Algoritmos y estructuras de datos

Búsqueda binaria

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

22 de julio de 2024

Citas

Hamlet (Shakespeare)

Ser o no ser, esa es la cuestión.

John Milton

Comparar las cosas grandes con las pequeñas.

Arzobispo Boiardo (Orlando enamorado)

Las comparaciones son odiosas.

Gary Winston (Antitrust, 2001)

Este negocio es binario, eres un uno o un cero. Vivo o muerto.

Problema abstracto

Dado un arreglo A con n elementos y un dato x, se desea saber si x está en el arreglo y, en ese caso, dónde está.

Problema concreto

Dado un arreglo `int a[MAX]` con `int n` elementos en las posiciones `a[0], ..., a[n-1]` y un dato `int x`, se desea saber si `x` está en el arreglo y, en ese caso, un índice `int i` tal que `a[i]==x`.

Búsqueda binaria

Idea principal

Tenemos un arreglo ordenado `a[0] <= a[1] <= ... <= a[n-1]` en el que queremos saber si está `x`. Sea `m = (n-1)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[n-1]`.
3. Si `x > a[m]` entonces `x` no está en `a[0]...a[m]`.

Búsqueda binaria

Idea principal

Tenemos un arreglo ordenado `a[0] <= a[1] <= ... <= a[n-1]` en el que queremos saber si está `x`. Sea `m = (n-1)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[n-1]`.
3. Si `x > a[m]` entonces `x` no está en `a[0]...a[m]`.

Ejemplo (Buscando 42)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79

Búsqueda binaria

Idea principal

Tenemos un arreglo ordenado `a[0] <= a[1] <= ... <= a[n-1]` en el que queremos saber si está `x`. Sea `m = (n-1)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[n-1]`.
3. Si `x > a[m]` entonces `x` no está en `a[0]...a[m]`.

Ejemplo (Buscando 42)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79
								42	43	51	55	67	70	72	79

Búsqueda binaria

Idea principal

Tenemos un arreglo ordenado `a[0] <= a[1] <= ... <= a[n-1]` en el que queremos saber si está `x`. Sea `m = (n-1)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[n-1]`.
3. Si `x > a[m]` entonces `x` no está en `a[0]...a[m]`.

Ejemplo (Buscando 42)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79
								42	43	51	55	67	70	72	79
								42	43	51					

Búsqueda binaria

Idea principal

Tenemos un arreglo ordenado `a[0] <= a[1] <= ... <= a[n-1]` en el que queremos saber si está `x`. Sea `m = (n-1)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[n-1]`.
3. Si `x > a[m]` entonces `x` no está en `a[0]...a[m]`.

Ejemplo (Buscando 42)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79
								42	43	51	55	67	70	72	79
								42	43	51					
								42							

Búsqueda binaria

Idea más general

Tenemos un arreglo ordenado `a[i] <= a[i+1] <= ... <= a[j]` en el que queremos saber si está `x`. Sea `m = (i+j)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[j]`.
3. Si `x > a[m]` entonces `x` no está en `a[i]...a[m]`.

En los últimos dos casos, la búsqueda continúa en el intervalo no descartado.

Búsqueda binaria

Idea más general

Tenemos un arreglo ordenado `a[i] <= a[i+1] <= ... <= a[j]` en el que queremos saber si está `x`. Sea `m = (i+j)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[j]`.
3. Si `x > a[m]` entonces `x` no está en `a[i]...a[m]`.

En los últimos dos casos, la búsqueda continúa en el intervalo no descartado.

Tiempo de ejecución

Sea $T(n)$ el número de comparaciones que hacemos en el peor de los casos. Entonces $T(1) = 1$ y $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1$ si $n \geq 2$. Finalmente, $T(n) = \lceil \log_2 n \rceil + 1$.

Búsqueda binaria

Implementación iterativa con arreglos

```
int  binaria(int i, int j, int a[], int x) {
  while (i <= j) {
    int m = i + (j-i)/2; // m = (i+j)/2
    if (x == a[m])
      return m;
    if (x < a[m])
      j = m-1;
    else
      i = m+1;
  }
  return -1;
}
```

Búsqueda binaria

Implementación iterativa con apuntadores

```
int* binaria(int *p, int *q, int x) {
  while (p <= q) {
    int *m = p + (q-p)/2; // resta de apuntadores
    if (x == *m)
      return m;
    if (x < *m)
      q = m-1;
    else
      p = m+1;
  }
  return NULL;
}
```

Podemos hacer búsqueda binaria de x en el arreglo $(a_i, \ldots, a_j)$ así:

$$f(i,j) = \begin{cases} -1 & \text{si } i > j \\ m & \text{si } x = a_m \\ f(i, m-1) & \text{si } x < a_m \\ f(m+1, j) & \text{si } x > a_m \end{cases}$$

donde $m = \lfloor \frac{i+j}{2} \rfloor$.

Búsqueda binaria

Ejemplo: buscando 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79
								42	43	51	55	67	70	72	79
								42	43	51					
								42							

$$
\begin{aligned}
f(0,15) &= f(\lfloor \tfrac{0+15}{2} \rfloor + 1, 15) \\
&= f(8,15) \\
&= f(8, \lfloor \tfrac{8+15}{2} \rfloor - 1) \\
&= f(8,10) \\
&= f(8, \lfloor \tfrac{8+10}{2} \rfloor - 1) \\
&= f(8,8) \\
&= 8
\end{aligned}
$$

Búsqueda binaria

Ejemplo

$$
\begin{aligned}
f(0,15) &= f(\lfloor \tfrac{0+15}{2} \rfloor + 1, 15) \\
&= f(8,15) \\
&= f(8, \lfloor \tfrac{8+15}{2} \rfloor - 1) \\
&= f(8,10) \\
&= f(8, \lfloor \tfrac{8+10}{2} \rfloor - 1) \\
&= f(8,8) \\
&= 8
\end{aligned}
$$

Árbol de recursión

$$
\begin{array}{c}
f(0,15) = f(8,15) \\
\downarrow \\
f(8,15) = f(8,10) \\
\downarrow \\
f(8,10) = f(8,8) \\
\downarrow \\
f(8,8) = 8
\end{array}
$$

Árbol de recursión

$$
\begin{array}{c}
(0,15) \\
\downarrow \\
(8,15) \\
\downarrow \\
(8,10) \\
\downarrow \\
(8,8)
\end{array}
$$

Búsqueda binaria

Implementación recursiva con arreglos

```
int  binaria(int i, int j, int a[], int x) {
  if (i <= j) {
    int  m = i + (j-i)/2;
    if (x == a[m])
      return m;
    if (x < a[m])
      return binaria(i, m-1, a, x);
    else
      return binaria(m+1, j, a, x);
  }
  return -1;
}
```

Búsqueda binaria

Implementación recursiva con apuntadores

```
int* binaria(int *p, int *q, int x) {
  if (p <= q) {
    int *m = p + (q-p)/2;
    if (x == *m)
      return m;
    if (x < *m)
      return binaria(p, m-1, a, x);
    else
      return binaria(m+1, q, a, x);
  }
  return NULL;
}
```

Búsqueda binaria

Generalización

Tenemos un arreglo ordenado `a[i] <= a[i+1] <= ... <= a[j-1]` de ceros y unos en el que queremos saber donde está el primer 1. Si `i < j`, sea `m = (i+j)/2` y revise `a[m]`.

1. Si `a[m]==1`, entonces el primer 1 no está en `a[m+1]...a[j]` y hacemos `j = m`.
2. Si `a[m]==0`, entonces el primer 1 no está en `a[i]...a[m]` y hacemos `i = m+1`.

En ambos casos, la búsqueda continúa en el intervalo no descartado (si no está vacío).

Búsqueda binaria

Generalización

Tenemos un arreglo ordenado `a[i] <= a[i+1] <= ... <= a[j-1]` de ceros y unos en el que queremos saber donde está el primer 1. Si `i < j`, sea `m = (i+j)/2` y revise `a[m]`.

1. Si `a[m]==1`, entonces el primer 1 no está en `a[m+1]...a[j]` y hacemos `j = m`.
2. Si `a[m]==0`, entonces el primer 1 no está en `a[i]...a[m]` y hacemos `i = m+1`.

En ambos casos, la búsqueda continúa en el intervalo no descartado (si no está vacío).

```
int primeruno(int i, int j, int a[]) {
  while (i < j) {
    int m = i + (j-i)/2; // m = (i+j)/2
    if (a[m]) j = m;
    else i = m+1;
  }
  return i;
}
```

Búsqueda binaria

Aplicación a cota superior

Tenemos un arreglo ordenado `a[i] <= a[i+1] <= ... <= a[j-1]` en el que queremos saber donde está el primer elemento mayor a `x`. Si `i < j`, sea `m = (i+j)/2` y revise `a[m]`.

1. Si `a[m] > x`, entonces el primero `> x` no está en `a[m+1]...a[j]` y hacemos `j = m`.
2. Si no, entonces el primero `> x` no está en `a[i]...a[m]` y hacemos `i = m+1`.

En ambos casos, la búsqueda continúa en el intervalo no descartado (si no está vacío).

Búsqueda binaria

Aplicación a cota superior

Tenemos un arreglo ordenado `a[i] <= a[i+1] <= ... <= a[j-1]` en el que queremos saber donde está el primer elemento mayor a `x`. Si `i < j`, sea `m = (i+j)/2` y revise `a[m]`.

1. Si `a[m] > x`, entonces el primero `> x` no está en `a[m+1]...a[j]` y hacemos `j = m`.
2. Si no, entonces el primero `> x` no está en `a[i]...a[m]` y hacemos `i = m+1`.

En ambos casos, la búsqueda continúa en el intervalo no descartado (si no está vacío).

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

Búsqueda por interpolación

Regresando a la idea de los saltos

Hasta ahora hemos usado saltos independientes de los datos.

Búsqueda por interpolación

Regresando a la idea de los saltos

Hasta ahora hemos usado saltos independientes de los datos.

Usemos los datos

Si los elementos del arreglo `a` son numéricos entonces podemos hacer cálculos con ellos. En particular, si `i < j` y `a[i] <= x <= a[j]` podemos usar una línea recta para estimar la posición `m` en la que debería estar `x`:

$$\frac{\texttt{m - i}}{\texttt{j - i}} = \frac{\texttt{x - a[i]}}{\texttt{a[j] - a[i]}}$$

de donde `m = i + (j-i)*(x-a[i])/(a[j]-a[i])`. ¿Qué pasa si `a[i] = a[j]`?

Búsqueda por interpolación

Regresando a la idea de los saltos

Hasta ahora hemos usado saltos independientes de los datos.

Usemos los datos

Si los elementos del arreglo `a` son numéricos entonces podemos hacer cálculos con ellos. En particular, si `i < j` y `a[i] <= x <= a[j]` podemos usar una línea recta para estimar la posición `m` en la que debería estar `x`:

$$\frac{\texttt{m - i}}{\texttt{j - i}} = \frac{\texttt{x - a[i]}}{\texttt{a[j] - a[i]}}$$

de donde `m = i + (j-i)*(x-a[i])/(a[j]-a[i])`. ¿Qué pasa si `a[i] = a[j]`?

Cantidad de comparaciones

Si los datos están uniformemente distribuidos entonces la búsqueda basada en esta idea hace $\approx \log_2 \log_2 n$ comparaciones en promedio.

1. Escribe una función `int inferior(int i, int j, int a[], int x)` que regrese el menor índice `k` tal que `a[k] >= x` o bien `j+1` si tal índice no existe.
2. Escribe una función `int superior(int i, int j, int a[], int x)` que regrese el menor índice `k` tal que `a[k] > x` o bien `j+1` si tal índice no existe.
3. Reescribe `inferior` y `superior` con apuntadores.
4. Estoy pensando en un entero positivo x arbitrariamente grande y tú quieres encontrarlo. Para ello puedes hacerme preguntas del estilo ¿cómo es tu número comparado con z? y yo te responderé si son iguales o cuál es mayor. ¿Cómo encontrarías x rápidamente?
5. Implementa la búsqueda por interpolación.