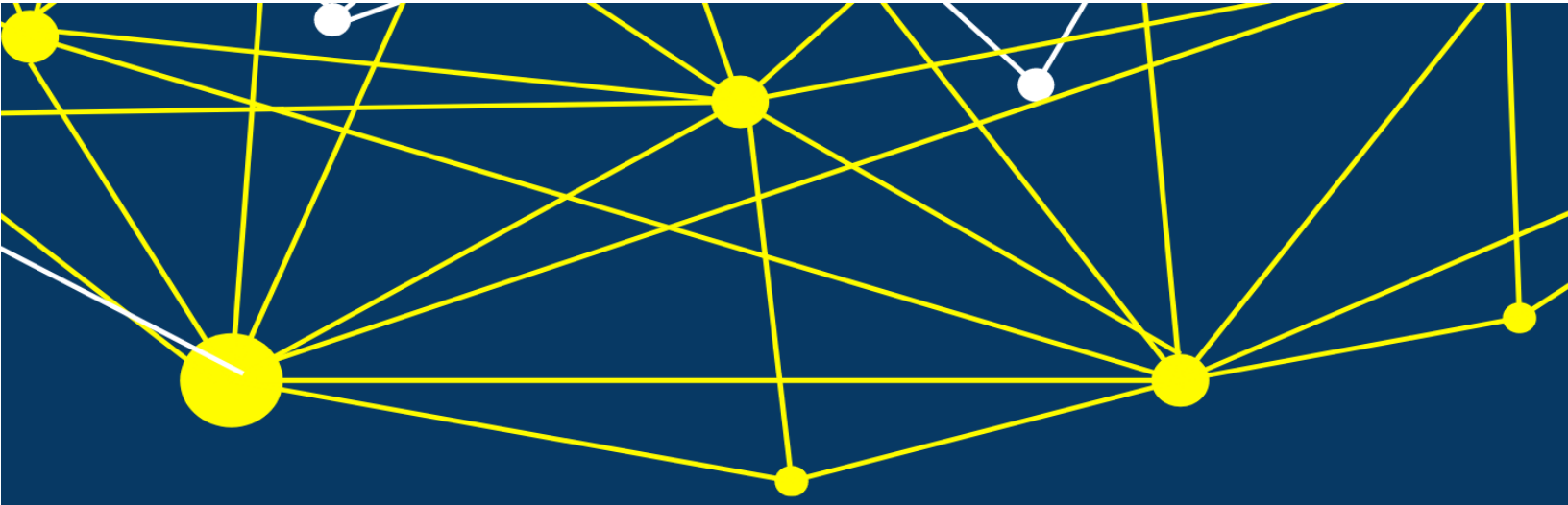


Adrián Borges Cano

Marco González Martínez

Laboratorio: Miércoles 15:00-17:00

Algoritmia

ÍNDICE

Tabla de contenido

PROBLEMA 3

Se tiene un número de Tam cifras almacenado en una cadena de texto; por ejemplo, la cadena dato = 1151451. Diseñar un algoritmo que mediante técnicas de Backtracking encuentre, de la manera más eficiente posible, todos los números distintos de N cifras que puedan formarse con los números de la cadena sin alterar su orden relativo dentro de la misma. Por ejemplo, si N = 4, son números válidos 1151, 1511 y 1541, pero no 4551 o 5411 que aunque pueden formarse con los dígitos de la cadena dato implican una reordenación.

Descripción del algoritmo:

Para resolver el problema hemos realizado un algoritmo que encuentra todas las combinaciones de índices de posición disponibles respetando los siguientes criterios necesarios para el enunciado del problema:

- Se escogen tantos índices como longitud de cadena de solución requiera el problema.

- Los índices escogidos no estarán repetidos en una misma lista de índices.

- Los índices dentro de una misma lista de índices tendrán un orden ascendente.

Casos de Prueba:

Con el fin de clarificar el funcionamiento del algoritmo, a continuación detallamos el procedimiento con el caso de prueba que figura en el enunciado. Para empezar, la cadena se interpretará por índices, en vez de por su contenido, entonces la cadena '1151451' pasa a ser [0,1,2,3,4,5,6] en esta lista de índices, buscaremos todas las de longitud n dada en este caso 4

Los resultados para el problema se hallarán de esta forma:

-Como primer elemento se iterará entre todos aquellos que cumplan estar en el rango entre el menor índice disponible, y la diferencia entre el mayor índice disponible, y la longitud de la cadena buscada sin contar el índice que se va añadir,ya que si se usa uno fuera de ese rango el resultado no respetará las condiciones del problema. Para este caso, se iterará entre 0 y 6-(4-1), se iterará 0, 1, 2 y 3.

- El resto de la lista de índices se calculará con la misma función teniendo como menor índice disponible el índice usado+1 y como longitud buscada la longitud usada-1. En el caso de la iteración en 0:

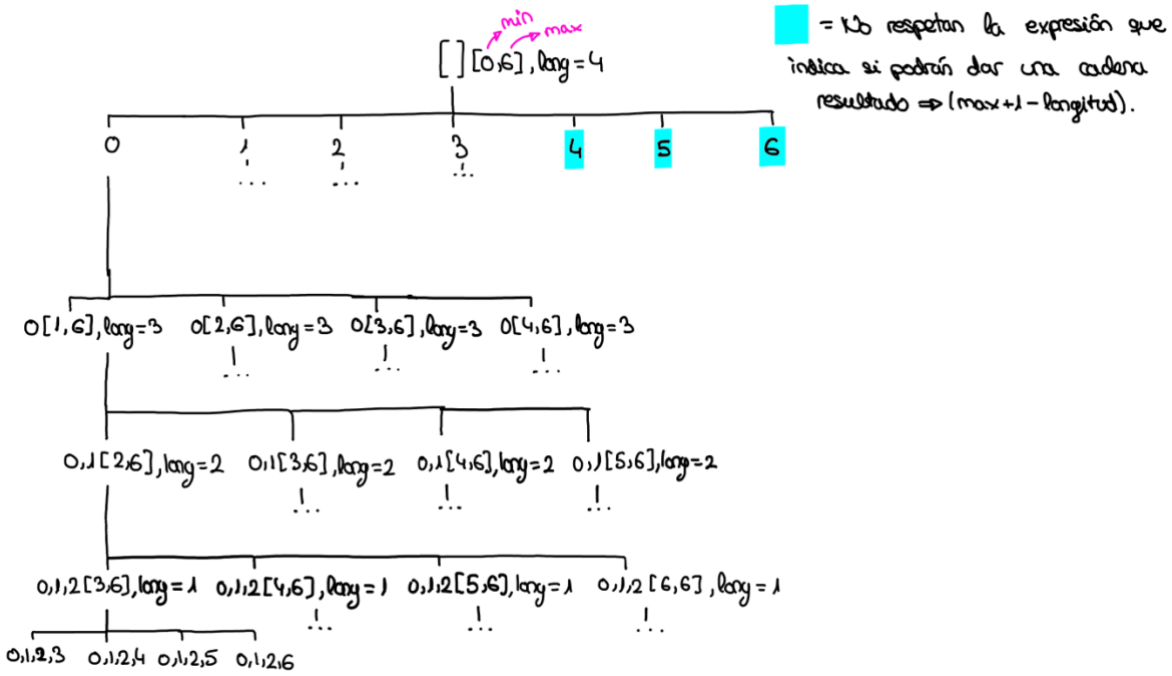
Se combinará el índice [0] con cada resultado devuelto por la solución al problema teniendo como índices disponibles el rango entre 1 y 6, buscando una cadena de longitud 3.

Todo este algoritmo profundizará recursivamente, así que por brevedad en la explicación, pasamos a explicar el funcionamiento en el caso de partida en el que se busca una cadena de longitud 1 usando los índices en el rango de 3 a 6.

Trivialmente, se pueden construir 4 cadenas de longitud 1, que serían [3],[4],[5] y [6].

Así que la profundación recursiva que hubiese llegado a este problema, construiría sus soluciones combinando, el número que está iterando con estas cadenas.

Para poder entender todo esto de una manera gráfica, a continuación hay un árbol explicando la profundización del algoritmo por su rama más a la izquierda, dado que el resto de ramas se podrían entender fácilmente por su procedimiento análogo.



Código del algoritmo:

```
def indices_bt(min,max,longitud):
    """
    int,int,int --> list(list(int))
    OBJ: Genera todas las listas posibles de longitud dada de índices comprendidos entre el mínimo y máximo dados ordenados ascendentemente
    """
    res = []
    # Caso de partida
    if longitud==1:
        for i in list(range(min,max)):
            res.append([i])
    # Caso general
    else:
```

```
        # Dejando de iterar en max+1-longitud excluimos las soluciones que darían una lista de menos índices
(no válidas para la solución)
        for i in range(min,max+1-longitud):
            # Los índices a añadir a la derecha deben ser mayores, o sino no se respeta el orden.
            for j in indices_bt(i+1,max,longitud-1):
                res.append([i]+j)
    return res


def subcadenas(cadena,n):
    """
    str, int --> list(str)
    OBJ: Encontrar todas las subcadenas de longitud n de la lista cadena, que respeten el orden de aparición
    PRE: n<=len(cadena)"""
    sol=[]
    y = len(cadena)
    indices = indices_bt(0,y,n)
    for l_ind in indices:
        subcadena = ''
        for ind in l_ind:
            subcadena += cadena[ind]
        sol.append(subcadena)
    return sol


print(subcadenas('1151451',4))
```

PROBLEMA 6

Se tiene la tabla de sustitución que aparece a continuación que se usa de la manera siguiente: en una cadena cualquiera, dos caracteres consecutivos se pueden sustituir por el valor que aparece en la tabla, utilizando el primer carácter como fila y el segundo carácter como columna. Por ejemplo, se puede cambiar la secuencia ca por una b, ya que M[c,a]=b. Implementar un algoritmo Backtracking que, a partir de una cadena no vacía texto y utilizando la información almacenada en una tabla de sustitución M, sea capaz de encontrar la forma de realizar las sustituciones que permite reducir la cadena texto a un carácter final, si es posible. Ejemplo: Con la cadena texto=acabada y el carácter final=d, una posible forma de sustitución es la siguiente (las secuencias que se sustituyen se marcan para mayor claridad): acabada → acacda → abcda → abcd → bcd → bc → d.

Descripción del algoritmo:

Para poder solucionar este ejercicio, hemos construído el árbol correspondiente que contiene todas las posibles combinaciones del ejercicio. Una vez hemos realizado esto, evaluamos si el resultado final coincide con el valor esperado, el cual se pasa como parámetro de entrada en la función. En caso afirmativo, la combinación hallada es guardada en el conjunto de soluciones; en caso contrario, la combinación obtenida se desprecia. Finalmente, como solo debemos mostrar las 5 primeras soluciones del problema, realizamos un bucle que itere y muestre estas 5 primeras soluciones, y se muestran por pantalla.

Casos de Prueba:

Para comprender mejor el funcionamiento del algoritmo, tomaremos como caso de prueba el mostrado en el enunciado:

Como se ha explicado previamente, calculamos todas las herencias posibles de las fusiones ofrecidas por la matriz M, de la siguiente forma:

Llamamos herencia a este proceso, en el que las letras se van sustituyendo por el valor correspondiente indicado en la matriz ofrecida. Por tanto, el método 'carácter_a_indice' es usado al realizar esto, ya que, en este caso, en lugar de buscar en la matriz el elemento pedido de la forma 'matriz[a][c]' (cosa que es imposible en Python), hacemos uso de la función previamente mencionada para que halle el valor en la posición 'matriz[0][2]'.

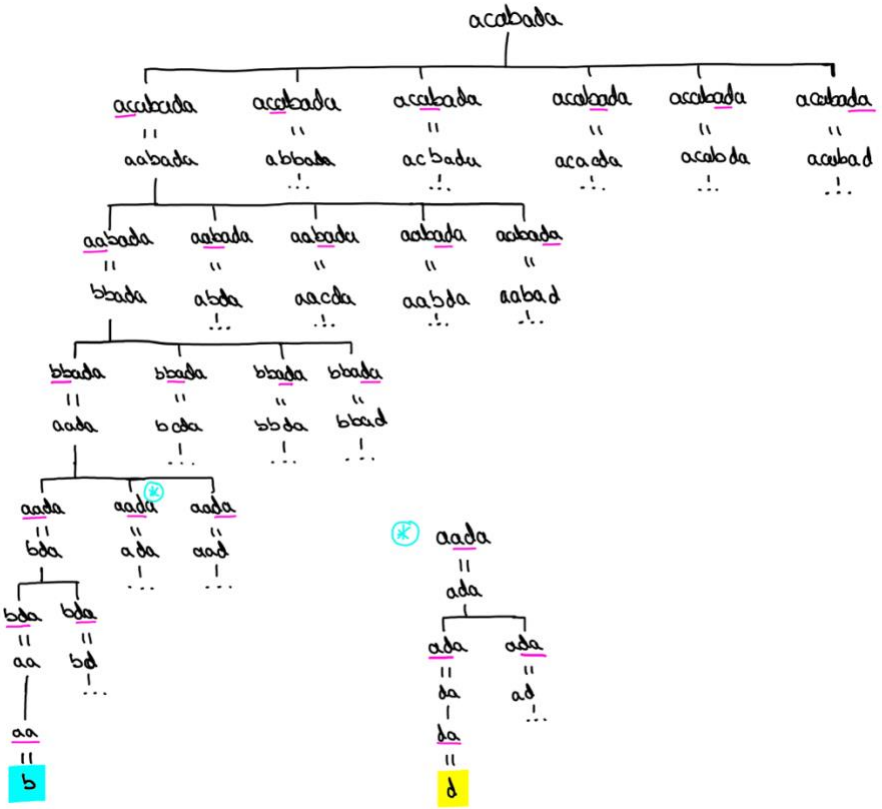
1.- Comenzamos a analizar las posibilidades de herencia que tendría la cadena.

2.- Una vez hemos realizado estas herencias, comenzamos a fusionar los elementos contiguos. Por ejemplo, para el primer caso, acabada → ac=a → aabada.

3.- El paso anterior se repite recursivamente hasta que nos quedemos solo con 2 letras; ya que esta cadena solo va a tener una posibilidad de herencia. En caso de que la letra final obtenida al aplicarle la herencia, coincida con la buscada, este conjunto de herencias se incluirá en la solución. De lo contrario, es desechada. En esta ocasión, la primera combinación obtenida será desechada, puesto que el resultado final es 'b', lo cual no coincide con el resultado esperado (b≠d).

Para facilitar la comprensión de lo realizado, adjuntaremos un diagrama del árbol realizado internamente para calcular todas las posibles soluciones mediante BackTracking. Las cadenas mostradas con puntos suspensivos (**…**) quiere decir que es una continuación del árbol, que obviamente no hemos hecho puesto que no es necesario realizar el árbol por completo para entenderlo correctamente.

La solución marcada en **azul**, es la primera solución hallada por el árbol, y la solución con la que se ha estado explicando el caso de prueba, la cual es desechada. Sin embargo, la que está marcada en **amarillo**, es la primera solución válida del árbol (d=d).



Código del algoritmo:

```
def letra_a_indice(letra):
    return ord(letra)-97

def herencia(M_sust, cadena, indice):
    x = cadena[indice]
    y = cadena[indice+1]
    z = M_sust[letra_a_indice(x)][letra_a_indice(y)]
    cadena = cadena[0:indice]+ z + cadena[indice+2:]
    return cadena

def desarrollo_herencia(M_sust, cadena, caracter):
    # Caso de partida
    if len(cadena)==2:
        if herencia(M_sust,cadena,0)==caracter:
            return [[0]]
        else:
```

```
        return []
    #Caso general
    else:
        sol=[]
        for i in range(len(cadena)-1):
            des_herencia = desarrollo_herencia(M_sust,herencia(M_sust,cadena,i),caracter)
            if des_herencia != []:
                for lista_herencia in des_herencia:
                    sol.append([i]+lista_herencia)
    return sol

def explicar_herencia(M_sust,cadena,desarrollo):
    if len(desarrollo)==1:
        return cadena + ' -> ' + herencia(M_sust,cadena,desarrollo[0])
    return cadena + ' -> ' + explicar_herencia(M_sust,herencia(M_sust,cadena,desarrollo[0]),desarrollo[1:])

matriz = [['b','b','a','d'],['c','a','d','a'],['b','a','c','c'],['d','c','d','b']]
des_her = desarrollo_herencia(matriz,'acabada','d')
for i in range(5):
    print(explicar_herencia(matriz,'acabada',des_her[i]))
```