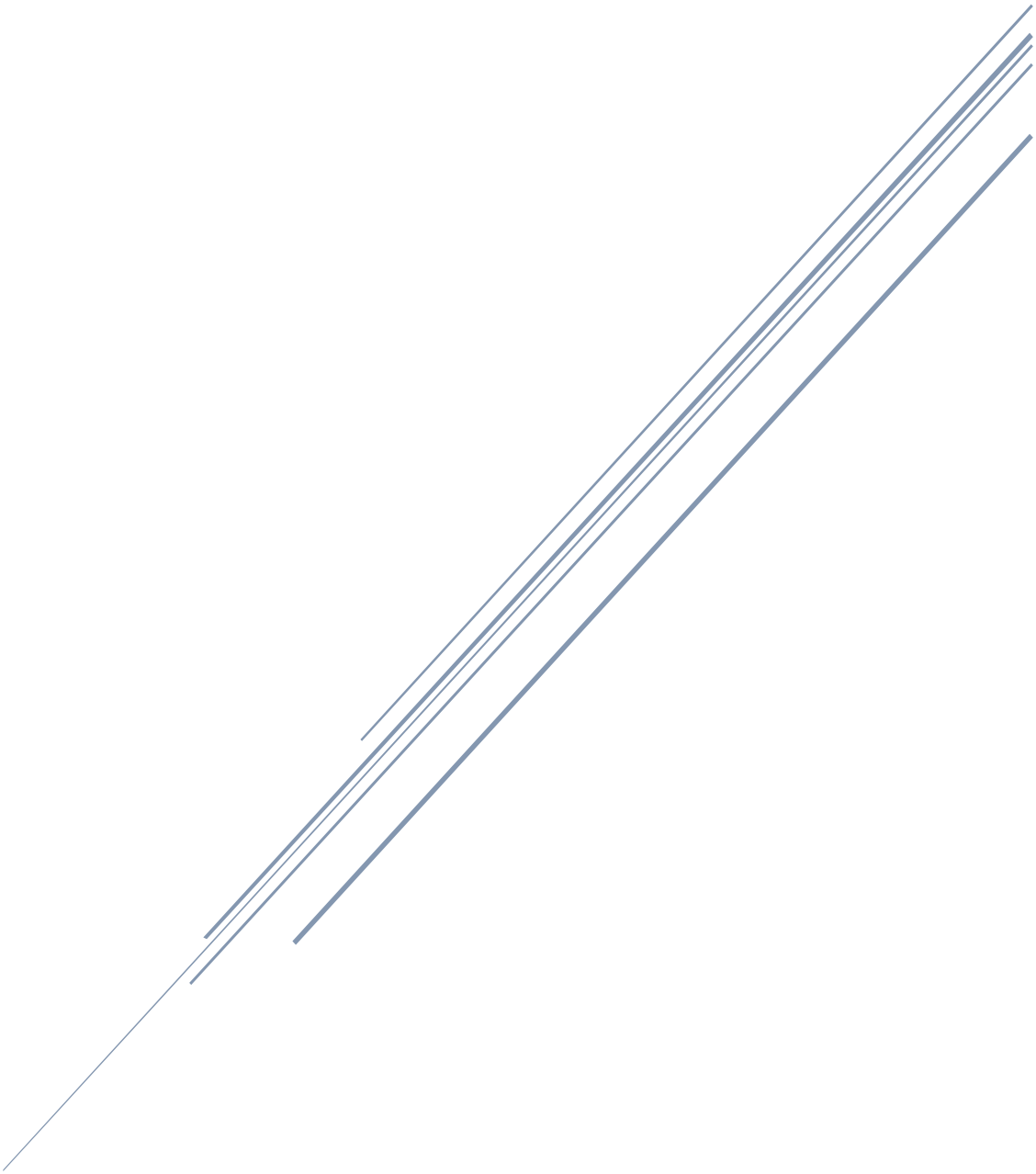
LABORATORI 2

Piles y transformació recursiva a iterativa



EPS UdL
Autors: Èric Bitriá i Adrià Fernández

Índex

1. Definició i implementació de piles

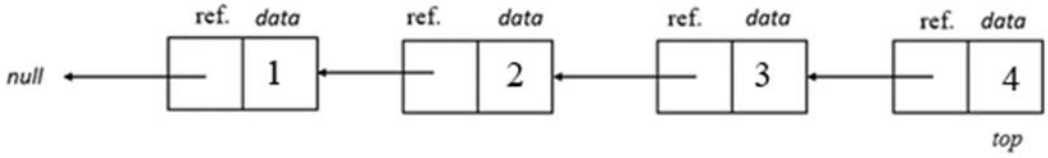
En aquest primer apartat, implementarem la classe LinkedStack<>, aquesta contindrà els mètodes i estructura necessaris per representar el funcionament d'una pila en Java. La pila es basa en el concepte LIFO (Last In First Out), on l'últim element que s'introdueix dins la pila es el primer en sortir, figura [1]. Estenent aquest concepte, qualsevol element que estigui dins la pila no ha de tenir cap altre element per sobre seu per ser accessible.

[1] Representació pila LIFO, on l'últim element introduït es el 4

Una vegada entès el concepte de la pila podem realitzar un raonament previ per implementar totes les funcionalitats necessàries d'aquesta:

1.1 Raonament

La implementació de la pila s'ha de realitzar de manera que cada element referenciï al següent, és a dir, la implementació de la pila, o com d'ara endavant anomenarem *"stack",* s'ha de basar en un sistema de nodes on cada node contindrà el seu element i una referència al següent node, figura [2].



[2] Representació d'un "stack" mitjançant nodes

Evidentment el primer element no pot referenciar a cap element, per tant apuntarà a un *null*. Amb aquest plantejament arribem a la conclusió que treballarem amb dos tipus de dades, els nodes: elements individuals que contindran un element i la referència a un altre node, i el *stack* que serà la representació de tots els nodes junts formant així una pila completa.

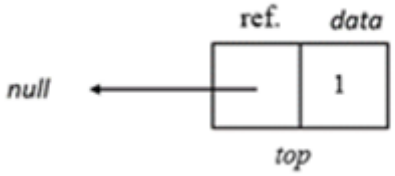
Amb la estructura del *stack* feta hem de crear les operacions que ens permetran manipular i accedir a les dades que conté. Al utilitzar la interfície Stack<> tindrem que implementar 4 mètodes principals:

- **void push(E elem):** Afegirà l'element enviat per paràmetre al *stack.*
- **E top():** Retornarà el primer element del *stack.*
- **void pop:** Eliminarà el primer element que a partir anomenarem "*top*".
- **boolean isEmpty():** Determinarà amb *true* o *false* si el *stack* està buit.

1.1.1 push()

El mètode push haurà d'afegir un nou element al *stack*, Quan afegim un nou element estarem creant un nou node i afegint una referència al anterior, podem deduir que sempre existiran dos casos, quan el *stack* es buit i quan el *stack* ja conté elements. Fixem-nos que passi el que passi sempre haurem de crear un nou node.

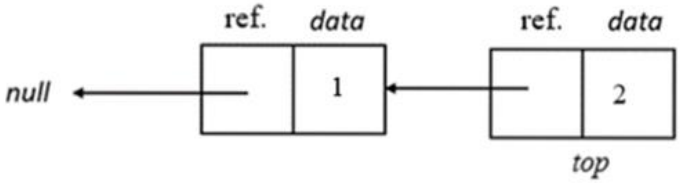
En el cas que el *stack* sigui buit, l'element enviat serà el *top* (el primer) i per tant la referència al següent haurà de ser *null*, figura [3].



[3] Stack d'un sol element

En canvi si el *stack* ja conté almenys un element haurem de fer una sèrie de modificacions al nou node i al que prèviament era el top:

1. El node nou agafarà com a referència el node anterior.
2. Actualitzarem el top per a que sigui el nou node.



[4] Representació del stack després d'un push

1.1.2 top()

El top simplement ha de retornar el primer element del *stack*. Com que el top és el node que tindrem que accedir constantment per la modificació i lectura del *stack* serà imprescindible tenir-lo present sempre com una variable apart de la classe LinkedStack.

1.1.3 pop()

El mètode pop seria la versió contrària al push. Hem d'eliminar el primer element del *stack*, i agafar l'element anterior com a nou top.

També hem de tenir en compte que el *stack* no estigui buit, i només realitzar aquesta operació quan existeixi almenys un element.

Una vegada realitzat el pop() la referència i l'element del node eliminat quedaran sense referencia ja que ningú podrà accedir a elles, seguidament el *"Java garbage collector"* eliminarà aquelles nodes que ja no s'utilitzaran.

1.1.4 isEmpty()

La funció *isEmpty* ha de comprovar que el *stack* estigui buit, una pràctica comú es definir una variable privada que representarà la mida, que podem anomenar *size*.

Simplement si *size* es igual a 0 retornarà *true,* i *false* en cas contrari.

*Nota: Amb la introducció de la variable *size* haurem d'actualitzar-la cada vegada que afegim o eliminem un element del *stack*.

1.2 Implementació LinkedStack

Amb el raonament anterior ens és molt més fàcil implementar tots els requisits, i si a més a més seguim les indicacions del enunciat de la pràctica, podrem començar amb la implementació d'una classe privada i estàtica que representarà els nodes del *stack*.

```
private static class Node<E> {
    private final E element;
    private Node<E> next;
    public Node(E element) {
        this.element = element;
        this.next = null;
    }
}
```

La classe Node<E> contindrà l'element que emmagatzema anomenat element i la referència al següent node next. Solament disposa del constructor de la classe, que guardarà l'objecte rebut a element i la referència al següent node serà *null,* ja que son els mètodes de la classe principal els que modificaran aquesta referència. Una vegada tenim cada element de la *stack* hem d'afegir els mètodes que permeten enllaçar-los.

Com ja hem mencionat prèviament, treballarem amb el últim node de la *stack*. Per això dins de la classe principal, definirem un nou node que el representarà, el qual anomenarem top:

```
private Node<E> top;
```

I com s'ha comentat en el mètode *isEmpty* definim una variable size:

```
private int size;
```

Amb les variables principals definides podem crear el constructor de la classe, el qual inicialitza un *stack* buit:

```
public LinkedStack() {
    top = null;
    size = 0;
}
```

1.2.1 Codi push(E elem)

Sempre que fem un push haurem de crear un nou node, per tant invocarem al constructor de la classe Node. Seguidament realitzarem els dos casos possibles, quan el *stack* esta buit, que aleshores el nou Node serà el top, o quan ja existeix almenys un element, que haurem d'actualitzar la referència del nou element al anterior, és a dir:

```
public void push(E elem) {
    Node<E> newNode = new Node<>(elem);
    if (isEmpty()) {
        top = newNode;
    } else {
        newNode.next = top;
        top = newNode;
    }
    size++;
}
```

Finalment incrementem el size de la *stack*.

1.2.2 Codi pop()

El mètode pop() haurà, primer de tot, comprovar que el *stack* no estigui buit, si resulta que ho està llençarà la excepció No Such Element Exception. Si conté almenys un element, agafarà la referència del següent node i aquesta passarà a ser el nou top, finalment haurem de reduir el size en 1.

```
public void pop() {
    if (isEmpty()) {
        throw new NoSuchElementException("Stack is empty");
    }
    top = top.next;
    size--;
}
```

1.2.3 Codi top()

El top() ens ha de retornar el element que conté el node top, en cas que el stack estigui buit ens llançarà l'excepció No Such Element Exception.

```
public E top() {
    if (isEmpty()) {
        throw new NoSuchElementException("Stack is empty");
    }
    return top.element;
}
```

1.2.4 Codi isEmpty()

Podem comprovar si el *stack* esta buit comparant si *size* es igual a 0:

```
public boolean isEmpty() {
    return size == 0;
}
```

1.2.5 Codi size()

La funció size() tindrà l'únic objectiu de retornar la mida del *stack*, la utilitzarem únicament per als tests de la classe.

```
public int size(){
    return size;
}
```

2. LinkedStackTest

En tota implementació d'una classe, amb mètodes que s'utilitzaran en altres programes, hem d'assegurar-nos que aquesta no conté errors, i el seu funcionament és exactament com s'havia plantejat al raonament. Per aquest motiu Java conté llibreries que permeten crear tests per comprovar funcionalitats d'altres classes.

A l'hora de crear un test s'ha de tenir en compte quina es la condició inicial, quin mètode actuarà i quin serà el resultat. No disposem de molt mètodes en la nostra classe LinckedStack, encara així hauríem de poder comprovar la majoria de les condicions o casos possibles.

2.1 Tests LinkedStack buida

Començarem provant el constructor, en principi ens ha de retornar un stack buit. Ho podem fer comprovant amb la funció isEmpty():

```
@Test
void isEmpty_on_empty_stack_should_return_true(){
    var emptyStack = new LinkedStack<Integer>();
    assertTrue(emptyStack.isEmpty());
}
```

Quan el *stack* es buit els mètodes pop() i top() haurien de retornar la excepció NoSuchElementException:

```
@Test
void pop_on_empty_stack_should_throw_nse_exception() {
    var emptyStack = new LinkedStack<Integer>();
    assertThrows(NoSuchElementException.class, () -> {
        emptyStack.pop();
    });
}
@Test
void top_on_empty_stack_should_throw_nse_exception(){
    var emptyStack = new LinkedStack<Integer>();
    assertThrows(NoSuchElementException.class,() ->{
        emptyStack.top();
    });
}
```

2.2 Tests LinkedList amb un o més elements

Quan treballem amb mètodes que s'encarreguen d'introduir elements, com es d'esperar, no podem provar la infinitat de possibilitats d'elements a introduir, normalment provant casos amb un o dos elements ja representaria les condicions per a qualsevol cas. En els tests següents solament provarem el comportament en *stacks* amb un o dos elements:

```
@Test
void pop_on_stack_with_one_element_should_empty_stack() {
    var oneElementStack = new LinkedStack<Integer>();
    oneElementStack.push(1);     //Add element
    oneElementStack.pop();       //Pop element
    assertTrue(oneElementStack.isEmpty());
}
@Test
void top_stack_one_element_returns_first_element_...(){
    var oneElementStack = new LinkedStack<Integer>();
    oneElementStack.push(1);
    assertEquals(1,oneElementStack.top());
    assertFalse(oneElementStack.isEmpty());
}
```

I finalment amb més d'un element tindríem mètodes com aquest d'entre altres:

```
@Test
void pop_stack_with_one_element_should_return_nse_...(){
    var elementsStack = new LinkedStack<Integer>();
    elementsStack.push(1);
    elementsStack.pop();
    assertThrows(NoSuchElementException.class,() ->{
        elementsStack.pop();
    });
}
```

Existeixen diferents combinacions entre tots els mètodes mencionats, en aquest informe solament se'n recullen els principals a mode d'exemple. L'objectiu es assegurar el correcte funcionament, amb la varietat de casos plantejats podem estar bastant segurs que no tindrem problemes amb la nostra implementació.

3. Explicació Fibonacci Iteratiu

En aquest apartat s'explicarà com amb la implementació d'una pila podem convertir una funció recursiva, en aquest cas Fibonacci, en una funció iterativa.

3.1 Fibonacci Recursiu

Partim de la successió de Fibonacci definida com una seqüència infinita de nombres naturals; a partir del 0 i l'1, on es van sumant a parells, de manera que cada número és igual a la suma dels dos anteriors. La funció per representar aquesta successió (fibonacciOrig), calcularà la suma d'aquesta seqüència un determinat nombre de vegades n de forma recursiva:

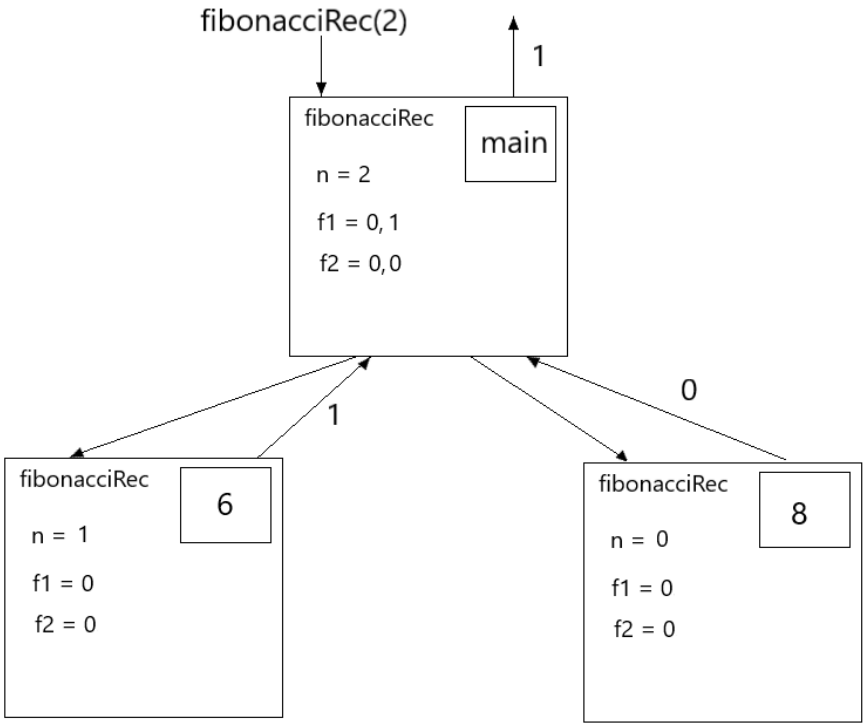
```
public static int fibonacciOrig(int n) {
    if (n <= 1)
        return n;
    else
        return fibonacciRec(n - 1) + fibonacciRec(n - 2);
}
```

Per implementar-ho de forma iterativa ens interessa utilitzar variables que emmagatzemin el resultat de cada crida recursiva per així arribar a un resultat final. Ja que la transformació iterativa no podrà cridar altres funcions amb paràmetres i, per tant, haurem d'emmagatzemar el resultat d'aquestes crides en algun lloc. És a dir:

```
public static int fibonacciRec(int n) {
    if (n <= 1)
        return n;
    else {
        int f1 = fibonacciRec(n - 1);
        int f2 = fibonacciRec(n - 2);
        return f1 + f2;
    }
}
```

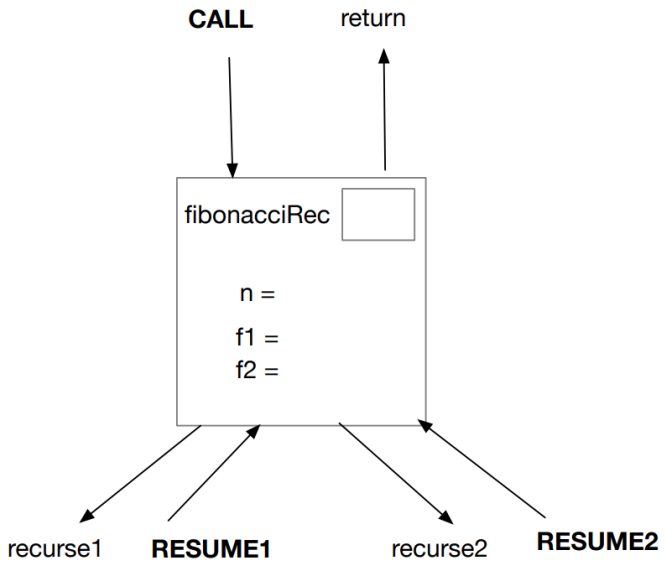
En aquesta funció utilitzem les variables f1 i f2 per guardar els valors de cada crida recursiva, d'on finalment el resultat de Fibonacci serà la suma d'aquestes dues crides f1+f2.

Per cada execució de la funció, en cas que aquesta no sigui un cas simple, realitzarem dues crides recursives, amb un i dos termes menys, respectivament. Podem representar aquesta funció en un diagrama de caixes per observar com evoluciona la funció al llarg de la seva execució:



[5] Representació de fibonacciRec(2)

L'objectiu de la transformació a iteratiu consistirà en representar cada crida recursiva (cada caixa) com un element que puguem manipular en una estructura de *stack*. Per tant cada caixa tindrà una crida inicial, dos casos recursius i dos retorns d'aquells casos recursius. Com que ja no treballarem en un context recursiu podem anomenar cada retorn com la continuació del cas recursiu o "resume":



[6] Representació dels elements de cada crida de Fibonacci

3.2 Identificació d'elements clau

Cada paràmetre d'entrada i sortida de les "caixes" de la funció els podem identificar el propi codi:

```
public static int fibonacciRec(int n) {
    // CALL
    if (n <= 1)
        return n;
    else {
        int f1 = /*recurse1*/ fibonacciRec(n - 1);
        // RESUME1
        int f2 = /*recurse2*/ fibonacciRec(n - 2);
        // RESUME2
        return f1 + f2;
    }
}
```

En la funció fibonacciRec podem identificar els elements prèviament mencionats:

- CALL: Serà la entrada a la crida de la funció.
- Recurse1: Un dels casos recursius, que guardarà el valor de Fibonacci de n-1 a la variable f1.
- RESUME1: Punt de reentrada de la funció una vegada el primer cas recursiu ens ha retornat el seu valor.
- Recurse2: L'altre cas recursiu, que retornarà el valor de Fibonacci de n-2 a la variable f2.
- RESUME2: Reentrada final al finalitzar el segon cas recursiu, d'on posteriorment realitzarem la suma dels resultats dels casos recursius.

3.3 Construcció de la classe Context

Solament ens interessarà identificar els punts d'entrada de la funció: CALL, RESUME1 i RESUME2. D'aquesta manera tindrem els diferents casos i tractaments diferenciats per l'execució del codi. Per representar cada cas utilitzarem un enumerat privat dins de la classe:

```
private enum EntryPoint{
    CALL, RESUME1, RESUME2
}
```

Seguidament haurem d'emmagatzemar cada valor de les variables en el context que ens trobem, és a dir, el valor de f1, f2, el punt d'entrada i el valor de n. Per representar-ho utilitzarem la classe interna privada i estàtica de Fibonacci anomenada Context, que contindrà tots els elements necessaris:

```
private static class Context {
    final int n; // Parameters
    // Local variables
    int f1;
    int f2;
```

```
        EntryPoint entryPoint; // Entry point
        Context(int n) { // Constructor
            this.n = n;
            this.f1 = 0;
            this.f2 = 0;
            this.entryPoint = EntryPoint.CALL;
        }
    }
```

3.4 Construcció de Fibonacci Iteratiu

Ara ja disposem de tot el necessari per començar a convertir la funció de forma iterativa. Com que treballarem amb una estructura de *stack* el primer pas serà inicialitzar un nou *stack* i els elements que contindrà seran instàncies de Context:

```
var stack = new LinckedStack<Context>();
```

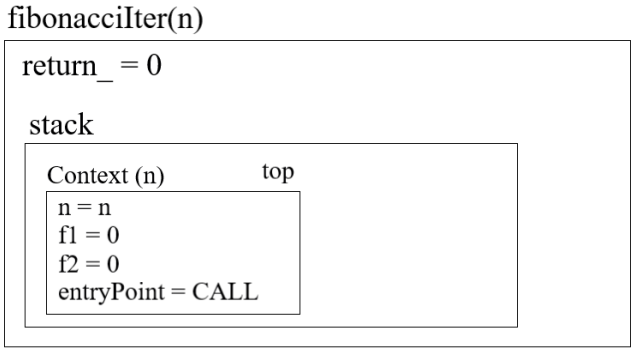
Com hem mencionat anteriorment, tindrem que emmagatzemar el resultat final en una variable, que s'anirà modificant al llarg de l'execució del codi, l'anomenarem return_:

```
int return_ = 0;
```

La nostra funció haurà de processar el resultat donat pel valor n, pel que requeríem fer una primera crida inicial d'on obtindrà aquest valor, és a dir, afegirem una nova instància de context al *stack* amb el primer valor de la funció:

```
stack.push(new Context(n));
```

Podem visualitzar la funció fins aquest punt (inicialització), de la següent forma:



[7] Representació de la inicialització de fibonacciIter

Dins del *stack* de la funció solament existirà el primer element, Context(n), aquest element l'anomenarem el top del *stack*. Fixem-nos que encara no s'ha processat res, estem en el context de crida i per tant el valor del entryPoint es CALL com s'ha definit al constructor de la classe Context.

A partir d'aquest punt hem de realitzar el procés iteratiu, aquest s'executarà sempre que existeixi almenys un element dins del *stack*. És a dir, realitzarem un bucle que comprovi la condició de no estar buit:

```
while (!stack.isEmpty()) { ... }
```

Com és d'esperar, quan haguem sortit del bucle i per tant el *stack* serà buit, haurem de retornar el valor del resultat final:

```
return return_;
```

A partir d'ara ens centrarem en la part principal del codi, el càlcul del resultat en sí, situat dins del bucle while. Tractarà d'afegir i traure contextos, atenent-se als casos del entry-point, fins arribar als casos simples i així anar processant el resultat final.

Amb cada volta que realitzem sempre estarem treballant amb el top del *stack*. Com que haurem de fer modificacions crearem una copia d'aquest element amb la funció top(), la qual ens retornarà l'últim element del stack:

```
var context = stack.top()
```

Al treballar amb Contexts, en funció de la crida en que es trobi el top del stack (el seu entrypoint) tindrem que executar codis diferents. Com que sabem que solament existiran tres casos definits en el enumerat, CALL, RESUME1 y RESUME2, podem utilitzar un switch per identificar cada crida:

```
switch(context.entryPoint){
    case CALL -> {...}
    case RESUME1 -> {...}
    case RESUME2 -> {...}
}
```

3.4.1 Case CALL

Si estem tractant un case CALL hem de decidir si es tracta d'un cas simple o recursiu, això ho podem comprovar amb el valor de `context.n`.

```
if (context.n <= 1)
    //simple
else {
    //recursive
}
```

Si estem en un cas simple solament haurem de retornar el valor de n quan aquest es 0 o 1, aleshores ja haurem acabat amb aquell context i solament requerirem d'un pop per anar buidant la *stack*.

```
return_ = context.n;
stack.pop();
```

De no ser el cas simple, tindrem que recórrer al recursiu. En Fibonacci ens trobem que existeixen dos casos recursius: n-1 i n-2, i en conseqüència dos RESUME.

Al no modificar cap variable hem de convertir el entryPoint CALL en RESUME1, així quan el tornem a trobar sabrem que es tracta de la continuació del primer cas recursiu. Modificarem el entryPoint per a que indiqui RESUME1 com punt d'entrada:

```
context.entryPoint = EntryPoint.RESUME1;
```

A continuació haurem d'actualitzar la pila amb el nou cas recursiu n-1:

```
stack.push(new Context(context.n - 1));
```

3.4.2 Cas RESUME1

Una vegada tornem a la crida RESUME1 i per tant hem sortit del primer cas recursiu haurem de guardar el valor resultant en la variable return_:

```
context.f1 = return_;
```

Haurem d'actualitzar el context actual i que així indiqui que es trobarà la continuació del segon cas recursiu, per això com hem fet amb RESUME1 actualitzem el entryPoint.

```
context.entryPoint = EntryPoint.RESUME2;
```

I al ser nova crida recursiva afegirem un nou Context al *stack*:

```
stack.push(new Context(context.n - 2));
```

3.4.3 Cas RESUME2

Si estem en crida RESUME2, continuació del cas rescuriu2, haurem de calcular la suma de Fibonacci. Primer guardarem el valor resultant return_ a la variable context.f2, que representa el resultat per la crida recursiva n-2.

```
context.f2 = return_;
```

A continuació haurem de realitzar la suma de Fibonacci que no es més que sumar el context f1 + f2:

```
return_ = context.f1 + context.f2;
```

Finalment, el context actual s'elimina de la pila cridant a stack.pop(). Eliminar el context indica que el càlcul d'aquest valor "n" en particular, s'ha completat i que la funció està preparada per seguir processant la resta de Contexts que hi puguin haver.

```
stack.pop();
```

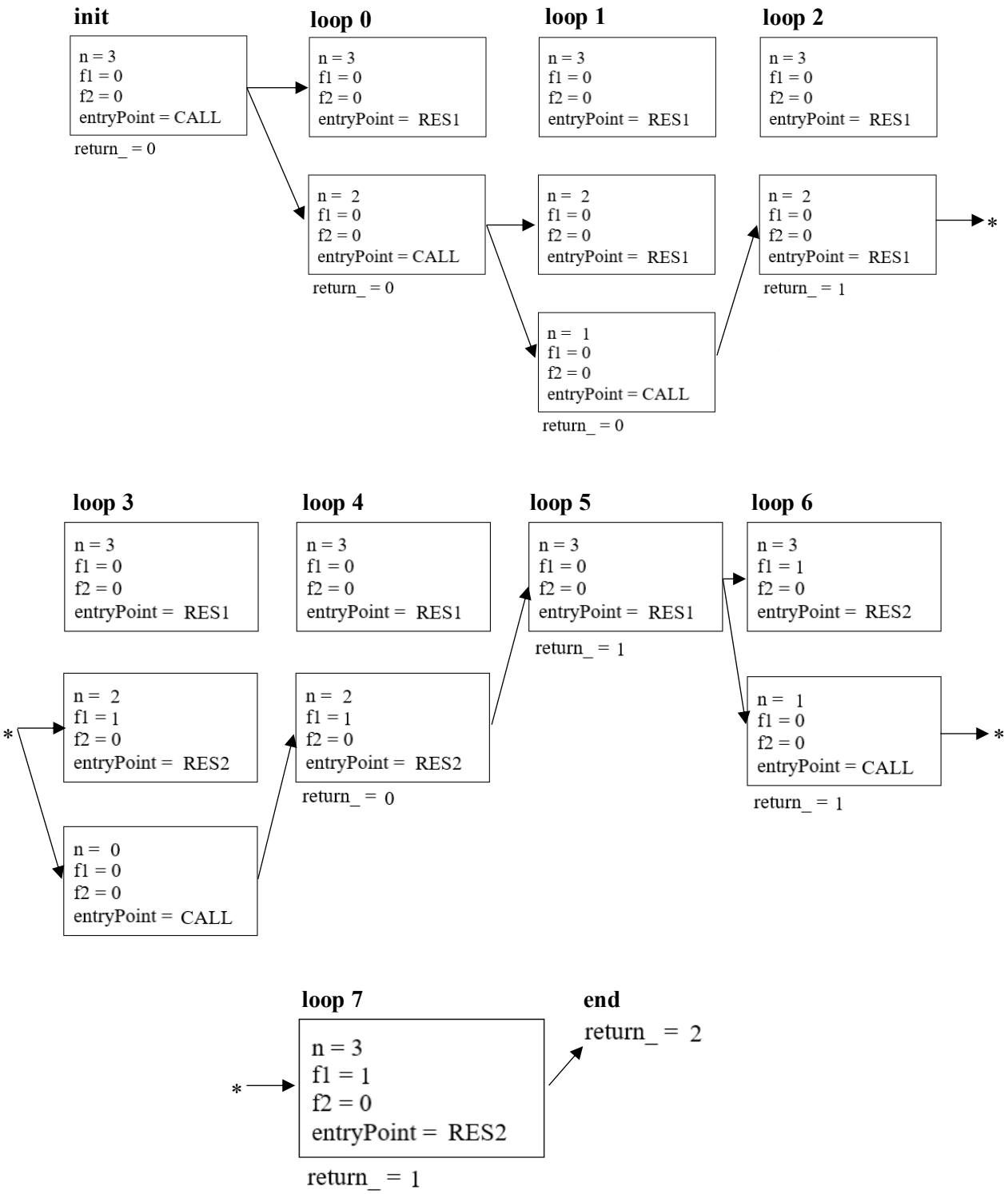
3.5 Codi Final

Unint tots els elements prèviament mencionats ja podem completar la funció fibonacciIter:

```
public static int fibonacciIter(int n) {
    int return_ = 0;
    var stack = new LinkedStack<Context>();
    stack.push(new Context(n));
    while (!stack.isEmpty()) {
        var context = stack.top();
        switch (context.entryPoint) {
            case CALL -> {
                if (context.n <= 1) {
                    return_ = context.n;
                    stack.pop();
                } else {
                    context.entryPoint = EntryPoint.RESUME1;
                    stack.push(new Context(context.n - 1));
                }
            }
            case RESUME1 -> {
                    context.f1 = return_;
                    context.entryPoint = EntryPoint.RESUME2;
                    stack.push(new Context(context.n - 2));
            }
            case RESUME2 -> {
                    context.f2 = return_;
                    return_ = context.f1 + context.f2;
                    stack.pop();
            }
        }
    }
    return return_;
}
```

3.6 Diagrama Fibonacci Iteratiu

En aquests diagrames es representen l'evolució dels contextos del *stack* de FibonacciIter(3) [8].



[8] Diagrama dels contexts dins del stack (de dalt a baix, sent baix el top) per cada volta d'execució de fibonacciIter(3)

4. Explicació Particions Iteratiu

L'objectiu d'aquest apartat es raonar i transformar a forma iterativa la funció recursiva partitionsRec utilitzant la classe LinkedStack prèviament implementada.

4.1 Particions Recursiu

El problema de les particions d'un nombre es basa en descompondre de totes les maneres possibles un nombre, de forma que si es suma cada descomposició obtindrem el número original. En aquest problema no considerarem repeticions ordenades, és a dir, la descomposició de 3 podria ser 2 + 1 o 1 + 2, però per reduir el nombre de descomposicions els tractarem com iguals.

Seguint el raonament explicat en l'enunciat de la pràctica s'arriba a la següent funció recursiva:

```
private static int partitionsRec(int n, int minAddend) {
    // numPartitions where min term is >= minAddend
    assert n > 0 && minAddend > 0 : "parameters … positive";
    if (minAddend > n) {
        return 0;
    } else if (minAddend == n) {
        return 1;
    } else {
        return // Min term is = minAddend
        partitionsRec(n - minAddend, minAddend)
        // Min term is > minAddend
        + partitionsRec(n, minAddend + 1);
    }
}
```

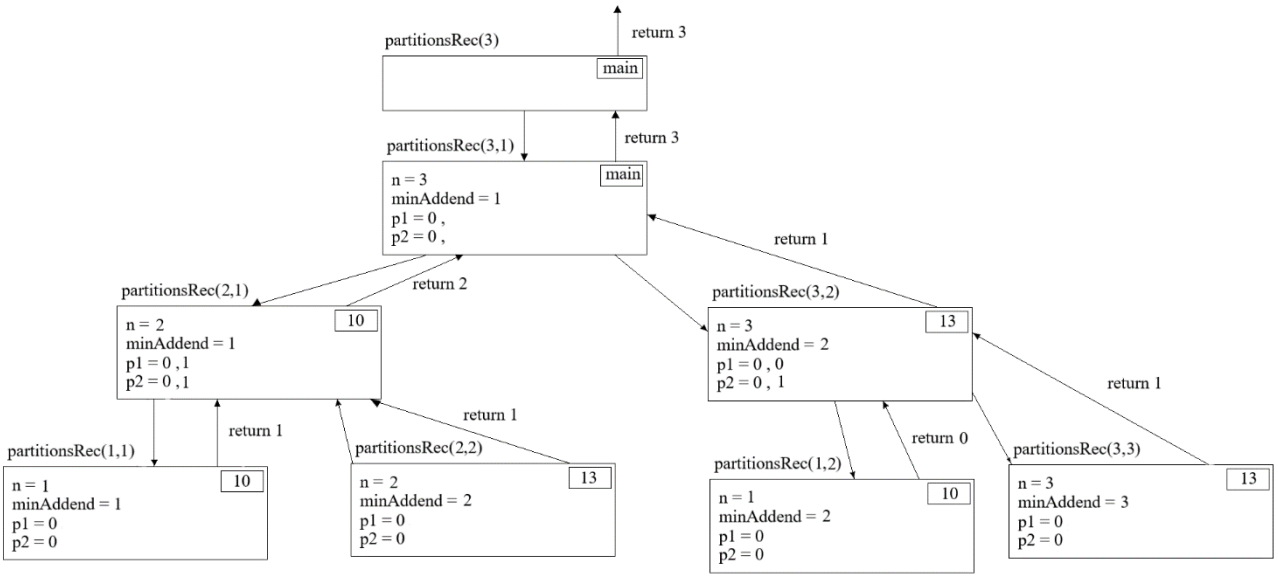
4.2 Identificació elements clau

Com hem vist en la conversió de Fibonacci primer hem de modificar la funció recursiva de forma que: puguem identificar cada cas d'entrada i a continuació, guardar els resultats de cada crida recursiva en variables.

Fixem-nos que aquesta funció presenta dues crides recursives, per tant haurem d'atribuir el seu resultat a dues variables diferents:

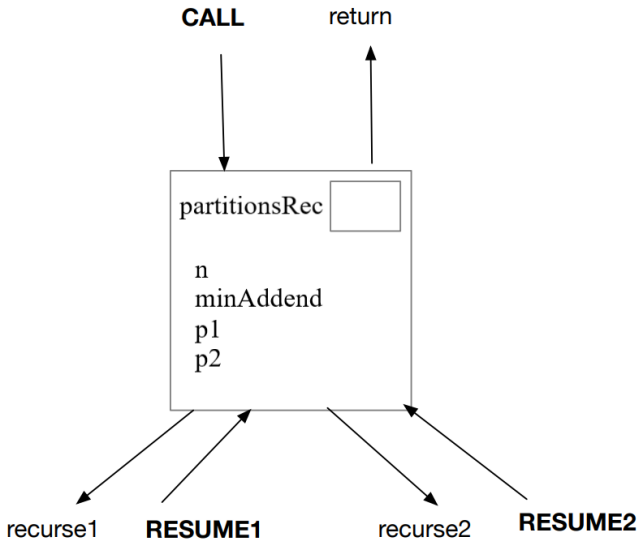
```
//...
else {
    // Min term is = minAddend
    int p1 = partitionsRec(n - minAddend, minAddend);
    // Min term is > minAddend
    int p2 = partitionsRec(n, minAddend + 1);
    return p1 + p2;
}
```

Ara podem representar les crides de la funció partitionsRec en un format de caixes on cada caixa representa un pas recursiu de la funció, en aquest exemple o farem per n=3.



[9] Il·lustració crides recursives de partitionsRec(3)

Aleshores cada crida recursiva es pot dividir en dos casos recursius, dos continuacions i una crida:



[10] Representació Crides de la funció partitionsRec

Ara que hem identificat les crides de la funció ho podem veure en el propi codi: Fixem-nos que presenta dos casos simples i que seran els primers a comprovar-se un cop s'executa. Seguidament ens trobarem els casos recursius, al entrar al primer cas recursiu s'emmagatzemarà el seu resultat a la variable p1, a continuació farà el mateix amb el segon cas i finalment sumarà els dos valors previs. Per tant podem identificar 3 crides:

```
private static int partitionsRec(int n, int minAddend) {
    // numPartitions where min term is >= minAddend
    assert n > 0 && minAddend > 0 : "parameters … positive";
    // CALL
    if (minAddend > n) {
        return 0;
    } else if (minAddend == n) {
        return 1;
    } else {
        // Min term is = minAddend
        int p1 = partitionsRec(n - minAddend, minAddend);
        // RESUME1
        // Min term is > minAddend
        int p2 = partitionsRec(n, minAddend + 1);
        // RESUME2
        return p1 + p2;
    }
}
```

- CALL: Entrada a la funció (crida inicial), comprovació dels casos simples, en cas contrari acudeix als recursius.
- RESUME1: Continuació del cas recursiu 1 una vegada s'ha acabat la seva execució i guardat el seu resultat.
- RESUME2: Continuació del cas recursiu 2 una vegada s'ha acabat la seva execució i guardat el seu resultat. Finalment es sumen p1 + p2 i es retorna el resultat.

4.3 Construcció de la classe Context

Per començar la conversió a iteratiu, haurem de definir quins elements i característiques d'aquests representaran cada fase durant la execució de la funció. Aquests elements els anomenarem Context i contindran els resultats de cada variable p1 i p2, l'element "n" en el que ens trobem i el mínim sumand d'aquest element.

Per representar cada crida dins de la funció utilitzarem un enumerat amb cada cas possible:

```
private enum EntryPoint{
    CALL, RESUME1, RESUME2
}
```

També haurem de guardar els paràmetres que representaran el element a tractar “n”, i el sumand mínim “minAddend”:

```
        // Parameters
        final int n;
        final int minAddend;
```

Finalment guardem les variables de cada cas recursiu:

```
        // Local variables
        int p1;
        int p2;
```

Si ara ajuntem tots aquests elements i afegim un constructor de la classe, que inicialitzarà les variables, ja haurem acabat amb la Classe context:

```
    private static class Context {
        // Parameters
        final int n;
        final int minAddend;
        // Local variables
        int p1;
        int p2;
        // Entry point
        EntryPoint entryPoint;
        // Constructor
        Context(int n, int minAddend) {
            this.n = n;
            this.minAddend = minAddend;
            this.p1 = 0;
            this.p2 = 0;
            this.entryPoint = EntryPoint.CALL;
        }
    }
```

Ara ens queda trobar una forma d’utilitzar els Contexts en una estructura de *stack* per obtenir el nombre de particions del element enviat per paràmetre. És a dir construir la funció.

4.4 Construcció particions Iteratiu

Com a tota funció que vulguem retornar alguna cosa, haurem de crear una variable per emmagatzemar aquest resultat, per tant inicialitzarem return_:

```
    int return_ = 0;
```

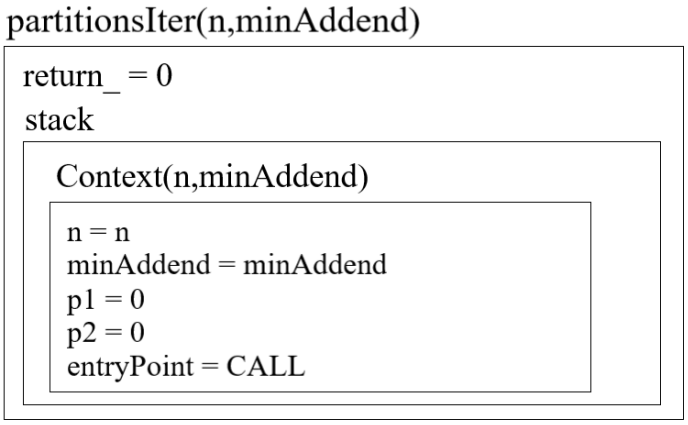
Com que treballarem amb Contexts dins d’una LinkedStack ens en crearem una nova:

```
    var stack = new LinckedStack<Context>();
```

De moment tenim la *stack* buida per tant haurem d'afegir el primer element abans de fer cap processament:

```
stack.push(new Context(n, minAddend));
```

Fins ara tindríem aquesta representació:



[11] Representació Inicialització de partitionIter

Executarem el processament menters el *stack* no sigui buit, podem complir aquesta condició realitzant un *while*:

```
while (!stack.isEmpty()){...}
```

Si sortim del *while* significarà que el *stack* es buit i haurem obtingut el resultat final, només ens quedaria retornar-lo.

```
return return_;
```

Ara "solament" ens queda processar els diferents casos en funció del entryPoint que ens trobem, com que estarem treballant contínuament amb el primer element del *stack*, aquest tindrà que ser el primer pas: obtenir el top del *stack*:

```
var context = stack.top();
```

I en funció de quin estat es el seu entryPoint executar el codi necessari. En el nostre cas utilitzarem un switch on cada condició seran els elements del enumerat privat que hem definit anteriorment:

```
switch(context.entryPoint){
    case CALL -> {...}
    case RESUME1 -> {...}
    case RESUME2 -> {...}
}
```

4.4.1 Case CALL

Com hem definit anteriorment el cas CALL és la crida inicial i contindrà la comprovació dels casos simples. En tindrem dos, quan els paràmetres són iguals (n = minAddend), és a dir, el nombre de formes de sumar n amb termes que siguin el propi n, solament existeix una única forma i per tant haurem de retornar 1:

```
if(context.n == context.minAddend){
    return_ = 1;
    //...
} //...
```

El segon cas simple es quan minAddend és major a n, el que representa sumar n amb termes on el mínim del valor és més gran que el propi n, de manera que el resultat en aquest cas és 0.

```
else if(context.minAddend > context.n ){
    return_ = 0;
    //...
} //...
```

En tots dos casos ja haurem acabat de processar el resultat i solament ens quedaria eliminar aquell element:

```
stack.pop();
```

Si no es compleixen cap de les dues condicions haurem d'acudir al primer cas recursiu. Per això actualitzarem el entryPoint del context actual a RESUME1 així quan el tornem a trobar sabrem que es tracta d'aquest cas:

```
Context.entryPoint = EntryPoint.RESUME1;
```

Finalment haurem de fer la crida recursiva i afegir un nou element modificant els paràmetres d'entrada.

El primer cas recursiu seran les descomposicions en les que el menor terme que les forma es minAddend, i per tant tindrem que sumar n - minAddend amb termes on el menor nombre sigui minAddend, expressat en la nostra funció seria afegir un nou Context amb aquestes noves modificacions als paràmetres:

```
stack.push(
    new Context(context.n - context.minAddend, context.minAddend)
);
```

4.4.2 Case RESUME1

Si ens trobem en el cas RESUME1, tindrem que retornar el valor d'aquesta primera crida recursiva, simplement actualitzant la variable del context:

```
context.p1 = return_;
```

Com que hem sortit del primer cas recursiu ara tindrem que accedir al següent, per tant primer actualitzarem el entryPoint per saber en quina crida ens trobem en les posteriors iteracions del stack:

```
Context.entryPoint = EntryPoint.RESUME2;
```

Finalment tindrem que representar la segona crida recursiva que correspon a les descomposicions on tots els termes son més grans o igual a minAddend + 1. Per tant requerim d'afegir un nou Context actualitzant el paràmetre minAddend:

```
stack.push( new Context(context.n, context.minAddend + 1));
```

4.4.3 Case RESUME2

El case RESUME2 correspon a la finalització del segon cas recursiu, per tant haurem de guardar el seu resultat en una variable:

```
context.p2 = return_;
```

I com que ja haurem acabat tots els casos recursius solament en faltarà sumar el resultat final:

```
return_ = context.p1 + context.p2;
```

Al ja estar processat el resultat, solament ens queda eliminar aquest context amb un pop:

```
stack.pop();
```

4.5 Codi Final

Doncs ja estaria, seguint la metodologia de conversió a iteratiu hem pogut convertir una funció recursiva en la seva forma iterativa gràcies a un estructura de *stack*. Si ara juntem totes les parts del codi prèviament mencionat obtindríem el resultat final:

```
private static int partitionsIter(int n, int minAddend){
    assert n > 0 && minAddend > 0;
    int return_ = 0;
    var stack = new LinckedStack<Context>();
    stack.push(new Context(n, minAddend));
    while (!stack.isEmpty()){
        var context = stack.top();
        switch(context.entryPoint){
            case CALL -> {
                if(context.n == context.minAddend){
                    return_ = 1;
                    stack.pop();
                } else if(context.minAddend > context.n ){
                    return_ = 0;
                    stack.pop();
                } else {
                    context.entryPoint = EntryPoint.RESUME1;
                    stack.push(new Context(...));
                }
            }
```

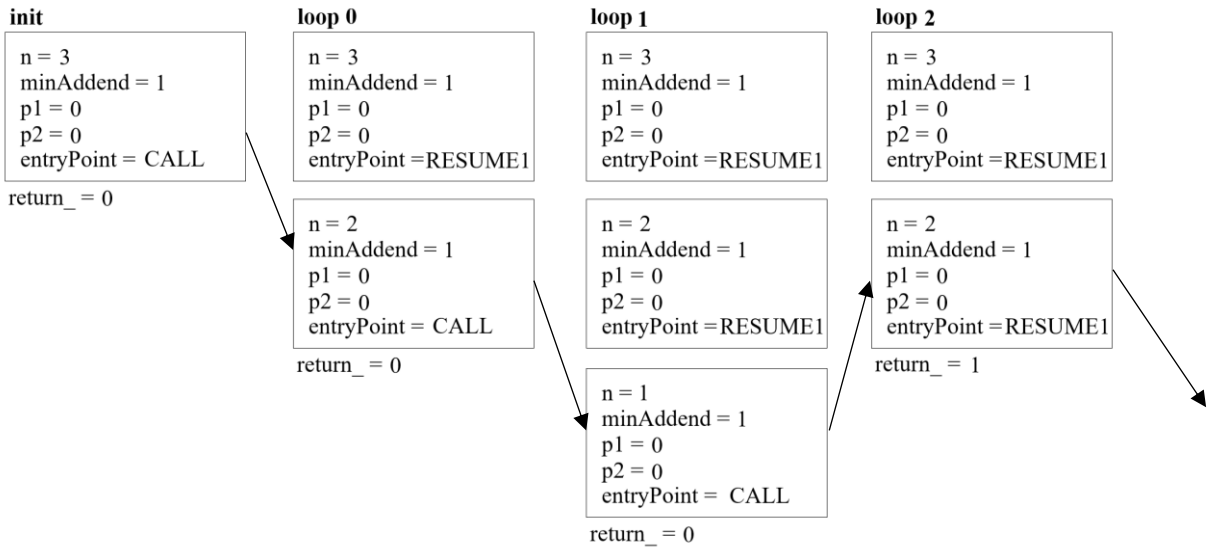
```
                case RESUME1 -> {
                    context.p1 = return_;
                    context.entryPoint = EntryPoint.RESUME2;
                    stack.push( new Context(…));
                }
                case RESUME2 -> {
                    context.p2 = return_;
                    return_ = context.p1 + context.p2;
                    stack.pop();
                }
            }
        }
        return return_;
}
```

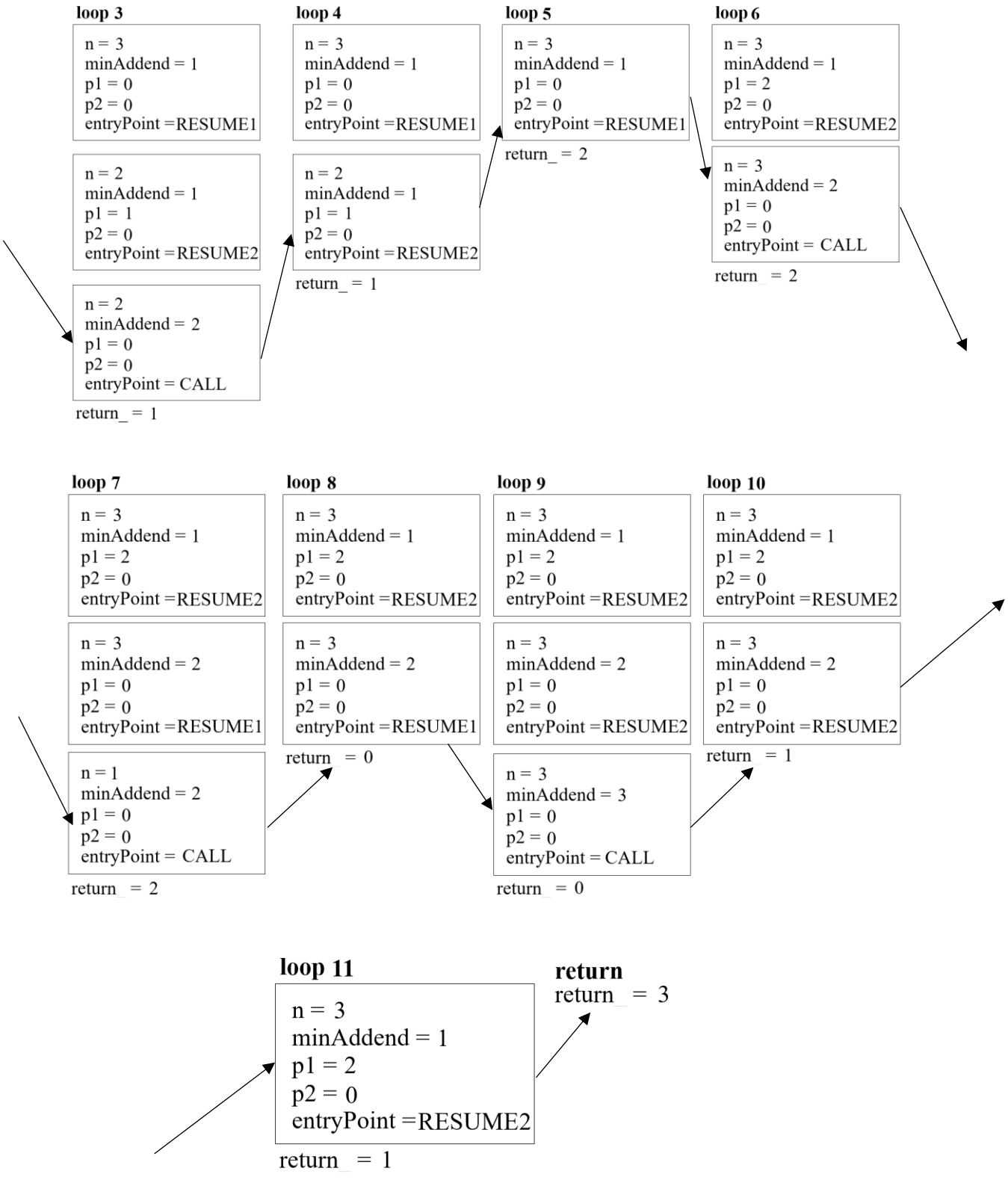
Hem pogut observar com seguint el raonament plantejat en el enunciat s'ha convertit una funció recursiva a iterativa, és més aquest codi l'hem implementat tot just acabat de realitzar aquest apartat del informe i passant els tests a la primera.

Això demostra que si seguim un raonament ordenat, on cada pas tingui una justificació més el fet d'entendre què estem fent arribarem a la solució de la forma correcta.

A continuació per completar l'explicació de les particions d'un nombre natural il·lustrarem com la funció iterativa treballa amb el *stack* en cada volta del while, ho farem per a partitionsIter(3).

4.6 Diagrama Particions Iteratiu





[12] Diagrama de l'evolució del stack i els seus elements Context al llarg de l'execució de partitrionsIter

5. Conclusions

Al llarg del desenvolupament d'aquesta pràctica s'ha tingut molt en compte el plantejament a l'hora de solucionar els problemes plantejats. Ser capaços de visualitzar ja sigui amb una imatge, fent un llistat dels requeriments que ha de complir una funció o un petit esbós ens ajuda a comprendre millor quina metodologia i passos hem d'aplicar per arribar a la solució final.

Aquesta metodologia juntament amb la presentada al enunciat de la pràctica, ha ajudat a convertir un mètode recursiu, que a primera vista pot semblar complicat, a simplement un seguit d'instruccions individuals i fàcils d'executar gràcies a una descomposició molt metòdica. Ser capaços d'abordar un problema gran en funció d'altres de més petits és dels majors avantatges que podem tenir a l'hora de programar.