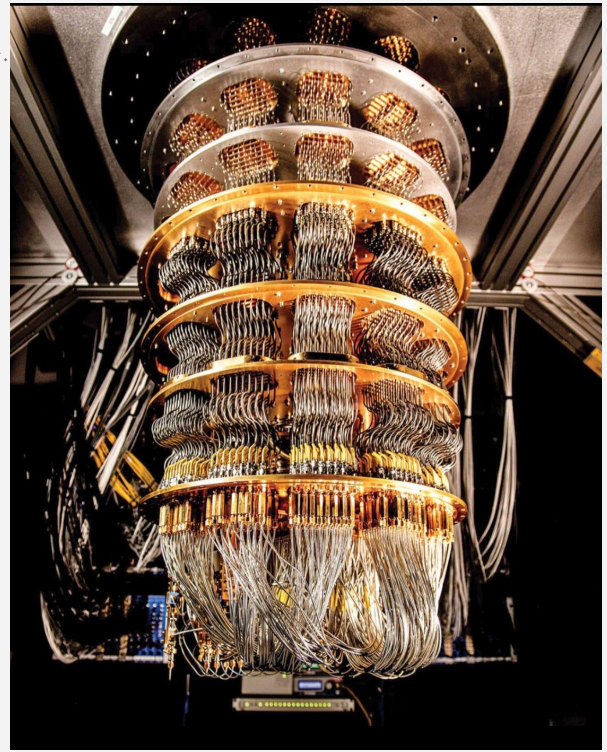
Functional Paradigms in Quantum Computing

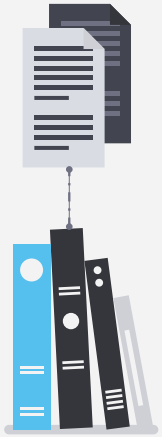


GitHub Link

Team – 5

Saanvi Vishal (IMT2021043)
Arjun Subhedar (IMT2021069)
Ritik Kumar Gupta (IMT2021098)
Pandey Shourya Prasad (IMT2021535)

Table of contents



Quantum Computing: What and Why?

- Quantum computing uses specialized technology including computer hardware that uses principles of fundamental physics(quantum mechanics) to solve complex problem that classical computers or even supercomputers can’t solve, or can’t solve quickly.

- Complex problems are problems with lots of variables interacting in complicated ways. Modeling the behavior of individual atoms in a molecule is a complex problem, because of all the different electrons interacting with one another.

- The real world runs on quantum physics. Computers that make calculations by using the quantum states of quantum bits should in many situations be our best tools for understanding it.

Why are Quantum Computers faster?

- A classical computer/supercomputer might try to simulate molecular behavior with brute force, by using its many processors to explore every possible way every part of the molecule might behave. No computer has the working memory to handle all the possible permutations of molecular behavior by using any known methods

- Quantum algorithms try a different method for solving tough problems by making multi-dimensional spaces for calculations.

- In regular computers, information is stored and processed in a linear way. But in quantum computing, information can exist in multiple states at once.

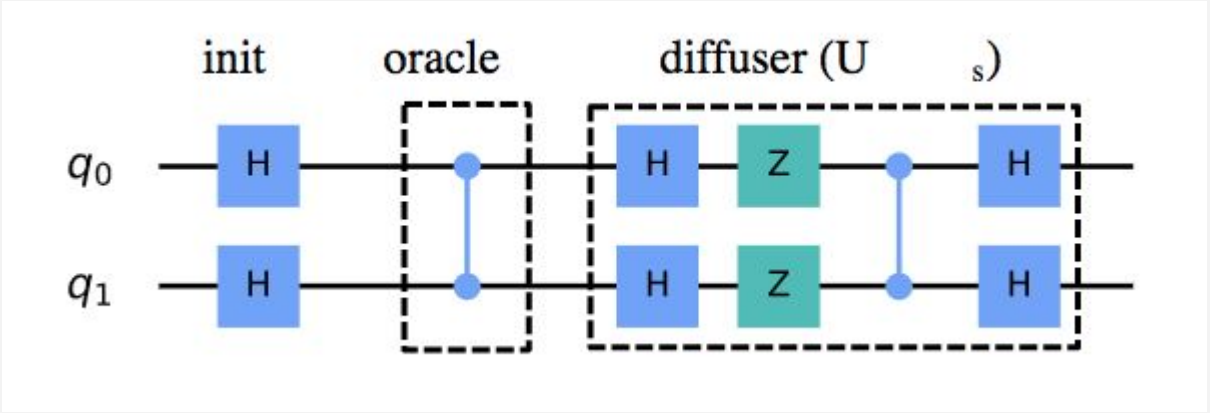
Basic Terms of QC to Know:

- **Qubit:** Quantum bit, the basic unit of quantum information. Unlike classical bits (0 or 1) qubits can exist in multiple states simultaneously due to superposition.
- **Superposition:** A fundamental principle where a qubit can be in a combination of states simultaneously, rather than being restricted to just one state like classical bits.
- **Entanglement:** When qubits become correlated with each other in such a way that the state of one qubit depends on the state of another, even when they are physically separated.
- **Quantum Gate:** Analogous to classical logic gates, quantum gates manipulate qubits to perform operations necessary for computation. Examples include the Hadamard gate and the CNOT gate.
- **Measurement:** The process of extracting classical information from a quantum system, which causes the collapse of the superposition into a definite state.

Grover’s Algorithm

Grover’s Algorithm (1)

- Grover’s algorithm is a quantum algorithm that solves the unstructured search problem. In an unstructured search problem, given a set of N elements and want to find a single marked element. A classical computer would need to search through all N elements in order to find the marked element, which would take time O(N). Grover’s algorithm, on the other hand, can find the marked element in time O(√ N).



Grover’s Algorithm Circuit

Grover's Algorithm (2)

- Let's denote the element we want to find as $|x_0\rangle$. Functions implemented as an oracle (Rf) and a diffusion operator (RD) respectively. Rf applies a negative sign before $|x_0\rangle$ and leaves all the other states as it is and RD applies a negative sign to all states orthogonal to the state $|00\ldots0\rangle$.

- In Grover's algorithm we start with a state which is the superposition of all the n-qubit states, let's call this state $|D\rangle$, we then apply Rf followed by RD.

$$|D\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle$$

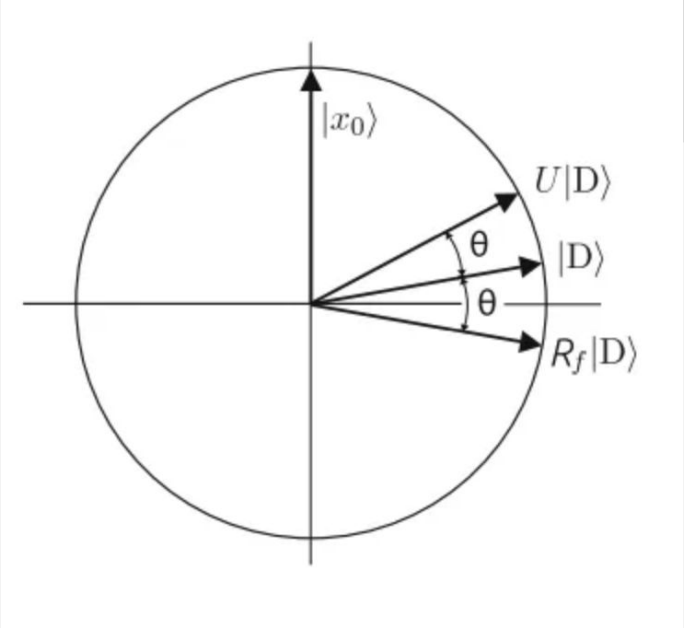
$$R_f = I - 2|x_0\rangle\langle x_0|$$

$$R_D = 2|D\rangle\langle D| - I$$

Grover's Algorithm (3)

- Let's assume a 2-D subspace where the y-axis is the vector $|x_0\rangle$ and the x-axis denotes all the vector orthogonal to $|x_0\rangle$. We have $|D\rangle$ making an angle of $\theta/2$ with the x-axis.

- We first apply the Rf operator which reflects the state $|D\rangle$ about the x-axis and gives the state Rf $|D\rangle$ then we apply the RD operator which reflects the state about $|D\rangle$.



Grover's Algorithm Analysis (1)

- |xb> is the vector orthogonal to $|x_0\rangle$, i.e. x axis represented in the 2D subspace

$$|D\rangle = sin(\theta)|x_0\rangle + cos(\theta)|x_b\rangle$$

- the conjugate of |D> can be written as follows ->

$$|D_0\rangle = cos(\theta)|x_0\rangle - sin(\theta)|x_b\rangle$$

$$|x_0\rangle = sin(\theta)|D\rangle + cos(\theta)|D_0\rangle$$

$$|x_b\rangle = cos(\theta)|D\rangle + sin(\theta)|D_0\rangle$$

- The search algorithm starts with the |D> state, then applying Rf will give the state ->

$$R_f|D\rangle = -sin(\theta)|x_0\rangle + cos(\theta)|x_b\rangle = cos(2\theta)|D\rangle - sin(2\theta)|D_0\rangle$$

Grover's Algorithm Analysis (2)

- Then after applying operator RD we get the state ->

$$R_D R_f|D\rangle = cos(2\theta)|D\rangle + sin(2\theta)|D_0\rangle = sin(3\theta)|x_0\rangle + cos(3\theta)|x_b\rangle$$

- It is easy to verify by induction that after k iterations of the Grover iterator (U = RD.Rf) starting with state |D> = H|00…..0> we get ->

$$R_D R_f k|D\rangle = cos(2k\theta)|D\rangle + sin(2k\theta)|D_0\rangle = sin((2k+1)\theta)|x_0\rangle + cos((2k+1)\theta)|x_b\rangle$$

- In order to obtain $|x_0\rangle$ with high probability we must select k such that $\sin((2k + 1)\theta) \approx 1$, which means that we would like $(2k + 1)\theta \approx \pi/2$

$$k \approx \frac{\pi}{4}\theta - \frac{1}{2} \approx \frac{\pi}{4}\sqrt{N}$$

- We get the desired result after only k iterations and k is of the order of √N hence the time complexity of Grover's algorithm is O(√N)!

Shor’s Algorithm

- Shor's algorithm is a quantum algorithm for finding the prime factors of an integer.
- When the integers are very large, no efficient classical integer factorization algorithm is known. The hardest factorization problems are semiprime numbers, the product of two prime numbers.
- The presumed difficulty of this semiprime factorization problem underlines many encryption algorithms, such as RSA, which is used in online credit card transactions, amongst other applications.

Role of Prime Factorisation in RSA

- To use the RSA algorithm to encrypt a message, the sender first converts the message into a numerical form using a predetermined encoding scheme. The sender then raises this numerical value to the power of the public key and takes the result modulo the public key to produce the encrypted message.
- The recipient can then use their knowledge of the prime factors of the public key to decrypt the message by raising the encrypted message to the power of a certain exponent and taking the result modulo the private key, which is derived from the prime factors of the public key.
- Now the public key is known to everyone but its prime factors are not known. And since the private key is basically a function of the prime factors of the public key, the problem to decrypt the message becomes very difficult.

The Algorithm

- Reduction of factorization of N to the problem of finding the period of an integer x less than N and greater than 1 depends on the following result from number theory: Theorem: The function F(a) = x^a mod N is a periodic function, where x is an integer coprime to N and a ≥ 0.
- Since F(a) is a periodic function, it has some period r. Knowing that x^0 mod N = 1, this means that x^r mod N = 1 since the function is periodic, and thus r is just the first nonzero power where x^r = 1(modN).
- Given this information and through the following algebraic manipulation.
- From this, the product is an integer multiple of N. Thus, as long as both the multiples are not multiples of N, then at least one of them has a non-trivial factor with N, which gives us the answer.
- The Quantum part of this algorithm comes from the period calculation, which is done by Quantum Fourier Transform.

$$x^r \equiv 1 \bmod N$$

$$x^r = (x^{r/2})^2 \equiv 1 \bmod N$$

$$(x^{r/2})^2 - 1 \equiv 0 \bmod N$$

$$(x^{r/2} + 1)(x^{r/2} - 1) \equiv 0 \bmod N$$

Implementation

```ocaml
open Random

(* Custom random function *)
let generate_random_number n =
  Random.self_init ();
  1 + Random.int (n/2)

(* GCD finder function *)
let rec gcd_finder a b =
  print_string (string_of_int a ^ "a " ^ ", b " ^string_of_int b ^ "\n");
  if b = 0 then (
    print_int a;  (* Print a when returning *)
    print_newline ();
    a
  ) else
    gcd_finder b (a mod b)


(* Brute force period finding algorithm *)
let find_period_classical x n1 =
  let rec find_period_helper n t =
    if t = 1 then
      n
    else
      let t' = (t * x) mod n1 in
      find_period_helper (n + 1) t'
  in
  find_period_helper 1 x

let rec find_even_period x n1 =
  let r = find_period_classical x n1 in (* Step three *)
  (* print_string (string_of_int r); *)
  if r mod 2 = 0 then
    r
  else
    find_even_period x n1

let shors_algorithm_classical n1 =
  let x = generate_random_number (n1 + 1) in
  (* let x = 206 in *)
  print_string (string_of_int x ^ " hiii \n");
  if gcd_finder x n1 <> 1 then (* Step two *)
    x, 0, gcd_finder x n1, n1 / gcd_finder x n1
  else

    let r = find_even_period x n1 in
    (* let r = find_even_period () in *)
    (* let p = gcd_finder (x *(2 ** ((r / 2))) + 1) n1 in Step four
    let q = gcd_finder (x *(2 ** ((r / 2))) - 1) n1 in *)
    let p = gcd_finder ((x * (1 lsl (r / 2)+1))) n1 in (* Step four *)
let q = gcd_finder ((x * (1 lsl (r / 2)-1))) n1 in

    (* print_string (string_of_int ((x * (1 lsl (r / 2)-1))) ^ " vvvb  \n"); *)
    print_string (string_of_int p ^ "\n");
    print_string (string_of_int q ^ "\n");
    print_string (string_of_int r ^ "r \n");
    x, r, p, q

let () =
  let n1 = 16 in
  let x, r, p, q = shors_algorithm_classical n1 in
  Printf.printf "semiprime n1 = %d, coprime x = %d, period r = %d, prime factors = %d and %d\n" n1 x r p q
```

Classical Implementation in Ocaml

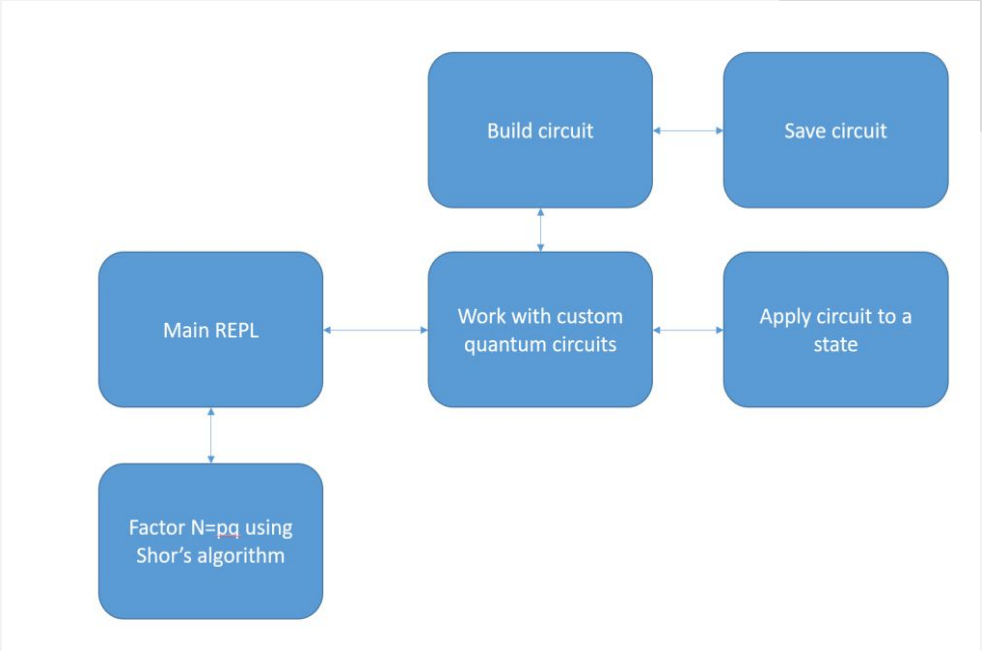
```python
def circuit_aperiod15(qc,qr,cr,a):
    if a == 11:
        circuit_11period15(qc,qr,cr)
        return

    # Initialize q[0] to |1>
    qc.x(qr[0])

    # Apply a**4 mod 15
    qc.h(qr[4])
    #   controlled identity on the remaining 4 qubits, which is equivalent to doing nothing
    qc.h(qr[4])
    #   measure
    qc.measure(qr[4],cr[0])
    #   reinitialise q[4] to |0>
    qc.reset(qr[4])

    # Apply a**2 mod 15
    qc.h(qr[4])
    #   controlled unitary
    qc.cx(qr[4],qr[2])
    qc.cx(qr[4],qr[0])
    #   feed forward
    qc.u(math.pi/2.,0, 0, qr[4]).c_if(cr, 1)
    qc.h(qr[4])
    #   measure
    qc.measure(qr[4],cr[1])
    #   reinitialise q[4] to |0>
    qc.reset(qr[4])

    # Apply a mod 15
    qc.h(qr[4])
    #   controlled unitary.
    circuit_amod15(qc,qr,cr,a)
    #   feed forward
    qc.u(3.*math.pi/4.,0 ,0 ,qr[4]).c_if(cr, 3)
    qc.u(math.pi/2.,0,0,qr[4]).c_if(cr, 2)
    qc.u(math.pi/4.,0,0,qr[4]).c_if(cr, 1)
    qc.h(qr[4])
    #   measure
    qc.measure(qr[4],cr[2])
```

Qiskit Implementation

Functional Quantum Algorithm Framework for Shor’s

Architecture

- At the beginning, users can choose between two options: use shor's algorithm for factoring or create his own circuit through a REPL.
- If the user selects a pre-built circuit, it will processes the input and displays the result.
- For custom circuit, users constructs and save the circuit, input the state and then run the algorithm.
- The user can also return to the REPL for further action.



System and Module Design

Our architecture comprises five modules, each serving a distinct purpose and interdependent functionality.

1. **Arithmetic Module**
2. **State Module**
3. **Gate Module**
4. **Shor Module**
5. **Main Module**

1. Arithmetic Module

Consists of required arithmetic and number theoretic functions that are required to implement specific gates and parts of Shor's factoring algorithm.

- **Integer exponentiation**:
  - Computes the exponentiation of the integer.
  - *int_exp : int -> int -> int*
- **Integer Exponentiation with Modulus:**
  - Computes the exponentiation of an integer with modulus.
  - *int_exp_mod : int -> int -> int -> int*
- **Bitwise XOR Operation**:
  - Performs a bitwise XOR operation on two digit lists.
  - *add_mod_2 : dlist -> dlist -> dlist*
- **Splitting Digit List**:
  - Splits a digit list into two parts, with the length of the first part specified.
  - *split : int -> dlist -> dlist * dlist*
- **Base 10 Representation**:
  - Converts a number from base b to its base 10 representation.
  - *base_10_rep : int -> dlist -> int*

```
type dlist = int list

(* [int_exp a b] produces integer exponentiation a^b *)
val int_exp : int -> int -> int

(* [int_exp_mod a b m] produces integer exponentiation a^b mod m. *)
val int_exp_mod : int -> int -> int -> int

(* [digit_list n] produces a list of digits of [n].
 * Example: [digit_list 123] evaluates to [1;2;3] *)
val digit_list : int -> dlist

(* [base_10_rep b dlst] converts a number in base [b] represented by
 * the digit list [dlst] to its base 10 representation *)
val base_10_rep : int -> dlist -> int

(* [base_convert bold bnew dlst] converts a number in base [bold]
 * represented by the digit list [dlst] to its base [bnew] repsentation
 * as a dlst *)
val base_convert : int -> int -> dlist -> dlist

(* [dlst_to_string dlst] converts a digit list to a string.
 * Example: [dlst_to_string [1;2;3]] evaluates to "123" *)
val dlst_to_string : dlist -> string

(* [pad_zeros n dlst] adds zeros to the head of [dlst] until
 * it is of length [n] *)
val pad_zeros : int -> dlist -> dlist
```

2. State Module (1)

Consists of required types and functions for a quantum state. This will implement qubits in the quantum computer to be processed by other functions in other modules.

- **State Type**:
  - Defines the overall type of a state for a certain number of qubits, represented as a tree structure.
  - *type st = Tree of st * st | Node of Complex.t*
- **Creating States:**
  - Generates a state of n qubits from a list of complex numbers, with each coefficient representing the state's amplitude.
  - *make : Complex.t list -> st*
- **Accessing Specific Qubits**:
  - Retrieves the nth qubit from a state, with m number of bits.
  - *state_of : int -> int -> st*
- **Empty State**:
  - Creates an n-bit qubit with all coefficients set to 0.
  - *empty : int -> st*
- **Number of Bits**:
  - Returns the number of bits in a state.
  - *n_bits : st -> int*

2. State Module (2)

- **Tensor Product**:
  - Computes the tensor product of multiple states, creating a composite state.
  - *tensor : st list -> st*
- **Normalization:**
  - Normalizes a state based on the coefficients of its components.
  - *normalize : st -> st*
- **Accessing Specific Qubits**:
  - Retrieves the nth qubit from a state, with m number of bits.
  - *state_of : int -> int -> st*
- **State Operations**:
  - Adds two states and returns the result, not normalized.
  - *add : st -> st -> st*
  - Adds an overall phase to all elements in a state.
  - *phase : float -> st -> st*
- **State Properties**:
  - Returns the coefficients of a state as a list of complex numbers.
  - *coeff : st -> Complex.t list*
  - Returns a formatted string of a state's coefficients.
  - *val_string : st -> string*

```
open Complex

(* the overall type of the state of a certain number of qubits *)
type st = Tree of st*st | Node of Complex.t

(* [make lst n] takes as input a list of complex numbers and an
 * integer n to produce a state of n qubits where elements of the list
 * serve as coefficients of states for their respective indices in the
 * list. For example, the list [1+i; 0; 5; 3i] will become
 * the state (1/6)((1+i)|00> + 0|01> + 5|10> + 3i|11>). 1/6 is the
 * normalization factor
 * raises: Failure if the number of elements in [lst] is not 2^[n] *)
val make : Complex.t list -> st

(* [state_of n m] returns the [n]'th qubit, with [m] number of bits.
 * For example, [state_of 5 3] will return [0,0,0,0,0,1,0,0].
 * If n > 2^m, the top bits will be truncated. For example,
 * [state_of 22 3] will return [0,0,0,0,0,0,1,0]. *)
val state_of : int -> int -> st

(* [empty n] returns an n bit qbit with all coefficients as 0 *)
val empty : int -> st

(* [n_bits s] returns the number of bits in a state. *)
val n_bits : st -> int

(* [zero] returns a 1 bit zero qbit *)
val zero : st

(* [one] returns a 1 bit one qbit *)
val one : st

(* [tensor sl] Returns the tensor product of all the states in sl. The first
 * elements of the list will become the significant bits in the new state. *)
val tensor : st list -> st
```

3. Gate Module

- **Gate Type**:
  - Defines the overall type of an n-qubit gate.
  - *type t*
- **Creating Gates:**
  - Constructs a gate based on an input string that specifies the gate type and the qubits it acts upon.
  - *gate_of : string -> t*
- **Controlled Gates**:
  - Generates a gate controlled by specified qubits, ensuring all control qubits are active for the gate to be effective.
  - *control : int list -> t -> t*
- **Applying Gates**:
  - Applies a gate to a given state, producing the resultant state.
  - *apply : t -> st -> st*
- **Composite Gates**:
  - Computes the tensor product of multiple gates, defining a composite gate where gates are applied sequentially.
  - *product : t list -> t*

```
val gate_of : string -> t

(* [control l s] returns a gate that is controlled by the numbers in l. Every
 * value of the bits in l must be on in order for the gate [s] to be active.
 * precondition: the numbers in [l] must not be any of the bits that are being
 * operated on in the [s] gate *)
val control : int list -> t -> t

(* [apply s g] takes in inputs state [s] and gate [g] and outputs the
 * resultant state of applying [g] to [s].
 * Post-condition: resulting state is normalized.
 *)
(* val apply : t -> st -> st *)

(* [product l] takes in a list of gates and outputs their tensor product.
 * In the producted gate, the first element gate is applied first,
 * and then the second, and so forth. *)
val product : t list -> t

(* [gate_to_string t] returns a string that allows the user to store a gate in
 * string form. *)
val gate_to_string : t -> string

(* [string_to_gate s] returns the gate of string [s] that is returned after
 * calling gate_to_string on a gate *)
val string_to_gate : string -> t
```

4. Shor Module

- **Factoring Function**:
  - Factors an integer using Shor's algorithm and efficient classical algorithms, printing statistics comparing their efficiency.
  - *factor : int -> unit*
- **Quantum Fourier Transform Gate:**
  - Generates a quantum Fourier transform (QFT) circuit as a single gate, specifying the number of qubits involved in the QFT out of the total number of qubits.
  - *qft_gate : int -> int -> Gate.t*

```
open Gate

  (* [factor i] takes in an int [i] and factors and prints its prime factors.
   * This function simulates factoring using shor's algorithm as well as
   * efficient classical algorithms and prints statistics comparing efficiency
   * of the 2 techniques
   *
   * Pre: i is a product of primes p q and i > 0 *)
  val factor : int -> unit


 (* [qft_gate in total] returns a quantum fourier transform circuit as a
  * single gate with [in] number of qubits involved in the QFT out of
  * [total] number of qubits. The indices of the qubits involved in the QFT
  * is counted in decreasing order.
  * Example: [qft_gate 2 4] returns a QFT gate that performs the QFT on
  * qubit 2 and 3 (zero-indexed) and leaves qubit 0 and 1 alone. *)
 val qft_gate : int -> int -> Gate.t
```

5. Main Module

- **REPL (Read-Eval-Print Loop)**:
  - The REPL provides users with two primary choices:
    - Building quantum circuits and applying states to them
    - Factoring integers using Shor's algorithm
- **Building Quantum Circuits**
  - Users can build quantum circuits by listing out gates or sequences of gates separated by spaces.
  - They can save the circuits they've built and view currently available circuits.
  - The format for entering gates is provided, guiding users on how to construct quantum circuits.
- **Applying Gates to States**
  - Users can apply saved circuits to states, observing the resulting state after passing through the circuit.
  - They can view available circuits, apply gates, or quit this mode.
- **Integration with Shor's Algorithm**
  - Users can factor natural numbers using Shor's algorithm, with the program providing statistics comparing its efficiency with classical algorithms.
  - This functionality relies on the Shor module.

Comparing Functional & Imperative Implementations in QC

Advantages of Functional Paradigms in QC

- **Immutable State:** Functional programming emphasis on immutability is a big plus for quantum programming, ensuring that states remain stable and reducing the risk of unintended changes.

- **Higher-Order Functions:** Using higher-order functions enables concise and expressive code for quantum operations. Functions can be passed as arguments, allowing for dynamic gate creation and manipulation, thus enhancing flexibility in quantum algorithm design.

- **Concurrency and Parallelism:** Functional programming encourages a stateless, side-effect-free approach, making it inherently well-suited for concurrent and parallel execution. In quantum computing, where operations often occur simultaneously or in parallel, functional paradigms facilitate efficient resource utilization and scalability.

Challenges of Functional Paradigms in QC

- **Performance Overhead:** Functional programming tools like higher-order functions and immutable data structures might slow down performance, especially in heavy quantum simulations or computations, compared to imperative methods.

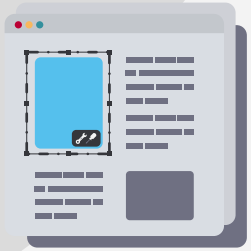
- **Learning Curve:** Quantum programming itself presents a steep learning curve, and adopting functional paradigms adds another layer of complexity for developers.

- **Debugging and Testing:** Functional programming encourages the use of pure functions, which lack side effects and rely solely on input parameters. While this enhances predictability and testability, it may complicate debugging processes, especially in complex quantum algorithms.

Challenges Faced:

- **Implementation Limitation:** Significant portion of our time was given to learning the foundational concepts of quantum computing theory, such as quantum states, gates etc. Understanding these theoretical aspects consumed the majority of our allocated time, leaving limited resources for actual implementation.

- **Internal Operation:** We're familiar with the input and output of methods in Qiskit, IBM's quantum computing library, but lack insight into its internal workings. This knowledge gap poses a challenge as efficient internal processes are crucial for handling different quantum computations like parallelism.



Thanks!

Do you have any questions?