# Exposición: Problemas de Análisis y Diseño de Algoritmos

## Brute Force vs Greedy

Delgado Romero, Gustavo
Torres Reategui, Joaquin
Quintanilla Quispe, Dylan

Universidad Nacional de Ingeniería
CC371 - Análisis y Diseño de Algoritmos

25 de septiembre de 2025

# Tabla de Contenidos

# Resumen de Problemas

## Problemas a Desarrollar

1. **Activity Selection Problem** - Selección de actividades
2. **Graph Coloring** - Coloreo de grafos
3. **Job Scheduling with Deadlines** - Programación de trabajos
4. **Huffman Codes** - Códigos de Huffman

## Enfoques Comparados

- **Fuerza Bruta**: Solución óptima garantizada, complejidad exponencial
- **Greedy**: Solución eficiente, no siempre óptima, complejidad polinomial

# Tabla de Contenidos

# Activity Selection Problem - Enunciado

## Descripción del Problema

Seleccionar el máximo número de actividades mutuamente compatibles de un conjunto dado.
Cada actividad tiene:

- Tiempo de inicio y finalización
- Compatibilidad: No deben solaparse en el tiempo
- Objetivo: Maximizar el número de actividades seleccionadas

## Ejemplo

Actividades: [(1,2), (3,4), (0,6), (5,7), (8,9), (5,9)]
Solución óptima: 4 actividades

# Activity Selection - Fuerza Bruta

## Estrategia

- Generar todos los subconjuntos posibles de actividades
- Verificar compatibilidad para cada subconjunto
- Seleccionar el subconjunto válido de mayor tamaño

## Complejidad

$O(2^n \cdot n^2)$ - Exponencial

# Activity Selection - Pseudocódigo Fuerza Bruta

**Entrada:** Lista de actividades $(start, end)$
**Salida:** Subconjunto de máximo tamaño de actividades compatibles

$n \leftarrow$ número de actividades
$best \leftarrow$ lista vacía
**for** $r = 1$ to $n$ **do**
  **for** cada $subset$ en $combinations(activities, r)$ **do**
    **if** $is\_valid(subset) \wedge |subset| > |best|$ **then**
      $best \leftarrow subset$
    **end if**
  **end for**
**end forreturn** $best$

# Activity Selection - Ejemplo Fuerza Bruta

## Actividades: [(1,3), (2,5), (4,6)]

- **Subconjuntos de tamaño 1**: Todos válidos
- **Subconjuntos de tamaño 2**:
  - {A,B}: Inválido (se solapan)
  - {A,C}: Válido
  - {B,C}: Inválido (se solapan)
- **Subconjunto de tamaño 3**: Inválido

## Solución óptima

{A,C} con 2 actividades

# Activity Selection - Enfoque Greedy

## Estrategia Óptima

- Ordenar actividades por tiempo de finalización
- Seleccionar siempre la siguiente actividad que termina primero y no se solape

## Complejidad

$O(n \log n)$ - Polinomial (óptimo)

# Activity Selection - Pseudocódigo Greedy

**Entrada:** Arreglos *start* y *finish* de $n$ actividades
**Salida:** Número máximo de actividades compatibles

```
1: Ordenar actividades por finish
2: count ← 1, j ← 0
3: for i = 1 to n − 1 do
4:     if start[i] > finish[j] then
5:         count ← count + 1
6:         j ← i
7:     end if
8: end forreturn count
```

## Actividades ordenadas: [(1,2), (3,4), (0,6), (5,7), (8,9), (5,9)]

- Seleccionar (1,2)
- Seleccionar (3,4) - No se solapa
- Omitir (0,6) - Se solapa
- Seleccionar (5,7) - No se solapa
- Seleccionar (8,9) - No se solapa
- Omitir (5,9) - Se solapa

## Resultado

4 actividades seleccionadas: [(1,2), (3,4), (5,7), (8,9)]

# Activity Selection - Análisis Comparativo

| **Enfoque** | **Complejidad** | **Optimalidad** |
|---|---|---|
| Fuerza Bruta | $O(2^n \cdot n^2)$ | ✓ |
| Greedy | $O(n \log n)$ | ✓ |

## Conclusión

- Greedy es óptimo y eficiente para este problema
- Fuerza bruta solo práctico para instancias pequeñas ($n < 20$)
- El ordenamiento por tiempo de finalización garantiza optimalidad

# Tabla de Contenidos

# Graph Coloring - Enunciado

## Descripción del Problema

Asignar colores a vértices de un grafo tal que:

- Vértices adyacentes tengan colores diferentes
- Minimizar el número de colores usado (número cromático)
- Problema NP-completo en general

## Aplicaciones

- Scheduling de procesos
- Asignación de registros
- Sudoku

# Graph Coloring - Fuerza Bruta

## Estrategia

- Probar todas las asignaciones de $k$ colores a $n$ vértices
- Verificar validez para cada asignación
- Encontrar el mínimo $k$ que permita coloreo válido

## Complejidad

$O(k^n \cdot n \cdot m)$ - Exponencial

# Graph Coloring - Pseudocódigo Fuerza Bruta

**Entrada:** Grafo $G$, número máximo de colores $k$
**Salida:** Asignación válida de colores o null

```
for cada assignment en product(range(k), repeat = n) do
    if is_valid(G, assignment) then return assignment
    end if
end forreturn null
```

```
function IS_VALID(G, colors)
    for u = 0 to |G| - 1 do
        for cada v en G[u] do
            if colors[u] = colors[v] then return false
            end if
        end for
    end forreturn true
end function
```

# Graph Coloring - Ejemplo Fuerza Bruta

## Grafo con 3 vértices: Triángulo

- **k=2**: Probar $2^3 = 8$ asignaciones, ninguna válida
- **k=3**: Probar $3^3 = 27$ asignaciones
- Encontrar solución: [0,1,2] - 3 colores

## Limitaciones

- 10 vértices con 3 colores: $3^{10} = 59,049$ pruebas
- 20 vértices: $3^{20} > 3.4$ billones de pruebas

# Graph Coloring - Enfoque Greedy

## Estrategia

- Procesar vértices en orden fijo
- Asignar el menor color disponible no usado por vecinos
- No garantiza número cromático mínimo

## Complejidad

$O(V^2 + E)$ - Polinomial

# Graph Coloring - Pseudocódigo Greedy

**Entrada:** Grafo $G$ con $V$ vértices

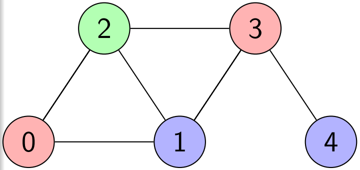
**Salida:** Asignación válida de colores

$result \leftarrow$ arreglo de tamaño $V$ inicializado en -1
$result[0] \leftarrow 0$
**for** $u = 1$ to $V - 1$ **do**
    Marcar colores de vecinos como no disponibles
    Encontrar primer color disponible $cr$
    $result[u] \leftarrow cr$
    Resetear marcas de disponibilidad
**end forreturn** *result*

# Graph Coloring - Ejemplo Greedy

**Grafo de 5 vértices**

- V0: Color 0
- V1: Color 1 (vecino de V0)
- V2: Color 2 (vecino de V0,1)
- V3: Color 0 (vecino de V1,2)
- V4: Color 1 (vecino de V3)




**Resultado**

3 colores usados (óptimo para este grafo)

# Graph Coloring - Análisis Comparativo

| **Enfoque** | **Complejidad** | **Optimalidad** |
|---|---|---|
| Fuerza Bruta | $O(k^n \cdot n \cdot m)$ | ✓ |
| Greedy | $O(V^2 + E)$ | × |

## Conclusión

- Fuerza bruta garantiza optimalidad pero es impráctica
- Greedy es eficiente pero puede usar más colores de los necesarios
- En la práctica: Greedy para grafos grandes, heurísticas avanzadas para mejor aproximación

# Tabla de Contenidos

# Job Scheduling - Enunciado

## Descripción del Problema

Programar trabajos con:

- Deadline: Tiempo límite para completar
- Profit: Ganancia por completar a tiempo
- Cada trabajo toma 1 unidad de tiempo
- Objetivo: Maximizar ganancia total

## Aplicaciones

- Scheduling de procesos en CPU
- Planificación de producción
- Asignación de recursos limitados

# Job Scheduling - Fuerza Bruta

## Estrategia

- Generar todas las permutaciones de trabajos
- Para cada permutación, programar trabajos en orden
- Solo incluir trabajos que cumplan deadline
- Seleccionar permutación con máxima ganancia

## Complejidad

$O(n! \cdot n)$ - Factorial

# Job Scheduling - Pseudocódigo Fuerza Bruta

**Entrada:** Lista de trabajos $(id, deadline, profit)$
**Salida:** Programación con máxima ganancia

```
max_profit ← 0, best_schedule ← []
for cada perm en permutations(jobs) do
    time ← 0, profit ← 0, schedule ← []
    for cada job en perm do
        if time < job.deadline then
            schedule.append(job), profit ← profit + job.profit
            time ← time + 1
        end if
    end for
    if profit > max_profit then
        max_profit ← profit, best_schedule ← schedule
    end if
end forreturn best_schedule, max_profit
```

# Job Scheduling - Ejemplo Fuerza Bruta

### Trabajos: [(a,2,100), (b,1,19), (c,2,27), (d,1,25), (e,3,15)]

- **Permutación 1**: [a,b,c,d,e] $\rightarrow$ Ganancia: 142
- **Permutación 2**: [b,a,c,d,e] $\rightarrow$ Ganancia: 134
- **Permutación 3**: [d,a,c,b,e] $\rightarrow$ Ganancia: 140
- **Mejor**: [a,c,e] con ganancia 142

### Limitaciones

5 trabajos = 120 permutaciones, 10 trabajos = 3.6 millones

## Job Scheduling - Enfoque Greedy

### Estrategia

- Ordenar trabajos por profit descendente
- Para cada trabajo, asignar al slot más tardío disponible dentro de su deadline

### Complejidad

$O(n \log n + n \cdot d)$ - Polinomial

# Job Scheduling - Pseudocódigo Greedy

**Entrada:** Lista de trabajos $(id, deadline, profit)$
**Salida:** Programación y ganancia total

Ordenar trabajos por profit descendente
$max\_d \leftarrow \text{máx}(deadlines)$
$slots \leftarrow$ arreglo de tamaño $max\_d + 1$ inicializado en -1
$total\_profit \leftarrow 0$
**for** cada *job* en trabajos ordenados **do**
    **for** $t = job.deadline$ to 1 **do**
        **if** $slots[t] = -1$ **then**
            $slots[t] \leftarrow job.id$, $total\_profit \leftarrow total\_profit + job.profit$
            **break**
        **end if**
    **end for**
**end forreturn** trabajos en *slots*, $total\_profit$

# Job Scheduling - Ejemplo Greedy

### Trabajos ordenados: [a(100), c(27), d(25), b(19), e(15)]

- **a**(deadline=2): Slot 2 → Profit=100
- **c**(deadline=2): Slot 1 → Profit=127
- **d**(deadline=1): No hay slot disponible
- **b**(deadline=1): No hay slot disponible
- **e**(deadline=3): Slot 3 → Profit=142

### Resultado

Programación: [c, a, e] con ganancia 142 (óptima)

# Job Scheduling - Análisis Comparativo

| **Enfoque** | **Complejidad** | **Optimalidad** |
|---|---|---|
| Fuerza Bruta | $O(n! \cdot n)$ | ✓ |
| Greedy | $O(n \log n + n \cdot d)$ | × |

## Conclusión

- Greedy no siempre es óptimo pero es muy eficiente
- En la práctica funciona bien para la mayoría de casos
- Fuerza bruta solo para instancias muy pequeñas ($n < 10$)

# Tabla de Contenidos

# Huffman Codes - Enunciado

## Descripción del Problema

Encontrar código de prefijo óptimo para compresión de datos:

- Símbolos con frecuencias dadas
- Código de longitud variable
- Objetivo: Minimizar longitud promedio
- Costo $= \sum f_i \cdot l_i$

## Aplicaciones

- Compresión de archivos (ZIP, JPEG, MP3)
- Transmisión eficiente de datos

# Huffman Codes - Fuerza Bruta

## Estrategia

- Generar todos los árboles binarios completos con n hojas
- Para cada árbol, calcular longitudes de código
- Probar todas las asignaciones de símbolos a hojas
- Seleccionar asignación con mínimo costo

## Complejidad

$O(C_{n-1} \cdot n! \cdot n)$ - Super-exponencial

# Huffman Codes - Pseudocódigo Fuerza Bruta

**Entrada:** Símbolos y frecuencias
**Salida:** Asignación óptima de códigos

```
n ← número de símbolos
best_cost ← ∞, best_assignment ← null
for cada tree en generate_trees(n) do
    lengths ← code_lengths(tree)
    for cada perm en permutations(range(n)) do
        cost ← ∑ freqs[perm[i]] · lengths[i]
        if cost < best_cost then
            best_cost ← cost, best_assignment ← asignación
        end if
    end for
end forreturn best_cost, best_assignment
```

# Huffman Codes - Ejemplo Fuerza Bruta

## 4 símbolos: [A:5, B:2, C:1, D:1]

- Número de árboles: $C_3 = 5$
- Permutaciones por árbol: $4! = 24$
- Total: $5 \times 24 = 120$ asignaciones
- **Mejor**: A→1, B→2, C→2, D→3 (Costo=14)

## Limitaciones

6 símbolos: $C_5 = 42$ árboles $\times$ $6! = 720 \times 6 = 181{,}440$ operaciones

## Estrategia Óptima

- Usar min-heap por frecuencias
- Combinar siempre los dos nodos de menor frecuencia
- Construir árbol de abajo hacia arriba

## Complejidad

$O(n \log n)$ - Polinomial (óptimo)

# Huffman Codes - Pseudocódigo Greedy

**Entrada:** Símbolos y frecuencias
**Salida:** Códigos Huffman óptimos

```
pq ← min-heap vacío
for cada símbolo do
    pq.push(Node(freq))
end for
while |pq| > 1 do
    l ← pq.pop(), r ← pq.pop()
    new ← Node(l.freq + r.freq)
    new.left ← l, new.right ← r
    pq.push(new)
end while
root ← pq.pop()
Generar códigos con recorrido preorden return códigos
```

# Huffman Codes - Ejemplo Greedy

## Símbolos: [a:5, b:9, c:12, d:13, e:16, f:45]

- Combinar $a(5) + b(9) = 14$
- Combinar $c(12) + d(13) = 25$
- Combinar $14 + e(16) = 30$
- Combinar $25 + 30 = 55$
- Combinar $f(45) + 55 = 100$

## Resultado

Códigos óptimos: f$\to$0, c$\to$100, d$\to$101, a$\to$1100, b$\to$1101, e$\to$111

# Huffman Codes - Análisis Comparativo

| **Enfoque** | **Complejidad** | **Optimalidad** |
|---|---|---|
| Fuerza Bruta | $O(C_{n-1} \cdot n! \cdot n)$ | ✓ |
| Greedy | $O(n \log n)$ | ✓ |

## Conclusión

- Huffman greedy es óptimo y eficiente
- Raro caso donde greedy garantiza optimalidad
- Fuerza bruta completamente impráctica
- Algoritmo ampliamente usado en compresión

# Tabla de Contenidos

# Conclusiones Generales

## Comparativa de Enfoques

| **Problema** | **BF Optimal** | **Greedy Optimal** | **BF Complejidad** | **Greedy Complejidad** |
|---|---|---|---|---|
| Activity Selection | ✓ | ✓ | $O(2^n n^2)$ | $O(n \log n)$ |
| Graph Coloring | ✓ | × | $O(k^n nm)$ | $O(V^2 + E)$ |
| Job Scheduling | ✓ | × | $O(n!n)$ | $O(n \log n + nd)$ |
| Huffman Codes | ✓ | ✓ | $O(C_{n-1} n! n)$ | $O(n \log n)$ |

## Insights Clave

- Greedy es óptimo cuando el problema exhibe **subestructura óptima**
- La **elección greedy** debe preservar la posibilidad de solución óptima
- Fuerza bruta es invaluable para verificar correctitud de algoritmos

# Recomendaciones Prácticas

## Cuándo usar cada enfoque

- **Fuerza Bruta**:
  - Problemas pequeños ($n \leq 20$)
  - Verificación de algoritmos
  - Casos donde se necesita optimalidad garantizada
- **Greedy**:
  - Problemas grandes donde optimalidad no es crítica
  - Cuando el problema exhibe subestructura óptima
  - Aplicaciones en tiempo real que requieren rapidez

## Estrategia Híbrida

En la práctica: Empezar con greedy, usar fuerza bruta para casos pequeños o como verificación

# ¿Preguntas?

**Contacto**

- Delgado Romero
- Torres Reategui
- Quintanilla Quispe

¡Gracias por su atención!