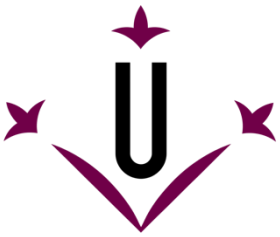

Universitat de Lleida

Escola Politècnica Superior

Grau en Enginyeria Informàtica

Programació II

Pràctica 1

BigNaturals

Aniol Serrano Ortega

GPraLab 2

Índex

Índex de figures

1. Introducció

En aquesta pràctica es tracta de resoldre una sèrie de problemes en forma de funcions. La majoria d'aquestes funcions retornen una sèrie d'enters en forma d'*array* que contenen la solució al problema plantejat. La particularitat d'aquesta pràctica és que la representació dels números està feta de forma inversa a la natural, és a dir, el bit de més pes està a la ultima posició del *array*. Per exemple, el número 593 és representaria pel *array* [3, 9, 5].

L'objectiu principal d'aquesta pràctica és resoldre aquestes funcions d'operacions de forma que no hi hagi un límit determinat pel tipus de variable com pot ser *int* amb 32 bits o *long* amb 64 bits. Per a resoldre aquests problemes, en alguns dels casos s'ha hagut plantejar la funció usant els mètodes tradicionals d'operació. Mitjançant la combinació de funcions prèviament realitzades, és possible implementar operacions més complexes com la multiplicació o el factorial utilitzant la suma i la multiplicació respectivament.

2. Documentació

La documentació detallada de cada funció és important per comprendre millor la pràctica i les solucions als problemes plantejats. D'altra banda, en totes aquestes funcions se suposa que les representacions dels números són correctes. Això vol dir que totes les posicions del vector contenen dígits vàlids i que no conté dígits no significatius.

2.1. Les funcions *Zero* i *One*

Aquestes dues funcions són realment senzilles, però tenen certa utilitat per a ser usades en altres funcions. En la funció zero s'inicialitza un *array* d'enters amb una única posició i valor, en el cas de *Zero* és el 0 i en el cas de *One* és el 1.

2.2. La funció de comparació *equals*

La funció *equals* rep com a paràmetres dos *arrays* d'enters i retorna un booleà segons si aquests dos enters són iguals o no ho són. Aquests dos vectors d'enters són la representació dels seus nombres naturals.

Per a implementar aquesta funció correctament primerament s'ha de verificar primerament si la llargada d'aquests números és la mateixa, ja que al ser números correctament representats, si la llargada d'un és diferent respecte a la de l'altre, per tant, no correspon al mateix número.

Seguidament, es fa un bucle *for* per iterar sobre cada element del vector i en el moment en què es troba amb una posició on els dos números no són iguals, retorna falç. D'altra banda, si recorre els dos vectors i no troba cap diferència entre cap parell d'enters, llavors retorna verdader.

2.3. La suma de dos números

La funció *add* rep dos *arrays* d'enters i ha de retornar la suma. Aquesta funció resulta molt útil per a altres funcions que requereixen matemàtiques bàsiques com la multiplicació, ja que aquesta es pot descompondre en l'operació suma.

La dificultat d'aquesta funció, entre d'altres, ha estat el fet que la suma de dos nombres pot tindre diferents mides. Per exemple, en el cas de 3 + 5, el resultat és 8 i per tant ocupa una posició en l'array. D'altra banda, la suma de 8 + 9 el resultat és 17 i, per tant, el resultat ocupa dues posicions en l'*array*.

Primerament, s'obté la mida del nombre més gran dels dos enters utilitzant la funció *Math.max*. D'aquesta forma es pot inicialitzar la variable *result* amb una posició extra per al carry, en cas que sigui necessari.

Seguidament, s'itera sobre els dos *arrays* per a obtenir el dígit en la posició actual. Un cop es tenen els dos dígits, aquests se sumen i en cas que hi hagi carry aquest se suma. A continuació, s'obté el mòdul 10 del resultat de la suma i s'assigna a la posició

corresponent de l'array result. Un cop sumats tots els dígits, es comprova si hi ha un *carry* sobrant en l'última posició i s'assigna al resultat.

Finalment, es recorre l'array result per eliminar els 0 innecessaris i es retorna l'array resultant. En les següents figures s'exemplifica de forma més esquemàtica aquest algorisme. En aquest cas es verifica que la suma de 72 més 38 és igual a 110.

Num1	2	7
Num2	8	3
Posició	0	1

- Per i = 0

Carry	0	1	
Num1	2	7	
Num2	8	3	
Result	0		
	0	1	3

- Per i = 1

Carry	0	1	1
Num1	2	7	
Num2	8	3	
Result	0	1	
	0	1	3

- Per i = 2

Carry	0	1	1
Num1	2	7	0
Num2	8	3	0
Result	0	1	1
	0	1	3

Figura 1: Representació del funcionament de add

2.4. El desplaçament cap a l'esquerra

La funció *shiftLeft* rep com a paràmetres un *array* d'enters anomenat *number* i un enter anomenat *positions*. Aquesta funció retorna aquest *array* d'enters però els seus enters desplaçats a la dreta tantes posicions com el valor de *positions*, és a dir, s'afegeixen numeros al final de la representació natural del nombre.

Per a veure que fa aquesta funció de forma més esquemàtica, es pot veure en la següent figura la representació del número 485 abans i després d'usar la funció *shitLeft*:

5	8	4	
0	1	2	3

- *Positions* = 2

0	0	5	8	4	
0	1	2	3	4	5

Que fa referència al número: 48500.

Figura 2: Representació del funcionament de shiftLeft

Per resoldre aquesta funció, primer s'ha de verificar si es compleixen les condicions necessàries per aplicar el desplaçament. Això implica comprovar si l'*array* conté només un zero o si no hi ha cap desplaçament perquè la variable *positions* és igual a zero. En cas de no complir aquestes condicions, es retorna el número original.

Seguidament, es crea un *array* anomenat *result* per a emmagatzemar el resultat amb les posicions originals més les ocasionades pel desplaçament. Com que en Java al crear un *array* s'inicialitza tot a zeros, no és necessari afegir els zeros del desplaçament. Finalment, es copia el número original en les posicions desplaçades del vector i es retorna el resultat.

2.5. Multiplicació d'un número per un dígit

En la funció de *multiplyByDigit* es rep una cadena d'enters anomenat *number* i un enter anomenat *dígit*. Aquesta funció retorna el vector que és resultat de multiplicar el nombre de l'*array* pel dígit. El dígit que es passa per paràmetre es garanteix que és un número entre el zero i el nou.

En la multiplicació manual d'un dígit per un altre, primerament es multipliquen els últims dígits i es va recorrent de forma inversa el número. En aquest cas, l'últim dígit és el primer del vector i, per tant, es fa una iteració ascendent sobre el número.

En la implementació es fa servir la funció *add* tantes vegades com el dígit, ja que multiplicar un nombre per un altre és el mateix que afegir-li a un tantes vegades com l'altre. Per exemple, 12 * 3 = 36 és equivalent a fer 12 + 12 + 12 = 36.

Seguidament, el resultat de la suma es guarda en el *array result* inicialitzat en zero. Com que la funció *add* ja retorna un *array* amb la mida adequada, no és modifica el vector resultat, sinó que aquest apunta al nou objecte creat per la funció. Finalment, a l'acabar el bucle es retorna el resultat de l'operació.

2.6. Multiplicació de dos números

En aquesta funció es passen dos *arrays* de enters i s'ha de retornar el resultat de la multiplicació d'aquests dos nombres en forma de vector. El algorisme implementat és basa en la tècnica de multiplicació manual de dos nombres, és a dir, multiplicació columna a columna.

Per a la multiplicació d'aquests dos nombres s'ha de iterar sobre els dos vectors i, de forma similar a la suma, calcular el resultat del producte i afegir-li el *carry*. El *carry* es calcula amb el resultat de dividir el producte per deu i agafar la part sencera. El dígit del producte és el residu de la divisió. Els resultats parcials s'acumulen al vector *partialResult* i on es desplaça i + j posicions a l'esquerra per a cada producte amb la funció *shiftLeft* prèviament implementada.

Finalment, amb el desplaçament provocat pel *shiftLeft* i el resultat parcial, es sumen amb la funció *add* per a retornar el vector resultant.

2.7. El Factorial

En la funció del *factorial* es rep un *array* de enters i s'ha de retornar el factorial d'aquest. Per tal de implementar una funció iterativa per a calcular el factorial d'un nombre donat, primer s'ha de entendre quin és el funcionament del factorial i les funcions que hi estan implicades.

El factorial d'un nombre n, representat com n!, és el producte de tots els nombres naturals des de 1 fins a n. Per exemple, 5! és $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que és igual a 120. Com és evident no és necessari multiplicar per 1, per tant, el bucle es comença al dos. La definició matemàtica del factorial és la següent:

$$n! = \prod_{k=1}^{n} k$$

On n és el numero enter que es vol calcular i k el iterador de cada numero des de 1 fins a n.

Un cop entès quin és el funcionament de l'operació factorial, ja es pot implementar en el context de la pràctica. Primerament, s'avalua el cas en què el nombre sigui igual a un o zero, ja que 0! i 1! es igual a 1 per convenció. Per comparar si el nombre donat és igual a zero o un és fa ús de la funció *equals* i es passen per paràmetres el nombre i *zero* i *one*. Aquesta comprovació és necessària realitzar-la perquè el programa principal no contempla aquestes opcions, per tant, s'han d'implementar prèviament al cos de l'algorisme.

D'altra banda, en el cas en què el nombre donat sigui major a 1, és a dir, 2 o més, es realitza un bucle que incrementa per a cada nombre inferior al nombre donat. Per a incrementar aquest bucle es fa ús de la funció *equals* per a verificar que el factor en el que es trobi sigui diferent al nombre donat, i *add* per a incrementar el bucle.

Per a inicialitzar el factor a 2, es fa ús de la funció *add* i *one* per a generar un vector amb una posició i un enter igual a 2. Aquesta crida '*add(one(), one())*' és equivalent a '*intToArray(2)*. Finalment, es realitza la multiplicació entre el resultat de la volta anterior i el nombre actual amb la funció *multiply* i és guarda el resultat.

En la següent figura es pot veure d'una forma més esquemàtica l'algorisme implementat:

- Per a calcular el factorial de 4:

2	3	4	
0	1	2	3

- currentFactor = 3

6	4	
3 * 2! = 3 * 2 = 6	1	2

- currentFactor = 4

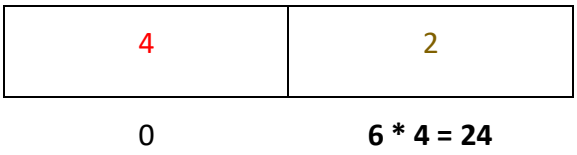

Figura 3: Representació del funcionament del factorial

2.8. El Fibonacci

En la funció del Fibonacci es passa un vector que representa la posició de la successió de Fibonacci. La successió de Fibonacci és una successió de números en la que la posició n d'un nombre està es calcula amb la suma dels dos anteriors, és a dir, n - 1 i n - 2. Es comença la successió de valors per $n_0 = n_1 = 1$. Per tant, la successió dels 12 primers números és:

1, 1, 2, ,3, 5, 8, 13, 21, 34, 55, 89, 144...

Per a la implementació d'aquesta successió de forma similar al factorial, primer s'avaluen els casos desfavorables i després la resta de casos. En el cas que el nombre donat sigui 0 o 1, es retorna 0 o 1 respectivament.

Seguidament, es fa un bucle for fins al nombre donat, però al ser un *array* s'ha fet una funció auxiliar per a representar aquest número com a un enter (*getIntFromArray*). En el cas del Fibonacci és particular perquè aquesta funció retorna un enter, però al ser la posició de la successió, el resultat d'aquest enter creix molt ràpidament i, per tant, aquesta solució ja serveix. Per a posar-ho en context, el resultat la posició 200 de la successió és un nombre aproximat a 2.81 * 10^47.

Per a una futura millora d'aquesta implementació, en comptes d'iterar sobre un enter es podria iterar sobre un vector tal com es fa en el factorial. Per aquesta resolució es requeriria una funció auxiliar que retornes un booleà si el primer nombre és més petit o igual al segon.

Finalment, es guarda la suma dels dos factors anteriors, és a dir, de n - 1 i n - 2 i es preparen les variables per a la següent volta del bucle. Un cop arribat al nombre donat es retorna el resultat del factor actual.

3. Conclusions

En aquesta pràctica s'ha implementat un seguit de funcions sobre operacions matemàtiques amb vectors. La forma de tractar amb els vectors on el bit més significatiu és a la dreta i dígit té una posició, provoca haver de pensar per a una solució més creativa per a resoldre el problema.

El fet de disposar d'una bona previsió de pràctica ha facilitat el fet de poder resoldre els problemes de múltiples formes i trobar aquella que resulta més entenedora (Annex a.). També s'han descartat aquelles funcions que no resultaven útils per a resolució de la pràctica, però sí que han estat el camí per a entendre el problema i trobar una bona solució (Annex b.).

Finalment, aquesta pràctica ha estat molt útil per a construir uns bons fonaments sobre Java i millorar les meves habilitats, especialment en el tractament de vectors.

Annex

a. Versió alternativa de la funció *multiplyByDigit*

En aquesta versió en comptes d'utilitzar les funcions prèviament definides com són la funció *zero()* i *add()*, es resol el problema de forma totalment aïllada de les altres funcions. Aquesta solució funciona correctament, però la seva llegibilitat és clarament pitjor a més que no cohesiona el codi i, per tant, no fa ús del treball prèviament fet.

```
public int[] multiplyByDigit(int[] number, int digit) {
        if ((number.length == 1 && number[0] == 0) || digit == 0) {
            return new int[]{0};
        }

        int carry = 0, product;
        int[] result = new int[number.length + 1];
        for (int i = 0; i < number.length; i++) {
            product = (digit * number[i]) + carry;
            result[i] = product % 10;
            carry = product / 10;
        }

        if (carry > 0) {
            result[number.length] = carry;
            return result;
        }

        int[] shortResult = new int[result.length - 1];
        for (int i = 0; i < shortResult.length; i++) {
            shortResult[i] = result[i];
        }
        return shortResult;
    }
```

En canvi, el codi final ha estat molt més esclaridor, ja que sí que usa les funcions prèviament realitzades i, per tant, el codi resulta més senzill i intuïtiu.

```
public int[] multiplyByDigit(int[] number, int digit) {
        int[] result = zero();
        for (int i = 1; i <= digit; i++) {
            result = add(result, number);
        }
        return result;
    }
```

b. Funcions auxiliars descartades

Aquestes versions són funcions que s'havien implementat primerament però no funcionen per a números molt grans, i per tant, no implementen correctament la solució d'acord amb l'enunciat de la pràctica. Igualment aquestes funcions auxiliars són útils en cas que s'usés per a números que no provoquen desbordament, és a dir, petits.

```
// From int gets the length as if it was in one array
public int getLength(int num) {
        int length = 1;
        while (num >= 10) {
            num /= 10;
            length++;
        }
```

```
        return length;
}
```

Aquesta funció finalment s'ha implementat de forma similar però amb l'estructura adequada per ser usada al fibonacci.

```
// From array returns it in an int
public int getIntFromArray(int[] num) {
        int result = 0;
        for (int i = 0; i < num.length; i++) {
            int currentNum = num[i];
            result += currentNum * Math.pow(10, (num.length - i - 1));
        }
        return result;
}
```

Aquest codi, es trobaba inicialment en la funció *add* però finalment s'ha descartat ja que el codi principal ja contempla la posibilitat que un nombre sigui igual a zero.

```
if (equals(num1, zero())) {
        return num2;
} else if (equals(num2, zero())) {
        return num1;
}
```