UNIVERSITÀ DEGLI STUDI DI VERONA

DIPARTIMENTO DI INFORMATICA

QUANTUM DESK CALCULATOR IN QISKIT

Luca Verdolini VR461133
Federico Graziola VR481884

A.A 2021/2022

Indice

1 Definizione del problema

Il compito è costruire circuiti quantistici che eseguano operazioni aritmetiche tra le rappresentazioni binarie di due interi positivi.
Il circuito deve essere implementato in Qiskit e dimostrato in una o due istanze.

Costruzione Quantum Desk Calculator

Partiremo introducendo la DFT e successivamente la QFT. Sulla base delle considerazioni scritte sulla QFT implementeremo tutte le operazioni quali: addizione, sottrazione, moltiplicazione, divisione ed esponenziale.

Il codice sorgente di questo progetto è disponibile su:
https://github.com/ilverdo97/QuantumDeskCalculator

2 Introduzione della Trasformata discreta di Fourier (DFT)

Prima di immeggerci nel QFT, è importante conoscere il DFT perchè il QFT è derivato dal DFT.
La DFT è nota come rappresentazione nel dominio della frequenza di una sequenza di input.
Viene utilizzato nell'analisi spettrale di vari segnali, come la riduzione della varianza di uno spettro. Viene anche utilizzato nella compressione con perdita di dati di immagini e suoni.
Oltre alle precedenti applicazioni citate, la DFT viene utilizzata anche in matematica, ad esempio per la risoluzione di equazioni differenziali alle derivate parziali e anche nella moltiplicazione polinomiale. Poiché la trasformata di Fourier a tempo discreto (DTFT) di una sequenza rappresenta una trasformazione continua e periodica della sequenza di input, se campioniamo il DTFT a intervalli periodici, otteniamo la DFT della sequenza di input.
Matematicamente, il DFT è definito dalla seguente equazione:

$$F(\omega) = \sum_{j=0}^{N-1} e^{\frac{2\pi jik}{N}} X_j \tag{1}$$

Dall'equazione matematica della DFT, è chiaro che esiste una mappatura dal dominio di input, Xj, al dominio di output, F(), che è il dominio della frequenza dell'input. N è il numero totale di sequenze di input. L'equazione precedente è la trasformazione DFT. Poiché la DFT è una trasformazione da una funzione all'altra, in modo molto simile, possiamo derivare l'analogo quantistico in cui possiamo trasformare uno stato quantistico da uno stato a un altro stato.

3 Trasformata di Fourier quantistica (QFT)

Nella sezione precedente, abbiamo visto che il DFT classico è un processo di mappatura o una trasformazione di una funzione da un dominio all'altro. In modo molto simile, il QFT è una trasformazione degli stati quantistici dalla base Z alla base X (la base di Hadamard). L'operazione di Hadamard è il QFT a un qubit.
Il QFT esegue un'operazione DFT ma non su una sequenza classica, piuttosto su uno stato quantistico. Il QFT è un algoritmo importante da studiare poichè viene utilizzato come subroutine per altri algoritmi quantistici, come l'algoritmo di stima della fase quantistica. Matematicamente la trasformazione QFT è definita come segue:

$$QFT\,|x\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{\frac{2\pi ixy}{N}} |y\rangle \tag{2}$$

La scoperta finora piu‘ importante in computazione quantistica consiste nella dimostrazione che fattorizzare un mumero di n bit su un computer quantistico richiede un numero di operazioni dell'ordine di:

$$O(n^2 \log n \log\log n) \tag{3}$$

mentre il migliore algoritmo classico che si conosce richiede tempo esponenziale.

Il problema di stabilire se e quali altri problemi che risultano intrattabili su un computer classico possono essere risolti efficientemente con un computer quantistico e di fondamentale importanza, rappresenta un punto cruciale della ricerca corrente.
Un ingrediente fondamentale dell'algoritmo quantistico per la fattorizzazione è la Trasformata di Fourier Quantistica (QFT) che ha applicazioni anche nella soluzione di altri problemi classicamente noti come problemi `NP`.

Un processo fisico, in generale, può essere espresso sia nel dominio dei tempi, con una funzione h(t) che descrive la variazione di una generica grandezza, sia nel dominio delle frequenze. La trasformata di Fourier è un metodo matematico per passare dal dominio dei tempi a quello delle frequenze. Questo metodo permette di trasformare una funzione di periodo `r` in una funzione che assume valori diversi da zero solo in corrispondenza dei multipli della frequenza

$$\frac{2\pi}{r}$$

La Trasformata di Fourier ‘e alla base di una procedura generale nota come stima delle fasi che permette di ottenere una stima degli autovalori di una matrice unitaria. Questa procedura ‘e a sua volta parte essenziale di molti algoritmi quantistici.

4 Funzione Addizione (QFT)

Prima di iniziare a discutere l'implementazione dell'addizione utilizzando la trasformata di Fourier quantistica (QFT), vediamo come è possibile implementare il circuito che esegue questa operazione. Come accennato in precedenza, la QFT è l'implementazione quantistica della trasformata discreta di Fourier sulle ampiezze di una funzione d'onda e che trasforma la base computazionale nella base di Fourier.

L'algoritmo di addizione ideale per un computer quantistico potrebbe non essere simile alla sua controparte classica. Un' alternativa quantistica è la Trasformata Quantistica di Fourier (QFT). In particolare, l'uso del QFT consente di risparmiare memoria e tempo.
La QFT può essere interpretata come un cambio di base: la base computazionale (Z) e la base di Fourier. Nella base computazionale, memorizziamo i numeri in binario usando gli stati $|0\rangle \, e \, |1\rangle$.

Nella base di Fourier, tuttavia, memorizziamo i numeri utilizzando diverse rotazioni attorno all'asse Z. Il numero che vogliamo memorizzare regola l'angolo con cui ogni qubit viene ruotato attorno all'asse Z. Intuitivamente, la somma di due qbit utilizzando il QFT può essere vista come la somma delle rotazioni attorno all'asse di Fourier.
Inizia codificando il primo numero usando il QFT. Quindi partendo dal numero appena codificato, sommo per ogni qbit le rotazioni del secondo numero. Infine, si applica l'inverso del QFT, che porta il risultato nella base computazionale.
È come se contassimo il numero di rotazioni per ogni qbit.
Abbiamo diviso il codice usando le funzioni, quindi spieghiamole una per una.

4.1 Implementazione QFT

4.1.1 Execute QFT

La funzione `executeQFT` calcola il QFT del primo numero di input, cambiando di fatto la base da quella computazionale a quella di Fourier. Il primo ingrediente per costruire un circuito QFT è la porta di Hadamard, una per ogni registro qbit. Dobbiamo quindi applicare tante rotazioni controllate (cr) quante sono le posizioni binarie del qbit. La funzione è mostrato nel Listato 1.

```
    '''
qc: circuito quantistico di ingresso
     reg: inserire il registro per eseguire QFT
     n: n-esimo qbit per applicare hadamard e rotazione di fase
     pie: numero del pie
    '''

    def executeQFT(qc, reg, n, pie):
         # Esegue il QTF di reg, un qubit alla volta
         # Applicare una porta Hadamard all'n-esimo qubit del registro
         # quantistico reg, e
         # quindi applica rotazioni di fase ripetute con i parametri pi divisi
         # per potenze crescenti di due

        qc.h(reg[n])
        for i in range(0, n):
            # cp(theta, control_qubit, target_qubit[,    ])
            qc.cp(pie / float(2 ** (i + 1)), reg[n - (i + 1)], reg[n])

        #print(qc.draw())

```

Listing 1: QFT:Funzione executeQFT

La funzione evolveQFTStateSum evolve lo stato codificato precedente

$$|| F(\psi(rega))\rangle \tag{4}$$

del primo numero a

$$|| F(\psi(rega + regb))\rangle \tag{5}$$

considerando ora anche il secondo numero.
Intuitivamente, partendo dal numero appena codificato con il QFT, somma per ogni qbit le corrispondenti rotazioni del secondo numero.
Il funzionamento è molto simile alla precedente funzione appena descritta sopra.
Il codice è mostrato nel Listato 2.

4.1.2 Evolve QFT State Sum

```
        '''
         qc: circuito quantistico di ingresso
         reg: inserire il registro per eseguire QFT
         n: n-esimo qbit per applicare hadamard e rotazione di fase
         pie: numero del pie
        '''

        def executeQFT(qc, reg, n, pie):
            # Esegue il QTF di reg, un qubit alla volta
            # Applicare una porta Hadamard all'n-esimo qubit del registro
            # quantistico reg, e
            # quindi applica rotazioni di fase ripetute con i parametri pi
            # divisi per potenze crescenti di due

```

```
            qc.h(reg[n])
            for i in range(0, n):
                # cp(theta, control_qubit, target_qubit[,    ])
                qc.cp(pie / float(2 ** (i + 1)), reg[n - (i + 1)], reg[n])

            #print(qc.draw())

```

Listing 2: QFT:Funzione evolveQFTStateSum

4.1.3 Inverse QFT

Dopo aver eseguito le due funzioni precedenti, è ora di tornare alla base computazionale. Per calcolarlo, si applica la funzione inverseQFT ripetuta alle rotazioni di fase e quindi applicando una porta di Hadamard.
A differenza della funzione executeQFT, l'inverso viene applicato a partire dall'ultimo qbit ovvero il meno significativo e andando al primo qbit il più significativo.
Tuttavia, la procedura rispetta sempre la regola sopra descritta, dove per per ogni qbit applichiamo tante rotazioni controllate quante sono la posizione della cifra nel sistema binario.
Il codice è mostrato nel Listato 3.

```
    '''
    qc: circuito quantistico di ingresso
    reg: inserire il registro per eseguire QFT
    n: n-esimo qbit per applicare hadamard e rotazione di fase
    pie: numero del pie
    '''

    def inverseQFT(qc, reg, n, pie):
        # Esegue il QFT inverso su un registro reg.
        # Applicare rotazioni di fase ripetute con i parametri pi divisi per
        # potenze decrescenti di due, quindi applicare una porta Hadamard
        # all'ennesimo qubit
        # del registro reg.

        for i in range(n):
            # cp(theta, control_qubit, target_qubit[, ...])
            qc.cp(-1 * pie / float(2 ** (n - i)), reg[i], reg[n])
        qc.h(reg[n])

```

Listing 3: QFT:Funzione inverseQFT

Abbiamo anche creato la funzione initQubits per codificare un classico numero binario in qbit, capovolgendo nell'ordine inverso il qbit corrispondente.
Quella è perché prima di tutto il bit meno significativo di Qiskit ha l'indice più basso (0) e inoltre, quando eseguiamo la somma, di solito partiamo dalla fine del numero, prendiamo le cifre meno significative e passiamo la cifra più significativa.
Il codice è mostrato nel Listato 4.

4.1.4 Init Qubits

```
    def initQubits(str, qc, reg, n):
        # Capovolgere il qubit corrispondente nel registro se un bit nella
        # stringa e' un 1
        for i in range(n):
            if str[i] == "1":
                qc.x(reg[n-(i+1)])

```

Listing 4: QFT:Funzione initQubits

La funzione `"sum"` raccoglie tutte le funzioni precedenti e le chiama per eseguire l'aggiunta QFT. Il codice è mostrato nel Listato 4.

```
    def sum(a, b, qc):
    n = len(a) - 1
    # Calcola la trasformata di Fourier del registro a

    for i in range(n + 1):
        executeQFT(qc, a, n - i, pie)

    # Somma i due numeri evolvendo la trasformata di Fourier F(  (reg_a))>
    # to |F(  (reg_a+reg_b))>
    for i in range(n + 1):
        evolveQFTStateSum(qc, a, b, n - i, pie)

    # Calcola la trasformata di Fourier inversa del registro a
    for i in range(n + 1):
        inverseQFT(qc, a, i, pie)

```

Listing 5: QFT:Funzione Sum

Abbiamo utilizzato i colori per stampare i risultati e l'interfaccia utente, utizzando la funzione "printResult". La funzione Misura il risultato memorizzandolo nel `ClassicalRegister`. La funzione esegue l'operazione effettiva nel file simulatore quantistico, stampando il risultato.

```
##############################  Quantum Desk Calculator  ##############################

Select one operator [+ addition, - subtraction, * multiplication, / division, ^ exponential]: +
Enter a first positive integer between 0 and 2000:
1008
Enter a second positive integer between 0 and 2000:
1008
#######################################################################################################
You want to perform the following operations:
1008 + 1008 = ...
Create and Connecting to local simulator...
100% (100 of 100) |######################| Elapsed Time: 0:00:06 Time:  0:00:06

1008 + 1008 = 2016 with a probability of 100%
```

Figura 1: Stampa esecuzione e risultato Funzione Somma

5 Funzione Sottrazione (QFT)

Se pensiamo a come avviene la sottrazione tra numeri binari, allora in calcolo classico, sappiamo che possiamo ottenere il risultato usando "un complemento" dell'addendo, aggiungendone "1" e facendo quindi la somma tra il risultato ottenuto e il primo numero originale, rimuovendo la prima cifra della somma.
Se non abbiamo una cifra in più, cerchiamo di sottrarre una cifra più grande del numero da uno più piccolo.
Se facciamo un semplice esempio abbiamo:

$$\begin{aligned} 101_2 - 011_2 &= 101_2 + 100_2 \\ &= 101_2 + (100_2 + 001_2) \\ &= 101_2 + 101_2 \\ &= 1010_2 + \not{1}010_2 \\ &= 010_2 \end{aligned}$$

Infatti:

$$5_{10} - 3_{10} = 2_{10} = 010_2$$

Ancora una volta l'utilizzo del QFT ci consente di saltare numerosi passaggi e risparmiare qubit. L'algoritmo che seguiamo per la sottrazione è essenzialmente identico a quello seguito per l'addizione, ma invece di eseguire rotazioni positive nella funzione evolveQFTStateSum che come accennato prima evolve la precedente stato codificato:

$$|| \, F(\psi(rega)\rangle \tag{6}$$

dal primo numero a

$$|| \, F(\psi(rega + rega))\rangle \tag{7}$$

,noi eseguiamo rotazioni negative.
La nuova funzione evolveQFTStateSub è mostrata nel Listato 6.

5.1 Evolve QFT State Sub

```
    '''
    qc: circuito quantistico di ingresso
    reg_a: primo registro di input per eseguire QFT
    reg_b: secondo registro di input per eseguire QFT
    n: n-esimo qbit per applicare hadamard e rotazione di fase
    pie: numero del pie
    '''

    def evolveQFTStateSub(qc, reg_a, reg_b, n, pie):
        # Fa evolvere lo stato |F(  (reg_a))> to |F(  (reg_a+reg_b))>
        # usando il quatum
        # Trasformata di Fourier condizionata dai qubit del reg_b.
        # Applicare rotazioni di fase ripetute con i parametri pi divisi per
        # potenze crescenti di due.

        l = len(reg_b)
        for i in range(n + 1):
            if (n - i) > l - 1:
```

```
                pass
            else:
                qc.cp(-1 * pie / float(2 ** (i)), reg_b[n - i], reg_a[n])


```

Listing 6: QFT:Funzione evolveQFTStateSub

Intuitivamente, poiché vogliamo eseguire la sottrazione, prima convertiamo i due numeri nelle rispettive basi di Fourier e quindi eseguiamo altrettanti rotazioni negative sull'asse Z come numero codificato nel registro B. La sottofunzione è mostrata nel Listato 7.

```
    def sub(a, b, qc):
        n = len(a)

        # Calcola la trasformata di Fourier del registro a
        for i in range(0, n):
            executeQFT(qc, a, n - (i + 1), pie)
        # Somma i due numeri evolvendo la trasformata di Fourier
        # F(  (reg_a))> to |F(  (reg_a-reg_b))>
        for i in range(0, n):
            evolveQFTStateSub(qc, a, b, n - (i + 1), pie)
        # Calcola la trasformata di Fourier inversa del registro a
        for i in range(0, n):
            inverseQFT(qc, a, i, pie)

```

Listing 7: QFT:Funzione Subtraction

La funzione principale che calcola la sottrazione `"sub"` richiama le funzioni definite precedentemente nell'addizione come `executeQFT` e `inverseQFT` che insieme alla funzione `evolveQFTStateSub` definita sopra ci permettono di calcolare la sottrazione con QFT.
Per l'esecuzione facciamo riferimento al file quantumDeskCalculator.py.

```
############################  Quantum Desk Calculator  ############################

Select one operator [+ addition, - subtraction, * multiplication, / division, ^ exponential]: -
Enter a first positive integer between 0 and 2000:
50
Enter a second positive integer between 0 and 2000:
24
#################################################################################################
You want to perform the following operations:
50 - 24 = ...
Create and Connecting to local simulator...
100% (100 of 100) |######################| Elapsed Time: 0:00:03 Time:  0:00:03

50 - 24 = 26 with a probability of 100%
```

Figura 2: Stampa esecuzione e risultato Funzione Sottrazione

6 Funzione Moltiplicazione (QFT)

La Function Multiplication è una successione della Function SUM. Una moltiplicazione può essere vista come una successione di somme.

`Es: Supponiamo di voler svolgere 5*5, sappiamo che possiamo riscrivere l'operazione come 5+5+5+5+5`

Sfrutteremo quindi il circuito di SUM senza dover creare un nuovo circuito. In questo caso non è necessario svolgere un collasso del circuito poichè i registri del circuito vengono aggiornati in progressione.
Notiamo che l'operazione Function SUM viene iterata un numero di volte pari al secondo numeratore, di seguito la Function Multiplication al listato 8.

```
    def multiply(a, secondDec, result, qc):
        n = len(a) - 1
        # Calcola la trasformata di Fourier del registro 'result'
        for i in range(n + 1):
            executeQFT(qc, result, n - i, pie)

        # Somma i due numeri evolvendo la trasformata di Fourier
        # F(  (reg_a))>
        # to |F(  ((second * reg_a))>, dove giriamo sul
        # somma tante volte quanto dice 'second',
        # facendo somme incrementali
        for j in range(secondDec):
            for i in range(n + 1):
                evolveQFTStateSum(qc, result, a, n - i, pie)

        # Calcola la trasformata di Fourier inversa del registro a
        for i in range(n + 1):
            inverseQFT(qc, result, i, pie)

```

Listing 8: QFT:Funzione Multiplication

Viene proposto un esempio dell'esecuzione del programma:

```
############################## Quantum Desk Calculator ##############################

Select one operator [+ addition, - subtraction, * multiplication, / division, ^ exponential]: *
Enter a first positive integer between 0 and 2000:
5
Enter a second positive integer between 0 and 2000:
5
#####################################################################################################
You want to perform the following operations:
5 * 5 = ...
Create and Connecting to local simulator...
100% (100 of 100) |######################| Elapsed Time: 0:00:05 Time:  0:00:05

5 * 5 = 25 with a probability of 100%
```

Figura 3: Stampa esecuzione e risultato Funzione Moltiplicazione

7 Funzione Divisione (QFT)

La Function Division è una successione di Function SUB. Una divisione è una sottrazione ripetuta più volte fino a quando il divisore risulta più piccolo o uguale al dividendo, che per semplicità non abbiamo implementato con i numeri dietro la virgola. Mantenendo questa idea non abbiamo quindi creato un nuovo circuito per la Division, ma riutilizzato il circuito della SUB facendolo collassare ogni volta che viene effettuata una sottrazione per poi riattivarlo. Svolgendo questo otteniamo il risultato di un passo della successione delle sottrazioni e carichiamo nel nuovo circuito il risultato ottenuto.

`Es: Supponiamo di voler svolgere 15/5:`

- Svolgiamo 15 - 5 = 10
- 10 è il risultato dato dal collasso del circuito
- Ricreiamo il circuito
- Inseriamo 10 come dividendo per svolgere 10 - 5
- Ripeteremo questa operazione fino a quando il dividendo è $\geq$ al divisore.

L'operazione di collasso del cirucito è implemetata come segue nel listato 9.

```
def collass(qc, measure, cl, nqubit):

    # Misura i qubit
    for i in range(nqubit + 1):
        qc.measure(measure[i], cl[i])

    job = execute(qc, backend=Aer.get_backend('qasm_simulator'), shots=100)
    # Ottieni risultati del programma
    job_stats = job.result().get_counts()

    for key, value in job_stats.items():
        tmp = key
        prob = value

    return tmp, prob
```

Listing 9: QFT:Funzione Collass

Qui di seguito il codice della Function Division, notiamo che non sappiamo per quante volte il nostro algoritmo dovrà effettuare le sottrazioni, infatti esso si fermerà solo quando il dividendo è $\geq$ al divisore. Il numero di volte che effettuaiamo i passi è pari a numero intero del risultato della divisione.

```
    def div(first, second, dividend, divisor, qc, nqubit, cl):

    result = 0
    while True:
        sub(dividend, divisor, qc)
        tmp, prob = collass(qc, dividend, cl, nqubit)

        numdividend = int(tmp, 2)
        numdivisor = int(second, 2)

        result = result + 1
        if numdividend >= numdivisor:

```

```
            numdividend = '{0:{fill}11b}'.format(numdividend, fill='0')
            numdivisor = '{0:{fill}11b}'.format(numdivisor, fill='0')

            len1 = len(numdividend)
            len2 = len(numdivisor)
            nqubit, len1, len2, newnum, second = NQubit("/", len1, len2,
    numdividend, numdivisor)

            dividend = qiskit.QuantumRegister(nqubit + 1, "a")
            divisor = qiskit.QuantumRegister(nqubit + 1, "b")
            cl = qiskit.ClassicalRegister(nqubit + 1, "cl")
            qc = qiskit.QuantumCircuit(dividend, divisor, cl, name="qc")
            initQubits(numdividend, qc, dividend, nqubit)
            initQubits(numdivisor, qc, divisor, nqubit)

        else:
            break

    return int(first, 2), numdivisor, result, prob

```

Listing 10: QFT:Funzione Division

Viene proposto un esempio dell'esecuzione del programma:

```
############################  Quantum Desk Calculator  ############################

Select one operator [+ addition, - subtraction, * multiplication, / division, ^ exponential]: /
Enter a first positive integer between 0 and 2000:
15
Enter a second positive integer between 0 and 2000:
5
##################################################################################################
You want to perform the following operations:
15 / 5 = ...
Create and Connecting to local simulator...
100% (100 of 100) |######################| Elapsed Time: 0:00:03 Time:  0:00:03

15 / 5 = 3 with a probability of 100%
##################################################################################################
```

Figura 4: Stampa esecuzione e risultato Funzione Divisione

8 Funzione Esponenziale (QFT)

La Function Exponential segue l'ideologia della Function Division, ovvero sfrutta un circuito già presente, in questo caso la SUM. Ricordiamo che un esponenziale non è altro che una successione di moltiplicazioni.

`Prendiamo come esempio 5**3, possiamo riscrivere l'operazione come 5x5x5, sappiamo però che 5x5 è pari a 5+5+5+5+5.`
`Quindi 5**3 è una successione di somme:  5+5+...+5.`

La Function Exponential sfrutterà quindi la Function Multiplication, quest' ultima, come già spiegato precedentemente, implementa il circuito della SUM. Il nostro algoritmo sarà quindi una successione di moltiplicazioni.
Abbiamo però ora lo stesso problema della Function Division, ovvero dobbiamo far collassare il circuito per ottenere il risultato temporaneo.

`Riprendiamo l'esempio 5**3, abbiamo detto che possiamo riscrivere come 5x5x5, i passi quindi dell'algoritmo sono noti e sono pari all'esponente meno 1.`
`Al primo passo avremo la Function Multiplication che svolgerà 5x5 dando come risultato 25.`
`25 è il risultato del collasso del circuito, quindi dovremo ricreare il circuito, ponendo ora come ingresso 25x5.`

Di seguito il codice della Function Exponential che come per la Function Division è presente la Funzione Collass (Listato 8) che ci permette di ottenere il risultato temporaneo dell'operazione ed evitarci di dover ricreare un nuovo circuito.

```
    def exponential(a, firstDecBinary, firstDec, secondDec, result, qc, cl,
    nqubit):

    for x in range(secondDec - 1):

        multiply(a, firstDec, result, qc)
        tmp, prob = collass(qc, result, cl, nqubit)

        firstDec = int(tmp, 2)

        if x < (secondDec - 2):
            a = qiskit.QuantumRegister(nqubit + 1, "a")
            b = qiskit.QuantumRegister(nqubit + 1, "b")
            cl = qiskit.ClassicalRegister(nqubit + 1, "cl")
            qc = qiskit.QuantumCircuit(a, b, cl, name="qc")
            initQubits(firstDecBinary, qc, a, nqubit)

    if secondDec == 0:
        firstDec = 1
        prob = 100
    elif secondDec == 1:
        firstDec = int(firstDecBinary, 2)
        prob = 100

    return int(firstDecBinary, 2), secondDec, firstDec, prob


```

Listing 11: QFT:Funzione Exponential

Viene proposto un esempio dell'esecuzione del programma:

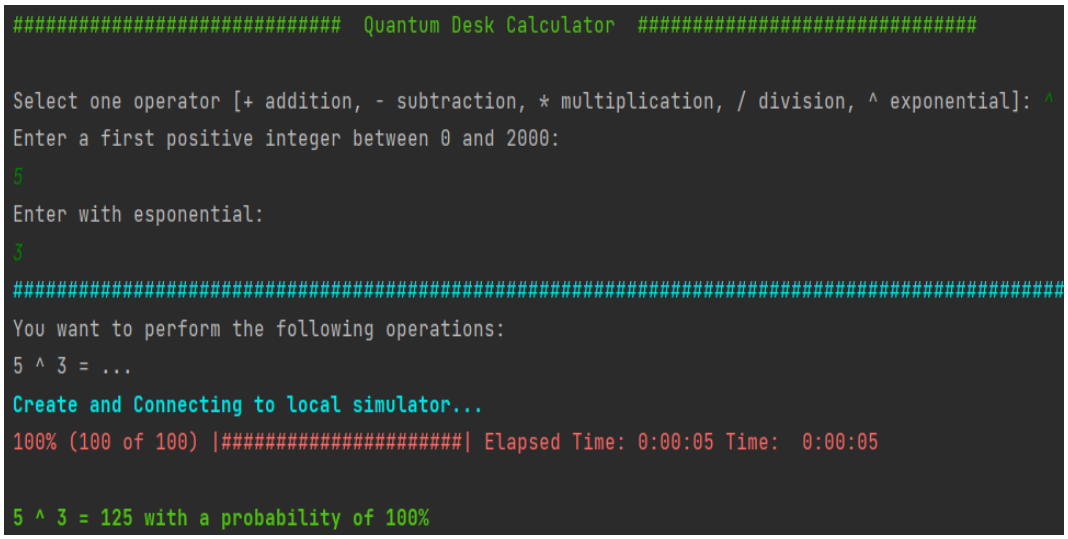

Figura 5: Stampa esecuzione e risultato Funzione Esponenziale

9 Bibliografia

[1] Srinjoy Ganguly e Thomas Cambier "Quantum Computing with Silq Programming": Get up and running with quantum computing with the simplicity of this new high-level programming language. April 2021

10 Sitografia

[1] https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html