Contents

Organizacion de la materia

Asistencia: no se toma.

Parciales: 2 y un recuperatorio

Promocion:

- 15 o mas en los parciales
- analogo en lab

Lenguaje de la materia

Variables

var a1,. . . , an : < tipo>

Comentario

{- Esto es un comentario -}

Procedimiento

Definicion

Encapsula un bloque de codigo con su respectiva declaracion de variables.

proc nombre(**<in|out|in/out>** p1: T1, . . .,**<in|out|in/out>** pn: Tn)
<declaraciones de variables>
<sentencias>
end proc

Observaciones

- $p1, \ldots, pn$ son nombres de variables.
- $T1, \ldots, Tn$ son sus respectivos tipos
- in: el parametro es de entrada (se puede leer pero no escribir)
- out: el parametro es de salida (podes escribir pero no leer)
- in/out: el parametro es de entrada/salida (podes leer y escribir)

Funcion

Definicion

Son como los procedimientos pero todos los parametros son **in** y devuelven algo

fun nombre(p1: T1, . . .,pn: Tn) **ret** r : T
<declaraciones de variables>
<sentencias>
end fun

Observaciones

- No tenemos sentencia "return", se devuelve lo asignado a la variable declarada como **ret**
- Las llamadas a funciones no son sentencias si no expresiones
- No tiene efectos colaterales en el estado

Observaciones funciones y procedimientos

- Las variables declaradas dentro de funciones y procedimientos no existen fuera de estas.
- Las funcs y procs pueden llamarse entre si y a si mismas.
- No importa el orden en que se declaran. Un proc puede llamar a otro definido más adelante
- No se pueden definir procs o funs adentro de otros procs o funs

Tipos nativos

Tipos basicos

- bool: true y false
- int: enteros
- nat: naturales
- real: reales
- char: caracteres (‘a’, ‘j’)
- string: secuencias de caracteres

Se permiten usar constantes como infinito o -infinito

Tipos estructurados

- array: arreglos
- pointer: punteros

Arreglos

Declaracion

var : **array**[N1..M1] . . . [Nk..Mk] **of** T

Donde

- N1, M1, . . . , Nk, Mk son numeros
- T es el tipo de los elementos del arreglo
- M-N+1 es el tamaño del arreglo

Acceso

a[i1]. . . [ik]

Asignacion

a[i1]. . . [ik] := E

Sentencias

skip

La sentencia que no hace nada

Asignacion

v := E

Donde v es una variable y E una expresion

No existe la asignacion multiple

Llamada a proc

nombreproc(v1, . . . , vn)

Condicional

```
if B then
  S1
else
  S2
fi
```

Donde B es una expresion booleana y S1, S2 son sentencias

No existe if multi-guarda

Repeticion

```
while B do
  S
od
```

En donde B es una expresion booleana y S es una sentencia

for to

```
for i := N to M do
  S
od
```

Donde N y M son expresiones de tipo int y S es sentencia

for downto

```
for i := N downto M do
  S
od
```

Donde N y M son expresiones de tipo int y S es sentencia

Analisis de Algoritmos

Ordenacion por seleccion

Idea

1) Seleccionar el menor elemento
2) Intercambiarlo con el primer elemento de la parte no ordenada
3) Repetir hasta que el array esté ordenado

Invariante

- El arreglo resultante es una permutacion del original
- El segmento inicial a[0,i) del arreglo esta ordenado
- Dicho segmento contiene los elementos minimos del arreglo

Pseudocodigo

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

```
proc swap(in/out a: array[1..n] of T, in i,j: nat)
  var tmp: T
  tmp = a[i]
  a[i] = a[j]
  a[j] = tmp
end proc
```

Complejidad

	Comp.	Swap
Mejor:	$O(n^2)$	$O(1)$
Promedio:	$O(n^2)$	$O(n)$
Peor:	$O(n^2)$	$O(n)$

Cantidad de operaciones

Bucle for

for i := N **to** M **do** C(k) **od**
= ops(C(n) + ops(C(n+1) + ... + ops(C(m)

Comando if

$$ops(\textbf{if } \mathrm{B} \textbf{ then } \mathrm{S1} \textbf{ else } \mathrm{S2} \textbf{ fi}) = \begin{cases} ops(B) + ops(S1) & \text{caso b = V} \\ ops(B) + ops(S2) & \text{caso b = F} \end{cases}$$

Asignacion

$$ops(x := e) \begin{cases} ops(e) + 1 & \text{contando asignacion/modificaciones de memoria} \\ ops(e) & \text{en caso contrario} \end{cases}$$

Expresiones

$$ops(e < f) \begin{cases} ops(e) + ops(f) + 1 & \text{contando comparaciones} \\ ops(e) + ops(f) & \text{en caso contrario} \end{cases}$$

AÑADIR SUMATORIAS UTILES

Ordenacion por Insercion

Idea

Empezar desde la posicion 2:

1. Ver si el elemento anterior al puntero es mayor
2. Si es mayor hacer un swap
3. Repetir hasta que (1) no se cumpla
4. Incrementar puntero

Invariante

- El arreglo a es una permutacion del original
- a[1, i] **sin celda j** está ordenado
- a[j,i] está ordenado

Pseudocodigo

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

Complejidad

	Comp.	Swap
Mejor:	$O(n)$	$O(1)$
Promedio:	$O(n^2)$	$O(n^2)$
Peor:	$O(n^2)$	$O(n^2)$

Merge Sort

Idea

Si la estructura tiene más de 2 elementos:
  Dividir en "mitades"
  Usar merge en la primera mitad
  Usar merge en la segunda mitad
  Intercalar las mitades ordenadas
Sino:
  si la cantidad de componentes es 2
  Comparar e intercambiar

Pseudocodigo

```
proc merge_sort_rec(in/out a: array[1..n] of T, in lft, rgt: nat)
  var mid: nat
  if rgt > lft then
    mid:= (rgt + lft) / 2
    merge_sort_rec(a, lft, mid)
    merge_sort_rec(a, mid+1, rgt)
    merge(a, lft, mid, rgt)
  fi
end proc

proc merge_sort(in/out a: array[1..n] of T)
  merge_sort_rec(a, 1, n)
end proc
```

```
proc merge(in/out a: array[1..n] of T, in lft, mid, rgt: nat)
  var tmp: array[1..n] of T
  var j, k: nat
  for i := lft to mid do
    tmp[i]:= a[i]
  od
  j:= lft
  k:= mid+1
  for i := lft to rgt do
    if j ≤ mid ∧ (k > rgt ∨ tmp[j] ≤ a[k]) then
      a[i]:= tmp[j]
      j:= j+1
    else
      a[i]:= a[k]
      k:= k+1
    fi
  od
end proc
```

Quicksort

Idea

1) Elegir un elemento arbitrario como pivot
2) Usar el pivot para partir el array
3) Aplicar quicksort recursivamente a la particion izquierda
4) Aplicar quicksort recursivamente a la particion derecha

Pseudocodigo

```
proc quick_sort_rec(in/out a: array[1..n] of T, in lft, rgt: nat)
  var ppiv: nat
  if rgt > lft →
    partition(a, lft, rgt, ppiv)
    quick_sort_rec(a, lft, ppiv-1)
    quick_sort_rec(a, ppiv+1, rgt)
  fi
end proc

proc quick_sort(in/out a: array[1..n] of T)
  quick_sort_rec(a,1,n)
end proc
```

```
proc partition(in/out a: array[1..n] of T, in lft, rgt: nat, out ppiv: nat)
  var i,j: nat
  ppiv := lft
  i:= lft+1
  j:= rgt
  while i ≤ j do
    if a[i] ≤ a[ppiv] then
      i:= i+1
    else if a[j] ≥ a[ppiv] then
      j:= j-1
    else if a[i] > a[ppiv] ∧ a[j] < a[ppiv] then
      swap(a, i, j)
      i:= i+1
      j:= j-1
    fi
  od
  swap(a, ppiv, j)
  ppiv:= j
end proc
```

Recurrencias y jerarquia de funciones

Algoritmo divide y venceras

Caracteristicas

- Hay una solucion para los casos sencillos
- Para los casos complejos se divide o descompone el problema en subproblemas
  - Cada subproblema es de igual naturaleza que el original
  - Cada subproblema es una fraccion del original
  - Se resuelven los subproblemas apelando al mismo algoritmo
- Se combinan esas soluciones para obtener una solucion del original

Pseudocodigo

```
fun DyV(x) ret y
  if x suficientemente pequeño o simple then y:= ad_hoc(x)
  else descomponer x en x_1, x_2, ..., x_a
    for i := 1 to a do
      y_i := DyV(x_i )
    od
    combinar y_1, y_2, ..., y_a para obtener la solucion y de x
  fi
end fun
```

Normalmente los x_i son fracciones de x: $|x_i| = \frac{|x|}{b}$ Para algun b fijo mayor que 1

Conteo a: numero de llamadas recursivas a DyV b: relacion entre el tamaño de x y el de x_i, satisface $|x_i| = \frac{|x|}{b}$ k: el orden de descomponer y combinar es n^k c: constante que representa el costo de la funcion ad_hoc g(n): es el costo de los procesos de descomposicion y combinacion

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

Orden de t(n) Si t(n) es no decreciente y g(n) es del orden de n^k, entonces

$$\text{t(n) es del orden de } \begin{cases} n^{log_b a} & \text{si } a > b^k \\ n^k logn & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

Busqueda Binaria

Algoritmo

```
fun binary_search_rec(a: array[1..n] of T, x:T, lft, rgt: nat) ret i : nat
  var mid: nat
  if lft > rgt then i:= 0
    lft ≤ rgt then
      mid:= (lft+rgt) / 2
      if x < a[mid] then i:= binary_search_rec(a, x, lft, mid-1)
        x = a[mid] then i:= mid
        x > a[mid] then i:= binary_search_rec(a, x, mid+1, rgt)
      fi
  fi
end fun

fun binary_search(a: array[1..n] of T, x: T) ret i : nat
  i:= binary_search_rec(a, x, 1, n)
end fun
```

Comparar ordenes de algoritmos

Notacion

Escribimos $f(n) $

Tecnicas de resolucion de problemas

Algoritmos voraces

Idea

- Normalmente se trata de algoritmos que resuelven problemas de optimizacion.

- Intentan construir la solucion optima buscada paso a paso
- Eligen en cada paso el componente de la solucion que parece mas apropiado
- No revisan elecciones ya realizadas
- No todos los problemas admiten solucion voraz

Ingredientes

- Los candidatos se van clasificando en 3: los aun no considerados, los incorporados a la solucion parcial y los descartados
- una manera de saber si los candidatos ya incorporados completan una solucion del problema
- Una funcion que comprueba si un candidato es factible de formar parte de la solucion
- Una funcion que selecciona de entre los candidatos no considerados el mas promisorio
- En cada paso se utiliza la funcion de seleccion para elegir cual candidato considerar
- Se chequea que el candidato considerado sea factible para incorporarlo a la solucion
- Se repiten los pasos anteriores hasta que la coleccion de candidatos elegidos sea una solucion

Esquema

```
fun voraz(C: Set of "Candidato") ret S : "Solucion a construir"
  S:= "solucion vacia"
  while S "no es solucion" do
    c:= "Seleccionar" de C
    elim(C, c)
    if "agregar c a S es factible" then
      "agregar c a S"
    fi
  od
end fun
```

Recorrida de grafos

Definicion

Recorrer un grafo significa procesar los vertices de manera tal que:

- Todos los vertices sean procesados
- Que ningun vertice sea procesado mas de una vez

Depth first search

Definicion

Primero recorre el arbol en profundidad

Pre-order

Se recorre primero la raiz, luego el sub-arbol izquierdo y luego el sub-arbol derecho

In-order

Se recorre primer el sub-arbol izquierdo, luego la raiz y luego el sub-arbol derecho

Pos-order

Se recorre primero sub-arbol izquierdo, luego el sub-arbol derecho y por ultimo la raiz.

Formas alternativas

Las anteriores formas tambien se pueden realizar cambiando el orden del subarbol que se recorre primero.

Por ejemplo, en pre-order-der-izq se recorre primero la raiz, el subarbol derecho y luego el subarbol izquierdo

Breadth first search

Definicion

Recorre el arbol en ancho.

Arboles finitarios

Definicion

Son arboles en los que cada nodo tiene una cantidad finita (pero posiblemente variable) de hijos

Recorrida

Una recorrida in-order deja de tener sentido ya que se dificulta saber en que orden debe visitarse el elemento de la raiz.

Las recorridas DFS y BFS siguen teniendo sentido