Análisis y diseño de algoritmos

Semana 10

Logro de la sesión

Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos con programación dinámica

Agenda

- Programación dinámica – El problema de la mochila / comparación de textos
- Programación dinámica – Números de Fibonacci

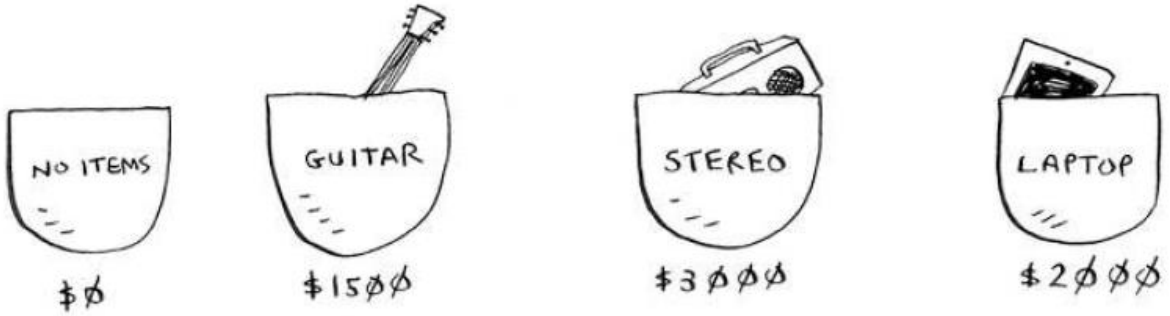
Programación dinámica – El problema de la mochila

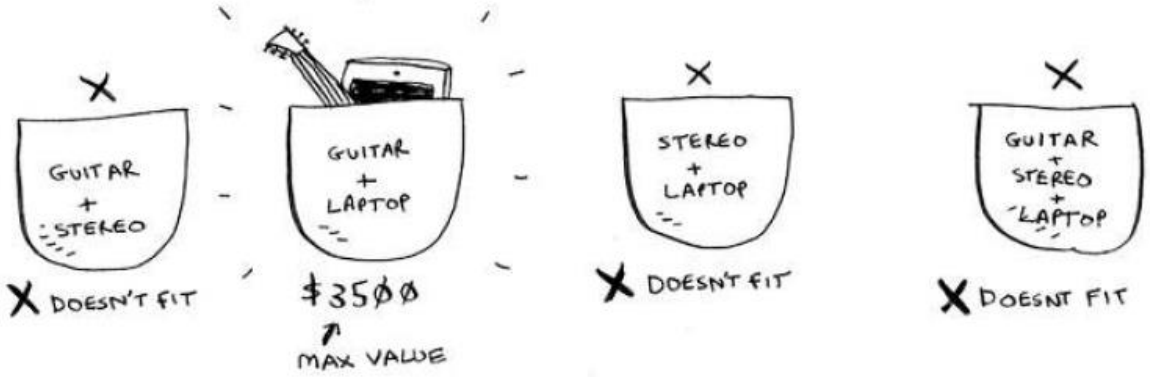
Reevaluemos el problema de la mochila. Eres un ladrón con una mochila que puede cargar 4 libras de bienes. Tienes 3 objetos que puedes llevar en la mochila

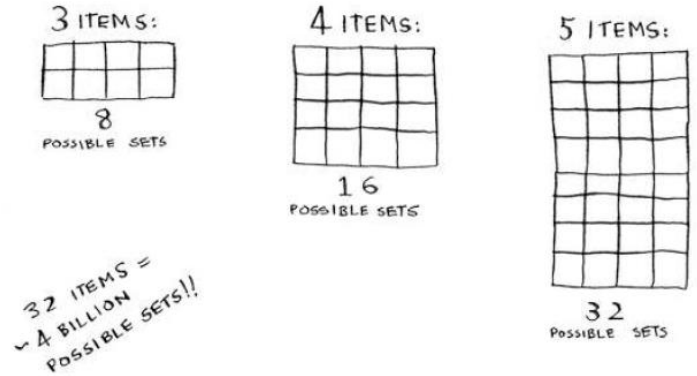
¿Cuál deberías robar para maximizar el valor de los bienes robados?

La solución simple:

1. Intentas cada posible conjunto de objetos y escoges el que otorgue el mayor valor

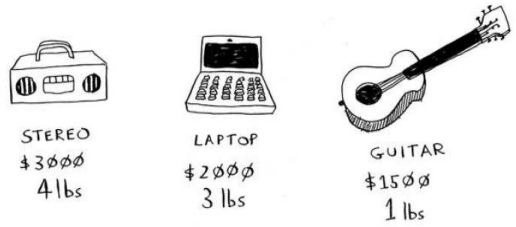






Esto funciona pero realmente es lento. Para 3 elementos (ítems) tienes que calcular 8 posibles conjuntos, para 4 elementos necesitas 16 conjuntos. Con cada objeto que agregues, la cantidad de conjuntos se duplica. Este algoritmo lleva $O(2^n)$ que es muy, pero muy lento.

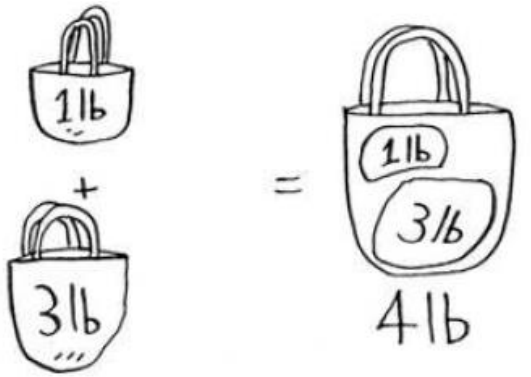
Programación dinámica – El problema de la mochila



Entonces, ¿Cómo calculamos la solución óptima?

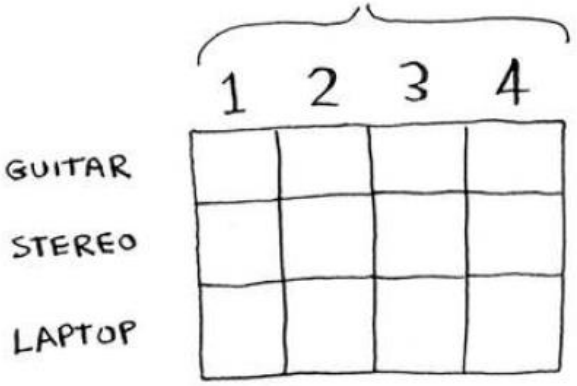
¡Con programación dinámica!, esta comienza por resolver subproblemas y va construyendo la solución hasta resolver el problema mayor

Para el problema de la mochila, comienzas resolviendo el problema con mochilas más pequeñas (o submochilas) y luego vas trabajando hasta que resuelves el problema original

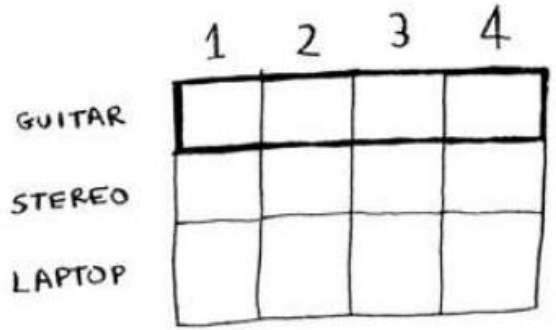


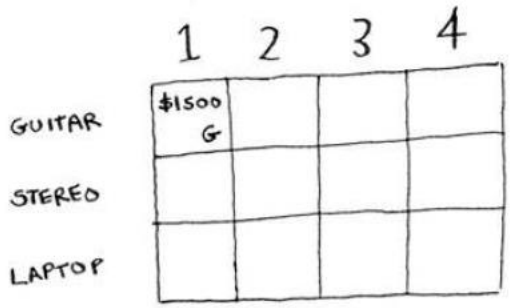
Cada algoritmo de programación dinámica comienza con una matriz, aquí tienes la matriz del problema de la mochila

Las columnas son pesos de mochilas de 1 a 4 libras



Una fila por cada elemento a escoger



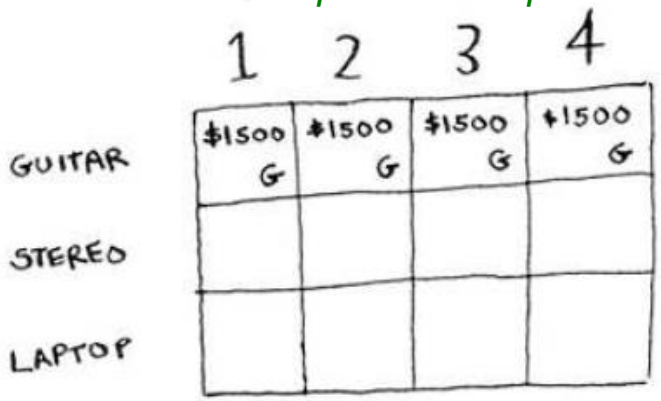


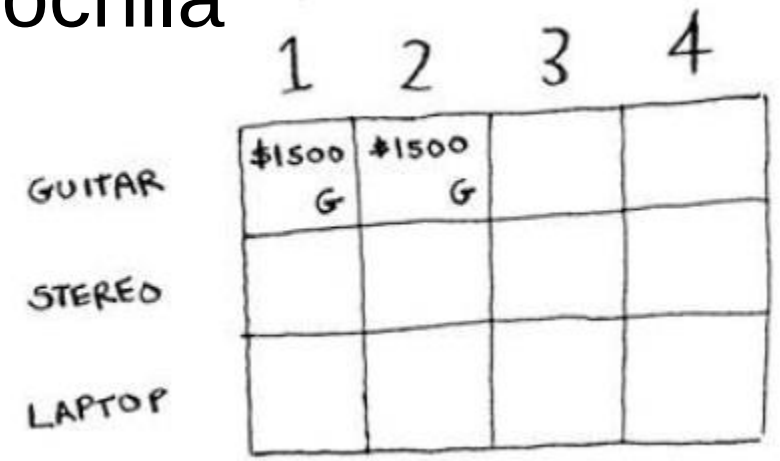
La primera fila corresponde a la guitarra, lo que significa que estás intentando meter la guitarra en la mochila. En cada celda, hay una decisión simple: ¿robas la guitarra o no? La primera celda tiene una mochila de 1 libra de capacidad. La guitarra también pesa 1 libra, lo que significa que está en la mochila. Entonces, el valor de esta celda es de $ 1,500 y contiene una guitarra.

Programación dinámica – El problema de la mochila

De manera similar, cada celda en la matriz contendrá una lista con los objetos que caben en ese punto. Veamos la siguiente celda, aquí tienes una mochila con capacidad de 2 lb. Bueno, la guitarra definitivamente cabe aquí también. Lo mismo se cumple para el resto de la fila.

Recuerda que esta es la primera fila, por lo cual sólo puedes escoger a la guitarra por ahora. Estás asumiendo que los otros dos objetos no se encuentran disponibles por ahora. La matriz se vería así

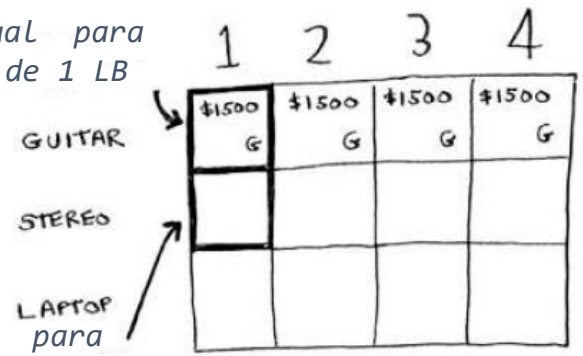




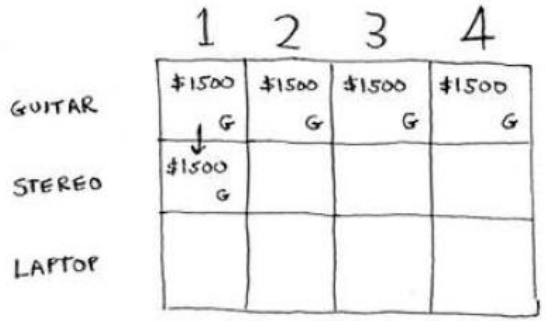
Nuestro mejor estimado actual que debería robar el ladrón: LA GUITARRA DE $1500

Continuemos con la siguiente fila, que corresponde al estereo. Ahora que te encuentras en la segunda fila puedes escoger entre la guitarra y el estéreo. En cada fila puedes usar el objeto de dicha fila o los objetos de las filas que se encuentren por encima de ella

Máximo actual para una mochila de 1 LB



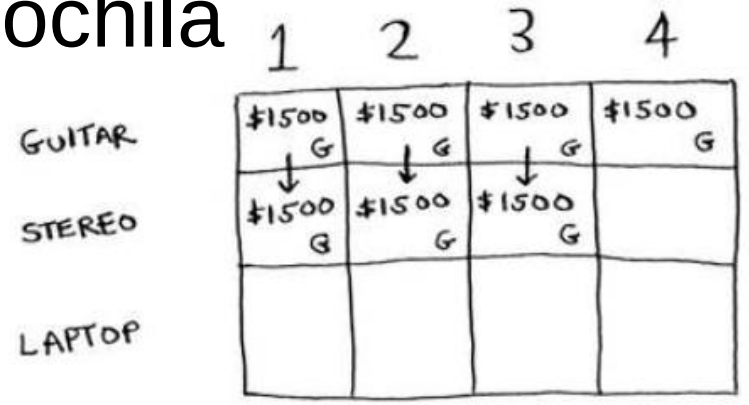
Nuevo máximo para una mochila de 1 LB



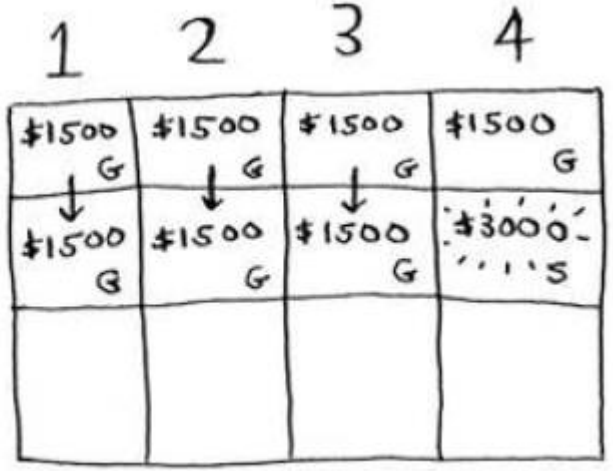
Comencemos con la primera celda, una mochila con 1 lb de capacidad. El mayor valor actual que puedes llevar en una mochila de 1 libra es de $1,500. ¿Deberías robar el estéreo o no? Con una mochila de capacidad de 1 lb. ¿cabe el estéreo? ¡No, es demasiado pesado! Debido a que no puedes cargar el estéreo, se mantiene como máximo valor $1,500, la suposición actual para mochilas de 1 lb.

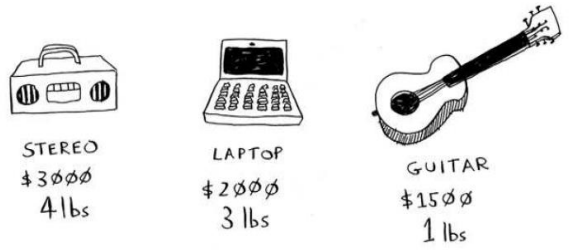
Programación dinámica – El problema de la mochila

Pasa lo mismo con las siguientes dos celdas. Estas mochilas tienen capacidad de 2 y 3 libras. El viejo máximo valor para ambas era $1500. El estéreo aún no cabe, por tanto, tus suposiciones se mantienen sin cambio.

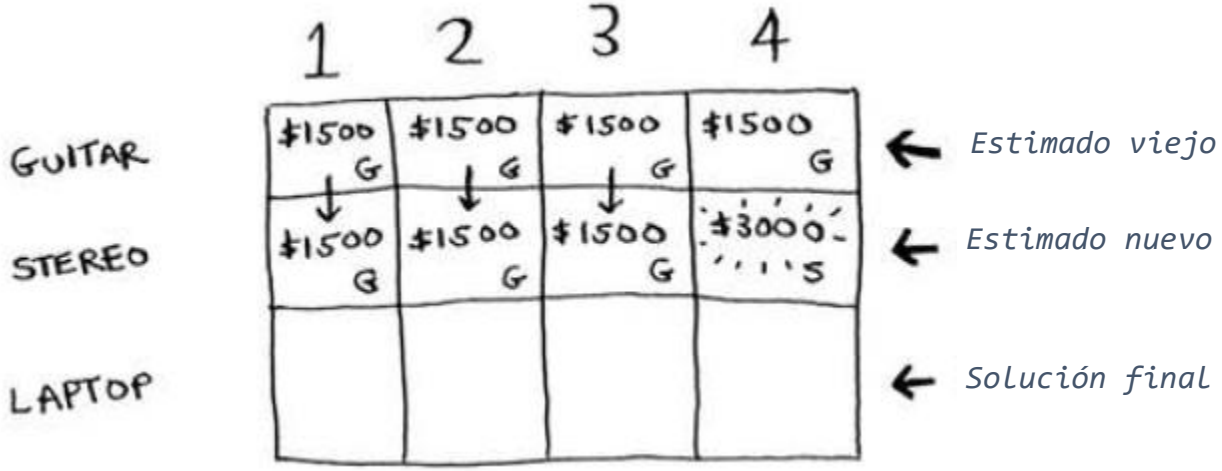


¿Qué pasaría con una mochila de 4 lb? ¡Ajá! ¡El estéreo finalmente cabe! El viejo valor era $1500, pero si llevas e estéreo en su lugar, el valor es de $ 3000 . Llevemos entonces el estéreo

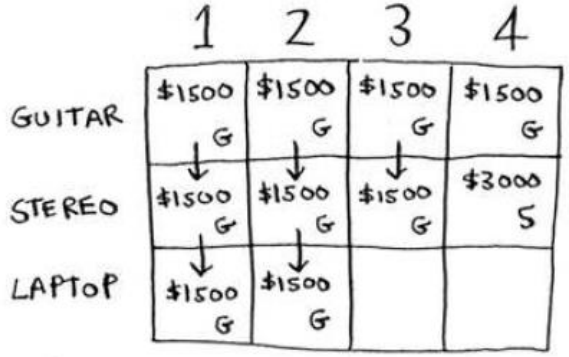




!Acabas de actualizar tu estimado! Si tienes una mochila de 4 lb, puedes llevar al menos objetos por un valor de $3000 en ella. Puedes ver en la matriz cómo has actualizado incrementalmente tu estimado.



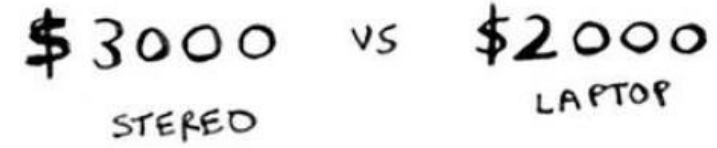
¡Hagamos lo mismo con la laptop! El peso de la laptop es 3 lb, por tanto, no cabe en las mochilas de 1 o 2 libras. El estimado para estas celdas se mantiene en $1500



Programación dinámica – El problema de la mochila

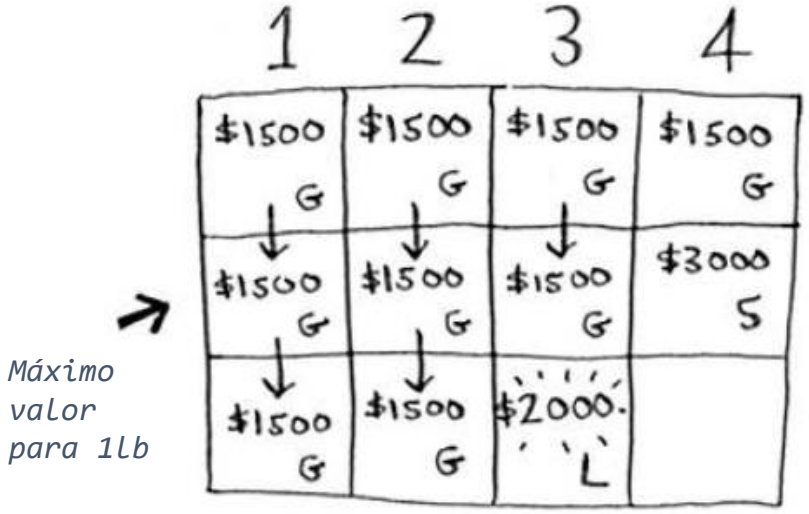
En 3lb, el viejo estimado era $1500, pero ahora puedes elegir la laptop que aporta $2000 ¡Entonces el nuevo estimado para esta celda es $2000!

En 4lb, el estimado actual es $3000 Puedes poner la laptop en la mochila, pero sólo se obtendría $2000

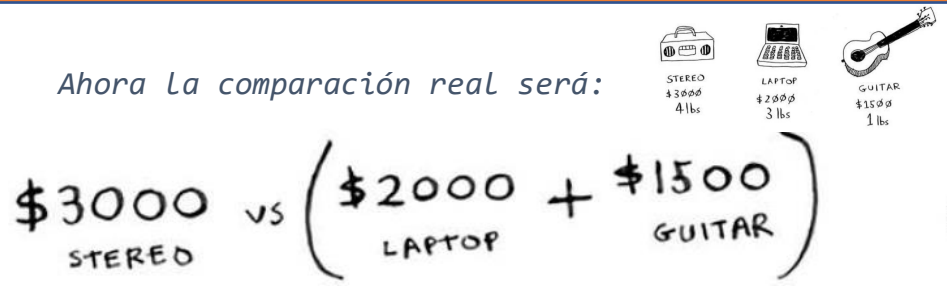


Eso no es bueno como el estimado anterior, pero.. La laptop pesa solamente 3 lb, así que tienes 1lb disponible. Podrías poner algo en el espacio restante

¿Cuál es el mayor valor que puedes cargar con 1lb de capacidad? Bueno has estado calculándolo todo este tiempo. Puedes poner la guitarra en el espacio restante y eso suma unos $1500.

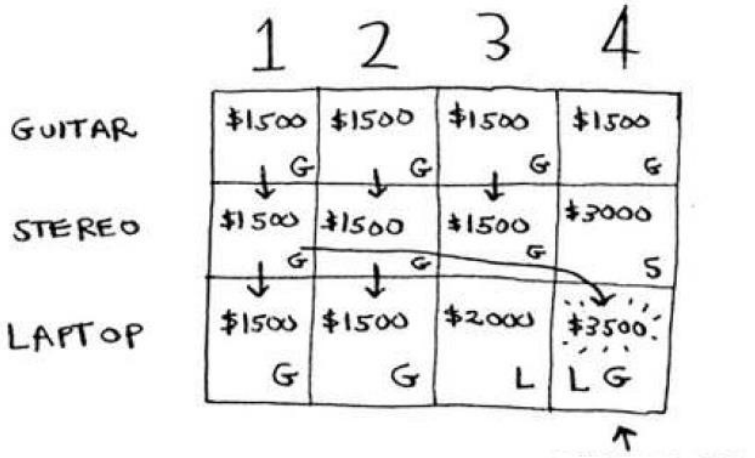


Ahora la comparación real será:



Es posible que se haya preguntado por qué estaba calculando valores máximos para mochilas más pequeñas. ¡Espero que ahora tenga sentido! Cuando tenga espacio, puede usar las respuestas de esos subproblemas para definir qué poner en ese espacio. Es mejor llevarse la laptop + la guitarra por $ 3500.

La matriz final se vé así

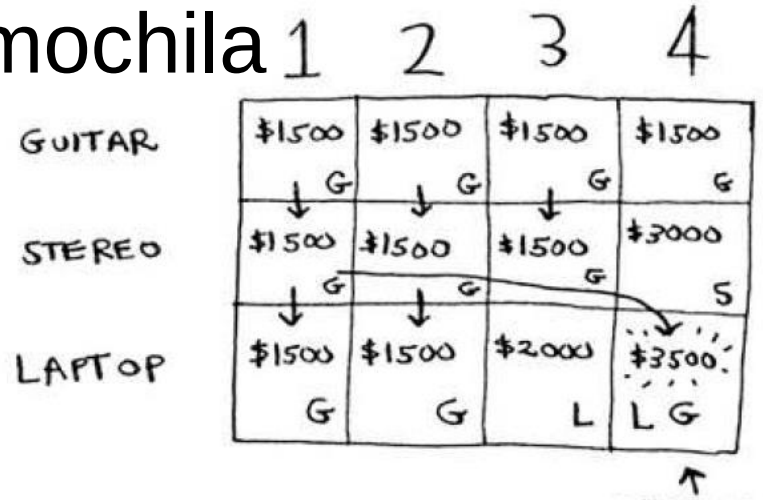


Programación dinámica – El problema de la mochila

Podríamos calcular el valor de cada celda con esta fórmula

FILA COLUMNA

CELDA[I] [J] = MÁX. DE { 1. MÁXIMO ANTERIOR (VALOR EN LA CELDA [I -1][J])
VS.
2. VALOR DEL ELEMENTO ACTUAL + VALOR DEL ESPACIO RESTANTE
CELDA[I - 1][J - PESO DEL ELEMENTO] }

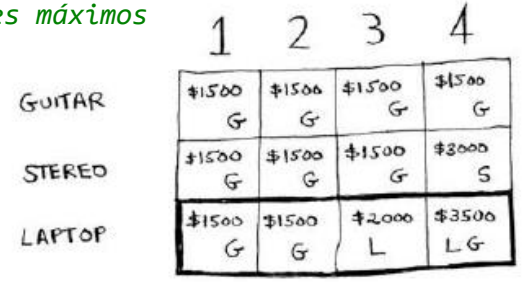


Se puede emplear esta fórmula con cada casilla de la matriz y debes terminar con la misma matriz.

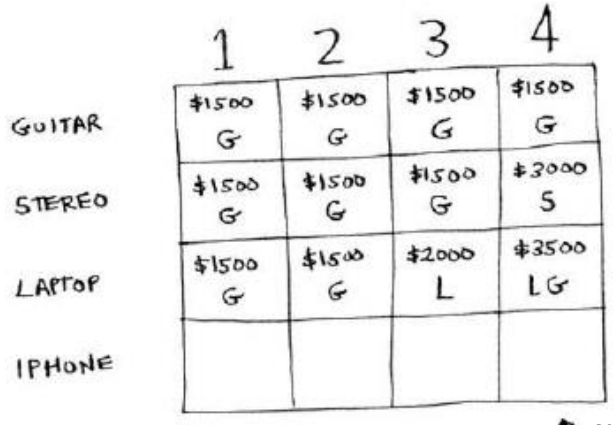
De esta forma combinas las soluciones a dos subproblemas para resolver un problema de mayor tamaño

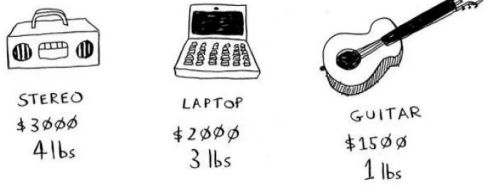
¿Qué pasa si añades un objeto?

Supón que tienes un cuarto objeto que robar que no habías descubierto antes. Ahora puedes robar un iPhone.
¿Tienes que recalcular todo para tener en cuenta este objeto? No. Recuerda que en programación dinámica vas construyendo progresivamente un estimado. Hasta ahora estos son los valores máximos



Debemos añadir una nueva fila para el iPhone





Nueva respuesta

Programación dinámica – El problema de la mochila

Comencemos con la primera celda. El iPhone cabe en la mochila de 1 lb. El viejo máximo era $1500, pero el iPhone vale $2000. Tomemos el iPhone en su lugar

	1	2	3	4
GUITAR	$1500 G	$1500 G	$1500 G	$1500 G
STEREO	$1500 G	$1500 G	$1500 G	$2000 S
LAPTOP	$1500 G	$1500 G	$2000 L	$3500 LG
IPHONE	$2000 I			

En la próxima celda, puedes incluir el iPhone y la guitarra

$1500 G	$1500 G	$1500 G	$1500 G
$1500 G	$1500 G	$1500 G	$3000 S
$1500 G	$1500 G	$2000 L	$3500 LG
$2000 I	$3500 IG		

Para la celda 3, no puedes mejorar el estimado del iPhone y la guitarra, así que la dejamos así.

$1500 G	$1500 G	$1500 G	$1500 G
$1500 G	$1500 G	$1500 G	$3000 S
$1500 G	$1500 G	$2000 L	$3500 LG
$2000 I	$3500 IG	$3500 IG	

Para la última celda, las cosas se vuelven interesantes. El máximo actual es $3500. Puedes robar el iPhone y tendrías 3lb de espacio restante

$3500 LAPTOP + GUITAR vs ($2000 IPHONE + ??? → 3 lbs disponible)

La matriz final se vé así

$1500 G	$1500 G	$1500 G	$1500 G
$1500 G	$1500 G	$1500 G	$3000 S
$1500 G	$1500 G	$2000 L	$3500 LG
$3500 I	$3500 IG	$3500 IG	$4000 IL

Nueva respuesta

Estas 3lb valen unos $2000.
Obtenemos entonces un nuevo máximo: $2000 del iPhone más $2000 de los anteriores subproblemas que suman $4000

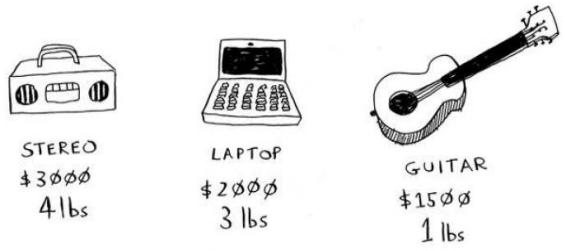
Ejercicio

Caso 1

Supongamos que puedes robar otro artículo: un reproductor de MP3. Su peso es de 1 lb y vale $1000 ¿Deberías robarlo?

Caso 2

Pregunta: ¿Qué pasa si cambias el orden de las filas? ¿Cambia la respuesta? Supón que llenaste las filas en el siguiente orden: Stereo, laptop, guitarra. ¿Cómo se ve la matriz?



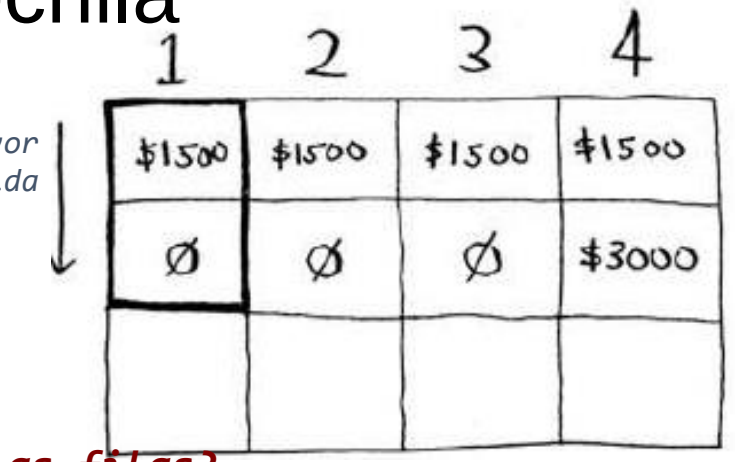
	1	2	3	4
Stereo	0	0	0	3000 - R
Laptop	0	0	2000-L	3000-R
Guitarra	1500-G	1500-G	2000-L	2000+1500 GL

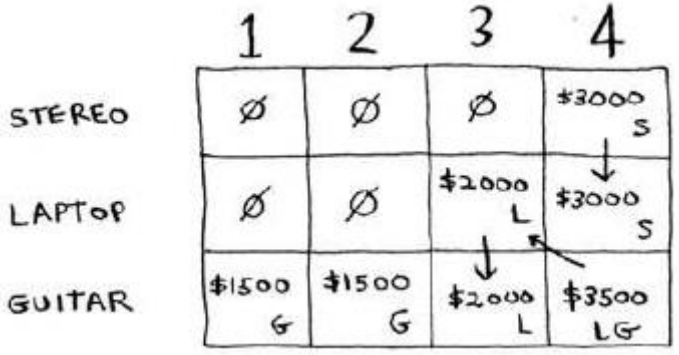
Programación dinámica – El problema de la mochila

Pregunta: ¿El valor de una columna disminuirá en algún momento? ¿Es esto posible?

La respuesta: NO. En cada iteración, estas guardando el actual mejor estimado. ¡El estimado nunca será peor que antes!

El valor mayor disminuye a medida que avanzamos





Pregunta: ¿Qué pasa si cambias el orden de las filas?

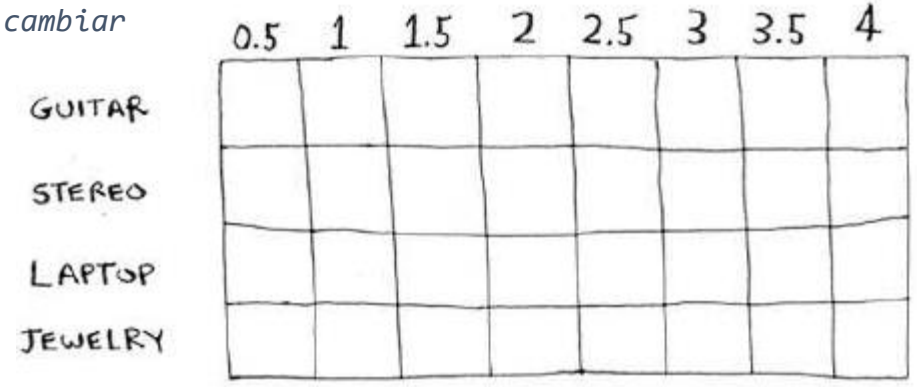
¿Cambia la respuesta? Supón que llenaste las filas en el siguiente orden: estéreo, laptop, guitarra. ¿Cómo se ve la matriz?

Pregunta: ¿Puedes llenar la matriz en el sentido de las columnas, en lugar de en el sentido de las filas?

En este problema no hace ninguna diferencia, podría hacer diferencia en otros problemas

¿Qué pasa si añades un elemento menor? Supón que se puede robar un collar. Pesa 0.5 libras y vale $ 1,000. Hasta ahora, la matriz asume que todos los pesos son números enteros. Ahora decides robar el collar. Tienes 3,5 libras de sobra. ¿Cuál es el valor máximo que puede tener en 3,5 libras? ¡No lo sabes! Solo se calculó valores para mochilas de 1 lb, 2 lb, 3 lb y 4 lb. Necesita saber el valor de una mochila de 3,5 libras.

Debido al collar, tienes que tener en cuenta una granularidad más fina, por lo tanto , la matriz tiene que cambiar



Programación dinámica – El problema de la mochila con fracciones de objetos

Pregunta: ¿Puedes robar fracciones de un objeto?

Suponga que es un ladrón en un mercado de víveres. Puedes robar bolsas de lentejas y arroz. Si una bolsa entera no cabe en la mochila, puedes abrirla y tomar tanto como puedas cargar. Así que ahora no es todo o nada, puedes escoger una fracción de un artículo. ¿Cómo resuelves este caso usando programación dinámica?

Respuesta: No es posible. Con una solución de programación dinámica tienes que tomar el objeto entero o no tomarlo. No hay forma de decidir qué parte del objeto deberías tomar

¡Pero este caso también se resuelve fácilmente usando un algoritmo voraz! Primero, tome todo lo que pueda del artículo más valioso. Cuando se acabe, toma todo lo que puedas del siguiente artículo más valioso, y así sucesivamente. Por ejemplo, suponga que tiene estos elementos para elegir.

La quinoa es más costosa. Entonces, llevas toda la quinoa que puedas cargar. Si eso llena tu mochila, es lo mejor que puedas hacer.
Si la quinoa se acaba y aún tienes espacio en tu mochila, continuas con el siguiente producto más caro y así sucesivamente

Programación dinámica – El problema de la mochila código 1

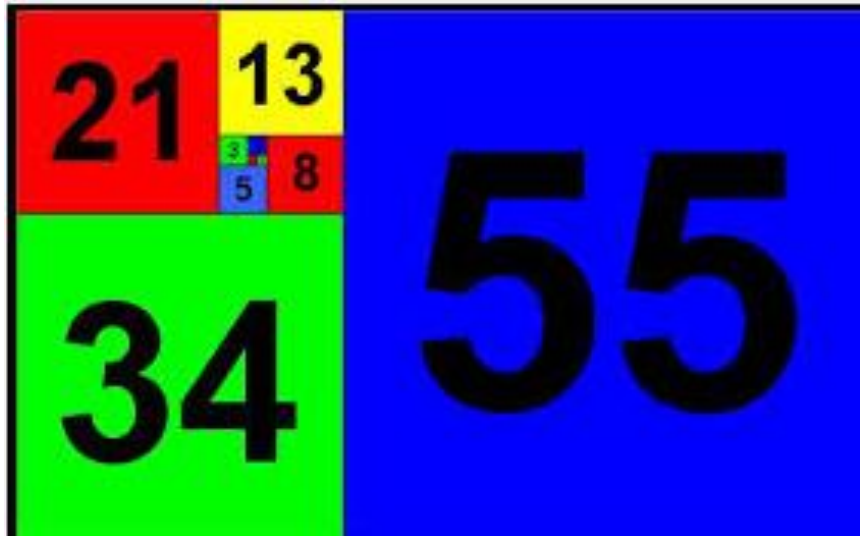
```
#include <cstdlib>
#include <iostream>
#define NUM_OBJETOS 3
#define PESO_MAX 4
using namespace std;
int M[NUM_OBJETOS][PESO_MAX+1];
int W[] = {1, 3, 4}; // Vector de pesos de los objetos
int V[] = {1500, 2000, 3000}; // Vector de valores de los objetos
void imprimeMatriz(){
  for(int j=0; j<=PESO_MAX; j++)
    cout<< j << " ";
  cout<< endl << endl;
  for(int i=0; i<NUM_OBJETOS; i++){
    for(int j=0; j<=PESO_MAX; j++){cout<< M[i][j] << " ";};
    cout << endl;
    };
  return;
}
int main(int argc, char *argv[]){
  for(int i=0; i<NUM_OBJETOS; i++)
    M[i][0] = 0;
  for(int j=1; j<=PESO_MAX; j++)
    M[0][j]= 1500;
  for(int i=1; i<NUM_OBJETOS; i++){
    for(int j=1; j<=PESO_MAX; j++){
      if( W[i] > j)
        M[i][j] = M[i-1][j];
      else
        M[i][j] = max( M[i-1][j], V[i] + M[i-1][j-W[i]] );
      };
    };
  imprimeMatriz();
  system("PAUSE");
  return EXIT_SUCCESS;
  }
```

Programación dinámica: Números de Fibonacci

- Divida un gran problema en problemas más pequeños
  …
- ¿Suena familiar?
- ¿Recursividad?
  N! = 1 for N == 0
  N! = N * (N - 1)! for N > 0

Programación dinámica: Números de Fibonacci

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 114, …
- $F_1 = 1$
- $F_2 = 1$
- $F_N = F_{N-1} + F_{N-2}$
- ¿Solución recursiva?



Programación dinámica: Números de Fibonacci

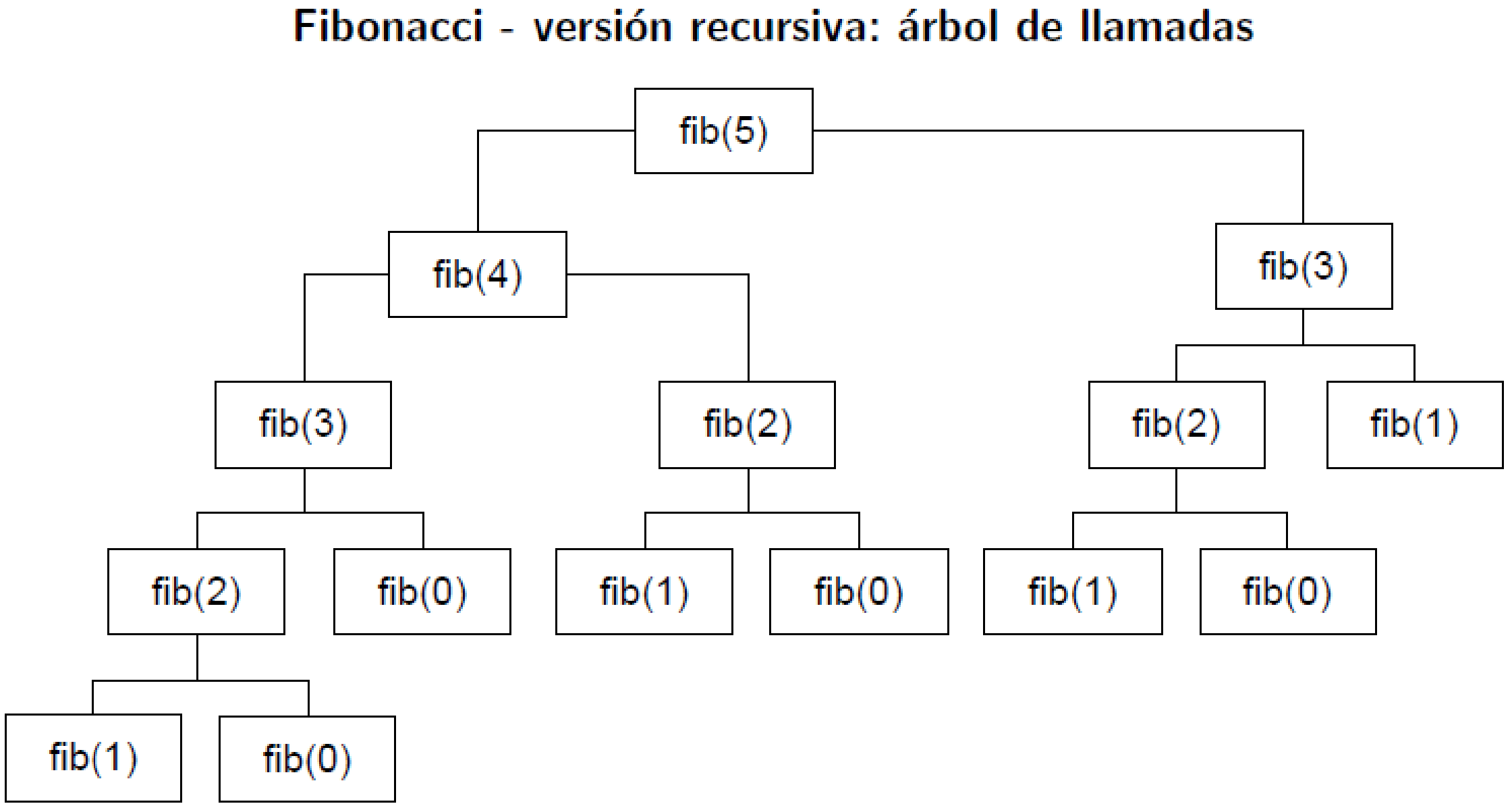
- Método recursivo ingenuo

```
// pre: n > 0
// post: return the nth Fibonnaci number
public int fib(int n) {
    if(n <= 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

- ¿Orden de este método?

A. O(1) B. O(log N) C. O(N) D. $O(N^2)$ E. $O(2^N)$

Programación dinámica: Números de Fibonacci



Programación dinámica: Números de Fibonacci

- Consideremos el problema de obtener el término n de la serie de Fibonacci, que se define recursivamente como:

$$F_n = F_{n-1} + F_{n-2} \qquad \text{con } F_0 = F_1 = 1$$

- La solución recursiva es inmediata:

```
def fib_rec(n):
    if n<2:
        return 1
    else:
        return fib_rec(n-1)+fib_rec(n-2)
```

- Pero esta solución produce (y resuelve) los mismos subproblemas muchas veces, como ya vimos en el tema de *Divide y Vencerás*, y como consecuencia el coste temporal es exponencial.
- El coste espacial de esta solución será de orden lineal, ya que el número de llamadas almacenadas en la pila del sistema por la rama izquierda del árbol llegará a ser exactamente n (por la otra rama serán $n/2$).

Programación dinámica: Números de Fibonacci

- Para no repetir cálculos, podemos almacenarlos en una tabla:

```
# t se define como un diccionario y se inicializa t={0:1, 1:1}
def fib_rec_mem(n):
    global t
    if not n in t:
        t[n]=fib_rec_mem(n-1)+fib_rec_mem(n-2)
    return t[n]
```

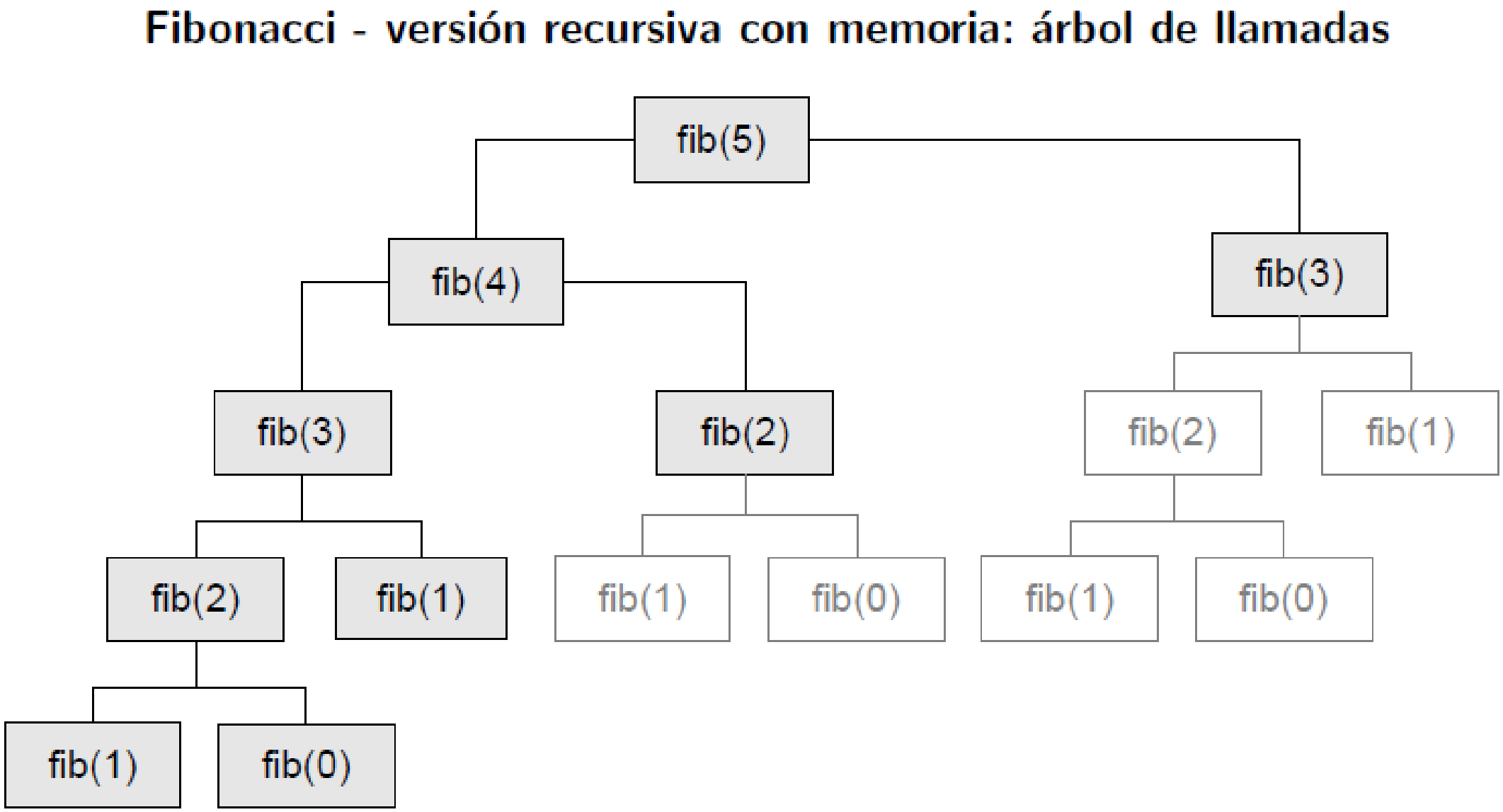
- Esta solución calculará cada término una sola vez, de modo que el coste temporal será lineal. Nótese que el coste espacial también será lineal.

Sin embargo, la versión anterior sigue siendo recursiva y se le llama top-down dynamic programming. Los algoritmos de programación dinámica son iterativos y operan de abajo arriba. En este caso, a partir del caso base iremos calculando y almacenando los términos sucesivos de la serie:

```
def fib_ite(n):
    t={0:1, 1:1}
    for i in range(2,n+1):
        t[i]=t[i-1]+t[i-2]
    return t[n]
```

- Los costes temporal y espacial de esta solución siguen siendo lineales.

Programación dinámica: Números de Fibonacci



Programación dinámica: Números de Fibonacci

- Sin embargo, como el término F_n depende sólo de F_{n-1} y F_{n-2}, no es necesario almacenar *toda* la tabla, sino tan sólo los dos últimos términos calculados:

```
def fib_ite(n):
    f2=1  # f2 es el termino F_{n-2}
    f1=1  # f1 es el termino F_{n-1}
    for i in range(2,n+1):
        f=f1+f2  # F_{n} = F_{n-1} + F_{n-2}
        f2=f1
        f1=f
    return f
```

- De esta forma, hemos llegado a un algoritmo de programación dinámica iterativo, de coste temporal lineal y coste espacial constante.

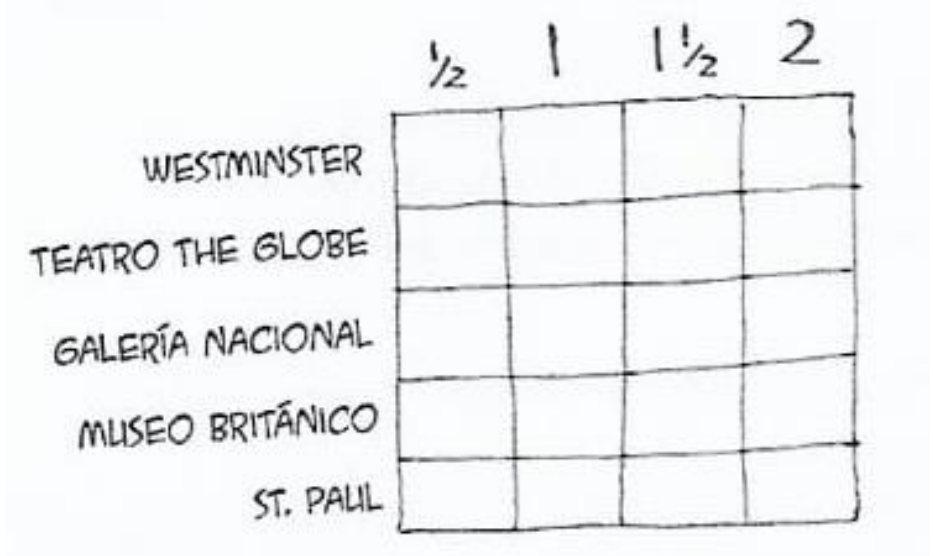
Programación dinámica – Optimizando tu itinerario de viaje

Supón que viajas a Londres a una agradable vacaciones. Tienes dos días allí y muchas cosas que hacer. No puedes visitar todos los lugares, así que haces una lista

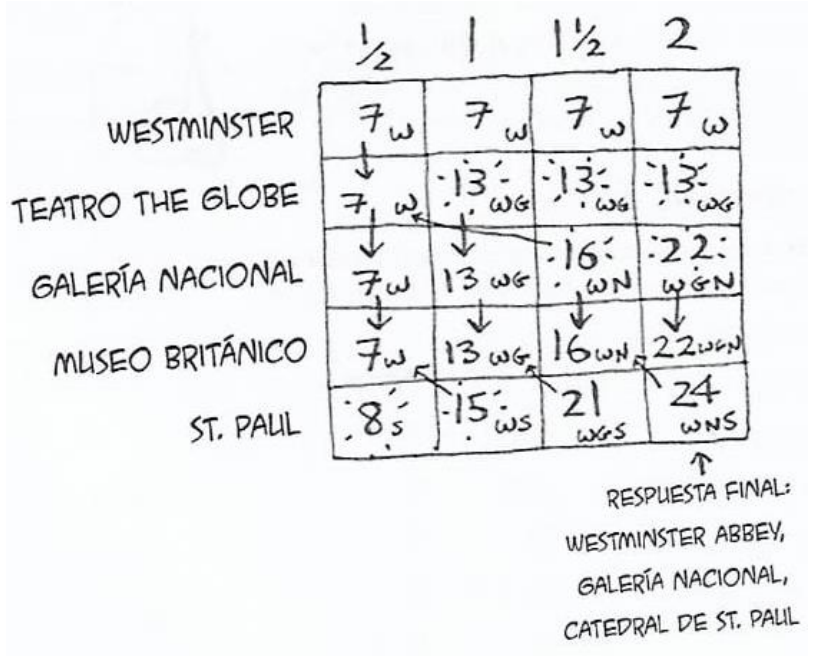
Por cada lugar que quieres ver, escribes cuánto tiempo tomaría la visita y evalúa cuánto quieres conocerlo. ¿Puedes encontrar los lugares que deberías visitar, basándose en esta lista?
¡Es el problema de la mochila de nuevo! En lugar de una mochila, tienes una cantidad limitada de tiempo. Y en lugar de estéreos y laptops, tienes una lista de lugares a los que quieres ir.

ATRACCIÓN	TIEMPO	EVALUACIÓN
WESTMINSTER	½ DÍA	7
TEATRO THEGLOBE	½ DÍA	6
GALERÍA NACIONAL	1 DÍA	9
MUSEO BRITÁNICO	2 DÍAS	9
ST. PAUL	½ DÍA	8

Así es como se ve la matriz:



Esta sería el caso resuelto



Ejercicio

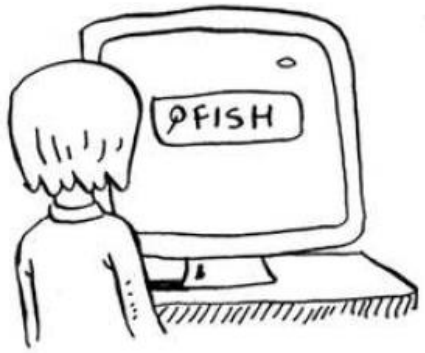
Caso 1

Supongamos que vas a acampar. Tienes una mochila que contiene 6lb y puede guardar los siguientes artículos. Cada uno tiene un valor y cuanto mayor sea el valor, más importante es el elemento:

- Agua, 3 libras, 10
- Libro, 1 libra, 3
- Comida, 2 libras, 9
- Chaqueta, 2 libras, 5
- Cámara, 1 libra, 6

¿Que llevaría al campamento?

Programación dinámica – Mayor subcadena común



Supón que administras el sitio dictionary.com. Alguien escribe una palabra y le muestras su definición

Pero si alguien se equivoca en una palabra, quieres ser capaz de adivinar qué palabra quería utilizar. Alex está buscando la palabra "fish" pero accidentalmente escribió hish. Esta no está en el diccionario, pero tienes una lista de palabras similares.

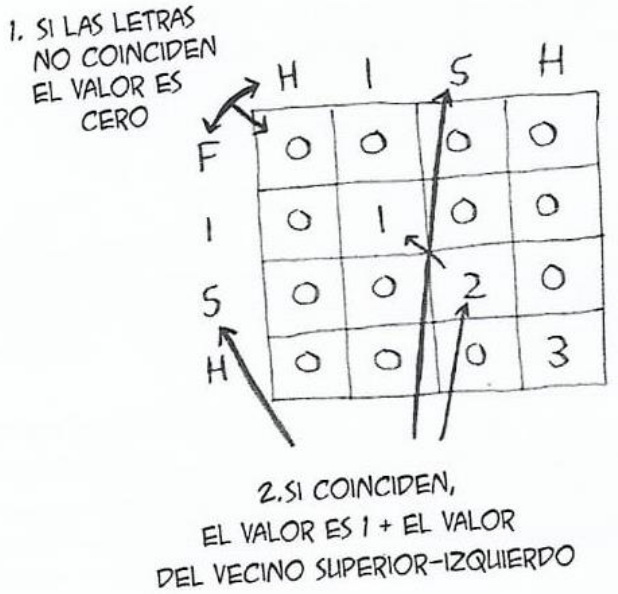
Este es un ejemplo de prueba, así que limitaremos la lista a dos palabras, en realidad la lista tendría miles de palabras

¿Cómo sería la matriz en este problema? Tienes que responder:
- ¿Cuáles son los valores de las celdas?
- ¿Cómo divides el problema en subproblemas?
- ¿Cuáles son los ejes de la matriz?

En programación dinámica el objetivo es maximizar/minimizar (optimizar) alguna medida. En este caso, quieres buscar la subcadena más larga que dos palabras tienen en común ¿Qué subcadena tienen hish y fish en común? ¿Qué tal hish y vista?, Los valores de las celdas es lo que estás intentando optimizar, en este caso un número, el tamaño de la subcadena más larga que tienen en común las dos cadenas

		H	I	S	H
	F				
	I				
	S				
	H				

		H	I	S	H
	F	0	0	0	0
	I	0	1	0	0
	S	0	0	2	0
	H	0	0	0	3




```
if palabra_a[i] == palabra_b[j]:
  celda[i][j]=celda[i-1][j-1]+1
else:
  celda[i][j]=0
```

Ejercicio

Caso 1

Completa la matriz para las palabras hish vs vista

Caso 2

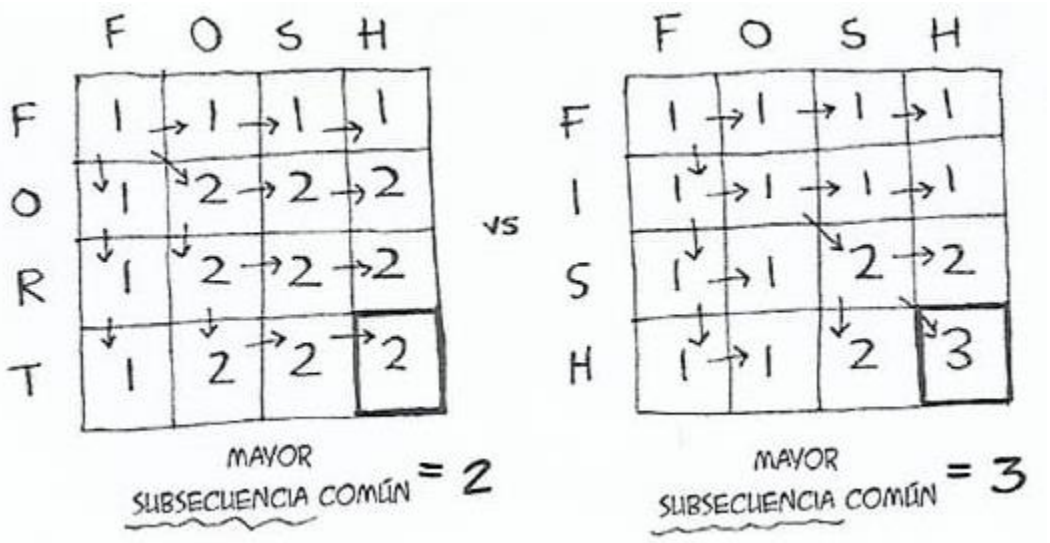
Supón que Alex accidentalmente busco fosh. Usando la fórmula de la mayor subcadena en común ¿Qué palabra quiso decir fish o fort? ¿Tenemos solución?

Programación dinámica – Mayor subsecuencia común

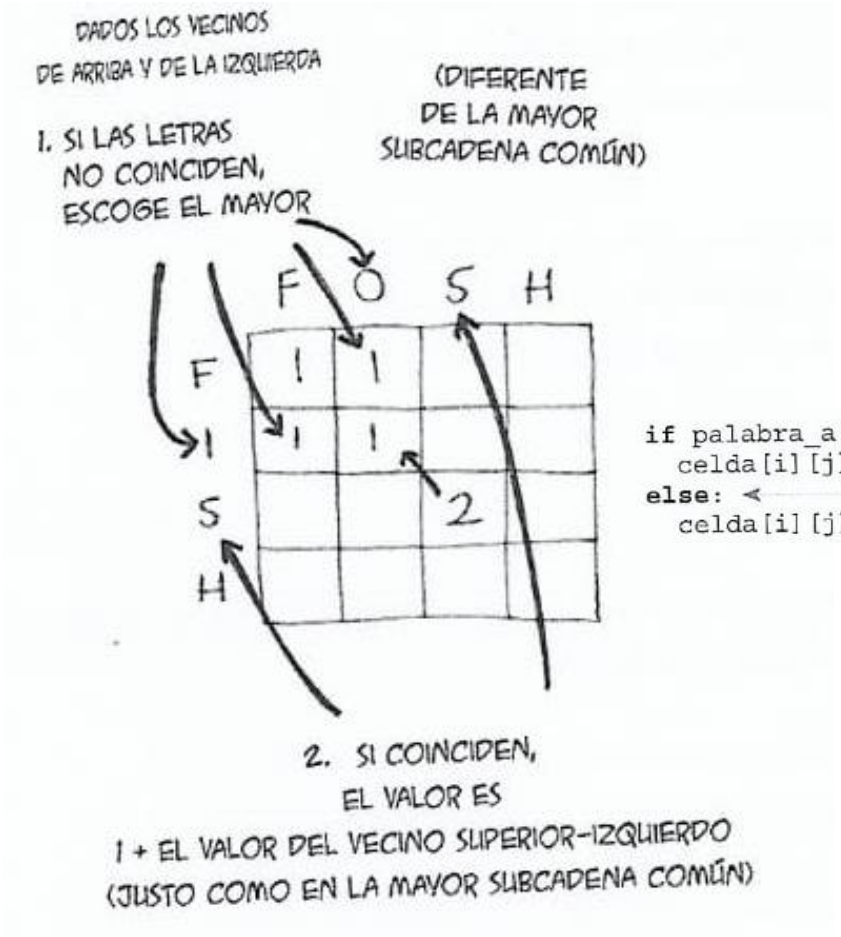
Para el último ejercicio, estamos comparando la subcadena en común más larga, pero lo que realmente necesitas comparar es la subsecuencia en común más larga:

El número de letras en una secuencia que ambas palabras tienen en común

Aquí tienes la matriz final



La fórmula para llenar la matriz es como sigue:



El pseudocódigo es el siguiente:

```
if palabra_a[i] == palabra_b[j]:      ← Las letras coinciden
  celda[i][j]=celda[i-1][j-1]+1
else:                                 ← Las letras no coinciden
  celda[i][j]=max(celda[i-1][j], celda[i][j-1])
```

Algunas conclusiones de la programación dinámica

Es útil cuando intentas optimizar algo dado cierta restricción. En el problema de la mochila tenías que maximizar el valor de los objetos bajo la limitación del tamaño de la mochila

Puedes usarlo cuando el problema se puede dividir en subproblemas discretos y estos no dependen entre sí

Cada solución de programación dinámica involucra una matriz

Los valores en las celdas son usualmente lo que intentas optimizar

Cada celda es un subproblema, por tanto, debes pensar en cómo dividir tu problema en subproblemas. Esto te ayudará a definir qué significa cada eje

¿Dónde se utiliza la programación dinámica?

Los biólogos utilizan la mayor subsecuencia común para encontrar similitudes en las hebras de ADN. Pueden emplear esta información para decir cuan similares son dos animales o dos enfermedades. La mayor subsecuencia común se está utilizando para buscar una cura para la esclerosis múltiple.

¿Alguna vez has utilizado diff (como en git diff)? Diff muestra las diferencias entre dos archivos, y utiliza programación dinámica para hacerlo.

Hemos hablado sobre similitud. La distancia de Levenshtein mide qué tan similares son dos cadenas y utiliza programación dinámica. La distancia de Levenshtein se usa para todo, desde la revisión ortográfica hasta determinar si un usuario ha subido datos con derechos de autor.

¿Alguna vez has utilizado una aplicación que haga ajuste de línea, como Microsoft Word? ¿Cómo define la aplicación donde cortar el texto para que la distancia de la línea se mantenga consistente? ¡Programación dinámica!

Codificación del problema de la mochila

- Item:

Número de item	Peso del artículo	Valor del item	Valor por unidad de Peso
1	1	6	6.0
2	2	11	5.5
3	4	1	0.25
4	4	12	3.0
5	6	19	3.167
6	7	12	1.714

- Peso de la mochila = 8

- Una solución golosa: tome el elemento de mayor valor por unidad de peso que se ajuste: (1, 6), (2, 11) y (4, 12)
- Valor total = 6 + 11 + 12 = 29
- Pregunta - ¿Es óptimo? A. No B. Sí

Mochila – Una solución recursiva

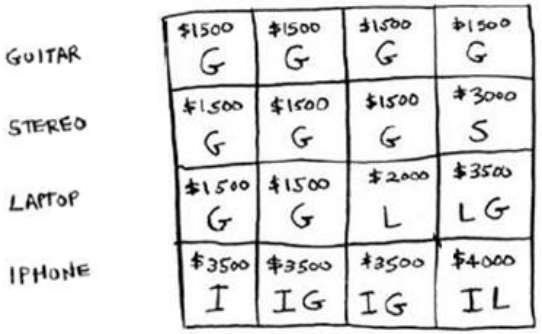
```
private static int knapsack(ArrayList<Item> items,
        int current, int capacity) {

    int result = 0;
    if(current < items.size()) {
        // don't use item
        int withoutItem = knapsack(items, current + 1, capacity);
        int withItem = 0;
        // if current item will fit, try it
        Item currentItem = items.get(current);
        if(currentItem.weight <= capacity) {
            withItem += currentItem.value;
            withItem += knapsack(items, current + 1,
                    capacity - currentItem.weight);
        }
        result = Math.max(withoutItem, withItem);
    }
    return result;
}
```

Mochila - Programación Dinámica

- El algoritmo recursivo comienza con la capacidad máxima y elige los items. Las opciones son:
  - Tomar el item si cabe
  - No tomar el item
- Programación Dinámica, comienza con problemas más simples
- Reduce el número de items disponibles.
- … Y reduce el límite de peso en la mochila
- Crea una matriz 2d de posibilidades.

Mochila - Función Óptima

- OptimalSolution(items, peso) es la mejor solución dado un subconjunto de items y un límite de peso
- 2 opciones:
- OptimalSolution no selecciona el i^{th} item
  - Selecciona la mejor solución para los items 1 a i - 1 con un límite de peso de w
- OptimalSolution selecciona el i^{th} item
  - Nuevo límite de peso = w - peso del i^{th} item
  - seleccione la mejor solución para los items 1 a i - 1 con el Nuevo límite de peso



Mochila Función Óptima

- OptimalSolution (items, límite de peso) =

  0 si 0 items

  OptimalSolution (items - 1, peso) si el peso del i^{th} item es mayor que el peso permitido
  $w_i > w$ (En otros , el i^{th} item no encaja)

  max of (OptimalSolution (items - 1, w),
  valor de i^{th} item + OptimalSolution (items - 1, $w - w_i$)

Mochila - Algoritmo

- Cree una matriz 2d para almacenar el valor de la mejor opción dado un subconjunto de items y posibles pesos

Número de item	Peso del item	Valor del item
1	1	6
2	2	11
3	4	1
4	4	12
5	6	19
6	7	12

- En nuestro ejemplo 0 a 6 items y límites de peso

  de 0 a 8

- Complete la tabla usando la función OptimalSolution

Algoritmo de la mochila

Dado N items y WeightLimit

Crear una Matriz M con N + 1 filas y WeightLimit + 1 columnas

```
For weight = 0 to WeightLimit
        M[0, w] = 0

For item = 1 to N
        for weight = 0 to WeightLimit
                if(weight of ith item > weight)
                        M[item, weight] = M[item - 1, weight]
                else
                        M[item, weight] = max of
                        M[item - 1, weight] Y
                        value of item + M[item - 1, weight - weight of item]
```

Mochila – Con Matriz

Dado N items y WeightLimit

Crear una Matriz M con N + 1 filas y WeightLimit + 1 columnas

```
For weight = 0 to WeightLimit
        M[0, w] = 0
For item = 1 to N
        for weight = 0 to WeightLimit
                if(weight of ith item > weight)
                        M[item, weight] = M[item - 1, weight]

                else
                        M[item, weight] = max of
                        M[item - 1, weight] Y
                        value of item + M[item - 1, weight - weight of item]
```

Item	Peso	Valor
1	1	6
2	2	11
3	4	1
4	4	12
5	6	19
6	7	12

items / capacidad	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{ 1 }									
{1, 2 }									
{1, 2, 3 }									
{1, 2, 3, 4 }									
{1, 2, 3, 4, 5 }									
{1, 2, 3, 4, 5, 6 }									

Mochila – Matriz Completa

items/peso	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{1 } [1, 6]	0	6	6	6	6	6	6	6	6
{1,2 } [2, 11]	0	6	11	17	17	17	17	17	17
{1, 2, 3 } [4, 1]	0	6	11	17	17	17	17	18	18
{1, 2, 3, 4 } [4, 12]	0	6	11	17	17	18	23	29	29
{1, 2, 3, 4, 5 } [6, 19]	0	6	11	17	17	18	23	29	30
{1, 2, 3, 4, 5, 6 } [7, 12]	0	6	11	17	17	18	23	29	30

item	Peso	Valor
1	1	6
2	2	11
3	4	1
4	4	12
5	6	19
6	7	12

Mochila - Items para llevar

artículos / peso	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{1 } [1, 6]	0	6	6	6	6	6	6	6	6
{1,2 } [2, 11]	0	6	11	17	17	17	17	17	17
{1, 2, 3 } [4, 1]	0	6	11	17	17	17	17	17	17
{1, 2, 3, 4 } [4, 12]	0	6	11	17	17	18	23	29	29
{1, 2, 3, 4, 5 } [6, 19]	0	6	11	17	17	18	23	29	30
{1, 2, 3, 4, 5, 6 } [7, 12]	0	6	11	17	17	18	23	29	30

Programación dinámica – El problema de la mochila código 2

```java
package Sesion11;
public class ProgDina {
    public static void mochilaProgDina(int W[], int V[], int M, int n) {
        int B[][] = new int[n + 1][M + 1];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= M; j++) {
                B[i][j] = 0;
            }
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= M; j++) {
                B[i][j] = B[i - 1][j];

                if ((j >= W[i - 1]) && (B[i][j] < B[i - 1][j - W[i - 1]] + V[i - 1])) {
                    B[i][j] = B[i - 1][j - W[i - 1]] + V[i - 1];
                }
                System.out.print(B[i][j] + " ");
            }
            System.out.print("\n");
        }
        System.out.println("Valor Máximo:\t" + B[n][M]);
        System.out.println("Items seleccionados: ");
        while (n != 0) {
            if (B[n][M] != B[n - 1][M]) {
                System.out.println("\tItem " + n + " con peso W = " + W[n - 1] + " y valor = " + V[n - 1]);
                M = M - W[n - 1];
            }
            n--;
        }
    }
}
```

```java
package Sesion11;
public class TestProgDina {
    public static void main(String[] args) {
        /*Items Peso y Valor*/
        //int W[] = new int[]{3, 4, 5, 9, 4};
        //int W[] = new int[]{12, 2, 1, 1, 4};
        int W[] = new int[]{1, 3, 4};
        //int V[] = new int[]{3, 4, 4, 10, 4};
        //int V[] = new int[]{4, 2, 1, 2, 10};
        int V[] = new int[]{1500, 2000, 3000};
        /*Máximo Peso*/
        //int M = 11;
        int M = 4;
        int n = V.length;
        /*Ejecutar el Algoritmo*/
        ProgDina.mochilaProgDina(W, V, M, n);
    }
}
```

Mochila con programación dinámica

```
public static int knapsack(ArrayList<Item> items, int maxCapacity) {
    final int ROWS = items.size() + 1;
    final int COLS = maxCapacity + 1;
    int[][] partialSolutions = new int[ROWS][COLS];

    for(int item = 1; item <= items.size(); item++) {
        for(int capacity = 0; capacity <= maxCapacity; capacity++) {
            Item currentItem = items.get(item - 1);
            int best = partialSolutions[item - 1][capacity];
            if(currentItem.weight <= capacity) {
                int withItem = currentItem.value;
                int capLeft = capacity - currentItem.weight;
                withItem += partialSolutions[item - 1][capLeft];
                if(withItem > best)
                    best = withItem;
            }
            partialSolutions[item][capacity] = best;
        }
    }
    return partialSolutions[ROWS - 1][COLS - 1];
```

Mochila con programación dinámica

- ¿Qué enfoque del problema de la mochila usa más memoria?

A. El enfoque recursivo

B. El enfoque de programación dinámica

C. Utilizan aproximadamente la misma cantidad de memoria

Programación dinámica: memoria que ahorra tiempo

- Algunos problemas se pueden dividir en partes, resolver cada parte por separado y combinar las soluciones. Eso es lo que hacemos de forma recursiva en la técnica conocida como *Divide y Vencerás* (DyV). En estos casos, los subproblemas generados son independientes y no hay esfuerzos redundantes.
- Sin embargo, ciertas soluciones recursivas producen los mismos subproblemas muchas veces y resultan altamente ineficientes, ya que deben resolver dichos subproblemas cada vez que se generan. Esto es lo que se conoce como **solapamiento de subproblemas** (p.e. Fibonacci recursivo).
- Manteniendo el esquema top-down de los algoritmos recursivos, es posible reducir drásticamente el esfuerzo computacional mediante lo que se conoce como **memorización**: cada vez que se ha de realizar un cálculo, primero se busca en una tabla; si está, se utiliza; si no, se realiza el cálculo y se almacena el resultado en la tabla.
- Esta pequeña modificación implica usar memoria auxiliar para almacenar los resultados ya calculados, lo cual, dependiendo del tamaño del problema, podría ser inviable, o incluso, si el espacio de parámetros no es discreto, imposible de implementar. En algunos textos, el método descrito se denomina **top-down dynamic programming**.

Programación dinámica: primero los casos pequeños

- Los algoritmos recursivos operan según un esquema *top-down*, planteando la solución a un problema en función de la solución a problemas más pequeños que derivan del primero. Para resolver éstos, se producen problemas todavía más pequeños, etc. hasta que se llega a problemas de tamaño tan pequeño que se pueden resolver directamente.
- Los algoritmos de **Programación Dinámica** (PD) operan justo al revés, según un esquema **bottom-up**: resuelven primero los problemas más pequeños, que almacenan para resolver después problemas mayores a partir de las soluciones ya obtenidas, y así van resolviendo problemas cada vez mayores hasta obtener la solución al problema planteado originalmente.
- **IMPORTANTE**: el espacio de parámetros ha de ser discreto (es decir, indexable), de modo que pueda construirse una tabla.
- Así pues, los algoritmos PD son iterativos y pueden tener fuertes requerimientos de memoria. En muchos casos, sin embargo, la solución a un cierto problema se podrá expresar en términos de unos pocos problemas más pequeños, de modo que la memoria necesaria se puede reducir drásticamente si los cálculos se organizan adecuadamente.
- Los algoritmos PD se aplican a problemas de optimización y parten siempre de una definición recursiva (*ecuación de Bellman*), que expresa la solución óptima a un problema como combinación de las soluciones óptimas a ciertos subproblemas que se derivan del primero.

Programación dinámica: esquema de diseño

- Los algoritmos de programación dinámica tratan de resolver un problema de optimización, de modo que maximizarán (o minimizarán) el valor de una función objetivo que mide la bondad (o el coste) de una solución. Como resultado se obtendría una solución óptima (podría haber varias).

El proceso consta típicamente de 4 pasos:

1. Definir la función objetivo

2. Definir recursivamente el valor óptimo de la función objetivo correspondiente a un problema en términos de los valores óptimos de dicha función para ciertos subproblemas

3. Calcular los valores óptimos de la función objetivo de abajo arriba, empezando por problemas elementales y aplicando la definición recursiva del paso (2) hasta llegar al problema planteado originalmente

4. Recuperar la solución óptima a partir de información almacenada durante el proceso de optimización

El paso (4) puede omitirse si sólo interesa el valor óptimo de la función objetivo:

- IMPORTANTE: para reconstruir la solución óptima, sería necesario guardar el camino seguido (la secuencia de decisiones) en el proceso de optimización.

El problema de la mochila (discreto)

Enunciado: Considerese una mochila capaz de albergar un peso máximo M, y n elementos con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$. **Tanto M como los p_i serán enteros, lo que permitirá utilizarlos como índices en una tabla.** Se trata de encontrar qué combinación de elementos, representada mediante la tupla $x = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$, maximiza el beneficio:

$$f(x) = \sum_{i=1}^{n} b_i x_i \tag{1}$$

sin sobrepasar el peso máximo M:

$$\sum_{i=1}^{n} p_i x_i \leq M \tag{2}$$

- Empezaremos con una versión recursiva que, partiendo de un prefijo x de longitud k y un peso disponible r, devuelve una solución óptima junto con el máximo valor alcanzable de la función objetivo (1).
- Para ello, a partir del prefijo x, se explorarán las dos opciones posibles: añadir el objeto k (comprobando la condición (2)) o no añadirlo.

El problema de la mochila (discreto)

Versión recursiva de referencia

```
# Solucion y maximo beneficio alcanzable
# con los objetos 0..i-1, teniendo un peso m
# disponible en la mochila
def mochila_d_rec(i,m,p,b):
    # base de la recurrencia: 0 objetos
    if i==0:
        return [],0
    # opcion 1: el objeto i-1 NO se introduce
    sol_NO, max_b_NO = mochila_d_rec(i-1,m,p,b)
    # opcion 2: el objeto i-1 SI se introduce
    if p[i-1]<=m:
        sol_SI, max_b_SI = mochila_d_rec(i-1,m-p[i-1],p,b)
        if b[i-1] + max_b_SI > max_b_NO:
            return [1]+sol_SI, b[i-1]+max_b_SI
    return [0]+sol_NO, max_b_NO
```

El problema de la mochila (discreto)

Versión de programación dinámica (1)

El beneficio acumulado de la mejor solución para cada caso (i, m) se almacena en una matriz t, de tamaño $(n+1) \times (M+1)$

```
# Devuelve el maximo beneficio alcanzable con los objetos
# de pesos p y beneficios b, teniendo un peso M disponible
def mochila_d_pd1(p,b,M):
    n=len(p)
    t=[[0 for m in range(M+1)] for i in range(n+1)]
    for i in range(1,n+1):
        for m in range(1,M+1):
            # si se puede introducir el objeto i y el beneficio
            # es mayor que no haciendolo, lo introducimos
            if p[i-1]<=m and b[i-1]+t[i-1][m-p[i-1]]>t[i-1][m]:
                t[i][m]=b[i-1]+t[i-1][m-p[i-1]]
            # en caso contrario, no lo introducimos
            else:
                t[i][m]=t[i-1][m]
    return t[n][M]
```

El problema de la mochila (discreto)

Versión de programación dinámica (1) - Ejemplo

p = [2, 5, 3, 6, 1]

b = [28, 33, 5, 12, 20]

i \ m	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0
1	0	0	28	28	**28**	28	28	28	28	28	28
2	0	0	28	28	28	33	33	61	61	**61**	61
3	0	0	28	28	28	33	33	61	61	**61**	66
4	0	0	28	28	28	33	33	61	61	**61**	66
5	0	20	28	48	48	48	53	61	81	81	**81**

El problema de la mochila (discreto)

Versión de programación dinámica (2)

El procedimiento sólo requiere dos filas de la matriz: la actual y la anterior, puesto que la optimización de la fila i sólo depende de la fila $i-1$.

```
# Devuelve el maximo beneficio alcanzable con los objetos
# de pesos p y beneficios b, teniendo un peso M disponible
def mochila_d_pd2(p,b,M):
    n=len(p)
    ant=[0 for m in range(M+1)]
    act=[0 for m in range(M+1)]
    for i in range(1,n+1):
        for m in range(1,M+1):
            # si se puede introducir el objeto i y el beneficio
            # es mayor que no haciendolo, lo introducimos
            if p[i-1]<=m and b[i-1]+ant[m-p[i-1]]>ant[m]:
                act[m]=b[i-1]+ant[m-p[i-1]]
            # en caso contrario, no lo introducimos
            else:
                act[m]=ant[m]
        ant=act[:]
    return act[M]
```

El problema de la mochila (discreto)

Versión de programación dinámica (3)

Recuperación de la solución: una matriz d guarda las decisiones tomadas; se retrocede fila a fila desde el elemento $d[n][M]$ hasta la fila 0

```
def mochila_d_pd3(p,b,M):
    n=len(p)
    ant=[0 for m in range(M+1)]
    act=[0 for m in range(M+1)]
    d=[[0 for m in range(M+1)] for i in range(n+1)]
    for i in range(1,n+1):
        for m in range(1,M+1):
            if p[i-1]<=m and b[i-1]+ant[m-p[i-1]]>ant[m]:
                act[m]=b[i-1]+ant[m-p[i-1]]
                d[i][m]=1
            else:
                act[m]=ant[m]
                d[i][m]=0
        ant=act[:]
    x=[]
    m=M
    for i in range(n,0,-1):
        x.insert(0,d[i][m])
        if d[i][m]==1:
            m=m-p[i-1]
    return x, act[M]
```

El problema de la mochila (discreto)

Versión de programación dinámica (4)

Recuperación de la solución: los vectores ant y act almacenan el beneficio máximo alcanzable junto con la lista de decisiones correspondiente. Al terminar, el elemento act[M] contiene el beneficio máximo alcanzable y la solución.

```
# Devuelve el maximo beneficio alcanzable con los objetos
# de pesos p y beneficios b, teniendo un peso M disponible,
# asi como la solucion que lo produce
def mochila_d_pd4(p,b,M):
    n=len(p)
    ant=[[0,[]] for m in range(M+1)]
    for i in range(1,n+1):
        act=[[0,[0 for j in range(i)]]]
        for m in range(1,M+1):
            if p[i-1]<=m and b[i-1]+ant[m-p[i-1]][0]>ant[m][0]:
                act.append([b[i-1]+ant[m-p[i-1]][0],ant[m-p[i-1]][1][:]+[1]])
            else:
                act.append([ant[m][0],ant[m][1][:]+[0]])
        ant=act
    return act[M]
```

Ejercicio

Sean u y v dos cadenas de caracteres. Se desea transformar u en u aplicando el menor número posible de operaciones de edición (borrados, inserciones y sustituciones). Escribir en lenguaje Python un algoritmo de programación dinámica que obtenga el número mínimo de operaciones de edición (y cuáles son esas operaciones) para transformar u en v. Calcular la complejidad temporal del algoritmo en función de las longitudes de u y v.

Si $u = abcac$ y $v = babba$, la conversión óptima implica una inserción (la primera b de v), una sustitución (la primera c de u por la tercera b de v) y un borrado (la última c de u). Para desarrollar el algoritmo deberá minimizarse la suma de operaciones de edición sobre una matriz E que guarda el número óptimo de operaciones de edición para convertir la subcadena de $u[1 : i]$ en la subcadena de $v[1 : j]$. En cada paso deberá elegirse la operación menos costosa:

$$E[i][j] = \text{mín}(E[i-1][j-1] + s(u[i], v[j]), E[i-1][j] + 1, E[i][j-1] + 1)$$

donde $s(x, y)$ es el coste de sustituir x por y (que será 1 si $x \neq y$ y 0 si $x = y$). Para recuperar el camino de edición (esto es, las operaciones realizadas), deberá definirse una matriz auxiliar C que guarde memoria del paso óptimo en cada casilla de E.

Ejercicio

Considerese el problema de calcular el coeficiente binomial, definido como sigue:

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 0 & \text{en caso contrario} \end{cases}$$

Escribir en lenguaje Python cuatro formas distintas de calcular el coeficiente binomial:

(a) Mediante un algoritmo recursivo según la definición.
(b) Mediante un algoritmo recursivo con memoria, que evite repetir cálculos.
(c) Mediante un algoritmo de programación dinámica que utilice una matriz de resultados intermedios (el conocido como *triángulo de Pascal*).
(d) Mediante un algoritmo de programación dinámica que utilice una única lista de resultados intermedios, que se va actualizando de izquierda a derecha (ya que los coeficientes de orden n sólo dependen de los de orden $n-1$, que se han calculado en la iteración anterior y están almacenados en la lista).

Aplicar los algoritmos desarrollados para calcular $\binom{10}{5}$ y $\binom{20}{10}$.

En Python, una matriz es una lista de listas, de manera que cada una de las listas (filas) que forman dicha matriz puede tener un número distinto de columnas. Esto permite almacenar de manera sencilla los resultados previos en los métodos (b) y (c). En lo que respecta al método (d), la definición del coeficiente binomial permite usar una única lista de longitud k que empieza teniendo los resultados de la iteración anterior ($n-1$) y se va actualizando de izquierda a derecha con los resultados de la iteración n.

¿Preguntas?

FIN