



Dinamica top-down

Algoritmos y estructuras de datos III, 1er cuatrimestre 2023

Repaso

Conceptos básicos

Objetivos



Motivacion

Tratamos con problemas que pueden ser de:

- Optimización
- Busqueda
- Decisión
- Conteo
- Calculo
- Etc

Y nos interesa poder resolverlos eficientemente mediante programación dinámica.

¡Entendamos primero la motivación detrás de esta técnica!

King Army



Enunciado

Enunciado

El rey Cambyses está interesado en armar ejércitos en una serie de días consecutivos.

Mas aun, le interesa que el número de personas de su ejército en el dia d_i sea equivalente a la suma del número de personas del ejército que formó el dia $i-1$ e $i-2$.

La excepción para esto es en el día 0 y 1, en cuyo caso la cantidad de personas en esos dia va a ser siempre 1.

Para el es muy complicado determinar este número, entonces nos pidió que lo ayudemos.

Dado un dia N , tenemos que devolver el número de personas de su ejército.

Es un ejercicio de **cálculo**.

Ejemplos para motivar

iJuguemos!

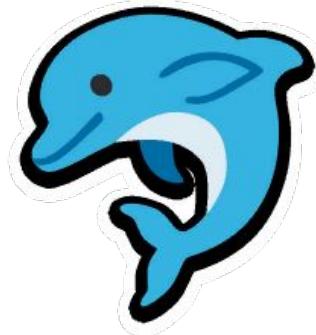


Dory vs Delfín

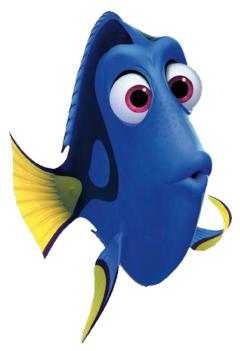
Vamos a jugar a ser Dory o Delfín:



Cada vez que vamos a computar el número de personas para un dia, no vamos a recordar ningún resultado para un día previo.



Igual que antes pero ahora si recordamos resultados de dias previos que calculamos anteriormente.

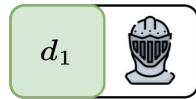


Dia 0



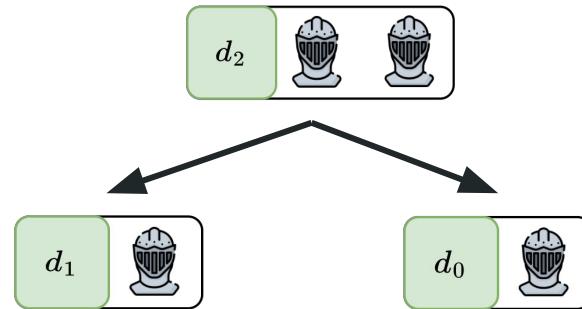


Dia 1



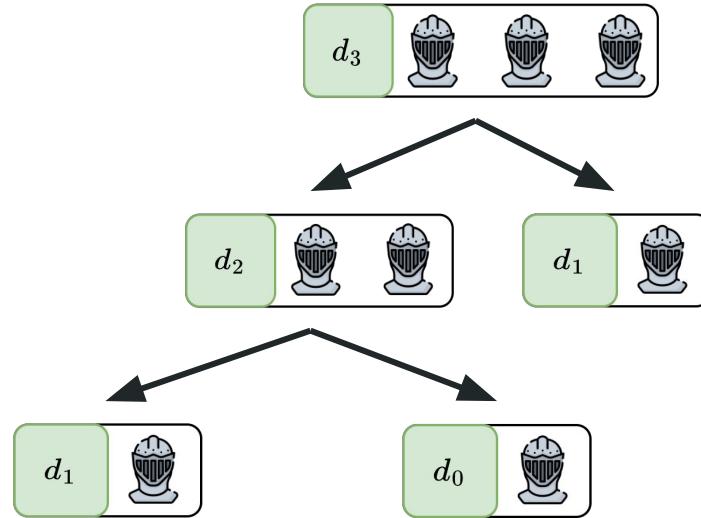


Dia 2



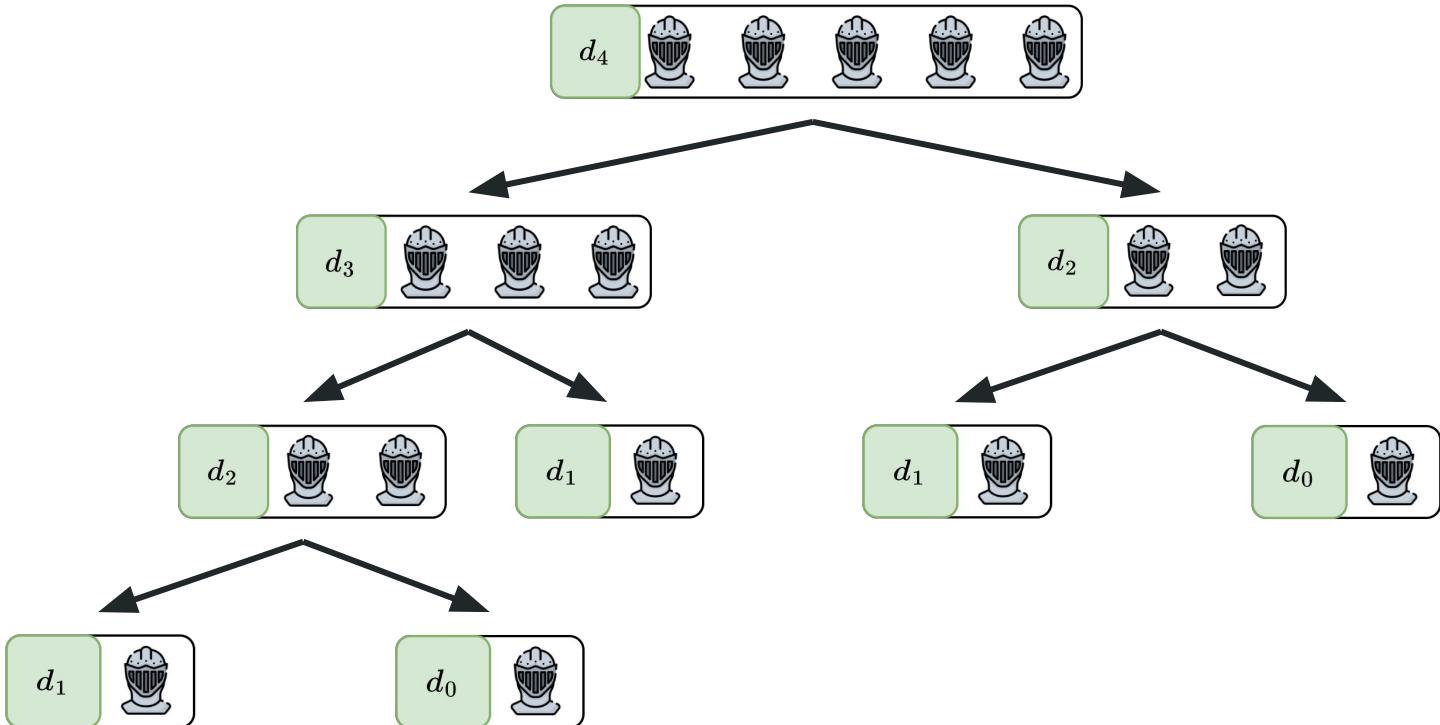


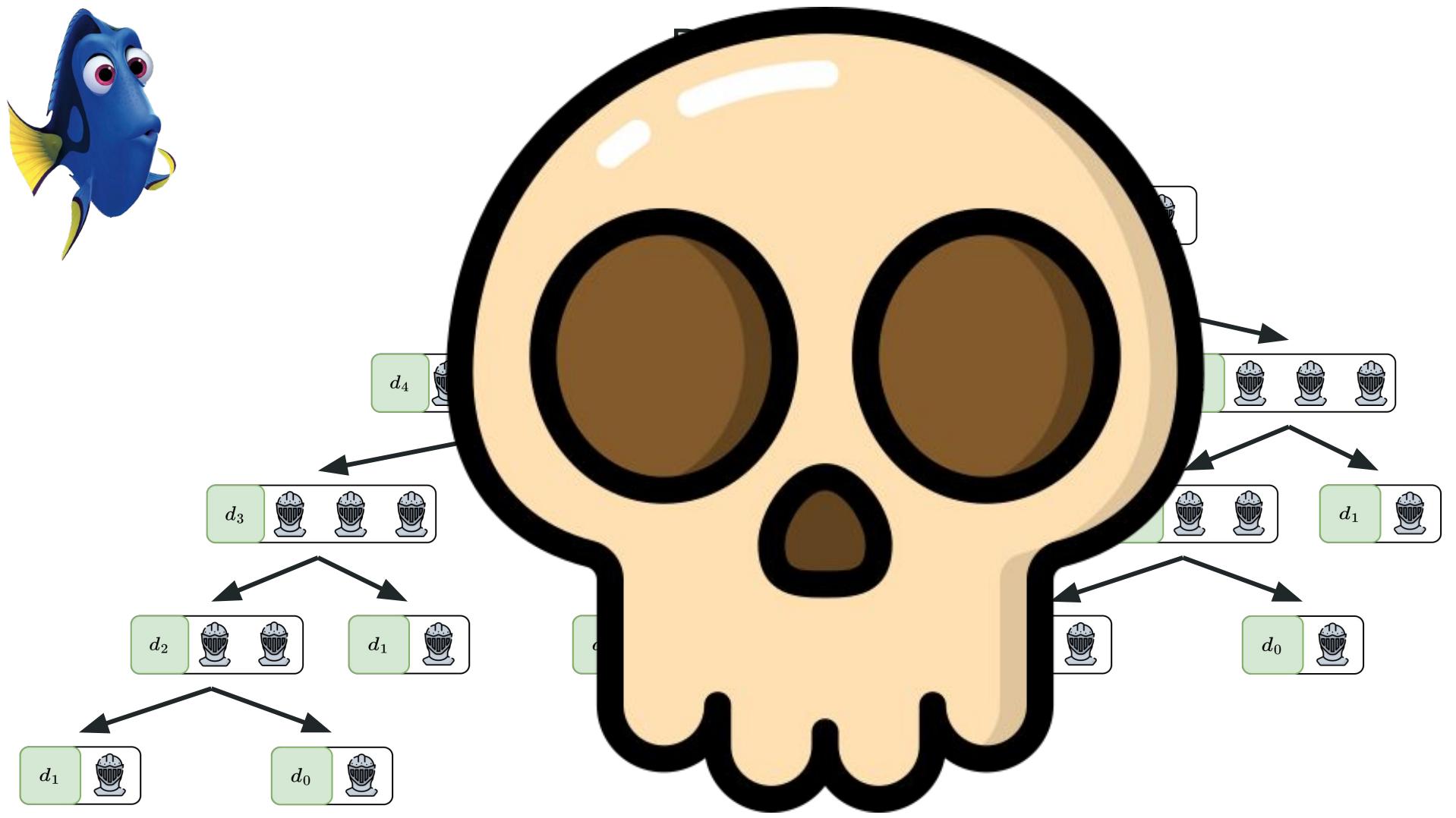
Dia 3

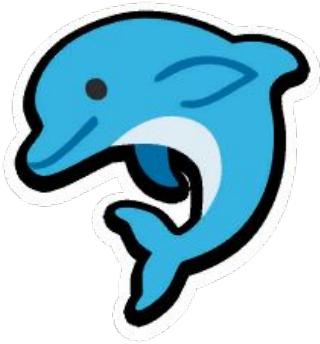




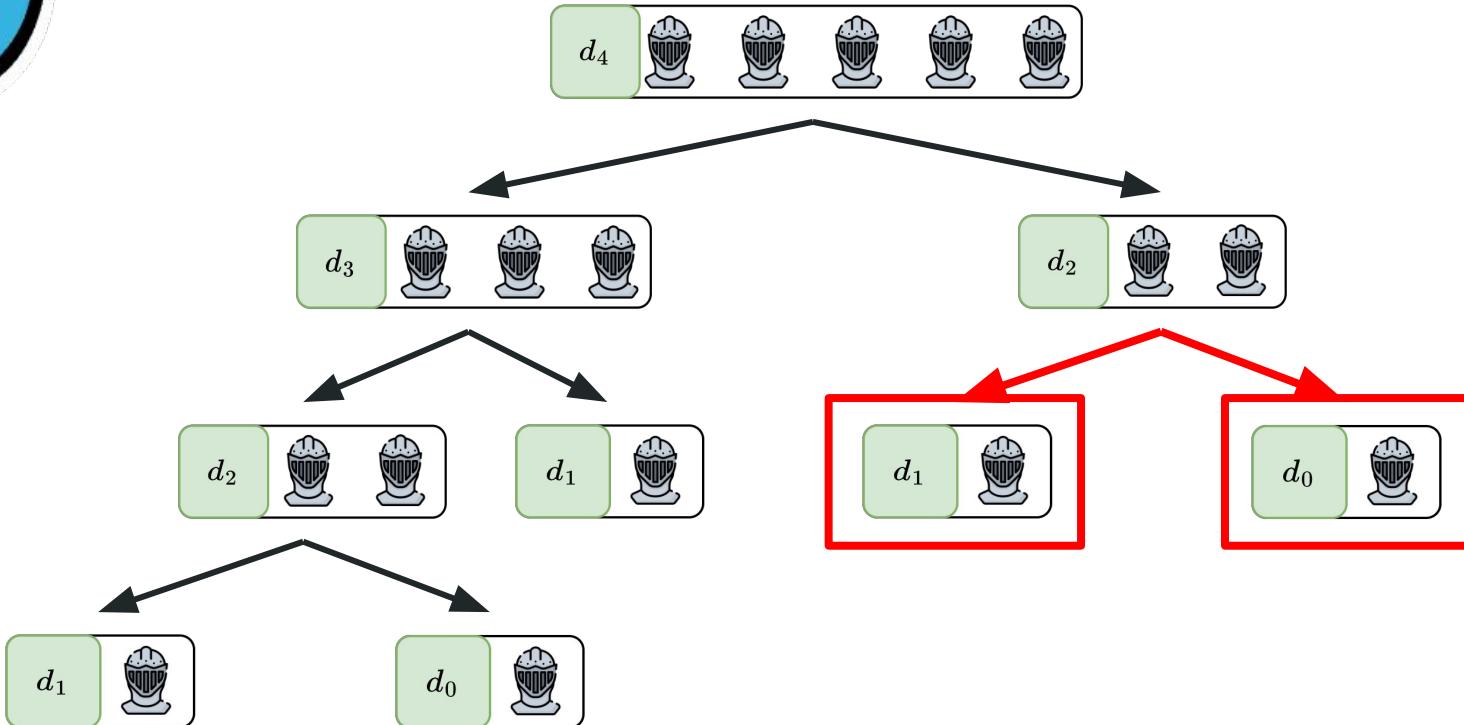
Dia 4

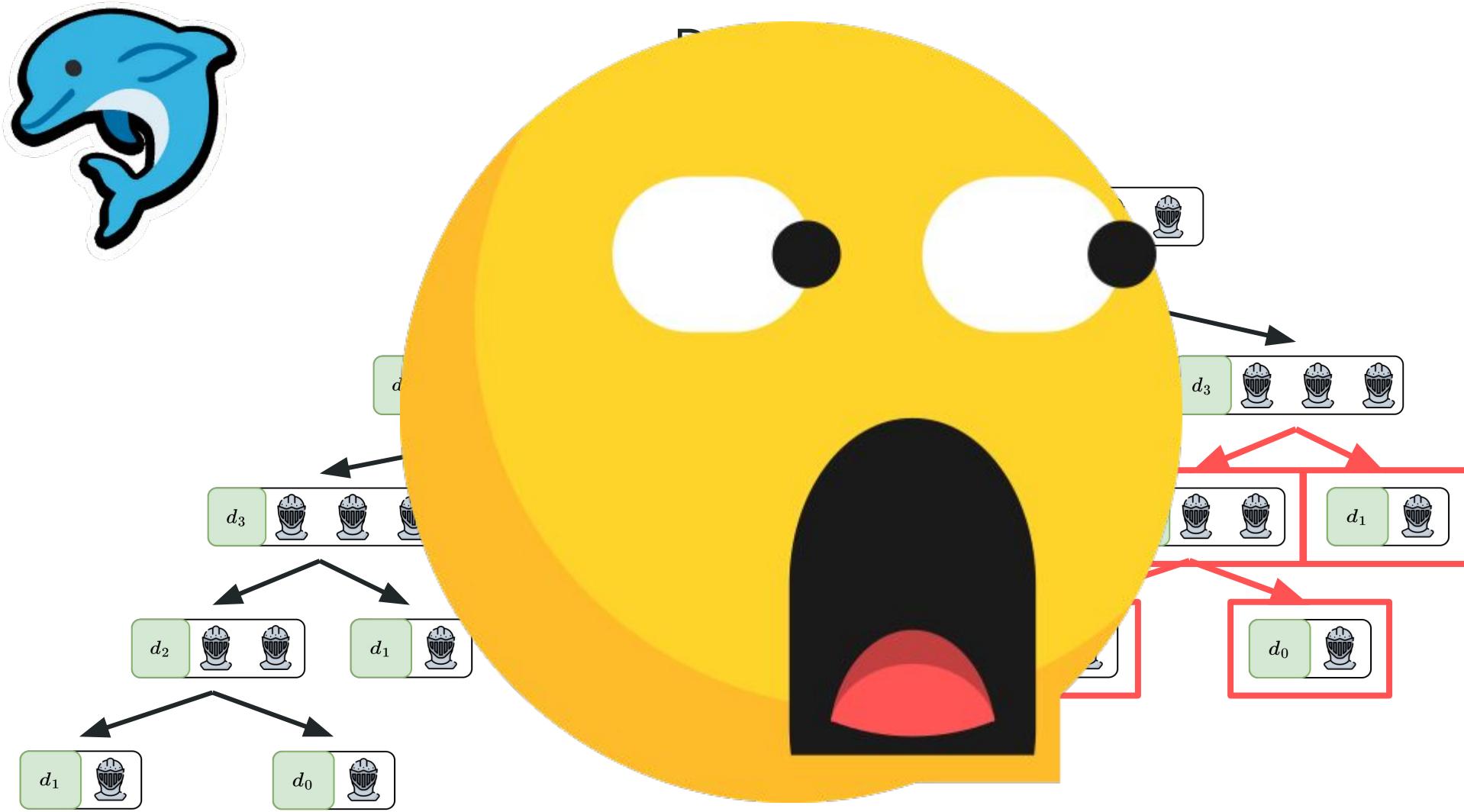




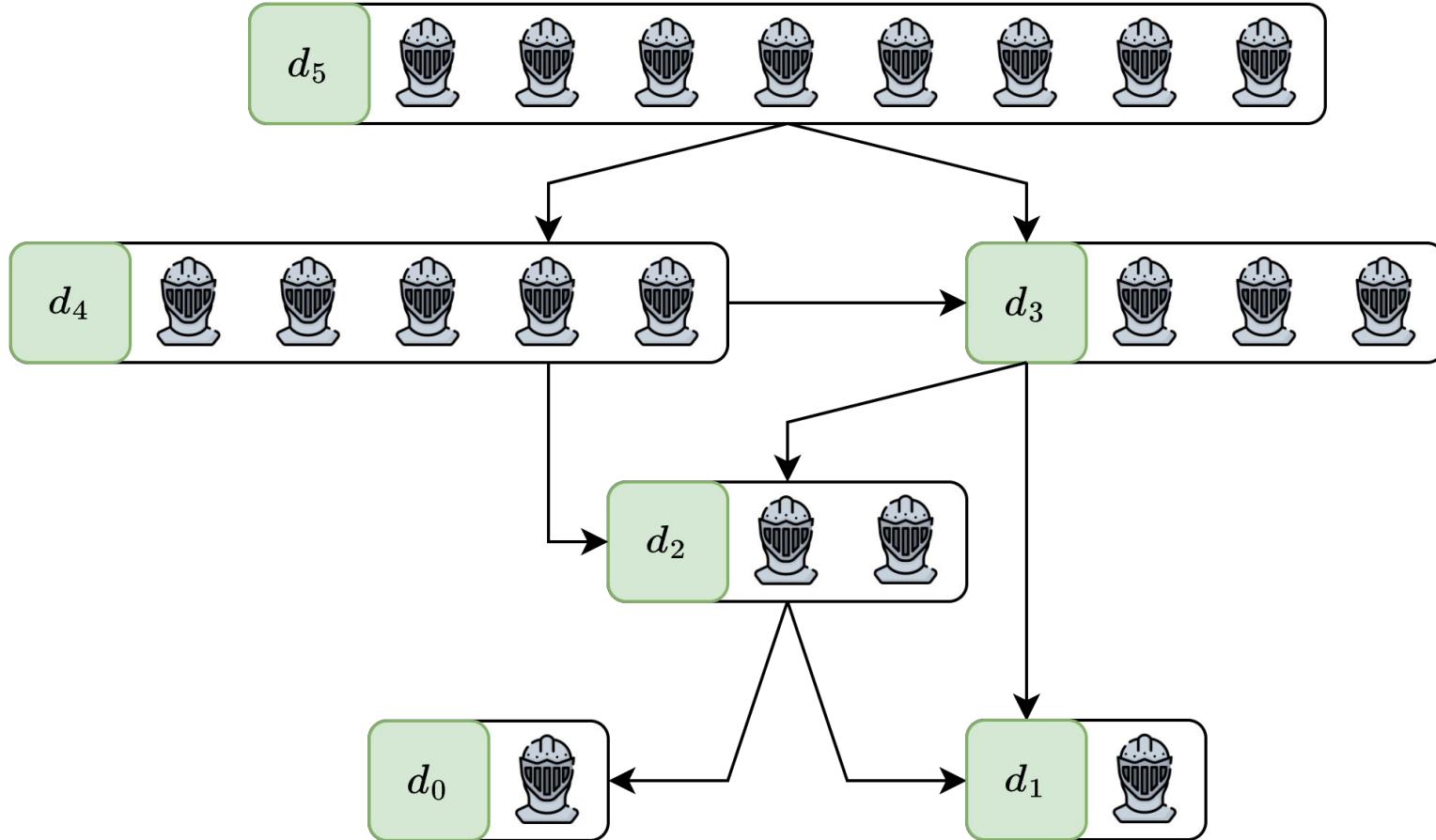


Dia 4





Dependencias de estados



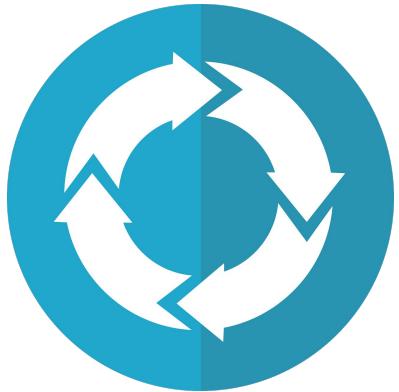
Resolución



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

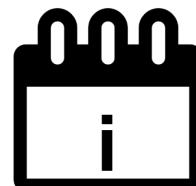


Estado del problema

Tenemos que entender cuál es la información mínima necesaria en cada paso recursivo, que nos permita calcular lo que buscamos.

En nuestro caso la pregunta seria:

¿Que necesito en el paso recursivo para calcular el número de personas para el ejército en el dia d_i?

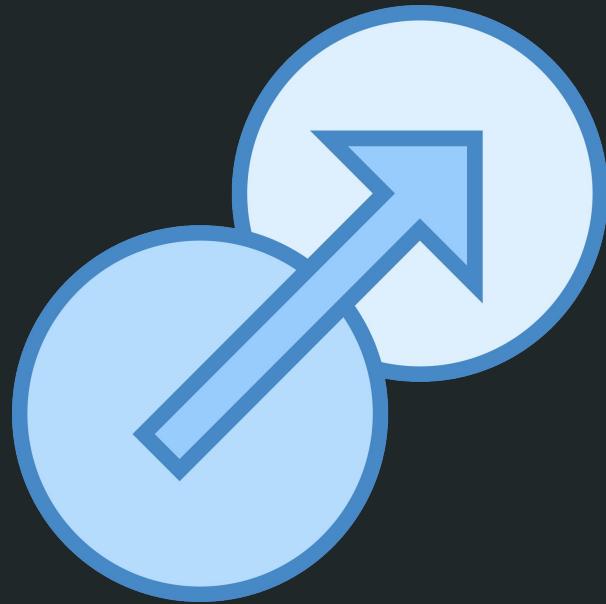


Parametros de funcion recursiva

$$f(i)$$



Tip #2



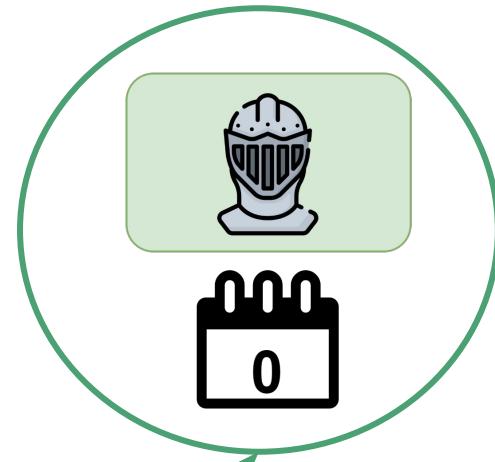
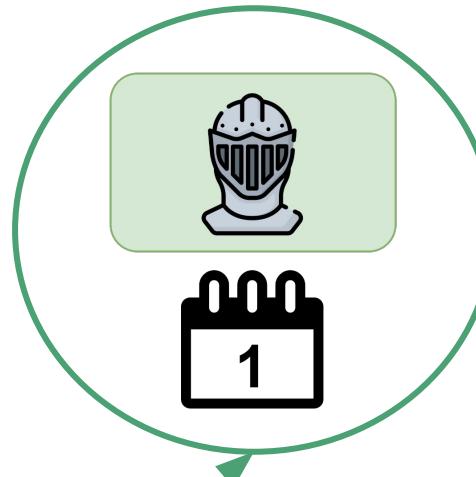
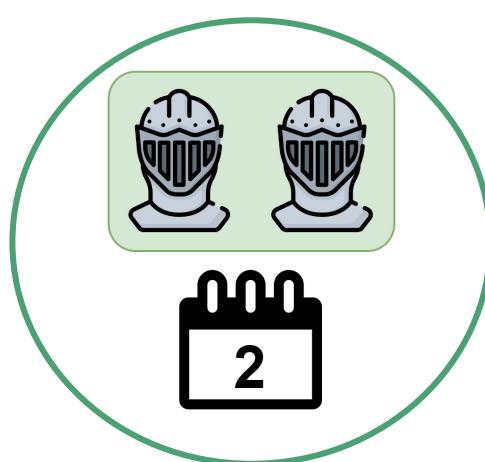
Transiciones desde un estado

Transiciones desde un estado

Dado un estado del problema, queremos ver a que estados podemos transicionar desde este cumpliendo las restricciones.

En nuestro caso la pregunta seria:

¿Dado un dia d_i , a que días debo transicionar para calcular lo que necesito?



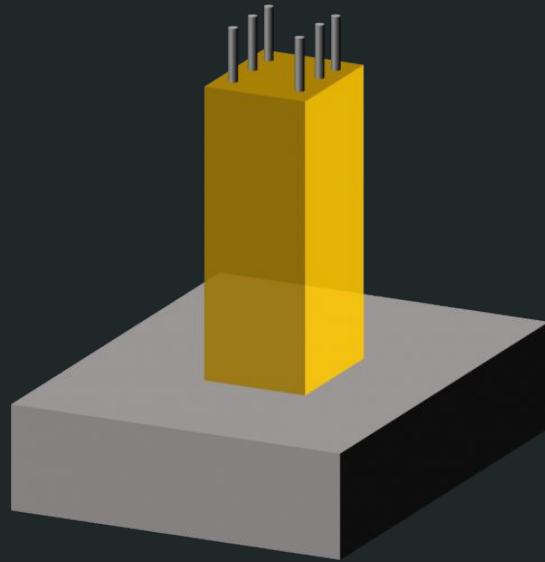
Paso recursivo de f

$$f(i) = \{f(i - 1) + f(i - 2) \quad si \dots$$



Tip #3

Estados base del problema



Estados base del problema

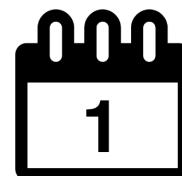
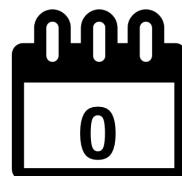
Cuando modelamos la recursión, tenemos que definir casos base.

Estos están ligados a los estados “base” de nuestro problema, donde ya sabemos la respuesta directo y es simple de calcular.

Es clave identificarlos porque, sino lo hacemos, terminamos modelando varios casos innecesarios, que ya están expresados a través de la parte recursiva en realidad (**Mas de esto mas adelante**).

En nuestro caso la pregunta seria:

¿Para que días d_i ya se la respuesta y no requiere ningun calculo complejo ?



Completando con caso base + condiciones para cada caso

$$f(i) = \begin{cases} 1 & i \leq 1 \\ f(i - 1) + f(i - 2) & \text{sin o} \end{cases}$$

Alternativa a casos base redundante

Un ejemplo de casos base innecesarios es el siguiente, donde los estados base son 0, 1 y , donde este ultimo se podria resolver con el paso recursivo en realidad:

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

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:

- El parámetro **i** representa el i-esimo dia para el cual se quiere armar un ejercito.
- El caso base representa los dias donde ya sabemos el numero de personas del ejercito directo, que son **i = 0, 1**.
- El paso recursivo determina por un lado el numero de tropas de los dos dias anteriores al i-esimo, y luego suma los resultados para obtener el del dia i.



Paso 3

Llamado/s que resuelven el problema

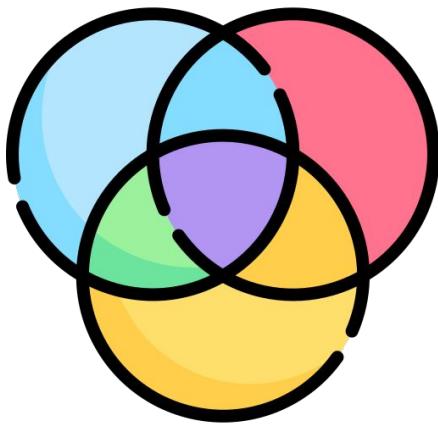
Esta parte es bastante sencilla.

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

¿Qué llamado/s hacemos para resolver esto?

En este caso es super sencillo.

Si queremos obtener el ejército para el n-esimo dia, 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.

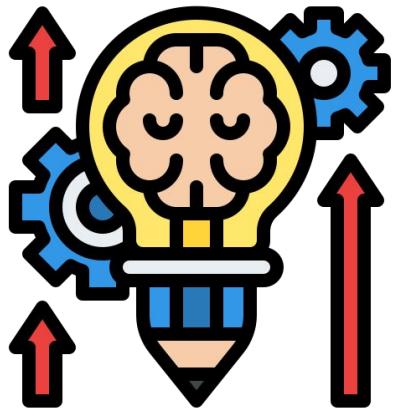
Si esto pasa, significa que estoy viendo estados repetidos si o si, porque hago más llamados recursivos que posibles estados, y programación dinámica me sirve.

¿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 el mejor caso de llamados recursivos de **f** si es facil (sino un estimado de peor caso que se asemeje a algun caso real de **f**):
 - Vemos cuantos llamados recursivos se hacen en cada momento, y cómo cambian los parámetros de **f** en el mejor caso.
 - En este caso es $\Omega(2^{N/2})$, porque cada paso recursivo tiene 2 llamados, baja el **N** en 1 y 2 en cada caso, entonces siempre vamos a tener un arbol con al menos **N/2** de altura.
- 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/2}**, 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(i):
    if i <= 1:
        return 1

    tropas = dp(i-1) + dp(i-2)

    return tropas
```



Tip #4

Pensar la memoria basada en el estado



¿Cómo diseño la memoria para recordar llamados repetidos?

El algoritmo necesita una memoria / memoria 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 #5

“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:

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

Parche en el algoritmo

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


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

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

    tropas = dp(i-1) + dp(i-2)

    # paso 3: guardo el resultado que calculé en este paso en la memoria
    memoria[i] = tropas

return tropas
```



Tip #6



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 de que NO hacer (funciona, si 😅)

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

    diaMenos1, diaMenos2 = None, None
    if memoria[i - 1] != UNDEFINED:
        diaMenos1 = memoria[i - 1]
    else:
        diaMenos1 = dp(i-1)
        memoria[i - 1] = diaMenos1

    if memoria[i - 2] != UNDEFINED:
        diaMenos2 = memoria[i - 2]
    else:
        diaMenos2 = dp(i-2)
        memoria[i - 2] = diaMenos2

    return diaMenos1 + diaMenos2
```



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 este problema, 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) * (tamaño del resultado para cada estado)

En este caso, como sabemos que:

- Tenemos **O(N)** estados.
- Cada estado tiene como resultado un número, que es **O(1)**.

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.

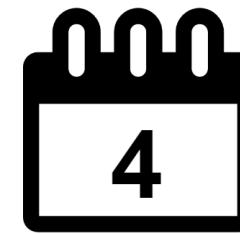
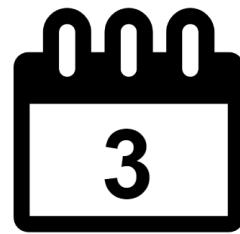
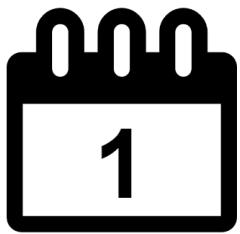
Tomas en cada día puede:

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

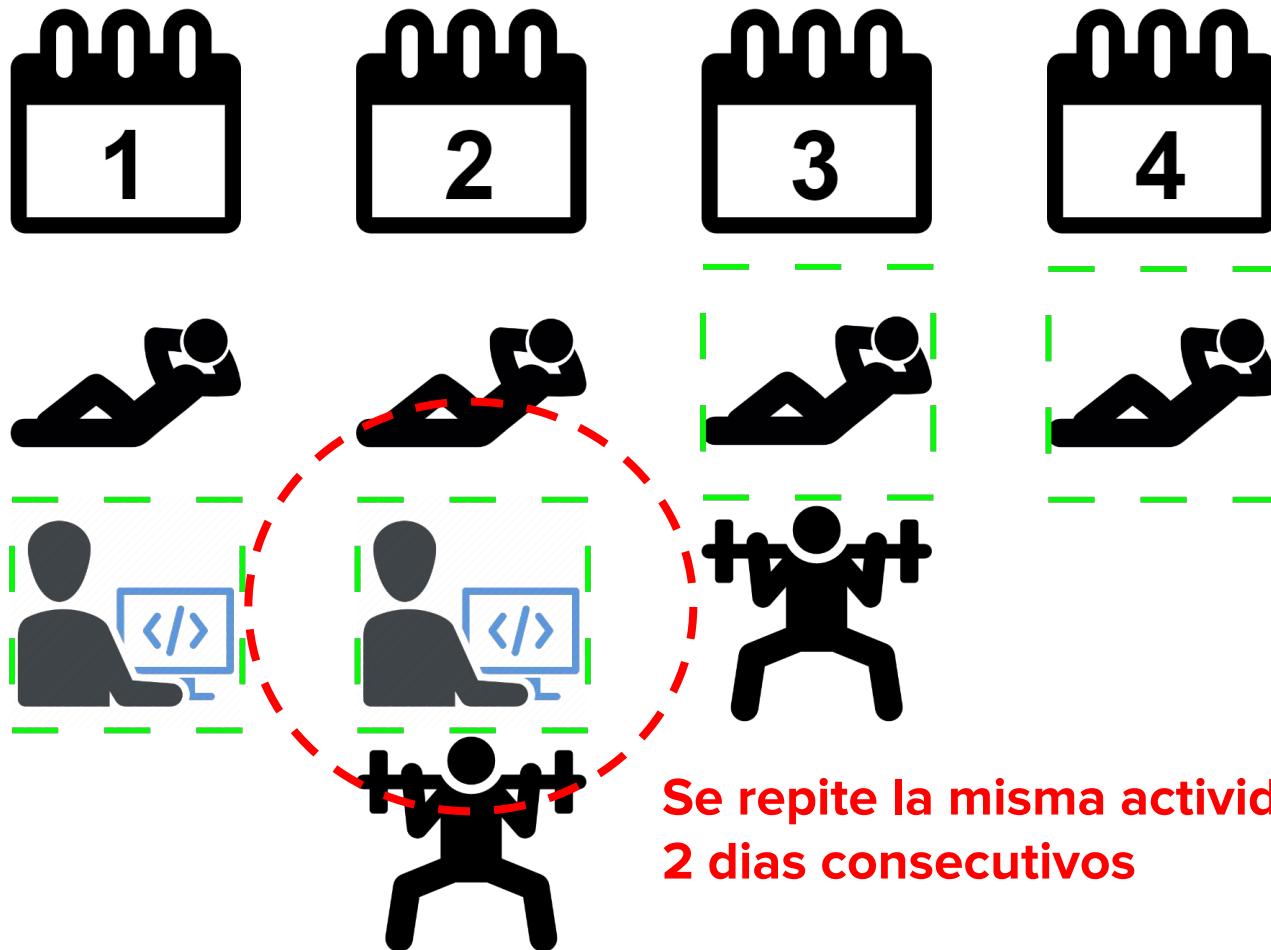
La idea es minimizar la cantidad de días de descanso.

Es un ejercicio de **optimización**.

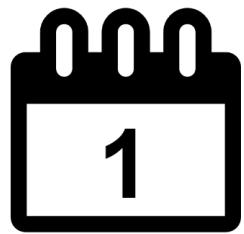
Ejemplo



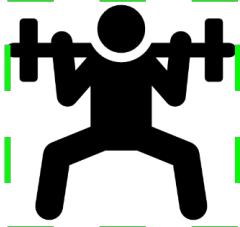
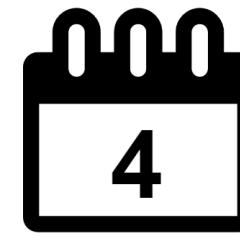
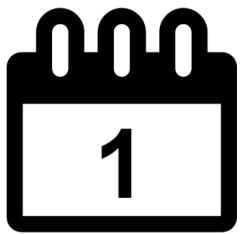
Actividades permitidas



Combinación invalida



Combinación válida pero no óptima, descansó 3 veces.



Combinación válida óptima

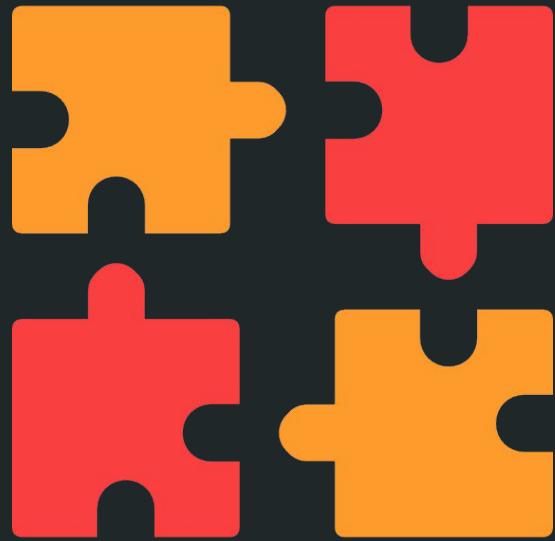
Resolución

Función recursiva



Tip #7

Dividiendo en subproblemas



Dividiendo en subproblemas

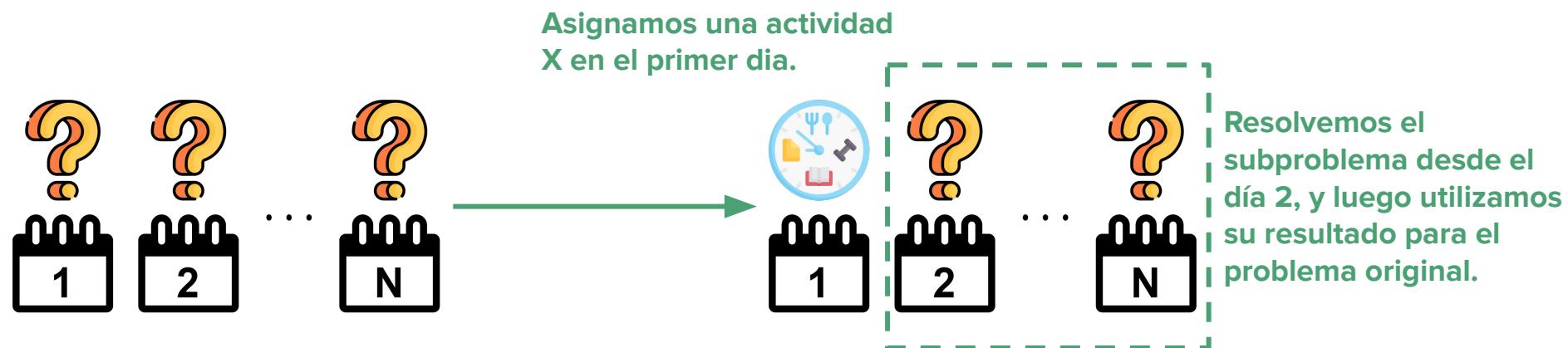
Para entender como modelar el estado y la recursión que resuelve el problema, es clave:

- Entender como dividirlo en subproblemas.
- Combinar los resultados de estos para resolver el original.

Veamos como podemos hacer esto con este problema.

Dividiendo en subproblemas

Queremos asignarle a cada dia una actividad, cumpliendo las restricciones. Esto se puede modelar para resolverlo mediante subproblemas asi:



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 #8

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. (**Ejercicio 11 de la guía 1**)

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**.



Tip #9

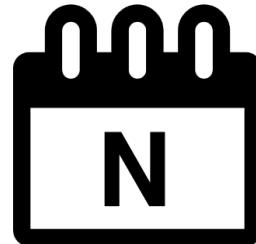
Representando los neutros de la operación.

Representando los neutros de la operación

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 el caso donde no tenemos mas días disponibles para hacer actividades. En ese caso lo que queremos es:

- Retornar 0, porque no podemos hacer ninguna actividad ya que no tenemos días, y tampoco importa la ultima actividad que hicimos porque no vamos a hacer nada mas.



Firma de la función con casos base

Actualizando los casos base:

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

Definiendo casos recursivos de f

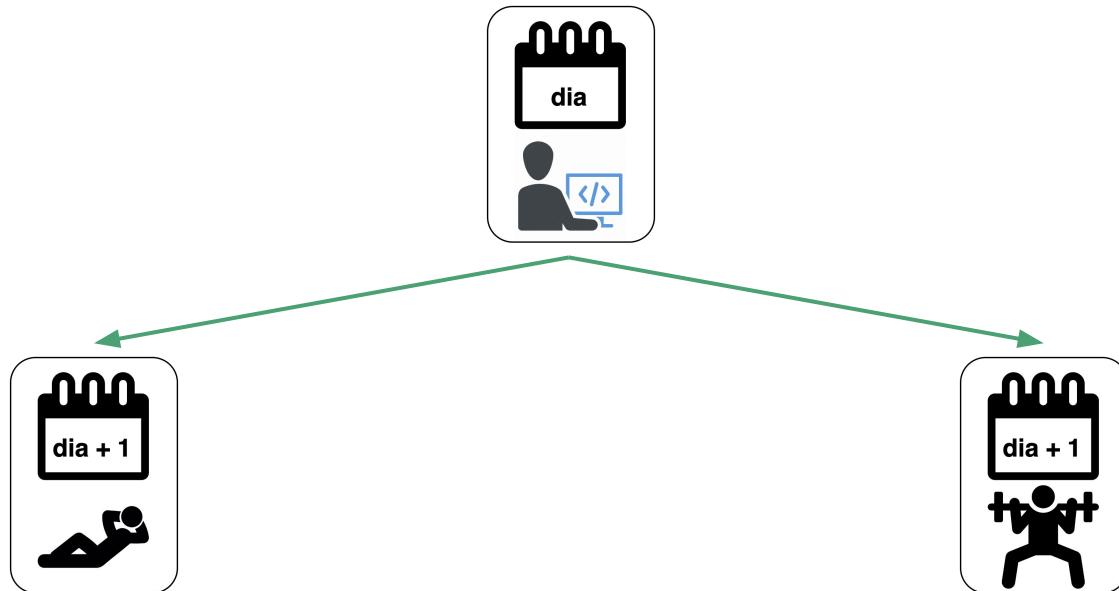
Sabemos que estamos en un día válido, porque ya cubrimos los casos base.

¿Que estados posibles pueden darse si estamos en un día i?

- El dia anterior hice gimnasio.
- El dia anterior hice competencia.
- El dia anterior hice gimnasio.

Veamos qué transiciones tenemos en cada caso.

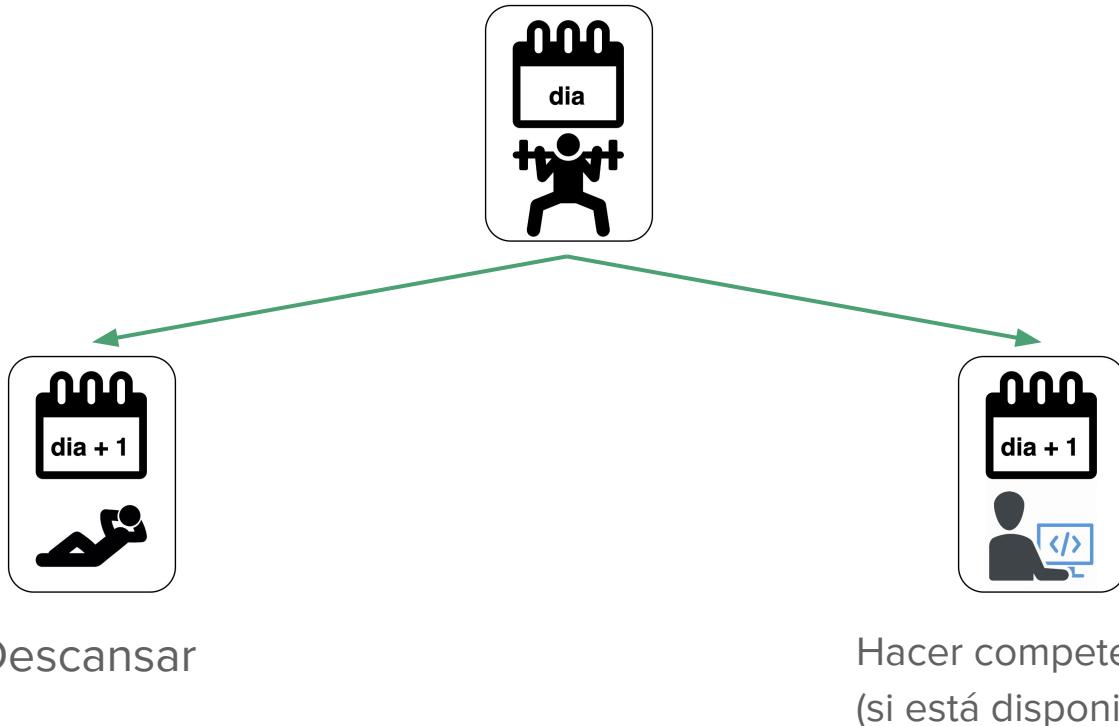
El dia anterior hice competencia



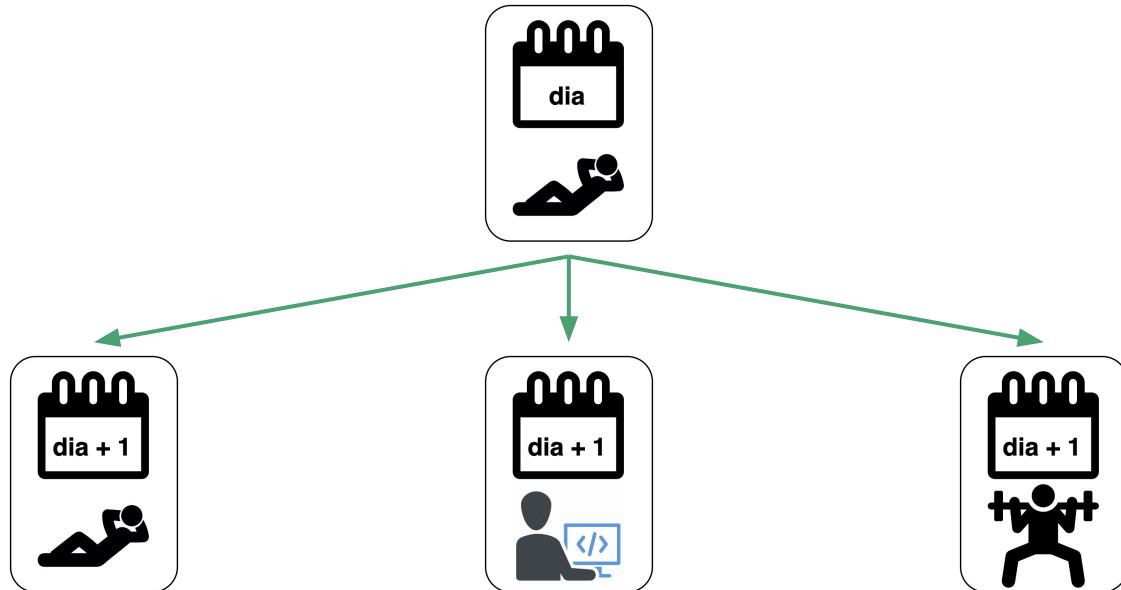
Descansar

Hacer gimnasio (si
está disponible)

El dia anterior hice gimnasio



El dia anterior descansé

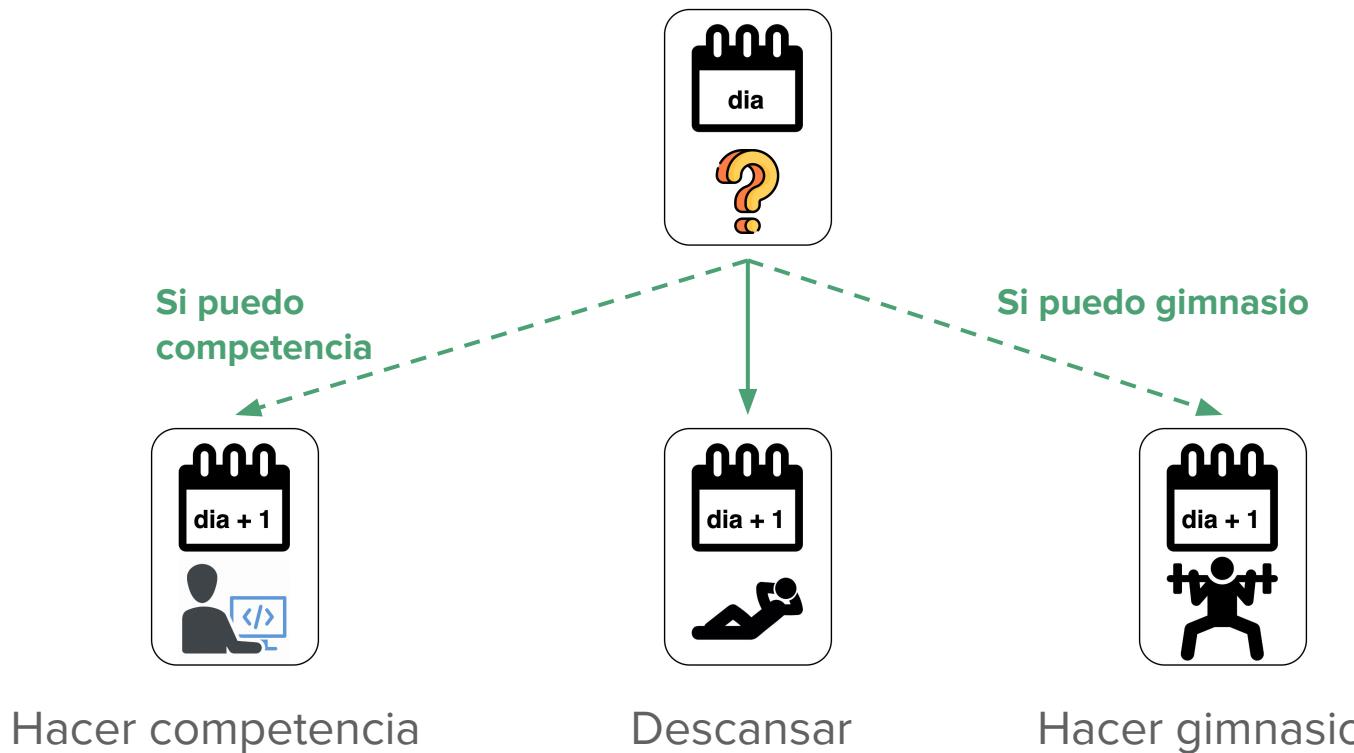


Descansar

Hacer competencia
(si está disponible)

Hacer gimnasio (si
esta disponible)

Otra forma de ver todo lo anterior



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:

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

Otra forma de expresar la función

$$f(\text{dia}, \text{ultAct}) = \begin{cases} 0 & \text{dia} = n \\ \min(\text{desc}(\text{dia}, \text{ultAct}), \text{gym}(\text{dia}, \text{ultAct}), \text{comp}(\text{dia}, \text{ultAct})) & \text{sino} \end{cases}$$

$$\text{desc}(\text{dia}, \text{ultAct}) = 1 + f(\text{dia} + 1, \text{DESC})$$

$$\text{gym}(\text{dia}, \text{ultAct}) = \begin{cases} f(\text{dia} + 1, \text{GYM}) & \text{puedeGym}(\text{dia}, \text{ultAct}) \\ \infty & \text{sino} \end{cases}$$

$$\text{comp}(\text{dia}, \text{ultAct}) = \begin{cases} f(\text{dia} + 1, \text{COMP}) & \text{puedeComp}(\text{dia}, \text{ultAct}) \\ \infty & \text{sino} \end{cases}$$

Aca usamos el neutro del mínimo, que es infinito.

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 podes 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 podes 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 **O(3N) = O(N)** estados.

- Llamados recursivos:
 - Si siempre podemos hacer todas las actividades, seguro tenemos siempre 2 llamados recursivos en cada paso (descanso y otra actividad que no sea la del día anterior).
 - En cada paso recursivo el día sube en 1.

Nos quedan **$\Omega(2^N)$** llamados recursivos.

En este caso podemos afirmar directamente que no necesitamos condiciones sobre **N**, porque siempre se cumple que **$N << 2^N$** .

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

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

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

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

    if memoria[dia][ultAct] != -1:
        return memoria[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))

    memoria[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 minimo.
 - Tenemos luego $O(1)$ aca.

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

Caesar's Legions



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

Enunciado

Enunciado

Al famoso general Caesar le gusta poner en línea sus tropas, que son **patos** y **dodos**:

- Tiene un ejército de **P** patos, y **D** dodos.
- No le gusta que la forma de poner en línea a sus tropas incumpla con que:
 - Hay más de **MP** patos en fila seguidos.
 - Hay más de **MD** dodos en fila seguidos.
- Los patos son indistinguibles entre ellos, al igual que los dodos.

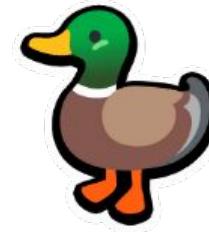
Le interesa encontrar todas las combinaciones posibles de formar esta línea.

Es un ejercicio de **conteo**.

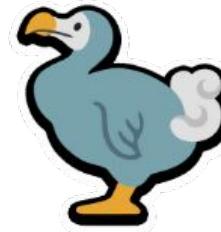
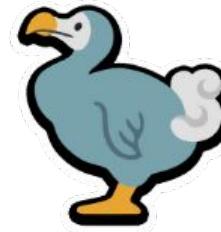
Ejemplo

Datos iniciales

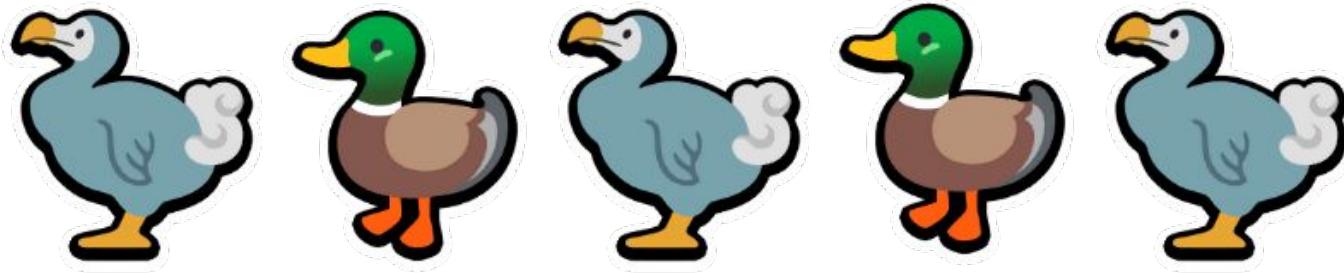
P = 3
MP = 2



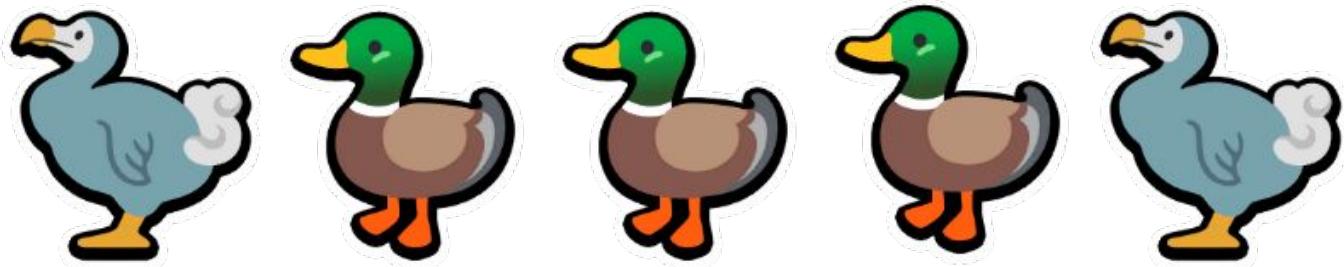
D = 2
MD = 1



Combinaciones inválidas para P = 3, MP = 2, D = 2 y MD = 1

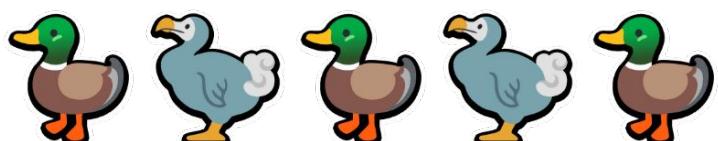


Mas dodos de los que existen



Exceso de patos consecutivos

Combinaciones válidas para MP = 2 y MD = 1



Resolución

Función recursiva

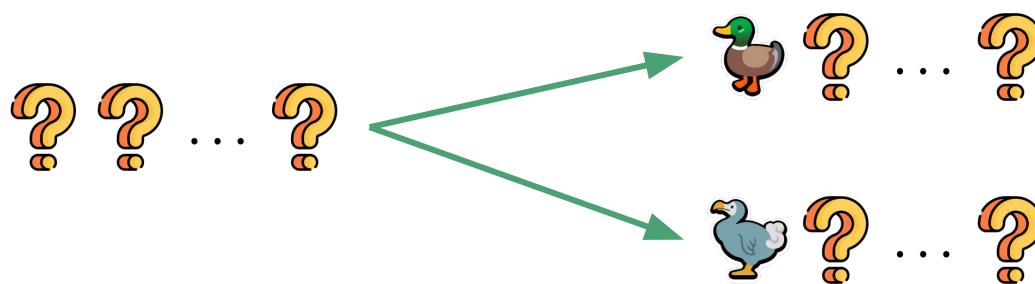
Modelemos el estado del problema

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

Pensemos en las condiciones del problema:

- Tenemos **P** patos y **D** dodos que tenemos que ubicar en linea.
- No podemos poner más de **MP** patos seguidos, y más de **MD** dodos tampoco.

Pensemos en la forma de las combinaciones y los subproblemas que aparecen:



Probamos poner primero un pato y calculamos todas las combinaciones que empiezan con este.

Idem pero con dodo.

Modelemos el estado del problema

Entonces, la información que necesitamos en todo momento para saber que podemos utilizar es:

- Si nos quedan tropas para poner todavía.
- Cuantos patos y dodos nos quedan por ubicar de los **P** y **D** respectivamente.
- Cuantos patos y dodos podemos seguir ubicando consecutivamente sin excedernos de **MP** y **MD** respectivamente.

Firma de la función

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

Cantidad de tropas totales restantes para ubicar

$$f(t, n_P, n_D, k_P, k_D)_{MP,MD} = \{$$

Cantidad de tropas consecutivos que pueden ponerse de pato y dodo, considerando los que quedan disponibles de cada uno.

Cantidad de tropas restantes de cada uno, pato y dodo.

Nota: Los subíndices **MP** y **MD** hacen alusión a los límites **MP** y **MD**, que podemos usar en todo momento, y nos evitamos pasarlas como parámetro.

Definiendo casos base

Tenemos un total de tropas igual a **P + D**, que va decrementando cada vez que ponemos un pato o dodo. ¿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 patos o dodos?
 - No, porque deberían ser 0 ya que el total lo es.
 - ¿Importa lo que podemos poner de patos o dodos 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:

$$f(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} 1 & t = 0 \\ \dots \end{cases}$$

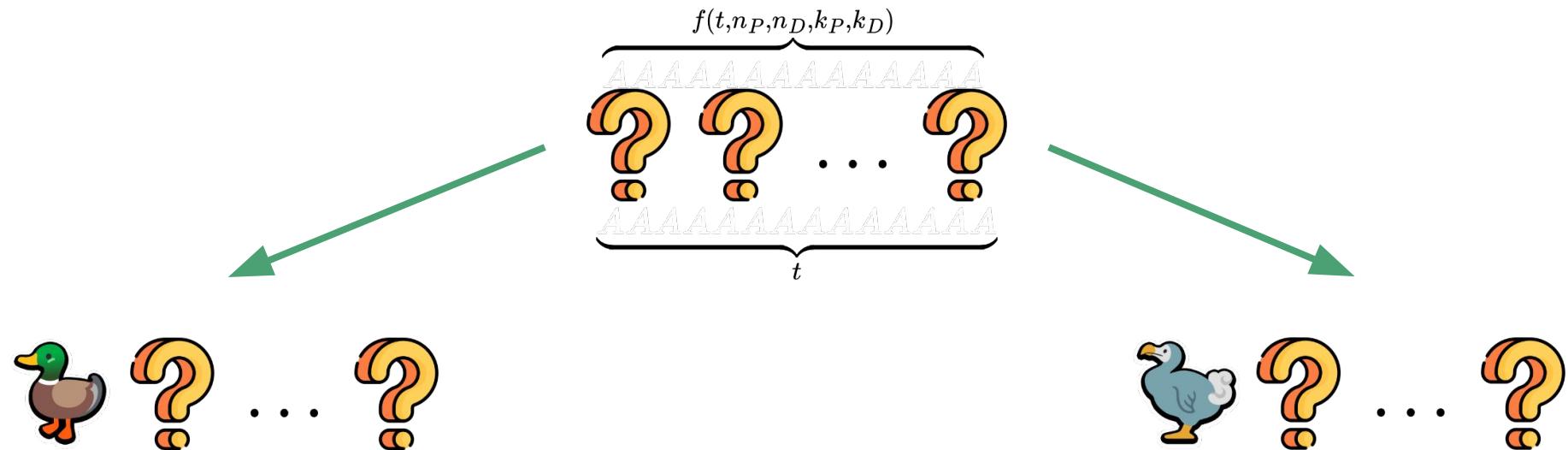
Definiendo casos recursivos

Sabemos que $t > 0$, y luego hay 1 tropa de algún tipo, porque ya cubrimos los casos base donde $t = 0$.

- ¿A que estados posibles puedo transicionar en base a los patos y dodos que tengo disponibles y los permitidos poner consecutivamente?
- ¿Como puedo calcular el resultado en cada caso?

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

Definiendo casos recursivos



Firma de la función con casos recursivos

Podemos pensar en sumar las combinaciones de poner un pato o un dodo:

$$f(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} 1 & t = 0 \\ \text{ponerPATO}(t, n_P, n_D, k_P, k_D)_{MP,MD} + \text{ponerDODO}(t, n_P, n_D, k_P, k_D)_{MP,MD} & \text{sino} \end{cases}$$

¿Cómo definimos **ponerPATO** y **ponerDODO**?

$$f(t-1, n_P-1, n_D, k_P-1, MD)$$

$\overbrace{AAAAAA \dots AAAA}^{t-1}$



$n_P = n_P - 1$

$$f(t, n_P, n_D, k_P, k_D)$$

$\overbrace{AAAAAA \dots AAAA}^t$



$$f(t-1, n_P, n_D-1, MP, k_D-1)$$

$\overbrace{AAAAAA \dots AAAA}^{t-1}$



$n_D = n_D - 1$

$$n_P = n_P - 1 \qquad k_P = k_P - 1$$

$$n_P = n_P \qquad \qquad k_P = MP$$

$$n_D = n_D \qquad \qquad k_D = MD$$

$$n_D = n_D - 1 \qquad k_D = k_D - 1$$

Firma de la función con casos recursivos V1

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

$$f(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} 1 & t = 0 \\ \text{ponerPATO}(t, n_P, n_D, k_P, k_D)_{MP,MD} + \text{ponerDODO}(t, n_P, n_D, k_P, k_D)_{MP,MD} & \text{sino} \end{cases}$$

$$\text{ponerPATO}(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} f(t - 1, n_P - 1, n_D, k_P - 1, MD)_{MP,MD} & n_P > 0 \wedge k_P > 0 \\ 0 & \text{sino} \end{cases}$$

$$\text{ponerDODO}(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} f(t - 1, n_P, n_D - 1, MP, k_D - 1)_{MP,MD} & n_D > 0 \wedge k_D > 0 \\ 0 & \text{sino} \end{cases}$$

Firma de la función con casos recursivos V2

Hay algo que podemos cambiar levemente de la función f.

- ¿Que simboliza **kp** y **kd** para nosotros?
 - La cantidad de patos y dodos que podemos seguir poniendo consecutivamente.
- Esto último cubre 2 aspectos:
 - No pasarse de la cantidad permitida de tropas del mismo tipo consecutivos.
 - No pasarse de la cantidad disponible de ese tipo de tropa.

Luego, si pensamos en esto de esa forma, podemos simplificar un poco f, ¿Como?

- Cada vez que reiniciamos el kp / kd, no le asignamos directo el máximo **MP** / **MD**, sino...
 - El minimo entre **np** / **nd** (lo que queda de ese tropa) y **MP** / **MD** (lo máximo que podríamos poner consecutivamente).

Firma de la función con casos recursivos V2

Nueva versión con la idea anterior (**Modificaciones en rojo**):

$$f(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} 1 & t = 0 \\ \text{ponerPATO}(t, n_P, n_D, k_P, k_D)_{MP,MD} + \text{ponerDODO}(t, n_P, n_D, k_P, k_D)_{MP,MD} & \text{sino} \end{cases}$$

$$\text{ponerDODO}(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} f(t - 1, n_P, n_D - 1, \min(n_P, MP), k_D - 1)_{MP,MD} & k_D > 0 \\ 0 & \text{sino} \end{cases}$$

$$\text{ponerPATO}(t, n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} f(t - 1, n_P - 1, n_D, k_P - 1, \min(n_D, MD))_{MP,MD} & k_P > 0 \\ 0 & \text{sino} \end{cases}$$

Todo muy lindo pero... no funciona



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



Tip #10



Reduciendo estados eliminando variables redundantes

Reduciendo estados eliminando variables redundantes

A veces tenemos situaciones donde hay parámetros de nuestro estado que son redundantes porque se deducen del resto, o representan cierta información ocupando más espacio del necesario.

Veamos qué parámetros podemos eliminar / modificar / reemplazar.

Reduciendo estados eliminando variables redundantes

Hay un parametro que rápidamente notamos que es redundante, porque se puede expresar a partir de otros...

¿Tiene sentido que mantengamos el parametro **t** que es la suma de patos y dodos?

- No, porque podemos deducirlo a partir de cuantos nos quedan dc e cada tipo de tropa particular.

Función recursiva f con reducción de parámetros

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

$$f(n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} 1 & t = 0 \\ \text{ponerPATO}(n_P, n_D, k_P, k_D)_{MP,MD} + \text{ponerDODO}(n_P, n_D, k_P, k_D)_{MP,MD} & \text{sino} \end{cases}$$

$$\text{ponerPATO}(n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} f(n_P - 1, n_D, k_P - 1, \min(n_D, MD))_{MP,MD} & k_P > 0 \\ 0 & \text{sino} \end{cases}$$

$$\text{ponerDODO}(n_P, n_D, k_P, k_D)_{MP,MD} = \begin{cases} f(n_P, n_D - 1, \min(n_P, MP), k_D)_{MP,MD} & k_D > 0 \\ 0 & \text{sino} \end{cases}$$

Reduciendo estados eliminando variables redundantes

¿Terminamos?

- ¡JA! No.

¿Vale la pena que tengamos las variables **kp** y **kd** a la vez?

- Nunca va a pasarnos que usemos las 2, porque solo importa mantener el conteo del último tropa que viene repitiéndose. Cambiemos el nombre de la variable entonces a solo **k**.

¿Podemos sacar una de las 2 y ya?

- No, porque el contador que nos queda no sabemos si se refiere a que venimos acumulando patos o dodos.
- Necesitamos entonces un parámetro que nos diga si es pato o dodo. Llámemoslo **ultTropa**.

Función recursiva **f** con reducción de parámetros

Veamos como actualizar f con la idea de la ultima tropa:

$$f(n_P, n_D, k, \text{ultTropa})_{MP,MD} = \begin{cases} 1 & n_P + n_D = 0 \\ \text{ponerPATO}(n_P, n_D, k, \text{ultTropa})_{MP,MD} + \text{ponerDODO}(n_P, n_D, k, \text{ultTropa})_{MP,MD} & \text{sino} \end{cases}$$

$$\text{ponerPATO}(n_P, n_D, k, \text{ultTropa})_{MP,MD} = \begin{cases} f(n_P - 1, n_D, k - 1, \text{PATO})_{MP,MD} & \text{ultTropa} = \text{PATO} \wedge k > 0 \\ f(n_P - 1, n_D, \min(n_p, MP) - 1, \text{PATO})_{MP,MD} & \text{ultTropa} = \text{DODO} \wedge n_P > 0 \\ 0 & \text{sino} \end{cases}$$

$$\text{ponerDODO}(n_P, n_D, k, \text{ultTropa})_{MP,MD} = \begin{cases} f(n_P, n_D - 1, k - 1, \text{DODO})_{MP,MD} & \text{ultTropa} = \text{DODO} \wedge k > 0 \\ f(n_P, n_D - 1, \min(n_D, MD) - 1, \text{DODO})_{MP,MD} & \text{ultTropa} = \text{PATO} \wedge n_D > 0 \\ 0 & \text{sino} \end{cases}$$

Reduciendo estados eliminando variables redundantes

¿Ahora si terminamos?

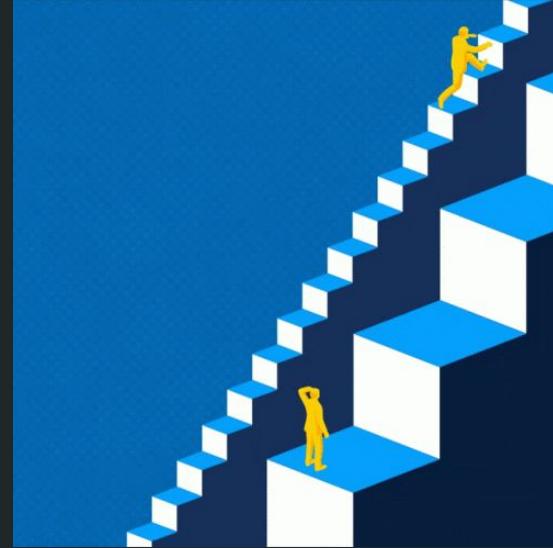
- Podría decirse, pero vamos a hacer un truco más.

¿Es necesario que en cada paso recursivo pongamos 1 sola tropa?

- Repensemose el esquema de recursion



Tip #11



Pequeños pasos VS Grandes pasos

Pequeños pasos VS Grandes pasos

Al momento de hacer la recursión, puede pasar que podamos enfocarlo de distintas formas.

Una elección clave a veces es si vamos a dar pequeños pasos o grandes pasos, con respecto a la variable que va decrementando en la recursion:

1. Pequeños: Bajamos en 1.
2. Grandes: Bajamos en 1, 2, ... X, probando todas las opciones posibles.

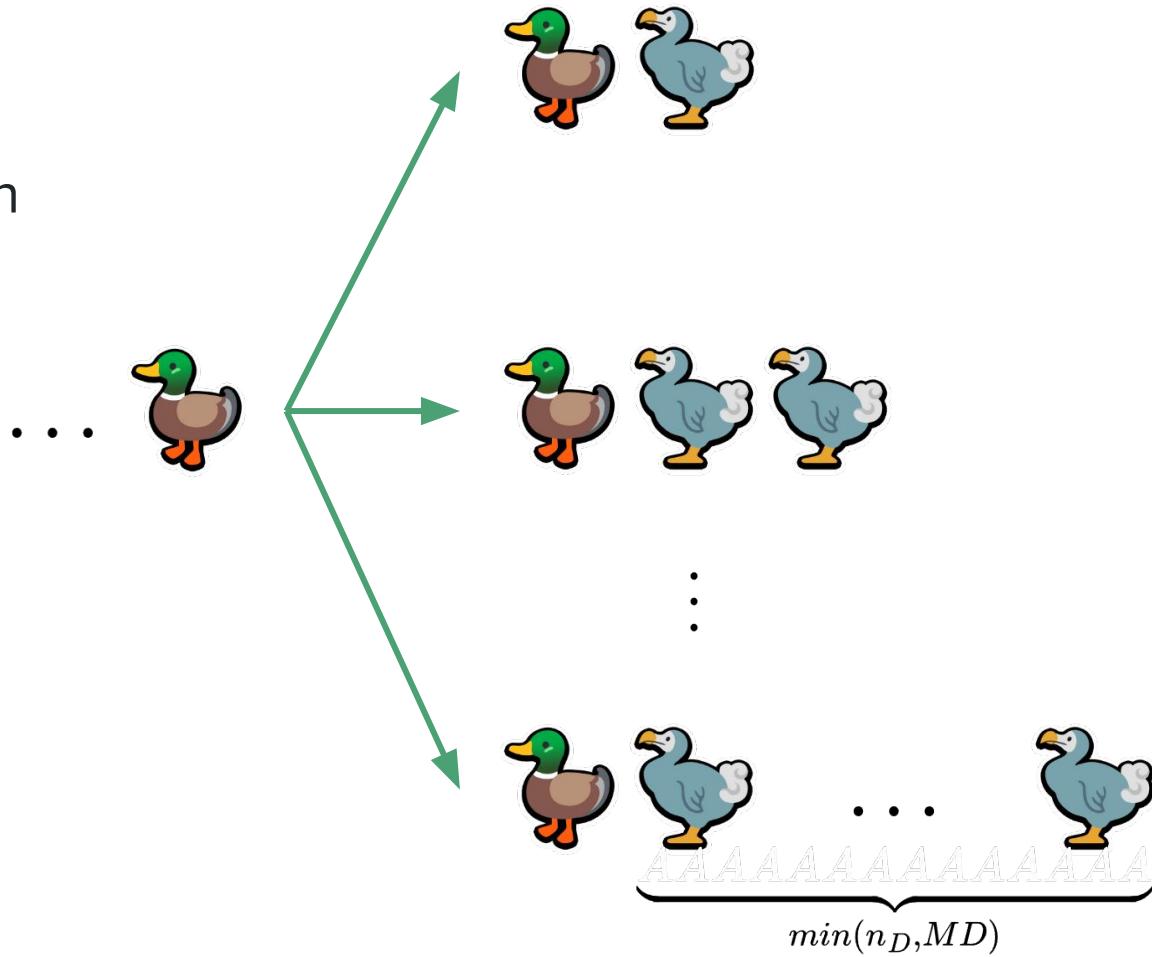
A veces puede ser mucho mas simple hacer la recursión con la segunda.

Reformulando la recursion

Hasta ahora veníamos utilizando solo 1 tropa en cada paso recursivo, pero...

¿Es necesario que en cada paso recursivo pongamos 1 sola tropa?

- No, podemos directamente decidir en cada paso la cantidad de tropas iguales que vamos a poner, para luego pasar a la otra.
- Con esto podremos remover un parámetro de f , a cambio de complejizar el paso recursivo. ¿Cuál es?
 - El k .



Función recursiva f con reducción de parámetros

Así se actualiza f con este ultimo truco:

$$f(n_P, n_D, \text{ultTropa})_{MP,MD} = \begin{cases} 1 & (n_P + n_D) = 0 \\ \text{ponerPATO}(n_P, n_D, \text{ultTropa}) + \text{ponerDODO}(n_P, n_D, \text{ultTropa}) & \text{sino} \end{cases}$$

$$\text{ponerPATO}(n_P, n_D, \text{ultTropa})_{MP,MD} = \begin{cases} \sum_{i=1}^{\min(n_P, MP)} f(n_P - i, n_D, \text{PATO})_{MP,MD} & \text{ultTropa} = \text{DODO} \\ 0 & \text{sino} \end{cases}$$

$$\text{ponerDODO}(n_P, n_D, \text{ultTropa})_{MP,MD} = \begin{cases} \sum_{i=1}^{\min(n_D, MD)} f(n_P, n_D - i, \text{DODO})_{MP,MD} & \text{ultTropa} = \text{PATO} \\ 0 & \text{sino} \end{cases}$$

Semántica de función

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

Nuestra función **f** te dice que:

- En caso de no tener más tropas, solo puedes conseguir 1 combinación, y es no poner nada.
- Si la última tropa que se usó fue pato:
 - Suma todas las formas de ordenar el resto de tropas que quedan, poniendo al frente de estas una cantidad de dodos valida mayor a 0 para luego seguir con patos devuelta.
- Si la última tropa que se usó fue dodo:
 - Idem que antes pero poniendo patos al frente.

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 **P** patos y **D** dodos inicialmente.
- Inicialmente no pusimos ningún tipo de tropa.
 - ¿Cómo permitimos esto mediante el parámetro **ultTropa**?
 - Tenemos que hacer que empiece con pato o con dodo al principio.

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

$$f(P, D, \text{PATO}) + f(P, D, \text{DODO})$$

Superposición de problemas

¿Condiciones para superposición?

- Estados: Combinación de tropas restantes de patos y dodos y del tipo de la última tropa:
 - La **ultTropa** puede ser solo **PATO** o **DODO**.
 - Hay **P** patos y **D** dodos.

Tenemos **O(2^{PD})** estados.

- Llamados recursivos:
 - En cada paso recursivo casi siempre hay mínimo 2 llamadas (si hay al menos 2 patos / dodos, y **MP, MD > 1**).
 - Como en cada paso casi siempre ponemos 1-2 animales, el arbol tiene aproximadamente **P + D** de altura.

Nos quedan **$\Omega(2^{P+D})$** llamados recursivos.

¿Cuándo pasa que **PD <<< 2^{P+D}**?

- Si **P+D > 1** y vale nuestra suposición de **MP, MD > 1**, seguro ocurre.

$$f(n_P, n_D, \text{ultTropa})_{MP,MD} = \begin{cases} 1 & (n_P + n_D) = 0 \\ \text{ponerPATO}(n_P, n_D, \text{ultTropa}) + \text{ponerDODO}(n_P, n_D, \text{ultTropa}) & \text{sino} \end{cases}$$

$$\text{ponerPATO}(n_P, n_D, \text{ultTropa})_{MP,MD} = \begin{cases} \sum_{i=1}^{\min(n_P, MP)} f(n_P - i, n_D, \text{PATO})_{MP,MD} & \text{ultTropa} = \text{DODO} \\ 0 & \text{sino} \end{cases}$$

$$\text{ponerDODO}(n_P, n_D, \text{ultTropa})_{MP,MD} = \begin{cases} \sum_{i=1}^{\min(n_D, MD)} f(n_P, n_D - i, \text{DODO})_{MP,MD} & \text{ultTropa} = \text{PATO} \\ 0 & \text{sino} \end{cases}$$

Algoritmo de función f

```
def f(np, nd, ultAnimal):
    if np == 0 and nd == 0:
        return 1

    if memoria[np][nd][ultAnimal] != -1:
        return memoria[np][nd][ultAnimal]

    combinaciones = ponerPATO(np,nd,ultAnimal) + ponerDODO(np,nd,ultAnimal)

    memoria[np][nd][ultAnimal] = combinaciones
    return combinaciones

memoria = [[[ -1 for _ in range(2)] for _ in range(D+1)] for _ in range(P+1)] # memoria[P][D][2]
```

```
def ponerPATO(np, nd, ultAnimal):
    combinaciones = 0
    if ultAnimal != PATO:
        for i in range(1, 1 + min(np, MP)): # pruebo cada cantidad de patos posible consecutivos
            combinaciones = (combinaciones + f(np - i, nd, PATO))
    return combinaciones

def ponerDODO(np, nd, ultAnimal):
    combinaciones = 0
    if ultAnimal != DODO:
        for i in range(1, 1 + min(nd, MD)): # pruebo cada cantidad de dodos posible consecutivos
            combinaciones = (combinaciones + f(np, nd-i, DODO))
    return combinaciones
```

Complejidad del algoritmo

Determinando complejidad de f

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

- Estados: Hay P y D de cada tropa, y solo 2 tipos de tropa:
 - Entonces tenemos $O(2PD)$.
- Costo por estado: Dependiendo de la última tropa, llamamos en cada paso una cantidad de veces igual al mínimo entre lo que quedan de tropas y el máximo permitido consecutivo:
 - Aca nos quedara $O(\max(\min(P, MP), \min(D, MD)))$.

Entonces como costo total tenemos $O(PD * \max(\min(P, MP), \min(D, MD)))$.

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:

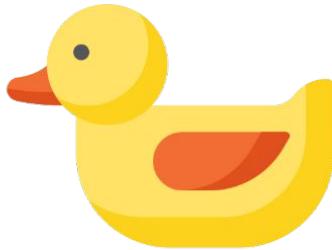
- t : El tiempo que estima el sabueso que le tomara salvarlo.
- d : El tiempo que estima el sabueso a partir del cual se quema el artículo y no sirve más.
- p : El valor del artículo para el sabueso.

El sabueso quiere maximizar la suma de los valores de los artículos que puede salvar, considerando que:

- Salva un artículo luego de otro.
- Si un artículo **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

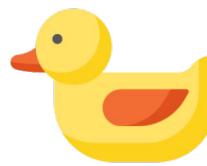
Ejemplo válido no óptimo 1



0



0



hourglass	3
flame	7
coin bag	4



hourglass	7
flame	10
coin bag	6



hourglass	2
flame	6
coin bag	5

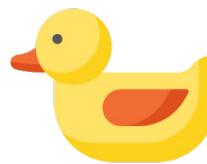
Ejemplo válido no óptimo 1



3



4



hourglass	3
flame	7
coin bag	4



hourglass	7
flame	10
coin bag	6



hourglass	2
flame	6
coin bag	5



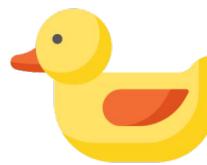
Ejemplo válido no óptimo 2



0



0



hourglass	3
flame	7
coin bag	4



hourglass	7
flame	10
coin bag	6



hourglass	2
flame	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



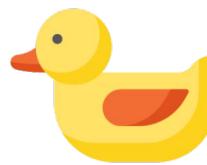
Ejemplo válido óptimo



0



0



hourglass	3
flame	7
coin bag	4



hourglass	7
flame	10
coin bag	6



hourglass	2
flame	6
coin bag	5

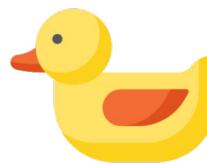
Ejemplo válido óptimo



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 siempre si ningun artículo llega a quemarse.
 - En cada paso recursivo bajamos de a un artículo.

Nos quedan $\Omega(2^n)$ 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 .

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

Algoritmo de función f

```
def dp(i, tiempoActual):  
    if i == n:  
        return 0  
  
    if memoria[i][tiempoActual] != -1:  
        return memoria[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)  
  
    memoria[i][tiempoActual] = maxValor  
    return maxValor  
  
memoria = [[-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 **$O(n^2)$** .



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

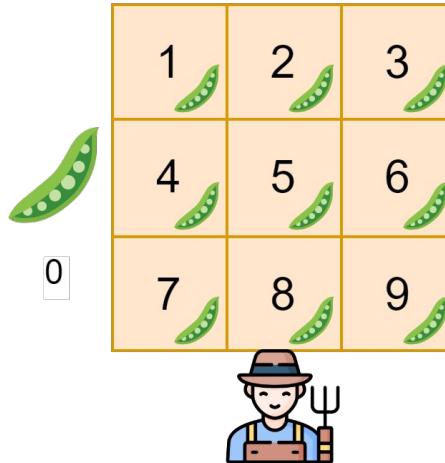
Ejemplo

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

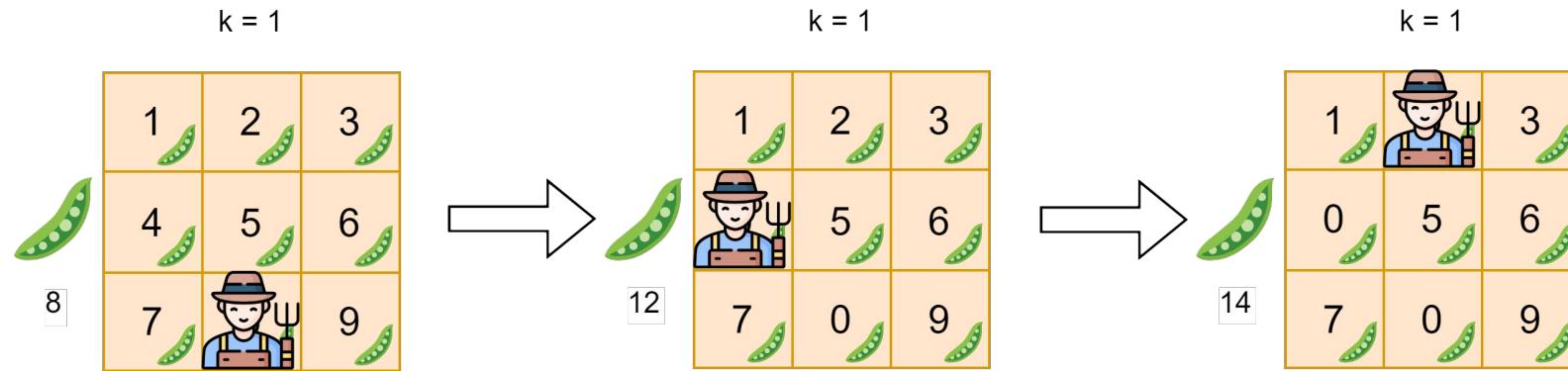
$k = 1$

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

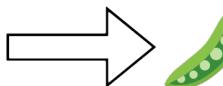
$k = 1$

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

Definiendo casos base

Pensemos que retornar en cada caso de arvejas:

1. Valido: Es fácil, retornamos 0 porque no podemos obtener más arvejas desde acá.
2. Invalido: 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.

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

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

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

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

donde $sigArvMod = (arvMod + terr[x][y]) \bmod (k + 1)$

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 memoria[y][x][arvMod] != -1:
        return memoria[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]
    memoria[y][x][arvMod] = maxArvs
    return maxArvs

memoria = [[[-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!

THE END

