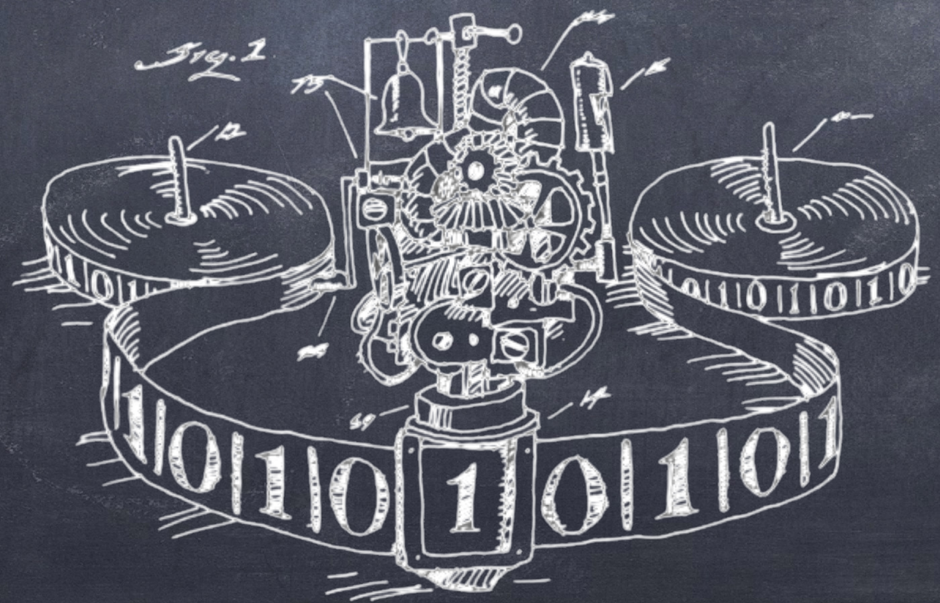
可计算性



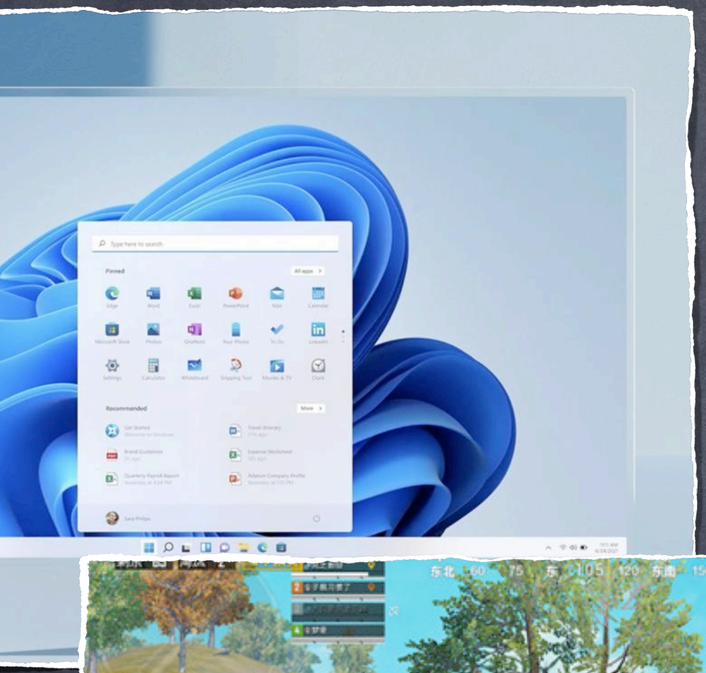
预警：这部分的内容可能有一点点难

大纲

- 一个问题
- 可计算性起源
- 图灵机
- 图灵机的一些性质

编程可以做到任何事吗？

似乎无所不能？



NX 编程可以做到任何事情吗?

编程是一种强大的工具，可以用来解决许多问题和实现各种任务，但并不是所有事情都可以通过编程来实现。

编程可以通过计算机来自动化许多任务，例如处理数据、执行复杂的算法、创建图形界面、编写网站和应用程序等。编程语言的能力范围也不断扩展，现在可以编写人工智能、机器学习、大数据处理等复杂的应用程序。

让我们看一个Hello, world!的例子

Hello, World!

- 写一段C语言的代码，接受一个输入，打印"Hello, world!"到屏幕上，然后结束。
  - 不考虑空间和时间的限制：这个程序在一个"理想"的计算机上。
- 非常简单，对吧？

判定Hello World!

- 我们现在的目标是：写一段判定程序G，判读任何一个给定的C语言代码P，为其输入I，是否实现了上述的打印功能。

$$G(P, I) = \begin{cases} Pass, & \text{如果P在I输入下打印“hello, world!”并停止} \\ Fail, & \text{否则} \end{cases}$$

我们能够写出G吗？

判定Hello, world!

```
#include <stdio.h>


int main(int argc, char *argv[])
{
    printf(“Hello, World!”);
    return 0;
}
```

```
#include <stdio.h>


int main(int argc, char *argv[])
{
    printf(“XXX!”);
    return 0;
}
```

判定Hello, world!

```
#include <stdio.h>


int main(int argc, char *argv[])
{
    if (argv[1][0] == ‘y’){
      printf(“Hello, World!”);
    }
    return 0;
}
```

```
#include <stdio.h>


int main(int argc, char *argv[])
{
    while(argv[1][0] == ‘y’);
    printf(“Hello, World!”);
    return 0;
}
```

判定Hello, world!

```
#include <stdio.h>
main(int argc, char *argv[]){
  long long P = 1,
            E = 2,
            T = 5,
            A = 61,
            L = 251,
            N = 3659,
            R = 271173410,
            G = 1479296389,
            x[] = { G * R * E * E * T , P * L * A * N * E * T };
  puts((char*)x);
}
```

判定Hello, world!

```
#include <stdio.h>
int i[(1+1)+1],*I=i;main(int argc, char *argv[]){*I+
+=(((1<<((1<<(1+1))*(1<<(1+1))))+((1<<((((1<<(1<<
(1+1)))-1)^(1<<1+1))-1))-(1<<1)-1+((1<<1)*(1<<1)*(1<<1)*(111-(((1<<(1<<(1+1)))-
1)^(1<<1+1))))))*(((((1<<(1+1))*(1<<(1+1)))*(1<<1)-((1<<(1+1))|1))*(((((1<<(1<<
(1+1)))-1)^(1<<1+1))*((((1<<(1<<(1+1)))-1)^(1<<1+1))-1)*((((1<<(1<<(1+1)))-1)^(
1<<1+1))-1)+(((1<<(1<<(1+1)))-1)^(1<<1+1)))-((((1<<(1<<(1+1)))-1)^(1<<1+1))*(((
(1<<(1<<(1+1)))-1)^(1<<1+1))-1)+1)))+((1<<1)|(1<<(1+1))))+(1<<1));*I++=i[1-(1<<
(1-1))]+(((1<<((((1<<(1<<(1+1)))-1)^(1<<1+1))+(1<<1)))+((1<<(((1<<(1<<(1+1)))-1
)^(1<<1+1)))-(1<<(1+1))*(1<<(1<<1))*((1<<(1<<1))-1)-(1<<1)-1))*(((1<<2)*(1<<2)*
((1<<(1+(1<<(1-1))))-1))*((((1<<(1<<(1+1)))-1)^(1<<1+1))-1)*((((1<<(1<<(1+1)))-
1)^(1<<1+1))-1)+(1<<1)+1));*I++=(((1<<(1<<(1<<(1<<1))))+(1<<(1<<((1<<1)+1)))+((
(((1<<(1<<(1+1)))-1)^(1<<1+1))-1)*(1<<1)+1))*((((1<<(1<<(1+1)))-1)^(1<<1+1))-1)
*((((1<<(1<<(1+1)))-1)^(1<<1<<1))-1)+(1<<((1<<1)*((1<<1)+1)))-(1<<1));puts(i);}
```

obfuscated code!

要不让程序运行起来，再看结果不就行了？

Collatz 猜想（3x+1猜想，冰雹猜想）

- 给定n，如果其为偶数，除以2，如果其是奇数，乘以3并加1，持续这个过程，那么最终会变为1。

```
int collatz(int x){
  while (x != 1) {
    if (x % 2 == 0) {//if even
       x = x / 2;
    } else { //if odd
       x = x * 3 + 1;
    }
  }
  return x;
}
```

Paul Erdős

Mathematics is not yet ready for such problems

那么这段代码怎么办？

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[]){
  int n = atoi(argv[1]);
  collatz(n);
  printf("Hello, World!");
}
```

我们实际上无法写出G

下面我们将给出一个informal的论证

无法写出的程序

假设存在

- 我们假设存在这样的一个程序G，其以任何给定的C程序代码文本p，以及对其任意的输入字符串i，为输入，判断其是否能出输出“Hello, world!”。
  - 程序中有这样一个函数 int G(char* p, char* i)，能够输出“Hello, World!”，返回1，否则返回0。

无法写出的程序

构造对抗程序

- 先构造如下程序P：

```
#include <stdio.h>
int main(int argc, char *argv[]){
    if (G(argv[1], argv[1]) == 0) {
        printf(“Hello, world!”);
    }
    printf(“We are sorry!”);
}
```

无法写出的程序

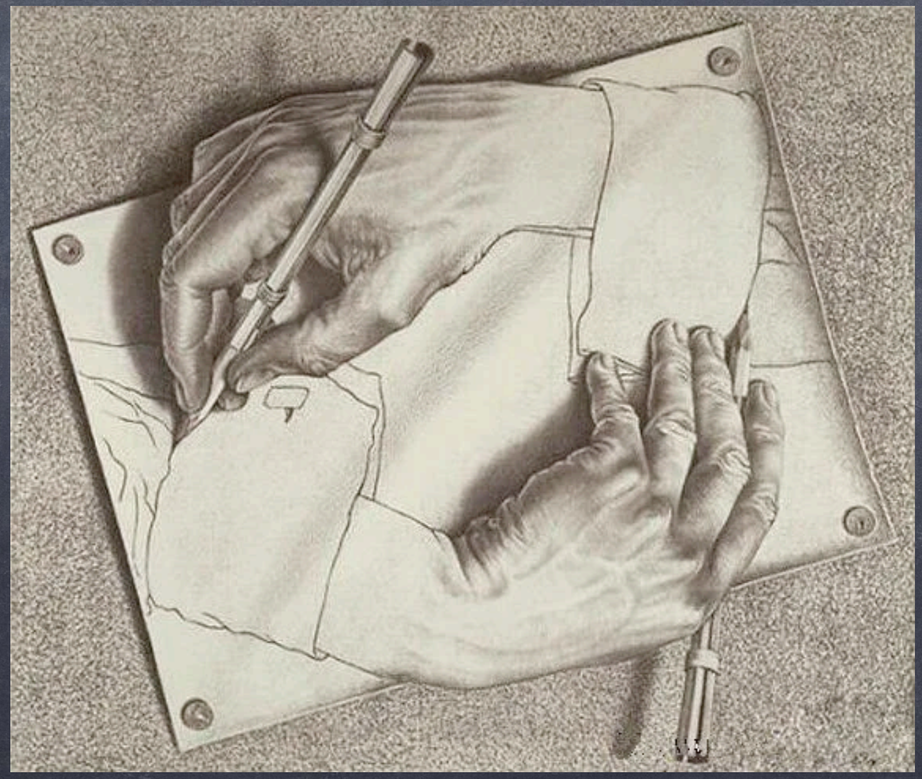
验证P

P

```
#include <stdio.h>
int main(int argc, char *argv[]){
    if (G(argv[1], argv[1]) == 0) {
        printf("Hello, world!");
    }
    printf("We are sorry!");
}
```

- 考察G(P, P)
  - 如果G(P, P)输出为1，其意味着P在输入以P为字符串输入下应该输出“Hello, world!”，然而在这种情况下，P其实输出的是“We are sorry!”。
  - 如果G(P, P)输出为0，其意味着P在输入以P为字符串输入不能输出“Hello, world!”，然而在这种情况下，P其实输出的是“Hello, world!”。

无论哪种情况都矛盾，因此，我们实际上无法写出一个这样的G



莫里茨·科内利斯·埃舍尔的《画手》

判断“Hello, world!”不是唯一不可写的程序！

停机问题（Halting problem）

- 给定任意程序P，和为其输入的I，
- 如果P能够在I的输入下停止，判定其输出为true.
- 如果P在I的输入下无法停止，判定输出为false.

再做一遍刚才的过程？

我们有更简单的方法：归约(Reduction)

归约

- 基本思想：
- 假定现在需要判定某个问题B：
- 只需要证明如果有一个程序可以判定B，那么一定可以通过某种方法构造另一个方法判定“Hello, world!”问题。
  - 即“B”可判定 → “Hello, world!”可判定
- 其逆否命题
  - “Hello, world”问题不可判定 → B 不可判定
- 因此，B不可判定

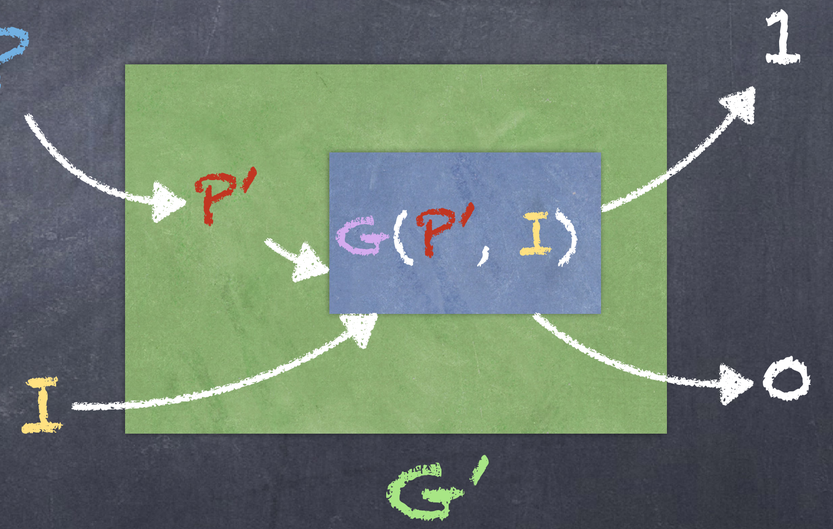
归约停机问题

1. 假定存在程序G可以判定停机问题，即
   - 对于任意程序P，任意输入I，
     - 如果P在I下可以停机，G(P, I) 输出1.
     - 否则P在I下不可停机，G(P, I) 输出0.

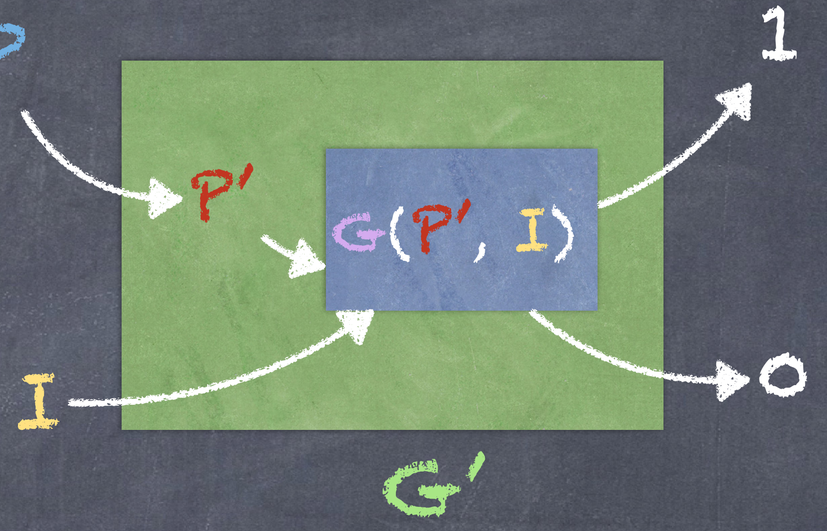
归约停机问题

2. 定义程序G′，其对于任意给定程序P和输入I做如下操作

- 创建程序P′，其是在程序P的基础上
  - 修改输出语句（如printf，或者puts等），使其在打印前
  - 先将要打印的字符记录在数组A中，当前12个字符不匹配
  - “Hello, world!”，加一句while (true);
  - 如果超过12个字符，加一句while (true);
- 运行程序G(P′, I)，G为可以判定停机问题的程序



归约停机问题



3. 根据假定，G(P', I) 要么为0要么为1

- 如果G(P', I)输出为1，即G程序告诉P'会在输入I下停机，那么意味着P在输入I下一定会最终输出"Hello, world!"并且终止，此时G'输出为1（其和G的输出一致）.
- 如果G(P', I)输出为0，即G程序告诉P'会在输入I下不停机，那么意味着P在输入I下一定不会最终输出"Hello, world!"并终止，此时G'输出为0
- 因此，G'可以判定任意程序P在任意输入I是否输出"Hello, world!"并终止。

归约停机问题

- 上述转换实际上让我们通过一个假定存在的可以判定停机问题的程序G构造出了可以判定是否输出"Hello, world!"并终止的程序G'
- 我们知道G'不存在，所以假定不成立，即G不存在！
- 因此，停机问题不可判定！

停机问题（Halting problem）是更加基础的问题。因此，一般证明不可判定性都会把问题转化为停机问题。

有关于什么可以被编程解决的问题叫做可计算理论（Computability Theory），是计算机科学的核心之一

一切的起源

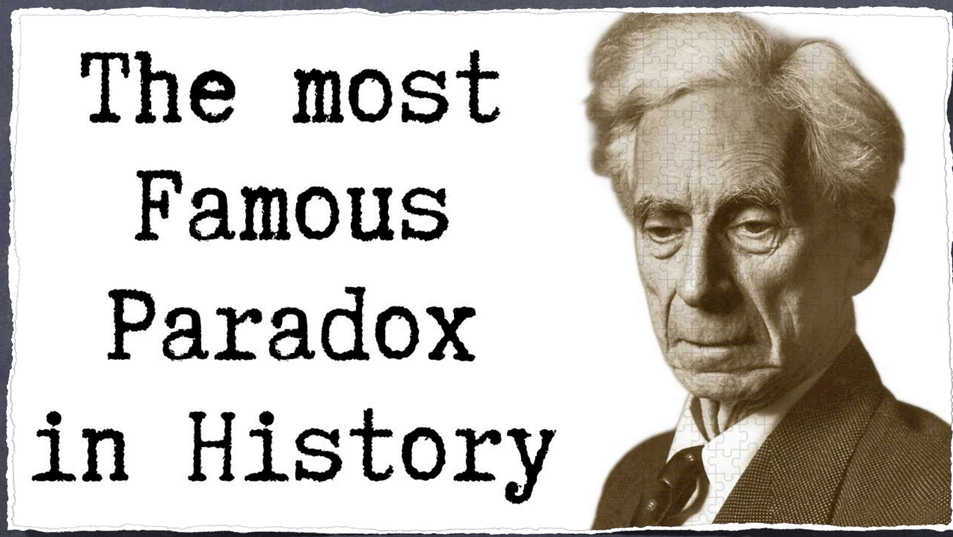
希尔伯特计划（Hilbert's Program）

- 大卫.希尔伯特："我们必须知道，我们终将知道！"

"Wir müssen wissen.
Wir werden wissen."

希尔伯特计划（Hilbert's Program ）

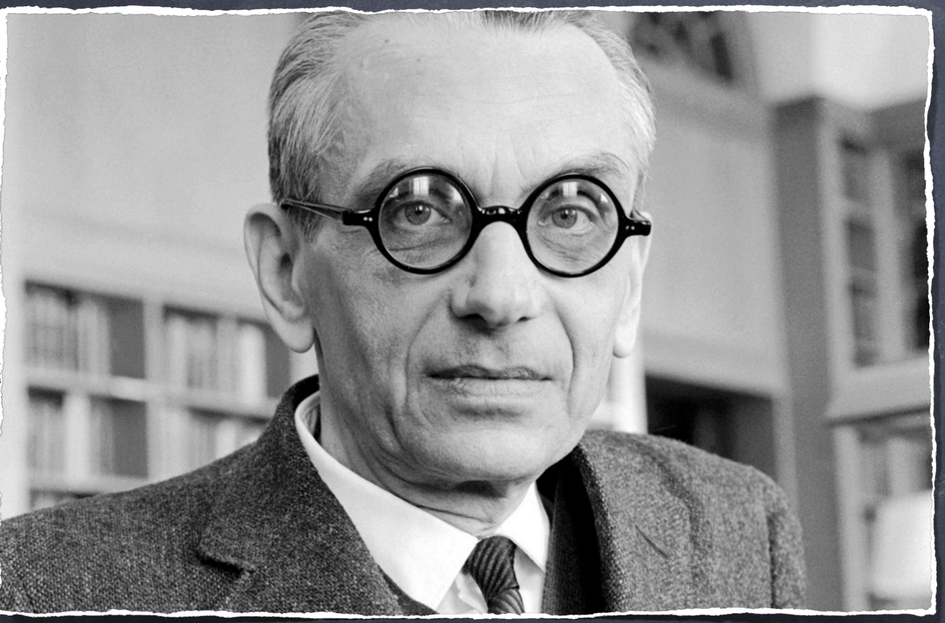
- 20世纪初，为了解决悖论尤其是罗素悖论所产生的第三次数学危机（The foundational crisis of mathematics），希尔伯特在1920年提出了一个计划，希望可以给"数学"提供一个坚实的保障！



希尔伯特计划（Hilbert's Program ）

- 该计划希望可以给所有数学提供一个形式化系统，该系统具备如下几个关键性质：
  - 完备性（completeness）：任何在该系统中的真命题，都可以得到证明。
  - 一致性（Consistency）：不可能推导出矛盾。
  - 可判定性（Decidability）：存在一个算法，来确定每一个形式化的命题是真命题还是假命题

哥德尔（Gödel）不完备定理



第一定理

"任何相容（一致的）的形式系统，只要蕴涵皮亚诺算术公理，就可以在其中构造在体系中既不能证明也不能否证的命题，即体系是不完备的。

哥德尔不完备定理

第二定理

"任何完备的形式系统，只要蕴涵皮亚诺算术公理，它就不能用于证明它本身的相容性。"

哥德尔不完备定理

- 哥德尔的两条不完备性定理直接宣告了希尔伯特计划的落空，因为其表明一个含有自然数公理的形式系统不能同时具备完备性和一致性。

希尔伯特的落日

It was over

哥德尔不完备定理

什么是可证？

- 在证明中，哥德尔构造了这样的一个命题："我无法被该形式系统证明"。
  - 如果你证明了这个命题，那么这个命题的内容便是不对的，或者说该命题为假。于是系统是有矛盾的。
  - 如果这个命题为真，根据它的内容，你无法证明它。

那么第三个问题呢？

- 由于当时还没有"算法"的明确定义，该问题在哥德尔证出不完备定理时，没有立即得到解答。
- 我们称该问题为 Entscheidungsproblem（decision problem 的德文，中文即判定问题）

图灵机的介绍

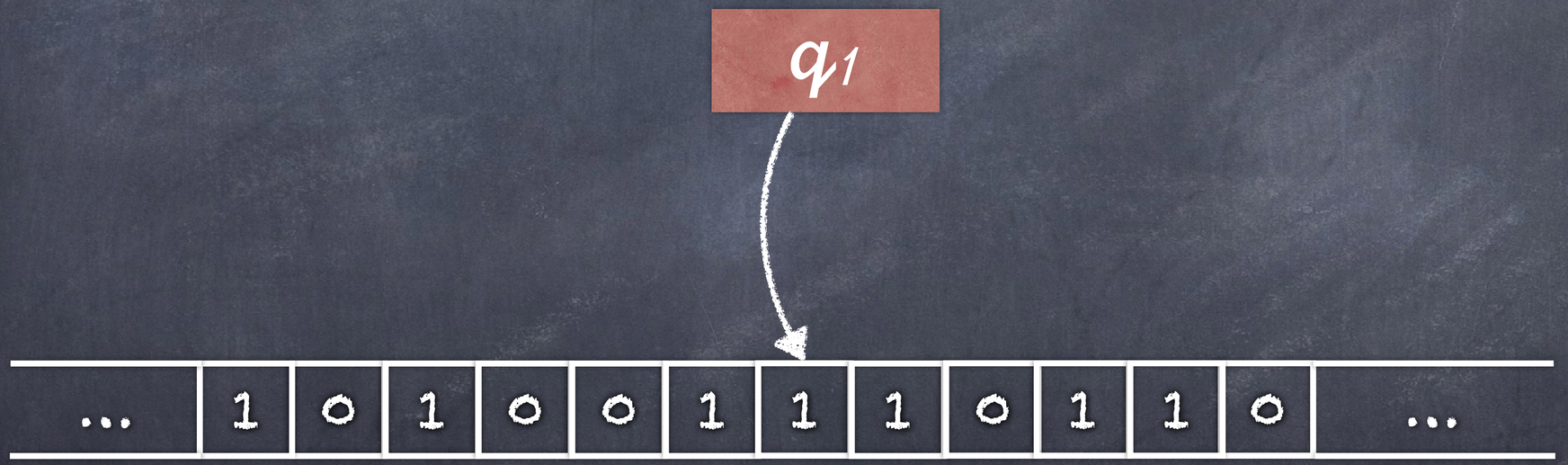
来自英国的天才:阿兰.图灵(Alan Turing)

- 到1935年夏天，躺在草地上休息的阿兰.图灵经历了一场头脑风暴，他想到了否定希尔伯特第三个问题的办法：用机器。

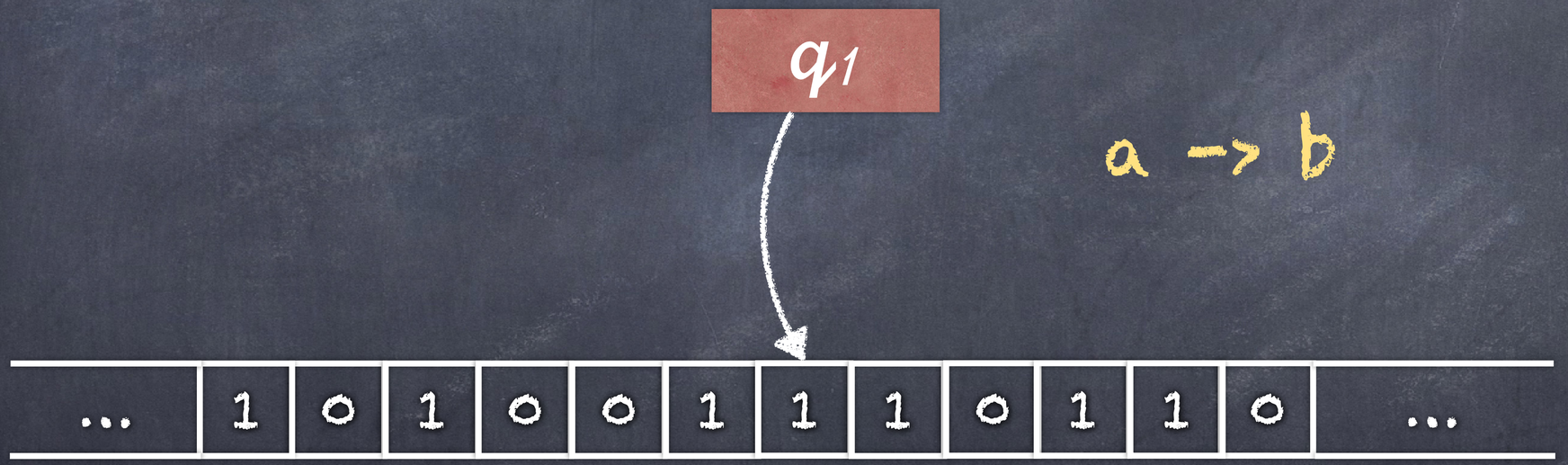
来自英国的天才:阿兰.图灵(Alan Turing)

- 他想象着一种虚构的机器，可以从一条无限长的纸带子上的读取命令进行操作，从而模拟人类所可能进行的任何计算过程。
- 图灵证明，我们不能用一个算法来判定一台给定的图灵机是否会停机，所以停机问题是一个无法判定的数学问题，即希尔伯特的第三个命题答案为否。

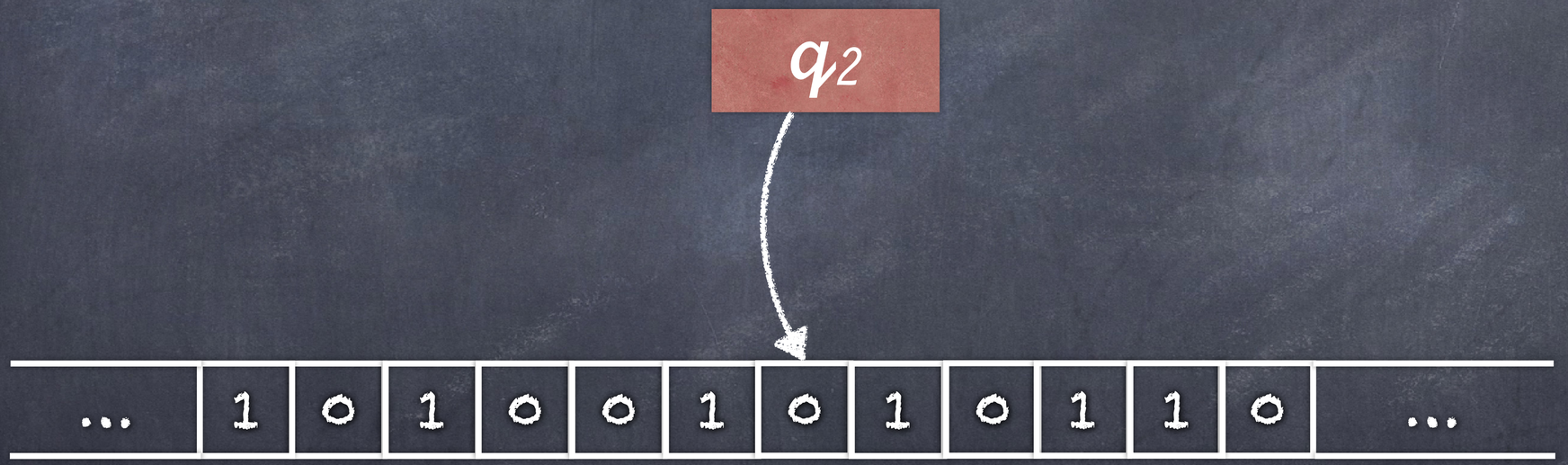
图灵机（Turing machine）



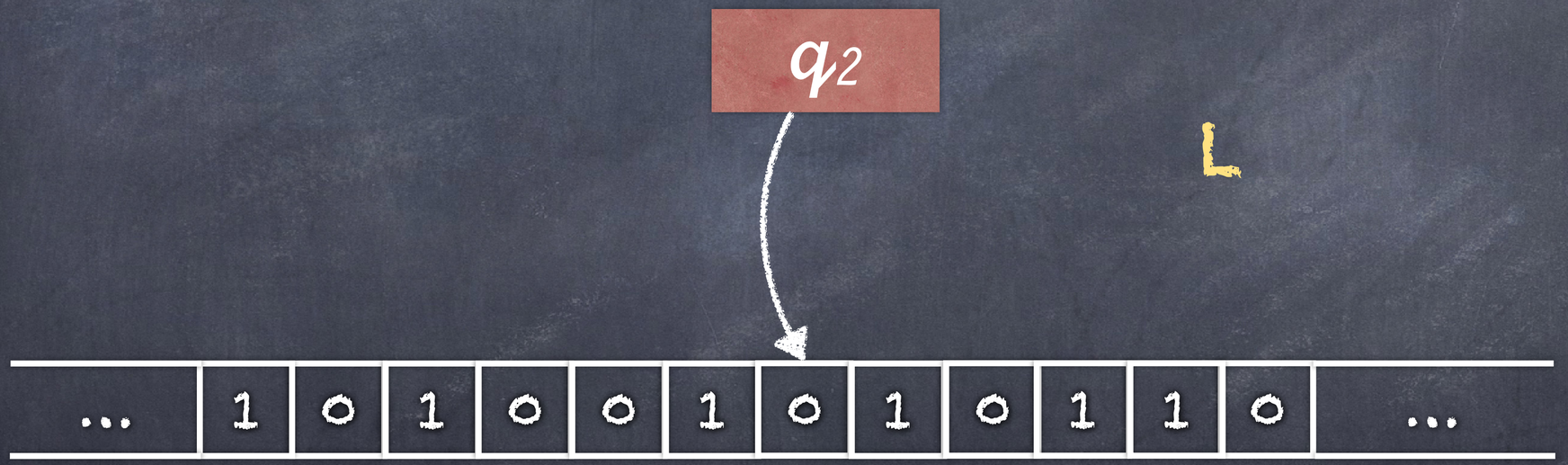
图灵机（Turing machine）



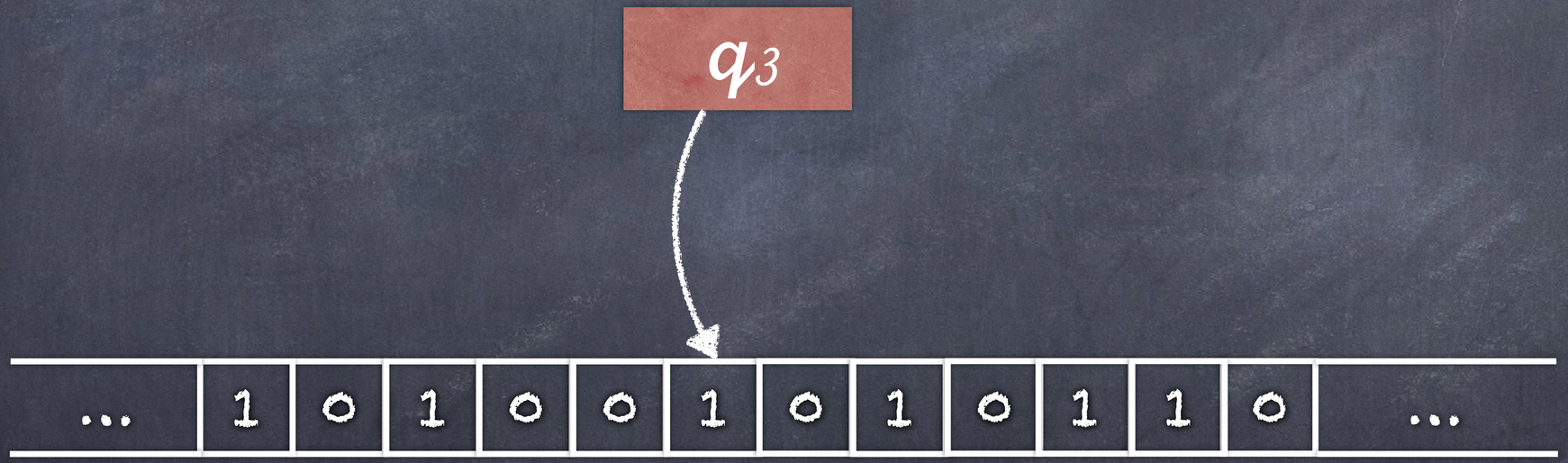
图灵机（Turing machine）



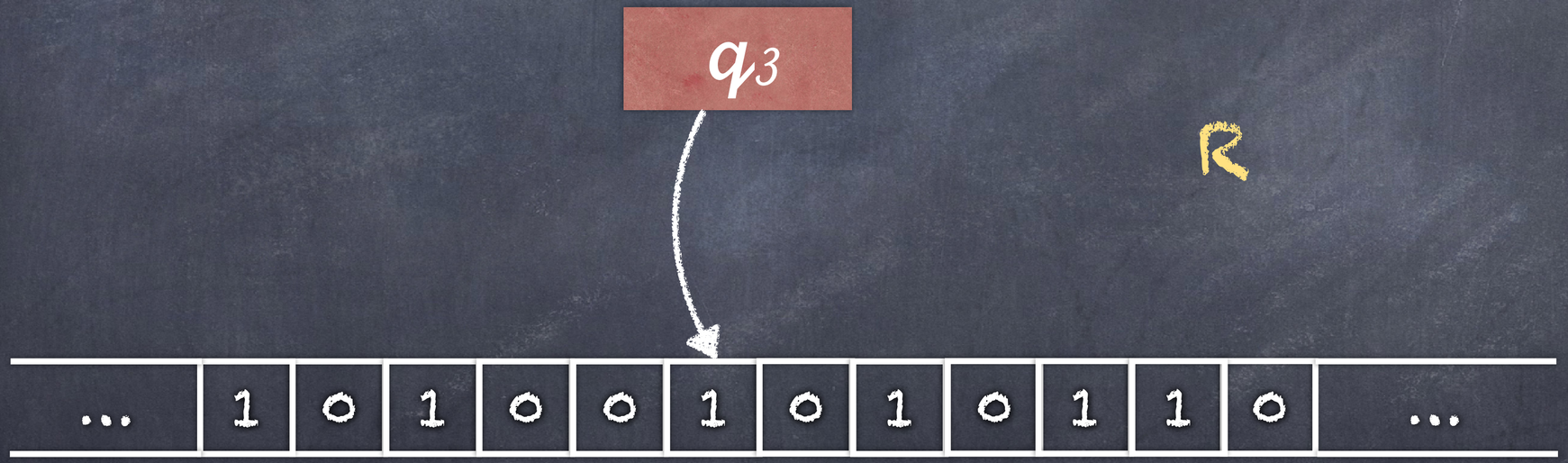
图灵机（Turing machine）



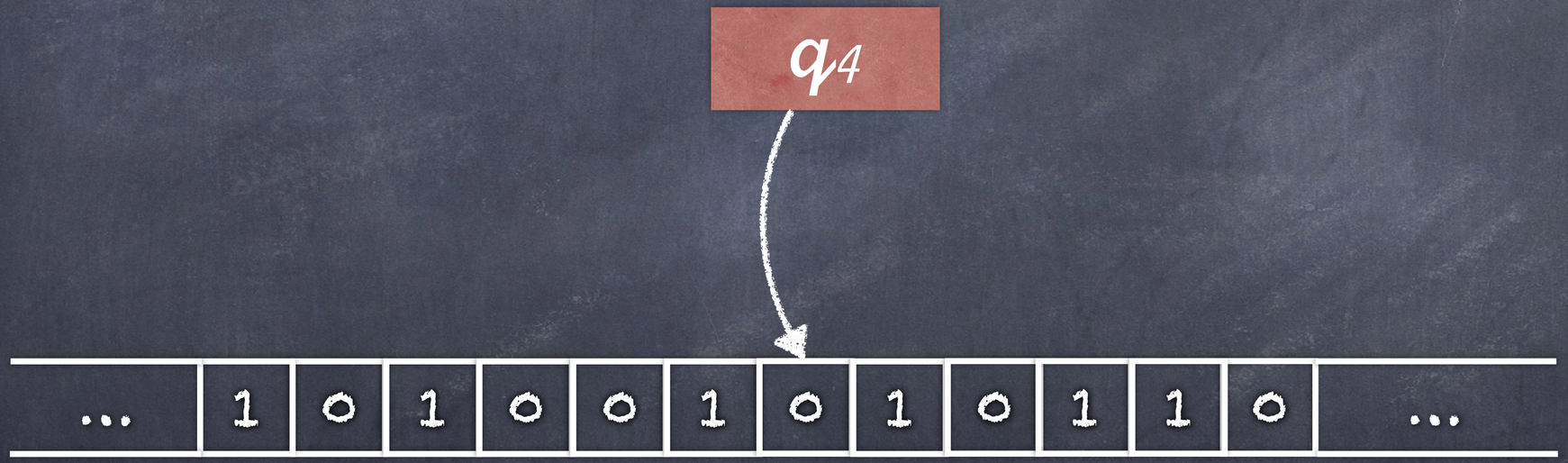
图灵机（Turing machine）



图灵机（Turing machine）



图灵机（Turing machine）



图灵机

- (1)一条无限延长的纸带，包含一个个小格子。这些格子可以从一个有限字符集Σ中选值填上。一般最简单就是$\Sigma = \{0,1\}$。其中0表示空白。
- (2)一个可以左右移动的读写头，每次扫描纸带上一个格子，可以读取格中符号，也能将其抹去填上新的符号。

图灵机

- (3)一个有穷状态集$Q=\{q_0,q_1,\ldots,q_n\},n\geq 1$在任何时刻下，图灵机总处于$Q$中的一个状态。
- (4)一个有穷指令集$\Delta=\{\delta_0,\delta_1,\ldots,\delta_m\},m\geq 0$，所谓一个指令$\delta_k\in\Delta$，其根据当前状态$q_i\in Q$和当前方格读到的符号$a\in\Sigma$，完成一个操作$O$，并进入下一个状态$q_j\in Q$。

图灵机

- (5) 操作$O \in \{L, R, b\}$，其中$b \in \Sigma$，如果$O = L$代表读写头左移，$O = R$代表右移，而$O = b$代表将现有格子中的符号a改为b。

图灵机

- 所以指令集Δ实际上是一个映射：
  - $Q \times \Sigma \to (\{L, R\} \cup \Sigma) \times Q$
- 其元素可以用四元组表达(q_i, a, O, q_j)
- 指令是确定的，即如果同一个图灵机有两个指令(q_i, a, O, q_j)和$(q_i, a, O^{'}, q_j^{'})$，那么$O = O^{'}$, $q_j = q_j^{'}$

图灵机

- 除自身外的信息：
  - 初始状态q_s和停机状态q_h
  - 输入：输入是一个1个或n个自然数，一个自然数m用$m+1$个1表达，多个自然数用0隔开。
  - 输出：停机时1的总数。

除了正常的q_h停机外，如果此刻状态和当前符号对无匹配任何指令，也停止（异常或错误）。注意：死循环不是停机

图灵可计算

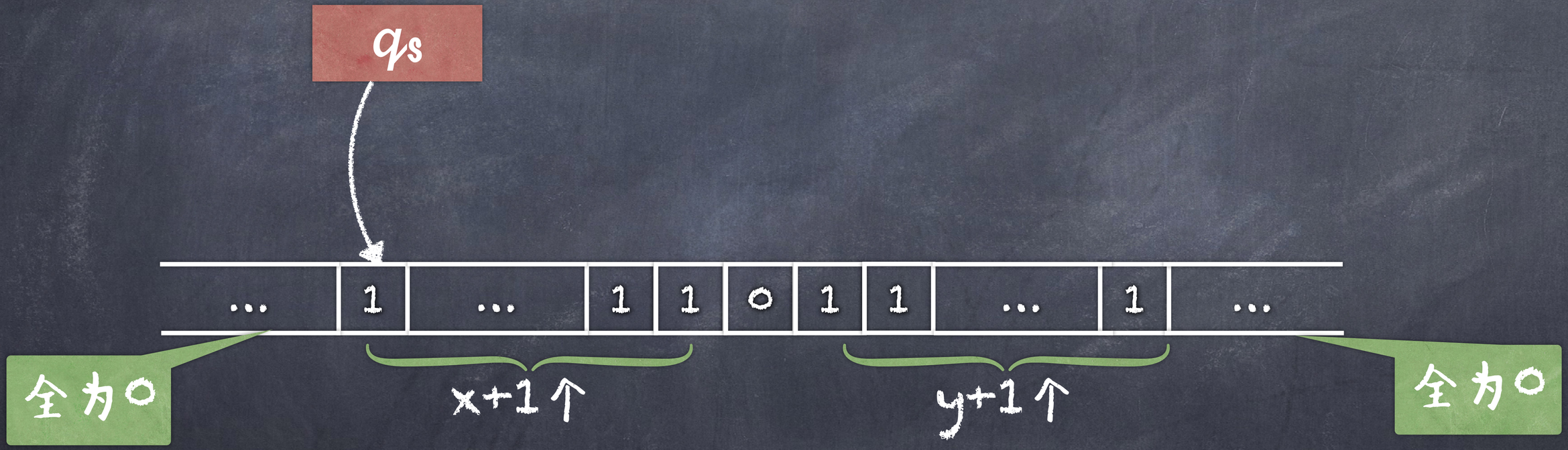
- 如果一个自然数上的函数是可计算的，当且仅当存在一个计算它的图灵机。
  - 即对于任意该函数定义域上的输入，都可以使得图灵机停机，并输出该函数要计算的值。

练习

- 给定下面的图灵机应该怎么运行
  - 函数 $x \dot{-} y = \begin{cases} x - y, & \text{如果} x > y; \\ 0, & \text{否则} \end{cases}$

练习

- 纸带上的输入形如：

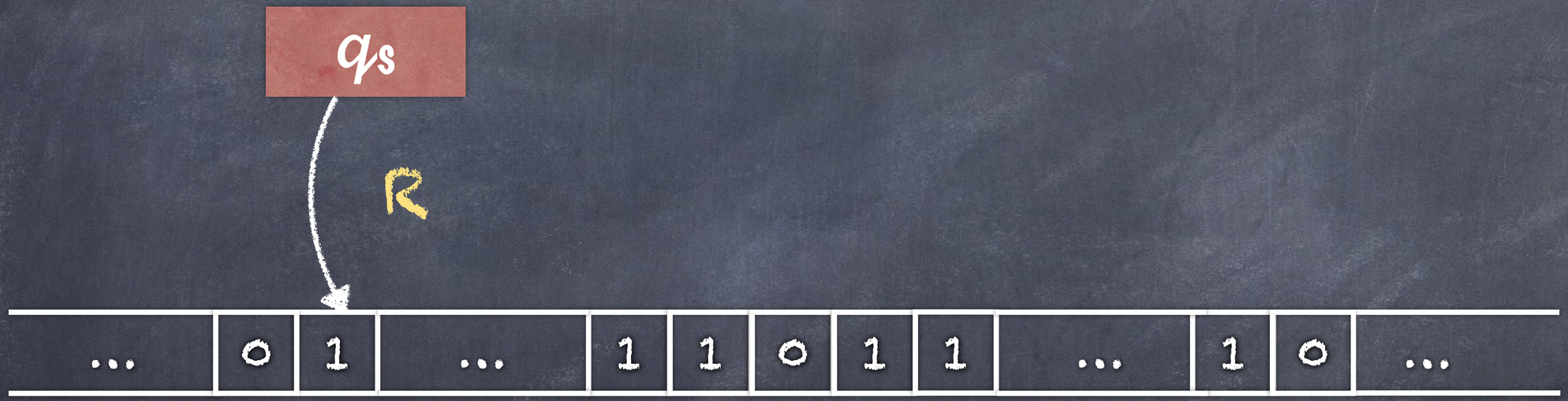


练习

- 思路：
  - 1、可以依次抹去中间空格两边的1，
  - 2、如果左边的1先被抹光，或者两边的1同时被抹光，则说明$x \leq y$，那么我们只要抹去剩下的1然后停机即可。
  - 3、否则，说明$x > y$，则剩余的1，恰好是$x - y$，直接停机即可。

练习

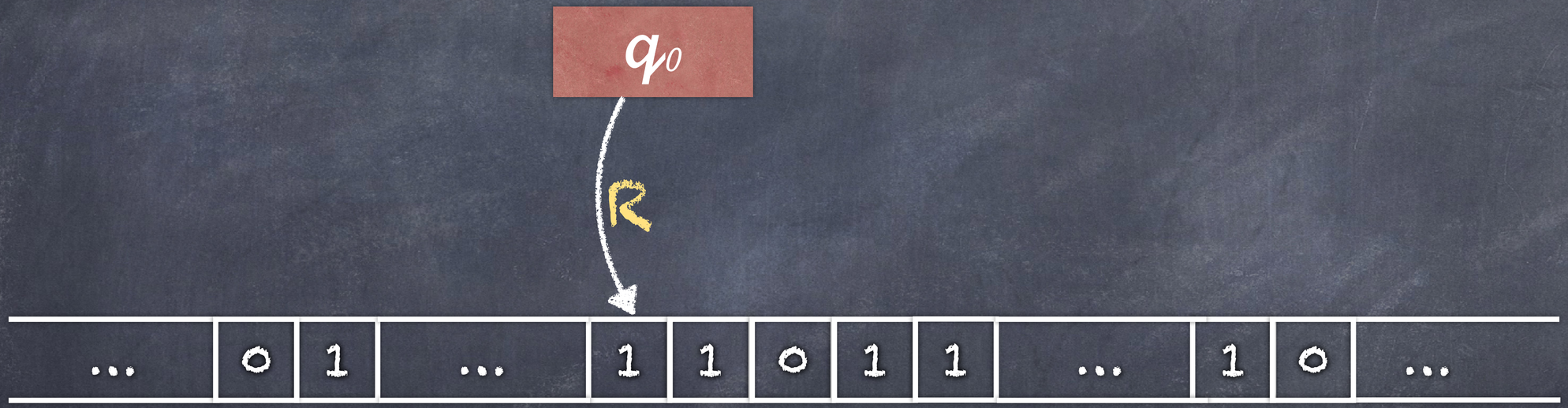
- 过程如下



$(q_s, 1, R, q_0)$

练习

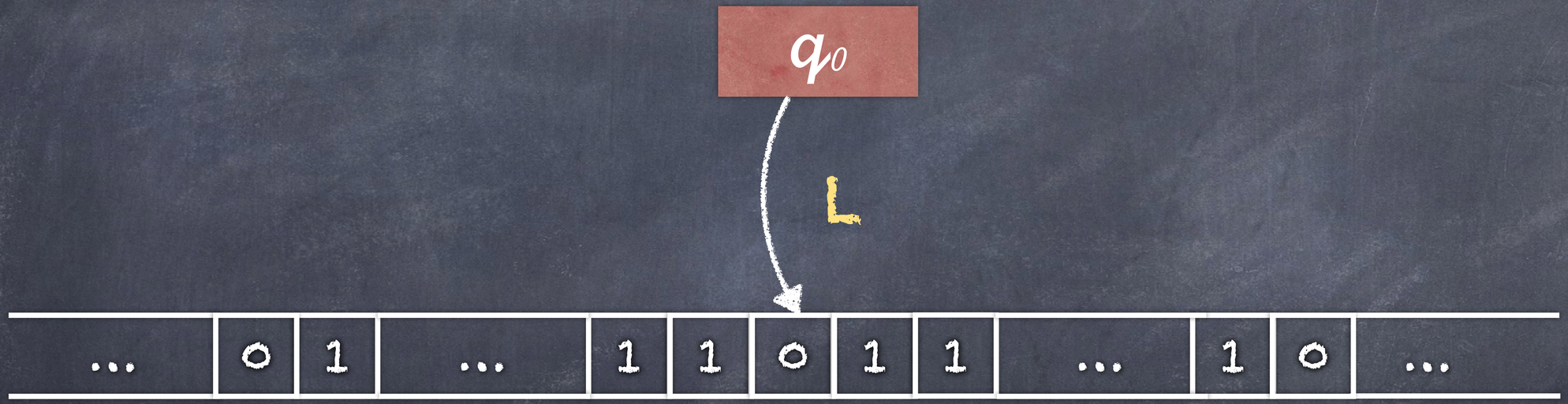
- 过程如下



$(q_0, 1, R, q_0)$

练习

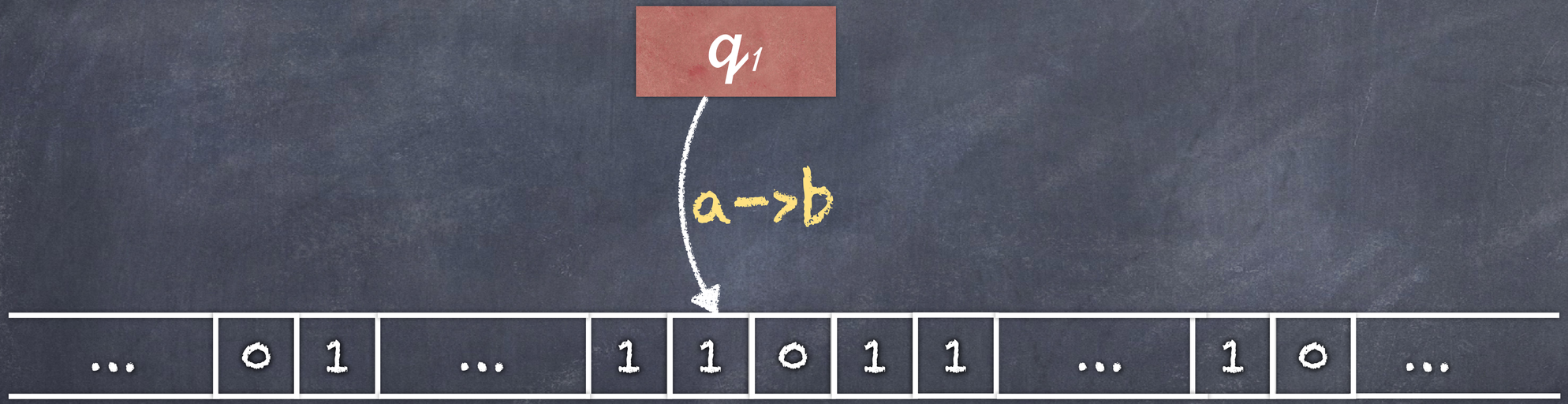
- 过程如下



$(q_0, 0, L, q_1)$

练习

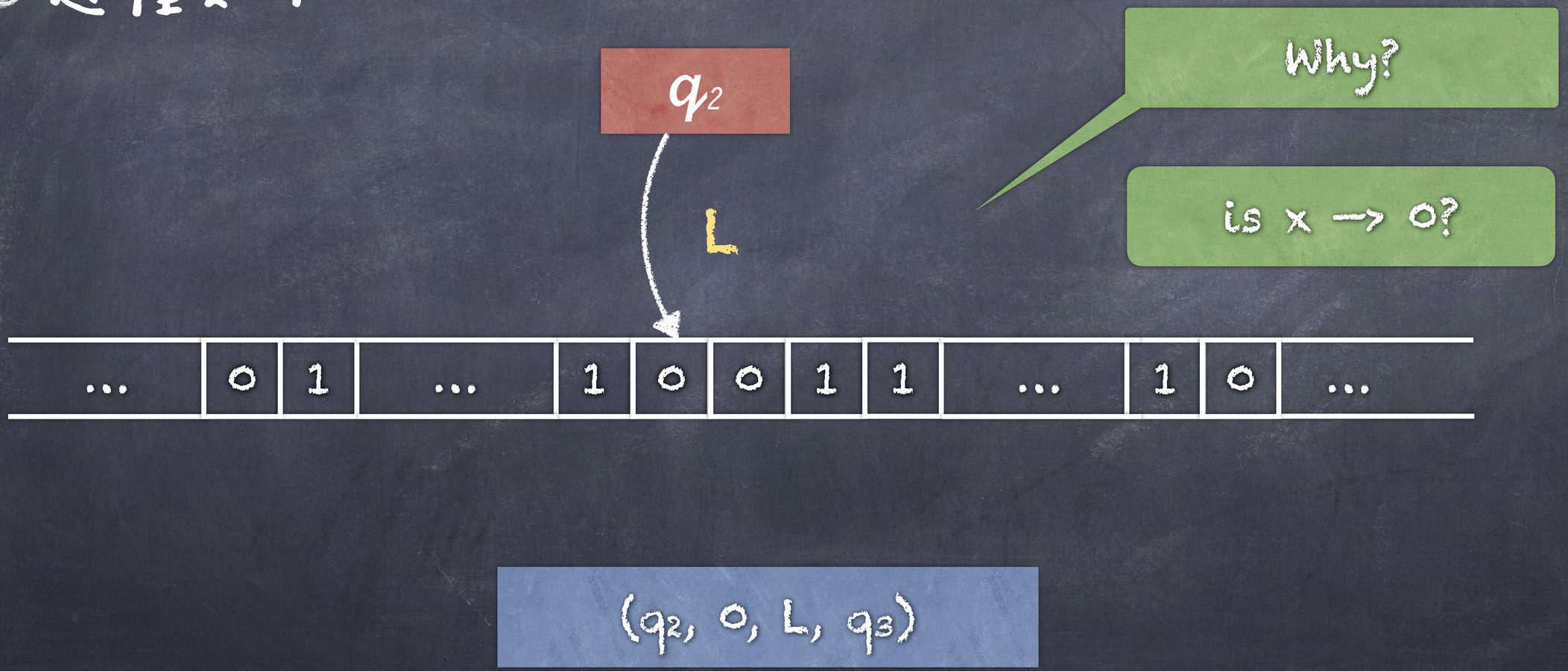
- 过程如下



$(q_1, 1, b, q_2)$

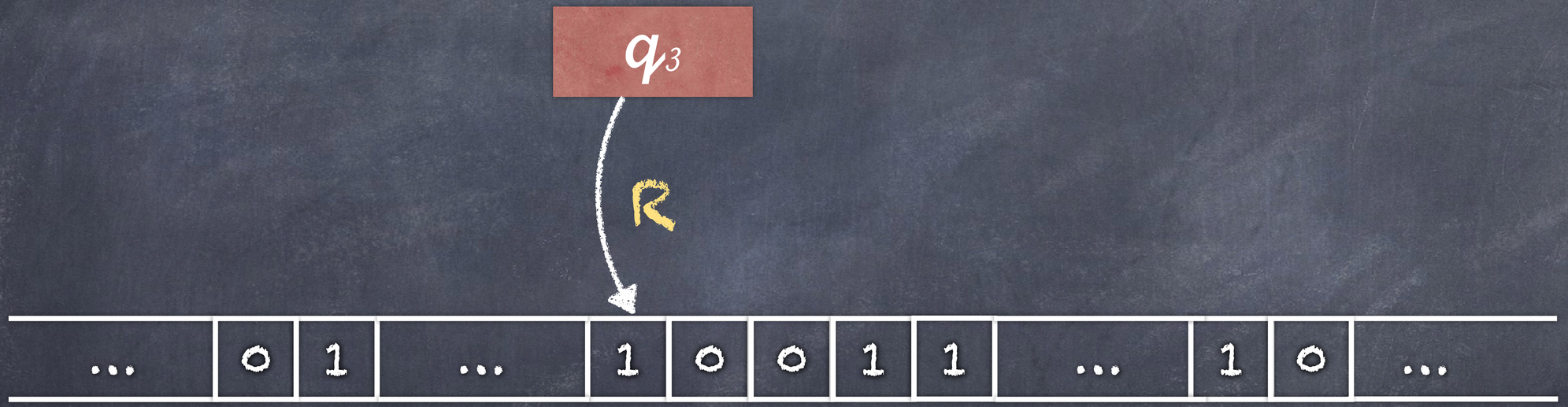
练习

- 过程如下



练习

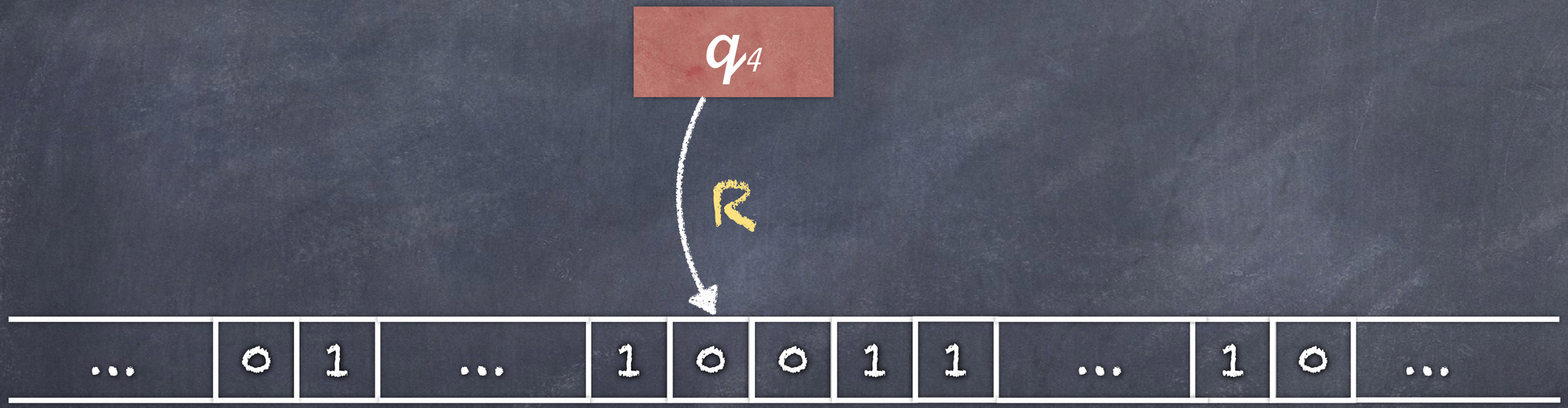
- 过程如下



$(q_3, 1, R, q_4)$

练习

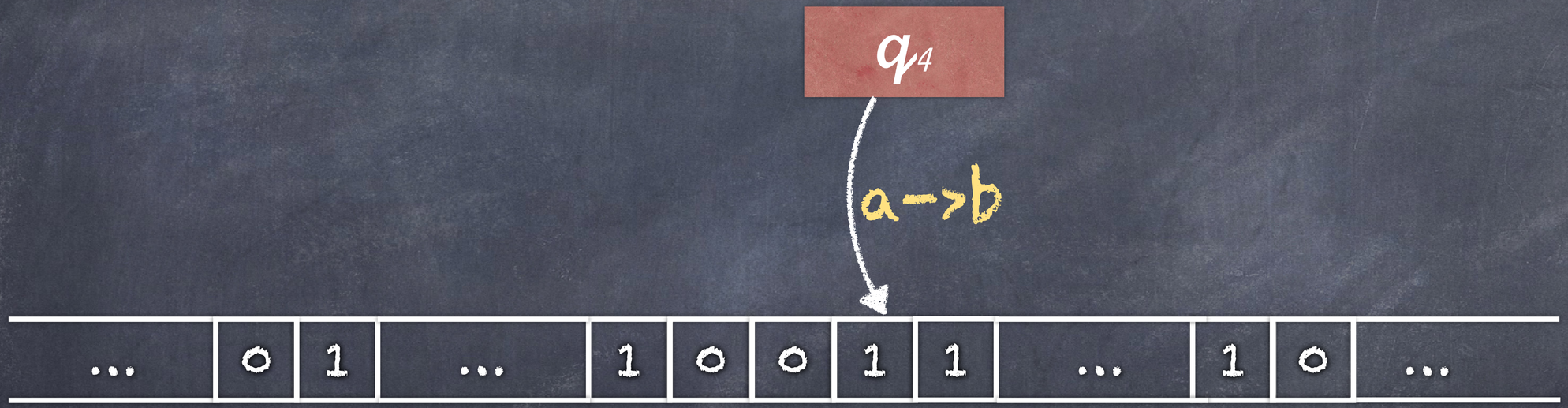
- 过程如下



$(q_4, 0, R, q_4)$

练习

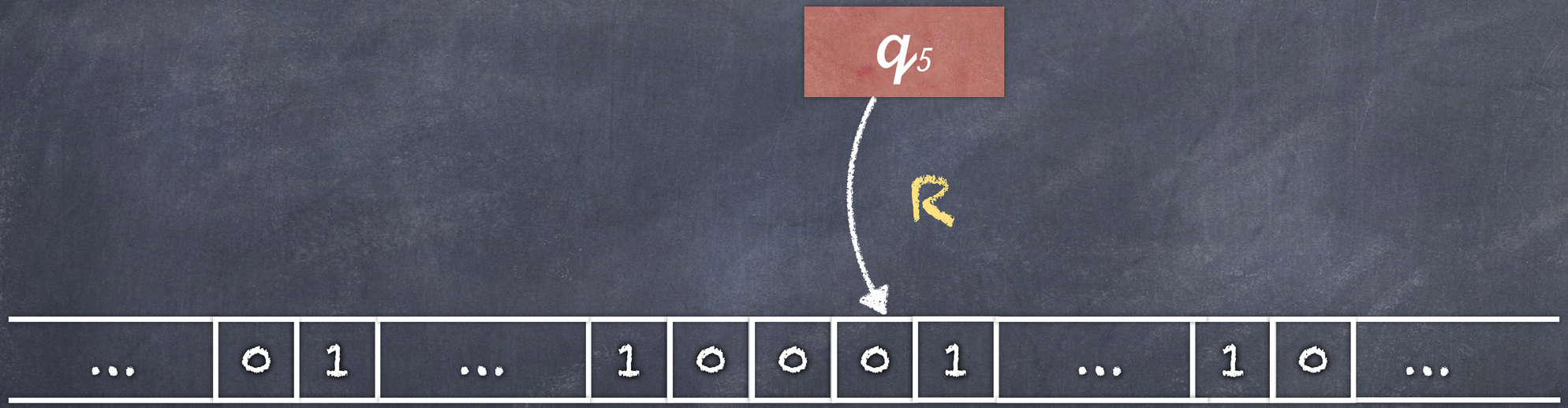
- 过程如下



$(q_4, 1, b, q_5)$

练习

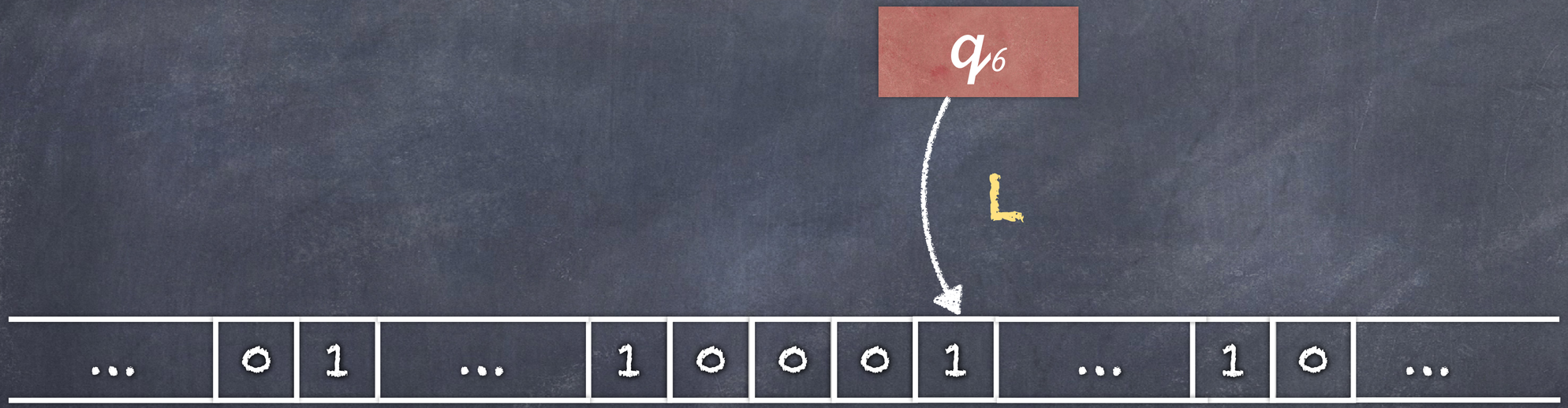
- 过程如下



$(q_5, 0, R, q_6)$

练习

- 过程如下

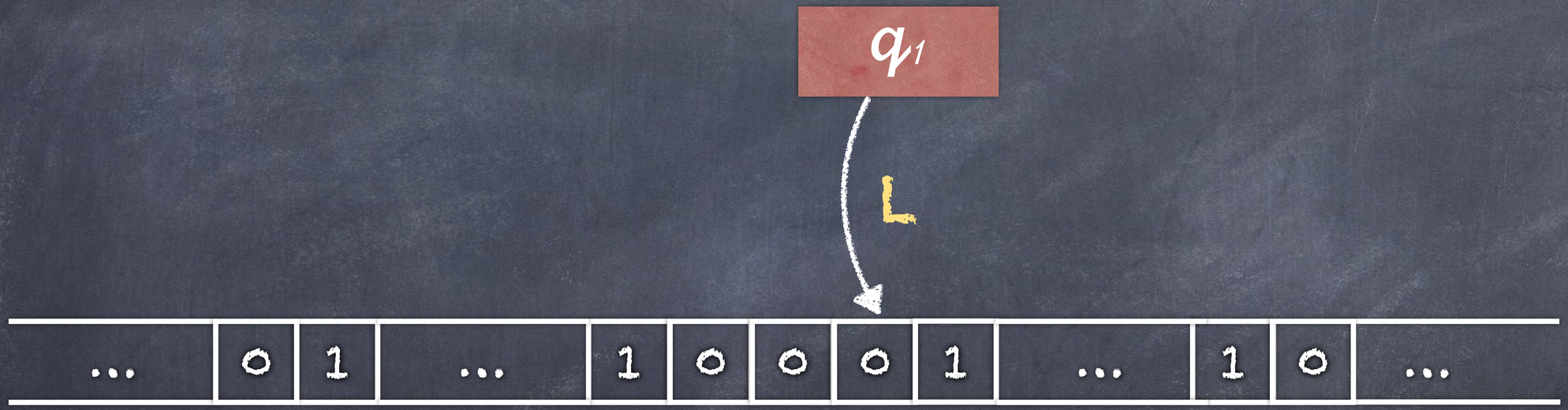


$(q_6, 1, L, q_1)$

Repeat the q_1 state

练习

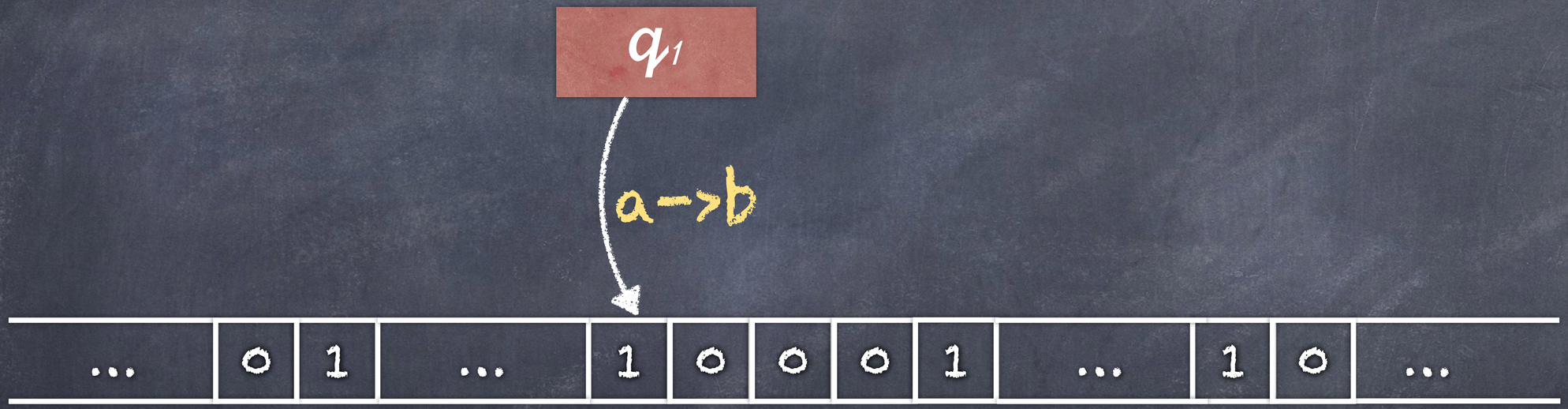
- 过程如下



$(q_1, 0, L, q_1)$

练习

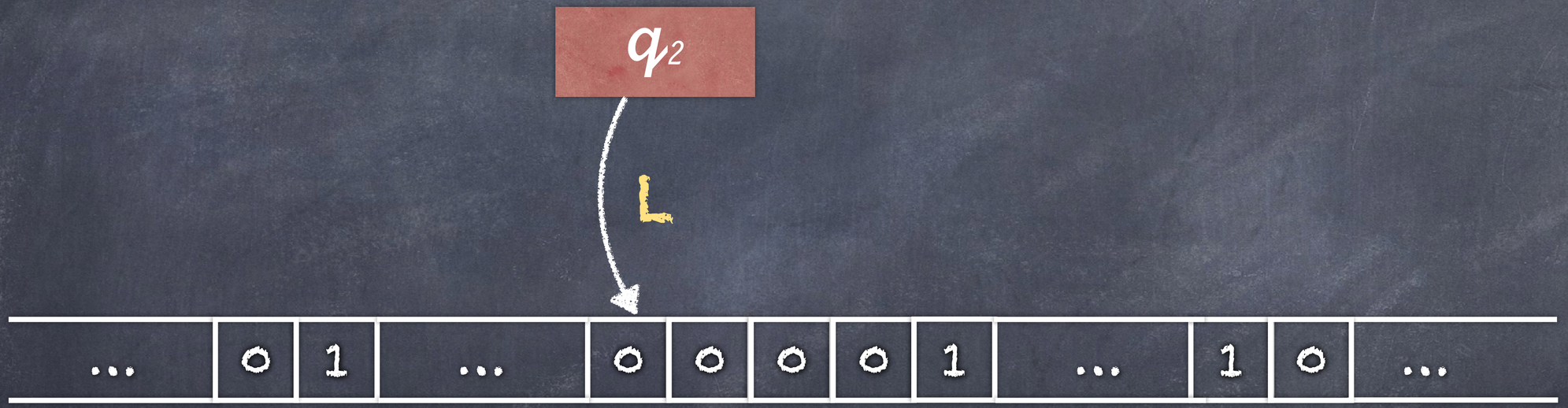
- 过程如下



$(q_1, 1, b, q_2)$

练习

- 过程如下



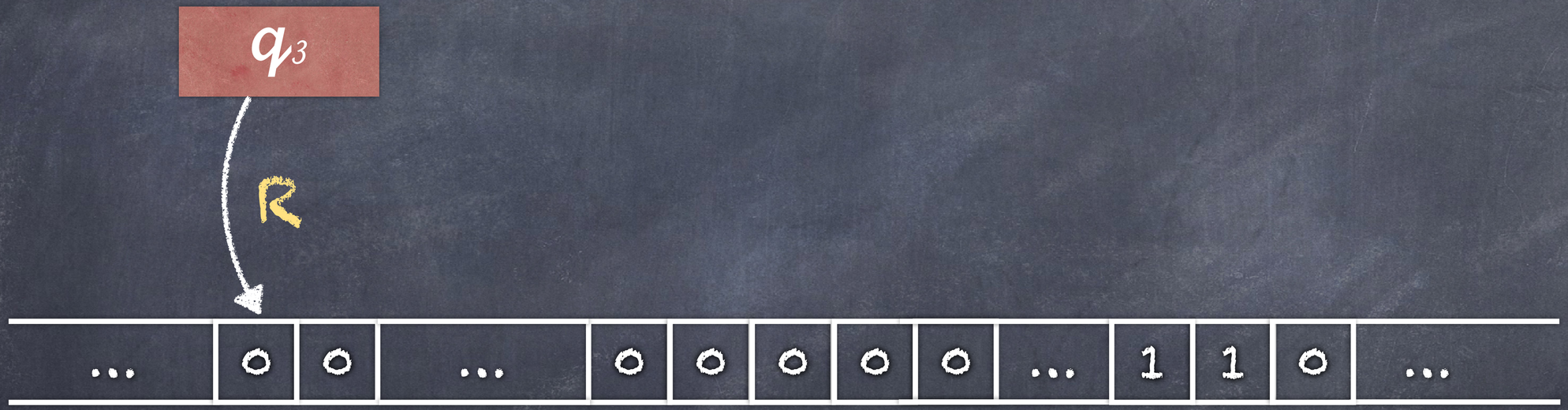
$(q_2, 0, L, q_3)$


2000 YEARS
LATER

情況1：×先到○

练习

情况1：x先到o



(q_3, o, R, q_7)

练习

情况1：x先到o

(q_7, o, R, q_7)

练习

情况1：x先到o

$(q_7, 1, b, q_8)$

练习

情况1：x先到o

$(q_8, 0, R, q_9)$

Why not q_7?

练习

情况1：x先到o

$(q_9, 1, b, q_8)$

练习

情况1：x先到o

(q_8, o, R, q_9)

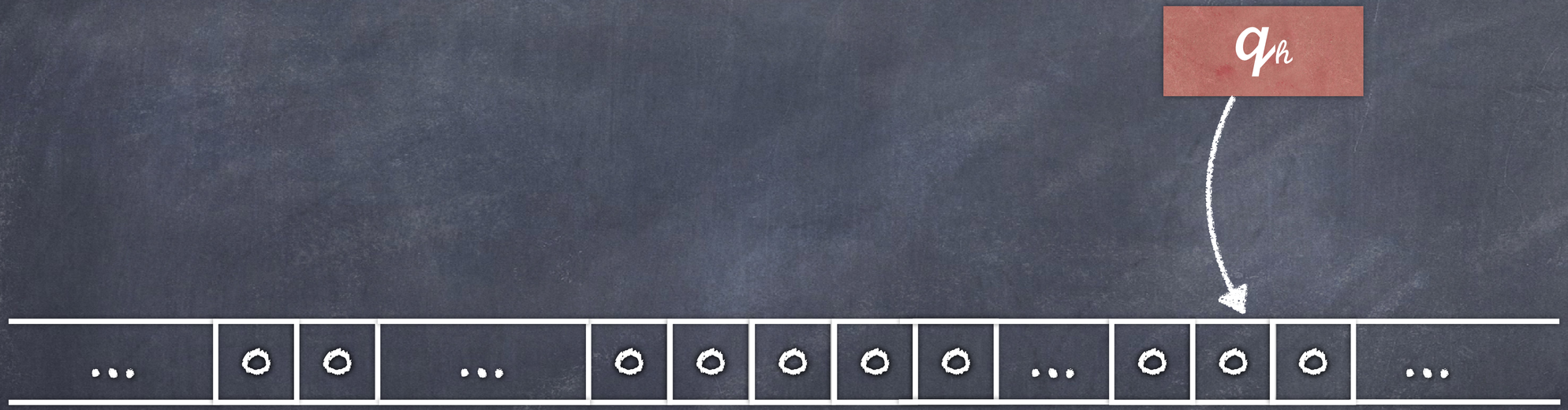
练习

情况1：x先到o

(q_9, o, L, q_h)

练习

情况1：x先到o



情况2：y先到O

练习

情况2: y先到0

$(q_6, 0, L, q_h)$

练习

情况2: y先到o

练习

- 完成函数 $x \dot{-} y$的图灵机即如下指令集

$(q_s, 1, R, q_0)$	$(q_0, 1, R, q_0)$	$(q_0, 0, L, q_1)$	$(q_1, 1, b, q_2)$	$(q_2, 0, L, q_3)$
$(q_3, 1, R, q_4)$	$(q_4, 0, R, q_4)$	$(q_4, 1, b, q_5)$	$(q_5, 0, R, q_6)$	$(q_6, 1, L, q_1)$
$(q_1, 0, L, q_1)$	$(q_3, 0, R, q_7)$	$(q_7, 0, R, q_7)$	$(q_7, 1, b, q_8)$	$(q_8, 0, R, q_9)$
$(q_9, 1, b, q_8)$	$(q_9, 0, L, q_h)$	$(q_6, 0, L, q_h)$		

上述图灵机输出的问题？

1个1代表0!

图灵机编码（Turing number）

- 假定字符集为$\Sigma = \{0,1\}$，定义左移$[L] = 2$和右移$[R] = 3$。
- 定义状态集为 $Q = \{4,5,...,n\}$。其中4为初始状态q_s（即$[q_s] = 4$），而状态集中的最大的自然数n为停机状态q_h
- 接下来，我们只要对四元组(q, a, a', q')，编码为

$$\#(q, a, a', q') = \underbrace{1...1}_{[q]+1个1}\ 0\ \underbrace{1...1}_{[a]+1个1}\ 0\ \underbrace{1...1}_{[a']+1个1}\ 0\ \underbrace{1...1}_{[q']+1个1}$$

- 那么对于指令集$\Delta = \{\delta_0, \delta_1, \ldots, \delta_m\}$而言，图灵机编码为

$$\#\Delta = \#\delta_0 00 \#\delta_1 00 \#\delta_2 \ldots 00 \#\delta_m$$

图灵编码（Turing number）

- #Δ唯一确定了一个图灵机M，我们可以将其和一个自然数对等（2进制）。
- 因此，所有图灵机是可以被枚举出来的。

可以被某个图灵机计算的函数就称为"可计算"函数

一个图灵机只能做一个单一的事情吗？

通用图灵机（Universal Turing Machine）

- 定理：存在一个通用的图灵机U，输入为任何一台图灵机M_1 的编码，以及相应的任何输入I，那么其输出和M_1在I输入下的输出一致。
- 即U可以模拟任何图灵机！

证明通用图灵机的存在

- 通过构造这样的图灵机（proof by construction）
- 在构造之前，我们需要3个工具图灵机：
  - 拷贝机（Copy machine）
  - 匹配机（Match machine）
  - 替换机（Term substitution machine）

工具1：拷贝机

- 拷贝机：
  - 让$和#为纸带间的不同于0和1的两个符号，之间只存在0和1的符号串。假设此刻读写头处于$处，那么我们可以构造一个图灵机，可以数紧挨着#之后的毗邻的1个数，并将这些1拷贝到紧挨着$之后（当然，我们假设#和$之后的空间足够）。拷贝顺序也可以相反，从$之后拷贝到#之后。

工具1：拷贝机

- 构造步骤：
- 添加一个新的字符B，与$、#、0和1都不同，初始读写头移到$处

工具1：拷贝机

- 构造步骤：
- 步骤1（a）：处向右移动一格，如果不是1，进入步骤4

工具1：拷贝机

- 构造步骤：
- 步骤1（b）：将其设为 B。

工具1：拷贝机

- 构造步骤：
- 步骤2：移到#外第一个格子，并填入1

工具1：拷贝机

- 构造步骤：
- 步骤3：移到上一个B处，然后跳到步骤1

工具1：拷贝机

- 构造步骤：
- 一直进行，直到遇到0

工具1：拷贝机

- 构造步骤：
- 步骤4：向左移动，遇到B就改为1，直到$为止。

工具2：匹配机

- 匹配机：
  - 让$和#为纸带间的不同于0和1的两个符号，之间只存在0和1的符号串。假设此刻读写头处于$处，那么我们可以构造一个图灵机，可以比较$之后连续的1和#之后连续的1，如果相同，以一个状态停止。否则不相同的话，以另一个状态停止。

工具2：匹配机

- 构造步骤：
- 添加一个新的字符B，与$、#、0和1都不同，初始读写头移到$处。

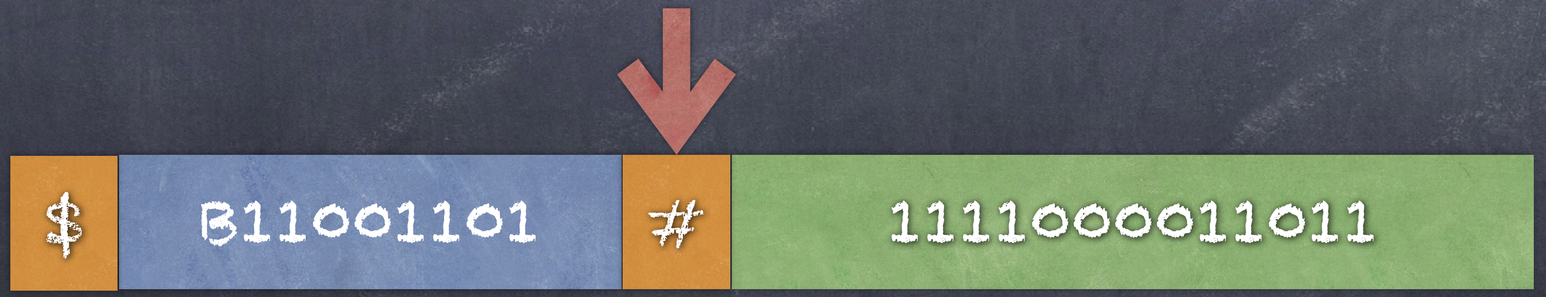
工具2：匹配机

- 构造步骤：

- 步骤1（a）：向右寻找1，如果找到，设为B，遇到B继续向右，否则如果遇到0或到#符，则跳到步骤2

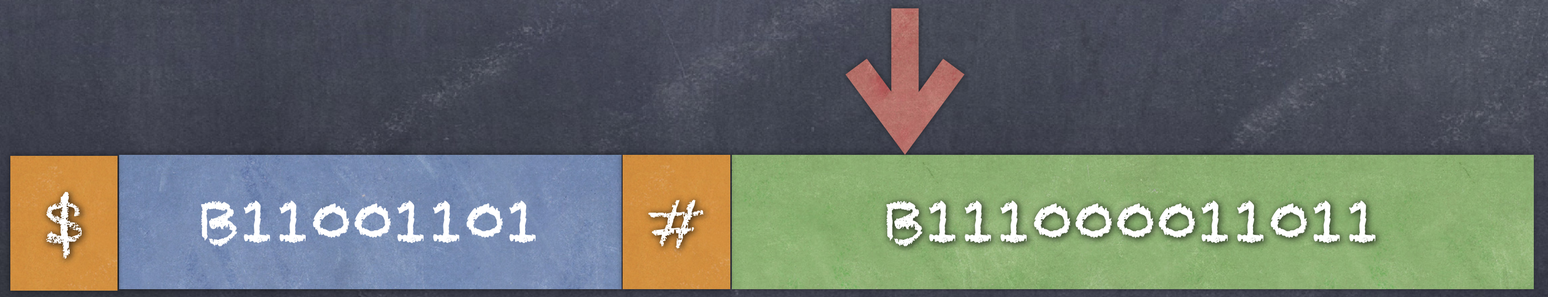
工具2：匹配机

- 构造步骤：
- 步骤1（b）：移动到#处



工具2：匹配机

- 构造步骤：
- 步骤1（c）：向右寻找1，如果找到，设为B，遇到B继续向右，否则如果遇到0或到其它符，跳到步骤3



工具2：匹配机

- 构造步骤：
- 步骤1（d）：移动到$处，跳到步骤1（a）。

工具2：匹配机

- 构造步骤：

- 步骤2：移到#处，继续向右寻找1，找到了，停止在状态q_{ne}，否则即为两者相等，停止在q_e。

工具2：匹配机

- 构造步骤：
- 步骤3：因为#后没有相应的1，停止在状态q_{ne}。

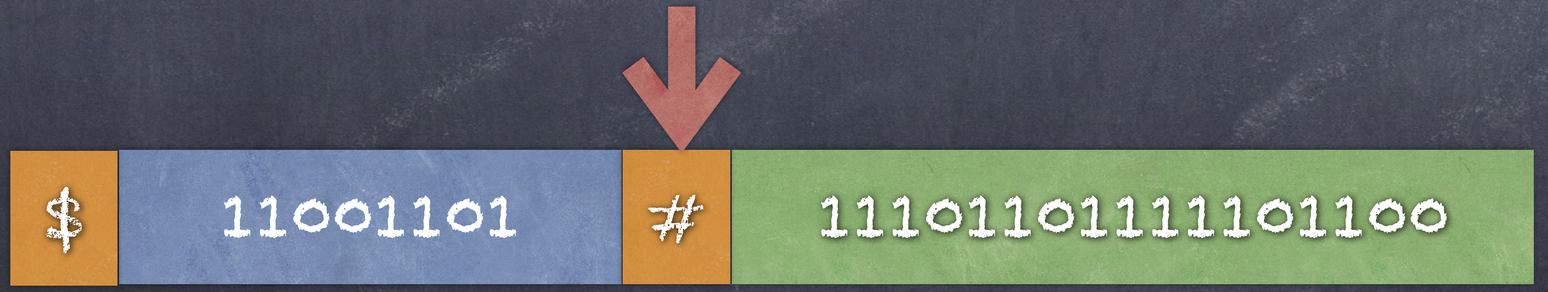
工具3: 替换机

- 替换机：
  - 让$和#为纸带间的不同于0和1的两个符号，之间只存在0和1的符号串。令S为紧挨着#之后0，1组成的串（有多个连续的1组成的项，项之间用0隔开），k为紧挨着$之后的一个连续的1串，假设此刻读写头处于$处，那么我们可以构造一个图灵机，用k来替换S的第一个项。

工具3: 替换机

- 构造步骤：

- 步骤1 收缩（a）：移动到#处，查看紧挨#之后为0还是1，如果为0，说明已经收缩完毕，进入步骤插值，否则进入收缩（b）。



工具3: 替换机

- 构造步骤：
- 步骤1 收缩（b）:先移到字符串的最末端（00），然后对每个字符进行一一向左移。

工具3: 替换机

q_m表达"记住"当下的符号之前是o。

- 构造步骤：
- 步骤1 收缩（c）细节：先用一个状态记住当下的字符，然后左移。再根据前面的状态进行修改，然后重复。

(q_m, o, L, q_n)

$(q_m, 1, L, q_n)$

Case 1:

(q_n, o, o, q_m)

$(q_n, 1, o, q_v)$

工具3: 替换机

q_v表达"记住"当下的符号之前是1。

- 构造步骤：
- 步骤1 收缩（c）细节：先用一个状态记住当下的字符，然后左移。再根据前面的状态进行修改，然后重复。

$(q_v, 0, L, q_z)$

$(q_v, 1, L, q_z)$

Case 2:

$(q_z, 0, 1, q_m)$

$(q_z, 1, 1, q_v)$

工具3：替换机

- 构造步骤：
- 步骤1 收缩（d）：跳到步骤 收缩（a），继续收缩。

一轮过后收缩了一位：

工具3：替换机

- 构造步骤：
- 步骤2 插值（a）：移到$或者B处，找下一个1，如果下一个为0或者#，则停机。

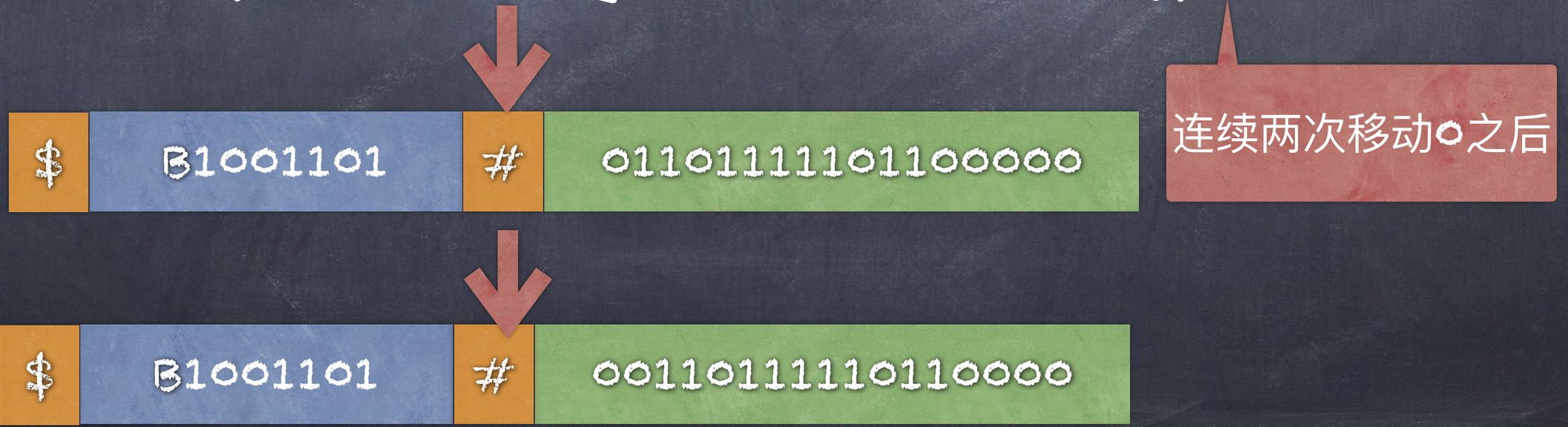
工具3: 替换机

- 构造步骤：
- 步骤2 插值（b）：将1设为B

工具3: 替换机

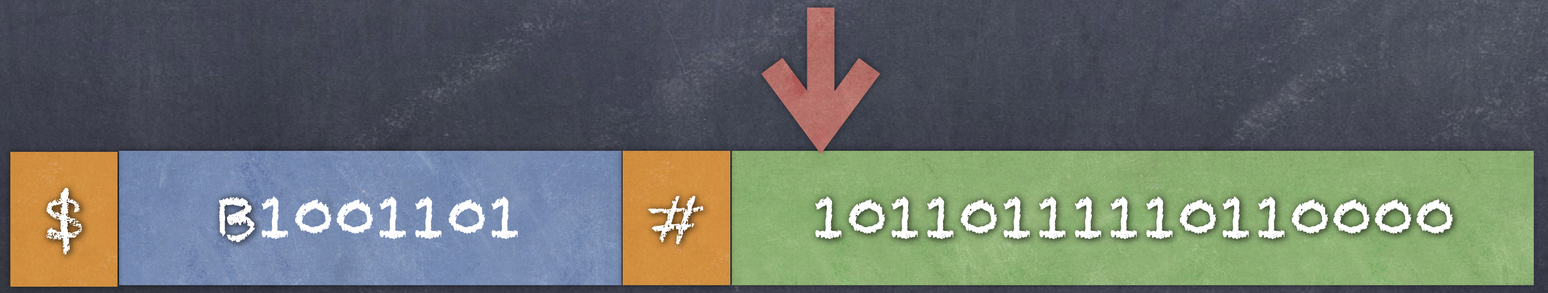
- 构造步骤：

- 步骤2 插值（c）：移动到#处，将#之后的所有的字符串统一向右移动一位（逆向收缩过程），再回到#处。



工具3: 替换机

- 构造步骤：
- 步骤2 插值（d）：将#后第一个0设为1，回到步骤2插值（a）。



工具3：替换机

- 构造步骤：
- 步骤2 插值（d）：将#后第一个0设为1，回到步骤2插值（a）。

最终就完成了替换：

除了三个工具图灵机外，我们还需要使得
多个图灵机能够相互配合

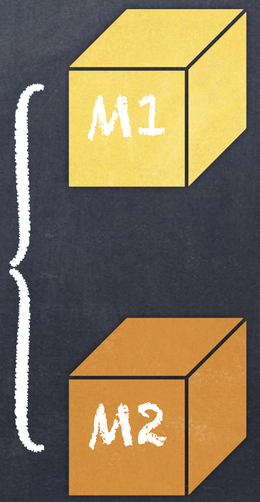
图灵机之间的调用

- 我们可以很容易的让两个图灵机配合做一件事情：
  - 比如对图灵机M_1，M_2，而言，如果我们想先做完M_1的计算之后再做M_1的事情，那么我们可以按照下面方式进行

1. 重新命名M_2的状态集，使得它的初始状态为M_1的停机状态。
2. 除此之外的M_2的所有状态都修改为不能在M_1状态中出现的状态符。
3. 最后相应地修改M_2中的指令集，使用更改后的状态表示。

图灵机之间的调用

- 我们可以很容易的让两个图灵机配合做一件事情：
  - 比如对图灵机M_1，M_2（假设通过重命名使得状态集不相交），可以进行当前纸带读的字符为0，则选择执行M_1，否则就执行M_2：
    1. 给出两个新状态q_x和q_y，使得其不在M_1和M_2中，此外q_{s1}和q_{h1}为M_1的起始和终止状态，q_{s2}和q_{h2}是M_2对应状态。
    2. 添加指令$(q_x,0,0,q_{s1})$，$(q_x,1,1,q_{s2})$
    3. 添加指令$(q_{h1},0,0,q_y)$，$(q_{h1},1,1,q_y)$，$(q_{h2},0,0,q_y)$，$(q_{h2},1,1,q_y)$



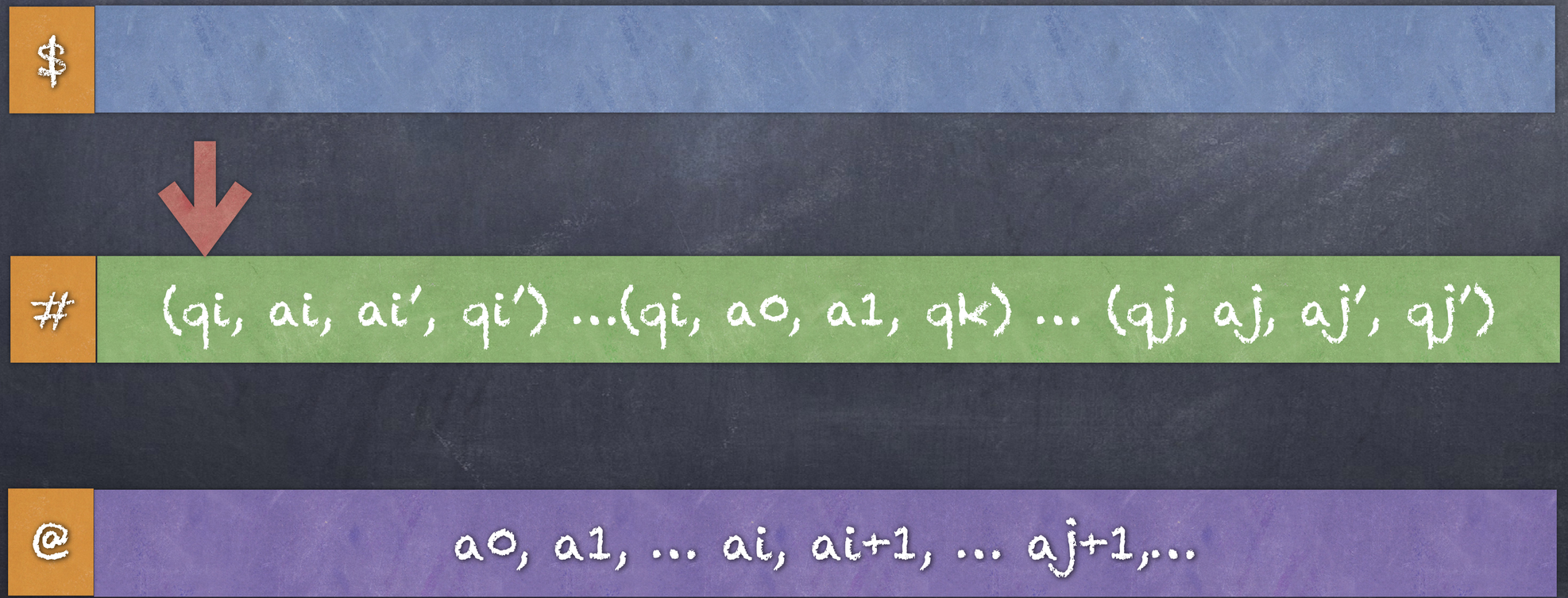
现在我们已经具备了构造通用图灵机的条件了

通用图灵机的算法

- 通用图灵机的纸带样式：

通用图灵机的算法

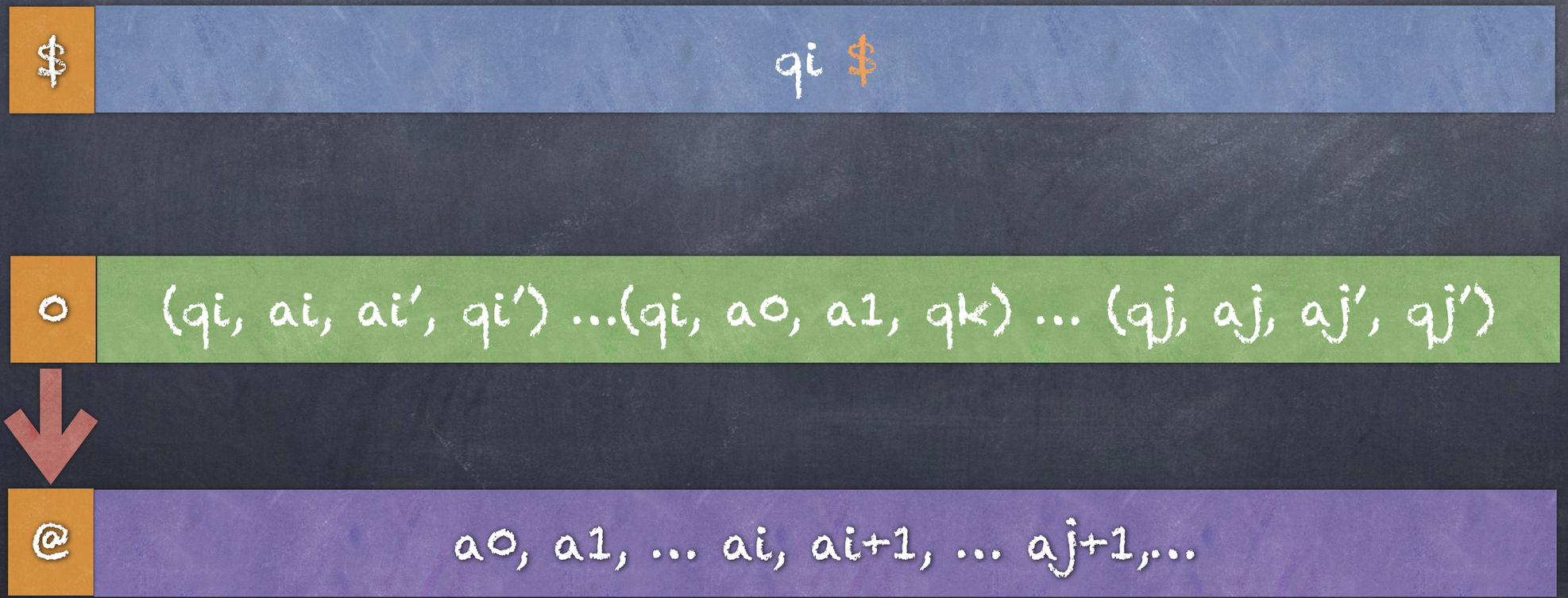
- 初始状态：指向第一个指令的开始



通用图灵机的算法

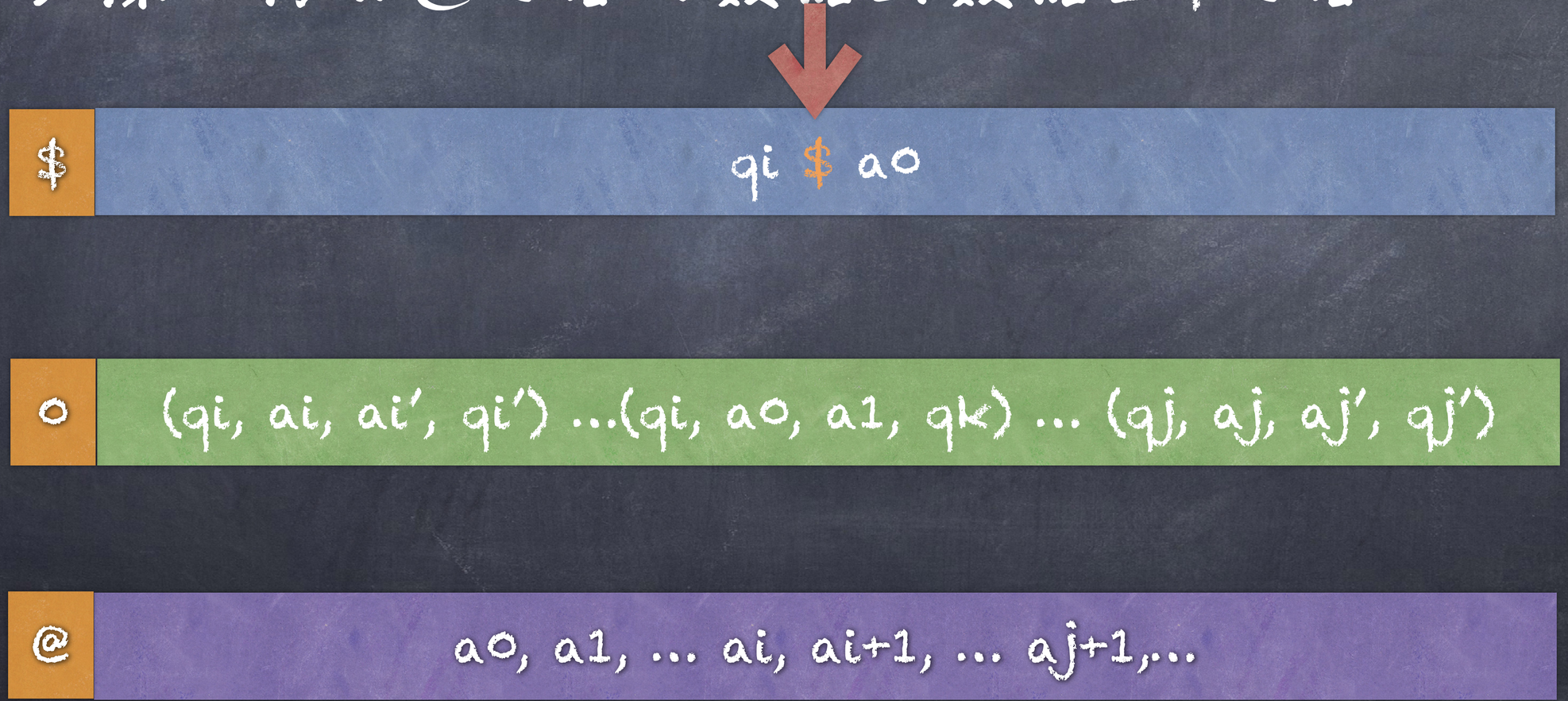
为什么

- 步骤1：拷贝当前状态进入buffer，指向数据区，擦除#



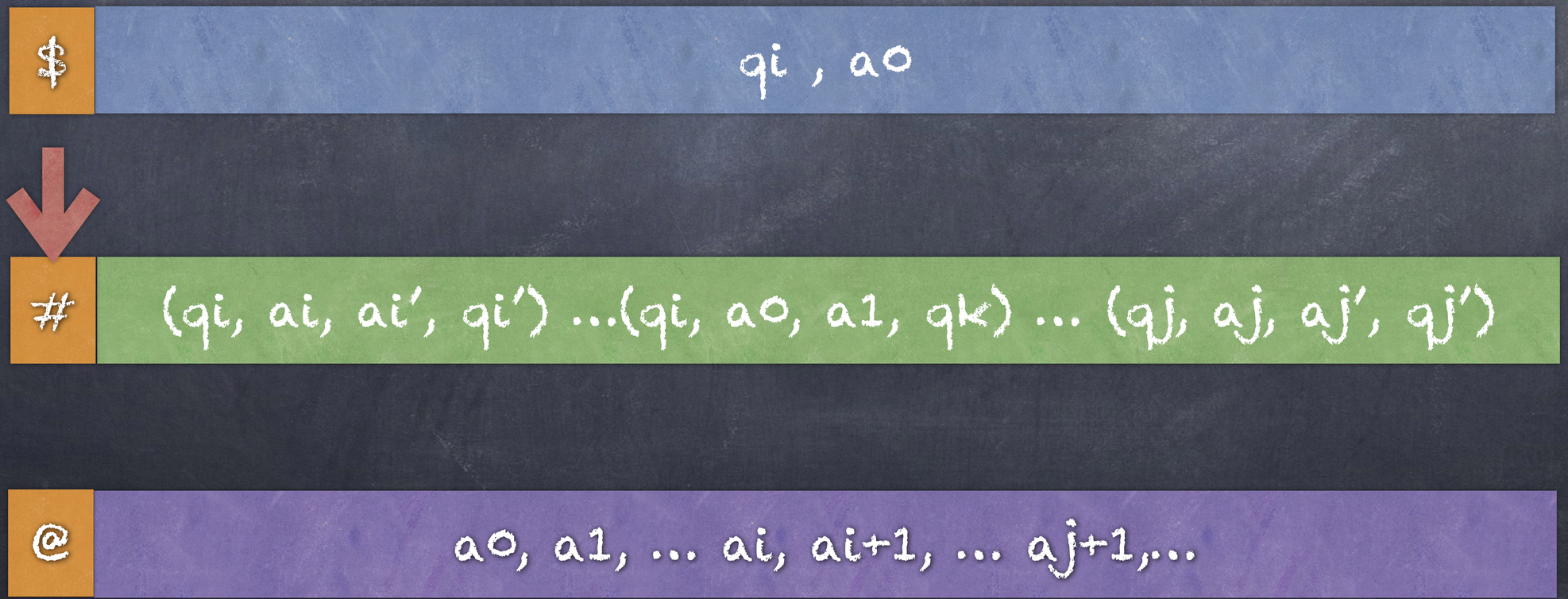
通用图灵机的算法

- 步骤2：拷贝@之后的数据到数据区$之后



通用图灵机的算法

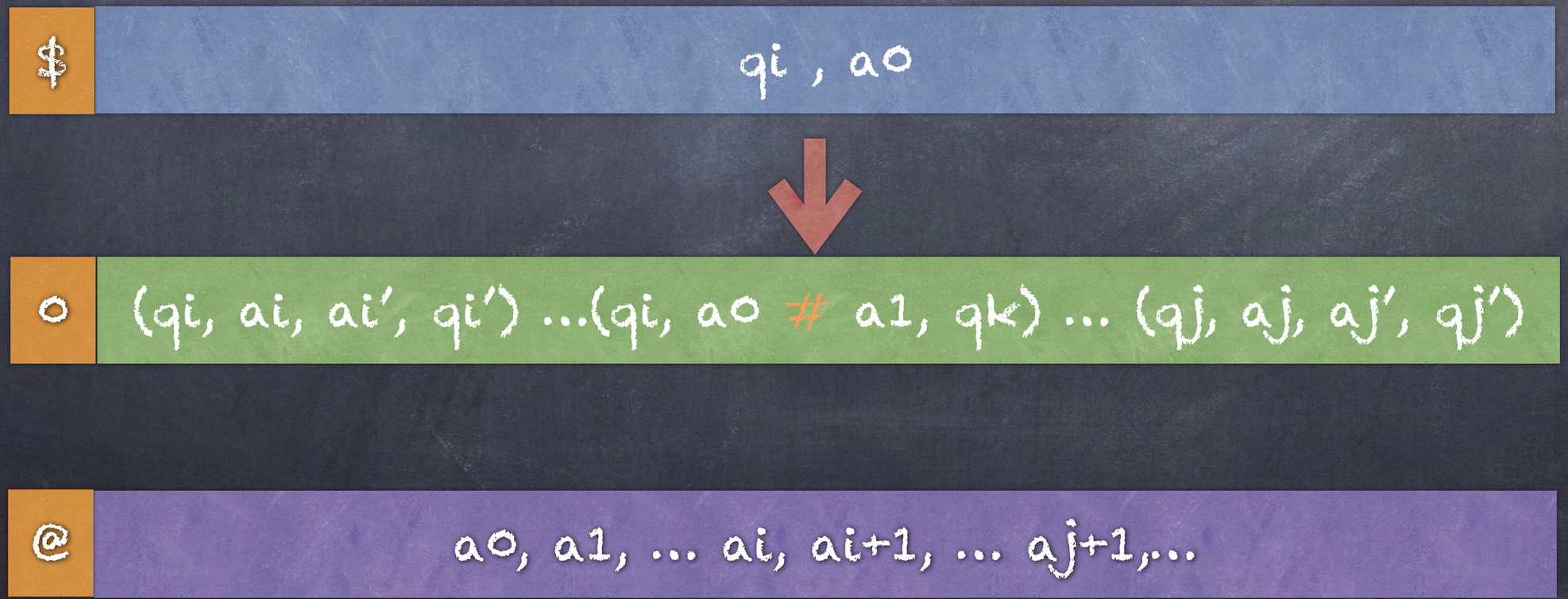
- 步骤3: 擦除$和重设设定#



通用图灵机的算法

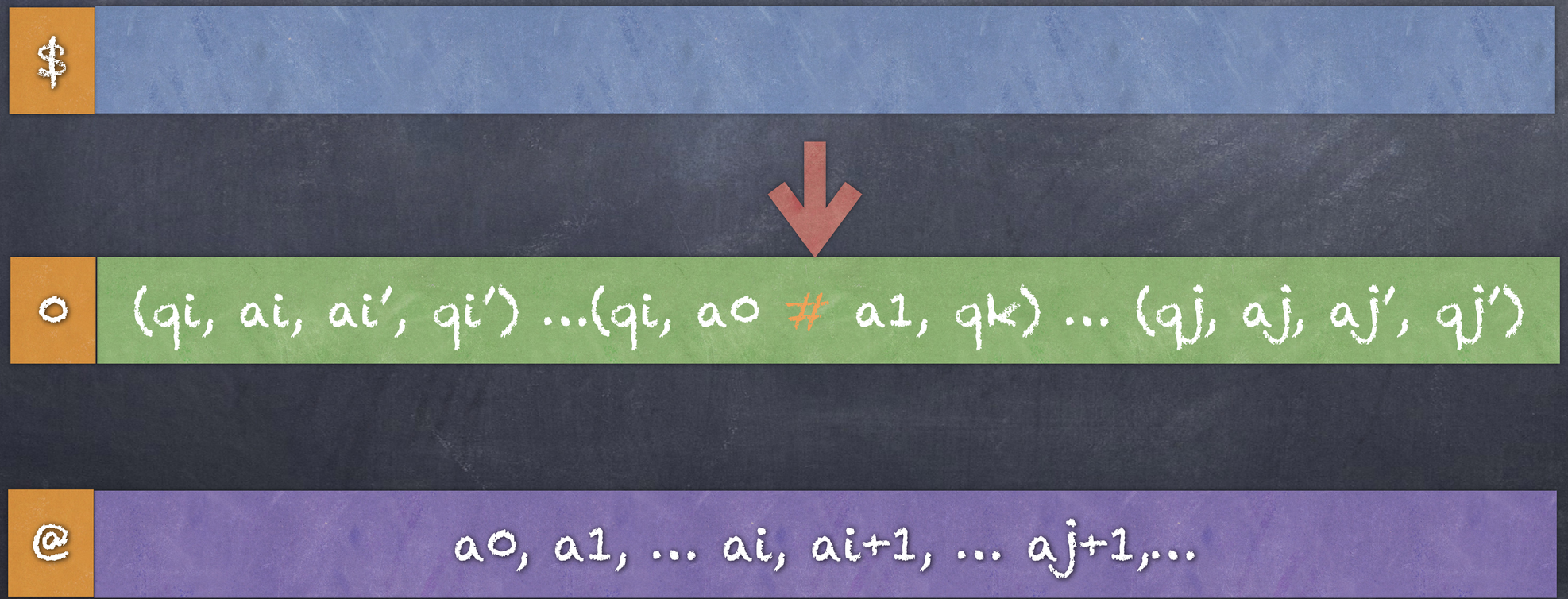
如果没有，即停机，输出

- 步骤4：在指令集中寻找匹配的 （q, a, a′, q′）



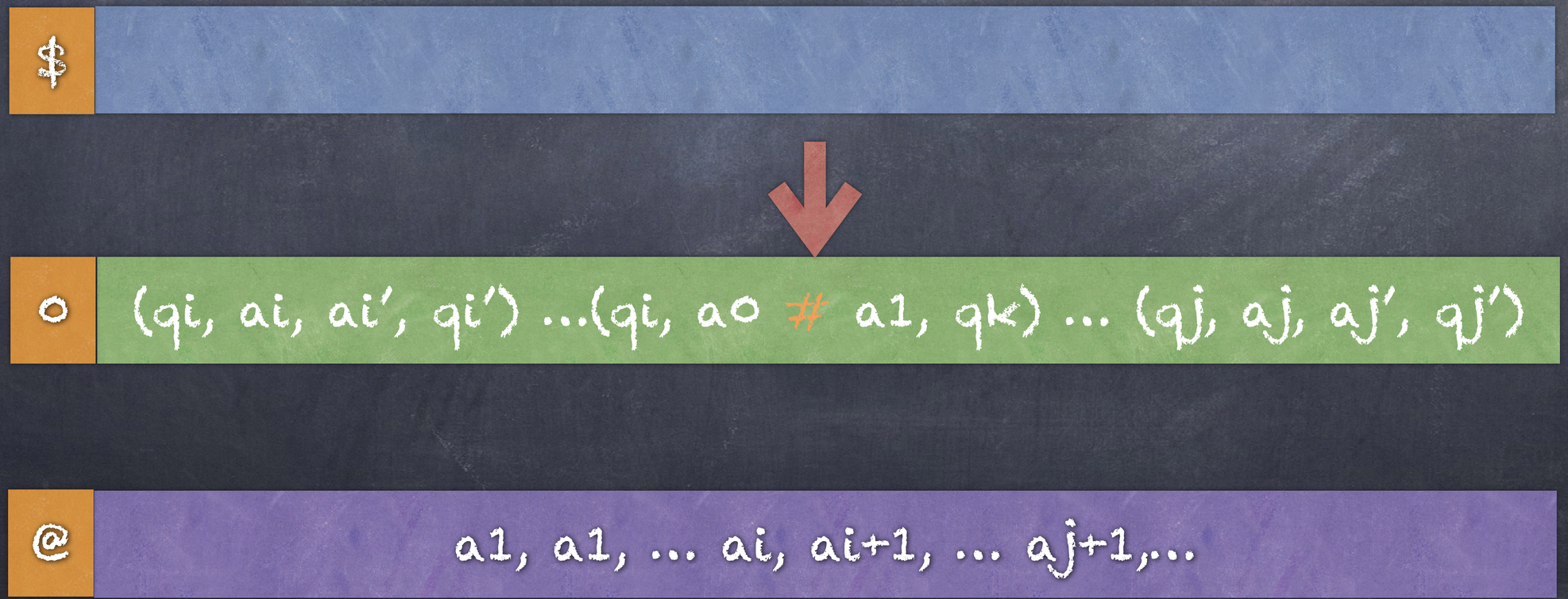
通用图灵机的算法

- 步骤5: 擦除buffer内容



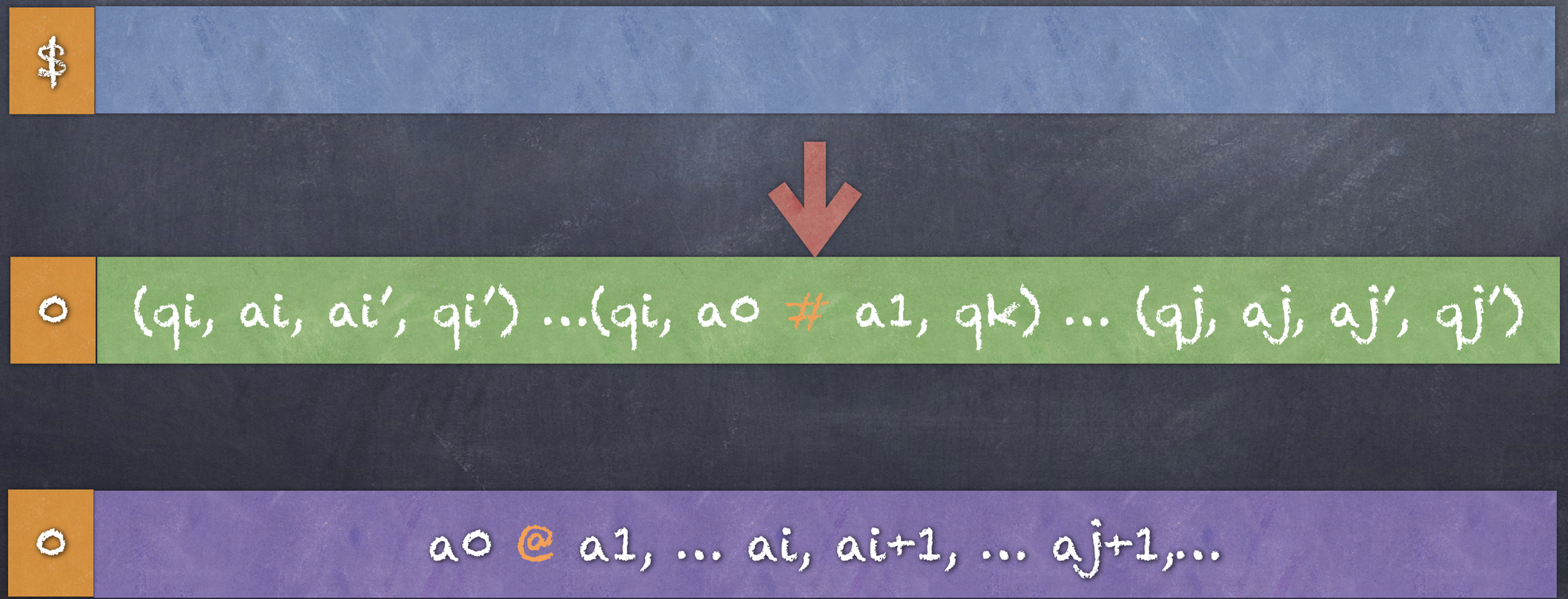
通用图灵机的算法

- 步骤6（a）：如果a′是改变符号操作，将#之后的新符号写入@之后的格中



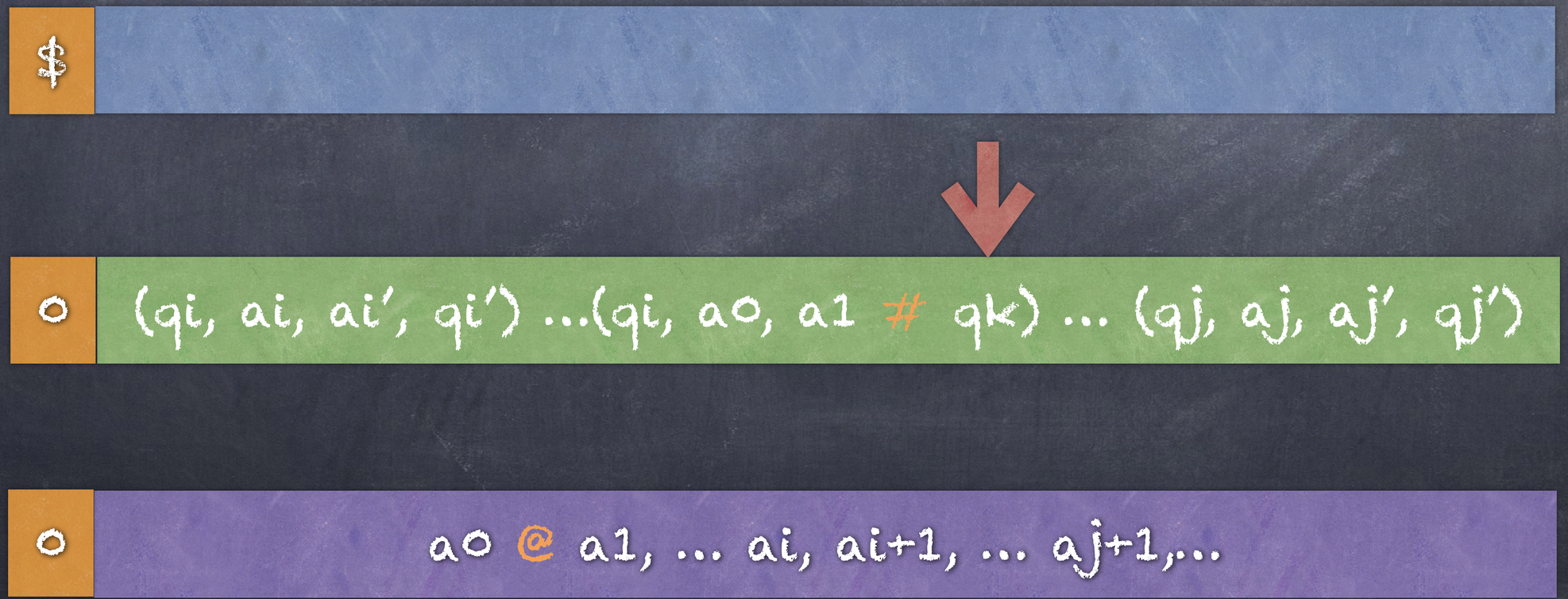
通用图灵机的算法

- 步骤6（b）：如果a′是移动操作，将@做相应的移动



通用图灵机的算法

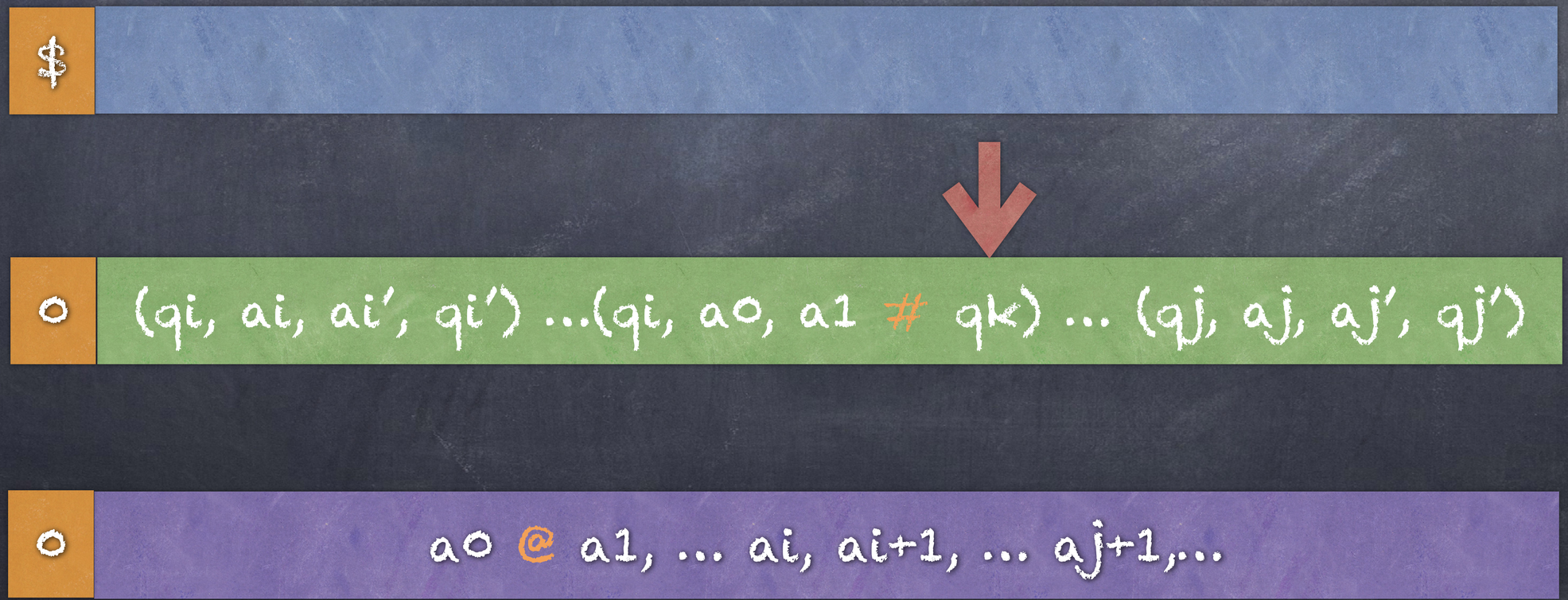
- 步骤7: 定位到下一个状态，然后回到步骤1



现在可以解释为什么一开始要把#设为0，再重设

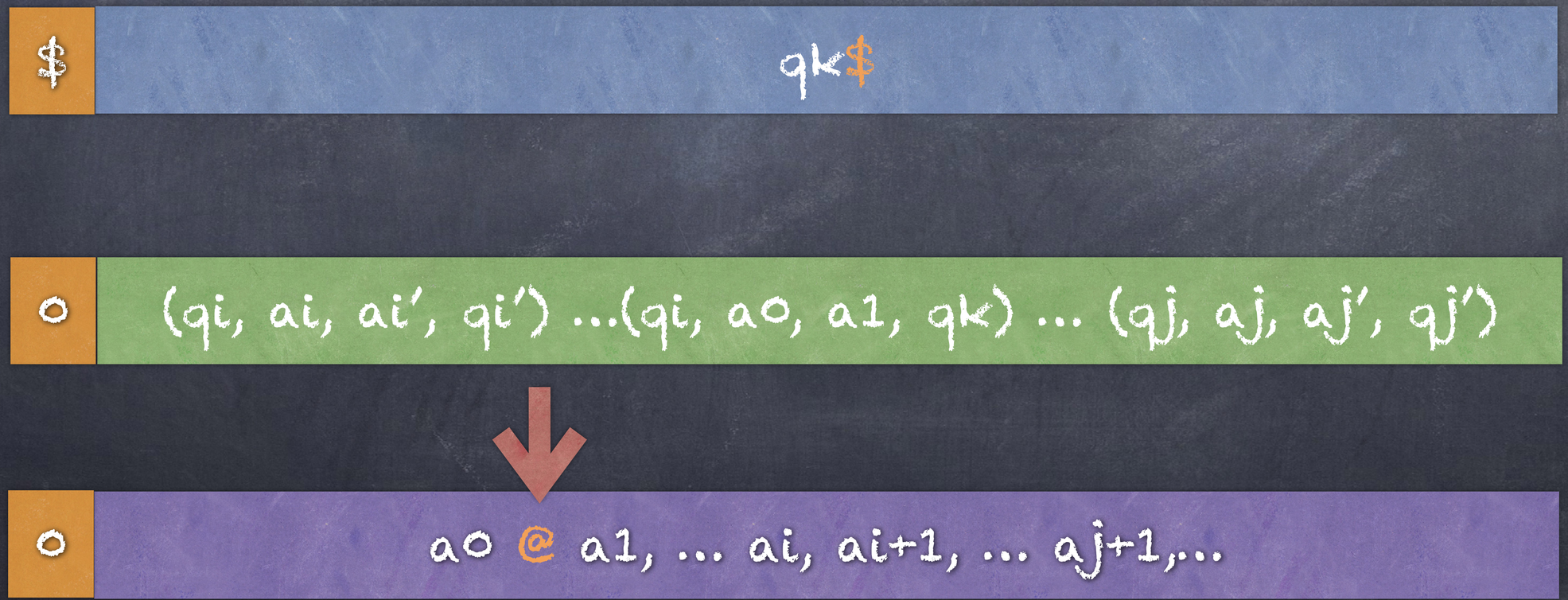
通用图灵机的算法

- 第二次迭代



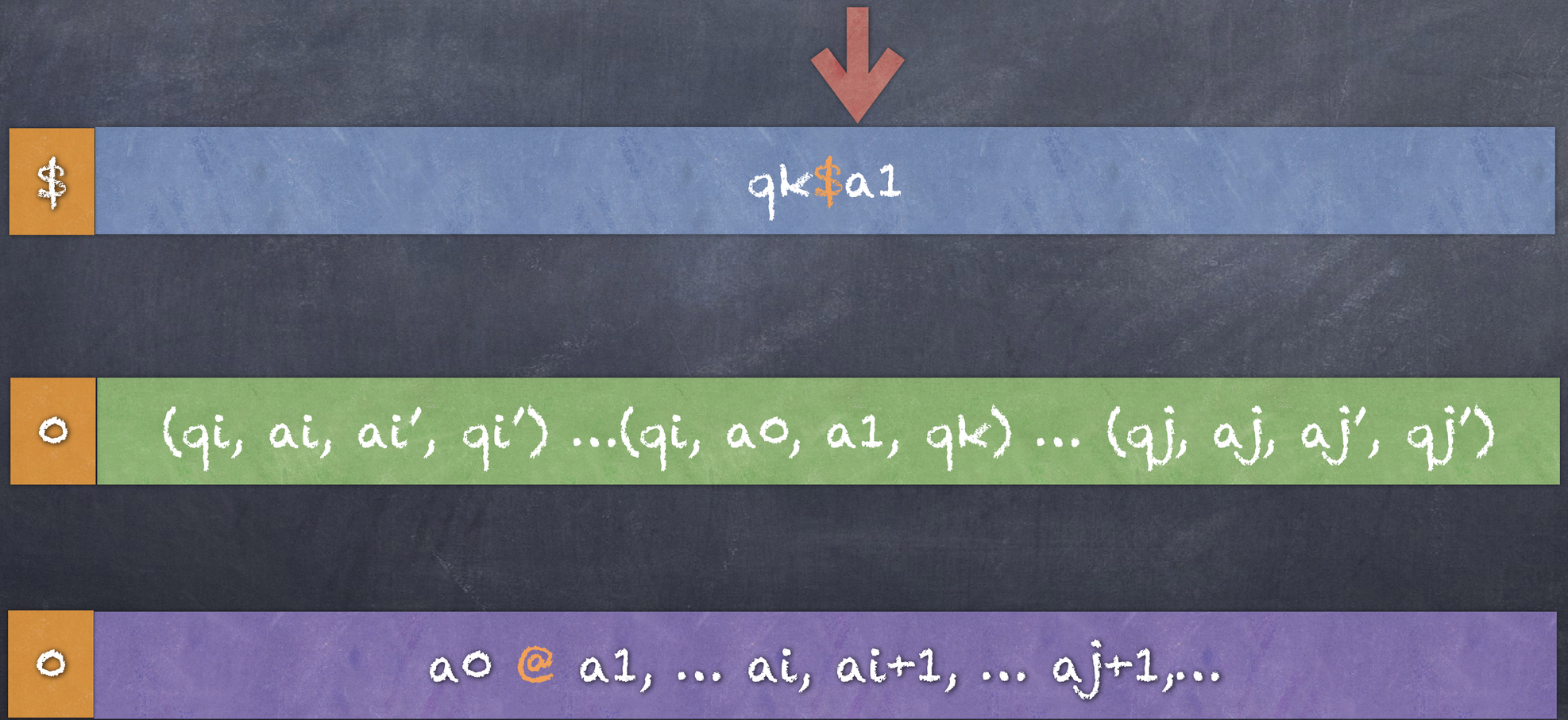
第二次迭代

- 步骤1：拷贝当前状态进入buffer，指向数据区，擦除#



第二次迭代

- 步骤2：拷贝@之后的数据到数据区$之后



第二次迭代

- 步骤3: 擦除$和重设设定#

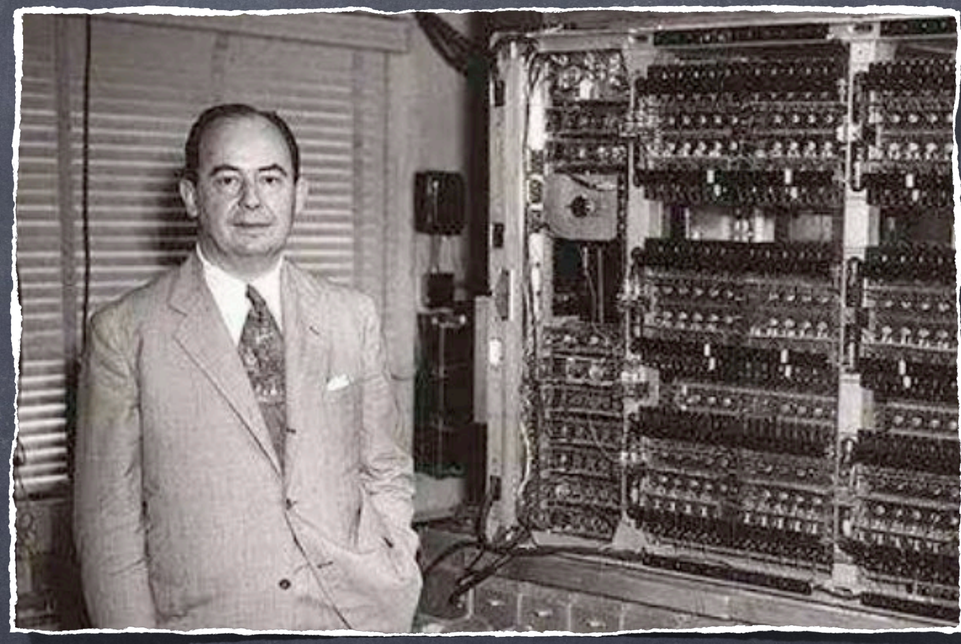
$	qk , a1

可以全局匹配了

#	(qi, ai, ai', qi') ...(qi, a0, a1, qk) ... (qj, aj, aj', qj')
o	a0 @ a1, ... ai, ai+1, ... aj+1,...

通用图灵机的意义

- 一个图灵机单凭自身就可以完成任何图灵机可能做的事情。其启发了存储式计算机的研发！
- 而且，这也是为什么我们有一个语言，就可以编功能不同的程序！



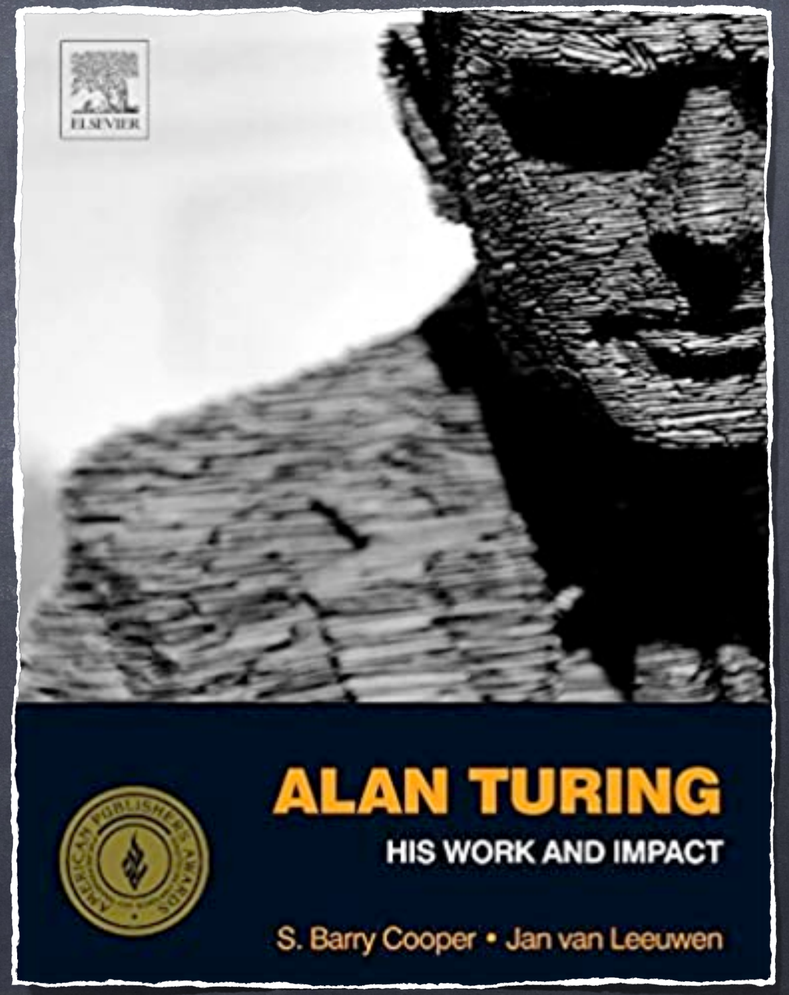
通用图灵机的意义

- 当然，通用机自身也是图灵机
  - 我们也可在通用机里模拟通用机！
  - 这也是为什么你们可以在一台宿主机的操作系统里面使用虚拟机跑windows、macos、linux！

图灵机对编程语言的启发

"Computability via Turing machines gave rise to imperative programming." – Cooper, S. B., Leeuwen, J. (2013) . Alan Turing: His Work and Impact.

"Imperative programming languages such as Fortran, Pascal etcetera as well as all the assembler languages are based on the way a Turing machine is instructed: by a sequence of statements." – Barendregt, H., Barendsen, E. (2000) . Introduction to Lambda Calculus



邱奇-图灵论题（Church-Turing thesis）

直觉上能行地可计算函数类等同图灵可计算函数类。

注意：这不是一个数学定理，而是一个信念，它也不可证。因为"能行"地计算没有精确定义。然而，不管怎样，目前没有一个反例是"可计算的"但不是图灵可计算的。

让我们再次回归停机问题

可判定性

- 让A为自然数集$\mathbb{N}$的子集，A是**可判定的**指A的特征函数χ_A是图灵可计算的。
  - 其中$\chi_A = \begin{cases} 1, & x \in A; \\ 0, & x \notin A; \end{cases}$
  - 即有图灵机M_A，其对于输入x，如果$x \in A$，那么输出1，否则输出0.

停机问题

- 定义如下两个集合：
  - $K = \{\#M : M\text{对于输入}\#M\text{停机，即}M(\#M)\text{停机}\}$
  - $K_0 = \{ < \#M, x > : \text{图灵机}M\text{对于输入}x\text{停机}\}$

哥德尔编码：
$2^{\#M}3^{x}$

停机问题

- 命题（自停机问题）：K是不可判定的。

停机问题

- 证明：我们假设存在这样的图灵机H_K可以计算χ_K，那么可以得到如下结果

	#M1	#M2	#M3	#M4	...
M1	0				
M2		1			
M3			1		
M4				1	
⋮					⋱

如果$M_i(\#Mi)$ 停机，为1，否则为0，即为$H_K(\#Mi)$

停机问题

- 证明：我们假设存在这样的图灵机H_K可以计算χ_K，那么可以得到如下结果

	#M1	#M2	#M3	#M4	...
M1	0				
M2		1			
M3			1		
M4				1	
⋮					⋱

令图灵机M_{diag}的停机与否与对角线结果相反，

$$\text{即}\begin{cases} M_{diag}(\#M_j)\text{停机}, & H_K(\#M_j)=0 \\ M_{diag}(\#M_j)\text{不停机}, & H_K(\#M_j)=1; \end{cases}$$

停机问题

- 证明：我们假设存在这样的图灵机H_K可以计算χ_K，那么可以得到如下结果

	#M1	#M2	#M3	#M4	…
M1	0				
M2		1			
M3			1		
M4				1	
⋮					⋱

令图灵机M_{diag}的停机与否与对角线结果相反，

$$\text{即}\begin{cases} M_{diag}(\#M_j)\text{停机}, & H_K(\#M_j)=0 \\ M_{diag}(\#M_j)\text{不停机}, & H_K(\#M_j)=1; \end{cases}$$

(90, 0, 0, 90) , (90, 1, 1, 90)

停机问题

- 证明：我们假设存在这样的图灵机H_K可以计算χ_K，那么可以得到如下结果

	#M1	#M2	#M3	#M4	…
M1	0				
M2		1			
M3			1		
M4				1	
⋮					⋱

那么$H_K(\#M_{diag})$?

令图灵机M_{diag}的停机与否与对角线结果相反，

即$\begin{cases} M_{diag}(\#M_j)\text{停机}, & H_K(\#M_j)=0 \\ M_{diag}(\#M_j)\text{不停机}, & H_K(\#M_j)=1; \end{cases}$

(90, 0, 0, 90) , (90, 1, 1, 90)

停机问题

- 假设使得推出矛盾，因此假设不成立，即χ_K是图灵不可计算的，既而自停机问题（集合K）不可判定。

停机问题

- 命题（停机问题）：K_0是不可判定的。

停机问题

- 证明：
  - 令χ_{K_0}为K_0的特征函数，假设存在H_{K_0}可以计算χ_{K_0}，即$\chi_{K_0}(<\#M, x>)$可以被计算为0还是1.
  - 那么有$\chi_{K_0}(<\#M, \#M>)$可以被H_{K_0}计算.
  - 然而，$\chi_K(\#M) = \chi_{K_0}(<\#M, \#M>)$，由于$\chi_K(\#M)$给不可图灵计算，因此假设不成立，$H_{K_0}$不存在。

习题

- 停机问题的另一个版本（全停机）：
  - $\hat{K} = \{\#M : 图灵机M对于所有输入停机\}$
  - 那么$\hat{K}$的可判定性如何？

停机问题

- 至此，希尔伯特的第三个问题，Entscheidungsproblem是证明不可行的。
- 除了图灵的证明外，还有一个人也独立地证明了该问题的不可行，他将是下次课的主角。

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of

归约

m-归约

- 我们称集合A可以m-归约到集合B，记为$A \leq_m B$，当存在一个全函数$g : \mathbb{N} \to \mathbb{N}$，满足对于任意$x$，$x \in A$，当且仅当$g(x) \in B$。

该函数必须图灵可计算，且对于任意x，都有定义（能够停机）。

m-归约

- 对于集合K和K_0，可以发现$K \leq_m K_0$
  - 构造函数，$g(x) = < x, x >$
  - 对于$x \in K$，当且仅当，$g(x) \in K_0$。

关于判定问题的更加一般的结论

莱斯定理（Rice theorem）

Henry Gordon Rice

指标集

其实$\mathbb{U}$可以映射到$\mathbb{N}$

- 定义$A \subseteq \mathbb{U}$为指标集（$\mathbb{U}$为全体图灵机编码集合），当且仅当如果 $\#M_1 \in A$，且M_1和M_2在任何输入下的输出都相同，那么$\#M_2 \in A$
  - 可以理解为如果两个程序功能相同，那么他们的编码（图灵编码或者哥德尔编码等）属于同一个集合。

莱斯定理

集合A可以看成具有某种性质的程序的集合（指标集限制了如果两个程序功能相同，他们必须具有相同的性质），这个定理可以解释为程序的非平凡的属性不可判定

- 对于指标集$A \subseteq \mathbb{U}$，且$A \neq \varnothing$和$A \neq \mathbb{U}$，那么A不可判定。

$$\exists x \in \mathbb{U}, x \notin A$$
$$\exists x \in \mathbb{U}, x \in A$$

莱斯定理

- 总体思路：
  - 将停机问题m一归约于到莱斯定理上。即构造一个函数$g(x)$，使得$x \in K$当且仅当$g(x) \in A$。其中K是自停机程序集合，A表示为一个非平凡的指标集（即存在属于该集合A的图灵机编号，也存在不属于该集合A的图灵机编号）。

莱斯定理

- 细节思路：
  - 情况一：
  - 由于A非平凡，不妨设图灵机编码$x_1 \in A$，设空函数$\uparrow \notin A$.
  - 那么对任何编号x，我们设M_x为其对应图灵机（即$\#M_x = x$），那我们可以构造一个图灵机M_x^e，其先运行$M_x(x)$，然后再们将串联一个图灵机编码为x_1的图灵机M_{x_1}。

M_x^e

莱斯定理

M_x^e

- 这里值的注意的是，如果$M_x(x)$运行停机，那么M_x^e等价于M_{x_1}，因为任何输入下M_x^e和M_{x_1}结果相同。根据指标集的定义，即然$x_1 \in A$（也就是$\#M_{x_1} \in A$），那么$\#M_x^e \in A$

莱斯定理

M_x^e

- 另一方面，如果$M_x(x)$运行不停机，那么M_x^e等价于↑，这是因为在任何输入下，M_x^e都不会输出任何结果并停机。

莱斯定理

- 因此，在情况一下，我们得到了一个$g(x) = \#M_x^e$，使得，$x \in K$当且仅当$g(x) \in A$。

此种情况下我们把K归约到A，因此A不可判定

莱斯定理

- 细节思路：
  - 情况二：
  - 由于A非平凡，不妨设图灵机编码$x_2 \notin A$，设空函数$\uparrow \in A$.
  - 同样地，那么对任何编号x，我们设M_x为其对应图灵机（即$\#M_x = x$），那我们可以构造一个图灵机M_x^f，其先运行$M_x(x)$，然后再们将串联一个图灵机编码为x_2的图灵机M_{x_2}。

M_x^f

莱斯定理

M_x^f

- 类似地，如果$M_x(x)$运行不停机，那么M_x^f等价于↑，那么$\#M_x^f \in A$
- 如果$M_x(x)$运行停机，那么M_x^f等价于M_{x_2}，那么$\#M_x^f \notin A$

莱斯定理

- 因此，在情况二下，我们得到了一个$g(x) = \#M_x^f$，使得，$x \in K$当且仅当$g(x) \notin A$，即当且仅当 $g(x) \in \mathbb{U} \backslash A$。

注意：此种情况我们把K归约到$\mathbb{U} \backslash A$，这里有$\mathbb{U} \backslash A$不可判定，即意味着A不可判定！因为两者的特征函数正好相反。

莱斯定理

- 因此，两种情况下，A都不可判定。

注：该证明是一个非正式的证明，比如我们省略了构造一个运行$M_x(x)$、M_{x_1}、M_{x_2}的过程，可以想象用通用机可以容易实现这两个过程（需要注意需要构造通用机所用的纸带），此外清理$M_x(x)$的运行结果也很容易，想象拷贝“oo...oo”串到通用机的纸带上。

习题

- 下面的集合可以判定吗？
  - $C = \{\#M : M$是一个全常数函数$\}$
  - $T = \{\#M : M$是一个全函数$\}$

Any questions ?