Estructuras de Datos y Algoritmos 2

Ignacio Rimini

March 2025

Índice

1. Unidad 1 - Modelo de Costos. 3
1.1. Notación asintótica. . . . 3
1.1.1. Definición. Notación Big O. . . . 3
1.1.2. Definición. Notación Big Omega. . . . 3
1.1.3. Definición. Notación Big Theta. . . . 3
1.1.4. Proposición. Caracterización de notación Big Theta. . . . 3
1.2. Convenciones matemáticas. . . . 4
1.2.1. Definición. Piso de un número. . . . 4
1.2.2. Definición. Techo de un número. . . . 4
1.2.3. Definición. Función asintóticamente no negativa. . . . 4
1.2.4. Funciones logarítmicas. . . . 4
1.2.5. Funciones exponenciales. . . . 4
1.2.6. Series matemáticas. . . . 5
1.3. Modelo de costos en algoritmos. . . . 5
1.3.1. Definición. Trabajo (W). . . . 5
1.3.2. Definición. Profundidad (S). . . . 5
1.3.3. Definición. Paralelismo. . . . 6
1.3.4. Principio del Scheduler Voraz (Brent). . . . 6
1.3.5. Pseudocódigo y costos de programas. . . . 6
1.4. Divide and Conquer. . . . 7
1.4.1. Algoritmo Divide and Conquer. . . . 7
1.4.2. Trabajo y profundidad de algoritmos Divide and Conquer. . . . 7
1.4.3. Algoritmo MergeSort. . . . 8

2. Unidad 2 - Resolución de Recurrencias. 10
2.1. Introducción. . . . 10
2.2. Método de sustitución. . . . 10
2.2.1. Definición. Método de sustitución. . . . 10
2.2.2. Ejemplo. Sustitución para el mergesort. . . . 10
2.2.3. Acerca de la adivinanza. . . . 11
2.3. Árboles de recurrencia. . . . 12
2.3.1. Definición. Árbol de recurrencia. . . . 12
2.3.2. Ejemplo. Árbol de recurrencia. . . . 12
2.4. Funciones suaves y reglas de suavidad. . . . 14
2.4.1. Introducción. . . . 14
2.4.2. Definición. Función eventualmente no decreciente. . . . 14
2.4.3. Definición. Función b-suave. . . . 14
2.4.4. Definición. Función suave. . . . 14
2.4.5. Proposición. Condición suficiente de suavidad. . . . 14
2.4.6. Teorema. Regla de suavidad. . . . 14
2.4.7. Ejemplo. Recurrencia del mergeSort. . . . 15
2.5. Teorema maestro. . . . 16

2.5.1. Definición. Teorema maestro. 16

3. Unidad 3 - Programación Funcional con Haskell **17**
3.1. Programación funcional. 17
3.1.1. ¿Qué es la programación funcional? 17
3.1.2. Ventajas de Haskell. 17
3.2. Introducción a Haskell. 17
3.2.1. Guía inicial para compilar y ejecutar un archivo. 17
3.2.2. Comentarios y palabras reservadas. 18
3.2.3. Offside rule. 18
3.2.4. Operadores infijos. 18
3.2.5. Tipos. 18
3.3. Listas en Haskell. 19
3.3.1. Definición. Lista. 19
3.3.2. Operaciones básicas con listas. 19
3.3.3. Funciones de listas. 20
3.3.4. Listas por comprensión. 21
3.3.5. Patrones de listas. 21
3.4. Tuplas en Haskell. 21
3.4.1. Definición. Tupla. 21
3.4.2. Acceso a los elementos. 22
3.5. Strings en Haskell. 22
3.6. Expresiones condicionales en Haskell. 22
3.6.1. If-then-else. 22
3.6.2. Ecuaciones con guardas. 23
3.6.3. Pattern Matching. Coincidencia de patrones. 23
3.7. Funciones en Haskell. 24
3.7.1. Definición. Función. 24
3.7.2. Aplicación de funciones, asociatividad y precedencia. 24
3.7.3. Declaración de funciones. 24
3.7.4. Llamada a funciones. 25
3.7.5. Currying. Currificación y aplicación parcial. 25
3.7.6. Variables locales en funciones. 25
3.7.7. Recursión. 26
3.7.8. Funciones Lambda. Funciones anónimas. 26
3.7.9. Polimorfismo paramétrico en funciones. 27
3.7.10. Polimorfismo ad-hoc de sobrecarga de funciones. 27
3.8. Clases de tipo. 28
3.8.1. Definición. Clases de tipos. 28
3.8.2. Algunas clases de tipo. 28

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.

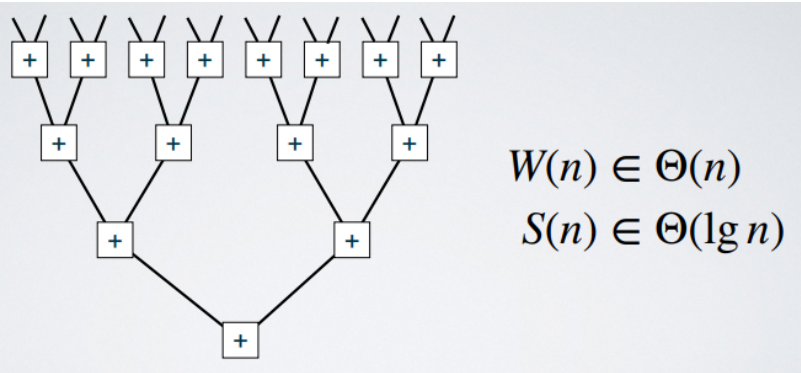
1.3.2. Definición. Profundidad (S).

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.

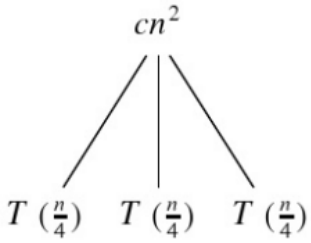
Cada rama de una llamada recursiva, a su vez, se expande en su propio árbol, y se procede así hasta llegar a un caso base, en el cual se deja de expandir el árbol. Por cada nivel del árbol se suma el total de operaciones, y estos a su vez se suman para dar el costo total de la recurrencia.

2.3.2. Ejemplo. Árbol de recurrencia.

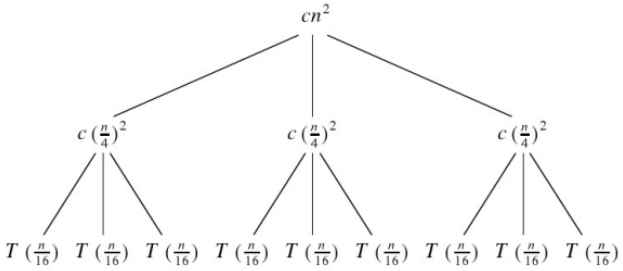
Consideremos la siguiente recurrencia:

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

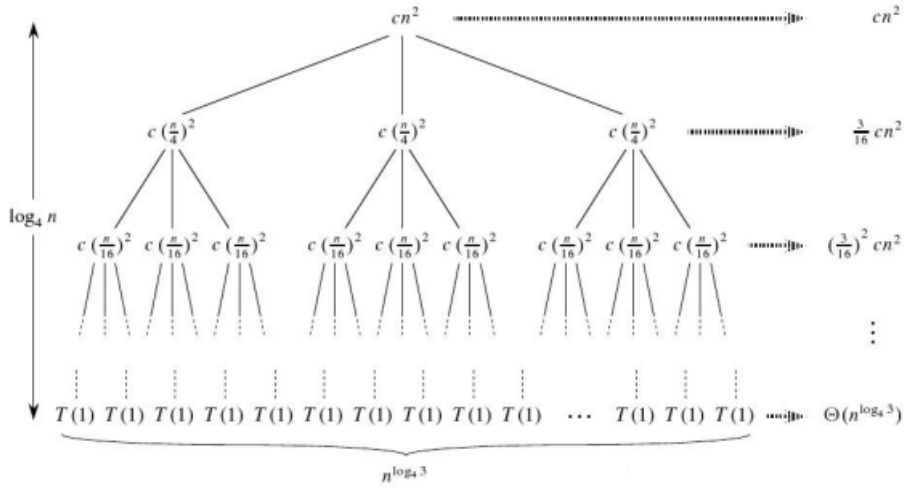
Hacemos un árbol que en su raíz tiene el costo para el n inicial, con ramas indicando cada una de las llamadas recursivas:



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 $(\frac{3}{16})^2cn^2$.

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:

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

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
```