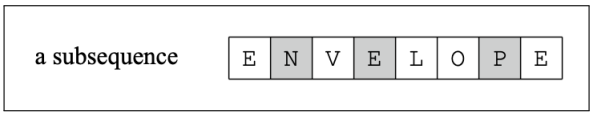
Asesoría 5: Programación Dinámica 2

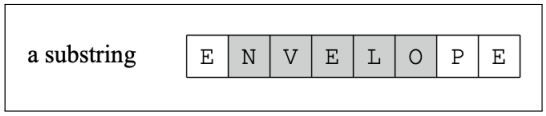
July 5, 2022

1 Longest Common Subsequence (LCS)

Para tener el problema más claro, dejemos clara la diferencia entre **substring** y **subsequence**. Un **subsequence** es cualquier secuencia de caracteres de un string siguiendo su orden original.



Por otro lado, un **substring** es una secuencia de caracteres **continuos** de un string.



Dadas estas dos definiciones, uno se puede dar cuenta que **todo *substring* es un *subsequence***, pero **no todo *subsequence* es un *substring***.

Ahora regresemos al problema. Dados dos strings **a** y **b** el longest common subsequence entre **a** y **b** (denotado: LCS(a, b)) es el string más largo que aparece como subsecuencia en **a** y **b**.

Por ejemplo, si tenemos que el primer string es *opera* y el segundo string es *tour*, entonces el LCS(opera, tour) es igual a "or".

Como es un problema de programación dinámica, tenemos que pensar en subproblemas, es decir, definir una recurrencia que da la respuesta "hasta el i-ésimo índice" o "de los i primeros elementos", etc. Por ello, dados dos strings **a** y **b** definamos un LCS(i, j) que da el tamaño del longest common subsequence entre los strings $a[0 \dots i]$ y $b[0 \dots j]$. La idea detrás de la recurrencia es que si $a[i] = b[j]$ entonces el LCS(i, j) es igual a 1 + el LCS de ambos strings sin su último caracter y si $a[i] \neq b[j]$, agarramos el LCS máximo entre quitar un caracter de a y quitar un caracter de b. La recurrencia es la siguiente:

$$LCS(i,j) = \begin{cases} 0 & \text{si } i-1 \leq 0 \text{ ó } j-1 \leq 0 \\ LCS(i-1, j-1) + 1 & \text{si } a[i] == b[j] \\ \max\{\ LCS(i-1, j),\ LCS(i, j-1)\ \} & \text{si } a[i] \neq b[j] \end{cases}$$

Notar que $LCS(a.length-1,\ b.length-1)$ (indexamos el string desde 0) tiene la respuesta a nuestro problema original. Una vez que ya tenemos nuestra recurrencia que define nuestro problema original en términos de sus subproblemas, podemos empezar a pensar en cómo lo tabularíamos. Tomando el mismo ejemplo donde **a** = "opera" y **b** = "tour", nuestra tabla se vería de la siguiente manera:

		O	P	E	R	A
	T	0	0	0	0	0
	O	1	1	1	1	1
	U	1	1	1	1	1
	R	1	1	1	2	2

Para llenar la matriz, hacemos un recorrido de **izquierda-derecha** y **arriba-abajo**. Siguiendo el criterio descrito por la recurrencia, podemos llenar la matríz en $\mathcal{O}(a.length,\ b.length)$, el cual es el tiempo de ejecución de este algoritmo para obtener el LCS.

Ahora, este algoritmo nos da el **tamaño** del LCS entre dos strings. Sin embargo, **¿cómo podríamos obtener el string de este LCS a partir de la matriz llena?** Digamos que empezamos con un string vació `s` que va a representar nuestra solución. Para construir nuestra solución, empezamos a hacer una suerte de *backtracking* desde la celda que contiene el LCS entre ambos strings (celda que está más al sureste). Empezamos desde ahí y nos preguntamos, ¿ambos caracteres en esta posición son iguales o no? En el primer caso, cuando los caracteres son iguales, hacemos `s.push_front(caracter_que_esta_en_ese_indice)` y nos movemos a la celda $(i-1,\ j-1)$, asumiendo que estamos en la posición $(i,\ j)$. En el segundo caso, cuando los caracteres no son iguales, entre las celdas $(i-1,\ j)$ y $(i,\ j-1)$, nos movemos a la celda de la que obtuvimos el valor para llenar la celda en la que estamos y seguimos haciendo la recursión. Paramos cuando hallamos llegado a la posición inicial de la matriz. Al final, `s` tendrá la solución.

Otro ejemplo, digamos que queremos hallar $LCS(LOVE, MOVIE)$:

LCS (LOVE, MOVIE)				
	L	O	V	E
M				
O				
V				
I				
E				

Ejercicio. Llenar la matriz, hallar el tamaño del LCS entre ambos strings y luego construir la solución para obtener el string que representa en LCS.

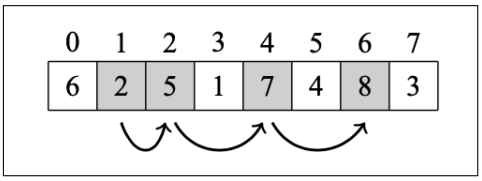
1.1 Material Adicional

- Cormen: página 390.
- `https://www.youtube.com/watch?v=ASoaQq66foQ`

2 Longest Increasing Subsequence (LIS)

El LIS de un array A de tamaño n es una **subsecuencia** de números de A donde cada elemento en la posición i de la subsecuencia, es mayor al elemento $i-1$ para todo i entre $2 \dots n$. Usaremos la misma definición de subsecuencia que usamos en el ejercicio anterior.

En el ejemplo, el arreglo es $[6, 2, 5, 1, 7, 4, 8, 3]$ y el LIS de ese arreglo sería $[2, 5, 7, 8]$.



Nuevamente, pensemos en subproblemas. Como se mencionó anteriormente, definiremos una recurrencia que calcula el LIS hasta el i-ésimo índice. Si tenemos un LIS que va hasta el índice $i-1$ y en el índice i tenemos un elemento que es mayor al último elemento del LIS$(i-1)$, entonces LIS(i) = 1 + LIS$(i-1)$. Así podemos ir armando una idea de cómo formular el problema en términos de sus subproblemas. La recurrencia es la siguiente:

$$LIS(i) = \begin{cases} 1 & \text{si } i = 1 \\ LIS(i) = \max_{1 \le j < i}\{LIS(j) + 1 : \ A[j] < A[i]\} & \text{si } i > 1 \end{cases}$$

Notar que $LIS(n)$ $(n = A.length)$ tiene la solución a nuestro problema original, ya que $LIS(n)$ se lee como "la subsecuencia creciente de tamaño máximo que termina en el índice n". Lo que dice esta recurrencia es que fijamos el i desde la posición 2 hasta la posición n. Luego de fijar el i, iteramos con un j desde 1 hasta $i-1$. Mientras vamos iterando por el j, preguntamos si $A[j] < A[i]$, si esto es verdad, entonces actualizamos el valor $LIS(i)$, entonces podemos hacer $LIS(i) = \max\{1 + LIS(j), LIS(i)\}$, caso contrario, continuamos iterando. De esta manera, nos aseguramos que al pasar de la iteración i a la iteración $i+1$, $LIS(k)$ para todo $k < i+1$ estará bien seteado, es decir, con el valor correcto, de tal manera que luego lo podamos usar para calcular $LIS(m)$ donde $m > i$.

2.1 Material Adicional

- `https://www.youtube.com/watch?v=fV-TF4OvZpk`

3 Rod Cutting

Tengo un rod (barra) de n pulgadas y una tabla que me indica el precio al que puedo vender distintos tamaños de la barra. Por ejemplo:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Esta tabla nos indica que si vendo un pedazo de 1 pulgada del rod, la podré vender por 1 dólar, si vendo un pedazo de 7 pulgadas del rod, la podré vender por 17 dólares, etc.

El problema es que dado un rod de n pulgadas y su tabla de precios, ¿cuál es la ganancia máxima que puedo obtener al cortar el rod en ciertos puntos y vender esos pedazos?

La idea detrás del algoritmo es bien parecida a la de knapsack: ¿corto el rod en este punto o no?, ¿qué implica el hecho de que corté el rod en un punto específico? Si no corto el rod en ese punto, entonces **hago la misma pregunta**, pero sin considerar el corte actual. Si corto el rod en ese punto, entonces primero tengo que sumarle a mi ganancia el precio de la tabla asociado a ese corte y luego **hacer la misma pregunta** pero ahora con un rod del tamaño original menos el tamaño del corte que acabo de hacer. Notar que **hacer la misma pregunta** simboliza **hacer una llamada recursiva**, ya que deseamos resolver un subproblema dejado por nuestra elección. Con esta intuición podemos definir una recurrencia. Supongamos que la tabla de precios es P. $ROD(i, j)$ me da la ganancia máxima de cortar un rod de tamaño i en cietos puntos si puedo solo hacer cortes de tamaño $\leq j$.

$$ROD(i,j) = \begin{cases} P[i] & \text{si } i = 1 \\ ROD(i, j-1) & \text{si } i > 1 \text{ y } j > i \\ \max\{ROD(i-j) + P[j], ROD(i, j-1)\} & \text{si } i > 1 \text{ y } j \leq i \end{cases}$$

Notar que nuestra solución se encuentra en $ROD(n, n)$, es decir, cuando queremos la ganancia máxima de cortar un rod de tamaño n y tenemos permitido hacer cortes de tamaño $\leq n$.

3.1 Material Adicional

- Cormen página 360.
- `https://www.youtube.com/watch?v=IRwVmTmN6go`

4 Variación del Knapsack Problem

Dado un conjunto de pesos $[w_1, w_2, \ldots, w_n]$, hallar todas las sumas posibles que se pueden construir usando dichos pesos.

Este problema es casi idéntico al de knapsack. Digamos que usaremos los k primeros pesos para consturir una suma x, para esto definiremos la recurrencia $possible(x, k)$ que nos indica si es posible o no formar la suma x usando los primeros k pesos. Entonces, tendríamos la siguiente recurrencia:

$$possible(x,k) = \begin{cases} true & \text{si } x = 0 \text{ y } k = 0 \\ false & \text{si } x \neq 0 \text{ y } k = 0 \\ possible(x, k-1) & \text{si } x - w_k < 0 \\ possible(x, k-1) \;||\; possible(x - w_k, k-1) & \text{caso contrario} \end{cases}$$

En este caso, varias celdas pueden tener la respuesta que queremos, ya que dada la matriz llena de este problema, podemos consultar si es posible cierta suma usando ciertos pesos. Por ejemplo, si tenemos que los pesos son $[1, 3, 3, 5]$ entonces la matriz se vería de la siguiente manera:

	0	1	2	3	4	5	6	7	8	9	10	11	12
$k = 0$	✓												
$k = 1$	✓	✓											
$k = 2$	✓	✓		✓	✓								
$k = 3$	✓	✓		✓	✓		✓	✓					
$k = 4$	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓

Notar que la matriz tiene m filas, donde m es el número de pesos que tenemos y tiene s columnas, donde s es la máxima suma posible dado el conjunto de pesos, ya que no podemos construir una suma mayor a esa. En el ejemplo anterior es 1+3+3+5=12.

4.1 Material Adicional

- Por si no les quedó claro el problema del Knapsack: `https://www.youtube.com/watch?v=xCbYmUPvc2Q`

5 Material de Apoyo

- **Link al Jamboard:** `https://jamboard.google.com/d/1_Ky2YpqZmKTL_6UCtVEzGYGFz2uJLT5Tkq3DP4X1o_s/edit?usp=sharing`
- **Playlist:** `https://www.youtube.com/watch?v=Zq4upTEaQyM&list=PLiQ766zSC5jM2OKVr8sooOuGgZkvnOCTI`
- **Playlist:** `https://www.youtube.com/watch?v=8LusJS5-AGo&list=PLrmLmBdmIlpsHaNTPP_jHHDx_os9ItYXr`
- **Tardos:** Sección 6.1, 6.2 y 6.4.
- **Cormen:** Capítulo 15.

6 Bibliografía

- Laaksonen, A. (2020). *Guide to Competitive Programming.* Springer. (Ed. 2).