Escrito por **Adrián Quiroga Linares**.

Los algoritmos son herramientas esenciales para resolver problemas de manera eficiente, minimizando tanto el tiempo como los recursos necesarios. Las **estrategias algorítmicas** establecen enfoques específicos que guían el diseño de algoritmos para optimizar la resolución de problemas. En este tema, exploraremos cuatro estrategias principales: **Divide y vencerás**, **Algoritmos voraces**, **Backtracking**, y **Ramificación y poda**.

Divide y Vencerás

La estrategia divide y vencerás consiste en:

1. **Dividir el problema** en subproblemas más pequeños.
2. **Resolver los subproblemas** de manera independiente.
3. **Combinar las soluciones** de los subproblemas para resolver el problema original.

Características

- Los subproblemas deben ser independientes y más manejables que el problema original.
- La solución de los subproblemas se combina de manera eficiente.
- Si los subproblemas no pueden dividirse o no son independientes, esta estrategia no es aplicable.

Ejemplos clásicos

- **Multiplicación de enteros largos**: Divide los números en partes más pequeñas y realiza multiplicaciones parciales.
- **Multiplicación rápida de matrices**: Optimiza la cantidad de multiplicaciones requeridas, como el método de Strassen.
- **Ordenación por mezcla (Merge Sort)**: Divide el array en mitades, ordénalas por separado y combina los resultados.
- **Ordenación rápida (Quick Sort)**: Usa un pivote para dividir el array en partes menores y mayores, ordenándolas recursivamente.

Ventajas

- Divide y vencerás suele ser eficiente para problemas recursivos.
- Genera algoritmos con estructuras claras y jerárquicas.

Desafío clave Diseñar una división efectiva y eficiente del problema. Si no es posible, la estrategia pierde utilidad.

Algoritmos Voraces

Los algoritmos voraces construyen una solución paso a paso, seleccionando en cada iteración la opción que parece mejor en ese momento (solución localmente óptima).

Componentes principales

1. **C**: Conjunto de candidatos disponibles para la solución.
2. **S**: Conjunto de candidatos ya seleccionados para formar la solución.
3. **R**: Conjunto de candidatos descartados.
4. **Funciones esenciales**:
    - **solución(S)**: Comprueba si SS es una solución válida.
    - **seleccionar(C)**: Escoge el mejor candidato en el conjunto CC.
    - **factible(S, x)**: Verifica si agregar xx a SS mantiene una solución válida.
    - **Insertar(S, x)**: Añade xx al conjunto SS.
    - **Objetivo(S)**: Calcula el costo o valor de la solución parcial SS.

Características

- **Eficiencia**: Tienen complejidad polinomial, lo que los hace rápidos.
- **Subóptimos**: No garantizan encontrar la solución óptima en todos los casos.
- **Aplicaciones típicas**:
    - Problemas de optimización como el *cambio de monedas* o el *problema del viajante* en casos simples.

Ejemplo En el problema del cambio de monedas, el algoritmo voraz selecciona las monedas de mayor denominación posible en cada paso. Aunque puede no ser óptimo, como en el caso de denominaciones no estándar, es rápido y fácil de implementar.

Backtracking (Vuelta atrás)

El backtracking explora el espacio de soluciones de manera sistemática. A diferencia de los algoritmos voraces, si una solución parcial no lleva al resultado óptimo, retrocede y elimina esa opción (de ahí el término “vuelta atrás”).

Componentes principales

1. **Espacio de soluciones**: Representado implícitamente como un árbol o grafo.
2. **Funciones esenciales**:
    - **s**: Solución parcial.
    - **sINICIAL**: Valor inicial de la solución parcial (usualmente vacío o no válido).
    - **nivel**: Nivel actual en el árbol de soluciones.
    - **fin**: Indica si se ha encontrado una solución.
    - **Generar(nivel, s)**: Genera una solución parcial para un nivel dado.
    - **Solución(nivel, s)**: Verifica si la solución parcial actual es válida.

- **Criterio(nivel, s)**: Comprueba si la solución parcial puede derivar en una solución válida. Si no, se poda.
- **MasHermanos(nivel, s)**: Determina si hay más candidatos en el nivel actual.
- **Retroceder(nivel, s)**: Elimina un candidato y vuelve al nivel anterior.

Características

- **Eficiencia**: Ineficiente, con complejidad factorial o exponencial.
- **Garantía**: Encuentra la solución óptima si existe.
- **Aplicaciones típicas**:
  - Problemas de optimización y combinatoria, como el *problema de las n-reinas* o el *problema del viajante*.

Mejoras posibles La poda de ramas irrelevantes puede reducir significativamente el espacio de búsqueda.

Ramificación y Poda (Branch & Bound)

Es una extensión del backtracking que utiliza estrategias específicas de ramificación y poda para mejorar la eficiencia.

Diferencias clave respecto al Backtracking

1. **Ramificación**:
   - No se recorre en profundidad exclusivamente.
   - Permite estrategias más flexibles (FIFO, LIFO, etc.).
2. **Poda**:
   - Usa **cotas** para estimar los beneficios posibles desde un nodo, reduciendo el número de nodos explorados.

Cotas

- **CS (Cota Superior)**: Valor máximo que se puede alcanzar desde un nodo.
- **CI (Cota Inferior)**: Valor mínimo que se puede alcanzar desde un nodo.
- **BE (Beneficio Estimado)**: Promedio entre CS y CI, usado para decidir qué nodo explorar primero.

Estrategias de ramificación

- **LC-FIFO**: Selecciona el nodo de menor BEBE, desempate según orden de llegada.
- **MB-FIFO**: Selecciona el nodo de mayor BEBE, desempate según orden de llegada.
- **LC-LIFO**: Igual que LC-FIFO, pero desempate según último en llegar.
- **MB-LIFO**: Igual que MB-FIFO, pero desempate según último en llegar.

Funciones principales

1. **Seleccionar(LNV)**: Selecciona un nodo de la lista de nodos vivos (LNV) siguiendo la estrategia.
2. **Solución(y)**: Comprueba si el nodo yy representa una solución completa.
3. **Valor(y)**: Retorna el valor de la solución parcial en el nodo yy.

Ventajas y limitaciones

- **Mejoras**: Reduce el espacio de búsqueda respecto al backtracking tradicional.
- **Costo adicional**: Calcular cotas requiere tiempo y puede ser costoso en el peor caso.

Casos de uso Muy efectivo para problemas de optimización combinatoria, como el *problema de asignación* o el *problema del viajante*.

Conclusión

Cada estrategia tiene ventajas y limitaciones. La elección depende del problema a resolver:

- **Divide y vencerás**: Problemas recursivos y divisibles.
- **Voraces**: Optimización rápida y aproximada.
- **Backtracking**: Solución exhaustiva y garantizada.
- **Ramificación y poda**: Optimización más inteligente en problemas complejos.