Qsort

The Programming Language

Version 0.0.1

Qsort is a theoretical, high-level programming language with the assumption that quantum computers work without flaw. Whereas in reality, and as of the year of this paper: 2023, quantum computers are in the early stages of development. The name Qsort is a portmanteau of quantum and sort. It is also a pun related to the classical quick sort algorithm. The language's name is derived from the first letter of my daughter's name, Quinn (featured below). She is the source of my inspiration for this programming language idea.

1. Introduction

The Qsort programming language is a fictional, high-level programming language used to communicate with the operating system of a quantum computer. At the time of writing this paper, quantum computing hardware is still in its infancy in terms of stable output – meaning the results from the machine are not always what researchers expect. This paper aims to create a high-level, language design concept with the working assumption that quantum computers are perfect and therefore, programmers can leverage this technology to create new software.

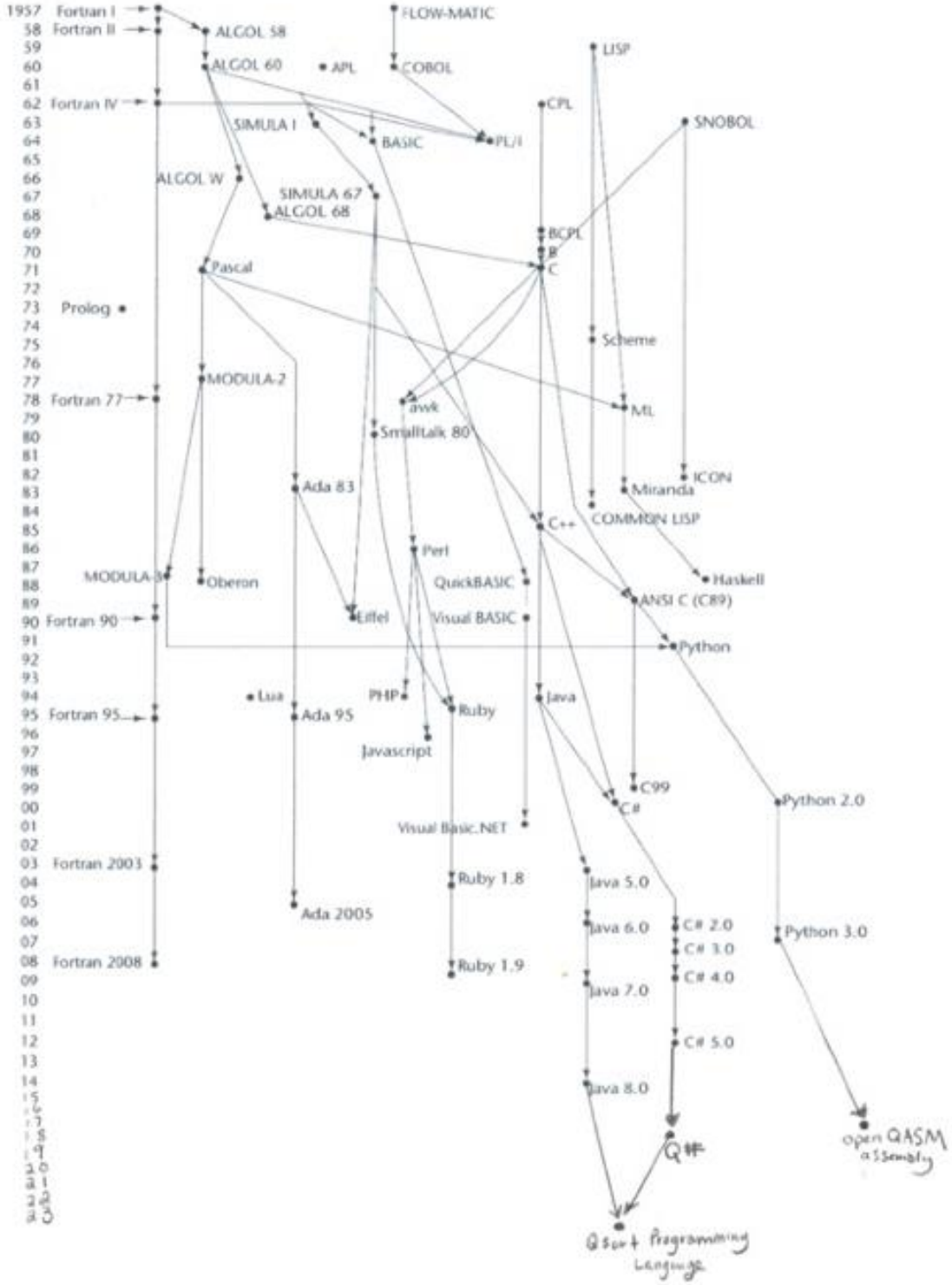
Quantum computers use a qubit which represents either state zero or one, or any linear combination between these states (e.g. a superposition of different states). One qubit can take the value of two classical binary bits. N qubits is the equivalent of 2^N classical bits. This technology has the potential to create more efficient looping data structures in software.

The language described throughout this paper is used as an abstraction, which is then used to communicate with the cQASM (Common Quantum Assembly Language) assembly language. Qsort is a procedural language that aims to simplify the task of programming applications that use object-oriented programming concepts. The Qsort language is a virtualized environment that can be deployed on many types of machines which makes use of the IoT cloud for binary machines, but compiles directly on quantum hardware. The language consists of elements of the programming languages Java and Sliq, but differs in the following ways:

1. Qsort leverages the hardware to create more efficient looping processes; arrays are considered a primitive type.
2. Qsort supports the ease of use for linear algebra operations.
3. Qsort's type conversion use the arrow syntax. Ie. "hello" -> [] = ["h", "e", "l","l","o"]
4. Qsort supports functional programming which directly supports matrix arrays.
5. Qsort supports emoji operators and variables.

1.1. Genealogy

Q: Where does your language fit into the programming language genealogy? Add your language to this diagram to highlight your language and its ancestry.



1.2. Hello World

```
import QuantumCircuits, BinaryCPU

Main( ){
   `prints the phrase "Hello world"
    print("Hello World");

}
```

1.3. Program Structure

The key organizational concepts in the Qsort programming language are as follows:

1. The main function acts as an entry point for execution and can be supported by external objects. This is similar to Java's "public static void main(String[] args)," however the main declaration is simplified to "main()."
2. Functions, procedures and classes are wrapped within curly brackets to show the beginning and end points. Object wrapping is permitted.
3. Supporting libraries can be linked using the "import" keyword.
4. Matrices and arrays are zero indexed.
5. When referring to positions of matrices, the syntax [][] is used. E.g. The top left element of a 2x2 matrix is referred to as [0][0].
6. To overwrite the value of a position, the following syntax is used: 100 @ [0][0], where 100 becomes the first element in the matrix.
7. The syntax {[][]} declares the size of an empty matrix. E.g. {[2][3]} creates a 2x3 matrix where all values are initialized to null.
8. There are built in standard matrices that can be called and assigned to matrix variables using the "as" keyword. E.g. integerMatrix{[4][4]} as vector^^1. This creates a 4x4 matrix in which every element is initialized to 1.
9. Arrays of unknown sizes can be declared as {[?]} and arrays of unknown dimensions can be initialized as {[?][?]}.

Example

```
import QuantumCircuits, BinaryCPU, Queue
Public Main( ){
    initialize a 2x2 matrix queue
    Public integerMatrix = new Queue <$int{[2][2]}> ;
    initialize every element in the matrix to have the value of one.
    integerMatrix = integerMatrix as vector^^1;  `assign unit vector to integerMatrix
    adds integer 5 to the third column (1st and 2nd row simultaneously)
    integerMatrix.enqueue($int 5 @ [1][3] & [2][3]);
    removes first elements in the first column of the array
    integerMatrix.dequeue();
    adds elements to the third column of the array
    integerMatrix.enqueue($string "A" @ [0][0]);
    returns ([A,5], [1,5])
    print(integerMatrix);
}
Protected Class Queue<T> {
    creates a matrix that allows any type to be passed in matrix of unknown size and dimension
    private items: T[][] = {[?][?]};

    enqueue(item: T) returns void{
        this.push(item);
    }
    dequeue() returns (T or undefined) {
        return this.shift();
    }
    size() returns Integer {
        return this.length;
    }
    peek(){
       return this.items{[0][0], [1][0], … [max][0]};
    }
}
```

This example declares a namespace that contains a main function that calls to an external class within the same file. The class named 'queue' is instantiated in the main namespace where it is procedurally modified. The queue class can accept any general data type passed to a matrix of unknown size and dimension. The class supports one private field and four member methods. A new instance of the class is initialized in the main() function.

1.4. Types and Variables

Classical variables and **quantum variables** in the Qsort programming language must begin with a lower-case letter and there can be no spaces in the variable name. Variable names can be composed of 32,768 alpha numeric characters or emojis (with the exception of 👍, 👍, 📝, and 😘). Primitive classic types start with the $ symbol, whereas quantum types do not.

There are two ways to classify variables that are used to reference memory: ***value types*** and ***reference types***. Variables that are of a value type contain information about how the data is stored within the same memory address. Whereas, variables of reference types are used to create objects by storing references to its associated data. Reference types make it possible for two variables to reference the same object. Reference types can be used to manipulate the referenced variables or alter copies of itself. The Qsort language implements static scoping to bind variables to types, which aims to reduce the number of errors at runtime. See Section 3 for details.

1.5. Visibility

Qsort uses the keywords *Public*, *Private*, and *Protected* as an access modifier for attributes, methods and constructors.

1. The ***Public*** keyword makes its associated fields, methods or classes accessible to any other class within the same file or any external file that is linked using the import keyword. The Public keyword is used to create classes, methods and variables that are global in scope.

2. The ***Private*** keyword restricts its associated fields, methods or classes to itself. The attributes are only accessible within the scope of its own class.

3. The ***Protected*** keyword restricts its associated fields, methods or classes to libraries that import the class.

1.6. Statements Differing from Qsort and Java

Statement	Example
Expression statement	`Public Main(){` ` `` declare a classic variable i of type int` `  $int i = 123;` `  `` declare a classic variable j of type char` ` $char j = 'a' ;` ` array[$int | $char] k = [?];` ` ``` declare a quantum qubit vector variable and assign to the variable m''` ` qubit m = {Sqrt(1/2) |0> + #imaginary#/Sqrt(4) |1>};`  `}`
Arrays	`Public Main(){` `  $char a = 'a';` `  $string b = 'b';` `  $int c = 123;` `  c -> $char;` ` `` declare a classical array of unknown size that accepts classical integers or classical string variables''` `  $array[$string] c = {[?]};` `  a @ d{[0]};` `  b @ d{[1]};` `  c @ d{[2]};` `  print(d); `` prints['a','b','c']` ` print(c{[0]}); `` prints['a']`  `  qubit e = {[|1>]};` `  qubit f = {[|0>]};` `  array[qubit] g = {[?][?]};` `  d @ g{[0]};` `  e @ g{[1]};` `  print(g); `` returns 2 dimension array of vectors` `}`
Type casting	`Public Main(){` `  $int i = 123;` `  $char j = 'a' ;` `  array[$string] k = {[?]};` `  i -> $string;` `  i @ k{[0]};` `  j @ k{[1]};` ` print(k); `` prints['123','a']` `}`
Access each element in an array simultaneously	`Pubic Main(){` ` `` declare a quantum matrix of size 2x2` `  array[qubit] g = {[2][2]};` `  `` assigns unit vector to array variable g` ` g = g as vector^^1;`  ` print(g); `` [1,1][1,1]` `}`

Alternative if statement	Main(){
	$boolean b = FALSE ;
	🤔(b == 👍){
	print("true");
	}else if(b == 👎){
	print("false");
	}
	}

2. Lexical Structure

2.1. Programs

A Qsort ***program*** consists of one or more ***source files***. A source file is an ordered sequence of characters. Logical flow of execution is specified in the main function. The scope of supporting classes, methods, conditionals and loops are denoted by the curly bracket separators. A function that calls to another external object is stack-dynamic; and once the external object is resolved through a dynamic link, the execution is passed back to the original function. Subprogram parameters can only be passed by value; and object parameters are passed by reference through a pointer to a memory address.

General overview of the compilation process:

1. The first step is to translate the source code into an intermediate code known as quantum-code.®
2. The second process of type checking involves two main phases: lexical analysis and syntactic analysis. In lexical analysis, a stream of Unicode input characters is translated into a series of tokens. These tokens are then further translated in syntactic analysis to produce executable code. As a part of this process, the compiler will check for types assigned to variables.
3. External libraries and modules are linked to source code through a dynamic stack and called in logical order.

2.2. Grammars

This specification presents the syntax of the Qsort programming language and where it differs from Java and Q#.

2.2.1. Lexical grammar (tokens) where Qsort differs from Java and Q#

Regular Expressions `includes Java arithmetic, comparison, logical and assignment operators however,`
`the following have been added for use in a different capacity:`
`{,},[,],|,&,?,@,…,.`

1. Array operators:
`Basic array operators:[?],[?]|[?],[?]|[?]|[?],[?]&[?],[?]&[?]&[?],{[][]}`

2. Array mathematics operators:
`amendments to Array mathematics operators:` *-> T* (transpose), {T} (tensor product), R* (reverse), I* (inner product), {I} identity matrix, E* (eigenvalue), {E} (eigenvectors)*

3. Emoji operators:
`Basic emoji operators:` 👍,👎, 📝,🤔

4. Literal operators:
`amendments to literal operators:` *as, to*

5. Typecast operators:
`amendments to typecast operators:` *->*

2.2.2. Syntactic (parse") grammar where Qsort differs from Java and Q#

1. <*array*> ⟶ <*vector*> | <*matrix*> | <*array*>
2. <*matrix*> ⟶ <*vector*><*vector*>
3. <*vector*> ⟶ <*type*><*dimension*>
4. <*type*> ⟶ <*qubit*> | <*qLong*> | <*qFloat*> | <*qDouble*>
5. <*dimension*> ⟶ < *qubitRange* >| <*qLongRange*> | <*qFloatRange*> | <*qDoubleRange*>
6. < *qubitRange* > ⟶ <(n)|0> + (1-n|1> >
7. <*qLongRange*> ⟶< -2^{64} to $((2^{64}) - 1)$>
8. <*qFloatRange*> ⟶< -2^{128} to $((2^{128}) - 1)$>
9. <*qDoubleRange*> ⟶< -2^{128} to $((2^{128}) - 1)$>
10. <*n*> ⟶ <*(0 => n <= 1)*>

2.3. Lexical Analysis

2.3.1. Comments

There are a few supported forms to comment codes. First, the note emoji "📝" and the backtick typographical mark (`) can be used as a ***Single-line comment*** which the complier will not execute any code that resides on the same source line. ***Delimited comments*** start and end the characters `` and can span across several lines of code.

2.4. Tokens

There are several kinds of tokens which will be mentioned in this section. White space and comments are not tokens, although they act as separators for tokens where needed.

tokens:
- classical identifiers
- quantum identifiers
- keyword
- integer-literal
- real-literal
- character-literal
- string-literal
- operator-or-punctuator
- emojis

This following is a list of valid keywords:

😚	*$float*	*protected*
@	*$int*	*public*
->	*$long*	*qDouble*
...	*$short*	*qFloat*
array[]	*else*	*qLong*
array[][]	*export*	*qubit*
array[][][]	*for*	*returns*
as	*foreach*	*scalar*
break	*if*	*singleton*
class	*import*	*string*
$boolean	*map*	*this*
$byte	*new*	*void*
$char	*null*	*while*
$double	*private*	*vector^^n*

2.4.1. Keywords different from Java or Q#

A ***keyword*** is an identifier-like sequence of characters that is reserved and cannot be used as an identifier except when prefaced by the \ escape sequence.

New keywords:

```
@       vector^^n   qubit       qLong             qDouble
->      qFloat      scalar      array[][]      array[][][]
returns
```

Removed keywords:

```
internal        Stringbuilder
```

3. Type System

As stated earlier, the Qsort programming language implements an early binding compile-time, type checking system. The language implements a **strong static** type system that aims to catch errors during the compilation process rather than at the runtime. There is a strong emphasis on type conversions. Primitive types can easily be converted to a larger type. Narrowing primitive and reference types can lead to a loss of precision. The complier will warn the user if a conversion is unsafe, such as an array to a vector.

3.1. Type Rules

The type rules for Qsort are as follows:

Boolean Literals:

$$\frac{}{\vdash \textit{true}: \text{boolean}} \qquad \frac{}{\vdash \textit{false}: \text{boolean}}$$

$$\frac{}{\vdash 👍: \text{boolean}} \qquad \frac{}{\vdash 👎: \text{boolean}}$$

$$\frac{}{\vdash |0\rangle : \text{boolean}} \qquad \frac{}{\vdash |1\rangle : \text{boolean}}$$

Qubit Literal:

$$\frac{\vdash e1 : \text{qubit state} \quad \vdash e2 : \text{qubit state}}{\vdash e1 + e2 : \text{qubit state}} \qquad \frac{\vdash e1 : \text{qubit state} \quad \vdash e2 : \text{qubit state}}{\vdash e1 - e2 : \text{qubit state}}$$

$$\frac{\vdash e1 : \text{qubit state} \quad \vdash e2 : \text{qubit state}}{\vdash e1 * e2 : \text{qubit state}} \qquad \frac{\vdash e1 : \text{qubit state} \quad \vdash e2 : \text{qubit state}}{\vdash e1 / e2 : \text{qubit state}}$$

$$\frac{\vdash e1 : \text{qubit state} \quad \vdash e2 : \text{qubit state}}{\vdash e1 \wedge e2 : \text{qubit state}} \qquad \frac{\vdash e1 : \text{qubit state} \quad \vdash e2 : \text{qubit state}}{\vdash e1 \surd e2 : \text{qubit state}}$$

Literals Addition:

$$\frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 + e2 : T}$$

Assignment:

$$\frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 = e2 : T}$$

Comparison:

$$\frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 == e2 : \text{boolean}} \qquad \frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 != e2 : \text{boolean}}$$

$$\frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 > e2 : \text{boolean}} \qquad \frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 < e2 : \text{boolean}}$$

$$\frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 >= e2 : \text{boolean}} \qquad \frac{\vdash e1 : T \quad \vdash e2 : T \quad T \text{ is a primitive type}}{\vdash e1 >= e2 : \text{boolean}}$$

Conditional Rule:

$$\frac{\{P \wedge c\} S1 \{Q\}, \{P \wedge \neg c\} S2 \{Q\}}{\{P\} \text{ IF } (c) \{S1\} \text{ELSE} \{S2\} \{Q\}} \qquad \frac{\{P \wedge c\} S1 \{Q\}, \{P \wedge \neg c\} S2 \{Q\}}{\{P\} \text{ 🤔 } (c) \{S1\} \text{ELSE } \{S2\} \{Q\}}$$

Type Casting:

⊢ *e1* : qubit state
⊢ *e2* : $boolean
⊢e1 -> e2 : $boolean

⊢ *e1* : scalar
⊢ *e2* : vector^^n
⊢e1 -> e2 : vector^^n

⊢ *e1* : qubit state
⊢ *e2* : boolean
⊢e1 -> e2 : boolean {or} ⊢e2 -> e1 : qubit state

⊢ *e1* : array[]
⊢ *e2* : T
T is a primitive type
⊢e1 -> e2 : T

⊢ *e1* : array[][]
⊢ *e2* : T
T is a primitive type
⊢e1 -> e2 : T

⊢ *e1* : array[][][]
⊢ *e2* : T
T is a primitive type
⊢e1 -> e2 : T

⊢ *e1* : $char
⊢ *e2* : $string
⊢e1 -> e2 : $string

⊢ *e1* : $byte
⊢ *e2* : $short
⊢ *e3* : $int
⊢ *e4* : $long
⊢ *e5* : double
⊢ *e6* : $qLong
⊢ *e7*: $qDouble
⊢e1 -> e2 -> e3 -> e4 -> e5 -> e6 -> e7 : qDouble

3.2. Value types (How Qsort differs from Q# and Java)

One of the biggest differences in this language is that quantum arrays are treated as primitive types since information can be process simultaneously. This is a result of the hardware having the ability to pass information between quantum registers and classical registers. Variable types can be easily type casted with the arrow operator -> (Type casting takes the form: **variable of type -> type**). This flexibility allows any object comprised of different types to be quickly modified. There is a hierarchy to type casting, i.e. an array cannot be converted to a vector, a string cannot be converted to a char, etc…

Qsort supports the following classical computer primitive types:
1. $boolean (0 or 1; true or false; 👍or 👎)
2. $byte (has an inclusive memory range of -128 to 127)
3. $short (has an inclusive memory range of -32,768 to 32,767)
4. $int (has an inclusive memory range of -2,147,483,648 to 2,147,483,647)
5. $long (has an inclusive memory range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
6. $char (has an inclusive memory range of '\u00000000' to '\uffffffff' or 0 to 4,294,967,295)
7. $float (has an inclusive memory range of -2^{32} to $((2^{32}) - 1)$)
8. $double (has an inclusive memory range of -2^{32} to $((2^{32}) - 1)$))
9. $string

Qsort also supports the following quantum primitive types:
1. qubit (allows access to the states and positions of the states)
2. singleton (a type that only contains one element)
3. boolean (amplitude of |0> and |1> state)
4. qLong (has an inclusive memory range of -2^{64} to $((2^{64}) - 1)$)
5. qDouble (has an inclusive memory range of -2^{128} to $((2^{128}) - 1)$)
6. qFloat (has an inclusive memory range of -2^{128} to $((2^{128}) - 1)$)
7. array[] (dynamic length array)
8. array[][] (2-D dynamic length array)
9. array[][][] (3-D length array)
10. scalar (quantity magnitude)
11. vector^^n (vector of length n)

3.3. Reference types (How Qsort differs from Q# and Java)

Reference types are store as a pointer to one memory location or possibly, many memory locations. Memory locations can be either quantum registers or classical registers, which allows copies of references to be easily manipulated.

4. Example Programs

Six (6) example programs that demonstrate language use:

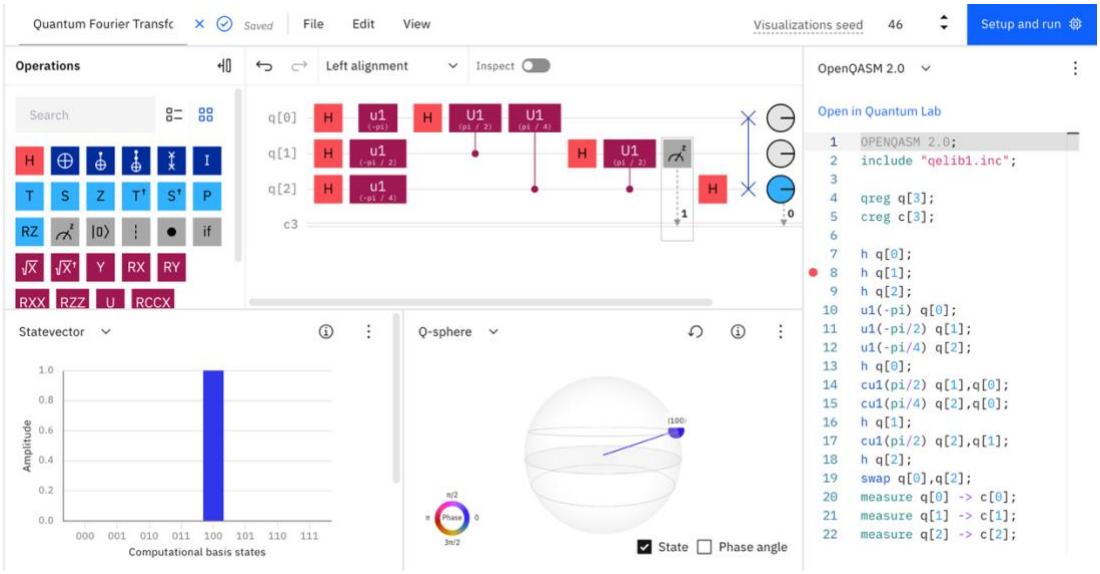
1. Caesar Cipher encrypt

```
import QuantumCircuits, BinaryCPU, Encrypt;
public Main(){
        $string message = "Hello World";
         $int key = 4;
         print("encrypted message: " + Encrypt.encrypt(message, key));
}


Public Class Encrypt(){
      encrypt($string message, $int key) returns $string{

     `create an empty string array
      new array[$string] stringArray = {[?]};

      `typecase message to the stringArray variable above
       message -> array[$string] stringArray;

      `typecast the stringArray into an array of ascii values.
       stringArray -> (array[$int] stringToAsciiValue).toAscii();

       `apply this formula to each element in the array
       stringToAsciiValue as ((stringToAsciiValue.toUpper()  - 65) % 26 + 65)

       return stringToAsciiValue -> $string;

       }
```

2. Caesar Cipher decrypt

```
import QuantumCircuits, BinaryCPU, Decrypt;
public Main(){

      $string message = "Hello World";
       $int key = 4;
       print("decrypted message: " + Decrypt.decrypt(message, key));
}
Public Class Decrypt(){

 decrypt($string message, $int key) returns $string{

     `create an empty string array
      new array[$string] stringArray = {[?]};

     `typecase message to the stringArray variable above
      message -> array[$string] stringArray;

     `typecast the stringArray into an array of ascii value.
      stringArray -> (array[$int] stringToAsciiValue).toAscii();

      `accounts for negatives
       🤔(key < 0 ){
              key = ((key+26)%26)
        }

      `change the shift value to represent a reversal
       key = (26 - (key % 26));

     `apply this formula to each element in the array
      stringToAsciiValue as ((stringToAsciiValue.toUpper() - 65) % 26 + 65);

    return (stringToAsciiValue -> $string);

  }
```

3. Factorial – This is one of the coolest algorithms to think about because the hardware could potentially handle factorial problems more efficiently than a classical computer. The hardware can leverage three qubits by first initializing the circuit using a Hadamard gate, and then utilizing controlled swap gate (cswap) that swaps the states of two qubits if a third qubit is in the state |1>. Shor's algorithm and the Quantum Fourier Transform (QFT) algorithm leverage this concept to make ease of factorization. See an example of the implementation in assembly written below.




```
` Computes factorial using quantum registers
Import QuantumCircuits, QuantumFactorial;

Public Main($int m) returns $int {

    ` Measure the qubits result and type cast to classical integer
    return (m -> QuantumFactorial(q, c));
}

Public QuantumFactorial(array[qubits] qu, $int n) returns singleton {
      `Apply the Hadamard gate to all qubit
        qu[n] as Hadamard;

       ` Apply the conditional phase-shift gates
       qu = QuantumCircuits.controlledPhaseShift(PI / ((2, j -n)^ [q[n]]) , q[n]);

    ` Measure the qubits and store the result in the classical register
    return qu -> singleton qu;
}
```

4. Quantum Fourier Transform

```
import QuantumCircuits, BinaryCPU,
Public QFT ($int numberOfQubits, array[Qubits] qbits) returns singleton {
   for ($int i = 0; i < numberOfQubits – 1; i++) {
      qbits [i] as Hadamard;  `apply Hadamard gate
      for ($int j =  i + 1; j < numberOfQubits - 1) {
         `Apply controlled phase-shift gate to qubit j
        (1.0 / (2 ^ (j – i)), [qbits [j]], qbits [i]) as CPhaseGate; `apply Controlled Phase
Shift
      }
   }
   ` Reverse the order of qubits in the register
   for ($int i = 0; i < (numberOfQubits / 2 – 1); i++) {
      QuantumCircuits.swapgate(qbits [i], qbits[numberOfQubits - i - 1]).;
   }
}
```

5. Greedy Merge

```
import QuantumCircuits, BinaryCPU, MergeSort, Queue, Quicksort;

Public Main( ){
    array[$int] testlists = {[3, 5, 9, 11, 16, 18, 20]};
    print("The minimum cost to merge testlists[] is "+ merge(testlists));
    `Expected output: 216

`supporting method
Public merge(array[$int] lists) returns $int
        Input: Assume lists in lists[] are sorted already
        The resulting list size = lists[i] + lists[j]
        Output: the minimum cost of merging all lists in lists[]

        array[$int] arr = {[?]};
        array[$int] mergedCost = {[?]};

        `copy the int array to the array list
        arr = lists;  `copies a copy of array to adjacent memory location

        `size is a representation of the size of a file stored as an element in an array
        $int size = 0;

        `so long as there are more than 2 elements...

        while(arr.length >= 2){

            ` basically, sum the lowest two element listed in ascending order
             size = arr[0] + arr.[1];

            `then remove the elements from the array list
             arr.dequeue( );
             arr.dequeue( );

            `add back the sum combined file size (s1+s2)
             arr.enqueue(size);

            `then sort the array list
            Quicksort.qSort(arr);
            `add the costs of each merger to a separate array
            mergedCost.enqueue(size);

        }

        `the combined total cost of the merged cost array is the total optimal cost
        $int totalCost = 0;
        for(int k=0; k < mergedCost.length; k++){

            `sum all the values in the array
             totalCost = mergedCost.get(k) + totalCost;
         }
         return totalCost;

    }
```

6. Quick Sort (Qsort)

```
import QuantumCircuits, BinaryCPU, Quicksort;

Public Class Program6 {

    Public Main(){
        ` Test cases
        array[$int] testarray1 = {[2, 4, 1, 6, 3, 7, 8]};
        array[$int] testarray2 = {[6, 3, 4, 5, 1]};

        quicksort_2pivots(testarray1, 0, testarray1.length-1);
        quicksort_2pivots(testarray2, 0, testarray2.length-1);

        `Output sorted arrays
        testarray1[$int] -> $string testarray1;
        print(testarray1 + "\n");

        testarray2[$int] -> $string testarray2;
        print(testarray2 + "\n");
      }`main


Public partition(array[$int] A, $int s, $int e) returns $int{
        $int pivot = A[s];
        $int i = s + 1;
        $int j = e;

        while (i<=j){
            while (i < e & A[i] < pivot){
                    i = i + 1;
             }

             while(j > s & A[j] >= pivot){
                    j = j - 1;
             }

             if (i >= j){
                   break;
             }
            QuantumCircuits.cswap(A, i, j);
}
             QuantumCircuits.cswap(A, s, j);
             return j;
}
```

```
Public quicksort(array[$int] A, $int s, $int e){
      🤔 ( s < e ){
         $int p = partition(A, s, e);
         quicksort(A, s, p-1);
         quicksort(A, p+1, e);
         }
 }


 Public partition_2pivots(array[$int] A, $int s, $int e) returns int[]{
```

📝 Randomly select 2 pivots and partition the array
📝 Return the position of 2 pivots after partition
📝 Complete the function
📝 Feel free to change the return type and parameters

```
      $int pivot1 = QuantumCircuits.random(e-s)+s;
      $int pivot2 = QuantumCircuits.random(e-s)+s;

      if(pivot1 > pivot2){
          QuantumCircuits.swap(pivot1, pivot2);
       }
```

` adapted from: https://www.geeksforgeeks.org/quicksort-using-random-pivoting/

```
    $int i = s+1;
    $int j = e-1;

    $int temp2 = A[pivot1];
    A[pivot1]= A[s];
    A[s] = temp2;


    $int temp3 = A[pivot2];
    A[pivot2]= A[e];
    A[e] = temp3;

    $int p = A[s];
    $int q = A[e];


    while (i <= j){
```

` If elements are less than the left pivot

```
       🤔 (A[i] < p)
      {

      QuantumCircuits.swap(A, i, k);
          k++;
      }
```

```
        ` If elements are greater than or equal to the right pivot
        else if (A[i] >= q){
            while (A[j] > q && i < j){
                    j--;
            }


           QuantumCircuits.swap (A, i, j);
                  j = j- 1;
           if (A[i] < p){
              QuantumCircuits.swap (A, i, k);
              k++;
            }
          }
       i= i +1;
     }


      ` Bring pivots to their appropriate positions.
     QuantumCircuits.swap (A, s, k);
     QuantumCircuits.swap (A, e, j);
     /* Returning the indices of the pivots because we cannot return two elements from a
   function, we do that using an array */
     return new array[int] {[ k, j ]};
   }
Public quicksort_2pivots(array[$int] A, $int s, $int e){
     `quicksort that uses the modified partition_2pivots()
          🤔 ( s < e ){
             int[] p;
             p = partition_2pivots(A, s, e);
              ` adaptation of : https://www.geeksforgeeks.org/dual-pivot-quicksort/
              /* p[] stores left pivot and right pivot. p[0] means left pivot and p[1] means right
   pivot */
              quicksort_2pivots(A, s, p[0] - 1 );          ` {2, 4, 1, 6, 3, 7, 8};
              quicksort_2pivots(A, p[0] + 1, p[1] - 1);
              quicksort_2pivots(A, p[1]+1, e);
          }
       }
   }//Program6
```