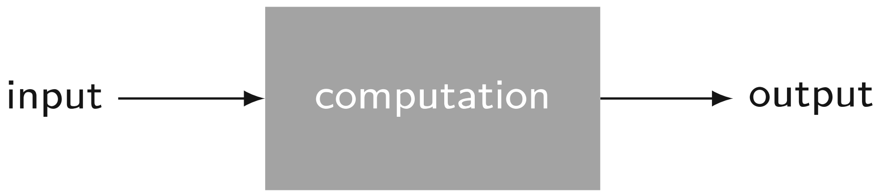
The query model of computation

When we model computations in mathematical terms, we typically have in mind the sort of process represented by the following figure, where information is provided as input, a computation takes place, and output is produced.

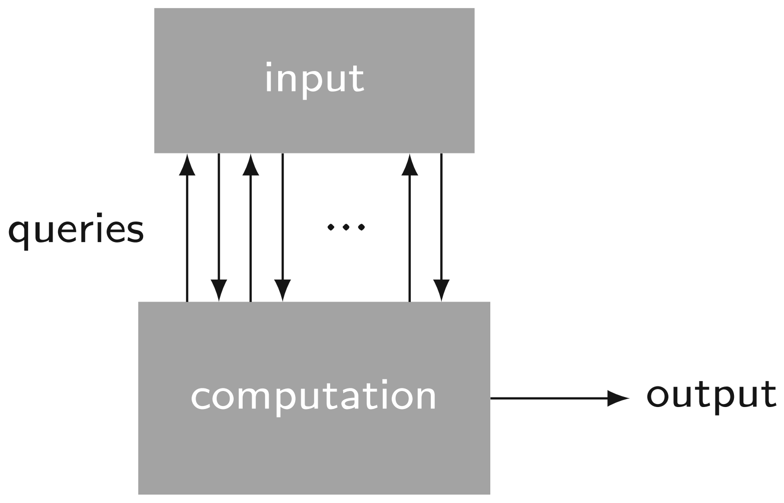


While it is true that the computers we use today continuously receive input and produce output, essentially interacting with both us and with other computers in a way not reflected by the figure, the intention is not to represent the ongoing operation of computers. Rather, it is to create a simple abstraction of computation, focusing on isolated computational tasks. For example, the input might encode a number, a vector, a matrix, a graph, a description of a molecule, or something more complicated, while the output encodes a solution to the computational task we have in mind.

The key point is that the input is provided to the computation, usually in the form of a binary string, with no part of it being hidden.

Description of the model

In the *query model* of computation, the entire input is not provided to the computation like in a more standard model suggested above. Rather, the input is made available in the form of a *function*, which the computation accesses by making *queries*. Alternatively, we may view computations in the query model as having random access to bits (or segments of bits) of the input.



We often refer to the input as being provided by an *oracle* or *black box* in the context of the query model. Both terms suggest that a complete description of the input is hidden from the computation, with the only way to access it being to ask questions. It is as if we're consulting the Oracle at Delphi about the input: she won't tell us everything she knows, she only answers specific questions. The term *black box* makes sense especially when we think about the input as being represented by a function; we cannot look inside the function and understand how it works, we can only evaluate it on arguments we select.

We're going to be working exclusively with binary strings in this lesson, as opposed to strings containing different symbols, so let's write $\Sigma = \{0, 1\}$ hereafter to refer to the binary alphabet for convenience. We'll be thinking about different computational problems, with some simple examples described shortly, but for all of them the input will be represented by a function taking the form

$$f : \Sigma^n \rightarrow \Sigma^m$$

for two positive integers n and m. Naturally, we could choose a different name in place of f, but we'll stick with f throughout the lesson.

To say that a computation makes a *query* means that some string $x \in \Sigma^n$ is selected, and then the string $f(x) \in \Sigma^m$ is made available to the computation by the oracle. The precise way that this works for quantum algorithms will be discussed shortly — we need to make sure that this is possible to do with a unitary quantum operation allowing queries to be made in superposition — but for now we can think about it intuitively at a high level.

Finally, the way that we'll measure efficiency of query algorithms is simple: we'll count the *number of queries* they require. This is related to the time required to perform a computation, but it's not exactly the same because we're ignoring the time for operations other than the queries, and we're also treating the queries as if they each have unit cost. We can take the operations besides the queries into account if we wish (and this is sometimes done), but restricting our attention just to the number of queries helps to keep things simple.

Examples of query problems

Here are a few simple examples of query problems.

- **OR.** The input function takes the form $f : \Sigma^n \rightarrow \Sigma$ (so $m = 1$ for this problem). The task is to output 1 if there exists a string $x \in \Sigma^n$ for which $f(x) = 1$, and to output 0 if there is no such string. If we think about the function f as representing a sequence of 2^n bits to which we have random access, the problem is to compute the OR of these bits.
- **Parity.** The input function again takes the form $f : \Sigma^n \rightarrow \Sigma$. The task is to determine whether the number of strings $x \in \Sigma^n$ for which $f(x) = 1$ is *even* or *odd*. To be precise, the required output is 0 if the set $\{x \in \Sigma^n : f(x) = 1\}$ has an even number of elements and 1 if it has an odd number of elements. If we think about the function f as representing a sequence of 2^n bits to which we have random access, the problem is to compute the parity (or exclusive-OR) of these bits.
- **Minimum.** The input function takes the form $f : \Sigma^n \rightarrow \Sigma^m$ for any choices of positive integers n and m. The required output is the string $y \in \{f(x) : x \in \Sigma^n\}$ that comes first in the lexicographic (that is, dictionary) ordering of Σ^m. If we think about the function f as representing a sequence of 2^n integers encoded as strings of length m in binary notation to which we have random access, the problem is to compute the minimum of these integers.

We also consider query problems where we have a *promise* on the input. What this means is that we're given some sort of guarantee on the input, and we're not responsible for what happens when this guarantee is not met. Another way to describe this type of problem is to say that some input functions (the ones for which the promise is not satisfied) are considered as "don't care" inputs. No requirements at all are placed on algorithms when they're given "don't care" inputs.

Here's one example of a problem with a promise:

- **Unique search.** The input function takes the form $f : \Sigma^n \rightarrow \Sigma$, and we are *promised* that there is exactly one string $z \in \Sigma^n$ for which $f(z) = 1$, with $f(x) = 0$ for all strings $x \neq z$. The task is to find this unique string z.

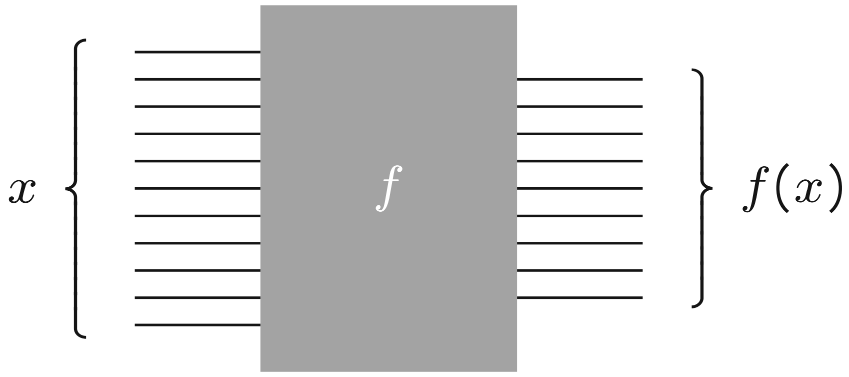
All four of the examples just described are natural, in the sense that they're easy to describe and we can imagine a variety of situations or contexts in which they might arise.

In contrast, some query problems aren't "natural" like this at all. In fact, in the study of the query model, we sometimes come up with very complicated and highly contrived problems where it's difficult to imagine that anyone would ever actually want to solve them in practice. This doesn't mean that the problems aren't interesting, though! Things that might seem contrived or unnatural at first can provide unexpected clues or inspire new ideas. Shor's quantum algorithm for factoring, which was inspired by Simon's algorithm, is a great example. It's also an important part of the study of the query model to look for extremes, which can shed light on both the potential advantages and the limitations of quantum computing.

Query gates

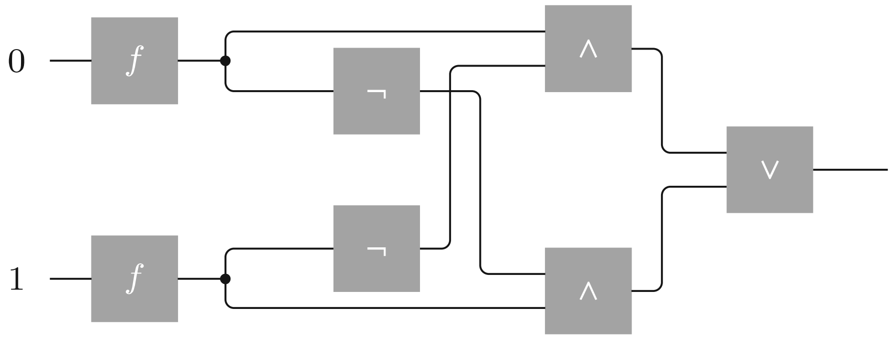
When we're describing computations with circuits, queries are made by special gates called *query gates*.

The simplest way to define query gates for classical Boolean circuits is to simply allow them to compute the input function f directly, as the following figure suggests.



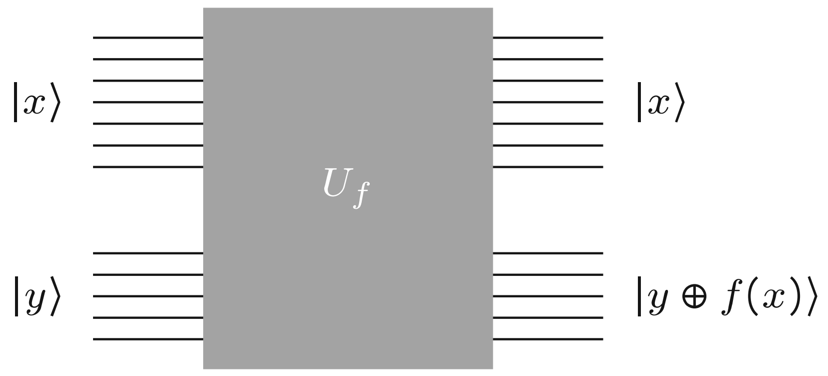
When a Boolean circuit is created for a query problem, the input function f is accessed through these gates, and the number of queries that the circuit makes is simply the number of query gates that appear in the circuit. The input wires of the Boolean circuit itself are initialized to fixed values, which should be considered as part of the algorithm (as opposed to being inputs to the problem).

For example, here's a Boolean circuit with classical query gates that solves the parity problem described above for a function of the form $f : \Sigma \rightarrow \Sigma$:



This algorithm makes two queries because there are two query gates. The way it works is that the function f is queried on the two possible inputs, 0 and 1, and the results are plugged into a Boolean circuit that computes the XOR. (This particular circuit appeared as an example of a Boolean circuit in the *Quantum circuits* lesson of the *Basics of quantum information* course.)

For quantum circuits, this definition of query gates doesn't work, because these gates will be non-unitary for some choices of the function f. So, what we do instead is to define *unitary query gates* that operate as this figure suggests on standard basis states:



Here, our assumption is that $x \in \Sigma^n$ and $y \in \Sigma^m$ are arbitrary strings. The notation $y \oplus f(x)$ refers to the *bitwise exclusive-OR* of two strings, which have length m in this case. For example, $001 \oplus 101 = 100$.

Intuitively speaking, what the gate U_f does (for any chosen function f) is to echo the top input string x and XOR the function value $f(x)$ onto the bottom input string y, which is a unitary operation for every choice for the function f. In fact, it's a deterministic operation, and it is its own inverse. This implies that, as a matrix, U_f is always a *permutation matrix*, meaning a matrix with a single 1 in each row and each column, with all other entries being 0. Applying a permutation matrix to a vector simply shuffles the entries of the vector (hence the term *permutation matrix*), and therefore does not change that vector's Euclidean norm — revealing that permutation matrices are always unitary.

It should be highlighted that, when we analyze query algorithms by simply counting the number of queries that a query algorithm makes, we're completely ignoring the difficulty of physically constructing the query gates — for both the classical and quantum versions just described. Intuitively speaking, the construction of the query gates is part of the preparation of the input, not part of finding a solution.

That might seem unreasonable, but we must keep in mind that we're not trying to describe practical computing or fully account for the resources required. Rather, we're defining a theoretical model that helps to shed light on the potential advantages of quantum computing. We'll have more to say about this point in the lesson following this one when we turn our attention to a more standard model of computation where inputs are given explicitly to circuits as binary strings.

Was this page helpful?

Yes No

Report a bug, typo, or request content on GitHub ↗.

Previous page

Next page