TEORÍA DE ALGORITMOS 1
Programación dinámica

Por: ING. VÍCTOR DANIEL PODBEREZSKI
vpodberezski@fi.uba.ar

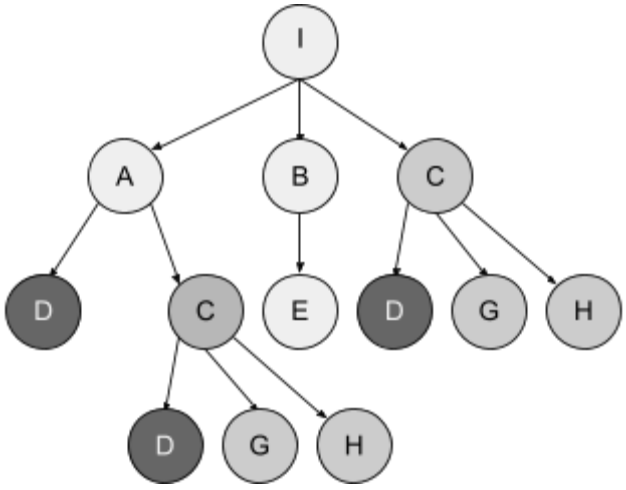
1. Introducción

La próxima metodología de resolución de problemas corresponde a una técnica poderosa y elegante. El nombre de esta es "**Programación dinámica**" (dynamic programming). A diferencia de otras, se conoce al momento y a la persona que la creó y nombró. Es una historia pintoresca. Su denominación fue pensada para ser confusa y enmascarar lo que realmente es. Su creador Richard Bellman lo explico en sus memorias[1] [2]. En 1949 el autor ingresó a trabajar en la RAND Corporation, que realizaba proyectos para la Fuerza Aérea de EE. UU. El campo de actividades que seleccionó fue la **teoría matemática de los procesos de decisión de múltiples etapas**. Sin embargo para lograr el apoyo y financiamiento tuvo que seleccionar un nombre para presentar, lograr apoyo y financiación. Quien tenía un peso preponderante en esta decisión era el secretario de defensa. Este individuo - según palabras de Bellman - tenía un rechazo patológico a la matemáticas y la investigación. Por lo que apeló a la creatividad para encontrar un nombre que vagamente explique su materia de estudió, tuviese un "sentimiento" positivo y sirviese como paraguas para sus actividades. En otoño de 1950 la programación dinámica había nacido.

Entonces, ¿qué es la programación dinámica? Es una metodología de resolución de problema de optimización donde queremos maximizar (o minimizar) un resultado. Comparte ciertas características de las metodologías greedy y de división y conquista: divide el problema en diferentes subproblemas. Por lo tanto - al igual que en estos - el problema debe exhibir una **subestructura óptima** para poder ser resuelto en forma óptima por ese mecanismo.

[1] Eye of the Hurricane, Richard Bellman, 1984, World Scientific Publishing Company
[2] Richard Bellman on the birth of dynamic programming, Stuart Dreyfus, 1 Febrero 2002, Operations Research

Sin embargo, el mejor camino para entenderlo es emparentarlo con la metodología de backtracking de la búsqueda exhaustiva. Podemos visualizar un problema de procesos de decisión de múltiples etapas como un árbol donde cada subproblema corresponde a un nodo. En backtracking para encontrar la solución óptima debemos recorrer ese árbol. Partimos de la raíz (el problema a resolver) y el camino realizado para llegar a una hoja representa un set de decisiones tomadas para llegar a una solución del mismo. Cada decisión en un subproblema nos lleva a un subproblema menor. Recorrer todo el árbol implica determinar entre todas las posibles soluciones cuál corresponde a la óptima. Tomar una decisión óptima para un determinado subproblema implica haber resuelto sus descendientes anteriormente. Por lo que determinar la solución del problema inicial implica revisar todas las decisiones posibles (que es lo mismo que resolver todos los subproblemas).



Un algoritmo de programación dinámica parte de la misma idea, pero su gran poder está en la naturaleza de los subproblemas en los que va trabajando. A medida que atraviesa a los mismos y los va resolviendo, va almacenando los resultados obtenidos. Este proceso se conoce como **memorización** ("memoization"). El objetivo es poder aprovechar estos resultados puesto que se espera que los mismos subproblemas vuelvan a aparecer en otras ramas de nuestro árbol. Si esto ocurre - y esperamos que lo haga - decimos que nuestro problema tiene la propiedad de **solapamiento de subproblemas** ("overlapping subproblems"). Gracias a eso disminuimos de forma considerable el tiempo de procesamiento total dado que evitamos recalcular porciones del árbol. Una elección inteligente (y afortunada) de la definición del subproblema es el punto de partida fundamental para este tipo de algoritmos.

En 1957 Bellman publica el libro "dynamic programming"[3] con un resumen de su investigación y un extenso conjunto de problemas de diversas disciplinas donde aplicarlo.

[3] Dynamic programming, Richard Bellman, 1957, Princeton University Press

Con el paso de los años se fueron sumando una gran cantidad de problemas donde la programación dinámica los resuelve de forma eficiente. Veremos algunos de ellos a continuación. Con la práctica entenderemos los conceptos antes vertidos.

2. Problema “Fraccionamiento de aceite de oliva”

El problema que analizaremos en primer lugar corresponde a un sencillo ejemplo de optimización. Lo enunciamos de la siguiente manera:

Fraccionamiento de aceite de oliva
Contamos con un depósito de “l” litros lleno de aceite de oliva. Queremos fraccionarlo y venderlo en el mercado. De acuerdo a las especificaciones vigentes solo se puede fraccionar en valores enteros de libros. Una tabla regulatoria determina el valor de venta para cada “i” litros como v_i. Podemos fraccionar como queramos, pero deseamos maximizar la ganancia.

Una instancia del problema podría corresponder al siguiente ejemplo:

Nuestra primera producción artesanal de 5 litros de aceite extra virgen y el costo de venta por litro es el siguiente:

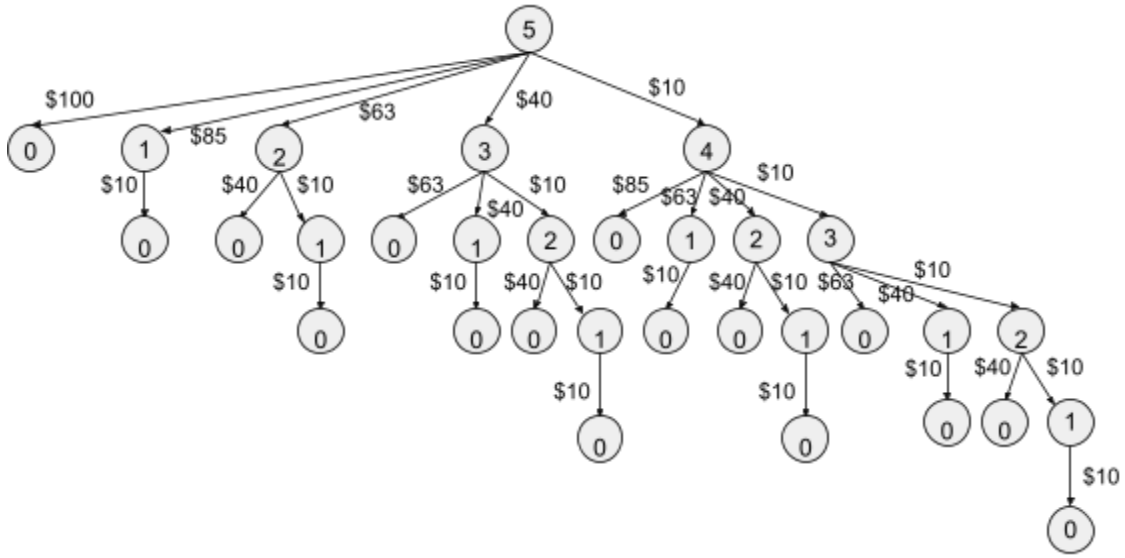
i	1	2	3	4	5
v_i	10	40	63	85	100

Una primera idea para construir un algoritmo de resolución es probar con la metodología greedy. El problema corresponde a una cantidad de litros disponibles en un determinado momento y la elección greedy podría variar. Tal vez primero vender el mayor tamaño posible, o el de mayor precio disponible, o el que da mayor ganancia por litro (litros/valor venta). Sin embargo para cada uno de ellos existirá un contraejemplo posible.

Si intentamos vender primero el de mayor costo o el de mayor cantidad comenzamos vendiendo 5 litros a $100. Sin embargo si elegimos vender primero el que nos otorga mayor ganancia por litro comenzamos vendiendo 4 litros a $85 y vendiendo el restante litro por $10. Un total de $95 que es menor a la opción anterior. Pero si vendemos primero 3 y luego 2 litros conseguiremos $103 que en este caso es la mayor ganancia posible.

Ciertamente podemos resolver el problema mediante backtracking. En ese caso partimos del problema inicial y podemos seleccionar diferentes ventas iniciales. Cada una de estas dará una ganancia y generará un nuevo problema a examinar con menor cantidad de litros disponibles. Seguiremos recursivamente hasta quedarnos en el caso base que equivale a no tener más aceite. Nos queda un árbol combinatorio. Podemos desde cada hoja hasta la raíz ir sumando las ganancias en su camino y de esa forma encontrar el conjunto de decisión que nos lleve a la solución óptima. Deberemos analizar un total de 2^l subproblemas. Lo que ciertamente no corresponde a un algoritmo eficiente.

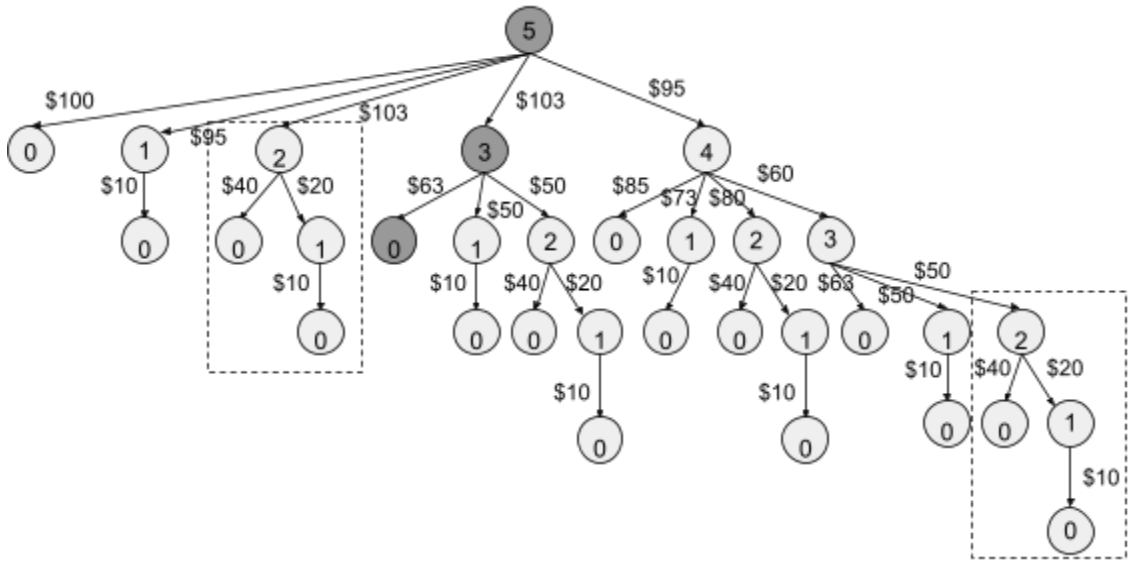
La raíz de nuestro árbol representa el problema inicial, donde tenemos 5 litros disponibles. De allí se desprenden 5 subproblemas. Respectivamente el resultante de vender 1, 2, 3, 4 o 5 litros. En estos nuevos subproblemas podremos elegir nuevamente cuántos litros vender, restringido por la cantidad aún disponible. Existen 16 alternativas de venta (número de hojas = 2^{l-1}) que encontramos luego de analizar 32 subproblemas ($2^l=2^5$). Encontramos que la opción de vender 3 y luego 2 litros (o también al revés) nos brinda la solución máxima (y por lo tanto óptima). En la siguiente imagen se pueden observar todos los subproblemas.



Analizaremos el árbol resultante. En primer lugar podemos marcar al camino “M” que nos brinda la maximización de la ganancia. Ese camino comienza en una hoja “o” y termina al nodo raíz “r”. La ganancia de “M” es la suma de las ganancias que obtengo por atravesar cada uno de los nodos de este. Supongamos que me paro en un determinado nodo interno “i” de ese camino “M”. Llamaremos “o→i” al trayecto del camino “M” que parte de “o” y llega

a “i”. La ganancia de “M” la puedo expresar como la ganancia de “o→i” más la ganancia de “i→r”. Pueden existir otros caminos que llegan a “i” partiendo de otras hojas del árbol. Tomemos un camino “o’→i” (con o’ ≠ o). También podemos computar su ganancia. Afirmamos que la ganancia de “o’→i” debe ser menor (o a lo sumo igual) a la ganancia “o→i”. Supongamos por un momento que esto no ocurre. En ese caso “M” no sería el camino máximo. Dado que simplemente reemplazando “o’→i” por “o→i” sería también un camino de una hoja a la raíz y tendría una mayor ganancia. Por lo tanto el camino para la ganancia máxima se arma utilizando las ganancias máximas de sus caminos interiores. Podemos extender el mismo principio para cualquier nodo de nuestro árbol. El camino que maximiza la ganancia para este nodo se arma utilizando las ganancias máximas de sus caminos interiores. Se aplica exactamente la misma demostración. Nuestro problema tiene una **subestructura óptima**.

En nuestro ejemplo llamamos “M” al camino que comienza con 5 litros en la raíz y vende inicialmente 2 para luego vender 3. Comenzando por la hoja podemos ver a “M” como 0→3→5. Para llegar a “3” realizamos el subcamino 0→3 con una ganancia de $63. Existen otras tres alternativas de caminos: 0→1→3 (ganancia:$50), 0→2→3 ($50), 0→1→2→3 ($30). Todas ellas con menor ganancia que la seleccionada para el camino máximo. En la imagen siguiente se observa la instancia de ejemplo. Se muestra la ganancia del camino mínimo a cada uno de los nodos del árbol.



Traslademos lo que sabemos hasta ahora a una expresión matemática. Utilizaremos una ecuación de recurrencia. Llamaremos X a la cantidad de litros con los que se cuenta en un determinado momento. Definimos a nuestro subproblema como la máxima ganancia que puedo obtener con X litros como OPT(X). Podemos elegir vender i=1,2,...,X litros. Para la elección "i" tendremos una ganancia v_i y me quedará el subproblema OPT(X-i). Como queremos maximizar nos quedamos con el máximo valor entre ellos. La ecuación de recurrencia es $OPT(X) = MAX_{i=1}^{X}(v_i + OPT(X - i))$. El caso base corresponde a cuando no tengo más aceite y por lo tanto la ganancia es $OPT(X = 0) = 0$.

Podemos expresar en pseudocódigo esta forma de resolver el problema:

Fraccionamiento de aceite de oliva - Backtracking

```
Sea L la cantidad de litros disponibles
Sea v[i] la ganancia por vender i litros de aceite

MaximaGanancia(X):
        Si X=0
                Retornar 0

        Max = 0
        Desde i=1 a X
                gan = v[i] + MaximaGanancia(X-i)
                Si gan>Max
                        Max = gan
        Retornar Max

MaximaGanancia(L)
```

Resolver este problema recursivamente tendrá (tal como vimos antes) una complejidad temporal exponencial y espacial constante. Sin embargo, podemos notar que el mismo subproblema aparece y vuelve a aparecer más de una vez. Por la naturaleza del problema, cada vez que tengo X litros la descomposición de subproblemas será la misma. No importa como llego desde la raíz al subproblema, el óptimo a X litros se encontrará entre todos los subproblemas descendientes a este. Por lo tanto cada vez que calcule el resultado será el mismo. Basta con hacerlo solo una vez. Estamos entonces al frente de la propiedad de **solapamiento de subproblemas**.

En la imagen antes mostrada se puede observar en un recuadro 2 apariciones del subproblema de fraccionar 2 litros. Prestando atención vemos que vender 4 litros aparece 1 vez (2^0) . Vender 3 litros aparece 2 veces (2^1). Vender 2 litros, 4 veces (2^2)... Finalmente tendremos el problema base, vender 0 litros que lo debemos resolver 2^l veces.

Mediante programación dinámica podemos evitar el recálculo de los subproblemas que ya aparecieron. Esto se realiza utilizando la técnica de **memorización**. Esta técnica implica utilizar memoria adicional para almacenar los subproblemas que se van resolviendo. El costo pagado es pasar a tener una complejidad espacial lineal. Almacenamos los l subproblemas la primera vez que lo solucionamos. En pseudocódigo lo podemos ver como:

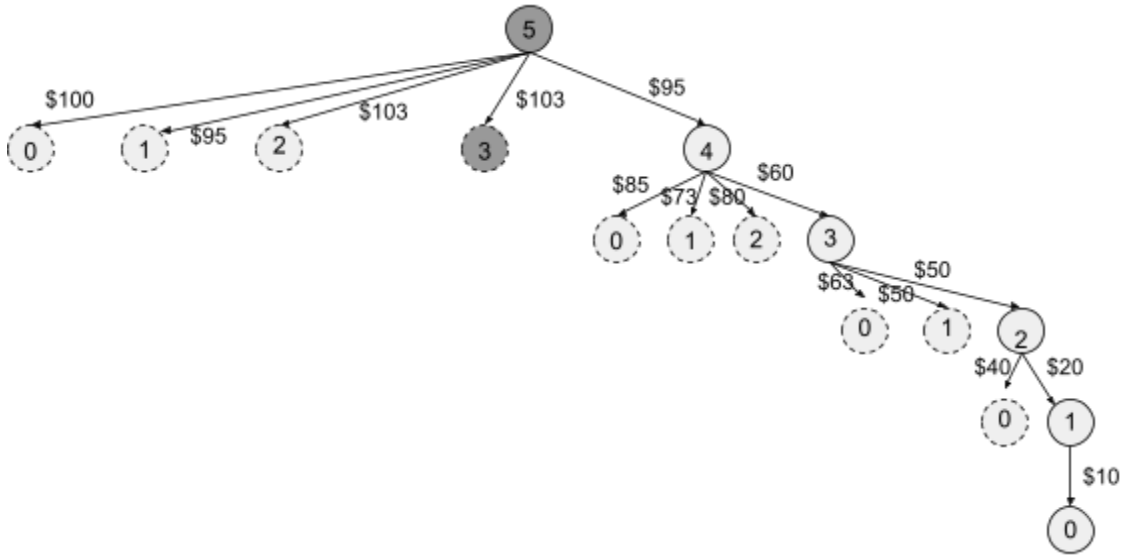
Fraccionamiento de aceite de oliva - Programación dinámica (De arriba hacia abajo)

```
Sea L la cantidad de litros disponibles
Sea v[i] la ganancia por vender i litros de aceite
Sea M[i] la máxima ganancia posible por vender i litros

MaxMemorizada(j):
        Si M[j] no está definido
                M[j] = MaximaGanancia(j)
        Retornar M[j]

MaximaGanancia(X):
        Si X=0
                Retornar 0

        Max = 0
        Desde i=1 a X
                gan = v[i] + MaxMemorizada(X-i)
                Si gan>Max
                        Max = gan
        Retornar Max

MaximaGanancia(L)
```

¿Cómo se modifica la complejidad temporal? Se observa que se podan del árbol de subproblemas todos aquellos previamente calculados. Para calcular el subproblema con cantidad de x litros se consulta x-1 subproblemas. Que estarán previamente calculados. En cada subnivel se realizan O(x-1) operaciones. Por lo que el proceso general será la suma de

O(1+2+...+l) que equivale a decir que corresponde globalmente a una complejidad de $O(l^2)$. Una mejora considerable. Podemos decir que estamos frente a un algoritmo bueno.

Aplicando memorización se puede observar que para calcular la solución óptima para 5 litros, se reduce considerablemente la cantidad de subproblemas. En la recursión se iniciará el intento de calcular siempre primero fraccionando de 1 litro. Por lo que se calculará primero el camino 5→4→3→2→1→0. Una vez calculado el subproblema 0. Se resuelve inmediatamente el 1. Luego para calcular el subproblema 2 se consulta el 1 y el 0 (ambos precalculados). Lo mismo pasará con el resto de los subproblemas mayores. En la imagen siguiente se muestran en línea continua los subproblemas que se invocan por primera vez y en línea con rayas aquellos que son consultados ya una vez calculados. Se puede observar que por cada nivel "x" solo calculo un único subproblema y consulto x-1 subproblemas previamente calculado.



La forma mostrada para resolver el problema se conoce como **enfoque de arriba hacia abajo** (Top-Down Approach). Corresponde a un algoritmo recursivo que comienza el proceso desde el problema más grande y va descendiendo a los más pequeños. Existe una manera alternativa conocida como **enfoque de abajo hacia arriba** (Bottom-Up Approach). En esta segunda forma la resolución inicia en el problema más pequeño e iterativamente se progresa a problemas mayores. Podemos expresar el algoritmo de resolución de nuestro problema utilizando este segundo enfoque como:

Fraccionamiento de aceite de oliva - Programación dinámica (De abajo hacia arriba)

```
Sea L la cantidad de litros disponibles
Sea v[i] la ganancia por vender i litros de aceite
Sea M[i] la máxima ganancia posible por vender i litros

M[0]=0
Desde j=1 a L
        M[j]=0
        Desde i=1 a j
                gan = v[i] + M[j-i]
                Si gan>M[j]
                        M[j] = gan

Retornar M[L]
```

De esta forma evitamos tener que realizar una recursión. Además queda más evidente las complejidades del algoritmo (son equivalentes a la utilizada con la otra aproximación).

Antes de finalizar queda pendiente un punto. En los pseudocódigos que se presentaron la salida luego de su ejecución muestra la ganancia máxima que se puede obtener. Sin embargo no permiten reconstruir qué elecciones nos permite obtener la misma. Esto se puede hacer sin mucho esfuerzo. En primer lugar para cada subproblema debemos almacenar qué opción seleccionamos para maximizar ese subproblema. Una vez resueltos todos los subproblemas podemos desde el problema más grande reconstruir el camino de decisiones. El siguiente pseudocódigo modificado contiene esta funcionalidad

Fraccionamiento de aceite de oliva - Mostrando fraccionamientos seleccionados

```
Sea L la cantidad de litros disponibles
Sea v[i] la ganancia por vender i litros de aceite
Sea M[i] la máxima ganancia posible por vender i litros
Sea F[i] el fraccionamiento seleccionado para lograr máxima ganancia al vender i litros.

M[0]=0
F[i]=0
Desde j=1 a L
        M[j]=0
        Desde i=1 a j
                gan = v[i] + M[j-i]
                Si gan>M[j]
```

```
                    F[j] = i
                    M[j] = gan

j=L
Mientras j≠0
        Imprimir F[j]
        j=j-F[j]

Imprimir M[L]
```

3. Máxima cantidad de intervalos con valor

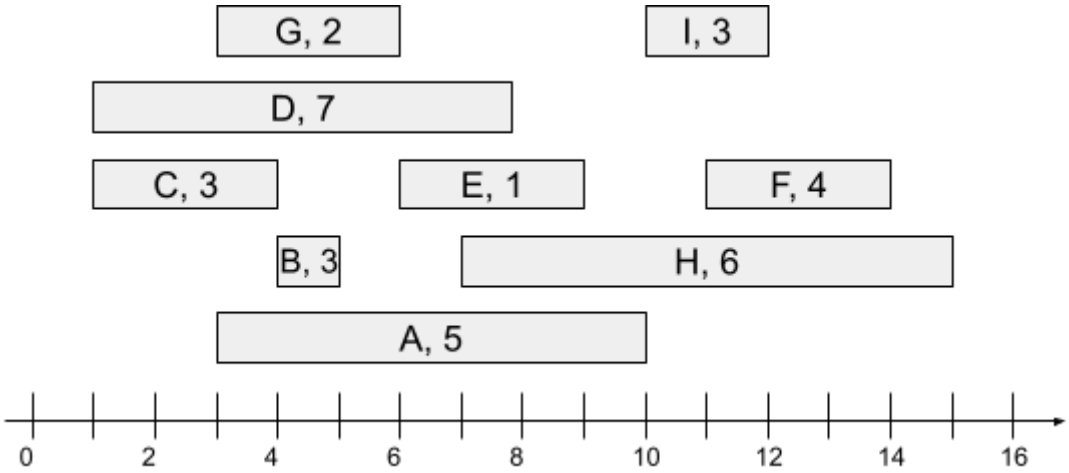
En ocasiones el agregado de una "pequeña" variante a un problema puede ocasionar que una estrategia exitosa de resolución deje de serlo. Cuando trabajamos sobre el problema de selección de tareas, intentamos maximizar la cantidad de elementos seleccionados. Un ejemplo de aplicación podría ser el siguiente ejemplo. Un salón de convenciones desea seleccionar la mayor cantidad de eventos a realizarse entre los "n" disponibles. La solución nos dará un subconjunto de eventos cuyo número es igual o mayor a cualquier otro subconjunto compatible entre sí. Recordemos que un par de eventos son compatibles entre sí, si no se superponen en el tiempo. Ahora bien, puede ocurrir que no todas los eventos representen el mismo beneficio ante su realización. Tal vez consigamos una ganancia por realizarlos. Nuestra propuesta previa ya no será eficaz. Nos veremos obligados a buscar una alternativa. La programación dinámica nos será de utilidad.

Comencemos definiendo esta variante - más general - del problema de selección de intervalos.

Selección de actividades con valor
Contamos con un conjunto de "n" actividades entre las que se puede optar por realizar. Cada actividad x tiene una fecha de inicio I_x, una fecha de finalización f_x y un valor v_x que obtenemos por realizarla. Seleccionar el subconjunto de actividades compatibles entre sí que maximice la ganancia a obtener (suma de los valores del subconjunto).

El emprendedor que compró una lancha para realizar paseos de pesca en un lago modificó su plan de negocios. Cada paseo que ofrece es personalizable tanto en sus fechas como en

servicios adicionales. Para la próxima quincena se le presenta un conjunto de excursiones a realizar. Cada excursión tiene un día de inicio, de finalización y una ganancia monetaria. Desea aceptar aquellas que le permitan maximizar su ganancia. Las 9 opciones son (identificador, inicio,fin,ganancia): (A,3,10,5), (B,4,5,3), (C,1,4,3), (D,1,8,7), (E,6,9,1), (F,11,14,4), (G,3,6,2), (H,7,15,6), (I,10,12,3)



Se pueden probar diferentes criterios greedy. Entre ellos, por mayor ganancia, fecha de inicio, fecha de finalización, duración, ganancia por unidad de tiempo. Para cada una de ellas encontraremos un contraejemplo.

Intentaremos encontrar para este problema una **subestructura óptima**. Una forma de dividir el problema en otros menores, cuya resolución nos facilite alcanzar la solución general. Cuando trabajamos con la metodología greedy evaluamos los intervalos ordenados por fecha de finalización. Considerando primero aquellos que lo hacían antes. De esta forma se obtiene la solución óptima. Además, encontramos una segunda estrategia óptima ordenando por fecha de inicio y considerando primero aquellos que lo hacen después. Para los siguientes planteamientos utilizaremos el primero de los criterios.

Desde la perspectiva de un intervalo y siguiendo el ordenamiento establecido, todos los intervalos previos finalizan antes (o al mismo tiempo) que él. Entre estos, algunos se solapan y otros no. Por el ordenamiento (fecha de finalización), los antecesores que se solapen estarán inmediatamente antes del mismo. A partir de un determinado momento, ya no se solapan. Lo mismo no se puede decir de observando los posteriores. En ese caso podrán estar totalmente intercalados con los que se solapa y con los que no. Con esto en

mente podemos idear una jerarquía de subproblemas considerando solo los anteriores y aprovechando la observación del solapamiento.

Ordenamos los intervalos por fecha de finalización y tendremos (C,1,4,3), (B,4,5,3), (G,3,6,2), (D,1,8,7), (E,6,9,1), (A,3,10,5), (I,10,12,3), (F,11,14,4), (H,7,15,6). Si consideramos el intervalo "E" podemos ver que el inmediatamente anterior "D" se solapa con el. Pero no se solapa con todos los anteriores a este último ("C","B","G"). Para el mismo intervalo si analizamos los siguientes vemos que se solapa con "A" y "H" (el inmediatamente posterior y el último en la lista respectivamente). No se solapa con "I" y "F".

Podemos inquirir si un intervalo se encuentra o no en una solución óptima. Si la respuesta es positiva, significa que obtendremos su ganancia. También, todos aquellos con los que se superpone no los podremos seleccionar. En particular, con el ordenamiento, imposibilita a algunos de los intervalos inmediatamentes anteriores a coexistir con este en la solución. A partir del primero con el que no se superpone podemos volver a preguntarnos si se encuentra o no en el subconjunto óptimo. De manera análoga si un intervalo no se encuentra en la solución no aporta su posible ganancia. El contrato inmediatamente anterior según el ordenamiento (tanto si se superpone como si no) podrá pertenecer al óptimo.

Si el intervalo "H" pertenece al óptimo entonces nos brindará una ganancia de $6. Además considerando el ordenamiento los intervalos anteriores que se superponen son "D", "E", "A", "I" y "F". Ninguno de ellos podría estar en la solución óptima. Recién el intervalo "G" podría estar acompañando a "H" en una solución óptima.

Antes de continuar, estableceremos algunas definiciones que nos serán útiles. Los intervalos ordenados quedan en una posición determinada. La utilizaremos para identificarlas. Aquel que finaliza primero que todo el resto tendrá el número 1, el que le sigue el 2 y de igual forma para el resto. Finalizamos con el número n para el último. Definimos la función "PrevioCompatible(i)" que nos indica la posición del primer intervalo previo a i con el que es compatible. Si no hay compatibles anteriores el resultado corresponde al valor cero. Llamamos Ganancia(i) al valor que obtenemos si incluimos al intervalo en la posición i en el conjunto

Según el orden de los intervalos podemos decir que "C" está en la posición 1. B en la posición 2. G tendrá la posición 3 y finalizamos con H en la posición 9. La función PrevioCompatible(4) busca el primer intervalo previo compatible a "D". Si observamos, notaremos que no hay ninguno anterior compatible. El resultado de la función será 0 (cero). Si ahora invocamos la funcion PrevioCompatible(5) estaremos haciendo referencia al intervalo "E". El valor que retorna la función es 3 que corresponde al intervalo "G".

Es momento de definir nuestro subproblema. Llamaremos MAX(i) a la ganancia máxima que podemos obtener teniendo en cuenta únicamente los intervalos de 1 a i (en el orden que definimos). Para el caso base diremos que no considerar ningún intervalo o MAX(0) nos brinda una ganancia de cero. Este corresponde a nuestro caso base. Si solo consideramos el intervalo 1 tenemos dos casos: optar por incluirlo o no. Es trivial en este caso ver que lo conveniente es hacerlo. Para un caso genérico i también tenemos las dos opciones. En ese caso nos quedaremos con la máxima ganancia posible entre ellas. Si se incluye el intervalo entonces tendremos su ganancia más la máxima ganancia que se puede conseguir del intervalo previo compatible a i. Si no se incluye entonces consideramos la máxima ganancia que se puede obtener con el intervalo i-1. Podremos expresar estos conceptos mediante una ecuación de recurrencia

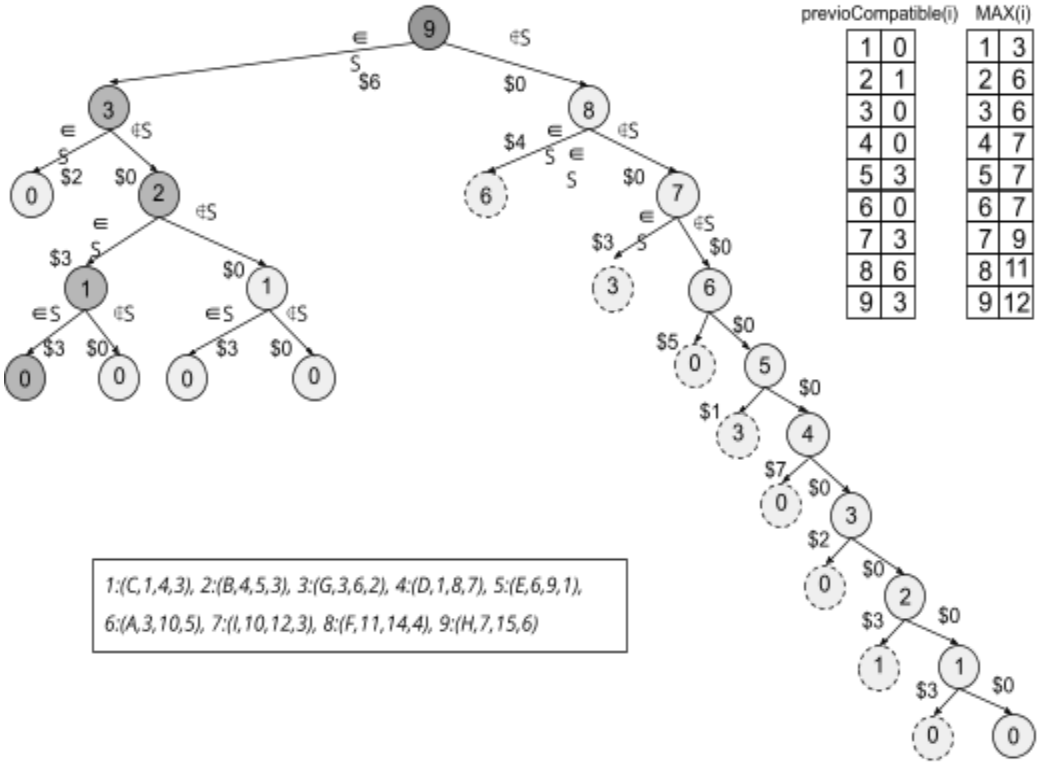
$$MAX(0) = 0$$

$$MAX(i) = Max\{Ganancia(i) + MAX(PrevioCompatible(i)), MAX(i+1)\}\ i \neq 0$$

Obtendremos el máximo valor del problema al calcular MAX(n). Observamos que para hacer este último cálculo deberemos calcular los máximos para los n-1 intervalos anteriores. Esto por consideramos en cada uno de los subproblemas no seleccionar el intervalo i. Muchos de los subproblemas pueden aparecer más de una vez. Contamos con **solapamiento de subproblemas** y podemos memorizarlos para evitar recalcular.

La jerarquía de subproblemas conforma un árbol binario. Para cada subproblema se puede decidir seleccionar el intervalo o no. En la imagen siguiente se puede ver el árbol para el ejemplo del emprendedor. Cada nodo interno tiene dos hijos. El de la izquierda corresponde a la acción de seleccionar el intervalo del subproblema. La derecha a no hacerlo. Se puede ver que varios subproblemas vuelven a aparecer y por lo tanto se puede calcular una vez, memorizar y utilizar el resultado al volver a aparecer. En la primera tabla se muestra para cada intervalo cual es el

previo compatible. En la segunda el resultado de cada subproblema. En el árbol se muestra expandida y en color más oscuro el camino de decisiones que lleva a la solución general máxima.



A continuación se muestra el pseudocódigo expresado de abajo hacia arriba. Se incluye la estructura para registrar en cada subproblema si a la elección del máximo se arriba seleccionando o no el intervalo analizado.

Fraccionamiento de aceite de oliva - Programación dinámica (De abajo hacia arriba)
Sea n intervalos ordenados por fecha de finalización Sea ganancia[i] la ganancia por incluir el intervalo i Sea previoCompatible[i] el primer intervalo anterior a i compatible Sea MAX[i] la máxima ganancia de los primeros i intervalos Sea seleccion[i] la conveniencia de seleccionar el intervalo i en el subproblema MAX[i] MAX[0]=0 Desde i=1 a n enOptimo = ganancia(i) + MAX[previoCompatible[i]] noEnOptimo = MAX[i-1]

```
        Si enOptimo>noEnOptimo
                MAX[i] = enOptimo
                seleccion[i] = true
        Sino
                MAX[i] = noEnOptimo
                seleccion[i] = true

Sea i=n
Sea intervalos el conjunto de los intervalos seleccionados en la solución óptima
Mientras i<>0
        Si seleccion[i] = true
                intervalos = intervalos ∪ i
                i=previoCompatible[i]
        Sino
                i= i-1

Retornar MAX[n] e intervalos
```

La complejidad de este algoritmo se puede calcular como la suma de las complejidad de cada una de sus partes. En primer lugar requerimos ordenar los “n” intervalos. Podemos hacerlo en O(nlogn). Posteriormente debemos calcular cada uno de los subproblemas. Son “n” en total y la complejidad de cada uno de ellos es O(1). Por lo que tenemos un proceso O(n). Reconstruir los intervalos de la solución óptima corresponde a O(n) dado que en el peor de los casos todos los intervalos están en la misma. Resta determinar cómo calcular el previoCompatible. Para eso se puede aprovechar el ordenamiento. Lo que se debe realizar es, para cada intervalo i ver su fecha de inicio. Teniendo la fecha, se busca de forma binaria en los i-1 primeros intervalos a partir de que fecha de finalización de estos puede incluirse i. Este proceso será O(nlogn). Por lo tanto, globalmente tendremos una complejidad temporal de O(nlogn). En cuanto a la complejidad espacial, por las estructuras necesarias tendremos un O(n) de requerimiento.

4. Problema del subvector máximo

El siguiente problema fue propuesto por el matemático Ulf Grenander en 1977. Corresponde a la simplificación de otro más complejo: la estimación de máxima verosimilitud de patrones en imágenes digitalizadas. Se lo conoce como “problema del

subvector máximo" (Maximum Subarray Problem) o como "problema de máxima suma de segmento" (maximum segment sum problem). Su enunciado lo podemos expresar como:

Máxima suma de segmento
Dado una lista L de n elementos. Cada elemento tiene asociado un valor numérico (positivo o negativo). Queremos encontrar el subconjunto contiguo de elementos que sume el mayor valor posible.

Supongamos que contamos con la siguiente lista L={-3, 5, -3, 4, 2, 1, -10, 2, 2, -2, 1, 5}. Un subconjunto contiguo posible es A={ 2, 2, -2, 1, 5} cuya suma corresponde a 2+2+(-2)+1+5=8. Sin embargo este no es el máximo posible. Buscando se puede hallar B={5, -3, 4, 2, 1} cuya suma total es 9. No es posible encontrar en este ejemplo otro subconjunto de mayor valor, por lo tanto podemos afirmar que este corresponde al subvector máximo para esta instancia del problema.

Existen varías formas de resolver este problema. Desde fuerza bruta, división y conquista y finalmente programación dinámica. Varias de las resoluciones fueron descriptas por Jon Bentley en un artículo "Programming pearls: algorithm design techniques"[4]. Allí narra además la historia del problema y de su propuesta de solución por Jay Kadane utilizando programación dinámica. El estadístico de la Universidad de Carnegie-Mellon solo requirió un minuto para hallar el método que hoy es conocido por su nombre.

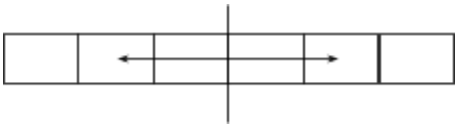
Comencemos por la resolución por fuerza bruta. En este caso requerimos evaluar cada uno de los posibles subconjuntos y seleccionar aquel cuya suma es mayor al resto. Se tendrá un subconjunto de n elementos (todo el vector), dos vectores de n-1 elementos, tres de n-2 elementos… hasta finalizar con n de 1 elemento. Sumarizando en total $O(n^2)$ posibles subconjuntos. En cada uno de ellos debemos sumar sus elementos. Proceso que se puede realizar en un O(n) adicional. Por lo tanto partimos de una complejidad temporal cúbica y una espacial de O(1). Un posible pseudocódigo podría ser el siguiente:

Máximo subarreglo: Fuerza bruta

[4] Programming Pearls: Algorithm Design Techniques,Jon Bentley, 1984, Communications of the ACM, 27 (9): 865–873

```
Sea L[x] el valor del elemento en la posición x
Sea intInicio, intFinal los índices del intervalo máximo
Maximo = -∞

Desde i=1 a n
        Desde j=i a n
                Acum =0
                Desde x=i a j
                        Acum = Acum + L[x]
                Si Máximo < Acum
                        Maximo = Acum
                        intInicio = i
                        intFinal  = j

Retornar Maximo
```

Existen varias propuestas de algoritmos utilizando división y conquista. Una opción es dividir recursivamente los elementos en mitades y suponer que el máximo pertenece a un lado o del otro. Esta propuesta no es completa si no considera que el máximo puede estar compuesto de elementos de las mitades. Por lo tanto en el proceso de combinación se debe atender este caso. Podemos llamar "maxPasando" al intervalo que se forma como la intersección del máximo subconjunto que finaliza en el último elemento de la primera mitad más el máximo subconjunto que inicia en el primer elemento de la segunda mitad. Expresando la complejidad mediante una ecuación de recurrencia podemos llegar a $T(n) = 2T(\frac{n}{2}) + O(n)$. Gracias al teorema maestro llegamos a la complejidad final de esta propuesta en O(nlogn) que es superador a la solución por fuerza bruta. Un posible pseudocódigo para la propuesta es:



Máximo subarreglo: División y conquista

```
Sea L[] el vector de n elementos

MaximoSubArreglo(L,inicio,final):
        Si final<inicio
                Retornar inicio
        Si final=inicio
```

```
                Retornar L[inicio]

        Sea mitad el valor intermedio entre inicio y fin

        maxM1 = MaximoSubArreglo(L,inicio,mitad-1)
        maxM2 = MaximoSubArreglo(L,mitad,final)
        maxPasando = MaximiIntermedio(L,inicio,mitad,final)

        Retornar máximo entre maxM1, maxM2 y maxPasando

MaximiIntermedio(L,inicio,mitad,final):
        MaxInicio=0
        AcumInicio=0
        Desde j=mitad-1 hasta inicio
                AcumInicio += L[j]
                Si AcumInicio>MaxInicio
                        MaxInicio=AcumInicio

        MaxFinal=0
        AcumFinal=0
        Desde j=mitad hasta final
                AcumFinal += L[j]
                Si AcumFinal>MaxFinal
                        MaxFinal=AcumInicio

        Retornar MaxInicio+MaxFinal

MaximoSubArreglo(L,1,n):
```

Realizando algunos cambios es posible no solo obtener el máximo posible sino además el intervalo que lo consigue.

Pero aún se puede construir una solución más eficiente. La solución de Kadane utilizando programación dinámica consigue una complejidad lineal temporal. Define el subproblema OPT[x] como el máximo subintervalo que termina en la posición x. El caso base corresponde al primer elemento del arreglo. OPT[1]=L[1]. Al no existir elementos anteriores el máximo subintervalo que termine en la posición 1 solo puede ser el valor del elemento allí ubicado.

Para los siguientes subproblemas nos encontraremos con la expresión general. Un subconjunto máximo que termina en la posición i incluye irremediablemente el valor del elemento en esa posición. Además puede iniciar en esa misma posición o en alguna de las anteriores. Pero no en cualquiera de los anteriores. El nuevo máximo que termina en la posición i si desea serlo debe incluir el intervalo máximo que termina en la posición i-1. Supongamos que el máximo que termina en la posición i-1 comienza en una posición x<=i-1. Si tomamos más o menos elementos de ese subarreglo solo se podrá empeorar el resultado. Entonces es lógico considerar ese mismo intervalo para calcular el óptimo que termina en i. Podemos utilizar la memorización de los resultados previos. Expresamos a este óptimo general como OPT[i]=OPT[i-1]+L[i]

Sin embargo hay un problema en esta formulación. Supongamos que el valor máximo que termina en la posición i-1 es negativo. En este caso no conviene incluirlo como parte del nuevo segmento máximo que termina en i. Cuando esto ocurre, conviene que el nuevo intervalo tenga únicamente el elemento de la posición i. Unificando ambas situaciones, el caso general lo podemos expresar utilizando una ecuación de recurrencia como OPT(i)=max {OPT(i−1),0}+L[i]

Una vez calculado el máximo intervalo que termina en cada una de las posiciones, nos quedamos con el máximo de ellos. Con esto obtenemos la solución al problema planteado.

Partiendo de la lista L={-3, 5, -3, 4, 2, 1, -10, 2, 2, -2, 1, 5} comenzamos calculando OPT[1]=L[1]=-3. El siguiente máximo a calcular es el máximo intervalo que termina en la posición 2: OPT[2]=max{OPT(1),0}+L(2)=max{-3,0}+5=5. Como se observa corresponde al caso donde seleccionar el óptimo anterior es contraproducente por ser negativo. Por lo tanto el máximo que termina en la posición 2 corresponderá al intervalo que solo contiene al elemento en la posición 2. Para la posición 3 tendremos OPT[3]=max{5,0}-3=2. Nos encontramos en el caso en el que conviene incluir el máximo anterior para maximizar el resultado del subconjunto que finaliza en la posición 3. Podemos continuar e ir calculando OPT[4]=6, OPT[5]=8, OPT[6]=9, OPT[7]=-1, OPT[8]=2... Al finalizar el cálculo de todos los óptimos nos quedamos con el mayor de ellos (OPT[6]).

Debemos calcular y almacenar “n” subproblemas. Para cada uno de ellos se requiere operaciones O(1). Finalmente revisar todos los subproblemas calculados (n) y seleccionar el máximo. Globalmente tenemos una complejidad temporal y espacial O(n).

Aún se puede mejorar a nivel espacial el algoritmo. Observar que podemos evitar recorrer al finalizar todos los máximos. Alcanza con registrar e ir actualizando en cada nuevo cálculo del máximo cual es al momento el máximo global (valor y posición). Más aún, con ese cambio no es necesario almacenar todos los óptimos anteriores. Nótese que para calcular el subproblema actual solo se requiere el anterior.

Unificando todas las ideas podemos llegar al pseudocódigo expresado de una manera compacta y elegante:

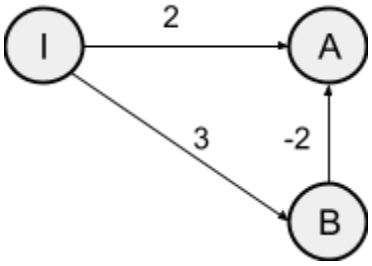
Máximo subarreglo: Programación dinamica - Algoritmo de Kadane

```
MaximoGlobal = L[1]
MaximoLocal = L[1]
IdxFinMaximo = 1
Desde i=2 a n // elementos
        MaximoLocal = max(MaximoLocal,0) + L[i]
        if MaximoLocal > MaximoGlobal
                MaximoGlobal = MaximoLocal
                IdxFinMaximo = i

Retornar MaximoGlobal
```

Esta solución tiene como complejidad temporal O(n) y espacial O(1) siendo la mejor posible. Recuperar el subconjunto final es muy sencillo y se puede hacer en O(n). Requiere ubicarse en la posición de IdxFinMaximo e ir iterando de forma decreciente mientras se acumula el valor de cada uno de los elementos. Al llegar al valor de MaximoGlobal se llega a la posición inicial del intervalo buscado.

5. Camino mínimo (con costos negativos)

Cuando previamente analizamos el problema del camino mínimo en un grafo solicitamos como condición que todos los costos de sus eje fuesen positivos. De esa forma pudimos utilizar el algoritmo Dijkstra correspondiente a la metodología greedy. Pero qué pasa si permitimos la existencia de ejes negativos? Un contraejemplo alcanza para demostrar que no necesariamente el resultado encontrado será óptimo para cualquier instancia del problema. La imagen presentada corresponde a un grafo de 3 nodos y 3 ejes donde aplicar Dijkstra da un resultado no óptimo. Tomemos el nodo "I" como el punto de partida. Podemos ver que tenemos un camino de costo 2 al nodo "A" y de costo 3 al nodo "B" (aquellos accesibles desde la frontera de los nodos ya accedidos). La elección greedy lleva a Dijkstra a seleccionar el nodo "A". En la segunda iteración contamos con dos nodos ya accedidos. Y "B" el único nodo aún no alcanzado. Solo existe un eje a seleccionar para llegar a él. El que se origina de "I" con costo 3. Finaliza la ejecución. Observamos que la solución encontrada no es óptima. Para llegar a "A" es conveniente hacerlo desde "B". El camino I→B→A tiene un costo de 1 comparado con el camino I→A de costo 2.



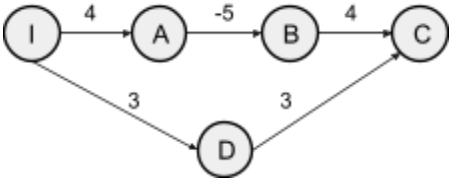
¿Podemos construir un algoritmo óptimo para cualquier instancia que tenga en cuenta costos negativos? Antes de responder la pregunta, formalicemos el problema en el que vamos a trabajar:

Camino mínimo en un grafo
Dado un grafo G=(V,E) de n vértices y m ejes con pesos. Sin ciclos negativos. Partiendo de un vértice x ∈ V deseamos conocer el camino mínimo al resto de los vértices de G.

El motivo por el que solicitamos la ausencia de ciclos tiene una importancia mayúscula. Lo analizaremos más adelante en profundidad.

Una idea que suele aparecer es modificar los costos para permitir usar el algoritmo Dijsktra. Podríamos buscar el menor peso negativo y sumarle a cada eje este valor. Con ese cambio todos los ejes tendrán pesos mayores o iguales a cero. Luego podemos encontrar el camino mínimo utilizando Dijkstra. Sin embargo posiblemente no lleguemos a la solución óptima.

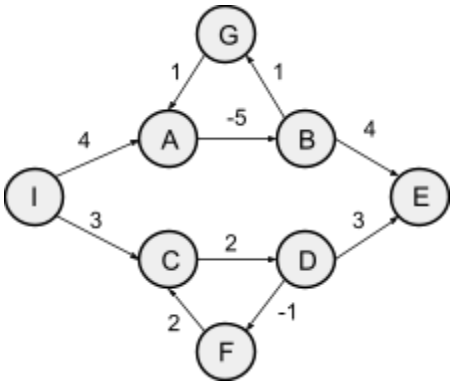
Utilizaremos como contraejemplo de esta idea el grafo de 5 vértices que acompaña este párrafo. El camino mínimo del vértice "I" al vértice "C" es I→A→B→C con costo 7. El menor costo negativo es -5. Siguiendo la propuesta, sumamos 5 a todos los vértices. Si aplicamos el algoritmo Dijkstra, encontraremos que el camino mínimo corresponderá a I→D→C con costo 13 (costo real 6). El costo del camino I→A→B→C (el óptimo) pasará a costar 18 después de la transformación del grafo. Esta idea penaliza los caminos largos frente a los cortos y no es óptimo.



Analicemos el problema desde el punto de vista de un vértice destino "v_d" en particular en un grafo con n vértices. Existen posiblemente múltiples caminos que llevan desde el vértice origen "v_o" a "v_d" (consideramos que tienen al menos un camino entre sí). Uno de ellos será el camino de costo mínimo. Representaremos un camino como $v_o \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \rightarrow v_d$. Los vértices que no corresponden al origen y destino son llamados internos. En un camino pueden no existir vértices internos. Corresponde al uso de un eje en el grafo que une "v_o" con "v_d". Medimos la longitud del camino por la cantidad de ejes que atraviesa. Por lo tanto este camino tiene una longitud de 1. El costo de este camino es el costo del eje. Se puede afirmar que un camino de longitud 0 corresponde al camino entre un vértice y sí mismo. Como no hay ejes en el mismo (consideramos que no existen auto ejes en el grafo) su costo es 0. En el otro extremo, ¿cual es la mayor longitud posible de un camino?

En un camino simple, es decir sin ciclos, se pueden recorrer n-2 vértices interiores diferentes entre un vértice origen y el destino. Por lo tanto tendrá como mucho n-1 ejes. ¿Qué pasa con la longitud y el costo de un camino si permitimos la existencia de ciclos en el mismo? Consideremos un camino simple correspondiente a un camino mínimo entre dos vértices con el agregado de un ciclo que ocurre en algún vértice n_x interior del mismo. Representamos este camino como $v_o \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_x \rightarrow v_{x+1} \rightarrow \ldots \rightarrow v_x \rightarrow \ldots \rightarrow v_d$. Podemos calcular el costo del ciclo como la suma de los costos de los ejes que lo conforman. Pueden ocurrir dos situaciones. En primer lugar, que el costo del ciclo sea cero o positivo. Como estamos buscando el camino mínimo simplemente nos conviene quitarlo para disminuir el costo total del camino. Convertimos $v_x \rightarrow v_{x+1} \rightarrow \ldots \rightarrow v_x$ en v_x. En segundo lugar, el ciclo puede tener un costo negativo. Pero entonces ¿Es realmente el camino

propuesto el del menor costo posible? Podemos pedirle al camino que vuelva a recorrer el ciclo, y nuevamente, y nuevamente... hasta el infinito. Cada vez que lo hagamos el costo total del camino disminuirá (tendiendo a -∞) y su longitud aumenta (tendiendo a +∞). En este caso, existe un camino mínimo, pero al recorrerlo nunca llegaremos a destino. Un camino infinito.



En el grafo que se presenta a continuación se puede observar un camino con un ciclo de "I" a "E" como I→C→(D→F→C→D)→E con costo 11. El ciclo D→F→C→D dentro del mismo tiene un coste de 3. Por lo tanto eliminandolo del camino y llegando a I→C→D→E logra disminuir el costo del mismo a 8. Además, existe otro camino con un ciclo I→A→(B→G→A→B)→E con costo 0. Si calculamos el costo del ciclo B→G→A→B vemos que corresponde al valor negativo -3. Por lo que podemos encontrar un mejor camino mínimo recorriendo nuevamente el ciclo I→A→B→G→A→(B→G→A→B)→E y ahora el costo del mismo es -3. Pero tampoco es el minimo, por que puedo hacerlo mas veces y seguir reduciendo el costo.

Basándonos en lo analizado podemos afirmar que si existe un camino mínimo finito este tendrá como mucho n-1 ejes y excluye la existencia de ciclos negativos en el grafo para lograrlo. Esta característica nos da una primera forma de resolver el problema utilizando búsqueda exhaustiva. Para cada vértice debemos calcular todos los posibles caminos, comenzando por aquellos de longitud 0 y finalizando por los de n-1. Nos quedaremos con el mínimo. Sin embargo, esto termina resultando en una complejidad exponencial.

Existen diversos autores que en diferentes momentos trabajaron sobre el problema[5]. En forma independiente llegaron a diferentes respuestas óptimas y eficientes. Muchas de ellas congruentes entre sí. El algoritmo más conocido que soluciona este problema se conoce como **Bellman-Ford**. Recibe el nombre por las publicaciones e investigaciones de Richard Bellman[6] en 1958 y Lester Jr. Ford[7] en 1956. Utiliza programación dinámica para resolverlo.

[5]On the History of the Shortest Path Problem, Alexander Schrijver, 2012.
[6]On a routing problem, Richard Bellman, 1958, Quarterly of Applied Mathematics. 16: 87–90
[7]Network Flow Theory, Lester R. Jr. Ford, 14 de Agosto de 1956, Paper P-923. Santa Monica, California: RAND Corporation

Concentrémonos por un instante en el camino mínimo a un vértice v_i. Los representamos como $v_o \rightarrow v_1 \rightarrow \ldots \rightarrow v_i$. Este camino tiene una longitud i. Inmediatamente antes de arribar a su destino debe pasar por alguno de los vértices predecesores a v_i (del que salga un eje que llegue a v_i). En este ejemplo corresponde a v_{i-1}. El camino de costo mínimo de v_{i-1} debe tener por lo tanto i-1 de longitud. De igual manera v_{i-1} tendrá un camino mínimo de longitud i-2. Hasta terminar (siempre que el camino sea finito) en v_o con longitud 0.

Siguiendo en el vértice v_i podemos enumerar todos sus caminos de acuerdo a su longitud. Pueden existir caminos de longitud 0 a n-1 como máximo (nuevamente si consideramos los finitos). Para cada longitud podemos quedarnos con el mínimo entre ellas. Para algunas longitudes posiblemente no existan caminos que lleguen al mismo. Diremos que su costo es +∞. Uno de ellos será el mínimo.

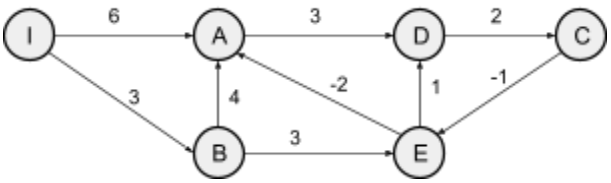
El algoritmo Bellman-Ford utiliza estas dos características. Define el problema a resolver como OPT(v,i) el costo mínimo para llegar al vértice v con un camino de longitud no mayor a i. Cuando la longitud es 0, todos los vértices excepto el de origen tendrán costo +∞. El vértice origen con longitud 0 tiene costo 0. Para el caso general tendremos que el OPT(v,i) corresponde al menor de los costos resultantes de llegar desde un predecesor x de v con costo c_x sumado al costo mínimo de llegar a x en un camino de longitud i-1. En forma resumida:

$$OPT(v_o, 0) = 0$$

$$OPT(v_j, 0) = \infty \; \forall j \neq o$$

$$OPT(v_j, i) = Min_{\forall v_x / (v_x, v_j) \in E} \left\{ OPT(v_x, i-1) + c_x \right\}$$

Siguiendo lo antes analizado, el valor máximo de longitud de un camino mínimo es n-1. Por lo tanto para obtener el mínimo camino para el nodo v tendremos que calcular OPT(v,n-1). Corresponde a un proceso recursivo donde buscaremos el camino mínimo de todos los nodos del grafo. Observaremos que un mismo subproblema vuelve aparecer. Por lo tanto memorizamos los diferentes subproblemas. Tendremos $O(n^2)$ subproblemas. La combinación de cada vértice con los caminos mínimos de longitud de 0 a n-1. Corresponde a una mejora sustancial a la esperada con fuerza bruta (exponencial).



Consideremos el gráfico de la imagen con 8 nodos. Tomaremos como nodo inicial "I". Tenemos 8 nodos. Tenemos que OPT["I",0]=0 y el resto de los nodos OPT[v,0]=+∞. Si nos paramos en el nodo "D" vemos que recibe ejes de "A" y "E". *Por lo tanto para calcular el camino mínimo de longitud i deberemos calcular OPT["D",i]=Min{ OPT["A",i-1] + 3, OPT["E",i-1] + 1 }. De igual forma para calcular el camino mínimo de longitud i a "A" debemos calcular OPT["A",i]=Min{ OPT["I",i-1] + 6, OPT["B",i-1] + 4, OPT["E",i-1] - 2 }. El mismo subproblema OPT["E",i-1] aparece en este caso en los dos cálculos.*

Para construir el pseudocódigo podemos optar entre realizarlo en forma recursiva (de arriba para abajo) o iterativa (de abajo hacia arriba). En este caso desarrollaremos únicamente este último.

Bellman-Ford: Programación dinámica (De abajo hacia arriba)

```
Sea w(v,u) el costo de ir del nodo v al u
Sea OPT[v,i] el menor costo de llegar a v por un camino de longitud i

OPT[0,0]=+∞
Desde v=1 a n-1
        OPT[v,0]=+∞

Desde i=1 a n-1 // max longitud del camino
        Desde v=0 a n-1 // nodo
                OPT[v,i] = OPT[v,i-1]
                Por cada p prededesor de v en Grafo G
                        si OPT[v,i] > OPT[l-1][p] + w(p,v)
                                OPT[l][v] = OPT[l-1][p] + w(p,v)
Retornar OPT
```

Las primeras líneas corresponden al establecimiento de los casos bases: El costo de llegar a cada nodo en un camino de longitud 0. Luego continúa con una doble iteración. La exterior comienza evaluando los caminos de longitud 1 hasta n-1. La interior evalúa para cada nodo el mínimo costo teniendo en cuenta los nodos predecesores al mismo. Al finalizar tendremos en OPT[n-1,x] el camino mínimo desde el origen a al nodo x.

La complejidad temporal y espacial dependerá de las estructuras de datos seleccionadas para representar el grafo y su peso. Es posible seleccionarlas de forma de acceder a cada nodo en O(1) y desde ahí recorrer cada uno de sus predecesores y tener su coste en O(1). Por lo que la iteración interior resuelta O(m). Sumado a que la iteración exterior es O(n), conseguimos una complejidad temporal total de O(mn). En cuanto a la complejidad espacial según el pseudocódigo requerimos almacenar $O(n^2)$ para ir almacenando los subproblemas parciales. No obstante se puede reducir a O(n). Dado que no se requiere almacenar más que los subproblemas de longitud -1 del que se está calculando.

Con la propuesta algorítmica anterior aún no podemos recuperar el camino mínimo. Para hacerlo simplemente podemos agregar por cada nodo cual es el predecesor seleccionado en la iteración interior. Solo hace falta conocer la del último paso. La reconstrucción será retroceder desde el nodo al que queremos llegar hasta el predecesor seleccionado. Se repite este proceso hasta llegar al nodo inicial. Esto no afecta ni la complejidad temporal ni espacial.

i / v	*I*	*A*	*B*	*C*	*D*	*E*
0	*0*	*+∞*	*+∞*	*+∞*	*+∞*	*+∞*
1	*0*	*6 (I)*	*3 (I)*	*+∞*	*+∞*	*+∞*
2	*0*	*6 (I)*	*3 (I)*	*+∞*	*9 (A)*	*6 (B)*
3	*0*	*4 (B)*	*3 (I)*	*11 (D)*	*7 (E)*	*6 (B)*
4	*0*	*4 (B)*	*3 (I)*	*9 (D)*	*7 (E)*	*6 (B)*
5	*0*	*4 (B)*	*3 (I)*	*9 (D)*	*7 (E)*	*6 (B)*

En la siguiente tabla se muestran los resultados de aplicar el algoritmo de Bellman Ford para el grafo de 6 nodos presentado anteriormente. La primera columna indica la iteración que conceptualmente corresponde a la longitud del camino mínimo buscado (igual o menor a este). La fila que corresponde a i=0 muestra el caso base. Solo podemos llegar al nodo inicial I con costo cero. Cada fila restante se apoya con la anterior para recalcular los costos. Por ejemplo en la fila con i=3 para el costo del camino a "A" se usan sus predecesores. I, B y E. Anteriormente el camino mínimo a "A" corresponde a llegar directamente desde "I" con costo 6. Pero existe el camino de longitud 3 I→B→E→A que permite llegar con costo 4. Por lo tanto en esa iteración se actualiza el costo mínimo.

Por último, resta analizar el comportamiento del algoritmo de Bellman-Ford ante la existencia de ciclos negativos. Este algoritmo calcula el costo mínimo progresivamente según longitudes del camino. No toma en cuenta si esa longitud corresponde a un camino simple o un camino utilizando un ciclo. Si nos interesan caminos simples y existen ciclos negativos este algoritmo no nos servirá. Incluso será problemático para el algoritmo si

intentamos reconstruir el camino mínimo, si hay un ciclo negativo mediante el cual llegamos a un determinado nodo. Registrar únicamente el predecesor nos hará ingresar en un loop del que no podrá salir (se podría subsanar a costa de una mayor complejidad temporal).

Podríamos mediante Bellman-Ford encontrar la existencia de un ciclo negativo. Una alternativa es ejecutar el mismo y luego ir nodo por nodo recorriendo a sus predecesores registrando por donde pasa. Si al seguir un camino en un determinado momento se repite un nodo, se verifica la existencia de un ciclo negativo. Pero, una manera mucho más eficiente es simplemente realizar una iteración adicional calculando los caminos mínimos de longitud n y comparándolos con los de n-1. Si al menos uno de los nodos reduce el valor de camino mínimo sabremos de la existencia de lo que buscamos. Recordemos que un camino simple con n nodos tiene como mucho longitud n-1, por lo que encontrar un camino que disminuya entre la longitud n-1 y n implica la existencia de un camino con un ciclo. Recorriendo para atrás el camino a partir de ese nodo podremos encontrar el ciclo.

6. Mínimos cuadrados segmentados

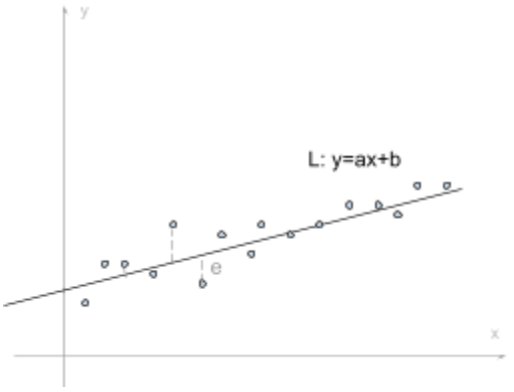
Imaginemos un experimento a realizar. Un objeto es arrojado desde un punto elevado y comienza a rodar en un terreno de diferentes materiales y con diferentes declives. Cada cierta cantidad de tiempo se registra su aceleración. Terminado el experimento se desea construir un modelo matemático que aproxime los resultados mediante segmentos. El modelo debe lidiar con errores de medición y las imperfecciones del terreno. Intentará suavizarlos y simplificarlos. Aun así se espera que sirva para predecir comportamientos futuros. En la construcción de la aproximación pueden existir muchos modelos diferentes. Utilizando más o menos segmentos. A mayor cantidad de segmentos es posible reducir los errores de estimación. También se complejiza la expresión del modelo por lo tanto se pretende utilizar la menor cantidad posible de ellos.

Este problema - ejemplificado en el experimento - es de gran aplicación en diferentes ámbitos. Lo podemos formalizar de la siguiente manera:

Mínimos cuadrados segmentados

Sea un conjunto P de “n” puntos en el plano. Tal que para cada par de puntos $p_i=(x_i,y_i)$ y $p_j=(x_j,y_j)$ se cumple que $x_i > x_j$ si i>j. Queremos aproximar mediante segmentos los puntos de P minimizando el error cometido con la menor cantidad posibles de segmentos

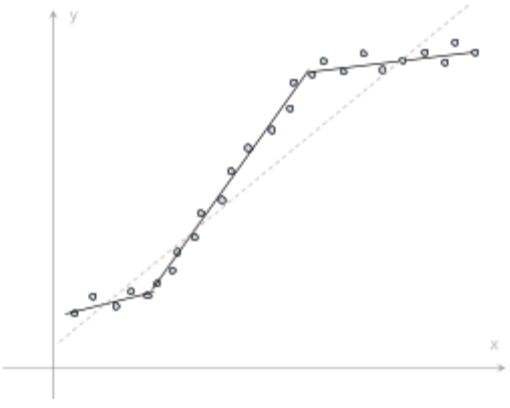
La cuantificación del error requiere algunas definiciones. En primer lugar de la forma de construir la aproximación. Comencemos con un ejemplo básico. Supongamos que únicamente se desea construir la aproximación mediante una única recta o segmento “L”. En ese caso, expresamos la ecuación de la misma como y=ax+b. Debemos encontrar un valor de a y b para minimizar el error. Utilizaremos las siguientes fórmulas para su cálculo:



$$a = \frac{n\sum_i (x_i y_i) - \sum_i (x_i)\sum_i (y_i)}{n\sum_i (x_i^2) - \sum_i (x_i)^2} \text{ y } b = \frac{\sum_i y_i - a\sum_i x_i}{n}$$

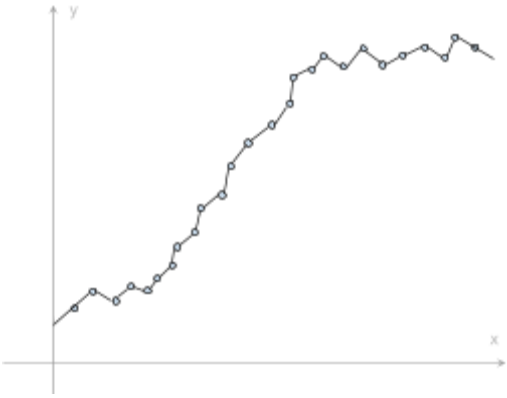
El error cometido en la aproximación corresponde a la suma de los errores entre cada punto y su ubicación en la recta. Se elevará al cuadrado para que los valores negativos no se cancelen con los positivos:

$$E(L,P) = \sum_i^n \left[y_i - (ax_i + b)\right]^2$$



En situaciones complejas solo un segmento es inadecuado. Determinar cuántos segmentos seleccionar y qué puntos incluir en cada uno de ellos nos ocasionará un error global determinado. La pendiente y la ordenada al origen de cada segmento

se calculará utilizando el subconjunto de puntos que lo conforman. De igual manera el error que acota cada punto se debe calcular de acuerdo al segmento al que pertenece.



Podría ser tentador utilizar tantos segmentos como mediciones realizadas. En ese caso el error cometido entre las mediciones y el modelo sería cero. Sin embargo se construiría un modelo complejo que difícilmente permita realizar producciones futuras. Esta situación se conoce como “overfitting”

Resulta evidente que a mayor cantidad de segmentos utilizados el error cometido global será menor. Por lo tanto un modelo que intente reducir la cantidad de estos deberá agregar una penalidad por cada adición. Llamaremos “C” a esta penalidad. Este valor será positivo y su ajuste determinará un equilibrio entre cantidad de segmentos y error cometido. A mayor valor del parámetro tendremos menos segmentos a costo de posiblemente un mayor error. De forma análoga a menor valor del parámetro más segmentos y menor error global.

El primer planteo de solución que realizaremos corresponde a una resolución por fuerza bruta. El primer segmento contendrá al primer punto y puede incluir uno o n-1 puntos restantes. Un segundo segmento comenzará donde terminó el primero y puede contener una cantidad limitada por los que aún no fueron cubiertos aquel. De igual manera se podría agregar un tercero, cuarto, etc. Se puede representar las diferentes posibilidades de segmentación como un vector de n posiciones. La posición i del vector corresponde al i ésimo punto. Un valor de 1 indica que en este finaliza un segmento. En contraposición a un valor de 0 que indica que el segmento que comenzó en algún punto anterior incluye a este. Esa representación nos permite observar que pueden existir $O(2^n)$ combinaciones posibles de segmentos. Para cada una de estas se deben calcular la expresión de la recta de los segmentos, el error cometido y el costo de tener el correspondiente número de segmentos. Observando las expresiones de a, b y E(L,P) se puede ver que estos procesos de cálculo agregan un factor de O(n) en total. Por lo que calcular todas las posibilidades y quedarse con aquella que minimiza el número de segmentos y error cometido será un

proceso de $O(n2^n)$. Estamos frente a un algoritmo que no es eficiente. Buscaremos una mejor solución utilizando programación dinámica.

Existen varias alternativas para resolver este problema. Podemos comenzar por el último punto. El punto "n" irremediablemente pertenece al último segmento. Este comenzará en un punto x anterior. Tendremos entonces el segmento x-n que aportar un error $e_{x,n}$ (calculado utilizando las fórmulas antes vistas). Además agrega un costo adicional de "C" por el agregado de un segmento. Nos quedan ahora n-x puntos. El punto x-1 corresponderá al final de otro segmento que comenzará antes. Se repetirá el proceso hasta llegar al primer punto del conjunto. El problema es que no sabemos a priori cuándo es conveniente realizar la segmentación. Para saber cuándo inicia el último segmento debemos evaluar cada uno de sus posibles puntos iniciales. Luego recursivamente evaluar los errores de los posibles subproblemas resultantes. Estos presentan la misma naturaleza que el anterior. Nos quedaremos con el que nos retorne el menor valor de todos.

Llamaremos OPT[X] al menor costo de segmentar los puntos 1 al x. Deseamos calcular entonces OPT[n]. Podemos jerarquizar los subproblemas al observar que el óptimo de un valor solo depende de los óptimos menores. La expresión corresponde a la siguiente:

$$OPT[x] = Min_{1 \le i \le i} \left(e_{i,x} + C + OPT[i-1] \right)$$

Se debe adicionar el caso base donde tenemos 0 puntos y corresponde a OPT[0]=0

Al analizar la recurrencia se puede observar que en un subproblema hay un proceso O(n) para calcular el mínimo. Los subproblemas pueden volver a aparecer en diferentes cálculos, por lo que memorizarlos reduce el tiempo de cálculo global. Tendremos en total n subproblemas diferentes. En este punto uno podría aventurar que el costo total para calcular el óptimo culmina en $O(n^2)$. Sin embargo es algo aventurado la afirmación. Falta dentro de los cálculos la obtención del error en cada subproblema. Calcular el error entre dos puntos corresponde en sí a un proceso O(n). A esta conclusión arribamos al resolver el problema por fuerza bruta. Por lo tanto esto nos agrega un grado de complejidad mayor. Este cálculo lo podemos hacer dentro de la misma ecuación de recurrencia o pre calcularlo previamente. En cualquier cosa terminaremos calculando para cada par de puntos i,x con i≤x el error. Un total de $O(n^2)$ variantes en O(n) cada uno. En este punto ya podemos afirmar que la complejidad temporal total de nuestro cálculo será $O(n^3)$. En cuanto al

espacio, es buena idea memorizar los cálculos de los errores. Los mismos vuelven a aparecer en algunos subproblemas. Para eso requerimos $O(n^2)$ de espacio. Además de O(n) para almacenar los óptimos.

Se puede expresar finalmente nuestro algoritmo de resolución en forma iterativa de la siguiente forma:

Mínimos cuadrados segmentados: Programación dinámica

```
OPT[0] = 0

Para todo par i,j con i<=j
        Calcular e[i][j]

Desde j=1 a n
        OPTIMO[j] = +∞

        Desde i=1 a n
                segmento = e[i][j] + C + OPT[i-1]

                si OPTIMO[j] > segmento
                        OPTIMO[j] = segmento

Retornar OPT[n]
```

7. Cambio en monedas

Un problema cotidiano cuando realizamos transacciones comerciales mediante dinero físico es el vuelto. Tenemos un monto a pagar, lo realizamos con un valor mayor y debemos recibir la diferencia en billetes o monedas. Es deseable - en la mayoría de los casos - que ese vuelto se realice con la menor cantidad de billetes posibles. Corresponde a una situación simple que también se puede extrapolar a otro ejemplo como dividir carga en paquetes de tamaño predeterminado y otras aplicaciones. Sin más demoras, definamos formalmente el problema:

Cambio en monedas

Contamos un conjunto de monedas de diferente denominación sin restricciones de cantidad. Debemos retornar un valor de X. Queremos minimizar la cantidad de monedas utilizadas para lograrlo.

Definimos como base monetaria $\$=(c_0,c_1,...,c_{n-1})$ como el conjunto ordenado de tamaño "n" de denominaciones de monedas. Por el ordenamiento $c_0<c_1<...<c_{n-1}$. Tomaremos a cada c_i como un valor entero positivo. Además c_0 tendrá como valor 1. Logrando con esto poder dar cambio a cualquier valor entero posible X.

Es probable, que dado la familiaridad con el problema, lo primero que surja como solución corresponde a un algoritmo greedy. La estrategia consiste en seleccionar el ci inmediatamente inferior al monto a dar. Utilizar tantas monedas de esa denominación como sea posible. Luego repetir el proceso hasta que el monto pendiente sea igual a cero. En pseudocódigo lo expresaremos de la siguiente manera:

Cambio en monedas: Greedy

```
Sea  C=X
Sea tot=0
Repetir hasta que C=0
        Buscar ci más grande en $ tal que ci < C
                Calcular r como cuantas veces entra ci en C
                tot = tot + r
                C = C - r*ci

Retornar tot
```

En el peor de los casos utilizaremos cada una de las denominaciones y por lo tanto la solución tiene una complejidad temporal O(n). Si nos interesa conocer la cantidad de monedas requeridas de cada denominación se puede almacenar en una lista y requiere en el peor de los casos también O(n).

En un país se utilizan monedas de 1,5,10,25 y 50 centavos. Debemos dar vuelto de 97 centavos. Según nuestro algoritmo comenzamos entregando 1 moneda de 50. Resta dar 47. Seleccionamos entonces una de 25 En ese punto restan dar 22 centavos. Daremos 2 monedas

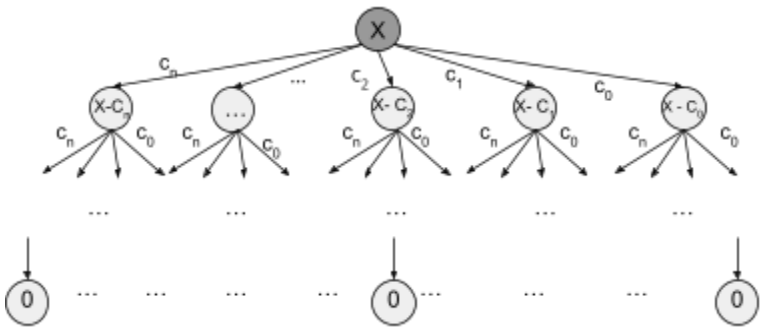
de 20 centavos y finalmente 2 monedas de 1 centavo. En total se darán 6 monedas. Revisando no encontraremos una forma de dar menos cantidad para ese monto y la base monetaria.

Un sistema de monedas $ (o base monetaria) es canónico (también conocido como "standard", "ordenado" o "codicioso") si para todo valor x el número de monedas entregado por el algoritmo greedy es igual al número de monedas de la solución óptima. Basta un contraejemplo para determinar que un sistema no es canónico.

La Primera República portuguesa existió en la historia entre 1910 y 1926. En ese lapso utilizó monedas de 1,2,4,5 y 10 centavos. Si alguien utilizando este sistema requiere cambio de 8 centavos cuenta con varias formas de obtenerlo. Si utilizamos el algoritmo greedy comenzamos con la denominación de 5. El resto - 3 centavos - se resolvería con una moneda de 2 y otra de 1. Por lo tanto el método indicaría la utilización de 3 monedas. Sin embargo, la cantidad mínima (y óptima para esta instancia) corresponde a solo 2 monedas de 4 centavos. Por lo tanto la base monetaria no es canónica.

Para poder afirmar que una base es canónica requiere una demostración matemática. Esto se puede realizar automáticamente y algorítmicamente. En "A polynomial-time algorithm for the change making problem"[8], David Pearson presenta un algoritmo polinomial $O(n^3)$ que lo determina. No ahondaremos en este aspecto. Lo que nos interesa es un algoritmo general que funcione de forma óptima en cualquier base que se nos presente. Nos ayuda la programación dinámica.

Comenzamos descomponiendo el problema en otros menores. Si tenemos que dar un valor de X podemos seleccionar cualquier moneda c_i entre las denominaciones disponibles. Pueden ocurrir 3 situaciones. Queda un resto negativo, igual a cero o positivo. En el primer caso, se debe desechar la elección. De proseguir se estaría otorgando un valor mayor de vuelto al requerido. Si queda igual a cero, termina el problema y se resuelve con esa denominación. Si queda un



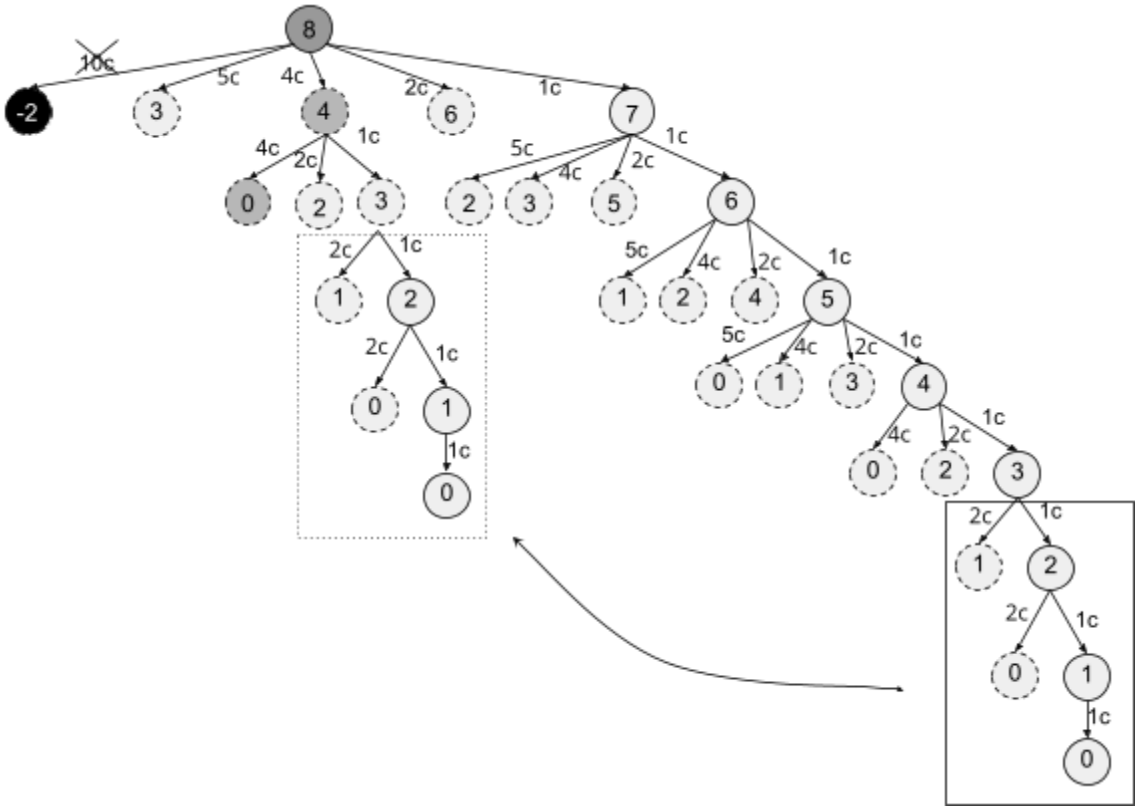
[8]A polynomial-time algorithm for the change making problem, David Pearson, 1994, Computer Science Technical Reports, Cornell University

valor positivo, significa que aún queda vuelto por dar ($X-c_i$) y este es un nuevo subproblema que se debe resolver recursivamente. Por cada elección se suma una moneda a las requeridas.

Por fuerza bruta podemos recorrer todas las opciones. Se puede expresar mediante un árbol de jerarquía de subproblemas este recorrido de opciones y elecciones. Parado en un nodo en particular se desplegará por debajo un subárbol. Las hojas del mismo corresponden al subproblema "0" o finalizar de entregar todo el cambio. El camino mínimo desde un nodo a una hoja corresponde a la cantidad mínima de monedas que puedo dar para resolver este subproblema.

En el análisis de ese árbol se podrá observar que varios subproblemas vuelven a aparecer. Los subproblemas repetidos - no importa cómo llegue a ellos - se resuelven de la misma manera y su mínimo corresponde al mismo. Podemos memorizarlos y evitar su recálculo.

A continuación se visualiza el árbol de subproblemas del ejemplo de la república Portuguesa resumido. Iniciamos con un cambio a dar de 8 centavos. En este primer subproblemas podemos utilizar 1,2,4 y 5 centavos. Se evita dar 10 al superar el monto. En negro se muestra el nodo inválido que expresa esta situación. Con cualquiera de las opciones válidas suma 1 en la cantidad de monedas necesarias (aumentamos en 1 la profundidad del árbol). La elección de una moneda genera un nuevo subproblema con nuevas opciones y un nuevo resto. Al tener que dar cambio de 8 y seleccionar una moneda de 2 llegamos al subproblema de tener que dar cambio de 6. Mismo subproblema al que arribamos al tener que dar cambio de 7 y seleccionar una moneda de 1. Los subproblemas que vuelven a aparecer se muestran en línea punteada. Basta con resolverlos una única vez y su resultado se puede utilizar para el resto. Un ejemplo de esta reutilización se muestra para el subproblema de dar vuelto de 3. El camino óptimo se muestra en gris oscuro y expandido.



Buscaremos representar la propuesta de solución mediante una relación de recurrencia. Comenzaremos con la definición del subproblema OPT[v] como la cantidad mínima de monedas a dar para dar como cambio el valor v. El caso base corresponde al tener que dar cambio de cero que requiere cero monedas. Además agregamos, para evitar dar más del cambio necesario un valor infinito para valores de cambio v negativos.

$$OPT(v < 0) = \infty$$

$$OPT(0) = 0$$

En el caso general, utilizaremos una moneda y seleccionaremos el subproblema que minimice la cantidad necesaria de monedas para dar cambio de "v" menos la denominación de la moneda c_i seleccionada. Lo expresamos como:

$$OPT(v) = 1 + Min_{c \in \$}(OPT(v - c))$$

Una vez que contamos con la relación de recurrencia se procede a construir el pseudocódigo. En este caso, utilizando el enfoque de abajo hacia arriba.

Cambio en monedas: Programación dinámica (De abajo hacia arriba)

```
OPT[0]=0
Desde i=1 a x
        minimo = +∞
        Desde j=1 a n
                resto = i – C[j]
                si resto >= 0 y minimo > OPT[resto]
                        minimo = OPT[resto]
        OPT[i] = 1 + minimo

Retornar OPT[x]
```

En la propuesta de resolución se evite el subproblema con valor v negativo verificando antes de llamarlo si el resto es mayor o igual a 0. Por lo demás, es un código sencillo y limpio. Claramente se puede ver que la complejidad temporal corresponde a la resultante de las iteraciones anidadas. Lo expresaremos como O(nX). El tiempo de ejecución depende tanto de la cantidad de denominaciones diferentes en la base monetaria como de la cantidad de cambio a dar. El primer parámetro es indicativo del tamaño de la instancia. Sin embargo el segundo - cambio a dar - corresponde a un valor numérico de la instancia (y no del tamaño de la misma). Cuanto ese valor sea mayor, lo mismo ocurrirá con el tiempo de ejecución. Cuando esto ocurre no podemos hablar de un algoritmo polinomial. Se conoce a este como **pseudopolinomial**.

En cuanto a la complejidad espacial requiere de igual manera memorizar un resultado por cada valor de cambio entre 1 y X. Por lo tanto, también es pseudo polinomial. No se conoce un algoritmo polinomial que resuelva este problema. Por lo que no se puede afirmar que corresponde a un algoritmo eficiente o bueno.

Finalmente, si deseamos poder reconstruir qué denominaciones se debe seleccionar para llegar a la cantidad mínima de monedas se debe almacenar por cada subproblema memorizado cual fue la denominación seleccionada. De esa forma, similar al problema de fraccionamiento del aceite de oliva, podremos presentar las elecciones realizadas.

8. El problema de la mochila y los elementos indivisibles.

Analizamos dentro de la metodología greedy el problema de la mochila. En la versión planteada era posible dividir cada uno de los elementos en fracciones según nuestra conveniencia. Utilizando una elección codiciosa, frente a una jerarquía de subproblemas que tiene en cuenta el valor unitario de cada elemento, se obtiene una solución óptima. Sin embargo, no siempre se puede partir los elementos en partes.

Consideremos una mochila, contenedor, transporte de carga o cualquier otro recipiente que pueda soportar una carga máxima. Además consideremos elementos no divisibles. Algunos ejemplos: libros, muebles, automóviles, entre muchos otros. Se mantiene el objetivo de maximizar la ganancia o el valor de los elementos seleccionados y transportados. Formalizando el problema:

Problema de la Mochila
Contamos con una mochila con una capacidad de K kilos y queremos introducir dentro de ellas un subconjunto del conjunto E de "n" elementos con el objetivo de maximizar la ganancia. Cada elemento i tiene un peso de c_i kilos y un valor de v_i. Los elementos no pueden ser divididos y el peso total seleccionado no puede superar la capacidad de la mochila.

Ante el incendio de la biblioteca de la abadía, Guillermo de Baskerville puede salvar sólo un subconjunto de obras manuscritas. Entre sus brazos y ropajes puede transportar como mucho 15 kilogramos a través de los pasillos laberínticos. Ante sí tiene libros con un valor y un peso determinado. Desea salvar el máximo valor posible sin superar su capacidad de carga. Las diferentes obras son (nombre,peso,valor): ("A",3,14), ("B",6,7), ("C",9,4), ("D",2,6), ("E",8,9), ("F",5,8), ("G",10,15), ("H",15, 18), ("I",8,7), ("J",9,6). Podría elegir "H" y llevarse 18 de valor. Pero, por otro lado, podría llevarse "G" y "F" combinando 23 de valor. Fuera de su posibilidad es llevar "H" y "D" dado que su peso en conjunto supera el máximo posible.

En el problema de la mochila cada elemento es único y no repetible. En ese aspecto es similar al problema de máxima cantidad de intervalos. Podríamos entonces analizar uno a uno los elementos de forma jerárquica. Cada subproblema corresponde a la elección o no de un determinado elemento. Su inclusión nos otorga cierta ganancia. Sin embargo existe una diferencia fundamental entre estos dos problemas. En el problema de los intervalos, al

resolver un subproblema, no importa las elecciones anteriores para llegar al resultado óptimo. Siempre será el mismo el camino de máxima ganancia a partir de él. Sin embargo, para el problema de la mochila se debe considerar la restricción de peso máximo que se puede cargar. Seleccionar un elemento disminuye la capacidad remanente de la mochila. Por lo tanto se puede llegar a analizar un mismo elemento con diferentes "tamaños de mochila". Estos pueden ir desde cero a la capacidad máxima K.

Guillermo seleccionó varios libros y aún puede cargar 12 kilos. Debe optar cuáles llevar entre los libros A,B y C. Puede el "A" y "B" combinando entre ellos 21 de valor. La otra opción "A" y "C" la descarta al ser de menor valor que la alternativa. No puede llevar los tres pues superan su capacidad. Luego de pensarlo brevemente, vuelve a realizar la selección. Llegando nuevamente a analizar entre A,B y C pero esta vez pudiendo cargar solo 5 kilos. En esta nueva situación opta por llevar el libro "A". Cualquiera de las otras alternativas superan su capacidad de carga.

Con el fin de solucionar el problema de forma óptima mediante programación dinámica debemos agregar una nueva dimensión a la definición del problema. Estas dos dimensiones lo complejizan y llevan a la necesidad de analizar una cantidad mayor de subproblemas derivados de estos. Mantenemos como parámetro del subproblema los elementos analizados. Agregaremos la capacidad remanente de la mochila. Llamaremos OPT(i,p) al subproblema de obtener el máximo valor posible entre los primeros "i" elementos con una capacidad remanente de la mochila de p kilos.

Existen dos casos base, uno para cada dimensión del problema. En primer lugar el óptimo para cualquier subproblema si la capacidad remanente es cero kilos es igual a cero. Si la mochila está llena, no importa el peso y valor de los elementos restantes, ninguno de ellos se podrá agregar y por lo tanto la ganancia máxima para estos subproblemas es cero. En segundo lugar si la cantidad de elementos restantes a evaluar es igual a cero el óptimo del subproblema también es cero. No importa el lugar disponible en la mochila, sin elementos restantes no se puede aumentar la ganancia obtenida. Podemos agregar una condición adicional, el subproblema correspondiente a si la cantidad remanente de la mochila es negativa. Corresponde a la situación de cargar más peso del posible. Para evitar esta situación se puede establecer que la ganancia en este caso es un número inmensamente negativo. De esa forma al buscar el máximo evitaremos estas situaciones. Esta última se

puede evitar al momento de construir el pseudocódigo simplemente evitando asignar elementos a la mochila si no entran en ella.

El caso general corresponde a evaluar o no la inclusión del i-ésimo elemento teniendo en cuenta el espacio disponible en la mochila. Si no se incluye entonces suma cero al valor acumulado. Se buscará la máxima ganancia posible con los i-1 elementos restantes y la misma capacidad disponible. Si se incluye el elemento entonces suma su valor al máximo valor posible entre los i-1 restantes en una capacidad disminuida por el peso del recién ingresado. La decisión sobre cuál de las dos opciones seleccionar corresponde a ver cual es mayor entre estas.

Uniendo todos los casos, la relación de recurrencia la podemos expresar como:

$$OPT(i,0) = 0$$
$$OPT(0,p) = 0$$
$$OPT(i,p<0) = -\infty$$
$$OPT(i,p) = Max\left\{OPT(i-1,p-c_i) + v_i;\ OPT(i-1,p)\right\}$$

El resultado del máximo valor a obtener del problema general, corresponde a la solución del subproblema OPT(n,K). Correspondiente a evaluar a todos los elementos con una mochila inicial vacía.

Guillermo desea determinar cuál es la selección óptima de los libros. Son n=10 libros y K=15 es el peso máximo que puede llevar con él. Evaluará inicialmente el décimo libro ("J",9,6). El problema general corresponde a OPT(10,15). Si no lo selecciona, debe calcular de los 9 libros restantes con aún la mochila llena. Es decir, debe calcular OPT(9,15). La alternativa es seleccionar el libro adicionando una ganancia de 6 y restando 9 a la disponibilidad de peso en la mochila. Buscará maximizar la ganancia entre los 9 libros restantes y 6 Kilos remanentes. Es decir que debe calcular 6 + OPT(9,6). Para cada uno de los subproblemas deberá resolver recursivamente sus subproblemas. Cuando tiene el resultado de las dos opciones escoge aquella que maximiza sus ganancias.

En ocasiones un mismo subproblema volverá a ocurrir, por lo tanto los memorizamos para evitar su recálculo. A continuación se puede observar el pseudocódigo expresado de abajo hacia arriba. Se agrega un control para evitar pasarse en lo cargado en la mochila.

Problema de la mochila: Programación dinámica (De abajo hacia arriba)

Sea v[i] la ganancia de incluir el elemento i
Sea c[i] el peso del elemento i
Sea OPT[i,p] la máxima ganancia a obtener al seleccionar algunos de los i-esimos primer elementos sin sobrepasar p kilos

```
Desde i=0 a K
        OPT[i][0] = 0

Desde p=0 a K
        OPT[0][p] = 0

Desde i=1 a n // elementos
        Desde p=1 a K // pesos
                Si c[i]<=p
                        enOptimo = v[i] + OPT[i-1,p-c[i]]
                Sino
                        enOptimo = -∞
                noEnOptimo = OPT[i-1,p]

                si enOptimo > noEnOptimo
                        OPT[l][p] = enOptimo
                sino
                        OPT[l][p] = noEnOptimo

Retornar OPT[n,K]
```

Modificando levemente el pseudocódigo podemos reconstruir las elecciones realizadas para la maximización de la ganancia. La manera es similar a la utilizada en el problema de máxima cantidad de intervalos. Por ese motivo no se volverá a explicar aquí.

Una cuestión no debe pasar desapercibida, en la resolución de abajo hacia arriba se calculan todos los subproblemas posibles. Incluso algunos que tal vez para la instancia particular del problema nunca se utilizan. Sin embargo, esto no termina impactando a la complejidad final del algoritmo.

En la siguiente tabla se muestra el paso a paso de la resolución utilizando el algoritmo propuesto para la instancia de ejemplo. El máximo valor que puede llevar Guillermo suma 35

que se obtiene en el subproblema OPT[J,15]. Para determinar el valor de cada celda se utilizan 2 subproblemas: incluir o no el elemento. Se muestra en detalle el cálculo de OPT[F,10]. Se cuenta con 10 kilogramos. Si no se selecciona se verifica al valor de OPT[E,10] = 21. Por el contrario si se selecciona se adicionan al valor del elemento 8 el resultado de OPT[E,5]=20. Se observa que incluir el elemento nos otorga una mayor ganancia que no realizarlo. El cálculo se realiza fila por fila aprovechando siempre lo calculado en la anterior.

i / p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	14	14	14	14	14	14	14	14	14	14	14	14	14
B	0	0	0	14	14	14	14	14	14	21	21	21	21	21	21	21
C	0	0	0	14	14	14	14	14	14	21	21	21	21	21	21	21
D	0	0	6	14	14	20	20	20	20	21	21	27	27	27	27	27
E	0	0	6	14	14	20	20	20	20	21	21	27	27	29	29	29
F	0	0	6	14	14	20	20	20	22	22	28	28	28	29	29	29
G	0	0	6	14	14	20	20	20	20	20	28	28	28	29	35	35
H	0	0	6	14	14	20	20	20	20	20	28	28	28	29	35	35
I	0	0	6	14	14	20	20	20	20	20	28	28	28	29	35	35
J	0	0	6	14	14	20	20	20	20	20	28	28	28	29	35	**35**

i	v	c
A	14	3
B	7	6
C	4	9
D	6	2
E	9	8
F	8	5
G	15	10
H	18	15
I	7	8
J	6	9

El algoritmo de resolución tiene 2 iteraciones, una por elemento y la otra por unidad de capacidad. Dentro de esas iteraciones se realizan operaciones O(1). Por lo tanto podemos afirmar que la complejidad temporal final es O(nK). Nos encontramos nuevamente frente a un algoritmo pseudo polinomial. La complejidad está atada a la capacidad de la mochila. Si esta es considerable en función a la cantidad de elementos o si la unidad de medida unitaria otorga mucha granularidad el tiempo de ejecución crecerá considerablemente. En igual medida la complejidad espacial consiste en memorizar cada uno de los subproblemas. Corresponde también a O(nK). Si no deseamos reconstruir el camino de elecciones y solo obtener el máximo valor, podemos reducirlo a O(n) almacenando

únicamente las últimas 2 filas en el cálculo. Para reconstruir el camino se debe almacenar en cada subproblema si se terminó o no incluyendo el elemento. Reconstruir será un proceso O(K) que comenzará en el subproblema general.

Otro problema conocido se llama Subset-Sum y la estrategia de solución recién elaborada es fácilmente trasladable a este. El enunciado del problema es el siguiente:

Problema de la Mochila
Contamos con un conjunto E de "n". Cada elemento "i" tiene un peso asociado w_i. Se desea seleccionar el subconjunto de elementos de "E" con el major peso combinado posible que no supere el valor W de peso máximo.

Este problema lo podemos pensar también como una mochila que queremos llenar. No contamos con un valor por elemento. Aunque el peso mismo puede tomar el papel de este. Vemos la mochila como el subconjunto de elementos a seleccionar. Un subconjunto de peso x es menor en valor a un subconjunto de peso y>x. Un elemento que pesa "a", es más valioso que un elemento que pesa "b" con "b"<"a". Tomaremos entonces el peso del elemento como su valor. Nada además de esto es necesario.

Podemos expresar este cambio con una simple modificación en el caso general de la relación de recurrencia:

$$OPT(i,0) = 0$$
$$OPT(0,p) = 0$$
$$OPT(i,p < 0) = -\infty$$
$$OPT(i,p) = Max\left\{OPT(i-1,p-w_i) + w_i;\ OPT(i-1,p)\right\}$$

Luego resolver y analizar el problema resulta a nivel algorítmico es prácticamente equivalente al problema de la mochila