Alineamientos y programación dinámica

Para poder establecer el parecido entre dos secuencias usaremos programación dinámica aplicado a las secuencias 1 y 2.

Se pueden hacer al menos dos tipos de alineamientos: globales y locales. Para ambos ocuparemos métodos basados en programación dinámica

Programación dinámica

Es un método que puede aplicarse en problemas que pueden ser partidos en subproblemas, y que el costo asociado de llegar entre A y B puede ser partido en el costo de llegar de A a C y luego de C a B, cuando C está entre A y B.

También es importante que los costos de los subproblemas los iremos guardando en alguna estructura de datos para onsultarlo mientas avanza el problema original.

El ejemplo que vimos en clase fue el de Fibonacci, esta corresponde a:

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_n &= f_{n-2} + f_{n-1}
\end{aligned}
$$

A continuación pueden ver la implementación en su forma recursiva

In [1]:

```
def fib_rec(n):
    if(n==0):
        return 0
    elif(n==1):
        return 1
    else:
        return fib_rec(n-2) + fib_rec(n-1)


```

In [2]:

```
print(fib_rec(8))
```

```
21
```

Intenten la forma recursiva con n grande

Ahora vamos a usar una lista para guardar los valores ya calculados

In [5]:

```
def fib_lista(n):
    L = [0,1]
    for i in range(n-1):
        v = L[-1] + L[-2]
        L.append(v)
    return L[-1]

```

In [7]:

```
print(fib_lista(100))
```

```
354224848179261915075
```

En la linea 4 de `fib_lista` estamos haciendo uso de los valores ya calculados y que se requieren para este paso. Hay más info en la [wiki de Fibonacci (https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci)](https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci)

Similitud entre cadenas

In [ ]:

```

```

Nuestro problema radica en que queremos comparar el parecido entre dos (o más cadenas). Primero consideremos el caso en el que las dos cadenas tienen el mismo tamaño con alguna diferencia, ejemplo

$$c1 = GATTACA$$
$$c2 = GATAACA$$

¿cómo le hacemos para decir cuánto difieren?

Claramente requerimos una función que tome como entrada las dos cadenas y como salida ¿qué se les ocurre?

Una opción es esta función

In [9]:

```
def similitud(c1, c2):
    s = 0
    for i,j in zip(c1,c2):
        if(i!=j):
            s+=1
    return s

print(similitud('GATTACA','GATAACA'))
```

```
1
```

Preguntas

1. Describe en lenguaje natural qué está haciendo esta función
2. ¿Para qué usamos la variable s?

Longest Common Subsequence

El problema con el método anterior es que requiere que las dos cadenas midan el mismo tamaño y esto no es biológicamente plausible. En la naturaleza hay inserciones o deleciones por lo que vamos a tener que "editar" alguna de las cadenas con algún símbolo que indique donde hubo un *indel*, este será el *gap* ("-")

Por ejemplo, queremos que nuestro algoritmo nos diga que

```
GATTACA
GATTACA
```

Son idénticas. Pero también

```
GATTACA
GATAACA
```

Difieren en una posición. Y qué entre `GATTACA` y `GATACA` hay que agregar un gap y cuál sería la mejor posición para esto, por decir algo:

```
GATTACA
GAT-ACA
```

In [ ]:

```
1
```

En el ejemplo de clase habíamos construido una función que partía en subproblemas el problema de maximizar la cantidad de lugares interesantes por visitar, y también permitía construir una ruta que pasaba por dichos lugares.

La función de score a usar era

$$\sigma_{i,j} = \max \begin{cases} \sigma_{i-1,j} + w[(i-1,j),(i,j)] \\ \sigma_{i,j-1} + w[(i,j-1),(i,j)] \end{cases}$$

Lo que rellena una matriz $\Sigma = \{\sigma_{i,j}\}$. E mn cada paso lo que hacemos es fijarnos en qué agrega más a la entrada (i,j), si llegar de la cuadra de arriba ($\sigma_{i-1,j} + w[(i-1,j),(i,j)]$) o de la izquierda ($\sigma_{i,j-1} + w[(i,j-1),(i,j)]$).

El algoritmo queda entonces:

1. $s_{0,0} \leftarrow 0$ _La entrada 0,0 tiene un valor de 0
2. for $i = 1..n$: $s_{i,0} \leftarrow s_{i-1,0} + w_{i,0}$ #caso base: la primer columna solo acumula valores desde la celda de arriba.
3. for $j = 1..m$: $s_{0,j} \leftarrow s_{0,j-1} + w_{0,j}$ #caso base: la primer fila solo va acumulando valores a la izquierda.
4. for $i = 1..n$:
5. for $j = 1..m$:
6. $s_{i,j} \leftarrow \sigma_{i,j}$

En todos los pasos tenemos que llenar otra matriz D que indique de que celda se escribió el máximo.

Ahora, nuestro problema es muy similar. Lo que vamos a hacer es comparar una cadena ($u = u_1 u_2 \dots u_n$) de forma progresiva con otra ($v = v_1 v_2 \dots v_m$), es decir en realidad vamos a estar comparando el parecido de las dos subcadenas caracter a caracter. la s

Nuestra pregunta en realidad es ¿Cómo transformo, entonces la cadenas $v = tcggaa$ en $u = caggta$?

Nuestra notación ahora será (v_i, u_j). El proceso para transformar v en u podría ser este:

```
(t,-)
(c,c)
(-,a)
(g,g)
(g,g)
(a,t)
(a,a)
```

Las secuencias al final pueden quedar como:

```
t c - g g a a
- c a g g t a
```

Para usar nuestro método de programación dinámica vamos a armar una matriz de $(n+1) \times (m+1)$ donde u quedará en la fila superior y v en la columna de la izquierda, amba secuencias tendrán una posición más correspondiente al gap. En cada entrada de esta matriz tendrá el *score* de la comparación hasta ese punto en ambas subcadenas.

	x	**-**	**c**	**a**	**g**	**g**	**t**	**a**
-	\							
t	.	|						
c	.	\						
g	.	.	>	\				
g	.	.	.	.	\			
a	.	.	.	.	.	\		
a	.	.	.	.	.	.	\	

Nota importante sobre notación:

1. (x, -) corresponde al escenario en el que el score máximo de (i, j) se obtiene de usar el gap en la secuencia u
2. (-, y) corresponde al escenario en el que el score máximo de (i, j) se obtiene de usar el gap en la secuencia v
3. (x, y) corresponde al escenario en el que el score máximo de (i, j) se obtiene de que comparar los simbolos en esa particular posición.

De este ejemplo podemos que la subcadena `agga` es común a las dos. Por subcadena nos referimos a una cadena de caracteres cuyo orden se preserva en una cadena más grande.

Pero este es un posible arreglo de varios posibles. Necesitamos un algoritmo que nos permita escoger el mejor alineamiento posilbe, para esto usaremos programación dinámica, donde el problema general es encontrar la subcadena común "alineada" de u en v agregando gaps en los "mejores" sitios. El subproblema que compone a este problema es encontrar la subcadena común de $u[:j]$ en $v[:i]$ (Estoy ocupando la notación de Python para indicar que vamos a comparar la subcadena u hasta l posición j con v hasta la posición i.

In [ ]:

```
1
```