Guía Detallada de Análisis de Algoritmos Iterativos y Recursivos

Basado en la Práctica 4 de AyED - UNLP

1. Análisis de Algoritmos Iterativos (Bucles)

El análisis de algoritmos iterativos se centra en determinar el número de veces que se ejecuta una operación fundamental. Este conteo se realiza analizando las condiciones de inicio, parada e incremento de las variables de control de los bucles.

1.1. Caso 1: Complejidad Logarítmica - $O(\log n)$

1.1.1. Patrón Identificativo

La variable de control del bucle no avanza de forma lineal (ej. 'i++' o 'i–'), sino que sufre una **multiplicación o división** por una constante mayor a 1 en cada iteración.

1.1.2. Análisis Detallado

Consideremos un bucle donde la variable se duplica en cada paso. Los valores que tomará serán $1, 2, 4, 8, \ldots, 2^k$. El bucle se detendrá cuando $2^k \geq n$. Para encontrar el número de iteraciones k, aplicamos logaritmo en base 2:

$$\log_2(2^k) = \log_2(n) \implies k = \log_2(n)$$

El número de operaciones es proporcional al logaritmo de n.

1.1.3. Ejemplo del PDF (Ejercicio 8)

```
int c = 1;
while (c < n) {
  // algo_de_O(1);
  c = 2 * c;
}
```

El análisis es el descrito anteriormente, resultando en una complejidad de $\mathbf{O}(\log \mathbf{n})$.

1.2. Caso 2: Complejidad Lineal - $O(n)$

1.2.1. Patrón Identificativo

La variable de control avanza en una cantidad constante (ej. 'i++', 'i+=2') hasta un límite que es una función lineal de n.

1.2.2. Análisis Detallado

Si un bucle se ejecuta desde un punto inicial hasta un límite $C \cdot n$, donde C es una constante, el número de iteraciones será proporcional a n. En la notación Asintótica (Big-O), las constantes multiplicativas se descartan.

1.2.3. Ejemplo del PDF (Ejercicio 7)

```
for(int k=0; k < n+n+n; ++k) {
  c[k] = c[k] + sum;
}
```

El bucle se ejecuta $3n$ veces. El tiempo de ejecución $T(n) = 3n$, lo que pertenece a $\mathbf{O(n)}$.

1.3. Caso 3: Complejidad Cuadrática - $O(n^2)$

1.3.1. Patrón Identificativo

Generalmente, un bucle anidado dentro de otro, donde ambos dependen de n. El caso más interesante es cuando el bucle interno depende del externo.

1.3.2. Análisis Detallado (Bucle Dependiente)

Este es un caso fundamental que aparece con frecuencia. Consideremos el siguiente ejemplo del PDF (Ejercicio 8):

```
for (i=1; i < n; i=i+2)    // Bucle Externo
  for (j=1; j <= i; j++)     // Bucle Interno
    r = r + 1;
```

El bucle externo se ejecuta aproximadamente $n/2$ veces. El bucle interno se ejecuta i veces. Para calcular el número total de operaciones, debemos resolver la siguiente sumatoria:

$$T(n) = \sum_{k=1}^{n/2} (2k - 1)$$

Una aproximación más sencilla es considerar un bucle que avanza de uno en uno:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} 1$$

La suma interna, $\sum_{j=1}^{i} 1$, es simplemente i. Ahora resolvemos la suma externa:

$$T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

El término dominante de esta expresión es n^2, por lo que la complejidad es $\mathbf{O(n^2)}$.

1.3.3. Ejemplo Complejo Adicional (Bucle Polinomial)

Consideremos un bucle anidado donde los límites son potencias de n:

```
for (int i = 1; i <= n*n; i++) {
    for (int j = 1; j <= i; j++) {
        // operacion O(1)
    }
}
```

El análisis es similar al anterior, pero el límite del bucle externo es n^2.

$$T(n) = \sum_{i=1}^{n^2} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n^2} i$$

Usando la fórmula de la suma, pero con n^2 como límite superior:

$$T(n) = \frac{n^2(n^2+1)}{2} = \frac{n^4+n^2}{2}$$

La complejidad de este algoritmo es $\mathbf{O(n^4)}$.

2. Análisis de Algoritmos Recursivos

Se analiza la función de coste $T(n)$ expandiendo la recurrencia para visualizar el patrón de trabajo total que se genera en el árbol de recursión.

2.1. Sub-sección: Decremento en Constante ($T(n-c)$)

2.1.1. Patrón 1: Una Sola Llamada - Complejidad Lineal $O(n)$

La función se llama a sí misma una vez, reduciendo el problema en una cantidad constante.

Fórmula Típica (Ejercicio 13):

$$T(n) = \begin{cases} c_0 & \text{si } n \leq 1 \\ T(n-1) + c_1 & \text{si } n > 1 \end{cases}$$

Análisis por Expansión:

$$\begin{aligned} T(n) &= T(n-1) + c_1 \\ &= (T(n-2) + c_1) + c_1 = T(n-2) + 2c_1 \\ &= (T(n-3) + c_1) + 2c_1 = T(n-3) + 3c_1 \\ &\vdots \\ &= T(n-k) + k \cdot c_1 \end{aligned}$$

La recursión se detiene cuando $n - k = 1$, es decir, $k = n - 1$.

$$T(n) = T(1) + (n-1)c_1 = c_0 + (n-1)c_1$$

El término dominante es n, por lo tanto, la complejidad es $\mathbf{O(n)}$.

2.1.2. Patrón 2: Múltiples Llamadas - Complejidad Exponencial $O(a^n)$

La función se llama a sí misma más de una vez.

Fórmula Típica (Ejercicio 13):

$$T(n) = \begin{cases} c_0 & \text{si } n \leq 1 \\ 2T(n-1) + c_1 & \text{si } n > 1 \end{cases}$$

Análisis por Expansión:

$$\begin{aligned} T(n) &= 2T(n-1) + c_1 \\ &= 2(2T(n-2) + c_1) + c_1 = 4T(n-2) + 2c_1 + c_1 \\ &= 4(2T(n-3) + c_1) + 3c_1 = 8T(n-3) + 4c_1 + 2c_1 + c_1 \end{aligned}$$

El número de llamadas recursivas en el nivel k es 2^k. Esto genera un crecimiento exponencial. El número total de operaciones es dominado por el número de hojas en el árbol de recursión, que es 2^n. La complejidad es $\mathbf{O(2^n)}$.

2.2. Sub-sección: División por Constante ($T(n/b)$)

2.2.1. Patrón 1: Una Sola Llamada - Complejidad Logarítmica $O(\log n)$

La función reduce el problema a una fracción de su tamaño original.

Fórmula Típica (Ejercicio 13):

$$T(n) = \begin{cases} c_0 & \text{si } n = 1 \\ T(n/2) + c_1 & \text{si } n > 1 \end{cases}$$

Análisis por Expansión:

$$\begin{aligned} T(n) &= T(n/2) + c_1 \\ &= (T(n/4) + c_1) + c_1 = T(n/4) + 2c_1 \\ &\vdots \\ &= T(n/2^k) + k \cdot c_1 \end{aligned}$$

La parada ocurre cuando $n/2^k = 1 \implies k = \log_2(n)$.

$$T(n) = T(1) + (\log_2 n) \cdot c_1$$

La complejidad es $\mathbf{O(\log n)}$.

2.2.2. Patrón 2: Múltiples Llamadas - Complejidad Lineal $O(n)$

La función se divide en ‘a‘ subproblemas de tamaño ‘n/b‘.

Fórmula Típica (Ejercicio 13):

$$T(n) = \begin{cases} c_0 & \text{si } n = 1 \\ 2T(n/2) + c_1 & \text{si } n > 1 \end{cases}$$

Análisis por Árbol de Recursión: Visualizamos el trabajo en un árbol.

- **Nivel 0 (Raíz):** Costo de c_1. 1 problema de tamaño n.
- **Nivel 1:** 2 subproblemas de tamaño $n/2$. Costo total del nivel: $2 \cdot c_1$.
- **Nivel 2:** 4 subproblemas de tamaño $n/4$. Costo total del nivel: $4 \cdot c_1$.
- **Nivel i:** 2^i subproblemas de tamaño $n/2^i$. Costo total del nivel: $2^i \cdot c_1$.

La profundidad del árbol es $k = \log_2(n)$. El costo total es la suma de los costos de todos los niveles más el costo de las hojas.

$$T(n) = \sum_{i=0}^{k-1} 2^i c_1 + (\text{Costo de las hojas})$$

El número de hojas es $2^k = 2^{\log_2 n} = n$. El costo de las hojas es $n \cdot T(1) = n \cdot c_0$. La suma es una serie geométrica: $c_1 \sum_{i=0}^{k-1} 2^i = c_1 \frac{2^k - 1}{2-1} = c_1(n-1)$.

$$T(n) = c_1(n-1) + c_0 n$$

Esta expresión es una función lineal de n, por lo que la complejidad es $\mathbf{O(n)}$.

3. Análisis de Casos de Estudio Complejos (Ejercicio 5)

A continuación, se analizan en profundidad los tres algoritmos propuestos en el Ejercicio 5 para generar una permutación aleatoria de n enteros. Este análisis revela diferencias sutiles pero críticas en su eficiencia.

3.1. Análisis de `randomUno`

```
public static int[] randomUno (int n) {
    int i, x=0, k;
    int[] a = new int[n];
    for (i=0; i<n; i++) {
        boolean seguirBuscando = true;
        while (seguirBuscando) {
            x = ran_int(0, n-1);
            seguirBuscando = false;
            for (k=0; k < i && !seguirBuscando; k++)
                if (x == a[k])
                    seguirBuscando = true;
        }
        a[i] = x;
    }
    return a;
}
```

Estrategia del Algoritmo: Para cada posición ‘i‘ del arreglo, genera un número aleatorio ‘x‘. Luego, verifica si ‘x‘ ya ha sido insertado previamente en las posiciones ‘0‘ a ‘i-1‘. Si ya fue usado, repite el proceso hasta encontrar un número no utilizado.

Análisis de Complejidad: El algoritmo tiene una estructura de tres bucles anidados.

- **Bucle externo (‘for i‘):** Se ejecuta n veces, una para cada elemento del arreglo.
- **Bucle intermedio (‘while‘):** Su número de ejecuciones es probabilístico. Al principio, cuando ‘i‘ es pequeño, es muy probable encontrar un número no utilizado a la primera. Sin embargo, cuando ‘i‘ se acerca a ‘n-1‘, la mayoría de los números ya han sido usados, y puede tomar muchos intentos generar el número correcto.
- **Bucle interno (‘for k‘):** Este bucle verifica la duplicidad. Se ejecuta ‘i‘ veces en el peor caso.

El trabajo realizado para insertar el elemento en la posición ‘i‘ es el producto del número de intentos del ‘while‘ y el costo de la verificación ‘O(i)‘. En el peor caso, para llenar la última posición, podríamos tardar un número muy grande de intentos. Sin embargo, un análisis simple del peor caso de los bucles deterministas (los dos ‘for‘) nos da una pista:

$$T(n) \approx \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \implies O(n^2)$$

Esta es una cota inferior. La verdadera complejidad en el peor caso es mucho mayor y podría no terminar si el generador aleatorio es "desafortunado". En la práctica, el rendimiento se degrada drásticamente a medida que el arreglo se llena. Es un algoritmo muy ineficiente, con una complejidad de caso promedio superior a $\mathbf{O(n^2)}$.

3.2. Análisis de `randomDos`

```
public static int[] randomDos (int n) {
    int i, x;
    int[] a = new int[n];
    boolean[] used = new boolean[n];
    for (i=0; i<n; i++) used[i] = false;
    for (i=0; i<n; i++) {
        x = ran_int(0, n-1);
        while (used[x]) x = ran_int(0, n-1);
        a[i] = x;
        used[x] = true;
    }
    return a;
}
```

Estrategia del Algoritmo: Utiliza un arreglo booleano auxiliar 'used' para marcar qué números ya han sido seleccionados. Esto evita el costoso bucle de verificación de 'randomUno'.

Análisis de Complejidad:

- **Inicialización ('for' y 'new'):** La creación e inicialización de los arreglos 'a' y 'used' tiene un costo de $\mathbf{O(n)}$.
- **Bucle principal ('for i'):** Se ejecuta n veces.
- **Bucle interno ('while'):** Al igual que en 'randomUno', el número de iteraciones es probabilístico. Genera un número 'x' y comprueba 'used[x]'. Esta comprobación es $\mathbf{O(1)}$, una mejora fundamental sobre el $\mathbf{O(i)}$ de 'randomUno'.

El análisis del 'while' se relaciona con el **Problema del coleccionista de cupones**. El número esperado de intentos para obtener el k-ésimo cupón nuevo es $n/(n-k+1)$. El tiempo total esperado es la suma de estos intentos para todos los elementos:

$$T(n) = n \sum_{k=1}^{n} \frac{1}{k} = n \cdot H_n$$

Donde H_n es el n-ésimo número armónico, que se aproxima a $\ln(n)$. Por lo tanto, la complejidad de caso promedio de este algoritmo es $\mathbf{O(n \log n)}$. Aunque es una gran mejora, el peor caso sigue siendo teóricamente infinito.

3.3. Análisis de `randomTres`

```
public static int[] randomTres (int n) {
    int i;
    int[] a = new int[n];
    for (i=0; i<n; i++)
        a[i] = i;
    for (i=1; i<n; i++)
        swap(a, i, ran_int(0, i-1));
    return a;
}
```

Estrategia del Algoritmo: Este es el moderno y eficiente algoritmo de **Fisher-Yates shuffle**. Primero, inicializa el arreglo con los números del 0 a $n-1$ en orden. Luego, para cada posición 'i', intercambia el elemento en 'a[i]' con un elemento en una posición aleatoria anterior a 'i' (incluida la propia posición 'i' en algunas variantes).

Análisis de Complejidad:

- **Inicialización ('for'):** Llena el arreglo con valores secuenciales. Este paso tiene una complejidad de $\mathbf{O(n)}$.
- **Bucle de barajado ('for i'):** Se ejecuta $n-1$ veces.
- **Operaciones internas:** Dentro del bucle, se llama a 'ran$_i$*nt*'*ya*'*swap*'.*Ambasoperacionessondetie*

El costo total es la suma del costo de inicialización y el costo del bucle de barajado:

$$T(n) = O(n) + O(n) \cdot O(1) = O(n)$$

La complejidad total es $\mathbf{O(n)}$. Este algoritmo no solo es el más eficiente en términos de tiempo, sino que también es determinista en su finalización y garantiza una permutación uniformemente aleatoria.