8.1. Aplicar el algoritmo de programación dinámica para el problema del cambio de monedas sobre el siguiente ejemplo: **n** = 3, **P** = 9, **c** = (1, 3, 4). ¿Qué ocurre si multiplicamos **P** y **c** por un valor constante, por ejemplo por 1000000? ¿Ocurre lo mismo con el algoritmo voraz? ¿Cómo se podría solucionar?

8.2. El número de combinaciones de **m** objetos entre un conjunto de **n**, denotado por $\binom{n}{m}$, para $n \geq 1$ y $0 \leq m \leq n$, se puede definir recursivamente por:

$$\binom{n}{m} = 1 \quad \text{Si } m = 0 \text{ ó } m = n$$

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \quad \text{Si } 0 < m < n$$

Conociendo que el resultado puede ser calculado también con la fórmula:
n!/(m!·(n-m)!)

a) Dar una función recursiva para calcular $\binom{n}{m}$, usando la primera de las definiciones. ¿Cuál será el orden de complejidad de este algoritmo? Sugerencia: la respuesta es inmediata.

b) Diseñar un algoritmo de programación dinámica para calcular $\binom{n}{m}$. Nota: la tabla construida por el algoritmo es conocida como "el triángulo de Pascal". ¿Cuál será el tiempo de ejecución en este caso?

8.3. Una variante del problema de la mochila es la siguiente. Tenemos un conjunto de enteros (positivos) $A = \{a_1, a_2, ..., a_n\}$ y un entero K. El objetivo es encontrar si existe algún subconjunto de A cuya suma sea exactamente K.

a) Desarrollar un algoritmo para resolver este problema, utilizando programación dinámica. ¿Cuál es el orden de complejidad del algoritmo?
b) Mostrar cómo se puede obtener el conjunto de objetos resultantes (en caso de existir solución) a partir de las tablas utilizadas por el algoritmo.
c) Aplicar el algoritmo sobre el siguiente ejemplo A = {2, 3, 5, 2}, K= 7. ¿Cómo se puede comprobar que la solución no es única?

8.4. Considerar el problema del cambio de monedas. Tenemos monedas de **n** tipos distintos (cada uno con valor $\mathbf{c_i}$), y queremos devolver una cantidad **P**. Dar un algoritmo, con programación dinámica, para calcular el número de formas diferentes de devolver la cantidad **P** (independientemente del número de monedas que se use). ¿Cuál es el orden de complejidad de este algoritmo?

Aplicar el algoritmo sobre el siguiente ejemplo: **n**= 4, **c**= {1, 3, 4, 7}, **P**= 7.

8.5. En el problema del cambio de monedas, en lugar de utilizar la ecuación de recurrencia:

Cambio (i, Q) = min (Cambio(i-1, Q), Cambio(i, Q - c_i)+1)

Decidimos usar la siguiente:

Cambio (i, Q) = $\min_{k=0, \ldots, \lfloor Q/c[i] \rfloor}$ { k + Cambio (i - 1, Q - k·c[i]) }

a) ¿Es correcta esta ecuación de recurrencia para encontrar la solución? Explicar cuál es el significado de esta fórmula.

b) Suponiendo que modificamos el algoritmo de programación dinámica para usar la segunda fórmula, mostrar el resultado del algoritmo para el siguiente ejemplo: n= 4, c= {1, 3, 4}, P= 7.

c) Estimar el orden de complejidad del algoritmo. Compararlo con el algoritmo visto en clase.

8.6. (TG 11.4) Resolver con programación dinámica el problema del cambio de monedas, pero teniendo una cantidad limitada de monedas. La cantidad a devolver es ***C***, tenemos monedas de ***n*** tipos (cuyos valores están dados en ***tipos***: **array**[1,..,n] **de** entero), y de cada tipo tenemos una cierta cantidad de monedas (almacenadas en un array ***cantidad***: **array**[1,..,n] **de** entero). Es decir, de la moneda de valor *tipos[i]* podemos dar una cantidad entre *0* y *cantidad[i]*. **Sugerencia:** observar la ecuación del ejercicio anterior.

8.7. Una agencia de turismo realiza planificaciones de viajes aéreos. Para ir de una ciudad A a B puede ser necesario coger varios vuelos distintos. El tiempo de un vuelo directo de I a J será T[I, J] (que puede ser distinto de T[J, I]). Hay que tener en cuenta que si cogemos un vuelo (de A a B) y después otro (de B a C) será necesario esperar un tiempo de "escala" adicional en el aeropuerto (almacenado en E[A, B, C]).

a) Diseñar una solución para resolver este problema utilizando programación dinámica. Explicar cómo, a partir de las tablas, se puede obtener el conjunto de vuelos necesarios para hacer un viaje concreto.

b) Mostrar la ejecución del algoritmo sobre la siguiente matriz T, suponiendo que todos los E[A, B, C] tienen valor 1. ¿Cuál es el orden de complejidad del algoritmo?

T[i, j]	**A**	**B**	**C**	**D**
A	-	2	1	3
B	7	-	9	2
C	2	2	-	1
D	3	4	8	-

8.8. (TG 11.2) Supongamos una serie de **n** trabajos denominados **a**, **b**, **c**, ... y una tabla **B[1..n, 1..n]**, en la que cada posición **B[i, j]** almacena el beneficio de ejecutar el trabajo **i** y a continuación el trabajo **j**. Se quiere encontrar la sucesión de **m** trabajos que dé un beneficio óptimo. No hay límite en el número de veces que se puede ejecutar un trabajo concreto.

a) Idear un algoritmo por programación dinámica que resuelva el problema. Para ello, definir un subproblema (que permita realizar la combinación de problemas

pequeños para resolver problemas grandes), especifica la ecuación de recurrencia para el mismo (con sus casos base) y después describe las tablas necesarias y cómo son rellenadas.

b) Ejecutar el algoritmo sobre la siguiente tabla, suponiendo que m= 5.

B[i, j]	**a**	**b**	**c**
A	2	2	5
B	4	1	3
C	3	2	2

c) Estimar el tiempo de ejecución del algoritmo. El tiempo estimado ¿es un orden exacto o una cota superior del peor caso?

8.9. En una cierta aplicación del problema de la mochila 0/1, los pesos de los objetos están definidos como valores reales. Por ejemplo, tenemos 5 objetos con pesos **p** = (3.32, 2.15, 2.17, 3.21, π/2) y beneficios **b** = (10.2, 9.2, 8.3, 9.1, 6.5) y capacidad de la mochila **M** = 7.7. ¿Qué problema ocurre al intentar aplicar el algoritmo de programación dinámica? Intenta resolverlo de alguna forma y muestra el resultado. ¿La solución encontrada es la óptima?

8.10. (TG 11.3) Dada una tabla de tamaño **n**x**n** de números naturales, se pretende resolver el problema de obtener el camino de la casilla (1, 1) a la casilla (n, n) que minimice la suma de los valores de las casillas por las que pasa. En cada casilla (i, j) habrán sólo dos posibles movimientos: ir hacia abajo (i+1, j), o hacia la derecha (i, j+1).

a) Resolver el problema utilizando programación dinámica. Indica la ecuación de recurrencia usada, con los casos base necesarios, las tablas para llegar a la solución óptima y para recomponer el camino correspondiente a esa solución óptima.

b) Mostrar la ejecución del algoritmo sobre la siguiente entrada.

2	8	3	4
5	3	4	5
1	2	2	1
3	4	6	5

c) Formular el principio de optimalidad de Bellman sobre este problema y comprobar si se cumple.

8.11. Los algoritmos de divide y vencerás y los de programación dinámica se basan en la resolución de un problema en base a subproblemas. Usando las mismas ecuaciones de recurrencia de los problemas vistos en el tema de programación dinámica (cambio de monedas, mochila 0/1 y algoritmo de Floyd), diseñar algoritmos que resuelvan esos problemas pero con divide y vencerás. Compara la complejidad obtenida con los dos tipos de algoritmos.

En los dos primeros casos, ¿por qué los algoritmos no son comparables (al menos de forma directa) con los algoritmos voraces correspondientes?

8.12. En el algoritmo para el cambio de monedas visto en clase, ¿es posible que al acabar de ejecutarse obtengamos que D[n, P] = +∞? En caso afirmativo, ¿qué indica esta situación? Muéstralo con un ejemplo. En caso contrario, ¿por qué no es posible esa situación?

8.13. (EX) Considerar el siguiente problema: dado un conjunto de números enteros X = $\{x_1, x_2, ..., x_n\}$ y otro entero P, queremos encontrar si existe algún subconjunto $\{y_1, ..., y_k\}$ de X, tal que $P = y_1 * y_2 * ... * y_k$.

Resolver el problema utilizando programación dinámica. No es necesario programar el algoritmo, habrá que dar una ecuación recurrente para resolver el problema, con los casos base, indicar cómo son las tablas que se deben utilizar y la forma de rellenarlas.

A partir de las tablas, mostrar cómo podemos saber si existe tal conjunto o no, y en caso de existir cómo se puede obtener el conjunto solución $\{y_1, y_2, ..., y_k\}$.

Hacer una estimación del orden de complejidad del algoritmo.

Ejecutar sobre el siguiente ejemplo: X= {2, 4, 3, 9, 10}, P= 18.

Nota: tener en cuenta que el problema no es de optimización, sino de encontrar si existe una solución o no.

8.14. En el problema de la mochila (igual que en el problema del cambio de monedas) puede existir en general más de una solución óptima para unas entradas determinadas. ¿Cómo se puede comprobar si una solución óptima es única o no, suponiendo que hemos resuelto el problema utilizando programación dinámica? Dar un algoritmo para que, a partir de las tablas resultantes del problema de la mochila, muestre todas las soluciones óptimas existentes.

8.15. Resolver el siguiente problema usando programación dinámica. Dada una secuencia de enteros positivos ($a_1, a_2, a_3, ..., a_n$), encontrar la subsecuencia creciente más larga de elementos no necesariamente consecutivos. Es decir, encontrar una subsecuencia ($a_{i1}, a_{i2}, ..., a_{ik}$), con ($a_{i1} < a_{i2} < ... < a_{ik}$) y ($1 \le i1 < i2 < ... < ik \le n$). Por ejemplo, para la siguiente secuencia la solución sería longitud 6 (formada por los números señalados en negrita): 3, **1**, 3, **2**, **3**, 8, **4**, 7, **5**, 4, **6**.

8.16. (EX) Usando la fórmula recursiva para el problema de la mochila 0/1 (vista en clase), escribe un procedimiento que resuelva el problema pero con divide y vencerás. El cuerpo del procedimiento debe ser:

Mochila(**i**: entero; **M**: entero; **b**, **p**: array[1..n] de entero): entero

Siendo:
i = Número de objetos a usar (desde **1** hasta **i**). **M** = Capacidad de la mochila.
b, **p** = Beneficio y peso de los objetos. **Valor devuelto** = Beneficio óptimo.

8.17. (EX M02) Considera la variante de los números de Fibonacci, que denominaremos "números de **cuatrinacci**", definida a continuación. El n-ésimo número de cuatrinacci es igual a 3 veces el número (n–1)-ésimo, más 2 veces el (n–2)-ésimo, menos el n-ésimo número de cuatrinacci. El primer y el segundo números de cuatrinacci valen 1 y 3, respectivamente. Se pide lo siguiente.

a) Escribe tres posibles implementaciones para el cálculo del n-ésimo número de cuatrinacci usando: un método descendente de resolución de problemas (por

ejemplo, un algoritmo de divide y vencerás), un método ascendente (por ejemplo, de programación dinámica), y un procedimiento que devuelva el resultado de forma directa, mediante una simple operación aritmética. **Ojo:** las implementaciones deben ser muy simples y cortas.

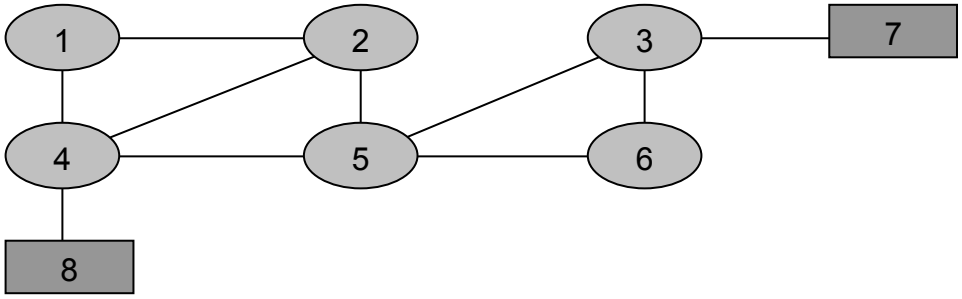
b) Haz una estimación del orden de complejidad de los tres algoritmos del apartado anterior. Compara los órdenes de complejidad obtenidos, estableciendo una relación de orden entre los mismos.

8.18. (EX D02) Considerar que en el problema de los ratones (ejercicio 7.21) todos los pasadizos requieren 1 unidad de tiempo, es decir **P[i, j]** = 1 si existe el pasadizo entre **i** y **j**. En este experimento se coloca sólo un ratón, en una celda dada **S**, y estamos interesados en calcular la probabilidad de que consiga salir del laberinto en el tiempo máximo $\mathbf{t_{max}}$.

En este caso, suponemos que el ratón se mueve de forma completamente aleatoria por el laberinto y que en cada instante de tiempo hace un movimiento, excepto si ya ha salido del laberinto, en cuyo caso no volverá a entrar.

Por ejemplo, si en el laberinto de abajo se coloca el ratón en la celda 1, en el instante siguiente estará en la celda 2 con probabilidad 0.5 y en la celda 4 con probabilidad 0.5. Por lo tanto, la probabilidad de que haya salido en el instante 2 será 0.

Resolver el problema utilizando programación dinámica. Definir la ecuación recurrente, con sus casos base, la estructura de tablas necesarias y escribir el algoritmo para resolver el problema. Aplicarlo al ejemplo para $\mathbf{t_{max}}$= 3 y **S**=1.



8.19. Resolver el siguiente problema con programación dinámica. Tenemos un conjunto de **n** objetos, cada uno con un peso **p** = ($\mathbf{p_1}$, $\mathbf{p_2}$, ..., $\mathbf{p_n}$). El objetivo es repartir los objetos entres dos montones diferentes, de manera que queden lo más equilibrados posible en peso. Esto es, se debe minimizar la diferencia entre los pesos totales de ambos montones. Aplicar sobre el ejemplo con **n**= 4 y **p**= (2, 1, 3, 4).

8.20. (EX M04) En el problema de la mochila 0/1 disponemos de dos mochilas, con capacidades M1 y M2. El objetivo es maximizar la suma de beneficios de los objetos transportados en ambas mochilas, respetando las capacidades de cada una. Resolver el problema mediante programación dinámica, definiendo la ecuación recurrente, las tablas usadas y el algoritmo para rellenarlas.

Datos del problema: **n** objetos, **M1** capacidad de la mochila 1, **M2** capacidad de la mochila 2, **p**= ($\mathbf{p_1}$, $\mathbf{p_2}$, ..., $\mathbf{p_n}$) pesos de los objetos, **b**= ($\mathbf{b_1}$, $\mathbf{b_2}$, ..., $\mathbf{b_n}$) beneficios de los objetos.

8.21. (EX S04) Considerar el problema de la mochila 0/1. En este ejercicio estamos interesados en calcular el número de formas distintas de meter o no los objetos en la

mochila, pero respetando la capacidad máxima de la mochila. Por ejemplo, si todos los **n** objetos cupieran en la mochila, existirían $\mathbf{2^n}$ formas posibles. Pero en general, si no caben todos, habrán muchas menos. Resolver mediante programación dinámica el problema de calcular el número de formas distintas de completar total o parcialmente la mochila. Datos del problema: **n** objetos, **M** capacidad de la mochila, $\mathbf{p}= (\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_n})$ pesos de los objetos).

8.22. (EX) Tenemos una secuencia de palabras, sacadas del diccionario de la RAE, que denotaremos por $\mathbf{p_1}$, $\mathbf{p_2}$, ..., $\mathbf{p_M}$. Queremos seleccionar subsecuencias de palabras, en el mismo orden pero no necesariamente consecutivas, de manera que cada palabra sea un prefijo de la siguiente. El objetivo es encontrar la subsecuencia más larga posible. Por ejemplo, si **p**= (*ca*, *p*, *d*, *pre*, *casa*, *de*, *prenda*, *precio*, *prendadora*, *decágono*), la solución sería (*p*, *pre*, *prenda*, *prendadora*). Resolver el problema usando programación dinámica. Mostrar la ejecución sobre el ejemplo anterior. Suponer que disponemos de una operación *Prefijo(cad1, cad2) : bool*, para conocer si una palabra es prefijo de otra.

8.23. (EX J05) Resolver el problema del ejercicio 7.23 mediante programación dinámica, definiendo la ecuación recurrente, con sus casos base, las tablas usadas y el algoritmo para rellenarlas. No es necesario mostrar la ejecución del ejemplo.

8.24. (EX J05) Nos vamos de compras al mercado. Tenemos **K** euros en el bolsillo y una lista de **m** productos que podemos comprar. Cada producto tiene un precio, $\mathbf{p_i}$ (que será siempre un número entero), y una utilidad, $\mathbf{u_i}$. De cada producto podemos comprar como máximo 3 unidades. Además, tenemos una oferta según la cual la segunda unidad nos cuesta 1 euro menos, y la tercera 2 euros menos. Queremos elegir los productos a comprar, maximizando la utilidad de los productos comprados. Resolver el problema por programación dinámica, indicando la ecuación recurrente, con sus casos base, las tablas, el algoritmo para rellenarlas y la forma de componer la solución a partir de las tablas.

8.25. (EX) En cierto teclado, asignamos las teclas especiales a cadenas de más o menos tamaño, (por ejemplo: F1→abc, F2→ca, F3→ab, F4→bcc). Dada otra cadena más larga (por ejemplo: abccabcabb) se quiere escribirla pulsando el menor número de teclas. Se pueden usar las teclas normales o las especiales. Explicar cómo se resuelve el problema por programación dinámica: hay que dar la fórmula de recursión, el valor de los casos base, y las tablas que se usan en la solución del problema. Explicar el funcionamiento con el ejemplo dado. **Sugerencia**: estudiar la descomposición recurrente de una función **MinTeclas (i: entero): entero**, que devuelve el menor número de pulsaciones para escribir los **i** primeros caracteres de la cadena larga.

8.26. (EX) Considerar que tenemos un sistema monetario con monedas de tres tipos, con valores: 2, 3 y 4. Queremos dar la cantidad 9, usando el menor número posible de monedas. Aplicar el algoritmo de programación dinámica visto en clase sobre el ejemplo. Deducir razonadamente la forma de la ecuación de recurrencia, con sus casos base. Mostrar la tabla resultante para este caso concreto. A partir de ella, obtener el número de monedas de cada tipo, explicando el proceso. ¿Cómo se puede saber si la solución óptima es única?

8.27. (EX) Suponer que en el problema del corral del ejercicio 7.26 no se pueden partir los tablones. Queremos resolver la cuestión de saber si es posible construir un corral de forma cuadrada, con lados de longitud **K**. Resolver el problema por programación dinámica. Dar la fórmula recursiva del problema, con sus casos base, e indicar las tablas que usa el algoritmo. **Sugerencia:** dar una definición recursiva para el problema **Corral (m, p1, p2, p3, p4): booleano**, que significa: "comprobar si se pueden construir cuatro paredes de longitudes **p1**, **p2**, **p3** y **p4**, pudiendo utilizar los **m** primeros tablones".

8.28. (EX J06) Resolver el problema del juego de la oca del ejercicio 7.27 de forma óptima por programación dinámica. Dar la fórmula recursiva del problema, con sus casos base, e indicar las tablas que usa el algoritmo. Mostar la ejecución sobre el ejemplo del ejercicio 7.27. A partir de las tablas, indicar para el ejemplo cómo se obtiene la secuencia de movimientos óptima.