Theory of Computation

Unit 4-6: Turing Machines and Computability
Decidability and Encoding Turing Machines
Complexity and NP Completeness

Syedur Rahman
syedurrahman@gmail.com

Turing Machines

Q The set of finite states

Σ The finite set of input symbols

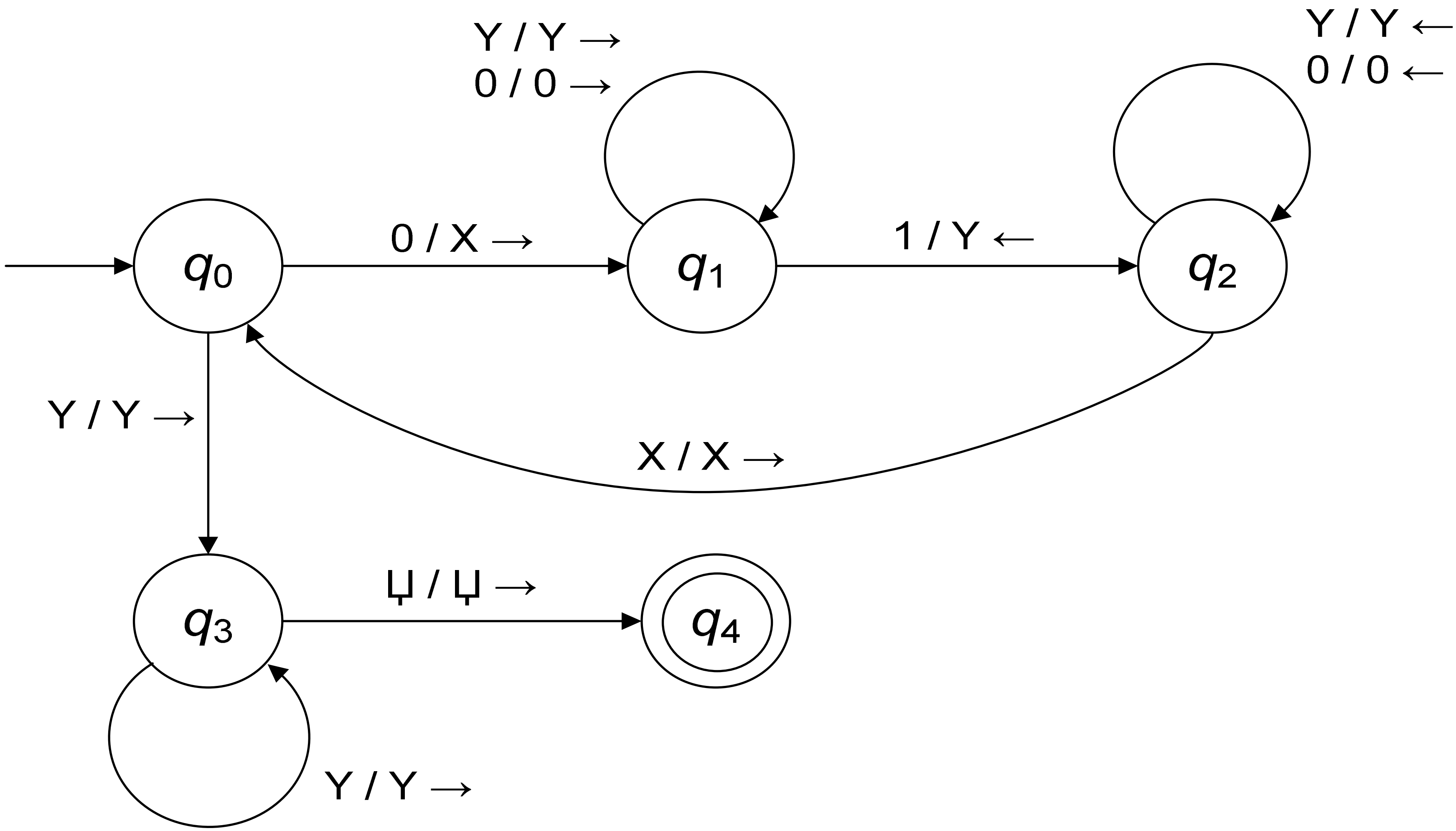
Γ The complete set of tape symbols. $\Sigma \subseteq \Gamma$

F A set of finite or accepting states. $F \subseteq Q$

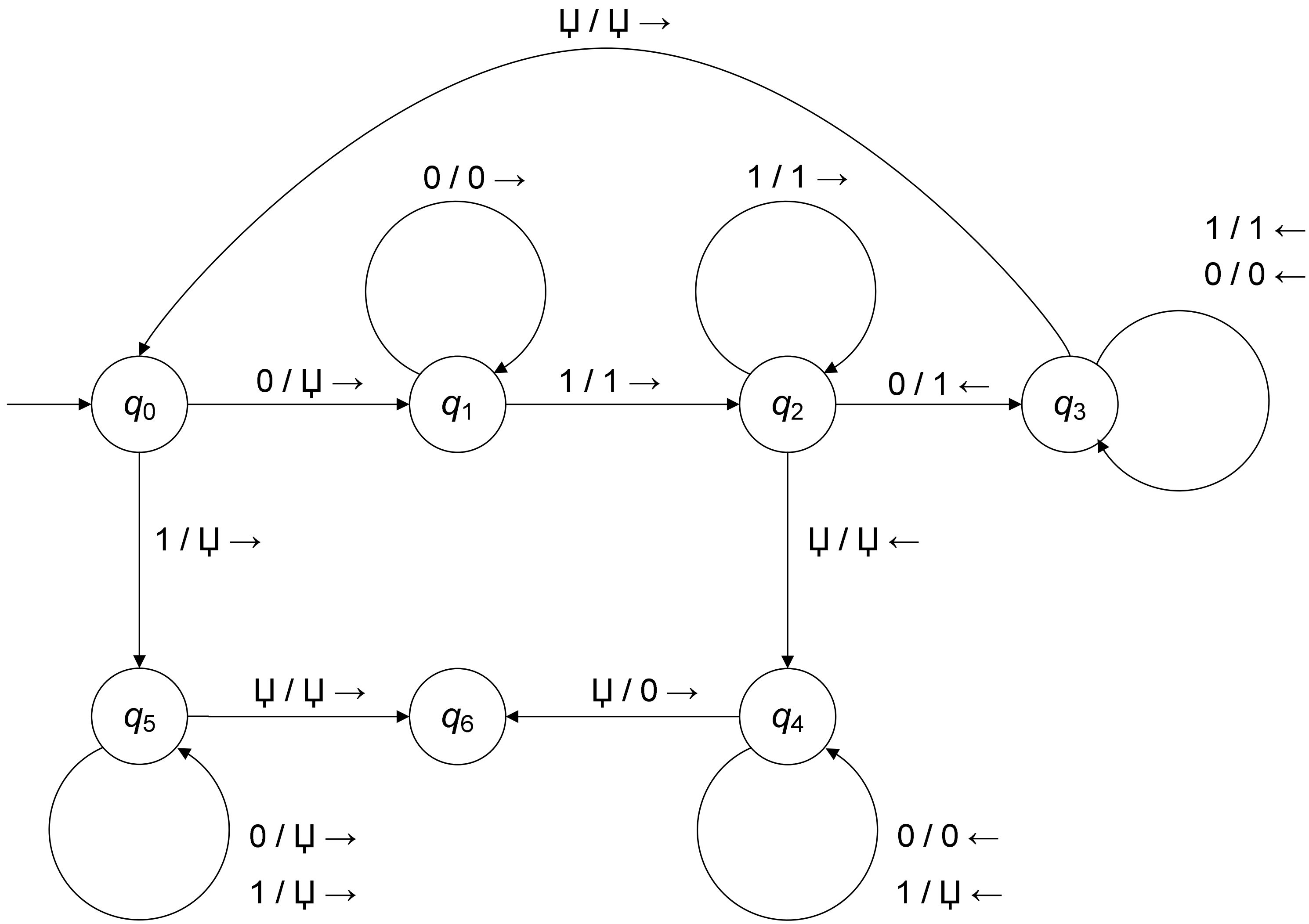
δ The transition function, where arguments (q, X) are the current state q and the tape symbol the head is on, X, such that $\delta(q, X)$ gives a triple (p, Y, D), the destination state p, the tape symbol Y that will replace X, and the direction the tape head moves afterwards, i.e. $\leftarrow$ or $\rightarrow$

B or ␣ The blank tape symbol.

The following Turing Machine M_1 accepts the language expressed as 0^n1^n where $n \in \mathbf{N} \wedge n>0$. The tape alphabet is {0, 1, X, Y, ␣) where ␣ is the blank symbol.



The given Turing Machine is designed to perform ***proper subtraction*** on two numbers m and n input on the tape as 0^m10^n. After the completion of its operation, it leaves $0^{\max(m - n, 0)}$ surrounded by blanks on the tape.



Further Reading

You should read up on the following topics which were covered in class:

- A run of a Turing Machine showing each configuration

i.e. (tape symbols on left of head, current state, rest of tape symbols)

- Non-deterministic Turing Machines (not examinable)
- Multi-tape Turing Machines (not examinable)

Encoding Turing Machines

We shall assume the states are $q_1, q_2, q_3, \ldots, q_k$ for some k, where the start state is q_1 and q_2 is the only accepting state.

We shall assume the tape symbols are $X_1, X_2 \ldots, X_m$ for some m. X_1 and X_2 will always be 0 and 1, whereas X_3 will be ⊔. Other symbols can be $X_4, X_5 \ldots$ etc.

We shall refer to directions $\leftarrow$ and $\rightarrow$ as D_1 and D_2 respectively.

A sample encoding

$M = (\{q1, q2, q3\}, \{0, 1\}, \{0, 1, ⊔\}, \delta, q1, ⊔, \{q2\})$

Where δ consists of the following rules:

$\delta(q1, 1) = (q3, 0, \rightarrow)$

$\delta(q3, 0) = (q1, 1, \rightarrow)$

$\delta(q3, 1) = (q2, 0, \rightarrow)$

$\delta(q3, ⊔) = (q3, 1, \leftarrow)$

A sample encoding

$M = (\{q1, q2, q3\}, \{0, 1\}, \{0, 1, \sqcup\}, \delta, q1, \sqcup, \{q2\})$

Where δ consists of the following rules:

	Binary Encoding:
$\delta(q1, 1) = (q3, 0, \rightarrow)$	0100100010100
$\delta(q3, 0) = (q1, 1, \rightarrow)$	0001010100100
$\delta(q3, 1) = (q2, 0, \rightarrow)$	00010010010100
$\delta(q3, \sqcup) = (q3, 1, \leftarrow)$	0001000100010010

Complete Encoding for Machine M

0100100010100110001010100100110001001001010011000100010001000010010

Decidability

Problems that can be solved are called decidable.

A language L is ***recursively enumerable*** (RE) if $L = L(M)$ for some TM M, i.e. there exists a Turing Machine that will halt and accept when a string that is a member of L is input into it. However, if strings that are not members of L are input into M, the machine may either halt in a non-accepting state or continue to run indefinitely. In any case it will not accept that particular string.

For a ***decidable or recursive language*** however the TM must not only halt and accept strings which are members of L but it also must halt and reject (by entering a non-accept state) string which are not members of L. Intuitively, all decidable languages are RE but not all RE languages are decidable.

RE but Undecidable: An Example

There are Recursively Enumerable languages that are undecidable. E.g. the ***Universal Language*** L_U consists of pairs (M, w) such that:

1. M is the binary coding of a Turing Machine
2. w is a string of 0's and 1's
3. M accepts input w

This problem is RE since one can construct the TM M and then run w on it, and M would halt if it is accepted. However we can not be sure what will happen is w is not a member of $L(M)$, so the language is not decidable.

Not RE: An Example

Some languages are not recursively enumerable at all. E.g.: The ***self-diagonalisation language*** L_d, is the set of strings w_i such that w_i is not in L(M_i). That is L_d consists of all strings *w* such that the TM *M* whose code is *w* does not accept when given *w* as input.

Lets say we constructed a TM M_i such that L(M_i) = L_d and the coding for M_i is w_i.

- If w_i is in L_d, then M_i accepts w_i. But w_i can not be a member of Ld if it is accepted by its machine M_i. So w_i can not be a member of L_d.
- If w_i is not a member of L_d, it will not be accepted by M_i. So, since w_i is not accepted by its own machine M_i, w_i must be a member of L_d.

As a result of this contradiction, we conclude such a Turing Machine M_i such that L(M_i) = L_d can never be constructed so the language L_d is not recursively enumerable.

Reduction

If we can convert instances of a problem *P1* into instances of another problem *P2* that have the same answer, then we say that *P1* reduces to *P2*.

We can therefore say *P2* is at least as hard as *P1*.

Theorem: If *P1* is reducible to *P2* then:

If *P1* is undecidable, then so is *P2*.

If *P1* is non-RE, then so is *P2*.

Please note this does not work the other way round!

Classes *P* and *NP*

A Turing Machine M is said to be of time complexity T(n) if whenever M is given an input of length n, M halts after making at most T(n) moves, regardless of whether this leads to an accept state. T(n) can be any function. E.g. $T(n) = 10n$ or $T(n) = n^2+10$ or $T(n) = 2^n+10n$

We say a language L is in class P if a deterministic TM M exists such that $L(M) = L$ and its time complexity T(n) is a polynomial.

We say a language L is in class NP if a non-deterministic TM M exists such that $L(M) = L$ and its time complexity T(n) is a polynomial.

Needless to say, all P problems are NP problems as well.

Quick Revision of Complexity

Big-O (upper bound): Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that: $|f(x)| \leq C|g(x)|$ whenever $x>k$. We say that $f(x)$ grows no faster than $g(x)$.

E.g. $x^2 + x + 1$ is $O(x^2)$, can be proved if C=3 and k=2

$7x^2$ is $O(x^3)$, can be proved if C=1 and k=7

x^2 is $O(x^2 + x + 1)$, can be proved if C=1 and k=1

Big-Omega (lower bound): Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are constants C and k such that: $|f(x)| \geq |g(x)|$ whenever $x>k$. Generally if $f(x)$ is $\Omega(g(x))$ then $g(x)$ is $O(f(x))$. We say that $f(x)$ grows no faster than $g(x)$.

Big-Theta (upper and lower bound): Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$, if and only if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. We say that $f(x)$ is of order $g(x)$.

Commonly Used Terminology for Complexity of Algorithms

Complexity	Terminology
$O(1)$	Constant Complexity
$O(\log n)$	Logarithmic Complexity
$O(n)$	Linear Complexity
$O(n \log n)$	$n \log n$ Complexity
$O(n^k)$	Polynomial Complexity
$O(k^n)$, where $k>1$	Exponential Complexity
$O(n!)$	Factorial Complexity

This list is in order of increasing complexity.
n is a variable whereas *k* is a constant.

P, NP and Reductions

Let us assume *P1* is reducible to *P2*.

Therefore, in order for an instance of *P1* to be reduced to instance(s) of *P2*, a construction algorithm *C* must perform some conversion. If *P2* is in class *P* then Is *P1* in *P*?

P, NP and Reductions

Let us assume *P1* is reducible to *P2*.

Therefore, in order for an instance of *P1* to be reduced to instance(s) of *P2*, a construction algorithm *C* must perform some conversion. If *P2* is in class *P* then Is *P1* in *P*?

Answer: Only if the construction algorithm *C* is of polynomial time complexity.

What if *C* is of exponential time complexity?

If *P2* is $O(n^k)$ and C is $O(e^n)$, then *P1* is $O(n^k+e^n)$ i.e. $O(e^n)$, i.e. *P1* is not in class *P*

NP Complete and NP Hard

A language problem L is ***NP-Complete*** if:

- L is in NP
- For every language L' in NP, there is a polynomial-time reduction of L' to L.

$$NP\text{-}Complete \subseteq NP - P$$

If L is not in NP, we call it ***NP-Hard***. It is generally acceptable to use "intractable" to mean NP-Hard and it is enough to show that L can only be solved in exponential time or worse, to prove it is NP-Hard.