Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Programación imperativa - clase 4

Algoritmos de ordenamiento

Ordenamiento de un arreglo

- Tenemos un arreglo de un tipo T con una relación de orden ($\leq$).
- Queremos modificar el arreglo para que sus elementos queden en orden creciente.
- Adoptamos como estrategia que modificamos el arreglo permutando elementos, para garantizar que nuestro algoritmo no pierde ningún elemento del arreglo.

La especificación

problema sort<T> $(a : [\mathrm{T}], n : \mathrm{Int})\{$
 requiere $1 \leq n == |a|$;
 modifica a;
 asegura mismos$(a, \mathrm{pre}(a))$ $\wedge$ $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$;
$\}$

aux cuenta$(x : \mathrm{T}, a : [\mathrm{T}]) : \mathrm{Int} = |[y \mid y \leftarrow a,\ y == x]|$;

aux mismos$(a, b : [\mathrm{T}]) : \mathrm{Bool} = |a| == |b| \wedge$
 $(\forall x \leftarrow a)$ cuenta$(x, a) ==$ *cuenta*(x, b);

T tiene que ser un tipo con una relación de orden $\leq$ definida

El algoritmo Upsort

- Ordenamos el arreglo de derecha a izquierda.
- El segmento desordenado va desde el principio hasta una posición que vamos a llamar *actual*.
- Comenzamos con $actual = n - 1$
- Mientras $actual > 0$:
  - Encontrar el mayor elemento del segmento todavía no ordenado,
  - intercambiarlo con el de la posición *actual*, y
  - decrementar *actual*.

El programa

```
void upsort (int a[], int n) {
   int m, actual = n-1;
   while (actual > 0) {
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
   }
}
```

problema maxPos(a : [Int], $desde, hasta$: Int) $= pos$: Int{
  requiere $0 \leq desde \leq hasta \leq |a| - 1$;
  asegura $desde \leq pos \leq hasta \;\wedge\; (\forall x \leftarrow a[desde..hasta])\; x \leq a_{pos}$;
}

problema swap(x, y : Int){
  modifica x, y;
  asegura $x == \text{pre}(y) \;\wedge\; y == \text{pre}(x)$;
}

Correctitud de Upsort

```
void upsort (int a[], int n) {
```

// vale $P : 1 \leq n == |a|$ (es la precondición del problema)

```
   int m, actual = n-1;
```

// vale $P_C : a == \text{pre}(a) \wedge actual == n - 1$ (es la precondición del ciclo)

```
   while (actual > 0) {
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
   }
```

// vale Q_C : mismos$(a, \text{pre}(a)) \;\wedge\; (\forall j \leftarrow [0..n-1))\; a_j \leq a_{j+1}$
(es la poscondición del ciclo)

// vale Q : mismos$(a, \text{pre}(a)) \;\wedge\; (\forall j \leftarrow [0..n-1))\; a_j \leq a_{j+1}$
(es la poscondición del problema)

```
}
```

Como $Q_C \Rightarrow Q$, solo resta probar que el ciclo es correcto para su especificación.

Especificación del ciclo

```
void upsort (int a[], int n) {
   int m, actual = n-1;
```

// vale $P_C : a == \text{pre}(a) \wedge actual == n - 1$

```
   while (actual > 0) {
```

// invariante $I : 0 \leq actual \leq n - 1 \wedge$ mismos$(a, \text{pre}(a))$
$\wedge (\forall k \leftarrow (actual..n-1))\; a_k \leq a_{k+1}$
$\wedge\; actual < n - 1 \rightarrow (\forall x \leftarrow a[0..actual])\; x \leq a_{actual+1}$
// variante $v : actual$

```
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
   }
```

// vale Q_C : mismos$(a, \text{pre}(a)) \;\wedge\; (\forall j \leftarrow [0..n-1))\; a_j \leq a_{j+1}$

```
}
```


Correctitud del ciclo

// vale P_C
// implica I

```
   while (actual > 0) {
```

// estado E
// vale $I \wedge actual > 0$

```
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
```

// vale I
// vale $v < v@E$

```
   }
```

// vale $I \wedge \neg(actual > 0)$
// implica Q_C

1. El cuerpo del ciclo preserva el invariante
2. La función variante decrece
3. Si la función variante pasa la cota, el ciclo termina: $v \leq 0 \Rightarrow \neg(actual > 0)$
4. La precondición del ciclo implica el invariante
5. La poscondición vale al final

El Teorema del Invariante nos garantiza que si valen 1, 2, 3, 4 y 5, el ciclo termina y es correcto respecto de su especificación.

1. El cuerpo del ciclo preserva el invariante

// estado E (invariante + guarda del ciclo)
// vale $0 < actual \leq n-1 \wedge \text{mismos}(a, \text{pre}(a))$
$\wedge(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
$\wedge(actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1})$

```
m = maxPos(a,0,actual);
```

// estado E_1

Recordar especificación de MaxPos:
P_{MP} : $0 \leq desde \leq hasta \leq |a|-1$, se cumple porque $0 < atual \leq n-1$
Q_{MP} : $desde \leq pos \leq hasta \ \wedge\ (\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos}$

// vale $0 \leq m \leq actual \ \wedge\ (\forall x \leftarrow a[0..actual])\ x \leq a_m$
// vale $a == a@E \ \wedge\ actual == actual@E$
// implica $0 < actual \leq n-1 \wedge \text{mismos}(a, \text{pre}(a))$
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
// implica $(actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1})$

Justificación de los implica: *actual* y *a* no cambiaron
Lo dice el segundo vale de este estado

1. El cuerpo del ciclo preserva el invariante (cont.)

// estado E_1
// vale $0 \leq m \leq actual \ \wedge\ (\forall x \leftarrow a[0..actual])\ x \leq a_m$
// implica $0 < actual \leq n-1 \wedge \text{mismos}(a, \text{pre}(a))$
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
// implica $actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$

```
swap(a[m],a[actual]);
```

// estado E_2
// vale $a_m == (a@E_1)_{actual} \ \wedge\ a_{actual} == (a@E_1)_m$ (por poscon. de *swap*)
// vale $(\forall i \leftarrow [0..n), i \neq m, i \neq actual)\ a_i == (a@E_1)_i$ (idem)
// vale $actual == actual@E_1 \wedge m == m@E_1$
// implica $0 < actual \leq n-1$ (*actual* no se modificó)
// implica $\text{mismos}(a, \text{pre}(a))$ (el *swap* no agrega ni quita elementos)
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
($a(actual..n-1]$ no se modificó porque $m \leq actual$)
// implica $(\forall k \leftarrow (actual-1..n-1))\ a_k \leq a_{k+1}$
// implica $(\forall x \leftarrow a[0..actual])\ x \leq a_{actual}$ (del primer vale de E_1 y E_2)

// estado E_2
// implica $0 < actual \leq n-1$
// implica $\text{mismos}(a, \text{pre}(a))$
// implica $(\forall k \leftarrow (actual-1..n-1))\ a_k \leq a_{k+1}$
// implica $(\forall x \leftarrow a[0..actual])\ x \leq a_{actual}$

```
actual--;
```

// estado E_3
// vale $actual == actual@E_2 - 1 \ \wedge\ a == a@E_2$
// implica $0 \leq actual@E_2 - 1 < n-1$ (de primer vale de E_3)
// implica $0 \leq actual \leq n-1$ (reemplazando $actual@E_2 - 1$ por *actual*)
// implica $\text{mismos}(a, \text{pre}(a))$ (de segundo implica de E_2 y $a == a@E_2$)
// implica $(\forall k \leftarrow (actual@E_2 - 1..n-1))\ a_k \leq a_{k+1}$ (por E_2)
// implica $(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
(reemplazo $actual@E_2 - 1$ por *actual*)
// implica $(\forall x \leftarrow a[0..actual+1])\ x \leq a_{actual+1}$ (por E_2 + reemplazo)
// implica $(\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$
(por ser un selector más acotado)
// implica $actual < n-1 \rightarrow (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$
(pues q implica $p \rightarrow q$)

2 y 3 son triviales

2. La función variante decrece:

// estado E (invariante + guarda del ciclo)
// vale $I \wedge B$

```
m = maxPos(a,0,actual);
swap(a[m],a[actual]);
actual--;
```

// estado F
// vale $actual == actual@E - 1$
// implica $v@F == v@E - 1 < v@E$

3. Si la función variante pasa la cota, el ciclo termina:

$actual \leq 0$ es $\neg B$

4. La precondición del ciclo implica el invariante

Recordar que

- $P : 1 \leq n == |a|$
- $P_C : a == \text{pre}(a) \land actual == n - 1$
- $I : 0 \leq actual \leq n - 1 \land \text{mismos}(a, \text{pre}(a))$
  $\land(\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
  $\land actual < n-1 \to (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$

Demostración de que $P_C \Rightarrow I$:

- $1 \leq n \land actual == n - 1 \quad \Rightarrow \quad 0 \leq actual \leq n - 1$
- $a == \text{pre}(a) \quad \Rightarrow \quad \text{mismos}(a, \text{pre}(a))$
- $actual == n - 1 \quad \Rightarrow \quad (\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1}$
  porque el selector actúa sobre una lista vacía
- $actual == n - 1 \quad \Rightarrow$
  $actual < n - 1 \to (\forall x \leftarrow a[0..actual])\ x \leq a_{actual+1}$
  porque el antecedente es falso

5. La poscondición vale al final

Quiero probar que: $(\neg B \land I) \Rightarrow Q_C$

Recordemos

$$\begin{aligned}\neg B \land I \quad : \quad & 0 \leq actual \leq n-1 \land \\ & \text{mismos}(a, \text{pre}(a)) \land (\forall k \leftarrow (actual..n-1))\ a_k \leq a_{k+1} \land \\ & actual < n-1 \to (\forall x \leftarrow a[0..actual+1])\ x \leq a_{actual+1} \land \\ & actual \leq 0\end{aligned}$$

$$Q_C : \text{mismos}(a, \text{pre}(a)) \ \land \ (\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$$

Veamos que vale cada parte de Q_C:

- $\text{mismos}(a, \text{pre}(a))$: trivial porque está en I
- $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$:
  - primero observar que $actual == 0$
  - si $n == 1$, no hay nada que probar porque $[0..n-1) == []$
  - si $n > 1$
    - sabemos $(\forall k \leftarrow (0..n-1))\ a_k \leq a_{k+1}$
    - sabemos que $(\forall x \leftarrow a[0..1])\ x \leq a_1$, entonces $a_0 \leq a_1$
    - concluimos $(\forall j \leftarrow [0..n-1))\ a_j \leq a_{j+1}$

Implementación de maxPos

problema maxPos $(a : [\text{Int}], desde, hasta : \text{Int}) = pos : \text{Int}\{$
  requiere $P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1;$
  asegura $Q_{MP} : desde \leq pos \leq hasta \land$
    $(\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos};$
$\}$

```
int maxPos(const int a[], int desde, int hasta) {
      int mp = desde, i = desde;
      while (i < hasta) {
            i++;
            if (a[i] > a[mp]) mp = i;
      }
      return mp;
}
```


Correctitud de maxPos

problema maxPos $(a : [\text{Int}], desde, hasta : \text{Int}) = pos : \text{Int}\{$
  requiere $P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1;$
  asegura $Q_{MP} : desde \leq pos \leq hasta \land$
    $(\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos};$
$\}$

```
int maxPos(const int a[], int desde, int hasta) {
```
//vale $P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1$ (precondición del problema)
```
      int mp = desde, i = desde;
```
//vale $P_C : mp == i == desde$ (precondición del ciclo)
```
      while (i < hasta) {
            i++;
            if (a[i] > a[mp]) mp = i;
      }
```
//vale $Q_C : desde \leq mp \leq hasta \land (\forall x \leftarrow a[desde..hasta])\ x \leq a_{mp}$
(poscondición del ciclo)
```
      return mp;
```
//vale $Q_{MP} : desde \leq pos \leq hasta \ \land (\forall x \leftarrow a[desde..hasta])\ x \leq a_{pos}$
```
}
```

Especificación del ciclo

// vale $P_C : mp == i == desde$

```
while (i < hasta) {
```

// invariante $I : desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$
// variante $v : hasta - i$

```
        i++;
        if (a[i] > a[mp]) mp = i;
}
```

// vale $Q_C : desde \leq mp \leq hasta \wedge (\forall x \leftarrow a[desde..hasta])\ x \leq a_{mp}$

Correctitud del ciclo

// vale P_C
// implica I

```
while (i < hasta) {
```

// estado E
// vale $I \wedge i < hasta$

```
    i++;
    if (a[i] > a[mp]) mp = i;
```

// vale I
// vale $v < v@E$

```
}
```

// vale $I \wedge i \geq hasta$
// implica Q_C

1. El cuerpo del ciclo preserva el invariante
2. La función variante decrece
3. Si la función variante pasa la cota, el ciclo termina: $v \leq 0 \Rightarrow i \geq hasta$
4. La precondición del ciclo implica el invariante
5. La poscondición vale al final

El Teorema del Invariante nos garantiza que si valen 1, 2, 3, 4 y 5, el ciclo termina y es correcto con respecto a su especificación.

1. El cuerpo del ciclo preserva el invariante

Recordar que *desde*, *hasta* y *a* no cambian porque son variables de entrada que no aparecen en *local* ni *modifica*

// estado E (invariante + guarda del ciclo)
// vale $desde \leq mp \leq i < hasta \wedge (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$

```
i++;
```

// estado E_1
// vale $i == i@E + 1 \wedge mp = mp@E$
// implica $P_{if} : desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i))\ x \leq a_{mp}$

(de E, reemplazando $i@E$ por $i - 1$ y cambiando el límite del selector adecuadamente)

```
if (a[i] > a[mp]) mp = i;
```

// vale $Q_{if} : desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$

(observar que en este punto, tratamos al `if` como una sola gran instrucción que convierte P_{if} en Q_{if}. La justificación de este paso es la transformación de estados de la hoja siguiente)

// implica I

($I = Q_{if}$, en general alcanza con que Q_{if} implique I)

Especificación y correctitud del If

$P_{if} : desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i))\ x \leq a_{mp}$
$Q_{if} : desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$

// estado E_{if}
// vale $desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i))\ x \leq a_{mp}$

```
if (a[i] > a[mp]) mp = i;
```

// vale $(a_i > a_{mp@E_{if}} \wedge mp == i@E_{if} \wedge i == i@E_{if})$
$\vee (a_i \leq a_{mp@E_{if}} \wedge mp == mp@E_{if} \wedge i == i@E_{if})$
// implica $desde \leq mp \leq i \leq hasta$

(operaciones lógicas; observar que *desde*, *hasta* y *a* no pueden modificarse porque son variables de entrada que no aparecen ni en *modifica* ni en *local*; observar que $i == i@E_{if}$)

// implica $a_{mp@E_{if}} \leq a_{mp} \wedge a_i \leq a_{mp}$ (Justificar...)
// implica $(\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$
// implica Q_{if}

2. La función variante decrece

// estado E (invariante + guarda del ciclo)
// vale $desde \leq i < hasta$

```
i++;
```

// estado E_1
// vale $i == i@E + 1$
// implica $desde \leq i \leq hasta$

```
if (a[i] > a[mp]) mp = i;
```

// estado F
// vale $i == i@E_1$
// implica $i == i@E + 1$

¿Cuánto vale $v = hasta - i$ en el estado F?

$$\begin{aligned} v@F &== (hasta - i)@F \\ &== hasta - i@F \\ &== hasta - (i@E + 1) \\ &== hasta - i@E - 1 \\ &< hasta - i@E == v@E \end{aligned}$$

3, 4 y 5 son triviales

3. Si la función variante pasa la cota, el ciclo termina:

$$hasta - i \leq 0 \quad \Rightarrow \quad hasta \leq i$$

4. La precondición del ciclo implica el invariante:
Recordar que la precondición de maxPos dice

$$P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1$$

y que $desde$ y $hasta$ no cambian de valor. Entonces

$$P_C : i == desde \wedge mp == desde$$
$$\Downarrow$$
$$I : desde \leq mp \leq i \leq hasta \wedge (\forall x \leftarrow a[desde..i])\ x \leq a_{mp}$$

5. La poscondición vale al final: es fácil ver que $I \wedge i \geq hasta$ implica Q_C

Entonces el ciclo es correcto con respecto a su especificación.

Complejidad de maxPos

¿Cuántas comparaciones hacemos como máximo?

- El ciclo itera $hasta - desde$ veces, y cada iteración hace:
  - Una comparación (en la guarda),
  - una asignación (el incremento),
  - otra comparación (guarda del condicional), y
  - otra asignación (si se cumple esa guarda).
- Antes del ciclo se hacen dos asignaciones.
- El peor caso es $desde == 0$ y $hasta == |a| - 1$, por lo tanto la complejidad de maxPos es $O(|a|)$ para el peor caso.

Complejidad de Upsort

- El ciclo de Upsort itera n veces:
  - Empieza con $actual == n - 1$,
  - termina con $actual == 0$, y
  - en cada iteración decrementa $actual$ en 1.
- Cada iteración hace:
  - Una búsqueda de maxPos,
  - un swap, y
  - un decremento.
- De estos pasos, el único que no es O(1) es el primero.
- ¿Cuántos pasos hace maxPos en cada iteración? Depende del valor de $actual$!
- En general, hace $O(hasta - desde + 1) = O(actual + 1)$ pasos.
- Luego, el total de pasos de upSort es $O(n + (n-1) + \ldots + 2) = O(\frac{n(n+1)}{2} - 1) = O(n^2)$.

Cota inferior de la complejidad

- **Teorema.** Todo algoritmo de ordenamiento basado en comparaciones y permutaciones de elementos toma al menos $\log_2 n!$ pasos para ordenar n elementos, en el peor caso.
- **Observación.** $\log_2 n! = \Omega(n \log_2 n)$.
- El algoritmo upSort es $O(n^2) > O(n \log_2 n)$.
- Existen algoritmos de ordenamiento de arreglos con complejidad $O(n \log_2 n)$: quicksort (en el caso promedio) mergesort, heapsort.