Estructuras de Datos y Algoritmos 2

Ignacio Rimini

March 2025

Índice

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 $f, g : \mathbb{N} \to \mathbb{R}$. 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:

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

1.1.2. Definición. Notación Big Omega.

Sean $f, g : \mathbb{N} \to \mathbb{R}$. 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:

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

1.1.3. Definición. Notación Big Theta.

Sean $f, g : \mathbb{N} \to \mathbb{R}$. 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 $f, g : \mathbb{N} \to \mathbb{R}$. 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:

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

1.2. Convenciones matemáticas.

1.2.1. Definición. Piso de un número.

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

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

1.2.2. Definición. Techo de un número.

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

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

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 $f : \mathbb{N} \to \mathbb{R}$ 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:

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

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

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

1.2.5. Funciones exponenciales.

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

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

1.2.6. Series matemáticas.

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

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.

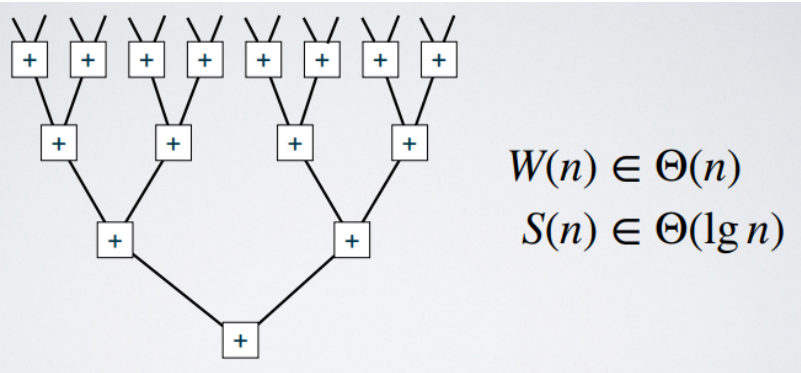
1.3.2. Definición. Profundidad (S).

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 $Max_{p \in Path(g)}|p|$ (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 determinar 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.

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

Profundidad (S) del lenguaje pseudocódigo.

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

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:

- $W(n) = Wdividir(n) + \sum_{i=1}^{k} W(n_i) + Wcombinar(n)$
- $S(n) = Sdividir(n) + \underset{i=1}{\overset{k}{\text{máx}}}\, S(n_i) + Scombinar(n)$

1.4.3. Algoritmo MergeSort.

El algoritmo **MergeSort** de ordenación de listas es un ejemplo de aplicación de la estrateia 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.

```
msort : [Int] -> [Int]
msort []  = []
msort [x] = [x]
msort xs  = let
                (ls, rs) = split xs
                (ls', rs') = (msort ls || msort rs)
            in
                merge(ls',rs')
```

```
split : [Int] -> [Int] x [Int]
split []       = ([], [])
split [x]      = ([x], [])
split (x:y:zs)= let
                    (xs, ys) = split zs
                in
                    (x:xs, y:ys)
```

```
merge : [Int] x [Int] -> [Int]
merge ([], ys)       = ys
merge (xs, [])       = xs
merge (x:xs, y:ys)   = if x <= y
                         then x:merge(xs, y:ys)
                         else y:merge(x:xs, ys)
```

Trabajo del algoritmo.

- $Wmsort(0) = c_0$
- $Wmsort(1) = c_1$
- $Wmsort(n) = Wsplit(n) + 2Wmsort(\frac{n}{2}) + Wmerge(n) + c_2$
- $Wsplit(0) = c_3$
- $Wsplit(1) = c_4$
- $Wsplit(n) = Wsplit(n-2) + c_5$
- $Wmerge(0) = c_6$
- $Wmerge(n) = Wmerge(n-1) + c_7$

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

$$\boxed{Wmsort(n) = 2Wmsort\left(\frac{n}{2}\right) + c_3 n \quad \in O(n\ lg\ n)}$$

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 $msort$:

$$\boxed{Smsort(n) = Smsort\left(\frac{n}{2}\right) + k_3 n \quad \in O(n)}$$

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:

- $W(0) = c_0$
- $W(1) = c_1$
- $W(n) = 2W(\lfloor \frac{n}{2} \rfloor) + c_2 n$

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).

$$\begin{aligned} W(n) = 2W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c_2 n &\leq^{HI} 2\left(c\left\lfloor \frac{n}{2} \right\rfloor lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) + c_2 n \\ &\leq \cancel{2} c \frac{n}{\cancel{2}} lg\left(\frac{n}{2}\right) + c_2 n \\ &= cn(lg_2(n) - lg_2(2)) + c_2 n \\ &= cn(lg(n) - 1) + c_2 n \\ &= cnlg(n) - cn + c_2 n \\ &\leq cnlg(n) \end{aligned}$$

  Y la última igualdad ocurre si $-cn + c_2 n \leq 0 \Rightarrow -c + c_2 \leq 0 \Rightarrow c_2 \leq c$. 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 est caso base.
- **n = 1:** Queremos ver si $W(1) \leq clg(1)$. 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 $W(2) \leq 2clg(2)$.

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

  Luego debemos escoger c tal que $W(2) = 2(c_1 + c_2) \leq 2clg(2) = 2c$. 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 $W(3) \leq 3clg(3)$.

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

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

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

$$c \geq \text{máx}\{c_2, c_1 + c_2, \frac{2c_1 + 3c_2}{3lg(3)}\}$$

concluimos que $\forall n \geq 2$, $W(n) \leq cnlg(n)$ 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.

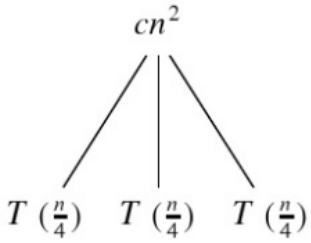
Cada rama de una llamada recursiva, a su vez, se expande en su propio árbol, y se procede así hasta llegar a un caso base, en el cual se deja de expandir el árbol. Por cada nivel del árbol se suma el total de operaciones, y estos a su vez se suman para dar el costo total de la recurrencia.

2.3.2. Ejemplo. Árbol de recurrencia.

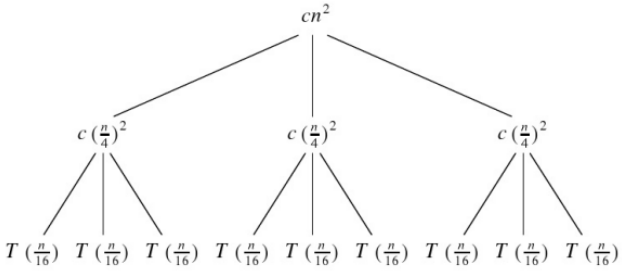
Consideremos la siguiente recurrencia:

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

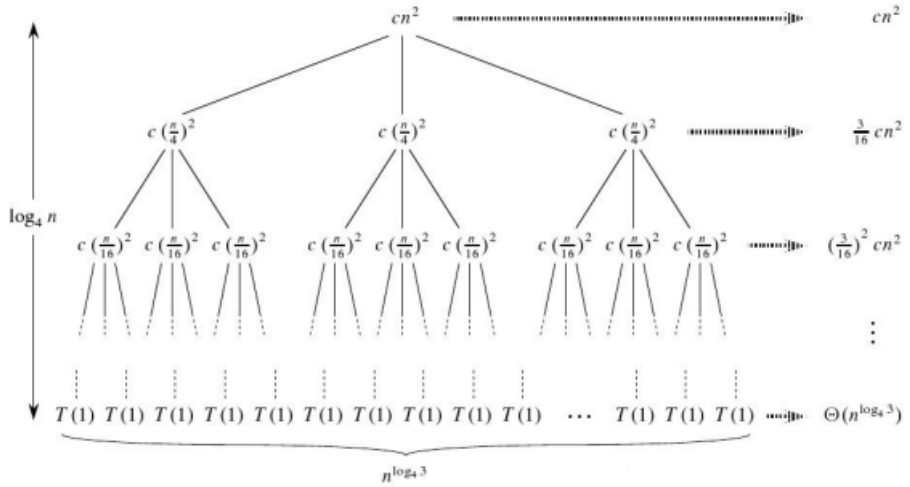
Hacemos un árbol que en su raíz tiene el costo para el n inicial, con ramas indicando cada una de las llamadas recursivas:



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

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

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

- f es eventualemente 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 trbajo del mergeSort es:

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

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:

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

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 queriamos probar. □

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

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

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:

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

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:

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

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 poderoso.
- Funciones recursivas.
- Fácilidad para probar propiedades de programas.
- Funciones de alto orden.
- Evaluación peresoza.
- Facilidad paara 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.

```haskell
-- Definimos el modulo principal.
module Main where

-- Definiciones.
numero = 5
lista = [1, 2, 3]
```

3. **Compilar y ejecutar un archivo haskell.**

```haskell
-- Compilar el archivo con ghci.
ghci archivo.hs

-- Ejecutar el modulo.
main

-- Mostrar variables.
numero
lista

-- Salir de ghci.
: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.

```
-- Comentario en linea
{-
    Bloque de comentarios.
    Util para comentarios en varias lineas.
-}
```

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.

```
a = b + c
    where
        b = 1
        c = 2
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:

$$10\ `div`\ 4 = div\ 10\ 4$$

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).

$$infixr\ 6\ (**)$$

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 *True :: Bool* y *False :: 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, (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.

```
listaBool :: [Bool]
listaBool = [True, True, False, True]

listaPalabra :: [Char]
listaPalabra = ['h', 'o', 'l', 'a']

listaEjemplo :: [[Char]]
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.

```
-- Concatenacion: se utiliza el operador (++).
lista1 ++ lista2

-- Agregar elemento al principio de la lista (cons). Se utiliza el operador (:).
1 : [2, 3, 4, 5]

-- Obtener longitud de lista: se utiliza la funcion (length) que devuelve la
    longitud de una lista.
length [1, 2, 3, 4, 5]

-- Acceder segun indice: se utiliza la funcion infija <lista> !! <indice>. Asi
    obtendremos el elemento de indice indicado, en la lista argumento.
-- El siguiente codigo devuelve el tercer elemento (indice 2)
[1, 2, 3, 4, 5] !! 2
```

3.3.3. Funciones de listas.

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

```haskell
-- HEAD: devuelve el primer elmento de una lista
lista = [1, 2, 3, 4, 5]
primerElemento = head lista
-- primerElemento = 1

-- TAIL: devuelve todos los elementos de una lista excepto el primero.
lista = [1, 2, 3, 4, 5]
restoLista = tail lista
-- restoLista = [2, 3, 4, 5]

-- LAST: devuelve el ultimo elemento de una lista.
lista = [1, 2, 3, 4, 5]
ultimoElemento = last lista
-- ultimoElemento = 5

-- INIT: devuelve todos los elementos de una lista excepto el ultimo.
lista = [1, 2, 3, 4, 5]
sinUltimoElemento = init lista
-- sinUltimoElemento = [1, 2, 3, 4]

-- NULL: comprueba si una lista esta vacia.
listaVacia = []
estaVacia = null listaVacia
-- estaVacia = True

-- REVERSE: invierte una lista.
lista = [1, 2, 3, 4, 5]
listaInvertida = reverse lista
-- listaInvertida = [5, 4, 3, 2, 1]

-- TAKE: toma los primeros n elementos de una lista.
lista = [1, 2, 3, 4, 5]
primerosTresElementos = take 3 lista
-- primerosTresElementos = [1, 2, 3]

-- DROP: elimina los primeros n elementos de una lista.
lista = [1, 2, 3, 4, 5]
sinPrimerosTresElementos = drop 3 lista
-- sinPrimerosTresElementos = [4, 5]

-- ELEM: comprueba si un elemento esta presente en una lista.
lista = [1, 2, 3, 4, 5]
estaElTres = elem 3 lista
-- estaElTres = True

-- FILTER: filtra los elementos de una lista segun un predicado.
lista = [1, 2, 3, 4, 5]
pares = filter even lista
-- pares = [2, 4]

-- MAP: aplica una funcion a cada elemento de una lista.
lista = [1, 2, 3, 4, 5]
cuadrados = map (\x -> x^2) lista
-- cuadrados = [1, 4, 9, 16, 25]

-- 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.
lista = [1, 2, 3, 4, 5]
suma = foldr (+) 0 lista
-- 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:

$$[x^2 \mid x \leftarrow [1 \,..\, 5]]$$

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

Ejemplos.

```
-- Genera una lista de los cuadrados del 1 al 5.
cuadrados = [x^2 | x <- [1..5]]
-- [1,4,9,16,25]

-- Una lista por comprension puede tener varios generadores, separados por coma.
[(x, y) | x <- [1,2,3], y <- [4,5]]
-- [(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]

-- Un generador puede depender de un generador anterior. A estos los llamamos
    GENERADORES DEPENDIENTES.
-- La siguiente es la lista de todos los pares (x,y) tal que x, y estan en [1 .. 3]
     e y >= x.
[(x,y) | x <- [1..3], y <- [x..3]]
-- [(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

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> a
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, $(t_1, t_2, ...t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i.

Ejemplos.

```haskell
-- miTupla :: (Bool, Bool)
miTupla = (True, True)

-- miTupla2 :: (Bool, Char, Char)
miTupla2 = (True, 'a', 'b')

-- miTupla3 :: (Char, (Bool, Char))
miTupla3 = ('a', (True, 'c'))

-- miTupla4 :: (([Char], Char), Char)
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.

```haskell
-- Definicion de nuestra tupla.
miTupla = ('a', 24, True)

-- Definimos las funciones.
primero (x, _, _) = x
segundo (_, y, _) = y
tercero (_, _, z) = z

-- Usando funciones predefinidas. Validas para tuplas con dos elementos nomas.
miTupla2 = ('a', True)
fst miTupla2
snd miTupla2
```

3.5. Strings en Haskell.

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

```haskell
"Hola" :: String
"Hola" = ['H', 'o', 'l', 'a']
```

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

```haskell
cantminusc :: String -> Int
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.

```haskell
esPositivo :: Int -> Bool
esPositivo x = if x > 0 then True else False

abs :: Int -> Int
abs n = if n >= 0 then n else -n

signum :: Int -> Int
signum n = if n < 0 then -1 else
               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 le expresión correspondiente. **Otherwise = True** según el preludio.

```haskell
esPositivo :: Int -> Bool
esPositivo x
    | x > 0     = True
    | 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.

```haskell
-- Funcion not booleano.
not :: Bool -> Bool
not False = True
not True = False

-- Funcion que determina si un entero es 0.
esCero :: Int -> Bool
esCero 0 = True
esCero _ = False

-- Funcion que devuelve la longitud de una lista de cualquier tipo.
longitud :: [a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs

-- Funcion AND condicional.
(and) :: Bool -> Bool -> Bool
True (and) True     = True
True (and) False    = False
False (and) True    = False
False (and) False   = False

-- Funcion AND condicional resumida.
```

```haskell
(and) :: Bool -> Bool -> Bool
True (and) True = True
_ (and) _       = False

-- Funcion que devuelve si un dia pertenece al fin de semana.
data DiaSemana = Lunes | Martes | Miercoles | Jueves | Viernes | Sabado | Domingo
esFinDeSemana :: DiaSemana -> Bool
esFinDeSemana Sabado = True
esFinDeSemana Domingo = True
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 $t_1 \to t_2$ 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:

$$f :: a \to b$$

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.

```haskell
-- Recibe un entero y devuelve su doble.
doble :: Int -> Int
doble x = x * 2

-- Funcion que mapea valores de un tipo en otro.
not :: Bool -> Bool
isDigit : Char -> Bool

-- Funciones con multiples argumentos usando tuplas y listas.
add :: (Int, Int) -> Int
add (x,y) = x + y

deceroA :: Int -> [Int]
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:

```haskell
resultado = doble 5
-- 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:

```haskell
suma :: Int -> Int -> Int
suma x y = x + y

suma :: Int -> (Int -> Int)
suma x = \y -> x + y
```

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

```haskell
sumaDos :: Int -> Int
sumaDos = suma 2

resultado = sumaDos 3
-- resultado = 5
```

3.7.6. Variables locales en funciones.

- **Claúsula where.**

  La claúsula **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.

```haskell
areaCirculo :: Double -> Double
areaCirculo radio = pi * radioCuadrado
    where
        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.

```haskell
areaCirculo :: Double -> Double
areaCirculo radio =
    let radioCuadrado = radio * radio
    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.

```haskell
-- Calcular factorial de un numero.
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- Calcular size de una lista.
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

3.7.8. Funciones Lambda. Funciones anónimas.

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

$$\lambda x \to x + x$$

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.

```haskell
-- Funcion que suma dos numeros.
suma :: Int -> Int -> Int
suma = \x y -> x + y

-- Multiplicacion de 3 numeros.
\x y z -> x * y * z

-- Evitamos darle nombre a una funcion que se usa una vez.
impares n = map f [0 .. n - 1]
    where f x = x * 2 + 1

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.

```haskell
-- Para cualquier lista de cualquier tipo a, la funcion length que calcula la
    cantidad de elementos de una lista es la misma.
length :: [a] -> Int

-- Funcion polimorfica que intercambia los elementos de una tupla.
intercambiar :: (a, b) -> (b, a)
intercambiar (x, y) = (y, x)

intercambiar (1, "Hola")
-- ("Hola", 1)

intercambiar ("Mundo", True)
-- (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:

$$(+) :: Num\ a \Rightarrow a \to a \to a$$

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).

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

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

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

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

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

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

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

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

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

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

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

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

```
class Fractional a where
    (/) :: a -> a -> a
    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:

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

Ejemplos usando función zip.

```
-- Funcion que toma una lista y devuelve una lista de pares de los elementos
    adyacentes de la lista.
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
-- pairs [1,2,3,4] -> [(1,2),(2,3),(3,4)]

-- Funcion que toma una lista y devuelve True si esta ordenada; False en caso
    contrario.
-- 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.
sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs xs]
-- sorted [5,1,2,3] -> False

-- Funcion que toma dos enteros y una lista, y devuelve los elementos de la lista
    desde el indice del primer argumento al otro argumento.
-- Genera una lista de pares entre la lista argumento y los naturales del 0 a n,
    que simulan ser indices, para compararlos con i.
rangeof :: Int -> Int -> [a] -> [a]
rangeof low hi xs = [x | (x,i) <- zip xs [0..], i >= low, i <= hi]
-- 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 determinar 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.

```haskell
type Pos = (Int, Int)

origen :: Pos
origen = (0,0)

izq :: Pos -> Pos
izq (x,y) = (x-1,y)
```

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

```haskell
-- Con parametros.
type Par a = (a,a)

copiar :: a -> Par a
copiar x = (x,x)

-- Anidados.
type Punto = (Int, Int)
type Trans = Punto -> Punto

-- No pueden ser recursivos.
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:

```haskell
data Respuesta = Si | No | Desconocida

respuestas :: [Respuesta]
respuestas = [Si, No, Desconocida]

invertir :: Respuesta -> Respuesta
invertir Si = No
invertir No = Si
invertir Desconocida = Desconocida
```

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

```haskell
data Shape = Circle Float | Rect Float Float

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y

> :t Circle -- Circle :: Float -> Shape
> :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:

```haskell
-- Definicion de tipo de datos para guardar datos de alumnos.
data Alumno = A String String Int String deriving Show

-- Definimos un alumno.
juan = A "Juan" "Perez" 21 "jperez999@gmail.com"

-- Para acceder a los datos usamos funciones.
nombre :: Alumno -> String
nombre (A n _ _ _) = n

apellido :: Alumno -> String
apellido (A _ a _ _) = a

edad :: Alumno -> Int
edad (A _ _ e _) = e

email :: Alumno -> String
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:

```haskell
-- Definicion de record.
data Alumno = A{nombre :: String, apellido :: String, edad :: Int, email :: String}
     deriving Show

-- Definicion sin importar el orden.
```

```haskell
juan = A{apellido = "Perez", nombre = "Juan", email = "jperez999@gmail.com", edad = 21}

-- Acceso a los campos.
nombreDeJuan :: String
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:

```haskell
-- Haciendo total a la funcion head de listas.
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)

-- 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.
lookup :: Eq c => c -> [(c, val)] -> Maybe val
lookup _ [] = Nothing
lookup k ((c,v):xs) | k == c = Just v
                    | otherwise = lookup k xs
```

4.6. Contructor 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*:

$$\text{Either} :: * \to * \to *$$

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:

```haskell
-- Funcion head de listas.
safehead :: [a] -> Either String a
safehead [] = Left "head de lista vacia!"
safehead xs = Right (head xs)

-- Ejemplo de funcion al dividir por cero.
divide :: Float -> Float -> Either String Float
divide _ 0 = Left "No se puede dividir por cero!"
divide x y = Right (x/y)

-- Ejemplos de elementos de tipo Either Bool Bool (son todos los posibles).
Left True
Left False
Right True
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.

```haskell
-- Tipo de datos para numeros naturales.
data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add n Zero = n
add n (Succ m) = Succ (add n m)

mult :: Nat -> Nat -> Nat
mult n Zero = Zero
mult n (Succ m) = add n (mult n m)

-- Listas definidas recursivamente. Son isomorfos a las listas predefinidas.
data List a = Nil | Cons a (List a)

to :: List a -> [a]
to Nil = []
to (Cons x xs) = x : (to xs)

from :: [a] -> List a
from [] = Nil
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:

```haskell
case expresion of
    patron1 -> expresion1
    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 identación para marcar un bloque de casos.

```
esCero :: Nat -> Bool
esCero n = case n of
    Zero -> True
    _ -> 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.*

```
-- Arbol con informacion solamente en las hojas.
data T1 a = Tip a | Bin (T1 a) (T1 a)

-- Arbol con informacion solamente en los nodos.
data T2 b = Empty | Branch (T2 b) b (T2 b)

-- Arbol con informacion de tipo a en las hojas y de tipo b en los nodos.
data T3 a b = Leaf a | Node (T3 a b) b (T3 a b)
-- >t Node -> Node :: T3 a b -> b -> T3 a b -> T3 a b

-- Arbol con informacion solamente en los nodos, pero que puede tener nodos con dos
    hijos o tres hijos.
data T4 a = E | N2 a (T4 a) (T4 a) | N3 a (T4 a) (T4 a) (T4 a)

-- Arbol con multiples hijos. Los hijos de un nodo se expresan en forma de lista.
data T5 a = Rose a [T5 a]
```

4.9.2. Programando con árboles.

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

```
data T1 a = Tip a | Bin (T1 a) (T1 a)

size :: T1 a -> Int
size (Tip _) = 1
size (Bin t1 t2) = size t1 + size t2

depth :: T1 a -> Int
depth (Tip _) = 0
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 llevaramos en la estructura un campo para la medida del árbol:

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

-- Funcion que obtiene la medida es O(1).
weight :: T a -> Weight
weight (Tip w _) = w
```

```haskell
weight (Bin w _ _) = w

-- Queremos preservar la siguiente invariante al construir un arbol nuevo.
-- weight (Bin w t1 t2) = weight t1 + weight t2
bin :: T a -> T a -> T a
bin t1 t2 = let w = weight t1 + weight t2
            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 $t = 1, s = 01, e = 00$.

```
         NULL
    NULL     t
e        s
```

Representación de árboles de Huffman en Haskell.

```haskell
type Weight = Int
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.

```haskell
-- Representar los pasos (Izquierda o Derecha). Un camino es una lista de pasos.
data Step = I | D deriving show
type Camino = [Step]

-- Funcion que sigue un camino completo hasta llegar a una hoja, y devuelve el valor de esa hoja.
trace1 :: T a -> Camino -> a
trace1 (Tip _ x) [] = x
trace1 (Bin _ t1 t2) (I:xs) = trace1 t1 xs
trace1 (Bin _ t1 t2) (D:xs) = trace1 t2 xs

-- Esta funcion recorre parcialmente el arbol hasta encontrar una hoja, y devuelve: (el valor de esa hoja a, el resto del camino que quedo sin recorrer).
-- Es util para decodificar una secuencia larga de bits (pasos), y queremos ir "consumiendo" de a partes el camino.
trace :: T a -> Camino -> (a, Camino)
trace (Tip _ x) resto = (x, resto)
trace (Bin _ t1 t2) (I:xs) = trace t1 xs
trace (Bin _ t1 t2) (D:xs) = trace t2 xs

-- Esta es la funcion principal de decodificacion. Toma un arbol de Huffman y una lista de pasos, y va decodificando uno a uno los simbolos hasta que se termina el camino.
decodexs :: T a -> Camino -> [a]
decodexs t ps = case trace t ps of
                    (a, []) -> [a]
                    (a, ps')-> a : decodexs t ps'
```

Codificación de árboles de Huffman.

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

```haskell
-- Codifica una lista de valores en un unico camino.
codexs :: Eq a => T a -> [a] -> Camino
codexs t xs = concat (map (codex t) xs)

codex :: Eq t => T t -> t -> Camino
code t x = head (go t x)
    where
        go (Tip _ y) x  | x == y = [[]]
                        | otherwise = []
        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:

$$\mathbf{zs} = \mathbf{xs} +\!+ \mathbf{ys}$$

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

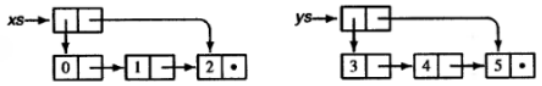


Figura 1: Listas xs e ys.

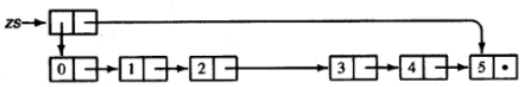


Figura 2: Concatenación de listas efímeras.

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.

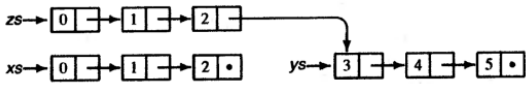


Figura 4: Concatenación de listas inmutables.

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.

```haskell
update :: [a] -> Int -> a -> [a]
update [] _ _ = []
update (x:xs) 0 x' = x':xs
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:

```haskell
data Bin a = Hoja | Nodo (Bin a) a (Bin a)
```

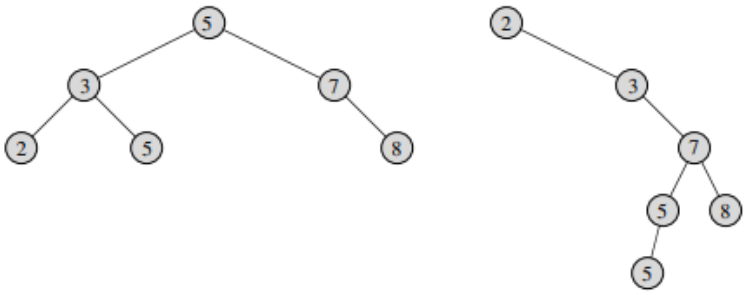
Definimos funciones sobre los árboles mediante pattern-matching y recursión:

```haskell
-- Funcion que determina si un elemento pertenece a un arbol binario.
member :: Eq a => a -> Bin a -> Bool
member a Hoja = False
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.

```haskell
data Bin a = Hoja | Nodo (Bin a) a (Bin a)

-- Funcion que determina si un elemento pertenece a un BST.
member :: Ord a => a -> Bin a -> Bool
member a Hoja = False
member a (Nodo l b r) | a == b = True
                      | a < b = member a l
                      | a > b = member a r

-- Recorrido inorder de un BST.
inorder :: Bin a -> [a]
inorder Hoja = []
inorder (Nodo l b r) = inorder l ++ [b] ++ inorder r

```

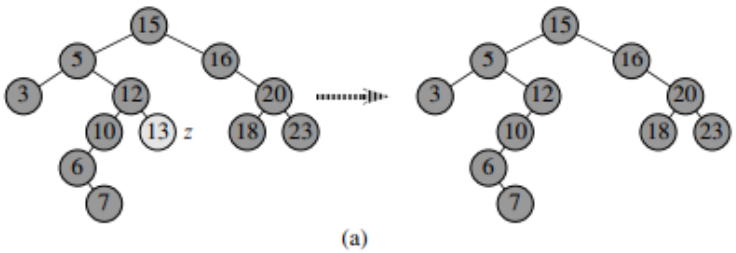
```haskell
-- Encontrar valor minimo de un BST.
minimum :: Bin a -> a
minimum (Nodo Hoja a r) = a
minimum (Nodo l a r) = minimum l

-- Encontrar valor maximo de un BST.
maximum :: Bin a -> a
maximum (Nodo l a Hoja) = a
maximum (Nodo l a r) = maximum r

-- Chequear si un arbol dado es un BST (si cumple las condiciones invariantes).
checkBST :: Ord a => Bin a -> Bool
checkBST Hoja = True
checkBST (Nodo l a r) =
    let bstIzq = checkBST l
        bstDer = checkBST r
        invarianteIzq = esHoja l || (maxBST l <= a)
        invarianteDer = esHoja r || (minBST r > a)
    in bstIzq && bstDer && invarianteIzq && invarianteDer
    where
        esHoja :: Bin a -> Bool
        esHoja Hoja = True
        esHoja _ = False

-- Insertar elemento en un BST (mantener invariantes). Recorremos el arbol hasta
    encontrar una hoja, que transformamos en un nuevo nodo.
insert :: Ord a => a -> Bin a -> Bin a
insert a Hoja = Nodo Hoja a Hoja
insert a (Nodo l b r)
    | a <= b = (Nodo (insert a l) b r)
    | otherwise = (Nodo l b (insert a r))

-- Borrar elemento de un BST.
delete :: Ord a => a -> Bin a -> Bin a
delete _ Hoja = Hoja
delete x (Nodo l b r)
    | x < b = Nodo (delete x l) b r
    | x > b = Nodo l b (delete x r)
    | otherwise =
        case (l, r) of
            (Hoja, Hoja) -> Hoja
            (Hoja, _) -> r
            (_, Hoja) -> l
            (_, _) -> let y = minBST r
                      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.

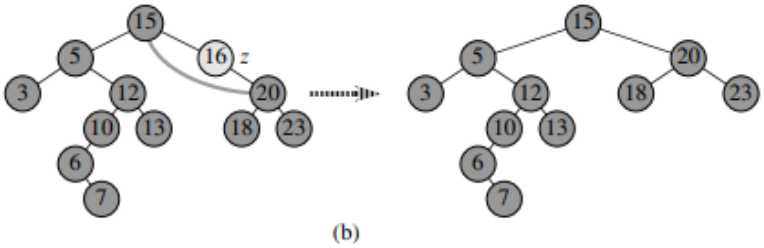
```haskell
case (l, r) of
    (Hoja, Hoja) -> Hoja
```



(a)

2. El nodo tiene un solo subárbol con datos.

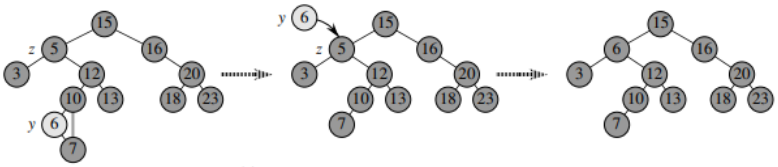
```haskell
case (l, r) of
    (Hoja, _) -> r
    (_, Hoja) -> l
```



(b)

3. El nodo tiene dos subárboles con datos.

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



(c)

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).

```haskell
-- Definicion del tipo de datos para Red-Black Trees (RBT).
data Color = R | B
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.

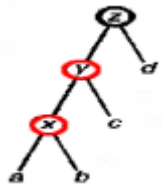
```haskell
-- Tipo de datos para Red-Black Trees.
data Color = R | B deriving Show
data RBT a = E | T Color (RBT a) a (RBT a) deriving Show

-- Funcion que determina si un elemento pertenece a un RBT.
member :: Ord a => a -> RBT a -> Bool
member a E = False
member a (T _ l b r)
    | a == b = True
    | a < b = member a l
    | a > b = member a r

-- Insertar elemento en un RBT.
insert :: Ord a => a -> RBT a -> RBT a
insert x t = makeBlack (ins x t)
    where
        ins :: Ord a => a -> RBT a -> RBT a
        ins x E = T R E x E
        ins x (T c l y r)
            | x < y = balance c (ins x l) y r
            | x > y = balance c l y (ins x r)
            | otherwise = T c l y r -- Si el valor ya existia, omitimos el caso.

        makeBlack :: RBT a -> RBT a
        makeBlack E = E
        makeBlack (T _ l x r) = T B l x r

-- Rebalanceo de Red-Black Trees para arreglar posible violacion del invariante 1:
    que un nodo rojo tenga un hijo rojo: B-R-R.
balance :: Color -> RBT a -> a -> RBT a -> RBT a
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)
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)
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)
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)
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 $B - R - R$.

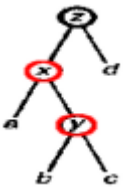
La función balance va arreglando y propagando hacia arriba esta violación. La (única) violación, puede aparecer en cuatro configuraciones. En todos casos la solución es la misma: reescribir el nodo como un padre rojo con dos hijos negros:

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



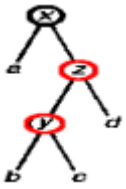

```haskell
balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
```

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



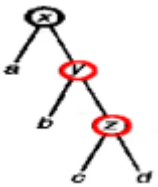

```haskell
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
```

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

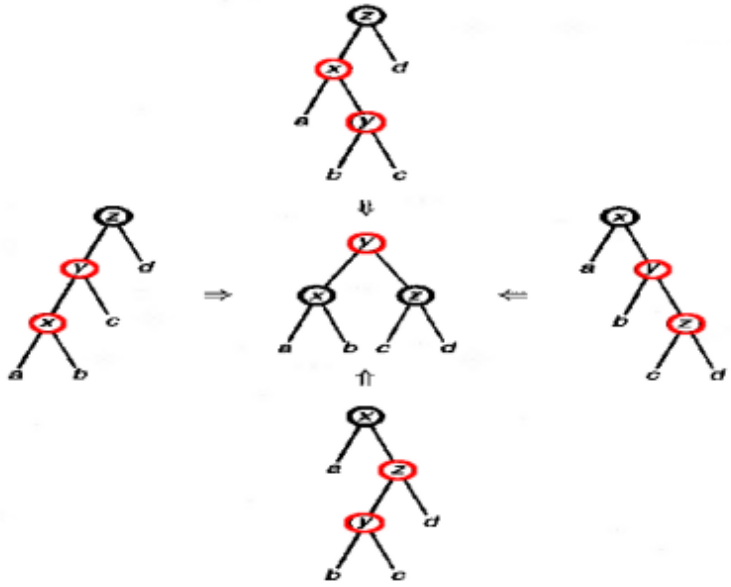


```
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
```

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



```
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:

```
insert :: Ord a => a -> Heap a -> Heap a
findMin :: Ord a => Heap a -> a
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.

```haskell
-- El rango es el camino mas corto desde la raiz hasta una hoja por el subarbol derecho.
type Rank = Int
data Heap a = E | N Rank a (Heap a) (Heap a) deriving Show

-- Funcion que devuelve el rango de un heap.
rank :: Heap a -> Rank
rank E = 0
rank (N r _ _ _) = r

-- Funcion que mergea dos heaps O(lg n) (es la operacion mas importante de heaps).
--  Fusionamos siempre con el subarbol derecho porque en leftist heaps el derecho es mas liviano.
merge :: Ord a => Heap a -> Heap a -> Heap a
merge h1 E = h1
merge E h2 = h2
merge h1@(N _ x a1 b1) h2@(N _ y a2 b2) =
    if x <= y   then makeH x a1 (merge b1 h2)
                else makeH y a2 (merge h1 b2)
    where
        -- Despues de fusionar, nos aseguramos de que el hijo izquierdo (a) tenga mayor rank que el derecho (b).
        makeH x a b = if rank a >= rank b
            then N (rank b + 1) x a b
            else N (rank a + 1) x b a

-- Funcion para insertar un elemento en un heap O(lg n).
insert :: Ord a => a -> Heap a -> Heap a
insert x h = merge (N 1 x E E) h

-- Funcion que devuelve el minimo de un heap O(1).
findMin :: Ord a => Heap a -> a
findMin (N _ x a b) = x

-- Funcion que borra el elemento minimo de un heap O(lg n).
deleteMin :: Ord a => Heap a -> Heap a
deleteMin (N _ x a b) = merge a b
```

6. Unidad 6 - Tipos abstractos de datos (TADs).

6.1. Tipos abstractos de datos.

6.1.1. Definición. TAD.

Los **tipos abstractos de datos (TADs)** son una forma de estructurar y organizar datos en programación. Definen un conjunto de valores y un conjunto de operaciones que pueden realizarse con esos valores, pero ocultan los detalles internos de cómo se implementan esos valores y operaciones.

Esto permite que los usuarios utilicen los tipos de datos sin necesidad de conocer los detalles de su implementación subyacente, lo que promueve la modularidad y la abstracción en el diseño de programas.

- La idea de un tipo abstracto de datos es abstraer detalles de implementación.
- El usuario es alguien que simplemente usa la abstracción.
- El implementador provee una implementación que se ajusta al comportamiento esperado.
- El usuario solo puede suponer el comportamiento descripto.
- Podemos ser más precisos sobre el comportamiento mediante una **especificación.**

Un **tipo abstracto de datos (TAD)** consiste de:

1. **Un nombre de tipo.**
2. **Operaciones.**
3. **Especificación del comportamiento.**
   - **Especificación algebraica:** se describen operaciones y ecuaciones entre operaciones.
   - **Modelos:** se describen operaciones y cómo se interpretan en un modelo matemático.

6.1.2. Ejemplo. TAD de Colas.

Una **cola** es una estructura a la cual:

- Podemos agregar elementos.
- Podemos obtener el primer elemento.
- Podemos quitar el primer elemento.
- Podemos preguntar si está vacía.
- Existe una relación entre el orden en que se agregan elementos y se sacan (FIFO).

Esta descripción es abstracta porque refleja el comportamiento y no la implementación.

Tipo abstracto de datos para colas.

- **Nombre de tipo:** Cola.
- **Operaciones:**

```
1 tad Cola (A:Set) where
2     import Bool
3     vacia : Cola A
4     poner : A -> Cola A -> Cola A
5     primero : Cola A -> A
6     sacar : Cola A -> Cola A
7     esVacia : Cola A -> Bool
```

- **Especificación algebraica:**

$$\begin{aligned}
\text{esVacia vacia} &= \text{True} \\
\text{esVacia (poner x q)} &= \text{False} \\
\text{primero (poner x vacia)} &= \text{x} \\
\text{primero (poner x (poner y q))} &= \text{primero (poner y q)} \\
\text{sacar (poner x vacia)} &= \text{vacia} \\
\text{sacar (poner x (poner y q))} &= \text{poner x (sacar (poner y q))}
\end{aligned}$$

6.1.3. Especificaciones algebraicas de un TAD.

Las **especificaciones algebraicas** deben ser mediante ecuaciones. Solo deben aparecer operaciones del TAD y variables libres (se suponen siempre cuantificadas universalmente).

Por ejemplo, la manera correcta de escribir:

$$size\ x = \begin{cases} 0 & si\ esVacia\ x \\ 1 + size(sacar\ x) & en\ otro\ caso \end{cases}$$

Sería de la siguiente forma:

$$\text{size x} = \textbf{if}\ (\text{esVacia x})\ \textbf{then}\ 0\ \textbf{else}\ 1 + \text{size (sacar x)}$$

donde **0, 1** son operadores nularios, y **+** es un operador binario de un TAD de Naturales. Además, **if then else** es un operador ternario de un TAD de Booleanos.

En algunos casos puede quedar comportamiento sin definir. Por ejemplo, qué dice la especificación de Colas sobre *primero vacia*?

6.1.4. Modelos en un TAD.

Como modelo de colas tomamos las **secuencias** $\langle x_1, x_2, ...x_n \rangle$ y para cada operación damos una función equivalnte sobre este modelo:

$$\begin{aligned}
\text{vacia} &= \langle\rangle \\
\text{poner x } \langle x_1, x_2, ...x_n \rangle &= \langle x, x_1, x_2, ...x_n \rangle \\
\text{sacar } \langle x_1, x_2, ...x_n \rangle &= \langle x_1, x_2, ...x_{n-1} \rangle \\
\text{primero } \langle x_1, x_2, ...x_n \rangle &= x_n \\
\text{esVacia } \langle x_1, x_2, ...x_n \rangle &= \text{True si n} = 0 \\
\text{esVacia } \langle x_1, x_2, ...x_n \rangle &= \text{False en otro caso}
\end{aligned}$$

Luego, a cada cola c, le corresponde un modelo $[\![c]\!]$ y a cada operación op le corresponde una operación $[\![op]\!]$ que trabaja sobre el modelo. Una **implementación** se ajusta al modelo si:

$$[\![op\ c]\!] = [\![op]\!][\![c]\!]$$

6.1.5. Implementaciones de un TAD.

Cada TAD admite diferentes implementaciones. Por ejemplo, para colas, existen dos implementaciones distintas con listas:

- Primer elemento de la cola al principio de la lista: se agregan elementos al final y se sacan del principio.

- Primer elemento de la cola al final de la lista: se agregan elementos al principio y se sacan del final.

Primera implementación de colas con listas: primer elemento de la cola al principio de la lista, se agregan elementos al final y se sacan del principio.

```haskell
type Cola a = [a]

-- Wvacia = O(1)
vacia :: Cola a
vacia = []

-- Wponer = O(n)
poner :: a -> Cola a -> Cola a
poner v [] = [v]
poner v (x:xs) = x : (poner v xs)

-- Wprimero = O(1)
primero :: Cola a -> a
primero [] = error "Cola vacia"
primero (x:xs) = x

-- Wsacar = O(1)
sacar :: Cola a -> Cola a
sacar [] = error "Cola vacia"
sacar (x:xs) = xs

-- WesVacia = O(1)
esVacia :: Cola a -> Bool
esVacia xs = null xs
```

Segunda implementación de colas con listas: primer elemento de la cola al final de la lista, se agregan elementos al principio y se sacan del final.

```haskell
type Cola a = [a]

-- Wvacia = O(1)
vacia :: Cola a
vacia = []

-- Wponer = O(1)
poner :: a -> Cola a -> Cola a
poner x xs = x:xs

-- Wprimero = O(1)
primero :: Cola a -> a
primero [] = error "Cola vacia"
primero (x:xs) = last (x:xs)

-- Wsacar = O(n)
sacar :: Cola a -> Cola a
sacar [] = error "Cola vacia"
sacar [x] = vacia
sacar (x:xs) = x : (sacar xs)

-- WesVacia = O(1)
esVacia :: Cola a -> Bool
esVacia xs = null xs
```

Tercera implementación de colas (eficiente).

Las dos implementaciones anteriores son lineales en alguna operación, por eso, daremos otra implementación más eficiente de colas. Implementaremos colas usando un par de listas (xs, ys) tal que los elementos en orden sean:

$$xs \ ++ \ reverse \ ys$$

Y la implementación seguirá la siguiente invariante: **si** xs **es vacía, entonces** ys **también** (las operaciones deben conservar este invariante).

Por ejemplo la cola $\langle 1, 2, 3, 4, 5\rangle$ podría ser $([1, 2], [5, 4, 3])$ o $([1], [5, 4, 3, 2])$.

```haskell
type Cola a = ([a], [a])

-- Wvacia = O(1)
vacia :: Cola a
vacia = ([], [])

-- Wponer = O(1)
poner :: a -> Cola a -> Cola a
poner x (ys, zs) = validar (ys, x:zs)

-- Wprimero = O(1)
primero :: Cola a -> a
primero (x:xs, ys) = x

-- Wsacar = O(|ys|), O(1) (amortizado)
sacar :: Cola a -> Cola a
sacar (x:xs, ys) = validar (xs,ys)

-- WesVacia = O(1)
esVacia :: Cola a -> Bool
esVacia (xs, ys) = null xs

validar :: Cola a -> Cola a
validar (xs,ys) = if null xs
                    then (reverse ys, [])
                    else (xs, ys)
```

Observación. El costo amortizado de una operación es una medida promedio del costo de realizar una secuencia de operaciones en una estructura de datos durante un período de tiempo.

En la función **validar,** la llamada a **reverse** es $O(n)$ pero se hace cada tanto, solo en algunos pocos casos. Como en todos los otros casos de llamada a **validar** son $O(1)$, nos estamos ahorrando costo para cuando tengamos que aplicar el reverse. Es decir, la operación **sacar** termina siendo $O(1)$, pues hemos acumulado tantos $O(1)$ (como monedas) al momento de costear una operación $O(n)$ que es **reverse.**

6.1.6. TADs en Haskell.

Una forma de implementar un **TAD en Haskell** es mediante una **clase de tipos:**

```haskell
class Cola t where
    vacia :: t a
    poner :: a -> t a -> t a
    sacar :: t a -> t a
    primero :: t a -> a
    esVacia :: t a -> Bool
```

Y la implementación de un TAD es una **instancia:**

```haskell
instance Cola [] where
    vacia = []
    poner x xs = x:xs
    sacar xs = init xs
    primero = last
    esVacia xs = null xs
```

Usamos un TAD con una función polimórfica en el TAD. Notar que la siguiente función *ciclar* funciona para cualquier instancia de Cola. O lo que es lo mismo, funciona para cualquier implementación del TAD:

```
ciclar :: Cola t -> Int -> t a -> t a
ciclar 0 cola = cola
ciclar n cola = ciclar (n-1) (poner (primero cola) (sacar cola))
```

6.1.7. Conclusión.

- **Especificación:** qué operaciones tiene el TAD y cómo se comportan. Es única.
- **Implementación:** cómo se realizan las operaciones y cuánto cuestan. Puede haber varias implementaciones (con diferentes costos). Todas deben garantizar el comportamiento dado por la especificación.
- **Uso:** sólo puede suponer el comportamiento dado por la especificación. Se elije implementación de acuerdo al uso (menor costo para un determinado uso).
- Los TADs ocultan detalles de implementación, el comportamiento se describe algebraicamente o proveyendo un modelo y cada implementación debe tener una especificación de costo.

6.1.8. TADs básicos.

TAD de Booleanos.

```
tad Bool where
    true : Bool
    false : Bool
    not : Bool -> Bool
    and : Bool -> Bool -> Bool
    or : Bool -> Bool -> Bool
    => : Bool -> Bool -> Bool
    if _ then _ else _ : Bool -> E -> E -> E
```

$$\begin{aligned}
\text{not true} &= \text{false} \\
\text{not false} &= \text{true} \\
\text{and true } x &= x \\
\text{or true } x &= \text{true} \\
\text{or false } x &= x \\
\text{true} \Rightarrow x &= x \\
\text{false} \Rightarrow x &= \text{true} \\
\text{if true then } x \text{ else } y &= x \\
\text{if false then } x \text{ else } y &= y
\end{aligned}$$

TAD de Naturales.

```
tad Nat where
    import Bool
    0 : Nat
    succ : Nat -> Nat
    (>) : Nat -> Nat -> Bool
    (=) : Nat -> Nat -> Bool
    (+) : Nat -> Nat -> Nat
    (-) : Nat -> Nat -> Nat
```

$$\begin{aligned}
x + 0 &= x \\
x + (\text{succ } y) &= \text{succ}(x + y) \\
0 - x &= 0 \\
x - 0 &= x \\
(\text{succ } x) - (\text{succ } y) &= x - y \\
0 \equiv 0 &= true \\
(\text{succ } x) \equiv 0 &= false \\
0 \equiv (\text{succ } y) &= false \\
0 > x &= false \\
(\text{succ } x) > 0 &= true \\
(\text{succ } x) > (\text{succ } y) &= x > y
\end{aligned}$$

TAD de Maybe.

```
1 tad Maybe (A:Set) where
2     import Bool
3     Nothing : Maybe A
4     just : A -> Maybe A
5     isNothing : Maybe A -> Bool
6     fromJust : Maybe A -> A
```

$$\text{isNothing (Nothing)} = true$$
$$\text{isNothing (Just x)} = false$$
$$\text{fromJust (Just x)} = x$$

6.2. Verificación de especificaciones.

Dada una implementación TAD, ¿cómo sabemos que es correcta? → si implementa las operaciones y si estas operaciones verifican la especificación.

Dada una implementación de un TAD en Haskell, el sistema de tipos asegura que los tipos de las operaciones sean correctos, pero la verificación de la especificación la debe hacer el programador.

6.2.1. Razonamiento ecuacional.

Haskell permite razonar ecuacionalmente acerca de las definiciones en forma similar al álgebra. Por ejemplo, si tenemos la siguiente función:

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Podemos probar la propiedad: reverse [x] = [x]:

$$
\begin{aligned}
&= \text{reverse } [x] && \\
&= \text{reverse } (x : [\,]) && \langle \text{def. listas} \rangle \\
&= \text{reverse } [\,] \text{ ++ } [x] && \langle \text{reverse 2da def.} \rangle \\
&= [\,] \text{ ++ } [x] && \langle \text{reverse 1era def} \rangle \\
&= [x] && \langle \text{++ 1era def} \rangle
\end{aligned}
$$

6.2.2. Patrones disjuntos.

Al verificar la correctitud de un programa, es fundamental garantizar que su especificación sea clara y libre de ambigüedades. Una técnica clave para lograr esto es el uso de **patrones disjuntos** en definiciones por casos, ya que simplifican el razonamiento ecuacional y evitan solapamientos no deseados.

Ejemplo.

Consideremos la siguiente función **esCero,** que verifica si un entero es cero:

```haskell
esCero :: Int -> Bool
esCero 0 = True
esCero n = False
```

Aunque esta definición es correcta, introduce una dependencia implícita del orden de las ecuaciones: el segundo patrón (n) cubre todos los enteros, incluyendo el 0, pero solo se evalúa si el primer patrón no coincide.

Esto obliga al verificador a recordar que el orden importa, lo que puede complicar el razonamiento formal.

Versión con patrones disjuntos explícitos.

Una alternativa más robusta es restringir el segundo caso para que solo aplique cuando n no sea cero, haciendo los patrones disjuntos:

```haskell
esCero' :: Int -> Bool
esCero' 0 = True
esCero' n | n /= 0 = False
```

Aquí los patrones no se solapan, por lo que el verificador no necesita considerar prioridades. Cada ecuación cubre un conjunto de valores mutuamente excluyentes.

6.2.3. Extensionalidad.

En programación y matemáticas, el **principio de extensionalidad** nos da un criterio claro para determinar cuándo dos funciones son iguales, centrándose únicamente en su **comportamiento observable** y no en sus detalles internos.

Dados dos funciones $f, g :: A \to B$, decimos que f y g **son iguales,** si para todo valor de entrada x en el dominio A, se cumple que $f(x) = g(x)$.

Es decir, no importa cómo estén implementadas f y g (su código o eficiencia) sino que produzcan el mismo resultado para cada entrada posible. Por ejemplo, los algoritmos quicksort, insertionsort, mergesort y bubble sort son iguales, pues para una misma entrada (una lista) devuelven la misma salida (la lista ordenada).

De esta manera, podemos hacer **análisis por casos** sobre los elementos posibles del dominio A de la función.

Ejemplo.

Probaremos la siguiente propiedad utilizando los posibles casos de x, que es una variable booleana y cuyos valores son únicamente True y False:

$$\text{not (not x)} = \text{x}$$

```
1 not :: Bool -> Bool
2 not False = True
3 not True = False
```

Caso x = False.

$$\begin{aligned} &= \text{not (not False)} && \\ &= \text{not True} && \langle \text{not def 1} \rangle \\ &= \text{False} && \langle \text{not def 2} \rangle \end{aligned}$$

Caso x = True.

$$\begin{aligned} &= \text{not (not True)} && \\ &= \text{not False} && \langle \text{not def 2} \rangle \\ &= \text{True} && \langle \text{not def 1} \rangle \end{aligned}$$

6.2.4. Definición. Inducción.

Para poder probar propiedades acerca de programas recursivos usualmente necesitamos usar **inducción.** La inducción nos da una forma de escribir una prueba infinita de una manera finita.

1. **Inducción sobre los naturales.**

   Para probar $P(n)$ para todo $n \in \mathbb{N}$, probamos $P(0)$ y probamos que para cualquier m, si vale $P(m)$ entonces vale $P(m+1)$.

   - La prueba $P(0)$ se llama **caso base.**
   - La prueba de que $P(m) \to P(m+1)$ es el **paso inductivo.**
   - El suponer $P(m)$ verdadero es la **hipótesis de inducción.**

2. **Inducción fuerte sobre los naturales.**

   Para probar $P(n)$ para todo $n \in \mathbb{N}$, probamos que para cualquier m, si vale $P(i)$ para todo $i < m$, entonces vale $P(m)$.

- No hay caso base.
- Suponemos verdadero $P(i)$ para todo $i < m$ (hipótesis de inducción).

3. **Inducción sobre otros conjuntos.**

   Podemos usar la inducción sobre los naturales para obtener inducción sobre otros conjuntos. Por ejemplo, podemos hacer inducción sobre la altura de un árbol o la longitud de una lista.

   En general, dada una función $f : A \to \mathbb{N}$, y una propiedad P sobre elementos de A, podemos definir:

$$Q(n) = \text{si vale } f(a) = n \text{ entonces vale } P(a) \ (\forall a \in A)$$

   y así transformamos una propiedad sobre A en una sobre $\mathbb{N}$.

6.2.5. Ejemplo inducción sobre otros conjuntos.

Sea T un árbol binario definido como sigue, queremos probar la siguiente propiedad:

$$\forall t :: Bin : \ \text{cantleaf t} \leq \text{cantnode t} + 1$$

```
1  data Bin = Null | Leaf | Node Bin Bin
2
3  cantleaf :: Bin -> Int
4  cantleaf Null = 0
5  cantleaf Leaf = 1
6  cantleaf (Node t u) = cantleaf t + cantleaf u
7
8  cantnode :: Bin -> Int
9  cantnode (Node t u) = 1 + cantnode t + catnode u
10 cantnode _ = 0
11
12 height :: Bin -> Int
13 height (Node t u) = 1 + max (height t) (height u)
14 height _ = 0
```

Prueba por inducción. Definimos la siguiente propiedad a probar:

$$Q(n) = \text{height(t)} = n \ \Rightarrow \ \text{cantleaf t} \leq \text{cantnode t} + 1 \quad \forall t :: Bin$$

Usaremos la segunda forma de inducción (inducción fuerte) y suponemos por hipótesis inductiva (HI):

$$\forall i < n, \text{ si height(t)} = \text{i, entonces cantleaf t} \leq \text{cantnode t} + 1$$

Ahora haremos un análisis por casos de n:

- Si $n = 0$, entonces la hipótesis inductiva no se aplica y debemos probar directamente. Si height(t) $= 0$, entonces por definición de la función **height,** t tiene una hoja o está vacío:
  - Si tiene una hoja: cantleaf t $= 1 \leq$ cantnode t $+ 1 = 0 + 1 = 1$
  - Si está vacío: cantleaf t $= 0 \leq$ cantnode t $+ 1 = 0 + 1 = 1$

- Si $n > 0$ y height(t) = n, entonces podemos calcular:

$$
\begin{aligned}
&= \text{cantleaf t} && \\
&= \text{cantleaf (Node u v)} && \langle \text{height(t)} > 0 \rangle \\
&= \text{cantleaf u} + \text{cantleaf v} && \langle \text{cantleaf def 3} \rangle \\
&\leq \text{cantnode u} + 1 + \text{cantnode v} + 1 && \langle \text{HI (height u} < n \text{) y (height v} < n \text{)} \rangle \\
&= (1 + \text{cantnode u} + \text{cantnode v}) + 1 && \langle \text{asociatividad} \rangle \\
&= \text{cantnode (Node u v)} + 1 && \langle \text{cantnode def 1} \rangle \\
&= \text{cantnode t} + 1 &&
\end{aligned}
$$

Y luego por inducción hemos probado la propiedad Q. Así, hemos podido probar una propiedad sobre árboles usadno inducción sobre naturales.

Sin embargo, es más práctico hacer inducción directamente sobre la estructura del árbol, utilizando inducción estructural.

6.2.6. Definición. Inducción estructural.

Dada una propiedad P sobre un tipo de datos algebraico T, para probar que vale $P(t)$ $\forall t :: T$:

1. Probamos $P(t)$ para todo t dado por un constructor no recursivo.
2. Para todo t dado por un constructor con instancias recursivas $t_1, ...t_k$, probamos que si vale $P(t_i)$ para cada i, entonces vale $P(t)$.

6.2.7. Ejemplo inducción estructural para árboles.

Sea T un árbol binario definido, probaremos la misma propiedad que antes pero utilizando inducción estructural:

$$\forall t :: Bin : \ \text{cantleaf t} \leq \text{cantnode t} + 1$$

- **Caso Null:** cantleaf Null $= 0 \leq 1 = 0 + 1 =$ cantnode Null + 1
- **Caso Leaf:** cantleaf Leaf $= 1 \leq 1 = 0 + 1 =$ cantnode Leaf + 1
- **Caso Node u v:** utilizaremos las siguientes hipótesis inductivas:

$$\text{cantleaf u} \leq \text{cantnode u} + 1$$
$$\text{cantleaf v} \leq \text{cantnode v} + 1$$

$$
\begin{aligned}
&= \text{cantleaf (Node u v)} && \\
&= \text{cantleaf u} + \text{cantleaf v} && \langle \text{cantleaf def 3} \rangle \\
&= \text{cantnode u} + 1 + \text{cantnode v} + 1 && \langle \text{HI (height u} < n \text{) y (height v} < n \text{)} \rangle \\
&= (1 + \text{cantnode u} + \text{cantnode v}) + 1 && \langle \text{asociatividad} \rangle \\
&= \text{cantnode (Node u v)} + 1 && \langle \text{cantnode def 1} \rangle \\
&= \text{cantnode t} + 1 &&
\end{aligned}
$$

6.2.8. Ejemplo inducción estructural para listas.

Dada una propiedad P sobre listas, para probar que vale $P(xs)$ $\forall xs :: [a]$:

- Probamos $P([\,])$
- Probamos que si vale $P(xs)$ entonces vale $P(x : xs)$.

Ejemplo.

Probaremos la siguiente propiedad:

$$\text{reverse (xs ++ ys)} = \text{reverse ys ++ reverse xs}$$

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Probaremos por inducción estructural para listas sobre xs.

- **Caso xs = [].**

$$\begin{aligned}
&= \text{reverse ([ ] ++ ys)} && \\
&= \text{reverse ys} && \langle \text{++ def 1} \rangle \\
&= \text{reverse ys ++ [ ]} && \langle \text{++ def 1} \rangle \\
&= \text{reverse ys ++ reverse [ ]} && \langle \text{reverse def 1} \rangle
\end{aligned}$$

- **Caso xs.** Suponemos que $P(xs)$ es verdadero (HI) y probaremos para $P(x : xs)$.

$$\begin{aligned}
&= \text{reverse (x:xs ++ ys)} && \\
&= \text{reverse (xs ++ ys) ++ [x]} && \langle \text{reverse def 2} \rangle \\
&= \text{reverse ys ++ reverse xs ++ [x]} && \langle \text{HI} \rangle \\
&= \text{reverse ys ++ reverse (x:xs)} && \langle \text{reverse def 2 (al otro lado)} \rangle
\end{aligned}$$

6.2.9. Ejemplo. Compilador correcto.

Dado un lenguaje aritmético simple, cuyo AST se ve expresado en la definición data y su semántica denotacional está dada por el evaluador, queremos compilar el lenguaje a la máquina de stack:

```haskell
data Expr = Val Int | Add Expr Expr

-- Evaluador.
eval :: Expr -> Int
eval (Val n) = n
eval (Add x y) = eval x + eval y

-- Maquina de stack.
type Stack = [Int]
type Code = [Op]
data Op = PUSH Int | ADD

-- Dado un codigo y un stack, devuelve el stack luego de procesar el codigo.
exec :: Code -> Stack -> Stack
exec [] s = s
exec (PUSH n:c) s = exec c (n:s)
exec (ADD:c) (m:n:s) = exec c (n + m : s)
```

```
18
19 -- Compilador: traduce una expresion Expr a Code que puede ejecutar la maquina de
       pila.
20 comp :: Expr -> Code
21 comp (Val n) = [PUSH n]
22 comp (Add x y) = comp x ++ comp y ++ [ADD]
23
24 -- Ejemplo.
25 > e = Add (Add (Val 2) (Val 3)) (Val 4) -- :: Expr
26 > eval e = 2 + 3 + 4 = 9 -- :: Int
27
28 > comp e = comp (Add (Val 2) (Val 3)) ++ comp (Val 4) ++ [ADD]
29          = (comp (Val 2) ++ comp (Val 3) ++ [ADD]) ++ [PUSH 4] ++ [ADD]
30          = [PUSH 2] ++ [PUSH 3] ++ [ADD] ++ [PUSH 4] ++ [ADD]
31          = [PUSH 2, PUSH 3, ADD, PUSH 4, ADD]
32
33 > exec (comp e) [] = exec [PUSH 2, PUSH 3, ADD, PUSH 4, ADD] []
34                   = exec [PUSH 3, ADD, PUSH 4, ADD] [2]
35                   = exec [ADD, PUSH 4, ADD] [3, 2]
36                   = exec [PUSH 4, ADD] [5]
37                   = exec [ADD] [4, 5]
38                   = exec [] [9]
39                   = [9]
```

El compilador será correcto si se cumple la siguiente propiedad:

$$\text{exec (comp e) } [] = [\text{eval e}] \quad \forall e$$

7. Unidad 7 - Programando en paralelo.

7.1. Algoritmo Mergesort.

7.1.1. Definición. Algoritmo Mergesort.

El algoritmo de ordenación **mergesort** es un clásico ejemplo de Divide & Conquer: dividimos la entrada en dos (split), ordenamos recursivamente y juntamos las dos mitades ordenadas (merge).

Pseudocódigo del algoritmo.

```
msort : [Int] -> [Int]
msort []  = []
msort [x] = [x]
msort xs  =
    let (ls, rs) = split xs
        (ls', rs') = (msort ls || msort rs)
    in merge(ls',rs')

split : [Int] -> [Int] x [Int]
split []      = ([], [])
split [x]     = ([x], [])
split (x:y:zs)=
    let (xs, ys) = split zs
    in (x:xs, y:ys)

merge : [Int] x [Int] -> [Int]
merge ([], ys)      = ys
merge (xs, [])      = xs
merge (x:xs, y:ys)  = if x <= y
                        then x:merge(xs, y:ys)
                        else y:merge(x:xs, ys)
```

Trabajo del algoritmo.

- $Wsplit(n) \in O(n)$
- $Wmerge(n) \in O(n)$
- $Wmsort(n) = c_0 n + 2Wmsort(\frac{n}{2}) + c_1 n + c_2$

Por lo tanto, $Wmsort(n) \in O(n\ lg(n))$

Profundidad del algoritmo.

- $Ssplit(n) \in O(n)$
- $Smerge(n) \in O(n)$
- $Smsort(n) = k_0 n + Smsort(\frac{n}{2}) + k_1 n + k_2$

Por lo tanto, $Smsort(n) \in O(n)$. Este resultado nos muestra que el algoritmo mergesort no es muy paralelizable.

7.1.2. Paralelizando Mergesort.

Si recordamos la definición de paralelismo, $P = log(n)$ para este algoritmo, lo cual nos dice que muy pocos procesadores pueden usarse de forma eficiente. El problema aquí está en la profundiadd, ya que el trabajo no puede ser mejorado (se puede demostrar que cualquier algoritmo de ordenamiento basado en comparaciones es al menos $n\ log(n)$ en el trabajo).

Si analizamos la recurrencia de profundidad de *msort*:

$$Smsort(n) = Ssplit(n) + Smsort(\tfrac{n}{2}) + Smerge(n) + k$$

notamos que son las funciones merge y split quienes incurren un término lineal en la recurrencia. ¿Se pueden entonces mejorar las profundidades de ambas?

La respuesta es sí y no. Sí se pueden mejorar las profundidades de las tres funciones, pero no trabajando sobre listas. El problema no es el algoritmo de mergesort, sino la utilización de listas. En general las listas no son buenas para paralelizar: su modelo de construcción es claramente secuencial, nodo tras nodo hasta llegar al final.

La elección de la estructura de datos influye en la profundidad del algoritmo. **En lugar de listas trabajaremos con el siguiente tipo de árboles:**

data BT a = Empty — Node (BT a) a (BT a)

Paralelización más efectiva: debido a la estructura jerárquica de los árboles binarios de búsqueda, es más fácil dividir y procesar los datos de manera paralela. Cada subárbol puede ser procesado de forma independiente, lo que facilita la paralelización del algoritmo y puede llevar a una mejor utilización de los recursos de hardware en entornos paralelos. Veremos que podemos implementar msort sobre árboles con $W(n) = O(n\ log(n))$ y $S(n) = O(log(n)^3)$

7.1.3. Definición. Árboles de búsqueda (BT).

Un elemento de un **árbol de búsqueda (BT a)** está ordenado sii:

1. Es el árbol Empty.
2. Es un (Node l x r) y además:
   - l está ordenado.
   - r está ordenado.
   - todos los elementos en l son $\leq x$.
   - todos los elementos en r son $> x$.

Además, un árbol ordenado induce una lista ordenada (recorrido inorder):

```
listar :: BT a -> [a]
listar Empty = []
listar (Node l x r) = listar l ++ [x] ++ listar r
```

7.1.4. Mergesort sobre árboles.

Split en mergesort sobre árboles.

Tenemos que $Wsplit = O(1)$ y $Ssplit = O(1)$, pues el árbol ya está dividido en sus subárboles izquierdo y derecho.

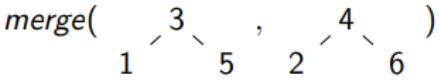
Msort en mergesort sobre árboles.

```
msort :: BT a -> BT a
msort Empty = Empty
msort (Node l x r) =
    let (l', r') = msort l || msort r
    in merge (merge l' r') (Node Empty x Empty)
```

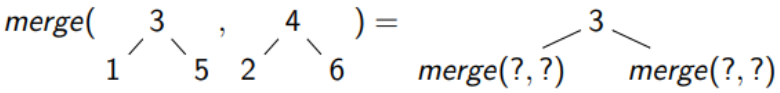
Merge en mergesort sobre árboles.

Queremos definir la función merge tal que $Wmerge = O(n)$ y $Smerge$ sea mejor que $O(n)$. ¿Cómo lo definimos?

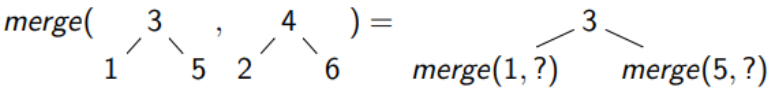
1. Consideremos el siguiente caso:



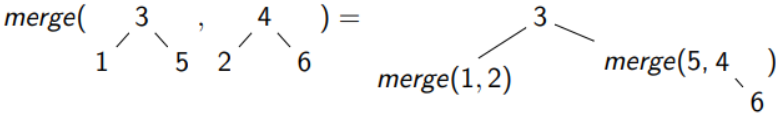
2. Elegimos guiarnos por el primer argumento y utilizar su raíz en el nuevo árbol mergeado:



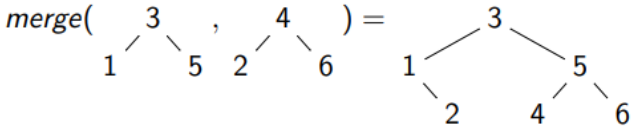
3. El 1 seguro va a la izquierda y el 5 a la derecha:



4. Separamos el segundo argumento en árboles menores a 3 y mayores a 3:



5. Finalmente obtenemos:



Implementación merge.

```haskell
merge :: BT Int -> BT Int -> BT Int
merge Empty t2 = t2
merge (Node l1 x r1) t2 =
    let (l2, r2) = splitAt t2 x
        (l', r') = merge l1 l2 || merge r1 r2
    in (Node l' x r')

splitAt :: BT Int -> Int -> BT Int x BT Int
splitAt Empty _ = (Empty, Empty)
splitAt (Node l x r) y =
    if y < x then let (ll, lr) = splitAt l y
                  in (ll, Node lr x r)
             else let (rl, rr) = splitAt r y
                  in (Node l x rl, rr)
```

7.1.5. Profundidad de merge en mergesort sobre árboles.

Profundidad de splitAt.

Sea h la altura del árbol, entonces $SsplitAt(h) = k + SsplitAt(h-1)$, y por lo tanto $SsplitAt \in O(h)$.

Profundidad de merge.

Sean h_1 y h_2 las alturas de los árboles argumento:

- $Smerge(h_1, h_2) \leq k + SsplitAt(h_2) + max(Smerge(h_1 - 1, h_{21}), Smerge(h_1 - 1, h_{22})$

  donde h_{21} y h_{22} son las alturas de los árboles devueltos por $splitAt$.

- $Smerge(h_1, h_2) \leq k + SsplitAt(h_2) + max(Smerge(h_1 - 1, h_2), Smerge(h_1 - 1, h_2))$

  pues estamos acotando $h_{21} \leq h_2$ y $h_{22} \leq h_2$.

- $Smerge(h_1, h_2) \leq k'h_2 + Smerge(h_1 - 1, h_2)$

  pues $SsplitAt(h_2) \in O(h_2)$. Luego, del segundo término $Smerge(h_1 - 1, h_2)$ resulta que estamos sumando h_1 veces el término $(k'h_2)$. Por lo tanto, $Smerge(h_1, h_2) \in O(h_1 \cdot h_2)$.

  Si n es el tamaño del árbol, y el árbol está balanceado entonces $h \in O(lg(n))$, y por lo tanto $Smerge(n) \in O(lg(n)^2)$.

Profundidad de msort.

- $Smsort(n) \leq k + max(Smsort(\frac{n}{2}), Smsort(\frac{n}{2})) + Smerge(lg(n), lg(n)) + Smerge(2lg(n), 1)$,

  donde el término $Smerge(2lg(n), 1))$ proviene de *merge (merge l' r') (Node Empty x Empty)*, y además *(merge l r)* $\leq$ *altura l + altura r*.

- $Smsort(n) \leq k + Smsort(\frac{n}{2}) + k_1(lg(n))^2 + k_2lg(n)$

  pues $Smerge(h_1, h_2) \in O(h_1, h_2)$ para el tercer término.

- $Smsort(n) \leq k + Smsort(\frac{n}{2}) + k_3(lg(n))^2$

  Y por lo tanto, $Smsort(n) \in O(lg(n)^3)$.

7.1.6. Función rebalance.

Pero este análisis de la profundidad tiene un error grave. La profundidad de merge suponía **árboles balanceados,** pero en msort llamamos a merge con el resultado de las llamadas recursivas. Este problema lo arreglamos con una función que rebalancee el árbol: **rebalance.**

```
msort :: BT a -> BT a
msort Empty = Empty
msort (Node l x r) =
    let (l', r') = msort l || msort r
    in rebalance (merge (merge l' r') (Node Empty x Empty))
```

7.2. Funciones de alto orden en árboles.

7.2.1. Funciones map, sum, flatten y reduce.

Trabajaremos ahora sobre otra variantes de árboles binarios:

```
data T a = Empty | Leaf a | Node (T a) (T a)
```

- **Función Map.** Recibe una función y se la aplica a cada elemento de un árbol dado.

```
mapT :: (a -> b) -> T a -> T b
mapT f Empty = Empty
mapT f (Leaf x) = Leaf (f x)
mapT f (Node l r) =
    let (l', r') = mapT f l || mapT f r
    in Node l' r'
```

  Si suponemos que $f \in O(1)$, entonces la función map tiene $Wmap = O(n)$ y $Smap = O(lg\ n)$.

- **Función Sum.** Suma todos los elementos de un árbol de enteros.

```
sumT :: T Int -> Int
sumT Empty = 0
sumT (Leaf x) = x
sumT (Node l r) =
    let (l', r') = sumT l || sumT r
    in l' + r'
```

  Para esta función tenemos $Wsum = O(n)$ y $Ssum = O(lg\ n)$.

- **Función Flatten.** Dado un árbol de cadenas, lo aplana uniendo todas sus cadenas en una sola.

```
flattenT :: T String -> String
flattenT Empty = []
flattenT (Leaf xs) = xs
flattenT (Node l r) =
    let (l', r') = flattenT l || flattenT r
    in l' ++ r'
```

- **Función Reduce.** Dada una operación binaria f, un neutro e y un árbol T a, combina recursivamente los valores del árbol usando f, devolviendo un único valor del mismo tipo.

```
reduceT :: (a -> a -> a) -> a -> T a -> a
reduceT f e Empty = e
reduceT f e (Leaf x) = x
reduceT f e (Node l r) =
    let (l', r') = reduceT f e l || reduceT f e r
    in f l' r'
```

  Si suponemos que $f \in O(1)$, entonces $Wreduce(n) \in O(n)$ y $Sreduce(n) \in O(lg\ n)$. Podemos utilizar la función reduce para expresar las funciones sumT y flattenT:

$$\text{sumT} = \text{reduceT}\ (+)\ 0$$

$$\text{flattenT} = \text{reduceT}\ (++)\ []$$

7.2.2. Ejemplo. Contar longitud de la línea con más palabras en un texto.

Queremos saber la longitud (en palabras) de la línea con más palabras en un texto:

$$lolile : String \to Int$$

```haskell
-- Recibe un string, le aplica la funcion words que divide una cadena en un arbol de palabras (separa por espacios) y le aplica la funcion que convierte cada palabra al numero 1. Finalmente, suma todos los numeros 1, equivalente a la cantidad de palabras que tiene un string (linea).
wordcount :: String -> Int
wordcount = sumT . mapT (\_ -> 1) . words

-- Recibe un string y aplica la funcion lines que divide una cadena en un arbol de lineas. Luego aplica la funcion wordcount a cada linea para contar cuantas palabras tiene. Por ultimo, con el reduce y la funcion de combinacion max 0, se queda con el maximo entre todos los conteos de palabras.
lolile = reduceT max 0 . mapT wourdcount . lines

-- texto = "hola\nesto es una prueba\nok"
-- lines texto -> ["hola", "esto es una prueba", "ok"]
-- mapT wordcount ... -> [1, 4, 1]
-- reduceT max 0 ... -> 4
-- lolile texto == 4
```

7.2.3. Mapreduce.

Hacer un mapT y luego un reduceT es ineficiente, ya que mapT genera un árbol que es inmediatamente consumido por reduceT:

- **mapT f t** recorre el árbol **t** aplicando **f** a cada hoja, generando un nuevo árbol con los valores transformados.
- **reduceT g e** vuelve a recorrer ese nuevo árbol, combinando los elementos con la función **g.**

Es decir, se recorren dos veces los árboles, y además se construye uno intermedio que no sirve más que para ser reducido. Esto es ineficiente tanto en tiempo como en espacio.

Solución: mapreduce.

La idea es fusionar esas dos operaciones en una sola función que:

- Transforma cada hoja con f (como lo haría mapT).
- Combina los resultados con g (como lo haría reduceT).

```haskell
mapreduce :: (a -> b) -> (b -> b -> b) -> b -> T a -> b
mapreduce f g e = mr
    where mr Empty = e
          mr (Leaf a) = f a
          mr (Node l r) = let (l', r') = mr l || mr r
                          in g l' r'
```

8. Unidad 8 - Secuencias.

8.1. Secuencias.

8.1.1. Definición. Secuencias.

Seq es un TAD para representar secuencias de elementos. A continuación veremos algunas de sus operaciones y las especificaremos en términos de la noción matemática de secuencias.

- Denotamos la longitud (cantidad de elementos) de una secuencia s como $|s|$.
- Denotamos las secuencias como $\langle x_0, x_1, ..., x_n \rangle$.
- Un índice i es válido en s si $0 \leq i < |s|$.
- Para i válido y secuencia s, notamos como s_i a su i-ésima proyección (elemento).
- s' es subsecuencia de s si existe secuencia I de índices válidos en s estrictamente crecientes tal que $s'_i = S_{I_i}$. Por ejemplo, dada la secuencia $\langle 0, 1, 2, 3, 4, 5, 6 \rangle$, la secuencia $\langle 2, 5, 6 \rangle$ es un subsecuencia de la primera.

8.1.2. Operaciones en secuencias.

Sea s una secuencia y x un elemento, se definen las siguientes operaciones:

```
empty : Seq a
-- Representa la secuencia vacia <>

singleton : a -> Seq a
-- singleton x devuelve la secuencia <x>

length : Seq a -> Nat
-- length s devuelve |s| (longitud de la secuencia).

nth : Seq a -> Nat -> a
-- Si i es un indice valido, entonces nth s i devuelve la i-esima posicion de s.

toSeq : [a] -> Seq a
-- Dada una lista xs, toSeq xs nos devuelve la representacion de xs como una
    secuencia (respetando indices).
```

8.1.3. Operaciones de alto orden en secuencias.

```
tabulate : (Nat -> a) -> Nat -> Seq a
-- tabulate f n evalua a la secuencia <f 0, f 1, f 2, ... f (n - 1)>

map : (a -> b) -> Seq a -> Seq b
-- map f s evalua a la secuencia <f s_0, f s_1, f s_2, ... f s_(|s|-1)>

filter : (a -> Bool) -> Seq a -> Seq a
-- filter p s evalua a la subsecuencia mas larga en la que p vale para todos sus
    elementos

append : Seq a -> Seq a -> Seq a
-- append s t es la secuencia <s_0, s_1, ... s_(|s|-1), t_0, t_1, ... t_(|t|-1)>

take : Seq a -> Nat -> Seq a
-- take s n devuelve la secuencia hasta el indice n si n < |s|, o devuelve la misma
    secuencia si el n argumento es mayor que la longitud.

drop : Seq a -> Nat -> Seq a
-- drop s n devuelve la secuencia que es igual a s pero sin los primeros n-1
    indices: <s_n, ... s_(|s|-1)>
```

8.1.4. Vista de secuencias como árbol.

La siguiente operación nos permite ver a las secuencias como si fueran un árbol. La idea de descomponer la secuencia en una estructura de árbol es para facilitar ciertas operaciones. Notar que *TreeView* no es un tipo recursivo.

```
data TreeView a = EMPTY | ELT a | NODE (Seq a) (Seq a)

showt : Seq a -> TreeView a
-- Si |s| = 0, showt s evalua a EMPTY.
-- Si |s| = 1, showt s evalua a ELT s_0.
-- Si |s| > 1, showt s evalua a NODE (take s |s|/2) (drop s |s|/2)
```

8.1.5. La operación Foldr.

La función **foldr (fold right)** es una operación de plegado que procesa una secuencia de elementos aplicando una función binaria de derecha a izquierda, acumulando un resultado.

$$\boxed{\textbf{foldr : (a} \to \textbf{b} \to \textbf{b)} \to \textbf{b} \to \textbf{Seq a} \to \textbf{b}}$$

- $(a \to b \to b)$: es una función de 2 argumentos que toma un elemento de la secuencia (a) y un acumulador (b), y devuelve un nuevo acumulador (b).
- b: es el valor inicial del acumulador.
- *Seq a*: es la secuencia de elementos que se va a procesar.
- b : es el valor resultante después de aplicar la función a todos los elementos de la secuencia (acumulador final).

Ejemplo.

```
foldr (+) e s = s_0 + (s_1 + (... (s_n + e)))
-- e es el valor inicial del acumulador, en este caso, la secuencia vacia.
-- foldr va aplicando la funcion (+) a todos los elementos de la secuencia, en este
     caso de derecha a izquierda (por la r de foldR).

foldl :: (b -> a -> b) -> b -> Seq a -> b
-- foldl evalua de izquierda a derecha.
```

8.1.6. La operación Reduce.

La función **reduce** se utiliza para reducir una secuencia de valores a un único valor. Recibe como argumento una función que define como se va a combinar cada elemento de la secuencia para obtener un resultado final.

$$\boxed{\textbf{reduce : (a} \to \textbf{a} \to \textbf{a)} \to \textbf{a} \to \textbf{Seq a} \to \textbf{a}}$$

- $(a \to a \to a)$: toma dos elementos del mismo tipo y devuelve un solo elemento del mismo tipo.
- El segundo argumento a es el elemento neutro. En *reduce* $\oplus$ e, el valor e corresponde a la secuencia vacía.
- **Comportamiento para funciones asociativas y no asociativas.**

  Si $\oplus$ es es asociativa, *reduce* $\oplus$ e s, evalúa a la suma con respecto a $\oplus$ de la secuencia s. Si además e es neutro respecto de $\oplus$, reduce coincide con *foldr* $\oplus$ e y *foldl* $\oplus$ e.

  Si $\oplus$ no es asociativa, obtenemos diferentes resultados dependiendo del orden de reducción.

Limitarse a funciones asociativas no es una buena idea: la suma y multiplicación de punto flotante no es asociativa, por ejemplo.

- **Especificación del orden de reducción.**

  Para especificar el comportamiento de *reduce* debemos especificar el orden de reducción. El orden de reducción es parte del TAD y no de una implementación en particular.

8.1.7. Orden de reducción de reduce.

Definiremos la operación **reduce** usando un árbol de combinación para poder especificar el **orden de reducción** de una secuencia de manera clara y eficiente.

Un **árbol de combinación** es una estructura de datos utilizada en algoritmos de división y conquista para procesar eficientemente grandes conjuntos de datos. En el contexto del cálculo paralelo y distribuido, los árboles de combinación son especialmente útiles para paralelizar y optimizar operaciones como la reducción, la búsqueda y la clasificación.

En esencia, un árbol de combinación es un árbol binario que se utiliza para dividir una tarea en partes más pequeñas y luego combinar los resultados de manera eficiente.

```
1 data Tree a = Leaf a | Node (Tree a) (Tree a)
2
3 toTree :: Seq a -> Tree a
4 toTree s = case |s| of
5     1 -> (Leaf s_0)
6     2 -> Node (toTree (take s pp)) (toTree (drop s pp))
7     where
8         pp = 2^(ilg (n-1))
```

Donde *ilg* es el logaritmo entero. El propósito del valor *pp* es encontrar el tamaño más grande de una subsecuencia que equilibra el árbol de combinación. El primer subárbol buscara ser un árbol completo: esto sirve para maximizar la cantidad de operaciones que podemos paralelizar.

Si $|s| = 2^k$, el resultado es un árbol binario perfecto. La operación $2^{ilg(n-1)}$ devuelve la potencia de 2 más grande que entra en n.

Ahora podemos definir reduce sobre árboles:

```
1 reduceT (+) (Leaf x) = x
2 reduceT (+) (Node l r) = (reduceT (+) l) + (reduceT (+) r)
```

Con esto podemos especificar `reduce` para secuencias, utilizando la vista de árbol:

- Si $|s| = 0$, *reduce* $\oplus$ b s evalúa a b.
- Si $|s| > 0$ y *reduceT* $\oplus$ *(toTree s)* evalúa a v, entonces *reduce* $\oplus$ b s evalúa a $(b \oplus v)$.

Observación. Con esto no estamos dando una implementación de *reduce*, solo especificando el orden de reducción.

Ejemplo. Si tenemos la secuencia $s = \langle s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8 \rangle$, entonces (seguir orden con el valor pp del logaritmo entero):

$$\text{reduce} \oplus \text{b s} = b \oplus ((((s_0 \oplus s_1) \oplus (s_2 \oplus s_3)) \oplus ((s_4 \oplus s_5) \oplus (s_6 \oplus s_7))) \oplus (s_8 \oplus s_9))$$

8.1.8. Divide and Conquer con reduce.

Podemos definir un algoritmo divide and conquer genérico como sigue, solo faltaría instanciar los valores de **val, base** y **combine:**

```
dyc s = case showt s of
    EMPTY -> val
    ELT v -> base v
    NODE l r -> let (l', r') = dyc l || dyc r
                in combine l' r'
```

Pero esta función la podemos definir en una sola línea utilizando reduce:

reduce **combine val** (map **base** s)

8.1.9. MCSS usando reduce.

La **máxima subsecuencia contigua** es una subsecuencia de números en una lista tal que la suma de sus elementos es máxima entre todas las subsecuencias posibles.

Por ejemplo, si se considera la secuencia de números $\langle -2, 1, -3, 4, -1, 2, 1, -5, 4\rangle$, la máxima subsecuencia contigua sería:

$$\langle 4, -1, 2, 1\rangle$$

ya que la suma de estos elementos es 6, que es la mayor suma posible dentro de todas las subsecuencias contiguas de la secuencia original.

El problema de encontrar la máxima subsecuencia contigua se puede resolver con el patrón Divide and Conquer. La solución hace uso del tuppling: devolver no solo el resultado final, sino otros cómputos parciales que ayudarán al cómputo final. La solución de MCSS devuelve una tupla con:

1. El resultado deseado: representa la máxima subsecuencia contigua (el valor final que devolveremos).
2. El máximo prefijo: la máxima subsecuencia contigua que termina en la posicióna actual.
3. El máximo sufijo: la máxima subsecuencia contigua que comienza en la posición actual.
4. La suma total: la suma total de la subsecuencia actual.

El código de esta solución usando Divide and Conquer sigue la siguiente estructura:

```
val = (0, 0, 0, 0)

base v = let v' = max v 0
         in (v', v', v', v)

combine (m, p, s, t) (m', p', s', t') =
    let resDeseado = max (s + p') m m'
        maxPrefijo = max p (t + p')
        maxSufijo = max s' (s + t')
        sumTotal = t + t'
    in (resDeseado, maxPrefijo, maxSufijo, sumTotal)
```

En lugar de utilizar la función **dyc** definida en la sección anterior, podemos definir la solución en una sola línea utilizando reduce:

reduce **combine val** (map **base** s)

8.1.10. La operación Scan.

La operación **scan** es una variante de reduce que en lugar de devolver un solo valor, devuelve una secuencia de valores intermedios obtenidos durante el proceso de reducción. Esto permite ver cómo se acumulan los resultados parciales.

Por ejemplo, scan $\oplus$ 0 $\langle 1, 2, 3\rangle = (\langle 0, 1, 3, 6\rangle, 6)$

$$\boxed{\textbf{scan : (a} \to \textbf{a} \to \textbf{a)} \to \textbf{a} \to \textbf{Seq a} \to \textbf{(Seq a, a)}}$$

Cuando $\oplus$ es asociativa, *scan* $\oplus$ *b s* es equivalente a:

$$\text{scan } \oplus \text{ b s} = (\text{tabulate } (\lambda i \to \text{reduce } \oplus \text{ b (take s i)}), \text{ reduce } \oplus \text{ b s})$$

El tabulate significa: para cada i, con i variando de 0 a n, calcula la suma/operación acumulada hasta el elemento i-1. Por ejemplo, para el índice 1, toma el primer elemento de la secuencia y lo reduce con b; para el segundo índice, toma los dos primeros elementos de la secuencia y los reduce con b.

Notar que esta es una especificación: como implementación sería muy ineficiente. Aparte solo se especifica para $\oplus$ asociativa. Notablemente, **scan** puede implementarse con estos costos:

$$\text{W(scan } \oplus \text{ l s)} = O(|s|)$$
$$\text{S(scan } \oplus \text{ i s)} = O(lg|s|)$$

8.1.11. MCSS usando scan.

El problema de la máxima subsecuencia contigua puede escribirse matemáticamente como:

$$\begin{aligned} mcss\ s &= \max_{0 \le i \le j \le |s|} \left(\sum_{k=i}^{j-1} s_k \right) \\ &= \max_{0 \le i \le j \le |s|} \left(\sum_{k=0}^{j-1} s_k - \sum_{l=0}^{i-1} s_l \right) \\ &= \max_{0 \le i \le j \le |s|} (X_j - X_i) \quad \text{donde X = scan (+) 0 s} \\ &= \max_{0 \le j \le |s|} \left(X_j - \min_{0 \le i \le j} X_i \right) \end{aligned}$$

Esto se puede modelar con un algoritmo utilizando **scan:**

```
mcss s =
    let x = scan (+) 0 s
        m = scan min \infty x
    in max (tabulate (\j -> x_j - m_j) |s|)
```

8.2. Arreglos persistentes.

8.2.1. Definición. Arreglos persistentes.

Los **arreglos persistentes** son estructuras de datos que no destruyen el arreglo original al realizar un cambio, sino que lo copian y le agregan la modificación a la copia.

Al igual que los arrays normales, se puede acceder en tiempo constante a cualquier índice. Sin embargo, no se pueden hacer updates destructivos.

Tienen operaciones para crear arreglos a partir de una lista, a partir de funciones y a partir de otros arreglos:

```
length :: Arr a -> Int
-- length p devuelve el tamanio del arreglo p.

nth :: Arr a -> Int -> a
-- nth p i devuelve el i-esimo elemento del arreglo p.

fromList :: [a] -> Arr a
-- construye un arreglo persistente a partir de una lista.

tabulate :: (Int -> a) -> Int -> Arr a
-- tabulate f n construye un arreglo p de tamanio n tal que nth p i == f i.

subarray :: Arr a -> Int -> Int -> Arr a
-- subarray a i l construye el subarreglo de a que comienza en el indice i y es de
    longitud l
```

8.2.2. Costos de operaciones en arreglos persistentes.

Operación	W	S
length p	$O(1)$	$O(1)$
nth p i	$O(1)$	$O(1)$
fromList xs	$O(\|xs\|)$	$O(\|xs\|)$
tabulate f n	$O\left(\sum_{i=0}^{n} W(f\ i)\right)$	$O\left(\max_{i=0}^{n} S(f\ i)\right)$
subarray a i l	$O(1)$	$O(1)$

Observaciones.

- La longitud del arreglo se almacena explícitamente junto con el arreglo en la estructura de datos. Esto permite que la operación *length* simplemente acceda a este valor almacenado, en lugar de tener que recalcularlo. Debido a esto, la operación *length* puede ser realizada en tiempo constante $O(1)$.
- La complejidad de la función *subarray* es constante porque en la estructura del arreglo tenemos un puntero al inicio y la longitud del arreglo. Por lo tanto, para obtener un subarreglo, simplemente se cambia el valor de la longitud del arreglo que pasamos como argumento.

8.2.3. Especificación de costo de secuencias basada en arreglos persistentes.

Podemos implementar secuencias usando arreglos persistentes con los siguientes costos:

Operación	**W**	**S**
empty	$O(1)$	$O(1)$
singleton	$O(1)$	$O(1)$
length	$O(1)$	$O(1)$
nth	$O(1)$	$O(1)$
take s n	$O(1)$	$O(1)$
drop s n	$O(1)$	$O(1)$
showt s	$O(1)$	$O(1)$
tabulate f n	$O\left(\sum_{i=0}^{n-1} W(f\ i)\right)$	$O\left(\max_{i=0}^{n-1} S(f\ i)\right)$
map f s	$O\left(\sum_{x\in s} W(f\ x)\right)$	$O\left(\max_{x\in s} S(f\ x)\right)$
filter f s	$O\left(\sum_{x\in s} W(f\ x)\right)$	$O\left(lg(\|s\|) + \max_{x\in s} S(f\ x)\right)$
reduce $\oplus$ b s	$O(\|s\|)$	$O(lg(\|s\|)$
scan $\oplus$ b s	$O(\|s\|)$	$O(lg(\|s\|))$

Para las funciones *tabulate, map* y *filter* se ha especificado el costo para cualquier función argumento. Sin embargo, para *reduce* y *scan*, $W(n) \in O(|s|)$ y $S \in O(|s|)$ sólo si la función argumento es $O(1)$.

8.2.4. Análisis de costos de reduce en secuencias con arreglos persistentes.

En *reduce* $\oplus$ *b s*, el orden de reducción afecta el resultado obtenido si la operación $\oplus$ no es asociativa. Si además $\oplus \notin O(1)$, el orden de reducción también afecta los costos.

Consideremos el siguiente algoritmo de ordenación:

sort s = reduce merge empty (map singleton s)

Veamos un ejemplo. Analicemos el costo de *sort* si tenemos los siguientes datos:

- $W(merge\ s\ t) = O(|s| + |t|)$
- $S(merge\ s\ t) = O(lg(|s| + |t|))$
- Orden de reducción: $(x_0 \oplus (x_1 \oplus (... \oplus\ x_{n-1})...))$

Sea $n = |s|$, calculemos el trabajo y la profunidad de sort:

- Como en cada paso i, se mezcla una lista de tamaño 1 con otra de tamaño i (por el acumulador creciente). Así, el costo por paso es $O(1 + i)$, por el trabajo del *merge* para dos listas.

Multiplicando por todos los elementos de la secuencia obtenemos la recurrencia:

$$\begin{aligned} W(sort\ s) &\leq \sum_{i=1}^{n-1} c \cdot (1+i) \\ &= c\left(\sum_{i=1}^{n-1} 1 + \sum_{i=1}^{n-1} i\right) \\ &= c\left((n-1) + \frac{(n-1)n}{2}\right) \\ &= c\left((n-1) + \frac{n^2-n}{2}\right) \\ &= c \cdot \frac{n^2+n-2}{2} \in O(n^2) \end{aligned}$$

- Como cada *merge* tiene profundidad $O(lg(1+i))$, la suma de logaritmos nos da la recurrencia obtenida.

$$\begin{aligned} S(sort\ s) &\leq \sum_{i=1}^{n-1} c \cdot lg(1+i) \\ &= c \cdot (lg(2) + lg(3) + \cdots + lg(n)) \\ &= c \cdot lg(2 \cdot 3 \cdots n) \\ &= c \cdot lg(n!) \in O(n\ lg(n)) \quad \text{(aproximación de Stirling)} \end{aligned}$$

Si en cambio usamos el orden de reducción dado anteriormente con el valor $pp = 2^{ilg(n-1)}$ tenemos una mejora significante sobre el trabajo del sort:

$$W(sort\ s) \in O(n\ lg(n))$$

Conclusión. El orden de reducción afecta no sólo el resultado ($\oplus$ no asociativa), si no también el costo. Para definir el costo en general de *reduce* definimos el conjunto:

$$\vartheta_r(reduce \oplus b\ s) = \{\text{aplicaciones de } \oplus \text{ en el árbol de reducción}\}$$

Y usando este conjunto, el costo de *reduce* es:

- $W(reduce \oplus b\ s) = O\left(|s| + \sum_{(x\oplus y)\in\vartheta_r(\oplus,b,s)} W(x \oplus y)\right)$
- $W(reduce \oplus b\ s) = O\left(lg(|s|) \cdot \max_{(x\oplus y)\in\vartheta_r(\oplus,b,s)} W(x \oplus y)\right)$

8.2.5. Implementación de Scan.

Recordemos la función **scan:** es una variante de reduce que en lugar de devolver un solo valor, devuelve una secuencia de valores intermedios obtenidos durante el proceso de reducción. Esto permite ver cómo se acumulan los resultados parciales.

scan : (a → a → a) → a → Seq a → (Seq a, a)

Cuando $\oplus$ es asociativa, *scan* $\oplus$ *b s* es equivalente a:

scan ⊕ b s = (tabulate ($\lambda i \rightarrow$ reduce ⊕ b (take s i)), reduce ⊕ b s)

Pero esto no es una implementación eficiente: la operación scan parece ser inherentemente secuencial. ¿Cómo implementarla en paralelo?

Divide and Conquer vs. Contracción

- **Divide and Conquer:** partimos la secuencia en dos y llamamos recursivamente. ¡Pero la mitad derecha depende de la izquierda!
- **Contracción de entrada:** para resolver un problema, resolvemos una instancia más chica del mismo problema. A diferencia de Divide and Conquer, hacemos solo una llamada recursiva. Tenemos los siguientes pasos:
  1. Contraemos la instancia del problema a una instancia (mucho) más chica (contracción).
  2. Resolvemos recursivamente la instancia chica.
  3. Usamos la solución de la instancia chica para resolver la grande (expansión).

  Este enfoque es útil en algoritmos paralelos: contracción y expansión son usualmente parale-lizables y si contraemos a una fracción, al recursión tiene profundidad logarítmica.

Ejemplo de contracción en reduce.

1. Consideramos $reduce\ +\ 0\ \langle 2,3,6,2,1,4\rangle$.
2. Contraemos la secuencia de la siguiente manera: $\langle 2+3, 6+2, 1+4\rangle = \langle 5,8,5\rangle$
3. Calculamos recursivamente $reduce\ +\ 0\langle 5,8,5\rangle = 18$.
4. En este caso la expansión es la función identidad: el resultado es 18.

Ejemplo de contracción/expansión en Scan.

Supongamos que $\oplus$ es asociativa y comparemos la implementación dada de scan (1) con la implementación con contracción y expansión (2):

1. $scan \oplus b\ \langle x_0, x_1, x_2, x_3\rangle = (\langle b, b\oplus x_0, b\oplus x_0\oplus x_1, b\oplus x_0\oplus x_1\oplus x_2\rangle, b\oplus x_0\oplus x_1\oplus x_2\oplus x_3)$
2. $scan \oplus b\ \langle x_0, x_1, x_2, x_3\rangle = scan \oplus b\ \langle x_0\oplus x_1, x_2\oplus x_3\rangle = (\langle b, b\oplus x_0\oplus x_1\rangle, b\oplus x_0\oplus x_1\oplus x_2\oplus x_3)$

Implementación de contracción/expansión en Scan.

Para la implementación, buscamos que dada como entrada la secuencia s y el resultado s' del llamado recursivo sobre la contracción, obtener r como sigue:

- $s = \langle x_0, x_1, x_2, x_3\rangle$
- $s' = (\langle b, b\oplus x_0\oplus x_1\rangle, b\oplus x_0\oplus x_1\oplus x_2\oplus x_3)$
- $r = (\langle b, b\oplus x_0, b\oplus x_0\oplus x_1, b\oplus x_0\oplus x_1\oplus x_2\rangle, b\oplus x_0\oplus x_1\oplus x_2\oplus x_3)$

El siguiente proceso nos da un algoritmo paralelo para implementar **scan** y también fija el orden de reducción:

1. El total (segunda componente del par que devuelve scan) lo obtenemos del resultado del llamado recursivo s'.
2. Para construir la secuencia (primera componente del par que devuelve scan) se sigue el siguiente proceso, donde r_i son elementos de la secuencia final y s'_i de la secuencia del llamado recursivo contraído:
   a) $r_i = s'_{\frac{i}{2}}$ si i es par.
   b) $r_i = s'_{\lfloor \frac{i}{2}\rfloor} + s_{i-1}$

Observar en r lo siguiente: $r_0 = s'_0 = b$ (i par), $r_1 = s'_0 + s_0 = b + x_0$ (i impar), $r_2 = s'_1 = b\oplus x_0\oplus x_1$ (i par).

Orden de reducción de Scan.

Si la operación $\oplus$ no es asociativa el orden de reducción es importante. Con el algoritmo anteriormente descripto tenemos el siquiente orden de reducción que no coincide con el orden de reducción de *reduce* sobre los prefijos:

$$\begin{aligned} scan \oplus b\ \langle x_0, x_1, x_2, x_3, x_4, x_5\rangle = (\langle & b, \\ & b \oplus x_0, \\ & b \oplus (x_0 \oplus x_1), \\ & (b \oplus (x_0 \oplus x_1)) \oplus x_2, \\ & b \oplus ((x_0 \oplus x_1) \oplus (x_2 \oplus x_3)), \\ & (b \oplus ((x_0 \oplus x_1) \oplus (x_2 \oplus x_3))) \oplus x_4\rangle, \\ & b \oplus (((x_0 \oplus x_1) \oplus (x_2 \oplus x_3)) \oplus (x_4 \oplus x_5))) \end{aligned}$$

Costo de Scan.

Calculamos costos para el scan en la implementación con arreglos. Suponemos que $\oplus$ tiene costo constante. Los costos de Scan son:

- $W(n) = W(\frac{n}{2}) + kn \implies W(n) \in O(n)$
- $S(n) = S(\frac{n}{2}) + k \implies S(n) \in O(lg(n))$

Y definiendo el conjunto de aplicaciones de $\oplus$ en scan como:

$$\vartheta_s(scan \oplus b\ s) = \{\text{aplicaciones de } \oplus \text{ en el árbol de reducción}\}$$

Llegamos a los siguientes costos para scan:

- $W(scan \oplus b\ s) = O\left(|s| + \sum_{(x \oplus y) \in \vartheta_s(\oplus, b, s)} W(x \oplus y)\right)$
- $S(scan \oplus b\ s) = O\left(lg(|s|) \cdot \max_{(x \oplus y) \in \vartheta_s(\oplus, b, s)} S(x \oplus y)\right)$

8.3. Más funciones de alto orden para secuencias.

8.3.1. Definición. La operación Collect.

Supongamos que trabajamos con secuencias *Seq a*, donde existe un orden total para los elementos en a. Esto induce un orden lexicográfico sobre secuencias.

Sea a un tipo con un orden total (el tipo de las claves) y sea b un tipo cualquiera (el tipo de los datos), entonces:

La función **collect :: Seq (a, b) → Seq (a, Seq b)**, recolecta todos los datos asociados a cada clave. La secuencia resultado estará ordenada según el orden de a.

$$\begin{aligned} &= collect\ \langle(3,'EDyAII'), (1,'Prog1'), (2,'Prog2'), (2,'EDyAI'), (3,'SOI')\rangle \\ &= \langle(1, \langle'Prog1'\rangle), (2, \langle'Prog2','EDyAI'\rangle), (3, \langle'EDyAII','SOI'\rangle)\rangle \end{aligned}$$

8.3.2. Implementación de Collect.

Collect se puede implementar en dos pasos:

1. Ordenar la entrada según las claves. Esto tiene como efecto juntar claves iguales.

2. Juntar todos los valores de claves iguales.

Y el costo de Collect se puede dividir en:

- Ordenar tiene $W(n) \in O(W_c \cdot n \cdot lg(n))$ y $S(n) \in O(S_c \cdot lg^2(n))$, donde W_c y S_c son el trabajo y profundidad de la comparación de claves.
- Juntar todas las claves es $W(n) \in O(n)$ y $S(n) \in O(lg(n))$.

Por lo tanto, el costo del Collect está dominado por el costo de la ordenación.

8.3.3. La operación MAP/COLLECT/REDUCE: Map-reduce de Google.

Esta operación implica hacer un **map,** seguido de un **collect,** seguido de varios **reduce.**

Ejemplo: asociar a cada palabra clave una secuencia de documentos donde ésta aparece.

- *map apv* se aplica a una secuencia de documentos, donde *apv* transforma cada documento en secuencias de pares clave/valor.
- Esto último devuelve una secuencia de secuencias de pares clave/valor que se aplana con *join.*
- Luego, se aplica un *collect* para combinar las claves con todos los documentos donde esta aparece.
- Luego, cada par de clave y secuencia de valores es 'reducido' a un resultado con *red.*

Generalmente *apv* y *red* son funciones secuenciales. El paralelismo viene de aplicarlas en un *map:*

```
mapCollectReduce apv red s =
    let pairs = join (map apv s)
        groups = collect pairs
    in map red groups
```

Ejemplo. ¿Cuántas veces aparece una palabra en una colección de documentos?

```
countWords s = mapCollectReduce apv red s

-- Esta funcion aplica words sobre el documento d, que devuelve una secuencia de palabras separadas por espacios. Luego, a traves del map, asigna el par (palabra, 1) a cada palabra devuelta por words.
apv d = map (\w -> (w, 1)) (words d)

-- Como luego se aplica un join (map apvs s) en mapCollectReduce, se aplana esta secuencia de pares.

-- Luego, con el collect se recolectan pares (palabra, secuenciaDeUnos), donde la secuenciaDeUnos tendra un elemento por cada apariencia de la palabra. Es decir, length s = cantVeces Palabra.

-- Con el map red groups, sumamos la cantidad de unos que aparece en cada secuencia del par (palabra, secuenciaDeUnos) para obtener finalmente una secuencia de pares (palabra, cantVecesAparece).
red (w, s) = (w, reduce (+) 0 s)
```