

Ejercicio 4

El problema planteado se puede resolver de forma sencilla mediante programación dinámica. Para ello necesitaremos crear y mantener una tabla que en este caso contendrá resultados parciales de los que realmente se quieren obtener. La ventaja de hacer esto es que los resultados parciales se pueden calcular de forma sencilla y almacenar en un espacio reducido utilizando recursos mínimos para obtener los resultados deseados a partir de ellos.

Características de la tabla de resultados parciales:

- Los resultados que se almacenan en ella son la cantidad de objetos máximo que podemos tomar para obtener un resultado válido. Sabemos que si la cantidad de objetos máxima para un objeto concreto coincide con la cantidad de objetos máxima sin tener en cuenta dicho objeto es porque el objeto no pertenece al conjunto de los objetos que forman la solución.
- La tabla se crea por necesidad: cuando nos soliciten proporcionar resultados para un tamaño de alforja comprobaremos si nuestra tabla de resultados parciales cubre ya ese valor y en caso de no hacerlo la expandiremos hasta que lo haga.

Cálculo continuo de resultados:

La tabla de resultados parciales es un recurso valioso que cuesta calcular pero que agiliza enormemente el cálculo de múltiples consultas sobre los datos.

Para no dañar la tabla al realizar consultas cuando deseamos que los objetos tomados para una alforja no puedan ir uno a la otra haremos uso de un vector donde indicaremos si un objeto lo podemos tomar o sin dañar ni tener que duplicar la tabla de resultados parciales.

El vector de resultados parciales nos prohíbe por tanto incorporar los elementos marcados como usados al resultado.

Cálculo consecutivo de alforjas:

Parte del problema del cálculo consecutivo de alforjas queda resuelto con el vector de objetos usados. No obstante queda decidir el orden en el que las alforjas serán rellenadas para garantizar que los objetos que puedan ir solo en una de las alforjas acaben en ella y no sean desaparecidos. Debido a que los objetos de mayor peso son evaluados las primeras la primera alforja en llenarse deberá ser la de mayor tamaño de modo que esté se quede con los objetos grandes sin que le quede espacio para tomar los pequeños y no se los quite a la otra alforja.

Función que calcula los objetos para dos alforjas

```
/*
Permite calcular los objetos a introducir en dos alforjas sin perder la matriz de resultados parciales
*/
func obtenerObjetos(para alforja1 : Int, y alforja2 : Int) -> (alforja1:[(peso : Int, valor : Int)],alforja2:[(peso : Int, valor : Int)]){
    let max_alforja : Int
    let min_alforja : Int
    //Averiguamos que alforja es más grande
    if alforja1 > alforja2{
        max_alforja = alforja1
        min_alforja = alforja2
    } else {
        max_alforja = alforja2
        min_alforja = alforja1
    }
    if (max_alforja >= memory.count){//La matriz de resultados parciales se expande si no cubre el tamaño de la alforja máxima
        actualizarMatriz(hasta: max_alforja)
    }
    var used = [Bool].init(repeating: false, count: objetos.count)//Almacena los objetos que ya se tomaron
    //Se pretende evitar que la alforja grande le quete los objetos chicos a la pequeña de modo que pasa primero para llenarse de los
    //objetos que sean de mayor tamaño y que no pueda tomar los chicos, todos los que no queden en ella irán a la pequeña
    let objetos_max_alforja = obtenerObjetos(para: max_alforja, con: memory, y: objetos, en: objetos.count - 1, regarding: &used)
    let objetos_min_alforja = obtenerObjetos(para: min_alforja, con: memory, y: objetos, en: objetos.count - 1, regarding: &used)
    if max_alforja == alforja1{
        return (objetos_max_alforja,objetos_min_alforja)
    }else {
        return (objetos_min_alforja,objetos_max_alforja)
    }
}
```

Marcas si el objeto ya ha sido usado.

Índice o punto al que evaluar los objetos.

Lista de objetos

Matriz de resultados parciales

Tamaño de la alforja

Función que obtiene los objetos de una alforja

```
/*
Obtene un array con los objetos que forman parte del resultado óptimo
La variable inout (regarding used) indica a false los objetos disponibles para usar
*/
func obtenerObjetos (para alforja : Int, con mem : [[Int]], y obj : [(peso : Int, valor : Int)], en pos : Int, regarding used : inout [Bool]) -> [(peso : Int, valor : Int)] {
    if (pos == 0){//Estamos en el objeto más pequeño
        if (alforja >= obj[pos].peso) && !used[pos]{//Si cabe y no lo hemos usado es resultado
            used [pos] = true;
            return [obj[pos]];
        }
        return [(peso : Int, valor : Int)()];
    //Tratemos los objetos ya usados
    } else if (used[pos] || ((mem [alforja][pos] == mem [alforja][pos-1]) && !used[pos-1])){
        return obtenerObjetos(para: alforja, con: mem, y: obj, en: pos - 1, regarding : &used);
    } else{
        if alforja >= obj[pos].peso{//Es el objeto más grande de los que podemos introducir, si cabe es resultado
            var lista_objetos = [obj[pos]];
            used [pos] = true;
            //Añadimos el nuevo resultado a los resultados obtenidos recursivamente
            lista_objetos.append(contentsOf : obtenerObjetos(para: alforja - obj[pos].peso, con: mem, y: obj, en: pos - 1, regarding : &used));
            return lista_objetos;
        }
        return obtenerObjetos(para: alforja, con: mem, y: obj, en: pos - 1, regarding : &used);
    }
}
```

Función que calcula los resultados parciales

```
//Matriz de resultados parciales
var memory = [[Int]]()

/*
Expande la matriz de resultados parciales calculando la cantidad máxima de objetos que forman la solución
óptima hasta el tamaño de alforja indicado.
En la tabla se tienen en cuenta los objetos disponibles en cada momento
*/
func actualizarMatriz(hasta final_size: Int) {
    if memory.isEmpty {
        memory.append([Int].init(repeating: 0, count: objetos.count))
    }
    for _ in memory.count...final_size{
        var memory_fragment = [Int]()
        for index in 0 ..< objetos.count{
            if index == 0{//CASO BASE, objeto de menor tamaño
                if objetos [0].peso > memory.count { //el objeto de menor tamaño es más grande que la alforja
                    memory_fragment.append(0)
                } else {
                    memory_fragment.append(objetos[0].valor)
                }
            } else {
                if memory.count-objetos[index].peso < 0 || objetos [index].peso > memory.count{
                    //No se puede tomar el objeto nuevo
                    memory_fragment.append(memory_fragment[index-1])
                } else {
                    //Se toma el objeto nuevo si tomarlo es mejor resultado que no hacerlo
                    memory_fragment.append(max(memory_fragment[index-1],
                        memory[memory.count-objetos[index].peso][index-1]+objetos[index].valor))
                }
            }
        }
        memory.append(memory_fragment)
    }
}
```

Ejercicio 7

Para resolver el problema de encontrar el conjunto común ordenado más grande presente en dos vectores proporcionados se ha creado un algoritmo recursivo tradicional. Para hacer más eficiente dicho algoritmo se emplea una matriz de resultados parciales con tamaño tal que pueda almacenar resultados para todas las posiciones de los vectores de entrada.

Sobre el algoritmo recursivo básico se añade una nueva comprobación para que en el caso de que el valor que se vaya a calcular se haya calculado con anterioridad dicho cálculo no se repita si no que se reutilice el ya calculado con anterioridad.

Algoritmo recursivo:

- CASO BASE: si hemos llegado al final de ambos vectores cortamos la recursión.
- Si el resultado ya fue calculado lo devolvemos
- Si el elemento a evaluar en ambos vectores coincide es porque este forma parte de la secuencia común que se está generando.
- Si el elemento a evaluar en ambos vectores difiere la subsecuencia común ordenada de tamaño máximo será la más grande formada incluyendo al elemento de uno de los vectores pero no al del otro o viceversa.

```

//Almacena los resultados ya calculados
var memory = [[[Int?]].init(repeating: [[Int]?].init(repeating: nil, count: secuence2.count), count: secuence1.count)

/*
Función recursiva que toma dos vectores y proporciona de ellos la parte común a ambos que tenga un tamaño mayor.
Hace uso de una memoria de resultados parciales para evitar calcular varias veces los mismo resultados
*/
func largestCommonSecuence (of: secuence1 : [Int], and: secuence2 : [Int], index1 : Int = 0, index2 : Int = 0) -> [Int] {
    if (index1 == secuence1.count) || (index2 == secuence2.count) { //Hemos terminado (hemos recorrido ambos vectores)
        return [Int]()
    } else if memory[index1][index2] != nil { //El resultado ya fue calculado anteriormente
        return memory[index1][index2]!
    } else if secuence1[index1] == secuence2[index2]{ //Como el valor de los vectores coincide este valor formará parte de la cadena común
        memory[index1][index2] = [secuence1[index1]]
        + largestCommonSecuence(of: secuence1, and: secuence2, index1: index1+1, index2: index2 + 1)
        return memory[index1][index2]!
    } else { //El valor a evaluar de los vectores difiere
        //La cadena común será la más larga entre la formada incluyendo al valor del vector1 pero no al del dos o viceversa
        let l1 = largestCommonSecuence(of: secuence1, and: secuence2, index1: index1 + 1, index2: index2)
        let l2 = largestCommonSecuence(of: secuence1, and: secuence2, index1: index1, index2: index2 + 1)
        if l1.count >= l2.count {
            memory[index1][index2] = l1
        } else {
            memory[index1][index2] = l2
        }
        return memory[index1][index2]!
    }
}

//Almacenamos los resultados parciales

```