Práctico 1.3

Definiciones y teoremas:

⊏ y ≈:

$$f(n) \sqsubset g(n) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \sqsupset g(n) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

$$f(n) \approx g(n) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} \in (0, \infty)$$

Propiedades:

`a*g(n) ≈ g(n)`

`f(n) ⊏ g(n) ⇒ f(n) + g(n) ≈ g(n)`

Si `a, b > 1`:

$\quad$ $\log_a(n) \approx \log_b(n)$

Si `lim n → ∞ : f(n) > 0`:

$\quad$ `g(n) ⊏ h(n) ⇔ f(n)g(n) ⊏ f(n)h(n)`

$\quad$ `g(n) ≈ h(n) ⇔ f(n)g(n) ≈ f(n)h(n)`

Si `lim n → ∞ : h(n) = ∞`:

$\quad$ `f(n) ⊏ g(n) ⇒ f(h(n)) ⊏ g(h(n))`

$\quad$ `f(n) ≈ g(n) ⇒ f(h(n)) ≈ g(h(n))`

Orden de convergencia de algoritmos recursivos: 🤔

Sean:

g de orden n^k

```
t(n) = (□ caso_base(n) → c
        □ si no        → a * t(n/b) + g(n)
       )
```

```
     t(n) ≈ (□ a > b^k → n^(log_b(a))
             □ a = b^k → n^k * log(n)
             □ si no   → n^k
            )
```

Demostración:

((≈) es la igualdad de orden)

Caso $a > b^k$:

$t(n)$

≈

$a * t(n/b) + g(n)$

≈ {Estoy suponiendo $g(n) \approx n^k$}

$a * t(n/b) + n^k$

≈ {Hipótesis inductiva}

$a * n^{(\log_b(a))} + n^k$

≈ {$a > b^k \Rightarrow \log_b(a) > k \Rightarrow n^{(\log_b(a))} > n^k \Rightarrow n^k \in \mathcal{O}(n^{(\log_b(a))})$, osea, n^k es depreciable}

$a * n^{(\log_b(a))}$

≈ {*a es despreciable}

$n^{(\log_b(a))}$

Caso $a = b^k$:

$t(n)$

≈

$a * t(n/b) + g(n)$

≈ {Estoy suponiendo $g(n) \approx n^k$}

$a * t(n/b) + n^k$

≈ {Hipótesis inductiva}

$a * n^k * \log(n) + n^k$

≈ {$+ n^k$ es despreciable ya que $n^k \in \mathcal{O}(n^k * \log(n))$}

$a * n^k * \log(n)$

≈ {*a es despreciable}

$n^k * \log(n)$

Caso $a < b^k$:

$t(n)$

≈

$a * t(n/b) + g(n)$

≈ {Estoy suponiendo $g(n) \approx n^k$}

$a * t(n/b) + n^k$

≈ {Hipótesis inductiva}

$a * (n/b)^k + n^k$

≈

$a * n^k / b^k + n^k$

≈
```
      n^k * (a/b^k + 1)
```
≈ {Estoy en el caso a `<` `b^k` `⇒` `1 < a/b^k + 1 <` 2, osea, `a/b^k + 1` es una constante que no afecta al orden}
```
      n^k
```

Ejercicios:

1)

1. Calculá el orden de complejidad de los siguientes algoritmos:

(a) **proc** $f1$(**in** n : **nat**)
  if $n \leq 1$ **then skip**
  else
    for $i := 1$ **to** 8 **do** $f1(n$ **div** $2)$ **od**
    for $i := 1$ **to** n^3 **do** $t := 1$ **od**

(b) **proc** $f2$(**in** n : **nat**)
  for $i := 1$ **to** n **do**
    for $j := 1$ **to** i **do** $t := 1$ **od**
  od
  if $n > 0$ **then**
    for $i := 1$ **to** 4 **do** $f2(n$ **div** $2)$ **od**

1a)

```
proc f1(in n : nat)
      if n ≤ 1 →
            skip
      else
            for i ≔ 1 to 8 do
                  f1(n `div` 2)
            od
            for i ≔ 1 to n³ do
                  t ≔ 1
            od
      fi
end proc
```

Operación: asignaciones a `t`

Sea:
`r(n) = ops(f1(n))`

```
      r(n)
=
      ops(proc f1(in n : nat)
            if n ≤ 1 →
                  skip
            else
                  for i ≔ 1 to 8 do
                        f1(n `div` 2)
                  od
```

```
                    for i ≔ 1 to n³ do
                         t ≔ 1
                    od
               fi
          end proc)
=
     (□ n ≤ 1 → 0
      □ si no → ops(for i ≔ 1 to 8 do f1(n `div` 2) od)
               + ops(for i ≔ 1 to n³ do t ≔ 1 od)
     )
=
     (□ n ≤ 1 → 0
      □ si no → ⟨∑i = 1 to 8 : ops(f1(n `div` 2))⟩
               + ⟨∑i = 1 to n³ : 1⟩
     )
=
     (□ n ≤ 1 → 0
      □ si no → 8 * ops(f1(n `div` 2)) + n³
     )
=
     (□ n ≤ 1 → 0
      □ si no → 8 * r(n `div` 2) + n³
     )
```
≈ {Del mismo orden, teorema}

```
     n³ log(n)
```

1b)

```
proc f2(in n : nat)
     for i ≔ 1 to n do
          for j ≔ 1 to i do
               t ≔ 1
          od
     od
     if n > 0 →
          for i ≔ 1 to 4 do
               f2(n `div` 2)
          od
     fi
end proc
```

Sea:

```
r(n) = ops(f2(n))

     r(n)
=
     ops(for i ≔ 1 to n do
               for j ≔ 1 to i do
```

```
                    t ≔ 1
               od
          od
          if n > 0 →
               for i ≔ 1 to 4 do
                    f2(n `div` 2)
               od
          fi)
=
     〈∑i = 1 to n : 〈∑j = 1 to i : 1〉〉
     + ( □ n > 0 → 〈∑i = 1 to 4 : ops(f2(n `div` 2))〉
         □ si no → 0
       )
=
     〈∑i = 1 to n : i〉
     + ( □ n > 0 → 4 * r(n `div` 2)
         □ si no → 0
       )
=
     n*(n+1)/2 + ( □ n > 0 → 4 * r(n `div` 2)
                   □ si no → 0
                 )
=
     ( □ n > 0 → 4 * r(n `div` 2) + n²/2 + n/2
       □ si no → n²/2 + n/2
     )
=
     ( □ n > 0 → 4 * r(n `div` 2) + n²/2 + n/2
       □ si no → 0
     )
≃ {Del mismo orden, teorema}
     n² * log(n)
```

2)

2. Dado un arreglo a : **array**$[1..n]$ **of nat** se define una *cima* de a como un valor k en el intervalo $1, \ldots, n$ tal que $a[1..k]$ está ordenado crecientemente y $a[k..n]$ está ordenado decrecientemente.

   (a) Escribí un algoritmo que determine si un arreglo dado tiene cima.

   (b) Escribí un algoritmo que encuentre la cima de un arreglo dado (asumiendo que efectivamente tiene una cima) utilizando una búsqueda secuencial, desde el comienzo del arreglo hacia el final.

   (c) Escribí un algoritmo que resuelva el mismo problema del inciso anterior utilizando la idea de *búsqueda binaria*.

   (d) Calculá y compará el orden de complejidad de ambos algoritmos.

2a)

```
fun tieneCima(a : array[1..n] of nat) ret res : bool
      var j : nat
      j ≔ 1
      while j < n ∧ a[j] < a[j + 1] do
            j ≔ j + 1
      od
      res ≔ estaOrdenadoEntre(a, j, n, (>))
end fun

fun estaOrdenadoEntre(a : array[1..n] of nat, i, j : nat, orden : nat → nat → bool)
ret res : bool
      res ≔ True
      var k : nat
      k ≔ i
      while k < j ∧ res do
            res ≔ a[j] `orden` a[j+1]
            k ≔ k + 1
      od
end fun
```

Otra forma:

```
fun tieneCima(a : array[1..n] of nat) ret res : bool
      var j : nat
      res ≔ True
      j ≔ 1
      while j < n ∧ a[j] < a[j + 1] do
            j ≔ j + 1
      od
      while j < n ∧ res do
            res ≔ a[j] > a[j+1]
            j ≔ j + 1
      od
end fun
```

2b)

Suponiendo que a tiene cima:

```
fun posCima1(a : array[1..n] of nat) ret cima : nat
    cima ≔ 1
    while cima < n ∧ a[cima] < a[cima + 1] do
        cima ≔ cima + 1
    od
end fun
```

2c)

Suponiendo que a tiene cima:

```
fun posCima2(a : array[1..n] of nat) ret cima : nat
    cima = posCimaEntre(a, 1, n)
end fun

fun posCimaEntre(a : array[1..n] of nat, izq, der : nat) ret cima : nat
    var med : nat
    med ≔ (izq + der) `div` 2
    if izq > der →
        if estaOrdenadoEntre(a, izq, med, (<)) →
            cima ≔ posCimaEntre(a, med, der)
        else
            cima ≔ posCimaEntre(a, izq, med)
        fi
    else
        cima = med
    fi
end fun
```

2cv2)

Suponiendo que a tiene cima:

```
fun posCima2(a : array[1..n] of nat) ret cima : nat
    cima = posCimaEntre(a, 1, n)
end fun

fun posCimaEntre(a : array[1..n] of nat, izq, der : nat) ret cima : nat
    var med : nat
    med ≔ (izq + der) `div` 2
    if izq > der →
        if a[med] < a[med+1]→
```

```
                cima ≔ posCimaEntre(a, izq, med)
          else
                cima ≔ posCimaEntre(a, med + 1, der)
          fi
     else
          cima = med
     fi
end fun
```

2d)

Operación a contar: comparaciones (<)

```
     ops(posCima1)
```
= {El peor caso es cuando `a[cima] ≤ a[cima + 1]` nunca es `False`}

```
     n
```

⇒

Es de complejidad lineal

Para `posCima2`:

Sea:

```
t(izq + der) = ops(posCimaEntre)

     t(izq + der)
=
     ops(var med : nat
          med ≔ (izq + der) `div` 2
          if izq > der →
                if estaOrdenadoEntre(a, izq, med, (≤)) →
                      cima ≔ posCimaEntre(a, med, der)
                else
                      cima ≔ posCimaEntre(a, izq, med)
                fi
          else
                cima = med
          fi
          )
=
     (□ izq ≤ der → 0
      □ si no → ops(if estaOrdenadoEntre(a, izq, med, (≤)) →
                           cima ≔ posCimaEntre(a, med, der)
                      else
                           cima ≔ posCimaEntre(a, izq, med)
                      fi)
     )
```

= {En ambos casos del `if` el tamaño de los parámetros es el mismo}

```
    (□ izq ≤ der → 0
     □ si no → ops(estaOrdenadoEntre(a, izq, med, (≤)))
               + ops(posCimaEntre(a, izq, med))
    )
```

= {En el peor caso `estaOrdenadoEntre` hace `izq + der - 1` operaciones}

```
    (□ izq ≤ der → 0
     □ si no → izq + med - 1 + t((izq + der)/2)
    )
```

=

```
    (□ izq ≤ der → 0
     □ si no → 1 * t((izq + der)/2) + izq + med - 1
    )
```

≈ {Teorema}

$(izq + der)^0 * \log(izq + der)$

=

```
    log(izq + der)
```

⇒

Es de complejidad logarítmica

3)

3. El siguiente algoritmo calcula el mínimo elemento de un arreglo a : **array**$[1..n]$ **of nat** mediante la técnica de programación *divide y vencerás*. Analizá la eficiencia de $minimo(1, n)$.

fun $minimo(a$: **array**$[1..n]$ **of nat**$, i, k$: **nat**$)$ **ret** m : **nat**
  if $i = k$ **then** $m := a[i]$
  else
    $j := (i + k)$ **div** 2
    $m := min(minimo(a, i, j), minimo(a,j+1,k))$
  fi
end fun

```
fun mínimo(a : array[1..n] of nat, i, k : nat) ret m : nat
    if i = k →
        m ≔ a[i]
    else
        j ≔ (i + k) `div` 2
        m ≔ min(mínimo(a, i, j), mínimo(a, j+1, k))
    fi
end fun
```

Sea

```
r(k - i) = ops(mínimo(a, i, k))
```

Se cuentan la cantidad de llamadas a `min`

```
      r(k - i)
=
      ops(if i = k →
                  m ≔ a[i]
            else
                  j ≔ (i + k) `div` 2
                  m ≔ min(mínimo(a, i, j), mínimo(a, j+1, k))
            fi
          )
=
      (□ i = k → 0
       □ si no → 1 + ops(mínimo(a, i, (i + k) `div` 2)) + ops(mínimo(a, ((i + k)
`div` 2) + 1, k))
      )
=
      (□ i = k → 0
       □ si no → 1 + r(((i + k) `div` 2) - i) + r(k - ((i + k) `div` 2) + 1)
      )
≃ {Del mismo orden}
      (□ i = k → 0
       □ si no → 2 * r(((i + k) `div` 2) - i) + 1
      )
≃ {Del mismo orden, teorema}
      (k-i)^(log₂(2))
=
      (k-i)^1
=
      k-i
```

⇒

La complejidad es lineal

4)

4. Ordená utilizando $\sqsubset$ e $\approx$ los órdenes de las siguientes funciones. No calcules límites, utilizá las propiedades algebraicas.

(a) $n \log 2^n$ $\quad 2^n \log n$ $\quad n! \log n$ $\quad 2^n$

(b) $n^4 + 2\log n$ $\quad \log(n^{n^4})$ $\quad 2^{4\log n}$ $\quad 4^n$ $\quad n^3 \log n$

(c) $\log n!$ $\quad n \log n$ $\quad \log(n^n)$

4a) $n * \log(2^n) \quad - \quad 2^n * \log n \quad - \quad n! * \log n \quad - \quad 2^n$

$n * \log(2^n) = n^2 * \log(2) \approx n^2$

$n * \log(2^n) \sqsubset 2^n \sqsubset 2^n * \log n \sqsubset n! * \log n$

4b) $n^4 + 2 * \log n \quad - \quad \log(n^{(n^4)}) \quad - \quad 2^{(4 * \log n)} \quad - \quad 4^n \quad - \quad n^3 * \log n$

$n^4 + 2 * \log n \approx n^4$

$\log(n^{(n^4)}) = n^4 * \log(n)$

$2^{(4 * \log n)} = (2^{(\log n)})^4 = n^4$

$n^3 * \log n \sqsubset 2^{(4 * \log n)} \approx n^4 + 2 * \log n \sqsubset \log(n^{(n^4)}) \sqsubset 4^n$

4c)

$\log n! \quad - \quad n * \log n \quad - \quad \log(n^n)$

$\log(n^n) = n * \log n$

Ahora comparo $\log n!$ con $\log(n^n)$:

En primer lugar pruebo $\log n! \sqsubseteq \log(n^n)$:

$\log n! \sqsubseteq \log(n^n)$

$\Leftarrow$

$n^n \geq n!$

$\Leftrightarrow$

True

Después pruebo $\log(n^n) \sqsubseteq \log n!$:

$\log(n^n) \sqsubseteq \log n!$

$\Leftrightarrow$

$\log(n^n) \sqsubseteq 2 * \log n!$

$\Leftrightarrow$

$\log(n^n) \sqsubseteq \log n!^2$

$\Leftarrow$

$\log(n^n) \leq \log n!^2$

$\Leftrightarrow$

$n^n \leq n!^2$

$\Leftrightarrow$

True

$\log(n^n) = n * \log n \sqsubseteq \log n!$

5)

5. Sean K y L constantes, y f el siguiente procedimiento:

proc f(**in** n : **nat**)
  if $n \leq 1$ **then skip**
  else
    for $i := 1$ **to** K **do** $f(n$ **div** $L)$ **od**
    for $i := 1$ **to** n^4 **do** *operación_de_*$\mathcal{O}(1)$ **od**

Determiná posibles valores de K y L de manera que el procedimiento tenga orden:

(a) $n^4 \log n$ (b) n^4 (c) n^5

```
proc f(in n : nat)
     if n ≤ 1 then skip
     else
          for i ≔ 1 to K do f(n div L) od
          for i ≔ 1 to n⁴ do operación_de_O(1) od
     fi
end proc
```

Sea:

```
t(n) = ops(f(n))
```

```
     t(n)
=
     ops(if n ≤ 1 then skip
          else
               for i ≔ 1 to K do f(n div L) od
               for i ≔ 1 to n⁴ do operación_de_O(1) od
          fi)
=
     (□ n ≤ 1 → 0
      □ si no → ops(for i ≔ 1 to K do f(n div L) od)
                + ops(for i ≔ 1 to n⁴ do operación_de_O(1) od)
     )
=
     (□ n ≤ 1 → 0
      □ si no → ⟨∑i = 1 to K : ops(f(n div L))⟩)
                + ⟨∑i = 1 to n⁴ : operación_de_O(1)⟩)
     )
≈
     (□ n ≤ 1 → 0
      □ si no → K * t(n div L) + n⁴
     )
≈
```

```
(□ K > L^4 → n^(log_L(K))
 □ K = L^4 → n^4 * log(n)
 □ si no   → n^4
)
```

5a) n^4 * log n

Para que tanga orden n^4 * log n, se tiene que entrar en la segunda guarda, ósea se tiene que cumplir $K = L^4$

5b) n^4

Para que tenga orden n^4, se tiene que entrar en la tercera guarda, ósea tiene que cumplir $K < L^4$

5c) n^5

Para que tenga orden n^5, se tiene que entrar en la primer guarda con $\log_L(K) = 5$, osea, $K = L^5$, esto ya cumple la condición de entrada a la guarda, que es $K > L^4$, así que queda $K = L^5$

6) 🛠️

6. Escribí algoritmos cuyas complejidades sean (asumiendo que el lenguaje no tiene multiplicaciones ni logaritmos, o sea que no podés escribir **for** i:= 1 **to** $n^2 + 2\log n$ **do** . . . **od**):

(a) $n^2 + 2\log n$ (b) $n^2 \log n$ (c) 3^n

6a)

```
n^2 + 2 * log n ≈ n^2
proc ej6a(in n : nat)
    for j ≔ 1 to n do
        for k ≔ 1 to n do
            operaciónContabilizada
        od
    od
    var j : nat
```

```
    j ≔ n
    while j > 1 do
        operaciónContabilizada
        operaciónContabilizada
        j ≔ j `div` 2
    od
end proc
```

6b)

```
proc ej6b(in n : nat)
    for j ≔ 1 to n do
        for k ≔ 1 to n do
            var j : nat
                j ≔ n
                while j > 1 do
                    operaciónContabilizada
                    j ≔ j `div` 2
                od
        od
    od
end proc
```

6c)

```
proc ej6c(in n : nat)
    if n = 0 →
        operaciónContabilizada
    else
        ej6c(n-1)
        ej6c(n-1)
        ej6c(n-1)
    fi
end proc
```

7)

7. Una secuencia de valores $x_1, ..., x_n$ se dice que tiene *orden cíclico* si existe un i con $1 \leq i \leq n$ tal que $x_i < x_{i+1} < \ldots < x_n < x_1 < \ldots < x_{i-1}$. Por ejemplo, la secuencia $5, 6, 7, 8, 9, 1, 2, 3, 4$ tiene orden cíclico (tomando $i = 6$).

   (a) Escribí un algoritmo que determine si un arreglo almacena una secuencia de valores que tiene orden cíclico o no.

   (b) Escribí un algoritmo que dado un arreglo a : **array**$[1..n]$ **of nat** que almacena una secuencia de valores que tiene orden cíclico, realice una búsqueda secuencial en el mismo para encontrar la posición del menor elemento de la secuencia (es decir, la posición i).

   (c) Escribí un algoritmo que resuelva el problema del inciso anterior utilizando la idea de *búsqueda binaria*.

   (d) Calculá y compará el orden de complejidad de ambos algoritmos.

7a)

```
fun ordenCíclico(a : array[1..n] of nat) out res : bool
      res ≔ True
      var j : nat
      j ≔ 1
      while j < n ∧ a[j] < a[j+1] do
            j ≔ j+1
      od
      if j < n →
            j ≔ j+1
            while j < n ∧ res do
                  res ≔ a[j] < a[j+1]
                  j ≔ j+1
            od
            res ≔ res ∧ a[n] < a[1]
      fi
end fun
```

7b)

```
fun tamOrdenCíclico(a : array[1..n] of nat) out res : nat
      var j : nat
      j ≔ 1
      while j < n ∧ a[j] < a[j+1] do
            j ≔ j+1
      od
      res ≔ j + 1
end fun
```

7c)

```
fun gen_tamOrdenCíclico(a : array[1..n] of nat, izq, der : nat) out res : nat
      var med : nat
      med ≔ (izq + der) `div` 2
```

```
    if izq < der →
        if a[med] < a[1] →
            res ≔ gen_tamOrdenCíclico(a, izq, med)
        else
            res ≔ gen_tamOrdenCíclico(a, med, der)
        fi
    else
        res = med
    fi
end fun


fun tamOrdenCíclico(a : array[1..n] of nat) out res : nat
    res ≔ gen_tamOrdenCíclico(a, 1, n)
end fun
```

8)

8. Calculá el orden de complejidad del siguiente algoritmo:

```
proc f3(n : nat)
    for j := 1 to 6 do
        if n ≤ 1 then skip
        else
            for i := 1 to 3 do f3(n div 4) od
            for i := 1 to n⁴ do t := 1 od
    od
```

```
proc f3(n : nat)
    for j ≔ 1 to 6 do
        if n ≤ 1 then skip
        else
            for i ≔ 1 to 3 do
                f3(n `div` 4)
            od
            for i ≔ 1 to n⁴ do
                t ≔ 1
            od
        fi
    od
end proc
```

Cuento la cantidad de asignaciones a `t`, sea:
`r(n) = ops(f3(n))`

```
    r(n)
```

= {`ops` del `for`, divido en casos del `if`}

```
    ∑j = 1 to 6 : (□ n ≤ 1 → 0
                   □ si no → ops(for i ≔ 1 to 3 do f3(n `div` 4) od
                                 for i ≔ 1 to n⁴ do t ≔ 1 od
                                )
                  )
```

= {`j` no aparece en el término, y los `i` tampoco aparece dentro de los `for`}

```
    6 * (□ n ≤ 1 → 0
         □ si no → 3 * ops(f3(n `div` 4)) + n⁴
        )
```

= {Meto el producto en las guardas, definición `r`}

```
    (□ n ≤ 1 → 0
     □ si no → 18 * r(n `div` 4) + 6*n⁴
    )
```

≈ {Teorema del orden de convergencia, caso `a < b^k` (`a = 18`, `b = 4`, `k = 4`) }

n^4