Proyecto 1

Francisco José Guzmán Ángel – 202012332

Contenido

Descripción de la solución.

Para solucionar este problema se pensó específicamente que la mejor manera de obtener menos movimientos en un ordenamiento de inserción de fichas. Se necesita, que casi todas las torres queden del mismo tamaño. Por ello se procede a diseñar un algoritmo que puede resolver el problema.

```
funcion reorganizar_torres(n, torres) -> movimientos {
    cambios := verdadero
    movimientos := 0
    mientras cambios hacer
        cambios := falso
        para i desde 0 hasta n - 2 hacer
            si torres[i] < torres[i + 1] entonces
                torres[i] := torres[i] + 1
                torres[i + 1] := torres[i + 1] - 1
                cambios := verdadero
                movimientos := movimientos + 1
            fin_si
        fin_para
```

```
    fin_mientras

    retornar movimientos
}
```

1. El verdadero plan de acción del algoritmo comienza con la condición de que se necesitan cambios, lo que indica que las torres aún no están en orden descendente.
2. El algoritmo utiliza un bucle principal que se ejecuta repetidamente hasta que ya no haya cambios por hacer. El bucle principal recorre las torres de izquierda a derecha, una por una.
3. En cada iteración del bucle, compara la altura de la torre actual con la altura de la torre siguiente. Si la altura de la torre actual es menor que la altura de la siguiente, el algoritmo mueve una ficha de la siguiente torre a la torre actual.
4. Este movimiento se realiza para asegurar que las torres estén en orden descendente en términos de la cantidad de fichas. Después de realizar un movimiento de fichas, el algoritmo marca que se ha realizado un cambio estableciendo cambios = True.
5. Una vez que no se realizan más cambios en una iteración del bucle, significa que las torres están en orden descendente y el algoritmo sale del bucle.
6. Al final del proceso, todas las torres estarán en orden descendente en términos de la cantidad de fichas. El algoritmo devuelve el número total de movimientos realizados para reorganizar las torres en orden descendente.

Análisis Temporal:

- **Iteración del bucle while**: El ciclo `mientras cambios hacer` se repite hasta que ya no se realicen cambios adicionales en las torres. En situaciones extremas, este ciclo podría ejecutarse repetidamente, especialmente si hay numerosas modificaciones en las alturas de las torres. La complejidad temporal de este ciclo depende de la cantidad de movimientos requeridos para ordenar las torres. En el peor de los casos, sería O(m*n), donde 'm' representa el número de movimientos realizados y 'n' el número de torres.
- **Iteración del bucle for**: Dentro del ciclo while, hay un ciclo for que examina cada torre para realizar los ajustes necesarios. Este ciclo for tiene una complejidad temporal de O(n), donde 'n' es el número de torres.
- **Complejidad temporal total**: La complejidad temporal general del algoritmo es O(m*n), donde 'm' es el número total de movimientos realizados y 'n' es el número de torres.

Análisis Espacial:

- El algoritmo utiliza una cantidad constante de espacio adicional para almacenar variables temporales, como `cambios`, `movimientos` e `i`. Por lo tanto, la complejidad espacial del algoritmo es O(1), es decir, constante.
- La entrada del algoritmo consta de un conjunto de torres, que ocupa espacio en memoria. Sin embargo, este espacio no aumenta con el tamaño de la entrada, por lo que no contribuye a la complejidad espacial del algoritmo.

En resumen, el algoritmo tiene una complejidad temporal de O(m*n) y una complejidad espacial de O(1), donde 'm' es el número total de movimientos realizados y 'n' es el número de torres.

Producto final.

```
import sys
import time

def reorganizar_torres(n,torres):
    cambios = True
    movimientos = 0
    while cambios:
        cambios = False
        for i in range(n - 1):
            if torres[i] < torres[i + 1]:
                torres[i] += 1
                torres[i + 1] -= 1
                cambios = True
                movimientos += 1
    return movimientos


def main():
    if len(sys.argv)<2:
        print("ERROR: Número incorrecto de paramétros para ejecutar el
programa")
        pass
    entrada= sys.argv[1]
    salida = sys.argv[2]
    with open(entrada, 'r') as archive:
        with open(salida,'w') as output:

            m=archive.readline()
            m=int(m)
```

```
            for linea in archive:
                linea=linea.replace('\n','').split(" ")
                n=int(linea[0])
                del linea[0]
                linea=[int(i) for i in linea]
                start=time.time()
                output.write(str(reorganizar_torres(n, linea))+"\n")
                end=time.time()
                print(linea)
                print(end-start)
main()
```

Comprensión de problemas algorítmicos.

Escenario 1:

En un movimiento se puede mover más de una ficha:

1. **Nuevos retos que presupone este nuevo escenario.**
   Calculamos la diferencia entre la altura de la torre actual y la siguiente (diferencia).
   Si la diferencia es mayor que 1, lo que significa que hay espacio para mover más de una ficha, calculamos la cantidad de fichas que podemos mover (cantidad_a_mover).
   En este caso, movemos la mitad de la diferencia, o el número de fichas en la torre actual, lo que sea menor.
   Actualizamos las alturas de las torres en consecuencia y llevamos un registro de los movimientos realizados.
   Continuamos iterando hasta que ya no haya más cambios por hacer.
   Esta modificación permite manejar movimientos de múltiples fichas en cada iteración del algoritmo, adaptándose así al nuevo escenario donde se permite mover más de una ficha en un solo movimiento.
2. **Qué cambios le tengo que realizar a mi solución para que se adapte a este nuevo escenario.**

```
def reorganizar_torres_con_multiples_fichas(n, torres):
    cambios = True
    movimientos = 0
    while cambios:
        cambios = False
        for i in range(n - 1):
            diferencia = torres[i] - torres[i + 1]
            if diferencia > 1:
                # Movemos tantas fichas como sea necesario para
igualar las torres
                cantidad_a_mover = min(diferencia // 2, torres[i])
                torres[i] -= cantidad_a_mover
```

```
                torres[i + 1] += cantidad_a_mover
                cambios = True
                movimientos += cantidad_a_mover
    return movimientos
```

Escenario 2:

1) **Nuevos retos que presupone el nuevo escenario.**
   Este nuevo escenario introduce la posibilidad de mover fichas entre torres que no son necesariamente adyacentes, sino que están dentro de un radio específico 'r'.
   El desafío principal radica en determinar la mejor estrategia para seleccionar las torres entre las cuales se deben mover las fichas, considerando el radio 'r'.
   Se requerirá una lógica más compleja para decidir qué torres se deben considerar en cada paso, lo que podría aumentar la complejidad del algoritmo.

2) **Qué cambios le tengo que realizar a mi solución para que se adapte a este nuevo escenario.**

```
def reorganizar_torres_con_radio(n, torres, r):
    cambios = True
    movimientos = 0
    while cambios:
        cambios = False
        for i in range(n):
            for j in range(max(0, i - r), min(n, i + r + 1)):
                if torres[i] < torres[j]:
                    cantidad_a_mover = min(torres[j] - torres[i],
torres[i])
                    torres[i] += cantidad_a_mover
                    torres[j] -= cantidad_a_mover
                    cambios = True
                    movimientos += cantidad_a_mover
    return movimientos
```

   Iteramos sobre cada torre y comprobamos todas las torres dentro del radio 'r' alrededor de la torre actual.
   Si encontramos una torre más alta dentro del radio, movemos fichas desde la torre más corta hacia la torre más alta, hasta que la altura de ambas torres sea la misma o hasta que la torre actual esté llena.
   Realizamos este proceso para cada torre en cada iteración hasta que ya no haya más cambios por hacer.