Programación Dinámica

Adaptado de
"Algorithm Design" Goodrich and Tamassia

Calculando la serie de Fibonacci

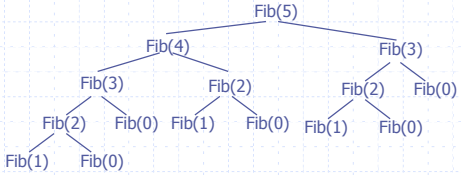
- 1,1,2,3,5,8,13,21,....

$$fib(n) = \begin{cases} 1 & \text{si } n = 0,1 \\ fib(n-1) + fib(n-2) & o.c. \end{cases}$$

Solución recursivo

```
int fib(int n){
  if (n==0 || n==1) return 1;
  return fib(n-1)+fib(n-2);
}
```

- Muy ineficiente pues recalcula el mismo resultado muchas veces.



Solución tabular

```
int fib(int n){
  int * fibTab = new int[n];
  fibTab[0]=fibTab[1]=1;
  for(int i=0;i<=n;i++)
      fibTab[i]=fibTab[i-1]+fibTab[i-2];
  return fibTab[n];
}
```

- Mucho más eficiente!

Solución usando memoización

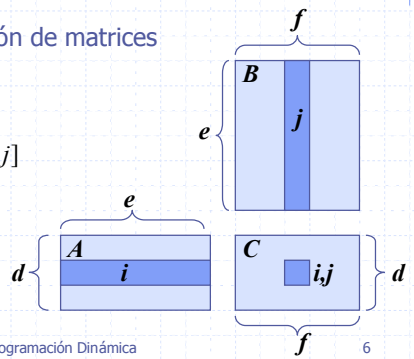
```
int fib(int n){
   int * fibTab = new int[n];
   for(int i=0;i<=n;i++)
       fibTab[i]= -1;
   return lookupFib(fibTab,n);
}
int lookupFib(int fibTab[],int n){
   if (fibTab[n]!= -1) return fibTab[n];
   fibTab[n]=lookupFib(fibTab,n-1)+
             lookupFib(fibTab,n-2);
   return fibTab[n];
}
```


Multiplicación óptima de Matrices

- Problema:
  - Cómo multiplicar una secuencia de matrices de manera óptima?
- Repaso: Multiplicación de matrices
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

  - $O(def)$ time



Matrix Chain-Products

- **Producto encadenado de matrices:**
  - Calcular $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i es $d_i \times d_{i+1}$
  - Problema: Cómo agrupar (poner paréntesis)?
- Ejemplo
  - B es 3×100
  - C es 100×5
  - D es 5×5
  - (B*C)*D necesita 1500 + 75 = 1575 ops
  - B*(C*D) necesita 1500 + 2500 = 4000 ops

Estrategia exhaustiva

- **Algoritmo:**
  - Intentar todas las posibles formas de agrupar $A=A_0*A_1*\ldots*A_{n-1}$
  - Calcular el número de ops para cada una
  - Escoger la mejor
- Tiempo de ejecución
  - El número de parentizaciones es igual al número de árboles binarios con n nodos.
  - Esto es **exponencial**
  - Aproximadamente 4^n (números de Catalán).

Estrategia voraz

- Idea #1: repetidamente seleccionar el producto que use el mayor número de ops.
- Contra-ejemplo:
  - A is 10 x 5
  - B is 5 x 10
  - C is 10 x 5
  - D is 5 x 10
  - Idea voraz #1 da (A*B)*(C*D), que produce 500+1000+500 = 2000 ops
  - A*((B*C)*D) produce 500+250+250 = 1000 ops

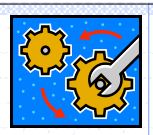
Otra estrategia voraz

- Idea #2: repetidamente seleccionar el producto que use el menor número de ops.
- Contra-ejemplo:
  - A is 101 x 11
  - B is 11 x 9
  - C is 9 x 100
  - D is 100 x 99
  - Idea voraz #2 da A*((B*C)*D)), el cual produce 109989+9900+108900=228789 ops
  - (A*B)*(C*D) produce 9999+89991+89100=189090 ops
- Una estrategia voraz definitivamente no funciona.

Una estrategia "Recursiva"

- Defina **subproblemas**:
  - Encuentre la mejor parentización de $A_i*A_{i+1}*\ldots*A_j$.
  - Sea $N_{i,j}$ el número de operaciones óptimas para este subproblema.
  - La solución óptima del problema total es es $N_{0,n-1}$.
- **Subproblemas óptimos**: La solución óptima puede ser definida en términos de subproblemas óptimos.
  - Tiene que haber una multiplicación final (raíz del árbol de la expresión) para la solución óptima.
  - Digamos que la multiplicación final sea en el índice i: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$.
  - La solución óptima $N_{0,n-1}$ es la suma de 2 subproblemas óptimos, $N_{0,i}$ and $N_{i+1,n-1}$ mas el número de operaciones de la última multiplicación.
  - Sí el óptimo global no tiene estos componentes óptimos nosotros podemos definir una "mejor" solución óptima.

Ecuación recursiva



- El óptimo global se debe definir en términos de subproblemas óptimos, dependiendo en el lugar de la última multiplicación.
- Consideremos todas los posibles lugares para la multiplicación final:
  - Recuerde que A_i es una matriz $d_i \times d_{i+1}$.
  - De manera que una definición recursiva de $N_{i,j}$ es la siguiente:

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- Note que los subproblemas son no independientes -- los **subproblemas se sobreponen**.

Un algoritmo de programación dinámica

- Puesto que los subproblemas se sobreponen, no se usa recursión.
- En cambio, se construyen los subproblemas óptimos de abajo hacia arriba "bottom-up."
- Los $N_{i,i}$'s son fáciles, de manera que se empieza con ellos.
- Seguir con subproblemas de longitud 2,3,... etc.
- Tiempo de ejecución: $O(n^3)$

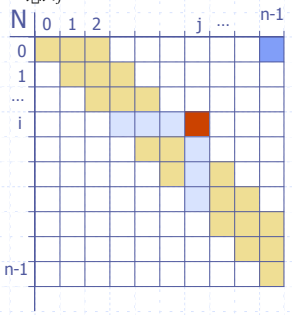
Algorithm *matrixChain*(*S*):
Input: sequence *S* of *n* matrices to be multiplied
Output: number of operations in an optimal paranethization of *S*
for *i* ← 1 **to** *n-1* **do**
  $N_{i,i} \leftarrow 0$
for *b* ← 1 **to** *n-1* **do**
  for *i* ← 0 **to** *n-b-1* **do**
    j ← *i+b*
    $N_{i,j} \leftarrow +\infty$
    for *k* ← i **to** *j-1* **do**
      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

Algoritmo en acción

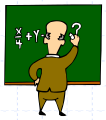
$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

solución

- La estrategia bottom-up llena la matriz N por diagonales
- $N_{i,j}$ obtiene valores basados en valores previos en la i-ésima fila y la j-ésima columna
- La obtención de la parentización óptima puede ser hecho recordando la "k" correspondiente a cada casilla



Técnica General de Programación Dinámica



- Aplica a un problema que inicialmente pareciera requerir una gran cantidad de tiempo (posiblemente exponencial) y que debe cumplir:
  - **Subproblemas simples:** los subproblemas puden ser definidos en términos de pocas variables.
  - **Optimalidad de subproblemas:** El óptimo global puede ser definido en términos de subproblemas óptimos
  - **Sobrelapamiento de subproblemas:** Los problemas no son independientes (por lo tanto, la solución debe construirse de abajo hacia arriba).