Overview

Contents

1. Integer factoring and computing GCDs

Integer factorization

Integer factorization

Input: an integer $N \geq 2$

Output: the prime factorization of N

The *prime factorization* of N is the list of prime factors of N and the powers to which they must be raised to obtain N by multiplication.

Prime factorizations are unique (by the Fundamental Theorem of Arithmetic).

Example

The prime factorization of 12 is

$$12 = 2^2 \cdot 3$$

Integer factorization

Integer factorization

Input: an integer $N \geq 2$

Output: the prime factorization of N

The *prime factorization* of N is the list of prime factors of N and the powers to which they must be raised to obtain N by multiplication.

Prime factorizations are unique (by the Fundamental Theorem of Arithmetic).

Example

The prime factorization of

$$340282366920938463463374074317682310984309834$$

is

$$\begin{aligned} &340282366920938463463374074317682310984309834 \\ &\quad = 3^2 \cdot 74519450661011221 \cdot 5073729280707932631243580787 \end{aligned}$$

Integer factorization

Integer factorization

Input: an integer $N \geq 2$

Output: the prime factorization of N

The *prime factorization* of N is the list of prime factors of N and the powers to which they must be raised to obtain N by multiplication.

Prime factorizations are unique (by the Fundamental Theorem of Arithmetic).

Example

The prime factorization of this number is unknown:

RSA1024

$$= \begin{aligned}&135066410865995223349603216278805969938881475605667027524485143\\&851526510604859533833940287150571909441798207282164471551373680\\&419703964191743046496589274256239341020864383202110372958725762\\&358509643110564073501508187510676594629205563685529475213500852\\&879416377328533906109750544334999811150056977236890927563\end{aligned}$$

Integer factorization

Integer factorization

Input: an integer $N \geq 2$

Output: the prime factorization of N

The *prime factorization* of N is the list of prime factors of N and the powers to which they must be raised to obtain N by multiplication.

Prime factorizations are unique (by the Fundamental Theorem of Arithmetic).

Example

The largest RSA challenge number factored thus far is RSA250, which was factored in 2020 using the *number field sieve.*

$$\begin{matrix}2140324650240744961264423072839333563008614715144755017797754920881418023447140136643345519095804679610992851872470914587687396261921557363047454770520805119056493106687691590019759405693457452230589325976697471681738069364894699871578494975937497937\end{matrix} = \begin{matrix}641352894770715802787901901705773890848250147429434472081168596320245323446302386235987526683477087376619255856946397988533367\end{matrix} \cdot \begin{matrix}333720275949781565562260106053551142279407603447675546667845209870238417292100370802574486732968818775657189862580369320627 11\end{matrix}$$

Greatest common divisor

Greatest common divisor (GCD)

Input: nonnegative integers N and M (not both zero)

Output: the greatest common divisor of N and M

The greatest common divisor of N and M is the largest integer d that evenly divides both N and M.

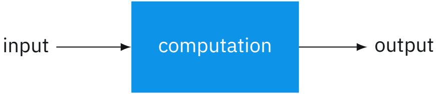
This is possible because we have ***efficient algorithms*** for computing GCDs, including Euclid's algorithm.

Could there be an efficient (classical) algorithm for integer factorization?

Yes — but we haven't found one yet.

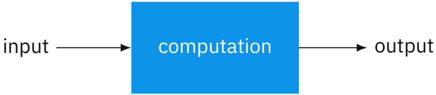
2. Measuring computational cost

An abstract view of computation



- Inputs and outputs are binary strings.
- The computation could be modeled in a variety of ways, including (but not limited) these:
  - Turing machines
  - *Boolean circuits*
  - *quantum circuits*
  - Python programs

Encodings and input length



- Inputs and outputs are binary strings.
- Through binary strings we can encode interesting objects:
  - numbers
  - vectors
  - matrices
  - graphs
  - descriptions of molecules
  - lists of these and other objects

Encodings and input length

Example

We can encode nonnegative integers using *binary notation:*

number	**encoding**	**length**
0	0	1
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4
10	1010	4
11	1011	4
12	1100	4
$\vdots$	$\vdots$	$\vdots$

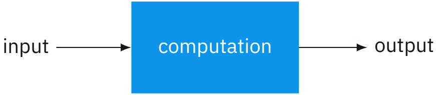
Length of the binary encoding of N:

$$\lg(N) = \begin{cases} 1 & N = 0 \\ 1 + \lfloor \log_2(N) \rfloor & N \geq 1 \end{cases}$$

A sign bit can be added to represent arbitrary integers.

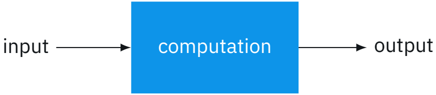
Leading zeros may be allowed to fill out a sufficiently large word length.

Encodings and input length



- Many objects of interest can be encoded as binary strings.
- Standard or universally agreed upon encoding schemes don't always exist — we just pick (or invent) them as needed.
- We generally don't concern ourselves too much with the specifics — converting back and forth between "reasonable" encoding schemes typically has negligible cost.
- In general, the *input length* is the length of the binary string encoding of the input, with respect to whatever encoding scheme has been selected.

Elementary operations



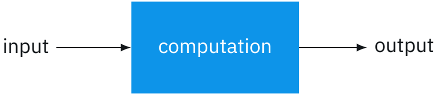
For circuit-based models of computation, it is typical that we view each *gate* as being an elementary operation.

A standard quantum gate set

- Single-qubit unitary gates from this list: X, Y, Z, H, S, $S^{\dagger}$, T, $T^{\dagger}$
- Controlled-NOT gates
- Single-qubit standard basis measurements

The unitary gates in this set are *universal* — any unitary operation can be closely approximated by a circuit of these gates.

Elementary operations



For circuit-based models of computation, it is typical that we view each *gate* as being an elementary operation.

A standard Boolean gate set

- AND
- OR
- NOT
- FANOUT

FANOUT gates are not always explicitly considered to be gates, but for this lesson it is important to do this.

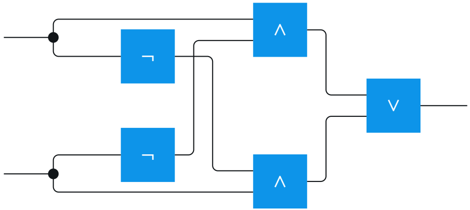
Circuit size (and depth)

Circuit size

The *size* of a circuit (Boolean or quantum) is the total number of gates it includes. We may write size(C) to refer to the size of a circuit C.

Example

This Boolean circuit has size 7:



Circuit size (and depth)

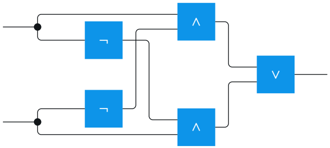
Circuit size

The *size* of a circuit (Boolean or quantum) is the total number of gates it includes. We may write size(C) to refer to the size of a circuit C.

Circuit size corresponds to *sequential running time.* (This is how we will measure computational cost in this lesson.)

Circuit depth

The *depth* of a circuit is the maximum number of gates encountered on any path from an input to an output wire.



Cost as a function of input length

When we analyze algorithms, we're generally interested in how their cost scales as inputs grow in size.

Each circuit has a fixed size — so we need a *family* $\{C_1, C_2, \ldots\}$ of circuits to describe an algorithm, typically one circuit for each input length.

Example

A classical algorithm for integer factorization could be described by a family of Boolean circuits, where C_n factors n-bit numbers.

The cost of such an algorithm is described by a *function:*

$$t(n) = \text{size}(C_n)$$

Example: integer addition

Integer addition

Input: integers N and M

Output: $N + M$

Animation involving half- and full-adders

Total number of gates required: $21(n-1)+10 = 21n-11$

Asymptotic notation

It's good to know precisely how many gates are needed to perform computations...

...but we'll be buried in secondary details if we try to do this in general.

Big-O notation

For two functions $g(n)$ and $h(n)$, we write that $g(n) = O(h(n))$ if there exists a positive real number $c > 0$ and a positive integer n_0 such that

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

for all $n \geq n_0$.

Example

$$17n^3 - 257n^2 + 65537 = O(n^3)$$

Asymptotic notation

Big-O notation

For two functions $g(n)$ and $h(n)$, we write that $g(n) = O(h(n))$ if there exists a positive real number $c > 0$ and a positive integer n_0 such that

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

for all $n \geq n_0$.

Example

There exists a family $\{C_1, C_2, \ldots,\}$ of Boolean circuits, where C_n adds two n-bit nonnegative integers together, such that

$$\text{size}(C_n) = O(n)$$

Addition of n-bit integers can be computed at cost $O(n)$.

Asymptotic notation

Examples

Addition of n-bit integers can be computed at cost $O(n)$.

Multiplication of n-bit integers can be computed at cost $O(n^2)$.

Integer multiplication

Input:	integers N and M
Output:	NM

By the *standard multiplication* algorithm, there are Boolean circuits of size $O(n^2)$ for multiplying n-bit integers.

More generally, there are circuits of size $O(nm)$ for multiplying an n-bit integer to an m-bit integer.

By the *Schönhage-Strassen* multiplication algorithm, multiplication of two n-bit integers can be computed at cost $O(n \lg(n) \lg(\lg(n)))$.

Asymptotic notation

Examples

Addition of n-bit integers can be computed at cost $O(n)$.

Multiplication of n-bit integers can be computed at cost $O(n^2)$.

Division of n-bit integers can be computed at cost $O(n^2)$.

Integer division

Input:	integers N and $M \neq 0$
Output:	integers q and r so that $0 \leq r < \lvert M \rvert$ and $N = qM + r$

The *standard division* algorithm solves this problem for n-bit integers at cost $O(n^2)$.

Asymptotic notation

Examples

Addition of n-bit integers can be computed at cost $O(n)$.

Multiplication of n-bit integers can be computed at cost $O(n^2)$.

Division of n-bit integers can be computed at cost $O(n^2)$.

GCDs of n-bit integers can be computed at cost $O(n^2)$.

Greatest common divisor (GCD)

Input:	nonnegative integers N and M (not both zero)
Output:	the greatest common divisor of N and M

Asymptotic notation

Examples

Addition of n-bit integers can be computed at cost $O(n)$.

Multiplication of n-bit integers can be computed at cost $O(n^2)$.

Division of n-bit integers can be computed at cost $O(n^2)$.

GCDs of n-bit integers can be computed at cost $O(n^2)$.

Modular exponentiation for n-bit integers can be computed at cost $O(n^3)$.

Modular exponentiation

Input: integers $K \geq 0$, $M \geq 1$, and N

Output: $N^K \pmod{M}$

Asymptotic notation

Examples

Addition of n-bit integers can be computed at cost $O(n)$.

Multiplication of n-bit integers can be computed at cost $O(n^2)$.

Division of n-bit integers can be computed at cost $O(n^2)$.

GCDs of n-bit integers can be computed at cost $O(n^2)$.

Modular exponentiation for n-bit integers can be computed at cost $O(n^3)$.

Integer factorization

Input: an integer $N \geq 2$

Output: the prime factorization of N

A simple *trial-division* algorithm has cost $O\left(n^2 2^{n/2}\right)$ to factor n-bit integers.

The *number field sieve* is conjectured to have cost $2^{O\left(n^{1/3} \lg^{2/3}(n)\right)}$.

Polynomial versus exponential cost

An algorithm's cost is *polynomial* if it is $O(n^b)$ for some fixed constant $b > 0$.

Examples

Integer addition, multiplication, and division; computing GCDs; and modular exponentiation all have polynomial cost.

As a rough, first-order approximation, algorithms having polynomial cost are abstractly viewed as representing *efficient* algorithms.

Acknowledgment

An algorithm whose cost scales as $n^{1{,}000{,}000}$ on inputs of length n is not reasonably categorized as efficient...

...but it must still doing something clever to avoid exponential cost!

In practice, the identification of a polynomial-cost algorithm for a problem is just a first step toward actual efficiency.

Polynomial versus exponential cost

An algorithm's cost is *polynomial* if it is $O(n^b)$ for some fixed constant $b > 0$.

An algorithm's cost scales *sub-exponentially* if it is

$$O\left(2^{n^{\varepsilon}}\right)$$

for every $\varepsilon > 0$. Otherwise it is *exponential* (or super-exponential).

- No sub-exponential cost classical algorithm is known for integer factorization.
- Shor's algorithm is a quantum algorithm with *polynomial cost* for integer factorization.
- NP-complete problems are conjectured not to have sub-exponential cost — this is a circuit-based formulation of the *exponential-time hypothesis.*

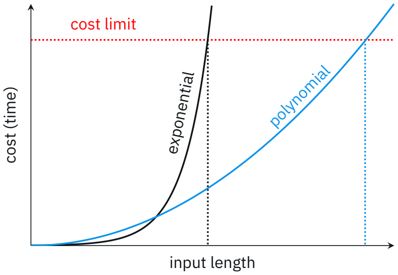
Polynomial versus exponential cost

An algorithm's cost is *polynomial* if it is $O(n^b)$ for some fixed constant $b > 0$.

An algorithm's cost scales *sub-exponentially* if it is

$$O\left(2^{n^{\varepsilon}}\right)$$

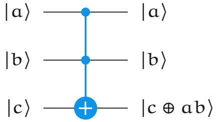
for every $\varepsilon > 0$. Otherwise it is *exponential* (or super-exponential).



3. Classical computations on quantum computers

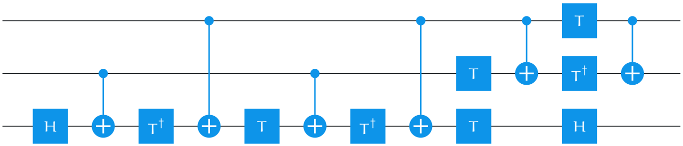
Toffoli gates

Recall that Toffoli gates are controlled-controlled-NOT gates:



We can also think about Toffoli gates as being query gates for the AND function.

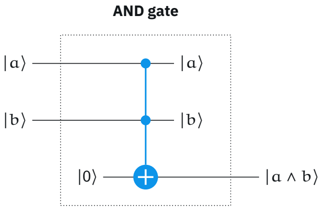
Toffoli gates can be implemented by elementary operations like this:

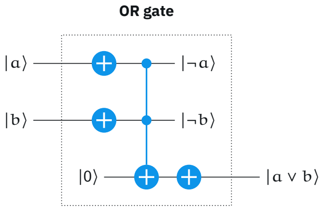


Simulating Boolean gates

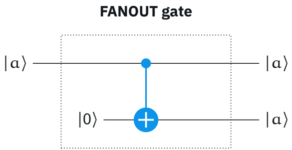
NOT gates can be left alone.

AND and OR gates can be simulated with Toffoli and NOT gates:

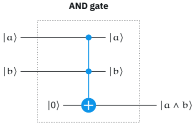


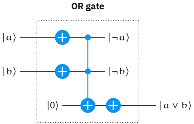


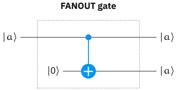
FANOUT gates can be simulated with controlled-NOT gates:



Simulating Boolean circuits



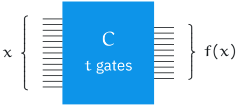


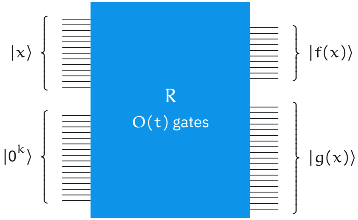


Suppose that we have a Boolean circuit C of size t that computes a function

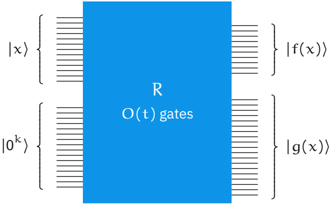
$$f : \Sigma^n \to \Sigma^m$$

Replace each AND, OR, and FANOUT gate of C with its quantum simulation:





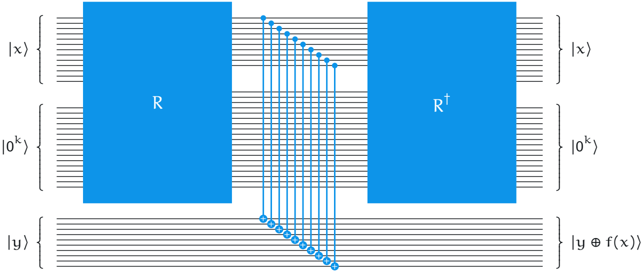
Simulating Boolean circuits



The string $g(x)$ represents *garbage.*

It will ruin the interference patterns that make quantum algorithms work.

To get rid of it, we can use the fact that R can be inverted...



Simulating Boolean circuits