Algoritmos y Estructuras de Datos II – 9 de diciembre de 2020

Examen Final Teórico-Práctico

Alumno: ..... Email: .....

Siempre se debe explicar la solución. Una respuesta correcta no es suficiente si no viene acompañada de una justificación lo más clara y completa posible. Los algoritmos no deben escribirse utilizando código c o de bajo nivel, sino el código de la materia y evitando la utilización innecesaria de punteros. La no observación de estas recomendaciones resta puntaje.

1. (Backtracking) No es posible correr una carrera de 800 vueltas sin reemplazar cada tanto las cubiertas (las ruedas) del auto. Como los mecánicos trabajan en equipo, cuando se cambian las cubiertas se reemplazan simultáneamente las cuatro. Reemplazar el set de cuatro cubiertas insume un tiempo T fijo, totalmente independiente de cuál sea la calidad de las cubiertas involucradas. Hay diferentes sets de cubiertas: algunas permiten mayor velocidad que otras, y algunas tienen mayor vida útil que otras, es decir, permiten realizar un mayor número de vueltas. Sabiendo que se cuenta con n sets de cubiertas, que $t_1, t_2, \ldots, t_n$ son los **tiempos por vuelta** que pueden obtenerse con cada uno de ellos, y que $v_1, v_2, \ldots, v_n$ es la vida útil medida en **cantidad de vueltas** de cada uno de ellos, se pide obtener un algoritmo que devuelva el tiempo de carrera mínimo cuando la carrera consta de m vueltas.

   Antes de dar la solución, especificá con tus palabras qué calcula la función recursiva que resolverá el problema, detallando el rol de los argumentos y la llamada principal.

2. (Programación dinámica) Escribí un algoritmo que utilice Programación Dinámica para resolver el ejercicio del punto anterior.

   (a) ¿Qué dimensiones tiene la tabla que el algoritmo debe llenar?
   (b) ¿En qué orden se llena la misma?
   (c) ¿Se podría llenar de otra forma? En caso afirmativo indique cuál.

3. (Comprensión de algoritmos) Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los siguientes incisos.

   (a) ¿Qué hace?
   (b) ¿Cómo lo hace?
   (c) El orden del algoritmo, analizando los distintos casos posibles.
   (d) Proponer nombres más adecuados para los identificadores (de variables y procedimientos).

```
proc p(a: array[1..n] of nat)
    var d: nat
    for i:= 1 to n do
        d:= i
        for j:= i+1 to n do
            if a[j] < a[d] then d:= j fi
        od
        swap(a, i, d)
    od
end proc
```

```
fun f(a: array[1..n] of nat) ret b : array[1..n] of nat
    var d: nat
    for i:= 1 to n do b[i] := i od
    for i:= 1 to n do
        d:= i
        for j:= i+1 to n do
            if a[b[j]] < a[b[d]] then d:= j fi
        od
        swap(b, i, d)
    od
end fun
```

4. Considere la siguiente especificación del tipo Listas de algún tipo T.

```
spec List of T where

constructors
    fun empty() ret l : List of T
    {- crea una lista vacía. -}

    proc addl (in e : T, in/out l : List of T)
    {- agrega el elemento e al comienzo de la lista l. -}

destroy
    proc destroy (in/out l : List of T)
    {- Libera memoria en caso que sea necesario. -}
```

operations

fun is_empty(l : List **of** T) **ret** b : bool
{- *Devuelve* True *si* l *es vacía.* -}

fun head(l : List **of** T) **ret** e : T
{- *Devuelve el primer elemento de la lista* l -}
{- **PRE:** not is_empty(l) -}

proc tail(**in**/**out** l : List **of** T)
{- *Elimina el primer elemento de la lista* l -}
{- **PRE:** not is_empty(l) -}

proc addr (**in**/**out** l : List **of** T,**in** e : T)
{- *agrega el elemento* e *al final de la lista* l. -}

fun length(l : List **of** T) **ret** n : nat
{- *Devuelve la cantidad de elementos de la lista* l -}

proc concat(**in**/**out** l : List **of** T,**in** l0 : List **of** T)
{- *Agrega al final de* l *todos los elementos de* l0 *en el mismo orden.*-}

fun index(l : List **of** T,n : nat) **ret** e : T
{- *Devuelve el n-ésimo elemento de la lista* l -}
{- **PRE:** length(l) > n -}

proc take(**in**/**out** l : List **of** T,**in** n : nat)
{- *Deja en* l *sólo los primeros* n *elementos, eliminando el resto* -}

proc drop(**in**/**out** l : List **of** T,**in** n : nat)
{- *Elimina los primeros* n *elementos de* l -}

fun copy_list(l1 : List **of** T) **ret** l2 : List **of** T
{- *Copia todos los elementos de* l1 *en la nueva lista* l2 -}

(a) A partir de la siguiente implementación de listas mediante punteros, implemente las operaciones copy_list, tail y concat.

```
implement List of T where

type Node of T = tuple
                   elem : T
                   next : pointer to (Node of T)
                 end tuple

type List of T = pointer to (Node of T)

fun empty() ret l : List of T
    l := null
end fun


proc addl (in e : T, in/out l : List of T)
  var p : pointer to (Node of T)
  alloc(p)
  p->elem := e
  p->next := l
  l := p
end proc
```

(b) Implemente una función que reciba una lista de enteros y decida si está ordenado de menor a mayor. Dicha función debe usar el tipo **abstracto** lista, sin importar cuál es su implementación.

5. (Para alumnos libres) Sea T un árbol (no necesariamente binario) y supongamos que deseamos encontrar la hoja que se encuentra más cerca de la raíz. ¿Cuáles son las distintas maneras de recorrer T? ¿Cuál de ellas elegirías para encontrar esa hoja y por qué?