Universidad del Valle de Guatemala
Análisis y Diseño de Algoritmos
Docentes: Paulo Mejía; Tomás Gálvez P.
Semestre 1, 2023

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Programación dinámica

El **paradigma de programación dinámica** se deriva del de *Divide and Conquer* que vimos anteriormente[1]. La programación dinámica es comúnmente empleada en **problemas de optimización,** en los que hay muchas posibles soluciones y buscamos la que tenga el máximo o mínimo valor según nuestras necesidades. Aquellos problemas de optimización a los cuales podemos aplicar programación dinámica deben tener dos características: subestructura óptima y subproblemas traslapados.

El concepto de **subestructura óptima** es intuitivo, pero algo enredado de expresar. Primero debemos comprender que, como se basa en DaC, la programación dinámica resuelve los problemas de manera recursiva. De modo que cuando la solución óptima a un problema de éstos se compone, a su vez, de las soluciones óptimas a sus correspondientes subproblemas, este problema exhibe una subestructura óptima. *Enter* el ***rod-cutting problem***.

Este problema nos presenta a una empresa que vende tubos de metal. La empresa considera un tubo de longitud n, y el precio de un trozo del tubo con longitud $i = 1,2,3, \dots, n$ será p_i.

1. Si queremos cortar el tubo en pedazos vendibles, ¿qué obtenemos al cortar el tubo de tamaño n en la longitud $0 \leq i < n$? ¿De cuántas formas podemos hacer el corte?

Obtenemos dos tubos, uno de tamaño i y otro de tamaño $n - i$. Podemos elegir cortar el tubo en n posiciones diferentes porque $0 \leq i < n$.

¿Han visto cuando venden las piezas de un *set* más caras por pieza que como parte del *set*? Considérese la siguiente tabla de precios para tamaños de tubo:

Longitud i	1	2	3	4	5	6	7	8	9	10
Precio p_i	1	5	8	9	10	17	17	20	24	30

Esta tabla muestra que un tubo de tamaño 4 puede producir una mejor ganancia vendido como dos subtubos de tamaño 2 que vendido entero. La empresa sabe que puede cortar el tubo de tamaño n de forma que los dos tubos resultantes produzcan la máxima ganancia posible. Naturalmente, los "subtubos" pueden seguir siendo cortados recursivamente hasta llegar a un tamaño de tubo que ya no se puede cortar (porque es la unidad de venta más pequeña, digamos).

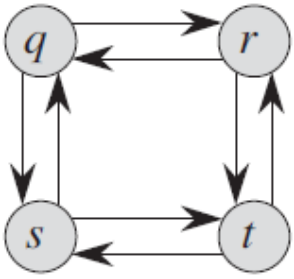
Imaginemos que tenemos dos tubos de tamaño n y que los cortamos en dos posiciones diferentes, respectivamente i_1 e i_2. Esto resulta en cuatro subtubos de tamaños i_1 y $(n - i_1)$ e i_2 y $(n - i_2)$. Ahora imaginemos que logramos maximizar la ganancia obtenida a partir de cada pareja de subtubos, cortando y vendiendo los subtubos de cada pareja, pero resulta que la ganancia de la pareja 1 es g_1 y la de la pareja 2 es g_2, con $g_1 > g_2$. Debería notarse que podríamos haber mejorado nuestra ganancia total si hubiéramos cortado ambos tubos en la posición i_1, porque $2g_1 > g_1 + g_2$.

1: un **paradigma**, en diseño de algoritmos, es una clasificación definida por la forma general que comparten los algoritmos de ese paradigma.

Con un razonamiento similar podemos concluir que habrá alguna posición i_p, con $0 \leq i_p < n$, que produzca una pareja de subtubos cuyas ganancias maximizadas y combinadas sean más que las de cualquier otra pareja obtenida al cortar el mismo tubo en otra posición $i_t \neq i_p$. Esta misma idea se aplica al buscar maximizar la ganancia de cada subtubo. Por otra parte, cortar los subtubos en posiciones no óptimas resultará en una ganancia menor a la que podríamos haber obtenido, aunque el tubo original haya sido cortado en i_p.

Podemos apreciar la naturaleza aditiva de la maximización de la ganancia. Debemos cortar el tubo original en la posición correcta para obtener la mejor ganancia posible. Pero esto requiere a su vez que los "subsubtubos" también sean cortados de manera óptima, y así recursivamente. En otras palabras, la solución óptima para un problema depende de la solución óptima para los subproblemas en los que se divide. Esto demuestra que el *rod-cutting problem* exhibe la propiedad de subestrucuta óptima.

Para contrastar, veamos el **problema del camino simple más largo en un grafo sin costos** (***unweighted longest simple path problem***). Este problema nos requiere encontrar, entre dos nodos de un grafo sin costos, el camino que posea la mayor cantidad de aristas posible. Considere el siguiente grafo y el camino entre los nodos q y t:



2. Explique por qué este problema no tiene subestructura óptima, basándose en el grafo de ejemplo.

En este grafo, el camino más largo entre los nodos q y t puede ser tanto $q \to s \to t$ como $q \to r \to t$. Sin importar la opción, partir de q siempre nos lleva a r o a s, entonces debemos resolver el subproblema $q \to s/r$. El camino más largo entre q y s es $q \to r \to t \to s$. Al haber pasado por t no podemos regresar a él sin formar un ciclo y, por tanto, no podemos resolver el problema original. Yendo más lejos, una vez que alcanzamos s necesitaríamos resolver el subproblema $s \to t$ para concluir, pero el camino más largo entre s y t es $s \to q \to r \to t$. Resolver el subproblema $s \to t$ nos obliga a pasar por q, produciendo un ciclo en contra de las restricciones del problema.

Este problema no posee una subestructura óptima porque los subproblemas que se obtienen de él no son **independientes**. La independencia entre subproblemas se refiere a que computar la solución de un subproblema no depende de los recursos de, o no se ve afectada (según las condiciones del problema) por, la solución para otro subproblema (en otras palabras, la solución a un subproblema no afecta la solución para un subproblema "hermano").

En el ejemplo, elegir las soluciones "óptimas" a los subproblemas considerados provee una solución inválida al problema inicial, ya que el camino deja de ser simple. Como la solución óptima a este problema no se puede construir a partir de soluciones óptimas a subproblemas, este es un problema sin subestructura óptima.

Dos características gobiernan la subestructura óptima de un problema: **número de decisiones**, y **número de subproblemas**. Para identificar la subestructura óptima de un problema normalmente seguimos los siguientes pasos:

1. Identificar la(s) decisión(es): en los problemas de DaC debemos tomar decisiones que producen los subproblemas a resolver, *e.g.*, dónde dividir una lista de números. Esas son las decisiones que debemos identificar.
2. Identificar los subproblemas: suponiendo que tomamos la o las decisiones que nos llevan a una solución óptima, identificamos los subproblemas que éstas producen y las características de este conjunto de subproblemas resultante.
3. Demostrar la necesaria optimalidad: por último, observamos que ya estamos en camino a solucionar óptimamente el problema original. Demostramos en este paso que, para solucionar el problema óptimamente, las soluciones a los subproblemas deben ser las óptimas. Para ello suponemos que tenemos la solución óptima compuesta de soluciones subóptimas a los subproblemas. La idea es revelar que si hubiéramos obtenido las soluciones óptimas habríamos producido una mejor solución general, contradiciendo que podemos producir una solución óptima usando soluciones subóptimas a los subproblemas.

3. Identifique las decisiones y los subproblemas en el *rod-cutting problem*.

Decisión: para cortar un tubo en piezas que maximicen la ganancia empezamos por hacer un corte al tubo original que produce dos tubos más pequeños. La decisión es, entonces, dónde vamos a cortar el tubo.

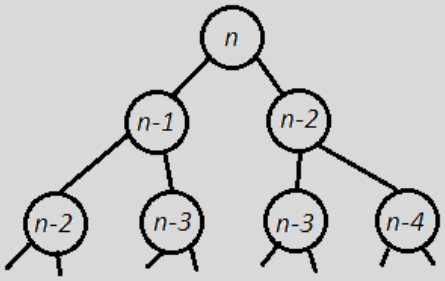
Subproblemas: si tenemos un tubo de longitud n y lo cortamos en i (para $0 \leq i \leq n$) producimos dos "subtubos". Suponemos que el corte en i lleva a la solución óptima y notamos que los subtubos que se producen tienen longitud i y $n - i$.

4. Demuestre que el *rod-cutting problem* exhibe subestructura óptima, completando la respuesta anterior con la demostración de necesaria optimalidad.

Supongamos que los subtubos producidos no son cortados óptimamente y que, luego de cortarlos, obtenemos de ellos las ganancias a y b. De acuerdo con nuestra suposición en el inciso anterior, el corte inicial en i produce la ganancia máxima, que en este caso sería $a + b$. Sin embargo, como no cortamos los subtubos óptimamente, deben existir las ganancias $x > a$ e $y > b$ que obtendríamos si cortamos los subtubos óptimamente. Con esos cortes la ganancia total sería $x + y$, que es mayor que $a + b$. Esto contradice la suposición de que nuestra solución con ganancias $a + b$ es la óptima, a pesar de que iniciamos con el corte i. Entonces, si nuestro corte inicial produce la solución óptima, es necesario que los subtubos producidos sean cortados también óptimamente.

La otra característica esencial de un problema que se desee resolver con programación dinámica es que tenga **subproblemas traslapados**. Esta es en realidad la característica que define a un problema como solucionable con programación dinámica, ya que la subestructura óptima es también característica del paradigma de diseño *Greedy*.

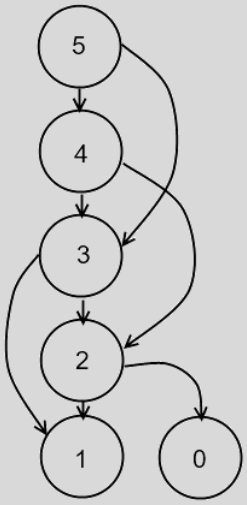
5. Esboce el árbol de recursión para el algoritmo recursivo $F(n) = F(n-1) + F(n-2)$ que produce el n-ésimo elemento en la sucesión de Fibonacci. ¿Qué ineficiencia encuentra?



Los subproblemas $n-2$ y $n-3$ aparecen más de una vez, y en niveles más profundos se repetirán estos y otros subproblemas.

6. Convierta el árbol de recursión de la pregunta anterior en un grafo de subproblemas, donde se elimine la ineficiencia planteada. Para que se plantee un grafo completo, considere el caso $n = 5$.

Tomemos el quinto elemento en la secuencia de Fibonacci. El grafo de subproblemas se vería así:



Nota: en la solución provista a la pregunta 6 no se dibujan las dependencias indirectas entre nodos, implícitas por las dependencias de las dependencias directas.

A eso se refiere la característica de subproblemas traslapados: a subproblemas que son engendrados y resueltos redundantemente. Programación dinámica refina el paradigma DaC atacando la ineficiencia de subproblemas traslapados con una entre dos maneras.

7. ¿Cuáles son las dos maneras en las que se puede resolver la ineficiencia de solucionar subproblemas traslapados? Considere las dependencias.

1. **Método de arriba a abajo con memoización (*top-down with memoization*)**: se **memoiza** el método DaC agregando una tabla de soluciones que es consultada cada vez que vamos a resolver un subproblema. Si no hallamos la solución a dicho subproblema en la tabla, se computa esa solución y se almacena en la tabla. De lo contrario se usa el resultado almacenado.
2. **Método de abajo a arriba (*bottom-up*)**: este método requiere que identifiquemos todos los subproblemas y los ordenemos por dependencia. Luego comenzamos por solucionar los casos triviales, y continuamos con los subproblemas que dependen de ellos; y después los subproblemas que dependen de éstos, y así hasta llegar al problema original.

Nota: en varios textos se emplea el término "programación dinámica" para referirse exclusivamente a método *bottom-up*, mientras que al *top-down* se le refiere únicamente como "memoización".

Para algunos problemas en los que aplica la programación dinámica será necesario o conveniente resolver absolutamente todos los subproblemas al menos una vez. En estos casos, el método *bottom-up* es más eficiente porque no es recursivo y tiene mejor manejo de la tabla de subproblemas (no debe estar buscando si ya se resolvió un problema o no, y generalmente ya sabe dónde se ubican las soluciones que necesita para cada subproblema).

En los casos en los que no se necesita o no es conveniente resolver todos los subproblemas es mejor el método *top-down* con memoización, claramente porque resuelve sólo aquello que se va a necesitar. Además, siempre es más intuitivo implementar el método *top-down* porque es simplemente DaC con memoización. Las diferencias en desempeño, sin embargo, serán casi siempre factores constantes entre ambos métodos. Asintóticamente, tendrán la misma tasa de crecimiento, *i.e.*, resultarán normalmente igual de eficientes.

Considerando las dos características esenciales de la programación dinámica observamos, entonces, que el árbol de recursión que usamos con el paradigma DaC se convierte en un **grafo de subproblemas** al fusionar los subproblemas que aparecen más de una vez. Este grafo es útil para visualizar el número de subproblemas y las dependencias entre ellos; y para construir un algoritmo con cualquiera de los acercamientos. En el caso de *top-down* con memoización se recorre el grafo con una *depth-first search*, mientras que para *bottom-up* debemos haber resuelto los subproblemas en los hijos del nodo x antes de resolver el subproblema en el nodo x.

En general, para construir un algoritmo usando programación dinámica debemos seguir cuatro pasos:

1. Caracterizar la solución óptima.
2. Describir el valor de la solución óptima de forma recursiva.
3. Computar el valor de la solución óptima.
4. Construir la solución óptima.

El primer paso identifica la subestructura óptima en el problema y, con base en ella, determina cómo hallar la solución óptima de manera recursiva. Recordemos que, en los pasos citados anteriormente para identificar la subestructura óptima, suponemos que nos es(son) dada(s) la(s) decisión(es) que hay que tomar sobre el problema para alcanzar la solución óptima. Este paso remueve esa suposición y determina cómo se va a buscar la decisión correcta que nos llevará a la solución óptima. Es como armar el esqueleto del algoritmo.

El segundo paso es construir la relación de recurrencia que calcula el valor de la solución óptima basándose en el paso anterior. Esta relación especifica la manera en la que las soluciones a subproblemas se combinan, formando la solución para su problema "padre". El paso anterior asegura que esta relación de recurrencia expresa el valor de la solución óptima cuando la solución siendo calculada es la del problema raíz.

El tercer paso conlleva escribir el algoritmo que, basado en la ecuación de recurrencia del segundo paso, permita calcular el valor de la solución óptima (ya el mero valor, no sólo una definición). En este paso es donde cobra importancia la característica de los problemas traslapados, pues el algoritmo debe reconocer y evitar la re-computación de soluciones a subproblemas recurrentes. En este paso es también donde se debe elegir el método de solución entre *top-down* y *bottom-up*, siendo *top-down* normalmente más fácil de programar, pero en ocasiones menos eficiente.

El cuarto paso es frecuentemente realizado, aunque no siempre necesario. Este paso se encarga de describir la solución óptima, *i.e.*, muestra las soluciones a los subproblemas resueltos para hallar el valor de la solución óptima general. Para lograrlo normalmente se agregan, al algoritmo del paso tres, instrucciones que almacenen información sobre los pasos en la computación del valor óptimo.

8. En (Soltys, 2012) se listan tres pasos para desarrollar una solución con programación dinámica: definir una clase de subproblemas, proveer una recurrencia que resuelva los problemas en términos de subproblemas; y proveer un algoritmo que compute la recurrencia. Identifique estos pasos con los descritos anteriormente. Explique su razonamiento.

El segundo paso es compartido entre autores. El tercer paso de Soltys comprende desarrollar el algoritmo que compute la recurrencia, que podría identificarse con el tercer (y tal vez el cuarto) paso de Cormen. Respecto al primer paso, caracterizar la estructura de la solución óptima conlleva identificar subproblemas. Definir una clase de subproblemas es identificar un conjunto por características, y la clase que nos interesa es la de subproblemas que se producen con la decisión óptima. Para hallarla probamos diferentes decisiones, produciendo una clase de subproblemas con cada una. La clase que provea los mejores resultados será la que cada problema padre resolverá para hallar su propia solución. En esencia, los pasos de ambos autores se refieren a identificar los componentes de la recurrencia del paso dos.

Retomaremos el ejemplo del *rod-cutting problem*, esta vez empleando formalmente los conceptos de programación dinámica que hemos descrito hasta este momento.

9. Caracterice la estructura de la solución óptima para el *rod-cutting problem*.

Ahora que ya sabemos que el problema exhibe subestructura óptima, la solución óptima debe buscarse realizando un corte hipotético sobre cada posible i (para $0 \leq i \leq n$) y obteniendo las soluciones óptimas de forma recursiva sobre los subtubos de longitud i y $n - i$. El corte i que produzca la mejor ganancia será el que pertenezca a la solución óptima. Nótese cómo está en nuestro poder la forma en que se buscan y prueban las diferentes decisiones.

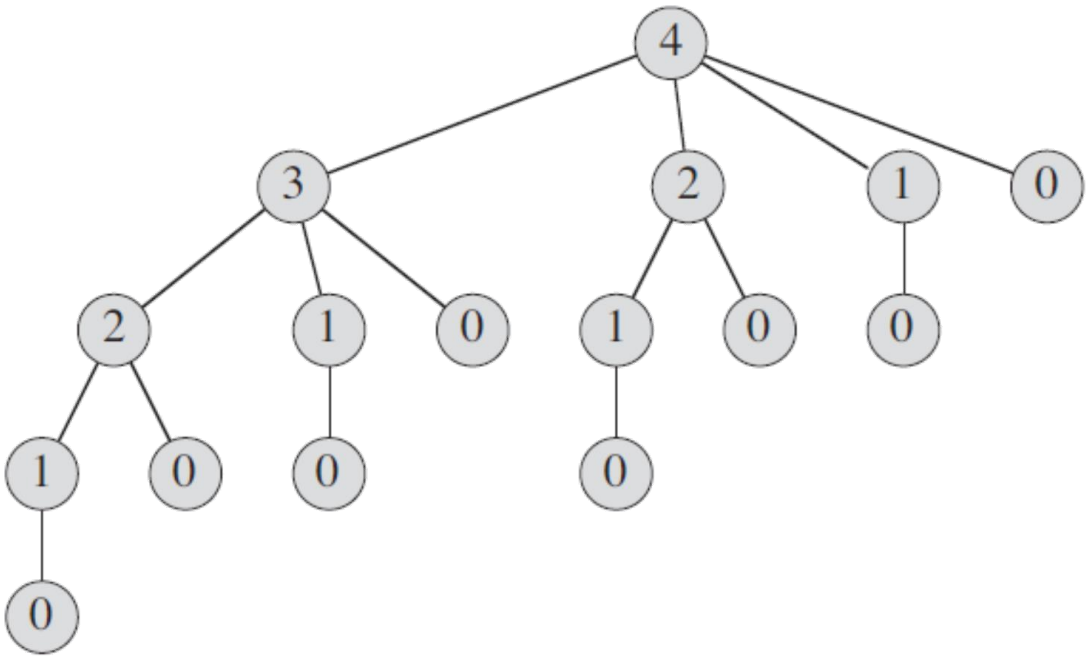
Una forma más sencilla de resolver este problema plantea que cada corte que realizamos nos proveerá una pieza que forma parte de la solución final (*i.e.*, ¿cuál trozo de tubo debo tomar para incluir en la solución óptima?) y un subtubo restante que debemos continuar cortando para obtener el resto de piezas. Entonces, la pieza que forma parte de la solución final ya no será cortada, y nos quedamos con un único subtubo que cortar en lugar de dos como con el acercamiento anterior.

10. Defina una relación de recurrencia que provea el valor de la solución óptima para el *rod-cutting problem*, tomando en cuenta la perspectiva que produce un único subproblema.

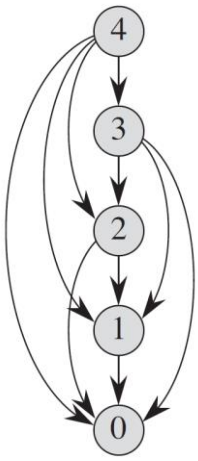
Denotemos la máxima ganancia que se obtiene de un tubo de longitud n como r_n y el precio de venta de un tubo de longitud i con p_i, entonces:

$$r_n = max_{1 \leq i \leq n}(p_i + r_{n-i})$$

Computar la ganancia máxima empleando un algoritmo que refleje recursivamente esta relación de recurrencia toma demasiado tiempo. La cantidad de resultados posibles sobre un tubo es exponencial sobre su longitud. Es en este momento donde debemos definir el algoritmo para computar la solución óptima usando la relación de recurrencia anterior. Este algoritmo, para ser de programación dinámica y ser más eficiente, debe aprovechar la propiedad de subproblemas traslapados. Para identificar los subproblemas traslapados, comencemos por dibujar el árbol de recursión para el *rod-cutting problem* sobre un tubo de tamaño $n = 4$:



Al fusionar los subproblemas que se repiten en el árbol formamos un grafo de subproblemas, que se vería así:



Apoyándonos en el grafo, procedamos a producir el pseudocódigo que calcula el valor de la solución óptima. Veamos primero la perspectiva *top-down* (**nota**: este algoritmo tiene AUX en su nombre porque es un método auxiliar para algoritmo en el libro de Cormen, que simplemente inicializa el arreglo r de n posiciones con valores centinela iguales a $(-\infty)$):

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

11. Provea el pseudocódigo que calcula el valor de la solución óptima para el *rod-cutting problem* usando el acercamiento *bottom-up.*

Al notar que cada nodo depende de todos los nodos "inferiores", podemos adoptar un acercamiento *bottom-up* y proponer el siguiente algoritmo:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

Observemos como este algoritmo va evaluando todos los cortes a partir de los casos triviales. El índice j determina la longitud del subtubo para el cual se está calculando la ganancia máxima obtenible, y el índice i va probando los posibles cortes sobre ese subtubo.

El último paso en este ejemplo es el de mostrar la construcción de la solución óptima, algo opcional dependiendo del problema que estemos resolviendo. A continuación, los pseudocódigos para ambos de los algoritmos descritos anteriormente, pero con las modificaciones necesarias para ir almacenando los cortes de la solución óptima conforme se calcula su valor.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

```
EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n, r, s)
 1  if r[n] ≥ 0
 2    return r[n]
 3  if n == 0
 4    q = 0
 5  else q = -∞
 6    for i = 1 to n
 7      res = EXTENDED-MEMOIZED-CUT-ROD-AUX(p, n - i, r, s)
 8      if q < (p[i] + res)
 9        q = (p[i] + res)
10        s[n] = i
11    r[n] = q
12  return q
```

Notemos el paso extra que almacena los cortes óptimos que deben realizarse para cada longitud de tubo en el arreglo s. Luego podemos extraer la lista de piezas que proveen la ganancia máxima de este arreglo s. Para el algoritmo *bottom-up* se haría de la siguiente forma:

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Terminemos el tema con algunos ejemplos adicionales. El problema de la **subsecuencia monótona más larga** (**LMS**, por *longest monotone subsequence*) nos pide que identifiquemos, dada una secuencia de cantidades $a_1, \dots, a_n \in \mathbf{N}$, la longitud de la secuencia $a_{i_1}, \dots, a_{i_k}$ donde $1 \leq i_1 < \cdots < i_k \leq n$ y $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_k}$. Primero demostremos que tiene subestructura óptima, por lo que necesitamos identificar las decisiones y subproblemas involucrados en la solución, así como demostrar la necesaria optimalidad de las soluciones a los subproblemas.

12. Caracterice la solución óptima para el problema de encontrar la longitud de la LMS para una secuencia $a_1, \dots, a_n$ dada.

Mientras más larga la secuencia original más grande podrá ser su LMS por haber una mayor cantidad de elementos. Pero el tamaño de la LMS estará limitado por la posición de la secuencia en la que termine. Si la LMS termina en la posición n habrá $n-1$ elementos adicionales que podrían pertenecer a la LMS, pero si la LMS termina en la posición 2 solo habrá un elemento (la posición 1) que podría pertenecer a esa LMS (aunque no necesariamente). La pregunta es, entonces, ¿en cuál posición termina la LMS del problema original?

Si la posición final de la LMS fuera la posición k tendríamos que buscar el resto de elementos de la LMS entre las posiciones 1 a $k-1$. Es más, los demás elementos de la LMS deberán ser menores a a_k, deberán estar ordenados de forma monótona ascendente y deberán conformar una secuencia de tamaño igual al de la LMS menos uno. En otras palabras, necesitaremos la LMS de $a_1, \dots, a_{k-1}$ cuyos elementos sean menores o iguales a a_k. Podemos aplicar nuevamente el mismo razonamiento para seguir buscando, de forma recursiva, la LMS que termina en a_k.

Notemos que puede haber más de una subsecuencia que termine en a_k. Supongamos que la LMS del problema original tiene tamaño m y es $a_{i_1}, \dots, a_{i_{m-1}}, a_k$, pero $a_{i_1}, \dots, a_{i_{m-1}}$ no es la LMS de elementos menores a a_k entre $a_1, \dots, a_{k-1}$. Eso querría decir que hay una subsecuencia monótona $a_{j_1}, \dots, a_{j_n}$ con tamaño $n > (m-1)$ y cuyos elementos son menores o iguales a a_k. Entonces, al agregar a_k a estas secuencias tendremos que nuestra LMS $a_{i_1}, \dots, a_{i_{m-1}}, a_k$ tiene tamaño m mientras que $a_{j_1}, \dots, a_{j_n}, a_k$ tendría tamaño $n+1 > m$. Esto contradice que $a_{i_1}, \dots, a_{i_{m-1}}, a_k$ sea la LMS del problema original. Por lo tanto, para que $a_{i_1}, \dots, a_{i_{m-1}}, a_k$ sea la LMS, la solución al subproblema de números menores a a_k entre $a_1, \dots, a_{k-1}$ debe ser la más larga posible, la óptima..., es decir, la LMS sobre esos números.

Nuestra solución procederá, entonces, revisando todos los números de la secuencia, uno por uno a partir del segundo (porque el primero conforma la subsecuencia trivial o caso base) y determinando la longitud de la subsecuencia más larga que termine en dicho número.

La recurrencia la definimos a continuación: sea $R(j)$ la longitud de la subsecuencia más larga que termina en a_j, con $R(1) = 1$. Entonces, para $j > 1$:

$$R(j) = \begin{cases} 1 \text{ si todo elemento precedente a } a_j \text{ es mayor que él} \\ 1 + \max\limits_{1 \le i < j} \{R(i) \mid a_i \le a_j\} \text{ de lo contrario} \end{cases}$$

El caso inferior dice que la longitud de la subsecuencia más larga que termine en a_j será 1 más la mayor de todas las longitudes de subsecuencias monótonas cuyos elementos finales sean cada uno de los $a_1, \dots, a_{j-1}$. Nótese que se considera la subsecuencia que termina en a_i si $a_i \le a_j$, porque de lo contrario la solución se tornaría inválida al agregar a a_j (ya que dejaría de ser una subsecuencia monótona).

El algoritmo *bottom-up* es el siguiente:

Algorithm 4.1 Longest monotone subsequence (LMS)

$R(1) \leftarrow 1$
for $j : 2..d$ **do**
    $\max \leftarrow 0$
    for $i : 1..j-1$ **do**
        if $R(i) > \max$ and $a_i \leq a_j$ **then**
            $\max \leftarrow R(i)$
        end if
    end for
    $R(j) \leftarrow \max + 1$
end for

Universidad del Valle de Guatemala
Análisis y Diseño de Algoritmos
Docentes: Paulo Mejía; Tomás Gálvez P.
Semestre 1, 2023

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Ejercicios

1. Suponga que al *rod-cutting problem* agregamos un límite piezas que se pueden vender o cortar para cada posible tamaño. Es decir que, para $0 < i \leq n, l_i \in \mathbf{N}$ es el número máximo de piezas de tamaño i que podemos usar en la solución. Demuestre que con esta nueva restricción el problema ya no exhibe una subestructura óptima. ***Hint***: ¿qué característica de los subproblemas se evaluó para determinar la ausencia de subestructura óptima en el *unweighted longest simple path problem*?
2. Volvamos al *rod-cutting problem* original. Resulta que no tenemos ni las herramientas ni las habilidades necesarias para cortar tubos, por lo que subcontratamos este servicio. Cada corte que queramos realizar costará una cantidad fija c. ¿Qué modificación se le tiene que hacer al algoritmo ya provisto para adaptarse a esta nueva condición?
3. Tenemos dos *strings* X e Y cuyos tamaños son m y n respectivamente. Para transformar X en Y le aplicamos una secuencia de operaciones cuyos resultados se van almacenando en un *string* Z. La transformación se lleva con índices i y j sobre X y Z respectivamente, iniciando con $i = j = 1$. Cuando la transformación concluye se debe tener que $i = m + 1, j = n$ y $Z = Y$. Las operaciones permitidas para la transformación son las siguientes:
    - ***Copy***: copia un caracter de X a Z haciendo $Z[j] = X[i]$, y luego incrementa tanto i como j.
    - ***Replace***: reemplaza un caracter de X con algún otro caracter c dado, haciendo $Z[j] = c$; y luego incrementa tanto i como j.
    - ***Delete***: ignora un caracter de X al incrementar i sin incrementar j.
    - ***Insert***: inserta en Z un caracter c dado, haciendo $Z[j] = c$, y luego incrementa j sin incrementar i.
    - ***Twiddle***: intercambia los siguientes dos caracteres de X copiándolos en orden inverso. Para lograrlo hace $Z[j] = X[i + 1]$ y $Z[j + 1] = X[i]$, y luego incrementa tanto i como j dos unidades (*i.e.*, $i = i + 2, j = j + 2$).
    - ***Kill***: ignora el resto de X haciendo $i = m + 1$. Esta operación, si se usa, debe ser la última.

    Cada operación tiene un costo constante propio, pero los costos de ***Copy*** y ***Replace*** son, cada uno, menores al costo de hacer ***Delete*** seguido de un ***Insert***. Considere el siguiente ejemplo ilustrativo que convierte la palabra *algorithm* en *altruistic*:

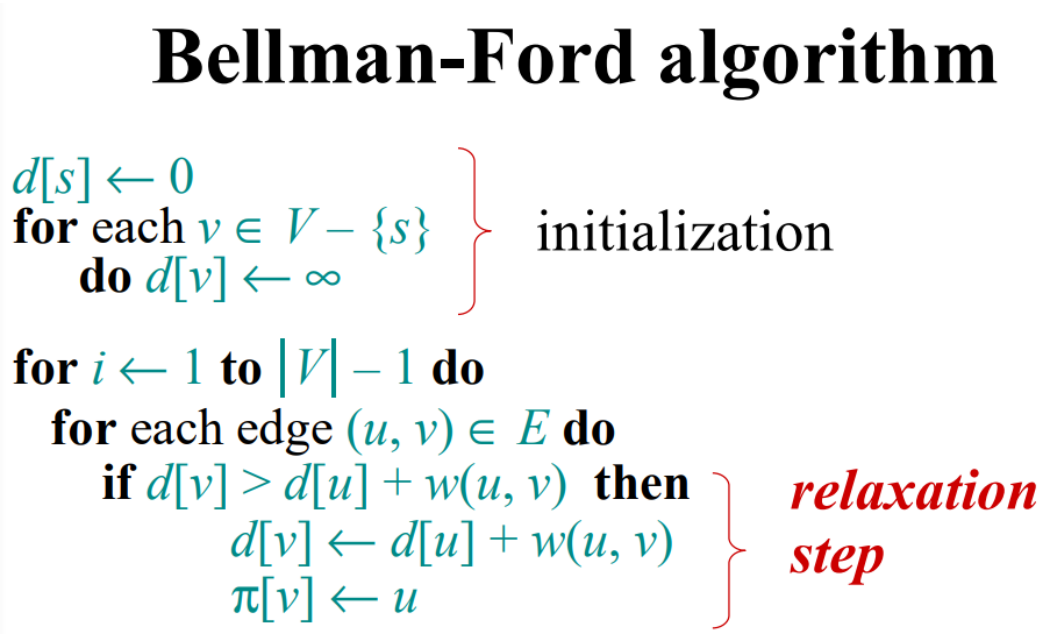
Operation	x	z
initial strings	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruis_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic_

Para este ejercicio deberá desarrollar un algoritmo apoyado en el acercamiento *bottom-up* de programación dinámica que permita encontrar la secuencia de operaciones que convierte un *string* en otro incurriendo en el menor costo posible (este costo mínimo es llamado la ***edit distance***). Para realizar el ejercicio siga los siguientes pasos:

a. Identifique la subestructura óptima siguiendo el procedimiento enseñado en clase. ***Hint***: para identificar los subproblemas, considere una secuencia de operaciones $(o_1, \dots, o_k)$ dada como óptima. Habiéndose aplicado alguna de las operaciones, ¿qué podemos decir que tiene que haber sucedido antes de aplicarse dicha operación? ¿Cuál de los pasos del ejemplo hace la conversión más sencilla (caso base)?
b. Esboce una tabla T con m filas y n columnas. Esta tabla debe llenarse durante la ejecución de su solución *bottom-up*. ¿Cuál es el significado del contenido de la celda $T[i, j]$?
c. Apoyándose en el inciso anterior, escriba la ecuación recurrente que computa el valor de la celda $T[i, j]$ tomando en cuenta las condiciones que restringen el uso de cada operación. Puede describir el costo de una operación como $\text{costo}(nombre\ de\ operación)$.

4. Considere el siguiente problema: dado un grafo dirigido y ponderado $G = (V, E)$, una función de peso $w: E \to \mathbf{R}^+$ y un vértice origen $s \in V$, encontrar el camino más corto desde s hasta cualquier otro vértice. Este problema es resuelto por el **algoritmo de Bellman-Ford**, presentado a continuación. En el algoritmo, d es el arreglo de soluciones que se llena con las distancias más cortas desde el vértice origen s hasta cada uno de los demás vértices en el grafo. π es el arreglo que contiene, para un vértice v dado, el vértice que le precede en el camino más corto de s hacia v.

Bellman-Ford algorithm



Busque o desarrolle un ejemplo de aplicación de este algoritmo por pasos. Apoyándose en el algoritmo, identifique la subestructura óptima y los subproblemas traslapados del problema. Luego provea y explique la recurrencia que calcula el valor de la solución óptima. Observe que, aunque d es un arreglo, todos los valores de d se actualizan varias veces conforme avanza el ciclo externo.

Fuentes:

- Chumbley, A., Moore, K., Ross, E., & Khim, J. (2019, may 19). *Bellman-Ford Algorithm*. Retrieved from Brilliant.org: https://brilliant.org/wiki/bellman-ford-algorithm/
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* Massachusetts: The MIT Press.
- Demaine, E., & Wenk, C. (2009). *Shortest Paths in Graphs.* Retrieved from CSc 545 Design and Analysis of Algorithms: https://www2.cs.arizona.edu/classes/cs545/fall09/ShortestPath2.prn.pdf
- Dunne, P. (n.d.). *Techniques for the Design of Algorithms*. Retrieved from Algorithm Design Paradigms - Overview of Course: http://cgi.csc.liv.ac.uk/~ped/teachadmin/algor/intro.html
- Neapolitan, R. E., & Naimipour, K. (1998). *Foundatious of Algorithms using C++ Pseudocode.* Sudbury, Massachusetts: Jones and Bartlett Publishers.
- Soltys, M. (2012). *An Introduction to the Analysis of Algorithms.* Singapore: World Scientific Publishing.