Apuntes de Análisis y Diseño de Algoritmos (ADA)

Jaime Hernández Delgado
@nickhernd

Curso 2023/24

Contents

1 Introducción al Análisis y Diseño de Algoritmos

1.1 Definición de Algoritmo

Un algoritmo es una serie finita de instrucciones no ambiguas que expresa un método de resolución de un problema.

Características clave:

- Máquina bien definida
- Recursos acotados por paso
- Terminación en tiempo finito

1.2 Complejidad Algorítmica

Medida de recursos necesarios para resolver un problema.

Factores principales:

- Tamaño del problema
- Parámetro representativo (ej. dimensión de matriz)

1.3 Importancia del ADA

- Comparación de soluciones
- Predicción de comportamiento con grandes datos
- Optimización de recursos
- Desarrollo de software eficiente

Nota: La eficiencia de un algoritmo puede marcar la diferencia entre resolver un problema en segundos o en años.

1.4 Ejemplos de Problemas Clásicos

1. Ordenación de datos
2. Búsqueda en estructuras
3. Problemas de optimización (ej. mochila, viajante)
4. Algoritmos en grafos

2 Eficiencia de Algoritmos

2.1 Análisis de Eficiencia

Existen dos enfoques principales:

2.1.1 1. Análisis Empírico (a posteriori)

- Se ejecuta el algoritmo con diferentes entradas
- Se mide el tiempo de ejecución real
- Ventajas: Medidas reales en el entorno de aplicación
- Desventajas: Depende de factores externos (hardware, compilador)

2.1.2 2. Análisis Teórico (a priori)

- Se calcula el número de operaciones elementales
- Independiente del hardware y la implementación
- Permite comparar algoritmos de forma abstracta

2.2 Notación Asintótica

Usada para describir el comportamiento de funciones:

- **O (Big O)**: Cota superior
- Ω **(Omega)**: Cota inferior
- Θ **(Theta)**: Cota ajustada

Ejemplo: $f(n) = 3n^2 + 2n + 1$ está en $O(n^2)$

2.3 Clases de Complejidad Comunes

Ordenadas de menor a mayor complejidad:

1. $O(1)$ - Constante
2. $O(\log n)$ - Logarítmica
3. $O(n)$ - Lineal
4. $O(n \log n)$ - Lineal logarítmica
5. $O(n^2)$ - Cuadrática
6. $O(2^n)$ - Exponencial

2.4 Análisis de Casos

- **Mejor caso**: Entrada que resulta en el menor tiempo de ejecución
- **Peor caso**: Entrada que resulta en el mayor tiempo de ejecución
- **Caso promedio**: Comportamiento esperado para una entrada aleatoria

2.5 Técnicas de Análisis

1. **Conteo de operaciones**: Sumar el costo de cada operación
2. **Regla de la suma**: $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
3. **Regla del producto**: $T_1(n) * T_2(n) = O(f(n) * g(n))$

2.6 Ejemplo: Análisis de Búsqueda Lineal

```
function busquedaLineal(arr, x):
    for i = 0 to n-1:
        if arr[i] == x:
            return i
    return -1
```

Análisis:

- Mejor caso: $O(1)$ (elemento al inicio)
- Peor caso: $O(n)$ (elemento al final o no presente)
- Caso promedio: $O(n)$

Nota: La eficiencia de un algoritmo puede variar drásticamente según el tamaño de la entrada. Es crucial considerar cómo escala el algoritmo con grandes conjuntos de datos.

3 Divide y Vencerás

3.1 Definición

Divide y Vencerás es una técnica de diseño de algoritmos que consiste en:

1. Dividir el problema en subproblemas más pequeños
2. Resolver cada subproblema de forma recursiva
3. Combinar las soluciones para obtener la solución del problema original

3.2 Esquema General

```
function DivideYVenceras(problema):
    if esPequeño(problema):
        return resolverDirectamente(problema)
    else:
        subproblemas = dividir(problema)
        for cada subproblema:
            solucionParcial = DivideYVencerás(subproblema)
        return combinar(solucionesParciales)
```

3.3 Análisis de Complejidad

Sea $T(n)$ el tiempo de ejecución para un problema de tamaño n:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c \\ aT(n/b) + f(n) & \text{si } n > c \end{cases}$$

Donde:

- a: número de subproblemas
- b: factor de reducción del tamaño
- $f(n)$: costo de dividir y combinar

3.4 Teorema Maestro

Para la recurrencia $T(n) = aT(n/b) + f(n)$, donde $a \geq 1$ y $b > 1$:

- Si $f(n) = O(n^{\log_b a - \epsilon})$, entonces $T(n) = \Theta(n^{\log_b a})$
- Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \log n)$
- Si $f(n) = \Omega(n^{\log_b a + \epsilon})$, entonces $T(n) = \Theta(f(n))$

3.5 Ejemplos de Algoritmos

3.5.1 1. Mergesort

- Divide: Partir el array en dos mitades
- Vencer: Ordenar recursivamente cada mitad
- Combinar: Mezclar las dos mitades ordenadas
- Complejidad: $O(n \log n)$

3.5.2 2. Quicksort

- Divide: Particionar el array alrededor de un pivote
- Vencer: Ordenar recursivamente las particiones
- Combinar: No necesario (in-place)
- Complejidad: Promedio $O(n \log n)$, Peor caso $O(n^2)$

3.5.3 3. Búsqueda Binaria

- Divide: Comparar con el elemento medio
- Vencer: Buscar en la mitad correspondiente
- Combinar: Retornar el resultado
- Complejidad: $O(\log n)$

3.6 Ventajas y Desventajas

Ventajas:

- Soluciones eficientes para problemas grandes
- Paralelización natural

Desventajas:

- Overhead de llamadas recursivas
- No siempre aplicable

3.7 Ejercicio Práctico

Implementa el algoritmo de la potencia usando Divide y Vencerás:

```
function potencia(base, exp):
    if exp == 0:
        return 1
    if exp % 2 == 0:
        temp = potencia(base, exp/2)
        return temp * temp
    else:
        return base * potencia(base, exp-1)
```

Analiza su complejidad y compárala con el método ingenuo.

Nota: Divide y Vencerás es una técnica poderosa que puede reducir significativamente la complejidad de muchos problemas. Sin embargo, es crucial identificar correctamente cómo dividir el problema y cómo combinar las soluciones.

4 Programación Dinámica

4.1 Definición

La Programación Dinámica (PD) es una técnica para resolver problemas de optimización mediante la combinación de soluciones de subproblemas superpuestos.

4.2 Características Clave

- Subestructura óptima
- Subproblemas superpuestos
- Memorización o tabulación de resultados parciales

4.3 Enfoque General

1. Identificar la estructura de la solución óptima
2. Definir recursivamente el valor de la solución óptima
3. Calcular el valor de la solución óptima de abajo hacia arriba
4. Construir la solución óptima a partir de la información calculada

4.4 Métodos de Implementación

4.4.1 1. Top-Down (Memoización)

- Enfoque recursivo con almacenamiento de resultados
- Evita recálculos de subproblemas ya resueltos

4.4.2 2. Bottom-Up (Tabulación)

- Enfoque iterativo
- Construye la solución de menor a mayor complejidad
- Generalmente más eficiente en espacio

4.5 Ejemplos Clásicos

4.5.1 1. Fibonacci

Recurrencia: $F(n) = F(n-1) + F(n-2)$

```
function fib(n):
    if n <= 1: return n
    memo = [0] * (n+1)
    memo[1] = 1
    for i in range(2, n+1):
        memo[i] = memo[i-1] + memo[i-2]
    return memo[n]
```

Complejidad: $O(n)$ tiempo, $O(n)$ espacio

4.5.2 2. Problema de la Mochila (0/1 Knapsack)

```
function knapsack(valores, pesos, capacidad):
    n = len(valores)
    K = [[0 for x in range(capacidad + 1)] for x in range(n + 1)]

    for i in range(n + 1):
        for w in range(capacidad + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif pesos[i-1] <= w:
                K[i][w] = max(valores[i-1] + K[i-1][w-pesos[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][capacidad]
```

Complejidad: $O(n \cdot W)$ tiempo y espacio, donde n es el número de items y W es la capacidad de la mochila.

4.6 Ventajas y Desventajas

Ventajas:

- Resuelve problemas de optimización complejos
- Evita recálculos innecesarios
- Puede mejorar significativamente la eficiencia temporal

Desventajas:

- Puede requerir mucho espacio de almacenamiento
- No siempre es fácil identificar la subestructura óptima

4.7 Cuándo Usar Programación Dinámica

- Problemas de optimización con subestructura óptima
- Cuando hay muchos subproblemas superpuestos
- Cuando el espacio adicional no es una limitación crítica

Nota: La Programación Dinámica es una técnica poderosa que puede transformar soluciones exponenciales en polinomiales para ciertos tipos de problemas. Es crucial practicar la identificación de subestructuras óptimas y la formulación de relaciones de recurrencia.

5 Algoritmos Voraces (Greedy)

5.1 Definición

Un algoritmo voraz es aquel que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción localmente óptima en cada paso con la esperanza de llegar a una solución globalmente óptima.

5.2 Características

- Toma decisiones locales óptimas en cada paso
- No reconsidera decisiones previas
- Generalmente eficiente en tiempo y espacio
- No siempre garantiza la solución óptima global

5.3 Esquema General

```
function Voraz(Problema):
    Solución =
    mientras Problema no esté resuelto:
        x = seleccionar(Problema)
        si es_factible(Solución  {x}):
            Solución = Solución  {x}
    return Solución
```

5.4 Ejemplos Clásicos

5.4.1 1. Problema del Cambio de Monedas

Dado un sistema monetario con monedas de valores $v_1, v_2, ..., v_n$, encontrar el mínimo número de monedas para dar un cambio C.

```
function cambioMonedas(C, monedas):
    resultado = []
    for moneda in ordenarDescendente(monedas):
        while C >= moneda:
            resultado.append(moneda)
            C -= moneda
    return resultado
```

Nota: Este algoritmo no siempre da la solución óptima para todos los sistemas monetarios.

5.4.2 2. Algoritmo de Kruskal (Árbol de Expansión Mínima)

```
function Kruskal(Grafo):
    A =
    para cada v  Grafo.V:
        MAKE-SET(v)
    ordenar las aristas de Grafo.E por peso ascendente
    para cada (u, v) en Grafo.E (en orden ascendente de peso):
        si FIND-SET(u)  FIND-SET(v):
            A = A  {(u, v)}
            UNION(u, v)
    return A
```

5.5 Ventajas y Desventajas

Ventajas:

- Generalmente sencillos de implementar
- Eficientes en tiempo y espacio
- Pueden proporcionar buenas aproximaciones

Desventajas:

- No siempre encuentran la solución óptima
- Pueden fallar en encontrar una solución válida
- La optimalidad local no garantiza optimalidad global

5.6 Cuándo Usar Algoritmos Voraces

- Problemas con subestructura óptima
- Cuando la elección voraz local lleva a una solución global óptima
- En problemas de optimización donde una aproximación es aceptable

5.7 Demostración de Corrección

Para demostrar que un algoritmo voraz es correcto, generalmente se utilizan dos propiedades:

1. **Propiedad de la elección voraz:** Una elección local óptima es consistente con una solución global óptima.
2. **Propiedad de subestructura óptima:** Una solución óptima al problema contiene dentro de ella soluciones óptimas a subproblemas.

6 Vuelta Atrás (Backtracking)

6.1 Definición

Vuelta Atrás es una técnica algorítmica para resolver problemas recursivamente intentando construir una solución de forma incremental, descartando las soluciones parciales ("backtracking") que no cumplen las restricciones del problema.

6.2 Características

- Explora sistemáticamente el espacio de soluciones
- Usa un árbol de recursión para representar el espacio de búsqueda
- Aplica una función de poda para descartar ramas no prometedoras
- Generalmente usado en problemas de satisfacción de restricciones y optimización

6.3 Esquema General

```
function backtrack(candidato):
    if es_solucion(candidato):
        guardar_solucion(candidato)
    else:
        for extension in generar_extensiones(candidato):
            if es_prometedor(extension):
                backtrack(extension)
```

6.4 Ejemplos Clásicos

6.4.1 1. Problema de las N-Reinas

Colocar N reinas en un tablero de ajedrez N×N sin que se amenacen mutuamente.

```
function n_reinas(tablero, columna):
    if columna >= N:
        return true
    for fila in range(N):
        if es_seguro(tablero, fila, columna):
            tablero[fila][columna] = 1
            if n_reinas(tablero, columna + 1):
                return true
            tablero[fila][columna] = 0
    return false
```

6.4.2 2. Sudoku

Rellenar una cuadrícula 9×9 con dígitos del 1 al 9, cumpliendo ciertas restricciones.

6.5 Ventajas y Desventajas

Ventajas:

- Encuentra todas las soluciones posibles
- Eficaz para problemas de satisfacción de restricciones
- Puede ser más eficiente que la fuerza bruta con buenas funciones de poda

Desventajas:

- Puede tener un tiempo de ejecución exponencial en el peor caso
- Consumo de memoria debido a la recursión
- No siempre es la mejor opción para problemas de optimización

6.6 Optimizaciones

- Ordenar las opciones de expansión por probabilidad de éxito
- Utilizar heurísticas específicas del problema para la función de poda
- Combinar con técnicas de memorización para subproblemas recurrentes

Nota: La eficacia de Vuelta Atrás depende en gran medida de una buena función de poda. Una poda efectiva puede reducir drásticamente el espacio de búsqueda.

7 Ramificación y Poda (Branch and Bound)

7.1 Definición

Ramificación y Poda es una técnica algorítmica para resolver problemas de optimización. Consiste en una enumeración sistemática de soluciones candidatas mediante búsqueda en árbol, donde ramas enteras del árbol se descartan utilizando límites superiores e inferiores de la función objetivo.

7.2 Características

- Combina una estructura de árbol de búsqueda con límites en la función objetivo
- Utiliza una función de cota para descartar soluciones subóptimas
- Mantiene la mejor solución encontrada hasta el momento
- Más eficiente que la búsqueda exhaustiva para muchos problemas de optimización

7.3 Componentes Principales

1. **Función de Ramificación:** Divide el problema en subproblemas
2. **Función de Cota:** Estima el mejor valor posible de una solución parcial
3. **Estrategia de Recorrido:** Determina el orden de exploración de los nodos (e.g., profundidad, anchura, mejor primero)

7.4 Esquema General

```
function branch_and_bound(problema):
    cola = crear_cola_prioridad()
    mejor_solucion = None
    cola.insertar(nodo_raiz(problema))
    while not cola.esta_vacia():
        nodo = cola.extraer_minimo()
        if es_solucion(nodo) and es_mejor(nodo, mejor_solucion):
            mejor_solucion = nodo
        else if es_prometedor(nodo, mejor_solucion):
            for hijo in ramificar(nodo):
                cola.insertar(hijo)
    return mejor_solucion
```

7.5 Ejemplo: Problema del Viajante de Comercio

- **Ramificación:** Expandir el tour parcial actual
- **Cota inferior:** Longitud del tour parcial + estimación del resto (e.g., MST)
- **Cota superior:** Longitud del mejor tour completo encontrado hasta ahora

7.6 Ventajas y Desventajas

Ventajas:

- Garantiza encontrar la solución óptima
- Más eficiente que la búsqueda exhaustiva para muchos problemas
- Flexible en cuanto a la estrategia de búsqueda

Desventajas:

- Puede requerir mucha memoria para problemas grandes
- La eficiencia depende en gran medida de la calidad de las cotas
- Puede ser lento para problemas muy grandes o con malas cotas

7.7 Comparación con Otras Técnicas

- **vs. Vuelta Atrás:** Más eficiente para problemas de optimización
- **vs. Programación Dinámica:** Mejor cuando el espacio de estados es muy grande
- **vs. Algoritmos Voraces:** Garantiza optimalidad, pero generalmente más lento

Nota: La efectividad de Ramificación y Poda depende crucialmente de la calidad de las funciones de cota. Invertir tiempo en desarrollar buenas cotas puede mejorar significativamente el rendimiento del algoritmo.