

# Trabajo Práctico 1

## Teoría de Algoritmos - 1c 2025

### La 12 devs

#### Integrantes:

Franco Bustos 110759

Camila Aleksandra Kotelchuk 110289

Lorenzo Kwiatkowski 110239

Bruno Pezman 110457

Joaquin Miranda Iglesias 97500

Fecha: 4/5/25

Lenguaje: Python

Entrega: 2

#### Índice:

<b>Parte 1: Solitario... en grupo.</b>	<b>1</b>
<b>Parte 2: Los puentes que se cruzan</b>	<b>2</b>
<b>Parte 3: Maximizando los puentes</b>	<b>5</b>
<b>Requisitos y procedimientos para compilar el código</b>	<b>10</b>
<b>Correcciones con respecto a la 1er entrega</b>	<b>10</b>
<b>Referencias y enlaces externos</b>	<b>10</b>

## Parte 1: Solitario... en grupo.

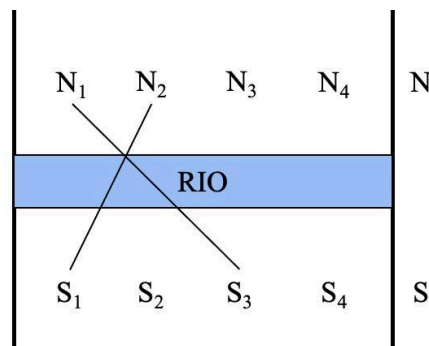
	Estructuras de datos	Complejidad temporal	Complejidad espacial	Estrategia Greedy	Óptimo local
Franco (F)	Una lista	$O(n \log(n))$	$O(n)$	Colocar la carta en la pila válida con el techo más cercano pero mayor a ella	Elegir la pila válida con el techo más cercano minimizando la cantidad total de pilas
Camila (C)	Dos listas de enteros	$O(n^2)$	$O(n)$	Elegir la pila con la cual tenga menor diferencia absoluta	Elegir la pila con la cual tenga menor diferencia absoluta
Lorenzo (L)	Una lista y una clase Pila	$O(n^2)$	$O(n)$	Seleccionar iterativa del óptimo local sin evaluar el efecto futuro de la elección	De ser posible ubicar la carta en la pila existente con la que tenga menor diferencia posible
Bruno (B)	Pilas necesarias y un array para guardar las pilas	$O(n^2)$	$O(n)$	Colocar cada carta en la primera pila donde pueda ir	Mínimo número de pilas en cada paso quedando la carta ubicada en la pila con el número de carta más alto que ella en su tope.
Joaquin (J)	Una lista y una pila	$O(n^2)$	$O(n)$	Agregar la carta en la primera pila que permite agregarla	Crear una pila solamente si se cumple la condición o si no quedan pilas en las que poder colocar

## Preguntas:

- 1) Sean las propuestas L,F,C,B,J una por cada integrante. Todas tuvieron enfoques similares utilizando todas estrategias Greedy, solamente que L,F y C tuvieron en cuenta la diferencia entre la carta a ubicar en la pila y la carta en el tope.
- 2) En cuanto a la complejidad, todos compartían la complejidad espacial  $O(n)$  dado que en el peor caso, los  $n$  elementos iban a estar ubicados en  $n$  pilas de tamaño 1, o iban a estar todos ubicados en una. Sin embargo, en el caso de la complejidad temporal, el caso F difiere de las otras dado que resulta ser  $O(n \log n)$  por el uso de una búsqueda binaria para ordenar las pilas, mientras que las demás fueron de  $O(n^2)$ . Por lo tanto, la propuesta F posee una mejor complejidad.
- 3) Optimalidad caso J: Supongamos que una entrada (de cartas, como las del problema) produce una solución óptima  $O$ , con  $p$  la cantidad de pilas. Supongamos que vuelvo a procesar la misma entrada, en el mismo orden, y me genera una solución Greedy, de  $p+1$  pilas. Esto implicaría que en algún momento la elección Greedy no ubico una carta en alguna de las pilas disponibles (en las cuales se podía colocar). Pero esto es absurdo porque el óptimo local minimiza la cantidad de pilas creadas por iteración.

## Parte 2: Los puentes que se cruzan

### Propuesta



La idea es la siguiente: si subíndice de  $N <$  subíndice de  $S$  o subíndice de  $N >$  subíndice de  $S$ , va a haber una cruz. Tenemos el orden de cada barrio en cada uno de los márgenes, Norte y Sur. Por ejemplo: Pienso a los puentes como índice

$$N = [N_1, N_2, N_3, N_4] \rightarrow 0, 1, 2, 3$$
$$S = [S_1, S_2, S_3, S_4] \rightarrow 0, 1, 2, 3$$

Entonces por ejemplo me dan una propuesta:

$$(N_1, S_3) \rightarrow (0, 2)$$
$$(N_2, S_1) \rightarrow (1, 0)$$
$$(N_3, S_2) \rightarrow (2, 1)$$
$$(N_4, S_4) \rightarrow (3, 3)$$

Entonces, se arma una lista:

$$\text{Lista} = [(0, 2), (1, 0), (2, 1), (3, 3)]$$

La ordeno por las primeras coordenadas (las de N) (en este caso ya están ordenadas así que la dejo igual) y luego me armo otra lista con las segundas coordenadas (las de S): [2, 0, 1, 3] y con esta lista cuento cuantas inversiones tengo: cuando hay un número mayor antes que un número menor, hay una inversión. Si hay una inversión sumamos un cruce. Si no hay cruces es factible.

### Ecuación de recurrencia

⇒ Cada problema con  $n$  elementos se divide en 2

⇒ La unión de los resultados se construye recorriendo una vez los  $n$  elementos.

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= 2T(n/2) + O(n) \end{aligned}$$

siendo  $a = 2$ ,  $b = 2$  y  $f(n) = O(n)$  tenemos según el caso 2 del teorema maestro:

$$\text{si } f(n) = O(n^{\log_2 2}) = O(n) \implies T(n) = O(n^{\log_2 2} \log n) = O(n \log n)$$

Entonces, nuestra complejidad temporal final es:

$$T(n) = O(n \log n)$$

### Pseudocódigo

Sea N y S diccionarios de los barrios del norte y sur respectivamente, de clave: nombre y valor: índice.

Sea P una lista de propuestas de los pares (n,s).

Se ordena P según las claves de N en orden creciente.

Se obtiene una lista L con los índices de los barrios del sur, en el orden de aparición de P.

Sea Inversiones las inversiones totales el resultado de realizar MergeSort(L)

MergeSort(L):

```

Si |L| <= 1:
    return L, 0
Sino
    izq: Los n/2 primeros elementos de L
    der: Los n/2 últimos elementos de L
    (izq, inv_izq) = mergeSort(izq)
    (der, inv_der) = mergeSort(der)
    (merged, inv_merge) = merge(izq, der)
    retornar (merged, inv_izq + inv_der + inv_merge)

```

Merge(izq, der):

```

Sea resultado R
i = j = inv = 0
mientras haya elementos para iterar en izq y der:
    si izq[i] <= der[j]:
        R.append(izq[i])
        i += 1
    sino:
        R.append(der[j])

```

```

        inv += len(izq) - i
        j += 1
    R += izq[i:]
    R += der[j:]
    return R, inv

```

Imprimimos Inversiones

## Funcionamiento

<p>Resultado de <b>procesar archivo barrios</b>, devuelvo dos diccionarios con clave barrio y valor índice según zona (norte y sur).</p>	<pre> orden_norte = {     'Barrancas': 0,     'San Pedro': 1,     'El hurón': 2,     'Palo seco': 3,     'Puente viejo': 4 } orden_sur = {     'Cienagas': 0,     'Don Corleone': 1,     'Barrio Este': 2,     'Portuario': 3,     'Torre verde': 4 } </pre>
<p>Resultado de <b>procesar archivo propuestas</b> y ordenado según barrios del norte:</p>	<pre> propuestas ordenadas = [     ('Barrancas', 'Cienagas'),     ('San Pedro', 'Barrio Este'),     ('El hurón', 'Portuario'),     ('Palo seco', 'Torre verde'),     ('Puente viejo', 'Don Corleone') ] </pre>
<p>Genero una lista "lista_sur" con los valores de los índices originales de los barrios del sur, pero ahora ordenados según la lista de propuestas con el orden de arriba.</p>	<pre> lista_sur = [     orden_sur['Cienagas']      0     orden_sur['Barrio Este']   2     orden_sur['Portuario']     3     orden_sur['Torre verde']   4     orden_sur['Don Corleone']  1 ] lista_sur = [0, 2, 3, 4, 1] </pre>
<p>Cuento los cruces dentro de <b>Contar Inversiones</b> con <b>merge-sort</b> modificado, suma recursivamente las inversiones necesarias en cada sub-lista izquierda y derecha + las inversiones necesarias <i>mergeandolas</i>.</p>	<pre> ContarInversiones([0, 2, 3, 4, 1]) ├─ merge_sort([0, 2]) │   ├── [0] → 0 inv │   ├── [2] → 0 inv │   └─ merge([0], [2]) → [0, 2], inv = 0 ├─ merge_sort([3, 4, 1]) │   ├── [3] → 0 inv │   ├── merge_sort([4, 1]) │   │   ├── [4] → 0 inv │   │   ├── [1] → 0 inv │   │   └─ merge([4], [1]) → [1, 4], inv = 1 │   └─ merge([3], [1, 4]) → [1, 3, 4], inv = 1 │       (porque 3 &gt; 1) </pre>

	<pre> └ merge([0, 2], [1, 3, 4]) → [0, 1, 2, 3, 4], inv = 1   (porque 2 &gt; 1)  TOTAL = 0 (izq) + 2 (der) + 1 (merging) = **3** </pre>
Imprimimos por salida estándar este valor	"3"

## Análisis

A la hora de proponer algoritmos, nosotros decidimos encararlo desde un algoritmo de *Merge-sort* modificado para contabilizar la cantidad de inversiones se realizan, sin embargo, esta modificación no tendría impacto en la complejidad del algoritmo.

$$T(n) = 2T(n/2) + O(n)$$

El cual se corresponde al caso 2 del Teorema Maestro y obteniendo:

$$T(n) = O(n \log n)$$

A la hora de pasarlo a código respetamos esta complejidad siendo las complejidades de los pasos:

- Lectura y parseo de archivos:  $O(n)$
- Ordenar propuestas por orden norte con *sort* de python:  $O(n \log n)$
- Creación lista índice sur:  $O(n)$
- Conteo inversiones:  $O(n \log n)$

Dando una complejidad que se corresponde a nuestra propuesta y es  **$O(n \log n)$**

## Parte 3: Maximizando los puentes

### 1 - Solución

#### Subproblema

Elegir una nueva subpropuesta que no forme un cruce ni actual ni con los anteriores.

#### Relación de Recurrencia

Sea  $dp$  la secuencia creciente más larga en la secuencia de índices del sur

$$\begin{aligned}
 dp[i] &= 1 + \max(dp[j]), \text{ si existe al menos un } j < i \text{ tal que } sur[j] < sur[i] \\
 dp[i] &= 1, \quad \text{ si no existe tal } j
 \end{aligned}$$

#### Pseudocódigo

Sean norte y sur listas con los nombres de barrios del norte y del sur, respectivamente.

Sea propuesta una lista de tuplas (barrio\_norte, barrio\_sur).

// Crear diccionarios con el orden de los barrios

Sea orden\_norte un diccionario: claves = barrios del norte, valores = índice correspondiente.  
Sea orden\_sur un diccionario: claves = barrios del sur, valores = índice correspondiente.

// Ordenar las propuestas por el orden del barrio norte  
Sea propuestas\_ordenadas una lista de tuplas (barrio\_norte, barrio\_sur),  
ordenada según el índice de barrio\_norte en orden\_norte.

// Construir la secuencia de índices del sur según el orden norte  
Sea secuencia una lista vacía.  
Para cada (norte, sur) en propuestas\_ordenadas:  
    Buscar índice\_sur en orden\_sur[sur]  
    Agregar índice\_sur a secuencia

// Algoritmo para encontrar la subsecuencia creciente más larga (LIS)  
Sea n la longitud de secuencia  
Sea dp una lista de tamaño n, inicializada con 1s  
Sea prev una lista de tamaño n, inicializada con -1s

Para i desde 0 hasta n - 1:  
    Para j desde 0 hasta i - 1:  
        Si secuencia[j] < secuencia[i] Y dp[j] + 1 > dp[i]:  
            dp[i] = dp[j] + 1  
            prev[i] = j

// Reconstruir la secuencia óptima  
Sea idx el índice con el valor máximo en dp  
Sea indices\_solucion una lista vacía

Mientras idx ≠ -1:  
    Agregar idx a indices\_solucion  
    idx = prev[idx]

Invertir indices\_solucion

// Obtener las propuestas finales  
Sea propuestas\_finales una lista vacía  
Para cada i en indices\_solucion:  
    Agregar propuestas\_ordenadas[i] a propuestas\_finales

Imprimir longitud de propuestas\_finales y propuestas\_finales

## **2 - Explicación de por que la propuesta funciona**

Construir la máxima cantidad de puentes sin cruces se puede ver como una variante del problema de la subsecuencia creciente más larga (LIS). Si ordenamos las propuestas por uno de los márgenes, entonces el problema se reduce a buscar la mayor cantidad de propuestas que formen una secuencia creciente en el otro margen. Para cada propuesta actual, buscamos entre las anteriores una con la que no haya cruce (que termine en un valor menor). Si la encontramos, la extendemos con la nueva propuesta y actualizamos el mejor resultado hasta el momento.

### 3 - Complejidad espacial y temporal

Sea  $2n$  la cantidad de barrios totales, siendo 'n' barrios del norte y 'n' barrios del sur.

#### ❖ Temporal

- Asociar a los barrios con índices en diccionarios es  $O(n)$
- Ordenar propuestas por orden norte con *sort* de python:  $O(n \log n)$
- Recorrer las propuestas ordenadas por barrio norte y hacer operaciones a cada barrio es  $n * O(1)$  *amortizado (trabajamos con diccionarios)*  $\Rightarrow O(n)$
- Recorrer la secuencia de índices del sur y buscar la secuencia creciente más larga, en el peor caso  $\Rightarrow O(n^2)$
- Invertir orden de índices  $\Rightarrow O(n)$
- Reconstruir la solución  $\Rightarrow O(n)$
- Total:  $O(n) + O(n \log n) + O(n) + O(n^2) + O(n) + O(n) \Rightarrow O(n^2)$

#### ❖ Espacial

- Dos diccionarios para almacenar n elementos  $\Rightarrow O(n)$
- Array para almacenar la secuencia más larga que en el peor caso tendrá los n barrios sur  $\Rightarrow O(n)$
- Array de los índices  $\Rightarrow O(n)$
- Array de las propuestas finales  $\Rightarrow O(n)$
- Total:  $4 * O(n) \Rightarrow O(n)$

### 4 - Ejemplo paso a paso de funcionamiento de su propuesta.

Resultado de <b>procesar archivo barrios</b> , devuelvo dos diccionarios con clave barrio y valor índice según zona (norte y sur).	<pre>orden_norte = {     'Barrancas': 0,     'San Pedro': 1,     'El hurón': 2,     'Palo seco': 3,     'Puente viejo': 4 } orden_sur = {     'Cienagas': 0,     'Don Corleone': 1,     'Barrio Este': 2,     'Portuario': 3,     'Torre verde': 4 }</pre>
Resultado de <b>procesar archivo propuestas</b> y ordenado según barrios del norte:	<pre>propuestas_ordenadas = [     ('Barrancas', 'Cienagas'),     ('San Pedro', 'Barrio Este'),     ('El hurón', 'Portuario'),     ('Palo seco', 'Torre verde'),     ('Puente viejo', 'Don Corleone') ]</pre>
Recorro <b>propuestas_ordenadas</b> y guardo en "secuencia" el índice original de cada barrio del sur	<pre>secuencia = [     orden_sur['Cienagas']      0     orden_sur['Barrio Este']   2     orden_sur['Portuario']     3     orden_sur['Torre verde']   4     orden_sur['Don Corleone']  1 ]</pre>



	<pre> ] secuencia = [0, 2, 3, 4, 1] </pre>
Inicializo dp y prev	<pre> dp = [1, 1, 1, 1, 1] prev = [-1, -1, -1, -1, -1] </pre>
Itero y obtengo lo siguiente:	

i	j	secuencia[i]	secuencia[j]	dp[i]	dp[j] + 1	prev[i]	dp	prev
1	0	2	0	4 2	2	-4 0	[1, 2, 1, 1, 1]	[-1, 0, -1, -1, -1]
2	0	3	0	4 2	2	-4 0	[1, 2, 2, 1, 1]	[-1, 0, 0, -1, -1]
2	1	3	2	2 3	3	0 1	[1, 2, 3, 1, 1]	[-1, 0, 1, -1, -1]
3	0	4	0	4 2	2	-4 0	[1, 2, 3, 2, 1]	[-1, 0, 1, 0, -1]
3	1	4	2	2 3	3	0 1	[1, 2, 3, 3, 1]	[-1, 0, 1, 1, -1]
3	2	4	3	3 4	4	4 2	[1, 2, 3, 4, 1]	[-1, 0, 1, 2, -1]
4	0	1	0	4 2	2	-4 0	[1, 2, 3, 4, 2]	[-1, 0, 1, 2, 0]
4	1	1	2	2	3	0	[1, 2, 3, 4, 2]	[-1, 0, 1, 2, 0]
4	2	1	3	2	4	0	[1, 2, 3, 4, 2]	[-1, 0, 1, 2, 0]
4	3	1	4	2	5	0	[1, 2, 3, 4, 2]	[-1, 0, 1, 2, 0]

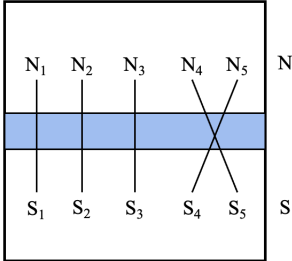
Obtengo el índice de la máxima longitud, a partir de dp	idx: 3
Reconstruyo la solución a partir del índice idx y las agrego en indices_solucion	indices_solucion = [3, 2, 1, 0]
Invierto indices_solucion y construyo propuestas_finales a partir de evaluar propuestas_ordenadas en cada índice de indices_solucion	<pre> indices_solucion = [0, 1, 2, 3] propuestas_finales = [     ('Barrancas', 'Cienagas'),     ('San Pedro', 'Barrio Este'),     ('El hurón', 'Portuario'),     ('Palo seco', 'Torre verde'), ] </pre>

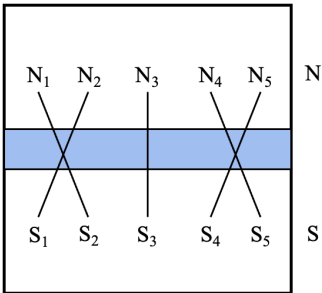
## 6 - ¿La complejidad de su propuesta es igual a la de su programa?

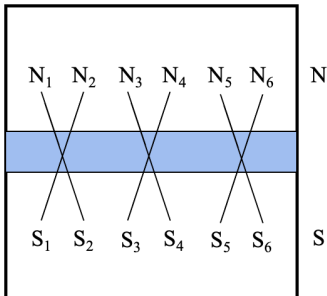
Si, ya que en el pseudocódigo se explica con qué estructuras se pensó la solución, y el código utiliza las mismas estructuras y respeta las complejidades de la propuesta.

7 - Analice: ¿El resultado que retorna su programa es el único posible? De un contraejemplo si no lo es o una demostración si lo es. En caso de no serlo, ¿podrían modificar rápidamente para obtener una solución diferente?

No, porque podríamos haber ordenado las propuestas por el orden del sur y obtener otro resultado. Además, cuando hay más de una subsecuencia creciente de la misma longitud, el algoritmo elige la que aparece primero al recorrer la secuencia. En los siguientes contraejemplos se van a poder apreciar esas cuestiones.

<p>CONTRAJEJEMPLO 1:</p> 	<p><b>SOL1: (1,1) (2,2) (3,3) (4,5)</b>  SOL2: (1,1) (2,2) (3,3) (5,4)</p>
--	--

<p>CONTRAJEJEMPLO 2:</p> 	<p><b>SOL1: (1,2) (3,3) (4,5)</b>  SOL2: (1,2) (3,3) (5,4)  SOL3: (2,1) (3,3) (4,5)  SOL4: (2,1) (3,3) (5,4)</p>
---	--

<p>CONTRAJEJEMPLO 3:</p> 	<p><b>SOL1: (1,2) (3,4) (5,6)</b>  SOL2: (1,2) (3,4) (6,5)  SOL3: (1,2) (4,3) (5,6)  SOL4: (1,2) (4,3) (6,5)  SOL5: (2,1) (3,4) (5,6)  SOL6: (2,1) (3,4) (6,5)  SOL7: (2,1) (4,3) (5,6)  SOL8: (2,1) (4,3) (6,5)</p>
--	--

\*las soluciones marcadas en **negrita** son las que devolvería nuestro algoritmo

## Requisitos y procedimientos para compilar el código

---

Para cada parte ejecutar su respectivo comando por terminal:

Parte2:

```
$ puentes_dq.py barrios.txt propuestas.txt
```

Parte3:

```
$ puentes_pd.py barrios.txt propuestas.txt
```

## Correcciones con respecto a la 1er entrega

---

Para la parte 1:

3) Optimalidad caso J: Se tiene una solución óptima  $O$  con  $p$  la cantidad de pilas. Si la solución greedy que obtengo al aplicar el algoritmo me devuelve  $p+1$  pilas, quiere decir que en algún momento la elección Greedy no ubico una carta en alguna de las pilas disponibles (en las cuales se podía colocar). Pero esto es absurdo porque el óptimo local minimiza la cantidad de pilas creadas por iteración.

Para la parte 3:

Definición de subproblema: La solución de un problema de tamaño  $i$  es la misma que la solución para el problema de tamaño  $i-1$ , agregándole un puente adicional si es factible, o no en caso contrario.

## Referencias y enlaces externos

---

- Material de Edpuzzle referido a División y Conquista, Programación Dinámica