Luís Soares Barbosa

Departamento de Informática da Escola de Engenharia
Universidade do Minho

Quantum Computing

Algorithms and Computability

Universidade do Minho, February, 2018

Abstract

This document contains the summaries of an introductory module on background concepts (sets, orders and groups) and computability. The module is part of the Quantum Computation curricular unit included in the syllabus of 4th year of MIEFis (MSc on Engineering Physics), offered at the University of Minho.

Lecture 1: Sets, Orders, Groups

Summary.
(1) Sets, functions, relations. Isomorphism and cardinality.

(2) Ordered structures: preorders, partial orders, lattices. Complete lattices. Ideals and filters. Boolean algebras. Application: semantics of recursive definitions. The theorem of Knaster-Tarski. Lattices as algebraic structures.

(3) Algebra and order. Reversibility. Groups as a prototypical algebraic structure. Groups of permutations. Action of a group. Application: Cayley theorem.

---

1 Sets, relations, cardinality

- Set, function, composition, isomorphism.
- Powerset ($\mathbf{2}^{A}$); partition.
- Binary relations; $\mathbf{2}^{A\times B} \cong \mathbf{2}^{A^{B}}$
- Equivalence relations. Quotient set as a partition.

Finite and infinite sets.

- equicardinality and isomorphism.
- finite *vs* infinite
- countable *vs* uncountable

Theorem
The union of a finite number of countably infinite sets is countably infinite.

$$\begin{tikzcd}
X & x_0 \arrow[d] & x_1 \arrow[d] & x_3 \arrow[d] & \cdots \\
Y & y_0 \arrow[d] & y_1 \arrow[d] & y_3 \arrow[d] & \cdots \\
Z & z_0 \arrow[uur] & z_1 \arrow[uur] & z_3 \arrow[uur] & \cdots
\end{tikzcd}$$

Does the argument generalise to the union of a *countably infinite* number of ...? E.g., is $\mathbb{N} \times \mathbb{N}$ countably infinite?

The diagonalization principle.

For a binary relation R, define the diagonal set $D = \{a \in A \mid (a,a) \notin R\}$. Then, forall $a \in A$ sets D and $R_a = \{b \in A \mid (a,b) \in R\}$ are distinct.

Theorem

$2^{\mathbb{N}}$ is uncountable.

Proof.
If this is not the case, and $2^{\mathbb{N}}$ is countably infinite, there is an enumeration of sets such that

$$2^{\mathbb{N}} = \{R_1, R_2, \cdots\}$$

Let $D = \{n \in \mathbb{N} \mid n \notin R_n\}$. Set D is a set of natural numbers and thus should appear somewhere in the enumeration $R_1, R_2, \cdots$. Suppose $D = R_j$ for some value j. Does $j \in R_j$? If yes, by definition of D, $j \notin D$, which contradicts $D = R_j$. If, alternatively, $j \notin R_j$ then $j \in D$ which is again a contradiction.

2 Orders

- pre-order
- partial order
- lattice, bounded lattice and complete lattice

Exercise 1

In a poset $(P, \sqsubseteq)$ define the supremum of S, represented by $\sqcup S$, as the least upper bound (lub) of S. The *dual* notion of infimum, $\sqcap S$ is defined as the greatest lower bound (glb) of S.

Characterise lub and glb in $(\mathcal{P}(X), \subseteq)$ and $(\mathbb{N}, div)$, where div is integer division. Suppose P is a poset with a top and a bottom element $\top$ and $\bot$, respectively, i.e. $\sqcap P = \bot$ and $\sqcup P = \top$. Explain why $\sqcap\emptyset = \top$ and $\sqcup\emptyset = \bot$.

Exercise 2

A morphism between posets $(P, \sqsubseteq)$ and $(Q, \subseteq)$ is a function $f : P \longrightarrow Q$ such that

$$x \sqsubseteq y \Rightarrow f(x) \subseteq f(y)$$

i.e. a monotonic function. What extra structure must a morphism between lattices, bounded lattices or complete lattice preserve?

$\boxed{\textbf{Exercise } 3}$

A lattice is complete if infimum and supremum are defined for arbitrary subsets. Characterise as complete lattices i) the set of all sub-spaces of a vectorial space; ii) the set of sub-groups of a group; iii) any finite lattice.

The Knaster-Tarski theorem.

A most relevant result about complete lattices for the semantics of computation is the theorem of Knaster-Tarski [16] on the existence of fixed points of a monotonic function. Such special points (for which $x = f(x)$ give meaning to recursive functions.

$\boxed{\text{Theorem}}$

Let $(U, \sqsubseteq)$ be a complete lattice and $f : U \longrightarrow U$ a monotonic function. The least and the greatest fixed points of f are given by

$$\begin{aligned} m &= \bigsqcap \{x \in U \mid f(x) \sqsubseteq x\} \\ M &= \bigsqcup \{x \in U \mid x \sqsubseteq f(x)\} \end{aligned}$$

respectively.

Proof.

Let us show that m is the least fixed point of f. Let $X = \{x \in U \mid f(x) \sqsubseteq x\}$ and choose $x \in X$ arbitrarily. Clearly, $m \sqsubseteq x$ and, f being monotonic, $f(m) \sqsubseteq f(x)$. On the other hand, $f(x) \sqsubseteq x$, because $x \in X$. Thus, we may conclude that, for all $x \in X$, $f(m) \sqsubseteq x$. This means that m is the *least* pre-fixed point of f. In particular, $f(m) \sqsubseteq m$, which leads us to $f(f(m)) \sqsubseteq f(m)$. We conclude that $f(m) \in X$ and, thus, $m \sqsubseteq f(m)$. But then $f(m) = m$ as expected. The second part of the theorem comes from this one; if f is monotonic in $(U, \sqsubseteq)$, then it is also monotonic in the complete lattice formed by the inverse order $(U, \sqsupseteq)$. If M is the least fixed point of f in $(U, \sqsupseteq)$, it will be the *greatest* fixed point of the same function in $(U, \sqsubseteq)$.

$\boxed{\text{Lattices as algebraic structures}}$

Lattices can as well be seem as algebraic structures taking $\sqcup$ and $\sqcap$ as binary operations satisfying the axioms for commutativity, associativity, idempotence, and the following absorption laws:

$$\begin{aligned} a \sqcup (a \sqcap b) &= a \\ a \sqcap (a \sqcup b) &= a \end{aligned}$$

Clearly, $a \leq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a$.

3 Groups

... as a prototypical algebraic structure ...

A group (G, θ, u) is a set G with a binary operation θ which is associative, and equipped with an identity element u and an inverse:

$$a^{-1}\theta a \; = \; u \; = \; a\theta a^{-1}$$

Note that in *semigroup* lacks inverse, and a *monoid* also drops the identity element.

Exercise 4

Show that $(\mathbb{R}^+, \times, 1)$ and $(\mathbb{R}^+, +, 0)$ are groups. Prove that a bijection between them is obtained by functions ln_e and e^-.

Exercise 5

Show that

$$S_n \; = \; (\{\sigma : n \longrightarrow n \mid \sigma \text{ is a permutation}\}, \cdot, id)$$

is a group. This is usually called the *symmetry group of degree* n.

Exercise 6

Prove the following properties:

1. $a\theta b = a\theta c \;\Rightarrow\; b = c$ (dually, $b\theta a = c\theta a \;\Rightarrow\; b = c$).
2. $a^{-1^{-1}} \; = \; a$.
3. $(a\theta b)^{-1} \; = \; b^{-1}\theta a^{-1}$.
4. $f(a^{-1}) \; = \; f^{-1}a$.
5. The equation $a\theta x = b$ has a unique solution $x = a^{-1}\theta b$.

Cayley Theorem.

The set of bijections $f : X \longrightarrow X$ over a set X with functional composition forms a group of *transformations* (which is the identity? And the inverse?). The following is a main result in the theory of groups:

$\boxed{\text{Theorem}}$
Every group is isomorphic to a group of transformations

<u>Proof.</u>
Let (G, θ, u) be a group. For each element a of G define a map $f_a : G \longrightarrow G$ such that $f_a(x) = a\theta x$.

Let us show that a new group T can be defined over the set of transformations above:

1. The (functional) composition of two elements of T is in T:

$$(f_a \cdot f_b)(x) = f_a(f_b(x)) = f_a(b\theta x) = a\theta(b\theta x) = (a\theta b)\theta x = f_{a\theta b}(x)$$

2. For identity,

$$f_u(x) = u\theta x = x$$

3. For inverse,

$$f_a \cdot f_{a^{-1}}(x) = a\theta(a^{-1}\theta x) = (a\theta a^{-1})\theta x = u\theta x = x$$

We have proved that T is a group (note that axioms are inherited from the properties of function composition restricted to bijections). It remains to show that T is *isomorphic* to G. Let $h : G \longrightarrow T$ be defined by $h(a) = f_a$.

- Cllearly $h(a\theta b) = f_{a\theta b} = f_a \cdot f_b = h(a) \cdot h(b)$ is a homomorphim between both groups.
- T is entirely composed of bijections f_a for every element $a \in G$, thus h is a surjection.
- If $a \neq b$, then $h(a) = f_a \neq f_b = h(b)$; thus h is injective.

$\boxed{\text{Action of a group}}$

A group (G, θ, u) acts over a set X through a function (the action) $\tau : G \times X \longrightarrow X$ which satisfies the following properties: $\tau(u, x) = x$ and $\tau(g\theta f) = \tau(g, (\tau(f, x)))$.

$\boxed{\textbf{Exercise } 7}$

Show that i) the group S_n acts over set n (initial fragment of $\mathbb{N}$ with n numbers), and ii) that every group (G, θ, u) acts over itself through the map $(g, x) \mapsto g\theta x\theta g^{-1}$.

References.
There are several introductory textbooks on the mathematical background stuff discussed in this lecture. I would recommend Paul Halmos' very well written introductions to set theory [8] and to modern logic from an algebraic perspective [9]. Davey and Priestley textbook [4] on ordered structures is recommended for the second topic in the summary. For a very pleasant and solid, although not elementary, reading on algebraic structures I can't but recommend *the* book [13].

Lecture 2: Computability

Summary.
(1) The quest for a formal definition of algorithm.

(2) Turing machines. Universal Turing machines.

(3) Recursive functions. The Church-Turing thesis. The uncomputable.

1 The question

- What does it mean for a function to be computable?
- Are there any noncomputable functions?
- How does computational power depend on programming constructs?

Models	Grammars (Chomsky hierarchy)
finite memory: finite automata	right-linear grammars
finite memory with stack: pushdown automata	context-free grammars
linear bounded automata	context-sensitive grammars
unrestricted memory:	
Turing machines (Alan Turing)	
Post systems (Emil Post)	
μ-recursive functions (K. Gödel, J. Herbrand)	unrestricted grammars
λ-calculus (A. Church, S. Kleene)	
Combinatory logic (M. Schönfinkel, Haskell Curry)	

For those wondering on the use of grammars to specify computational models, notice that symbol manipulation or parsing a sentence in a language bears a strong resemblance to computation.

The quest for formalising the concept of effective computability started around the beginning of the twentieth century with the development of the formalist school of mathematics (Hilbert's programme) with the prospect of reducing all of mathematics to the formal manipulation of symbols. The formalist program was eventually shattered by Kurt Gödel's incompleteness theorem, which states that no matter how strong a deductive system for number theory you take, it will always be possible to construct simple statements that are true but unprovable. This theorem is essentially a statement about computability.

Church-Turing thesis

All the formalisms above capture precisely the same intuition about what it means to be *effectively computable*.

2 Turing machines

$$M = (Q, \Sigma, \Gamma, \delta, \prec, \Yup, s, t, r)$$

A Turing machine consists of a finite set of states Q, a semi-infinite tape that is delimited on the left end by an endmarker $\prec$ and is infinite to the right, and a head that can move left and right over the tape, reading and writing symbols.

The input string is of finite length and is initially written on the tape in contiguous tape cells snug up against the left endmarker $\prec$. The infinitely many cells to the right of the input all contain a special blank symbol $\Yup$. The machine starts in the start state s with its head scanning the left endmarker. In each step it reads the symbol on the tape under its head. Depending on that symbol and the current state, it writes a new symbol on that tape cell, moves its head either left or right one cell, and enters a new state. The action it takes in each situation is determined by a transition function

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

The meaning of $\delta(p, a) = (q, b, d)$ is *when in state* p *scanning symbol* a*, write* b *on that tape cell, move the head in direction* d*, and enter state* q.

Restrictions:

$\forall_p \exists_q .\ \delta(p, \prec) = (q, \prec, R)$	never moves off to the left of $\prec$
$\delta(t, -) = (t, -, -)$ and $\delta(r, -) = (r, -, -)$	never leave the accept (reject) state

The Turing machine accepts its input by entering a special accept state t and rejects by entering a special reject state r. On some inputs it may run infinitely (loop on) without ever accepting or rejecting.

A configuration is a tuple in $Q \times \{w \Yup^\omega \mid w \in \Sigma^*\} \times \mathbb{N}$ and denotes a global state of the machine. The configuration $\alpha = (p, z, n)$ specifies a current state p of the finite control, current tape contents z, and current position of the read/write head ($n \geq 0$). Ex. the initial configuration on input $x \in \Sigma^*$: $(s, \prec x \Yup^\omega, 0)$.

The *transition relation*:

$$(p, z, n) \rightarrow \begin{cases} (q, z[b/n], n-1) \ \Leftarrow \ \delta(p, z_n) = (q, b, L) \\ (q, z[b/n], n+1) \ \Leftarrow \ \delta(p, z_n) = (q, b, R) \end{cases}$$

Example: $\{a^n b^n c^n \mid n \geq 0\}$

- Start in state s and scans to the right over the input string to check that it is of the form $a^*b^*c^*$.
- Does not write anything on the way across (formally, it writes the same symbol it reads).
- When founding the first blank symbol $\curlywedge$, it overwrites it with a right endmarker $\succ$
- Then it scans left, erasing the first c it sees, then the first b it sees, then the first a it sees, until it comes to $\prec$.
- Then scans right, erasing one a, one b, and one c.
- It continues to sweep left and right over the input, erasing one occurrence of each letter in each pass. If on some pass it sees at least one occurrence of one of the letters and no occurrences of another, it rejects. Otherwise, it eventually erases all the letters and makes one pass between $\prec$ and $\succ$ seeing only blanks, at which point it accepts.

Example: $\{ww \mid w \in \{a, b\}^*\}$

- In a first phase, scans out the input to the first blank symbol, counting the number of symbols mod 2 to make sure the input is of even length and rejecting immediately if not.
- It lays down a right endmarker $\succ$, then repeatedly scans back and forth over the input.
- In each pass from right to left, it marks the first unmarked a or b it sees with an overline.
- In each pass from left to right, it marks the first unmarked a or b it sees with an underline.
- It continues this until all symbols are marked. The objective is to find the center of the input string.

$$\begin{array}{l}
\prec\ a\,a\,b\,b\,a\,a\,a\,b\,b\,a\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ a\,a\,b\,b\,a\,a\,a\,b\,b\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,a\,b\,b\,a\,a\,a\,b\,b\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,a\,b\,b\,a\,a\,a\,b\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,\underline{a}\,b\,b\,a\,a\,a\,b\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\prec\ \underline{a}\,\underline{a}\,b\,b\,a\,a\,a\,\overline{b}\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots \\
\cdots \\
\prec\ \underline{a}\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\,\overline{a}\,\overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \curlywedge\ \curlywedge\ \curlywedge\ \cdots
\end{array}$$

In a second phase, repeatedly scans left to right over the input.

- In each pass it erases the first symbol it sees marked with underline but remembers that symbol in its finite control.

- It then scans forward until it sees the first symbol marked with overline, checks that that symbol is the same, and erases it.
- If the two symbols are not the same, it rejects. Otherwise, when it has erased all the symbols, it accepts.

$$\prec\ \underline{a}\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\,\overline{a}\,\overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \cdots$$

$$\prec\ \text{⅄}\,\underline{a}\,\underline{b}\,\underline{b}\,\underline{a}\ \text{⅄}\ \overline{a}\,\overline{b}\,\overline{b}\,\overline{a}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \cdots$$

$$\prec\ \text{⅄}\ \text{⅄}\ \underline{b}\,\underline{b}\,\underline{a}\ \text{⅄}\ \text{⅄}\,\overline{b}\,\overline{b}\,\overline{a}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \cdots$$

$$\cdots$$

$$\prec\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \underline{a}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\,\overline{a}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \cdots$$

$$\prec\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \text{⅄}\ \cdots$$

Exercise 1

Specify a total TM that accepts its input string if its length is prime.

Hint. Give an implementation of the Sieve of Eratosthenes. To check whether n is prime, start writing down all the numbers from 2 to n in order. Then repeat: find the smallest number in the list, declare it prime, then cross off all multiples of that number. Repeat until each number in the list has been either declared prime or crossed off as a multiple of a smaller prime.

A Turing Machine is *total* if it halts (either by accepting or rejecting) in all inputs.

A language is

- *recursively enumerable* if it is the language $L(M)$ recognised by some Turing Machine M, i.e.

$$\begin{cases} M \text{ halts} & \Leftarrow w \in L \\ M \text{ halts in the non-acceptance state or loops forever} & \Leftarrow w \notin L \end{cases}$$

- *co-recursively enumerable* if its complement is recursively enumerable;
- *recursive* if it is the language $L(M)$ recognised by some *total* Turing Machine M, i.e.

$$\begin{cases} M \text{ halts in the acceptance state} & \Leftarrow w \in L \\ M \text{ halts in the rejectionstate} & \Leftarrow w \notin L \end{cases}$$

A property ϕ (over strings) is

- *decidable* if the set of all strings exhibiting ϕ is *recursive*
- *semidecidable* if the set of all strings exhibiting ϕ is *recursively enumerable*

Exercise 2

Given a decidable property, what is the corresponding recursive set? Conversely, which decidable property corresponds to a given recursive set?

Universal Turing Machines.

Turing machines are not restricted to do just a single computational task but can be *programmed* to do many different ones. Actually a Turing machine can simulate other Turing machines whose descriptions are presented as part of the input.

The key issue is to fix a reasonable encoding scheme for Turing machines over the alphabet $\{0, 1\}$, e.g.

$$0^n 1 0^m 1 0^k 1 0^s 1 0^t 1 0^u 1 0^v 1,$$

and then construct U such that

$$L(U) = \{M\#s \mid s \in L(M)\}$$

If the encodings of M and s are valid, U makes a step-by-step simulation of machine M.

We may then discuss the expressive power and limitations of Turing machines using no other instruments other than the machines themselves. It is precisely this self-referential property that Gödel exploited to embed statements about arithmetics in statements of arithmetics in his Incompleteness Theorem. The embedding in the Theorem is the same as the encoding of Turing machines into input forms acceptable for universal machines and is achieved by converting the finite description of a Turing machine into a unique non-negative integer. The conversion is possible as we are only dealing here with machines having a finite number of states, a finite number of symbols in its alphabet, and only a finite number of movements for their heads.

3 Recursive functions

Alternative approach to computability, focussed on *what is computed* rather than on an explicit *model of computation*.

- k-ary *zero* functions: $z(n_1, \cdots, n_k) = 0$
- k-ary, j-*projection* functions: $id_k, j(n_1, \cdots, n_k) = n_j$
- the *successor* function: $s(n) = n + 1$
- function *composition* of a k-ary function g with k m-ary functions h_i:

$$f(n_1, \cdots, n_m) = g(h_1(n_1, \cdots, n_m), h_2(n_1, \cdots, n_m), \cdots, h_k(n_1, \cdots, n_m))$$

- recursive definition by a k-ary g and a $k+2$-ary h function:

$$\begin{aligned} f(n_1, \cdots, n_k, 0) &= g(n_1, \cdots, n_k) \\ f(n_1, \cdots, n_k, m+1) &= h(n_1, \cdots, n_k, m, f(n_1, \cdots, n_k, m)) \end{aligned}$$

Exercise 3

Show that if $f : \mathbb{N}^2 \longrightarrow \mathbb{N}$ is primitive recursive, so is function

$$\mathrm{sum}_f(n, m) \;=\; \sum_{i \in \{0,1,\cdots,m\}} f(n, i)$$

Exercise 4

Show that the set of primitive recursive functions is countable. Use this fact to prove that most functions in $\mathbb{N}$ are not primitive recursive, and therefore, according to Church thesis, uncomputable.

Exercise 5

Show that the factorial function, the function that computes the greatest common divisor and the prime predicate (which returns 1 if its argument is a prime number) are all primitive recursive functions.

μ-recursive functions

Let p be a $k+1$-ary function and define the following *unbounded iteration* scheme:

$$\mu_m[p(n_1, \cdots, n_k, m) = 1] \;=\; \begin{cases} \text{the least } m \text{ such that } p(n_1, \cdots, n_k, m) = 1 \;\Leftarrow\; \text{such a } m \text{ exists} \\ 0 \;\Leftarrow\; \text{otherwise} \end{cases}$$

The obvious way to compute this function is through unbounded iteration, i.e. through a `while` loop:

```
$m := 0$;
while $p(n_1, \cdots, n_k, m) \neq 1$ do
    $m := m + 1$
end
return $m$
```

This, however, may fail to terminate. We call function p *minimalizable* if the minimization scheme above terminates for every input.

Functions defined by the basic functions above, composition, recursion and minimization are called μ-*recursive*.

Example

Consider function $\mathrm{log}(b, n)$ standing for the logarithm of $n + 1$ over base $b + 2$. Formally,

$$\mathrm{log}(b, n) \;=\; \mu_m[\mathrm{geq}(b + 2)^m, n + 1)]$$

Exercise 6

Why are we using $b + 2$ and $n + 1$ above? Show that the minimization algorithm terminates, i.e. function

$$f(b, n, m) \;=\; \mathrm{geq}((b + 2)^m, n + 1)$$

is minimizable.

Theorem

A function $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ is μ-recursive iff it is recursive, i.e. computable by a Turing machine.

Exercise 7

Suppose that g is a μ-recursive bijection over $\mathbb{N}$. Show that its inverse g^{-1} is μ-recursive as well.

4 Undecidability

Our quest for a precise notion of an *algorithm* is concluded with the identification

$$\boxed{\text{Algorithm} \equiv \text{a Turing machine that halts on all inputs}}$$

This, however, opens the possibility of formally showing that there are some computational problems which cannot be solved by any algorithm.

Turing machines (like other formalisms such as finite or push-down automata, regular, context-free or unrestricted grammars) can be represented by sequences of symbols (cf, the discussion on universal Turing machines). As there is only a countable number of languages defined over an alphabet, the number of languages specified by Turing machines (cf, recursive and recursive enumerable languages) is also countable. Turing machines decide or semidecide on an infinitesimal fraction of all possible languages. According to the Church-Turing thesis, we have reached a fundamental limitation: *computational tasks that cannot be performed by a Turing machine are undecidable.*

The halting problem

Suppose one can write an algorithm $\mathrm{dec}(p, x)$ which receives a program p and a value x as input and decides whether it terminates or not. Such a wonderful program could be used to write the following procedure:

Algorithm 1: DecideHalt.

```
1 if dec(P,P) then
      go to 1;
  else
      halt;
  end
```

Exercise 8

Explain why $\mathrm{dec}(p, x)$ cannot exist and conclude that no algorithm to decide whether an arbitrary program would halt or loop does not exist.

We have now a formalised notion of an algorithm and a sort of universal programming language – the Turing machine – in which to define a language which is not recursive.

$$Y = \{\text{"}Mw\text{"} \mid \text{Turing machine } M \text{ halts on input } x\}$$

Clearly, Y is recursively enumerable: it is the language recognised by an universal Turing machine. Such a machine halts exactly when its input is in Y.

Suppose that Y is decidable by some machine N. Then, given a particular Turing machine M semideciding language $L(M)$, one could design another machine that actually *decides* $L(M)$ by writing "M""w" in its input tape and then simulating N on this input. If such is the case, every recursive language would be recursively enumerable.

However, Y is not recursive. Actually, if it were recursive, language

$$Z = \{\text{"}M\text{"} \mid \text{Turing machine } M \text{ halts on input "}M\text{"}\}$$

would also be recursive (cf. put "M""M" on the tape of a new machine and hand control to N). As it can be proved that recursive languages are closed for complemente, it suffices to show that the complement of Z is not recursive.

$$\overline{Z} = \{w \mid \Psi(w)\}$$

where $\Psi(w) = w$ is not encoding of a Turing machine, or it is the encoding of a Turing machine M that does not halt on "M".

$\overline{Z}$, finally, is not recursively enumerable, let alone recursive, Suppose that there exists a machine K semideciding $\overline{Z}$. Is "K" in Z?

- "K" $\in \overline{Z}$ iff K does not accept input "K";
- but, K is supposed to semidecide Y; so "K" $\in \overline{Z}$ iff K accepts input "K".

Thus, we get an example of a non recursive language and proved an important theorem: the set of recursive languages is a strict subset of the set of recursive enumerable functions.

The Halting problem: *there is no algorithm that decides, for an arbitrary Turing machine M and input w, whether or not M accepts w.*

Other undecidable problems for Turing machines:

- does M halt on the empty tape?
- is there any string at all on which M halts?
- does M halts on every input?
- given two Turing machines, do they halt on the same input?
- does M fails to halt on input w?

Other problems can be reduced to one of these (e.g. the tiling problem).

References.
Both the textbook of H. Lewis and D. Papadimitriou [12] or the lecture notes by D. Kozen [11] provide excellent introductions to computability and the (classical) theory of computation. A quite interesting book by N. Yanofsky [17] may help to build up the correct intuitions which often is as important as mastering the technicalities. As a side reading on Gödel's incompleteness theorem and connections to computability I suggest reference [3]. A. Hodges biography of Alan Turing [10] makes a most pleasant weekend reading (see also the book website at `www.turing.org.uk`).

Lecture 3: Computational complexity

Summary.
(1) Notion of growth rate of a function. Examples.

(2) Case study: closure operations.

(3) Computational complexity. The class P of polynomial decidable languages. $\mathsf{P} \neq \mathsf{NP}$?

---

1 Analysis of algorithms

Growth rate.

Example: transitive closure of $R \subseteq A^2$

from 'above' R^* is the smallest relations containing R that is transitive and reflexive.

from 'below'

$$R^* \; = \; \{(a, b) \mid a, b \in A \text{ there exists a path from } a \text{ to } b \text{ in } R\}$$

which suggests an algorithm:

Algorithm 2: TC1.

```
R* := ∅;
for i := 1..n do
    for each i-tuple (b_1, ··· , b_i) ∈ A^i do
        if it is a path then
            R* := R* ∪ {b_1, b_i}
    end
end
```

One may inquiry this algorithm on

- *correctness* vs *termination*
- ... but when?

Growth rate for functions

$$\mathcal{O}(f) \; = \; \{g \in \mathbb{N}^{\mathbb{N}} \mid \exists_{c,d \in \mathbb{N}+}. \, \forall_n. \; g(n) \leq c.f(n) + d\}$$

Exercise 9

Define $f \sim g$ iff $f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f)$. Prove $\sim$ is an equivalence relation.

Example: $p(n) = 31n^2 + 17n + 3$

Clearly $p(n) \leq 48n^2 + 3$, because $n^2 \geq n$. Thus $f \in \mathcal{O}(\text{—}^2)$ with constants 48 and 3.
However, $\text{—}^2 \in \mathcal{O}(p)$ with constants 1 and 0.

Theorem

For any polynomial $p(n) = c_k n^k + \cdots + c_1 n + c_0$, $p \in \mathcal{O}(\text{—}^k)$ with constants $\sum_{1 \leq i \leq k} c_i$ and c_0.

Theorem

Any two polynomials p and q with the same degree verify $p \sim q$.

Theorem

The growth rate of function 2^n is higher than the one of an arbitrary polynomial.

Proof.
We want to show that

$$n^i \ \in \ \mathcal{O}(2^n) \ \text{ i.e. } n^i \leq c2^n + d \tag{1}$$

Let $c = (2i)^i$ and $d = (i^2)^i$, and consider two cases:

- $n \leq i^2 \ \Rightarrow \ (1)$, because $n^i \leq d$
- $n > i^2 \ \Rightarrow \ (1)$, because $n^i \leq c2^n$

Note that $n^i \leq (iq+i)^i = i^i(q+1)^i$, for q the integer quotient of n by i (i.e. $iq \leq n \leq i(q+1)$).
Now,

$$\begin{array}{ll}
 & i^i(q+1)^i \\
\leq & \qquad \{ n \leq 2^n \} \\
 & i^i(2^{q+1})^i \\
\leq & \qquad \{ \text{definition of } c \} \\
 & c2^{qi} \\
\leq & \qquad \{ \text{definition of } q \} \\
 & c2^n
\end{array}$$

Observe now that if a polynomial had the same growth rate than —^2, then any polynomial of a higher degree would have the same rate (because we've just proved that no polynomial grows as fast as —^2). But this leads to a contradiction.

□

2 Case study: Closure algorithms

Computing R^*.

Transitive closure (TC1)

The algorithm examines each sequence $(b_1, \cdots, b_i)$; if this is a path add to the solution. Thus, the total number of operations is

$$n(1 + n + n^2 + \cdots + n^n)$$

Therefore, TC1 $\in \mathcal{O}(n^{n+1})$.

Transitive closure (TC2)

Algorithm 3: TC2.

```
$R^* := R \cup \{(a, a) \mid a \in A\}$;
while $\exists_{a_1, a_j, a_k \in A}.\ (a_i, a_j), (a_j, a_k) \in R,\ (a_i, a_k) \notin R$ do
    $R^* := R^* \cup \{(a_i, a_k)\}$
end
```

- In each iteration one pair (if any) is added. Thus, (TC2) makes n^2 iterations maximum.
- In each iteration the algorithm searches for n^3 triples.

Therefore, TC2 $\in \mathcal{O}(n^2 \times n^3) = \mathcal{O}(n^5)$.

Exercise 10

Obtain a better algorithm of $\mathcal{O}(n^2 \times n) = \mathcal{O}(n^3)$ by imposing an order to the triples so that a new pair added does not violate the transitivity condition established for triples already considered.

Closure problems.

A subset $C \subseteq A$ is *closed* for a relation $R \subseteq A^{n+1}$ if

$$b_{n+1} \in C \quad \Leftarrow \quad b_1, \cdots, b_n \in C \ \wedge \ (b_1, \cdots, b_n, b_n + 1) \in R$$

e.g.

- $\mathbb{N}$ is closed for $+$
- the set of ancestors is closed for the relation *parent-of*
- any set is closed for $\subseteq$

Closure property: *The set* C *is closed under relations* $R_1, \cdots, R_m$

cf, the usual construction *the smallest set that contains* A *and has property* ϕ. But note that not all properties guarantees the existence of a smallest set satisfying ϕ. However,

Theorem

If ϕ is a closure property in A and $B \subseteq A$, then there exists the smallest set C st $B \subseteq C$ and C is closed for ϕ.

Proof.
Let ϕ be defined by a realtion R and S denote the set of subsets of A containing B and closed for R. Clearly $S \neq \emptyset$ (why?). Then,

- $A \subseteq C$
- C is closed for R. Let $a_1, \cdots, a_{n-1} \in C$ and $(a_1, \cdots, a_{n-1}, a_n) \in R$. All sets in S contain $a_1, \cdots, a_{n-1}$ and because all of them are closed, all have a_n. Thus, $a_n \in C$.
- C is minimal: no strict subset C' of C exists (otherwise $C' \in S$).

□

Theorem

Any closure property over a finite set can be computed in polynomial time.

Proof.

Algorithm 4: Computing a generic closure.

```
$C^\circ := C$;
while $\exists_{1 \leq i \leq k \textit{ and } r_i \textit{ elements } a_{j1} \cdots a_{j_{r_{i-1}}} \in C^\circ \textit{ and } a_{j_{r_i}} \in D \setminus C^\circ} \cdot (a_{j1} \cdots a_{j_{r_i}}) \in R_i$ do
    $C^\circ := C^\circ \cup \{a_{j_{r_i}}\}$
end
```

Thus, the algorithm is $\mathcal{O}(n^{r+1})$ where n is the cardinal of D and r is the greatest arity of all relations considered.

□

Theorem

Any algorithm in polynomial time can be rendered as the computation of a closure over a set for a set of relations.

3 Computability *vs* complexity

The Travelling Salesman Problem.

Given a map with n cities and distances in Km, produce an itinerary that minimizes the total distance travelled.

- Clearly solvable (e.g. systematic examination of all itineraries)
- but unsolvable in any practical sense by current computers: too many itineraries ($(n-1)!$) to be explored. Notice that a $(n-1)!$ algorithm goes faster that 2^n.

What is a *practically feasible algorithm*?

... should run for a number of steps bounded by a *polynomial* in the length of the input.

Polynomially decidable languages.

A language is *polynomially decidable* if there is a polynomially bounded Turing machine that decides it, i.e. a Turing machine which always halts after at most $p(n)$, where $p(n)$ is a polynomial and n is the length of the input.

The class P of such languages is the quantitative analog of the class of recursive languages. As the latter it is closed under complement, union, intersection, concatenations and Kleene star. But, on the other hand, not all recursive languages are polynomially decidable.

$\boxed{\text{Theorem}}$

$S \notin P$, where

$$S = \{"M""w" \mid M \text{ accepts input } w \text{ after at most } 2^{|w|} \text{ steps}\}$$

Proof.

If $S \in P$, language

$$S' = \{"M" \mid M \text{ accepts input } "M" \text{ after at most } 2^{|"M"|} \text{ steps}\}$$

and its complement are also in P. This means that there exists a polynomially bounded Turing machine B which accepts all descriptions of Turing machines that fail to accept their own description in 2^n steps, where n is the length of the description, and halts in $p(n)$ steps for a polynomial $p(n)$.

Does B accept its own description "B"?

- If YES then B fails to accept "B" within $2^{|"B"|}$ steps. However, B halts in $|"B"|$ steps, i.e. much before $2^{|"B"|}$. Thus it should reject "B", which leads to a contradiction. Note that there is always an integer n_0 such that $p(n) \leq 2^n$ for all $n \geq n_0$, and we may safely assume $|"B"| \geq n_0$.

- If NO a similar argument also leads to contradiction.

Problems.

Reachability. Given two nodes of a finite graph decide if there is a path connecting them.

> Is a variant of the reflexive-transitive closure problem. Can be solved by computing this closure in time $\mathcal{O}(n^3)$ and inspect the result.

A *problem* is a set of inputs, typically infinite, with a Boolean question to be asked of each input. A problem needs to be encoded as a language problem so that its complexity can be analysed in a common setting. For example, the *Reachability* problem can be reduced to a decision problem for the language

$$R = \{K(G)s(i)s(j) \mid \text{there is a path in G connecting nodes } n_i \text{ to } n_j\}$$

where K and s are suitable binary encoding functions for graphs and integers.

Other problems

Euler Cycle. Given a graph is there a closed path in it that uses each edge exactly once?

> Note that the path can go many times through the same node (or even not at all if there are isolated nodes). It can be proved that the necessary condition on a graph to have such a path is that i) all nodes have equal numbers of incoming and outgoing edges, and ii) for each pair of nodes, neither of which isolated, there is a path connecting them. So, clearly the corresponding language
>
> $$G = \{K(G) \mid \text{G has an Euler cycle}\}$$
>
> where $K(G)$ is some encoding of graphs as strings, is in n P.

Hamilton Cycle. Given a graph is there a cycle that passes through each node exactly once?

> No polynomial algorithm is known. Of course the trivial one (generate all paths and choose) is not polynomial.

Equivalence of Finite Automata. Given two deterministic automata, determine whether they recognise the same language?

> The problem is polynomial, as it is the variant in which only regular expressions are considered. However, one cannot conclude about the latter just by reducing to the former: actually, the generation of a finite automaton from a regular expression may increase exponentially the number of states.

Integer Partition. Given a set of n nonnegative integers represented in binary, is there a subset S of the original set such that $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?

The algorithm is $\mathcal{O}(nV)$ where V is the sum of all numbers in the original set divided by 2. In spite of its polynomial appearance, the problem is not polynomial in the length of the input. The reason is that the integers are encoded in binary: if all integers are about $\mathbf{2}^n$, then S is close to $\mathbf{2}^n \times \frac{n}{2}$.

Satisfiability. Is a Boolean formula in conjunctive normal form satisfiable?

No polynomial algorithm is known. However, if reduced to formulas with a maximum of two literals, it becomes polynomial.

$\boxed{\text{Optimisation problems}}$

Require to find the best among many possible solutions, according to some cost function. The trick to transform optimisation into language problems is to fix each input with a *bound* on the cost function. For example, the *Traveling Salesman* problem can be rephrased as

Given an integer $n \geq 2$*, a* $n \times n$ *distance matrix*, and an integer $b \geq 0$, find a permutation of n such that its cost is less or equal to b (which, to build up intutition, may be regarded as a budget).

Independent Set. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \geq K$ such that for any two vertices in s there is no edge connecting them?

Clique. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \geq K$ such that for all vertices in s there is an edge connecting each pair?

Node Cover. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \leq K$ such that s covers all edges of the graph? (cf, minimising guards in a museum).

Note that a set of nodes covers an edge if it contains at least one endpoint of the edge.

No polynomial algorithms are known.

The class NP**.**

Most interesting problems mentioned above for which no polynomial algorithm exists — *Traveling Salesman, Satisfiability, Independent Set, Integer Partition*, etc., can be solved by *polynomially bounded nondeterministic Turing machines*. All computations of such machines do not continue for more than polynomially many steps.

This defines the class NP (of nondeterministic polynomial languages).

Although determinism and nondeterminism in the definition of Turing machines do not interfere on their expressiveness in what concerns decidability, separating determinism form nondeterminism at the polynomial level (the $\mathsf{P} \neq \mathsf{NP}$ conjecture), remains unsolved.

These problems share an important completeness property: All problems in NP can be reduced to them (just as all recursively enumerable languages reducing to the halting problem). Such problems are called NP-complete.

References.
My preference on complexity theory is Papadimitriou's wonderful book [15]; reference [6] provides an interesting alternative. S. Arora and B. Barak book [2] is a more recent textbook covering recent achievements in complexity theory (including challenges from quantum computation) and putting them in the context of the classical results. Worth reading.

Lecture 4: Introduction to Quantum Computing

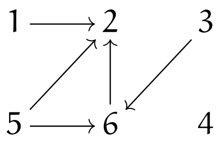
Summary.
(1) Nondeterministic, probabilistic and quantum transition systems.

(2) Introduction to quantum computing. The circuit model. The Deutsch-Jozsa algorithm.

1 Transition systems

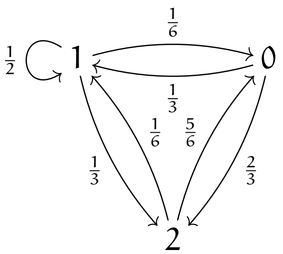
(... in dialogue with the "Compilers and Language Processing" course unit ...)

Nondeterministic automata



- The states of a system correspond to column vectors;
- The automata dynamics is encoded in Boolean matrices: $M[i, j] = 1$ if and only if there is an edge (path of length 1) from vertex j to vertex i;
- Multiplying the current state vector by matrix M yields progress from one state to another in one time step;
- Multiple step dynamics are obtained via matrix multiplication.

Probabilistic automata

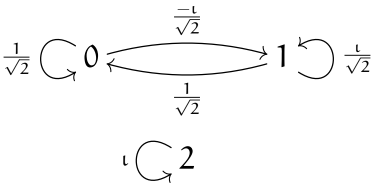


- The vectors that represent states of a probabilistic physical system express indeterminacy about the exact physical state of the system;
- The matrices that represent the dynamics express indeterminacy about the way the physical system will change over time;

- The entries of the evolution matrix enable the computation of the likelihood of transitioning from one state to the next: $M[i, j]$ gives the probability of a transition from vertex j to vertex i;
- Typically, but not necessarily, matrices encoding automata dynamics are *double stochastic*, which encodes the following two conditions:
  - the sum of all the weights leaving a vertex is 1 and
  - the sum of all the weights entering a vertex is 1.

  i.e. the dynamics is time symmetric.
- The way in which the indeterminacy progresses is simulated by matrix multiplication.

Quantum automata



- States in a quantum automaton are represented by column vectors of complex numbers whose sum of moduli squared is 1.
- The dynamics is represented by unitary matrices and is therefore reversible. $M^\dagger$ takes a state from time t to $t-1$. Note that the modulus squared of a unitary matrix forms a doubly stochastic matrix. Actually, the probabilities of quantum mechanics are always given as the modulus square of complex numbers.
- The weights on a quantum automaton are complex numbers whose modulus square is a real number between 0 and 1. Actually, if real number probabilities can only increase when added, complex numbers can cancel each other and lower their probability (therefore capturing *interference*): $|c_0 + c_2|^2$ need not be bigger than $|c_0|^2$ or $|c_0|^2$.
- Quantum states can be superposed, i.e. a physical system can be in more than one basic state simultaneously.

Exercise 11

Show that a unitary matrix preserves the sum of the modulus squares of a column vector multiplied on its right.

2 Design of quantum algorithms.

The setting: computing with a quantum device.

Data

The *qubit* provides the elementary assembly block in the quantum world. Formally,

$$\begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

such that $|c_0^2| + |c_1^2| = 1$. Complex c_0 is to be interpreted as the probability that after measuring the qubit, it will be found in state $|0\rangle$.

In general, the *state* of a closed physical system is wholly described by a vector of complex numbers

$$S \; = \; \begin{bmatrix} s_0 \\ \vdots \\ s_n \end{bmatrix}$$

such that $|s_0^2| + |s_1^2| + \cdots + |s_n^2| = 1$.

This condition is equivalent to S being *unitary*. Indeed,

$$|s_0^2| + |s_1^2| + \cdots + |s_n^2| = 1$$

$$\Leftrightarrow \qquad \{\, z^*z = |z|^2, \text{ for any complex number } z\}$$

$$s_0^* s_0 + s_1^* s_1 + \cdots + s_n^* s_n = 1$$

$$\Leftrightarrow \qquad \{\, \text{matrix multiplication}\}$$

$$[s_0^*, s_1^*, \cdots s_n^*] \cdot \begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{bmatrix} = 1$$

$$\Leftrightarrow \qquad \{\, \text{transpose}\}$$

$$\begin{bmatrix} s_0^* \\ s_1^* \\ \vdots \\ s_n^* \end{bmatrix}^{\mathsf{T}} \cdot \begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{bmatrix} = 1$$

$$\Leftrightarrow \qquad \{\, \text{definition of } \dagger\}$$

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{bmatrix}^{\dagger} \cdot \begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_n \end{bmatrix} = I_1$$

The number of components in the vector provide a measure of the system's complexity (its degrees of freedom or *dimension*).

Computation

- The algorithm is *initialised* by preparing a particular classical state;
- From there the system is *put into a superposition* of many states:
- ... which is transformed through the *application of a several reversible operations* (encoded as unitary matrices)
- A *measurement* is performed.

  When a physical system is measured, it collapses to (a classical) state with probability $p_i = |s_i^2|$:

$$\begin{bmatrix} s_0^* \\ \vdots \\ s_i^* \\ \vdots \\ s_n^* \end{bmatrix} \xrightarrow{|s_i^2|} \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

  with probability $p_i = |s_i^2|$. Of course, qubits collapse into classical bits:

$$\begin{tikzcd}
& {[c_0, c_1]^T} \arrow[dl, "|c_0|^2"'] \arrow[dr, "|c_1|^2"] & \\
{|0\rangle = [1, 0]^T} & & {|1\rangle = [0, 1]^T}
\end{tikzcd}$$

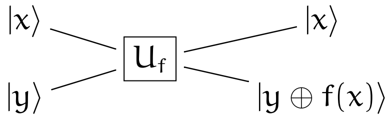
3 The Deutsch-Jozsa algorithm

Problem (simplified version)

Decide whether a function $f : \mathbf{2} \longrightarrow \mathbf{2}$ is *constant* with a unique evaluation of f.

Build a gate:

Evaluating a function f is equivalent to multiply a state through a matrix somehow encoding f. For exemple, if $f(x) = \neg x$, multiplying state $|0\rangle$ by $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ results in state $|1\rangle$. However, in quantum computing every gate must be unitary, and thus reversible. So, for the sake of this problem, we'll encode a generic function $f : \mathbf{2} \longrightarrow \mathbf{2}$ in gate



where $\oplus$ stands for exclusive disjunction. The first qubit $|x\rangle$ is the value that one wishes to evaluate; the second one is used for control. This gate takes input $|x, y\rangle$ to $|x, y \oplus f(x)\rangle$ (for $y = 0$ the output is $|x, f(x)\rangle$).

Exercise 12

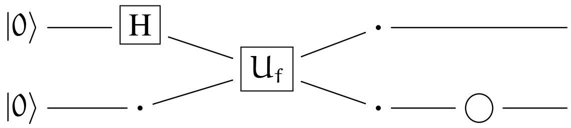
Give U_f for f the negation operator.

Exercise 13

Show U_f is a valid gate (i.e. a unitarian matrix).

Attempt 1

Explore superposition (resorting to the Hadamard gate).



$$U_f(H \otimes I)(|0\rangle \otimes |0\rangle)$$

$$|\sigma_1\rangle \;=\; (H \otimes I)(|0\rangle \otimes |0\rangle) \;=\; (H \otimes I)|0,0\rangle \;=\; |H|0\rangle,0\rangle \;=\; \frac{|0\rangle+|1\rangle}{\sqrt{2}}|0\rangle \;=\; \frac{|0,0\rangle+|1,0\rangle}{\sqrt{2}} \;=\; \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$$

$$|\sigma_2\rangle \;=\; U_f|\sigma_1\rangle \;=\; |H|0\rangle,0\rangle \;=\; U_f(\frac{|0,0\rangle+|1,0\rangle}{\sqrt{2}}) \;=\; \frac{|0,f(0)\rangle+|1,f(1)\rangle}{\sqrt{2}}$$

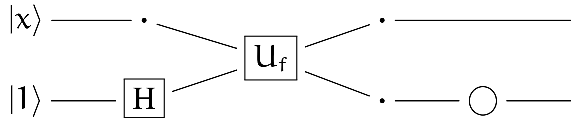
Unfortunately the problem was not solved: for $f(x) = \neg x$, we get

$$|\sigma_2\rangle \;=\; \frac{|0,1\rangle+|1,0\rangle}{\sqrt{2}} \;=\; \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$$

Measuring the second qubit the probability of it being in $|0\rangle$ (respectively, $|1\rangle$) is 50% (respectively, 50%). A similar conclusion arises for the first qubit.

Attempt 2

Keep the first qubit as given and lift the second one to a superposition state, i.e.



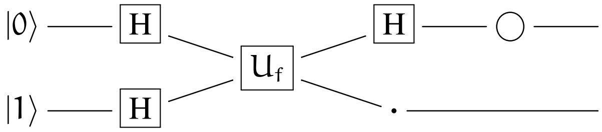
$$U_f(I \otimes H)(|x\rangle \otimes |1\rangle)$$

$$\begin{aligned}
|\sigma_1\rangle &= |x\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{|x,0\rangle - |x,1\rangle}{\sqrt{2}} \\
|\sigma_2\rangle &= U_f|\sigma_1\rangle = |x\rangle \frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} \\
&= \begin{cases} |x\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \Leftarrow f(x) = 0 \\ |x\rangle \frac{|1\rangle - |0\rangle}{\sqrt{2}} & \Leftarrow f(x) = 1 \end{cases} \\
&= (-1)^{f(x)} |x\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}}
\end{aligned}$$

Again, the exercise failed: no significative information can be retrieved from any of the qubits.

$\boxed{\text{Attempt 3}}$

Put both entries in superposition and compose the output again with a Hadamard gate.



$$(H \otimes I)\, U_f\, (H \otimes H)(|0\rangle \otimes |1\rangle)$$

$$
\begin{aligned}
|\sigma_1\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{|0,0\rangle - |0,1\rangle + |1,0\rangle - |1,1\rangle}{2} = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix} \\
|\sigma_2\rangle &= \left(\frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}}\right) \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \\
&= \begin{cases} (\pm 1) \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) & \Leftarrow \text{f constant} \\ (\pm 1) \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) & \Leftarrow \text{f not constant} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
|\sigma_3\rangle &= H|\sigma_2\rangle \\
&= \begin{cases} (\pm 1)\, |0\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) & \Leftarrow \text{f constant} \\ (\pm 1)\, |1\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) & \Leftarrow \text{f not constant} \end{cases}
\end{aligned}
$$

To answer the original problem is now enough to measure the first qubit: f is constant if it is in state $|0\rangle$.

$\boxed{\textbf{Exercise } 14}$

If f is constant we will measure $|0\rangle$ regardless if the state was $-|0\rangle$ or $|0\rangle$. Which information is discarded in this process?

Note: Observe the role of the Hadamard gates in first changing the basis of the first qubit to a superposition (in which the function is evaluated with the second qubit in a superposition as well), and later to revert back to the canonical basis.

The Deutsch-Jozsa algorithm

Problem: Decide whether a function $f : \mathbf{2}^n \longrightarrow \mathbf{2}$ (of which we are assured to be constant or balanced, i.e. mapping exactly half of the inputs to 0) is *constant* with a unique evaluation of f.

The previous simplified solution is the $n = 1$ instance of the general one.

Classically, the problem is solved by evaluating the function on different inputs: 2 in the best case (when two evaluations return different results), or $2^{n-1} + 1$ in the worst. The quantum solution brings an exponential speedup.

References.
References [14, 18, 19] will support your study of quantum computation along this course.

To close this module, a note on quantum computation and computability is in order. The physical variant of the Church-Turing thesis states that any function that can be computed by a *physical* system can be computed by a Turing Machine. It is therefore a strong statement of belief about the limits of both physics and computation. Whether, however, quantum computation will bring us beyond the classical computability framework is still not completely settled.

Most researchers agree that the shift from classical to quantum computers challenges the notion of complexity: some functions can be computed faster on a quantum computer than on a classical one. But, it does not challenge the physical Church-Turing thesis itself: a quantum computer can always be simulated by pen and paper, and thus, what it computes can be computed classically. David Deutsch original paper [5], which introduced quantum Turing machines, goes in this direction and is worth reading. You may refer to [7] for a more resent perspective.

For quantum complexity, I suggest S. Aaronson lecture notes [1], even if the stuff is a bit technical and clearly out of the scope of this course unit.

References

[1] S. Aaronson. The complexity of quantum states and transformations: From quantum money to black holes. *CoRR (Lecture Notes for the McGill Invitational Workshop on Computational Complexity, Bellairs Institute, Holetown, Barbados)*, abs/1607.05256, 2016.

[2] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[3] M. Baaz, C. H. Papadimitriou, H. W. Putman, D. S. Scott, and C. L. Harper. *Kurt Godel and the Foundations of Mathematics*. Cambridge University Press, 2011.

[4] D. A. Davey and H. A. Priestley. *Introduction to lattices and Order (Second Edition)*. Cambridge University Press, 2002.

[5] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A*, 400:97–117, 1985.

[6] D. Z. Du and K. I. Ko. *Theory of Computational Complexity*. Addison-Wesley, 2000.

[7] Stefano Guerrini, Simone Martini, and Andrea Masini. Towards A theory of quantum computability. *CoRR*, abs/1504.02817, 2015.

[8] P. Halmos. *Naive Set Theory*. Springer (Undergraduate texts in Mathematics), 1974.

[9] P. Halmos and S. Givant. *Logic as Algebra*. The Mathematical Association of America (Dolciani Mathematical Expositions, 21), 1998.

[10] A. Hodges. *Alan Turing, the Enigma*. Princeton University Press, 1983.

[11] D. C. Kozen. *Automata and Computability*. Springer, 1999.

[12] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall (2nd Edition), 1997.

[13] S. Mac Lane and G. Birkhoff. *Algebra (Third Edition)*. Cambridge University Press, 1988.

[14] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information (10th Anniversary Edition)*. Cambridge University Press, 2010.

[15] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[16] A. Tarski. A lattice–theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[17] N. S. Yanofsky. *The Outer Limits of Reason*. MIT Press, 2013.

[18] N. S. Yanofsky and M. A. Mannucc. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008.

[19] M. Ying. *Foundations of Quantum Programming*. Elsevier, 2016.