# Ejercicio Greedy DAA

Francisco Vicente Suárez Bellón C-412

Septiembre de 2024

# Índice

# 1. Enunciado del Problema

ErnKor está listo para hacer cualquier cosa por Julen, incluso nadar a través de pantanos infestados de cocodrilos. Decidimos poner a prueba este amor. ErnKor tendrá que nadar a través de un río con un ancho de 1 metro y una longitud de $n$ metros.

El río está muy frío. Por lo tanto, en total (es decir, durante toda la natación desde 0 hasta $n + 1$) ErnKor puede nadar en el agua no más de $k$ metros. Por el bien de la humanidad, hemos añadido no solo cocodrilos al río, sino también troncos sobre los que puede saltar. Nuestra prueba es la siguiente:

Inicialmente, ErnKor está en la orilla izquierda y necesita llegar a la orilla derecha. Estas se encuentran a 0 y $n + 1$ metros respectivamente. El río se puede representar como $n$ segmentos, cada uno con una longitud de 1 metro. Cada segmento contiene un tronco 'L', un cocodrilo 'C' o solo agua 'W'. ErnKor puede moverse de la siguiente manera:

- Determina si ErnKor puede llegar a la orilla derecha.
- La primera línea de cada caso de prueba contiene tres números $n$, $m$, $k$ (donde $0 \leq k \leq 2 \cdot 10^5$, $1 \leq n \leq 2 \cdot 10^5$, $1 \leq m \leq 10$) — la longitud del río, la distancia que ErnKor puede saltar y el número de metros que ErnKor puede nadar sin congelarse.
- La segunda línea de cada caso de prueba contiene una cadena $a$ de longitud $n$. $a_i$ denota el objeto ubicado en el $i$-ésimo metro. ($a_i \in \{'W', 'C', 'L'\}$)

## 1.1. Reglas de Movimiento

- Si está en la superficie (es decir, en la orilla o en un tronco), no puede saltar hacia delante más de $m$ metros (puede saltar a la orilla, a un tronco o al agua).
- Si está en el agua, solo puede nadar hasta el siguiente segmento de río (o hasta la orilla si está en el agua y el agua está a $n$-ésimos metros).
- ErnKor no puede aterrizar en un segmento con un cocodrilo de ninguna manera.

## 1.2. Pregunta

Determine si ErnKor puede llegar a la orilla derecha.

# 2. Soluciones

## 2.1. Descripción del problema

Este problema es un problema de satisfacción de restricciones donde:

- La cantidad de metros que se nada en total no puede superar las $k$ en total.
- Nunca se puede caer en una casilla de un cocodrilo.
- Desde una orilla o tronco se puede saltar hasta $m$ casillas ($m \leq 10$).
- El objetivo es llegar a la otra orilla.

Para ello puede crearse un código de backtrack donde por cada vez que se puede saltar se analizan todos los escenarios posibles, y si en alguno se llega a la otra orilla habiendo cumplido las restricciones se puede decir que es posible.

Como optimización de esta idea puede tomarse que en cada llamado recursivo cada vez que se llamen a ver una rama del árbol con respecto a ver a que lugar saltar si la cantidad de metros que se nado es mayor o igual que el mínimo que se tenia anteriormente (el cual cumplió con las restricciones) se poda esa rama de la recursividad, dado que no es posible disminuir la cantidad de metros nadados, solo se pueden mantener o aumentar.

## 2.2. Solución Fuerza Bruta

Para resolver el problema inicialmente podemos generar todos los caminos posibles, lo cual es análogo a generar todas las cadenas binarias y despues recorrer cada una de estas cadenas llevando el contador de k, y donde este en True contamos la posible instancia:

1. Si estamos encima de un tronco no variamos
2. Si estamos encima de un cocodrilo inmediatamente sabemos que esa cadena no lleva a la solución
3. Si estamos encima de una casilla de agua disminuimos w en una unidad.

### 2.2.1. Posibles mejoras de este

El algoritmo antes descrito tiene un coste **Temporal:** $O(2^n)$ el cual es muy costoso como para ser efectivo en terminos reales. Algunas podas que se aconseja son:

1. Cuando podemos hacer un generador lazy el cual espera a que verifiquemos la ultima cadena generada para generar la siguiente (Esta solución en la mayoría de casos encuentra más rápido la solución en caso de poder realizar el cruce) en el peor de los casos la complejidad sigue siendo la misma

2. Si conocemos que cierta casilla por tener un cocodrilo u otra razón siempre nos va a causar falla no generar las cadenas que tienen este.

3. Paralelizar esto, se puede tener un algoritmo lazy que entregue cadenas distintas a diferentes hilos o procesos con el objetivo de hacer más rápida la búsqueda.

Aún así el algoritmo no deja de tener un costo muy alto en el peor de los casos.

#### 2.2.2. Correctitud del algoritmo

El algoritmo al tener todas las cadenas binarias si existe una forma de cruzar que respete el problema existirá una cadena binaria que represente el caminos si en el camino no existe forma de cruzar respetando las restricciones cualquier forma de cruce representada en el algoritmo dará que es imposible.

#### 2.2.3. Análisis de complejidad

**Temporal:** El costo temporal es $O(2^n)$ dado que es el costo de generar los posibles mapas y a medida que se genera leerlos.

## 3. Solución Greedy

Para resolver el problema de manera greedy hay que tener en cuenta lo siguiente:

- Desde la orilla o cualquier tronco trataremos de saltar al tronco más cercano:
    - Esto no afecta en la solución dado que al saltar al tronco u orilla más cercano no aumenta la cantidad de metros nadados.
    - Saltar al primer tronco u orilla no afecta a poder llegar a otro tronco anteriormente alcanzable dado que si desde el tronco u orilla anterior se podía llegar al anterior porque estaba a una distancia $\leq m$, entonces desde este la distancia será también $\leq m$.
    - Si desde el tronco u orilla no es posible saltar a otra orilla u tronco en los $m$ metros posibles, saltar al pedazo de agua más lejano:
        - Saltar al pedazo de agua más lejano hace que no se tengan que nadar más metros que si se saltara a una distancia inferior.
        - Si hay algún cocodrilo a una distancia menor que la que se saltó, no afectará.
        - Si existe un cocodrilo en la casilla siguiente al lugar donde se saltó, puede ser que se llegó a una menor al máximo que se podía saltar y de cualquier forma que se salte es imposible no caer en la casilla del cocodrilo si es mediante nados consecutivos análogos.

- Como a cada tronco se llega tratando de minimizar la cantidad de metros nadados, eso implica que a la otra orilla, en caso de llegar, se hará en la menor cantidad de metros nadados.

# Notaciones

- Orilla izquierda: $O_{\text{izq}}$
- Orilla derecha: $O_{\text{der}}$
- $T_{\text{sección}}$: Son todos los cuadrantes a los que se puede saltar a la derecha desde el tronco $T$ en su rango $m$ (Si hay un tronco $T'$ en ese rango, la sección acaba en la casilla anterior de $T$) y del espacio anterior hasta $T$ que no sea de una $T_{\text{sección}}$ previa. (En cada $T_{\text{sección}}$ solo hay un tronco).
- Rango de $T_{\text{sección}}$: Todos los cuadrantes que se pueden saltar desde ese tronco hacia la derecha.
- Salto más lejano dentro del rango de la $T_{\text{sección}}$: $Max_{\text{jump}}$ sin que se salte arriba.
- Metros nadados en total: $s$.
- Metros nadados hasta el i-ésimo tronco: $s_i$.

## Observaciones

- Si dentro del rango de la $T_{\text{sección}}$ no existe ningún tronco u orilla a la que saltar, entonces se saltará al final de la $T_{\text{sección}}$.
  - Si en el final de la $T_{\text{sección}}$ hay un cocodrilo, entonces $s = \infty$.
- Si nadando el próximo segmento es un cocodrilo, entonces $s = \infty$.
- En caso de no haber $T_{\text{sección}}$, la forma de conocer el mínimo de metros a nadar es calcular todos los cuadrantes hasta la orilla; si hay algún cocodrilo, el mínimo no existirá ($s = \infty$).

## Demostración

Demostrar que un problema donde se aplique el salto entre $T_{\text{secciones}}$, siempre que sea posible, da el mínimo de metros a nadar.

### Caso base

Para cantidad de $T_{\text{sección}} = 1$:

- Se puede saltar desde $O_{\text{izq}}$ al tronco $T_1$, entonces $s_1 = 0$.
- Si no se puede saltar hasta la $T_{\text{sección}}$, entonces $s_1 =$ distancia hasta $T_1$.

- Si $O_{\text{der}}$ está en el rango de la $T_{\text{sección}}$, entonces $s = s_1$.
- En caso contrario, $s = s_1$+(Cantidad de metros que hay desde $Max_{\text{jump}}$ hasta la orilla), dándose cuenta que $s$ es mínimo dado que es obligatorio nadar hasta la $O_{\text{der}}$.

**Hipótesis**

Supongamos que para todo mapa con cantidad de $T_{\text{sección}} = m$, se cumple que el método anterior me arroja el mínimo de metros a nadar.

**Tesis**

Sea el mapa $Map$ con cantidad de $T_{\text{sección}} = m + 1$.

Si tomamos $Map_1$ como las primeras $T_{\text{secciones}}$, y donde acaba la m-ésima sección ponemos la $O_{\text{der}_1}$, y con $Map_2$ desde donde acababa $Map_1$ hasta la $O_{\text{der}}$, por tanto como $Map_1$ y $Map_2$ tienen $\leq m$ cantidad de $T_{\text{secciones}}$, cumplen la hipótesis.

Sea:

$$min_1 = s_{Map_1} \quad \text{y} \quad min_2 = s_{Map_2}.$$

Demostrar que:

$$s_{Map} = min_1 + min_2.$$

- Si en el rango de la última sección hay un tronco, entonces se puede saltar al tronco que está en $Map_2$, por tanto no hubo que aumentar s y como se llegó de forma óptima hasta el último tronco no afectará a lo hecho en $Map_2$, por lo tanto se cumple.

- Si no hay un tronco en el rango de la última sección: Entonces en $Map_2$, el siguiente cuadrante desde la orilla izquierda no es un tronco, por tanto se nadó hasta el tronco lo cual es obligatorio dado que no existe forma de llegar al tronco y por cómo se dividen las secciones tampoco en el mapa, entonces se cumple lo anterior.

## 3.1. Algoritmo

**Código Greedy**

```
def run() -> None:
    n,m,k = map(int, input().split())
    A = input()
    logs = [i for i in range(n) if A[i] == "L"]
    logs.append(n+1)
    i = -1
    next_log = 0
    while i < n-1:
        if m >= logs[next_log] - i:
            i = logs[next_log]
        else:
            i+=m
            if i > n-1:
                print("YES")
                return
            while i < n and i < logs[next_log]:
                if A[i] != "C" and k > 0:
                    i+=1
                    k-=1
                else:
                    print("NO")
                    return
        next_log +=1
    print("YES")

for _ in range(int(input())):
```

### 3.1.1. Correctitud

El algoritmo realiza correctamente la idea antes descrita dado que inicialmente guarda las posiciones de todos los troncos con la idea de tener forma de saber hasta cual puedo saltar El ciclo se va a generar hasta a lo sumo llegar a la otra orilla y en caso de finalizar dirá que fue posible llegar a esta. En el ciclo se comprueba si es posible realizar el salto al tronco mas cercano y en caso de no ser posible salta hasta la distancia maxima en ese caso comprueba que no se caiga en una casilla con un cocodrilo o que se quede sin posibles metros para nadar, en ese caso lanza que no es posible Durante ese momento nadará mientras las condiciones sean las anteriores y no este sobre un tronco. Por lo tanto el algoritmo realiza la idea demostrada anteriormente.

## 3.2. Complejidad

### 3.2.1. Temporal

La idea descrita anteriormente así como la implementación modelo tiene una complejidad de $O(n)$ siendo $n$ la longitud del camino.

### 3.2.2. Espacial

La implementación modelo tiene una complejidad espacial de $O(T)$ donde $T$ es la cantidad de troncos en el caminos dado que esta lo utiliza para el proximo salto $T \leq n$

# Solución de programación dinámica

**Solución usando programación dinámica**: $dp_i$ — el número mínimo de metros que deben ser nadados para llegar a la celda $i$-ésima. El caso base de la programación dinámica es $dp_0 = 0$. Luego, la regla de actualización es:

$$dp_i = \text{mín} \begin{cases} dp_{i-1} + 1 & \text{si } A_i = \text{'W'} \\ \text{mín}(dp_j) & \text{para todo } j, \text{ donde: } \text{máx}(0, i-m) \leq j < i \text{ y } A_j = \text{'L'} \end{cases}$$

## Subestructura Óptima

La subestructura óptima se cumple porque la solución óptima para llegar a una celda $i$ depende de las soluciones óptimas para las celdas anteriores. Se puede descomponer el problema de la siguiente manera:

- Si estamos en una celda tronco **'L'**, el mejor camino hacia esa celda se determina a partir de los caminos óptimos que llegan desde cualquier celda anterior dentro del rango de $m$ pasos.
- Si estamos en una celda de agua **'W'**, perderemos energía al pasar por ella, por lo que la solución óptima dependerá del número mínimo de pasos que podamos hacer sin perder demasiada energía.

La solución óptima para cada celda puede ser calculada usando programación dinámica, aprovechando los resultados previamente calculados para las celdas anteriores.

## Programación Dinámica

La idea es construir una tabla $dp$ donde cada posición $dp[i]$ representa la energía máxima disponible después de llegar a la celda $i$.

### Reglas de Transición

- **Caso base**: $dp[0] = k$ (la energía inicial en la primera celda). - **Transición**:

- Si la celda es tronco **'L'**, se considera la energía máxima de las celdas dentro del rango de $m$ pasos hacia atrás.
- Si la celda anterior fue de agua **'W'**, la energía se reduce en 1 al pasar.

- **Resultado final**: Si la energía al llegar a la última celda es mayor o igual a 0, es posible llegar al final, de lo contrario, no.

## Código en Python

El siguiente código implementa la solución descrita:

```
from collections import deque

t = int(input())
for _ in range(t):
    n, m, k = map(int, input().split())
    s = input()  # String del camino ('C', 'L', 'W')

    dp = [-1] * (n + 2)
    dp[0] = k

    for i in range(1, n + 2):
        if i != n + 1 and s[i - 1] == 'C':
            continue


        for t in range(1, m + 1):
            if i - t >= 0 and (i - t == 0 or s[i - t - 1] == 'L'):
                dp[i] = max(dp[i], dp[i - t])


        if i > 1 and s[i - 2] == 'W':
            dp[i] = max(dp[i], dp[i - 1] - 1)


    if dp[n + 1] >= 0:
        print("YES")
    else:
        print("NO")
```

# Complejidad

## Complejidad Temporal

La complejidad temporal del algoritmo es $O(nm)$, donde:

- $n$ es la longitud del camino.
- $m$ es el número máximo de pasos permitidos hacia atrás.

Para cada celda $i$, evaluamos hasta $m$ celdas anteriores para encontrar la solución óptima, lo que lleva a una complejidad $O(nm)$.

## Complejidad Espacial

La complejidad espacial es $O(n)$, donde $n$ es la longitud del camino. La tabla $dp$ utiliza $n+2$ posiciones para almacenar la energía máxima disponible en cada celda. Esto significa que el espacio utilizado por la solución es lineal respecto a la longitud del camino.