# PARTE 1: Backtracking

Analice, diseñe e implemente algoritmos para los siguientes problemas:

## Ejercicio 1

Implementar la función Dar Cambio que devuelve la cantidad mínima de monedas que hay que dar para cambiar *n* pesos con monedas de la denominación dada como parámetro.

**def darCambio(Cambio, Monedas)**
**Descripción:** Implementa la operación devolver cambio
**Entrada: Cambio** número que representa el monto del cambio, **Monedas,** un Array con las monedas que se dispone para dar ese cambio.
**Salida:** retorna el número mínimo de monedas que son utilizadas para devolver el cambio.

Nota: Asuma que en la lista de monedas siempre está la moneda con valor 1. Y que las monedas no se agotan.
Ejemplos:
monedas = [1, 2, 6, 8, 10], cambio = 14, solución: 2 (una moneda con denominación 6 y otra con 8)
monedas = [1, 3, 11, 7, 12], cambio = 20, solución: 3 (utilizando la combinación de monedas 12,7,1)

## Ejercicio 2

Se dispone de una mochila que acepta un peso máximo PesoMax, y de k latas de peso P1, P2, P3, …, Pk, todos diferentes. Se desea llevar la mayor cantidad de peso posible en la mochila. Implemente un método que decida que latas deben echarse con este fin.

**def mochila(PesoMax, latas):**
**Descripción:** Implementa la función mochila
**Entrada: PesoMax** número que representa el peso máximo que acepta la mochila, **latas** Array con el peso de las latas p1, p2, p3, …, $p_{length\_array}$.
**Salida:** retorna un array con las latas que maximizan el peso de la mochila.

## Ejercicio 3

Implementar la función SubsecuenciaCreciente que devuelva un array con la mayor cantidad de elementos del array de entrada que formen una secuencia monótona creciente. Los elementos en el resultado deben aparecer en el mismo orden en que aparecían en el array de entrada, y no tienen que ser consecutivos dentro de este. Por ejemplo, la mayor subsecuencia creciente en [ 5, 1, 2, 3, 100, 20, 17, 8, 19, 21 ] es [1, 2,3 , 8, 19, 21 ].

**def subsecuenciaCreciente(numeros):**
**Descripción:** Implementa la función SubsecuenciaCreciente
**Entrada: numeros** array de números naturales.
**Salida:** retorna **array** de números con la mayor subsecuencia creciente en el array de entrada **numero.**

Nota: puede haber más de una respuesta, el ejercicio solo exige que usted devuelva una de ellas.

## Ejercicio 4

Dado un array X de números enteros positivos y un número entero de T, implementar un algoritmo que devuelva si existe un subconjunto de elementos en X que suman el valor T. Por ejemplo si X = {8, 6, 7, 5, 3, 10, 9} y T = 15, la respuesta es **True**, porque los subconjuntos {8, 7} , {7, 5, 3} , {6, 9} , {5, 10} todos suman 15. Con este otro ejemplo X = {11, 6, 5, 1, 7, 13, 12} y T = 15, la respuesta es **False**.

```
def subconjuntoSuma(numeros, valor):
    Descripción: Implementa la función Subconjunto Suma
    Entrada: numeros array de enteros positivos, valor entero positivo.
    Salida: retorna True si existe un grupo de enteros en números cuya
    suma del valor de entrada.
```

# Parte 2: Greedy

## Ejercicio 5

**N** actividades requieren el uso exclusivo de un recurso común. Cada actividad dispone de un tiempo de inicio y fin. Seleccionar el mayor conjunto posible de actividades que no se superpongan.

```
def adminActividades(tareas, inicio, fin):
    Descripción: Implementa la función Administrar tareas
    Entrada: tareas array con las tareas donde cada tarea dispone de un
    tiempo de inicio t0 y un tiempo final tf, inicio entero positivo que
    representa desde cuando esta disponible el recurso común, fin entero
    positivo que representa hasta cuando esta disponible el recurso común.
    Toda tarea esta dentro de ese tiempo.
    Salida: retorna el listado de tareas que maximiza el uso del espacio
    en común sin que se solapen ninguna de estas.
```

## Ejercicio 6

Se tienen **n** números naturales distintos, siendo **n** una cantidad par, que tienen que juntarse formando parejas de dos números cada una. A continuación, de cada pareja se obtiene la suma de sus dos componentes, y de todos estos resultados se toma el máximo. Diseñar un algoritmo greedy que cree las parejas de manera que el valor máximo de las sumas de los números de cada pareja sea lo más pequeño posible. Ejemplo: suponiendo que los datos se encuentran en el siguiente vector

| 5 | 8 | 1 | 4 | 7 | 9 |
|---|---|---|---|---|---|

vamos a ver un par de formas de resolver el problema (no necesariamente la óptima):

- Seleccionamos como pareja los elementos consecutivos. De esta forma conseguimos las parejas (5, 8), (1, 4) y (7, 9); entonces, al sumar las componentes tenemos los valores 15, 5 y 16, por lo que el resultado final es 16.
- Seleccionamos como pareja los elementos opuestos en el vector Ahora tenemos las parejas (5, 9), (8, 7) y (1, 4); sumando conseguimos 14, 15 y 5, por lo que el resultado final es 15

(mejor que antes).

¿Habrá una resultado mejor para este ejemplo?

**def buscaPares(vector):**

**Descripción:** Implementa la función busca pares
**Entrada: vector** de tamaño par que contiene números enteros positivos.
**Salida:** retorna el valor mínimo de sumar los posibles pares que se pueden formar del **vector** (justo como se explica en la especificación anterior).

## Ejercicio 7

Se dispone de una mochila que acepta un peso máximo PesoMax, y de k latas de peso P1, P2, P3, …, Pk, todos diferentes. Cada lata dispone de un beneficio B1, B2, B3, …, Bk. Se desea llevar la mayor cantidad de beneficio posible en la mochila sin sobrepasar el peso. Implemente un método que decida que latas deben echarse con este fin.

**def mochila(PesoMax, latas):**

**Descripción:** Implementa la función mochila
**Entrada: PesoMax** número que representa el peso máximo que acepta la mochila, **latas** Array con el peso de las latas p1, p2, p3, …, $p_{length\_array}$.
**Salida:** retorna un array con las latas que maximizan el beneficio de la mochila.

NOTA: Parecido al ejercicio 2 solo notar que en este caso se le agrega un beneficio a cada lata y lo que se quiere es maximizar el beneficio en total sin sobrepasar el peso de la mochila.

# Parte 3: Divide y Vencerás

## Ejercicio 8

Búsqueda de un elemento X en una lista ordenada utilizando la técnica de divide y vencerás.

**def busquedaBinaria(lista, x):**

**Descripción:** Implementa la función búsqueda binaria
**Entrada: lista** de números ordenados de con monotonía creciente, **x** número.
**Salida: True** si **X** se encuentra en la lista, **False** en caso contrario.

## Ejercicio 9

Búsqueda del k-ésimo menor elemento de una lista.

**def busquedaKesimo(lista, k):**

**Descripción:** Implementa la función búsqueda del k-ésimo elemento
**Entrada: lista** de números, **k** número que representa el k-ésimo elemento en la lista si se ordena de menor a mayor.
**Salida: Valor** numérico que representa el k-ésimo menor elemento en la lista.

## Ejercicio 10

Implementar la función SubsecuenciaCreciente que devuelva un array con la mayor cantidad de elementos del array de entrada que formen una secuencia monótona creciente. Los elementos en el resultado deben aparecer en el mismo orden en que aparecían en el array de entrada, y no tienen que ser consecutivos dentro de este. Por ejemplo, la mayor subsecuencia creciente en [ 5, 1, 2, 3, 100, 20, 17, 8, 19, 21 ] es [1, 2,3 , 8, 19, 21 ].

**def subsecuenciaCreciente(numeros):**
**Descripción:** Implementa la función SubsecuenciaCreciente
**Entrada: numeros** array de números naturales.
**Salida:** retorna **array** de números con la mayor subsecuencia creciente en el array de entrada **números.**

Nota: este ejercicio ya lo tuvieron que realizar con una estrategia Backtracking, ahora es necesario usar la estrategia Divide y Vencerás para encontrar la solución.

## Ejercicio 11

Diseñar un algoritmo de coste en O(n) que dado un conjunto S con *n* números, y un entero positivo *k*<*n*, determine los *k* números de S más cercanos a la mediana de S.

## Ejercicio (opcional)

Dos cadenas se dicen que distan en uno si para obtener una de otra hay que insertar, cambiar o quitar un solo carácter. La distancia entre dos cadenas es la menor cantidad de transformaciones que hay que realizar para obtener una a partir de la otra.

**def distancia(string1, string2):**
**Descripción:** Implementa una función dadas dos cadenas, determinar la distancia entre ellas.
**Entrada: string1** y **string2** cadenas de texto.
**Salida: entero** que representa la distancia entre las cadenas **string1** y **string2.**

# Parte 4 Programación Dinámica

## Ejercicio 12

Implementar la función Dar Cambio que devuelve la cantidad mínima de monedas que hay que dar para cambiar **n** pesos con monedas de la denominación dada como parámetro.

**def darCambio(Cambio, Monedas)**
**Descripción:** Implementa la operación devolver cambio
**Entrada: Cambio** número que representa el monto del cambio, **Monedas,** un Array con las monedas que se dispone para dar ese cambio.
**Salida:** retorna el número mínimo de monedas que son utilizadas para devolver el cambio.

Nota: Asuma que en la lista de monedas siempre está la moneda con valor 1. Y que las monedas no se agotan. Es el mismo ejercicio de la Parte 1 pero esta vez hay solucionarlo utilizando la estrategia de Programación Dinámica.

## Ejercicio 13

Una variante del problema de la mochila es la siguiente. Tenemos un conjunto de enteros (positivos) A = {a1, a2, ..., an} y un entero K. El objetivo es encontrar si existe algún subconjunto de A cuya suma sea exactamente K. Implementar un algoritmo que resuelva este problema utilizando la estrategia de programación dinámica.

## Ejercicio 14

Dada una tabla de tamaño nxn de números naturales, se pretende resolver el problema de obtener el camino de la casilla (1, 1) a la casilla (n, n) que minimice la suma de los valores de las casillas por las que pasa. En cada casilla (i, j) habrá sólo dos posibles movimientos: ir hacia abajo (i+1, j), o hacia la derecha (i, j+1). Implementar un algoritmo que resuelva este problema utilizando la estrategia de programación dinámica.

## Ejercicio 15

Problema de la Secuencia Común más Larga (LCS). Dadas dos secuencias v = v1 v2…vm y w = w1 w2…wn la LCS de v y w es una secuencia de posiciones en v: $1 < i_1 < i_2 < \ldots < i_t < m$ y otra secuencia de posiciones en w: $1 < j_1 < j_2 < \ldots < j_t < n$ tal que la $i_t$-sima letra de v es igual a la $j_t$-sima letra de w y t es máximo. Implemente un algoritmo que encuentre la LCS más larga dada dos secuencias de entradas.