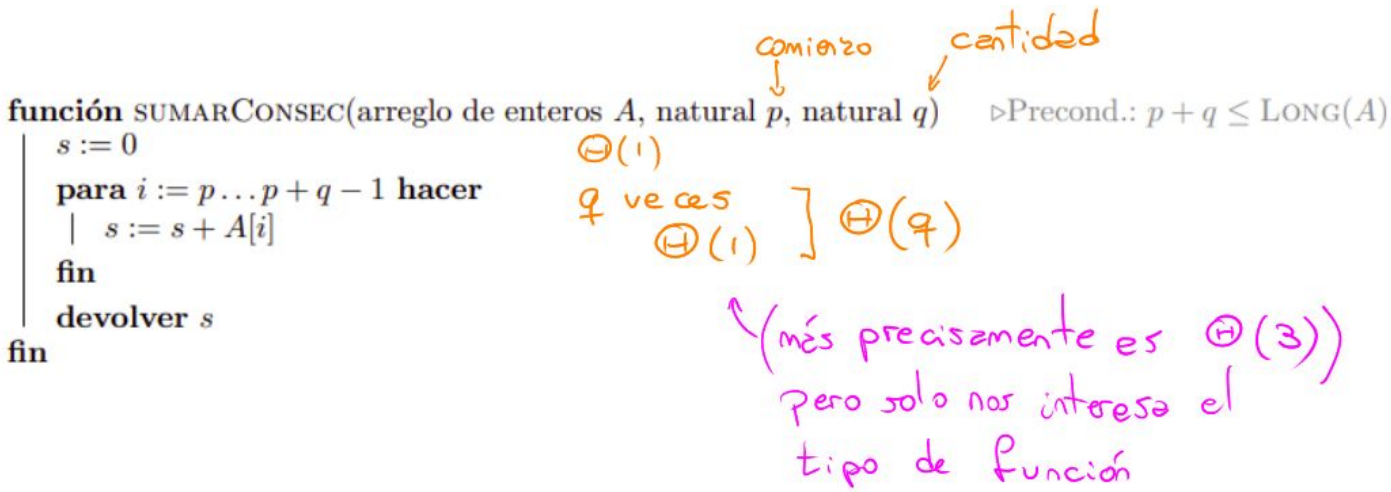
Algoritmos y Estructuras de Datos
Parcial 2

Alumno: Leandro Carreira
LU: 669/18

Ej. 1. Complejidad

Ej. 1.1. Suma de consecutivos

1. **`f(q) = q`**



La función sumarConsec toma un arreglo de enteros de alguna longitud, y suma **q** naturales consecutivos a partir del elemento **p** .

Como la cantidad de elementos que suma está determinada completamente por **q** (ya que la precondición asegura que siempre se sumen **q** elementos, y no menos), y cada una de las operaciones de suma dentro del ciclo es constante, entonces la cantidad total de operaciones, siempre será exactamente **q** .

2. Mejor caso de **`ConsecSuman0?(A) es Ω(1)`**

   El mejor caso se da cuando **`SumarConsec(A, pos, cuantos)`** es **cero** la primera vez (en la primer iteración de cada ciclo):

```
función CONSECSUMAN0?(arreglo de enteros A)
    n := LONG(A)
    para cuantos := 1 ... n hacer
        para pos := 0 ... n − cuantos hacer
            si SUMARCONSEC(A, pos, cuantos) = 0 entonces
                devolver true
            fin
        fin
    fin
    devolver false
fin
```

Ω(1)
1 vez
1 vez
Ω(cuantos) (por Ej 1)
Ω(1) pues si
Θ(g) => Ω(g)

Más precisamente, esto sucede cuando **EL PRIMER** elemento de la secuancia **es 0**, en cuyo caso, **`cuantos`** será **1**, la complejidad de mejor caso de SumarConsec será **Ω(1)**, y se retornará el valor true.

3. Respuesta: **$f(Long(A)) = (Long(A))^3$**

El peor caso es cuando ninguna suma de consecutivos sume 0 (por ejemplo todos elementos positivos)

```
función CONSECSUMAN0?(arreglo de enteros A)                    Peor caso
    n := LONG(A)
    para cuantos := 1 ... n hacer ..... n veces
        para pos := 0 ... n − cuantos hacer ..... (n - cuantos)
            si SUMARCONSEC(A, pos, cuantos) = 0 entonces ..... O(cuantos)
                devolver true
            fin
        fin
    fin
    devolver false ..... O(1)
fin
```

Si solo cuento iteraciones

$n-1$ veces
$n-2$ veces
$n-3$ veces
$\vdots$
$n-(n-1) = 1$ vez
$n-n = 0$ veces

$$\sum_{i=0}^{n-1} i = \frac{n\cdot(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Pero **cada una de estas n iteraciones** tiene complejidad que depende de la **cantidad a sumar**:

Complejidad de SUMARCONSEC en cada iteración

$n-1$ veces
$n-2$ veces
$n-3$ veces
$\vdots$
$n-(n-1) = 1$ vez
$n-n = 0$ veces

$$\sum_{i=0}^{n-1} i \cdot O(n-i) =$$

$$= \sum_{i=0}^{n-1} (n-i)\cdot O(i) = \sum_{i=0}^{n-1} n\,O(i) - i\,O(i)$$

$$= n \cdot \sum_{i=0}^{n-1} O(i) - \sum_{i=0}^{n-1} O(i^2)$$

CA:

$$\sum_{i=0}^{n-1} O(i^2) = \left(\sum_{i=1}^{n} O(i^2)\right) - O(n^2)$$

Tengo fórmula cerrada!

$$= O\left(\frac{n \cdot (n+1) \cdot (2n+1)}{6}\right) - O(n^2)$$

$$= O\left(\frac{2n^3}{6} + \frac{2n^2}{6} + \frac{n^2}{6} + \frac{n}{6}\right) - O(n^2)$$

$$= O\left(\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n\right)$$

$$= n \cdot O\left(\frac{n(n-1)}{2}\right) - O\left(\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n\right)$$

$$= O\left(\frac{n^3}{2} - \frac{n^2}{2} - \left(\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n\right)\right)$$

me quedo solo con el mayor exponente

$$= O\left(\frac{1}{2}n^3 - \frac{1}{3}n^3\right)$$

$$= O\left(\frac{1}{6}n^3\right)$$

$$= O(n^3)$$

Ej. 1.2

1. Por definición de b-suave:

   si una función f es b-suave =>

   f es eventualmente no decreciente y $f_b \in O(f)$

   con $f_b(n) = f(b*n)$

   Por definición de Big-O (para el dominio $\mathbb{Z}^+$ **de f)**

   si $f_b \in O(f) \Rightarrow \exists n_0 \in \mathbb{Z}^+, k \in \mathbb{R}^+ / f_b(n) \leq k*f(n) \quad \forall n > n_0$

   usando que $f_b(n) = f(b*n)$

   si $f_b \in O(f) \Rightarrow \exists n_0 \in \mathbb{Z}^+, k \in \mathbb{R}^+ / f(b*n) \leq k*f(n) \quad \forall n > n_0$

   qvq

   si una función f es b-suave entonces $f_b \in \Theta(f)$

   Sé que $f_b \in O(f)$, basta ver que $f_b \in \Omega(f)$

   Como f_b es no decreciente a partir de N:

   $f_b(x) \leq f_b(y)$ para todo $N \leq x \leq y$

   Dicho de otra manera, puedo asegurar que a partir de algún **m,** f será no decreciente y por lo tanto tendré una cota inferior para la misma.

   Por lo tanto, puedo decir que:

   si f es b-suave, entonces

   $\exists \mathbf{m} \in \mathbb{Z}^+, k_1, k_2 \in \mathbb{R}^+ / k_1*f(n) \leq f(b*n) \leq k_2*f(n) \quad \forall n > \mathbf{m}$

   en otras palabras:

   $$f_b \in \Theta(f)$$

2. Como f es b-suave, f es eventualmente no decreciente y $f_b \in O(f)$

   Si a < b, como $f_b(x) \leq f_b(y)$ a partir de algún **m** para todo $m \leq x \leq y$

   $f(a*n) \leq f(b*n)$

   $\exists \mathbf{m} \in \mathbb{Z}^+, k \in \mathbb{R}^+ / f(a*n) \leq f(b*n) \leq k*f(n) \quad \forall n > \mathbf{m}$

   Por lo tanto, como $f(a*n) = f_a(n)$ esta acotado por arriba a partir de algún **m** por $f_b(n)$ que es $O(f)$, entonces

   $f_a(n) \in O(f)$

   Y como también vale que es eventualmente no decreciente, puedo decir que:

   f es a-suave

5. f(n) = n^k

Es suave si eventualmente es no decreciente y $f_b \in O(f)$ con b en Z^+

Quiero ver que $f(b*n) \in O(f)$ (pues ya se que es creciente por ser exponencial de positivos)

f(b*n) = (b*n) ^ k
= b^k * n^k
Llamo c=b^k
= c*n^k

Por lo tanto, como es eventualmente creciente, a partir de un n0 puedo decir que
$\exists n_0 \in \mathbb{Z}^+, d \in \mathbb{R}^+ / f(b*n) \leq d*f(n) \quad \forall n > n_0$ para algún d > c

Que es la definición de O grande, y por lo tanto $f_b \in O(f)$

Y como si f es b-suave => es suave

f es suave

5. g(n) = n^log(n)
g(b*n) = (b*n)^log(b*n)
= b^log(b*n) * n^log(b*n)
= b^log(b) * b^log(n) * n^log(b) * n^log(n)
Veo que O(n^log(n)) es O(g), pero al multiplicar por n^log(b) y b^log(n), por propiedad de O grande, su complejidad es O del producto de funciones, por lo que deja de ser acotada por n^log(n) y NO es O(g)
Por lo tanto, **g NO es suave**.

5. h(n) = n^n
h(b*n) = (b*n)^(b*n)
= b^(b*n) * n^(b*n)
Noto que n^(b*n) es O(h), pero b^(b*n) es O(b^n)
Y por propiedad de O grande, su producto es O(b^n * n^(b*n)) que no puede ser acotado por O(h)
Por lo tanto, **h NO es suave**.

Ej. 2

scf **se representa con** estr, donde

estr es tupla
⟨ *esperando* : secu(Persona)
enCancha : dicc(Persona, Cancha)
libres : conj(Cancha)
asistencias : dicc(Cancha, dicc(Persona, nat))
⟩

Nota: Los escribo en el mismo órden que están en el enunciado.

Invariante:

1) No puede haber personas repetidas esperando.
2) Las personas que están actualmente en alguna cancha, no pueden estar esperando.
   (no hace falta pedir que no se repitan personas entre distintas canchas, pues la estructura de diccionario asegura que no haya claves repetidas)
3) Las canchas libres no pueden aparecer como significado de alguna persona en cancha (no puede haber personas en una cancha libre).
4) La cantidad de asistencias por cancha es múltiplo de 10 (pues todos los partidos son en grupos de 10).
5) Las personas que ya jugaron en alguna cancha, habrán asistido al menos una vez (no puede tener cero asistencias).
   (*las asistencias se actualizan cuando la gente se retira de la cancha, por lo que puede haber personas esperando o en la cancha, sin asistencias)*
6) Cada persona pudo haber asistido como máximo 1/10 de las veces de la cantidad de asistencias en esa cancha.
   (La cantidad de asistencias por cancha debe corresponder a grupos de 10 personas distintas: no puede haber solo una persona con 10 asistencias, ni 5 personas con 2 asistencias cada una).

Formal:

Rep :: estr → bool
Rep(e) = (1) ∧ (2) ∧ (3) ∧ (4) ∧ (5) ∧ (6)

(1): (∀ i,j : Nat) i, j < **long**(e.esperando) $\Rightarrow_L$
i ≠ j ⇒ ***iesimo***(e.esperando, i) ≠ ***iesimo***(e.esperando, j)

*Asumo que tengo función "**iesimo**" que devuelve el iésimo elemento de una secuencia dada una secuencia y un índice i.*

(2): (∀ p : Persona) p ∈ **claves**(e.enCancha) ⇒ ¬**está?**(p, e.esperando) ∧
está?(p, e.esperando) ⇒ p ∉ **claves**(e.enCancha)

(3): (∀ p : Persona) p ∈ **claves**(e.enCancha) $\Rightarrow_L$ **obtener(p,** e.enCancha) ∉ e.libres

(4): (∀ c : Cancha)(∀ p : Persona)
c ∈ **claves**(e.asistencias) $\Rightarrow_L$ ($\sum_{p \in claves(e.asistencias[c])}$ e.asistencias[c][p]) % **10 = 0**

(5)y(6): (∀ c : Cancha)(∀ p : Persona)
c ∈ **claves**(e.asistencias) $\Rightarrow_L$ p ∈ **claves**(e.asistencias[c]) $\Rightarrow_L$ *
* $\Rightarrow_L$ e.asistencias[c][p] > 0 ∧
e.asistencias[c][p] ≤ ($\sum_{p \in claves(e.asistencias[c])}$ e.asistencias[c][p]) / **10**

Uso sumatoria $\sum_{p \in claves(e.asistencias[c])}$ para recorrer todos los elementos del conjunto de claves de las personas que asistieron a la cancha c, y sumar las asistencias de cada uno de ellos.

Uso corchetes como notación corta para obtener un elemento de un diccionario:
e.asistencias[c] ≡ obtener(c, e.asistencias)
e.asistencias[c][p] ≡ obtener(p, obtener(c, e.asistencias))

Función de Abstracción:

TAD SCF

observadores básicos

Canchas : SCF $\longrightarrow$ conj(Cancha)

Ocupadas : SCF $\longrightarrow$ conj(Cancha)

Jugando : SCF s × Cancha c $\longrightarrow$ conj(Persona) $\{c \in \text{Canchas}(s) \wedge_L c \in \text{Ocupadas}(s)\}$

SalaEspera : SCF $\longrightarrow$ conj(Persona)

Asistencias : SCF s × Cancha c $\longrightarrow$ Dicc(Persona, nat) $\{c \in \text{Canchas}(s)\}$

scf **se representa con** estr, donde

estr es tupla

⟨ *esperando* : secu(Persona)
enCancha : dicc(Persona, Cancha)
libres : conj(Cancha)
asistencias : dicc(Cancha, dicc(Persona, nat))
⟩

```
Abs :: estr e ⟶ SCF s                                  { Rep(e) }
Abs(e) =obs s ⇔
          Canchas(s)     = claves(e.asistencias)            ∧
          Ocupadas(s)    = claves(e.asistencias) - e.libres ∧
          SalaEspera(s)  = secuAConj(e.esperando)           ∧
          (∀ Cancha c en Canchas(s))
               Jugando(s, c)  = personasEnCancha(e, c, claves(e.enCancha)) ∧
               Asistencias(s) = obtener(c, e.asistencias)

-- Filtra de entre todas las personas jugando, aquellas en la cancha c
personasEnCancha(e, c, personas) ≡
               if vacío?(personas) then
                    ∅
               else
                    if c = obtener(dameUno(personas), e.enCancha) then
                         Ag(dameUno(personas), personasEnCancha(e, c, sinUno(personas))
                    else
                         personasEnCancha(e, c, sinUno(personas))
                    fi
               fi

-- Pasa los elementos de una secuencia a un conjunto
secuAConj(s) ≡  if vacía?(s) then
                    ∅
               else
                    Ag(prim(s), secuAConj(fin(s)))
               fi
```