$is\ (h :: tl) = insert\ h\ (is\ tl)$

$$is\ l := \begin{cases} l, & \text{se } l = nil \\ insert\ h\ (is\ tl), & \text{se } l = h :: tl \end{cases}$$

onde

$$insert\ x\ l := \begin{cases} x :: nil, & \text{se } l = nil \\ x :: l, & \text{se } x \leq h \text{ e } l = h :: tl \\ h :: (insert\ x\ tl), & \text{se } x > h \text{ e } l = h :: tl \end{cases}$$

Qual é o número de comparações realizadas pelo algoritmo de ordenação por inserção, isto é, pela função $is$, para ordenar uma lista $l$? Vamos denotar por $T_{is}()$ a função que faz esta contagem. Se $l$ for a lista vazia então nenhuma comparação é feita, ou seja, $T_{is}(nil) = 0$. Se $l = h :: tl$ então é feita uma chamada à função $ins$, além da chamada recursiva à função $is$:

$$T_{is}(l) = \begin{cases} 0, & \text{se } l = nil \\ T_{is}(tl) + T_{ins}\ h\ (is\ tl), & \text{se } l = h :: tl \end{cases}$$

Observe que, $T_{is}(1 :: 2 :: 3 :: nil) = 2$, $T_{is}(3 :: 2 :: 1 :: nil) = 3$, $T_{is}(1 :: 2 :: 3 :: 4 :: nil) = 3$ e $T_{is}(4 :: 3 :: 2 :: 1 :: nil) = 6$, etc. Portanto o número de comparações pode ser diferente para listas de mesmo tamanho, o que é esperado pelas chamadas feitas à função $ins$. Como então definir a função $T_{is}^{w}(n)$ que nos dá um limite superior para o número de comparações feitas pelo algoritmo de ordenação por inserção para uma lista qualquer de tamanho $n$? Em outras palavras, qual a complexidade do pior caso para o algoritmo de ordenação por inserção? Sabemos que quando $n = 0$, nenhuma comparação é feita. Quando $n > 0$, o algoritmo é aplicado recursivamente na cauda da lista, isto é, em uma lista de tamanho $n - 1$, e é feita uma chamada à função $ins$ cuja complexidade já conhecemos. Isto nos permite escrever a função $T_{is}^{w}(n)$ como a seguir:

$$T_{is}^{w}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T_{is}^{w}(n-1) + T_{ins}^{w}(n-1), & \text{se } n > 0 \end{cases}$$ que pode ser simplificada como a seguir, já que $T_{ins}^{w}(n) = n$:

$$T_{is}^{w}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T_{is}^{w}(n-1) + (n-1), & \text{se } n > 0 \end{cases}$$

Podemos usar o método da substituição para encontrarmos uma solução para esta recorrência, e em seguida utilizar indução para verificarmos se a solução está correta. Pelo método da substituição, podemos ir aplicando a definição da recorrência, assumindo que $n > 0$:

$$\begin{aligned} T_{is}^{w}(n) &= T_{is}^{w}(n-1) + (n-1) \\ &= T_{is}^{w}(n-2) + (n-2) + (n-1) \\ &= T_{is}^{w}(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots \\ &= T_{is}^{w}(n-n) + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= 0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \end{aligned}$$

Para finalizar, precisamos utilizar indução em $n$ para provar que $T_{is}^{w}(n) = \frac{n(n-1)}{2}$. Se $n = 0$, o resultado é trivial. Se $n > 0$ então, por definição, $T_{is}^{w}(n) = T_{is}^{w}(n-1) + (n-1)$. A hipótese de indução, nos dá que $T_{is}^{w}(n-1) = \frac{(n-1)(n-2)}{2}$, e portanto, $T_{is}^{w}(n) = T_{is}^{w}(n-1) + (n-1) \stackrel{h.i.}{=} \frac{(n-1)(n-2)}{2} + (n-1) = \frac{n(n-1)}{2}$.

Nossa conclusão, portanto, é que o algoritmo de ordenação por inserção recursivo é correto, e sua complexidade no pior caso é quadrática, assim como na versão não-recursiva.

**Exercício 3.28.** *Resolva as seguintes relações de recorrência:*

1. $T(1) = 1$, $T(n) = 2T(n-1) + 1$, $n \geq 2$
2. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + 1/n$
3. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + ln(n)$

## 3.6 Mergesort

Algoritmos recursivos desempenham um papel fundamental em Computação. O algoritmo de ordenação *mergesort* é um exemplo de algoritmo recursivo, que se caracteriza por dividir o problema original em subproblemas que, por sua vez, são resolvidos recursivamente. As soluções dos subproblema são então combinadas para gerar uma solução para o problema original. Este paradigma de projeto de algoritmo é conhecido com *divisão e conquista*. Este algoritmo foi inventado por J. von Newmann em 1945.

O algoritmo *mergesort* é um algoritmo de ordenação que utiliza a técnica de divisão e conquista, que consiste das seguintes etapas:

1. **Divisão**: O algoritmo divide a lista (ou vetor) $l$ recebida como argumento ao meio, obtendo as listas $l_1$ e $l_2$;

2. **Conquista**: O algoritmo é aplicado recursivamente às listas $l_1$ e $l_2$ gerando, respectivamente, as listas ordenadas $l'_1$ e $l'_2$;

3. **Combinação**: O algoritmo combina as listas $l'_1$ e $l'_2$ através da função *merge* que então gera a saída do algoritmo.

Por exemplo, ao receber a lista $(4 :: 2 :: 1 :: 3 :: nil)$, este algoritmo inicialmente divide esta lista em duas sublistas, a saber $(4 :: 2 :: nil)$ e $(1 :: 3 :: nil)$. O algoritmo é aplicado recursivamente às duas sublistas para ordená-las, e ao final deste processo, teremos duas listas ordenadas $(2 :: 4 :: nil)$ e $(1 :: 3 :: nil)$. Estas listas são, então, combinadas para gerar a lista de saída $(1 :: 2 :: 3 :: 4 :: nil)$.

**Algorithm 7:** mergesort$(A, p, r)$

**if** $p < r$ **then**
  $q = \lfloor \frac{p+r}{2} \rfloor$;
  mergesort$(A, p, q)$;
  mergesort$(A, q+1, r)$;
  merge$(A, p, q, r)$;
**end**

A etapa de combinar dois vetores ordenados (algoritmo *merge*) é a etapa principal do algoritmo *mergesort*. O procedimento *merge*$(A, p, q, r)$ descrito a seguir recebe como argumentos o vetor $A$, e os índices $p, q$ e $r$ tais que $p \leq q < r$. O procedimento assume que os subvetores $A[p..q]$ e $A[q+1..r]$ estão ordenados.

**Algorithm 8:** merge$(A, p, q, r)$

$n_1 = q - p + 1$ ; `// Qtd. de elementos em` $A[p..q]$
$n_2 = r - q$ ; `// Qtd. de elementos em` $A[q+1..r]$
let $L[1..n_1+1]$ and $R[1..n_2+1]$ be new arrays;
**for** $i = 1$ *to* $n_1$ **do**
  $L[i] = A[p+i-1]$;
**end**
**for** $j = 1$ *to* $n_2$ **do**
  $R[j] = A[q+j]$;
**end**
$L[n_1+1] = \infty$;
$R[n_2+1] = \infty$;
$i = 1$;
$j = 1$;
**for** $k = p$ *to* $r$ **do**
  **if** $L[i] \leq R[j]$ **then**
    $A[k] = L[i]$;
    $i = i + 1$;
  **end**
  **else**
    $A[k] = R[j]$;
    $j = j + 1$;
  **end**
**end**

**Exercício 3.29.** *Prove que o algoritmo merge é correto.*

**Exercício 3.30.** *Prove que o algoritmo mergesort é correto.*

**Exercício 3.31.** *Faça a análise assintótica do algoritmo merge.*

**Exercício 3.32.** *Faça a análise assintótica do algoritmo mergesort.*

## 3.7 O Teorema Mestre

Nesta seção estudaremos as equações de recorrência utilizadas no paradigma de divisão de conquista [4]:

**Definição 3.33.** *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* eventualmente não-decrescente *se existir um número inteiro $n_0$ tal que $f(n)$ é não-decrescente no intervalo $[n_0, \infty)$, ou seja,*

$$f(n_1) \leq f(n_2), \forall n_2 > n_1 \geq n_0.$$

**Definição 3.34.** *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* suave *se for eventualmente não-decrescente e*

$$f(2.n) = \Theta(f(n))$$

**Exercício 3.35.** *Mostre que $f(n) = n$ é suave.*

**Exercício 3.36.** *Mostre que $f(n) = \lg n$ é suave.*

**Exercício 3.37.** *Mostre que $f(n) = n. \lg n$ é suave.*

**Exercício 3.38.** *Mostre que $f(n) = 3^n$ não é suave.*

**Lema 3.39.** *Sejam $c$ e $n_0$ constantes positivas, e $f(n)$ uma função tal que $f(2n) \leq c.f(n), \forall n \geq n_0$. Então $f(2^k n) \leq c^k.f(n), \forall n \geq n_0$ e $k \geq 1$.*

**Teorema 3.40.** *Seja $f(n)$ uma função suave. Então para qualquer $b \geq 2$ fixado,*

$$f(b.n) = \Theta(f(n))$$

O teorema a seguir é conhecido como *regra da suavização*

**Teorema 3.41.** *Seja $T(n)$ uma função eventualmente não-decrescente, e $f(n)$ uma função suave. Se $T(n) = \Theta(f(n))$ para valores de $n$ que são potências de $b$ ($b \geq 2$), então*

$$T(n) = \Theta(f(n)), \forall n.$$

A regra da suavização nos permite expandir a informação sobre a ordem de crescimento estabelecida para $T(n)$ de um subconjunto de valores (potências de $b$) para o domínio inteiro. O teorema a seguir é um resultado muito útil nesta direção conhecido como *teorema mestre*:

**Teorema 3.42.** *Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência*
$T(n) = a.T(n/b) + f(n), \quad$ *para* $n = b^k, k = 1, 2, 3, \ldots$
$T(1) = c$
*onde $a \geq 1, b \geq 2$ e $c \geq 0$ . Se $f(n) = \Theta(n^d)$, onde $d \geq 0$, então*

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & se\ a > b^d \\ \Theta(n^d. \lg n), & se\ a = b^d \\ \Theta(n^d), & se\ a < b^d \end{cases}$$

*Demonstração.* Considere que $f(n) = n^d$. Aplicando o método da substituição para a recorrência do teorema, obtemos:

$$T(b^k) = a^k.[T(1) + \sum_{j=1}^{k} f(b^j)/a^j]$$