Programmation avancée et Complexité

Frédéric Flouvat

Université de la Nouvelle-Calédonie

Présentation de l'EC

Objectifs : Approfondir les acquis en algorithmique/programmation et aborder la question de l'efficacité des algorithmes

Trois chapitres :

1. Rappels d'algorithmique
2. De Python au langage C
3. Complexité et ordre de grandeur

Volume horaire : 12h CM (6 séances) / 12h TD (6 séances) / 24h TP (12 séances)

- CM et TD mélangés (en fonction de la progression du cours)

Evaluation en Programmation Avancée :

- QCM en début de chaque cours (moyenne, coef. 0.1), un contrôle "papier" à la fin du cours (coef. 0.5), trois TP notés (moyenne, coef. 0.4)
- 2eme chance : un contrôle "papier" la dernière semaine de cours qui remplace les notes précédentes si la note est meilleure

Quelques références bibliographiques

T.H. Cormen.
Introduction à l'algorithmique : cours et exercices.
Sciences sup. Dunod, 2002.

Donald E. Knuth.
Art of Computer Programming (3rd Edition).
Addison-Wesley Professional, November 1997.

Un grand nombre de ressources sur internet

- Algorithmes récursifs, S. Vegel et M.-E. Voge, Université de Nice, 2007.
- C for Python programmers, C. Burch et E. Patitsas, Hendrix College et University of Toronto, 2012.
- ...

Plan

1. Rappels d'algorithmique
   - problèmes, algorithmes, programmes
   - Algorithmes itératifs et récursifs
   - Application aux algorithmes de tri

Programmation : le processus **classique**

1. Etude préalable : compréhension et modélisation du problème
2. Spécifier les données du problème et les résultats
   - décrire les informations en entrée (les données à utiliser/traiter) et en sortie (le résultat cherché)
3. Choisir une méthode pour résoudre le problème
   3.1 trouver des solutions/méthodes en langage naturel (pas informatique)
   3.2 décomposer en plusieurs algorithmes/modules
   3.3 écrire les algorithmes et décrire la représentation des données (cad les structures de données)
   3.4 choisir la méthode en fonction de critéres tels que l'efficacité ou la simplicité
4. Programmer dans un langage informatique
   - traduire la solution sous forme de programme
5. Tester et évaluer le travail réalisé
   - fonctionne-t-il correctement ? répond-il aux besoins initiaux ?
6. Documenter le logiciel

Qu'est-ce qu'un algorithme ?

Algorithme ➾ méthode permettant de résoudre un problème de calcul bien spécifié

- p.ex. trier une suite de nombres, analyser de l'ADN ...
- exprimé dans un langage entre le langage humain et les langages informatiques

Exemple de problème : calculer x^n

- entrée : un réel x $(\neq 0)$ et un entier n
- sortie : le réel x^n

Exemple de résolution :

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

1: res $\leftarrow$ 1
2: **Pour** i de 1 à n **faire**
3:     res $\leftarrow$ res . x
4: **Fin Pour**
5: **Retourner** res

Qu'est-ce qu'un algorithme ?

Problématiques de l'algorithmique :

- trouver une méthode de résolution (exacte ou approchée) du problème
- trouver une méthode efficace

Remarque : algorithmes vs programmes

- un algorithme est une méthode décrite dans un langage proche du langage naturel
  - ex : "Pour i de 1 à n faire"
- un programme est la réalisation (i.e. l'*implémentation*) d'un algorithme via un langage de programmation
  - ex : "for(int i = 1 ; i < =n ; i++) " (en langage C)

Méthodologie de conception d'un algorithme

Analyse descendante

- décomposer un problème complexe en sous problèmes et ces sous problèmes en d'autres sous problèmes jusqu'à obtenir des problèmes faciles à résoudre

Garder à l'esprit

- la modularité : un module résout un petit problème donné et doit être réutilisable
  - dans notre cas module = fonction/procédure
- l'efficacité : étudier la *complexité* de l'algorithme

Représenter les données

Un algorithme décrit une méthode qui effectue des traitements (opérations) sur des informations (données)

➫ la représentation des données est donc un aspect important en algorithmique

- une des étapes principales dans la construction d'un programme

Objectifs :

- représentation appropriée des données
  - des algorithmes moins complexes et plus efficaces
- réutilisation des mêmes structures de données dans beaucoup d'algorithmes

Algorithmes itératifs/récursifs : définitions

Pour une méthode de résolution, plusieurs algorithmes possibles

➫ p.ex. Itératif vs Récursif

Itératif : Un algorithme est dit itératif s'il utilise uniquement des boucles et/ou des instructions conditionnelles pour résoudre le problème

Récursif : Un algorithme est dit récursif s'il s'appelle lui-même (directement ou indirectement) une ou plusieurs fois pour traiter des sous-problèmes similaires

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n'

1: res ← 1
2: **Pour** i de 1 à n **faire**
3: res ← res . x
4: **Fin Pour**
5: **Retourner** res

Fonction Puissance(x,n)

Entrée: un réel x, un entier n
Sortie: le réel x^n

1: **Si** n = 1 **Alors**
2: **Retourner** x
3: **Fin Si**
4: res ← Puissance(x, n-1) . x
5: **Retourner** res

Les différents types de récursivité

Récursivité simple :

- fonction/procédure qui s'appelle elle-même une seule fois

```
Fonction Puissance(x,n)
Entrée:  un réel x, un entier n
Sortie:  le réel x^n
Si n = 0 Alors
    Retourner 1
Fin Si
res ← Puissance(x, n-1) . x
Retourner res
```

"Récursivité" croisée :

```
Fonction Pair(n)
Entrée:  un entier n
Sortie:  vrai ou faux selon que n soit pair ou non
Si n = 0 Alors
    Retourner vrai
Sinon
    Retourner Impair( n-1)
Fin Si
```

```
Fonction Impair(n)
Entrée:  un entier n
Sortie:  vrai ou faux selon que n soit impair ou non
Si n = 0 Alors
    Retourner faux
Sinon
    Retourner Pair( n-1)
Fin Si
```

Les différents types de récursivité

Récursivité multiple :

- fonction/procédure effectuant plusieurs appels récursifs

Procedure AffichageRec2(i, tabmots)

Entrée: un entier i représentant un indice du tableau, un tableau de chaînes de caractères $tabmots$

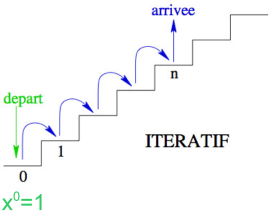
```
Si i < longueur[ tabmots ] Alors
    AffichageRec2( i+1, tabmots )
    AfficheEcran( tabmots[i] )
    AffichageRec2( i+1, tabmots )
Fin Si
```

Exemple d'exécution avec ['un ','deux ', 'trois '] et $i = 0$:

➾ affiche à l'écran : *trois deux trois un trois deux trois*

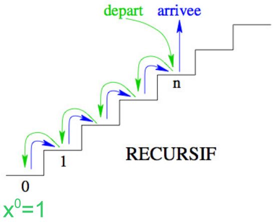
Principe de la récursivité simple : l'escalier

Exemple avec x^n (propriété $x^n = x^{n-1} \times x$)



- départ : information connue
- monte vers le résultat
- arrêt dés que le résultat est trouvé

➩ boucle (pour, tant que, répéter, ...)



- départ : information cherchée
- descend vers l'information connue
- arrêt en bas
- monte vers le résultat
- arrêt dés que le résultat est trouvé

➩ appels récursifs (algorithme monte/descend les marches)
➩ mais comment ?

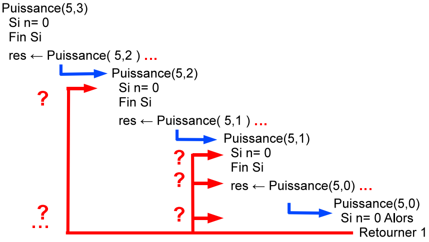
Fonctionnement de la récursivité simple

Utilisation d'une pile systéme (au niveau de la mémoire centrale) pour stocker les appels récursifs

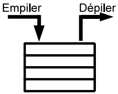
- permet au programme de se rappeler où il en était avant l'appel récursif

```
Fonction Puissance(x,n)

Entrée: un entier n≥ 0
Sortie: l'entier x^n
  Si n = 0 Alors
    Retourner 1
  Fin Si
  res ← Puissance(x, n-1) . x
  Retourner res
```



Rappel : une pile est une structure de données de type tableau où la derniére information entrée est aussi la premiére à sortir



Fonctionnement de la récursivité simple

A chaque appel d'un algorithme récursif, les paramétres transmis par valeur et les variables locales sont empilés

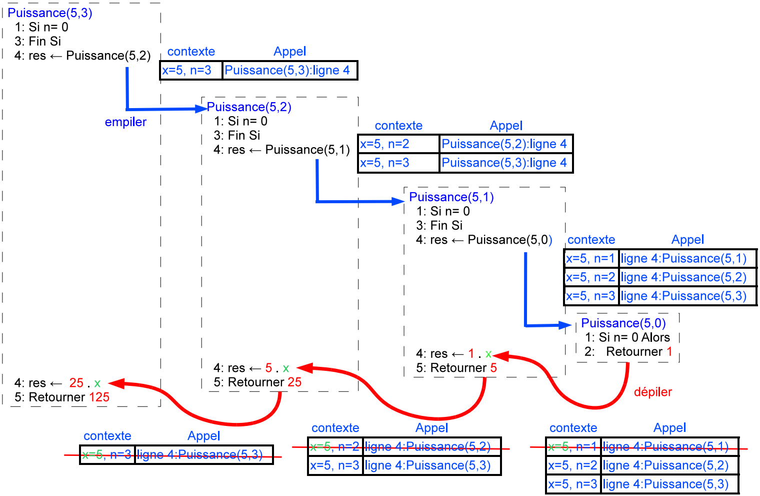
- paramétres + variables locales = contexte d'activation de l'appel

De manière plus générale, une exécution en 2 phases :

- empiler dans la pile jusqu'à arriver à un appel de la fonction pour lequel une condition d'arrêt est vérifiée
- dépiler en utilisant les résultats des appels récursifs pour terminer les appels précédents

Remarque : gestion de la pile transparente pour le programmeur et l'utilisateur

Exemple d'exécution avec une récursivité simple



Exercice

Ecrire la trace de l'exécution des deux algorithmes récursifs suivants. Donner uniquement le contenu de la pile à chaque récursion ainsi que les messages affichés à l'écran.

Hypothèses :

- l'appel est fait avec le tableau ['bonjour','je','suis', 'récursif']
- la procédure $AfficheEcran(str)$ existe et permet d'afficher la chaîne de caractéres str à l'écran

Procedure AffAp(i, tabmots)

```
Entrée: un entier i initialisé à 0, un tableau de chaînes de
        caractéres tabmots
Si  i < longueur[ tabmots ]  Alors
    espace ← ' '
    AfficheEcran( tabmots[i] )
    AfficheEcran( espace )
    AffAp( i+1, tabmots )
Fin Si
```

Procedure AffAv(i, tabmots)

```
Entrée: un entier i initialisé à 0, un tableau de chaînes de
        caractéres tabmots
Si  i < longueur[ tabmots ]  Alors
    espace ← ' '
    AffAv( i+1, tabmots )
    AfficheEcran( tabmots[i] )
    AfficheEcran( espace )
Fin Si
```

Principe de la récursivité multiple : couper un gâteau

Manger un gâteau

- gâteau trop gros pour être mangé directement
- on n'arrive à manger qu'une petite part (une bouchée) à la fois
- on sait comment couper le gâteau

➩ on coupe le gâteau jusqu'à ce que les parts soient suffisamment petites pour que l'on puisse les manger

Résoudre un problème

- problème trop difficile à traiter par une approche classique
- on ne sait résoudre simplement qu'une étape
- on sait comment décomposer le problème

➩ on décompose le problème jusqu'à ce qu'il soit suffisamment simple pour que l'on puisse le résoudre

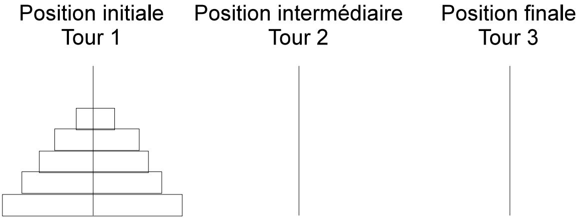
➩ Approche "diviser pour régner"

Principe de la récursivité multiple : couper un gâteau

Exemple du problème des tours de Hanoï

- Transférer n disques (les un après les autres) de l'axe 1 à l'axe 3, en utilisant B, de sorte que jamais un disque ne repose sur un disque de plus petit diamètre.

$n = 5$



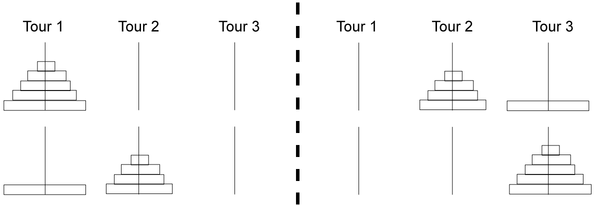
Principe de la récursivité multiple : couper un gâteau

Problème des tours de Hanoï difficile a priori, mais on sait :

- résoudre ce problème si $n-1$ disques sont déjà sur la tour intermediaire
- déplacer un disque

Solution :

- déplacer d'abord $n-1$ disques de la tour initiale vers la tour intermédiaire (en respectant la contrainte sur la taille des disques)
- déplacer le disque n de la tour initiale vers la tour finale
- déplacer $n-1$ disques de la tour intermédiaire vers la tour finale



Principe de la récursivité multiple : couper un gâteau

Algorithme résolvant le problème des tours de Hanoï

```
Procedure Hanoi( n, départ, final, intermédiaire)
Entrée: n > 1 un entier
Si n = 1 Alors
    déplacer( départ , final )
Sinon
    Hanoi( n-1, départ, intermédiaire, final)
    déplacer( départ , final )
    Hanoi( n-1, intermédiaire, final, départ )
Fin Si
```

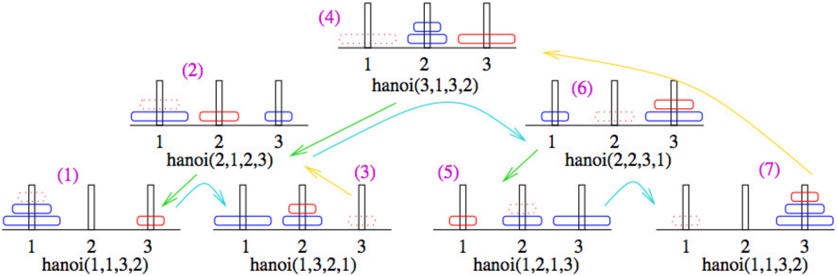
- étape 1 : déplacer les $n-1$ disques de la tour de départ vers la tour intermédiaire ➩ faire `Hanoi( n-1, départ, intermédiaire, final )`
- étape 2 : déplacer le dernier disque vers la destination finale ➩ faire `déplacer( départ , final )`
- étape 3 : déplacer les $n-1$ disques de la tour intermédiaire vers la tour finale ➩ faire `Hanoi( n-1, intermédiaire, final, départ )`

Fonctionnement de la récursivité multiple

Même mécanisme que la récursivité simple : utilisation par l'ordinateur de la pile système

➯ Mais un arbre d'appels récursifs à la place d'une simple chaîne

Exemple : execution de Hanoi(3,1,3,2)



Exercice

La suite de Fibonacci

- un petit peu d'histoire
  - découverte vers 1202 par Léonard de Pise
  - problème de la prolifération des lapins : possédant au départ un couple de lapins, combien de couples de lapins obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ?
- la formule : $u_1 = u_2 = 1$, et pour tout n entier : $u_n = u_{n-1} + u_{n-2}$

Ecrire l'algorithme permettant de calculer la suite de Fibonacci pour un entier n en entrée.

Donner la trace de l'exécution de cet algorithme (sous forme d'arbre des appels récursifs) pour $n = 4$. Pour chaque appel, vous donnerez également le contenu de la pile systéme.

Précautions à prendre

Attention aux appels récursifs infinis ....

Fonction F(n)

Entrée: un entier $n >$ -2
1: **Retourner** F(n-1)

Fonction P(n)

Entrée: un entier $n >$ 0
1: **Si** n = 0 **Alors**
2: **Retourner** 1
3: **Fin Si**
4: **Retourner** P(n+1)

- Définir une condition terminale
  - une condition terminale = un cas particulier pour lequel il n'y a pas d'autre appel récursif
  - p.ex. ajouter au début de F : **Si** n= -1 **Alors Retourner** 0 **Fin Si**
- Vérifier la terminaison de l'algorithme
  - aucune solution automatique pour vérifier la terminaison d'un algorithme
  - regarder au cas par cas (l'utilisation de la récurrence peut aider)

Ecrire un algorithme récursif

Un algorithme récursif doit comporter :

- un cas d'arrêt dans lequel aucun autre appel n'est effectué
  - conseil : la mettre au début de l'algorithme récursif
- un cas général dans lequel un ou plusieurs autres appels sont effectués
  - l'enchaînement des appels doit conduire au critére d'arrêt

Fonction Fact(n)

Entrée: un entier $n \geq 0$
Sortie: l'entier $n!$

```
Si n = 0 Alors
    Retourner 1
Fin Si
res ← Fact( n-1) . n
Retourner res
```

Récursif ou itératif ?

Les algorithmes itératifs peuvent toujours s'écrire sous forme récursive et inversement (mais dans ce dernier cas il faut certaines fois recoder l'utilisation d'une pile).

Avantage de la récursivité :

- simplicité d'expression des algorithmes
- parfois pas d'autres solutions

Inconvénient de la récursivité :

- des précautions à prendre
- peut être plus coûteux
  - taille mémoire de la pile
  - opérations sur la pile

Conseils :

- à utiliser pour des problèmes typiquement récursifs, ne pouvant être résolus de façon itérative
- éviter d'utiliser la récursivité lorsqu'on peut la remplacer par une définition itérative, à moins de bénéficier d'un gain considérable en simplicité.

Application aux algorithmes de tri

Problématique :

- étant donné une structure linéaire (tableau, liste,...) contenant des valeurs d'un type ordonné (entiers, réels, chaîne de caractères, ...), il faut trier les éléments en ordre croissant (ou décroissant)

Des dizaines d'algorithmes répondant à ce problème, mais des approches très différentes en fonction des algorithmes

- itératif, récursifs, ...

Tri sélection

Principe :

- parcourir le tableau pour trouver le plus grand élément, une fois arrivée au bout du tableau placer cet élément par permutation, et recommencer sur le reste du tableau

Procedure TriSelection(element[] tab)

Entrée: un tableau d'éléments à trier

Sortie: le tableau trié

$i \leftarrow tab.longueur - 1$
Tant que $i > 0$ **faire**
    $max \leftarrow 0$
    Pour j de 1 à i **faire**
        Si $tab[j] > tab[max]$ **Alors**
            $max \leftarrow j$
        Fin Si
    Fin Pour
    $temp \leftarrow tab[max]$
    $tab[max] \leftarrow tab[i]$
    $tab[i] \leftarrow temp$
    $i \leftarrow i - 1$
Fin Tant que

Remarque : i représente la dernière case à traiter

Tri sélection

Exemple : trier en ordre croissant le tableau

701	17	2	268	415	45	45	102

Première itération sur i (i.e. i = tab.longueur – 1 = 7)

- recherche du plus grand élément

	max							
j=1	701	$\boxed{17}$	2	268	415	45	45	102

	max							
j=2	701	17	$\boxed{2}$	268	415	45	45	102

	max							
j=3	701	17	2	$\boxed{268}$	415	45	45	102

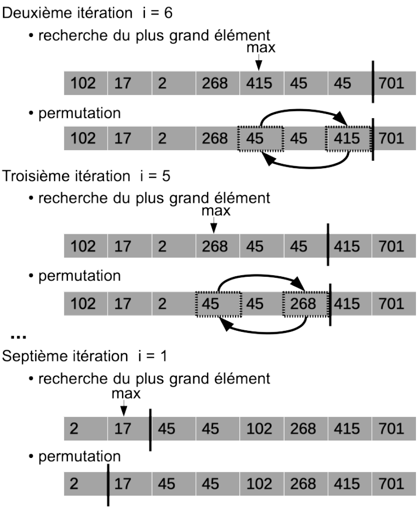
...

	max							
j=7	701	17	2	268	415	45	45	$\boxed{102}$

- permutation

102	17	2	268	415	45	45	701

Tri sélection



Tri fusion

Principe :

- découper le tableau en deux, trier chacune des parties (appels récursifs), puis fusionner
- algorithme récursif avec pour cas d'arrêt "si le tableau a un élément, il est trié"

Méthode principale

Procedure triFusion(element[] tab)
Entrée: un tableau d'éléments à trier
Sortie: le tableau trié

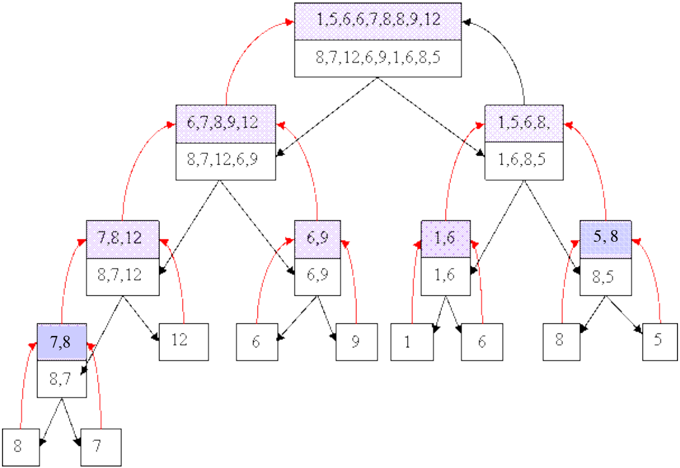
```
mergeSort(tab, 0, longueur( tab)-1 )
```

Méthode récursive

Procedure mergeSort(element[] tab, entier i, entier j)
Entrée: une partition du tableau tab située entre les indices i et j
Sortie: la partition triée

```
Si i < j Alors
    mergeSort(tab, i, (i+j)/2 )
    mergeSort( tab, (i+j)/2+1, j)
    fusion( tab, i, (i+j)/2, j )
Fin Si
```

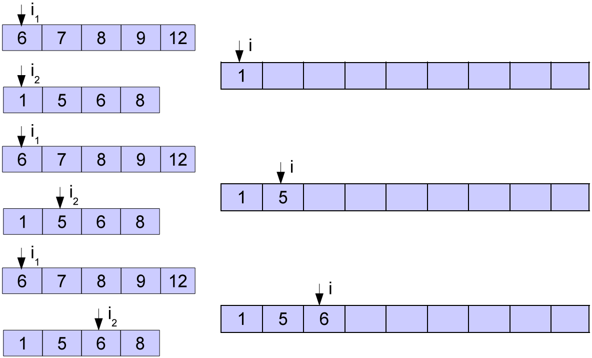
Tri fusion


1,5,6,6,7,8,8,9,12
8,7,12,6,9,1,6,8,5
6,7,8,9,12
8,7,12,6,9
1,5,6,8,
1,6,8,5
7,8,12
8,7,12
6,9
6,9
1,6
1,6
5, 8
8,5
7,8
8,7
12
6
9
1
6
8
5
8
7

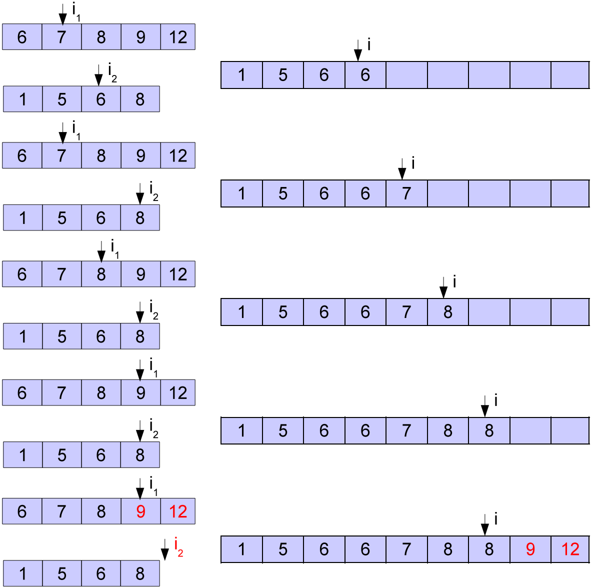
Tri fusion

Principe de la fusion :

- prend en entrée deux tableaux triés et réunis les éléments dans un tableau résultat de telle sorte que celui-ci soit trié
- parcourt les deux tableaux en parallèle, et insère au fur et à mesure les éléments dans le tableau résultat



Tri fusion



i1
6 7 8 9 12
i2
1 5 6 8
i
1 5 6 6
i1
6 7 8 9 12
i2
1 5 6 8
i
1 5 6 6 7
i1
6 7 8 9 12
i2
1 5 6 8
i
1 5 6 6 7 8
i1
6 7 8 9 12
i2
1 5 6 8
i
1 5 6 6 7 8 8
i1
6 7 8 9 12
i2
1 5 6 8
i
1 5 6 6 7 8 8 9 12

Tri fusion

Procedure fusion(element[] T, element[] T1, element[] T2, entier n1, entier n2)

Entrée: T le tableau résultat, T1 et T2 deux tableaux, de taille n1 et n2 , triés à fusionner

Sortie: le tableau T fusionné et trié

$i \leftarrow 0$; $i_1 \leftarrow 0$; $i_2 \leftarrow 0$

Tant que $i_1 < n1$ **et** $i_2 < n2$ **faire**

  Si $T1[i_1] < T2[i_2]$ **Alors**

    $T[i] \leftarrow T1[i_1]$

    $i_1 \leftarrow i_1 + 1$

  Sinon

    $T[i] \leftarrow T2[i_2]$

    $i_2 \leftarrow i_2 + 1$

  Fin Si

  $i \leftarrow i + 1$

Fin Tant que

Si $i_1 < n1$ **Alors**

  Pour j **de** i_1 **à** $n1 - 1$ **faire**

    $T[i] \leftarrow T1[j]$

    $i \leftarrow i + 1$

  Fin Pour

Sinon Si $i_2 < n2$ **Alors**

  Pour j **de** i_2 **à** $n2 - 1$ **faire**

    $T[i] \leftarrow T2[j]$

    $i \leftarrow i + 1$

  Fin Pour

Fin Si

Tri fusion

Amélioration de la fusion : plutôt que de passer trois tableaux en paramètres, travailler sur le tableau initial

---

Procedure fusion(element[] T, entier deb, entier mid, entier fin)

Entrée: **T le tableau initial triés entre deb et mid, et entre mid+1 et fin**
Sortie: **le tableau T trié entre deb et fin**

$i \leftarrow 0$; $i_1 \leftarrow deb$; $i_2 \leftarrow mid + 1$
Tant que $i_1 \leq mid$ **et** $i_2 \leq fin$ **faire**
    Si $T[i_1] < T[i_2]$ **Alors**
        $temp[i] \leftarrow T[i_1]$
        $i_1 \leftarrow i_1 + 1$
    Sinon
        $temp[i] \leftarrow T[i_2]$
        $i_2 \leftarrow i_2 + 1$
    Fin Si
    $i \leftarrow i + 1$
Fin Tant que
Si $i_1 < mid + 1$ **Alors**
    Pour j **de** i_1 **à** mid **faire**
        $temp[i] \leftarrow T[j]$
        $i \leftarrow i + 1$
    Fin Pour
Sinon Si $i_2 < fin + 1$ **Alors**
    Pour j **de** i_2 **à** fin **faire**
        $temp[i] \leftarrow T[j]$
        $i \leftarrow i + 1$
    Fin Pour
Fin Si
$k \leftarrow 0$
Pour i **de** deb **à** fin **faire**
    $T[i] \leftarrow temp[k]$; $k \leftarrow k + 1$
Fin Pour