Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

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.
  1. **Ejemplo:** int a[10];
- Nos referimos a los elementos a través del nombre del arreglo y un índice entre corchetes.
  1. `a[0]`, `a[1]`,..., `a[9]`

Arreglos

- Los arreglos se almacenan en espacios contiguos de memoria, lo cual tiene importantes consecuencias sobre su comportamiento:
  1. Solamente hay valores en las posiciones válidas de 0 a la dimensión menos 1.
  2. Una referencia a una posición fuera del rango da error (en tiempo de ejecución).
  3. El tamaño de un arreglo se mantiene constante

Arreglos y listas

- Ambos representan secuencias de elementos de un tipo.
  1. Longitud: Los arreglos tienen longitud fija; las listas, no.
  2. Acceso: Los elementos de un arreglo pueden accederse en forma independiente en $O(1)$. Para acceder al i-ésimo elemento de un arreglo, simplemente se usa el índice.
  3. 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++ se implementan por medio de una referencia al primer elemento.
- No hay forma de averiguar su tamaño una vez que fueron creados. El programa tiene que encargarse de almacenarlo en una variable entera.
- 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 $n == |a| \land n > 0$;
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

- El algoritmo de búsqueda lineal es un procedimiento para buscar un elemento en una secuencia de elementos del mismo tipo.
- Dado un conjunto totalmente ordenado (secuencia) y una condición booleana sobre los elementos ...
  1. ...recorrer secuencialmente el conjunto mientras hay elementos y no se cumple la condición booleana; ...
  2. ... cuando se encuentra un elemento que cumple la condición o se termina la secuencia, retornar el elemento encontrado, o la posición del elemento, o un booleano que informe si lo encontramos o no.

Búsqueda lineal sobre arreglos

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

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


Especificación del ciclo

```
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;
}
```


1. El cuerpo del ciclo preserva el invariante

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

```
// estado E                                  (invariante + guarda del ciclo)
// vale I ∧ B
// implica 0 ≤ i < n                        (juntando 0 ≤ i ≤ n y i < n)
// implica x ∉ a[0..i]                      (juntando x ∉ a[0..i) y a[i] ≠ x)
// implica x ∉ a[0..i + 1)                  (propiedad de secuencias)
  i = i + 1;
// estado F
// vale i == i@E + 1
// implica 1 ≤ i@E + 1 < n + 1 (está en E y sumando 1 en cada término)
// implica 0 ≤ i ≤ n          (usando que i == i@E + 1 y propiedad de ℤ)
// implica x ∉ a[0..i@E + 1)                (está en E; a no puede cambiar)
// implica x ∉ 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 $I \wedge B$
`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 $I \wedge \neg B$ implica Q_c

$$\overbrace{0 \le i \le n}^{1} \quad \wedge \quad \overbrace{x \notin a[0..i)}^{2} \quad \wedge \quad \overbrace{(i \ge n \vee x == a[i])}^{5}$$

implica

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

Demostración:
Supongamos $i \ge n$ (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:

$v \le 0$ implica $\neg B$

Recordemos $v : n - i$; $\quad \neg B : (i \ge n \vee x == a[i])$

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

[4.] La precondición del ciclo implica el invariante

Pc implica I

Recordemos $Pc : i == 0$; $\quad I : 0 \le i \le n \wedge x \notin a[0..i)$

$i == 0$ entonces $0 \le i \le n \wedge x \notin a[0..i)$

Concluímos que el ciclo es correcto respecto de su especificación.

`buscar` es correcto respecto de la especificación

```
bool buscar(int a[], int x, int n) {
  int i = 0;
// vale Pc : i == 0
  while (i < n && a[i] != x)
    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 computacional

- **Definición.** La función de complejidad de un algoritmo es una función $f : \mathbb{N} \to \mathbb{N}$ tal que $f(n)$ es la cantidad de operaciones que realiza el algoritmo en el peor caso cuando toma una entrada de tamaño n.
- Algunas observaciones:
  1. Medimos la cantidad de operaciones en lugar del tiempo total (por qué?).
  2. Nos interesa el peor caso del algoritmo (por qué??).
  3. La complejidad se mide en función del tamaño de la entrada y no de la entrada particular (por qué???).

Notación "O grande"

- **Definición.** Si f y g son dos funciones, decimos que $f \in O(g)$ si existen $\alpha \in \mathbb{R}$ y $n_0 \in \mathbb{N}$ tales que

$$f(n) \leq \alpha\, g(n) \qquad \text{para todo } n \geq n_0.$$

- Intuitivamente, $f \in O(g)$ si $g(n)$ "le gana" a $f(n)$ para valores grandes de n.
- Ejemplos:
  - Si $f(n) = n$ y $g(n) = n^2$, entonces $f \in O(g)$.
  - Si $f(n) = n^2$ y $g(n) = n$, entonces $f \notin O(g)$.
  - Si $f(n) = 100n$ y $g(n) = n^2$, entonces $f \in O(g)$.
  - Si $f(n) = 4n^2$ y $g(n) = 2n^2$, entonces $f \in O(g)$ (y a la inversa).

Complejidad computacional

- Utilizamos la notación "O grande" para especificar la función de complejidad f de los algoritmos (por qué?!?).
  - Si $f \in O(n)$, decimos que el algoritmo es lineal.
  - Si $f \in O(n^2)$, decimos que el algoritmo es cuadrático.
  - Si $f \in O(n^3)$, decimos que el algoritmo es cúbico.
  - En general, si $f \in O(n^k)$, decimos que el algoritmo es polinomial.
  - Si $f \in O(2^n)$ o similar, decimos que el algoritmo es exponencial.
- El algoritmo de búsqueda que presentamos antes tiene función de complejidad $f \in O(n)$. Decimos también "el algoritmo es $O(n)$".
  - ¿Cuál es el peor caso del algoritmo de búsqueda secuencial?
  - ¿Se puede hacer mejor?

Búsqueda binaria

- Suponemos que el arreglo está ordenado, y que el elemento a buscar está "dentro del rango" dado por el primer y el último elemento.

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

Código del programa y especificación del ciclo

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

// vale $Pc : i == 0 \land d == n - 1$

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

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

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

// vale $Qc : 0 \leq i == d - 1 < n - 1 \land a[i] \leq x < a[d]$

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


1. El cuerpo del ciclo preserva el invariante

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

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

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

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

// vale $m == m@F$
// vale $(x < a[m@F] \land d == m@F \land i == i@F) \lor$
$(x \geq a[m@F] \land i == m@F \land d == d@F)$

Falta ver que esto último implica el invariante

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

2. La función variante decrece

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

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

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

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

// vale $m == m@F$
// vale $(x < a[m@F] \land d == m@F \land i == i@F) \lor$
$(x \geq a[m@F] \land i == m@F \land d == d@F)$

¿Cuánto vale $v = d - i - 1$?

- caso $x < a[m]$: d decrece pero i queda igual
- caso $x \geq a[m]$: 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

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

5. La poscondición vale al final

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

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

implica

$$\underbrace{0 \leq i == d - 1 < n - 1}_{4} \quad \wedge \quad \underbrace{x \in a \leftrightarrow (a[i] == x)}_{5}$$

1. Por 1 y 3, resulta 4 verdadero.
2. La precondición que asegura que el arreglo está ordenado junto con 2 implican 5.

Entonces el ciclo es correcto con respecto a su especificación.

Complejidad del algoritmo de búsqueda binaria

¿Cuántas comparaciones hacemos como máximo?
En cada iteración nos quedamos con la mitad del espacio de búsqueda

Paramos 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 $O(\log_2 n)$.

Búsqueda binaria

- ¿Es bueno un algoritmo con complejidad logarítmica?

n	Búsqueda Lineal	Búsqueda Binaria
10	10	4
10^2	100	7
10^6	$1,000,000$	21
$2{,}3 \times 10^7$	$23,000,000$	25
7×10^9	$7,000,000,000$	33 (!)

- Sí! Un algoritmo con este orden es muy eficiente.

Conclusiones

Vimos dos algoritmos de búsqueda para problemas relacionados

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

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

La búsqueda binaria es mejor que la búsqueda lineal porque la complejidad tiempo para el peor caso $O(\log_2 n)$ es menor $O(n)$

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