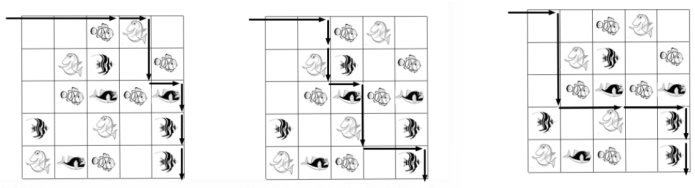
Facultad de Ciencias, UNAM
Análisis de Algoritmos
Tarea 3

Rubí Rojas Tania Michelle

11 de diciembre de 2020

1. Un pescador está sobre un océano rectangular. El valor del pez en el punto (i, j) está dado por un arreglo A de dimensión $2(n \times m)$. Diseña un algoritmo que calcule el máximo valor de pescados que un pescador puede atrapar en un camino desde la esquina superior izquierda a la esquina inferior derecha. El pescador sólo puede moverse hacia abajo o hacia la derecha, como se ilustra en la siguiente figura:



SOLUCIÓN: Queremos encontrar el máximo valor de peces que podemos obtener al realizar únicamente movimiéntos hacia la derecha y hacia abajo. Así, nuestro objetivo será modificar la matriz A de tal manera que en cada sub-instancia del problema tengamos en $A[i][j]$ el valor máximo de pescaditos que el pescador puede atrapar desde $A[0][0]$ hasta $A[i][j]$ realizando sólo dos tipos de movimiéntos. De esta forma, al tener cada celda de A actualizada como describimos anteriormente, al final bastará regresar el valor de $A[n-1][m-1]$, pues ésta contendrá el valor máximo de pescaditos que deseamos.

Así, consideremos los siguientes puntos:

- El caso base sería que la celda $A[0][0]$ no puede ser actualizada. Esto se debe a que el valor en esa celda es el máximo que el pescador puede obtener si va desde $A[0][0]$ a $A[0][0]$ (no se mueve).
- Para el primer renglón de A, bastará con ir obteniéndo la suma acumulada de la celda inmediatamente a la izquierda de la actual. Si pensamos que el pescador se está moviéndo siempre hacia la derecha en la primer fila, es decir, el pescador sólo se mueve sobre $A[0][j]$ con $j \in \{0, \ldots, m-1\}$; entonces

$$A[0][j] = A[0][j-1] + A[0][j]$$

Esto se debe a que al sumar el valor de la celda actual que tiene el valor del pez en ese lugar, junto con el máximo que puede obtener el pescador moviéndose solamente hacia la derecha desde $M[0][0]$ hasta una celda previa que ya contiene el máximo, entonces obtenemos el valor deseado.

Demostración. Inducción sobre la columna j.

- Caso base. Cuando $j = 0$, entonces $A[0][0]$ tiene ya el máximo valor que el pescador puede obtener (por el punto anterior).
- Hipótesis de inducción. Supongamos que se cumple para k, es decir, supongamos que $A[0][k]$ tiene el valor máximo que el pescador puede obtener moviéndose siempre a la derecha sobre el primer renglón hasta ese punto.

- Paso inductivo. Queremos probar que se cumple para $A[0][k+1]$. Entonces

$$A[0][k+1] = A[0][k] + A[0][k+1] \qquad \text{por definición}$$

Por hipótesis de inducción tenemos que $A[0][k]$ tiene el valor máximo que el pescador puede obtener moviéndose a la derecha desde $A[0][0]$ hasta $A[0][k]$. Así, para obtener el máximo hasta la celda que le sigue a la derecha, basta con sumarle el valor que tiene actualmente. Por lo tanto, si actualizamos $A[0][k+1]$ con $A[0][k] + A[0][k+1]$ entonces obtendremos el valor máximo hasta este punto.

Por lo tanto, si hacemos que $A[0][j] = A[0][j-1] + A[0][j]$, entonces obtendremos en cada celda del primer renglón el valor máximo que el pescador puede conseguir moviéndose siempre a la derecha sobre el primer renglón.

□

- De manera análoga, tenemos el caso para la primer columna de A. Será suficiente con ir obteniéndo la suma acumulada de la celda inmediatamente arriba de la actual. Si pensamos que el pescador siempre se está moviéndo hacia abajo en la primer columna, es decir, el pescador sólo se mueve sobre $A[i][0]$ con $i \in \{0, 1, \ldots, n-1\}$; entonces

$$A[i][0] = A[i-1][0] + A[i][0]$$

Esto se debe a que al sumar el valor de la celda actual que tiene el valor del pez en ese lugar, junto con el máximo que puede obtener el pescador moviéndose solamente hacia abajo desde $A[0][0]$ hasta una celda previa que ya contiene el máximo, entonces obtenemos el valor deseado.

Demostración. Inducción sobre el renglón i.

- Caso base. Cuando $i = 0$, entonces $A[0][0]$ tiene ya el máximo valor que el pescador puede obtener (por el primer punto).
- Hipótesis de inducción. Supongamos que se cumple para k, es decir, supongamos que se cumple que $A[k][0]$ tiene el valor máximo que el pescador puede obtener moviéndose siempre hacia abajo sobre la primer columna hasta ese punto.
- Paso inductivo. Queremos probar que se cumple para $A[k+1][0]$. Entonces

$$A[k+1][0] = A[k][0] + A[k+1][0] \qquad \text{por definición}$$

Por hipótesis de inducción tenemos que $A[k][0]$ tiene el valor máximo que el pescador puede obtener moviéndose a la derecha desde $A[0][0]$ hasta $A[k][0]$. Así, para obtener el máximo hasta la celda que le sigue abajo, basta con sumarle el valor que tiene actualmente. Por lo tanto, si actualizamos $A[k+1][0]$ con $A[k][0] + A[k+1][0]$, entonces obtendremos el valor máximo hasta este punto.

Por lo tanto, si hacemos que $A[i][0] = A[i-1][0] + A[i][0]$, entonces obtendremos en cada celda de la primer columna el valor máximo que el pescador puede conseguir moviéndose siempre hacia abajo sobre la primer columna.

□

- Para actualizar la parte de la matriz A que falta, entonces es suficiente considerar el valor actual de cada celda que contiene el valor del pez en ese lugar del océano junto con el máximo valor en las celdas inmediatamente arriba y a la izquierda de la actual, es decir,

$$A[i][j] = A[i][j] + max\,\{A[i-1][j], A[i][j-1]\}$$

Esto lo haremos para las entradas (i, j) con $i \in \{1, \ldots, n-1\}, j \in \{1, \ldots, j-1\}$.

Mostraremos que gracias a esta expresión, cada celda tendrá el valor máximo que el pescador puede conseguir moviéndose desde $M[0][0]$ hasta $M[i][j]$ con movimiéntos únicamente hacia la derecha o hacia abajo.

Demostración. Inducción sobre el renglón i.

- Caso base. Como $i = 1$, entonces debemos mostrar que

$$A[1][j] = A[1][j] + max\{A[0][j], A[1][j-1]\}$$

Demostración. Inducción sobre la columna j.

- Caso base. Como $j = 1$, entonces

$$A[1][1] = A[1][1] + max\{A[0][1], A[1][0]\}$$

Así, tendremos que $A[1][1]$ contiene el máximo valor que el pescador puede conseguir moviéndose solamente a la derecha o hacia abajo desde $A[0][0]$ y hasta $A[1][1]$. Por las demostraciones anteriores sabemos que $A[0][1]$ ya tiene el máximo valor que puede conseguir el pescador moviéndose solamente hacia la derecha de la posición inicial y $A[1][0]$ ya tiene el máximo valor que puede conseguir el pescador moviéndose solamente hacia abajo desde $A[0][0]$. Entonces basta con sumarle el valor de la celda actual $A[1][1]$ al valor más grande de las dos opciones que se tienen por la restricción de los movimientos para obtener el valor máximo deseado. Por lo tanto, este caso se cumple.

- Hipótesis de inducción. Supongamos que se cumple para k, es decir, supongamos que $A[1][k]$ tiene el valor máximo que el pescador puede obtener moviéndose con las restricciones dadas hasta ese punto desde $A[0][0]$.
- Paso inductivo. Queremos probar que se cumple para $A[1][k+1]$. Entonces

$$A[1][k+1] = A[1][k+1] + max\{A[0][k+1], A[1][k]\} \qquad \text{por definición}$$

Por el paso inductivo del primer renglón tenemos que $A[0][k+1]$ ya tiene el máximo valor que puede obtener el pescador moviéndose solamente hacia la derecha. Luego, por hipótesis de inducción, $A[1][k]$ ya tiene el máximo valor que el pescador puede obtener moviéndose con las restricciones dadas hasta ese punto desde $A[0][0]$. Así, al considerar el valor máximo de estos dos óptimos, se obtiene el óptimo al moverse a la derecha o hacia abajo desde $A[0][0]$; y solo falta sumar el valor actual de $A[1][k+1]$ para obtener el máximo valor deseado.

□

- Hipótesis de inducción. Supongamos que se cumple para k, es decir, supongamos que $A[k][j]$ tiene el valor máximo que el pescador puede obtener moviéndose hacia abajo o hacia la derecha desde $A[0][0]$ hasta $A[k][j]$.
- Paso inductivo. Queremos mostrar que se cumple para $k+1$. Entonces

$$A[k+1][j] = A[k+1][j] + max\{A[k][j], A[k][j-1]\}$$

Por hipótesis de inducción tenemos que $A[k][j], A[k][j-1]$ tienen ya el valor máximo que el pescador puede obtener moviéndose hacia abajo o hacia la derecha desde $A[0][0]$ hasta $A[k][j]$ y $A[k][j-1]$, respectivamente. Así, al considerar el valor máximo de estos dos, se obtiene el óptimo deseado. Así, sumando el valor actual de $A[k+1][j]$ obtenemos el valor máximo buscado.

Por lo tanto, cada entrada (i, j) de la matriz A con $i \in \{1, \ldots, n-1\}$ y $j \in \{1, \ldots, m-1\}$ tendrá el valor máximo que el pescador puede obtener moviéndose solamente a la derecha o hacia abajo desde el punto inicial $A[0][0]$ si

$$A[i][j] = A[i][j] + max\{A[i-1][j], A[i][j-1]\}$$

□

Esto lo podemos traducir como el siguiente algoritmo:

1. Definimos $i = j = 0$.
2. Mientras j sea menor que m, hacemos

   $$A[0][j] = A[0][j-1] + A[0][j]$$

   Aumentamos en una unidad a j.
3. Mientras i sea menor que n, hacemos

   $$A[i][0] = A[i-1][0] + A[i][0]$$

   Aumentamos en una unidad a i.
4. Definimos $i = j = 1$.
5. Mientras i sea menor que n, hacemos
   - Mientras j sea menor que m, hacemos

     $$A[i][j] = A[i][j] + max\{A[i-1][j], A[i][j-1]\}$$

     Aumentamos en una unidad a j.

   Aumentamos en una unidad a i.
6. Regresamos $A[n-1][m-1]$.

De esta forma, evitamos volver a calcular los mismos subproblemas usando memoización con el arreglo $A[n][m]$. Ahora bien, los ciclos `while` de los pasos 2 y 3 nos toman $\Theta(m)$ y $\Theta(n)$, respectivamente; pues estamos recorriendo una sola vez un renglón y una columna. Y el ciclo `while` del paso 5 nos toma $\Theta(nm)$, ya que estamos recorriendo lo que queda de la matriz A (entrada por entrada). Por lo tanto, la complejidad total del algoritmo es de $\Theta(nm)$.

2. Dados dos árboles generadores T y R de una gráfica G. Muestra cómo encontrar la secuencia más corta de árboles generadores $T_0, T_1, \ldots, T_k$ tal que $T_0 = T, T_k = R$, y cada árbol T_i difiere del anterior T_{i-1} agregando y borrando una arista.

   Solución: Sean T y R dos árboles generadores de G. Veamos que si S es un árbol generador de la gráfica G, entonces podemos llegar a otro árbol generador U eliminándo y agregando una arista.

   Demostración. Sea G una gráfica con un conjunto de vértices V(G) y un conjunto de aristas $E(G)$. Si el número de aristas es menor que $|V(G)| - 1$, entonces G no es conexa, por lo que no podemos tener un árbol generador (pues el número mínimo de aristas que tiene un árbol es $|V(G)| - 1$). De esta forma, supongamos que estamos trabajando sobre una gráfica G conexa. Entonces, tenemos dos casos:

   - El número de aristas de G es exactamente $|V(G)| - 1$, es decir, G tiene el mínimo número de aristas para ser conexa. Por la definición de árbol, entonces la gráfica G es el árbol generador y no hay que quitarle ni agregarle más aristas.
   - El número de aristas de G es mayor a $|V(G)| - 1$. Sea S el árbol generador de G. Por definición, S tiene $|V(G)| - 1$ aristas, y como G tiene más aristas que S entonces existe al menos un arista que está en G pero no está en S. Como $|V(G)| - 1$ es el mínimo número de aristas para que G sea conexo, entonces si ponemos una arista en S que está en G pero no está en S, entonces se forma un ciclo. De esta forma, podemos eliminar cualquier arista del ciclo (que no sea la que agregamos) y el árbol seguirá siendo conexo (pues los vértices extremos siguen estándo conectados). Por lo tanto, hay otro árbol generador distinto a S tal que éstos sólo difieren en una arista.

   □

Ahora bien, queremos encontrar la secuencia más corta entre los árboles generadores T y R de G. Así, podemos realizar el siguiente algoritmo:

1. Tomamos el conjunto de aristas que están en R pero no están en T, es decir, tomaremos el conjunto $E(R) - E(T)$.
2. Como ya sabemos que podemos ir de un árbol generador a otro difiriéndo sólo en una arista, entonces hacemos lo siguiente: tomamos una arista del conjunto $E(R) - E(T)$ y la ponemos en T. Como esa arista no estaba en T, entonces se forma un ciclo en este árbol. Sabemos que este ciclo no está en R, pues si esto sucediera entonces R no sería un árbol generador. Entonces, como tenemos este ciclo, al menos una de las aristas dentro del ciclo no pertenece a R. Tomamos cualquiera de las aristas dentro del ciclo que no está en R y la eliminamos de T. Así, obtenemos un árbol T_i que se parece una arista más a R.
3. Si $T_i = R$, entonces terminamos.
4. En caso contrario, volvemos a repetir el algoritmo desde el paso 2 sin volver a tomar en cuenta las aristas en el conjunto $E(R) - E(T)$ que ya hemos usado.

Mostraremos que este algoritmo efectivamente regresa la mínima secuencia entre los árboles generadores T y R.

Demostración. Procedemos por contradicción. Sea $C = \{T_0, T_1, \ldots, T_k\}$ la secuencia más corta entre los árboles generadores T y R obtenida por el algoritmo anterior. Supongamos que existe otra secuencia aún más corta C' entre los árboles generadores T y R. Esto quiere decir que C tiene al menos un árbol T_i que no está en C', pero como todos los árboles T_j dentro de C fueron generados difiriéndo en una arista, entonces esto significa que al menos hay dos árboles T_{i-i} y T_i en C que difieren en dos aristas (pues ya demostramos que podemos ir de un árbol generador a otro difiriéndo solo en una arista); lo que implica que ya no estamos borrando e insertando una sola arista. Contradicción.

Por lo tanto, efectivamente C es la secuencia más corta entre los subárboles T y R.

□

Ahora bien, la complejidad de este algoritmo depende mucho de como tratemos a nuestra gráfica. Sabemos que encontrar el conjunto $E(R) - E(T)$ nos toma tiempo lineal (pues debemos recorrer cada una de las gráficas y luego encontrar el conjunto diferencia). Luego, en el peor de los casos, nos tardamos $O(|V|^2)$ para el resto de los pasos del algoritmo, pues debemos estar revisándo los árboles generados n veces.

3. Sea G una gráfica con n vértices. Un subconjunto S de los vértices de G es independiente si cualesquiera dos elementos de S no son adyacentes. En general, el problema de encontrar el conjunto independiente de una gráfica es un problema $NP-$completo. Pero en algunos casos, este problema puede resolverse eficientemente. Sea T un árbol con raíz con n vértices. Cada nodo $v \in T$ tiene asociado un peso $w(v)$. Utilizándo programación dinámica, encuentre un algoritmo de tiempo lineal para encontrar el conjunto independiente de T de peso máximo.

   Solución: Queremos encontrar el conjunto de vértices en el árbol T tal que ninguno de sus vértices es adyacente a otro y que maximiza la suma de los pesos de los nodos, es decir, el conjunto V de vértices tal que para ningún par de ellos existe alguna arista que los conecte y que maximiza la suma de los pesos de los nodos. Para ello, caractetizaremos la substructura óptima del problema para resolverlo usando programación dinámica. Consideraremos cada subárbol de T como un subproblema, y por lo tanto nuestro objetivo es relacionar la solución de todo el árbol con las soluciones de los subárboles. De esta forma, notemos que si denotamos a la raíz del árbol como un vértice arbitrario v, entonces hay dos posibilidades:

   - El vértice v forma parte del conjunto independiente de peso máximo del subárbol T_i. Esto implica que los hijos de v no pueden pertenecer al conjunto deseado, pues sabemos que los vértices no deben de estar conectados por alguna arista. Denotaremos a este conjunto como $S(v)$. Así, $S(v)$ consiste en

v más la unión de los conjuntos independientes de peso máximo de los subárboles de los nietos de v. Esto se puede calcular como

$$S(v) = w(v) + \sum_{x \in \text{nietos}(v)} S(x)$$

Demostración. Inducción sobre v.

- Caso base. Si v es una hoja, entonces el tamaño del conjunto independiente de peso máximo es justamente el peso de v, pues éste no tiene subárboles (y por ende, no tiene nietos).
- Hipótesis de inducción.Supongamos que se cumple para k, es decir, supongamos que

$$S(k) = w(k) + \sum_{x \in \text{nietos}(k)} S(x)$$

  tiene el tamaño del conjunto independiente de peso máximo.
- Paso inductivo. Queremos probar que se cumple para $k+1$. Entonces

$$\begin{aligned} S(k+1) &= S(k) + w(u) \\ &= \left(w(k) + \sum_{x \in \text{nietos}(k)} S(x) \right) + w(u) \end{aligned}$$

  Como u fue agregado al conjunto S, eso significa que u es una hoja o es el nieto de un vértice v. Por hipótesis de inducción, sabemos que $S(k)$ tiene el tamaño del conjunto independiente de peso máximo. Así, al sumarle $w(u)$ a este valor, obtenemos que $S(k+1)$ tiene el tamaño del conjunto independiente de peso máximo.

□

- El vértice v no forma parte del conjunto independiente de peso máximo del subárbol T_i. Entonces este conjunto es simplemente la unión de los conjuntos independientes de peso máximo de los subárboles de los hijos de v. Esto se puede calcular como

$$S'(v) = \sum_{x \in \text{hijos}(v)} S'(x)$$

  Demostración. Inducción sobre v.

  - Caso base. Si v es una hoja, entonces el tamaño del conjunto independiente de peso máximo es 0, pues nuestro vértice no tiene hijos.
  - Hipótesis de inducción.Supongamos que se cumple para k, es decir, supongamos que

$$S'(k) = \sum_{x \in \text{hijos}(k)} S'(x)$$

    tiene el tamaño del conjunto independiente de peso máximo.
  - Paso inductivo. Queremos probar que se cumple para $k+1$. Entonces

$$\begin{aligned} S'(k+1) &= S'(k) + w(u) \\ &= \left(\sum_{x \in \text{hijos}(k)} S'(x) \right) + w(u) \end{aligned}$$

    Como u fue agregado al conjunto S', eso significa que u es una hoja o el hijo de un vértice v. Por hipótesis de inducción, sabemos que $S'(k)$ tiene el tamaño del conjunto independiente de

peso máximo. Así, al sumarle $w(u)$ a este valor, obtenemos que $S'(k+1)$ tiene el tamaño del conjunto independiente de peso máximo.

□

Con estas dos observaciones es posible escribir nuestro paso recursivo: Sea $T(v)$ el tamaño del conjunto independiente de peso máximo en el subárbol cuya raíz es v, entonces

$$T(v) = \begin{cases} 0 & \text{si } v \text{ es hoja y } v \in S' \\ w(v) & \text{si } v \text{ es hoja y } v \in S \\ max\left\{w(v) + \sum_{x \in \text{ nietos de } v} T(x), \sum_{x \in \text{ hijos}(v)} T(x)\right\} & \text{otherwise} \end{cases}$$

Mostraremos que $T(v)$ es correcto.

Demostración. Inducción sobre el vértice v.

- Caso base. Si v es una hoja, entonces tenemos dos casos:
  - $v \in S$. Por el caso base de una de las demostraciones anteriores, este caso se cumple.
  - $v \in S'$. Por el caso base de una de las demostraciones anteriores, este caso se cumple.
- Hipótesis de inducción. Supongamos que esta función se cumple para k, es decir, supongamos que

$$T(k) = max\left\{w(k) + \sum_{x \in \text{ nietos de } k} T(x), \sum_{x \in \text{ hijos de } k} T(x)\right\}$$

  contiene el tamaño del conjunto independiente de peso máximo en el subárbol cuya raíz es k.
- Paso inductivo. Veamos que esto se cumple para $k+1$. Entonces

$$\begin{aligned} T(k+1) &= T(k) + 1 \\ &= T(k) + w(u) \\ &= max\left\{w(k) + \sum_{x \in \text{ nietos de } k} T(x), \sum_{x \in \text{ hijos de } k} T(x)\right\} + w(u) && \text{H.I} \\ &= max\left\{\left(w(k) + \sum_{x \in \text{ nietos de } kT(x)}\right) + w(u), \left(\sum_{x \in \text{ hijos de } k} T(x)\right) + w(u)\right\} \end{aligned}$$

  Estos son los pasos inductivos de las demostraciones anteriores, por lo que las expresiones dentro del `max` ya contienen el conjunto independiente de peso máximo al considerar (o no) el vértice $k+1$ como parte del conjunto; por lo que al tomar el el máximo de estos dos óptimos obtenemos el conjunto independiente de peso máximo del subárbol cuya raíz es $k+1$.

Por lo tanto, el algoritmo es correcto.

□

Esto lo podemos traducir al siguiente algoritmo usando programación dinámica:

- Definimos a r como la raíz del árbol T.
- Para todos los nodos en T usando un recorrido `postorder`:
  - Si v es una hoja, entonces $S(v) = w(v)$ y $S'(v) = 0$.
  - Sino, $S(v) = w(v) + \sum_{x \in \text{nietos}(v)} S(x)$ y $S'(v) = \sum_{x \in \text{hijos}(v)} S'(x)\}$
- Regresamos $max\{S(r), S'(r)\}$

Usamos `postorder` pues así garantizamos que visitamos a los hijos antes que al padre.

Ahora bien, veamos que este algoritmo es de tiempo lineal. Notemos que debemos calcular el peso máximo del conjunto independiente que tiene cada vértice del árbol como raíz una sola vez (usando un recorrido postorder, el cual nos toma $O(|V|)$). En las siguientes veces que se desee consultar el valor, este ya estará almacenado en el arreglo de programación dinámica T y podemos indexarlo y regresar su valor en tiempo constante. Así, calcularlos toma tiempo $O(|V|)$. Luego, para cada vértice, el algoritmo sólo mira a sus hijos y a sus nietos; por lo que cada vértice v se mira sólo tres veces: cuando lo miramos como raíz, cuando lo miramos como hijo y cuando lo miramos como nieto. Así, a cada nodo lo miramos un número constante de veces y sólamente en una ocasión (la primera) es cuando hacemos la llamada para calcular su valor. Por lo tanto, la complejidad total de nuestro algoritmo es $O(|V|)$.

4. Mientras caminas por la playa encuentras un cofre de tesoros. El cofre contiene n tesoros con pesos $w_i, \ldots, w_n$ y valores $v_1, \ldots, v_n$. Desafortunadamente, sólo tienes una maleta que sólo tiene capacidad de carga M. Afortunadamente, los tesoros se pueden romper si es necesario. Por ejemplo, la tercera parte de un tesoro i tiene peso $\frac{w_i}{3}$ y valor $\frac{v_i}{3}$.

   - Describe un algoritmo voraz de tiempo $\theta(n \log n)$ que resuelve este problema.

     Solución: Como podemos romper los tesoros, entonces una buena idea para atacar este problema es calcular el valor del costo unitario de cada uno de los n tesoros que encontremos, esto con el objetivo de poder seleccionar aquellos tesoros que nos aportan mayor valor y tienen un menor peso. Luego, ordenamos los tesoros, en orden descendente, de acuerdo a su valor de costo unitario. Lo ordenamos de esta forma para ir tomándo siempre a los tesoros que nos aportan mayor valor en tiempo constante. Después, iremos metiéndo a la mochila los tesoros enteros que están al inicio de la lista ordenada si el peso de éstos no es mayor a la capacidad de la mochila. En otro caso, partimos el tesoro actual que intentamos meter para quedarnos sólo con una fracción del mismo. De esta forma, metemos a la mochila todos los tesoros con mayor costo unitario que podamos, y cuando ya no sea posible meter más tesoros enteros, simplemente cortamos el tesoro final en la fracción que nos falta. Terminamos cuando llenamos completamente la mochila.

     Así, esto se traduce al siguiente algoritmo:

     1. Obtenemos el valor del costo unitario de cada uno de los n tesoros, es decir, calculamos

        $$u_i = \frac{v_i}{w_i} \qquad \text{con } i \in \{1, 2, \ldots, n\}$$

     2. Ordenamos los n tesoros, en orden descendente, de acuerdo a su valor de costo unitario u_i. Así, supongamos que esta nueva lista es de la forma

        $$l = \{t_1, t_2, \ldots, t_n\}$$

     3. Sean M la capacidad de la mochila y c la capacidad actual de la misma en un momento dado. Escogemos el tesoro t_i con $i \in \{1, 2, \ldots, n\}$ de la lista l.
        - Si $c \geq w_i$, entonces metemos al tesoro t_i a la mochila (pues podemos cargar con la totalidad del tesoro) y la capacidad actual es actualizada como $c = c - w_i$.
        - Si $c < w_i$, entonces tomamos únicamente una parte del tesoro t_i. Esta fracción será lo que nos falte para terminar de llenar la mochila, es decir, tomamos la parte $\frac{c_i}{w_i}$ del tesoro. Terminamos.

     Este algoritmo funciona porque siempre garantizamos que metemos a la mochila los tesoros con mayor costo unitario, lo que implica que siempre metemos a aquellos tesoros que nos aportan mayor valor con menos peso. Metemos a los primeros k tesoros con mayor costo unitario (de esta forma, obtenemos la mayor ganancia en tesoros con el menor peso) hasta llenar la mochila. Los $k-1$ tesoros que elegimos se meten enteros a la mochila, mientras que el $k-$ésimo elemento termina de llenar la mochila con la parte fraccional que falta. De esta forma, como tomamos únicamente a los que nos aportan mayor costo unitario, logramos maximizar el valor de los tesoros que metemos a la mochila.

Ahora bien, calcular el valor de costo unitario para cada uno de los n tesoros nos toma tiempo lineal, ya que recorremos toda la lista de los n tesoros. Ordenar la lista usando `HeapSort` nos toma $\Theta(n \log n)$. Luego, en el peor de los casos, se verifica que cada uno de los n tesoros entre en la mochila, por lo que esto nos tomará tiempo lineal, pues recorremos toda la lista de los tesoros. Por lo tanto, la complejidad total del algoritmo es $\Theta(n \log n)$.

- ¿Se puede mejorar el tiempo de ejecución de tu algoritmo a $\theta(n)$? Si es un no, explica por qué; si es un sí, menciona el cambio.

  SOLUCIÓN: Sí es posible mejorar la complejidad, y para lograrlo seguiremos el siguiente algoritmo

  1. Obtenemos el valor del costo unitario de los n tesoros, esta vez sin ordenarlos. Así, supongamos que el conjunto de valores unitarios es
  $$\rho = \{\frac{v_1}{w_1}, \ldots, \frac{v_n}{w_n}\}$$
  2. Encontramos la mediana del conjunto ρ usando el algoritmo SELECT, visto en clase. Recordemos que SELECT funciona de la siguiente manera:
     a) Separar a los n elementos en $\lfloor \frac{n}{5} \rfloor$ grupos de 5 elementos cada uno (a lo más un grupo de tamaño $n \pmod 5$).
     b) Encontrar la mediana de cada uno de los $\lfloor \frac{n}{5} \rfloor$ grupos (los cuales están ordenados por inserción y tomamos al elemento de enmedio; si el grupo tiene un número par de elementos, tomamos a la mayor de las medianas)
     c) Usamos SELECT recursivamente para encontrar la mediana x del conjunto de $\lceil \frac{n}{5} \rceil$ medianas encontradas en el paso anterior (si el conjunto de medianas es de longitud par, entonces tomamos la más pequeña).
     d) Luego, dividimos el conjunto de entrada con elemento pivote x (la mediana de las medianas) usando el algoritmo PARTITION (del algoritmo QuickSort). Sea k el número de elementos en la parte inferior de la partición, de manera que x es el $k-$ésimo elemento y $n-k$ es el número de elementos en la parte superior.
     e) Si $i = k$, entonces regresamos a x. En otro caso, usamos SELECT recursivamente para encontrar el $i-$ésimo elemento más pequeño en el lado inferior si $i < k$, o el $(i-k)-$ésimo elemento más pequeño en el lado superior si $i > k$.

     Definimos a m como la mediana del conjunto ρ.
  3. Creamos tres nuevos conjuntos C_1, C_2, C_3 tal que
     - C_1 tendrá los costos unitarios cuyos valores sean estrictamente mayores a la mediana m, es decir,
     $$C_1 = \{\frac{v_i}{w_i} \mid \frac{v_i}{w_i} > m, 1 \leq i \leq n\} \qquad \text{con } W_1 = \sum_{i \in C_1} w_1$$
     Este conjunto se refiere a los tesoros que nos conviene tener, pues tienen mayor valor por unidad.
     - C_2 tendrá los costos unitarios cuyos valores sean igual a la mediana m, es decir,
     $$C_2 = \{\frac{v_i}{w_i} \mid \frac{v_i}{w_i} = m, 1 \leq i \leq n\} \qquad \text{con } W_2 = \sum_{i \in C_2} w_i$$
     - C_3 tendrá los costos unitarios cuyos valores sean estrictamente menores a la mediana m, es decir,
     $$C_3 = \{\frac{v_i}{w_i} \mid \frac{v_i}{w_i} < m, 1 \leq i \leq n\} \qquad W_3 = \sum_{i \in C_3} w_i$$
  4. De esta forma,
     - Si $W_1 > M$, es decir, si la suma de los costos unitarios de los tesoros en el conjunto C_1 es mayor que la capacidad de la mochila, entonces aplicamos recursivamente el algoritmo sobre el conjunto C_1 para quedarnos con los tesoros que valen más.

- Sino, mientras no excedamos la capacidad de la mochila y C_2 no sea vacío, entonces vamos metiéndo los tesoros del conjunto C_2 (cuyo costo unitario es igual a la mediana m) a la mochila.
  - Si se llena la mochila, entonces regresamos los los tesoros correspondientes al conjunto C_1 y a los tesoros que logramos agregar del conjunto C_2.
  - Sino, reducimos la capacidad de la mochila en $W_1 + W_2$, pues estamos considerándo los tesoros correspondientes a C_1 y C_2. Como aún hay espacio en la mochila, entonces hacemos recursión sobre el conjunto C_3; lo que hará que regresemos los tesoros en los conjuntos C_1, C_2 y los que logremos agregar a la mochila del conjunto C_3 durante esa llamda recursiva.

Este algoritmo funciona porque gracias a la obtención de la mediana podemos encontrar a aquellos tesoros cuyo costo unitario es el mayor (éstos son aquellos que nos importan). De esta forma, podemos garantizar que siempre vamos agregándo a la mochila aquellos tesoros que nos conviene meter a la mochila. Vamos revisándo conjunto por conjunto para poder seleccionar los tesoros que queremos, los cuales son aquellos que son mayores a la mediana (si logramos tomar todo el conjunto, entonces los siguientes que nos convienen son los que son iguales a la mediana, y así sucesivamente). Gracias a las llamadas recursivas logramos ir metiéndo los tesoros que más nos convienen en ese momento, esto mientras haya lugar en la mochila. Así, podemos obtener los tesoros que maximizan la ganancia.
Ahora bien, obtener el valor del costo unitario de cada uno de los n tesoros nos toma $\Theta(n)$, pues siemore debemos recorrer toda la lista de tesoros. Encontrar la mediana de un conjunto de tamaño n nos toma $O(n)$, por lo visto en clase. Crear los nuevos tres conjuntos nos toma $\Theta(n)$, pues basta con recorrer el arreglo para poder colocar a cada uno de los valores en el nuevo conjunto que le corresponde. Luego, el paso recursivo considera a lo más la mitad del conjunto actual (por cómo están separados los conjuntos), por lo que la función de recurrencia sería

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

donde $O(n)$ corresponde a las observaciones anteriores. Así, esta recurrencia podemos resolverla utilizándo el Teorema Maestro:
Como nuestra expresión es de la forma

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k$$

donde a, b, c, k son constantes, entonces podemos solucionar la recurrencia como sigue:
Si $a = 1, b = 2, c = 1, k = 1$, se cumple que

$$a < b^k = 1 < 2^k$$

por lo que $T(n) \in \Theta(n^k)$ y obtenemos la solución $\Theta(n^1) = \Theta(n)$. Así, la complejidad total de nuestro algoritmo es $\Theta(n)$.

5. Un grupo de n personas quiere comprar un ramo de m flores, cada flor va a tener un costo asociado c_i, pero si un cliente ha comprado c veces entonces el dueño del puesto le vende la flor i en costo $(c+1)v_i$. Diseña un algoritmo que en tiempo $O(m \log m)$ minimice el costo de comprar todas las flores.

   Solución: Sabemos que entre más flores compre una persona más grande será el costo que tendrá que pagar, por lo que el objetivo es minimizar el número máximo de flores que cualquier persona compra, esto para que las compras se distribuyan lo más uniformemente posible. Además, debemos optimizar el órden en que compramos las flores, por lo que tenemos que comprar primero las flores más caras (el costo adicional que necesitamos pagar después es lineal en c_i). Así, podemos intuir que primero debemos ordenar las flores (en órden descendente) de acuerdo a su costo c_i; y después distribuirlas uniformemente entre las personas que compran las más caras primero. Siguiéndo esta idea, planteamos el siguiente algoritmo:

1. Ordenamos las m flores, en órden descendente, de acuerdo a su costo asociado c_i. Supongamos que la lista ordenada es de la siguiente forma:

$$C = \{c_1, c_2, \ldots, c_m\}$$

2. Definimos una variable `cc=0`, la cual se encargará de llevar el valor del costo de comprar todas las flores. Además, definimos una variable `ad = 0`, la cual se encargará de indicar cuál es el costo adicional que debemos agregarle al costo de nuestras flores actuales.
   - Si el número de flores es menor o igual al número de personas, entonces regresamos la suma del costo de todas las flores, es decir,

$$cc = c_1 + c_2 + \ldots + c_m$$

   - En otro caso, mientras $i = 0$ sea menor que m hacemos:
     - Sumamos el valor del costo de las primeras m flores dentro de la lista C (pues el valor adicional es 0); luego sumamos el valor del costo de las siguientes m flores, pero multiplicándo el costo de cada una de las flores por $[ad + 1]$ con $i \in \{1, 2, \ldots\}$, y así sucesivamente. Esto se puede modelar con lo siguiente:

```
cc += (ad + 1) * C[i]
```

     - Si $i + 1$ (mód n) es igual a cero, esto implica que hemos agregado ya las n flores y debemos aumentar en una unidad nuestro costo adicional.
     - Incrementamos en una unidad a nuestra variable i.

- Regresamos `cc`.

Este algoritmo funciona porque siempre garantizamos que las flores con un costo asociado más grande sean las que tengan un valor adicional menor. Como tenemos una lista C con los costos ordenados, entonces podemos acceder a las flores con mayor valor en tiempo constante. El caso trivial es cuando el número de flores m es menor o igual que el número de personas, ya que podemos darle a lo más a cada persona una flor, así que nuestro costo adicional siempre es 0 y esto resulta en sumar los costos de todas las flores m. Luego, cuando $m > n$ lo que hacemos es darle las primeras n flores a cada una de las n personas, donde el costo adicional es de 0 (pues es la primer flor). Posteriormente, le damos las siguientes n flores a cada una de las n personas, lo que implica que nuestro costo adicional aumentará en una unidad y serán más caras que su precio original (pero como está ordenada la lista C entonces sabemos que la suma de estos nuevos costos no es más grande que si hubiéramos multiplicado por este costo adicional actual a la suma de las m flores anteriores). Repetimos este proceso hasta que las flores se hayan terminado, sumándo siempre bloques de n flores con un costo adicional dado y en caso de que lleguemos al punto donde el número de flores que nos quedan sin repartir es menor que el número de personas, entonces simplemente las distribuimos entre las personas que podamos, pero sin perder de vista el valor adicional que les corresponde en ese momento. Así, como las estamos distribuyéndo uniformemente entre las personas, siguiéndo esta idea podemos garantizar que obtenemos el mínimo costo de comprar todas las flores.

Ahora bien, sabemos que ordenar las flores nos toma $\Theta(m \log m)$. Obtener la suma de los costos de las m flores nos toma $\Theta(m)$, pues siempre debemos recorrer todo el arreglo C. Por lo tanto, la complejidad total del algoritmo es de $\Theta(m \log m)$.

6. Sean $k, n \in \mathbb{N}$. El problema de los huevos, es el siguiente: tenemos un edificio con n pisos y k huevos. Sabemos que hay un piso f tal que si dejamos caer un huevo desde el piso f, se estrellará. Si dejamos caer un huevo desde un piso r tal que $r < f$, el huevo no se estrellará, y si dejamos caer el huevo desde un piso $r \geq f$, el huevo se estrellará (es posible que $f = 1$, en cuyo caso, el huevo siempre se estrellará. Si $f = n+1$, el huevo nunca se estrellará). **Una vez que un huevo se estrella, no lo podemos usar nuevamente**. Si disponemos de k huevos, ¿cuál es el menor número de experimentos (dejar caer un huevo) que se tienen que hacer para determinar a f? Sea $E(k, n)$ el mínimo número de experimentos que tiene que hacer para determinar a f.

a) Pruebe que $E(1, n) = n$.

Demostración. Sin pérdida de generalidad, comenzamos a lanzar el huevo desde el piso 1. Notemos que debemos lanzar el huevo piso por piso, ya que si lo hacemos de otra manera (digamos, de dos en dos), como sólo tenemos un huevo, entonces no tenemos forma de saber en qué piso se rompe.

Por otro lado, en el peor caso, tenemos que $f \geq n$, por lo que tenemos dos casos:

- $f = n$. Como subimos piso por piso, entonces hasta que llegamos al piso n sabemos que el huevo se rompe. Por lo tanto, $E(1, n) = n$.
- $f > n$. Como subimos piso por piso, al llegar al piso n tenemos que el huevo no se rompe (el problema nos dice que en este caso el huevo nunca se romperá). Pero, como ya no podemos subir más pisos, entonces es suficiente con llegar al $n-$ésimo piso para saber que $f > n$. Por lo tanto, $E(1, n) = n$.

□

b) Encuentre una recurrencia para $E(k, n)$. Utilice programación dinámica para encontrar $E(k, n)$. ¿Qué tan rápido es su algoritmo?

Solución: Debemos tener en cuenta las siguientes observaciones

- Si un huevo se ha roto, entonces ya no podemos volver a utilizarlo.
- Un huevo se puede reutilizar si y sólo si sobrevive a la caída.
- Si un huevo se rompe en el piso t, entonces éste también se romperá en los todos los $t + 1$ pisos, es decir, el huevo se rompe también en los pisos que están por encima de t.
- Si un huevo sobrevive a la caída en el piso t, entonces éste también sobrevivirá a la caída de todos los $t - 1$ pisos, es decir, el huevo sobrevive también en los pisos que están por debajo de t.

De esta forma, si tenemos k huevos y n pisos, podemos considerar dos casos al momento de que un huevo es lanzado desde el piso t, con $t \in \{1, 2, \ldots, f\}$:

- El huevo se rompe. Esto implica que todos los pisos que están encima de t rompen el huevo, pero los que están por debajo del piso t debemos considerárlos aún (pues el piso seguro más alto está en algún lugar por debajo de t). Así, el problema se reduce a $k-1$ huevos y $t-1$ pisos por revisar.
- El huevo no se rompe. Esto implica que todos los pisos por debajo del piso t tampoco rompen el huevo, pero los que están por arriba del piso t debemos considerárlos aún (pues el piso seguro más alto está por arriba de t). Así, el problema se reduce a k huevos y $n - t$ pisos por revisar.

Notemos que necesitamos minimizar el número de experimentos en el peor de los casos, así que tomamos el máximo valor de estas dos situaciones, y seleccionamos el piso que produce el mínimo número de experimentos.

De esta forma, si $E(k, n)$ es la función que calcula el número mínimo de experimentos para determinar a f, entonces

$$E(k,n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ n & \text{si } k = 1 \\ 1 + min\left\{max\left\{E(k-1, t-1), E(k, n-t)\right\}\right\} & \text{otherwise} \end{cases}$$

con $t \in \{1, 2, \ldots, f\}$.

Esta función es correcta pues $E(k-1, t-1)$ y $E(k, n-t)$ con $t \in \{1, 2, \ldots, f\}$ tienen el número mínimo de experimentos para encontrar a f considerándo si un huevo en el piso t se rompe (o no); esto por las justificaciones dadas anteriormente. De esta forma, al obtener el máximo de éstos dos valores (pues debemos considerar el peor de los casos) y tomando luego el mínimo de estos peores casos obtenemos que efectivamente

$$min\left\{max\left\{E(k-1, t-1), E(k, n-t)\right\}\right\}$$

calcula el número mínimo de experimentos para encontrar a f. Finalmente, le sumamos una unidad a esta expresión pues también estamos realizando un experimento desde un piso en particular.

Esto lo podemos implementar, usando programación dinámica, de la siguiente manera:

1. Definimos la matriz $E[k+1][n+1]$ y la variable $i = 1$.
2. Mientras i sea menor o igual a k, hacemos:
   - $E[i][1] = 1$
   - $E[i][0] = 0$
   - Aumentamos en una unidad a i.
3. Definimos la variable $j = 1$.
4. Mientras j sea menor o igual a n, hacemos
   - $E[1][j] = j$
5. Definimos las variables $i = j = 2$, $t = 1$ y $m = 0$.
6. Mientras i sea menor o igual a k, hacemos:
   - Mientras j sea menor o igual a n, hacemos:
     - $E[i][j] = \infty$
     - Mientras t sea menor o igual a j
       - $m = 1 + min\{max\{E[i-1][t-1], E[i][j-t]\}\}$
       - Si m es menor que $E[i][j]$, entonces $E[i][j] = m$.
       - Incrementamos en una unidad a t.
     - Incrementamos en una unidad a j.
   - Incrementamos en una unidad a i.
7. Regresamos $E[k, n]$.

De esta forma, evitamos volver a calcular los mismos subproblemas memoizando la función $E(k, n)$ con una matriz $E[k, n]$. Ahora bien, los ciclos de los pasos 2. y 3. nos toman $\Theta(k)$ y $\Theta(n)$, respectivamente; pues recorren la matriz E en una sola dirección. El ciclo en el paso 6. nos toma $O(kn^2)$, pues usamos un ciclo k veces y un ciclo $n \times n$ veces para cada huevo. Por lo tanto, la complejidad total del algoritmo es $O(kn^2)$.

7. Construye el árbol de Huffman para codificar el siguiente texto:

"La rabia es como el picante. Una pizca te despierta, pero en exceso te adormece"

SOLUCIÓN: Primero, ignorándo mayúsculas, vamos a crear una tabla de frecuencias para los símbolos y letras en nuestro texto

símbolo	frecuencia
b	1
u	1
x	1
z	1
.	1
,	1
d	2
l	2
m	2
n	3
s	3
i	4
p	4
r	4
t	4
c	5
o	5
a	8
e	13
␣	14

Figura 1: Tabla de frecuencias ordenada

Luego, realizaremos las actualizaciones de la tabla de frecuencias:

símbolo	frecuencia
bu	2
xz	2
.,	2
d	2
l	2
m	2
n	3
s	3
i	4
p	4
r	4
t	4
c	5
o	5
a	8
e	13
␣	14

Tabla 3: Unimos los símbolos `.` y `,`

símbolo	frecuencia
.,	2
d	2
l	2
m	2
n	3
s	3
buxz	4
i	4
p	4
r	4
t	4
c	5
o	5
a	8
e	13
␣	14

Tabla 4: Unimos los símbolos `bu` y `xz`

símbolo	frecuencia
x	1
z	1
.	1
,	1
bu	2
d	2
l	2
m	2
n	3
s	3
i	4
p	4
r	4
t	4
c	5
o	5
a	8
e	13
␣	14

Tabla 1: Unimos los símbolos `b` y `u`

símbolo	frecuencia
.	1
,	1
bu	2
xz	2
d	2
l	2
m	2
n	3
s	3
i	4
p	4
r	4
t	4
c	5
o	5
a	8
e	13
␣	14

Tabla 2: Unimos los símbolos `x` y `z`

símbolo	frecuencia
l	2
m	2
n	3
s	3
buxz	4
.,d	4
i	4
p	4
r	4
t	4
c	5
o	5
a	8
e	13
␣	14

Tabla 5: Unimos los símbolos `.,` y `d`

símbolo	frecuencia
n	3
s	3
buxz	4
.,d	4
lm	4
i	4
p	4
r	4
t	4
c	5
o	5
a	8
e	13
␣	14

Tabla 6: Unimos los símbolos `l` y `m`

símbolo	frecuencia
buxz	4
.,d	4
lm	4
i	4
p	4
r	4
t	4
c	5
o	5
ns	6
a	8
e	13
␣	14

Tabla 7: Unimos los símbolos `n` y `s`

símbolo	frecuencia
p	4
r	4
t	4
c	5
o	5
ns	6
buxz.,d	8
lmi	8
a	8
e	13
␣	14

Tabla 9: Unimos los símbolos `lm` y `i`

símbolo	frecuencia
o	5
ns	6
buxz.,d	8
lmi	8
pr	8
a	8
tc	9
e	13
␣	14

Tabla 11: Unimos los símbolos `t` y `c`

símbolo	frecuencia
lm	4
i	4
p	4
r	4
t	4
c	5
o	5
ns	6
buxz.,d	8
a	8
e	13
␣	14

Tabla 8: Unimos los símbolos `buxz` y `.,d`

símbolo	frecuencia
t	4
c	5
o	5
ns	6
buxz.,d	8
lmi	8
pr	8
a	8
e	13
␣	14

Tabla 10: Unimos los símbolos `p` y `r`

símbolo	frecuencia
buxz.,d	8
lmi	8
pr	8
a	8
tc	9
ons	11
e	13
␣	14

Tabla 12: Unimos los símbolos `o` y `ns`

símbolo	frecuencia
pr	8
a	8
tc	9
ons	11
e	13
␣	14
buxz.,dlmi	16

Tabla 13: Unimos los símbolos `buxz.,d` y `lmi`

símbolo	frecuencia
e	13
␣	14
buxz.,dlmi	16
pra	16
tcons	20

Tabla 15: Unimos los símbolos `tc` y `ons`

símbolo	frecuencia
tcons	20
e ␣	27
buxz.,dlmipra	32

Tabla 17: Unimos los símbolos `buxz.,dlmi` y `pra`

buxz·,dlmipransotce ␣	79

Tabla 19: Unimos los símbolos `buxz.,dlmipra` y `tconse ␣`

símbolo	frecuencia
tc	9
ons	11
e	13
␣	14
buxz.,dlmi	16
pra	16

Tabla 14: Unimos los símbolos `pr` y `a`

símbolo	frecuencia
buxz.,dlmi	16
pra	16
tcons	20
e ␣	27

Tabla 16: Unimos los símbolos `e` y ␣

símbolo	frecuencia
buxz·,dlmipra	32
tconse ␣	47

Tabla 18: Unimos los símbolos `tcons` y `e ␣`

Así, el árbol de Huffman se vería de la forma:

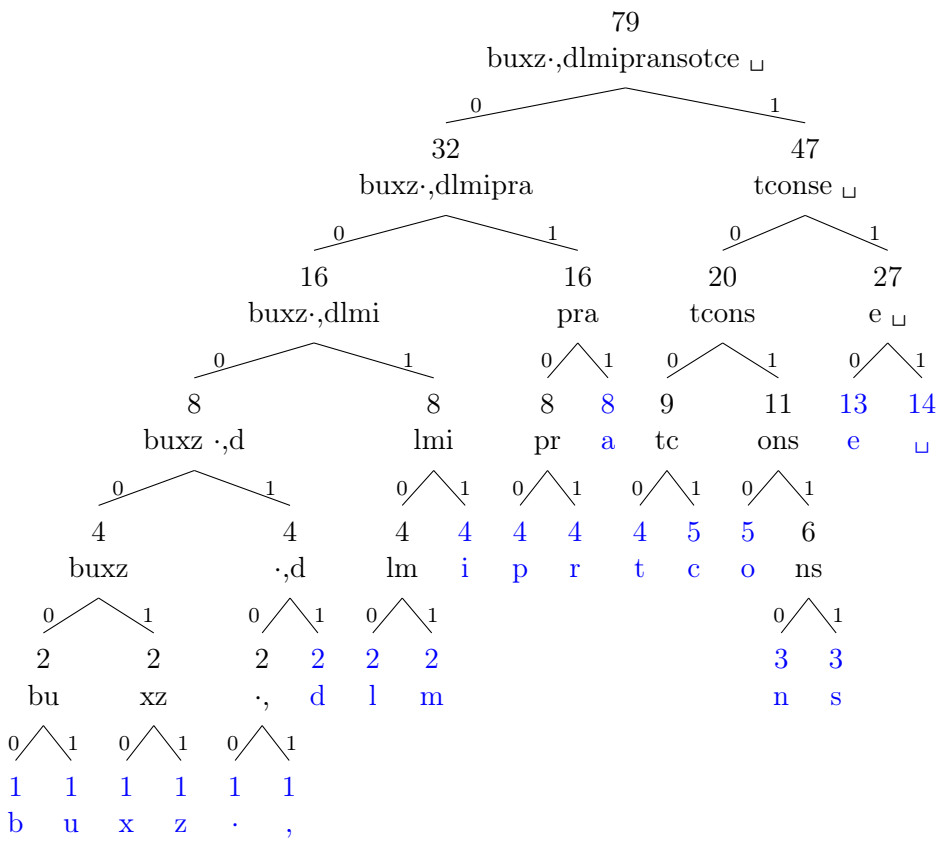


Figura 2: Árbol de Huffman

Por lo tanto, la codificación de cada símbolo sería

símbolo	codificación
b	000000
u	000001
x	000010
z	000011
·	000100
,	000101
d	00011
l	00100
m	00101
i	0011
p	0100
r	0101
a	011
t	1000
c	1001
o	1010
n	10110
s	10111
e	110
␣	111

8. Supongamos que el mago Merlín tiene un conjunto A de pociones de tamaño n y que quiere mezclarlas todas pero haciéndo parejas entre todas ellas pero sin tomarlas arbitrariamente, es decir, haciéndo pares entre pociones consecutivas. El costo de unir dos pociones es de $A[i] \times A[i+1]$ y genera la poción $(A[i]+A[i+1])$ (mód 10).

   a) Diseña un algoritmo que garantice unir todas las pociones con un costo mínimo.

   Solución: Sea $A = P_1, P_2, \ldots, P_n$ el conjunto de pociones de Merlín. Definimos a $M(i,j)$ como el costo mínimo de unir las pociones $P_{i\ldots j}$. Si tenemos n pociones que se multiplican $P_iP_{i+1}\cdots P_j$ para $1 \leq i \leq j \leq n$, entonces el mínimo costo para $A_{1\ldots n}$ será $M(1,n)$.

   Tomemos en cuenta las siguientes observaciones:

   - Si $i = j$, entonces la cadena de pociones es simplemente $P_{i\ldots i} = P_i$. Entonces, no se tienen que efectuar operaciones para multiplicar una sola matriz, por lo que $M(i,j) = 0$.
   - Si $i < j$, entonces podemos aprovechar la substructura óptima del problema que ya calculamos en el primer paso. Supongamos que vamos a separar la secuencia de pociones a multiplicar en dos segmentos dados por P_k y P_{k+1}, con $1 \leq k < j$. Así, la secuencia $P_iP_{i+1}\cdots P_j$ será $(P_iP_{i+1}\cdots P_k)(P_{k+1}\cdots P_j)$. Por lo tanto, el costo de multiplicar la secuencia será el costo de ambas subsecuencias sumado al costo de multiplicar ambas secuencias entre sí.

   De esta forma, la recurrencia del problema sería:

$$M(i,j) = \begin{cases} 0 & \text{si i = j} \\ min_{i \leq k < j} \{M(i,k) + M[k+1,j] + p_{i-1}p_kp_j\} & \text{si i < j} \end{cases}$$

   Esto lo podemos traducir al siguiente algoritmo:

   1. Definimos la matriz $M[n,n]$ y la variable $i = 1$.
   2. Mientras $i \leq n$, hacemos
      - M[i,i] = 0
      - Incrementamos en una unidad a i.
   3. Definimos las variable $i = d = 1$ y $j = 0$.
   4. Mientras d sea menor o igual a $n-1$, hacemos:
      - Mientras i sea menor o igual a $n-d$, hacemos
        - $j = (i+d)$ (mód 10)
        - $M[i,j] = \infty$
        - Mientras $k = i$, sea menor a $j-1$, hacemos
          - $M[i,j] = min\{M[i,j], M[i,k] + M[k+1,j] + d_{i-1}d_kd_j\}$
   5. Regresamos $M[0,n]$.

   Ahora bien, como todos los valores $M[i][j]$ deben ser calculados antes de que se pueda calcular $M[0,n]$, entonces esto nos toma $O(n^2)$; y cada calculo $M[i][j]$ nos toma tiempo lineal. Por lo tanto, la complejidad total del algoritmo es de $O(n^3)$.

   b) ¿Cuál es el mínimo costo si se tienen 5 pociones cuyos valores son: $1, 9, 6, 2, 3$?

   Solución:

P_i	R	C
$(1,9)$	0	9
$(9,6)$	5	54
$(6,2)$	8	12
$(2,3)$	5	6

Tabla 20: Conjuntos de longitud 2

P_i	R	C
$(1,(9,6))$	6	59
$((1,9),6)$	6	9
$(9,(6,2))$	7	84
$((9,6),2)$	7	64
$(6,(2,3))$	1	36
$((6,2),3)$	1	36

Tabla 21: Conjuntos de longitud 3

P_i	R	C
$(1,((9,6),2))$	8	71
$(1,(9,(6,2)))$	8	91
$(9,((6,2),3))$	0	45
$(9,(6,(2,3)))$	0	45
$(((1,9),6),2)$	8	21
$((1,(9,6)),2)$	8	21
$(((9,6),2),3)$	0	71
$((9,(6,2)),3)$	0	105

Tabla 22: Conjuntos de longitud 4

P_i	R	C
$(1,(9,((6,2),3)))$	1	45
$(1,(9,(6,(2,3))))$	1	45
$(1,(((9,6),2),3))$	1	71
$(1,((9,(6,2)),3))$	1	105
$((1,((9,6),2)),3)$	1	95
$((1,(9,(6,2))),3)$	1	115
$((((1,9),6),2),3)$	1	45
$(((1,(9,6)),2),3)$	1	45

Tabla 23: Conjuntos de longitud 5

Por lo tanto, el costo mínimo es 45 con una poción resultante 1.