FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo práctico n°2

Integrantes:

Dávila, Rebeca Analía	108672
Camilo Sassone Irrazabal	111135
Gabriela Yanes	108325
Ulises Distéfano	111883

Materia: Teoría de algoritmos

Curso: 03 - Echeverría

Cuatrimestre: 1er. cuatrimestre 2025

Índice

1. Problema de Programación Dinámica

Enunciado del problema:

Cualquier cadena puede ser descompuesta en secuencias de palíndromos. Por ejemplo, la cadena ARACALACANA se puede descomponer de las siguientes formas:

ARA CALAC ANA
ARA C ALA C ANA
A R A CALAC A N A
etc.

Desarrollar un algoritmo de programación dinámica que encuentre el menor número de palíndromos que forman una cadena dada. Por ejemplo, para ARACALACANA debería devolver 3.

1. **Supuestos**:

Se supone que:

1. La entrada es una cadena **no vacía** compuesta por caracteres, como números , letras o caracteres especiales.
2. La cadena ingresada está compuesta por caracteres colocados de manera consecutiva, sin separación explícita entre subcadenas y caracteres. Esto no quiere decir que no acepta espacios como caracter válido, pero si se lo usa como separador de subcadenas y caracteres de la cadena ingresada el resultado de la función podría ser incorrecto, ya que se supone que la cadena es un valor único sin separaciones.
3. Toda subcadena de un solo caracter es considerada un palíndromo válido.
4. Existe una función *es_palindromo* que verifica correctamente si una subcadena es un palíndromo.
5. Si se trata de una palabra, no debe tener acentos. Aunque esto no significa que no se acepte el caracter propio de la acentuación.

6. Se espera que la cadena esté completamente en minúsculas o mayúsculas ya que el algoritmo es sensible a letras mayúsculas y minúsculas.

Condiciones:

1. La cadena es indexable, es decir soporta operaciones de **slicing** "cadena[i:j]".
2. La cadena no contiene separaciones, está normalizada y sin espacios en ella.

Limitaciones:

1. El algoritmo solo devuelve el número mínimo de palíndromos, no las particiones.
2. No maneja cadenas vacías, es decir que no se considera una cadena vacía un palíndromo.
3. Es sensible a mayúsculas y minúsculas, por ejemplo, "Ana" no es palíndromo.
4. Al asumir que no hay separación entre caracteres o subcadenas de la cadena puede devolver resultados erróneos, por ejemplo si se ingresa: "ananá ", devolvería dos, cuando probablemente el usuario esperaba uno.
5. Si se ingresan palabras acentuadas, estas se evaluarán tal cual aparecen, y serán consideradas diferentes de sus versiones sin acento.

2. **Diseño:**

Ecuación de recurrencia:

Se define la función $OPT[i]$:

Sea $OPT[i]$ *el mínimo número de palíndromos en los que se puede descomponer el prefijo de longitud i de la cadena.*

La idea es recorrer todas las posibles posiciones **j < i** y verificar si la subcadena "**cadena[j:i]**" es un palíndromo. Si lo es, se puede formar una partición desde j hasta i (exclusivo), por lo tanto podemos considerar la opción $OPT[j] + 1$, que

sería el mínimo de palíndromos para el prefijo “cadena[0:j]” sumado al nuevo palíndromo encontrado “cadena[j:i]”.
Buscamos todas estas opciones para j < i y nos quedamos con la mínima, y ese es nuestro óptimo para el prefijo cadena[0:i]:

$$OPT[i] \;=\; min_{0 <= j < i} \{OPT[j] \;+\; 1 \; si \; la \; cadena[j{:}\; i] \; es \; palíndromo\}$$

- La solución final es $OPT[n]$, siendo n el largo de la cadena ingresada.

Requisitos:

Para poder utilizar programación dinámica se deben cubrir los requisitos de:

- Subestructura Óptima
- Subproblemas Superpuestos

Subestructura Óptima: En nuestra solución se puede observar que el requisito de subestructura óptima se cubre ya que el óptimo para el prefijo de largo i se obtiene a partir de el mínimo OPT[j] tal que cadena[j:i] sea un palíndromo. Como OPT[j] es la solución óptima al subproblema j < i, y lo usamos para construir el problema OPT[i], se cumple esta propiedad.

Problemas Superpuestos: Esta propiedad también se cumple en nuestra solución ya que vuelven a aparecer subproblemas resueltos previamente al calcular la solución de OPT[i]. A medida que se calcula OPT[i] para diferentes valores de i, se reutilizan los mismos valores de OPT[j] j < i.

Memoization: En la solución planteada se utiliza memoization guardando los valores de OPT[j] j < i, que se utilizan para construir soluciones de los siguientes j < i <= n. En este caso se utiliza un arreglo de largo n + 1 (n siendo la longitud de la cadena ingresada), esto evita recálculos innecesarios.

Pseudocódigo:

```
función ES_PALINDROMO(s):
    retornar s == reverso(s)

función OBTENER_MINIMO_PALINDROMOS(cadena):
```

```
n ← longitud(cadena)
crear arreglo OPT de tamaño n + 1
OPT[0] ← 0

Para i desde 1 hasta n:
    minimo ← infinito
    para j desde 0 hasta i - 1:
        si ES_PALINDROMO(cadena[j:i]):
            minimo ← mínimo(minimo, OPT[j] + 1)
    OPT[i] ← minimo

retornar OPT[n]
```

Para la solución planteada se utilizaron las siguientes estructuras de datos:

- Arreglo OPT de tamaño n + 1, n siendo la longitud de la cadena ingresada.
- Variables auxiliares minimo para buscar el mínimo OPT[j] con i < j, y los índices i, j en el for anidado.

3. **Seguimiento:**

Procederemos a realizar el seguimiento del algoritmo para un set de datos reducido: **[“aba”, “abba”]**

Ejemplo 1: “aba”

`Longitud: n = 3`
`Inicializamos: OPT = [0, 0, 0, 0]` (índices de 0 a 3)

- `i = 1`:
    - `j = 0` → "a" es palíndromo → `minimo = min(∞, OPT[0] + 1) = 1`
      → `OPT[1] = 1`
- `i = 2`:

- j = 0 → "ab" no
- j = 1 → "b" sí → minimo = min(∞, OPT[1] + 1) = 2 → OPT[2] = 2

- i = 3:
  - j = 0 → "aba" sí → minimo = min(∞, OPT[0] + 1) = 1
  - j = 1 → "ba" no
  - j = 2 → "a" sí → minimo = min(1, OPT[2] + 1 = 3) → OPT[3] = 1

Resultado final: OPT = [0, 1, 2, 1] → obtener_minimo_palindromos("aba") = 1

Ejemplo 2: “abba”

Longitud: n = 4
Inicializamos: OPT = [0, 0, 0, 0, 0]

- i = 1:
  - j = 0 → "a" sí → OPT[1] = 1
- i = 2:
  - j = 0 → "ab" no
  - j = 1 → "b" sí → OPT[2] = OPT[1] + 1 = 2
- i = 3:
  - j = 0 → "abb" no
  - j = 1 → "bb" sí → OPT[3] = OPT[1] + 1 = 2
  - j = 2 → "b" sí → OPT[3] = min(2, OPT[2] + 1 = 3) → OPT[3] = 2

- `i = 4:`
    - `j = 0 → "abba" sí → OPT[4] = OPT[0] + 1 = 1`
    - `j = 1 → "bba" no`
    - `j = 2 → "ba" no`
    - `j = 3 → "a" sí → OPT[4] = min(1, OPT[3] + 1 = 3)`
      `→ OPT[4] = 1`

```
Resultado final: OPT = [0, 1, 2, 2, 1] →
obtener_minimo_palindromos("abba") = 1
```

4. **Complejidad:**

La complejidad temporal del algoritmo es **O(n³)**. Esto se debe a que:

- Se recorre la cadena con un bucle principal desde 1 hasta n → **O(n)**
- Por cada posición i, se analiza cada posible corte j < i → **O(n)**
- Para cada par (j, i), se verifica si cadena[j:i] es un palíndromo.
  Esta verificación se hace comparando la subcadena con su reversa, lo cual cuesta O(k), donde k es la longitud de la subcadena (i - j). En el peor caso, k ≈ n.

Por lo tanto, el coste total en el peor caso es:

O(n) × O(n) × O(n) = O(n³)

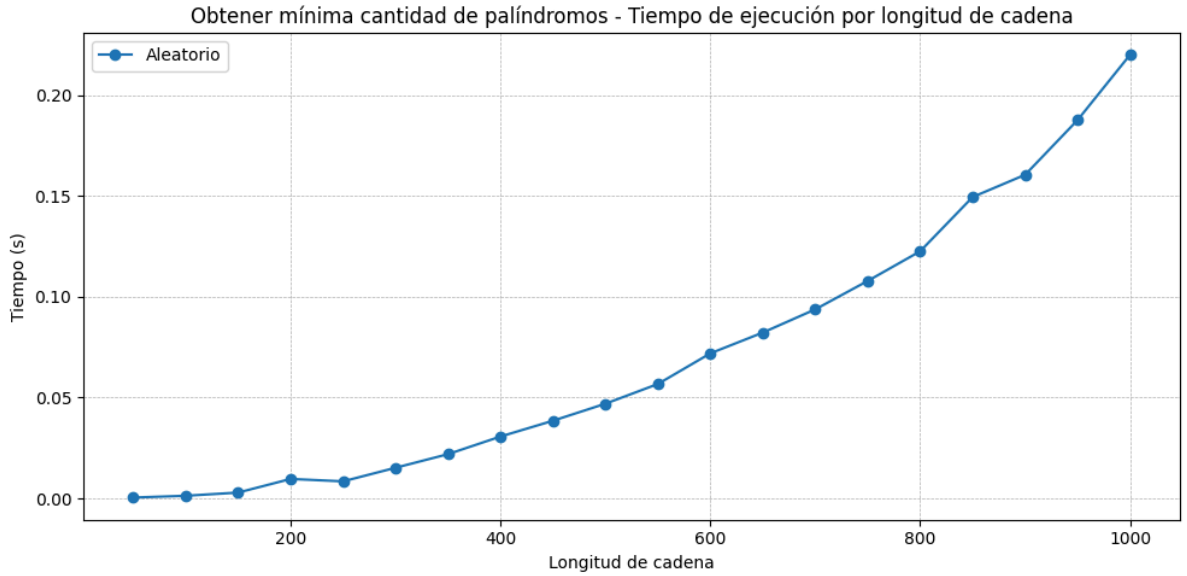
5. **Sets de datos:**

Se diseñaron set de datos de cadenas generadas aleatoriamente con una semilla fija, que van desde una longitud de cincuenta hasta mil caracteres, con un paso de cincuenta entre cada ejemplo.

Los sets de datos se generan con un programa, y se guardan en una carpeta llamada “casos_palindromos”, también se tiene otro archivo que se encarga de ejecutar el algoritmo y medir los tiempos, como también guardar los resultados. Finalmente se guarda cada caso dentro de la carpeta mencionada, y adentro de esta hay una carpeta que se llama “resultados” que contiene los resultados a cada set de datos. Por último, se incluye también un archivo csv que tiene los tiempos de ejecución de cada caso.

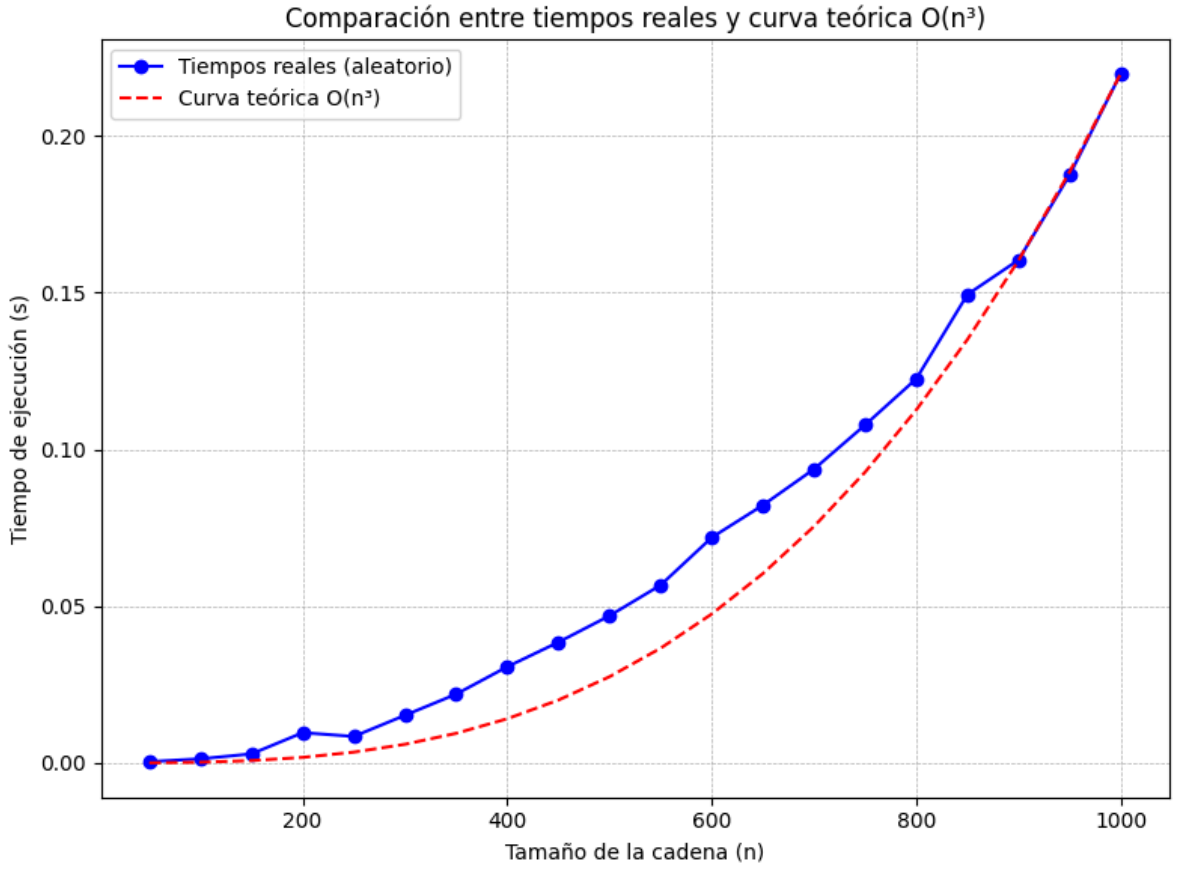
6. Tiempos de ejecución:

Los tiempos de ejecución para cada set de datos, se graficaron utilizando el módulo `matplotlib` de python.



7. Informe de resultados:

Se graficó los resultados de los tiempos de ejecución junto con la curva teórica para ver si se parece lo teórico a lo práctico:



Para analizar el comportamiento del algoritmo, medimos los tiempos de ejecución con cadenas de distintos tamaños y graficamos los resultados. A los datos reales les agregamos una curva de ajuste teórica de complejidad $O(n^3)$, que construimos usando NumPy. Para poder comparar bien ambas curvas, normalizamos la teórica para que esté en la misma escala que los tiempos reales (multiplicando por una constante de ajuste).

En la gráfica se puede ver que los tiempos reales (curva azul con puntos) tienen una tendencia creciente similar a la curva teórica (línea roja discontinua). Aunque hay pequeñas diferencias en algunos valores intermedios, sobre todo por la variabilidad en los datos aleatorios, en general se ajusta bastante bien al comportamiento cúbico esperado.

Esto confirma que, al menos empíricamente, el algoritmo tiene un tiempo de ejecución que crece con complejidad cúbica, lo cual coincide con el análisis teórico.

2. Problema de Programación Lineal

Enunciado del problema:

Concesiones Argentina 2000 SRL tiene la concesión de los espacios publicitarios en las paradas de colectivos de un municipio. Son en total 200 paradas y tiene ofertas de distintos productos para el próximo mes:

- Cliente A ofrece USD 50000 por ocupar 30 paradas
- Cliente B ofrece USD 100000 por ocupar 80 paradas o USD 120000 por 120 paradas (sólo una de ambas opciones)
- Cliente C ofrece USD 100000 por ocupar 75 paradas
- Cliente D ofrece USD 80000 por ocupar 50 paradas
- Cliente E ofrece USD 5000 por ocupar 2 paradas
- Cliente F ofrece USD 40000 por ocupar 20 paradas
- Cliente G ofrece USD 90000 por ocupar 100 paradas

Por ser competidores directos, no se puede hacer publicidad simultáneamente de los clientes A y D. Se desea determinar a qué clientes se concesionará la publicidad de las paradas para maximizar el beneficio total.

1. **Supuestos**:

- Se asume que todas las decisiones son binarias -*(las ofertas solo pueden ser aceptadas completamente o rechazadas en su totalidad)*-
    - *No es posible aceptar parcialmente ofertas*
- Se asume que cualquier parada puede ser asignada cualquier cliente
- El modelo está proyectado para solo el próximo mes
- Todas las ecuaciones de solución *(e incluso el beneficio obtenido)* van a ser **lineales**
- Se asume que no hay otras restricciones de exclusividad como sucede con los clientes A y D

2. Variables:

- **A**: binaria/indicativa
  - Indica si se acepta la oferta del cliente A para ocupar 30 paradas por USD 50000
- **B1 y B2**: binarias/indicativas
  - Indican si se acepta la oferta del cliente B para ocupar 80 paradas por USD 100000 o si se acepta la oferta del cliente B para ocupar 120 paradas por 120000 respectivamente
- **C**: binaria/indicativa
  - Indica si se acepta la oferta del cliente C para ocupar 75 paradas por USD 100000
- **D**: binaria/indicativa
  - Indica si se acepta la oferta del cliente D para ocupar 50 paradas por USD 80000
- **E**: binaria/indicativa
  - Indica si se acepta la oferta del cliente E para ocupar 2 paradas por USD 5000
- **F**: binaria/indicativa
  - Indica si se acepta la oferta del cliente F para ocupar 20 paradas por USD 40000
- **G**: binaria/indicativa
  - Indica si se acepta la oferta del cliente G para ocupar 100 paradas por USD 90000

3. Modelo de Programación Lineal:

- **Función objetivo:**
  - `Z(MAX) = 50000A + 100000B1 + 120000B2 + 100000C + 80000D + 5000E + 40000F + 90000G`

- **Restricciones:**
  - `30A + 80B1 + 120B2 + 75C +50D + 2E + 20F + 100G <= 200`
    - Limita que no se supere el límite de 200 paradas
  - `B1 + B2 <= 1`

        - Limita que solo se pueda aceptar ***una oferta*** del cliente B
    - `A + D <= 1`
        - La restricción que se pide en la consigna; no se puede hacer publicidad *simultáneamente* de los clientes A y D

4. Solución con Pulp en Python:

Código:

```
import pulp


prob = pulp.Problem("Publicidad_Paradas", pulp.LpMaximize)


# Variables binarias
A = pulp.LpVariable("A", cat='Binary')
B1 = pulp.LpVariable("B1", cat='Binary')
B2 = pulp.LpVariable("B2", cat='Binary')
C = pulp.LpVariable("C", cat='Binary')
D = pulp.LpVariable("D", cat='Binary')
E = pulp.LpVariable("E", cat='Binary')
F = pulp.LpVariable("F", cat='Binary')
G = pulp.LpVariable("G", cat='Binary')


prob += 50000*A + 100000*B1 + 120000*B2 + 100000*C + 80000*D +
5000*E + 40000*F + 90000*G


prob += 30*A + 80*B1 + 120*B2 + 75*C + 50*D + 2*E + 20*F + 100*G <=
200
prob += B1 + B2 <= 1
prob += A + D <= 1



prob.solve()
```

```
resultado = f"Beneficio máximo:
                ${pulp.value(prob.objective):,.0f}\n"

resultado += f"A: {pulp.value(A)}, B1: {pulp.value(B1)}, B2:
                {pulp.value(B2)}, C: {pulp.value(C)}\n"

resultado += f"D: {pulp.value(D)}, E: {pulp.value(E)}, F:
                {pulp.value(F)}, G: {pulp.value(G)}\n"

with open("resultado_paradas.txt", "w") as archivo:
     archivo.write(resultado)

print(resultado)
print("Resultado guardado en: resultado_paradas.txt")
```

Documentación:

- Descargar **Python3** desde https://www.python.org/
- Descargar **Pulp**: *pip install pulp*
- **Ejecutar**:
    - Desde una consola ejecutar "*python3 publicidad_paradas.py*"
    - El resultado será mostrado mostrado por consola y escrito en un archivo llamado "***Publicidad_paradas.txt***"
    - Ese archivo se guardará en el mismo directorio desde donde se ejecutó el script Python
        - *Si se tiene algún IDE se puede ejecutar según ese IDE*

- Luego de ejecutarse el código, el código te dirá el beneficio máximo y los valores las variables de decisión (1.0 para aceptar el cliente, 0.0 para rechazarlo)

5. **Informe**:

La solución hallada nos indica qué clientes se deben aceptar y qué clientes rechazar, siempre teniendo en cuenta que la solución va a respetar las 200

paradas posibles y la ya mencionada restricción de **incompatibilidad** entre los clientes A y D.
Esta solución prioriza a los clientes con mejor relación beneficio/paradas para llegar a la máxima ganancia obtenible dentro del dominio del problema.

3. Problema Ford Fulkerson:

Enunciado del problema:

Estamos construyendo una red **WAN** con n antenas y queremos que tenga un buen nivel de tolerancia a fallas. Dada una antena, su conjunto backup de tamaño k es el conjunto de k antenas que se encuentran a una distancia menor a D.
Queremos evitar que una antena pertenezca al conjunto de backup de más de b antenas, precisamente para evitar que un fallo pueda afectar a una porción importante de la red. Suponer que conocemos los valores D, b y k, y que tenemos una matriz **d[1..n, 1..n]** con las distancias entre antenas, de forma tal que d[i,j] es la distancia entre la antena i y la j.

Plantear un algoritmo de complejidad polinomial que encuentre el conjunto de backup de tamaño k de cada una de las n antenas, de forma tal que ninguna aparezca en más de b conjuntos de backup, o bien, que indique que no existe una solución posible. Debe apoyarse en el algoritmo estándar de Ford-Fulkerson, la variante escalada o la de Edmonds-Karp.

1. **Supuestos:**

Se supone que:

1. Número fijo de antenas = n.
2. El conjunto de backup de tamaño k es el conjunto de antenas k que se encuentran a una distancia < D.
3. Solo se puede seleccionar como backup *una antena* j para i si d[i][j] < D.
4. Tenemos la matriz **d[1..n, 1..n]** con las distancias entre antenas.

5. d[i,j] es la distancia entre la antena i y la j.
6. Se conocen los valores de D, b y k.

2. **Diseño:**

a. Explicación en prosa de cómo se adaptan los datos de entrada

Grafo de flujo dirigido:

S —> Fuente
T —> Sumidero
Ai —> Nodos de antenas de origen
Bj —> Nodos de antenas para ser candidatas a backups

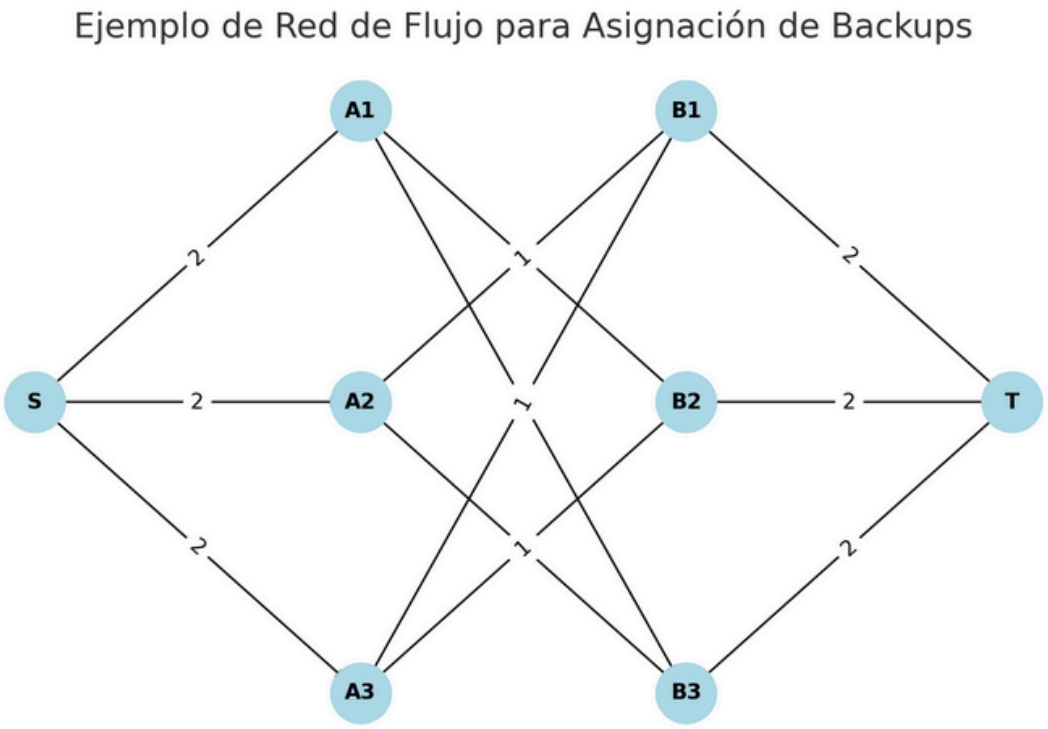
Aristas:

- Desde S a cada nodo Ai con capacidad k, teniendo en cuenta que cada antena requiere k backups
- Desde cada Ai a cada Bj tal que d[i][j] < D y i ≠ j, con capacidad 1.
- Desde cada nodo Bj a T con capacidad b, teniendo en cuenta que cada antena puede ser backup de a lo sumo b otras).

El objetivo es encontrar un flujo máximo = n * k
Si el flujo máximo alcanza ese valor, quiere decir que se puede construir una solución válida, si no, no existe una solución que respete las restricciones del grafo.

- **Ejemplo gráfico**:

```
      S
     /|\
    / | \
   A1 A2 ... An     (cada uno con capacidad k desde S)
   |   |     |
  /    |     \
B1    B2 ... Bn     (con capacidad b hacia T)
   \   |    /
       T
```



b. Pseudocódigo y estructuras de datos:

Pseudocodigo de Ford-Fulkerson:

```
Flujo-Max(G, s, t):
     Inicializar f(u, v) = 0 para todas las aristas (u, v) en G

     Mientras exista un camino s-t P en el grafo residual Gf:
          c = capacidad mínima residual en el camino P
          Para cada arista (u, v) en P:
                    f(u, v) += c
                    f(v, u) -= c  // Flujo residual en la dirección
inversa
          Actualizar Gf según los nuevos valores de flujo

     Devolver f
```

Pseudocodigo de resolución del problema:

```
  Algoritmo Backup-Antenas(d[1..n,1..n], D, k, b, n):

       Crear grafo dirigido G = (V, E)
       Agregar nodos:
            Fuente S
            Sumidero T
            for antena in range(1, n):
                 Nodo requisitor Ai
                 Nodo servidor Bi

       for i in range(1, n):
            Agregar arista (S → Ai) con capacidad k
            Agregar arista (Bi → T) con capacidad b

       for par i, j con i ≠ j:
            if d[i][j] < D:
                 Agregar arista (Ai → Bj) con capacidad 1

       // Ejecutar Ford-Fulkerson sobre este grafo
```

```
      f = Flujo-Max(G, S, T)

      // Verificar si se logró el flujo total requerido
      Si flujo total en f desde S < n × k:
            Devolver "No existe solución posible"
      Si no:
            for i in range(1, n):
                  Inicializar backup[i] = []
                  Para cada j tal que (Ai → Bj) tiene flujo 1 en
f:
                        Agregar j al conjunto backup[i]

      Devolver los backup[i] para cada antena i
```

3. **Seguimiento:**

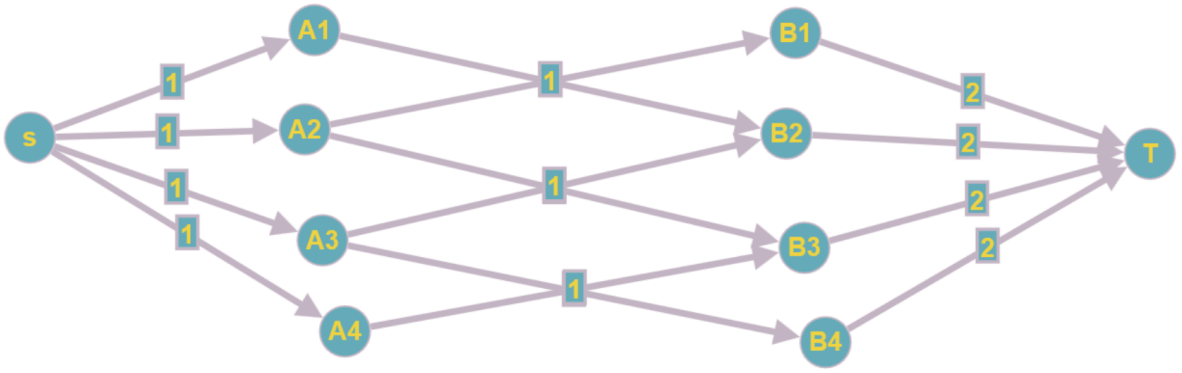
Vamos a hacer el seguimiento con matriz1.txt
n=4, D=3, k=1, b=2

- S —> An, cada antena con capacidad k=1
- Ai —> Bj solo si i != j y d[i][j] < 3
- B1 a B4 —> T, cada una con capacidad b=2

Matriz de distancias (d[i][j]):

	A1	A2	A3	A4
A1	0	2	3	4
A2	2	0	1	5
A3	3	1	0	2
A4	4	5	2	0

Dibujo del grafo:



Paso 1: Construcción del grafo de flujo

Nodos:

- Fuente S
- 4 nodos Ai = A1, A2, A3, A4 (cada antena necesita backups)
- 4 nodos Bi = B1, B2, B3, B4 (cada antena puede ser backup)
- Sumidero T

Aristas:

- De S a cada Ai con capacidad k = 1
- De cada Bi a T con capacidad b = 2
- De Ai a Bj si $i \neq j$ y $d[i][j] < D$

Aristas válidas entre Ai → Bj:

- A1 → B2 (d=2)
- A2 → B1 (d=2), B3 (d=1)
- A3 → B2 (d=1), B4 (d=2)
- A4 → B3 (d=2)

Paso 2: Aplicar Ford-Fulkerson (con BFS)

Buscamos caminos aumentantes desde S a T, mientras haya capacidad disponible en las aristas:

1. Camino 1:
   S → A1 → B2 → T
   Capacidad mínima: 1

Aumentamos flujo en esas aristas.

2. Camino 2:
   S → A2 → B1 → T
   Capacidad mínima: 1
   Flujo actualizado.

3. Camino 3:
   S → A3 → B4 → T
   Capacidad mínima: 1
   Flujo actualizado.

4. Camino 4:
   S → A4 → B3 → T
   Capacidad mínima: 1
   Flujo actualizado.

Paso 3: Verificar flujo total

Se necesitan $n \times k = 4$ unidades de flujo (una por cada Ai).
Flujo total alcanzado: 4

Paso 4: Leer solución

Backup asignado a cada antena según flujo de Ai → Bj:

- A1 usa B2 → Backup de A1: [2]
- A2 usa B1 → Backup de A2: [1]
- A3 usa B4 → Backup de A3: [4]
- A4 usa B3 → Backup de A4: [3]

el número entre [corchetes] es el numero i del backup Bi.

Puede haber variantes, como que A2 utilice B3 también y A3 use B2. Esto pasa porque el algoritmo de Ford-Fulkerson no es determinista. Puede devolver diferentes soluciones válidas dependiendo del orden en que se exploran los caminos

4. **Complejidad:**

Le damos dirección al flujo y ponemos las cargas:
Se construye un grafo con:
2n + 2 nodos:

- 1 fuente S
- n nodos Ai
- n nodos Bi
- 1 sumidero T

Aristas:

- n aristas de la fuente a cada Ai: O(n)
- n aristas de cada Bi al sumidero: O(n)
- Hasta n(n-1) aristas entre Ai y Bj, si cumplen d[i][j] < D: $O(n^2)$ as

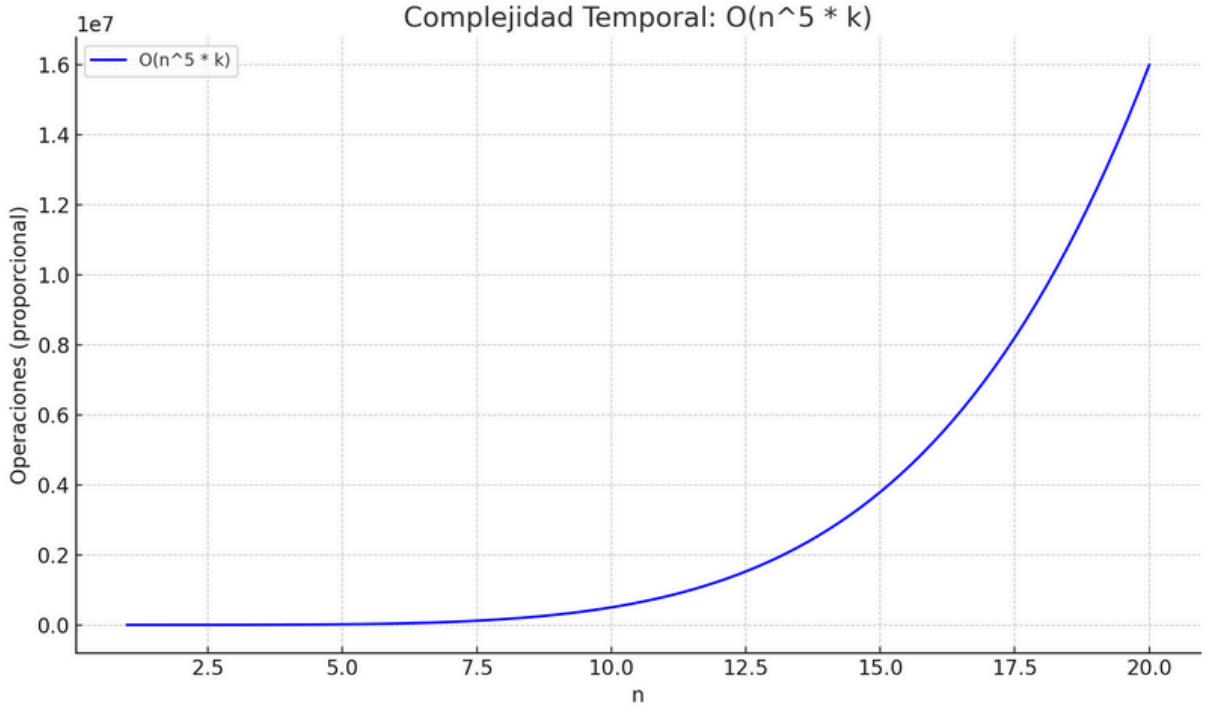
Implementación del BFS utilizando Ford-Fulkerson:
Complejidad por ejecución: $O(V^2)$, ya que cada nodo revisa todos los demás nodos en la matriz.

En el peor caso, el número de iteraciones es O(E * F) donde:

E es el número de aristas →
F es el valor del flujo máximo → En este problema a lo sumo n*k
Entonces el número de iteraciones utilizando Ford-Fulkerson es $O(n^2 \times n \times k) = O(n^3k)$

Cada iteración toma $O(n^2)$, entonces el total es $\mathbf{O(n^2) \times O(n^3k) = O(n^5k)}$
Podemos afirmar que es polinomial, ya que el número de iteraciones está acotado por el flujo máximo total.

Grafico de complejidad:



5. **Sets de datos:**

Para el set de datos, el código utiliza diferentes matrices donde siempre el primer renglón tiene los parámetros n, D, k, b y luego la matriz de distancia del la red de flujo. Para esto se presentaron 8 matrices cada una con estos valores:

- Matriz 1: n=4, D=3, k=1, b=2

ej: 4 3 1 2
0 2 3 4
2 0 1 5
3 1 0 2
4 5 2 0

- Matriz 2: n=12 , D=5 , k=2, b=3
- Matriz 3: n=30 , D=12 , k=3, b=5
- Matriz 4: n=100 , D=80 , k=20, b=30
- Matriz 5: n=150 , D=100 , k=40, b=60
- Matriz 6: n=300 , D=140 , k=50, b=70
- Matriz 7: n=360 , D=174 , k=66, b=88

- Matriz 8: n=400 , D=198 , k=74, b=90

Estas mismas matrices se pueden encontrar en el codigo dentro de la carpeta **casos_red_de_flujo**

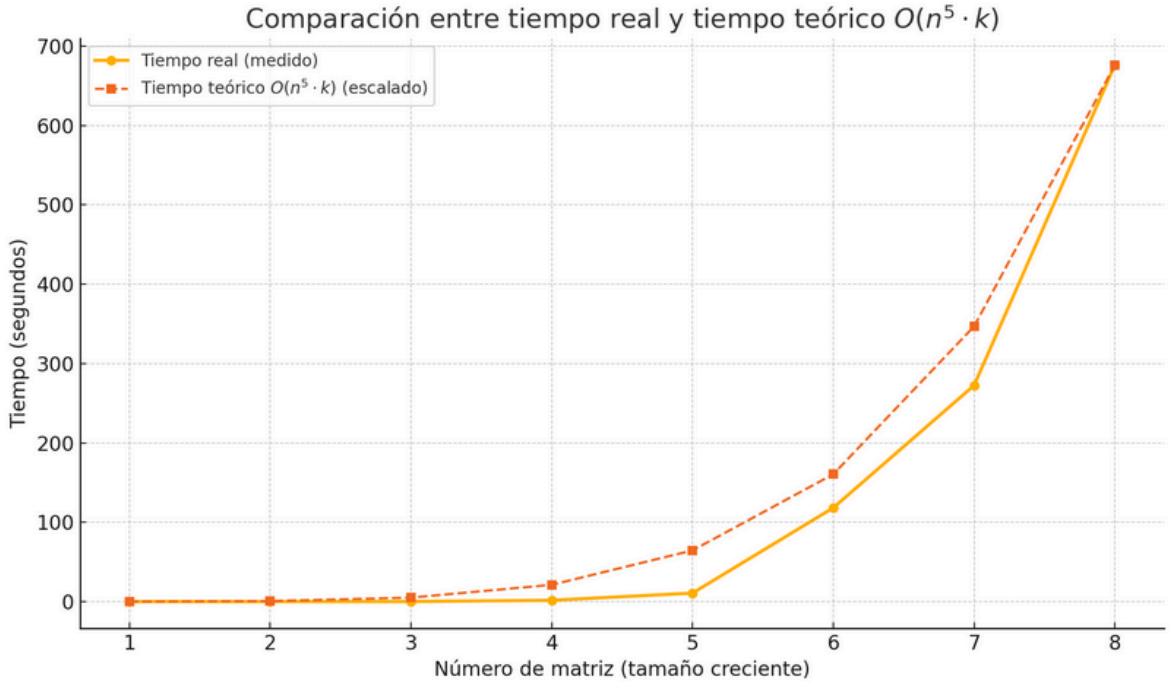
Observando los tiempos de ejecución medidos, se nota que a medida que aumenta n (la cantidad de antenas) y k (la cantidad de backups), el tiempo crece bastante rápido. Por ejemplo, al pasar de n = 100 y k = 20 (1.67 segundos) a n = 150 y k = 40, el tiempo salta a más de 10 segundos. Y con n = 400, el tiempo pasa a más de 10 minutos.

Este crecimiento fuerte en los tiempos se alinea con la complejidad teórica del algoritmo, que es $O(n^5k)$.

6. Tiempos de ejecución:

Matriz 1: 0,001 segundos
Matriz 2: 0,003 segundos
Matriz 3: 0,017 segundos
Matriz 4: 1,67 segundos
Matriz 5: 10,68 segundos
Matriz 6: 118,18 segundos
Matriz 7: 272,4 segundos
Matriz 8: 676,13 segundos

Comparación entre tiempo teórico **$O(n^5k)$** y el tiempo real:



7. Informe de resultados:

Para evaluar cómo se comporta nuestro algoritmo, lo probamos con 8 casos de prueba distintos. En cada caso cambiamos los valores de n, D, k y b, y usamos una matriz de distancias entre antenas que respeta las condiciones del problema como, por ejemplo, que no se puede usar de backup a una antena que está muy lejos.

Vimos, que para casos chicos el algoritmo es muy rápido, además, con n = 100, tarda menos de 2 segundos. Pero a partir de ahí, el tiempo empieza a crecer bastante rápido, llegando a más de 10 minutos para el caso más grande. Esto tiene sentido, porque la complejidad teórica es $O(n^5k)$, y es de carácter polinomial, así que no nos sorprendió que aumente mucho el tiempo cuando n y k son grandes.

El algoritmo fue capaz de determinar correctamente si existía una solución viable y retornar los conjuntos de antenas backup, validando su correcto funcionamiento lógico y estructural.

.