ALGORITMOS Y ESTRUCTURAS DE DATOS · ESCUELA DE NEGOCIOS, UTDT · PRIMER SEMESTRE 2024

Guía de Ejercicios 4: Recursión y complejidad

[Version: 6 de septiembre de 2024]

Objetivos:

- Repasar la resolución recursiva de problemas computacionales e introducir la técnica Divide & Conquer.
- Introducir la definición formal de órden de complejidad algorítmica y entrenar su aplicación en el análisis de algoritmos imperativos y recursivos.

Recursión y Divide & Conquer

Ejercicio 1. Escribir funciones recursivas para resolver los siguientes problemas.

(a) `int fibonacci(int n)`: calcula el `n`-esimo número de la sucesión de Fibonacci.

(b) `bool es_par(int n)`: indica si `n` es par o no. No se debe usar el operador módulo `%`.

Ejercicio 2. Escribir funciones recursivas para resolver los siguientes problemas. Si es necesario cambiar los parámetros de entrada para poder realizar la recursión, utilizar una función auxiliar.

(a) `int productoria(const vector<int> & v)`: dado un vector de enteros no vacío `v`, calcular el resultado de multiplicar todos los números de `v`.

(b) `int cantidad_ocurrencias(string s, const vector<string> & v)`: dado un vector de strings `v` y un string `s`, devolver la cantidad de veces que aparece `s` en `v`.

(c) `int contar_coincidencias(const vector<int> & v)`: dado un vector de enteros `v`, contar cuántas veces es cierto que la `i`-ésima posición tiene el número `i` (es decir, cuántas veces `v[i]==i`).

(d) `vector<int> solo_positivos(const vector<int> & v)`: dado un vector de enteros `v`, devuelve un vector igual a `v` excepto que sólo contiene sus elementos positivos. Es decir, se filtra `v`, descartando los elementos que no sean positivos.

(e) `void multiplicar_todos(vector<int> & v, int n)`: dado un vector de enteros `v` y un entero `n`, modifica todos los elementos de `v` multiplicándolos por `n`.

Ejercicio 3.

(a) Escribir las funciones `cantidad_ocurrencias` y `multiplicar_todos` del Ejercicio 2 aplicando el esquema Divide & Conquer dividiendo el vector de entrada en dos mitades.

(b) Dibujar el árbol de recursión para `cantidad_ocurrencias` con `s = "a"` y `v = {"a", "bs", "a","a"}`.

(c) Dibujar el árbol de recursión para `multiplicar_todos` con `v = {3,2,5,8,1,9}` y `n = 2`.

Ejercicio 4. Escribir funciones recursivas para resolver los siguientes problemas. Debe aplicar el esquema Divide & Conquer diviendo la instancia de entrada en dos.

(a) Potencia: toma dos números enteros n y m, y calcula n^m.

(b) Vector montaña: toma un vector montaña de enteros, y devuelve el valor que está en el pico. Un vector es montaña si tiene una primera parte que es estrictamente creciente y una segunda parte que es estrictamente decreciente. El pico es el máximo valor del vector, que separa la parte creciente de la parte decreciente. Por ejemplo, el vector `{-1,4,6,9,3,0}` es montaña y su pico es `9`; pero el vector `{0,3,4,2,-1,8}` no es montaña.

Ejercicio 5. Más a la izquierda. Escriba un algoritmo Divide & Conquer que determine si un arreglo de tamaño potencia de 2 está "más a la izquierda". Se dice que un arreglo está "más a la izquierda" si cumple las siguientes condiciones:

- La suma de los elementos de la mitad izquierda del arreglo es mayor que la suma de los elementos de la mitad derecha.
- Ambas mitades del arreglo, a su vez, también están "más a la izquierda".

Por ejemplo, el arreglo `{4, 2, 3, 0}` está "más a la izquierda", mientras que el arreglo `{4, 2, 3, 7}` no lo está.

Ejercicio 6. Sublista de suma máxima. Resolver aplicando el esquema Divide & Conquer. Dado un vector de enteros `v`, encontrar el subvector contiguo (y con al menos un elemento) que tenga la suma más grande y devolver su suma. Ejemplos:

- para `v = {-1,2,-10,8,1,-3}` se debe devolver `9`, ya que `{8,1}` es el subvector de suma máxima.
- para `v = {-3,10,-4,5,-6,1}` se debe devolver `11`, ya que `{10,-4,5}` es el subvector de suma máxima.
- para `v = {-10,-1,-7,-42}` se debe devolver `-1`, ya que `{-1}` es el subvector de suma máxima.

Ejercicio 7. Escribir las funciones

- `int productoria(const vector<int> & v, int desde, int hasta)` y
- `int sumatoria(const vector<int> & v, int desde, int hasta)`

utilizando la técnica Divide & Conquer, pero **dividiendo el vector en 3 partes en lugar de 2**. Dibujar el árbol de recursión de ambas con `v = {1,2,3,4,5,6}`, `desde = 0`, `hasta = 6`.

Complejidad

Ejercicio 8. Indique si las siguientes afirmaciones son verdaderas o falsas. Justifique su respuesta haciendo uso de la definicion de O.

(a) $12n + 3 \in O(n^2)$

(b) $n^2 + 5n^3 \in O(n^2)$

(c) $2^n + 5 \in O(n^{10})$

(d) $\sqrt{n} \in O(n)$

(e) $n \in O(\sqrt{n})$

(f) $n \in O(\log_2 n)$

Ejercicio 9. Para cada uno de los siguientes programas, 1) definir el tamaño de entrada n, 2) identificar el peor caso de ejecución del programa, 3) escribir la función de costo temporal $T(n)$ asociada al peor caso, 4) mostrar que $T(n)$ pertenece al órden de complejidad pedido.

(a)
```
int raiz(int k){
    // PRE: k >= 0

    int res = 0;
    while(res*res <= k){
        res = res + 1;
    }
    return res - 1;
}
```

Complejidad: $O(\sqrt{n})$

(b)
```
int contar_divs(int d, int c){
    int count = 0;
    for(int i = 0; i <= c; i++){
        if(i % d == 0){
            count = count + 1;
        }
    }
    return count;
}
```

Complejidad: $O(n)$

(c)
```
bool suma_mayor_a(const vector<int> & v, int c){
    int suma = 0;
    int i = 0;
    while(i < v.size() && suma <= c){
        suma = suma + v[i];
        i = i + 1;
    }
    return suma > c;
}
```

Complejidad: $O(n)$

(d)
```
bool hay_repetidos(const vector<int> & v){
    int i = 0;
    while(i < v.size()){
        int j = i + 1;
        while(j < v.size()){
            if(v[i] == v[j])
                return true;
            j++;
        }
        i++;
    }
    return false;
}
```

Complejidad: $O(n^2)$

(e)
```
void rellenar_vector(vector<int> & v){
    int i = 0;
    while(i < v.size()){
        int sum = 0;
        int j = 0;
        while(j < 10){
            sum = sum + j;
```

```
            j++;
        }
        v[i] = sum;
        i++;
    }
}
```

Complejidad: $O(n)$

(f)
```
bool pertenece(const vector<int> & v, int e){
  int izq = 0;
  int der = v.size();
  while (izq < der){
      int med = (izq+der)/2;
      if (v[med] == e){
          return true;
      }
      else if(v[med] < e){
          izq = med  + 1;
      } else {
          der = med;
      }
  }
  return false;
}
```

Complejidad: $O(\log_2 n)$

Ejercicio 10. Escribir el árbol de recursión de las siguientes ecuaciones de recurrencia. Para cada árbol 1) indicar su altura, 2) indicar el costo de cada nivel, 3) proponer una fórmula cerrada $f(n)$ y demostrar que $T(n) \in O(f(n))$ utilizando la técnica de inducción matemática.

(a) $T(n) = 2T(n/2) + n$

(b) $T(n) = 4T(n/4) + 1$
Tip: usar la serie geométrica $\sum_{i=0}^{m} r^i = \frac{1-r^{m+1}}{1-r}$, (si $r \neq 1$).

Puede asumir en todos los casos que el costo de $T(1)$ es constantes o es igual a 1.

Ejercicio 11. Calcular y demostrar el orden de complejidad O en peor caso de los siguientes algoritmos.

(a)
```
bool pertenece(const vector<int> & v, int e, int desde, int hasta){
  // PRE: 0 <= desde <= hasta <= v.size() y v está ordenado.

  if (hasta == desde)
      return false;

  int med = (desde+hasta)/2;
  if (v[med] == e){
      return true;
  }
```

```
  else if(v[med] < e){
      return pertenece(v, e, med  + 1, hasta);
  } else {
      return pertenece(v, e, desde, med);
  }
}
```

(b)
```
int sumatoria(const vector<int> & v, int desde, int hasta){
  // PRE: 0 <= desde <= hasta <= v.size()

  if (hasta == desde)
      return 0;

  int med = (desde+hasta)/2;
  int suma_izq = sumatoria(v, desde, med);
  int suma_der = sumatoria(v, med+1, hasta);
  return suma_izq + suma_der + v[med];
}
```