Tarea 4

Juárez Torres Carlos Alberto

2023-03-20

1 Caja Negra: Supongamos que tenemos acceso a un algoritmo, denominado Caja Negra. del cual sólo conocemos sus resultados, contesta: sí o no. Si se le da una secuencia de n números y un entero k, el algoritmo responde si o no, según exista un subconjunto de esos números cuya suma sea exactamente k. Mostrar como usar este algoritmo $O(n)$ veces en un programa que encuentre el subconjunto en cuestión, si es que existe, donde n es el tamaño de la secuencia.

- Inicializar un arreglo A de tamaño $n+1$ con valores booleanos, donde A_i indica si existe un subconjunto de los primeros i números de la secuencia que suma exactamente k.
- Establecer A_0 como verdadero, ya que el subconjunto vacío siempre suma 0.
- Para cada número x_i en la secuencia, realizar lo siguiente:
  - Recorrer el arreglo A de derecha a izquierda.
  - Si A_j es verdadero, entonces establecer A_{j+x_i} como verdadero.
- Si A_n es verdadero, entonces existe un subconjunto de la secuencia que suma exactamente k. Para encontrar dicho subconjunto, se puede reconstruir iterativamente el subconjunto a partir de los valores del arreglo A. Comenzando en $i = n$, se comprueba si A_{i-x_j} es verdadero para cada x_j en la secuencia. Si es verdadero, se añade x_j al subconjunto y se establece i como $i - x_j$. Se continúa hasta que i sea igual a 0.

El tiempo de ejecución del algoritmo es $O(n^2)$, ya que el recorrido del arreglo A para cada número de la secuencia implica un tiempo total de $O(n^2)$. Sin embargo, si se utiliza la Caja Negra para determinar si un subconjunto de los primeros i números suma exactamente k, entonces el tiempo de ejecución se reduce a $O(n)$, lo que permite resolver el problema de manera eficiente para entradas grandes.

2 Los dos mayores elementos de un Conjunto Dada una secuencia $S = [x_1, x_2, ..., x_n]$ encontrar a los dos mayores elementos de S usando la menor cantidad posible de comparaciones.

2.1 Diseñar, usando Inducción matematica, un algoritmo que resuelva el problema

Caso Base: Si la secuencia S tiene longitud 1, el algoritmo devuelve el único elemento como el mayor y el segundo mayor, y no se realiza ninguna comparación. Por lo tanto, la afirmación es verdadera para secuenciasde longitud 1.
Hipótesis de Inducción Supongamos que el algoritmo es correcto para todas las secuencias de longitud k, donde k es un entero positivo menor que n.
Paso Inductivo
Consideremos una secuencia S de longitud n. Dividimos S en dos mitades, S_1 y S_2, de longitud n_1 y n_2, respectivamente, donde n_1 y n_2 son enteros positivos que suman n. Por hipótesis de inducción, el algoritmo encuentra los dos mayores elementos de S_1 y S_2 utilizando la menor cantidad posible de comparaciones.

Encontrar los dos mayores elementos de S utilizando el algoritmo implica encontrar los dos mayores elementos de S_1 y S_2 y compararlos para obtener los dos mayores elementos de S. El número de comparaciones necesarias para encontrar los dos mayores elementos de S es el número de comparaciones necesarias para encontrar los dos mayores elementos de S_1 y S_2, más una comparación adicional para comparar los dos mayores elementos encontrados en S_1 y S_2.

Por lo tanto, el número de comparaciones necesarias para encontrar los dos mayores elementos de S es la menor cantidad posible de comparaciones, ya que se utiliza la menor cantidad posible de comparaciones para encontrar los dos mayores elementos de S_1 y S_2.

El algoritmo se puede describir utilizando el siguiente pseudocódigo:

```
function Dosgrandes(S):
    if length(S) == 1:
        return S[1], None
    elif length(S) == 2:
        if S[1] >= S[2]:
            return S[1], S[2]
        else:
            return S[2], S[1]
    else:
        n = length(S)
        n1 = floor(n/2)
        n2 = n - n1
        S1 = S[1:n1]
        S2 = S[n1+1:n]
        max1, max2 = Dosgrandes(S1)
```

```
max3, max4 = Dosgrandes(S2)
if max1 >= max3:
    if max1 >= max4:
        if max2 >= max4:
            return max1, max2
        else:
            return max1, max4
    else:
        return max3, max4
else:
    if max3 >= max2:
        if max4 >= max2:
            return max3, max4
        else:
            return max3, max2
    else:
        return max1, max2
```

2.2 Determinar el número de comparaciones que realiza el algoritmo propuesto

Depende del tamaño de la secuencia de entrada. Sea n la longitud de la secuencia S.

- Si $n = 1$, el algoritmo no realiza ninguna comparación.
- Si $n = 2$, el algoritmo realiza una comparación para determinar cuál de los dos elementos es mayor.
- Si $n > 2$, el algoritmo divide la secuencia en dos mitades de tamaño aproximadamente igual y llama recursivamente al algoritmo para cada mitad. Después, el algoritmo realiza tres comparaciones para determinar los dos mayores elementos de la secuencia original a partir de los dos mayores elementos de cada mitad. Por lo tanto, el número total de comparaciones es:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 3$$

  Donde $T(n)$ es el número total de comparaciones realizadas por el algoritmo para una secuencia de longitud n. Esta ecuación de recurrencia puede resolverse mediante el método de resolución de recurrencias. Podemos asumir que n es una potencia de 2 (es decir, $n = 2^k$ para algún entero k). En este caso, la ecuación de recurrencia se puede escribir como:

$$T(n) = 2T\left(\frac{n}{2}\right) + 3$$

  Aplicando la regla de sustitución repetida, podemos encontrar una expresión para $T(n)$ en términos de $T(1)$

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 3 \\
&= 2\left(2T\left(\frac{n}{2^2}\right) + 3\right) + 3 \\
&= 2^2 T\left(\frac{n}{2^2}\right) + 2^1 \cdot 3 + 2^0 \cdot 3 \\
&= 2^2 T\left(\frac{n}{2^2}\right) + 2^1 \cdot 3 + 2^0 \cdot 3 \\
&\vdots \\
&= k \cdot 3 + 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) \\
&= k \cdot 3 + 2^{k-1} T(2) \\
&= k \cdot 3 + 2^{k-1} \cdot 1 \\
&= 3n - 2
\end{aligned}
$$

Por lo tanto, el número de comparaciones que realiza el algoritmo para una secuencia de longitud n es $3n - 2$. En otras palabras, el algoritmo es $O(n)$ lineal en el peor de los casos.

3 Considerar el siguiente problema

Cambio de Base Dado un número en base 6 convertirlo a binario, la entrada es un arreglo de digitos en base 6 y la salida es un arreglo de bits

3.1 Diseñar usando Inducción Matematica un algoritmo eficiente que solucione el problema y que use el menor número de comparaciones

Caso Base: Podemos ver que para el caso base de $n = 0$, la representación binaria es 0, lo cual es cierto. Para el caso base de $n = 1$, la representación binaria es 1, lo cual también es cierto.
Hipotesis: Supongamos que la afirmación es cierta para todos los números en base 6 menores que n.
Paso Inductivo: Si n es impar, entonces $n = 2k + 1$ para algún k. Entonces, la representación binaria de n es la misma que la de k, seguida de un 1. Pero k es menor que n y, por lo tanto, la afirmación es cierta para k. Por lo tanto, la representación binaria de k se puede calcular utilizando el proceso recursivo, y la representación binaria de n es simplemente la de k seguida de un 1. Por lo tanto, la afirmación es cierta para n.

Finalmente obtenemos el siguiente algoritmo

```
convertir_a_binario(numero_en_base_6):
    if numero_en_base_6 == 0:
        return [0]
    elif numero_en_base_6 == 1:
        return [1]
    else:
      cociente = numero_en_base_6 // 2
      resto = numero_en_base_6 % 2
      representacion_binaria_cociente = convertir_a_binario(cociente)
      if resto == 0:
        representacion_binaria_cociente.append(0)
      else:
        representacion_binaria_cociente.append(1)

      return representacion_binaria_cociente
```

3.2 Determinar la complejidad del algoritmo obtenido

La complejidad temporal del algoritmo que diseñamos para convertir un número en base 6 a binario utilizando el proceso recursivo es de $O(\log n)$, donde n es el número en base 6.

Esto se debe a que en cada llamada recursiva, el número en base 6 se divide por 2 (es decir, se divide por 6 y se redondea hacia abajo), lo que reduce su tamaño a la mitad. Como resultado, el número de llamadas recursivas necesarias para convertir el número completo a binario es $\log_2(n)$, lo que da lugar a la complejidad temporal de $O(\log n)$.

4 Considerar el siguiente problema

Partición: Dada una lista con de n enteros positivos y distintos particionar (dividir) la lista en dos sublistas L_1 y L_2, cada una de tamaño $n/2$ tal que la diferencia entre las sumas de los enteros en las dos listas sea mínima; suponer que n es mútiplo de dos.

4.1 Diseñar usando Inducción Matematica un algoritmo eficiente que solucione el problema y que use el menor número de comparaciones

Caso Baso: Si la lista tiene tamaño 1, simplemente devolvemos la lista como una de las sublistas y una lista vacía como la otra.
Hipotesis de Inducción Supongamos que se tiene una lista de longitud n, y que se ha encontrado una manera de dividirla en dos sublistas L_1 y L_2 de longitud $n/2$ de tal manera que la diferencia entre las sumas de los elementos en las dos listas sea mínima.
Paso Inductivo Consideremos ahora una lista de longitud $n+2$. Podemos separar los dos elementos adicionales y dividir la lista restante en dos sublistas de longitud $n/2$ utilizando nuestra hipótesis de inducción. Luego, podemos comparar las sumas de las dos sublistas de longitud $n/2$ con la suma de los dos elementos adicionales, y seleccionar la partición que minimice la diferencia entre las sumas de las dos sublistas.

Por lo tanto, podemos utilizar este enfoque recursivo para dividir la lista original en dos sublistas de longitud $n/2$ con la diferencia mínima entre las sumas de los elementos en las dos sublistas. La complejidad de este algoritmo es $O(n \log n)$, ya que se realiza una división recursiva de la lista en cada paso de la inducción y cada división toma $O(\log n)$ tiempo. Además, se realiza una comparación adicional de cada elemento en la lista, lo que resulta en un total de $O(n)$ comparaciones.

De modo que quedamos con el siguiente algoritmo:

```
funcion particion(lista):
    if longitud(lista) == 2:
        return [lista[0]], [lista[1]]  # Caso base
    else:
        # Dividir la lista en dos sublistas de longitud n/2
        mitad = longitud(lista) // 2
        l1, l2 = particion(lista[:mitad]), particion(lista[mitad:])

        # Calcular la suma de los elementos en cada sublista
        suma_l1, suma_l2 = sum(l1), sum(l2)


        if suma_l1 > suma_l2:
            # Si la suma de l1 es mayor, mover un elemento de l1 a l2
            elemento = l1.pop()
            l2.append(elemento)
        else suma_l1 < suma_l2:
            # Si la suma de l2 es mayor, mover un elemento de l2 a l1
            elemento = l2.pop(0)
            l1.append(elemento)
```

```
    # Devolver las dos sublistas
    return l1, l2
```

4.2 Determinar la complejidad del algoritmo obtenido

El algoritmo consta de dos partes principales: la división recursiva de la lista en sublistas de tamaño $n/2$ y el cálculo de la diferencia mínima entre las sumas de los elementos de las sublistas.

La división recursiva de la lista en sublistas de tamaño $n/2$ se realiza a través de la recursión, y la profundidad de la recursión es $\log_2 n$. En cada nivel de recursión se realiza una operación de partición de la lista original en dos sublistas, que tiene una complejidad de $O(n)$, ya que debe copiar todos los elementos de la lista original. Por lo tanto, la complejidad total de la división recursiva es $O(n \log n)$.

El cálculo de la diferencia mínima entre las sumas de los elementos de las sublistas se realiza después de la división recursiva. Para cada par de sublistas, se calcula la diferencia de sus sumas y se compara con la diferencia actual mínima. Esto implica $n/2$ comparaciones, ya que hay $n/2$ pares de sublistas. Por lo tanto, la complejidad total de esta parte del algoritmo es $O(n)$.

Por lo tanto, la complejidad total del algoritmo de partición obtenido mediante la inducción matemática es $O(n \log n)$