Introducción

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Algoritmia

Introducción

Optimización

- Buscar la mejor solución (*solución óptima*) para un determinado problema.
- En computación, la bondad de una solución se mide en términos formales (objetivos).
- Estudiaremos problemas de optimización, así como su resolución mediante estrategias algorítmicas.

Introducción

¿Qué es un algoritmo?

- Serie finita de instrucciones claramente definidas que resuelven un problema particular.
- El concepto de algoritmo es antiguo (matemáticas).
- Actualmente incluye soluciones basadas en programación.

Introducción

¿Por qué estudiar algoritmos?

- Fundamental en cualquier rama de la computación.
- Innovación y desarrollo de nuevas soluciones.
- Desarrollo del pensamiento computacional.
- Entrevistas de trabajo.

La asignatura

Algoritmia y optimización

Profesorado

- **Jorge Calvo:** profesor de teoría y coordinador de la asignatura.
- **Juan Carlos Martínez**: profesor de prácticas.
- Adscripción:
  - Departamento de Lenguajes y Sistemas Informáticos
  - Instituto Universitario de Investigación Informática
  - Grupo de Reconocimiento de Formas e Inteligencia Artificial

Algoritmia y optimización

Contenidos

- Análisis de algoritmos: cálculo de complejidad
- Esquemas algorítmicos:
  - Divide y vencerás
  - Programación dinámica
  - Algoritmos voraces
  - Algoritmos de vuelta atrás
- Programación lineal
- Técnicas heurísticas

Algoritmia y optimización

Evaluación

- La evaluación de la asignatura consiste en:
  - **Exámenes parciales (50 %)**: a realizar durante el cuatrimestre
  - **Examen final (50 %)**: a realizar al final del cuatrimestre
    - **Trabajo opcional**: hasta 1 punto (extra) a contar en el examen.
- Para aprobar:
  - Más de un 5 de nota agregada
  - Al menos un 4 en cada bloque
- Convocatorias extraordinaria: **exámen teórico-práctico** (100 %).

Algoritmia y optimización

Evaluación: exámenes parciales

- Exámenes prácticos a realizar en una sesión de prácticas
- Habrá 4 exámenes parciales:
  - Eficiencia (30 %)
  - Divide y vencerás + programación dinámica (30 %)
  - Algoritmo voraces + algoritmos de vuelta atrás (30 %)
  - Programación lineal (10 %)

Algoritmia y optimización

Evaluación: examen final

- Examen final de enfoque eminentemente teórico sobre los contenidos de la asignatura.

Se podrá presentar un trabajo optativo en la última sesión de teoría.
Puede sumar hasta 1 punto en el examen de teoría de la convocatoria ordinaria.

Funcionamiento

Funcionamiento

Información de consulta

- Plataforma vehicular: Moodle
- Información de interés: guía docente

Clases de teoría

Funcionamiento

- Modalidad “clase magistral”
- Ejercicios en clase
- Material:
  - Apuntes de la asignatura
  - Transparencias

Clases de prácticas

Funcionamiento

- Sesiones de prácticas: consolidar los conceptos teóricos, practicar la resolución de problemas y resolver dudas.
- Cuadernos de prácticas: ejercicio autónomo
  - No se entregan (no se evalúan)
  - Preparan para el parcial (ejercicios similares)
- Lenguaje de programación vehicular: Python

Planificación del cuatrimestre

Calendario (I)

Semana	Teoría	Práctica
1 (09/09)	Introducción	-
2 (16/09)	Complejidad	Complejidad
3 (23/09)	Complejidad	Complejidad
4 (30/10)	Divide y vencerás	Complejidad
5 (7/10)	Divide y vencerás	**PARCIAL (30 %)**
6 (14/10)	Programación dinámica	Divide y vencerás
7 (21/10)	Programación dinámica	Programación dinámica

Planificación del cuatrimestre

Calendario (II)

Semana	Teoría	Práctica
8 (28/10)	Voraz	**PARCIAL (30 %)**
9 (4/11)	Vuelta atrás	Voraz
10 (11/11)	Vuelta atrás	Vuelta atrás
11 (18/11)	Vuelta atrás	Vuelta atrás
12 (25/11)	Programación lineal	**PARCIAL (30 %)**
13 (2/12)	Heurísticas	Programación lineal
14 (9/12)	Repaso	**PARCIAL (10 %)**
15 (16/12)	Trabajos	-

Introducción

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Análisis de algoritmos

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Análisis de algoritmos

Estudio de eficiencia

- El **análisis de algoritmos** permite evaluar y comparar algoritmos.
- Suele centrarse en medir la **eficiencia** en el uso de recursos:
  - Identificar si un algoritmo es eficiente.
  - Identificar si un algoritmo es más eficiente que otro.

Análisis de algoritmos

Complejidad

- Cuando hablamos de los recursos que consume un algoritmo, nos referimos a su **complejidad**.
- Un algoritmo es más complejo (menos eficiente) si consume más recursos.
- Recursos habituales a considerar:
  - Tiempo de ejecución (complejidad temporal).
  - Consumo de memoria (complejidad espacial).

Análisis de algoritmos

Tipos

- La complejidad puede verse afectada por factores tanto externos como internos.
  - **Externos**: potencia del hardware, el compilador o interprete, los datos de entrada.
  - **Internos**: número y tipo de instrucciones.
- Distinguiremos entre **análisis empírico** y **análisis teórico**.

Análisis de algoritmos

Análisis empírico

- **Ejecutar el algoritmo** y medir los recursos consumidos.
  - Por ejemplo, si nos referimos a recursos temporales, podemos cronometrar el tiempo de ejecución.
- **Ventaja**: medida real del comportamiento del algoritmo en un entorno concreto.
- **Desventaja**: puede verse afectado por cuestiones extrínsecas al propio algoritmo.

Análisis de algoritmos

Análisis teórico

- Consiste en obtener una **función matemática** que represente la complejidad del algoritmo.
- **Ventaja**: no es necesario ejecutar el algoritmo y el resultado depende exclusivamente del diseño del mismo.
- **Desventaja**: es difícil trasladar esta función a términos prácticos de una ejecución en un entorno real.

Complejidad teórica

Complejidad teórica

Nociones generales

- ¿Qué complejidad tienen los siguientes algoritmos?
  - Se suele considerar que el coste de las operaciones elementales es unitario.

```
función suma_uno(número)
    valor := número + 1
    devuelve valor
```

```
función suma_uno(número)
    devuelve número + 1
```

Complejidad teórica

Nociones generales

- ¿Qué complejidad tiene el siguiente algoritmo?

```
función acumulado(v)
    suma := 0
    para i := 1 hasta |v|
        suma += v[i]
    devuelve suma
```

- La complejidad depende del tamaño de `v`

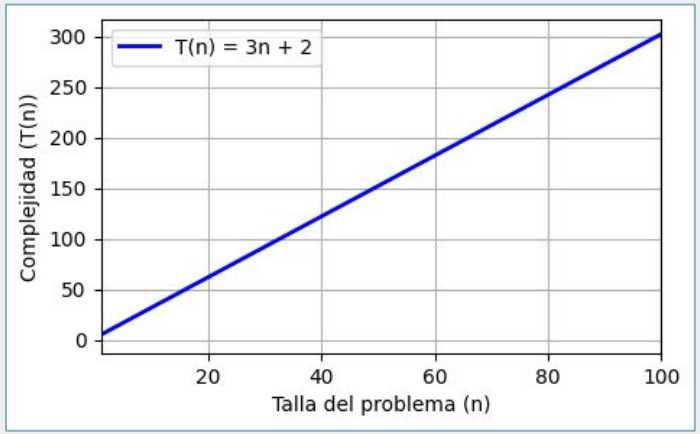
Complejidad teórica

Talla del problema

- La complejidad de un algoritmo se estima en función de su **talla**.
- La talla representa el **tamaño de la entrada**.

```
función acumulado(v)
    suma := 0
    para i := 1 hasta |v|
        suma += v[i]
    devuelve suma
```

$$T(n) = 1 + n(3) + 1 = 3n + 2$$



Complejidad teórica

Cotas de complejidad

- ¿Qué complejidad tiene el siguiente algoritmo?

```
función buscar(v, z)
    para i := 1 hasta |v|
        si v[i] = z
            devuelve i
    devuelve NO_ENCONTRADO
```

- La complejidad varía en función de los valores de la entrada.

Complejidad teórica

Cotas de complejidad

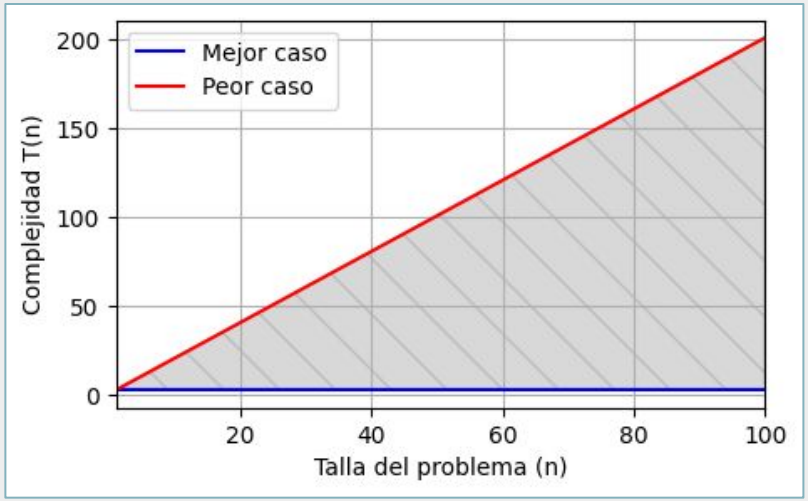
- ¿Cómo podemos calcular la complejidad cuando depende de algo **externo** al algoritmo?
  - Inferir **el mejor y el peor caso** y estimar los límites superiores e inferiores de su complejidad.
  - A esto se le llama **cotas de complejidad**.

Complejidad teórica

Cotas de complejidad

- ¿Qué complejidad tiene el siguiente algoritmo?

```
función buscar(v, z)
    para i := 1 hasta |v|
        si v[i] = z
            devuelve i
    devuelve NO_ENCONTRADO
```



Mejor caso: $T(n) = 3$

Peor caso: $T(n) = 2n + 1$

Notación asintótica

Notación asintótica

Concepto

- No es (tan) interesante medir la complejidad exacta de un algoritmo sino **cómo crece** cuando la talla es cada vez mayor.
- Para esto se puede utilizar la **notación asintótica *Big O***:
  - Proporciona una estimación de la complejidad cuando la talla del problema **tiende a infinito**.

Notación asintótica

Big O

- Formalmente:

$$T(n) \in \mathcal{O}(g(n)) \iff \exists n_0, c > 0 : T(n) \leq c \cdot g(n) \, \forall n \geq n_0$$

- Se centra en cómo **escala** el algoritmo.
- **Factores constantes** y **términos de menor orden** se ignoran.
- Es una manera **eficaz** de analizar y comparar algoritmos.

Notación asintótica

Big O

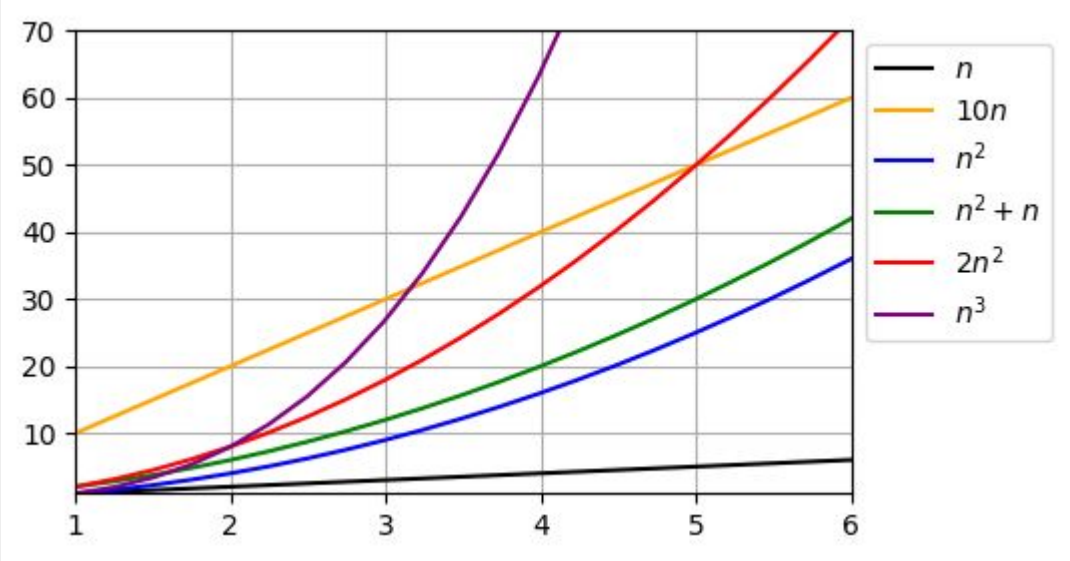
$$n^2 + 3 \in \mathcal{O}(n^2)$$

$$n^2 + n + 3 \in \mathcal{O}(n^2)$$

$$4n^2 \in \mathcal{O}(n^2)$$

$$n^2 + 3 \in \mathcal{O}(n^3)$$

$$n^2 + 3 \notin \mathcal{O}(n)$$



Notación asintótica

Órdenes de complejidad

- Complejidad constante: $O(1)$
- Complejidad logarítmica: $O(\log n)$
- Complejidad lineal: $O(n)$
- Complejidad lineal-logarítmica: $O(n \log n)$
- Complejidad cuadrática: $O(n^2)$
- Complejidad exponencial: $O(2^n)$

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(2^n)$$

Notación asintótica

Otras notaciones

- La notación *Big O* forma parte de un conjunto más amplio de notaciones.
  - Notación Ω: límites inferiores.
  - Notación Θ: funciones en la intersección de O y Ω.

Cálculo de complejidades

Cálculo de complejidades

Pasos

1. Determinar la talla del problema.
2. Determinar los casos mejor o peor (si los hubiera).
3. Calcular la complejidad asintótica de cada caso.

Cálculo de complejidades

Algoritmos iterativos

```
función producto_matrices_cuadradas(A, B)
    n := dimensión(A)
    C := ceros(A)
    para i := 1 hasta n
        para j := 1 hasta n
            suma := 0
            para k := 1 hasta n
                suma := suma + A[i][k] * B[k][j]
            C[i][j] := suma
    devuelve C
```

Cálculo de complejidades

Algoritmos iterativos

```
función ordenación_por_inserción(v)
    para i := 2 hasta |v|
        valor := v[i]
        j := i - 1
        mientras j > 0 y v[j] > valor
            v[j + 1] := v[j]
            j := j - 1
        v[j + 1] := valor
```

Cálculo de complejidades

Tipos

En la práctica, los algoritmos se categorizan como:

- **Iterativos**: análisis directo basado en la cantidad de veces que se ejecutan los distintos bloques del algoritmo.
- **Recursivos**: cuya complejidad se calcula recursivamente.

Cálculo de complejidades

Algoritmos recursivos

- La complejidad de un algoritmo recursivo depende del número y la naturaleza de sus llamadas recursivas.
- **Relación de recurrencia:** describe cómo el problema original se descompone en subproblemas más pequeños.
- Proporcionan un marco para analizar la complejidad de un algoritmo recursivo.

Cálculo de complejidades

Búsqueda binaria

```
función búsqueda_binaria(v, z)
    si |v| = 0
        devuelve NO_ENCONTRADO
    m := |v| / 2
    si v[m] = z
        devuelve m
    si v[m] > z
        devuelve búsqueda_binaria(v[:m], z)
    si_no
        devuelve búsqueda_binaria(v[m:], z)
```

Mejor caso: $T(n) \in \mathcal{O}(1)$

Peor caso: $T(n) \in \mathcal{O}(\log n)$

- ¿Qué complejidad tendría una **búsqueda secuencial**?

Cálculo de complejidades

Ordenación por selección

```
función ordenación_por_selección(v):
    si |v| = 1:
        devuelve

    índice_mínimo := 1
    para i=1 hasta |v|:
        si v[i] < v[índice_mínimo]:
            índice_mínimo = i

    intercambiar(v[1], v[índice_mínimo])

    ordenación_por_selección(v[2:])
```

$$T(n) \in \mathcal{O}(n^2)$$

- ¿Es más eficiente que la **ordenación por inserción**?

Ejercicios

Ejercicios

Pertenencias de complejidad

$n^3 \in \mathcal{O}(n^2)?$

$n^3 + n \in \mathcal{O}(n^3)?$

$n^2 \in \mathcal{O}(2^n)\,?$

$2^{n+1} \in \mathcal{O}(2^n)\,?$

$\log(2n) \in \mathcal{O}(\log n)\,?$

$\log n \in \mathcal{O}(n^{\frac{1}{2}})?$

$$f(n) \notin \mathcal{O}(g(n)) \implies g(n) \in \mathcal{O}(f(n))?$$

Ejercicios

Cálculo de complejidad iterativa (I)

```
función calculo(n)
    resultado := 0
    para i := 1 hasta n
        j := n
        mientras j > 1
            resultado := resultado + (i * j)
            j := j / 2
    devuelve resultado
```

Ejercicios

Cálculo de complejidad iterativa (II)

```
función conteo(n, a)
    count := 0
    i := 1
    mientras i < n
        j := n
        mientras j > 1
            count := count + 1
            j := j / 2
        si mod(a,2) = 0
            i := i * 2
        si no
            i := i + 1
    devuelve count
```

Ejercicios

Cálculo de complejidad recursiva (I)

Calcula el orden de complejidad de la siguiente relación de recurrencia:

$$T(n) = 4T\left(\frac{n}{2}\right) \qquad T(1) = n$$

Ejercicios

Cálculo de complejidad recursiva (II)

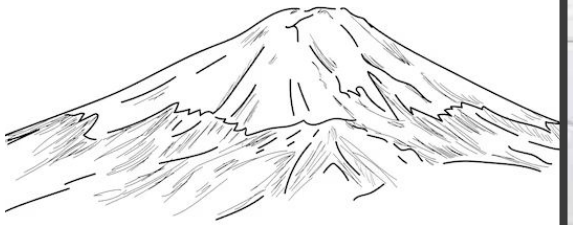
```
función recursiva(v)
    si |v| <= 1
        devuelve v[1]
    si v[1] < v[2]
        devuelve recursiva(v[2:])
    si no
        x := 0
        para i := 1 hasta |v|
            para j := i hasta |v|
                x := v[i] + v[j]
        devuelve x + recursiva(v[:-1])
```

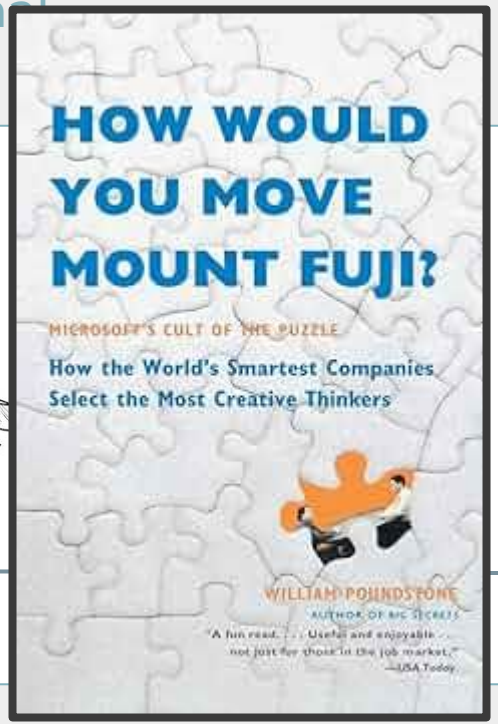
Divide y vencerás

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Resolución de problemas

Pensamiento computacional





Resolución de problemas

El reparto de bitcoins

- **Cinco** criptobros han descubierto una cartera digital con **100 bitcoins**...
  - Los bros están **jerarquizados**: 5 > 4 > 3 > 2 > 1.
  - Hacen **propuestas** en ese orden: se acepta la propuesta si hay **mayoría**; si no, **eliminan al que propone** y sigue el resto.
  - En caso de empate, el voto del proponente vale doble.
  - Todos son ***inteligentes y egoístas***: siempre van votar la mejor opción para ellos mismos.

¿Qué propuesta hará el criptobro 5?

Resolución de problemas

El reparto de bitcoins

- ¿Si fueran N criptobros y una cartera con K bitcoins?

Divide y vencerás

Divide y vencerás

Nociones

- **Estrategia** para resolver problemas complejos **dividiéndolos** en problemas más pequeños y manejables:
  - Se divide en partes más pequeñas, hasta que su resolución sea trivial.
  - Se combinan las soluciones para obtener la solución final.
- Gran cantidad de problemas, tanto computacionales como generales.

Divide y vencerás

Formalización

- Tres etapas:
  - **División**: Se divide el problema original en subproblemas más pequeños que son instancias del mismo tipo de problema.
  - **Resolución**: Cuando un subproblema es lo suficientemente simple (trivial), se resuelve de forma directa.
  - **Combinación**: Se combinan las soluciones de los subproblemas para obtener la solución del problema original.

Divide y vencerás

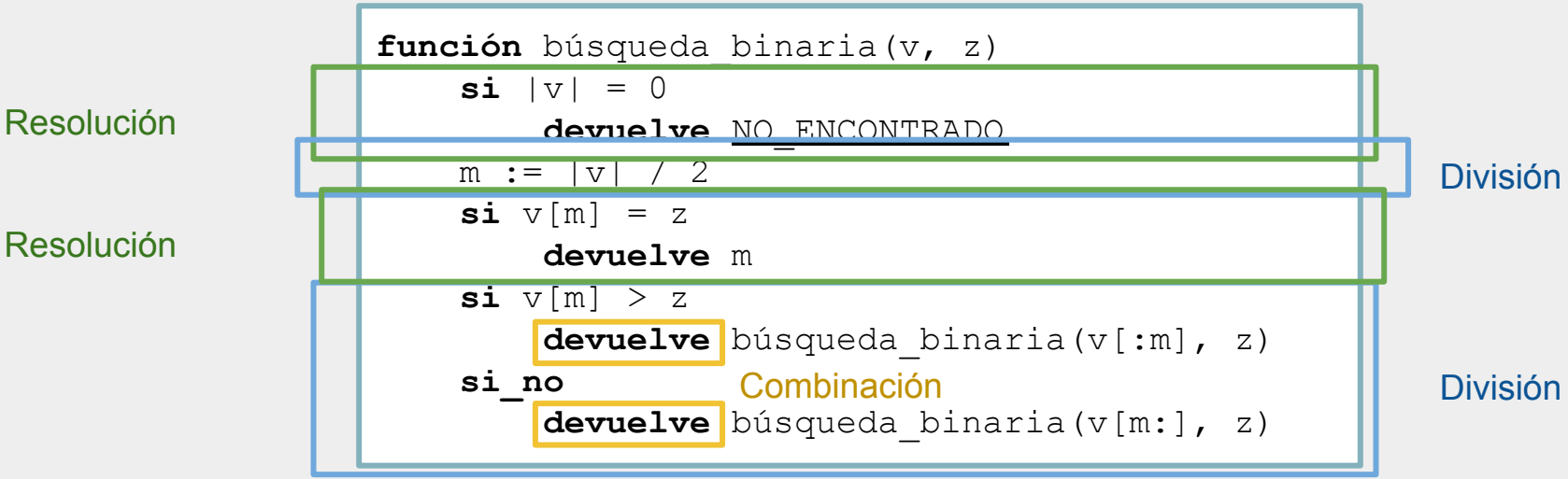
Esquema general

```
función dyv(n)
    si trivial(n)
        devuelve resolver(n)

    subproblemas = division(n)
    para i := 1 hasta |subproblemas|
        soluciones += dyv(subproblema)

    devuelve combinar(soluciones)
```

Divide y vencerás

Ejemplo: búsqueda binaria

La búsqueda binaria es un ejemplo de divide y vencerás:



Algoritmos de ordenación

Divide y vencerás

Algoritmos de ordenación

- ¿Cómo ordenar un vector siguiendo el esquema divide y vencerás?

4	3	7	8	9	2	7	9	0

Ordenación por mezcla: MergeSort

Idea principal

- **Idea principal**: es más simple obtener una lista ordenada a partir de dos sublistas ya ordenadas.
- En las etapas del esquema:
  - **División**: se divide el vector en dos partes iguales.
  - **Resolución**: cuando la lista es de tamaño 1, ya está ordenada.
  - **Combinación**: dadas dos listas ordenadas, mezclar sus elementos manteniendo el orden.

Ordenación por mezcla: MergeSort

Traza

4	3	7	8	9	2	7	9	0

Ordenación por mezcla: MergeSort

Algoritmo

```
función ordenación_por_mezcla(v):
    si |v| <= 1:
        devuelve v

    mitad := |v| / 2
    v_i := ordenación_por_mezcla(v[:mitad])
    v_d := ordenación_por_mezcla(v[mitad:])

    v := mezclar(v_i,v_d)
    devuelve v
```

Ordenación por mezcla: MergeSort

Función *mezclar*

```
función mezclar(v_i, v_d):
    i,d := 0
    resultado := []

    mientras i < |v_i| y d < |v_d|:
        si v_i[i] <= v_d[d]:
            resultado += v_i[i]
            i := i + 1
        si no:
            resultado += v_d[d]
            d := d + 1

     resultado += v_i[i:]
     resultado += v_d[d:]

    devuelve resultado
```

Ordenación por mezcla: MergeSort

Complejidad

```
función ordenación_por_mezcla(v):
    si len(v) <= 1:
        devuelve v

    mitad := |v| / 2
    v_i := ordenación_por_mezcla(v[:mitad])
    v_d := ordenación_por_mezcla(v[mitad:])

    v := mezclar(v_i,v_d)
    devuelve v
```

$$T(n) \in \mathcal{O}(n \log n)$$

Ordenación rápida: QuickSort

Idea principal

- Escoger un elemento (pivote) y colocarlo en su sitio **correcto**.
- Dividir el resto de elementos en dos conjuntos:
  - Los elementos **menores** que el pivote (izquierda)
  - Los elementos **mayores** que el pivote (derecha)
- Ordenar **recursivamente** estos dos conjuntos

Ordenación rápida: QuickSort

Algoritmo

```
función qsrt(v):
    si |v| <= 1
        devuelve v
    si no
        pivote := elegir-pivote(v)
        izquierda, derecha := partición(v,pivote)
        devuelve [qsrt(izquierda),pivote,qsrt(derecha)]
```

Ordenación rápida: QuickSort

Algoritmo: funciones auxiliares

- **elegir-pivote(v)**: ¿cómo se puede elegir un pivote? ¿cuál sería el *mejor* pivote? ¿qué coste tendrían estas elecciones?
- **partición(v, pivote):** ¿cómo se puede dividir el vector en dos partes a partir del pivote? ¿con qué coste?

Ordenación rápida: QuickSort

Traza

4	3	7	8	9	2	7	9	0

Ordenación rápida: QuickSort

Complejidad

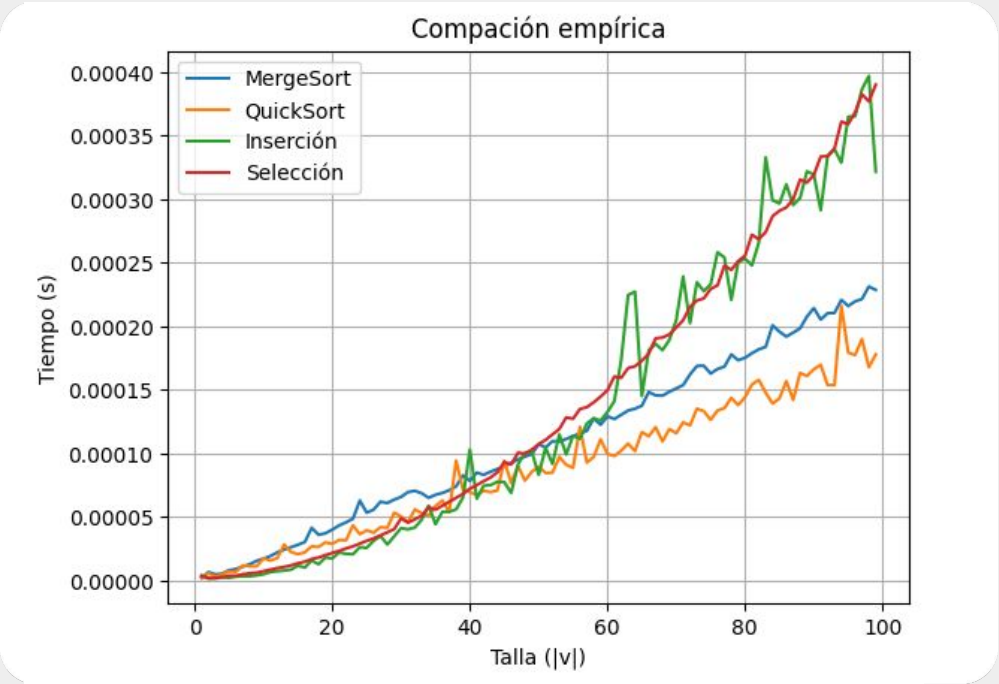
```
función qsrt(v):
    si |v| <= 1
        devuelve v
    si no
        pivote := elegir-pivote(v)
        izquierda, derecha := partición(v,pivote)
        devuelve [qsrt(izquierda),pivote,qsrt(derecha)]
```

Mejor caso: $T(n) \in \mathcal{O}(n \log n)$

Peor caso: $T(n) \in \mathcal{O}(n^2)$

Algoritmos de ordenación

Comparativa de complejidad



Consideraciones finales

Divide y vencerás

Consideraciones

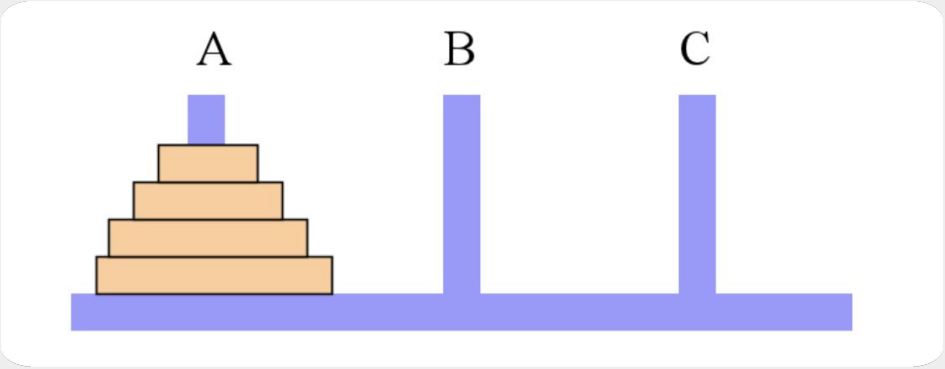
- No siempre un problema de talla menor es más fácil de resolver.
- La solución de los subproblemas no implica necesariamente que la solución del problema original se pueda obtener fácilmente
- Aplicable si encontramos:
  - Forma de descomponer un problema en subproblemas de talla menor
  - Forma directa de resolver problemas menores a un tamaño determinado
  - Forma de combinar las soluciones de los subproblemas que permita obtener la solución del problema original

Problemas

Divide y vencerás

Problema: Torres de Hanoi

- Llevar los discos de A a C, pudiendo usar B (auxiliar).



- **Reglas**: los discos solo se pueden mover uno a uno y nunca se puede poner un disco sobre uno más pequeño.

Divide y vencerás

Problema: el juego del Nim

- Hay N fichas en un tablero y cada jugador retira, alternativamente, 1 o más fichas de la mesa hasta un máximo de M.
- El juego termina cuando no quedan fichas sobre la mesa, y pierde el jugador que tiene el turno y no puede retirar ninguna ficha.
- ¿Hay una jugada ganadora para un N y M dados?

Divide y vencerás

Problema: Contar número de inversiones

- Dado un vector v, ¿cuántas inversiones contiene?
- El par (i,j) forma una inversión si v[i] > v[j] y i < j.

4	3	7	0

4 inversiones

- Búsqueda exhaustiva, ¿complejidad?
- ¿Podemos hacerlo mejor?

Programación dinámica

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Programación dinámica

La sucesión de Fibonacci

La sucesión de Fibonacci viene definida tal que:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

Programación dinámica

La sucesión de Fibonacci

```
función fibonacci(n)
    si n <= 1
        devuelve n
    si no
        devuelve fibonacci(n-1) + fibonacci(n-2)
```

$$F(n) \in \mathcal{O}(2^n)$$

¿Dónde está el problema?

Esquema general

Programación dinámica

Esquema general

- **Idea general**: descomposición del problema en subproblemas solapados.
- La eficiencia se consigue almacenando los resultados de los subproblemas, de manera que sólo se resuelven una única vez.

Programación dinámica

Principio de optimalidad

- Para que se pueda aplicar programación dinámica es esencial que se cumpla el **principio de optimalidad**.
- Un problema tiene una **subestructura óptima** si una solución **óptima** puede construirse **eficientemente** a partir de las **soluciones óptimas de sus subproblemas**.

Programación dinámica

Esquema general

Tipos de programación dinámica:

- **Memoización** (programación dinámica recursiva): Se almacenan los resultados de las llamadas recursivas realizadas.
- **Tabulación** (programación dinámica iterativa): Los resultados de subproblemas más pequeños se almacenan en una estructura, utilizándose para resolver los problemas más grandes

Programación dinámica

El número de Fibonacci (con memoización)

```
función fib(n, memo)
    si n <= 1
        devuelve n
    si n en memo
        devuelve memo[n]
    si no
        memo[n] := fib(n-1, memo) + fib(n-2, memo)
        devuelve memo[n]
```

$$T(n) \in \mathcal{O}(n)$$

Programación dinámica

El número de Fibonacci (con tabulación)

```
función fib(n, memo)
    si n <= 1
        devuelve n
    tabla[0] := 0
    tabla[1] := 1
    para i desde 2 hasta n:
        tabla[i] := tabla[i-1] + tabla[i-2]
    devuelve tabla[n]
```

$$T(n) \in \mathcal{O}(n)$$

Programación dinámica

Comparación con *Divide y vencerás*

Divide y vencerás	Programación dinámica
Divide el problema en **subproblemas independientes**	Divide el problema en **subproblemas independientes**
Los subproblemas se resuelven **por separado**	Los subproblemas se resuelven **una vez y se almacenan**
Combina los resultados de los subproblemas **después de resolverlos**	**Reutiliza** subproblemas **previamente resueltos** para construir la solución

El problema de la mochila

Programación dinámica

El problema de la mochila

- El **problema de la mochila** (*knapsack problem*) es un problema clásico en teoría de algoritmos y computación.
- Consiste en **seleccionar** un subconjunto de **elementos**, cada uno con un **peso** y un **valor**, de manera que se **maximice el valor total sin exceder el peso máximo** permitido.
- Muchos problemas **reales** pueden reducirse a **una instancia** del problema de la mochila.

Programación dinámica

El problema de la mochila

- Existen varias versiones del problema de la mochila.
  - Problema de la mochila *discreta* (**programación dinámica**): los pesos son enteros.
  - Problema de la mochila *continua* (**algoritmo voraz**): los objetos se pueden fraccionar.
  - Problema de la mochila *general* (**algoritmos de vuelta atrás**): los pesos no se pueden fraccionar

Programación dinámica

El problema de la mochila

Instancia

Valores $(v_1, v_2, \ldots, v_n)$

Pesos $(w_1, w_2, \ldots, w_n)$

Peso máximo W

Problema (versión discreta)

$$\arg \max_{\mathbf{x} \in \{0,1\}^n} \sum_{i=1}^{n} v_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq W \qquad w_i \in \mathbb{N}$$

Programación dinámica

El problema de la mochila (discreta)

Instancia:

- **Valores**: v = (6, 6, 2, 1)
- **Pesos**: w = (3, 2, 1, 1)
- **Capacidad**: W = 5

Valor máximo: 12

Solución: x = (1, 1, 0, 0)

Programación dinámica

El problema de la mochila (discreta)

¿Cómo se puede resolver recursivamente?

- La mejor opción entre coger el objeto *i*-ésimo o no cogerlo cuando aún hay capacidad *W*.

$$K(i,W) = \begin{cases} 0 & \text{si } i = 0 \text{ o } W = 0 \\ K(i-1,W) & \text{si } w_i > W \\ \max(K(i-1,W), v_i + K(i-1,W-w_i)) & \text{si } w_i \leq W \end{cases}$$

Programación dinámica

El problema de la mochila (discreta)

Solución recursiva

```
función knapsack(i, W, v, w)
    si i = 0
        devuelve 0
    si w[i] > W
        devuelve knapsack(i-1, W, v, w)
    si no
        devuelve máx( knapsack(i-1, W, v, w),
                      v[i] + knapsack(i-1, W-w[i], v, w) )
```

¿Qué complejidad tiene este algoritmo?

Programación dinámica

El problema de la mochila (discreta)

¿Cómo convertimos a Programación dinámica?

```
función knapsack(i, W, v, w)
    si i = 0
        devuelve 0
    si w[i] > W
        devuelve knapsack(i-1, W, v, w)
    si no
        devuelve máx( knapsack(i-1, W, v, w),
                      v[i] + knapsack(i-1, W-w[i], v, w) )
```

¿Qué complejidad tienen estas soluciones?

La distancia de edición

Programación dinámica

La distancia de edición

- La distancia de edición (o *distancia de Levenshtein*) es una métrica que indica cuán similares son dos cadenas de texto.
- La distancia se define como el número mínimo de operaciones necesarias para transformar una cadena en otra.
  - Las operaciones permitidas son la inserción, eliminación y sustitución de caracteres.
- Esta métrica es fundamental en diversas aplicaciones de la inteligencia artificial

Programación dinámica

La distancia de edición

Consideremos el ejemplo **d(“casa”, “costa”)**:

- Sustituimos 'o' por la primera 'a': “cosa”
- Insertamos 't' después de 's’: “costa”
- Distancia total: 2

Programación dinámica

Distancia de edición: planteamiento recursivo

Dada la cadena origen y destino, se comparan los últimos caracteres:

- Si es el mismo, la solución es la misma que para sus subcadenas.
- Si es diferente:
  - Consideramos la operación de menor coste entre:
    - Insertar el carácter de la cadena destino.
    - Sustituir el carácter de la cadena origen por el de la cadena destino.
    - Eliminar el carácter de la cadena origen.
  - Y llamar recursivamente a las subcadenas correspondientes.

Programación dinámica

Distancia de edición: planteamiento recursivo

Dada la cadena s1 (de tamaño i) y la cadena s2 (de tamaño j):

$$d(i,j) = \begin{cases} j & \text{si } i = 0 \\ i & \text{si } j = 0 \\ d(i-1, j-1) & \text{si } s1[i-1] = s2[j-1] \\ 1 + \min\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & \text{si } s1[i-1] \neq s2[j-1] \end{cases}$$

Programación dinámica

Distancia de edición: solución recursiva

```
función distancia_edición(s1, s2)

    si s1[i-1] = s2[j-1]
        devuelve distancia_edición(s1[:i-1],s2[:j-1])
    si no
        devuelve 1 + min( distancia_edición([s1[:i-1],s2)
                          distancia_edición(s1,s2[:j-1]),
                          distancia_edición(s1[:i-1]s2[:j-1]) )
```

Programación dinámica

Distancia de edición

```
función distancia_edición(s1, s2)
    n,m := |s1|,|s2|
    DP := matriz de tamaño (n+1, m+1)
    DP[i][0] := i para i desde 0 hasta n
    DP[0][j] := j para j desde 0 hasta m

    para i desde 1 hasta n:
        para j desde 1 hasta m:
            si s1[i-1] = s2[j-1]
              DP[i][j] := DP[i-1][j-1]
            si no
              DP[i][j] := 1 + min( DP[i-1][j], DP[i][j-1], DP[i-1][j-1] )

    devuelve DP[n][m]
```

Recuperación de soluciones

Programación dinámica

Recuperación de soluciones

- En los ejemplos anteriores hemos calculado los valores óptimos:
  - Máximo valor de la mochila
  - Distancia mínima de edición
- No proporcionamos las soluciones (qué objetos, qué operaciones de edición...)
- ¿Cómo podemos calcular las soluciones (eficientemente)?

Programación dinámica

Recuperación de soluciones

d(“casa”,“costa”)

	-	c	o	s	t	a
-	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2	1	1	2	3	3
s	3	2	2	1	2	3
a	4	3	3	2	2	2

Programación dinámica

Recuperación de soluciones

d(“casa”,“costa”)

	-	c	o	s	t	a
-	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2	1	1	2	3	3
s	3	2	2	1	2	3
a	4	3	3	2	2	2

Orden inverso:

- cost**a** ← cas**a** (‘a’ con ‘a’)
- cost ← cas (insertar ‘t’)
- cos ← cas (‘s’ con ‘s’)
- c**o** ← c**a** (‘a’ por ‘o’)
- **c** ← **c** (‘c’ con ‘c’)
- ←

Ejercicios

Programación dinámica

Ejercicio: búsqueda de soluciones

- Dada una matriz PD[·][·] utilizada para calcular la distancia de edición entre dos cadenas, escribe el algoritmo que recupera las operaciones de edición a realizar.
- Dada una matriz PD[·] utilizada para calcular el valor óptimo que cabe en una mochila, escribe el algoritmo que recupera qué objetos se han añadido.

Programación dinámica

Ejercicio: venta de oro

- Una empresa compra piezas de oro de **n** onzas y las trocea en piezas de **i** onzas (*i = 1, 2, ..., n*) que luego vende.
- El corte le sale gratis.
- El precio de venta de una pieza de **i** onzas es **v**$_{i}$
- ¿Cual es la forma óptima de trocear una pieza de **n** onzas para maximizar el precio de venta acumulado?

Programación dinámica

Ejercicio: cesta de la compra

- Queremos minimizar el coste de comprar N productos, que pueden adquirirse en cualquiera de K supermercados de la ciudad.
- Se conoce el coste c_k de desplazarse hasta el sitio s_k y los precios $p_{k1}, p_{k2}, \ldots, p_{kN}$ de cada producto en s_k.
- El gasto c_k sólo se suma al coste total (una sola vez) si se adquiere, al menos, un producto en s_k.
- ¿Cómo se calcula el coste mínimo de la compra?

Algoritmos voraces

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Algoritmos voraces

Introducción

- Hay determinados problemas que se pueden resolver **tomando decisiones secuencialmente**, sin tener en cuenta más posibilidades.
- Un **algoritmo voraz** sigue un criterio de selección que elige una opción **óptima local**, con la esperanza de llegar a una solución **óptima global**.

Algoritmos voraces

Esquema general

1. Inicializar la solución.
2. Repetir hasta completar una solución:
    a. Seleccionar el mejor candidato según un criterio voraz.
    b. Comprobar si este candidato puede ser añadido a la solución.
    c. Si se puede: añadir el candidato a la solución.
3. Devolver la solución obtenida.

El problema de la mochila (continua)

Algoritmos voraces

El problema de la mochila (continua)

Instancia

Valores $(v_1, v_2, \ldots, v_n)$

Pesos $(w_1, w_2, \ldots, w_n)$

Peso máximo W

Problema (versión continua)

$$\arg \max_{\mathbf{x} \in [0,1]} \sum_{i=1}^{n} v_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq W$$

Algoritmos voraces

El problema de la mochila (continua)

- La versión continua del problema de la mochila se puede resolver mediante un **algoritmo voraz**:
    1. **Ordenar** los elementos en relación **descendente** valor / peso
    2. Iterativamente: añadir la **fracción del objeto** correspondiente que cabe en la mochila.

Algoritmos voraces

El problema de la mochila (continua)

Instancia:

- **Valores**: v = (6, 6, 2, 1)
- **Pesos**: w = (3, 4, 1, 1)
- **Capacidad**: W = 5

Valor máximo: 9.5

Solución: x = (1, 0.25, 1, 0)

Algoritmos voraces

El problema de la mochila (continua)

```
función mochila_continua(W, n, pesos, valores):
    peso_total, valor_total := 0

    ordenar_por_ratio(valores, pesos)
    para i desde 1 hasta n:
        si peso_total + pesos[i] <= W
            fraccion := 1
        si no
            fraccion := (W - peso_total) / pesos[i]
        valor_total := valor_total + valores[i] * fraccion
        peso_total := peso_total + pesos[i] * fraccion

    devuelve valor_total
```

Solución completa: ordenar los id iniciales

Algoritmos voraces

El problema de la mochila

- ¿Funciona este esquema en el caso discreto?
  - Puede dar una solución pero, ¿es óptima?
  - ¿Podemos hablar de *algoritmo*?
- ¿Es óptimo en el caso voraz?
  - Los algoritmos voraces requieren una **demostración de optimalidad** (o un contraejemplo para lo contrario).

Algoritmo de Kruskal

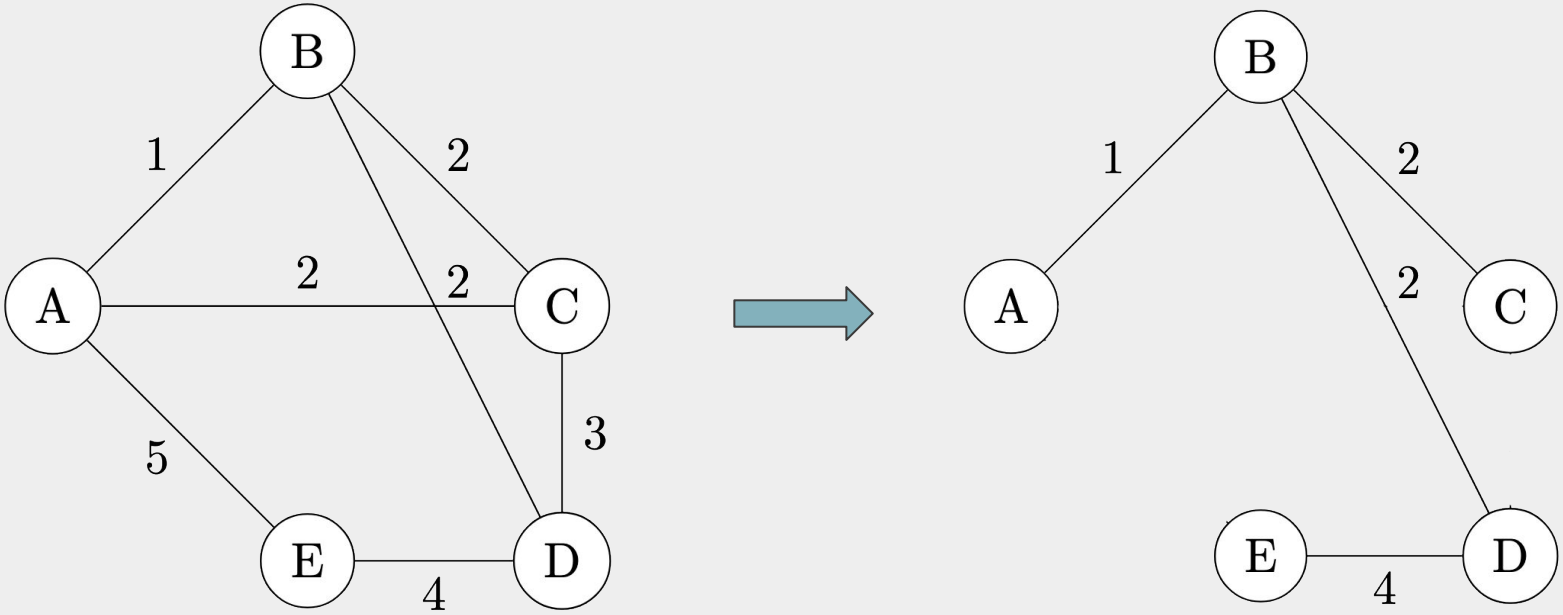
Algoritmos voraces

Árbol de expansión mínima

- Dado un grafo ponderado no dirigido, el **árbol de expansión mínima** (*Minimum Spanning Tree, o MST*) es el **subgrafo** que:
  - Conecta todos los vértices del grafo original.
  - Con el menor peso total posible.
  - Y que no contiene ciclos.
- El MST es fundamental en muchas áreas de la computación.

Algoritmos voraces

Árbol de expansión mínima



Algoritmos voraces

Árbol de expansión mínima: Algoritmo de Kruskal

- El **algoritmo de Kruskal** es un enfoque voraz para calcular el MST.
- Mantiene una **estructura de datos** con todos los sub-árboles producidos hasta el momento.
- Consulta las aristas por **orden creciente** de peso:
  - Si la arista conecta dos vértices de árboles distintos, la incluye en la solución y los dos árboles se unen.
  - Si no, se descarta.

Algoritmos voraces

Árbol de expansión mínima: Algoritmo de Kruskal

1. Inicializar un conjunto de árboles, cada uno con un solo vértice.
2. Crear una lista de todas las aristas del grafo, ordenadas por peso en orden ascendente.
3. Para cada arista en la lista ordenada:
    a. Si la arista conecta dos árboles distintos, añadirla a la solución y unir los dos árboles
4. Repetir el paso 3 hasta que todos los vértices estén conectados en un único árbol.

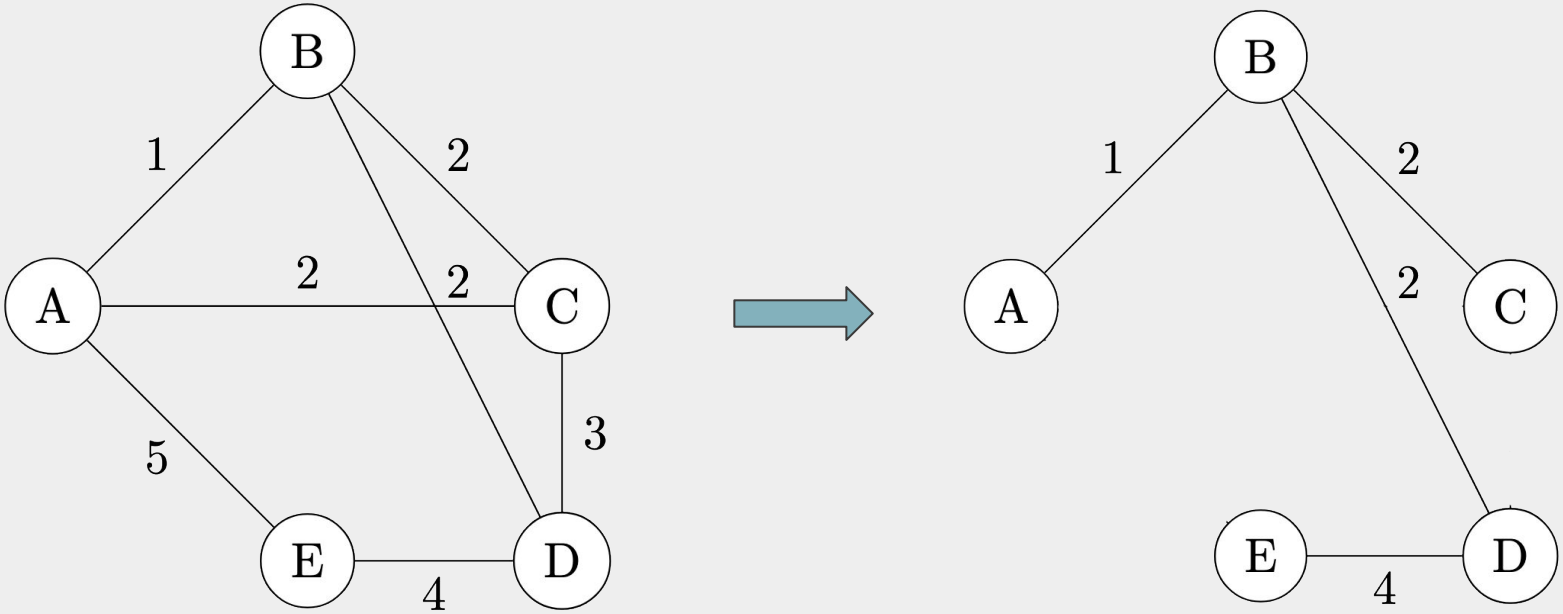
Algoritmos voraces

Árbol de expansión mínima: Algoritmo de Kruskal

```
función kruskal(V,A):
    MST := ∅
    T := estructura de conjuntos
    para i desde 1 hasta |A|
        T[i] = {i}

    ordenar(A)

    para cada (u, v) en A:
        si T[u] != T[v]
            MST = MST ∪ {(u, v)
            unión(T[u],T[v])

    devuelve MST
```

Algoritmos voraces

Árbol de expansión mínima: Algoritmo de Kruskal



Algoritmos voraces

Árbol de expansión mínima: Algoritmo de Kruskal

- ¿Qué estructura de datos utiliza el algoritmo de Kruskal?
  - Estructura ***union-find / disjoint-set***.
    - Operaciones de pertenencia (*find*) y unión (*union*) casi constantes.
    - Complejidad del algoritmo dominada por la ordenación de aristas.
  - Con una **estructura convencional** (vector de pertenencia):
    - Operaciones de pertenencia y unión lineales.
    - Complejidad dominada por el bucle sobre las aristas y operaciones.
- Importancia de usar la **estructura de datos apropiada**.

Ejercicios

Algoritmos voraces

El cambio de monedas

- Dado un conjunto de tipos de monedas $D = (d_1, d_2, \ldots, d_n)$, se busca el **mínimo número de monedas** cuya suma sea una cantidad de dinero C.
- ¿Se puede encontrar un **algoritmo voraz**?

Algoritmos de vuelta atrás

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Vuelta atrás

Introducción

- Hay determinados problemas cuya única solución es **enumerar todas las posibles soluciones** y guardar la mejor.
- Este enfoque es óptimo **por definición** pero lleva a complejidades asintóticas **exponenciales**.
- La **vuelta atrás** es un estrategia para enumerar todas las posibles soluciones, adecuada para añadir **mejoras prácticas a la eficiencia**.

El problema de la mochila (general)

Vuelta atrás

El problema de la mochila

Instancia

Valores $(v_1, v_2, \ldots, v_n)$

Pesos $(w_1, w_2, \ldots, w_n)$

Peso máximo W

Problema (versión general)

$$\arg \max_{\mathbf{x} \in \{0,1\}^n} \sum_{i=1}^{n} v_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq W$$

$$\mathbf{v}, \mathbf{w} \in \mathbb{R}$$

Vuelta atrás

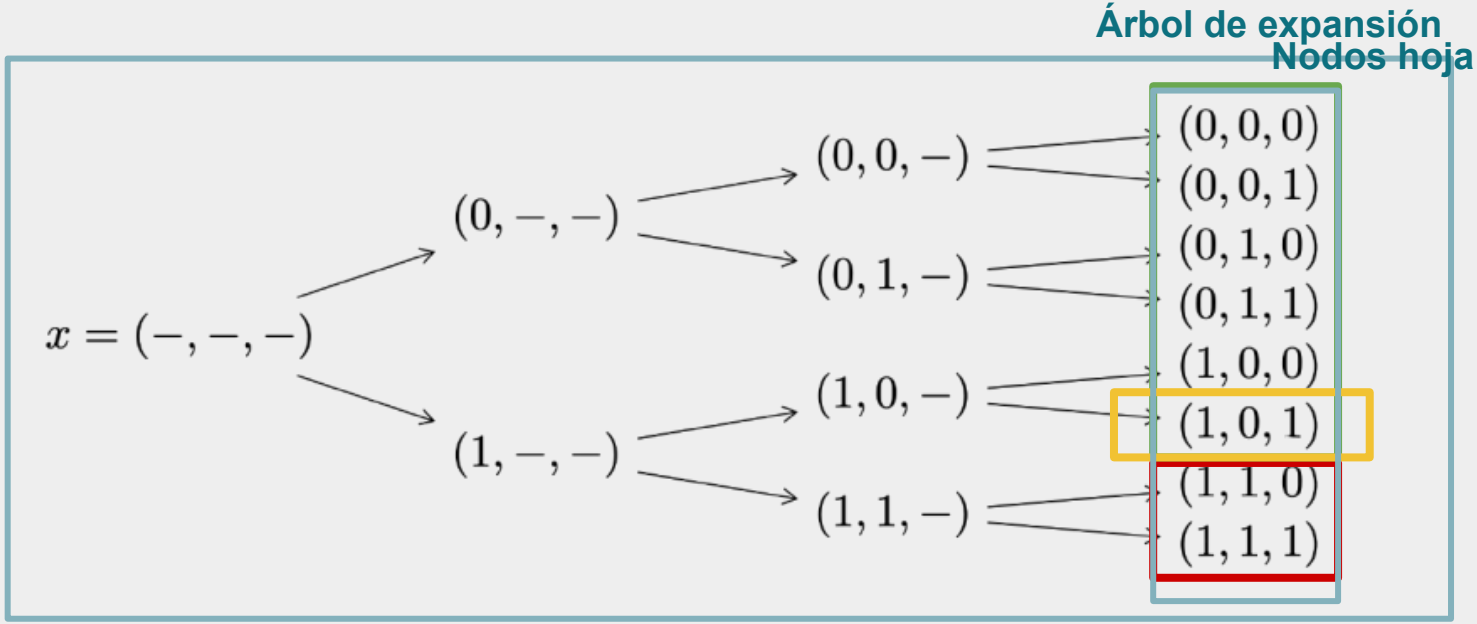
El problema de la mochila (general)

- Posibles soluciones:
  - **Programación dinámica**: infinitos subproblemas.
  - **Estrategia voraz**: no es óptima.
- **Enumerar** todas las posibles formas de llenar la mochila:
  - Soluciones que **cumplen con la restricciones** (soluciones factibles)
  - Solución que **maximiza el valor** (solución óptima)

Vuelta atrás

El problema de la mochila (general)

Asumiendo v = {6, 5, 1}, w = {25, 10, 5} y W = 30.



Vuelta atrás

El problema de la mochila (general)

- **Vuelta atrás**: forma sistemática de generar todas las posibles configuraciones de la mochila.
  - **Codificamos** la solución en una tupla (vector binario).
  - **Versión recursiva**: cada expansión del árbol es una llamada recursiva considerando una opción en un índice concreto de la tupla.
    - Cuando la llamada *vuelve*, se considera la siguiente opción.
    - Una vez agotadas las opciones, se *vuelve atrás* al elemento anterior del vector solución.

Vuelta atrás

El problema de la mochila (general): enumeración

Asumimos v, w, W accesibles globalmente (no cambian)

Llamada inicial: v_atras(0,|x|=n, best=0)

```
función v_atras(i, x, best):
    si i = n
        si peso(x,w) <= W:
            best = max(best, valor(x,v))

    si no
        x[i] = 0
        v_atras(i+1, x, best)
        x[i] = 1
        v_atras(i+1, x, best)
```

peso y *valor* tienen coste lineal: ¿podemos mejorar?

Vuelta atrás

El problema de la mochila (general): aprovechando cálculos

Llamada inicial: v_atras(0,|x|=n, best=0, 0, 0)

```
función v_atras(i, x, best, v_acc, w_acc):
    si i = n
        si w_acc <= W:
            best = max(best, v_acc)

    si no
        x[i] = 0
        v_atras(i+1, x, best, v_acc, w_acc)
        x[i] = 1
        v_atras(i+1, x, best, v_acc + v[i], w_acc + w[i])
```

¿Debemos esperar a una hoja para mirar la restricción de peso?

Vuelta atrás

El problema de la mochila (general): podando

Llamada inicial: v_atras(0,|x|=n, best=0, 0, 0)

```
función v_atras(i, x, best, v_acc, w_acc):
    si w_acc <= W:
        si i = n
            best = max(best, v_acc)

        si no
            x[i] = 0
            v_atras(i+1, x, best, v_acc, w_acc)
            x[i] = 1
            v_atras(i+1, x, best, v_acc + v[i], w_acc +
    w[i])
```

Vuelta atrás

Cotas optimistas

- Definimos como **solución parcial prometedora** aquella que **podría** mejorar al mejor valor obtenido hasta el momento.
- Interesa **podar** cualquier solución parcial **no prometedora**.
- ¿Podemos saber si una solución parcial es **prometedora**?
  - Asumiendo que todos los objetos restantes van a caber.
  - Asumiendo que todos los objetos restantes se pueden fraccionar.
- A estas estimaciones se les llama **cota optimista**.

Vuelta atrás

Cotas optimistas

- **Relajar las restricciones** del problema para obtener un cálculo optimista desde una solución parcial.
  - **Restricciones muy relajadas**: cota demasiado optimista, menos podas.
  - **Restricciones demasiado estrictas**: podrían podar soluciones prometedoras (*no es cota optimista*).

Vuelta atrás

El problema de la mochila (general): poda optimista

Llamada inicial: v_atras(0,|x|=n, best=0, 0, 0)

```
función v_atras(i, x, best, v_acc, w_acc):
    si w_acc <= W && es_prometedora(i, x, best):
        si i = n
            best = max(best, v_acc)

        si no
            x[i] = 0
            v_atras(i+1, x, best, v_acc, w_acc)
            x[i] = 1
            v_atras(i+1, x, best, v_acc + v[i], w_acc +
    w[i])
```

Vuelta atrás

Solución inicial

- No podemos **podar** hasta hasta tener una primera solución.
- ¿Podemos adelantarnos? → Utilizando una cota pesimista.
  - Una **cota pesimista** es una solución (sub)óptima de un problema.
  - Es importante que la solución sea **posible** (cumpla los requisitos del problema) y **eficiente** (para que sea útil).
  - Podemos utilizar un **algoritmo voraz**.

Vuelta atrás

El problema de la mochila (general): solución voraz

Llamada inicial: v_atras(0,|x|=n, best=voraz(v,w,W), 0, 0)

```
función v_atras(i, x, best, v_acc, w_acc):
    si w_acc <= W && es_prometedora(i, x, best):
        si i = n
            best = max(best, v_acc)

        si no
            x[i] = 0
            v_atras(i+1, x, best, v_acc, w_acc)
            x[i] = 1
            v_atras(i+1, x, best, v_acc + v[i], w_acc +
    w[i])
```

Vuelta atrás

El problema de la mochila (general): uso de cota pesimista

Llamada inicial: v_atras(0,|x|=n, best=voraz(v,w,W), 0, 0)

```
función v_atras(i, x, best, v_acc, w_acc):
    best = max(best, v_acc + voraz(i, v, w, W))

    si w_acc <= W && es_prometedora(i, x, best):
        si i = n
            best = max(best, v_acc)
        si no
            x[i] = 0
            v_atras(i+1, x, best, v_acc, w_acc)
            x[i] = 1
            v_atras(i+1, x, best, v_acc + v[i], w_acc +
    w[i])
```

Vuelta atrás

Resumen mejoras prácticas

Promedio del número de llamadas para 100 instancias aleatorias del problema de la mochila con 100 objetos.

Básico	Optimista	Inicio voraz	Pesimista
2,5e+30	4491	277	253

Esquema general

Vuelta atrás

Esquema general

- **Vuelta atrás**: forma sistemática de generar todas las soluciones.
- Para el elemento i-ésimo del vector solución:
  - Considerar una de las posibles opciones y continuar con el siguiente recursivamente.
  - Cuando la solución *vuelve*, considerar la siguiente opción.
  - Una vez agotadas las opciones, se *vuelve atrás* al elemento anterior del vector solución.

Vuelta atrás

Esquema general

Llamada inicial: v_atras(|x|=n, 0, n, –)

```
función v_atras(x, i, n, best):
    si i = n
        si factible(x):
            best = mejor(best, valor(x))

    si no
        para cada o en opciones(i)
            x[i] = o
            v_atras(x, i+1, n, best)
```

Vuelta atrás

Esquema general

- **Vuelta atrás**: forma sistemática de generar todas las soluciones.
- ¿Podemos hacerlo mejor?
  - Si una solución parcial no es prometedora, se “poda”.
  - ¿Podemos adelantar si una solución es prometedora? **Cota optimista**
  - ¿Podemos empezar a podar sin ninguna hoja? **Solución inicial voraz**
  - ¿Podemos actualizar la “mejor solución” antes de una hoja? **Cota pesimista**

Vuelta atrás

Esquema general

Llamada inicial: vuelta_atras([], 0, n, voraz())

```
función v_atras(x, i, n, best):
    si factible(x):
        si i = n
            best = mejor(best, valor(x))
        si no
            best = mejor(best, pesimista(x))
            si optimista(x) > best:
                para cada o en opciones(i)
                    x[i] = o
                    v_atras(x, i+1, n, best)
```

Vuelta atrás

Puntos claves

- **Formulación:** cómo codificar un vector solución.
- **Ahorrar cálculos:** reutilizar algunos cálculos relacionados con el valor de la solución o sus restricciones.
- **Cota optimista**: relajar las restricciones del problema para calcular rápidamente si la solución parcial es prometedora.
- **Cota pesimista**: calcular rápidamente una posible solución a partir de una solución parcial (o inicial), asegurando que sea factible.

Vuelta atrás

Consideraciones

- La vuelta atrás hace un recorrido en **profundidad**.
  - Existen variantes que exploran por **prioridad** (*ramificación y poda*).
- La versión recursiva se puede convertir a una **versión iterativa**.
- Las mejoras prácticas no reducen la complejidad asintótica sino únicamente empírica: **dependiente del problema y la instancia**.

El viajante de comercio

El viajante de comercio

Introducción

- El **problema del viajante de comercio** (*Travelling Salesman Problem*, o TSP):
  - Imaginemos a un vendedor que debe visitar una lista de ciudades exactamente una vez y regresar al punto de partida.
  - Su objetivo es encontrar la ruta más corta posible para minimizar el tiempo y el coste de viaje.

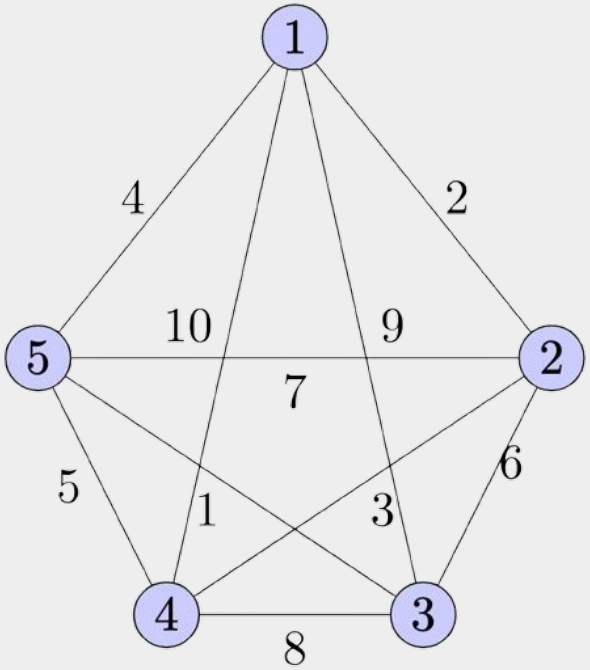
El viajante de comercio

Introducción

- Dado un grafo ponderado **g = (V, E)** con pesos no negativos, el problema es encontrar un ciclo hamiltoniano de mínimo coste.
  - Un **ciclo hamiltoniano** es un recorrido en el grafo que recorre todos los vértices sólo una vez y regresa al de partida.
  - El **coste** de un ciclo viene dado por la **suma de los pesos de las aristas** que lo componen.
  - Es posible que **no haya arista** entre dos nodos**.**

El viajante de comercio

Ejemplo



Ciclo de coste mínimo:

1 → 2 → 4 → 5 → 3 → 1

Coste:

2 + 3 + 5 + 1 + 9 = 20

El viajante de comercio

Formulación

- Antes de implementar, hay que considerar:
  - Vector solución
  - Bucle de expansión
  - Reusar cálculos
  - Cota optimista
  - Cota pesimista

El viajante de comercio

Formulación

- Antes de implementar, considerar:
  - Vector solución: x = |V| donde x[i] indica el nodo i-ésimo a visitar o x = [] e ir añadiendo.
  - Bucle de expansión: mantener un visitados booleano para evitar repetir
  - Reusar cálculos: llevar el coste del camino en la llamada
  - Cota optimista: Kruskal !!
  - Cota pesimista: voraz:
    - Voraz

Técnicas heurísticas

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Técnicas heurísticas

Introducción

- Familia de estrategias para abordar problemas **impracticables** con métodos convencionales o exactos.
- **No garantizan** encontrar la solución óptima a un problema
- Contienen estrategias para encontrar soluciones *suficientemente buenas* dentro de un margen *aceptable* de eficiencia.

Técnicas heurísticas

Introducción

- Aproximación práctica (no formal) a un problema:
  - Se basan en reglas prácticas o experimentales conocidas como **heurísticas**, que guían el proceso de búsqueda de soluciones.
  - Estas reglas se derivan del conocimiento del problema específico y la experiencia previa en dominios similares.
- Los métodos heurísticos no pueden estudiarse como un esquema general sino que deben conocerse uno a uno.

Técnicas clásicas

Técnicas heurísticas

Algoritmos aleatorios

- Estrategia simple: tomar decisiones **al azar**.
- En la mayoría de casos, no sólo no da la solución óptima sino que dará una **solución pobre**.
- **Ventaja**: complejidad lineal con respecto al tamaño de la solución.
- Se puede repetir *K* la ejecución del algoritmo, aumentando así la probabilidad de encontrar una buena solución

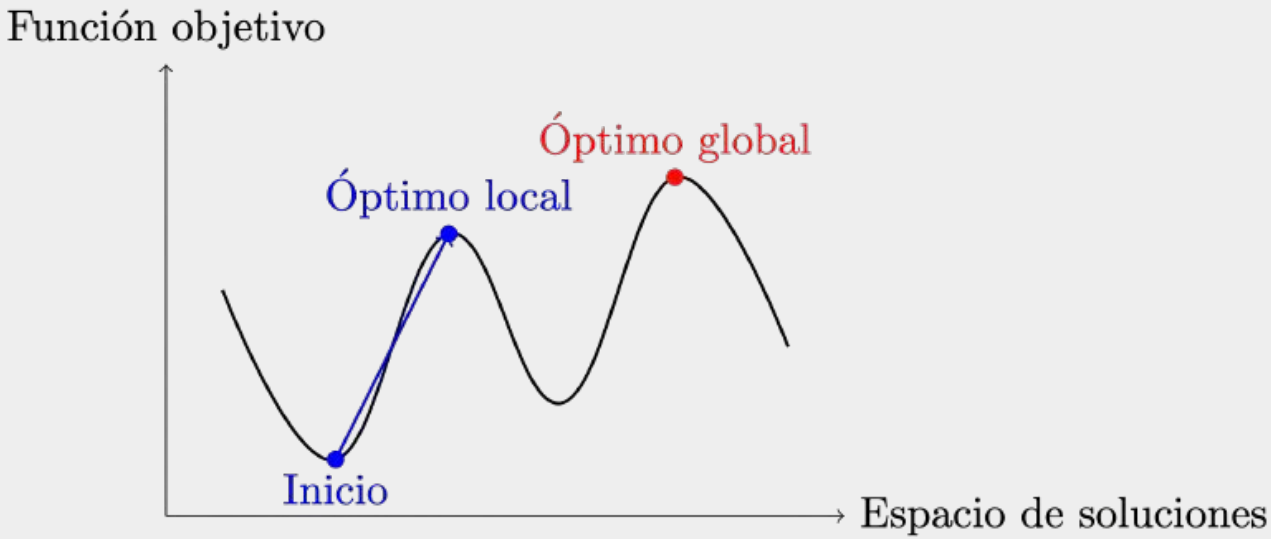
Técnicas heurísticas

Búsqueda local

- Exploración **iterativa** de las **soluciones vecinas** de una solución.
- Comienza con una solución inicial y se examinan las soluciones que están en su *vecindad*.
- Si se encuentra una **mejor solución** en esta vecindad, el algoritmo se *mueve* a ella y repite el proceso.
- Método simple y eficiente pero **muy sensible a óptimo locales**.

Técnicas heurísticas

Búsqueda local



Técnicas avanzadas

Técnicas heurísticas

Temple simulado (*Simulated Annealing*, SA)

- Acepta soluciones peores inicialmente para evitar quedar atrapado en óptimos locales.
- Gradualmente reduce la probabilidad de aceptar peores soluciones, acercándose al óptimo global.

Técnicas heurísticas

Búsqueda tabú (*Tabu Search*, TS)

- Búsqueda local con memoria adaptativa para evitar ciclos.
- Explora nuevas áreas del espacio de soluciones al prohibir revisitar ciertas soluciones.

Técnicas heurísticas

Optimización por enjambre de partículas (*Particle Swarm Optimization*, PSO)

- Basado en el movimiento colaborativo de partículas para optimizar funciones.
- Equilibrio entre la exploración del espacio de soluciones y la explotación de las mejores soluciones.

Técnicas heurísticas

Optimización por colonia de hormigas (*Ant Colony Optimization*, ACO)

- Inspirado en la búsqueda de caminos en colonias de hormigas, reforzando el paso por soluciones prometedoras.
- Uso de *feromonas* para guiar a futuras soluciones hacia opciones de mayor calidad.

Técnicas heurísticas

Algoritmos genéticos (*Genetic Algorithms*, GA)

- Optimización iterativa de poblaciones de soluciones hacia mejores resultados.
- Nuevas soluciones a través de procesos evolutivos como selección, cruce y mutación.

Técnicas heurísticas

Árbol de búsqueda de Monte Carlo (*Monte Carlo Tree Search*, MCTS)

- Toma de decisiones secuenciales mediante búsqueda en árbol y simulaciones aleatorias.
- Equilibrio entre la exploración de nuevas opciones y la explotación de las mejores conocidas.

Programación lineal

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Programación lineal

Introducción

- Optimizar un **objetivo lineal** sujeto a un conjunto de restricciones **lineales**.

Variables

$$x_1, x_2, \ldots, x_n \geq 0$$

Objetivo

$$Z = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

Restricciones

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

Programación lineal

Ejemplo

- Una empresa que produce dos productos A y B: **Variables**
  - El producto A genera 40 euros por unidad.
  - El producto B genera 30 euros por unidad. **Objetivo**
- La producción está limitada por la disponibilidad de recursos:
  - Cada unidad de A requiere 2 horas y 3 unidades de materia.
  - Cada unidad de B requiere 1 hora y 2 unidades de materia.
- Se dispone de 100 horas de trabajo y 180 unidades de materia prima.

Restricciones

Programación lineal

Ejemplo

Maximizar: $Z = 40x_1 + 30x_2$

Sujeto a:

$$2x_1 + x_2 \leq 100$$
$$3x_1 + 2x_2 \leq 180$$
$$x_1, x_2 \geq 0$$

Programación lineal

Interpretación geométrica

- En un espacio definido por *n* variables y *m* restricciones, las soluciones factibles forman un **politopo**.
- Cada vértice de este politopo representa una solución factible básica, que satisface exactamente *n* restricciones en igualdad.
- La función objetivo se representa como un plano que se desplaza sobre el politopo.

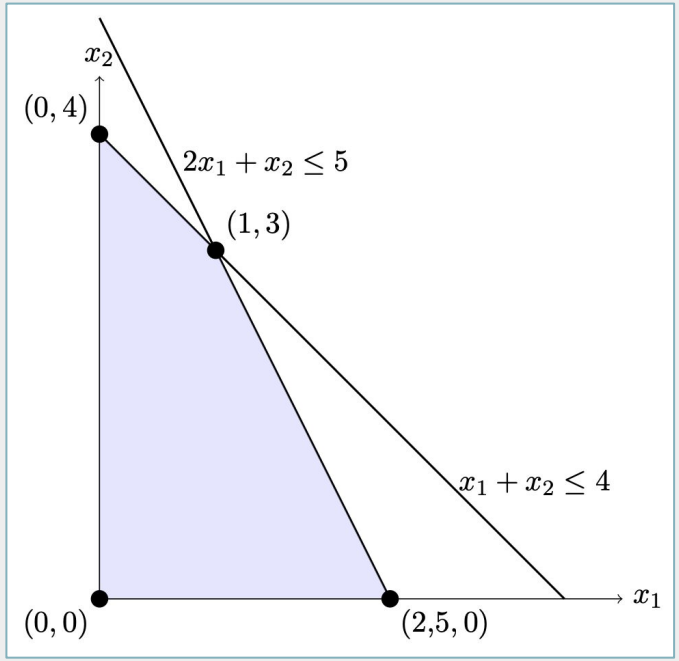
Programación lineal

Interpretación geométrica

$$Z = 3x_1 + 2x_2$$

$$x_1 + x_2 \leq 4$$

$$2x_1 + x_2 \leq 5$$



Programación lineal

Búsqueda de soluciones

- La programación lineal puede resolverse de manera exacta mediante algoritmos específicos.
- Se garantiza encontrar la solución óptima cuando existe e identifican cuando no la hay (inviable o no acotada).
- Vamos a introducir el método **Simplex**: el algoritmo más popular, ampliamente utilizado por su eficiencia práctica.

Simplex

Programación lineal

Método Simplex

- Desarrollado por George Dantzig en 1947.
- Recorrer las soluciones potencialmente óptimas y factibles del problema.
- Encuentra la solución óptima, de manera eficiente en la mayoría de los casos prácticos.

Programación lineal

Método Simplex

- Desde un punto de vista geométrico, el método Simplex recorre los vértices del politopo que forma el problema.
- Se mueve de manera iterativa entre los vértices hasta encontrar la solución óptima.

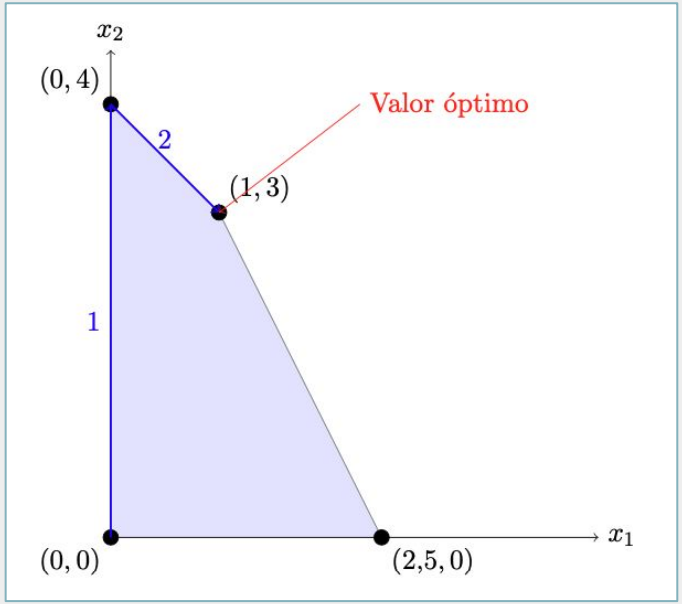
Programación lineal

Método Simplex

Maximizar: $Z = 3x_1 + 2x_2$

$$x_1 + x_2 \leq 4$$

$$2x_1 + x_2 \leq 5$$



Implementación

Programación lineal

Implementación

- El reto en programación lineal no radica en la resolución computacional del problema, sino en su correcta identificación.
- Existen multitud de librerías que implementan el método Simplex y otros algoritmos eficientes.
- Es esencial identificar la función objetivo, las variables de decisión y las restricciones que modelan el problema real.

Programación lineal

Implementación

$$\min c^T \mathbf{x}, \quad A\mathbf{x} \leq \mathbf{b}, \quad \mathbf{x} \geq 0$$

Maximizar: $Z = 3x_1 + 2x_2$

$$x_1 + x_2 \leq 4$$

$$2x_1 + x_2 \leq 5$$

```
from scipy.optimize import linprog

c = [-3, -2]

A = [[1, 1],    # x1 + x2 <= 4
     [2, 1]]    # 2x1 + x2 <= 5
b = [4, 5]

result = linprog(c, A_ub=A, b_ub=b, bounds=(0, None))

print("Valor óptimo:", result.fun)
print("Valores de las variables:", result.x)
```

Ejercicios

Programación lineal

Un nutricionista quiere diseñar una dieta utilizando dos alimentos: avena y manzanas. Cada ración de avena cuesta 2 euros y cada ración de manzanas 3 euros. Cada ración de avena aporta 3 unidades de proteína y 4 unidades de fibra, mientras que cada ración de manzanas aporta 2 unidades de proteína y 5 unidades de fibra. Se desea cubrir al menos 18 unidades de proteína y 20 unidades de fibra con el menor coste posible.

Programación lineal

Una empresa de marketing tiene un presupuesto de 15.000 € para anunciarse en redes, televisión y prensa. Cada euro gastado en redes llega a 40 personas, en televisión llega a 70 y en prensa a 50 personas. Por políticas internas se requiere que:

- Al menos 25% del presupuesto debe destinarse a redes sociales.
- El gasto en televisión no debe superar al gasto en prensa escrita.
- No se puede gastar más de 5,000 euros en prensa escrita.
- Plantea un modelo de programación lineal para maximizar el alcance publicitario de la empresa.

Programación lineal

Una fábrica produce un compuesto químico utilizando los ingredientes A, B y C. Cada kg del compuesto final debe contener al menos un 30 % de A, un 20 % de B y no más de un 50 % de C. El coste de cada ingrediente es, respectivamente, 5 €/kg, 8 €/kg y 3 €/kg. La fábrica necesita producir al menos 500 kg del compuesto final. Debido a limitaciones de inventario, solo hay disponible 300 kg de A, 200 kg de B y 400 kg de C. La fábrica vende el compuesto final a 12 €/kg y tiene contratos pendientes para entregar 400 kg del producto en las próximas dos semanas. Hay que determinar la cantidad de cada ingrediente que debe usar para minimizar el coste de producción.