UNIVERSIDAD DE GRANADA

Modelos Avanzados de Computación

Apuntes de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Modelos Avanzados de Computación

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

Capítulo 1

Alcance de estos apuntes

El documento sigue el programa de la guía docente de la asignatura (código 296113D, Grado en Ingeniería Informática, especialidad de Computación y Sistemas Inteligentes). Recoge la teoría de los seis temas, las relaciones de problemas resueltas y las diez convocatorias comprendidas entre 2021 y 2025.

La cobertura no es uniforme, y conviene saberlo antes de usarlo:

Tema del programa	Cobertura
1. Máquinas de Turing. Funciones y lenguajes calculables	Definiciones, teorema de los complementarios y catálogo de problemas
2. Otros modelos de cálculo. Tesis de Church-Turing	Simulaciones entre modelos y coste asociado
3. Clases de complejidad	Cadena de inclusiones, definiciones y criterios de medida
4. NP-completitud	Método de reducción y mapa de reducciones clásicas
5. Complejidad de problemas aproximados	Solo las clases APX, PTAS y FPTAS, y la mochila como ejemplo
6. Otras clases: jerarquía polinómica, PESPACIO y P	Parcial, dentro de la cadena de inclusiones

Los seminarios de la guía docente —los programas en Python de *What Can Be Computed* y la discusión sobre P frente a NP— no se recogen aquí.

Capítulo 2

Calculabilidad y modelos de cómputo

Cubre los temas 1 y 2 del programa: la máquina de Turing como modelo de referencia, las funciones y lenguajes calculables, los modelos alternativos de cálculo y la tesis de Church-Turing.

2.1 Lenguajes recursivos y recursivamente enumerables

- Recursivo (decidible): existe una MT que siempre se detiene y responde SÍ/NO para cualquier entrada.
- Recursivamente enumerable, r.e. (semidecidible): existe una MT que se detiene y acepta cuando la respuesta es SÍ, pero puede ciclar indefinidamente cuando es NO.
- No semidecidible: no existe siquiera esa MT que acepte los casos afirmativos.

Teorema de los complementarios: 1. Si $\mathtt{L}$ es recursivo, $\mathtt{L}^-$ también lo es. 2. Si $\mathtt{L}$ y $\mathtt{L}^-$ son ambos r.e., entonces $\mathtt{L}$ es recursivo. 3. Si $\mathtt{L}$ es r.e. pero no decidible, entonces $\mathtt{L}^-$ no es semidecidible.

Propiedades de cierre útiles: r.e. es cerrada bajo unión e intersección finitas. La intersección de finitos r.e. es r.e.; por ello un "para todo sobre un conjunto finito de aceptaciones" sigue siendo r.e. (semidecidible), pero no por ello decidible.

Teoría de reducciones ($\mathtt{P1} \propto \mathtt{P2}$ = "P1 se reduce a P2"; P2 es al menos tan difícil): - Si $\mathtt{P1}$ es indecidible, $\mathtt{P2}$ es indecidible. - Si $\mathtt{P1}$ no es semidecidible, $\mathtt{P2}$ no es semidecidible. - Si $\mathtt{P2}$ es semidecidible, entonces $\mathtt{P1}$ también lo es.

2.2 Catálogo de problemas de referencia

Problema	Enunciado	Clasificación
UNIVERSAL (Lu)	¿$\mathtt{M}$ acepta $\mathtt{w}$?	Semidecidible, no decidible
C-UNIVERSAL	¿$\mathtt{M}$ NO acepta $\mathtt{w}$?	No semidecidible
PARADA	¿$\mathtt{M}$ se detiene con $\mathtt{w}$?	Semidecidible, no decidible
DIAGONAL (Ld)	¿$\mathtt{M}$ no acepta su código $\langle\mathtt{M}\rangle$?	No semidecidible
C-DIAGONAL	¿$\mathtt{M}$ sí acepta $\langle\mathtt{M}\rangle$?	Semidecidible
EMPTY / VACÍO (Le)	¿$\mathtt{L(M)}=\emptyset$?	No semidecidible
C-VACÍO (Lne)	¿$\mathtt{L(M)}\neq\emptyset$?	Semidecidible
PCP (Post)	Correspondencias de Post	Indecidible si |alfabeto|$\geq$2; decidible si alfabeto $\mathtt{\{1\}}$

Problema	Enunciado	Clasificación
GIC ambigua	¿una gramática indep. del contexto es ambigua?	Indecidible
GIC genera w / AP acepta w	—	Decidible (CYK)

2.3 Otros modelos de cálculo y tesis de Church-Turing

Simulaciones entre modelos

a) Programa con variables hacia programa Post-Turing:

- Organización: las variables `X1,...,Xm,Z1,...,Zk,Y` se disponen en la cinta separadas por el blanco `#`, con la forma `#V1#V2#...#Vn#`.
- Instrucciones: añadir un símbolo (`Vj` ← `ai Vj`) o borrar el último (`Vj` ← `Vj`−) se simulan con macros Post-Turing que desplazan el cabezal hasta la variable, desplazan el bloque a la derecha para crear o cerrar hueco, y escriben o borran.

b) Programa Post-Turing hacia MT:

- Organización: mismo alfabeto. Por cada instrucción `Ii` un estado `qi`, más un estado final `qf`.
- Traducción uno a uno: `PRINT a` → δ`(qi,b)=(qi+1, a, S)`; `RIGHT` → δ`(qi,b)=(qi+1, b, D)`; `IF ak GOTO L` → desde `qi` a `qL` si se lee `ak`, y a `qi+1` en otro caso.

c) MT hacia programa con variables:

- Organización: `X` = parte izquierda del cabezal, `Z` = símbolo leído, `Y` = parte derecha.
- Transición δ`(qi,aj)=(qm,ak,D)`: se escribe `ak` en `X`, se vacía `Z`, se lee el primer símbolo de `Y` y se traslada a `Z`, y se salta a la etiqueta de `qm`.

Sobrecarga de las simulaciones de cintas

Simulación	Coste
MT multicinta hacia 1 cinta	tiempo $O(t^2)$
Cinta doble infinita hacia semiinfinita (por la derecha)	mismo orden $O(t)$, con 2 pistas
MT multipista hacia 1 cinta	mismo número de pasos

Variables numéricas frente a variables de palabras

- Numéricas: `Xi,Zi,Y`; instrucciones `A`←`A+1`, `A`←`A`−`1`, `IF A`≠`0 GOTO L`.
- Equivalencia: existe una biyección entre palabras y naturales (`N(w)=n`); `+1` equivale a la palabra siguiente en orden lexicográfico.

Tesis de Church-Turing

- Enunciado: "Toda función efectivamente calculable mediante un proceso mecánico bien definido puede ser calculada por una Máquina de Turing".
- "Está demostrada": FALSO (es una tesis, no un teorema). "La computación cuántica la pone en duda": FALSO (afecta a la eficiencia).

Macros y algoritmos clásicos

- `U←V` (copia con variable auxiliar para invertir y reinvertir), `V←−V` (borrar el primer símbolo), suma de cadenas, número siguiente, `V←V+1`, comprobar múltiplo.

Capítulo 3

Clases de complejidad

Cubre los temas 3 a 6 del programa: las clases de complejidad y sus relaciones, la NP-completitud, la complejidad de los problemas de optimización aproximados y las clases de espacio.

3.1 Clases y relaciones entre ellas

Cadena: $\mathtt{L} \subseteq \mathtt{NL} \subseteq \mathtt{P} \subseteq \mathtt{NP} \subseteq \mathtt{PESPACIO} = \mathtt{NPESPACIO} \subsetneq \mathtt{EXP}$. Estrictas demostradas: $\mathtt{L} \subsetneq \mathtt{PESPACIO}$, $\mathtt{NL} \subsetneq \mathtt{PESPACIO}$, $\mathtt{P} \subsetneq \mathtt{EXP}$. PESPACIO=NPESPACIO por Savitch.

Definiciones: $\mathtt{P} = \bigcup \mathtt{TIEMPO}(n^j)$; NP = MT no determinista en tiempo polinómico, o equivalentemente existencia de certificado verificable en tiempo polinómico; coNP = complementario en NP; L/NL = espacio logarítmico det./no det.; PESPACIO = espacio polinómico; $\mathtt{EXP} = \bigcup \mathtt{TIEMPO}(2^{n^j})$.

Medida: sobre la longitud de la representación $\mathtt{n}=\log(\mathtt{x})$. En espacio no se cuentan la cinta de entrada (si no se sobrescribe) ni la de salida (si se escribe en una sola dirección).

Funciones: FNP (búsqueda), FP (se calcula la solución en tiempo polinómico), TFNP (la solución existe siempre). Aproximación: APX (δ constante), PTAS (cualquier precisión en tiempo polinómico en $\mathtt{n}$), FPTAS (polinómico en $\mathtt{n}$ y en $1/(\delta-1)$). DP = $\mathtt{L1} \cap \mathtt{L2}$ con $\mathtt{L1} \in \mathtt{NP}$, $\mathtt{L2} \in \mathtt{coNP}$.

3.2 NP-completitud y reducciones

Cuatro pasos de $\mathtt{P1} \propto \mathtt{P2}$: (1) algoritmo de transformación; (2) eficiencia (espacio logarítmico o tiempo polinómico); (3) SÍ en origen $\Longrightarrow$ SÍ en destino; (4) SÍ en destino $\Longrightarrow$ SÍ en origen.

Para demostrar B NP-completo: probar $\mathtt{B} \in \mathtt{NP}$ y reducir $\mathtt{A} \propto \mathtt{B}$ con A NP-completo conocido.

Mapa: $\mathtt{SAT} \to \mathtt{3\text{-}SAT} \to \mathtt{VC} \to \mathtt{CH} \to \mathtt{TSP}$; $\mathtt{3\text{-}SAT} \to \mathtt{ACTRI} \to \mathtt{SUMA} \to \mathtt{PARTICIÓN} \to \mathtt{MOCHILA}$; $\mathtt{Clique} \equiv \mathtt{Conjunto\ Independiente} \equiv \mathtt{VC}$.

Variantes de SAT: SAT, 3-SAT, MAX2SAT y NAESAT son NP-completos; 2-SAT y Horn-SAT son polinómicos.

Reducibilidad: Karp/logarítmica (una llamada, mapea la respuesta) es más fuerte; Turing (oráculo, varias llamadas) es más débil. NP-difícil: un NP-completo se reduce Turing a él.

3.3 Problemas clásicos y su clasificación

Problema	Clase	Idea
Caminos en grafos dirigidos	NL	adivinar el camino guardando nodo actual y contador

Problema	Clase	Idea
Parejas (emparejamiento perfecto)	P	reducción a Flujo Máximo (Ford-Fulkerson)
Red Feliz	TFNP (PLS)	mejora local: voltear nodos infelices hasta converger
Isomorfismo de grafos	GI	en NP, no demostrado NP-completo ni P
PRIMO(n)	NP (de hecho P)	certificado de Pratt
ACTRI (tripletas)	NP-completo	versión tridimensional de Parejas
Mochila (optimización)	NP-difícil, FPTAS	programación dinámica `O(n·B)`

Capítulo 4

Relaciones de problemas

Ejercicios resueltos de las relaciones de la asignatura. Cada apartado enuncia el problema y desarrolla la demostración completa.

4.1 Relación 1

4.1.1 Ejercicio 1

(a) Palabras sobre el alfabeto {0,1} con el mismo número de ceros que de unos **Base teórica:** Este es un lenguaje incontextual (libre de contexto), pero que una MT puede decidir fácilmente utilizando una estrategia de emparejamiento cruzado ("zig-zag"), demostrando que la MT no está limitada por el orden de aparición de los símbolos gracias a su cinta de acceso bidireccional.

Diseño detallado (Algoritmo sin ambigüedad): 1. **Estado inicial (q0):** El cabezal lee el primer símbolo no marcado. * Si es **0**, lo marca con **X**, se mueve a la derecha (R) y pasa a un estado **q1** (buscando un 1). * Si es **1**, lo marca con **Y**, se mueve a la derecha (R) y pasa a un estado **q2** (buscando un 0). * Si es un blanco # o si todos los símbolos restantes son marcas (**X** o **Y**), pasa al estado de aceptación **q_acc** y se detiene. 2. **Estado de búsqueda (q1 - busca un 1):** * Lee **0**, **X** o **Y**: avanza a la derecha (R). * Lee **1**: lo marca con **Y**, cambia de dirección a la izquierda (L) y pasa al estado **q_rewind**. 3. **Estado de búsqueda (q2 - busca un 0):** * Lee **1**, **X** o **Y**: avanza a la derecha (R). * Lee **0**: lo marca con **X**, cambia de dirección a la izquierda (L) y pasa al estado **q_rewind**. 4. **Estado de rebobinado (q_rewind):** * Se mueve hacia la izquierda (L) saltando cualquier símbolo (**0**, **1**, **X**, **Y**) hasta encontrar un blanco #. * Al encontrar el blanco #, da un paso a la derecha (R) y vuelve al estado **q0** para iniciar el siguiente ciclo. 5. **Rechazo implícito:** Si en **q1** o **q2** se encuentra un blanco # antes de hallar la pareja correspondiente, la máquina no tiene transición definida y se detiene rechazando la palabra.

(b) $L = \{a^n b^n c^n \mid n \geq 1\}$ **Base teórica:** Este es el ejemplo canónico de un lenguaje sensible al contexto (no incontextual). Las gramáticas libres de contexto no pueden generar este lenguaje (demostrable vía el Lema del Bombeo), pero una MT lo decide emparejando tríos de símbolos.

Diseño detallado (Algoritmo sin ambigüedad): 1. **Estado inicial (q0):** * Lee **a**, la sustituye por **X**, se mueve a la derecha y pasa a **q1**. * Si lee **Y** (lo que indicaría que ya no quedan **a**'s), pasa a un estado de verificación **q_verify**. 2. **Estado q1 (buscar la b):** * Salta las **a** y las **Y** moviéndose a la derecha. * Al leer la primera **b**, la sustituye por **Y**, se mueve a la derecha y pasa a **q2**. 3. **Estado q2 (buscar la c):** * Salta las **b** y las **Z** moviéndose a la derecha. * Al leer la primera **c**, la sustituye por **Z**, se mueve a la izquierda y pasa al estado de rebobinado **q3**. 4. **Estado q3 (rebobinar):** * Se mueve a la izquierda saltando **a**, **b**, **Y**, **Z** hasta encontrar la última **X** escrita. * Da un paso a la derecha y vuelve a **q0**. 5. **Estado q_verify (comprobar fin de cadena):** * Se

mueve a la derecha saltando únicamente marcas **Y** y **Z**. * Si encuentra un blanco #, la cadena es válida y transita al estado de aceptación **q_acc**. Si encuentra alguna **a**, **b** o **c** suelta, rechaza.

(c) $\{ww^{-1} \mid w \in \{0,1\}^*\}$ **Base teórica:** Este lenguaje describe palíndromos de longitud par. Aquí la notación formal

$$w^{-1}$$

representa la palabra invertida de la cadena binaria

$$w$$

. Utilizaremos el no determinismo implícito o un diseño determinista que recuerda símbolos utilizando el propio estado finito de la máquina.

Diseño detallado (Algoritmo determinista sin ambigüedad): 1. **Memorizar y marcar (q0):** * Si lee **0**, lo marca con **X**, avanza a la derecha y entra en **q_find_0** (recordando que debe emparejar un 0). * Si lee **1**, lo marca con **X**, avanza a la derecha y entra en **q_find_1** (recordando que debe emparejar un 1). * Si lee **X** o un blanco #, significa que la cadena se ha analizado completamente con éxito y pasa a **q_acc**. 2. **Ir al final no procesado (q_find_0 / q_find_1):** * Salta los **0** y **1** hacia la derecha hasta encontrar un blanco # o una marca **X**. * Retrocede un paso a la izquierda y entra en un estado de verificación respectivo **q_check_0** o **q_check_1**. 3. **Verificar el extremo (q_check_0 / q_check_1):** * En **q_check_0**: Si el símbolo es **0**, lo marca con **X**, se mueve a la izquierda y pasa a **q_rewind**. Si no es **0**, rechaza. * En **q_check_1**: Si el símbolo es **1**, lo marca con **X**, se mueve a la izquierda y pasa a **q_rewind**. Si no es **1**, rechaza. 4. **Rebobinar (q_rewind):** * Se mueve a la izquierda saltando **0** y **1** hasta encontrar la última marca **X** de la izquierda. * Da un paso a la derecha y vuelve a **q0**.

(d) $\{wcw \mid w \in \{0,1\}^*\}$ **Base teórica:** Este es el lenguaje copia, un lenguaje que no es libre de contexto. El diseño de esta MT ilustra la técnica de "Múltiples pistas" o marcado de símbolos, donde el alfabeto de trabajo es un producto cartesiano. El documento de referencia proporciona el modelo exacto detallado para este caso.

Diseño detallado (extraído de la demostración algorítmica formal): La MT asume un alfabeto de entrada **A = {0, 1, c}**. La idea fundamental radica en recordar el primer símbolo leído mediante el estado y verificar que coincida con el primer símbolo no marcado después del separador **c**. 1. **Marcar y recordar (q1):** Leemos el primer símbolo **a** (donde **a** puede ser 0 o 1), lo recordamos en el estado interno pasando a [**q2, a**], lo marcamos como leído (escribiendo el equivalente a [**a, ***] en una pista virtual) y nos movemos a la derecha. 2. **Buscar el centro (q2):** En el estado [**q2, a**], nos movemos hacia la derecha saltando **0**s y **1**s hasta encontrar el separador **c**. Al encontrarlo, cambiamos al estado [**q3, a**]. 3. **Buscar y comparar destino (q3):** En [**q3, a**], saltamos todo lo que ya esté marcado a la derecha de **c**. Cuando encontremos el primer símbolo no marcado, verificamos que sea exactamente el símbolo **a** que la máquina recordó en el paso 1. Lo marcamos y pasamos al estado de retorno [**q4, #**]. 4. **Retorno (q4 a q6):** Nos movemos hacia la izquierda recorriendo todo hasta llegar a **c** (estado **q5**) y seguimos a la izquierda saltando los símbolos no marcados de la primera mitad hasta chocar con el símbolo que marcamos en el paso 1. Al encontrarlo, damos un paso a la derecha y volvemos a iniciar el ciclo en **q1**. 5. **Aceptación (q7 a q9):** Cuando en la primera fase encontramos directamente la **c**, verificamos que todos los

símbolos a la derecha de la **c** también estén marcados. Si es así, la máquina acepta.

Te animo a trazar estos pasos con cadenas de prueba cortas (por ejemplo, **0011** para el primer problema, o **0c0** para el último) simulando las transiciones estado por estado. Esto consolidará definitivamente tu comprensión de la lógica interna de estos autómatas.

Errores comunes en el apartado b

1. Fundamento Teórico: Ampliación del Alfabeto (Marcado de Símbolos) En la teoría de autómatas, cuando necesitamos contabilizar o "tachar" un símbolo sin perder el espacio físico que ocupa en la cinta (para no destruir la contigüidad de los datos), recurrimos a una técnica conocida en textos clásicos (como Hopcroft y Ullman) como **ampliación del alfabeto de cinta** o el uso de **pistas múltiples**.

En lugar de borrar el símbolo con un blanco general (`-` o `#`), lo reemplazamos por una versión "marcada" de sí mismo. Por ejemplo, si en tu Fase 1 conviertes el alfabeto original en X, Y, Z, en tu Fase 2 puedes "tachar" usando letras minúsculas: x, y, z.

De este modo, el cabezal puede "caminar" sobre los símbolos tachados (x, y, z) sabiendo que ya fueron procesados, pero sin confundirlos jamás con el límite exterior de la cinta (`-`).

2. (Algunos errores) La Solución Detallada (Corrección de Fase 2) Vamos a reescribir la Fase 2 de tu algoritmo utilizando x, y, z como símbolos de tachado. Fíjate bien en cómo el estado de rebobinado (`q-rewind`) ahora funciona perfectamente.

- **Paso A: Iniciar el tachado en `q-verify`** El cabezal está sobre la primera X disponible. La se tacharon (escribiendo x) y vamos a buscar la Y. `q-verify X x r qeliminaY`
- **Paso B: Buscar y tachar la Y en `qeliminaY`** Hay que saltar las X aún sin procesar y las y ya tachadas en ciclos anteriores. `qeliminaY X X r qeliminaY` (salta las X restantes) `qeliminaY y y r qeliminaY` (salta las Y ya tachadas) `qeliminaY Y y r qeliminaZ` (encuentra la primera Y válida, la tacha con y y avanza)
- **Paso C: Buscar y tachar la Z en `qeliminaZ`** Igual que antes, saltamos lo necesario. **Nota crucial:** Una vez que tachas la Z, el movimiento inmediato debe ser a la izquierda (`l`), ¡no a la derecha!, para empezar a volver. `qeliminaZ Y Y r qeliminaZ` (salta las Y restantes) `qeliminaZ z z r qeliminaZ` (salta las Z ya tachadas) `qeliminaZ Z z l q-rewind` (encuentra la Z, la tacha con z y **gira a la izquierda**)
- **Paso D: El Rebobinado seguro en `q-rewind`** Ahora retrocedemos saltando todo (marcados y no marcados) hasta chocar con el símbolo x que acabamos de marcar en el Paso A. `q-rewind i i l q-rewind` (con $i \in \{X, Y, Z, y, z\}$) `q-rewind x x r q-verify` (Al chocar con la x, damos un paso a la derecha. ¡El cabezal queda exactamente sobre la siguiente X lista para el próximo ciclo!)

3. La Condición de Parada (Aceptación) Con este diseño, ¿qué ocurre cuando ya no quedan más X que tachar? El cabezal, estando en `q-verify`, leerá una y (la primera Y que se tachó). Esto indica que se han agotado las X. Para que la MT acepte formalmente, hay que verificar que **solo** quedan símbolos tachados (y, z) hasta llegar al blanco (`-`), lo que demuestra que no sobraron letras.

Añadimos estas transiciones finales: `q-verify y y r q-check-end` `q-check-end y y r q-check-end` (salta todas las y) `q-check-end z z r q-check-end` (salta todas las z) `q-check-end - - * halt-accept` (Si llega al final y todo estaba tachado, la palabra es perfecta).

4.1.2 Ejercicio 2

1. **Almacenamiento de símbolo:** El conjunto de estados pasa a ser $Q' \times B$, permitiendo que la unidad de control "recuerde" un símbolo (el estado se denota como $[q, b]$).
2. **Pistas Múltiples:** El alfabeto de trabajo de la cinta se define como B^k. Esto nos permite tener los datos intactos en una pista y utilizar otra pista exclusivamente para colocar marcas de control (ej. un símbolo sería $[a, *]$ o $[a, \#]$).
3. **Subrutinas:** Definimos un conjunto de estados con un punto de entrada y uno de retorno para realizar tareas mecánicas repetitivas (como rebobinar o desplazar bloques) sin tener que reescribir la lógica.

La solución sería:

(a) Palabras sobre el alfabeto {0,1} con el mismo número de ceros que de unos **Técnica aplicada:** Pistas múltiples + Almacenamiento de símbolo.

En el diseño original, "ensuciábamos" la cinta sobrescribiendo con X e Y. Ahora, utilizaremos una cinta de dos pistas. La pista superior tendrá los datos y la inferior las marcas (usaremos $*$ para "emparejado" y $\#$ para "no procesado"). El alfabeto será $B = \{0, 1, \#\} \times \{\#, *\}$. Además, almacenaremos en el estado el símbolo que buscamos.

Lógica del programa: 1. **Inicio y Memorización:** La máquina lee el primer símbolo no marcado, por ejemplo, $[0, \#]$. Lo marca en la segunda pista escribiendo $[0, *]$ y "guarda" en su estado finito el símbolo complementario que debe buscar (el 1) transitando al estado $[q_{buscar}, 1]$. 2. **Búsqueda (Subrutina):** En el estado $[q_{buscar}, 1]$, el cabezal avanza a la derecha ignorando cualquier símbolo que ya esté marcado en la segunda pista $[*, *]$ y saltando también los de la misma clase no marcados $[0, \#]$. 3. **Emparejamiento:** Al encontrar el símbolo complementario $[1, \#]$, lo marca escribiendo $[1, *]$. 4. **Rebobinado (Subrutina):** Llama a una subrutina estándar que retrocede a la izquierda hasta encontrar el límite o el último símbolo procesado, y reinicia el ciclo principal.

(b) $L = \{a^n b^n c^n \mid n \geq 1\}$ **Técnica aplicada:** Pistas múltiples (Multipista) y Subrutinas.

Al igual que en el caso anterior, evitaremos borrar las letras originales para mantener la integridad de los datos, usando el alfabeto $B = \{a, b, c, \#\} \times \{\#, *\}$.

Lógica del programa: 1. **Marcar A:** Leemos $[a, \#]$, escribimos $[a, *]$ y pasamos a la subrutina `BUSCAR_B`. 2. **Subrutina `BUSCAR_B`:** El cabezal avanza a la derecha. Caben tres lecturas: $[a, \#]$ (otras a pendientes), $[a, *]$ (a ya procesadas) o $[b, *]$ (b ya procesadas de ciclos anteriores). Saltamos todas ellas. Al leer la primera $[b, \#]$, la marcamos como $[b, *]$ y pasamos a `BUSCAR_C`. 3. **Subrutina `BUSCAR_C`:** Saltamos las $[b, \#]$ restantes y las $[c, *]$ ya procesadas. Al leer la primera $[c, \#]$, la marcamos como $[c, *]$ y pasamos a la subrutina `REBOBINAR`. 4. **Subrutina `REBOBINAR`:** Nos movemos hacia la izquierda incondicionalmente hasta encontrar la última $[a, *]$ que marcamos. Damos un paso a la derecha y reiniciamos el ciclo.

(c) $\{ww^{-1} \mid w \in \{0, 1\}^*\}$ **(Palíndromos pares)** **Técnica aplicada:** Almacenamiento de símbolo en el estado.

En tu diseño clásico, necesitabas bifurcar tu programa completamente: una rama entera de estados

para procesar si leías un 0, y otra rama entera si leías un 1. La técnica de almacenamiento de símbolo compacta esto drásticamente.

Lógica del programa: 1. Definimos los estados como $[q, \sigma]$ donde $\sigma \in \{0, 1, \#\}$. 2. **Leer y Recordar:** En el estado inicial $[q_0, \#]$, leemos un símbolo $a \in \{0, 1\}$, lo sobrescribimos con un blanco # (o lo marcamos), y lo "cargamos" en el estado pasando a $[q_{buscar\_fin}, a]$. 3. **Avanzar:** La transición genérica $\delta([q_{buscar\_fin}, a], b) = ([q_{buscar\_fin}, a], b, D)$ (para cualquier b) nos lleva al final de la palabra. Al chocar con el blanco, retrocedemos un paso a la izquierda pasando a $[q_{verificar}, a]$. 4. **Verificación Mágica:** Aquí ocurre la elegancia técnica. Definimos la transición $\delta([q_{verificar}, a], a) = ([q_{rebobinar}, \#], \#, I)$. Esta única regla obliga matemáticamente a que el símbolo en la cinta coincida con el a almacenado en el estado. Si no coincide, la MT no tiene transición y rechaza inmediatamente.

(d) $\{wcw \mid w \in \{0,1\}^*\}$ **Técnica aplicada:** Almacenamiento de símbolo + Pistas Múltiples.

Este es el ejemplo canónico que se resuelve de forma oficial en el material de clase (Tema 1, diapositivas 53 a 58). Utiliza el alfabeto $B = \{0, 1, c, \#\} \times \{\#, *\}$ y estados formados por la tupla $Q' \times \{0, 1, \#\}$.

Lógica del programa (extraída del modelo formal): 1. **Fase de memorización:** Leemos el primer símbolo no marcado a (que será 0 o 1), lo registramos en la unidad de control cambiando al estado $[q_2, a]$, lo marcamos en la cinta escribiendo $[a, *]$ y nos movemos a la derecha. 2. **Transición al segundo bloque:** En el estado $[q_2, a]$, avanzamos hacia la derecha hasta que detectamos el separador central `c` (representado como $[c, \#]$), momento en el que transitamos a $[q_3, a]$. 3. **Comprobación de identidad:** En $[q_3, a]$, saltamos todos los símbolos que ya tengan la marca `*` en la segunda pista. Cuando encontramos el primer símbolo no marcado, exigimos mediante la función de transición que coincida exactamente con el símbolo a almacenado en el estado. Si coincide, se marca escribiendo $[a, *]$ y pasamos al estado de retorno $[q_4, \#]$. 4. **Retorno guiado:** Rebobinamos hacia la izquierda cruzando la `c` (pasando a q_5) y seguimos hasta encontrar un símbolo marcado $[a, *]$ del primer bloque. Damos un paso a la derecha hacia el nuevo símbolo no marcado, cambiamos al estado inicial $[q_1, \#]$ y repetimos.

4.1.3 Ejercicio 3

Para este problema, el requerimiento nos impone una restricción muy estricta: el alfabeto de trabajo es únicamente $B = \{0, 1, \#\}$. Esto significa que **no cabe usar pistas múltiples ni símbolos especiales de marcado** (como X o Y) para recordar dónde empezamos.

1. Fundamentos Teóricos: "Acarreo" y Almacenamiento de Símbolo Para solucionar este desafío, hay que aplicar la técnica de **almacenamiento de símbolo en el estado** (recordando el símbolo leído en la unidad de control). La estrategia algorítmica es un "acarreo" (shift) en cadena: 1. Leemos el símbolo actual, lo borramos (escribiendo un blanco #) y se almacena en el estado. 2. Nos movemos a la derecha. Leemos el siguiente símbolo, escribimos el que traíamos guardado, y guardamos el nuevo. 3. Repetimos hasta encontrar el primer blanco #. Allí depositamos el último símbolo que traíamos. 4. Retrocedemos a la izquierda hasta chocar con el blanco # dejado en el paso 1. Esa es la posición original.

2. Diseño Detallado de las Transiciones Asumiremos que los movimientos posibles son **D** (Derecha) e **I** (Izquierda). Los estados de la subrutina son:

- **Estado inicial de la subrutina:** q_{inicio}
- **Estados de acarreo:** q_{lleva_0} y q_{lleva_1}
- **Estado de rebobinado:** $q_{retorno}$
- **Estado final de la subrutina (punto de salida):** q_{fin}

Fase 1: Extraer el primer símbolo y dejar el hueco Si leemos un 0 o un 1, lo guardamos pasando al estado correspondiente, se deja un # que marca el punto de retorno y se avanza a la derecha. Si leemos #, no hay nada que desplazar y terminamos directamente. * $\delta(q_{inicio}, 0) = (q_{lleva\_0}, \#, D)$ * $\delta(q_{inicio}, 1) = (q_{lleva\_1}, \#, D)$ * $\delta(q_{inicio}, \#) = (q_{fin}, \#, S)$ *(Nota: Si la MT permite el movimiento estático **S**, nos quedamos ahí; si no, la subrutina simplemente asume que ya está en la posición correcta).*

Fase 2: El desplazamiento en cadena (Acarreo) En estos estados, soltamos el símbolo transportado, se recoge el de la cinta para el estado siguiente y se avanza a la derecha. * $\delta(q_{lleva\_0}, 0) = (q_{lleva\_0}, 0, D)$ *(Traigo un 0, leo un 0 → dejo un 0, me llevo un 0)* * $\delta(q_{lleva\_0}, 1) = (q_{lleva\_1}, 0, D)$ *(Traigo un 0, leo un 1 → dejo el 0, me llevo un 1)* * $\delta(q_{lleva\_1}, 0) = (q_{lleva\_0}, 1, D)$ *(Traigo un 1, leo un 0 → dejo el 1, me llevo un 0)* * $\delta(q_{lleva\_1}, 1) = (q_{lleva\_1}, 1, D)$ *(Traigo un 1, leo un 1 → dejo el 1, me llevo un 1)*

Fase 3: Fin del bloque y depositar el último símbolo Cuando el acarreo encuentra el primer blanco #, significa que el bloque ha terminado. Escribimos el último símbolo que traíamos y comenzamos a retroceder a la izquierda. * $\delta(q_{lleva\_0}, \#) = (q_{retorno}, 0, I)$ * $\delta(q_{lleva\_1}, \#) = (q_{retorno}, 1, I)$

Fase 4: Rebobinar hasta la posición de inicio Ahora nos movemos hacia la izquierda saltando los 0s y 1s que acabamos de desplazar. Cuando topemos con el blanco # (que es exactamente el que creamos en la Fase 1), nos detenemos. * $\delta(q_{retorno}, 0) = (q_{retorno}, 0, I)$ * $\delta(q_{retorno}, 1) = (q_{retorno}, 1, I)$ * $\delta(q_{retorno}, \#) = (q_{fin}, \#, S)$ *(O si no se permite movimiento **S**, pasamos a q_{fin} y damos el control a la siguiente instrucción del programa principal).*

4.2 Relación 2

Problema 10 Dado el siguiente programa con variables:

```
IF X ENDS 0 GOTO A
IF X ENDS 1 GOTO B
HALT
```

[A]
```
X <- X-
Y <- 0Y
IF X ENDS 0 GOTO A
```

[B]
```
IF X ENDS 1 GOTO B
HALT
X <- X-
Y <- 1Y
IF X ENDS 0 GOTO A
IF X ENDS 1 GOTO B
HALT
```

Construir un programa Post-Turing equivalente (se pueden usar macros).

4.3 Relación 4

4.3.1 Ejercicio 1

(a) Todo grafo finito dirigido acíclico (DAG) tiene una fuente.

Prueba: Supóngase, por reducción al absurdo, que el grafo no tiene ninguna fuente. Por definición, una fuente es un nodo con grado de entrada igual a 0. Si no existe ninguna fuente, significa que **todos los nodos del grafo tienen al menos un arco entrante**.

Se parte de un nodo cualquiera v_1. Como v_1 tiene al menos un arco entrante, debe existir un nodo v_2 tal que $(v_2, v_1) \in E$. Del mismo modo, como v_2 no es una fuente, existe un v_3 tal que $(v_3, v_2) \in E$. El proceso se prolonga de forma indefinida construyendo un camino hacia atrás:

$$\cdots \longrightarrow v_3 \longrightarrow v_2 \longrightarrow v_1$$

Como el grafo es **finito** (tiene un número n finito de nodos), por el **principio del palomar** (*pigeonhole principle*), eventualmente habrá que repetir un nodo en la secuencia de longitud mayor a n. Es decir, existen índices $i < j$ tales que $v_i = v_j$. Esto demuestra la existencia de un ciclo dirigido $(v_j \to v_{j-1} \to \cdots \to v_i)$, lo cual contradice directamente la hipótesis de que el grafo es acíclico. Por lo tanto, el grafo debe tener al menos una fuente.

(b) Caracterización de numeración topológica.

Prueba:

- **Implicación directa ($\Rightarrow$):** Supóngase que el grafo es acíclico. Por el apartado (a), se sabe que tiene al menos una fuente. Le asignamos a esta fuente el número 1 y la eliminamos del grafo junto con todos sus arcos salientes. El grafo restante sigue siendo finito y acíclico, por lo que tendrá otra fuente en el subgrafo resultante, a la cual le asignamos el número 2. Repitiendo este proceso de forma inductiva, numeramos los nodos del 1 al n. Como en cada paso eliminamos un nodo que no tiene arcos entrantes en el grafo restante, cualquier arco del grafo original irá necesariamente de un nodo eliminado antes (número menor) a uno eliminado después (número mayor).
- **Implicación inversa ($\Leftarrow$):** Supóngase que existe dicha numeración. Se trata de probar que el grafo no tiene ciclos. Supóngase, por reducción al absurdo, que existe un ciclo dirigido de la forma:

$$v_{i_1} \longrightarrow v_{i_2} \longrightarrow \cdots \longrightarrow v_{i_k} \longrightarrow v_{i_1}$$

Por la propiedad de la numeración, se debe cumplir simultáneamente que:

$$\text{num}(v_{i_1}) < \text{num}(v_{i_2}) < \cdots < \text{num}(v_{i_k}) < \text{num}(v_{i_1})$$

Lo cual implica que $\text{num}(v_{i_1}) < \text{num}(v_{i_1})$, una contradicción matemática estricta. Por tanto, el grafo no contiene ciclos.

(c) Algoritmo en tiempo polinómico (Algoritmo de Kahn).

El algoritmo se basa en la eliminación iterativa de fuentes descrita en el apartado anterior:

1. Calcular el grado de entrada (*in-degree*) de todos los nodos del grafo.
2. Introducir en una cola todos los nodos cuyo grado de entrada sea 0.
3. Mientras la cola no esté vacía:

– Extraer un nodo u de la cola.
– Para cada vecino v tal que exista el arco dirigido (u, v), decrementar el grado de entrada de v en 1.
– Si el grado de entrada de v se reduce a 0, añadir v a la cola.

4. Si al finalizar el bucle se han extraído los n nodos, el grafo es **acíclico**. Si quedan nodos sin procesar, el grafo contiene **al menos un ciclo**.

Complejidad: Calcular el grado de entrada inicial cuesta $O(|V| + |E|)$. Cada nodo y cada arco se procesa exactamente una vez en la cola, por lo que la complejidad total es $O(|V| + |E|)$, que es de tiempo polinómico (lineal).

4.3.2 Ejercicio 2

(a) Un grafo es bipartito si y solo si todos sus ciclos son de longitud par.

Prueba:

– **Implicación directa ($\Rightarrow$):** Sea $G = (V, E)$ un grafo bipartito con partición $V = V_1 \cup V_2$. Cualquier arista conecta un nodo de V_1 con uno de V_2. Por tanto, cualquier camino en el grafo debe alternar estrictamente entre ambos conjuntos. Para empezar en un nodo de V_1, recorrer un camino cerrado (ciclo) y regresar al mismo nodo de V_1, el camino debe realizar obligatoriamente un número par de transiciones (ir a V_2 y volver). Así, la longitud de todo ciclo es par.
– **Implicación inversa ($\Leftarrow$):** Supóngase que todos los ciclos tienen longitud par. Asumiendo que el grafo es conexo (si no lo es, se aplica a cada componente), elegimos un nodo origen v_0. Definimos la partición de vértices como:

$$V_1 = \{u \in V : \text{dist}(v_0, u) \text{ es par}\}$$

$$V_2 = \{u \in V : \text{dist}(v_0, u) \text{ es impar}\}$$

Supóngase, por reducción al absurdo, que existen dos nodos $x, y \in V_1$ conectados por una arista $(x, y) \in E$. Como ambos están en V_1, la longitud de los caminos mínimos desde v_0 a x y a y tienen la misma paridad. El circuito cerrado formado por el camino mínimo $v_0 \rightsquigarrow x$, la arista (x, y) y el camino mínimo inverso $y \rightsquigarrow v_0$ tendría una longitud total de $\text{dist}(v_0, x) + 1 + \text{dist}(v_0, y)$, lo cual es un número impar (par + 1 + par = impar). Todo circuito impar contiene necesariamente un ciclo de longitud impar, contradiciendo la hipótesis. Por tanto, no existen aristas internas y el grafo es bipartito.

(b) Algoritmo en tiempo polinómico.

Se utiliza un recorrido en anchura (BFS) para intentar colorear el grafo con 2 colores:

1. Inicializar todos los nodos como *no visitados*.
2. Para cada nodo no visitado, asignarle el "Color 1" y meterlo en una cola BFS.
3. Mientras la cola no esté vacía:

– Extraer el nodo u.
– Para cada vecino v de u:
– Si v no ha sido visitado, asignarle el color opuesto a u y meterlo en la cola.
– Si v ya está visitado y tiene el mismo color que u, detener el algoritmo y devolver **FALSO** (se detectó un ciclo impar).

4. Si el recorrido termina sin conflictos, devolver **VERDADERO**.

Complejidad: Al ser una modificación directa de BFS, su complejidad temporal es $O(|V| + |E|)$, que es polinómica.

4.3.3 Ejercicio 3

Demostrar que P es cerrada para la unión y la intersección.

Sean $L_1, L_2 \in$ P. Por definición, existen dos Máquinas de Turing Deterministas (MTD) M_1 y M_2 que deciden L_1 y L_2 en tiempos acotados por los polinomios $p_1(n)$ y $p_2(n)$ respectivamente.

– **Clausura para la Unión ($L_1 \cup L_2$):** Se construye una MTD $M_\cup$ que recibe una entrada x de longitud n. $M_\cup$ simula en primer lugar la ejecución de $M_1(x)$. Si M_1 acepta, $M_\cup$ acepta inmediatamente. Si M_1 rechaza, $M_\cup$ pasa a simular $M_2(x)$, aceptando si esta acepta y rechazando en caso contrario. El tiempo total de ejecución está acotado por $p_1(n) + p_2(n)$, que sigue siendo una función polinómica.
– **Clausura para la Intersección ($L_1 \cap L_2$):** Se construye de igual forma una MTD $M_\cap$. Ante una entrada x, ejecuta secuencialmente $M_1(x)$. Si M_1 rechaza, $M_\cap$ rechaza inmediatamente. Si M_1 acepta, ejecuta $M_2(x)$, devolviendo el mismo resultado que M_2. El tiempo máximo vuelve a ser $p_1(n) + p_2(n)$, un orden polinómico.

4.3.4 Ejercicio 4

Siguiendo la nota del documento, evaluamos el comportamiento sustituyendo polinomios genéricos por $p(n) = n^l$ $(l > 0)$.

(a) $\{n^k : k > 0\}$

– **Izquierda:** $p(f(n)) = (n^k)^l = n^{k \cdot l}$. Como $k \cdot l > 0$, la función pertenece a la clase. **Sí es cerrada**.
– **Derecha:** $f(p(n)) = (n^l)^k = n^{l \cdot k}$. Mismo caso. **Sí es cerrada**.

(b) $\{n \cdot k : k \geq 0\}$

- **Izquierda:** $p(f(n)) = (n \cdot k)^l = n^l \cdot k^l$. Si $l > 1$, la función resultante crece como $O(n^l)$, que supera la escala lineal de la clase. **No es cerrada**.
- **Derecha:** $f(p(n)) = n^l \cdot k$. Para cualquier $l > 1$, no pertenece a la clase. **No es cerrada**.

(c) $\{k^n : k > 0\}$

- **Izquierda:** $p(f(n)) = (k^n)^l = k^{l \cdot n} = (k^l)^n$. Como k^l es una constante, la función resultante mantiene la forma exponencial de la clase. **Sí es cerrada**.
- **Derecha:** $f(p(n)) = k^{(n^l)}$. Para $l > 1$, la función de orden k^{n^2} o superior crece mucho más rápido que cualquier simple exponencial m^n. **No es cerrada**.

(d) $\{2^{n^k} : k > 0\}$

- **Izquierda:** $p(f(n)) = (2^{n^k})^l = 2^{l \cdot n^k}$. Al aplicar la notación O, se cumple que $2^{l \cdot n^k} \leq 2^{n^{k+1}}$ para n suficientemente grande, cuya función base está en la clase $(k + 1 > 0)$. **Sí es cerrada**.
- **Derecha:** $f(p(n)) = 2^{(n^l)^k} = 2^{n^{l \cdot k}}$. Como $l \cdot k > 0$, la función pertenece a la clase. **Sí es cerrada**.

(e) $\{\log^k(n) : k > 0\}$

- **Izquierda:** $p(f(n)) = (\log^k n)^l = \log^{k \cdot l} n \in C$. **Sí es cerrada**.
- **Derecha:** $f(p(n)) = \log^k(n^l) = (l \cdot \log n)^k = l^k \cdot \log^k n = O(\log^k n) \in C$. **Sí es cerrada**.

(f) $\{\log(\log n)\}$

- **Izquierda:** $p(f(n)) = (\log(\log n))^l \notin C$ para $l > 1$. **No es cerrada**.
- **Derecha:** $f(p(n)) = \log\big(\log\big(n^l\big)\big) = \log(l \cdot \log n) = \log l + \log(\log n) = O(\log(\log n)) \in C$. **Sí es cerrada**.

4.3.5 Ejercicio 5

Demostrar que $L = \{wcw : w \in \{0,1\}^*\}$ ***está en espacio*** $\log(n)$**.**

Para diseñar un algoritmo en espacio logarítmico, contamos con una cinta de entrada de *solo lectura* y cintas de trabajo con espacio limitado a $O(\log n)$. No cabe almacenar la palabra w en la memoria de trabajo porque ocuparía espacio lineal.

Algoritmo:

1. Recorrer la cinta de entrada con un contador en binario para localizar la posición exacta del carácter especial 'c'. Almacenamos este índice m en la cinta de trabajo (ocupa $\approx \log_2 n$ bits).
2. Verificar que la longitud total del string sea exactamente $2m + 1$. Si no es así, rechazar.
3. Utilizar un segundo contador en binario, i, inicializado en 0.
4. Mientras $i < m$:

- Mover el cabezal de la cinta de entrada a la posición i y leer el símbolo $w[i]$.

- Mover el cabezal de entrada a la posición $m + 1 + i$ y leer el símbolo correspondiente.
- Comparar ambos símbolos. Si difieren, **rechazar**.
- Incrementar el contador i en 1.

5. Si el bucle finaliza con éxito, **aceptar**.

Análisis de espacio: Únicamente se han utilizado dos contadores enteros binarios (m e i) cuyos valores máximos están acotados por el tamaño de la entrada n. Por tanto, el espacio de trabajo utilizado es estrictamente $O(\log n)$.

4.3.6 Ejercicio 6

Demostrar que si $L \in$ ***P, entonces*** $L^* \in$ ***P.***

Resolvemos este problema mediante un enfoque de **programación dinámica**. Sea x una palabra de entrada de longitud n. Se trata de determinar si x se puede descomponer en $x = w_1 w_2 \dots w_k$ tal que cada subpalabra $w_i \in L$.

Definimos un vector booleano T de tamaño $n+1$, donde $T[i] =$ VERDADERO si y solo si el prefijo de la palabra de longitud i ($x[1 \dots i]$) pertenece a L^*.

- **Caso base:** $T[0] =$ VERDADERO (la palabra vacía ϵ siempre pertenece a L^*).
- **Paso inductivo:** Para cada i desde 1 hasta n:

$$T[i] = \text{VERDADERO} \iff \exists j \in \{0, \dots, i-1\} \text{ tal que } (T[j] == \text{VERDADERO} \wedge x[j+1 \dots i] \in L)$$

Como $L \in$ P, la verificación de si la subpalabra de entrada $x[j+1 \dots i]$ pertenece a L tarda un tiempo polinómico $p(n)$. El algoritmo realiza como máximo $O(n^2)$ subconsultas. La complejidad temporal total estará acotada por $O(n^2 \cdot p(n))$, que es polinómica, por tanto $L^* \in$ P.

4.3.7 Ejercicio 7

Demostrar que si $L \in$ ***NP, entonces*** $L^* \in$ ***NP.***

Si $L \in$ NP, existe una Máquina de Turing No Determinista (MTND) M que decide el lenguaje L en tiempo polinómico. Diseñamos una nueva MTND M^* para decidir L^*:

Ante una palabra de entrada x de longitud n:

1. **Adivinar de forma no determinista** una partición de la cadena x en k subpalabras ($1 \leq k \leq n$). Esto equivale a seleccionar un conjunto de índices de corte intermedios.
2. Para cada una de las subpalabras de la partición $(w_1, w_2, \dots, w_k)$, ejecutar de forma secuencial (o en paralelo no determinista) la MTND original M.
3. Si todas las simulaciones de las subpalabras entran en estado de aceptación, entonces M^* **acepta**. En caso de que alguna subpalabra sea rechazada, esa rama del cómputo no determinista **rechaza**.

Complejidad: La fase de generación no determinista de los cortes toma tiempo lineal $O(n)$, y la verificación de las subpalabras toma a lo sumo $n \cdot p(n)$ pasos. Al ser la composición de procesos polinómicos, M^* opera en tiempo polinómico no determinista, luego $L^* \in$ NP.

4.3.8 Ejercicio 8

Demostrar que NP es cerrada para la unión y la intersección.

Sean $L_1, L_2 \in$ NP decididos por las MTND M_1 y M_2 en tiempo polinómico acotado por $p_1(n)$ y $p_2(n)$ respectivamente.

- **Unión ($L_1 \cup L_2$):** Diseñamos una MTND $M_\cup$ que, ante una entrada x, realiza en su primer paso una ramificación no determinista con dos opciones. La primera opción ejecuta $M_1(x)$ y la segunda ejecuta $M_2(x)$. Si al menos una de las ramas alcanza un estado de aceptación, la máquina acepta. El tiempo global del árbol de cómputo es máx$(p_1(n), p_2(n))$, que es polinómico.
- **Intersección ($L_1 \cap L_2$):** Diseñamos una MTND $M_\cap$ que, ante una entrada x, ejecuta secuencialmente la simulación de $M_1(x)$. Si M_1 alcanza una configuración de aceptación, la máquina no se detiene, sino que toma esa configuración y procede a simular de forma no determinista $M_2(x)$. Si esta segunda fase también encuentra una ruta de aceptación, la máquina $M_\cap$ acepta. La profundidad del cómputo está acotada por $p_1(n)+p_2(n)$, manteniendo la condición polinómica.

4.3.9 Ejercicio 9

Demostrar que si NP $\neq$ coNP entonces P $\neq$ NP.

Demostramos esta propiedad utilizando el razonamiento **contrarrecíproco**: probaremos que si P = NP, entonces obligatoriamente NP = coNP.

Por la propia naturaleza de las Máquinas de Turing Deterministas, la clase P es cerrada bajo complementación (P = coP), dado que para calcular el complemento de un lenguaje en P basta con intercambiar los estados de aceptación y rechazo de la MTD que lo resuelve.

Si se asume como premisa que P = NP, cabe aplicar el operador de complemento a ambos lados de la igualdad, obteniendo que coP = coNP. Combinando todas las equivalencias:

$$\text{NP} = \text{P} = \text{coP} = \text{coNP} \implies \text{NP} = \text{coNP}$$

Por tanto, si la comunidad matemática demuestra que NP $\neq$ coNP, se deduce de forma directa y rigurosa que P $\neq$ NP.

4.3.10 Ejercicio 10

Demostrar que determinar si una entrada de paréntesis está correctamente emparejada y anidada está en L.

Para verificar el anidamiento correcto sin recurrir a una estructura de pila en memoria (que requeriría espacio lineal), cabe emplear un único contador entero dinámico.

Algoritmo:

1. Inicializar un contador binario $c = 0$ en la cinta de trabajo.

2. Leer la cadena de entrada de izquierda a derecha, símbolo a símbolo:

– Si el símbolo actual es `(`, incrementar el contador: $c = c + 1$.
– Si el símbolo actual es `)`, decrementar el contador: $c = c - 1$.
– Si en algún punto intermedio del recorrido se cumple que $c < 0$, **rechazar inmediatamente** (indica un exceso de paréntesis de cierre mal balanceados).

3. Al alcanzar el final de la cinta de entrada, comprobar el valor de c:

– Si $c == 0$, **aceptar**.
– Si $c > 0$, **rechazar** (quedaron paréntesis abiertos sin cerrar).

Espacio: El valor máximo que puede alcanzar el contador c es el tamaño de la entrada n. Representar el valor entero n en binario requiere exactamente $\lceil \log_2(n+1) \rceil$ casillas en la cinta de memoria de trabajo, lo que es del orden de $O(\log n)$. Por tanto, el problema pertenece a L.

4.3.11 Ejercicio 11

Caso con dos tipos de paréntesis `()` y `[]`.

– **Demostración de que está en L:** Si disponemos de la capacidad de mover el cabezal de la cinta de entrada en cualquier dirección (hacia adelante y hacia atrás), cabe resolver el problema en espacio logarítmico combinando dos fases:

1. Verificar que los paréntesis `()` y los corchetes `[]` están balanceados de manera independiente numéricamente utilizando dos contadores logarítmicos (tal como se implementó en el Ejercicio 10).
2. Para comprobar que el anidamiento es correcto y no existen cruces ilegales del tipo `([)]`, aplicamos el siguiente análisis geométrico: para cada símbolo de cierre (por ejemplo, un `]` en la posición i), hacemos un recorrido hacia atrás (a la izquierda) buscando su apertura correspondiente. Para ello, se usa un contador que se incrementa con cierres del mismo tipo y se decrementa con aperturas. Cuando el contador vuelve a 0, localizamos su apertura correspondiente en el índice j. En ese momento, validamos que la subcadena interna contenida entre j e i tenga un balance de paréntesis correcto. Como solo almacenamos los índices de las posiciones de los cabezales y pequeños contadores locales de valor inferior a n, el espacio consumido en las cintas es estrictamente $O(\log n)$, lo que prueba que el problema está en L.

– **Demostración de que requiere $O(n)$ de memoria si no se puede volver hacia atrás:** Si se restringe la lectura a un único recorrido secuencial de izquierda a derecha sin retroceso (*algoritmo online*), la máquina se ve obligada a recordar el orden exacto de los tipos de paréntesis abiertos que todavía no han sido cerrados para verificar la validez del cierre inmediato (estructura LIFO). Consideremos una entrada que comienza con una secuencia de longitud $m = n/2$ formada exclusivamente por símbolos de apertura arbitrarios (elección libre entre `(` y `[`). Existen exactamente 2^m posibles combinaciones distintas de apertura. Para que la máquina responda correctamente ante cualquier secuencia posterior de cierre, debe ser capaz de almacenar un estado interno único por cada una de estas combinaciones. Por la teoría de la información, para diferenciar de manera unívoca entre 2^m estados posibles de almacenamiento, la memoria interna de trabajo debe tener una capacidad mínima de:

$$\log_2(2^m) = m = \frac{n}{2} = O(n) \text{ bits}$$

Por consiguiente, sin retroceso de cabezal, el problema requiere espacio lineal $O(n)$.

4.3.12 Ejercicio 12

Demostrar que el problema del palíndromo requiere $O(n)$ ***de memoria si no puede volver hacia atrás.***

Prueba: Supóngase una Máquina de Turing restringida a leer la entrada de izquierda a derecha sin posibilidad de retroceder su cabezal principal. Consideremos el conjunto de todas las cadenas binarias posibles de longitud $m = n/2$ que forman la primera mitad del palíndromo. Existen 2^m cadenas distintas.

Si la máquina dispusiera de menos de m bits de memoria de trabajo, por el principio de las casillas, existirían al menos dos cadenas iniciales distintas, $w_1 \neq w_2$, que llevarían a la máquina exactamente al mismo estado de memoria interna tras leer los primeros m caracteres.

Si la entrada continuase con el reverso exacto de la primera cadena (w_1^R), el string total sería $w_1 w_1^R$, el cual es un palíndromo válido y la máquina debería aceptarlo. Sin embargo, al recibir la entrada $w_2 w_1^R$, como el estado interno de la memoria tras los primeros m pasos es idéntico al caso anterior, la máquina procesaría la segunda mitad de la misma manera y también la **aceptaría**. Esto representa un error algorítmico, ya que $w_2 w_1^R$ no es un palíndromo si $w_1 \neq w_2$.

Para evitar este solapamiento de configuraciones, la máquina debe registrar de forma unívoca cada uno de los 2^m estados informacionales posibles. Por tanto, la memoria de trabajo requiere un número de bits de orden:

$$\log_2(2^m) = m = \frac{n}{2} = O(n)$$

4.3.13 Ejercicio 13

Demostrar que NP $\neq$ ***ESPACIO***(n)***.***

Siguiendo la sugerencia del enunciado, analizamos el comportamiento de ambas clases respecto al operador de reducción en espacio logarítmico (L-reducciones, denotado como $\propto_{\mathrm{L}}$):

1. **La clase NP es cerrada bajo L-reducciones:** Sean dos problemas tales que $P_1 \propto_{\mathrm{L}} P_2$ y $P_2 \in$ NP. Por definición de reducción en espacio logarítmico, existe una MTD que transforma cualquier instancia x de P_1 en una instancia y de P_2 utilizando un espacio de trabajo acotado por $O(\log |x|)$. Dado que el espacio es logarítmico, la longitud de la salida estará acotada polinómicamente respecto a la entrada ($|y| = O(|x|^c)$). Como $P_2 \in$ NP, existe una MTND que decide P_2 en tiempo polinómico respecto a la longitud de su entrada y, lo cual se traduce en un tiempo polinómico respecto a la entrada original x. Componiendo ambos procesos mediante la técnica de "recálculo de bits de salida bajo demanda en espacio logarítmico" (evitando escribir la cadena intermedia y completa en una cinta de trabajo), obtenemos que el problema completo se puede decidir en tiempo no determinista polinómico. Por tanto, $P_1 \in$ NP. **NP es cerrada bajo L-reducciones**.
2. **La clase ESPACIO**(n) **NO es cerrada bajo L-reducciones:** Por el **Teorema de la Jerarquía en Espacio**, consta de forma rigurosa que un incremento en la cota de espacio permite decidir lenguajes estrictamente más complejos; por ejemplo, ESPACIO$(n) \subsetneq$ ESPACIO(n^2). Mediante una reducción en espacio logarítmico, es posible mapear (comprimir)

instancias de un problema de una clase espacial superior en instancias de tamaño lineal de un problema base. Si la clase ESPACIO(n) fuese cerrada bajo este tipo de reducciones, provocaría un colapso en cadena de la jerarquía espacial, permitiendo resolver problemas de orden cuadrático o superior en espacio estrictamente lineal, lo cual contradice el teorema fundamental de la jerarquía.

3. **Conclusión:** Al haber demostrado que la clase de complejidad NP posee la propiedad estructural de clausura ante reducciones en espacio logarítmico y la clase ESPACIO(n) carece de ella, concluimos de forma matemática que ambas clases representan conjuntos de lenguajes diferentes, luego $\mathbf{NP} \neq \mathbf{ESPACIO}(n)$.

4.4 Relación 5

4.4.1 Ejercicio 2: Máquinas de Turing No Deterministas Fuertes y NP ∩ coNP

Se trata de demostrar que un lenguaje L es decidido por una Máquina de Turing No Determinista Fuerte (MTND-F) en tiempo polinómico si y solo si $L \in \text{NP} \cap \text{coNP}$.

Implicación directa ($\Rightarrow$)

Supóngase que L es decidido por una MTND-F polinómica M.

- **Probar que $L \in$ NP:** Por definición de la máquina, si $x \in L$, existe al menos un camino de cómputo que termina en 'Si'. Si $x \notin L$, ningún camino termina en 'Si' (todos terminan en 'No' o 'Duda'). Si ignoramos las salidas 'No' y tratamos 'Duda' como un estado de no-aceptación estándar, M se comporta exactamente como una MTND polinómica clásica que acepta L. Por tanto, $L \in$ NP.
- **Probar que $L \in$ coNP (es decir, $\bar{L} \in$ NP):** Se construye una nueva máquina M' a partir de M modificando únicamente los estados finales: intercambiamos las salidas 'Si' por 'No', y las salidas 'No' por 'Si', manteniendo 'Duda' intacto. Si $x \in \bar{L}$ (es decir, $x \notin L$), la máquina original M garantizaba al menos un camino hacia 'No', el cual ahora será un camino hacia 'Si' en M'. Si $x \in L$, ningún camino en M' llegará a 'Si'. Así, M' es una MTND polinómica que acepta $\bar{L}$, lo que implica que $\bar{L} \in \text{NP} \implies L \in \text{coNP}$.

Al cumplirse ambas condiciones, $L \in \text{NP} \cap \text{coNP}$.

Implicación inversa ($\Leftarrow$)

Supóngase ahora que $L \in \text{NP} \cap \text{coNP}$.

- Como $L \in$ NP, existe una MTND polinómica M_1 que acepta L.
- Como $L \in \text{coNP} \implies \bar{L} \in$ NP, existe otra MTND polinómica M_2 que acepta $\bar{L}$.

Se construye una MTND Fuerte M que, ante una entrada x, comienza con una **bifurcación no determinista** de dos ramas principales:

1. La primera rama simula el cómputo de $M_1(x)$. Si la simulación de M_1 acepta, la máquina M finaliza devolviendo **'Si'**. Si la simulación termina sin aceptar, devuelve **'Duda'**.
2. La segunda rama simula el cómputo de $M_2(x)$. Si la simulación de M_2 acepta (lo que significa que $x \in \bar{L}$), M finaliza devolviendo **'No'**. Si no acepta, devuelve **'Duda'**.

Verificación de las condiciones:

- Si $x \in L$, $M_1(x)$ tiene al menos un camino de aceptación (que dará 'Si') , y $M_2(x)$ no tiene ninguno (todos darán 'Duda'). Todos los caminos terminan en 'Si' o 'Duda' con al menos un 'Si'.
- Si $x \notin L$, $M_1(x)$ no tiene caminos de aceptación (todos darán 'Duda') , y $M_2(x)$ tiene al menos uno (que dará 'No'). Todos los caminos terminan en 'No' o 'Duda' con al menos un 'No'.

El tiempo de ejecución sigue estando acotado polinómicamente. Queda demostrado.

4.4.2 Ejercicio 3: Certificado de Pratt para $p = 13$

El teorema de Pratt establece que un número $p > 1$ es primo si y solo si existe un testigo r $(1 < r < p)$ tal que:

1. $r^{p-1} \equiv 1 \pmod{p}$
2. $r^{\frac{p-1}{q}} \not\equiv 1 \pmod{p}$ para todos los divisores primos q de $p-1$.

Paso 1: Aplicación a $p = 13$

- Los componentes son $p - 1 = 12$. Los divisores primos de 12 son $q_1 = 2$ y $q_2 = 3$.
- Elegimos como testigo $r = 2$.
- **Condición 1:** $2^{12} = 4096$. Calculando el módulo: $4096 \pmod{13} = 1$. Se cumple.
- **Condición 2 (para $q = 2$):** $\frac{12}{2} = 6 \implies 2^6 = 64$. Calculando el módulo: $64 \equiv 12 \not\equiv 1 \pmod{13}$. Se cumple.
- **Condición 2 (para $q = 3$):** $\frac{12}{3} = 4 \implies 2^4 = 16$. Calculando el módulo: $16 \equiv 3 \not\equiv 1 \pmod{13}$. Se cumple.

Paso 2: Certificación recursiva de los divisores primos

Siguiendo la estructura del certificado de Pratt (pág. 33-34 del Tema 4), el divisor 2 es un caso base que no requiere desglose , pero el divisor 3 debe ser certificado a su vez de forma recursiva.

- Para $p = 3$, se tiene $p - 1 = 2$, cuyo único divisor primo es $q = 2$.
- Elegimos el testigo $r = 2$.
- **Condición 1:** $2^{3-1} = 2^2 = 4 \equiv 1 \pmod{3}$. Se cumple.
- **Condición 2:** $2^{\frac{2}{2}} = 2^1 = 2 \not\equiv 1 \pmod{3}$. Se cumple.

Formato Final del Certificado Estructural

Siguiendo la sintaxis formal explicada por el profesor Serafín Moral:

$$\text{Cert}(13) = \Big(\big(13 : 2\ (2,3)\big),\ \big(3 : 2\ (2)\big) \Big)$$

4.4.3 Ejercicio 4: Demostración de que la Exponenciación Modular está en P

Deseamos probar que el problema de determinar si $a^b \equiv c \pmod{p}$ es resoluble en tiempo polinómico respecto a la longitud de la entrada n.

Consideremos que n representa la cantidad total de bits necesarios para almacenar los cuatro números en memoria. El valor del exponente b puede ser de un orden de magnitud de hasta 2^n. Por tanto, un bucle de multiplicaciones sucesivas de tamaño b tardaría un tiempo exponencial $O(2^n)$.

Para resolverlo en tiempo polinómico aplicamos el **Algoritmo de Exponenciación Binaria** (también conocido como *Square-and-Multiply*):

1. Expresar el exponente b en su representación binaria: $b = (b_k b_{k-1} \dots b_0)_2$. El número de bits del exponente está estrictamente acotado por la longitud de la entrada: $k \leq n$.
2. Inicializar una variable acumuladora $R = 1$ y otra variable para las potencias de la base $A = a \pmod{p}$.
3. Para cada bit b_i desde 0 hasta k:

– Si $b_i == 1$, hacer $R = (R \cdot A) \pmod{p}$.
– Hacer $A = (A \cdot A) \pmod{p}$.

4. Finalmente, comprobar si $R == c$.

Análisis de Complejidad Temporal

– El bucle principal se ejecuta exactamente k veces, por lo que realiza un máximo de n **iteraciones**.
– En cada iteración se efectúan multiplicaciones y operaciones de módulo con números cuyo tamaño máximo de almacenamiento está acotado por n bits. El coste de multiplicar y aplicar el módulo a dos enteros de n bits mediante métodos estándar es de orden $O(n^2)$.
– La complejidad total del algoritmo es $O(n \cdot n^2) = \mathbf{O(n^3)}$, lo cual constituye una cota estrictamente polinómica. Por tanto, el problema está en P.

4.4.4 Ejercicio 5: El Problema de la Factorización está en NP ∩ coNP

El problema consiste en determinar si un número x tiene un divisor k en el rango abierto $1 < k < y$.

– **Demostración de que está en NP:** El certificado es simplemente el valor del propio divisor k. El verificador polinómico realiza únicamente dos comprobaciones:

1. Validar que $1 < k < y$.
2. Efectuar la división entera y comprobar que el resto de la operación $x \pmod{k}$ sea exactamente 0. Dado que una división de números de tamaño polinómico se computa de forma eficiente en tiempo $O(n^2)$, el problema se puede verificar de forma polinómica, luego está en NP.

– **Demostración de que está en coNP:** Para probar que pertenece a coNP, es preciso demostrar que el problema complementario ("x **NO** tiene ningún divisor k en el rango $1 < k < y$") se encuentra en la clase NP.
– **Certificado de la instancia NO:** El certificado consiste en proporcionar la **descomposición**

en factores primos completa de x, expresada como $x = p_1^{e_1} p_2^{e_2} \dots p_m^{e_m}$, adjuntando además el correspondiente **certificado de Pratt** para cada uno de los factores primos p_i indicados.

– **Algoritmo Verificador de co-Factorización:**

1. Multiplicar todos los factores de la lista aportada y comprobar que el producto resultante sea exactamente igual a x.
2. Validar la primalidad de cada factor p_i utilizando sus respectivos certificados de Pratt adjuntos (lo cual toma tiempo polinómico).
3. Revisar de forma directa que todos los factores primos de la lista cumplan la condición de ser **mayores o iguales que** y (es decir, $p_i \geq y$). Si todos los divisores primos de un número son $\geq y$, es matemáticamente imposible construir un divisor compuesto que sea menor que y y mayor que 1. Al ser todas estas comprobaciones polinómicas, el problema complementario está en NP $\implies$ el problema original pertenece a coNP.

Como consecuencia directa de ambos apartados, el problema se encuadra en NP $\cap$ coNP.

4.4.5 Ejercicio 6: Correspondencia entre NP ∩ coNP y la Clase TFNP

La clase **TFNP** (*Total Functional NP*) representa el conjunto de los problemas de búsqueda de funciones dentro de FNP para los cuales se garantiza matemáticamente que **siempre existe al menos una solución válida** para cualquier valor de entrada posible.

Dado un lenguaje arbitrario $L \in$ NP$\cap$coNP, por las caracterizaciones de los problemas de verificación se sabe que:

– Si $x \in L$, existe un certificado polinómico c_1 que satisface a un verificador $V_1(x, c_1) = 1$.
– Si $x \notin L$, existe un certificado polinómico c_2 que satisface a un verificador de su complemento $V_2(x, c_2) = 1$.

Por tanto, cada lenguaje en esta intersección sugiere el siguiente problema de búsqueda de funciones en TFNP:

> **Problema sugerido:** "Dada una cadena de entrada x, encontrar un objeto c que cumpla alguna de estas dos condiciones verificables de forma polinómica: o bien c es un certificado válido de que $x \in L$ (haciendo $V_1(x, c) = 1$), o bien c es un certificado válido de que $x \notin L$ (haciendo $V_2(x, c) = 1$)."

¿Por qué pertenece a TFNP?

1. **Es un problema total:** Debido a que por la ley del bando excluido una cadena de entrada necesariamente se cumple que $x \in L$ o bien que $x \notin L$, la existencia de la solución (ya sea el testigo c_1 o el testigo c_2) está **absolutamente garantizada** para todo elemento x. No existen instancias con respuesta vacía (ϵ).
2. **Es verificable en tiempo polinómico:** Un verificador determinista puede tomar la solución c suministrada y evaluar los algoritmos V_1 y V_2 eficientemente para comprobar su validez.

4.4.6 Ejercicio 7: Demostración de que FSAT es FNP-completo

Para demostrar la FNP-completitud de FSAT (*la versión de búsqueda de soluciones de SAT*), es preciso validar los dos pasos de la teoría de funciones de complejidad:

1. Pertenencia a la clase: FSAT $\in$ FNP

El problema recibe como datos un conjunto de cláusulas C. Su relación asociada es $R(C, A) = 1 \iff A$ es una asignación de valores de verdad que satisface la fórmula booleana C. La longitud en bits de la asignación A está estrictamente acotada de forma lineal por el número de variables de C, y comprobar si la asignación satisface las cláusulas toma tiempo polinómico lineal. Por tanto, FSAT pertenece a FNP.

2. Dificultad: Cualquier problema $\Pi \in$ FNP se reduce a FSAT

Sea Π un problema genérico de funciones de la clase FNP, definido por una relación polinómica $R_\Pi(x, y)$. Por la propia definición de la clase, el problema de decisión subyacente ("¿Existe un objeto y tal que $R_\Pi(x, y) = 1$?") pertenece por derecho propio a la clase NP.

Por el **Teorema de Cook-Levin**, se sabe que cualquier problema de decisión en la clase NP se puede reducir en espacio logarítmico al problema de decisión SAT. Existe una transformación polinómica $f(x)$ que genera un conjunto de cláusulas $C = f(x)$ que reproduce fielmente el comportamiento de la Máquina de Turing No Determinista que evalúa la relación.

La reducción estándar de Cook-Levin tiene la propiedad fundamental de ser constructiva respecto a las variables : dentro de las variables proposicionales del diseño de componentes de la reducción, un subconjunto específico de variables de la fórmula —denotémoslas como el grupo de variables de opciones o configuración inicial— codifica de forma directa, bit a bit, los elementos correspondientes a la cadena solución y que satisface la relación del problema original.

Definimos las dos funciones de la reducción en FNP (siguiendo el esquema formal de la pág. 38 del Tema 4):

1. **Transformador de entrada (R):** Toma la instancia x de el problema original y aplica la reducción de Cook-Levin para generar la fórmula de cláusulas de FSAT: $C = R(x)$.
2. **Transformador de soluciones (S):** Es un algoritmo que toma la asignación de variables de verdad A que satisface a la fórmula calculada C y, leyendo de forma directa las posiciones asignadas a las variables que representaban las opciones de la máquina, reconstruye y escribe el string solución $y = S(A, x)$.

Dado que ambas funciones de transformación operan eficientemente en espacio logarítmico, queda demostrado que FSAT es FNP-completo.

4.4.7 Ejercicio 8: (a) El Problema de los Spines con $J_{ij} \geq 0$ está en P

Dada la función de energía del sistema sobre un grafo $G = (V, E)$:

$$R(s) = -\sum_{(i,j)\in E} J_{ij} s_i s_j$$

Donde cada nodo tiene asignado un spin $s_i \in \{-1, +1\}$. Analicemos algebraicamente el término multiplicativo de la interacción para cada arista del grafo:

- Si los dos extremos conectados tienen asignado el mismo signo ($s_i = s_j$), entonces el producto es par y positivo: $s_i s_j = 1$.
- Si los extremos tienen signos opuestos ($s_i \neq s_j$), entonces el producto es negativo: $s_i s_j = -1$.

El sumatorio se reescribe analíticamente de la energía total descomponiéndolo en estos dos conjuntos de aristas excluyentes:

$$R(s) = -\left(\sum_{s_i = s_j} J_{ij} - \sum_{s_i \neq s_j} J_{ij} \right)$$

Definamos $W = \sum_{(i,j)\in E} J_{ij}$ como la constante fija que representa la suma total de los pesos de todas las aristas del grafo original. Es evidente que la suma de las interacciones de los nodos con igual signo se puede calcular restando del total las aristas de signo opuesto: $\sum_{s_i=s_j} J_{ij} = W - \sum_{s_i \neq s_j} J_{ij}$. Sustituyendo este término en la ecuación de energía:

$$R(s) = -\left(W - \sum_{s_i \neq s_j} J_{ij} - \sum_{s_i \neq s_j} J_{ij} \right) = -W + 2 \sum_{s_i \neq s_j} J_{ij}$$

Se trata de determinar si existe una configuración de spines tal que $R(s) \leq K$. Despejando el sumatorio de la equivalencia matemática obtenemos:

$$-W + 2 \sum_{s_i \neq s_j} J_{ij} \leq K \iff \sum_{s_i \neq s_j} J_{ij} \leq \frac{K + W}{2}$$

Reducción al Problema del Corte Mínimo

El término $\sum_{s_i \neq s_j} J_{ij}$ representa exactamente la definición matemática del **peso de un corte** en un grafo; es decir, la suma de los valores de las aristas que conectan el conjunto de vértices de la partición asignada al bando $+1$ con los vértices de la partición asignada al bando -1.

Como la hipótesis del enunciado nos garantiza estrictamente que $J_{ij} \geq 0$ **para todas las aristas del grafo**, el problema se reduce de forma directa a encontrar el **Corte Mínimo Global** (*Global Minimum Cut*) de un grafo con capacidades no negativas.

Este problema clásico se resuelve de forma exacta y eficiente en tiempo polinómico utilizando algoritmos estándar de flujos sobre redes (como el algoritmo de Edmonds-Karp o el algoritmo de Ford-Fulkerson por caminos cortos estudiado en el Tema 3), que operan en un coste polinómico de $O(|V| \cdot |E|^2)$.

Por tanto, al poder calcular el valor mínimo absoluto de la energía en tiempo polinómico y evaluar si es menor o igual que la cota solicitada, **el problema está en P**.

4.5 Simulacro de la tercera relación

Ejercicio 1 (El Palíndromo en Clase L)

Enunciado: Demuestre formalmente que el problema de determinar si una palabra de entrada es un palíndromo (se lee igual de izquierda a derecha que de derecha a izquierda) pertenece a la clase

de complejidad **L**.

Resolución detallada: Para demostrar que el problema pertenece a la clase **L** (espacio logarítmico), se utiliza el modelo computacional estándar para la evaluación de espacio sublineal: una Máquina de Turing equipada con una **cinta de entrada de solo lectura**, cuyo espacio no se contabiliza y que permite mover el cabezal libremente hacia adelante y hacia atrás, y una cinta de trabajo de lectura/escritura separada.

Diseño del Algoritmo: 1. Inicializamos dos punteros (contadores numéricos) en la cinta de trabajo: $i = 1$ apuntando al principio de la palabra de entrada, y $j = n$ apuntando al símbolo final. 2. Iniciamos un bucle que se ejecuta mientras $i < j$. 3. En cada iteración, la máquina mueve el cabezal de la cinta de entrada a la posición i para leer su símbolo, y posteriormente lo mueve a la posición j para leer el suyo. 4. Si los símbolos leídos en i y j son diferentes, la máquina **rechaza** inmediatamente, ya que la palabra no es simétrica. 5. Si los símbolos coinciden, se incrementa el puntero i en 1 y se decrementa el puntero j en 1, continuando la comprobación hacia el centro de la palabra. 6. Si el bucle finaliza sin haber encontrado ninguna diferencia, la máquina **acepta** la entrada.

Justificación de Complejidad (Espacio): Los únicos datos que se han almacenado en la cinta de trabajo durante toda la ejecución han sido los contadores i y j. Como el valor numérico máximo que pueden alcanzar estos contadores está estrictamente acotado por la longitud total de la entrada n, representarlos en formato binario requiere exactamente $\lceil \log_2(n+1) \rceil$ casillas de memoria. Al emplear un número constante de contadores de este tamaño, el espacio de memoria de trabajo total consumido es $O(\log n)$, lo que demuestra formalmente que el problema se encuadra en la clase **L**.

Ejercicio 2 (Análisis de Conjuntos Numéricos)

Enunciado: Se plantean los dos siguientes problemas de decisión sobre conjuntos de números: * **Problema A:** Dado un conjunto de números, determinar si existe un elemento que sea la mediana. Si la cantidad total de números en la entrada no es impar, la máquina debe rechazar automáticamente. * **Problema B:** Dado un conjunto de números, determinar si es posible agrupar todos sus elementos en parejas de tal forma que la suma de los dos elementos de cada pareja sea exactamente la misma.

Sabiendo que uno de los problemas pertenece a la clase **L** y el otro pertenece a la clase **P**, identifique a qué clase de complejidad pertenece cada uno y demuestre formalmente su clasificación detallando el algoritmo utilizado y analizando su consumo de recursos.

Resolución detallada:

1. El Problema A (La Mediana) pertenece a la Clase L Para demostrar que este problema se puede decidir en espacio logarítmico, diseñamos un algoritmo usando la misma máquina de Turing con cinta de entrada de solo lectura descrita anteriormente. Es fundamental **no copiar nunca la entrada en la cinta de trabajo**, ya que eso supondría gastar un espacio de tamaño lineal.

Algoritmo y Justificación: * La máquina primero cuenta el número total de elementos n. Si n es par, rechaza automáticamente la entrada. * Si es impar, utiliza un contador i para iterar por cada elemento del array, tratándolo como el "candidato a mediana". * Para cada candidato $Array[i]$, lanza un segundo bucle interno con un índice j que recorre todo el conjunto. Durante este recorrido, utiliza dos contadores numéricos auxiliares: `C_menores` (que suma 1 si $Array[j] < Array[i]$) y

`C_mayores` (que suma 1 si $Array[j] > Array[i]$). * Al finalizar el recorrido interno, si `C_menores` es exactamente igual a `C_mayores`, la máquina **acepta** (ha encontrado la mediana). Si no coinciden, reinicia los contadores y pasa al siguiente candidato con el índice i. * **Espacio:** La máquina solo ha utilizado 4 variables enteras (i, j, `C_menores` y `C_mayores`). Como sus valores están acotados por la longitud de la entrada n, cada una ocupa un espacio binario de $\lceil \log_2(n+1) \rceil$. Al utilizar un número fijo de contadores logarítmicos, el espacio total consumido es $O(\log n)$, justificando su pertenencia a **L**.

2. El Problema B (Parejas de sumas iguales) pertenece a la Clase P Este problema no se puede resolver eficientemente en espacio L sin alterar la entrada, por lo que demostraremos que pertenece a la clase **P** diseñando una Máquina de Turing determinista que lo resuelva en un tiempo acotado por un polinomio $O(n^c)$.

Algoritmo y Justificación: * **Paso 1:** La máquina ordena todos los elementos del array de menor a mayor. Los algoritmos de ordenación deterministas más eficientes toman un tiempo de $O(n \log n)$. * **Paso 2:** Por pura lógica matemática, para que todos los pares sumen lo mismo, el número más pequeño ($Array$) debe estar emparejado obligatoriamente con el número más grande ($Array[n]$). La máquina suma ambos extremos para obtener la única constante de "Suma Objetivo" posible: $S = Array + Array[n]$. Esto cuesta un tiempo constante $O(1)$. * **Paso 3:** La máquina emplea dos punteros (uno al inicio y otro al final) y los va moviendo hacia el centro del array paso a paso. En cada iteración suma los dos elementos a los que apuntan y verifica que su resultado sea exactamente igual a S. Este recorrido toma un tiempo lineal $O(n)$. * Si alguna pareja falla, **rechaza**. Si los punteros se cruzan en el centro y todas han sumado S, **acepta**. * **Tiempo:** El paso de mayor coste es la ordenación ($O(n \log n)$). Como el tiempo global de ejecución está dominado por este paso y, por tanto, acotado superiormente por la función polinómica cuadrática genérica $O(n^2)$, queda formalmente demostrado que el problema pertenece a la clase **P**.

Ejercicio 3 (Cuestión Conceptual: El Óptimo del Viajante de Comercio)

Enunciado: En relación con el Problema del Viajante de Comercio, justifique detalladamente por qué comprobar si un circuito dado es la solución ÓPTIMA absoluta del grafo no pertenece a la clase NP ni a la clase coNP, sino que su resolución requiere utilizar clases de ESPACIO.

Resolución detallada: El problema de certificar que un circuito es el **óptimo absoluto** en el Viajante de Comercio es estructuralmente distinto a simplemente comprobar si existe un camino con un coste menor o igual a un límite dado.

Para poder convencer a un verificador de que un circuito concreto es la solución óptima, el certificado aportado tendría que demostrar dos afirmaciones de naturaleza opuesta simultáneamente: 1. **Condición existencial:** Demostrar que el camino existe, es válido y tiene un coste X. Esta propiedad es característica de la clase **NP** ("Existe un camino..."). 2. **Condición universal:** Demostrar que **NO existe** en la totalidad del grafo ningún otro circuito posible cuyo coste sea estrictamente menor que X. Para refutar la existencia de alternativas mejores necesitamos evaluar todas las combinaciones, lo cual es una propiedad característica de la clase **coNP** ("Para todo camino, ninguno es mejor...").

Dado que evaluar una "solución óptima" exige simultáneamente verificar un "Existe" y un "Para todo", un certificado de tamaño polinómico simple no puede abarcar la evaluación de todas las rutas alternativas, por lo que certificar el óptimo no se encuadra de forma aislada ni en **NP** ni en **coNP**.

Resolución mediante clases de ESPACIO: A pesar de lo anterior, el problema sí puede resolverse de forma determinista si apelamos a los recursos de **ESPACIO** (dentro de clases como PSPACE). Una Máquina de Turing determinista puede recorrer de manera sistemática todo el árbol de caminos posibles del grafo mediante un algoritmo de búsqueda en profundidad (backtracking). La clave radica en que la máquina **reutiliza constantemente la memoria**. Al explorar rama por rama, solo necesita almacenar el estado de la ruta actual y llevar la cuenta del "mejor coste encontrado hasta el momento". Por lo tanto, el espacio de memoria que consume está acotado de forma **lineal** $O(n)$ respecto al tamaño del grafo, logrando encontrar la solución óptima sin agotar la memoria, a pesar de que el proceso requiera un tiempo de ejecución de orden exponencial.

Ejercicio 4 (Estructuras Anidadas y la restricción sin retroceso)

Enunciado: Dada una cadena de entrada compuesta por dos tipos de paréntesis (por ejemplo, `()` y `[]`), demuestre que determinar si están correctamente emparejados y anidados pertenece a la clase L. Indique teóricamente por qué este problema requeriría obligatoriamente un espacio lineal $O(n)$ si la cabeza de lectura no tuviera permitido retroceder.

Resolución detallada:

Parte 1: Pertenencia a la Clase L (Modelo estándar) Se utiliza una Máquina de Turing que dispone de una cinta de entrada de solo lectura **con capacidad para retroceder** (releer) y una cinta de trabajo separada. Para que la cadena de paréntesis múltiples sea válida, se deben verificar dos propiedades: el balance global de la cadena ignorando los tipos, y que cada cierre coincida en tipo con su apertura correspondiente en el anidamiento.

Algoritmo en Espacio $O(\log n)$: 1. La máquina recorre la palabra con un **contador de balance global** b, incrementándolo en las aperturas y decrementándolo en los cierres, y comprobando que nunca sea menor a 0 para validar el balance. 2. Cada vez que el cabezal de lectura encuentra un símbolo de cierre (por ejemplo en la posición j), la máquina detiene su avance, memoriza el índice j y debe localizar su apertura asociada. 3. La apertura asociada a ese cierre será exactamente la **última posición** $i < j$ tal que el segmento intermedio de la palabra $(x[i+1..j-1])$ está **perfectamente balanceado**. 4. Para encontrar ese punto operativamente, el algoritmo lanza un segundo recorrido auxiliar desde el inicio hasta j. Lleva un contador sub-balance y memoriza el índice i de la última apertura que provocó que el sub-balance se situara en cero justo antes del cierre. 5. Una vez localizada la posición de apertura i, la máquina comprueba de manera directa que el símbolo situado en i y el situado en j **sean exactamente del mismo tipo** (ambos redondos o ambos cuadrados). Si difieren, rechaza.

Justificación Matemática: Durante toda esta operación, la máquina ha evitado utilizar estructuras de datos dinámicas como pilas. Solamente ha necesitado almacenar en la cinta de trabajo unos pocos contadores enteros (como b) y punteros a índices de la cinta (como i y j). Como el valor numérico máximo de todas estas variables está estrictamente limitado por la longitud total n, su escritura binaria precisa un número constante de contadores de tamaño $\lceil \log_2(n+1) \rceil$. En consecuencia, el espacio total empleado en la memoria es $O(\log n)$, lo que demuestra formalmente su pertenencia a **L**.

Parte 2: Por qué requiere espacio $O(n)$ si NO se permite retroceder (Modelo Online) Si le prohibimos a la máquina mover su cabezal hacia atrás, le impedimos ejecutar la estrategia de "buscar emparejamientos bajo demanda". Teóricamente, esto la obliga a emplear un espacio **lineal** $\Omega(n)$, lo cual se demuestra usando un **argumento de distinguibilidad (fooling set)**.

1. Consideremos un conjunto de 2^m cadenas distintas que están formadas exclusivamente por un prefijo de m símbolos de apertura combinados de redondos y cuadrados (por ejemplo, `([(([` vs `([[[[`).
2. Para cualquier par de prefijos diferentes $u \neq u'$, siempre existirá una posición concreta donde el tipo de paréntesis elegido no coincide.
3. Si le suministramos a la máquina la continuación exacta de cierres del primer prefijo (su "espejo" lógicamente correcto), esta concatenación formará una palabra válida y la máquina la **aceptará**. Sin embargo, si le adjuntamos esa misma continuación al segundo prefijo u', los tipos chocarán y la máquina debe **rechazar**.
4. Como la máquina no puede retroceder en la cinta para revisar qué prefijo leyó al principio, su decisión futura depende única y exclusivamente de la "fotografía" o estado en el que quedó su memoria tras leer la primera mitad de la entrada.
5. Para evitar equivocarse al evaluar las 2^m posibles continuaciones en el futuro, la máquina **debe obligatoriamente alcanzar una configuración de memoria completamente distinta para cada uno de los 2^m prefijos posibles**.
6. Sabiendo que una memoria de tamaño s puede codificar como máximo $2^{O(s)}$ configuraciones diferentes, la desigualdad $2^{O(s)} \geq 2^m$ requiere matemáticamente que $s = \Omega(m)$. Tomando la longitud de los prefijos como la mitad de la palabra ($m = \lfloor n/2 \rfloor$), se demuestra formalmente que la máquina precisa almacenar todos los tipos, necesitando forzosamente un espacio lineal $O(n)$.

Ejercicio 5 (Cierre de la Clase NP para la Estrella de Kleene L^*)

Enunciado del Simulacro: Demuestre, definiendo rigurosamente su certificado y su verificador, que si un lenguaje L pertenece a la clase **NP**, entonces la clausura de Kleene L^* también pertenece a **NP**.

Resolución detallada: La definición de la estrella de Kleene L^* establece que una palabra $x \in L^*$ si, y solo si, se puede particionar en k trozos consecutivos $(w_1, w_2, \ldots, w_k)$ de forma que cada subpalabra o bloque w_j pertenezca obligatoriamente al lenguaje original L.

1. Hipótesis de partida: Dado que el enunciado nos garantiza que $L \in$ NP, la teoría garantiza que existe un verificador determinista polinómico V y que toda palabra válida cuenta con un certificado individual de tamaño polinómico acotado por un polinomio $q(|w|)$.

2. El Diseño del Certificado para L^* (El Súper-Certificado): Para demostrar que la palabra completa x (de longitud n) pertenece a L^*, no cabe comprobarlo de golpe, por lo que el certificado aportado debe constar de dos elementos clave: * **Los puntos de partición (Cortes):** Una secuencia de índices $i_0 = 0 < i_1 < i_2 < \cdots < i_k = n$ que delimitan exactamente dónde empieza y dónde termina cada trozo w_j dentro de la palabra x. * **Las pruebas individuales:** Una lista de sub-certificados $(c_1, c_2, \ldots, c_k)$, donde cada c_j es el certificado individual que convence al verificador original de que ese trozo concreto $w_j \in L$.

3. El Verificador Determinista V^* para L^*: El algoritmo verificador recibe la entrada x y el súper-certificado, operando de la siguiente manera: 1. Comprueba lógicamente que los índices de corte (i_j) están ordenados, no se solapan y cubren toda la longitud n. 2. Para cada subpalabra w_j delimitada por los cortes, ejecuta el verificador original de la hipótesis: $V(w_j, c_j) == 1$. 3. Si el verificador devuelve "1" (Acepta) para todos y cada uno de los trozos, V^* **acepta**. Si alguna subpalabra es rechazada, V^* rechaza automáticamente.

4. Justificación Matemática (El Cierre Polinómico): Para probar la pertenencia a NP, es preciso justificar que estos recursos no explotan exponencialmente: * **Tamaño del Súper-Certificado:** Como la palabra se puede partir como máximo en n trozos de 1 carácter ($k \leq n$), hay a lo sumo n índices de corte, y cada índice requiere $O(\log n)$ bits. Por otro lado, la suma de las longitudes de todos los trozos es exactamente n ($\sum |w_j| = n$). Dado que cada certificado individual c_j está acotado por $q(|w_j|)$, el tamaño total de la suma de todos los certificados no superará $n \cdot q(n)$, lo cual es una medida estrictamente polinómica. * **Tiempo de Ejecución:** Comprobar los cortes toma tiempo lineal. La ejecución del verificador original V es de coste polinómico y se llama, a lo sumo, n veces. Como multiplicar un polinomio por n da como resultado otro polinomio, el proceso entero requiere un tiempo determinista polinómico. * **Conclusión:** Existe un certificado de tamaño polinómico evaluable en tiempo polinómico, por lo que queda demostrado que $L^* \in \mathbf{NP}$.

Ejercicio 6 (Composición de Funciones)

Enunciado del Simulacro: Evalúe el comportamiento de la clase de funciones exponenciales de la forma $\{k^n : k > 0\}$ bajo la composición polinómica por la izquierda $p(f(n))$ y por la derecha $f(p(n))$, asumiendo para la prueba que $p(n) = n^l$ ($l > 0$). ¿Es la clase cerrada en ambos casos? Demuéstrelo matemáticamente.

Resolución detallada: Para evaluar si la clase sobrevive o es "absorbida" bajo composición polinómica, se introduce la función genérica $p(n) = n^l$ (donde l es una constante entera ≥ 2 para evaluar el crecimiento no lineal).

1. Composición polinómica por la izquierda $p(f(n))$: En este escenario, sustituimos la función completa dentro del polinomio, lo que equivale a elevar la función a una potencia constante l:

$$p(f(n)) = (k^n)^l$$

Por las propiedades de las potencias, multiplicamos los exponentes:

$$(k^n)^l = k^{n \cdot l} = (k^l)^n$$

Dado que k y l son dos valores constantes positivas, cabe definir una nueva constante $K = k^l$. Si sustituimos esto, nos queda la función K^n. Como el resultado sigue teniendo la misma estructura exacta de una base constante elevada a n ($K^n \in \{k^n\}$), la función no se ha salido de la familia. **Conclusión:** La familia exponencial de base constante **SÍ es cerrada** bajo la composición por la izquierda.

2. Composición polinómica por la derecha $f(p(n))$: En este caso, se introduce el polinomio n^l directamente como el argumento (la variable n) de la función exponencial, lo que se traduce en colocar el polinomio en el exponente:

$$f(p(n)) = k^{(n^l)}$$

Para $l \geq 2$, la expresión k^{n^l} crece de forma dramáticamente más violenta que la clase base. Para que estuviera dentro de la familia original $\{k^n : k > 0\}$, tendría que existir alguna constante mágica enorme K' tal que $k^{n^l} \leq (K')^n$ para cualquier n grande. Sin embargo, es matemáticamente imposible acotar k^{n^l} usando una función con exponente puramente lineal (n), ya que un grado de libertad adicional introducido en el exponente (n^l) "explota" superando a cualquier constante

K' que intentemos fijar en la base. **Conclusión:** La familia no contiene ninguna función capaz de dominar ese crecimiento masivo. Por lo tanto, la clase exponencial de base constante **NO es cerrada** bajo la composición por la derecha.

Pregunta 7 (Separación de Clases: NP $\neq$ ESPACIO(n))

Enunciado: Demuestre formalmente que NP $\neq$ ESPACIO(n). Para ello, base su demostración en el comportamiento que tiene cada una de estas dos clases frente a las reducciones computables en espacio logarítmico (L-reducciones).

Resolución detallada: Para demostrar matemáticamente que ambas clases no son iguales, no vamos a evaluar si una contiene a la otra, sino que emplearemos la táctica de buscar una propiedad estructural que una clase posea y la otra no. Dos clases idénticas tendrían las mismas propiedades; al diferir en al menos una, queda demostrado que son distintas. La propiedad a evaluar es la **clausura bajo reducciones en espacio logarítmico ($\propto_L$)**.

Paso 1: Demostrar que NP sí es cerrada para L-reducciones. Decir que es cerrada significa que si $P_1 \propto_L P_2$ y $P_2 \in$ NP, entonces obligatoriamente $P_1 \in$ NP. Una reducción calculable en espacio logarítmico $O(\log n)$ puede alcanzar como máximo un número polinómico de configuraciones de memoria distintas ($2^{O(\log n)} = n^{O(1)}$) sin entrar en bucle infinito. Por tanto, cualquier reducción en espacio L se ejecuta a lo sumo en tiempo polinómico. Al encadenar esta reducción polinómica con el verificador polinómico del problema P_2, el tiempo de cómputo global sigue siendo estrictamente polinómico, demostrando que **la clase NP es cerrada bajo estas reducciones**.

Paso 2: Demostrar que ESPACIO(n) NO es cerrada para L-reducciones. Para este paso, nos apoyamos en el **Teorema de la Jerarquía de Espacio**, el cual garantiza que disponer de más memoria estricta nos permite resolver más problemas; en concreto: ESPACIO$(n) \subsetneq$ ESPACIO(n^2). Gracias a esto, cabe fijar un problema testigo "muy difícil" A tal que consta de forma absoluta que $A \in$ **ESPACIO**(n^2) **pero** $A \notin$ **ESPACIO**(n).

Se aplica ahora la técnica del **Padding** (Relleno). Se construye un lenguaje disfrazado B añadiendo caracteres neutros # a la entrada original $x \in A$ hasta que el tamaño total de la nueva cadena sea el cuadrado de la original ($N = |x|^2$). * $B \in$ **ESPACIO**(n)**:** Para que una máquina decida B, solo tiene que extraer x y ejecutar el decididor original de A. Como este decididor consumía un espacio de $|x|^2$, y la entrada de ahora mide exactamente $N = |x|^2$, el espacio consumido respecto al nuevo tamaño es exactamente lineal $O(N)$. * **La reducción:** El proceso de transformar la entrada corta x en la entrada gigante $x\#^{|x|^2-|x|}$ consiste en ir contando caracteres, lo cual requiere apuntadores logarítmicos. Así que A se reduce a B en espacio L ($A \propto_L B$).

Conclusión final: Queda demostrado que $A \propto_L B$ y que $B \in$ ESPACIO(n). Si la clase ESPACIO(n) fuera verdaderamente cerrada bajo estas reducciones, obligaría a que A estuviera en ESPACIO(n). Sin embargo, la Jerarquía de Espacio nos prohíbe esto ($A \notin$ ESPACIO(n)). Para evitar esta contradicción matemática, la única salida es que **ESPACIO**(n) **NO es cerrada**. Al diferir en esta propiedad fundamental, concluimos formalmente que **NP** $\neq$ **ESPACIO**(n).

Pregunta 8 (Tratabilidad Numérica de la Exponenciación Modular)

Enunciado: Se desea probar que el problema de la Exponenciación Modular (determinar si $a^b \equiv c \pmod{p}$) pertenece a la clase **P**. Demuestre esta pertenencia detallando el algoritmo eficiente

necesario, y justifique previamente por qué un enfoque de multiplicaciones sucesivas estándar tendría un coste exponencial respecto al tamaño de la entrada en bits.

Resolución detallada: El problema exige demostrar que la decisión $a^b \equiv c \pmod{p}$ se puede computar en tiempo polinómico respecto a la longitud de la entrada.

1. Análisis de los recursos y la trampa del enfoque estándar: En la teoría de complejidad, el tamaño de la entrada n representa la **cantidad total de bits** necesarios para almacenar los cuatro números (a, b, c, p) en la memoria de la máquina. Si decidiéramos abordar el problema con un bucle simple de fuerza bruta que multiplicase consecutivamente $a \cdot a \cdot a \ldots$ dando tantas vueltas como indique el exponente b, caeríamos en una trampa exponencial. Dado que el tamaño de la entrada está en bits, el valor numérico del exponente b puede ascender a un orden de magnitud enorme de hasta 2^n. Por consiguiente, dar esa cantidad de vueltas requeriría efectuar un tiempo de ejecución exponencial $O(2^n)$, un coste inasumible para la clase P.

2. El algoritmo eficiente (Exponenciación Binaria por Cuadrados Sucesivos): Para situar el problema en la clase **P**, es preciso evadir el valor gigante de b y fijarnos únicamente en su longitud en bits. Diseñamos una Máquina de Turing determinista que emplea la **exponenciación binaria**: * La máquina reserva una variable acumuladora inicializada a 1 y recorre, uno a uno, los bits que componen el exponente b (que son a lo sumo n bits). * En cada paso (iteración del bucle), la máquina **eleva al cuadrado** su variable de base actual, aplicando la operación módulo p inmediatamente para garantizar que el número resultante nunca supere el tamaño de la entrada (evitando desbordamientos de memoria). * Si el bit evaluado en esa vuelta es un 1, la máquina además multiplica el acumulador por el valor actual y vuelve a aplicar el módulo p. * Al concluir el bucle por los bits, si el resultado contenido en el acumulador coincide exactamente con c, la máquina acepta. En caso contrario, rechaza.

3. Justificación Matemática del Tiempo: A diferencia del caso exponencial, este bucle determinista da un número de iteraciones estrictamente equivalente a la cantidad de bits del exponente. Como el exponente es parte de la entrada, el bucle da a lo sumo $O(n)$ vueltas. En el interior de cada ciclo, la máquina solo realiza multiplicaciones y divisiones enteras (el módulo) entre variables cuyo tamaño en bits está matemáticamente acotado. Dichas operaciones aritméticas básicas entre números de tamaño n se pueden realizar eficientemente en tiempo $O(n^2)$. Por consiguiente, el tiempo total global del algoritmo es el producto del número de vueltas por el coste interno $(O(n) \cdot O(n^2))$, lo que resulta en un tiempo estrictamente acotado por un polinomio $O(n^3)$. Queda demostrado así formalmente que el problema se ubica dentro de la **clase P**.

Pregunta 9 (El Problema de la Factorización)

Enunciado: Dado un número x, se desea determinar si tiene un divisor k en el rango abierto $1 < k < y$. Demuestre con todo detalle que este problema pertenece a la intersección $\mathrm{NP} \cap \mathrm{coNP}$, indicando los certificados exactos requeridos para las instancias afirmativas y negativas.

Resolución detallada: Para demostrar que el problema de la factorización pertenece a la intersección **NP ∩ coNP**, es imperativo estructurar la demostración validando su pertenencia a ambas clases de complejidad por separado.

1. Demostración de pertenencia a NP (Instancias afirmativas): Si la respuesta a la pregunta es "SÍ", existe un divisor pequeño oculto en ese rango. * **Certificado:** Definimos como certificado simplemente el propio valor numérico del divisor k. * **Verificador:** Una Máquina de Turing Determinista (MTD) tomará la entrada (x, y) y el certificado k, y evaluará en tiempo

polinómico dos propiedades: 1. Validar las cotas numéricas: comprobar que $1 < k < y$. 2. Efectuar la división entera para constatar que el resto $x \pmod{k} == 0$. * **Justificación de complejidad:** La operación aritmética de división entera entre números de un tamaño n medido en bits se computa eficientemente en un tiempo algorítmico acotado por $O(n^2)$. Dado que un certificado de tamaño polinómico se puede verificar en tiempo polinómico, queda demostrado que el problema **pertenece a NP**.

2. Demostración de pertenencia a coNP (Instancias negativas): Para probar que el problema original pertenece a coNP, es preciso demostrar teóricamente que el problema complementario ("¿Es cierto que x NO tiene divisores en el rango $1 < k < y$?") pertenece a la clase NP. * **Certificado para el NO:** Para refutar la existencia de divisores en ese rango, el certificado debe proporcionar la **descomposición completa en factores primos de** x ($x = p_1^{e_1} \cdot p_2^{e_2} \cdots p_m^{e_m}$). El elemento crucial de la demostración es que se debe adjuntar obligatoriamente el **Certificado de Pratt** para cada uno de los factores primos p_i suministrados en la lista. * **Verificador:** Con este súper-certificado, la MTD ejecutará tres pasos deterministas: 1. Multiplicará todos los factores de la lista para corroborar que el resultado general iguala exactamente a la entrada x. 2. Validará irrefutablemente que cada uno de esos factores es primo comprobando la validez de sus respectivos Certificados de Pratt (cuyo formato formal sería, por ejemplo, $\text{Cert}(13) = \Big(\big(13 : 2\ (2,3)\big),\ \big(3 : 2\ (2)\big)\Big)$). Esto se realiza en tiempo polinómico. 3. Comprobará visualmente que todos y cada uno de los factores primos validados cumplen la condición de ser **mayores o iguales que** y ($p_i \geq y$). * **Justificación matemática:** El Teorema Fundamental de la Aritmética nos asegura que, si todos los componentes o "ladrillos" primos indivisibles de un número son estrictamente mayores o iguales que y, es algebraicamente imposible que combinándolos podamos construir un divisor compuesto menor que y. Dado que todo el proceso de validación opera en tiempo polinómico, el problema complementario está en NP.

Conclusión: Como el problema original pertenece a NP y su complementario también (lo que lo ubica en coNP), el lenguaje queda clasificado lógicamente en la región $\mathbf{NP} \cap \mathbf{coNP}$.

Pregunta 10 (Problemas de Búsqueda y TFNP)

Enunciado: Todo lenguaje L arbitrario que pertenezca a la intersección NP ∩ coNP sugiere un problema de búsqueda de funciones. Defina dicho problema sugerido y demuestre teóricamente que pertenece a la clase TFNP (Total Functional NP).

Resolución detallada: La clase **TFNP (Total Functional NP)** abarca problemas de búsqueda donde, además de operar con certificados polinómicos, existe una garantía matemática inquebrantable de que **siempre existirá al menos una solución válida** para cualquier entrada (la relación es total).

1. Definición del problema sugerido: Sea un lenguaje arbitrario $L \in \text{NP} \cap \text{coNP}$. Por la definición de pertenencia simultánea a estas clases, se sabe que: * Al estar en NP, existe una Máquina de Turing verificadora polinómica M_1 que valida un certificado c_1 demostrando que $x \in L$. * Al estar en coNP (es decir, $\bar{L} \in \text{NP}$), existe otro verificador polinómico M_2 que evalúa un certificado c_2 demostrando que $x \notin L$.

Basándonos en esto, el problema de búsqueda que la intersección sugiere se define formalmente como: * *"Dada una cadena de entrada x, encontrar un objeto c que actúe como un testigo comprobable de forma eficiente de que, o bien $x \in L$ (satisfaciendo a M_1), o bien $x \notin L$ (satisfaciendo a M_2)."*

2. Demostración de pertenencia a TFNP: Para clasificar este problema en **TFNP**, es preciso

validar las dos condiciones principales de la clase:

- **Verificación polinómica (Condición FNP):** Tanto c_1 como c_2 son certificados cuya longitud en memoria está matemáticamente acotada por polinomios. Un algoritmo evaluador general puede simplemente recibir el certificado c suministrado y ejecutar secuencialmente los verificadores M_1 y M_2 en tiempo polinómico para determinar si la solución cumple las condiciones del problema. Por tanto, es tratable en el esquema NP funcional.
- **Totalidad garantizada (Condición T):** Para demostrar que la función nunca queda "indefinida", invocamos el principio lógico de la **Ley del bando excluido**. Para cualquier cadena x que se introduzca como entrada, independientemente de su estructura, rige una dualidad estricta: **obligatoriamente se debe cumplir que** $x \in L$ **o que** $x \notin L$. Como forzosamente una de estas dos afirmaciones será una verdad matemática, la existencia del testigo (ya sea la solución c_1 que prueba el "Sí", o el testigo c_2 que prueba el "No") está garantizada para el 100 % de las instancias. Nunca existirá una entrada x que carezca de solución.

Conclusión: Al haber construido un problema de búsqueda que es verificable algorítmicamente en tiempo polinómico y cuya solución está siempre garantizada mediante la totalidad lógica, concluimos que se inserta perfectamente en los márgenes teóricos de la clase **TFNP**.

Pregunta 11 (Completitud Funcional: FSAT)

Enunciado: Demuestre formalmente, validando los dos pasos exigidos por la teoría de complejidad de funciones, que el problema **FSAT** (la versión de búsqueda que exige devolver la asignación de valores de verdad) es **FNP-completo**.

Resolución detallada: Para demostrar que la versión de búsqueda funcional de SAT (FSAT) es completa para la clase FNP, es preciso justificar formalmente dos pasos: su pertenencia a la clase y su completitud (dificultad funcional).

Paso 1: Demostrar que FSAT $\in$ FNP Un problema pertenece a la clase de búsqueda FNP si la solución solicitada tiene un tamaño acotado y es verificable en tiempo polinómico: 1. Dada una entrada formada por un conjunto de cláusulas lógicas C, el objeto solución que buscamos es una asignación de verdad A (qué variables son Verdaderas y cuáles Falsas). 2. La longitud en bits de esta asignación A está estrictamente **acotada de forma lineal** por el número total de variables lógicas presentes en C. Por tanto, el tamaño de la solución cumple el requisito polinómico. 3. El algoritmo verificador simplemente toma la asignación A propuesta, sustituye los valores en las cláusulas de C y evalúa las puertas lógicas. Esta comprobación se ejecuta eficientemente en **tiempo polinómico**. Al cumplir ambas propiedades, concluimos que FSAT $\in$ FNP.

Paso 2: Demostrar la completitud (FNP-dificultad) Es preciso probar que cualquier problema de búsqueda genérico $\Pi \in$ FNP puede ser reducido a FSAT operando en espacio logarítmico. A diferencia de los problemas de decisión, aquí hacen falta dos funciones (transformador de entrada R y transformador de solución S): 1. Por el **Teorema de Cook-Levin**, se sabe que la ejecución de cualquier Máquina de Turing No Determinista se puede traducir a un conjunto de cláusulas proposicionales (SAT) computándolo en espacio logarítmico. 2. La propiedad fundamental de esta reducción es que es **"constructiva respecto a las variables"**. Esto significa que, dentro de la enorme fórmula proposicional generada por la máquina, hay un pequeño grupo de variables lógicas que codifican y almacenan bit a bit la solución original del problema. 3. Definimos el **transformador** R para que aplique Cook-Levin y convierta la entrada de Π en cláusulas. A continuación, definimos

el **transformador** S para que tome la asignación A devuelta por FSAT, lea únicamente los valores V/F de esas variables constructivas clave, y traduzca esos bits para imprimir la solución exacta de II. 4. Como ambos transformadores R y S son computables eficientemente y permiten que cualquier problema construya su solución a través de FSAT, queda demostrado que **FSAT es FNP-completo**.

Pregunta 12 (El Problema de los Spines / Red Feliz)

Enunciado: Dado un grafo $G = (V, E)$ donde cada arista (i, j) tiene asignada una fuerza de interacción $J_{ij} \geq 0$, y donde cada nodo tiene un spin $s_i \in \{-1, +1\}$. La energía total se define como $R(s) = -\sum_{(i,j)\in E} J_{ij} s_i s_j$. Demuestre analíticamente que el problema de determinar si existe una asignación tal que la energía sea menor o igual a una cota K ($R(s) \leq K$) se puede resolver en tiempo polinómico (**Clase P**).

Resolución detallada: Para demostrar que este problema es tratable (Clase P), no cabe probar todas las combinaciones de spines (lo cual sería un tiempo exponencial $O(2^n)$), sino que es preciso realizar un análisis algebraico sobre la fórmula de la energía que explote el hecho de que todos los pesos son positivos o cero ($J_{ij} \geq 0$).

1. Descomposición del Sumatorio: Al multiplicar los spines de los dos extremos de una arista ($s_i \cdot s_j$), los valores resultantes solo pueden ser dos: * Si ambos nodos tienen el **mismo signo** (por ejemplo, ambos $+1$ o ambos -1), su producto es positivo: $s_i s_j = 1$. * Si tienen **signos opuestos**, su producto es negativo: $s_i s_j = -1$.

Con esta premisa, cabe dividir el cálculo de la energía separando las aristas del grafo en esos dos conjuntos excluyentes:

$$R(s) = -\left(\sum_{s_i = s_j} J_{ij} - \sum_{s_i \neq s_j} J_{ij} \right)$$

2. Sustitución Matemática: Definimos una constante universal W equivalente a la suma total de todos los pesos del grafo: $W = \sum_{(i,j)\in E} J_{ij}$. Sabiendo que las aristas de igual signo equivalen al total menos las de distinto signo ($\sum_{s_i = s_j} J_{ij} = W - \sum_{s_i \neq s_j} J_{ij}$), lo sustituimos en la ecuación inicial:

$$R(s) = -\left(W - \sum_{s_i \neq s_j} J_{ij} - \sum_{s_i \neq s_j} J_{ij} \right) = -W + 2 \sum_{s_i \neq s_j} J_{ij}$$

3. Condición de Decisión y Algoritmo Determinista: El problema nos pide evaluar si se puede cumplir que $R(s) \leq K$, lo que matemáticamente equivale a despejar la nueva fórmula:

$$\sum_{s_i \neq s_j} J_{ij} \leq \frac{K + W}{2}$$

El sumatorio $\sum_{s_i \neq s_j} J_{ij}$ representa el coste de las aristas que conectan nodos con signos diferentes (el "peso del corte"). Como el enunciado garantiza que absolutamente todas las interacciones son nulas o positivas ($J_{ij} \geq 0$), **este sumatorio nunca puede ser negativo**.

Por tanto, el valor absoluto mínimo que puede alcanzar la energía se da cuando ese sumatorio vale exactamente 0. El algoritmo para lograr este mínimo es trivial: **asignar a todos los nodos del grafo exactamente el mismo signo** (por ejemplo, todos $+1$). En este caso, la energía colapsa a

su mínimo posible: $-W$.

4. Conclusión (Pertenencia a P): Una Máquina de Turing determinista simplemente tiene que recorrer las aristas para sumar su peso total y calcular el valor extremo $-W$. Si $-W \leq K$, acepta; si no, rechaza. Como la suma y la comparación de n aristas es un proceso aritmético lineal evaluable en tiempo estrictamente polinómico, el problema **pertenece a la Clase P**.

Capítulo 5

Convocatorias resueltas, 2021-2025

Cada apartado incluye: enunciado, identificación del tipo, clasificación y desarrollo (construcción de la máquina, búsqueda o reducción según corresponda).

5.1 2021

5.1.1 22 de junio de 2021

Ejercicio 1. Decidibilidad

1. *Dada MT* `M` *y palabra* `w`*, ¿es cierto que* `M` *NO acepta* `w`*?*
   - Tipo: comportamiento sobre una entrada fija; es directamente el problema base C-UNIVERSAL.
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: el problema UNIVERSAL ("¿`M` acepta `w`?") es semidecidible y no decidible (se semidecide con el simulador universal: acepta si `M` acepta `w`, y cicla en caso contrario). Por el Teorema de los complementarios, si UNIVERSAL es r.e. y no recursivo, su complementario "¿`M` no acepta `w`?" no es r.e. Por tanto, no es semidecidible.
2. *Dada una GIC, ¿es ambigua?*
   - Tipo: propiedad estructural de gramáticas independientes del contexto.
   - Clasificación: NO DECIDIBLE (indecidible).
   - Desarrollo: la ambigüedad de una GIC es un problema clásico indecidible (se demuestra por reducción del Problema de Correspondencias de Post: a partir de una instancia de PCP se construye una gramática que es ambigua si y solo si la instancia tiene solución). No se exige reproducir la reducción, pero conviene citarla.
3. *Dada una GIC* `G` *y una palabra* `u`*, ¿*`G` *genera* `u`*?*
   - Tipo: pertenencia de una palabra concreta al lenguaje de una GIC.
   - Clasificación: DECIDIBLE.
   - Desarrollo: el algoritmo CYK decide la pertenencia en tiempo $O(|u|^3)$ tras pasar `G` a forma normal de Chomsky. El procedimiento es finito y siempre termina con SÍ o NO.
4. *¿Existe una palabra de longitud* $\leq$*100 aceptada por* `M` *en menos de 1000 pasos?*
   - Tipo: doble cota finita (longitud de entrada y número de pasos).
   - Clasificación: DECIDIBLE.
   - Desarrollo: el conjunto de palabras de longitud $\leq$100 sobre `{0,1}` es finito (a lo sumo `2^{101}-1`). Para cada una se simula `M` durante 999 pasos como máximo. Como tanto el número de palabras como el de pasos por palabra son finitos, la simulación completa termina; se responde SÍ si alguna acepta dentro del límite, y NO en caso contrario.

Ejercicio 2. Verdadero/Falso

1. *Todo problema NP puede resolverse en espacio polinómico y tiempo exponencial.* VERDADERO. Se recorre el árbol de cálculo no determinista por backtracking: cada rama tiene profundidad polinómica, luego se reutiliza el espacio (polinómico) y el tiempo total es exponencial. Equivale a NP $\subseteq$ PESPACIO $\subseteq$ EXP.
2. *Una MT universal puede calcular cualquier cosa que cualquier otra MT pueda calcular.* VERDADERO por definición de máquina universal: recibe $\langle$M$\rangle$ y w y reproduce el cómputo de M sobre w.
3. *Se sabe que el complementario de SAT no tiene un algoritmo de tiempo polinómico.* FALSO. Afirmarlo equivaldría a demostrar P$\neq$coNP (y P$\neq$NP), cuestión abierta.
4. *Si A es NP-completo y A$\propto$B, entonces B es NP-completo.* FALSO. La reducción solo prueba que B es NP-difícil; para ser NP-completo debe además pertenecer a NP, lo que aquí no se afirma.
5. *Si P=NP, el Viajante de Comercio se resuelve en tiempo polinómico determinista.* VERDADERO. La versión decisión del TSP es NP-completa; si P=NP, toda la clase NP cae en P.

Ejercicio 3. NP-completitud — *Reducción de SUMA(A,s,B) a PARTICIÓN(A',s').* - Enunciados: SUMA pregunta si existe un subconjunto de A cuyos tamaños (función s) sumen exactamente B. PARTICIÓN pregunta si un conjunto se puede dividir en dos partes de igual suma. - (1) Transformación: sea T = Σ_{a$\in$A} s(a). Se construye A' = A $\cup$ {y, z} con s'(y) = T + B y s'(z) = 2T $-$ B (ambos positivos si 0 $\leq$ B $\leq$ T). La suma total de A' es T + (T+B) + (2T$-$B) = 4T, de modo que cada mitad debe sumar 2T. - (2) Eficiencia: el cálculo de T y la creación de dos elementos es espacio logarítmico. - (3) Implicación directa: si en A existe S con suma B, entonces S $\cup$ {z} suma B + (2T$-$B) = 2T; su complementario (A\S) $\cup$ {y} suma (T$-$B) + (T+B) = 2T. Hay partición. - (4) Implicación recíproca: en cualquier partición de A', los elementos y y z no pueden caer en la misma parte (sumarían 3T$-$... > 2T); por tanto quedan separados. La parte que contiene z debe completar 2T $-$ (2T$-$B) = B con elementos de A; ese subconjunto resuelve SUMA. - Ejemplo: A={3,1,2}, B=3. T=6, y=9, z=9. A'={3,1,2,9,9}, total 24, mitad 12. La partición {3,9} y {1,2,9} corresponde al subconjunto {3} (suma 3) de SUMA.

Ejercicio 4. Grafos — *Complejidad en espacio del problema de caminos en grafos dirigidos.* - Pertenece a NL (espacio logarítmico no determinista). Algoritmo: partiendo del nodo origen, se adivina de forma no determinista el siguiente nodo del camino, guardando únicamente el nodo actual y un contador del número de pasos (a lo sumo n, es decir log n bits). Se acepta si se alcanza el destino en $\leq$ n pasos. El espacio es logarítmico. En versión determinista pertenece a P (recorrido en anchura/profundidad).

Ejercicio 5. Simulaciones — *MT con cinta ilimitada en ambas direcciones mediante MT con cinta ilimitada solo por la derecha.* - Organización: se pliega la cinta bi-infinita por la casilla 0 y se representa con 2 pistas: la pista superior almacena las posiciones 0, 1, 2, . . . y la inferior las posiciones -1, -2, -3, ... La cinta resultante es semiinfinita (tope a la izquierda en la casilla 0). - Simulación: la MT marca en qué pista trabaja; un movimiento a la derecha en la cinta original avanza en la pista superior (o retrocede en la inferior, según la zona); al cruzar la casilla 0 se cambia de pista. El control finito duplica los estados (uno por pista). - Complejidad: cada paso de la original es un paso (o un cambio de pista) en la simulación, luego el orden temporal es el mismo, O(t).

5.1.2 12 de julio de 2021

Ejercicio 1. Decidibilidad

1. *Dada M y w, ¿el número de estados de M es $\leq$ |w|?*

  - Tipo: propiedad sintáctica de $\langle M \rangle$, sin ejecutar M.
  - Clasificación: DECIDIBLE.
  - Desarrollo: se cuentan los estados que aparecen en la codificación $\langle M \rangle$ y se compara con |w|. Es una comprobación finita y directa que no requiere simular la máquina.
2. *¿M acepta al menos 3 palabras distintas?*
  - Tipo: propiedad del lenguaje (cardinalidad ≥ 3), de tipo existencial.
  - Clasificación: SEMIDECIDIBLE, no decidible.
  - Desarrollo: no decidible por Rice (la propiedad "tener al menos 3 palabras" es no trivial: hay lenguajes que la cumplen y otros que no). Semidecidible mediante dovetailing: se simulan en anchura todas las palabras; en cuanto se acumulan 3 aceptaciones distintas, se acepta. Si el lenguaje tiene menos de 3 palabras, la máquina no se detiene.
3. *¿M acepta exactamente 3 palabras distintas?*
  - Tipo: conjunción de "al menos 3" (existencial, r.e.) y "como mucho 3" (universal sobre el resto, co-r.e.).
  - Clasificación: NO SEMIDECIDIBLE.
  - Desarrollo: "como mucho 3" exige garantizar que ninguna otra palabra es aceptada, lo cual no es r.e. (es una condición universal sobre infinitas palabras, emparentada con el complementario del problema de no vacuidad). La intersección de un r.e. con un problema que no es r.e. no es r.e. Se prueba reduciendo un problema no semidecidible (del tipo EMPTY ampliado): se construye una máquina que acepta 3 palabras testigo fijas y, además, simula M para añadir aceptaciones; el resultado tiene exactamente 3 palabras si y solo si M no acepta ninguna, que es EMPTY.
4. *¿M acepta w sin usar más de 2|w| casillas?*
  - Tipo: cota finita de espacio.
  - Clasificación: DECIDIBLE.
  - Desarrollo: con el espacio limitado a 2|w| casillas, el número de configuraciones posibles es finito (acotado por $|Q| \cdot (2|w|) \cdot |\Gamma|^{2|w|}$). Se simula M sobre w respetando ese límite: si acepta dentro del espacio, SÍ; si intenta usar la casilla 2|w|+1, NO; si el número de pasos supera la cota de configuraciones sin aceptar, hay ciclo y se responde NO. Siempre termina.

Ejercicio 2. Verdadero/Falso

1. *La clase L es distinta de PESPACIO.* VERDADERO. Por el Teorema de la Jerarquía de Espacio, $L \subsetneq PESPACIO$ es una inclusión estricta demostrada.
2. *El problema SAT para cláusulas Horn es NP-completo.* FALSO. Horn-SAT se resuelve en tiempo polinómico por propagación unitaria; pertenece a P.
3. *Si* $P1 \propto P2$ *y* P2 *es semidecidible, entonces* P1 *lo es.* VERDADERO. Componiendo el algoritmo de transformación con el semidecisor de P2 se obtiene un semidecisor de P1.
4. *Si un problema está en NL, se resuelve en tiempo polinómico determinista.* VERDADERO. $NL \subseteq P$.
5. *La computación cuántica pone en duda la tesis de Church-Turing.* FALSO. Afecta a la eficiencia (posibles aceleraciones), no a lo que es computable; todo lo cuánticamente calculable lo es por una MT.

Ejercicio 3. Clase NP — *Definición con certificados y verificadores, con ejemplo.* - Definición: un problema pertenece a NP si existe una relación R(x,y) verificable en tiempo polinómico y un polinomio p tales que x es instancia positiva si y solo si existe un certificado y con $|y| \leq p(|x|)$ y R(x,y) cierto. El verificador es una MT determinista que, dados x e y, comprueba R en tiempo polinómico. - Ejemplo (SAT): la entrada x es una fórmula; el certificado y es una asignación de verdad de sus variables; el verificador sustituye y evalúa todas las cláusulas en tiempo polinómico.

Si alguna asignación la satisface, existe certificado; en caso contrario, ninguno funciona.

Ejercicio 4. Decidibilidad (P y su contrario) — *Relaciones entre* `P(x)` *y* `C-P(x)`. - Resultados: (i) si `P` es decidible, `C-P` es decidible (se invierte la salida del decisor). (ii) Si `P` y `C-P` son ambos semidecidibles, entonces `P` (y `C-P`) son decidibles: se ejecutan en paralelo los dos semidecisores y, como exactamente uno se detiene, se obtiene siempre una respuesta. (iii) Si `P` es semidecidible pero no decidible, entonces `C-P` no es semidecidible (de lo contrario, por (ii), `P` sería decidible). Conviene ilustrarlo con UNIVERSAL (semidecidible) y C-UNIVERSAL (no semidecidible).

Ejercicio 5. Simulaciones — *MT multicinta mediante MT de una sola cinta y relación temporal.* - Organización: una sola cinta con multipista; se dedican dos pistas por cada cinta original (una para el contenido y otra para marcar la posición de su cabezal). - Simulación de un paso: la MT recorre toda la zona usada de izquierda a derecha para leer los símbolos bajo todos los cabezales, decide la transición conjunta, y vuelve a recorrer para escribir y mover las marcas de cabezal. - Complejidad: si la multicinta ejecuta `t(n)` pasos, la zona usada tiene longitud `O(t(n))`; cada paso simulado cuesta `O(t(n))` recorridos, luego el total es $O(t(n)^2)$.

5.2 2022

5.2.1 27 de junio de 2022

Ejercicio 1. Decidibilidad

1. *¿*`u` *es subcadena de* $\langle$`M`$\rangle$*?*
   - Tipo: propiedad sintáctica de la codificación.
   - Clasificación: DECIDIBLE.
   - Desarrollo: $\langle$`M`$\rangle$ es una cadena finita; se aplica un algoritmo de búsqueda de subcadena. No requiere ejecutar `M`. Termina siempre.
2. *¿*`M` *NO acepta la palabra* `011`*?*
   - Tipo: comportamiento sobre entrada fija (`w=011`).
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: es C-UNIVERSAL con `w=011`. UNIVERSAL es r.e. no recursivo; su complementario no es r.e. (Teorema de los complementarios).
3. *¿*`L(M)` *es exactamente el conjunto de palíndromos sobre* `{0,1}`*?*
   - Tipo: igualdad del lenguaje con un lenguaje fijo; propiedad del lenguaje.
   - Clasificación: NO DECIDIBLE (Rice) y, además, NO SEMIDECIDIBLE.
   - Desarrollo: por Rice es indecidible (propiedad no trivial de `L(M)`). No es semidecidible porque `L(M) = PAL` exige `PAL` $\subseteq$ `L(M)` (universal sobre infinitas palabras) y `L(M)` $\subseteq$ `PAL`. La parte `PAL` $\subseteq$ `L(M)` se reduce desde un problema no semidecidible del tipo "acepta todas" (usando el gadget de simulación durante `|x|` pasos restringido a palíndromos): así se transforma C-UNIVERSAL en esta igualdad.
4. *¿*`M` *acepta alguna palabra de la forma* `0^{2n}`*?*
   - Tipo: propiedad del lenguaje, existencial.
   - Clasificación: SEMIDECIDIBLE, no decidible.
   - Desarrollo: no decidible por Rice. Semidecidible por dovetailing sobre las palabras `0^0, 0^2, 0^4, . . .`: se simulan en anchura y se acepta en cuanto una es aceptada. Si ninguna lo es, no se detiene.

Ejercicio 2. Verdadero/Falso

1. *Si una MT escribe en su primera cinta, su complejidad en espacio es* `O(n)`. FALSO. El

espacio depende de cuántas casillas distintas visite o escriba, no del hecho de escribir; puede ser sublineal (logarítmica) o superlineal según el algoritmo.

2. *Si un problema de optimización tiene umbral de aproximación 2, entonces tiene un algoritmo δ-aproximado polinómico con δ=2.* FALSO. El umbral es el ínfimo de las razones alcanzables; puede no alcanzarse con igualdad, de modo que existan algoritmos con δ>2 pero no exactamente 2.
3. *La composición de dos algoritmos espacio-logarítmicos es espacio-logarítmica.* VERDADERO. No se almacena la salida intermedia y=ALG1(x): cuando ALG2 necesita el bit i de y, se relanza ALG1(x) y se cuenta hasta ese bit. El espacio adicional es un índice logarítmico.
4. *Si* P1$\propto$P2$\propto$P3 *y* P1 *es semidecidible, entonces* P3 *lo es.* FALSO. La semidecidibilidad se transmite del destino al origen, no al revés: de P1$\propto$P2 y P2 semidecidible se deduce P1 semidecidible, pero P1 semidecidible no dice nada de P3.
5. *Si* NL=NP*, entonces* NP=coNP. VERDADERO. NL es cerrada bajo complementario (Immerman–Szelepcsényi); si NL=NP, esa propiedad se traslada a NP, luego NP=coNP.

Ejercicio 3. NP-completitud — *Reducción de 3-SAT(V,C) a Cubrimiento por Vértices VC(G,K), con ejemplo.* - (1) Transformación: por cada variable xi se crea un gadget de 2 vértices unidos por una arista (xi, ¬xi). Por cada cláusula de 3 literales se crea un triángulo de 3 vértices (gadget de cláusula). Se conecta cada vértice del triángulo con el vértice del literal correspondiente del gadget de variable. Se fija K = |V| + 2|C| (un vértice por variable más dos por cláusula). - (2) Eficiencia: la construcción del grafo es espacio logarítmico (recorre variables y cláusulas). - (3) Directa: dada una asignación que satisface la fórmula, se toma de cada gadget de variable el vértice del literal FALSO, y de cada triángulo los 2 vértices cuyos literales no satisfacen (dejando fuera uno que sí satisface). Esos |V| + 2|C| vértices cubren todas las aristas. - (4) Recíproca: un cubrimiento de tamaño K = |V| + 2|C| debe usar exactamente 1 vértice por gadget de variable (cubrir su arista) y 2 por triángulo (cubrir las 3 aristas internas). El literal no elegido en cada variable define una asignación; las aristas de conexión obligan a que cada cláusula tenga al menos un literal verdadero, luego la fórmula es satisfacible. - Ejemplo: (x1 $\vee$ x2 $\vee$ x3). Gadgets de variable {x1,¬x1}, {x2,¬x2}, {x3,¬x3}; un triángulo para la cláusula; K = 3 + 2 = 5. La asignación x1=V cubre con el vértice ¬x1, etc.

Ejercicio 4. Existencia de caminos — *Complejidad en espacio determinista, con justificación.* - Pertenece a NL en versión no determinista (adivina el camino, guarda nodo actual y contador, espacio logarítmico). En determinista se ubica en su clase logarítmica/polinómica y, en todo caso, en P. Por el Teorema de Savitch, el problema (no determinista en espacio log n) se resuelve en espacio determinista O($\log^2 n$).

Ejercicio 5. Church-Turing — *Simular una MT con un programa con variables; detalle de la transición.* - Organización: tres variables X (izquierda del cabezal, almacenada invertida para tener el símbolo más cercano accesible), Z (símbolo bajo el cabezal) e Y (derecha del cabezal). Un bloque de etiquetas por estado. - Transición δ(qi,aj)=(qm,ak,D): estando en la etiqueta de qi y con Z=aj, se escribe ak y se añade a X (el cabezal avanza, lo escrito queda a la izquierda), se toma el primer símbolo de Y y se coloca en Z (si Y está vacío, Z recibe el blanco), y se salta a la etiqueta de qm. Un movimiento a la izquierda es simétrico (se traslada de X a Z).

Ejercicio 6. Reducciones — *Diferencias entre reducibilidad Turing y logarítmica; cuál es más fuerte.* - La reducción logarítmica (Karp) transforma una instancia de P1 en una de P2 con una única llamada y devuelve directamente la respuesta de P2; preserva las clases NP, coNP, etc. - La reducción Turing resuelve P1 en tiempo polinómico empleando un oráculo de P2 al que puede llamar varias veces y combinar las respuestas con lógica adicional. - La logarítmica es más fuerte (más restrictiva): toda reducción logarítmica es una reducción Turing, pero no al revés. La de Turing es más permisiva.

5.2.2 15 de julio de 2022 (Extraordinario)

Ejercicio 1. Decidibilidad

1. *Dada* `M`*,* `u` *y* `n` *(en binario), ¿*`M` *acepta* `u` *usando* `n` *o más casillas?*
   - Tipo: existencia de un cálculo aceptador con consumo de espacio $\geq$ `n`; no hay cota superior.
   - Clasificación: SEMIDECIDIBLE, no decidible.
   - Desarrollo: se simula `M` sobre `u`; si `M` acepta y, en esa computación, llegó a usar al menos `n` casillas, se acepta. La aceptación es semidecidible; al detenerse aceptando se conoce el espacio usado. Si `M` no acepta `u`, la simulación no se detiene. No es decidible porque el caso negativo (no acepta) no se puede confirmar.
2. *Igual, pero usando* `n` *o menos casillas.*
   - Tipo: cota finita de espacio ($\leq$ `n`).
   - Clasificación: DECIDIBLE.
   - Desarrollo: con el espacio limitado a `n`, las configuraciones son finitas; se simula con detección de ciclo. Si acepta dentro de `n` casillas, SÍ; si excede `n` casillas o entra en ciclo sin aceptar, NO.
3. *¿Existe un palíndromo que NO sea aceptado por* `M`*?*
   - Tipo: existencia de una palabra (palíndromo) en el complementario del lenguaje.
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: "no aceptada" remite al complementario, que no es r.e. Para confirmar que un palíndromo concreto no es aceptado habría que descartar la no parada, lo cual no es semidecidible. Se reduce C-UNIVERSAL: a partir de `(M,w)` se construye una máquina que sobre un palíndromo testigo simula `M` sobre `w`, de modo que "existe palíndromo no aceptado" equivale a "M no acepta w".
4. *Dada* `M` *y* `u`*, ¿en el cálculo sobre* `u` *la MT nunca se mueve a la izquierda?*
   - Tipo: comportamiento sobre una entrada fija, sin movimientos a la izquierda.
   - Clasificación: DECIDIBLE.
   - Desarrollo: si la MT nunca se mueve a la izquierda, el cabezal solo avanza o se mantiene; nunca revisita casillas escritas. Tras pasar la entrada `u` lee blancos, y el comportamiento sobre blancos depende solo del estado (finitos). Se simula: si en algún paso se mueve a la izquierda, NO; si el cabezal avanza indefinidamente o repite un par (estado, símbolo bajo cabezal) en la zona de blancos sin moverse a la izquierda, se concluye que nunca lo hará, SÍ. El proceso termina.

Ejercicio 2. Verdadero/Falso

1. *Una MT multicinta se puede simular con una MT multipista con la misma complejidad en tiempo.* VERDADERO. La multipista es una sola cinta cuyo alfabeto agrupa los símbolos de todas las pistas; un acceso simultáneo a las pistas es un único paso, luego se mantiene el orden temporal.
2. *Si un problema se resuelve con complejidad* `f(n)`*, también con* `0.25·f(n)+n`. VERDADERO. Por el teorema de aceleración lineal (compresión de cinta mediante un alfabeto ampliado), se puede reducir el factor constante; el término `+n` cubre la lectura inicial de la entrada.
3. *Si SAT se resolviera en tiempo polinómico, FSAT también.* VERDADERO. Por autorreducibilidad: se fija `x1=V` y se consulta SAT sobre la fórmula resultante; si sigue siendo satisfacible se mantiene, si no `x1=F`; se repite variable a variable, con un número polinómico de llamadas, hasta construir una asignación.

4. *Si* P1$\propto$P2 *en espacio logarítmico y* P1 *es NP-completo, entonces* P2 *también.* FALSO. De ello se deduce que P2 es NP-difícil, pero para ser NP-completo debe pertenecer a NP, lo que no se afirma.
5. *Si una MT se mueve a la izquierda en la cinta de salida, su complejidad en espacio es al menos* O(n). VERDADERO. La cinta de salida solo deja de contabilizarse si se escribe en una única dirección sin retroceder; al moverse a la izquierda en ella, deja de ser de solo escritura y pasa a contar como espacio de trabajo, que para producir una salida de tamaño O(n) es al menos lineal.

Ejercicio 3. Complejidad — *Problema de las parejas: enunciado y complejidad.* - Enunciado: dado un grafo bipartito (o dos conjuntos con compatibilidades), determinar si existe un emparejamiento perfecto que asocie todos los elementos uno a uno. - Complejidad: pertenece a P. Se reduce en tiempo polinómico al problema de Flujo Máximo (fuente conectada a un lado, sumidero al otro, capacidades 1) y se resuelve con Ford–Fulkerson; un flujo entero máximo igual al número de elementos equivale a un emparejamiento perfecto.

Ejercicio 4. Decidibilidad — *Correspondencias de Post: descripción y decidibilidad.* - Enunciado: dadas dos listas de palabras (α1,. . . ,αk) y (β1,. . . ,βk) sobre un alfabeto, determinar si existe una secuencia de índices i1,. . . ,im tal que αi1. . . αim = βi1. . . βim. - Decidibilidad: indecidible cuando el alfabeto tiene al menos 2 símbolos (se demuestra reduciendo el problema de la palabra/parada de las MT). Semidecidible en general: se enumeran en anchura todas las secuencias de índices y se acepta si alguna iguala las concatenaciones. Con alfabeto de un solo símbolo {1} el problema se reduce a una condición aritmética y es decidible.

Ejercicio 5. Church-Turing — *Programa con variables para* V$\leftarrow$V+1 *sobre* {a,b}. - Se interpreta la palabra como un número en base 2 con dígitos a=0, b=1, leído de modo que la operación sea sumar 1 con acarreo. Recorriendo la palabra desde el dígito menos significativo: mientras el dígito sea b (1) se cambia a a (0) y se propaga el acarreo; al encontrar el primer a (0) se cambia a b (1) y se detiene; si todos eran b, se añade un b al final. Equivale a calcular la palabra siguiente en orden lexicográfico.

Ejercicio 6. Problemas — *Red Feliz: enunciado y clase de complejidad.* - Enunciado: en una red de nodos con estados, cada nodo es "feliz" o "infeliz" según una condición local respecto a sus vecinos; se busca una configuración estable (todos felices o sin posibilidad de mejora local). - Clase: TFNP (funciones totales), concretamente del tipo búsqueda local (PLS). El equilibrio existe siempre, por lo que la decisión es trivialmente "sí"; el algoritmo cambia el estado de un nodo infeliz de forma iterativa y, como cada cambio mejora una función potencial acotada, converge. Encontrar el equilibrio puede no ser polinómico.

5.3 2023

5.3.1 12 de junio de 2023

Ejercicio 1. Decidibilidad

1. *PCP con alfabeto* A={1}.
   - Tipo: caso particular del Problema de Correspondencias de Post.
   - Clasificación: DECIDIBLE.
   - Desarrollo: con un único símbolo, cada palabra αi, βi se reduce a su longitud; la existencia de una secuencia se convierte en una condición lineal entera (existencia de coeficientes no negativos que igualen sumas de longitudes), que es decidible. Con

alfabeto de ≥2 símbolos sería indecidible.

2. *¿La complejidad del lenguaje aceptado por* `M` *es polinómica?*
   - Tipo: propiedad del lenguaje aceptado.
   - Clasificación: NO DECIDIBLE (Rice); no semidecidible.
   - Desarrollo: por Rice es indecidible (propiedad no trivial de `L(M)`). No es semidecidible: afirmar que la complejidad es polinómica cuantifica sobre el comportamiento en infinitas entradas y no admite confirmación finita; se reduce un problema no r.e.
3. *Dado un programa Post-Turing* `P`*, ¿*`P` *NO acepta la palabra vacía?*
   - Tipo: comportamiento sobre la entrada fija λ.
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: los programas Post-Turing son equivalentes a las MT, por lo que "P no acepta λ" es C-UNIVERSAL con `w=`λ, que no es r.e.
4. *Dadas* `M1` *y* `M2`*, ¿*$\langle$`M1`$\rangle$ *es subcadena de* $\langle$`M2`$\rangle$*?*
   - Tipo: propiedad sintáctica de las codificaciones.
   - Clasificación: DECIDIBLE.
   - Desarrollo: ambas codificaciones son cadenas finitas; se aplica búsqueda de subcadena. Termina siempre.

Ejercicio 2. Verdadero/Falso

1. *Si existe un algoritmo δ-aproximado polinómico con δ=2, también con δ=3.* VERDADERO. Una razón de aproximación mayor es una cota más laxa; el mismo algoritmo, que garantiza estar dentro del factor 2, también está dentro del factor 3.
2. *Caracterización de coNP: una MT no determinista tal que para "SÍ" todos los cálculos aceptan y para "NO" al menos uno rechaza.* VERDADERO. Es la definición de coNP por ramas universales (aceptación si y solo si todas las ramas aceptan).
3. *Si* `P1`$\propto$`P2`$\propto$`P3` *y* `P3` *no es semidecidible, entonces* `P1` *tampoco.* FALSO. La no semidecidibilidad se transmite del origen al destino (`P1`$\propto$`P3` con `P1` no r.e. implica `P3` no r.e.), no en sentido contrario; que `P3` sea no r.e. no fuerza a `P1`.
4. *Un problema NP-difícil es siempre NP-completo.* FALSO. NP-completo exige además pertenecer a NP; existen problemas NP-difíciles fuera de NP (por ejemplo, problemas indecidibles o de clases superiores).

Ejercicio 3. SAT — *Relación entre 2-SAT y la búsqueda de caminos en grafos dirigidos.* - Construcción: para una fórmula 2-SAT se define el grafo de implicaciones, con un vértice por cada literal (`x` y `¬x`). Cada cláusula `(a` $\vee$ `b)` genera dos aristas dirigidas: `¬a` $\rightarrow$ `b` y `¬b` $\rightarrow$ `a` (si `a` es falso, `b` debe ser verdadero, y viceversa). - Criterio: la fórmula es insatisfacible si y solo si existe alguna variable `x` con caminos dirigidos `x` $\rightarrow$ `¬x` y `¬x` $\rightarrow$ `x` (es decir, `x` y `¬x` en la misma componente fuertemente conexa). La búsqueda de tales caminos es un problema de alcanzabilidad en grafos dirigidos. - Consecuencia: 2-SAT se resuelve mediante alcanzabilidad, luego pertenece a NL (y a P).

Ejercicio 4. Clases — *Definición de P y EXP, y relación.* - `P` = $\bigcup$`_{j>0} TIEMPO(n`^{j}`)` (tiempo polinómico determinista). `EXP` = $\bigcup$`_{j>0} TIEMPO(2^{n`^{j}`})` (tiempo exponencial determinista). - Relación: `P` $\subseteq$ `EXP` y, por el Teorema de la Jerarquía de Tiempo, la inclusión es estricta: `P` $\subsetneq$ `EXP`.

Ejercicio 5. Reducciones — *Reducción de Conjunto Independiente a Cubrimiento por Vértices (decisión).* - (1) Transformación: se conserva el mismo grafo `G` de `n` vértices; si la pregunta de Conjunto Independiente es "¿existe un independiente de tamaño $\geq$ k?", se traduce a "¿existe un cubrimiento de tamaño $\leq$ n$-$k?". - (2) Eficiencia: la transformación es trivial (solo cambia el umbral), espacio logarítmico. - (3) y (4): `S` es un conjunto independiente si y solo si su complementario `V\S` es un cubrimiento por vértices (toda arista tiene al menos un extremo fuera de `S`, es decir, en `V\S`). Luego existe independiente de tamaño $\geq$ k si y solo si existe cubrimiento de tamaño $\leq$ n$-$k. La

equivalencia es exacta en ambos sentidos.

Ejercicio 6. MT — *Simulación de multicinta a una cinta; relación de pasos* `t(n)`. - Idéntica a la del 12 de julio de 2021, Ejercicio 5: multipista con marcas de cabezal; cada paso simulado cuesta `O(t(n))` y el total es $O(t(n)^2)$.

5.3.2 6 de julio de 2023

Ejercicio 1. Decidibilidad

1. *Dadas* `M1` *y* `M2`*, ¿existe un palíndromo aceptado por* `M1` *y NO aceptado por* `M2`*?*
   - Tipo: existencial sobre `M1`, combinado con "no aceptada" por `M2` (complementario).
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: aunque "existe palíndromo aceptado por `M1`" sería semidecidible, la condición "y no aceptada por `M2`" depende del complementario de `L(M2)`, que no es r.e. Se reduce un problema no r.e. (tipo C-UNIVERSAL/EMPTY) construyendo `M2` para que "no acepta" equivalga a la condición no semidecidible.
2. *Dadas* `M1` *y* `M2`*, ¿existe* `u` *aceptada por* `M1` *tal que* u^{-1} *es aceptada por* `M2`*?*
   - Tipo: existencial con dos condiciones de aceptación.
   - Clasificación: SEMIDECIDIBLE, no decidible.
   - Desarrollo: ambas condiciones son de aceptación (r.e.). Se hace dovetailing: se enumeran las palabras `u`; para cada una se simulan en anchura `M1(u)` y $M2(u^{-1})$; se acepta cuando ambas aceptan. Si no existe tal `u`, no se detiene.
3. *¿*`M` *acepta todas las palabras del alfabeto de entrada?*
   - Tipo: propiedad universal del lenguaje ($L(M)=\Sigma^*$).
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: no decidible por Rice. No es semidecidible: se reduce C-UNIVERSAL con el gadget de "simulación durante `|x|` pasos". Dada `(M,w)`, se construye `M'` que ante `x` simula `M` sobre `w` durante `|x|` pasos y acepta salvo que `M` haya aceptado `w` en ese tramo. Entonces `M` no acepta `w` (caso C-UNIVERSAL positivo) si y solo si `M'` acepta todas las palabras. Como C-UNIVERSAL no es r.e., "acepta todas" tampoco.
4. *¿*`M` *nunca escribe en sus cintas cuando lee la palabra vacía?*
   - Tipo: comportamiento sobre entrada fija (λ).
   - Clasificación: DECIDIBLE.
   - Desarrollo: si `M` no escribe, la cinta permanece toda en blanco; la configuración queda determinada por el estado y la posición del cabezal sobre celdas indistinguibles (blancas). El comportamiento reading-blank en cada estado es determinista, de modo que sin escribir se entra en un ciclo de estados en a lo sumo `|Q|` pasos. Se simula: si en algún paso `M` escribe, NO; si transcurren más de `|Q|` pasos sin escribir, se concluye que nunca escribirá, SÍ. Termina.

Ejercicio 2. Variables numéricas — *Qué son y cómo se simulan con palabras.* - Un programa con variables numéricas opera sobre naturales con instrucciones elementales `A←A+1`, `A←A−1` (con tope en 0) e `IF A≠0 GOTO L`. - Simulación con variables que contienen palabras: se fija un alfabeto y una biyección `N:` $\Sigma^* \to \mathbb{N}$ (orden lexicográfico). Cada variable numérica se representa por la palabra cuya imagen es su valor. `A←A+1` se simula calculando la palabra siguiente; `A←A−1`, la anterior; `IF A≠0` comprueba si la palabra es la que codifica el 0 (la vacía). La equivalencia es exacta porque `N` es biyectiva.

Ejercicio 3. Verdadero/Falso

1. *¿Es posible que un problema de decisión no trivial tenga complejidad en espacio* `O(1)`*?* En general NO de forma significativa: un problema no trivial requiere al menos leer la entrada; con espacio de trabajo constante solo se reconocen lenguajes regulares. Cualquier problema que dependa de la estructura global de la entrada necesita más que `O(1)` de trabajo.
2. *Si un problema está en TFNP, su problema de decisión asociado se resuelve en tiempo lineal.* VERDADERO para la decisión "¿existe solución?", que es trivialmente "sí" (la solución siempre existe); FALSO si se interpreta como encontrar la solución, que puede ser difícil.
3. *Se sabe que la clase L es distinta de PESPACIO.* VERDADERO. `L` $\subsetneq$ `PESPACIO` está demostrado por la jerarquía de espacio.
4. *El Viajante de Comercio tiene un algoritmo δ-aproximado polinómico con δ=2.* VERDADERO solo para la variante métrica/euclídea (por ejemplo, el algoritmo de Christofides da `3/2`, y la cota `2` es alcanzable con árbol generador mínimo). El TSP general no admite aproximación constante salvo que `P=NP`.

Ejercicio 4. NP-completitud — *Reducción de Circuito Hamiltoniano a Viajante de Comercio (decisión).* - (1) Transformación: dado el grafo `G=(V,E)` de Circuito Hamiltoniano, se construye una instancia de TSP con las mismas `n` ciudades y matriz de distancias `d(i,j)=1` si `{i,j}`$\in$`E` y `d(i,j)=2` si no. El umbral de coste es `B=n`. - (2) Eficiencia: construir la matriz es espacio logarítmico. - (3) Directa: si `G` tiene un circuito hamiltoniano, ese mismo recorrido usa `n` aristas de peso `1`, coste total `n` $\leq$ `B`. - (4) Recíproca: una ruta TSP de coste $\leq$ `n` que visita las `n` ciudades usa `n` aristas; como cada una pesa al menos `1`, todas deben pesar exactamente `1`, es decir, todas son aristas de `G`; esa ruta es un circuito hamiltoniano. - Ejemplo: un cuadrilátero `1-2-3-4-1` con aristas presentes da una ruta de coste `4=n`; si faltara una arista, la ruta mínima costaría `5>4`.

Ejercicio 5. Verificadores — *Justificar que PRIMO(n) está en NP.* - El certificado de primalidad de Pratt para `n` consiste en una raíz primitiva `g` módulo `n` y la factorización de `n−1`, junto con certificados recursivos de la primalidad de cada factor primo. El verificador comprueba que `g^{n−1}` $\equiv$ `1 (mod n)` y que `g^{(n−1)/q}` $\not\equiv$ `1 (mod n)` para cada factor primo `q` de `n−1`, todo con exponenciación modular en tiempo polinómico en `log n`. La recursión tiene profundidad logarítmica, luego el certificado es de tamaño polinómico y verificable en tiempo polinómico. Por tanto, PRIMO $\in$ NP.

Ejercicio 6. Complejidad — *Parejas y Acoplamiento por Tripletas (ACTRI): complejidad.* - Parejas (emparejamiento bipartito) pertenece a P (reducción a Flujo Máximo). - ACTRI (acoplamiento por tripletas, la variante con tres conjuntos) es NP-completo: pertenece a NP (un acoplamiento es un certificado verificable) y es NP-difícil (se reduce 3-SAT a ACTRI). El salto de dimensión de 2 a 3 conjuntos cambia la complejidad de P a NP-completo.

5.4 2024

5.4.1 12 de junio de 2024

Ejercicio 1. Decidibilidad

1. *¿Es cierto que siempre que* `M` *acepta* `u` *también acepta* `u⁻¹`*?*
   - Tipo: propiedad del lenguaje (cierre bajo inversión), de forma universal.
   - Clasificación: NO DECIDIBLE (Rice) y NO SEMIDECIDIBLE.
   - Desarrollo: por Rice es indecidible (propiedad no trivial de `L(M)`: hay lenguajes cerrados bajo inversión y otros que no). No es semidecidible porque la condición es universal ("para toda palabra aceptada…"); confirmarla exigiría comprobar infinitas palabras

y descartar no paradas. Se reduce un problema no r.e. (tipo EMPTY): si L(M)=∅, la propiedad se cumple vacuamente; con la construcción adecuada, decidir el cierre equivaldría a decidir un problema no r.e.

2. *¿Toda palabra de longitud ≤100 se acepta en un número de pasos ≤ $|u|^3$?*
   - Tipo: cota de longitud de entrada y cota explícita de pasos.
   - Clasificación: DECIDIBLE.
   - Desarrollo: hay un número finito de palabras de longitud ≤100. Para cada una se simula M durante a lo sumo $|u|^3$ pasos (cota finita) y se comprueba si acepta dentro de ese límite. Conjunto finito de simulaciones finitas, luego termina. La presencia de la cota de pasos es lo que lo hace decidible.
3. *¿La complejidad en tiempo de M NO es n^3?*
   - Tipo: propiedad asintótica del comportamiento.
   - Clasificación: NO DECIDIBLE.
   - Desarrollo: la complejidad en tiempo depende del comportamiento sobre infinitas entradas; es una propiedad no decidible (se reduce desde el problema de la parada: no se puede determinar en general la cota de pasos de una MT arbitraria).
4. *Dada una MT no determinista M y u, ¿todos los caminos terminan en menos de 100 pasos?*
   - Tipo: cota explícita de pasos (100) sobre un árbol de cómputo no determinista.
   - Clasificación: DECIDIBLE.
   - Desarrollo: el árbol de configuraciones hasta profundidad 100 es finito (el factor de ramificación está acotado por el número de transiciones posibles). Se explora por completo y se comprueba que toda rama se detiene en menos de 100 pasos. Termina siempre.

Ejercicio 2. Decidibilidad teórica — *Relaciones entre P(x) y C-P(x).* - Mismos resultados que en el 12 de julio de 2021, Ejercicio 4, demostrados: complementario de decidible es decidible; si ambos son semidecidibles, son decidibles (ejecución en paralelo); si uno es semidecidible no decidible, el otro no es semidecidible.

Ejercicio 3. Verdadero/Falso

1. *Si se tiene una MT multipista, se puede simular con una MT de una sola cinta con el mismo número de pasos.* VERDADERO. La multipista ya es una sola cinta con alfabeto compuesto; el acceso a todas las pistas en una posición es un único paso.
2. *PESPACIO = NPESPACIO.* VERDADERO por el Teorema de Savitch ($NESPACIO(f) \subseteq ESPACIO(f^2)$, y f^2 sigue siendo polinómica si f lo es).
3. *Descomponer un número en factores primos es de la clase FNPT, pero sin algoritmo polinómico conocido.* VERDADERO. La factorización siempre tiene solución (todo número tiene factorización), luego es total; encontrarla no se sabe hacer en tiempo polinómico.
4. *Todo programa con variables numéricas se simula con un programa con variables (palabras) ejecutándose con el mismo número de pasos para cualquier entrada.* FALSO. Ambos modelos son equivalentes en potencia, pero la traducción introduce sobrecarga (cada operación numérica se simula con varias instrucciones sobre palabras); el "mismo número de pasos" no se mantiene.

Ejercicio 4. NP-completitud (a elegir) - Opción A — Partición a Mochila: la Mochila (decisión) pregunta si existe un subconjunto con peso ≤ W y valor ≥ V. Tomando una instancia de Partición con suma total 2T, se define cada objeto con peso = valor = su tamaño, W = T y V = T. Existe partición en dos mitades de suma T si y solo si existe un subconjunto de peso ≤ T y valor ≥ T, es decir, de peso y valor exactamente T. Esto prueba que Mochila es NP-difícil; como está en NP (un subconjunto es certificado), es NP-completa. - Opción B — Cubrimiento por Vértices a Circuito

Hamiltoniano: cada arista {u,v} de G se sustituye por un gadget (dispositivo) de 12 vértices que solo puede atravesarse de tres maneras consistentes (entrar y salir por el lado de u, por el de v, o por ambos). Se añaden k vértices selectores conectados a todos los gadgets. El grafo resultante tiene circuito hamiltoniano si y solo si G tiene un cubrimiento de tamaño k; cada selector "elige" un vértice de la cobertura que recorre sus gadgets incidentes.

Ejercicio 5. Clases — *Clase de complejidad de la existencia de caminos en grafos dirigidos.* - Pertenece a NL. Algoritmo no determinista en espacio logarítmico: se guarda el nodo actual (inicialmente el origen) y un contador de pasos; en cada paso se adivina un vecino y se actualiza el nodo actual; se acepta si se alcanza el destino en $\leq$ n pasos. Solo se almacenan un nodo y un contador, ambos en log n bits.

Ejercicio 6. Problemas — *Red Feliz: enunciado, algoritmo y clase.* - Igual que en el 15 de julio de 2022, Ejercicio 6: TFNP (búsqueda local); algoritmo de voltear nodos infelices con función potencial decreciente que garantiza la convergencia.

5.4.2 8 de julio de 2024

Ejercicio 1. Decidibilidad

1. *¿M acepta todas las palabras de longitud $\leq$100?*
   - Tipo: propiedad universal sobre un conjunto finito de palabras, pero SIN cota de pasos; la condición es "aceptar".
   - Clasificación: SEMIDECIDIBLE, no decidible.
   - Desarrollo: hay un número finito de palabras de longitud $\leq$100, pero "¿M acepta una palabra dada?" es semidecidible (UNIVERSAL), no decidible. La conjunción finita de condiciones r.e. es r.e.: se simulan en anchura M sobre todas las palabras de longitud $\leq$100 y se acepta cuando todas han aceptado. Si alguna no es aceptada (rechazo por parada o ciclo), no se detiene. No es decidible porque ese caso negativo no se puede confirmar; además, por Rice, la propiedad "contener todas las palabras $\leq$100" es no trivial, luego indecidible. (Contraste con el 12 de junio de 2024, Ej.1.2, que sí es decidible por incluir cota de pasos.)
2. *Dado uno de sus estados* q*, ¿M nunca visita* q *sea cual sea la entrada?*
   - Tipo: propiedad universal del comportamiento sobre todas las entradas.
   - Clasificación: NO DECIDIBLE.
   - Desarrollo: "para cualquier entrada nunca alcanza q" cuantifica sobre infinitas entradas y cómputos posiblemente no terminantes. Se reduce el problema de la PARADA/UNIVERSAL: se transforma M para que alcanzar q equivalga a aceptar w, de modo que decidir esta propiedad decidiría un problema indecidible.

Ejercicio 2. Church-Turing — *Simular un programa con variables (palabras) mediante un programa Post-Turing.* - Organización de la información: las variables se concatenan en la cinta separadas por el blanco #, con la forma #V1#V2#. . . #Vn#. El programa Post-Turing mantiene la convención de posición para localizar cada variable. - Simulación de instrucciones básicas: Vj $\leftarrow$ ai Vj (añadir el símbolo ai al principio de Vj) se simula localizando el separador de Vj, desplazando hacia la derecha todo el bloque a partir de ahí para abrir una casilla, y escribiendo ai. Vj $\leftarrow$ Vj$-$ (borrar el último símbolo) localiza el final de Vj y desplaza hacia la izquierda el resto para cerrar la casilla. Las comprobaciones de tipo IF Vj ENDS a se simulan leyendo el último símbolo de Vj.

Ejercicio 3. Verdadero/Falso

1. *Si existiera un algoritmo polinómico para decidir la existencia de circuito hamiltoniano,*

también existiría uno para encontrarlo. VERDADERO. Por autorreducibilidad: se prueba a eliminar cada arista y se consulta el oráculo de decisión; si tras eliminarla el grafo sigue teniendo circuito, se descarta; si no, la arista es necesaria y se conserva. Con un número polinómico de llamadas se reconstruye el circuito.

2. *Una MT con 1 cinta ilimitada en ambas direcciones se simula con una de cinta limitada solo por la derecha con el mismo orden de complejidad en tiempo.* VERDADERO. La técnica de las 2 pistas (plegado por la casilla 0) mantiene el orden `O(t)`.
3. *El problema del isomorfismo de grafos es NP-completo.* FALSO. Está en NP, pero no se ha demostrado que sea NP-completo ni que esté en P; se ubica en la clase GI.
4. *Si* `L2`$\propto$`L1` *y* `L3`$\propto$`L1`*, y* `L1` *es r.e., entonces* `L2`$\cup$`L3` *es r.e.* VERDADERO. Si `L1` es r.e., de `L2`$\propto$`L1` se sigue que `L2` es r.e., e igualmente `L3`; la unión de dos lenguajes r.e. es r.e. (se semidecide ejecutando ambos semidecisores en paralelo).

Ejercicio 4. NP-completitud — *Complejidad de SAT para cláusulas Horn, 2-SAT y NAESAT.* - Horn-SAT: polinómico. Una cláusula Horn tiene a lo sumo un literal positivo; se resuelve por propagación unitaria (se fijan los literales obligados y se propagan hasta saturar o hallar contradicción). - 2-SAT: polinómico (pertenece a NL). Se resuelve con el grafo de implicaciones y la búsqueda de `x` y `¬x` en la misma componente fuertemente conexa. - NAESAT: NP-completo. A pesar de exigir que en cada cláusula no todos los literales sean iguales, sigue siendo tan difícil como SAT (se demuestra por reducción desde 3-SAT con un literal de control).

Ejercicio 5. Jerarquía — *Definir P, NPESPACIO, NP, L, PESPACIO, NL y sus relaciones.* - Definiciones: `L` y `NL` = espacio logarítmico determinista y no determinista; `P` = $\bigcup$`TIEMPO(`n^j`)`; `NP` = tiempo polinómico no determinista; `PESPACIO` = espacio polinómico determinista; `NPESPACIO` = espacio polinómico no determinista. - Relaciones: `L` $\subseteq$ `NL` $\subseteq$ `P` $\subseteq$ `NP` $\subseteq$ `PESPACIO`, y `PESPACIO` = `NPESPACIO` por Savitch. Estrictas demostradas: `L` $\subsetneq$ `PESPACIO` y `NL` $\subsetneq$ `PESPACIO`. El resto de inclusiones se conjeturan estrictas pero no se ha demostrado.

5.5 2025

5.5.1 11 de junio de 2025

Ejercicio 1. Decidibilidad

1. *MT* `M` *de dos cintas y* `|u|`$\leq$`100`*: ¿nunca usa más de 100 casillas en la primera cinta al leer* `u`*?*
   - Tipo: cota finita de espacio (100 casillas) y de entrada.
   - Clasificación: DECIDIBLE.
   - Desarrollo: limitando la primera cinta a 100 casillas (y observando la segunda en consonancia), el número de configuraciones distintas es finito. Se simula `M` sobre `u`: si intenta acceder a la casilla 101 de la primera cinta, se responde NO; si se detiene o repite configuración (ciclo) sin superar las 100 casillas, se responde SÍ. Termina siempre porque las configuraciones acotadas en espacio son finitas.
2. *¿*`M` *acepta todas las palabras de longitud* $\leq$*10?*
   - Tipo: propiedad universal sobre conjunto finito de palabras, SIN cota de pasos.
   - Clasificación: SEMIDECIDIBLE, no decidible.
   - Desarrollo: análogo al 8 de julio de 2024, Ej.1.1. Las palabras de longitud $\leq$10 son finitas, pero la aceptación de cada una es semidecidible. Se semidecide simulando en anchura todas ellas y aceptando cuando todas hayan aceptado; no se detiene si alguna

no es aceptada. No es decidible (por Rice, la propiedad del lenguaje es no trivial; y el caso negativo no es confirmable). Contraste con el apartado 1, que sí es decidible por tener cota de espacio.

3. *Dadas* `M1` *y* `M2`*, ¿toda palabra de longitud* $\leq$*100 aceptada por* `M1` *lo es también por* `M2` *con un número de pasos* $\leq$ *al empleado por* `M1`*?*
   - Tipo: condición universal con aceptación de `M1` (semidecidible) en el antecedente.
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: el complementario sí es semidecidible: para refutar la afirmación basta encontrar una palabra `w` ($\leq$100) que `M1` acepte en `k1` pasos y que `M2` no acepte en $\leq$ `k1` pasos; esto se busca por dovetailing (al aceptar `M1` se conoce `k1` y se simula `M2` durante `k1` pasos). Como el complementario es r.e., la afirmación es co-r.e.; y no es r.e., porque su verdad incluye casos en que `M1` cicla sobre alguna `w` (antecedente falso, no confirmable). Por tanto, no es semidecidible.
4. *Dadas* `M1` *y* `M2` *(con* `M2` *garantizada de terminar para toda entrada) y una palabra* `u`*, ¿*`M1` *da más pasos que* `M2` *al leer* `u`*?*
   - Tipo: comparación de pasos sobre una entrada fija, con `M2` total.
   - Clasificación: DECIDIBLE.
   - Desarrollo: como `M2` siempre termina, se simula `M2` sobre `u` y se obtiene su número exacto de pasos `k2` (finito). Después se simula `M1` sobre `u` durante `k2+1` pasos: si `M1` se detiene en $\leq$ `k2` pasos, no da más que `M2`, NO; si tras `k2+1` pasos `M1` sigue en ejecución, entonces dará al menos `k2+1 > k2` pasos (incluso si cicla indefinidamente), SÍ. La simulación está acotada por `k2+1`, luego siempre termina.

Ejercicio 2. Máquinas de Turing — *Describir una MT que reconozca* `uu` *(*`u`$\in${`0,1`}`*`*) en espacio logarítmico.* - Idea de alto nivel: la entrada `w` tiene longitud par `2m` (si es impar, se rechaza). Hay que comprobar que la primera mitad coincide con la segunda. - Construcción en espacio logarítmico: se usa una cinta de trabajo como contador binario (a lo sumo `log(2m)` casillas). Primero se determina `m` contando símbolos. Después, para cada índice `i` de `1` a `m`, se guardan en binario los índices `i` e `i+m`, se sitúa el cabezal sobre la posición `i` (recorriendo con el contador), se memoriza el símbolo en el control finito, se desplaza a la posición `i+m` y se compara; si difieren, se rechaza. Solo se almacenan índices en espacio logarítmico (no se copia la mitad de la palabra), de modo que el espacio de trabajo es `O(log m)`. No es necesario detallar los estados.

Ejercicio 3. Verdadero/Falso

1. *Una MT ilimitada en ambos lados se simula con una ilimitada a la derecha con los mismos pasos.* FALSO. Se simula con el mismo orden de complejidad `O(t)` mediante 2 pistas, pero no exactamente con el mismo número de pasos (los cambios de pista y la gestión del plegado añaden pasos constantes).
2. *Para que una MT tenga complejidad en espacio* $O(n^2)$ *necesita más de una cinta para no escribir en la cinta de entrada.* VERDADERO. La cinta de entrada no se contabiliza solo si no se sobrescribe; para disponer de espacio de trabajo $O(n^2)$ sin tocar la entrada se requiere al menos una cinta de trabajo separada.
3. *El conocimiento actual es compatible con que* $P \neq NP$ *y* $NP = coNP$*.* VERDADERO. Ninguna de las dos relaciones está demostrada ni refutada; son escenarios posibles con el conocimiento actual.
4. *Está demostrado que la tesis de Church-Turing es cierta.* FALSO. Es una tesis, no un teorema: "efectivamente calculable" no es un concepto matemático formalizable, por lo que no admite demostración.

Ejercicio 4. NP-completitud — *Problema de la SUMA: enunciado y complejidad.* - Enunciado:

dados un conjunto de naturales y un objetivo B, ¿existe un subconjunto cuya suma sea exactamente B? - Complejidad: NP-completo. Pertenece a NP (el subconjunto es un certificado y la suma se comprueba en tiempo polinómico). Es NP-difícil porque problemas NP-completos se reducen a él (3-SAT mediante ACTRI, o directamente desde otras variantes numéricas); de SUMA se reduce además a PARTICIÓN y a MOCHILA.

Ejercicio 5. Complejidad — *Definir EXP y su relación con PESPACIO y P.* - EXP = $\bigcup_{j>0}$ TIEMPO(2^{n^j}) (tiempo exponencial determinista). - Relaciones: P $\subseteq$ PESPACIO $\subseteq$ EXP. Además P $\subsetneq$ EXP (jerarquía de tiempo). PESPACIO $\subseteq$ EXP porque una MT con espacio polinómico p(n) tiene un número de configuraciones acotado por $2^{O(p(n))}$; recorrerlas (o detectar ciclo) cuesta tiempo exponencial. No se sabe si PESPACIO $\subsetneq$ EXP es estricta.

Ejercicio 6. Reducciones (a elegir) — *Reducción Turing de FSAT a SAT, o de FHAMILTONIANO a HAMILTONIANO.* - FSAT a SAT: con un oráculo para SAT (decisión), se construye una asignación satisfactoria. Se comprueba primero que la fórmula es satisfacible; luego, variable a variable, se fija xi=V y se pregunta al oráculo por la fórmula resultante; si sigue siendo satisfacible se mantiene, si no se fija xi=F. Tras n llamadas se obtiene una asignación que satisface la fórmula. Es una reducción Turing (varias llamadas al oráculo, con lógica adicional). - FHAMILTONIANO a HAMILTONIANO: con un oráculo de decisión, se prueban aristas: se elimina una arista y se pregunta si el grafo conserva circuito hamiltoniano; si sí, se descarta; si no, la arista es necesaria y se conserva. Al final quedan exactamente las aristas del circuito.

5.5.2 7 de julio de 2025

Ejercicio 1. Decidibilidad

1. *¿Existe una entrada* u *tal que* M *cicla con* u*?*
   - Tipo: existencial sobre el comportamiento de no parada.
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: "ciclar con u" es no detenerse, propiedad co-r.e. (no r.e.) para una entrada fija. "Existe u que cicla" combina un cuantificador existencial con un predicado no r.e., y no es r.e. Confirmar que una u concreta cicla no es semidecidible; tampoco lo es la existencia. Se reduce el complementario del problema de la parada.
2. *Dadas* M1 *y* M2*, ¿para toda* u*, si* M1 *cicla entonces* M2 *también cicla?*
   - Tipo: condición universal con predicados de no parada.
   - Clasificación: NO SEMIDECIDIBLE.
   - Desarrollo: el "para toda u" junto con "ciclar" (no r.e.) impide la semidecidibilidad. No hay forma de confirmar en tiempo finito que la implicación se cumple para todas las entradas, ya que "M1 cicla" no es confirmable. Se reduce un problema no r.e. (tipo C-UNIVERSAL/no parada).
3. *MT* M *sin ningún movimiento a la izquierda y palabra* u*: ¿*M *acepta* u*?*
   - Tipo: aceptación sobre entrada fija con una restricción estructural (sin movimientos a la izquierda).
   - Clasificación: DECIDIBLE.
   - Desarrollo: sin movimientos a la izquierda, el cabezal solo avanza (o se mantiene) y nunca revisita una casilla escrita. Al pasar la entrada u, lee blancos; la conducta en la zona de blancos depende únicamente del estado, con un número finito de posibilidades. Se simula: si acepta, SÍ; si se detiene sin aceptar, NO; si repite un par (estado, símbolo leído) en blancos sin avanzar hacia la aceptación, se concluye que no aceptará, NO. El proceso termina en un número acotado de pasos.

4. *¿Para ninguna entrada* `M` *se mueve a la izquierda en sus 5 primeros movimientos?*
   - Tipo: propiedad universal sobre las entradas, pero limitada a los 5 primeros pasos.
   - Clasificación: DECIDIBLE.
   - Desarrollo: en 5 movimientos el cabezal solo puede leer las primeras casillas de la entrada; el comportamiento durante esos pasos depende solo de un prefijo de longitud $\leq$5. Hay un número finito de prefijos de longitud $\leq$5 sobre `{0,1}`. Se simulan los 5 primeros pasos para cada prefijo y se comprueba si en algún caso hay un movimiento a la izquierda. Finito, luego decidible.

Ejercicio 2. Church-Turing — *Programa con variables numéricas frente a variables con palabras.* - Un programa con variables numéricas usa naturales (`A←A+1`, `A←A−1`, `IF A≠0 GOTO L`). Un programa con variables que contienen palabras opera sobre cadenas (`A←aA`, `A←A−`, comprobaciones de símbolo). - Equivalencia: mediante la biyección `N:` $\Sigma^* \to \mathbb{N}$ (orden lexicográfico), cada natural se corresponde con una palabra. Incrementar un natural equivale a calcular la palabra siguiente, y decrementar, la anterior. Por ello cualquier cómputo de un modelo se reproduce en el otro; son equivalentes en potencia (aunque no paso a paso).

Ejercicio 3. Verdadero/Falso

1. *El PCP sobre el alfabeto* `{1}` *es semidecidible pero no decidible.* FALSO. Con un solo símbolo, el PCP es decidible (se reduce a una condición aritmética sobre longitudes).
2. *Si una MT tiene una sola cinta, su complejidad en espacio es al menos* `O(n)`*.* FALSO. Si la cinta de entrada no se sobrescribe (o si hay una cinta de trabajo separada), el espacio puede ser sublineal, por ejemplo logarítmico.
3. *Los problemas FNPT se resuelven en tiempo polinómico determinista.* FALSO. La solución existe siempre (totales), pero encontrarla puede no ser polinómico (por ejemplo, la factorización).
4. *Caracterización de coNP: una MT no determinista que, si la respuesta es "NO", rechaza en al menos un cálculo, y si es "SÍ", acepta en todos.* VERDADERO. Es la definición de coNP por ramas universales (aceptación si y solo si todas las ramas aceptan).

Ejercicio 4. NP-completitud — *Cláusulas Horn: enunciado y complejidad.* - Enunciado: una cláusula Horn es una disyunción con a lo sumo un literal positivo (equivalente a una implicación `a1` $\wedge$ `. . .` $\wedge$ `ak` $\to$ `b`). Horn-SAT pregunta si un conjunto de cláusulas Horn es satisfacible. - Complejidad: polinómico. Algoritmo de propagación unitaria: se ponen a verdadero los literales forzados por cláusulas unitarias positivas y se propagan; si se deduce una contradicción (una cláusula sin literal positivo cuyos antecedentes son todos verdaderos), es insatisfacible; en caso contrario, la asignación construida la satisface. Pertenece a P.

Ejercicio 5. Máquinas de Turing — *MT no deterministas frente a deterministas: ventajas y ejemplo.* - Una MT no determinista puede tener varias transiciones posibles por configuración; acepta si existe algún camino de cómputo que acepte. Es equivalente en potencia a la determinista (toda MTND se simula con una MTD explorando su árbol de cómputo), aunque con coste temporal potencialmente exponencial. - Ventaja: simplifica el diseño cuando basta "adivinar" un testigo y verificarlo. Ejemplo: para reconocer `L = {uu}` o palabras compuestas, la MTND adivina el punto de corte (o el factor) y solo verifica; la versión determinista debe probar todas las posibilidades.

Ejercicio 6. Optimización — *Mochila en versión de optimización: complejidad.* - Enunciado: dados objetos con peso y valor y una capacidad `W`, maximizar el valor total sin superar `W`. - Complejidad: NP-difícil (su versión decisión es NP-completa). Admite un algoritmo de programación dinámica pseudopolinómico `O(n·W)` (polinómico en el valor de `W`, no en su longitud). Admite un esquema de aproximación totalmente polinómico (FPTAS), por lo que pertenece a la mejor clase de aproximabilidad.

Capítulo 6

Apéndice. Método de resolución

Reúne los criterios de clasificación, las plantillas de redacción y los errores que con más frecuencia restan calificación. No forma parte del temario: es la sistemática con la que se abordan los ejercicios de las convocatorias.

6.1 Estructura y puntuación del examen

Ej	Tema	Contenido	Puntos 2024-25	Puntos 2022-23
1	T1-T2 Decidibilidad	Clasificar 4 problemas: decidible / semidecidible / no semidecidible	2.4	2.0
2	T2 Church-Turing	Simulación entre modelos o programa a bajo nivel	1.3	2.0
3	Varios	4-5 afirmaciones de Verdadero/Falso razonadas	2.4	1.5
4	T4	SAT / NP-completitud / reducción concreta	1.3	1.5
5	T3	Definición de clases de complejidad y relaciones	1.3	1.5
6	T3-T4-T5-T6	Problema específico (grafos, parejas, Red Feliz, mochila…)	1.3	1.5

Los Ejercicios 1 y 3 suman casi 5 puntos (la mitad del examen).

6.2 Criterio de clasificación de un problema

Pregunta inicial: ¿la propiedad depende del lenguaje `L(M)` (qué palabras acepta) o del comportamiento de la máquina `M` (pasos, casillas, estados, movimientos)?

```
Existe un LÍMITE FINITO explícito de pasos o de espacio (≤N pasos, ≤c casillas)
        -> DECIDIBLE (simulación acotada). No aplicar Rice.

Solo se acota la LONGITUD de la entrada, pero NO los pasos, y la condición es "aceptar"
      -> CUIDADO: "aceptar" es semidecidible. Suele quedar SEMIDECIDIBLE (no decidible),
        no decidible. La finitud del conjunto de entradas no basta si no hay cota de pasos.

Propiedad del LENGUAJE L(M) y NO trivial (acepta palíndromo, L regular, L vacío,
acepta 011, L = {todas}, acepta u^i...)
        -> NO DECIDIBLE por el TEOREMA DE RICE; después se matiza:
           · "¿Existe una palabra...?"           -> SEMIDECIDIBLE
           · "¿Para TODA palabra...?" / "L1⊆L2"  -> NO SEMIDECIDIBLE

Propiedad de la MÁQUINA M sin límite finito (visita el estado q, no se detiene nunca,
cicla para alguna entrada...)
        -> NO DECIDIBLE, no por Rice -> REDUCCIÓN desde PARADA / UNIVERSAL / DIAGONAL.
```

Distinción crítica (origen de la mayoría de errores): "¿acepta toda palabra ≤K en ≤f(K) pasos?" es DECIDIBLE (hay cota de pasos), mientras que "¿acepta toda palabra ≤K?" sin cota de pasos es SEMIDECIDIBLE no decidible.

6.3 Plantillas de redacción

Plantilla 1 — DECIDIBLE por simulación acotada: > "Es decidible. Se construye una MT que simula `M` durante exactamente `N` pasos (o sobre las palabras de un conjunto finito). Como el espacio de búsqueda es finito, la simulación siempre termina y devuelve SÍ o NO. Si solo se acota el espacio (`c` casillas) sin acotar pasos, el número de configuraciones distintas es finito ($\leq |Q| \cdot c \cdot |\Gamma|^c$); si la simulación supera ese número, hay ciclo y se responde en consecuencia."

Plantilla 2 — NO DECIDIBLE por Rice (propiedad del lenguaje): > "Es una propiedad no trivial de los lenguajes r.e.: existe una MT que la cumple (p. ej. una que acepta `{011}`) y otra que no (p. ej. una que acepta $\emptyset$). Por el Teorema de Rice, el problema es indecidible."

Plantilla 3 — SEMIDECIDIBLE por búsqueda en anchura: > "Es semidecidible. Se diseña una MT que enumera las palabras `w0, w1, w2, . . .` y aplica dovetailing: en la etapa `k` simula `k` pasos de `M` sobre `w0, . . . , wk`. Si existe la palabra buscada, en una etapa finita se detecta y se acepta. Si no existe, la máquina no se detiene, lo cual es compatible con la semidecidibilidad." > La búsqueda debe ser en anchura: en profundidad, si una palabra cicla, nunca se probaría la siguiente.

Plantilla 4 — NO SEMIDECIDIBLE por reducción: > "No es semidecidible. Se reduce un problema conocido como no semidecidible (`C-UNIVERSAL`, `DIAGONAL` o `EMPTY`) a este. Se describe el algoritmo `F` que transforma la instancia y se comprueba que la respuesta se conserva en ambos sentidos."

Gadget de "simulación durante |x| pasos" (clave para reducir a propiedades del tipo "acepta todas" o "existe palabra no aceptada"): dada `(M, w)`, se construye `M'` que, ante una entrada `x`, ignora su contenido y simula `M` sobre `w` durante `|x|` pasos; `M'` acepta `x` salvo que en esos `|x|` pasos `M` haya aceptado `w`. Entonces, si `M` nunca acepta `w`, `M'` acepta toda palabra (`L(M')=Σ*`); y si `M` acepta `w` en `k` pasos, `M'` rechaza toda `x` con `|x|≥k`. Este artificio transforma `C-UNIVERSAL` en "¿`L(M')=Σ*`?".

6.4 Banco de afirmaciones de verdadero y falso

Afirmación	V/F	Justificación
PESPACIO = NPESPACIO	V	Teorema de Savitch: NESPACIO(f) $\subseteq$ ESPACIO(f^2).
L $\neq$ PESPACIO	V	L $\subsetneq$ PESPACIO por la jerarquía de espacio.
Se sabe que L = PESPACIO	F	Se sabe que son distintas.
NL $\subseteq$ P (en NL implica tiempo polinómico)	V	Inclusión de la cadena.
NP $\subseteq$ PESPACIO y NP $\subseteq$ EXP	V	Cadena de inclusiones.
Si P=NP, TSP en tiempo polinómico determinista	V	TSP(decisión) es NP-completo.
Si NL = NP, entonces NP = coNP	V	NL cerrada bajo complementario.
El complementario de SAT no tiene algoritmo polinómico (se sabe)	F	No demostrado.
Si A NP-completo y A$\propto$B, entonces B NP-completo	F	Falta B$\in$NP (solo NP-difícil).
NP-difícil es siempre NP-completo	F	Falta pertenecer a NP.
Horn-SAT es NP-completo	F	Es polinómico.
P = EXP	F	P $\subsetneq$ EXP (jerarquía de tiempo).
FNPT se resuelven en tiempo polinómico determinista	F	La existencia está garantizada; el cálculo no.
La tesis de Church-Turing está demostrada	F	Es una tesis.

6.4.1 Tabla de referencia rápida del Ejercicio 1

Si el enunciado indica...	Respuesta inmediata	Justificación
Cota de PASOS ($\leq$N pasos) o de ESPACIO ($\leq$c casillas), aun con entrada acotada	DECIDIBLE	Simulación acotada / configuraciones finitas
Solo cota de LONGITUD de entrada, sin cota de pasos, y condición "aceptar"	SEMIDECIDIBLE, no decidible	Aceptar es r.e.; conjunción finita de r.e. es r.e.
"¿es subcadena de $\langle$M$\rangle$?", número de estados	DECIDIBLE	Propiedad sintáctica de la codificación
Propiedad no trivial de L(M): palíndromo, regular, finito, "acepta 011"	NO DECIDIBLE	Rice
"¿Existe palabra...?", "¿acepta alguna...?"	SEMIDECIDIBLE (no decidible por Rice)	Búsqueda en anchura / MT no determinista
"¿Para TODA palabra...?", "¿acepta todas?", "L1$\subseteq$L2", "L(M)=$\emptyset$"	NO SEMIDECIDIBLE	Reducir EMPTY / C-UNIVERSAL / DIAGONAL

Si el enunciado indica...	Respuesta inmediata	Justificación
"¿M NO acepta w?"	NO SEMIDECIDIBLE	Es C-UNIVERSAL
"¿visita el estado q (cualquier entrada)?", "¿cicla para alguna entrada?", "¿no se detiene nunca?"	NO DECIDIBLE / NO SEMIDECIDIBLE	Reducción desde PARADA / su complementario
"PCP"	Indecidible ($\|A\|\geq 2$) / Decidible ($A=\{1\}$)	Caso particular del alfabeto
"GIC ambigua"	Indecidible. "GIC genera u"	Decidible (CYK)

6.4.2 Errores frecuentes que reducen la calificación

1. Aplicar Rice a propiedades de la máquina (pasos, casillas, estados, movimientos): Rice solo vale para propiedades no triviales de $L(M)$.
2. Confundir "cota de pasos" con "cota de longitud de entrada": con cota de pasos es decidible; sin ella, "acepta todas las palabras $\leq$K" es solo semidecidible.
3. Afirmar que B es NP-completo solo con $A \propto B$: falta probar $B \in NP$.
4. Afirmar que la tesis de Church-Turing está demostrada.
5. Confundir "se sabe que" con "se conjetura": $L=PESPACIO$ es falso (son distintas), $P=NP$ permanece abierto.
6. Omitir la implicación recíproca en las reducciones.
7. Realizar búsqueda en profundidad en las pruebas de semidecidibilidad (debe ser en anchura / dovetailing).

6.4.3 Formulaciones recomendadas para la justificación

– "Por el Teorema de Savitch, $NESPACIO(f) \subseteq ESPACIO(f^2)$, luego $PESPACIO=NPESPACIO$."
– "Por el Teorema de la Jerarquía (de tiempo o de espacio), la inclusión es estricta ($P \subsetneq EXP$, $L \subsetneq PESPACIO$)."
– "Por el Teorema de Rice, toda propiedad no trivial de los lenguajes r.e. es indecidible."
– "Por el Teorema de Cook-Levin, SAT es NP-completo (codifica el cómputo de una MT no determinista como fórmula booleana)."
– "Por el Teorema de los complementarios, si L es r.e. no recursivo, L^- no es r.e."
– En toda reducción se enuncian los cuatro pasos: transformación, eficiencia, implicación directa e implicación recíproca.

Capítulo 7

Bibliografía

Referencias de la guía docente de la asignatura (código 296113D, Grado en Ingeniería Informática, especialidad de Computación y Sistemas Inteligentes, Universidad de Granada).

7.1 Fundamental

- Davis, M. D.; Sigal, R.; Weyuker, E. J. *Computability, Complexity, and Languages*. 2.ª edición. Academic Press, 1994.
- Hopcroft, J. E.; Motwani, R.; Ullman, J. D. *Introducción a la Teoría de Autómatas, Lenguajes y Computación*. 3.ª edición. Addison Wesley, 2010.
- Garey, M. R.; Johnson, D. S. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freeman, 1979.
- MacCormick, J. *What Can Be Computed? A Practical Guide to the Theory of Computation*. Princeton University Press, 2018.
- Moore, C.; Mertens, S. *The Nature of Computation*. Oxford University Press, 2011.
- Papadimitriou, C. H. *Computational Complexity*. Addison Wesley, 1995.

7.2 Complementaria

- Arora, S.; Barak, B. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- Ausiello, G.; Crescenzi, P. et al. *Complexity and Approximation*. Springer-Verlag, 1999.
- Greenlaw, R.; Hoover, H. J.; Ruzzo, W. L. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.
- Sipser, M. *Introduction to the Theory of Computation*. 2.ª edición. Course Technology, 2005.