P3 – Grafos. Programación Dinámica

Juan José Sánchez Peña

Requisitos de entrega (1/2)

- Crear una carpeta con el nombre p3**NN**(1) e incluir los siguientes ficheros:
  - p3**NN**.py: incluirá el código de las funciones implementadas en la práctica y los `imports` estrictamente necesarios.
  - p3**NN**_optional.py: contendrá el código implementado para la parte opcional III.
  - p3**NN**.pdf: memoria que contenga las respuestas a las cuestiones de la práctica.
  - Los ficheros .py auxiliares que hayáis creado para el correcto funcionamiento de toda la práctica.

(1) **NN** indica el número de pareja.

Requisitos de entrega (2/2)

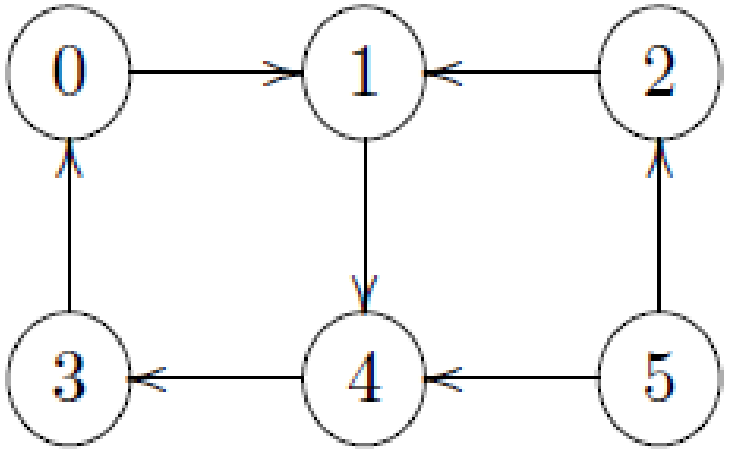
- Observaciones
  - Los nombres y parámetros de las funciones definidas en p3**NN**.py deben ajustarse EXACTAMENTE a lo definido en el enunciado.
  - En la memoria se identificará claramente el nombre de los estudiantes y el número de pareja. Si se añaden figuras o gráficos, deben realizarse sobre fondo blanco.
  - Para la entrega, comprimir la carpeta en un fichero llamado p3**NN**.zip. No añadir ninguna estructura de subdirectorios a dicha carpeta.
  - **La práctica no se corregirá hasta que el envío no siga esta estructura**.

Corrección

- Ejecución del script que importará p3**NN**.py, que comprobará la corrección de dicho código. **¡¡IMPORTANTE!! La práctica no se corregirá mientras este script no se ejecute correctamente, penalizando las segundas entregas**.
- Revisión de ciertas funciones implementadas en p3**NN**.py.
- Revisión de la memoria con las respuestas a las cuestiones planteadas.

TAD: grafo dirigido no ponderado (1/6)

- Grafo dirigido: conjunto de vértices (V) y ramas (E) que conectan dichos vértices en un sentido.
- Grafo dirigido no ponderado: las ramas no tienen peso.



TAD: grafo dirigido no ponderado (2/6)

```
def __init__(self):
    self._V = dict()                # Dictionary with G nodes: Dict[str, Dict[str]]
    self._E = dict()                # Dictionary with G edges: Dict[str, Set]
```

- `self._V`: es un diccionario donde se almacenan los nodos. La clave es un nodo y el valor es otro diccionario con los atributos de los nodos que se utilizarán en los diferentes algoritmos que implementaremos en la práctica.
- `self._E`: es un diccionario que implementa las listas de adyacencia de los nodos. La clave es un nodo y el valor es un conjunto de nodos a los que está conectado.

TAD: grafo dirigido no ponderado (3/6)

```
def add_node(self, vertex) -> None:
```

- Inicializar _V: llamar a la función `_init_node(self, vertex)`.
- Inicializar _E: inicializar la lista de adyacencia.

```
def add_edge(self, vertex_from, vertex_to) -> None:
```

- Añade un arco entre `vertex_from` a `vertex_to`. Los nodos se añaden si no existen en el grafo.
- Añadir nodo `vertex_from`.
- Añadir nodo `vertex_to`.
- Añadir a la lista de adyacencia.

TAD: grafo dirigido no ponderado (4/6)

```
def nodes(self) -> KeysView[str]:
```

- `KeysView`: tipo de vista que proporciona acceso a las claves de un diccionario de manera dinámica.

```
def adj(self, vertex) -> Set[str]:
```

- Devuelve los nodos adyacentes al nodo `vertex`.

```
def exists_edge(self, vertex_from, vertex_to)-> bool:
```

- Devuelve `True`/`False` si el nodo `vertex_to` se encuentra en la lista de adyacencia de `vertex_from`.

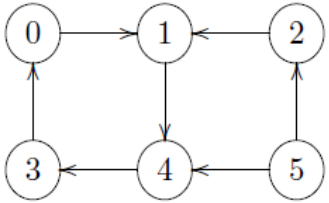
TAD: grafo dirigido no ponderado (5/6)

```
def __str__(self) -> str:
```

- Imprime por pantalla las diferentes salidas.

```
Vertices:
0: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
1: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
2: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
4: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
3: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}
5: {'color': 'WHITE', 'parent': None, 'd_time': None, 'f_time': None}

Aristas:
0: {1}
1: {4}
2: {1}
4: {3}
3: {0}
5: {2, 4}
```

TAD: grafo dirigido no ponderado (6/6)



- Ejemplo:

```
print('Nodes:')
print(G.nodes())
```

```
Nodes:
dict_keys([0, 1, 2, 4, 3, 5])
```

```
print('Adyacentes nodo 5')
print(G.adj(5))
```

```
Adyacentes nodo 5
{2, 4}
```

```
print('¿Es 2 adyacente de 5?')
print(G.exists_edge(5,2))
print('¿Es 3 adyacente de 5?')
print(G.exists_edge(5,3))
```

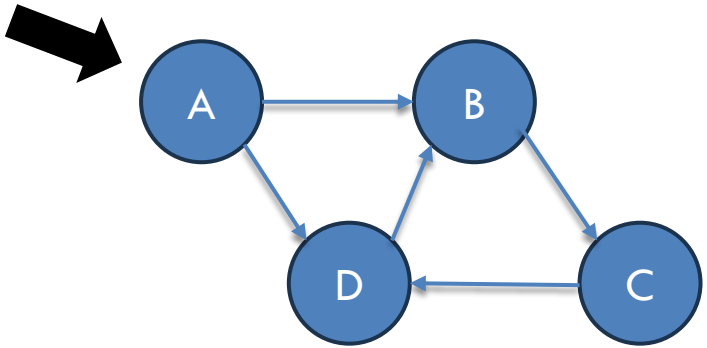
```
¿Es 2 adyacente de 5?
True
¿Es 3 adyacente de 5?
False
```

Recorrido en grafos y componentes fuertemente conexas (1/11)

- DFS (***D**epth **F**irst **S**earch*) (1/5)
  - Es un algoritmo de búsqueda para lo cual recorre los nodos de un grafo.
  - Su funcionamiento consiste en ir expandiendo cada uno de los nodos que va localizando, de forma recurrente (desde el nodo padre al nodo hijo).
  - Cuando ya no quedan más nodos que visitar en dicho camino, regresa al nodo predecesor, de modo que repite el mismo proceso con cada uno de sus vecinos.
  - Indicar que si se encuentra el nodo a buscar antes de recorrer todos los nodos, termina la búsqueda.

Recorrido en grafos y componentes fuertemente conexas (2/11)

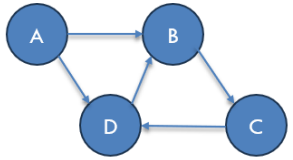
- DFS (***D**epth **F**irst **S**earch*) (2/5)
  - Ejemplo:

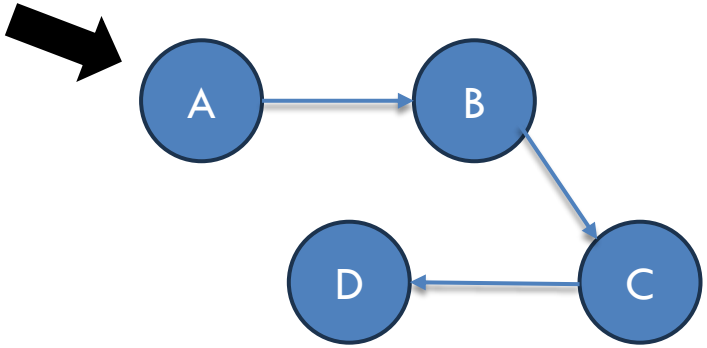


- Elegimos el vértice A.

Recorrido en grafos y componentes fuertemente conexas (3/11)

- DFS (***D**epth **F**irst **S**earch*) (3/5)
  - Ejemplo:

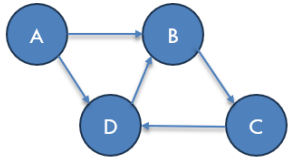


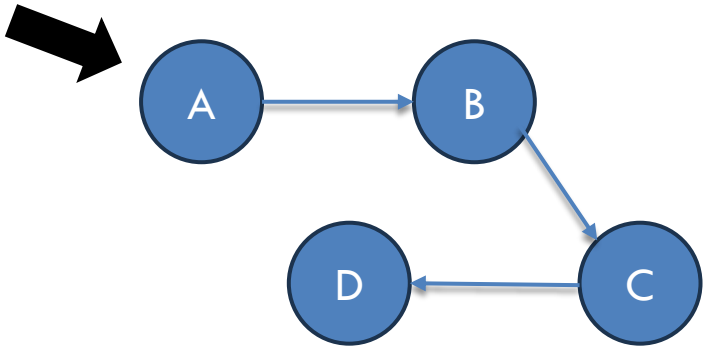


- Los vecinos de A: son B y D. Si no se indica ningún criterio, podríamos elegir B o D. Elegimos B.
- El vecino de B es C.
- El vecino de C es D.

Recorrido en grafos y componentes fuertemente conexas (4/11)

- DFS (***D**epth **F**irst **S**earch*) (4/5)
  - Ejemplo:





SOLUCIÓN: A → B → C → D

- El vecino de D es B, pero ya ha sido visitado. Retrocedemos.
- El vecino de C es D, que ya ha sido visitado. Retrocedemos.
- El vecino de B es C, que ya ha sido visitado. Retrocedemos.
- El otro vecino de A es D, que ya ha sido visitado. FIN.

Recorrido en grafos y componentes fuertemente conexas (5/11)

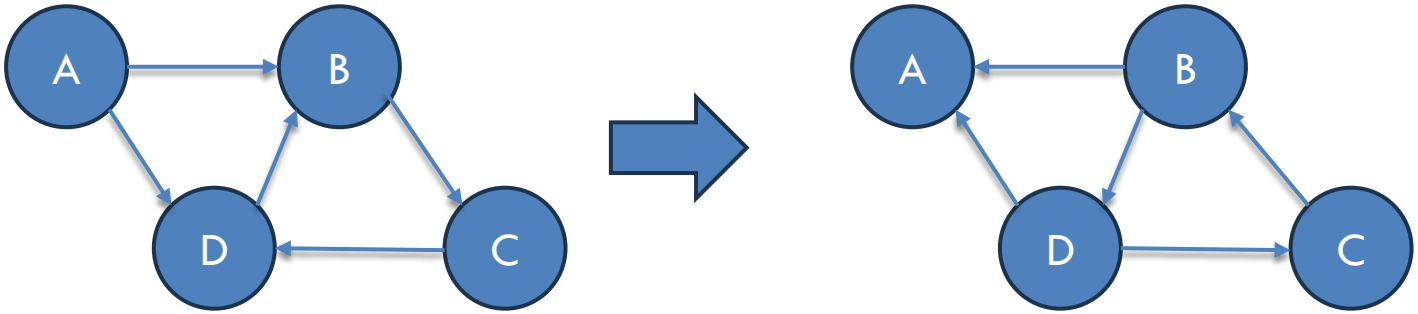
- DFS (***D**epth **F**irst **S**earch*) (5/5)

```
def dfs(self, nodes_sorted: Iterable[str] = None) -> List[List[Tuple]]:
```

  - Si se le pasa un iterable, el bucle principal de DFS se iterará según el orden del iterable.
  - Devuelve un bosque DFS en la que cada una de las sublistas es un árbol DFS. Cada elemento del árbol es una tupla (nodo – vertex, parent).

Recorrido en grafos y componentes fuertemente conexas (6/11)

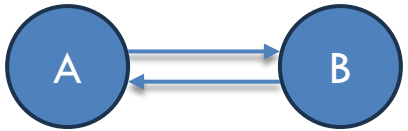
- Grafo conjugado
  - Es el grafo inverso al que original.
  - Ejemplo:




```
def graph_conjugate(G: Graph) -> Graph:
```

Recorrido en grafos y componentes fuertemente conexas (7/11)

- Algoritmo de Tarjan (1/5)
  - Permite encontrar componentes fuertemente conexas en un grafo.
  - Un grafo dirigido es fuertemente conexo si para cada par de nodos podemos encontrar un camino de ida y otro de vuelta.
  - Una componente fuertemente conexa es el subgrafo más grande fuertemente conexo.



Recorrido en grafos y componentes fuertemente conexas (8/11)

- Algoritmo de Tarjan (2/5)
  - El algoritmo toma el grafo inicial y devuelve las particiones fuertemente conexas.
  - Cada nodo solo pertenece a una componente.
  - Puede haber componentes que solo tenga un nodo.
  - La idea es realizar búsquedas en profundidad primero sobre el primer nodo a escoger y luego sobre el resto de nodos que no se han encontrado en las búsquedas anteriores.

Recorrido en grafos y componentes fuertemente conexas (9/11)

- Algoritmo de Tarjan (3/5)
  - Una búsqueda en profundidad, elige un nodo raíz y va haciendo un camino siguiendo todas las aristas, sin preferencia por dónde pasar.
  - Estos nodos quedan añadidos a una lista hasta que finalmente ya no puede avanzar más, porque ya no hay más nodos adyacentes (vecinos) o ha llegado a la raíz.
  - Si llega al nodo raíz, entonces tiene una componente fuertemente conexa.
  - Si no, el algoritmo va mirando desde el último nodo de la lista al primero para poder desviarse en algún punto y tomar otro camino.
  - Peor caso: complejidad O(V+A), es decir se ejecuta en tiempo lineal.

Recorrido en grafos y componentes fuertemente conexas (10/11)

- Algoritmo de Tarjan (4/5)

```
def tarjan(self)-> List[List[str]]:
```

  - **DFS en el grafo original:** realizar una primera búsqueda en profundidad (DFS) en el grafo original para determinar el **orden de finalización** de los nodos (en qué orden los visitamos y terminamos). Esto nos permite organizar los nodos en un orden tal que cualquier nodo que alcance a otro en su componente fuertemente conexa será alcanzado en este orden inverso en el grafo transpuesto.
  - **DFS en el grafo transpuesto:** crear el grafo transpuesto (con las direcciones de las aristas invertidas). Este grafo transpuesto permite explorar las componentes fuertemente conexas desde los nodos de "salida" hacia los "interiores" de cada componente.

Recorrido en grafos y componentes fuertemente conexas (11/11)

- Algoritmo de Tarjan (5/5)

```
def tarjan(self)-> List[List[str]]:
```

(Cont.)

  - **DFS en el grafo transpuesto:** usar el orden inverso de finalización de la primera DFS para recorrer el grafo transpuesto y, a partir de cada nodo aún no visitado, ejecutar otra DFS. Cada DFS en este paso descubre exactamente una componente fuertemente conexa.

Componente gigante y umbral de percolación (1/8)

- Grafo de Erdös-Renyi (1/2)
  - Como ya vimos, es un grafo aleatorio con ***n*** nodos.
  - Cada nodo tiene de media ***m*** vecinos (***n*** y ***m*** son los parámetros que definen el grafo).
  - La probabilidad de que dos nodos cualesquiera estén conectados es p=m/n.
  - ¿Por qué es p=m/n?
    - En un grafo aleatorio, las conexiones entre nodos se generan de manera independiente con la misma probabilidad ***p***.
    - Si cada nodo tiene ***m*** vecinos en promedio, y como hay ***n*** nodos, para garantizar este promedio, debemos establecer ***p*** de forma proporcional al número total de nodos.

Componente gigante y umbral de percolación (2/8)

- Grafo de Erdös-Renyi (2/2)
  - Ejemplo:
    - Supongamos un grafo de Erdös-Renyi con:
      - n=100 nodos.
      - Cada nodo tiene m=10 vecinos en promedio.

    Entonces la probabilidad de conexión entre dos nodos es:

$$p = \frac{m}{n} = \frac{10}{100} = 0.1$$

    Esto significa que cada par de nodos tiene una probabilidad del 10% de estar conectado.

Componente gigante y umbral de percolación (3/8)

- Umbral de percolación (1/3)
  - Valor crítico que marca un cambio significativo en la estructura del grafo: componentes desconectadas vs componente gigante.
  - Cuando $m < m_c$ (antes del umbral) → $m < 1$
    - **Desconexión generalizada**: el grafo está formado principalmente por pequeñas componentes desconectadas.
      - Muchos nodos no tienen conexiones.
      - Si un nodo está conectado, lo hará con 1 ó 2 nodos como máximo, formando componentes muy pequeñas (a menudo un único nodo o pares aislados).
    - Esto sucede porque, con un promedio de menos de 1 vecino por nodo, la probabilidad de que se formen caminos largos entre nodos es extremadamente baja.

Componente gigante y umbral de percolación (4/8)

- Umbral de percolación (2/3)
  - Cuando $m > m_c$ (después del umbral) → $m > 1$
    - **Aparece una componente gigante.**
      - Los nodos comienzan a conectarse en mayor medida.
      - Se forma una componente conexa gigante, que incluye la mayoría de los nodos del grafo.
      - Este fenómeno es conocido como **condensación del grafo**: la mayoría de los nodos pasan a formar parte de esta gran componente conexa.
    - **Otras componentes son pequeñas:**
      - Aunque haya pequeños componentes desconectados, la mayoría de los nodos (prácticamente todos) pertenecen a la componente gigante.

Componente gigante y umbral de percolación (5/8)

- Umbral de percolación (3/3)
  - Cuando $m_c = 1$
    - **Transición de fase. ¿Qué es?**
      - Es un cambio abrupto en la estructura del grafo que ocurre al pasar el valor crítico m_c.
      - Similar a los cambios en los estados físicos de la materia (como agua que se congela), en los grafos se observa una transformación radical en su conectividad.
    - **Cómo ocurre en grafos:**
      - Antes del umbral ($m<1$): el grafo está fragmentado; conectividad muy baja.
      - En el umbral ($m\approx1$): los pequeños componentes comienzan a conectarse rápidamente, formando la componente gigante.
      - Después del umbral ($m>1$): la componente gigante se estabiliza y contiene la mayoría de los nodos.

Componente gigante y umbral de percolación (6/8)

- ¿Qué tenemos que hacer? (1/3)
  - Crear grafos aleatorios dirigidos todos del mismo tamaño ***n*** pero con diferentes valores de ***m*** y analizar el tamaño de la mayor de las componentes fuertemente conexas (normalizado por ***n***) frente al valor de ***m***.
  - Recomendaciones:
    - Utilizar grafos de al menos 10^3 nodos.
    - Representar un número apreciable de puntos en tus gráficas.
    - Implementar las siguientes funciones.

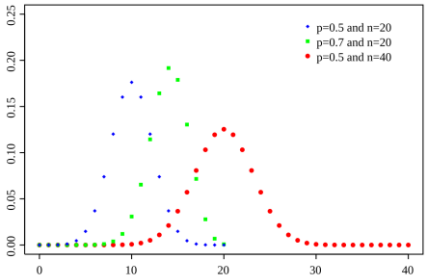
```
def erdos_renyi(n: int, m: float = 1.) -> g.Graph:

def size_max_scc(n: int, m: float) -> Tuple[float, float]:
```

Componente gigante y umbral de percolación (7/8)

- ¿Qué tenemos que hacer? (2/3)

```
def erdos_renyi(n: int, m: float = 1.) -> g.Graph:
```

  - Devuelve un grafo aleatorio dirigido.
  - ***n***: número de nodos del grafo; ***m***: número medio de vecinos de un nodo.
  - El número de vecinos de cada nodo del grafo se obtiene a partir de una muestra de una distribución binomial de probabilidad p=m/n.
    - `Vecinos = binom.rvs(n, p, size=n)  #genera números aleatorios`



Componente gigante y umbral de percolación (8/8)

- ¿Qué tenemos que hacer? (3/3)

```
def size_max_scc(n: int, m: float) -> Tuple[float, float]:
```

  - Genera un grafo dirigido aleatorio de parámetros ***n*** y ***m***.
    - Llamar a la función ***erdos_renyi***.
  - Calcula el tamaño de la mayor de las componentes fuertemente conexas.
    - Llamar a la función ***tarjan***.
    - Obtener el tamaño de la mayor scc.
  - Devuelve una tupla con el tamaño de la mayor scc del grafo normalizada entre ***n*** y el valor de ***m***.
    - `return (tamano_valor scc/n, m)`

Programación Dinámica

- Técnica que se aplica a problemas de optimización, es decir, a aquellos problemas en los cuales de un conjunto de soluciones posibles (normalmente un número grande) debe elegir aquella que maximiza o minimiza un parámetro dado.
- La solución de problemas mediante esta técnica se basa en el llamado principio de óptimo enunciado por Bellman en 1957 y que dice: "*en una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima*".

Programación Dinámica – La distancia de edición (1/2)

- También conocida como distancia de Levenshtein es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra.
- Las operaciones de edición que se pueden hacer son:
  - Insertar un carácter. Ej: abc → abca
  - Eliminar un carácter. Ej: abc → ac
  - Sustituir un carácter. Ej: abc → adc

Programación Dinámica – La distancia de edición (2/2)

- Ejemplo: calcular la distancia de edición entre casa y calle.
  - "casa" → "cala" (sustitución de "s" por "l")
  - "cala" → "calla" (inserción de "l" entre "l" y "a")
  - "calla" → "calle" (sustitución de "a" por "e")
- Por tanto, la distancia es de 3.

```
def edit_distance(str1: str_1, str_2: str) -> int:
```

  - Función que devuelve la distancia de edición entre las cadenas ***str_1*** y ***str_2***. **OJO**: usando la menor cantidad de memoria posible.
  - Información en los apuntes en las páginas 202 y 203.

Programación Dinámica – LCS (1/2)

- La subsecuencia común más larga (LCS – ***L**ongest **C**ommon **S**ubsequence*) consiste en encontrar la subsecuencia más larga común a dos secuencias de caracteres. Una subsecuencia es una secuencia que aparece en el mismo orden en ambas secuencias, pero no necesariamente de forma consecutiva.
- Ejemplo:
  - La subsecuencia común más larga de las cadenas “ABCD” y “ACDF” es “ACD”.

Programación Dinámica – LCS (2/2)

```
def max_subsequence_length(str_1: str, str_2: str)-> int:
```

- Función que devuelve la longitud de una subsecuencia común a las cadenas ***str_1*** y ***str_2*** no necesariamente consecutiva. Dicha función deberá usar la menor cantidad de memoria posible.

```
def max_common_subsequence(str_1: str, str_2: str) -> str:
```

- Función que devuelve una subcadena común a las cadenas ***str_1*** y ***str_2*** aunque no necesariamente consecutiva.

- Información en los apuntes en las páginas 206 y 207.

Programación Dinámica – Multiplicación de matrices (1/3)

- Dada la dimensión de una secuencia de matrices en un array, donde la dimensión de la matriz i^{th} es (array[i-1]*array[i]).

- Encontrar la forma más eficiente de multiplicar dichas matrices tal que el total de multiplicaciones sea mínimo.

- Cuando dos matrices de tamaño ***m*n*** y ***n*p*** se multiplican, se genera una matriz de tamaño ***m*p*** y el número de multiplicaciones realizadas es ***m*n*p***.

Programación Dinámica – Multiplicación de matrices (2/3)

- Ejemplo:
  - array = [2, 1, 3, 4]
  - Son tres matrices de dimensiones 2x1 (M1), 1x3 (M2) y 3x4 (M3).
  - Tendríamos dos formas de multiplicar las mismas:
    - (M1xM2)xM3 y M1x(M2xM3).
    - Tenemos que tener en cuenta que la multiplicación de M1xM2 nos da una matriz de 2x3; la multiplicación de M2xM3 nos da una matriz de 1x4.
  - (M1xM2)xM3 → 2x1x3 + 2x3x4 = 6 + 24 = 30
  - M1x(M2xM3) → 1x3x4 + 2x4x1 = 12 + 8 = 20

Programación Dinámica – Multiplicación de matrices (3/3)

```
def min_mult_matrix(l_dims: List[int])-> int:
```

- Función que devuelve el mínimo número de productos para multiplicar n matrices cuyas dimensiones están contenidas en la lista ***l_dims*** con n+1 ints, el primero de los cuales nos da las filas de la primera matriz y el resto las columnas de todas ellas.

- Información en los apuntes en las páginas 209 y 212.

Apartado extra (1/2)

- El tamaño de la mayor de las componentes fuertemente conexas de un grafo dirigido es una variable aleatoria.
- Si se generasen ***n_rep*** grafos llamando ***n*** veces a la función ***erdos_renyi()***, manteniendo los parámetros de ***n*** (número de nodos del grafo) y ***m*** (número medio de vecinos de un nodo) obtendríamos valores diferentes.
- Añadir un intervalo de confianza que sea representativo de la variabilidad, volviendo a representar los datos y discutir los resultados obtenidos.

Apartado extra (2/2)

- El intervalo de confianza se utiliza para cuantificar y visualizar la variabilidad inherente al tamaño de la mayor componente fuertemente conexa cuando se generan varios grafos aleatorios con el mismo número de nodos (***n***) y el mismo número medio de conexiones por nodo (***m***).