Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2014

Departamento de Computación - FCEyN - UBA

Algoritmos - clase 10

Introducción a la complejidad computacional y algoritmo de búsqueda binaria

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.
2. Nos interesa el peor caso del algoritmo.
3. La complejidad se mide en función del tamaño de la entrada y no de la entrada particular.

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.

- 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 lineal tiene complejidad $O(n)$. Decimos también "el algoritmo es $O(n)$". ¿Se puede dar un algoritmos de búesuqeda más eficiente?

Búsqueda binaria

Problema: Determinar si un arreglo contiene un elemento dado.

1. Conocemos el algoritmo de búsqueda lineal, que tiene una complejidad de ... $O(n)$.
2. ¿Se puede hacer mejor?

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

Especificación del ciclo

$Pc : i == 0 \wedge d == n-1$

$Qc : x \in a \leftrightarrow (a[i] == x \vee a[d] == x)$

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

$B : i + 1 < d$

$v : d - i - 1$, cota 0

Código del programa

```
bool buscarBin(int a[], int x, int n) {
   int i = 0,  d = n - 1;
   while ( i+1 < d ) {
      int m = (i + d) / 2;
      if ( a[m] <= x )
         i = m;
      else
         d = m;
   }
   return (a[i] == x || a[d] == x);
}
```


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 ∧ d == n − 1
   while ( i+1 < d ) {
// invariante I : 0 ≤ i ≤ d < n  ∧  a[i] ≤ x ≤ a[d]
// variante  v : d − i − 1
      int m = (i + d) / 2;
      if ( a[m] <= x )
         i = m;
      else
         d = m;
   }
// vale Qc : x ∈ a ↔ (a[i] == x ∨ a[d] == x)
   return (a[i] == x || 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 ( a[m] <= x ) i = m; else d = m;
```

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

Falta ver que esto último implica el invariante

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

2. La función variante decrece

// estado E (invariante junto con 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 ( a[m] <= x ) i = m; else d = m;
```

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

¿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

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

implica

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

5. La poscondición vale al final

Sabiendo que el arreglo está ordenado queremos probar que

$$I \wedge \neg B \text{ implica } Qc$$

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

implica

$$\overbrace{x \in a}^{4} \leftrightarrow \overbrace{(a[i] == x \vee a[d] == x)}^{5}$$

Por 1 y 3, resulta $(i == d - 1 \vee i == d)$

- Supongamos 4. Usando 2 y $(i == d - 1 \vee i == d)$, implica 5.
- Supongamos 5. Directamente implica 4.

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 longitud 1 hacemos t iteraciones
$1 = n/2^{t-1}$ entonces $2^{t-1} = n$ entonces $t = 1 + \log_2 n$.
Luego, la complejidad de la bú squeda 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.

Búsqueda lineal versus búsqueda binaria

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 $a[0] \leq x \leq a[n-1]$;
  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 que $O(n)$.
- En general, más propiedades en los datos de entrada permiten dar algoritmos más eficientes.

Búsqueda lineal versus búsqueda binaria

Consideremos la variante del problema de búsqueda que retorna la posición en la secuencia donde se encuentra el elemento x, y retorna un valor negativo en caso de que x no esté en la secuencia.

- Búsqueda lineal retorna la primera posición donde está el elemento x buscado.
- En general la busqueda binaria retorna alguna de las posiciones donde está x, pero no necesariamente la primera.
- La versión del algoritmo de búsqueda binaria que dimos encuentra la primera posición del elemento buscado (si es que existe). Simplemente hay que adaptar el algoritmo para que retorne un número entero. Y demostrar que el algoritmo encuentra la menor posición, si es que existe.

Búsqueda binaria retornando la menor posición

problema buscarBinMP $(a : [\text{Int}], x : \text{Int}, n : \text{Int}) = pos : \text{Int}$
requiere $|a| == n > 1$;
requiere $(\forall j \in [0..n-1))\ a[j] \leq a[j+1]$;
requiere $a[0] \leq x \leq a[n-1]$;
asegura $(x \in a \wedge (pos == cabeza([j|j \leftarrow [0..|a|), a[j] == x]))$
$\bigvee\ (x \notin a \wedge (pos == -1))$;

Búsqueda binaria retornando la menor posición

```
int buscarBinMP(int a[], int n, int x) {
    int i = 0, d = n - 1;
    int m;
    while (d > i+1) {
        m = i + (d - i) / 2;
        if (x <= a[m]) d=m;
        else i=m;
    }
    if (x == a[i]) return i;
    else if (x == a[d]) return d;
    else return -1;
}
```


Búsqueda binaria retornando la menor posición

P : $|a| == n \wedge (\forall j \in [0..n-1))\ a[j] \leq a[j+1]$

Q : $(x \in a \wedge pos == cabeza([j|j \leftarrow [0..|a|), a[j] == x])\ \bigvee$
$(x \notin a \wedge pos == -1)$

Especificación del ciclo

P_c : $i == 0 \wedge d == n-1 \wedge a[i] \leq x \leq a[d]$

I : $(0 \leq i \leq d < n)\ \wedge\ (a[i] \leq x \leq a[d])\ \wedge\ (i > 0 \Rightarrow a[i-1] \neq x)$

Q_c : $(x \in a \wedge\ i == cabeza([j|j \leftarrow [0..|a|), a[j] == x]) \bigvee$
$(x \in a \wedge d == cabeza([j|j \leftarrow [0..|a|), a[j] == x]) \bigvee$
$(x \notin a)$

$v = d - i - 1$, cota 0

Búsqueda ternaria

Resuelve exactamente el mismo problema que la búsqueda binaria
problema buscarT $(a : [\text{Int}], x : \text{Int}, n : \text{Int}) = res : \text{Bool}$
requiere $|a| == n > 1$;
requiere $(\forall j \in [0..n-1))\ a[j] \leq a[j+1]$;
requiere $a[0] \leq x \leq a[n-1]$;
asegura $res == (x \in a)$;

Búsqueda ternaria

```
bool buscarTern(int a[], int n, int x) {
    int i = 0, d = n - 1;
    bool res;
    int m1, m2;
    while (d > i +2) {
        m1 = i + (d - i) / 3;
        m2 = i + (d - i) * 2 / 3;
        if(x <= a[m1]) d = m1;
        else if(x <= a[m2]) {
            i = m1;
            d = m2;
        }
        else i = m2;
    }
    res = ((a[i] == x) || (a[d] == x) || (a[(i+d)/2]==x));
    return res;
}
```


Búsqueda ternaria

El invariante del ciclo es:
// invariante I: $(0 \leq i \leq d < n) \ \wedge (a[i] \leq x \leq a[d])$
// variante $d - i - 2$ cota 0.

La complejidad del algoritmo de búsqueda ternaria es $O(\log_3 |a|)$. Es el mismo orden de complejidad que la búsqueda binaria, ya que $\log_3 n = (\log_2 n)/log_2(3)$, y $\log_2(3)$ es una constante (no depende de n).

De mismo modo que extendimos la búsqueda binaria a ternaria s puede extender para cualquier número de segmentos. Así obtenemos la búsqueda n-aria.

La búsqueda binaria es óptima

Cosideremos los algoritmos de búsqueda sobre arreglos ordenados, que operan evaluando condiciones booleanas exclusivamente.

Teorema: cualquier algoritmo de búsqueda (inventado o por inventarse) sobre una secuencia ordenada de longitud n toma para el peor caso al menos $\log n$ pasos.

La búsqueda binaria es óptima

Demostración. La entrada es una secuencia a ordenada de n elementos, y un elemento x a ser buscado. Hay $(n+1)$ posibilidades para x: que sea menor al $a[0]$ o que x esté entre $a[i]$ y $a[i+1]$ para $i = 0, 1.., n-2$ o que x sea mayor al $a[n-1]$.

Visualicemos todas las ejecuciones como un árbol de ejecuciones. Cada nodo es la evaluación de una expresión booleana, donde el árbol se ramifica en dos. Cada camino desde la raíz hasta una hoja es una ejecución del programa. La altura del árbol da el peor caso de cantidad de instrucciones ejecutadas por el programa.

Para cada una de las $(n+1)$ posibilidades que tiene x respecto de la secuencia a, la ejecución bebe alcanzar una hoja diferente. El árbol debe tener al menos $n+1$ hojas.

Sabemos que un árbol binario con L hojas tiene altura al menos $\log_2 L$. Por lo tanto nuestro árbol tiene altura mayor o igual que $\log_2(n+1)$.

Concluímos que la búsqueda, en el peor caso, requiere al menos $\log_2(n+1)$ pasos. □