Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2012

Departamento de Computación - FCEyN - UBA

Programación imperativa - clase 3

Arreglos, Búsqueda lineal, Búsqueda binaria

Arreglos

Secuencias de una cantidad fija de variables del mismo tipo.
Se declaran con un nombre, un tamaño y un tipo.

Ejemplo: int a[10]; // arreglo de 10 elementos enteros

Nos referimos a los elementos a través del nombre del arreglo y un índice, entre corchetes, que va de 0 a la dimensión menos uno.

`a[0], a[1],..., a[9]`

Una referencia a una posición fuera del rango da error (en tiempo de ejecución)

El tamaño y el tipo de un arreglo se mantienen invariantes a lo largo de la ejecución.

Arreglos versus listas

Ambos son secuencias de elementos de un tipo dado.
En el lenguaje de especificiación tratamos a ambos con secuencias por comprensión.

Longitud
Los arreglos tienen longitud fija; las listas, no.

Acceso
Los elementos de un arreglo se acceden de manera directa, e independiente de los demás.
`a[i]` accede al elemento en posición *i*-ésima del arreglo. Tiene dirección de memoria propia, y por lo tanto se le puede asignar valores.

Los elementos de una lista se acceden secuencialmente, empezando por la cabeza. Para acceder al *i*-ésimo elemento de una lista, hay que obtener *i* veces la cola y luego la cabeza.

Arreglos en C++

Los arreglos en C++ son referencias.

No hay forma de averiguar su tamaño una vez que fueron creados;
el programa tiene que encargarse de almacenarlo de alguna forma.

Al ser pasados como argumentos, en la declaración de un parámetro no se indica su tamaño.

Leyendo un arreglo

problema sumarray(a : [Int], tam : Int) = res : Int{
requiere P: $tam == |a|$;
asegura Q: $res == \sum a[0..|a|)$; }

```
int sumarray(int a[], int tam) {
    int j = 0;
    int s = 0;
    // Pc: j == 0 ∧ s == ∑ a[0..j)
    while (j < tam) {
    // invariante 0 ≤ j ≤ tam ∧ s == ∑ a[0..j)
    // variante tam − j;
        s= s + a[j];
        j++;
  }
    // Qc: s == ∑ a[0..|a|)
}
return s;
// vale Q: res == ∑ a[0..|a|);
}
```


Inicialización de arreglos

problema init (a : [Int], x : Int, n : Int){
requiere $n == |a| \wedge n > 0$;
modifica a;
asegura $todos([a[j] == x, j \in [0..n)])$;
}

```
void init (int a[], int x, int n) {
  int i = 0;
// vale Pc : i == 0
  while (i < n) {
//  invariante I: 0 ≤ i ≤ n ∧ todos([a[j] == x, j ∈ [0..i)])
 // variante v : n − i
    a[i]=x;
    i = i + 1;
  }
//  vale Qc : i == n ∧ todos([a[j] == x, j ∈ [0..i)])
}
```


Modificando un arreglo

problema ceroPorUno(a : [Int], tam : Int){
requiere $tam == |a|$;
modifica a;
asegura $a ==$ [if $i == 0$ then 1 else i | $i \leftarrow$ pre(a)[0..tam)]; }

```
void ceroPorUno(int a[], int tam) {
  int j = 0;
  while (j < tam) {
    // invariante 0 ≤ j ≤ tam ∧ a[j..tam) == pre(a)[j..tam)∧
    // a[0..j) == [if i == 0 then 1 else i | i ← pre(a)[0..j)];
    // variante tam − j;
    if (a[j] == 0) a[j] = 1;
    j++;
  }
}
```


Búsqueda lineal sobre arreglos

problema buscar (a : [Int], x : Int, n : Int) = res : Bool{
requiere $n == |a| \wedge n > 0$;
asegura $res == (x \in a)$;
}

Algoritmo de búsqueda lineal (ya lo vimos en funcional para listas)

Dada una secuencia y una condición booleana sobre sus elementos,

recorre de izqueirda a derecha la secuencia mientras no se ha encontrado un elemento que cumpla la condición booleana.

Retornar el elemento encontrado, o la posición del elemento, o la condición booleana de si lo encontramos o no.

Es el más ingenuo de los algoritmos de búsqueda. El índice va del 0 en adelante). En el peor caso tantas iteraciones como el tamaño del arreglo.

Especificación del ciclo de búsqueda lineal

Dado el arreglo a de tamaño n y el elemento x.
Buscamos la primer posición que contiene al elemento x.
Usaremos una variable i entera para recorer las posiciones de a.
El arreglo no se modifica a lo largo de las iteraciones.

invariante $I : 0 \leq i \leq n \wedge x \notin a[0..i)$

variante $v : n - i$

$B : \ i < n \ \wedge \ a[i]! = x$

$Pc : i == 0$

$Qc : i < n \leftrightarrow x \in a[0..n)$

Código del algoritmo de búsqueda lineal

```
bool buscar (int a[], int x, int n) {
  int i = 0;
  while (i < n && a[i] != x) {
    i = i + 1;
  }
  return i < n;
}
```

Notar que la guarda es una conjunción que nunca se indefine.

Correctitud del ciclo de búsqueda lineal

```
bool buscar(int a[], int x, int n) {
  int i = 0;
// vale Pc : i == 0
  while (i < n && a[i] != x) {
  // invariante I : 0 ≤ i ≤ n ∧ x ∉ a[0..i)
  // variante v : n − i
    i = i + 1;
}
// vale Qc : i < n ↔ x ∈ a[0..n)
return i < n;
}
```

El Teorema del Invariante nos garantiza que si valen las 5 condiciones, entonces el ciclo termina y es correcto con respecto a su especificación.

Correctitud del ciclo de búsqueda lineal

```
// vale Pc : i == 0
// implica I
while (i < n && a[i] != x) {
  // invariante I : 0 ≤ i ≤ n ∧ x ∉ a[0..i)
  // variante v : n − i
  // estado E
  // vale I ∧ i < n ∧ a[i] ≠ x
  i = i+1;
  // estado F
  // vale I
  // vale v < v@E
}
  // vale I ∧ ¬(i < n ∧ a[i] ≠ x)
  // implica Qc : i < n ↔ x ∈ a[0..n)
```

1. El cuerpo del ciclo preserva el invariante

Recordar que el invariante es $I : 0 \leq i \leq n \land x \notin a[0..i)$
y la guarda es $B : i < n \land a[i] \neq x$

// estado E	(invariante + guarda del ciclo)
// vale $I \land B$	
// implica $0 \leq i < n$	(juntando $0 \leq i \leq n$ y $i < n$)
// implica $x \notin a[0..i]$	(juntando $x \notin a[0..i)$ y $a[i] \neq x$)
// implica $x \notin a[0..i+1)$	(propiedad de secuencias)
`i = i + 1;`	
// estado F	
// vale $i == i@E + 1$	
// implica $1 \leq i@E + 1 < n + 1$	(está en E y sumando 1 en cada término)
// implica $0 \leq i \leq n$	(usando que $i == i@E + 1$ y propiedad de $\mathbb{Z}$)
// implica $x \notin a[0..i@E + 1)$	(está en E; a no puede cambiar)
// implica $x \notin a[0..i)$	(usando que $i == i@E + 1$)
// implica I	(se reestablece el invariante)

2. La función variante decrece

// estado E (invariante + guarda del ciclo)
// vale $I \land B$
`i = i + 1;`
// estado F
// vale $i == i@E + 1$

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

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

5. La poscondición vale al final

Quiero probar que $I \land \neg B$ implica Q_c

$$\overbrace{0 \leq i \leq n}^{1} \land \overbrace{x \notin a[0..i)}^{2} \land \overbrace{(i \geq n \lor x == a[i])}^{5}$$

implica

$$Qc : \underbrace{i < n}_{3} \leftrightarrow \underbrace{x \in a[0..n)}_{4}$$

Demostración:
Supongamos $i \geq n$ (i.e. 3 es falso).
Por 1, $i == n$.
Por 2, tenemos $x \notin a[0..n)$.
Luego 4 también es falso.

Supongamos $i < n$ (i.e. 3 es verdadero).
Por 5, $x == a[i]$.
De 1 concluimos que 4 es verdadero.

3 y 4 son triviales

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

$v \leq 0$ implica $\neg B$

Recordemos $v : n - i$; $\neg B : (i \geq n \lor x == a[i])$

$n - i \leq 0$ entonces $n \leq i$ por lo tanto $\neg B$

[4.] La precondición del ciclo implica el invariante

Pc implica I

Recordemos $Pc : i == 0$; $I : 0 \leq i \leq n \land x \notin a[0..i)$

$i == 0$ entonces $0 \leq i \leq n \land x \notin a[0..i)$

Concluímos que el ciclo es correcto con respecto a su especificación.

buscar es correcto respecto de la especificación

problema buscar $(a : [\text{Int}], x : \text{Int}, n : \text{Int}) = res : \text{Bool}\{$
requiere $n == |a| \wedge n > 0$;
asegura $res == (x \in a)$;
$\}$

```
bool buscar(int a[], int x, int n) {
  int i = 0;
// vale Pc : i == 0
  while (i < n && a[i] != x)
// invariante I : 0 ≤ i ≤ n ∧ x ∉ a[0..i)
// variante v : n − i
    i = i + 1;
}
// estado H
// vale Qc : i < n ↔ x ∈ a[0..n)
return i < n;
// vale res == (i < n)@H ∧ i = i@H
// implica res == x ∈ a[0..n)    (esta es la poscondición del problema)
}
```


Complejidad de un algoritmo

La complejidad temporal $T(n)$ de un algoritmo es la máxima cantidad de operaciones que va a ejecutar. Se la define como función del tamaño de la entrada n.

Para dar cotas superiores de la complejidad se usa la notación asimptótica O grande.

$T(n)$ es $O(f(n))$ sii $\exists c, n_0\ \forall n \geq n_0\ T(n) \leq c \cdot f(n)$

Es decir, $T(n)$ es del orden de $f(n)$ cuando $T(n)$ es a lo sumo una constante por $f(n)$, excepto para valores pequeños de n.

Nos interesa ver cómo crece la complejidad cuando crece la estructura.

Complejidad de la búsqueda lineal

¿Cuándo hace la máxima cantidad de operaciones?

Cuando el elemento no está en la secuencia.

¿Cuántas operaciones son?

El agoritmo ejecuta tantas iteraciones como tamaño de la secuencia. La cantidad de operaciones es la cantidad de iteraciones por una constante.

Notemos que en nuestro programa cada iteración hace 4 operaciones (dos comparaciones, una conjunción y el incremento del índice).

El algoritmo de búsqueda lineal tiene complejidad $O(n)$, es decir, tiene complejidad lineal.

Mejorando la búsqueda

Supongamos un diccionario y queremos buscar una palabra x.

Abrimos el diccionario a la mitad

Si x está en allí, terminamos.

Si x es menor (según el orden de diccionario) que las palabras de la posición actual, x no puede estar en la parte derecha.

Si x es mayor (según el orden de diccionario) que las palabras de la posición actual, x no puede estar en la parte izquierda.

Seguimos buscando sólo en la parte izquierda o derecha (según sea el caso).

¿Cuándo termina?

Búsqueda binaria

Supongamos que en lugar de un diccionario tenemos un arreglo ordenado de números enteros. Buscamos un número entero.

Inicialmente el segmento de búsqueda es todo el arreglo.

Si el número que buscamos es menor que el primer elemento o mayor que el último, entonces seguro que no está.

Revisaremos sólo algunas posiciones del arreglo: en cada iteración descartaremos la mitad del segmento actual de búsqueda.

Termina cuando encuentro el elmento en la posición que reviso,
o bien, cuando el segmento de búsqueda tiene menos que tres elementos
(no tiene medio). Es decir, tiene uno o dos elementos.

El problema con solución búsqueda binaria

problema buscarBin $(a : [\text{Int}], x : \text{Int}, n : \text{Int}) = res : \text{Bool}\{$
  requiere $|a| == n \wedge n > 0$;
  requiere $(\forall j \in [0..n-1))\ a[j] \leq a[j+1]$;
  asegura $res == (x \in a)$;
$\}$

Demostraremos que el algoritmo de búsqueda binaria es correcto respecto de esta especificicación.

Difiere de la especificación que usamos para la búsqueda lineal en que la precondición exige que el arreglo esté ordenado,

Especificación del ciclo del algoritmo de búsqueda binaria

invariante $I : 0 \leq i \leq d < n\ \wedge\ a[i] \leq x \leq a[d]$

variante $v : d - i - 1$

$B : d > i + 1$

$Pc :\ n > 0 \wedge |a| == n \wedge\ (\forall j \in [0..n-1))\ a[j] \leq a[j+1] \wedge$
$i == 0 \wedge d == n - 1\ \wedge\ a[i] \leq x \leq a[d]$

$Qc : 0 \leq i < n \wedge 0 \leq d < n \wedge$
$x \in a \leftrightarrow (a[i] == x \vee a[d] == x)$

Código de la búsqueda binaria

```
bool buscarBin(int a[], int x, int n) {
   int  i = 0;
   int  d = n - 1;
   int m;
   bool res;

   if (x < a[i] || x > a[d]) res = false;
   else {
        while (d > i + 1) {
                m = (i + d) / 2;
                if (x == a[m])  { i = m;   d= i;}
                else if (x < a[m])  d = m;
                else i = m;
        }
        res = (a[i] == x || a[d] == x);
    }
    return res;
}
```

Código del programa y especificación del ciclo

```
bool buscarBin(int a[], int x, int n) {
   int i = 0, d = n - 1;  int m;  bool res;
   if (x < a[i] || x > a[d]) res = false;
   else {
```

// vale $Pc : n > 0 \wedge |a| == n \wedge\ (\forall j \in [0..n-1))\ a[j] \leq a[j+1] \wedge$
$i == 0 \wedge d == n-1\ \wedge\ a[i] \leq x \leq a[d]$

```
      while (d > i + 1) {
```

// invariante $I : 0 \leq i \leq d < n\ \wedge\ a[i] \leq x \leq a[d]$
// variante $v : d - i - 1$

```
         m = (i + d) / 2;
         if (x == a[m])  { i = m; d = i; }
         else if (x < a[m]) d = m;
         else i = m;
      }
```

// vale $Qc : 0 \leq i < n \wedge 0 \leq d < n \wedge\ \ x \in a \leftrightarrow (a[i] == x \vee a[d] == x)$

```
      res = (a[i] == x || a[d] == x);
   }
   return res;
}
```


Correctitud del ciclo

// vale $Pc : n > 0 \wedge |a| == n \wedge\ (\forall j \in [0..n-1))\ a[j] \leq a[j+1] \wedge$
$i == 0 \wedge d == n-1\ \wedge\ a[i] \leq x \leq a[d]$
// implica I

```
while (d > i + 1) {
```

// estado E
// vale $I \wedge B$

```
   m = (i + d) / 2;
   if (x == a[m]) { i = m; d = i; }
   else if (x < a[m]) d = m;
   else i = m;
```

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

```
}
```

// vale $I \wedge \neg B$
// implica $Qc : 0 \leq i < n \wedge 0 \leq d < n \wedge\ x \in a \leftrightarrow (a[i] == x \vee a[d] == x)$

1. El cuerpo del ciclo preserva el invariante

// estado E (invariante + guarda del ciclo)
// vale $0 \leq i \wedge i + 1 < d < n \wedge a[i] \leq x \leq a[d]$

```
m = (i + d) / 2;
```

// estado F
// vale $m = (i + d)@E\ div\ 2 \wedge i = i@E \wedge d = d@E$
// implica $0 \leq i < m < d < n$

```
if (x == a[m]) { i =m;  d = i; }
else if (x < a[m]) d = m;
else i = m;
```

// vale $m == m@F$
// vale $(x == a[m@F] \wedge i == d == m@F) \vee$
$(x < a[m@F] \wedge d == m@F \wedge i == i@F) \vee$
$(x > a[m@F] \wedge i == m@F \wedge d == d@F)$

Falta ver que esto último implica el invariante

- $0 \leq i \leq d < n$: sale del estado F
- $a[i] \leq x \leq a[d]$:
  - caso $x == a[m]$: es trivial pues $a[i] == a[m] == a[d] == x$
  - caso $x < a[m]$: tenemos $a[i] \leq x < a[m] == a[d]$
  - caso $x > a[m]$: tenemos $a[i] == a[m] < x \leq a[d]$

2. La función variante decrece

// estado E (invariante + guarda del ciclo)
// vale $0 \leq i \wedge i + 1 < d < n \wedge a[i] \leq x \leq a[d]$
// implica $d - i - 1 > 0$

```
m = (i + d) / 2;
```

// estado F
// vale $m == (i + d)\ div\ 2 \wedge i == i@E \wedge d == d@E$
// implica $0 \leq i < m < d < n$

```
if (x == a[m]) { i = d; d = i;};
else if (x < a[m]) d = m;
else i = m;
```

// vale $m == m@F$
// vale $(x == a[m@F] \wedge i == d == m@F) \vee$
$(x < a[m@F] \wedge d == m@F \wedge i == i@F) \vee$
$(x > a[m@F] \wedge i == m@F \wedge d == d@F)$

¿Cuánto vale $v = d - i - 1$?

- caso $x == a[m]$: es trivial pues $v = -1$
- caso $x < a[m]$: d decrece pero i queda igual
- caso $x > a[m]$: i crece pero d queda igual

3 y 4 son triviales

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

$$d - i - 1 \leq 0 \quad \text{implica} \quad d \leq i + 1$$

4. La precondición del ciclo implica el invariante

$$Pc : i == 0 \land d == n - 1 \land a[i] \leq x \leq a[d]$$

implica

$$I : 0 \leq i \leq d < n \land a[i] \leq x \leq a[d]$$

5. La poscondición vale al final

Quiero probar que $I \land \neg B$ implica Qc

$$\overbrace{0 \leq i \leq d < n}^{1} \quad \land \quad \overbrace{a[i] \leq x \leq a[d]}^{2} \quad \land \quad \overbrace{d \leq i + 1}^{3}$$

implica

$$\underbrace{0 \leq i < n \land 0 \leq d < n}_{4} \quad \land \quad \underbrace{x \in a}_{5} \leftrightarrow \underbrace{(a[i] == x \lor a[d] == x)}_{6}$$

Demostración:
Por 1, resulta 4 verdadero.

Supongamos 6 verdadero. De 1 concluimos que 5 es verdadero.

Supongamos $a[i] \neq x \land a[d] \neq x$ (i.e. 6 es falso): $a[j] == x$ para algún j

- $i < j < d$: contradice 3, $d \leq i + 1$
- $j < i$: gracias al orden, $x == a[j] < a[i]$; contradice 2, $x \geq a[i]$
- $j > d$: gracias al orden, $x == a[j] > a[d]$; contradice 2, $x \leq a[d]$

Entonces el ciclo es correcto con respecto a su especificación.

Correctitud del algoritmo de búsqueda binaria

```
bool buscarBin(int a[], int x, int n) {
   int i = 0, d = n - 1;  int m;  bool res;
   if (x < a[i] || x > a[d]) res = false;
   else {
// vale Pc : n > 0 ∧ |a| == n ∧ (∀j ∈ [0..n − 1)) a[j] ≤ a[j + 1]∧
            i == 0 ∧ d == n − 1 ∧ a[i] ≤ x ≤ a[d]
      while (d > i + 1) {
// invariante I : 0 ≤ i ≤ d < n ∧ a[i] ≤ x ≤ a[d]
// variante v : d − i − 1
         m = (i + d) / 2;
         if (x == a[m])  { i = m; d = i; }
         else if (x < a[m]) d = m;
         else i = m;
      }
// vale Qc : 0 ≤ i < n ∧ 0 ≤ d < n ∧  x ∈ a ↔ (a[i] == x ∨ a[d] == x)
      res = (a[i] == x || a[d] == x);
   }
   return res;
// vale (x < a[0] ∨ x > a[n − 1] ∧ res == false)∨
   (a[i] == x ∨ a[d] == x ∧ res == true)∨
   (a[i] ≠ x ∧ a[d] ≠ x∧ == false)
//implicaQ : res == (x ∈ a)
}
```

Usamos que el arreglo está ordenado para implicar Q.
si $x < a[0] \lor x > a[n-1]$, *res* es falso (ok, por el orden).
si $a[i] == x \lor a[d] == x$, *res* es verdadero (trivial).
si $a[i] \neq x \land a[d] \neq x$: supongamos que $a[j] == x$ para algún j

$i < j < d$: no puede ser porque $d == i$ ó $d == i + 1$

$j < i$: gracias al orden, $x == a[j] < a[i]$; contradice $x \geq a[i]$

$j > d$: gracias al orden, $x == a[j] > a[d]$; contradice $x \leq a[d]$

Complejidad del algoritmo de búsqueda binaria

¿Cuántas iteraciones hace el algoritmo como máximo?
En cada iteración nos quedamos con la mitad del espacio de búsqueda.

Termina cuando el segmento de búsqueda tiene longitud 1 o 2.

número de iteración	longitud del espacio de búsqueda
1	n
2	$n/2$
3	$(n/2)/2 = n/2^2$
4	$(n/2^2)/2 = n/2^3$
$\vdots$	$\vdots$
t	$n/2^{t-1}$

Para llegar al espacio de búsqueda de tamaño 1 hacemos t iteraciones

$$1 = n/2^{t-1} \quad \text{entonces} \quad 2^{t-1} = n \quad \text{entonces} \quad t = 1 + \log_2 n.$$

Luego, la complejidad de la búsqueda binaria es $O(\log_2 n)$.
Esta cota superior es mucho mejor que la búsqueda lineal, que es $O(n)$.

Variantes del problema de búsqueda

problema buscarBin $(a : [\mathrm{Int}], x : \mathrm{Int}, n : \mathrm{Int}) = res : T\{$

requiere $|a| == n \land n > 0$;

requiere $(\forall j \in [0..n-1))\ a[j] \leq a[j+1]$;

asegura .....

$\}$

donde *res* puede ser de tipo

1. Bool, indicando si el valor buscado está o no en el arreglo.

   asegura $res == (x \in a)$;

2. Int, indicando una posición del elemento o la dimensión si no está. Se pueder requerir que sea la *menor* posición.

   asegura $(res == n \land (x \notin a)) \ \lor \ (a[res] == x \land x \notin a[0..res))$;

   El algoritmo de búsqueda lineal lo asegura automáticamente. El de búsqueda binaria no. Hay distintas formas de extender el algoritmo.

3. Int (o el tipo del arreglo)cuando buscamos un elemento que cumpla una condición booleana $C(z)$ apropiada: para todo x, y, si $C(x) \land C(y)$ entonces cada z tal que $x \leq z \leq y$ debe cumplir $C(z)$. Además debemos disponer de un valor del tipo del arreglo para indicar el caso de que el elemento buscado no fue encontrado.

Conclusiones

Vimos dos algoritmos de búsqueda para problemas relacionados

problema buscar $(a : [\mathrm{Int}], x : \mathrm{Int}, n : \mathrm{Int}) = res : \mathrm{Bool}$

requiere $|a| == n > 0$;

asegura $res == (x \in a)$;

problema buscarBin $(a : [\mathrm{Int}], x : \mathrm{Int}, n : \mathrm{Int}) = res : \mathrm{Bool}$

requiere $|a| == n > 0$;

requiere $(\forall j \in [0..n-1))\ a[j] \leq a[j+1]$;

asegura $res == (x \in a)$;

La búsqueda binaria es mucho más eficiente que la búsqueda lineal, porque en el peor caso la busqueda lineal accede a todos los elementos de la secuencia. mientras que la búsqueda binaria solamente accede a una cantidad logarítmica del tamaño de la secuencia.
Las cotas superiores de complejidad temporal son también cotas inferiores del peor caso.

Moraleja: en general, más propiedades en los datos de entrada permiten dar algoritmos más eficientes.