75.569 · Grafos y Complejidad · 2025-26

PEC3 - Tercera prueba de evaluación continua

Apellidos: *López Henestrosa*
Nombre: José Carlos

Presentación

Esta PEC profundiza en el concepto de complejidad computacional que cubre los contenidos estudiados en los módulos 6 y 7 de la asignatura. Los ejercicios trabajan los conceptos de medida de complejidad, la reducción y completitud, la clase NP-completo y algunos de los problemas intratables más importantes que se conocen.

Competencias

En esta PEC se trabajan las siguientes competencias del Grado de Ingeniería Informática:

- Capacidad para utilizar los fundamentos matemáticos, estadísticos y físicos para comprender los sistemas TIC.
- Capacidad para analizar un problema en el nivel de abstracción adecuada en cada situación y aplicar las habilidades y conocimientos adquiridos para resolverlos.

Objetivos

Los objetivos concretos de esta PEC son:

- Entender los conceptos de intratabilidad y no-determinismo.
- Conocer las diferentes clases de complejidad y saber clasificar los problemas en cada una de estas.
- Entender el concepto de reducción entre problemas y saber demostrar cuando un problema es NP-completo.
- Reconocer problemas intratables que aparecen de forma habitual en informática y en ingeniería.
- Entender y saber aplicar las técnicas básicas de reducción polinómica de los problemas NP-completos.

Respuestas

Ejercicio 1 [25 %]

Definimos el problema X como, dado un grafo $G = (V, A)$ y dos vértices $u, v \in V$, determinar la distancia entre u y v.

Definimos el problema Y como, dado un grafo $G = (V, A)$, dos vértices $u, v \in V$, y un número natural k, determinar si hay un camino en G con, como mucho, k aristas que conecte u y v.

a) **[2 %]** Indica si X es un problema de decisión, de optimización, o ninguna de las dos cosas. Si es un problema de optimización, di también cuál es la función de valoración, y si se minimiza o maximiza.

Es un problema de **optimización**, ya que busca encontrar la mejor solución entre todas las soluciones factibles, donde "mejor" está definido por una función de valoración que se debe, en este caso, minimizar.

La función de valoración es la longitud del camino, medida como el número de aristas que lo componen. Si definimos un camino P como una secuencia de aristas, la función de valoración $f(P)$ es:

2

$$f(P) = |P| \qquad (número\ de\ aristas\ en\ P)$$

b) **[2 %]** Indica si Y es un problema de decisión, de optimización, o ninguna de las dos cosas. Si es un problema de optimización, di también cuál es la función de valoración, y si se minimiza o maximiza.

Es un problema de **decisión**, ya que busca verificar la existencia de un camino que cumpla una condición dada. La respuesta sólo puede ser "Sí, existe" o "No, no existe".

2

Dado que Y no es un problema de optimización, no tiene una función de valoración que se deba minimizar o maximizar como parte de la salida del problema.

c) **[3 %]** ¿Qué algoritmo de los vistos en los módulos usarías para resolver el problema X lo más eficientemente posible? En notación O, ¿qué coste tiene en términos de $|V|$ y $|A|$?

El algoritmo de Dijkstra, ya que encuentra el camino más corto desde un vértice de origen (u) hasta el resto de los vértices de un grafo. El resultado del algoritmo será la distancia mínima hasta el vértice de destino (v).

1

Como no hay pesos se puede usar BFS, que es más eficiente

El coste del algoritmo en notación O es $O((A + V) \log V)$.

d) **[8 %]** Supongamos que tenemos un procedimento P_x que resuelve X: dado un grafo $G = (V, A)$ y dos vértices $u, v \in V$, devuelve $P_x(G, u, v)$, la distancia entre u y v.

Usando P_X, implementa en pseudocódigo un procedimiento P_Y que resuelva Y:

Entrada: grafo $G = (V, A)$; vértices $u, v \in V$; número natural k
Salida: SÍ si hay un camino en G con, como mucho, k aristas que conecta u y v; NO en caso contrario.

algoritmo $P_Y(G, u, v, k)$
    inicio
        $distancia\_minima = P_X(G, u, v)$

        si $distancia\_minima <= k$ **entonces**
            retorno SÍ;
        sino
            retorno NO;
        finsi
    fin

Se valorará que uses el menor número posible de llamadas a P_X por llamada a P_Y .
En el peor de los casos, ¿cuántas se hacen? Usa notación O.

El problema de decisión Y pregunta si la distancia entre u y v es menor o igual a k. Dado que ya tenemos un procedimiento P_X que calcula esta distancia, podemos resolver Y con una única llamada a P_X. Por lo tanto, el coste en términos del número de llamadas a P_X es $O(1)$.

e) **[10 %]** Supongamos que tenemos un procedimento Q_Y que resuelve Y: dado un grafo $G = (V, A)$, dos vértices $u, v \in V$ y un número natural k, devuelve $Q_Y(G, u, v, k)$, que es:

- SÍ si hay un camino en G con, como mucho, k aristas que conecta u y v.
- NO en caso contrario

Usando Q_Y, implementa en pseudocódigo un procedimiento Q_X que resuelva X:

Entrada: grafo $G = (V, A)$; vértices $u, v \in V$
Salida: la distancia entre u y v

algoritmo $Q_X(G, u, v)$
    inicio
        $n = |V|$ // Número de vértices en G

        // Rango de búsqueda inicial: [0, n-1]
        $inicio\_rango = 0$
        $fin\_rango = n - 1$
        $distancia\_optima = \infty$ // Valor por defecto si no hay camino

```
// Verificación rápida: ¿Existe conexión?
si Q_Y(G, u, v, n − 1) == NO entonces
    retorno ∞;
finsi

// Búsqueda binaria para encontrar el k mínimo
mientras inicio_rango <= fin_rango hacer
    k = (inicio_rango + fin_rango) div 2  // División entera

    si Q_Y(G, u, v, k) == SÍ entonces
        // Si es posible con k, intentamos buscar una distancia menor
        distancia_optima = k  // Valor por defecto si no hay camino
        fin_rango = k − 1
    sino
        // Si no es posible con k, la distancia debe ser mayor
        inicio_rango = k + 1
    finsi
finmientras

retorno distancia_optima
fin
```

Se valorará que uses el menor número posible de llamadas a Q_Y por llamada a Q_X. En el peor de los casos, ¿cuántas se hacen? Usa notación O.

10

El número de llamadas necesarias se determina por el tamaño del espacio de búsqueda, que corresponde al rango de posibles distancias entre 0 y $|V| - 1$. Al aplicar una estrategia de búsqueda binaria, dividimos este intervalo de búsqueda por la mitad en cada iteración del bucle. Esto significa que, en lugar de probar cada valor secuencialmente, reducimos el problema exponencialmente paso a paso. En consecuencia, el número de consultas realizadas a Q_Y es proporcional al logaritmo en base 2 del número de vértices del grafo. Por tanto, en el peor de los casos, la complejidad en términos de llamadas es $O(\log V)$. Si hubiésemos utilizado una búsqueda lineal simple, este coste habría ascendido ineficientemente a $O(V)$.

Ejercicio 2 [25 %]

El problema T se define así: dados $N > 0$, un vector de naturales $w = (w_0, \cdots, w_{N-1})$, un vector de naturales $v = (v_0, \cdots, v_{N-1})$, y dos naturales W y V, determinar si existe un subconjunto $S \subseteq \{0, \cdots, N-1\}$ tal que $\Sigma_{i \in S} w_i \leq W$ y $\Sigma_{i \in S} v_i \leq V$.

a) **[3 %]** Identifica T con uno de los problemas vistos en los módulos. Describe del modo más preciso posible su clase de complejidad.

El problema T se identifica con el **problema de la mochila (KNAPSACK)**.

KNAPSACK está en la clase **NP**, ya que dado un subconjunto $S \subseteq \{0, \cdots, N-1\}$, podemos verificar en tiempo polinómico que $\Sigma_{i \in S} w_i \leq W$ y $\Sigma_{i \in S} v_i \leq V$.

Se ha demostrado que KNAPSACK es NP-Difícil, puesto que el problema VERTEX_COVER (un problema NP-Completo conocido) se puede reducir polinómicamente al KNAPSACK (VERTEX_COVER $\leq_p$ KNAPSACK). Esta demostración es técnica, pero confirma que KNAPSACK es al menos tan difícil como cualquier otro problema en NP.

Como el problema KNAPSACK está en NP y es NP-Difícil, se concluye que es **NP-Completo**.

b) **[5 %]** Completa el pseudocódigo del siguiente procedimiento P_T para resolver T:

Pista: la parte que falta se evalúa en tiempo $O(1)$.

Entrada: vector de naturales $w = (w_0, \cdots, w_{N-1})$; vector de naturales $v = (v_0, \cdots, v_{N-1})$; naturales W y V
Salida: SÍ si existe un subconjunto $S \subseteq \{0, \cdots, N-1\}$ tal que $\Sigma_{i \in S} w_i \leq W$ y $\Sigma_{i \in S} v_i \leq V$, NO en caso contrario

```
algoritmo P_T = (w, v, W, V)
    inicio
        para i ← 0 hasta N
            para j ← 0 hasta W
                m[i][j] = 0;
            finpara
        finpara
        para i ← 1 hasta N
            para j ← 0 hasta W
                si w[i − 1] > j
                    entonces m[i][j] ← m[i − 1][j];
                    sino m[i][j] ← máx(m[i − 1][j − w[i − 1]] + v[i − 1], m[i − 1][j]);
                finsi
            finpara
        finpara
```

```
        si m[N][W] ≤ V entonces
            retorno SÍ;
        sino
            retorno NO;
        finsi
    fin
```

c) **[6 %]** El estudiante A afirma que el problema T es NP-Difícil. El estudiante B afirma que el procedimiento P_T tiene coste $O(N \cdot W)$. El estudiante C dice que A y B no pueden tener los dos la razón, porque eso implicaría que $P = NP$. Para cada estudiante di si tiene razón o no, y por qué.

Estudiante A

El problema T es una variación del problema de decisión KNAPSACK, que es un problema NP-Completo. Por definición, cualquier problema NP-Completo es también NP-Difícil, lo que significa que es, al menos, tan difícil como cualquier otro problema en la clase NP. Por lo tanto, **sí tiene razón**.

Estudiante B

El procedimiento P_T utiliza un algoritmo de programación dinámica para llenar una matriz $m[i][j]$ de tamaño $(N + 1) \times (W + 1)$. La matriz tiene $O(NW)$ celdas, por lo que, para calcular el valor de cada celda de la matriz, el algoritmo realiza un número fijo de operaciones que se hacen en tiempo $O(1)$. Por lo tanto, el coste del procedimiento es proporcional al número de celdas, que es $O(NW)$. Como conclusión, **sí tiene razón**.

Estudiante C

Aunque el problema T (KNAPSACK) es NP-Difícil y el procedimiento P_T tiene un coste de $O(NW)$, estas dos afirmaciones son compatibles. La razón es que la complejidad $O(NW)$ es solo pseudopolinómica, no polinómica. La complejidad se considera polinómica si depende únicamente del número de bits necesarios para codificar la entrada. Como el peso límite W forma parte de la entrada y su valor puede ser exponencial respecto al número de bits usados para representarlo, el coste real $O(NW)$ es en realidad exponencial.

Dado que el algoritmo P_T no es de tiempo polinómico, su existencia no obliga a que P sea igual a NP. Por lo tanto, el estudiante **no tiene razón**.

d) **[8 %]** El problema U se define así: dado un multiconjunto de $P > 0$ naturales $M = \{m_0, \cdots, m_{P-1}\}$, y un natural Q, determinar si existe un subconjunto S de M tal que la suma de todos los elementos de S sea igual a Q. Da una reducción polinómica de U a T indicando claramente qué entrada de T devuelve la reducción si se le pasa como argumento una entrada (M, Q) de U. Justifica por qué se trata de una reducción polinómica.

Para realizar la reducción del problema U al problema T, debemos transformar una instancia de U en una de T de modo que la solución de T nos dé la respuesta a U en tiempo polinómico.

Dada una entrada de U (M, Q), donde $M = \{m_0, \cdots, m_{P-1}\}$ y $Q \in \mathbb{N}$, definimos la entrada de T de la siguiente manera:

- Tamaño N: Igual a P (el número de elementos en el multiconjunto M).
- Vector w: Los mismos elementos de M. Es decir, $w_i = m_i$ para $0 \leq N$.
- Vector v: Los mismos elementos de M. Es decir, $v_i = m_i$ para $0 \leq N$.
- Límite W: Igual a Q.
- Límite V: Igual a Q.

6

Falta ver que (M, Q) es positiva para U si y sólo si (w, v, W, V) es positiva para T

Con esto, la entrada para T es (N, w, v, Q, Q).

La reducción es polinómica porque el número de elementos N es igual a P. Solo estamos duplicando el vector M para crear w y v. Esta operación es $O(P)$.

Por otro lados los valores W y V se asignan directamente desde Q, lo cual es $O(1)$ o proporcional al número de bits de Q.

Por último, el tamaño de la nueva instancia es aproximadamente el doble de la original, lo que mantiene una relación lineal $O(n)$ respecto al tamaño de la entrada.

Al ser una transformación lineal en el tamaño de la entrada, queda demostrado que es una reducción en tiempo polinómico.

e) **[3 %]** ¿Existe una reducción (no hace falta darla explícitamente) de T a U? ¿Por qué?

Para demostrar la existencia de la reducción $T \leq_p U$ sin construirla explícitamente, recurrimos a las definiciones de las clases de complejidad NP y NP-Completo:

1. **El problema de destino (U) es NP-Completo**
   El problema Subset Sum (U) es un problema clásico conocido por ser NP-Completo. Por definición, un problema es NP-Completo si pertenece a NP y cualquier otro problema de la clase NP puede reducirse polinómicamente a él.

2. **El problema de origen (T) pertenece a NP**
   Un problema de decisión pertenece a la clase NP si, dada una "candidata a solución" (certificado), podemos verificar su validez en tiempo polinómico. Para el problema T, dado un subconjunto S, el algoritmo de verificación consiste simplemente en sumar los elementos w_i indexados por S y comparar con W y sumar los elementos v_i indexados por S y comparar con V. Ambas operaciones son lineales $O(N)$, por lo tanto, T es NP.

3

De forma precisa, lo que se usa es que U es NP-difícil

Como podemos apreciar, T es NP y U es NP-Completo. Por la propiedad de universalidad de los problemas NP-Completos, podemos demostrar matemáticamente que existe una función computable en tiempo polinómico que transforma instancias de T en instancias de U preservando la respuesta.

Ejercicio 3 [25 %]

a) **[10 %]** Para cada una de las siguientes fórmulas proposicionales, indica si está en forma normal conjuntiva (FNC). Si lo está, indica el número de cláusulas y el número de literales de cada una. Si no lo está, explica brevemente por qué y cómo podría transformarse para que lo estuviera.

1) **[2 %]** $F_1 = (a \vee \neg b) \wedge (b \vee c \vee d) \wedge (\neg a \vee \neg c)$

Sí está en FNC. Tiene 3 cláusulas:

- Cláusula $(a \vee \neg b)$. Tiene 2 literales (a y $\neg b$).
- Cláusula $(b \vee c \vee d)$. Tiene 3 literales (b, c y d)
- Cláusula $(\neg a \vee \neg c)$. Tiene 2 literales ($\neg a$ y $\neg c$).

2) **[2 %]** $F_2 = (a \wedge b) \vee (\neg a \wedge c)$

No está en FNC, ya que el operador principal debería ser una conjunción, no una disyunción. Para convertirla en FNC, primero tenemos que aplicar la ley distributiva para distribuir el primer término $(a \wedge b)$ sobre el segundo $(\neg a \wedge c)$:

$$((a \vee \neg b) \vee \neg a) \wedge ((a \wedge b) \vee c)$$

Aún no está en FNC, por lo que aplicamos la ley distributiva nuevamente dentro de cada bloque:

- Bloque 1: $(a \vee \neg a) \wedge (b \vee \neg a)$. Como $(a \vee \neg a)$ es una tautología, la podemos eliminar, por lo que quedaría $(\neg a \vee b)$.
- Bloque 2: $(a \vee c) \wedge (b \vee c)$.

Como resultado, obtenemos $(\neg a \vee b) \wedge (a \vee c) \wedge (b \vee c)$, el cual está en FNC.

3) **[2 %]** $F_3 = (\neg a \vee b) \wedge \neg(\neg b \vee c)$

No está en FNC. Aunque el operador principal es una conjunción, el segundo término $\neg(\neg b \vee c)$ no cumple con la definición de literal, ya que tiene una negación justo fuera del paréntesis Para convertirla en FNC, primero tenemos que aplicar la ley de De Morgan para "empujar" dicha negación hacia adentro:

$$\neg(\neg b \vee c) \equiv (\neg(\neg b) \wedge \neg c)$$

A continuación, aplicamos la doble negación $\neg(\neg b) \equiv b$. Por lo tanto, el término queda $(b \wedge \neg c)$.

Ahora sustituimos el término original por el simplificado. Dado que el operador resultante es $\wedge$, simplemente se añaden como nuevas cláusulas:

$$(\neg a \vee b) \wedge b \wedge \neg c$$

Ahora sí que quedaría en FNC, con 3 cláusulas resultantes $(\neg a \vee b)$, b y $\neg c$.

4) **[2 %]** $F_4 = (a \leftrightarrow b) \vee (\neg b \wedge c)$

No está en FNC, ya que contiene el operador de doble implicación, el cual no forma parte del conjunto de operadores básicos de la FNC $(\wedge, \vee, \neg)$. Además, el operador principal es una disyunción en lugar de una conjunción.

Para convertirla en FNC, primero tenemos que eliminar el operador $\leftrightarrow$. Sabemos que $a \leftrightarrow b \equiv (a \rightarrow b) \wedge (b \rightarrow a) \equiv (\neg a \vee b) \wedge (\neg b \vee a)$. Por lo tanto, lo sustituimos en la fórmula:

2

$$[(\neg a \vee b) \wedge (\neg b \vee a)] \vee (\neg b \wedge c)$$

A continuación, aplicamos la ley distributiva para distribuir la disyunción sobre la conjunción:

$$[(\neg a \vee b) \vee (\neg b \wedge c)] \wedge [(\neg b \vee a) \vee (\neg b \wedge c)]$$

Ahora volvemos a aplicar la ley distributiva para cada corchete:

- **Corchete 1**: $(\neg a \vee b \vee \neg b) \wedge (\neg a \vee b \vee c)$. Podemos eliminar el primer paréntesis, ya que $(\neg a \vee b \vee \neg b)$ es una tautología, ya que contiene $b \vee \neg b$. Por lo tanto, queda como $(\neg a \vee b \vee c)$.

- **Corchete 2**: $(\neg b \vee a \vee \neg b) \wedge (\neg b \vee a \vee c)$. Simplificamos $(\neg b \vee a \vee \neg b)$ como $(\neg b \vee a)$, por lo que el contenido del corchete queda así: $(\neg b \vee a) \wedge (\neg b \vee a \vee c)$.

  Por la ley de absorción $A \wedge (A \vee B) \equiv A$, el término queda simplemente como $(\neg b \vee a)$.

Por último, unimos el resultado del corchete 1 y el corchete 2:

$$(\neg a \vee b \vee c) \wedge (a \wedge \neg b)$$

Ahora sí que quedaría en FNC, con 2 cláusulas resultantes $(\neg a \vee b \vee)$ y $(a \vee \neg b)$.

5) **[2 %]** $F_5 = (a \vee b) \wedge (c) \wedge (\neg d \vee \neg c \vee b)$

Sí está en FNC. Tiene 3 cláusulas:

2

- Cláusula $(a \vee b)$. Tiene 2 literales (a y b).
- Cláusula (c). Tiene 1 literal (c)
- Cláusula $(\neg d \vee \neg c \vee b)$. Tiene 3 literales ($\neg d$, $\neg c$ y b).

b) **[5 %]** Sea la siguiente fórmula booleana en **Forma Normal Conjuntiva (FNC)**:

$$F = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

1) **[2,5 %]** Transforma F en una fórmula F' equivalente en **Forma Normal Disyuntiva (FND)**. Se valorará que la FND tenga el mínimo número de términos.

Para ello, podemos utilizar las propiedades del álgebra de Boole, como la propiedad distributiva. En primer lugar, asignamos a cada cláusula una notación para identificarlas de una forma más sencilla:

- $C_1 = (x_1 \vee \neg x_2 \vee x_3)$
- $C_2 = (\neg x_1 \vee \neg \vee x_4)$
- $C_3 = (x_2 \vee \neg x_3 \vee x_4)$

Ahora que tenemos este desglose, observamos que las cláusulas C_1 y C_2 comparten los literales x_2 y x_4. Podemos agruparlos para aplicar la propiedad distributiva inversa: $(A \vee B) \wedge (A \vee C) \equiv A \vee (B \wedge C)$.

Para ello, definimos $Z = (x_2 \vee x_4)$. Entonces:

$$C_2 \wedge C_3 = (\neg x_1 \vee Z) \wedge (\neg x_3 \vee Z)$$

Aplicamos la propiedad:

$$C_2 \wedge C_3 = (\neg x_1 \wedge \neg x_3) \vee Z$$

Sustituimos Z:

$$C_2 \wedge C_3 = (\neg x_1 \wedge \neg x_3) \vee x_2 \vee x_4$$

Ahora F se ve así:

$$F = (x_1 \vee \neg x_2 \vee x_3) \wedge [(\neg x_1 \wedge \neg x_3) \vee x_2 \vee x_4]$$

Para pasar a FND, distribuimos el término de la izquierda (C_1) sobre cada uno de los tres componentes del corchete de la derecha.

$$F = [C_1 \wedge (\neg x_1 \wedge \neg x_3)] \vee [C_1 \wedge x_2] \vee [C_1 \wedge x_4]$$

A continuación, analizamos cada bloque por separado:

- **Bloque 1**: $C_1 \wedge (\neg x_1 \wedge \neg x_3)$ es igual a $(x_1 \vee \neg x_2 \vee x_3) \wedge \neg x_1 \wedge \neg x_3$
  Distribuimos $(\neg x_1 \wedge \neg x_3)$ dentro del paréntesis:

- $x_1 \wedge \neg x_1 \wedge \neg x_3 = 0$. Hay una contradicción $(x_1 \wedge \neg x_1)$
- $\neg x_2 \wedge \neg x_1 \wedge \neg x_3$
- $x_3 \wedge \neg x_1 \wedge \neg x_3 = 0$. Hay una contradicción $(x_3 \wedge \neg x_3)$

Como resultado del bloque 1, obtenemos $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$.

- **Bloque 2**: $C_1 \wedge x_2$ es igual a $(x_1 \vee \neg x_2 \vee x_3) \wedge x_2$
  Distribuimos x_2 dentro del paréntesis:
  - $x_1 \wedge x_2$
  - $\neg x_2 \wedge x_2 = 0$. Contradicción.
  - $x_3 \wedge x_2$

  Como resultado del bloque 2, obtenemos $(x_1 \wedge x_2) \vee (x_2 \wedge x_3)$.

- **Bloque 3**: $C_1 \wedge x_4$ es igual a $(x_1 \vee \neg x_2 \vee x_3) \wedge x_4$
  Distribuimos x_2 dentro del paréntesis:
  - $x_1 \wedge x_4$
  - $\neg x_2 \wedge x_4$
  - $x_3 \wedge x_4$

  Como resultado del bloque 3, obtenemos $(x_1 \wedge x_2) \vee (\neg x_2 \wedge x_4) \vee (x_3 \wedge x_4)$.

Por último, unimos los resultados de los tres bloques mediante disyunciones para obtener F':

25

$$F' = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_4) \vee (\neg x_2 \wedge x_4) \vee (x_3 \wedge x_4)$$

Esta fórmula es equivalente a F y se encuentra en FND, ya que es una disyunción de conjunciones de literales.

2) **[2,5 %]** Determina si F es satisfacible. En caso afirmativo, proporciona una asignación de verdad para las variables que satisfagan la fórmula.

Una fórmula es satisfacible si existe al menos una interpretación (asignación de valores de verdad) que la haga verdadera.

Al transformar F a su Forma Normal Disyuntiva (FND), obtuvimos una serie de conjunciones unidas por disyunciones (∨). Cualquiera de esas conjunciones representa una condición suficiente para que F sea verdadera.

Por ejemplo, para que el término más sencillo que obtuvimos en la FND

$(x_1 \wedge x_2)$ sea verdadero, basta con que x_1 sea verdadero y x_2 sea verdadero. El valor de las otras variables (x_3 y x_4) es irrelevante para este término específico (son condiciones *don't care* en este caso), pero debemos asignarles un valor para completar la interpretación formal.

Dicho todo esto, podemos determinar que una asignación válida v que satisface F es la siguiente:

2.5

- $v(x_1) = 1$ (verdadero)
- $v(x_2) = 1$ (verdadero)
- $v(x_3) = 0$ (falso) [elección arbitraria]
- $v(x_4) = 0$ (falso) [elección arbitraria]

Para verificar que la asignación propuesta es válida, sustituimos estos valores en $F = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$:

- **Cláusula 1**: $(1 \vee 0 \vee 0) \equiv 1$ (verdadero)
- **Cláusula 2**: $(0 \vee 1 \vee 0) \equiv 1$ (verdadero)
- **Cláusula 3**: $(1 \vee 1 \vee 0) = 1$ (verdadero)

$$F = 1 \wedge 1 \wedge 1 = 1$$

Por lo tanto, la asignación satisface la fórmula.
$F = 1 \wedge 1$

c) **[10 %]** Se consideran dos problemas numéricos clásicos:

- **Factorización**: dado un entero $N > 1$, encontrar (si existen) dos enteros $p, q > 1$ tales que $N = p \cdot q$.
- **Primalidad**: dado un entero $N > 1$, determinar si N es primo.

1) **[5 %]** ¿Qué tipo de problema son? Indica si cada uno es un **problema de decisión** o de **optimización/construcción** y justifica tu respuesta.

Factorización
Este no es un problema de decisión porque la salida no es un simple SÍ/NO. Tampoco es estrictamente un problema de optimización, ya que no existe una función de coste que estemos intentando minimizar o maximizar (cualquier par de factores p, q válidos es una solución igualmente válida).

La entrada del problema es N y la salida es un objeto matemático complejo (en este caso, la tupla de enteros (p, q)) que satisface una relación específica $(p \cdot q = N)$. Si N es primo, el algoritmo debe indicar que no existe tal objeto (devolver una señal de fallo).

Por lo tanto, es un problema de **construcción**.

Primalidad
Un problema de decisión es aquel que se formula como una pregunta cuya única respuesta posible es SÍ o NO.En el caso de la primalidad, el algoritmo recibe una entrada N y debe devolver un valor de verdad booleano:

5

- Devuelve SÍ si N es primo.
- Devuelve NO si N no es primo.

Por lo tanto, la primalidad es un problema de **decisión**.

2) **[5 %]** En términos de complejidad, ¿en qué se diferencian?

la versión de decisión

Factorización
Es relativamente fácil verificar una solución para este problema. Si le damos p y q, multiplica ambos números para comprobar si da N. Esto implica que sea NP. No obstante, si la factorización fuera NP-Completa, implicaría consecuencias teóricas extrañas, como el colapso de la jerarquía polinómica, algo que la mayoría de teóricos dudan.

Por otro lado, no se sabe si es P, ya que no se ha encontrado ningún algoritmo clásico polinómico que lo resuelva. Se cree que es un problema "difícil". Por ello, se cree que la factorización es más difícil que los problemas en P, pero no tan difícil como los problemas NP-Completos, por lo que podemos catalogarlo como NP-Intermedio.

3

Primalidad
Este problema pertenece a la clase P, puesto que existe un algoritmo determinista que puede determinar si un número N es primo en un tiempo que

crece polinómicamente respecto al número de dígitos de N. Es decir, respecto a $\log N$.

Comparativa

Dada esta información sobre ambos problemas, sabemos que la clase de complejidad de la factorización es NP-Intermedio, sin conocer si está en P, mientras que la de la primalidad es P. Esto significa que la dificultad computacional de la factorización es difícil, mientras que la de primalidad es sencilla. Existen algoritmos, como el algoritmo AKS, que resuelven la primalidad, mientras que hay algoritmos, como el *General Number Field Sieve* que resuelven el problema sub-exponencialmente, no polinómicamente.