Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2011

Departamento de Computación - FCEyN - UBA

Programación imperativa - clase 4

Algoritmos de ordenamiento

Ordenamiento de un arreglo

Tenemos un arreglo de elementos de un tipo T.
Queremos modificar el arreglo

- para que sus elementos queden en orden creciente
- vamos a hacerlo permutando elementos

Si pre(a) == $[2, 1, 2, 1]$, debo terminar con $a == [1, 1, 2, 2]$.

La especificación

Usamos $\leq$ para denotar una relación de orden entre elementos de T.

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

aux cuenta(x : T, a : [T]) : Int = $|[y \mid y \in a,\ y == x]|$;

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

El algoritmo Upsort

- ordenamos de derecha a izquierda
- el segmento a ordenar va desde el principio hasta la posición que vamos a llamar *actual*
- comenzamos con *actual* $= n - 1$
- mientras *actual* > 0
  - encontrar el mayor elemento del segmento
  - intercambiarlo con el de la posición *actual*
  - decrementar *actual*

Ejemplo de Upsort

9	5	2	7	1	2
2	5	2	7	1	9
2	5	2	1	7	9
2	1	2	5	7	9
2	1	2	5	7	9
1	2	2	5	7	9

Especificación del ciclo

arreglo `a[]` de dimensión `n`, variable `actual`

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

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

$B : \ actual > 0$

invariante $I : 0 \leq actual \leq n-1 \land$
$\text{mismos}(a, \text{pre}(a)) \land$
$(\forall k \in (actual..n-1))\ a_k \leq a_{k+1} \land$
$actual < n-1 \rightarrow (\forall x \in a[0..actual])\ x \leq a_{actual+1}$

variante $v : actual$

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 $desde \leq pos \leq hasta \ \land \ (\forall x \in a[desde..hasta])\ x \leq a_{pos}$;
}

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

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

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

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

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

```
}
```

Como $Q_C \equiv Q$, lo que queda es 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 == pre(a) ∧ actual == n − 1

   while (actual > 0) {

// invariante I : 0 ≤ actual ≤ n − 1 ∧ mismos(a, pre(a))
//              ∧ (∀k ∈ (actual..n − 1)) a_k ≤ a_{k+1}
//              ∧ actual < n − 1 → (∀x ∈ a[0..actual]) x ≤ a_{actual+1}
// variante v : actual

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

// vale Q_C : mismos(a, pre(a)) ∧ (∀j ∈ [0..n − 1)) a_j ≤ a_{j+1}

}
```


Correctitud del ciclo

```
// vale P_C
// implica I
   while (actual > 0) {
// estado E
// vale I ∧ actual > 0
         m = maxPos(a,0,actual);
         swap(a[m],a[actual]);
         actual--;
// vale I
// vale v < v@E
   }
// vale I ∧ ¬(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 $1, 2, 3, 4, 5$ el ciclo termina y es correcto con respecto a su especificación.

1. El cuerpo del ciclo preserva el invariante

```
// estado E (invariante + guarda del ciclo)
// vale 0 < actual ≤ n − 1 ∧ mismos(a, pre(a))
//         ∧(∀k ∈ (actual..n − 1)) a_k ≤ a_{k+1}
//         ∧(actual < n − 1 → (∀x ∈ a[0..actual]) x ≤ 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 $0 < atual \leq n - 1$
$Q_{MP} : desde \leq pos \leq hasta \ \wedge \ (\forall x \in a[desde..hasta])\ x \leq a_{pos}$)

```
// vale 0 ≤ m ≤ actual ∧ (∀x ∈ a[0..actual]) x ≤ a_m
// vale a == a@E ∧ actual == actual@E
// implica 0 < actual ≤ n − 1 ∧ mismos(a, pre(a))
// implica (∀k ∈ (actual..n − 1)) a_k ≤ a_{k+1}
// implica (actual < n − 1 → (∀x ∈ a[0..actual]) x ≤ 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 ≤ m ≤ actual ∧ (∀x ∈ a[0..actual]) x ≤ a_m
// implica 0 < actual ≤ n − 1 ∧ mismos(a, pre(a))
// implica (∀k ∈ (actual..n − 1)) a_k ≤ a_{k+1}
// implica actual < n − 1 → (∀x ∈ a[0..actual]) x ≤ a_{actual+1}

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

// estado E_2
// vale a_m == (a@E_1)_actual ∧ a_actual == (a@E_1)_m (por poscon. de swap)
// vale (∀i ∈ [0..n), i ≠ m, i ≠ actual) a_i == (a@E_1)_i          (idem)
// vale actual == actual@E_1 ∧ m == m@E_1
// implica 0 < actual ≤ n − 1                    (actual no se modificó)
// implica mismos(a, pre(a))      (el swap no agrega ni quita elementos)
// implica (∀k ∈ (actual..n − 1)) a_k ≤ a_{k+1}
                    (a(actual..n − 1] no se modificó porque m ≤ actual)
// implica (∀k ∈ (actual − 1..n − 1)) a_k ≤ a_{k+1}
```

(del tercer vale de E_2: $m == m@E_1$ y $actual == actual@E_1$;
del primer vale de y último implica de E_1: $(a@E_1)_m \leq (a@E_1)_{actual+1}$;
del primer y segundo vale de E_2: $a_{actual} \leq a_{actual+1}$)

```
// implica (∀x ∈ a[0..actual]) x ≤ a_actual     (del primer vale de E_1 y E_2)
```

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

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

```
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 mismos$(a, \text{pre}(a))$ (de segundo implica de E_2 y $a == a@E_2$)
// implica $(\forall k \in (actual@E_2 - 1..n-1))\ a_k \leq a_{k+1}$ (por E_2)
// implica $(\forall k \in (actual..n-1))\ a_k \leq a_{k+1}$
(reemplazo $actual@E_2 - 1$ por $actual$)
// implica $(\forall x \in a[0..actual+1])\ x \leq a_{actual+1}$ (por E_2 + reemplazo)
// implica $(\forall x \in a[0..actual])\ x \leq a_{actual+1}$
(por ser un selector más acotado)
// implica $actual < n-1 \rightarrow (\forall x \in a[0..actual])\ x \leq a_{actual+1}$
(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

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

Demostración de que $P_C \Rightarrow I$:

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

5. La poscondición vale al final

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

$\neg B \wedge I \ : \ 0 \leq actual \leq n-1 \wedge$
$\text{mismos}(a, \text{pre}(a)) \wedge (\forall k \in (actual..n-1))\ a_k \leq a_{k+1} \wedge$
$actual < n-1 \rightarrow (\forall x \in a[0..actual+1])\ x \leq a_{actual+1} \wedge$
$actual \leq 0$

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

Veamos que vale cada parte de Q_C:

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

Implementación de maxPos

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

```
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 : [Int], $desde$, $hasta$: Int) = pos : Int{
  requiere $P_{MP} : 0 \leq desde \leq hasta \leq |a| - 1$;
  asegura $Q_{MP} : desde \leq pos \leq hasta \wedge$
    $(\forall x \in a[desde..hasta])\ x \leq a_{pos}$;
  }

```
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 $Q_C : desde \leq mp \leq hasta \wedge (\forall x \in a[desde..hasta])\ x \leq a_{mp}$
(poscondición del ciclo)
```
      return mp;
```
//vale $Q_{MP} : desde \leq pos \leq hasta\ \wedge (\forall x \in a[desde..hasta])\ x \leq a_{pos}$
(poscondición del problema)
```
}
```


Especificación del ciclo

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

```
      while (i < hasta) {
```
// invariante $I : desde \leq mp \leq i \leq hasta \wedge (\forall x \in a[desde..i])\ x \leq a_{mp}$
// variante $v : hasta - i$
```
            i++;
            if (a[i] > a[mp]) mp = i;
      }
```
// vale $Q_C : desde \leq mp \leq hasta \wedge (\forall x \in a[desde..hasta])\ x \leq a_{mp}$

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 $desde \leq mp \leq i < hasta \wedge (\forall x \in a[desde..i])\ x \leq a_{mp}$

```
i++;
```

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

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

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

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

(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

(observar que I es igual que Q_{if}, pero en general alcanzaría con que Q_{if} implique I)

Especificación y correctitud del If

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

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

```
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 $(\forall x \in a[desde..i])\ x \leq a_{mp}$
// 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

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

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
Cada iteración hace:

- una comparación (en la guarda)
- una asignación (el incremento)
- otra comparación (guarda del condicional)
- otra asignación (si se cumple esa guarda)

Antes del ciclo se hacen dos asignaciones.

Luego, la cantidad de operaciones de MaxPos es O(longitud del segmento) $= O(hasta - desde + 1)$.

MaxPos es invocada la primera vez con $desde == 0$ y $hasta == |a| - 1$, luego con segmentos sucesitamente más cortos hasta llegar a tener longitud 1,

Complejidad de Upsort

- el ciclo de Upsort itera n veces
  - empieza con $actual == n - 1$
  - termina con $actual == 0$
  - en cada iteración decrementa $actual$ en uno
- una iteración hace
  - una búsqueda de maxPos
  - un swap
  - un decremento
- de estos pasos, el único que no es O(1), constante, es el primero
- ¿cuántos pasos hace maxPos en cada iteración?
  - siempre $O(hasta - desde + 1) = O(actual + 1)$
  - en la primera hace n (busca el máximo del segmento a ordenar)
  - en la segunda hace $n - 1$ (busca el segundo mayor)
  - en la tercera hace $n - 2$
  - y así (podríamos verlo por inducción) hasta 2
- el total de pasos es
  $O(n + (n - 1) + \ldots + 2) = O(n * (n + 1)/2 - 1) = O(n^2)$
- los mejores algoritmos de ordenamiento son $O(n \log n)$

Cota inferior de complejidad tiempo para sorting

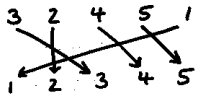
Consideremos algoritmos de ordemiento basados, exclusivamente, en la operaciones de comparar elementos y permutarlos.

Teorema: Sea cualquier método de ordenamiento (inventado o por inventarse).
Tomará al menos $\log_2 n!$ pasos para ordenar n elementos.

Demostración

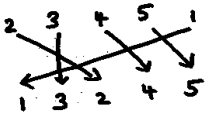
Supongamos que ordenamos esta secuencia: **3, 2, 4, 5, 1**.

El método comparará los números, y según sean los resultados de las comparaciones permutará los elementos. El efecto neto será:



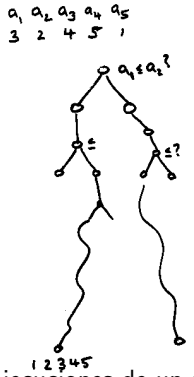
El método no puede ejecutar exactamente de la misma manera si la secuencia es otra, por ejemplo: **2, 3, 4, 5, 1**

¿por qué? Si lo hiciera, el resultado neto sería



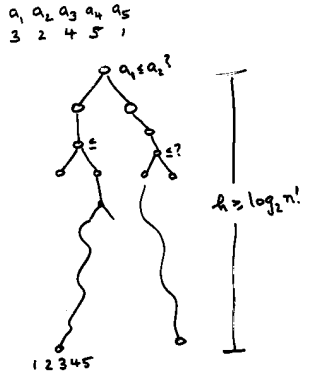
Concluímos que la ejecución debe ser una sucesión de pasos **distinta** para cada permutación posible de la secuencia de entrada.

Visualicemos todas las ejecuciones como un árbol de ejecuciones.
Cada nodo es, o bien una asignación, o bien una comparación.
El árbol se ramifica en cada comparación.
Cada camino desde la raíz hasta una hoja es una ejecución del programa.
La altura del árbol el da el peor caso de cantidad de instrucciones ejecutadas por el programa.



Si éste es el árbol de ejecuciones de un programa de ordenamiento, que ordena n elementos, para cada una de las $n!$ permutaciones de la secuencia de entrada, la ejecución bebe alcanzar una hoja diferente.

El árbol debe tener al menos $n!$ hojas.



Debe tener al menos $n!$ hojas.

Sabemos que un árbol binario con L hojas tiene altura al menos $\log_2 L$.

Por lo tanto nuestro árbol tiene altura al menos $\log_2 n!$.

Concluímos que el ordenamiento, en el peor caso, requiere al menos $\log_2 n!$ pasos.

Para poder hacer un ordenamiento más rápido necesitamos...

saber más acerca de la secuencia de entrada, de forma tal de poder ahorrarnos comparaciones entre de elementos.