UNIVERSITÁ DEGLI STUDI DI VERONA

DIPARTIMENTO DI INFORMATICA

QUANTUM DESK CALCULATOR IN QISKIT

Luca Verdolini VR461133
Federico Graziola VR481884

A.A 2021/2022

Indice

1 Problem definition

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.

1.1 Building the calculator

Partiremo implementando sulla base delle considerazioni scritte sulla QFT, 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 Function QFT Addiction

Prima di iniziare a discutere l'implementazione dell'addizione utilizzando la trasformata di Fourier quantistica (QFT), presentiamo la formula matematica e 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.

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

Figura 1: QFT

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 $\mid 0 > e \mid 1 >$ .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, somma 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.

La funzione executeQFT calcola il QFT del primo numero di input, cambiando di fatto la base da quella computazionale a quella di Fourier uno.
Il codice è mostrato nel Listato 1.

2.1 ExecuteQFT

```
    '''
    qc: input quantum circuit
    reg: input register to execute QFT
    n: n-th qbit to apply hadamard and phase rotation
    pie: pie number
    '''

    def executeQFT(qc, reg, n, pie):
        # Executes the QTF of reg, one qubit a time
        # Apply one Hadamard gate to the n-th qubit of the quantum register reg, and
        # then apply repeated phase rotations with parameters being pi divided by
        # increasing powers of two

        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:Function executeQFT

La funzione evolveQFTStateSum evolve lo stato codificato precedente

$$| F(\psi(rega)) >$$

del primo numero a

$$| F(\psi(rega + regb)) >$$

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.

2.2 EvolveQFTStateSum

```
    '''
    qc: input quantum circuit
    reg: input register to execute QFT
    n: n-th qbit to apply hadamard and phase rotation
    pie: pie number
    '''

    def executeQFT(qc, reg, n, pie):
        # Executes the QTF of reg, one qubit a time
        # Apply one Hadamard gate to the n-th qubit of the quantum register reg, and
        # then apply repeated phase rotations with parameters being pi divided by
        # increasing powers of two

        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:Function evolveQFTStateSum

2.3 InverseQFT

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 applicaando 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 quanta la posizione della cifra nel sistema binario.
Il codice è mostrato nel Listato 3.

```
'''
```

```
    qc: input quantum circuit
    reg: input register to execute QFT
    n: n-th qbit to apply hadamard and phase rotation
    pie: pie number
    '''

    def inverseQFT(qc, reg, n, pie):
        # Executes the inverse QFT on a register reg.
        # Apply repeated phase rotations with parameters being pi divided by
        # decreasing powers of two, and then apply a Hadamard gate to the nth qubit
        # of the register 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:Function 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 di il numero, le cifre meno significative e passare a quella più significativa cifra.
Il codice è mostrato nel Listato 4.

2.4 InitQubits

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

```

Listing 4: QFT:Function 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
# Compute the Fourier transform of register a

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

# Add the two numbers by evolving the Fourier transform F
(   (reg_a))>
# to |F(   (reg_a+reg_b))>
for i in range(n + 1):
    evolveQFTStateSum(qc, a, b, n - i, pie)

    # Compute the inverse Fourier transform of register a
for i in range(n + 1):
    inverseQFT(qc, a, i, pie)

```

Listing 5: QFT:Function Sum

Abbiamo utilizzato i colori per stampare i risultati e l’interfaccia utente, utizzando la funzione ”printResult”. La funzione Misura il risultato memorizzandolo nel Classi calRegister cl. 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 2: Print Result Function Sum

3 Function QFT Subtraction

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:

$$
\begin{gathered}
101_2 - 011_2 = 101_2 + 100_2 \\
= 101_2 + (100_2 + 001_2) \\
= 101_2 + 101_2 \\
= 1010_2 + \cancel{1}010_2 \\
= 010_2
\end{gathered}
$$

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:

$$| \; F(\psi(rega) >$$

dal primo numero a

$$| \; F(\psi(rega + rega)) >$$

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

3.0.1 EvolveQFTStateSub

```
'''
qc: input quantum circuit
reg_a: first input register to execute QFT
reg_b: second input register to execute QFT
n: n-th qbit to apply hadamard and phase rotation
pie: pie number
'''

def evolveQFTStateSub(qc, reg_a, reg_b, n, pie):
    # Evolves the state |F(  (reg_a))> to |F(  (reg_a+reg_b))> using the quantum
    # Fourier transform conditioned on the qubits of the reg_b.
    # Apply repeated phase rotations with parameters being pi divided by
    # increasing powers of two.

    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:Function evolveQFTStateSub

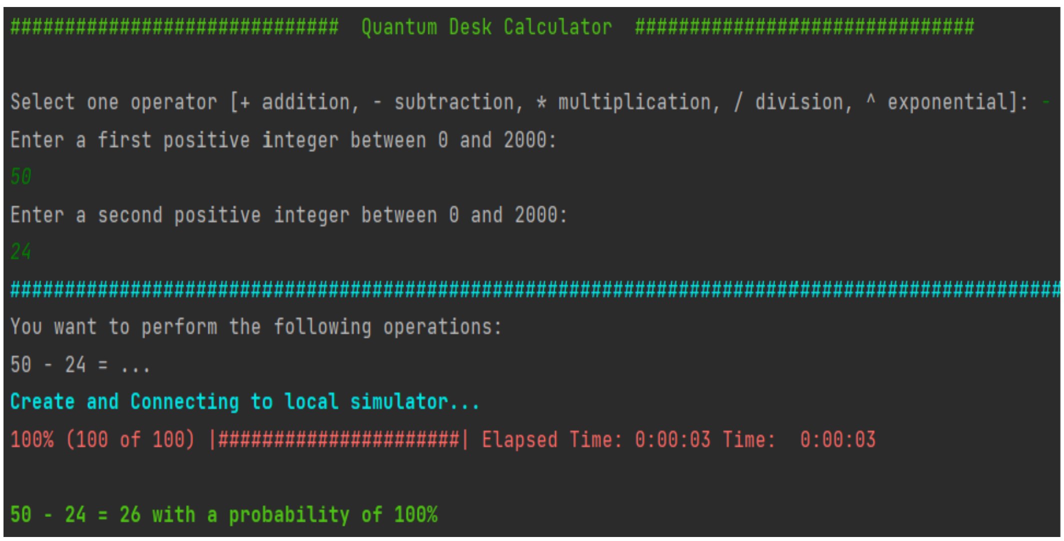
Intuitivamente, poiché vogliamo eseguire la sottrazione, prima convertiamo i due numeri nelle rispettive basi di Fourier e quindi eseguiamo altrettanti rotazioni negative sull'asse Z come numero codificato nel registro B. La sottofunzione è mostrata nel Listato 7.

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

    # Compute the Fourier transform of register a
    for i in range(0, n):
        executeQFT(qc, a, n - (i + 1), pie)
    # Add the two numbers by evolving the Fourier
    # transform F(  (reg_a))> to |F(  (reg_a-reg_b))>
    for i in range(0, n):
        evolveQFTStateSub(qc, a, b, n - (i + 1), pie)
```

```python
        # Compute the inverse Fourier transform of register a
        for i in range(0, n):
            inverseQFT(qc, a, i, pie)

```

Listing 7: QFT:Function Subtraction

Abbiamo deciso di saltare la funzione principale poiché è molto simile alla funzione principale utilizzato nella somma. Se sei interessato a eseguirlo, fai riferimento al file deskCalcu lator.py.
Vengono eseguite le stesse operazioni, ma chiamando il metodo secondario.



```
##############################  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 3: Print Result Function Sum

4 Function Multiplication

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

Es: Supponiamo di voler svolgere 5x5, 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
        # Compute the Fourier transform of register 'result'
        for i in range(n + 1):
            executeQFT(qc, result, n - i, pie)

        # Add the two numbers by evolving the Fourier
        # transform F(  (reg_a))>
        # to |F(  ((second * reg_a))>, where we loop on the
        # sum as many times as 'second' says,
        # doing incremental sums
        for j in range(secondDec):
            for i in range(n + 1):
                evolveQFTStateSum(qc, result, a, n - i, pie)

        # Compute the inverse Fourier transform of register a
        for i in range(n + 1):
            inverseQFT(qc, result, i, pie)

```

Listing 8: QFT:Function Multiplication

Viene proposto un esempio dell'esecuzione del programma:

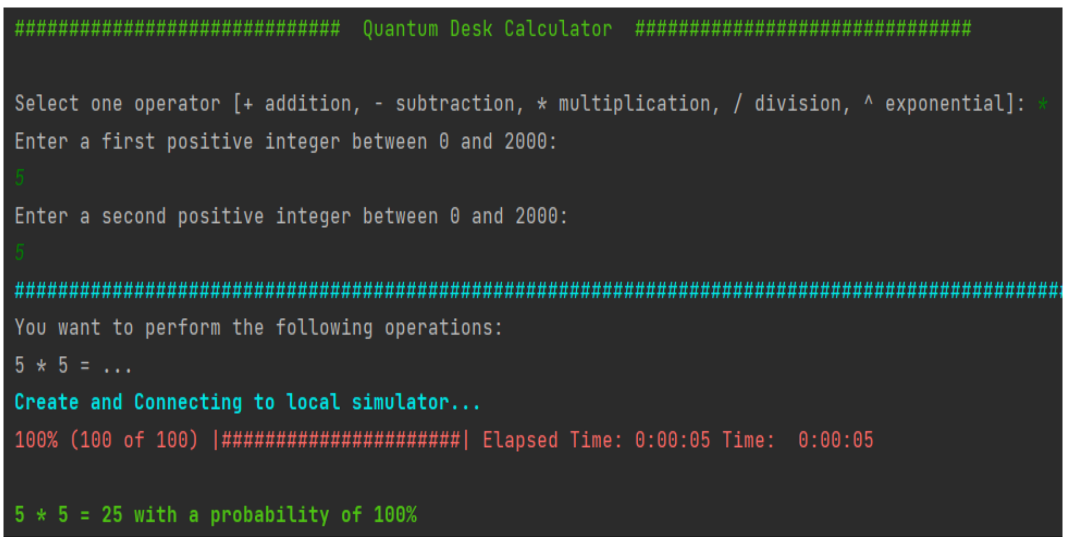


Figura 4: Print Result Function Multiplication

5 Function Division

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, l'algoritmo quindi svolgerà 15-5=10. 10 è il risultato del collasso del circuito, per cui non abbiamo più il circuito.
Per poter svolgere il passo successivo, ovvero 10-5, dobbiamo ricreare il circuito, inserendo ora 10 come dividendo.
Ripeteremo questa operazione fino a quando il dividendo è $\geq$ al divisore.

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):

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

    job = execute(qc, backend=Aer.get_backend('qasm_simulator
    '), shots=100)
```

```python
    # Get results of program
    job_stats = job.result().get_counts()

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

    return tmp, prob
```

Listing 9: QFT:Function 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.

```python
    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")
```

```python
            initQubits(numdividend, qc, dividend, nqubit)
            initQubits(numdivisor, qc, divisor, nqubit)

        else:
            break

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

```

Listing 10: QFT:Function Division

Viene proposto un esempio dell'esecuzione del programma:

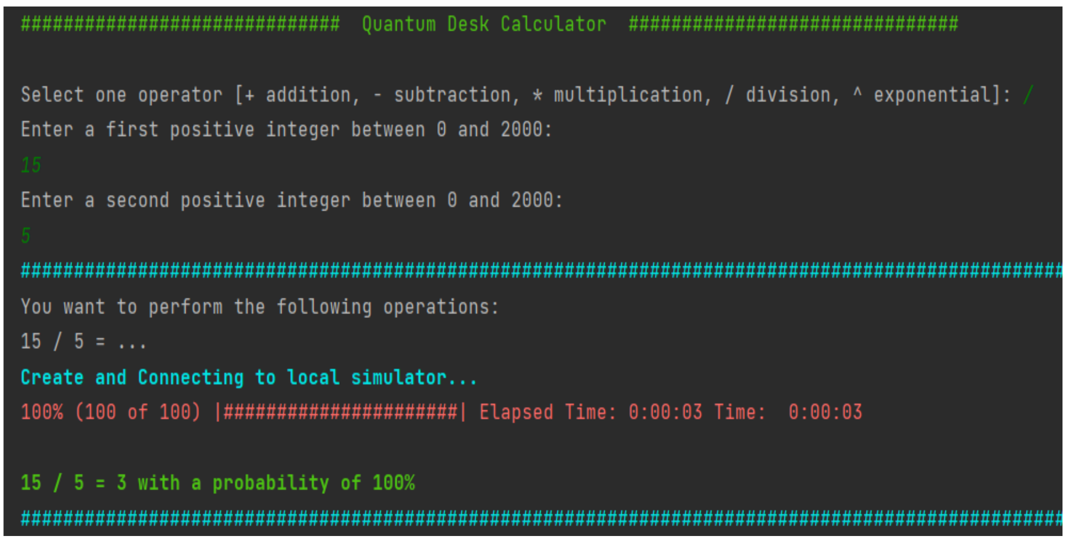


Figura 5: Print Result Function Division

6 Function Exponential

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 altri che una successione di addizioni, ma anche 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)

```

```python
        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:Function Exponential

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 with esponential:
3
################################################################################################
You want to perform the following operations:
5 ^ 3 = ...
Create and Connecting to local simulator...
100% (100 of 100) |####################| Elapsed Time: 0:00:05 Time:  0:00:05

5 ^ 3 = 125 with a probability of 100%
```

Figura 6: Print Result Function Exponential