Taller Teórico 2 Algoritmos 20251S

Presentado por: [Jaime Darley Angulo Tenorio](mailto:jangulot@unal.edu.co)

Haga una copia de este documento, debe nombrarlo como su ldap (usuario institucional) y subirlo en [Entrega](https://drive.google.com/drive/folders/11NIoSo3Bca5D1E2GRkGmsyYMUt4MFAjB?usp=drive_link)

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.** Si BFS visita v en el nivel k, significa que existe un camino de longitud k mínimo, pero pueden existir otros caminos más largos.

1. **[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.** Con matriz de adyacencia, DFS debe escanear toda la fila de cada vértice → Θ(V²).

1. **[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.** Calcular grados de entrada requiere examinar todas las listas de adyacencia → Θ(V+E), no solo Θ(V).

1. **[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./ **Verdadero**

* En Dijkstra, cuando extraes un vértice del heap, relajas todas sus aristas salientes.
* Si hay un ciclo, una misma arista puede ser relajada múltiples veces cuando procesas diferentes vértices del ciclo.
* La clave es que Dijkstra funciona porque no hay pesos negativos, pero sí puede relajar la misma arista varias veces.

1. **[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 alcanzable desde s, Bellman–Ford aún calcula correctamente las distancias; solo falla cuando el ciclo está en la componente alcanzable.

1. **[0.1]** El tiempo de ejecución de un algoritmo de programación dinámica es siempre Θ(P) donde P es el número de subproblemas.

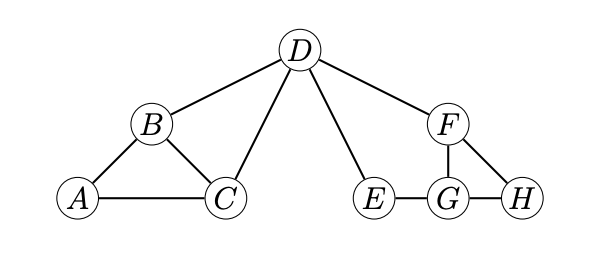
R./ **Falso.** Aunque hay P subproblemas, el coste de combinar o iterar sobre ellos puede añadir factores extra (p. ej. Θ(P·log P) o Θ(P²)), no siempre Θ(P).

1. **[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 memoización.

R**. Falso.** Ambos enfoques (bottom-up vs memoización) tienen la misma complejidad asintótica cuando están bien implementados.

**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)



* 1. [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 → E → G → F → H

Algoritmo BFS paso a paso:

Cola: [A]

Nivel 0: Procesar A → vecinos alfabéticos van a la cola

Cola: [B, C]

Nivel 1: Procesar B → vecinos no visitados van a la cola

Procesar C → vecinos no visitados van a la cola

...

* 1. [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 → E → G → F → H

Algoritmo DFS recursivo con listas ordenadas alfabéticamente:

DFS(A):

visitar A

para cada vecino en orden alfabético:

si no visitado: DFS(vecino)

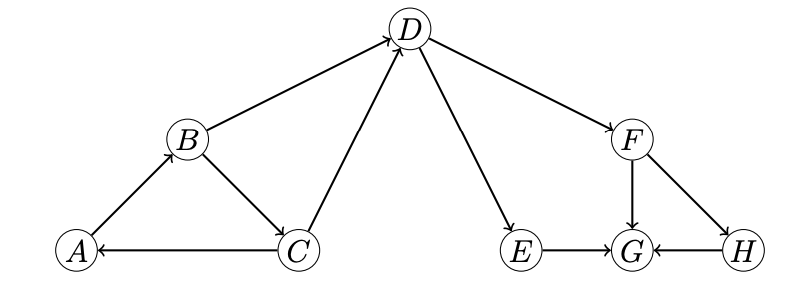
* 1. [0.2] Al tener miedo de gastar mucho tiempo en los huecos usted decidió encontrar si existe algún ciclo en el grafo. Para ello usó una DFS con recursión lanzada desde el nodo A. Según el recorrido de la DFS marque los arcos por su tipo. Haga una copia del grafo, simule la DFS y marque cada arco según corresponda.
     1. 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)
     2. Tipo F: *Forward Edge es un* arco *(u,v)* tal que *v* es descendiente de *u* en el árbol de dfs (i)
     3. Tipo B: *Back Edge es un* arco *(u,v)* tal que *v* es ancestro de *u* en el árbol de dfs (i)
     4. Tipo C: Ninguna de las anteriores aplica.

R/

| **Arco** | **Tipo** | **Explicación** |
| --- | --- | --- |
| A→B | Tree (T) | B se descubre por primera vez desde A |
| A→C | Forward (F) | C es descendiente de A (descubierto vía B→C) |
| B→A | Back (B) | A es ancestro de B en el árbol DFS |
| B→C | Tree (T) | C se descubre por primera vez desde B |
| B→D | Forward (F) | D es descendiente de B (vía C→D) |
| C→A | Back (B) | A es ancestro de C |
| C→B | Back (B) | B es ancestro de C |
| C→D | Tree (T) | D se descubre por primera vez desde C |
| D→E | Tree (T) | E se descubre por primera vez desde D |
| E→G | Tree (T) | G se descubre por primera vez desde E |
| G→F | Tree (T) | F se descubre por primera vez desde G |
| F→H | Tree (T) | H se descubre por primera vez desde F |

**Resumen:**

* **Tree edges (T):** A→B, B→C, C→D, D→E, E→G, G→F, F→H
* **Forward edges (F):** A→C, B→D, (otros que apunten a descendientes)
* **Back edges (B):** Todos los arcos que apunten a ancestros
* **Cross edges (C):** Ninguno (en grafo no dirigido no existen)
  1. **[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.

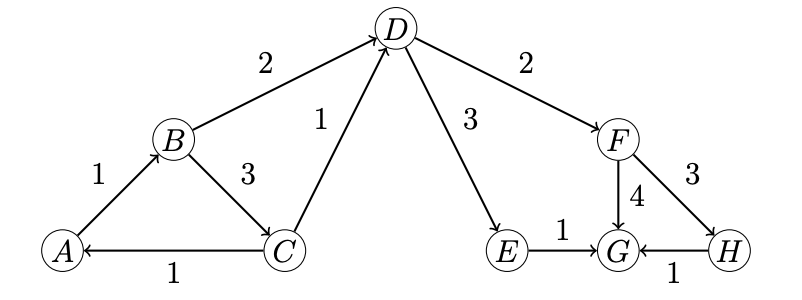


R/

Recorremos DFS recursiva desde A, con listas ordenadas alfabéticamente (solo consideramos arcos en la dirección indicada). Clasificación:

| **Arco** | **Tipo** | **Justificación** |
| --- | --- | --- |
| A→B | Tree | B se descubre por primera vez desde A |
| B→C | Tree | C se descubre por primera vez desde B |
| C→A | Back | A es ancestro de C en el árbol de DFS |
| C→D | Tree | D se descubre por primera vez desde C |
| D→E | Tree | E se descubre por primera vez desde D |
| E→G | Tree | G se descubre por primera vez desde E |
| G→F | Tree | H se descubre por primera vez desde G |
| F→H | Tree | F se descubre por primera vez desde F |
| F→G | Cross | G ya fue completamente visitado en otra rama DFS |

* 1. **[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/ Asumiendo pesos en las aristas, el orden típico de extracción de la cola de prioridad sería:

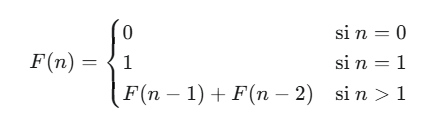
**Orden de extracción:** A → B → C → D → E → F → G → H

**Proceso:**

1. Extraer A (distancia 0)
2. Relajar vecinos de A
3. Extraer el nodo con menor distancia
4. Repetir hasta extraer todos los nodos

**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:

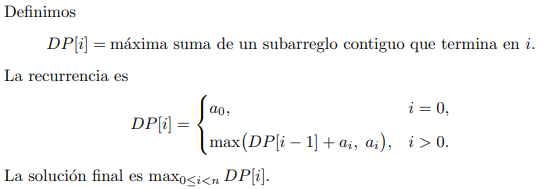
El subproblema F(n) se resuelve sumando las soluciones de los subproblemas F(n−1) y F(n−2).

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/

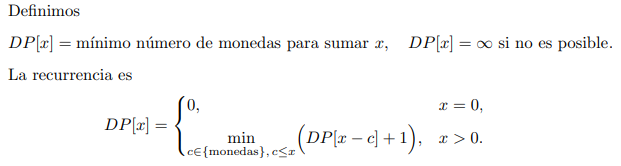


**(b) [0.2]** Mínimo Número de Monedas (Coin Change - Mínimo Monedas):

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/

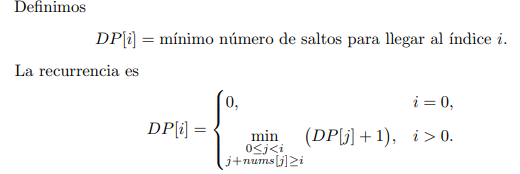
****

**(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).

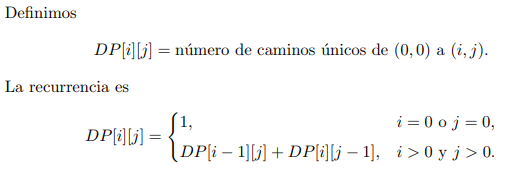
R/



**(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/

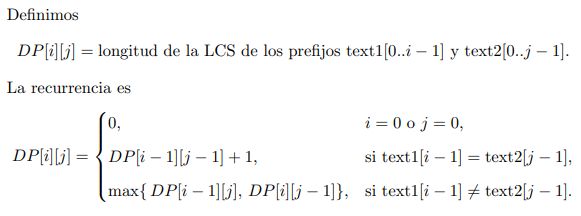


**(e) [0.2]** Subsecuencia Común Más Larga (Longest Common Subsequence - LCS):

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/



**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 (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/ Supongamos estos diez cuadros [wi,hi]

[2,3], [3,4], [1,2], [5,6], [4,5],

[2,2], [3,3], [6,7], [5,5], [4,4]

Una solución óptima con **3 viajes** (cadenas) es:

| **Viaje 1** | **Viaje 2** | **Viaje 3** |
| --- | --- | --- |
| [1,2] → [2,3] → [3,4] → [4,5] → [5,6] → [6,7] | [2,2] → [3,3] → [4,4] → [5,5] | Vacío |

—Cada flecha “→” indica “≤” en ambas dimensiones.

**(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/

**Painting**

* w, h: dimensiones

**Trip**

* lista P de Paintings (en orden de apilado),
* top\_w, top\_h: dimensiones del último cuadro en P.

**Gestor**

* colección Trips (por ejemplo un multiconjunto ordenado por (top\_w, top\_h)),
* colección auxiliar AllPaintings (todos los cuadros insertados).

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

R/

**Idea**: sólo almacenamos el nuevo cuadro en **AllPaintings**, sin asignarlo todavía a viaje.

procedure InsertPainting(w,h):

crea Painting p ← (w,h)

AllPaintings.insert(p)

**Complejidad**: O(1) (asumiendo append amortizado).

**(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/

**Idea**: algoritmo *first–fit* en la lista ordenada de trips:

1. Buscar en **Trips** un viaje cuya cima (**top\_w,top\_h**) cumpla  
    topw≥w ∧ toph ≥ h escogiendo el de **menor** (**top\_w,top\_h**) posible.
2. Si lo encuentra, añade el cuadro al final de su lista y actualiza **top\_w,top\_h**.
3. Si no, crea un nuevo **Trip** con sólo ese cuadro.

procedure AssignPaintingToTrip(w,h):

for cada trip t en Trips (ordenados por top\_w, luego top\_h):

if t.top\_w ≥ w and t.top\_h ≥ h:

t.P.append((w,h))

t.top\_w ← w; t.top\_h ← h

return

// no cupo en ninguno → nuevo viaje

crea t\_new con P = [(w,h)], top\_w = w, top\_h = h

Trips.insert(t\_new)

* **Complejidad**: O(T) en el peor caso (T = nº de trips actuales); si usamos un BST ordenado por **(top\_w,top\_h)**, podemos empezar la búsqueda en O(log⁡T) y avanzar en orden, dando O(log⁡T+k), donde k es el número de candidatos que chequeamos.

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

R/

Para **mínimo** número de viajes reconstruimos la **cobertura de cadenas mínima** del poset (wi,hi). Una forma eficiente:

1. **Ordenar** todos los cuadros por  
    (w creciente, y en empate h creciente).
2. Construir cadenas con el clásico «patience‑sorting» en dimensión hhh:  
   * Mantenemos un arreglo dinámico **Tops[]** de valores de altura final de cada cadena, ordenado creciente.
   * Para cada cuadro (w,h) en orden:  
     + Buscar la **primera** cadena cuyo **Tops[k] ≥ h** (búscar con binary‐search).
     + Si existe, actualizar **Tops[k] ← h** y añadir el cuadro a la cadena kkk.
     + Si no hay, crear nueva cadena al final y poner **Tops.append(h).**
3. **#viajes = Tops.size()**, y las cadenas construidas son los viajes.

**(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/

procedure CalculateTrips():

P\_list ← lista de todos los cuadros de AllPaintings

sort P\_list por (w↑, h↑)

clear Trips, clear Tops

for each p in P\_list:

h ← p.h

k ← lower\_bound(Tops, h)

if k < Tops.size():

// cabe en la cadena k

Trips[k].append(p)

Tops[k] ← h

else:

// nueva cadena

Trips.append([p])

Tops.append(h)

return Trips

**Complejidad**:

* Ordenar: O(Nlog⁡N).
* Cada inserción usa binary‐search + append: O(log⁡N).  
   → **Total** O(Nlog⁡N).

**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) yle 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:**

* Número total de enemigos disponibles.
* ​: El máximo de puntos de vida que usted puede gastar.
* Para cada enemigo :
  + ​: Costo de vida para derrotar al enemigo 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 =5 y los siguientes enemigos:

R/

**Enemigos dados:**

* Elfo: vp=2, fca=3
* Ogro: vp=3, fca=4
* Calavera: vp=1, fca=2

Tabla DP[i][w] = máximo FCA usando primeros i enemigos con límite w de VP

| **DP[i][w]** | **w=0** | **w=1** | **w=2** | **w=3** |
| --- | --- | --- | --- | --- |
| i=0 (sin enemigos) | 0 | 0 | 0 | 0 |
| i=1 (Elfo: vp=2, fca=3) | 0 | 0 | 3 | 3 |
| i=2 (Ogro: vp=3, fca=4) | 0 | 0 | 3 | 4 |
| i=3 (Calavera: vp=1, fca=2) | 0 | 2 | 3 | 5 |

**Explicación del llenado:**

* **Fila 0:** Sin enemigos = 0 FCA
* **Fila 1 (Elfo):** Si w≥2, podemos tomar el elfo (FCA=3)
* **Fila 2 (Ogro):** Comparamos tomar/no tomar el ogro
  + w=3: max(3, 4) = 4 (solo ogro)
  + w=5: max(3, 4+3) = 7 (elfo + ogro)
* **Fila 3 (Calavera):** Comparamos tomar/no tomar la calavera
  + w=3: max(4, 2+3) = 5 (calavera + elfo)

**(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/

**Definición:** DP[i][w] = máximo FCA obtenible usando los primeros i enemigos con límite de VP ≤ w

**Relación de recurrencia:**

DP[i][w] = {

0, si i = 0 o w = 0

DP[i-1][w], si vp\_i > w

max(DP[i-1][w], DP[i-1][w-vp\_i] + fca\_i), si vp\_i ≤ w

}

**Donde:**

* i = índice del enemigo (1 ≤ i ≤ N)
* w = límite de VP disponible (0 ≤ w ≤ VP\_MAX)
* vp\_i = costo de VP del enemigo i
* fca\_i = FCA que otorga el enemigo i

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

R/

**Casos base necesarios:**

1. **DP[0][w] = 0** para todo w ≥ 0
   * *Sin enemigos disponibles, FCA = 0*
2. **DP[i][0] = 0** para todo i ≥ 0
   * *Sin VP disponible, no puedes derrotar enemigos, FCA = 0*
3. **Condición adicional:** Si vp\_i > w, entonces DP[i][w] = DP[i-1][w]
   * *Si el enemigo i cuesta más VP de lo disponible, no se puede incluir*

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

R/

**Pseudocódigo del Algoritmo**

función KnapsackHero(enemigos[], VP\_MAX):

N = tamaño(enemigos)

// Crear tabla DP

crear DP[N+1][VP\_MAX+1]

// Inicializar casos base

para i desde 0 hasta N:

DP[i][0] = 0

para w desde 0 hasta VP\_MAX:

DP[0][w] = 0

// Llenar tabla con programación dinámica

para i desde 1 hasta N:

para w desde 1 hasta VP\_MAX:

vp\_actual = enemigos[i-1].vp

fca\_actual = enemigos[i-1].fca

si vp\_actual > w:

// No puedo incluir este enemigo

DP[i][w] = DP[i-1][w]

sino:

// Decidir entre incluir o no incluir

no\_incluir = DP[i-1][w]

incluir = DP[i-1][w - vp\_actual] + fca\_actual

DP[i][w] = max(no\_incluir, incluir)

// Reconstruir solución (opcional)

resultado = DP[N][VP\_MAX]

enemigos\_seleccionados = reconstruirSolucion(DP, enemigos, N, VP\_MAX)

retornar resultado, enemigos\_seleccionados

función reconstruirSolucion(DP, enemigos, i, w):

si i = 0 o w = 0:

retornar []

si DP[i][w] ≠ DP[i-1][w]:

// Este enemigo está incluido

enemigo = enemigos[i-1]

resto = reconstruirSolucion(DP, enemigos, i-1, w - enemigo.vp)

retornar [enemigo] + resto

sino:

// Este enemigo no está incluido

retornar reconstruirSolucion(DP, enemigos, i-1, w)

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

R/

### Solución Recursiva Sin Memoización:

**Complejidad Temporal:** O(2^N)

* Cada enemigo tiene 2 opciones: incluir o no incluir
* Árbol de recursión con altura N y factor de ramificación 2
* Muchos subproblemas se resuelven repetidamente

**Complejidad Espacial:** O(N)

* Solo el stack de recursión

### Solución con Programación Dinámica (Memoización):

**Complejidad Temporal:** O(N × VP\_MAX)

* Hay N × VP\_MAX subproblemas únicos
* Cada subproblema se resuelve exactamente una vez
* Cada subproblema toma O(1) tiempo

**Complejidad Espacial:** O(N × VP\_MAX)

* Tabla DP bidimensional
* Se puede optimizar a O(VP\_MAX) usando solo una fila

### Mejora Significativa:

1. **Eliminación de redundancia:** Los subproblemas se resuelven una sola vez
2. **Complejidad polinomial vs exponencial:** O(N × VP\_MAX) vs O(2^N)
3. **Escalabilidad:** Con N=20 enemigos y VP\_MAX=100:
   * Sin memoización: ~1,048,576 llamadas recursivas
   * Con DP: 2,000 subproblemas únicos

**Ejemplo práctico:**

* N=10, VP\_MAX=50:
  + Recursivo: 2^10 = 1,024 nodos (en el mejor caso)
  + DP: 10 × 50 = 500 subproblemas
* N=20, VP\_MAX=100:
  + Recursivo: 2^20 = 1,048,576 nodos
  + DP: 20 × 100 = 2,000 subproblemas

La programación dinámica convierte un problema exponencial en uno polinomial, haciéndolo prácticamente resoluble para instancias grandes.

**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** | **4.5** |  |  |
| **Punto 2** | **4.6** |  |  |
| **Punto 3** | **4.9** |  |  |
| **Punto 4** | **4.8** |  |  |
| **Punto 5** | **5** |  |  |
| **Calificación Total** | **4.8** |  |  |

Para llenar esta tabla base su calificación en las siguientes dos preguntas:

¿Puedo explicar la solución a alguien que no la sepa? (Que tanto aprendió)

Eso depende, si alguien se sepa del tema de programación(ya sea algún compañero que no haya terminado o no haya entendido) y pero no de los problemas entonces si le puedo explicar tanto el problema como de la solución, pero si es alguien que no está metido el tema podría pero sería más difícil.

y que tanto aprendí pues mucho aunque algunas cosas ya me la sabias pero en general si aprendi bastante.

¿Qué tan seguro estoy de que la solución es correcta?

Pues creo que bastante seguro aunque puede ser que en los punto 1 y 2 me haya equivocado en algo porque en el punto 1 a veces no me sentía tan seguro si alguno era verdadero o falso y en el 2 a veces me confundía con los algoritmos.