Algoritmos y Estructuras de Datos

Ejercicios y Notas de la clase en Nerja

Lars-Åke Fredlund

lfredlund@fi.upm.es

Universidad Politécnica de Madrid

Curso 2021/2022

Recursion: repetición

- Recursión es una tecnica basica en la programacion, que se puede usar en todos los lenguajes de programación (casi)
- Los métodos recursivos en Java tiene:
  - uno o mas casos bases elegido por statements `if` donde termina la recursíon
  - una o mas llamadas recursivas

Recursion: terminación

Cuidado con la non-terminación de llamadas recursivas

- Es obligatorio tener casos bases
- Si tenemos un metodo con una llamada recursiva:

```
type metodo_m(parm : type) {
  if (condicion)    // condicion para entrar en el
                    // caso base

    return ...;     // caso base
  else {            // caso recursivo
    ...
    return metodo_m(value);
  }
}
```

Es obligatorio que el parametro `value` en la llamada recursiva es mas "cercana" a cumplir la condición para entrar en el caso base que su valor en la llamada anterior ("$value < parm$")

Recursion sobre numeros: ejemplo

```
int fact(n : int) {
  if (n==1)                  // condicion para entrar en el caso base
    return 1;                // caso base
  else {                     // caso recursivo
    return n*fact(n-1);      // n-1 < n (es mas cercano al caso base)
  }
}
```

Recursion sobre indices de arrays: receta

- Necesitamos por obligación un método auxiliar que al menos tiene el array y un indice como argumentos
- La condición para entrar en el caso base es: el indice es igual que el tamaño del array (final del array) – y quiza otras
- En cada llamada recursivo el parametro indice tiene que ser mas cerca al final del array (comparado con la llamada anterior)

Recursion sobre indices de arrays: ejemplo

```
int sum(int[] arr) {
  return sum(arr, 0);
}

int sum(int[] arr, int i) {
  if (i == arr.length) {                  // CONDICION para entrar
                                          // en el caso base
    return 0;                             // CASO BASE
  } else {
    return arr[i] + sum(arr, i+1);        // CASO RECURSIVO
                                          // i+1 es ``mas cerca
                                          // al final del array'' que i
  }
}
```

Tail recursion: ejemplo

- Podemos acumular resultados como un parametro extra en el método auxiliar
- El caso base devuelve este parametro

```
int sum(int[] arr) {
  return sum(arr, 0, 0);
}

int sum(int[] arr, int i, int sum) {
  if (i == arr.length) {
    return sum
  } else {
    return sum(arr, i+1, sum+arr[i]);
  }
}
```

Recursion sobre positionlist: receta

- Necesitamos por obligación un método auxiliar que al menos tiene la lista y un cursor como argumentos
- La condición para entrar en el caso base es: el cursor es null (final de la lista) – y quiza otras
- En cada llamada recursivo el parametro cursor tiene que ser mas cerca al final de la lista (comparado con la llamada anterior)

Recursion sobre positionlist: ejemplo

```
int sum(PositionList<Integer>[] list) {
  return sum(arr, list.first());
}

int sum(PositionList<Integer> list, Position<Integer> cursor) {
  if (cursor == null) {                    // CONDICION para entrar
                                           // en el caso base
    return 0;                              // CASO BASE
  } else {
    return cursor.element()                // CASO RECURSIVO
    + sum(list, list.next(cursor));        // list.next(cursor) es
                                           // 'mas cerca al final''
                                           // que cursor
  }
}
```

Ejercicios: normas

- Está **prohibido** usar bucles `for`, `for-each`, `while`, `do-while`, o iteradores.
- Es **obligatorio** usar recursión en la implementación.
- Está **permitido** (y muchas veces **necesario**) añadir métodos auxiliares para implementar correctamente un ejercicio.
- Está **prohibido** añadir nuevos atributos a clases.

Ejercicios sobre recursion con numeros

- `int sum(int i, int j)` – devuelve la suma de todos los numeros entre `i` y `j` inclusivo (sin recursión de cola)
- `int sum(int i, int j)` – devuelve la suma de todos los numeros entre `i` y `j` inclusivo (con recursión de cola)
- `boolean isPrime(int n)` – dado un numero natural n devuelve true si n es un numero primo. Algoritmo naivo: confirma que dado un $a \in (2..\sqrt{n}-1)$ la division n/a siempre tiene resto.
- `String decToHex(int n)` – dado un numero natural devuelve el correspondente numero hexadecimal como un String

Ejercicios con arrays, indexedlists y positionslists

- `int search(E[] list, E elem)` o
  `int search(IndexedList<E> list, E elem)` o
  `Position<E> search(PositionList<E> list, E elem)` – buscar un elemento, devolviendo un indice (o -1) para indexedlists o un array, y una posicion (o null) para un positionlist.
- `int decimalToNat(int[] arr)` – Dado un array `arr` de numeros naturales $0 \leq n \leq 9$ escrito en notacion decimal calcula el numero entero que corresponde. Por ejemplo, si

```
arr = [0,0,1,2,3] decimalToNat(arr) =>
    0*1000+0*1000+1*100+2*10+3 = 123
```

- `int media(int[] arr)` – calcula la media, con aritmetica de enteros. Por ejemplo,

```
media([1,2,3]) = (1+2+3)/3 = 2
```

Ejercicios con arrays, indexedlists y positionslists

En lo de abajo `X x` significa un array `E[]`, o un indexedlist `IndexedList<E> l` o un positionlist `PositionList<E> l`. Prueba con las estructuras que quereis (o con todos!):

- `boolean allNull(X x)` – todos los elementos en `x` son null?
- `int countNonNull(X x)` – cuantos elementos en `x` son null?
- `void duplicateBigNums(PositionList<Integer> list, int n)`

  duplica todos los numeros en `list` que son majores que `n`. `list` no contiene nulls. Por ejemplo:

```
Si list=[1,10,2,6,4,5,10,7,1] y llamamos
    duplicateBigNums(list,6)
despues list=[1,10,10,2,6,4,5,10,10,7,1,1]
```

- `PositionList<Integer> natToDecimal(int n)` – Dado un numero natural n devuelve una positionlist en notacion decimal con los numeros naturales $0 \leq x \leq 9$ que corresponde. Por ejemplo:

```
natToDecimal(0) => [0]
natToDecimal(123) => [1,2,3]
```

Ejercicios con arrays, indexedlists y positionslists

- multiplicación de dos arrays
  `int[] multiply(int[] arr1, int[] arr2)` – devuelve un array nuevo an que donde para cada indice `i` el elemento `an[i]=arr1[i]*arr2[i]`. Lanza una excepcion si los tamaños de los arrays son distintos (prueba con dos indexedlists, y dos positionslists tambien).
- `int[] numPares(int[] arr)` – devuelve un array nuevo que solo contiene los numeros pares en arr, y que *tiene tamaño minimo*
- `int[] natToDecimal(int n)` – Dado un numero natural n devuelve el array en notacion decimal (minimo para hacerlo mas dificil, o no) `arr` de numeros naturales $0 \leq x \leq 9$ que corresponde. Por ejemplo:

```
natToDecimal(0) => [0]
natToDecimal(123) => [1,2,3]
```

- `PositionList<E> reverse(PositionList<E> list)` – invertir un positionlist `list`

Ejercicios con arrays, indexedlists y positionslists

- `insertar(PositionList<E> list, E e, Comparator<E> cmp)` – insertar un elemento en un positionlist ordenada (en listas ordenada)
- `boolean isSubset(PositionList<E> list1, PositionList<E> list2)`

  Dado dos conjuntos (sets) representados como positionlists (no tiene ningun elemento repetido, no contienen nulls) devuelve true si list1 es un subconjunto de list2 (es decir, list2 contiene al menos todos los elementos que estan en list1).
- `boolean equalSets(PositionList<E> list1, PositionList<E> list2)`

  Dado dos conjuntos (sets) representados como positionlists (no tiene ningun elemento repetido, no contienen nulls) devuelve true si list1 y list2 contiene los mismos elementos.

Ejercicios con arrays, indexedlists y positionslists

- Implementa el método

```
static <E extends Comparable<E>>
PositionList<E> merge(PositionList<E> l1,
                      PositionList<E> l2) {
  ....
 }
```

  Por ejemplo: si `l1=[1,2,3,8,9]` y `l2=[2,4,10]` la lista nueva deberia ser `[1,2,2,3,4,8,9,10]`. Las listas nuncan contienen nulls.

- Implementa mergesort (lee sobre mergesort primero, por ejemplo en Wikipedia) – un algoritmo de ordenación muy conocido. Usad vuestro método merge(l1,l2) en la implementación de mergesort.