Análisis de eficiencia de algoritmos

Jorge Mario Londoño Peláez & Varias AI

February 19, 2025

1 Análisis de Algoritmos

1.1 Eficiencia de algoritmos

La eficiencia de un algoritmo es una medida de la cantidad de recursos computacionales (tiempo y espacio) que consume al resolver un problema de un tamaño determinado. Evaluar la eficiencia de los algoritmos es crucial por varias razones:

- **Optimización de recursos:** Permite utilizar los recursos computacionales de manera más efectiva, lo que se traduce en un menor costo y un mejor rendimiento.
- **Escalabilidad:** Ayuda a predecir cómo se comportará un algoritmo a medida que aumenta el tamaño del problema, lo que es fundamental para aplicaciones que deben manejar grandes cantidades de datos.
- **Comparación de algoritmos:** Facilita la comparación de diferentes algoritmos para resolver el mismo problema, lo que permite elegir el más adecuado para una situación específica.

Las principales métricas de eficiencia son:

- **Tiempo:** La cantidad de tiempo que tarda un algoritmo en completar su ejecución. Se mide en unidades de tiempo (segundos, milisegundos, etc.) o en términos de operaciones elementales (comparaciones, asignaciones, etc.).
- **Espacio:** La cantidad de memoria que utiliza un algoritmo durante su ejecución. Se mide en unidades de memoria (bytes, kilobytes, megabytes, etc.).

Existen dos metodologías básicas para medir la eficiencia de un algoritmo:

- **Experimental:** Consiste en implementar el algoritmo y ejecutarlo con diferentes entradas para medir su tiempo y espacio de ejecución.
- **Analítica:** Consiste en analizar el algoritmo para determinar su tiempo y espacio de ejecución en función del tamaño de la entrada.

1.2 Descripción de la eficiencia por medio de un modelo analítico

Los modelos de eficiencia se expresan como funciones que relacionan el tamaño de la entrada con la cantidad de recursos consumidos. Estas funciones tienen la forma:

$$T : \mathbb{N} \to \mathbb{R}^+$$

Donde:

- T representa la función de eficiencia (tiempo o espacio).
- $\mathbb{N}$ representa el conjunto de los números naturales, que se utiliza para modelar el tamaño de las entradas.
- $\mathbb{R}^+$ representa el conjunto de los números reales positivos, que se utiliza para modelar la cantidad de recursos consumidos.

La variable independiente de la función T es el tamaño de la entrada, que se denota por n. El tamaño de la entrada puede ser, por ejemplo, el número de elementos en un arreglo, el número de nodos en un grafo, o el número de bits en un número.

La variable dependiente de la función T es la cantidad de tiempo o espacio que consume el algoritmo al procesar una entrada de tamaño n. El tiempo se puede medir en unidades de tiempo (segundos, milisegundos, etc.) o en términos de operaciones elementales (comparaciones, asignaciones, etc.). El espacio se mide en unidades de memoria (bytes, kilobytes, megabytes, etc.).

1.3 Metodología experimental para evaluar la eficiencia

La metodología experimental para evaluar la eficiencia de un algoritmo se basa en los pasos del método científico:

1. **Observaciones (mediciones):** Se implementa el algoritmo y se ejecuta con diferentes entradas de diferentes tamaños. Se miden el tiempo y el espacio de ejecución para cada entrada.
2. **Hipótesis:** Se formula una hipótesis sobre la relación entre el tamaño de la entrada y el tiempo o espacio de ejecución. Por ejemplo, se puede hipotetizar que el tiempo de ejecución es lineal con respecto al tamaño de la entrada.
3. **Predicción:** Se utiliza la hipótesis para predecir el tiempo o espacio de ejecución para nuevas entradas.
4. **Verificación y validación:** Se ejecutan el algoritmo con nuevas entradas y se comparan los resultados medidos con las predicciones. Si los resultados coinciden, la hipótesis se considera verificada y validada. En caso contrario, se debe formular una nueva hipótesis y repetir el proceso.

Es importante tener en cuenta que la metodología experimental puede verse afectada por factores externos, como la carga del sistema, el lenguaje de programación utilizado, y la implementación del algoritmo. Por lo tanto, es importante realizar múltiples mediciones y utilizar técnicas estadísticas para analizar los resultados.

La metodología experimental se puede ayudar de herramientas de *profiling* para medir el tiempo de ejecución y la memoria utilizada por los algoritmos. Además, se puede utilizar herramientas como *benchmarking* para comparar diferentes implementaciones del mismo algoritmo. En la siguiente sección se referencias algunas herramientas para mediciones experimentales y profiling.

1.3.1 Evaluación experimental del tiempo de ejecución

- Python:
  - timeit
  - 5 Ways to Measure Execution Time in Python

- Java:
  - Clase Stopwatch
  - Measure Elapsed Time in Java
- C#:
  - How to Calculate the Code Execution Time in C#?

1.3.2 Evaluación experimental del consumo de memoria

- Python
  - Memory profiling in Python
  - Introduction to Memory Profiling in Python

1.4 Metodología analítica

La metodología analítica proporciona un marco para predecir el rendimiento de un algoritmo mediante el análisis de su estructura. Este enfoque se basa en el principio de que el tiempo de ejecución de un programa puede ser expresado como la suma del costo de cada operación elemental multiplicada por su frecuencia de ejecución.

Operaciones Elementales Las operaciones elementales son aquellas cuyo tiempo de ejecución está acotado superiormente por una constante. Ejemplos comunes incluyen asignaciones, comparaciones, operaciones aritméticas básicas, acceso a elementos de un arreglo y operaciones de salto.

Principio de Knuth El análisis se fundamenta en el principio de Knuth, que establece que el tiempo total de ejecución $T(n)$ puede ser calculado como la suma de los productos del tiempo t_i requerido por cada instrucción y su frecuencia de ejecución f_i:

$$T(n) = \sum_i t_i \cdot f_i$$

En la práctica, determinar con precisión los valores de t_i puede ser complejo y dependiente de la máquina. Por lo tanto, el análisis a menudo se centra en identificar las operaciones dominantes que tienen el mayor impacto en el tiempo de ejecución.

Notación Tilde La notación tilde, denotada como $T(n) \sim f(n)$, se utiliza para indicar que el límite de $T(n)/f(n)$ tiende a 1 cuando n tiende a infinito. En otras palabras, $f(n)$ proporciona una aproximación asintótica de $T(n)$ que se vuelve más precisa a medida que n crece. Formalmente,

$$T(n) \sim f(n) \quad \text{si y solo si} \quad \lim_{n\to\infty} \frac{T(n)}{f(n)} = 1$$

Orden de Crecimiento El orden de crecimiento simplifica aún más el análisis al eliminar las constantes multiplicativas. Si $f(n) \sim a \cdot g(n)$, donde a es una constante, entonces se dice que $f(n)$ tiene un orden de crecimiento de $g(n)$. Esto permite comparar la eficiencia de los algoritmos en términos de cómo escalan con el tamaño de la entrada.

Metodología Simplificada Una metodología simplificada para el análisis analítico consta de los siguientes pasos:

1. Identificar la operación de mayor frecuencia en el ciclo más interno del algoritmo.
2. Estimar la frecuencia de ejecución de esta operación en función del tamaño de la entrada n.
3. Determinar la función tilde que aproxima el tiempo de ejecución y el orden de crecimiento.

Ejemplo 1: Determinar T(n) para el siguiente programa

```
double max(double[] a) {
    double x = a[0];                    // t1, f1=1
    for(int i=1; i<a.length; i++)       // t2, f2=1; t3, f3=n; t4, f4=n-1;
        if (a[i]>x) x=a[i];             // t5, f5=n-1; t6, f6=$0...n-1$
    return x;                           // t7, f7=1
}
```

Listing 1: Encontrar el mayor elemento en un arreglo

$$
\begin{aligned}
T(n) &\leq t_1 + t_2 + t_3 n + t_4(n-1) + t_5(n-1) + t_6(n-1) + t_7 \\
T(n) &\leq n(t_3 + t_4 + t_5 + t_6) + (t_1 + t_2 - t_4 - t_5 - t_6 + t_7) \\
T(n) &\sim an \qquad \text{Funcion tilde} \\
T(n) &\sim n \qquad \text{Orden de crecimiento: Lineal}
\end{aligned}
$$

Ejemplo 2: Determinar T(n) para el siguiente programa

```
int paresCero(int[] a) {
    int conteo=0;
    for(int i=0; i<a.length; i++)
        for(int j=0; j<a.length; j++)
            if (i!=j && a[i]+a[j]==0)
                conteo++;
    return conteo;
}
```

Listing 2: Encontrar parejas que suman cero

Ejercicio Desarrollar las solución del ejemplo 2 por la metodología Knuth.

Ejemplo Solución del ejemplo 2 por medio de la operación elemental representativa.

$$
\begin{aligned}
T(n) &\leq \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \\
&\leq \sum_{i=0}^{n-1} n \\
&\leq n^2 \\
&\sim n^2 \qquad \text{Función title} \\
&n^2 \qquad \text{Orden de crecimiento: Cuadrático}
\end{aligned}
$$

1.4.1 Modelos de costo

Ejemplo 3: Contabilizar sumas, multiplicaciones y accesos al arreglo

```
public static double evalPoly(double[] a, double x)
{
    double s = 0;
    for(int i=0; i<a.length; i++)
        s += a[i]*Math.pow(x,i);
    return s;
}
```

Listing 3: Evaluar un polinomio - siguiendo definición

Ejemplo 4: Contabilizar sumas, multiplicaciones y accesos al arreglo

```
public static double horner(double[] a, double x) {
    double s = a[a.length-1];
    for(int i=a.length-2; i>=0; i--)
        s = s*x + a[i];
    return s;
}
```

Listing 4: Evaluar un polinomio - método de Horner

1.4.2 Notación asintótica

La notación asintótica es una herramienta matemática que nos permite describir el comportamiento de una función cuando su argumento tiende a infinito. En el análisis de algoritmos, utilizamos esta notación para describir el crecimiento del tiempo de ejecución o el uso de memoria.

Notación Big-O (O) La notación Big-O proporciona un límite superior asintótico para una función. Formalmente, para funciones $f(n)$ y $g(n)$:

$f(n) = O(g(n))$ si existen constantes positivas c y n_0 tales que: $0 \leq f(n) \leq cg(n)$ para toda $n \geq n_0$

Notación Omega (Ω) La notación Omega proporciona un límite inferior asintótico. Formalmente:

$f(n) = \Omega(g(n))$ si existen constantes positivas c y n_0 tales que: $0 \leq cg(n) \leq f(n)$ para toda $n \geq n_0$

Notación Theta (Θ) La notación Theta proporciona un límite ajustado asintóticamente. Una función es $\Theta(g(n))$ si es tanto $O(g(n))$ como $\Omega(g(n))$.

1.5 Análisis de casos

En el análisis de algoritmos, consideramos diferentes escenarios:

- **Mejor caso:** El escenario más favorable para el algoritmo.
- **Peor caso:** El escenario menos favorable, más pesimista.
- **Caso promedio:** El desempeño promedio de una invocación del algoritmo.
- **Análisis amortizado:** El desempeño promedio a lo largo de una serie de invocaciones.

Conjunto	Nombre
$O(1)$	Constante
$O(\log(n))$	Logaritmico
$O(n)$	Lineal
$O(n\log(n))$	Linearitmético
$O(n^2)$	Cuadrático
$\vdots$	
$O(n^k)$	Polinómico
$\vdots$	
$O(b^n), b > 1$	Exponencial

Table 1: Jerarquía de conjuntos Big-O

Ejemplo 5: Estimar el tiempo promedio

```
int buscar(T[] datos, T item) {
    for(int i=0; i<datos.length; i++)
        if (item.equals(datos[i]))
            return i;
    return -1;
}
```

Listing 5: Búsqueda secuencial en un arreglo

$$
\begin{aligned}
\overline{T(N)} &\leq \frac{1}{n}\sum_{i=1}^{n} i \quad \text{Número promedio de comparaciones} \\
&\leq \frac{1}{n}\left(\frac{n(n+1)}{2}\right) \\
&\leq \frac{n+1}{2} \\
&\sim \frac{n}{2} \qquad \text{Función title} \\
n \qquad & \text{Orden de crecimiento: Lineal}
\end{aligned}
$$

Ejemplo 6: Análizar el tiempo amortizado[1]

```
private void resize(int capacity) {
    Item[] copy = (Item[]) new Object[capacity];
    for (int i = 0; i < n; i++)
        copy[i] = a[i];
    a = copy;
}

public void add(Item item) {
    if (n == a.length) resize(2*a.length);
    a[n++] = item;
}
```

Listing 6: Cambio de tamaño de un arreglo

[1] `ResizingArrayBag.java` de Sedgewick & Wayne