Análisis de Algoritmos 2022-1
Tarea 6

Alumnos:
Hernández Sánchez Oscar José
Altamirano Niño Luis Enrique

17 de junio de 2022

Ejercicios

a) Propón un algoritmo recursivo (de complejidad exponencial) que resuelva el problema, en su versión de cálculo del valor óptimo, pero no de la estructura óptima. Analiza su corrección y su complejidad.

Respuesta:

Algoritmo 1 Algoritmo que calcula el valor óptimo para una subinstancia

```
#Los i-1 en W[i-1] son porque los índices del arreglo comienzan en 0.
def opt(W,i):
    if i == 0:
        return 0
    elif i == 1:
        return W[i-1]
    else:
        return max(opt(W,i-1),opt(W,i-2)+W[i-1])
```

Algoritmo 2 Algoritmo que calcula el valor del conjunto independiente de peso máximo

```
    def ind(W):
        return opt(W,len(W))
```

Corrección:

Ahora demostremos que el algoritmo es correcto.

Primero tomemos en cuenta que como los índices del arreglo comienzan en 0 y la ecuación de Bellman supone que comienzan en 1, entonces los $W[i]$ en la ecuación de Bellman son equivalentes a $W[i-1]$ en el algoritmo, por ejemplo, $W[1]$ en la ecuación de Bellman hace una referencia al primer elemento del arreglo, sin embargo, en el algoritmo, el primer elemento del arreglo es $W[0]$.

Demostración.

Por inducción en el tamaño del arreglo.

Caso base: Se tienen dos casos base:

- Si el tamaño del arreglo es 0, entonces el algoritmo devuelve `opt(W,len(W))=opt([],0)`, y entonces el algoritmo `opt` se encuentra un caso base y regresa 0, que es justamente el resultado correcto según la ecuación de Bellman.
- Si el tamaño del arreglo es 1, entonces el algoritmo devuelve `opt(W,len(W))=opt(W,1)`, y entonces el algoritmo `opt` se encuentra un caso base y regresa `W[i-1]`, que es justamente el resultado correcto según la ecuación de Bellman.

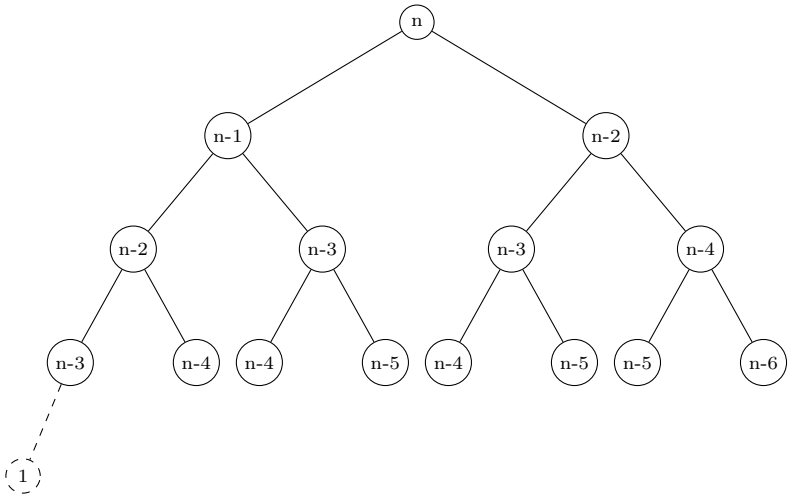
Hipótesis de inducción: Supongamos que el algoritmo regresa el resultado correcto para un arreglo de tamaño k.

Paso inductivo: Ahora debemos mostrar que el algoritmo regresa el resultado correcto para un arreglo de tamaño $k+1$, entonces el algoritmo reresará `opt(W,len(W))=opt(W,k+1)` y `opt` devolverá
`max(opt(W,(k+1)-1),opt(W,(k+1)-2)+W[i-1])=max(opt(W,k),opt(W,k-1)+W[i-1])`
y por la hipótesis de inducción las llamadas recursivas `opt(W,k)` y `opt(W,k-1)` devolverán el resultado correcto, además según la ecuación de Bellman `max(opt(W,(k+1)-1),opt(W,(k+1)-2)+W[i-1])` nos devuelve el peso máximo de algún conjunto independiente en la trayectoria, que es justo el resultado esperado.

Por lo tanto el algoritmo es correcto. ■

Complejidad:

Si el arreglo es de tamaño n, entonces el árbol de recursión tomará en general la siguiente forma:



entonces como el algoritmo tiene como caso base a 1, la altura del árbol será n, además en cada nivel del árbol habrá a lo más 2^l nodos, donde l es el nivel del árbol, entonces en el peor caso se tendrán

$$\sum_{j=0}^{n} 2^j = 2^{n+1} - 1$$

llamadas recursivas, y entonces la complejidad del algoritmo será $O(2^n)$.

b) Propón la versión recursiva con memorización del ejercicio anterior. Asegúrate que devuelve el mismo valor que el algoritmo anterior, y por eso se transfiere su correción, y argumenta su complejidad contando el número máximo de subinstancias distintas en el árbol de recursión.

Respuesta:

Algoritmo 3 Algoritmo que calcula el valor óptimo para una subinstancia usando memorización.

```
#Los i-1 en W[i-1] son porque los índices del arreglo comienzan en 0.
def optMem(W,i,mem):
    if mem[i]!=-1:
        return mem[i]
    else:
        sol = max(optMem(W,i-1,mem),optMem(W,i-2,mem)+W[i-1])
        mem[i] = sol
        return sol
```

Algoritmo 4 Algoritmo que calcula el valor del conjunto independiente de peso máximo

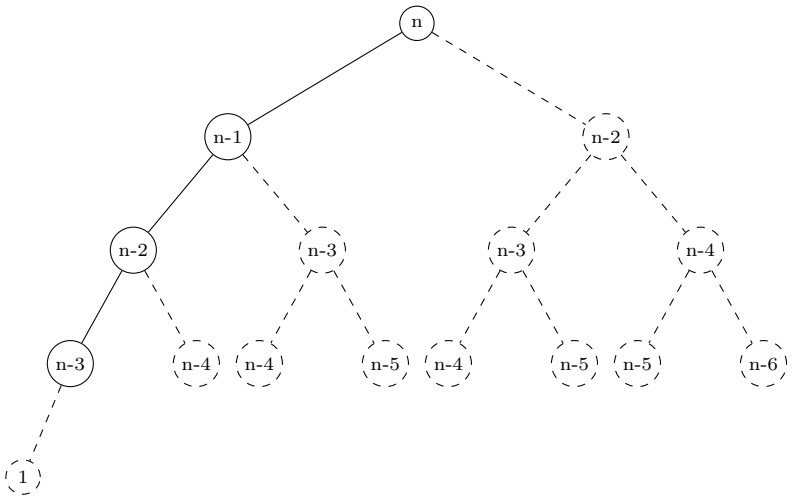
```
    def indMem(W):
        if len(W)== 0:
            return 0
        #Creamos el arreglo donde iremos almacenando los valores ya calculados
        mem = [-1]*(len(W)+1)
        #Llenamos los casos base
        mem[0] = 0
        mem[1] = W[0]
        return optMem(W,len(W),mem)
```

Corrección:

La corrección se transfiere por lo mostrado en el inciso anterior, pues se tiene exactamente el mismo algoritmo con la excepción de que ahora se guardan los valores ya calculados en un arreglo para no calcularlos más de una vez.

Complejidad:

Precisamente por el hecho de que ya no estamos calculando mas de una vez el valor de una subinstancia, y suponiendo que siempre se terminan de evaluar primero los nodos a la izquierda, entonces el árbol de recursión se "podará", y quedará de la siguiente forma:



y por lo tanto solo tendremos una trayectoria del árbol original que tiene profundidad n, por lo que esa trayectoria tendrá longitud n, y entonces ahora solo tendremos n subinstancias distintas en el árbol de recursión y entonces la complejidad del algoritmo será $0(n)$.

c) Propón la versión iterativa de programación dinámica del algoritmo anterior (declara las matrices M_{opt} y M_{choices}, establece la semántica de cada una de sus celdas, y propón un orden de llenado correcto, describe la función de llenado). Asegúrate que devuelve el mismo valor que el algoritmo anterior, y por eso se transfiere su corrección, y demuestra que su complejidad es $O(n)$.

Respuesta:

Algoritmo 5 Versión iterativa del algoritmo

```
def optM(W):
    Mopt = [-1]*(len(W)+1)
    Mopt[0] = 0
    Mopt[1] = W[0]
    global Mchoice
    Mchoice = [-1]*(len(W)+1)
    Mchoice[1] = 0
    for i in range (2,len(W)+1):
        option_a = Mopt[i-1]
        option_b = Mopt[i-2]+W[i-1]
        if option_a < option_b:
            Mopt[i] = option_b
            Mchoice[i] = 0
        else:
            Mopt[i] = option_a
            Mchoice[i] = 1
    return Mopt[len(W)]
```

La matriz M_{opt} será un arreglo de tamaño $n+1$ que en un inicio tendrá -1 en cada entrada, en cada celda i contendrá el valor del conjunto independiente de peso máximo para la trayectoria de los primeros i vértices de W, entonces comenzaremos llenando las celdas 0 y 1, con los valores 0 y $W[0]$ respectivamente, que eran los casos base en el algoritmo recursivo, después iremos llenando el arreglo de izquierda a derecha pues para calcular el valor de una entrada, necesitamos saber el valor de las dos anteriores. Esto es lo mismo que el arreglo de memorización del algoritmo anterior, pero llenado de manera iterativa.

La matriz M_{choices} será un arreglo de tamaño $n+1$ que en un inicio tendrá -1 en cada entrada, e irá almacenando en cada celda i el valor 0 o 1, 0 si $M_{\text{opt}}[i-1] < M_{\text{opt}}[i-2] + W[i-1]$, o 1 en el caso contrario, todo esto con el fin de saber en cada subtrayectoria de longitud i, cual que opción nos llevó a encontrar el valor óptimo y posteriormente poder construir la solución.

c) Propón el algoritmo que, a partir de la tabla M_{choice} generada por el algoritmo anterior, calcula el conjunto independiente de peso máximo.

Respuesta:

Se tiene el siguiente algoritmo:

Algoritmo 6

```
    def construyeSolucion(Mchoice):
        sol = []
        i = len(Mchoice)-1
        while i>= 1:
            #No incluimos el indice actual y pasamos al anterior
            if Mchoice[i] == 1:
                i -=1
            #Sí incluimos el indice actual y retrocedemos 2.
            else:
                sol = [i] + sol
                i-=2
        return sol
```