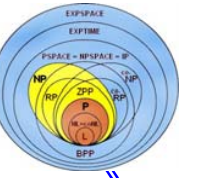
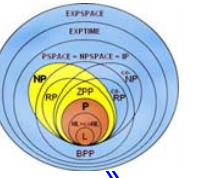
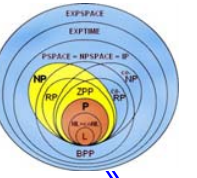


Session 13

- Constructing Turing Machines
- Other Models of Turing Machines

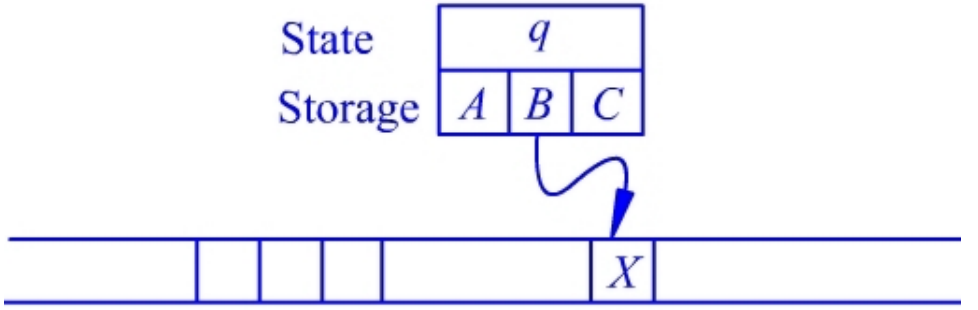


Constructing Turing Machines

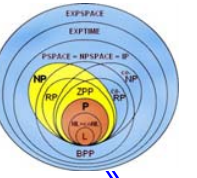


Storage in the State

We can use the finite control not only to represent a position of the Turing machine, but to hold a finite amount of data.



This "trick" does not extend the basic model of the TM. It's only a convenience.

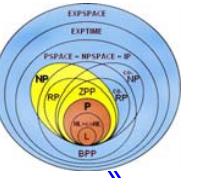


Example Design a TM accepting the language $L(\mathbf{01}^* + \mathbf{10}^*)$.

We construct a TM M, and think of its finite control consisting of not only a state, but a data element.

The Turing machine M remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. Thus, M will accept the language $L(\mathbf{01}^* + \mathbf{10}^*)$.

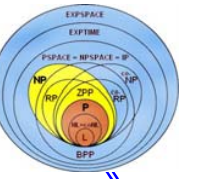
Let $M = (\{q[0], q[1], q[B], p[0], p[1], p[B]\}, \{0, 1\}, \{0, 1, B\}, \delta, q[B], \{p[B]\})$.



The set of states may be thought of as pairs with two components: $\{q, p\} \times \{0, 1, B\}$.

- A control portion, q and p, that remembers what the TM is doing. State q indicates that M has not yet read its first symbol. State p indicates that M has read the symbol, and is checking that it does not appear elsewhere.
- A data portion, which remembers the first symbol seen, 0 or 1. The symbol B in this component means that no symbol has been read.

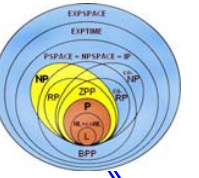
The states $q[0], q[1]$ are useless, $q[B]$ is start state, and $p[B]$ is final state.



The transition function δ of M is as follows:

1. $\delta(q[B], a) = (p[a], a, R)$ for $a = 0$ or $a = 1$.
2. $\delta(p[a], \overline{a}) = (p[a], \overline{a}, R)$ where $\overline{a}$ is the "complement" of a. That is, 0 if $a = 1$ and 1 if $a = 0$.
3. $\delta(p[a], B) = (p[B], B, R)$ for $a = 0$ or $a = 1$.

Notice that M has no definition for $\delta(p[a], a)$ for $a = 0$ and $a = 1$. It halts then.



Example Design a TM M adding substring 101 as suffix of its input strings.

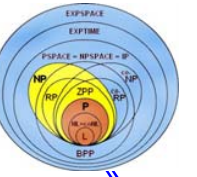
In general, for given x, we can use $q[x]$ indicates state on which M is searching the tail of the input strings, and define $\delta(q[ay], b) = (q[ay], b, R)$ and $\delta(q[ay], B) = (q[y], a, R)$ for our goal. The TM M (as a *transducers*) we shall design is:

$$M = (\{q[101], q[01], q[1], q[\epsilon]\}, \{0, 1\}, \{0, 1, B\}, \delta, q[101], B, \{q[\epsilon]\})$$

where

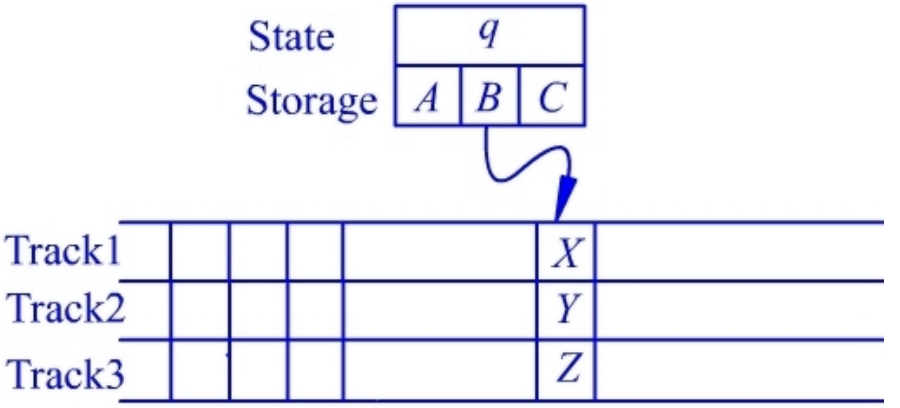
$$\delta(q[101], 0) = (q[101], 0, R), \quad \delta(q[101], 1) = (q[101], 1, R)$$

$$\delta(q[101], B) = (q[01], 1, R), \quad \delta(q[01], B) = (q[1], 0, R), \quad \delta(q[1], B) = (q[\epsilon], 1, R)$$

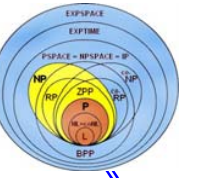


Multiple Tracks

Another useful trick is to think of the tape of a Turing machine as composed of several tracks.



Using multiple tracks does not extend what the Turing machine can do.



A common use of multiple tracks is to treat one track as holding the data and a second track as holding a mark.

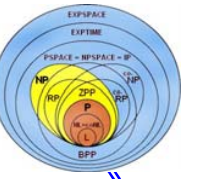
Example Design a TM explicitly to recognize the context-free language

$$L_{wcw} = \{wcw \mid w \in \{0,1\}^+\}.$$

The Turing machine we shall design is:

$$M = (Q, \Sigma, \Gamma, \delta, q_1[B], [B,B], \{q_9[B]\}).$$

We shall give a detail description for each component.

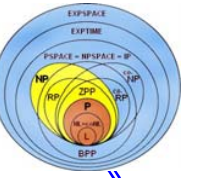


Q : The set of states is $\{q_1, \cdots, q_9\} \times \{0, 1, B\}$. We again use the technique of storage in the finite control.

Γ : The set of tape symbols is $\{B, *\} \times \{0, 1, c, B\}$. The first track can be either blank or mark represented by $*$. The second track is what we think of as the tape symbol itself.

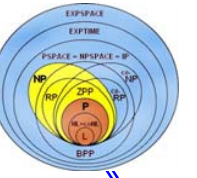
That is, we may think of the symbol $[B, X]$ as if it were the tape symbol X, for $X = 0, 1, c, B$. The initial blank symbol on the tape is $[B, B]$.

Σ : The input symbols are $[B, 0]$ and $[B, 1]$, which we identify with 0 and 1, respectively.



δ : Defined by following rules, in which a and b each stands for either 0 or 1.

1. $\delta(q_1[B], [B, a]) = (q_2[a], [*, a], R)$. M picks up a, stores it in its control, goes to state q_2, and marks a.
2. $\delta(q_2[a], [B, b]) = (q_2[a], [B, b], R)$. M moves right, looking for marker c.
3. $\delta(q_2[a], [B, c]) = (q_3[a], [B, c], R)$. When M finds c, it continues to move right, but change state q_3.
4. $\delta(q_3[a], [*, b]) = (q_3[a], [*, b], R)$. M continues past all checked symbols.



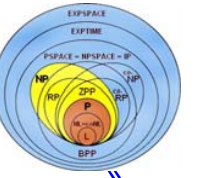
5. $\delta(q_3[a], [B, a]) = (q_4[B], [*, a], L)$. M finds the first matched symbol, goes state q_4. Dropping the symbol from control, M starts moving left.

6. $\delta(q_4[B], [*, b]) = (q_4[B], [*, b], L)$. M moves left over checked symbols.

7. $\delta(q_4[B], [B, c]) = (q_5[B], [B, c], L)$. When M encounters c, it switches to state q_5 and continues left.

In state q_5, M must make a decision, depending on whether or not the symbol immediately to the left of c is checked or unchecked.

We now have two branches.

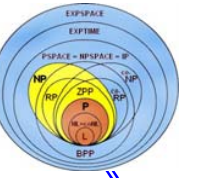


The first branch.

8. $\delta(q_5[B],[B,a]) = (q_6[B],[B,a],L)$. The symbol to the left of c is unchecked. M goes to state q_6 and continues left, looking for a checked symbol.

9. $\delta(q_6[B],[B,b]) = (q_6[B],[B,b],L)$. As long as symbols are unchecked, M remains in q_6 and proceeds left.

10. $\delta(q_6[B],[*,a]) = (q_1[B],[*,a],R)$. When M finds the checked symbol, it goes the state q_1. M repeats a new search.



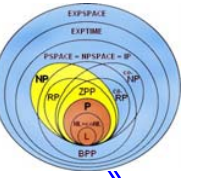
The second branch.

11. $\delta(q_5[B], [*, a]) = (q_7[B], [*, a], R)$. The symbol to the left of c is checked. M starts moving right again, entering state q_7.

12. $\delta(q_7[B], [B, c]) = (q_8[B], [B, c], R)$. M enters state q_8, and proceeds right.

13. $\delta(q_8[B], [*, a]) = (q_8[B], [*, a], R)$. M moves right, skipping over any checked symbols.

14. $\delta(q_8[B], [B, B]) = (q_9[B], [B, B], R)$. If M reaches a blank cell without encountering any unchecked symbol, then M accepts.

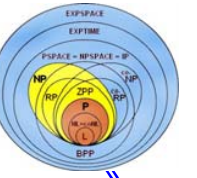


Subroutines

In general, it helps to think of Turing machine as built from a collection of interacting components, or subroutines.

A Turing-machine subroutines is a set of states that perform some useful process.

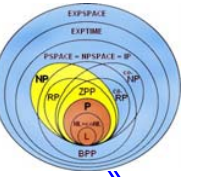
Example Design a TM to implement the function "multiplication". That is, our TM will start with 0^m10^n1 on its tape, and will end with 0^{mn} on the tape.



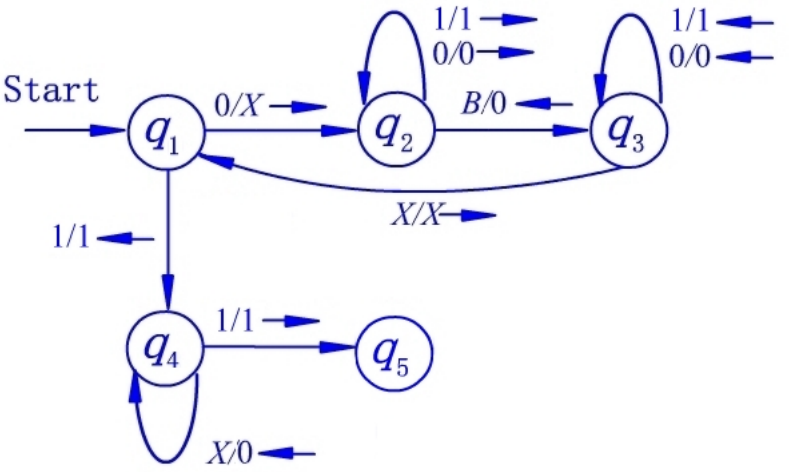
An outline of the strategy is:

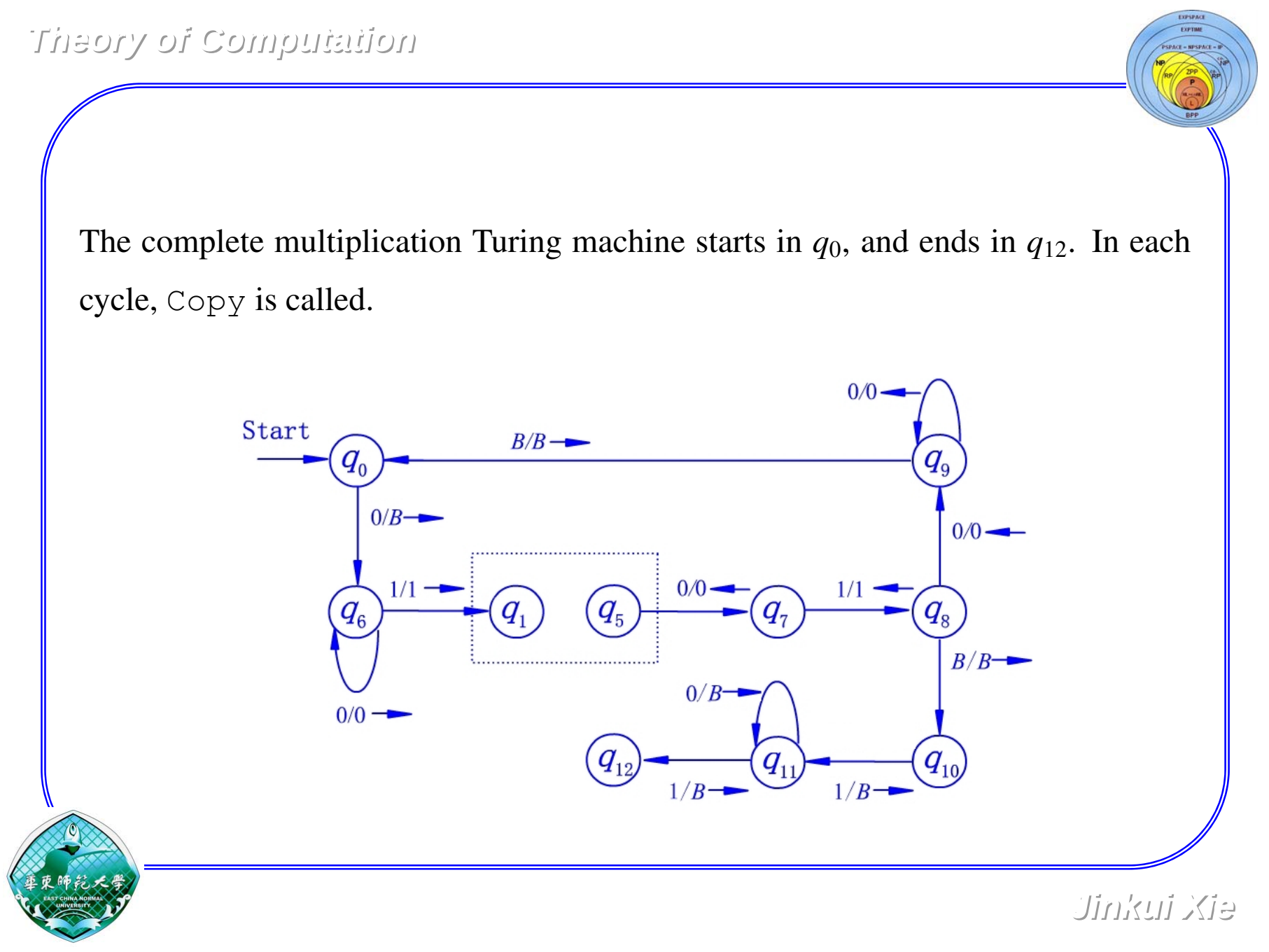
1. Change a 0 in first group to B.
2. Add n 0's to the end of the string.
3. As a result, we copy the group of n 0's to the end m times, once each time we change a 0 in the first group to B.

   When the first group of 0's is completely changed to blanks, there will be mn 0's in the last group.
4. The final step is to change the leading 10^n1 to blanks, and we are done.

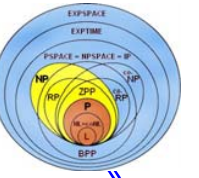


The heart of this algorithm is a subroutine `Copy`, which implemented step (2) above. That is, it converts an ID of the form $0^{m-k}1q_10^n10^{(k-1)n}$ to $0^{m-k}1q_50^n10^{kn}$ for some k.



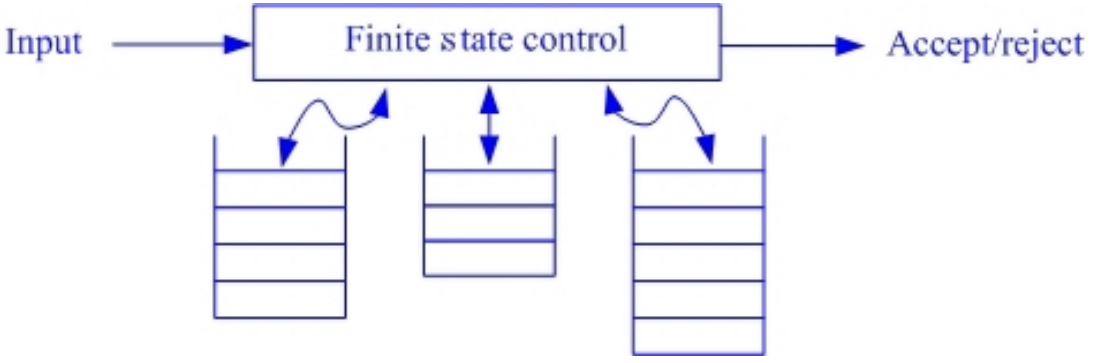


The complete multiplication Turing machine starts in q_0, and ends in q_{12}. In each cycle, `Copy` is called.

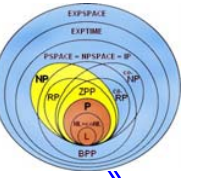


Multistack Machines

We already know that a Turing machine can accept languages that are not accepted by any PDA with one stack. What happens when we give the PDA several stacks?



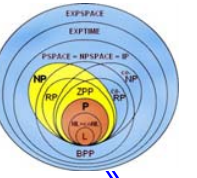
PDA with two stacks can accept any language that a Turing machine can accept.



A *k*-stack machine is a deterministic PDA with k stacks. It has a finite control, and a finite stack alphabet, which it uses for all its stacks.

A move of the multistack machine is based on:

1. The state of the finite control.
2. The input symbol read from the finite input alphabet. Alternatively, the multistack machine can make a move using ϵ input, but to make the machine deterministic, there cannot be a choice of an ϵ-move or a non-ϵ-move in any situation.
3. The top stack symbol on each of its stacks.



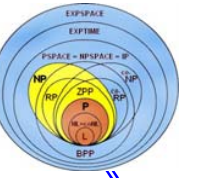
In one move, the multistack machine can:

1. Change to a new state.
2. Replace the top symbol of each stack with a string of zero or more stack symbols.

Thus, a typical transition rule for a k-stack machine looks like:

$$\delta(q, a, X_1, X_2, \cdots, X_k) = (p, \gamma_1, \gamma_2, \cdots, \gamma_k)$$

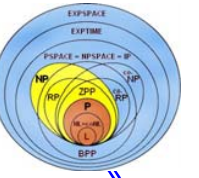
The multistack machine accepts by entering a final state.



We add one capability that simplifies input processing by this deterministic machine: we assume there is a special symbol £, called the *endmarker*, that appears only at the end of the input and its not part of that input.

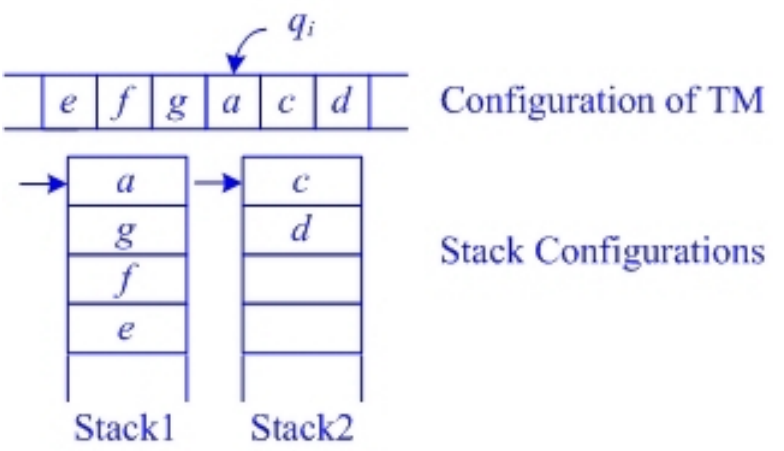
We shall see in the next theorem how the endmarker makes it easy for the multistack machine to simulate a Turing machine.

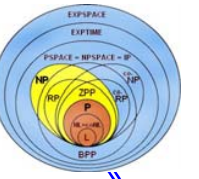
Notice that the conventional TM needs no special endmarker, because the first blank serves to make the end of the input.



Theorem 8.1 *If a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.*

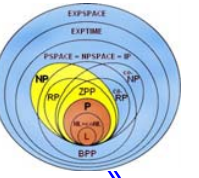
Proof The essential idea is that two stacks can simulate on Turing-machine tape, with one stack holding what is to the left of the head and other stack holding what is to the right of the head.





In more detail, let L be $L(M)$ for some TM M. Our two-stack machine S will do the following first:

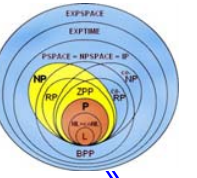
1. S begins with a *bottom-of-stack marker* on each stack.
2. Suppose that w£ is on the input of S. S copies w onto its first stack, ceasing to copy when it reads the endmarker on the input.
3. S pops each symbol in turn from its first stack and pushes it onto its second stack. Now, the first stack is empty, and the second stack holds w, with the left end of w at the top.



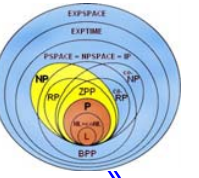
Now S enters the start state of M.

S simulates a move of M as follows:

- S knows the state of M, say q, because S simulates the state of M in its own finite control.
- S knows the symbol X scanned by M's tape head; it is the top of S's second stack. As an exception, if the second stack has only the bottom-of-stack marker, the M has just moved to a blank; S interprets the symbol scanned by M as the blank.
- Thus, S knows the next move of M.



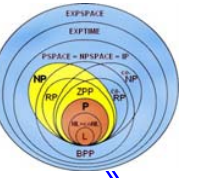
- The next state of M is recorded in a component of S's finite control, in place of the previous state.
- If M replaces X by Y and moves right, then S pushes Y onto its first stack, representing the fact that Y is now to the left of M's head. X is popped off the second stack of S. However, there are two exceptions:
  1. If the second stack has only a bottom-of-stack marker, then the second stack is not changed; M has moved to yet another blank further to the right.
  2. If Y is blank, and the first stack is empty, then that stack remains empty. The reason is that there are still only blanks to the left of M's head.



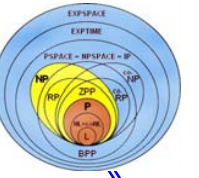
- If M replaced X by Y and moves left, S pops the top of the first stack, say Z, then replaces X by ZY on the second stack. This change reflects the fact that what used to be one position left of the head is now at the head. As an exception, if Z is the bottom-of-stack marker, then M must push BY onto the second stack and not pop the first stack.

S accepts if the new state of M is accepting. Otherwise, S simulates another move of M in the same way. ◄


BREAK FOR 15 MINUTES



Other Models of Turing Machine



Turing Machines with Stay-Option

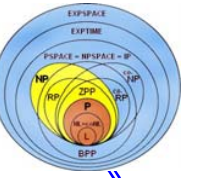
We can define a Turing machine with stay-option by replacing δ in the definition of the standard Turing machine by

$$\delta : \ Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

with the interpretation that S signifies no movement of the tape head.

This option does not extend the power of the automaton.

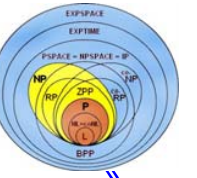
Theorem 8.2 *If M_S is a TM with stay-option, then there is a standard TM M such that $L(M_S) = L(M)$.*



Proof Let $M_S = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a TM with stay-option to be simulated by a standard TM $M = (\hat{Q}, \Sigma, \Gamma, \hat{\delta}, \hat{q}_0, B, \hat{F})$. The simulating machine can be constructed by M_S by defining $\hat{\delta}$ as follows:

- For each transition $\delta(q, X) = (p, Y, L \text{ or } R)$, we put into $\hat{\delta}(\hat{q}, X) = (\hat{p}, Y, L \text{ or } R)$.
- For each S-transition $\delta(q, X) = (p, Y, S)$, we put into $\hat{\delta}$ the corresponding transitions $\hat{\delta}(\hat{q}, X) = (\hat{r}, Y, R)$ and $\hat{\delta}(\hat{r}, Z) = (\hat{p}, Z, L)$ for all $Z \in \Gamma$.

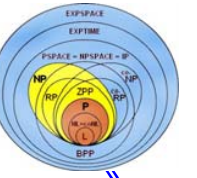
It is reasonably obvious that every computation of M_S has a corresponding computation of M, so that M can simulate M_S. ◄



☞ Simulation is a standard technique for showing the equivalence of automata.

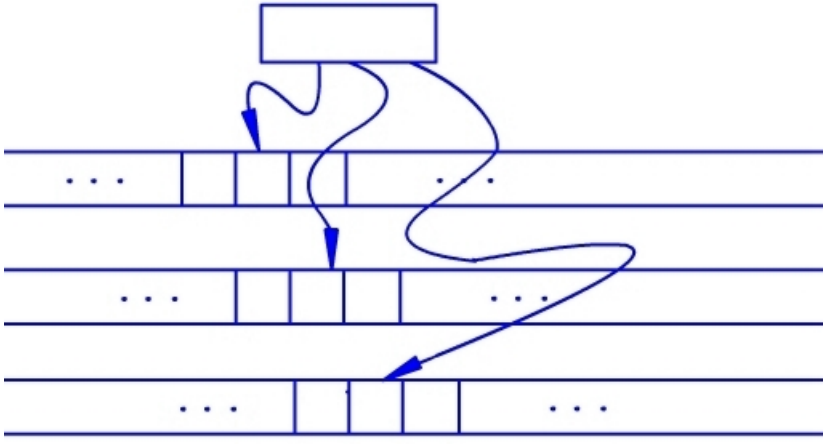
In our subsequent discussion, we use the notion of simulation frequently, but we generally make no attempt to describe everything in a rigorous and detailed way. Complete simulation with Turing machines are often cumbersome.

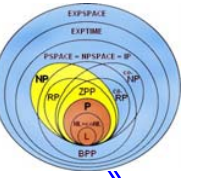
To avoid this, we keep our discussion descriptive, rather than in theorem-proof form. The simulations are given only in broad outline, but it should not be hard to see how they can be made rigorous.



Multitape Turing Machines

A multitape Turing machine is a Turing machine with several tapes, each with its own independently controlled tape head.





We define an n-tape Turing machine by $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where Q, Σ, Γ, q_0, F are as the definition of the standard Turing machine, but where

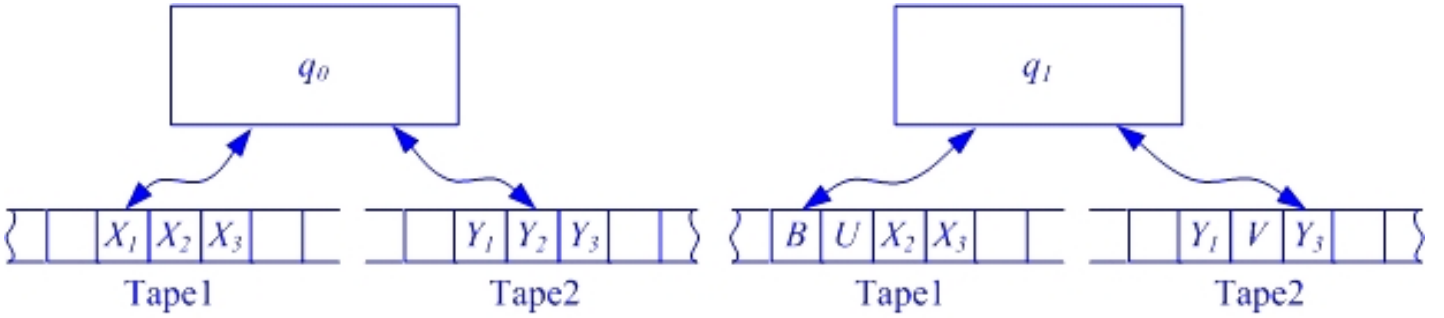
$$\delta : Q \times \Gamma^n \to Q \times \Gamma^n \times \{L, R\}^n$$

specifies what happens on all the tapes.

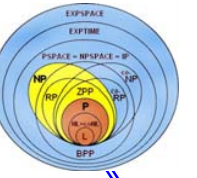
For example, if $n = 2$, then

$$\delta(q_0, X_1, Y_2) = (q_1, U, V, L, R)$$

is interpreted as follows.



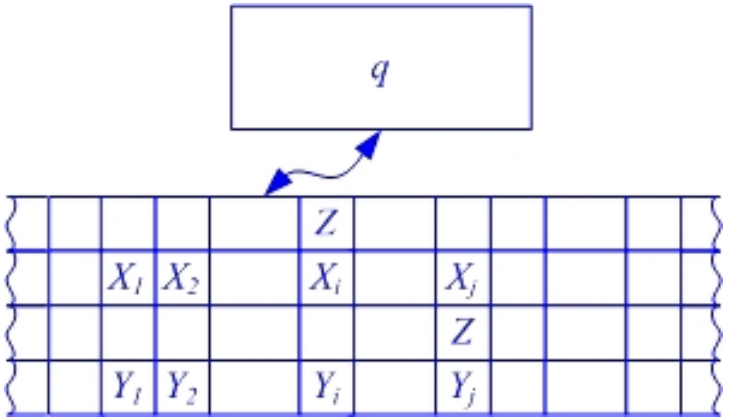
Surely, one-tape TM *is* a multitape TM. On the other hand, we can simulate multitape TM with a one-tape TM. As the result, we know the extra tapes in multitape add no power to the model, as far as the ability to accept languages is concerned.



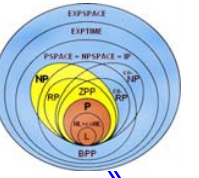
Theorem 8.3 *If M_n is a multitape TM, then there is a one-tape TM M_1 such that $L(M_n) = L(M_1)$. That is, language accepted by multitape TM is recursively enumerable.*

Proof The proof is suggested by following figure. We simulate M_n with a one-tape TM M_1 whose tape we think of as having $2n$ tracks.

Half these tracks hold the tapes of M_n, and the other half of the tracks each hold only single marker that indicates where the head for the corresponding tape of M_n is currently located.



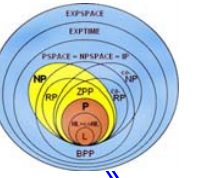
Above figure assumes $n = 2$. The second and fourth tracks hold the contents of the first and second tapes of M_2, track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.



To simulate a move of M_n, M_1's head must visit the n head markers. So that M_1 not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of M_1's finite control.

After visiting each head marker and storing the scanned symbol in a component of its finite control, M_1 knows what tape symbols are being scanned by each of M_n's heads. M_1 also knows the state of M_n, which stores in M_1's own finite control.

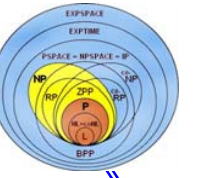
Thus, M_1 knows what move M_n will make.



M_1 now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M_n, and moves the head markers left or right, if necessary.

Finally, M_1 changes the state of M_n as recorded in its own finite control. At this point, M_1 has simulated one move of M_n.

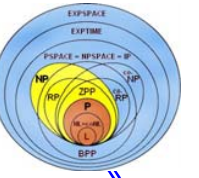
We select as M_1's accepting states all those states that record M_n's state as one of the accepting states of M_n. Thus, whenever the simulated M_n accepts, M_1 also accepts, and M_1 does not accept otherwise. ◄



Example Consider the language $\mathbf{0}^n\mathbf{1}^n$. We described a laborious method by which this language can be accepted by a Turing machine with one tape.

Using a two-tape machine makes the job much easier.

Assume that an initial string 0^n1^m is written on tape 1 at the beginning of the computation. We then read all the 0's, copying them onto tape 2. When we reach the end of the 0's, we match the 1's on tape 1 against the copied 0's on tape 2. This way, we can determine whether there are an equal number of 0's and 1's without repeated back-and-forth movement of the tape head.



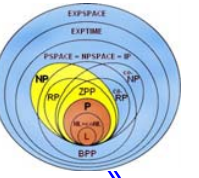
Nondeterministic Turing Machines

A nondeterministic Turing machine (NTM) differs from the deterministic variety we have been studying by having a transition function δ such that for each state q and tape symbol X, $\delta(q, X)$ is a set of tripes

$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \cdots, (q_k, Y_k, D_k)\}$$

where k is any finite integer.

The NTM can choose, at each step, any of the triples to be the next move.

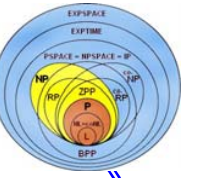


NTM accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state.

Example Let NTM $M = (\{q, p, r\}, \{0, 1\}, \{0, 1, B\}, \delta, q, B, \{r\})$, where δ is given by

δ	0	1	B
q	$\{(q, 1, R)\}$	$\{(p, 0, R)\}$	–
p	$\{(p, 0, R), (q, 0, L)\}$	$\{(p, 1, R), (q, 1, L)\}$	$\{(r, B, R)\}$
r	–	–	–

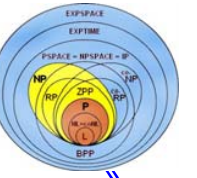
Show the ID's reachable from the initial ID if the input is 0101.



We have several branches:

$$q0101 \vdash 1q101 \vdash 10p01 \vdash \begin{cases} 1q001 \vdash 11q01 \vdash 111q1 \vdash 1110pB \vdash 1110BrB \\ 100p1 \vdash \begin{cases} 10q01 \vdash 101q1 \vdash 1010pB \vdash 1010rB \\ 1001pB \vdash 1001BrB \end{cases} \end{cases}$$

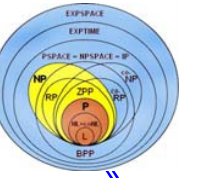
◀



The NTM's accept no languages not accepted by a deterministic TM. The proof involves showing that for every NTM M_N, we can construct a DTM M_D (designed as a multitape TM) that explores the ID's that M_N can reach by any sequence of its choices. That is, we have

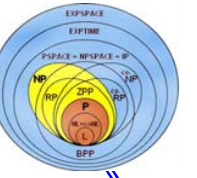
Theorem 8.4 *If M_N is a nondeterministic TM, then there is a deterministic TM M_D such that $L(M_N) = L(M_D)$.*

Notice that the constructed deterministic TM may take exponentially more time than the nondeterministic TM. It is unknown whether or not this exponential slowdown is necessary.


EXPSPACE
EXPTIME
PSPACE = NPSPACE = IP
NP
RP
ZPP
P
NP
RP
L
BPP

Thank you!


華東師範大學
EAST CHINA NORMAL UNIVERSITY


EXPSPACE
EXPTIME
PSPACE = NPSPACE = IP
NP
RP
ZPP
P
L
BPP


華東師範大學
EAST CHINA NORMAL UNIVERSITY