Ejercicio5

Saturday, August 24, 2024 5:09 PM

$a)$ Sea $n = |C|$ la cantidad de elementos de C. Considerar la siguiente función recursiva $\text{ss'}_C\colon \{0,\dots,n\} \times \{0,\dots,k\} \to \{V, F\}$ (donde V indica verdadero y F falso) tal que:

Convencerse de que esta es una definición equivalente de la función ss del inciso $e)$ del Ejercicio 1, observando que $\text{ss}(C,k) = \text{ss'}_C(n,k)$. En otras palabras, convencerse de que el algoritmo del inciso $f)$ es una implementación por *backtracking* de la función ss'_C. Concluir, pues, que $\mathcal{O}(2^n)$ llamadas recursivas de ss'_C son suficientes para resolver el problema.

$$\text{ss'}_C(i,j) = \begin{cases} j = 0 & \text{si } i = 0 \\ \text{ss'}_C(i-1,j) & \text{si } i \neq 0 \wedge C[i] > j \\ \text{ss'}_C(i-1,j) \vee \text{ss'}_C(i-1, j - C[i]) & \text{si no} \end{cases}$$

equivalente

$$\text{ss}(\{c_1,\dots,c_n\},k) = \begin{cases} k = 0 & \text{si } n = 0 \\ \text{ss}(\{c_1,\dots,c_{n-1}\},k) \vee \text{ss}(\{c_1,\dots,c_{n-1}\},k - c_n) & \text{si } n > 0 \end{cases}$$

Implementado con BT

$f)$ Convencerse de que la siguiente es una implementación recursiva de ss en un lenguaje imperativo y de que retorna la solución para C, k cuando se llama con $C, |C|, k$. ¿Cuál es su complejidad?

1) subset_sum(C, i, j): // implementa $\text{ss}(\{c_1,\dots,c_i\}, j)$
2) Si $i = 0$, retornar $(j = 0)$
3) Si no, retornar subset_sum$(C, i-1, j) \vee$ subset_sum$(C, i-1, j - C[i])$

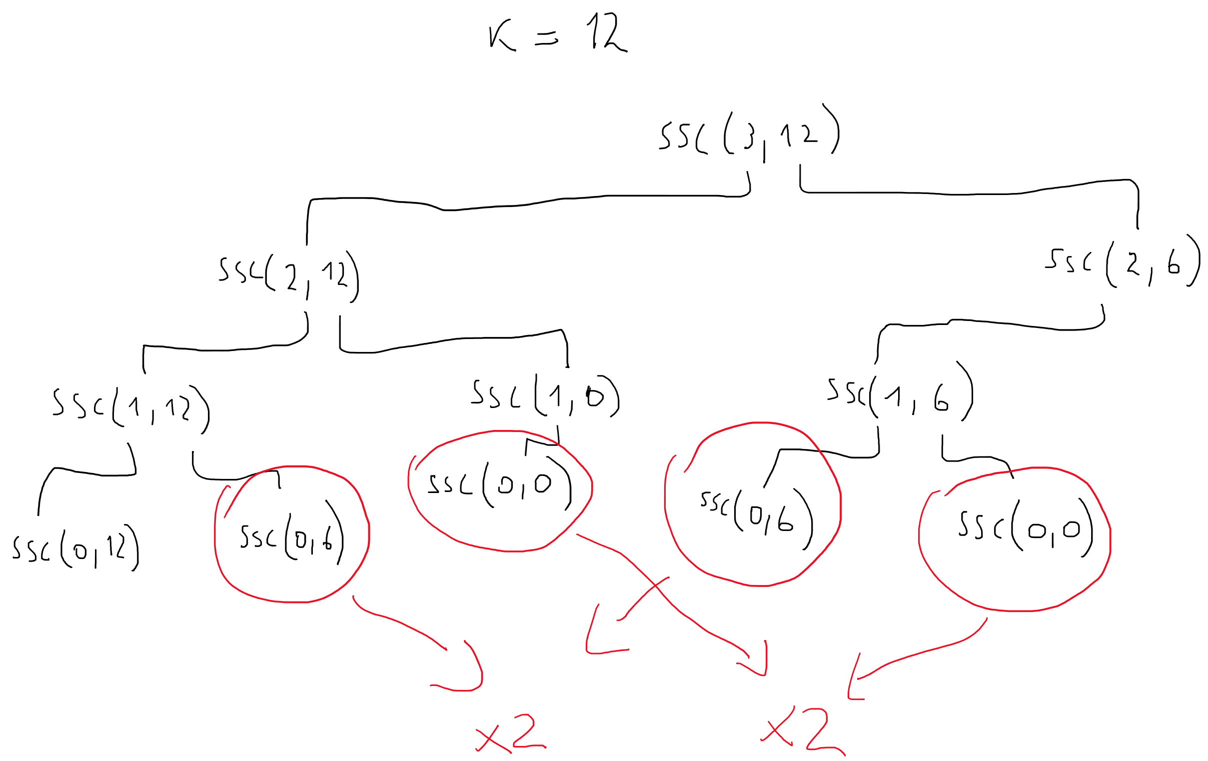
$O(2^n)$ llamadas recursivas

↳ Pongo o No Pongo el elemento

$b)$ Observar que, como C no cambia entre llamadas recursivas, existen $\mathcal{O}(nk)$ posibles entradas para ss'_C. Concluir que, si $k \ll 2^n/n$, entonces necesariamente algunas instancias de ss'_C son calculadas muchas veces por el algoritmo del inciso $f)$. Mostrar un ejemplo donde se calcule varias veces la misma instancia.

$C = [6, 12, 6]$

$n = 3$



c) Considerar la estructura de memoización (i.e., el diccionario) M implementada como una matriz de $(n+1)\times(k+1)$ tal que $M[i,j]$ o bien tiene un valor indefinido $\perp$ o bien tiene el valor $\text{ss'}_C(i,j)$, para todo $0 \leq i \leq n$ y $0 \leq j \leq k$. Convencerse de que el siguiente algoritmo *top-down* mantiene un estado válido para M y computa $M[i,j] = \text{ss'}_C(i,j)$ cuando se invoca $\text{ss'}_C(i,j)$.

1) Inicializar $M[i,j] = \perp$ para todo $0 \leq i \leq n$ y $0 \leq j \leq k$.
2) subset_sum(C,i,j): // implementa $\text{ss}(\{c_1,\ldots,c_i\},j) = \text{ss'}_C(i,j)$ usando memoización
3)   Si $j < 0$, retornar **falso**
4)   Si $i = 0$, retornar $(j = 0)$
5)   Si $M[i,j] = \perp$:
6)   Poner $M[i,j] = \text{subset\_sum}(C,i-1,j) \vee \text{subset\_sum}(C,i-1,j-C[i])$
7)   Retornar $M[i,j]$

```python
def subset_sum(conjunto, i, j, mem):
    if j < 0:
        return False
    if i == 0:
        return j == 0
    if mem[i][j] == None:
        if conjunto[i-1] > j:
            mem[i][j] = subset_sum(conjunto, i-1, j, mem)
        else:
            mem[i][j] = subset_sum(conjunto, i-1, j, mem) or subset_sum(conjunto, i-1, j-conjunto[i-1],mem)
    return mem[i][j]
```

d) Concluir que subset_sum(C,n,k) resuelve el problema. Calcular la complejidad y compararla con el algoritmo subset_sum del inciso *f)* del Ejercicio 1. ¿Cuál algoritmo es mejor cuando $k \ll 2^n$? ¿Y cuándo $k \gg 2^n$?

	PD	BT
Temporal	$O(n.k)$	$O(2^n)$
Espacial	$O(n.k)$	$O(n)$

Cuando el k << 2^/n conviene utilizar la técnica de programación dinámica ya que es generalmente mejor en este caso porque evita explorar subconjuntos irrelevantes y se beneficia de la memoización para reducir el número de subproblemas que necesita resolver.

En cambio cuando k >> 2n/n el problema al utilizár una estrategia de programacíon dinámica es que la estructura de memoización puede llegar a tener un costo espacial muy grande. Entonces convendria utilizar la estrategia de backtracking a que no almacena todas las soluciones parciales.

e) Supongamos que queremos computar todos los valores de M. Una vez computados, por definición, obtenemos que

$$M[i,j] \overset{\text{def}}{=} \text{ss'}_C(i,j) \overset{\text{ss'}}{=} \text{ss'}_C(i-1,j) \vee \text{ss'}_C(i-1,j-C[i]) \overset{\text{def}}{=} M[i-1,j] \vee M[i-1,j-C[i]]$$

cuando $i > 0$, asumiendo que $M[i-1, j-C[i]]$ es falso cuando $j - C[i] < 0$. Por otra parte, $M[0,0]$ es verdadero, mientras que $M[0,j]$ es falso para $j > 0$. A partir de esta observación, concluir que el siguiente algoritmo *bottom-up* computa M correctamente y, por lo tanto, $M[i,j]$ contiene la respuesta al problema de la suma para todo $\{c_1, \dots, c_i\}$ y j.

subset_sum(C, k): // computa $M[i,j]$ para todo $0 \leq i \leq n$ y $0 \leq j \leq k$.
    Inicializar $M[0,j] := (j = 0)$ para todo $0 \leq j \leq k$.
    Para $i = 1, \dots, n$ y para $j = 0, \dots, k$:
        Poner $M[i,j] := M[i-1,j] \vee (j - C[i] \geq 0 \wedge M[i-1, j-C[i]])$

```python
def subset_sum(C, k):
    mem = [[False] * (k + 1) for _ in range(len(C) + 1)]

    for j in range(k+1):
        mem[0][j] = j == 0

    for i in range (1,len(C)+1):
        for j in range (k+1):
            if C[i-1] <= j:
                mem[i][j] = mem[i-1][j] or mem[i-1][j-C[i-1]]
            else:
                mem[i][j] = mem[i-1][j]

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