Algoritmos de búsqueda en el modelo de comparación

Capítulo 5 — Análisis de Algoritmos con Julia

Eric S. Téllez

Tabla de contenidos

1. Algoritmos de búsqueda en el modelo de comparación

Analizar algoritmos de búsqueda en arreglos ordenados basados en funciones de comparación, con el objetivo de localizar elementos y posiciones específicas, usando técnicas de peor caso y adaptables a la distribución de los datos para una solución eficiente de problemas informáticos.

1.1. Problema

Sea $A[1..n] = a_1, \cdots, a_n$ un arreglo ordenado con $n \geq 1$ y un operador $<$ (menor que); por simplicidad, también usaremos $\leq$ (menor o igual que). Supondremos que no hay elementos duplicados en A; nótese que esto no implica una pérdida de generalidad.

La tarea será: dado el valor x a ser localizado en A, el problema consiste en determinar la posición de inserción p tal que suceda alguna de las siguientes condiciones:

- si $p = 1$ entonces $x \leq A[p]$.
- si $2 \leq p \leq n$ entonces $A[p-1] < x \leq A[p]$.
- si $p = n+1$ entonces $A[n] < x$.

1.1.1. Costo de peor caso

Para $A[1..n]$ y el valor x cuya posición de inserción se desea localizar, el resultado puede ser cualquiera de las $n+1$ posiciones posibles, i.e., instancias del problema. Un algoritmo naïve utilizaría n comparaciones para resolverlo.

```
"""
    seqsearch(A, x, sp=1)

Búsqueda exhaustiva con inicio
"""
function seqsearch(A, x, sp=1)
    n = length(A)
    while sp <= n && x > A[sp]
        sp += 1
    end

    sp
end

let S=[10, 20, 30, 40, 50, 60, 70]
    (seqsearch(S, 0), seqsearch(S, 69), seqsearch(S, 70), seqsearch(S, 71))
end
```

```
(1, 7, 7, 8)
```

Sin embargo, dado que el arreglo está ordenado y no hay duplicados, se puede mejorar mucho el tiempo de búsqueda.

[1]

El costo de búsqueda para cualquier instancia es $O(\log n)$, y viene de la búsqueda binaria:

```
"""
    binarysearch(A, x, sp=1, ep=length(A))

Encuentra la posición de inserción de `x` en `A` en el rango `sp:ep`
"""
function binarysearch(A, x, sp=1, ep=length(A))
    while sp < ep                                                    ③
```

[1] Si se permiten duplicados se pueden mejorar mucho los tiempos, sobre todo si podemos preprocesar el arreglo, i.e., para determinar las zonas con duplicados.

```
        mid = div(sp + ep, 2)                                    ①
        if x <= A[mid]
            ep = mid                                             ②
        else
            sp = mid + 1
        end
    end

    x <= A[sp] ? sp : sp + 1                                     ④
end

let S=[10, 20, 30, 40, 50, 60, 70]
    (binarysearch(S, 0), binarysearch(S, 69), binarysearch(S, 70), binarysearch(S, 71))
end
```

① Para el rango de búsqueda $sp : ep$ se determina su punto central mid y se compara con x,
② Si el elemento x está a la izquierda, se ajusta el límite superior ep, o de lo contrario se ajusta sp.
Ambos ajustes se hacen tomando en cuenta la posición comparada.
③ Se itera mientras no se junten los dos extremos del rango.
④ Finalmente, se ajusta para valores fuera del rango.

```
(1, 7, 7, 8)
```

Este algoritmo es simple y efectivo, y es capaz de resolver cualquier instancia en tiempo logarítmico, y esto lo hace al dividir el rango siempre a la mitad por cada iteración. El costo de búsqueda binaria es de $C_{\text{bin}}(n) = \lfloor \log n \rfloor + O(1)$ comparaciones antes de colapsar el rango donde puede estar la posición de inserción.

Es importante hacer notar que la búsqueda binaria es muy eficiente en memoria y tiene un peor caso óptimo, ya que es idéntico al costo del problema, i.e., así lo determinamos. Si fuera posible probar varios puntos, i.e., m segmentos en una sola operación, el costo estaría acotado por $\lceil \log_m n \rceil$. Esto tiene sentido para estructuras de datos que trabajan en diferentes niveles de memoria, donde aunque las comparaciones en hardware moderno sean binarias, por la diferencia de velocidad entre los diferentes niveles de memoria se puede considerar que el costo dominante es, por ejemplo, acceder a una zona de disco y obtener una decisión entre $m - 1$ posibles, que particionan los rangos en m divisiones.

1.2. Búsqueda *no* acotada

Cuando el tamaño del arreglo es demasiado grande, o la relación entre p/n es significativamente pequeña, la búsqueda acotada no es la mejor opción. Aun cuando en la práctica el límite superior n podría estar determinado, y por lo tanto, se pueden resolver búsquedas en $O(\log n)$, es posible obtener una cota relativa a p, independiente de n, por lo que los casos de interés se verán beneficiados.

Una estrategia simple y poderosa es la siguiente:

1. Determinar un *buen* rango que contenga la respuesta.
2. Aplicar búsqueda binaria en ese rango para obtener la respuesta.

Bentley y Yao (1976) describen en detalle una familia de algoritmos casi óptimos para la búsqueda no acotada siguiendo la estrategia anteriormente mencionada; en particular, poniendo un énfasis importante en la determinación del rango. Lo consiguen mediante la definición de algoritmos como se describe en el resto de la sección.

1.2.1. Algoritmo B_0 (búsqueda unaria)

Es el algoritmo más simple, y ya lo vimos con anterioridad, realiza una búsqueda exhaustiva de la posición de inserción, haciendo pruebas para toda posición $x \leq A[1], x \leq A[2], \cdots, x \leq A[p+1]$, por lo que su costo será de $p+1$.

Sea $F_0(n)$ una secuencia de puntos para un arreglo de longitud n, donde se harán comparaciones para determinar el rango que contenga la respuesta para el algoritmo B_0 y $C_0(p)$ el costo de búsqueda. Entonces:

- $F_0(n) = 1, 2, \cdots, n, n+1$.
- $C_0(p) = p+1$; no requiere búsqueda binaria.

1.2.2. Algoritmo B_1 (búsqueda doblada: *doubling search/galloping*)

Consiste en comparar las posiciones 2^i, i.e., $2^1, 2^2, 2^3, \cdots, 2^{\lfloor \log_2 p+1 \rfloor+1}$, tal que $A[2^{\lfloor \log_2 p \rfloor+1}] \leq x \leq A[2^{\lfloor \log_2 p+1 \rfloor+1}]$. De manera similar a B_0, definimos $F_1(n)$ y C_1:

- $F_1(n) = 2^1, 2^2, \cdots, 2^{\log \lfloor p \rfloor+1}$;
- $C_1(p) = C_{\text{bin}}(2^{\log_2 p+1}) + \log_2 (p+1) + 1 < 2 \log_2 p + O(1)$.

La explicación viene a continuación. El número de comparaciones para determinar el rango está determinado por $\lfloor \log_2 p + 1 \rfloor + 1$. Una vez determinado el rango, la búsqueda binaria sobre

$$A[2^{\lfloor \log_2 p \rfloor+1} : 2^{\lfloor \log_2 (p+1) \rfloor+1}],$$

lo cual corresponde a $\log_2 2^{\log (p+1)+1}/2 = \log_2 (p+1)$. El costo $C_1(p)$ puede ser escrito como $2 \log_2 p + O(1)$, con un poco de manipulación algebraica.

Es importante saber cuándo usar un algoritmo u otro, y por lo tanto determinar cuándo $2 \log_2 p + O(1) < \log_2 n + O(1)$. Para simplificar este análisis ignoraremos algunos detalles de la expresión:

$$2 \log_2 p < \log_2 n, \tag{1}$$
$$2^{\log_2 p^2} < 2^{\log_2 n}, \tag{2}$$
$$p^2 < n, \tag{3}$$
$$p < \sqrt{n}; \tag{4}$$
$$\tag{5}$$

esto indica que si p es menor a $\sqrt{n}$ entonces hay una ventaja al usar B_1; lo cual nos dice que para posiciones cercanas al inicio el uso de B_1 puede llevar a búsquedas más veloces. Note que en la práctica es necesario tener en cuenta la memoria; interesantemente, para valores de p pequeños es posible que esto beneficie al algoritmo ya que podría mantener las listas en caché.

El siguiente código implementa B_1

```
function doublingsearch(A, x, sp=1)
    n = length(A)
    p = 0
    i = 1

    while sp+i <= n && A[sp+i] < x                                    ①
        p = i
        i += i
    end

    binarysearch(A, x, sp + p, min(n, sp+i))                          ②
end

let S=[10, 20, 30, 40, 50, 60, 70]
    (doublingsearch(S, 0), doublingsearch(S, 69), doublingsearch(S, 70), doublingsearch(S, 71))
end
```

① Determinación del rango.
② Aplicar un algoritmo de búsqueda eficiente en el rango que contiene la respuesta.

```
(1, 7, 7, 8)
```

Es cierto que estos algoritmos son oportunistas, pero hay aplicaciones donde esto realmente sucede. En el peor caso, el costo será apenas dos veces el óptimo.

1.2.3. Algoritmo B_2 (búsqueda doblemente doblada, *doubling-doubling search*)

Aquí será más clara la dinámica. B_2 consiste en comparar las posiciones 2^{2^i}, i.e., $2^4, 2^{16}, 2^{256}, \cdots, 2^{2^{\lfloor \log_2 \lfloor \log_2 p+1 \rfloor +1 \rfloor +1}}$, tal que

$$A[2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor +1 \rfloor +1}}] \leq x \leq A[2^{2^{\lfloor \log_2 \lfloor \log_2 p+1 \rfloor +1 \rfloor +1}}];$$

La determinación de este rango requiere $\lfloor \log_2 \lfloor \log_2 p + 1 \rfloor + 1 \rfloor + 1$ comparaciones; sin embargo, este rango seguramente será muy grande, por el tamaño de los saltos que se están dando entre puntos de comparación, por lo que no conviene usar búsqueda binaria y podemos aplicar B_1 para resolver en ese rango acotado.

$$F_2(n) = 2^{2^1}, 2^{2^2}, \cdots, 2^{2^{\lfloor \log_2 \lfloor \log_2 n \rfloor + 1 \rfloor + 1}}; \tag{6}$$
$$C_2(p) = \lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1 + C_1(p') \tag{7}$$
$$< \log_2 p + 2 \log_2 \log_2 p + O(1); \tag{8}$$
$$\tag{9}$$

donde $p' = p - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}}$, es decir, la posición de inserción en el rango ya acotado.

Note cómo el término de mayor peso es muy similar a B_1 pero destaca la inclusión del término $\log \log$ que permite adaptarse a p muy grandes con un pequeño costo adicional, que en términos prácticos se puede ver como una constante.

La idea principal es como sigue: una vez determinado el rango, en lugar de usar búsqueda binaria y tener un costo $\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1 + C_{\text{bin}}(2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}})$ es preferible usar B_1 y conseguir un algoritmo que se adapte a la entrada. De manera más precisa, tomar ventaja de

$$C_1(2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}}) < C_{\text{bin}}(2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}})$$

cuando

$$p' < \sqrt{2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}}}$$

.

Simplificando las expresiones, la relación que nos describe cuándo es mejor usar B_2 que la búsqueda binaria es como sigue:

$$\log_2 p + 2 \log_2 \log_2 p < \log_2 n \tag{10}$$
$$2^{\log_2 p + \log_2 \log_2^2 p} < 2^{\log_2 n} \tag{11}$$
$$2^{\log_2 (p \log_2^2 p)} < 2^{\log_2 n} \tag{12}$$
$$2p \log_2 p < n \tag{13}$$
$$p \log_2 p^2 < n \tag{14}$$
$$\tag{15}$$

Si $p = \sqrt{n}$ entonces $\sqrt{n} \log_2 n$ claramente es menor que n incluso para valores relativamente pequeños de n, por lo que B_2 funciona mejor para p relativamente grandes en comparación con B_1.

1.2.4. Algoritmo B_k

Bentley y Yao (1976) generalizan la estrategia para cualquier k. De manera simplificada:

- $F_k(n) = 2^{\cdot^{\cdot^{2^{2^i}}}}$ (exponenciando k veces) para i desde 1 a $\log_2^{(k)} n$;
- $C_k(p) = \log_2^{(k)}(p) + C_{k-1}(2^{\cdot^{\cdot^{2^{\log_2^{(k)}(p)}}}} - 2^{\cdot^{\cdot^{2^{\log_2^{(k)}(p)-1}}}})$;

donde $\log_2^{(k)}(n) = \log_2(\lfloor \log_2^{(k-1)}(n) \rfloor + 1)$, con el caso base de $\log_2^{(1)} n = \lfloor \log_2 n \rfloor + 1$.

La estrategia lleva a que el valor casi óptimo para la búsqueda por comparación se da cuando $k = \log_2^\star n$ donde $\log_2^\star$ es el logaritmo iterado, que está definido como las veces que se debe iterar aplicando el logaritmo para obtener un valor de 1 o menor que 1, i.e., la k más pequeña tal que $\log_2^{(k)} n \leq 1$.

1.3. Ejercicios

- Implementar y probar B_2.
- Derivar el costo $C_2(p)$.
- ¿Cuándo B_1 es mejor que B_2?
- Haga un pseudo-código para B_k.
- ¿Cuál es el costo C_k?
- ¿Qué es un árbol binario de búsqueda?
- ¿Cuál es el costo de búsqueda en un árbol? ¿Qué se debe hacer para asegurar los costos?
- ¿Qué es un finger tree?
- ¿Cuál es el costo de búsqueda de la *skip list*?
- ¿Cómo se puede hacer la *skip list* adaptativa? ¿Qué otra forma podría aplicar?

1.4. Material audio-visual

En el siguiente video se adentrará en diferentes estrategias de búsqueda, notoriamente aquellas que llamaremos oportunistas o adaptables (adaptive). Estas técnicas nos permitirán tomar provecho de instancias sencillas de problemas e incrementar el desempeño en ese tipo de instancias.

Tenga en cuenta que, honrando la literatura, usaremos de forma indiscriminada listas ordenadas como sinónimo de arreglos ordenados.

1.5. Actividades

1. Implementa y compara los siguientes algoritmos de búsqueda:

- Búsqueda binaria acotada
- Búsqueda secuencial B_0
- Búsqueda no acotada B_1
- Búsqueda no acotada B_2

2. Utiliza los diferentes archivos proporcionados, los cuales tienen datos con diferentes distribuciones. Usa las listas de posteo de la Unidad 3 y los archivos de consulta para obtener las consultas que deberán realizarse. Mide tanto el número de comparaciones como el tiempo necesario para solucionar las consultas.
3. Compara todos los métodos implementados por archivo de consulta, usa figuras o tablas de datos (número de comparaciones y tiempo por separado).
4. Discute tus resultados.

1.5.1. Entregable

Su trabajo se entregará en PDF y con el notebook fuente, se entregará en el sistema en archivos separados. El código se debe documentar, así como mantener una estructura del documento que permita a un lector interesado entender el problema, sus experimentos y metodología, así como sus conclusiones. Tenga en cuenta que los notebooks pueden alternar celdas de texto y código.

No olvide estructurar su reporte, en particular el reporte debe cubrir los siguientes puntos:

- Título del reporte, su nombre.
- Introducción.
- Desarrollo de las actividades.
- Conclusiones
- Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

Nota: no olvides revisar la rúbrica del curso.

Bentley, Jon Louis, y Andrew Chi-Chih Yao. 1976. "An almost optimal algorithm for unbounded searching". *Information processing letters* 5 (SLAC-PUB-1679).