Análisis y Diseño de Algoritmos

Ejercicios Tema 4

Francisco Javier Mercader Martínez

1) Diseñar un algoritmo para calcular el mayor y el segundo mayor elemento de un array de enteros utilizando la técnica divide y venceras.
Calcular el número de comparaciones realizadas en el peor y el mejor caso suponenido `n` potencia de 2.
¿Sería el orden obtenido extrapolable a un `n` que no sea potencia de 2?

1) Algoritmo de Divide y Vencerás

La idea principal es dividir el problema en subproblemas más pequeños, resolverlos y luego cominar las soluciones.

Esquema General:

1) División:
   - Divide el arreglo en dos mitades iguales.
2) Conquista:
   - Encuentra el mayor y el segundo mayor en cada mitad de forma recursiva.
   - Combinación:
     - Combina las soluciones de las dos mitades:
       - El mayor de las dos mitades es el mayor global.
       - El segundo mayor será el mayor de:
         - El segundo mayor de la mitad que contiene el mayor global.
         - El mayor de la otra mitad

Implementación

```python
def encontrar_mayores(arr):
    """
    Encuentra el mayor y el segundo mayor elemento de un arreglo utilizando divide y
        vencerás.

    - arr: Lista de enteros.
    Retorna: (mayor, segundo_mayor, comparaciones)
    """
    def dividir_y_vencer(arr):
        # Caso base: Si hay solo dos elementos, compara directamente
        if len(arr) == 2:
            if arr[0] > arr[1]:
                return arr[0], arr[1], 1 # mayor, segundo mayor, comparaciones
            else:
                return arr[1], arr[0], 1
```

```
        # Divide el arreglo en dos mitades
        mid = len(arr) // 2
        izq_mayor, izq_segundo, izq_comparaciones = dividir_y_vencer(arr[:mid])
        der_mayor, der_segundo, der_comparaciones = dividir_y_vencer(arr[mid:])

        # Combina las soluciones
        comparaciones = izq_comparaciones + der_comparaciones

        if izq_mayor > der_mayor:
            mayor = izq_mayor
            segundo_mayor = max(izq_segundo, der_mayor)
        else:
            mayor = der_mayor
            segundo_mayor = max(der_segundo, izq_mayor)

        comparaciones += 2 # Comparaciones para determinar mayor y segundo mayor
        return mayor, segundo_mayor, comparaciones

    # Llamar a la función recursiva
    return dividir_y_vencer(arr)

# Ejemplo de uso
if __name__ == '__main__':
    arr = [10, 3, 5, 7, 9, 2, 6, 8]
    mayor, segundo_mayor, comparaciones = encontrar_mayores(arr)
    print(f"Mayor elemento: {mayor}")
    print(f"Segundo mayor elemento: {segundo_mayor}")
    print(f"Total de comparaciones: {comparaciones}")
```

2) Análisis del número de comparaciones

Peor caso y mejor caso

El número de comparaciones es el mismo en el peor y el mejor caso, ya que cada división implica el mismo número de subproblemas y combinaciones.

1) Caso base:
   - Para $n = 2$: Se realiza una única comparación.
2) Generalización para $n = 2^k$:
   - Hay $n - 1$ comparaciones para encontrar el mayor (esto se debe a que cada elemento pierde contra el mayor exactamente una vez).
   - Hay $\log_2(n) - 1$ comparaciones adicionales para encontrar el segundo mayor, porque el segundo mayor es el que pierde contra el mayor en el "torneo".

     Total de comparaciones:

$$T(n) = (n-1)(log_2(n) - 1) = n \log_2(n) - 2.$$

3) Orden para n no potencia de 2

Cuando n no es potencia de 2, el algoritmo sigue siendo aplicable:

1) Se rellena el arreglo con elementos adiciones $(-\infty)$ para completar una potencia de 2.

2) Las comparaciones adicionales no afectan significativamente el orden del algoritmo, ya que $T(n) = O(n + \log_2(n))$ sigue siendo válido.

2) Sea `T[1..n]` un array ordenado pormado por eventos diferentes, algunos de los cuales pueden ser negativos. Dar un algoritmo DyV que pueda hallar bf`i` tal que $1 \leq i \leq n$ y `T[i]=1`, siempre que este índice exista. El algoritmo debe tener un $O(logn)$.

El problema es encontrar un índice `i` tal que `T[i]=1` en un arreglo ordenado `T` tamaño n, donde los elementos son únicos y pueden incluir números negativos. Este arreglo está ordeado, lo que permite usar una búsqeda binaria para alcanzar una complejidad $O(\log n)$.

Algoritmo

Usaremos el enfoque de **Divide y Vencerás** a través de una implementaación de búsqueda binaria para encontrar el índice donde `T[i]=1`.

Esquema

1) División:
   - Divide el arreglo en dos mitades en cada paso, seleccionando el elemento medio.
2) Conquista:
   - Si `T[mid]=1`, se ha encontrado el índice esperado.
   - Si `T[mid]>1`, busca en la mitad izquierda del arreglo.
   - Si `T[mid]<1`, busca en la mitad derecha del arreglo.
3) Combinación:
   - En este caso, no es necesario combinar resultados ya que solo buscamos uun único índice.

Implementación

```python
def encontrar_indice_uno(T, inicio, fin):
    """
    Encuentra un índice i tal que T[i] = 1 en un arreglo ordenado.
    - T: Arreglo ordenado de enteros.
    - inicio: índice inicial de la región a buscar.
    - fin: índice final de la región a buscar.
    Retorna el índice i si existe, o -1 si no encuentra.
    """
    if inicio > fin:
        return -1 # No se encuentra el elemento

    mid = (inicio + fin) // 2 # Índice medio

    if T[mid] == 1:
        return mid # Elemento encontrado
    elif T[mid] > 1:
        return encontrar_indice_uno(T, inicio, mid - 1) # Buscar en la mitad izquierda
    else:
        return encontrar_indice_uno(T, mid + 1, fin) # Buscar en la mitad derecha

def buscar_uno(T):
    """
    Función principal para encontrar el índice de T[i] = 1 en un arreglo ordenado.
    - T: Arreglo ordenado de enteros.
    Retorna el índice si existe, o -1 si no se encuentra.
```

```python
    """
    return encontrar_indice_uno(T, 0, len(T) - 1)

# Ejemplo de uso
if __name__ == '__main__':
    T = [-10, -5, -2, 0, 1, 3, 5, 8] # Arreglo de ejemplo
    indice = buscar_uno(T)
    if indice != -1:
        print(f"El índice donde T[i] = 1 es: {indice}")
    else:
        print("No se encontró ningún índice donde T[i] = 1.")
```

Explicación del código

1) Caso base:
   - Si el inventario de búsqeuda es inválido (`inicio > fin`), significa que 1 no está en el arreglo, y retornamos -1.
2) Cállculo del índice medio:
   - Dividimos el arreglo por la mitad (`mid = (inicio + fin)//2`).
3) Condiciones:
   - Si `T[mid]=1`, hemos encontrado el índice deseado y lo retornamos.
   - Si `T[mid]>1`, buscamos en la mitad izquierda (`inicio` a `mid-1`).
   - Si `T[mid]<1`, vuscamos en la mitad derecha (`mid+1` a `fin`).
4) Llamada inicial:
   - La función `buscar_uno` inicia la búsqueda con los límites completos del arreglo.

Análisis de complejidad

El algoritmo realiza $\log(n)$ comparaciones en el peor caso, ya que el tamaño del arreglo se reduce a la mitad en cada llamada recursiva.

Ejemplo de salida

Dado el arreglo `T=[-10, -5, -2, 0, 1, 3, 5, 8]`:

`## El índice donde T[i] = 1 es: 4`

Si `T` con contiene el valor 1, por ejemplo, `T=[-10, -5, -2, 0, 2, 3, 5, 8]`:

`## No se encontró ningún índice donde T[i] = 1.`

3) Resuelve por DyV este problema. Dado un array de `N` números enteros, buscar la cadena de `n` celdas consecutivas ($n \leq N$) cuya suma sea máxima.
¿Sería conveniente aplicar aquí DyV? ¿Cómo sería una resolución directa? Calcula el orden de ambos algoritmos.

1) Descomposición del problema con Divide y Vencerás

   1) División:
      - Divide el arreglo de dos mitades: izquierda y derecha.
   2) Conquista:
      - Encuentra la subcadena de suma máxima en la mitad izquierda.
      - Encuentra la subcadena de suma máxima en la mitad derecha.

- Encuentra la subcadena de suma máxima que cruza las dos mitades.

3) Combinación:

   - Compara las tres sumas obtenidas (izquierda, derecha y cruzada) y selecciona la máxima.

2) Implementación

```python
def suma_maxima_cruzada(arr, inicio, medio, fin, n):
    """
    Calcula la suma máxima de una subcadena de longitud n que cruza el punto medio del
        arreglo.
    - arr: Arreglo de enteros.
    - inicio: índice de inicio del intervalo.
    - medio: índice medio del intervalo.
    - fin: índice final del intervalo.
    - n: Longitud de la subcadena buscada.
    Retorna la suma máxima encontrada.
    """
    max_izquierda = float('-inf')
    suma_actual = 0

    # Considerar las subcadenas del lado izquierdo que terminan en medio
    for i in range(medio, max(inicio, medio - n + 1) - 1, -1):
        suma_actual += arr[i]
        max_izquierda = max(max_izquierda, suma_actual) if medio - i + 1 <= n else
            max_izquierda

    max_derecha = float('-inf')
    suma_actual = 0

    # Considerar las subcadenas del lado derecho que comienzan en medio + 1
    for i in range(medio + 1, min(fin + 1, medio + n + 1)):
        suma_actual += arr[i]
        max_derecha = max(max_derecha, suma_actual) if i - medio <= n else max_derecha

    return max_izquierda + max_derecha


def suma_maxima_dyv(arr, inicio, fin, n):
    """
    Divide y vence para encontrar la suma máxima de una subcadena de longitud n.
    - arr: Arreglo de enteros.
    - inicio: índice de inicio del intervalo.
    - fin: índice final del intervalo.
    - n: Longitud de la subcadena buscada.
    Retorna la suma máxima encontrada.
    """
    # Caso base: Si el intervalo es exactamente de longitud n
    if fin - inicio + 1 == n:
        return sum(arr[inicio:inicio + n])

    if fin - inicio + 1 < n:
        return float('-inf')

    # Dividir el arreglo en dos mitades
```

```python
    medio = (inicio + fin) // 2

    # Resolver las subcadenas máximas en las dos mitades
    suma_izquierda = suma_maxima_dyv(arr, inicio, medio, n)
    suma_derecha = suma_maxima_dyv(arr, medio + 1, fin, n)

    # Resolver la subcadena máxima que cruza el punto medio
    suma_cruzada = suma_maxima_cruzada(arr, inicio, medio, fin, n)

    # Retornar la máxima de las tres
    return max(suma_izquierda, suma_derecha, suma_cruzada)


def buscar_suma_maxima(arr, n):
    """
    Función principal para encontrar la subcadena de longitud n con suma máxima.
    - arr: Arreglo de enteros.
    - n: Longitud de la subcadena buscada.
    Retorna la suma máxima encontrada.
    """
    if len(arr) < n:
        raise ValueError("La longitud de la subcadena no puede ser mayor que el tamaño
            del arreglo.")
    return suma_maxima_dyv(arr, 0, len(arr) - 1, n)

# Ejemplo de uso
if __name__ == "__main__":
    arr = [2, -1, 3, 5, -2, 8, -1, 4]
    n = 3
    resultado = buscar_suma_maxima(arr, n)
    print(f"La suma máxima de una subcadena de longitud {n} es: {resultado}")
```

3) Explicación del algoritmo

   1) `suma_maxima_cruzada`:
      - Calcula la suma máxima de una subcadena de longitud n que cruza el punto medio del arreglo.
      - Se consideran:
        - Las subcadenas a la izquierda que terminan en el punto medio.
        - Las subcadenas a la derecha que comienzan justo después del punto medio.
      - La suma máxima cruzada es la combinación de ambas partes.
   2) `suma_maxima_dyv`:
      - Implementa la estrategia de divide y vencerás:
        - Resuelve recursivamente el problema en las mitades izquierda y derecha.
        - Calcula la suma máxima cruzada entre las ods mitades.
        - Retorna la mayor de estas tres sumas.
   3) `buscar_suma_maxima`:
      - Valida que n sea menor o igual al tamaño del arreglo.
      - Llama a la función recursiva para encontrar la suma máxima.

4) Complejidad

Tiempo

- El algoritmo divide el arreglo en dos mitades en cada paso, lo que prodce una recursión de $logN$ niveles.
- En cada nivel, calcular la suma cruzada tiene un costo lineal $O(n)$.
- **Complejidad total:** $O(N \cdot \log N)$.

5) Ejemplo de salida

Con el arreglo `arr = [2, -1, 3, 5, -2, 8, -2, 4]` y `n = 3`:

```
## La suma máxima de una subcadena de longitud 3 es: 14
```

6) Análisis

Ventajas de Divide y Vencerás:

- Divide el problema en partes más pequeñas y combina soluciones.
- Útil cuando el problema puede descomponerse de manera eficiente.

Desventajas en este caso:

1) El problema no se divide de manera uniforme porque la longitud n es fija. La combinación de resultados de subarreglos de tamaño n no es directa.
2) La combinación requere considerar subarreglos que cruzan las divisiones, lo que introduce sobrecarga adicional.

Por lo tanto, **no es ideal usar Divide y Vencerás para este problema.** Una resolución directa (lienal) es más eficiente.