Análisis y Diseño de Algoritmos

Examen Diciembre 2023

Francisco Javier Mercader Martínez

1. a) Calcular el tiempo de ecución de este algoritmo y expresarlo con O, Ω y Θ:

```
def Al-Juarismi(M,n): # M es un array nxn
    i = 1
    while i<=sqrt(n):
        if M[i,i] % 2 == 0:      # si M[i,i] es par
            for j in range(1, n+1)
                M[i,i] += M[j,j]
        else:
          cont=0
          j=1
          while (M[i,j] % 2 != 0) and (j<=n):
              cont = 0
              j = 1
              while (M[i,j] % 2 != 0) and (j<=n):
                  cont += M[i,j]
                  for k in range(1, n+1):
                      cont += M[k,k]
                  j = j*2
              M[i,i] = acum
          i=i+1
```

Análisis del código:

1. El ciclo externo **`while`** itera hasta **`sqrt(n)`**. El número de iteraciones es proporcional a $O(\sqrt{n})$.
2. Dentro del ciclo:
   - Si el elemento diagonal es par, hay un bucle anidado que itera $O(n)$ veces para sumar elementos diagonales. Por tanto, esta rama toma $O(n)$.
   - Si el elemento es impar:
     - Hay un ciclo **`while`** controlado por **`j`** que itera $O(n)$ veces.
     - Dentro de este ciclo, el bucle anidado **`for`** también recorre $O(n)$.
     - Por lo tanto, esta rama tiene un costo $O(n^2)$.

Costo total del algoritmo:

- El ciclo externo tiene $O(\sqrt{n})$ iteraciones, y en el peor caso, la rama más costosa $O(n^2)$ domina.
- Entonces, el costo total del algoritmo es:

$$O(\sqrt{n} \cdot n^2) = O\left(n^{\frac{5}{2}}\right)$$

Clasificación asintótica:

- $O\left(n^{\frac{5}{2}}\right)$: Tiempo de ejecución en el peor caso.
- $\Omega(n^2)$: Mejor caso, si todas las operaciones son $O(n)$ dentro del ciclo externo.
- $\Theta\left(n^{\frac{2}{5}}\right)$: Representa el comportamiento promedio si ocurre la combinación de ramas más costosa

b) Determinar el orden de esta ecuación de recurrencia:

$$\begin{array}{ll} t(n) = 1 & \text{si } n < 1 \\ t(n) = 8t(n/4) + n^2 & \text{si } n \geq 1 \end{array}$$

Resolución mediante el teorema maestro:

- $a = 8, b = 4, f(n) = n^2$.
- Calculamos el exponente crítico $p_c = \log_b a = \log_4 8 = \frac{3}{2}$
- Comparación con $f(n) = n^2$:
  - $p_c = \frac{3}{2} < 2 \longrightarrow f(n)$ domina.

Por lo tanto, $t(n) \in \Theta(n^2)$.

c) Resolver esta ecuación de recurrencia:

$$\begin{array}{ll} t(n) = 1 & \text{si } n < 1 \\ t(n) = 3t(n-1) + 3 & \text{si } n \geq 1 \end{array}$$

Resolución iterativa: Expandimos los términos:

$$t(n) = 3t(n-1) + 3 = 3[t(n-2) + 3] + 3 = 3^2 t(n-2) + 3^2 + 3$$

Continuando:

$$t(n) = 3^k t(n-k) + 3(3^{k-1} + \cdots + 3 + 1).$$

Cuando $k = n, t(n-k) = t(0) = 1$:

$$t(n) = 3^n \cdot 1 + 3(3^{n-1} + \cdots + 3 + 1).$$

La suma geométrica $3^{n-1} + \cdots + 1$ es:

$$\frac{3^n - 1}{3 - 1}$$

2. Sustituyendo:

$$t(n) = 3^n + 3 \cdot \frac{3^n - 1}{2} = 3^n + \frac{3^n - 3}{2}.$$

Divide y vencerás

Subsecuencia ascendente. Nos piden encontrar dentro de una secuencia S de n números indexados por `i=1..n`, cuál es la subsecuencia ascendente más larga.

Por ejemplo, si

`S = [ 1 2 4 8 10 13 7 8 9 10 12 6],`

el resultado sería la subsecuencia `[1 2 4 8 10 13]`, entre los índices 1 y 6, con longitud 6.

```
def DyV(p, q):
    if p == q: # Caso base: subsecuencia de un solo elemento
        return (p, q) # Solución directa
    else:
        m = (p+q) // 2 # Dividir la secuencia
        s1 = DyV(p, m) # Resolver la mitad izquierda
        s2 = DyV(m+1, q) # Resolver la mitad derecha
        return combinar(s1, s2, p, q, m) # Combinar resultados

def combinar(s1, s2, p, q, m):
    if S[m] >= S[m + 1]: # No hay continuidad en la frontera
        return cadena_mas_larga(s1, s2) # Devuelve la cadena más larga entre s1 y s2
    # Si hay continuidad en la frontera
    (p1, q1) = s1
    (p2, q2) = s2

    if q1 == m: # s1 pegada a la frontera izquierda
        if p2 == m + 1: # s2 pegada a la frontera derecha
            return (p1, q2) # Concatenación de s1 y s2
        else:
            q1 = extender_hacia_derecha(q1)
            return cadena_mas_larga(s1, s2)

    if p2 == m + 1: # s2 pegada a la frontera derecha
        p2 = extender_hacia_izquierda(p2)
```

```
        return cadena_mas_larga(s1,s2)

    # Ni s1 ni s2 pegadas a la frontera
    p3 = extender_hacia_izquierda(m)
    q3 = extender_hacia_derecha(m + 1)
    s3 = (p3, q3)
    return cadena_mas_larga(s1, s2, s3)

def cadena_mas_larga(s1, s2, s3=None):
    (p1, q1) = s1
    (p2, q2) = s2
    if s3 is None: # Comparación entre dos subsecuencias
        if q1 - p1 > q2 - p2:
            return s1
        else:
            return s2
    else: # Comparación entre tres subsecuencias
        (p3, q3) = s3
        longitud1 = q1 - p1
        longitud2 = q2 - p2
        longitud3 = q3 - p3
        if longitud1 >= longitud2 and longitud1 >= longitud3:
            return s1
        elif longitud2 >= longitud1 and longitud2 >= longitud3:
            return s2
        else:
            return s3

def extender_hacia_izquierda(i):
    while S[i - 1] < S[i]: # Extender mientras la secuencia sea ascendente
        i -= 1
    return i

def extender_hacia_derecha(i):
    while S[i] < S[i + 1]: # Extender mientras la secuencia sea ascendente
        i += 1
    return i

# Ejemplo de uso
S = [1, 2, 4, 8, 10, 13, 7, 8, 9, 10, 12, 6] # Secuencia de entrada
n = len(S)
resultado = DyV(0, n - 1)
print(f"La subsecuencia ascendente más larga es: {S[resultado[0]:resultado[1] + 1]}")
```

```
## La subsecuencia ascendente más larga es: [1, 2, 4, 8, 10, 13]
```

```
print(f"Longitud de la subsecuencia: {resultado[1] - resultado[0] + 1}")
```

```
## Longitud de la subsecuencia: 6
```

Explicación del código

1. **`DyV(p, q)`:** - - Divide la secuencia en dos mitades y llama recursivamente a S_L (izquierda) y S_R (derecha). - Combina las soluciones usando la función **`combinar`**.
2. **`combinar(s1, s2, p, q, m)`:**
   - Verifica si hay continuidad en la frontera ($S[m] < S[m+1]$).
   - Extiende las subsecuencias hacia la izquierda o la derecha si están pegadas a la frontera.
   - Devuelve la subsecuencia más larga.
3. **`cadena_mas_larga(s1,s2,s3)`:**
   - Compara hasta tres subsecuencias y devuelve la que tiene mayor longitud.
4. **`extender_hacia_izquierda(i) y extender_hacia_derecha(i)`:**

- Extienden la subsecuencia hacia la izquierda o derecha mientras sea ascendente.

Camino con transporte público. Para poder moverte por tu ciudad de forma eficiente usando el transporte público, has pensado en diseñar un algoritmo que te permita llega desde un origen a un destino en el menor tiempo posible. Para ello, se modela la ciudad como un table **D** n**x**m, en donde cada celda indica el coste de llegar a dicha celda en autobús (A) o taxi (T) desde una casilla adyacente. En este sentido, se supone que el origen se encuentra en la casilla superior izquierda (1,1) y el destino en la casilla inferior derecha (n,m). Además, sólo se permiten los movimientos de izquierda a derecha y de arriba abajo. Se puede cambiar de medio de transporte siempre que se quiera. Un ejemplo de dicha tabla sería el siguiente

A:0,T:0	A:1,T:2	A:2,T:5	A:6,T:6
A:2,T:3	A:1,T:3	A:5,T:2	A:1,T:1
A:1,T:2	A:2,T:3	A:2,T:4	A:4,T:5

Así, la tupla en la casilla fila 1, columna 2 (A:1, T:2) indica que el tiempo para llega a dicha casilla en autobús (A) es 1 unidad de tiempo mientras que en taxi (T) es de 2 unidades de tiempo.

3. Diseñar un algoritmo voraz que encuentre una buena forma de resolver el problema. Hay que ajustarse al esquema y desarrollar sus funciones. ¿Garantiza ese algoritmo la solución óptima? Razonarlo.

```python
def camino_voraz(D):
    n = len(D) # Número de filas
    m = len(D[0]) # Número de columnas
    i, j = 0, 0 # Posición inicial
    tiempo_total = 0
    transporte = "A" # Empezamos con cualquier transporte, aquí elegimos autobús

    while i < n - 1 or j < m - 1: # Mientras no lleguemos al destino (n-1, m-1)
        # Opciones de movimiento: derecha o abajo
        coste_derecha = float('inf')
        coste_abajo = float('inf')

        if j + 1 < m: # Movimiento a la derecha
            coste_derecha = min(D[i][j + 1]['A'], D[i][j + 1]['T'])

        if i + 1 < n: # Movimiento hacia abajo
            coste_abajo = min(D[i + 1][j]['A'], D[i + 1][j]['T'])

        # Decisión voraz: elegir el menor coste inmediato
        if coste_derecha < coste_abajo:
            j += 1 # Moverse a la derecha
            transporte = 'A' if D[i][j]['A'] < D[i][j]['T'] else 'T'
            tiempo_total += D[i][j][transporte]
        else:
            i += 1 # Moverse hacia abajo
            transporte = 'A' if D[i][j]['A'] < D[i][j]['T'] else 'T'
            tiempo_total += D[i][j][transporte]

    return tiempo_total

# Ejemplo de entrada
D = [
    [{'A': 0, 'T': 0}, {'A': 1, 'T': 2}, {'A': 2, 'T': 5}, {'A': 6, 'T': 6}],
    [{'A': 2, 'T': 3}, {'A': 1, 'T': 3}, {'A': 5, 'T': 2}, {'A': 1, 'T': 1}],
    [{'A': 1, 'T': 2}, {'A': 2, 'T': 3}, {'A': 2, 'T': 4}, {'A': 4, 'T': 5}]
]
# Llamada a la función
resultado = camino_voraz(D)
print("El tiempo total con el algoritmo voraz es:", resultado)
```

```
## El tiempo total con el algoritmo voraz es: 10
```

Explicación del algoritmo

1. **Posición inicial:** Empezamos en la esquina superior izquierda $(0, 0)$.

2. **Evaluación de opciones:** En cada paso, el algoritmo evalúa los **costes inmediatos** de moverse:
   - **A la derecha** $(i, j+1)$.
   - **Hacia abajo** $(i+1, j)$.
3. **Decisión voraz:** Elige la dirección y el transporte con el menor coste inmediato.
4. **Suma de tiempos:** Acumula el tiempo correspondiente.
5. **Finalización:** El algoritmo termina cuando alcanza la esquina inferior derecha $(n-1, m-1)$.

El algoritmo **no siempre** garantiza la solución óptima, ya que toma **decisiones locales óptimas** sin considerar el efecto futuro de estas elecciones. Aunque es rápido y fácil de implementar, en este problema concreto, la **programación dinámica** es la única forma de asegurar la solución óptima.

4. Resolver el problema de forma óptima por *backtracking*. Se deberán utilizar los esquemas visto en clase. Definir la forma de representar la solución, el tipo de árbol usado, el esquema y las funciones genéricas del esquema. Seguir los pasos de desarrollo vistos en clase. Se valorará cualquier mecanismo de poda que se desarrolle.

```python
def bactracking_solucion(inicial):
    nivel = 0
    solucion_parcial = inicial
    mejor_solucion = None
    mejor_tiempo = float('inf')
    fin = False

    def backtrack(i, j, tiempo_acumulado, transporte_actual):
        nonlocal mejor_solucion, mejor_tiempo

        # Caso base: si hemos llegado a la celda destino
        if i == n - 1 and j == m - 1:
            if tiempo_acumulado < mejor_tiempo:
                mejor_tiempo = tiempo_acumulado
                mejor_solucion = list(solucion_parcial)
        return

        # Poda: si el tiempo acumulado ya es mayor al mejor tiempo encontrado
        if tiempo_acumulado >= mejor_tiempo:
            return

        # Generar hijos (movimientos posibles: derecha y abajo)
        movimientos = []
        if j + 1 < m: # Movimiento a la derecha
            movimientos.append((i, j + 1))
        if i + 1 < n: # Movimientos hacia abajo
            movimientos.append((i + 1, j))

        for (nueva_i, nueva_j) in movimientos:
            # Evaluar los costes con ambos transportes
            tiempo_a = D[nueva_i][nueva_j]["A"]
            tiempo_t = D[nueva_i][nueva_j]["T"]

            # Movimiento con autobús
            solucion_parcial.append((nueva_i, nueva_j, "A"))
            backtrack(nueva_i, nueva_j, tiempo_acumulado + tiempo_a, "A")
            solucion_parcial.pop()

            # Movimiento con taxi
            solucion_parcial.append((nueva_i, nueva_j, "T"))
            backtrack(nueva_i, nueva_j, tiempo_acumulado + tiempo_t, "T")
            solucion_parcial.pop()

    # LLamada inicial
    solucion_parcial = [(0,0,"A")] # Empezamos con autobús
    backtrack(0,0, D[0][0]["A"], "A")

    solucion_parcial = [(0,0,"T")] # Empezamos con taxi
    backtrack(0,0, D[0][0]["T"], "T")
```

```
    return mejor_tiempo, mejor_solucion

# Ejemplo de entrada
D = [
    [{'A': 0, 'T': 0}, {'A': 1, 'T': 2}, {'A': 2, 'T': 5}, {'A': 6, 'T': 6}],
    [{'A': 2, 'T': 3}, {'A': 1, 'T': 3}, {'A': 5, 'T': 2}, {'A': 1, 'T': 1}],
    [{'A': 1, 'T': 2}, {'A': 2, 'T': 3}, {'A': 2, 'T': 4}, {'A': 4, 'T': 5}]
]
# Llamada a la función
resultado = camino_voraz(D)
print("El tiempo total con el algoritmo voraz es:", resultado)
```

```
## El tiempo total con el algoritmo voraz es: 10
```

Explicación de la ejecución

1. El algoritmo comienza en $(0, 0)$ y prueba los dos transportes disponibles: autobús y taxi.
2. En cada celda, evalúa los **movimientos posibles** (derecha y abajo) y toma ambos transportes en cada caso.
3. La **poda** evita continuar por caminos cuyo tiempo acumulado supera la mejor solución encontrada.
4. El algoritmo **retrocede** al nivel anterior después de probar cada movimiento y transporte

Mecanismos de poda

- **Poda por tiempo acumulado:** Si el tiempo acumulado en una ruta parcial es mayor que el mejor tiempo actual, se detiene la exploración.
- **Eliminación de caminos redundantes:** Si un nodo ya ha sido visitado con menor tiempo acumulado, no se vuelve a explorar.

5. Resolver este problema mediante programación dinámica indicando la ecuación de recurrencia utilizada, los casos base, las tablas necesarias, la forma de rellenar las tablas, la forma de reconstruir la solución y el resto de los pasos vistos en clase. Indicar cómo se sabe si hay varias soluciones óptimas distintas. ¿Habrías alguna manera de contar el número de transbordos que el usuario realiza a lo largo de un viaje?

```
def tiempo_minimo(D):
    n = len(D)      # Número de filas
    m = len(D[0])  # Número de columnas

    # Inicializar tablas A y T con infinito
    A = [[float("inf")] * m for _ in range(n)]
    T = [[float("inf")] * m for _ in range(n)]

    # Casos base: celda inicial
    A[0][0] = D[0][0]["A"]
    T[0][0] = D[0][0]["T"]

    # Rellenar las tablas
    for i in range(n):
        for j in range(m):
            if i > 0:   # Desde arriba
                A[i][j] = min(A[i][j], min(A[i-1][j], T[i-1][j]) + D[i][j]["A"])
                T[i][j] = min(T[i][j], min(A[i-1][j], T[i-1][j]) + D[i][j]["T"])
            if j > 0:   # Desde abajo
                A[i][j] = min(A[i][j], min(A[i][j-1], T[i][j-1]) + D[i][j]["A"])
                T[i][j] = min(T[i][j], min(A[i][j-1], T[i][j-1]) + D[i][j]["T"])


    # Solución óptima: mínimo entre A y T en la celda destino
    tiempo_minimo = min(A[-1][-1], T[-1][-1])
    return tiempo_minimo, A, T
```

```
def reconstruir_ruta(A, T, D):
    i, j = len(A) - 1, len(A[0]) - 1
    ruta = []
```

```
    transporte = 'A' if A[i][j] <= T[i][j] else 'T'

    while i > 0 or j > 0:
        ruta.append((i, j, transporte))
        if transporte == 'A':
            if i > 0 and A[i][j] == A[i-1][j] + D[i][j]['A']:
                i -= 1
            else:
                j -= 1
        else:
            if i > 0 and T[i][j] == T[i-1][j] + D[i][j]['T']:
                i -= 1
            else:
                j -= 1
        transporte = 'A' if A[i][j] <= T[i][j] else 'T'

    ruta.append((0, 0, transporte))
    return ruta[::-1]
```

```
def contar_transbordos(ruta):
    transbordos = 0
    for k in range(1, len(ruta)):
        if ruta[k][2] != ruta[k - 1][2]:  # Cambia el transporte
            transbordos += 1
    return transbordos
```

```
D = [
    [{'A': 0, 'T': 0}, {'A': 1, 'T': 2}, {'A': 2, 'T': 5}, {'A': 6, 'T': 6}],
    [{'A': 2, 'T': 3}, {'A': 1, 'T': 3}, {'A': 5, 'T': 2}, {'A': 1, 'T': 1}],
    [{'A': 1, 'T': 2}, {'A': 2, 'T': 3}, {'A': 2, 'T': 4}, {'A': 4, 'T': 5}]
]

tiempo_minimo, A, T = tiempo_minimo(D)
ruta = reconstruir_ruta(A, T, D)
transbordos = contar_transbordos(ruta)

print("El tiempo mínimo es:", tiempo_minimo)
```

```
## El tiempo mínimo es: 9
```

```
print("La ruta óptima es:", ruta)
```

```
## La ruta óptima es: [(0, 0, 'A'), (0, 1, 'A'), (1, 1, 'A'), (1, 2, 'T'), (1, 3, 'A'), (2, 3, 'A')]
```

```
print("Número de transbordos:", transbordos)
```

```
## Número de transbordos: 2
```