FACULTAD DE INGENIERÍA
Universidad de Buenos Aires

75.29 / 95.06
Teoría de Algoritmos 1 - 1c 2025

Trabajo Práctico Nº 1

ACORN

Fecha de Entrega:

21 de abril de 2025

Integrantes:

Alumno	Padrón
Tomás Arián Lofano	101721
Rogger Aldair Paredes Tavara	97976
Santiago Nahuel Ruiz Sugliani	106768
Maria Mercedes Slepowron Majowiecki	109454
Jose Ignacio Adelardi	111701

Parte 1: Solitario

1. ¿Todas las propuestas fueron greedy?

Sí, todas las propuestas individuales del trabajo práctico 0 fueron greedy. Comparando una a una las diferentes propuestas de todos los integrantes se pueden llegar a observar diferencias de implementación, pero en cada una de ellas se ha tomado la misma estrategia greedy. Dicha estrategia consiste en insertar la carta desapilada (y desconocida hasta ese momento) e intentar insertarla en alguna pilas existentes, siempre y cuando la carta superior sea de mayor denominación a la carta desapilada. Esto generará un posible subset de pilas candidatas a recibir dicha carta. Dentro de este conjunto, tomaremos la pila cuyo extremo contenga la menor denominación entre los extremos disponibles. En el caso que no exista una pila que cumpla la condición descrita anteriormente, se crea una nueva pila para colocar la carta desapilada.

2. ¿Cuáles fueron las complejidades de cada una de ellas? ¿Podemos encontrar entre ellas mejores y peores?

Las complejidades de las distintas propuestas fueron en todos los casos la misma:

- Complejidad temporal: $O(n^2)$
- Complejidad espacial: $O(n)$

Si bien las complejidades teóricas coinciden entre las diferentes implementaciones, pueden observarse algunas decisiones de diseño que mejoran aspectos prácticos del algoritmo. Algunas de estas diferencias incluyen:

- En varias implementaciones se realiza una búsqueda lineal sobre todas las pilas para encontrar aquella cuyo tope sea mayor que la carta actual, pero a su vez, el menor de los topes. Teniendo en cuenta que los topes de las pilas se mantienen ordenados de menor a mayor, es posible optimizar esta búsqueda utilizando una estrategia de búsqueda binaria, lo cual reduciría la complejidad temporal a O(n log n). Ninguna de nuestras implementaciones optó por dicha estrategia.
- Una de las implementaciones únicamente guarda el valor del tope de cada pila, en lugar de almacenar todas las cartas de cada pila. Si bien esta diferencia no modifica la complejidad espacial del algoritmo, si presenta un mejor nivel de mejora práctica que puede tener impacto positivo en el rendimiento del algoritmo.

3. Fueron todas óptimas? En caso negativo, brinde un contraejemplo para las que no. Y realice la demostración de optimalidad para una de las que sí. (si ninguna fue óptima, deben proponer una que lo sea)

Sí, todas las propuestas fueron óptimas ya que en todos los trabajos prácticos individuales se compartió la misma estrategia greedy. En particular se demostrará la optimalidad de la elección greedy específica: “Las pilas de cartas disponibles estarán ordenadas de menor a mayor dependiendo la denominación de la carta que se encuentre en su tope. Al desapilar una carta se la inserta en primera

pila de cartas que cumpla con la condición de que el tope sea mayor que la carta desapilada. En caso de que ninguna pila cumpla con la condición significa que la carta elegida es la mayor de todas las cartas tope y se debe crear una nueva pila de cartas que será ubicada al final de las pilas, asegurando que los topes de las pilas sigan ordenados de menor a mayor en todo momento."

Dado un conjunto de cartas $C=\{C_1,C_2,\ldots,C_n\}$ y considerando las pilas ordenadas de menor a mayor en base a la denominación de la carta tope.

Se definen:

- O → Es la solución óptima con el mínimo número de pilas posible
- P → Es una solución al problema

Se procederá a demostrar si O=P, para ello se asume que P>O ⇒ Dado que la solución al problema contiene un número mayor de pilas que la solución óptima, se implica que hay al menos una carta que fue colocada en una nueva pila, cuando podría haber sido colocada en alguna de las pilas existentes, pero dado que:

1. El algoritmo tiene las pilas candidatas ordenadas de mayor a menor (según el tope) y coloca la carta en la primera pila que tenga una carta estrictamente mayor a la carta desapilada
2. En caso que el algoritmo no encuentra una pila que cumpla la condición establecida en el paso anterior, se crea una pila nueva

Habíamos asumido que P>O ⇒ Hubo alguna carta C_k que se colocó en una nueva pila cuando debería haber ido a una pila existente, pero dado que esto no es posible porque el algoritmo coloca la carta desapilada en la pila cuya carta superior sea mayor a C_k ⇒ Es absurdo que P>O ⇒ Por lo tanto P=O ⇒ P es óptimo.

Parte 2: Los puentes que se cruzan

1. Presentar una propuesta mediante división y conquista que resuelva este problema.

Nuestra propuesta de división y conquista toma por parámetro una lista de propuestas representadas por tuplas de la forma (N, S) siendo N y S el índice del barrio norte y sur respectivamente, los cuales serán unidos por un puente. Estos índices indican desde qué barrio del norte hacia qué barrio del sur cruza la propuesta en cuestión al río. Supongamos que tenemos los barrios del norte “Yrigoyen” (0), “Barracas” (1), “La Boca” (2) y los barrios del sur “Piñeyro” (0), “Pueyrredón” (1) e “Isla Maciel” (2). Habiendo enunciado la forma de estas tuplas, ordenaremos esta lista pasada por parámetro según N de forma ascendente.
Como paso base tenemos el caso donde la lista del llamado actual tiene una sola propuesta, en este caso devolvemos una lista con ella y un cero que representa la cantidad de cruces para tal llamado. En caso contrario dividiremos la lista del llamado actual a la mitad, llamando recursivamente para ambas mitades y posteriormente juntando las soluciones resultantes.
Una vez obtengamos los resultados para ambas mitades del llamado actual, debemos juntar sus soluciones, para esto haremos uso de:

- Un arreglo al cual iremos agregando las propuestas a medida que juntamos ambas soluciones.
- Índices que apuntarán a las propuestas de las soluciones de la primera y segunda mitad que estaremos evaluando (i, j respectivamente).

Por otro lado debemos considerar que:

- Ordenamos la lista original en un principio (según el índice del barrio norte).
- Sabemos de antemano que un barrio no puede estar en más de una propuesta por condición del enunciado.
- Para que se de un cruce entre propuestas debe ocurrir que la propuesta de la lista izquierda (nos referimos a ella anteriormente como solución de la primera mitad) tenga como barrio sur (llamémoslo S_i) un valor mayor al barrio sur de la lista derecha (S_d) y en caso de que S_i < S_d no hay cruces entre propuestas.

Dicho esto, tenemos dos casos posibles en donde encontrarnos:

1) S_i < S_d
   En este caso las propuestas no se cruzan, para los ejemplos nombrados al principio pensemos en las propuestas (Yrigoyen, Piñeyro) y (La Boca, Isla Maciel) para nuestra propuesta representan (0, 0) y (2, 2). Por lo tanto no se contabilizan cruces entre puentes, se agrega (Yrigoyen, Piñeyro) al arreglo y se actualiza el índice i en i + 1.
2) S_i > S_d
   Este es el caso donde se observa un cruce de puentes, supongamos el caso de (Yrigoyen, Isla Maciel) y (La Boca, Piñeyro) que para nuestra propuesta representa (0, 2), (2, 0). Aquí se agrega (La Boca, Piñeyro) al arreglo, se actualiza el índice j en j + 1 y se contabilizan los cruces que esto representa. La cantidad de cruces se verá calculada como la cantidad de propuestas en la lista izquierda menos el índice i, lo cual incluye a la propuesta apuntada por el índice i más las restantes (ninguna en este caso).

Una vez alguno de los índices haya alcanzado un valor igual al largo de la mitad que representa, se agregan al arreglo los elementos restantes de la lista izquierda y posteriormente se agregan al mismo los elementos restantes de la lista derecha. Finalmente se devuelve el arreglo y la cantidad de cruces.

2. Brindar la ecuación de recurrencia y obtener su complejidad mediante el teorema maestro.

Al recurrir al teorema maestro tenemos:

A = 2 subproblemas por cada nivel de recursión

B = 2 partes iguales en las cuales dividimos al subproblema en cada nivel.

f(n) = n dado juntamos esas partes iguales realizando comparaciones en O(1)
y posteriormente agregamos los elementos restantes al arreglo a devolver
(con n la cantidad de elementos en el subproblema actual).

Vamos a comenzar probando con el caso número dos de teorema, donde igualamos:

$$f(n) = \Theta\left(n^{\log_B A}\right)$$

Tras reemplazar con las variables de nuestro problema tenemos que:

$$n = \Theta\left(n^{\log_2 2}\right)$$

$$n = \Theta\left(n^{1}\right)$$

$$n = \Theta\left(n\right)$$

Finalmente observamos que se cumple satisfactoriamente que $f(n) = n$
sea acotada tanto superior como inferiormente por n
y por lo tanto se tiene que:

$$T(n) = \Theta\left(n^{\log_B A} * log * n\right)$$

$$T(n) = \Theta\left(n^{\log_2 2} * log * n\right)$$

$$T(n) = \Theta\left(n * log * n\right)$$

3. Presentar pseudocódigo.

```
evaluar_factibilidad(puentes_propuestos):
    merge_sort(puentes_propuestos, N asc)
    devolver _evaluar_factibilidad(propuestas_ordenadas, 0, len(propuestas_ordenadas) - 1

_evaluar_factibilidad(puentes_propuestos, ini, fin):
    Si me queda una sola propuesta la devuelvo junto con cero cruces

    solucion_izq, cruzados_izq = _evaluar_factibilidad(puentes_propuestos, primera_mitad)
    solucion_der, cruzados_der = _evaluar_factibilidad(puentes_propuestos, segunda_mitad)

    devolver merge_puentes(solucion_izq, solucion_der, cruzados_izq + cruzados_der)

merge_puentes(solucion_izq, solucion_der, cruzados):
    i = j = 0
    resultado = []

    Mientras i < len(izq) y j < len(der):
        propuesta_izq = izq[i]
        propuesta_der = der[j]
        Si la propuesta izquierda termina en un barrio anterior a la derecha:
            res.resultado(propuesta_izq)
            i += 1
        Caso contrario:
            res.resultado(propuesta_der)
            cruzados += cantidad_restante_en_solucion_izq
            j += 1

    Agrego los restantes de solucion_izq si los hay
    Agrego los restantes de solucion_der si los hay

    devolver resultado, cruzados
```

4. Brindar un ejemplo de funcionamiento.

Recordemos los barrios dados como ejemplo en la propuesta de la solución junto con su órden y por ende su índice dentro la parte de la ciudad a la cual corresponden.
Los barrios del norte:

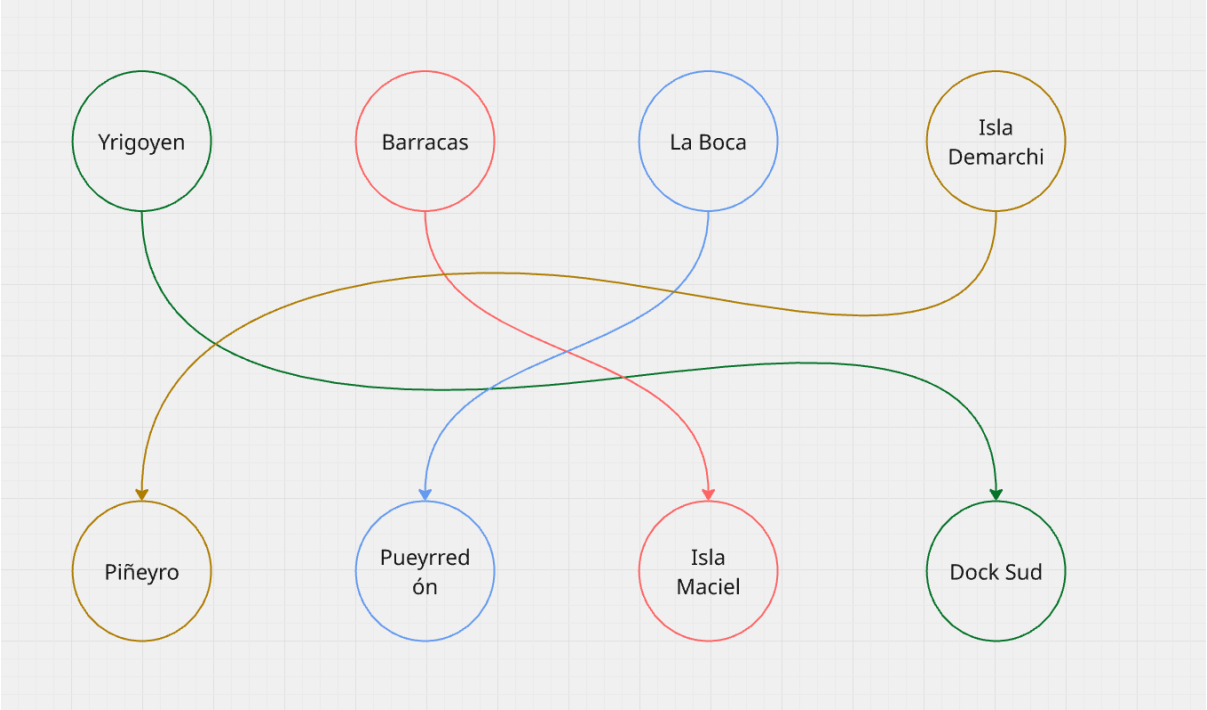
- Yrigoyen (0)
- Barracas (1)
- La Boca (2)
- Isla Demarchi (3)

Los barrios del sur:

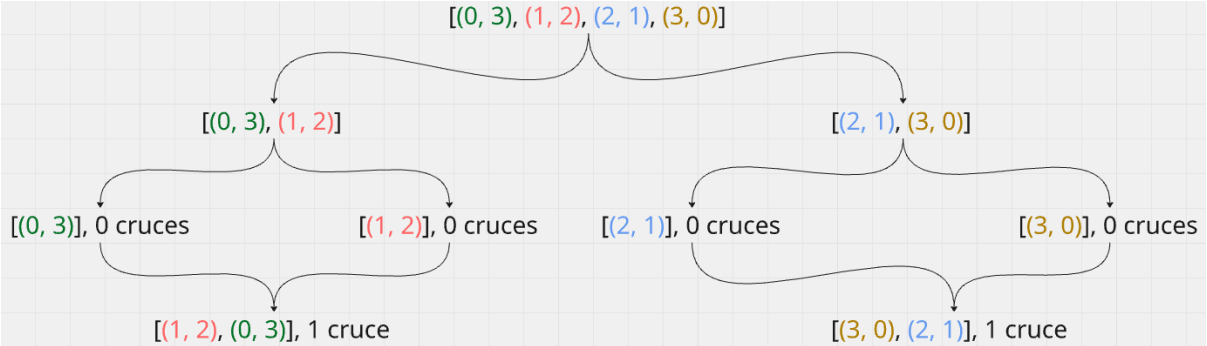
- Piñeyro (0)

- Pueyrredón (1)
- Isla Maciel (2)
- Dock Sud (3)

Teniendo en cuenta estos barrios y su órden, nuestro ejemplo tomará por parámetro la siguiente lista: [(Barracas, Isla Maciel), (La Boca, Pueyrredón), (Isla Demarchi, Piñeyro), (Yrigoyen, Dock Sud)], y por lo tanto [(1, 2), (2, 1), (0, 3), (3, 0)] será la entrada correspondiente.



Tras el ordenamiento realizado por nuestro algoritmo previo a comenzar la división y conquista, la lista queda en el siguiente estado [(0, 3), (1, 2), (2, 1), (3, 0)] y se invoca a la función recursiva provocando que el arreglo se divida en dos mitades de dos elementos cada uno, por un lado [(0, 3), (1, 2)] y por el otro [(2, 1), (3, 0)]. Posteriormente (y considerando el lado izquierdo, es decir [(0,3), (1, 2)] como subproblema) se vuelve a dividir el arreglo en dos mitades que son [(0,3)] y [(1, 2)] respectivamente, en cada uno de ellos llegamos al caso base y devuelven cada uno el mismo sub-arreglo y una cantidad de cruces igual a cero (ídem para el subproblema [(2, 1), (3, 0)]).
Veamos cómo se juntan las soluciones al subproblema [(0, 3), (1, 2)], en este caso al tratarse de dos elementos habrá una única comparación donde observamos que para la propuesta izquierda (0, 3) el barrio sur es posterior al de la propuesta derecha (1, 2) siendo entonces que al subarreglo resultante se agrega (1, 2) y se contabiliza 1 cruce dado que del lado izquierdo había una sola propuesta, posteriormente se agregará al arreglo resultante la propuesta (0, 3). Al juntarse las soluciones al subproblema [(2, 1), (3, 0)] sucede exactamente lo mismo y por lo tanto se contabiliza un único cruce como se puede observar en la imagen siguiente.



Ahora resta juntar los resultados de la solución de cada mitad respecto del primer llamado, al comenzar comparamos las propuestas (1, 2) y (3, 0) donde observamos que la propuesta izquierda "termina" luego de la propuesta derecha, encontrando un cruce. De esta forma se agrega (3, 0) al arreglo resultante, se actualiza el índice J y se contabilizan 2 cruces dado que (3, 0) también se cruza con (0, 3), es decir, una vez encontrado un cruce en realidad existen LEN(ARREGLO_IZQ) - I cruces dado que el subproblema de la izquierda se encuentra ordenado ascendentemente por el barrio sur y debido a que el barrio sur de la derecha tiene un índice menor al apuntado por el índice I además de un valor mayor en el barrio norte (por el ordenamiento realizado al principio) es que las LEN(ARREGLO_IZQ) - I propuestas del subproblema izquierdo se cruzan con el mismo. Tras esta iteración tenemos entonces que el arreglo resultante es [(3, 0)] y se contabilizan 2 + 2 = 4 cruces (por los 2 que habíamos encontrado al resolver cada subproblema previamente).

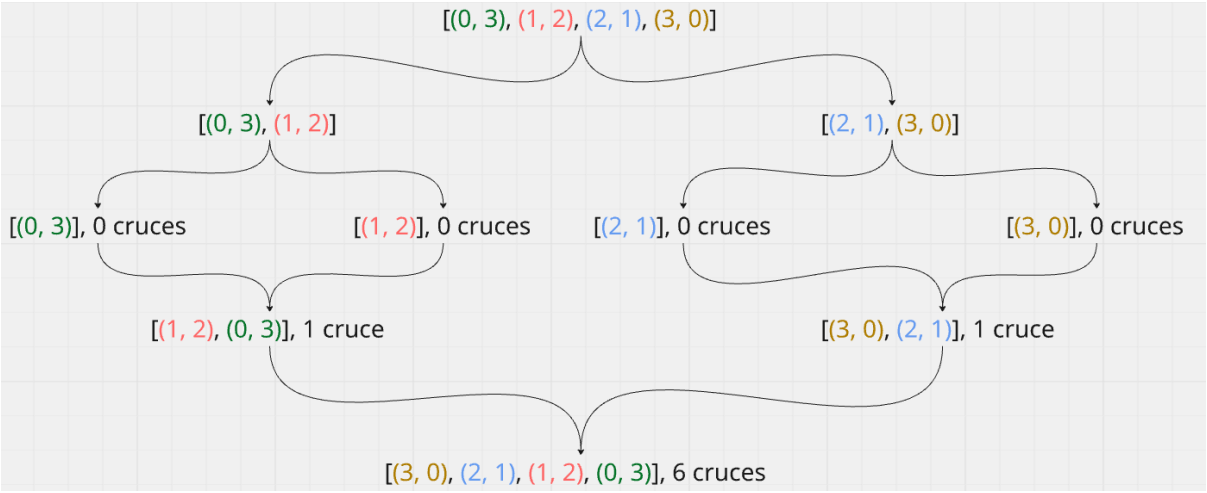
N° Iteración	Valor índice I	Valor índice J	Propuesta Izquierda	Propuesta Derecha	Arreglo Resultante	Cantidad de Cruces
1	0	0	(1, 2)	(3, 0)	[(3, 0)]	4

Resta realizar una nueva comparación donde la propuesta izquierda sigue siendo (1, 2) y la nueva propuesta derecha es (2, 1) donde vuelve a darse un cruce. Agregamos (2, 1) al arreglo resultante y contabilizamos dos cruces más ya que como dijimos anteriormente, siendo que el arreglo izquierdo se encuentra ordenado de forma ascendente según el barrio sur por llamados anteriores y los barrios del norte son necesariamente menores a los de las propuestas de la derecha, eso implica que, al encontrar una propuesta J que se cruza con alguna propuesta I, esto significa que las propuestas del arreglo izquierdo con K >= I (siendo K su posición) también se cruzan con J.

N° Iteración	Valor índice I	Valor índice J	Propuesta Izquierda	Propuesta Derecha	Arreglo Resultante	Cantidad de Cruces
1	0	0	(1, 2)	(3, 0)	[(3, 0)]	4
2	0	1	(1, 2)	(2, 1)	[(3, 0), (2, 1)]	6

Finalmente se agregan al arreglo resultante las propuestas del subproblema izquierdo y se devuelve el mismo junto con la cantidad de cruces, en la siguiente imagen se encuentra el resultado de la división

y conquista realizada por el algoritmo, como se puede observar el arreglo resulta ordenado según el barrio sur de forma ascendente.



5. Presentar un programa en python que resuelva el problema.

La solución de este problema se encuentra en la carpeta adjuntada en la entrega, siendo el archivo ejecutable ***puentes_dq.py.*** Se adjuntan también dos archivos de prueba: *barrios.txt y propuestas.txt* para poder probar el programa.
Para poder ejecutarlo, correr la siguiente línea:

```
python3 puentes_dq.py barrios.txt propuestas.txt
```

Siendo el archivo barrios.txt el archivo con los nombres de los barrios y propuestas.txt el archivo con la lista de las propuestas para los cruces.

6. Analizar: Es la complejidad de su programa igual al de su propuesta. Justifique.

Por un lado, dado el análisis realizado en el punto 2 de esta sección, habiendo demostrado la aplicabilidad del teorema maestro para nuestra solución propuesta, podemos afirmar que la complejidad temporal de la propuesta realizada es de ***O(n log n).*** En cuanto a la complejidad espacial, esta misma resulta ser ***O(n),*** dado que lo que se necesita almacenar es la lista de las propuestas simplemente.

Por otro lado, analizando el programa desarrollado:

- Primero se realiza una lectura de los archivos. Siendo n la cantidad de propuestas, debemos leer a lo sumo *2n* líneas para el archivo de barrios, y *n* líneas para el archivo de propuestas. Además, se le asigna un índice numérico a cada barrio para poder operar fácilmente con ellos a lo largo del programa, lo cual es una operación constante. La complejidad de esto termina siendo O(n)
- Luego, se realiza una operación de merge sort para evaluar la factibilidad de las propuestas realizadas. En cada nivel de recursión, se realizan operaciones lineales para combinar y ordenar las listas. La complejidad de este proceso resulta O(n log n)

Acotando por *n log n,* llegamos a la misma conclusión de complejidad que para la solución propuesta, siendo la complejidad temporal del programa planteado: ***O(n log n)***

En cuanto a la complejidad espacial del algoritmo, esta misma también coincide con la de la propuesta: ***O(n),*** ya que únicamente estamos almacenando la lista de propuestas, siendo *n* la cantidad de puentes propuestos.

Parte 3: Maximizando los puentes

1. Resolver el problema utilizando programación dinámica. (incluya en su solución definición del subproblema, relación de recurrencia y pseudocódigo)

Definimos el subproblema como:
Sea *OPT[i]* el largo de la subsecuencia más larga de puentes no cruzados que finaliza en el puente i

Definimos la relación de recurrencia:
$OPT[i] = max(OPT[j]) + 1$ *con j < i, j un puente compatible a i*
OPT[i] = 1 ya que podemos pensar que en un principio la cantidad óptima de puentes a construir hasta el puente i es solo el puente i.

Pseudocódigo de la solución propuesta:

```
evaluar_factibilidad_pd(puentes_propuestos):
    n = cantidad de puentes propuestos
    opt = vec[] //tamaño n, inicializado con todos 1
    anterior = vec[] //tamaño n, inicializado con todos -1
    index_ultimo_ubicado = 0

    for i desde 1 hasta n - 1{
        for j desde 0 hasta i - 1{
            if el barrio sur de j < barrio sur de i y opt[j] + 1 > opt[i]{
                anterior[i] ← j
                opt[i] ← opt[j] + 1
                if opt[i] > opt[index_ultimo_ubicado]{
                    index_ultimo_ubicado ← i
                }
            }
        }
    }

    cant_puentes_construibles = opt[index_ultimo_ubicado]
    puntes_seleccionados = vec[]
    while index_ultimo_ubicado != -1:
        puntes_seleccionados.agregar(puentes_propuestos[index_ultimo_ubicado])
        index_ultimo_ubicado = anterior[index_ultimo_ubicado]

    puntes_seleccionados.reverse()
    return cant_opt, res
```

2. Explique por qué su propuesta funciona.

Nuestra propuesta utiliza programación dinámica y memorización para encontrar la mayor cantidad de puentes que se pueden construir sin que se crucen dado que:

Primero, se ordenan las propuestas de puentes según el orden de los barrios del norte. Este ordenamiento garantiza que, si se eligen puentes en ese orden, no habrá cruces por el lado norte. Por lo tanto, solo es necesario verificar que los extremos del lado sur también mantengan ese orden para que los puentes sean compatibles.

Luego, para cada puente, el algoritmo evalúa todos los puentes anteriores. Si encuentra uno cuya conexión en el sur es menor y que permite una mejor solución hasta ese punto, actualiza el valor óptimo en un arreglo de memorización (opt). Dicho arreglo guarda, para cada posición, la cantidad máxima de puentes compatibles que se pueden construir hasta ese puente inclusive.

Además, se utiliza una estructura adicional (anterior) para recordar el camino recorrido, es decir, de qué puente precede al actual. Gracias a esto, al finalizar la ejecución, se puede reconstruir la secuencia exacta de puentes elegidos sin necesidad de recalcular ninguna resolución.

Este enfoque evita resolver nuevamente subproblemas ya evaluados, lo que reduce significativamente el tiempo de ejecución y asegura que se obtenga una solución **óptima** sin cruces, aplicando una estrategia eficiente basada en **memorización**.

Este enfoque garantiza:

- Que los puentes elegidos no se crucen ni por el norte ni por el sur.
- Que se obtenga la mayor cantidad posible de puentes compatibles.
- Que el tiempo de ejecución sea eficiente, ya que se evita recalcular gracias al uso de memorización.

En resumen, el algoritmo encuentra la solución óptima aplicando programación dinámica, almacenando resultados parciales y reconstruyendo el camino final sin necesidad de evaluar todas las combinaciones posibles, lo cual lo convierte en una estrategia eficiente, clara y correcta.

3. Analice la complejidad espacial y temporal de su propuesta.

Complejidad temporal:
La complejidad temporal es de O(n^2), siendo n la cantidad de puentes propuestos. Se debe al siguiente for anidado:

```
for i desde 1 hasta n - 1{
      for j desde 0 hasta i - 1{
```

Para cada propuesta “ i” se calcula cuál es la mejor solución considerando todas las propuestas anteriores “j”. En cada iteración se verifica si se puede extender una solución óptima anterior sin que

haya cruces, lo que hace que se deba comparar las posiciones de los barrios del sur. Entonces, en el peor de los casos se realizan aproximadamente n(n−1)/2 comparaciones, es decir O(n^2).

Complejidad espacial:
La complejidad espacial es de O(n), ya que se utilizan tres vectores de tamaño n: opt[ ], anterior[ ] y puentes_seleccionados[ ]. Cada vector almacena como máximo n elementos entonces, el uso de memoria crece linealmente con el número de puentes propuestos.

4. De un breve ejemplo paso a paso del funcionamiento de su propuesta.

Suponemos para el ejemplo que tenemos 3 barrios en cada margen, en el norte tenemos Yrigoyen (0), Barracas (1) y La Boca (2), y al sur tenemos Piñeyro (0), Pueyrredón (1) e Isla Maciel (2) . Y teniendo en cuenta estos barrios y su órden la propuestas de sería: (Yrigoyen, Pueyrredón), (Barracas, Piñeyro) y (La Boca, Isla Maciel); o (0,1), (1,0) y (2,2).
Al empezar el algoritmo, cada puente es considerado como una posible solución por sí solo, por lo que inicializamos todo en 1. Después al comparar el segundo puente con el primero, (Barracas, Piñeyro) con (Yrigoyen, Pueyrredón), vemos que no se puede construir junto a él porque se cruzan, así que no se pueden construir juntos, Opt[0] = Opt[1] = 1 donde también Anterior[0] = Anterior[1] = -1. Luego, al llegar al tercer puente (La Boca, Isla Maciel) y lo comparamos con el primero (Yrigoyen, Pueyrredón) vemos que no se cruzan, lo que significa que es posible extender la solución anterior. Actualizamos la mejor cantidad de puentes sin cruces hasta ese punto a 2 (Opt[2] = 2), y marcamos que viene desde el primero (Anterior[2] = 0), dado que sucede lo mismo al comparar con (Barracas, Piñeyro) no se obtiene un mejor óptimo al ya encontrado. Finalmente, al reconstruir la solución desde el último puente válido, tenemos los puentes 0 y 2, que se pueden construir sin que se crucen entre sí. Así, el algoritmo encuentra la máxima cantidad de puentes sin cruces que es de 2, y devuelve UNA de las posibles combinaciones .

5. Programe su solución.

La solución de este problema se encuentra en la carpeta adjuntada en la entrega, siendo el archivo ejecutable ***puentes_pd.py.*** Se adjuntan también dos archivos de prueba: *barrios.txt y propuestas.txt* para poder probar el programa.
Para poder ejecutarlo, correr la siguiente línea:

```
python3 puentes_pd.py barrios.txt propuestas.txt
```

Siendo el archivo barrios.txt el archivo con los nombres de los barrios y propuestas.txt el archivo con la lista de las propuestas para los cruces.

6. Analice: ¿La complejidad de su propuesta es igual a la de su programa?

Si, la complejidad de nuestra propuesta y la del programa son iguales. En ambos casos se realizan dos bucles anidados que comparan todas las combinaciones posibles de puentes previos y verificar si se construyen sin cruzarse. Esto implica una complejidad de O(n^2), siendo n la cantidad el número de propuestas. También ambos usan 3 vectores que crecen linealmente con n elementos (opt, anterior y res) , por lo que la complejidad espacial también es O(n).

En conclusión, nuestro programa es una traducción directa y fiel a nuestra propuesta descrita en el pseudocódigo, y comparten las mismas complejidades, tanto espaciales como computacionales.

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, el resultado que devuelve el programa no es el único posible. Nuestro algoritmo usa programación dinámica resolviendo los subproblemas asociados a cada propuesta, y encuentra una posible solución óptima, un solo conjunto máximo de puentes que no se cruzan. Pero pueden existir varias soluciones distintas con la misma cantidad de puentes pero con distintas combinaciones. La solución está determinada por el orden en que se recorren las propuestas, si se invirtiera el orden, la solución podría variar.
Un contraejemplo sencillo para demostrar que nuestro programa no retorna la única solución posible sería:
Tenemos 3 barrios en cada margen, en el norte tenemos Yrigoyen (0), Barracas (1) y La Boca (2), y al sur tenemos Piñeyro (0), Pueyrredón (1) e Isla Maciel (2) . Y teniendo en cuenta estos barrios y su órden la propuestas sería: (Yrigoyen, Pueyrredón), (Barracas, Piñeyro) y (La Boca, Isla Maciel); o (0,1), (1,0) y (2,2). Al buscar la subsecuencia creciente más largas usando estos índices para evitar cruces, encontramos 2 posibles soluciones óptimas de igual longitud: (1,0) y (2,2) o también (0,1) y (2,2). Ambas son válidas, ambas tienen la cantidad máxima de puentes sin cruces, pero son diferentes. Este ejemplo sencillo demuestra que el resultado que devuelve el programa no es único.

Si quisiéramos obtener un resultado correcto y óptimo pero diferente bastaría con cambiar el orden en que se recorren las propuestas en la etapa de programación dinámica, por ejemplo usando un recorrido inverso en el segundo bucle., que sería cambiar la línea:
`for j in range(i):`
por:
`for j in reversed(range(i)):`
Esto cambiaría el camino que se elige al reconstruir la solución óptima y permitiría encontrar una combinación dentista de puentes sin cruces pero con la misma cantidad máxima.