Turing Machines

Sections 17.3 – 17.5

Turing Machine Extensions

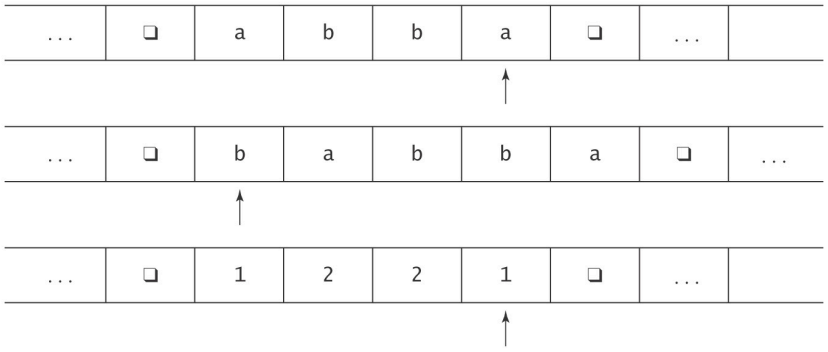
There are many extensions we might like to make to our basic Turing machine model. But:

We can show that every extended machine has the same power as the basic machine.

Some possible extensions:

- Multiple tape TMs
- Nondeterministic TMs

Multiple Tapes



Multiple Tapes

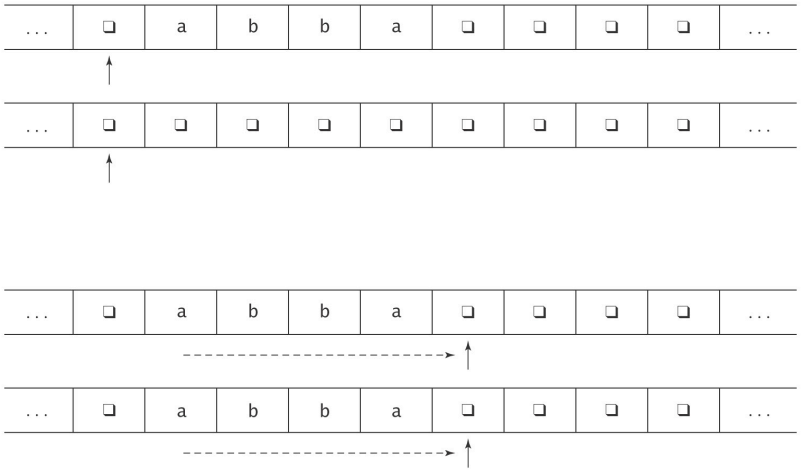
The transition function for a *k*-tape Turing machine:

$((K\text{-}H), \Gamma_1, \Gamma_2, \ldots, \Gamma_k)$ to $(K, \Gamma_{1'}, \{\leftarrow, \rightarrow, \uparrow\}, \Gamma_{2'}, \{\leftarrow, \rightarrow, \uparrow\}, \ldots, \Gamma_{k'}, \{\leftarrow, \rightarrow, \uparrow\})$

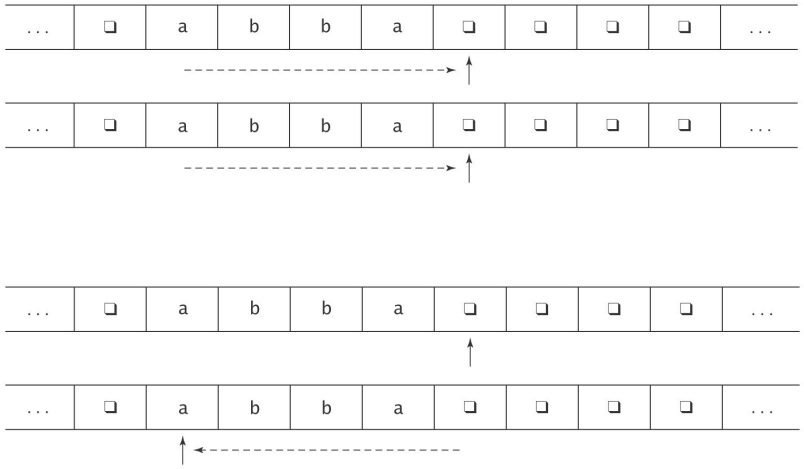
Input: as before on tape 1, others blank.
Output: as before on tape 1, others ignored.

Note: tape head is allowed to stay put.

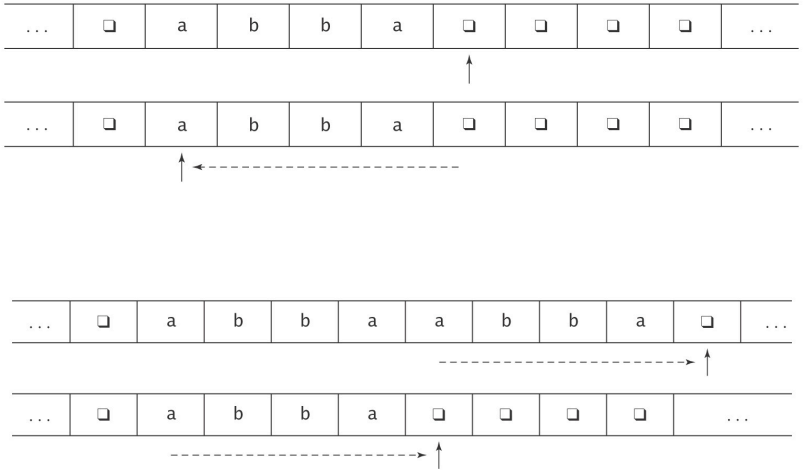
Example: Copying a String



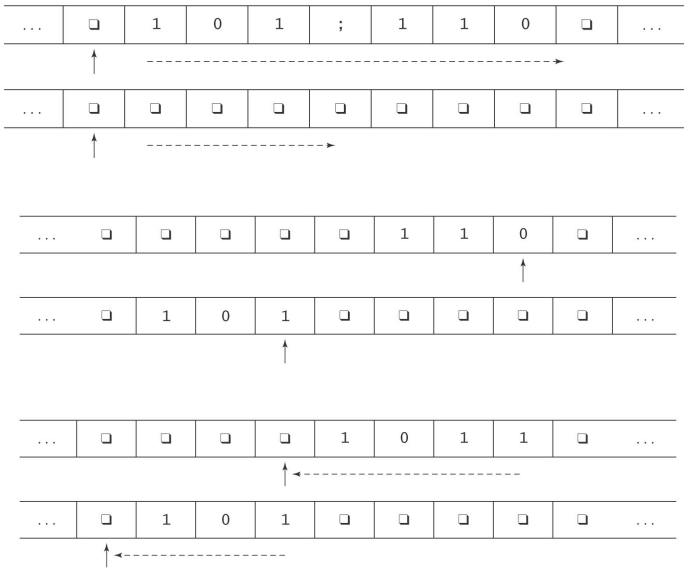
Example: Copying a String



Example: Copying a String



Another Two Tape Example: Addition



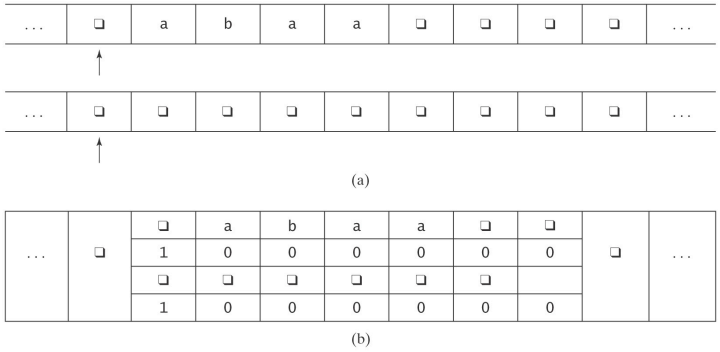
Adding Tapes Adds No Power

Theorem: Let M be a k-tape Turing machine for some $k \geq 1$. Then there is a standard TM M' where $\Sigma \subseteq \Sigma'$, and:

- On input x, M halts with output z on the first tape iff M' halts in the same state with z on its tape.
- On input x, if M halts in n steps, M' halts in $\mathcal{O}(n^2)$ steps.

Proof: By construction.

The Representation



Alphabet (Σ') of M' = $\Gamma \cup (\Gamma \times \{0, 1\})^k$:

❑, a, b, (❑, 1, ❑, 1), (a, 0, ❑ ,0), (b, 0, ❑, 0), …

The Operation of *M'*

…	❑	❑	a	b	a	a	❑	❑	❑	…
		1	0	0	0	0	0	0		
		❑	❑	❑	❑	❑	❑			
		1	0	0	0	0	0	0		

1. Set up the multitrack tape.
2. Simulate the computation of M until (if) M would halt:
   - 2.1 Scan left and store in the state the k-tuple of characters under the read heads. Move back right.
   - 2.2 Scan left and update each track as required by the transitions of M. If necessary, subdivide a new square into tracks. Move back right.
3. When M would halt, reformat the tape to throw away all but track 1, position the head correctly, then go to M' s halt state.

How Many Steps Does *M'* Take?

Let: w be the input string, and
n be the number of steps it takes M to execute.

Step 1 (initialization): $\mathcal{O}(|w|)$.

Step 2 (computation):
Number of passes = n.
Work at each pass: 2.1 = 2 · (length of tape).
= 2 · ($|w| + n$).
2.2 = 2 · ($|w| + n$).
Total: $\mathcal{O}(n \cdot (|w| + n))$.

Step 3 (clean up): $\mathcal{O}$(length of tape).

Total: $\mathcal{O}(n \cdot (|w| + n))$.
$= \mathcal{O}(n^2)$. *

* assuming that $n \geq w$

Impact of Nondeterminism

- FSMs
  - Power NO
- PDAs
  - Power YES
- Turing machines
  - Power NO

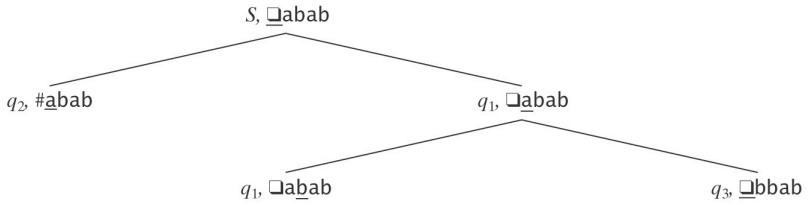
Nondeterministic Turing Machines

A **nondeterministic** TM is a sixtuple $(K, \Sigma, \Gamma, \Delta, s, H)$.

Δ is a ***subset*** of:

$((K - H) \times \Gamma) \times (K \times \Gamma \times \{\leftarrow, \rightarrow\})$

Nondeterministic Turing Machines



What does it mean for a nondeterministic Turing machine to:

- Decide a language
- Semidecide a language
- Compute a function

Nondeterministic Deciding

Let $M = (K, \Sigma, \Gamma, \Delta, s, \{y, n\})$ be a nondeterministic TM.

Let w be an element of Σ^*.

M ***accepts*** w iff *at least one* of its computations accepts.

M ***rejects*** w iff all of its computations reject.

M ***decides*** a language $L \subseteq \Sigma^*$ iff, $\forall w$:

- There is a finite number of paths that M can follow on input w,
- All of those paths halt, and
- $w \in L$ iff M accepts w.

An Example of Nondeterministic Deciding

$L = \{w \in \{0, 1\}^* : w$ is the binary encoding of a composite number$\}$.

M decides L by doing the following on input w:

1. Nondeterministically choose two positive binary numbers such that:

   $2 \leq |p|$ and $|q| \leq |w|$.

   Write them on the tape, after w, separated by ;

   ❑110011;111;1111❑❑

2. Multiply p and q and put the answer, A, on the tape, in place of p and q.

   ❑110011;1011111❑❑

3. Compare A and w. If equal, go to y. Else go to n.

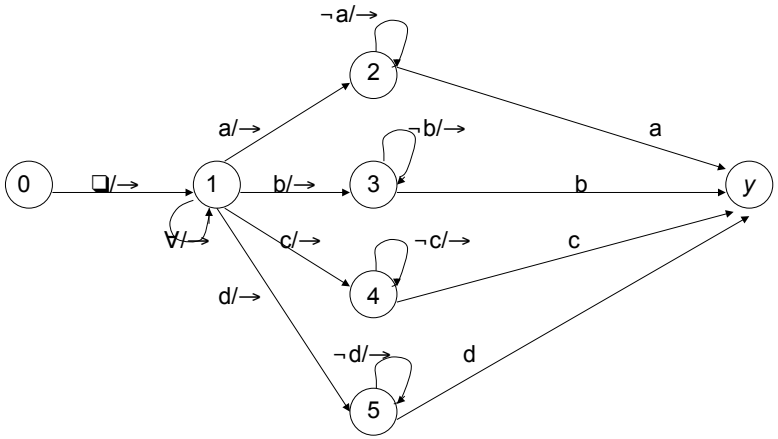
Nondeterministic Semideciding

Let $M = (K, \Sigma, \Gamma, \Delta, s, H)$ be a nondeterministic TM.

We say that M ***semidecides*** a language $L \subseteq \Sigma^*$ iff:
for all $w \in \Sigma^*$:
$w \in L$ iff
(s, ❑w) yields a least one accepting configuration.

An Example

$L = \{w \in \{a, b, c, d\}^* :$ there are two of at least one letter$\}$



But, in this case, we can do better.

A Harder Example

Let L = {descriptions of TMs that halt on at least one string}.

Let <M> mean the string that describes some TM M.

S semidecides L as follows on input <M>:

1. Nondeterministically choose a string w in Σ_M^* and write it on the tape.
2. Run M on w.

We'll prove later that, in this case, semideciding is the best we can do.

Nondeterministic Function Computation

M ***computes*** a function *f* iff, $\forall w \in \Sigma^*$:

- All of *M*' s computations halt, and
- All of *M*' s computations result in *f*(*w*).

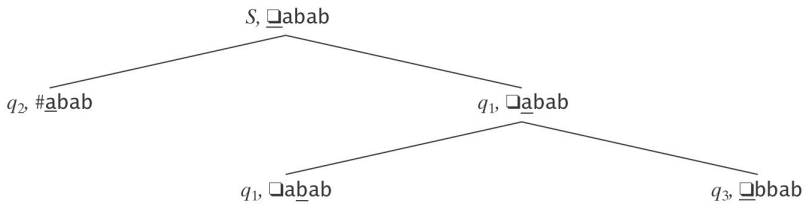
Equivalence of Deterministic and Nondeterministic Turing Machines

Theorem: If a nondeterministic TM *M* decides or semidecides a language, or computes a function, then there is a standard (deterministic) TM *M'* deciding or semideciding the same language or computing the same function.

Proof: (by construction). We must do separate constructions for deciding/semideciding and for function computation.

For Deciding/Semideciding

Build *M'*, which tries the possible computations of *M*. If one of them accepts, *M'* accepts.

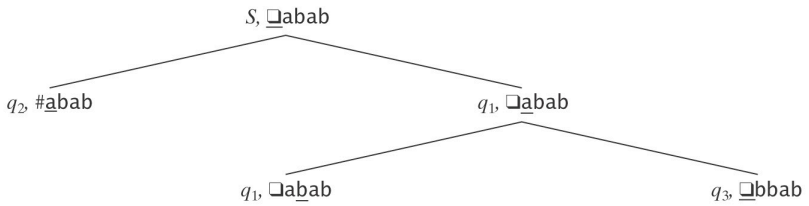


For Deciding/Semideciding

Recall the way we did this for FSMs: simulate being in a combination of states.

Will this work here?

For Deciding/Semideciding

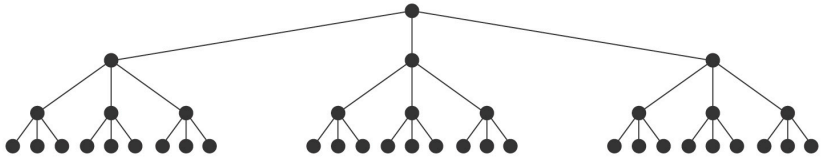


What about: Try path 1. If it accepts, accept. Else
Try path 2. If it accepts, accept. Else
•
•

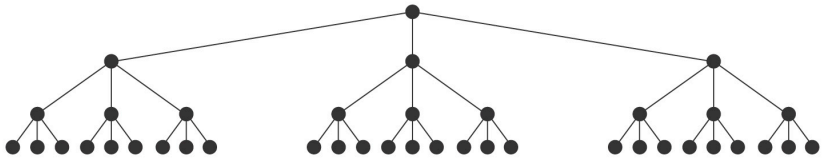
Iterative Deepening

Iterative deepening:

1. $d = 1$ /* set the initial depth limit to 1 */
2. Loop until a solution is found
   - 2.1 Start at the root node; explore all paths of depth d, depth first.
   - 2.2 If a solution is found, exit.
   - 2.3 $d = d + 1$.

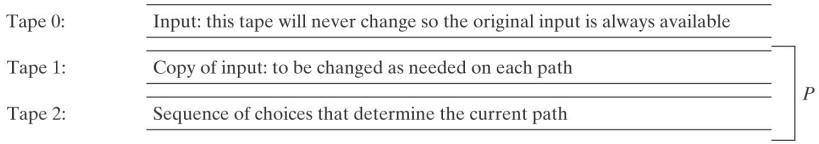


Restart Iterative Deepening



Start over at the root each time.

M′ Uses Three Tapes



If *M* is a Deciding Machine

M′ should halt as soon as one of the following things happens:

- It discovers a path along which *M* halts and accepts. In this case, *M′* accepts.
- It has tried all paths until they halt, but all have rejected. In this case, *M′* rejects.

Simulating a Real Computer

- An unbounded number of memory cells addressed by the integers starting at 0.
- An instruction set composed of basic operations including load, store, add, subtract, jump, conditional jump, and halt. Here's a simple example program:

R	10	read 2 bits from tape into accumulator
MIR	10	move input pointer 2 bits right
CJUMP	1001	conditional jump; if acc = 0, go to 1001
A	10111	add location 10111 value to accumulator
ST	10111	store result at location 10111

- A program counter.
- An address register.
- An accumulator.
- A small fixed number of special purpose registers.
- An input file.
- An output file.

Simulating a Real Computer

Theorem: A random-access, stored program computer can be simulated by a Turing Machine.

Proof: By construction.

simcomputer will use 7 tapes:

- Tape 1: the computer's memory.
- Tape 2: the program counter.
- Tape 3: the address register.
- Tape 4: the accumulator.
- Tape 5: the op code of the current instruction.
- Tape 6: the input file.
- Tape 7: the output file, initially blank.

Representing Memory

Memory will be organized as a series of (address, value) pairs, separated by delimiters:

#0,val_0#1,val_1#10,val_2#11,val_3#100,val_4# …#

Instructions: four bit operation code followed by address. So our example program could look like:

#0,000110010#1,11111001#10,001110011#
11,001010111#....

Must delimit words because no bound on their length:

- Addresses may get longer as the simulated program uses more words of its memory.
- Numeric values may increase as old values are added to produce new ones.

Simcomputer

simcomputer(*program*) =

1. Move the input string to tape 6.
2. Initialize the program counter (tape 2) to 0.
3. Loop:
   - 3.1 Starting at the left of the nonblank portion of tape 1, scan right looking for an index that matches the contents of tape 2 (the program counter).

   /* Decode the current instruction and increment the program counter.
   - 3.2 Copy the operation code to tape 5.
   - 3.3 Copy the address to tape 3.
   - 3.4 Add 1 to the value on tape 2.

   /* Retrieve the operand.
   - 3.5 Starting at the left again, scan right looking for the address stored on tape 3.

   /* Execute the instruction.
   - 3.6 If the operation is Load, copy the operand to tape 4.
   - 3.7 If the operation is Add, add the operand to the value on tape 4.
   - 3.8 If the operation is Jump, copy the value on tape 3 to tape 2
   - 3.9 And so forth for the other operations.

Alternative TM Definitions

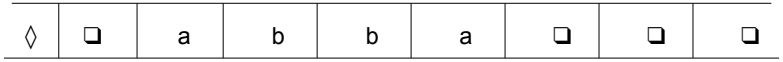
An alternative definition of a Turing machine: one-way tape

◊ is a wall. The TM cannot move to the left past ◊.

So δ is constrained as follows:

(a) if the input symbol is ◊, the action is →, and

(b) ◊ can never be written.



Does This Difference Matter?

Remember the goal:

Define a device that is:

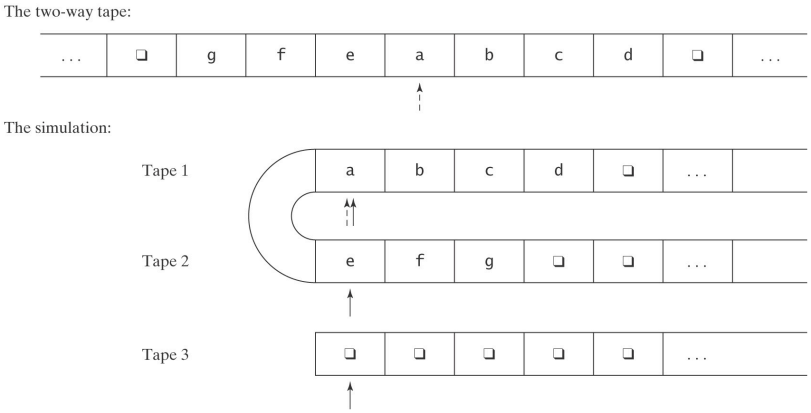
- powerful enough to describe all computable things,
- simple enough that we can reason formally about it

Both definitions are simple enough to work with, although details may make specific arguments easier or harder.

But, do they differ in their power?

Answer: No.

The Simulation



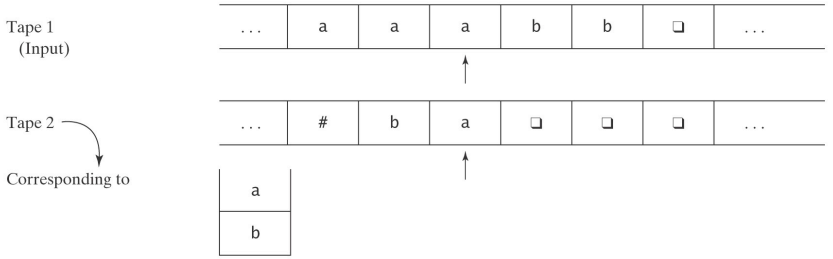
Stacks vs. a Tape

- Did we lose anything by giving up the PDA's stack in favor of the TM's tape?
- Could we have gotten the power of a TM's tape just with stacks?

Simulating a PDA

The components of a PDA:

- Finite state controller
- Input stream
- Stack



Simulating a Turing Machine with a PDA with Two Stacks