Taller Teórico 2 Algoritmos 20251S
Presentado por: John Alejandro Pastor Sandoval

Haga una copia de este documento, debe nombrarlo como su ldap (usuario institucional) y subirlo en Entrega

Fecha de entrega: máximo el 21 jul 2025

Punto 1. Verdadero o falso

1. **[0.2]** Dado un grafo conectado G = (V, E), si un vértice v ∈ V es visitado durante el nivel k de una BFS desde el vértice fuente s ∈ V, entonces cada camino de s a v tiene una longitud máxima de k.
   R./ Falso. BFS nos da la mínima distancia entre s y v (osea k), sin embargo, esto no implica que no existan caminos más largos, por lo que no todos los caminos tendrán una longitud máxima de k.

2. **[0.2]** La búsqueda en profundidad (DFS) tomará Θ(V^2) tiempo en un grafo G = (V, E) representado como una matriz de adyacencia.
   R./ Verdadero. Si porque para la matriz toca leer fila por fila por cada vértice para explorar completamente.

3. **[0.2]** Dada una representación de lista de adyacencia de un grafo dirigido G = (V, E), se tarda O(V) tiempo en calcular el grado de entrada de cada vértice.
   R./ Falso. No es O(V) porque la complejidad en tiempo de calcular el grado de cada entrada es O(V + E)

4. **[0.1]** El algoritmo de caminos más cortos de Dijkstra puede relajar una arista más de una vez en un grafo con un ciclo.
   R./ Falso. Cada vértice se extrae de la cola una sola vez si no el algoritmo falla. Además apenas se determina la distancia mínima la distancia no varía más.

5. **[0.1]** Dado un grafo dirigido ponderado G = (V, E, w) y una fuente s ∈ V, si G tiene un ciclo de peso negativo en algún lugar, entonces el algoritmo de Bellman-Ford necesariamente calculará un resultado incorrecto para algún δ(s, v).
   R./ Falso. Si el ciclo de peso negativo no es accesible desde s, los caminos más cortos a los vértices alcanzables siguen correctos

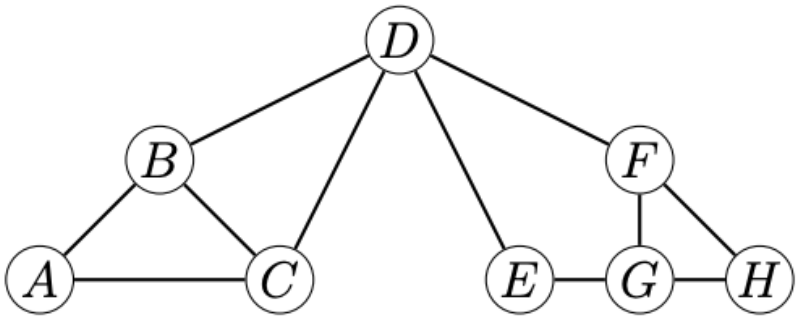
6. **[0.1]** El tiempo de ejecución de un algoritmo de programación dinámica es siempre Θ(P) donde P es el número de subproblemas.
   R./ Falso. El tiempo de ejecución no solo se basa en el número de problemas P, si no también en el tiempo combinando los resultados para la solución del problema principal.

7. **[0.1]** Para un algoritmo de programación dinámica, calcular todos los valores de forma ascendente (bottom-up) es asintóticamente más rápido que usar recursión y memorización.

   R./ Falso. En ambos casos existe la misma complejidad ya que cada subproblema se resuelve y se vuelve a utilizar después para la solución del problema principal.

Punto 2: Grafos y recorridos

1. [1.0] Supongamos que usted realizó un mapa de la UNAL representado como el siguiente grafo que conecta varios edificios. Usted decidió usar una lista de adyacencia, y **cada lista está ordenada ascendentemente**, es decir, los nodos en cada lista aparecen alfabéticamente. (Esto influye el orden en que itera los nodos en las búsquedas)



   A: [B, C]
   B: [A, C, D]
   C: [A, B, D]
   D: [B, C, E, F]
   E: [D, G]
   F: [D, G, H]
   G: [E, F, H]
   H: [G, F]

   a. [0.2] Supongamos que usted finalizó clase en el edificio A y tiene que ir al H, usa una BFS para encontrar la ruta mínima. Escriba el orden de los edificios que pertenecen a la ruta mínima de A a H. (Solo escribir el orden de los nodos y ya)

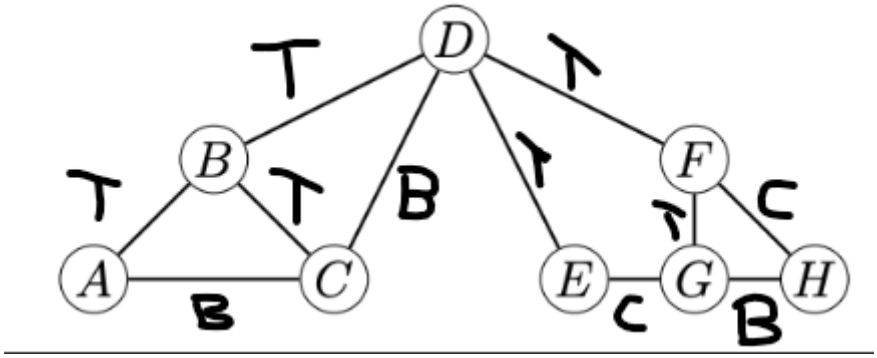
      R/ A → B → D → F → H

   b. [0.2] Supongamos que usted prefiere realizar una DFS, ya que tiene un hueco entre clases y cualquier camino le sirve. El orden de los edificios por los que pasa si usa una DFS para encontrar el camino desde A a H es: (Solo escribir el orden de los nodos y ya)

R/ A → B → C → D → F → G → H

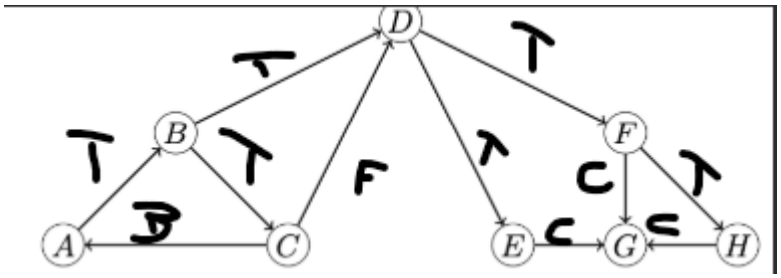
c. [0.2] Al tener miedo de gastar mucho tiempo en los huecos usted decidió encontrar si existe algún ciclo en el grafo. Para ello usó una DFS con recursión lanzada desde el nodo A. Según el recorrido de la DFS marque los arcos por su tipo. Haga una copia del grafo, simule la DFS y marque cada arco según corresponda.
   i. Tipo T: *Tree Edge*. Se identifica cuando se llega a un nodo no visitado. Es decir si v fue descubierto por primera vez explorando una arista (u,v)
   ii. Tipo F: *Forward Edge es un* arco *(u,v)* tal que *v* es descendiente de *u* en el árbol de dfs (i)
   iii. Tipo B: *Back Edge es un* arco *(u,v)* tal que *v* es ancestro de *u* en el árbol de dfs (i)
   iv. Tipo C: Ninguna de las anteriores aplica.

   R/

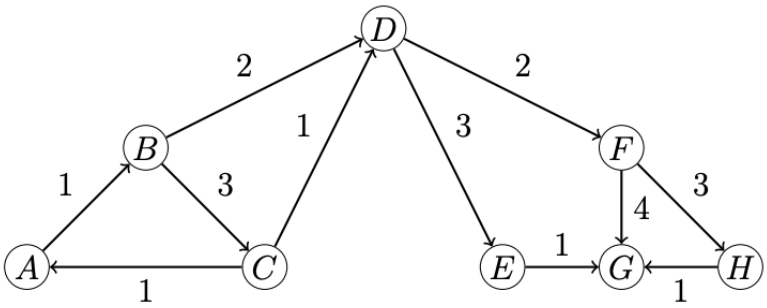


d. **[0.2]** Debido a tropeles en la universidad algunos caminos fueron modificados para solo ir en una dirección. Dado el nuevo grafo, simule una DFS con recursión y marque los arcos según el tipo.

   R/



e. **[0.2]** ¿Torniquetes en la Universidad? Con esta nueva medida ahora cada camino tiene un tiempo para cruzarlo, dados los tiempos de cada camino encuentre la ruta mínima de A al resto de los nodos usando dijkstra con cola de prioridad y escriba el orden en el que los nodos fueron removidos de la cola (Solo escribir el orden de los nodos y ya)



R/ A → B → C→ D→ E→ G→ H→ F

Punto 3. DP - Relación de recurrencias

En el estudio de la programación dinámica, uno de los pasos más cruciales es la correcta identificación y formulación de la **relación de recurrencia**. Esta relación describe cómo la solución de un subproblema más grande se construye a partir de las soluciones de subproblemas más pequeños. Es la parte fundamental de cualquier solución de programación dinámica.

Por ejemplo la secuencia de Fibonacci se define como F(0)=0, F(1)=1, y F(n)=F(n−1)+F(n−2) para n>1. Y su relación de recurrencia es:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

El subproblema F(n) se resuelve sumando las soluciones de los subproblemas F(n−1) y F(n−2).

Para las siguientes DPs defina que representa DP[i] o DP[i][j] y formule la **relación de recurrencia como una funcion matemática por partes.**

(a) [0.2] Suma Máxima de Subarreglo Contiguo (Kadane's Algorithm - DP): Dado un arreglo de números enteros, encuentre la suma máxima de un subarreglo contiguo (que contenga al menos un número). Defina DP[i] y formule su relación de recurrencia

Ejemplo: Para [-2, 1, -3, 4, -1, 2, 1, -5, 4], la suma máxima es 6 (correspondiente a [4, -1, 2, 1]).

R/

$$DP[i] = \max(arr[i],\ DP[i-1] + arr[i])$$

$$DP[i] = \begin{cases} arr[0], & \text{si } i = 0 \\ \max(DP[i-1] + arr[i],\ arr[i]), & \text{si } i > 0 \end{cases}$$

(b) [0.2] Mínimo Número de Monedas (Coin Change - Mínimo Monedas):
Dado un conjunto de monedas con diferentes denominaciones y una cantidad total, calcule el mínimo número de monedas necesarias para sumar esa cantidad. Si la cantidad no se puede sumar con las monedas dadas, retorne -1. Asuma que tiene una cantidad ilimitada de cada tipo de moneda. Defina DP[i] y formule su relación de recurrencia

Ejemplo: Monedas = [1, 2, 5], Cantidad = 11. El resultado es 3 (5 + 5 + 1).

R/

$$DP[i] = \min\left(DP[i],\ DP[i-c] + 1\right) \quad \text{si } i \geq c$$

$$DP[i] = \begin{cases} 0 & \text{si } i = 0 \\ \min\limits_{c \in \text{monedas},\ i \geq c} (DP[i-c] + 1) & \text{si } i > 0 \end{cases}$$

(c) [0.2] Saltos en un Arreglo (Jump Game - Mínimo Saltos):
Dado un arreglo de enteros nums, donde cada valor representa la longitud máxima de salto desde la posición i. Su objetivo es alcanzar el último índice con el mínimo número de saltos. Siempre puede llegar al último índice**.** Defina DP[i] y formule su relación de recurrencia

Ejemplo: nums = [2, 3, 1, 1, 4]. El resultado es 2 (saltar 1 paso desde el índice 0 a 1, luego 3 pasos al último índice).

R/

$$DP[i] = \min(DP[i],\ DP[j] + 1)$$

$$DP[i] = \begin{cases} 0 & \text{si } i = 0 \\ \min\limits_{0 \leq j < i,\ j + nums[j] \geq i} (DP[j] + 1) & \text{si } i > 0 \end{cases}$$

(d) [0.2] Caminos Únicos en una Cuadrícula (Unique Paths): Un robot está en la esquina superior izquierda de una cuadrícula de m×n. El robot solo puede moverse hacia abajo o hacia la derecha en cualquier momento. El robot intenta llegar a la esquina inferior derecha.. ¿Cuántos caminos únicos hay?. Defina DP[i]][j] y formule su relación de recurrencia

Ejemplo: Para una cuadrícula de 3×7, el número de caminos únicos es 28.

R/

$$DP[i][j] = DP[i-1][j] + DP[i][j-1]$$

$$DP[m-1][n-1]$$

(e) [0.2] Subsecuencia Común Más Larga (Longest Common Subsequence - LCS):
Dadas dos cadenas *texto1* y *texto2*, encuentre la longitud de su subsecuencia común más larga. Una subsecuencia de una cadena es una nueva cadena formada eliminando cero o más caracteres de la cadena original sin cambiar el orden de los caracteres restantes. Una

subsecuencia común es una subsecuencia que es común a ambas cadenas. Defina DP[i] y formule su relación de recurrencia

Ejemplo: texto1 = "abcde", texto2 = "ace". La subsecuencia común más larga es "ace" y su longitud es 3.

R/ $$DP[i][j] = \begin{cases} DP[i-1][j-1] + 1 & \text{si } texto1[i-1] = texto2[j-1] \\ \max(DP[i-1][j],\ DP[i][j-1]) & \text{si } texto1[i-1] \neq texto2[j-1] \end{cases}$$

Punto 4. Ballerina Capuccina

La galería de Ballerina Capuccina tiene récord históricos de la menor asistencia de visitantes desde que el arte por inteligencia artificial fue creado. Debido a la dura situación tuvo que cerrar su galería y declararse en quiebra.

La colección de Ballerina cuenta con *N* grandiosos cuadros rectangulares de diferentes medidas, que deben ser transportados desde la galería hasta su casa. Para ello fue rentada una carretilla rectangular con suficiente espacio para colocar cada uno de los cuadros.

Por la lastimosa situación económica de Ballerina, la carretilla debe ser retornada lo más pronto posible. Usted fue contratado para calcular el mínimo número de viajes necesarios y cuales cuadros llevar en cada viaje, cumpliendo las demandas artísticas de Ballerina:

- Cada cuadro debe ser colocado en la carretilla sin rotarlo, ella quiere ver su arte en el camino a casa.
- Cada cuadro debe colocarse exponiendo su arte hacia el cielo, Ballerina quiere que los dioses admiren su arte por última vez.
- Cualquier cuadro puede colocarse en la carretilla. Ballerina considera todos sus cuadros igual de importantes.
- Solo se puede poner un cuadro sobre otro si está totalmente contenido, es decir, el cuadro con medidas *x,y* puede ir sobre uno con medidas *w,h* sii *x* <= *w && y*<=*h*. Para Ballerina ninguna pieza puede opacar a otra.

Si existen múltiples soluciones mínimas, Ballerina acepta cualquiera como válida. Cada cuadro se puede representar como una pareja $[w_i, h_i]$ (ancho y altura).Quiere implementar una estructura de datos que soporte las siguientes operaciones de datos de la manera más eficiente posible.

1. **InsertPainting(w,h):** Insertar un cuadro con dimensiones w, h
2. **AssignPaintingToTrip(w,h)**: Asignar una pintura con dimensiones w,h a alguno de los viajes existentes, o crear un nuevo viaje si es necesario.
3. **CalculateTrips():** Utilizar la funcion 2. Para calcular los viajes de las pinturas insertadas hasta ese momento.. Debe calcular el número de viajes y la lista de dimensiones de cada pintura por viaje.

Se espera que para las respuestas describa las **ideas y pseudocódigo**. No se espera código de ningún lenguaje o pseudocódigo sin explicación de la idea

(a) [0.1] Proporcione un ejemplo con al menos 10 cuadros donde calcule el número de viajes y que cuadros enviar en cada viaje.

R/ Número mínimo de viajes = 3

Viaje 1: [5,5], [4,4], [3,3], [2,2]

Viaje 2: [5,4], [4,3], [3,2], [2,1]

Viaje 3: [5,3], [1,1]

P1	[5,5]
P2	[4,4]
P3	[3,3]
P4	[2,2]
P5	[5,4]
P6	[4,3]
P7	[3,2]
P8	[2,1]
P9	[5,3]
P10	[1,1]

(b) [0.1] Explique qué estructura de datos necesita y cuales son los miembros de su estructura para poder guardar la lista de viajes y cuadros.

R/ Se necesitaría un arreglo de pilas, ya que tendremos una pila que servirá para representar cómo se apilan los cuadros en la carretilla, siempre tendremos acceso al cuadro que está arriba de la carretilla (el último que se guardó). Una lista con todos los viajes, la cual podemos recorrer para asignar a cada nuevo cuadro al primer viaje cuyo cuadro superior sea lo bastante grande.

(c) [0.2] Proporcione un algoritmo que implemente InsertPainting(w,h) y calcule su complejidad

R/

```
function InsertPainting(w, h):
    newPainting = Painting(width = w, height = h, next = null)

    for each trip in trips:
    if trip.topWidth ≥ w AND trip.topHeight ≥ h:
            newPainting.next = trip.top
            trip.top = newPainting
            trip.topWidth = w
            trip.topHeight = h
            return

    newTrip = Trip(
        top = newPainting,
        topWidth = w,
        topHeight = h
    )
    trips.append(newTrip)
```

(d) [0.2] Proporcione un algoritmo que implemente AssingPaintingToTrip(w,h), calcule su complejidad y explique cómo encontrar cual es la mejor opción de viaje o si debe crear un viaje nuevo.

R/

```
function AssignPaintingToTrip(w, h):
    mejorTrip ← null
  mejorSlack ← ∞

    for each trip in trips:
    if trip.topWidth ≥ w AND trip.topHeight ≥ h:
      slackW ← trip.topWidth - w
      slackH ← trip.topHeight - h
            slack ← slackW * slackH
            if slack < mejorSlack:
                mejorSlack ← slack
                mejorTrip ← trip

  if mejorTrip ≠ null:
        newPainting.next ← mejorTrip.top
        mejorTrip.top ← newPainting
        mejorTrip.topWidth  ← w
        mejorTrip.topHeight ← h
        return
```

```
    newPainting.next ← null
    newTrip ← Trip(
        top       = newPainting,
        topWidth  = w,
        topHeight = h
    )
    trips.append(newTrip)
```

(e) [0.2] Proporcione un algoritmo que implemente CalculateTrips() usando el método que definió en (e). Calcule su complejidad

R/

```
function CalculateTrips():
    trips = empty List<Trip>

    for each p in paintings_inserted:
        AssignPaintingToTrip(p.width, p.height)

       result = []  // lista de viajes, donde cada viaje es lista de
pares (w,h)
    for each trip in trips:
        stack_list = []
        node = trip.top
     while node ≠ null:
            stack_list.append((node.width, node.height))
            node = node.next
        result.append(stack_list)

    return result
```

(f) [0.2] Ballerina, considerando si es posible reducir el número de viajes, le pregunta si se puede reducir el número de viajes al permitir una rotación de 90 grados al colocar los cuadros en la carretilla. Explique si es posible y qué cambios necesita en sus métodos.

R/ Sí hay posibilidad de disminuir los viajes si se permiten rotar 90 grados los cuadros en la carretilla, y los cuadros que antes no cabían, ahora podrían si se cambia la orientación de cómo se guardan, entonces se harían los siguientes cambios:

InsertPainting (w,h): Primero se intenta colocar el cuadro con la orientación normal, si no es posible se intenta con la nueva orientación del cuadro. En caso de que esto tampoco funcione entonces se debería crear un nuevo viaje, siendo este el caso más extremo.

AssignPaintingToTrip (w,h) : Debemos considerar las dos rotaciones posibles al calcular la mejor opción de viaje Para cada viaje, entonces hay verificar si se puede apilar como [w,

h] o como [h, w]. Si al calcular el slack para ambas orientaciones encontramos una orientación que produzca el viaje con menor slack aplicar el cuadro en esa orientación.

Punto 5. La leyenda de la Mochila

Imagine que usted es un héroe en las Tierras Antiguas de la **Unión de Nobles Aventureros Legendarios (UNAL)**, un vasto reino asolado por criaturas míticas. Se aventura en una mazmorra ancestral y se ha topado con varios enemigos formidables. Cada enemigo, al ser derrotado, le consume **puntos de vida(VP)** que usted debe "gastar" para poder derrotarlo (representando el esfuerzo y los recursos invertidos en el combate) y le otorga **Fragmentos de Conocimiento Arcana (FCA)** que usted puede absorber para volverse más fuerte y sabio. Sin embargo, su capacidad para enfrentar batallas es limitada, y su objetivo es elegir un subconjunto de enemigos para derrotar, maximizando la suma total de Fragmentos de Conocimiento Arcana que puede obtener, sin exceder un límite de Puntos de Vitalidad que puede gastar en su campaña por la mazmorra.

Objetivo: Maximizar el total de **Fragmentos de Conocimiento Arcana (FCA)** obtenido de los enemigos derrotados, sin exceder el **límite de Puntos de Vida (VP Max)** que usted puede gastar.

Parámetros:

- NNúmero total de enemigos disponibles.
- VP_{MAX}: El máximo de puntos de vida que usted puede gastar.
- Para cada enemigo i:
    - vp_i: Costo de vida para derrotar al enemigo i.
    - fca_i:: Fragmentos de Conocimiento Arcana que otorga el enemigo i al ser derrotado

(a) [0.2] Muestre una tabla de los estados para un caso sencillo, donde se ilustre cómo se llenaría la tabla de programación dinámica. Considere VP_{MAX}=5 y los siguientes enemigos:

Enemigo	vp	fca
Elfo	2	3
Ogro	3	4
Calavera	1	2

R/

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
E	0	0	3	3	3	3
O	0	0	3	4	4	7
C	0	2	3	5	6	7

(b) [0.2] Defina la relación de recurrencia que describe el problema de optimización para su héroe. Esta relación debe expresar cómo se calcula el valor de un subproblema en función de los valores de subproblemas más pequeños, presentada como una función matemática por partes.

R/

$$DP[i][w] = \begin{cases} DP[i-1][w], & \text{si } v_{p_i} > w \\ \max(DP[i-1][w],\ DP[i-1][w - v_{p_i}] + fca_i), & \text{si } v_{p_i} \leq w \end{cases}$$

(c) [0.2] Enumere y describa claramente los casos base necesarios para la relación de recurrencia que usted ha definido.

R/

1) Cuando no hay VP o capacidad de vida:
   $$\forall i, DP[i][0] = 0$$
2) Si no hay enemigos:
   $$\forall w, DP[0][w] = 0$$

(d) [0.2] Escriba el pseudocódigo para el algoritmo de programación dinámica que resuelve el problema

R/

```
Algorithm KnapsackPD(enemies, VPMAX):
    let N ← length(enemies)
    for i from 0 to N:
        for w from 0 to VPMAX:
            if i == 0 or w == 0:
                DP[i][w] ← 0
            else:
                (vp, fca) ← enemies[i]
                if vp > w then
                    DP[i][w] ← DP[i-1][w]
                else
                    DP[i][w] ← max(
                        DP[i-1][w],
                        DP[i-1][w - vp] + fca)
    return DP[N][VPMAX]
```

(e) [0.2] Compare la **complejidad temporal** y **espacial** de la solución de programación dinámica (con memorización) con una posible solución recursiva sin memorización para este desafío. Explique cómo la programación dinámica mejora significativamente la eficiencia en este contexto.

R/

1) **Programación dinámica:** Mejora la eficiencia ya que calculamos subproblemas varias veces y lo almacena una sola vez cada uno, de esta forma podemos resolver los problemas con subproblemas ya resueltos de forma práctica y relativamente rápida.

$$tiempo: O(N \times VP_{max})$$
$$espacio: O(N \times VP_{max})$$

2) **Recursivo sin memorización:**

$$tiempo: O(2^N)$$
$$espacio: O(N)$$

Punto 6. Calificaciones

Valide sus respuestas, basado en la correctitud, su aprendizaje y el valor de cada parte, calcule su calificación de cada punto (en el rango decimal [0,1]) y sumarlas para calcular la calificación total del taller. Si el profesor no está de acuerdo asignará una calificación de 'Coevaluación' y la calificación total del punto será el promedio

	Autoevaluación	**Coevaluación**	**Total**
Punto 1	**5**		
Punto 2	**5**		
Punto 3	**4**		
Punto 4	**3,5**		
Punto 5	**4**		
Calificación Total	**21,5**		