PAC2: Segona Prova d'Avaluació Continuada

Format i data de lliurament

Cal lliurar la solució en un fitxer de tipus **pdf** a l'apartat de lliuraments d'AC de l'aula de teoria.

La data límit per lliurar la solució és el **dilluns, 5 de novembre de 2018** (a les 23:59 hores).

Presentació

El propòsit d'aquesta segona PAC és comprovar que has adquirit els conceptes explicats en els capítols '*Recursivitat*' i '*TADs*'.

Competències

Transversals

- Capacitat de comunicació en llengua estrangera.
- Coneixements de programació amb llenguatge algorísmic.

Específiques

- Capacitat de dissenyar i construir algorismes informàtics mitjançant tècniques de desenvolupament, integració i reutilització.

Objectius

Els objectius d'aquesta PAC són:

- Adquirir els conceptes teòrics explicats sobre les tècniques d'anàlisi d'algorismes i recursivitat.
- Dissenyar funcions recursives, identificant els casos base i recursius, sent capaços de simular la seqüència de crides donada una entrada.
- Implementar un algorisme iteratiu a partir d'un algorisme recursiu.
- Manipular operacions dels TADs bàsics implementats amb punters.
- Dissenyar un TAD complex fruït de la combinació de TADs bàsics.

Descripció de la PAC a realitzar

Raona i justifica totes les respostes.

Les respostes incorrectes **no** disminueixen la nota.

Tots els dissenys i implementacions han de realitzar-se en llenguatge algorísmic. Els noms dels tipus, dels atributs i de les operacions s'han d'escriure en anglès. Els comentaris i missatges d'error no és obligatori fer-los en anglès, tot i que es valorarà positivament que es faci, ja que és l'estàndard.

Recursos

Per realitzar aquesta prova disposes dels següents recursos:

Bàsics

- Materials en format **web de l'assignatura.**
- **Fòrum de l'aula de teoria.** Disposes d'un espai associat en l'assignatura on pots plantejar els teus dubtes sobre l'enunciat.

Complementaris

- **Cercador web.** La forma més ràpida d'obtenir informació ampliada i extra sobre qualsevol aspecte de l'assignatura és mitjançant un cercador web.
- Solució de la PAC d'un semestre anterior.

Criteris de valoració

Per a la valoració dels exercicis es tindrà en compte:

- L'adequació de la resposta a la pregunta formulada.
- Utilització correcta del llenguatge algorísmic.
- Claredat de la resposta.
- Completesa i nivell de detall de la resposta aportada.

Avís

- Aprofitem per recordar que **està totalment prohibit copiar en les PACs** de l'assignatura. S'entén que hi pot haver un treball o comunicació entre els alumnes durant la realització de l'activitat, però el lliurament d'aquesta ha de ser individual i diferenciat de la resta.
- Així doncs, els lliuraments que continguin alguna part idèntica respecte a lliuraments d'altres estudiants seran considerats còpies i tots els implicats (sense que sigui rellevant el vincle existent entre ells) suspendran l'activitat lliurada.

Exercici 1: Conceptes bàsics de recursivitat (20%)

Tasca: Respon les preguntes següents justificant les respostes:

i) Qué necessita un algorisme recursiu per a que acabi?

Un algorisme perquè sigui recursiu ha de tenir una crida recursiva (cas recursiu) i perquè aquest finalitzi ha de tenir una condició de finalització (cas base).

ii) Quines són les avantatges i desavantatges d'utilitzar memòria dinàmica en la implementació d'un TAD ?

L'espai necessari per emmagatzemar una estructura dinàmica es reserva en temps d'execució, per la qual cosa no és necessari reservar ni conèixer l'espai de memòria a utilitzar abans de l'execució. D'aquesta forma es fa un ús òptim de la memòria, ja que també si no es necessita un espai prèviament reservat es pot alliberar.
D'altra banda podria succeir que no hi hagués memòria disponible suficient per continuar l'execució.

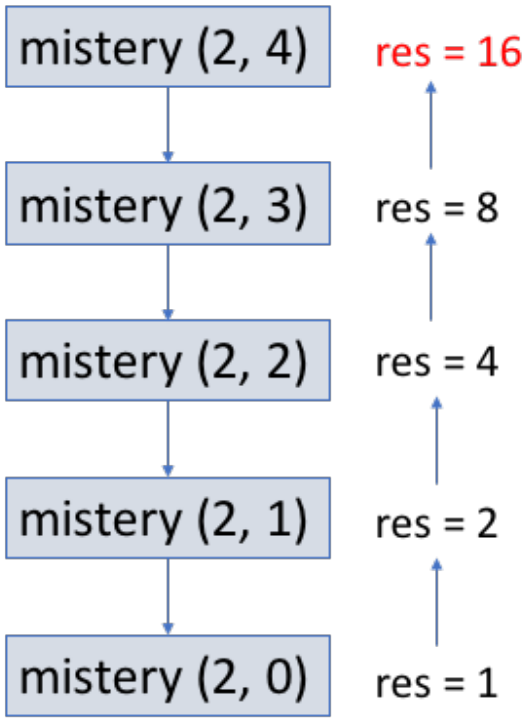
iii) Indica quin és l'objectiu de la funció *duplicate* en un TAD.

La funció duplicate realitza una còpia exacta d'un objecte sobre un altre del mateix TAD. Aquesta funció retorna un punter al nou objecte resultat. D'aquesta forma s'obté independència de la implementació del TAD. La implementació d'aquesta funció és depenent de la implementació del TAD triada i de l'estructura de dades concreta

iv) Donada la funció recursiva **mystery**, calcula quin valor retorna la crida **mystery (2, 4)** i completa el **model de les còpies** per veure com has arribat al resultat:

```
function mystery (x : integer, y:  integer ) : integer
{ pre: y ≥ 0, x ≥ 0 }
var res : integer; end var
        if y = 0 then
                res := 1;
        else
                res := mystery (x, y-1)*x;
        end if
        return res;
end function
```

El resultat és **16**, que resulta de la seqüència de crides que estan representades en la següent figura:


mistery (2, 4)
res = 16
mistery (2, 3)
res = 8
mistery (2, 2)
res = 4
mistery (2, 1)
res = 2
mistery (2, 0)
res = 1

Exercici 2: Disseny d'algorismes recursius (20%)

Tasca: Donada la descripció dels problemes següents, dissenya els algorismes recursius que els resolen.

Consell: Abans de començar a escriure cada algorisme, has d'identificar els casos base i recursius.

i) Dissenya la funció recursiva **sum_multiple_3** que retorna la suma dels **n** primers nombres múltiples de 3 a partir d'un enter positiu **x** donat (inclòs).

Exemple: sum_multiple_3 *(4, 2)* retorna 30.

```
function sum_multiple_3 ( n: integer, x: integer ) : integer
Pre: { 0 ≤ n, 0 < x }
var res : integer; end var
        if n = 0 then
                res := 0;
        else
                if x mod 3 = 0 then
                        res := x + sum_multiple_3 (n-1, x+1);
                else
                        res := sum_multiple_3 (n, x+1);
                end if
        end if
        return res;
end function
```

ii) Dissenya l'acció recursiva **stack_split** que donada una pila **p** d'enters elimina d'aquesta pila els elements menors que un valor n donat i els col·loca mantenint l'ordre original en una pila nova **q**. L'acció retorna també el nombre d'elements que han estat moguts de la pila p a la pila q.

Exemple: **stack_split** ([1, 8, 2, -9, 5, 3>, 4, q, res) retorna les piles modificades de la següent forma: la pila p (primer paràmetre): [8, 5> i la pila q: [1, 2, -9, 3>. El valor de res és 4.

```
action stack_split ( inout p: stack(integer), in n: integer,
                     out q: stack(integer), out res: integer)
```

```
action stack_split ( inout p: stack(integer), in n: integer,
                     out q: stack(integer), out res: integer)
var
        e : integer;
fvar
        if emptyStack(p) then
                q := createStack();
                res := 0;
        else
                e := top(p);
                pop(p);
                split_stack (p, n, q, res);
                if (e < n) then
                        push(e, q);
                        res := res + 1;
                else
                        push(e, p);
                end if
        end if
end action
```

Exercici 3: Convertir algorismes recursius en iteratius (20%)

Tasca: Donada una acció recursiva transforma-la en iterativa

i) Omple els requadres per finalitzar el disseny de la funció recursiva **sum_array** que calcula la suma dels n primers elements del vector v.

```
function sum_array (v: array[MAX] of integer, n: integer): integer
Pre: { 0 ≤ n ≤ MAX }
var
        res: integer;
fvar
        if n = 0 then
                res := 0 ;
        else
                res := sum_array (v, n-1);
                res := res + v[n];
        end if
        return res;
end function
```

ii) Transforma la funció recursiva **sum_array** en una funció iterativa.

```
function sum_array (m: integer, n: ) : integer
Pre: { 0 ≤ n ≤ MAX }
var
        res: integer;
fvar
        res := 0;
        while n ≠ 0 do
                res := res + v[n];
                n := n - 1;
        end while
        return res;
end function
```

Exercici 4: Modificació de TAD bàsics (20%)

Tasca: Donada la següent implementació del TAD cua (**queue**) amb punters (també explicada en els apunts):

```
type
        node = record
                e : elem;
                next : pointer to node;
        end record

        queue = record
                first, last : pointer to node;
        end record
end type
```

Estén el tipus afegint les operacions següents:

i) **dequeue_nesim**: l'acció retorna la cua, però sense l'element situat en la posició n-ésim (l'element apuntat pel punter *first* es considera en la posició 1). Per l'altre banda, si la posició no existeix, llavors la funció retorna un error.

```
action dequeue_nesim (inout q: queue(elem), in pos: integer)
Pre: { pos > 0 }

var
    cur, prev : pointer to node;
    e : elem;
fvar
    cur := q.first;
    while (cur ≠ NULL) and (pos > 1) do
            prev := cur;
            cur := cur^.next;
            pos := pos - 1;
    end while
    if cur = NULL then
            error { the position is greater than the number of elements }
    else
```

```
            if cur = q.first then
                q.first := q.first.next;
            else
                if cur = q.last then
                    q.last := prev;
                else
                    prev^.next := cur^.next;
                end if
            end if
            destroy(cur);
        end if
    end action
```

ii) **compare_queue**: funció que, donades dues cues, retorna cert si ambdues cues són iguals, és a dir que en posicions anàlogues els elements són iguals. En cas contrari la funció retorna fals.

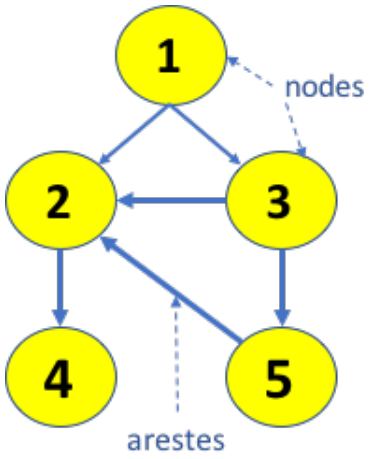
```
function compare_queue (q1: queue(elem), q2: queue(elem)): boolean
var
    cur1, cur2 : pointer to node;
    res : boolean;
fvar
    if (q1.first = NULL) and (q2.first = NULL) then
        res := true;
    else
        cur1 := q1.first;
        cur2 := q2.first;
        res := true;
        while (cur1 ≠ NULL) and (cur2 ≠ NULL) and (res) do
            res := equal (cur1^.elem, cur2^.elem);
            cur1 := cur1^.next;
            cur2 := cur2^.next;
        end while
        if (res) and (cur1 ≠ NULL or cur2 ≠ NULL)
            res := false;
        end if
    end if
    return res;
end function
```

Per dissenyar aquestes accions no pots utilitzar les operacions del tipus queue (head, enqueue, dequeue...). Així doncs, has de treballar directament amb la implementació del tipus que us hem facilitat en l'enunciat.

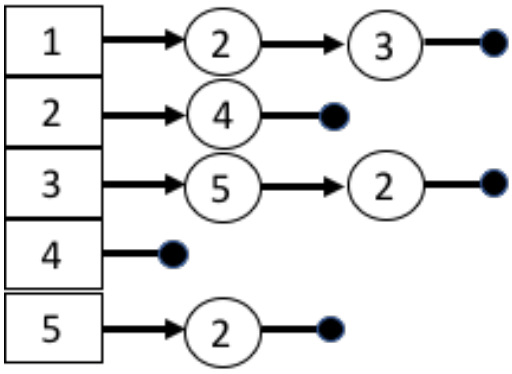
Exercici 5: Disseny d'un tipus amb punters (20%)

Tasca: Fins ara hem treballat amb els tipus de dades abstractes pila, cua i llista, però a vegades aquests no ens permeten reflectir la realitat i necessitem crear tipus més complexos (com per exemple, una llista ordenada).

En aquest exercici definim el TAD **tGraph** que implementa un graf dirigit. Un graf és un conjunt d'elements (nodes), en el nostre cas de tipus enters, connectats per arestes. Donats dos nodes A i B, diem que A està connectat a B si existeix una aresta que va des del node A fins al node B. Un exemple en el qual es pot utilitzar grafs per modelar la realitat serien les xarxes socials tals com *twitter*. On els nodes representen els diferents usuaris. Un node (usuari) està connectat a un altre node (usuari), si el primer "segueix" al segon.
Per facilitar la seva comprensió, us proporcionem un exemple d'un graf dirigit amb 5 nodes i les arestes que connecten els nodes:



La implementació d'aquest graf utilitzant el tipus de dades **tGraph** quedaria com es mostra a la figura següent:



El conjunt d'arestes es representa com un vector, on cada posició correspon a un node i les connexions (arestes) a nodes adjacents es representa com un punter des del node a una llista de nodes. Per exemple el node 1, està

connectat amb els nodes 2 i 3. Si un node no té connexions, per exemple el node 4, llavors el punter a la llista de nodes adjacents té el valor NULL.

i) A partir d'aquesta explicació, completa la definició del TAD **tGraph** utilitzant punters:

```
const
        NUM_VERTEX : integer := 1000;
end const
type
        tNode = record
                elem:  integer;
                next : pointer to tNode;
        end record

        tGraph = array [NUM_VERTEX] of pointer to tNode;
end type
```

i) Aquest nou TAD oferirà diverses operacions, entre elles et demanem que implementis l'operació que donats dos nodes A i B, retorna cert si estan connectats per una aresta (des de A fins a B o des de B fins a A) o retorna fals en cas contrari.

function connected (**in** g: tGraph, **in** A: **integer, in** B: **integer**): **boolean**
Pre: { 0 < A ≤ NUM_VERTEX i 0 < B ≤ NUM_VERTEX }

```
var
        l: list(integer);
        found: boolean;
end var

        l := g[A]^.next;
        found := false;
        while ( l ≠ NULL) and not found do
                if l^.elem = B then
                        found := true;
                else
                        l := l^.next;
                end if
        end while
        if not found then
                l := g[B]^.next;
```

```
            found := false;
            while ( l ≠ NULL) and not found do
                if l^.elem = B then
                    found := true;
                else
                    l := l^.next;
                end if
            end while
        end if

        return found;

    end function
```