# Programación dinámica

## Top Down

Joaquín Laks - Lautaro Lasorsa - Luciana Skakovsky - Ezequiel Companeetz

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

$2^{do}$ Cuatrimestre de 2025

# Kahoot Time

Ahora vamos a repasar algunas propiedades útiles de programación dinámica y backtracking.
Para eso van a tener que entrar ~~https://kahoot.it/~~ e ingresar el código escrito en el pizarrón.

## Fibonacci

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

## Solución Top Down

Sea $M \in \mathbb{N}^n$ indefinido en todas las posiciones
F(i):

- Si M[i] está definido, retornar M[i].
- Si no, y $i \leq 1$, retornar 1.
- Si no, guardar F(i-1) + F(i-2) en M[i] y retornarlo.

Complejidad temporal: $O(n)$, espacial: $O(n)$.

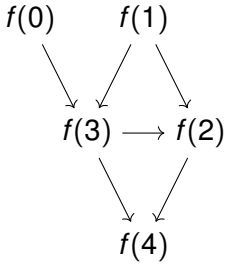
# Top down $\Rightarrow$ Bottom Up

Estábamos resolviendo el problema de la misma forma que la función recursiva, empezando por el estado que queríamos ($f(n)$) y recorriendo los estados necesarios hasta tener la información suficiente para calcularlo.

# Top down $\Rightarrow$ Bottom Up

¿Y si primero resolvemos las cosas que no dependen de nada?

# Órden de dependencia para $n = 4$




¿En qué orden podemos recorrer estas dependencias?

## Órden de dependencias

Si resolvemos en el orden $f(1) \to f(2) \to \cdots \to f(n)$, se cumple que cuando llegamos a $f(i)$, todas sus dependencias ya están calculadas.

Recordemos el caso recursivo: $f(n) = f(n-1) + f(n-2)$. Todos los llamados son a un $n$ más chico.

## Solución Bottom Up

Sea $M \in \mathbb{N}^{n+1}$ con M[0] = M[1] = 1.
F(n):

- Para *i* desde 2 hasta *n* inclusive:
  - M[i] $\leftarrow$ M[i-1] + M[i-2]
- Retornar M[n]

# Optimización de memoria

Ahora que estamos recorriendo los estados de forma ordenada, nos podemos dar cuenta de que el estado $f(i)$ solo depende de los dos anteriores, y los previos a esos nunca se van a usar.

# Optimización de memoria

## Solución Bottom Up

F(n):

- último, anteúltimo $\leftarrow 1$.
- Repetir $n - 2$ veces:
  - actual $\leftarrow$ último + anteúltimo
  - anteúltimo $\leftarrow$ último
  - último $\leftarrow$ actual
- Retornar último

# Optimización de memoria

## Solución Bottom Up

F(n):

- último, anteúltimo $\leftarrow 1$.
- Repetir $n - 2$ veces:
  - actual $\leftarrow$ último + anteúltimo
  - anteúltimo $\leftarrow$ último
  - último $\leftarrow$ actual
- Retornar último

Complejidad temporal: $O(n)$
Complejidad espacial: $O(1)$

# Astro Trade

## Astro Trade

Lu se dedica a la compra de asteroides. Sea $p \in \mathbb{N}^n$ tal que $p_i$ es el precio de un asteroide el $i$-ésimo día en una secuencia de $n$ días. Lu quiere comprar y vender asteroides durante esos $n$ días de manera tal de obtener la mayor ganancia neta posible. Debido a las dificultades que existen en el transporte y almacenamiento de asteroides, Lu puede comprar a lo sumo un asteroide cada día, puede vender a lo sumo un asteroide cada día y comienza sin asteroides. Además, el Ente Regulador Asteroidal impide que Lu venda un asteroide que no haya comprado. Queremos encontrar la máxima ganancia neta que puede obtener Lu respetando las restricciones indicadas. Por ejemplo, si $p = (3, 2, 5, 6)$ el resultado es 6 y si $p = (3, 6, 10)$ el resultado es 7. Notar que en una solución óptima, Lu debe terminar sin asteroides.

# Recordando casos base y pasos recursivos

- ¿Cuáles son nuestros casos base?
  - Si tenemos más asteroides que días, no vamos a poder vender todos, por lo que es una solución inválida.
  - Si vendimos un asteroide cuándo no teníamos ninguno rompemos la restricción, por lo que es una solución inválida.
  - Si ya no quedan días, nuestra m.g.n es 0.
- ¿Cuál es nuestro paso recursivo en el día $d$ con $a$ asteroides? El máximo entre:
  - la m.g.n. de finalizar el día $d-1$ con $a-1$ asteroides y comprar uno en el día $d$,
  - la m.g.n. de finalizar el día $d-1$ con $a+1$ asteroides y vender uno en el día $d$,
  - la m.g.n. de finalizar el día $d-1$ con $a$ asteroides y no operar el día $d$.

# Función recursiva simplificada

$$mgn(a,d) = \begin{cases} -\infty & \text{si } a < 0 \text{ o } a > d \\ 0 & \text{si } d = 0 \\ \max(mgn(a-1,d-1) - p[d], & \text{si no} \\ \quad mgn(a+1,d-1) + p[d], & \\ \quad mgn(a,d-1)) & \end{cases}$$

# Solución PD - Top Down

## Pseudocódigo con dinámica

- Sea $M$ una matriz de tamaño $(|p|+1) \times (|p|+1)$ inicializada en $-\infty$.

**Función** mgn(asteroid, day):

- Si $asteroid < 0$ o $asteroid > day$ devolver $-\infty$
- Si $day = 0$ devolver 0
- Si $M[day][asteroid]$ está definido devolver $M[day][asteroid]$
- Sea $p \leftarrow prices[day]$
- $ans \leftarrow$ máx$(mgn(asteroid, day-1,\ mgn(asteroid-1, day-1) - p\, mgn(asteroid+1, day-1) + p)$
- $M[day][asteroid] \leftarrow ans$
- devolver $ans$

# Complejidades

- Complejidad espacial: utilizamos una matriz de tamaño $O(|p|^2)$ la cuál tiene datos de tamaño $O(1)$ en cada posición, por lo que la complejidad espacial es $O(|p|^2)$
- Complejidad temporal:
  - Las operaciones dentro de cada llamado cuestan $O(1)$, pues son todas operaciones elementales
  - La cantidad de estados esta acotada superiormente por $O(|p|^2)$, así que la complejidad temporal es $O(|p|^2)$.

# Complejidades

- Complejidad espacial: utilizamos una matriz de tamaño $O(|p|^2)$ la cuál tiene datos de tamaño $O(1)$ en cada posición, por lo que la complejidad espacial es $O(|p|^2)$
- Complejidad temporal:
  - Las operaciones dentro de cada llamado cuestan $O(1)$, pues son todas operaciones elementales
  - La cantidad de estados esta acotada superiormente por $O(|p|^2)$, así que la complejidad temporal es $O(|p|^2)$.

## Futura optimización

¿Se puede tener una complejidad espacial auxiliar menor estricta a $O(n^2)$? Lo veremos en la clase de Bottom Up

# Solución PD - Bottom Up

## Pseudocódigo iterativo con dinámica

- Sea $M$ una matriz de tamaño $(n+1) \times (n+1)$ inicializada en $-\infty$.

**Algoritmo**:

- $M[0][0] \leftarrow 0$
- Para $d \leftarrow 1 \dots n$:
  - Para $a \leftarrow 0 \dots n$:
    - $ans \leftarrow M[d-1][a]$
    - Si $a > 0$: $ans \leftarrow \max(ans, M[d-1][a-1] - prices[d-1])$
    - Si $a < d$: $ans \leftarrow \max(ans, M[d-1][a+1] + prices[d-1])$
    - $M[d][a] \leftarrow ans$
- **Devolver** $M[n][0]$

# Implementación en Python

```python
def max_gain(prices):
    n = len(prices)
    M = [[-10**18] * (n + 1) for _ in range(n + 1)]
    M[0][0] = 0

    for d in range(1, n + 1):
        for a in range(0, d + 1):
            ans = M[d - 1][a]
            if a > 0:
                ans = max(ans, M[d - 1][a - 1] - prices[d - 1])
            if a < d:
                ans = max(ans, M[d - 1][a + 1] + prices[d - 1])
            M[d][a] = ans

    return M[n][0]
```

# Optimizando el espacio

- ¿Cuál es nuestra complejidad espacial?

# Optimizando el espacio

- ¿Cuál es nuestra complejidad espacial? Sigue siendo $O(n^2)$, no mejoramos la complejidad previa.
- ¿Qué podemos hacer para reducir la memoria?

# Optimizando el espacio

- ¿Cuál es nuestra complejidad espacial? Sigue siendo $O(n^2)$, no mejoramos la complejidad previa.
- ¿Qué podemos hacer para reducir la memoria? La clave es darnos cuenta que no necesitamos tener los resultados de los últimos $n$ días, con tener los del día anterior ya es suficiente.
- De esta forma podemos modificar nuestra estructura de memoización $M$ para guardar solo los últimos dos días.
- Veamos cómo nos queda nuestro algoritmo con complejidad espacial $O(n)$, la complejidad temporal sigue siendo la misma.

# Solución PD - Optimización de memoria

## Pseudocódigo iterativo con memoria reducida

- Sea $M$ una matriz de $2 \times (n+1)$ inicializada en $-\infty$.

**Algoritmo**:

- $M[0][0] \leftarrow 0$
- Para $d \leftarrow 1 \dots n$:
  - Sea $prev \leftarrow (d-1) \bmod 2$, $curr \leftarrow d \bmod 2$
  - Para $a \leftarrow 0 \dots n$:
    - $ans \leftarrow M[prev][a]$
    - Si $a > 0$: $ans \leftarrow \max(ans, M[prev][a-1] - prices[d-1])$
    - Si $a < d$: $ans \leftarrow \max(ans, M[prev][a+1] + prices[d-1])$
    - $M[curr][a] \leftarrow ans$
- **Devolver** $M[n \bmod 2][0]$

# Demostración - Consigna

## Enunciado

Formalmente, el problema consiste en determinar el máximo $g = \sum_{i=1}^{n} x_i p_i$ para un vector $x = (x_1, \ldots, x_n)$ tal que: $x_i \in \{-1, 0, 1\}$ para todo $1 \leq i \leq n$ y $\sum_{i=1}^{j} x_i \leq 0$ para todo $1 \leq j \leq n$. Demostrar que la formulación recursiva es correcta.

# Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día $n$.

# Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día $n$.
- Sea $o$ una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día $n$.

# Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día $n$.
- Sea $o$ una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día $n$.
  - Si $k = 0$, entonces la solución óptima tiene 0 asteroides.

# Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día $n$.
- Sea $o$ una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día $n$.
  - Si $k = 0$, entonces la solución óptima tiene 0 asteroides.
  - Si $k < 0$, no se cumple la restricción de vender lo que no tenes, por lo que no es una solución válida, así que $k$ no puede ser menor a 0.

# Solución óptima - Asteroides finales

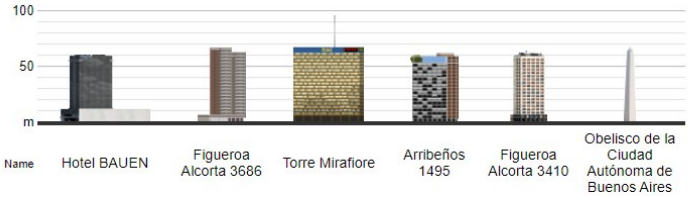
- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día $n$.
- Sea $o$ una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día $n$.
  - Si $k = 0$, entonces la solución óptima tiene 0 asteroides.
  - Si $k < 0$, no se cumple la restricción de vender lo que no tenes, por lo que no es una solución válida, así que $k$ no puede ser menor a 0.
  - Si $k > 0$, eso significa que nos quedó un asteroide por vender. Sea $i$ el último día que Lu compró un asteroide, podemos construir una solución $o'$ en la cuál no compró ese día. Luego la ganancia $g$ de $o'$ es $g(o') = g(o) + p_i$, ya que que ese día no compró el asteroide. Por lo tanto $g(o') > g(o)$, ABS!, pues $o$ es óptima. Por lo que $k$ no puede ser mayor a 0.

# Corrección de la recurrencia por inducción (I)

- **Proposición:** Para todo $d \in \{0, \ldots, n\}$ y todo $a$, la función $mgn(a, d)$ devuelve la máxima ganancia neta alcanzable al final del día $d$ poseyendo exactamente $a$ asteroides. Vamos a realizar una inducción en $d$, la cantidad de días.
- **Caso base ($d = 0$):**
  - El único estado posible es $a = 0$ (por lo probado anteriormente), con ganancia 0.
  - Si $a \neq 0$, el estado es imposible y se define como $-\infty$.
  - Esto coincide con la definición del problema: al inicio no se poseen asteroides ni se han hecho transacciones.
- **Hipótesis inductiva:** Supongamos que la proposición es válida para $d - 1$.

# Corrección de la recurrencia por inducción (II)

- **Paso inductivo ($d$):** Consideremos el día $d$ y un número de asteroides $a$. Toda secuencia factible que lleva al estado $(a, d)$ proviene de un estado factible en el día $d-1$ mediante exactamente una de estas acciones:
  - **No hacer nada:** estado previo $(a, d-1)$, ganancia $mgn(a, d-1)$.
  - **Comprar un asteroide:** estado previo $(a-1, d-1)$, ganancia $mgn(a-1, d-1) - p_d$.
  - **Vender un asteroide:** estado previo $(a+1, d-1)$, ganancia $mgn(a+1, d-1) + p_d$.
- Por hipótesis inductiva, cada término anterior ya representa la ganancia máxima posible en el estado previo correspondiente.
- Si $a < 0$ o $a > d$ caemos en un caso base y no hay que probar nada, pues no sería una solución válida u óptima (ya que no termina con 0 asteroides).
- Luego, tomar el máximo de los tres casos equivale a seleccionar la mejor acción en el día $d$.
- **Conclusión:** $mgn(a, d)$ computa correctamente la máxima ganancia neta alcanzable al final del día $d$ con $a$ asteroides.




## Mi Buenos Aires Crecido

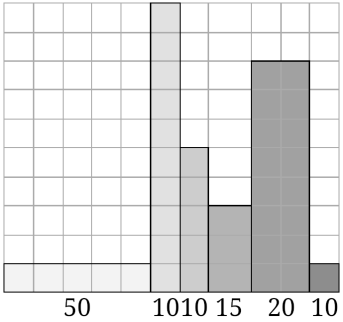
Sasha vive en San Nicolás hace mucho tiempo, y tiene una eterna discusión con su vecina Tasha. Sasha dice que si se mira el horizonte de izquierda a derecha terminando en el obelisco, los edificios están ordenados en un perfil principalmente ascendente para que la altura del obelisco sea más impresionante. Tasha le dice que con las nuevas torres que se construyeron en la zona eso ya no es verdad, y que en realidad ahora menos de la mitad del ancho del horizonte está en orden ascendente.

## Entrada

Recibimos una lista de edificios ordenados de izquierda a derecha con dos datos para cada uno, por un lado, el ancho de cada edificio y por el otro el alto.

Buscamos la longitud de la máxima subsecuencia ascendente, es decir, la subsecuencia creciente de edificios que ocupe el mayor espacio horizontal.

# Ejemplo




La máxima subsecuencia ascendente tiene 85 de largo, usando 3 edificios. Un edificio por su cuenta es también una subsecuencia ascendente.

# Función recursiva

Tenemos $N$ edificios y tenemos las funciones $alto(i)$ y $largo(i)$ que nos dan el alto y largo del edificio $i$.

Queremos tener una función $f$ que compute el ancho de la máxima subsecuencia ascendente.

Pensemos la función recursiva.
¿Cuál es nuestro caso base?
¿Cómo es el paso recursivo?

# Una posible respuesta

Sea $alto(-1) = 0$.

$$f(i, ult) = \begin{cases} 0 & \text{si } i \geq N \\ f(i+1, ult) & \text{si } alto(ult) \geq alto(i) \\ \max\{f(i+1, ult), f(i+1, i) + largo(i)\} & \text{c.c.} \end{cases}$$

La subsecuencia creciente más larga es $f(0, -1)$.

# Una posible respuesta

Sea $alto(-1) = 0$.

$$f(i, ult) = \begin{cases} 0 & \text{si } i \geq N \\ f(i+1, ult) & \text{si } alto(ult) \geq alto(i) \\ \max\{f(i+1, ult), f(i+1, i) + largo(i)\} & \text{c.c.} \end{cases}$$

La subsecuencia creciente más larga es $f(0, -1)$.

## ¿Cuántos llamados recursivos hace esta función?

# Una posible respuesta

Sea $alto(-1) = 0$.

$$f(i, ult) = \begin{cases} 0 & \text{si } i \geq N \\ f(i+1, ult) & \text{si } alto(ult) \geq alto(i) \\ \max\{f(i+1, ult), f(i+1, i) + largo(i)\} & \text{c.c.} \end{cases}$$

La subsecuencia creciente más larga es $f(0, -1)$.

## ¿Cuántos llamados recursivos hace esta función?

En el peor de los casos, los edificios están todos ordenados en forma creciente y el árbol de recursión explora todos los posibles subconjuntos de edificios. Teniendo $\Omega(2^N)$ hojas. (Una bocha).

Pero... ¿Cuántos resultados distintos nos puede dar?

# Solución Dinámica

Ahora, cada vez que necesitemos algún $f(i, ult)$ nos podemos primero fijar si ya lo tenemos calculado en $M$ y posiblemente usarlo en $O(1)$

## Pseudocódigo sin dinámica

F(i, ult):

- Si $i \geq N$ devolver 0
- Si $alto(ult) \geq alto(i)$ devolver $F(i+1, ult)$
- devolver $\max\{F(i+1, ult),\ F(i+1, i) + largo(i)\}$

## Pseudocódigo con dinámica

- Sea $M$ una matriz en $\mathbb{N}^{N \times N}$ inicializada con valores indefinidos.

F(i, ult):

- Si $M[i][ult]$ está definido devolver $M[i][ult]$
- Si $i \geq N$ devolver 0
- Si $alto(ult) \geq alto(i)$ devolver $F(i+1, ult)$
- $M[i][ult] \leftarrow \max\{F(i+1, ult),\ F(i+1, i) + largo(i)\}$
- devolver $M[i][ult]$

# Nueva complejidad

Todos los llamados a F que ya estén guardados en la matriz los consideramos iguales a simplemente leer memoria y, por lo tanto, despreciables.

La complejidad de un algoritmo dinámico es la cantidad de estados multiplicado por lo que cuesta resolver internamente cada estado. En nuestro caso, eso es $O(N^2) \cdot O(1) = O(N^2)$ (mucho mejor).

¿Perdimos algo a cambio de esta mejora en complejidad temporal?

¿Perdimos algo a cambio de esta mejora en complejidad temporal? Ahora tenemos una **complejidad espacial** de $O(N^2)$.

Hay una versión con la misma complejidad temporal pero con complejidad espacial $O(N)$, ¿cómo podemos adaptar nuestra nueva solución?

¿Perdimos algo a cambio de esta mejora en complejidad temporal? Ahora tenemos una **complejidad espacial** de $O(N^2)$.

Hay una versión con la misma complejidad temporal pero con complejidad espacial $O(N)$, ¿cómo podemos adaptar nuestra nueva solución?

Si queremos que nuestra memoria sea $O(N)$ entonces nuestra cantidad de estados debe ser $O(N)$. ¿Cómo podemos modificar nuestra función recursiva? ¿Que necesitamos si o si contabilizar en nuestro estado?

# Función recursiva

$$LIS(pos) = \begin{cases} anchos[pos] & \text{si } \nexists j < pos, t.q\ alt[j] < alt[pos] \\ anchos[pos] + \max\limits_{j<pos,\ alturas[j]<alturas[pos]} LIS(j) & \text{en otro caso} \end{cases}$$

## Pseudocódigo con memoización

LIS(pos):

- Si $memo[pos] \neq -1$ devolver $memo[pos]$
- ancho_máximo_anterior $\leftarrow 0$
- Para cada $j \in [0, pos - 1]$:
  - Si $alturas[j] < alturas[pos]$ entonces
    - ancho_máximo_anterior $\leftarrow$ máx(ancho_máximo_anterior, LIS($j$))
- ancho_total_máximo $\leftarrow anchos[pos] +$ ancho_máximo_anterior
- $memo[pos] \leftarrow$ ancho_total_máximo
- devolver $memo[pos]$

# Implementación en C++

```
1  int LIS(int pos, const vector<int>& alturas, const vector<int>&
       anchos, vector<int>& memo) {
2      if (memo[pos]==-1) {
3          int ancho_maximo_anterior = 0;
4          for (int j=0;j<pos;j++) {
5              if (alturas[j]<alturas[pos]) {
6                  ancho_maximo_anterior = max(ancho_maximo_anterior,
                       LIS(j,alturas,anchos,memo));
7              }
8          }
9          int ancho_total_maximo = anchos[pos]+ancho_maximo_anterior;
10         memo[pos] = ancho_total_maximo;
11     }
12     return memo[pos];
13 }
```

# Complejidad

- Complejidad espacial: nuestra complejidad espacial se reduce a $O(N)$.

# Complejidad

- Complejidad espacial: nuestra complejidad espacial se reduce a $O(N)$.
- Complejidad temporal: nuestra complejidad temporal sigue igual, ya que el costo de calcular cada estado es $O(pos)$, luego la complejidad total será $O(N^2)$.
- Logramos optimizar el espacio utilizado, pero no el tiempo. ¿Se les ocurre cómo podrían resolver este problema en complejidad temporal $O(N \log N)$?

# Ejercicio 3

## Garland

Lean adora decorar su árbol de Rosh Hashaná, y recibió como regalo una guirnalda con $n$ bombillas dispuestas en una fila. Cada bombilla lleva un número distinto entre 1 y $n$, en orden arbitrario. Mientras Lean resolvía problemas, su gato Sasha retiró algunas bombillas, y ahora él debe volver a colocarlas. Sea $a$ el arreglo de bombillas. Lean define la complejidad de la guirnalda como la cantidad de pares de bombillas adyacentes cuyos números tienen diferente paridad (es decir, uno es par y el otro impar). Por ejemplo:

- Para la secuencia 1 4 2 3 5, la complejidad es 2.
- Para la secuencia 1 3 5 7 6 4 2, la complejidad es 1.

Lean quiere volver a colocar todas las bombillas de forma que la complejidad sea mínima. ¿Podés ayudarlo a encontrar ese valor mínimo?

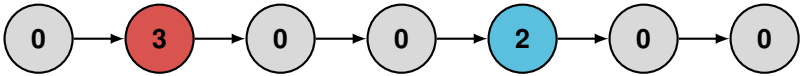
# Especificaciones

- **Formato de entrada:**
  - Una línea que contiene un entero $n$ ($1 \leq n \leq 100$) — el número total de bombillas.
  - Una línea con $n$ enteros $p_1, p_2, \ldots, p_n$ ($0 \leq p_i \leq n$), donde $p_i$ es el número de la bombilla en la posición $i$, o 0 si fue retirada.
- **Formato de salida:**
  - Un único entero: la complejidad mínima posible tras volver a colocar todas las bombillas.

# Garland — Relleno y costo

Secuencia objetivo con mínimos cambios de paridad:

$[1, 3, 5, 7, 2, 4, 6] \Rightarrow \text{costo} = 1$




**Dados:** $n = 7,\ p_2 = 3$ (impar), $p_5 = 2$ (par)
Disponibles: $\{\text{impares } 1, 5, 7\},\ \{\text{pares } 4, 6\}$

- El **costo** es la cantidad de aristas entre posiciones adyacentes con paridades distintas.
- La secuencia final $[1, 3, 5, 7, 2, 4, 6]$ tiene un único cambio (entre 7 y 2).

# Garland — Relleno y costo

Secuencia objetivo con mínimos cambios de paridad:

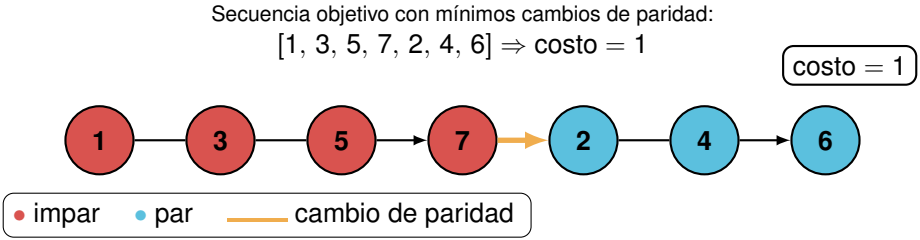
$[1, 3, 5, 7, 2, 4, 6] \Rightarrow \text{costo} = 1$


**Idea:** agrupar por paridad para minimizar cambios.
Antes de la pos. 5 (par) ubicamos impares: $1, 3, 5, 7$.

cambios hasta ahora $= 1$

- El **costo** es la cantidad de aristas entre posiciones adyacentes con paridades distintas.
- La secuencia final $[1, 3, 5, 7, 2, 4, 6]$ tiene un único cambio (entre 7 y 2).

# Garland — Relleno y costo




- El **costo** es la cantidad de aristas entre posiciones adyacentes con paridades distintas.
- La secuencia final $[1, 3, 5, 7, 2, 4, 6]$ tiene un único cambio (entre 7 y 2).

# Intuición de la solución - Estados

- ¿Que vamos a querer calcular en cada estado?

# Intuición de la solución - Estados

- ¿Que vamos a querer calcular en cada estado? El **costo**
- ¿Cómo hacemos para calcular el costo? ¿Que valores necesitamos tener en cuenta?

# Intuición de la solución - Estados

- ¿Que vamos a querer calcular en cada estado? El **costo**
- ¿Cómo hacemos para calcular el costo? ¿Que valores necesitamos tener en cuenta?

# Intuición de la solución - Estados

- ¿Que vamos a querer calcular en cada estado? El **costo**
- ¿Cómo hacemos para calcular el costo? ¿Que valores necesitamos tener en cuenta?

## Qué necesitamos recordar en cada paso

Para decidir qué hacer en la posición *i* debemos saber:

- Cuántos pares ya usamos hasta ahora.
- La paridad del último elemento colocado (0 par, 1 impar).

# Intuición de la solución - Opciones

¿Que elemento $a_i$ nos podemos encontrar en cada posición? ¿Cuáles son

- Si $a_i \neq 0$: no podemos elegir, seguimos con esa paridad.
- Si $a_i = 0$: tenemos dos opciones:
  - Poner un par (si todavía quedan disponibles).
  - Poner un impar (si todavía quedan disponibles).
- Cada elección puede generar un cambio de paridad, que suma costo.
- Por eso definimos una función $gar(i, e, p)$ que representa el costo mínimo desde la posición $i$, con $e$ pares usados y último elemento de paridad $p$.

# Función recursiva - Definiciones

Sea $gar(i, e, p) =$

# Función recursiva - Definiciones

Sea $gar(i, e, p) =$

- el costo mínimo desde la posición **i**,
- con **e** pares usados entre ceros hasta **i**,
- siendo **p** $\in \{0, 1\}$ la paridad del último elemento (0 par, 1 impar).

# Función recursiva - Definiciones

Sea $gar(i, e, p) =$

- el costo mínimo desde la posición **i**,
- con **e** pares usados entre ceros hasta **i**,
- siendo **p** $\in \{0, 1\}$ la paridad del último elemento (0 par, 1 impar).

$$\text{par}(x) = x \text{ mód } 2, \qquad \text{odds\_used}(i, e) = \text{prefZero}[i] - e,$$

¿Cuáles son nuestras dos opciones? ¿Cuándo podemos tomar cada una?

# Función recursiva - Definiciones

Sea $gar(i, e, p) =$

- el costo mínimo desde la posición **i**,
- con **e** pares usados entre ceros hasta **i**,
- siendo **p** $\in \{0, 1\}$ la paridad del último elemento (0 par, 1 impar).

$$\text{par}(x) = x \text{ mód } 2, \qquad \text{odds\_used}(i, e) = \text{prefZero}[i] - e,$$

¿Cuáles son nuestras dos opciones? ¿Cuándo podemos tomar cada una?

$$S(i, e) = \Big\{ \underbrace{(e + 1, 0)}_{\text{poner par}} \text{ si } e < \text{avail\_even} \Big\} \cup \Big\{ \underbrace{(e, 1)}_{\text{poner impar}} \text{ si odds\_used}(i, e) < \text{avail\_odd} \Big\}.$$

# Función recursiva

$$\text{costoCambio}(i, par, par') = \begin{cases} 1 & \text{si } i > 0 \text{ y } par \neq par', \\ 0 & \text{sino} \end{cases}$$

# Función recursiva

$$\text{costoCambio}(i, par, par') = \begin{cases} 1 & \text{si } i > 0 \text{ y } par \neq par', \\ 0 & \text{sino} \end{cases}$$

$$gar(i, e, p) = \begin{cases} 0 & \text{si } i = n, \\ \text{costoCambio}\big(i, p, \text{par}(a_i)\big) + f\big(i+1,\ e,\ \text{par}(a_i)\big) & \text{si } a_i \neq 0, \\ \min\limits_{(e', p') \in S(i,e)} \Big(\text{costoCambio}(i, p, p') + gar(i+1,\ e',\ p')\Big) & \text{si } a_i = 0. \end{cases}$$

Convención: para el primer elemento no se penaliza el cambio, por eso el costo del cambio pide que $i > 0$.

# Implementación Top Down (I) - C++

```
1   int gar(int i, int e_used, int last_par) {
2       if (i == n) return 0;
3
4       int &res = memo[i][e_used][last_par];
5       if (res != -1) return res;
6       res = INF;
7
8       bool first = (i == 0);
9
10      auto relax = [&](int next_e_used, int cur_par) {
11          int add = (first ? 0 : (cur_par != last_par));
12          res = min(res, add + gar(i + 1, next_e_used, cur_par));
13      };
14  }
```

# Implementación Top Down (II) - C++

```
1         if (a[i] != 0) {
2             int par = (a[i] & 1);
3             relax(e_used, par);
4         } else {
5             // a[i] == 0 -> podemos elegir colocar un par o un impar, si
                  quedan disponibles
6             int zeros_used = prefZero[i];
7             int odds_used  = zeros_used - e_used;
8
9             // colocar par
10            if (e_used + 1 <= avail_even) {
11                relax(e_used + 1, 0);
12            }
13            // colocar impar
14            if (odds_used + 1 <= avail_odd) {
15                relax(e_used, 1);
```

# Implementación Bottom Up (I) - C++

```
1     // Transicion: de i = n-1 hacia 0
2     for (int i = n - 1; i >= 0; --i) {
3         for (int e = 0; e <= avail_even; ++e) {
4             for (int last_par = 0; last_par <= 1; ++last_par) {
5                 int best = INF;
6                 bool first = (i == 0); // no penaliza cambio en el
                      primero
7
8                 auto updateBest = [&](int next_par, int next_e) {
9                     int add = (first ? 0 : (next_par != last_par));
10                    best = min(best, add + dp[i + 1][next_e][next_par]);
11                };
```

# Implementación Bottom Up (II) - C++

```
                if (a[i] != 0) {
                    int par = (a[i] & 1);
                    updateBest(par, e);
                } else {
                    // Elegimos par o impar si quedan disponibles
                    int zeros_used = prefZero[i];
                    int odds_used  = zeros_used - e;

                    // Colocar par
                    if (e + 1 <= avail_even)
                        updateBest(0, e + 1);
                    // Colocar impar
                    if (odds_used + 1 <= avail_odd)
                        updateBest(1, e);
                }
                dp[i][e][last_par] = best;
```

# Caesar's Legions

## Caesar's Legions

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 sus tropas incumpla que:
  - No puede haber más de $MP$ patos consecutivos.
  - No puede haber más de $MD$ dodos consecutivos.
- Los patos son indistinguibles entre ellos, al igual que los dodos.

Le interesa contar la cantidad de formas posibles de formar esta línea.

## Consigna de hoy

**Pensar como optimizar la solución para que sea $O(P * D)$ en tiempo y espacio.**

# Solución actual

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = \sum_{i=1}^{\min(n_P, MP)} f(n_P - i, n_D, PATO)$$

$$ponerDODO(n_P, n_D) = \sum_{i=1}^{\min(n_D, MD)} f(n_P, n_D - i, DODO)$$

# Solución actual

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = \sum_{i=1}^{\text{mín}(n_P, MP)} f(n_P - i, n_D, PATO)$$

$$ponerDODO(n_P, n_D) = \sum_{i=1}^{\text{mín}(n_D, MD)} f(n_P, n_D - i, DODO)$$

- Cantidad de estados: $P * D * 2 \in \Theta(P * D)$

# Solución actual

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = \sum_{i=1}^{\text{mín}(n_P, MP)} f(n_P - i, n_D, PATO)$$

$$ponerDODO(n_P, n_D) = \sum_{i=1}^{\text{mín}(n_D, MD)} f(n_P, n_D - i, DODO)$$

- Cantidad de estados: $P * D * 2 \in \Theta(P * D)$
- Cantidad de transiciones por estado $O(MP + MD) \in O(P + D)$

# Solución actual

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = \sum_{i=1}^{\min(n_P, MP)} f(n_P - i, n_D, PATO)$$

$$ponerDODO(n_P, n_D) = \sum_{i=1}^{\min(n_D, MD)} f(n_P, n_D - i, DODO)$$

- Cantidad de estados: $P * D * 2 \in \Theta(P * D)$
- Cantidad de transiciones por estado $O(MP + MD) \in O(P + D)$
- Asumiendo $P, D \in O(N)$, la complejidad total es $O(N^3)$

# Optimizando la solución

**Objetivo:** Reducir la complejidad temporal de $O(P * D * (P + D))$ a $O(P * D)$

# Optimizando la solución

**Objetivo:** Reducir la complejidad temporal de $O(P * D * (P + D))$ a $O(P * D)$
**Observación:** *ponerPATO* y *ponerDODO* son sumas de rangos.

# Optimizando la solución

**Objetivo:** Reducir la complejidad temporal de $O(P * D * (P + D))$ a $O(P * D)$
**Observación:** *ponerPATO* y *ponerDODO* son sumas de rangos.
**Idea:** Podemos utilizar una **Tabla Aditiva** para calcular estas sumas en $O(1)$, y la podemos construir a medida que calculamos los estados.

# Optimizando la solución

**Objetivo:** Reducir la complejidad temporal de $O(P * D * (P + D))$ a $O(P * D)$
**Observación:** *ponerPATO* y *ponerDODO* son sumas de rangos.
**Idea:** Podemos utilizar una **Tabla Aditiva** para calcular estas sumas en $O(1)$, y la podemos construir a medida que calculamos los estados.
**¿Cómo?**:

# Optimizando la solución

**Objetivo:** Reducir la complejidad temporal de $O(P * D * (P + D))$ a $O(P * D)$
**Observación:** *ponerPATO* y *ponerDODO* son sumas de rangos.
**Idea:** Podemos utilizar una **Tabla Aditiva** para calcular estas sumas en $O(1)$, y la podemos construir a medida que calculamos los estados.
**¿Cómo?**:

- Definimos $T_P$ y $T_D$:
  - $T_P[i][n_D] = \sum_{j=0}^{i-1} f(j, n_D, PATO) = T_P[i-1][n_D] + f(i-1, n_D, PATO)$
  - $T_D[i][n_P] = \sum_{j=0}^{i-1} f(n_P, j, DODO) = T_D[i-1][n_P] + f(n_P, i-1, DODO)$

# Optimizando la solución

**Objetivo:** Reducir la complejidad temporal de $O(P * D * (P + D))$ a $O(P * D)$
**Observación:** *ponerPATO* y *ponerDODO* son sumas de rangos.
**Idea:** Podemos utilizar una **Tabla Aditiva** para calcular estas sumas en $O(1)$, y la podemos construir a medida que calculamos los estados.
**¿Cómo?**:

- Definimos $T_P$ y $T_D$:
  - $T_P[i][n_D] = \sum_{j=0}^{i-1} f(j, n_D, PATO) = T_P[i-1][n_D] + f(i-1, n_D, PATO)$
  - $T_D[i][n_P] = \sum_{j=0}^{i-1} f(n_P, j, DODO) = T_D[i-1][n_P] + f(n_P, i-1, DODO)$
- Al calcular un valor $f(n_P, n_D, ultTropa)$ actualizamos $T_P$ o $T_D$ según corresponda.
- $ultTropa = PATO \rightarrow T_P[n_P + 1][n_D] = f(n_P, n_D, PATO) + T_P[n_P][n_D]$
- $ultTropa = DODO \rightarrow T_D[n_P][n_D + 1] = f(n_P, n_D, DODO) + T_D[n_P][n_D]$
- Inicializamos $T_P[0][n_D] = 0$ y $T_D[n_P][0] = 0$ para todo $n_P, n_D$

# Actualizamos $f$

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = T_P[n_P][n_D] - T_P[\text{máx}(0, n_P - MP)][n_D]$$

$$ponerDODO(n_P, n_D) = T_D[n_P][n_D] - T_D[n_P][\text{máx}(0, n_D - MD)]$$

# Actualizamos $f$

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = T_P[n_P][n_D] - T_P[\text{máx}(0, n_P - MP)][n_D]$$

$$ponerDODO(n_P, n_D) = T_D[n_P][n_D] - T_D[n_P][\text{máx}(0, n_D - MD)]$$

- Cantidad de estados: $P * D * 2 \in \Theta(P * D)$
- Cantidad de transiciones por estado $O(1)$
- Ahora la complejidad es $O(P * D)$

# Actualizamos $f$

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = T_P[n_P][n_D] - T_P[\text{máx}(0, n_P - MP)][n_D]$$

$$ponerDODO(n_P, n_D) = T_D[n_P][n_D] - T_D[n_P][\text{máx}(0, n_D - MD)]$$

- Cantidad de estados: $P * D * 2 \in \Theta(P * D)$
- Cantidad de transiciones por estado $O(1)$
- Ahora la complejidad es $O(P * D)$
- **Problema:** Necesitamos garantizar que los estados se calculen en el orden correcto para poder usar las tablas aditivas.

# Actualizamos $f$

$$f(n_P, n_D, ultTropa) = \begin{cases} 1 & \text{si } (n_P + n_D) = 0 \\ ponerPATO(n_P, n_D) & \text{si } ultTropa = DODO \\ ponerDODO(n_P, n_D) & \text{si } ultTropa = PATO \end{cases}$$

$$ponerPATO(n_P, n_D) = T_P[n_P][n_D] - T_P[\text{máx}(0, n_P - MP)][n_D]$$

$$ponerDODO(n_P, n_D) = T_D[n_P][n_D] - T_D[n_P][\text{máx}(0, n_D - MD)]$$

- Cantidad de estados: $P * D * 2 \in \Theta(P * D)$
- Cantidad de transiciones por estado $O(1)$
- Ahora la complejidad es $O(P * D)$
- **Problema:** Necesitamos garantizar que los estados se calculen en el orden correcto para poder usar las tablas aditivas.
- **Solución:** Podemos usar Bottom-Up para garantizar el orden correcto.

# Implementación Bottom-Up

```
np, nd, kp, kd = map(int, input().split())
mod, PATO, DODO = 10**8, 0, 1

f: list[list[list[int]]] = [[[0,0] for _ in range(nd+2)] for _ in range(np+2)]
tp: list[list[int]] = [[0]*(nd+2) for _ in range(np+2)]
td: list[list[int]] = [[0]*(nd+2) for _ in range(np+2)]

for p in range(0,np+1):
    for d in range(0, nd+1):
        for ultTropa in range(2):
            if (p+d) == 0: f[p][d][ultTropa] = 1
            elif ultTropa == DODO:
                f[p][d][DODO] = (tp[p][d] - tp[max(0,p-kp)][d] + mod) % mod
            else:
                f[p][d][PATO] = (td[p][d] - td[p][max(0,d-kd)] + mod) % mod
            if ultTropa == PATO: tp[p+1][d] = (tp[p][d] + f[p][d][PATO]) % mod
            else: td[p][d+1] = (td[p][d] + f[p][d][DODO]) % mod
print((f[np][nd][PATO] + f[np][nd][DODO]) % mod)
```

# Implementación Bottom-Up

```
np, nd, kp, kd = map(int, input().split())
mod, PATO, DODO = 10**8, 0, 1

f: list[list[list[int]]] = [[[0,0] for _ in range(nd+2)] for _ in range(np+2)]
tp: list[list[int]] = [[0]*(nd+2) for _ in range(np+2)]
td: list[list[int]] = [[0]*(nd+2) for _ in range(np+2)]

for p in range(0,np+1):
    for d in range(0, nd+1):
        for ultTropa in range(2):
            if (p+d) == 0: f[p][d][ultTropa] = 1
            elif ultTropa == DODO:
                f[p][d][DODO] = (tp[p][d] - tp[max(0,p-kp)][d] + mod) % mod
            else:
                f[p][d][PATO] = (td[p][d] - td[p][max(0,d-kd)] + mod) % mod
            if ultTropa == PATO: tp[p+1][d] = (tp[p][d] + f[p][d][PATO]) % mod
            else: td[p][d+1] = (td[p][d] + f[p][d][DODO]) % mod
print((f[np][nd][PATO] + f[np][nd][DODO]) % mod)
```

**AC: 156ms**

# Lagunas - I

Marcos tiene un terreno en el que puede haber varias lagunas, y quiere aprovechar esos cuerpos de agua para colocar algunos barquitos. El terreno puede representarse como una grilla de dimensiones $1 \times N$ donde cada una de sus $N$ casillas está formada enteramente por tierra o por agua. Cada barquito que se consigue en el mercado de barquitos tiene un largo $k$ para algún entero positivo $k$, y puede colocarse en el terreno ocupando $k$ casillas de agua que sean consecutivas. No es posible colocar más de un barquito en la misma casilla de agua.

Marcos puede conseguir la cantidad que quiera de barquitos, de los largos que quiera. Sin embargo, para que el paisaje se vea estético y diverso, Marcos decidió que todos los barquitos que coloque en el terreno deben tener largos diferentes y estar ordenados por largo de forma (estrictamente) creciente de izquierda a derecha.

# Laguna - II

Por cada barquito que Marcos consiga colocar, obtiene una ganancia $G$. Sin embargo, Marcos no está conforme con la distribución de su terreno, por lo que está dispuesto a excavar algunas casillas de tierra, para que pasen a ser de agua y así poder colocar más barquitos. Excavar la i-ésima casilla del terreno tiene un costo variable $T_i$, ya que no todas las casillas tienen la misma cantidad de tierra. Si Marcos elige convenientemente qué casillas excavar y dónde colocar los barquitos, ¿cuál es la máxima ganancia neta que puede conseguir?

# Consigna

1. Resolver el problema con un algoritmo de PD en $O(N^3)$.
2. Optimizar la solución anterior para que funcione en $O(N^2 * \sqrt{N}) = (N^{\frac{5}{2}})$.
3. Optimizar la solución anterior para que funcione en $O(N * \sqrt{N}) = (N^{\frac{3}{2}})$.
4. Optimizar las soluciones anteriores para que utilicen $O(N)$ memoria.

# $O(N^3)$

- Tenemos $N$ casillas y $N$ posibles barquitos.

# $O(N^3)$

- Tenemos $N$ casillas y $N$ posibles barquitos.
- *Estado* : $(i, b)$: Máxima ganancia posible considerando las primeras $i$ casillas y los primeros $b$ barcos.

# $O(N^3)$

- Tenemos $N$ casillas y $N$ posibles barquitos.
- *Estado* : $(i, b)$: Máxima ganancia posible considerando las primeras $i$ casillas y los primeros $b$ barcos.
- *Transición* : Puedo usar el barco $b$ y considerar desde las celdas $i - b$, no usarlo y considerar desde la celda $i - 1$ o descartarlo y considerar desde la celda $i$.

# $O(N^3)$

- Tenemos $N$ casillas y $N$ posibles barquitos.
- *Estado* : $(i, b)$: Máxima ganancia posible considerando las primeras $i$ casillas y los primeros $b$ barcos.
- *Transición* : Puedo usar el barco $b$ y considerar desde las celdas $i - b$, no usarlo y considerar desde la celda $i - 1$ o descartarlo y considerar desde la celda $i$.
-

$$f(i,b) = \begin{cases} 0 & \text{si } i = 0 \text{ o } b = 0 \\ \max\left(f(i-1,b), f(i,b-1), \mathit{usar\_barco}(i,b)\right) & \text{si } i > 0 \\ -\infty & \text{si } i < 0 \text{ o } b < 0 \end{cases}$$

$$\mathit{usar\_barco}(i,b) = \begin{cases} f(i-b, b-1) + G - \left(\sum_{j=i-b+1}^{i} T_j\right) & \text{si } i - b \geq 0 \\ -\infty & \text{si } i - b < 0 \end{cases}$$

# $O(N^3)$

- Tenemos $N$ casillas y $N$ posibles barquitos.
- *Estado* : $(i, b)$: Máxima ganancia posible considerando las primeras $i$ casillas y los primeros $b$ barcos.
- *Transición* : Puedo usar el barco $b$ y considerar desde las celdas $i - b$, no usarlo y considerar desde la celda $i - 1$ o descartarlo y considerar desde la celda $i$.
- $$f(i,b) = \begin{cases} 0 & \text{si } i = 0 \text{ o } b = 0 \\ \max(f(i-1,b), f(i,b-1), usar\_barco(i,b)) & \text{si } i > 0 \\ -\infty & \text{si } i < 0 \text{ o } b < 0 \end{cases}$$
$$usar\_barco(i,b) = \begin{cases} f(i-b,b-1) + G - (\sum_{j=i-b+1}^{i} T_j) & \text{si } i - b \geq 0 \\ -\infty & \text{si } i - b < 0 \end{cases}$$
- La respuesta es $f(N, N)$.

# $O(N^3)$

- Tenemos $N$ casillas y $N$ posibles barquitos.
- *Estado* : $(i, b)$: Máxima ganancia posible considerando las primeras $i$ casillas y los primeros $b$ barcos.
- *Transición* : Puedo usar el barco $b$ y considerar desde las celdas $i - b$, no usarlo y considerar desde la celda $i - 1$ o descartarlo y considerar desde la celda $i$.
-

$$f(i,b) = \begin{cases} 0 & \text{si } i = 0 \text{ o } b = 0 \\ \max\left(f(i-1,b), f(i,b-1), \mathit{usar\_barco}(i,b)\right) & \text{si } i > 0 \\ -\infty & \text{si } i < 0 \text{ o } b < 0 \end{cases}$$

$$\mathit{usar\_barco}(i,b) = \begin{cases} f(i-b, b-1) + G - \left(\sum_{j=i-b+1}^{i} T_j\right) & \text{si } i - b \geq 0 \\ -\infty & \text{si } i - b < 0 \end{cases}$$

- La respuesta es $f(N, N)$.
- $N^2$ estados y una sumatoria $O(N)$ por estado $\implies O(N^3)$

# Optimización a $O(N^2 * \sqrt{N})$

- Notar que nunca me conviene saltar un barco. Es decir, si uso $b$ barcos, son los de longitudes $1, 2, \ldots, b$.

# Optimización a $O(N^2 * \sqrt{N})$

- Notar que nunca me conviene saltar un barco. Es decir, si uso $b$ barcos, son los de longitudes $1, 2, \ldots, b$.
- Sea $L$ la longitud total de los barcos usados, entonces $L = \sum_{i=1}^{b} i = \frac{b(b+1)}{2} \implies b = O(\sqrt{L})$.

# Optimización a $O(N^2 * \sqrt{N})$

- Notar que nunca me conviene saltar un barco. Es decir, si uso $b$ barcos, son los de longitudes $1, 2, \ldots, b$.
- Sea $L$ la longitud total de los barcos usados, entonces $L = \sum_{i=1}^{b} i = \frac{b(b+1)}{2} \implies b = O(\sqrt{L})$.
- Como $L \leq N$, puedo acotar $b$ por una constante $B$ tal que $\frac{B(B+1)}{2} \leq N$ y $\frac{(B+1)(B+2)}{2} > N$, y entonces $b \leq B = O(\sqrt{N})$.

# Optimización a $O(N^2 * \sqrt{N})$

- Notar que nunca me conviene saltar un barco. Es decir, si uso $b$ barcos, son los de longitudes $1, 2, \ldots, b$.
- Sea $L$ la longitud total de los barcos usados, entonces $L = \sum_{i=1}^{b} i = \frac{b(b+1)}{2} \implies b = O(\sqrt{L})$.
- Como $L \leq N$, puedo acotar $b$ por una constante $B$ tal que $\frac{B(B+1)}{2} \leq N$ y $\frac{(B+1)(B+2)}{2} > N$, y entonces $b \leq B = O(\sqrt{N})$.
- Entonces, la respuesta es $f(N, B)$.

# Optimización a $O(N^2 * \sqrt{N})$

- Notar que nunca me conviene saltar un barco. Es decir, si uso $b$ barcos, son los de longitudes $1, 2, \ldots, b$.
- Sea $L$ la longitud total de los barcos usados, entonces $L = \sum_{i=1}^{b} i = \frac{b(b+1)}{2} \implies b = O(\sqrt{L})$.
- Como $L \leq N$, puedo acotar $b$ por una constante $B$ tal que $\frac{B(B+1)}{2} \leq N$ y $\frac{(B+1)(B+2)}{2} > N$, y entonces $b \leq B = O(\sqrt{N})$.
- Entonces, la respuesta es $f(N, B)$.
- $N * B = O(N * \sqrt{N})$ estados y una sumatoria $O(N)$ por estado $\implies O(N^2 * \sqrt{N})$.

# Optimización a $O(N * \sqrt{N})$

- Notar que la sumatoria $\sum_{j=i-b+1}^{i} T_j$ es una suma de rango.

# Optimización a $O(N * \sqrt{N})$

- Notar que la sumatoria $\sum_{j=i-b+1}^{i} T_j$ es una suma de rango.
- Podemos usar una tabla aditiva $T$ para calcularla en $O(1)$, y construirla a medida que calculamos los estados.

# Optimización a $O(N * \sqrt{N})$

- Notar que la sumatoria $\sum_{j=i-b+1}^{i} T_j$ es una suma de rango.
- Podemos usar una tabla aditiva $T$ para calcularla en $O(1)$, y construirla a medida que calculamos los estados.
- Definimos $T[i] = \sum_{j=1}^{i} T_j = T[i-1] + T_i$, con $T[0] = 0$.

# Optimización a $O(N * \sqrt{N})$

- Notar que la sumatoria $\sum_{j=i-b+1}^{i} T_j$ es una suma de rango.
- Podemos usar una tabla aditiva $T$ para calcularla en $O(1)$, y construirla a medida que calculamos los estados.
- Definimos $T[i] = \sum_{j=1}^{i} T_j = T[i-1] + T_i$, con $T[0] = 0$.
- Entonces, $\sum_{j=i-b+1}^{i} T_j = T[i] - T[i-b]$.

# Optimización a $O(N * \sqrt{N})$

- Notar que la sumatoria $\sum_{j=i-b+1}^{i} T_j$ es una suma de rango.
- Podemos usar una tabla aditiva $T$ para calcularla en $O(1)$, y construirla a medida que calculamos los estados.
- Definimos $T[i] = \sum_{j=1}^{i} T_j = T[i-1] + T_i$, con $T[0] = 0$.
- Entonces, $\sum_{j=i-b+1}^{i} T_j = T[i] - T[i-b]$.
- La transición queda:

$$f(i,b) = \begin{cases} 0 & \text{si } i = 0 \text{ o } b = 0 \\ \max(f(i-1,b), f(i,b-1), usar\_barco(i,b)) & \text{si } i > 0 \\ -\infty & \text{si } i < 0 \text{ o } b < 0 \end{cases}$$

$$usar\_barco(i,b) = \begin{cases} f(i-b,b-1) + G - (T[i] - T[i-b]) & \text{si } i-b \geq 0 \\ -\infty & \text{si } i-b < 0 \end{cases}$$

# Optimización a $O(N * \sqrt{N})$

- Notar que la sumatoria $\sum_{j=i-b+1}^{i} T_j$ es una suma de rango.
- Podemos usar una tabla aditiva $T$ para calcularla en $O(1)$, y construirla a medida que calculamos los estados.
- Definimos $T[i] = \sum_{j=1}^{i} T_j = T[i-1] + T_i$, con $T[0] = 0$.
- Entonces, $\sum_{j=i-b+1}^{i} T_j = T[i] - T[i-b]$.
- La transición queda:

$$f(i,b) = \begin{cases} 0 & \text{si } i = 0 \text{ o } b = 0 \\ \max(f(i-1,b), f(i,b-1), usar\_barco(i,b)) & \text{si } i > 0 \\ -\infty & \text{si } i < 0 \text{ o } b < 0 \end{cases}$$

$$usar\_barco(i,b) = \begin{cases} f(i-b,b-1) + G - (T[i] - T[i-b]) & \text{si } i-b \geq 0 \\ -\infty & \text{si } i-b < 0 \end{cases}$$

- $N * B = O(N * \sqrt{N})$ estados y $O(1)$ por estado $\implies O(N * \sqrt{N})$.

# Optimización a $O(N)$ memoria

- Notar que para calcular los estados para un dado valor de $b = b_0$, solo se usan estados con valores de $b = b_0$ y $b = b_0 - 1$.

# Optimización a $O(N)$ memoria

- Notar que para calcular los estados para un dado valor de $b = b_0$, solo se usan estados con valores de $b = b_0$ y $b = b_0 - 1$.
- Entonces, si vamos calculando por *capas* de $b$, solo necesitamos guardar la capa $b = b_0$ y la capa $b = b_0 - 1$.

# Optimización a $O(N)$ memoria

- Notar que para calcular los estados para un dado valor de $b = b_0$, solo se usan estados con valores de $b = b_0$ y $b = b_0 - 1$.
- Entonces, si vamos calculando por *capas* de $b$, solo necesitamos guardar la capa $b = b_0$ y la capa $b = b_0 - 1$.
- Podemos utilizar PD bottom-up para controlar el orden en el que calculamos los estados.

# Optimización a $O(N)$ memoria

- Notar que para calcular los estados para un dado valor de $b = b_0$, solo se usan estados con valores de $b = b_0$ y $b = b_0 - 1$.
- Entonces, si vamos calculando por *capas* de $b$, solo necesitamos guardar la capa $b = b_0$ y la capa $b = b_0 - 1$.
- Podemos utilizar PD bottom-up para controlar el orden en el que calculamos los estados.
- Complejidad final: $O(N)$ memoria y $O(N * \sqrt{N})$ tiempo.

# Implementación Bottom-Up

```cpp
int main(){
    cin.tie(0); cin.sync_with_stdio(0);
    int n; ll g;
    cin>>n>>g;
    vector<ll> t(n+1), m[2];
    forn(i,n){ cin>>t[i+1]; t[i+1] += t[i];}
    forn(i,2) m[i].resize(n+1,0);
    forsn(l,1,maxl){
        swap(m[0],m[1]);
        forsn(i,1,n+1){
            m[0][i] = max(m[0][i-1],m[1][i]);
            if(i>=l) m[0][i] = max(m[0][i], m[1][i-l] - t[i] + t[i-l] + g);
        }
    }
    cout<<m[0][n]<<"\n";
}
```

# Implementación Bottom-Up

```cpp
int main(){
    cin.tie(0); cin.sync_with_stdio(0);
    int n; ll g;
    cin>>n>>g;
    vector<ll> t(n+1), m[2];
    forn(i,n){ cin>>t[i+1]; t[i+1] += t[i];}
    forn(i,2) m[i].resize(n+1,0);
    forsn(l,1,maxl){
        swap(m[0],m[1]);
        forsn(i,1,n+1){
            m[0][i] = max(m[0][i-1],m[1][i]);
            if(i>=l) m[0][i] = max(m[0][i], m[1][i-l] - t[i] + t[i-l] + g);
        }
    }
    cout<<m[0][n]<<"\n";
}
```

**AC: 343ms**

# Tips - I

- Resolver ejemplos pequeños a mano puede ayudar a ganar una intuición de que es lo que buscamos plasmar en la función.
- Es clave poder explicar la firma de su función en palabras, utilizando cada parámetro para la explicación.
- Piensen con cuidado cuáles pueden ser sus casos borde y base, para no olvidarse ninguno.
- Recuerden de utilizar el peor caso para la complejidad de dinámica y backtracking.

# Tips - II

- Pueden pensar los estados en base a la memoria y la complejidad necesaria.
- Es recomendable empezar con la función recursiva y luego pasar al código, la transición es directa.
- Sus casos base deben ser el neutro de una operación, por ejemplo el $-\infty$ es el neutro del máximo.
- Para calcular superposición de problemas recuerden utilizar $\omega$ y $O$ dónde corresponde, no prueben un caso puntual!
- Hagan los ejercicios de las guías y consúltenlos, aprovechen los espacios de clase.
- Si necesitan reducir la complejidad de su función piensen cómo pueden modificar sus parámetros para lograr esto.

# Ejercicios de la clase de hoy

- Astro Trade: https://codeforces.com/problemset/problem/866/D
- Mi Buenos Aire Crecido: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2890
- Guirnaldas: https://codeforces.com/contest/1286/problem/A
- Caesars Legion: https://codeforces.com/problemset/problem/118/D
- Lagunas: https://codeforces.com/gym/106054/problem/L
- Ejercicio bonus: Ejercicio mega picante