Análisis de Algoritmos

Notas de clase

Estructuras de datos y algoritmos

Facultad TIC - UPB

Spring 2025

Jorge Mario Londoño Peláez & Varias AI

August 22, 2025

Descripción de la unidad

En esta unidad se analizan las metodologías para analizar la eficiencia de algoritmos y para describirla por medio de ordenes de crecimiento y de notación asintótica.

Contents

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

Para derivar el tiempo de ejecución, se analiza la frecuencia de cada operación. La inicialización (‘t1‘, ‘t2‘, ‘t7‘) ocurre una sola vez. El ciclo ‘for‘ se evalúa n veces (para ‘i‘ desde 1 hasta n, donde falla la condición), por lo que la frecuencia de ‘t3‘ es n. El cuerpo del ciclo se ejecuta $n-1$ veces (para ‘i‘ de 1 a $n-1$), estableciendo las frecuencias de ‘t4‘ y ‘t5‘. La asignación ‘t6‘ ocurre en el peor de los casos $n-1$ veces (si el arreglo está ordenado ascendentemente).

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

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

Estos dos ejemplos presentan una importante lección sobre eficiencia. El método 'evalPoly', que sigue la definición matemática directa, es ineficiente. La operación 'Math.pow(x,i)' no es de tiempo constante; su costo es proporcional a $\log(i)$. Esto hace que el costo total del ciclo sea aproximadamente $\sum_{i=0}^{n-1} \log(i)$, resultando en una complejidad de $O(n \log n)$ o incluso $O(n^2)$ si 'pow' se implementa con multiplicaciones repetidas. En contraste, el método de Horner es marcadamente superior. Realiza únicamente n multiplicaciones y n sumas, lo que le otorga una complejidad de $O(n)$. Este es un claro ejemplo de cómo una reestructuración inteligente del algoritmo puede conducir a una mejora drástica en el rendimiento.

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 $\Omega(g(n))$.

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

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.

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

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

El análisis anterior calcula el costo promedio para una búsqueda exitosa, bajo la suposición de que cada elemento tiene la misma probabilidad de ser buscado. Es igualmente importante considerar otros casos. El **mejor caso** ocurre cuando el elemento buscado está en la primera posición, requiriendo una sola comparación ($O(1)$). El **peor caso** ocurre cuando el elemento está en la última posición o no se encuentra en el arreglo, lo que requiere N comparaciones ($O(N)$).

Ejemplo 6: Analizar 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

Este es un ejemplo clásico de análisis amortizado. Una operación 'add' individual puede ser costosa. Si el arreglo está lleno, la llamada a 'resize' toma un tiempo proporcional al tamaño del arreglo, $O(N)$. Sin embargo, la mayoría de las llamadas a 'add' son muy rápidas, tomando tiempo constante, $O(1)$. El costo de redimensionar se distribuye (o se "amortiza") entre muchas operaciones baratas. Si comenzamos con un arreglo vacío y realizamos N adiciones, el costo total de todas las redimensionaciones es en realidad $O(N)$, no $O(N^2)$. Por lo tanto, el costo total de N adiciones es $O(N)$, y el costo promedio o **amortizado** por operación es $O(1)$.

1.6 Análisis de espacio

Determinar los requerimientos de memoria de un algoritmo es crucial para asegurar que el programa pueda ejecutarse eficientemente, especialmente cuando se manejan grandes volúmenes de datos. Un análisis de espacio permite prever la cantidad de memoria que un algoritmo necesitará, optimizando el uso de los recursos disponibles y evitando posibles errores por falta de memoria.

La metodología analítica para el análisis de espacio se basa en un enfoque bottom-up, donde se evalúa el espacio consumido por los componentes básicos y luego se combinan para obtener el espacio total requerido por el algoritmo.

Espacio consumido por datos de tipos primitivos Los tipos de datos primitivos (enteros, booleanos, caracteres, números de punto flotante, etc.) consumen una cantidad fija de memoria que depende del lenguaje de programación y la arquitectura del sistema. Por ejemplo, en Java, un entero ("int") típicamente ocupa 4 bytes, mientras que un número de punto flotante de doble precisión ("double") ocupa 8 bytes[2]. En la tabla 2 se resume el espacio requerido por los tipos de datos primitivos.[3]

Espacio consumido por referencias Las referencias (o punteros) son variables que almacenan la dirección de memoria de un objeto. El espacio consumido por una referencia también es fijo y depende de la arquitectura del sistema. En sistemas de 64 bits, una referencia típicamente ocupa 8 bytes.

Espacio consumido por objetos sencillos Un objeto sencillo es una instancia de una clase que contiene solo datos de tipos primitivos y/o referencias. El espacio consumido por un objeto

[1] `ResizingArrayBag.java` de Sedgewick & Wayne
[2] Los tipos punto flotante siguen el estándar IEEE 754
[3] En Python realmente son clases y siguen el modelo de objetos con overhead discutido más abajo

Tipo	Espacio (bits)		
	Java	C#	Python
byte	8	8	
short	16	16	
int	32	32	28 o *más*
long	64	64	
float	32	32	192
double	64	64	-
boolean	8	8	224
char	16	16	8-32

Table 2: Espacio para tipos de datos primitivos

sencillo se calcula sumando el espacio consumido por cada uno de sus campos, más un overhead adicional que depende del lenguaje de programación y la implementación de la máquina virtual. En el caso de Java este overhead es de 12 bytes. Adicional a esto, la máquina virtual siempre reserva una cantidad de memoria múltiplo de 8 bytes (64 bits), para asegurar la *alineación* de objetos en memoria. Para completar un múltiplo de 8, se agrega un campo "padding" que no se usa.

Ejemplo Una clase wrapper

```
public class Integer {
    int value;
}
```

Listing 7: Espacio consumido por un objeto Integer

$$\begin{array}{rcll} \textit{Espacio} & = & 12 & \text{Overhead} \\ & & +4 & \text{int} \\ \hline & & 16 & \text{Total} \end{array}$$

Ejemplo Python ofrece el metodo `sys.getsizeof` para obtener el espacio asociado a una variable[4]

```
import sys
a=1
b=3.14159
c=True
d="Hello World"           # 11 char
print(sys.getsizeof(a))  # Size of an 'int'  = 28
print(sys.getsizeof(b))  # Size of a 'float' = 24
print(sys.getsizeof(c))  # Size of a 'bool'  = 28
print(sys.getsizeof(d))  # Size of a 'str'   = 60
```

Listing 8: Espacio consumido por distintos tipos de variables en Python

[4]Es importante notar que 'sys.getsizeof()' puede ser engañoso. Para tipos de datos contenedores como strings, listas o diccionarios, este método devuelve el tamaño del objeto "contenedor" en sí, pero no incluye el tamaño de los elementos que contiene. El uso de memoria real puede ser significativamente mayor.

Espacio consumido por objetos con referencias y clases internas Si un objeto contiene referencias a otros objetos, el espacio total consumido incluye el espacio del objeto en sí, más el espacio de todos los objetos a los que hace referencia. Si la clase del objeto contiene clases internas, también se debe considerar el espacio consumido por las instancias de estas clases internas. En Java, una clase interna contiene adicionalmente una referencia hacia la instancia de la clase externa que la contiene.

Ejemplo Una clase wrapper

```
public class List<T> {
    class Node {
        T data;
        Node next;
    }
    Node head;
    int n;
}
```

Listing 9: Espacio consumido por un objeto con clase interna

$Espacio\ Node$	=	12	Overhead
		+8	ref data
		+8	ref next
		+8	ref externa
		+4	padding
		40	Total Node
$Espacio\ Lista$	=	12	Overhead
		+8	ref head
		+4	int
		24	Total Lista
$Lista(N)$	=	24	Instancia de List
		$+40 * N$	Instancias de Node
		$24 + 40 * N$	Total Lista(N)

Espacio consumido por estructuras dinámicas Las estructuras dinámicas, como las listas enlazadas, utilizan memoria de forma dinámica, es decir, la cantidad de memoria que utilizan puede variar durante la ejecución del programa. En el caso de una lista simplemente enlazada, cada nodo de la lista contiene un dato y una referencia al siguiente nodo. Por lo tanto, el espacio total consumido por la lista es proporcional al número de nodos en la lista.

Espacio consumido por arreglos Los arreglos son estructuras de datos que almacenan una colección de elementos del mismo tipo en posiciones de memoria contiguas. El espacio consumido por un arreglo depende del tipo de los elementos que almacena y del número de elementos en el arreglo. Adicionalmente se almacena el overhead del objeto y la longitud del arreglo.

- **Arreglos unidimensionales de tipo primitivo:** El espacio consumido por un arreglo unidimensional de tipo primitivo se calcula multiplicando el tamaño del tipo primitivo por el número de elementos en el arreglo, más el espacio del overhead y la longitud del arreglo.

Ejemplo Un arreglo 1-D de enteros

```
1         int[] a = new int[N];
2
```

Listing 10: Espacio consumido por un arreglo primitivo

$E(N)$	$=$	12	Overhead
		$+4$	int length
		$+4 * N$	N int
		$+0$ ó $+4$	padding
$E(N)$	$\leq$	$16 + 4 * N + 4$	Total peor caso

- **Arreglos bidimensionales de tipo primitivo:** Un arreglo bidimensional se debe descomponer como un arreglo de arreglos. El arreglo principal (*arreglo filas*) es un arreglo de referencias a arreglos del tipo primitivos (*arreglos columnas*).
- **Arreglos de tipo referencia:** En el caso de arreglos de tipo referencia, cada elemento del arreglo es una referencia a un objeto. Por lo tanto, el espacio consumido por el arreglo incluye el espacio de las referencias, más el espacio de todos los objetos a los que hacen referencia.

En resumen, el análisis de espacio es una herramienta fundamental para comprender y optimizar el uso de la memoria en los algoritmos. Al aplicar una metodología analítica bottom-up, es posible prever los requerimientos de memoria y evitar posibles problemas de rendimiento y errores por falta de memoria.

Bibliografía

[1] Aditya Bhargava. *Grokking Algorithms*. Manning Publications, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Narasimha Karumanchi. *Data Structures and Algorithms Made Easy*. CareerMonk Publications, 2011.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2005.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.