Minecraft. Diseño y análisis de algoritmos

Diamis Alfonso Pérez C411, Abraham González Rivero C412

5 de abril de 2023

1. Problema

Dado un arreglo de enteros de tamaño n y el costo de realizar tres operaciones(aumentar, disminuir y mover), se desea hallar el número, (altura), que minimice el costo de igualar el valor de todas las posiciones.

2. Ideas iniciales

1. El número que minimiza el costo está entre el valor mínimo y el valor máximo del arreglo porque cualquier número menor o mayor implica disminuir o aumentar los valores innecesariamente.

2. Si el costo combinado de quitar y poner es menor que el de mover, no tiene sentido realizar ninguna operación de movimiento, pues estarías obteniendo el mismo resultado a un costo aún mayor.

3. Para una altura dada solo tiene sentido mover los valores hacia posiciones que no llegan a la altura, pues se estarían realizado operaciones innecesarias.

3. Primera solución. Fuerza bruta

La idea es por todos los valores posibles entre el mínimo y el máximo del arreglo encontrar el mínimo costo de llevar todos los valores a ese, probando cada movimiento válido, y quedarse con el mínimo.

Pseudocódigo:

```
def MinecraftBruteForce(swap,put,quit,wall):
    for i in range(min(wall),max(wall):
       return min( BestWall(swap, put, quit, wall, i))

def Check(height, wall):
    for i in range(len(wall)):
       return true  if wall[i] == height:
```

```
def BestWall(swap,put,quit,wall,height):
    if Check(height,wall) :
        return 0

    for i in range(len(wall)):
        if wall[i] > height:
            wall[i] = wall[i] - 1
            quitCost = min(quitCost, quit + BestWall(swap, put, quit, wall, height))
            for j in range(len(wall)):
                if  i != j and wall[j] < height :]
                    wall[j] = wall[j] + 1
                    wall[i] = wall[i] - 1
                    moveCost = min(moveCost, swap + BestWall(swap, put, quit, wall, heigh

        if wall[i] < height:
            wall[i] = wall[i] + 1
            putCost = min(putCost, put + BestWall(swap, put, put, wall, height))

    return min(quitCost, moveCost, putCost)
```

4. Segunda solución

Pudimos notar que para una altura h específica, contando la cantidad de bloques que se encuentran a alturas superiores a h y faltantes a alturas inferiores a h, podemos calcular el costo mínimo de igualar el arreglo a ese nivel. La solución se basa en la siguiente idea: si las operaciones de quitar y poner son menos costosas que mover, el mejor costo se reduce a la cantidad de bloques de altura superior a h por el costo de quitarlos más el costo de adicionar los bloques faltantes a alturas menores o iguales que h. En caso contrario, mover es mejor que quitar y poner, la solución se reduce a mover todos los bloques posibles de alturas superiores a h a alturas inferiores, para luego poner o quitar los restantes. Realizando esto para cada altura posible obtenemos la solución óptima.

Esta solución tiene una complejidad temporal $O(nH)$ y constituye una mejora considerable en eficiencia respecto a la anterior.

A continuación se muestra una formalización de nuestra fórmula para calcular el costo de una altura determinada:

$$c(h) = \begin{cases} p * down(h) + q * up(h) & si \quad q + p \leq s \\ s * move(h) + p * (down(h) - move(h)) + q * (up(h) - move(h)) & e.o.c \end{cases} \tag{1}$$

Siendo:

- $down(h) = n * h - \sum_{i=0}^{h} b(h-i)$
- $up(h) = \sum_{i=h+1}^{H} b(i)$
- $move(h) = min(down(h), up(h))$
- H: máximo valor del arreglo.
- h: valor específico para el que estamos buscando el costo.
- q: costo de disminuir en uno un valor.
- p: costo de aumentar en uno un valor.
- s: costo de mover una unidad de un valor hacia otro.
- $b(h)$: cantidad original de bloques en una altura determinada h.

En esta solución recorremos las alturas posibles(los valores entre el mínimo y el máximo del arreglo) encontrando el costo mínimo para cada altura y nos quedamos con el menor.

Pseudocódigo:

```
def MinecraftNH(wall, putC, quitC, swapC):
    for i in range(min(wall), max(wall)):
        return min( Calculate(weights, putC, quitC, swapC, i))
```

5. Función para calcular el costo y consideraciones sobre la búsqueda binaria

Para esta solución nos apoyamos en la fórmula anterior para calcular el costo mínimo:

Pseudocódigo:

```
def BinSearchMinecraft(weigths, swapC, quitC, putC):
    low = 0
    top = max(weigths)

    while top - low > 1 :
        mid = low + top // 2
        midCost = Calculate(weigths, putC, quitC, swapC, mid)
        nextCost = Calculate(weigths, putC, quitC, swapC, mid + 1)
        if midCost >= nextCost:  low = mid
        else: top = mid

    return Calculate(weigths, putC, quitC, swapC, top),top
```

Demostración de la concavidad de $\Delta^2 c(h)$:

Si $p + q \leq s$**:**

El costo mínimo para una altura h es:
$c(h) = p * (n * h - \sum_{i=0}^{h} b(h-i)) + q * (\sum_{i=h+1}^{H} b(i))$

El incremento de la función costo es la variación de los valores entre una altura y la siguiente. Obtenemos:
$\Delta c(h) = p(n - b(h)) - qb(h)$

El incremento de la función incremento es:
$\Delta^2 c(h) = -(p+q) * \Delta b(h)$

Como la cantidad de bloques disminuye o se mantiene igual a medida que aumenta la altura(no vas a tener bloques flotando),entonces $\Delta b(h) \leq 0$

Luego como $\Delta b(h) \leq 0, p \geq 0, q \geq 0 \Rightarrow \Delta^2 c(h) = -p\Delta b(h) - q\Delta b(h) \geq 0$

Entonces el incremento del incremento($\Delta^2 c(h)$) es siempre positivo.

Si $p + q > s$ **:**

$c(h) = s * move(h) + p * (down(h) - move(h)) + q * (up(h) - move(h))$

Si $move(h) = down(h)$:

$c(h) = s * down(h) + q * up(h) - q * down(h)$

$c(h) = s * [n * h - \sum_{i=0}^{h} b(h-i)] + q * [-n * h + \sum_{i=0}^{H} b(i)]$

$\Delta c(h) = s * [n - b(h)] - q * n$

$\Delta^2 c(h) = -s * \Delta b(h)$

Como demostramos anteriormente $\Delta b(h) \leq 0$ entonces $\Delta^2(h) \geq 0$ y por tanto el incremento del incremento de la función costo es positivo también para este caso.

Si $move(h) = up(h)$:

$c(h) = s * up(h) + p * (down(h) - up(h))$

$c(h) = s * \sum_{i=h+1}^{H} b(i) + p * [(n * h - \sum_{i=0}^{h} b(h-i)) - \sum_{i=h+1}^{H} b(i)]$

$\Delta c(h) = -s * b(h) + p * (n - b(h))$

$\Delta c(h) = -s * b(h) + p * n - p * b(h))$

$\Delta^2 c(h) = -(s+p) * \Delta b(h)$

Como $\Delta b(h) \leq 0$ entonces $\Delta^2 c(h) \geq 0$ y por tanto la función es positiva para este caso también.

Como demostramos anteriormente el incremento del incremento de la función costo es siempre positivo. Por tanto la función costo es cóncava.

Por tanto el predicado utilizado en la búsqueda binaria, comparando alturas consecutivas, no permite conocer en todo momento en qué lado del óptimo nos encontramos. Deberíamos realizar una búsqueda ternaria.

6. Solución final

Pseudocódigo:

```
def Blocks(weights, putC, quitC, swapC):
    blocks = [0 for _ in range(max(weights)+1)]
    cant_bloques = 0

    for i in weights:
        cant_bloques += i
        blocks[i]+=1

    for i in range(len(blocks)-2,-1,-1):
        blocks[i]+=blocks[i+1]


    if putC + quitC <= swapC:
        costo = costo_1
    else:
        costo = costo_2

    down = len(weights)-blocks[0]
    up = cant_bloques
    c = costo(putC,quitC,swapC, down, up)
    for i in range(1,len(blocks)):
        up -= blocks[i]
        down += len(weights) - blocks[i]
        c_new = costo(putC,quitC,swapC, down, up)
        print(c_new)
        if c_new > c:
            return c
        else:
            c = c_new
    return c
```

Esta solución está basada en las ideas a las que llegamos intentando realizar una búsqueda binaria. La base es la función de costo encontrada anteriormente.

Lo relevante de esta solución es el cálculo de $b(h)$, la cantidad de bloques hasta una altura determinada, para todo h en una complejidad lineal. Esto lo logramos apoyándonos en la siguiente idea:

Tenemos un arreglo de tamaño h, donde almacenaremos la cantidad de columnas que alcanzan esa altura, esto lo logramos recorriendo el arreglo de los

pesos y recorriendo a la inversa el arreglo de las alturas y actualizando el valor h[i]= h[i] + h[i + 1] pues aquellas columnas que alcancen una altura mayor también alcanzan una altura menor. Este procedimiento se realiza en la siguiente sección del pseudocódigo:

```
blocks = [0 for _ in range(max(weights)+1)]
   cant_bloques = 0

   for i in weights:
       cant_bloques += i
       blocks[i]+=1

   for i in range(len(blocks)-2,-1,-1):
       blocks[i]+=blocks[i+1]
```

Esto se realiza en tiempo lineal: $O(max(h, n))$.

Este cálculo nos permite acceder a la función $b(h)$ en tiempo constante y obtener el costo de una altura en tiempo constante apoyándonos en nuestra expresión de la función de costo. La concavidad de esta nos permite detener la búsqueda si mi altura actual tiene un costo mayor que la anterior.

Luego esta solución tiene complejidad temporal $O(max(h, n))$.