Arnaud Strapart
Basé sur le cours LEPL1402

Cours Magistral 3 : Complexity 2a

Complexité temporelle = temps pour un algorithme pour faire qqch en fonction de la taille de l'entrée => elle ne dépend pas de l'ordinateur et du langage (ex : C et Java)

➔ Modèle RAM = Random Accces Machine

Pour un programme basique, la complexité est globalement linéaire (compare temps en fonction de l'entrée) (ex : addition de tous les éléments dans une liste)

Opération simple = 1 pas de temps (dans le temps réel, il va dépendre de l'ordinateur)
Accès à la mémoire = 1 pas de temps
Dans boucle : multiplier le nombre de fois que la boucle est répétée
!! En règle générale, une multiplication plus rapide que division => MAIS en général, les résultats en RAM obtenus sont proches de la réalité

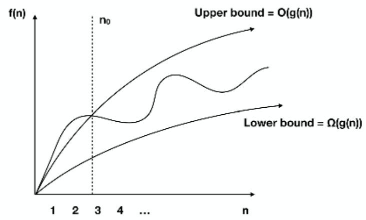
La classe Big Oh permet de simplifier les notations :

- upper-bound: $f(n) \in \mathcal{O}(g(n))$ means there exits some constant c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

$$f(n) \in \mathcal{O}(g(n)) \iff \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : f(n) \leq c \cdot g(n)\ \forall n \geq n_0$$

Ex :
- $3n^2 - 100n + 6 \in \mathcal{O}(n^2)$ as rule is respected for $c = 3, n_0 = 0$:
  $3n^2 \geq 3n^2 - 100n + 6\ \forall n \geq 0$
- $3n^2 - 100n + 6 \notin \mathcal{O}(n)$ as its impossible to find c, n_0 such that
  $c \cdot n \leq 3n^2 - 100n + 6\ \forall n \geq n_0$

Cad si on multiplie la fonction par une certaine constante suffisamment grande, elle sera, à partir d'un certain temps, toujours plus grande que la fonction f



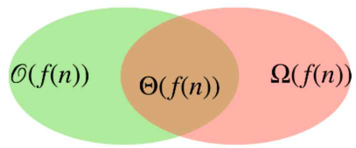
- lower-bound: $f(n) \in \Omega(g(n))$ means there exits some constant c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$

$$f(n) \in \Omega(g(n)) \iff \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : f(n) \geq c \cdot g(n)\ \forall n \geq n_0$$

Ex :
- $3n^2 - 100n + 6 \in \Omega(n)$ as $3n^2 - 100n + 6 \geq n\ \forall n \geq 34$
- $3n^2 - 100n + 6 \in \Omega(n^2)$ as $3n^2 - 100n + 6 \geq n^2\ \forall n \geq 50$
- $3n^2 - 100n + 6 \notin \Omega(n^3)$

Cad si on multiplie la fonction par une certaine constante suffisamment grande, elle sera, à partir d'un certain temps, toujours plus petite que la fonction f

- tight-bound: $f(n) \in \Theta(g(n))$ means there exits some constant c_1, c_2 and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$



$$f(n) \in \Theta(g(n)) \iff \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \forall n \geq n_0$$

Les 2 d'un coup => borne supérieur et inférieur en même temps = intersection des 2 premières

Ex :
- $3n^2 - 100n + 6 \in \Theta(n^2)$ because function is both $\in \mathcal{O}(n^2)$ and $\in \Omega(n^2)$
- $3n^2 - 100n + 6 \notin \Theta(n^3)$ because function is $\notin \Omega(n^3)$

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

!! Le moment n peut être très grand mais dans la réalité il est petit.

For example

$f(n) = c \cdot n^a + d \cdot n^b$ with $a \geq b \geq 0$ and $c, d \geq 0$

We have that

$f(n) \in \Theta(n^a)$

Even if c is very small and d very big!

Arnaud Strapart
Basé sur le cours LEPL1402

Simplifying Big Oh functions

$\mathcal{O}(c \cdot f(n)) = \mathcal{O}(f(n))$ (For a positive c)

$\mathcal{O}(f(n) + g(n)) \subseteq \mathcal{O}(\max(f(n), g(n)))$ Not true for multiplication, of course (what is the rule?)

We can thus simplify this way:

$$\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(c^n) \subseteq \mathcal{O}(n!)$$

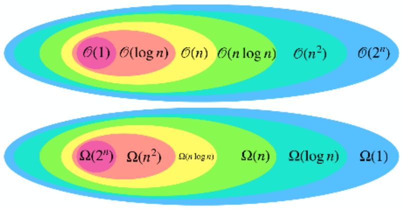
Usually, you must simplify if possible:

$$f(n) = n^4 - 10n^3 + 20n^2 + 8 \iff f(n) \in \mathcal{O}(n^4 - 10n^3 + 20n^2 + 8)$$

Is true, but $f(n) \in \mathcal{O}(n^4)$ **is more concise**

!! Il y a quelques règles pour pouvoir comparer les algorithmes

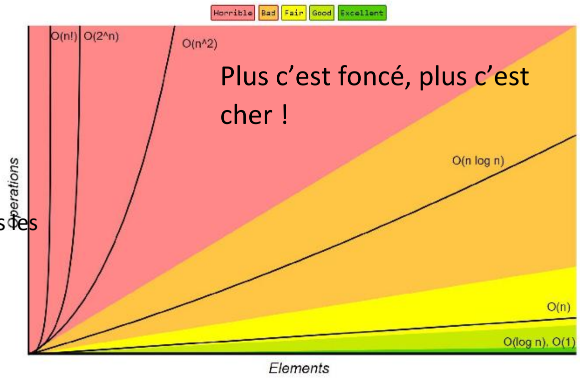
Les O et Ω sont des ensembles (sets)



Most frequent functions when counting the steps

$\mathcal{O}(1)$: constant	Sum of two integers
$\mathcal{O}(\log n)$: logarithmic	Finding a word in the dictionary
$\mathcal{O}(n)$: linear	Sum of an array
$\mathcal{O}(n \log n)$: linearithmic	Sorting (efficiently) = Tri efficace
$\mathcal{O}(n^2)$: quadratic	Sorting (inefficiently) = Tri inefficace cad teste toutes les possibilités
$\mathcal{O}(n^3)$: cubic	Enumerating triples
$\mathcal{O}(c^n)$: exponential (with $c > 1$)	Subset sum
$\mathcal{O}(n!)$: factorial	Finding longest path in a graph

More and more costly



Plus c'est foncé, plus c'est cher !

Qd on dit : gain de 2 = on divise le temps par 2

Quand on cherche une complexité :
. Constant si pas de boucles ect mais peut y avoir des if
. Linéaire si opération puis boucle x opération puis opération
. Quadratique (boucle dans boucle)
. Plus que quadratique (boucle dans boucle dans boucle ...)

Pour les algorithmes de recherche séquentiel cad on regarde chaque éléments à la suite => linéaire
O(n) => pire cas est le dernier de la liste
Ω(1) => meilleur cas est le premier de la liste
=> Mieux = recherche dichotomique
Pour les algorithmes de recherche dichotomique / binaire cad on regarde le milieu de la liste etc
Si n est puissance de 2 alors $\log_2(n)$ car on divise à chaque fois en 2
$O(\log_2(n))$ => pire des cas
Ω(1) => meilleur cas

Implémentation de l'algorithme de recherche dichotomique / binaire :

```java
private static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

Executed O(log(n)) times

```java
private static int binarySearchRecur(int[] a, int key) {
    return binarySearchRecur(a,key,0,a.length-1);
}

private static int binarySearchRecur(int[] a, int key, int low, int high) {
    if (low <= high) {
        int mid = low + (high - low) / 2;
        int midVal = a[mid];
        if (midVal < key)
            return binarySearchRecur(a,key,mid+1,high);
        else if (midVal > key)
            return binarySearchRecur(a,key,low,mid-1);
        else
            return mid; // key found
    } else return -(low + 1); // key not found.
}
```

=> Niveau complexité, les 2 sont équivalents

Arnaud Strapart
Basé sur le cours LEPL1402

Mais on peut toujours essayer de faire mieux en passant par des recherches dichotomiques

```java
public static void triplesBetter(int obj, int[] values) {
  int n = values.length;
  for(int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      if(binarySearch(obj - values[i] - values[j], values) >= 0)
        System.out.println(values[i] + " " + values[j] + " " + (obj - values[i] - values[j]));
    }
  }
}
```

First solution in

$$\mathcal{O}(n^3), \Omega(n^3), \Theta(n^3)$$

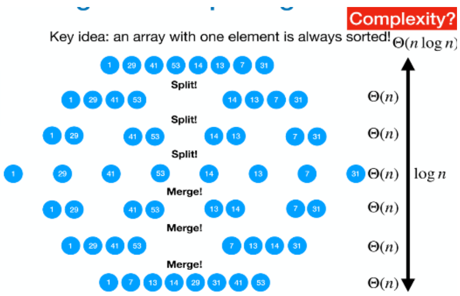
Now we can do better!

$$\mathcal{O}(n^2 \log n), \Omega(n^2)$$

<u>Merge Sort</u> = algorithme qui trie des éléments récursivement en séparant les tableaux en 2 et avec des fusions (tri fusions)

On regarde le premier élément des 2 tableaux et on insère le plus petit dans la liste. Puis on regarde les nouveaux premiers éléments (l'ancien du tableau non utilisé et le nouveau de l'autre tableau) et on refait.

=> Complexité linéaire : O(n)



Merge Sort : split en plein de tableau de taille 1 (ils sont donc tous trier puisque il y a que 1 élément)
Ensuite on rassemble en des listes trier de 2 éléments, puis de 3 puis de 4 etc ... Jusqu'à n'avoir qu'une seule liste

Complexité O(nlog(n))

Méthode d'implémentation :

```java
public static void mergeSortBetter(int from, int to, int[] values) {
  if(from + 1 == to) //size 1 is always sorted
    return;
  int mid = (from+to)/2;
  mergeSortBetter(from, mid, values);
  mergeSortBetter(mid, to, values);

  //create a temp array
  int[] tmp = new int[to-from];
  int tmpIdx = 0;
  int aIdx = from;
  int bIdx = mid;
  while (aIdx != mid || bIdx != to) {
    if(bIdx == to || (aIdx != mid && values[aIdx] < values[bIdx])) {
      tmp[tmpIdx] = values[aIdx];
      aIdx++; tmpIdx++;
    }
    else {
      tmp[tmpIdx] = values[bIdx];
      bIdx++; tmpIdx++;
    }
  }

  //copy back
  for(int i = 0; i < to-from; i++)
    values[from+i] = tmp[i];
}
```

<= Meilleure méthode parce qu'on crée TMP après récursivité car consommation de mémoire total est bien moins inférieur log(n) > n

```java
public static void mergeSort(int[] values) {
  if(values.length == 1)
    return;

  int mid = values.length/2;

  int[] A = new int[mid];
  int[] B = new int[values.length-mid];

  //copy to A
  for(int i = 0; i < mid; i++)
    A[i] = values[i];
  //copy to B
  for(int i = mid; i < values.length; i++)
    B[i-mid] = values[i];

  //sort A and B
  mergeSort(A);
  mergeSort(B);

  //Merge A and B back
  int vIdx = 0;
  int aIdx = 0;
  int bIdx = 0;
  while (aIdx != A.length || bIdx != B.length) {
    if(bIdx == B.length || (aIdx != A.length && A[aIdx] < B[bIdx])) {
      values[vIdx] = A[aIdx];
      aIdx++; vIdx++;
    }
    else {
      values[vIdx] = B[bIdx];
      bIdx++; vIdx++;
    }
  }
}
```

<u>Counting/Bucket Sort</u> = on crée des seaux et pour chacune dans valeurs on ajoute au seau et on prend tous les élément pour les ajouter dans le tableau
Complexité = O(n)
!! Il est impossible de trier plus rapidement que O(nlog(n)) !!

$$\mathcal{O}(f_{\text{space}}(n)) \subseteq \mathcal{O}(f_{\text{time}}(n))$$

➔ Pas cohérent par rapport au résultat au-dessus ?
Il n'y a pas que la complexité temporelle mais aussi la complexité spatiale

Arnaud Strapart
Basé sur le cours LEPL1402

Complexité Spatiale = quantité de mémoire prise pour un certains input => les inputs ne sont pas compris dans la taille (calculée en bits)

	Time complexity	Space complexity
Merge sort	$\Theta(n \log n)$	$\Theta(n)$
Counting/bucket sort	$\Theta(n + maxVal)$	$\Theta(maxVal)$

Complexité dépend de taille de l'input cad le nombre de bits => O(n) = O(2^b)

➔ Complexité dépend que de la taille, pas du contenu !

Arnaud Strapart
Basé sur le cours LEPL1402

Cours Magistral 4 : Complexity 2b

Un type abstrait de données = interface avec une série de méthode spécifiée MAIS ne donne pas la manière dont s'est implémentée ➔ Concret
Une structure de données = implémentation de l'interface du type d'abstrait de données
=> possible d'avoir différentes structures de données pour la même interface

// Implémenter interface itérable : créer un itérateur = objet qui permet d'avoir le next()

Collection de type abstrait :

Bag = collection non ordonnée (sorte de sac de bille)
On peut la construire, ajouter un élément, vérifier si elle est vide et demander sa taille

```
public class Bag<Item> implements Iterable<Item>

                Bag()               create an empty bag
        void    add(Item item)      add an item
     boolean    isEmpty()           is the bag empty?
         int    size()              number of items in the bag
```

Implémentation :

2 possibilités : liste chainée ou tableau
1) Liste Chainée

```
private static class Node<Item> {
    private Item item;
    private Node<Item> next;
}
```

=> Par défaut, si il y a rien, on pointe vers le nul
item = élément qu'on va stocker

LinkedBag

```
public class LinkedBag<Item> implements Bag<Item> {

    private Node<Item> first;     // beginning of bag
    private int n;                // number of elements in bag

    private static class Node<Item> {
        private Item item;
        private Node<Item> next;
    }

    public LinkedBag() {
        first = null;
        n = 0;
    }
    public boolean isEmpty() {
        return first == null;
    }
    public int size() {
        return n;
    }
    public void add(Item item) {
        Node<Item> oldfirst = first;
        first = new Node<Item>();
        first.item = item;
        first.next = oldfirst;
        n++;
    }
    public Iterator<Item> iterator()  {
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Item> {
      // TODO
    }
}
```

Node is a static inner/nested class:
- It doesn't need to exist in its own file Node.java
- The outer-class LinkedBag has access to its private fields (no getter/setter)

Ajoute un élément en premier

The Iterator a non static inner/nested class:
- Its existence it bind to a particular instance of LinkedBag.
- It has access to the instance variables of its attached LinkedBag instance.

Iterator = classe statique
=> que 3 méthodes : hasNext() et next() (et remove())

Classe Node est privée car utile que ds Bag et que la classe peut modifier les nodes

Complexité temporelle (param n) :
isEmpty() : O(1)
add() : O(1)
iterateur :
Création sans utiliser : O(1)
Itérer tous les éléments : O(n)

Implémentation d'un itérator :

```
private class ListIterator implements Iterator<Item> {

    private Node<Item> current = first;

    public boolean hasNext()  { return current != null;                         }
    public void remove()      { throw new UnsupportedOperationException();  }
    public Item next() {
       if (!hasNext()) throw new NoSuchElementException();
         Item item = current.item;
         current = current.next;
         return item;
    }
}
```

Si à la fin : current = null => retourner faux

!! On ne va jamais demander une implémentation de remove() => très compliqué

Arnaud Strapart
Basé sur le cours LEPL1402

```java
public static void main(String[] args) {
    Bag<String> bag = new LinkedBag<String>();
    bag.add("Computer");
    bag.add("Table");
    Iterator ite = bag.iterator();
    ite.next();
    ite.next();
    bag.add("Table");
    if (ite.hasNext()) {
        System.out.println("how come you have some next !?!?");
    }
}
```

Table → Computer → Null

Table → Table → Computer → Null

Pose un problème => 2 stratégies pour éviter
Fail-Fast : interdit de modifier collection alors que tu utilises itérator
Fail-Safe : on ne tient pas compte de *table* car on « gêle » à partir de la création de l'itérator

<u>Fail-fast</u>

```java
private class ListIterator implements Iterator<Item> {

    private Node<Item> current = first;
    private final int nInit = n;

    private boolean failFastCheck() {
        if (n != nInit)
            throw new ConcurrentModificationException("bag modified while iterating on it");
        return true;
    }
    public boolean hasNext()  { return failFastCheck() && current != null;            }
    public void remove()      { throw new UnsupportedOperationException();  }
    public Item next() {
        failFastCheck();
        if (!hasNext()) throw new NoSuchElementException();
        Item item = current.item;
        current = current.next;
        return item;
    }
}
```

Remember the size at the creation of the iterator

If size has changed, something was added while iterating

On va utiliser la méthode add() car c'est la seule qui peut modifier la collection

2) ArrayBag = Tableau qui se redimensionne
On utilise un tableau de taille supérieur à n => on évite de dépasser la taille du tableau (double qd arrive limite)

```java
public class ArrayBag<Item> implements Iterable<Item> {
    private Item[] a;         // array of items
    private int n;            // number of elements on bag

    public ArrayBag() {
        a = (Item[]) new Object[2];
        n = 0;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    // resize the underlying array holding the elements
    private void resize(int capacity) {
        assert capacity >= n;
        Item[] temp = (Item[]) new Object[capacity];
        for (int i = 0; i < n; i++)
            temp[i] = a[i];
        a = temp;
    }

    public void add(Item item) {
        if (n == a.length) resize(2*a.length);    // double size of array if necessary
        a[n++] = item;                            // add item
    }

    public Iterator<Item> iterator() {
        return new ArrayIterator();
    }
}
```

Equivalent to
`a = (Item[]) java.util.Arrays.copyOf(a,capacity);`

Next slide

Object[2] = initialise un tableau de 2 (minimum possible car 1 serait inutile)
La méthode add() vérifie que la taille de mon tableau = nombre d'éléments contenu
=> redimensionne taille tableau si c'est égal à n

Complexité temporelle
resize() : O(n) (equivalent est plus rapide)
isEmpty() et size() : O(1)
add() : O(n)/n = O(1) = complexité amortie

Implémentation de l'itérator

```java
private class ArrayIterator implements Iterator<Item> {

    private int i = 0;
    private final int nInit = n;

    private boolean failFastCheck() {
        if (n != nInit) throw new ConcurrentModificationException("bag modified while iterating on it");
        return true;
    }

    public boolean hasNext()  { return failFastCheck() && i < n;                    }
    public void remove()      { throw new UnsupportedOperationException();  }

    public Item next() {
        failFastCheck();
        if (!hasNext()) throw new NoSuchElementException();
        return a[i++];
    }
}
```

La plus simple est tableau mais complexité est identique. Mais pour ne pas payer de O(n), on prend la structure chainée.

Queue = collection de type FIFO (ordonnée) (file d'attraction pour Disney)
On peut ajouter des éléments à la fin et les retirer au début

`public class Queue<Item> implements Iterable<Item>`

	Queue()	*create an empty queue*
void	enqueue(Item item)	*add an item*
Item	dequeue()	*remove the least recently added item*
boolean	isEmpty()	*is the queue empty?*
int	size()	*number of items in the queue*

Enqueue() à la fin et dequeue() au début

Stack = collection LIFO (ordonnée) (Pezz)
On peut ajouter et retirer au début

Arnaud Strapart
Basé sur le cours LEPL1402

```
public class Stack<Item> implements Iterable<Item>

              Stack()             create an empty stack
         void push(Item item)     add an item
         Item pop()               remove the most recently added item
      boolean isEmpty()           is the stack empty?
          int size()              number of items in the stack
```

2 implémentations possibles : tableau ou structure chainée
=> Pop et push doit être fait au début de la structure chainée
=> Pop et push doit être fait à la fin du tableau

Ds les queues et stack, on peut retirer des éléments cad taille de la capacité du tableau doit être réduit

Preuve de Programme

=> Le plus simple pour écrire des programmes est de savoir prouver que le programme est correct
Comment faire ça ? On peut faire des preuves mathématiques
2 techniques : preuve par invariant ou induction

Loop variant: i
Loop invariant: m is equal to the maximum value on a[0..i-1]

```java
public static int max(int [] a) {
    int m = a[0];
    int i = 1;
    // inv: m is equal to the maximum value on a[0..0]
    while (i != a.length) {
        // inv: m is equal to the maximum value on a[0..i-1]
        if (m < a[i]) {
            m = a[i];
        }
        // m is qual to the maximum value on a[0..i]
        ++i;
        // inv: m is equal to the maximum value on a[0..i-1]
    }
    // m is equal to the maximum value on a[0..i-1] and
    //    i == a.length (escape condition)
    return m;
}
```

Initialization
Maintenance
Termination

Pour prouver que algorithme itératif est correct, on utilise des invariants de boucle (= propriété logique qui doit être vrai avant = **Initialisation**/pendant = **Maintenance** et sortie = **Terminaison** à chaque exécution de boucle)
Vu que Initialisation = Terminaison => vérifié

Implémentation récursive est une implémentation avec une condition initiale et un rappel à la fonction si elle n'est pas respectée
!! Elle doit toujours se terminer

```java
public static int max(int [] a, int i) {
    if (i == 0) return a[i];
    else return Math.max(a[i],max(a,i-1));
}
```

- Base case: i=0 max(a[0]) = a[0]
- Induction:
  - max(a[0],…a[i-1],a[i]) =
  - max(max(a[0],…a[i-1]),max(a[i])) = (associativity)
  - max(max(a[0],…a[i-1]),a[i])
  - assuming max(a[0],…a[i-1]) is correct (induction hypothesis), this is what we compute

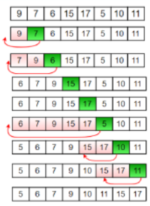
➔ Comment prouver que c'est juste ? Cas de l'induction
Pour prouver que l'algorithme fonctionne, on doit prouver d'abord que le cas de base (i=0) est juste puis qu'il est correct pour i – 1 en supposant que le cas i est vrai

Tri par insertion (boucle while)

Permet de trier un tableau => moins efficace que le Murge Sort

```java
public static void InsertionSort(int [] a) {
    int j = 1;
    while (j < a.length) {
        // Invariant1 Iter
        int v = a[j];
        int i = j-1;
        while (i >= 0 && a[i] > v) {
            // Invariant2
            a[i+1] = a[i];
            i = i-1;
        }
        a[i+1] = v;
        j = j+1;
    }

}
```



On va décaler ce qui n'est pas trié pour venir mettre l'élément où il doit se trouver
Complexité : tableau déjà trier : O(n), qd tout doit être décalé : O(n^2)
Pour prouver qu'il est correct => méthode des invariants

```java
public static void InsertionSort(int [] a) {
    int j = 1;
    while (j < a.length) {
        // Invariant1 Iter
        int v = a[j];
        int i = j-1;
        while (i >= 0 && a[i] > v) {
            // Invariant2
            a[i+1] = a[i];
            i = i-1;
        }
        a[i+1] = v;
        j = j+1;
    }

}
```

- a is a permutation of the original array
- prefix a[0..j-1] is a sorted version of the same prefix of the original array

Need to prove that Invariant1 holds here. Therefore we need also Invariant2

Termination: a = sort(old a)

```java
public static void InsertionSort(int [] a) {
    int j = 1;
    while (j < a.length) {
        // Invariant1
        int v = a[j];
        int i = j-1;
        while (i >= 0 && a[i] > v) {
            // Invariant2
            a[i+1] = a[i];
            i = i-1;
        }
        a[i+1] = v;
        j = j+1;
    }

}
```

- a is a permutation of the original array
- prefix a[0..j-1] is a sorted version of the same prefix of the original array

- a[0..i] + v + a[i+2..n-1] is a permutation of the original array
- a[0..j-1] is sorted
- a[i+1..j] is sorted
- v <= a[i+1..j]

- a[0..i] + v + a[i+2..n-1] is a permutation of the original array
- a[0..j-1] and a[i+1..j] are sorted
- a is not a permutation of the original array (value v is missing and value a[i+1] present twice).
- We reinsert v at the right position i+1 by overriding the value present twice

Arnaud Strapart
Basé sur le cours LEPL1402

!! Pour les boucle For => Méthode des invariants aussi (identique pour les boucles While)

<u>Prouver qu'une classe est correct :</u>
Ce qui change : l'état => On crée des invariants de classes qui doit pouvoir être vérifié à tous moments par l'état des instances de la classes
!! Si tout est en publique, on peut modifier les invariants

```java
public class SList<T> implements Iterable<T> {
    private class SListNode { … }
    // Class invariant:
    // size is the number of elements in the list formed
    // by the following the chain next pointers starting at head
    // and finishing with a null reference:
    // head -> head.next -> head.next.next ... -> null
    // if size == 0 then head == null
    private SListNode head;
    private int size;
    public void insertFront(T item) {
        head = new SListNode(item, head);
    }

    public int size() {
        int s = 0;
        SListNode n = head;
        while (n != null) {
            n = n.next;
            s++;
        }
        return s;
    }
    public Iterator<T> iterator() {/* TODO */}
}
```

😕 Is my invariant correctly maintained ?

InsertFront() viole la condition de size de l'invariant
Solution :

```java
public void insertFront(T item) {
    head = new SListNode(item, head);
    size++;
}
```

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

!! Quand il y a trop d'appel imbriqué dans les programmes récursifs => on dépasse la taille initiale du programme acceptée par Java => erreur de type « StackOverFlow »
Comment résoudre :
Souvent récursion infinie ou paramètre trop élevé => modifie taille de la stack de Java (Xss4m) ou modifie la récursion en un programme itératif (boucle while)

Arnaud Strapart
Basé sur le cours LEPL1402

Exemple vu en cours :

1) Fibonacci

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

```java
int fiboRecursive(int index) {
    if(index <= 1)
        return index;
    return fiboRecursive(index-1) +
fiboRecursive(index-2);
}
```

Temporal complexity in function of n?

$\mathcal{O}(2^n)$

```java
int fiboIterative(int index) {
    if(index <= 1)
        return index;
    int f0 = 0;
    int f1 = 1;
    for(int i = 2; i <= index; i++) {
        int f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
    }
    return f1;
}
```

$\mathcal{O}(n)$

2) Maximum sum subarray

Somme partielle = permet de calculer les éléments d'un tableau en O(1) à la place de O(n)

```java
int[] answer = new int[]{a[0], 0, 0};
int[] cumsum = new int[a.length+1];

cumsum[0] = 0;
for(int i = 0; i < a.length; i++)
    cumsum[i+1] = cumsum[i] + a[i];

for(int start = 0; start < a.length; start++) {
    for(int end = start; end < a.length; end++) {
        int sum = cumsum[end+1]-cumsum[start];

        if(sum > answer[0]) {
            answer[0] = sum;
            answer[1] = start;
            answer[2] = end;
        }
    }
}

return answer;
```

Cette méthode a une complexité de O(n^2) MAIS on peut encore l'améliorer à l'aide de

$$\text{best}[i] = \begin{cases} \text{best}[i-1] + \text{a}[i] & \text{if best}[i-1] + \text{a}[i] > \text{a}[i] \\ \text{a}[i] & \text{otherwise} \end{cases}$$

The answer is then the maximum of the array best.

3) Finding mountains and valleys

```java
int[] down = new int[array.length+1];
int[] up = new int[array.length+1];

down[0] = 0;
up[array.length] = 0;

for(int i = 0; i < array.length; i++) {
    if(array[i] <= 0)
        down[i+1] = down[i] - array[i];
    else
        down[i+1] = 0;
}

for(int i = array.length - 1; i >= 0; i--) {
    if(array[i] >= 0)
        up[i] = up[i+1] + array[i];
    else
        up[i] = 0;
}

int highestValley = 0;
for(int i = 0; i < array.length; i++) {
    int v = Math.min(down[i], up[i]);
    if(v > highestValley)
        highestValley = v;
}

return highestValley;
```

Pour retrouver les vallées, on va prendre 2 listes (une qui représente les vallées et une autre les montagnes)

Array	1	1	1	-1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1
Down	0	0	0	1	2	3	4	5	0	0	0	0	0	0	1	2
Up	3	2	1	0	0	0	0	0	6	5	4	3	2	1	0	0

Pour retrouver les montages, on regarde l'opposé des listes

```java
int[] neg = new int[array.length];
for(int i = 0; i < array.length; i++)
    neg[i] = -array[i];

return new int[]{findValley(array),findValley(neg)};
```

Arnaud Strapart

4) Merge sort

```java
void sort(int[] a) {
    int n = a.length;
    int[] aux = new int[a.length];

    for(int s = 1; s <= n; s*=2)
        for(int i = 0; i <= n-s; i+=2*s)
            merge(a, aux, i, i+s, Math.min(i+2*s, n));
}

void merge(int[] a, int[] aux, int lo, int mid, int hi) {
    int a_idx = lo;
    int b_idx = mid;
    int aux_idx = 0;
    while(a_idx != mid && b_idx != hi) {
        if(a[a_idx] < a[b_idx]) {
            aux[aux_idx] = a[a_idx];
            a_idx++;
        }
        else {
            aux[aux_idx] = a[b_idx];
            b_idx++;
        }
        aux_idx++;
    }
    while(a_idx != mid) {
        aux[aux_idx] = a[a_idx];
        a_idx++;
        aux_idx++;
    }
    while(b_idx != hi) {
        aux[aux_idx] = a[b_idx];
        b_idx++;
        aux_idx++;
    }

    for(int i = 0; i < (hi-lo); i++)
        a[lo+i] = aux[i];
}
```

Le but est de faire des petites listes de 2 puis de les trier par 2 et ainsi de suite

5) Hanoï tower

```java
void towerOfHanoi(int n, Stack<Disk> a, Stack<Disk> b, Stack<Disk> c)
{
    if(n == 0)
        return;
    towerOfHanoi(n-1, a, c, b); //move n-1 disks from a to c
    b.push(a.pop());            //move last disk to b
    towerOfHanoi(n-1, c, b, a); //move n-1 disks from c to b
}
```

On déplace les n-1 élément sur b puis on déplace le $n^{ième}$ sur c et on redéplace les n-1 sur c