Introduction to computer science

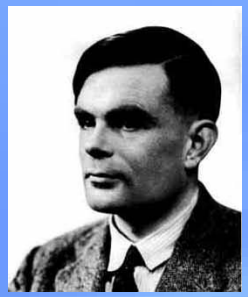
Michael A. Nielsen

University of Queensland

Goals:

1. Introduce the notion of the *computational complexity* of a problem, and define the major computational complexity classes.
2. Explain how to compute *reversibly.*

Q: Is there a general algorithm to determine whether a mathematical conjecture is true or false?



Church-Turing:
NO!

≡

Turing Machine

Computer science

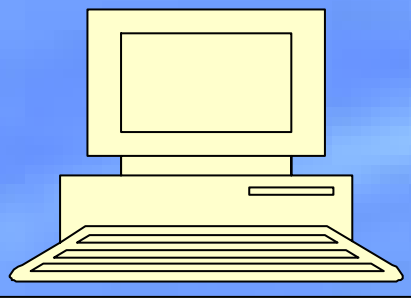
Ad hoc empirical justification!

Church-Turing thesis: Any algorithmic process can be simulated on a Turing machine – an idealized and rigorously defined mathematical model of a computing device.

"Turing machines"

Many different models of computation are equivalent to the Turing machine (TM).

We will use a model other than the TM for reasons of both pedagogy and utility.



INPUT
x = "I like quantum information science"

PROGRAM

```
FOR j = 1:LENGTH(x)
      OUTPUT x
NEXT
```

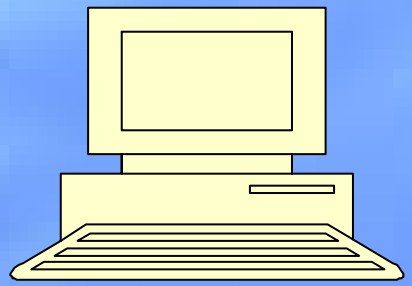
Program: 01001100…010

Input: 11101100101……

Workspace: 000……

Output: 11001……

How to number programs and their inputs



The program and input for a universal computer can be uniquely encoded as a pair of positive integers.

INPUT
x = "I like quantum information science"

PROGRAM

```
FOR j = 1:LENGTH(x)
      OUTPUT x
NEXT
```

binary → 0001100001… → 123456789

binary → 110011001… → 987654321

The halting problem

Does program number x halt on input of x?

$$h(x) \equiv \begin{cases} 0 \text{ if program } x \text{ halts on input } x \\ 1 \text{ otherwise} \end{cases}$$

Is there an algorithm to solve the halting problem, that is, to compute $h(x)$?

Suppose such an algorithm exists.

Let T be the program number for TURING.

Contradiction!

```
PROGRAM: TURING(x)

IF h(x) = 1 THEN
    HALT
ELSE
    loop forever
```

$h(T) = 0$  TURING(T) halts  $h(T) = 1$

Exercise: Show that there is no algorithm to determine whether program number x halts with input y.

Exercise: Show that there is no algorithm to determine whether program number x halts with input 0.

What is the relationship of the halting problem to Hilbert's problem?

Conjecture: Program number *x* halts on input of *x*.

No algorithm exists to prove or refute this conjecture, in general.

Isn't this a rather artificial conjecture?

Conjecture: Topological space *X* is topologically equivalent to topological space *Y*.

Many other "natural" mathematical conjectures cannot be algorithmically decided.

Computational complexity theory

Goal: A general theory of the *resources* needed to solve *computational problems*.

What types of resources?

time

energy

space

What types of computational problems?

composing a poem

optimization

sorting a database

decision problem

Decision problems

A decision problem is a computational problem with a yes or no answer.

Example: Is the number n prime?

Why focus on decision problems?

Decision problems are simple: This makes it easy to develop a rigorous mathematical theory.

Decision problems are surprisingly general: Many other problems can be recast in terms of decision problems that are essentially equivalent.

Recasting other problems as decision problems

Multiplication problem: What is the product of m and n?

Multiplication decision problem: Is the kth bit of the product of m and n a one?

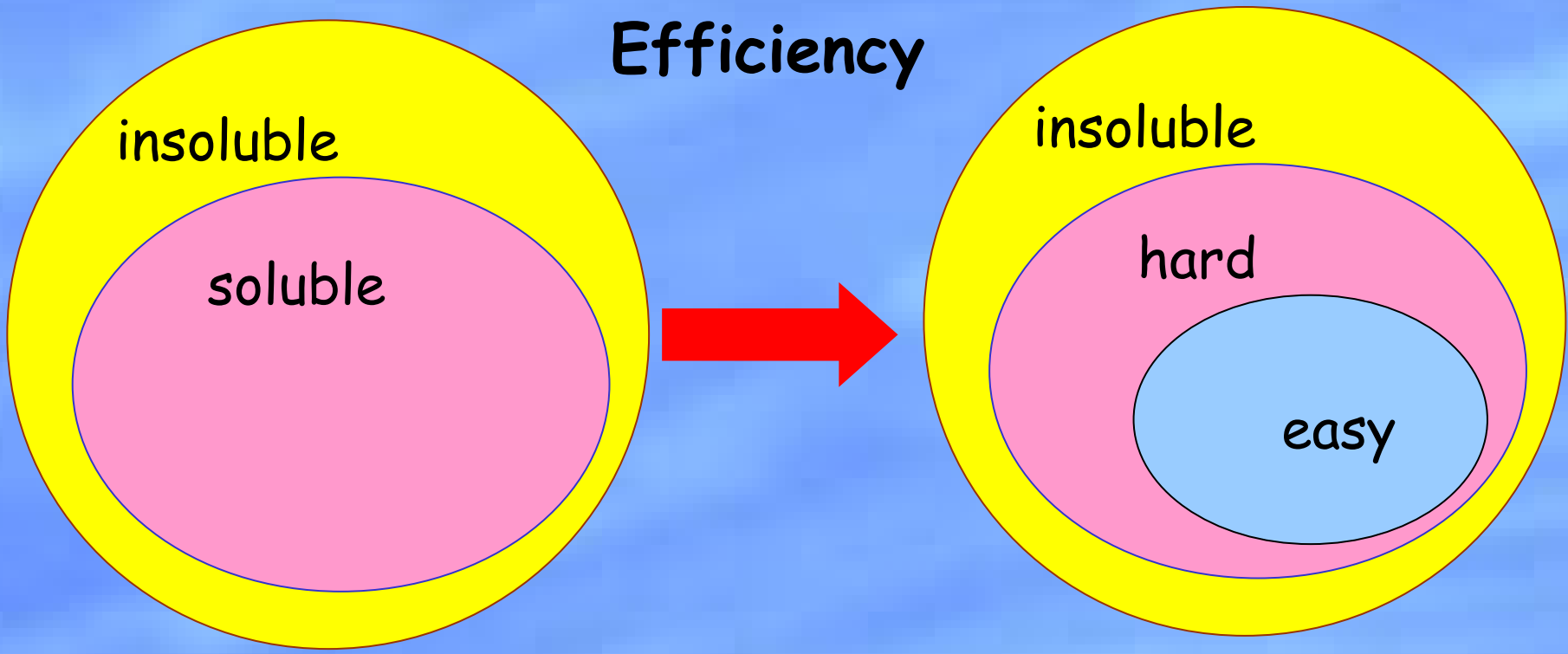
Time required to solve one of these problems is the same (to within a small overhead) as the time required to solve the other.

Factoring problem: What is the smallest non-trivial factor of n?

Factoring decision problem: Does n have a non-trivial factor smaller than k?

Time required to solve one of these problems is the same (to within a small overhead) as the time required to solve the other.

Efficiency



Nomenclature: easy = "tractable" = "efficiently computable"
hard = "intractable" = "not efficiently computable"

Definition: A problem is *easy* if there is a Turing machine to solve the problem that runs in time *polynomial* in the size of the *problem input*. Otherwise the problem is hard.

This definition is usually applied to both decision problems and more general problems.

Why polynomial-time = "easy"

It's a rule of thumb: in practice, problems with polynomial-time solutions are usually found to be easily soluble, while problems without are usually found to be rather difficult.

In practice, a super-polynomial algorithm whose running time as a function of the input size, n, is $n^{\log(n)}$ is likely preferable to a poly-time algorithm taking time n^{10000}.

The way to think about the polynomial versus non-polynomial distinction is as a *first-cut analysis*. Finer distinctions can wait until later.

Why polynomial-time = "easy"

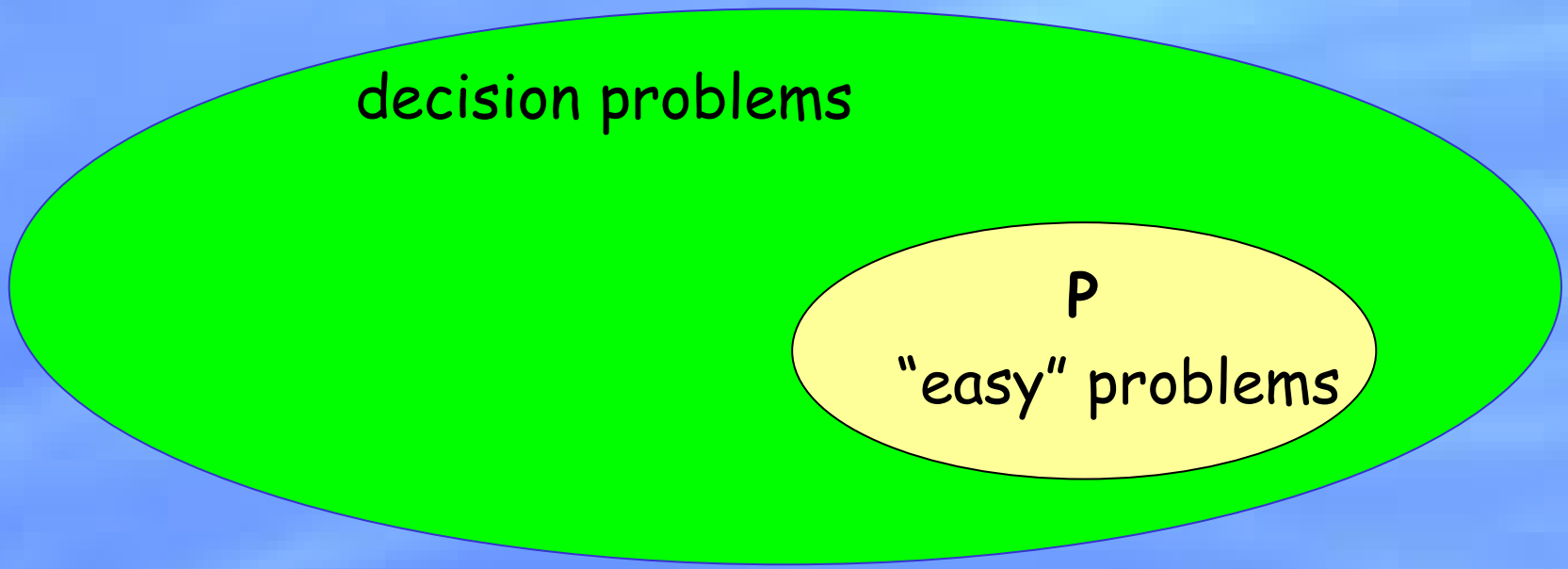
Christos H. Papadimitriou, "Computational Complexity":

"It should not come as a surprise that our choice of polynomial algorithms as the mathematical concept that is supposed to capture the informal notion of "practically efficient computation" is open to criticism from all sides."

"[…] Ultimately, our argument for our choice must be this: *Adopting polynomial worst-case performance as our criterion of efficiency results in an elegant and useful theory that says something meaningful about practical computation, and would be impossible without this simplification.*"

Our first computational complexity class: "P"

Definition: The set of all decision problems soluble in polynomial time on a Turing machine is denoted **P**.



Terminology: "Multiplication is in **P**" means "The multiplication decision problem is in **P**".

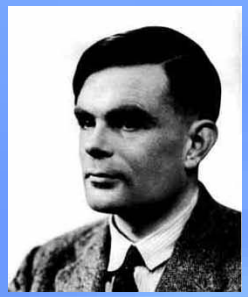
"Factoring is thought not to be in **P**" means "The factoring decision problem is thought not to be in **P**".

Technical caveat: Allowing the Turing machine to do random coin flips appears to help.

The set of all decision problems soluble on a (randomized) Turing machine is denoted **BPP**.

The strong Church-Turing thesis

Doesn't the definition of **P** depend upon the computational model used in the statement of the definition, namely, the Turing machine?



Church-Turing thesis: Any algorithmic process can be simulated on a Turing machine.

Strong Church-Turing thesis: Any physically reasonable algorithmic process can be simulated on a Turing machine, with at most a polynomial slowdown in the number of steps required to do the simulation.

***Ad hoc* empirical justification!**

The strong Church-Turing thesis implies that the problems in **P** are precisely those for which a polynomial-time solution is the best possible, in any *physically reasonable* model of computation.

The strong Church-Turing thesis

Strong Church-Turing thesis: Any physically reasonable algorithmic process can be simulated on a Turing machine, with at most a polynomial slowdown in the number of steps required to do the simulation.

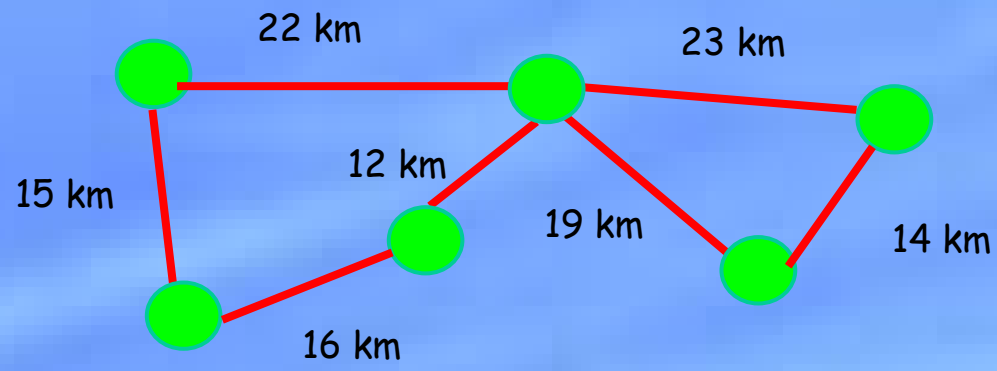
Deutsch: Maybe computers based on quantum mechanics might violate the strong Church-Turing thesis?

Nonetheless, a remarkably wide range of computational models satisfy the strong Church-Turing thesis, and it can serve as the basis for a useful theory of computational complexity.

Many important problems aren't known to be in P

Example: Factoring.

Example: The traveling salesman problem (TSP).



Goal: Find the shortest tour through all the cities.

Traveling salesman decision problem: Given a network and a number, *k*, is there a tour through all the cities of length less than *k*?

It is widely believed that neither of these problems is in **P**.

Witnesses

Determining whether 7747 has a factor less than 70 is thought to be hard.

Verifying that 61 is a factor of 7747 (= 127 x 61) less than 70 is easy: just check that 61 < 70 and use the efficient long-division algorithm you learnt in school.

The factoring decision problem has *witnesses* to yes instances of the problem – informally, solutions to the factoring problem, which can be checked in polynomial time.

It does not follow that there are easily-checkable witnesses for no instances, like "Does 7747 have a factor less than 60?"

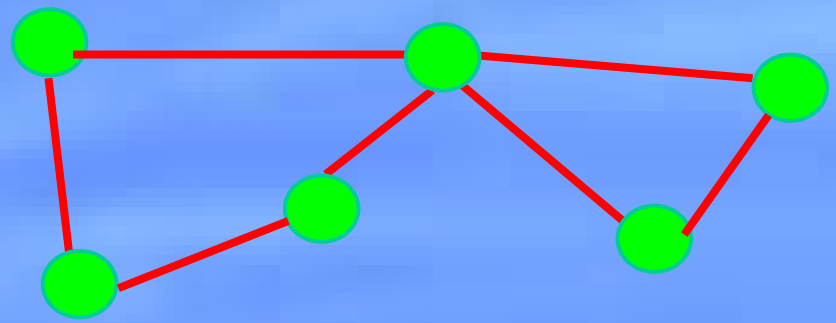
TSP seems to be hard, but also has easily-checked witnesses for yes instances.

NP

Definition: The complexity class **NP** consists of all decision problems for which yes instances of the problem have witnesses checkable in polynomial time on a Turing machine.

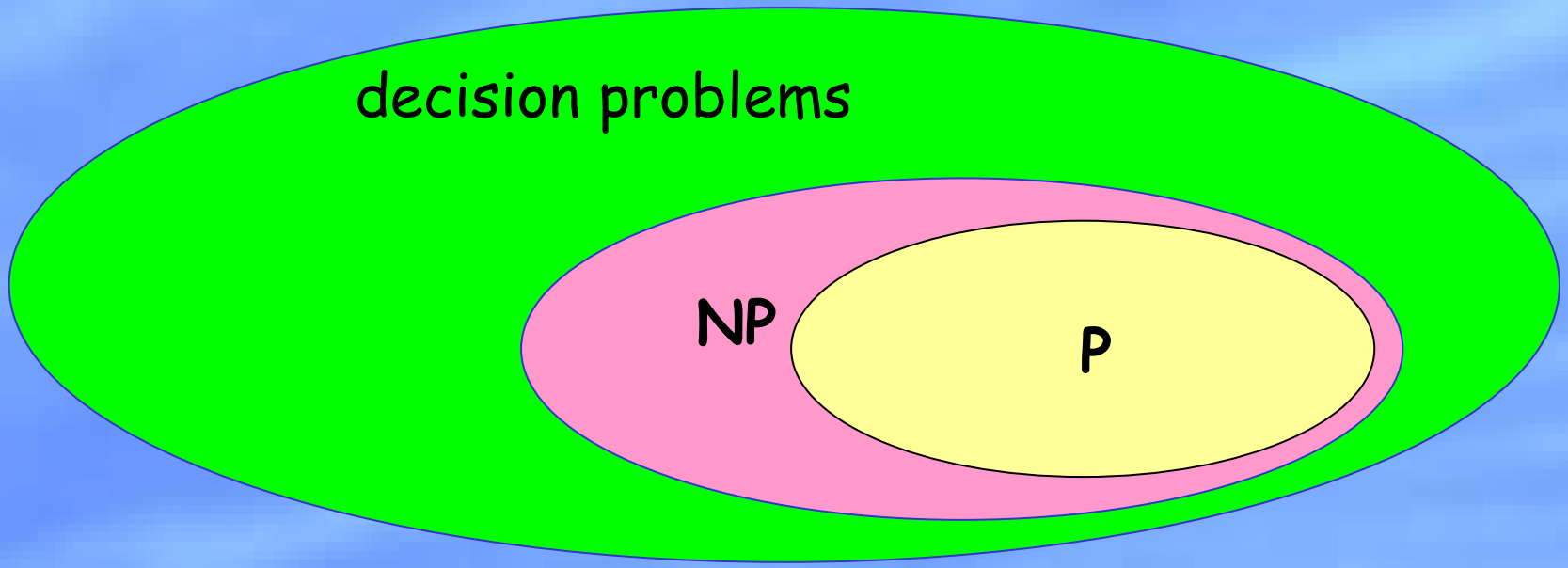
Examples: The factoring and TSP decision problems.

Example: The Hamiltonian cycle problem: is there a tour that goes through each vertex in the graph exactly once?



Many important optimization problems are in **NP** – it is often hard to find a solution, but easy to verify a solution once it is found.

The relationship of P to NP



P $\subseteq$ **NP**: Yes instances for problems in **P** can be verified in polynomial time, even without a witness.

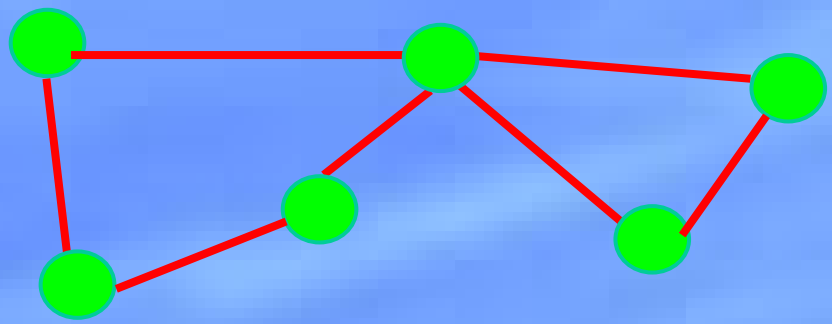
Is **P** $\neq$ **NP**?

(US) $1,000,000 dollars for a solution:
http://www.claymath.org

Why proving P ≠ NP is harder than it looks

Intuition: There is no better way of finding a solution than searching through most possible witness.

Example: The Hamiltonian cycle problem: is there a tour that goes through each vertex in the graph exactly once?

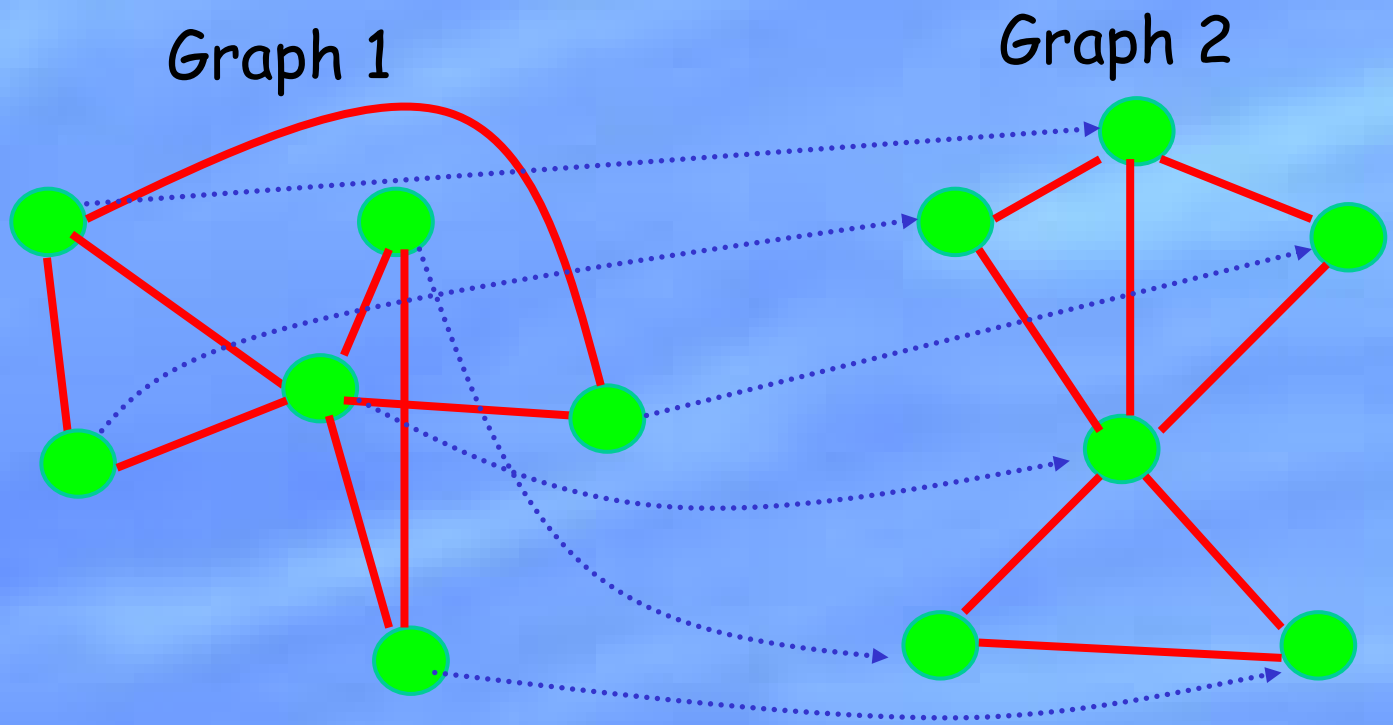


Intuition: To determine whether there is a Hamiltonian cycle, the best way is to search through all possible cycles.

Euler's problem: Is there a tour of a graph that visits each edge exactly once?

Exercise: Prove *Euler's theorem*: A connected graph contains an Euler cycle iff each vertex has an even number of incident edges. Use this to argue that Euler's problem is in P.

Example: Graph isomorphism problem



Problem: Is Graph 1 isomorphic to Graph 2?

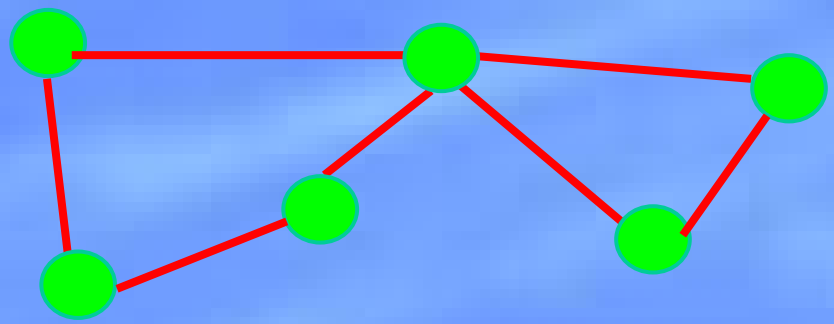
Worked exercise: Prove that the graph isomorphism problem is in NP.

Research problem: Prove that the graph isomorphism problem is not in P.

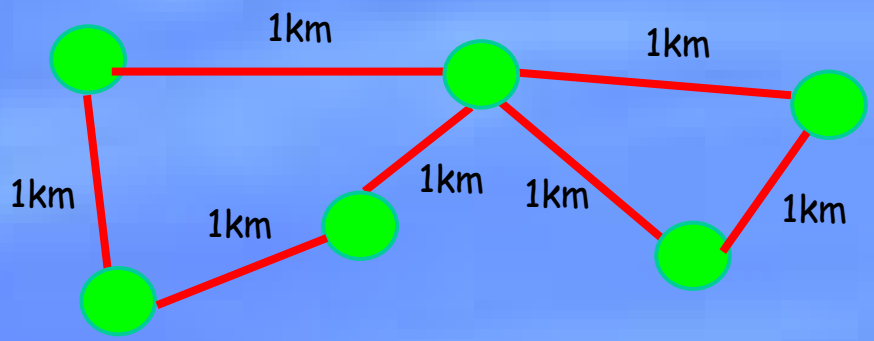
Reducibility

We say problem X is reducible to problem Y if, given an oracle to solve problem Y in one step, there is an algorithm to solve problem X in polynomial time.

Example: The Hamiltonian cycle problem can be reduced to the travelling salesman problem.

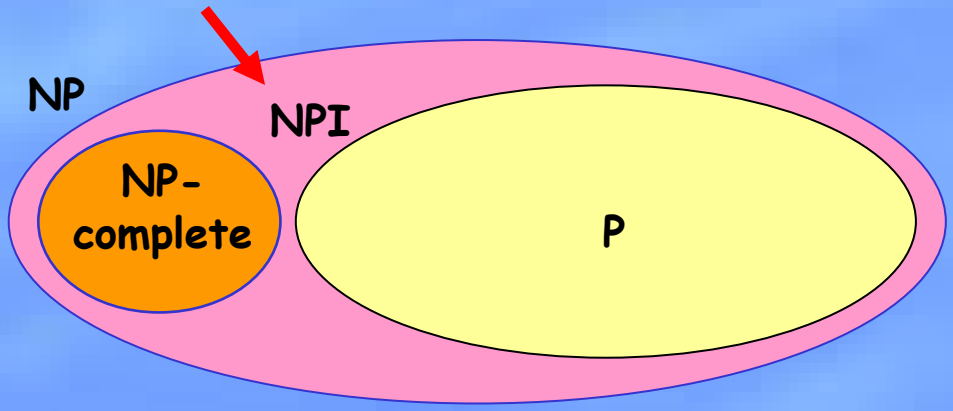


Does this graph have a Hamiltonian cycle?



Is there a tour of less than 8 kilometers?

The structure of NP



Definition: *A* problem is **NP-complete** if it is in **NP** and every other problem in **NP** reduces to it.

Not obvious that **NP-complete** problems even exist!

Cook-Levin Theorem: Provided the first example of an **NP-complete problem** – the *satisfiability problem*.

Thousands of other problems are now known to be **NP-complete**, including the travelling salesman and Hamiltonian cycle problems.

Many people believe that quantum computers won't efficiently solve **NP-complete** problems, but maybe can solve some problems in **NPI**. Candidates for **NPI** include factoring and graph isomorphism.

Research problem: Find an efficient quantum algorithm to solve the graph isomorphism problem.

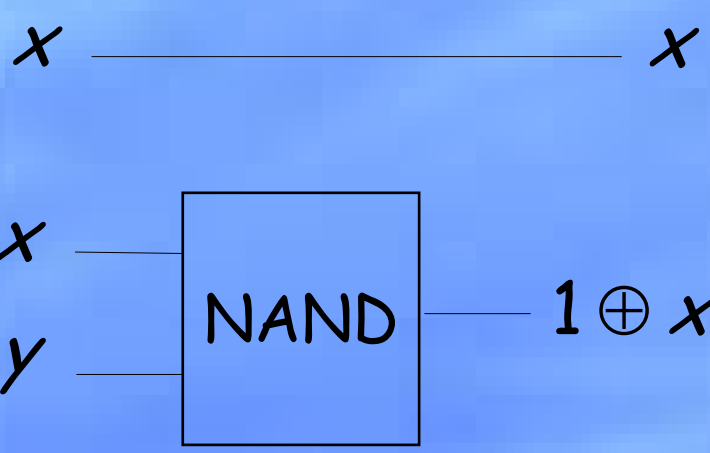
The circuit model of computation

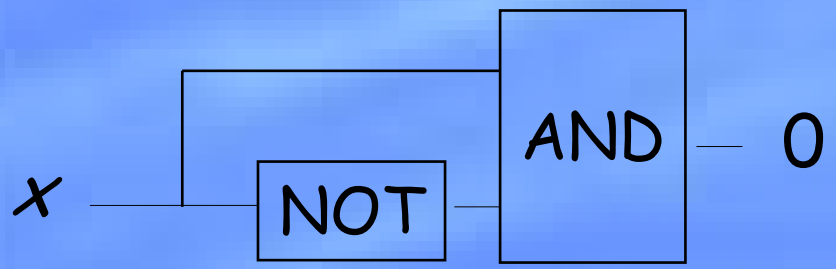
Real computers are finite devices, not infinite, like the Turing machine.

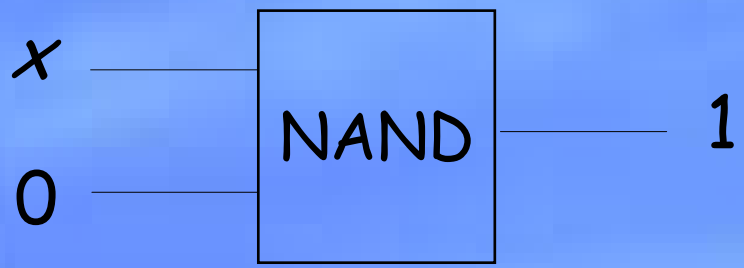
We will investigate a circuit model of computation, and explain how to define computational complexity classes in that model.

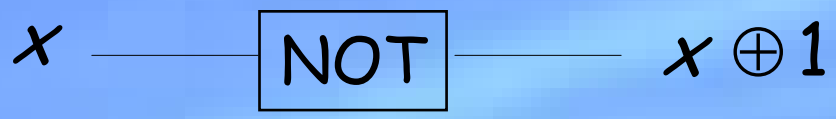
Quantum computers are most conveniently understood in terms of the quantum circuit model of computation.

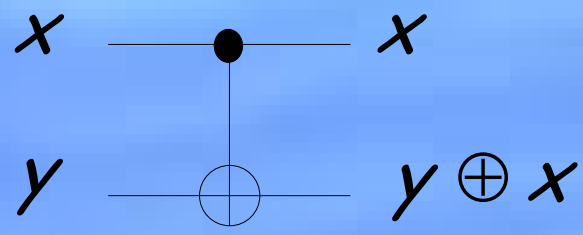
Examples of circuits











controlled-not gate (CNOT)

Basic elements

wires (memory)

one- and two-bit gates

fanout

ancilla – bits in pre-prepared states.

Universality in the circuit model

Exercise: Prove that the NAND gate can be used to simulate the AND, XOR and NOT gates, provided wires, fanout, and ancilla are available.

Exercise: Prove that the NAND gate, wires, fanout and ancilla can be used to compute an arbitrary function $f(.)$ with n input bits, and one output bit. (**Hint:** Induct on n.)

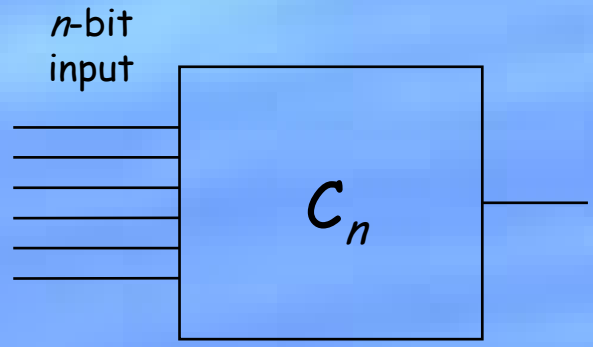
We say that the NAND gate, wires, fanout and ancilla form a universal set of operations for computation.

Exercise: Prove that there is a function $f(.)$, as above, that takes at least 2^n gates to compute.

The circuit model and complexity classes

Can we connect the circuit model to complexity classes like **P** and **NP**?

Basic idea is to introduce a family of circuits, C_n, indexed by the problem size, n.



Main point is that (modulo some technicalities) a problem is in **P** iff there is a family of circuits to solve the problem, C_n, containing a number of elements polynomial in n.

Technicalities (in brief): Our intrepid engineer needs a construction algorithm (i.e. a Turing machine) to build the circuit C_n, given n. This algorithm should run in polynomial time. Such a circuit family is called a uniform circuit family.

Irreversibility of circuit elements

From the output of the NAND gate it is impossible to determine if the input was (0,1), (1,0), or (0,0).

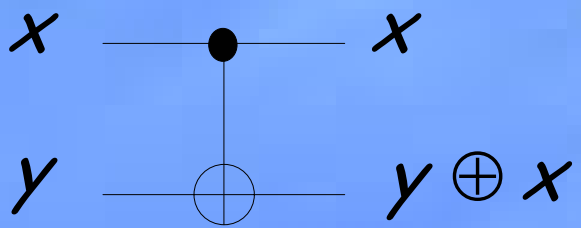
The NAND gate is irreversible: there is no logic gate capable of inverting the NAND.

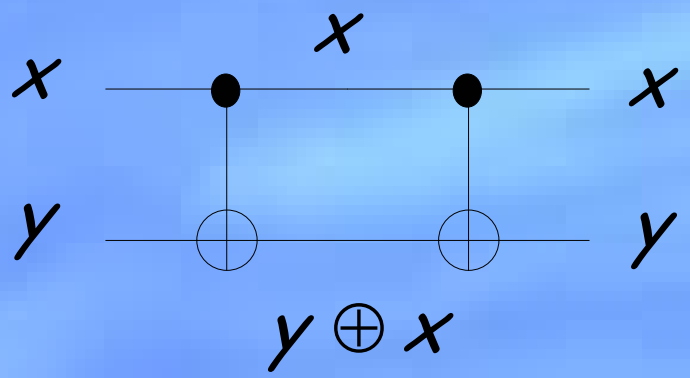
Landauer's principle: Any irreversible operation in a circuit is necessarily accompanied by the dissipation of heat.

Can we compute without dissipating heat? The trick is to compute using only reversible circuit elements!

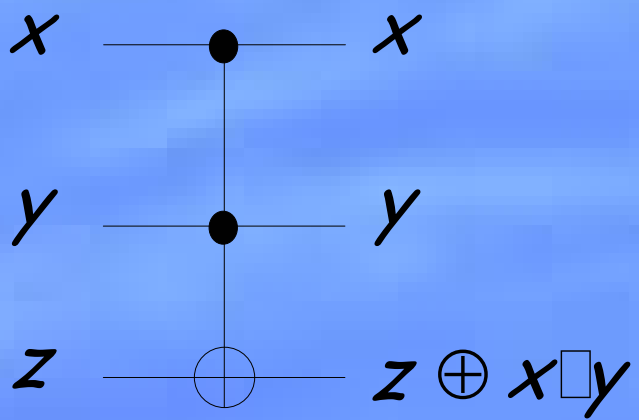
Importance to us: not heat dissipation, but the fact that quantum gates are most naturally viewed as reversible gates.

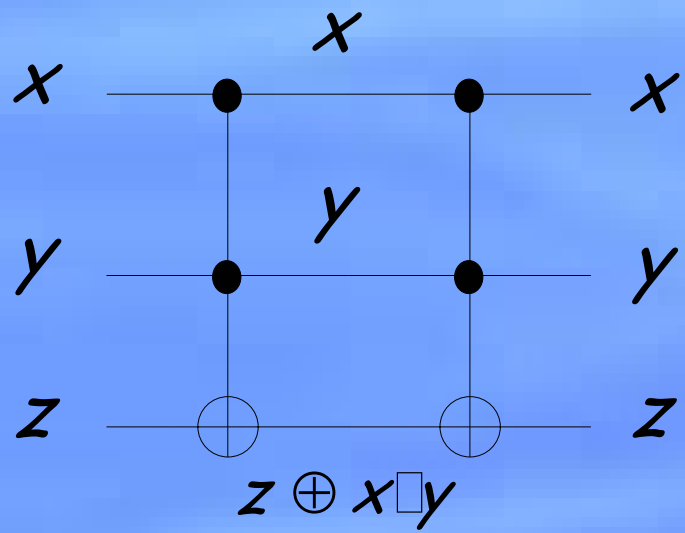
Some reversible circuit elements





The Toffoli gate (or controlled-controlled-not).

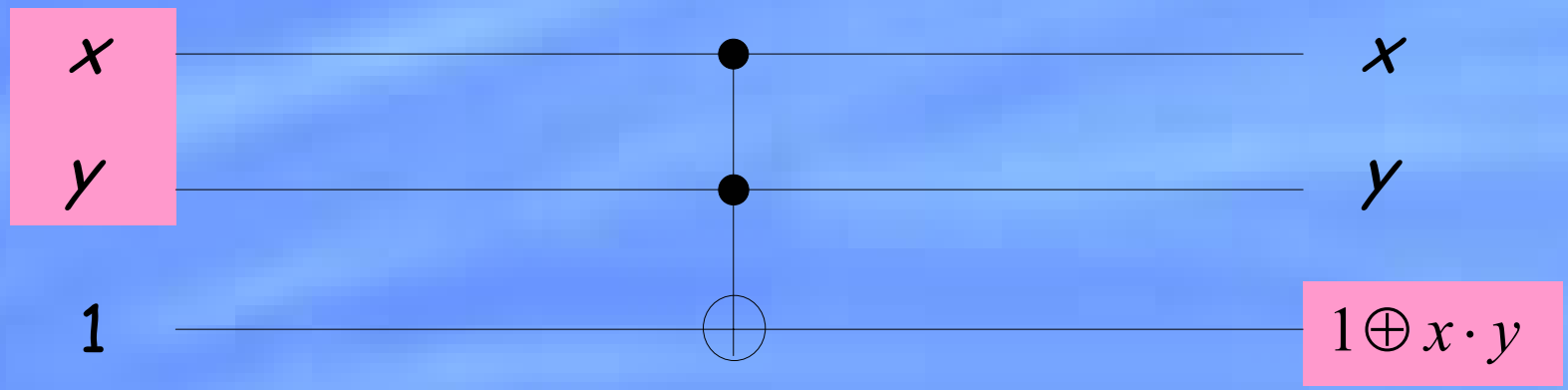




How to compute using reversible circuit elements

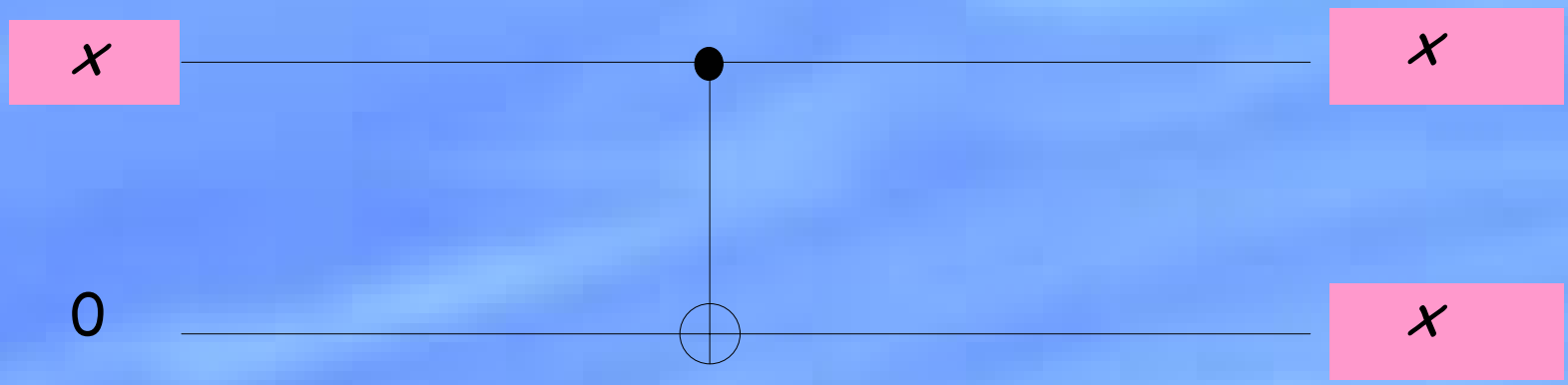
Idea: embed irreversible gates in reversible gates, making use of extra ancilla bits.

Example: The reversible NAND gate.



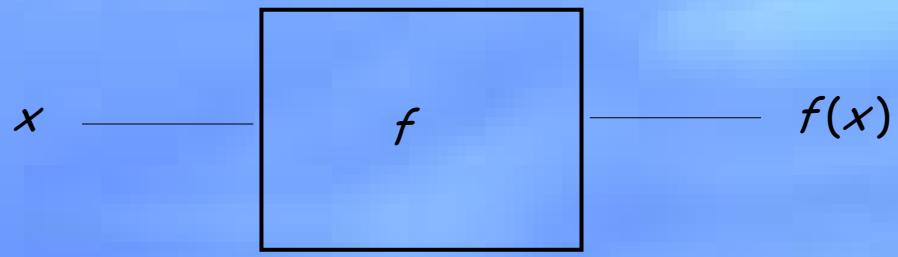
How to compute using reversible circuit elements

Example: fanout.

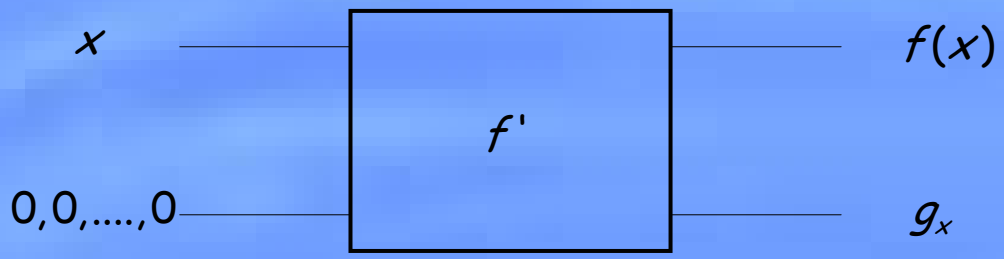


How to compute using reversible circuit elements

Original circuit: uses NAND gates, wires, fanout, and ancilla

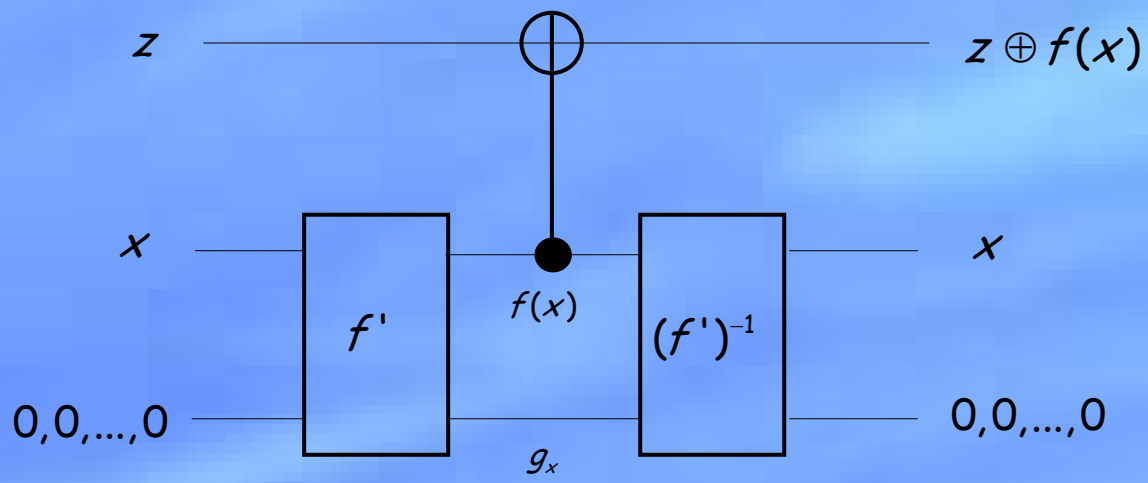


Reversible circuit: uses CNOT and Toffoli gates, wires, and ancilla



Up to constant factors the resource requirements in the two models are the same, so complexity classes like **P** and **NP** do not change in a reversible circuit model of computation.

Can we eliminate the garbage?



Canonical form: $(x,z) \rightarrow (x, z \oplus f(x))$

Complexity classes like **P** and **NP** are again unchanged.

Exercise: Show that it is impossible to do universal computation using only one- and two-bit reversible logic gates.

Digression: what can be measured in quantum mechanics?

Computer science can inspire fundamental questions about physics.

We may take an "informatic" approach to physics.

(Compare the physical approach to information.)

Problem: What measurements can be performed in quantum mechanics?

Digression: what can be measured in quantum mechanics?

"Traditional" approach to quantum measurements:
A quantum measurement is described by an *observable*, ***M***, that is, a Hermitian operator acting on the state space of the system.

Measuring a system prepared in an eigenstate of ***M*** gives the corresponding eigenvalue of ***M*** as the measurement outcome.

> "The question now presents itself – Can every observable be measured? The answer theoretically is yes. In practice it may be very awkward, or perhaps even beyond the ingenuity of the experimenter, to devise an apparatus which could measure some particular observable, but the theory always allows one to imagine that the measurement could be made."
>
> **- Paul A. M. Dirac**

The halting observable

Consider a quantum system with an infinite-dimensional state space with orthonormal basis $|0\rangle, |1\rangle, |2\rangle, \ldots$

$$M \equiv \sum_{x=0}^{\infty} h(x)|x\rangle\langle x|$$

Can we build a measuring device capable of measuring the halting observable?

Yes: Would give us a procedure to solve the halting problem.

No: There is an interesting class of "superselection" rules controlling what observables may, in fact, be measured.

Research problem: Is the halting observable really measurable? If so, how? If not, why not?