EJEMPLOS DE PROBLEMAS RESUELTOS

UNIVERSIDAD DE MÁLAGA
E.T.S. INGENIERÍA INFORMÁTICA

Fco Javier Moreno Barea

[Parcial 1 – 19/20] Sea un array ordenado estrictamente creciente, de números naturales, con A[0]=k>0. Se pide obtener el menor elemento superior a k que no está en el array. Si están todos, el elemento que falta es el número siguiente. Ejemplos:

Input: A[] = [21, 22, 24, 30, 36, 37] Output: El número que falta es 23

Input: A[] = [21, 22, 23, 24, 25, 26] Output: El número que falta es 27

Fuerza bruta

```
public int menorElemSup(int[] A){
        int k = A[0];
        int n = A.length();
        int i = 1;
        while ((i < n) && (A[i]-i ==k)) {
                i++;
        }
        return k + i;
}
```

Recursivo – Divide y Vencerás

La diferencia entre A[i] e i se mantendrá constante igual a k, mientras el contenido vaya creciendo de 1 en 1, por ser el vector ordenado estrictamente creciente. Entonces, si en la última componente la diferencia también es k, no falta ningún número y será n+k.
En caso contrario hay que buscar el mínimo dentro del array:

```
public int menorElemSup(int[] A){
        int k = A[0];
        int n = A.length();
        if (A[n-1] – (n-1) == k) {    min = n + k; }
        else {                        min = Buscar(A, 0, n-1); }
        return min;
}
```

[Parcial 1 – 19/20] Sea un array ordenado estrictamente creciente, de números naturales, con A[0]=k>0. Se pide obtener el menor elemento superior a k que no está en el array. Si están todos, el elemento que falta es el número siguiente. Ejemplos:

Input: A[] = [21, 22, 24, 30, 36, 37] Output: El número que falta es 23

Input: A[] = [21, 22, 23, 24, 25, 26] Output: El número que falta es 27

Ahora vamos a desarrollar Buscar en el array A y entre los límites inf y sup, que devuelve el menor valor omitido. Si los dos límites de búsqueda coinciden, entonces: (a) puede ser que en esa posición se conserve la diferencia k, entonces el mínimo omitido es esa componente más 1; (b) la diferencia es mayor que k, entonces el menor omitido es el elemento de la posición anterior (que sí está) más 1. Si, por el contrario, los dos límites de búsqueda no coinciden entonces se procede según búsqueda binaria: calcular el punto medio, ver la diferencia en el punto medio, si es k, buscar en la parte superior, si no es k (que será mayor que k), buscar en la parte inferior.

```
public int Buscar (int[] A, int inf, int sup) {
    if (inf==sup) {
        if ( A[inf]-inf == k) { min = A[inf] + 1; }
        else {                          min = A[inf-1] + 1; }
    } else {
        med = (inf + sup) / 2;
        if (A[med]-med == k) {
            min = Buscar (A, med+1, sup); }
        else {
            min = Buscar (A, inf, med-1); }
    }
    return min;
}
```

[Parcial 1 – 18/19] Dado un array A, ordenado creciente, de números naturales, todos distintos, se pide encontrar el menor elemento que no está. Ejemplos:

Input: A[] = [0, 1, 2, 6, 9, 11, 15] Output: El menor elemento omitido es 3

Input: A[] = [1, 2, 3, 4, 6, 9, 11,15] Output: El menor elemento omitido es 0

Input: A[] = [0, 1, 2, 3, 4, 5, 6] Output: El menor elemento omitido es 7

Fuerza bruta:

```
public static int menorOm(int[] A) {
        int minOmitido;
        int i=0;
        if (A[0]>0) {      minOmitido = 0;}
        else {             i = 1;}
        while ((i<A.length) && (A[i]==i)) {
                i++;
        }
        return i;
}
```

Recursivo – Divide y Vencerás

Sea n +1 la longitud del array A con índices de componentes desde 0 hasta n. Veamos primero los casos base:

- Si A[0] > 0, todas las componentes del array son números mayores que 0. Entonces, ¿cuál es el menor que no está? Pues el 0.

-Sino, si A[n]= n, como tienen que ser todos distintos y están en orden creciente, esto obliga a que estén todos desde el 0. Entonces el primero omitido es n+1.

En el resto de los casos A[0]=0 y A[n]>n, lo que obliga a que se ha omitido uno, o muchos, en medio. Pero igual que A[n]=n obliga a que estén todos, A[i]=i, obliga a que estén todos los anteriores a i.

[Parcial 1 – 18/19] Dado un array A, ordenado creciente, de números naturales, todos distintos, se pide encontrar el menor elemento que no está. Ejemplos:

Input: A[] = [0, 1, 2, 6, 9, 11, 15] Output: El menor elemento omitido es 3

Input: A[] = [1, 2, 3, 4, 6, 9, 11,15] Output: El menor elemento omitido es 0

Input: A[] = [0, 1, 2, 3, 4, 5, 6] Output: El menor elemento omitido es 7

Por lo tanto, hay que buscar la primera vez que A[i]>i, y entonces el resultado será i (piénsalo un poco).

Conclusión: me sitúo en la componente mitad si A[med]= med, entonces lo que busco está hacia arriba. Si A[med]>med, lo que busco está hacia abajo. Tengo que repetir el proceso hasta que me quede solo uno.

Entonces ocurrirá que A[med]> med y A[med-1] = med-1, con lo cual el resultado es med.

En resumen: una variante de búsqueda binaria.

```
public static int menorOm (int[] A, int inf, int sup) {
    int minOm; int n = a.length;
    if (A[inf] > inf) {                minOm = inf;}
    else if ( A[sup] == sup ) {    minOm = sup+1;}
    else if (sup-inf <= 1) {        minOm = inf;}
    else {
        int med = (inf + sup) / 2;
        if (A[med] == med) {
            minOm = menorOm(A, med+1, sup); }
        else {
            minOm = menorOm(A, inf, med-1); } }
    return minOm;
}
```

1) Dado un vector ordenado V de n enteros distintos, escribir un algoritmo que en tiempo O(log n) encuentre un número i tal que 1≤i≤n y V[i] = i, siempre que exista.

```
public static int localiza(int[] vector, int prim, int ult){
    int indice = -1;
    if(prim<=ult){
        int medio = (prim+ult) / 2;
        if(vector[medio]==medio){
            indice = medio; }
        else if(vector[medio]>medio){
            indice = localiza(vector, prim,medio-1); }
        else{
            indice = localiza(vector,medio+1,ult); }
    }
    return indice;
}
```

Para calcular la complejidad, T(n) supone, en el caso peor tres comparaciones y una llamada recursiva al problema con tamaño n/2, luego:

$$T(n) = T(n/2) + 4$$

Teorema Maestro simplificado:

$$a = 1, b = 2, d = 0 \Rightarrow a = b^d$$

$$T(n) \in \Theta(n^0 \log n) \in \Theta(\log n)$$

2) Dados dos vectores de n enteros cada uno y ordenados de forma creciente, escribir un algoritmo para hallar la mediana del vector formado por los 2n enteros, cuya complejidad sea log n.

```
public int mediana(int[] x, int[] y, int primX, int ultX, int primY, int ultY){
        int nitems, posX, posY;
        if((primX>=ultX) && (primY>=ultY)){      return Math.min(x[ultX], y[ultY]); }
        nitems = ultX – primX + 1;
        if(nitems==2){
                if(x[ultX]<y[primY]){              return x[ultX]; }
                else if(y[ultY]<x[primX]){         return y[ultY]; }
                else{                              return Math.max(x[primX], y[primY]); }
        }
        nitems = (nitems-1)/2;
        posX = primX + nitems;
        posY = primY + nitems;
        if(x[posX]==y[posY]){              return (x[posX]); }
        else if(x[posX]<y[posY]){          return (mediana(x, y, ultX - nitems, ultX, primY, primY + nitems)); }
        else{                              return (mediana(x, y, primX, primX + nitems, ultY - nitems, ultY)); }
}
```

3) Dado un vector V de *n* elementos (necesariamente ordenables), se dice que un elemento *x* es mayoritario en V cuando el número de veces que *x* aparece en V es mayor que *n*/2. Escribir un algoritmo que en tiempo O(*n* log *n*) decida si un vector tiene un elemento mayoritario y lo devuelva si lo tiene.

```
divisionenpares (A[0…n-1]){
    sea B[0…(n-1)/2]== vacio;
    Para cada i∈{0…n-1}{
        Si A[i]==A[i+1]{
            B[j]=A[i];
            j++;
            i++;
        }
    }
    return B;
}
```

```
candidatomayoritario (A[0…n-1]) {
    si n=0{
        return -1;
    }
    si n=1{
        return A[0];
    }
    sea B=divisionenpares(A);
    return candidatomayoritario(B);
}
```

```
apariciones(A[0…n-1],x,i,j){
    sea c=0;
    para cada k ∈ {i…j} {
        si A[k]==x{
            c++;}
    return c;
}
```

```
comprobacioncandidato(A[0…n-1]){
    ordenar(A);
    sea m=candidatomayoritario(A);
    Si m=-1 ó apariciones(A,m,0,n-1)≤n/2){
        return -1;
    } else{
        return m;
    }
}
```

Ejemplo: A = [0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 5, 5, 6]
B1 = [0, 1, 2, 2, 2, 2, 5, ∅]
B2 = [2, 2, ∅]
B3 = [2] ----> return m=2

4) Dado un vector V de n enteros. Escribir un algoritmo que en tiempo O(n log n) encuentre el subvector (formado por elementos consecutivos) cuya suma sea máxima.

```
public static vector subVectorM(int[] v,int inf,int sup) {
        if (inf > sup) return new vector(inf,sup,0);
        if (inf == sup) return new vector(inf,inf,v[inf]);

        int medio = (inf+sup)/2;
        vector v1 = subVectorM(v,inf,medio);
        vector v2 = subVectorM(v,medio+1,sup);
        int max1 = v[medio];
        int s1 = v[medio]; int j1 = medio;

        for (int i=medio-1; i>=inf; i--){
                s1= s1+v[i];
                if (s1 > max1){
                        max1 = s1;
                        j1 = i;
                }
        }
        int max2 = v[medio+1]; int s2 = v[medio+1];
                        ⋮
                int j2 = medio+1;
                for (int i=medio+2; i<=sup; i++){
                        s2= s2+v [i];
                        if (s2 > max2){
                                max2 = s2;
                                j2 = i;
                        }
                }
                int max = max1 + max2;
                if (v1.suma() >= v2.suma() && v1.suma() >= max){
                        return v1;
                } else if (v2.suma() >= v1.suma() && v2.suma() >= max){
                        return v2;
                } else {
                        return new vector(j1,j2,max);
                }
}
```

5) Dado un vector V de enteros todos distintos y un número entero S. Diseñar un algoritmo que en tiempo O(n log n) determine si existen o no dos elementos de V tales que su suma sea S.

- Paso 1: Ordenar V de menor a mayor. Para ello podemos utilizar directamente el método mergesort.
Este método que vamos a denominar QM(n) es conocido y tiene complejidad $\mathrm{QS(n)} \in \mathrm{O}(n \log n)$.

- Paso 2: Para cada i desde 0 hasta n-1 hacemos una búsqueda binaria S-V[i] en las posiciones de i+1 hasta n-1.
Este método que vamos a denominar BB(n) es conocido y tiene complejidad $\mathrm{BB(n)} \in \mathrm{O}(\log n)$.

(Comentario: como mergesort y busquedaBinaria son conocidos, no es necesario escribir la implementación)

```
public static boolean sumaS(int[] V, int S) {
        boolean b = false;
        quickSort(V, 0, V.length-1);
        for (int i=0; i< V.length; i++){
                int res = busquedaBinaria(V, i+1, V.length-1, S-V[i]);
                if (res != -1) {
                        b = true;
                        break;
                }
        }
        return b;
}
```

- Paso 2 (sig.): Como BB(n) tenemos que llamarlo para cada i desde 0 hasta n-1, tenemos:

$$\mathrm{P2(n)} \approx \sum_{i=0}^{n-1} \mathrm{BB(n)} = n \cdot \mathrm{BB(n)} = n \log n$$

De esta forma, tenemos que nuestra implementación es una composición de ambas partes, por lo tanto:

$$\mathrm{Coste(n)} = QM(n) + P2(n) = n \log n + n \log n$$

$$\mathrm{Coste(n)} \in \mathrm{O}(n \log n)$$

5) [Tema 1] / 1) [Complementario Tema 3] Dado un valor numérico x y un valor natural n≥0, queremos escribir un algoritmo para calcular x^n. Se pide:

a) Diseñar un algoritmo para resolver el problema mediante un enfoque iterativo, e indicar su complejidad en términos del numero de multiplicaciones ejecutadas.
b) Diseñar un algoritmo alternativo siguiendo un enfoque de divide y vencerás, de manera que sea mas eficiente que el enfoque anterior.
c) Plantear y resolver (quizás mediante el Teorema Maestro) una recurrencia para el coste computacional del algoritmo de divide y vencerás.

```
public int potencia(int x, int n){
        int res = 1;
        for (int i=0; i<n; i++) {
                res = res*x;
        }
        return res;
}
```

Para calcular la complejidad T(n):

$$T(n) \approx \sum_{i=0}^{n-1} 1 = n \Longrightarrow T(n) \in \Theta(n)$$

```
public int potencia(int x, int n){
        if (n==1){
                return x;
        } else if (n % 2 == 0) {
                int y = potencia(x, n/2);
                return y*y;
        } else {
                int y = potencia(x, (n-1)/2);
                return y*y*x;
        }
}
```

Para calcular la complejidad, T(n) supone, en el caso peor dos comparaciones y una llamada recursiva al problema con tamaño n/2, luego:

$$T(n) = T(n/2) + 4$$

Teorema Maestro simplificado:

$$a = 1, b = 2, d = 0 \Rightarrow a = b^d$$

$$T(n) \in \Theta(n^0 \log n) \in \Theta(\log n)$$

7) Sea A un array unimodal consistente en una secuencia estrictamente creciente de números enteros seguida por una secuencia estrictamente decreciente. Se pretende construir una función int pico (int[] A); que, dado un array unimodal A, devuelva su pico, o sea, el valor del array a partir del cual la secuencia pasa de ser creciente a decreciente. (Nota: se asumirá que las secuencias creciente y decreciente no son vacías y que el array que se recibe es efectivamente unimodal y no únicamente una secuencia de números creciente o decreciente). Por ejemplo:

si A = {1,5,7,9,6}, entonces pico(A)=9.

Se pide:

a) Diseñar un algoritmo para resolver el problema mediante un enfoque de fuerza bruta, e indicar su complejidad en términos del número de operaciones elementales ejecutadas (se puede simplificar realizando el cálculo con respecto al número de comparaciones entre elementos realizadas).

```
public int picoFuerzaBruta() {
        int pico;
        int n = vector.length;
        if (n==2) {
                pico = vector[1];
        } else {
                int i=2;
                while (vector[i-1] < vector[i]
                        && vector[i] < vector[i+1]) {
                        i++;
                }
                pico = vector[i];
        }
        return pico;
}
```

$T(n) \in \Theta(n)$

```
public int picoDivideYVenceras() {
        return picoDivideYVenceras(0, vector.length-1);
}
protected int picoDivideYVenceras(int inf, int sup) {
        int pico;
        int n = sup-inf+1;
        if (n==2) {
                pico = Math.max(vector[inf], vector[inf+1]);
        } else {
                n = n/2;
                int i = inf + n;
                if ( vector[i-1]< vector[i] && vector[i] < vector[i+1]) {
                // crecimiento
                        pico = picoDivideYVenceras(i+1, sup);
                } else if (vector[i+1] < vector[i] && vector[i] < vector[i-1]) {
                // decrecimiento
                        pico = picoDivideYVenceras(inf, i-1);
                } else {
                // porque no hay valles
                        pico = vector[i];
                }
        }
        return pico;
}
```

b) Diseñar un algoritmo alternativo siguiendo un enfoque de divide y vencerás, de manera que sea más eficiente que el enfoque de fuerza bruta.

c) Plantear y resolver (quizás mediante el Teorema Maestro) una recurrencia para el coste computacional del algoritmo de divide y vencerás.

Para calcular la complejidad, T(n) supone, en el caso peor cuatro comparaciones y una llamada recursiva al problema con tamaño n/2, luego:

$$T(n) = T(n/2) + 4$$

Teorema Maestro simplificado:

$$a = 1, b = 2, d = 0 \Rightarrow a = b^d$$

$$T(n) \in \Theta(n^0 \log n) \in \Theta(\log n)$$

4) [Complementario] En una habitación oscura se tienen n tornillos de varios tamaños y las correspondientes n tuercas. Es necesario emparejar cada tornillo con su tuerca correspondiente pero no es posible comparar tornillos con tornillos ni tuercas con tuercas, la única comparación posible es la de tuercas con tornillos para decidir si es demasiado grande, demasiado pequeña o se ajusta al tornillo. Escribir un algoritmo que en tiempo O(n logn) empareje tornillos con tuercas.

```
public void emparejar(int[] tuercas, int[] tornillos, int p, int u) {
    if (p < u) {
        // Usamos un tornillo para separar las tuercas
        int m = partirTT(tuercas, p, u, tornillos[u]);

        // Usamos la tuerca asociada al tornillo anterior
        // para separar los tornillos
        partirTT(tornillos, p, u, tuercas[m]);

        // Como en QuickSort, repetimos la operación con la
        // mitad inferior (menores que el pivote) y la superior
        // (mayores que el pivote) de los arrays
        emparejar(tuercas, tornillos, p, m - 1);
        emparejar(tuercas, tornillos, m + 1, u);
    }
}
```

```
public int partirTT(int[] a, int p, int u, int pivote) {
    int m = p, t1, t2;
    for (int j = p; j < u; j++) {
        if (a[j] < pivote) {
            t1 = a[m];    a[m] = a[j];      a[j] = t1;
            m++;
        } else if (a[j] == pivote) {
            t1 = a[j];      a[j] = a[u];      a[u] = t1;
            j--;
        }
    }
    t2 = a[m]; a[m] = a[u]; a[u] = t2;
    return m;
}
```

[Ejercicio Voluntario] Sea A un array unimodal consistente en una secuencia estrictamente creciente de números enteros seguida por una secuencia estrictamente decreciente y seguida por otra secuencia estrictamente creciente. Se pretende construir una función int desnivel (int[] A); que, dado un array unimodal A, devuelva su desnivel, o sea, la diferencia entre el valor del array a partir del cual la secuencia pasa de ser creciente a decreciente, con el valor del array a partir del cual la secuencia pasa de ser decreciente a creciente. (Nota: se asumirá que las secuencias creciente y decreciente no son vacías y que el array que se recibe es efectivamente unimodal y no únicamente una secuencia de números creciente o decreciente).

Por ejemplo:

si A = {100, 120, 134, 142, 150, 154, 149, 141, 130, 120, 119, 102, 96, 91, 98, 104, 108, 111}, entonces desnivel(A) = 63.

Se pide:

a) Realizar la especificación formal de la precondición y la postcondición del procedimiento anterior.
b) Diseñar un algoritmo para resolver el problema mediante un enfoque de fuerza bruta (FB).
c) Indicar la complejidad del algoritmo de FB en términos del número de operaciones elementales ejecutadas (se puede simplificar realizando el cálculo con respecto al número de comparaciones entre elementos realizadas).
d) Diseñar un algoritmo alternativo siguiendo un enfoque de divide y vencerás (DyV), de manera que sea más eficiente que el enfoque de fuerza bruta.
e) Plantear y resolver el coste computacional del algoritmo de DyV (quizás mediante el Teorema Maestro).
f) Si tuviéramos que el coste del anterior algoritmo de DyV es T(n) tal y como está señalado más abajo, calcular el coste exacto de complejidad.

$$T(n) = 2\,T(n/3) + \ n^2 + 6\log n$$