Introduction to Quantum Computing

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 500

1. Quantum Versions of Classical Computations
   - From Reversible Classical to Quantum Computations
   - Reversible Implementations of Classical Circuits
   - A Language for Quantum Implementations
   - Some Example Programs for Arithmetic Operations

Subsection 1

From Reversible Classical to Quantum Computations

Reversibility

- Any sequence of quantum transforms effects a unitary transformation U on the quantum system.
- As long as no measurements are made, the initial quantum state of the system prior to a computation can be recovered from the final quantum state $|\psi\rangle$ by running $U^{-1} = U^\dagger$ on $|\psi\rangle$.
- Thus, any quantum computation is reversible prior to measurement in the sense that the input can always be computed from the output.

Irreversibility in the Classical Case

- In contrast, classical computations are not in general reversible.
- It is not usually possible to compute the input from the output.
- E.g., while the classical NOT operation is reversible, the AND, OR and NAND are not.
- Every classical computation has a classical reversible analog that takes only slightly more computational resources.

Seeking Reversibility in the Classical Case

- We will see how to make basic Boolean gates reversible.
- Further we will see how to make entire Boolean circuits reversible in a resources efficient way, considering:
  - Space;
  - The number of bits required;
  - The number of primitive gates.
- This construction of efficient classical reversible versions of arbitrary Boolean circuits easily generalizes to a construction of quantum circuits that efficiently implement general classical circuits.

Classical Reversible Computations

- Any classical reversible computation with n input and n output bits simply permutes the $N = 2^n$ bit strings.
- Thus, for any such classical reversible computation, there is a permutation $\pi : \mathbb{Z}_N \to \mathbb{Z}_N$ sending an input bit string to its output bit string.
- This permutation can be used to define a quantum transformation

$$U_\pi : \sum_{x=0}^{N-1} a_x |x\rangle \mapsto \sum_{x=0}^{N-1} a_x |\pi(x)\rangle,$$

  that behaves on the standard basis vectors, viewed as classical bit strings, exactly as π did.
- The transformation U_π is unitary, since it simply reorders the standard basis elements.

Making Classical Computations Reversible

- Any classical computation on n input and m output bits defines function
$$\begin{array}{rccc} f: & \mathbb{Z}_N & \to & \mathbb{Z}_M; \\ & x & \mapsto & f(x). \end{array}$$
- f mapps the $N = 2^n$ input bit strings to the $M = 2^m$ output bit strings.
- Such a function can be extended in a canonical way to a reversible function π_f acting on $n + m$ bits partitioned into two registers:
  - The n-bit input register;
  - The m-bit output register.

Making Classical Computations Reversible (Cont'd)

- The function π_f is specified by

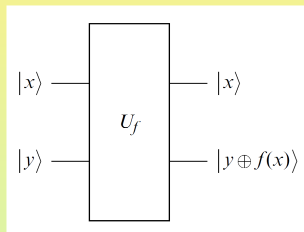
$$\begin{array}{rccl} \pi_f: & \mathbb{Z}_L & \to & \mathbb{Z}_L; \\ & (x,y) & \mapsto & (x, y \oplus f(x)). \end{array}$$

- Here, $\oplus$ denotes the bitwise exclusive-OR.
- The function π_f acts on the $L = 2^{n+m}$ bit strings, each made up of an n-bit string x and an m-bit string y.
- For $y = 0$, the function π acts like f, except that:
  - The output appears in the output register;
  - The input register retains the input.

Issues of Implementation

- Since π_f is reversible, there is a corresponding unitary transformation

$$U_f : |x, y\rangle \mapsto |x, y \oplus f(x)\rangle.$$



- Most unitary operators do not have an efficient implementation.
- U_f has an efficient implementation as long as there is a classical circuit that computes f efficiently.
- The method for constructing an efficient implementation of U_f from an efficient classical circuit for f involves two steps.
  - We construct an efficient reversible classical circuit that computes f;
  - We substitute quantum gates for each of the reversible gates that make up the reversible classical circuit.

NOT

- Let b_1 and b_0 be two binary variables, taking only values 0 or 1.
- The NOT gate is already reversible.
- We will use X to refer to both:
  - The classical reversible gate;
  - The single-qubit operator

$$X = |0\rangle\langle 1| + |1\rangle\langle 0|.$$

  I performs a classical not operation on classical bits encoded as the standard basis elements.

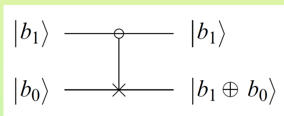
XOR

- The controlled negation performed by the

$$C_{\text{not}} = \bigwedge_1 X$$

gate amounts to an XOR operation on its input values.

- It retains the value of the first bit b_1, and replaces the value of the bit b_0 with the XOR of the two values.



- The quantum version behaves like the reversible version on the standard basis vectors.
- Its behavior on all other states can be deduced from the linearity of the operator.

AND

- It is impossible to perform a reversible AND operation with only two bits.
- The three-bit controlled-controlled-NOT gate, or **Toffoli gate**, $T = \bigwedge_2 X$ can be used to perform a reversible AND operation:

$$T|b_1, b_0, 0\rangle = |b_1, b_0, b_1 \wedge b_0\rangle,$$

  where $\wedge$ is notation for the classical AND of two bit values.
- The Toffoli gate is defined for all inputs.
- When the value of the third bit is 1,

$$T|b_1, b_0, 1\rangle = |b_1, b_0, 1 \oplus b_1 \wedge b_0\rangle.$$

Toffoli Gate and Other Boolean Connectives

- By varying the values of input bits, the Toffoli gate T can be used to construct a complete set of Boolean connectives.
- Thus, Toffoli gates suffice to construct any combinatorial circuit.
- The Toffoli gate computes NOT, AND, XOR and NAND in the following way ($\neg$ is the classical NOT acting on a single bit):

$$\begin{array}{llcl} \text{NOT}: & T|1,1,x\rangle & = & |1,1,\neg x\rangle \\ \text{AND}: & T|x,y,0\rangle & = & |x,y,x \wedge y\rangle \\ \text{XOR}: & T|1,x,y\rangle & = & |1,x,x \oplus y\rangle \\ \text{NAND}: & T|x,y,1\rangle & = & |x,y,\neg(x \wedge y)\rangle. \end{array}$$

.

The Fredkin Gate

- An alternative to the Toffoli gate, the **Fredkin gate** F, acts as a **controlled swap**:

$$F = \bigwedge_1 S,$$

where S is the two-bit swap operation $S : |xy\rangle \to |yx\rangle$.

- The Fredkin gate F, like the Toffoli gate T, can implement a complete set of classical Boolean operators ($\vee$ is the classical OR of two bits):

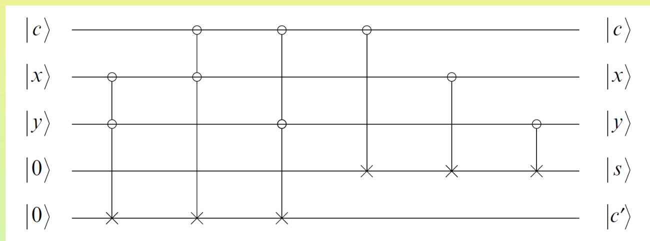
$$\begin{array}{lrcl} \text{NOT}: & F|x,0,1\rangle & = & |x,x,\neg x\rangle \\ \text{OR}: & F|x,y,1\rangle & = & |x, y \vee x, y \vee \neg x\rangle \\ \text{AND}: & F|x,0,y\rangle & = & |x, y \wedge x, y \wedge \neg x\rangle. \end{array}$$

Using Toffoli Gates for Classical Circuits

- We saw that, using just the Toffoli gate T, or the Fredkin gate F, we can implement a complete set of classical Boolean connectives.
- So, by combining T or F gates, we can realize arbitrary Boolean circuits.
- We will describe explicit implementations of certain classical functions.
- The operations C_{not} and X can be implemented by Toffoli gates with the addition of one or two bits permanently set to 1.
- So we use C_{not} and X gates in our construction, but all constructions can be done using only Toffoli gates.

Example

- The circuit shown implements a one-bit full adder using Toffoli and controlled-NOT gates:



  - x and y are the data bits;
  - s is their sum (modulo 2);
  - c is the incoming carry bit;
  - c' is the new carry bit.
- Several one-bit adders can be strung together to achieve full n-bit addition.

Subsection 2

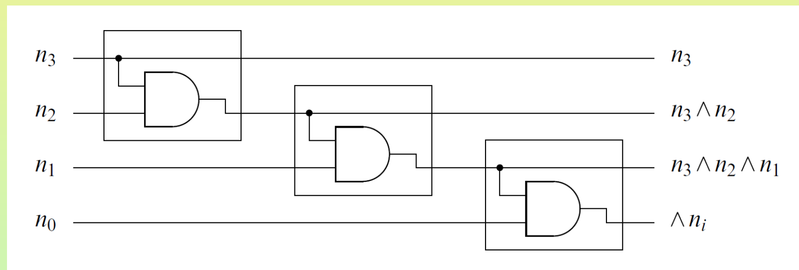
Reversible Implementations of Classical Circuits

A Naive Reversible Implementation

- We consider a classical machine that consists of:
    - A register of bits;
    - A processing unit.
- The processing unit:
    - Performs simple Boolean operations or gates on one or two of the bits in the register at a time;
    - Stores the result in one of the register's bits.
- We assume that, for a given size input, the sequence of operations and their order of execution are fixed and do not depend on the input data or on other external control.
- In analogy with quantum circuits, we draw bits of the register as horizontal lines.

Example

- A simple program (for four-bit conjunction) is depicted below.

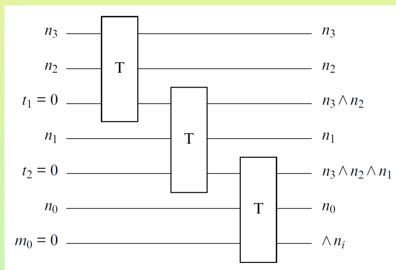


Space Complexity and Irreversibility

- An arbitrary Boolean circuit can be transformed into a sequence of operations on a register large enough register to hold:
  - Input bits;
  - Output bits;
  - Intermediate bits.
- The **space complexity** of a circuit is the size of the register.
- Computations performed by this machine are not reversible in general.
- By reusing bits in the register, the machine erases information that cannot be reconstructed later.

Trivial Solution of the Irreversibility Issue

- A trivial, but highly space inefficient, solution to this problem is not to reuse bits during the entire computation.
- The following figure illustrates how the circuit can be made reversible by assigning the results of each operation to a new bit.



- The operation that reversibly computes the conjunction and leaves the result in a bit initially set to 0 is the Toffoli gate.

Using NOT and AND Gates

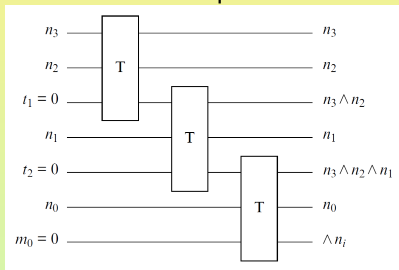
- The NOT gate is reversible.
- Moreover, NOT and AND form a complete set of Boolean operations.
- So the construction can be generalized to turn any computation using Boolean logic operations into one using only reversible gates.
- This implementation needs an additional bit for every AND performed.
- So if the original computation takes t steps, then a reversible one, constructed in this way, requires up to t additional bits of space.

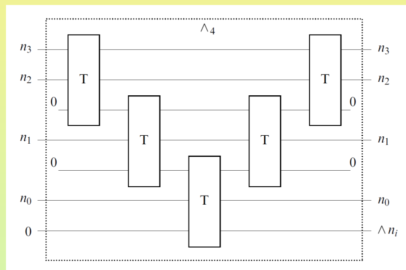
Reusing Bits: Resetting versus Reversing

- After computation, the additional space is no longer in the 0 state.
- So it cannot be directly reused, e.g., to compose two reversible circuits.
- Reusing temporary bits is crucial in keeping space requirements close to that of the original nonreversible classical computation.
- Resetting a bit to zero is not reversible (it loses information).
- So it cannot be used as part of a reversible computation.
- Reversible computations cannot reclaim space through resetting.
- They can uncompute any bit set during a reversible computation by reversing the part of the computation that computed the bit.

Example

- Consider the computation shown.





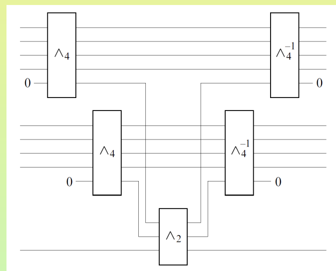
- Bits t_1 and t_0 are temporarily used to obtain the output in bit m_0.
- We may uncompute these bits, resetting them to their original 0 value, by reversing all but the last step of the circuit on the left.
- These may be reused as part of a continuing computation.
- The temporary bits are reclaimed at the cost of roughly doubling the number of steps.

Uncomputing and Reusing Bits

- We can reduce the number of qubits needed by uncomputing them and reusing them in the course of the algorithm.
- The method of uncomputing bits by performing all of the steps in reverse order, except those giving the output, works for any classical Boolean subcircuit.
- Consider a classical Boolean subcircuit consisting of t gates operating on an s-bit register.
- The naive construction requires up to t additional bits in the register.

Example

- Suppose we want to construct the conjunction of eight bits.
- Simply reversing the steps, generalizing the approach for four bits, requires six additional temporary bits and one bit for the final output.
- We can save space by using the four-bit AND circuit involving the T gates four times and then combining the results as shown on the right.
- This construction uses two temporary bits in addition to the two temporary bits used in each of the four-bit ANDs.



- Since each of the four-bit ANDs uncomputes its temporary bits, these bits can be reused by the subsequent four-bit ANDs.
- This circuit uses only a total of four additional temporary bits, but it does require more gates.

Additional Bits: Invertibility and Uncomputing

- There is an art to deciding when to uncompute which bits to:
  - Maintain efficiency;
  - Retain subresults used subsequently in the computation.
- The key ideas, which are the main ingredients of the general construction, are:
  - Adding bits to obtain reversibility;
  - Uncomputing their values so that they may be reused.
- By choosing carefully when and what to uncompute, it is possible to make a positive tradeoff.
  - We sacrifice some additional gates;
  - We obtain a much more efficient use of space.
- We will see, e.g., an explicit efficient implementation of an m-way AND.

General Construction: Goals

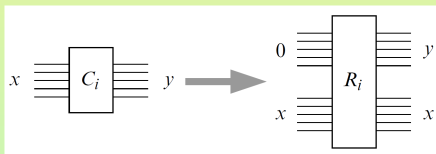
- We look at how, by carefully choosing which bits to uncompute when, a reversible version of any classical computation can be achieved with only minor increases in the number of gates and bits.
- We show that any classical circuit using t gates and s bits, has a reversible counterpart using only:
  - $O(t^{1+\varepsilon})$ gates;
  - $O(s \log t)$ bits.
- For $t \gg s$, this construction:
  - Uses significantly less space than the $(s+t)$ space of the naive approach described previously;
  - Incurs only a small increase in the number of gates.

Analysis of Number of Bits Used

- Let C be a classical circuit, composed of AND and NOT gates.
- Suppose C uses no more than t gates and s bits.
- The circuit C can be partitioned into $r = \lceil \frac{t}{s} \rceil$ subcircuits $C_1, \ldots, C_r$, each containing s or fewer consecutive gates,

$$C = C_1 C_2 \ldots C_r.$$

- Each subcircuit C_i has s input and s output bits, some of which may be unchanged.
- We know each circuit C_i can be replaced by a reversible circuit R_i using at most s additional bits.



- The circuit R_i returns its input as well as the s output values used in the subsequent computation.
- The input values will be used to uncompute and recompute R_i in order to save space.

General Methodology: Three Steps

- In general, R_i can be constructed using at most $3s$ gates.

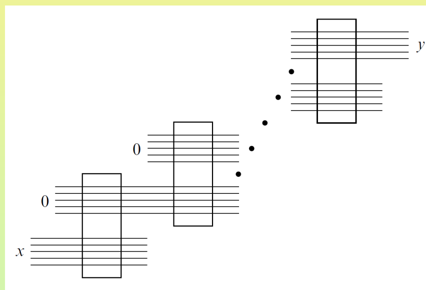
Step 1: Compute all of the output values in a reversible way.
For every AND or NOT gate in the original circuit C_i, the circuit R_i has a Toffoli or NOT gate.
This step uses the same number of gates, s, as C_i, and uses no more than s additional bits.

Step 2: Copy all of the output values, the values used in subsequent parts of the computation, to the output register, a set of no more than s additional bits.

Step 3: Perform the sequence of gates used to carry out step 1, but this time in reverse order.
In this way all bits, except those in the output register, are reset to their original values.
Specifically, all temporary bits are returned to 0, and we have recovered all of the input values.

A Space-Inefficient Configuration

- The circuits $R_1 \ldots R_r$ are combined as in the figure.



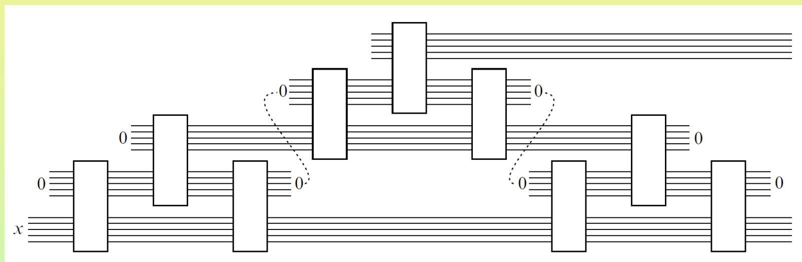
- The configuration performs the computation C in a reversible but space-inefficient way.
- The subcircuits R_i can be combined in a special way that uses space more efficiently by uncomputing and reusing some of the bits.

Improving on Space-Efficiency

- We would like to uncompute and reuse some of the bits.
- Uncomputing requires additional gates.
- So we must choose carefully when to uncompute in order to reduce the usage of space without needing too many more gates.
- First, we show how to obtain a reversible version using:
  - $O(t^{\log_2 3})$ gates;
  - $O(s \log t)$ bits
- Then we improve on this method to obtain:
  - $O(t^{1+\varepsilon})$ gates;
  - $O(s \log t)$ bits.

Uncomputing and Recomputing to Reuse Space

- The basic principle for combining the $r = \lceil \frac{t}{s} \rceil$ circuits R_i is indicated in the following:



- The idea is to uncompute and recompute parts of the state selectively to reuse the space.
- We systematically modify the computation $R_1 R_2 \ldots R_r$ to:
  - Reduce the total amount of space used;
  - Reset all the temporary bits to zero by the end of the computation.

Transformation $\mathcal{B}$

- To simplify the analysis, we take r to be a power of two, $r = 2^k$.
- For $1 \leq i \leq k$, let $r_i = 2^i$.
- We perform a recursive transformation $\mathcal{B}$ that:
  - Breaks a sequence into two equal-sized parts;
  - Recursively transforms the parts;
  - Composes the part as shown.

$$\begin{aligned}\mathcal{B}(R_1,\ldots,R_{r_{i+1}}) &= \mathcal{B}(R_1,\ldots,R_{r_i})\mathcal{B}(R_{1+r_i},\ldots,R_{r_{i+1}}) \\ &\qquad (\mathcal{B}(R_1,\ldots,R_{r_i}))^{-1}, \\ \mathcal{B}(R) &= R.\end{aligned}$$

- Note that $(\mathcal{B}(R_1,\ldots,R_{r_i}))^{-1}$ acts on exactly the same bits as $\mathcal{B}(R_1,\ldots,R_{r_i})$ and so requires no additional space.

Space Requirements

- The transformed computation uncomputes all space except the output of the last step.
- So the additional space usage is bounded by s.
- Thus, $\mathcal{B}(R_1, \ldots, R_{r_i})$ requires at most s more space than $\mathcal{B}(R_1, \ldots, R_{r_{i-1}})$.
- The recursion proceeds for $k = \log_2 r$ steps.
- For the space $S(i)$ required for step i, we have

$$S(i) \leq s + S(i-1), \quad S(1) \leq 2s.$$

- It follows that the final computation $\mathcal{B}(R_1, \ldots, R_r)$ requires space

$$S(r) \leq (k+1)s = s(\log_2 r + 1).$$

Gate Requirements

- Let $T(i)$ be the number of circuits R_j executed by the computation $\mathcal{B}(R_1, \ldots, R_{r_i})$.
- By the definition of $\mathcal{B}$, it follows that

$$T(i) = 3T(i-1), \quad T(1) = 1.$$

- So the number of reversible circuits R_i that our reversible version of C uses is

$$T(2^k) = 3T(2^{k-1}) = \cdots = 3^k = 3^{\log_2 r} = r^{\log_2 3}.$$

- Moreover, each requires fewer than $3s$ gates.
- Thus, any classical computation of t steps and s bits can be done reversibly in:
  - $O(t^{\log_2 3})$ steps;
  - $O(s \log_2 t)$ bits.

Improving the Bound

- To obtain the $O(t^{1+\varepsilon})$ bound, instead of using a binary decomposition, we consider an m-ary decomposition.
- For simplicity, suppose that r is a power of m, $r = m^k$.
- For $1 \leq i \leq k$, let $r_i = m^i$.
- We use the abbreviation

$$\vec{R}_{x,i} := R_{1+(x-1)r_i}, \ldots, R_{xr_i}.$$

- We have

$$\begin{array}{rcl} \mathcal{B}(\vec{R}_{1,i+1}) & = & \mathcal{B}(\vec{R}_{1,i}, \vec{R}_{2,i}, \ldots, \vec{R}_{m,i}) \\ & = & \mathcal{B}(\vec{R}_{1,i})\mathcal{B}(\vec{R}_{2,i})\cdots\mathcal{B}(\vec{R}_{m-1,i}), \\ & & \mathcal{B}(\vec{R}_{m,i}), \\ & & \mathcal{B}(\vec{R}_{m-1,i})^{-1}\cdots\mathcal{B}(\vec{R}_{2,i})^{-1}\mathcal{B}(\vec{R}_{1,i})^{-1}, \\ \mathcal{B}(R) & = & R. \end{array}$$

Improving the Bound (Cont'd)

- In each step of the recursion, each block is split into m pieces and replaced with $2m-1$ blocks.
- Since $r = m^k$, we stop recursing after k steps.
- At this point $r = m^k$ subcircuits C_1 have been replaced by $(2m-1)^k$ reversible circuits R_i.
- So the total number of circuits R_i for the final computation is $(2m-1)^k$.
- We rewrite this terms of r.

$$(2m-1)^{\log_m r} = r^{\log_m (2m-1)} \approx r^{\log_m 2m} = r^{1+\frac{1}{\log_2 m}}.$$

- The number of primitive gates in R_i is bounded by $3s$ and $r = \lceil \frac{t}{s} \rceil$.
- The total number of gates for a reversible circuit of t gates is

$$T(t) \approx 3s\left(\frac{t}{s}\right)^{1+\frac{1}{\log_2 m}} < 3t^{1+\frac{1}{\log_2 m}}.$$

Improving the Bound (Cont'd)

- For any $\epsilon > 0$, we may choose m large enough that $\frac{1}{\log_2 m} < \epsilon$.
- So the number of gates required is $O(t^{1+\epsilon})$.
- The space bound remains the same as before, $O(s \log_2 t)$.
- Reversible versions of classical Boolean circuits constructed in this manner can be turned directly into quantum circuits consisting entirely of Toffoli and X gates.
- Our argument was given in terms of Boolean circuits.
- The same argument shows that any classical Turing machine can be turned into a reversible one.
- Given any classical circuit for f, an implementation of U_f can be constructed of comparable number of gates and bits.

Uncomputing to Disentangle

- The care needed in uncomputing and reusing bits generalizes to qubits where the need for uncomputing values is even greater.
- Uncomputing ensures that temporary qubits are no longer entangled with output qubits.
- Unentangling temporary values at the end of a computation is one of the differences between classical and quantum implementations.
- Quantum transformations, being reversible, cannot reset qubits.
- One might think that temporary qubits could be reset by:
  - Measuring the qubit;
  - Depending on the measurement outcome, performing a transformation to set them to $|0\rangle$.
- But the temporary qubits may be entangled with qubits containing the desired result, or results used later in the computation.
- So measuring the temporary qubits may alter those results.

Subsection 3

A Language for Quantum Implementations

Notation for Registers

- Quantum variables are names for registers.
- Registers are taken to be subsets of qubits of a single global quantum register.
- Suppose x is the variable name for an n-qubit register.
- We write $x[n]$ if we wish to make the number of qubits in x explicit.
- We use x_i to refer to the i-th qubit of x.
- We wite $x_i \cdots x_k$ for qubits i through k of the register denoted by x.
- We order the qubits of a register from highest index to lowest index.
- So, if register x contains a standard basis vector $|b\rangle$, then

$$b = \sum_i x_i 2^i.$$

Notation for Transformations

- Suppose U is a unitary transformation on n qubits.

  Let x, y and z be names for registers with total length n qubits.

  Then the program step

  $$U|x,y,z\rangle = U|x\rangle|y\rangle|z\rangle$$

  means "apply U to the qubits denoted by the register names in the order given".
- It is illegal to use any qubit twice in this notation.

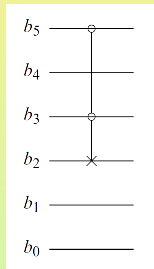
  So the registers x, y and z must be disjoint.

  This restriction is necessary because "wiring" different input values to the same quantum bit is not possible or even meaningful.
- Note that the ket notation is slightly abused, being used to stand for:
  - A placeholder, a qubit, that can contain a qubit value, a quantum state;
  - The qubit value itself.

  Context should keep the two uses clear.

Example

- The Toffoli gate, with control bits b_5 and b_3 and target bit b_2, has the graphical representation shown.
- It is awkward to represent in the standard tensor product notation because the qubits it acts on are not adjacent.



- In our notation, this transformation can be written as

$$T|b_5, b_3, b_2\rangle.$$

- The notation $T|b_2, b_3, b_2\rangle$ is not allowed, since it repeats qubits.

Example (Cont'd)

- Consider the notation
$$(T \otimes C_{\text{not}} \otimes H)|x_5 \cdots x_3\rangle|x_1, x_0\rangle|x_7\rangle,$$
for a transformation acting on six qubits of a ten-qubit register $x = x_9 x_8 \cdots x_0$.
- It is another way of representing the transformation
$$I \otimes I \otimes H \otimes I \otimes T \otimes I \otimes C_{\text{not}},$$
where the separate kets indicate which qubits the transformation making up the tensor product is acting upon.
    - The Toffoli gate T acts on qubits x_5, x_4 and x_3.
    - The C_{not} acts on qubits x_1 and x_0.
    - The Hadamard gate H acts on qubit x_7.
- The notation
$$(T \otimes C_{\text{not}} \otimes H)|x_5 \cdots x_3\rangle|x_4, x_0\rangle|x_7\rangle$$
is illegal, as the first and second registers are not disjoint.

Control

- Controlled operations are so frequently used that we give them their own notation.
- Let b and x be disjoint registers.
- The notation
$$|b\rangle \textbf{ control } U|x\rangle$$
means that, on any standard basis vector, the operator U is applied to the contents of register x only if all of the bits in b are 1.
- Writing $\neg|b\rangle$ **control** $U|x\rangle$ is a shorthand for the sequence
$$\begin{array}{l} X \otimes \cdots \otimes X|b\rangle \\ |b\rangle \textbf{ control } U|x, y\rangle \\ X \otimes \cdots \otimes X|b\rangle. \end{array}$$
- If we write a sequence of state transformations, they are intended to be applied in order.

Qubit for Local Temporary Registers

- We allow programs to declare local temporary registers using

$$\textbf{qubit } t[n].$$

- However, the program must restore the qubits in these registers to their initial $|0\rangle$ state.
- This condition ensures that:
  - Temporary qubits can be reused for different executions of the program;
  - The overall storage requirement is bounded.
- It also ensures that the temporary qubits do not remain entangled with the other registers.

Define

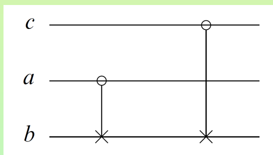
- We allow naming sequences of program steps.
- This is done using command **define**.
- Unlike commands such as **control**, the command **define** does not do anything to the qubits.
- It defines a new function by telling the machine what sequence of commands a new function variable name represents.

Example

- Define addition modulo 2, with an incoming carry bit.

$$\begin{aligned}&Sum: |c, a, b\rangle \to |c, a, (a+b+c) \mod 2\rangle\\&\textbf{define } Sum\ |c\rangle|a\rangle|b\rangle =\\&|a\rangle \textbf{ control } X|b\rangle\\&|c\rangle \textbf{ control } X|b\rangle.\end{aligned}$$

- It operates on three single qubits by adding the value of a and the value of the carry c to the value of b.
- The program would be drawn as the circuit



Carry

- A corresponding carry operator is of the form

$$Carry : |c, a, b, c'\rangle \to |c, a, b, c' \oplus C(a, b, c)\rangle,$$

where the carry $C(a, b, c)$ is 1 if two or more of the bits a, b, c, are 1,

$$C(a, b, c) = (a \wedge b) \oplus (c \wedge (a \oplus b)).$$

- A program for Carry might look like

define $Carry\ |c, a, b, c'\rangle$ =	
$|a\rangle|b\rangle$ **control** $X|c'\rangle$	Compute $a \wedge b$ in register c'
$|a\rangle$ **control** $X|b\rangle$	Compute $a \oplus b$ in register b
$|c\rangle|b\rangle$ **control** $X|c'\rangle$	Toggle result c' if c and current value of b
$|a\rangle$ **control** $X|b\rangle$	Reset b to original value

Carry (Cont'd)

- A program for Carry.

define $Carry\ \|c, a, b, c'\rangle =$	
$\|a\rangle\|b\rangle$ **control** $X\|c'\rangle$	Compute $a \wedge b$ in register c'
$\|a\rangle$ **control** $X\|b\rangle$	Compute $a \oplus b$ in register b
$\|c\rangle\|b\rangle$ **control** $X\|c'\rangle$	Toggle result c' if c and current value of b
$\|a\rangle$ **control** $X\|b\rangle$	Reset b to original value

- Starting in Step (2), register b temporarily holds the XOR of the original values of a and b.
- In Step (3), this value of register b means that c' is toggled if c and exactly one of the original values of a and b is 1.
- Register b is reset to its original value in Step (4).

Classical versus Quantum Registers

- Repetition and conditional execution of sequences of quantum state transformations can be controlled using classical programming constructs.
- Only classical, not quantum, information can be used in the control structure.
- However, in quantum algorithms there is a choice as to:
  - Which classical input values are placed in quantum registers;
  - Which input values are used simply as part of the classical control structure.

Classical versus Quantum Registers (Cont'd)

- For instance, one program to add x to itself n times might take classical input n and use it only as part of the classical control.
- Another might place n in an additional quantum register.
- The two programs would be of the form

$$\begin{aligned} A_n &: |x, 0\rangle \mapsto |x, nx\rangle, \\ A &: |x, n, 0\rangle \mapsto |x, n, nx\rangle, \end{aligned}$$

  respectively.
- This distinction will be more important when we consider quantum algorithms that act on superpositions of input values.
- Only input values placed in quantum registers, not input values that are part of the classical control structure, can be in superposition.

Recursion

- The definition of a new program may use the same program recursively, provided that the recursion can be unwound classically.
- Recursive application of functions is allowed only as a shorthand for a classically specified sequence of quantum transformations.
- We can use the

$$\textbf{qubit}\ t[n]$$

  construction recursively, as long as the recursion depth is bounded by a static classical constant.

Subsection 4

Some Example Programs for Arithmetic Operations

Transformation *Flip*

- We define a transformation *Flip* that generalizes the Toffoli gate T.
- The transformation *Flip* acts on:
  - An m-qubit register $a = |a_{m-1} \dots a_0\rangle$;
  - An $(m-1)$-qubit register $b = |b_{m-2} \dots b_0\rangle$.
- It negates qubit b_i exactly when the following $(i+2)$-conjunction holds,

$$\bigwedge_{j=0}^{i+1} a_j.$$

- We define *Flip* in terms of Toffoli gates T.
- These Toffoli gates perform bit flips on some of the qubits of register b depending on the contents of a.

Transformation *Flip* (Cont'd)

- We encode as follows.

define $Flip\ |a[2]\rangle|b[1]\rangle =$ (base case $m = 2$)
$T|a_1\rangle|a_0\rangle|b\rangle$
define $Flip\ |a[m]\rangle|b[m-1]\rangle =$ (general case $m \geq 3$)
$T|a_{m-1}\rangle|b_{m-3}\rangle|b_{m-2}\rangle$
$Flip\ |a_{m-2}\dots a_0\rangle|b_{m-3}\dots b_0\rangle$
$T|a_{m-1}|b_{m-3}|b_{m-2}\rangle$.

- An inductive argument shows that *Flip*, when defined in this way, behaves as described.
- The transformation *Flip*, when applied to an m-qubit register a and an $(m-1)$-qubit register b, uses $2(m-2)+1$ Toffoli gates T.

The *AndTemp* Operation

- Next we define an *AndTemp* operation that acts on:
  - An m-qubit register a;
  - An $(m-2)$-qubit register c;
  - A single qubit register b.
- It stores in b the conjunction of bits in a, making temporary use of the qubits in c.
- We will use *AndTemp* to construct an AND operation that makes more efficient use of qubits.

The *AndTemp* Operation (Cont'd)

- We encode as follows.

  define *AndTemp* $|a[2]\rangle|b[1]\rangle =$ (base case $m = 2$)
  $T|a_1\rangle|a_0\rangle|b\rangle$
  define *AndTemp* $|a[m]\rangle|b[1]\rangle|c[m-2]\rangle =$ (general case $m - 3$)
  Flip $|a\rangle(|b\rangle|c\rangle)$ Compute conjunction in b (1)
  Flip $|a_{m-2}\dots a_0\rangle|c\rangle$ Reset c (2)

- The parentheses in $Flip|a\rangle(|b\rangle|c\rangle)$ indicate that *Flip* is applied to:
  - The m-qubit register a;
  - The $m-1$ qubit register that is the concatenation of registers b and c.
- By the definition of *Flip*, Step (1) leaves the conjunction of the a_j in b but changes the contents of c in the process.
- Step (2) undoes these changes to c.

The *AndTemp* Operation (Cont'd)

- *AndTemp* requires $4m - 8$ gates:
  - The first *Flip* uses $2(m-2)+1$ Toffoli gates;
  - The second *Flip* uses $2(m-3)+1$ Toffoli gates.
- The $m-2$ qubits in register c can be in any state at the start of the computation.
- They will be returned to their original states by the end of the computation.
- So we can use these to compute the m-way AND, provided there are sufficiently many computational qubits already ($n \geq 2m-2$).
- Clever use of this property of *AndTemp* will allow us to define an *And* on up to n qubits that uses only 1 additional temporary qubit.

Efficient Implementation of AND

- To construct the conjunction using less space, we recursively use *AndTemp* on one half of the qubits, using the other half temporarily and vice versa.
- Thus, a general *And* operator that requires a single temporary qubit can be defined as follows.
- Let $k = \lfloor \frac{m}{2} \rfloor$ and define

$$j = \begin{cases} k-2, & \text{for even } m, \\ k-1, & \text{for odd } m. \end{cases}$$

- The operator *And* flips b if and only if all bits of a are 1.

define $And\ |a[1]\rangle|b[1]\rangle =$ Trivial unary case, $m = 1$
  $C_{\text{not}}|a_0\rangle|b\rangle$
define $And\ |a[2]\rangle|b[1]\rangle =$ Binary case, $m = 2$
  $T|a_1\rangle|a_0\rangle|b\rangle$

Efficient Implementation of AND (Cont'd)

- For the general case, we encode as follows.

define *And* $\|a[m]\rangle\|b\rangle =$	General case, $3 \leq m$
qubit $t[1]$	use a temporary qubit
AndTemp $\|a_{m-1} \dots a_k\rangle\|t\rangle\|a_j \dots a_0\rangle$	(1)
AndTemp $(\|t\rangle\|a_j \dots a_0\rangle)\|b\rangle\|a_{k+j-2} \dots a_k\rangle$	(2)
AndTemp $\|a_{m-1} \dots a_k\rangle\|t\rangle\|a_j \dots a_0\rangle$	(3)

- Step (1) computes the conjunction of the high-order bits using the low-order bits temporarily.
- In step (2) we compute the conjunction of the low-order bits using the high-order bits temporarily.
- Since *AndTemp* uses a linear number of gates, so does *And*.

Multiply Controlled Single-Qubit Transformations

- The linear implementation of *And* enables a linear implementation of the multiply controlled single-qubit transformations $\bigwedge_x^i Q$.
- Given an m-bit bit string z, let $X^{(z)}$ be the transformation

$$X^{(z)} = X \otimes I \otimes \cdots \otimes X \otimes X,$$

  which contains:
  - An X at any position where z has a 0 bit;
  - An I at any position where z has a 1 bit.
- We implement the transformation $\mathit{Conditional}(z, Q)$, which acts on qubit b with single-qubit transformation Q if and only if the bits of register a match bit string z.

Multiply Controlled Single-Qubit Transformations (Cont'd)

- We encode as follows.

define $Conditional(z, Q)|a[m]\rangle|b[1]\rangle =$

qubit t	use a temporary qubit	(1)
$X^{(z)}|a\rangle$	if a and z match, a becomes all 1's	(2)
$And\ |a\rangle|t\rangle$	AND bits of a	(3)
$|t\rangle$ **control** $Q|b\rangle$	if a matched z, apply Q to b	(4)
$And\ |a\rangle|t\rangle$	uncompute AND	(5)
$X^{(z)}|a\rangle$	uncompute match	(6)

- This construction uses 2 additional qubits and only $O(m)$ simple gates.
- When z is $11\ldots1$ and $Q = X$, then $Conditional(z, Q)$ is simply the *And* operator.

In-Place Addition

- We define an *Add* transformation that adds two n-bit binary numbers.
- The transformation

$$Add : |c\rangle|a\rangle|b\rangle \to |c\rangle|a\rangle|(a+b+c) \mod 2^{n+1}\rangle$$

  acts on:
  - Two n-qubit registers a and c;
  - An $(n+1)$-qubit register b.
- It adds two n-bit numbers, placed in registers a and b, and puts the result in register b, when register c and the highest order bit, b_n, of register b are initially 0.
- The implementation of *Add* uses n recursion steps, where n is the number of bits in the numbers to be added.
- The ith step in the recursion adds the $n-i$ highest bits, with the carry in the lowest of these $n-i$ highest bits having first been computed.
- The construction uses *Sum* and *Carry* defined previously.

In-Place Addition (Cont'd)

- We consider the two cases, $n = 1$ and $n > 1$:

define $Add\ |c\rangle|a\rangle|b[2]\rangle =$	base case $n = 1$	
$Carry\ |c\rangle|a\rangle|b_0\rangle|b_1\rangle$	carry in high bit of b	(1)
$Sum\ |c\rangle|a\rangle|b_0\rangle$	sum in low bit of b	(2)
define $Add\ |c[n]\rangle|a[n]\rangle|b[n+1]\rangle =$	general case $n > 1$	
$Carry\ |c_0\rangle|a_0\rangle|b_0\rangle|c_1\rangle$	compute the carry for low bits	(3)
$Add\ |c_{n-1} \dots c_1\rangle|a_{n-1} \dots a_1\rangle|b_n \dots b_1\rangle$	add $n-1$ highest bits	(4)
$Carry^{-1}|c_0\rangle|a_0\rangle|b_0\rangle|c_1\rangle$	uncompute the carry	(5)
$Sum\ |c_0\rangle|a_0\rangle|b_0\rangle$	compute the low order bit	(6)

- Step (5) is needed to ensure that the carry register is reset to its initial value.
- The $Carry^{-1}$ operator is implemented by running, in reverse order, the inverse of each transformation in the definition of the *Carry* operator.

Modular Addition

- The following program defines modular addition for n-bit binary numbers a and b,

$$AddMod\ |a\rangle|b\rangle|M\rangle \to |a\rangle|(b+a) \mod M\rangle|M\rangle,$$

  where:
  - a and M are n-qubit registers;
  - b is an $(n+1)$-qubit register.
- When the highest order bit, b_n, of register b is initially 0, the transformation $AddMod$ replaces the contents of register b with $b+a$ mod M, where M is the contents of register M.
- The contents of registers a and M (and the temporaries c and t) are unchanged by $AddMod$.
- The construction makes use of the Add transformation we defined previously.

Modular Addition (Cont'd)

- We encode as follows.

define $AddMod\ |a[n]\rangle|b[n+1]\rangle|M[n]\rangle =$

qubit t	use a temporary bit	(1)
qubit $c[n]$	storage for the n-bit carry	(2)
$Add\ \|c\rangle\|a\rangle\|b\rangle$	add a to b	(3)
$Add^{-1}\|c\rangle\|M\rangle\|b\rangle$	subtract M from b	(4)
$\|b_n\rangle$ **control** $X\|t\rangle$	toggle t when underflow	(5)
$\|t\rangle$ **control** $Add\ \|c\rangle\|M\rangle\|b\rangle$	when underflow, add M back to b	(6)
$Add^{-1}\|c\rangle\|a\rangle\|b\rangle$	subtract a again	(7)
$\neg\|b_n\rangle$ **control** $X\|t\rangle$	reset t	(8)
$Add\ \|c\rangle\|a\rangle\|b\rangle$	construct final result	(9)

- Classically, Steps (3) through (6) are all that are needed.
- In Step (4), if $M > b$, subtracting M from b causes b_n to become 1.
- Steps (7) through (9) are needed to reset t.

Modular Addition (Cont'd)

- Note that each *Add* operation internally resets $|c\rangle$ back to its original value.
- The condition $0 \leq a, b < M$ is necessary.
- For values outside that range, an operation that sends $|a, b, M\rangle$ to $|a, b + a \mod M, M\rangle$ is not reversible and therefore not unitary.
- If this condition does not hold, for example if $b \geq M$ initially, then the final value of b may still be greater than M, since the algorithm subtracts M at most once.

Modular Multiplication

- The *TimesMod* transformation multiplies two n-bit binary numbers a and b modulo another n-bit binary number M.
- The transformation

$$TimesMod\ |a\rangle|b\rangle|M\rangle|p\rangle \to |a\rangle|b\rangle|M\rangle|(p+ba) \mod M\rangle$$

  is defined by the following program that successively adds $b_i 2^i a$ mod M to the result register p.
- It is assumed that $a < M$, but b can be arbitrary.
- Both a and p are $(n+1)$-qubit registers.
- The additional high-order bit is needed for intermediate results.
- The operation *Shift* simply cyclically shifts all bits by 1.
- This can be done by swapping bits a_{i+1} with a_i for all i, starting with the high-order bits.
- *Shift* acts as multiplication by 2, since the high-order bit of a will be 0.

Modular Multiplication (Cont'd)

- We encode as follows.

define $\mathit{TimesMod}\ |a[n+1]\rangle|b[k]\rangle|M[n]\rangle|p[n+1]\rangle =$

Code	Comment	#
qubit $t[k]$	use k temporary bits	(1)
qubit $c[n]$	carry register for addition	(2)
for $i \in [0 \dots k-1]$	iterate through bits of b	(3)
$\mathit{Add}^{-1}|c\rangle|M\rangle|a\rangle$	subtract M from a	(4)
$|a_n\rangle$ **control** $X|t_i\rangle$	$t_i = 1$ if $M > a$	(5)
$|t_i\rangle$ **control** $\mathit{Add}\ |c\rangle|M\rangle|a\rangle$	add M to a if t_i is set	(6)
$|b_i\rangle$ **control** $\mathit{AddMod}\ |a_{n-1} \dots a_0\rangle|p\rangle|M\rangle$	add a to p if b_i is set	(7)
$\mathit{Shift}\ |a\rangle$	multiply a by 2	(8)
for $i \in [k-1 \dots 0]$	clear t and restore a	(9)
$\mathit{Shift}^{-1}|a\rangle$	divide a by 2	(10)
$|t_i\rangle$ **control** $\mathit{Add}^{-1}|c\rangle|M\rangle|a\rangle$	perform all steps in reverse	(11)
$|a_n\rangle$ **control** $X|t_i\rangle$	clear ith bit of t	(12)
$\mathit{Add}\ |c\rangle|M\rangle|a\rangle$	add M to a	(13)

Modular Multiplication (Cont'd)

- Lines (4)-(6) compute the $a \mod M$.
- The second loop, (9)-(13), undoes all the steps of the first one, (3)-(8), except the conditional addition to the output p (Line 7).
- The transformation that sends

$$|a, b, M\rangle \mapsto |a, ab \mod M, M\rangle$$

is not unitary.

This is seen, e.g., by

$$\begin{array}{rcl} |2,1,4\rangle & \mapsto & |2,2,4\rangle; \\ |2,3,4\rangle & \mapsto & |2,2,4\rangle. \end{array}$$

- So modular multiplication cannot be defined as an in-place operation.

Modular Exponentiation (Copy Transformation)

- We implement modular exponentiation,

$$ExpMod\ |a\rangle|b\rangle|M\rangle|0\rangle \to |a\rangle|b\rangle|M\rangle|ab \mod M\rangle$$

  using $O(n^2)$ temporary qubits where a, b and M are n-qubit registers.
- The *Copy* transformation

$$Copy : |a\rangle|b\rangle \to |a\rangle|a \oplus b\rangle$$

  copies the contents of an n-bit register a to another n-bit register b whenever the register b is initialized to 0.
- The operation *Copy* can be implemented as bitwise XOR operations between the corresponding bits in registers a and b.

define $Copy\ |a[n]\rangle|b[n]\rangle =$
  for $i \in [0 \dots n-1]$ bit-wise
    $|a_i\rangle$ **control** $X|b_i\rangle$ XOR a with b

Modular Exponentiation (SquareMod Transformation)

- The modular squaring operation *SquareMod*

$$SquareMod : |a\rangle|M\rangle|s\rangle \to |a\rangle|M\rangle|(s+a^2) \mod M\rangle$$

places the result of squaring the contents of register a, modulo the contents of register M, in the register s.

- We encode as follows.

$$\begin{array}{lll} \textbf{define } SquareMod\ |a[n+1]\rangle|M[n]\rangle|s[n+1]\rangle = & & \\ \quad \textbf{qubit } t[n] & \text{use } n \text{ temporary bits} & (1) \\ \quad Copy\ |a_{n-1}\dots a_0\rangle|t\rangle & \text{copy } n \text{ bits of } a \text{ to } t & (2) \\ \quad TimesMod\ |a\rangle|t\rangle|M\rangle|s\rangle & \text{compute } a^2 \mod M & (3) \\ \quad Copy^{-1}\ |a_{n-1}\dots a_0\rangle|t\rangle & \text{clear } t & (4) \end{array}$$

Modular Exponentiation

- We give a recursive definition of modular exponentiation

$$ExpMod : |a\rangle|b\rangle|M\rangle|p\rangle|e\rangle \to |a\rangle|b\rangle|M\rangle|p\rangle|e \oplus (pa^b) \mod M\rangle.$$

- We encode fist the base case.

$$\begin{array}{lll}
\textbf{define } \mathit{ExpMod}\ |a[n+1]\rangle|b[1]\rangle|M[n]\rangle|p[n+1]\rangle|e[n+1]\rangle = & & \text{base case} \\
\quad \neg|b_0\rangle \textbf{ control } \mathit{Copy}|p\rangle|e\rangle & \text{result is } p & (1) \\
\quad |b_0\rangle \textbf{ control } \mathit{TimesMod}\ |a\rangle|p\rangle|M\rangle|e\rangle & \text{result is } pa^1 \mod M & (2)
\end{array}$$

Modular Exponentiation (Cont'd)

- For the general case, we have

define $\mathit{ExpMod}\ \|a[n+1]\rangle\|b[k]\rangle\|M[n]\rangle\|e[n+1]\rangle =$	general case $k > 1$	
qubit $u[n+1]$	for $a^2 \mod M$	(3)
qubit $v[n+1]$	for $(pa^{b_0}) \mod M$	(4)
$\neg\|b_0\rangle$ **control** $\mathit{Copy}\|p\rangle\|v\rangle$	$v = pa^0 \mod M$	(5)
$\|b_0\rangle$ **control** $\mathit{TimesMod}\ \|a\rangle\|p\rangle\|M\rangle\|e\rangle$	$e = pa^1 \mod M$	(6)
$\mathit{SquareMod}\ \|a\rangle\|M\rangle\|u\rangle$	compute $a^2 \mod M$ in u	(7)
$\mathit{ExpMod}\ \|u\rangle\|b_{k-1}\dots b_1\rangle\|M\rangle\|v\rangle\|e\rangle$	compute $v(a^2)^{b/2} \mod M$	(8)
$\mathit{SquareMod}^{-1}\|a\rangle\|M\rangle\|u\rangle$	uncompute u	(9)
$\|b_0\rangle$ **control** $\mathit{TimesMod}^{-1}\|a\rangle\|p\rangle\|M\rangle\|e\rangle$	uncompute e	(10)
$\neg\|b_0\rangle$ **control** $\mathit{Copy}^{-1}\|p\rangle\|v\rangle$	uncompute v	(11)

Modular Exponentiation (Comments)

- The program unfolds recursively k times, once for each bit of b.
- Steps (5)-(8), together with the base case, Steps (1) and (2), perform the classical computation.
- The division $\frac{b}{2}$ in Step (8) is integer division.
- Each recursive step requires two temporary registers of size $n+1$ that are reset at the end in Steps (9) and (11).
- Thus, the algorithm requires a total of $2(k-1)(n+1)$ temporary qubits.
- The algorithm for modular multiplication requires $O(n^2)$ steps to multiply two n-bit numbers.
- Thus, the modular exponentiation requires $O(kn^2)$ steps.
- More efficient multiplication algorithms are possible and the complexity can be reduced to $O(kn \log n \log \log n)$.