

Universidad Autónoma del Estado de Hidalgo  
Instituto de Ciencias Básicas e Ingeniería  
Licenciatura en Ciencias Computacionales

## REPORTE DE PRÁCTICA NO. 2

### 1.8 PRÁCTICA 2: AFD Y AFND

Autómatas y Compiladores  
Dr. Eduardo Cornejo-Velázquez  
Semestre 6 Grupo 3  
Alumna Ashley Torres Perez



## 1. Introducción

La teoría de la computación o autómatas se centra en el estudio de máquinas abstractas para comprender sus capacidades y limitaciones de computación mediante el análisis de modelos matemáticos de cómo pueden realizar cálculos. El foco del estudio es una computadora idealizada, o modelo computacional; sin embargo, un modelo no podría ser suficiente para describir todas las máquinas o computadoras, por lo que existen varios tipos de modelos computacionales. El modelo más simple de todos es la máquina de estados finitos o autómata finito, que describe una computadora con una cantidad de memoria muy limitada (Sisper, 2006). Un ejemplo de dicho modelo es un controlador de interruptor de encendido/apagado para un dispositivo electrónico (consulte la Figura 1).

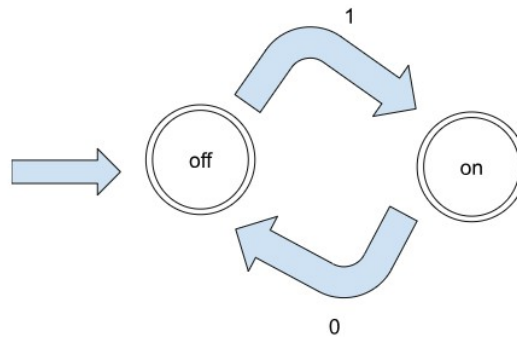


Figure 1: OFF/ON switch controller as a finite state machine.

Un autómata finito se puede clasificar en dos categorías: determinista y no determinista. Un autómata finito determinista (AFD) es un modelo matemático de computación que consta de un conjunto finito de estados y reglas bien definidas para transitar entre ellos en función de una entrada. En cada estado y para cada símbolo de entrada, existe una única transición posible, y por lo tanto, no hay ambigüedad en la ejecución, dada una cadena de entrada, el autómata sigue un solo camino en su diagrama de estados. A diferencia, un autómata finito no determinista (AFND) tiene la característica que para un mismo estado y símbolo de entrada, pueden existir varias transiciones posibles (es decir, múltiples caminos de ejecución), y Puede haber transiciones vacías ( $\epsilon$ -transiciones), donde el autómata cambia de estado sin consumir un símbolo de entrada (Kozen, 1997). Aunque es importante resaltar que todos los autómatas finitos deterministas son autómatas finitos no deterministas, y un AFND se puede convertir en un AFD (Sisper, 2006).

Para los propósitos de esta práctica, examinaré un autómata determinista finito e implementaré algorítmicamente un programa que lea un autómata como una 5-tupla y determine si el autómata definido acepta una cadena de caracteres.

## 2. Marco teórico

Formalmente, un autómata finito es una 5-tupla,  $(Q, \Sigma, \delta, q_0, F)$ , donde:

1.  $Q$  es un conjunto finito llamado estados.
2.  $\Sigma$  es un conjunto finito llamado alfabeto
3.  $\delta: Q \times \Sigma \rightarrow Q$  es la función de transición
4.  $q_0$  es el estado de inicio, y
5.  $F \subseteq Q$  es el conjunto de estados de aceptación o estados finales (Sisper, 2006).

Entonces un autómata finito es un modelo computacional que consiste de estados, y uno de estos estados debe ser el estado de inicio. Un autómata finito puede tener desde 0 a  $n$  estados de aceptación, siempre y cuando  $n$  sea menor o igual a  $|Q|$ . Un autómata requiere de un conjunto que defina un alfabeto o símbolos,  $\Sigma$ , para poder definir las transiciones de un estado a otro. Por tanto,  $\delta$  es la función utilizada para pasar de un estado a otro mediante símbolos válidos del alfabeto.

Anteriormente establecimos que la diferencia entre un AFD y un AFND era que **cada estado de un AFD siempre tiene exactamente una flecha de transición existente para cada símbolo del alfabeto** (Kozen, 1997). Esto significa que un autómata  $M_1, (Q, \Sigma, \delta, q_0, F)$ , tiene exactamente  $T$  transiciones, donde  $T$  es el producto de la cardinalidad del conjunto  $Q$  y la cardinalidad del conjunto  $\Sigma$ . En la Figura 2 se muestra un ejemplo de un AFD.

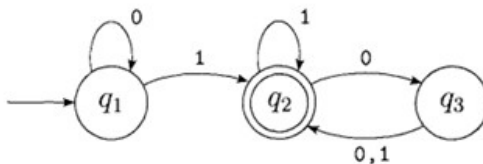


Figure 2: El autómata  $M_1$  tiene  $Q = \{q_1, q_2, q_3\}$  y  $\Sigma = \{0, 1\}$ , donde  $|Q| = 3$  y  $|\Sigma| = 2$ , por lo que las transiciones totales,  $T$ , son  $T = 6$

Además, el autómata  $M_1$  puede describirse mediante su función de transición  $\delta$ , que es:

	0	1
$q_1$	$q_1$	$q_1, q_2$
$q_2$	$q_1$	$q_1, q_2$

La definición formal de un autómata finito se ajusta a la descripción de un autómata determinista, por lo que es la que vamos a utilizar para nuestro propósito de diseñar un algoritmo que dado un AFD  $M = (Q, \Sigma, \delta, q_0, F)$  y un conjunto de cadenas determine si una cadena es aceptada por  $M$ .

### Problemática

Sea  $M = (Q, \Sigma, \delta, q_0, F)$  una AFD y sea  $w = w_0w_1w_2...w_n$  una cadena de símbolos donde  $w_i \in \Sigma$ , entonces  $M$  acepta  $w$  si existe una secuencia de estados  $r_0r_1r_2...r_n \in Q$  que cumplan las siguientes condiciones:

1.  $r_0 = q_0$ , el primer estado de la secuencia es igual al estado de inicio
2.  $\delta(r_i, w_{i+1}) = r_{i+1}$ , para  $i = 0, 1, \dots, n - 1$ , existe una transición de un estado a otro dada por la función de transición y
3.  $r_n \subseteq F$ , el último estado de la secuencia debe pertenecer al conjunto de estados aceptados.

El autómata reconoce un lenguaje  $A$  si acepta todas las palabras  $w$  que pertenecen a  $A$ . La tarea es escribir un programa que, dado un lenguaje  $A$  y un autómata  $M$ , determinar si una palabra  $W \in A$  es aceptada por  $M$ . Una máquina puede aceptar varias cadenas, pero siempre reconoce un solo lenguaje. Si la máquina no acepta ninguna cadena, todavía reconoce un lenguaje: el lenguaje vacío  $\emptyset$  (Sisper, 2006).

## Entradas

La primera línea de entrada contiene seis enteros  $N, S, D, q_0, T$  y  $C$ , ( $1 \leq N, S, C \leq 100, 1 \leq D \leq 10^4, 1 \leq q_0 \leq N, 0 \leq T \leq N$ ), donde  $N = |Q|$ ,  $S = |\Sigma|$ ,  $D = N \times S$  es el número de transiciones en el autómata,  $q_0$  es el estado inicial,  $T = |F|$ , y  $C$  es la cantidad de cadenas a verificar si son aceptadas o no por el autómata  $M$ . Cada estado  $q \in Q$  se identifica de manera implícita por un número entero con valor entre 1 y  $N$ .

La segunda línea contiene el alfabeto  $\Sigma$ , representado por una secuencia de  $S$  símbolos  $s_i$  separados por espacios, tal que cada símbolo  $s_i$  puede ser una letra, un dígito o cualquier carácter del código ASCII excepto por el espacio.

La tercera línea contiene el conjunto de estados de aceptación  $F$ , representado por una secuencia de  $T$  enteros  $t_i$  ( $1 \leq t_i \leq N$ ) separados por espacios.

Las siguientes  $D$  líneas especifican las transiciones del autómata. Cada línea define una transición  $\delta(I, X) = J$  por medio de un entero  $I$ , un carácter  $X$  y un entero  $J$  ( $I, J \in Q$  y  $X \in \Sigma$ ) separados por espacios, representando la transición desde el estado  $I$  hacia el estado  $J$  cuando el símbolo de la entrada sea  $X$ .

Finalmente, cada una de las siguientes  $C$  líneas contienen una cadena  $W$ , compuesta por símbolos que pertenecen al alfabeto  $\Sigma$ . La longitud de la cadena  $W$  está entre 0 y 100 caracteres (OmegaUp, 2022).

## Salida

Para cada una de las  $C$  cadenas  $W$  se deberá imprimir el mensaje ACEPTADA si el autómata  $M$  acepta la cadena  $W$ , ó RECHAZADA en caso contrario.

## Ejemplo 1

Según la Figura 3:

### Entrada

1. **Primera línea:** 3 2 6 1 1 3

- $N = 3$ : El autómata tiene 3 estados.
- $S = 2$ : El alfabeto tiene 2 símbolos.
- $D = 6$ : Hay 6 transiciones en el autómata.
- $q_0 = 1$ : El estado inicial es el estado 1.
- $T = 1$ : Hay 1 estado de aceptación.
- $C = 3$ : Se deben verificar 3 cadenas.

2. **Segunda línea:** 0 1

- El alfabeto  $\Sigma$  consiste en los símbolos 0 y 1.
3. **Tercera línea:** 2
- El conjunto de estados de aceptación  $F$  contiene solo el estado 2.
4. **Transiciones:**
- 1 0 1: Desde el estado 1, con el símbolo 0, se transita al estado 1.
  - 1 1 2: Desde el estado 1, con el símbolo 1, se transita al estado 2.
  - 2 0 3: Desde el estado 2, con el símbolo 0, se transita al estado 3.
  - 2 1 2: Desde el estado 2, con el símbolo 1, se transita al estado 2.
  - 3 0 2: Desde el estado 3, con el símbolo 0, se transita al estado 2.
  - 3 1 2: Desde el estado 3, con el símbolo 1, se transita al estado 2.
5. **Cadenas a verificar:**
- 100: La primera cadena a verificar.
  - 0: La segunda cadena a verificar.
  - 11: La tercera cadena a verificar.

Input	Output
3 2 6 1 1 3	ACEPTADA
0 1	RECHAZADA
2	ACEPTADA
1 0 1	
1 1 2	
2 0 3	
2 1 2	
3 0 2	
3 1 2	
100	
0	
11	

Figure 3: Ejemplo de una entrada y su respectiva salida

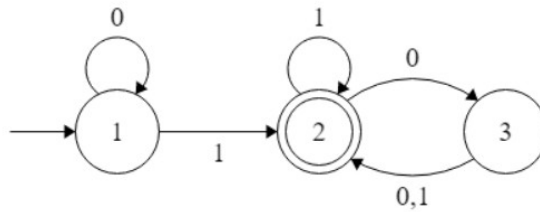


Figure 4: Representación gráfica del autómata de ejemplo 1 de la Figura 3

## Ejemplo 2

El siguiente ejemplo, Figura 5, es un caso de prueba creado para poder desarrollar un algoritmo que sea capaz de manejar diferentes casos y no sea solo un enfoque heurístico.

Input	Output
4 3 12 1 2 5	ACEPTADA
a b c	ACEPTADA
2 4	RECHAZADA
1 a 1	ACEPTADA
1 c 3	ACEPTADA
1 b 3	
3 a 3	
3 c 4	
3 b 4	
4 a 4	
4 c 2	
4 b 4	
2 b 3	
2 a 2	
2 c 2	
bccccccaaaaac	
bbaa	
aaaaaaa	
bbbaaaaaaccc	
caba	

Figure 5: Ejemplo de una entrada y su respectiva salida

**Conjunto de estados:**

$$Q = \{1, 2, 3, 4\}$$

**Alfabeto de entrada:**

$$\Sigma = \{a, b, c\}$$

**Estado inicial:** El estado inicial es  $q_0 = 1$ .

**Estados finales:** Los estados finales son 2 y 4.

**Función de transición:**

$$\delta(q, x) = \begin{cases} 1 & \text{si } q = 1 \text{ y } x = a \\ 3 & \text{si } q = 1 \text{ y } x = b \text{ o } x = c \\ 2 & \text{si } q = 2 \text{ y } x = a \text{ o } x = c \\ 4 & \text{si } q = 2 \text{ y } x = b \\ 3 & \text{si } q = 3 \text{ y } x = a \\ 2 & \text{si } q = 3 \text{ y } x = b \\ 4 & \text{si } q = 3 \text{ y } x = c \\ 4 & \text{si } q = 4 \text{ y } x = a \text{ o } x = b \end{cases}$$

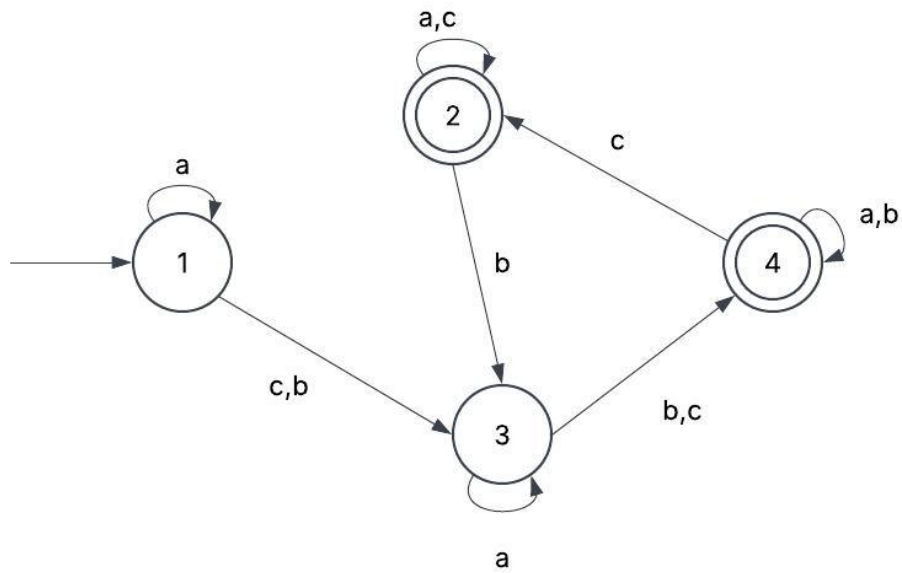


Figure 6: Representación gráfica del autómata de ejemplo 2 de la Figura 5

### 3. Herramientas empleadas

1. OmegaUp: es una plataforma en línea diseñada para ayudar a estudiantes, programadores y entusiastas de la informática a mejorar sus habilidades en programación y algoritmos. La plataforma ofrece una variedad de problemas y desafíos que cubren diferentes temas, desde conceptos básicos hasta técnicas avanzadas de programación competitiva.
2. Code::Blocks: es un entorno de desarrollo integrado (IDE) gratuito y de código abierto diseñado para programación en C, C++ y Fortran.
3. C++: es un lenguaje de programación de propósito general y es preferido por concursantes de programación competitiva.



## 4. Desarrollo

### Análisis de requisitos

#### Condiciones

El programa a desarrollar debe tener la capacidad de evaluar si una palabra  $W$  es aceptada por una autómata  $M$  de acuerdo a las condiciones ya establecidas previamente:

1.  $r_0 = q_0$ , que el primer estado de la secuencia sea igual al estado de inicio
2.  $\delta(r_i, w_{i+1}) = r_{i+1}$ , para  $i = 0, 1, \dots, n-1$ , que debe existir una transición de un estado a otro dada por la función de transición y
3.  $r_n \subseteq F$ , que el último estado de la secuencia debe pertenecer al conjunto de estados aceptados.

#### Estructuras de datos

Para representar un autómata finito determinista (AFD)  $M = (Q, \Sigma, \delta, q_0, F)$  en C++, es necesario emplear estructuras de datos adecuadas, tanto primitivas como no primitivas. El problema especifica que el autómata debe manejar un conjunto de estados  $Q$ , un conjunto de símbolos  $\Sigma$  y un conjunto de estados finales  $F$ , todos con una cardinalidad máxima de 100. Estos pueden representarse de manera eficiente mediante un arreglo unidimensional o una estructura de datos lineal dinámica, como un `vector` en C++.

Sin embargo, no es necesario asignar memoria adicional para el alfabeto, ya que el programa no opera explícitamente con los símbolos. Además, el problema garantiza que cualquier palabra de entrada estará construida únicamente con símbolos válidos del alfabeto, lo que elimina la necesidad de almacenamiento extra.

Dado que el conjunto de estados finales  $F$  es siempre un subconjunto de  $Q$  (es decir,  $F \subseteq Q$ ), todos los estados, incluidos los estados finales, pueden almacenarse en una única estructura de datos. Como el autómata consta exactamente de  $N$  estados, una representación óptima es un `vector<bool>`, donde el índice (en el rango de 1 a  $N$ ) corresponde a un estado, y el valor almacenado en cada índice (`true` o `false`) indica si el estado es final o no. Esta estructura simplifica la verificación de aceptación de una secuencia de entrada: la secuencia será válida si el último estado alcanzado tiene un valor `true` en el vector de estados finales.

El estado inicial puede representarse mediante una única variable entera, ya que, por definición, cada autómata  $M$  tiene exactamente un estado inicial.

El aspecto más complejo de la implementación es la representación de las transiciones de estado definidas por la función de transición  $\delta$ . Inicialmente, el autómata fue conceptualizado como un grafo dirigido ponderado, que podría representarse mediante una matriz de adyacencia. En este modelo, cada estado es un nodo, y cada nodo mantiene una lista de pares ordenados (estado\_vecino, símbolo), donde *estado\_vecino* representa el estado destino y *símbolo* corresponde al símbolo de transición. Este enfoque da lugar a un arreglo de vectores que almacena una estructura de datos definida, formando efectivamente una matriz bidimensional.

Sin embargo, esta estructura resulta ineficiente, ya que acceder al estado de transición correcto implica una complejidad de  $O(N)$ , donde  $N$  es el número total de estados. Para optimizar el tiempo de acceso, se adoptó un enfoque más eficiente: una estructura de datos lineal dinámica que guarda tablas hash (una estructura de datos asociativa que mapea claves a valores (key-value pair)).

En C++, esta estructura se implementa como un `vector<unordered_map<char, int>>`, donde los índices de 1 a  $N$  en el `vector<>` externo corresponden a los estados en  $Q$ . Cada índice  $i$  almacena una tabla hash (`unordered_map<char, int>`), que consiste en pares clave-valor: un `char` como clave y un `int` como valor.

La clave representa una letra  $w_i \in \Sigma$ , que esencialmente es un símbolo que desencadena una transición de un estado a otro. El valor, un `int`, representa la salida de la función de transición  $\delta$ , es decir, el estado al que se mueve el autómata con dicho símbolo. En conjunto, la estructura `vector<unordered_map<char, int>>` codifica todas las transiciones de estado  $\delta(I, X) = J$ , donde  $I, J \in Q$  y  $X \in \Sigma$ .

En esta notación,  $I$  es el índice  $i$  del `vector<>` externo,  $X$  es la clave dentro de la tabla hash almacenada en el índice  $i$ , y  $J$  es el valor correspondiente del par clave-valor.

¿Por qué el valor del par clave-valor es un `int`? La razón es que el problema garantiza implícitamente que cada estado tiene exactamente una transición para cada símbolo en el alfabeto. Como resultado, la entrada siempre consta de exactamente  $D$  transiciones, donde  $D = |Q| \times |\Sigma|$ .

Finalmente, el uso de un `vector<unordered_map<char, int>>` es la estructura más eficiente para representar las transiciones de estado, ya que permite acceder al siguiente estado en tiempo constante  $O(1)$  a partir de un símbolo de entrada. Esto representa una mejora significativa en la complejidad temporal en comparación con el enfoque inicial.

Para visualizar mejor cómo se representará un AFD mediante las estructuras de datos descritas anteriormente, usemos el siguiente AFD  $M_2$ :

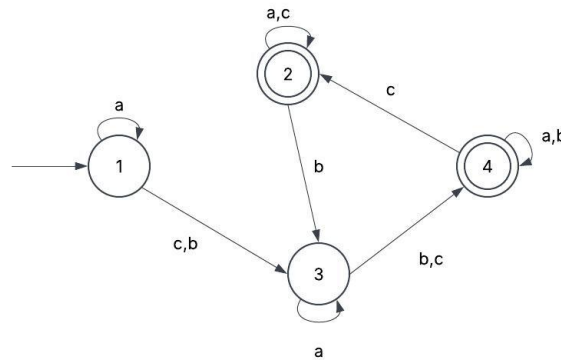


Figure 7: Representación gráfica del autómata  $M_2$

$M_2 = (Q, \Sigma, \delta, q_0, F)$  tiene que  $Q = \{1, 2, 3, 4\}$ ,  $\Sigma = \{a, b, c\}$ ,  $F = \{2, 4\}$ ,  $q_0 = 1$ , y  $\delta$  representada por la siguientes transiciones  $\delta(I, X) = J$  donde  $I, J \in Q$  y  $X \in \Sigma$ :

1.  $\delta(1, a) = 1$
2.  $\delta(1, b) = 3$
3.  $\delta(1, c) = 3$
4.  $\delta(2, a) = 2$
5.  $\delta(2, b) = 3$
6.  $\delta(2, c) = 2$
7.  $\delta(3, a) = 3$

8.  $\delta(3, b) = 4$
9.  $\delta(3, c) = 4$
10.  $\delta(4, a) = 4$
11.  $\delta(4, b) = 4$
12.  $\delta(4, c) = 2$

Para la siguiente explicación, consulte la Figura 8.

Dado que  $F \subseteq Q$ ,  $F$  y  $Q$  serán representados mediante `vector<bool> Q`, donde el índice  $i$ -ésimo representa el estado  $i$  en  $Q$ .

Cada índice de `vector<bool> Q` indica si un estado es final o no: *true* significa que  $q \in F$ , mientras que *false* indica que  $q$  es parte de  $Q$  pero no un estado final ( $q \notin F$ ).

Como se muestra en la Figura 8, solo los índices 2 y 4 tienen el valor *true*, lo que indica que los estados 2 y 4 son los únicos estados finales del autómata  $M_2$ . Esto también garantiza que todos los estados en  $Q$  estén correctamente identificados.

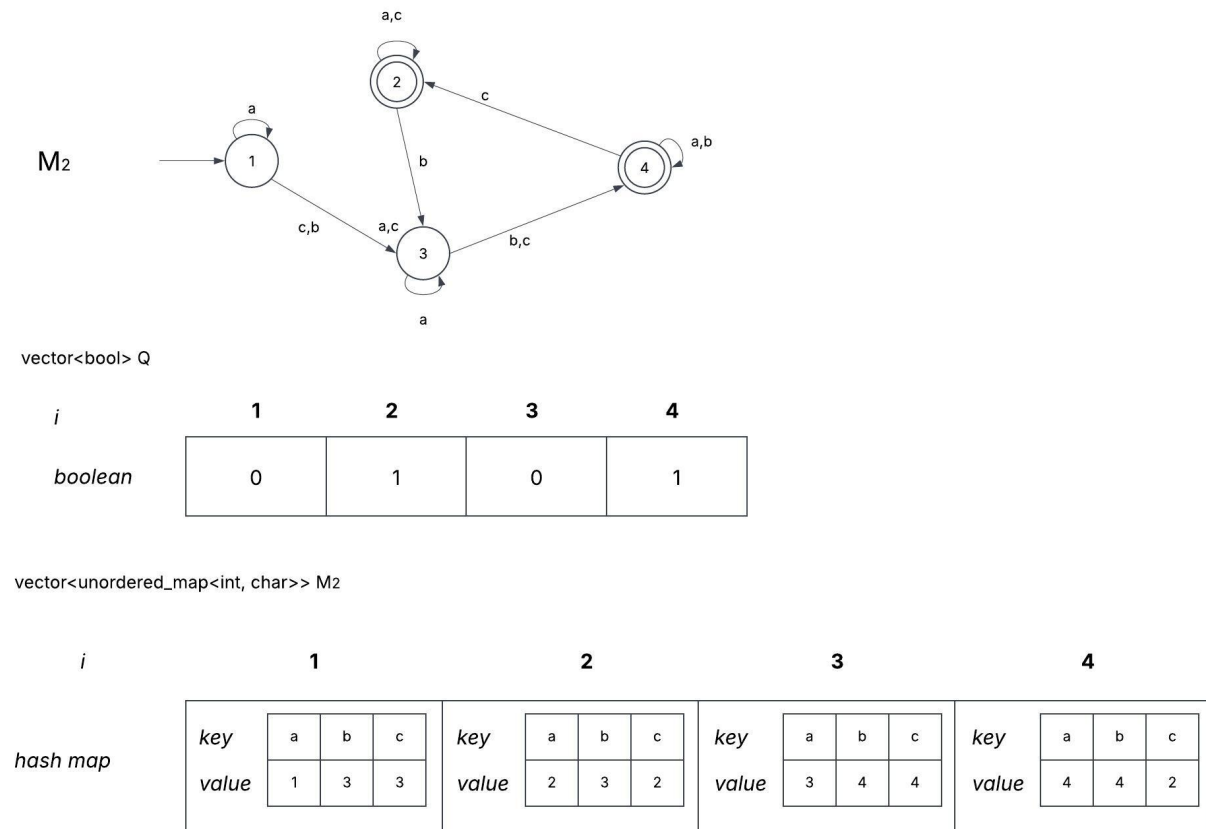


Figure 8: Representación gráfica del autómata  $M_2$  en estructuras de datos de C++

Ahora, todas las transiciones de  $M_2$ , representadas como  $\delta(I, X) = J$ , se almacenan en un `vector<unordered_map<int, char>> M2`. En la Figura 8, se puede visualizar la estructura de  $M_2$ , donde cada índice  $i$  del vector externo corresponde a un estado  $q \in Q$ . En cada índice se almacena una tabla hash que funciona como un diccionario, donde cada clave única se asocia a un valor correspondiente. En el contexto de un autómata, la clave representa un símbolo del alfabeto  $\Sigma$ , mientras que el valor representa un estado en  $Q$ . Esta estructura es particularmente eficiente, ya que permite acceder directamente al siguiente estado (el valor/value en la tabla hash) dado un carácter de entrada (la clave/key en la tabla hash).

## Análisis de técnicas y algoritmos

The structural requirements for solving the problem have already been established, but now we need to define how to determine if a string of characters  $W$  is accepted or rejected by a DFA  $M = (Q, \Sigma, \delta, q_0, F)$ .

## Recursion

La recursión es una técnica computacional en la que una función se define en términos de sí misma para resolver un problema dividiéndolo en subproblemas más pequeños y manejables de la misma naturaleza. Una función recursiva consta de:

1. **Caso base(s):** Una condición que termina las llamadas recursivas, evitando la recursión infinita.
2. **Caso recursivo(s):** Una regla que reduce el problema en uno o más subproblemas más pequeños que progresivamente se acercan al caso base.

**Definición matemática:** Una función recursiva puede expresarse formalmente como:

$$f(n) = \begin{cases} \text{Caso base,} & \text{si } n \text{ satisface una condición de detención} \\ \text{Caso recursivo,} & \text{en caso contrario} \end{cases}$$

Donde  $f(n)$  se define en términos de  $f(k)$  para algún  $k < n$ , asegurando la convergencia hacia el caso base (Koshy, 2004).

¿Por qué usar recursión? Para comprender mejor por qué la recursión es un enfoque adecuado, analicemos un ejemplo utilizando el DFA  $M_2$  de la Figura 7. Supongamos que queremos determinar si la cadena  $W = \text{"abaacbac"}$  es aceptada por  $M_2$ .

Podemos descomponer el problema en subproblemas más pequeños. Primero, verificamos si el primer carácter  $w_0$  de  $W$  corresponde a una transición válida en  $M_2$ , conduciendo a un estado de salida. Luego, comprobamos si el segundo carácter  $w_1$  es aceptado como entrada en el estado resultante, generando otra transición válida. Este proceso se repite para cada carácter subsiguiente, reduciendo el problema a la misma pregunta fundamental en cada paso:

*Dado el estado actual, ¿existe una transición que acepte el símbolo  $w_i$  y conduzca a otro estado?*

Aplicando recursión, resolvemos sistemáticamente el problema manejando una transición a la vez hasta que la cadena se procese completamente o no haya ninguna transición válida disponible.

Para nuestra aplicación, podemos definir la función recursiva  $f(n)$  como:

$$f(s, n) = \begin{cases} \text{false} & \text{si } n = |W| \text{ y } s, \text{ el estado actual no es un estado final,} \\ \text{true} & \text{si } n = |W| \text{ y } s \text{ y el estado actual es un estado final,} \\ \text{true} & \text{si existe una transición válida desde } s, \text{ el estado actual, para el símbolo en la posición } n, \\ f(\delta(s, W(n)), n + 1) & \text{si existe una transición válida y hacemos la recursión al siguiente símbolo en la cadena,} \\ \text{false} & \text{si no existe una transición válida para el símbolo actual en la posición } n. \end{cases}$$

Donde  $n$  es la  $i$ -ésima posición de un carácter en la cadena  $W$ ,  $w \in W$ , y  $\delta(s, W(n))$  define el siguiente estado.

Gráficamente, las llamadas recursivas para encontrar si la cadena  $W = \text{"abaacbac"}$  es aceptada por  $M_2$  se muestran en la siguiente figura:

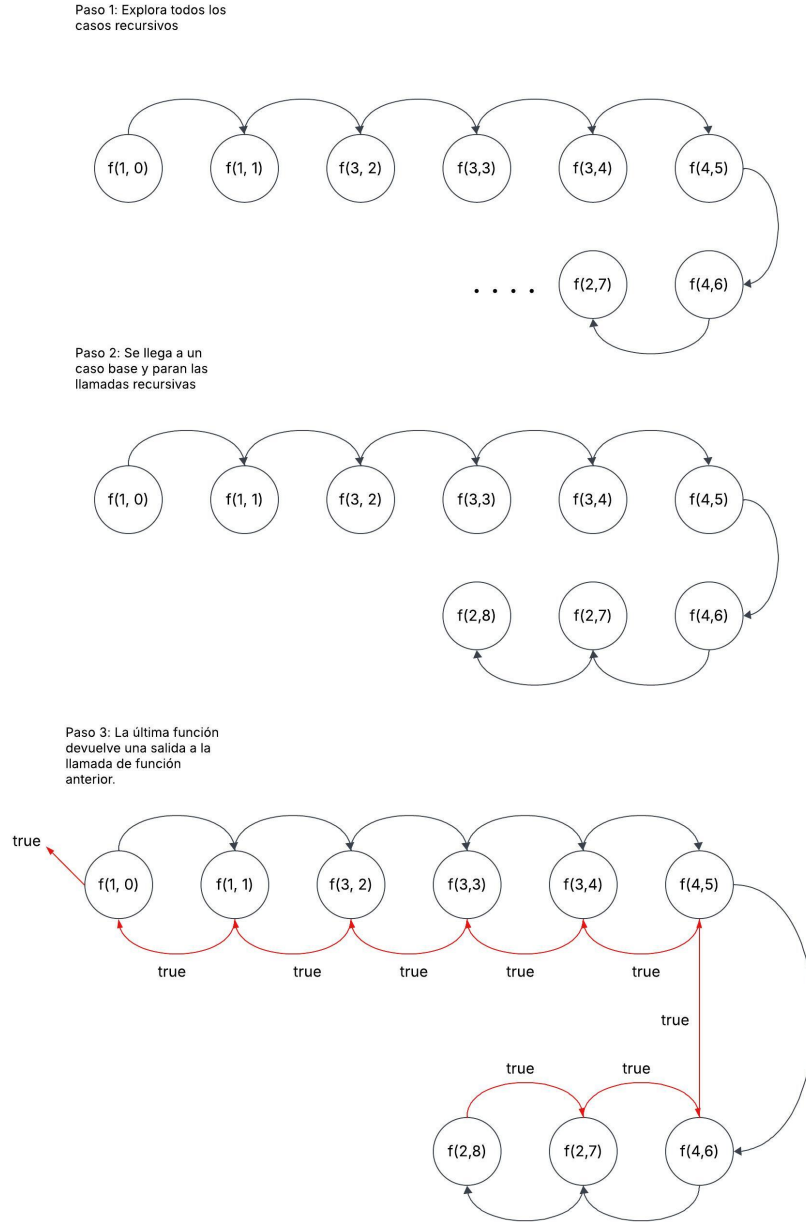


Figure 9: Representación gráfica del autómata  $M_2$  en estructuras de datos de C++

La cadena  $W = \text{"abaacbac"}$  es aceptada por  $M_2$  porque la llamada recursiva inicial finalmente devuelve **true**.

**Paso 1: Llamada Inicial** Comenzamos con  $f(1,0)$ , que verifica si existe una transición válida para el símbolo en la posición  $n = 0$  de  $W$ , donde  $w = a$ , desde el estado  $s = 1$ . Dado que existe una transición

válida, la función realiza una llamada recursiva a  $f(\delta(1, W(0)), 1)$ , donde  $\delta(1, W(0)) = 1$ .

**Paso 2: Llamadas Recursivas** Cada llamada de función verifica si existe una transición válida para el siguiente símbolo en  $W$  y avanza al estado correspondiente. La recursión continúa mientras se encuentren transiciones válidas, siguiendo la secuencia:

$$f(1, 0) \rightarrow f(1, 1) \rightarrow f(3, 2) \rightarrow f(3, 3) \rightarrow f(3, 4) \rightarrow f(4, 5) \rightarrow f(4, 6) \rightarrow f(2, 7) \rightarrow f(2, 8)$$

La recursión continúa hasta que  $n$  alcanza la longitud de  $W$ , es decir,  $n = |W| = 8$ .

**Paso 3: Caso Base y Propagación de Valores** En  $f(2, 8)$ , dado que  $n = |W|$ , verificamos si el estado actual  $s = 2$  es un estado final (de aceptación). Como  $s = 2 \in F$ , la función devuelve **true**.

Cada llamada de función previa propaga este valor **true** hacia arriba en la cadena recursiva:

$$f(2, 8) \rightarrow f(2, 7) \rightarrow f(4, 6) \rightarrow f(4, 5) \rightarrow f(3, 4) \rightarrow f(3, 3) \rightarrow f(3, 2) \rightarrow f(1, 1) \rightarrow f(1, 0)$$

Finalmente, la llamada inicial  $f(1, 0)$  recibe **true**, confirmando que  $W$  es aceptada por  $M_2$ .

Es importante destacar que nuestra cadena recursiva refleja el comportamiento de la computación determinista (ver Figura 10), ya que sigue un único camino fijo, a diferencia de la computación no determinista, donde se exploran múltiples caminos en paralelo (Sisper, 2006).

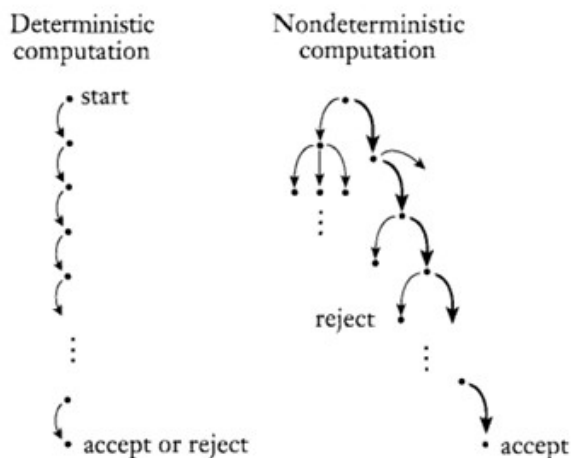


Figure 10: Representación gráfica de la diferencia entre computación determinista y computación no determinista

## Función find()

En la sección anterior sobre recursión, mencionamos que cada llamada a la función requiere un estado de transición válido para poder proceder a otra llamada. Pero, ¿cómo se determina exactamente un estado de transición válido  $\delta(s, w)$ , donde  $s$  representa el estado actual y  $w$  es un símbolo del alfabeto  $\Sigma$ ? Afortunadamente, los estados de transición se almacenan en un `vector<unordered_map<char, int>>`.

En C++, `unordered_map` es un contenedor de la Standard Template Library (STL) que proporciona una forma eficiente de almacenar pares clave-valor, ofreciendo una complejidad temporal promedio de  $O(1)$  para

las operaciones de búsqueda, inserción y eliminación (GeeksforGeeks, 2024).

La función `.find()` de `unordered_map` permite buscar una clave específica en el mapa. Devuelve un iterador al elemento si la clave se encuentra; de lo contrario, devuelve `unordered_map::end()`. En nuestra representación de un autómata finito determinista mediante `vector<unordered_map<char, int>>`, la clave `char`, que corresponde a un símbolo de entrada del alfabeto  $\Sigma$ , se puede encontrar de manera eficiente en tiempo constante  $O(1)$ .

## Implementación de solución en C++

Utilizando las estructuras de datos, técnicas, algoritmos y condiciones mencionadas anteriormente, se desarrolló un programa en C++ que lee un autómata finito determinista  $M = (Q, \Sigma, \delta, q_0, F)$ , procesa  $D$  cadenas y determina cuáles de las  $D$  cadenas son aceptadas o rechazadas por  $M$ .

```
1  /*
2  author: ash_c4t
3  */
4  #include <bits/stdc++.h>
5
6  using namespace std;
7
8  bool verifyW(vector<unordered_map<char, int>>& AFD, int currentSt, string W, int
   wIdx, vector<bool> finalStates){
9      if(wIdx == W.size()) return finalStates[currentSt]; //base case: when position
   is the same as size of string W
10     if(AFD[currentSt].find(W[wIdx]) != AFD[currentSt].end()){ //recursive case:
   when a valid transition state is found
11         return verifyW(AFD, AFD[currentSt][W[wIdx]], W, wIdx+1, finalStates);
12     }
13     return false; //case: when no valid transition state is found
14 }
15
16 int main()
17 {
18     ios_base::sync_with_stdio(0);
19     cin.tie(0);
20
21     int N, S, D, q, T, C;
22     //size of Q, size of alphabet, #of transitions, initial state, # of final
   states, # of strings to be tested
23     cin>>N>>S>>D>>q>>T>>C;
24
25     //read chars in alphabet
26     //char alphabet[S+5]; no need to store alphabet
27     for(int i=0; i<S; i++){
28         char letter;
29         cin>>letter;
30         //cin>>alphabet[i];
31     }
32
33     //read final states
34     //int setF[T+5];
35     vector<bool> finalStates(N+1, false);
36     for(int i=0; i<T; i++){
37         int s;
38         cin>>s;
39         finalStates[s] = true;
```

```

40     }
41
42     //read state transitions
43     int I, J;
44     char X;
45     //represent automaton as a vector of hash maps
46     vector<unordered_map<char, int>> AFD(D+5); //vector idx -> state (I), char key
47         -> ASCII that belongs to alphabet (X),
48     //int val -> transition states (J)
49     for(int i=1; i<=D; i++){
50         cin>>I>>X>>J;
51         AFD[I][X]=J;
52     }
53
54     //read and evaluate strings
55     string W;
56     cin.ignore();
57     while(C--){
58         bool isValid = false; //assume that current string is not accepted by AFD
59         getline(cin, W);
60         if(T==0){ //no accepted states/final states
61             cout<<"RECHAZADA\n"; //no string is accepted, not even empty string,
62                 only recognizes the empty language
63         }
64         else{
65             //check if a start state is a final state
66             if(finalStates[q] && W==""){
67                 cout<<"ACEPTADA\n"; //empty strings are accepted
68             }
69             else{
70                 //call function
71                 if(verifyW(AFD, q, W, 0, finalStates)) cout<<"ACEPTADA\n";
72                 else cout<<"RECHAZADA\n";
73             }
74         }
75     }

```

## Leyendo entradas

El programa comienza leyendo seis enteros que representan el tamaño del conjunto de estados  $Q$ , el tamaño del alfabeto  $\Sigma$ , el número de transiciones, el estado inicial, el número de estados finales y la cantidad de cadenas que se probarán. Luego, se leen los caracteres del alfabeto, aunque no se almacenan explícitamente. Posteriormente, se leen los estados finales y se guardan en un vector de valores booleanos para facilitar su verificación. Las transiciones del autómata se almacenan en un vector de mapas no ordenados, donde la clave es un carácter del alfabeto y el valor es el estado de destino. Finalmente, el programa lee y evalúa las cadenas de entrada para determinar si son aceptadas o rechazadas por el autómata finito determinista.

## Consideración de Casos Límite

Es absolutamente necesario considerar los casos límite para construir una solución completa que determine si  $C$  cadenas  $W$  son aceptadas por un autómata finito determinista  $M = (Q, \Sigma, \delta, q_0, F)$  con  $D$  transiciones, respetando las siguientes restricciones:

$$1 \leq |Q|, |\Sigma|, C \leq 100, \quad 1 \leq D \leq 10^4, \quad 1 \leq q_0 \leq |Q|, \quad 0 \leq |F| \leq |Q|, \quad 0 \leq |W| \leq 100.$$



Un caso límite se refiere a un escenario único en el que un programa o algoritmo es probado con entradas que representan los valores extremos de su rango esperado (valores de frontera) o con entradas inusuales pero válidas. En nuestro problema, se nos indica explícitamente que el programa debe ser capaz de procesar una cadena vacía  $W$  y un conjunto vacío  $F$ , por lo que son casos límite que debemos considerar. ¿Cómo maneja el programa estos casos?

Primero, revisemos la definición de un autómata finito determinista (AFD):

Un autómata finito consta de cinco componentes: un conjunto de estados, un alfabeto de entrada, reglas de transición, un estado inicial y un conjunto de estados de aceptación, formando una 5-tupla.

La definición formal establece que un AFD sin estados de aceptación ( $F = \emptyset$ ) es válido. Además, la función de transición  $\delta$  garantiza que cada estado tiene exactamente una transición para cada símbolo de entrada, lo que significa que cada estado tiene un camino definido para cada entrada posible (Sipser, 2006).

El problema permite explícitamente que  $F$  esté vacío ( $F = \emptyset$ ). Según la definición de un AFD, si no hay estados de aceptación, ninguna cadena ingresada será aceptada, ni siquiera la cadena vacía. Esto implica que el AFD solo reconoce el lenguaje vacío. Para manejar esto en el programa, se introduce una condición antes de llamar a la función recursiva que verifica si una cadena es aceptada. Esta condición se implementa como:

```
if(T == 0)
```

donde  $T = |F|$ . Si esta condición se cumple, cualquier cadena de entrada  $W$  es rechazada inmediatamente y el programa no continúa con la función recursiva. La siguiente figura modela este caso especial:

Supongamos que tenemos  $M_3 = (Q, \Sigma, \delta, q_0, F)$ , donde  $Q = \{1, 2\}$ ,  $\Sigma = \{A, S\}$ ,  $q_0 = 2$ ,  $F = \emptyset$ , y  $\delta$  está definida por:

```

delta(1, A) = 1
delta(1, S) = 2
delta(2, A) = 2
delta(2, S) = 1

```

Otro caso especial a considerar ocurre cuando el estado inicial  $q_0$  también es un estado final ( $q_0 \in F$ ). En este caso:

- Si  $W$  es una cadena vacía y  $q_0 \in F$ , es aceptada por  $M$ .
- De lo contrario, debe ser evaluada por la función recursiva.

Esta lógica se implementa con una estructura **if-else** en el código:

```

if (finalStates[q] && W == ""){ . . . }
else{ . . . }

```

donde:

- **finalStates[]** representa el conjunto de estados finales  $F$ ,
- $q$  es el estado inicial  $q_0$ ,
- **finalStates[q]** es un valor booleano que indica si un estado es final o no,

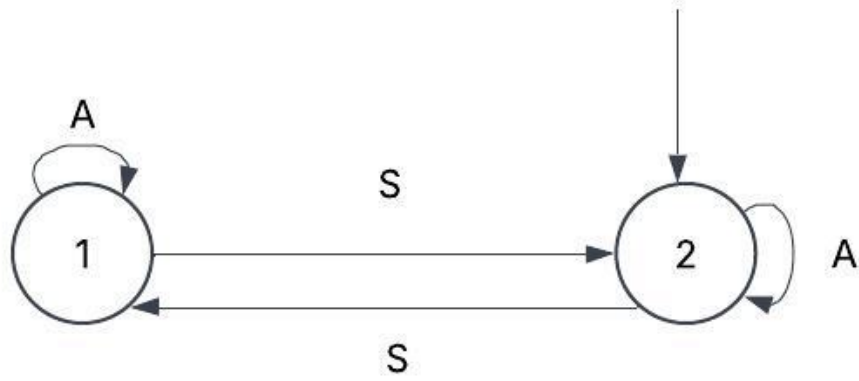


Figure 11:  $M_4 = (Q, \Sigma, \delta, q_0, F)$

-  $W$  es la cadena de entrada.

Si la condición es falsa, el programa procede a procesar  $W$  en el bloque **else**.

La siguiente figura representa gráficamente este caso especial.

Supongamos que tenemos  $M_3 = (Q, \Sigma, \delta, q_0, F)$ , donde  $Q = \{1, 2, 3, 4, 5\}$ ,  $\Sigma = \{x, y, z\}$ ,  $q_0 = 1$ ,  $F = \{1, 4, 5\}$ , y  $\delta$  está definida por:

$$\begin{aligned}
 \delta(1, x) &= 1 \\
 \delta(1, y) &= 2 \\
 \delta(1, z) &= 3 \\
 \delta(2, x) &= 2 \\
 \delta(2, y) &= 3 \\
 \delta(2, z) &= 5 \\
 \delta(3, x) &= 4 \\
 \delta(3, y) &= 3 \\
 \delta(3, z) &= 3 \\
 \delta(4, x) &= 4 \\
 \delta(4, y) &= 5 \\
 \delta(4, z) &= 4 \\
 \delta(5, x) &= 5 \\
 \delta(5, y) &= 5 \\
 \delta(5, z) &= 5
 \end{aligned}$$

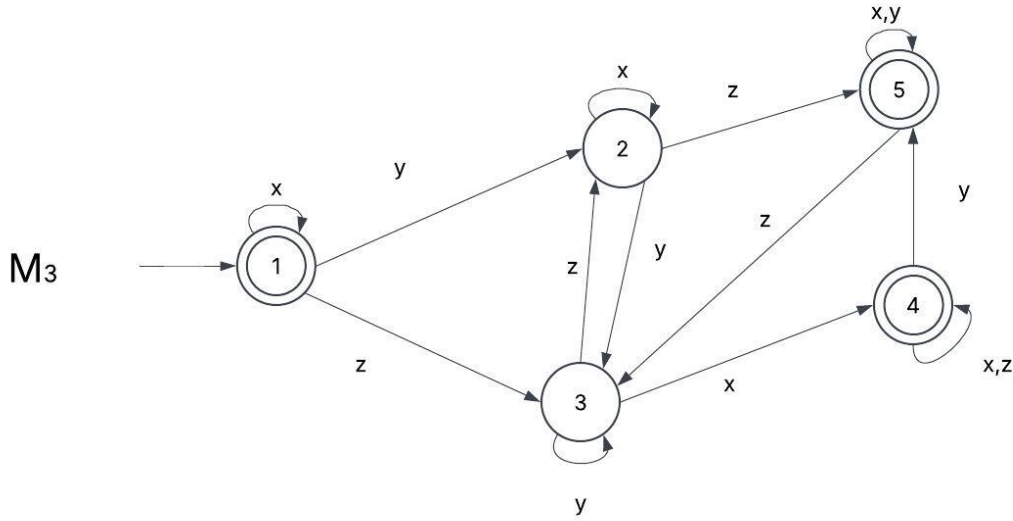


Figure 12:  $M_4 = (Q, \Sigma, \delta, q_0, F)$

### Función recursiva verifyW()

La función `verifyW()` declarada en el programa anterior se puede refactorizar a la siguiente función:

```

1  bool verifyW(vector<unordered_map<char, int>>& AFD, int currentSt, string W,
2      int wIdx, vector<bool> finalStates){
3      if(wIdx == W.size()) return finalStates[currentSt]; //base case
4      if(AFD[currentSt].find(W[wIdx]) == AFD[currentSt].end()) return false; //base
5      case
6      return verifyW(AFD, AFD[currentSt][W[wIdx]], W, wIdx+1, finalStates); //
7      recursive case
8  }
```

Sin embargo, la función recursiva  $f(s, n)$  todavía aplica:

$$f(s, n) = \begin{cases} \text{false} & \text{si } n = |W| \text{ y } s, \text{ el estado actual no es un estado final,} \\ \text{true} & \text{si } n = |W| \text{ y } s \text{ y el estado actual es un estado final,} \\ \text{true} & \text{si existe una transición válida desde } s, \text{ el estado actual, para el símbolo en la posición } n, \\ f(\delta(s, W(n)), n+1) & \text{si existe una transición válida y hacemos la recursión al siguiente símbolo en la cadena,} \\ \text{false} & \text{si no existe una transición válida para el símbolo actual en la posición } n. \end{cases}$$

, donde  $n$  es la posición de un carácter en una cadena  $W$  y  $s$  es un estado tal que  $s \in Q$ . Variable  $n$  corresponds to the `int wIdx` parameter and  $s$  corresponds to `int currentSt` in function `verifyW()`. El resto de los parámetros en `verifyW()`, `vector<unordered_map<char, int>>& AFD`, `String W`, y `vector<bool> finalStates` se utilizan para establecer casos base o para operaciones de comparación simples, pero nunca cambian: es decir, los argumentos de estos parámetros permanecen iguales cada vez que la función recursiva

se llama a sí misma.

1. **Primer caso base:** `if(wIdx == W.size())`

El primer caso base de la función recursiva `verifyW()` establece que la recursión debe detenerse y retornar el valor booleano almacenado en el índice `currentSt` del `vector<bool> finalStates` cuando `wIdx` alcanza la longitud de la cadena `W`. Esto significa que no quedan más caracteres en `W` por procesar, por lo que la función verifica si el estado actual es un estado final.

2. **Segundo caso base:** `if(AFD[currentSt].find(W[wIdx]) == AFD[currentSt].end())`

El segundo caso base garantiza que la recursión se detenga y retorne `false` si el estado actual no tiene una transición para el carácter de entrada `W[wIdx]`. En otras palabras, si no existe una transición válida desde el estado actual con el símbolo de entrada actual, la cadena es rechazada inmediatamente.

3. **Caso recursivo:** `verifyW(AFD, AFD[currentSt][W[wIdx]], W, wIdx+1, finalStates)`

Si no se cumple ninguno de los casos base—es decir, aún no hemos llegado al final de la cadena y existe una transición válida para el carácter de entrada actual—entonces la función realiza una llamada recursiva. El estado se actualiza al siguiente estado de acuerdo con la función de transición  $\delta(\text{currentSt}, W[wIdx])$ , y `wIdx` se incrementa para procesar el siguiente carácter en `W`. Este proceso continúa hasta que se alcanza un caso base.

## Análisis de complejidad computacional

La complejidad computacional se refiere al estudio de los recursos necesarios para resolver un problema, enfocándose particularmente en el tiempo (cuánto tiempo tarda una computación) y el espacio (cuánta memoria se necesita). Analiza la eficiencia de los algoritmos y clasifica los problemas según la cantidad de tiempo y espacio que requieren, típicamente en relación con el tamaño de la entrada. El objetivo principal es comprender cómo crecen los requisitos de recursos a medida que aumenta el tamaño de la entrada. El esfuerzo computacional principal se distribuye entre dos fases del programa: leer y almacenar el autómata y las cadenas a procesar, y procesar la cadena de manera recursiva (Stanford Encyclopedia of Philosophy, 2015).

### Análisis de la Complejidad Temporal

La entrada consiste en  $N$  estados,  $S$  caracteres ASCII,  $D$  transiciones, un estado inicial  $q$ ,  $T$  estados finales y  $C$  cadenas. Solo leemos  $S$  caracteres,  $T$  estados finales y  $D$  transiciones, por lo que podemos decir que realizamos  $S + T + D$  operaciones. Según los límites definidos por el problema, realizaremos como máximo  $100 + 100 + 10^4$  operaciones, lo que podemos aproximar a  $10^4$  operaciones. Esto se clasifica como una complejidad lineal,  $O(N)$ , porque la cantidad de operaciones realizadas es proporcional al tamaño de la entrada de  $S + T + D$  (ver figura 13).

Las cadenas se procesan un carácter a la vez por llamada recursiva, por lo que una llamada recursiva significa que se realiza una sola operación. ¿Por qué? Cada caso, ya sea recursivo o base, realiza una operación, incluso al utilizar la función `.find()`. Sin embargo, toda la cadena necesita ser procesada, por lo que como máximo se harán  $|W|$  llamadas recursivas. Por lo tanto, procesar una cadena completa implica que se realicen  $L$  operaciones, donde  $L$  es equivalente a la longitud de la cadena  $W$  o  $|W|$ . Los límites definidos por el problema indican que una cadena  $W$  tendrá como máximo una longitud de 100, por lo que podemos aproximar la complejidad temporal a constante,  $O(L)$ .

Ahora, el problema nos pide procesar  $C$  cadenas, y en el peor de los casos tendríamos que procesar cada cadena con la función recursiva. Esto significa que tendremos que realizar como máximo  $C \times L$  operaciones, porque cada carácter de cada cadena debe ser explorado. Los límites indican que tendremos que leer como máximo 100 cadenas, y dado que el límite máximo de  $C$  es igual al límite máximo de  $L$ , podemos decir que  $L = C$ . Esto implica una complejidad temporal cuadrática,  $C \times C$ , que puede expresarse como  $O(M^2)$  (ver

figura 13).

En general, la complejidad temporal es  $O(N + M^2)$ , donde  $N = 10^4$  y  $M = 100$ , lo que puede simplificarse a  $O(M^2)$ . En conclusión, el programa realiza como máximo  $10^4$  operaciones y tiene una complejidad temporal cuadrática.

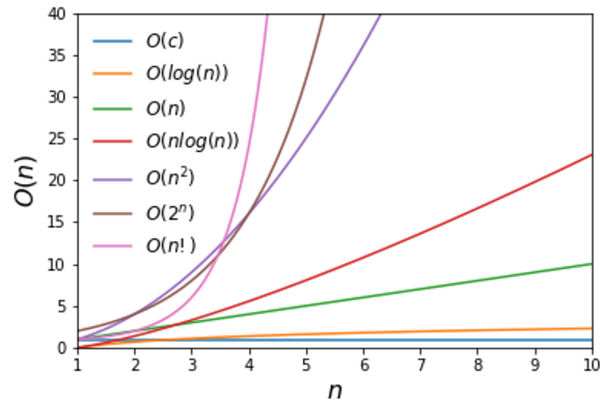


Figure 13: Big O Notation

### Análisis de la Complejidad Espacial

El programa utiliza dos estructuras de datos para almacenar  $N$  estados y  $D$  transiciones. Además, cuando una cadena es procesada mediante la función recursiva, se forma una pila de recursión que alcanza una profundidad de  $|W|$ . Se procesan  $C$  cadenas, por lo que la complejidad espacial de la pila de recursión es  $C \times L$ , donde  $L$  es la longitud de una cadena. En general, la complejidad espacial es  $O(N + C \times L)$ , la cual puede reducirse a una complejidad espacial cuadrática  $O(N^2)$ .

## Veredicto del juez en línea en OmegaUp

### Submissions







 Date and Time	Language	Percentage	Execution	Output	 Memory	 Runtime	Actions
2025-02-23 17:38	cpp11-gcc	100.00%	Finished	Correct 	3.37 MB	0.01 s	

Figure 14: Representación gráfica de la diferencia entre computación determinista y computación no determinista

### Code

```
1  /*
2  author: ash_c4t
3  */
4  #include <bits/stdc++.h>
5
6  using namespace std;
7
8  void verifyS(vector<unordered_map<char, int>>& AFD, int currentSt, string W, int wIdx, bool& isValid, vector<bool> finalStates){
9      if(wIdx == W.size()){
10         if(!finalStates[currentSt]) isValid = false;
11         return;
12     }
```

### Cases

Group	Case	Verdict	Score
1			0.16666666666666666 / 0.16666666666666666
	1	AC	0.16666666666666666 / 0.16666666666666666
2			0.16666666666666666 / 0.16666666666666666
	2	AC	0.16666666666666666 / 0.16666666666666666
3			0.16666666666666666 / 0.16666666666666666
	3	AC	0.16666666666666666 / 0.16666666666666666
4			0.16666666666666666 / 0.16666666666666666
	4	AC	0.16666666666666666 / 0.16666666666666666
5			0.16666666666666666 / 0.16666666666666666
	5	AC	0.16666666666666666 / 0.16666666666666666
6			0.16666666666666666 / 0.16666666666666666
	6	AC	0.16666666666666666 / 0.16666666666666666

### Download

- Download code
-

Figure 15: Representación gráfica de la diferencia entre computación determinista y computación no determinista

## 5. Conclusiones

Los autómatas finitos son modelos computacionales fundamentales utilizados para reconocer lenguajes regulares y resolver problemas de búsqueda de patrones. Funcionan mediante transiciones entre estados basadas en símbolos de entrada, lo que los hace útiles en aplicaciones como el análisis léxico, la búsqueda de texto y la verificación de protocolos.

Un Autómata Finito Determinista (AFD) es un tipo de autómata en el que cada estado tiene exactamente una transición por símbolo de entrada, lo que garantiza que la computación siga un camino único y predecible. Los AFD son sencillos de implementar y aseguran un procesamiento eficiente con una complejidad temporal de  $O(|W|)$ , donde  $|W|$  es la longitud de la cadena de entrada. Sin embargo, en algunos casos pueden requerir un número elevado de estados.

Por otro lado, un Autómata Finito No Determinista (AFND) permite múltiples transiciones posibles para un mismo símbolo de entrada, lo que permite representar ciertos lenguajes de manera más compacta y flexible. A pesar de su aparente complejidad, todo AFND puede convertirse en un AFD equivalente, aunque el AFD resultante puede tener un número exponencialmente mayor de estados. Los AFND son especialmente útiles en el análisis teórico y en el diseño de algoritmos de búsqueda de patrones, como los utilizados en expresiones regulares (Sisper, 2006).

En general, los autómatas finitos constituyen la base para el estudio de los lenguajes formales y la computación. Mientras que los AFD ofrecen eficiencia y simplicidad, los AFND proporcionan mayor expresividad y flexibilidad. Su estudio es crucial en áreas como el diseño de compiladores, la inteligencia artificial y la seguridad informática, donde el reconocimiento de patrones y la toma de decisiones automatizada desempeñan un papel clave.

## Referencias Bibliográficas

## References

- [1] GeeksforGeeks. (2024). *unordered\_map find in C++ STL*. [https://www.geeksforgeeks.org/unordered\\_map-find-in-c-stl/](https://www.geeksforgeeks.org/unordered_map-find-in-c-stl/)
- [2] Koshy, T. (2004). *Discrete mathematics with applications*. Elsevier Science & Technology.
- [3] Kozen, D. C. (1997). *Automata and computability*. Springer New York.
- [4] OmegaUp. (2022). 15320. *Simulación de un Autómata Finito Determinista*.
- [5] Sipser, .(2006). *Introduction to the Theory of Computation* (2nd ed.). Massachusetts Institute of Technology.
- [6] Stanford Encyclopedia of Philosophy. (2015). *Computational Complexity Theory*. <https://plato.stanford.edu/entries/computational-complexity/#Bib>