UADE

Programación III – Diseño y Análisis de Algoritmos

Temas a desarrollar

1. Repaso Clase Anterior
2. Validación Resolución Problemas
3. Elemento Mayoritario
4. Función de Fibonacci
5. Introducción a Greedy
6. Problema del Cambio
7. Problema de la Mochila

Repaso Clase Anterior

Métodos de Ordenamiento

- Los métodos de ordenamiento denominados iterativos como ser “Selección” o “Inserción” tienen una complejidad temporal de Θ (n^2), siendo n la cantidad de elementos a ordenar.

- En los métodos de ordenamiento que siguen una estrategia por Divide y Conquista, como ser Merge Sort o Quick Sort, se pueden llegar a complejidades temporales de Θ (n log n), siendo **n** la cantidad de elementos a ordenar.

Métodos de Ordenamiento Divide y Conquista

- **Merge Sort**, consiste en dividir la secuencia de entrada en dos mitades, ordenar cada una de las mitades y luego mezclar las mitades ordenadas en una nueva secuencia ordenada. La complejidad temporal está dada por (siendo **n** la cantidad de elementos a ordenar):

$$\boxed{\Theta(n \log(n))}$$

- **Quick Sort**, El método consiste en seleccionar un elemento al azar denominado *Pivot*. A partir de ese *Pivot* se divide la secuencia en dos partes, a la izquierda del *Pivot* todos los elementos menores y a la derecha del *Pivot* todos los elementos mayores al mismo. La complejidad temporal está dada por (siendo **n** la cantidad de elementos a ordenar):

Caso Promedio $\boxed{\Theta(n \log(n))}$ Peor Caso $\boxed{\Theta(n^2)}$

Validación Resolución Problemas

Validación Resolución Problemas

Lea detenidamente cada problema, por cada uno de ellos deberá responder los siguientes puntos:

a) Estrategia de la solución aplicando Divide y Conquista e indicando cada uno de los elementos que se requieren para la resolución de esa técnica.

b) Pseudocódigo del Algoritmo de la resolución del problema, el cual debe seguir la estrategia definida en el punto a)

c) Cálculo de la Complejidad Temporal, justificando la misma en forma detallada.

3. Sea **A[1..n]**, **n** ≥ 1, un vector de enteros diferentes y ordenados crecientemente, tal que algunos de los valores pueden ser negativos. Diseñar un algoritmo que devuelva un índice natural **k**, **1 ≤ k ≤ n**, tal que **A[k] = k**, siempre que tal índice exista.

4. Dada una secuencia de números no ordenada, determinar cuál es el valor **k-esimo** menor de la misma (se entiende que el valor **k** se recibe como entrada del algoritmo). El algoritmo deberá tener una complejidad temporal en el **caso promedio** de **O(n)**, pudiendo tener mayor complejidad en el peor de los casos.

Definición de elemento mayoritario

Elemento mayoritario

- Dado un vector ***A*** de números enteros, calcular elemento mayoritario. Si se tiene un vector ***A*** de ***n*** enteros, un elemento ***x*** se denomina mayoritario de ***A*** si ***x*** aparece en el vector ***A*** más de ***n/2*** veces.

Considerar que no puede haber más de un elemento mayoritario, debido a que no puede haber dos elementos que puedan cumplir la condición mayor a n/2*.*

Elemento mayoritario – Algunas observaciones

- Una forma simple de resolverlo es verificar para cada elemento del vector, si aparece más de *n/2* veces. Este algoritmo tiene la forma de dos bucles anidados con lo que podríamos decir que esta estrategia tiene un costo de ***O(n²)*.**

- Un algoritmo máseficiente podría ser ordenar el vector ***O(n log(n)*)**. Si existe un elemento mayoritario, este se encuentra en la posición media del vector, entonces se toma el elemento de la posición del medio y se cuenta si aparece más de *n/2* veces, lo cual tiene costo ***O(n)***. Con lo que esta estrategia está marcada por el ordenamiento y la solución queda en ***O(n log(n))***.

- Sin embargo, existe una resolución por divide y conquista más eficiente en complejidad temporal.

Estrategia de solución

Elemento mayoritario – Estrategia divide y conquista

- Analizando si un elemento es mayoritario examinando de a pares siempre aparecen juntos, o en el caso de una secuencia impar, podría ser el elemento que se encuentre al final.

- Basado en lo anterior se puede plantear una estrategia de la siguiente manera:

Si un elemento ***x*** es mayoritario, dividimos el vector original en sub-vectores de tamaño 2 (salvo el ultimo que si ***n*** es impar quedara de tamaño 1), debe aparecer ***x*** repetido en alguno de los sub-vectores o debe ser el elemento que está en el vector de un solo elemento para el caso de **n** impar.

→

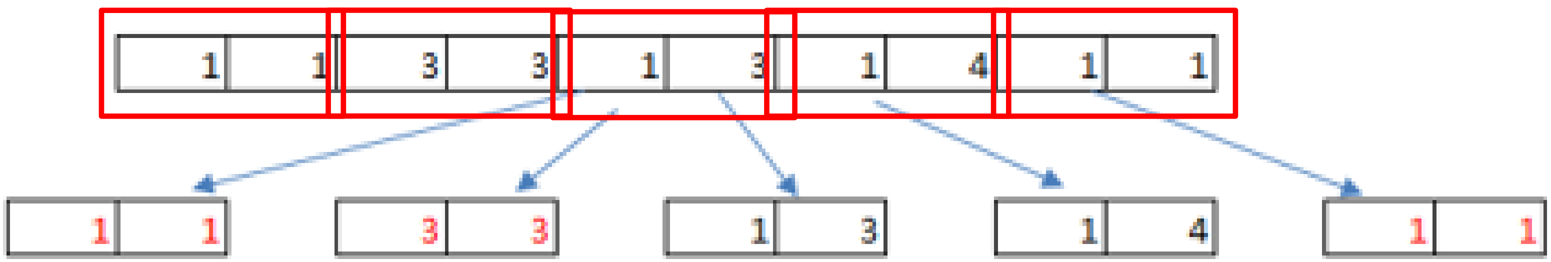
Si esto lo repetimos hasta obtener un único elemento, al mismo lo llamaremos candidato, luego nos resta verificar si ese candidato es elemento mayoritario o no.

→

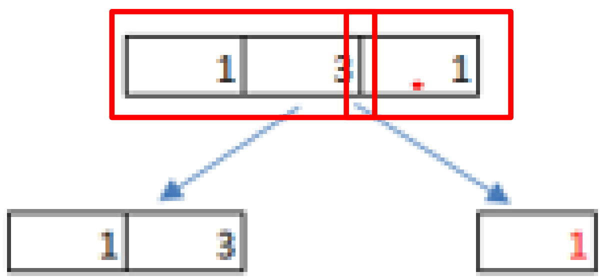
Si nos quedamos con todos los elementos que cumplen con esta condición, podremos reducir nuestro vector original a un vector que tiene a lo sumo *n/2* elementos.

Elemento mayoritario – Estrategia divide y conquista

- Consideramos el siguiente ejemplo para ejemplificar la estrategia mencionada anteriormente, entonces:



- Luego de la primera iteración nos quedamos con aquellos elementos que a pares fueron iguales, dando lugar al subproblema, volviendo a hacer el mismo análisis:



- En el análisis anterior vemos que de a pares no son iguales, y al ser un vector impar nos quedamos con el valor en la posición tres como posible candidato mayoritario. Luego comprobaremos si lo es o no recorriendo el vector original. En este caso sí es el 1 mayoritario.

UADE

Algoritmo

Elemento mayoritario – Algoritmo

```
ALGORITMO BUSCARCANDIDATO
Entrada: V: Vector<entero>, inicio:entero, fin:entero
Salida: c: entero
  si fin < inicio
    devolver No hay cadidato
  sino
    si inicio = fin
      devolver V[inicio]
    sino
      j ← inicio
      si esPar(fin − inicio + 1)
        i ← inicio + 1
        mientras i ≤ fin
          si V[i − 1] = V[i]
            V[j] ← V[i]
            j ← j + 1
          fin si
          i ← i + 2
        fin mientras
        devolver BuscarCandidato(V, inicio, j-1)
      sino
        i ← inicio + 1
        mientras i < fin
          si V[i − 1] = V[i]
            V[j] ← V[i]
            j ← j + 1
          fin si
          i ← i + 2
        fin mientras
        x ← BuscarCandidato(V, inicio, j-1))
        si tieneSolucion (x)
          devolver x
        sino
          devolver V[fin]
        fin si
      fin si
    fin si
  fin si
```

El algoritmo comienza en la parte izquierda y continúa a la derecha

Elemento mayoritario – Algoritmo

ALGORITMO ELEMENTOMAYORITARIO

Entrada: V: Vector<entero>
Salida: x: entero

```
  entero x ← BuscarCandidato(V, 0, longitud(S) − 1)
  si existe(x)
    suma ← 0
    para entero i = 0 hasta longitud(V) − 1
      si x = V[i]
        suma ← suma + 1
      fin si
    fin para
    si suma > longitud(V)/2
      devolver x
    sino
      devolver No existe elemento mayoritario
    fin si
  sino
  devolver No existe elemento mayoritario
fin si
```

En este caso la solución al problema del elemento mayoritario se compone de dos algoritmos, el primero que es el "*BuscarCandidatos*", que sigue la estrategia divide y conquista. El segundo es el "*ElementoMayoritario*", que invoca al primero y valida que ese elemento dado, en caso de que haya un candidato, realmente sea mayoritario.

Complejidad temporal

Elemento mayoritario – Complejidad temporal

```
ALGORITMO BUSCARCANDIDATO
Entrada: V: Vector<entero>, inicio:entero, fin:entero
Salida: c: entero
  si fin < inicio
    devolver  No hay cadidato
  sino
    si inicio = fin
      devolver  V[inicio]
    sino
      j ← inicio
      si esPar(fin − inicio + 1)
        i ← inicio + 1
        mientras i ≤ fin
          si V[i − 1] = V[i]
            V[j] ← V[i]
            j ← j + 1
          fin si
          i ← i + 2
        fin mientras
        devolver  BuscarCandidato(V, inicio, j − 1)
      sino
        i ← inicio + 1
        mientras i < fin
          si V[i − 1] = V[i]
            V[j] ← V[i]
            j ← j + 1
          fin si
          i ← i + 2
        fin mientras
        x ← BuscarCandidato(V, inicio, j − 1))
        si tieneSolucion  (x)
          devolver  x
        sino
          devolver  V[fin]
        fin si
      fin si
    fin si
  fin si
```

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n/2) + P^1(n) & \text{si } n > 1 \end{cases}$$

$a = 1,\ b = 2 \text{ y } k = 1$ → $\mathcal{O}(n)$

Elemento Mayoritario – Complejidad Temporal

Algoritmo ElementoMayoritario

Entrada: V: Vector<entero>

Salida: x: entero

```
entero x ← BuscarCandidato(V, 0, longitud(S) − 1)        O(n)
si existe(x)
   suma ← 0
   para entero i = 0 hasta longitud(V) − 1               O(n)
      si x = V[i]
         suma ← suma + 1
      fin si
   fin para
   si suma > longitud(V)/2
      devolver x
   sino
      devolver No existe elemento mayoritario
   fin si
sino
   devolver No existe elemento mayoritario
fin si
```

La complejidad de todo el algoritmo es: $\boxed{\mathcal{O}(n)}$

Función de Fibonacci

Función de Fibonacci

La función de Fibonacci se define con la siguiente función:

$$F(n) = \begin{cases} 1 & \text{si } n < 2 \\ F(n-1) + F(n-2) & \text{sino} \end{cases}$$

En donde el valor de la función es 1 para ***n** < 2,* y sino para un ***n** ≥ 2* se define como la suma de *Fibonacci (n -1) + Fibonacci (n-2).*

Función de Fibonacci - Estrategia

La estrategia por divide y conquista para la función de Fibonacci se puede describir como:

Caso base	Por definición de la función si $n < 2$ se devuelve 1(uno)
Subproblemas	Por definición de la función, el problema general se divide en dos problemas menores, uno con $n-1$ y otro con $n-2$
Combinación	Sumatoria del resultado de $n-1$ y de $n-2$

Función de Fibonacci - Algoritmo

ALGORITMO FIBONACCI

Entrada: n: número para el cálculo de la función

Salida: valor correspondiente al cálculo de Fibonacci

```
sí (n<2)
  devolver 1
fin si
sino
      devolver Fibonacci(n-1) + Fibonacci(n-2)
fin sino
fin Fibonacci
```

Función de Fibonacci – Complejidad Temporal

ALGORITMO FIBONACCI

Entrada: n: número para el cálculo de la función

Salida: valor correspondiente al cálculo de Fibonacci

sí (n<2)

devolver 1

fin si

sino

devolver Fibonacci(n-1) + Fibonacci(n-2)

fin sino

fin Fibonacci

Caso Resta:

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

a=2 ; b = 1; k=0

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

Por las opciones de resolución de las funciones se considera *a > 1*, entonces nos queda:

$$\boxed{\Theta\ (2^n)}$$

Función de Fibonacci – Análisis resolución

- ¿Es una buena solución la obtenida para resolver Fibonacci utilizando la técnica divide y conquista?
  Si analizamos la complejidad temporal se podría decir que no.

- Uno de los puntos que no favorece la solución dada es que un mismo valor se calcula varias veces, sin poder ser reutilizado.

- Esos valores que se repiten se pueden ver en la siguiente secuencia, en dónde ***F(3)*** o ***F(2)*** se ven más de una vez, lo mismo sucedería con ***F(1)*** si se sigue la apertura del cálculo:

$$F(5) = \underbrace{F(4) + F(3)}_{F(5)} = \underbrace{\underbrace{F(3) + F(2)}_{F(4)} + \underbrace{F(2) + F(1)}_{F(3)}}_{F(5)} =$$

Greedy

Qué es la técnica de diseño de algoritmos Greedy

- Es una técnica de diseño de algoritmos que también se suele conocer como “Algoritmos Voraces”, por su traducción desde el inglés.

- Un algoritmo Greedy es aquel que va construyendo la solución a partir de decisiones parciales basadas en la información disponible en cada momento. No mira hacia adelante, es decir que no ve los efectos de las decisiones a futuro y nunca reconsidera una decisión ya tomada.

- En general se utilizan para resolver problemas de ***optimización***.

- Suelen ser muy ***eficientes*** pero se debe validar su ***correctitud*** mediante una demostración teórica.

Algoritmos Greedy - Candidatos

- La estrategia de resolución de los problemas por Greedy se basa en identificar un conjunto de candidatos a formar parte de la solución.

- En cada paso se toma uno de los candidatos, el mas apropiado según un criterio definido para la solución del problema, y se evalúa si sirve o no, si sirve se agrega a la solución y si no se descarta.

- Para ello se debe poder saber en todo momento dado un candidato si esta pendiente de ser evaluado, si fue evaluado y agregado a la solución o si fue descartado.

Algoritmos Greedy - Estrategia

En un algoritmo Greedy se pueden identificar cuatro funciones:

- ***La función selección***: es la que selecciona el mejor candidato dentro de los pendientes.

- ***La función factibilidad***: evalúa si un candidato seleccionado es factible de formar parte de la solución.

- ***La función solución***: evalúa si un conjunto solución propuesto conforma la solución al problema.

- ***La función objetivo***: es la que define el resultado buscado de maximizar o *minimizar.*

Algoritmos Greedy - Esquema

ALGORITMO GREEDY

Entrada: C: conjunto de candidatos

Salida: S solución del problema

```
mientras C ≠ ∅ Y NO esSolucion(S)
    x ← Seleccionar(C)
    C ← C \ {x}
    si esFactible(S ∪ {x})
        S ← S ∪ {x}
    fin si
fin mientras
si esSolucion(S)
    devolver S
sino
    devolver No hay solución
fin si
```

Problema del Cambio

Problema del Cambio Greedy

El problema del cambio consiste en encontrar la forma de devolver un vuelto de valor **v** con monedas, utilizando la mínima cantidad de monedas. Las monedas son de denominaciones **d1; d2; …; dn,** con una cantidad infinita cada denominación.

EJEMPLO

Monedas {**$25, $10, $100, $50, $5, $1**}

Si debemos dar un cambio **$ 1,77**:

La mejor solución sería: {$100, $50, $25, $1, $1}

Problema del Cambio *Greedy* - Estrategia

Conjunto candidatos	Las diferentes denominaciones de monedas
Función selección	Intentar seleccionar una moneda de mayor valor
Función factibilidad	Validar que la monedad seleccionada no supere el monto que debe pagarse
Función solución	Verificar que no se haya alcanzado el valor que queremos pagar
Función objetivo	Minimizar la cantidad de monedas a devolver para un valor dado

Problema del Cambio Greedy - Algoritmo

Algoritmo Cambio

Entrada: v: entero
Salida: n: entero

entero $n \leftarrow 0$
entero $s \leftarrow 0$
entero $i \leftarrow 0$
Vector $monedas = [100, 50, 25, 10, 5, 1]$
mientras $(s < v)$ **Y** $(i < longitud(monedas))$
  si $s + monedas[i] < v$
    $s \leftarrow s + monedas[i]$
    $n \leftarrow n + 1$
  sino
    $i \leftarrow i + 1$
  fin si
fin mientras
si $i < longitud(monedas)$
  devolver n
sino
  devolver No hay solución
fin si

Suponer el vector de monedas ordenado de mayor a menor.

Se comienza a utilizar la moneda de mayor valor, y se utiliza tantas veces mientras sea posible. Cuando esa moneda no se puede utilizar porque supera el valor a buscar se intenta con la siguiente.

En éste caso el algoritmo devuelve la cantidad de monedas, podría modificarse y devolver qué monedas, siendo la estrategia la misma

Problema del Cambio Greedy – Complejidad Temporal

ALGORITMO CAMBIO

Entrada: v: entero
Salida: n: entero

```
entero n ← 0
entero s ← 0
entero i ← 0
Vector monedas = [100, 50, 25, 10, 5, 1]
mientras (s < v) Y (i < longitud(monedas))
    si s + monedas[i] < v
        s ← s + monedas[i]
        n ← n + 1
    sino
        i ← i + 1
    fin si
fin mientras
si i < longitud(monedas)
    devolver n
sino
    devolver No hay solución
fin si
```

El análisis de complejidad esta dado por el ciclo, que en el peor caso puede iterar hasta v, que es el valor de la entrada, por lo tanto la complejidad temporal es:

$$\boxed{\mathcal{O}(n)}$$

Se debe tener en cuenta que si el ordenamiento de las monedas se tuviera que realizar llevaría la siguiente complejidad temporal, siendo n en éste caso la cantidad de monedas

$$\boxed{\Theta(n \log(n))}$$

Problema del Cambio Greedy - Correctitud

- La solución al problema del cambio dada por Greedy para el conjunto de monedas dado {$25, $10, $100, $50, $5, $1} es correcto por las denominaciones particulares de monedas.

- Ahora si consideramos el caso {$6, $4, $1} y debemos devolver por ejemplo $8, el algoritmo daría como solución {$6, $1, $1} cuando en realidad la solución es {$4, $4}, por lo tanto, deberemos analizar para éste caso si se puede resolver con otra técnica.

Nota: AHO, J.; HOPCROFT, John E. y ULLMAN, Jeffrey D.. Estructuras de Datos y Algoritmos. 1988. Wilmington : Addison Wesley Iberoamericana. 438 p. ISBN: 0201640249684443455.

Problema de la Mochila

Problema de la Mochila Greedy

El problema consiste en que se tienen **n** objetos y una mochila. Para ***i = 1; 2; ..n***, el objeto ***i*** tiene un peso positivo ***Pi*** y un valor positivo ***Vi***. La mochila puede llevar un peso que no sobrepase **P**. El objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos transportados, respetando la limitación de capacidad impuesta.

Los objetos pueden ser fraccionados, si una fracción ***Xi*** (0 <= Xi <= 1) del objeto ***i*** es ubicada en la mochila contribuye en ***Xi * Pi*** al peso total de la mochila y en ***Xi * Vi*** al valor de la carga.

Problema de la Mochila Greedy - Ejemplo

Supongamos que tenemos 4 objetos y una mochila con capacidad P = 17. Los valores de los objetos son (v1,v2,v3,v4) = (5,2,7,4) y los pesos de los objetos son (p1,p2,p3,p4) = (9,5,3,6).

Algunas combinaciones para completar la mochila son:

x1	x2	x3	x4	xi*pi	xi*vi
1	1/2	1/2	1/2	17	11,50
1	1	1	0	17	14,00
1	0	1	1/2	17	14,00
8/9	0	1	1	17	15,44

La opción que maximiza respeta la relación v/p de mayor a menor:

	Valor				Peso			
	5	2	7	4	9	5	3	6
v/p	0,5556	0,4	2,3333	0,6667				

Problema de la mochila *Greedy* - Estrategia

Conjunto candidatos	Los diferentes objetos aún disponibles a utilizar
Función selección	Seleccionar el objeto de mayor relación *v/p*
Función factibilidad	Validar que el objeto no supere el peso de la mochila, sino ingresar una fracción del mismo
Función solución	Verificar que la mochila se haya completado o se hayan utilizado todos los objetos
Función objetivo	Maximizar el beneficio obtenido en base a los objetos ingresados en la mochila

Problema de la Mochila Greedy - Algoritmo

ALGORITMO MOCHILA

Entrada: O: Vector<Objeto>, p: entero
Salida: R: Vector<real>

$Ordenar(O)$ //según la razón valor/peso
para $i = 0$ **hasta** $n - 1$

$R[i] \leftarrow 0$

fin para
$suma \leftarrow 0$
$objeto \leftarrow 0$
mientras $suma < p$
$R[objeto] \leftarrow MIN(1, (p - suma)/O[objeto].peso)$
$suma \leftarrow suma + MIN(1, (p - suma)/O[objeto], peso) * O[objeto].peso$
$objeto \leftarrow objeto + 1$
fin mientras
devolver R

Problema de la Mochila Greedy – Complejidad Temporal

ALGORITMO MOCHILA

Entrada: O: Vector<Objeto>, p: entero
Salida: R: Vector<real>

$Ordenar(O)$ //según la razón valor/peso — $\Theta(n\log(n))$

para $i = 0$ hasta $n-1$ — $\mathcal{O}(n)$

$R[i] \leftarrow 0$

fin para

$suma \leftarrow 0$

$objeto \leftarrow 0$

mientras $suma < p$ — $\mathcal{O}(n)$

La cantidad máxima que itera es utilizando todos los objetos

$R[objeto] \leftarrow MIN(1, (p - suma)/O[objeto].peso)$

$suma \leftarrow suma + MIN(1, (p - suma)/O[objeto], peso) * O[objeto].peso$

$objeto \leftarrow objeto + 1$

fin mientras

devolver R

Complejidad del Algoritmo $\Theta(n\log(n))$

Problema de la Mochila Greedy – Correctitud

La solución del problema de la mochila por Greedy para elementos que se pueden fraccionar es una solución correcta.

Se van a ver otras variantes del problema que se deben resolver con otras técnicas de diseño de algoritmo, para el caso de que los elementos no se puedan fraccionar.

Nota: BRASSARD, G.; BRATLEY, P. Fundamental of Algorithmics. 1996. Prentice Hall. xiii, 524 p. : ill. ISBN: 0133350681.

¡Muchas gracias!