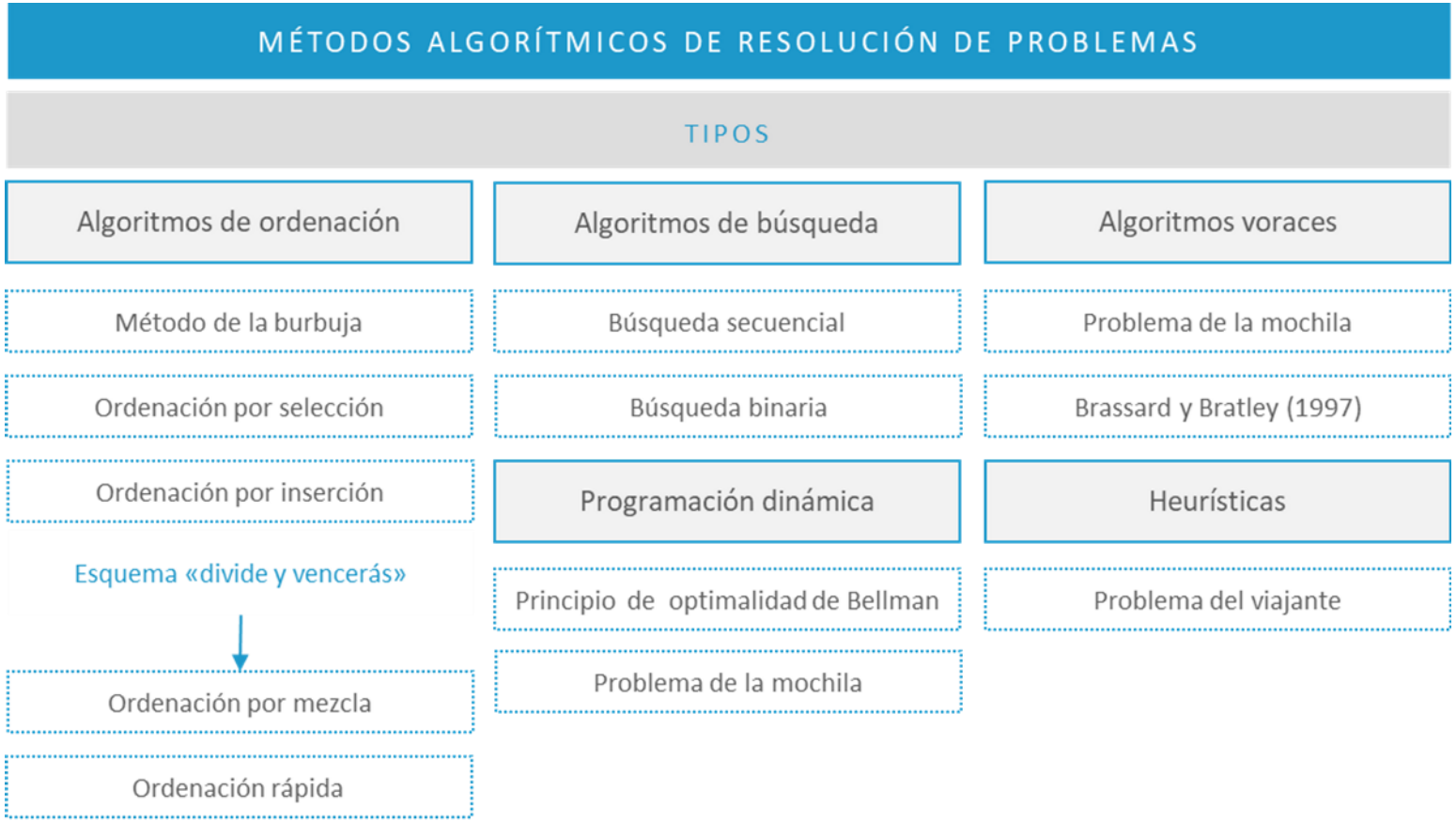
Programación Científica y HPC

Métodos algorítmicos de resolución de problemas

Índice



Esquema

Ideas clave

5.1. Introducción y objetivos

En este tema se presentarán algoritmos de distinta naturaleza y con distintas funcionalidades, que puedan ser aplicados sobre las estructuras de datos y los tipos abstractos de datos o sobre algunas propuestas nuevas. Se verá primero la conceptualización y el diseño del algoritmo y, posteriormente, su implementación en Python.

Antes de empezar, es importante conocer la etimología de la palabra *algoritmo*, que nos informará a su vez del origen. La palabra algoritmo proviene del nombre del matemático persa del siglo IX, al-Jwārizmī. Tal y como se concibe, un algoritmo no es más que un **conjunto de reglas que se aplican para realizar algún cálculo y obtener un resultado.** Estas se podrían aplicar a mano o en una máquina.

El algoritmo más famoso es el algoritmo de Euclides (siglo III a. C.) para el cálculo del máximo común divisor de dos números enteros.

En general, se debe tener en cuenta que **la ejecución de un algoritmo no requiere de decisiones subjetivas, ni de la intuición o la creatividad.** Un algoritmo **es un conjunto de reglas que se deben ejecutar de la forma y orden que se haya establecido.** Por tanto, se puede suponer que la aplicación de las reglas proporcionará una respuesta correcta.

El objetivo de usar algoritmos es el de dar solución a un problema. Sin embargo, hay problemas para los que no se conocen algoritmos de resolución aplicables y no queda otro remedio que tratar de encontrar algunas reglas que proporcionen una aproximación en un tiempo razonable de la respuesta correcta, aunque tampoco siempre es posible demostrar la bondad de la aproximación. Estos algoritmos, que se

conocen como algoritmos heurísticos, o simplemente se dice que es una heurística, tienen una base teórica mínima y se puede decir que están basados en el «optimismo».

Como ya se ha dicho anteriormente, **el objetivo de los algoritmos es la resolución de un problema,** pero no siempre existe un algoritmo único para resolverlo y, por tanto, se deberá decidir cual usar en cada caso. La selección puede depender de factores de tiempo de ejecución, de cantidad de espacio requerido o del tipo de los datos a los que se aplica.

Los algoritmos **se pueden clasificar por su funcionalidad,** que es la clasificación elegida para presentarlos en este tema, **o por su naturaleza.** De acuerdo con esta última y, que es la que se adopta en Brassard y Bratle (1997), los algoritmos se pueden clasificar en:

- **Algoritmos voraces:** con un enfoque sencillo y fáciles de implementar. Se utilizan fundamentalmente para resolver problemas de optimización.

- **Algoritmo «divide y vence»:** en realidad, es una técnica que descompone el problema en subproblemas que se resuelven de forma sucesiva e independiente a los otros subproblemas. Es un método de refinamiento progresivo en el que se pasa del problema completo a los subproblemas.

- **Algoritmos de programación dinámica:** esta es una técnica ascendente que comienza por los subproblemas más pequeños y, mediante la combinación de sus soluciones, se obtienen soluciones para problemas de tamaño cada vez mayor.

- **Algoritmos probabilistas:** procedimientos que efectúan elecciones aleatorias acerca de lo que se debe realizar para encontrar una solución. Aunque su fundamento parece que choca con la idea de algoritmo, que se concibe como un conjunto de reglas que se aplican de forma determinista, son aceptados como tales.

- **Algoritmos paralelos:** están pensados para la ejecución simultanea de partes del algoritmo sobre diferentes conjuntos de datos para mejorar su complejidad temporal.

- **Algoritmos sobre grafos:** Para resolver problemas que se pueden formular mediante grafos.

- **Algoritmos heurísticos:** procedimientos que pueden encontrar o no una solución y, en el caso de encontrarla, puede ser óptima, buena o no. Se aplica a problemas en los que es difícil, o incluso no se pueda resolver, mediante un algoritmo conocido.

En este tema se verán algunos **algoritmos según su funcionalidad, que puede ser ordenar y buscar,** entre otros. Para algunos **se presentarán distintas técnicas.** Se revisarán, así mismo, algunas técnicas de programación dinámica y heurísticas.

5.2. Algoritmos de ordenación

Método de la burbuja

Se trata del procedimiento de ordenación **más conocido.** Está considerado como uno de los algoritmos de ordenación **menos eficientes** y solo se recomienda para ordenar listas de datos pequeñas, de tamaño máximo 1 000.

Realiza múltiples pasadas por la lista de datos comparando el dato siguiente y, si no están ordenados, los intercambia. Tras cada pasada, el dato de valor más grande queda en su lugar.

Cada pasada recorrerá un elemento menos. En la Tabla 1 se muestra la ejecución de la primera pasada, en la que queda al final el dato más grande.

Listas						Hay intercambio
35	36	17	73	8	0	No
35	36	17	73	8	0	Sí
35	36	17	73	8	0	No
35	17	36	73	8	0	Sí
35	17	36	8	73	0	Sí
35	17	36	8	0	73	El 73 queda ordenado

Tabla 1. Primera pasada del algoritmo. Fuente: elaboración propia.

A continuación, se muestra el código en Python:

```
from time import time
def ordenacionBurbuja(lista):
    'Variable global que puede ser usada fuera  de la función'
    global numComparaciones
    n = len(lista)
    for i in range(1, n):
        for j in range(n-i):
            numComparaciones += 1
            if lista[j] > lista[j+1]:
                'intercambio de valores'
                lista[j], lista[j+1] = lista[j+1], lista[j]

lista = [35, 36, 17, 73,8, 0]
numComparaciones = 0
t0 = time()
ordenacionBurbuja(lista)
t1 = time()
print ("Lista ordenada:")
print(lista)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
print ("Comparaciones:", numComparaciones)
----------------------------------------------------------------
Ejecución:
 Lista ordenada:
[0, 8, 16, 17, 35, 73]
Tiempo: 0.000008 segundos
Comparaciones: 15
```

Figura 1. Código y resultado del algoritmo de ordenación burbuja. Fuente: elaboración propia.

Vean el uso de la declaración de una variable global usada por la función, pero que pertenece al ámbito global y, por tanto, se puede obtener su valor fuera de ella tras la ejecución. Esto es un recurso que no se considera de muy buena

práctica en programación, pero es útil. Miren también el uso del intercambio, en el que se hacen dos asignaciones a la vez

Su complejidad en tiempo es de orden $O(n^2)$, siendo n el número de elementos que se van a ordenar. Las instrucciones del bucle interno se ejecutan $n - i$ veces.

Ordenación por selección

Se basa en la **selección sucesiva de los valores mínimos.** En cada iteración del algoritmo se selecciona el elemento mínimo de los no ordenados y se intercambia con el primero. Tras cada iteración se conseguirá ordenar un elemento.

En la Tabla 2 se muestra la ejecución del algoritmo, en donde cada fila representa una iteración.

Listas						Comparaciones/ intercambios
35	36	17	73	8	0	5/Sí
0	36	17	73	8	35	4/Sí
0	8	17	73	36	35	3/No
0	8	17	73	36	35	2/Sí
0	8	17	35	36	73	1/Sí
0	8	17	35	36	73	Fin

Tabla 2. Iteraciones de la ordenación por selección. Fuente: elaboración propia.

A continuación, se muestra el código en Python:

```
from time import time
def ordenacionSeleccion(lista):
    global numComparaciones
    n = len(lista)
    for i in range(n - 1):
        minimo = i
        for j in range(i + 1, n):
            numComparaciones += 1
            if lista[j] < lista[minimo]:
                minimo = j
        lista[i], lista[minimo] = lista[minimo], lista[i]

lista = [35, 36, 16, 17, 73,8, 0, 50, 62, 4]
numComparaciones = 0
t0 = time()
ordenacionSeleccion(lista)
t1 = time()
print ("Lista ordenada:")
print(lista)

print ("Tiempo: {0:f} segundos".format(t1 - t0))
print ("Comparaciones:", numComparaciones)
------------------------------------------------------------
Ejecución:
Lista ordenada:
[0, 4, 8, 16, 17, 35, 36, 50, 62, 73]
Tiempo: 0.000014 segundos
Comparaciones: 45
```

Figura 2. Código y resultado del algoritmo de ordenación por selección. Fuente: elaboración propia.

Su complejidad en tiempo es de orden $O(n^2)$, siendo n el número de elementos que se van a ordenar. Las instrucciones del bucle interno se ejecutan $n - i$ veces.

Ordenación por inserción

Se basa en **ir examinando todos los elementos desde el segundo hasta el último** e insertarlo en el lugar adecuado entre sus predecesores.

En la tabla 3 se muestra la ejecución del algoritmo, donde cada fila representa una iteración.

Lista de números						Comparaciones/ intercambio
Iteración 1						
35	36	17	73	8	0	0/No
Iteración 2						
35	36	17	73	8	0	1/Si
35	17	36	73	8	0	1/Si
Iteración 3						
17	35	36	73	8	0	0/No
Iteración 4						
17	35	36	73	8	0	1/Si
17	35	36	8	73	0	1/Si
17	35	8	36	73	0	1/Si
17	8	35	26	73	0	1/Si
Iteración 5						
8	17	35	36	73	0	1/Si
8	17	35	36	0	73	1/Si
8	17	35	0	36	73	1/Si
8	17	0	35	36	73	1/Si
8	0	17	35	36	73	1/Si
0	8	17	35	36	73	Fin

Tabla 3. Una iteración del algoritmo por inserción. Fuente: elaboración propia.

A continuación, se muestra el código en Python:

```
from time import time
def ordenacionInsercion(lista):
    n = len(lista)
    global numComparaciones
    for i in range(1, n):
        val = lista[i]
        j = i
        while j > 0 and lista[j-1] > val:
            lista[j] = lista[j-1]
            j -= 1
            numComparaciones += 1
        lista[j] = val

lista = [35, 36, 17, 73,8, 0]
numComparaciones = 0
t0 = time()
ordenacionInsercion(lista)
t1 = time()
print ("Lista ordenada:")
print(lista)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
print ("Comparaciones:", numComparaciones)
-------------------------------------------------
Ejecución:
Lista ordenada:
[0, 8, 17, 35, 36, 73]
Tiempo: 0.001783 segundos
Comparaciones: 11
```

Figura 3. Código y resultado del algoritmo de ordenación por inserción. Fuente: elaboración propia.

Su complejidad en tiempo es de orden $O(n^2)$, siendo n el número de elementos que se van a ordenar. Las instrucciones del bucle interno se ejecutan i veces en el peor de los casos.

Ahora se verán ahora dos **algoritmos de ordenación que aplican un esquema de «divide y vencerás»** para realizar la ordenación

Ordenación por mezcla (*mergesort*)

Consiste en **dividir la lista en dos sub-listas de tamaño similar y ordenar cada una de ellas usando la recursividad.** Posteriormente, se mezclan las dos sub-listas ordenadas manteniendo el orden.

La mezcla de las dos sub-listas es más eficiente, si se cuenta con un espacio adicional que se pueda usar como centinela.

Lista de números						
35	36	17	73	8	0	
La lista se divide en dos						
35	36	17		73	8	0
Se hace llamada recursiva para ordenar cada una de las partes						
17	35	36		0	8	73
Se mezclan						
0	8	17	35	36	73	

Tabla 4. Interacciones de la ordenación por mezclas. Fuente: elaboración propia.

A continuación, se muestra el código en Python:

```
from time import time
import math
def mergeSort(lista):
    if len(lista) <= 1:
        return lista
    medio = len(lista) // 2
    izquierda = lista[:medio]
    derecha = lista[medio:]
    izquierda = mergeSort(izquierda)
    derecha = mergeSort(derecha)
    return merge(izquierda, derecha)

def merge(listaA, listaB):
    global numComparaciones
    lista = [math.inf for _ in range(len(listaA)+len(listaB))]
    listaA.append(math.inf)
    listaB.append(math.inf)
    i=0
    j=0
    for k in range(0,len(lista)):
        numComparaciones+=1
        if listaA[i]<listaB[j]:
            lista[k]=listaA[i]
            i+=1
        else:
            lista[k]=listaB[j]
            j+=1
    return lista

lista = [35, 36, 17, 73,8, 0]
numComparaciones = 0
t0 = time()
lista = mergeSort(lista)
t1 = time()
print ("Lista ordenada:")
print(lista)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
print ("Comparaciones:", numComparaciones)
-----------------------------------------------
Ejecución:
 Lista ordenada:
[0, 8, 17, 35, 36, 73]
Tiempo: 0.013349 segundos
Comparaciones: 16
```

Figura 4. Código y resultado del algoritmo de ordenación por mezcla. Fuente: elaboración propia.

Para analizar la complejidad del algoritmo se debe considerar que **la división en dos matrices requiere tiempo lineal.** Fusionar también requiere tiempo lineal, de forma que $(n) = t\left(\frac{n}{2}\right) + t\left(\frac{n}{2}\right) + g(n)$ donde $g(n) \in \Theta(n)$.

Por tanto, se produce una recurrencia de la forma $t(n) = 2t\left(\frac{n}{2}\right) + g(n)$.

Para resolver la recurrencia se aplica la ecuación *(1)* de resolución de recurrencias para $l = 2, b = 2, k = 1$.

$$(n) \in \begin{cases} \Theta(n^k) & si\ l < b^k \\ \Theta(n^k \log n) & si\ l = b^k \\ \Theta(n^{\log_b l}) & si\ l > b^k \end{cases}, siendo\ t(n) = lt\left(\frac{n}{b}\right) + g(n), \quad (1)$$

$$si\ existe\ un\ entero\ k\ tal\ que\ g(n) \in \Theta(n^k)$$

Entonces, su complejidad en tiempo es de orden $\Theta(nlog(n))$, siendo n el número de elementos que se van a ordenar.

Ordenación rápida (*quicksort*)

Consiste en **dividir la lista en dos sub-listas de tamaño similar.** La mayor parte del trabajo no recursivo se invierte en crear las sub-listas y no en la combinación de ellas.

En la primera etapa, el algoritmo selecciona un elemento que se conoce como **pivote,** la lista se parte en dos y se desplazan los elementos menores que el pivote hacia la izquierda de este y los mayores hacia la derecha. Cada una de **las partes se ordenan mediante llamadas recursivas al algoritmo y se obtiene una lista ordenada.**

La mezcla de las dos sub-listas es más eficiente si se cuenta con un espacio adicional que se pueda usar como centinela.

5.3. Algoritmos de búsqueda

Búsqueda secuencial

Busca un elemento dentro de una lista comenzando por el primero de ellos. Luego, recorre la lista hasta que lo encuentra o hasta que no quedan más elementos por consultar.

En la Figura 5 se muestra la versión iterativa del algoritmo.

```
from time import time
def busquedaSecuencial(lista, x):
    for i in range(len(lista)):
        if lista[i] == x:
            return i
    return -1

lista=[1, 2, 32, 8, 17, 19, 42, 13, 0]
print(lista)
t0 = time()
indice = busquedaSecuencial(lista,42)
t1 = time()
print ("Indice encontrado:")
print(indice)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
-------------------------------------------------
Ejecución:
[1, 2, 32, 8, 17, 19, 42, 13, 0]
Indice encontrado:
3
Tiempo: 0.000003 segundos
```

Figura 5. Código y resultado del algoritmo de búsqueda secuencial. Fuente: elaboración propia.

Este algoritmo requiere, en el peor de los casos, un tiempo que está en $\Theta(i)$, donde i es el índice en el que está el elemento. Por tanto, está en $O(n)$ en el peor de los casos, siendo n el número de elementos de la lista y en $O(1)$ en el mejor caso.

Como el número medio de iteraciones es $n + 1/2$, esta búsqueda requiere un tiempo promedio que está en $\Theta(n)$.

La búsqueda mejorará, si la lista está ordenada y si se puede determinar si el elemento está en la primera mitad o en la segunda mitad de la lista. En esto se basa el algoritmo de búsqueda binaria.

Búsqueda binaria

Es uno **de los más sencillos** en el que se aplica una técnica de «divide y vencerás». Parte de la base que la lista debe estar ordenada y va reduciendo el espacio o sección de búsqueda descartando segmentos de ese espacio en los que es seguro que el valor no va a estar. Inicialmente, el espacio de búsqueda es la lista total.

Se tendrán en cuenta las siguientes consideraciones en cada ejecución del algoritmo:

- Se selecciona el valor central de la sección de búsqueda y si es el valor buscado se retorna su índice.
- Si el valor central es mayor que el buscado, se descarta la parte de derecha de la sección y la búsqueda se limita a la parte izquierda.
- Si el valor central es menor que el buscado, se descarta la parte de izquierda de la sección y la búsqueda se limita a la parte derecha.
- Si la sección de búsqueda tiene longitud 0, significa que el valor no se encuentra en la lista y devuelve un índice que no pueda existir, en el caso de Python-1.

```
from time import time
def busquedaBinariaRecursiva(lista,x):
    if len(lista)==0:
        return -1
    else:
        return busquedabinariarec(lista,x,0,len(lista)-1)

def busquedabinariarec(lista,x, primero, ultimo):
    if primero>ultimo:
        return -1;
    medio = (primero+ultimo) // 2
    if lista[medio] < x:
        return busquedabinariarec(lista,x,medio+1,ultimo)
    elif (lista[medio] > x):
        return busquedabinariarec(lista,x,primero,medio-1)
    else:
        return medio

lista=[1,3,5,7,9]
print(lista)
t0 = time()
indice = busquedaBinariaRecursiva(lista,3)
t1 = time()
print ("Indice encontrado:")
print(indice)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
------------------------------------------------------------
Ejecución:
[1, 3, 5, 7, 9]
Indice encontrado:
1
Tiempo: 0.000004 segundos
```

Figura 6. Código y ejecución del algoritmo recursivo de búsqueda binaria. Fuente: elaboración propia.

El tiempo que requiere una llamada recursiva de este algoritmo es $t(m)$ donde $m = ultimo - primero + 1$, que es el número de elementos que quedan para la búsqueda, y el tiempo requerido para la primera llamada es $t(n)$, que es el tamaño de la lista.

Por tanto, cuando el número de elementos que quedan por buscar en una llamada recursiva es mayor que uno requiere una cantidad constante de tiempo, además de la llamada recursiva con $m/2$. Esto quiere decir que $t(m) = t\left(\frac{m}{2}\right) + g(m)$ con $g(m) \in O(1)$. Aplicando la ecuación (1) vista anteriormente con $l = 1, b = 2\,, k = 0$ se concluye que $t(m) \in \Theta(\log m)$. Luego, este algoritmo requiere un tiempo logarítmico, incluso en el caso mejor.

5.4. Algoritmos voraces

Se utilizan, fundamentalmente, **para resolver problemas de optimización.** Las características relevantes de estos problemas son:

- Se cuenta con un conjunto de candidatos a ser solución del problema.
  - En cada paso un candidato se incorpora al conjunto de seleccionados o conjuntos de los rechazados, que se empiezan a construir en la primera etapa del algoritmo.

- Existe una función que comprueba si algo es solución.

- Existe una función que comprueba si se ha encontrado una solución factible, que será un subconjunto de los candidatos que satisfacen ciertas restricciones.

- Existe una función de selección que permite escoger un nuevo candidato de los que quedan por comprobar.

- Se debe obtener la solución factible, que se denomina solución óptima, la cual maximiza o minimiza una función objetivo.

En Brassard y Bratley (1997) se muestra el pseudocódigo de un algoritmo voraz.

```
función voraz(C:conjunto):conjunto
  {C es el conjunto de candidatos}
  S←Ø{S será el conjunto solucion}
  mientras C≠Ø y no solución(S) hacer
                                x←seleccionar(C)
                                C←C-{x}
                                si factible(SU{x}) entonces S←SU{x}
  si solución(S) entonces devolver S
                                                    sino devolver "no hay soluciones"
```

Figura 7. Pseudocódigo algoritmo voraz. Fuente: basado en Brassard y Bratley, 1997.

El objetivo de este problema es llenar una mochila, de forma que se maximice el valor de los objetos transportados de acuerdo con la capacidad de esta. Los objetos cuentan con un peso y un valor.

5.5. Programación dinámica

Es una técnica ascendente, estos algoritmos comienzan resolviendo los casos más sencillos y combinan las soluciones para resolver casos cada vez mayores hasta obtener la solución del problema original.

El modelado de problemas de este tipo no es el mismo para todos, para cada problema será necesario especificar sus componentes y fases.

Se basa en el principio de optimalidad de Bellman. Este principio se enuncia de esta forma en el diccionario de la Real Academia de Ingeniería (s. f.):

> «Principio aplicado en programación dinámica que consiste en que una secuencia óptima de decisiones que resuelve un problema debe cumplir la propiedad de que cualquier subsecuencia de decisiones, que tenga el mismo estado final, debe ser también óptima respecto al subproblema correspondiente».

Este principio no es aplicable a cualquier problema y, por tanto, se debe considerar la posibilidad de que no sea posible resolver el problema mediante programación dinámica.

Las características relevantes de la programación dinámica son:

- Las decisiones se toman en secuencia.

- El problema se divide en etapas que dependen de una política de decisión.
- Los datos que se necesitan conocer para describir cada etapa son pocos.
- Los resultados dependen de una serie de variables.
- En cada capa se cumplen una serie de condiciones que determina el estado del sistema asociado.
- En cada etapa, la decisión modifica los valores de las variables relacionadas y transforman el estado de la etapa actual, en el estado de la siguiente etapa. Sin embargo, no aumentan ni disminuye, los factores de los que depende la solución.
- El diseño del procedimiento basado en la política de decisión está pensado para encontrar una solución óptima.
- Se resuelven todos los subcasos, para determinar los que son relevantes. Una vez que se determinan, se combinan para encontrar la solución óptima del problema original.

Un problema ejemplo sencillo de este tipo es el cálculo de los coeficientes de la potencia de un binomio.

Se sabe que los coeficientes son números combinatorios que se calculan para una fila del triángulo, a partir de números de la fila superior, entre los que se encuentra comprendido el número que se calcula.

La fórmula de cálculo es:

$$\binom{n}{k} = \begin{cases} 1 & si\ k = 0\ \acute{o}\ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & si\ 0 < k < n \\ 0 & en\ otro\ caso \end{cases}$$

La función que lo calcularía es:

```
def numCombinatorio(n,k):
    if k==0 or  k==n:
        return 1
    else:
        return numCombinatorio(n-1, k-1)+numCombinatorio(n-1, k)
```

Figura 8. Función para calcular un número combinatorio. Fuente: elaboración propia.

Nótese que para un número combinatorio cualquiera será necesario, calcular todos los anteriores, calculándose algunos resultados varias veces. Por ejemplo, para calcular `numCombinatorio(6,4)` se necesita calcular `numCombinatorio(5,3)` y `numCombinatorio(5,4)`, pero cada uno de ellos necesita calcular `numCombinatorio(4,3)`, que se calculará más de una vez. Su complejidad está en el orden de $\Omega\left(\frac{n}{k}\right)$. Se podría realizar un algoritmo más eficiente mediante el cálculo del triángulo de Pascal con una tabla que se iría llenando línea por línea.

Características de las técnicas **divide y vencerás y programación dinámica.** Análisis de las técnicas mediante el ejemplo de aplicación de la sucesión de Fibonacci.

Accede al vídeo

5.6. Heurísticas

Son procedimientos que pueden producir o no una buena solución para un problema, es decir, **son procedimientos que puede producir una solución óptima**, pero que no es la mejor o, incluso, no encuentra una solución.

Estos procedimientos se utilizan en casos de problemas en los que puede no existir un algoritmo que lo resuelva y priman el tiempo de respuesta sobre la optimización de la solución.

Existen heurísticas deterministas o probabilistas. En cualquier caso, **se suelen aplicar a problemas difíciles de resolver,** que se suelen conocer como problemas *NP-hard*.

Según Rego *et al*. (2011) «un método heurístico es un procedimiento para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución».

Se muestra a continuación un ejemplo de un problema clásico para el que no se ha encontrado un algoritmo que pueda resolverlo en tiempo al menos polinómico.

Problema del viajante

En el problema del viajante **se cuenta con una serie de ciudades para las que se conocen las distancias entre ellas. El viajante deberá salir de una de esas ciudades y visitar todas las demás una vez, retornando al punto de partida y habiendo recorrido la menor distancia posible.** Se debe tener en cuenta, que encontrar los métodos exactos para este problema necesitan un tiempo exponencial, por lo que no son viables para un número grande de nodos.

En este caso, se trata del recorrido de un grafo del que se conoce el origen.

Se puede usar un grafo completo $G = (V, E)$ no dirigido con tantos nodos en V como ciudades. Se deberá obtener un ciclo hamiltoniano de mínimo coste. Un ciclo hamiltoniano es un ciclo que contiene todos los nodos de un grafo una sola vez.

Existen varias formas de tratarlo mediante métodos heurísticos. Una de las más habituales será la forma iterativa, usando un algoritmo voraz. En cada iteración del

algoritmo se seleccionaría la arista con menor distancia que no se haya considerado todavía y que cumpla:

- No formar un ciclo con las aristas seleccionadas, excepto si es la última iteración.
- No es la tercera arista que llega a un mismo vértice de entre los seleccionados.

La Figura 9 muestra el grafo de distancias entre seis ciudades. Se han numerado las ciudades o vértices a partir de 0 para que se entienda mejor la implementación.

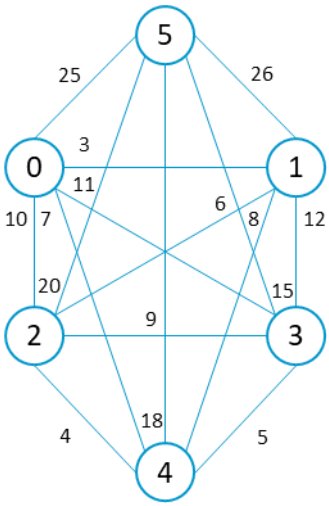


Figura 9. Grafo distancias entre ciudades. Fuente: elaboración propia.

Su representación en forma de matriz sería:

	1	2	3	4	5
Vértice 0	3	10	11	7	25
Vértice 1		6	12	8	26
Vértice 2			9	4	20
Vértice 3				5	15
Vértice 4					18

Tabla 5. Representación matricial del grafo del problema del viajante. Fuente: elaboración propia.

Según el algoritmo, se necesitan ordenar las aristas de menor a mayor distancia y se seleccionarían {0,1},{2,4},{3,4},{1,2}.

- Por valor tocaría la {0,4}, pero no se selecciona porque no cumple ninguna de las dos condiciones indicadas, ya que formaría un ciclo completo con las anteriores y sería la tercera arista que llega al vértice 4.
- El siguiente que sería el {1,4} que no cierra el ciclo, pero si fuera la tercera arista que llega al 4, tampoco se coge.
- El siguiente en valor {2,3} tampoco se selecciona por el mismo motivo que el anterior, en este caso para el vértice 2.
- El {0,2} no se selecciona porque no cumple ninguna de las dos condiciones y el {0,3} cierra ciclo, así que tampoco se selecciona.
- El {1,3} visitaría por tercera vez el vértice 1.

Se elegiría entonces el {3,5} y ya no se podría seleccionar ninguno más, excepto el {0,5} que cerraría el ciclo (este caso se permite por ser el último).

En este algoritmo **se reorganizarían las aristas**, que son simétricas, **para mostrar el camino enlazado,** es decir, se muestra el ciclo correcto y se obtiene el siguiente orden de aristas:

$$\{0,1\},\{1,2\},\{2,4\},\{4,3\},\{3,5\},\{5,0\}$$

el ciclo $[0,1,2,4,3,5,0]$ y la distancia total 58.

Esta solución es bastante buena, pero no es óptima, de hecho, existe un recorrido donde la distancia es 56 y sería:

$$\{0,1\},\{1,2\},\{2,5\},\{5,3\},\{3,4\},\{4,0\}$$

```
from grafnodirigido import GrafoNoDirigido
import ordenacionInserciongenerico
class Viajante:
    def __init__(self,numNodos,listaDistancias=[]):
        self.grafo=GrafoNoDirigido(numNodos)
        """grafo no dirigido, en su matriz de adyacencia"""
        """ almacena ternas, origen,destino"""
        """distancia. Es una matriz triangular"""
        for i in range(numNodos-1):
            for j in range(i+1,numNodos):
                self.grafo.matrizady[i][j]=[i,j,listaDistancias[i][j]]
        self.visitas=[0]*numNodos
        """indica las veces que se ha visitado un vértice"""
        self.recorrido=list()
        """almacena las ternas de recorrido"""
        self.distancia=0
        """distancia recorrida"""
        self.viaje=list()
        """almacena el ciclo sin datos adicionales"""

    @staticmethod
    def compararTerna(terna1,terna2):
        """metodo de comparación de distancias que permite"""
        """ordenar de menor a mayor"""
        return terna1[2]>terna2[2]

    def ordenarAristas(self):
        """metodo que ordena la lista de ternas creada"""
        """a partir de la matriz ordena por inserción, """
        """algoritmo al que se le pasa el método de comparación"""
        listaAux=list()
        for i in range(self.grafo.numNodos):
            for j in range(i+1,self.grafo.numNodos):
                listaAux.append(self.grafo.matrizady[i][j])
        ordenacionInserciongenerico.ordenacionInsercion(listaAux, Viajante.compararTerna)
        return listaAux

    def comprobarTercera(self,arista):
        """comprueba si los vértices de la arista ya han sido visitados dos veces"""
        return self.visitas[arista[0]]==2  or self.visitas[arista[1]]==2
```

Figura 10. Código y ejecución de una heurística voraz para el problema del viajante. Fuente: elaboración propia.

```
def generaCiclo(self,arista):
        """Comprueba si una arista cierra ciclo"""
        return arista[0]==0 or arista[1]==0

    def buscarSiguiente(self,nodo):
        """Devuelve siguiente arista cuando se construye ciclo final"""
        for arista in self.recorrido:
            if arista[0]==nodo:
                self.recorrido.remove(arista)
                return arista
            elif arista[1]==nodo:
                self.recorrido.remove(arista)
                arista[0],arista[1]=arista[1],arista[0]
                return arista
        return -1

    def esUltimo(self,arista):
        """Busca la última arista que cierra el ciclo"""
        ultimodestino=self.recorrido[-1][1]
        return (arista[0]==ultimodestino  or arista[1]==ultimodestino) and (arista[0]==0 or arista[1]==0)

    def generarCiclo(self):
        """Construye el ciclo final ordenando aristas seleccionadas"""
        recorridoAux=list()
        recorridoAux.append(self.recorrido[0])
        self.viaje.append(0)
        self.recorrido.pop(0)

        while len(self.recorrido)>0:
            nodoOrig=recorridoAux[-1][1]
            aristaSig=self.buscarSiguiente(nodoOrig)
            recorridoAux.append(aristaSig)
            self.viaje.append(aristaSig[0])
        self.recorrido=recorridoAux
```

Figura 10. Código y ejecución de una heurística voraz para el problema del viajante. Fuente: elaboración propia.

```
def heuristica(self):
        """Método heurístico aplicado"""
        primera=self.grafo.matrizady[0][1]
        """primera arista desde el origen conocido"""
        self.recorrido.append(primera)
        self.visitas[0]+=1
        self.visitas[1]+=1
        listaAristas=self.ordenarAristas()
        self.distancia+=primera[2]

        """for que selecciona las aristas"""
        for arista in listaAristas:
            if  not arista in self.recorrido and not self.generaCiclo(arista) and not self.comprobarTercera(arista):
                primero=arista[0]
                segundo=arista[1]
                self.recorrido.append(arista)
                self.visitas[primero]+=1
                self.visitas[segundo]+=1
                self.distancia+=arista[2]
        self.generarCiclo()
        """añade la última arista para cerrar"""
        for arista in listaAristas:
           if not arista in self.recorrido and self.esUltimo(arista):
               if arista[0]==self.recorrido[-1][1]:
                   self.recorrido.append(arista)
                   break;
               else:
                   arista[0],arista[1]=arista[1],arista[0]
                   self.recorrido.append(arista)
                   break;
        self.distancia+=arista[2]
        self.viaje.append(arista[0])
        self.viaje.append(0
```

Figura 10. Código y ejecución de una heurística voraz para el problema del viajante. Fuente: elaboración propia.

```
listad=[[0,3,10,11,7,25],[0,0,6,12,8,26],[0,0,0,9,4,20],[0,0,0,0,5,15],[0,0,0,0,0,18]]
viajante=Viajante(6,listad)
viajante.heuristica()
print("Las aristas seleccionadas son: ")
print(viajante.recorrido)
print("El viaje es :")
print(viajante.viaje)
print("La distancia recorrida es: ")
print(viajante.distancia)
--------------------------------------------------------------------------------
Ejecución:
Las aristas seleccionadas son:
[[0, 1, 3], [1, 2, 6], [2, 4, 4], [4, 3, 5], [3, 5, 15], [5, 0, 25]]
El viaje es :
[0, 1, 2, 4, 3, 5, 0]
La distancia recorrida es:
58
```

Figura 10. Código y ejecución de una heurística voraz para el problema del viajante. Fuente: elaboración propia.

5.7. Referencias bibliográficas

Brassard, G. y Bratley, P. (1997). *Fundamentos de algoritmia* (1ra ed.). Prentice Hall.

Real Academia de Ingeniería. (s. f.). Principio de optimalidad de Bellman En *Diccionario de la Real Academia de Ingeniería*. Recuperado en 3 de abril de 2021, de http://diccionario.raing.es/es/lema/principio-de-optimalidad-de-bellman

Rego, C., Gamboa, D., Glover, F. y Osterman, C. (2011). Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research, 211*(3), pp. 427-441. https://doi.org/10.1016/j.ejor.2010.09.010

5.8. Cuaderno de ejercicios

Ejercicio 1. Ordenación rápida (*quicksort*)

En la Tabla 6 se muestra el estado de la lista [35,36,17,73,8,0] en la primera etapa. En esta, se toma como pivote 35 y se van desplazando hacia la izquierda los elementos menores que 35, y hacia la derecha los elementos mayores que 35. Al final se coloca el pivote en su sitio.

El proceso continúa, aplicando el algoritmo nuevamente a las sublistas de menores y mayores

Lista de números						
35	36	17	73	8	0	i en 36>pivote j en 0< pivote
35	0	17	73	8	36	i en 73>pivote j en 8<pivote
35	0	17	8	73	36	Se coloca el pivote en su sitio
0	17	8	35	73	36	Se procede con cada sublista de la misma forma

Tabla 6. Estado de la lista en las distintas etapas del algoritmo. Fuente: elaboración propia.

A continuación, se muestra el código en Python.

```
from time import time
def obtenerPivote(lista):
    pivote = lista[0]
    i=0
    j=len(lista)-1
    while i<j:
        while lista[i]<=pivote and i<j:
            i+=1
        while lista[j]>pivote and j>=0:
            j-=1
        if i<j:
            lista[j],lista[i]=lista[i],lista[j]
    lista[0]=lista[j]
    lista[j]=pivote
    return j

def quicksort(lista):
    if len(lista)<=1:
        return lista
    else:
        l=obtenerPivote(lista)
        l+=1
        menores=lista[:l]
        mayores=lista[l:]
        lista=quicksort(menores)+quicksort(mayores)
        return  lista

lista = [35, 36, 17, 73,8, 0]
t0 = time()
lista=quicksort(lista)
t1 = time()
print ("Lista ordenada:")
print (lista)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
-----------------------------------------------------
Ejecución:
Lista ordenada:
[0, 8, 17, 35, 36, 73]
Tiempo: 0.000015 segundos
```

Figura 11. Implementación y prueba del ejercicio 1. Algoritmo *quicksort*. Fuente: elaboración propia.

Este algoritmo es ineficiente, si en las llamadas recursivas se produce, de forma sistemática, que los subcasos están desequilibrados. En el peor caso requiere un tiempo cuadrático, sin embargo, en el caso medio, su complejidad en tiempo es de orden $\Theta(nlog(n))$, siendo n el número de elementos que se van a ordenar. Esto se podría demostrar.

Brassard y Bratley (1997) proporcionan esta demostración a partir de la resolución de recurrencias.

Ejercicio 2. Implementación de la búsqueda secuencial de forma recursiva

Solución:

En este ejercicio se muestra una implementación recursiva de la búsqueda secuencial. Para ello, se invoca a la función que busca el elemento pasando en cada llamada como primer elemento, el siguiente elemento al último consultado en la llamada anterior.

```
from time import time
def busquedaSecuencialRecursiva(lista,x):
    if len(lista)==0:
        return -1
    else:
        return busquedasecuencialrec(lista,x,0,len(lista)-1)

def busquedasecuencialrec(lista,x,primero,ultimo):
    if primero>ultimo:
        return -1
    elif lista[primero]==x:
        return primero
    else:
        return busquedasecuencialrec(lista,x,primero+1,ultimo)

lista= [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(lista)
t0 = time()
indice = busquedaSecuencialRecursiva(lista,8)
t1 = time()
print ("Indice encontrado:")
print(indice)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
------------------------------------------------------------
Ejecución:
[1, 2, 32, 8, 17, 19, 42, 13, 0]
Indice encontrado:
3
Tiempo: 0.000005 segundos
```

Figura 12. Implementación ejercicio 2. Búsqueda secuencial recursiva. Fuente: elaboración propia.

Ejercicio 3. Implementación de la búsqueda binaria de forma iterativa

Solución:

```
from time import time
def busquedaBinaria(lista, x):
    primero=0
    ultimo=len(lista)-1
    if x>lista[ultimo] or x<lista[primero]:
        return -1
    while primero<=ultimo:
        medio=(primero+ultimo)//2
        if lista[medio]< x:
            primero=medio+1
        elif lista[medio]==x:
            return medio
        else:
            ultimo=medio

lista=[1,3,5,7,9]
print(lista)
t0 = time()
lista = busquedaBinaria(lista,9)
t1 = time()
print ("Lista ordenada:")
print(lista)
print ("Tiempo: {0:f} segundos".format(t1 - t0))
```

Figura 13. Implementación ejercicio 3. Algoritmo iterativo de búsqueda binaria. Fuente: elaboración propia.

Ejercicio 4. Implementación del problema de la mochila mediante un algoritmo voraz con fraccionamiento

Solución:

Esta es una de las soluciones más sencillas de este problema. Se supone que los objetos pueden ser fraccionados. Básicamente se tiene n objetos, que cuentan con un peso y un valor. Se puede tomar una fracción de un objeto i. Se representa como f_{obj_i}

Se trata de maximizar

$$\sum_{i=1}^{n} f\_obj_i valor_i$$

Sujeto a la restricción

$$\sum_{i=1}^{n} f\_obj_i peso_i \leq peso\_maximo$$

Para mejorar la implementación se usará un enfoque orientado a objetos.

Las opciones para llenar la mochila sería ir escogiendo los de más valor primero, los de menor peso, para que la mochila se llene más despacio o los objetos cuyo valor por unidad de peso sea mayor. Es precisamente este enfoque el que se va a implementar, ya que asegura que la solución que se encuentra es óptima.

Puede consultar la demostración en Brassard y Bratley (1997), teorema 6.5.1.

```
import quicksortgenerico
from lista import Lista
class ObjetoMochila:
    def __init__ (self, peso, valor, indice):
        self.indice = indice
        self.peso = peso
        self.valor = valor
        self.valorporpeso = valor/peso
    def __str__(self):
        return str(self.peso)

class ListaObjetosMochila(Lista):
    """Hereda la clase Lista con el atributo lista publico"""
    def __init__(self,tipo):
        Lista.__init__(self,tipo)
        """invoca al constructor de la clase padre"""
    @staticmethod
    def compararValorporPeso(objeto,x):
        """metodo estatico para expresar el criterio de comparacion para la ordenacion"""
        return objeto.valorporpeso>=x.valorporpeso

    @staticmethod
    def compararValor(objeto,x):
        """metodo estatico para expresar el criterio de comparacion para la ordenacion"""
        return objeto.valor>=x.valor

    def ordenarPorValorporPeso(self):
        """ordenacion por quicksort de la lista, se pasa el criterio de comparacion"""
"""como parametro"""
        return quicksortgenerico.quicksort(self.lista,ListaObjetosMochila.compararValorporPeso)

    def ordenarPorValor(self):
        """ordenacion por quicksort de la lista, se pasa el criterio de comparacion"""
"""como parametro"""
        return quicksortgenerico.quicksort(self.lista,ListaObjetosMochila.compararValor)
```

Figura 14. Implementación ejercicio 4. Problema de la mochila, algoritmo voraz. Fuente: elaboración propia.

```
class Mochila:
    def __init__(self, pesoMaximo):
        self.pesoMaximo=pesoMaximo
        self.objetosMochila=[]
        self.valor=0

    def obtenerMaxMochilaValorPorPeso (self,listaObjetos):
        self.objetosMochila = [0 for i in range(len(listaObjetos.lista))]
        valorMax=0
        peso=0
        listaOrdenada=listaObjetos.ordenarPorValorporPeso()
        """ordena la lista de objetos por el criterio definido"""
        i=0
        """bucle voraz"""
        while peso<self.pesoMaximo and i<len(listaObjetos.lista):
            if peso+listaOrdenada[i].peso<self.pesoMaximo:
                self.objetosMochila[listaOrdenada[i].indice]=1
                peso+=listaOrdenada[i].peso
            else:
                self.objetosMochila[listaOrdenada[i].indice]=(self.pesoMaximo-peso)/listaOrdenada[i].peso
                peso=self.pesoMaximo
            i+=1

        for i in range(len(listaObjetos.lista)):
            valorMax+=self.objetosMochila[i]*listaObjetos.lista[i].valor
        self.valor=valorMax

    def obtenerMaxMochilaValor (self,listaObjetos):
        self.objetosMochila = [0 for i in range(len(listaObjetos.lista))]
        valorMax=0
        peso=0
        listaOrdenada=listaObjetos.ordenarPorValor()
        """ordena la lista de objetos por el criterio definido"""
        i=0
        """bucle voraz"""
        while peso<self.pesoMaximo and i<len(listaObjetos.lista):
            if peso+listaOrdenada[i].peso<self.pesoMaximo:
                self.objetosMochila[listaOrdenada[i].indice]=1
                peso+=listaOrdenada[i].peso
            else:
                self.objetosMochila[listaOrdenada[i].indice]=(self.pesoMaximo-peso)/listaOrdenada[i].peso
                peso=self.pesoMaximo
            i+=1

        for i in range(len(listaObjetos.lista)):
            valorMax+=self.objetosMochila[i]*listaObjetos.lista[i].valor
        self.valor=valorMax

    def valorMochila(self,listaObjetos):
        self.valor
```

Figura 14. Implementación ejercicio 4. Problema de la mochila, algoritmo voraz. Fuente: elaboración propia.

```
listaObjetos=ListaObjetosMochila(ObjetoMochila)
listaObjetos.insertarElemento(ObjetoMochila(10,20,0))
listaObjetos.insertarElemento(ObjetoMochila(20,25,1))
listaObjetos.insertarElemento(ObjetoMochila(30,66,2))
listaObjetos.insertarElemento(ObjetoMochila(40,40,3))
listaObjetos.insertarElemento(ObjetoMochila(50,60,4))

mochila=Mochila(100)

mochila.obtenerMaxMochilaValorPorPeso(listaObjetos)
print("Mochila llena con criterio valor por peso")
print(mochila.objetosMochila)
print(mochila.valor)

mochila.obtenerMaxMochilaValor(listaObjetos)
print("Mochila llena con criterio valor ")
print(mochila.objetosMochila)
print(mochila.valor)
-------------------------------------------------------
Ejecución:
Mochila llena con criterio valor por peso
[1, 1, 1, 0, 0.8]
159.0
Mochila llena con criterio valor
[0, 0, 1, 0.5, 1]
105.0
```

Figura 14. Implementación ejercicio 4. Problema de la mochila, algoritmo voraz. Fuente: elaboración propia.

Se ve que, si el criterio de comparación es el valor, se obtiene un valor menor que si se usa la ratio valor/peso. La solución óptima es esta última.

Se añade a continuación el algoritmo de *quicksort* genérico en el que se puede usar cualquier criterio de ordenación.

```
def obtenerPivote(lista,comparacion):
    pivote = lista[0]
    i=0
    j=len(lista)-1
    while i<j:
        while comparacion(lista[i],pivote) and i<j:
            i+=1
        while not comparacion(lista[j],pivote) and j>=0:
            j-=1
        if i<j:
            lista[j],lista[i]=lista[i],lista[j]
    lista[0]=lista[j]
    lista[j]=pivote
    return j

def quicksort(lista,comparacion):
    if len(lista)<=1:
        return lista
    else:
        l=obtenerPivote(lista,comparacion)
        l+=1
        menores=lista[:l]
        mayores=lista[l:]
        lista=quicksort(menores,comparacion)+quicksort(mayores,comparacion)
        return lista
```

Figura 15. Algoritmo *quicksort* genérico que se aplica en el código 13 para ordenación Fuente: elaboración propia.

Ejercicio 5. Problema de la mochila con programación dinámica

En este caso no se pueden fraccionar los objetos, por lo que la solución se complica bastante. El objetivo y las restricciones son las mismas que en el ejercicio anterior.

Solución:

Para poderlo resolver, se usa una tabla con las filas representando los objetos y las columnas, los pesos. La lista de objetos posibles se proporciona ordenada por pesos.

En este caso se ha incorporado la lista de objetos como atributo para poder rescatar la información de estos, que se introducen en la mochila en cualquier momento.

El algoritmo va calculando el valor por filas, suponiendo que solo se cuenta con los objetos de la fila correspondiente y las anteriores. El valor de cada elemento de la tabla se calcula de acuerdo con la siguiente fórmula:

$$tabla[i,j] = \max\left(tabla[i-1,j], tabla[i-1, j-peso_i] + valor_i\right)$$

Excepto $tabla[0,j] = 0\ si\ j \geq 0$.

En los casos en los que $j - peso_i < 0$, se toma $tabla[i][j] = V[i-1,j]$.

En todo caso, en cada celda no puede haber valores de objetos pesen más a el representado por la columna.

A continuación, se muestra un ejemplo de construcción de la tabla, se cuenta con 4 objetos que se muestran en cada fila y un peso máximo de 7.

Peso	0	1	2	3	4	5	6	7
$objeto\ 0$ $peso_0 = 6$ $valor_0 = 8$	0	0	0	0	0	0	8	8
$objeto\ 1$ $peso_1 = 2$ $valor_1 = 9$	0	0	9	9	9	9	9	9
$objeto\ 2$ $peso_2 = 5$ $valor_2 = 18$	0	0	9	9	9	18	18	27
$objeto\ 3$ $peso_3 = 7$ $valor_3 = 20$	0	0	9	9	9	18	18	27

Tabla 7. Tabla de memoria para el problema de la mochila resuelto mediante programación dinámica. Fuente: elaboración propia.

La tabla demuestra que existe una carga óptima con valor 27. A partir de la tabla se pueden recuperar cuáles son los objetos introducidos.

El proceso sería el siguiente:

- Se toma la celda de la última fila y columna, $tabla[3,7]$, se debe comprobar si el objeto de la fila está en la mochila o no.

- $tabla[3,7] = tabla[2,7]$, pero $tabla[3,7] \neq tabla[2,7 - \llbracket peso \rrbracket\_3\ ] + \llbracket valor \rrbracket\_3$, luego el objeto 3 no está en la mochila

- Se toma ahora $tabla[2,7] = tabla[1,7 - \llbracket peso \rrbracket\_2\ ] + \llbracket valor \rrbracket\_2$, entonces el objeto 2 está en la mochila.

- Como ya se ha determinado un objeto en la mochila de valor 18, se buscará en la fila la celda de valor $27 - 18 = 9$ y peso restante $7 - 5$, es decir, $tabla[1,2]$. Descontando el peso del objeto introducido, se cumple que $tabla[1,2] = tabla[0,2 - \llbracket peso \rrbracket\_1\ ] + \llbracket valor \rrbracket\_1$. El objeto 1 estará en la mochila.

- El peso restante es 0, luego no encontrará ningún objeto más.

Por tanto, se introducirán en la mochila dos objetos, el objeto 1 y el objeto 2.

A continuación, se muestra la implementación del algoritmo y se prueba para los mismos valores que en el ejercicio 3.

```
from lista import Lista
class ObjetoMochila:
    def __init__ (self, peso, valor, indice):
        self.indice = indice
        self.peso = peso
        self.valor = valor
        self.valorporpeso = valor/peso
    def __str__(self):
        cadena="Objeto ( "+str(self.indice)+", "+str(self.peso)+", "+str(self.valor)+" )"
        return cadena

class ListaObjetosMochila(Lista):
    """Hereda la clase Lista con el atributo lista publico"""
    def __init__(self,tipo):
        Lista.__init__(self,tipo)
        """invoca al constructor de la clase padre"""

class Mochila:
    def __init__(self, pesoMaximo,listaObjetos):
        self.pesoMaximo=pesoMaximo
        self.objetosMochila=[]
        self.listaObjetos=listaObjetos
        self.valor=0
    def obtenerTabla(self):
        tabla = [[0] * (self.pesoMaximo+1) for i in range(len(self.listaObjetos.lista))]
        for j in range(1,self.pesoMaximo+1):
            if (self.listaObjetos.lista[0].peso<=j):
                tabla[0][j]=self.listaObjetos.lista[0].valor
        for i in range(1,len(self.listaObjetos.lista)):
            for j in range(1,self.pesoMaximo+1):
                if j-self.listaObjetos.lista[i].peso<0:
                    tabla[i][j]=tabla[i-1][j]
                else:
                    tabla[i][j]=max(tabla[i-1][j],tabla[i-1][j-self.listaObjetos.lista[i].peso]+self.listaObjetos.lista[i].valor)
        return tabla
```

Figura 16. Código ejercicio 5. Problema de la mochila con programación dinámica. Fuente: elaboración propia.

```
def obtenerMax(self):
        valorMax=0
        self.objetosMochila=[0]*len(self.listaObjetos.lista)
        tabla=self.obtenerTabla()

        j=self.pesoMaximo
        for i in range(len(self.listaObjetos.lista)-1,0,-1):
            peso=self.listaObjetos.lista[i].peso
            valor=self.listaObjetos.lista[i].valor
            if tabla[i][j]==tabla[i-1][j-peso]+valor:
                self.objetosMochila[i]=1
                j-=peso
        for i in range(len(listaObjetos.lista)):
            valorMax+=self.objetosMochila[i]*self.listaObjetos.lista[i].valor
        self.valor=valorMax
    def valorMochila(self,listaObjetos):
        self.valor

listaObjetos=ListaObjetosMochila(ObjetoMochila)
listaObjetos.insertarElemento(ObjetoMochila(10,20,0))
listaObjetos.insertarElemento(ObjetoMochila(20,25,1))
listaObjetos.insertarElemento(ObjetoMochila(30,66,2))
listaObjetos.insertarElemento(ObjetoMochila(40,40,3))
listaObjetos.insertarElemento(ObjetoMochila(50,60,4))
```

Figura 16. Código ejercicio 5. Problema de la mochila con programación dinámica. Fuente: elaboración propia.

```
mochila=Mochila(100,listaObjetos)
print(mochila.obtenerTabla())
mochila.obtenerMax()
print(mochila.objetosMochila)
print(mochila.valor)
```

```
Ejecución:
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 45, 45, 45, 45, 45, 45,
45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45,
45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45,
45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45, 45,
45, 45, 45, 45, 45, 45, 45, 45], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 20, 20,
20, 20, 20, 20, 20, 20, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 66, 66,
66, 66, 66, 66, 66, 66, 66, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 91,
91, 91, 91, 91, 91, 91, 91, 91, 111, 111, 111, 111, 111, 111, 111, 111,
111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111,
111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111,
111, 111, 111], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 66, 66, 66, 66, 66, 66,
66, 66, 66, 66, 86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 91, 91, 91, 91, 91,
91, 91, 91, 91, 91, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111,
111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 126, 126, 126, 126, 126,
126, 126, 126, 126, 131, 131, 131, 131, 131, 131, 131, 131, 131, 131, 151],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 25,
25, 25, 25, 25, 25, 25, 25, 25, 25, 66, 66, 66, 66, 66, 66, 66, 66, 66, 66,
86, 86, 86, 86, 86, 86, 86, 86, 86, 86, 91, 91, 91, 91, 91, 91, 91, 91, 91,
91, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111,
111, 111, 111, 111, 111, 111, 126, 126, 126, 126, 126, 126, 126, 126, 126,
126, 146, 146, 146, 146, 146, 146, 146, 146, 146, 146, 151]]
[0, 1, 1, 0, 1]
151
```

Figura 16. Código ejercicio 5. Problema de la mochila con programación dinámica. Fuente: elaboración propia.

A fondo

Algorithms

Geeks for Geeks (https://www.geeksforgeeks.org/fundamentals-of-algorithms/)

Recurso en línea muy completo para indagar sobre algoritmos, su análisis y diseño, además de su aplicación para la resolución de problemas

Uso de heurísticas

Fox, P. (s. f.) *Usar heurísticas*. Khan Academy. Usar heurísticas (artículo) | Algoritmos | Khan Academy

Recurso para la profundización sobre el uso de heurísticas y los problemas difíciles de resolver.

Ejercicios resueltos de programación dinámica

Wextensibles (https://www.wextensible.com/temas/programacion-dinamica/)

Recurso para profundizar sobre la programación dinámica y ejercicios resueltos siguiendo esta técnica

Test

1. El método de ordenación de la burbuja:

A. Solo realiza una pasada por la lista de datos que va a ordenar.

B. Realiza múltiples pasadas por la lista de datos que va a ordenar.

C. Divide la lista en sub-listas y las ordena por separado.

D. Ninguna de las anteriores es verdadera.

2. El método de ordenación de selección:

A. Solo realiza una pasada por la mitad de la lista que va a ordenar.

B. Realiza múltiples pasadas por la lista de datos completa que va a ordenar.

C. Realiza múltiples pasadas por partes de la lista de datos sin ordenar.

D. Ninguna de las anteriores es verdadera.

3. El método de ordenación por inserción:

A. Busca el mínimo de los que quedan por ordenar.

B. En cada iteración coloca el primer elemento de la parte que queda por ordenar en su lugar.

C. Divide la lista en dos sub-listas y las ordena por separado.

D. Ninguna de las anteriores es verdadera.

4. En un esquema de «divide y vencerás»:

A. Se divide el problema en subproblemas, pero no se usa recursividad para solucionarlos.

B. Se divide el problema en subproblemas y se usa siempre una tabla de memoria en la que se guardan los valores de los subproblemas resueltos.

C. Se divide siempre el problema en tres subproblemas.

D. Ninguna de las anteriores es verdadera.

5. En el método de ordenación por mezcla:

A. Se divide la lista que se va a ordenar en dos listas, una con los elementos menores que el primero y otra con los mayores, se ordenan y se concatenan.

B. Se divide la lista que se va a ordenar en dos listas de tamaño similar, y se ordenan por separado y luego se mezclan.

C. Se divide la lista que se va a ordenar en dos listas, una con los elementos pares y otra con los impares, se ordenan y se concatenan.

D. Ninguna de las anteriores es verdadera.

6. En el método de ordenación rápida:

A. Se cuenta con dos listas, una con los elementos menores que el pivote y otra con los mayores, y se ordenan recursivamente.

B. Se divide la lista que se va a ordenar en dos de tamaño similar, y se ordenan por separado y luego se mezclan

C. Se divide la lista que se va a ordenar en dos, una con los elementos pares y otra con los impares, se ordenan y se concatenan.

D. Ninguna de las anteriores es verdadera.

7. Los algoritmos voraces:

A. Son siempre algoritmos de ordenación.

B. Son siempre algoritmos de búsqueda.

C. Se usan fundamentalmente para problemas de optimización.

D. Ninguna de las anteriores es verdadera.

8. En la programación dinámica:

A. Se divide el problema en subproblemas y se usa la recursividad para solucionarlos.

B. Se divide el problema en subproblemas y se usa una estructura en la que se guardan los valores de los subproblemas resueltos.

C. Se divide siempre el problema en tres subproblemas.

D. Ninguna de las anteriores es verdadera

9. Una heurística:

A. Siempre produce soluciones óptimas.

B. Siempre encuentra una solución.

C. Puede no encontrar una solución o, si la encuentra, que no sea óptima.

D. Ninguna de las anteriores es verdadera

10. Un ciclo hamiltoniano es:

A. Es un ciclo en el que aparecen todos los nodos de un grafo una sola vez.

B. Es un ciclo en el que aparecen todos los nodos de un grafo dos veces.

C Es un ciclo en el que aparecen todos los nodos de un grafo con todos sus sucesores, aunque estos ya estuviesen en el ciclo.

D. Ninguna de las anteriores es verdadera.