Lógica y Algorítmica
Práctica 8
Recurrencia

Resumen

En esta práctica vamos a resolver un mismo problema usando un algoritmo iterativo y otro recursivo, y compararemos el tiempo de ejecución de ambas soluciones. Los pasos a seguir son:

1. **Implementar** los algoritmos en Java.
2. Realizar diferentes experimentos, doblando el tamaño de la entrada, y obtener los tiempos medios de ejecución de los algoritmos del mismo modo que en la práctica 7.
3. Análisis de datos: elaborar **tablas** comparativas con los resultados de los tiempos medios, y obtener las **gráficas** de las tasas de crecimiento.

1. Sesión 08. Exponenciación

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

Implementar los siguientes Algoritmos:

1 Algoritmo Iterativo. Algoritmo fuerza bruta: multiplicar a por sí mismo n veces.

Nota: NO podemos implementar este algoritmo como una llamada al método `Math.pow()` de Java.

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

Clase que necesitamos:

- El paquete para este ejercicio es: `org.lya.sesion08`
- Clase `Potencia.java` :

  Propiedades:
  - campos privados para **base**, **exponente**. La **base** será un valor positivo entre 2 y 10.
  - Dos constructores:

    - `Potencia ()` constructor por defecto

    - `Potencia(int a, int n)`

- 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)`

- Clase `TiemposPotencia.java` : con el `main()` y las pruebas para diferentes tamaños de exponente.

  - Crear un programa principal, en el que se calcule a^n, a un valor fijo, por ejemplo 2.

  - **Para una MISMA potencia**: Ejecutar **10 veces cada algoritmo** y calcular el tiempo medio de ejecución que ha empleado cada uno de ellos. Es decir, **para cada potencia de exponente** n **obtenemos 2 tiempos medios**.

  - 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.

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 el libro de Brassard para obtener el orden de complejidad del algoritmo recursivo. Conclusiones sobre la eficiencia de ambos algoritmos, especialmente el algoritmo recursivo.

2. Sesión 09. Calcular coeficiente binomial

Se define como:

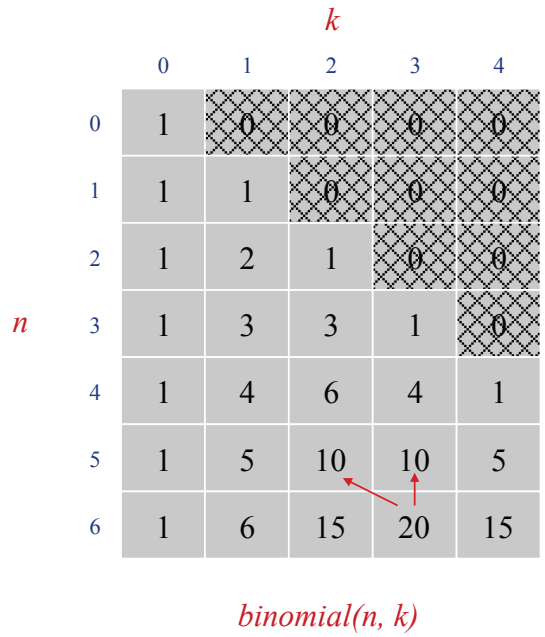
$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0\ , \quad k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{otro caso } (k > n) \end{cases}$$

Algoritmos:

1 Algoritmo básico recursivo: aplica la definción de forma recursiva.

2 Algoritmo basado en Programación Dinámica: usa una tabla para almacenar los valores ya calculados. (Ver Anexo)

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

Clases que necesitamos:

- **Paquete para este ejercicio:** `org.lya.sesion09` .

- `CoeficienteBin.java` :

  Propiedades:

  - campos privados para **N** y **K** (de tipo int) : $K \leq N$
  - 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:

    - `long coefBinomialRecursivo()` que llama a `coefBinomialRecursivo(int n, int k)`
    - `long coefBinomialProgDinam()` que calculan el valor y devuelven el resultado.

- 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.

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 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.
5. Demostrar que el método recursivo, con los valores de entrada que hemos considerado, es de orden $O(4^N)$.
6. ¿Cuanto tardaría cada algoritmo en calcular $\binom{900}{450}$?
7. Conclusiones sobre la eficiencia de ambos algoritmos.

Nota: Incluir las tablas y las soluciones a las preguntas del análisis de datos DE LAS DOS SESIONES en un solo documento pdf llamado practica8- seguido de tu DNI.

Anexo. Paradigma de Programación Dinámica

- Igual que la técnica divide y vencerás, resuelve el problema original **combinando las soluciones para subproblemas más pequeños**
- Sin embargo, la programación dinámica **no utiliza recursividad** sino que, a medida que se resuelven los subproblemas, almacena los resultados en una **tabla**
- Con esto se pretende evitar el problema de 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 tabla*

Elementos de un algoritmo de Programación Dinámica

- La programación dinámica resuelve los **subproblemas generados** de **forma NO recursiva**, guardando los valores computados en un **tabla**.
- En un algoritmo de P.D. es necesario definir:
  - **Tabla** 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 tabla a partir de lo que se ha calculado en filas y/o columnas anteriores.
  - **Casos base**, o como inicializamos los primeros valores de la tabla
  - Especificar en que posición de la tabla se encuentra la **solución final**

Tiempo de ejecución de los algoritmos de P.D.

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

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))
```

- Ejemplo: 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 tabla 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 tabla $= (n+1) \times (k+1)$, 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 $CB[n,k]$

Cómo implementar el algoritmo que calcule $\binom{n}{k}$:

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