Análisis y diseño de algoritmos

Relación de ejercicios Tema 3: Divide y Vencerás

(web con ejercicios: http://www.geeksforgeeks.org/)

Ejercicios obligatorios

1. Dado un vector ordenado V de *n* enteros distintos, escribir un algoritmo que en tiempo O(lg *n*) encuentre un número *i* tal que $1 \leq i \leq n$ 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;
    }
```

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 2*n* enteros, cuya complejidad sea lg *n*.

```
Int Mediana (int [ ]X,int [ ]Y, int primX,ultX,primY,ultY){
    int posX, posY; int numitems;
IF (primX ≥ ultX) && (primY ≥ ultY) // Nuestro caso Base
        return Min2(X[ultX], Y[ultY])
 numitems=ultX-primX+1;
    if (numitems==2){ // Si solo hay dos elementos
        if (X[ultX]<Y[primY])  return X[ultX];
        elseif (Y[ultY]<X[primX]) return Y[ultY];
else return Max2(X[primX], Y[primY]);
 }
 numitems=(numitems-1) /2; // Estamos partiendo el problema en dos mitades
 posX=primX+numitems; posY=primY+numitems;
 if (X[posX]==Y[posY]) return X[posX]); // La mediana coincide en el punto
    elseif (X[posX]<Y[posY])
        return Mediana(X,Y,ultX-numitems,ultX,primY,primY+numitems);
    else
        return Mediana(X,Y,primX,primX+nitems,ultY-nitems,ultY);
} //fin mediana
```

```
public int mediana(int x[],int y[], int primX, int ultX, int primY, int
ultY){

		if((primX<=ultX) && (primY>=ultY)){
			return Math.min(x[ultX], y[ultY]);
		}
		int nitems;
		int posX;
		int 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 (no 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* lg *n*) decida si un vector tiene un elemento mayoritario y lo devuelva si lo tiene.

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

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

        si n es par ó m==-1{
                return m;
        }
        else{
        return A[n];
        }
}

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

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

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

• Solución: Un primer acercamiento podría consistir en dividir el vector por la mitad y considerar los subvectores izquierdo y derecho. Sin embargo, esta solución no sería correcta al no considerarse los subvectores que atraviesan la mitad del vector. Por tanto, habrá que considerar los tres casos: subvector izquierdo, subvector derecho y subvector central.

• Subvector izquierdo: Lo primero que se ha de hacer es comparar la longitud del vector con la del subvector; si la longitud del primero es mayor, se disminuye su tamaño a la mitad y se vuelve a aplicar el método, esta vez para calcular el subvector izquierdo. Por tanto, el coste de la llamada recursiva es , siendo n la longitud del vector. En el momento en que la longitud del vector sea menor que la del subvector, se suman todas las posiciones consecutivas hasta llegar al final y se asignan los límites del subvector (inicio y fin). Es posible que dicho subvector tenga menos longitud que la deseada; por ello, cuando acaba la llamada para el subvector izquierdo, es necesario aumentar su longitud por la derecha hasta llegar a la longitud pedida. Según se aumenta la longitud, también habrá que ir actualizando la suma total del subvector.

• Subvector derecho: Inmediatamente después de haber calculado el subvector izquierdo, se hará exactamente lo mismo con el subvector derecho, ya que las anteriores llamadas han ido dividendo el vector en dos. El coste para la llamada del subvector derecho también es . Igual que antes, cuando se haya terminado de calcular el subvector derecho, es posible que este no tenga la longitud pedida, por lo cual se aumenta su longitud (esta vez por la izquierda) hasta llegar a la longitud pedida. Al finalizar el proceso, se actualiza su suma.

• Subvector central: Lo primero que se hará será dividir el vector por la mitad. Acto seguido, se comprueba hasta dónde se puede calcular con la longitud que se está pidiendo, teniendo en cuenta que el subvector tiene que pasar por el punto medio del vector.

```
import java.util.*;

public class SubVectorMaximo {
/**
* @param args
*/
public static class vector{
private final int inf,sup,suma;

public vector(int inf,int sup,int suma){
this.inf = inf;
this.sup = sup;
this.suma = suma;
}
public int inf() {
return inf;
}
public int sup() {
return sup;
}
```

```
public int suma() {
return suma;
}
public String tostring() {
return inf+" .. "+sup+":"+suma;
}
}


public static vector subVectorM(int[] v,int inf,int sup){

if (inf > sup) return new Vector(inf,sup,O);
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+l,sup);

int max1 = v[medio]; int s1 = v[medio]; int j1 = medio;
for(int i=medio-l; i>=inf; i--){
     s1= s1+v[i];
     if (s1 > max1){
               maxl = s1;
               j1 = i;
     }
 }
int max2 = v[medio+l]; int s2 = v[medio+l]; 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 (vl.suma() >= v2.suma() && vl.suma() >= max){
return v1;
} else if (v2.suma() >= vl.suma() && v2.suma() >= max){
return v2;
} else return new vector(j1,j2,max);
}

Public static void main(string[] args) {
jj TODO Auto-generated method stub
Random r = new Random();
i.nt lon = r.nextlnt(10)+I;
int[] v new int[lon];
```

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

Paso 1: Ordenar V de menor a mayor. Complejidad O(*n* lg *n*)
Paso 2, Para cada i desde 0 hasta n-1 buscar binario S-V[i] en las posiciones de i+1 hasta n-1. Complejidad O(*n* lg *n*).

Total: O(*n* lg *n*)

6. Sea un array A[] que contiene valores numéricos naturales todos distintos. Decimos que dos posiciones i, j (con i<j) forman una inversión si se cumple que A[i]>A[j]; es decir, si los elementos en las posiciones i y j no están relativamente ordenados de manera creciente entre sí.
   Deseamos obtener una función "int numInversiones (int[] A);" que devuelva el número total de inversiones del array. Por ejemplo, si A = {2, 4, 1, 3, 5}, entonces numInversiones(A)=3 (correspondientes a las posiciones (1,3), (2,3) y (2,4)).

   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 comparaciones entre elementos realizadas.
   b) Diseñar un algoritmo alternativo siguiendo un enfoque de divide y vencerás, de manera que sea más eficiente que el enfoque de fuerza bruta (Indicación: Si el alumno lo ve necesario para la resolución del problema, se podrán mezclar los elementos).
   c) Plantear y resolver mediante el Teorema Maestro una recurrencia para el coste computacional del algoritmo de divide y vencerás.

   SOLUCIÓN: (ejemplo A = (6, 9, 1, 14, 8, 12, 3, 2))

   Buen desarrollo en inglés: http://www.geeksforgeeks.org/counting-inversions/

   CONCEPTO DE INVERSIÓN
"Sea (A(1), A(2) , ... , A(n)) una secuencia de n distintos números. Si i < j y A(i) > A(j), entonces el par (i, j) es llamado una inversión de A."

Por ejemplo en la secuencia A = [2, 4, 1, 3, 5], tenemos 3 inversiones (A[1], A[3]) = (2, 1), (A[2], A[3]) = (4, 1) y (A[2], A[4]) = (4, 3) con respecto a un orden ascendente.

a)

```
int function inversionBruteForce(array A) {
  var inversions = 0;
  for (i in array) {
    for (j in array) {
      if ((i < j) && (array[i] > array[j]))
        inversions++;
    }
  }
  return inversions;
}
```

Complejidad: Θ(n^2)
(Para entradas muy grandes sería muy ineficiente, ya que precisamente su complejidad se debe a que hace todas las comparaciones posibles).

Usaremos Merge Sort añadiendo un paso más en cada recursión, que será contar las inversiones.
Veamos un ejemplo.

Si tuviéramos la secuencia [4, 2, 1, 3] y quisiéramos encontrar el número de inversiones (que es 4) vamos a dividir el problema en dos, la primera secuencia [4, 2] (izquierda) y la segunda secuencia [1, 3] (derecha).
En la secuencia de la izquierda vemos que tenemos 1 inversión y en la derecha no tenemos ninguna, entonces hasta ahora contamos 1. Ya que ya hemos analizado ambas partes por separado, ahora vamos ordenarlas para comparar elementos entre las dos partes, este paso no modificará el problema en absoluto pues ya llevamos la cuenta de las inversiones que tienen la subsecuencias que hemos ordenado, y el hecho de estar ordenadas nos da una ventaja, veamos porque.

Si ordenamos las subsecuencias tenemos entonces [2, 4] (izquierda) y [1, 3] (derecha), entonces comparamos ahora el primer elemento de la secuencia de la derecha, es decir 1, con el primer elemento de la secuencia de la izquierda, es decir 2, notamos entonces que 1, al estar en la derecha tiene una posición mayor (3) pero su valor es menor que elemento de la izquierda, con lo cual adicionamos uno mas a la cuenta de inversiones. Ahora fíjense que no tenemos que comparar ese elemento 1 con los demás elementos después de 2 (en este caso 4) ya que al estar ordenada la secuencia de la izquierda los elementos sobrantes tienen que ser mayores con lo que concluimos que si los comparáramos con 1, obtendríamos también mas inversiones (en este caso 1 mas), con lo cual el numero de inversiones adicionales seria el numero de elementos restantes en la secuencia de la izquierda. De igual manera procederíamos con el 3, con lo cual nos dará una inversión mas al compararla con 4. Y así al final el numero de inversiones que tendríamos será de 4 si repasamos todas las lineas subrayadas anteriormente.

Realizar la busqueda de inversiones de esta manera tiene la ventaja que al estar montado en el algoritmo Merge Sort va a tener su misma complejidad, la cual es **O(n*logn).**

Ahora en pseudocódigo.

Construiremos nuestra función **"sortAndCount"**, la cual tendrá como entrada una secuencia de números y tendrá como salida la secuencia de números ordenada y el numero de inversiones que hay en ella.

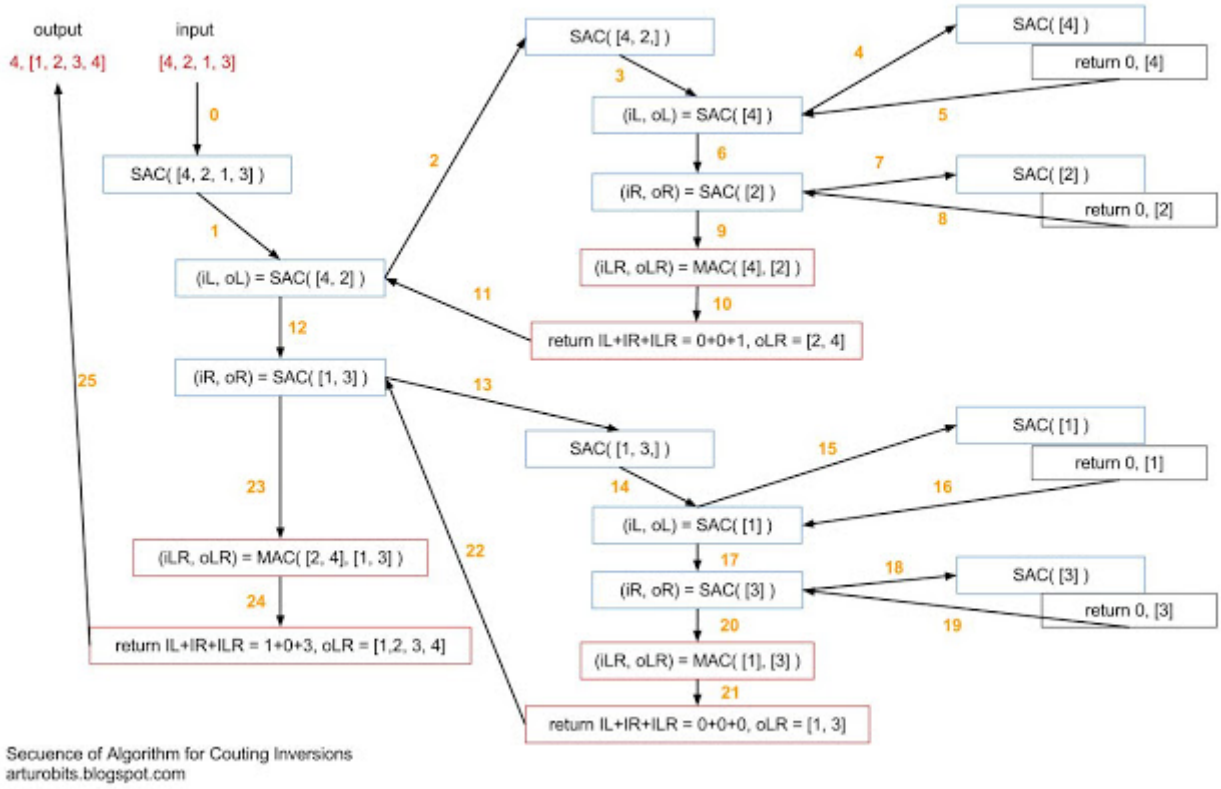
```
1. sortAndCount(A)
2.  /* L es la mitad izquierda
3.   * R es la mitad derecha
4.   * iL el numero de inversiones en L
5.   * iR el numero de inversiones en R
6.   * oL la secuencia L pero en forma ordenada
7.   * oR la secuencia R pero en forma ordenada
8.   * LR la unión de L Y R en forma ordenada
9.   * iLR el numero de inversiones de LR
10   */
11.
12.  if A tiene un elemento return 0
13.  else
14.     divide A into L, R
15.     (iL, oL) = sortAndCount(L)
16.     (iR, oR) = sortAndCount(R)
17.     (iLR, oLR) = mergeAndCount(oL,oR)
18.  return iL+iR+iLR, LR
```

Al igual que en el algoritmo Merge Sort vamos a dividir la secuencia, hasta el caso base en que A sea de tamaño 1 y resolverla recursivamente para lo cual nos ayudaremos como vemos en la linea 17 de otra función llamada "mergeAndCount", la cual tendra como pseucódigo:

```
1.  mergeAndCount(oLeft,oRight)
2.    ;  oLeft,oRight dos secuencias ordenadas
3.    ;  result  secuencia unida de oLeft y oRight en forma ordenada
4.    ;  i,j indices de las secuencias
5.    ;  oLeft(i), oRight(j) elementos con indice i, j
6.    ;  count numero of inversiones, inicializa en 0
7.
8.    while oLeft,oRight != empty
9.      append min(oRight(i), oLeft(j)) to result
10.     if oRight(j) < oLeft(i)
11.        count += numero de elementos restantes en oLeft
12.        j++
13.     else
14.        i++
15.   ; ahora si una lista esta vacia
16.   agregar el resto de la lista a result
17.   return count, C
```

Esta función tomara dos listas ordenadas y devolverá la unión de las dos en forma ordenada y el numero de inversiones que hay entre ellas.

Podemos ver como funcionaría toda la secuencia en el siguiente gráfico.



Aquí podemos ver toda la recursión y como se van acumulando los valores y las secuencias ordenadas, **note que SAC se refiere a una llamada a sortAndCount y MAC a una llamada a mergeAndCount** que no aparece en el gráfico pero se asume que se devolvieron los valores esperados para continuar con la recursión.

Implementación del algoritmo en código JavaScript.

```
/* Counting Inversions Algorithm */

function sortAndCount(array) {
  if (array.length < 2)
    return [0, array];

  var mLeft = array.slice(0, array.length / 2);
  var mRight = array.slice(array.length / 2);

  var x = sortAndCount(mLeft);
  var y = sortAndCount(mRight);
  var z = mergeAndCount(x[1], y[1]);

  return [x[0] + y[0] + z[0], z[1]];
}

function mergeAndCount(oLeft, oRight) {
  var count = 0;
  var arrayOrdered = [];

  while (oLeft.length && oRight.length) {
    if (oRight[0] < oLeft[0]) {
      arrayOrdered.push(oRight.shift());
      count += oLeft.length;
    } else {
      arrayOrdered.push(oLeft.shift());
    }
  }

  while (oLeft.length)
      arrayOrdered.push(oLeft.shift());

  while (oRight.length)
      arrayOrdered.push(oRight.shift());

  return [count, arrayOrdered];

}
```

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).

   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.

a)

```
Nitems= sup-inf:
Si Nitems ==2 ENTONCES PICO = A[inf+1]
SINO // CASO GENERAL
    i=2;
    MIENTRAS A[i-1] < A[i] && A[i] < A[i+1] HACER
            i++
    FIN-MIENTRAS
PICO = A[i];
```

En el caso mejor hace 0 comparaciones; el caso peor es cuando el pico está en la posición n-1; el bucle se ejecuta desde i=2, hasta i= n-1, que son n-2 veces, y cada vez hace 2 comparaciones, luego T(n) = 2(n-2) que pertenece a Θ(n).

b) Nos situamos ahora en un punto intermedio: si al comparar con el anterior y posterior la situación es de crecimiento, entonces debemos buscar desde ese punto hacia adelante; si es decrecimiento, entonces debemos buscar desde ese punto hacia atrás; y sino ese es el pico. Como sabemos el mejor punto intermedio es la mitad luego el algoritmo es del tipo PICO (a, inf, sup)

```
Nitems= sup-inf:
Si Nitems ==2 ENTONCES PICO = A[inf+1]
SINO // CASO GENERAL
    Nitems=Nitems/2;
    i= inf + Nitems;
    SI A[i-1] < A[i] && A[i] < A[i+1] ENTONCES //crecimiento
            PICO (A, i+1, sup)
    SINO-SI A[i+1] < A[i] && A[i] < A[i-1] ENTONCES //decrecimiento
            PICO (A, inf, i-1)
    SINO PICO=I // porque no hay valles
```

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

$T(n) = T(n/2) + 4$
Para aplicar el teorema Maestro en versión reducida tenemos que $a=1$, $b=2$, $d=0$, luego $a=b^d$, por tanto

$T(n)$ pertenece a $\Theta(n^0 \log n) = \Theta(\log n)$.

CODIGOS JAVA DE ESTE EJERCICIO:

FUERZA BRUTA

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

DIVIDE Y VENCERÁS

```
    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 {
            // caso general
            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;
    }



    public static void main(String arg[]) {
        int[] v = {1,2,5,6,9,8,6,4,2,1};
        VectorEnteros ve = new VectorEnteros(v);

        System.out.println(ve.picoFuerzaBruta());
        System.out.println(ve.picoDivideYVenceras());

    }
}
```

Ejercicios adicionales

1. Confeccione una función basada en el enfoque de Divide y Vencerás que tome como parámetros dos números naturales *x* y *n* y devuelva x^n. Calcule su complejidad. ¿Es mejor que la del algoritmo de fuerza bruta? Si no lo es, ¿puede modificarse la función para que lo sea?

Como primer paso y para darnos una idea de cómo se resolvería este problema de forma recursiva, presentamos el siguiente algoritmo.

```
función potencia(entero x, entero n)
comienza
     si n = 0 entonces
          regresar 1
     sino
          regresar x * potencia(x, n-1)
     fin si
fin
```

Ahora podemos modificar este algoritmo de forma tal que, encada llamada a la función potencia, el problema se reduzca en un subproblema del mismo tamaño.

Podemos definir a la función x^n de la siguiente forma

xn=xn/2xn/2, si n es par
xn=x(n-1)/2x(n-1)/2x, si n es impar

Entonces, el algoritmo recursivo que utiliza la técnica de divide y vencerás se muestra a continuación.

```
función potencia(entero x, entero n)
     entero y
comienza
     si n = 0 entonces
          regresar 1
     sino si n = 1 entonces
          regresar x
     sino si n mod 2 = 0 entonces
          y ← potencia(x, n/2)
          regresar y*y
     sino
          y ← potencia(x, (n-1)/2)
          regresar y*y*x
     fin si
fin
```

LA LIGA

La siguiente tabla muestra el calendario de una liga de n = 8 equipos:

		Jornada 1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8
	2	1	4	3	6	5	8	7
	3	4	1	2	7	8	5	6
equipos	4	3	2	1	8	7	6	5
	5	6	7	8	1	2	3	4
	6	5	8	7	2	1	4	3
	7	8	5	6	3	4	1	2
	8	7	6	5	4	3	2	1

Nótese como en la esquina superior izquierda nos encontramos con una liguilla entre los equipos 1 : : : 4. El calendario de los equipos 5 : : : 8 durante estas tres jornadas es el mismo que el de los equipos 1 : : : 4 con un desplazamiento de 4. Asimismo, en las últimas tres jornadas el calendario de 1 : : : 4 es el mismo que el de 5 : : : 8 durante los tres primeros días y viceversa.
Finalmente, nótese que en la jornada 4 los equipos 1 : : : 4 juegan contra 5 : : : 8 respectivamente.
Explotando esta estructura defínase un enfoque general de Divide y Vencerás para planificar una liga de $n = 2^k$ equipos.

CuadroJornadas (↓int k, ↑ vector con 2^k filas y $2^k - 1$ columnas){

```
if (k == 1){
        v[1,1]= 2;
        v [2,1] = 1;
}else {
        cuadroJornada (↓ k-1, ↑ v 2^(k-1) filas 2^(k-1) − 1 columnas);
        for (int i = 1; i <= 2^(k-1); i++) {
                v[i, 2^(k-1)] = 2^(k-1) + i ;
                v [2^(k-1) + i, 2^(k-1)] = i;
         }
        for (int i = 1; i<= 2^(k-1);i++){
                for(int j =1; j <= 2^(k-1) − 1;j++){
                        v[2^(k-1) + i, j] = v [i, j] + 2^(k-1);
                        v [i, 2^(k-1) + j] = v [i, j] + 2^(k-1);
                        v [2^(k-1) + i, 2^(k-1) + j] = v [i, j];
                }
        }
}
```

Complejidad para n equipos

$$T(n) = T\left(\frac{n}{2}\right) + \frac{3}{4}n^2 + n \; ; \qquad T(n) \in \theta(n^2)$$

Resolución Divide y Vencerás – Liga de Futbol

	J1	J2	J3	J4	J5	J6	J7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

1			$2^{k-1}-1$	2^{k-1}	$2^{k-1}+1$		2^k-1
2^{k-1}							
$2^{k-1}+1$							
$2^{k-1}+2$							
..							
2^k							

CuadroJornada (K,V con 2^k filas y 2^{k-1} columnas)
//número de equipos = 2^k, nº de jornadas 2^{k-1}
Si k=1 // Caso Base sólo dos equipos
V[1,1] =2; V[2,1]=1;
Else

CuadroJornada (K-1,V con 2^k filas y $2^{k-1}-1$ columnas)
//JornadaCentral ParteSuperior
For (int i=1; i≤2^{k-1}; $i++$)
V[i, 2^{k-1}]= $2^{k-1}+i$;
//JornadaCentral Parte Inferior
For (int i=$2^{k-1}+1$; i≤2^k; $i++$)
V[i, 2^{k-1}]= $i-2^{k-1}$;
//Inferior Izquierda
For (int i=$2^{k-1}+1$; i≤2^k; $i++$)
For (int j=1; j≤$2^{k-1}-1$; $j++$)
V[i,j] =V[i-2^{k-1}, j]+2^{k-1};
//Parte Superior Derecha
For (int i=1; i≤2^{k-1}; $i++$)
For (int j=$2^{k-1}+1$; j≤2^{k-1}; $j++$)
V[i,j] =V[$2^{k-1}+i, j-2^{k-1}$];
//Parte Inferior Derecha
For (int i=$2^{k-1}+1$; i≤2^k; $i++$)
For (int j=$2^{k-1}+1$; j≤2^k-1; $j++$)
V[i,j] =V[$i-2^{k-1}, j-2^{k-1}$];

Complejidad para este problema para N equipos:

T(n) = T($\frac{n}{2}$) + ($\frac{n}{2}$)+ ($\frac{n}{2}$)+ ($\frac{n}{2}$)^2+($\frac{n}{2}$)^2 + ($\frac{n}{2}$)^2
= T($\frac{n}{2}$) +($\frac{3}{4}$)n^2+n
Teorema Maestro; a=1; b=2; d=2
a b^d = 1 < 2^2 = 4 T(N) existe en el orden exacto de Θ(n^2)

SEPTIEMBRE 2016
Supón que se te proporciona una secuencia ordenada A= A1… An de n números enteros distintos, escogidos desde 1 a m, y donde n < m. Implementa (quizás en seudocódigo) un algoritmo de complejidad O(lgn) que permita encontrar el menor entero s< m que no esté presente en A.

SOLUCIÓN:

Pensamos en A como un array con índices de 1 a n, y contenidos de A[1] hasta A[n].
BuscarMenor(A, 1, n)
Ese menor valor s, o está fuera del array o está dentro.
Fuera del array por abajo: Si A[1]>1, entonces el menor es s=1.
Fuera por arriba: Si A[1]=1 y A[n]=n, dentro no hay huecos y el menor es s=n+1.

Estará dentro si A[1]=1 y A[n]>n, entonces llamo a un método BuscarMenorDentro(A, inf, sup) donde tomo medio = (inf+sup)/2 (la primera llamada será con inf=1, sup=n).
Si A[medio]=medio, por debajo no hay hueco, entonces llamo a BuscarMenorDentro(A, medio+1, sup).
Si A[medio]>medio, entonces el hueco está por debajo, y llamo a BuscarMenorDentro(A, inf, medio).

```
static int BuscarMenorDentro(int A[], int inf, int sup)  {
int medio, s=-1;

if (sup-inf==1) {

        if (A[inf] > inf) s=A[inf-1]+1;
        else s=A[inf]+1;

} else {

        medio = (inf+sup)/2;

        if (A[medio]==medio)   {

                s= BuscarMenorDentro(A, medio+1, sup);

        }   else   {

                s= BuscarMenorDentro(A, inf, medio);
        }
}
    return s;
}
```

```
static int BuscarMenor(int A[], int 1, int n, int m)  {
int s=-1;

   if(A[1]>1)   { // el menor que falta está fuera por abajo
          s=1;
  }   else if (A[n]==n)   { // el menor que falta está fuera por arriba
          s= n+1;
   }   else{              // el menor que falta está dentro del array A

          s=  BuscarMenorDentro(A, 1, n)
   }
      return s;
}
```

2. Dado un vector V de n elementos y un número natural k diseñar un algoritmo que transponga los k primeros elementos de V con los elementos de las $n - k$ últimas posiciones, sin hacer uso de un vector auxiliar. Por ejemplo, si V={a,b,c,d,e,f,g,h,i,j} y $k = 3$, entonces el resultado deseado debe ser V={d,e,f,g,h,i,j,a,b,c}.

3. Diseñar un algoritmo de búsqueda "ternaria", que primero compare con el elemento en posición $n/3$ de la lista, si éste es menor que el elemento x a buscar entonces compara con el elemento en posición $2n/3$, y si no coincide con x busca recursivamente en una sublista de tamaño 1/3 de la original. Comparar este algoritmo con el de búsqueda binaria. Definir una expresión recurrente para el número de comparaciones de elementos que se realizan en el peor caso, y encuentre el orden de crecimiento mediante el Teorema Maestro. A continuación, resuelva de manera exacta la anterior recurrencia.

4. En una habitación oscura se tienen dos cajones en uno de los cuales hay n tornillos de varios tamaños y en el otro las correspondientes n tuercas. Es necesario emparejar cada tornillo con su tuerca correspondiente pero no es posible comparar tornillos con tornillos ni tuercas con turcas, 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 $O(n \ lg \ n)$ empareje los tornillos con las tuercas.

5. Diseñar un algoritmo de búsqueda binaria que, en vez de dividir la lista de elementos en dos mitades del mismo tamaño, la divida en dos partes de tamaños 1/3 y 2/3. Comparar este algoritmo con el

DESARROLLOS PARA CLASE

Busqueda Binaria en JAVA

Implementación del algoritmo de búsqueda binaria de manera no recursiva en Java.

Se utiliza una función estática de la clase BusquedaAlgoritmo.

Recordar que para que funcione correctamente los valores del arreglo deben estar ordenados.

```
class BusquedaAlgoritmo {
    public static int buscar( int [] arreglo, int dato) {
 int inicio = 0;
 int fin = arreglo.length - 1;
 int pos;
 while (inicio <= fin) {
     pos = (inicio+fin) / 2;
     if ( arreglo[pos] == dato )
       return pos;
     else if ( arreglo[pos] < dato ) {
  inicio = pos+1;
     } else {
  fin = pos-1;
     }
 }
 return -1;
    }
}

public class BusquedaBinaria {
    public static void  main (String args[]) {
 // Llenar arreglo
 int [] edades = new int [35];
 for (int i = 0; i < edades.length ; i++)
     edades[i] = i*i ;

 // Mostrar arreglo.
 for (int i = 0; i < edades.length ; i++)
     System.out.println ( "edades["+i+"]: "+  edades[i]);

 int resultado = BusquedaAlgoritmo.buscar(edades, 9);

 if (resultado != -1) {
     System.out.println ( "Encontrado en: "+  resultado);
 } else {
     System.out.println ( "El dato no se encuentra en el arreglo, o el
arreglo no estÃ¡ ordenado."  );
 }

    }
}
```

Busqueda Binaria Recursiva en Java

```
import java.util.*;
class programa11
{
 public static void main(String[] args)
 {
  int a[],n,n1,indice,Iabajo,Iarriba;
  Scanner sc=new Scanner(System.in);
  System.out.print("Ingresa tamaño de arreglo: ");
  n=sc.nextInt();
  a=new int[n];
  a=inicializa(n);
  a=ordenarArreglo(a,n);
  muestra(a);
  Iabajo=0;
  Iarriba=n-1;
  System.out.print("Ingresa numero a buscar: ");
  n1=sc.nextInt();
  indice=busquedaBinariaRecursion(a,n1,Iabajo,Iarriba);
  if(indice==-1)   {
   System.out.println("tu número no esta en la lista");
  }   else   {
   System.out.println("tu número esta en el indice: "+indice);
  }
 }
  static int[] inicializa(int n)  {
  int i,j,a[]=new int[n];
  for(i=0;i<n;i++)   {
   a[i]=randomxy(1,50);
  }
   return a;
 }
  static int busquedaBinariaRecursion(int a[],int n,int Iabajo,int Iarriba)  {
  int Icentro,indice=-1;
   if(Iarriba<Iabajo)   {
   return -1;
  }   else   {
   Icentro=(Iabajo+Iarriba)/2;
   if (n<a[Icentro])    {
    return(busquedaBinariaRecursion(a,n,Iabajo,Icentro-1));
   }    else    {
     if (n>a[Icentro])     {
      return(busquedaBinariaRecursion(a,n,Icentro+1,Iarriba));
     }    else     {
      return Icentro+1;
     }
   }
  }
 }
```

```
static int[] ordenarArreglo(int a[], int n)
{
   int i,j,t;

 for(i=0;i<n-1;i++)
 {
  for(j=0;j<n-1;j++)
  {
   if(a[j]>a[j+1])
   {
    t=a[j];
    a[j]=a[j+1];
    a[j+1]=t;
   }
  }
 }
 return a;
}

static void muestra(int a[])
{
 int n=a.length;
 for(int i=0;i<n;i++)
 {
  System.out.print(a[i]+" ");
 }
 System.out.print("nn");
}

static int randomxy(int x,int y)
{
 int ran=(int) (Math.floor(Math.random()*(y-x+1))+x);
 return ran;
}
}
```