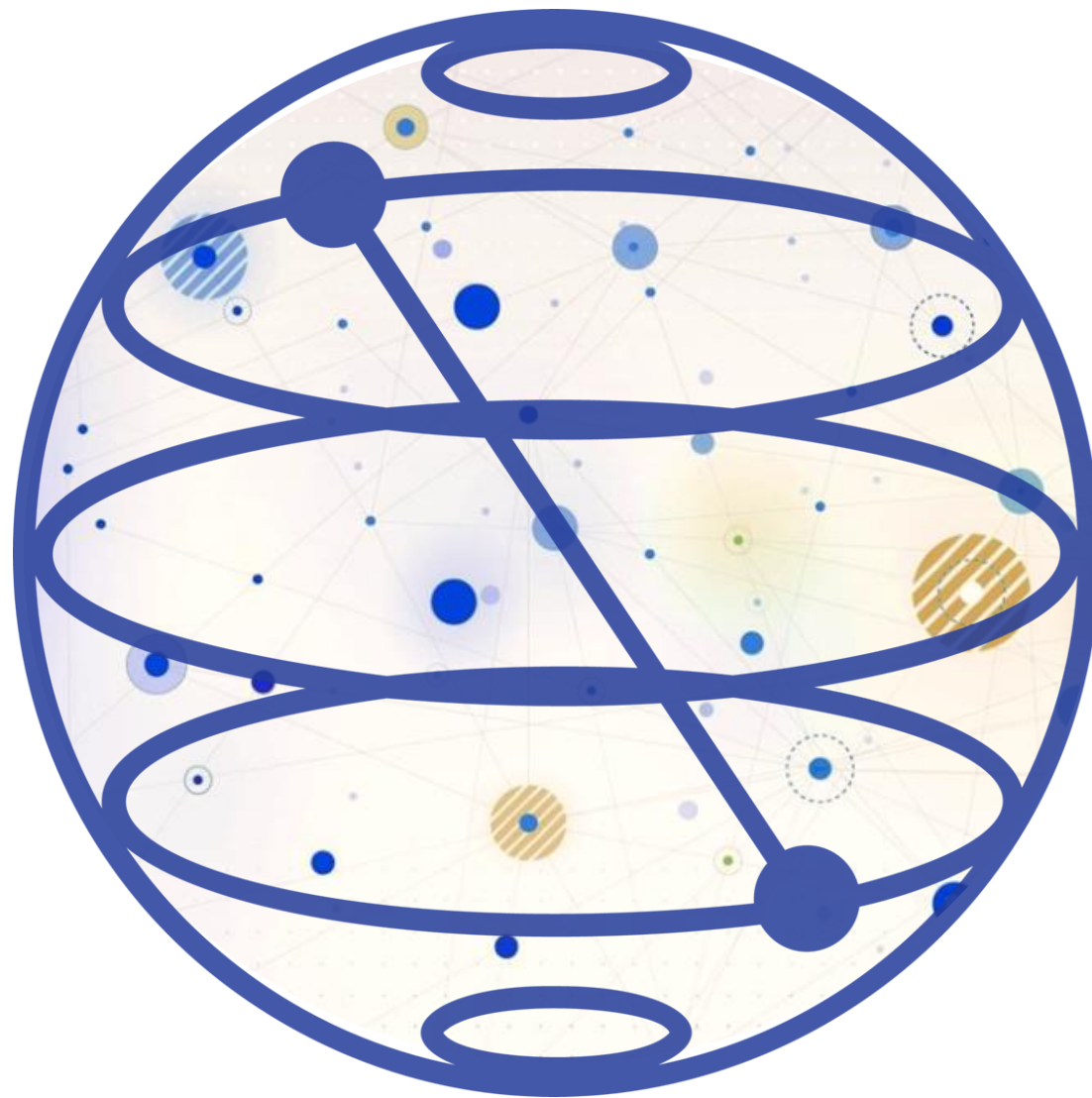


# Corso di Quantum Computing - Giorno 3

Corso per Epigenesys s.r.l.

Docenti: Sara Galatro e Lorenzo Gasparini

Supervisore: Prof. Marco Pedicini



# Funzioni booleane periodiche

---

- Sia  $f$  una funzione booleana in  $\{0,1\}^n$ . Diremo che  $f$  ha **periodo**  $s$  se per ogni  $x, y \in \{0,1\}^n$  abbiamo che:

$$f(x) = f(y) \Leftrightarrow y = x \oplus s \vee x = y.$$

- Se una funzione booleana  $f$  ha periodo  $s$  non nullo  $s \neq 0^n$ , allora  $f$  è sicuramente non iniettiva. In particolare  **$f$  è una funzione 2 a 1**. Ci sono esattamente due valori che hanno la stessa immagine.
- Se una funzione booleana  $f$  ha periodo nullo  $s = 0^n$ , allora la funzione è biunivoca e dunque invertibile.
- Passiamo ora al problema di Simon, fulcro del prossimo algoritmo che vedremo.

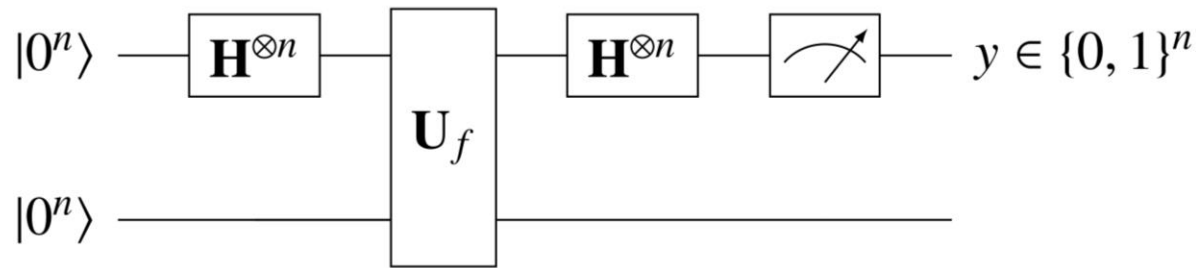
# Algoritmo di Simon

---

- Il **problema di Simon** è così definito. Sia  $f: \{0,1\}^n \rightarrow \{0,1\}^n$  una funzione booleana periodica di periodo  $s$ . Trovare  $s$ .
- L'**algoritmo quantistico di Simon** è un algoritmo di tipo ibrido. Include una parte di calcolo classico ed una subroutine quantistica.
- L'idea alla base dell'algoritmo è quello di iterare più volte la routine quantistica per ottenere ogni volta un'informazione sul periodo  $s$ . Tale informazioni sarà codificata tramite un'equazione lineare in  $s$ .
- Una volta ottenute abbastanza informazioni sul periodo, si passa al post-processing classico per l'ottenimento di  $s$ . Lo si fa avendo a disposizione abbastanza equazioni lineari da poter risolvere il sistema in maniera univoca.
- Vedremo prima tutti gli step dell'algoritmo e ci soffermeremo in seguito sulla subroutine quantistica.

# Algoritmo di Simon

- 1) Inizializziamo un insieme  $E = \emptyset$ .
- 2) Fintanto che  $E$  non ha soluzione unica, eseguiamo la subroutine quantistica tramite il seguente circuito quantistico



L'equazione  $y \cdot s = 0$  viene aggiunta al sistema lineare  $E$

- 3) Risolviamo il sistema lineare  $E$  in  $s$  tramite algoritmo classico, ottenendo così il periodo come soluzione del problema.

# Algoritmo di Simon

- Analizziamo la subroutine quantistica dell'algoritmo, che opera su un registro quantistico di  $2n$  qubit.

- 1) Inizializziamo il circuito nello stato  $|0^n\rangle \otimes |0^n\rangle$ .
- 2) Applichiamo la porta di Hadamard sui primi  $n$  qubit:

$$\left(H^{\otimes n} \otimes \mathbb{I}^{\otimes n}\right)\left(|0^n\rangle \otimes |0^n\rangle\right) = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle |0^n\rangle$$

- 3) Calcoliamo ora la funzione  $f$  in sovrapposizione tramite  $U_f$ :

$$U_f\left(\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle |0^n\rangle\right) = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle |f(x)\rangle$$

# Algoritmo di Simon

---

4) Riappliciamo Hadamard sul primo registro ad  $n$  qubit:

$$(H^{\otimes n} \otimes \mathbb{I}^{\otimes n})s_f = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (H^{\otimes n} |x\rangle) \otimes |f(x)\rangle$$

$$\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} \left( \frac{1}{\sqrt{2^n}} \sum_{y \in \{0,1\}^n} (-1)^{x \cdot y} |y\rangle \right) \otimes |f(x)\rangle = \frac{1}{2^n} \sum_{x,y} (-1)^{x \cdot y} |y\rangle |f(x)\rangle$$

5) Infine misuriamo il primo registro. Otteniamo una stringa  $y \in \{0,1\}^n$  tale che  $y \cdot s = 0$  da aggiungere all'insieme di equazioni  $E$ .

# Algoritmo di Simon

- Il punto fondamentale per capire il funzionamento dell'algoritmo è dimostrare il perché, misurando il primo registro, otteniamo una stringa  $y$  tale che  $y \cdot s = 0$ .
- Calcoliamo la probabilità di ottenere una stringa  $y$ :

$$p(y) = \left\| \frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{x \cdot y} |f(x)\rangle \right\|^2$$

- Evidenziando rispetto all'immagine di  $f$ :

$$p(y) = \left\| \frac{1}{2^n} \sum_{z \in \text{Im}(f)} (-1)^{f^{-1}(z) \cdot y} |z\rangle \right\|^2$$

- Supponendo che  $f$  non sia biunivoca, dobbiamo aggiungere una somma sulle possibili preimmagini di  $z$ .

# Algoritmo di Simon

- Riscriviamo la probabilità come:

$$p(y) = \left\| \frac{1}{2^n} \sum_{z \in \text{Im}(f)} \left( \sum_{x \in f^{-1}(z)} (-1)^{x \bullet y} \right) |z\rangle \right\|^2 = \frac{1}{2^{2n}} \sum_{z \in \text{Im}(f)} \left| \sum_{x \in f^{-1}(z)} (-1)^{x \bullet y} \right|^2$$

- Ricordiamo ora che  $f$  è una funzione due a uno e dunque la preimmagine contiene due valori  $x_1$  e  $x_2$ .

$$\left| \sum_{x \in f^{-1}(z)} (-1)^{x \bullet y} \right|^2 = \left| (-1)^{y \bullet x_1} + (-1)^{y \bullet x_2} \right|^2 = \left| (-1)^{y \bullet x_1} + (-1)^{y \bullet (x_1 \oplus s)} \right|^2 = \begin{cases} 4 & \text{se } y \bullet s = 0 \\ 0 & \text{se } y \bullet s = 1 \end{cases}$$



# Algoritmo di Simon

---

- Sostituendo nell'equazione di prima abbiamo:

$$p(y) = \begin{cases} \frac{1}{2^{2n}} \sum_{z \in Im(f)} 4 = \frac{1}{2^{n-1}} & \text{se } y \bullet s = 0 \\ \frac{1}{2^{2n}} \sum_{z \in Im(f)} 0 = 0 & \text{se } y \bullet s = 1 \end{cases}$$

- Questo dimostra il fatto che misurando il primo registro otteniamo sicuramente, con probabilità uniforme, una stringa  $y$  tale che  $y \cdot s = 0$ .
- La subroutine quantistica ci permette dunque di ottenere un'equazione lineare in  $s$ . Naturalmente avremo bisogno di iterare il circuito fino a che non otteniamo  $n - 1$  equazioni indipendenti così da poter univocamente calcolare il periodo.
- La domanda che ci poniamo è: quante volte è necessario iterare il processo in media per arrivare ad una soluzione?

# Algoritmo di Simon

---

- Analizzando l'algoritmo nel modello query, l'algoritmo di Simon trova  $s$  in media con un numero polinomiale di chiamate all'oracolo  $U_f$  (iterazioni del circuito).
- La dimostrazione è puramente probabilistica e si basa sul dimostrare la probabilità di ottenere vettori linearmente indipendenti, pescando in maniera uniforme nell'insieme  $\{0,1\}^n$ .
- Si ragiona ricorsivamente, sapendo che i vettori indipendenti da  $x_1, \dots, x_i$  sono in tutto  $2^n - 2^i$  e quindi la probabilità di ottenerne uno indipendente è pari a  $\frac{2^n - 2^i}{2^n}$ .
- Si riesce a stimare che la probabilità che l'algoritmo fallisca dopo  $2k$  iterazioni è al massimo

$$\frac{1}{2^{2k}} = \frac{1}{4^k} < e^{-k}.$$

- Dunque **sono necessarie in media un numero polinomiale query gate.**

# Complessità quantistica

---

- Affronteremo ora in maniera più approfondita un modello computazionale che mette in gioco il costo dell'implementazione dei gate quantistici.
- Parleremo di **universalità** legato al concetto di operazione elementare, così da poter analizzare il numero di **gate elementari** presenti in un circuito.
- Infine presenteremo alcune **classi di complessità computazionali quantistiche** e le metteremo a paragone con quelle classiche.
- Abbandoniamo il modello query e vedremo precisamente come si costruisce un gate  $U_f$  tramite la **simulazione di circuiti booleani**.

# Universalità di gate

---

- Ricordiamo che qualsiasi gate può essere costruito da altri tramite composizione sequenziale e in parallelo.
- Si definisce un insieme di gate elementari che permettono di costruire, tramite le regole di sopra, ogni altro gate quantistico. Un insieme di gate che soddisfa tale proprietà, viene detto **universale**.
- Gli insiemi universali sono alla base del **costo computazionale** di un circuito quantistico.
- Quando si esegue un conto dei gate utilizzati, si deve fare riferimento ad uno specifico insieme di gate universale. Ogni gate nell'insieme considerato, nell'analisi computazionale, viene assunto di costo unitario.
- Presenteremo adesso alcuni dei set universali più standard usati in ambito computazionale.

# Universalità di gate

---

- L'insieme formato da **tutti i gate unari con il controlled-NOT** formano un insieme universale. Dal punto di vista pratico risulta poco utile. Sia perché ce ne sono infiniti e sia per la difficoltà nell'implementazione perfetta di ognuno di essi.
- L'insieme contenente solo i gate **controlled-NOT, Hadamard, Phase-shift con  $\alpha = \frac{\pi}{4}$** , è un insieme universale per approssimazione.
- L'insieme contenente solamente **Hadamard e Toffoli** è un insieme universale per approssimazione ristretto ai soli operatori reali.
- Precisato l'insieme universale di riferimento si può passare a parlare di costo computazionale.

# Costo computazionale

---

- Faremo riferimento alla **dimensione del circuito** come metro di misurazione del costo computazionale di un circuito.
- In generale si possono considerare altre metriche, come la profondità di un circuito. Nel nostro caso ci soffermeremo solo sul conto totale di gate elementari presenti.
- Sposteremo l'attenzione al **costo computazionale** in riferimento a come aumenta all'aumentare della dimensione dell'input.
- Parleremo di una famiglia di circuiti  $Q_n$  ognuno dei quali agisce su un input di dimensione  $n$ . Ci riferiremo al costo computazionale con la funzione  $t(n)$ .
- Analizzeremo dunque la funzione costo in riferimento al suo **comportamento asintotico**.

# Costo computazionale

---

- Trattando la complessità nel contesto asintotico risulta di fondamentale importanza la **notazione  $O$** . Tale notazione è stata adottata per mettere in evidenza, nella funzione costo, il termine dominante.
- Siano  $g(n)$  e  $h(n)$  due funzioni. Diremo che  $g = O(h)$  se esiste una costante  $c$  ed un intero  $n_0$  tale che, per ogni  $n \geq n_0$ , si ha

$$g(n) \leq c \cdot h(n).$$

- Naturalmente si tende a scegliere  $h$  nella forma più semplice. Per esempio se  $g(n) = 2n^2 + 17$ , allora  $g = O(n^2)$ . In generale, per i polinomi, faremo riferimento al termine di grado più alto.
- Se  $t(n) = O(n^d)$  per qualche  $d$ , diremo che  $Q_n$  ha un **costo computazionale polinomiale** e il circuito viene considerato asintoticamente efficiente. Se invece  $t(n) = O(2^n)$ , diremo che  $Q_n$  ha **costo computazionale esponenziale** e viene considerato non efficiente.

# Classi di complessità

---

- Un problema di decisione computazionale corrisponde a quello che viene chiamato **linguaggio**, ovvero un sottoinsieme delle stringhe binarie  $L \subset \{0,1\}^*$ .
- Un problema è associato ad una sequenza di funzioni booleane  $f_n: \{0,1\}^n \rightarrow \{0,1\}$ . La funzione  $f_n$  vale 1 nelle stringhe in input che sono in  $L$ .
- Una **classe computazionale** è un insieme di problemi di decisione che hanno tutti la stessa complessità computazionale asintotica.
- Enunceremo le principali classi computazionali sia nel caso classico che nel caso quantistico.



# Classi di complessità

---

- **P (Polynomial Time)**: insieme di problemi risolvibili da un computer classico deterministico in tempo polinomiale  $O(n^d)$ .
- **BPP (Bounded-error Probabilistic Polynomial Time)**: insieme di problemi risolvibili da un computer classico in tempo polinomiale con probabilità d'errore minore di  $\frac{1}{3}$ .
- **NP (Non-deterministic Polynomial Time)**: classe di problemi, per cui data una stringa  $x$ , si verifica in tempo polinomiale se tale  $x$  è soluzione.
- **NP-Complete**: sottoclasse di NP contenente tutti quei problemi a cui ogni altro problema in NP si riconduce. L'esempio chiave di problema NPC è il problema **SAT** (determinare se una certa formula booleana è soddisfacibile).
- **PSPACE (Polynomial Space)**: classe di problemi risolvibili da un computer classico usando spazio polinomiale nella lunghezza dell'input.

# Classi di complessità

---

- **BQP (Bounded-error Quantum Polynomial Time)**: classe di problemi risolvibili da un computer quantistico in tempo polinomiale con probabilità di errore  $\leq \frac{1}{3}$ . Tale classe è riconosciuta come la classe di problemi efficientemente risolvibili con un computer quantistico.
- **QMA (Quantum Merlin-Arthur)**: classe di problemi  $L$  per cui, quando una stringa  $x$  è in  $L$ , esiste una **quantum proof** di dimensione polinomiale che convince un **quantum verifier** sul fatto che  $x \in L$  con alta probabilità.
- **QMA-Complete**: sottoclasse contenente tutti quei problemi a cui ogni problema in QMA si riduce. L'esempio più famoso, strettamente legato al problema SAT, è il problema dell'Hamiltoniana locale.

# Complessità quantistica

---

- Riportiamo di seguito alcuni risultati che mettono a paragone alcune classi complessità.

$$P \subseteq NP$$

Inclusione ovvia e nonostante non ci sia ancora una dimostrazione si crede che l'inclusione debba essere stretta.

$$BPP \subseteq BQP$$

Partendo dal fatto che un computer quantistico può simulare un circuito classico e può generare azioni probabilistiche tramite degli Hadamard gate. Si crede che l'inclusione sia stretta.

$$BQP \subseteq PSPACE$$

Si dimostra facendo vedere che un computer classico può simulare efficientemente un computer quantistico in termini di spazio (non di tempo).

$$NP \subseteq QMA$$

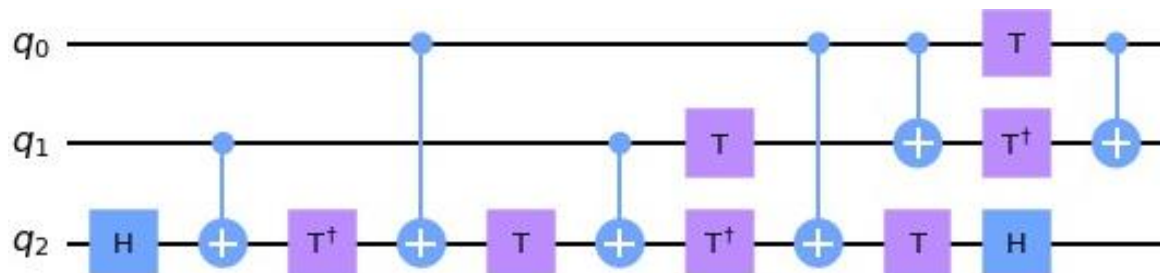
Si crede anche qui che esistano problemi in QMA ma non in NP.

# Simulazione di circuiti classici

---

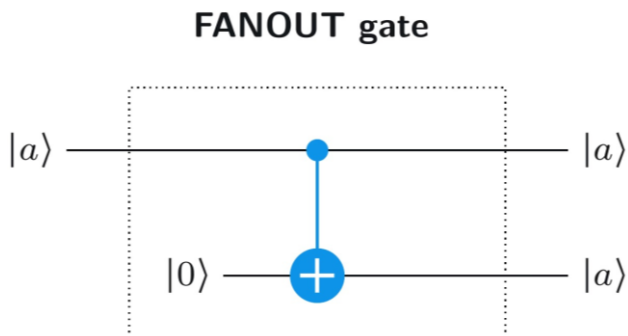
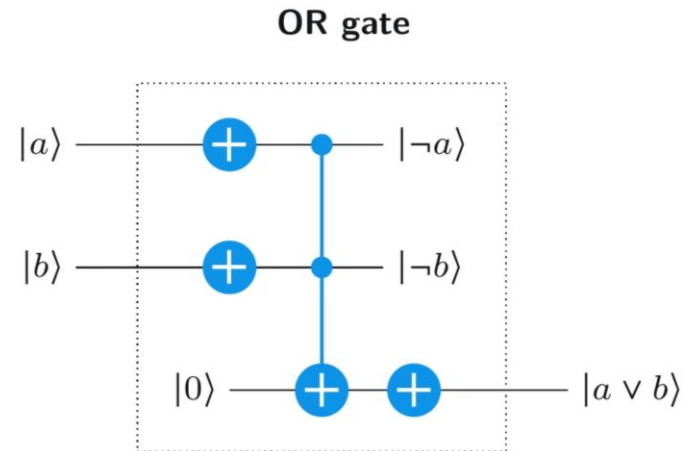
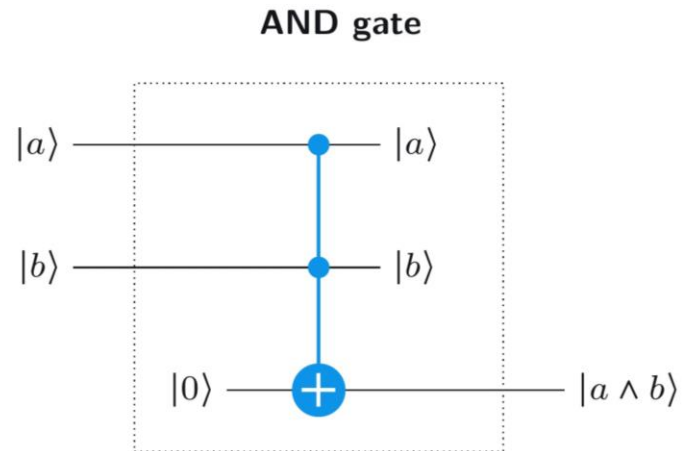
- Passeremo ora all'analisi dell'implementazione di circuiti booleani su un computer quantistico.
- Ogni calcolo classico può essere eseguito su un computer quantistico con lo stesso costo computazionale asintoticamente.
- Per poter simulare un qualsiasi circuito booleano, dobbiamo essere in grado di **simulare i gate logici classici** alla base di ogni circuito.
- I circuiti booleani sono formati da **AND, OR, NOT** e **FANOUT** gate.
- Mostreremo dunque i 4 gate quantistici che simulano tali 4 gate classici. Faremo uso, per la simulazione, dei soli gate **NOT, CNOT, CCNOT**. Sono tutte e 3 operazioni deterministiche e unitarie.

- Ricordiamo che il gate **Toffoli** agisce come segue su un circuito a 3 qubit:



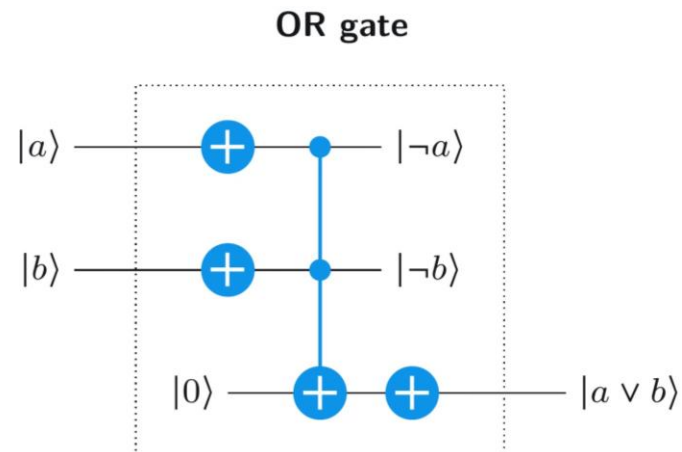
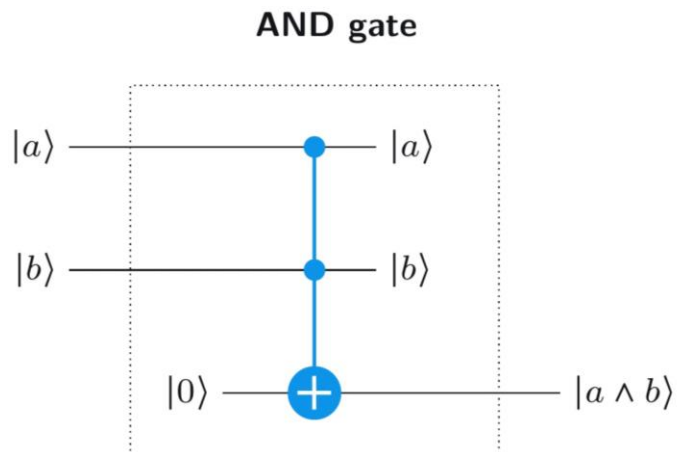
# Simulazione di circuiti classici

- Mostriamo ora come possiamo simulare i gate classici AND, OR, FANOUT (il NOT classico è già implementato con il gate  $X$ ) usando i gate  $X, CX, CCX$ .



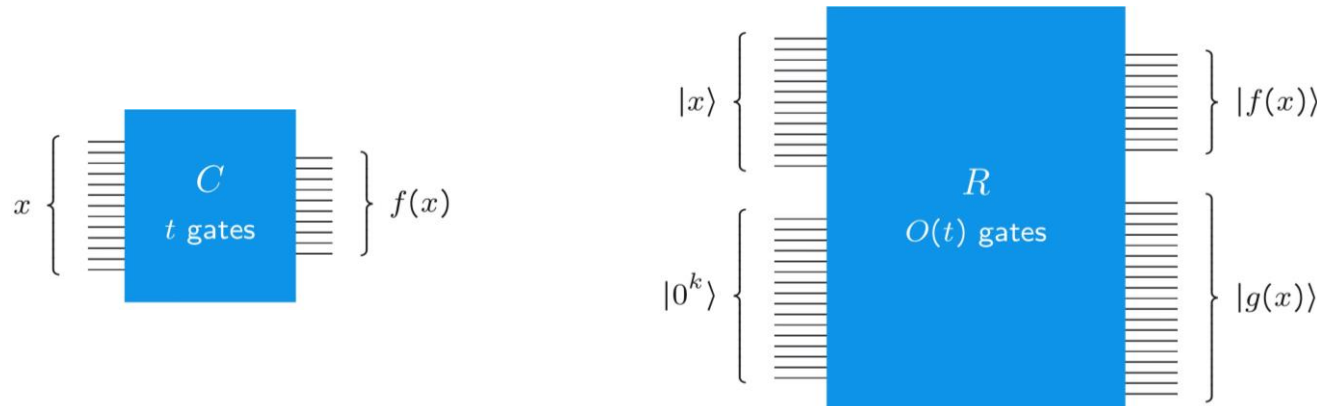
# Simulazione di circuiti classici

- Per tutti e 3 i gate simulati è stato necessario l'utilizzo di un **qubit workspace, in cui viene salvato il risultato**.
- Per l'AND e l'OR i qubit che vengono passati come input al circuito vengono poi ignorati dato che non sono output del circuito logico booleano che stiamo simulando.
- Tali qubit vengono detti **qubit ancilla**.



# Simulazioni di circuiti classici

- Supponiamo quindi di avere una funzione  $f: \{0,1\}^n \rightarrow \{0,1\}^m$  implementata tramite un circuito  $C$  che fa uso di  $t$  gate classici (AND, OR, NOT, FANOUT).
- **Simuliamo gate per gate** il circuito  $C$  secondo quanto descritto prima includendo i qubit per workspace in un circuito quantistico  $R$ . Sono necessari quindi  $O(t)$  **gate quantistici**.
- I qubit di  $R$  sono ordinati in modo che gli  $n$  bit di input del circuito  $C$  sono sul registro superiore, mentre i qubit di workspace sono sul registro inferiore. Possiamo però semplicemente **scambiare i due registri usando degli  $SWAP$  gate**.





# Esercizi

---

- Costruire tramite circuito a due qubit uno *SWAP* gate facendo uso di pochi *CNOT* gate.
- Costruire tramite un circuito a 3 qubit, la funzione XOR tra 3 qubit in input e l'output salvato sul primo qubit:

$$|a\rangle|b\rangle|c\rangle \mapsto |a \oplus b \oplus c\rangle|\dots\rangle|\dots\rangle.$$

- Provare a generalizzare spiegando in che modo si possa costruire un circuito che implementi lo XOR tra due stringhe binarie  $a, b \in \{0,1\}^n$ :

$$|a\rangle|b\rangle \mapsto |a \oplus b\rangle|\dots\rangle.$$

# Simulazioni di circuiti classici

---

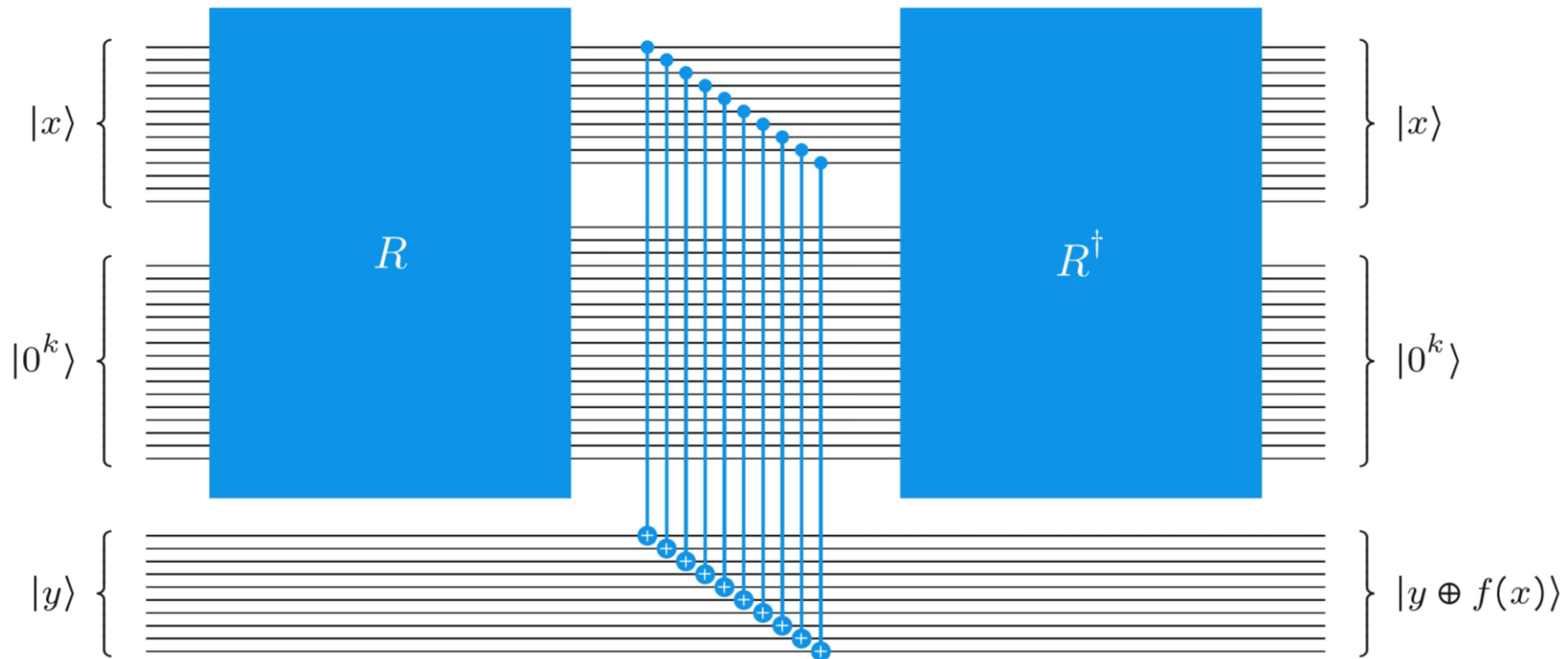
- $k$  è il numero di qubit di workspace che sono necessari e  $g: \{0,1\}^n \rightarrow \{0,1\}^{n+k-m}$  è la funzione che descrive in che stato vengono lasciati i qubit ausiliari dopo l'esecuzione di  $R$ .
- In generale  $g$  è una funzione determinata dal circuito  $C$ , ma è una cosa che ignoriamo dato che sono qubit ausiliari.
- La funzione  $g$  descrive il cosiddetto **garbage**, il quale è stato riordinato sul secondo registro per un motivo che vedremo a breve.
- Se siamo solo interessati al calcolo della funzione  $f$  la costruzione gate per gate è sufficiente, in cui ci sono alcuni qubit che rimangono inutilizzati.
- Ci dedichiamo ora invece ad un modo per ripulire questo garbage.

# Simulazione di circuiti classici

---

- Se volessimo implementare calcoli classici come subroutine all'interno di computazioni quantistiche più grandi è necessario **ripulire il garbage**, così che i qubit del garbage non disturbino eventuali fenomeni di interferenza alla base del funzionamento di molti algoritmi quantistici.
- Per fare ciò faremo uso di un **ulteriore registro quantistico ad  $m$  qubit** in cui verrà copiato l'output  $|f(x)\rangle$  dopo l'esecuzione di  $R$ . Questo verrà eseguito tramite dei  $CNOT$  i quali implementano uno XOR.
- Si **applica a questo punto il circuito  $R$  invertito** così da poter resettare il registro in input  $|x\rangle$  e il registro workspace  $|0^k\rangle$ . Notiamo che, dato che  $R$  è formato da Toffoli,  $CNOT$  e  $NOT$ , il suo inverso  $R^\dagger$  è  $R$  stesso.

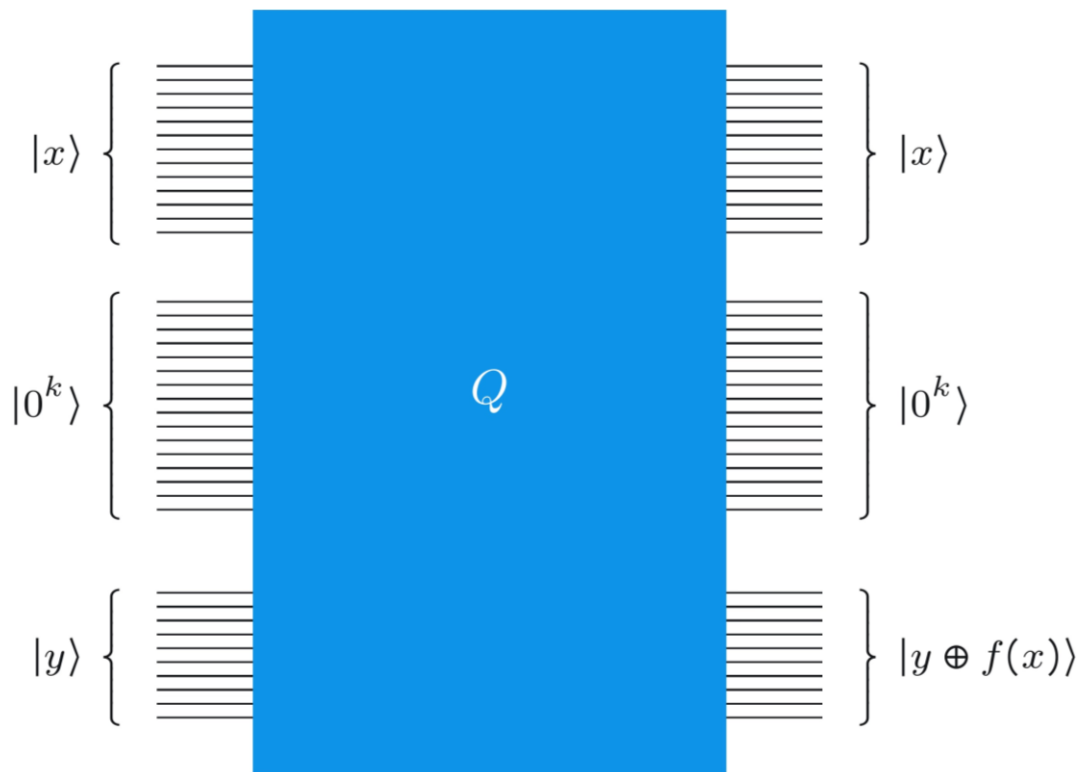
# Simulazione di circuiti classici



- Questa tecnica viene spesso detta **compute-copy-uncompute**.

# Simulazione di circuiti classici

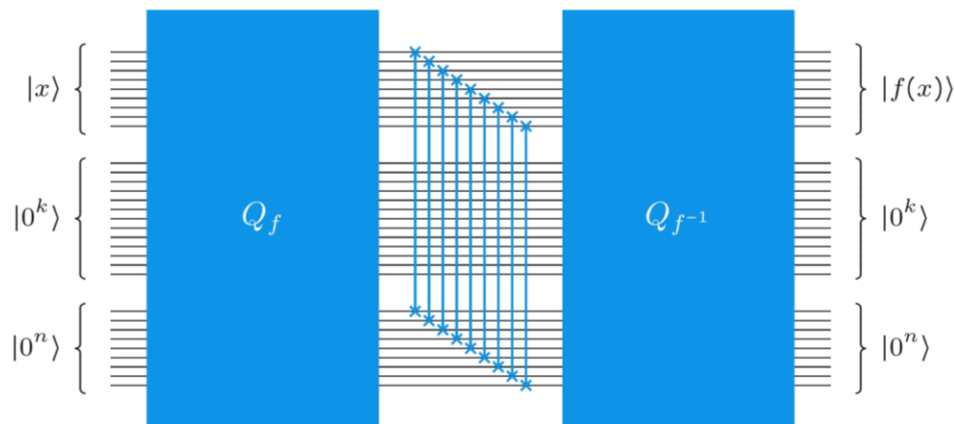
- Possiamo raggruppare tutte queste operazioni in un unico circuito  $Q$ .



- Se  $C$  è formato da  $t$  gate classici allora  $Q$  è **formato da  $O(t)$  gate quantistici elementari** il che vuol dire che la costruzione è efficiente.
- Non contando il registro workspace nel circuito, il circuito  $Q$  **implementa a tutti gli effetti il query gate  $U_f$** .
- Inizializzando l'ultimo registro  $|y\rangle$  a  $|0^m\rangle$  abbiamo in output sull'ultimo registro  $|f(x)\rangle$  la valutazione della funzione  $f$  sull'input  $x$ .

# Simulazione di circuiti classici

- Analizziamo un ultimo caso in cui  $f$  stessa è invertibile. Ciò vuol dire che l'operazione che trasforma  $|x\rangle$  in  $|f(x)\rangle$  è unitaria.
- Vogliamo quindi costruire un circuito che implementi l'operazione  $U|x\rangle = |f(x)\rangle$ .
- Tralasciando il workspace,  $U$  è diversa dalle operazioni che vengono implementate nel circuito  $Q$  dato che qui non dobbiamo salvare l'output in una copia in XOR con una stringa arbitraria ma trasformiamo direttamente  $x$  in  $f(x)$ . Possiamo costruire questa tecnica facendo uso dei circuiti che implementano  $f$  e  $f^{-1}$ .



# Quantum Fourier Transform

---

- La **Quantum Fourier Transform (QFT)** è una delle operazioni fondamentali nell'ambito della computazione quantistica. Risulta essere uno strumento alla base di molti algoritmi quantistici importanti come Shor, Quantum Phase Estimation e Hidden Subgroup Problem.
- La QFT è l'equivalente della DFT ed agisce sugli stati della base standard ad  $n$  qubit nel seguente modo:

$$QFT |x\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \omega_N^{xj} |j\rangle$$

- La QFT applica una trasformazione ad uno stato quantistico  $|x\rangle$  mappandolo in una **sovrapposizione uniforme di stati  $|j\rangle$  con fasi diverse**.

# Quantum Fourier Transform

- Vediamo anche la QFT nella sua **forma matriciale** come operatore unitario. Possiamo specificare le entrate della matrice così:

$$QFT_N = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} e^{\frac{2\pi i j k}{N}} |j\rangle \langle k|$$

- Specificando che nell'entrata  $(j, k)$  avremo la radice  $N$ -esima dell'unità  $e^{\frac{2\pi i j}{N}}$  elevata alla  $k$ . **Esplicitando la matrice** abbiamo:

$$QFT_N = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix}$$



# Quantum Fourier Transform

- Le colonne della QFT formano una base ortonormale, detta **base di Fourier**. Ogni vettore in questa base è una sovrapposizione degli stati della base standard. In particolare indichiamo con  $|\tilde{j}\rangle$  lo stato ottenuto dallo stato standard  $|j\rangle$  dopo l'applicazione della QFT:

$$QFT |j\rangle = |\tilde{j}\rangle$$

- Le ampiezze relative a  $|\tilde{j}\rangle$  seguono le potenze di  $\omega_N^j$ .
- Usando la base di Fourier, lo stato  $|\tilde{j}\rangle$  ci indica di quale angolo ogni qubit deve essere ruotato rispetto all'asse Z, dopo che si trova in una sovrapposizione  $|+\rangle$ .
- Possiamo comprendere meglio questo concetto osservando che  $|\tilde{j}\rangle$  è scomponibile come prodotto tensore di più qubit in sovrapposizione:

$$QFT_N |j\rangle = \frac{1}{\sqrt{N}} \bigotimes_{k=1}^n \left( |0\rangle + e^{\frac{2\pi i j k}{2^k}} |1\rangle \right)$$

# Esempio QFT

- Prediamo la QFT su due qubit. L'operatore è il seguente:

$$QFT_4 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

- Vogliamo scomporre  $|\tilde{3}\rangle$ . Facciamo il calcolo matriciale a partire dall'applicazione della matrice:

$$\begin{aligned} |\tilde{3}\rangle &= QFT_4 |3\rangle = \frac{1}{2} (|00\rangle - i|01\rangle - |10\rangle + i|11\rangle) = \\ &= \frac{1}{2} (|0\rangle (|0\rangle - i|1\rangle) - |1\rangle (|0\rangle - i|1\rangle)) = \\ &= \frac{1}{2} (|0\rangle - i|1\rangle) \otimes (|0\rangle - |1\rangle) \end{aligned}$$

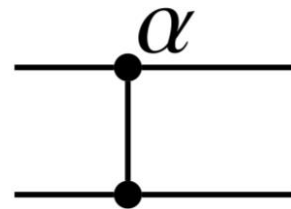
Entrambi i qubit coinvolti nello stato prodotto sono stati in sovrapposizione  $|+\rangle$  ruotati rispetto all'asse Z rispettivamente di un angolo  $\frac{3}{2}\pi$  e  $\pi$ .

# Implementazione QFT

- Vediamo ora in che modo implementare la Quantum Fourier Transform tramite gate elementari e verifichiamo che la si può costruire efficientemente.
- Faremo ampiamente uso della versione controllata del phase-gate  $P_\alpha$  definita come:

$$CP_\alpha = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\alpha} \end{bmatrix}$$

- Rappresenteremo un cambio di fase controllato nel seguente modo

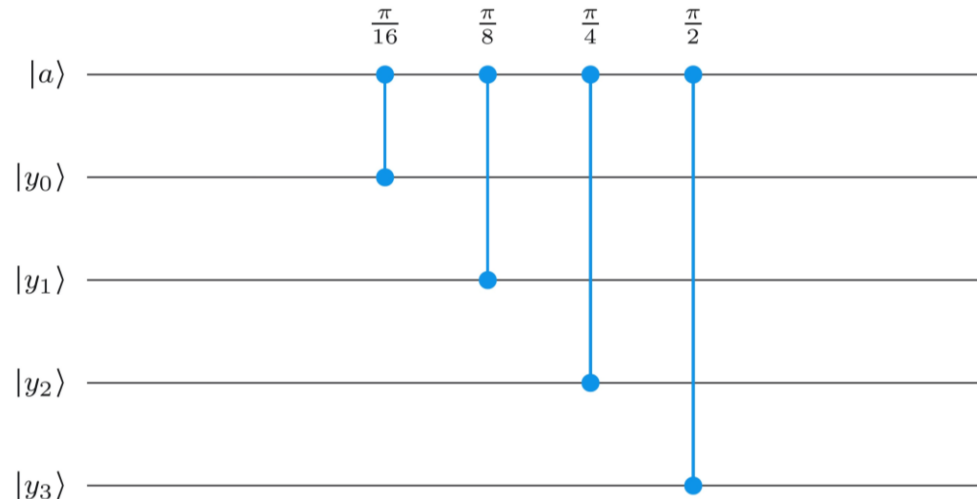


# Implementazione QFT

- Facendo uso dei gate  $CP_\alpha$  riusciamo ad implementare una phase injection così definita:

$$|a\rangle |y\rangle \mapsto \omega_{2^n}^{ay} |a\rangle |y\rangle$$

- Un'operazione di questo tipo viene costruita utilizzando vari  $CP_\alpha$  con  $|a\rangle$  come qubit di controllo e  $|y_i\rangle$  come qubit target per ogni  $i = 0, \dots, n - 1$ . In particolare se il qubit target è  $|y_i\rangle$ , l'angolo  $\alpha = \frac{\pi}{2^{n-1}} \cdot 2^i$ . Per esempio con  $n = 5$  avremmo la seguente costruzione:



# Implementazione QFT

- Applicando questo procedimento, se  $a = 0$ , lo stato  $|a\rangle|y\rangle$  viene mappato in se stesso. Se  $a = 1$  invece abbiamo l'applicazione di tutti i gate  $P_\alpha$ :

$$|1\rangle |y_0\rangle \dots |y_{n-1}\rangle \mapsto |1\rangle \otimes P_{\frac{\pi}{2^{n-1}}} |y_0\rangle \otimes \dots \otimes P_{\frac{\pi}{2}} |y_{n-1}\rangle$$

$$|1\rangle \otimes e^{i\frac{\pi}{2^{n-1}}y_0} |y_0\rangle \otimes \dots \otimes e^{i\frac{\pi}{2}y_{n-1}} |y_{n-1}\rangle$$

$$e^{i\frac{\pi}{2^{n-1}}y_0 + \dots + i\frac{\pi}{2}y_{n-1}} |1\rangle |y_0\rangle \dots |y_{n-1}\rangle = e^{i\frac{\pi}{2^{n-1}}(y_0 + 2y_1 + \dots + 2^{n-1}y_{n-1})} |1\rangle |y\rangle = \omega_{2^n}^y |1\rangle |y\rangle$$

# Implementazione QFT

- Procediamo a questo punto con i passi per eseguire la QFT su un registro di  $n$  qubit. L'implementazione seguirà una naturale definizione ricorsiva partendo dalla base  $QFT_2$  che non è altro che Hadamard.
- Eseguiamo la QFT su  $n$  qubit descrivendo la sua azione sugli stati della base  $|a\rangle|x\rangle$ .

1) Applichiamo la  $QFT_{2^{n-1}}$  agli  $n - 1$  qubit descritti dallo stato  $|x\rangle$ :

$$|a\rangle \left( QFT_{2^{n-1}} |x\rangle \right) = \frac{1}{\sqrt{2^{n-1}}} \sum_{y=0}^{2^{n-1}-1} \omega_{2^{n-1}}^{xy} |a\rangle |y\rangle$$

2) Applichiamo ora il circuito della phase-injection visto prima:

$$\frac{1}{\sqrt{2^{n-1}}} \sum_{y=0}^{2^{n-1}-1} \omega_{2^{n-1}}^{xy} \omega_{2^n}^{ay} |a\rangle |y\rangle$$

# Implementazione QFT

3) Applichiamo una porta di Hadamard sul primo qubit descritto da  $|a\rangle$ :

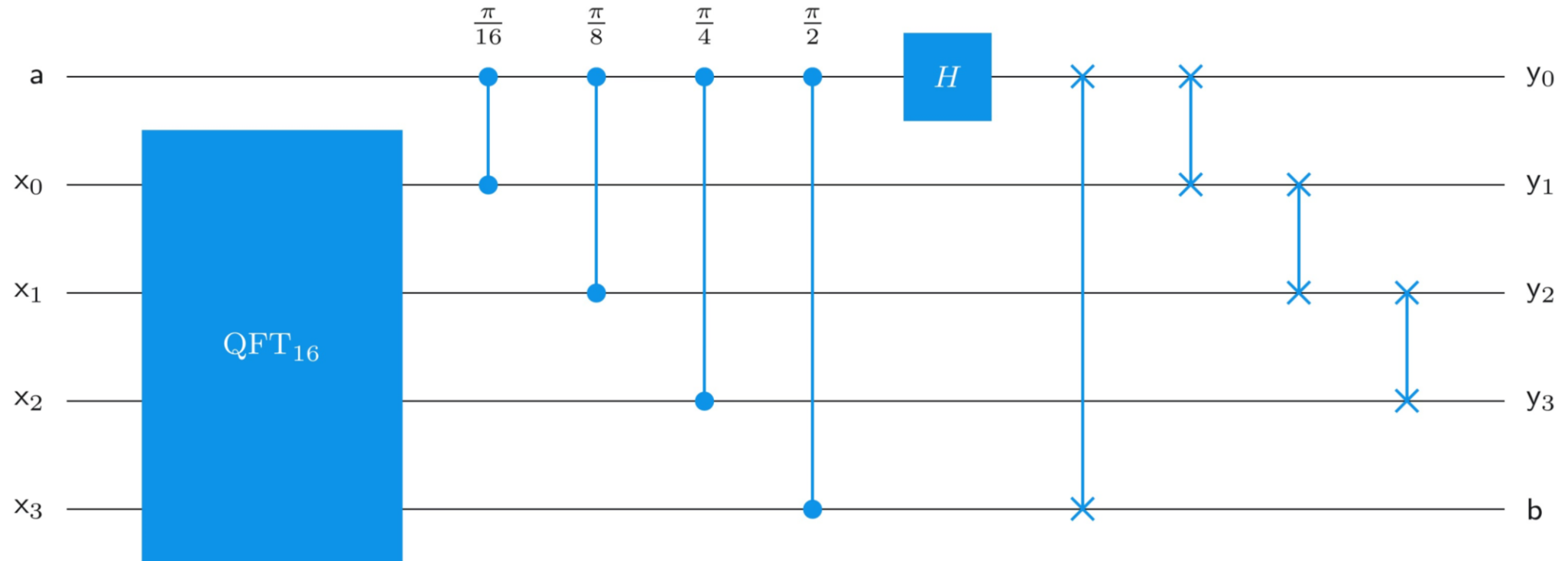
$$\frac{1}{\sqrt{2^n}} \sum_{b=0}^1 \sum_{y=0}^{2^{n-1}-1} (-1)^{ab} \omega_{2^{n-1}}^{xy} \omega_{2^n}^{ay} |b\rangle |y\rangle$$

4) Infine permutiamo l'ordine dei qubit, in modo che il bit  $a$  finisca alla fine, tramite l'utilizzo di gate *SWAP* ripetuti, facendo ruotare la stringa  $by$  di un posto verso sinistra per ottenere  $yb$ :

$$\frac{1}{\sqrt{2^n}} \sum_{b=0}^1 \sum_{y=0}^{2^{n-1}-1} (-1)^{ab} \omega_{2^{n-1}}^{xy} \omega_{2^n}^{ay} |y\rangle |b\rangle$$

# Implementazione QFT

- Riportiamo di seguito un esempio della costruzione della QFT su  $n = 5$  qubit.





# Analisi computazionale QFT

---

- Contiamo il numero di gate elementari che vengono usati nel circuito. Specifichiamo che i gate  $CP_\alpha$  non fanno parte dell'insieme universale di gate, ma per semplicità sui conti lo considereremo come gate elementare.
- Denotiamo con  $t_n$  il numero di gate applicati nel circuito a seconda del numero di qubit  $n$
- Se  $n = 1$  ovviamente  $QFT_2 = H$ , quindi ha costo unitario e  $t_1 = 1$ .
- Consideriamo ora  $n \geq 2$ , necessiteremo allora di:
  - $t_{n-1}$  gate per implementare (ricorsivamente)  $QFT_{2^{n-1}}$
  - $n - 1$  controlled-phase gate  $CP_\alpha$
  - Una singola porta Hadamard  $H$
  - $n - 1$  gate  $SWAP$

# Analisi computazionale QFT

---

- Riassumendo, il numero totale di gate è dato da

$$t_n = t_{n-1} + (2n - 1) = \sum_{k=0}^n (2k - 1) = n^2$$

- Questo dimostra il fatto che la QFT può essere implementata con  $n^2$  gate.
- Per la precisione dovremmo specificare la costruzione  $CP_\alpha$  tramite gate elementari. Ciò dipende dal tipo di precisione desiderata. In ogni caso si può implementare  $CP_\alpha$  in maniera efficiente con pochi gate.
- Concludiamo che la  $QFT_N$  è efficientemente implementabile con l'utilizzo di  $O(n^2)$  gate quantistici elementari.

# L'algoritmo di Grover



# Formulazione del problema



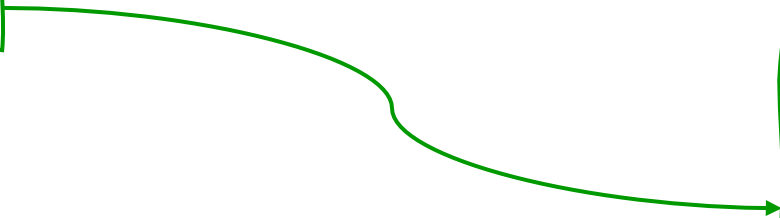
- Supponiamo di avere una **funzione binaria**  $f: \Sigma^n \rightarrow \Sigma$ , dove  $\Sigma = \{0,1\}$ , che sappiamo di poter calcolare efficientemente e che non abbia una struttura specifica da poter sfruttare.
- Grover cercherà una stringa  $x \in \Sigma^n$  tale che  $f(x) = 1$ . Ci riferiremo a tale stringa come a una **soluzione** al nostro problema di ricerca.
- Il problema associato a Grover è detto **ricerca non strutturata**, dato che non possiamo fare affidamento a delle proprietà specifiche di  $f$  o della nostra struttura dati.
- Nel **peggior caso classico** (sia deterministico che probabilistico), in uno spazio con  $|\Sigma^n| = 2^n =: N$  elementi, dovremmo valutare la nostra funzione su ognuno di essi, portando il nostro algoritmo classico ad avere un costo dell'ordine di  $O(N)$ .
- L'algoritmo di Grover sfrutta le sovrapposizioni e le interferenze quantistiche per ridurre il numero di iterazioni all'ordine  $O(\sqrt{N})$ , trovando una soluzione con alta probabilità.

# Una prima applicazione: funzioni hash



- Per sua definizione, l'algoritmo di Grover garantisce uno speedup negli algoritmi di ricerca a **forza bruta**, che possiamo sfruttare in particolare per attacchi a crittosistemi a chiave simmetrica, come ad esempio le funzioni hash.
- Una **funzione hash crittografica**  $h$  è una funzione che prende in input una stringa di lunghezza arbitraria, detta **messaggio**, e restituisce in output una stringa di lunghezza fissata, chiamata **digest**, che soddisfa le seguenti proprietà:

a) **Uniformità**



Gli output di  $h$  devono simulare randomicità, così da non fornire informazioni sull'input.

# Una prima applicazione: funzioni hash

- Per sua definizione, l'algoritmo di Grover garantisce uno speedup negli algoritmi di ricerca a **forza bruta**, che possiamo sfruttare in particolare per attacchi a crittosistemi a chiave simmetrica, come ad esempio le funzioni hash.
- Una **funzione hash crittografica  $h$**  è una funzione che prende in input una stringa di lunghezza arbitraria, detta **messaggio**, e restituisce in output una stringa di lunghezza fissata, chiamata **digest**, che soddisfa le seguenti proprietà:

a) **Uniformità**

b) **Determinismo**

Dato un input,  $h$  deve sempre restituire sempre lo stesso digest.

# Una prima applicazione: funzioni hash

- Per sua definizione, l'algoritmo di Grover garantisce uno speedup negli algoritmi di ricerca a **forza bruta**, che possiamo sfruttare in particolare per attacchi a crittosistemi a chiave simmetrica, come ad esempio le funzioni hash.
- Una **funzione hash crittografica  $h$**  è una funzione che prende in input una stringa di lunghezza arbitraria, detta **messaggio**, e restituisce in output una stringa di lunghezza fissata, chiamata **digest**, che soddisfa le seguenti proprietà:

- a) **Uniformità**
- b) **Determinismo**
- c) **Irreversibilità**

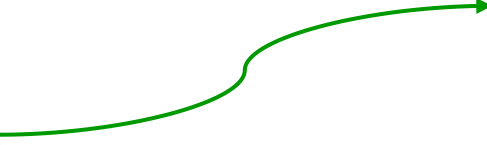
Dato un qualsiasi digest, non deve essere possibile invertire  $h$  e ottenere l'input.



# Una prima applicazione: funzioni hash



- Per sua definizione, l'algoritmo di Grover garantisce uno speedup negli algoritmi di ricerca a **forza bruta**, che possiamo sfruttare in particolare per attacchi a crittosistemi a chiave simmetrica, come ad esempio le funzioni hash.
- Una **funzione hash crittografica**  $h$  è una funzione che prende in input una stringa di lunghezza arbitraria, detta **messaggio**, e restituisce in output una stringa di lunghezza fissata, chiamata **digest**, che soddisfa le seguenti proprietà:
  - a) **Uniformità**
  - b) **Determinismo**
  - c) **Irreversibilità**
  - d) **Quasi-iniettività**



Vogliamo simulare una funzione iniettiva sfruttando la differenza di cardinalità tra dominio e codominio e definendo una funzione altamente variabile.



# Una prima applicazione: funzioni hash



- Date queste caratteristiche, le funzioni hash sono molto utilizzate per garantire l'autenticità e l'integrità dei servizi associati.
- Tale sicurezza è data dalla resistenza a due tipi di attacchi:
  - a) **Pre-image Resistance**: dato  $y = h(x)$ , deve essere computazionalmente difficile trovare  $x$ : l'unica possibilità è una ricerca di forza bruta.
  - b) **Collision Resistance** : data  $h$ , deve essere computazionalmente difficile trovare  $x_1, x_2$  tali che  $h(x_1) = h(x_2)$ .
- Se ad esempio, prendessimo una funzione hash sicura a **256 bit**...

# Impostazione dell'algoritmo

- Supponiamo di avere  $k$  soluzioni tra gli  $N$  elementi che formano il nostro spazio.
- Nel nostro spazio di Hilbert, definiamo uno **stato «fortunato»**  $|w\rangle \in \mathbb{C}^N$  le cui entrate sono tali che

$$w_x = \frac{1}{\sqrt{k}} \cdot \mathbf{1}_{\{x \text{ è soluzione}\}} + 0 \cdot \mathbf{1}_{\{x \text{ non è soluzione}\}}$$

- L'obiettivo dell'algoritmo di Grover sarà di avvicinare il più possibile lo stato aleatorio di partenza al nostro vettore fortunato, sfruttandone la struttura di **solution-smoothness**.
- Altri vettori che godono di tale proprietà sono i vettori nello  $\text{Span}(|w\rangle, |s\rangle)$  e lo **stato «fallimento»**  $|l\rangle$  definito come

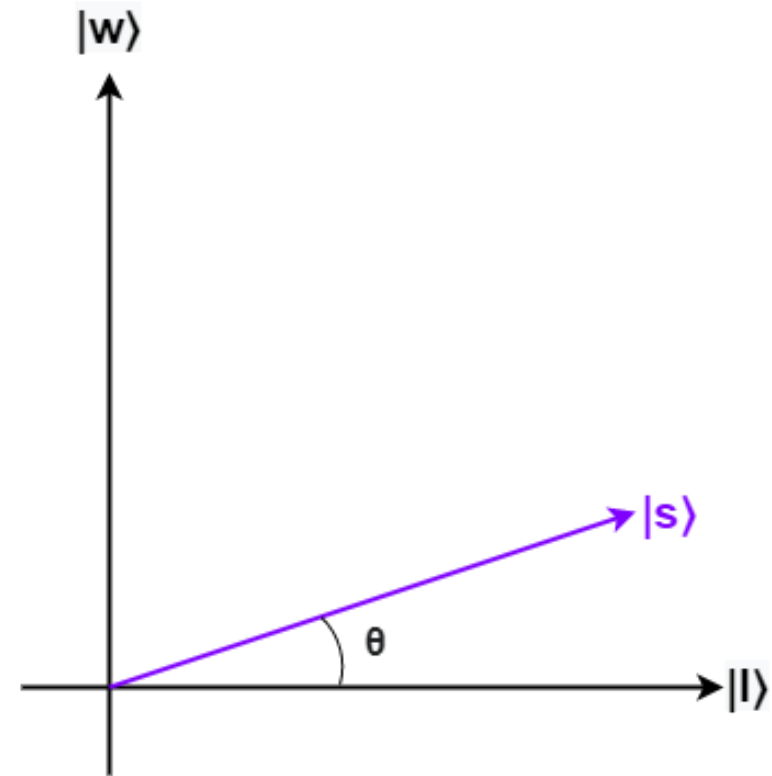
$$|l\rangle = \frac{1}{\sqrt{N-k}} (\sqrt{N} |s\rangle - \sqrt{k} |w\rangle)$$

# Impostazione dell'algoritmo

- L'algoritmo di Grover opera tramite **riflessioni**: una rispetto al vettore  $|l\rangle$  e una rispetto al vettore calcolato all'iterazione precedente  $|a\rangle$ .
- La riflessione rispetto a  $|l\rangle$  è effettuata tramite **oracolo di Grover**:

$$U_f[x, x] = (-1)^{f(x)} = \\ = (-1) \cdot \mathbf{1}_{\{x \text{ è soluzione}\}} + 1 \cdot \mathbf{1}_{\{x \text{ non è soluzione}\}}$$

- L'idea è di immaginare i nostri stati sul **piano generato da  $|w\rangle$  e  $|l\rangle$** , associandoli rispettivamente all'asse delle ordinate e delle ascisse.



# Impostazione dell'algoritmo

- In questo modo,  $|s\rangle$  si troverà a un angolo  $\theta \in \left[0, \frac{\pi}{2}\right]$ , da cui otteniamo che

$$\cos \theta = \frac{\langle s | l \rangle}{\| |s\rangle \| \cdot \| |l\rangle \|} = \sqrt{\frac{N - k}{N}}$$

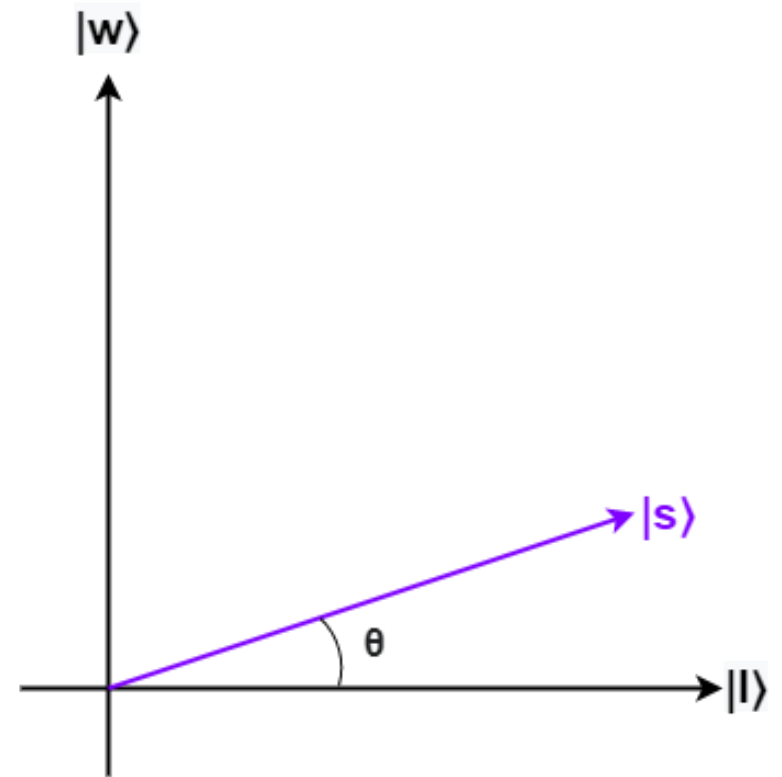
ossia

$$\sin^2 \theta = \frac{k}{N}$$

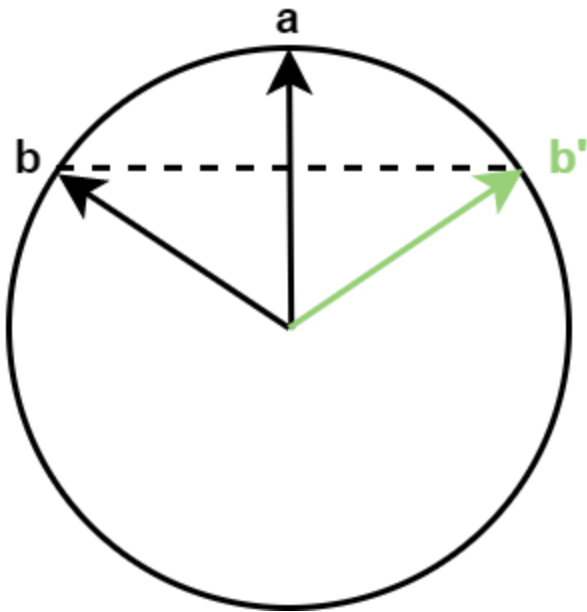
- In generale, possiamo sempre scrivere

$$|\psi\rangle = \cos \theta |l\rangle + \sin \theta |w\rangle$$

ottenendo che la probabilità di misurare una soluzione sia  $\sin^2 \theta_i$ .



# Focus on: Riflessioni e Rotazioni



- Matematicamente, le riflessioni sono definite via proiezioni.

- Dati due vettori  $\mathbf{a}, \mathbf{b}$  la **proiezione** di  $\mathbf{b}$  su  $\mathbf{a}$  è data da
$$\mathbf{a}' = \mathbf{a}\langle \mathbf{a}, \mathbf{b} \rangle$$

- Dunque, la riflessione rispetto ad  $\mathbf{a}$  di  $\mathbf{b}$  è data da

$$\mathbf{b}' = 2(\mathbf{b} - \mathbf{a}') = \mathbf{b} - 2\mathbf{b} + 2\mathbf{a}\langle \mathbf{a}, \mathbf{b} \rangle = 2(\mathbf{P}_a - \mathbb{I})\mathbf{b}$$

dove  $\mathbf{P}_a = \mathbf{a}^T \mathbf{a} = |a\rangle\langle a|$  è **l'operatore proiezione**.

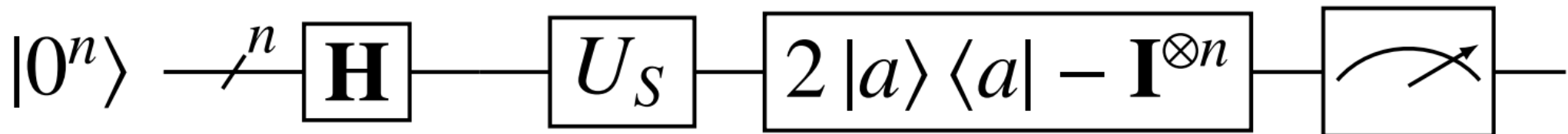
# L'algoritmo



- 1) Inizializzare lo stato  $|0^n\rangle$  come una sovrapposizione equiprobabile  $|a\rangle := |s\rangle$ ;
- 2) Calcolare  $\theta = \arcsin\left(\sqrt{\frac{k}{N}}\right)$  e  $t_k = \left\lfloor \frac{\pi}{4\theta} \right\rfloor$ ;
- 3) Ripetere per  $t_k$  volte le seguenti operazioni:
  - i. Applica  $Ref_w$  allo stato  $|a\rangle$  tramite oracolo di Grover tramite  $U_f$  per ottenere lo stato  $|a'\rangle$ ;
  - ii. Applica  $Ref_a = 2|a\rangle\langle a| - \mathbb{I}^{\otimes n}$  allo stato  $|a'\rangle$  per ottenere un nuovo stato  $|a\rangle$ ;
- 4) Misura lo stato finale  $|a\rangle$ , ottenendo una stringa  $x \in \{0,1\}^n$ ;
- 5) Se  $x$  è soluzione, **stop**; altrimenti riparti dal punto 1 a causa del collasso del sistema.

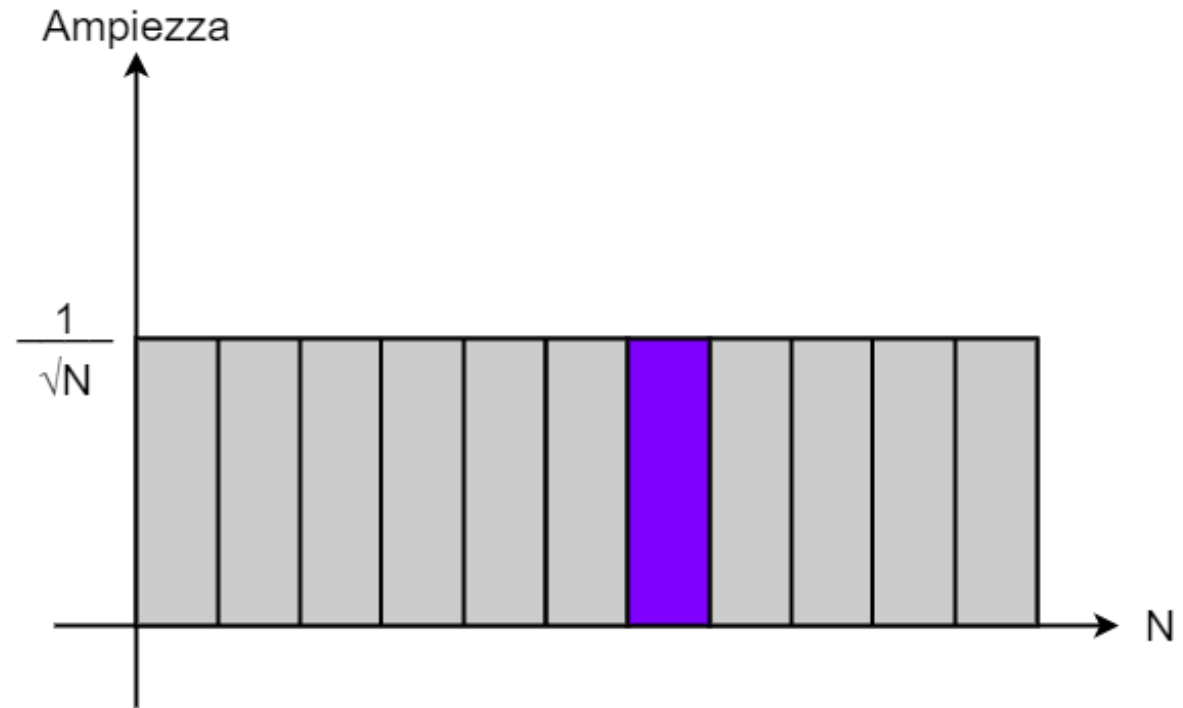
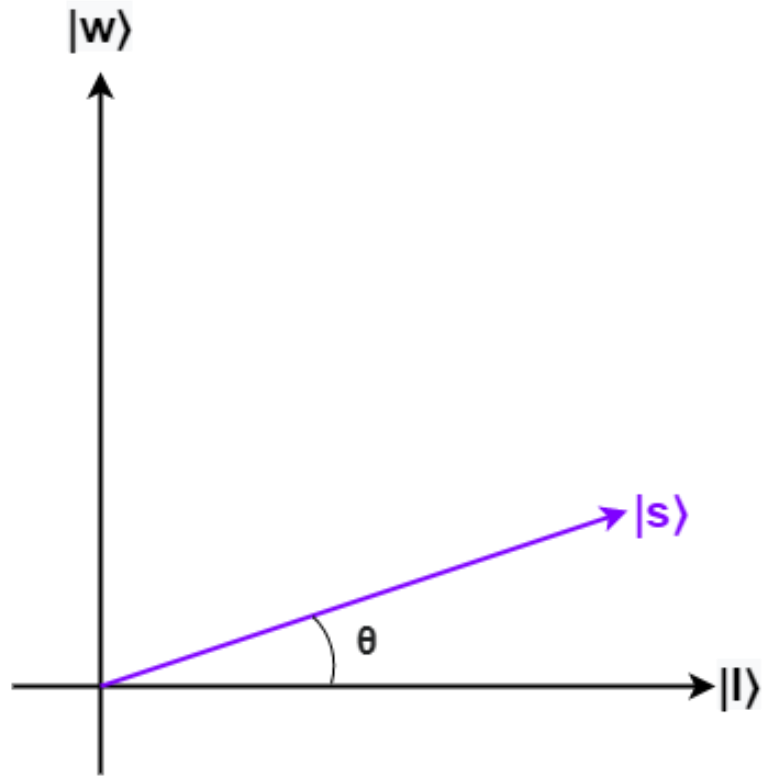
# L'algoritmo

- 1) Inizializzare lo stato  $|0^n\rangle$  come una sovrapposizione equiprobabile  $|a\rangle := |s\rangle$ ;
- 2) Calcolare  $\theta = \arcsin\left(\sqrt{\frac{k}{N}}\right)$  e  $t_k = \left\lfloor \frac{\pi}{4\theta} \right\rfloor$ ;
- 3) Ripetere per  $t_k$  volte le seguenti operazioni:
  - i. Applica  $Ref_w$  allo stato  $|a\rangle$  tramite oracolo di Grover tramite  $U_f$  per ottenere lo stato  $|a'\rangle$ ;
  - ii. Applica  $Ref_a = 2|a\rangle\langle a| - \mathbb{I}^{\otimes n}$  allo stato  $|a'\rangle$  per ottenere un nuovo stato  $|a\rangle$ ;
- 4) Misura lo stato finale  $|a\rangle$ , ottenendo una stringa  $x \in \{0,1\}^n$ ;
- 5) Se  $x$  è soluzione, **stop**; altrimenti riparti dal punto 1 a causa del collasso del sistema.



# L'algoritmo

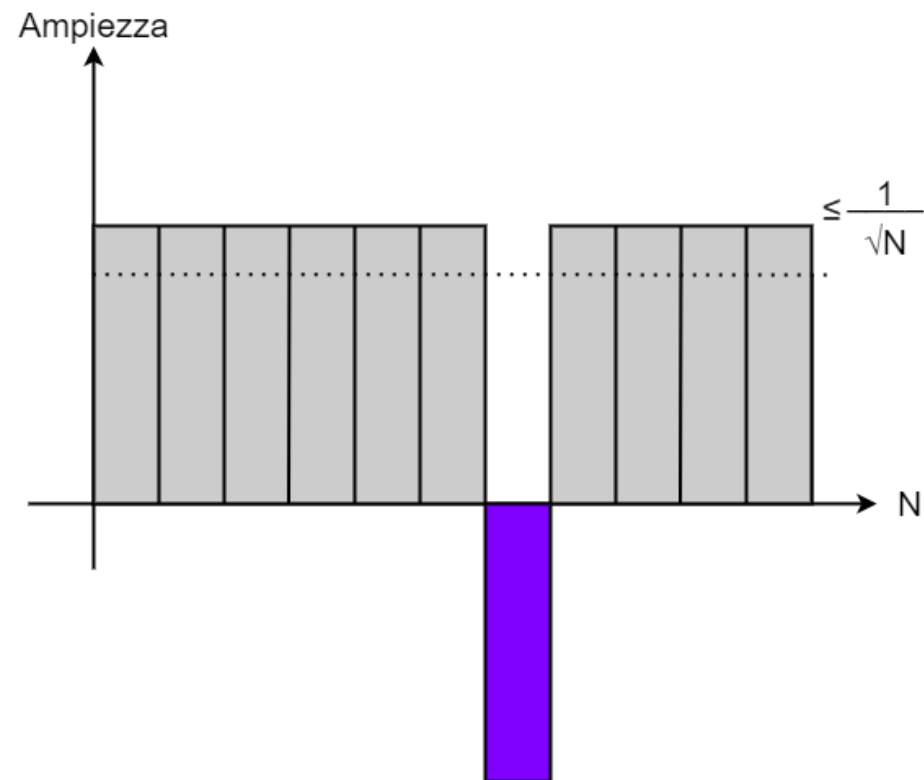
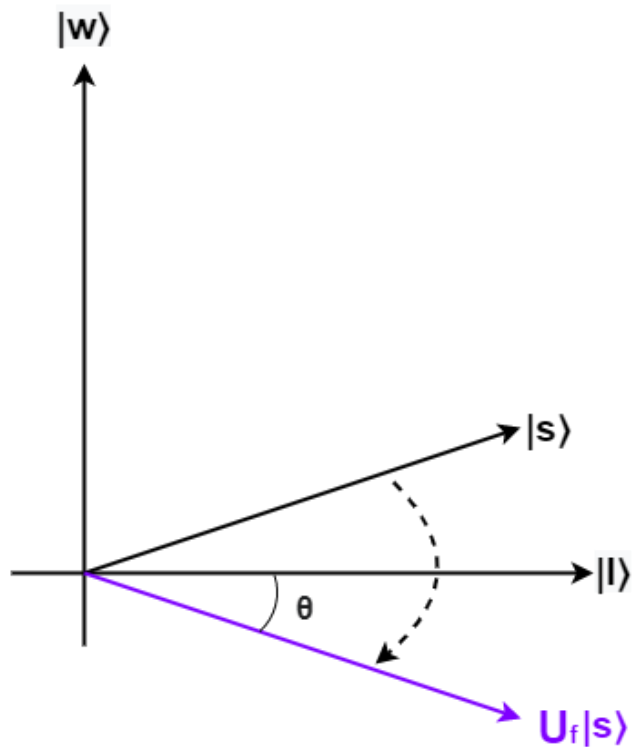
- 1) Inizializzare lo stato  $|0^n\rangle$  come una sovrapposizione equiprobabile  $|a\rangle := |s\rangle$ ;





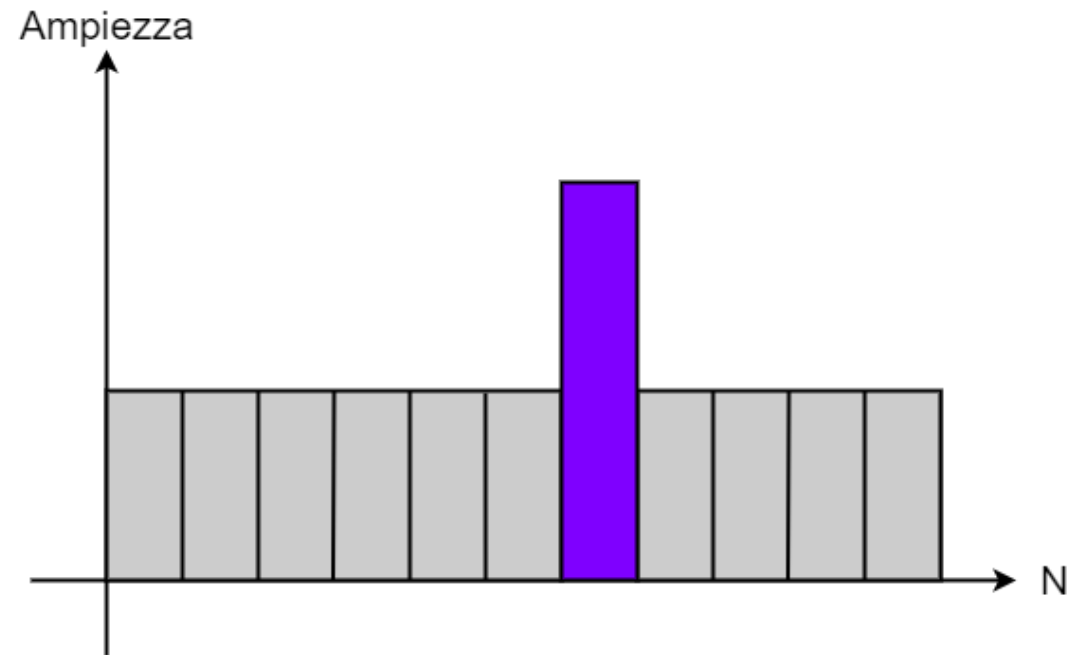
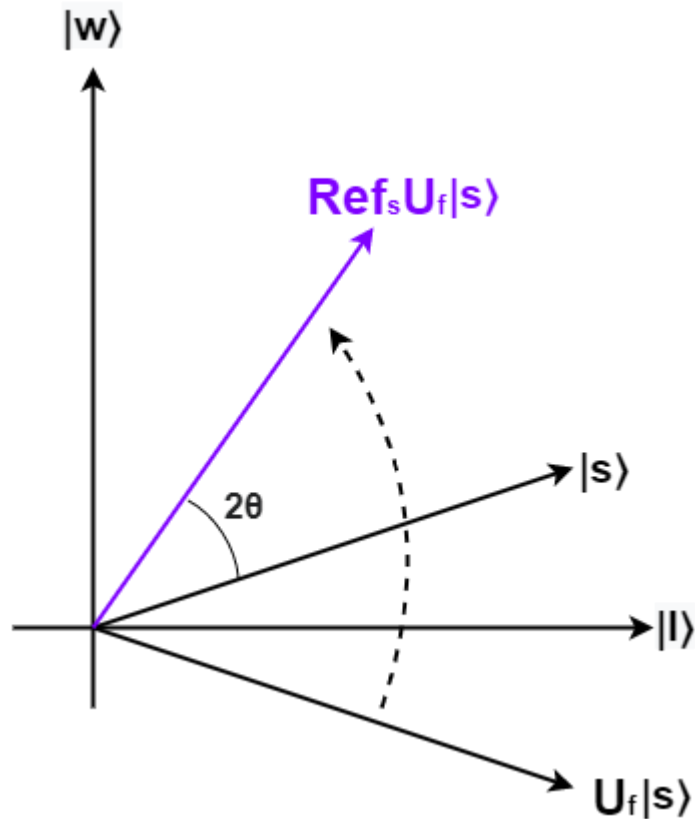
# L'algoritmo

3.i) Applica  $Ref_w$  allo stato  $|a\rangle$  tramite oracolo di Grover tramite  $U_f$  per ottenere lo stato  $|a'\rangle$ ;



# L'algoritmo

3.ii) Applica  $Ref_a = 2|a\rangle\langle a| - \mathbb{I}^{\otimes n}$  allo stato  $|a'\rangle$  per ottenere un nuovo stato  $|a\rangle$ ;



# Analisi dell'algoritmo

- Alla base del corretto funzionamento dell'algoritmo c'è un'importante proprietà geometrica: **la composizione di due riflessioni**  $Ref_{AC}, Ref_{AB}$  per linee non parallele **è una rotazione** rispetto al punto di intersezione delle due linee pari al doppio dell'angolo formato da esse, ossia

$$Ref_{AC} \circ Ref_{AB} = Rot_{A, 2\angle BAC}$$

- Partiamo dunque da un angolo iniziale pari a  $\theta$  e sia  $\theta_t \in [\theta, \frac{\pi}{2}]$  l'angolo alla  $t$ -esima iterazione. **All'iterazione  $t + 1$**  l'angolo sarà

$$\theta_{t+1} = -\theta_t + 2\theta + 2\theta_t = \theta_t + 2\theta$$

- **Dopo  $t_k$  iterazioni** avremmo un angolo pari a

$$\theta_{t_k} = (2t_k + 1)\theta$$

e vogliamo che valga  $\frac{\pi}{2}$ .

# Analisi dell'algoritmo



- Dopo  $t_k$  iterazioni avremmo un angolo pari a

$$\theta_{t_k} = (2t_k + 1)\theta$$

e vogliamo che valga  $\frac{\pi}{2}$ .

- Risolvendo tale equazione, otteniamo che

$$t_k = \left\lfloor \frac{\pi}{4\theta} - \frac{1}{2} \right\rfloor = \left\lfloor \frac{\pi}{4\theta} \right\rfloor \approx \frac{\pi}{4} \sqrt{\frac{N}{k}}$$

- In questo modo, l'angolo finale si troverà nel range  $\frac{\pi}{2} \pm \theta$  e, ricordando che  $\theta \leq \pi/4$ , otteniamo che **la probabilità di successo sarà almeno pari a  $\frac{1}{2}$  dopo  $t_k$  iterazioni.**

# Numero di iterazioni: caso $k$ sconosciuto

- Supponiamo ora di avere  $k$  soluzioni, ma di non conoscere esplicitamente tale valore.
- Una **prima opzione** per scegliere il numero di iterazioni  $t$  potrebbe essere di sceglierlo uniformemente a caso tra  $1, \dots, \left\lfloor \frac{\pi\sqrt{N}}{4} \right\rfloor$ 
  - In questo modo l'algoritmo ha una probabilità di successo maggiore del 40%. Ripetendo l'algoritmo  $m$  volte nel caso di fallimento, la probabilità sale abbastanza velocemente e si avvicina ad 1.
- Una **strategia più raffinata** è la seguente: si sceglie  $t$  uniformemente a caso in  $\{1, \dots, T\}$ , con  $T$  che aumenta ad ogni iterazione.
  - Il miglior modo per incrementare  $T$  è aggiornarlo come  $\left\lceil \frac{5}{4}T \right\rceil$  ad ogni iterazione. In questo modo, anche non conoscendo il numero di soluzioni, otteniamo che l'algoritmo riesce a trovare una soluzione dopo sole  $O\left(\sqrt{\frac{N}{k}}\right)$  iterazioni.

# Numero di iterazioni: caso $k=0,N$



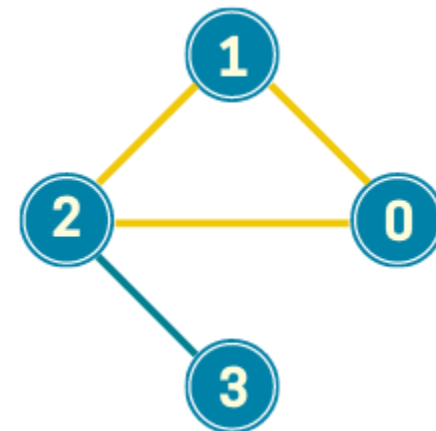
- Matematicamente, ciò equivale ad avere  **$f$  costante** sul nostro spazio, ossia  $|l\rangle$  o  $|w\rangle$  con tutte le entrate nulle.
- Ciò vuol dire anche che  $U_f|a\rangle = \pm|a\rangle$  e che quindi

$$\text{Grover}|a\rangle = \pm(2|a\rangle\langle a| - \mathbb{I})|a\rangle = \pm|a\rangle$$

Dunque, a prescindere del numero di iterazioni che eseguiamo, l'algoritmo restituirà sempre una **stringa scelta uniformemente a caso** in  $\Sigma^n$ .

# Ottimizzazione dell'algoritmo

- Nell'inizializzare lo stato del nostro algoritmo, abbiamo implicitamente assunto **di non sapere nulla** sulle nostre soluzioni.
- Alcune volte però saremo a conoscenza di alcune caratteristiche o proprietà delle nostre soluzioni, che quindi possiamo sfruttare.
- Altre possibili inizializzazioni, a patto che siano sempre solution-smooth, sono lo stato  $|GHZ\rangle$  o lo stato  $|W\rangle$  generalizzati, oppure uno degli **stati di Dicke**  $|Dnk\rangle$ .
- Ad esempio, se volessimo cercare i **triangoli in un grafo con 4 nodi**...



# Mani alle tastiere!

Possiamo passare all'implementazione dell'algoritmo, concludendo poi con alcuni esempi applicativi relativi a problemi di reale interesse.

