Carnegie Mellon University in Qatar
The most general model of computation
Computations of a TM are described by a sequence of configurations.
  - Accepting Configuration
  - Rejecting Configuration
Turing-recognizable languages
  - TM halts in an accepting configuration if $w$ is in the language.
  - TM may halt in a rejecting configuration or go on indefinitely if $w$ is not in the language.
Turing-decidable languages
  - TM halts in an accepting configuration if $w$ is in the language.
  - TM halts in a rejecting configuration if $w$ is not in the language.
Example TM-2

- A Turing machine that decides $A = \{0^{2^n} \mid n \geq 0\}$
- $M =$ “On Input string $w$
  1. Sweep left-to-right across the tape, crossing off every other 0.
  2. If in 1) that tape has one 0 left, accept (Why?)
  3. If in 1) tape has more than one 0, and the number of 0’s is odd, reject. (Why?)
  4. Return the head to the left end of the tape.
  5. Go to 1)”

- Basically every sweep cuts the number of 0’s by two.
- At the end only 1 should remain and if so the original number of zeroes was a power of 2.’
Configurations for input 0000.

1. $q_1 0000 \; \square$
2. $\square q_2 000 \; \square$
3. $\square xq_3 00 \; \square$
4. $\square x0q_4 0 \; \square$
5. $\square x0xq_3 \; \square$
6. $\square x0q_5 x \; \square$
7. $\square xq_5 0x \; \square$
8. $\square q_5 x0x \; \square$
9. $q_5 \; \square x0x \; \square$
10. $\square q_2 0x \; \square$
11. $\square xq_2 0x \; \square$
12. $\square xq_3 0x \; \square$
13. $\square xxq_3 x \; \square$
14. $\square xxq_5 x \; \square$
15. $\square xq_5 xx \; \square$
16. $\square q_5 xxx \; \square$
17. $q_5 \; \square xxx \; \square$
18. $\square q_2 xxx \; \square$
19. $\square xq_2 xx \; \square$
20. $\square xxq_2 x \; \square$
21. $\square xxxq_2 \; \square$
22. $\square xxx \; \square q_{accept}$
A TM to add 1 to a binary number (with a 0 in front)

\[ M = \text{“On input } w \text{”} \]

1. Go to the right end of the input string
2. Move left as long as a 1 is seen, changing it to a 0.
3. Change the 0 to a 1, and halt.

For example, to add 1 to \( w = 0110011 \)

- Change all the ending 1’s to 0’s \( \Rightarrow \) \( 0110000 \)
- Change the next 0 to a 1 \( \Rightarrow \) \( 0110100 \)

Now let’s design a TM for this problem.
We defined the basic Turing Machine

- Single tape (infinite in one direction)
- Deterministic state transitions

We could have defined many other variants:

- Ordinary TMs which need not move after every move.
- Multiple tapes – each with its own independent head
- Nondeterministic state transitions
- Single tape infinite in both directions
- Multiple tapes but with a single head
- Multidimensional tape (move up/down/left/right)
A computational model is robust if the class of languages it accepts does not change under variants.
- We have seen that DFA’s are robust for nondeterminism.
- But not PDAs!

The robustness of Turing Machines is by far greater than the robustness of DFAs and PDAs.

We introduce several variants on Turing machines and show that all these variants have equal computational power.

When we prove that a TM exists with some properties, we do not deal with questions like
- How large is the TM? or
- How complex is it to “program” that TM?

At this point we only seek existential proofs.
Suppose in addition moving Left or Right, we give the option to the TM to stay (S) on the current cell, that is:

\[ \delta : Q \times \Gamma = Q \times \Gamma \times \{L, R, S\} \]

Such a TM can easily simulate an ordinary TM: just do not use the S option in any move.

An ordinary TM can easily simulate a TM with the stay option.

For each transition with the S option, introduce a new state, and two transitions

- One transition moves the head right, and transits to the new state.
- The next transition moves the head back to left, and transits to the previous state.
Multitape Turing Machines

Finite control of $M$

Tape 1: 1 1 __ __ __ __ __

Tape 2: a b __ __ __ __ __

Tape 3: u v __ __ __ __ __
A multitape Turing Machine is like an ordinary TM
- There are $k$ tapes
- Each tape has its own independent read/write head.

The only fundamental difference from the ordinary TM is $\delta$ – the state transition function.

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

The $\delta$ entry $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, L, \ldots L)$ reads as:
- If the TM is in state $q_i$ and
- the heads are reading symbols $a_1$ through $a_k$,
- Then the machine goes to state $q_j$, and
- the heads write symbols $b_1$ through $b_k$, and
- Move in the specified directions.
Simulating a Multitape TM with an Ordinary TM

Finite control of $M$

Finite control of $S$

Tape1

Tape2

Tape3
Simulating a Multitape TM with an Ordinary TM

- We use # as a delimiter to separate out the different tape contents.
- To keep track of the location of heads, we use additional symbols
  - Each symbol in $\Gamma$ has a “dotted” version.
  - A dotted symbol indicates that the head is on that symbol.
  - Between any two #’s there is only one symbol that is dotted.
- Thus we have 1 real tape with $k$ “virtual” tapes, and
- 1 real read/write head with $k$ “virtual” heads.
Simulating a Multitape TM with an Ordinary TM

- Given input $w = w_1 \cdots w_n$, $S$ puts its tape into the format that represents all $k$ tapes of $M$

  $\# \cdot w_1 w_2 \cdots w_n \# \mathop{\sqcup} \# \mathop{\sqcup} \# \cdots \#$

- To simulate a single move of $M$, $S$ starts at the leftmost $\#$ and scans the tape to the rightmost $\#$.
  - It determines the symbols under the “virtual” heads.
  - This is remembered in the finite state control of $S$. (How many states are needed?)

- $S$ makes a second pass to update the tapes according to $M$.
  - If one of the virtual heads, moves right to a $\#$, the rest of tape to the right is shifted to “open up” space for that “virtual tape”. If it moves left to a $\#$, it just moves right again.
Simulating a Multitape TM with an Ordinary TM

Thus from now on, whenever needed or convenient we will use multiple tapes in our constructions.

You can assume that these can always be converted to a single tape standard TM.
We defined the state transition of the ordinary TM as
\[ \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} \]

A nondeterministic TM would proceed computation with multiple next configurations. \( \delta \) for a nondeterministic TM would be
\[ \tilde{\delta} : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}) \]

(\( \mathcal{P}(S) \) is the power set of S.)

This definition is analogous to NFAs and PDAs.
A computation of a Nondeterministic TM is a tree, where each branch of the tree is looks like a computation of an ordinary TM.
If a single branch reaches the accepting state, the Nondeterministic TM accepts, even if other branches reach the rejecting state.

What is the power of Nondeterministic TMs?

Is there a language that a Nondeterministic TM can accept but no deterministic TM can accept?
Theorem

Every nondeterministic Turing machine has an equivalent deterministic Turing Machine.

Proof Idea

- Timeshare a deterministic TM to different branches of the nondeterministic computation!
- Try out all branches of the nondeterministic computation until an accepting configuration is reached on one branch.
- Otherwise the TM goes on forever.
Deterministic TM $D$ simulates the Nondeterministic TM $N$.

Some of branches of the $N$’s computations may be infinite, hence its computation tree has some infinite branches.

If $D$ starts its simulation by following an infinite branch, $D$ may loop forever even though $N$’s computation may have a different branch on which it accepts.

This is a very similar problem to processor scheduling in operating systems.

- If you give the CPU to a (buggy) process in an infinite loop, other processes “starve”.

In order to avoid this unwanted situation, we want $D$ to execute all of $N$’s computations concurrently.
Nondeterministic Computation

Configurations of the nondeterministic computation

Initial Configuration

Nondeterministic choices available from C4

C1

$q_0 w_1 w_2 \ldots w_n$
Nondeterministic Computation

Configurations of the nondeterministic computation

Initial Configuration

Nondeterministic choices available from C4

A rejecting branch
Nondeterministic Computation

Configurations of the nondeterministic computation

Initial Configuration

q_0 w_1 w_2 \ldots w_n

Nondeterministic choices available from C4

Accepting Configuration

u q_{accept} v

An accepting branch
Nondeterministic Computation

Configurations of the nondeterministic computation

Initial Configuration

q_0 w_1 w_2 ... w_n

Nondeterministic choices available from C4

A nonterminating branch
Simulating Nondeterministic Computation

![Diagram of a nondeterministic computation process with states and transitions labeled from 1 to 11 and an initial configuration marked.]

- Initial Configuration: $q_0 w_1 w_2 \ldots w_n$
- Order of simulation: States transition through the computation process.
**Simulating Nondeterministic Computation**

During simulation, $D$ processes the configurations of $N$ in a breadth-first fashion.

Thus $D$ needs to maintain a queue of $N$’s configurations (Remember queues?)

- $D$ gets the next configuration from the head of the queue.
- $D$ creates copies of this configuration (as many as needed)
- On each copy, $D$ simulates one of the nondeterministic moves of $N$.
- $D$ places the resulting configurations to the back of the queue.
Structure of the Simulating DTM

- $N$ is simulated with 2-tape DTM, $D$
  - Note that this is different from the construction in the book!
Built into the finite control of $D$ is the knowledge of what choices of moves $N$ has for each state and input.
**How \( D \) Simulates \( N \)**

1. \( D \) examines the state and the input symbol of the current configuration (right after the dotted separator).
2. If the state of the current configuration is the accept state of \( N \), then \( D \) accepts the input and stops simulating \( N \).
HOW \( D \) SIMULATES \( N \)

1. \( D \) copies \( k \) copies of the current configuration to the scratch tape.

2. \( D \) then applies one nondeterministic move of \( N \) to each copy.
**How \( D \) Simulates \( N \)**

D then copies the new configurations from the scratch tape, back to the end of tape 1 (so they go to the back of the queue), and then clears the scratch tape.

D then returns to the marked current configuration, and “erases” the mark, and “marks” the next configuration.

D returns to step 1), if there is a next configuration. Otherwise rejects.
Let $m$ be the maximum number of choices $N$ has for any of its states.

Then, after $n$ steps, $N$ can reach at most 

$$1 + m + m^2 + \cdots + m^n$$

configurations (which is at most $nm^n$)

Thus $D$ has to process at most this many configurations to simulate $n$ steps of $N$.

Thus the simulation can take exponentially more time than the nondeterministic TM.

It is not known whether or not this exponential slowdown is necessary.
Corollary

A language is Turing-recognizable if and only if some nondeterministic TM recognizes it.

Corollary

A language is decidable if and only if some nondeterministic TM decides it.