CS70, Fall 2003 **Discussion #12** **Amir Kamil**
UC Berkeley **12/5/03**

Topics: Computability, Quantum Factoring

1 Computability

In this section, we take a look at some advanced topics in computational theory. Besides the halting problem, our discussion is for entertainment purposes only. You are not responsible for the rest of the material here.

1.1 Introduction

In computability and complexity theory, all problems are reduced to language recognition, where a *language* is defined as a set of strings over some alphabet, typically $\{0, 1\}$. For example, $L_1 = \{10, 11, 101, \cdots\}$ is the language that contains all prime numbers, represented in binary. Then the problem of determining whether or not a number x is a prime is the same as determining whether or not $x \in L_1$.

As a more complicated example, consider the problem of factoring a number N. The language corresponding to this problem is $L_2 = \{\langle x, k\rangle : x \text{ has a nontrivial factor} \leq k\}$, where $\langle x, k\rangle$ denotes an encoding of (x, k) in binary. Then we can determine the factors of N by repeatedly determining whether $\langle N, m\rangle \in L_2$, doing a binary search on m. (For example, in the case of $N = 35$, we check $\langle 35, 35\rangle$, $\langle 35, 17\rangle$, $\langle 35, 8\rangle$, $\langle 35, 4\rangle$, $\langle 35, 6\rangle$, and finally $\langle 35, 5\rangle$ in order to determine that 5 is a factor of 35.)

With this formulation of a problem, we show that there exist problems that are not computable.

1.2 Uncomputability of Problems

It may seem at first that any problem we can come up with can be solved on a computer. This is not the case, and we can give a non-constructive proof from basic countability theory. Later, we will see actual examples of uncomputable problems.

Consider the set of all problems P. In our formulation above, this is the set of all languages over the alphabet $\{0, 1\}$. We prove that P is uncountable using diagonalization. Suppose that P is countable. Then there exists an enumeration over P, $\{L_1, L_2, \cdots\}$. We construct a new language L' not in this enumeration as follows. If $i \in L_i$, then let $i \notin L'$, but if $i \notin L_i$, then let $i \in L'$. Note that since L' differs from each L_i on the string i, L' must not be in the enumeration. But L' is a language, since it is a set of strings. This is a contradiction, so no such enumeration can exist, and P is uncountable.

Now consider the set of all algorithms A. Any algorithm can be represented as a bit string, since it is just a sequence of symbols. Thus the set of algorithms is a subset of the set of all bit strings. Since the latter is countable, A is also countable.

Since there are more problems than there are algorithms, there must be problems that are algorithmically unsolvable. In fact, there are uncountably infinitely many more problems that are unsolvable than there are that are solvable.

1.3 The Halting Problem

The classic example of an uncomputable problem is that of determining whether or not a program P halts on a given input x. We show that there is no algorithm to solve this problem.

Suppose an algorithm `halts(`P`, `x`)` exists to solve the halting problem. Then consider the following function:

```
Turing($P$):
1.  If halts($P$, $P$) then:  loop forever;
2.  Else:  halt;
```

This program is very simple. It checks if its input, which is a program, halts when given itself as an input. It then does the opposite.

Now what happens when we call `Turing(Turing)`? Well if `Turing(Turing)` halts, then in line 1, `Turing(Turing)` will go into an infinite loop, which is a contradiction. And if `Turing(Turing)` loops, then `Turing(Turing)` halts in line 2, which is also a contradiction. The only assumption that we made was that `halts(`P`, `x`)` exists, so this assumption must have been invalid.

1.4 Program Minimality

Consider another interesting question. Given a program `P(`x`)` with a particular behavior (where *behavior* is defined by the output and side effects of a program given a particular input), is it the minimal program (in terms of size) that behaves in this way? Here we show that no algorithm exists that can answer this question.

Suppose that there does exist an algorithm `minimal(`P`)` that solves the minimality problem. Then we write the following function:

```
paradox($x$):
1.  Let $y$ = own source code.
2.  For $i$ := $y+1$ to $\infty$ do:
3.    If $i$ is a valid program and minimal($i$) then:
4.      Simulate $i$ on $x$;
5.      Halt;
```

There are a few subtle points about this program. First, the program can obtain its own source code, by reading itself from disk or memory. (If you are familiar with Turing machines, the *recursion theorem* allows a Turing machine to obtain a description of itself.) Second, a program is just a bit string, so we can add a number to this bit string to obtain another program. We can also check that a bit string is a valid program by running a compiler on it. Lastly, it is possible for a program to simulate another, using something similar to the metacircular evaluator you've seen in CS61A. Thus this is a valid program.

Now let's turn to the behavior of this program. It searches for a program longer than itself that it is minimal. Note that it will always find one, since the sets of possible programs and program behaviors are (countably) infinite. Once it finds one, it simulates that program.

But this is a contradiction! The program i that `paradox(`x`)` found must be minimal, and it also must be longer than `paradox(`x`)`. But `paradox(`x`)` simulates i, so i cannot be minimal. Thus `minimal(`P`)` must not exist.

1.5 Membership

Consider the set of boolean programs that take in a bit string as input[1], i.e. of the form

```
boolean prog($x$):
  ...
  return true;
  ...
  return false;
```

We can define the language L_{prog} that corresponds to such a program as the set of strings that the program *accepts*, i.e for which it returns ***true***. The inputs the program *rejects*, or returns ***false*** for, and those for which it loops are not in its language. (Notice that two programs that have the same behavior have equivalent languages.) The membership question, then is, given a particular program P and its input x, is $x \in L_P$? We prove that this problem is uncomputable. Note that the language corresponding to membership is $A = \{\langle P, x\rangle : P \text{ accepts } x\}$.

[1]We could always convert a function that takes in no inputs to one that does and ignores its input. Similarly, we could convert a program that takes in multiple inputs to one that takes in only one, by using a sufficient encoding of the inputs.

Suppose we had a function `accepts(`P`, `x`)` that solves the membership problem. Recall that the set of all programs is countable, so we can enumerate it as $\{P_1, P_2, \cdots\}$. Then we can construct the following program:

```
boolean $P'(x)$:
1.  If accepts($P_x$, $x$) then:  return *false*;
2.  Else:  return *true*;
```

Note that this is a program, so it must appear in our enumeration. However, it differs from every program in the enumeration, specifically from P_i at input i. This is a contradiction, so `accepts(`P`, `x`)` must not exist.

Now we can give an alternative proof that the halting problem is unsolvable, using a reduction. A *reduction* is a conversion from an input to one language to an input to another, such that the converted input is in the second language if and only if the original is in the first. In other words, a reduction from a language X to a language Y is a function R such that $x \in X \Longleftrightarrow R(x) \in Y$. This means that if the second language is computable, so is the first, since we can solve it by running the converter followed by the solver for the second. The contrapositive is that if the first is unsolvable, so is the second. We can reduce the membership problem to the halting problem using the following function:

```
reduce($P$, $x$):
1.  Return the following program:
    "$P'(x)$:
    1.  If $P(x)$ then:  return *true*;
    2.  Else:  loop forever;";
```

Now notice that $P'(x)$ halts if and only if $P(x)$ accepts. Thus if we had a function `halts` that solves the halting problem, we could solve the membership problem by calling `halts(`P'`, `x`)` or `halts(reduce(`P`, `x`) `x`)`. Since we know that the membership problem is unsolvable, such a function must not exist, and the halting problem must be unsolvable.

Another useful fact is that if a language is computable, so is its complement, since we can just run the solver for the former and negate the output. (Recall that the complement of a language L is $\overline{L} = \{x : x \notin L\}$.) For example, the complement of membership, non-membership, is $\overline{A} = \{\langle P, x\rangle : P \text{ does not accept } x\}$. Since membership is uncomputable, neither is non-membership.

1.6 Rice's Theorem

We now prove a general result, *Rice's theorem*, that any nontrivial question about the behavior of a program is uncomputable. By *nontrivial*, we mean a question for which the answer is not the same for every program. Again, we restrict ourselves to boolean programs that take in a single bit string as input.

Consider an arbitrary question about the behavior of a program. This problem has a corresponding language, Q, such that a program P's language L_P is in Q if the answer to the question is yes for P, and L_P is not in Q if the answer is no. Suppose we have a solver `q(`P`)` for Q. Then we can reduce the membership or non-membership problems to Q. There are two cases:

Case 1: $\Phi \in Q$, where Φ is the empty language.
We use the following function to reduce non-membership to Q:

```
reduce($P$, $x$):
1.  Enumerate all programs until one $R_1$ is found such that q($R_1$) = *false*, i.e.  $L_{R_1} \notin Q$.
2.  Return the following program:
    "$P'(y)$:
    1.  If $P(x)$ then:  return $R_1(x)$;
    2.  Else:  return *false*;";
```

Now consider the language of P', $L_{P'}$. If P accepts x, then P simulates R_1, so $L_{P'} = L_{R_1} \notin Q$. Thus

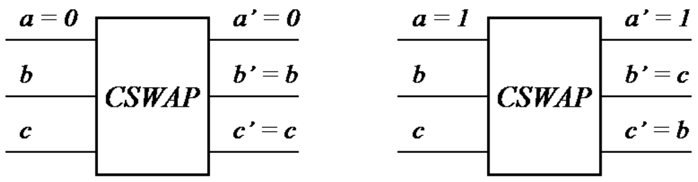

Figure 1: The $CSWAP$ gate swaps its last two inputs if the first is 1.

q will reject P'. On the other hand, if P does not accept x, either by looping or rejecting, then P' either loops on or rejects all inputs, so $L_{P'} = \Phi \in Q$ and q will accept P'. Thus $\langle P, x\rangle \in \overline{A}$ if and only if $L_{P'} \in Q$. Since $\overline{A}$ is uncomputable, this implies that Q is uncomputable and q does not exist.

Case 2: $\Phi \notin Q$.
We use the following function to reduce membership to Q:

```
reduce($P$, $x$):
1.  Enumerate all programs until one $R_2$ is found such that q($R_2$) = *true*, i.e.  $L_{R_2} \in Q$.
2.  Return the following program:
    "$P'(y)$:
    1.  If $P(x)$ then:  return $R_2(x)$;
    2.  Else:  return *false*;";
```

Now consider the language of P', $L_{P'}$. If P accepts x, then P simulates R_2, so $L_{P'} = L_{R_2} \in Q$. Thus q will accept P'. On the other hand, if P does not accept x, either by looping or rejecting, then P' either loops on or rejects all inputs, so $L_{P'} = \Phi \notin Q$ and q will reject P'. Thus $\langle P, x\rangle \in A$ if and only if $L_{P'} \in Q$. Since A is uncomputable, this implies that Q is uncomputable and q does not exist.

Note that the only assumptions made in the above proof are that the set of all programs is enumerable and that there exist programs R_1 and R_2 such that $L_{R_1} \notin Q$ and $L_{R_2} \in Q$. The former is true since we know that the set of programs is countable, and the latter is true since Q is nontrivial.

2 Quantum Computation

By now, you've probably all heard of quantum computation and how it is so much faster than classical computation. We show how to convert an arbitrary classical circuit into a quantum circuit, introduce quantum factoring, and finally discuss the speedups that quantum computation has to offer.

2.1 Reversible Computation

Quantum mechanics imposes a restriction on computation: the computation must be reversible. Is this a problem? Recall that any circuit can be produced from two basic pieces: the NOT gate and the AND gate. The NOT gate is reversible, since it is its own inverse. The AND gate, however, is not, since it destroys information. For example, if we know the output of an AND gate is 0, we cannot tell what the input was. Does this mean that quantum computation cannot do what classical computation can?

Fortunately, C. Bennet proved in 1973 that we can convert any non-reversible circuit to a reversible one. The conversion involves the three-input $CSWAP$ gate in figure 1. This gate swaps its second and third inputs if its first is 1, but does nothing if its first is 0. In either case, the first input remains unchanged. Note that the gate is reversible, since it is its own inverse.

Using the $CSWAP$, we can build an AND gate. We send in the AND inputs a and b as the first two inputs to the $CSWAP$, and 0 as the third. Then the third output will be $c' = a \wedge b$ (check this!). We ignore the other two outputs.

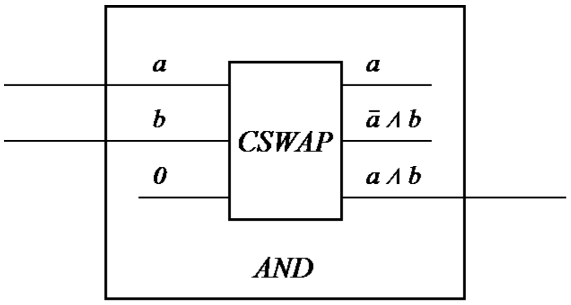

Figure 2: The $CSWAP$ gate used to implement the AND gate.

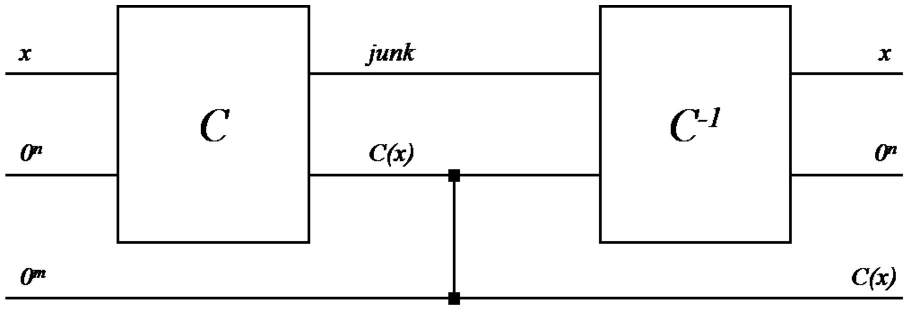

Figure 3: A reversible circuit with clean output.

Thus we can convert any circuit built of NOT and AND into a reversible one composed only of NOT and $CSWAP$. This conversion, however, requires extra zeros on the input, and produces extra garbage on output. For a quantum computer, the extra garbage is bad, since the output state may be entangled with the garbage. However, we can remove the garbage by copying the output and then running the output and garbage through the reverse of the circuit, ending up with only the input, output, and some unchanged zeros, as in figure 3. This circuit takes $(x, 0^m)$ to $(x, f(x))$, ignoring the unchanged zeros.

2.2 Quantum Circuits

Using the construction above, given a classical circuit F, we can build a quantum circuit that takes the state $|x\rangle|0\rangle$ to $|x\rangle|F(x)\rangle$. But quantum circuits are linear, so if we send in a superposition[2] $(|x_1\rangle + |x_2\rangle)|0\rangle$, we get the superposition $|x_1\rangle|F(x_1)\rangle + |x_2\rangle|F(x_2)\rangle$ as output. If we send in a general superposition $(\sum_i |x_i\rangle)|0\rangle$, we get $\sum_i |x_i\rangle|F(x_i)\rangle$.

But this seems like we can do a large amount of computation in general! Unfortunately, there is a caveat. When we actually take a look at the answer, we only get $|x_i\rangle|F(x_i)\rangle$ for a single i. Worse, we have no control over which i we get an answer to. So while we are doing parallel computation, the results remain beyond our grasp for the most part.

Though we cannot in general use the output superposition of a quantum circuit directly, for some problems we can manipulate the superposition in order to reveal information that would classically require individually computing each term in the superposition. Factoring is one such problem.

2.3 Quantum Computation

Recall that if we know two numbers x and y such that $x^2 \equiv y^2 \pmod{N}$ and $x \not\equiv \pm y \pmod{N}$, we can factor N. If we set $y = 1$, we can compute such an x by choosing a random number a such that $\gcd(a, N) = 1$, and then finding an r such that $a^r \equiv 1 \pmod{N}$. (Such an r always exists, since $a^{\phi(N)} \equiv 1 \pmod{N}$ for all such a, so $r|\phi(N)$.) It turns out that for a random a, the probability that r is even and $a^{\frac{r}{2}} \not\equiv \pm 1 \pmod{N}$

[2]We will ignore all normalization factors for the purposes of this discussion.

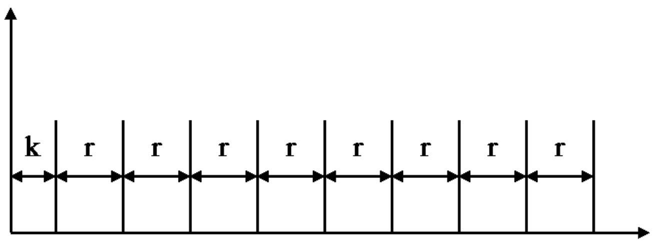

Figure 4: The result of measuring the output superposition after applying the exponentiation circuit.

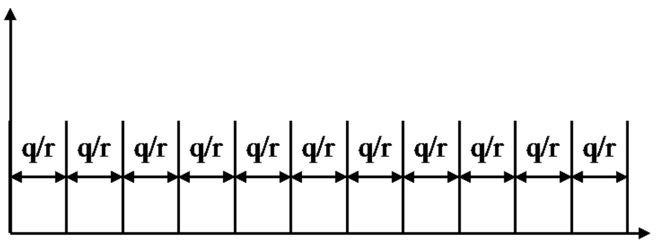

Figure 5: The result of applying the quantum Fourier transform to the superposition in figure 4.

is greater than $\frac{1}{2}$. Thus we only need to pick a a few times in order to guarantee that one choice will reveal a nontrivial square root of 1 $\pmod N$, i.e. a useable value for x.

Now consider the function $f(x) = a^x \pmod N$. This function is periodic with period r, since $a^r \equiv 1 \pmod N$. Thus factoring N is equivalent to finding the period of $f(x)$. Classically, the only way to find its period is to compute the $f(x)$ for r values of x. But r in general could be on order N^k for some k, so this would be an exponential process.

On a quantum computer, however, we can compute the period in polynomial time. We send in the superposition $(\sum_{i=0}^{N-1} |i\rangle)|0\rangle$ to the quantum circuit for $f(x)$, and receive the superposition $\sum_{i=0}^{N-1} |i\rangle|f(i)\rangle$ as output. Now we measure the piece of the output corresponding with $f(x)$, while leaving the piece for x alone. Recall that measuring the value of a superposition removes all elements that are inconsistent with the measured value. So if we measure some value $f(x) = y$, only the states $|i\rangle|y\rangle$ remain where $a^i \equiv y \pmod N$, as in figure 4. These states are multiples of r apart, and will be offset from 0 by some value k, depending on y.

Now just directly measuring one of the remaining states will not give us r, since we get some value $nr + k$. But since we don't know k or r, this appears as some random value to us. Instead, we apply an inverse *quantum discrete Fourier transform* modulo q, where q is some large multiple of r,[3] to the superposition in order to obtain a new superposition composed only of states that are multiples of $\frac{q}{r}$ for some large multiple q of r, as in figure 5. (I won't go into the details of this result. See *Quantum Computation and Quantum Information* by Nielsen and Chuang for more details.) Then we can measure in order to obtain $n\frac{q}{r}$, from which we can determine r by computing $\gcd(n\frac{q}{r}, q)$ to get $\frac{q}{r}$, and then dividing q by this.

Now let's analyze the running time of this algorithm. The exponentiation circuit $f(x)$ takes $O(\log^3 N)$ time, using fast exponentiation. The quantum Fourier transform takes $O(\log^2 N)$ time[4] (see Nielsen and Chuang for this result). Thus the total time for the quantum factoring algorithm is $O(\log^3 N)$.

2.4 Efficiency of Quantum Computation

Though the quantum factoring algorithm is exponentially faster than the fastest known classical algorithm, exponential speedups do not appear to be general characteristic of quantum computation. Only a handful of problems are known that are exponentially faster on a quantum computer, and most have very little practical

[3]We assume for simplicity that a large multiple q of r is known. It turns out that this procedure works even if we don't, but it is more difficult to analyze.

[4]Note this is an exponential speedup over the fastest known classical algorithm, which is of order $O(N \log N)$.

value. Factoring, for example, is only an important problem since it is thought to be hard. Once it ceases to be hard, it is no longer an important problem.

While the exact bound on the general speedup of quantum computation is not known, the current evidence indicates that the maximum speedup is quadratic. That is, if the fastest classical algorithm for solving a particular problem is in $\mathrm{O}(f(n))$, the fastest quantum algorithm is in general no faster than $\mathrm{O}(\sqrt{f(n)})$. Thus the problems for which quantum computation is exponentially faster are the exceptions, not the rule.

3 Further Information

CS 172 goes into great detail on the theory of computation. Its text, *Introduction to the Theory of Computation* by M. Sipser, is an excellent reference.

Again, CS/Chem/Physics 191 discusses quantum computation, including the factoring algorithm. However, it does not discuss quantum complexity. A hopefully understandable discussion of quantum complexity is in "Quantum Computability and Complexity and the Limits of Quantum Computation" by Benjamin, Huang, Kamil, and Kittiyachavalit, which is available at `www.eecs.berkeley.edu/~kamil`.