Recurrencia

Práctica A2: sesiones 03 y 04

Lógica y Algorítmica

Resumen

En esta práctica vamos a resolver un problema usando dos algoritmos: uno iterativo y otro recursivo, y compararemos el tiempo de ejecución de ambos para poder elegir el más adecuado. Los pasos a seguir son similares a los reaizados en la práctica anterior:

1. **Implementar** los algoritmos en Java y pasar los juegos de pruebas JUNIT.
2. Realizar diferentes experimentos partiendo de un valor inicial para el tamaño de la entrada n, doblando el tamaño de la entrada en cada nuevo experimento, y obtener los tiempos medios de 10 ejecuciones de cada método con los mismos datos de entrada, del mismo modo que en la práctica A1.
3. Análisis de datos: elaborar **tablas** comparativas con los resultados de los tiempos medios, obtención de las **gráficas** y las tasas de crecimiento.

Se incluye un apartado final con una introducción al paradigma de **Programación Dinámica** que ha de usarse en la sesión 04.

1. Sesión 03. Métodos a evaluar: Exponenciación

Dados dos números enteros a y n queremos calcular a^n .

1. **Implementar los siguientes Algoritmos:**

   a) Algoritmo Iterativo. Algoritmo fuerza bruta: multiplicar a por sí mismo n veces.
   Nota: en este algoritmo NO podemos usar el método `Math.pow()` de Java. Hay que hacer un bucle for.

   b) Algoritmo Recursivo. Algoritmo Divide y Vencerás. Está en $O(\log n)$. Se basa en que:
   - $a^n = (a^{\frac{n}{2}})^2$ si n par
   - Produce la recurrencia:

$$a^n = \begin{cases} a & \text{si } n = 1 \\ (a^{\frac{n}{2}})^2 & \text{si } n \text{ es par} \\ a.a^{n-1} & \text{si } n \text{ es impar} \end{cases}$$

   Ejemplo: $a^{29} = a.a^{28} = a.(a^{14})^2 = a((a^7)^2)^2 = ...$

2. **Clases que hay que implementar:** paquete `org.lya.sesion03`

   a) Clase `Potencia.java` :
   - campos privados para **base**, **exponente**. La **base** será un valor positivo mayor que cero y el **exponente** un número mayor o igual que 0.
   - Dos constructores:
     - `Potencia ()` constructor por defecto
     - `Potencia(int a, int n)` : debe comprobar que los valores de **a** y **n** son correctos.
   - Y los métodos:
     - `double exponenFuerzaBruta()`
     - `double exponenRecursivoDyV()`: desde este método se llamará al método recursivo por primera vez :

```
private double exponenRecursivoDyV(int a, int n)
```

b) Clase `TiemposPotencia.java` :

Contiene el `main()` con las pruebas para diferentes tamaños de exponente.

- Crear un programa principal, en el que se calcule a^n, a un valor fijo.

- **Para una MISMA potencia**: Ejecutar **10 veces cada algoritmo** y calcular el tiempo medio de ejecución (descartando el valor máximo) que ha empleado cada uno de ellos. Es decir, **para cada potencia de exponente** n obtenemos tiempo medio del método de Fuerza Bruta y el tiempo medio del método DyV.

- Realizar las pruebas **duplicando** el tamaño del exponente: 64,128, 256, 512, 1024, 2048, 4096, 8192, ... (si aumentamos n podemos tener algún problema de overflow).

- Nota: medir el tiempo en **nanosegundos**.

2. Sesión 03. Análisis de los datos

1. Elaborar una tabla tomando la base = 1, con los tiempos medios obtenidos con los diferentes exponentes, **para cada algoritmo**:

n	$\bar{T}_{Iter}(n)$	$\bar{T}_{Recur}(n)$
64	-	-
128	-	-
256	-	-
512	-	-
1024	-	-
2048	-	-
4096	-	-
8192	-	-
...	-	-

2. A partir de esta tabla obtener una gráfica en la que representamos los tiempos de los dos algoritmos: tiempo de ejecución *vs* tamaño del exponente.

   Dibujar una curva para cada algoritmo, e incluir ambas en la misma gráfica.

3. Obtener de forma teórica el orden de complejidad del **algoritmo iterativo**.

4. Consultar la bibliografía indicada en aula virtual para obtener el orden de complejidad del algoritmo recursivo. Conclusiones sobre la eficiencia de ambos algoritmos, especialmente el algoritmo recursivo.

3. Sesión 04. Métodos a evaluar: Calcular coeficiente binomial

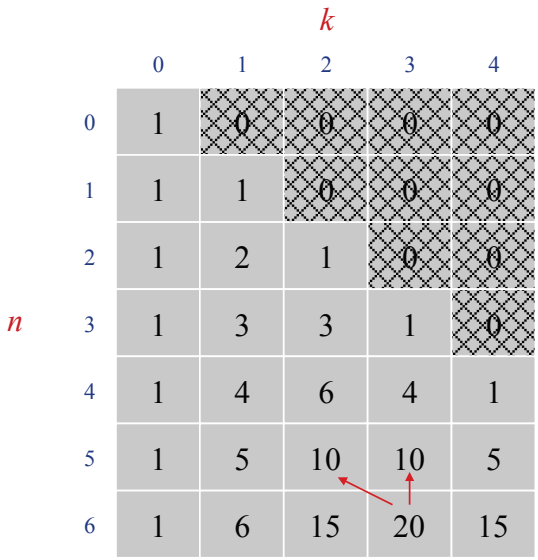
Se define como:

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0\ , \quad k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{si } n = 0 \end{cases}$$

1. **Algoritmos:**

   a) Algoritmo básico recursivo: aplica la definición de forma recursiva.

   b) Algoritmo basado en Programación Dinámica: usa una matriz para almacenar los valores ya calculados. (Ver Anexo)
   Por ejemplo, esta sería la matriz que se crea al calcular $\binom{6}{4}$:



$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$20 = 10 + 10$

binomial(n, k)

   Podemos comprobar que:

   - La matriz tiene **7 filas y 5 columnas**
   - Almacena $\binom{n}{k}$ en matriz[n][k]
   - Se incluye caso base: $\binom{n}{k} = 1 \quad$ si $k = 0$ (primera columna)
   - Se incluye caso base: $\binom{n}{k} = 0 \quad$ si $n = 0$ (primera fila)
   - A partir de la ecuación recurrente calculamos:
     matriz[n][k] = matriz[$n-1$][$k-1$] + matriz[$n-1$][k]
   - El resultado está en matriz[6][4]

2. **Clases que hay que implementar: Paquete** `org.lya.sesion04`

   a) `CoeficienteBin.java` :

   - campos privados para n y k (de tipo int) : *cbN* y *cbK*
   - Métodos de acceso `getN()`, `getK()` , `setN()`, `setK()`.
   - Constructor:
     - `CoeficienteBin (int n, int k)` tiene que comprobar que $k \leq n$ y lanzar una excepción `IllegalArgumentException` si no es así.

- Métodos que implementan los dos algoritmos y devuelven el resultado:
  - `int coefBinomialRecursivo()` que comprueba errores y llama al método privado `int coefBinomialRecursivo(int n, int k)`
  - `int coefBinomialProgDinam()` que calcula el valor a partir de una matriz de valores de casos más pequeños de n y k

b) Clase `TiemposCoeficienteBin.java` : con el `main()`

- Crear un programa principal que muestre el tiempo de ejecución de los dos algoritmos para el mismo coeficiente.

- Los coeficientes de entrada los consideramos del modo $\binom{2n}{n}$.

- Realizar las pruebas con coeficientes para valores de n = 4,5,6, 7, 8, 9, 10, .. 25.

4. Sesión 04. Análisis de los datos

1. Elaborar una tabla con los tiempos medios obtenidos para ambos algoritmos:

CoeficienteBin $(2n, n)$	$\bar{T}_{Recursivo}$	$\bar{T}_{Prog.Din}$
(8,4)	-	-
(10,5)	-	-
(12,6)	-	-
(14,7)	-	-
...	-	-
...	-	-

2. A partir de esta tabla, obtener una gráfica representando los tiempos de ejecución de ambos algoritmos frente a n (las dos curvas en la misma gráfica).
3. Consultar los apuntes e indicar y justificar cuál es el orden de complejidad de cada algoritmo.
4. Demostrar que los resultados experimentales obtenidos para el algoritmo de **programación dinámica** coinciden con los teóricos. Para ello debe recopilar datos de tiempos medios y realizar las gráficas (como en la práctica A1) ejecutando SOLO el método `coefBinomialProgDinam()` desde n =32, doblando el tamaño de la entrada, hasta un tamaño de n =8192.
5. El **método recursivo** con los valores de entrada que hemos considerado, es de orden $O(4^n)$. Obtener la constante c de la última fila de la tabla de tiempos, e indicar la expresión de $T(n)$ para este método con esta implementación.
6. En base a la expresión de $T(n)$ ¿cuanto tardaría (teóricamente) cada algoritmo en calcular $\binom{1000}{500}$? El tiempo del algoritmo recursivo se debe indicar en años.
7. Conclusiones sobre la eficiencia de ambos algoritmos.

Entrega: Crear un documento con las soluciones a las preguntas del análisis de datos DE LAS DOS SESIONES en **formato pdf** con el nombre PA2_Apellido1Apellido2Nombre.pdf.

☞ El código se sube al repositorio svn. Debe pasar los tests de pruebas.

Anexo. Paradigma de Programación Dinámica

- Resuelve el problema original **combinando las soluciones para subproblemas más pequeños**, igual que la técnica divide y vencerás.
- Sin embargo, la programación dinámica **no utiliza recursividad** sino que, a medida que se resuelven los subproblemas, almacena los resultados en una **matriz**.
- Con esto se pretende evitar el problema de la técnica divide y vencerás de calcular varias veces las soluciones de problemas pequeños.
- Ejemplo. Sucesión de Fibonacci:
  - Con **Divide y Vencerás** ➜ $T(n) \in \Phi^n$

```
funcion FibREC(n)
    IF n < 2 devolver n
    ELSE
     devolver FibREC(n − 1) + FibREC(n − 2)
```

  - Con **Programación Dinámica** ➜ $T(n) \in \Theta(n)$

```
funcion FibProgDIN(n)
    T[0] ← 0;
    T[1] ← 1;
    FOR i ← 2 TO n
       T[i] ← T[i − 1] + T[i − 2]
    Devolver T[n]
```

    ☞ Calcula los valores de menor a mayor, empezando por 0, y los va guardando en una matriz.

- **Elementos de un algoritmo de Programación Dinámica**
  - La programación dinámica resuelve los **subproblemas generados** de **forma NO recursiva**, guardando los valores computados en un **matriz**.
  - En un algoritmo de P.D. es necesario definir:
    - **matriz** utilizada por el algoritmo, su tamaño y cómo se calcula.
    - **Ecuación recurrente**: para calcular la solución de un problema. grande en función de instancias más pequeñas del mismo problema vamos a ir rellenando la matriz a partir de lo que se ha calculado en filas y/o columnas anteriores.
    - **Casos base**, o cómo inicializamos los primeros valores de la matriz.
    - Especificar en qué posición de la matriz se encuentra la **solución final**.
- **Tiempo de ejecución de los algoritmos de Prog. Dinámica:**

  Es el producto de:

  Tamaño de la matriz $*$ Tiempo de rellenar cada elemento de la matriz

- Ejemplo. Cálculo de Coeficientes Binomiales. Solución recursiva

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0\ , \quad k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{otro caso} \end{cases}$$

  - Solución recursiva ineficiente: ➜ $T(n,k) \in \Omega(\binom{n}{k})$

```
funcion BinomREC(n, k)
        IF( k = 0 OR k = n)    Devolver 1
        ELSE
          Devolver( BinomREC(n − 1,k − 1) + BinomREC(n − 1,k))
```

- Caso particular: si tomamos los valores de entrada como $\binom{2k}{k}$
  BinomREC$(2n, n)$ tiene un **orden exponencial** $\approx c \times 4^n$
  ya que $T(n+1)/T(n) \approx 4$

- Ejemplo. Coeficientes Binomiales. Solución NO recursiva

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0\ , \quad k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{otro caso} \end{cases}$$

- Solución con programación dinámica:
  Esta sería la matriz CB donde vamos almacenando los resultados:

	0	1	2	...	$k-1$	k
0	1	0	0	...	0	0
1	1	1	0	0	0	0
2	1	2	1	0	0	0
..	1					
$n-1$	1				•	•
n	1					$CB[n,k]$

  - Tamaño de la matriz $= (n+1) \times (k+1)$, la llamaremos CB
  - Ecuación recurrente:
    $$CB[n,k] = CB[n-1,k-1] + CB[n-1,k]$$
  - Casos Base:
    para $k = 0$, $CB[n,0] = 1$ y para $k > n$, $CB[0,k] = 0$
  - Solución final: en la matriz en la posición $CB[n,k]$

☞ **Cómo implementar con programación dinámica el algoritmo que calcula $\binom{n}{k}$:**

1. Reservamos memoria para una matriz de enteros de dimensiones $(n+1) \times (k+1)$.
2. Se inicializan los casos base:
   - primera columna: $(k = 0)$ vale 1.
   - primera fila: $(k > n)$ vale 0.
3. Después calculamos el resto de los valores, rellenando la matriz por FILAS, de izquierda a derecha, aplicando la ecuación recurrente.
4. El resultado buscado está en la última posición de la matriz: $CB[n,k]$.