

Clase dinamica top-down

Algoritmos y estructuras de datos III, 1er cuatrimestre 2023

Repaso

Conceptos básicos

Motivacion

Tratamos con problemas que pueden ser de:

- Optimización
- Busqueda
- Decisión
- Conteo
- Etc

Y nos interesa poder resolverlos eficientemente mediante programación dinámica.

¡Entendamos primero la motivación detrás de esta técnica!

¡Juguemos!





Fibonacci Army

<https://codeforces.com/problemset/problem/72/G>

Enunciado

Enunciado

Al rey Cambyses le encantan los números de Fibonacci

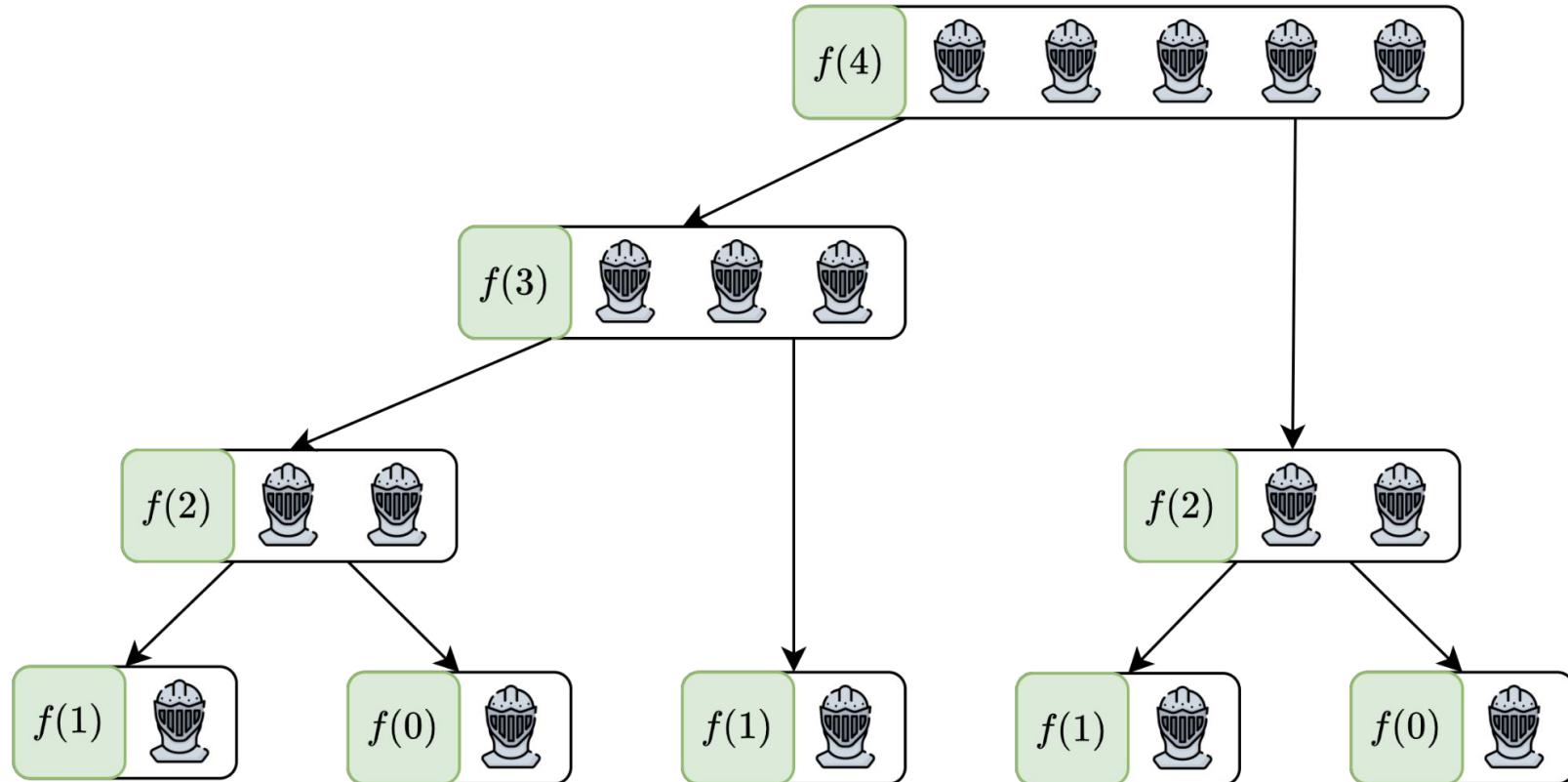
Quiere formar un ejército para él y le interesa que la cantidad de personas de su ejército sea equivalente al n-ésimo número de Fibonacci.

El n-ésimo número de Fibonacci se calcula con la siguiente fórmula:

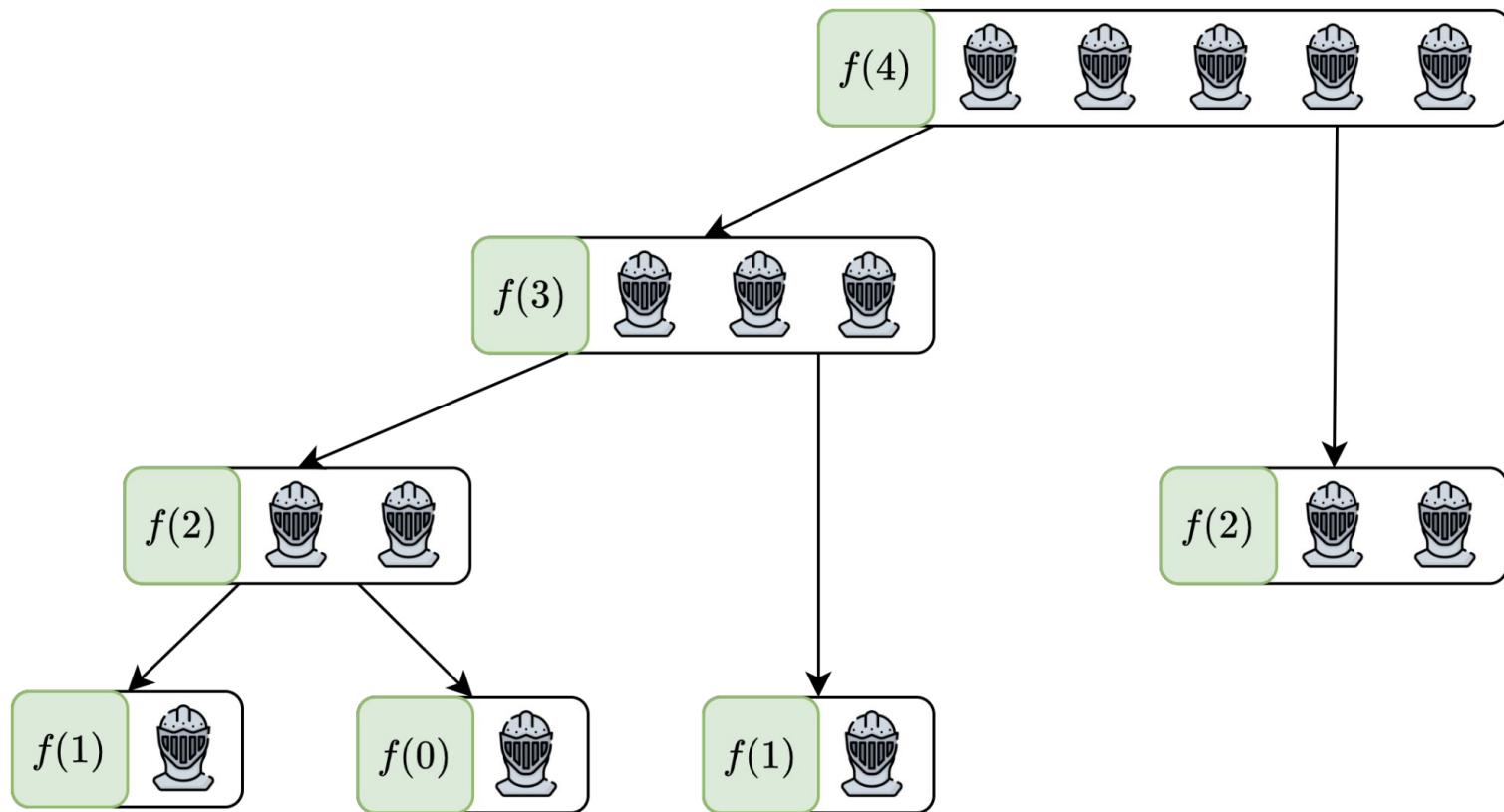
$$f(n) = \begin{cases} f(n - 1) + f(n - 2) & n > 1 \\ 1 & \text{sino} \end{cases}$$

Ejemplos para motivar

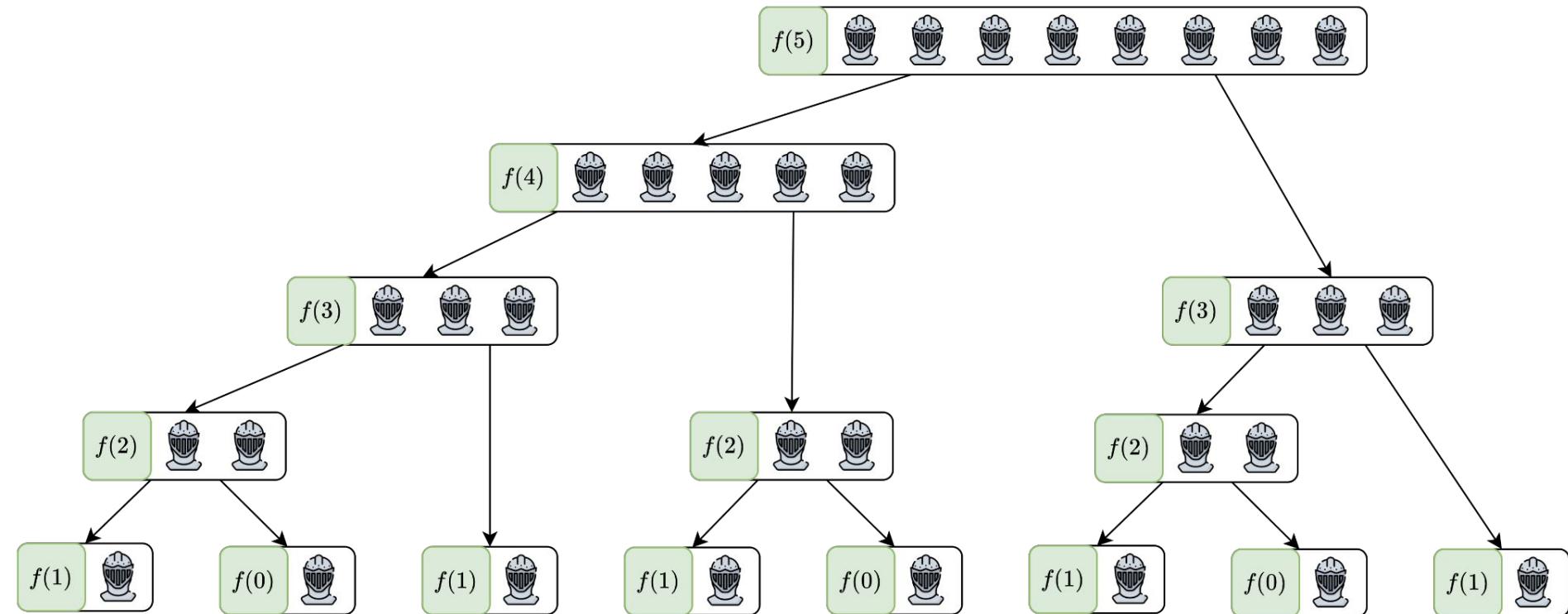
Recursión sin dinámica para $f(4)$



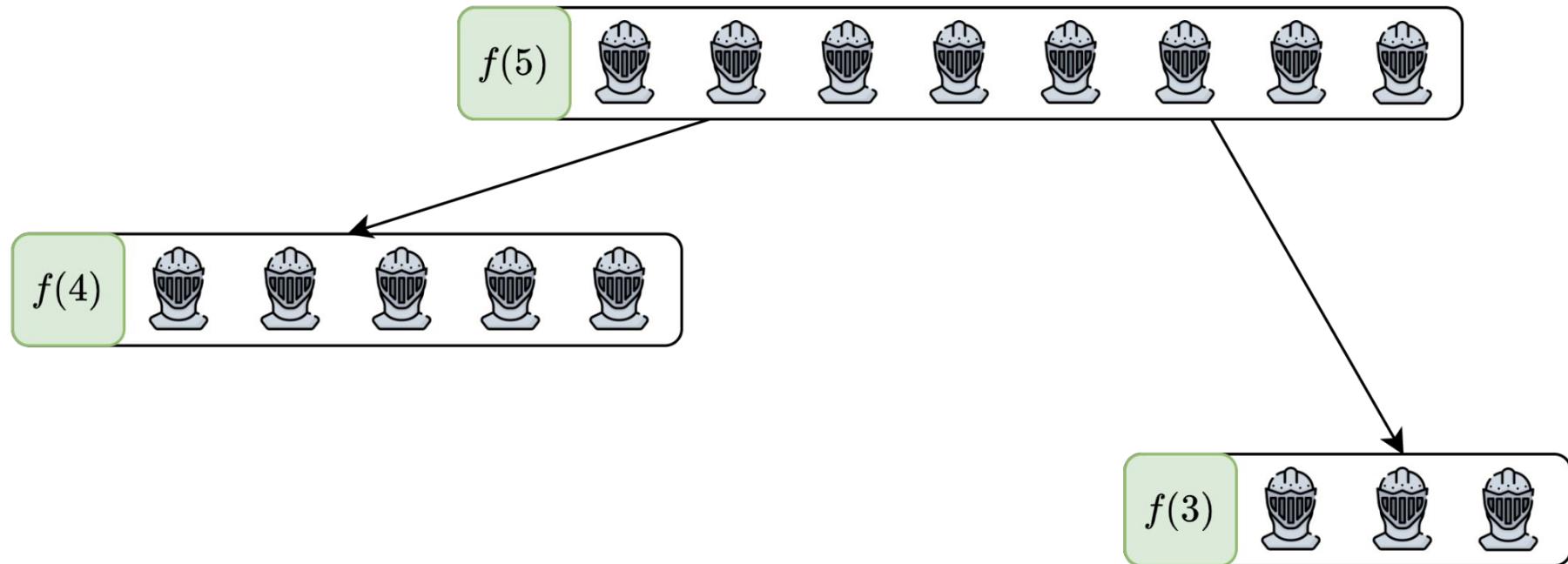
Recursión con dinámica para $f(4)$



Recursión sin dinámica después de correr $f(4)$



Recursión con dinámica aprovechando la corrida de $f(4)$ 😲



Resolución

Receta

Pasos para resolver un problema
de dinamica

1. Definir función recursiva **f**.
 2. Explicar la semántica de **f**.
 3. Llamado/s a **f** que resuelven el problema.
 4. Probar que **f** cumple con la propiedad de superposición de problemas.
 5. Definir algoritmo para **f**.
 6. Determinar complejidad del algoritmo.
-

Paso 1

Funcion recursiva f

Vamos a definir una función partida en la cual especificamos:

- Los parámetros necesarios asociados al problema.
 - El / los casos base acorde al problema.
 - El / los pasos recursivos acorde al problema.
-



Tip #1

Estado del problema



Estado del problema

¿Qué información necesito tener en cada paso recursivo para saber cómo calcular lo que necesito?

En nuestro caso la pregunta seria:

¿Que necesito en el paso recursivo para calcular el n-ésimo número de fibonacci?

En este caso es fácil, necesitamos el n y con eso sabemos calcular el paso recursivo.

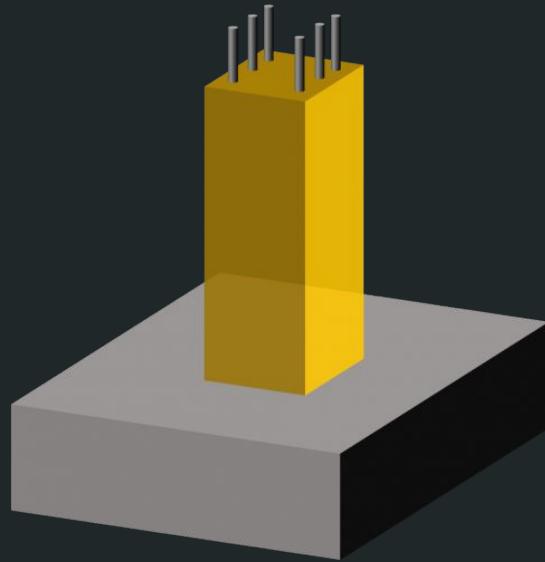
¿Qué función recursiva definimos?

$$f(n) = \begin{cases} 1 & n \leq 1 \\ f(n - 1) + f(n - 2) & \text{sino} \end{cases}$$



Tip #2

Estados base del problema



Estados base del problema

Cuando modelamos la recursión, tenemos que definir casos base.

Estos están ligados a los estados “base” de nuestro problema, y es clave identificarlos.

Si no lo hacemos, terminamos modelando varios casos innecesarios, que ya están expresados a través de la parte recursiva en realidad.

Un ejemplo de esto es el siguiente, donde los estados base son 0 y 1, y el 2 es recursivo en realidad:

$$f(n) = \begin{cases} 1 & n \leq 1 \\ 2 & n = 2 \\ f(n - 1) + f(n - 2) & \text{sino} \end{cases}$$

Es importante notar que no está mal definida la función y es válida igualmente, pero puede resultarnos más difícil convencernos que está bien.

Paso 2

Semántica de función f

Una vez definida la función recursiva, tenemos que explicar qué es lo que hace y su relación con el problema.

Para esto simplemente contamos que simbolizan los:

1. Parámetros
2. Casos base
3. Pasos recursivos

Para nuestro problema.

¿Cuál es la semántica de la función recursiva?

En general, lo que buscamos con esto es establecer una conexión entre la naturaleza recursiva del problema y nuestra función recursiva.

En este caso es sencillo de explicar, porque ya el problema es una fórmula.

- El parámetro **n** representa el n-ésimo número de Fibonacci que estamos buscando.
- El caso base representa los casos donde Fibonacci no es recursivo, que son **n** igual a 0 o 1.
- El paso recursivo determina por un lado los números de Fibonacci de los 2 **n**'s anteriores, y luego junta los resultados para obtener el n-ésimo número de Fibonacci.

Paso 3

Llamado/s que resuelven el problema

Esta parte es bastante sencilla.

Lo único que necesitamos es determinar los estados candidatos de nuestro problema que determinan la solución.

¿Qué llamado/s hacemos para resolver Fibonacci?

En este caso es super sencillo.

Si queremos obtener el n -ésimo número de Fibonacci, simplemente ejecutamos $f(n)$.

Paso 4

f tiene superposición de problemas

Queremos encontrar condiciones para que la cantidad de llamados recursivos de **f** sea mucho más grande que sus posibles estados.

Si esto pasa, significa que estoy viendo estados repetidos si o si, porque hago más llamados recursivos que posibles estados, y programación dinámica me sirve.

¿Cómo evalúo la superposición de problemas?

Queremos contar las combinaciones posibles de los parámetros de la función recursiva y llamados que hace mi función recursiva, y determinar condiciones para que haya superposición.

La receta es:

- Determinar llamados recursivos de **f**:
 - Vemos cuantos llamados recursivos se hacen en cada momento, y cómo cambian los parámetros de **f**.
 - En este caso es 2^N , porque cada paso recursivo tiene 2 llamados, y baja el **n** en 1 y 2 en cada caso.
- Determinar la cantidad de combinaciones de los parámetros de mi función
 - Para los casos que tratamos basta con multiplicar los rangos de valores de cada parámetro de **f**.
 - Una vez más acá es sencillo, es simplemente **N**.
- Entender en qué casos hay muchos más llamados que formas de llamar a la función
 - Determinar cómo deben ser los parámetros para que sus combinaciones sean mucho menos que los llamados.
 - En este caso es ver que **N <<< 2^N**, que claramente vale.

Paso 5

Diseño de algoritmo

¡Casi terminamos!

Definimos en pseudocódigo el algoritmo que utiliza programación dinámica.

Pensemos en el algoritmo sin dinámica por un segundo...

```
def dp(n):
    if n <= 1:
        return 1

    number = dp(n-1) + dp(n-2)

    return number
```



Tip #3

Pensar la memoria basada en el estado



¿Cómo diseño la memoria para recordar llamados repetidos?

El algoritmo necesita una memoria / cache que recuerde llamados recursivos computados previamente, para no repetirlos.

Tenemos que determinar la estructura de la memoria con esta receta:

- La **f** tiene X parámetros.
- Tenemos que recordar cada llamado recursivo posible.
- Armamos una matriz de X dimensiones.
- Cada dimensión es un parámetro, y es equivalente a su rango de valores.

En nuestro caso es simplemente una tabla de 1 dimensión de tamaño **n**, porque no puedo tener llamados recursivos con un **n' > n**.

Aclaración: Para nuestros casos suele ser una matriz la memoria, pero no siempre ocurre.



Tip #4

“Parcheando” el algoritmo para usar dinámica.



“Parcheando” el algoritmo para usar dinámica

Agregar programación dinámica a una función recursiva es super fácil.

Solo tenemos que agregar estas 3 partes:

- Inicializar la memoria.
- Agregar un chequeo al principio de la función para saber si ya resolvimos ese llamado recursivo.
- Antes de retornar el resultado en un llamado recursivo guardamos el resultado en la posición correcta de la memoria.

Parche en Fibonacci

```
# paso 1: inicializo la memoria
cache = [UNDEFINED for _ in range(N+1)]


def dp(n):
    if n <= 1:
        return 1

    # paso 2: chequeo si ya calcule este resultado previamente
    if cache[n] != UNDEFINED:
        return cache[n]

    number = dp(n-1) + dp(n-2)

    # paso 3: guardo el resultado que calcule en este paso en la memoria
    cache[n] = number

return number
```



Tip #5



Como NO parchear un algoritmo con dinámica

Como NO parchear un algoritmo con dinámica

- Al agregar la parte de dinámica, se agregan los chequeos de memoria de los pasos 2 y 3 en demasiados lugares.
- Se chequea la memoria antes de hacer cada llamado recursivo dentro de la función, y se guarda el valor de un sub llamado recursivo en **f**.

Esto puede resultar en un algoritmo correcto, pero es un riesgo:

- Es riesgoso porque al intentar acceder a la posición correcta de la memoria uno puede terminar accediendo a una posición inválida y se rompe todo.

Para no caer en esto, pensemos que la responsabilidad de **f** es chequear si su estado fue resuelto o no, y no preocuparse por los estados de otros llamados.

Ejemplo con Fibonacci de que NO hacer (funciona, si 😅)

```
def dp(n):
    if n <= 1:
        return 1

    numberN1, numberN2 = None, None
    if cache[n - 1] != UNDEFINED:
        numberN1 = cache[n - 1]
    else:
        numberN1 = dp(n-1)
        cache[n - 1] = numberN1

    if cache[n - 2] != UNDEFINED:
        numberN2 = cache[n - 2]
    else:
        numberN2 = dp(n-2)
        cache[n - 2] = numberN2

    return numberN1 + numberN2
```

Paso 6

Complejidad de algoritmo

Tenemos que definir la complejidad del algoritmo de programación dinámica.

Complejidad de algoritmo

La complejidad de un algoritmo con programación dinámica es equivalente a:

(cantidad de estados) * (costo de calcular estado)

En el caso particular de Fibonacci, como la memoria es **O(n)**, sabemos que:

- Tenemos **O(n)** estados.
- Cada estado tiene un costo **O(1)** para calcularlo, porque solo llamamos recursivamente 2 veces y sumamos los resultados.

Luego la complejidad final en este caso es **O(n)**.

Bonus

Complejidad espacial del algoritmo

La complejidad espacial de un algoritmo con programación dinámica es igual a:

cantidad de estados

En el caso particular de Fibonacci, como la memoria es **O(n)**, sabemos que:

- Tenemos **O(n)** estados.

Luego la complejidad final en este caso es **O(n)**.

¿Se puede mejorar esto?

Veanlo en el próximo capítulo de programación dinámica bottom-up 😊

Vacations



<https://codeforces.com/problemset/problem/698/A>

Enunciado

Enunciado

Tomás tiene n días de vacaciones, donde puede hacer actividades:

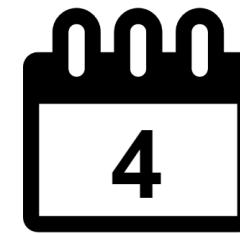
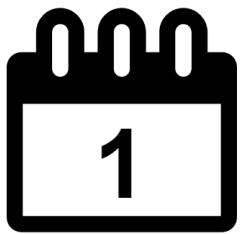
- Se pueden hacer 2 actividades: gimnasio y competencias.
- Cada día puede tener disponible ninguna, alguna o ambas.

Tomas en cada día puede:

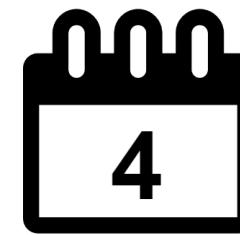
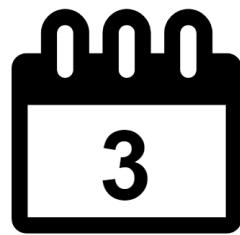
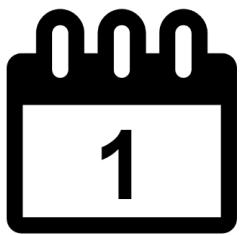
- Hacer una actividad que esté disponible, siempre que no la haya hecho el día anterior.
- Descansar.

La idea es minimizar la cantidad de días de descanso.

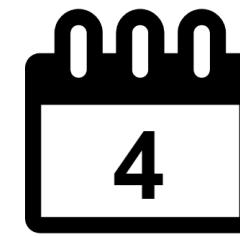
Ejemplo



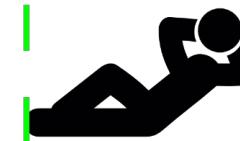
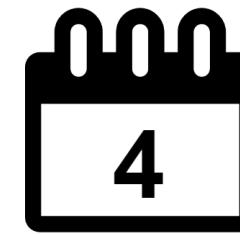
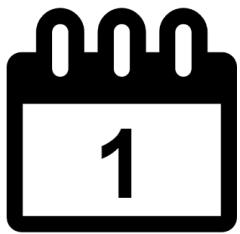
Actividades permitidas



Combinación invalida



Combinación válida pero no óptima



Combinación válida óptima

Resolución

Función recursiva

Modelemos el estado del problema

¿Qué necesitamos para calcular la mínima cantidad de días de descanso?

Pensemos en las condiciones del problema:

- Cada día podemos hacer ciertas actividades.
- No podemos repetir actividades de gimnasio y competencia.

Entonces... para saber qué hacer en cada dia necesitamos saber:

- El día en el que estamos.
- La última actividad que hicimos.

Firma de la función

Basado en el estado que hicimos, quedaria asi la funcion:

$$f(dia, ultAct) = \{$$



Tip #6

Dirección de la recursión



¿En qué sentido hago la recursión?

A veces puede resultar más sencillo hacer la recursión en un sentido que en otro.

Por ejemplo, para este caso particular, tenemos **n** días, de 0 a **n-1**. Podríamos:

- Hacer recursión de 0 a **n-1**.
- Idem pero de **n-1** a 0.

En este caso particular no hay diferencia, pero en otros ejercicios está bueno pensar si una forma es más beneficiosa que otra.

Nosotros vamos a hacerla de 0 a **n-1**.

Definiendo casos base

Pensemos en situaciones donde no tenemos que seguir haciendo recursión y la respuesta es obvia.

Tenemos **n** días, de 0 a **n-1**, ¿Cuáles son los estados base para nosotros?

- Cuando nos pasamos del día **n-1**. ¿Importa la última actividad?
 - No.
- En este caso podemos hacer 0 días de descanso.

Firma de la función con casos base

Actualizando los casos base:

$$f(dia, ultAct) = \{0 \quad dia = n$$

Definiendo casos recursivos

Sabemos que estamos en un día válido, porque ya cubrimos los casos base.

Que estados posibles pueden darse si estamos en un día X y como puedo calcular el resultado en cada caso?

- El día anterior hice gimnasio.
 - Puedo descansar o hacer competencia (si se puede).
- El día anterior hice competencia.
 - Puedo descansar o hacer gimnasio (si se puede).
- El día anterior descansé.
 - Puedo descansar o hacer competencia (si se puede) o hacer gimnasio (si se puede).

Firma de la función con casos recursivos

Supongamos que tenemos las funciones **puedeGym** y **puedeComp** que me indican dado un dia y una actividad anterior si puedo hacer gimnasio o competencia un dia particular.

Pensemos a partir de esto en los distintos casos que hay que agregarle a la función que armamos hasta ahora:

$$f(\text{dia}, \text{ultAct}) = \begin{cases} 0 & \text{dia} = n \\ \dots \end{cases}$$

Firma de la función con casos recursivos

Puedo hacer solo Gym:

$$f(dia, ultAct) = \begin{cases} 0 & dia = n \\ min(1 + f(dia + 1, DESC), f(dia + 1, GYM)) & puedeGym(dia, ultAct) \wedge \neg puedeComp(dia, ultAct) \end{cases}$$

Firma de la función con casos recursivos

Puedo hacer solo Competencia:

$$f(dia, ultAct) = \begin{cases} 0 & dia = n \\ \min(1 + f(dia + 1, DESC), f(dia + 1, GYM)) & puedeGym(dia, ultAct) \wedge \neg pudeComp(dia, ultAct) \\ \min(1 + f(dia + 1, DESC), f(dia + 1, COMP)) & puedeComp(dia, ultAct) \wedge \neg pudeGym(dia, ultAct) \end{cases}$$

Firma de la función con casos recursivos

Puedo hacer ambas:

$$f(\text{dia}, \text{ultAct}) = \begin{cases} 0 & \text{dia} = n \\ \min(1 + f(\text{dia} + 1, \text{DESC}), f(\text{dia} + 1, \text{GYM})) & \text{puedeGym}(\text{dia}, \text{ultAct}) \wedge \neg \text{puedeComp}(\text{dia}, \text{ultAct}) \\ \min(1 + f(\text{dia} + 1, \text{DESC}), f(\text{dia} + 1, \text{COMP})) & \text{puedeComp}(\text{dia}, \text{ultAct}) \wedge \neg \text{puedeGym}(\text{dia}, \text{ultAct}) \\ \min(1 + f(\text{dia} + 1, \text{DESC}), f(\text{dia} + 1, \text{GYM}), f(\text{dia} + 1, \text{COMP})) & \text{puedeGym}(\text{dia}, \text{ultAct}) \wedge \text{puedeComp}(\text{dia}, \text{ultAct}) \end{cases}$$

Firma de la función con casos recursivos

No puedo hacer ninguna:

$$f(\text{dia}, \text{ultAct}) = \begin{cases} 0 & \text{dia} = n \\ \min(1 + f(\text{dia} + 1, \text{DESC}), f(\text{dia} + 1, \text{GYM})) & \text{puedeGym}(\text{dia}, \text{ultAct}) \wedge \neg \text{puedeComp}(\text{dia}, \text{ultAct}) \\ \min(1 + f(\text{dia} + 1, \text{DESC}), f(\text{dia} + 1, \text{COMP})) & \text{puedeComp}(\text{dia}, \text{ultAct}) \wedge \neg \text{puedeGym}(\text{dia}, \text{ultAct}) \\ \min(1 + f(\text{dia} + 1, \text{DESC}), f(\text{dia} + 1, \text{GYM}), f(\text{dia} + 1, \text{COMP})) & \text{puedeGym}(\text{dia}, \text{ultAct}) \wedge \text{puedeComp}(\text{dia}, \text{ultAct}) \\ 1 + f(\text{dia} + 1, \text{DESC}) & \text{sino} \end{cases}$$

Semántica de función

¿Cuál es la idea detrás de la función recursiva f ?

Nuestra función f nos retorna, dado un día i y una última actividad j , la mínima cantidad de descansos que se puede tomar en el intervalo $[i, n]$ si en el $i-1$ se hizo j . Esto lo logra de la siguiente manera:

- Si no hay días para hacer actividades, entonces se puede descansar 0 días.
- Si hay, entonces retorna la mínima cantidad de días de descanso contemplando que:
 - Si puedes solo gym o competencia, entonces buscas el mínimo entre hacerla y prohibirle para los días restantes, o descansar y sumarle 1 día de descanso al resultado para los días restantes.
 - Si puedes hacer gym y competencia, entonces es chequear el mínimo entre los casos de arriba, de intentar cada actividad, o descansar.
 - Si no se puede hacer nada calculamos el mínimo de días de descanso a partir del día siguiente y le sumamos 1 por el día actual de descanso.

Llamados para resolver el problema

Llamados a la función **f** para resolver el problema

Pensemos en que dia y última actividad tenemos que empezar para poder obtener la respuesta a nuestro problema:

- Tenemos **n** días para hacer actividades, y el primero es el 0.
- En el primer día podemos hacer cualquier actividad permitida.
 - ¿Cómo permitimos esto mediante el parámetro de la última actividad realizada?
 - Podemos definir que la última fue un día de descanso.

Entonces... ¿Cuál sería la llamada inicial basada en estas observaciones?

f(0, DESC)

Superposición de problemas

¿Cómo tenemos superposición?

Pensemos mirando la función abajo que cantidades tenemos de:

- Estados: Es la combinación de un día y la última actividad.
 - Hay n días.
 - Hay 3 actividades posibles.

Tenemos $\mathbf{O(3N)} = \mathbf{O(N)}$ estados.

- Llamados recursivos:
 - La función tiene 3 llamados recursivos en el peor caso.
 - En cada paso recursivo sube en 1 día.

Nos quedan $\mathbf{O(3^N)}$ llamados recursivos.

En este caso podemos afirmar directamente que no necesitamos condiciones sobre \mathbf{N} , porque siempre se cumple que $\mathbf{N} <<< 3^N$.

$$f(dia, ultAct) = \begin{cases} 0 \\ min(1 + f(dia + 1, DESC), f(dia + 1, GYM)) \\ min(1 + f(dia + 1, DESC), f(dia + 1, COMP)) \\ min(1 + f(dia + 1, DESC), f(dia + 1, GYM), f(dia + 1, COMP)) \\ 1 + f(dia + 1, DESC) \end{cases}$$

$dia = n$
 $puedeGym(dia, ultAct) \wedge \neg puedeComp(dia, ultAct)$
 $puedeComp(dia, ultAct) \wedge \neg puedeGym(dia, ultAct)$
 $puedeGym(dia, ultAct) \wedge puedeComp(dia, ultAct)$
sino

Algoritmo de función f

Algoritmo sin programación dinámica

```
def dp(dia, ultAct):
    if dia == n:
        return 0

    minDiasDesc = 1 + dp(dia+1, DESC)

    if puedeGym(dia, ultAct):
        minDiasDesc = min(minDiasDesc, dp(dia+1, GYM))
    if puedeComp(dia, ultAct):
        minDiasDesc = min(minDiasDesc, dp(dia+1, COMP))

    return minDiasDesc

print(dp(0, DESC))
```

Algoritmo con programación dinámica

```
cache = [[-1 for _ in range(3)] for _ in range(n)] # cache[dia][ultAct]

def dp(dia, ultAct):
    if dia == n:
        return 0

    if cache[dia][ultAct] != -1:
        return cache[dia][ultAct]

    minDiasDesc = 1 + dp(dia+1, DESC)

    if puedeGym(dia, ultAct):
        minDiasDesc = min(minDiasDesc, dp(dia+1, GYM))
    if puedeComp(dia, ultAct):
        minDiasDesc = min(minDiasDesc, dp(dia+1, COMP))

    cache[dia][ultAct] = minDiasDesc
    return minDiasDesc

print(dp(0, DESC))
```

Complejidad del algoritmo

Determinando complejidad de f

Usemos la técnica de los 2 factores que determinan la complejidad:

- Estados: Hay N días y 3 actividades
 - Tenemos $O(3N) = O(N)$.
- Costo por estado: A lo sumo hacemos 3 llamados recursivos, y luego tomamos el máximo.
 - Tenemos luego $O(1)$ aca.

Entonces como costo total tenemos $O(N)$.

Farmer



<https://codeforces.com/problemset/problem/41/D>

Enunciado

Enunciado

Un granjero tiene un terreno de **N** metros de largo, y **M** de ancho, dividido en celdas de 1 metro cuadrado. En algunas celdas hay una cantidad arbitraria de arvejas.

El granjero tiene como objetivo recolectar la mayor cantidad de arvejas, respetando que:

- Empieza desde alguna celda en el comienzo del terreno ($y = 0$).
- Se puede mover en 2 sentidos:
 - Adelante e izquierda.
 - Adelante y derecha.
- Tiene que llegar al final del terreno ($y = \mathbf{N}$) con una cantidad de arvejas recolectadas divisible por **K+1**, con **K** un número fijo dado.

Ejemplo

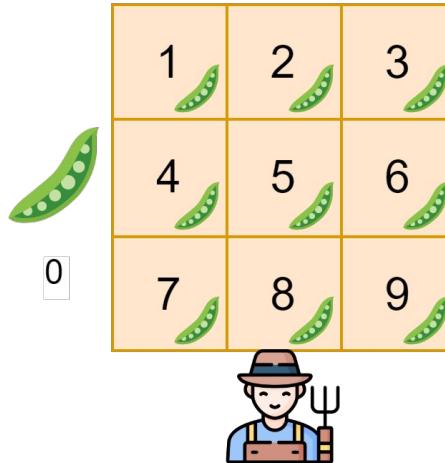
Ejemplo

En este caso tenemos:

- $N = 3$
- $M = 3$
- $K = 1$

Y las arvejas en cada casillero son las que se muestran en el tablero.

$$k = 1$$



Ejemplo inválido

$k = 1$

1	2	3
4	5	6
7		8

Peas: 1, 2, 3, 4, 5, 6, 7, 8, 9

$k = 1$

1	2	3
4		6
0	8	9

Peas: 1, 2, 3, 4, 5, 6, 7, 8, 9
Index: 12

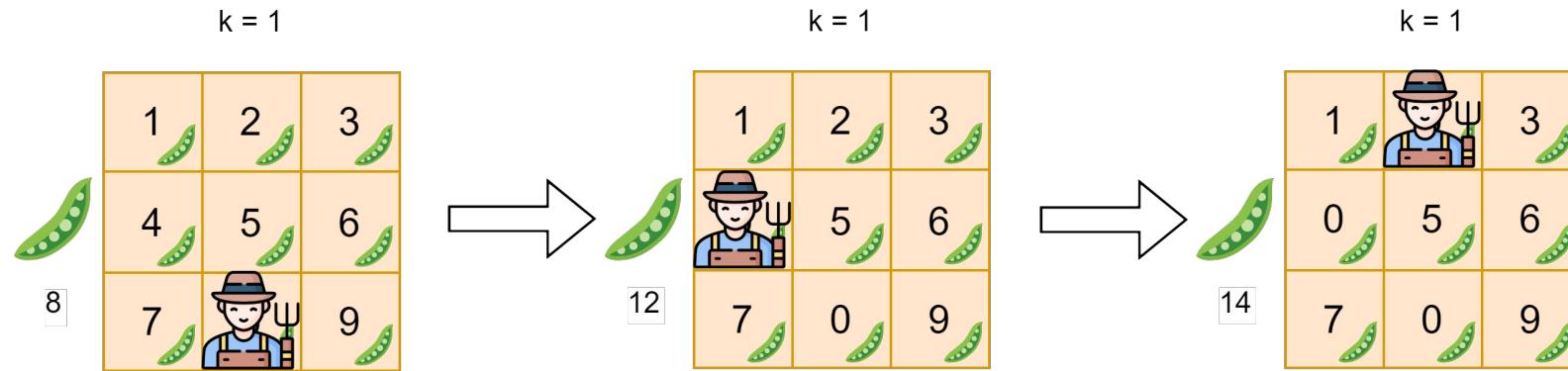
$k = 1$

1	2	
4	0	6
0	8	9

Peas: 1, 2, 3, 4, 5, 6, 7, 8, 9
Index: 15

No es solución válida, porque $15 \bmod 2$ no es 0.

Ejemplo válido no óptimo



Es una solución válida, porque $14 \bmod 2$ es 0, pero no es la óptima.

Ejemplo válido óptimo

$k = 1$

1	2	3
4	5	6
7	8	9

8



$k = 1$

1	2	3
4	5	6
7	0	9

14



$k = 1$

1	2	3
4	5	0
7	0	9

16



Es una solución válida, porque 16 módulo 2 es 0, y es la óptima.

Resolución

Función recursiva

Modelemos el estado del problema

¿Qué necesitamos saber en todo momento para poder decidir cómo calcular la máxima cantidad de arvejas que puede recolectar el granjero?

Pensemos en las condiciones del problema:

- El granjero en todo momento se encuentra en una celda del terreno.
- El granjero viene acumulando una cantidad de arvejas.

Entonces... para saber qué hacer en cada momento necesitamos:

- Las coordenadas (x, y) del terreno.
- La cantidad de arvejas que recolectamos hasta ahora.

Firma de la función

Basado en el estado que hicimos, quedaria asi la funcion:

$$f(x, y, arv) = \{$$

Definiendo casos base

Pensemos cuando podemos determinar el máximo de arvejas sin recursión.

Cuando llegamos arriba de todo en el terreno no podemos avanzar más...

¿Cuáles son los estados base para nosotros pensando en esto?

- Llegamos a la fila y
 - ¿Importa lo que acumulamos de arvejas?
 - Si.
- Tenemos 2 casos entonces, la cantidad de arvejas es:
 - Válida.
 - Invalida.



Tip #7

Representando los neutros de la operación.

Representando los neutros de la operación recursiva

Cuando retornamos los resultados de los casos base, es importante saber que queremos representar. Estos suelen ser el neutro de la operación que usamos recursivamente.

Para este problema, queremos representar 2 casos cuando no podemos avanzar más:

1. Llegamos a una cantidad de arvejas válida.
2. Llegamos a una cantidad de arvejas invalida.

Pensemos que retornar en cada caso:

1. Es fácil, retornamos 0 porque no podemos obtener más arvejas desde acá.
2. Acá casi estamos como en el caso de arriba, pero en una situación invalida.
 - a. Para saber que retornar, pensemos la naturaleza del problema.
 - i. Buscamos maximizar la cantidad de arvejas.
 - b. Entonces... ¿Que retornamos para no afectar la maximización si llegamos a una situación invalida?
 - i. Retornamos **-INFINITO**.

Firma de la función con casos base

Actualizando los casos base:

$$f(x, y, arv) = \begin{cases} 0 & y = n \wedge arv \bmod (k + 1) = 0 \\ -\infty & y = n \wedge arv \bmod (k + 1) \neq 0 \end{cases}$$

Definiendo casos recursivos

Estoy en coordenadas **(0, y)**, o sea, a la izquierda de todo.

- Puedo únicamente moverme arriba a la derecha.

$$f(x, y, arv) = \begin{cases} 0 & y = n \wedge arv \bmod (k + 1) = 0 \\ -\infty & y = n \wedge arv \bmod (k + 1) \neq 0 \\ terr[x][y] + f(x + 1, y + 1, sigArv) & x = 0 \\ \text{donde } sigArv = arv + terr[x][y] \end{cases}$$

Definiendo casos recursivos

Estoy en coordenadas **(M-1, y)**, o sea, a la derecha de todo.

- Puedo únicamente moverme arriba a la izquierda.

$$f(x, y, arv) = \begin{cases} 0 & y = n \wedge arv \bmod (k + 1) = 0 \\ -\infty & y = n \wedge arv \bmod (k + 1) \neq 0 \\ terr[x][y] + f(x + 1, y + 1, sigArv) & x = 0 \\ terr[x][y] + f(x - 1, y + 1, sigArv) & x = m - 1 \end{cases}$$

donde $sigArv = arv + terr[x][y]$

Definiendo casos recursivos

Estoy en coordenadas **(x, y)** donde **0 < x < M-1**, o sea, en el medio.

- Puedo moverme arriba a la izquierda o derecha.

$$f(x, y, arv) = \begin{cases} 0 & y = n \wedge arv \bmod (k + 1) = 0 \\ -\infty & y = n \wedge arv \bmod (k + 1) \neq 0 \\ terr[x][y] + f(x + 1, y + 1, sigArv) & x = 0 \\ terr[x][y] + f(x - 1, y + 1, sigArv) & x = m - 1 \\ terr[x][y] + \max(f(x - 1, y + 1, sigArv), f(x + 1, y + 1, sigArv)) & \text{sino} \\ \text{donde } sigArv = arv + terr[x][y] \end{cases}$$

Todo muy lindo pero... no funciona :(

Verdict	Time	Memory
Time limit exceeded on test 24	2000 ms	129100 KB

¿Por qué no funciona esto?

Sabemos que vamos a terminar con una memoria de estas dimensiones:

- Rango de x
 - $O(M)$
- Rango de y
 - $O(N)$
- Cota máxima de arvejas (considerando que cada celda tiene a lo sumo 9)
 - $O(MN*10) = O(MN)$

En total tenemos luego $O(MNMN) = O(M^2 N^2)$.

Esto es muy pesado, agrega mucha complejidad al algoritmo.

Refinemos el estado del problema

La pregunta clave acá es... ¿Necesitamos saber cuantas arvejas venimos acumulando?

- ¿Cambia en algo si $k = 1$ y estoy en una coordenada (x, y) con 10 arvejas en un caso, y en otro estado también estoy en (x, y) pero tengo 8 por ejemplo?
 - No, porque ambos en módulo 2 son 0.
- Entonces... me basta con saber cuanto es el módulo $(k+1)$ de arvejas que vengo recolectando, ¿No?

Empecemos desde el principio reajustando la función recursiva f.

Función recursiva f refinada

$$f(x, y, arvMod) = \begin{cases} 0 & y = n \wedge arvMod = 0 \\ -\infty & y = n \wedge arvMod \neq 0 \\ terr[x][y] + f(x + 1, y + 1, sigArvMod) & x = 0 \\ terr[x][y] + f(x - 1, y + 1, sigArvMod) & x = m - 1 \\ terr[x][y] + \max(f(x - 1, y + 1, sigArvMod), f(x + 1, y + 1, sigArvMod)) & \text{sino} \\ \text{donde } sigArvMod = (arvMod + terr[x][y]) \bmod (k + 1) \end{cases}$$

Semántica de función

¿Que representa la función recursiva f ?

Repasemos que hace la función refinada:

- Si no se puede avanzar más, entonces retorna 0 si y sólo si se recolectó una cantidad de arvejas múltiplo de **$k+1$** . Si no se retorna -**INFINITO** simbolizando que es invalida.
- Si se puede avanzar, retorna la máxima cantidad de arvejas que se puede recolectar múltiplo de **$k+1$** contemplando que:
 - Si solo podemos ir a la derecha entonces vemos cuantas podemos recolectar a partir de esa dirección, considerando el módulo de arvejas actualizado con las de la celda actual.
 - Si solo podemos movernos a la izquierda es análogo.
 - Si podemos movernos en ambas direcciones basta con evaluar cada dirección y quedarnos con la que retorne la máxima cantidad de arvejas de ambas.

Llamados para resolver el problema

Llamados a la función f para resolver el problema

¿Con qué coordenadas y que módulo de arvejas tenemos que empezar para poder obtener la respuesta a nuestro problema?

- Empezamos siempre desde la primera fila del terreno.
 - Entonces $y = 0$.
- Tenemos permitido empezar en cualquier celda de la primera fila.
 - Tenemos $0 \leq x < M$.
- Cuando empezamos no recolectamos ninguna arveja (considerando que la misma función agrega las arvejas en las que está parado el granjero)
 - Por lo tanto tenemos que el módulo de arvejas es 0.

¿Nos alcanza con una única llamada para resolver el problema?

- No, necesitamos considerar varias.
- Llamamos a la función con $f(x, 0, 0)$, donde $0 \leq x < M$.
- Nos quedamos con el resultado máximo de todos estos.

Superposición de problemas

¿Hay superposición?

Pensemos mirando la función abajo que cantidades tenemos de:

- Estados: Es la combinación de las coordenadas y el módulo de arvejas
 - Las coordenadas cumplen que $0 \leq x < M$, y $0 \leq y < N$
 - El módulo tiene $K+1$ valores.

Tenemos $O(MN(K+1)) = O(MNK)$ estados.

- Llamados recursivos:
 - La función tiene 2 llamados recursivos en el peor caso.
 - En cada paso recursivo avanzamos una fila en el terreno.

Nos quedan $O(2^N)$ llamados recursivos, y hacemos una llamada por cada celda de la primera fila, por lo que obtenemos $O(M2^N)$.

¿Es cierto que siempre $MNK <<< M2^N$?

- Debería ocurrir que $K <<< 2^N / N$ y no podemos asegurarlo sin más información de K y N .

$$f(x, y, arvMod) = \begin{cases} 0 & y = n \wedge arvMod = 0 \\ -\infty & y = n \wedge arvMod \neq 0 \\ terr[x][y] + f(x + 1, y + 1, sigArvMod) & x = 0 \\ terr[x][y] + f(x - 1, y + 1, sigArvMod) & x = m - 1 \\ terr[x][y] + \max(f(x - 1, y + 1, sigArvMod), f(x + 1, y + 1, sigArvMod)) & \text{sino} \\ \text{donde } sigArvMod = (arvMod + terr[x][y]) \bmod (k + 1) & \end{cases}$$

Mirando cotas para la complejidad

Sabemos que el problema tiene cotas para las variables de esta forma:

$$2 \leq N \leq 100 \text{ y } 0 \leq K \leq 10$$

Si hacemos el cálculo con los valores máximos, vamos a notar que vale que:

$$10 <<< 2^{100} / 100$$

Sin embargo, podría ocurrir en otros problemas que la complejidad exponencial sea más útil que la que nos da la programación dinámica por las cotas.

La *moraleja* es que hay que evaluar cada caso para ver si programación dinámica lo amerita o la implementación sin esta es mejor.

Algoritmo de función f

```
def dp(x, y, arvMod):
    if y == n:
        if arvMod == 0:
            return 0
        return -INF
    arvMod = (arvMod + grid[y][x]) % (k+1)
    if cache[y][x][arvMod] != -1:
        return cache[y][x][arvMod]

    maxArvs = -INF
    if x > 0:
        maxArvs = max(maxArvs, dp(x-1, y+1, arvMod))
    if x < m-1:
        maxArvs = max(maxArvs, dp(x+1, y+1, arvMod))

    maxArvs += grid[y][x]
    cache[y][x][arvMod] = maxArvs
    return maxArvs

cache = [[[-1 for _ in range(k + 2)] for _ in range(m + 1)] for _ in range(n+1)]

optimo = -1
for c in range(m):
    res = dp(c, 0, 0)
    optimo = max(optimo, res)
```

Complejidad del algoritmo

Complejidad del algoritmo

Sabemos que nuestro nuevo estado depende de las combinaciones del:

- Rango de x
 - $O(M)$
- Rango de y
 - $O(N)$
- Rango del módulo de arvejas
 - $O(K+1) = O(K)$

En total tenemos luego **$O(MNK)$** .

Si bien vamos a llamar al algoritmo para cada celda de la primera fila, ¡El algoritmo va a calcular una única vez cada estado!

Caesar's Legions



<https://codeforces.com/problemset/problem/118/D>

Enunciado

Enunciado

Al famoso general Caesar le gusta poner en línea sus soldados, que son lacayos y jinetes:

- Tiene un ejército de n_1 lacayos, y n_2 jinetes.
- No le gusta que la forma de poner en línea a sus soldados incumpla con que:
 - Hay más de k_1 lacayos en fila seguidos.
 - Hay más de k_2 jinetes en fila seguidos.
- Los lacayos son indistinguibles entre ellos, al igual que los jinetes.

Le interesa encontrar todas las combinaciones posibles de formar esta línea.

Ejemplo

$k_1 = 1$  $k_2 = 2$ 

Combinaciones inválidas para $k_1 = 1$ y $k_2 = 2$



Combinaciones válidas para $k_1 = 1$ y $k_2 = 2$



Resolución

Función recursiva

Modelemos el estado del problema

¿Qué necesitamos para calcular las formas válidas de organizar las tropas?

Pensemos en las condiciones del problema:

- Tenemos n_l lacayos y n_j jinetes.
- No podemos poner más de k_1 lacayos seguidos, y más de k_2 jinetes tampoco.

Entonces... para saber qué hacer al generar estas combinaciones necesitamos:

- Cantidad de tropas que nos quedan por poner para saber cuando terminar.
- Cantidad de lacayos y jinetes que nos quedan por poner.
- Cantidad de lacayos y jinetes que podemos seguir poniendo seguidos.

Firma de la función

Basado en el estado que hicimos, quedaria asi la funcion:

$$f(t, n_l, n_j, k_l, k_j)_{k_1, k_2} = \{$$

Nota: Los subíndices **k1** y **k2** hacen alusión a las variables globales del código, que podemos usar en todo momento, y nos evitamos pasarlas como parámetro.

Definiendo casos base

Tenemos un total de tropas igual a **n1 + n2**, que va decrementando cada vez que ponemos tropas de lacayos o jinetes. ¿Cuáles son los estados base para nosotros con esto?

- Cuando tenemos un total de 0 tropas.
 - ¿Importa la última tropa que pusimos?
 - No.
 - ¿Importa lo que nos queda de lacayos o jinetes?
 - No, porque deberían ser 0 ya que el total lo es.
 - ¿Importa lo que podemos poner de lacayos o jinetes seguidos?
 - No, porque seguro es 0 o más y no podemos poner más.
- En este caso podemos hacer una única combinación, que es no poner nada.

Firma de la función con casos base

Actualizando los casos base:

$$f(t, n_l, n_j, k_l, k_j)_{k_1, k_2} = \{1 \quad t = 0$$

Definiendo casos recursivos

Tenemos al menos 1 tropa de algún tipo, porque ya cubrimos los casos base.

¿Que estados posibles pueden darse combinando cuantos lacayos y jinetes me quedan y cuantos puse seguidos de cada uno por ultima vez, y como puedo calcular el resultado en cada caso?

- Estamos en un estado donde nos quedan lacayos y podemos poner al menos 1 más en fila.
- Lo mismo pero no podemos poner más lacayos en fila.
- Estamos en un estado donde nos quedan jinetes y podemos poner al menos 1 más en fila.
- Lo mismo pero no podemos poner más jinetes en fila.

Firma de la función con casos recursivos

Pensemos a partir de esto en los distintos casos que hay que agregarle a la función que armamos hasta ahora:

$$f(t, n_l, n_j, k_l, k_j)_{k_1, k_2} = \{1 \quad t = 0$$

Firma de la función con casos recursivos

Podemos pensar en sumar las combinaciones de poner un lacayo o un jinete:

$$f(t, n_l, n_j, k_l, k_j)_{k_1, k_2} = \begin{cases} 1 & t = 0 \\ \text{ponerLacayo}(t, n_l, n_j, k_l, k_j)_{k_1, k_2} + \text{ponerJinete}(t, n_l, n_j, k_l, k_j)_{k_1, k_2} & \text{sino} \end{cases}$$

¿Cómo definimos ponerLacayo y ponerJinete?

Firma de la función con casos recursivos

Pensemos a partir de esto en los distintos casos que hay que agregarle a la función que armamos hasta ahora:

$$f(t, n_l, n_j, k_l, k_j)_{k_1, k_2} = \begin{cases} 1 & t = 0 \\ ponerLacayo(t, n_l, n_j, k_l, k_j)_{k_1, k_2} + ponerJinete(t, n_l, n_j, k_l, k_j)_{k_1, k_2} & \text{sino} \end{cases}$$

$$ponerLacayo(t, n_l, n_j, k_l, k_j)_{k_1, k_2} = \begin{cases} f(t - 1, n_l - 1, n_j, k_l - 1, k_2)_{k_1, k_2} & n_l > 0 \wedge k_l > 0 \\ 0 & \text{sino} \end{cases}$$

$$ponerJinete(t, n_l, n_j, k_l, k_j)_{k_1, k_2} = \begin{cases} f(t - 1, n_l, n_j - 1, k_1, k_j - 1)_{k_1, k_2} & n_j > 0 \wedge k_j > 0 \\ 0 & \text{sino} \end{cases}$$

Paremos acá porque ya saben que pasa al final... 😱

Verdict	Time	Memory
Time limit exceeded on test 8	2000 ms	83800 KB



Tip #8



Reduciendo estados eliminando variables redundantes

Reduciendo estados eliminando variables redundantes

A veces tenemos situaciones donde hay parámetros de nuestro estado que son redundantes.

Un parámetro es redundante cuando puede deducirse a partir del resto.

En nuestro problema actual, hay un parametro innecesario:

- El que nos dice cuántas tropas faltan posicionar. ¿Por qué?
 - Podemos deducirlo a partir de cuantos nos quedan de cada tipo de tropa particular.

En base a esta observación, apliquemos la reducción a la función recursiva f .

Función recursiva **f** con reducción de parámetros

Esta sería la nueva definición removiendo el parámetro innecesario que indica el total de tropas restantes:

$$f(n_l, n_j, k_l, k_j)_{k_1, k_2} = \begin{cases} 1 & n_l + n_j = 0 \\ ponerLacayo(n_l, n_j, k_l, k_j)_{k_1, k_2} + ponerJinete(n_l, n_j, k_l, k_j)_{k_1, k_2} & \text{sino} \end{cases}$$

$$ponerLacayo(n_l, n_j, k_l, k_j)_{k_1, k_2} = \begin{cases} f(n_l - 1, n_j, k_l - 1, k_2)_{k_1, k_2} & n_l > 0 \wedge k_l > 0 \\ 0 & \text{sino} \end{cases}$$

$$ponerJinete(n_l, n_j, k_l, k_j)_{k_1, k_2} = \begin{cases} f(n_l, n_j - 1, k_1, k_j - 1)_{k_1, k_2} & n_j > 0 \wedge k_j > 0 \\ 0 & \text{sino} \end{cases}$$

Semántica de función

¿Cuál es la idea detrás de la función recursiva f?

Nuestra función **f** te dice que:

- En caso de no tener más tropas solo podes conseguir 1 combinación, y es no poner nada.
- Si podes poner tropas de algún tipo, entonces te retorna la cantidad de combinaciones válidas que podrías hacer con estas, considerando que:
 - Si podes poner lacayos, entonces pones un lacayo y contar todas las formas de poner el resto de tropas considerando que podes poner 1 lacayo menos seguido de ahora en más.
 - Idem pero para jinetes.

Es importante notar que en caso de que no se pueda poner ninguna tropa en una de las 2 situaciones detalladas arriba la cantidad de combinaciones es 0.

Llamados para resolver el problema

Llamados a la función **f** para resolver el problema

Pensemos de qué forma queremos empezar a determinar todas las posibles combinaciones de tropas:

- Tenemos **n1** lacayos y **n2** jinetes inicialmente.
- Inicialmente no pusimos ningún tipo de tropa.
 - ¿Cómo permitimos esto mediante los parámetros **k1** y **kj**?
 - Los inicializamos como **k1** y **k2** respectivamente.

Entonces... ¿Cuál sería la llamada inicial basada en estas observaciones?

$$f(n1, n2, k1, k2)$$

Superposición de problemas

¿Condiciones para superposición?

- Estados: Combinación de tropas restantes de lacayos y jinetes y los que podemos poner de corrido de cada uno
 - Hay n_1 lacayos y n_2 jinetes.
 - Hay un límite de k_1 lacayos y k_2 jinetes de corrido respectivamente.

Tenemos $O(n_1 n_2 k_1 k_2)$ estados.

- Llamados recursivos:
 - La función tiene 2 llamados recursivos en el peor caso.
 - En cada paso recursivo bajamos de a 1 tropa.

Nos quedan $O(2^{n_1+n_2})$ llamados recursivos.

Entonces... ¿Cuándo pasa que $n_1 n_2 k_1 k_2 <<< 2^{n_1+n_2}$? Sería ver equivalentemente $k_1 k_2 <<< 2^{n_1+n_2} / (n_1 n_2)$

- Si k_1 y k_2 son mucho más chicos que n_1 y n_2 hay. Por las cotas del problema $1 \leq n_1, n_2 \leq 100$, y $1 \leq k_1, k_2 \leq 10$ tenemos superposición.

$$f(n_1, n_2, k_l, k_j) = \begin{cases} 1 & n_1 + n_2 = 0 \\ ponerLacayo(n_1, n_2, k_l, k_j) + ponerJinete(n_1, n_2, k_l, k_j) & \text{sino} \end{cases}$$

Algoritmo de función f

```
def ponerLacayo(n1, n2, k1, kj):
    if n1 > 0 and k1 > 0:
        return dp(n1 - 1, n2, k1 - 1, kj)
    else:
        return 0

def ponerJinete(n1, n2, k1, kj):
    if n2 > 0 and kj > 0:
        return dp(n1, n2 - 1, k1 , kj - 1)
    else:
        return 0

def dp(n1, n2, k1, kj):
    if (n1 + n2) == 0:
        return 1

    if cache[n1][n2][k1][kj] != -1:
        return cache[n1][n2][k1][kj]

    combinaciones = ponerLacayo(n1, n2, k1, kj) + ponerJinete(n1, n2, k1, kj)

    cache[n1][n2][k1][kj] = combinaciones
    return combinaciones

cache = [[[-1 for _ in range(k2+1)] for _ in range(k1+1)] for _ in range(n2+1)] for _ in range(n1+1)] # cache[n1][n2][k1][kj]
```

Complejidad del algoritmo

Determinando complejidad de f

Usemos la técnica de los 2 factores que determinan la complejidad:

- Estados: Hay n_1 y n_2 de cada tropa, y se pueden poner hasta k_1 y k_2 seguidos de cada uno.
 - Entonces tenemos $O(n_1 n_2 k_1 k_2)$.
- Costo por estado: A lo sumo hacemos 2 llamados recursivos, y luego sumamos.
 - Esto es $O(1)$.

Entonces como costo total tenemos $O(n_1 n_2 k_1 k_2)$.

Bonus

Función recursiva alternativa 1

$$f(n_l, n_j, k, \text{ultTropa})_{k_1, k_2} = \begin{cases} 1 & n_l + n_j = 0 \\ \text{ponerLacayo}(n_l, n_j, k, \text{ultTropa})_{k_1, k_2} + \text{ponerJinete}(n_l, n_j, k, \text{ultTropa})_{k_1, k_2} & \text{sino} \end{cases}$$

$$\text{ponerLacayo}(n_l, n_j, k, \text{ultTropa})_{k_1, k_2} = \begin{cases} f(n_l - 1, n_j, k - 1, \text{LACAYO})_{k_1, k_2} & n_l > 0 \wedge k > 0 \wedge \text{ultTropa} = \text{LACAYO} \\ f(n_l - 1, n_j, k_1 - 1, \text{LACAYO})_{k_1, k_2} & n_l > 0 \wedge \text{ultTropa} = \text{JINETE} \\ 0 & \text{sino} \end{cases}$$

$$\text{ponerJinete}(n_l, n_j, k, \text{ultTropa})_{k_1, k_2} = \begin{cases} f(n_l, n_j - 1, k - 1, \text{JINETE})_{k_1, k_2} & n_j > 0 \wedge k > 0 \wedge \text{ultTropa} = \text{JINETE} \\ f(n_l, n_j - 1, k_2 - 1, \text{JINETE})_{k_1, k_2} & n_j > 0 \wedge \text{ultTropa} = \text{LACAYO} \\ 0 & \text{sino} \end{cases}$$

Función recursiva alternativa 2

$$f(n_l, n_j, ultTropa)_{k_1, k_2} = \begin{cases} 1 & (n_l + n_j) = 0 \\ \sum_{i=1}^{\min(n_l, k_1)} f(n_l - i, n_j, LACAYO)_{k_1, k_2} & ultTropa = JINETE \\ \sum_{i=1}^{\min(n_j, k_2)} f(n_l, n_j - i, JINETE)_{k_1, k_2} & \text{sino} \end{cases}$$

Fire



<https://codeforces.com/problemset/problem/864/E>

Enunciado

Enunciado

El sabueso de preguntas está en problemas, ¡Su casa se incendia! Es tiempo de salvar los artículos más valiosos. Cada artículo tiene los parámetros:

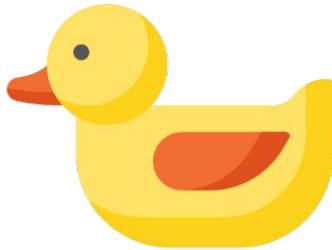
- t : El tiempo que estima el sabueso que le tomara salvarlo.
- d : El tiempo que estima el sabueso a partir del cual se quema el artículo y no sirve más.
- p : El valor del artículo para el sabueso.

El sabueso quiere encontrar la máxima cantidad de artículos que puede salvar, considerando que:

- Salva un artículo luego de otro.
- Si un articulo **A** le tomo ta segundos y luego salva el **B**, entonces le habrá tomado $ta + tb$ segundos en total.

Ejemplo

Ejemplo de 3 artículos



	3
	7
	4

	7
	10
	6

	2
	6
	5

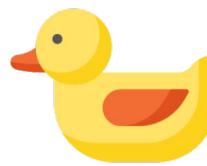
Ejemplo válido no óptimo 1



0



0



hourglass	3
flame	7
coin bag	4



hourglass	7
flame	10
coin bag	6



hourglass	2
flame	6
coin bag	5

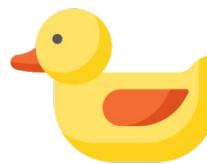
Ejemplo válido no óptimo 1



3



4



hourglass	3
flame	7
coin bag	4



hourglass	7
flame	10
coin bag	6



hourglass	2
flame	6
coin bag	5



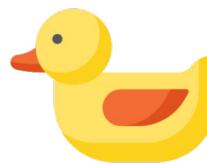
Ejemplo válido no óptimo 2



0



0



hourglass	3
fire	7
coin bag	4



hourglass	7
fire	10
coin bag	6



hourglass	2
fire	6
coin bag	5

Ejemplo válido no óptimo 2



7



6



hourglass	3
flame	7
gold coin	4



hourglass	7
flame	10
gold coin	6



hourglass	2
flame	6
gold coin	5



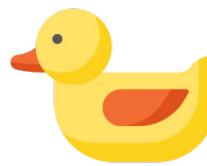
Ejemplo válido óptimo



0



0



hourglass	3
flame	7
coin bag	4



hourglass	7
flame	10
coin bag	6



hourglass	2
flame	6
coin bag	5

Ejemplo válido óptimo



2



5



hourglass	3
fire	7
coin bag	4



hourglass	7
fire	10
coin bag	6



hourglass	2
fire	6
coin bag	5



Ejemplo válido óptimo



9



11



hourglass	3
fire	7
coin bag	4



hourglass	7
fire	10
coin bag	6



hourglass	2
fire	6
coin bag	5



Resolución

Función recursiva

Modelemos el estado del problema

¿Qué factores están involucrados para encontrar la forma válida de rescatar artículos maximizando el valor total?

Pensemos las condiciones del problema:

- Tenemos **n** artículos.
- Cada artículo tiene un
 - Tiempo para salvarse.
 - Tiempo a partir del cual se quema.
 - Valor.

¿Qué necesitamos saber en todo momento para saber cómo seguir eligiendo qué artículo salvar?

- El artículo actual que vamos a decidir salvar o no.
- El tiempo que pasó hasta ahora, para saber si el artículo actual ya se quemó o no.

Firma de la función

Basado en el estado que hicimos, quedaria asi la funcion:

$$f(nroArt, tActual) = \{$$

Definiendo casos base

Pensemos en situaciones donde no tenemos que seguir haciendo recursión y la respuesta es obvia.

Tenemos **n** artículos, y vamos recorriendo desde el 0 al **n-1**. ¿Cuáles son los estados base para nosotros con esto?

- Cuando llegamos a recorrer todos los artículos, estando en el **n**.
 - ¿Importa el tiempo actual?
 - No, porque todo lo que pudimos salvar ya pasó.
- En este caso podemos salvar 0 cosas con un valor total de 0.

Firma de la función con casos base

Actualizando los casos base:

$$f(nroArt, tActual) = \{0 \quad nroArt = n$$

Definiendo casos recursivos

Sabemos que nos queda al menos 1 artículo por salvar o no, porque ya cubrimos los casos base sin artículos restantes.

Que estados posibles pueden darse combinando los artículos que me falta evaluar salvar y el tiempo actual, y como puedo calcular el resultado en cada caso?

- Estamos en un estado donde el artículo actual no se puede salvar en base al tiempo que toma salvarlo más el que tarda en quemarse, combinado con el tiempo actual.
- Lo mismo pero podemos salvarlo.

Firma de la función con casos recursivos

Pensemos a partir de esto en los distintos casos que hay que agregarle a la función que armamos hasta ahora:

$$f(nroArt, tActual) = \begin{cases} 0 & nroArt = n \\ \dots & \dots \end{cases}$$

Firma de la función con casos recursivos

Pensemos a partir de esto en los distintos casos que hay que agregarle a la función que armamos hasta ahora:

$$f(nroArt, tActual) = \begin{cases} 0 & nroArt = n \\ f(nroArt + 1, tActual) & tActual + art.t \geq art.d \end{cases}$$

donde
 $art = articulos[nroArt]$

Firma de la función con casos recursivos

Pensemos a partir de esto en los distintos casos que hay que agregarle a la función que armamos hasta ahora:

$$f(nroArt, tActual) = \begin{cases} 0 & nroArt = n \\ f(nroArt + 1, tActual) & tActual + art.t >= art.d \\ \max(f(nroArt + 1, tActual), art.p + f(nroArt + 1, tActual + art.t)) & \text{donde} \\ & art = articulos[nroArt] \\ & \text{sino} \end{cases}$$

Semántica de función

¿Cuál es la idea detrás de la función recursiva f?

Nuestra función **f** te dice que:

- En caso de no tener más artículos para evaluar salvar, solo podemos salvar 0 con un valor de 0.
- Si todavía te quedan artículos para evaluar salvar, entonces te retorna el máximo valor que puedes obtener, considerando que:
 - Si el artículo actual se puede salvar porque se da que $(\text{tiempo actual}) + (\text{tiempo para salvarlo}) < (\text{tiempo para quemarse})$, entonces te quedas con el máximo entre salvarlo y ver qué hacer con el resto considerando que el tiempo avanza lo que cuesta salvar este artículo, o no salvarlo.
 - Si no se puede salvar simplemente vemos cuánto obtenemos de máximo valor intentando con el resto de artículos.

¡Super fácil e intuitivo! No puede ser cierto...

Llamados para resolver el problema

Llamados a la función **f** para resolver el problema

¿Cómo determinamos el máximo valor posible?

- Tenemos **n** artículos que evaluar.
- El tiempo actual al principio es 0.

Entonces... ¿Cuál sería la llamada inicial basada en estas observaciones?

$$f(0, 0)$$

¿Esto realmente nos asegura obtener el óptimo?

¿Se imaginan si fuera tan fácil siendo el último ejercicio?



El análisis está casi bien, pero no podemos afirmar que obtenemos la respuesta al problema.

Similar pero diferente a knapsack

Este problema es muuy similar al knapsack donde uno tiene artículos con un valor y peso, y busca armar el subconjunto que no excede cierto peso pero maximiza el valor total.

La diferencia crucial acá es que tenemos que salvar los artículos **en orden**, y no podemos decir que vamos a salvar X elementos sin especificar **el orden en el cual lo hacemos**.

La función anterior cae en la misma solución subóptima del segundo ejemplo, por salvar los artículos en un orden incorrecto.

¿Cómo arreglamos esto fácil?

La observación clave es la siguiente:

- Si tenemos el subconjunto óptimo de artículos a salvar, mi mejor apuesta va a ser salvarnos en orden creciente del tiempo que tardan en quemarse.
- Si no logró salvar en un momento alguno así, nunca voy a poder, porque fui salvando lo primero que se podía quemar en cada paso, y en cualquier otro orden se quemaría igual.

Entonces, en lugar de procesar los artículos secuencialmente como vienen, los ordenamos por el tiempo que tarda en quemarse ascendentemente y procesamos esto.

Superposición de problemas

¿Hay superposición?

- Estados: Combinación de cantidad de artículos y el tiempo actual
 - Hay n artículos y cada uno tiene un tiempo para ser salvado, que es a lo sumo 100.
 - El tiempo actual puede ser como máximo $100n$.

Tenemos $O(n^*100n) = O(n^2)$ estados.

- Llamados recursivos:
 - La función tiene 2 llamados recursivos en el peor caso.
 - En cada paso recursivo bajamos de a un artículo.

Nos quedan $O(2^n)$ llamados recursivos.

En el límite de n al infinito, ¿Cuando vale que $n^2 <<< 2^n$?

- Vale siempre, así que tenemos superposición para cualquier n !

$$f(nroArt, tActual) = \begin{cases} 0 & nroArt = n \\ f(nroArt + 1, tActual) & tActual + art.t >= art.d \\ \max(f(nroArt + 1, tActual), art.p + f(nroArt + 1, tActual + art.t)) & \text{donde} \\ & art = articulos[nroArt] \end{cases}$$

$nroArt = n$
 $tActual + art.t >= art.d$
sino

Algoritmo de función f

```
def dp(i, tiempoActual):  
    if i == n:  
        return 0  
  
    if cache[i][tiempoActual] != -1:  
        return cache[i][tiempoActual]  
  
    maxValor = dp(i + 1, tiempoActual)  
    d, t, p, _ = items[i]  
    if tiempoActual + t < d:  
        maxValor = max(maxValor, dp(i + 1, tiempoActual + t) + p)  
  
    cache[i][tiempoActual] = maxValor  
    return maxValor  
  
cache = [[-1 for _ in range(20 * 100 + 1)] for _ in range(n+1)]
```

Complejidad del algoritmo

Determinando complejidad de f

Usemos la técnica de los 2 factores que determinan la complejidad:

- Estados: Hay n artículos, y el tiempo actual puede llegar a ser **$100n$** .
 - Entonces tenemos **$O(n^2)$** .
- Costo por estado: A lo sumo hacemos 2 llamados recursivos, y luego tomamos el máximo
 - Esto es **$O(1)$** .

Entonces como costo total tenemos **$O(n^2)$** .

THE END

