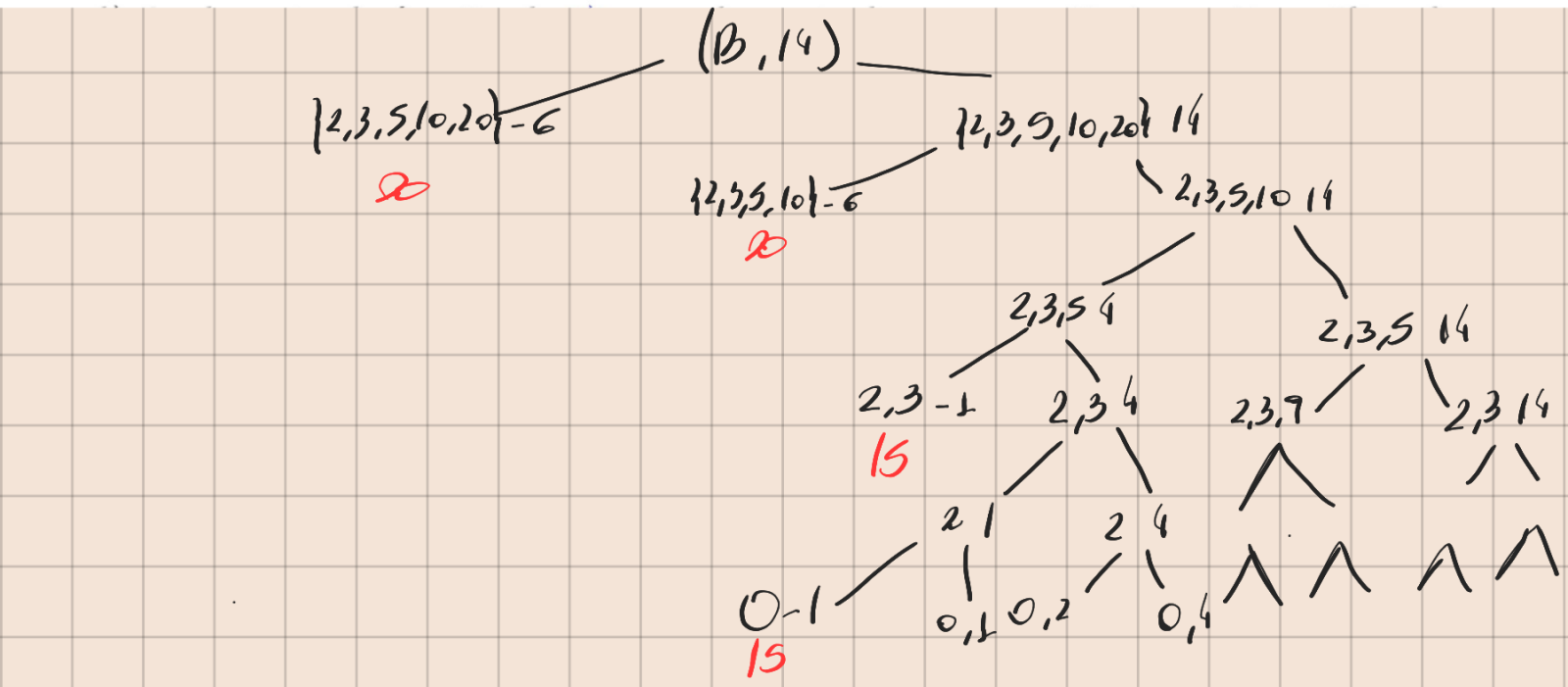
OptiPago

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.



$$CC(B,C) = \begin{cases} (+\infty, +\infty) & \text{si } |B| = 0 \wedge C > 0 \\ (0,0) & \text{si } C \leq 0 \\ \min \begin{cases} CC(\{b_1, \ldots, b_{n-1}\}, C) \\ CC(\{b_1, \ldots, b_{n-1}\}, C - b_n) + \text{primera pos}(b_n) + \text{segunda pos} + 1 \end{cases} & \text{C.C} \end{cases}$$

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?

```
<int,int> CC (B, i, j){
    if i<0 && j>0 then
        return <inf, inf>
    endif
    if j<=0 then
        return <0,0>
    else
        n = cc(B, i-1, j-B[i])
        return min(cc(B, i-1, j), {n.π1 + B[i], n.π2 + 1})
```

Le voy a tener que sumar cosas, lo almacena

c) Reescribir cc como una función recursiva $\text{cc'}_B(i, j) = \text{cc}(\{b_1, \ldots, b_i\}, j)$ que implemente la idea anterior **dejando fijo el parámetro** B. A partir de esta función, determinar cuándo cc'_B tiene la propiedad de *superposición de subproblemas*.

$$cc(B, n, c) = \begin{cases} (+\infty, +\infty) & \text{si } n = 0 \wedge c > 0 \\ (0,0) & \text{si } c \leq 0 \\ \min \begin{cases} cc(B, n-1, c) \\ cc(B, n-1, c-b_n) + \langle \text{primera pos}(B[n]) + \text{segunda pos} + 1 \rangle \end{cases} & \text{C.C} \end{cases}$$

(prefijo de B)

\# Llamados Recursivos $\Omega(2^n)$

\# Subinstancias: $O(nj)$ $\longrightarrow$ $nj << 2^n \longleftrightarrow j << 2^n/n$

Si $j << 2^n/n$ hay Suf de Subp

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 $0 \leq j \leq k$.

Matriz M con $M[i,j] = cc'(i,j)$, tamaño $n \cdot k$

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

```
matriz M de N x K inicializada en ⊥
<int,int> CC(B, i, j){
    if i<0 || j>0 then
        return <inf, inf>
    endif
    if j≤0 then
        return <0,0>
    else
        if M[i][j] = ⊥
            n = CC(B, i-1, j-B[i])
            M[i][j] = min(CC(B, i-1, j), {n.π1 + B[i], n.π2 + 1})
        endif
        return M[i][j]
```