Algoritmos y Estructuras de Datos II - 29 de abril de 2013

Primer Parcial

Alumno: .....

1.

```
proc A(in/out a: array[1..n] of T)
    var x: nat
    for i:= 1 to n-1 do
        x:= i
        for j:= i+1 to n do
            if a[j] < a[x] then x:= j fi
        od
        swap(a,i,x)
    od
end proc

proc B (in/out a: array[1..n] of T)
    var q: bool
    q:= true
    i:= 1
    do i ≤ n-1 ∧ q →
        q:= false
        for j:= n-1 downto i do
            if a[j] > a[j+1] then swap(a,j,j+1)
                                  q:= true
            fi
        od
        i:= i+1
    od
end proc

proc C (in/out a: array[1..n] of T)
    for i:= 2 to n do
        j:= i
        do j > 1 ∧ a[j] < a[j − 1] → swap(a,j,j-1)
                                      j:= j-1
        od
    od
end proc

proc D (in/out a: array[1..n] of T)
    for i:= 1 to n-1 do
        for j:= n-1 downto i do
            if a[j] > a[j+1] then swap(a,j,j+1) fi
        od
    od
end proc

proc E (in/out a: array[1..n] of T)
    F(a,1,n)
end proc

proc F (in/out a: array[1..n] of T, in u,v: nat)
    var t: nat
    if v > u → G(a,u,v,t)
               F(a,u,t-1)
               F(a,t+1,v)
    fi
end proc

proc G (in/out a: array[1..n] of elem, in u,v: nat, out t: nat)
    var i,j: nat
    t:= u
    i:= u+1
    j:= v
    do i ≤ j → if a[i] ≤ a[piv] → i:= i+1
               a[j] > a[t] → j:= j-1
               a[i] > a[t] ∧ a[j] ≤ a[t] → swap(a,i,j)
                                           i:= i+1
                                           j:= j-1
               fi
    od
    swap(a,t,j)
    t:= j
end proc
```

Unir con líneas según corresponda, y **justificar**. Puede haber cero, una o más líneas saliendo del mismo algoritmo a diferentes afirmaciones. Igualmente, a una misma afirmación pueden llegar cero, una o varias líneas.

algoritmos	afirmaciones
A	(1) en el mejor caso es lineal
B	(2) en el peor caso es cuadrático
C	(3) en la práctica es del orden de $n \log n$
D	(4) siempre es del orden de $n \log n$
E	(5) siempre es del orden de n^2

2. *a*) Dado el siguiente algoritmo, plantear la recurrencia que indica la cantidad de asignaciones realizadas a la variable m en función de la entrada n:

```
fun f (n: nat) ret m: nat
    if n ≤ 2 then
        m := n
    else
        m := f(n-1) + f(n-2) - f(n-3)
    fi
end
```

b) Resolver la siguiente recurrencia (notar que NO es la recurrencia solicitada en el ítem 2*a*.):

$$t(n) = \begin{cases} n & \text{si } n \leq 2 \\ t(n-1) + t(n-2) - t(n-3) & \text{si } n > 2 \end{cases}$$

3. Ordenar las siguientes funciones según el orden creciente de sus $\mathcal{O}$.
   a) $\log_2(n^n)$ *b*) $n^{\log_n n}$ *c*) $n^{1.001}$ *d*) $n^{0.001}$ *e*) 1.001^n

4. Un listado de palabras es una cadena de (cero o más) caracteres que pueden ser letras o espacios en blanco. Cada una de las palabras de un listado está separada de las otras por uno o más espacios.

   El TAD *listado de palabras* tiene como constructores **vacío** que genera el listado vacío (sin caracteres) y **agrega-char** que agrega un carácter al final de un listado, y las operaciones:

   cuenta que dados un listado y un carácter devuelve la cantidad de apariciones del carácter en el listado
   reducir que reduce los espacios de un listado al mínimo: elimina los del principio y del final y deja sólo un espacio entre palabras consecutivas
   cuenta-palabras que devuelve la cantidad de palabras en un listado
   quitar-última que elimina la última palabra de un listado (con al menos una palabra)

   Completar la siguiente especificación de *listado*, donde ' ' representa un espacio en blanco (puede utilizar otras ecuaciones en vez de seguir el esquema planteado):

   TAD listado

   constructores
   vacío :
   agrega-char : listado × char →

   operaciones
   cuenta :
   reducir :
   cuenta-palabras :
   quitar-última :

   ecuaciones
   cuenta(vacío,c) =
   c = c' ⇒ cuenta(agrega-char(l,c), c') =
   c ≠ c' ⇒ cuenta(agrega-char(l,c), c') =
   reducir(vacío) =
   c = ' ' ⇒ reducir(agrega-char(l,c)) =
   c ≠ ' ' ⇒ reducir(agregar-char(vacío,c)) =
   c = ' ' ∧ c' ≠ ' ' ⇒ reducir(agregar-char(agregar-char(l,c),c')) =
   c ≠ ' ' ∧ c' ≠ ' ' ⇒ reducir(agregar-char(agregar-char(l,c),c')) =
   reducir(l) = vacío ⇒ cuenta-palabras(l) =
   reducir(l) ≠ vacío ⇒ cuenta-palabras(l) =
   c ≠ ' ' ⇒ quitar-última(agregar-char(vacío,c)) =
   c = ' ' ⇒ quitar-última(agregar-char(l,c)) =
   c = ' ' ∧ c' ≠ ' ' ⇒ quitar-última(agregar-char(agregar-char(l,c),c')) =
   c ≠ ' ' ∧ c' ≠ ' ' ⇒ quitar-última(agregar-char(agregar-char(l,c),c')) =

   En caso de creerlo conveniente, puede utilizar otras ecuaciones en vez de seguir el esquema planteado.

5. Implementar utilizando un **nat** el TAD bin que se especifica a continuación. Intuitivamente, uno $\lhd_0 \lhd_0 \lhd_1 \lhd_0$ corresponde al número binario 10010.

   TAD bin

   constructores
   uno : bin
   _ $\lhd_0$: bin → bin
   _ $\lhd_1$: bin → bin

   operaciones
   es_uno : bin → booleano
   es_par : bin → booleano
   mover : bin → bin { pre: argumento ≠ uno }
   suc : bin → bin
   sum : bin × bin → bin

   ecuaciones
   es_uno(uno) = verdadero
   es_uno(b $\lhd_0$) = falso
   es_uno(b $\lhd_1$) = falso
   es_par(uno) = falso
   es_par(b $\lhd_0$) = verdadero
   es_par(b $\lhd_1$) = falso
   mover(b $\lhd_0$) = b
   mover(b $\lhd_1$) = b
   suc(uno) = uno $\lhd_0$
   suc(b $\lhd_0$) = b $\lhd_1$
   suc(b $\lhd_1$) = suc(b) $\lhd_0$
   sum(uno,c) = suc(c)
   sum(b,uno) = suc(b)
   sum(b $\lhd_0$,c $\lhd_0$) = sum(b,c) $\lhd_0$
   sum(b $\lhd_0$,c $\lhd_1$) = sum(b,c) $\lhd_1$
   sum(b $\lhd_1$,c $\lhd_0$) = sum(b,c) $\lhd_1$
   sum(b $\lhd_1$,c $\lhd_1$) = suc(sum(b,c)) $\lhd_0$