Estructuras de Datos y de la Información

Ingeniería Técnica en Informática de Gestión. Curso 2007/2008

Ejercicios del Tema 2

Diseño de algoritmos recursivos

1. Dado un vector de enteros de longitud N, diseñar un algoritmo de complejidad O(N·logN) que encuentre el par de enteros más cercanos. ¿Se puede hacer con complejidad O(N)?

2. Dado un vector de enteros de longitud N, diseñar un algoritmo de complejidad O(N·logN) que encuentre el par de enteros más lejanos. ¿Se puede hacer con complejidad O(N)?

3. Diseñar un algoritmo recursivo que realice el cambio de base de un número binario dado en su correspondiente decimal.

4. Diseñar un algoritmo recursivo que, dado un array de caracteres, cuente el número de vocales que aparecen en él.

5. Escribir una función recursiva que devuelva el producto de los elementos de un array mayores o iguales que un determinado número b

6. Implementa una función recursiva simple *cuadrado* que calcule el cuadrado de un número natural *n*, basándote en el siguiente análisis de casos:

   Caso directo: Si `n = 0`, entonces $n^2 = 0$

   Caso recursivo: Si `n > 0`, entonces $n^2 = (n-1)^2 + 2*(n-1) + 1$

7. Implementa una función recursiva *log* que calcule la parte entera de $\log_b n$, siendo los datos *b* y *n* enteros tales que $b \geq 2 \wedge n \geq 1$. El algoritmo obtenido deberá usar solamente las operaciones de suma y división entera.

8. Implementa una función recursiva *bin* tal que, dado un número natural *n*, *bin*(*n*) sea otro número natural cuya representación decimal tenga los mismos dígitos que la representación binaria de *n*. Es decir, debe tenerse: *bin*(0) = 0; *bin*(1) = 1; *bin*(2) = 10; *bin*(3) = 11; *bin*(4) = 100; etc.

9. Implementa una función recursiva doble que satisfaga la siguiente especificación:

```
int sumaVec( int v[], int a, int b ) {
// Pre:  0 <= a <= num && -1 <= b <= num-1 && a <= b+1
//       siendo num el número de elementos de v

// Post: devuelve la suma de las componentes de v entre a y b
}
```

   Básate en el siguiente análisis de casos:

   Caso directo: `a ≥ b`

   `v[a..b]` tiene a lo sumo un elemento. El cálculo de *s* es simple.

   Caso recursivo: a < b

   `v[a..b]` tiene al menos dos elementos. Hacemos llamadas recursivas para sumar `v[a..m]` y `v[m+1..b]`, siendo `m = (a+b) / 2`.

10. Implementa un procedimiento recursivo simple *dosFib* que satisfaga la siguiente especificación pre/post:

```
void dosFib( int n, int& r, int& s ) {
// Pre:  n >= 0
// Post: r = fib(n) && s = fib(n+1)
}
```

En la postcondición, `fib(n)` y `fib(n+1)` representan los números que ocupan los lugares n y n+1 en la sucesión de Fibonacci, para la cual suponemos la definición recursiva habitual.

11. Implementa una función recursiva que calcule el número combinatorio $\binom{n}{m}$ a partir de los datos *m, n: Integer* tales que $n \geq 0 \wedge m \geq 0$. Usa la recurrencia siguiente:

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m} \quad \text{siendo } 0 < m < n$$

12. Una palabra se llama *palíndroma* si la sucesión de sus letras no varía al invertir el orden. Especifica e implementa una función recursiva final que decida si una palabra dada, representada como vector de caracteres, es o no palíndroma.

13. El problema de las torres de Hanoi consiste en trasladar una torre de n discos de diferentes tamaños desde la varilla *ini* a la varilla *fin*, con ayuda de la varilla *aux*. Inicialmente los n discos están apilados de mayor a menor, con el más grande en la base. En ningún momento se permite que un disco repose sobre otro menor que él. Los movimientos permitidos consisten en desplazar el disco situado en la cima de una de las varillas a la cima de otra, respetando la condición anterior. Construye un procedimiento recursivo *hanoi* tal que la llamada *hanoi*(*n, ini, fin, aux*) produzca el efecto de escribir una serie de movimientos que represente una solución del problema de Hanoi. Supón disponible un procedimiento *movimiento*(*i, j*), cuyo efecto es escribir "Movimiento de la varilla *i* a la varilla *j* ".

Análisis de algoritmos recursivos

14. En cada uno de los casos que siguen, plantea una ley de recurrencia para la función T(n) que mide el tiempo de ejecución del algoritmo en el caso peor, y usa el método de desplegado para resolver la recurrencia.
    (a) Función *log* (ejercicio 2).
    (b) Función *bin* (ejercicio 3).
    (c) Función *sumaVec* (ejercicio 4).
    (d) Procedimiento *dosFib* (ejercicio 5).
    (e) La función del ejercicio 6.
    (f) Función *palindroma* (ejercicio 7).
    (g) Procedimiento *hanoi* (ejercicio 8).

15. Aplica las reglas de análisis para dos tipos comunes de recurrencia a los algoritmos recursivos del ejercicio anterior. En cada caso, deberás determinar si el tamaño de los datos del problema decrece por sustracción o por división, así como los parámetros relevantes para el análisis.

16. En cada caso, calcula a partir de las recurrencias el orden de magnitud de T(n). Hazlo aplicando las reglas de análisis para dos tipos comunes de recurrencia.

(a) $T(1) = c_1;\ T(n) = 4 \cdot T(n/2) + n$, si $n > 1$

(b) $T(1) = c_1;\ T(n) = 4 \cdot T(n/2) + n^2$, si $n > 1$

(c) $T(1) = c_1;\ T(n) = 4 \cdot T(n/2) + n^3$, si $n > 1$

17. Usa el método de desplegado para estimar el orden de magnitud de T(n), suponiendo que T obedezca la siguiente recurrencia:

$$T(1) = 1;\ T(n) = 2 \cdot T(n/2) + n \cdot \log n, \text{ si } n > 1$$

¿Pueden aplicarse en este caso las reglas de análisis para dos tipos comunes de recurrencia? ¿Por qué?

Eliminación de la recursión final

18. Aplica la transformación de recursivo final a iterativo sobre la función *palindroma* del ejercicio 7.

19. La primera versión (la que aparece a la izquierda) de la búsqueda binaria es la misma que aparecía en el ejercicio 6 de la hoja 1 de ejercicios, mientras que la segunda (la que aparece a la derecha), vista en teoría, es el resultado de transformar a iterativo la versión recursiva de este algoritmo. ¿En qué se diferencian estos dos algoritmos iterativos?

Escribe un algoritmo recursivo final con la misma idea de la primera versión iterativa.

```
int buscaBin( TElem v[], int num, TElem x )
{
  int izq, der, centro;

  izq = -1;
  der = num;
  while ( der != izq+1 ) {
    centro = (izq+der) / 2;
    if ( v[centro] <= x )
      izq = centro;
    else
      der = centro;
  }
  return izq;
}
```

```
int buscaBin( TElem v[], int num, TElem x )
{
  int a, b, p, m;

  a = 0;
  b = num-1;
  while ( a <= b ) {
    m = (a+b) / 2;
    if ( v[m] <= x )
      a = m+1;
    else
      b = m-1;
  }
    p = a - 1;
  return p;
}
```

Técnicas de generalización

20. Se trata de obtener un algoritmo recursivo de complejidad lineal que dado un polinomio de coeficientes enteros, representado como un vector, y el valor de la incógnita, calcule el valor del polinomio. En el vector, de tipo *int v[ ]*, se almacenan los coeficientes del polinomio ordenados por exponentes: en la posición 0 el coeficiente de x^0, en la posición 1 el coeficiente de x, y así sucesivamente.

    Idea: Implementa primero una función recursiva *evaluaGen* de complejidad cuadrática donde las potencias de x se obtengan de forma iterativa. A continuación, plantea otra generalización *evaluaEfi* que permita obtener el algoritmo de complejidad lineal, y escribe su implementación transformando el código de *evaluaGen*. Puede conseguirse recursión final.

21. Comprueba que el procedimiento *combiGen* especificado como sigue es una generalización de la función *combi* del ejercicio 6, y que *combiGen* admite un algoritmo recursivo simple, más eficiente que la recursión doble de *combi*, implementándola.

```
void combiGen ( int a, int b, int v[], int num ) {
// Pre:  0 <= b <= a && b < num
// Post: para cada i entre 0 y b se tiene v[i] = (a i)}
```

22. Especifica e implementa un algoritmo recursivo que dado un vector de enteros calcule el número de posiciones de dicho vector que cumplan la condición de que la suma de las componentes a su izquierda coincida con la suma de las componentes a su derecha. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

23. Especifica e implementa un algoritmo recursivo que dado un vector de enteros lo reorganice de forma que los valores pares ocupen la parte izquierda del vector y los valores impares la parte derecha. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

24. Especifica e implementa un algoritmo recursivo que dado un vector de enteros determine si el vector tiene forma de *montaña*. Un vector tiene forma de *montaña* si contiene una secuencia estrictamente creciente seguida de una secuencia estrictamente decreciente −una de las dos secuencias puede ser vacía–. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

25. Especifica e implementa un algoritmo recursivo que dado un vector *v* de enteros, que viene dado en orden estrictamente creciente, determine si el vector contiene alguna posición *i* que cumpla *v[i]=i*. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

26. Especifica e implementa un algoritmo recursivo que dados dos vectores *u*, *v* de enteros, ordenados en orden estrictamente creciente, obtenga el número de elementos que aparecen en los dos vectores. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

27. Especifica e implementa un algoritmo recursivo que dado un vector *v* de enteros determine si contiene un *punto de inflexión*. Un punto de inflexión es una posición del vector tal que todas las componentes que aparecen a su izquierda son negativas, y todas las que aparecen a su derecha son positivas o 0. Si el vector contiene punto de inflexión la función devolverá su posición, y si no lo contiene devolverá –1. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

28. Especifica e implementa un algoritmo recursivo que dado un vector *v* de enteros determine si el vector tiene la forma $1^n\,0^n\,0^n\,1^n$, es decir: un cierto número *n* de unos, seguidos del doble (2*n*) de ceros, seguidos a su vez de otros *n* unos. Por ejemplo, un vector de tamaño 8 con las componentes [1,1,0,0,0,0,1,1] cumpliría lo anterior. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

29. Especifica e implementa un algoritmo recursivo que dado un vector *v* de enteros determine si el vector representa una escalera con peldaños de ancho estrictamente creciente, donde:
   - Un peldaño queda constituido por una secuencia de apariciones consecutivas del mismo elemento. Por ejemplo, el vector [1, 6, 6, 6, 2, 2, 5] tiene 4 peldaños.
   - El ancho de un peldaño es el número de apariciones consecutivas del elemento que constituye el peldaño. Por ejemplo, los anchos de los 4 peldaños del vector anterior son 1, 3, 2 y 1 (por orden de aparición).
   - Un vector forma una escalera con peldaños de ancho estrictamente creciente si cada peldaño es más ancho que el anterior, como, por ejemplo, el vector [3, 3, 1, 1, 1, 1, 5, 5, 5, 5, 5, 5].

   Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

30. Especifica e implementa un algoritmo recursivo que dado un vector *u* de enteros obtenga la longitud de la *meseta* más larga. Una *meseta* es un conjunto de posiciones consecutivas del vector que contienen el mismo valor. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.