that* $f(x) = f(y)$ *iff.* $y \oplus x \in \{0, s\}$, *find the value of* s. *If no such string* s *exists, return* 0^n.

The classical implementation requires testing up to $2^{n-1} + 1$ inputs until two inputs with the same output are found. If none are found, s is 0 and f is 1 to 1.

Simon's algorithm consists of a quantum circuit, which is shown in figure 4. The circuit involves n helper qubits, instead of just 1, as in the case of DJ and BV algorithms. The oracle U_f satisfies equation (1), except with the modification that b is now a bitstring of length n.

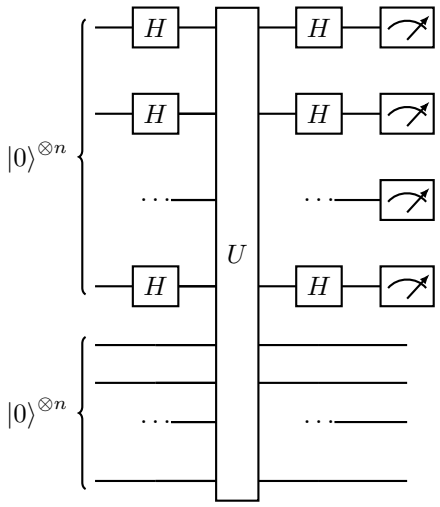


Figure 4: Simon algorithm

Each time Simon's circuit is run, the result is a y which satisfies $y \cdot s = 0$. If the circuit is measured n-1 times, and all values of y are independent, the system of equations can be solved with 2 values of s, one of them being 0. However, the n-1 values of y may not be all independent so it may need to be measured more times. Though it's possible to keep measuring dependent vectors, there is a 99% probability that $4 * m * (n - 1)$ measurements, where $m = 5$, will result in n-1 independent equations.

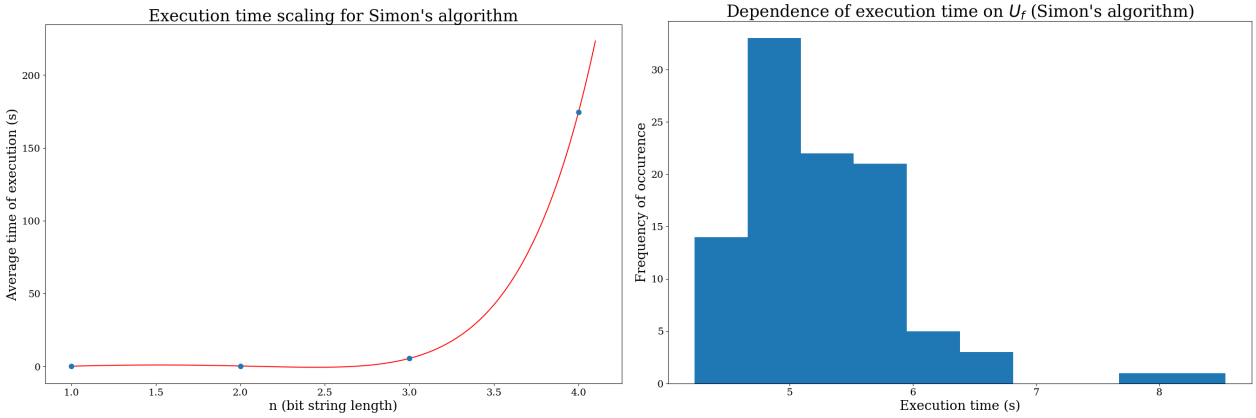
Now we discuss various aspects of our code design:

- **Implementation of U_f**: Similarly, we construct a matrix from the function f. The pseudo-code is described below.
  1. Initialize U_f to be a $2^{2n} \times 2^{2n}$ zero matrix.
  2. For every possible bit strings x and b of length n, compute the bit string $y = b \oplus f(x)$. Since $x + b$ maps to $x + y$, we set the corresponding matrix element to 1 (here $+$ denotes string concatenation.)
- **Code readability:** We define a class `Program` which contains the various functions needed for our implementation. We allow the user access to higher level implementation, as well as lower level implementation. We have also included functions for analysis and plotting of benchmarking of the algorithm.
  Here is a description of the various function we have defined:
  1. `simon_solution(f, num_shots)`: For a higher level implementation of the algorithm. `f` denotes the input function, and `num_shots` which is set to 20 by default, denotes the number of times y is to be measured.
  2. `generate_random_s(n)`: Creates a random string s of length n (useful for testing and benchmarking).
  3. `s_function(s)`:Given a string s which may or may not be all zeros, construct a function f corresponding to s.
  4. `create_Uf(f)` Given a function f, construct the corresponding U_f matrix.
  5. `get_Simon_circuit(Uf)`: Creates a `QuantumCircuit` object consisting of Simon's circuit, using the U_f provided as an input.
  6. `run_created_circuit(circ, num_shots)`: Runs the circuit `num_shots` number of times. By default, this value is set to 20. Returns a list of y values.
  7. `s_solution(list_y)`: Obtains solutions for the set of equations $y.s = 0$, with y being an element of `list_y`.
  8. `verify_Simon_output(f, list_s)`: Checks if the output of Simon's algorithm is correct for a given f.

  We also include a number of functions for automated computation, storage and plotting of time scaling with n, checking correctness for random input functions, and dependence of execution time on U_f.

By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code readable. We also include docstrings and useful comments in each function.

- **Preventing access to U_f implementation:** We discuss this aspect at the end of the report.
- **Parametrizing the solution in n:** The circuit is different for every value of n. Similar to DJ and BV, we parametrize n by building the matrix dynamically. The `get_Simon_circuit(Uf)` takes the matrix U_f as an argument, obtains the value of $2n$ from the dimension of U_f, and creates a circuit with $2n$ qubits. By using `QuantumCircit.h(i)` in for loops we add the required Hadamard gates to the first n qubits. We use `QuantumCircuit.iso()` in order to add U_f to the circuit.
- **Code testing:** We defined a function `verify_Simon_output(f,s_list)`, which checks if the output `s_list` obtained from Simon's algorithm contains the correct solution(s) corresponding to function `f`. Within the function `check_correctness`, for any given value of n, and for a specified value of `num_shots`, we create a specified number of random Simon's functions f, run the algorithm `num_times` number of times, and verify the correctness of each output. We found that for n ranging from 1 to 4, and `num_times` set to 100, all the outputs were verified to be correct by the `verify_Simon_output` function, since we the number of measurements to $4m(n-1)$. This is expected, since the probability of success is $> 99\%$ theoretically, for such a high number of measurements.
- **Similarities in our codes with the other algorithms**: Since Simon's algorithm is based on a completely difference circuit, the majority of the code is different from that of the other algorithms. Additionally, Simon's algorithm also includes some classical post-processing, which is not present in the algorithms.
- **Dependence of execution time on U_f**: For each value of n ranging from 1 to 3, we obtained 100 random U_f's and recorded the execution times (see figure 5b for the histogram for $n = 3$). Although the dependence on U_f is not significant, we found that if $s = 0$, then the execution time is larger, which agrees with the fact that in such cases, U_f has more off-diagonal entries than if s were non-zero.



(a) Execution time scaling.

(b) Execution time histogram ($n = 3$).

Figure 5: Simon's algorithm: Benchmarking.

- **Scalability:** Similar to the the other algorithms, we observed an exponential increase in the execution time with n. For each value of n ranging from one to four, we obtained the average execution time for 20 randomly chosen U_f's. As can be seen in figure 5a, the execution time grows exponentially, with $n = 4$ requiring about 3 minutes execution time. We limited ourselves to $n < 5$, since it took more than half an hour for a single run of the $n = 5$ circuit.

1.4 Grover's search algorithm

First, we summarize the problem statement:
Given a function $f : \{0,1\}^n \to \{0,1\}$ that returns $f(x) = 1$ for exactly $|x| = a << n$ inputs, find one of the inputs x that make f return 1.

Solving this problem on a classical computer requires 2^n queries to f, but with Grover's algorithm we can solve it with high probability with $O(\sqrt{2^n})$ operations of the oracle Z_f. The algorithm involves the so-called Grover's

operator G, defined as

$$G \coloneqq -H^{\otimes n} Z_0 H^{\otimes n} Z_f \left|x\right\rangle, \tag{2}$$

with Z_f and Z_0 defined through their action on computational basis vectors $|z\rangle$ as follows:

$$Z_f \left|x\right\rangle = (-1)^{f(x)} \left|x\right\rangle, \tag{3}$$

$$\text{and } Z_0 \left|x\right\rangle = \begin{cases} -\left|x\right\rangle & \text{, if } \left|x\right\rangle = 0^n \\ \left|x\right\rangle & \text{, if } \left|x\right\rangle \neq 0^n. \end{cases} \tag{4}$$

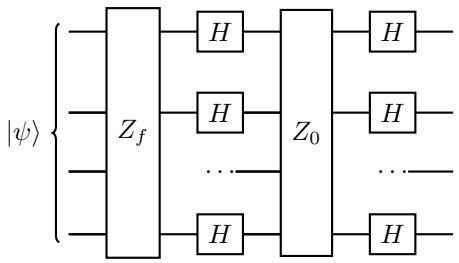
Grover's operator, G, is shown in Fig. 6a. In order to maximize the probability of success, the operator G is applied k number of times, with k being

$$\begin{aligned} k &= CI\left(\frac{\pi}{4\theta} - \frac{1}{2}\right) \\ \text{with } \theta &\coloneqq \arcsin\frac{a}{N}. \end{aligned} \tag{5}$$

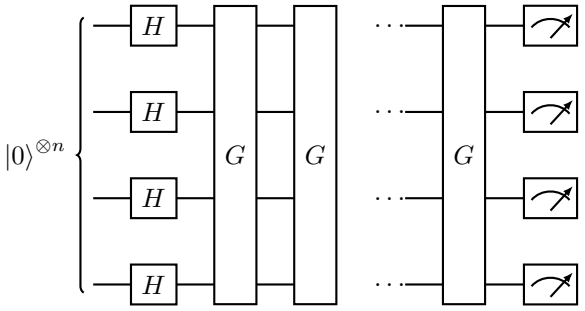
Here, $CI(y)$ denotes the integer closest to y. The complete circuit is shown in figure 6b. The probability of success (i.e. the probability of the final measurement resulting in a bitstring satisfying $f(x) = 1$) is given by

$$P_{success} = \sin^2((2k+1)\theta). \tag{6}$$

k (5) number of Grover's iterations G results in measurement of state $|x\rangle$ which satisfies $f(x) = 1$ with high probability, on the order of $95 - 99\%$.



(a) Grover's iteration G.



(b) Grover's circuit: Grover's operator G occures $k = \mathcal{O}(\sqrt{N})$ times.

Figure 6: Grover's search algorithm

Now we discuss various aspects of our code design:

- **Implementation of U_f, i.e.** G: Before implementing the unitary matrix U_f that corresponds to (2), it is first necessary to obtain the matrix representation in the computational basis. To that end, we need to implement Z_0 and Z_f with the following pseudo-code:
  1. Initialize Z_0 and Z_f to be a $2^n \times 2^n$ identity matrix.
  2. For Z_0 set $I_{00} = -1$. For Z_f set all $I_{f(x)=1} = -1$.
  3. Create a $2^n \times 2^n$ Hadamard matrix
  4. Obtain G using (2) given 2/3
  5. Obtain k (number of Grover iterations) which maximize the success probability.

  This procedure ensures that G in accordance with (3),(4),(2). Once the matrix is defined, we call `Operator()` on it from Qiskit and create a Gate object for (2). This Gate object can then be appended to the circuit, applying its operation to the number of qubits that the gate-defining matrix was built for.
- **Code readability:** We split our code for Grover's algorithm into an object with multiple small functions, each of which executes a simple task. A user with a function f only needs to access one function in a copy of our $grover$ object: (i) The function `grover.run_grover(f,n,a)` which executes the Grover algorithm, and returns a $|x^n\rangle$ in the form of a list that matches an input x to the function f(x) that evaluates to 1. It also returns the run time for execution of 1000 trials, the numerical probability of success based

on those trials, and the calculated theoretical probability of success. The oracle function f is passed in to `grover.run_grover(f,n,a)` as well as n, the number of qubits, and a, the number of inputs x that evaluate to f(x) = 1.
We will briefly describe the role of each member function in our $grover$ object that defines our code below:

1. `grover.run_grover(f,n,a)`: Executes the Grover algorithm and returns the result, the run time for 1000 trials, the numerical probability of success based on those trials, and the calculated theoretical probability of success. The inputs are the oracle gate definition, the value of n, the number of qubits, and a the size of the domain of f that evaluates to 1.
2. `grover.create_circuit(G,n,a)`: Creates the qiskit circuit object that runs a grover's algrorithm iteration k times on the oracle function f for n qubits and a, the number of inputs x that evaluate to 1. Also returns the theoretical probability of success. The inputs are the oracle gate definition from qiskit's Operator() function on the numpy matrix that corresponds to G, the value of n, the number of qubits, and a the size of the domain of f that evaluates to 1.
3. `grover.H_tensored(n)`: Creates an 2^nx2^n Hadamard matrix. Used to implement the $H^{\otimes n}$ at the start and end of (2).
4. `grover.execute_grover(circuit,n,f,num_shots=1000,backend='qasm_simulator')`: Executes the input qiskit circuit defintion num_shots times that runs grovers algorithm k times on the oracle function f. Uses the simulator defined by $back_end$.
5. `grover.check_correctness(func,result,n)`: Checks to see if the result from Grover's algorithm evaluates to 1, assumes func is an array that maps the domain to range of f by indexing func[x]. The implementation does type checking to see if the func passed in is a function definition and or an array and checks correctness. This assumes the input to f is an int representing an unsigned bit string of n bits.
6. `grover.reverse_bit_int(i)`: Converts an integer to a bit string, flips the bit string, then converts the integer back to an int.
7. `grover.create_G(f,n)`: Creates the unitary matrix G that corresponds to a single iteration of Grover quantum circuit iteration 6b for n qubits.
8. `grover.all_f(n,a)`: Given a function f, creates and returns as a numpy matrix the corresponding oracle matrix Z_f in the computational basis for all possible functions f with a given number of qubits and number of values in the domain that evaluate to 1. There are $\binom{2^n}{a}$ such functions
9. `grover.calc_lim(a,n)`: Given a the number of qubits and the size of the domain of f that evaluates to 1, `calc_lim(a,n)` returns k, the number of times to run Grover's algorithm and the theoretical probability of success given that k. This function returns the floor or ceil of k that maximizes the probability of Grover's algorithm successfully finding a bit-string that is part of the domain of f that evaluates to 1.
10. `grover.convert_n_bit_string_to_int(x,n)`: Converts a list x of length n with entries of 0/1 that correspond to an unsigned encoding of an integer to an int x. `grover.convert_int_to_n_bit_string(integer, n)`: Converts an int integ to a list of length n with entries of 0/1 that correspond to an unsigned encoding of an integer.

By modularizing the algorithm simulation process into small functions executing well-defined tasks, we have made our code more readable and reduce code replication.

- **Preventing access to U_f implementation:** We discuss this aspect at the end of the report.
- **Parametrizing the solution in n:** The circuit is different for every value of n. In the function `grover.run_grover(f,n,a)`, we pass in a dynamically created G by calling `grover.create_G(f,n)` to make the matrix that corresponds to G and `grover.create_circuit(G,n,a)` with an arbitrary value of n and a. We then execute the dynamically sized circuits by using the `Aer.get_backend('qasm_simulator')` quantum simulator in `grover.execute_grover(circuit,n,f,num_shots=1000,backend='qasm_simulator')`, and run the circuit a sufficient number of times, k, such that the $|x\rangle$ input vector is rotated to match a vector $|x\rangle$ that evaluates as input on the oracle function to 1.
- **Code testing:** In order to verify that the algorithm's output is correct, we define a function `check_correctness`, which determines (classically) whether the measured state x satisfies $f(x) = 1$.
Grover's algorithm is probabilistic, succeeding with a high probability, but with a small probability of failure. In order to test that our implementation succeeds at a rate similar to the theoretical success rate (6), we implemented our code for a variety of combinations of n and a. For each combination, we ran our code over 1000 times, recording the fraction of times the algorithm succeeded. The theoretical results match the numerical results very well, as can be seen in the table below:

Row	n	a	Theoretical success rate	Numerical success rate
1	2	1	100%	100%
2	3	1	94.53%	94.7%
3	3	2	100%	100%
4	3	3	84.38%	84%
5	4	1	96.13%	96.5%
6	4	2	94.53%	94.30%
7	4	3	94.92%	95.30%

Table 1: Grover's Algorithm: Numerical rate of success matches with the theoretical value

- **Dependence of execution time on U_f**: In order to analyze the variance of the execution time with U_f, we implemented the following:
  1. We tested all 16 possible functions for $n = 4$ and $a = 1$. The fastest U_f was approximately $3x$ faster than the slowest U_f (see 7).
  2. We expect that for a fixed n and small a's, increasing the value of a should increase the execution time. We expect this because the larger the value of a, the more the deviation of G from the identity matrix. Hence, for $a = 1, 2, 3, 4, 5, 6$, we computed the average compilation and computation time over 1000 iterations for each case. Contrary to our expectation the computation time for $n = 6$ was roughly 3 seconds. A possible explanation for this is that Qiskit's simulator is quite fast and powerful, and computation time only depends on the size of the array, not how closely it matches the identity matrix. (see 8).

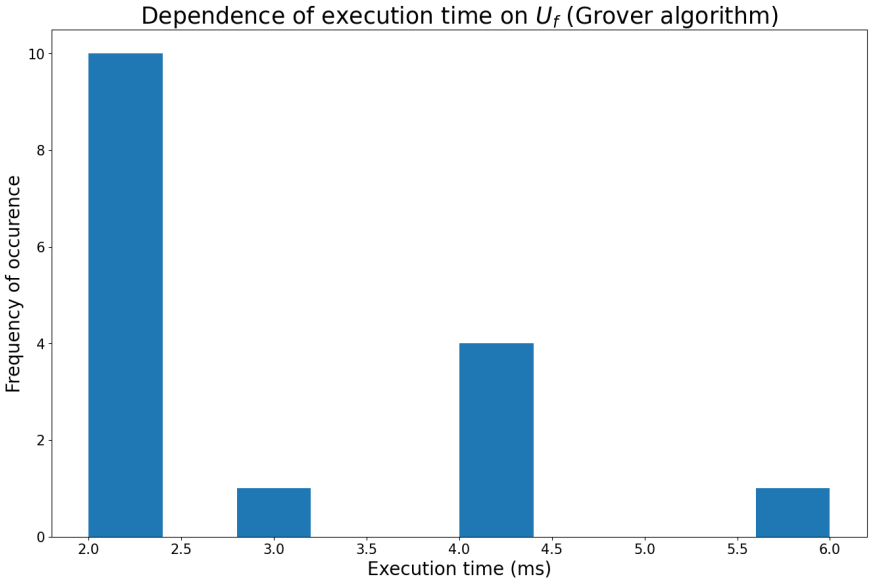


Figure 7: Execution time for all 16 possible U_f when $n = 4$ and $a = 1$.

- **Scalability**: We benchmarked the average execution time of the virtual quantum computer for every $n \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $a = 1$. We determined the average execution time for a given n by averaging over all possible functions for that n and $a = 1$. We observed that the execution time for the quantum circuit simulation grows exponentially with n. As can be seen in figure **??** the execution time grew exponentially, with $n = 6$ requiring about 18 seconds of execution time. We decided to limit our analysis to $n \leq 10$, since we observed that at $n = 11$ it was unfeasible to average over all possible functions as that would be roughly $11! * 18$ seconds of run time. Our observations are in line with the fact that the matrices grow larger exponentially as we increase n.

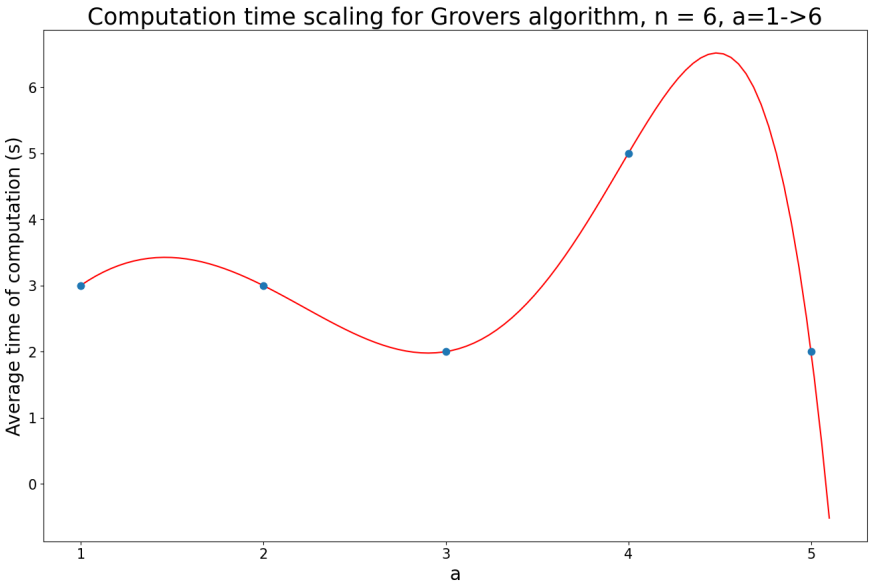


Figure 8: Average Execution time for U_f when $n = 6$ and $a = 1 \rightarrow 6$.

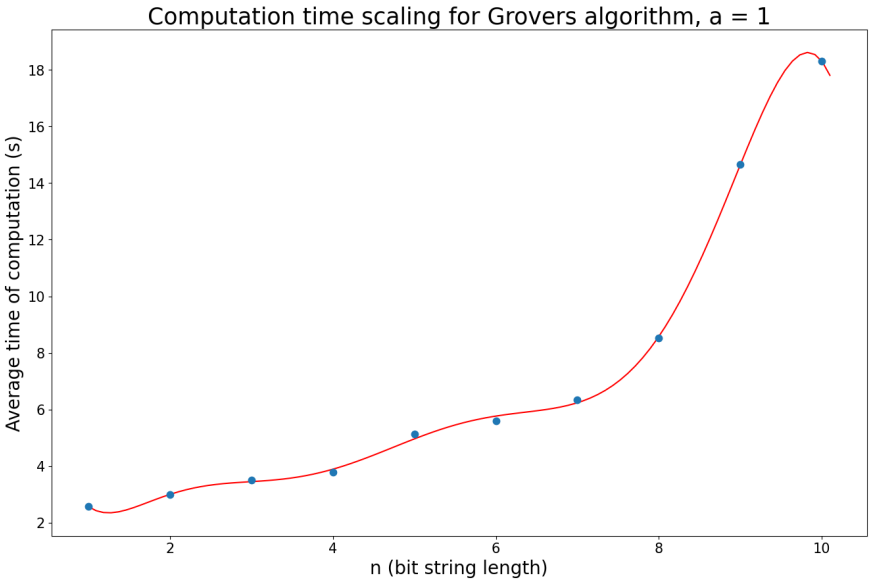


Figure 9: Execution time scaling for Grover's algorithm.

2 Outline: The Language & Documentation

Most of the assignment prompts have been answered in the sections above. We also summarize our approach and observations below.

1. Design and evaluation (Qiskit)
   (a) Present the design of how you implemented the black-box function U_f. Assess how visually neat and easy to read it is.
   $\rightarrow$ In order to create a readable and clean code, for each of the algorithms, we have a separate `.py` module. Within each module, based on functionality, we split the procedure into multiple small and readable functions, which can be run for a lower level implementation. We also have a provision for a higher level implementation, where the only input the user needs to provide is the function f. We provide docstrings

and useful code comments in order to make the code easier to understand.
As for the implementation of the oracle, for each algorithm, given a function f, we compute the corresponding matrix U_f, and add the gate object corresponding to it to the circuit in Qiskit (using either `iso`, or `Operator` methods). For more details, we refer the reader to the relevant sections above.

(b) Present the design for how you prevent the user of U_f from accessing the implementation of U_f. Assess how well you succeeded.
→ We interpreted this instruction in two ways, and provide comments on the two interpretations below:

i. If Alice provided Bob with an oracle U_f, and Bob uses a Qiskit script to simulate a quantum circuit, is it possible for Alice to ensure that Bob cannot easily determine the matrix elements of U_f, which would be sufficient for Bob to solve the problem classically?
It is our understanding that currently, it is always possible for Bob to access the internal working of U_f in Qiskit. The reason for this is that in order for Bob to implement U_f in his circuit, he needs an `Operator` (or an `isometry`) object from the unitary matrix. With access to that object, he can print it and see the matrix. From there he could reverse engineer the function f. Even if he only had access to the circuit, Qiskit made no attempt to limit a user's access of the internals of the circuit. They could access the member variables of the circuit and gain access to the gate.

ii. If Bob provides Alice (the user) with a Qiskit script which takes as input the function f, can Bob ensure that Alice will not be able to access the implementation of U_f? (This was the interpretation suggested by an answer on Piazza).
Although it is possible for Alice to view the script itself in order to figure out how U_f has been implemented, if Bob provides Alice with only a way for a high-level implementation of the quantum algorithm, Alice will simply input f, and receive the final output of the quantum circuit at the end, without interacting with Qiskit at all. We have implemented this approach in our programs, with the user having access to a high-level function, e.g. `run_DJ()`, which only needs f as the input, and provides 0 or 1 as the output.
We also note however, that the user can use `inspect.getsource` in Python to inspect the implementation of U_f in our codes.

(c) Present the design of how you parameterized the solution in n.
→ Each value of n requires a different circuit. Hence our programs create the circuits dynamically. Whenever only f is provided, the matrix U_f is created and the circuit is constructed, compiled and executed. In order to add a variable number of gates (which depend on n), we used `for` and `while` loops. In order to connect multiple wires to a single gate, we were able to use the `range` function to refer to multiple wires.

(d) Discuss the number of lines and percentage of code that your two programs share. Assess how well you succeeded in reusing code from one program to the next.
→ DJ and BV algorithms share the same circuit, and as a consequence many of the functions we used in their implementations were identical. However, Simon's algorithm and Grover's algorithm involved qualitatively different circuits, and dealt with distinct problems, resulting in less code sharing with other algorithms. Simon's algorithm required a classical solver which was unique to it. The testing generation functions were different as well. Individual pieces of the circuit shared code though, such as adding Hadamard gates in a for loop and applying U_f. Creating U_f shared some code as well but there were some differences because of the varity of n. Qiskit allowed a good portion of the circuit code to be shared.

(e) Discuss your effort to test the two programs and present results from the testing. Discuss whether different cases of U_f lead to different execution times.
→ Testing is discussed in each algorithm since it required different test functions. Different cases of U_f lead to different execution times. We observed that there was some variance in the execution time for different U_fs. However, there was a much smaller range in Qiskit then in PyQuil. It seems the execution time is more reliable. For histograms for each algorithm, please see the relevant sections above.
Since Grover's search algorithm has a possibility of failure, we implemented our code 1000 times for a variety of values of n and a, and observed a close match between the success rate of our implementation, and the success rate expected theoretically.

(f) What is your experience with scalability as n grows? Present a diagram that maps n to execution time.
→ We discussed the scalability for each algorithm and observed an exponential increase in execution time with the number of qubits. We did note that the execution time was faster than on PyQuil. We were able to run circuits up to 11 qubits before the simulator began to crash.

2. Qiskit:
As a reflection on both homeworks on programming in Qiskit, address the following points.

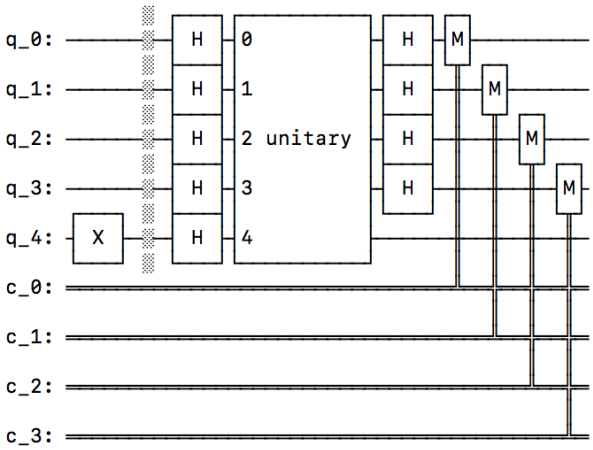


Figure 10: Example of Qiskit's visualization showing the Deutsch-Josza circuit for n = 4.

(a) List three aspects of quantum programming in Qiskit that turned out to be easy to learn and list three aspects that were difficult to learn.
→ Aspects of Qiskit which were easy to learn:
   i. Adding custom gates to a quantum circuit is very convenient in Qiskit. In our experience, this was easier to do in Qiskit, than in Pyquil. Specifically, in PyQuil one has to define a `GateDef` object, as well as its constructor. In Qiskit, it was as easy as passing the matrix into the `Operator` function to get a gate.
   ii. Qiskit handles gate objects very well, with intuitive syntax for adding in-built gates in few number of lines.
   iii. It is very convenient to simulate a quantum circuit in qiskit without having to create a server using command line. As a result, installing and getting Qiskit to work locally on a computer is simple and quick.

   → Aspects of Qiskit which were difficult to learn:
   i. It was difficult to understand how to correctly convert a unitary matrix to a quantum gate in Qiskit, because of the peculiar qubit ordering that Qiskit uses.
   ii. The qubit ordering in Qiskit is the opposite of what is used almost universally in quantum computing/-physics textbooks (including the current course). As a result the measured result is always swapped, which was not easy to understand at first.
   iii. There seemed to be some inconsistent errors. When running with a large number of qubits, the program would terminate with an exception that data could not be find. The error would not happen consistently and was likely due to memory errors. However, this makes it hard to debug, especially if the exception does not reference memory.

(b) List three aspects of quantum programming in Qiskit that the language supports well and list three aspects that Qiskit supports poorly.
→ Good support:
   i. Qiskit supports a variety of visualizations, especially printing circuits, and a wavefunction simulator. It was easy to learn how to uses these features in order to ensure that our quantum circuits are correct. In contrast, in Pyquil, the lack of circuit visualization made it hard to debug problems with the circuit construction. Figure 10 shows an example of the circuit visualization for the DJ circuit. This shows the circuit exactly how we draw and visualize it. It allows the user to pass in hints such as barriers to make columns of one step line up.
   ii. Performance was handled well. The circuits compiled and measured really quickly. This could be a huge time saver when developing more complex circuits.
   iii. The concept of the backend was handled really well. Since there were names of real machines such as "Melbourne", it was clear that this is the quantum machine the code was execute on. In PyQuil, the string passed in seemed somewhat arbitrary.

   Poor support:

i. Although installing qiskit itself is simple, running it requires having the latest version of Visual Studio C compiler installed on your computer, an issue which was cumbersome to resolve. Visual Studio takes up a lot of space, of the order of $\approx 5 - 10$ Gbs.
ii. The results of measurements in circuits were flipped. It would be helpful to have the option of using the regular bit ordering for measurements.
iii. There wasn't much support for the obfuscation or hiding of the implementation of U_f. With access to the custom gate, one can simply print it to view the matrix and would be able to recreate the function.

(c) Which feature would you like Qiskit to support to make the quantum programming easier?
i. A tool or a function which translates matrices and gates from the the usual qubit and bit ordering, to that used in Qiskit might be very useful.
ii. In order to simplify installation of qiskit without requiring a lot of space being used for Visual Studio C compiler, etc., it might be useful to reduce the requirements, in the case of space constraints, which happened to one of us.

(d) List three positives and three negatives of the documentation of Qiskit.
→ Positives:
i. The documentation is fairly detailed. It provided examples for anything we needed. It showed examples of defining gates in different ways which helped give ideas on how to program.
ii. The documentation is aesthetically pleasing and followed the standard convention of python modules documentation.
iii. The documentation provides an introduction to quantum computing. All the basics are really well explained and serve as a good review before starting any programs.
iv. The documentation felt very unified. They gave a list of possible backends and it was clear on how to run on a simulator and how to run on a remote machine.
v. There was a very rich set of examples. They walked through fairly complex circuits which taught you how to use it.

Negatives:
i. It was hard to figure out how to separate compilation from runtime. It was very easy to find the `execute` method but it wasn't clear how to compile the circuit and then rerun it without recompiling.
ii. There are often multiple ways of implementation and incorporation of certain features in qiskit. For example, one can use `iso`, as well as using `Operator` and `append` methods in order to add a custom gate to a circuit. The relevant documentation is scattered across the (vast) documentation that Qiskit has, and consequently, it is hard to understand how these methods compare with each other.
iii. We could only find a few paragraphs about the qubit ordering convention used in Qiskit. A majority of our problems arose because of insufficient documentation about this issue. Having access to an example of how to construct a custom gate, in the qubit ordering qiskit uses, would have been very helpful. The documentation would be especially helpful with arbitray n and n=2 qubit custom gate examples.

(e) In some cases, Qiskit 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 Qiskit.
→ Dictionary
i. **Backend**: The machine that the code will execute on.
ii. **Operator**: Custom defined gate.
iii. **Transpiler**: Similar to a compiler. Converting the code into something that can be used by the quantum computer.
iv. **Provider**: Gives access to the backends.
v. **Aer**: This is a package built into Qiskit that provides a high-performance simulator framework with highly optimized C++ backends for the simulator, such as the 'qasm simulator' itself.
vi. **Terra**: This is the package built into Qiskit that enables access to the transpiler and quantum circuit builder.
vii. **Ignis**: This is a package for characterization of errors, creating error-correcting codes, and dealing with noise, it comes built in to Qiskit as well.
viii. **Aqua**: This is a package for quantum algorithms that are built into qiskit for real world applications in chemistry and AI for example.
ix. **Dagger**: Conjugate transpose.

(f) How much type checking does the Qiskit implementation do at run time when a program passes data from the classical side to the quantum side and back?
→ We interpreted this question in two ways, and discuss the two below:

i. Type checking done by Qiskit itself: With Python, all type checking is done at run time. Qiskit does not appear to be adding type checking in every place. Calling `Operator` on a matrix of strings throws a type error. However, appending a matrix instead of a gate object to a circuit results in a stack track that shows Qiskit tried to call a method on the matrix. Another issue in line with type checking is non-unitary matrices. We can create a gate from a non-unitary matrix and there is no error until we try to append it to the circuit.

ii. Type checking implemented by us (the interpretation suggested by an instructor answer on Piazza): We included type checking in the functions we defined, in order to catch type errors which the user is most likely to commit. For example, in DJ and BV algorithms, we ensure that f is iterable, has an acceptable size, and includes only zeros and ones. Similarly, we check that any Qiskit objects that were passed between function calls such as operators or circuits were type checked to catch errors immediately.