Universidad del Valle de Guatemala

Facultad de Ingeniería

Análisis y Diseño de Algoritmos


# Examen Parcial 2

Gabriel Paz 221087

06 de abril del 2025, Guatemala de la Asunción

**Problema 1 (Contar combinaciones en teclado Nokia)**

Dado un número positivo n y un teclado numérico de un antiguo teléfono Nokia, en el que sólo se pueden usar las teclas 0 al 9 (pero no * ni #), se quiere contar cuántas secuencias de longitud n se pueden formar bajo la regla:

- Se puede iniciar con cualquier dígito (0–9).
- Para pasar de un dígito a otro, solo se pueden presionar las teclas adyacentes a ese dígito en la cuadrícula.

El teclado está dispuesto en una matriz de 4×3 (similar a):

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
| * | 0 | # |

1. **Explique porque este problema exhibe subestructura optima**
*Pista:* "Subestructura óptima" significa que la solución completa se puede construir a partir de soluciones a subproblemas. En este caso, **contar** combinaciones de longitud nnn se puede descomponer en contar combinaciones de longitud n−1n-1n−1 que terminen en diferentes dígitos adyacentes.

2. **Explique una idea/solución que exhiba subproblemas traslapados e indique cómo los mismos subproblemas se computan repetidamente.**
*Pista:* al contar combinaciones de longitud n que terminan en, por ejemplo, el dígito 1, necesitarás contar cuántas combinaciones de longitud n−1n-1n−1 terminan en cada uno de los dígitos que llegan a 1. Esos recuentos se reutilizan múltiples veces.

3. **Idea de solución con Programación Dinámica (memoización top-down).**

- Define una función recursiva, por ejemplo count_sequences(d, length), que dé el número de formas de formar una secuencia de longitud length que termine en el dígito d.
- Si length == 1, el número de formas es 1 (pues la secuencia de un solo dígito que termina en d es simplemente [d]).
- En caso contrario, "sumar" las maneras de llegar al dígito d desde todos sus adyacentes, considerando secuencias de longitud length - 1.

- Utiliza un diccionario o arreglo de memoización para no recalcular las mismas combinaciones.

4. **Encuentre el tiempo de complejidad para este algoritmo. Recuerde, deje su procedimiento.**

El programa define un estado a partir de un dígito d (0 al 9) y una longitud m (de 1 hasta n). Cada estado (d,m) se calcula una sola vez gracias a la técnica de memoización. Además, la cantidad de dígitos se considera una constante (10 dígitos), y cada dígito tiene un número fijo de vecinos (entre 2 y 5).

Número de estados: Hay 10 × n posibles estados en total.

Costo de cada estado: Al calcular un estado nuevo, se hace una suma de resultados de sus vecinos. Dado que cada dígito tiene solo unos pocos vecinos, esta parte toma un tiempo constante.

Conclusión: El algoritmo recorre 10×n estados, con O(1) trabajo en cada uno. Por lo tanto, la complejidad total es proporcional a n, es decir, se expresa como O(n).

5. **Usando su programa, encuentre las combinaciones totales posibles para n = 10.**

Usando el programa con la definición de adyacencias que excluye la posibilidad de “quedarse en el mismo dígito” (es decir, el teclado Nokia estándar donde cada dígito solo se conecta a sus vecinos físicos)

Es decir, existen **1,806,282** combinaciones posibles cuando la longitud de la secuencia es 10. Este valor se obtiene ejecutando el programa en Python y revisando la salida en pantalla.

***Programa .py para la resolución del problema 1***

```python
def construir_adyacencias():

    return {
        '0': ['0','8'],
        '1': ['1','2','4'],
        '2': ['1','2','3','5'],
        '3': ['2','3','6'],
        '4': ['1','4','5','7'],
        '5': ['2','4','5','6','8'],
        '6': ['3','5','6','9'],
        '7': ['4','7','8'],
        '8': ['5','7','8','9','0'],
        '9': ['6','8','9']
    }

def contar_secuencias(digito, largo, memo, adyacencias):

    if largo == 1:
        return 1

    if (digito, largo) in memo:
        return memo[(digito, largo)]

    total = 0
    for vecino in adyacencias[digito]:
        total += contar_secuencias(vecino, largo - 1, memo, adyacencias)

    memo[(digito, largo)] = total
    return total

def contar_todas_secuencias(n):

    adyacencias = construir_adyacencias()
    memo = {}
    total_final = 0

    # Se suma para cada dígito inicial (0..9)
    for d in ['0','1','2','3','4','5','6','7','8','9']:
        total_final += contar_secuencias(d, n, memo, adyacencias)

    return total_final

resultado_n10 = contar_todas_secuencias(10)
print("La cantidad de combinaciones posibles para n=10 es:", resultado_n10)
```

***Resultado en consola***

**Problema 2**

Para generar las matrices left, right, top y bottom, se recorre la matriz en cuatro direcciones diferentes. Primero, se avanza de izquierda a derecha para construir left, donde cada celda obtiene el número de 1's consecutivos a su izquierda (incluyendo la propia posición). De forma análoga, se recorre la matriz en sentido contrario (de derecha a izquierda) para right, de arriba hacia abajo para top y de abajo hacia arriba para bottom. El criterio es simple: si la celda actual contiene un 1, se añade 1 al valor calculado en la posición anterior correspondiente (por ejemplo, la celda izquierda en el caso de left); si encuentra un 0, se asigna 0, pues no puede haber continuidad de 1's en esa dirección.

Una vez se han llenado las cuatro matrices, se ubica la cruz más grande tomando, en cada posición (i, j), la mínima de las cuatro direcciones (left[i][j], right[i][j], top[i][j], bottom[i][j]), a la que se llamará r. El tamaño de la cruz centrada en (i, j) se determina con la fórmula 1 + 4 * (r - 1), y el valor máximo que se obtenga en toda la matriz se reporta como resultado final.

En cuanto a la complejidad, se sabe que cada una de las cuatro matrices se calcula en un recorrido de orden O(M×N), ya que cada celda se procesa en tiempo constante. El proceso de encontrar la cruz más grande también se realiza en O(M×N). Por tanto, la complejidad total se mantiene en O(M×N). Si la matriz es cuadrada, se simplifica a $O(n^2)$.

Al probar este método con la matriz del enunciado que se espera produzca 17, se constata que el resultado es efectivamente 17 para el tamaño de la cruz más grande, el análisis de complejidad en notación Big-O se basa exclusivamente en el razonamiento descrito.

**Codigo .py**

```python
import time

def largest_cross_size(grid):

    if not grid or not grid[0]:
        return 0

    rows = len(grid)
    cols = len(grid[0])


    # Se inicializan en 0
    left   = [[0]*cols for _ in range(rows)]
    right  = [[0]*cols for _ in range(rows)]
    top    = [[0]*cols for _ in range(rows)]
    bottom = [[0]*cols for _ in range(rows)]

    # (A) left[i][j]
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                if j == 0:
                    left[i][j] = 1
                else:
                    left[i][j] = left[i][j-1] + 1
            else:
                left[i][j] = 0

    # (B) right[i][j]
    for i in range(rows):
        for j in range(cols - 1, -1, -1):
            if grid[i][j] == 1:
                if j == cols - 1:
                    right[i][j] = 1
                else:
                    right[i][j] = right[i][j + 1] + 1
            else:
                right[i][j] = 0

    # (C) top[i][j]
    for j in range(cols):
        for i in range(rows):
            if grid[i][j] == 1:
                if i == 0:
                    top[i][j] = 1
```

```
    # (D) bottom[i][j]
    for j in range(cols):
        for i in range(rows - 1, -1, -1):
            if grid[i][j] == 1:
                if i == rows - 1:
                    bottom[i][j] = 1
                else:
                    bottom[i][j] = bottom[i + 1][j] + 1
            else:
                bottom[i][j] = 0

    max_cross_size = 0

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
                r = min(left[i][j], right[i][j], top[i][j], bottom[i][j])
                current_size = 1 + 4*(r - 1)  # tamaño de la cruz con centro en (i,j)
                if current_size > max_cross_size:
                    max_cross_size = current_size

    return max_cross_size


if __name__ == "__main__":

    grid_ejemplo = [
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 0, 1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
        [1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
        [1, 0, 0, 0, 1, 0, 0, 1, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 0, 1, 1],
        [1, 1, 0, 0, 1, 0, 1, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 0, 0]
    ]

    start_time = time.time()
    result = largest_cross_size(grid_ejemplo)
    end_time = time.time()

    print("Tamaño de la cruz más grande en la matriz de ejemplo:", result)
```

**Resultado en consola:**

```
PS C:\UVG\ALGORITMOS\Parcial 2> & "C:/Users/DELL I7/AppData/Local/Programs/Python/Python311/python.exe" "c:/UVG/ALGORITMOS/Parcial 2/Ejercicio2.py"
Tamaño de la cruz más grande en la matriz de ejemplo: 17
Tiempo real transcurrido (para este input): 0.00000 segundos
PS C:\UVG\ALGORITMOS\Parcial 2>
```

**Problema 3: Problema de Knapsack**

La afirmación es falsa. En el problema de knapsack de tipo 0/1, no siempre se logra la solución óptima simplemente eligiendo el objeto que tenga la mayor razón valor–costo ($v_i/c_i$ ). Esta estrategia funciona para el knapsack fraccionario, donde se puede tomar una parte de cada objeto y es válido priorizar los que tienen mejor relación valor/peso, pero en el knapsack 0/1 los objetos se toman de forma completa o no se toman. Por lo tanto, pueden existir combinaciones más ventajosas (en términos de valor total) que no incluyan ese objeto con la mayor relación ($v_i/c_i$)..

**Problema 4: Caminos con penalización si superan 10 aristas**

El grafo tiene un peso “normal” para cada arista, pero si un camino utiliza más de 10 aristas, el costo total se duplica. Se quiere hallar la forma de aplicar Floyd–Warshall (o cualquier método de caminos más cortos) para resolverlo.

Idea de la reducción

- Floyd–Warshall, en su forma más simple, no se fija en cuántas aristas se usan, sino que solo combina rutas según sea más barato pasar por cada vértice intermedio.
- Para “contar” la cantidad de aristas, se puede crear un grafo expandido en el cual cada vértice se “replica” en distintos niveles (del nivel 0 al nivel 10, y un nivel adicional para “más de 10”).
- Cuando se está en el nivel k (donde k≤10) y se toma una arista de peso w, se pasa al nivel k+1 con costo w. Si se pasa del nivel 10 al 11, se aplica la duplicación del costo (en la práctica, se modela poniendo un peso doble a las aristas usadas a partir de ese momento).
- Luego, aplicar Floyd–Warshall o un método para caminos más cortos en esa versión “expandida” del grafo encuentra la ruta con la penalización incorporada.

En otras palabras, se “codifica” la penalización en un grafo artificial, para que un algoritmo de caminos usual pueda manejarla.

**Problema 5: Demostrar que (S, I) es un matroide**

Se toma una matriz T de tamaño m×n, donde:

- S es el conjunto de las columnas de T.

- Un subconjunto $A \subseteq S$ está en I si y solo si esas columnas en A son linealmente independientes.

Para que (S, I) sea un **matroide**, deben cumplirse dos propiedades:

1. **Hereditaria**: si A está en I, cualquier subconjunto de A también está en I. Eso es cierto porque la independencia lineal no se "rompe" al quitar columnas.

2. **Intercambio**: si hay dos conjuntos independientes A y B y $|A|<|B|$, entonces existe al menos una columna en $B \setminus A$ que se puede añadir a A sin perder la independencia. Esto es una propiedad de los conjuntos de vectores: siempre puede incluirse alguno de los que no estén en A, hasta que iguale la dimensión (o cardinalidad) de B.

Así, se demuestra que cumple la definición de matroide. Estos matroides se conocen con frecuencia como "matroides lineales".

**Problema 6: Scheduling Problem**

1. **Localizar el deadline máximo** en la lista (en este ejemplo, dmax = 6), de modo que se tiene un máximo de 6 unidades de tiempo para agendar.

2. **Ordenar** las tareas por peso (valor) de mayor a menor.

    - T7 (70), T6 (60), T5 (50), T4 (40), T3 (30), T2 (20), T1 (10).

3. **Crear un arreglo de "slots"** de longitud dmax. Por ejemplo, slots=[__,__,__,__,__,__] para 6 ranuras (tiempos 1 a 6).

4. **Para cada tarea** en el orden de mayor peso, colocarla en la ranura más tardía posible donde aún pueda cumplir su deadline.

    - Si la tarea tiene di=4 , se intenta primero colocar en el slot 4. Si está ocupado, se intenta el slot 3, luego 2, etc.

    - Si no hay espacio en ninguno de los slots anteriores a di , la tarea quedará "fuera" (tardía).

5. **El schedule resultante** se lee del slot 1 al slot 6, y las tareas que no se ubican en ningún slot son las que generan la penalización (sumando sus pesos).

Con este "método de slots" se ve de forma más clara y sintética dónde se coloca cada tarea y cómo se descartan las que no tienen cabida. El resultado final y la penalización son idénticos a los obtenidos en el proceso detallado:

- Tiempo 1: Tarea 5

- Tiempo 2: Tarea 3
- Tiempo 3: Tarea 4
- Tiempo 4: Tarea 6
- Tiempo 5: (vacío)
- Tiempo 6: Tarea 7

Tareas 2 y 1 quedan fuera (penalización total = 30).

Este es el enfoque típico de “programación de tareas con deadlines y ganancias/pesos” cuando cada tarea ocupa una unidad de tiempo; se describe como un algoritmo greedy que ordena por valor descendente y rellena huecos de derecha a izquierda. Este modo de proceder es más conciso que repasar verbalmente todas las opciones de hueco en cada paso.