# Análisis y Diseño de Algoritmos

Examen Diciembre 2024

Francisco Javier Mercader Martínez

**Tema 2.** Divide y vencerás

**Subsecuencia mayor a un número.** Nos piden encontrar, dentro de una secuencia $S$ de $n$ números indexados por $i = 1..n$, cuál es la subsecuencia más larga con valores por encima de límite $t$.

Por ejemplo, si

```
S = [1,6,7,4,5,8,6,9,2,10]
t = 5
```

el resultado sería la subsecuencia `[5, 8, 6, 9]`, entre los índices 5 y 8, con longitud 4.

```python
def solucion_directa(S, t, ini, fin):
    # Busca la mejor subsecuencia directamente en el rango [ini, fin]
    mejor_ini = mejor_fin = -1
    actual_ini = None
    mejor_longitud = 0

    for i in range(ini, fin + 1):
        if S[i] >= t:
            if actual_ini is None:
                actual_ini = i # Comienza nueva subsecuencia
        else:
            if actual_ini is not None:
                # Finaliza la subsecuencia actual
                longitud = i - actual_ini
                if longitud > mejor_longitud:
                    mejor_ini = actual_ini
                    mejor_fin = i - 1
                    mejor_longitud = longitud
                actual_ini = None

    # Verifica si la última subsecuencia llegó hasta el final
    if actual_ini is not None:
        longitud = fin - actual_ini + 1
        if longitud > mejor_longitud:
            mejor_ini = actual_ini
            mejor_fin = fin
            mejor_longitud = longitud

    return (mejor_ini, mejor_fin, mejor_longitud)

def combinar(S, t, ini, mid, fin):
    # Combina subsecuencias que cruzan el punto medio

    # Parte izquierda: desde el centro hacia la izquierda
    i = mid
    while i >= ini and S[i] >= t:
        i -= 1
    izquierda_ini = i + 1
    izquierda_long = mid - i

    # Parte derecha: desde el centro+1 hacia la derecha
```

```
    j = mid + 1
    while j <= fin and S[j] >= t:
        j += 1
    derecha_fin = j - 1
    derecha_long = j - (mid + 1)

    # Longitud total de la subsecuencia que cruza el medio
    total_long = izquierda_long + derecha_long

    if total_long == 0:
        return (-1, -1, 0) # No hay subsecuencia cruzada válida
    else:
        return (izquierda_ini, derecha_fin, total_long)

def DyV(S, t, ini, fin):
    # Algoritmo Divide y Vencerás
    if ini == fin:
        # Caso base: un solo elemento
        if S[ini] >= t:
            return (ini, ini, 1)
        else:
            return (-1, -1, 0)

    mid = (ini + fin) // 2 # Punto medio

    # Resuelve las mitades
    izquierda = DyV(S, t, ini, mid)
    derecha = DyV(S, t, mid + 1, fin)
    cruzada = combinar(S, t, ini, mid, fin)

    # Devuelve la subsecuencia más larga entre las tres opciones
    return max([izquierda, derecha, cruzada], key=lambda x: x[2])

# Función principal que llama al algoritmo DyV
def subsecuencia_mayor_DyV(S, t):
    ini, fin, longitud = DyV(S, t, 0, len(S) - 1)
    if longitud == 0:
        return [] # No hay subsecuencia válida
    return S[ini:fin + 1]
```

ENUNCIADO DE PROBLEMA para los temas 3 a 5:

---

**Volver a mi sombrila sin quemarme.** Tenemos que atravesar de izquierda a derecha una playa llena de sombrillas para llegar hasta nuestra propia sombrilla. La playa está descrita por un tablero 2D $T$ con $n \times n$ casillas ($n$ impar). Cada casilla $(i, j)$ contiene un número $T[i, j]$ que es o bien $-1$ para indicar que hay una sombrilla (no puedo pasar) o un natural que indica cuánto quema esa casilla. Desde la casilla actual, $(i, j)$, te puedes mover a las casillas de la columna a la derecha que estén en la misma fila o en la superior o inferior, es decir, $(i-1, j+1), (i, j+1)$ ó $(i+1, j+1)$.

Tu objetivo es, partiendo de la casilla $((n+1)/2, 1)$ (casilla central de la primera columna), llegar a tu sombre, en la casilla central de la últimaa columna $((n+1)/2, n)$, no ahbindo pisado ninguna sombrilla, y minimizando el chamuscado de pies (la suma de los números correspondientes a las casillas atravesadas). Date cuenta de que puede no haber solución, por ejemplo, tribialmente, si $T[(n+1)/2, 1] = -1$. Se adjunt un ejemplo de tablero, con la solución óptima en negrita:

| | | | | | | |
|---|---|---|---|---|---|---|
| | 8 | 8 | 3 | 8 | 8 | |
| | 8 | 2 | 2 | -1 | 8 | |
| Inicio≫ | **1** | -1 | -1 | 2 | **1** | ≫fin |
| | 8 | **1** | -1 | **1** | 8 | |
| | 8 | 8 | **1** | 8 | 8 | |

**Tema 3 (AR).** Diseñar un algoritmo voraz que encuentre una bunea forma de resolver el problema (código). Hay que ajustarse al esquema y desarrollar sus funciones (código). ¿Garantiza ese algoritmo la solución óptima? Razonarlo.

```
def generar_candidatos(s, T, n):
```

```python
    # Genera movimientos posibles desde la posición actual hacia la derecha
    i, j = s[-1] # Última posición en el camino actual
    candidatos = []
    for di in [-1, 0, 1]: # Movimientos, diagonal arriba, derecha, diagonal abajo
        ni, nj = i + di, j + 1
        # Verifica que la nueva posición esté dentro del tablero y no sea una sombrilla (-1)
        if 0 <= ni < n and nj < n and T[ni][nj] != -1:
            candidatos.append((ni, nj))
    return candidatos

def seleccionar(candidatos, T):
    # Selecciona el candidato con menor valor (menor chamusqueo)
    return min(candidatos, key=lambda x: T[x[0]][x[1]])

def factible(s, x):
    # Comprueba que no se repita la casilla (evitar bucles)
    return x not in s

def insertar(s, x):
    # Agrega la casilla seleccionada al camino
    s.append(x)

def solucion(s, n):
    # Verifica si se ha alcanzado la última columna del tablero
    return s[-1][1] == n - 1

def quedan_pasos(s, n):
    # Verifica si aún no se ha llegado al final del tablero
    return s[-1][1] < n - 1

def objetivo(s, T):
    # Calcula el chamusqueo total (suma de valores del camino recorrido)
    return sum(T[i][j] for i, j in s)

def voraz(T):
    n = len(T) # Tamaño del tablero (n x n)
    fila_inicio = n // 2 # Fila central (posición inicial)
    s = [(fila_inicio, 0)] # Inicializamos el camino con la casilla inicial

    while True:
        c = generar_candidatos(s, T, n) # Obtener candidadtos posibles desde la posición actual
        if not c:
            break # No hay movimientos posibles
        x = seleccionar(c, T) # Elegir el mejor candidato
        if factible(s, x):
            insertar(s, x) # Agregarlo al camino
        if solucion(s, n) or not quedan_pasos(s, n):
            break # Si se alcanza el final o no hay más pasos posibles, terminar

    if not solucion(s, n):
        return "No se puede encontrar solución"
    return s, objetivo(s, T) # Devolver el camino y el chamusqueo total

if __name__ == '__main__':
    T = [
        [8,  8,  3,  8, 8],
        [8,  2,  2, -1, 8],
        [1, -1, -1,  2, 1],
        [8,  1, -1,  1, 8],
        [8,  8,  1,  8, 8]
    ]

    camino, coste = voraz(T)
    print("Camino:", camino)
```

```
    print("Chamusqueo total:", coste)
```

```
Camino: [(2, 0), (3, 1), (4, 2), (3, 3), (2, 4)]
Chamusqueo total: 5
```

El algoritmo voraz no garantiza encontrar la solución óptima, porque las decisiones locales no aseguran el mejor camino global. Puede ignorar caminos con costo inicial alto pero que después resultan mejores.

**Tema 4 (BT).** Resolver el problema de forma óptima por *backtracking*. Se deberán utilizar los esquemas vistos en clase. Definir la forma de representar la solución, el tipo de árbol usado, el esquema y las funciones y las funciones genéricas del esquema (código). Seguir los pasos de desarrollo vistos en clase. Se valorarán mecanismos de mejora de la eficiencia que se desarrollen o al menos expliquen.

```
# Representación de la solución: lista de posiciones (i, j)
# Árbol: árbol implícito de decisiones, cada nivel corresponde a una columna del tablero

# Funciones y algoritmo siguiendo el esquema clásico de backtracking
def backtracking(T):
    n = len(T) # Tamaño del tablero
    mejor_camino = [] # Lista que almacenará la mejor secuencia de pasos encontrada
    mejor_coste = float('inf') # Inicializamos el mejor coste como infinito

    # Posición inicial: casilla central de la primera columna
    fila_inicio = n // 2
    inicio = [(fila_inicio, 0)] # Comenzamos en la posición central de la primera columna
    coste_inicial = T[fila_inicio][0] # Coste inicial es el valor de la casilla

    def bact(nivel, camino, coste):
        nonlocal mejor_camino, mejor_coste # Variables externas que queremos modificar dentro de
            la función

        # Caso base: si estamos en la última columna
        if nivel == n - 1:
            if coste < mejor_coste: # Si encontramos un coste mejor, lo guardamos
                mejor_camino = camino[:] # Copiamos el camino actual como mejor
                mejor_coste = coste
            return # Salimos de este rama

        i, j = camino[-1] # Tomamos la última posición visitada

        # Exploramos las tres posibles direcciones (diagonal arriba, derecha, diagonal abajo)
        for di in [-1, 0, 1]:
            ni, nj = i + di, j + 1 # Nueva posición a la derecha
            # Verificamos que esté dentro del tablero, no sea sombrilla y no repetida
            if 0 <= ni < n and nj < n and T[ni][nj] != -1 and (ni, nj) not in camino:
                nuevo_coste = coste + T[ni][nj] # Actualizamos el coste
                # Poda: si ya sabemos que esta ruta no puede ser mejor, la evitamos
                if nuevo_coste < mejor_coste:
                    camino.append((ni, nj)) # Avanzamos a la nueva casilla
                    bact(nivel + 1, camino, nuevo_coste) # Llamada recursiva al siguiente nivel
                    camino.pop() # Volve atrás para explorar otros caminos (backtrack)

    # Comenzamos el backtracking desde el primero nivel (columna 0)
    bact(0, inicio, coste_inicial)

    # Si encontramos solución, la devolvemos junto al coste mínimo
    if mejor_camino:
        return mejor_camino, mejor_coste
    else:
        return "No se puede encontrar solución"

if __name__ == '__main__':
    T = [
        [8,  8,  3,  8, 8],
        [8,  2,  2, -1, 8],
```

```
        [1, -1, -1,  2, 1],
        [8,  1, -1,  1, 8],
        [8,  8,  1,  8, 8]
    ]

    camino, coste = backtracking(T)
    print(f"Camino óptimo: {camino}")
    print(f"Chamusqueo mínimo: {coste}")
```

```
Camino óptimo: [(2, 0), (3, 1), (4, 2), (3, 3), (2, 4)]
Chamusqueo mínimo: 5
```

**Tema 5 (PD).** Resolver este problema mediante programación dinámica indicando la ecuación de recurrencia utilizada, los casos base, las tablas necesarias, la forma de rellenar las tablas (código), la forma de reconstruir la solución (código) y el resto de los pasos vistos en clase. Indicar cómo se save si hay varias soluciones óptimas distintas.

```
# --------------------------------------
# Formulación del problema:
# D[i][j] = mínimo chamusqueo para llegar a la casilla (i, j) desde la casilla inicial (n//2, 0)
# Recurrencia:
# D[i][j] = T[i][j] + min(D[i-1][j-1], D[i][j-1], D[i+1][j-1])  (si la casilla (i,j) es válida)
# Casos base:
# D[n//2][0] = T[n//2][0] (posición inicial)
# En el resto de D se inicializa en infinito excepto sombrillas (-1)
# --------------------------------------

def programacion_dinamica(T):
    n = len(T) # Tamaño del tablero (n x n)
    INF = float('inf') # Valor representando infinito (usado para inicializar)

    D = [[INF] * n for _ in range(n)] # Matriz para almacenar el coste mínimo para llegar a cada
        casilla
    P = [[None] * n for _ in range(n)] # Matriz para guardar el predecesor de cada casilla (para
        reconstruir)

    fila_inicio = n // 2 # Fila central (punto de partida)
    if T[fila_inicio][0] == -1:
        return "No se puede encontrar solución" # Si se empieza sobre una sobrilla, no hay soluci
            ón

    D[fila_inicio][0] = T[fila_inicio][0] # Caso base: coste de la casilla inicial

    # Rellenar la tabla D con programación dinámica
    for j in range(1, n): # Recorremos columnas de izquierda a derecha
        for i in range(n): # Recorremos filas
            if T[i][j] == -1:
                continue # Ignoramos sombrillas (caillas no transitables)
            for di in [-1, 0, 1]: # Posibles movimientos dese la columna anterior
                pi = i + di
                if 0 <= pi < n and D[pi][j - 1] != INF:
                    nuevo_coste = D[pi][j - 1] + T[i][j] # Suma del coste anterior más el actual
                    if nuevo_coste < D[i][j]: # Si encontramos un coste más bajo, lo actualizamos
                        D[i][j] = nuevo_coste
                        P[i][j] = (pi, j - 1) # Guardamos el predecesor (de dónde venimos)

    # Buscar la casilla de menor coste en la última columna
    mejor_coste = INF
    fin_fila = -1
    for i in range(n):
        if D[i][n - 1] < mejor_coste:
            mejor_coste = D[i][n - 1]
            fin_fila = i # Recordamos la fila final con menor coste

    if mejor_coste == INF:
        return "No se pudo encontrar solución" # Si ningún camino llega a la última columna
```

```python
    # Reconstrucción del camino desde la casilla final hacia atrás
    camino = []
    i, j = fin_fila, n - 1
    while i is not None and j is not None:
        camino.append((i, j)) # Añadimos la casilla actual al camino
        i, j = P[i][j] if P[i][j] is not None else (None, None) # Saltamos al predecesor

    camino.reverse() # Invertimos el camino para que vaya de izquierda a derecha

    return camino, mejor_coste # Devolvemos el camino y el coste mínimo

# --------------------------------------
# ¡Cómo saber si hay múltiples soluciones óptimas?
# Durante el llenado de D[i][j], si encontramos otro predecesor con el mismo coste mínimo,
# se puede llevar un contador o lista de caminos posibles para identificar soluciones múltiples
# --------------------------------------

if __name__ == '__main__':
    T = [
        [8,  8,  3,  8, 8],
        [8,  2,  2, -1, 8],
        [1, -1, -1,  2, 1],
        [8,  1, -1,  1, 8],
        [8,  8,  1,  8, 8]
    ]

    resultado = programacion_dinamica(T)
    if isinstance(resultado, str):  # Si la función devolvió un mensaje de error
        print(resultado)
    else:
        camino, coste = resultado
        print("Camino óptimo:", camino)
        print("Chamusqueo mínimo:", coste)
```

```
Camino óptimo: [(2, 0), (3, 1), (4, 2), (3, 3), (2, 4)]
Chamusqueo mínimo: 5
```