Ejercicio6

Sunday, August 25, 2024 12:32 PM

6. Tenemos un multiconjunto B de valores de billetes y queremos comprar un producto de costo c de una máquina que no da vuelto. Para poder adquirir el producto debemos cubrir su costo usando un subconjunto de nuestros billetes. El objetivo es pagar con el mínimo exceso posible a fin de minimizar nuestra pérdida. Más aún, queremos gastar el menor tiempo posible poniendo billetes en la máquina. Por lo tanto, entre las opciones de mínimo exceso posible, queremos una con la menor cantidad de billetes. Por ejemplo, si $c = 14$ y $B = \{2, 3, 5, 10, 20, 20\}$, la solución es pagar 15, con exceso 1, insertando sólo dos billetes: uno de 10 y otro de 5.

   a) Considerar la siguiente estrategia por *backtracking* para el problema, donde $B = \{b_1, \ldots, b_n\}$. Tenemos dos posibilidades: o agregamos el billete b_n, gastando un billete y quedando por pagar $c - b_n$, o no agregamos el billete b_n, gastando 0 billetes y quedando por pagar c. Escribir una función recursiva $\text{cc}(B, c)$ para resolver el problema, donde $\text{cc}(B, c) = (c', q)$ cuando el mínimo costo mayor o igual a c que es posible pagar con los billetes de B es c' y la cantidad de billetes mínima es q.

$$optipago(B,i,c,p) \begin{cases} (\infty,\infty) & \text{si } i=0 \\ (0,0) & \text{si } j \leq 0 \\ \min\left(optipago(B,i-1,c),\ (optipago(B,i-1,c-B[i-1])+B[i-1],q+1)\right) & \text{si } i>0 \end{cases}$$

b) Implementar la función de *a)* en un lenguaje de programación imperativo utilizando una función recursiva con parámetros B, i, j que compute $\text{cc}(\{b_1, \ldots, b_i\}, j)$. ¿Cuál es la complejidad del algoritmo?

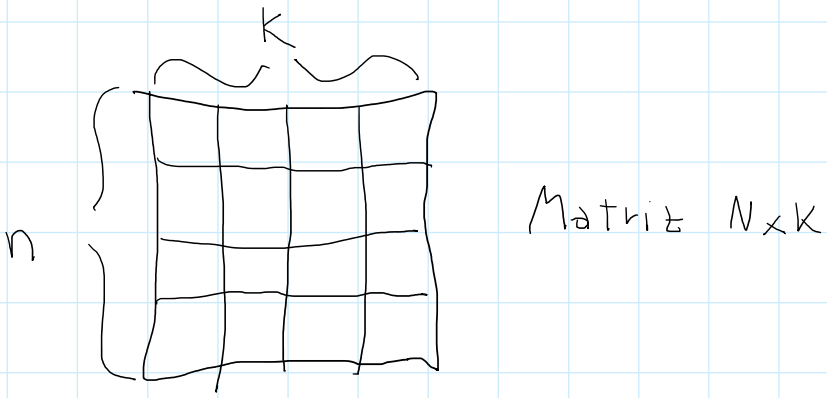
```python
def optipagoBT(B, i, c):
    if i == 0:
        return (float('inf'),float('inf'))
    if c <= 0:
        return (0,0)

    costo_sin_billete, billetes_sin_billete = optipagoBT(B,i-1,c)

    costo_con_billete, billetes_con_billete = optipagoBT(B,i-1,c-B[i-1])
    costo_con_billete += B[i-1]
    billetes_con_billete += 1

    if costo_con_billete < costo_sin_billete or (costo_con_billete == costo_sin_billete and billetes_con_billete < billetes_sin_billete):
        return costo_con_billete, billetes_con_billete
    else:
        return costo_sin_billete, billetes_sin_billete
```

d) Definir una estructura de memoización para cc'_B que permita acceder a $\text{cc'}_B(i, j)$ en $\mathcal{O}(1)$ tiempo para todo $0 \leq i \leq n$ y $0 \leq j \leq k$.



e) Adaptar el algoritmo de *b*) para incluir la estructura de memoización.

```python
def optipagoTP(B, i, c, mem):
    if i == 0:
        return (float('inf'),float('inf'))

    if c <= 0:
        return (0,0)

    if mem[i][c] != None:
        return mem[i][c]
    else:
        costo_sin_billete,billetes_sin_billete = optipagoTP(B,i-1,c,mem)

        costo_con_billete,billetes_con_billete = optipagoTP(B,i-1,c-B[i-1],mem)
        costo_con_billete += B[i-1]
        billetes_con_billete +=1

        if costo_con_billete < costo_sin_billete or (costo_con_billete == costo_sin_billete and billetes_con_billete < billetes_sin_billete):
            mem[i][c] = costo_con_billete, billetes_con_billete
        else:
            mem[i][c] = costo_sin_billete, billetes_sin_billete

    return mem[i][c]
```

f) Indicar cuál es la llamada recursiva que resuelve nuestro problema y cuál es la complejidad del nuevo algoritmo.

optipago(len(B), K)

Temporal O(N.K) . O(1)

Espacial O(N.K)

g) (Opcional) Escribir un algoritmo *bottom-up* para calcular todos los valores de la estructura de memoización y discutir cómo se puede reducir la memoria extra consumida por el algoritmo.

```python
def optipagoBU(B, k):
    mem = [[None for _ in range(k+1)] for _ in range(len(B)+1)]

    for i in range(len(B)+1):
        mem[i][0] = (0,0)

    for j in range(k+1):
        mem[0][j] = (float('inf'),float('inf'))

    for i in range(1, len(B)+1):
        for j in range(1, k+1):
            costo_sin_billete, billetes_sin_billete = mem[i-1][j]

            costo_con_un_billete,billete = float('inf'),float('inf')
            if B[i-1] >= j:
                costo_con_un_billete,billete = B[i-1],1

            c,d = float('inf'),float('inf')
            if j-B[i-1] >= 0:
                e,f = mem[i-1][j-B[i-1]]
                c,d = e+B[i-1],f+1

            costo_usando_billetes,billetes_usando_billetes = min((costo_con_un_billete,billete),(c,d))
            mem[i][j] = min((costo_usando_billetes,billetes_usando_billetes),(costo_sin_billete,billetes_sin_billete))

    return mem[len(B)][k]
```

Para reducir la complejidad espacial se puede usar 2 vectores de tamaño k Uno es la fila actual y el otro es la fila anterior