Diseño de Algoritmos
Prática 2

David Ruiz Rodríguez, Mohamed Rodrigo El Badry,
Nicolas Recinella Vidán, Denilson Palomino Adán

19 de diciembre de 2024

Índice

1. Ejercicio 1

Traduzca el código dado de Python a Scheme, utilizando *list*, *car* y *cdr*, y que el código sea tambiés valido si se utiliza el tipo vector.

Para poder implementar el algoritmo de búsqueda por biparticipación de acuerdo al pseudocódigo, se han tenido que añadir varias funciones auxiliares: posición, lista-ref y len. Posición obtiene el indice de un objeto en la lista, len la longitud de una lista y lista-ref que obtiene un objeto a partir de su indice.

El algoritmo consiste en buscar un elemento en una lista ordenada, en el proceso se divide el programa de forma recursiva en 2 sublistas y se decide en que lista esta contenido el objeto ha buscar. El algoritmo termina cuando llega a una lista de tamaño 1 que corresponde al objeto si se encuentra en la lista en caso contrario da error.

```
1  #lang racket
2  (define (busca x lista)
3    (busca-aux x lista 0 (len lista)))
4
5  (define (busca-aux x lista i j)
6    (let ((m (quotient (+ i j) 2)))
7    (if (>= i j)
8        #f  ; Retorna falso en vez de None
9          (cond
10           ((= (posicion lista m) x) m)
11           ((< (posicion lista m) x) (busca-aux x lista (+ m 1) j))
12           (else (busca-aux x lista i m))))))
13
14 (define (posicion objeto m)
15   (if (pair? objeto)
16       (list-ref objeto m)
17   (vector-ref objeto m)))
18
19 (define (list-ref lista m)
20   (if (= m 0)
21       (car lista)
22   (list-ref (cdr lista) (- m 1))))
23
24
25 (define (len seq)
26   (cond
27     ((vector? seq) (vector-length seq))
28     ((list? seq) (length seq))
29     (else (error "El tipo no es ni una lista ni un vector"))))
30
31 (define lista '(1 3 5 7 9 11 13 15))
32
33 (busca 7 lista)
34 (busca 4 lista)
35
36 (define mi-vector (vector 1 3 5 7 9 11 13 15 17 19))
37
38 (busca 9 mi-vector)
39 (busca 6 mi-vector)
40
```

Figura 1: Código recursivo

2. Ejercicio 2

Calcula una cota superior para la cantidad de llamadas a `busca_aux` que realiza el algoritmo en función de la longitud de la lista de entrada. Acota la complejidad espacial y temporal de la búsqueda por bipartición.

La cota superior de las llamadas es: $2 \cdot log_2(n) + 1$, esto se debe a que hay dos llamadas recursivas al dividir el problema y la altura máxima del árbol generado es el resultado de un logaritmo de 2, ya que este se va dividiendo en 2 sucesivamente.

La complejidad temporal es de $2 \cdot log_2(n)$, esto es ya que el tamaño máximo del árbol de búsqueda es la altura que va crecer de acuerdo al logaritmo.
La complejidad espacial sin embargo es constante ya que solo se guardan los indices para separar el array y no este entero. Esto es porque en memoria de los indices en el peor de los casos es siempre menor al array.

3. Ejercicio 3

Suponiendo que la extensión de la representación binaria de cada uno de los números de la lista está acotada por $log_2 n$, deduce una cota superior asintótica para la complejidad computacional del algoritmo en términos de operaciones bit.

Al hacer que sean operaciones en bits, esto hace que aumente la complejidad temporal de las operaciones que se realizan pasando operaciones como la comparación de ser lineales a ser logarítmicas creciendo de acuerdo a la longitud de los números a comparar. Por tanto habría que multiplicar la complejidad obtenida en el apartado anterior y multiplicarla por la complejidad de la comparación con lo que obtenemos: $log_2(n) \cdot log_2(m)$, siendo m la longitud del numero menor que se compara en bits.

Desde el punto de vista espacial aumenta el uso de memoria en $log_2(m)$, ya que ahora la longitud de los números provoca que se tengan que guardar más o menos bits en las comparaciones.

4. Ejercicio 4

¿Se puede aprovechar el teorema maestro para las dos preguntas anteriores?

En otras palabras:

1. ¿Se puede usar el teorema maestro para calcular una cota superior para la cantidad de llamadas a `busca_aux` en función de la longitud del parámetro de entrada?

   Siempre y cuando tengas los demás datos, sí, puedes hacerlo, porque si el teorema maestro es:

   $$T(n) = a \cdot T(\lfloor \frac{n}{b} \rfloor) + f(n)$$

   - a: Es el número de llamadas recursivas.
   - b: Es el número de sub-problemas en los que divides el problema original.
   - d: es el grado (exponente mayor del polinomio) de $f(n)$.

   Sabiendo que:

   $$\begin{cases} \text{Si } f(n) \in O(n^d), \text{ con } d < log_b a, & T(n) \in \Theta(n^{log_b a}); \\ \text{Si } f(n) \in \Theta(n^d), \text{ con } d = log_b a, & T(n) \in \Theta(n^{log_b a} \cdot logn); \\ \text{Si } f(n) \in \Omega(n^d), \text{ con } d > log_b a, & T(n) \in \Theta(f(n)). \end{cases}$$

   Pero para nuestro caso, el teorema maestro sería:

   $$T(n) = 2 \cdot T(\lfloor \frac{n}{2} \rfloor) + n$$

   Donde $a = 1$, $b = 2$ y $d = 0$. Lo cual nos coloca en la posición de $d = log_b a \rightarrow T(n) \in \Theta(n^{log_b a} \cdot log(n))$

   Para saber cuantas veces se llama a la función `busca_aux`, tenemos que contar cuantas veces se llama a esa función por llamada recursiva y las veces que se hará la llamada recursiva hasta alcanzar el caso base, lo cual es:

   $$a \cdot log_b(n) = log_2(n)$$

2. ¿Se puede usar para acotar su complejidad espacial y temporal?

   Siguiendo lo explicado en el apartado anterior, como nos encontramos en el caso $d = log_b a \rightarrow T(n) \in \Theta(n^{log_b a} \cdot logn)$, la complejidad temporal del algoritmo es:

   $$\Theta(n^{log_2 1} \cdot log(n)) = \Theta(1 \cdot log(n)) = \Theta(log(n)) \rightarrow \Theta(log_2(n))$$

   Por otro lado, la complejidad espacial es equivalente al tamaño del dato inicial, es decir, n.

3. ¿Podría usarse para estimar una cota superior asintótica para la complejidad computacional del algoritmo en términos de operaciones de bit?

   Para saber esto, primero tenemos que saber la complejidad de las operaciones básicas usadas en el algoritmo en términos de operaciones de bit.

   - Comparación: Asumiendo que la comparación se hace de forma iterativa bit a bit, su complejidad es m, siendo m la longitud de una palabra del procesador.
   - Suma: En este caso sabemos que se hace de forma iterativa porque se tiene que tener en cuenta el carry en todo momento. Por lo que su complejidad es m, siendo m la longitud de una palabra del procesador.
   - Resta: Misma complejidad que la suma, porque sigue habiendo carry.
   - Desplazamiento: Se usa para desplazar todos los bits, en este caso, hacia la derecha. Su implementación es, seguramente, iterativa bit a bit, por lo que su complejidad es m, siendo m la longitud de una palabra del procesador.

   En el código, hay tres condicionales, una suma, una resta y un desplazamiento, quizás dos sumas, dependiendo de qué llamada recursiva se haga. Lo cual hace que estas operaciones cuesten $(m+m+m)+m+m+m+(m) = 7 \cdot m$

   Y como esto se hace cada vez que se ejecuta el código, significa que la complejidad computacional en términos de bits es:

$$log_2(n) \cdot 7m \Rightarrow log_2(n) \cdot m$$

5. Ejercicio 5

Escribe una versión iterativa del algoritmo de búsqueda por bipartición, dado arriba en forma recursiva.

En esta implementación se ha añadido la función busca-iter que toma dos argumentos i y j que representan el inicio y final de un segmento de la lista. Esta función primero comprueba si el elemento medio es el elemento que estamos buscando, si no lo es se vuelve a llamar la función busca-iter pero con las dos mitades del segmento.

El programa funciona de la siguiente forma: se pasan a la función busca un elemento a buscar y una lista. Dentro de busca se invoca la función busca-iter con la toda la longitud de la lista. En las siguientes iteraciones busca-iter divide el programa hasta encontrar el elemento.

```
#lang racket
(define (busca x lista)
  (define (busca-iter i j)
    (if (>= i j)
        #f  ; Retorna falso si en vez de None
        (let ((m (quotient (+ i j) 2)))
          (cond
            ((= (posicion lista m) x) m)
            ((< (posicion lista m) x) (busca-iter (+ m 1) j))
            (else (busca-iter i m))))))
  (busca-iter 0 (len lista)))

(define (posicion objeto m)
  (if (pair? objeto)
      (list-ref objeto m)
  (vector-ref objeto m)))


(define (list-ref lista m)
  (if (= m 0)
      (car lista)
  (list-ref (cdr lista) (- m 1))))


(define (len seq)
  (cond
    ((vector? seq) (vector-length seq))
    ((list? seq) (length seq))
    (else (error "El tipo no es ni una lista ni un vector"))))

(define lista '(1 3 5 7 9 11 13 15))

(busca 7 lista)
(busca 4 lista)

(define mi-vector (vector 1 3 5 7 9 11 13 15 17 19))

(busca 9 mi-vector)
(busca 6 mi-vector)
```

Figura 2: Código iterativo