

INF220 - Algorithmique

Page web du cours : <http://tinyurl.com/INF120-2012S2>

• Contact

- Courriel : philippe.gambette@gmail.com (INF220 doit apparaître dans le sujet du courriel)
- Avant ou après le cours
- Possibilité de poser des questions, de demander des exercices supplémentaires d'entraînement.

• Notes et devoirs

Chez soi :

- Projet Monopoly : exercices à trous à remplir au plus tard la veille des cours/TP/TD pour vous inciter à travailler **régulièrement** et corriger vos erreurs. **Annonce sur IRIS 6 jours avant.**

Pendant les cours :

- Note globale de TP à la fin du semestre, prenant en compte votre avancée à chaque séance.
- Devoir final le 18 juin.

• Sources

- Cours de Jean-François Berdjugin à l'IUT de Grenoble
<http://berdjugin.com/enseignements/inf/inf220/>
- <http://xkcd.com>, <http://xkcd.free.fr>

- Cours de Xavier Heurtebise à l'IUT de Provence
<http://x.heurtebise.free.fr>
- *Le livre de Java premier langage*, d'A. Tasso

Récurtivité

2 façons de le voir :

- Une mise en abyme

Comment dessiner ces images ?
Pour dessiner la grande image, j'utilise
le dessin de l'image en plus petit.



- Une baguette magique algorithmique

Si je sais construire le n -ième
objet à partir du $(n-1)$ -ième
objet, et que je sais construire
le premier objet, alors je sais construire tous les objets.



Exemple des factorielles :

$$\text{factorielle}(1) = 1$$

$$\text{factorielle}(2) = 1 \times 2 = 2$$

$$\text{factorielle}(3) = 1 \times 2 \times 3 = 6$$

$$\text{factorielle}(4) = 1 \times 2 \times 3 \times 4 = 24$$

Initialisation :

$$\text{factorielle}(1) = 1$$

Formule d'hérédité :

$$\text{factorielle}(n+1) = \text{factorielle}(n) \times (n+1)$$

La "minute xkcd" - Jeu de rôle sur table



<http://xkcd.com/244>

<http://xkcd.free.fr?id=244>

Algorithme récursif de calcul de la factorielle

Exemple des factorielles :

- **Initialisation** : la première factorielle est 1
- **Hérédité** : pour passer de factorielle($n-1$) à factorielle(n), je **multiplie par n**

Algorithme **factorielle**

Entrées :

Type de sortie :

Variable :

Début

Fin

Trace de **factorielle(4)** :

La “minute mathématique”

Théorème : Je sais monter une échelle

Démonstration :

Initialisation : je sais monter depuis le sol jusqu'au premier barreau.

Hérédité : si je sais monter jusqu'au $(n-1)$ -ième barreau, je saurai monter jusqu'au n -ième barreau.

Algorithme récursif de calcul du produit

Exemple du produit :

- Initialisation : $a \times 0 = 0$
- Formule d'hérédité : $a \times n = a \times (n-1) + a$

Algorithme **produit**

Entrées :

Type de sortie :

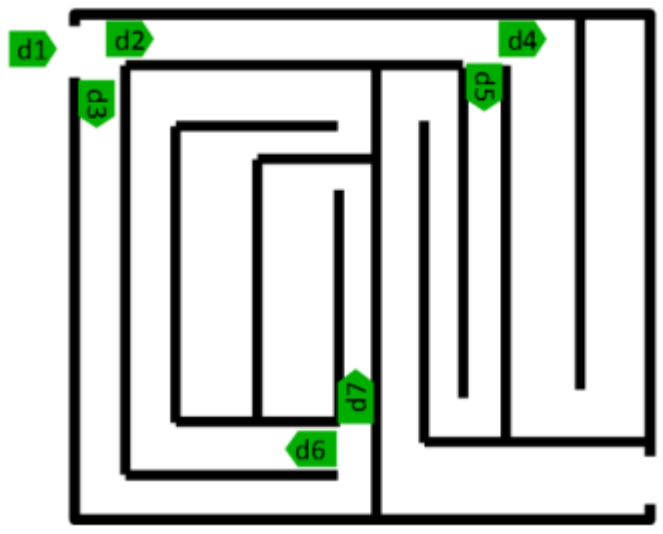
Variable :

Début

Fin

Trace de **produit**(4) :

Algorithme récursif de parcours groupé d'un labyrinthe



Le labyrinthe et les robots tueurs :
10 personnes dans un labyrinthe poursuivies par des robots tueurs. Si on appuie sur le bouton stop dans les 5 minutes, les robots sont désactivés.

Algorithme **trouveBouton**

Entrées : entier *nombrePersonnes*, flottant *tempsRestant*, chaîne de caractères *direction*

Type de sortie : booléen



Si on trouve le bouton à temps, on appuie dessus, et l'algorithme renvoie VRAI. Si on trouve une intersection, le groupe se sépare en deux. Si on ne trouve pas le bouton à temps, on renvoie FAUX.

Que renvoie **trouveBouton**(10,5,"d1") ?

Arbre d'exécution de **trouveBouton**(10,5,"d1").

trouveBouton(10,5,"d1")

Principe d'un algorithme de tri

Entrée :

Ensemble d'éléments tous comparables deux à deux par un ordre \leq

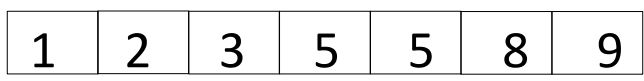
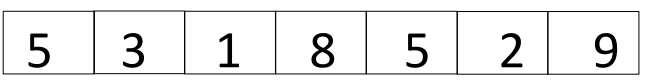
Sortie :

Éléments **triés selon cet ordre**, du plus petit au plus grand

Ensemble d'éléments	Ordre	Tous comparables deux à deux ?
tableau d'entiers	plus petit	
tableau de chaînes de caractères	lexicographique (du dictionnaire)	
tableau de cartes à jouer	$2 \leq 3 \leq \dots \leq 10 \leq V \leq D \leq R \leq 1$	
tableau de cartes à jouer	$\clubsuit \leq \diamondsuit \leq \heartsuit \leq \spadesuit$ puis $2 \leq 3 \leq \dots \leq 10 \leq V \leq D \leq R \leq 1$	

Exemple avec un tableau d'entiers :

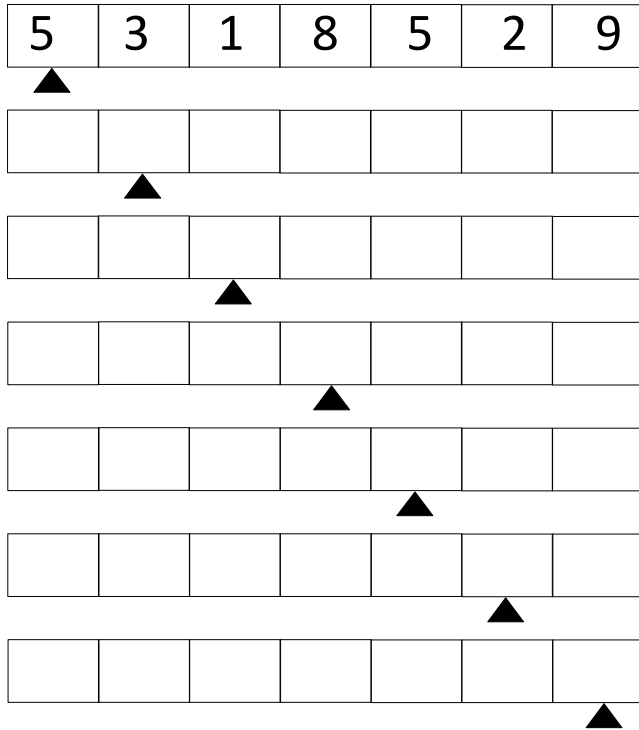
en anglais : trié = sorted / trier = to sort



Tri par sélection (ou tri par extraction)

Idée : à la i -ième étape, on prend le plus petit élément à partir de la i -ième case (comprise) et on l'échange avec l'élément de la i -ième case.

Exemple avec un tableau d'entiers :



Algorithme **positionMinimum**

Entrée : tableau d'entiers tab ,
entier $debut$

Type de sortie : entier

Variables : entiers $position$, i , min

Début

$position \leftarrow debut$

$i \leftarrow debut$

$min \leftarrow \mathbf{Case}(tab, i)$

Tant que $i \leq \mathbf{Longueur}(tab)$ faire :

si $\mathbf{Case}(tab, i) < min$ alors :

$position \leftarrow i$

$min \leftarrow \mathbf{Case}(tab, i)$

$i \leftarrow i+1$

Fin Tant que

renvoyer $position$

Fin

Algorithme **triSelection**

Entrée : tableau d'entiers tab

Type de sortie : tableau d'entiers

Variables : entiers i , $temp$, $posMin$

Début

$i \leftarrow 1$

Tant que $i < \mathbf{Longueur}(tab)$ faire :

$posMin \leftarrow \mathbf{positionMinimum}(tab, i)$

// $posMin$ peut être égal à i

$temp \leftarrow \mathbf{Case}(tab, i)$

$\mathbf{Case}(tab, i) \leftarrow \mathbf{Case}(tab, posMin)$

$\mathbf{Case}(tab, posMin) \leftarrow temp$

$i \leftarrow i+1$

Fin Tant que

renvoyer tab

Fin

Tri à bulles

Idée : tant que le tableau n'est pas trié, on le parcourt en effectuant l'opération suivante à la i -ième case : si elle est supérieure à la suivante, on les échange.

Exemple avec un tableau d'entiers :

5	3	1	8	5	2	9
---	---	---	---	---	---	---

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

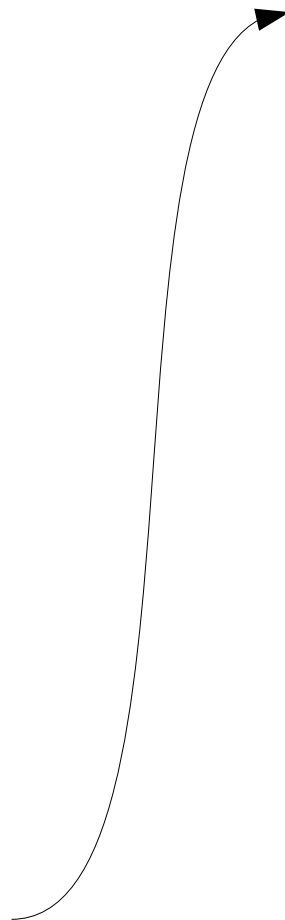
--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--



--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

Les listes

Le problème des tableaux :

- taille fixe
- insertion "difficile" d'un élément
- suppression "difficile" d'un élément

L'intérêt des listes :

- taille variable
- insertion facile d'un élément
- suppression facile d'un élément

Mais :

- accès "difficile" au *i*-ième élément

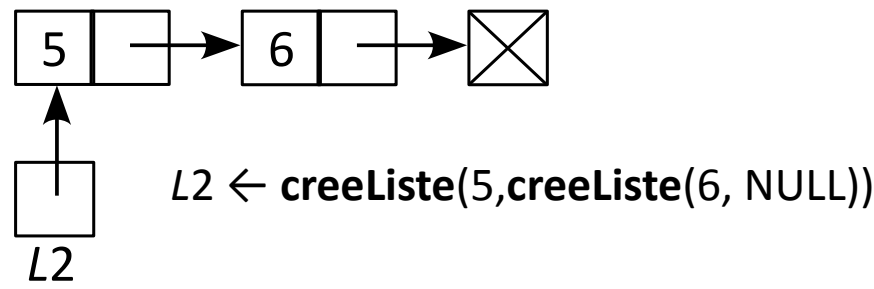
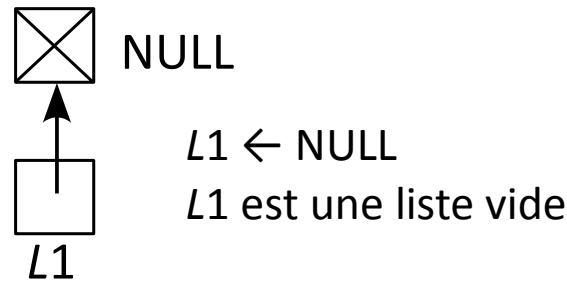
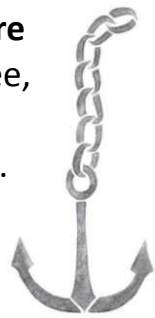
Une liste "simplement chaînée" est :

- soit une liste vide
- soit une première case qui contient une valeur, et qui pointe vers une liste

Une liste d'entiers est :

- soit une liste vide
- soit une première case qui contient un entier, et qui pointe vers une liste d'entiers

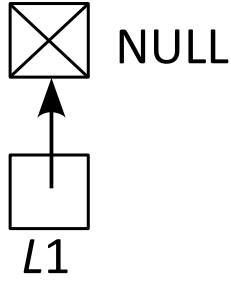
Liste "simplement chaînée" : la chaîne d'une ancre
Quand l'ancre est jetée et toute la chaîne déroulée, on a facilement accès au premier maillon (sur le bateau). Pour les suivants il faut remonter l'ancre. Quand on a un maillon sous les yeux on peut facilement le supprimer (le casser et accrocher le précédent au suivant), en insérer un...



Fonctions de base sur les listes

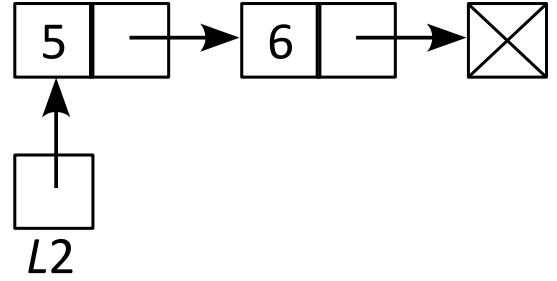
Sur une liste $L1$ ou $L2$, on peut :

• savoir si elle est vide :
 $L1 = \text{NULL}$?



si $L2$ est non vide :
• accéder à la valeur de son premier élément :
tete($L2$) renvoie ...

si $L2$ est non vide :
• accéder à la liste qui commence à la case suivante :
suisvant($L2$) renvoie ...

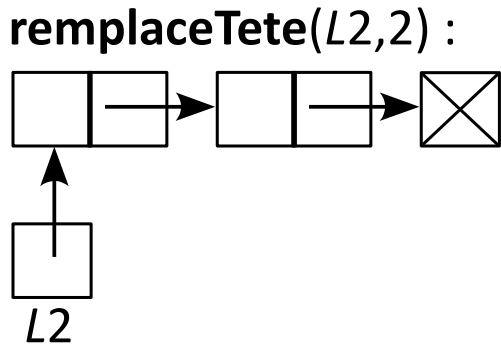


Modification de l'élément en tête d'une liste (facile pour liste, facile pour tableau)

Écrire l'algorithme **remplaceTete** qui prend en entrée une liste d'entiers L et un entier a , et renvoie la liste L dont le premier élément est remplacé par a

Algorithme **remplaceTete**
Entrées : liste d'entiers L , entier a
Type de sortie : liste d'entiers
Début

renvoyer
Fin

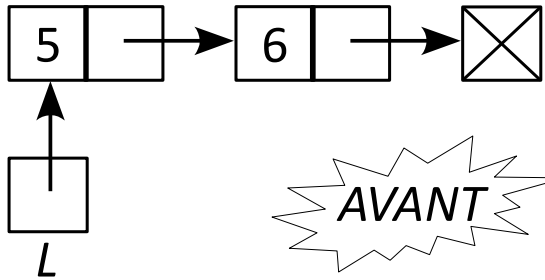


Fonctions de base sur les listes

Suppression de l'élément en tête d'une liste (facile pour liste, pas facile pour tableau)

Par exemple, supprimer 5 au début de la liste

```
L=creeListe(5,creeListe(6, NULL))
```



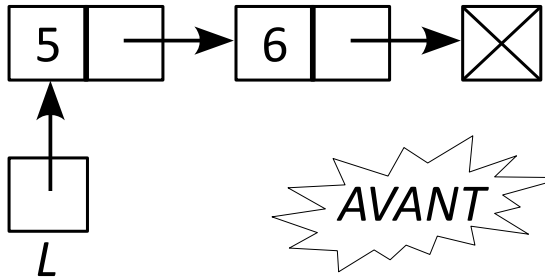
AVANT

APRES

Insertion d'un élément en tête d'une liste (facile pour liste, pas facile pour tableau)

Par exemple, insérer 2 au début de la liste

```
L=creeListe(5,creeListe(6, NULL))
```



AVANT

APRES

Fonctions de base sur les listes

Valeur du i -ième élément de la liste (pas facile pour une liste, facile pour un tableau)

Nécessite de **parcourir la liste** !

Ecrire la fonction **valeur** qui prend en entrée une liste d'entiers L et un entier i , et qui renvoie la valeur de la i -ième case de L , si elle existe, et -1 sinon.

Algorithme **valeur**

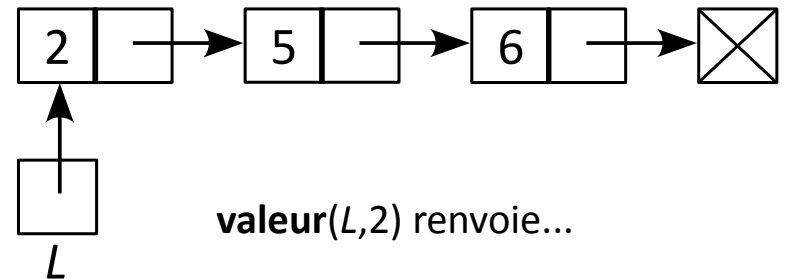
Entrées : liste d'entiers L , entier a

Type de sortie :

Variables :

Début

Fin



Piles et files

Files

FIFO “First In First Out”

File d'attente à la Poste

Quatre fonctions :

- **créer** : créer une file vide
- **enfiler** : ajouter un élément **en fin** de file
- **défiler** : lire la valeur de l'élément **au début** de la file en l'enlevant de la file
- **tester si vide** : savoir si la file est vide

implémentable avec liste
doublement chaînée

Piles

LIFO “Last In First Out”

Pile de documents à travailler

Quatre fonctions :

- **créer** : créer une pile vide
- **empiler** : ajouter un élément **en haut** de la pile
- **dépiler** : lire la valeur de l'élément **en haut** de la pile en l'enlevant de la pile
- **tester si vide** : savoir si la pile est vide

implémentable avec liste
simplement chaînée

Pourquoi utiliser ces structures de données ?

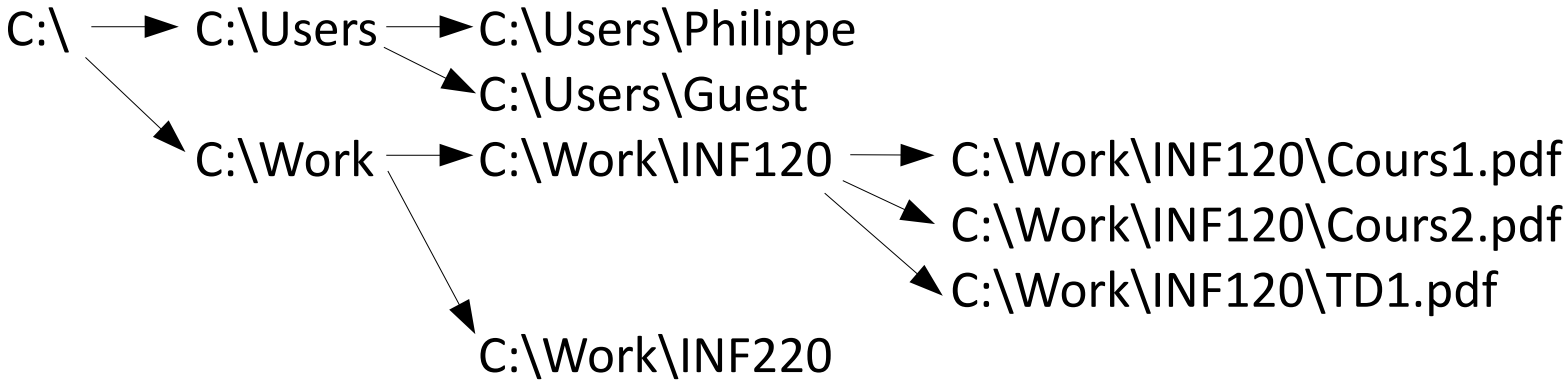
- Plus simple à programmer qu'une liste
- Fonctions plus limitées, adaptées aux besoins

Intérêt des arbres

Intérêt des arbres :

- représentation d'un objet qui a une structure arborée :

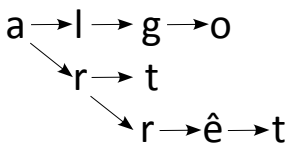
→ contenu d'un disque dur :



- stockage astucieux d'éléments

→ trouver rapidement le minimum des valeurs stockées

→ stocker un dictionnaire de manière compacte :

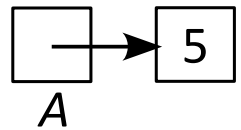


La "minute xkcd" - Arbre
<http://xkcd.com/835>
<http://xkcd.free.fr/?id=835>

Définition réursive et illustrée des arbres

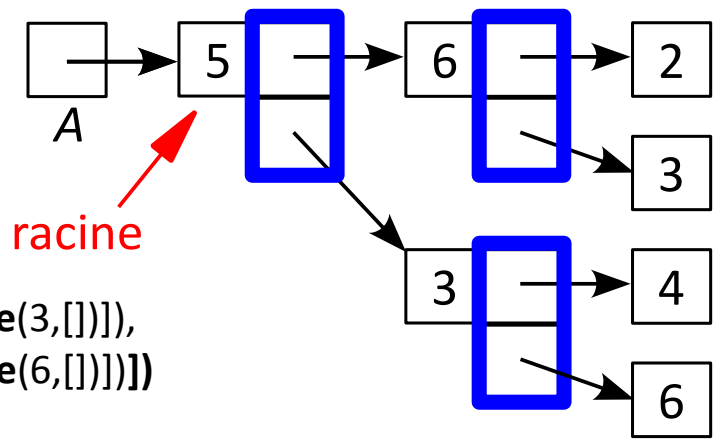
Un **arbre** :

- soit une **feuille**, qui contient une valeur, mais n'a pas d'enfant



$A \leftarrow \text{CreeArbre}(5, [])$

- soit un **noeud** qui contient une valeur, et un **tableau d'enfants** dont chaque case pointe vers un **arbre**



$A \leftarrow \text{CreeArbre}(5, [\text{CreeArbre}(6, [\text{CreeArbre}(2, []), \text{CreeArbre}(3, [])]), \text{CreeArbre}(3, [\text{CreeArbre}(4, []), \text{CreeArbre}(6, [])])])$

Si chaque **tableau d'enfants** a au plus deux cases, c'est un **arbre binaire**.

Fonctions de base sur les arbres

Tester si l'arbre est une feuille :
estFeuille(A)

Récupérer la valeur de la racine de l'arbre
racine(A)

Récupérer le tableau des enfants de la racine de l'arbre
enfants(A)

