拟态语言技术手册

概念册

之恪提案

2025 年 11 月 25 日

目录

1 模拟任意状态数的数胞

1.1 介绍

在计算机中，基本上不可能有一种存储介质可以存储状态数无限大的静态数据。因此，计算机存储数据时，会选择一种存储方式，将上限确定的数据存储在有限大小的存储单元中。例如，在二进制的字长为 64 的计算机中，一次操作可以处理的数据的最大值为 18446744073709551615，即 $2^{64}-1$。但是，在不同的计算机中，进制和字长可能是不同的，进而可能导致一次操作可以处理的数据的上限不同。为了使在不同的计算机中，存在一种方式使得对数据的操作具有结果一致性，于是数胞诞生了。

为什么数胞能保证不同的计算机对数据的操作具有结果一致性呢？数胞的上限一般在程序执行前被指定为一个常数，因此它所占有的存储空间是确定的。

假设计算机 A 一次操作可以处理的数据的上限为 9，计算机 B 一次操作可以处理的数据的上限为 15，它们的存储上限与处理时的上限一致。那么当计算机 A 计算 $8+4$ 时，结果为 $12 \bmod 10 = 2$。因为其存储上限为 9，所以计算的结果需要保持在 0 到 9 内。超出此范围的结果需要进行模运算的处理。

当计算机 B 计算 $8+4$ 时，结果为 $12 \bmod 16 = 12$。因为其存储上限为 15，问题来了，现在发现相同的算式在两个计算机内计算的结果不同，怎么办?

通过指定数胞的上限，可以解决这个问题。数胞的上限相当于计算机数据的存储上限，只要设定上限为定值，那么计算机 A 和计算机 B 通过执行特定的算法，可以获得相同的结果。这样的规定相当于，无论正在运行的计算机的存储上限是多少，数胞都无视计算机的存储上限，而是使用数胞的上限获取运算结果，从而实现了对数据的操作具有结果一致性。

具体的算法将在下文详细描述。

1.2 任意进制的长除法

在计算机中，我们常常需要处理超出单个数胞表示范围的大数。虽然现代编程语言通常提供了大数运算库，但理解其底层原理——特别是如何在任意进制下进行除法运算——对于深入理解计算机算术至关重要。

长除法是一种适用于任意进制的算法，它通过迭代的方式，将复杂的除法问题分解为一系列简单的单数胞运算。这种方法的核心思想与我们在小学学习的十进制长除法完全相同，只是将其推广到了任意进制 $\mathtt{b}$。

1.2.1 数学原理与定义

考虑两个自然数：被除数 $\mathtt{D}$ 和除数 $\mathtt{d}$，其中 $\mathtt{d} \neq 0$。我们的目标是找到商 $\mathtt{Q}$ 和余数 $\mathtt{R}$，使得：

$$\mathtt{D} = \mathtt{Q} \times \mathtt{d} + \mathtt{R}, \text{其中} 0 \leq \mathtt{R} < \mathtt{d}$$

现在，假设我们不是在十进制下，而是在 $\mathtt{b}$ 进制下表示这些数字 ($\mathtt{b} \geq 2$)。那么 $\mathtt{D}$ 和 $\mathtt{d}$
可以表示为：

$$\mathtt{D} = (\mathtt{d}_{n-1}\mathtt{d}_{n-2}\dots\mathtt{d}_1\mathtt{d}_0)_{\mathtt{b}} = \sum_{i=0}^{n-1} \mathtt{d}_i \times \mathtt{b}^i$$

$\mathtt{d}$ = 除数在进制$\mathtt{b}$的表示

长除法的过程就是从被除数的高位开始，逐位 (或逐小组) 地处理，每一步计算部分被除数除以除数的结果。

1.2.2 算法步骤

以下是任意进制 $\mathtt{b}$ 下长除法的通用算法步骤：

$$\left[\begin{array}{c} \mathtt{R} := NC_\infty(D), \mathtt{Q} := NC_\infty(0) \\ \mathtt{i} := NC_\infty(0) \\ \left[\begin{array}{c} \text{当} : \mathtt{i} < \mathtt{n} \\ \mathtt{local} := \lfloor \mathtt{R} \oslash \mathtt{b}^{n-1-i} \rfloor \\ \left\{\begin{array}{ll} \mathtt{multiple} := \mathtt{local} \oslash \mathtt{d} \otimes \mathtt{d} & \text{如果} : \mathtt{local} \geq \mathtt{d} \\ \mathtt{R} :=^{\ominus} \mathtt{multiple} \otimes \mathtt{b}^{n-1-i} & \\ \mathtt{Q} :=^{\oplus} \mathtt{multiple} \otimes \mathtt{b}^{n-1-i} & \end{array}\right. \end{array}\right] \end{array}\right]$$

执行完成后，$\mathtt{Q}$ 就是长除法得到的商，$\mathtt{R}$ 就是长除法得到的余数。

在算法中，由于我们使用了 NC_∞，导致此算法无法直接表示为真正可执行的算法。因此，在实现真正的算法时，我们需要使用数组通过线性组合多个数胞来表示 $\mathtt{D}, \mathtt{d}, \mathtt{R}, \mathtt{Q}$, $\mathtt{local}$。$\oslash\mathtt{b}^{n-1-i}$ 相当于删除 $\mathtt{b}^{n-1-i}$ 个低位，将第 $\mathtt{n}-1-\mathtt{i}$ 位作为个位。$\otimes\mathtt{b}^{n-1-i}$ 相当于在原本的数后方添加 $\mathtt{n}-1-\mathtt{i}$ 个 0。它们都是位操作的模仿。

此算法展示了如何通过迭代和模运算，将复杂的大数除法分解为一系列简单的、可在单个数胞上完成的操作。这正是计算机处理超出其原生字长大数的核心思想：通过合适的算法和数据结构，用多个小单元协作完成大任务。

这个算法虽然简单，但却是理解计算机如何处理大数运算的基础。通过将复杂问题分解为可管理的步骤，我们能够在有限的硬件资源上实现任意精度的算术运算。

任意状态数的数胞的除法和取余可以通过任意进制的长除法模拟。

1.3 将数胞的值限制在状态数内

在数胞的定义中，$0 \leq$ 值 $<$ 指定状态数。那么对于一个超出范围 $[0, \text{指定状态数})$ 的值，如何让该值符合要求呢？根据定义，如果值 $\mathtt{t} \notin [0, \text{指定状态数})$，则将值 $\mathtt{t}$ 映射为 $\mathtt{t}$ mod 指定状态数。

具体算法如下：

$$\left[\begin{array}{c}\left\{\begin{array}{ll}\mathtt{t} := \mathtt{t} \text{ rem } 指定状态数 & 如果：\mathtt{t} \geq 指定状态数\end{array}\right. \\ 输出：\mathtt{t}\end{array}\right]$$

当 指定状态数 超出当前计算机的存储和计算上限时，就需要使用**任意进制的长除法**来计算 t 解决问题。具体算法中的 rem 运算就需要使用任意进制的长除法来实现。

1.4 任意进制的竖式加法

加法是最基本的算术运算。在计算机中，当处理大于单个数胞表示范围的大数时，我们需要一种系统的方法来执行加法，这种方法必须独立于计算机的原生字长和进制。任意进制的竖式加法提供了这样一种通用且高效的算法。

竖式加法的核心思想与我们小学学习的十进制竖式加法完全相同：将两个数的各位数字对齐，从最低位 (最右边) 开始逐位相加，并将产生的进位传递到更高位。此方法可以自然地推广到任意进制 $\mathtt{b}(\mathtt{b} \geq 2)$。

1.4.1 数学原理与定义

考虑两个自然数 A 和 B，它们在进制 b 下的表示分别为：

$$\mathtt{A} = (\mathtt{a}_{\mathtt{m}-1}\mathtt{a}_{\mathtt{m}-2}\ldots\mathtt{a}_1\mathtt{a}_0)_{\mathtt{b}} = \sum_{\mathtt{i}=0}^{\mathtt{m}-1} \mathtt{a}_{\mathtt{i}} \times \mathtt{b}^{\mathtt{i}}$$

$$\mathtt{B} = (\mathtt{b}_{\mathtt{n}-1}\mathtt{b}_{\mathtt{n}-2}\ldots\mathtt{b}_1\mathtt{b}_0)_{\mathtt{b}} = \sum_{\mathtt{i}=0}^{\mathtt{n}-1} \mathtt{b}_{\mathtt{i}} \times \mathtt{b}^{\mathtt{i}}$$

其中，$0 \leq \mathtt{a}_{\mathtt{i}}, \mathtt{b}_{\mathtt{i}} < \mathtt{b}$。

我们的目标是计算它们的和 $\mathtt{S} = \mathtt{A} + \mathtt{B}$，并同样用进制 b 表示结果：

$$\mathtt{S} = (\mathtt{s}_{\mathtt{k}-1}\mathtt{s}_{\mathtt{k}-2}\ldots\mathtt{s}_1\mathtt{s}_0)_{\mathtt{b}}$$

其中，$\mathtt{k} = \max(\mathtt{m}, \mathtt{n}) + 1$(考虑最高位可能的进位)。

1.4.2 算法步骤

以下是任意进制 b 下竖式加法的通用算法步骤。该算法模拟了手工计算的过程，使用一个进位变量 carry 来在每一步暂存并传递进位值。

1. **初始化**：
   - 令进位 $\mathtt{carry} := 0$。
   - 令结果位计数器 $\mathtt{i} := 0$。

- 可选：将 A 和 B 的位数补齐到相同长度 (例如，在较短的数字前补零)，设最大长度为 $\texttt{len} = \max(\texttt{m}, \texttt{n})$。

2. **逐位相加**：对于位索引 j 从 0 到 $\texttt{len} - 1$(从最低位到最高位)：

   (a) 获取当前位的数字 (若该位不存在则视为 0)：

$$\texttt{digitA} := \begin{cases} \texttt{a}_\texttt{j}, & \text{如果} \texttt{j} < \texttt{m} \\ 0, & \text{否则} \end{cases}$$

$$\texttt{digitB} := \begin{cases} \texttt{b}_\texttt{j}, & \text{如果} \texttt{j} < \texttt{n} \\ 0, & \text{否则} \end{cases}$$

   (b) 计算当前位的和 (包括低位来的进位)：

$$\texttt{sumTemp} = \texttt{digitA} + \texttt{digitB} + \texttt{carry}$$

   (c) 计算当前结果位的值和新进位：

$$\texttt{s}_\texttt{i} := \texttt{sumTemp} \bmod \texttt{b}$$

$$\texttt{carry} := \left\lfloor \frac{\texttt{sumTemp}}{\texttt{b}} \right\rfloor$$

   (d) 结果位计数器 i 增加 1。

3. **处理最高位进位**：在处理完所有位的数字后, 如果 $\texttt{carry} > 0$:

   (a) $\texttt{s}_\texttt{i} := \texttt{carry}$

   (b) $\texttt{i} := \texttt{i} + 1$

4. **得到结果**：最终结果 S 是一个由 i 位数字组成的序列 $(\texttt{s}_{\texttt{i}-1}, \texttt{s}_{\texttt{i}-2}, \ldots, \texttt{s}_0)_\texttt{b}$，其数值等于 $\texttt{A} + \texttt{B}$。

此算法的关键在于第 2(b) 步和第 2(c) 步。单个数胞可以轻松计算 digitA+digitB+carry，但其结果可能大于等于 b。利用模运算 (mod) 和整数除法 ($\lfloor\rfloor$)，我们可以将结果分解为属于 $[0, \texttt{b} - 1]$ 范围的本位值 $\texttt{s}_\texttt{i}$ 和传递给下一位的进位值 carry(其值为 0 或 1，因为最大和 $(\texttt{b} - 1) + (\texttt{b} - 1) + 1 = 2\texttt{b} - 1 < 2\texttt{b}$，故进位最大为 1)。

1.4.3 示例

示例 1(二进制，$\texttt{b} = 2$): 计算 $A = (1101)_2$(即 13) 与 $B = (1011)_2$(即 11) 的和。

		1	1	0	1	(A)
$+$		1	0	1	1	(B)
Carry:	1	1	1	1	0	
Sum (S):	1	1	0	0	0	

- 初始化：$carry = 0$。
- 第 0 位 (最低位): $1 + 1 + 0 = 2$。$s_0 = 2 \bmod 2 = 0$, $carry = \lfloor \frac{2}{2} \rfloor = 1$。
- 第 1 位: $0 + 1 + 1 = 2$。$s_1 = 2 \bmod 2 = 0$, $carry = \lfloor \frac{2}{2} \rfloor = 1$。
- 第 2 位: $1 + 0 + 1 = 2$。$s_2 = 2 \bmod 2 = 0$, $carry = \lfloor \frac{2}{2} \rfloor = 1$。
- 第 3 位: $1 + 1 + 1 = 3$。$s_3 = 3 \bmod 2 = 1$, $carry = \lfloor \frac{3}{2} \rfloor = 1$。
- 处理最高位进位: $carry = 1$，所以 $s_4 = 1$。
- 结果: $S = (11000)_2 = 16 + 8 = 24$，等于 $13 + 11$。

示例 2(十六进制，$\mathtt{b} = 16$): 计算 $A = (\mathrm{A2})_{16}$(即 162) 与 $B = (\mathrm{F9})_{16}$(即 249) 的和。(注：A=10, F=15)

		A	2	(A)
+		F	9	(B)
Carry:	1	0	0	
Sum (S):	1	9	B	

- 初始化：$carry = 0$。
- 第 0 位: $2 + 9 + 0 = 11$。$11 < 16$, 所以 $s_0 = 11 = \mathrm{B}$, $carry = 0$。
- 第 1 位: $10 + 15 + 0 = 25$。$s_1 = 25 \bmod 16 = 9$, $carry = \lfloor \frac{25}{16} \rfloor = 1$。
- 处理最高位进位: $carry = 1$，所以 $s_2 = 1$。
- 结果: $S = (19\mathrm{B})_{16} = 1 \times 256 + 9 \times 16 + 11 = 256 + 144 + 11 = 411$，等于 $162 + 249$。

任意进制的竖式加法算法是构建大数运算库的基础。通过与之前介绍的任意进制长除法相结合，可以实现在任何指定进制和数胞状态数下的完整算术运算，确保了数值计算的结果在不同计算平台和存储限制下的一致性，这正是模拟任意状态数的数胞的核心目标之一。

任意状态数的数胞的加法可以通过组合任意进制的长除法和任意进制的竖式加法这两种计算方法模拟。

1.5 任意进制的竖式减法

减法是与加法相对应的基本算术运算。在处理大数减法时，我们需要一种系统的方法来处理借位，这种方法同样必须独立于计算机的原生字长和进制。任意进制的竖式减法提供了这样一种通用且高效的算法。

竖式减法的核心思想与我们小学学习的十进制竖式减法完全相同：将两个数的各位数字对齐，从最低位 (最右边) 开始逐位相减，并在需要时向高位借位。此方法可以自然地推广到任意进制 $\mathtt{b}(\mathtt{b} \geq 2)$。

1.5.1 数学原理与定义

考虑两个非负大整数 A 和 B，且 $A \geq B$(确保结果非负)。它们在进制 b 下的表示分别为:

$$A = (a_{m-1}a_{m-2}\ldots a_1a_0)_b = \sum_{i=0}^{m-1} a_i \times b^i$$

$$B = (b_{n-1}b_{n-2}\ldots b_1b_0)_b = \sum_{i=0}^{n-1} b_i \times b^i$$

其中，$0 \leq a_i, b_i < b$。

我们的目标是计算它们的差 $D = A - B$，并同样用进制 b 表示结果:

$$D = (d_{k-1}d_{k-2}\ldots d_1d_0)_b$$

其中，$k \leq \max(m, n)$(结果的位数可能小于原数的位数)。

1.5.2 算法步骤

以下是任意进制 b 下竖式减法的通用算法步骤。该算法模拟了手工计算的过程，使用一个借位变量 borrow 来在每一步暂存借位状态。

1. **初始化**:
   - 令借位 $borrow := 0$。
   - 令结果位计数器 $i := 0$。
   - 将 A 和 B 的位数补齐到相同长度 (在较短的数字前补零)，设最大长度为 $len = \max(m, n)$。
2. **逐位相减**: 对于位索引 j 从 0 到 $len - 1$(从最低位到最高位):
   (a) 获取当前位的数字 (若该位不存在则视为 0):

$$digitA := \begin{cases} a_j, & \text{如果} j < m \\ 0, & \text{否则} \end{cases}$$

$$digitB := \begin{cases} b_j, & \text{如果} j < n \\ 0, & \text{否则} \end{cases}$$

   (b) 计算当前位的临时差 (考虑借位):

$$tempDiff = digitA - digitB - borrow$$

(c) 处理借位并计算当前结果位：

如果$\texttt{tempDiff} < 0$ 则

$\texttt{borrow} := 1$

$\texttt{d}_\texttt{i} := \texttt{tempDiff} + \texttt{b}$

否则

$\texttt{borrow} := 0$

$\texttt{d}_\texttt{i} := \texttt{tempDiff}$

(d) 结果位计数器 i 增加 1。

3. **处理最高位借位**：在处理完所有位后，如果 $\texttt{borrow} > 0$，则说明 $\texttt{A} < \texttt{B}$，这与我们的前提 $\texttt{A} \geq \texttt{B}$ 矛盾。在实际实现中，这一步可用于检测错误或处理负数情况。

4. **去除前导零**：结果可能包含高位多余的零，需要去除这些零以得到规范形式。

5. **得到结果**：最终结果 D 是一个由 i 位数字组成的序列 $(\texttt{d}_{\texttt{i}-1}, \texttt{d}_{\texttt{i}-2}, \ldots, \texttt{d}_0)_\texttt{b}$，其数值等于 $\texttt{A} - \texttt{B}$。

1.5.3 示例

示例 1(二进制，$\texttt{b} = 2$): 计算 $A = (1101)_2$(即 13) 与 $B = (1011)_2$(即 11) 的差。

		1	1	0	1	(A)
-		1	0	1	1	(B)
Borrow:	0	0	1	0	0	
Diff (D):		0	0	1	0	

- 初始化：$\texttt{borrow} = 0$。
- 第 0 位 (最低位): $1 - 1 - 0 = 0$。$\texttt{d}_0 = 0$, $\texttt{borrow} = 0$。
- 第 1 位: $0 - 1 - 0 = -1$。$\texttt{tempDiff} < 0$, 所以 $\texttt{borrow} = 1$, $\texttt{d}_1 = -1 + 2 = 1$。
- 第 2 位: $1 - 0 - 1 = 0$。$\texttt{d}_2 = 0$, $\texttt{borrow} = 0$。
- 第 3 位: $1 - 1 - 0 = 0$。$\texttt{d}_3 = 0$, $\texttt{borrow} = 0$。
- 结果: $D = (0010)_2 = 2$，等于 $13 - 11$。去除前导零后为 $(10)_2$。

示例 2(十六进制，$\texttt{b} = 16$): 计算 $A = (\text{A2})_{16}$(即 162) 与 $B = (\text{7C})_{16}$(即 124) 的差。

		A	2	(A)
-		7	C	(B)
Borrow:	0	1	0	
Diff (D):		2	6	

- 初始化：$\texttt{borrow} = 0$。
- 第 0 位: $2 - 12 - 0 = -10$。$\texttt{tempDiff} < 0$, 所以 $\texttt{borrow} = 1$, $\texttt{d}_0 = -10 + 16 = 6$。
- 第 1 位: $10 - 7 - 1 = 2$。$\texttt{d}_1 = 2$, $\texttt{borrow} = 0$。
- 结果: $D = (26)_{16} = 2 \times 16 + 6 = 38$，等于 $162 - 124$。

任意进制的竖式减法算法与竖式加法相结合，构成了完整的大数算术运算基础。通过正确处理借位和进制转换，该算法确保了在不同计算平台和存储限制下减法运算的一致性，进一步支持了模拟任意上限数胞的目标。

任意状态数的数胞的减法可以通过组合任意进制的长除法和任意进制的竖式减法这两种计算方法模拟。

1.6 任意进制的竖式乘法

乘法是算术运算中的重要组成部分，它比加法和减法更为复杂。在处理大数乘法时，我们需要一种系统的方法来逐位相乘并处理进位。任意进制的竖式乘法提供了一种通用且高效的算法，可以处理任意进制下的乘法运算。

竖式乘法的核心思想与我们小学学习的十进制竖式乘法相同：将一个乘数的每一位与另一个乘数的每一位相乘，然后将结果按位对齐并相加。此方法可以自然地推广到任意进制 $\texttt{b}(\texttt{b} \geq 2)$。

1.6.1 数学原理与定义

考虑两个非负大整数 $\texttt{A}$ 和 $\texttt{B}$，它们在进制 $\texttt{b}$ 下的表示分别为：

$$\texttt{A} = (\texttt{a}_{\texttt{m}-1}\texttt{a}_{\texttt{m}-2}\ldots\texttt{a}_1\texttt{a}_0)_\texttt{b} = \sum_{\texttt{i}=0}^{\texttt{m}-1} \texttt{a}_\texttt{i} \times \texttt{b}^\texttt{i}$$

$$\texttt{B} = (\texttt{b}_{\texttt{n}-1}\texttt{b}_{\texttt{n}-2}\ldots\texttt{b}_1\texttt{b}_0)_\texttt{b} = \sum_{\texttt{j}=0}^{\texttt{n}-1} \texttt{b}_\texttt{j} \times \texttt{b}^\texttt{j}$$

其中，$0 \leq \texttt{a}_\texttt{i}, \texttt{b}_\texttt{j} < \texttt{b}$。

我们的目标是计算它们的乘积 $\texttt{P} = \texttt{A} \times \texttt{B}$，并同样用进制 $\texttt{b}$ 表示结果：

$$\texttt{P} = (\texttt{p}_{\texttt{k}-1}\texttt{p}_{\texttt{k}-2}\ldots\texttt{p}_1\texttt{p}_0)_\texttt{b}$$

其中，$\texttt{k} \leq \texttt{m} + \texttt{n}$(乘积的位数最多为两个乘数位数之和)。

1.6.2 算法步骤

以下是任意进制 $\mathtt{b}$ 下竖式乘法的通用算法步骤。该算法模拟了手工计算的过程，使用一个中间结果数组来存储部分乘积。

1. **初始化**：
   - 创建一个长度为 $\mathtt{m}+\mathtt{n}$ 的结果数组 $\mathtt{result}$，初始化为全零。
   - 这个数组将用于存储乘法过程中的所有中间结果和最终乘积。
2. **逐位相乘**：对于乘数 $\mathtt{B}$ 的每一位索引 $\mathtt{j}$ 从 0 到 $\mathtt{n}-1$(从最低位到最高位)：
   (a) 初始化进位 $\mathtt{carry} := 0$。
   (b) 对于乘数 $\mathtt{A}$ 的每一位索引 $\mathtt{i}$ 从 0 到 $\mathtt{m}-1$(从最低位到最高位)：
      i. 计算当前位的乘积：$\mathtt{product} = \mathtt{a_i} \times \mathtt{b_j} + \mathtt{result}[\mathtt{i}+\mathtt{j}] + \mathtt{carry}$
      ii. 计算当前位的值和新进位：

$$\mathtt{result}[\mathtt{i}+\mathtt{j}] := \mathtt{product} \bmod \mathtt{b}$$
$$\mathtt{carry} := \lfloor \frac{\mathtt{product}}{\mathtt{b}} \rfloor$$

   (c) 处理最高位进位：如果 $\mathtt{carry} > 0$，则将其加到结果数组的下一个位置：

$$\mathtt{result}[\mathtt{j}+\mathtt{m}] := \mathtt{result}[\mathtt{j}+\mathtt{m}] + \mathtt{carry}$$

3. **处理最终进位**：遍历整个结果数组，从低位到高位处理可能的进位：
   (a) 对于每个位置 $\mathtt{k}$ 从 0 到 $\mathtt{m}+\mathtt{n}-2$：

$$\mathtt{carry} := \lfloor \frac{\mathtt{result}[\mathtt{k}]}{\mathtt{b}} \rfloor$$
$$\mathtt{result}[\mathtt{k}] := \mathtt{result}[\mathtt{k}] \bmod \mathtt{b}$$
$$\mathtt{result}[\mathtt{k}+1] := \mathtt{result}[\mathtt{k}+1] + \mathtt{carry}$$

4. **去除前导零**：从最高位开始，去除结果数组中的前导零，得到规范形式的乘积。
5. **得到结果**：最终结果 $\mathtt{P}$ 是一个由非零位数字组成的序列，其数值等于 $\mathtt{A} \times \mathtt{B}$。

1.6.3 示例

示例 1(二进制，$\mathtt{b}=2$): 计算 $A = (1101)_2$(即 13) 与 $B = (1011)_2$(即 11) 的乘积。

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & & \\
 & 1 & 1 & 0 & 1 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
$$

- 初始化：创建长度为 $4+4=8$ 的结果数组，初始为 $[0,0,0,0,0,0,0,0]$。
- 第 0 轮 ($\texttt{j}=0$, $\texttt{b}_0=1$):
  - $\texttt{i}=0$: $1\times 1+0+0=1$, $\texttt{result}[0]=1$, $\texttt{carry}=0$
  - $\texttt{i}=1$: $0\times 1+0+0=0$, $\texttt{result}[1]=0$, $\texttt{carry}=0$
  - $\texttt{i}=2$: $1\times 1+0+0=1$, $\texttt{result}[2]=1$, $\texttt{carry}=0$
  - $\texttt{i}=3$: $1\times 1+0+0=1$, $\texttt{result}[3]=1$, $\texttt{carry}=0$
- 第 1 轮 ($\texttt{j}=1$, $\texttt{b}_1=1$):
  - $\texttt{i}=0$: $1\times 1+0+0=1$, $\texttt{result}[1]=0+1=1$, $\texttt{carry}=0$
  - $\texttt{i}=1$: $0\times 1+0+0=0$, $\texttt{result}[2]=1+0=1$, $\texttt{carry}=0$
  - $\texttt{i}=2$: $1\times 1+0+0=1$, $\texttt{result}[3]=1+1=2$, $2 \bmod 2=0$, $\lfloor 2/2\rfloor=1$
  - $\texttt{i}=3$: $1\times 1+0+1=2$, $\texttt{result}[4]=0+2=2$, $2 \bmod 2=0$, $\lfloor 2/2\rfloor=1$
  - 处理进位: $\texttt{result}[5]=0+1=1$
- 继续处理剩余位...
- 最终结果: $P=(10001111)_2=128+8+4+2+1=143$，等于 13×11。

示例 2(十进制，$\texttt{b}=10$): 计算 $A=123$ 与 $B=45$ 的乘积。

$$
\begin{array}{cccccc}
 & & 1 & 2 & 3 & (A) \\
 & \times & 4 & 5 & & (B) \\
\hline
 & & 6 & 1 & 5 & (123\times 5) \\
+ & 4 & 9 & 2 & & (123\times 4,\ \text{左移 1 位}) \\
\hline
 & 5 & 5 & 3 & 5 & (P)
\end{array}
$$

- 第 0 轮 ($\texttt{j}=0$, $\texttt{b}_0=5$):
  - $3\times 5=15$, $\texttt{result}[0]=5$, $\texttt{carry}=1$

  - $2 \times 5 + 1 = 11$, $\texttt{result}[1] = 1$, $\texttt{carry} = 1$
  - $1 \times 5 + 1 = 6$, $\texttt{result}[2] = 6$, $\texttt{carry} = 0$

- 第 1 轮 ($\texttt{j} = 1$, $\texttt{b}_1 = 4$):
  - $3 \times 4 + 0 = 12$, $\texttt{result}[1] = 1 + 2 = 3$, $\texttt{carry} = 1$
  - $2 \times 4 + 1 = 9$, $\texttt{result}[2] = 6 + 9 = 15$, $15 \bmod 10 = 5$, $\lfloor 15/10 \rfloor = 1$
  - $1 \times 4 + 1 = 5$, $\texttt{result}[3] = 0 + 5 = 5$, $\texttt{carry} = 0$

- 最终结果: $P = 5535$，等于 123×45。

任意进制的竖式乘法算法是构建完整大数算术运算库的关键组成部分。通过正确处理每一位的乘法和进位，该算法确保了在不同计算平台和存储限制下乘法运算的一致性。与之前介绍的竖式加法和竖式减法相结合，我们可以实现完整的任意进制大数运算，为模拟任意上限的数胞提供了坚实的基础。

需要注意的是，竖式乘法的时间复杂度为 $O(\texttt{m} \times \texttt{n})$，其中 $\texttt{m}$ 和 $\texttt{n}$ 分别是两个乘数的位数。对于非常大的数，可以使用更高效的算法来优化性能，但竖式乘法作为最基础和最直观的算法，仍然是理解乘法原理的重要工具。

任意状态数的数胞的乘法可以通过组合任意进制的长除法和任意进制的竖式乘法这两种计算方法模拟。

1.7 使用伪代码模拟任意状态数的数胞

从现在开始，我们将使用详细的、尽可能高效的、但可能不是最简单的、可能与之前所述简单方法有差别的算法来模拟任意状态数的数胞。

首先，我们需要一些用于辅助的结构和算法来帮助我们完成编写。

1.7.1 视图

$$\begin{bmatrix} \text{结构体} : SPAN\{\texttt{T}\} \\ \texttt{p} \in PTR_{\texttt{T}} \\ \texttt{l} \in NC_{\texttt{WORDSIZEVALUE}} \end{bmatrix}$$

$$\text{定义} : SPAN_{\texttt{T}} = SPAN\{\texttt{T}\}$$

我们需要一个结构，用于关注一个线性存储空间的一部分。例如，有一个数组，我们可以取这个数组中某个位置的子数组作为视图。

$$\text{定义} : \text{GetElement}\{\texttt{T}\}(\texttt{span} \in SPAN_{\texttt{T}}, \texttt{index} \in NC_{\texttt{WORDSIZEVALUE}}) \to \texttt{T} =$$

$$\Big[\text{输出} : (\texttt{span} \to p)[\texttt{index}] \quad \blacksquare\text{要求满足}\texttt{index} < \texttt{l}\Big]$$

$$\text{定义}:\texttt{span}[\texttt{index}]=\text{GetElement}\{\texttt{T}\}(\texttt{span},\texttt{index}),$$

$$\texttt{span}\in SPAN_{\texttt{T}},\texttt{index}\in NC_{\texttt{WORDSIZEVALUE}}$$

这个算法及这些定义简化了从一个视图中获取一个元素的步骤。

$$\text{定义}:\text{SubSpan}(\texttt{span}\in SPAN_{\texttt{T}},\texttt{index}\in NC_{\texttt{WORDSIZEVALUE}},\texttt{length}\in NC_{\texttt{WORDSIZEVALUE}})$$

$$\to SPAN_{\texttt{T}}=\quad\blacksquare\text{要求满足}\texttt{index}\oplus\texttt{length}\le\texttt{span}\to\texttt{l}$$

$$\begin{bmatrix}\texttt{newspan}\in SPAN_{\texttt{T}}\\ \texttt{newspan}\to p:=(\texttt{span}\to p)\boxplus\texttt{index}\\ \texttt{newspan}\to l:=\texttt{length}\\ \text{输出}:\texttt{newspan}\end{bmatrix}$$

$$\text{定义}:\texttt{span}\to\text{subspan}(\texttt{index},\texttt{length})=\text{SubSpan}\{\texttt{T}\}(\texttt{span},\texttt{index},\texttt{length}),$$

$$\texttt{span}\in SPAN_{\texttt{T}},\texttt{index}\in NC_{\texttt{WORDSIZEVALUE}},\texttt{length}\in NC_{\texttt{WORDSIZEVALUE}}$$

这个算法及这些定义简化了从一个视图中截取一个子视图的步骤。

1.7.2 数组形式的模拟数胞之间的比较

因为任意状态数的数胞需要的存储空间可能大于 $NC_{\texttt{WORDSIZEVALUE}}$ 所占的存储空间，所以我们需要使用数组来模拟数胞。以确保任意大的状态数都可以进行存储和计算。

那么，这种数组形式的模拟数胞之间的比较算法，就是比较数组有效位的长度，或在有效位长度一致时从高位到低位，对数组进行比较。

下面我们实现数组形式的模拟数胞之间相等判断和小于判断，其余比较算法同理，不做实现。

$$\text{定义}:\text{IsEqual}\{\texttt{T}\}(\texttt{left}\in PTR_{\texttt{T}},\texttt{ll}\in NC_{\texttt{WORDSIZEVALUE}},$$

$$\texttt{right}\in PTR_{\texttt{T}},\texttt{lr}\in NC_{\texttt{WORDSIZEVALUE}})\to NC_2=$$

$$\begin{bmatrix}\blacksquare\text{要求满足}\texttt{ll}>0,\texttt{lr}>0\\ \begin{cases}\text{输出}:\text{boolvalue}(\mathbf{F})\quad\text{如果}:\texttt{ll}\neq\texttt{lr}\\ \blacksquare\text{要求满足}\texttt{left},\texttt{right}\text{都没有无效位，当有效位长度不一致时，一定不相等}\end{cases}\\ \blacksquare\text{此时长度一定一致，逐个比较高位到低位}\\ \texttt{i}:=NC_{\texttt{WORDSIZEVALUE}}(\texttt{ll}\ominus 1)\\ \begin{bmatrix}\begin{cases}\text{输出}:\text{boolvalue}(\mathbf{F})\quad\text{如果}:\texttt{left}[\texttt{i}]\neq\texttt{right}[\texttt{i}]\\ \blacksquare\text{当任何对应位不相等，}\texttt{left},\texttt{right}\text{就不相等}\\ \texttt{i}:=^{\oplus}1\end{cases}\end{bmatrix}\\ \text{输出}:\text{boolvalue}(\mathbf{T})\quad\blacksquare\text{此时满足}\texttt{left}=\texttt{right}\end{bmatrix}$$

$$\text{定义}:\text{IsLess}\{\texttt{T}\}(\texttt{left}\in PTR_{\texttt{T}},\texttt{ll}\in NC_{\texttt{WORDSIZEVALUE}},$$
$$\texttt{right}\in PTR_{\texttt{T}},\texttt{lr}\in NC_{\texttt{WORDSIZEVALUE}})\to NC_2=$$
$$\left[\begin{array}{c}\blacksquare\text{要求满足}\texttt{ll}>0,\texttt{lr}>0\\ \left\{\begin{array}{l}\text{输出}:\text{boolvalue}(\mathbf{T})\quad\text{如果}:\texttt{ll}<\texttt{lr}\\ \text{输出}:\text{boolvalue}(\mathbf{F})\quad\text{如果}:\texttt{ll}>\texttt{lr}\\ \blacksquare\text{要求满足}\texttt{left},\texttt{right}\text{都没有无效位，当有效位长度不一致时，一定不相等}\\ \blacksquare\text{此时长度大的就是大数，长度小的就是小数}\end{array}\right.\\ \blacksquare\text{此时长度一定一致，逐个比较高位到低位}\\ \texttt{i}:=NC_{\texttt{WORDSIZEVALUE}}(\texttt{ll}\ominus 1)\\ \left[\left\{\begin{array}{l}\text{输出}:\text{boolvalue}(\mathbf{T})\quad\text{如果}:\texttt{left}[\texttt{i}]<\texttt{right}[\texttt{i}]\\ \text{输出}:\text{boolvalue}(\mathbf{F})\quad\text{如果}:\texttt{left}[\texttt{i}]>\texttt{right}[\texttt{i}]\\ \blacksquare\text{当任何对应位不相等，}\texttt{left},\texttt{right}\text{就不相等}\\ \blacksquare\text{此时此位，大的就是大数，小的就是小数}\texttt{i}:=^{\oplus}1\end{array}\right.\right]\\ \text{输出}:\text{boolvalue}(\mathbf{F})\quad\blacksquare\text{此时满足}\texttt{left}=\texttt{right}\end{array}\right]$$

1.7.3 任意进制的长除法的具体算法

接下来，我们给出任意进制的长除法的具体算法。

$$\text{定义}:\text{SubBetweenHugeValue}(\texttt{minuend}\in SPAN_{NC_{\texttt{BASE}}},\texttt{subtrahend}\in SPAN_{NC_{\texttt{BASE}}})$$

设minuend所占存储空间中的自然数为m, subtrahend所占存储空间中的自然数为s

执行之后，m变为减去数次s的结果，结果r满足$\texttt{r}>0\wedge\texttt{r}<\texttt{s}$

这个算法用于在长除法中进行减法。

$$\text{定义}:\text{Cast}\{\texttt{T},\texttt{t}\}(\texttt{p}\in PTR_{\texttt{T}})\to PTR_{\texttt{t}}$$

用于将一个T类型的指针变量p转换成一个t类型的指针变量输出，其指向的地址不变

$$\text{定义}:\text{LongMod}(\texttt{dividend}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}},\texttt{divisor}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}})$$
$$\to DA_{NC_{\texttt{WORDSIZEVALUE}}}=$$

$$
\left[\begin{array}{c}
\texttt{dividendbitspan} \in SPAN_{\texttt{BASE}} \\
\texttt{dividendbitspan} \to p := \text{Cast}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{BASE}}\}(\texttt{dividend} \to p) \\
\texttt{dividendbitspan} \to l := (\texttt{dividend} \to l) \otimes \texttt{BYTELENGTH} \otimes \texttt{WORDSIZE} \\
\texttt{divisorbitspan} \in SPAN_{\texttt{BASE}} \\
\texttt{divisorbitspan} \to p := \text{Cast}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{BASE}}\}(\texttt{divisor} \to p) \\
\texttt{divisorbitspan} \to l := (\texttt{divisor} \to l) \otimes \texttt{BYTELENGTH} \otimes \texttt{WORDSIZE} \\
\blacksquare\text{将输入的视图转换成位视图，要求满足} \texttt{dividend} \to l > 0 \wedge \texttt{divisor} \to l > 0 \\
\texttt{i} := NC_{\texttt{WORDSIZEVALUE}}((\texttt{dividendbitspan} \to l) \ominus 1) \\
\left[\begin{array}{c}
\text{当} : (\texttt{dividendbitspan} \to p)[\texttt{i}] = 0 \wedge \texttt{i} > 0 \\
(\texttt{dividendbitspan} \to l) :=^{\ominus} 1 \\
\texttt{i} :=^{\ominus} 1
\end{array}\right] \\
\texttt{i} := (\texttt{divisorbitspan} \to l) \ominus 1 \\
\left[\begin{array}{c}
\text{当} : (\texttt{divisorbitspan} \to p)[\texttt{i}] = 0 \wedge \texttt{i} > 0 \\
(\texttt{divisorbitspan} \to l) :=^{\ominus} 1 \\
\texttt{i} :=^{\ominus} 1
\end{array}\right] \\
\blacksquare\text{为两个位视图去除无效位（值为 0 的视图保留一位）} \\
\left\{\begin{array}{l}
\text{如果} : \texttt{dividendbitspan} \to l < \texttt{divisorbitspan} \to l \\
\texttt{result} :=\uparrow DA_{NC_{\texttt{WORDSIZEVALUE}}}(\texttt{dividend} \to l) \\
\texttt{i} := 0 \\
\left[\begin{array}{c}
\text{当} : \texttt{i} < \texttt{dividend} \to l \\
\texttt{result}[\texttt{i}] := (\texttt{dividend} \to p)[\texttt{i}] \\
\texttt{i} :=^{\oplus} 1
\end{array}\right] \\
\text{输出} : \texttt{result} \\
\blacksquare\text{如果能确定被除数小于除数，那么直接输出被除数，不用进行计算}
\end{array}\right. \\
\text{输出} : \text{LongModProc}(\texttt{dividendbitspan}, \texttt{divisorbitspan})
\end{array}\right]
$$

$$
\text{定义} : \text{Copy}\{\texttt{T}, \texttt{t}\}(\texttt{P} \in PTR_{\texttt{T}}, \texttt{p} \in PTR_{\texttt{t}}, \texttt{length} \in NC_{\texttt{WORDSIZEVALUE}})
$$

执行之后P所指向的存储空间中的连续length个字节被更改为

p所指向的存储空间中的连续length个字节

$$
\text{定义} : \text{LongModProc}(\texttt{dividendbitspan} \in SPAN_{NC_{\texttt{BASE}}},
$$

$$
\texttt{divisorbitspan} \in SPAN_{NC_{\texttt{BASE}}}) \to DA_{NC_{\texttt{WORDSIZEVALUE}}} =
$$

$$
\left[\begin{array}{c}
\texttt{i} := NC_{\texttt{WORDSIZEVALUE}}(0) \\
\left[\begin{array}{c}
\text{当} : \texttt{i} < \texttt{dividendbitspan} \to l \\
\texttt{startindex} := (\texttt{dividendbitspan} \to l) \ominus 1 \ominus \texttt{i} \\
\texttt{length} := \texttt{i} \oplus 1 \\
\texttt{subdividend} := \texttt{dividendbitspan} \to \text{subspan}(\texttt{startindex}, \texttt{length}) \\
\blacksquare\texttt{subdividend}\text{是长除法中关注的余数的子位视图} \\
\texttt{j} := NC_{\texttt{WORDSIZEVALUE}}((\texttt{subdividend} \to l) \ominus 1) \\
\left[\begin{array}{c}
\text{当} : (\texttt{subdividend} \to p)[\texttt{j}] = 0 \wedge \texttt{j} > 0 \\
(\texttt{subdividend} \to l) :=^{\ominus} 1 \\
\texttt{j} :=^{\ominus} 1
\end{array}\right] \\
\blacksquare\text{为子位视图去除无效位（值为 0 的视图保留一位）} \\
\left\{\begin{array}{l}
\text{如果} : \text{IsLess}\{NC_{\texttt{BASE}}\}(\texttt{subdividend} \to p, \texttt{subdividend} \to l, \\
\texttt{divisorbitspan} \to p, \texttt{divisorbitspan} \to l) \\
\text{SubBetweenHugeValue}(\texttt{subdividend}, \texttt{divisorbitspan}) \\
\blacksquare\text{如果子位视图小于除数，那么进行相减}
\end{array}\right. \\
\texttt{i} :=^{\oplus} 1
\end{array}\right] \\
\texttt{remaindersize} := (\texttt{dividendbitspan} \to l) \oslash (\texttt{BYTELENGTH} \otimes \texttt{WORDSIZE}) \\
\texttt{remainder} :=\uparrow DA_{NC_{\texttt{WORDSIZEVALUE}}}(\texttt{remaindersize}) \\
\text{Copy}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{BASE}}\}(\texttt{remainder} \to addr, \texttt{dividendbitspan} \to p, \\
\texttt{remaindersize}) \\
\texttt{i} := \texttt{remaindersize} \ominus 1 \\
\left[\begin{array}{c}
\text{当} : \texttt{remainder}[\texttt{i}] = 0 \wedge \texttt{i} > 0 \\
\texttt{remainder} \to \text{remove}() \\
\texttt{i} :=^{\ominus} 1
\end{array}\right] \\
\blacksquare\text{初始化存储余数的数组，并去除无效位（值为 0 时保留一位）} \\
\text{输出} : \texttt{remainder}
\end{array}\right]
$$

以上算法使用任意进制的长除法求余数。但是求商也是一样的，初始化一个长度与被除数数组相同的数组，然后把它转换成位视图。在 subdividend 的值小于 divisorbitspan 时，记录 subdividend ÷ divisorbitspan 的整除值，然后把整除值放在代表商的位视图从 0 开始的第 (dividendbitspan → l) ⊖ 1 ⊖ i 位上。最后去除无效位就可以输出得到商了。求商的算法将不进行详细描述，直接略过。

1.7.4 任意状态数的数胞的加法

接下来，我们给出任意状态数的数胞的加法的具体算法。

定义 : PropagateCarryInIncrease(parts $\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}}$, carry $\in NC_2$) $\to NC_2$

执行这个算法之后，parts所表示的自然数变为原来的值加上carry的值，

如果溢出，则新值为 0，并输出 1，否则输出 0

定义 : LimitValueAfterIncrease(n $\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}}$, v $\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}}$),

执行这个算法之后，v所表示的自然数变为原来的值减去n的值，

如果溢出，则不需要特殊处理，由硬件解释计算溢出的结果

定义 : IsOverflow() $\rightarrow NC_2$,

这个算法判断上一条伪代码语句是否产生了计算溢出，并输出为布尔值

定义 : LimitRightValueThenIncrease(left $\in DA_{NC_{\texttt{WORDSIZEVALUE}}}$, right $\in DA_{NC_{\texttt{WORDSIZEVALUE}}}$,

n $\in DA_{NC_{\texttt{WORDSIZEVALUE}}}$) =

$$\left[\begin{array}{c}
\texttt{copyright} :=\uparrow DA_{NC_{\texttt{WORDSIZEVALUE}}}(\texttt{right} \rightarrow l) \\
\text{Copy}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{WORDSIZEVALUE}}\}(\texttt{copyright} \rightarrow addr, \\
\texttt{right} \rightarrow addr, (\texttt{right} \rightarrow l) \otimes \texttt{WORDSIZE}) \\
\texttt{copyrightspan} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{copyrightspan} \rightarrow p := \texttt{copyright} \rightarrow addr \\
\texttt{copyrightspan} \rightarrow l := \texttt{right} \rightarrow l \\
\blacksquare\text{将right指向的数据复制到copyright，} \\
\text{防止使用LongMod时更改right指向的数据} \\
\texttt{nspan} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{nspan} \rightarrow p := \texttt{n} \rightarrow addr \\
\texttt{nspan} \rightarrow l := \texttt{n} \rightarrow l \\
\texttt{remainder} := \text{LongMod}(\texttt{copyrightspan}, \texttt{nspan}) \\
\text{Add}(\texttt{left}, \texttt{remainder}) \\
\downarrow \texttt{remainder} \\
\downarrow \texttt{copyright} \\
\blacksquare\text{如果right的值大于n，则将right的值限制在n的范围内}
\end{array}\right]$$

定义 : Add(left $\in DA_{NC_{\texttt{WORDSIZEVALUE}}}$, right $\in DA_{NC_{\texttt{WORDSIZEVALUE}}}$, n $\in DA_{NC_{\texttt{WORDSIZEVALUE}}}$) =

$$
\left[\begin{array}{c}
\blacksquare\text{此算法会将}\texttt{right}\text{的值加到}\texttt{left}\text{上，并限制在状态数}\texttt{n}\text{的范围内} \\
\blacksquare\text{要求满足}\texttt{right},\texttt{n}\text{皆没有无效位，且}\texttt{left}\text{的长度与}\texttt{n}\text{相同} \\
\left\{\begin{array}{l}
\text{如果}:\text{IsGreaterOrEqual}(\texttt{right}\to addr,\texttt{right}\to l,\texttt{n}\to addr,\texttt{n}\to l) \\
\blacksquare\text{IsGreaterOrEqual并未给出实现，请参考之前有关模拟数胞之间的比较} \\
\text{LimitRightValueThenIncrease}(\texttt{left},\texttt{right},\texttt{n}) \\
\blacksquare\text{这个分支确保之后Add执行时}\texttt{right}\text{指向的值小于}\texttt{n}
\end{array}\right. \\
\texttt{i} := NC_{\texttt{WORDSIZEVALUE}}(0) \\
\texttt{carry} := NC_2(0) \\
\left[\begin{array}{c}
\text{当}:\texttt{i}<\texttt{left}\to l\wedge\texttt{i}<\texttt{right}\to l \\
\texttt{copycarry} := NC_{\texttt{WORDSIZEVALUE}}(0) \\
\texttt{copycarry} :=^{\oplus} \texttt{carry} \\
\texttt{left[i]} := \texttt{left[i]}\oplus\texttt{right[i]}\oplus\texttt{copycarry} \\
\texttt{carry} := \text{IsOverflow}() \\
\texttt{i} :=^{\oplus} 1
\end{array}\right] \\
\left\{\begin{array}{l}
\text{如果}:\texttt{carry}=1\wedge\texttt{i}<\texttt{left}\to l\wedge\texttt{i}\geq\texttt{right}\to l \\
\texttt{parts}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{parts}\to p := \texttt{left}\to addr \\
\texttt{parts}\to l := \texttt{left}\to l \\
\texttt{parts}\to p := \texttt{parts}\to p\boxplus\texttt{i} \\
\texttt{patys}\to l :=^{\ominus} \texttt{i} \\
\texttt{carry} := \text{PropagateCarryInIncrease}(\texttt{parts},\texttt{carry}) \\
\blacksquare\text{如果}\texttt{right}\text{的长度小于}\texttt{left}\text{的长度，且进位不为 0，} \\
\text{则将进位处理到}\texttt{left}\text{的相应位置}
\end{array}\right. \\
\left\{\begin{array}{l}
\text{如果}:\texttt{carry}=1\vee\text{IsGreaterOrEqual}(\texttt{left}\to addr,\texttt{left}\to l,\texttt{n}\to addr,\texttt{n}\to l) \\
\texttt{nspan}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{nspan}\to p := \texttt{n}\to addr \\
\texttt{nspan}\to l := \texttt{n}\to l \\
\texttt{vspan}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{vspan}\to p := \texttt{left}\to addr \\
\texttt{vspan}\to l := \texttt{left}\to l \\
\text{LimitValueAfterIncrease}(\texttt{nspan},\texttt{vspan}) \\
\blacksquare\text{如果最后产生进位，或没有进位但是值相对状态数溢出，进行值的修正}
\end{array}\right.
\end{array}\right]
$$

1.7.5 任意状态数的数胞的减法

接下来，我们给出任意状态数的数胞的减法的具体算法。

定义：PropagateBorrowInDecrease($\texttt{parts} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{borrow} \in NC_2) \to NC_2$

执行这个算法之后，parts所表示的自然数变为原来的值减去borrow的值，

如果溢出，则新值为$0 \ominus 1$，并输出 1，否则输出 0

定义：LimitValueAfterDecrease($\texttt{n} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{v} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}}$)，

执行这个算法之后，v所表示的自然数变为原来的值加上n的值，

如果溢出，则不需要特殊处理，由硬件解释计算溢出的结果

定义：LimitRightValueThenIncrease($\texttt{left} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{right} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}$,

$\texttt{n} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}$) =

$$\left[\begin{array}{c}
\texttt{copyright} :=\uparrow DA_{NC_{\texttt{WORDSIZEVALUE}}}(\texttt{right} \to l) \\
\text{Copy}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{WORDSIZEVALUE}}\}(\texttt{copyright} \to addr, \\
\texttt{right} \to addr, (\texttt{right} \to l) \otimes \texttt{WORDSIZE}) \\
\texttt{copyrightspan} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{copyrightspan} \to p := \texttt{copyright} \to addr \\
\texttt{copyrightspan} \to l := \texttt{right} \to l \\
\blacksquare\text{将right指向的数据复制到copyright，} \\
\text{防止使用LongMod时更改right指向的数据} \\
\texttt{nspan} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{nspan} \to p := \texttt{n} \to addr \\
\texttt{nspan} \to l := \texttt{n} \to l \\
\texttt{remainder} := \text{LongMod}(\texttt{copyrightspan}, \texttt{nspan}) \\
\text{Sub}(\texttt{left}, \texttt{remainder}) \\
\downarrow \texttt{remainder} \\
\downarrow \texttt{copyright} \\
\blacksquare\text{如果right的值大于n，则将right的值限制在n的范围内}
\end{array}\right]$$

定义：Sub($\texttt{left} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{right} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{n} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}$) =

$$
\left[\begin{array}{c}
\blacksquare\text{此算法会将}\texttt{right}\text{的值减到}\texttt{left}\text{上，并限制在状态数}\texttt{n}\text{的范围内} \\
\blacksquare\text{要求满足}\texttt{right},\texttt{n}\text{皆没有无效位，且}\texttt{left}\text{的长度与}\texttt{n}\text{相同} \\
\left\{\begin{array}{l}
\text{如果}:\text{IsGreaterOrEqual}(\texttt{right}\to addr,\texttt{right}\to l,\texttt{n}\to addr,\texttt{n}\to l) \\
\blacksquare\text{IsGreaterOrEqual并未给出实现，请参考之前有关模拟数胞之间的比较} \\
\text{LimitRightValueThenDecrease}(\texttt{left},\texttt{right},\texttt{n}) \\
\blacksquare\text{这个分支确保之后Sub执行时}\texttt{right}\text{指向的值小于}\texttt{n}
\end{array}\right. \\
\texttt{i} := NC_{\texttt{WORDSIZEVALUE}}(0) \\
\texttt{borrow} := NC_2(0) \\
\left[\begin{array}{c}
\text{当}:\texttt{i}<\texttt{left}\to l\wedge\texttt{i}<\texttt{right}\to l \\
\texttt{copyborrow} := NC_{\texttt{WORDSIZEVALUE}}(0) \\
\texttt{copyborrow} :=^{\oplus} \texttt{borrow} \\
\texttt{left[i]} := \texttt{left[i]}\ominus\texttt{right[i]}\ominus\texttt{copyborrow} \\
\texttt{borrow} := \text{IsOverflow}() \\
\texttt{i} :=^{\oplus} 1
\end{array}\right] \\
\left\{\begin{array}{l}
\text{如果}:\texttt{borrow}=1\wedge\texttt{i}<\texttt{left}\to l\wedge\texttt{i}\geq\texttt{right}\to l \\
\texttt{parts}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{parts}\to p := \texttt{left}\to addr \\
\texttt{parts}\to l := \texttt{left}\to l \\
\texttt{parts}\to p := \texttt{parts}\to p\boxplus\texttt{i} \\
\texttt{patys}\to l :=^{\ominus} \texttt{i} \\
\texttt{borrow} := \text{PropagateBorrowInDecrease}(\texttt{parts},\texttt{borro}) \\
\blacksquare\text{如果}\texttt{right}\text{的长度小于}\texttt{left}\text{的长度，且进位不为 0，} \\
\text{则将进位处理到}\texttt{left}\text{的相应位置}
\end{array}\right. \\
\left\{\begin{array}{l}
\text{如果}:\texttt{borrow}=1\vee\text{IsGreaterOrEqual}(\texttt{left}\to addr,\texttt{left}\to l,\texttt{n}\to addr,\texttt{n}\to l) \\
\texttt{nspan}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{nspan}\to p := \texttt{n}\to addr \\
\texttt{nspan}\to l := \texttt{n}\to l \\
\texttt{vspan}\in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{vspan}\to p := \texttt{left}\to addr \\
\texttt{vspan}\to l := \texttt{left}\to l \\
\text{LimitValueAfterDecrease}(\texttt{nspan},\texttt{vspan}) \\
\blacksquare\text{如果最后的值相对状态数溢出，或没有溢出但是产生借位，进行值的修正}
\end{array}\right.
\end{array}\right]
$$

1.7.6 任意状态数的数胞的乘法

接下来，我们给出任意状态数的数胞的乘法的具体算法。

$$\text{定义}：\texttt{HALFSIZEVALUE} = \lfloor \texttt{WORDSIZEVALUE} \div 2 \rfloor$$

$$\text{定义}：\text{PropagateCarryInMultiply}(\texttt{parts} \in SPAN_{NC_{\texttt{HALFSIZEVALUE}}}, \texttt{carry} \in NC_{\texttt{HALFSIZEVALUE}})$$

执行这个算法之后，parts所表示的自然数变为原来的值加上carry的值，

需要确保parts不会溢出，算法没有输出

$$\text{定义}：\text{LimitRightValueThenMultiply}(\texttt{left} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{right} \in DA_{NC_{\texttt{WORDSIZEVALUE}}},$$

$$\texttt{n} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}) =$$

$$\left[\begin{array}{c}
\texttt{copyright} :=\uparrow DA_{NC_{\texttt{WORDSIZEVALUE}}}(\texttt{right} \to l) \\
\text{Copy}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{WORDSIZEVALUE}}\}(\texttt{copyright} \to addr, \\
\texttt{right} \to addr, (\texttt{right} \to l) \otimes \texttt{WORDSIZE}) \\
\texttt{copyrightspan} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{copyrightspan} \to p := \texttt{copyright} \to addr \\
\texttt{copyrightspan} \to l := \texttt{right} \to l \\
\blacksquare\text{将right指向的数据复制到copyright，} \\
\text{防止使用LongMod时更改right指向的数据} \\
\texttt{nspan} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{nspan} \to p := \texttt{n} \to addr \\
\texttt{nspan} \to l := \texttt{n} \to l \\
\texttt{remainder} := \text{LongMod}(\texttt{copyrightspan}, \texttt{nspan}) \\
\text{Mul}(\texttt{left}, \texttt{remainder}) \\
\downarrow \texttt{remainder} \\
\downarrow \texttt{copyright} \\
\blacksquare\text{如果right的值大于n，则将right的值限制在n的范围内}
\end{array}\right]$$

$$\text{定义}：\text{FillZero}\{\texttt{T}\}(\texttt{p} \in PTR_{\texttt{T}}),$$

执行此算法之后，p指向的数据所占的存储空间的每个字节都被设为 0

$$\text{定义}：\text{Mul}(\texttt{left} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{right} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}, \texttt{n} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}) =$$

$$
\left[\begin{array}{c}
\blacksquare\text{此算法会将}\texttt{right}\text{的值乘到}\texttt{left}\text{上，并限制在状态数}\texttt{n}\text{的范围内} \\
\blacksquare\text{要求满足}\texttt{right},\texttt{n}\text{皆没有无效位，且}\texttt{left}\text{的长度与}\texttt{n}\text{相同} \\
\left\{\begin{array}{l}
\text{如果}:\text{IsGreaterOrEqual}(\texttt{right}\to addr,\texttt{right}\to l,\texttt{n}\to addr,\texttt{n}\to l) \\
\blacksquare\text{IsGreaterOrEqual并未给出实现，请参考之前有关模拟数胞之间的比较} \\
\text{LimitRightValueThenMultiply}(\texttt{left},\texttt{right},\texttt{n}) \\
\blacksquare\text{这个分支确保之后Mul执行时}\texttt{right}\text{指向的值小于}\texttt{n}
\end{array}\right. \\
\texttt{halfformleft}\in SPAN_{NC_{\texttt{HALFSIZEVALUE}}} \\
\texttt{halfformleft}\to p:=\text{Cast}\{NC_{\texttt{WORDSIZEVALUE}},NC_{\texttt{HALFSIZEVALUE}}\}(\texttt{left}\to addr) \\
\texttt{halfformleft}\to l:=(\texttt{left}\to l)\otimes 2 \\
\texttt{halfformright}\in SPAN_{NC_{\texttt{HALFSIZEVALUE}}} \\
\texttt{halfformright}\to p:=\text{Cast}\{NC_{\texttt{WORDSIZEVALUE}},NC_{\texttt{HALFSIZEVALUE}}\}(\texttt{right}\to addr) \\
\texttt{halfformright}\to l:=(\texttt{right}\to l)\otimes 2 \\
\texttt{intermediatesize}:=(\texttt{halfformleft}\to l)\oplus(\texttt{halfformright}\to l) \\
\texttt{intermediatedata}:=\uparrow DA_{NC_{\texttt{HALFSIZEVALUE}}}(\texttt{intermediatesize}) \\
\text{FillZero}\{ARR_{NC_{\texttt{HALFSIZEVALUE}}}(\texttt{intermediatesize})\}(\texttt{intermediatedata}\to addr) \\
\texttt{i}:=NC_{\texttt{WORDSIZEVALUE}}(0) \\
\texttt{carry}:=NC_{\texttt{HALFSIZEVALUE}}(0) \\
\blacksquare\text{外层循环，结束时代表}\texttt{right}\text{的每一位都已与自身的每一位相乘} \\
\left[\begin{array}{c}
\text{当}:\texttt{i}<\texttt{halfformright}\to l \\
\texttt{rightdigital}:=(halfformright\to p)[\texttt{i}] \\
\blacksquare\text{内层循环，结束时代表}\texttt{right}\text{的第 i 位都已与自身的每一位相乘} \\
\text{InnerLayerOfMultiply}(\texttt{i},\texttt{rightdigital},\texttt{halfformleft}, \\
\&\texttt{carry},\texttt{intermediatedata}) \\
\left\{\begin{array}{ll}
\texttt{lastparts}\in SPAN_{NC_{\texttt{HALFSIZEVALUE}}} & \text{如果}:\texttt{carry}>0 \\
\texttt{lastparts}\to p:=(\texttt{intermediatedata}\to addr)\boxplus\texttt{i}\boxplus(\texttt{halfformleft}\to l) & \\
\texttt{lastparts}\to l:=1 & \\
\text{PropagateCarryInMultiply}(\texttt{lastparts},\texttt{carry}) &
\end{array}\right. \\
\texttt{i}:=^{\oplus}1
\end{array}\right] \\
\text{AfterMultiplyProc}(\texttt{intermediatedata},\texttt{left},\texttt{n})
\end{array}\right]
$$

$$
\begin{array}{c}
\text{InnerLayerOfMultiply}(\texttt{i}\in NC_{\texttt{WORDSIZEVALUE}},\texttt{rightdigital}\in NC_{\texttt{HALFSIZEVALUE}}, \\
\texttt{halfformleft}\in SPAN_{NC_{\texttt{HALFSIZEVALUE}}},\texttt{pcarry}\in PTR_{NC_{\texttt{HALFSIZEVALUE}}}, \\
\texttt{intermediatedata}\in DA_{NC_{\texttt{HALFSIZEVALUE}}})=
\end{array}
$$

$$\left[\begin{array}{c}
\texttt{j} := NC_{\texttt{WORDSIZEVALUE}}(0) \\
\left[\begin{array}{c}
\text{当} : \texttt{j} < \texttt{halfformleft} \to l \\
\texttt{leftdigital} := NC_{\texttt{WORDSIZEVALUE}}(0) \\
\texttt{leftdigital} :=^{\oplus} (halfformleft \to p)[\texttt{i}] \\
\texttt{leftdigital} :=^{\otimes} \texttt{rightdigital} \\
\texttt{lastcarry} := \texttt{carry} \\
\texttt{temporary} := \texttt{leftdigital} \oslash \texttt{HALFSIZEVALUE} \\
\texttt{carry} := 0 \\
\texttt{carry} :=^{\oplus} \texttt{temporary} \\
\blacksquare\text{取相乘结果的高半部分作为下次乘法的进位} \\
\texttt{temporary} := \texttt{leftdigital} \text{ rem } \texttt{HALFSIZEVALUE} \\
\texttt{low} := NC_{\texttt{HALFSIZEVALUE}}(0) \\
\texttt{low} :=^{\oplus} \texttt{temporary} \\
\blacksquare\text{低半部分} \\
\texttt{intermediatedata}[\texttt{i} \oplus \texttt{j}] :=^{\oplus} \texttt{low} \\
\texttt{parts} \in SPAN_{NC_{\texttt{HALFSIZEVALUE}}} \quad \text{如果} : \text{IsOverflow}() \\
\texttt{parts} \to p := (\texttt{halfformleft} \to p) \boxplus \texttt{i} \boxplus \texttt{j} \boxplus 1 \\
\texttt{parts} \to l := (\texttt{halfformleft} \to l) \ominus \texttt{i} \ominus \texttt{j} \ominus 2 \\
\left\{\begin{array}{l}
\text{PropagateCarryInMultiply}(\texttt{parts}, 1) \\
\blacksquare\text{这个分支用于处理相加导致的进位}
\end{array}\right. \\
\texttt{parts} \to p := (\texttt{parts} \to p) \boxminus 1 \\
\texttt{parts} \to l :=^{\oplus} 1 \\
\text{PropagateCarryInMultiply}(\texttt{parts}, \texttt{lastcarry}) \\
\left\{\begin{array}{l}
\texttt{carry} :=^{\oplus} 1 \quad \text{如果} : \texttt{low} \oplus \texttt{lastcarry} < \texttt{low} \\
\blacksquare\text{如果相乘结果加上进位导致再次进位，修正进位}
\end{array}\right. \\
\texttt{j} :=^{\oplus} 1
\end{array}\right]
\end{array}\right]$$

定义 : AfterMultiplyProc($\texttt{intermediatedata} \in DA_{NC_{\texttt{HALFSIZEVALUE}}}, \texttt{left} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}$,

$\texttt{n} \in DA_{NC_{\texttt{WORDSIZEVALUE}}}$) =

$$
\left[\begin{array}{c}
\texttt{newsize} := \texttt{intermediatedata} \to l \\
\texttt{i} := NC_{\texttt{WORDSIZEVALUE}}((\texttt{intermediatedata} \to l) \ominus 1) \\
\left[\begin{array}{c}
\text{当} : \texttt{intermediatedata}[\texttt{i}] = 0 \wedge \texttt{i} \geq 0 \\
\texttt{newsize} :=^{\ominus} 1 \\
\texttt{i} :=^{\ominus} 1 \\
\blacksquare\text{设定新尺寸，去除无效位的长度}
\end{array}\right] \\
\begin{cases}
\texttt{newsize} :=^{\oplus} 1\text{如果} : \texttt{newsize} \text{ rem } 2 = 0 \\
\blacksquare\text{确保}\texttt{newsize}\text{是偶数}
\end{cases} \\
\texttt{intermediatedata} \to \text{resize}(\texttt{newsize}) \\
\texttt{final} \in SPAN_{NC_{\texttt{WORDSIZEVALUE}}} \\
\texttt{final} \to p := \text{Cast}\{NC_{\texttt{HALFSIZEVALUE}}, NC_{\texttt{WORDSIZEVALUE}}\}(\texttt{intermediatedata} \to addr) \\
\texttt{final} \to l := (\texttt{intermediatedata} \to l) \oslash 2 \\
\texttt{finalda} :=\uparrow DA_{NC_{\texttt{WORDSIZEVALUE}}}(\texttt{final} \to l) \\
\text{Copy}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{WORDSIZEVALUE}}\}(\texttt{finalda} \to addr, \\
\texttt{final} \to p, \texttt{final} \to l) \\
\begin{cases}
\text{如果} : \text{IsGreaterOrEqual}(\texttt{final} \to p, \texttt{final} \to l, \texttt{n} \to addr, \texttt{n} \to l) \\
\downarrow \texttt{finalda} \\
\texttt{nspan} \in SPAN_{\texttt{WORDSIZEVALUE}} \\
\texttt{nspan} \to p := \texttt{n} \to addr \\
\texttt{nspan} \to l := \texttt{n} \to l \\
\texttt{finalda} := \text{LongMod}(\texttt{final}, \texttt{nspan}) \\
\blacksquare\text{将结果限制在状态数内}
\end{cases} \\
\text{FillZero}\{ARR_{NC_{\texttt{WORDSIZEVALUE}}}(\texttt{left} \to l)\}(\texttt{left} \to p) \\
\text{Copy}\{NC_{\texttt{WORDSIZEVALUE}}, NC_{\texttt{WORDSIZEVALUE}}\}(\texttt{left} \to p, \texttt{finalda} \to addr, \\
(\texttt{finalda} \to l) \otimes \texttt{WORDSIZE}) \\
\blacksquare\text{写入结果到数组中}
\end{array}\right]
$$

1.7.7 任意状态数的数胞的除法和求余

对于任意状态数的数胞的除法和求余，本质上是对**任意进制的长除法**的运用。

这里就不做实现，但是声明算法实现时需要注意的一些内容。

对于任意状态数的数胞的除法，要求除数不为 0。如果被除数和除数都只有一位，则可以直接进行除法得到结果。如果被除数小于除数，结果一定为 0。

对于任意状态数的数胞的求余，要求除数不为 0。如果被除数和除数都只有一位，则可以直接进行求余得到结果。如果被除数小于除数，结果一定为被除数。

2 正则表达式

2.1 基本概念

2.1.1 字母表

字母表是一个有限的符号集合，通常使用 Σ 表示。

2.1.2 字符与字符串

字符是字母表中的元素，字符串是由字符组成的有限序列，空字符串记为 ε。
用 Σ^* 表示 Σ 上的所有字符串的全体，包含空字符串 ε。

2.1.3 基本运算

Σ^* 的子集 $\mathtt{U}$ 和 $\mathtt{V}$ 的连接 (积) 定义为

$$\mathtt{UV} = \{\alpha\beta \mid \alpha \in \mathtt{U} \land \beta \in \mathtt{V}\}$$

$\mathtt{V}$ 自身的 $\mathtt{n}$ 次积记为

$$\mathtt{V}^{\mathtt{n}} = \mathtt{VV}\cdots\mathtt{V}(\text{其中有}\mathtt{n}\text{个}\mathtt{V})$$

$\mathtt{V}^0 = \varepsilon$

$\mathtt{V}^*$ 是 $\mathtt{V}$ 的闭包，即

$$\mathtt{V}^* = \mathtt{V}^0 \cup \mathtt{V}^1 \cup \mathtt{V}^2 \cup \cdots = \bigcup_{i=0}^{\infty} \mathtt{V}^i$$

$\mathtt{U} \mid \mathtt{V}$ 是或运算，即

$$\mathtt{U} \cup \mathtt{V}$$

$\sim \mathtt{V}$ 是反运算，意为与 $\mathtt{V}$ 长度相同，但不完全与 $\mathtt{V}$ 相同的任意字符串，即

$$|\sim \mathtt{V}| = |\mathtt{V}| \land \sim \mathtt{V} \neq \mathtt{V}$$

2.2 非确定性有限状态自动机 (NFA)

非确定性有限状态自动机是一种用于描述正则语言的计算模型。NFA 在状态转移过程中允许存在不确定性，即从一个状态读取某个字符后可能转移到多个不同的状态。

2.2.1 NFA 的组成成分

NFA 由五个基本成分组成：

1. **状态集合**：一个有限的状态集合，通常用 $\mathtt{Q}$ 表示。每个状态代表自动机在识别过程中的一个特定配置。

2. **输入字母表**：一个有限的输入符号集合，通常用 Σ 表示。自动机只能识别属于该字母表的字符序列。

3. **状态转移函数**：定义状态之间转移关系的函数，通常用 δ 表示。对于 NFA，转移函数的形式为：

$$\delta : \mathtt{Q} \times \Sigma \to 2^{\mathtt{Q}}$$

其中 $2^{\mathtt{Q}}$ 表示 $\mathtt{Q}$ 的幂集（即 $\mathtt{Q}$ 的所有子集的集合）。这意味着对于每个状态和输入符号的组合，NFA 可以转移到多个可能的状态。

4. **初始状态**：自动机开始运行时的起始状态，通常用 $\mathtt{q}_0 \in \mathtt{Q}$ 表示。NFA 可能允许多个初始状态，此时初始状态为一个状态集合。

5. **接受状态集合**：也称为终结状态集合，通常用 $\mathtt{F} \subseteq \mathtt{Q}$ 表示。当自动机处理完整个输入字符串后，如果最终处于某个接受状态，则认为该字符串被自动机接受。

2.2.2 形式化定义

一个 NFA 可以形式化地定义为一个五元组：

$$\mathtt{M} = (\mathtt{Q}, \Sigma, \delta, \mathtt{q}_0, \mathtt{F})$$

其中：

- $\mathtt{Q}$ 是有限状态集合
- Σ 是有限输入字母表
- $\delta : \mathtt{Q} \times \Sigma \to 2^{\mathtt{Q}}$ 是状态转移函数
- $\mathtt{q}_0 \in \mathtt{Q}$ 是初始状态
- $\mathtt{F} \subseteq \mathtt{Q}$ 是接受状态集合

2.2.3 工作原理

NFA 通过以下方式处理输入字符串：从初始状态开始，依次读取输入字符串中的每个字符，根据转移函数确定可能转移到的状态集合。由于转移的非确定性，NFA 在每一步都可能处于多个状态的叠加中。当处理完整个输入字符串后，如果当前状态集合中包含至少一个接受状态，则该输入字符串被 NFA 接受。

2.2.4 扩展转移函数

为了处理整个字符串而不仅仅是单个字符，需要定义扩展转移函数 $\hat{\delta} : \mathtt{Q} \times \Sigma^* \to 2^{\mathtt{Q}}$：

- 对于空字符串 ε，有 $\hat{\delta}(\mathtt{q}, \varepsilon) = \{\mathtt{q}\}$
- 对于非空字符串 $\mathtt{w} = \mathtt{xa}$（其中 $\mathtt{x} \in \Sigma^*$, $\mathtt{a} \in \Sigma$），有：

$$\hat{\delta}(\mathtt{q}, \mathtt{w}) = \bigcup_{\mathtt{p} \in \hat{\delta}(\mathtt{q}, \mathtt{x})} \delta(\mathtt{p}, \mathtt{a})$$

2.2.5 语言接受性

NFA $\mathtt{M}$ 接受的语言定义为所有能够使 NFA 从初始状态转移到某个接受状态的字符串集合：

$$\mathtt{L}(\mathtt{M}) = \{\mathtt{w} \in \Sigma^* \mid \hat{\delta}(\mathtt{q}_0, \mathtt{w}) \cap \mathtt{F} \neq \emptyset\}$$

NFA 的特例

在具体的实现过程中，我们偏向于实现 NFA 的特例，而不是完整的 NFA。

- 状态转移函数只接受单个字符。
- 初始状态只有一个。
- 不建议设计从一个状态读取某个字符后可能转移到多个不同的状态的 NFA，因为我们打算使用占有型量词进行匹配，相关内容之后会介绍。

2.3 匹配规则

2.3.1 贪婪型量词匹配

贪婪型量词匹配是正则表达式中最基本的匹配策略。采用贪婪匹配的量词会尽可能多地匹配字符，只有在后续匹配失败时才会回溯并减少匹配的字符数量。
特点：

- 最大长度匹配：优先尝试匹配尽可能多的字符
- 回溯机制：当后续模式无法匹配时，会回溯并尝试较短的匹配
- 默认行为：多数正则表达式引擎默认采用贪婪匹配

示例：对于模式 $a\Sigma * b$ 和字符串 "$aabab$"

- 贪婪匹配结果："$aabab$"
- 匹配过程：$\Sigma*$ 首先匹配到字符串末尾，然后回溯直到找到最后一个 b

2.3.2 勉强型量词匹配

勉强型量词匹配（也称为懒惰匹配或最小匹配）与贪婪匹配相反，它会尽可能少地匹配字符，只有在后续模式需要时才会增加匹配的字符数量。

特点：

- 最小长度匹配：优先尝试匹配尽可能少的字符
- 渐进扩展：当后续匹配失败时，逐步增加匹配范围
- 显式指定：需要通过特定语法明确指定

示例：对于模式 $a\Sigma*b$ 和字符串 "$aabab$"

- 勉强匹配结果："aab"
- 匹配过程：$\Sigma*$ 在遇到第一个 b 时就停止匹配

2.3.3 占有型量词匹配

占有型量词匹配类似于贪婪匹配，但一旦匹配成功就不会进行回溯，即使后续模式匹配失败也不会释放已匹配的字符。

特点：

- 无回溯匹配：匹配成功后不会释放已占有的字符
- 性能优化：避免回溯过程，提高匹配效率
- 严格匹配：一旦匹配失败就立即返回失败结果

示例：对于模式 $a\Sigma*b$ 和字符串 "$aabab$"

- 占有匹配结果：匹配失败
- 匹配过程：$\Sigma*$ 匹配所有剩余字符，不释放给后面的 b 匹配

2.3.4 三种匹配规则对比

特性	贪婪匹配	勉强匹配	占有匹配
匹配原则	尽可能多	尽可能少	尽可能多（不回溯）
回溯行为	有回溯	有回溯	无回溯
性能	中等	中等	较高
适用场景	默认情况	需要精确匹配	性能敏感场景

这三种匹配规则为正则表达式提供了灵活的匹配策略，可以根据具体需求选择合适的匹配方式，在匹配精度和性能之间取得平衡。但是在拟态语言中，为了安全性和效率，我们通常只使用 NFA 的特例配上占有匹配。

2.4 转义字符

2.4.1 转义字符的基本概念

转义字符是一种特殊的字符序列，用于表示那些在正则表达式中具有特殊含义的字符的字面值，或者表示不可见的控制字符。转义字符通常以反斜杠（\）开头，后跟一个或多个字符。

2.4.2 反斜杠的转义规则

反斜杠在正则表达式中具有双重作用：

- **消除元字符的特殊含义**：当反斜杠置于元字符之前时，会取消其特殊含义，使其作为普通字符匹配
- **赋予普通字符特殊含义**：当反斜杠置于某些普通字符之前时，会赋予其特殊的匹配功能

示例：

- `\.` 匹配字面意义的点号（取消.的通配含义）
- `\n` 匹配换行符（赋予n特殊含义）

2.4.3 ASCII 码中的转义字符

以下是正则表达式中常用的 ASCII 转义字符：

转义序列	ASCII 值	含义
`\a`	0x07	响铃符（BEL）
`\b`	0x08	退格符（BS）
`\t`	0x09	水平制表符（TAB）
`\n`	0x0A	换行符（LF）
`\v`	0x0B	垂直制表符（VT）
`\f`	0x0C	换页符（FF）
`\r`	0x0D	回车符（CR）
`\e`	0x1B	转义符（ESC）
`\0`	0x00	空字符（NUL）

2.4.4 八进制和十六进制转义

除了预定义的转义序列外，还可以使用数值转义：

- **八进制转义**：`\ddd`（1-3 位八进制数字）

- **十六进制转义**：`\xhh`（2 位十六进制数字）或`\x{hhhh}`（1-4 位十六进制数字）

示例：

- `\101` 匹配字符'A'（八进制 65）
- `\x41` 匹配字符'A'（十六进制 41）

在拟态语言的正则表达式中，默认不使用八进制和十六进制转义。

2.4.5 无法转义字符的处理

当反斜杠后接的字符不属于任何预定义的转义序列，且该字符本身也不具有特殊含义时，反斜杠将被忽略，后面的字符将按原义进行匹配。
示例：

- `\a` 匹配响铃符（有效转义）
- `\q` 匹配字符`q`（无效转义，按原义处理）
- `\@` 匹配字符`@`（非元字符，按原义处理）

这种设计确保了向后兼容性，即使未来引入新的转义序列，现有的模式也不会被破坏。在实际使用中，建议只对具有特殊含义的字符进行转义，以避免混淆。

2.5 表达式中的字符标记

首先，对正则表达式 R 中的每个字面字符出现进行标记，使每个出现具有唯一的位置索引。假设正则表达式 R 由字母表 Σ 中的字符、特殊字符（如操作符 $|, *$ 等）和空表达式 ε 组成。我们只关注字面字符（非特殊字符）的出现。

定义 P 为 R 中所有字面字符出现的集合。每个出现 $p \in P$ 具有：

- 字符 $c(p) \in \Sigma$（即该位置对应的字面字符）。
- 唯一位置索引 $i(p)$（默认从左到右按顺序编号）。

例如，对于正则表达式 $R =$ "aa*b"，字面字符出现为：位置 1 的'a'、位置 2 的'a'、位置 3 的'b'、'*' 由于是特殊字符 (闭包运算) 不计入其中。因此，$P = \{p_1, p_2, p_3\}$，其中 $c(p_1) =' a', c(p_2) =' a', c(p_3) =' b'$。

注意，对于反斜杠与其紧接字符形成的转义或原义组合，将其视为一个字面字符。反运算紧接一个普通的字面字符，将其视为一个反字面字符，是字面字符的一种特例。

2.6 状态节点

基于标记的字符出现，定义状态节点集合 Q。每个状态节点对应一个字符出现或用于控制流程（如开始状态）。状态节点有唯一 ID 和类型。

- 令 $Q = \{q_p \mid p \in P\} \cup \{q_0\}$，其中 q_0 是开始状态。
- 每个状态节点 q_p（对于 $p \in P$）有三个成员：
  - 转移字符：即 $c(p)$，表示进入该状态时需要匹配的字符。
  - 状态类型：例如普通字符匹配，用于非反字面字符 $c(p)$。任意字符匹配。空转移，用于 ε 转移的状态。反字符匹配，用于反字面字符 $c(p)$。
  - 连接的其他状态节点引用：即从该状态出发的转移目标列表。
- 状态节点 q_0 是开始状态，没有转移字符（空转移）。

2.7 构建规则

接下来我们需要将正则表达式字面量构建成 NFA，我们需要定义一些构建规则，以指导构建的过程。

首先，对于 $q_1, q_2 \in Q$，如果 $i(q_1) + 1 = i(q_2)$，则 $\exists \delta(q_1, c(q_2)) = q_2$。

对于 $q_1, q_2, q_3 \in Q$，如果 $i(q_1) + 1 = i(q_2), i(q_2) + 1 = i(q_3)$，且 $\delta(q_1, \varepsilon) = q_2$，则 $\exists \delta(q_1, c(q_3)) = q_3$。

设表达式 R_0

- 设 $q_{start}^{R_0}$ 为分支 R_0 的起始状态
- 设 $q_{end}^{R_0}$ 为分支 R_0 的结束状态

对于或运算 $R_1|R_2|\cdots|R_n$，定义以下构建规则：

状态定义：

- 设 Q^{or} 为进入或运算 R 的所有前驱状态的集合
- 设 Q_f^{or} 为或运算 R 的所有后继状态的集合
- 接下来默认小写变量为字符，大写变量为字符串。
- 对于表达式 $R = aA$，且 a 对应的状态为 q_a，则 $q_{start}^{R} = q_a$
- 对于表达式 $R = Aa$，且 a 对应的状态为 q_a，$q_{end}^{R} = q_a$
- 对于表达式 $R = R_1|R_2|\cdots|R_n$，且 $R_1, R_2, \cdots, R_n$ 均不为空字符串，有 $q_{start}^{R} \in \{q_{start}^{R_1}, q_{start}^{R_2}, \cdots, q_{start}^{R_n}\}$

- 对于表达式 $R = R_1|R_2|\cdots|R_n$，且 $R_1, R_2, \cdots, R_n$ 均不为空字符串，有 $q_{end}^{R} \in \{q_{end}^{R_1}, q_{end}^{R_2}, \cdots, q_{end}^{R_n}\}$

起始状态集合确定：

- 如果 R 是顶层表达式，则 $Q^{or} = \{q_0\}$，其中 q_0 是整个 NFA 的起始状态
- 如果 R 出现在连接运算 $P \cdot R$ 中，且 $P \neq \varepsilon$，则 $Q^{or} = \{q_{end}^{P}\}$
- 如果 R 出现在或运算 $(\cdots|R|\cdots)$ 中，则 Q^{or} 为该或运算的前驱状态集合
- 如果 R 出现在闭包运算 $R^* = (RS)*$ 中，则 $Q^{or} = q_{end}^{R^*}$

结束状态集合确定：

- 如果 R 是顶层表达式，则 Q_f^{or} 是 NFA 的结束状态
- 如果 R 出现在连接运算 $R \cdot S$ 中，且 $S \neq \varepsilon$，则 $Q_f^{or} = \{q_{start}^{S}\}$
- 如果 R 出现在或运算 $(\cdots|R|\cdots)$ 中，则 Q_f^{or} 为该或运算的后继状态集合
- 如果 R 出现在闭包运算 $R^* = (PR)*$ 中，则 $Q_f^{or} = q_{start}^{R^*}$

起始状态与结束状态之间的转移函数定义：

$$\forall i \in \{1, 2, \ldots, n\}, \forall q^{or} \in Q^{or}, \exists \delta(q^{or}, c(q_{start}^{R_i})) = q_{start}^{R_i}$$
$$\forall i \in \{1, 2, \ldots, n\}, \forall q_f^{or} \in Q_f^{or}, \exists \delta(q_{end}^{R_i}, c(q_f^{or})) = q_f^{or}$$

对于闭包运算 R_0*，定义以下构建规则：
状态定义：

- $(\varepsilon)* = \varepsilon$，以下的 $R_0 \neq \varepsilon$
- 对于表达式 $R = (R_0) * S$，且 $S \neq \varepsilon$，有 $q_{start}^{R} \in \{q_{start}^{R_0}\} \cup \{q_{start}^{S}\}$
- 对于表达式 $R = P(R_0)*$，且 $P \neq \varepsilon$，有 $q_{end}^{R} \in \{q_{end}^{R_0}\} \cup \{q_{end}^{P}\}$
- 对于表达式 $R = \cdots|(R_0) * |\cdots$，且 Q_f^{or} 为 R 所在的或运算的所有后继状态的集合，则 $q_{start}^{R} \in \{q_{start}^{R_0}\} \cup Q_f^{or}$
- 对于表达式 $R = \cdots|(R_0) * |\cdots$，且 Q^{or} 为 R 所在的或运算的所有前驱状态的集合，则 $q_{end}^{R} \in \{q_{end}^{R_0}\} \cup Q^{or}$
- 如果闭包运算在表达式顶层，且连接表达式的起始状态，那么起始状态存在转移路径到闭包运算的后继状态。

- 如果闭包运算在表达式顶层，且连接表达式的结束状态，那么闭包运算的前驱状态存在转移路径到结束状态。

闭包运算的转移函数定义：

$$\forall q_{end}^{R_0}, \forall q_{start}^{R_0}, \exists \delta(q_{end}^{R_0}, c(q_{start}^{R_0})) = q_{start}^{R_0}$$
$$R = P \cdot R_0 * \cdot S, \forall q_{end}^{P}, \forall q_{start}^{S}, \exists \delta(q_{end}^{P}, c(q_{start}^{S})) = q_{start}^{S}$$