接下来加入一个所谓的“interlude”，介绍一个计算机最底层的理论，也是最重要的理论 — 计算理论，的一些基本知识（prerequisites）。没有这个理论，计算机科学就不能称为是科学，只能是 computing 或者 computing technology。计算理论研究的是：

（1）计算模型（包括形式语言、自动机）

（2）哪些问题是可计算的，哪些是不可计算的（可计算性理论及算法）

（3）计算需要多少时间、消耗多少存储（计算复杂度理论）

在介绍之前，我们先树立一些思想，定义和理解一些必要的概念。首先将任何计算机想象成人。人需要懂一些语言。现在要学习的是，计算机到底可以掌握哪些语言？如果计算机 M 可以“识别”某个语言 L 的一个 string S，则称为 M accepts S，称 L 为 M 的语言。

Symbol – a number, letter, or sign used in mathematics, music, science like a, b, c, 0, 1, 2, ⊆…

Alphabet (Σ) – a collection of symbols（一种语言要使用的字符表）

String (s) – a sequence of symbols like a, ab, 101, sy78（string 可以为空，表示为ε）

Language (L) – a set of strings

Let Σ = {0, 1}.

L1 = set of all strings of length 2 = {00, 01, 10, 11}

L2 = set of all strings of length 3 = {000, 001, 010, 011, 100…}

L3 = set of all strings that begin with 0 = {0, 00, 01, 000, 001, 010…}

可以观察到，当Σ确定后，有的 language 是有限的，比如 L1、L2；有的是无限的，比如 L3。

Powers of Σ:

Σ^0 = set of all strings of length 0 = {ε}，这个符号读作 eplison

Σ^1 = set of all strings of length 1 = {0, 1}

Σ^2 = set of all strings of length 2 = {00, 01, 10, 11}

…

Σ^n = set of all strings of length n

Cardinality of Σ: number of symbols in Σ

Cardinality of $\Sigma^n = 2^n$

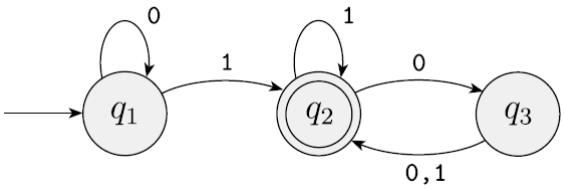
Σ^* = set of all possible strings of all lengths over Σ = $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3$…

定义了这些符号和概念之后我们就可以进入到计算理论的正文。首先是最简单的计算模型，叫 finite state automata，简称 FSA。其中，如果 FSA 有输出，还可以分为 Moore machine 和 Mealy machine。如果 FSA 没有输出，则可以分为 DFA、NFA、ε-NFA。我们忽略前者，只关注没有输出的 FSA。

DFA – deterministic finite automata（也叫 finite automaton）

（1）it is the simplest computational model

（2）it has a very limited memory



看上面的图，里面有圆圈 q1, q2, q3，称为状态（state）；还有箭头（arrow or directed edge），箭头上有 0 和 1 这样的 symbol。一个特殊的箭头来自 nowhere，代表其所进入的状态是初始状态，一个特殊的圆圈是两个套在一起（double circled），代表其终止状态。整个计算的过程就发生在刚才介绍的这些元素上，计算必须从初始状态开始，从终止状态终止。下面对刚才的模型和计算过程进行 formalize：

DEFINITION 1.5

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

整个的计算过程特别适合将自己想象成一台机器进行模拟（模拟人生，小时候玩的一款游戏，作者根据马斯洛需求层次理论构建了游戏中的人工智能）。假如此时此刻你站在初始状态 q1，读到了一个 symbol（信号），发现是 0；此时你打开一本操作手册，查看一下此时在 q1 点读到 0 该做什么，这个操作集就是 transition function。按照它的要求，需要从 q1 跳转到 q1，尽管你依然站在原地，但是这已经是发生了一次状态转换了。相当于在原地搭起帐篷过了一夜。用符号可以表示为：δ(q1, 0) → q1。按照这种方式，你可以踏过千山万水，只要不想回家，永远可以在外面闲逛。一个状态已经路过 1000 遍了，都可以再走第 1001 遍。但最终，你要在一个合适的时机到达终止状态，完成整个旅程。这里注意，到达终止状态不代表一定要终止，而是终止时一定要在终止状态，这个逻辑要搞清楚。你可以在终止状态上逗留一下再离开去往别的地方，但最终的最终，要再回来在这里结束整个旅程。初始状态 q0 是唯一的，但是终止状态可以是多个，所以上面用 F 这个集合来表示。

DFA 之所以叫 DFA 是指，transition function 针对一个“状态+信号”对儿，只会将你带到唯一下一个状态；整个过程是 deterministic 的。不存在你在 q1 状态读到 symbol 0，下一步有多种状态可选，不知道该往哪里走的情况出现。成为 DFA 还有一个要求，就是在其每个状态上，针对字符表中的任意 symbol，都有一个 arrow 指向下一个状态。不能字符表里面有 {0,1}，结果某个状态上只能读 1，却不能读 0 的情况出现。

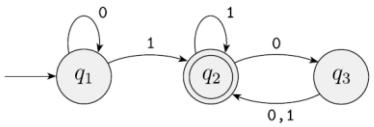


FIGURE **1.6**
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

4. q_1 is the start state, and
5. $F = \{q_2\}$.

如果存在 a set A of strings 能被上面 M1 这个 DFA 所接受，我们说 A 是 M1 的 language，写作 L(M) = A。这时，称为 M1 recognizes A 或者 M accepts A。因为在后面 accepts 这个词会在表达 M1 accepts strings 和 M1 accepts language 时体现的语义不同，这里我们使用 recognizes 来避免疑惑。所以，L(M) = A 等价于 M1 recognizes A。

如果一个 DFA M 的初始状态恰好同时是 M 其中的一个终止状态，那么意味着 M 可以接受 empty string ε。到这里，你可以发现每一个 DFA 都对应着一个 language，且这个 language 是可计算的 language，是 M-recognizable 的 language。

我们定义了什么是 DFA，接下来定义 DFA 上的计算。让 M = (Q, Σ, δ, q0, F)为一个 DFA 且 $w = w_1w_2\cdots w_n$ 为一个 string，且 w_i（1 《=i 《=n）属于Σ。则 M accepts w，当存在一系列的状态 $r_0,\cdots,r_n$ in Q，它满足三个条件：（1）$r_0 = q_0$，（2）$\delta(r_i, w_{i+1}) = r_{i+1}$, for i = 0,⋯,n-1，且（3）$r_n \in F$。我们说 M recognizes language L，当 L = {w | M accepts w}。

讲到这里，你会发现一个 DFA 对应一种 language。那么一定会产生一个疑问，是不是所有的 strings 都可以被 DFA accepts? 答案当然是否定的。如果某些 DFA 可以 accepts 一个语言 L，则 L 称为 regular language。regular 这个词从何处而来？只知道来自其作者 Stephen Cole Kleene，他是 Alonzo Church 的 PhD 学生，Alan Turing 的师兄。但语义上的解释并没有。

接下来你肯定会有疑问，什么样子的语言是 regular language，什么样子的不是（non-regular language）。有人说能被 DFA accepts 的就是，那不成了互相定义、循环定义。你肯定想知道满足 regular language 的内涵是什么？具体有哪些规律或者 pattern? 这里，我们首先定义几种运算，称为 regular operations，分别是 union、concatenation、star。

DEFINITION **1.23**

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

- **Union**: $A \cup B = \{x | \, x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{xy | \, x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1x_2 \ldots x_k | \, k \geq 0 \text{ and each } x_i \in A\}$.

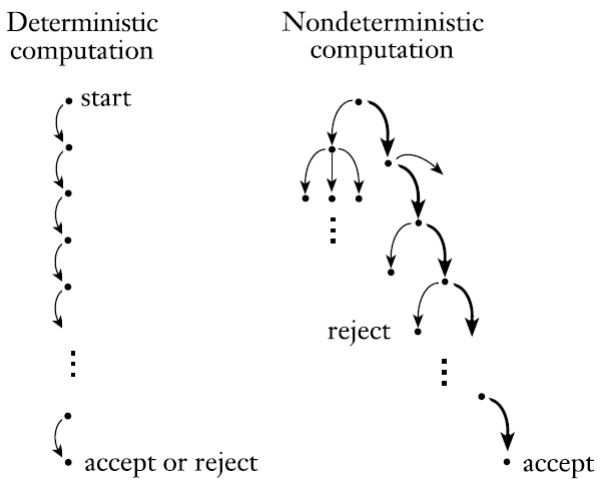
这里需要定义一个叫做 closed 的概念。和逻辑语言里面的 closed 一样，是定义在某个运算

在某个集合上。我们说某个集合针对于某个运算是 closed，意味着对这个集合里面任何元素进行这个运算，其结果都已经在这个集合里面。We say that N (the set of natural numbers) is closed under multiplication, we mean that for any x and y in N, the product x * y is also in N。很明显 N is not closed under division，因为 1/2 已经不再是自然数。基于此定义，我们可以得到 the class of regular language is closed under union and concatenation（Theorem 1.25 1.26）。

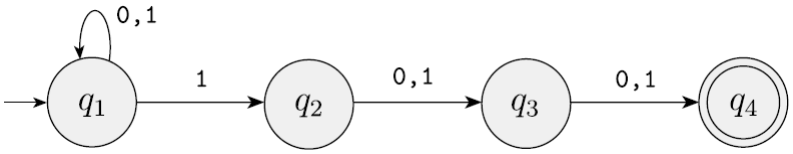
上面看到从某个状态，读到一个 symbol 之后，转移到另一个状态，这个 transition function 是 functional 的，也就是说去往的状态是确定的，唯一的。相当于来到一个分岔路口，指向标告诉你接下来要走哪一条路。那么有没有一种情况，当处在某个分叉路口的时候，不再有指向标，而是需要在多个选择中去做抉择。这种情况，我们叫做 non-deterministic finite machine，简称 NFA。

NFA 的状态数依然是有限的；字符表集合和 DFA 也没什么不同，只不过在状态转化的时候，读到的 symbol 不仅仅来自字符表集合，还可以是ε（empty string)。ε是一个很重要的符号，它不属于字符表，当某个状态上读到它的时候，意味着你可以在不读任何 symbol 的情况下去到下一个状态；当然最大的不同还是在于 transition function 不再是 functional 的，意味着在某个状态上读到某个 symbol 后选择不止一种，可以去到 a 状态，还可以是 b 状态等等。那具体的计算过程是什么样子的呢？依然把你自己想象成那个移动的主角。现在当你在路口面临多个选择时会怎么做？一种方法是先走其中一条路径，走完了之后返回到这个路口再走另外一条，类似于深度优先搜索。NFA 是这样运行吗？答案是不是的。而是，在路口处你会像孙悟空一样变身为多个分身（copy)，以并行的方式继续所有的路径。到了下一个路口，又面临多个抉择，那么分身也会继续分身进行并行计算。如果到达某个状态发现 transition function 无法读取下一个 symbol b，换句话说，不存在 input 为 b 的 transition function，再换句话说，并不存在一个 label 为 b 的 arrow 从该状态出发去往别的状态，那么这个分身连同整个计算路径到这里就死亡了，因为它 rejects 了这个字符串。但是其它的分身继续工作，其它的计算路径继续活着。如果上述任意分身读到 string 最后一个 symbol 时恰好到达了 accept state，则 NFA accept this string。上面介绍了ε这个符号，如果计算过程中读到它，则同样地，将变为多个分身，一个分身停留在原地，其他分身去往每个当前状态上能到达的下一个状态。

NFA 的计算过程无论从定义中，还是刚才的职场模拟中，都可以看作是 tree of possibilities。



下面看一个例子来加深理解，以及在计算过程中遇到“guess”这个词：



直观上，这个例子是想表示一种语言？该语言包含的所有字符串都至少长度为 3，且倒数第三个位置一定是 1。此时你站在 q1 初始位置，读到 symbol 1，有两个选择，一个是继续停留在 q1，另一个是去往 q2。如果遇到一个 string 0101111，当读到第一个 0 的时候，计算路径是确定的，依然停留在 q1，但是读到第一个 1 的时候，到底是停留在 q1 还是去往 q2，换句话说，NFA 怎么才能知道当前这个 1 是不是倒数第三个位置的 1。答案是不知道，所以每次遇到 1，NFA 都必须 guess 这个 1 是倒数第三个，派出一个分身去往 q2，则在 q2 出读到第二个 0，前往 q3，在 q3 读到第二个 1，前往终止状态 q4，但是 string 中还有三个 1 没有被读到，但是 q4 状态上已经无法前往任何地方，所以这条路径到这里会被 kill 掉。同样地，读到第二个 1 的时候（倒数第四个 1）也会遇到同样的事情，也会被 kill 掉一次。这就是 guess 的概念，在它拿不准的时候，需要赌一下这次就是了，然后去看看到底是不是？然后被拒绝。再去，再被拒绝…

这样，一个 NFA 的正式定义就可以得到了：

DEFINITION 1.37

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

从定义可以看出，NFA 与 DFA 唯一的不同在于 3，transition function 不仅仅是 Q 中某一个状态，而是 Q 的任意一个子集里面的所有状态，所以我们用 Q 的 power set 表示 P(Q)。除此之外，还有一些定义上体现不出的不同。这些不同中，一些是硬性规定，另一些是定义和规定中推导出的“隐性不同”。硬性规定中记住两条：（1）DFA 不接受ε，而 NFA 接受；（2）DFA 在任意状态对字符表中每一个 symbol 都有对应的 transition function，而 NFA 可以只对其中一部分 symbol 有对应的 transition function。

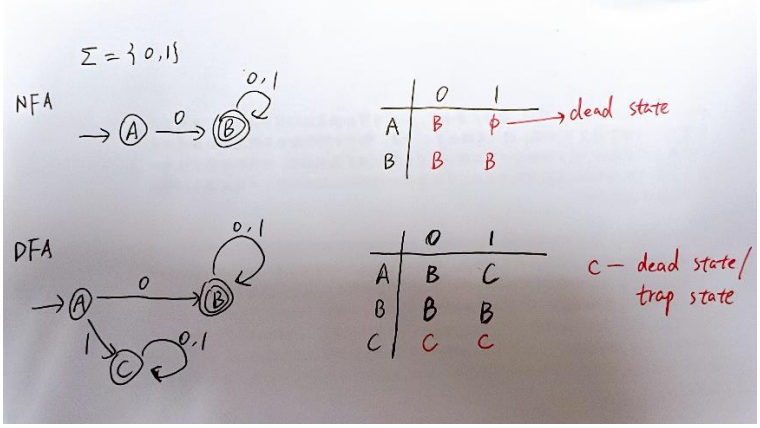
接下来是一个很重要的结论：NFA 与 DFA 的等价性，也就是一个 NFA 与它对应的 DFA，二者 recognize 同样的语言。这一点很多人可能没有想到，有些意外。当然，NFA 也有好处，它让多个任务并行的能力可以节省状态数量（节省内存）。

想证明 DFA 和 NFA 的等价性，需要证明每一个 DFA 都是 NFA，反之亦然。按照定义，每一个 DFA 天然就是一个 NFA，但是反过来就不容易证明了。我们需要证明针对每一个 NFA，都存在且可以构建一个对应的 DFA（该证明我们留给学生自己去看）。

如果一台 NFA 有 k 个状态，则它最多有 2^k 个 subsets of states（一个 state 或者在集合里面，或者不在，所以是 2 选 1，一共有 k 个元素，所以是 2^k）。现在我们想为每一个 NFA 找到其

对应的 DFA，有没有一个算法？答案是有。我们先将这个算法展示出来，再解释算法的正确性（soundness）。

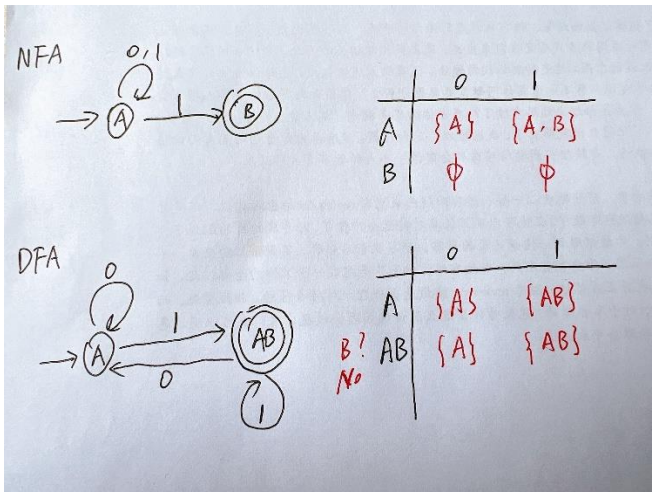
我们看一个例子来获得一些灵感。我们让 NFA 和 DFA 分别 recognize 一个 language L = {set of all strings over (0, 1) that starts with 0}



看上图，拿到一道题目，NFA 其实比 DFA 要好找、好画。画 NFA 需要一些经验和一点点规律，我们这里就不赘述了。画好了 NFA 之后我们将其转化为等价的表格（见 NFA 右边的图）。图中信息表示在状态 A，读到 symbol 0，跳转到状态 B；在状态 A，读到 symbol 1，跳转到一个叫做 dead state 的地方，用符号 phi 来表示。这是因为 NFA 中，不需要对字符表中每一个 symbol 都能反应，这里在状态 A，就只能读到 0 才有反应，针对 1 并没有一个 transition function。这时候，去到的状态叫做 dead state。接下来我们开始画对应的 DFA。我们首先根据上面你的表格来画 DFA 的表格，再转成 DFA 图。依然是从状态 A 初始，当 A 读到 1 的时候，NFA 中去往 dead state，但是在 DFA 中，任意状态都必须针对字符串的 symbol 有 transition function，所以这里我们需要创造一个新的状态，也就是 DFA 表格中的 C。当我们创造了状态 C，必然就有从 C 出去的 arrows，一个是读到 0 之后，另一个是读到 1 之后。而显然，一个 dead state 读到 symbol 时候，只能指向自己。所以这个表就做出来了。再根据这个图画出对应的 DFA 图。

这个简单的例子如果讲到这里，很多同学可能觉得已经明白了如何转化 DFA。但其实，里面有很多小细节我们都没讲到，错过的话以后画 DFA 会有很大几率发生错误。哪些细节呢？首先，当我们画 DFA 的表时，要按照顺序来生成状态。首先是从初始状态 A 进入，这个没问题，当在 A 读到 0，1 之后会跳转到 B，C。这时候，你的 DFA 里面有了三个状态：A，B，C。接下来你可以为 B 写出后面的跳转状态情况，再为 C 写。如果你发现，在 B 或者 C，读到 0 或者 1 的时候跳转到一个新的状态 D，意味着你的 DFA 里面又引入了新的状态，那么你还要为状态 D 写 transition function，直到整个系统不再引入新的状态（闭包）。这个很好理解。但是难理解的是，当“现有的表”中没有出现某个状态，是不能为其写跳转情况的，哪怕状态曾经出现在 NFA 的表格中。看下面这个例子：

让 NFA 和 DFA 分别 recognize 一个 language L = {set of all strings over (0, 1) that ends with 1}：



上图首先画了 NFA。主要关注 NFA 的表格转到 DFA 的表格：初始状态依然是 A，读到 0 的时候，跳转到 A；读到 1 的时候，跳转到 A 和 B 两个状态。而在 DFA 中，当在某个状态读到某个 symbol 的时候，只能跳转到某一个状态，那怎么办？解决方法是将 A、B 两个状态合并为一个状态，称为 AB。至此，DFA 表格中只有初始状态 A 和读到 1 跳转到的状态 AB，没有状态 B（NFA 里面才有状态 B）。接下来要为状态 AB 设计 transition function。在状态 AB 读到 symbol 0，该去往哪个状态呢？应该是 NFA 里面在 A 状态读到 0 去往的状态和在 B 状态读到 0 去往的状态的并集，也就是{A}∪∅。同样地，在 AB 读到 1 去往的状态也应该是 NFA 里面状态 A 和 B 读到 1 去往的状态的并集，也就是{A, B}∪∅。而状态 A, B 已经合并为 AB，所以我们得到该 DFA 的表格。DFA 的表格里面不再有状态 B。

NFA 转换 DFA 的思想显而易见，就是从起始状态开始，将读到同一个 symbol 而去往的状态“合并”为一个状态，并将这些被合并状态上的“发生的故事”继续合并，也就是读到某个 symbol 而发生的行为取并集，直到终止状态。从状态数量上看，是 NFA 存在优势，因为如上面所分析，最多需要 2^k 个状态才能画出对应的 DFA，k 是 NFA 的状态数量。

接下来，我们使用 FSA（finite state automata）来指代 DFA 和 NFA。每一个计算模型都对应着它所能处理的语言，FSA 也是一样。FSA 对应的语言称为 regular language。其字符表、语法、语义我们不做讲解，可以自己私底下去了解（非本门课程内容）。这里一定会有一个疑问：这个 regular language 有没有什么特点或者叫边界，可以帮助我们确定一个语言是不是 regular language？又或者是不是所有的语言都是 regular language？答案是否定的。不然的话，FSA 岂不是取代了图灵机成为了通用计算模型？根据上面的分析，我们首先可以得到一些重要的定理或者引理：

（1）A language is regular iff some regular expression describes it.

（2）A language is described by a regular expression iff it is regular.

上面两句话不是一回事，第一句说一个语言是 regular 的，则一定存在一个 regular expression 描述它；反过来也一样。暗示着 regular expression 的表达力比所有 regular language 的集合的表达力要大。第二句才说明了二者的等价性。

要判断一个语言是不是 regular language，我们引入一个 pumping lemma 的概念。这里不再使用模型论的 bisimulation，因为 regular language 的语义解释不是基于模型论的。

THEOREM **1.70**

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Recall the notation where $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.

如果一个语言 A 是 regular language，则存在一个数字 p，称为 pumping length，使得 A 中任意长度不短于 p 的 string s（|s|⩾p），可以分为三个部分 s=xyz，并且进一步满足以上三个条件。以上定义用数学语言描述 pumping lemma，那该如何从现实世界的角度理解呢?

（1）如果一个语言 A 是 regular language，则一定存在一个 FSA 可以 recognize A，则 A 中的任意 string 都可以被这个 FSA 所 accepts。

（2）FSA 的状态是有限的，而 A 中的 string 是无限的。如果 FSA 的状态数量是 p，则 A 中必然存在一些 strings，它们的长度，也就是符号的数量大于 p。根据 pigeon hole principle 的结论，5 个物体放入 4 个盒子一定至少存在至少一个盒子里面的物体⩾2。则同样的道理，如果 FSA accepts 长度大于 p 的 strings，则必然至少存在一个 state 被访问了超过一次，比如下列的计算过程（一组状态被访问的记录）：

$$q_1, q_2, q_3, \cdots, q_k, \cdots, q_k, \cdots, q_{n-1}, q_n$$

这就意味着必然存在一个 substring 会被 $q_k, \cdots, q_k$ 读到。这个 substring 就是 lemma 里面的 y，y 之前的 substring 就是 x，y 之后的是 z。显然，y 可以被多次重复（pumped），比如 xz，xyz，xyyz，xyyyz，其产生的新的 string 依然在 A 里面。

（3）Pumping Lemma 中条件（2）要求 y 这个字符串的长度至少是 1。没有这个条件，整个 lemma 就会无意义为真，我们叫 trivially/vacuously true，因为没有 pumped 的字符串则 xy^iz 变为了 xz，也就没有了变量部分，s = xz。则 Pumping Lemma 自动成立。

（4）Pumping Lemma 中条件（3）要求 substring y 起始的地方一定要在 p 范围之内。因为第一次开始 pumped 的时候一定是 p 之内的。不然会误伤很多本来是 regular language 的语言。

Pumping Lemma 的发明者是 1976 年图灵奖得主 Michael O. Rabin 和 Dana Scott，两位都是 Alonzo Church 的博士学生，Alan Turing 的师弟。

这里注意：pumping lemma 只能证明一个语言不是 regular language，但是不能证明一个语言是 regular language（只负责找茬，不负责善后）。下面步骤证明一个语言 A 不是 regular 的：

（1）假设 A 是 regular
（2）则一定存在一个 pumping length p
（3）所有长度大于 p 的 strings 都可以压缩（pumped），|s|⩾p
（4）现在找到一个 string s in A，满足|s|⩾p
（5）将 s 分为三部分 s=xyz
（6）证明 xy^iz 不属于 A，for some i
（7）接下来考虑所有 S 分为 xyz 的情况（有哪些分法？）
（8）证明任何一种分法都不可能满足上述 pumping 的三个条件
（9）得出结论：s 不能被 pumped（与（3）矛盾）

接下来看一个使用 pumping lemma 证明不是 regular language 的例子：$A = \{0^k1^k \mid n \geqslant 0\}$
假设 A 是 regular 的，则存在一个 pumping length p
$s = xyz = a^pb^p = aaaaaaabbbbbbb$ (p = 7)
Case 1：y 在 a 部分里面，比如 aa aaaa abbbbbbb
Case 2：y 在 b 部分里面，比如 aaaaaaab bbbbb b
Case 3：y 在 a 和 b 部分里面，比如 aaaaa aabb bbbbb
$xy^iz = xy^2z$，也就是让 i = 2。你会发现，无论是上面哪种 case，都不可能。

之所以跳出这门课内容范围之外，介绍几种计算模型，包括 pumping lemma，是想说明，每一种计算模型都对应着其可计算模型（某种语言）。对于可计算语言，我们有各种方法判断其表达力强弱，侧面反应一个计算模型的“计算能力”强弱。在基于模型论解释的逻辑语言中，我们求助于 bisimulation；而在 regular language 上，我们求助于 pumping lemma。

接下来，我们要介绍的是一种新的计算模型，叫做 pushdown automata。介绍它之前，我们先看它对应的语言，叫 Context Free Language，简称 CFL。

CFL：a language generated by some context free grammar (CFG)；CFG 正式定义如下：

DEFINITION 2.2

A *context-free grammar* is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

一个 4-tuple。其中，V 中的 variables 同时也叫 non-terminals 或者 non-terminal symbols，所以 V 需要与Σ是 disjoint 的关系。说的比较模糊的是 R，也就是 a finite set of rules，我们通常把这些 rules 称为 production rules。它的形式是：

$$A \rightarrow a$$

其中，$A \in V$ 且 $a = \{V \cup \Sigma\}^*$。举一个例子，依然是之前熟悉的例子，regular language 的处理不了那个例子 a^nb^n。用 CFG 定义结果如下：

$$G = \{V, \Sigma, R, S\} = \{(S, A), (a, b), (S \rightarrow aAb, A \rightarrow aAb \mid \epsilon), S\}$$

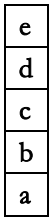
观察，S 是我们的初始变量，所以规则中第一个是 S 起始的：

$$\begin{aligned} S &\rightarrow aAb \\ &\rightarrow aaAbb \text{ (by } A \rightarrow aAb) \\ &\rightarrow aaaAbbb \text{ (by } A \rightarrow aAb) \\ &\rightarrow aaabbb \text{ (by } A \rightarrow \epsilon) \\ &\rightarrow a^3b^3 \end{aligned}$$

CFL 比起 regular language 表达力更强。同样的方式，我们可以判断一个语言不是 CFL，用的是专为 CFL 涉及的 Pumping Lemma，这里不再赘述，感兴趣的可以去阅读相关资料，难度不大。现在转回 pushdown automata（PDA）的介绍，它有这样几个特点：

（1）it is more powerful than FSA
（2）FSA has a very limited memory but PDA has more memory
（3）PDA = FSA + a stack（with infinite size）

stack: an abstract data type that serves as a collection of elements, with two main principal operations:



上面图里就是一个典型的 stack，一个“抽象的”数据存储模型。像一个个的小格子，每个格子里面存储一个 symbol，采用“后进先出”的方式 last in first out (LIFO)进行读写操作。这里，a 是第一个进入到 stack 的元素，接下来是 b，直到 e 最后一个进入。

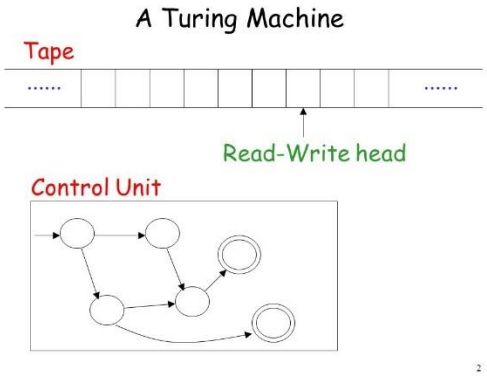
push (down): adds an element into the stack
pop (up): removes the topmost element from the stack

push 的操作是“写”的操作，将某个 symbol 写入 stack 并存储在 stack 最上面一个格子；pop 的操作是“读”的操作，读到 stack 中最上面的那个 element，并将它从 stack 中弹出去。所以，pushdown automata 是一个能在 input string 上进行“读”操作，这点与 FSA 一样；但还可以在 stack 上进行“读”和“写”双操作，这是比 FSA 厉害的地方。我们用现实生活中的其他例子来比喻下这种“只读”和“可读可写”方式带来的不同。我们在学习的时候，有的人喜欢边学边记笔记，无论是记到纸质还是电子文件，以便随时调出来查阅；有的人对自己的记忆力非常自信，觉得“读到了”和“理解了”就是“写下了”和“记住了”且“不会忘了”。对于后者，事实证明，很可能是一个陷阱（pitfall），一个记忆陷阱。大脑的 memory 是有限的，记忆的东西增多，很多以前记下的东西会按照先进先出（first in first out，FIFO）的方式被遗弃掉。所以当你可以执行 PDA 操作时，不要只做 FSA 的事情，莫名放弃掉 stack 的作用。要找到一个类似于 stack 可以永久存储的地方，比如笔记本或者各种 Pad，将知识保护好。尤其是那些需要二次加工才能产生的知识。很多知识是你随时拿起课本看一下就能理解的，而还有很多是需要很长时间深入思考进去才能在书本上的浅层知识或者基础知识上衍生出来的“深层知识”，也就是所谓的“理解”。允许在某个地方进行写的操作相当于给了一个机器记忆的能力。针对之前的例子：a^nb^n。它可以记住读到了多少个 a，然后通过 pop a 的方式找到同样数量的 b。

PDA 相当给了人们一个可以无限存储的地方，但存储的方式非常受限：只能按照顺序一个一个写和读，换句话说，PDA 的所有操作都只能在 stack 的最顶上的元素上进行，很像是自动售货机，只在每个 tray 的最前面那个物体上进行操作。而一个 stack 不能记住那么多东西，或者同时记住一些东西。于是在经典 PDA（也就是 FSA + 1 stack）的基础上，扩展为 FSA + 2 stacks，或者更多。它们的计算能力不同。我们在作业中，要求证明 PDA + 2 stacks is more powerful than PDA + 1 stack，然后证明 PDA + 3 stacks is not more powerful than PDA + 2 stacks。给一个提示，比如为什么本体里面只需要有二元关系，不需要三元关系？是不是所有的 3 stack PDA 可以转换为 2 stack PDA？而 2 stack PDA 不可以转化为 1 stack PDA？依然是上面的例子，如果字符串变为 $a^nb^nc^n$，1 stack PDA 能否处理？2 stack PDA 呢？如果变为 $a^nb^nc^nd^n$，2 stack PDA 能否处理？3 stack PDA 呢？

这样的模型依然满足不了更复杂的计算需要，或者对应表达力更强的语言。于是还有新的

计算模型，比如 Linear Bounded Automata（LBA），比如在 PDA 的基础上有 Turing Machine（TM）。我们接下来介绍下 TM，这是最重要的计算模型，也是计算的通用模型。一台计算机做的任何任务，TM 都可以做。TM 做不了的事情，说明这个事情是计算不可行的（computationally infeasible）或者叫不可计算的（not computable or uncomputable）。针对于某个语言 L，如果一个 TM 不能 recognizes it，则称为 undecidable。



看上面的图，TM 有一个 tape，长度无限（tape 左边有界，右边没有）；tape 被分为一个个离散的小格子，称为 cell，cell 上可以存储一个 symbol；有一个 tape head，可以在这个 tape 上左右移动，且每次只能移动一步；head 可以读 cell 上的 symbol，还可以将其擦掉，写上另一个 symbol；tape 上有几个位置是 accept cells，有几个位置是 reject cells。就这样一个简单的东西，就是 TM，它的计算过程是：input string 会从往右停留在 tape 上等待 TM 检阅，其它 cell 都用空白符号填充。在执行完所有操作后，head 停留在了 accept cell 的位置，则 TM accepts 这个 string；如果停留在 reject cell 位置，则 TM rejects 该 string；还有一种情况，TM 进入了一个 infinite loop，在这个 loop 里面永远走不出来，所以结果是 never halting。这是 TM 的三种终止形式。我们把上面的语言描述用数学语言定义如下：

DEFINITION 3.3

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

上图中清楚地定义了一个 standard TM 应该是什么样子的。与其他计算模型不同的地方：

2. Σ规定的是 input string 可以使用哪些 symbols（tape head 可以读到哪些 symbols），这里面不包括 blank symbol，这里我们用字母 b 来表示 blank symbol；

3. Γ规定的是 tape string 可以使用哪些 symbols，这里面包括了 blank symbol，所以能够写到 tape 上的 symbols 应该是既可以来自Σ，也可以是 b；

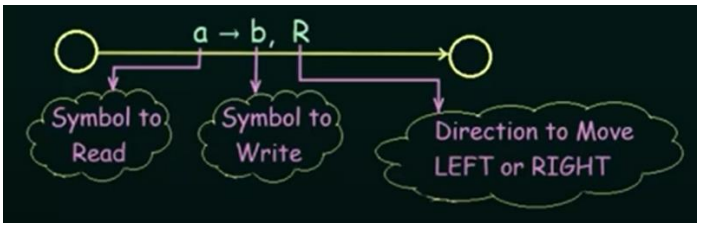
4. transition function 是最最最重要的，它的输入是一个 2-tuple，包括（1）当前所在状态，也就是当前所在的 cell，以及（2）读到的 symbol；输出是一个 3-tuple，包括（1）接下来要去到的状态，（2）当前状态上写入的 symbol（注意注意这里是写入到当前状态，不是下一个状态，{L, R}指定到了下一个状态后，head 往左移动还是往右移动。

*所有的空的 cells 都会自动填充上 b，而不是就空置着

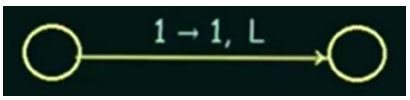
*既然Σ里面不包含 b，则第一次遇到 b 的时候意味着字符串的终结

*因为 tape 左边是有界的，当 head 已经在最左边的 cell 上，但是又接到了向左移动的指令怎么办？答案是继续停留在当前的 cell，不做移动。TM 的计算除非到了 accept 或者 reject state，不然会一直持续下去；换句话说，TM 不可以在非终止节点终止。

上面是 TM 的基本元素的介绍，下面我们看一下 TM 的计算过程：



（1）tape head 读所在 cell 的 symbol（图中左边 node）
（2）tape head 在当前 cell 写入新的 symbol（图中，a 是当前 cell 读到的 symbol，b 是写入到当前 cell 的 symbol，R 是移动方向，这个 R 是在上一个状态的 transition function 就决定好了的）
（3）如果当前 cell 的 symbol 不变，则依然做一步写入操作，只不过用当前 cell 的 symbol 代替它自己，见下图用 symbol 1 代替 1。



Standard TM 称为标准 TM 或者经典 TM，只有一条 tape。但是实际还有很多 TM 的变种，比如多个读写头的 TM，比如多个 tapes 的 TM，比如 tape 左边也无边界的 TM 等等，再比如 non-deterministic TM，但不管是什么样的变种，都可以 simulated by standard TM。TM 在 1936 年被 Alan Turing 发明，TM 这个名字是他的老师 Alonzo Church 起的，并且 Church 独立发明了 lambda-calculus，它与 TM 具有相同的计算能力。二者追随哥德尔的工作，形成了一个伟大的开创性的工作，它是计算的基础，称为 Church-Turing thesis。用直白的话解释它就是：任何计算行为，计算任务，都可以被转化为 TM 上的计算问题；反过来，凡是不能被 TM 计算的问题，就是问题本身不可计算，不存在一个更强大的计算模型，使得该问题可以被这个模型所解决，但不能被 TM 所解决。

我们看一个具体的例子 $L = \{0^n1^n2^n \mid n \geqslant 1\}$。这个例子能被 1 stack PDA 解决吗？不能吧？我猜是不能，但你们在作业中要证明。看看它怎么被 TM 完美解决。这个语言收集了所有 strings，满足 string s = xyz，其中 x 部分是 n 个 0，y 部分是 n 个 1，z 部分是 n 个 2。我说下 TM 的思路，首先某个字符串比如 s = 001122 会进入到 tape 上等待 tape head 读写。tape head 读到第一个 0，在这里将其擦掉，写入 X（用 X 替换 0），接着向右移动，读到第二个 0，写入 0（本状态上 symbol 保持不变），继续向右移动，读到第一个 1，写入 Y，向右移动，读到第二个 1，写入 1，继续向右移动，读到第一个 2，写入 Z，然后向左移动，每次移动一步，直到遇到 X（撞墙了），向右移动，如果再读到 0，改写为 X，然后重复上述动作。在这个过程中，你会遇到下面几种情况：
（1）最后 head 再一次读到 X，按照上面方式向右移动，发现下一步读到是 Y，意味着所有的 0 都被替换为了 X，那么后面就不该再有任何 1 或者 2，如果有，则 reject；没有，且没有任何其它非 XYZ symbol，则来到了 b 状态，accept；
（2）head 在将某一个 0 改为 X 后，向右移动，发现后面不再有 1，则 reject；
（3）同样的道理将某一个 1 改为 Y 后，向右移动，发现后面不再有 2，则 reject；

（4）如果期间遇到任何 021XYZ 之外的Σ-symbol，则 reject；

如上文所提到的，TM 判断一个 string 是否可计算，有三种结果。其中两种是 halt，一种是 not halt。而 halt 里面，又分为 accept 和 reject。如果一个语言 L，TM 对 L 有可能返回三种结果，则我们说 L 是 Turing-recognizable；TM 对 L 返回的只可能是 halt 的两种结果，则说 L 是 Turing-decidable。有没有语言是 Turing recognizable 但不是 Turing-Decidable？有，比如 first-order logic。有没有语言甚至不是 Turing recognizable？有，但例子不好举。

TM 是后面用于分析一个算法复杂度的默认使用的计算模型。比如给定一个语言，由无数个属于该语言的 strings 组成。比如 L = $\{0^n1^n2^n \mid n \geqslant 1\}$是 L 的内涵式定义，而 012、001122 这些是它的外延式定义。TM 接收某个 string 会有一个长度 n，则在该语言 decidable 的情况下，会判断 TM 需要多少时间和空间来确定 accept 还是 reject 该 string。请记住：某个语言要不是 decidable，要不是 undecidable。如果是 undecidable，则不再分析其复杂度，因为语言是不可决定的，计算可能没有终点，分析到可计算性这里就结束了（computational theory）；只有先确定是 decidable 的，才能进行复杂度分析（complexity theory）。

首先是时间复杂度分析，比如给定一个 string s 长度|s| = n，需要用多长时间 TM 才能判断 accept 还是 reject s？而且这个结果一定与 n 有关系，也就是说，我们会有一个函数，这个函数的 input 是 n，output 是一个与 n 有关的数字，比如 f(n)，我们需要 f(n)这么久的时间才能完成这个任务。我们可以得到一个非常精准的结果么？举个例子，判断 0011222 和判断 0101222 能否属于 L = $\{0^n1^n2^n \mid n \geqslant 1\}$。很明显，前者需要按照上述方法，知道最后一个 2 才可以判断不属于 L，而 0101222 到第二个 0 给出 reject 的结果。可见，同样是长度为 7 的 string，判断的时间也不一样，所以我们无法得到一个针对 input n 的精确结果，而是一个最坏结果。最差情况下，多久可以得到 accept 或者 reject 的结果？采用的是 worst-case analysis 的方式。另外，我们一直用 string 来作为 TM 的输入，用 string 的长度来作为时间复杂度分析的输入。其实很多情况下，我们的数据结构不一定是传统的 string，还可以是比如一个图，那么这时候，我们需要多少 steps 去完成一个任务，可能取决于这个图的节点数量，边的数量，或者这个图的 outdegree 的数量，又或者是上述这些参数的某些组合。

还有，当 n 是 string 的长度，我们得到一个针对 n 的时间复杂度 $f(n) = 6n^3 + 2n^2 + 20n + 45$，我们只考虑这个多项式的最高项（the highest order term），且系数（coefficient）也会被忽略掉。这是因为我们采取一种渐进式（asymptotic notation）的分析方法，称 f is asymptotically at most n^3，也叫 big-O notation，写作 $f(n) = O(n^3)$。这是因为，随着 n 的增大，f(n)的取值将最大程度，或者叫被 n^3 所 dominant。我们只抓重点，不看那些影响较小的部分。

我们用这种方法简单分析下 decides L = $\{0^n1^n \mid n \geqslant 1\}$的时间复杂度：
TM = “on input string w”：
（1）首先会读这个 tape，如果发现 0 出现在 1 的右边，则 reject；
这一步最坏的情况是直到最后一步，才发现 0 出现在 1 的右边，比如 0001110，所以最坏情况是对整个 string 进行遍历，需要 n steps，最后把 head 归位到最左边的 cell 又需要 n steps，一共需要 2n steps。因为采用 big-O notation 分析法，这一步需要 O(n) steps。凡是这种只影响系数的操作过程，我们都会忽略掉（omit）。
（2）再去读这个 tape，读到一个 0，换为 X，一直向右移动，读到一个 1，换为 Y，一直向左移动，遇到 X，向右移动，读到一个 0…repeat

这一步是遍历整个 tape，并且会替换掉一个 0 和一个 1，也就是一次 scan 替换 2 个元素，那么最多我们会 scan n/2 次，假设每次需要走 O(n) steps（事实是越到后面，走的 steps 越少，不需要走全部的 n steps，但是别忘了是 big-O notation 分析法），所以我们最多在这一步会走 $n/2 * O(n) = O(n) * O(n) = O(n^2)$。

（3）最后一步，查看当按照 0 与 1 一对一兑子之后，是否还剩下 0 或者是否还剩下 1，如果是，则 reject；如果 tape 上只有 X 和 Y，则 accept。

这一步又需要 scan 一次整个 tape，需要 O(n) steps。

所以完成整个 decide 任务一共需要 $O(n) + O(n^2) + O(n)$ steps，按照 big-O notation 分析法，decide 这个 L 的时间复杂度是 $O(n^2)$。

针对这个问题，有没有更好的算法使得更快地 decides L，答案是肯定的。你可以按照参考书 page 280 的算法得到一个 O(n logn)的 bound，甚至在 multi-tape TM 上得到一个 O(n)的结果。我们甚至得到更 general 的结论（Theorem 7.8）：

THEOREM 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

可以看到使用 multi-tape TM 比 single-tape TM 要在时间上，有优势，优势是从多项式时间到线性时间，对于人来说，够大么？够大。对于计算机呢？不算大。所以使用 multi-tape 还是 single-tape TM 并不影响大局。那么下面这个呢？

THEOREM 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

可见，一个 non-deterministic TM 有着至少一个指数量级的时间优势。计算机在意么？答案是在意。说明使用 non-deterministic TM 还是 deterministic TM 区别非常大，所以在做时间复杂度的时候，一定要说明使用的 TM 是哪一种，如果不说，默认为经典 TM。对于计算机来说，一个重要的界限就在于这里：多项式时间还是指数时间。如果证明了决定某个 L 最坏情况下需要 $O(n^3)$，还是 $O(n^6)$，其实本质上并无量级的差别，当你找到了一个多项式时间的 bound，就不需要去寻找一个 linear 的 bound，尽管 linear 的 bound 当然比 polynomial 的更紧，时间上更漂亮，但是对于真实计算机的计算速度来说区别不大。但是一旦突破了指数界限，那就不能任性了，如果你可以找到两倍指数的 bound，就千万别停留在三倍指数，就像你要是一个亿万富豪，可以肆无忌惮的消费；但是如果你是一个朝九晚五的打工仔，过日子就得精打细算，一个道理。如果你能为一个目前指数时间复杂度算法才能解决的问题找到一个多项式的算法，那绝对值得发表一篇漂亮的 paper。我们做这门课的时间复杂度分析，停留在这条界线上足够，不需要找到更精确的 bound。但是到了 exponential 的界限，就需要精打细算，到底是 $O(2^n)$ 还是 $O(2^{2^{n}})$ 区别很大。

对于一个计算问题来说目前已知的最好算法时间复杂度是 $O(n^3)$，则它是一个在 polynomial 时间内可以解决的问题，我们把这一类问题称为 P 问题，是计算问题里面最为简单的问题。

DEFINITION 7.12

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

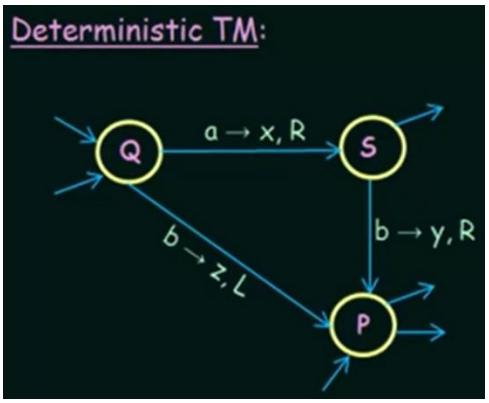
这里，polynomial time 指的是 the number of steps the algorithm has to perform to solve the problem is in the order of a polynomial in the input size。见上图，无论 k 赋值于几，本质上都是 P 问题，不会因为指数位置上数字变大，就会有什么不同。当然，也不能太过分，比如 n^{100} 结果大不大，当然大，但是实际生活中，什么时候才能遇到 for 循环 100 次的情况呢？我们把 P 问题称为 realistically solvable 的问题。

现实生活中典型的 P 问题有很多，包括 PATH 问题，RELPRIME 问题，另外，decides CFL 语言也属于 P 问题。

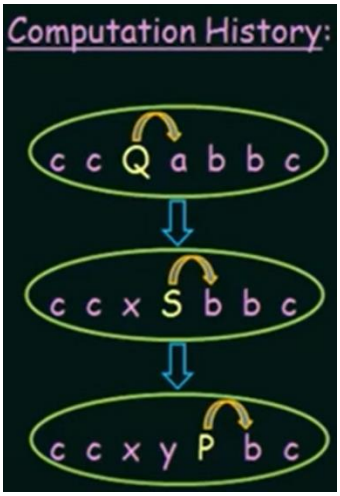
除了 P 问题，还有一种问题类型称为 NP 问题。这一类问题也比较好理解，它指的是能在 polynomial time 内由 non-deterministic TM 决定的问题。在我们的作业中，涉及到了关于使用 non-deterministic TM 的问题，这里我们详细讲一下它，与 deterministic TM 区别在于：

$$\delta\colon Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

Deterministic TM 的 transition function 可以去到(Q, Γ, {L, R})，而在 non-deterministic TM 里，变为了它的 powerset，P(Q, Γ, {L, R})。依然难以理解，看一个例子：

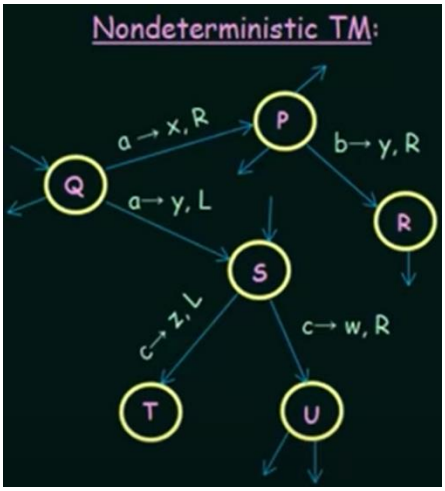


可以看到这是一个 deterministic TM 的片段，我们依然用传统的 state 呈现形式来代替 TM 的 tape，这样显得更加直观。将上述的 states Q S P 想象成 tape 的 cell:

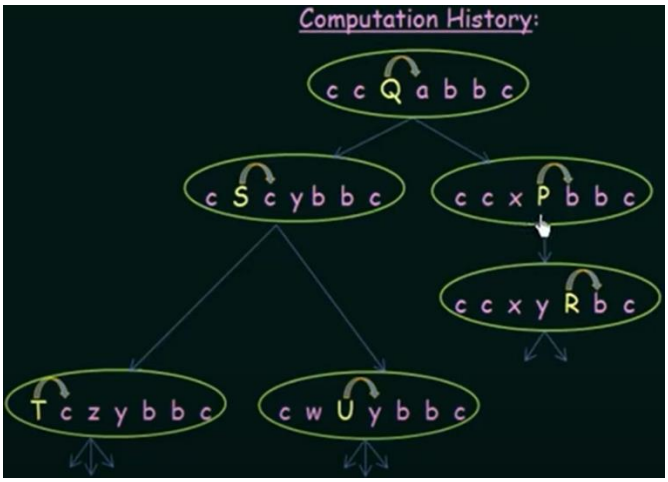


这是其中的计算过程，每个椭圆成为一个 configuration，每个 configuration 记录了当前状态，当前 tape 的内容，以及当前的 tape head 在哪里。具体表示为 substring1 Q substring2，其中 Q 表示当前所处的状态，substring1 和 substring2 表示 Q 状态前的 string 和 Q 状态后 string，而 Q 状态后的 substring2 的第一个 symbol 就是 head 现在读到的 symbol，比如 abcQedf 表示

当前状态为 Q，之前已经读过了 abc，目前 head 读到了 e。按照这样的解释，上图表示的计算过程就是目前来到了 Q 状态，读到了 symbol a，并将 a 这个 symbol 改写为 x，转移到下一个状态 S，在 S 状态读到了 symbol b，并将 b 改写为 y，转移到状态 P，在 P 状态读到了 symbol b。在这样一段过程中，deterministic TM 无论来到哪一个状态，在这个状态读到某一个 symbol 之后都只能有唯一的下一个“configuration”：（1）将当前状态改写为什么 symbol；（2）走向哪一个状态；（3）接下来去往哪个方向，左还是右？而在一个 non-deterministic 中情况不一样，当来到某个状态，在这个状态上读到某个 symbol，可能面临多种选择：at each moment in the computation there can be more than one successor configuration。



比如上图，在 Q 状态读到 symbol a，可以将当前 symbol a 改写为 x，去往 P 状态，并在 P 状态上向右移动；还可以将当前 symbol a 改写为 y，去往 S 状态，并在 S 状态上向左移动。相当于在每一步给了我们更多选择的自由，计算的效率增高了（计算过程中走的 steps 整体上变少了）。如果将这个计算过程用和刚才一样的 configuration 的形式表示出来：



可以看出，形状变为了树状结构。按照其结构的特点，如果任何一个 branch 达到了 accept 状态，则 non-deterministic TM accepts the input string；否则当所有的 branches 都 reject，则该 TM rejects the input string。很明显，如果要“验证”（verify）某个 branch 是否被该 TM accepts 是不是就是一个 deterministic 的路径？所以可以在 polynomial time 内被一个 deterministic TM 所解决，但只是“验证”，不是“解决”。一个典型的问题是 traveling salesman problem，it is possible for a postman to visit all cities while traveling at most k distance。可否在 polynomial time 找出所有这样的路径？不知道。但是很明显，如果给定一条路径，去验证这条路径是否满足 distance 的要求可以在 polynomial time 解决。这就是注明的 P＝NP？问题：一个 polynomial

time 内可验证的问题是否也可以在 polynomial time 内可解决。

Non-deterministic TM 并没有比 deterministic TM 增加了计算能力：

THEOREM 3.16

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

由此可以得出推论，一个语言 L 是 Turing-recognizable iff 有一些 non-deterministic TM 可以 recognizes L；一个语言 L 是 Turing-decidable iff 有一些 non-deterministic TM 可以 decides L。

根据上述 Theorem 7.11，我们知道一个到目前为止确定的结论：一台 non-deterministic TM N 可以 simulated by 另一台 deterministic TM T。并且存在一个 constant c，使得，对于任意 input string s，如果 N 可以在 t steps 内 accepts s，则 T 最多可以在 c^t steps 内 accepts s。下面是分析的过程，最坏情况下，N 可能会不得不在任何一个状态上进行 non-deterministic 的选择，这就会产生一个 computation tree，像是一个完整的 binary tree of depth t(n)，其中 n 的 s 的长度。这个 tree 会产生 $2^{t(n)}$个不同的 branches，每一个 branch 都是一个 valid computation of N。那么用 T 去 simulate 整个计算过程需要 depth * |branches|：

$$t(n) * 2^{t(n)} = 2^{t(n) + \log 2t(n)} \leqslant c^{t(n)}$$

for some sufficiently large constant c > 2。没有人知道是否有更好的 T simulates N 的办法，让这个计算过程更加简单，steps 更少！现在关于复杂度类之间最好的结论是：

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k}),$$

NP 问题是可以被一个 deterministic TM 在 EXPTIME 内解决的。NP 与 EXPTIME 之间有没有其它的类可以插入进去？不知道！如果知道，可以申请 100 万美元奖金。

接下来我们引入 NP-complete 和 NP-hard 这两个概念，来帮助我们更好的划分复杂度的类。除了 P 问题和 NP 问题，我们还有更多的类。这些类依然是围绕 NP 提问展开的。上世界 70 年代，一位美国科学家 Stephen Cook（1982 年图灵奖得主）和一位俄裔美国科学家 Leonid Levin（Donald E. Knuth Prize 得主），他们发现 NP 问题里面有一类子问题，它们的复杂度与整个 NP 问题类的复杂度有非常大的关系。换句话说，NP 问题里面的各个问题之间可能会存在某种联系。什么联系呢？这一类子问题非常有“代表性”，就是如果这些子问题能被一个 deterministic TM 在 polynomial time 内解决，则 NP 的其它问题也一定能被 deterministic TM 在 polynomial time 内解决。换句话说，这些子问题是 NP 问题里面最难的问题，称为 NP-complete 问题。他们为什么要去找到这样一类子问题呢？试想一下，如果此时我想证明 NP 问题都可以被 deterministic TM 在 polynomial time 内所解决，是不是需要证明每一个 NP 问题都能被 deterministic TM 在 polynomial time 所解决。问题是 NP 问题是无限的，这意味着这个证明永远都做不出来。因为理论上我们无法总结出所有的 NP 问题，并去一一证明。但是如果又这么一个 NP 问题，解决了它，则其它所有的 NP 问题都可以解决了，是不是就说明这个问题是“有代表性的”？第一个被证明是 NP-complete 的问题，也是被上述两位学者独立证明，是 Boolean satisfiability problem (简称 propositional logic 上的 SAT 问题)：

SAT: determine if a propositional formula can be made true by an appropriate assignment of truth values to its variables.

显然，如果验证一个 assignment 是否让某个命题逻辑表达式为真，相对简单；但是要直接

判断该表达式是否可满足（是否存在某个 assignment 使得其为真），用的最好的方法就是对所有可能的情况进行穷举（最坏情况下），查看每一种可能的取值。假如这个表达式包含 n 的 proposition symbol，则最多有 2^n 种 truth value 的可能性。这意味着只要为 SAT 问题找到一个deterministic TM上的polynomial time的算法，那么就等于证明了P＝NP。可以总结为：

SAT belongs to P iff P = NP

Donald E. Knuth 教授大家都认识吧？TEX 打字系统的发明者，算法分析方面的引领者，著有《The Art of Computer Programming》一书，可以说在我们上学那个年代，无人不知，无人不晓。不知道现在 Python 流行的年代还是否这样。当时编程的两本启蒙教材，不是学校里面发的教科书（学校里面的教科书差点给我学抑郁了），而是《Thinking in Java》和《The Art of Computer Programming》。以他名字命名的 Donald E. Knuth Prize 主要是奖励在理论计算机方面的杰出个人，与哥德尔奖奖励的是杰出的工作（某一篇 paper）不同。1996 年设立该奖项后的第一位得主是姚期智教授，而最近的 2021 年得主是美国莱斯大学的 Moshe Vardi 教授，他是逻辑和知识表示与推理方面的顶级学者，著有《Reasoning About Knowledge》和《Finite Model Theory and Its Applications》，其中很多内容都是我们上课学到的，也是我博士导师在逻辑方面的领路人之一。看一下图灵奖得主 Stephen Cook 教授（照片来自维基百科，公开授权）：

年纪轻轻就做出了数学上非常漂亮的，计算理论底层的开拓性贡献。

接下来讲一个概念叫 reducibility，它的思想基础是：如果一个问题A可以reduced to问题B，则问题 B 的解也可以拿来解决问题 A。比如我想考上南京大学，需要高考考 660+，则考上南京大学的问题（问题 A）就被 reduced to 高考考 660+的问题（问题 B），那么只需要去寻找高考考 660+的方法即可。解决了问题B，A 也就解决了。但是这个 reduced 的过程不都是计算上非常简单，我们定义一种转换，叫做 polynomial time reduction：

DEFINITION 7.28

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

首先定义一个转化函数，称为 polynomial time 可计算的函数，然后：

DEFINITION 7.29

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_P B$, if a polynomial time computable function $f: \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B.

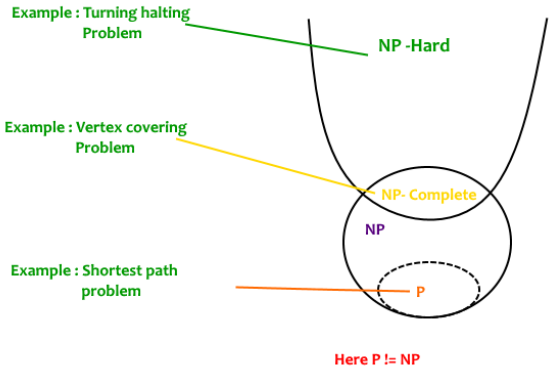
一个语言 A 到语言 B 之间存在一个 polynomial time 的转换函数 f，意味着对 A 中任意 string，都存在一个 f(w)属于 B。这个 f 就是 A 到 B 的 polynomial time reduction。至此，我们得到一个新的结论：

THEOREM **7.31**

If $A \leq_P B$ and $B \in P$, then $A \in P$.

如果 AB 之间存在一个这样的 f，且 B 属于 P 问题类，则 A 也属于 P 问题类。

还有一类问题是 NP-hard 问题，它指的是任何 NP 问题都可以在 polynomial time 内转化为该类问题，则这类问题被称为 NP-hard 问题；如果刚刚好，这类问题也在 NP 类里面，则称为 NP-complete 问题。所以 NP-complete 问题是 NP-hard 问题的子集，同时也是 NP 问题的子集，用图表示的话如下：



想要证明一个问题 A 是 NP-hard，只需要找到一个已知的 NP-hard 问题 B，再证明存在一个 polynomial time reduction 使得 B 可以在 polynomial time 内 reduced 为 A：$A \leqslant_P B$。
想要证明一个问题 A 是 NP-complete，首先要证明问题 A 在 np 中（给定一个 solution，证明可以在 polynomial time 进行验证），其次需要找到一个已知的 NP-complete 问题 B，证明存在一个 polynomial time reduction 使得 B 可以在 polynomial time 内 reduced 为 A：$A \leqslant_P B$。

上面是最简单的证明方法；否则按照定义，上述二者都需要我们去证明任何 NP 问题都可以在 polynomial time 内 reduced 为该问题，这一步我们很难做到。所以我们只能借助现有的东西，去做这个证明。正所谓站在爸爸的肩膀上，嘲笑爸爸矮。而如果想证明某个问题 A 属于 P 问题或者 NP 问题，只需要找到对应的 TM recognizes it 即可。

除了关注在时间复杂度，还有一个评价计算复杂度的方法就是用了多少的空间。这里面，很显然对于同一个问题，空间复杂度不会超过时间复杂度，比如完成一个任务需要 n steps，那么不可能完成这个任务需要大于 n 个 states（cells）；相反如果完成一个任务需要 n states，倒是有可能需要超过 n states 的数量，比如需要 n^3 steps 可能吗？可能。但是也不是无限的，没有边界，我们得到的结论是：

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME.$$

如果解决一个问题需要 TM 占用 tape cells 的长度是 f(n)，确实 TM 可以走超过 f(n)的步骤来完成整个计算步骤，但是它最多可以有 $f(n) * 2^{O(f(n))}$个不同的 configurations，证明请看计算理论参考书 Lemma 5.8 或者任意其它资料（建议阅读，因为重要）。

另外，在这里还要介绍一个重要的定义，关于上述 inclusions 排列中的 PSPACE = NPSPACE 这一结论。我们知道 PTIME = NPTIME 是百年悬而未决的问题，但是空间上早已被证明是

等价的，证明者为 Stephen Cook 教授的学生，Walter John Savitch。他已经于 2021 年去世，去世之前是美国加州大学，圣地亚哥分校的教授。这个结论叫做 Savitch's theorem。我们先用自然语言描述下这个定理：deterministic TM 可以使用非常小的增加的空间代价去 simualte non-deterministic TM。如果你还记得上面的定理，目前我们知道，deterministic TM 需要指数时间才可以 simulate non-deterministic TM 的计算行为，但是在空间上的结果却非常令人惊讶！具体说，就是 Savitch's theorem 证明了任意使用 f(n) space 的 non-deterministic TM 都可以等价转化为使用 f^2(n) space 的 deterministic TM，数学表达式为：

THEOREM 8.5

Savitch's theorem For any[1] function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

又因为：the square of any polynomial is still a polynomial，我们完全可以安全地认为 PSPACE 类和 NPSPACE 是同一个类。所以在证明中，如果你使用了 non-deterministic TM 证明了某个问题是 f(n) space 可解的，换句话说是 NPSPACE 的，那么就等于证明了它是 PSPACE 可解的。

在空间上所有的 completeness 和 hardness 概念，以及其证明与时间上的一模一样。