Programación II
Práctica 06: Recursividad

Como vimos en clase un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva.

Resolver mediante algoritmos recursivos los siguientes ejercicios

Ejercicio 1)

En vectores

1. Sumar los elementos de un arreglo de enteros.
2. Buscar el mínimo elemento en un arreglo de enteros.
3. Dado un arreglo de enteros devolverlo invertido.
    a) Utilizando un arreglo auxiliar
    b) Sobre el mismo arreglo.

En Strings

1. Devolver la cantidad de apariciones de un carácter c en un String S.
2. Decir si un String tiene un orden lexicográfico, o sea si está ordenado de menor a mayor.

Ejercicio 2)

Escribir un algoritmo recursivo del método toString para una lista de enteros.

Ejercicio 3)

Hacer un programa que devuelva verdadero si dos listas son iguales. Sobrescribir el metodo ***equals*** de Java.

Ejemplo:

{6, 2, 9, 4}, {6, 2, 9, 4} −−> True
{6, 2, 9, 4}, {4, 9, 2, 6} −−> False

Ejercicio 4)

Hacer un programa que devuelva verdadero si una lista de enteros es palíndromo (Se lee igual de izquierda a derecha)
Ejemplo:

{} −−> T rue (7)
{1, 26, 73, 26, 1} −−> T rue
{26, 8, 8, 1} −−> False

Ejercicio 5) Desafío

Agregar un dígito. Dada una lista enlazada de enteros, en la cual cada uno de sus elementos representa un dígito de un número, escribir una función recursiva que le sume un digito.
Ejemplo:

{9, 8, 7} y quiero agregar el 2 −→ {9, 8, 9}
{9, 9, 3} y quiero agregar el 7 −→ {1, 0, 0, 0}

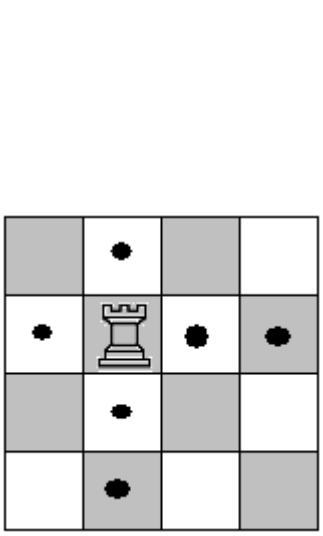
Ejemplos y Ejercicios Adicionales (no son obligatorios)
– Interesantes problemas de la algoritmia que se resuelven con recursividad.

1) El problema de las n torres

El problema de las n torres es una variación del problema de las 8 reinas propuesto por el ajedrecista alemán Max Bezzel en 1848[1].

En el juego del ajedrez la torre amenaza a aquellas piezas que se encuentren en su misma fila o columna (Figura1a).

El juego de las n torres consiste en colocar sobre un tablero de ajedrez ocho torres sin que estas se amenacen entre ellas. (Figura1b).



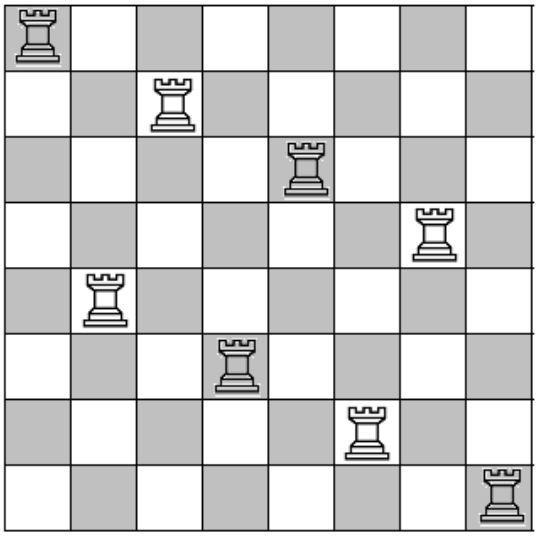


Figura1a: Movimientos posibles de una torre en un tablero de 4x4.

Figura 1b: Una posible solución en un tablero de 8x8

Planteamiento del Problema

[1] http://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas

Como cada torre puede amenazar a todas las torres que estén en la misma fila, cada una ha de situarse en una fila diferente.

Como generar una solución

Podemos generar un vector, donde índice representa una fila y el valor una columna.

Por ejemplo, el vector (**2**,3,4,**2**) significa que:
La torre1 esta en la columna2
La torre2 esta en la columna3
La torre3 esta en la columna4
La torre4 esta en la columna2

Por tanto el vector "solución" para n = 4, correspondería a una permutación de los 4 primeros números enteros.

Como se puede apreciar esta solución es incorrecta (ver Figura2) ya que estarían la torre1 y la 4 en la misma columna.

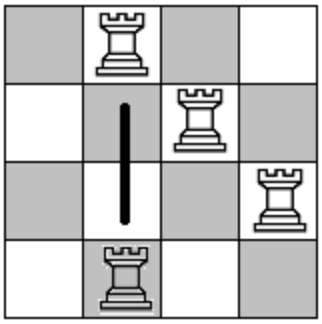


Figura2: Ejemplo de dos torres amenazadas

a) Resolver el problema de las n torres:
Implementar un algoritmo de fuerza bruta iterativo que devuelva todas soluciones.
Comentar hasta que valor de n se pudo ejecutar de manera que el problema tarde menos de 10 minutos.

b) Calcular el **orden asintótico** exacto de 1a) .

Soluciones

1) Solución Iterativa

Esta solución se basa en generar una tabla con todas la combinaciones de números de 1 a n

Ejemplo1: n = 2

Periodicidad		Es solución (Cantidad)
2	1	
Columna1	Columna2	
1	**1**	
1	2	Si
2	1	Si
2	2	
Tamaño $2^2 = 4$ filas		(2! = 2)

Tabla1

Como se puede observar en Tabla1 y Tabla2, la cantidad de combinaciones de n números diferentes en una lista de longitud n resulta de combinar todos los números en todas las posiciones.

En total hay n^n combinaciones posibles.

En el caso de no poder repertir el mismo número, como en el caso de las torres o si queremos saber la cantidad de formas de sentar n personas en n sillas (la persona no se puede repetir en mas de una silla) resulta de permutar los números.

En total hay n! combinaciones posibles.

Ejemplo2: n =3

Periodicidad			Es solución (cantidad)
9	3	1	
Columna1	Columna2	Columna3	
1	**1**	1	
1	**1**	2	
1	**1**	3	
1	2	1	
1	2	2	
1	2	3	si
1	3	1	
1	3	2	si
1	3	3	
2	1	1	
2	1	2	
2	1	3	si
2	2	1	
2	2	2	
2	2	3	

2	3	1	si
2	3	2	
2	3	3	
3	1	1	
3	1	2	si
3	1	3	
3	2	1	si
3	2	2	
3	2	3	
3	3	1	
3	3	2	
3	3	3	
Tamaño 3^3 = 27 filas			(3! = 6)

Tabla2

2) Solución recursiva

Esta idea resulta mas intuitiva ya que el algoritmo refleja las n^n llamadas recursivas

La complejidad esta en O($n*n^n$) porque cuando el algoritmo termina se realizaron n^n llamadas recursivas de funciones, que por separadas son O(n)

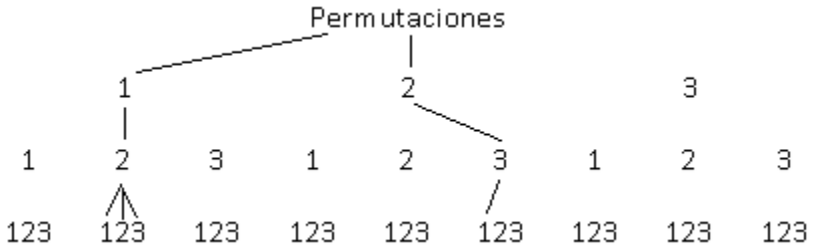
Nota: $n*n^n = n^{n+1}$

3) Solución recursiva con Back Traking2

La idea del Back Traking(o vuelta atrás) es aún más intuitiva.

El objetivo es no mirar los caminos que –se saben- no van a generar ninguna solución.

Una vez que se recortan dichas llamadas, las llamadas restantes coinciden con el Árbol de permutaciones:



Árbol de permutaciones: Se marcan las permutaciones (1,2,1); (1,2,2); (1,2,3); (2,3,1)

La complejidad esta en O($n*n!$) porque cuando el algoritmo termina se realizaron n! llamadas recursivas de funciones, que por separadas son O(n)

Nota: $n*n! = (n+1)!$

[2] http://es.wikipedia.org/wiki/Vuelta_atr%C3%A1s

Código Fuente

```
public class NTorres {

	private int n;
	private int [][] soluciones;
	private int nn; // todas las combinaciones posibles con n
	private int acum = 0;
	private int cont = 1;
	private int cantLlamadas = 0;


	public NTorres(int n){ //invariante: n > 0
		this.n = n;
		this.nn = (int)Math.pow(n, n);
		soluciones =new  int [nn][n];
	}

	public void iterativo(){

		int i;
		int nn = (int)Math.pow(n, n);

		int indice = 0;

		while (indice<n){

			for (i=0;i<nn;i++){
				soluciones [i][indice] = sigiente(indice,i);
			}

			indice++;
			acum = 0;
			cont = 1;

		}

		for (i=0;i<nn;i++){
			if (esSolucion(soluciones[i],n)){
				imprimir(soluciones[i]);
			}
		}


	}
```

```
public void recursivo(boolean bk){
	int [] v = new int [n];

	cantLlamadas= 0;

	if (bk){
		recursivoBK(v,0);
	}else{
		recursivoFB(v,0);
	}

	System.out.println("cantLlamadas:"+cantLlamadas);
}

public void recursivoBK(int [] v, int i){
	int j;

	if (esSolucion(v,n)){
			imprimir(v);

	}else{
		for (j=0;j<n;j++){
			v[i]=j+1;
			if (esSolucion(v,i+1)){
				recursivoBK(v,i+1);
			    cantLlamadas++;
			}

		}
	}

}

public void recursivoFB(int [] v, int i){
	int j;

	if (i==n){
		if (esSolucion(v,n)){
			imprimir(v);
		}

	}else{
		for (j=0;j<n;j++){
			v[i]=j+1;
			recursivoFB(v,i+1);
			cantLlamadas++;
		}
	}

}
```

```
private void imprimir(int [] v1){
		for (int j=0;j<n;j++){
			System.out.print( v1[j]) ;
		}System.out.println("") ;
	}

	public boolean esSolucion(int [] v1, int hasta){
		int [] v2  = new int [n];
		int ret=0;
		int j;

		for (j=0;j<n;j++){
			if (v1[j]-1>=0){
				v2[v1[j]-1]++;
			}
		}

		for (j=0;j<n;j++){
			if (v2[j]>0){
				ret++;
			}
		}
		return ret>=hasta;
	}

	private int sigiente( int indice, int i){
		int paso;
		paso  = (int)(nn / Math.pow(n, indice+1));
		if (acum < paso){
		acum++;
		}else{
		acum = 1;
		cont++;

		if (cont > n){
			cont = 1;
		}
	}
	return cont;
    }

public String toString(){
			String ret = "";
			int nn = (int)Math.pow(n, n);
			for (int j=0;j<nn;j++){
				for (int i=0;i<n;i++){
					ret = ret + soluciones[j][i] ;
				}ret = ret + "\n";
			}
			return ret;
	}
}
```

2) *La función de Achermann*

La función de Ackermann es una función recursiva encontrada por Wilhelm Ackermann en 1926, una función matemática, con un crecimiento extremadamente rápido.

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0; \\ A(m-1,1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m-1, A(m,n-1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Implemente el algoritmo de la función de Ackermann, utilice long en lugar de int.

Probar que: ackermann(3,1) da 13, ackermann(3,2) da 29, ackermann(3,3) da 61, ackerman(3,4) da 125, ackerman(3,5) da 253, ackerman(3,6) da 509, ackerman(3,7) da 1021, ackerman(3,8) da 2045, ackerman(3,9) da `java.lang.StackOverflowError`.

Ayuda para codificar funciones parciales:

La regla: A(m-1),1 si m>0 y n = 0

Se puede escribir como

If (m>0 && n == 0)
A(m-1,1)

3) *Problema del ordenamiento de arreglos - Quick Sort*

Quick Sort funciona particionando el vector que va a ser ordenado Cormen [1990].
Luego, de manera recursiva ordena cada partición.
En Partition (Figura 1), uno de los elementos del vector es elegido como pivote. Los valores menores al pivote son colocados a la izquierda de él, mientras que los mayores son colocados a la derecha.

```
int function Partition (Array A, int Lb, int Ub);
  begin
  select a pivot from A[Lb]...A[Ub];
  reorder A[Lb]...A[Ub] such that:
    all values to the left of the pivot are <= pivot
    all values to the right of the pivot are >= pivot
  return pivot position;
  end;
```

```
procedure QuickSort (Array A, int Lb, int Ub);
  begin
  if Lb < Ub then
    M = Partition (A, Lb, Ub);
    QuickSort (A, Lb, M - 1);
    QuickSort (A, M + 1, Ub);
  end;
```

FIGURA 1: QUICKSORT

En la Figura 2a el pivote seleccionado es 3.
Luego se intercambian los números, de manera de tener los menores a 3 en la parte izquierda, y los mayores en la parte derecha (Figura 2b).
Por último, se ejecuta Quick Sort recursivamente hasta llegar a la Figura 2c.

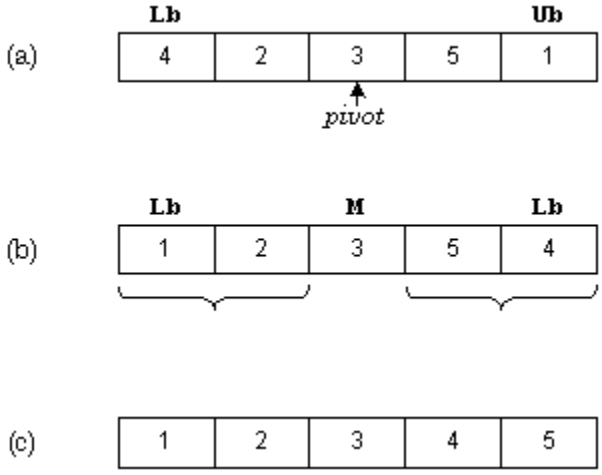


FIGURA 2: EJEMPLO

Para calcular el caso promedio, asumir que el pivote en general estará en el medio.
Para calcular el peor caso, asumir que el pivote en general estará en la punta.

4) Problema del ordenamiento de arreglos - Merge-Sort

Un ejemplo de algoritmo de este tipo es el algoritmo Merge-Sort u Ordenamiento por mezcla.

MergeS-Sort utiliza la técnica de "**divide y conquistaras**", que consiste en resolver instancias mas chicas del problema y luego componerlas para formar una instancia mas grande.

Fue desarrollado en 1945 por John Von Neumann. Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

1. Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:
2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.

3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

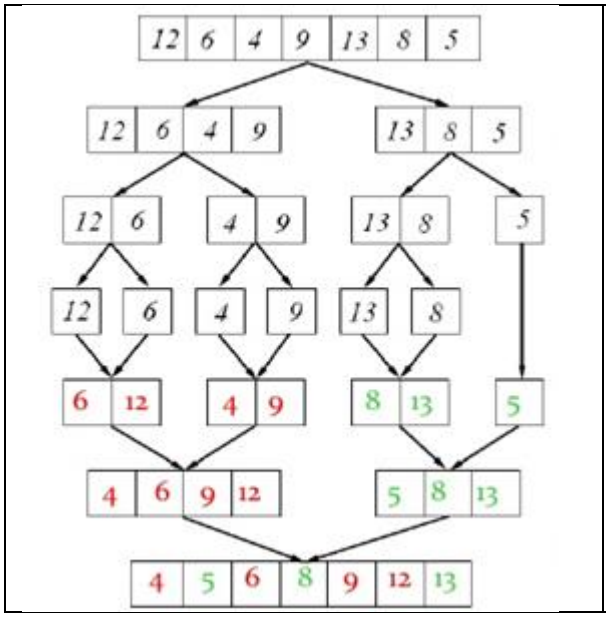
1. Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
2. Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

A continuación se describe el algoritmo en Java:

```
private static int inputCopy[];
private static int input[] = {10,6,4,2,8,0,14,12};

private static void mergeSort(int start, int end) {
	int mid = (start + end) / 2;
	if(start < end){
		/** DIVIDE: Tomar la 1er mitad*/
		mergeSort(start, mid);
		/** DIVIDE: Tomar la 2da mitad*/
		mergeSort(mid+1, end);
		/** CONQUISTAR: Rearmar(merge) el arreglo*/
		merge(start, mid, end);
	}
}
private static void merge(int start, int mid, int end) {
	//Buscamos el comienzo de cada arreglo
	int firstArrStart = start, secondArrStart = mid + 1;
	//Copiamos a una estructura auxiliar antes del merge
	for(int i = start ; i <= end ; i ++)
		inputCopy[i] = input[i];

	while(secondArrStart <= end && firstArrStart <= mid)
		if(inputCopy[firstArrStart] >= inputCopy[secondArrStart])
			input[start++] = inputCopy[secondArrStart++];
		else
			input[start++] = inputCopy[firstArrStart++];

	while(firstArrStart <= mid)
		input[start++] = inputCopy[firstArrStart++];

	while(secondArrStart <= end)
		input[start++] = inputCopy[secondArrStart++];
```

```
}

public static void main(String[] args) {
	System.out.println("INPUT Unsorted : " + Arrays.toString(input));
	inputCopy = new int[input.length];
	mergeSort(0, input.length-1);
	System.out.println("OUTPUT Sorted  : " + Arrays.toString(input));
}
```

Ejemplo de Ordenamiento por Mezcla



Llamada Recursiva 1

Llamada Recursiva 2

Llamada Recursiva 3

Volver a 2

Volver A 1

Combina

Referencias:

http://informatica.utem.cl/~mcast/PROGRAMACION/20101/recursividad/FP.RP04.pdf
http://www.lcc.uma.es/~pepeg/modula/temas/tema11.pdf