

Estructuras de Datos y Algoritmos 2

Ignacio Rimini

March 2025

Índice

1. Unidad 1 - Modelo de Costos.	4
1.1. Notación asintótica.	4
1.1.1. Definición. Notación Big O.	4
1.1.2. Definición. Notación Big Omega.	4
1.1.3. Definición. Notación Big Theta.	4
1.1.4. Proposición. Caracterización de notación Big Theta.	4
1.2. Convenciones matemáticas.	5
1.2.1. Definición. Piso de un número.	5
1.2.2. Definición. Techo de un número.	5
1.2.3. Definición. Función asintóticamente no negativa.	5
1.2.4. Funciones logarítmicas.	5
1.2.5. Funciones exponenciales.	5
1.2.6. Series matemáticas.	6
1.3. Modelo de costos en algoritmos.	6
1.3.1. Definición. Trabajo (W).	6
1.3.2. Definición. Profundidad (S).	6
1.3.3. Definición. Paralelismo.	7
1.3.4. Principio del Scheduler Voraz (Brent).	7
1.3.5. Pseudocódigo y costos de programas.	7
1.4. Divide and Conquer.	8
1.4.1. Algoritmo Divide and Conquer.	8
1.4.2. Trabajo y profundidad de algoritmos Divide and Conquer.	8
1.4.3. Algoritmo MergeSort.	9
2. Unidad 2 - Resolución de Recurrencias.	11
2.1. Introducción.	11
2.2. Método de sustitución.	11
2.2.1. Definición. Método de sustitución.	11
2.2.2. Ejemplo. Sustitución para el mergesort.	11
2.2.3. Acerca de la adivinanza.	12
2.3. Árboles de recurrencia.	13
2.3.1. Definición. Árbol de recurrencia.	13
2.3.2. Ejemplo. Árbol de recurrencia.	13
2.4. Funciones suaves y reglas de suavidad.	15
2.4.1. Introducción.	15
2.4.2. Definición. Función eventualmente no decreciente.	15
2.4.3. Definición. Función b-suave.	15
2.4.4. Definición. Función suave.	15
2.4.5. Proposición. Condición suficiente de suavidad.	15
2.4.6. Teorema. Regla de suavidad.	15
2.4.7. Ejemplo. Recurrencia del mergeSort.	16
2.5. Teorema maestro.	17

2.5.1.	Definición. Teorema maestro.	17
3.	Unidad 3 - Programación Funcional con Haskell	18
3.1.	Programación funcional.	18
3.1.1.	¿Qué es la programación funcional?	18
3.1.2.	Ventajas de Haskell.	18
3.2.	Introducción a Haskell.	18
3.2.1.	Guía inicial para compilar y ejecutar un archivo.	18
3.2.2.	Comentarios y palabras reservadas.	19
3.2.3.	Offside rule.	19
3.2.4.	Operadores infijos.	19
3.2.5.	Tipos.	19
3.3.	Listas en Haskell.	20
3.3.1.	Definición. Lista.	20
3.3.2.	Operaciones básicas con listas.	20
3.3.3.	Funciones de listas.	21
3.3.4.	Listas por comprensión.	22
3.3.5.	Patrones de listas.	22
3.4.	Tuplas en Haskell.	22
3.4.1.	Definición. Tupla.	22
3.4.2.	Acceso a los elementos.	23
3.5.	Strings en Haskell.	23
3.6.	Expresiones condicionales en Haskell.	23
3.6.1.	If-then-else.	23
3.6.2.	Ecuaciones con guardas.	24
3.6.3.	Pattern Matching. Coincidencia de patrones.	24
3.7.	Funciones en Haskell.	25
3.7.1.	Definición. Función.	25
3.7.2.	Aplicación de funciones, asociatividad y precedencia.	25
3.7.3.	Declaración de funciones.	25
3.7.4.	Llamada a funciones.	26
3.7.5.	Currying. Currificación y aplicación parcial.	26
3.7.6.	Variables locales en funciones.	26
3.7.7.	Recursión.	27
3.7.8.	Funciones Lambda. Funciones anónimas.	27
3.7.9.	Polimorfismo paramétrico en funciones.	28
3.7.10.	Polimorfismo ad-hoc de sobrecarga de funciones.	28
3.8.	Clases de tipo.	29
3.8.1.	Definición. Clases de tipos.	29
3.8.2.	Algunas clases de tipo.	29
3.9.	Ejemplos importantes.	30
3.9.1.	Función zip.	30
3.9.2.	Notación polaca.	30
4.	Unidad 4 - Tipos en Haskell	31
4.1.	Tipos en Haskell.	31
4.2.	Sinónimos de tipo (type).	31
4.3.	Declaraciones data.	31
4.4.	Sintaxis para records/registros.	32
4.5.	Constructor de tipos Maybe.	33
4.6.	Constructor de tipos Either.	33
4.7.	Tipos recursivos.	34
4.8.	Expresiones case.	34
4.9.	Árboles.	35
4.9.1.	Declaración de tipo de datos árbol.	35
4.9.2.	Programando con árboles.	35

4.9.3. Árboles de Huffman.	36
5. Unidad 5 - Estructuras Inmutables	38
5.1. Introducción.	38
5.1.1. Estructuras de datos funcionales vs imperativas.	38
5.1.2. Inmutabilidad y sharing.	38
5.2. Árboles binarios en Haskell.	39
5.2.1. Árboles binarios.	39
5.2.2. Árboles binarios de búsqueda (BST).	39
5.2.3. Red-Black Trees.	41
5.2.4. Heaps.	43

1. Unidad 1 - Modelo de Costos.

1.1. Notación asintótica.

Cuando analizamos algoritmos para instancias grandes de su entrada, de manera que sólo el orden de crecimiento sea relevante, decimos que hacemos un **análisis asintótico** de su eficiencia.

Para comparar la eficiencia de los algoritmos utilizamos una notación que permita capturar la noción intuitiva de orden de crecimiento.

1.1.1. Definición. Notación Big O.

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}$. Decimos que f tiene orden de crecimiento $O(g)$ (y escribimos $f \in O(g)$), si existen constantes $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$, tales que:

$$0 \leq f(n) \leq c \cdot g(n), \quad \forall n \geq n_0$$

1.1.2. Definición. Notación Big Omega.

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}$. Decimos que f tiene orden de crecimiento $\Omega(g)$ (y escribimos $f \in \Omega(g)$), si existen constantes $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$, tales que:

$$0 \leq c \cdot g(n) \leq f(n), \quad \forall n \geq n_0$$

1.1.3. Definición. Notación Big Theta.

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}$. Decimos que f tiene orden de crecimiento $\Theta(g)$ (y escribimos $f \in \Theta(g)$), si $f \in O(g)$ y $g \in O(f)$.

Observaciones.

- La notación Big O establece una cota superior al crecimiento de la función f . A partir de cierto valor natural, la función g es una cota superior de la función f .
- La notación Big Omega establece una cota inferior de la función f .
- La notación Big Theta es una cota superior e inferior de la función f .
- Algunos ejemplos típicos de complejidad asintótica son:

$$O(1) \subset O(\lg n) \subset O(n) \subset O(n \lg n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!)$$

- Algunos algoritmos conocidos tienen las siguientes complejidades. Búsqueda simple es $O(n)$, Búsqueda binaria es $O(\lg n)$, Quicksort es $O(n \lg n)$, Selection Sort es $O(n^2)$ y Vendedor viajero es $O(n!)$.

1.1.4. Proposición. Caracterización de notación Big Theta.

Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}$. Decimos que f tiene orden de crecimiento $\Theta(g)$ si y solo si existen constantes $c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tales que:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$

1.2. Convenciones matemáticas.

1.2.1. Definición. Piso de un número.

Sea $x \in \mathbb{R}$, definimos el **piso** de x como:

$$\lfloor x \rfloor = \max\{n \mid n \leq x, n \in \mathbb{Z}\}$$

1.2.2. Definición. Techo de un número.

Sea $x \in \mathbb{R}$, definimos el **techo** de x como:

$$\lceil x \rceil = \min\{n \mid n \geq x, n \in \mathbb{Z}\}$$

Observación. Para cualquier $x \in \mathbb{R}$ se cumple que:

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

1.2.3. Definición. Función asintóticamente no negativa.

Una función $f : \mathbb{N} \rightarrow \mathbb{R}$ se dice **asintóticamente no negativa** si existe $N \in \mathbb{N}$ tal que:

$$\boxed{f(n) \geq 0} \quad \forall n > N$$

1.2.4. Funciones logarítmicas.

Utilizaremos la siguiente notación al utilizar logaritmos:

$$\lg n = \log_2(n), \quad \ln n = \log_e(n)$$

Luego, tenemos las siguientes propiedades de logaritmo. Sean $a, b, x, y \in \mathbb{R}^+$:

- $\log_a(xy) = \log_a(x) + \log_a(y)$
- $\log_a(x^n) = n \cdot \log_a(x)$
- $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$
- $x^{\log_a(y)} = y^{\log_a(x)}$
- $\log_a\left(\frac{1}{x}\right) = -\log_a(x)$
- $\log_a(x) = \frac{1}{\log_x(a)}$

1.2.5. Funciones exponenciales.

Sea $a \in \mathbb{R}^+$, luego se cumplen las siguientes propiedades:

- $a^{-1} = \frac{1}{a}$
- $(a^m)^n = a^{m \cdot n}$
- $a^m \cdot a^n = a^{m+n}$

1.2.6. Series matemáticas.

- $\sum_{k=0}^n a + bk = (n+1)\left(a + \frac{1}{2}bn\right)$
- $\sum_{k=0}^n ax^k = \frac{a - ax^{n+1}}{1 - x}, \quad \text{para } x \neq 1$
- $\sum_{k=1}^n k = \frac{n(n+1)}{2}$
- $\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$

1.3. Modelo de costos en algoritmos.

Utilizaremos el modelo de costos para especificar cual es el costo de un algoritmo para una entrada determinada.

1.3.1. Definición. Trabajo (W).

El **trabajo** representa el **costo secuencial** de un programa, es decir, el costo de ejecutarlo con un solo procesador. Puede ser pensado como el cómputo total que hace el programa.

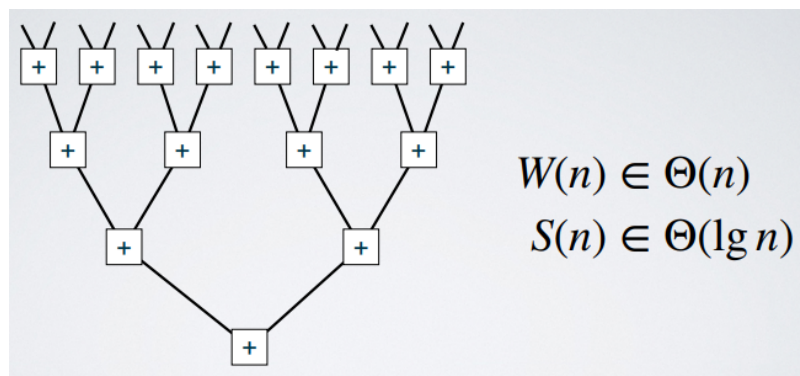
1.3.2. Definición. Profundidad (S).

La profundidad representa el **costo paralelo** de un programa, esto es, el costo de ejecutarlo con infinitos procesadores. Si bien esto es imposible, nos basta con que haya disponibilidad de procesadores siempre que se necesite.

Observación. Si representamos un programa como si fuera un grafo, el trabajo equivale a $|E|$ (cantidad de aristas), y la profundidad equivale a $\max_{p \in \text{Path}(g)} |p|$ (tamaño del mayor camino del grafo).

Ejemplo. Suma de n números. La idea más directa y sencilla es sumar los primeros $n - 1$ recursivamente y luego sumar el último. Esto nos daría que $W(n) \in O(n)$ y $S(n) \in O(n)$, ya que en ningún momento podemos dividir el cómputo en varios procesadores.

Esto se puede mejorar si dividimos la lista de números a la mitad, realizamos la suma de cada mitad en paralelo, y luego sumamos ambos valores. Esto resultaría en $W(n) \in O(n)$ y $S(n) \in O(\lg n)$.



1.3.3. Definición. Paralelismo.

El paralelismo es una medida que determina cuántos procesadores se pueden usar de forma eficiente. Su fórmula está dada por:

$$P = \frac{W}{S}$$

Observación. Siempre se abogará por algoritmos que mayor paralelismo brinden. Sin embargo, el paralelismo, al ser un cociente, podría aumentar incrementando el valor de W , el costo secuencial.

Esto no es lo que buscamos. El criterio que usaremos para elegir entre varias implementaciones de un algoritmo será escoger dentro de los que menor costo secuencial tienen, el de mayor paralelismo.

Para el ejemplo de la suma anterior, tenemos:

$$P = \frac{kn}{k' \lg n} \in O\left(\frac{n}{\lg n}\right)$$

1.3.4. Principio del Scheduler Voraz (Brent).

Un **scheduler** es una pieza del software del sistema operativo encargada de asignar a cada tarea pendiente un procesador para que éste se encargue de llevarla a cabo.

Decimos que un scheduler es **voraz**, si cuando: hay un procesador libre y hay tareas para ejecutar, entonces la tarea es asignada inmediatamente.

Principio del Scheduler Voraz. En una máquina con un scheduler voraz, el tiempo T de ejecución de un programa con trabajo W , profundidad S y p procesadores reales, cumple:

$$T < \frac{W}{p} + S$$

Observación. Esto es considerado una buena cota para el tiempo, y además, podemos reformular la desigualdad utilizando el paralelismo P :

$$T < \frac{W}{p} + S = \frac{W}{p} + \frac{W}{P} = \frac{W}{p} \left(1 + \frac{p}{P}\right)$$

Observar que si $p \ll P$ (p mucho menor que P), entonces $\frac{p}{P}$ tiende a cero y luego obtenemos una cota óptima para el tiempo. Nuevamente, estas cotas valen bajo los supuestos de un scheduler voraz y de baja latencia de comunicación entre procesadores o en red.

1.3.5. Pseudocódigo y costos de programas.

Se usará un modelo de costos basado en lenguajes para expresar los costos de nuestros programas. Utilizaremos el análisis asintótico, ya que solo nos interesan cotas para los costos de los programas.

Además, queremos que nuestro modelo se abstraiga del hardware y lenguaje sobre los cuales funcionan nuestros programas. Para esto, usaremos un pseudocódigo:

- **Constantes:** 0, 1, 2, True, False, []
- **Operadores:** +, -, *, /, <, >, &&, ||, if-then-else, ▷
- **Pares ordinarios y paralelos:** (3,4), (3+4 || 5+6).
- **Expresiones let:** let $x = 3 + 4$ in $x + x$

- **Secuencias:** $[x * 2 \mid x \leftarrow xs]$

Trabajo (W) del lenguaje pseudocódigo.

- $W(c) = 1$
- $W(op\ e) = 1 + W(e)$
- $W(e_1, e_2) = 1 + W(e_1) + W(e_2)$
- $W(e_1 || e_2) = 1 + W(e_1) + W(e_2)$
- $W(let\ x = e_1\ in\ e_2) = 1 + W(e_1) + W(e_2(x \rightarrow Eval(e_1)))$
- $W([f(x) \mid x \leftarrow xs]) = 1 + \sum_{x \in xs} W(f(x))$

Profundidad (S) del lenguaje pseudocódigo.

- $S(c) = 1$
- $S(op\ e) = 1 + S(e)$
- $S(e_1, e_2) = 1 + S(e_1) + S(e_2)$
- $S(e_1 || e_2) = 1 + \max(S(e_1), S(e_2))$
- $S(let\ x = e_1\ in\ e_2) = 1 + S(e_1) + S(e_2(x \rightarrow Eval(e_1)))$
- $S([f(x) \mid x \leftarrow xs]) = 1 + \max_{x \in xs} S(f(x))$

1.4. Divide and Conquer.

1.4.1. Algoritmo Divide and Conquer.

Es una estrategia de resolución de problemas muy útil cuando se tienen problemas cuya solución se puede plantear en términos de una solución del mismo problema pero de tamaño más chico.

Por su naturaleza, Divide and Conquer es fácil de plantear mediante recursión, y se compone de dos partes:

- **Caso base.** El tamaño del problema es chico, y se puede dar una solución específica para esta instancia del problema.
- **Caso recursivo.** Se divide el problema en subproblemas, se resuelve cada problema recursivamente, y por último se combinan todas las soluciones en una solución al problema general.

1.4.2. Trabajo y profundidad de algoritmos Divide and Conquer.

Bajo la estructura de los algoritmos de Divide and Conquer, podemos plantear el trabajo y profundidad para un problema de tamaño n :

- $W(n) = Wdividir(n) + \sum_{i=1}^k W(n_i) + Wcombinar(n)$
- $S(n) = Sdividir(n) + \max_{i=1}^k S(n_i) + Scombinar(n)$

1.4.3. Algoritmo MergeSort.

El algoritmo **MergeSort** de ordenación de listas es un ejemplo de aplicación de la estrategia Divide and Conquer:

1. Divide la lista en dos sublistas.
2. Ordena las sublistas recursivamente.
3. Junta los resultados: lista completamente ordenada.

Pseudocódigo del algoritmo.

```
1 msort : [Int] -> [Int]
2 msort [] = []
3 msort [x] = [x]
4 msort xs = let
5     (ls, rs) = split xs
6     (ls', rs') = (msort ls || msort rs)
7     in
8     merge(ls', rs')
```

```
1 split : [Int] -> [Int] x [Int]
2 split [] = ([], [])
3 split [x] = ([x], [])
4 split (x:y:zs) = let
5     (xs, ys) = split zs
6     in
7     (x:xs, y:ys)
```

```
1 merge : [Int] x [Int] -> [Int]
2 merge ([], ys) = ys
3 merge (xs, []) = xs
4 merge (x:xs, y:ys) = if x <= y
5     then x:merge(xs, y:ys)
6     else y:merge(x:xs, ys)
```

Trabajo del algoritmo.

- $W_{msort}(0) = c_0$
- $W_{msort}(1) = c_1$
- $W_{msort}(n) = W_{split}(n) + 2W_{msort}(\frac{n}{2}) + W_{merge}(n) + c_2$
- $W_{split}(0) = c_3$
- $W_{split}(1) = c_4$
- $W_{split}(n) = W_{split}(n-2) + c_5$
- $W_{merge}(0) = c_6$
- $W_{merge}(n) = W_{merge}(n-1) + c_7$

Veamos que $W_{split}(n) \in O(n)$ y $W_{merge}(n) \in O(n)$. Luego juntando estos dos trabajos y resumiéndolos con el valor n que multiplica a la constante, resulta para la función $msort$:

$$W_{msort}(n) = 2W_{msort}\left(\frac{n}{2}\right) + c_3n \in O(n \lg n)$$

Profundidad del algoritmo.

- $Smsort(0) = k_0$
- $Smsort(1) = k_1$
- $Smsort(n) = Ssplit(n) + Smsort(\frac{n}{2}) + Smerge(n) + k_2$
- $Ssplit(0) = k_3$
- $Ssplit(1) = k_4$
- $Ssplit(n) = Ssplit(n - 2) + k_5$
- $Smerge(0) = k_6$
- $Smerge(n) = Smerge(n - 1) + k_7$

Veamos que $Ssplit(n) \in O(n)$ y $Smerge(n) \in O(n)$. Luego juntando estas dos profundidades y resumiéndolas con el valor n que multiplica a la constante, resulta para la función $msort$:

$$Smsort(n) = Smsort\left(\frac{n}{2}\right) + k_3n \in O(n)$$

2. Unidad 2 - Resolución de Recurrencias.

2.1. Introducción.

Una recurrencia es una función definida en términos de sí misma, osea, una función definida recursivamente.

Hemos visto que al calcular el trabajo (W) y profundidad (S) de funciones recursivas, siempre surgen recurrencias. Lo ideal sería poder obtener una ley exacta para una recurrencia.

Por ejemplo, $f(0) = 1, f(n) = 2f(n-1)$ podemos resumirla a $f(n) = 2^n \forall n \in \mathbb{N}$. Sin embargo, la mayoría de veces no llegaremos a una expresión cerrada para una recurrencia. Esto no es problema, ya que nuestro objetivo es dar cotas para estas recurrencias, y a continuación se explican algunos métodos.

2.2. Método de sustitución.

2.2.1. Definición. Método de sustitución.

El **método de sustitución** consiste en los siguientes pasos:

1. Adivinar la forma de la solución.
2. Probar que la forma es correcta usando inducción matemática.

Es decir, debemos tratar de adivinar alguna cota que pueda valer para una recurrencia y luego verificarla usando inducción sobre los naturales.

2.2.2. Ejemplo. Sustitución para el mergesort.

Recordemos la recurrencia de trabajo del mergesort:

- $W(0) = c_0$
- $W(1) = c_1$
- $W(n) = 2W(\lfloor \frac{n}{2} \rfloor) + c_2n$

Supongamos que sospechamos que vale $W(n) \in O(n \lg(n))$. Para probarlo debemos ver que existen $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tales que $0 \leq W(n) \leq cn \lg(n) \forall n \in \mathbb{N}$.

- **Caso inductivo.** Supongamos que existe $c \in \mathbb{R}^+$ tal que $W(k) \leq ck \lg(k), k < n$ (HI).

$$\begin{aligned} W(n) &= 2W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c_2n \stackrel{HI}{\leq} 2\left(c\left\lfloor \frac{n}{2} \right\rfloor \lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) + c_2n \\ &\leq 2c \frac{n}{2} \lg\left(\frac{n}{2}\right) + c_2n \\ &= cn(\lg_2(n) - \lg_2(2)) + c_2n \\ &= cn(\lg(n) - 1) + c_2n \\ &= cn\lg(n) - cn + c_2n \\ &\leq cn\lg(n) \end{aligned}$$

Y la última igualdad ocurre si $-cn + c_2n \leq 0 \Rightarrow -c + c_2 \leq 0 \Rightarrow c_2 \leq c$. Luego, el caso inductivo vale siempre que tomemos $\boxed{c \geq c_2}$.

- **Caso base.**

- **n = 0:** Queremos ver si $W(0) \leq c \cdot 0 \cdot \lg(0)$. Pero veamos que $\lg(0)$ no está definido, por lo tanto no vale este caso base.
- **n = 1:** Queremos ver si $W(1) \leq c \lg(1)$. Veamos que $\lg(1) = 0$, por lo tanto no existirá constante c tal que valga la desigualdad.

No valen los casos bases para 0 y 1, pero recordemos que la notación big O solo nos pide que la desigualdad comience a valer a partir de un n_0 . Probemos a partir de 2 y 3: usamos dos casos bases porque el piso $\lfloor \frac{n}{2} \rfloor$ recae en 0 o 1, pero como no valen recaerán en 2 y 3.

- **n = 2:** Queremos ver si $W(2) \leq 2c \lg(2)$.

$$\begin{aligned} W(2) &= 2W\left(\left\lfloor \frac{2}{2} \right\rfloor\right) + 2c_2 = 2W(1) + 2c_2 \\ &= 2c_1 + 2c_2 \\ &= 2(c_1 + c_2) \end{aligned}$$

Luego debemos escoger c tal que $W(2) = 2(c_1 + c_2) \leq 2c \lg(2) = 2c$. Es decir, un c tal que $2(c_1 + c_2) \leq 2c \Rightarrow \boxed{c_1 + c_2 \leq c}$, para que valga el caso base para $n = 2$.

- **n = 3:** Queremos ver si $W(3) \leq 3c \lg(3)$.

$$\begin{aligned} W(3) &= 2W\left(\left\lfloor \frac{3}{2} \right\rfloor\right) + 3c_2 = 2W(1) + 3c_2 \\ &= 2c_1 + 3c_2 \end{aligned}$$

Luego debemos escoger c tal que $W(3) = 2c_1 + 3c_2 \leq 3c \lg(3)$, es decir, un c tal que

$$\boxed{\frac{2c_1 + 3c_2}{3\lg(3)} \leq c}$$

Tomando todos los casos bases y el inductivo, tenemos que tomando:

$$c \geq \max\left\{c_2, c_1 + c_2, \frac{2c_1 + 3c_2}{3\lg(3)}\right\}$$

concluimos que $\forall n \geq 2, W(n) \leq cn \lg(n)$ y por lo tanto, $W(n) \in O(n \lg(n))$.

2.2.3. Acerca de la adivinanza.

Como adivinanza de una solución $O(n \lg(n))$ utilizamos la función $f(n) = cn \lg(n)$ para una constante arbitraria c . Es obvio que $f(n) \in O(n \lg(n))$ pero podemos utilizar cualquier función que pertenezca a $O(n \lg(n))$ para las pruebas. Por ejemplo:

- $f(n) = cn \lg(n) + c_1 n$
- $f(n) = cn \lg(n) - c_2$
- $f(n) = cn \lg(n) - c_1 n + c_2$.

El sumar o restar elementos de menor orden puede ser crucial para poder terminar la prueba.

2.3. Árboles de recurrencia.

2.3.1. Definición. Árbol de recurrencia.

Esta técnica es útil para hallar estimaciones o posibles cotas a una recurrencia. Este método consiste en dibujar un árbol, donde en el nodo raíz se tiene el costo de la recurrencia para la entrada n y hay una rama por cada llamada recursiva.

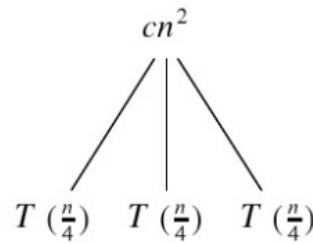
Cada rama de una llamada recursiva, a su vez, se expande en su propio árbol, y se procede así hasta llegar a un caso base, en el cual se deja de expandir el árbol. Por cada nivel del árbol se suma el total de operaciones, y estos a su vez se suman para dar el costo total de la recurrencia.

2.3.2. Ejemplo. Árbol de recurrencia.

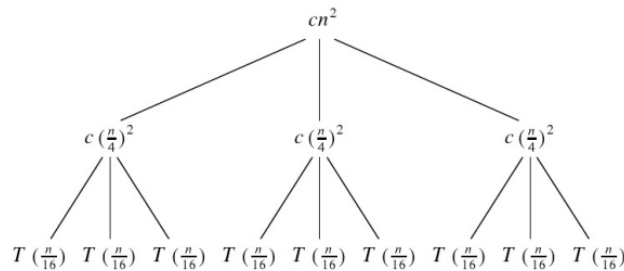
Consideremos la siguiente recurrencia:

- $T(1) = c_1$
- $T(n) = 3T(\frac{n}{4}) + cn^2$

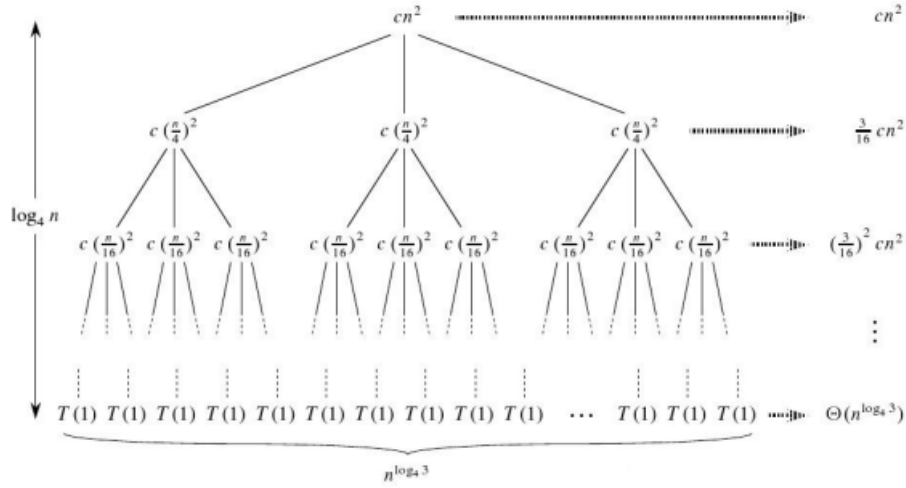
Hacemos un árbol que en su raíz tiene el costo para el n inicial, con ramas indicando cada una de las llamadas recursivas:



Las ramas correspondientes a las llamadas recursivas se expanden, ahora tienen el costo correspondiente a cada llamada y generan nuevas hojas:



Se sigue expandiendo el árbol hasta llegar a un caso base ($T(1)$ por ejemplo). En cada nivel se suma el total de operaciones por nivel. El costo total de la recurrencia es la suma de todos los niveles.



En este ejemplo, el nivel 1 suma cn^2 a la suma total. El segundo nivel suma $\frac{3}{16}cn^2$, el tercer nivel $(\frac{3}{16})^2 cn^2$.

Luego, tenemos $\log_4(n)$ niveles y $n^{\log_4(3)}$ hojas. Por lo tanto, el último nivel es de orden $O(n^{\log_4(3)})$.

Finalmente, la suma obtenida se manipula algebraicamente para llegar al resultado. Si somos prolijos, el método nos da una solución exacta, si no, al menos nos da un candidato para usar en el método de sustitución:

$$\begin{aligned}
 T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4(n)-1} cn^2 + kn^{\log_4(3)} \\
 &= \sum_{i=0}^{\log_4(n)-1} \left(\frac{3}{16}\right)^i cn^2 + kn^{\log_4(3)} \\
 &= \frac{\left(\frac{3}{16}\right)^{\log_4(n)} - 1}{\left(\frac{3}{16} - 1\right)} cn^2 + kn^{\log_4(3)} \\
 &\in O(n^2)
 \end{aligned}$$

Luego de obtener la cota con el árbol de recurrencia se debe utilizar el método de sustitución para probarlo.

Observación. Si en una definición de recurrencia no se menciona caso base, suponer que es una constante, por ejemplo, $T(1) = c_1$.

2.4. Funciones suaves y reglas de suavidad.

2.4.1. Introducción.

Si recurrimos al planteo sobre el trabajo del mergesort, podremos notar que se omitieron los pisos ($\lfloor \cdot \rfloor$) y techos ($\lceil \cdot \rceil$), siendo que ambos corresponden ya que n podría ser impar, quedando una mitad de la lista de longitud $\lfloor \frac{n}{2} \rfloor$ y la otra de longitud $\lceil \frac{n}{2} \rceil$.

Entonces, ¿cuándo es válido omitir los pisos y techos? Podríamos omitirlos si $n = b^k$ (n es una potencia de b), así entonces $\frac{n}{b} = \lfloor \frac{n}{b} \rfloor = \lceil \frac{n}{b} \rceil$.

Notar que no alcanza con que n sea múltiplo de b , debe ser **potencia**.

2.4.2. Definición. Función eventualmente no decreciente.

Una función $f : \mathbb{N} \rightarrow \mathbb{R}^+$ es **eventualmente no decreciente** si:

$$\exists N \in \mathbb{N} : f(n) \leq f(n+1) \quad \forall n \geq N$$

Es decir, es una función que a partir de cierto N no decrece: se mantiene constante o aumenta nomás.

2.4.3. Definición. Función b-suave.

Una función $f : \mathbb{N} \rightarrow \mathbb{R}^+$ es **b-suave** si:

- f es eventualmente no decreciente y
- $f(bn) \in O(f(n))$

Esto quiere decir que el crecimiento de f está acotado y por lo tanto no crece exponencialmente rápido.

2.4.4. Definición. Función suave.

Una función es **suave** si es b-suave para todo $b \in \mathbb{N}$.

2.4.5. Proposición. Condición suficiente de suavidad.

Si f es b-suave para un $b \geq 2$, entonces es suave para todo b .

Ejemplos. Las funciones $n^2, n^r, n \lg(n)$ son funciones suaves. Las funciones $n^{\lg(n)}, 2^n, n!$ no lo son.

2.4.6. Teorema. Regla de suavidad.

Sea f una función suave y sea g eventualmente no decreciente, entonces para todo $b \geq 2$:

$$g(b^k) \in \Theta(f(b^k)) \Rightarrow g(n) \in \Theta(f(n))$$

Observación. Esta regla nos sirve para eliminar los pisos y techos en las recurrencias, pues si en vez de considerar n cualquiera, solamente consideramos potencias de b , es decir $n = b^k$, entonces podemos omitir pisos y techos de nuestro análisis, ya que resulta $\frac{n}{b} = \lfloor \frac{n}{b} \rfloor = \lceil \frac{n}{b} \rceil$.

Luego, la regla de suavidad nos dice que si para nuestro análisis consideramos exclusivamente a potencias de b , podemos omitir pisos y techos; y si el resultado al que llegamos es una función suave, entonces nuestro análisis es correcto para cualquier entrada.

2.4.7. Ejemplo. Recurrencia del mergeSort.

Podemos ahora volver al caso del mergeSort. La verdadera recurrencia que surge del trabajo del mergeSort es:

$$Wmsort(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 & \text{si } n = 1 \\ Wmsort\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + Wmsort\left(\left\lceil \frac{n}{2} \right\rceil\right) + Wsplit(n) + Wmerge(n) & \text{si } n \geq 2 \end{cases}$$

Probaremos primero que $Wmsort(n)$ es eventualmente no decreciente, para luego, con regla de suavidad, eliminar los pisos y techos.

Lema: $Wmsort(n)$ es eventualmente no decreciente.

Demostración.

Probaremos por inducción sobre los naturales. Suponemos como HI que vale $Wmsort(k) \leq Wmsort(k+1)$ con $k+1 \leq n$. Luego:

$$\begin{aligned} Wmsort(n) &= Wmsort\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + Wmsort\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n) \\ &\stackrel{HI}{\leq} Wmsort\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + Wmsort\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + O(n) \\ &= Wmsort(n+1) \end{aligned}$$

Y vemos entonces que $Wmsort(n) \leq Wmsort(n+1)$. Como caso base podemos ver que $Wmsort(2) \geq Wmsort(1)$ y concluimos que $Wmsort(n)$ es eventualmente no decreciente, como queríamos probar. \square

Siguiendo con la recurrencia de $Wmsort(n)$, notemos que si $n = 2^k$, entonces podemos eliminar pisos y techos y tenemos que:

$$\begin{aligned} Wmsort(n) &= Wmsort\left(\frac{n}{2}\right) + Wmsort\left(\frac{n}{2}\right) + Wsplit(n) + Wmerge(n) \\ &= 2Wmsort\left(\frac{n}{2}\right) + Wsplit(n) + Wmerge(n) \\ &= 2Wmsort\left(\frac{n}{2}\right) + O(n) \\ &\in O(n \lg(n)) \end{aligned}$$

Luego, como la función $f(n) = n \lg(n)$ es suave y $Wmsort(n)$ es eventualmente no decreciente, entonces por la regla de suavidad podemos concluir que $Wmsort(n) \in O(n \lg(n))$.

2.5. Teorema maestro.

2.5.1. Definición. Teorema maestro.

Dados $a \geq 1, b \geq 1$ y la recurrencia:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

entonces tenemos las siguientes cotas:

$$T(n) \in \begin{cases} \Theta(n^{\log_b(a)}) & \text{si} & \exists \epsilon > 0 : \boxed{f(n) \in O(n^{\log_b(a)-\epsilon})} \\ \Theta(n^{\log_b(a)} \lg(n)) & \text{si} & \boxed{f(n) \in \Theta(n^{\log_b(a)})} \\ \Theta(f(n)) & \text{si} & \begin{aligned} & \exists c > 0 : \boxed{f(n) \in \Omega(n^{\log_b(a)+\epsilon})} \\ & \wedge \exists c < 1, N \in \mathbb{N} : \boxed{af\left(\frac{n}{b}\right) \leq cf(n)} \forall n > N \end{aligned} \end{cases}$$

Observaciones.

- El teorema maestro nos indica que término de la recurrencia tiene más peso asintótico en la ecuación. Los casos se deciden comparando $f(n)$ con $n^{\log_b(a)}$:
 1. Caso 1: $n^{\log_b(a)}$ es de mayor orden.
 2. Caso 2: $f(n)$ y $n^{\log_b(a)}$ tienen el mismo orden.
 3. Caso 3: $f(n)$ es de mayor orden.
- No basta con que las funciones sean una más grande que la otra, sino que debe ser polinomialmente más grande.
- Los 3 casos del teorema maestro cubren todas las posibilidades.

Teorema maestro reducido. Este teorema maestro sirve cuando en el término por fuera de la recurrencia tenemos una potencia de n (no sirve para casos logarítmicos). Se debe utilizar para darnos un primer indicio, pues la prueba formal utiliza el teorema anterior.

Si tenemos un algoritmo cuya ecuación de recurrencia es:

$$T(n) = a T\left(\frac{n}{b}\right) + O(n^c)$$

entonces tenemos las siguientes cotas:

$$T(n) \in \begin{cases} \Theta(n^{\log_b(a)}) & \text{si } \log_b(a) > c \\ \Theta(n^c \lg(n)) & \text{si } \log_b(a) = c \\ \Theta(n^c) & \text{si } \log_b(a) < c \end{cases}$$

Ejemplo. Para $W(n) = 2W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c_2 n$ estamos en el 2do caso ya que $\log_2(2) = 1$ y $g(n) = n \in O(n^1)$, por lo tanto, $W(n) = \Theta(n \lg(n))$.

3. Unidad 3 - Programación Funcional con Haskell

3.1. Programación funcional.

3.1.1. ¿Qué es la programación funcional?

La **programación funcional** es un estilo de programación que no usa un modelo de computación basado en máquinas, sino en un lenguaje simple y elegante (el λ -cálculo).

En la programación funcional, el método básico de computar es **aplicar funciones a argumentos**.

3.1.2. Ventajas de Haskell.

Estaremos utilizando **Haskell** como lenguaje de programación funcional. Sus ventajas son:

- Programas concisos.
- Sistemas de tipos poderosos.
- Funciones recursivas.
- Facilidad para probar propiedades de programas.
- Funciones de alto orden.
- Evaluación perezosa.
- Facilidad para definir DSLs.
- Efectos monádicos.

3.2. Introducción a Haskell.

3.2.1. Guía inicial para compilar y ejecutar un archivo.

1. **Instalación de GHC y GHCi.** Para compilar y ejecutar programas en Haskell, se necesita instalar GHC (Glasgow Haskell Compiler).
2. **Escribir un archivo en Haskell.** Para escribir código Haskell, se usa un editor de texto como VSCode, Vim, Emacs o simplemente el Bloc de notas.

```
1 -- Definimos el modulo principal.
2 module Main where
3
4 -- Definiciones.
5 numero = 5
6 lista = [1, 2, 3]
```

3. Compilar y ejecutar un archivo haskell.

```
1 -- Compilar el archivo con ghci.
2 ghci archivo.hs
3
4 -- Ejecutar el modulo.
5 main
6
7 -- Mostrar variables.
8 numero
9 lista
10
11 -- Salir de ghci.
12 :quit
```

3.2.2. Comentarios y palabras reservadas.

Las palabras reservadas son: **case**, **class**, **data**, **default**, **deriving**, **do**, **else**, **if**, **import**, **in**, **infix**, **infixl**, **infixr**, **instance**, **let**, **module**, **newtype**, **of**, **then**, **type**, **where**.

Los comentarios se escriben con un doble guión (--) para comentarios en línea y con {- -} para un bloque de comentarios.

```
1 -- Comentario en línea
2 {-
3     Bloque de comentarios.
4     Util para comentarios en varias líneas.
5 -}
```

3.2.3. Offside rule.

En una serie de definiciones, cada definición debe empezar en la misma columna. Gracias a esta regla, no hace falta una sintaxis explícita para agrupar definiciones.

```
1 a = b + c
2   where
3     b = 1
4     c = 2
5 d = a + 2
```

3.2.4. Operadores infijos.

Los **operadores infijos** son funciones como cualquier otra, pero que se aplican entre medio de dos argumentos. Una función se puede hacer infija con backquotes:

$$10 \text{ 'div' } 4 = \text{div } 10 \ 4$$

Se pueden definir nuevos operadores infijos usando alguno de los símbolos disponibles.

$$a ** b = (a * b) + (a + 1) * (b - 1)$$

La asociatividad y precedencia se indica usando **infixr** (asociatividad derecha), **infixl** (asociatividad izquierda), o **infix** (si los paréntesis deben ser obligatorios).

$$\text{infixr } 6 \ (**)$$

3.2.5. Tipos.

Un **tipo** es un nombre para una colección de valores. Por ejemplo, *Bool* contiene los valores *True* y *False*. Escribimos entonces $\text{True} :: \text{Bool}$ y $\text{False} :: \text{Bool}$.

En general, si una expresión *e* tiene tipo *t* escribimos:

$$e :: t$$

En Haskell, toda expresión válida tiene un tipo. El tipo de cada expresión es calculado previo a su evaluación mediante la *inferencia de tipos*. Si no es posible encontrar un tipo (por ejemplo, $(\text{True} + 4)$), el compilador protestará con un *error de tipo*.

Algunos de los tipos básicos de Haskell son:

- *Bool*: booleanos.

- *Char*: caracteres.
- *Int*: enteros de precisión fija.
- *Integer*: enteros de precisión arbitraria.
- *Float*: números de punto flotante en precisión simple.

3.3. Listas en Haskell.

3.3.1. Definición. Lista.

Una **lista** es una estructura de datos que se utiliza para almacenar colecciones de elementos del mismo tipo.

En Haskell, las listas se declaran utilizando corchetes `[]` para delimitar los elementos de la lista, separados por comas.

En general, `[t]` es una lista con elementos de tipo `t`, donde `t` puede ser cualquier tipo válido.

Ejemplos.

```
1 listaBool :: [Bool]
2 listaBool = [True, True, False, True]
3
4 listaPalabra :: [Char]
5 listaPalabra = ['h', 'o', 'l', 'a']
6
7 listaEjemplo :: [[Char]]
8 listaEjemplo = [['a'], ['b', 'c'], []]
```

Observaciones.

- No hay restricción con respecto a la longitud de las listas.
- Por convención, los argumentos de las listas usualmente tienen como sufijo la letra *s* para indicar que pueden contener múltiples valores.
 - `ns`: lista de números.
 - `xs`: lista de valores arbitrarios.
 - `xss`: lista de lista de caracteres.

3.3.2. Operaciones básicas con listas.

```
1 -- Concatenación: se utiliza el operador (++).
2 lista1 ++ lista2
3
4 -- Agregar elemento al principio de la lista (cons). Se utiliza el operador (:).
5 1 : [2, 3, 4, 5]
6
7 -- Obtener longitud de lista: se utiliza la función (length) que devuelve la
  longitud de una lista.
8 length [1, 2, 3, 4, 5]
9
10 -- Acceder según índice: se utiliza la función infija <lista> !! <índice>. Así
   obtendremos el elemento de índice indicado, en la lista argumento.
11 -- El siguiente código devuelve el tercer elemento (índice 2)
12 [1, 2, 3, 4, 5] !! 2
```

3.3.3. Funciones de listas.

Haskell proporciona una serie de funciones predefinidas para trabajar con listas.

```
1 -- HEAD: devuelve el primer elemento de una lista
2 lista = [1, 2, 3, 4, 5]
3 primerElemento = head lista
4 -- primerElemento = 1
5
6 -- TAIL: devuelve todos los elementos de una lista excepto el primero.
7 lista = [1, 2, 3, 4, 5]
8 restoLista = tail lista
9 -- restoLista = [2, 3, 4, 5]
10
11 -- LAST: devuelve el ultimo elemento de una lista.
12 lista = [1, 2, 3, 4, 5]
13 ultimoElemento = last lista
14 -- ultimoElemento = 5
15
16 -- INIT: devuelve todos los elementos de una lista excepto el ultimo.
17 lista = [1, 2, 3, 4, 5]
18 sinUltimoElemento = init lista
19 -- sinUltimoElemento = [1, 2, 3, 4]
20
21 -- NULL: comprueba si una lista esta vacia.
22 listaVacía = []
23 estaVacía = null listaVacía
24 -- estaVacía = True
25
26 -- REVERSE: invierte una lista.
27 lista = [1, 2, 3, 4, 5]
28 listaInvertida = reverse lista
29 -- listaInvertida = [5, 4, 3, 2, 1]
30
31 -- TAKE: toma los primeros n elementos de una lista.
32 lista = [1, 2, 3, 4, 5]
33 primerosTresElementos = take 3 lista
34 -- primerosTresElementos = [1, 2, 3]
35
36 -- DROP: elimina los primeros n elementos de una lista.
37 lista = [1, 2, 3, 4, 5]
38 sinPrimerosTresElementos = drop 3 lista
39 -- sinPrimerosTresElementos = [4, 5]
40
41 -- ELEM: comprueba si un elemento esta presente en una lista.
42 lista = [1, 2, 3, 4, 5]
43 estaElTres = elem 3 lista
44 -- estaElTres = True
45
46 -- FILTER: filtra los elementos de una lista segun un predicado.
47 lista = [1, 2, 3, 4, 5]
48 pares = filter even lista
49 -- pares = [2, 4]
50
51 -- MAP: aplica una funcion a cada elemento de una lista.
52 lista = [1, 2, 3, 4, 5]
53 cuadrados = map (\x -> x^2) lista
54 -- cuadrados = [1, 4, 9, 16, 25]
55
56 -- FOLDR/FOLD: realiza un plegado sobre una lista, combinando los elementos usando
    una funcion. Toma una funcion (suma), un valor inicial (0) y una lista. Aplica
    la funcion acumulativa a cada elemento de la lista, partiendo desde el lado
    derecho (r en foldr). La suma acumulada se inicia desde 0.
57 lista = [1, 2, 3, 4, 5]
58 suma = foldr (+) 0 lista
59 -- suma = 15
```

3.3.4. Listas por comprensión.

Haskell permite construir listas de forma concisa mediante la especificación de reglas de generación y filtros, es decir, generar listas por comprensión. La notación para generar listas por comprensión es:

$$[x^2 \mid x \leftarrow [1 \dots 5]]$$

Esta última sentencia genera la lista $[1, 4, 9, 16, 25]$. Notar que la expresión $x \leftarrow [1 \dots 5]$ es un *generador*, ya que dice como se generan los valores de x .

Ejemplos.

```
1 -- Genera una lista de los cuadrados del 1 al 5.
2 cuadrados = [x^2 | x <- [1..5]]
3 -- [1,4,9,16,25]
4
5 -- Una lista por comprension puede tener varios generadores, separados por coma.
6 [(x, y) | x <- [1,2,3], y <- [4,5]]
7 -- [(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
8
9 -- Un generador puede depender de un generador anterior. A estos los llamamos
   GENERADORES DEPENDIENTES.
10 -- La siguiente es la lista de todos los pares (x,y) tal que x, y estan en [1 .. 3]
   e y >= x.
11 [(x,y) | x <- [1..3], y <- [x..3]]
12 -- [(1, 1), (1,2), (1,3), (2,2), (2,3), (3,3)]
```

3.3.5. Patrones de listas.

Toda lista (no vacía) se construye usando el operador $(:)$ llamado cons, que agrega un elemento al principio de la lista.

$$[1, 2, 3, 4] = 1 : (2 : (3 : (4 : [])))$$

Por lo tanto, se pueden definir funciones usando el patrón $(x:xs)$. Este patrón solo matchea en el caso de listas no vacías. Por lo tanto, la función head con una lista vacía arrojaría un error

```
1 head :: [a] -> a
2 head (x:_) = x
3
4 tail :: [a] -> [a]
5 tail (_:xs) = xs
```

3.4. Tuplas en Haskell.

3.4.1. Definición. Tupla.

Una **tupla** es una secuencia **finita** de valores de tipos (posiblemente) distintos. A diferencia de las listas, las tuplas tienen un tamaño fijo y pueden contener elementos de tipos diferentes.

En Haskell, las tuplas se declaran utilizando paréntesis $()$ para delimitar los elementos de la tupla, separados por comas.

En general, (t_1, t_2, \dots, t_n) es el tipo de una n -tupla cuyas componente i tiene tipo t_i .

Ejemplos.

```

1 -- miTupla :: (Bool, Bool)
2 miTupla = (True, True)
3
4 -- miTupla2 :: (Bool, Char, Char)
5 miTupla2 = (True, 'a', 'b')
6
7 -- miTupla3 :: (Char, (Bool, Char))
8 miTupla3 = ('a', (True, 'c'))
9
10 -- miTupla4 :: ([[Char], Char], Char)
11 miTupla4 = ((['a', 'b'], 'a'), 'b')

```

3.4.2. Acceso a los elementos.

En Haskell no se puede acceder directamente a los elementos de una tupla como lo haríamos en otros lenguajes utilizando los índices. En su lugar, debemos utilizar *pattern matching* o funciones predefinidas.

```

1 -- Definicion de nuestra tupla.
2 miTupla = ('a', 24, True)
3
4 -- Definimos las funciones.
5 primero (x, _, _) = x
6 segundo (_, y, _) = y
7 tercero (_, _, z) = z
8
9 -- Usando funciones predefinidas. Validas para tuplas con dos elementos nomas.
10 miTupla2 = ('a', True)
11 fst miTupla2
12 snd miTupla2

```

3.5. Strings en Haskell.

Un **string** es una lista de caracteres.

```

1 "Hola" :: String
2 "Hola" = ['H', 'o', 'l', 'a']

```

Por lo tanto, todas las funciones sobre listas son aplicables a Strings.

```

1 cantminusc :: String -> Int
2 cantminusc xs = length[x | x <- xs, isLower x]

```

3.6. Expresiones condicionales en Haskell.

3.6.1. If-then-else.

Las expresiones condicionales son una forma de control de flujo que permite ejecutar diferentes bloques de código según una condición específica. En Haskell se realizan a través de **if-then-else**.

Para que la expresión condicional tenga sentido, ambas ramas de la misma deben tener el mismo tipo. Además, siempre deben tener la rama **else** para que no haya ambigüedades en caso de anidamiento.

Ejemplos.

```

1 esPositivo :: Int -> Bool
2 esPositivo x = if x > 0 then True else False
3
4 abs :: Int -> Int
5 abs n = if n >= 0 then n else -n
6
7 signum :: Int -> Int
8 signum n = if n < 0 then -1 else
9             if n == 0 then 0 else 1

```

3.6.2. Ecuaciones con guardas.

Una alternativa a los condicionales es el uso de **ecuaciones con guardas**. En este caso, '|' se utiliza para definir múltiples casos y 'otherwise' se utiliza como un patrón de 'cualquier otro caso'.

Las guardas se evalúan en orden, si una guarda se evalúa como True, se ejecuta la expresión correspondiente. **Otherwise = True** según el prelude.

```

1 esPositivo :: Int -> Bool
2 esPositivo x
3   | x > 0      = True
4   | otherwise = False

```

3.6.3. Pattern Matching. Coincidencia de patrones.

El **pattern matching** es una característica que permite escribir funciones que se comportan de manera diferente según la forma de los datos de entrada. En esencia, el pattern matching permite descomponer estructuras de datos y tomar decisiones basadas en la forma de esas estructuras.

El pattern matching se utiliza principalmente en la definición de funciones para manejar diferentes casos según los valores de entrada. Se puede aplicar a listas, tuplas, tipos algebraicos y otros tipos de datos. (Es como definir una función por partes en matemática).

Los patrones de coincidencia se evalúan en orden. Si el primer patrón es cierto, se accede a él y termina la función. El valor (-) representa cualquier argumento, se suele poner al final de los patrones para usarlo como un *else*.

Ejemplos.

```

1 -- Funcion not booleano.
2 not :: Bool -> Bool
3 not False = True
4 not True = False
5
6 -- Funcion que determina si un entero es 0.
7 esCero :: Int -> Bool
8 esCero 0 = True
9 esCero _ = False
10
11 -- Funcion que devuelve la longitud de una lista de cualquier tipo.
12 longitud :: [a] -> Int
13 longitud [] = 0
14 longitud (x:xs) = 1 + longitud xs
15
16 -- Funcion AND condicional.
17 (and) :: Bool -> Bool -> Bool
18 True (and) True = True
19 True (and) False = False
20 False (and) True = False
21 False (and) False = False
22
23 -- Funcion AND condicional resumida.

```



```

24 (and) :: Bool -> Bool -> Bool
25 True (and) True = True
26 _ (and) _ = False
27
28 -- Funcion que devuelve si un dia pertenece al fin de semana.
29 data DiaSemana = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
30 esFinDeSemana :: DiaSemana -> Bool
31 esFinDeSemana Sabado = True
32 esFinDeSemana Domingo = True
33 esFinDeSemana _ = False

```

3.7. Funciones en Haskell.

3.7.1. Definición. Función.

Una **función** mapea valores de un tipo en valores de otro tipo. En general, una función de tipo $t_1 \rightarrow t_2$ mapea valores de tipo t_1 en valores de tipo t_2 .

Las funciones son fundamentales en Haskell pues son tratadas como ciudadanos de primera clase, lo que significa que pueden ser pasadas como argumentos a otras funciones, devueltas como resultados, y asignadas a variables.

3.7.2. Aplicación de funciones, asociatividad y precedencia.

- En Haskell la aplicación se denota con un espacio y asocia a la izquierda.

Matemáticas	Haskell
$f(x)$	$f\ x$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, (g(y)))$	$f\ x\ (g\ y)$
$f(x)\ g(y)$	$f\ x\ * \ g\ y$

- La aplicación tiene mayor precedencia que cualquier otro operador:

$$f\ x + y = (f\ x) + y$$

- Las funciones y sus argumentos deben empezar con minúscula, y pueden ser seguidos por cero o más letras (mayúsculas o minúsculas), dígitos, guiones bajos, y apóstrofes.

3.7.3. Declaración de funciones.

En Haskell, las funciones se definen utilizando la sintaxis:

$$f :: a \rightarrow b$$

donde f es el nombre de la función, a es el tipo de los argumentos de entrada y b el tipo del resultado de la función.

Ejemplos.

```

1 -- Recibe un entero y devuelve su doble.
2 doble :: Int -> Int
3 doble x = x * 2
4
5 -- Funcion que mapea valores de un tipo en otro.
6 not :: Bool -> Bool
7 isDigit :: Char -> Bool
8
9 -- Funciones con multiples argumentos usando tuplas y listas.
10 add :: (Int, Int) -> Int
11 add (x,y) = x + y
12
13 deceroA :: Int -> [Int]
14 deceroA n = [0 .. n]

```

3.7.4. Llamada a funciones.

Las funciones en Haskell se llaman simplemente escribiendo el nombre de la función seguido de los argumentos entre espacios. Por ejemplo:

```

1 resultado = doble 5
2 -- resultado = 10

```

3.7.5. Currying. Currificación y aplicación parcial.

El **currying** es una técnica en la que una función que toma múltiples argumentos se convierte en una secuencia de funciones que toman un solo argumento.

En Haskell, todas las funciones son curriadas de forma predeterminada, lo que significa que podemos aplicar parcialmente una función para obtener una nueva función que espera el resto de los argumentos.

Ejemplo.

La función suma toma dos argumentos enteros y devuelve su suma. Sin embargo, en Haskell, también podemos entender esta función como una función que toma un solo argumento y devuelve otra función que toma el segundo argumento. Entonces podemos reescribir la definición de suma:

```

1 suma :: Int -> Int -> Int
2 suma x y = x + y
3
4 suma :: Int -> (Int -> Int)
5 suma x = \y -> x + y

```

Ahora, podemos aplicar parcialmente la función suma y definir funciones como la siguiente:

```

1 sumaDos :: Int -> Int
2 sumaDos = suma 2
3
4 resultado = sumaDos 3
5 -- resultado = 5

```

3.7.6. Variables locales en funciones.

■ Claúsula where.

La cláusula **where** se utiliza para definir nombres locales dentro de una función. Estos nombres locales son visibles únicamente dentro de la función en la que se definen.

Es útil para evitar la repetición de cálculos y para mejorar la legibilidad del código al encapsular detalles de implementación dentro de la función donde son relevantes.

```
1 areaCirculo :: Double -> Double
2 areaCirculo radio = pi * radioCuadrado
3   where
4     radioCuadrado = radio * radio
```

■ Claúsulas let/in.

Las palabras clave **let** e **in** también se utilizan para definir nombres locales dentro de una expresión.

Let se utiliza para introducir definiciones locales. Se pueden definir uno o más nombres locales, cada uno seguido por una expresión que le asigna un valor.

In se utiliza para separar las definiciones locales de la expresión principal. Indica dónde comienza la expresión principal que utiliza los nombres locales previamente definidos.

```
1 areaCirculo :: Double -> Double
2 areaCirculo radio =
3   let radioCuadrado = radio * radio
4   in pi * radioCuadrado
```

3.7.7. Recursión.

La **recursión** o **función recursiva** es una función que en su cuerpo/definición se llama a sí misma.

Este tipo de funciones tienen una entrada de cierto tamaño y en su llamado recursivo, se llaman a sí mismas con un tamaño menor de entrada. A su vez, cuentan con un caso base que hará que la función finalice.

Esta técnica se utiliza en Haskell para reemplazar a los bucles for. Haskell admite la recursión tanto en funciones como en estructuras de datos.

Ejemplos.

```
1 -- Calcular factorial de un numero.
2 factorial :: Int -> Int
3 factorial 0 = 1
4 factorial n = n * factorial (n - 1)
5
6 -- Calcular size de una lista.
7 length :: [a] -> Int
8 length [] = 0
9 length (x:xs) = 1 + length xs
```

3.7.8. Funciones Lambda. Funciones anónimas.

Se pueden construir funciones sin darles nombres usando **expresiones lambda**. La sintaxis de una función lambda es:

$$\lambda x \rightarrow x + x$$

En Haskell escribimos el símbolo λ como una barra (`\`), x es el argumento de la función y la expresión luego de la flecha es el cuerpo de la función.

Ejemplos.

```

1 -- Funcion que suma dos numeros.
2 suma :: Int -> Int -> Int
3 suma = \x y -> x + y
4
5 -- Multiplicacion de 3 numeros.
6 \x y z -> x * y * z
7
8 -- Evitamos darle nombre a una funcion que se usa una vez.
9 impares n = map f [0 .. n - 1]
10   where f x = x * 2 + 1
11
12 odds n = map (\x -> x * 2 + 1) [0 .. n - 1]

```

3.7.9. Polimorfismo paramétrico en funciones.

El **polimorfismo** en Haskell se refiere a la capacidad de una función de trabajar con múltiples tipos de datos de manera genérica.

Una función es **polimórfica** si su tipo contiene variables de tipo. Es decir, es una función que puede trabajar con múltiples tipos de datos. Puede tomar argumentos de diferentes tipos y realizar las mismas operaciones independientemente del tipo de datos que reciba.

Ejemplos.

```

1 -- Para cualquier lista de cualquier tipo a, la funcion length que calcula la
   cantidad de elementos de una lista es la misma.
2 length :: [a] -> Int
3
4 -- Funcion polimorfica que intercambia los elementos de una tupla.
5 intercambiar :: (a, b) -> (b, a)
6 intercambiar (x, y) = (y, x)
7
8 intercambiar (1, "Hola")
9 -- ("Hola", 1)
10
11 intercambiar ("Mundo", True)
12 -- (True, "Mundo")

```

3.7.10. Polimorfismo ad-hoc de sobrecarga de funciones.

Este tipo de polimorfismo es una característica que permite definir múltiples versiones de una función u operador con el mismo nombre, pero de distintos tipos de argumentos.

Por ejemplo, la función suma (+) permite sumar Ints, Floats y otros tipos numéricos. Si vemos el tipo, tenemos que:

$$(\+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

Es decir, la función suma está definida para cualquier tipo que sea una *instancia* de la *clase* Num. A diferencia del polimorfismo paramétrico, hay una definición distinta de la función (+) para cada instancia.

En Haskell, los números y operaciones aritméticas están sobrecargadas. Por ejemplo, ¿cuál es el tipo de $3 + 2$? Podría tomar dos Ints, dos Floats, o generalizando, dos Num.

3.8. Clases de tipo.

3.8.1. Definición. Clases de tipos.

Las **clases de tipo** son un mecanismo que permite definir interfaces de tipos y proporcionar implementaciones para esas interfaces. Se utilizan para agregar restricciones sobre los tipos de datos que pueden ser utilizados con ciertas funciones u operadores.

Una clase de tipo define un conjunto de funciones o métodos que deben ser implementados para que un tipo de datos sea considerado miembro de esa clase. Esas funciones se conocen como 'métodos de clase/tipo'.

Cuando se declara una instancia de una clase de tipo para un tipo de datos específico, se proporcionan implementaciones concretas para los métodos de clase definidos en la clase de tipo.

3.8.2. Algunas clases de tipo.

- **Clase EQ.** La clase de tipo **EQ** define la interfaz para la igualdad entre valores. A esta clase van a pertenecer los tipos de datos que pueden ser comparables (las funciones no pueden ser comparables entonces no pertenecen).

Para que un tipo de datos sea miembro de la clase **eq**, debe proporcionar una implementación de la función `'=='` (igualdad) y `'/=='` (desigualdad).

```
1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
```

- **Clase ORD.** La clase de tipo **ORD** son los tipos que además de ser instancias de **EQ** poseen un orden total:

```
1 class Ord a where
2   (<), (<=), (>), (>=) :: a -> a -> Bool
3   min, max :: a -> a -> a
```

- **Clase SHOW.** Son los tipos de datos cuyos valores pueden ser convertidos en una cadena de caracteres.

```
1 class Show a where
2   show :: a -> String
```

- **Clase READ.** Read es la clase dual. Son los tipos de datos que se pueden obtener de una cadena de caracteres.

```
1 class Read a where
2   read :: String -> a
```

- **Clase NUM.** Son los tipos de datos que son numéricos (Int, Integer, Float, etc) Sus instancias deben implementar las funciones:

```
1 class Num a where
2   (+), (-), (*) :: a -> a -> a
3   negate, abs, signum :: a -> a
```

- **Clase INTEGRAL.** Son los tipos de datos que son Num y además implementan:

```
1 class Integral a where
2   div, mod :: a -> a -> a
```

- **Clase FRACTIONAL.** Son los tipos de datos que son Num y además implementan:

```
1 class Fractional a where
2   (/) :: a -> a -> a
3   recip :: a -> a
```

3.9. Ejemplos importantes.

3.9.1. Función zip.

La función **zip** mapea dos listas a una lista con los pares de elementos correspondientes:

```
1 zip :: [a] -> [b] -> [(a,b)]
2 --> zip ['a', 'b', 'c'] [1,2,3,4] -> [('a',1),('b',2), ('c',3)]
```

Ejemplos usando función zip.

```
1 -- Funcion que toma una lista y devuelve una lista de pares de los elementos
   -- adyacentes de la lista.
2 pairs :: [a] -> [(a,a)]
3 pairs xs = zip xs (tail xs)
4 -- pairs [1,2,3,4] -> [(1,2),(2,3),(3,4)]
5
6 -- Funcion que toma una lista y devuelve True si esta ordenada; False en caso
   -- contrario.
7 -- Genera una lista de pares con los elementos de la lista, y por cada par pregunta
   -- si x <= y, en tal caso, tendremos una lista de Booleanos [Bool] a la cual se
   -- le aplica un and generalizado.
8 sorted :: Ord a => [a] -> Bool
9 sorted xs = and [x <= y | (x,y) <- pairs xs]
10 -- sorted [5,1,2,3] -> False
11
12 -- Funcion que toma dos enteros y una lista, y devuelve los elementos de la lista
   -- desde el indice del primer argumento al otro argumento.
13 -- Genera una lista de pares entre la lista argumento y los naturales del 0 a n,
   -- que simulan ser indices, para compararlos con i.
14 rangeof :: Int -> Int -> [a] -> [a]
15 rangeof low hi xs = [x | (x,i) <- zip xs [0..], i >= low, i <= hi]
16 -- rangeof 2 4 [1,2,3,4,5,6,7,8] -> [3,4,5]
```

3.9.2. Notación polaca.

4. Unidad 4 - Tipos en Haskell

4.1. Tipos en Haskell.

- **Sistema de tipos expresivo:** Haskell permite definir tipos de datos de manera flexible y expresiva.
- **Tipado estático:** todos los valores y expresiones tienen un tipo conocido en tiempo de compilación. Esto significa que el tipo de cada expresión se determina antes de que se ejecute el programa, lo que ayuda a detectar errores de tipo en etapas tempranas.
- **Inferencia de tipos:** el compilador puede inferir automáticamente muchos tipos basados en el contexto del programa. Sin embargo, dar el tipo de las funciones es ventajoso.

4.2. Sinónimos de tipo (type).

En Haskell se puede definir un nuevo nombre para un tipo existente usando una declaración **type**. Los sinónimos de tipo hacen que ciertas declaraciones de tipos sean más fáciles de leer.

Su declaración sigue la siguiente sintaxis:

type String = [Char]

Ejemplo.

```
1 type Pos = (Int, Int)
2
3 origen :: Pos
4 origen = (0,0)
5
6 izq :: Pos -> Pos
7 izq (x,y) = (x-1,y)
```

Además, los sinónimos de tipo pueden tener parámetros, pueden anidarse, pero no pueden ser recursivos.

```
1 -- Con parametros.
2 type Par a = (a,a)
3
4 copiar :: a -> Par a
5 copiar x = (x,x)
6
7 -- Anidados.
8 type Punto = (Int, Int)
9 type Trans = Punto -> Punto
10
11 -- No pueden ser recursivos.
12 type Tree = (Int, [Tree])
```

4.3. Declaraciones data.

Los **data** declaran un nuevo tipo cuyos valores se especifican en la declaración.

data Bool = False | True

En el ejemplo anterior, True y False son los **constructores** del tipo Bool. Dos constructores diferentes contruyen diferentes valores del tipo. Además, los nombres de los constructores deben empezar con mayúsculas.

Los valores de un nuevo tipo se usan igual que los predefinidos:

```

1 data Respuesta = Si | No | Desconocida
2
3 respuestas :: [Respuesta]
4 respuestas = [Si, No, Desconocida]
5
6 invertir :: Respuesta -> Respuesta
7 invertir Si = No
8 invertir No = Si
9 invertir Desconocida = Desconocida

```

Los constructores son en realidad funciones de complejidad constante $O(1)$. Por esto, pueden tener parámetros:

```

1 data Shape = Circle Float | Rect Float Float
2
3 square :: Float -> Shape
4 square n = Rect n n
5
6 area :: Shape -> Float
7 area (Circle r) = pi * r^2
8 area (Rect x y) = x * y
9
10 > :t Circle -- Circle :: Float -> Shape
11 > :t Rect    -- Rect  :: Float -> Float -> Shape

```

4.4. Sintaxis para records/registros.

Los **registros/records** son una extensión de Haskell que permite definir tipos de datos con campos etiquetados. Esto facilita la manipulación de datos estructurados al proporcionar nombres descriptivos para cada campo dentro de un registro. Sintaxis:

```
data NombreTripo = Constructor campo1 :: Tipo1, campo2 :: Tipo2, ...
```

En vez de tener que usar la declaración **data** de la manera antes mostrada, que sería de la siguiente forma:

```

1 -- Definicion de tipo de datos para guardar datos de alumnos.
2 data Alumno = A String String Int String deriving Show
3
4 -- Definimos un alumno.
5 juan = A "Juan" "Perez" 21 "jperez999@gmail.com"
6
7 -- Para acceder a los datos usamos funciones.
8 nombre :: Alumno -> String
9 nombre (A n _ _ _) = n
10
11 apellido :: Alumno -> String
12 apellido (A _ a _ _) = a
13
14 edad :: Alumno -> Int
15 edad (A _ _ e _) = e
16
17 email :: Alumno -> String
18 email (A _ _ _ m) = m

```

Utilizamos la sintaxis de records proveída por Haskell. De esta forma no tenemos que definir las proyecciones por separado, y no tenemos que acordarnos del orden de los campos:

```

1 -- Definicion de record.
2 data Alumno = A{nombre :: String, apellido :: String, edad :: Int, email :: String}
3   deriving Show
4 -- Definicion sin importar el orden.

```



```

5 juan = A{apellido = "Perez", nombre = "Juan", email = "jperez999@gmail.com", edad =
   21}
6
7 -- Acceso a los campos.
8 nombreDeJuan :: String
9 nombreDeJuan = nombre juan

```

4.5. Constructor de tipos Maybe.

Maybe es un tipo de datos algebraico en Haskell que se utiliza para representar valores que pueden estar o no presentes. Es útil para manejar casos en los que una función podría devolver un valor válido o un valor 'nulo' (ausencia de valor). Se define de la siguiente manera:

$$\text{data Maybe } a = \text{Nothing} \mid \text{Just } a$$

- *Maybe a* es un tipo de datos paramétrico que indica que puede contener un valor de tipo *a*, o puede estar vacío.
- *Nothing* es un constructor de datos que representa la ausencia de un valor. No lleva ningún argumento.
- *Just a* es un constructor de datos que representa la presencia de un valor. Lleva un argumento de tipo *a*, que es el valor que está presente.

Maybe es un constructor de tipos, ya que dado un tipo *a*, contruye el tipo *Maybe a*. Por lo tanto, no tiene un *tipo de datos* sino que tiene *kind*. El *kind* de *Maybe* es:

$$\text{Maybe} :: * \rightarrow *$$

Un uso particular de *Maybe* es para señalar una condición de error, o hacer total a una función parcial:

```

1 -- Haciendo total a la funcion head de listas.
2 safehead :: [a] -> Maybe a
3 safehead [] = Nothing
4 safehead xs = Just (head xs)
5
6 -- Ejemplo de funcion que recibe una clave y una lista 'clave-valor' y devuelve el
   valor asociado a la clave pasada como argumento si la encuentra en la lista.
7 lookup :: Eq c => c -> [(c, val)] -> Maybe val
8 lookup _ [] = Nothing
9 lookup k ((c,v):xs) | k == c = Just v
10                      | otherwise = lookup k xs

```

4.6. Constructor de tipos Either.

Either describe un tipo que puede tener elementos de dos tipos. En sus elementos está claro de qué tipo es el elemento almacenado. Su definición es:

$$\text{data Either } a \ b = \text{Left } a \mid \text{Right } b$$

- *Either a b*: es un tipo de datos paramétrico que indica que puede contener un valor de tipo *a* o un valor de tipo *b*.
- *Left a*: constructor de datos que pro lo general representa un resultado fallido. Lleva un argumento de tipo *a*, que generalmente se usa para almacenar información sobre el error.
- *Right b*: es un constructor de datos que representa un resultado exitoso. Lleva un argumento de tipo *b*, que es el valor exitoso que ha producido.

Nuevamente, como `Either` es un constructor de tipos, no tiene un *tipo de dato* sino un *kind*:

`Either :: * -> * -> *`

`Either` es útil para representar resultados que pueden ser uno de dos posibles valores diferentes. Por ejemplo, manejar casos en los que una función puede devolver un resultado exitoso o un error:

```
1 -- Funcion head de listas.
2 safehead :: [a] -> Either String a
3 safehead [] = Left "head de lista vacia!"
4 safehead xs = Right (head xs)
5
6 -- Ejemplo de funcion al dividir por cero.
7 divide :: Float -> Float -> Either String Float
8 divide _ 0 = Left "No se puede dividir por cero!"
9 divide x y = Right (x/y)
10
11 -- Ejemplos de elementos de tipo Either Bool Bool (son todos los posibles).
12 Left True
13 Left False
14 Right True
15 Right False
```

4.7. Tipos recursivos.

Los **tipos recursivos** son una característica de Haskell que permite definir tipos de datos que se refieren a sí mismos de forma recursiva. Esto permite crear estructuras de datos complejas y recursivas, como árboles, listas y otras estructuras anidadas.

Para declarar los tipos recursivos usamos las declaraciones **data** nuevamente. Los tipos recursivos además pueden tener parámetros.

```
1 -- Tipo de datos para numeros naturales.
2 data Nat = Zero | Succ Nat
3
4 add :: Nat -> Nat -> Nat
5 add n Zero = n
6 add n (Succ m) = Succ (add n m)
7
8 mult :: Nat -> Nat -> Nat
9 mult n Zero = Zero
10 mult n (Succ m) = add n (mult n m)
11
12 -- Listas definidas recursivamente. Son isomorfos a las listas predefinidas.
13 data List a = Nil | Cons a (List a)
14
15 to :: List a -> [a]
16 to Nil = []
17 to (Cons x xs) = x : (to xs)
18
19 from :: [a] -> List a
20 from [] = Nil
21 from (x:xs) = Cons x (from xs)
```

4.8. Expresiones case.

Case es una expresión que se utiliza para hacer coincidencia de patrones en el valor de una expresión y ejecutar diferentes acciones basadas en el patrón que coincida. Su sintaxis es:

```
1 case expression of
2   patron1 -> expresion1
3   patron2 -> expresion2
```

- *expresion*: es la expresión cuyo valor queremos analizar.
- *patron1*, *patron2*, ...: son patrones que se comparan con el valor de *expresion*.
- *expresion1*, *expresion2*, ...: son las expresiones que se evalúan si el valor coincide con el patrón correspondiente.

Los patrones de los diferentes casos son intentados en orden. Se usa indentación para marcar un bloque de casos.

```
1 esCero :: Nat -> Bool
2 esCero n = case n of
3     Zero -> True
4     _ -> False
```

4.9. Árboles.

4.9.1. Declaración de tipo de datos árbol.

Para representar diferentes tipos de **árboles**, podemos definir un tipo de datos recursivo utilizando la declaración *data*.

```
1 -- Arbol con informacion solamente en las hojas.
2 data T1 a = Tip a | Bin (T1 a) (T1 a)
3
4 -- Arbol con informacion solamente en los nodos.
5 data T2 b = Empty | Branch (T2 b) b (T2 b)
6
7 -- Arbol con informacion de tipo a en las hojas y de tipo b en los nodos.
8 data T3 a b = Leaf a | Node (T3 a b) b (T3 a b)
9 -- >t Node -> Node :: T3 a b -> b -> T3 a b -> T3 a b
10
11 -- Arbol con informacion solamente en los nodos, pero que puede tener nodos con dos
12 -- hijos o tres hijos.
13 data T4 a = E | N2 a (T4 a) (T4 a) | N3 a (T4 a) (T4 a) (T4 a)
14
15 -- Arbol con multiples hijos. Los hijos de un nodo se expresan en forma de lista.
16 data T5 a = Rose a [T5 a]
```

4.9.2. Programando con árboles.

Declaremos funciones que midan el tamaño y altura de un árbol.

```
1 data T1 a = Tip a | Bin (T1 a) (T1 a)
2
3 size :: T1 a -> Int
4 size (Tip _) = 1
5 size (Bin t1 t2) = size t1 + size t2
6
7 depth :: T1 a -> Int
8 depth (Tip _) = 0
9 depth (Bin t1 t2) = 1 + max (depth t1) (depth t2)
```

Podemos ver que medir un árbol de tamaño n es de complejidad $O(n)$. Podríamos hacerlo en tiempo constante $O(1)$ si lleváramos en la estructura un campo para la medida del árbol:

```
1 type Weight = Int
2 data T a = Tip Weight a | Bin Weight (T a) (T a)
3
4 -- Funcion que obtiene la medida es O(1).
5 weight :: T a -> Weight
6 weight (Tip w _) = w
```

```

7 weight (Bin w _ _) = w
8
9 -- Queremos preservar la siguiente invariante al construir un arbol nuevo.
10 -- weight (Bin w t1 t2) = weight t1 + weight t2
11 bin :: T a -> T a -> T a
12 bin t1 t2 = let w = weight t1 + weight t2
13             in Bin w t1 t2

```

4.9.3. Árboles de Huffman.

Un **árbol de Huffman** es una estructura que se usa para comprimir datos. La idea es representar los caracteres de un mensaje con códigos binarios (ceros o unos) más cortos para los más frecuentes, y más largos para los menos usados. Esto permite reducir el tamaño total del mensaje

La tabla de códigos se codifica en un árbol binario con información en las hojas. Se asignan los valores 0 o 1 (o (I)zquierda y (D)erecha) según las ramas de cada bifurcación: si vamos a la izquierda agregamos un 0, si vamos a la derecha agregamos un 1.

La secuencia *test* nos da el siguiente árbol, con los códigos correspondientes $t = 1, s = 01, e = 00$.

```

1      NULL
2  NULL  t
3 e      s

```

Representación de árboles de Huffman en Haskell.

```

1 type Weight = Int
2 data T a = Tip Weight a | Bin Weight (T a) (T a)

```

Decodificación de árboles de Huffman.

Leer una secuencia de símbolos (camino en un árbol) para recuperar el mensaje original.

```

1 -- Representar los pasos (Izquierda o Derecha). Un camino es una lista de pasos.
2 data Step = I | D deriving show
3 type Camino = [Step]
4
5 -- Funcion que sigue un camino completo hasta llegar a una hoja, y devuelve el
6   valor de esa hoja.
7 trace1 :: T a -> Camino -> a
8 trace1 (Tip _ x) [] = x
9 trace1 (Bin _ t1 t2) (I:xs) = trace1 t1 xs
10 trace1 (Bin _ t1 t2) (D:xs) = trace1 t2 xs
11
12 -- Esta funcion recorre parcialmente el arbol hasta encontrar una hoja, y devuelve:
13   (el valor de esa hoja a, el resto del camino que quedo sin recorrer).
14 -- Es util para decodificar una secuencia larga de bits (pasos), y queremos ir "
15   consumiendolo" de a partes el camino.
16 trace :: T a -> Camino -> (a, Camino)
17 trace (Tip _ x) resto = (x, resto)
18 trace (Bin _ t1 t2) (I:xs) = trace t1 xs
19 trace (Bin _ t1 t2) (D:xs) = trace t2 xs
20
21 -- Esta es la funcion principal de decodificacion. Toma un arbol de Huffman y una
22   lista de pasos, y va decodificando uno a uno los simbolos hasta que se termina
23   el camino.
24 decodexs :: T a -> Camino -> [a]
25 decodexs t ps = case trace t ps of
26   (a, []) -> [a]
27   (a, ps') -> a : decodexs t ps'

```

Codificación de árboles de Huffman.

Asignarle un camino único a cada símbolo según su frecuencia en la palabra.

```

1 -- Codifica una lista de valores en un unico camino.
2 codexs :: Eq a => T a -> [a] -> Camino
3 codexs t xs = concat (map (codex t) xs)
4
5 codex :: Eq t => T t -> t -> Camino
6 code t x = head (go to x)
7     where
8         go (Tip _ y) x | x == y = [[]]
9                       | otherwise = []
10        go (Bin _ t1 t2) x = map (I:) (go t1 x) ++ map (D:) (go t2 x)

```

5. Unidad 5 - Estructuras Inmutables

5.1. Introducción.

5.1.1. Estructuras de datos funcionales vs imperativas.

Muchos de los algoritmos tradicionales están pensados para estructuras **efímeras**. Las estructuras efímeras soportan una sola versión y son coherentes con un modelo secuencial. Los cambios en estas estructuras son destructivos.

Las estructuras **inmutables** soportan varias versiones y son más fácilmente paralelizables. Esta flexibilidad tiene un costo:

- Debemos adaptar las estructuras y algoritmos al modelo inmutable.
- Hay ciertas cotas de las estructuras efímeras que no siempre se van a poder alcanzar.

5.1.2. Inmutabilidad y sharing.

Las **estructuras inmutables** son estructuras de datos cuyos valores no pueden ser modificados una vez que han sido creados. En programación funcional, la inmutabilidad es una característica fundamental, donde los datos se tratan como valores constantes que no cambian con el tiempo. En lugar de modificar los datos existentes, se crean nuevos datos basados en los datos originales.

En lenguaje funcional puro, todas las estructuras son inmutables. Estas no se destruyen al hacer un cambio, más bien se copian los datos y se modifica la copia.

Por ejemplo, el caso de **listas enlazadas simples**. La concatenación de listas enlazadas simples efímeras:

$$zs = xs ++ ys$$

destruye las listas **xs** e **ys** pero la operación es $O(1)$.

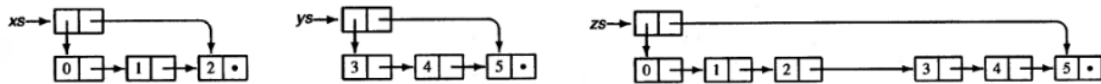


Figura 1: Listas xs e ys.

Figura 2: Concatenación de listas efímeras.

En cambio la concatenación de **listas enlazadas simples inmutables** no destruye las listas, sino que copia **xs** y hace que el último nodo apunte al primero de **ys**. Así, tendremos las tres listas **xs**, **ys**, **zs**, donde **zs** se obtuvo de copiar los nodos de **xs** pero comparte los nodos de **ys** (sharing). En este caso, la operación de copiar todos los nodos de **xs** en **zs** es del orden de $O(|xs|)$.

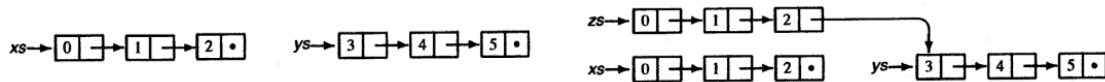


Figura 3: Listas xs e ys.

Figura 4: Concatenación de listas inmutables.

Ejemplo de sharing en función de listas.

La siguiente función que modifica un sólo elemento de una lista utiliza sharing: copia los nodos hasta la posición que queremos modificar, modificar el valor en el último nodo copiado, y éste apunta al siguiente en la lista original.

```

1 update :: [a] -> Int -> a -> [a]
2 update [] _ _ = []
3 update (x:xs) 0 x' = x':xs
4 update (x:xs) i x' = x:(update xs (i-1) x')

```

5.2. Árboles binarios en Haskell.

5.2.1. Árboles binarios.

Un **árbol binario** es un árbol en el que cada nodo tiene exactamente dos hijos. En Haskell representamos un árbol binario con la siguiente definición recursiva:

```

1 data Bin a = Hoja | Nodo (Bin a) a (Bin a)

```

Definimos funciones sobre los árboles mediante pattern-matching y recursión:

```

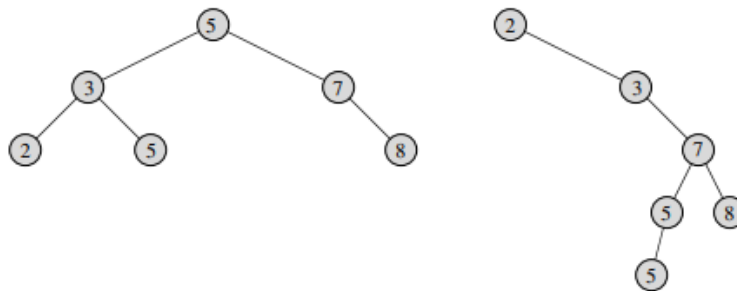
1 -- Funcion que determina si un elemento pertenece a un arbol binario.
2 member :: Eq a => a -> Bin a -> Bool
3 member a Hoja = False
4 member a (Nodo l b r) = (a == b) || member a l || member a r

```

5.2.2. Árboles binarios de búsqueda (BST).

Un **árbol binario de búsqueda** es un árbol binario t tal que:

- o bien t es una hoja,
- o bien t es un *Nodo* l a r , y se cumple que:
 - l y r son árboles binarios de búsqueda
 - si y es una clave en algún nodo de l entonces $y \leq a$
 - si y es una clave en algún nodo de r entonces $a < y$.



Operaciones sobre BSTs.

```

1 data Bin a = Hoja | Nodo (Bin a) a (Bin a)
2
3 -- Funcion que determina si un elemento pertenece a un BST.
4 member :: Ord a => a -> Bin a -> Bool
5 member a Hoja = False
6 member a (Nodo l b r) | a == b = True
7                       | a < b = member a l
8                       | a > b = member a r
9
10 -- Recorrido inorder de un BST.
11 inorder :: Bin a -> [a]
12 inorder Hoja = []
13 inorder (Nodo l b r) = inorder l ++ [b] ++ inorder r
14

```

```

15 -- Encontrar valor minimo de un BST.
16 minimum :: Bin a -> a
17 minimum (Nodo Hoja a r) = a
18 minimum (Nodo l a r) = minimum l
19
20 -- Encontrar valor maximo de un BST.
21 maximum :: Bin a -> a
22 maximum (Nodo l a Hoja) = a
23 maximum (Nodo l a r) = maximum r
24
25 -- Chequear si un arbol dado es un BST (si cumple las condiciones invariantes).
26 checkBST :: Ord a => Bin a -> Bool
27 checkBST Hoja = True
28 checkBST (Nodo l a r) =
29     let bstIzq = checkBST l
30         bstDer = checkBST r
31         invarianteIzq = esHoja l || (maxBST l <= a)
32         invarianteDer = esHoja r || (maxBST r > a)
33     in bstIzq && bstDer && invarianteIzq && invarianteDer
34     where
35         esHoja :: Bin a -> Bool
36         esHoja Hoja = True
37         esHoja _ = False
38
39 -- Insertar elemento en un BST (mantener invariantes). Recorremos el arbol hasta
40 -- encontrar una hoja, que transformamos en un nuevo nodo.
41 insert :: Ord a => a -> Bin a -> Bin a
42 insert a Hoja = Nodo Hoja a Hoja
43 insert a (Nodo l b r)
44     | a <= b = (Nodo (insert a l) b r)
45     | otherwise = (Nodo l b (insert a r))
46
47 -- Borrar elemento de un BST.
48 delete :: Ord a => a -> Bin a -> Bin a
49 delete _ Hoja = Hoja
50 delete x (Nodo l b r)
51     | x < b = Nodo (delete x l) b r
52     | x > b = Nodo l b (delete x r)
53     | otherwise =
54         case (l, r) of
55             (Hoja, Hoja) -> Hoja
56             (Hoja, _) -> r
57             (_, Hoja) -> l
58             (_, _) -> let y = minBST r
59                       in Nodo l y (delete y r)

```

Borrado de elementos en un BST.

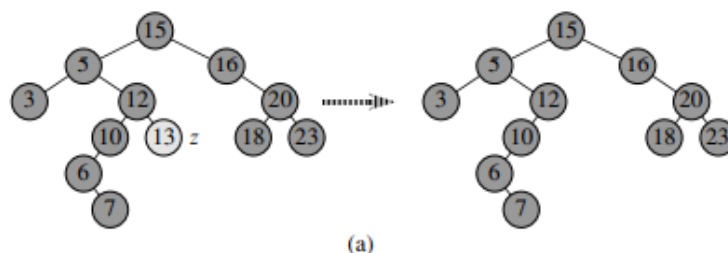
Una vez encontrado el elemento en el árbol binario de búsqueda, tenemos que hacer 3 consideraciones:

1. El nodo tiene hojas como subárboles.

```

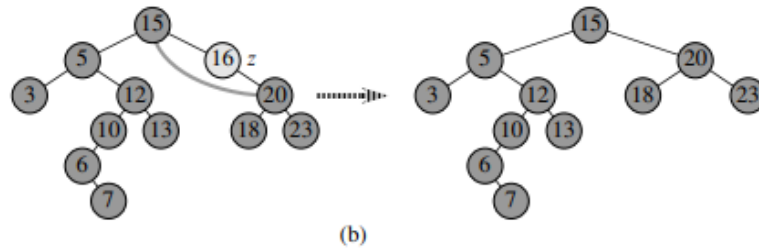
1 case (l, r) of
2     (Hoja, Hoja) -> Hoja

```



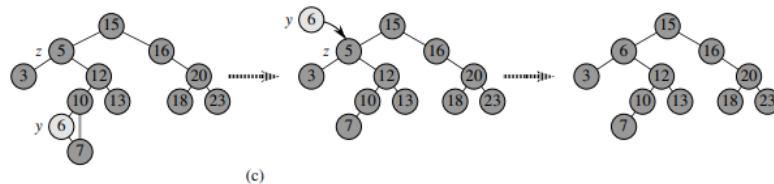
2. El nodo tiene un solo subárbol con datos.

```
1 case (l, r) of
2   (Hoja, _) -> r
3   (_, Hoja) -> l
```



3. El nodo tiene dos subárboles con datos.

```
1 case (l, r) of
2   (_, _) -> let y = minBST r
3               in Nodo l y (delete y r)
```



5.2.3. Red-Black Trees.

En los árboles binarios o árboles binarios de búsqueda las operaciones de búsqueda, inserción y borrado son del orden de la altura del árbol. En el mejor caso son $O(\lg n)$, pero en el peor caso pueden ser $O(n)$, por ejemplo al insertar datos ordenados, el árbol degenera en una lista.

La solución es mantener el árbol balanceado. Para eso existen algunas estructuras como AVLs y Red-Black Trees. En este cursado se estudiarán los Red-Black Trees.

Red-Black Trees.

Los **Red-Black Trees** son árboles binarios de búsqueda con nodos *coloreados* rojos o negros, que cumplen las siguientes invariantes para mantener al árbol balanceado:

1. **INV 1:** Ningún nodo rojo tiene hijos rojos.
2. **INV 2:** Todos los caminos de la raíz a una hoja tienen el mismo número de nodos negros (altura negra).

```
1 -- Definición del tipo de datos para Red-Black Trees (RBT).
2 data Color = R | B
3 data RBT a = E | T Color (RBT a) a (RBT a)
```

Observaciones:

- En un RBT, el camino más largo es a lo sumo el doble que el camino más corto.
- En un RBT la altura es $O(\lg n)$ (osea es un árbol balanceado).

Operaciones sobre RBTs.

```

1 -- Tipo de datos para Red-Black Trees.
2 data Color = R | B deriving Show
3 data RBT a = E | T Color (RBT a) a (RBT a) deriving Show
4
5 -- Funcion que determina si un elemento pertenece a un RBT.
6 member :: Ord a => a -> RBT a -> Bool
7 member a E = False
8 member a (T _ l b r)
9     | a == b = True
10    | a < b = member a l
11    | a > b = member a r
12
13 -- Insertar elemento en un RBT.
14 insert :: Ord a => a -> RBT a -> RBT a
15 insert x t = makeBlack (ins x t)
16     where
17         ins :: Ord a => a -> RBT a -> RBT a
18         ins x E = T R E x E
19         ins x (T c l y r)
20             | x < y = balance c (ins x l) y r
21             | x > y = balance c l y (ins x r)
22             | otherwise = T c l y r -- Si el valor ya existia, omitimos el caso.
23
24         makeBlack :: RBT a -> RBT a
25         makeBlack E = E
26         makeBlack (T _ l x r) = T B l x r
27
28 -- Rebalanceo de Red-Black Trees para arreglar posible violacion del invariante 1:
29   que un nodo rojo tenga un hijo rojo: B-R-R.
30 balance :: Color -> RBT a -> a -> RBT a -> RBT a
31 balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
32 balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
33 balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
34 balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
35 balance c l a r = T c l a r

```

Rebalanceo de Red-Black Trees luego de una inserción.

Luego de insertar un nuevo nodo rojo hay a lo sumo una única violación del invariante 1, que ocurre cuando el padre es rojo. Por lo tanto, la violación siempre ocurre en un camino $B - R - R$.

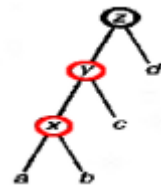
La función `balance` va arreglando y propagando hacia arriba esta violación. La (única) violación, puede aparecer en cuatro configuraciones. En todos casos la solución es la misma: reescribir el nodo como un padre rojo con dos hijos negros:

Caso 1: raíz z negra, subárbol izquierdo y rojo y subárbol izquierdo x de este último también rojo.

```

1 balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)

```

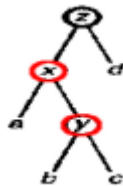


Caso 2: raíz z negra, subárbol izquierdo x rojo y subárbol derecho y de este último también rojo.

```

1 balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)

```



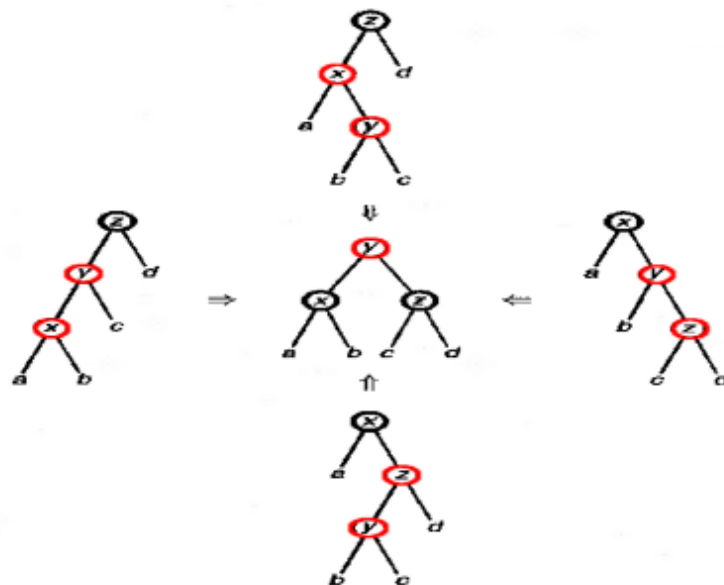
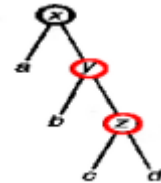
Caso 3: raíz x negra, subárbol derecho z rojo y subárbol izquierdo y de este último también rojo.

```
1 balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
```



Caso 4: raíz x negra, subárbol derecho y rojo y subárbol derecho z de este último también rojo.

```
1 balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
```



5.2.4. Heaps.

Los **heaps** (montículos) son árboles que permiten un acceso eficiente al mínimo elemento del árbol, al manter la invariante de que todo nodo es menor a todos los valores de sus nodos hijos. Por lo tanto, el mínimo siempre está en la raíz.

Un heap debe soportar eficientemente las siguientes operaciones:

```
1 insert :: Ord a => a -> Heap a -> Heap a
2 findMin :: Ord a => Heap a -> a
3 deleteMin :: Ord a => Heap a -> Heap a
```

Leftist heaps.

Un **Leftist Heap** (montículo izquierdista) es una estructura de datos para implementar colas de prioridad. Se parece a un árbol binario, pero con dos propiedades especiales:

- **Propiedad de heap:** el valor de un nodo es menor (o igual) que el de sus hijos (min-heap).
- **Invariante Leftist:** el **rango** de cualquier hijo izquierdo es mayor o igual que el de su hermano de la derecha.

Esto quiere decir que para cada nodo, el camino más corto hasta una hoja pasa por el hijo derecho. Es decir, el subárbol izquierdo siempre es *más pesado* o *más profundo* que el derecho.

Estas propiedades traen las siguientes consecuencias:

- La espina derecha es el camino más corto hasta una hoja/nodo vacío.
- La longitud de la espina derecha es como mucho $\lg(n + 1)$.
- Los elementos de la espina derecha están ordenados (como consecuencia de la propiedad del heap).

Implementación de Leftist Heaps.

```

1  -- El rango es el camino mas corto desde la raiz hasta una hoja por el subarbol
   derecho.
2  type Rank = Int
3  data Heap a = E | N Rank a (Heap a) (Heap a) deriving Show
4
5  -- Funcion que devuelve el rango de un heap.
6  rank :: Heap a -> Rank
7  rank E = 0
8  rank (N r _ _ ) = r
9
10 -- Funcion que mergea dos heaps O(lg n) (es la operacion mas importante de heaps).
11 -- Fusionamos siempre con el subarbol derecho porque en leftist heaps el derecho
   es mas liviano.
12 merge :: Ord a => Heap a -> Heap a -> Heap a
13 merge h1 E = h1
14 merge E h2 = h2
15 merge h1@(N _ x a1 b1) h2@(N _ y a2 b2) =
16     if x <= y then makeH x a1 (merge b1 h2)
17     else makeH y a2 (merge h1 b2)
18     where
19         -- Despues de fusionar, nos aseguramos de que el hijo izquierdo (a) tenga
           mayor rank que el derecho (b).
20         makeH x a b = if rank a >= rank b
21                       then N (rank b + 1) x a b
22                       else N (rank a + 1) x b a
23
24 -- Funcion para insertar un elemento en un heap O(lg n).
25 insert :: Ord a => a -> Heap a -> Heap a
26 insert x h = merge (N 1 x E E) h
27
28 -- Funcion que devuelve el minimo de un heap O(1).
29 findMin :: Ord a => Heap a -> a
30 findMin (N _ x a b) = x
31
32 -- Funcion que borra el elemento minimo de un heap O(lg n).
33 deleteMin :: Ord a => Heap a -> Heap a
34 deleteMin (N _ x a b) = merge a b

```