A. Minimizing Coins

- Hay n valores de monedas $c = [c_1, c_2, ..., c_n]$. Queremos el mínimo número de monedas que necesitamos para sumar un valor x.
- Ejemplo: $n = 3, c = [2, 3, 5]$ y $x = 10$ basta usar dos monedas de valor 5

  $$5 + 5 = 10$$

  y es el mínimo.
- Límites $1 \leq n \leq 10^2$, $1 \leq x \leq 10^6$ y $1 \leq c_i \leq 10^6$.

Primera idea (No sirve)

Intentamos un greedy, agregamos a nuestra suma la moneda más grande posible sin que se pase de x.

Esto no funciona.

- Podría pasar que no logremos sumar exactamente x cuando si es posible. Por ejemplo, si queremos sumar 11 en el ejemplo anterior, empezamos con 5, luego agregamos otro 5 y quedamos con $5 + 5$ y no podemos agregar 1 entonces quedamos sin solución. Pero $2 + 3 + 3 + 3 = 11$ si había solución.
- Podría pasar que obtengamos una suma que usa más monedas que la mínima posible. Por ejemplo
  - $n = 4$,
  - $c = [1, 7, 8, 9]$,
  - y $x = 15$.

  Si empezamos con 9 estamos obligados a rellenar con 1's y queda

  $$9 + 1 + 1 + 1 + 1 + 1 + 1 = 15$$

  que usa 7 monedas pero

  $$7 + 8 = 15$$

  usa 2.

Abandonamos la idea de usar greedy, pasamos a programación dinámica.

Segunda idea (Si sirve)

Sea dp_x el mínimo número de monedas que debemos usar para lograr una suma igual a x.

Para calcular dp_x:

- Están las formas tales que la última moneda usada es c_1. Estas formas se ven como

  $$? + c_1.$$

de todas estas, la que usa menos monedas usa $dp_{x-c_1} + 1$ monedas pues dp_{x-c_1} es el mínimo número de monedas que podemos usar para la parte desconocida y hay que agregarle el último uso de la moneda c_1.

- Están las formas tales que la última moneda es c_2 donde la mínima sería $dp_{x-c_2} + 1$.
- Y así hasta la última moneda c_n.
- El mínimo número de monedas a usar tal que la última es c_i es $dp_{x-c_i} + 1$.

  Pero ¿cómo podemos determinar cuál moneda usar como última para lograr un óptimo para dp_x? No importa, las probamos todas y nos quedamos con el mínimo:

$$dp_x = \min_{i=0}^{n-1}(dp_{x-c_i} + 1)$$

  Esto funciona para $x > 0$. Entonces dp_0 es un caso base y vale 0 pues no necesitamos usar monedas para tener una suma igual a 0.

  Cuidado cuando $x - c_i < 0$.

¿Es suficientemente buena?

Tiene $O(x)$ estados y el cálculo de cada uno de ellos es $O(n)$ pues iteramos sobre todas las monedas. La complejidad temporal queda $O(x \cdot n)$ y para los límites del problema esto es 10^8. Pasamos a la implementación.

B. Investigating Zeroes and Ones

- Tenemos un arreglo binario $a = [a_0, a_1, ..., a_{n-1}]$ de n elementos.
- Queremos contar el número de subarreglos con una cantidad impar de 1's.
- Límites: $1 \leq n \leq 10^5$.

Primera idea (No sirve)

Intentamos todos los subarreglos.

El número de subarreglos que terminan en un índice i es $i + 1$. Entonces el número total de subarreglos es:

$$\sum_{i=0}^{n-1} i + 1 = \frac{n \cdot (n+1)}{2}$$

El número de subarreglos es $O(n^2)$, aunque probar uno fuese $O(1)$ seguiría siendo demasiado costoso pues n^2 para el máximo n es 10^{10}.

Segunda idea (Si sirve)

Usamos programación dinámica.

Sea

- PA_i el número de intervalos que terminan en i con una cantidad par de 1's e
- IM_i el número de intervalos que terminan en i con una cantidad impar de 1's.

Para calcular PA_i:

- Si a_i es 1:
  - El intervalo de largo 1 que termina en i corresponde a $[1]$ y tiene un número impar de 1's. Este intervalo no es parte de la cuenta.
  - Los intervalos más largos que terminan en i con una cantidad par de 1's corresponden a intervalos con una cantidad par de 1's que terminan en $(i-1)$ junto al 1 en la posición i.
  - Queda $PA_i = IM_{i-1}$

  Si a_i es 0:
  - El intervalo de largo 1 que termina en i corresponde a $[0]$ y tiene un número par de 1's. Este intervalo es parte de la cuenta.
  - Los intervalos más largos que terminan en i con una cantidad par de 1's corresponden a intervalos con una cantidad par de 1's que terminan en $(i-1)$ junto al 0 en la posición i.
  - Queda $PA_i = 1 + PA_{i-1}$

Para calcular IM es análogo

$$PA_i = \begin{cases} IM_{i-1} & \text{si } a_i = 1 \\ 1 + PA_{i-1} & \text{si } a_i = 0 \end{cases}$$

$$IM_i = \begin{cases} 1 + PA_{i-1} & \text{si } a_i = 1 \\ IM_{i-1} & \text{si } a_i = 0 \end{cases}$$

La respuesta será

$$\sum_{i=0}^{n-1} IM_i$$

Hay $O(n)$ estados y cada estado se calcula en tiempo $O(1)$ constante. La complejidad temporal queda $O(n)$.

C. Bishwock

- Tenemos una matriz de "X" y "." de tamaño $2 \times n$.
- Queremos la máxima cantidad de L's que podemos colocar tal que no se solapan, no se salen de la matriz, están alineada a los ejes y solo usan posiciones con ".".
- Límites: $1 \leq n \leq 10^2$.

Primera idea (Si sirve)

Idea greedy.

Vamos de atrás hacia adelante y colocamos L's siempre que podamos. Para saber si podemos colocar una L usando las columnas i e $i + 1$ basta con saber si el número de casillas libres es 3 o más.

- Si las casillas libres son 3 entonces las usamos todas.
- Si las casillas libres son 4 entonces usamos la columna i completa (pues de todas formas estas casillas no nos servirán en el futuro) y una casilla de la columna $i + 1$. ¿Importa cuál casilla de la columna $i + 1$ usemos? No pues solo nos importa el número de casillas libres.

Funciona pues en la columna i, no volveremos a ver las casillas de esta columna por lo que nos interesa usarlas todas, y en la columna $i + 1$, si no usamos una casilla y la usamos en un paso futuro, simplemente tendremos menos casillas para futuras L's.

Como vamos avanzando columna por columna, y en cada columna la decisión se toma en tiempo constante, la complejidad temporal queda $O(n)$.

Segunda idea (Si sirve)

Con programación dinámica:

Tenemos $dp[i][b]$ = máximo número de L's que podemos dejar tal que solo consideramos hasta la $i-$ésima columna y

- Si $b = 0$ entonces la columna i queda libre. Podemos colocar lo que queramos en la columna.
- Si $b = 1$ entonces el primer valor de la columna i está ocupado y el segundo no.
- Si $b = 2$ entonces el segundo valor de la columna i está ocupado y el segundo no.
- Si $b = 3$ entonces ambos valores de la columna i están ocupados.

b lo vemos como una máscara de bits que indica cómo termina la columna i.

La dp la empezamos con -1.

Vamos avanzando por las columnas. Si consideramos la primera columna solo podemos dejar 0 piezas. Entonces dejamos $dp[0][b] = 0$ con b indicando las posiciones ocupadas de la columna 0.

Para $i > 0$. Primero obtenemos el b que dejan los peones de la columna i Este valor lo dejamos con el máximo considerando las columnas anteriores.

$$dp[i][b] = \max_{0 \le b' \le 3} \{dp[i-1][b']\}$$

Luego vemos si podemos dejar alguna pieza.

- Si b es 0 entonces podemos dejar piezas de 4 formas distintas.

  $$dp[i][3] = \max\{1 + dp[i-1][0], 1 + dp[i-1][1], 1 + dp[i-1][2], dp[i][3]\}$$
  $$dp[i][2] = \max\{1 + dp[i-1][0], dp[i][2]\}$$
  $$dp[i][1] = \max\{1 + dp[i-1][0], dp[i][1]\}$$

  Para cada $dp[i-1][x]$ debemos verificar que no sea -1 para usarlo.

- Si b es 1 entonces podemos dejar piezas de 1 sola forma

  $$dp[i][3] = \max\{1 + dp[i-1][0], dp[i][3]\}$$

- Si b es 2 entonces similar a lo anterior.
- Si b es 3 no podemos hacer nada y seguimos a la siguiente columna.

Luego imprimimos el máximo de $dp[n-1][k]$ con k entre 0 y 3.

Esta dp tiene $O(n \cdot 2^2) = O(n)$ estados y el cálculo de cada estado es constante $O(1)$. La complejidad temporal queda $O(n)$.

D. Grid Paths

- Tenemos una grilla G de $n \times n$ que contiene "." y "*".
- Queremos el número de caminos que van desde $(0, 0)$ hasta (n, n) en la grilla tal que solo se puede avanzar hacia la derecha o abajo que no pasen por casillas con "*".
- Debemos entregar el resultado módulo $10^9 + 7$.
- Límites: $1 \leq n \leq 10^3$.

Primera idea (No sirve)

Podemos probar todos los caminos posibles, cada vez que llegamos a la casilla (n, n) sumamos 1.

Esto es demasiado lento. El número total de caminos se da cuando $n = 1000$ y no hay casillas con "*" y este valor es $\binom{1000}{500}$, $O(2^n)$. Es demasiado para probar todos los caminos.

Segunda idea (Si sirve)

Usamos programación dinámica.

Sea $dp_{i,j}$ el número de caminos que van de $(0, 0)$ a (i, j) módulo $10^9 + 7$.

Para calcular $dp_{i,j}$:

- Si $G_{i,j}$ es "." entonces $dp_{i,j} = 0$.
- De otra forma
  - Los caminos que terminan en (i, j) tal que el último movimiento fue hacia la derecha son los caminos que terminan en $(i, j - 1)$ a los que les agregamos un movimiento a la derecha. Entonces el número de caminos que terminan en (i, j) con último movimiento hacia la derecha es $dp_{i,j-1}$.
  - Los caminos que terminan en (i, j) tal que el último movimiento fue hacia abajo son los caminos que terminan en $(i - 1, j)$ a los que les agregamos un movimiento hacia abajo. Entonces el número de caminos que terminan en (i, j) con último movimiento hacia abajo es $dp_{i-1,j}$.
  - Queda

$$dp_{i,j} = (dp_{i-1,j} + dp_{i,j-1}) \mod (10^9 + 7)$$

    Consideramos $dp_{i,j}$ como 0 si $i < 0$ o $j < 0$.
- Esto solo funciona si $i > 0$ o $j > 0$. El caso $dp_{0,0}$ es un caso base y $dp_{0,0} = 1$ pues existe el camino sin movimientos.

La dp tiene n^2 estados el cálculo de un estado es $O(1)$. Entonces la complejidad temporal queda $O(n^2)$ que para los límites del problema es 10^6 y pasa entero bien.

E. Interactivity

- Tenemos un árbol G de n nodos.
- El valor de un nodo es la suma de valores de sus hojas.

  Una configuración corresponde a los valores de un conjunto de nodos.

  Una configuración buena minimal es una configuración que nos permite conocer el valor de todos los nodos del árbol tal que si eliminamos un nodo de la configuración, deja de permitirnos conocer el valor de todos los nodos del árbol.

  ¿Cuántas configuraciones buenas minimales existen que nos permiten conocer el valor de cada nodo? Este valor debe entregarse módulo $10^9 + 7$.

Primera idea (No sirve)

Podemos intentar todas las 2^n configuraciones y ver si sirven o no. Esto es demasiado lento para los límites de n.

Segunda idea (Si sirve)

Programación dinámica.

Tendremos $dp_{i,0}$ como el número de configuraciones minimales sobre el subárbol de i que permiten conocer el valor de todos los nodos del subárbol de i tal que no podemos conocer el valor del nodo i usando el resto del grafo (pero podríamos conocerlo si lo metemos a la configuración).

Tendremos $dp_{i,1}$ como el número de configuraciones minimales sobre el subárbol de i que permiten conocer el valor de todos los nodos del subárbol de i tal que si podemos conocer el valor del nodo i usando el resto del grafo.

- Para $dp_{i,1}$: Usando el resto del grafo podemos conocer el valor del nodo i (la raíz del subárbol sobre el cual estamos trabajando).

  Necesitamos conocer el valor de todos los hijos de i excepto 1 que despejaremos usando los valores conocidos.

  Sea h_j el $j-$ésimo hijo del nodo i (partiendo en 0) y sea m el número de hijos de i. Calculamos

$$L_j = \prod_{k=0}^{j-1} dp_{h_k,0} = L_{j-1} \cdot dp_{h_{j-1},0}$$

  donde L_j es el número de configuraciones sobre los subárboles de los hijos de i hasta el $(j-1)-$ésimo que nos permiten conocer todos estos nodos sin ayuda del resto del grafo. Y

$$R_j = \prod_{k=j+1}^{m-1} dp_{h_k,0} = R_{j+1} \cdot dp_{h_{j+1},0}$$

donde R_j es el número de configuraciones sobre los subárboles de los hijos de i desde el $(j+1)-$ésimo hasta el último que nos permiten conocer todos estos nodos sin ayuda del resto del grafo.

Luego, si queremos dejar al hijo $j-$ésimo como el cual despejaremos usando al resto, el número de configuraciones que permite esto es

$$L_j \cdot dp_{h_j,1} \cdot R_j$$

Y como podemos dejar a cualquier hijo como el incógnito, obtenemos la suma

$$dp_{i,1} = \sum_{j=0}^{m-1} L_j \cdot dp_{h_j,1} \cdot R_j$$

- Para $dp_{i,0}$:
  - Como no podemos conocer el nodo i usando el resto del grafo, tenemos la opción de que esté en la configuración. El número de configuraciones tal que el nodo i está es $dp_{i,1}$ pues es el caso donde asumimos que podemos determinar el valor del nodo i.
  - El número de configuraciones donde no podemos determinar el valor del nodo i usando el resto del grafo y el nodo i no está en la configuración es igual al número de configuraciones que nos permiten determinar todos los subárboles de los hijos de i pues es la única forma en la que podemos despejar i.

    Nuevamente tomamos h_j como el $j-$ésimo hijo de i y m como el número de hijos de i.

    El número de configuraciones donde podemos determinar a todos los hijos es $\prod_{j=0}^{m-1} dp_{h_j,0}$ que es L_{m-1}

    Queda

$$dp_{i,0} = dp_{i,1} + L_{m-1}$$

Hay $n \times 2$ estados, el cálculo del estado (i, b) toma tiempo $O(v_i)$ donde v_i es el número de hijos del nodo i. Entonces el tiempo total será

$$\sum_{b=0}^{1} \sum_{i=0}^{n-1} v_i = \sum_{b=0}^{1} (n-1) = 2 \cdot (n-1)$$

La complejidad temporal queda $O(n)$.

F. Interesting Problem (Easy Version)

- Tenemos un arreglo $a = [a_1, a_2, ..., a_n]$ de n elementos.
- Podemos realizar la siguiente operación el número de veces que queramos:
  - Escogemos un índice i con $1 \leq i < |a|$ con $a_i = i$.
  - Eliminamos a_i y a_{i+1} del arreglo y concatenamos el resto. Los índices de los valores del arreglo se reasignan.

  ¿Cuál es el número máximo de veces que se puede realizar esta operación?
- Límites: $1 \leq n \leq 100$, $1 \leq a_i \leq n$.

Primera idea (No sirve)

Un greedy donde vamos de izquierda a derecha eliminando siempre que podamos.

No sirve.

El caso $a = [1, 2, 3, 4]$, si eliminamos el 1 y el 2 ya no podremos eliminar el 3 ni el 4, es mejor eliminar el 3 primero.

Segunda idea (No sirve)

Un greedy donde vamos de derecha a izquierda eliminando siempre que podamos.

No sirve.

El caso $a = [1, 1, 3, 4, 1, 2, 2, 1]$:

- Eliminamos $4, 1$ queda $a = [1, 1, 3, 2, 2, 1]$.
- Eliminamos $3, 2$ queda $a = [1, 1, 2, 1]$.
- Eliminamos $1, 1$ queda $a = [2, 1]$.

Realizamos la operación 3 veces.

Pero:

- Eliminamos $3, 4$ queda $a = [1, 1, 1, 2, 2, 1]$.
- Eliminamos $1, 1$ queda $a = [1, 2, 2, 1]$.
- Eliminamos $2, 2$ queda $a = [1, 1]$.
- Eliminamos $1, 1$ queda $a = []$.

Se puede realizar la operación 4 veces.

Tercera idea (Si sirve)

Tendremos $dp_{i,j,v}$ que es el máximo número de elementos que podemos eliminar en el rango $[i, j]$ considerando que el rango completo sigue existiendo, que el índice real de a_i es v (o que hay $v-1$ elementos antes del rango) y que solo haremos operaciones en este rango.

Si logramos calcular la dp la respuesta será $dp_{1,n,1}$.

Para obtener $dp_{i,j,v}$ tenemos dos opciones.

- Nunca eliminamos a a_i. En este caso la respuesta será simplemente $dp_{i+1,j,v+1}$ pues queda ver lo mejor que podemos hacer con el resto del rango, y como a_i no se eliminó, el elemento a_{i+1} queda en índice $v + 1$.
- De alguna manera eliminamos al elemento i.

  Si queremos eliminar al elemento i, en algún momento este elemento a_i debe haber cumplido $a_i = i$. Para que esto se cumpla a_i debe ser mayor o igual a i (pues su posición solo puede disminuir) y v debe ser menor o igual a i para asegurar que ha pasado por la posición correcta. Además debemos asegurar que no se ha saltado la posición correcta, recuerden que los elementos se mueven de a 2. Entonces a_i debe tener misma paridad que i.

  Si eliminamos el elemento i con un elemento k tal que $i < k \leq j$ se debe cumplir que todo el rango entre $i + 1$ y $k - 1$ se pueda eliminar con a_i en una posición mayor o igual a i, entonces necesitamos que $dp_{i+1,k-1,a_i+1}$ sea igual a $(k - 1) - (i + 1) + 1$, el largo del rango. Luego, asumimos que pudimos eliminar todo desde i hasta k y lo que sigue lo resolvemos con $dp_{k+1,j,v}$.

  Entonces queda

$$dp_{i,j,v} = \max\{dp_{i+1,j,v+1}, \max_{i<k\leq j, k \text{ bueno}} \{k - i + 1 + dp_{k+1,j,v}\}\}$$

Finalmente respondemos $dp_{1,n,1}/2$.

La dp tiene $O(n^3)$ estados y cada estado toma tiempo $O(n)$ en calcularse, la complejidad temporal es $O(n^4)$ y con los límites queda en 10^8.

G. Black and white

- Tenemos una matriz M de $n \times n$ valores “B” o “N”.
- Queremos rellenarla con piezas de 1×2 de forma horizontal (no se puede poner vertical).

  ¿Cuál es la máxima cantidad de estas piezas que podemos colocar?
- Límites: $1 \leq n \leq 50$.

Primera idea (Si sirve)

Como no se pueden poner vertical, podemos resolver cada fila por separado.

El greedy donde vamos de izquierda a derecha colocando piezas apenas podamos funciona por la misma razón del problema Bishwock.

La complejidad temporal queda $O(n^2)$.

H. Barcode

- Tenemos una matriz M de $n \times m$ llena de "." y "#".
- Una matriz M es un código de barras si
  - Toda columna contiene solo uno de los dos caracteres.
  - El número de columnas de cada secuencia maximal de columnas consecutivas de mismo caracter se encuentra entre x e y (inclusive).

  ¿Cuál es el mínimo número de posiciones que debemos cambiar para que la matriz M sea un código de barras?
- Límites: $1 \leq n, m, x, y \leq 1000$.

Primera idea (Si sirve)

Programación dinámica.

Notamos primero que en una columna solo nos importa el número de posiciones con "." y con "#". La fila en la que se encuentran es irrelevante. Para cada columna contamos el número de posiciones con valor "#".

Tenemos un arreglo $a = [a_1, a_2, ..., a_m]$ donde a_i es el número de posiciones con valor "#" en la columna $i-$ésima.

Usamos $dp_{i,0}$ como el mínimo número de colores que debemos cambiar para que las columnas desde la 1 hasta la i sean un código de barras cuya última columna es de caracteres "#".

Usamos también $dp_{i,1}$ análogo a lo anterior pero con caracter ".".

También tendremos $c(i, j, 0)$ que es el costo de dejar solo el caracter "." en las columnas de la i hasta la j y $c(i, j, 1)$ análogo pero para el caracter "#".

Para calcular $dp_{i,0}$:

- Si la última secuencia consecutiva maximal de columnas con caracter "." es de largo j entonces tenemos $dp_{i-j,1} + c(i - j + 1, i)$.

  El largo j puede estar entre x e y entonces

  $$dp_{i,0} = \min_{j=x}^{y} \{dp_{i-j,1} + c(i - j + 1, i)\}$$

  Hay que tener cuidado con los casos en donde no se puede. Se recomienda dejar ∞ en esos estados. El caso base sería $dp_{0,0} = 0$ y $dp_{0,1} = 0$ pues en este problema indexamos de 1 y la columna 0 sería una columna que no existe.

Para calcular $dp_{i,1}$ es análogo.

Hay $O(n)$ estados y cada estado toma tiempo $O(n)$ en calcularse, la complejidad temporal queda $O(n^2)$.

I. Coin Combinations I

- Hay n valores distintos de monedas $c = [c_1, c_2, ..., c_n]$. Queremos el número de formas con orden en que podemos sumar un valor x.
- Ejemplo: $n = 3, c = [2, 3, 5]$ y $x = 10$ hay muchas formas de sumar 10, algunas son:

$$2 + 2 + 3 + 3$$
$$2 + 3 + 2 + 3$$
$$5 + 2 + 3$$

- Límites $1 \leq n \leq 10^2$, $1 \leq x \leq 10^6$ y $1 \leq c_i \leq 10^6$.

Primera idea (Si sirve)

Programación dinámica.

Sea dp_x el número de formas en que podemos sumar x.

Para calcular dp_x:

- Están las formas que usan como última moneda la moneda c_1. Estas corresponden a $dp_{i,x-c_1}$ pues la parte desconocida de la suma se puede hacer de esa cantidad de formas.
- Están las formas tales que la última moneda es c_2 que corresponden a dp_{x-c_2}.
- Y así hasta la última moneda c_n.
- El número de formas de sumar x usando como última moneda la moneda c_i es dp_{x-c_i}.

  Ahora solo debemos sumar todas estas posibilidades

$$dp_x = \sum_{i=0}^{n-1} (dp_{x-c_i})$$

  Esto funciona para $x > 0$. Entonces dp_0 es un caso base y vale 1 pues no tener monedas es una forma de sumar 0.

  Cuidado cuando $x - c_i < 0$.

¿Es suficientemente buena?

Tiene $O(x)$ estados y el cálculo de cada uno de ellos es $O(n)$ pues iteramos sobre todas las monedas. La complejidad temporal queda $O(x \cdot n)$ y para los límites del problema esto es 10^8. Pasamos a la implementación.

J. Two Ends

- Tenemos un arreglo $a = [a_0, a_2, ..., a_{n-1}]$ de largo n par.
- Hay dos jugadores y juegan por turnos. En un turno pueden tomar una carta de un extremo y agregar el valor de la carta a su puntaje.
- El segundo jugador juega de forma greedy tomando siempre la carta de mayor valor entre las cartas de los extremos. El primer jugador puede hacer lo que quiere.
- Límites $1 \leq n \leq 10^3$, $0 \leq \sum a_i \leq 10^6$.

Primera idea (Si sirve)

Programación dinámica.

Sea $dp_{i,j}$ la diferencia de puntaje entre el primer jugador y el segundo cuando solo están las cartas desde la i hasta la j (y esas son las de los extremos).

Para calcular $dp_{i,j}$: El primer jugador tiene dos opciones

- El primer jugador toma la carta i, y el segundo jugador toma la mejor carta entre la $i+1$ y la j.
  - Si la carta que toma el segundo jugador es la $i+1$ entonces quedan las cartas de $i+2$ hasta j y el puntaje se modifica por $a_i - a_{i-1}$. Toma el valor $dp_{i+2,j} + a_i - a_{i-1}$.
  - Si la carta que toma el segundo jugador es la j entonces quedan las cartas de $i+1$ a $j-1$ y el puntaje se modifica por $a_i - a_j$. Toma el valor $dp_{i+1,j-1} + a_i - a_j$.
- El primer jugador toma la carta j, y el segundo jugador toma la mejor carta entre la i y la $j-1$.
  - Si la carta que toma el segundo jugador es la i entonces quedan las cartas de $i+1$ a $j-1$ y el puntaje se modifica por $a_j - a_i$. Toma el valor $dp_{i+1,j-1} + a_j - a_i$.
  - Si la carta que toma el segundo jugador es la $j-1$ entonces quedan las cartas de i a $j-2$ y el puntaje se modifica por $a_j - a_{j-1}$. Toma el valor $dp_{i,j-2} + a_j - a_{j-1}$.

Claramente se queda con el mayor valor entre ambas opciones.

La diferencia queda en $dp_{0,n-1}$.

Hay $O(n^2)$ estados y el cálculo de cada estado toma tiempo constante $O(1)$. La complejidad temporal queda $O(n^2)$.

K. Mountains

- Tenemos un arreglo $a = [a_1, a_2, ..., a_n]$ de "+" y "-".
- Queremos encontrar el k que maximiza

$$\sum_{i=1}^{k} (a_i = +) - (a_i = -)$$

- Límites $1 \leq n \leq 10^5$.

Primera idea (Si sirve)

Programación dinámica.

Primero obtenemos un arreglo b de largo n tal que $b_i = 1$ si a_i es + y -1 si no.

Tenemos $dp_i = \sum_{j=1}^{i} b_i = dp_{i-1} + b_i$. Considerando dp_0 como 0.

Ahora solo buscamos el i tal que dp_i es máximo.

dp tiene $O(n)$ estados y cada estado se calcula en tiempo constante $O(1)$. La complejidad temporal queda $O(n)$.