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

(a)

```
proc f1(in n : nat)
    if n ≤ 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
```

- caso simple: $n \le 1$
- caso complejo: partir al medio a n.
  - $a = 8$
  - $b = 2$
  - g(n) = ops(for i := 1 to n^3 do t := 1 od) = sumatoria desde i = 1 to n^3 (ops(t:=1))
    = sumatoria desde i = 1 to n^3 (1) = n^3
  - Luego $k = 3$
  - como a = b^k, es decir, $8 = 2^3 \rightarrow$ t(n) es de orden **$n^3.\log(n)$**

(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
```

- Caso simple ($n < 0$) : 0
- Caso complejo g(n) = ops(for i := 1 to n do C(i) od)
  = sumatoria desde i = 1 to n (ops C(i))
  = sumatoria desde i = 1 to n (ops for j := 1 to i do t := 1 od)
  = s desde i = 1 to n (s desde j = 1 to i (ops(t := 1)))
  = s desde i = 1 to n (s desde j = 1 to i 1)
  = s desde i = 1 to n (i) = **$(n*(n+1))/2 = n^2$**
- Caso complejo ($n > 0$)= se parte al medio n 4 veces.
  - a = 4 (llamadas recursivas)
  - b = 2 (se divide a n en dos)
  - k = 2 ya que el orden de g(n) = n^2
  - a = b^k puesto que $4 = 2^2$. Por lo tanto t(n) es de orden **$n^2.\log(n)$**

2. Dado un arreglo a : array[1..n] of nat se define una cima de a como un valor k en el intervalo 1, . . . , 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) Calcule y compare el orden de complejidad de ambos algoritmos.

a)

```
fun tieneCima(a:array[1..n] of nat) ret result : bool
      result := false
      if (n = 1) → result := true
        (n = 2) → if (a[1] ≤ a[2]) → result := true fi
        (n ≥ 2) →
                  var i : nat
                        i := 2
                  while (i < n && !(result)) then
                        if(a[i] > a[i-1] && a[i] < a[i+1]) →
                              result := creciente(a,1,i) &&
                        decreciente(a,i,n)
                        fi
                        i := i + 1
                  od
      fi
end fun

proc creciente(in a:array[1..n] of nat, in lft, rgt : nat) ret r :
bool
      var i : nat
      i := lft
      while (i < rgt && a[i] ≤ a[i+1]) do
            i := i + 1
      od
      r := (i ≡ rgt)
end proc

proc decreciente(in a:array[1..n] of nat, in lft, rgt : nat) ret r :
bool
      var i : nat
      i := lft
      while (i < rgt && a[i] ≥ a[i+1]) do
            i := i + 1
      od
      r := (i ≡ rgt)
```

```
end proc
```

b)

```
proc tieneCima(in a:array[1..n]of nat) ret cima : nat
      if (n = 0) → cima := 0
      if (n = 1) → cima := n
      if (n = 2) → if (a[1] ≤ a[2]) → cima := 2
                   else → cima := 1
                   fi
      if (n ≥ 2) →
      for i = 1 to n-1 do
            if (a[i] > a[i+1]) →
                  cima := i
            fi
      od
      if (i == n - 1) →
            cima := n
      fi
      fi
end proc
```

c)

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

fun tieneCimaRet(a:array[1..n] of nat, lft,rgt : nat) ret res : nat
      var mid : nat
      if (lft > rgt) → res := 0
        (lft ≤ rgt) → mid := (lft + rgt)/2
                        if(a[mid] > a[mid-1] && a[mid] > a[mid+1])→
                              res := mid
                          (a[mid] < a[mid-1] && a[mid] > a[mid+1]) →
                              res := tieneCimaRet(a,lft,mid-1)
                          (a[mid] > a[mid-1] && a[mid] < a[mid+1]) →
                              res := tieneCimaRet(a,mid+1,rgt)
                        fi
      fi
end fun
```

d)

En el peor de los casos la búsqueda secuencial/lineal es de orden n
En el peor de los casos la búsqueda binaria es de orden log(n)

3. El siguiente algoritmo calcula el ḿınimo elemento de un arreglo a : array[1..n] of nat mediante la técnica de programación divide y venceras. Analiza la eficiencia de minimo(1,n).

```
fun mínimo(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(mínimo(a, i, j), mínimo(a,j+1,k))
        fi
end fun
```

contamos la cantidad de asignaciones a m:
p es el tamaño del arreglo.
f(p) = (si p = 1 → 1

- si p > 1 → 1 + f (p div 2) + f (p div 2) = 1 + 2 * f (p div 2)

Caso simple: (i = k) → 1
Caso complejo: a = 2, b = 2, k = 0
ahora como a > b^k puesto que 2 > 2^0
el orden del algoritmo es n^log_2(2) = n (es lineal)

4. Ordena utilizando ❁ e ≈ los órdenes de las siguientes funciones. No calcules límites, utiliza las propiedades algebraicas.

(a) n log(2^n)   2^n log(n)   n! log(n)   2^n

- n log (2^n) = n^2 * log(2) = n^2
- 2^n * log(n)
- n! log(n)
- 2^n
    - n^2 ❁ 2^n ❁ 2^n * log(n) ❁ n! log(n)

(b) n^4 + 2log(n)   log(((n)^n)^4)   2^(4*log(n))   4^n   n^3 log(n)

- n^4 + 2log(n)
- log(((n)^n)^4) = 4*log((n)^n) = 4*n*log(n) = n*log(n)
- 2^(4*log(n)) = (2^log(n))^4 como 2^log(n) = n nos queda n^4
- 4^n = (2^2)^n = 2^2n
- n^3 log(n)
    - n*log(n) ❁ n^3 log(n) ❁ n^4 ≈ n^4 + 2log(n) ❁ 2^2n

(c) log(n!)   n*log(n)   log(n^n)

- log(n!)
- n*log(n)
- log(n^n) = n * log(n)
  - log(n!) ❁ n * log(n) ~ n * log(n)

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

```
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 n^4 do
                        operacion_de_O(1)
                od
end proc
```

Determina posibles valores de K y L de manera que el procedimiento tenga orden:
(a) n^4 log(n)
(b) n^4
(c) n^5

a) K = 16 , L = 2
   - en este caso tenemos que g(n) es de orden n^4, es decir k = 4
   - luego como L = 2, b = 2
   - finalmente como K = 16, se llama 16 veces al proc f, por lo que a = 16
   - tenemos a = b^k es decir 16 = 2^4, lo cual da n^4 log(n)

b) De forma general, K ≤ L^4 y L puede ser cualquier valor.
   En un caso particular K = 1 y L = 2
   - g(n) es de orden n^4 → k = 4
   - Como L = 2, b = 2
   - Como K = 1, a = 1
   - Tenemos a < b^k, es decir, 1 < 2^4, lo cual da orden n^4

c) La parte recursiva debe aportar un orden de n
   La parte g(n) nos aporta un orden de n^4
   - segun la formula : a * t(n/b) + n^4
   - Pendiente

6. Escribı algoritmos cuyas complejidades sean (asumiendo que el lenguaje no tiene multiplicaciones ni logaritmos, o sea que no podes escribir

for i:= 1 to n^2 + 2 log (n) do . . . od): 1

(a) n^2 + 2 log(n) = n^2 + log(n^2)

(b) n^2 log(n)

(c) 3^n

a)

escribimos uno de complejidad n^2 y otro de complejidad 2*log(n)

```
var x : nat
x := 0
for i := 1 to n do
      for j := 1 to n do
            x := x + 1
      od
od     {- hasta aca, n^2 -}
for i := 1 to 2 do
      k := n
      while (k > 1) do
            k := k div 2
      od  {- esto es de orden log(n) -}
od
```

b)

```
proc f(in n : nat)
      if n ≤ 1 then
            skip
      else
            for i := 1 to 4 do
                  f(n div 2)
            od
            for i := 1 to n^2 do
                  operacion_de_O(1)
            od
end proc
```

analizando:

- se llama 4 veces a f(n), a=4
- se divide a n por la mitad por cada llamada, b = 2

- el orden de g(n) es n^2
- tenemos que a = b^k, es decir, 4 = 2^2. Por lo que el orden es de n^2 * log(n)

c)

```
fun producto3 (n : nat) ret result : nat
      if (n = 0) →
            result := result * 1
      else →
            result := 3 * producto3 (n-1)
      fi
end fun
```

analizando: esto literalmente da 3^n xdd