Programación III

Ricardo Wehbe

UADE

31 de agosto de 2021

Programa

La técnica *divide & conquer*

La técnica *divide & conquer*

- Es una técnica que consiste en dividir un problema "grande" en una serie de problemas "pequeños" de resolución más simple y luego combinar las soluciones de los problemas "pequeños" para obtener una solución del problema "grande."

La técnica *divide & conquer*

- Es una técnica que consiste en dividir un problema "grande" en una serie de problemas "pequeños" de resolución más simple y luego combinar las soluciones de los problemas "pequeños" para obtener una solución del problema "grande."
- La técnica es recursiva: los problemas "pequeños" se subdividen a su vez hasta llegar a un problema "mínimo" de resolución trivial (el caso base de recurrencia).

La técnica *divide & conquer*

- Es una técnica que consiste en dividir un problema "grande" en una serie de problemas "pequeños" de resolución más simple y luego combinar las soluciones de los problemas "pequeños" para obtener una solución del problema "grande."
- La técnica es recursiva: los problemas "pequeños" se subdividen a su vez hasta llegar a un problema "mínimo" de resolución trivial (el caso base de recurrencia).
- Se efectúan entonces los siguientes pasos: 1. dividir, 2. conquistar (resolver los problemas mínimos) y 3. combinar las soluciones obtenidas.

Un algoritmo *divide & conquer* genérico

Un algoritmo *divide & conquer* genérico

```
1.   Algoritmo DnC (x)
2.       if isSmall(x) {
3.           return TrivialSolution (x)
4.       } else {
5.           ⟨x_1, ..., x_n⟩ ← decompose (x)
6.           for (i = 0; n < n; i++) {
7.               y_i ← DnC(x_i)
8.           }
9.           return combine(y_1, ..., y_n)
10.      }
```

Of course none of us are greedy. It's only the other fellow who's greedy.

(Por supuesto, ninguno de nosotros es codicioso. Es siempre el otro el que es codicioso.)

Milton Friedman en una entrevista en TV con Phil Donahue in 1979

Algoritmos *Greedy*

Algoritmos *Greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.

Algoritmos *Greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.
- Un algoritmo *greedy* construye la solución a partir de decisiones parciales basadas en la información disponible en el momento. No considera los efectos de sus decisiones en el futuro y nunca reconsidera una decisión ya tomada.

Algoritmos *Greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.
- Un algoritmo *greedy* construye la solución a partir de decisiones parciales basadas en la información disponible en el momento. No considera los efectos de sus decisiones en el futuro y nunca reconsidera una decisión ya tomada.
- Se suelen utilizar para resolver problemas de optimización. Son muy eficientes, pero hay que demostrar formalmente su corrección.

Algoritmos *Greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.
- Un algoritmo *greedy* construye la solución a partir de decisiones parciales basadas en la información disponible en el momento. No considera los efectos de sus decisiones en el futuro y nunca reconsidera una decisión ya tomada.
- Se suelen utilizar para resolver problemas de optimización. Son muy eficientes, pero hay que demostrar formalmente su corrección.
- Como el nombre lo sugiere, son "cortos de vista."

Algoritmos *greedy*. Candidatos y criterios

Algoritmos *greedy*. Candidatos y criterios

- Un algoritmo *greedy* selecciona en cada momento el mejor candidato para incluirlo en una solución basándose en un criterio determinado.

Algoritmos *greedy*. Candidatos y criterios

- Un algoritmo *greedy* selecciona en cada momento el mejor candidato para incluirlo en una solución basándose en un criterio determinado.
- En otras palabras, en cada paso se evalúa un candidato de una lista y, dependiendo del resultado de esta evaluación, se lo agrega a la solución o se o descarta, por lo menos por el momento.

Los elementos de un algoritmo *greedy*

Los elementos de un algoritmo *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.

Los elementos de un algoritmo *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.

Los elementos de un algoritmo *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.
- **Función de factibilidad**: el criterio para determinar si un candidato puede ser incluido en la solución.

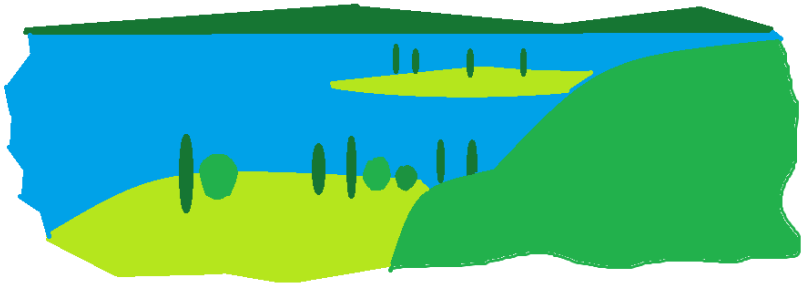
Los elementos de un algoritmo *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.
- **Función de factibilidad**: el criterio para determinar si un candidato puede ser incluido en la solución.
- **Función solución**: el criterio para determinar si un conjunto solución resuelve efectivamente el problema.

Los elementos de un algoritmo *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.
- **Función de factibilidad**: el criterio para determinar si un candidato puede ser incluido en la solución.
- **Función solución**: el criterio para determinar si un conjunto solución resuelve efectivamente el problema.
- **Objetivo**: la magnitud que se quiere maximizar o minimizar.

Un primer ejemplo. Frutillas en Thurgau



Un primer ejemplo. Frutillas en Thurgau

- Imagine que se encuentra en Thurgau, donde dispone de las siguientes monedas: 5 francos, 2 francos, un franco, 50 centavos (medio franco), 20 centavos, 10 centavos, 5 centavos, y un centavo.

Un primer ejemplo. Frutillas en Thurgau

- Imagine que se encuentra en Thurgau, donde dispone de las siguientes monedas: 5 francos, 2 francos, un franco, 50 centavos (medio franco), 20 centavos, 10 centavos, 5 centavos, y un centavo.
- Usted debe pagar 8.78 francos por un paquete de frutillas (un producto típico local), y quiere usar la menor cantidad posible de monedas.

Un primer ejemplo. Frutillas en Thurgau

- Imagine que se encuentra en Thurgau, donde dispone de las siguientes monedas: 5 francos, 2 francos, un franco, 50 centavos (medio franco), 20 centavos, 10 centavos, 5 centavos, y un centavo.
- Usted debe pagar 8.78 francos por un paquete de frutillas (un producto típico local), y quiere usar la menor cantidad posible de monedas.
- Una manera intuitiva es: elegir siempre la mayor moneda que cubre una parte de la suma sin pasarse; repetir esto hasta completar el pago.

Un primer ejemplo. Frutillas en Thurgau

- Imagine que se encuentra en Thurgau, donde dispone de las siguientes monedas: 5 francos, 2 francos, un franco, 50 centavos (medio franco), 20 centavos, 10 centavos, 5 centavos, y un centavo.
- Usted debe pagar 8.78 francos por un paquete de frutillas (un producto típico local), y quiere usar la menor cantidad posible de monedas.
- Una manera intuitiva es: elegir siempre la mayor moneda que cubre una parte de la suma sin pasarse; repetir esto hasta completar el pago.
- Tenemos entonces la siguiente solución que es óptima:

$$8.78\,\text{fr} = 5\,\text{fr} + 2\,\text{fr} + 1\,\text{fr} + 50\,¢ + 20¢ + 5¢ + 3 \times\ 1¢.$$

El algoritmo del cambio

```
1.    int Algoritmo Cambio(int v)      // input: monto; output: número de monedas
2.        int n = 0                    // número de monedas usadas
3.        int accum = 0                // monto pagado
4.          int i = 0
5.          int [] coins = [500, 200, 100, 50, 20, 10, 5, 1]
6.          while (accum < v) && (i < length(coins)) {
7.                if (accum + coins[i] <= v) {
8.                      accum = accum + coins[i]
9.                      n = n + 1      // mayor moneda que podemos usar
10.               } else {
11.                     i = i + 1      // seguimos buscando
12.               }
13.         }
14.         if (i < length(coins))
15.               return n             // devolvemos el número de monedas usadas
16.         else
17.               return -1            // no hay solución
18.   }
```

El algoritmo del cambio

```
1.    int Algoritmo Cambio(int v)      // input: monto; output: número de monedas
2.        int n = 0                    // número de monedas usadas
3.        int accum = 0                // monto pagado
4.          int i = 0
5.          int [] coins = [500, 200, 100, 50, 20, 10, 5, 1]
6.          while (accum < v) && (i < length(coins)) {
7.              if (accum + coins[i] <= v) {
8.                  accum = accum + coins[i]
9.                  n = n + 1        // mayor moneda que podemos usar
10.             } else {
11.                 i = i + 1        // seguimos buscando
12.             }
13.         }
14.         if (i < length(coins))
15.             return n             // devolvemos el número de monedas usadas
16.         else
17.             return -1            // no hay solución
18.   }
```

$\Theta(v)$

Complejidad y correción del algoritmo del cambio

Complejidad y correción del algoritmo del cambio

- El algoritmo tiene un ciclo que en el peor caso itera hasta v. Estamos entonces en $\Theta(v)$.

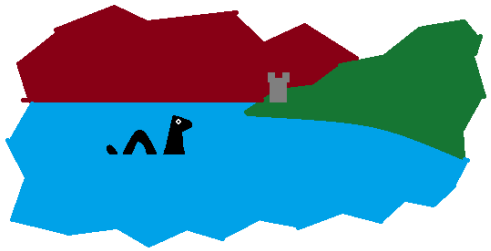
Complejidad y correción del algoritmo del cambio

- El algoritmo tiene un ciclo que en el peor caso itera hasta v. Estamos entonces en $\Theta(v)$.
- Observe que el vector de monedas se trata como una constante. Asumimos que la cantidad de denominaciones de monedas es constante, por lo que es irrelevante desde el punto de vista de la complejidad.

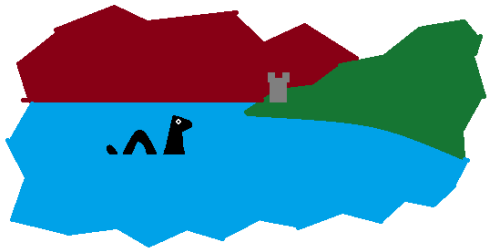
Complejidad y correción del algoritmo del cambio

- El algoritmo tiene un ciclo que en el peor caso itera hasta v. Estamos entonces en $\Theta(v)$.
- Observe que el vector de monedas se trata como una constante. Asumimos que la cantidad de denominaciones de monedas es constante, por lo que es irrelevante desde el punto de vista de la complejidad.
- En lo que respecta a corrección, considere el siguiente ejemplo.

Un segundo ejemplo. Whisky en Loch Ness, antes de 1971

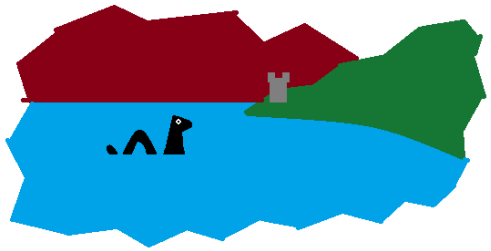


Un segundo ejemplo. Whisky en Loch Ness, antes de 1971



- Suponga ahora que está en Loch Ness, donde se dispone de las siguientes monedas: una libra, una corona (un cuarto de libra), media corona (dos chelines y seis peniques), un florín (dos chelines) un chelín (12 peniques), 6 peniques, 3 peniques, un penique.

Un segundo ejemplo. Whisky en Loch Ness, antes de 1971



- Suponga ahora que está en Loch Ness, donde se dispone de las siguientes monedas: una libra, una corona (un cuarto de libra), media corona (dos chelines y seis peniques), un florín (dos chelines) un chelín (12 peniques), 6 peniques, 3 peniques, un penique.
- Usted debe pagar una libra y 48 peniques por un vaso de whisky (un producto típico local) y decide usar el algoritmo que tan buen resultado le dio en Thurgau.

Whisky en Loch Ness, continuación

Whisky en Loch Ness, continuación

- Resumiendo, tenemos las siguientes denominaciones: 1£; 60p (1 corona); 30p ($\frac{1}{2}$ corona); 24p (1 florín); 12p (1 chelín); 6p, 3p, 1p. Cuatro coronas hacen una libra, que por lo tanto vale 240p.

Whisky en Loch Ness, continuación

- Resumiendo, tenemos las siguientes denominaciones: 1£; 60p (1 corona); 30p ($\frac{1}{2}$ corona); 24p (1 florín); 12p (1 chelín); 6p, 3p, 1p. Cuatro coronas hacen una libra, que por lo tanto vale 240p.
- Solución propuesta por el algoritmo para pagar 1£48p:

$$1£ + \underbrace{0.5\ corona}_{30p} + 12p + 6p \qquad \text{(cuatro monedas)}$$

Whisky en Loch Ness, continuación

- Resumiendo, tenemos las siguientes denominaciones: 1£; 60p (1 corona); 30p ($\frac{1}{2}$ corona); 24p (1 florín); 12p (1 chelín); 6p, 3p, 1p. Cuatro coronas hacen una libra, que por lo tanto vale 240p.
- Solución propuesta por el algoritmo para pagar 1£48p:

$$1£ + \underbrace{0.5\ corona}_{30p} + 12p + 6p \qquad \text{(cuatro monedas)}$$

- Solución óptima:

$$1£ + \underbrace{2 \times 1florín}_{48p} \qquad \text{(tres monedas)}$$

Whisky en Loch Ness, continuación

- Resumiendo, tenemos las siguientes denominaciones: 1£; 60p (1 corona); 30p ($\frac{1}{2}$ corona); 24p (1 florín); 12p (1 chelín); 6p, 3p, 1p. Cuatro coronas hacen una libra, que por lo tanto vale 240p.
- Solución propuesta por el algoritmo para pagar 1£48p:

$$1£ + \underbrace{0.5\ corona}_{30p} + 12p + 6p \qquad \text{(cuatro monedas)}$$

- Solución óptima:

$$1£ + \underbrace{2 \times 1florín}_{48p} \qquad \text{(tres monedas)}$$

- A partir de 1971, este abstruso sistema fue abandonado. Actualmente existen las siguientes denominaciones: 2£, 1£, (100p), 50p, 20p, 10p, 5p and 1p. Y el algoritmo da una solución óptima.

Los elementos del problema del cambio

Los elementos del problema del cambio

- El *conjunto de candidatos* es el conjunto de denominaciones de monedas que tenemos.

Los elementos del problema del cambio

- El *conjunto de candidatos* es el conjunto de denominaciones de monedas que tenemos.
- La *función de selección* elige siempre la mayor moneda.

Los elementos del problema del cambio

- El *conjunto de candidatos* es el conjunto de denominaciones de monedas que tenemos.
- La *función de selección* elige siempre la mayor moneda.
- La *función de factibilidad* verifica que la moneda seleccionada no se pase del monto por pagarse.

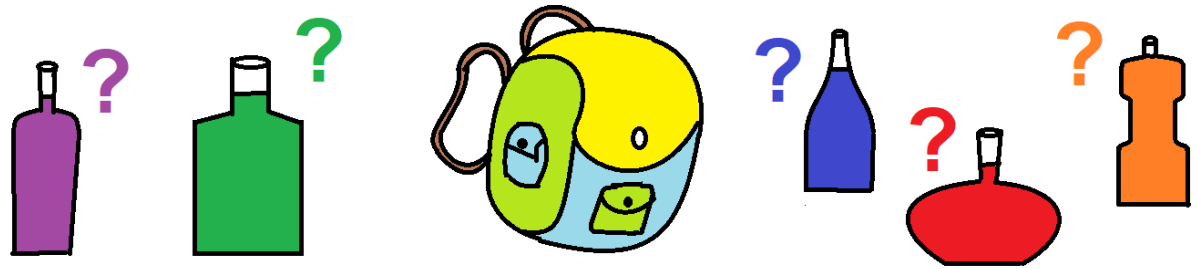
Los elementos del problema del cambio

- El *conjunto de candidatos* es el conjunto de denominaciones de monedas que tenemos.
- La *función de selección* elige siempre la mayor moneda.
- La *función de factibilidad* verifica que la moneda seleccionada no se pase del monto por pagarse.
- La *función de solución* verifica que hayamos completado el pago.

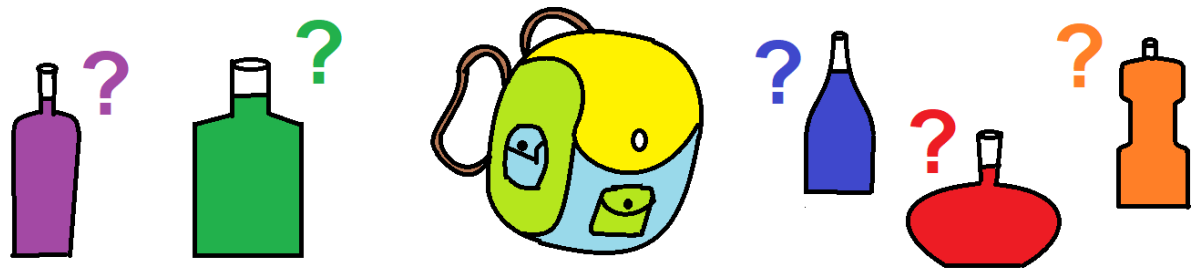
Los elementos del problema del cambio

- El *conjunto de candidatos* es el conjunto de denominaciones de monedas que tenemos.
- La *función de selección* elige siempre la mayor moneda.
- La *función de factibilidad* verifica que la moneda seleccionada no se pase del monto por pagarse.
- La *función de solución* verifica que hayamos completado el pago.
- El *objetivo* es minimizar el número de monedas utilizado para el pago.

El problema de la mochila

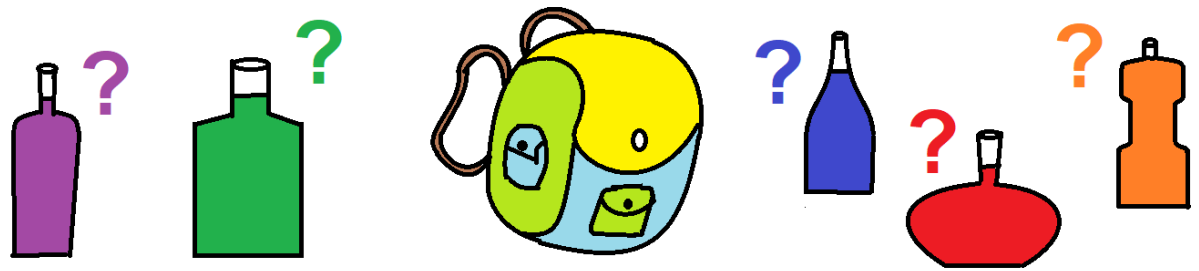


El problema de la mochila



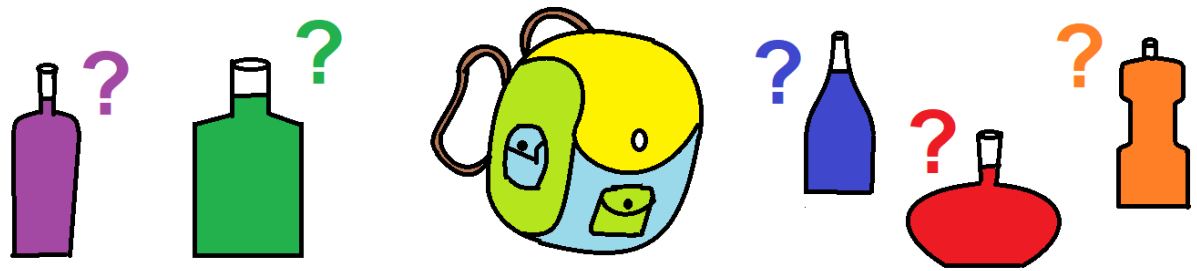
- Se trata de otro problema clásico de optimización. Se tiene una mochila que puede llevar un peso de hasta P. Hay por otro lado n objetos identificados por sus números, de 1 a n.

El problema de la mochila



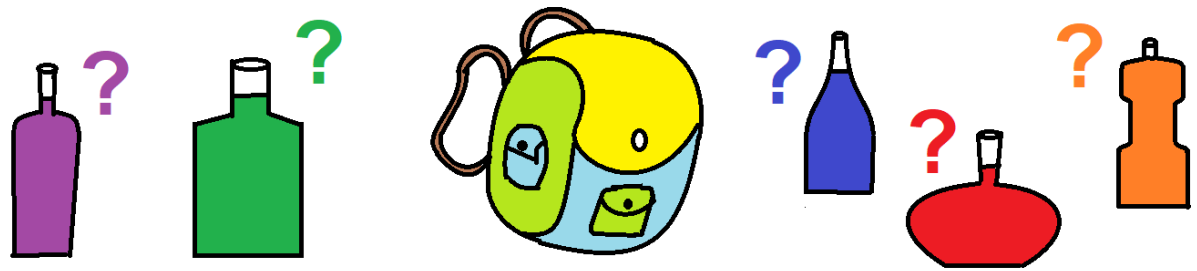
- Se trata de otro problema clásico de optimización. Se tiene una mochila que puede llevar un peso de hasta P. Hay por otro lado n objetos identificados por sus números, de 1 a n.
- Cada objeto $i \in \{1, \ldots, n\}$ tiene un valor $V_i > 0$ y un peso $P_i > 0$.

El problema de la mochila



- Se trata de otro problema clásico de optimización. Se tiene una mochila que puede llevar un peso de hasta P. Hay por otro lado n objetos identificados por sus números, de 1 a n.
- Cada objeto $i \in \{1, \ldots, n\}$ tiene un valor $V_i > 0$ y un peso $P_i > 0$.
- El objetivo es colocar objetos en la mochila de manera que se maximice el valor almacenado respetando la limitación de peso.

El problema de la mochila



- Se trata de otro problema clásico de optimización. Se tiene una mochila que puede llevar un peso de hasta P. Hay por otro lado n objetos identificados por sus números, de 1 a n.
- Cada objeto $i \in \{1, \ldots, n\}$ tiene un valor $V_i > 0$ y un peso $P_i > 0$.
- El objetivo es colocar objetos en la mochila de manera que se maximice el valor almacenado respetando la limitación de peso.
- Los objetos pueden ser fraccionados. Para un valor $0 \leq X_i \leq 1$, el valor almacenado es $X_i \times V_i$ y el peso correspondiente es $X_i \times P_i$.

Un ejemplo e problema de la mochila

Un ejemplo e problema de la mochila

- Supongamos que tenemos cuatro objetos con valores $[4, 7, 2, 5]$ y pesos$[3, 5, 4, 4]$. Nuestra mochila tiene una capacidad máxima de 10.

Un ejemplo e problema de la mochila

- Supongamos que tenemos cuatro objetos con valores $[4, 7, 2, 5]$ y pesos$[3, 5, 4, 4]$. Nuestra mochila tiene una capacidad máxima de 10.
- Algunas combinaciones posibles son las siguientes:

X_1	X_2	X_3	X_4	$\sum X_i \times P_i$	$\sum X_i \times V_i$
0.5	0.5	1	0.5	10	10
1	1	0.5	0	10	12
0	1	0.25	1	10	12.5
0.33	1	0	1	10	13.33
0.33	0.8	0.45	0.8	10	11.83

Un ejemplo e problema de la mochila

- Supongamos que tenemos cuatro objetos con valores $[4, 7, 2, 5]$ y pesos$[3, 5, 4, 4]$. Nuestra mochila tiene una capacidad máxima de 10.
- Algunas combinaciones posibles son las siguientes:

X_1	X_2	X_3	X_4	$\sum X_i \times P_i$	$\sum X_i \times V_i$
0.5	0.5	1	0.5	10	10
1	1	0.5	0	10	12
0	1	0.25	1	10	12.5
0.33	1	0	1	10	13.33
0.33	0.8	0.45	0.8	10	11.83

- La mejor opción es $X_1 = 1$, $X_2 = 1$, $X_3 = 0$ y $X_4 = 0.5$, que da un valor de 13.5.

Los elementos del problema de la mochila

Los elementos del problema de la mochila

- El *conjunto de candidatos* son los objetos disponibles.

Los elementos del problema de la mochila

- El *conjunto de candidatos* son los objetos disponibles.
- La *función de selección* toma el objeto con mejor relación valor / peso.

Los elementos del problema de la mochila

- El *conjunto de candidatos* son los objetos disponibles.
- La *función de selección* toma el objeto con mejor relación valor / peso.
- La *función de factibilidad* verifica que no nos excedamos del peso máximo.

Los elementos del problema de la mochila

- El *conjunto de candidatos* son los objetos disponibles.
- La *función de selección* toma el objeto con mejor relación valor / peso.
- La *función de factibilidad* verifica que no nos excedamos del peso máximo.
- La *función de solución* verifica que la mochila esté llena o que hayamos colocado todos los objetos.

Los elementos del problema de la mochila

- El *conjunto de candidatos* son los objetos disponibles.
- La *función de selección* toma el objeto con mejor relación valor / peso.
- La *función de factibilidad* verifica que no nos excedamos del peso máximo.
- La *función de solución* verifica que la mochila esté llena o que hayamos colocado todos los objetos.
- El *objetivo* es lograr el máximo valor con el peso máximo.

El algoritmo de la mochila

```
1.   float [] Algoritmo Knapsack (obj [] O, int max) {      // O[i].weight, O[i].value
2.        float [] R                                        // La salida
3.        sort O by value / weight                          // Θ(n log n)
4.        for (i = 0; i < n; i++) {                         // Θ(n)
5.             R[i] = 0
6.        }
7.        i = 0
8.        int accum = 0
9.        while (accum < max) && (i < n) {                  // Θ(n)
10.            R[i] = min(1, (max − accum)/O[i].weight)
11.            accum = accum + R[i] ∗ O[i].weight
12.            i++
13.       }
14.       return R
15.  }
```

El algoritmo de la mochila

```
1.    float [] Algoritmo Knapsack (obj [] O, int max) {      // O[i].weight, O[i].value
2.          float [] R                                        // La salida
3.          sort O by value / weight                          // Θ(n log n)
4.          for (i = 0; i < n; i++) {                         // Θ(n)
5.                R[i] = 0
6.          }
7.          i = 0
8.          int accum = 0
9.          while (accum < max) && (i < n) {                  // Θ(n)
10.               R[i] = min(1, (max − accum)/O[i].weight)
11.               accum = accum + R[i] ∗ O[i].weight
12.               i++
13.         }
14.         return R
15.   }
```

$$\Theta(n \log n) + \Theta(n) + \Theta(n) = \Theta(n \log n)$$

Complejidad y corrección del algoritmo de la mochila

Complejidad y corrección del algoritmo de la mochila

- La complejidad del algoritmo está dada sobre todo por el proceso de ordenamiento previo. La complejidad es entonces $\Theta(n \log n)$.

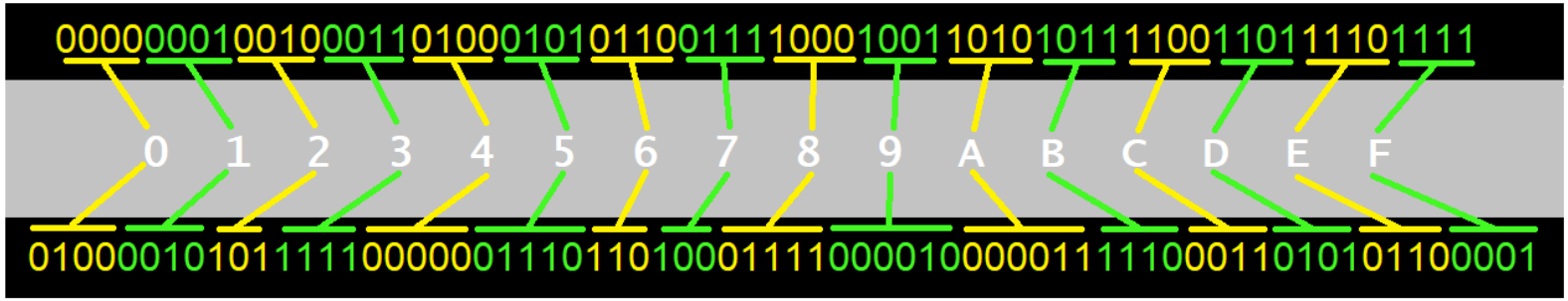
Complejidad y corrección del algoritmo de la mochila

- La complejidad del algoritmo está dada sobre todo por el proceso de ordenamiento previo. La complejidad es entonces $\Theta(n \log n)$.
- El algoritmo es correcto. La solución que da es óptima.

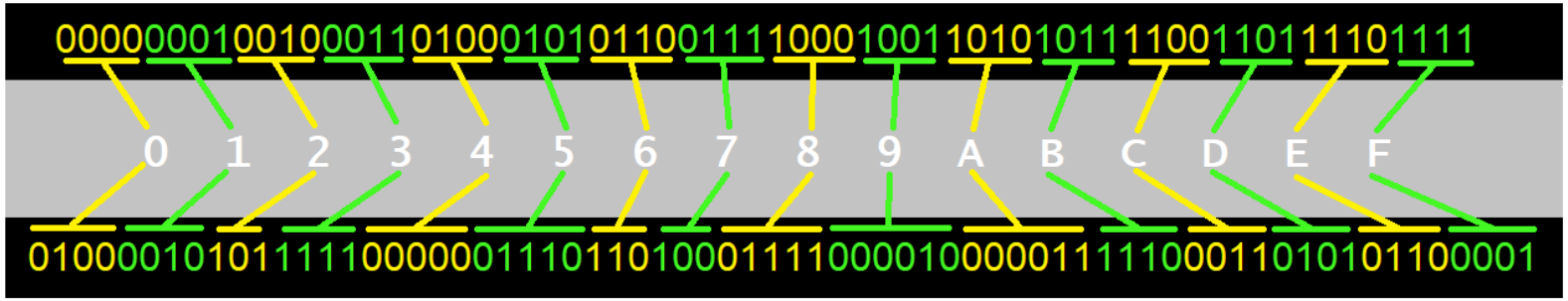
Complejidad y corrección del algoritmo de la mochila

- La complejidad del algoritmo está dada sobre todo por el proceso de ordenamiento previo. La complejidad es entonces $\Theta(n \log n)$.
- El algoritmo es correcto. La solución que da es óptima.
- Se verán en el curso otras variantes de este algoritmo que requerirán ajustes sobre este algoritmo.

Códigos de longitud fija y variable

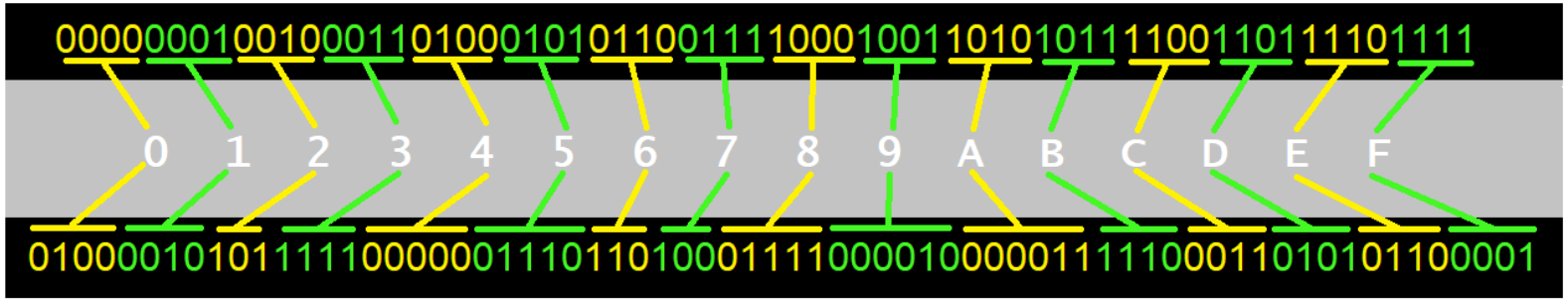


Códigos de longitud fija y variable



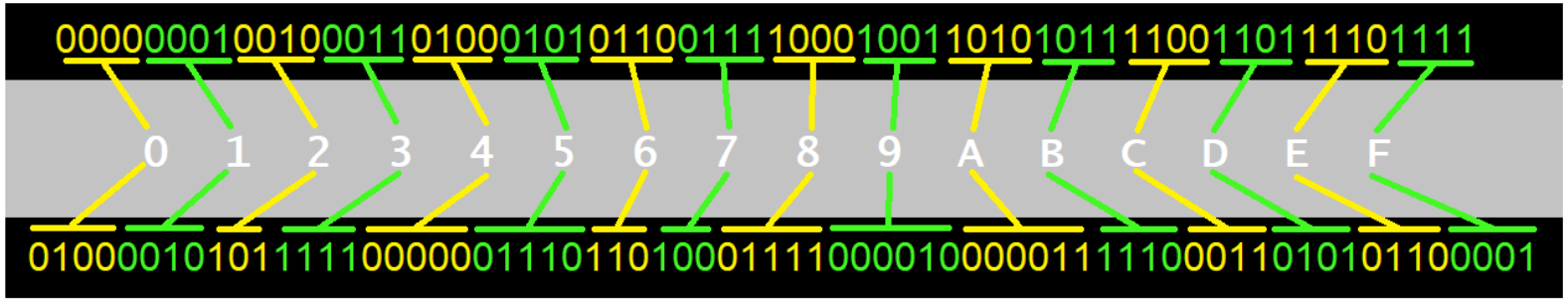
- En un *código de longitud fija* es un código todas las palabras de código tienen la misma longitud. Ejemplo: codificamos $\{A, B, C, D, E, F\}$ con $A \to 000$, $B \to 001$, $C \to 010$, $D \to 011$, $E \to 100$ y $F \to 101$.

Códigos de longitud fija y variable



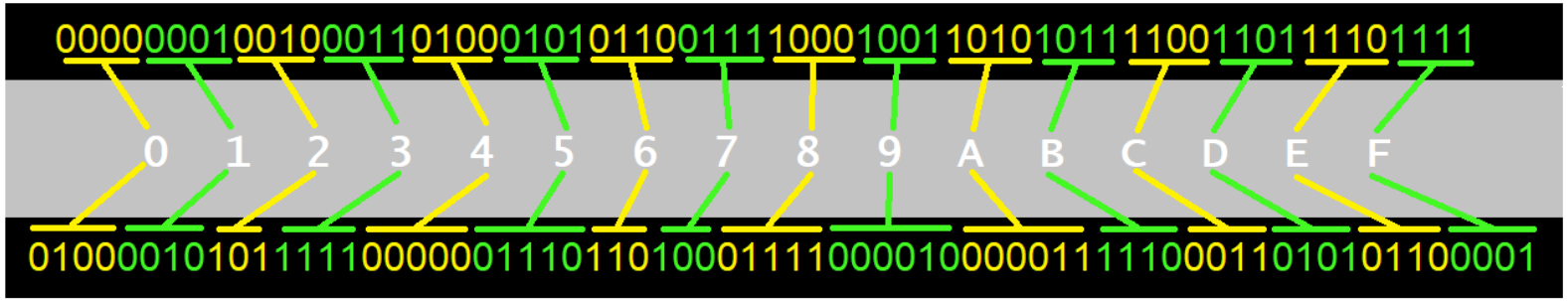
- En un *código de longitud fija* es un código todas las palabras de código tienen la misma longitud. Ejemplo: codificamos $\{A, B, C, D, E, F\}$ con $A \to 000$, $B \to 001$, $C \to 010$, $D \to 011$, $E \to 100$ y $F \to 101$.
- En un *código de longitud variable* las palabras de código tienen diferente longitud. Ejemplo: $A \to 0$, $B \to 101$, $C \to 100$, $D \to 111$, $E \to 1101$ y $F \to 1100$.

Códigos de longitud fija y variable



- En un *código de longitud fija* es un código todas las palabras de código tienen la misma longitud. Ejemplo: codificamos $\{A, B, C, D, E, F\}$ con $A \to 000$, $B \to 001$, $C \to 010$, $D \to 011$, $E \to 100$ y $F \to 101$.
- En un *código de longitud variable* las palabras de código tienen diferente longitud. Ejemplo: $A \to 0$, $B \to 101$, $C \to 100$, $D \to 111$, $E \to 1101$ y $F \to 1100$.
- En un *código sin prefijos* ninguna palabra es un prefijo de otra. Los dos códigos considerados son sin prefijos.

Códigos de longitud fija y variable



- En un *código de longitud fija* es un código todas las palabras de código tienen la misma longitud. Ejemplo: codificamos $\{A, B, C, D, E, F\}$ con $A \to 000$, $B \to 001$, $C \to 010$, $D \to 011$, $E \to 100$ y $F \to 101$.
- En un *código de longitud variable* las palabras de código tienen diferente longitud. Ejemplo: $A \to 0$, $B \to 101$, $C \to 100$, $D \to 111$, $E \to 1101$ y $F \to 1100$.
- En un *código sin prefijos* ninguna palabra es un prefijo de otra. Los dos códigos considerados son sin prefijos.
- En un mensaje de 10000 símbolos la frecuencia (en cientos) es A, 45; B, 13; C, 12; D, 16; E, 9; y F, 5. La codificación con longitud fija requiere 30000 bits; la de longitud variable, 22400 bits.

Souvenir. Colas de prioridad (*priority queues*)

Souvenir. Colas de prioridad (*priority queues*)

- Una *cola de prioridad* es una estructura de datos que contiene un conjunto S de elementos, cada uno de ellos asociado a una *clave*.

Souvenir. Colas de prioridad (*priority queues*)

- Una *cola de prioridad* es una estructura de datos que contiene un conjunto S de elementos, cada uno de ellos asociado a una *clave*.
- Hay dos tipos de colas de prioridad: de máxima y de mínima. Las respectivas implementaciones son simétricas.

Souvenir. Colas de prioridad (*priority queues*)

- Una *cola de prioridad* es una estructura de datos que contiene un conjunto S de elementos, cada uno de ellos asociado a una *clave*.
- Hay dos tipos de colas de prioridad: de máxima y de mínima. Las respectivas implementaciones son simétricas.
- Las operaciones asociadas con las colas de prioridad mínima (las que nos interesan aquí) son:

Souvenir. Colas de prioridad (*priority queues*)

- Una *cola de prioridad* es una estructura de datos que contiene un conjunto S de elementos, cada uno de ellos asociado a una *clave*.
- Hay dos tipos de colas de prioridad: de máxima y de mínima. Las respectivas implementaciones son simétricas.
- Las operaciones asociadas con las colas de prioridad mínima (las que nos interesan aquí) son:
  - $insert(S, x)$: inserta el elemento x en la cola S. La complejidad de esta operación si está bien implementada es $\Theta(\log n)$.

Souvenir. Colas de prioridad (*priority queues*)

- Una *cola de prioridad* es una estructura de datos que contiene un conjunto S de elementos, cada uno de ellos asociado a una *clave*.
- Hay dos tipos de colas de prioridad: de máxima y de mínima. Las respectivas implementaciones son simétricas.
- Las operaciones asociadas con las colas de prioridad mínima (las que nos interesan aquí) son:
  - *insert*(S, x): inserta el elemento x en la cola S. La complejidad de esta operación si está bien implementada es $\Theta(\log n)$.
  - *extract-min*(S): devuelve el elemento x con la menor clave y lo elimina de la cola S. La complejidad de esta operación si está bien implementada es $\Theta(\log n)$.

El código de Huffman. Introducción

El código de Huffman. Introducción

- El código de Huffman se construye a través de un algoritmo *greedy*.

El código de Huffman. Introducción

- El código de Huffman se construye a través de un algoritmo *greedy*.
- La idea es construir un árbol binario comenzando por las hojas, que son los símbolos por ser codificados.

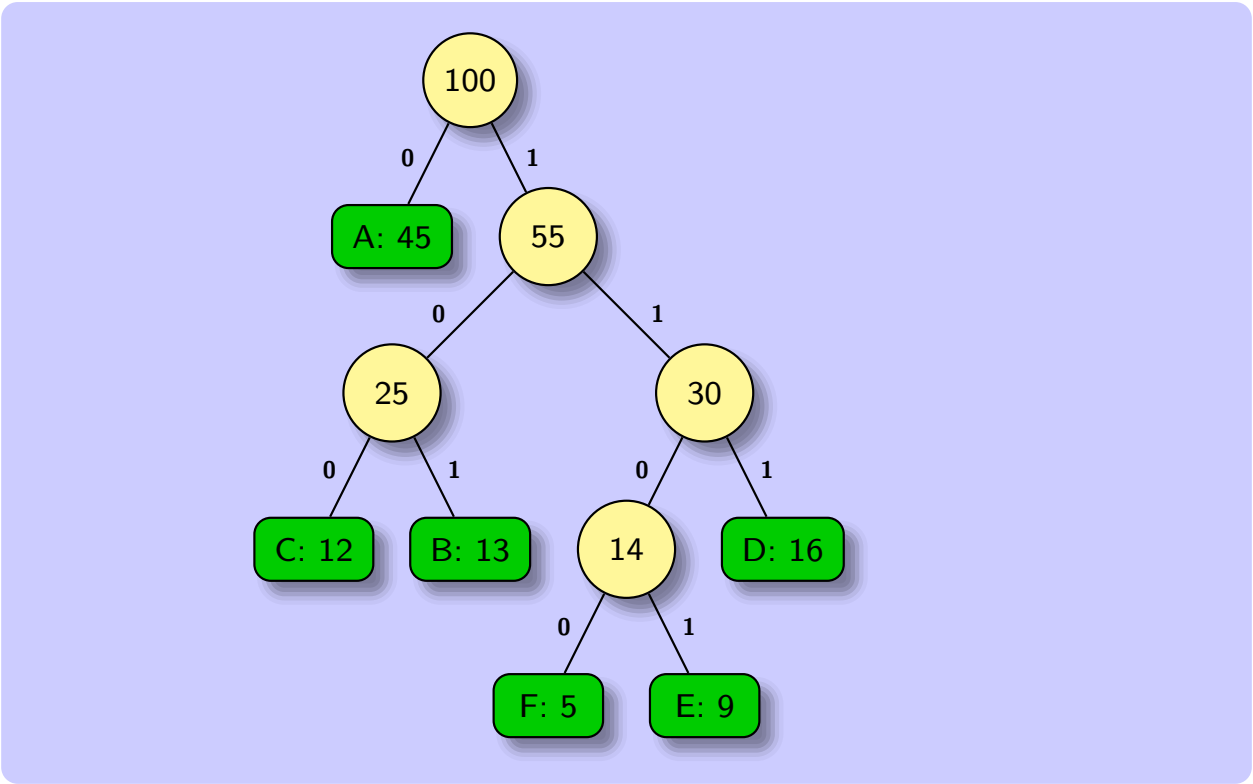
El código de Huffman. Introducción

- El código de Huffman se construye a través de un algoritmo *greedy*.
- La idea es construir un árbol binario comenzando por las hojas, que son los símbolos por ser codificados.
- Para cada símbolo, el camino desde la raíz nos da la codificación.

El código de Huffman. Introducción

- El código de Huffman se construye a través de un algoritmo *greedy*.
- La idea es construir un árbol binario comenzando por las hojas, que son los símbolos por ser codificados.
- Para cada símbolo, el camino desde la raíz nos da la codificación.
- Cada nodo interno del árbol está asociado con una clave de una cola de prioridad míma.

Un ejemplo del código de Huffman



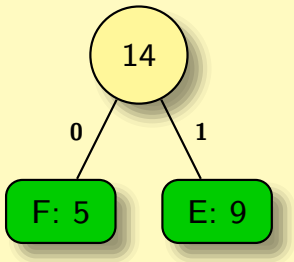
$$A \to 0 \quad B \to 101 \quad C \to 100 \quad D \to 111 \quad E \to 1101 \quad F \to 1100$$

La construcción del árbol de Huffman

La construcción del árbol de Huffman

<F:5, E:9, C:12, B:13, D:16, A:45>

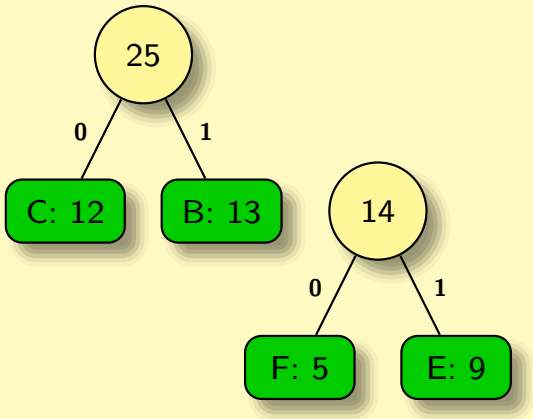
La construcción del árbol de Huffman



<F:5, E:9, C:12, B:13, D:16, A:45>

<C:12, B:13, X_1:14, D:16, A:45>

La construcción del árbol de Huffman

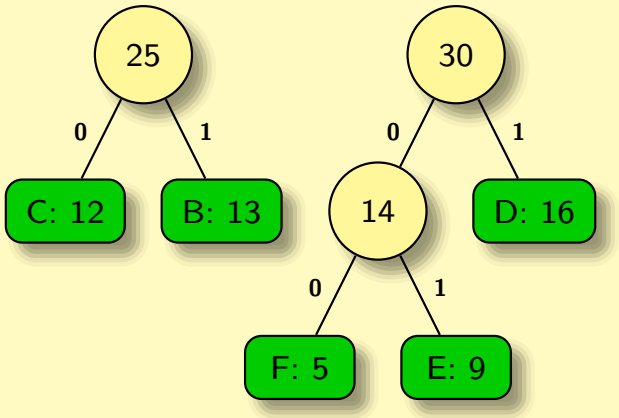


<F:5, E:9, C:12, B:13, D:16, A:45>

<C:12, B:13, X_1:14, D:16, A:45>

<X_1:14, D:16, X_2:25, A:45>

La construcción del árbol de Huffman



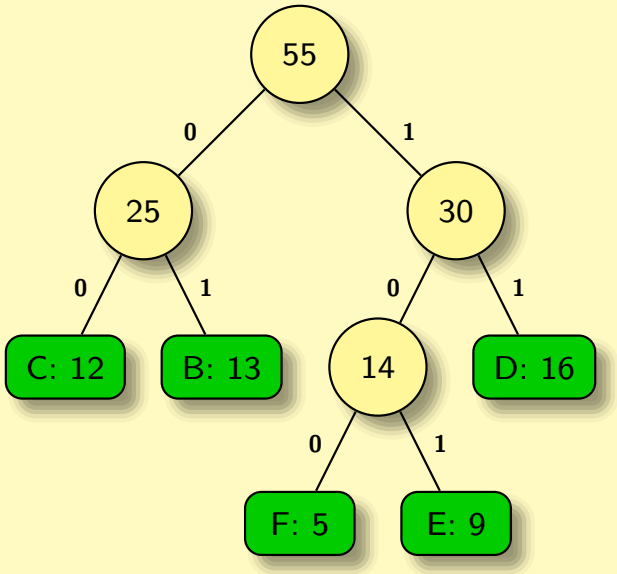
<F:5, E:9, C:12, B:13, D:16, A:45>

<C:12, B:13, X_1:14, D:16, A:45>

<X_1:14, D:16, X_2:25, A:45>

<X_2:25, X_3:30, A:45>

La construcción del árbol de Huffman



<F:5, E:9, C:12, B:13, D:16, A:45>

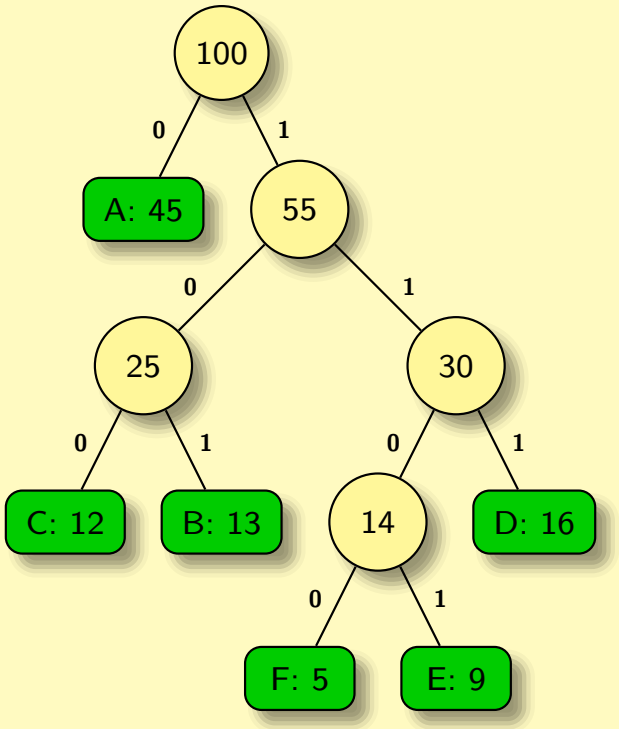
<C:12, B:13, X_1:14, D:16, A:45>

<X_1:14, D:16, X_2:25, A:45>

<X_2:25, X_3:30, A:45>

<A:45, X_4:55>

La construcción del árbol de Huffman



<F:5, E:9, C:12, B:13, D:16, A:45>

<C:12, B:13, X_1:14, D:16, A:45>

<X_1:14, D:16, X_2:25, A:45>

<X_2:25, X_3:30, A:45>

<A:45, X_4:55>

<X_5:100>

El algoritmo de Huffman

```
1.    void Algoritmo Huffman(set C)                  // símbolos
2.         n = |C|
3.         initialize min-priority queue Q           // la cola de prioridad
4.         Q ← C                                     // Θ(n log n)
5.         for (i = 1; i < n; i++) {                 // Θ(n)
6.              x = extract-min(Q)                   // Θ(log n)
7.              y = extract-min(Q)                   // Θ(log n)
8.              create new node z
9.              left(z) = x
10.             right(z) = y
11.             f(z) = f(x) + f(y)                   // f(x) is freq. of x
12.             insert(Q, z)                         // Θ(log n)
13.        }
14.        return z                                  // la raíz del árbol
15.   }
```

El algoritmo de Huffman

	1.	`void Algoritmo` *Huffman*`(set` C`)`	// símbolos
	2.	$n = \|C\|$	
$\Theta(n \log n)$	3.	`initialize min-priority queue` Q	// la cola de prioridad
	4.	$Q \leftarrow C$	// $\Theta(n \log n)$
$\Theta(n)$	5.	`for` $(i = 1; i < n; i++)$ `{`	// $\Theta(n)$
$\Theta(\log n)$	6.	$x =$ `extract-min(`Q`)`	// $\Theta(\log n)$
$\Theta(\log n)$	7.	$y =$ `extract-min(`Q`)`	// $\Theta(\log n)$
	8.	`create new node` z	
	9.	`left(`z`)` $= x$	
	10.	`right(`z`)` $= y$	
	11.	$f(z) = f(x) + f(y)$	// $f(x)$ is freq. of x
$\Theta(\log n)$	12.	`insert(`Q, z`)`	// $\Theta(\log n)$
	13.	`}`	
	14.	`return` z	// la raíz del árbol
	15.	`}`	

$$\Theta(n \log n) + \Theta(n \log n) = \Theta(n \log n)$$

Complejidad del algoritmo de Huffman

Complejidad del algoritmo de Huffman

- La inicialización tiene un costo $\mathcal{O}(n)$ (ver ejercicios).
- El ciclo `for` tiene un costo $\mathcal{O}(n)$.

Complejidad del algoritmo de Huffman

- La inicialización tiene un costo $\mathcal{O}(n)$ (ver ejercicios).
- El ciclo `for` tiene un costo $\mathcal{O}(n)$.
- Las inserciones tienen un costo $\mathcal{O}(\log n)$.

Complejidad del algoritmo de Huffman

- La inicialización tiene un costo $\mathcal{O}(n)$ (ver ejercicios).
- El ciclo `for` tiene un costo $\mathcal{O}(n)$.
- Las inserciones tienen un costo $\mathcal{O}(\log n)$.
- Por lo tanto, estamos en $\mathcal{O}(n \log n)$.

Ejercicios propuestos 1

1. El conservatorio de música de Fraile Muerto es una institución con limitados recursos. Cuando se acercan los exámenes, un solo piano de cola está disponible para que los estudiantes puedan practicar. Los estudiantes deben presentar una solicitud para usar el piano. La solicitud debe indicar la hora de inicio y de finalización.

   Tenemos el conjunto de solicitudes para un día determinado y queremos maximizar el número de estudiantes que pueden usar el piano. Proponga un esquema *greedy* para resolver este problema.

2. Varios cursos están programados en el campus de la Universidad de Fraile Muerto. Por supuesto, no puede haber dos cursos al mismo tiempo en la misma aula. Dé un algoritmo *greedy* que calcule el número mínimo de aulas necesarias para programar todos los cursos. Usted tiene a su disposición un número arbitrario de aulas.

Ejercicios propuestos 2

3. Tenemos un conjunto de tareas que deben ser realizadas por un único recurso, de manera que no se pueden realizar dos al mismo tiempo. Cada tarea j tiene un tiempo de procesamiento t_j y un plazo d_j. Si la tarea j comienza en el tiempo s_j, terminará en el tiempo $f_j = s_j + t_j$. Definimos la demora ℓ_j de la tarea j como $\ell_j = \max(0, f_j - d_j)$.

   Queremos un esquema *greedy* que minimice la máxima demora, es decir $L = \max_j \ell_j$.

Ejercicios propuestos 3

4. El rincón surrealista. El Centro de Investigaciones Espaciales de Fraile Muerto tiene un proyecto secreto sobre antimateria cuántica. La antimateria viene en frascos. Cada frasco p tiene asociado un entero positivo p_k. Es importante llevar este número a 1 en la menor cantidad de pasos que sea posible. Para ello, podemos realizar tres operaciones:
   - Incrementar el número p_k en 1: $p_k \rightarrow p_k + 1$.
   - Decrementar el número p_k en 1: $p_k \rightarrow p_k - 1$.
   - Dividir el número por 2: $p_k \rightarrow p_k/2$. Debido a la energía destructiva liberada cuando se realiza este proceso, los controles de seguridad sólo permiten que suceda si el número p_k es par.

   Encuentre un proceso para minimizar el número de pasos requerido para llevar un valor p_k dado a 1