Algoritmos básicos

Algoritmos y Estructuras de Datos II, DC, UBA.

1. Introducción

En este apunte presentamos un conjunto de algoritmos básicos que sirven como útiles a la hora de resolver los ejercicios de algoritmia de la materia (dividir y conquistar y ordenamiento) y también como ejemplos adicionales de cómo presentar algoritmos.

No se presenta en el apunte la justificacion de correctitud ni de las cotas de complejidad provistas, aunque sí se asegura que son correctas. Para aprovecharlo mejor, se recomienda leer y entender la idea de los algoritmos y justificar los órdenes de complejidad. Los elementos para hacerlo están presentes en las clases teóricas de la materia.

2. Interfaces

Para expresar todos los órdenes de complejidad se utiliza la convención $n = \text{tam}(A)$. Para simplificar, la notación $a \leq x \leq b$ es abreviatura de $a \leq x \wedge x \leq b$, y de forma similar para otras combinaciones con $<$ estricto.

Swap(**in/out** x: `nat`, **in/out** y: `nat`)
Pre $\{x_0 = x \wedge y_0 = y\}$
Pos $\{y = x_0 \wedge x = y_0\}$
Compl $\mathcal{O}(1)$

PrimeroMayorA(**in** A: `arreglo(nat)`, **in** x: `nat`) $\to$ res:nat
Pre $\{\text{ordenado?}(A)\}$
Pos $\{0 \leq res \leq \text{tam}(A) \wedge_{\text{L}} (res > 0 \Rightarrow_{\text{L}} A[res-1] \leq x) \wedge (res \leq \text{tam}(A) - 1 \Rightarrow_{\text{L}} A[res] > x)\}$
Compl $\mathcal{O}(\log n)$

ÚltimoMenorA(**in** A: `arreglo(nat)`, **in** x: `nat`) $\to$ res:nat
Pre $\{\text{ordenado?}(A)\}$
Pos $\{-1 \leq res < \text{tam}(A) \wedge_{\text{L}} (res \geq 0 \Rightarrow_{\text{L}} A[res] < x) \wedge (res < \text{tam}(A) - 1 \Rightarrow_{\text{L}} A[res+1] \geq x)\}$
Compl $\mathcal{O}(\log n)$

PosMáxima(**in** A: `arreglo(nat)`) $\to$ res:nat
Pre $\{(\exists res\text{:nat})\ 0 \leq res < \text{tam}(A) \wedge_{\text{L}} \forall j : \text{nat}(0 \leq j < res \Rightarrow_{\text{L}} A[j] < A[j+1]) \wedge (res \leq j < \text{tam}(A) - 1 \Rightarrow_{\text{L}} A[j] > A[j+1])\}$
Pos $\{0 \leq res < \text{tam}(A) \wedge_{\text{L}} \forall j : \text{nat}(0 \leq j < res \Rightarrow_{\text{L}} A[j] < A[j+1]) \wedge (res \leq j < \text{tam}(A) - 1 \Rightarrow_{\text{L}} A[j] > A[j+1])\}$
Compl $\mathcal{O}(\log n)$

BubbleSort(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n^2)$

InsertionSort(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n^2)$

SelectionSort(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n^2)$

QuickSort(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n^2)$

Merge(**out** A: `arreglo(nat)`, **in** B: `arreglo(nat)`, **in** C: `arreglo(nat)`)
Pre $\{\text{ordenado?}(B) \wedge \text{ordenado?}(C)\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A,\text{concatenar}(B,C))\}$
Compl $\mathcal{O}(n)$

MergeSort(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n \log n)$

MaxHeapify(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{esMaxHeap}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n)$

HeapSort(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n \log n)$

CountingSort(**in/out** A: `arreglo(nat)`)
Pre $\{A_0 = A\}$
Pos $\{\text{ordenado?}(A) \wedge \text{esPermutación}(A, A_0)\}$
Compl $\mathcal{O}(n + \text{máx}(A))$

InsertarOrdenado(**in/out** A: `arreglo(nat)`, **in** x: `nat`)
Pre $\{\text{ordenado?}(A) \wedge A_0 = A\}$
Pos $\{(\exists B : \text{arreglo(nat)})\ \text{tam}(B) = 1 \wedge_{\text{L}} B[0] = x \wedge \text{esPermutación}(A, \text{concatenar}(A_0,B)) \wedge \text{ordenado?}(res)\}$
Compl $\mathcal{O}(n)$

dónde,

ordenado?(A)	$\equiv$	$(\forall i : \text{nat})0 \leq i < \text{tam}(A) - 1 \Rightarrow_{\text{L}} A[i] \leq A[i+1]$
esMaxHeap?(A)	$\equiv$	$(\forall i : \text{nat})(0 \leq i \wedge 2 \times i + 1 < \text{tam}(A)) \Rightarrow_{\text{L}} A[i] \geq A[2 \times i + 1]) \wedge$ $(0 \leq i \wedge 2 \times i + 2 < \text{tam}(A) \Rightarrow_{\text{L}} A[i] \geq A[2 \times i + 2])$
esPermutación(A, B)	$\equiv$	$\text{tam}(A) = \text{tam}(B) \wedge (\exists P : \text{arreglo(nat)})\,\text{tam}(P) = \text{tam}(A) \wedge \text{esPermutación1N}(P) \wedge$ $(\forall i : \text{nat})0 \leq i < \text{tam}(A) \Rightarrow_{\text{L}} A[i] = B[P[i]]$
esPermutación1N(A)	$\equiv$	$(\forall i : \text{nat})0 \leq i < \text{tam}(A)\ \Rightarrow\ (\exists j : \text{nat})0 \leq j < \text{tam}(A) \wedge_{\text{L}} A[j] = i$
concatenar(A, B)	$\equiv$	función del mundo de TADs que concatena dos arreglos.

3. Algoritmos

Las funciones de arreglo CrearArreglo, Subarreglo y Copiar las suponemos dadas. Subarreglo da una vista con aliasing de una porción del arreglo, es decir, Subarreglo$(A, i, j)[k]$ es la misma variable que $A[i + k - 1]$ y tam(Subarreglo(A, i, j)) es $j - i + 1$ (siempre restringido a $0 \leq i \leq j < \text{tam}(A)$). CrearArreglo y Copiar toman tiempo $\mathcal{O}(n)$ dónde n es el tamaño del arreglo y Subarreglo toma tiempo $\mathcal{O}(1)$.

En varios de los casos, especialmente las funciones recursivas, el pseudocódigo dado no responde a una implementación especialmente eficiente en términos de constantes y ahorro de memoria, sino a una forma pedagógica de ilustrar el algoritmo cumpliendo con la cota prometida de complejidad temporal.

Swap(**in/out** x: `nat`, **in/out** y: `nat`)
nat $aux \leftarrow x$
$x \leftarrow y$
$y \leftarrow aux$

```
PrimeroMayorA(in A: arreglo(nat), in x: nat) → res:nat
nat i ← 0, j ← tam(A)
while i < j − 1 do
  nat m ← (i + j)/2
  if A[m] > x then
    j ← m
  else
    i ← m
  end if
end while
if A[i] > x then
  res ← i
else
  res ← j
end if
```

```
ÚltimoMenorA(in A: arreglo(nat), in x: nat) → res:nat
nat i ← −1, j ← tam(A) − 1
while i < j − 1 do
  nat m ← (i + j)/2
  if A[m] < x then
    i ← m
  else
    j ← m
  end if
end while
if A[j] < x then
  res ← j
else
  res ← i
end if
```

```
PosMáxima(in A: arreglo(nat)) → res:nat
nat i ← 0, j ← tam(A) − 1
while i < j − 1 do
  nat m ← (i + j)/2
  if A[m − 1] < A[m] then
    i ← m
  else
    j ← m
  end if
end while
if A[i] > A[i + 1] then
  res ← i
else
  res ← i + 1
end if
```

```
BubbleSort(in/out A: arreglo(nat))
for i ← 0 to tam(A) − 1 do
  for j ← i + 1 to tam(A) − 1 do
    if A[i] > A[j] then
      Swap(A[i], A[j])
    end if
  end for
end for
```

```
InsertionSort(in/out A: arreglo(nat))
for i ← 1 to tam(A) − 1 do
  nat j ← i, x ← A[i]
  while j > 0 ∧ A[j − 1] > x do
    A[j] ← A[j − 1], j ← j − 1
  end while
  A[j] ← x
end for
```

```
SelectionSort(in/out A: arreglo(nat))
for i ← 0 to tam(A) − 1 do
  nat m ← i
  for j ← i + 1 to tam(A) − 1 do
    if A[m] > A[j] then
      m ← j
    end if
  end for
  Swap(A[i], A[m])
end for
```

```
QuickSort(in/out A: arreglo(nat))
nat p ← número al azar entre 0 y tam(A) − 1
Swap(A[p], A[0])
nat i = 0, j = tam(A) − 1
while i < j do
  if A[i] < A[i + 1] then
    Swap(A[i], A[i + 1]), i ← i + 1
  else
    Swap(A[j], A[i + 1]), j ← j − 1
  end if
end while
if i > 0 then
  QuickSort(Subarreglo(A, 0, i − 1))
end if
if i < tam(A) − 1 then
  QuickSort(Subarreglo(A, i + 1, tam(A) − 1))
end if
```

```
Merge(out A: arreglo(nat), in B: arreglo(nat), in C: arreglo(nat))
nat i_B ← 0, i_C ← 0
A ← CrearArreglo(tam(B) + tam(C))
for i_A ← 0 to tam(A) − 1 do
  if i_B < tam(B) ∧ (i_C ≥ tam(C) ∨ B[i_B] < C[i_C]) then
    A[i_A] ← B[i_B], i_B ← i_B + 1
  else
    A[i_A] ← C[i_C], i_C ← i_C + 1
  end if
end for
```

```
MergeSort(in/out A: arreglo(nat))
if tam(A) > 1 then
  nat m ← tam(A)/2
  arreglo(nat) B ← Copiar(Subarreglo(A, 0, m − 1)), C ← Copiar(Subarreglo(A, m, tam(A) − 1))
  MergeSort(B)
  MergeSort(C)
  Merge(A, B, C)
end if
```

```
MaxHeapify(in/out A: arreglo(nat))
nat i ← tam(A) − 1
while i ≥ 0 do
  nat j ← i
  while 2j + 1 < tam(A) do
    nat k ← 2j + 1
    if 2j + 2 < tam(A) ∧ A[2j + 2] > A[k] then
      k ← 2j + 2
    end if
    if A[j] < A[k] then
      Swap(A[k], A[j]), j ← k
    else
      j ← tam(A)
    end if
  end while
  i ← i − 1
end while
```

```
HeapSort(in/out A: arreglo(nat))
MaxHeapify(A)
nat i ← tam(A) − 1
while i ≥ 0 do
  Swap(A[i], A[0])
  nat j ← 0
  while 2j + 1 < i do
    nat k ← 2j + 1
    if 2j + 2 < i ∧ A[2j + 2] > A[k] then
      k ← 2j + 2
    end if
    if A[j] < A[k] then
      Swap(A[k], A[j]), j ← k
    else
      j ← tam(A)
    end if
  end while
  i ← i − 1
end while
```

```
CountingSort(in/out A: arreglo(nat))
nat m ← 0
for i ← 0 to tam(A) − 1 do
  m ← máx(m, A[i])
end for
arreglo(nat) C ← CrearArreglo(m + 1)
for i ← 0 to tam(A) − 1 do
  C[A[i]] ← C[A[i]] + 1
end for
nat i ← 0
for x ← 0 to m do
  for t ← 0 to C[x] − 1 do
    A[i] ← x, i ← i + 1
  end for
end for
```

```
InsertarOrdenado(in/out A: arreglo(nat), in x: nat)
arreglo(nat) B ← CrearArreglo(1), res ← CrearArreglo(tam(A) + 1)
B[0] ← x
Merge(A, A, B)
```