# Spells for Quantum Programmers: Expressive High-Level Commands in Qutes

**Simone Faro**
Department of Mathematics and Computer Science, University of Catania, Italy

**Francesco Pio Marino**
Department of Mathematics and Computer Science, University of Catania, Italy
Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, CNRS NormaSTIC FR 3638, IRIB, Rouen, France

**Gabriele Messina**
Department of Mathematics and Computer Science, University of Catania, Italy
INAF – Astrophysical Observatory of Catania, Catania, Italy

## Abstract

Classical computers are powerful but fundamentally mundane: they operate through explicit, step-by-step instructions that force programmers to think locally and procedurally. Quantum technologies, by contrast, resemble magical instruments, capable of acting on superpositions, correlations, and entire collections of states at once. Yet, despite their magical nature, quantum computers are still programmed as if they were ordinary machines, relying on long sequences of low-level operations that obscure the underlying algorithmic ideas. Magic, however, is not performed by improvisation alone: it requires a spellbook. In this paper we position Qutes as a book of spells for quantum programmers, providing high-level commands that capture complex quantum behaviour behind concise and expressive syntax. We focus on the manipulation of quantum arrays, which naturally arise in many algorithms but are notoriously cumbersome to manage at the circuit level. We introduce three new array-oriented spells in Qutes: global initialization, parallel pairwise comparison, and parallel (optionally controlled) swap. Individually, these spells encapsulate non-trivial quantum procedures; when combined, they enable programmers to express algorithms in a way that is immediate, readable, and surprisingly elegant. As a final demonstration, we show how these spells can be assembled to produce a remarkably compact implementation of Bubble Sort. While algorithmically simple, this example illustrates a broader message: when the right abstractions are available, quantum programming can feel less like engineering machinery and more like magic.

**2012 ACM Subject Classification** Software and its engineering; Theory of computation

**Keywords and phrases** Quantum programming languages, High-level abstractions, Quantum arrays

**Digital Object Identifier** 10.4230/LIPIcs.FUN.2026.15

## 1 From Muggle Machines to Wizard Computers

Classical computers have achieved extraordinary levels of performance and reliability, yet they remain fundamentally grounded in a mundane computational paradigm: every operation must be spelled out explicitly, step by step, and every algorithm is ultimately reduced to a long sequence of local instructions [22]. This model has shaped the way programmers think about computation for decades, encouraging a procedural mindset in which complex behaviour emerges only after carefully orchestrating many simple actions.

Quantum computing challenges this worldview. Quantum devices natively operate on superpositions, interference patterns, and entangled collections of states, enabling transformations that act globally on entire registers of qubits [5, 3]. From an algorithmic perspective, these machines behave less like refined classical engines and more like instruments capable of performing inherently non-local actions. In this sense, quantum computers resemble *wizard*


*machines*: they can accomplish effects that would appear magical from a classical standpoint, such as evaluating many computational paths simultaneously or correlating distant pieces of information through entanglement [17, 25].

Despite this fundamental difference, most quantum programs are still written as if quantum computers were merely faster or more exotic versions of muggle ones. Contemporary quantum programming frameworks expose developers to low-level circuit descriptions, individual gate applications, and explicit qubit management [27]. As a result, programmers are often forced to reason locally, even when the underlying algorithm is global in nature. This mismatch between the conceptual structure of quantum algorithms and their concrete implementations leads to verbose code, obscured intent, and a steep barrier to experimentation [14, 21].

Magic, however, is not performed by raw power alone. Even the most powerful magical artefacts require a disciplined use of well-defined incantations. In other words, wizard machines demand a spellbook. In the context of quantum computing, such a spellbook takes the form of a programming language that provides high-level abstractions aligned with the natural operations of quantum systems [7]. Rather than manipulating individual gates, programmers should be able to invoke expressive commands that encapsulate complex quantum behaviour behind a clear and compact syntax.

In recent years, considerable effort has been devoted to the design of quantum programming languages, with the goal of raising the level of abstraction and making quantum software development more accessible [21]. While these languages often describe themselves as high-level, many of them remain closely tied to circuit-centric models and gate-by-gate reasoning, largely reflecting the low-level programming style traditionally adopted by physicists [14]. As a consequence, programmers are still required to think in terms of individual qubits, explicit gate sequences, and manual data movement, even when working with algorithms whose structure is inherently global. This situation highlights the need to move quantum programming away from a hardware-driven, physicist-oriented perspective and toward a more genuinely algorithmic and computer-scientific view of computation [7].

It is precisely in this gap between declared abstraction and actual programming practice that Qutes [10] is positioned.[1] Qutes is a high-level quantum programming language designed to move beyond circuit-oriented thinking and align programming constructs with algorithmic intuition. Instead of exposing programmers to low-level quantum operations, Qutes provides concise primitives that seamlessly integrate classical control and quantum computation, making quantum programs more readable and enjoyable to write. [11].

In this work, we fully embrace this perspective and present Qutes as a genuine spellcasting language for quantum programmers. Our focus is on one of the most pervasive yet systematically underserved aspects of quantum programming: the manipulation of quantum arrays. Arrays arise naturally in a wide spectrum of quantum algorithms, from search and optimisation to data reordering and sampling [17, 9], yet they remain notoriously awkward to handle when expressed at the level of individual qubits and gates [15, 21]. To address this gap, we introduce a small but expressive collection of array-oriented incantations, summarised in the spellbook of Table 1, which capture common global patterns of quantum computation in a compact and declarative form.

We show that enriching Qutes with these carefully designed array spells makes it possible to express non-trivial quantum algorithms in a way that is readable, compositional, and unexpectedly elegant. While the spells do not extend the computational power of quantum

[1] Further information on Qutes, including documentation, examples, and source code, can be found at the official website `https://qutes-lang.org/`.

**Table 1** A concise spellbook of array-oriented incantations introduced in this work, summarising their purpose and pointing to the examples where each spell is presented.

| Spell | Array-level operation | Ref. |
|---|---|---|
| Rotatio Totalis | Cyclic rotation of a quantum array. | Ex. 1 |
| Patternus Revelare | Substring search over a quantum array. | Ex. 2 |
| Primaevus Array | Direct array initialization from explicit values. | Ex. 3 |
| Revelio Elementum | Indexed access to array elements (classical or quantum). | Ex. 4 |
| Imprintio Uniformis | Uniform initialization of all array elements. | Ex. 5 |
| Aleae Iacta | Random classical initialization of an array. | Ex. 6 |
| Divinatio Multiplex | Parallel comparisons on disjoint array pairs. | Ex. 7 |
| Divinatio Superposita | Superposed comparison over array indices. | Ex. 8 |
| Concordia Motus | Parallel swaps on disjoint array positions. | Ex. 9 |
| Reflexio Totalis | Array reversal via parallel symmetric swaps. | Ex. 10 |
| Ordinis Reparatio | Conditional reordering of adjacent array elements. | Ex. 11 |

machines per se, they fundamentally reshape how programmers interact with it: global structure replaces local bookkeeping, intent replaces wiring, and parallelism emerges naturally from high-level descriptions. As a concrete and perhaps surprising side effect, composing these spells yields a fully quantum odd–even Bubble Sort whose compiled circuit, for an array of $n$ elements encoded on $d$ qubits each, has depth $\Theta(nd)$ (linear in $n$ when $d = O(1)$) and size $\Theta(n^2 d)$. This result does not contradict known lower bounds [20], but it illustrates how high-level array abstractions can expose parallelism that is difficult to perceive at the circuit level, reinforcing our central message: better spells do not make machines more powerful, but they allow programmers to use their power far more effectively.

The remainder of the paper is organised as follows. Section 2 reviews the quantum computation model and the complexity measures adopted throughout the paper. Section 3 presents Qutes as a spellcasting language and illustrates its expressiveness through simple examples. Section 4 introduces three new array-oriented spells, i.e. global initialization, parallel comparison, and parallel and controlled transposition, together with their circuit-level realizations. In Section 5, these spells are composed into a highly parallel Bubble Sort, serving as a case study on expressiveness and depth/size trade-offs. Section 6 concludes with a broader discussion on abstraction and parallelism in quantum programming.

## 2 Of Superposition, Entanglement, and Other Magical Objects

Quantum computation differs from classical computation not only in its physical realisation, but in the way information is represented and manipulated [5, 3]. Classical bits take definite values, whereas quantum information is encoded in qubits, whose states are unit vectors in a complex Hilbert space. A single qubit may exist in a superposition $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, with $|\alpha|^2 + |\beta|^2 = 1$, allowing a single computational object to encode multiple classical configurations at once.

When multiple qubits are combined, their joint state lives in the tensor product space $(\mathbb{C}^2)^{\otimes n}$, whose dimension grows exponentially with the number of qubits [3]. A general state $|\Psi\rangle = \sum_{x\in\{0,1\}^n} \alpha_x |x\rangle$ encodes exponentially many basis configurations simultaneously. Unitary operators act globally on this space, transforming all components of the superposition in a single coherent step. This global action is often referred to as quantum parallelism [5, 25].

Entanglement further distinguishes quantum information from its classical counterpart. A state is entangled when it cannot be written as a tensor product of its subsystems, implying that the behaviour of individual qubits cannot be described independently [1]. Entangled

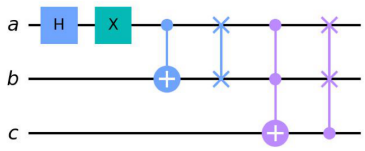


**Figure 1** Elementary quantum gates used throughout the paper. The figure illustrates representative instances of single-qubit gates (Hadamard and Pauli-X), two-qubit controlled operations such as CNOT and SWAP, and multi-qubit control and data-movement primitives, such as Toffoli and controlled-SWAP (Fredkin) gates.

registers therefore act as unified computational objects, whose properties arise from collective structure rather than from isolated components. In this sense, arrays of qubits naturally behave as indivisible, enchanted artefacts.

Measurement marks the interface between quantum and classical computation. Measuring a quantum register collapses its state to a single basis outcome, irreversibly destroying superposition and entanglement [23]. Algorithmically, measurement extracts classical information from a global quantum object, but only at the cost of discarding the richer structure that enabled quantum computation in the first place [23].

Throughout the paper we assume the standard circuit model with bounded fan-in gates and unit-cost single- and two-qubit operations [23]. Our constructions rely on a conventional gate set including Pauli-X and Hadamard gates for state preparation and superposition, CNOT and Toffoli gates for controlled and reversible logic, and SWAP and controlled-SWAP (Fredkin) gates for data movement and conditional transpositions (see Figure 1) [23]. More complex operations are obtained by composing these gates in structured patterns.

From an operational perspective, quantum computation exhibits two complementary forms of parallelism. At the circuit level, gates acting on disjoint sets of qubits can be executed simultaneously, making circuit depth a more faithful measure of execution time than gate count [23]. At a deeper level, superposition allows the same unitary transformation to be applied coherently to many basis states at once. These computational paths are not independently observable, but interfere through their amplitudes, enabling the extraction of global information through carefully designed algorithms [5, 17, 25].

To reason about the cost of quantum programs, we adopt the standard notion of circuit complexity [23]. A quantum circuit is characterised by its *size*, defined as the total number of elementary gates, and its *depth*, defined as the number of sequential layers of gates acting on disjoint qubits. While circuit size captures the overall resource consumption, circuit depth more accurately reflects the execution time in the circuit model, as gates within the same layer can be applied in parallel [23]. Throughout the paper, we therefore use circuit depth as the primary measure of time complexity, understood as the number of sequential layers of logical quantum gates assuming Toffoli and Fredkin operations as constant-depth primitives, while circuit size is used to quantify the total amount of required quantum hardware activity.

Despite this inherently global nature of quantum computation, most quantum programs are still expressed in a low-level, circuit-centric style, where arrays of qubits are treated as collections of individually addressed wires [21, 14]. This approach obscures the fact that quantum algorithms are, semantically, transformations over high-dimensional vector spaces [3]. Arrays therefore represent a crucial abstraction boundary: while their implementation relies on local gates, their natural manipulation is inherently collective.

## 3 Qutes as a Spellcasting Language

Over the past years, a rich ecosystem of quantum programming languages and frameworks has emerged, each aiming to make quantum computation more accessible and expressive [18, 6, 4, 26, 16, 2]. From an abstract perspective, these languages provide programmers with a variety of magical instruments: gates, circuits, oracles, and control structures that enable interaction with quantum hardware. However, many of these tools resemble raw magical artefacts rather than refined spells: powerful, yet demanding careful low-level manipulation and a deep familiarity with the underlying mechanics.

Circuit-based frameworks such as Qiskit, Cirq, and OpenQASM expose quantum programs directly as circuits, offering precise and fine-grained control [18, 6, 4]. In these environments, every qubit, gate, and control dependency must be specified explicitly, and even simple algorithmic ideas quickly expand into long sequences of low-level operations. Programming in this style often feels like assembling a spell one rune at a time: exact and expressive for experts, but verbose and error-prone when reasoning at the algorithmic level, especially for operations that are inherently global.

Other quantum programming languages, including Q#, Quipper, and Silq, attempt to raise the level of abstraction by introducing classical control flow, functional constructs, or richer type systems [26, 16, 2]. Nevertheless, despite their declared high-level nature, many of these approaches remain closely tied to the circuit model. Their abstractions often leak gate-level details, forcing programmers to reason explicitly about ancilla management, reversibility, and data movement even when manipulating structured quantum data.

At the current stage of the field, much of what is described as "high-level" quantum programming is high-level primarily from a physicist's perspective. From a computer scientist's point of view, these abstractions often remain uncomfortably close to the circuit description, exposing details that would traditionally belong to a low-level implementation layer. This reflects the historical roots of quantum computing rather than a limitation of the models themselves, and highlights the need for a language that is indecently computer-scientific. What is missing is not computational power, but a spellbook: a language offering concise incantations that directly capture common quantum algorithmic patterns, allowing programmers to operate on enchanted objects rather than on individual components.

Against this background, Qutes [11, 10] takes a deliberately different approach. Rather than exposing quantum computation through circuits or gate sequences, Qutes is designed as a high-level, domain-specific language whose goal is to align programming constructs with the way quantum algorithms are conceived. In the magical metaphor adopted throughout this paper, Qutes plays the role of a spellbook: a curated collection of expressive incantations that allow programmers to invoke complex quantum behaviour without assembling it manually from low-level components, making programming in Qutes *fun by design*.

At its core, Qutes treats quantum data as first-class objects. Quantum variables, registers, and arrays are manipulated through concise language primitives that abstract away circuit wiring, gate placement, and register bookkeeping. Classical and quantum computation coexist naturally: classical control structures such as conditionals and loops interact seamlessly with quantum data, while implicit casting and measurement rules govern transitions between the quantum and classical realms. We do not delve here into the formal definition of Qutes types and core structures, and refer the interested reader to [10] for a detailed presentation.

A defining feature of Qutes is its emphasis on expressiveness over verbosity. Common quantum patterns, such as state preparation, conditional execution, or structured iteration over quantum data, are captured by compact constructs whose semantics correspond to

**Table 2** A playful comparison of quantum programming languages through a magical lens. Circuit-centric frameworks provide powerful but low-level tools, requiring programmers to manually assemble complex behaviour. Qutes aims to offer higher-level spells.

| Language | Level of Magic | Primary Tool | Composition | Friendliness |
|---|---|---|---|---|
| Qiskit | Low | Raw runes (gates) | Manual | Expert-only |
| Cirq | Low | Hardware-tuned runes | Manual | Expert-only |
| Q# | Medium | Structured rituals | Partial | Intermediate |
| Quipper | Medium | Functional scrolls | Explicit | Advanced |
| Silq | Medium–High | Uncomputate charms | Semi-automatic | Advanced |
| Qutes | High | Semantic constructs | Compositional | Wizard-friendly |

well-defined quantum operations. While the underlying execution model remains fully unitary and compatible with standard circuit semantics, these details are intentionally hidden behind higher-level abstractions. Qutes programs are compiled into executable quantum circuits, ensuring compatibility with existing simulators and hardware backends, while freeing programmers from circuit-centric reasoning during development.

To give a concrete taste of Qutes' expressive power, we present two simple examples in which non-trivial quantum behaviour emerges from compact, spell-like commands.

▶ **Example 1** (Rotatio Totalis)**.** Cyclic rotations of quantum registers are a common building block in quantum algorithms, yet their circuit-level realization typically requires a non-trivial sequence of controlled swaps or permutation networks. In Qutes, the same operation is expressed as a single high-level command acting on an entire register [24, 13]. Unlike the corresponding circuit-based implementation, which requires an $O(n)$-step manipulation of individual components, the following Qutes spell treats the register as a single enchanted entity and operates in constant depth [13]:

```
qustring s = "110111";
s « k;
print s;
```

▶ **Example 2** (Patternus Revelare)**.** Grover's algorithm [17] is a canonical example of quantum speedup, but its implementation is often obscured by the explicit construction of oracles, diffusion operators, and ancillary registers. In Qutes, substring verification based on Grover search can be written in a compact and declarative style, where these details are encapsulated within high-level constructs [10]. Compared to the corresponding circuit description, the following Qutes program reads as a direct invocation of a search spell over a structured quantum space:

```
qustring a = "1110111";
if("01" in a){
  print "Substring Found!";}
else {
  print "Substring Not Found!";}
```

Taken together, these examples illustrate a recurring pattern. While circuit-level descriptions faithfully capture the mechanics of quantum computation, they often obscure the structure of the algorithm itself. Qutes allows programmers to express quantum routines at the level of conceptual operations, delegating low-level details to the compiler and runtime. This expressive gap motivates the introduction of the array-oriented spells presented in the remainder of the paper, which show how a small set of high-level constructs suffices to encode non-trivial quantum algorithms.

## 4 Three New Spells for Quantum Arrays

Quantum arrays play a central role in quantum algorithm design, providing a natural abstraction for structured data and intermediate computational states. While arrays are ubiquitous and conceptually simple in classical computation, their quantum counterparts introduce additional challenges related to reversibility, entanglement, and coherent data access [15, 19]. These difficulties arise from the mismatch between the global nature of quantum states and the inherently local character of circuit-level descriptions.

Existing quantum programming languages and frameworks offer mechanisms to manipulate quantum registers that resemble arrays [18, 6, 4], but these mechanisms often expose low-level circuit details. As a result, even simple array transformations tend to unfold into verbose gate-level constructions, obscuring the underlying algorithmic intent.

Qutes adopts a different approach by treating quantum arrays as first-class enchanted objects. Programmers interact with arrays through high-level spells that describe global transformations, while the circuit-level realization is deliberately hidden [24, 13]. This shift from procedural qubit manipulation to declarative, array-oriented operations allows quantum programs to be expressed in terms of semantic intent rather than implementation detail.

In Qutes, quantum arrays are provided as a native language abstraction and can be instantiated directly through a dedicated construct. This allows programmers to work with structured quantum collections from the outset, much like declaring an array in a classical language, while avoiding explicit low-level register management.

The first fundamental operation concerns the initialization of a quantum array from a finite sequence of elements. In a classical, muggle-oriented setting, this task naturally unfolds as a linear-time procedure that assigns values to array cells one by one. In Qutes, by contrast, the initialization of an array from a prescribed sequence is expressed as a single high-level command, semantically executed in constant time. From a magical perspective, this corresponds to conjuring an enchanted artefact directly in its intended configuration, rather than assembling it piece by piece.

▶ **Example 3** (Primaevus Array)**.** Suppose that we want to declare and initialise a quantum array in a single step, assigning a specific quantum value to each element at creation time. In Qutes, this can be achieved directly at declaration, as below:

```
quint[] v = [1q, 3q, 0q, 2q];
```

The array is initialised as an atomic operation, with each element prepared independently, allowing the overall initialisation to be performed in constant depth.

A second fundamental operation is access to a specific array position. Classically, reading the content of the cell at position $i$ is treated as a primitive whose cost is hidden by the memory abstraction. In the quantum setting, however, data access must respect reversibility and unitarity constraints, making the notion of random access considerably more delicate [15]. Unlike QRAM-based models, the access mechanism adopted in Qutes does not assume specialised hardware support, but is realised entirely within the standard circuit model.

In Qutes, access to the content stored at position $i$ is realised through a simple two-step spell. First, the array is cyclically rotated so that the element originally at position $i$ is brought to the front. Then, the content of the first cell is transferred into a dedicated quantum variable, without exposing the rearrangement of the remaining elements. This decomposition mirrors the programmer's intent while preserving a clean and uniform abstraction.

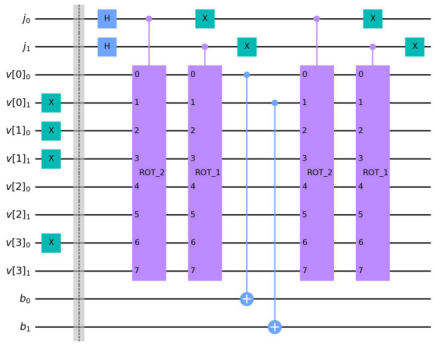


**Figure 2** Quantum circuit for initializing and accessing a quantum array. Four quint elements, encoded on two qubits each, are initialized in constant depth. A logarithmic-size index register $|j\rangle$, controls conditional cyclic rotations that move $|v[j]\rangle$ into position $|v[0]\rangle$, from which it is transferred to the output register $b$. A final uncomputation stage restores the original array configuration.

▶ **Example 4** (Revelio Elementum)**.** Suppose that we want to access elements of a quantum array using both classical and quantum indices. In Qutes, array access can be performed uniformly, regardless of whether the index is fixed, classically parametrised, or in superposition:

```
quint a = v[2];
quint i = 0,2q;
quint b = v[i];
quint j = 0..3q;
quint c = v[j];
```

The same access mechanism applies in all cases: the selected array element is coherently retrieved and assigned to a quantum variable, enabling both deterministic and superposed access without changing the programming model.

To appreciate the expressive gap bridged by Qutes, it is instructive to briefly step back into the muggle world of circuit-based quantum computation, where the same operations must be realised explicitly at the level of gates and wires. Figure 2 illustrates a circuit for the initialisation of a small quantum array and a parametric access to its elements.

The array consists of four logical elements, each encoded using two qubits[2], for a total of eight data qubits arranged in the register $|v\rangle$. The initialization phase (Example 3) appears at the beginning of the circuit and is performed in constant depth, as single-qubit gates act on disjoint subsets of qubits and can therefore be executed simultaneously [23].

Access to the array (Example 4) is controlled by the index register $|j\rangle$, whose size is logarithmic in the array length. Conditional cyclic rotations shift the array so that the selected element is moved into the first position $|v[0]\rangle$, from which its content is transferred to the output register $|b\rangle$. A final uncomputation stage restores the array to its original configuration, ensuring reversibility and preventing the accumulation of unwanted entanglement [15, 23].

From a complexity standpoint, indexed array access is realised with polylogarithmic depth. Each cyclic rotation can be implemented with depth $O(\log n)$ and size $O(n)$ [24, 13], and since $O(\log n)$ such rotations are applied, the overall depth of the access operation is $O(\log^2 n)$, while the total circuit size is $O(n \log n)$.

[2] The number of qubits allocated to values of type `quint` is a compile-time parameter that is fixed prior to execution, allowing the bit-width of quantum integers to be chosen independently of the program logic.

## 4.1 The Summoning Spell: Global Initialization

The most elementary interaction with a quantum array consists in bringing it into existence with a well-defined initial content. In a classical, muggle-oriented setting, array initialization typically proceeds by assigning values to individual cells, resulting in a procedure whose cost grows linearly with the array size. While a similar strategy is conceptually possible in the quantum setting (see Example 3), it quickly becomes impractical as the number of elements increases, due to the constraints imposed by reversibility and coherent state preparation [23].

For this reason, Qutes distinguishes between explicit initialization from a full sequence of values and a more compact form of global initialization. The former is convenient for small arrays or illustrative examples, but becomes unwieldy when dealing with large quantum data structures. To address this limitation, Qutes provides a *summoning spell* that allows an entire array to be initialized with all elements set to a value specified at declaration time.

▶ **Example 5** (Imprintio Uniformis)**.** Suppose that we want to initialise a quantum array by replicating the same quantum value across all its positions. In Qutes, this can be expressed by first preparing a quantum value and then using it as a template for the entire array:

```
quint a = 2q;
quint[] v = [a]*n;
```

The replication is interpreted as a conceptual broadcasting of the quantum value to the array elements. Importantly, this does not involve physical cloning of an unknown quantum state, but rather the coherent propagation of the source state to the target registers [23].

At the language level, this operation appears as a constant-time instruction: a single summoning spell brings the entire array into existence in its intended configuration. At the circuit level, however, the implementation is non-trivial. Conceptually, the value is propagated across the array through a logarithmic number of stages, arranged in a tree-like pattern, where at each stage the number of initialized elements is doubled. After $O(\log n)$ stages, all $n$ array elements have been coherently prepared [23]. This illustrates a recurring theme in Qutes: the separation between semantic simplicity and circuit-level complexity.

Uniform initialization is not the only useful global pattern. Qutes also supports the summoning of arrays whose elements are drawn from a genuinely random classical configuration, obtained by exploiting intrinsic quantum randomness. In this case, each array element is first prepared in a uniform superposition and then measured, yielding a classical value sampled uniformly at random [23].

▶ **Example 6** (Aleae Iacta)**.** Suppose that we want to initialise an array with values drawn uniformly at random, using quantum randomness rather than pseudo-random classical generators. This mode of initialization is particularly useful in randomized algorithms, sampling-based protocols, and probabilistic exploration of large state spaces:

```
quint a = 0;
quint[] v = [a]*k;
hadamard v;
measure v;
```

Each array element is measured independently, yielding a uniformly random classical bitstring while preserving a compact and declarative programming style.

When working directly within the circuit model, uniform initialization must be implemented explicitly through a structured propagation of quantum information across the array [23]. Although no physical cloning takes place, the circuit realises a coordinated replication of

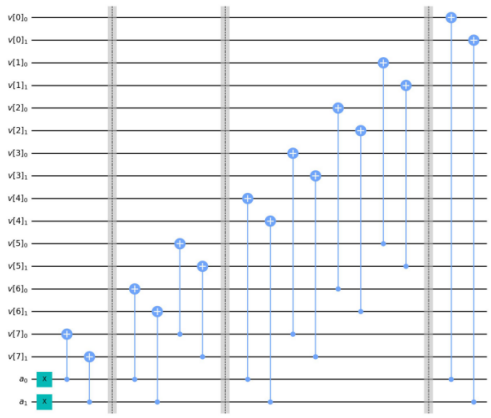

**Figure 3** Circuit-level implementation of uniform array initialization in the circuit-based model. Starting from a single register prepared in the desired initial value, the circuit propagates this value across the array through a tree-like sequence of controlled operations. At each stage, the number of initialized elements is doubled, yielding a logarithmic-depth construction with linear size.

a known value through reversible data movement and independent state preparation (see Figure 3) [23]. The resulting construction has logarithmic depth in the array size and linear circuit size [23]. This non-trivial orchestration of gates and ancillae is entirely hidden when the same operation is invoked in Qutes as a single summoning spell.

The following index-based initialisation constructs a quantum array whose $i$-th element encodes the binary representation of its position. Assuming an array of length $n$ and elements represented on $d = \lceil \log n \rceil$ qubits, this can be expressed declaratively in Qutes as:

```
quint[] v = [0..n-1];
```

At the circuit level, this operation admits a particularly efficient implementation. Since the value of each index $i$ is known at compile time, the compiler generates, for each array cell, a constant-depth state-preparation circuit consisting solely of Pauli-X gates applied to the qubits corresponding to the 1-bits of the binary encoding of $i$ [23]. All such preparations act on disjoint registers and can therefore be executed fully in parallel. The resulting circuit has depth $O(1)$ and size $O(n \log n)$, reflecting the need to initialise $n$ registers of logarithmic width, while achieving maximal parallelism in time.

This construction generalises immediately to pattern-based initialisation of the form $v[i] = f(i)$, where $f$ is a fixed function mapping indices to values encoded on $d$ qubits. In Qutes, such an initialisation can be written schematically as:

```
quint[] v = for i in [0..n-1] yield f(i);
```

Here the index $i$ again plays the role of a static parameter rather than a quantum datum. At the circuit level, the compiler instantiates $n$ independent copies of a reversible circuit $C_f$ that prepares the state $|f(i)\rangle$ from $|0\rangle$, each specialised with classical constants depending on $i$ [23, 8]. Because these circuits act on disjoint subsets of qubits, they can be executed in parallel, yielding an overall depth equal to $\text{depth}(C_f)$, independent of the array size $n$, and total size $O(n \cdot \text{size}(C_f))$.

## 4.2 The Divination Spell: Seeing Order in Chaos

Once a quantum array has been summoned, possibly in a highly entangled or randomly initialized state, the next natural task is to extract meaningful information from it. In the classical, muggle world, this typically involves scanning the array element by element, computing predicates or aggregations through explicit iteration. In the quantum setting, however, such a procedural approach clashes with the principles of superposition and reversibility [23].

In Qutes, this challenge is addressed through what we refer to as a *divination spell*: a class of high-level operations designed to reveal global properties of a quantum array without exposing or manipulating its individual elements explicitly. From a magical perspective, divination does not alter the enchanted object itself, but rather uncovers latent structure hidden beneath apparent disorder.

Typical examples of divination spells include predicates that test whether all elements satisfy a given condition, whether at least one element exhibits a particular property, or whether the array contains repeated or extremal values. While their classical realizations rely on sequential inspection, in Qutes such queries are expressed as single semantic operations acting on the array as a whole [14, 21].

▶ **Example 7** (Divinatio Multiplex)**.** Suppose that we want to compare multiple pairs of elements of a quantum array in parallel, using different comparison patterns over disjoint subsets of indices. In Qutes, such collective comparisons can be expressed declaratively, leaving the coordination of parallel operations to the underlying execution model:

```
quint[] res = [0q]*v.length;
for i in [0..n-2]
    res[i] = (v[i] > v[i+1]);
for i in [0..n-2:2] parallel
    res[i] = (v[i] < v[i+1]);
```

Each loop performs a family of independent comparisons simultaneously, producing a classical array of results. From a magical perspective, the array is inspected through a set of divination spells that reveal relational properties without requiring explicit, element-by-element control.

At the circuit level, these spells are realized through coherent aggregations of information, relying on controlled operations and ancillary qubits to accumulate partial results. The resulting constructions may exhibit non-trivial depth and width, yet this complexity remains entirely hidden from the programmer, who interacts only with the declarative query form [23].

The divination spell thus exemplifies the ability of Qutes to impose order on quantum chaos: it allows to reason about global structure and collective properties, even when the underlying data originate from random initialization or exist in quantum superposition [5, 17].

The structure of the code fragment above highlights how Qutes exposes parallelism at the semantic level while leaving its concrete realization to the underlying circuit. In the first loop, the comparisons `(v[i] > v[i+1])` operate on overlapping segments of the array and therefore cannot all be executed simultaneously. Nevertheless, they can be grouped into two disjoint sets of independent comparisons, allowing the corresponding circuit to be scheduled in two parallel layers. By contrast, the comparisons performed in the second loop, `(v[i] < v[i+1])`, act on pairwise disjoint array positions. As a result, all these operations can be executed concurrently within a single circuit layer, yielding a fully parallel evaluation step. To relieve the developer of manual verification, the `parallel` keyword mandates classical array indexing and statically checks for single-layer parallelization, raising a compilation error if either condition is unmet. From a divinatory perspective, the spell first resolves partially entangled omens in successive waves, and then unveils a set of independent signs all at once.

▶ **Example 8** (Divinatio Superposita)**.** Suppose that we want to lift a comparison over array elements from a position-wise evaluation to a quantum-superposed one. Rather than comparing elements independently, the index register is prepared in superposition so that the comparison predicate is applied coherently to all relevant pairs of positions at once. The collective outcome of these comparisons is then accumulated into a single quantum variable:

```
int len = v.length;
quint i = [0..len];
qubit res = (v[i] > v[i+1]);
```

Here the comparison is evaluated simultaneously for all indices encoded in the superposition of `i`, and the resulting information is compressed into the state of `res`. Leveraging the cyclic access method (Section 4), the algorithm wraps around to compare the final element with the initial one. From a magical perspective, a single divination spell reveals a global property of the array, rather than inspecting each position individually [5, 17].

Formally, any predicate $p(v, i)$ over the array $v$ and index $i$ can be coherently evaluated so that the resulting configuration of the register $|i\rangle\,|res\rangle$ takes the form

$$|i\rangle\,|res\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle\,|p(v,i)\rangle\,,$$

where the second register is a single qubit. Measuring this qubit does not reveal which individual comparisons succeeded, but whether the predicate holds within the computation [23].

The predicate `(x > y)` provides a high-level abstraction for comparing two registers and determining whether one encodes a value greater than the other. In Qutes, this operation is invoked as an atomic instruction, yet its implementation corresponds to a structured quantum circuit that evaluates the comparison reversibly, without intermediate measurements [23].

Given two registers $|x\rangle$ and $|y\rangle$ encoding integers on $d$ qubits, the comparison is performed according to the standard lexicographic rule on binary representations. Formally, the output qubit $|out\rangle$ is flipped if and only if there exists an index $i$ such that $x_i = 1$, $y_i = 0$, and all more significant bits satisfy $x_j = y_j$ for $j > i$. This condition is evaluated coherently by cascading controlled operations that test local inequalities while masking contributions from less significant bits once a more significant difference has been detected [8].

At the circuit level, this logic is realised through a sequence of reversible comparison blocks, each acting on a corresponding pair of qubits from $|x\rangle$ and $|y\rangle$ and contributing conditionally to the output register. The circuit coherently accumulates the result into a single qubit without destroying superposition or entanglement, making the predicate compatible with superposed inputs and higher-level quantum control flow [23]. Figure 4 illustrates this construction for the case $d = 3$, but the same pattern extends to arbitrary register sizes.

From a complexity standpoint, the comparison circuit for $d$-qubit registers has linear size $O(d)$, since each bit position contributes a constant number of gates. Its depth is also $O(d)$, reflecting the sequential dependency imposed by bit significance. While this low-level cost is unavoidable in the circuit model, Qutes encapsulates it behind a single comparison spell, allowing to reason in terms of algorithmic intent rather than gate-level mechanics [23].

Figure 5 shows the circuit-level realisation of a Qutes instruction that compares, in parallel, two disjoint pairs of elements in a quantum array. In the depicted setting, the array has four elements encoded on two qubits each, and the program evaluates the predicates `(v[0] > v[1])` and `(v[2] > v[3])`, storing their outcomes in the result register $|res\rangle = |res_0 res_1\rangle$. Since the comparisons act on disjoint subsets of the data register, they can be executed simultaneously by instantiating independent copies of the `gt` subcircuit and scheduling them in the same circuit layers.

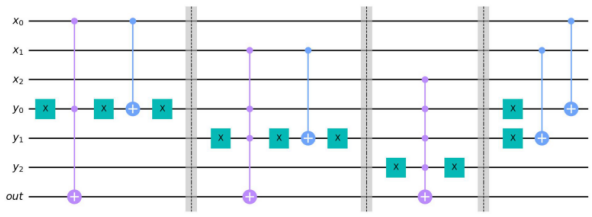


**Figure 4** Quantum circuit implementing the > (greater-than) predicate for two quantum registers. The circuit compares two registers encoding binary values by evaluating the lexicographic ordering of their bits and coherently accumulates the result into a single output qubit. While the figure shows the case of 3-qubit registers, the construction generalises directly to registers of arbitrary size $d$.

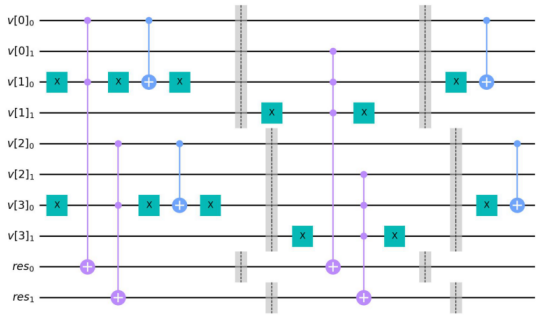


**Figure 5** Parallel circuit implementing two instances of the > predicate on a quantum array. The circuit compares the disjoint pairs $(v[0], v[1])$ and $(v[2], v[3])$ in parallel, storing the outcomes in separate result qubits. While the figure shows an array of four elements encoded on two qubits each, the construction generalises to larger arrays and wider registers, with linear size and constant-depth overhead for parallel comparisons.

This construction generalises directly. For an array whose elements are $d$-qubit registers, each comparison > can be implemented with size $O(d)$ and depth $O(d)$ using a lexicographic cascade. If $p$ comparisons are scheduled over pairwise-disjoint array positions, the resulting circuit has size $O(p\,d)$ while preserving depth $O(d)$, as all instances can be executed in parallel. Thus, parallel comparison spells scale linearly in size with the number of compared pairs while maintaining the same asymptotic depth as a single comparison, reflecting the disjointness-driven parallelism exposed at the Qutes level.

Before concluding this section, we note that the parallel comparison techniques introduced above naturally support higher-level predicates over quantum arrays. A simple example is the predicate `is-sorted`, which checks whether an array is already ordered by evaluating, in parallel, all adjacent comparisons `(v[i] < v[i+1])` and coherently aggregating their outcomes into a single result qubit. If no inversion is detected, the array is certified as sorted. From a divinatory perspective, this spell reveals global order in a single glance, demonstrating how collective comparison spells can express non-trivial global properties with minimal semantic effort.

## 4.3 The Transposition Spell: Parallel and Controlled Swaps

Reordering the elements of an array is a fundamental operation in both classical and quantum computation. In the muggle world, such rearrangements are typically expressed as sequences of elementary swaps, whose execution order must be carefully orchestrated

to avoid conflicts [20]. At the circuit level, quantum transpositions are realized through networks of controlled swaps, often resulting in non-trivial depth even for conceptually simple permutations [23].

In Qutes, reordering is captured by the *transposition spell*, a high-level construct that allows multiple swap operations to be specified simultaneously. When the swaps act on disjoint pairs of array positions, the spell exposes their inherent parallelism, enabling all transpositions to be executed within a single circuit layer [23]. From a magical perspective, this corresponds to a synchronized reshuffling of enchanted compartments, performed in unison rather than through a sequence of manual exchanges.

▶ **Example 9** (Concordia Motus)**.** Suppose that we want to rearrange a quantum array by swapping multiple pairs of adjacent elements simultaneously. In this simplest instance of the transposition spell, swaps are specified over disjoint array positions, ensuring that no two operations interfere with each other:

```
for i in [0..n-2:2] parallel
    swap(v[i],v[i+1]);
```

Each invocation of `swap(v[i],v[i+1])` acts on a distinct pair of elements, allowing all swaps to be executed safely in parallel within a single circuit layer [23]. Unlike comparison-based spells, which leave array positions unchanged and typically require multiple rounds of parallel evaluation, transposition spells directly modify the array structure, performing a collective rearrangement through a single, well-defined incantation.

▶ **Example 10** (Reflexio Totalis)**.** Suppose that we want to reverse the order of the elements in a quantum array. Using the transposition spell, this transformation can be expressed by specifying a family of parallel swaps between symmetric positions:

```
for i in [0..n/2] parallel
    swap(v[i],v[n-i-1]);
```

Each swap exchanges a pair of symmetrically placed elements. Since all pairs are disjoint, the swaps can be executed in parallel, allowing the entire reversal to be realised through a small number of coordinated transposition layers [23, 24]. From a magical perspective, the spell inverts the structure of the enchanted object in a single, orchestrated gesture, highlighting the expressive power of parallel swaps in Qutes.

In the case of swaps, however, the situation is fundamentally different from that of purely observational operations such as comparisons or divination spells. While predicates inspect array elements without modifying their positions, a swap actively changes the structure of the array. Once an element is exchanged, its position is no longer guaranteed to remain unchanged for subsequent operations. As a consequence, sequences of swap instructions cannot, in general, be arbitrarily partitioned into independent parallel groups. To preserve semantic correctness, Qutes must therefore perform a feasibility analysis at compile time, ensuring that the specified transpositions can indeed be scheduled in parallel without introducing conflicts or unintended data dependencies [23].

The transposition spell also admits controlled variants, where the execution of swaps is conditioned on the value of a classical or quantum control. In this case, the same array may be coherently rearranged along different permutation paths, depending on the state of the control register. Such controlled transpositions are a key ingredient in more elaborate transformations, including conditional rotations, quantum shuffling, and data-dependent reordering [23, 24, 13].

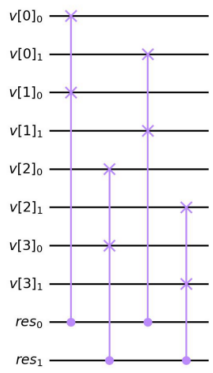


**Figure 6** Parallel circuit implementing controlled swaps on a quantum array. Each control qubit enables a conditional transposition of a disjoint element pair, allowing simultaneous swaps. The figure shows the case of two swaps on elements encoded with two qubits each, but the construction generalises to $k$ parallel controlled swaps on $d$-qubit elements with linear size and constant depth.

While the circuit-level realization of these operations may involve a substantial number of controlled gates and ancillary qubits, this complexity remains hidden at the language level. The programmer specifies *which* elements should be exchanged and *under which conditions*, leaving the compilation process to synthesize the corresponding swap network. As in the previous spells, Qutes separates the semantic simplicity of the operation from the intricate machinery required to enact it [23].

▶ **Example 11** (Ordinis Reparatio)**.** Suppose that we want to reorder a quantum array by conditionally swapping adjacent elements whenever they are found to be out of order. In Qutes, this pattern can be expressed by combining a parallel comparison phase with a coordinated set of controlled swaps:

```
for i in [0..n-2:2] parallel
    if(v[i] > v[i+1]) {
        swap(v[i],v[i+1]); }
```

Each comparison predicate `v[i] > v[i+1]` acts on a disjoint pair of array elements and can therefore be evaluated fully in parallel, following the divination pattern discussed earlier. The resulting comparison outcomes serve as control signals for the corresponding swaps, yielding a parallel layer of controlled transpositions [23]. From a magical perspective, the array is first inspected to reveal local order, and is then rearranged through a coordinated spell that exploits this information, combining global inspection with coherent reordering.

Figure 6 illustrates the circuit-level implementation of a parallel, controlled transposition spell. The data register $|v\rangle$ encodes an array of quantum elements (here, each element consists of two qubits), and the control register $|res\rangle$ provides one control qubit per swap. Each control qubit $|res_t\rangle$ enables a conditional swap between a specific pair of array elements, e.g., $(v[0], v[1])$ and $(v[2], v[3])$, so that multiple pairwise transpositions can be executed simultaneously whenever they act on disjoint positions [23].

At the circuit level, each conditional swap between two $d$-qubit elements is realised as $d$ parallel controlled-SWAP (Fredkin) gates, one per qubit pair in the two elements [23]. Since the swapped element pairs are disjoint, all controlled transpositions can be placed in the

same circuit layer without conflicts: the procedure is therefore inherently parallel across the $k$ swaps. The figure shows the case $d = 2$ and $k = 2$, but the construction generalises directly to arbitrary $d$ and any set of $k$ disjoint swap pairs.

From a complexity standpoint, a single controlled swap on $d$-qubit elements has size $O(d)$ and depth $O(1)$ (up to the constant-depth realisation of a controlled-SWAP gate in the chosen gate set) [23]. Executing $k$ such swaps in parallel over disjoint pairs yields total size $O(kd)$ while preserving depth $O(1)$, highlighting the key advantage of the transposition spell: coordinated reordering with constant-depth overhead when the swaps do not overlap.

Taken together, parallel and controlled transpositions provide the structural backbone for many of the array manipulations discussed in this work, allowing complex permutations to be expressed compactly and reasoned about at a high level, without descending into the details of individual qubit movements.

## 5 A Surprisingly Elegant Bubble Sort

Although sorting is one of the oldest problems studied in computer science, it remains a fundamental task, underpinning a vast range of algorithms and computational workflows [12]. Bubble Sort [20] is often regarded, in the muggle world of classical algorithms, as a pedagogical curiosity rather than a practical sorting technique. Its simplicity comes at the price of poor asymptotic performance, and more efficient methods are typically preferred whenever large datasets are involved. Nevertheless, Bubble Sort possesses a structural regularity that makes it an unexpectedly appealing candidate for illustrating high-level array manipulation in a quantum programming language.

In Qutes, the interest of Bubble Sort does not lie in its classical time complexity, but in the way its logic can be expressed through a small number of composable spells. At each iteration, the algorithm alternates between global comparison phases and parallel conditional transpositions, closely matching the divination and transposition spells introduced in the previous sections [23, 24]. What appears as a naive algorithm at the classical level thus becomes a remarkably clean orchestration of high-level quantum operations.

From a magical perspective, Bubble Sort can be seen as a repeated ritual in which local disorder is gradually revealed and corrected. Divination spells expose adjacent inversions, while transposition spells act in parallel to restore order, pushing larger elements toward their destined positions. Crucially, each step operates on the array as a whole, without requiring the programmer to track individual qubit movements or intermediate circuit states [13].

While the underlying circuit realization inevitably involves multiple layers of controlled operations and incurs non-trivial depth, this complexity remains entirely hidden at the language level. The resulting program offers a compact and declarative description of sorting, emphasizing clarity and structure over low-level efficiency. In this sense, Bubble Sort serves not as an endorsement of inefficiency, but as a compelling demonstration of how Qutes transforms even the simplest classical algorithms into expressive and conceptually elegant quantum spells.

```
void Qute-Bubble-Sort(quint[] v, int n){
for k in [1..n-1]
    for i in [(k%2)..n-2:2] parallel
        if(v[i]>v[i+1]) {
            swap(v[i],v[i+1]); }
}
```

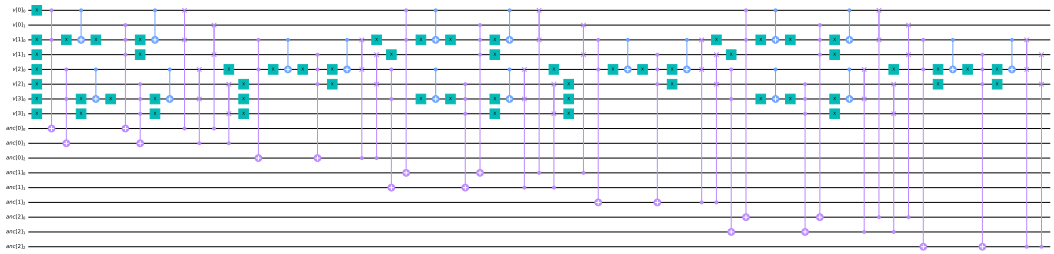

**Figure 7** Quantum circuit obtained from the Qutes implementation of the odd–even Bubble Sort algorithm. The circuit alternates parallel comparison phases and controlled transposition layers acting on disjoint adjacent pairs of array elements. While the figure shows a concrete instance with $n = 4$ elements encoded on $d = 2$ qubits each, the construction generalises to larger arrays and wider registers, preserving parallelism within each sorting phase.

The procedure `Qute-Bubble-Sort(v,n)` depicted above implements a well-known *odd–even* variant of Bubble Sort [20], expressed in a form that matches the transposition spell introduced earlier. The outer loop, indexed by $k$, performs $n - 1$ passes over the array. Each pass is meant to reduce local disorder by eliminating adjacent inversions, gradually pushing larger elements toward the right end of the array (and, symmetrically, smaller elements toward the left), until the whole sequence becomes sorted.

The key idea lies in the inner loop, which alternates between two complementary sets of adjacent pairs. The starting index is ($k$ mod 2): when $k$ is odd the loop starts from $i = 1$, and when $k$ is even it starts from $i = 0$. This alternation partitions the array into disjoint adjacent pairs. Concretely, in an *even* pass the algorithm considers the pairs

$(0, 1),\ (2, 3),\ (4, 5),\ \ldots$ whereas in an *odd* pass it considers $(1, 2),\ (3, 4),\ (5, 6),\ \ldots$

In both cases, the step size 2 ensures that no element belongs to more than one pair within the same pass. This is exactly what enables the annotation `parallel`: all comparisons `v[i] > v[i+1]` refer to disjoint pairs and can therefore be evaluated concurrently, and the resulting conditional swaps can likewise be executed in a single transposition layer [23].

Within each selected pair, the `if` statement performs a local correction: if the left element exceeds the right one, a swap is applied, removing that inversion. Each pass thus acts as a localized "sorting wave" that fixes inversions on half of the edges of the array; the following pass targets the complementary edges. By repeatedly alternating these two phases for $n - 1$ rounds, every element has the opportunity to move step by step toward its final position. In other words, the algorithm can be understood as a simple ritual of successive divination and transposition: adjacent disorder is detected and immediately corrected, but always in a way that preserves full parallelism within each round.

The circuit obtained by compiling the Qutes procedure `Qute-Bubble-Sort(v,n)` for a concrete instance with $n = 4$ array elements, each encoded on $d = 2$ qubits, is shown in Figure 7. The circuit exhibits the characteristic *odd–even* structure of the algorithm: each outer iteration $k$ selects a family of disjoint adjacent pairs, computes in parallel the predicates $v[i] > v[i + 1]$ (divination), and then applies the corresponding controlled swaps (transposition) to those same disjoint pairs. At the circuit level, this is realised by (i) instantiating one comparison subcircuit per active pair to coherently write the outcome into a control qubit (or a small ancillary workspace), and (ii) using these outcomes as controls for a layer of conditional transpositions acting on the data qubits of the array [23, 24]. Since the pairs within a phase are disjoint, all comparisons (and swaps) within that phase can be scheduled in parallel, which is precisely the semantic meaning of the `parallel` annotation in Qutes. Finally, any ancillary information created by the comparison stage is uncomputed to restore workspace qubits and prevent unwanted entanglement from accumulating [23].

The above construction generalises directly to arrays of $n$ elements, each represented by $d$ qubits. In a single odd–even phase, the algorithm processes $p = \Theta(n)$ disjoint adjacent pairs (specifically, $p = \lfloor n/2 \rfloor$). Each pair requires one $d$-bit greater-than predicate; with the standard lexicographic cascade, a single `>` instance can be implemented with depth $O(d)$ and size $O(d)$, and (when workspace must be cleaned) an additional uncomputation stage contributes only a constant-factor overhead [23]. Because all $p$ comparisons act on disjoint qubits, the phase depth remains $O(d)$, while the phase size scales linearly as $O(p\,d) = O(nd)$. The subsequent controlled transposition layer consists of $p$ controlled swaps on $d$-qubit elements, which can be implemented with size $O(d)$ per pair and constant depth (up to the chosen decomposition of a controlled-SWAP gate), hence contributing $O(nd)$ size and only $O(1)$ additional depth [23, 13]. Since the outer loop performs $n - 1 = \Theta(n)$ phases, the overall circuit depth for Qute-Bubble-Sort is $D(n, d) = \Theta(n \cdot d)$, where the $d$ factor reflects the intrinsic bit-significance dependency of comparisons. The overall circuit size is $S(n, d) = \Theta(n \cdot nd) = \Theta(n^2 d)$, as each of the $\Theta(n)$ phases contains $\Theta(n)$ pairwise operations, each costing $O(d)$ gates. Thus, while Bubble Sort remains asymptotically quadratic in circuit size (as expected), Qutes exposes a structured parallelism that keeps each phase shallow (polylogarithmic in the element domain size $2^d$) and expresses the entire construction as a compact composition of divination and transposition spells.

We conclude this section by clarifying its relationship with known lower bounds for sorting. In the circuit model, depth provides a natural notion of execution time on a quantum machine, as it captures the number of sequential layers of operations that cannot be parallelised [23], while circuit size measures the total amount of work performed.

The quadratic size reflects the fact that the algorithm performs $\Theta(n^2)$ pairwise comparisons overall, exceeding the lower bound of $\Omega(n \log n)$ comparisons required to sort an arbitrary input [20]. This is consistent with classical and quantum decision-tree lower bounds, which constrain the number of comparisons but do not restrict their degree of parallelisation [23].

If $d = \Theta(\log n)$, corresponding to elements drawn from a polynomial-size domain, the circuit depth becomes $\Theta(n \log n)$, matching the classical lower bound up to constant factors. When the element size is smaller, and in particular when $d = o(\log n)$ or even $d = O(1)$, individual comparisons can be implemented with sublogarithmic or constant depth, and the overall circuit depth reduces to $\Theta(n)$. While the total circuit size remains quadratic, the depth (and the execution time) becomes asymptotically smaller than the classical $\Omega(n \log n)$.

This separation between size and depth is not an artefact of Bubble Sort itself, but a consequence of the circuit model and of how parallelism is exposed. Unlike classical analyses, which collapse all operations into a single sequential cost, the circuit model distinguishes total work from independent execution layers. As a result, algorithms that are suboptimal in size may still be attractive when circuit depth is taken as the primary measure of running time. In this sense, the construction exhibits a genuine quantum advantage in parallel time when $d = o(\log n)$, arising from the ability to evaluate many independent comparisons simultaneously rather than from a reduction in total informational complexity.

## 6 Discussion and Conclusions: Why Wizards Need Better Spells

This work explored how high-level, array-oriented abstractions can reshape how quantum algorithms are expressed and reasoned about. Rather than pursuing asymptotic performance improvements, the primary goal of Qutes is to close the gap between algorithmic intuition and executable quantum programs. By elevating common patterns, such as parallel comparison, controlled transposition, and indexed access, to first-class language constructs, Qutes allows programmers to describe quantum algorithms at a level that reflects their conceptual structure.

The quantum Bubble Sort case study illustrates this philosophy clearly. Although the underlying circuit respects known classical and quantum lower bounds, the corresponding Qutes program exposes a clean and highly parallel structure built from a small set of composable spells. The resulting circuit depth reflects the true sequential dependencies of the problem, while the complexity of gate orchestration, ancilla management, and uncomputation is delegated to the compiler. In this sense, Qutes does not obscure computational costs, but reorganises them in a way that is both explicit and meaningful at the algorithmic level.

This separation between semantic simplicity and circuit-level complexity highlights a fundamental trade-off in quantum programming languages. Low-level frameworks offer maximal control and optimisation opportunities, but force programmers to reason in terms of qubits, gates, and wiring. High-level abstractions, by contrast, may sacrifice some fine-grained control, but they are essential for scalability, readability, and correctness when algorithms grow beyond toy examples. Our results suggest that expressive, spell-like constructs are not merely syntactic sugar, but a necessary ingredient for managing the inherent parallelism and global behaviour of quantum computation.

Looking forward, the abstractions presented here open several directions for future work. Extending the repertoire of array spells, exploring optimisation strategies that preserve high-level semantics, and studying the interaction between such abstractions and fault-tolerant compilation are all natural next steps. More broadly, as quantum hardware continues to evolve, the success of quantum software will increasingly depend on languages that allow programmers to think and reason like wizards, not like circuit engineers. Wizard machines deserve better spells, and Qutes is a step in that direction.

## References


1 Charles H Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Physical Review Letters*, 70(13):1895, 1993.

2 Benjamin Bichsel, Daniel Zhan, Goran Sutter, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020. doi:10.1145/3385412.3386007.

3 Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):339–354, 1998.

4 Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

5 David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

6 Cirq Developers. Cirq. https://quantumai.google/cirq, 2024. doi:10.5281/zenodo.15191735.

7 Olivia Di Matteo, Santiago Nunez-Corrales, Michal Stechly, Steven P. Reinhardt, and Tim Mattson. An abstraction hierarchy toward productive quantum programming. In *IEEE Quantum Week 2024 (QCE 2024)*, pages 979–989, 2024.

8 Thomas G. Draper, Samuel A. Kutin, Eric M. Rains, and Krysta M. Svore. A logarithmic-depth quantum carry-lookahead adder, 2004. arXiv:quant-ph/0406142.

9 Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014. URL: https://arxiv.org/abs/1411.4028.

10 Simone Faro, Francesco Pio Marino, and Gabriele Messina. Extending qutes: a practical high-level language for quantum computing. *The Computer Journal*, page bxaf133, November 2025. doi:10.1093/comjnl/bxaf133.

11 Simone Faro, Francesco Pio Marino, and Gabriele Messina. Qutes: A high-level quantum programming language for simplified quantum computing. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2025, University of Notre Dame, Notre Dame, IN, USA*, pages 47:1–47:9. ACM, 2025. doi:10.1145/3731545.3744153.

12 Simone Faro, Francesco Pio Marino, and Stefano Scafiti. Fast-insertion-sort: a new family of efficient variants of the insertion-sort algorithm. In *SOFSEM 2020 Doctoral Student Research Forum*, volume 2568 of *CEUR Workshop Proceedings*, pages 37–48, 2020. URL: https://ceur-ws.org/Vol-2568/paper4.pdf.

13 Simone Faro, Arianna Pavone, and Caterina Viola. Families of constant-depth quantum circuits for rotations and permutations. In *Proceedings of the 25nd Italian Conference on Theoretical Computer Science*, 2024.

14 Hermann Fürntratt, Paul Schnabl, Florian Krebs, Roland Unterberger, and Herwig Zeiner. Towards higher abstraction levels in quantum computing. In *Service-Oriented Computing - ICSOC 2023 Workshops*, Lecture Notes in Computer Science, 2023. doi:10.1007/978-981-97-0989-2_13.

15 Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical Review Letters*, 100(16):160501, 2008. doi:10.1103/PhysRevLett.100.160501.

16 Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

17 Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, pages 212–219, 1996. doi:10.1145/237814.237866.

18 Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024. doi:10.48550/arXiv.2405.08810.

19 Iordanis Kerenidis and Anupam Prakash. Quantum recommendation systems. *Nature Communications*, 8(1):1–8, 2017. doi:10.1038/s41467-017-02092-4.

20 Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

21 Ryan LaRose. Overview and comparison of gate level quantum software platforms. *Quantum*, 3:130, 2019. doi:10.22331/Q-2019-03-25-130.

22 Francesco Pio Marino, Simone Faro, and Antonio Scardace. Practical implementation of a quantum string matching algorithm. In *Proceedings of the 2024 Workshop on Quantum Search and Information Retrieval, QUASAR*, pages 17–24. ACM, 2024. doi:10.1145/3660318.3660327.

23 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 10th anniversary edition edition, 2010. doi:10.1017/CBO9780511976667.

24 Arianna Pavone and Caterina Viola. The quantum cyclic rotation gate. *SN Comput. Sci.*, 5(7):830, 2024. doi:10.1007/S42979-024-03141-4.

25 Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2), 1999. doi:10.1137/S0036144598347011.

26 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, 2018. doi:10.1145/3183895.3183901.

27 Matan Vax, Peleg Emanuel, Eyal Cornfeld, Israel Reichental, Ori Opher, Ori Roth, Tal Michaeli, Lior Preminger, Lior Gazit, Amir Naveh, and Yehuda Naveh. Qmod: Expressive high-level quantum modeling, 2025. doi:10.48550/arXiv.2502.19368.