Apuntes de Teórico

PROGRAMACIÓN 3

Divide & Conquer

Versión 1.1

Índice

Introducción

Sea un problema *P* con una entrada de datos de tamaño *n*. La idea de *Divide & Conquer* consiste en descomponer el problema *P* en subproblemas, con la condición de que cada subproblema sea de la misma clase que el problema *P*.
Luego se resuelve cada subproblema, de manera independiente, y se combinan adecuadamente las subsoluciones de cada subproblema para encontrar la solución al problema *P*.

El criterio para descomponer el problema *P* en subproblemas depende de cada problema en particular, pero lo importante es que cada subproblema sea de la misma clase que *P* y que además tenga una entrada de datos de tamaño menor que *P*, lo implica menos datos para procesar.

Para resolver cada subproblema se debe tener en cuenta el tamaño de la entrada que recibe. Si el tamaño de la entrada es pequeño, entonces es resuelto directamente, en otro caso, como el subproblema es de la misma clase que el problema original, se aplica nuevamente la misma idea, es decir se lo vuelve a descomponer en subproblemas de la misma clase y con entradas de menor tamaño.

Formalmente

Dado un problema *P* con una entrada de tamaño *n*, se descompone *P* (según cierto criterio) en P_i subproblemas con $1 \leq i \leq R$, cada uno de ellos con una entrada de tamaño n_i con $0 \leq n_i < n$. Se resuelve cada uno de los P_i subproblemas (eventualmente aplicando a cada uno de ellos la misma idea de descomposición) y una vez resueltos se combinan las soluciones (según cierto criterio) de cada uno de ellos para hallar la solución al problema *P*.

Esta técnica de diseño de algoritmos puede ser esquematizada por el siguiente algoritmo:

```
Solucion DivideAndConquer(Problema P)
{
      Solucion S;
      Problema[] subproblemas;
      Solucion[] subsoluciones;
      int R;
      if (EsCasoBase(P))
      {
            S = ResolverDirectamente(P);
      }
      else
      {
            //divide P en subproblemas y retorna la cantidad
            R = Dividir(P, subproblemas);
            for(int i = 1; i <= R; i++)
            {
                  subsoluciones[i] = DivideAndConquer(subproblemas[i]);
            }
            S = Combinar(subsoluciones);
      }
      return S;
}
```

Al ser el tamaño de la entrada de cada subproblema menor que la del problema original se garantiza la convergencia al caso base (o los casos bases si los hubiera) donde el problema es resuelto directamente.

Además, a pesar de la iteración (`for`) que aparece en el esquema, generalmente esta técnica de diseño conduce a algoritmos recursivos (aunque esto no implica que no se puedan escribir algoritmos iterativos).

Los criterios para dividir en subproblemas y combinar las subsoluciones son particulares de cada problema a ser resuelto mediante esta técnica.

Ejemplos

Cálculo de números de Fibonacci

$$F(0) = F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

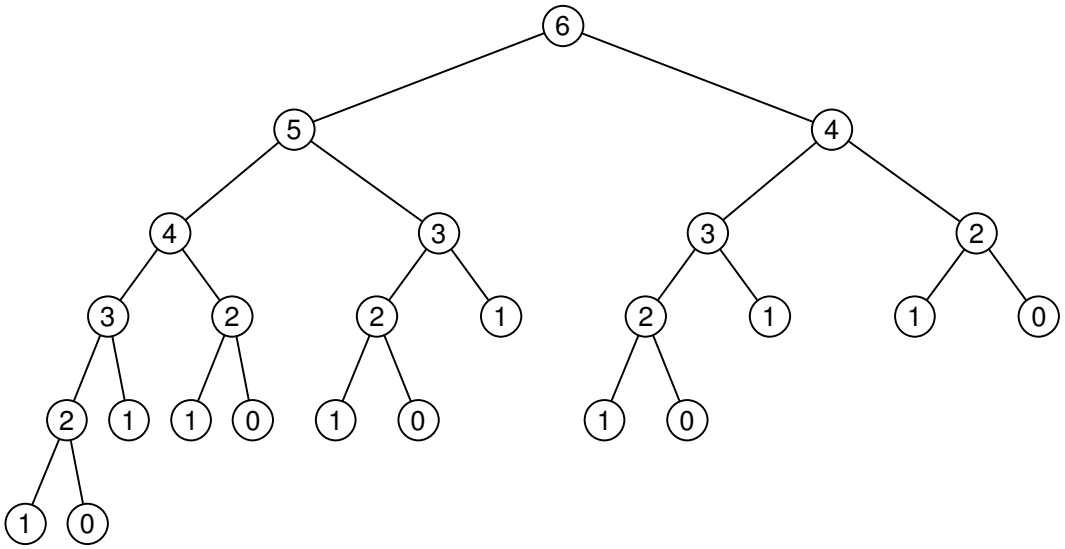
```
int Fibonacci(int n)
{
      if (n == 0) //caso base
      {
            return 1;
      }
      else
      {
            if (n == 1) //caso base
            {
                  return 1;
            }
            else
            {
                  return Fibonacci(n-1) + Fibonacci(n-2); //(1)
            }
      }
}
```

(1) En un solo paso se realiza la descomposición en dos subproblemas de la misma clase (calcular el valor de Fibonacci de un número) y combinación de las soluciones de cada uno de ellos.

Observar que cada subproblema `Fibonacci(n-1)` y `Fibonacci(n-2)` es resuelto de forma independientemente, es decir sin tener en cuenta el otro.

Un posible inconveniente de esta técnica se produce cuando se resuelven independientemente los subproblemas, dado que en algunos casos (como en el ejemplo) se puede estar resolviendo varias veces el mismo subproblema lo cual incide directamente en el orden del tiempo de ejecución de los algoritmos.

Por ejemplo, para calcular *Fibonacci*(6)



Fibonacci(4) se resuelve 2 veces.
Fibonacci(3) se resuelve 3 veces.
Fibonacci(2) se resuelve 5 veces.

Búsqueda binaria

Se quiere determinar si un elemento dado *x* pertenece a un arreglo *A* de tamaño *n*.
Una forma de aplicar *Divide & Conquer* es dividir el arreglo en "mitades" y buscar el elemento *x* en cada una de ellas.

```
bool Busqueda(arreglo A, int inicio, int fin, elemento x)
{
      if (inicio == fin)
      {
            return A[inicio] == x;
      }
      else
      {
            int medio;
            medio = (inicio + fin)/2;
            return Busqueda(A, inicio, medio, x) ||
                  Busqueda(A, medio+1, fin, x);
      }
}
```

el algoritmo se invoca como:

```
bool pertenece = Busqueda(A, 0, n-1, x);
```

Si el arreglo *A* está ordenado, una forma de aplicar *Divide & Conquer*, es haciendo uso de que el arreglo está ordenado y comparar el elemento *x* con el elemento *A[(inicio + fin) / 2]* y si:

- x = *A[(inicio + fin) / 2]* el algoritmo termina retornando verdadero.
- x < *A[(inicio + fin) / 2]* si el elemento *x* pertenece al arreglo, entonces se encuentra en la primer mitad.
- x > *A[(inicio + fin) / 2]* si el elemento *x* pertenece al arreglo, entonces se encuentra en la segunda mitad.

```
bool Busqueda(arreglo A, int inicio, int fin, elemento x)
{
      if (inicio == fin)
            return A[inicio] == x;
      else
      {
            int medio = (inicio + fin) / 2;
            if (x < A[medio])
                  return Busqueda(A, inicio, medio – 1, x);
            else
                  if (x > A[medio])
                        return Busqueda(A, medio + 1, fin, x);
                  else
                        return true;
      }
}
```

el algoritmo se invoca como:

```
bool pertenece = Busqueda(A, 0, n-1, x);
```

Observaciones:

- Se descompone el problema original en un único subproblema cuya entrada es la mitad.
- Al descomponerse en un único subproblema no es necesario la combinación de soluciones.

En general cuando se trabaja con arreglos (de una o varias dimensiones) es conveniente utilizar índices que indiquen entre que posiciones del arreglo el algoritmo va a trabajar.

Multiplicación de matrices

Sean A y B dos matrices cuadradas de dimensiones nxn, el objetivo del problema es encontrar la matriz C (de dimensión nxn), resultado de multiplicar A por B.

Sea c_{ij} el elemento que se encuentra en la fila i y columna j de C, la manera convencional de calcularlo es mediante el producto escalar de la i-ésima fila de A por la j-ésima columna de B,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Otro método para calcular este producto, es el método de Strassen, que consiste en dividir cada matriz en cuatro submatrices de dimensión $\frac{n}{2} x \frac{n}{2}$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

donde por ejemplo, $A_{11} = \begin{bmatrix} a_{11} & \dots & a_{1\frac{n}{2}} \\ \dots & \dots & \dots \\ a_{\frac{n}{2}1} & \dots & a_{\frac{n}{2}\frac{n}{2}} \end{bmatrix}$

y calcular el producto como, $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

donde:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Se puede escribir un algoritmo basado en ***Divide & Conquer***, que calcule el producto de matrices utilizando el método de Strassen, si se asume que n es potencia de 2.

```
const int N = xx;

int Matriz[N][N];

void SumaMat (Matriz A, Matriz B, Matriz &C, int ci,int cj, int n)
{
      int i,j;
      for (i=ci;i<ci+n;i++)
            for (j=cj;j<cj+n;j++)
                  C[i][j]=A[i][j]+B[i][j];
}

void Strassen (Matriz A, Matriz B, Matriz &C, int ai, int aj,
                  int bi, int bj, int n)
{
      Matriz C1,C2;
      int m;
      if (n==1)
            C[ai][bj]=A[ai][aj]*B[bi][bj];
      else
      {
            m=n/2;
            Strassen(A,B,C1,ai,aj,bi,bj,m);
            Strassen(A,B,C2,ai,aj+m,bi+m,bj,m);
            SumaMat(C1,C2,C,ai,bj,m);
            Strassen(A,B,C1,ai,aj,bi,bj+m,m);
            Strassen(A,B,C2,ai,aj+m,bi+m,bj+m,m);
            SumaMat(C1,C2,C,ai,bj+m,m);
            Strassen(A,B,C1,ai+m,aj,bi,bj,m);
            Strassen(A,B,C2,ai+m,aj+m,bi+m,bj,m);
            SumaMat(C1,C2,C,ai+m,bj,m);
            Strassen(A,B,C1,ai+m,aj,bi,bj+m,m);
            Strassen(A,B,C2,ai+m,aj+m,bi+m,bj+m,m);
            SumaMat(C1,C2,C,ai+m,bj+m,m);
      }
}
```