Informe Taller 2 – Funciones de Alto Orden

> **Curso:** Fundamentos de Programación Funcional y Concurrente
> **Tema:** Funciones de alto orden

Estudiantes:

- Jorge Luis Junior Lasprilla Prada - 2420662
- Andrés Gerardo González Rosero- 2416541

---

1. Informe de funciones de alto orden

Función	Forma(s) de uso de funciones de alto orden	Descripción
`insert`	Parámetro (comparator)	Recibe `comp: (T,T) => Boolean` para decidir posicionamiento.
`insertionSort`	Devuelve una función (AlgoritmoOrd)	Generador de algoritmo: `insertionSort(comp)` devuelve `AlgoritmoOrd[T]`. Usa `insert`.
`menoresQueNoMenoresQue`	Parámetro (comparator)	Particiona una lista respecto a un `pivote`, usando `comp`.
`quickSort`	Devuelve una función (AlgoritmoOrd)	Generador de algoritmo: `quickSort(comp)` devuelve `AlgoritmoOrd[T]`. Usa `menoresQueNoMenoresQue`.
`comparar`	Parámetro (recibe dos AlgoritmoOrd)	Aplica los dos algoritmos como funciones de orden superior a la misma entrada.

---

2. Informe de corrección

Sea $comp : A \times A \to \{\text{true}, \text{false}\}$.
Definimos el orden inducido por el comparador como:

$$x \leq_{comp} y \iff comp(x, y) = \text{true}.$$

Decimos que una lista l' es una **permutación** de otra lista l, y escribimos $l' \sim l$, si l' contiene exactamente los mismos elementos que l.

Lema 1. Correctitud de `insert`

Enunciado:
Sea $e \in A$ y l una lista ordenada según $\leq_{comp}$. Entonces:

$$insert(e, l, comp) = (l', k)$$

donde l' está ordenada, $l' \sim (e :: l)$ y k corresponde al número exacto de comparaciones realizadas.

Demostración:

- **Caso base ($l = []$):**
  $insert(e, [], comp) = ([e], 0)$.
  $[e]$ está ordenada, $[e] \sim (e :: [])$ y $k = 0$.
- **Caso inductivo ($l = h :: t$):**
  Se evalúa $comp(e, h)$ (una comparación).
  - Si $e \leq_{comp} h$:

    $$insert(e, h :: t, comp) = (e :: h :: t, 1)$$

    que está ordenada y es permutación de $e :: h :: t$.
  - Si $e \not\leq_{comp} h$:
    Por hipótesis inductiva,

    $$insert(e, t, comp) = (t', c), \quad t' \sim (e :: t).$$

    Entonces,

    $$insert(e, h :: t, comp) = (h :: t', c + 1), \quad (h :: t') \sim (e :: h :: t).$$

En ambos casos se cumple la propiedad.
$Q.E.D.$

Teorema 1. Correctitud de `insertionSort`

Enunciado:
Para toda lista l:

$$insertionSort(comp)(l) = (l', k)$$

donde l' está ordenada según $\leq_{comp}$, $l' \sim l$, y k es el número total de comparaciones.

Demostración:

- **Caso base ($l = []$):**
  $([], 0)$, que está ordenada y es permutación trivial.
- **Caso inductivo ($l = h :: t$):**
  Por hipótesis inductiva:

  $$insertionSort(comp)(t) = (t', k_t), \quad t' \sim t.$$

Luego, por el Lema 1:

$$insert(h, t', comp) = (t'', k_h), \quad t'' \sim (h :: t').$$

Entonces $t'' \sim l$ y el número total de comparaciones es $k_t + k_h$.

Por inducción, la función es correcta.
$Q.E.D.$

Lema 2. Correctitud de `menoresQueNoMenoresQue`

Enunciado:
Para toda lista l y pivote v:

$$menoresQueNoMenoresQue(l, v, comp) = (l_1, l_2, k)$$

donde:

1. $l_1 = \{\, x \in l \mid x \leq_{comp} v \,\}$
2. $l_2 = \{\, x \in l \mid \neg(x \leq_{comp} v) \,\}$
3. $l_1 \cup l_2 \sim l$
4. $k = |l|$

Demostración:

- **Caso base ($l = []$):**
  $([], [], 0)$, que cumple todas las condiciones.
- **Caso inductivo ($l = h :: t$):**
  Se compara h con v:
  - Si $h \leq_{comp} v$, se añade h a l_1.
  - Si no, se añade a l_2.

Por hipótesis inductiva, la partición de t es correcta y se cuentan $|t|$ comparaciones.
Sumando la de h, se obtiene $k = |l|$.

La unión $l_1 \cup l_2 \sim l$, así que el lema queda demostrado.
$Q.E.D.$

Teorema 2. Correctitud de `quickSort`

Enunciado:
Para toda lista l:

$$quickSort(comp)(l) = (l', k)$$

donde l' está ordenada según $\leq_{comp}$, $l' \sim l$, y k es el número total de comparaciones.

Demostración:

- **Caso base:**
  Si $l = []$ o $|l| = 1$, entonces $(l, 0)$, que ya está ordenada y es permutación de sí misma.
- **Caso inductivo ($l = h :: t$):**
  Sea $menoresQueNoMenoresQue(t, h, comp) = (l_1, l_2, k_p)$.
  Por el Lema 2, $l_1 \cup l_2 \sim t$.
  Además, $\forall x \in l_1,\ x \leq_{comp} h$ y $\forall y \in l_2,\ \neg(y \leq_{comp} h)$.

Aplicando la hipótesis inductiva a $quickSort(l_1)$ y $quickSort(l_2)$ se obtienen l_1' y l_2' ordenadas y permutaciones de sus entradas.

La concatenación:

$$l' = l_1' + +(h :: l_2')$$

es ordenada y $l' \sim l$.

El número total de comparaciones es $k = k_p + k_{l_1} + k_{l_2}$.

Por inducción, la función es correcta.
$Q.E.D.$

Teorema 3. Correctitud de `comparar`

Enunciado:
Para toda lista l y algoritmos de ordenamiento a_1, a_2:

$$comparar(a_1, a_2, l) = \begin{cases} (k_1, k_2) & \text{si } a_1(l) = (l', k_1),\ a_2(l) = (l', k_2) \\ (-1, -1) & \text{si } a_1(l) = (l_1, k_1),\ a_2(l) = (l_2, k_2),\ l_1 \neq l_2 \end{cases}$$

Demostración:

Sea $a_1(l) = (l_1, k_1)$ y $a_2(l) = (l_2, k_2)$.

- Si $l_1 = l_2$, entonces ambas funciones producen la misma lista ordenada.
  Por correctitud de a_1 y a_2, $l_1 \sim l$ y $l_2 \sim l$.
  Por unicidad del ordenamiento, $l_1 = l_2$. Entonces, $comparar$ devuelve (k_1, k_2).
- Si $l_1 \neq l_2$, la función devuelve $(-1, -1)$, cumpliendo la especificación.

Así, la función es correcta.
$Q.E.D.$

---

3. Casos de prueba

3.1 Archivo de pruebas

> Este es nuestro archivo de pruebas `pruebas.sc`, esta basado en el que el profesor proporcionó en el taller y usamos una metodologia similar para probar las funciones

> del paquete `Comparador`, tambien se puede encontrar por supuesto en el `.zip` de la entrega:

```
// pruebas.sc - Worksheet de pruebas para el paquete Comparador
import Comparador._
import scala.util.Random

// Crea un número aleatorio
val random = new Random()

// Crea una lista de enteros al azar
def listaAlAzar(long: Int): List[Int] = {
  // Crea una lista de `long` enteros con valores aleatorios entre 1 y
long*2
  val v = Vector.fill(long) {
    random.nextInt(long * 2) + 1
  }
  v.toList
}

// Compara dos numeros
def menorQue(a: Int, b: Int): Boolean = a < b
def mayorQue(a: Int, b: Int): Boolean = a > b

// Aplica a insertionSort las funciones de comparación
val iSortAsc  = insertionSort[Int](menorQue)
val iSortDesc = insertionSort[Int](mayorQue)
iSortAsc(List(4,5,6,1,2,3))
iSortDesc(List(4,5,6,1,2,3))

// Aplica a quickSort las funciones de comparación
val qSortAsc  = quickSort[Int](menorQue)
val qSortDesc = quickSort[Int](mayorQue)
qSortAsc(List(4,5,6,1,2,3))
qSortDesc(List(4,5,6,1,2,3))

// Compara los resultados de insertionSort y quickSort
comparar(iSortAsc, qSortAsc, List(4,5,6,1,2,3))
comparar(iSortDesc, qSortDesc, List(4,5,6,1,2,3))
comparar(iSortDesc, qSortAsc, List(4,5,6,1,2,3)) // Debe fallar y retornar
(-1, -1)
comparar(iSortAsc, qSortDesc, List(4,5,6,1,2,3)) // Debe fallar y retornar
(-1, -1)

// Listas ordenadas y reversas
val lAsc100   = (1 to 100).toList
val lAsc1000  = (1 to 1000).toList
val lDsc100   = (1 to 100).toList.reverse
val lDsc1000  = (1 to 1000).toList.reverse
```

```
// Pruebas con listas ordenadas y reversas
comparar(iSortAsc, qSortAsc, lAsc100)
comparar(iSortAsc, qSortAsc, lAsc1000)
comparar(iSortAsc, qSortAsc, lDsc100)
comparar(iSortAsc, qSortAsc, lDsc1000)

// Crea listas al azar de diferentes tamaños
val l5  = listaAlAzar(5)
val l10 = listaAlAzar(10)
val l20 = listaAlAzar(20)
val l50 = listaAlAzar(50)

// Pruebas de insertionSort con listas al azar
iSortAsc(l5)
iSortDesc(l5)
iSortAsc(l10)
iSortDesc(l10)
iSortAsc(l20)
iSortDesc(l20)
iSortAsc(l50)
iSortDesc(l50)

// Pruebas de quickSort con listas al azar
qSortAsc(l5)
qSortDesc(l5)
qSortAsc(l10)
qSortDesc(l10)
qSortAsc(l20)
qSortDesc(l20)
qSortAsc(l50)
qSortDesc(l50)

// Compara los resultados de insertionSort y quickSort ascendentes con
listas al azar
comparar(iSortAsc, qSortAsc, l5)
comparar(iSortAsc, qSortAsc, l10)
comparar(iSortAsc, qSortAsc, l20)
comparar(iSortAsc, qSortAsc, l50)

// Compara los resultados de insertionSort y quickSort descendentes con
listas al azar

comparar(iSortDesc, qSortDesc, l5)
comparar(iSortDesc, qSortDesc, l10)
comparar(iSortDesc, qSortDesc, l20)
comparar(iSortDesc, qSortDesc, l50)
```

3.2. Casos básicos con listas pequeñas

3.2.1 InsertionSort y QuickSort con lista [4,5,6,1,2,3]

Algoritmo	Entrada	Resultado esperado	Resultado observado
iSortAsc	List(4,5,6,1,2,3)	(List(1,2,3,4,5,6), 13)	(List(1, 2, 3, 4, 5, 6),9)
iSortDesc	List(4,5,6,1,2,3)	(List(6,5,4,3,2,1), 13)	(List(6, 5, 4, 3, 2, 1),13)
qSortAsc	List(4,5,6,1,2,3)	(List(1,2,3,4,5,6), 16)	(List(1, 2, 3, 4, 5, 6),9)
qSortDesc	List(4,5,6,1,2,3)	(List(6,5,4,3,2,1), 16)	(List(6, 5, 4, 3, 2, 1),9)

3.2.2 Comparar con lista [4,5,6,1,2,3]

Llamada	Resultado esperado	Resultado observado
comparar(iSortAsc, qSortAsc, List(4,5,6,1,2,3))	(13,16)	(9,9)
comparar(iSortAsc, qSortDesc, List(4,5,6,1,2,3))	(-1,-1)	(-1,-1)

3.3. Casos con listas grandes ordenadas y reversas

Lista	Descripción	Resultado esperado (iSortAsc vs qSortAsc)	Observado
lAsc100	(1 to 100)	(4950, 4950)	(4950,4950)
lAsc1000	(1 to 1000)	(499500, 499500)	(499500,499500)
lDsc100	(100 to 1)	(99, 4950)	(99,4950)
lDsc1000	(1000 to 1)	(999, 499500)	(999,499500)

3.4. Casos con listas aleatorias

> Aquí los resultados esperados **no son fijos** porque las listas se generan al azar.

3.4.1 Listas de 5 elementos (`l5`)

Algoritmo	Resultado esperado	Observado
iSortAsc(l5)	Lista ordenada ascendente con comparaciones	(List(2, 5, 6, 6, 8),10)
iSortDesc(l5)	Lista ordenada descendente con comparaciones	(List(8, 6, 6, 5, 2),8)

Algoritmo	Resultado esperado	Observado
qSortAsc(l5)	Lista ordenada ascendente con comparaciones	(List(1, 6, 7, 8, 10),7)
qSortDesc(l5)	Lista ordenada descendente con comparaciones	(List(10, 8, 7, 6, 1),7)
comparar(iSortAsc, qSortAsc, l5)	(c1,c2) si iguales, (-1,-1) si no	(7,7)

3.4.2 Listas de 10, 20 y 50 elementos (`l10`, `l20`, `l50`)

Lista	Resultado esperado	Observado
iSortAsc(l10), iSortDesc(l10), qSortAsc(l10), qSortDesc(l10)	Listas ordenadas según comparador + comparaciones	(List(2, 4, 4, 5, 7, 7, 7, 11, 13, 18),39), (List(18, 13, 11, 7, 7, 7, 5, 4, 4, 2),23), (List(2, 4, 4, 5, 7, 7, 7, 11, 13, 18),28), (List(18, 13, 11, 7, 7, 7, 5, 4, 4, 2),24)
comparar(iSortAsc, qSortAsc, l10)	(c1,c2)	(39,28)
iSortAsc(l20), iSortDesc(l20), qSortAsc(l20), qSortDesc(l20)	idem	Por cuestiones de practicidad y en vista de que agregar aquí la salida de cada función es contraproducente y anti-intuitivo, se insta al lector a correr cada prueba y función en el archivo de pruebas, después de todo estas corren con listas aleatorias.
comparar(iSortAsc, qSortAsc, l20)	(c1,c2)	(92,62)
iSortAsc(l50), iSortDesc(l50), qSortAsc(l50), qSortDesc(l50)	idem	Por cuestiones de practicidad y en vista de que agregar aquí la salida de cada función es contraproducente y anti-intuitivo, se insta al lector a correr cada prueba y función en el archivo de pruebas, después de todo estas corren con listas aleatorias.
comparar(iSortAsc, qSortAsc, l50)	(c1,c2)	(716,230)

3.5 Conclusiones de las pruebas

> Estas y mas pruebas se pueden encontrar en el archivo `Pruebas.sc` que se encuentra en el `.zip` de la entrega, ahi se hallan comentadas y explicadas (los resultados pueden variar por el uso de listas aleatorias).

Nos encontramos con que:

- Ambas implementaciones de ordenamiento son correctas, pues devuelven listas ordenadas según el comparador.
- `insertionSort` es más eficiente en listas casi ordenadas (ascendentes), mientras que `quickSort` es más eficiente en listas desordenadas o inversamente ordenadas.
- El número de comparaciones realizadas por cada algoritmo varía significativamente según la estructura inicial de la lista, lo que afecta su rendimiento.
- En listas pequeñas (5-10 elementos), la diferencia en el número de comparaciones es menos pronunciada, pero `quickSort` tiende a ser más eficiente a medida que la lista crece.
- La función `comparar` es útil para validar que ambos algoritmos producen el mismo resultado, aunque el número de comparaciones difiere.

---

4. Conclusiones

Las funciones de alto orden se consolidan como un recurso fundamental dentro del paradigma funcional, ya que permiten diseñar algoritmos más flexibles y reutilizables. En este taller, `insertionSort` y `quickSort` ejemplifican cómo un mismo esquema de ordenamiento puede adaptarse fácilmente a distintos criterios, simplemente parametrizando el comparador de elementos.

La separación entre la lógica del algoritmo y la definición del criterio de comparación resalta una de las principales ventajas de las funciones de alto orden: la modularidad. Este principio no solo mejora la claridad del código, sino que también fomenta la reutilización en diversos contextos, sin necesidad de reescribir la lógica central del algoritmo.

En términos de corrección y verificación, las funciones de alto orden ofrecen un marco más claro para el razonamiento formal. La argumentación inductiva utilizada para justificar cada función muestra que este paradigma facilita tanto la prueba matemática de corrección como el diseño de casos de prueba coherentes y consistentes.

La experimentación con distintos tamaños y distribuciones de listas permitió evidenciar cómo la elección del algoritmo de ordenamiento impacta en el rendimiento. Mientras que `insertionSort` resulta eficiente en listas pequeñas o parcialmente ordenadas, `quickSort` muestra un mejor comportamiento en entradas más grandes y variadas. Las funciones de alto orden no solo posibilitaron implementar estos algoritmos, sino también compararlos objetivamente en términos de número de comparaciones.

Finalmente, este taller demuestra que las funciones de alto orden trascienden el ámbito del ordenamiento. Su aplicabilidad se extiende a múltiples áreas de la informática. Una

vez comprendidas, su construcción resulta cómoda, su uso es intuitivo y su aporte a la modularidad y reutilización del código es sustancialmente superior al de otros enfoques.