# Algorithmique procédurale

## La récursivité

Équipe pédagogique

CY Tech

# Bibliographie - Sitographie


- Houcine Senoussi
- Cormen T.H., Leiserson C.E., Rivest R.L. et Stein C. Algorithmique. Dunod. 2010, 1188 pages.
- Laurence Pilard (Université de Versailles)
- Florent Hivert : `https://www.lri.fr/~hivert/COURS/CFA-L3/02-Recursivite.pdf`

# Introduction

- Les fonctions récursives jouent un rôle important en informatique.

# Introduction

- Les fonctions récursives jouent un rôle important en informatique.
- On parle de définition récursive lorsqu'un terme est décrit à partir de lui-même.

# Introduction

- Les fonctions récursives jouent un rôle important en informatique.
- On parle de définition récursive lorsqu'un terme est décrit à partir de lui-même.
- Exemple : une matriochka est une poupée russe dans laquelle est contenue une autre matriochka qui en contient une autre et ainsi de suite.

# Introduction

- Les fonctions récursives jouent un rôle important en informatique.
- On parle de définition récursive lorsqu'un terme est décrit à partir de lui-même.
- Exemple : une matriochka est une poupée russe dans laquelle est contenue une autre matriochka qui en contient une autre et ainsi de suite.
- La définition récursive de la matriochka est infinie car on ne voit pas comment elle s'achève.

# Introduction

- Les fonctions récursives jouent un rôle important en informatique.
- On parle de définition récursive lorsqu'un terme est décrit à partir de lui-même.
- Exemple : une matriochka est une poupée russe dans laquelle est contenue une autre matriochka qui en contient une autre et ainsi de suite.
- La définition récursive de la matriochka est infinie car on ne voit pas comment elle s'achève.
- Pour être calculables, les définitions récursives ont besoin d'une condition d'arrêt.

# Introduction

- Les fonctions récursives jouent un rôle important en informatique.
- On parle de définition récursive lorsqu'un terme est décrit à partir de lui-même.
- Exemple : une matriochka est une poupée russe dans laquelle est contenue une autre matriochka qui en contient une autre et ainsi de suite.
- La définition récursive de la matriochka est infinie car on ne voit pas comment elle s'achève.
- Pour être calculables, les définitions récursives ont besoin d'une condition d'arrêt.
- Exemple : il existe toujours une toute petite matriochka qu'on ne peut pas ouvrir.

# Introduction

- Principe de la récursivité : pour résoudre le problème, un algorithme fait appel à lui même, une ou plusieurs fois, directement ou indirectement. Chaque appel correspond à la résolution d'un sous-problème similaire au problème de départ.

# Introduction

- Algorithmiquement : La programmation récursive est une technique de programmation qui remplace les instructions de boucle (while, for, etc.) par des appels de fonction.

- Algorithmiquement : La programmation récursive est une technique de programmation qui remplace les instructions de boucle (tant que, pour, ...) par des appels de fonction.

Exemple de définition de fonction qui se rappelle elle même :

***Procédure* boucle()**

*Début*
  boucle()
*Fin*

## Introduction

On peut en profiter pour faire quelque chose :

**_Procédure_ boucle()**

```
Début
  écrire("Je tourne") ;
  boucle()
Fin
```

C'est ce qu'on appelle une récursivité simple.

# Introduction

Il faut encore ajouter un mécanisme de test d'arrêt. Ex. écrire 100 fois "Je tourne" : on a besoin d'un compteur. On choisit ici de le passer d'un appel de fonction à l'autre comme un paramètre.

***Procédure* boucle(n : Entier)**

```
Début
  si (n < 100) alors
    écrire("Je tourne");
    boucle(n+1)
  finsi
Fin
```

On lance par boucle(0).

# Introduction

- Principe de la récursivité : pour résoudre le problème, un algorithme fait appel à lui même, une ou plusieurs fois, directement ou indirectement. Chaque appel correspond à la résolution d'un sous-problème similaire au problème de départ.
- Exemple 1 : un algorithme récursif pour le calcul de la factorielle d'un entier $n$ s'appelle lui-même pour calculer la factorielle de $n-1$.

# Introduction

- Principe de la récursivité : pour résoudre le problème, un algorithme fait appel à lui même, une ou plusieurs fois, directement ou indirectement. Chaque appel correspond à la résolution d'un sous-problème similaire au problème de départ.
- Exemple 1 : un algorithme récursif pour le calcul de la factorielle d'un entier $n$ s'appelle lui-même pour calculer la factorielle de $n-1$.
- Exemple 2 : pour trier un tableau $t$, un algorithme récursif de tri fait appel à lui même deux fois : une fois pour le tri de la moitié gauche de $t$ et une fois pour le tri de sa moitié droite.

# Introduction : Algorithme récursif

L'expression d'algorithmes sous forme récursive permet des descriptions concises, qui se prêtent bien à des démonstrations par récurrence.

# Introduction : Algorithme récursif

L'expression d'algorithmes sous forme récursive permet des descriptions concises, qui se prêtent bien à des démonstrations par récurrence.

## Principe

Le principe est d'utiliser, pour décrire l'algorithme sur une donnée $D$, l'algorithme lui-même appliqué à $\bar{D}$, un sous-ensemble de $D$ ($\bar{D} \subset D$) ou à une donnée $D'$ plus petite.

# Introduction : Algorithme récursif

L'expression d'algorithmes sous forme récursive permet des descriptions concises, qui se prêtent bien à des démonstrations par récurrence.

## Principe

Le principe est d'utiliser, pour décrire l'algorithme sur une donnée $D$, l'algorithme lui-même appliqué à $\bar{D}$, un sous-ensemble de $D$ ($\bar{D} \subset D$) ou à une donnée $D'$ plus petite.

## Conception d'un algorithme récursif

La conceptions d'algorithmes récursifs doit se faire en suivant quelques règles de bon sens :

# Introduction : Algorithme récursif

L'expression d'algorithmes sous forme récursive permet des descriptions concises, qui se prêtent bien à des démonstrations par récurrence.

## Principe

Le principe est d'utiliser, pour décrire l'algorithme sur une donnée $D$, l'algorithme lui-même appliqué à $\bar{D}$, un sous-ensemble de $D$ ($\bar{D} \subset D$) ou à une donnée $D'$ plus petite.

## Conception d'un algorithme récursif

La conceptions d'algorithmes récursifs doit se faire en suivant quelques règles de bon sens :

- il faut s'assurer qu'on ne ré-applique pas l'algorithme à des données plus grandes et

# Introduction : Algorithme récursif

L'expression d'algorithmes sous forme récursive permet des descriptions concises, qui se prêtent bien à des démonstrations par récurrence.

## Principe

Le principe est d'utiliser, pour décrire l'algorithme sur une donnée $D$, l'algorithme lui-même appliqué à $\bar{D}$, un sous-ensemble de $D$ ($\bar{D} \subset D$) ou à une donnée $D'$ plus petite.

## Conception d'un algorithme récursif

La conceptions d'algorithmes récursifs doit se faire en suivant quelques règles de bon sens :

- il faut s'assurer qu'on ne ré-applique pas l'algorithme à des données plus grandes et
- qu'on a bien un test de *terminaison*, qui correspond à un cas où la donnée est suffisamment élémentaire pour être traitée directement sans ré-application de l'algorithme.

# Exemple : Algorithme récursif pour la factorielle

L'idée de la récursivité est d'utiliser une définition équivalente, à savoir une suite récurrente :

$$n! = \begin{cases} 1 \text{ si } n = 0 \\ n \times (n-1)! \text{ sinon} \end{cases}$$

# Exemple : Algorithme récursif pour la factorielle

L'idée de la récursivité est d'utiliser une définition équivalente, à savoir une suite récurrente :

$$n! = \begin{cases} 1 \text{ si } n = 0 \\ n \times (n-1)! \text{ sinon} \end{cases}$$

Ceci peut alors se traduire par le programme suivant en pseudo-langage :

***Fonction* factorielle(n : Entier) : Entier**

```
Début
   Si (n≤1) Alors Retourner 1
   Sinon Retourner n*factorielle(n-1)
   FinSi
Fin
```

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

  4*factorielle(3) $= ?$

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

    4*factorielle(3) $= ?$

    Appel à factorielle(3)

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

    4*factorielle(3) $= ?$

    Appel à factorielle(3)

        3*factorielle(2) $= ?$

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

  4*factorielle(3) $= ?$

  Appel à factorielle(3)

    3*factorielle(2) $= ?$

    Appel à factorielle(2)

      2*factorielle(1) $= ?$

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

    4*factorielle(3) $= ?$

    Appel à factorielle(3)

        3*factorielle(2) $= ?$

        Appel à factorielle(2)

            2*factorielle(1) $= ?$

            Appel à factorielle(1)

                1*factorielle(0) $= ?$

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)
    4*factorielle(3) $= ?$
    Appel à factorielle(3)
        3*factorielle(2) $= ?$
        Appel à factorielle(2)
            2*factorielle(1) $= ?$
            Appel à factorielle(1)
                1*factorielle(0) $= ?$
                Appel à factorielle(0)
                Retour de la valeur 1

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)
    4*factorielle(3) = ?
    Appel à factorielle(3)
        3*factorielle(2) = ?
        Appel à factorielle(2)
            2*factorielle(1) = ?
            Appel à factorielle(1)
                1*factorielle(0) = ?
                Appel à factorielle(0)
                Retour de la valeur 1
                1*1

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)
    4*factorielle(3) $= ?$
    Appel à factorielle(3)
        3*factorielle(2) $= ?$
        Appel à factorielle(2)
            2*factorielle(1) $= ?$
            Appel à factorielle(1)
                1*factorielle(0) $= ?$
                Appel à factorielle(0)
                Retour de la valeur 1
                1*1
            Retour de la valeur 1

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)
    4*factorielle(3) = ?
    Appel à factorielle(3)
        3*factorielle(2) = ?
        Appel à factorielle(2)
            2*factorielle(1) = ?
            Appel à factorielle(1)
                1*factorielle(0) = ?
                Appel à factorielle(0)
                Retour de la valeur 1
                1*1
            Retour de la valeur 1
            2*1

## Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

- 4*factorielle(3) $= ?$
- Appel à factorielle(3)
  - 3*factorielle(2) $= ?$
  - Appel à factorielle(2)
    - 2*factorielle(1) $= ?$
    - Appel à factorielle(1)
      - 1*factorielle(0) $= ?$
      - Appel à factorielle(0)
      - Retour de la valeur 1
      - 1*1
    - Retour de la valeur 1
    - 2*1
  - Retour de la valeur 2

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)
- 4*factorielle(3) $= ?$
- Appel à factorielle(3)
  - 3*factorielle(2) $= ?$
  - Appel à factorielle(2)
    - 2*factorielle(1) $= ?$
    - Appel à factorielle(1)
      - 1*factorielle(0) $= ?$
      - Appel à factorielle(0)
      - Retour de la valeur 1
      - 1*1
    - Retour de la valeur 1
    - 2*1
  - Retour de la valeur 2
  - 3*2

## Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

    4*factorielle(3) $= ?$

    Appel à factorielle(3)

        3*factorielle(2) $= ?$

        Appel à factorielle(2)

            2*factorielle(1) $= ?$

            Appel à factorielle(1)

                1*factorielle(0) $= ?$

                Appel à factorielle(0)

                Retour de la valeur 1

                1*1

            Retour de la valeur 1

            2*1

        Retour de la valeur 2

        3*2

    Retour de la valeur 6

# Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)
- 4*factorielle(3) = ?
- Appel à factorielle(3)
  - 3*factorielle(2) = ?
  - Appel à factorielle(2)
    - 2*factorielle(1) = ?
    - Appel à factorielle(1)
      - 1*factorielle(0) = ?
      - Appel à factorielle(0)
      - Retour de la valeur 1
      - 1*1
    - Retour de la valeur 1
    - 2*1
  - Retour de la valeur 2
  - 3*2
- Retour de la valeur 6
- 4*6

## Exemple : Algorithme récursif pour la factorielle

Appel à factorielle(4)

- 4*factorielle(3) $= ?$
- Appel à factorielle(3)
  - 3*factorielle(2) $= ?$
  - Appel à factorielle(2)
    - 2*factorielle(1) $= ?$
    - Appel à factorielle(1)
      - 1*factorielle(0) $= ?$
      - Appel à factorielle(0)
      - Retour de la valeur 1
      - 1*1
    - Retour de la valeur 1
    - 2*1
  - Retour de la valeur 2
  - 3*2
- Retour de la valeur 6
- 4*6

Retour de la valeur 24

# Remarque : pile d'exécution

## Définition

La pile d'exécution d'un programme récursif est un emplacement mémoire qui est destiné à mémoriser les paramètres, les variables locales ainsi que l'adresse de retour de chaque fonction en cours d'exécution.

# Remarque : pile d'exécution

## Définition

La pile d'exécution d'un programme récursif est un emplacement mémoire qui est destiné à mémoriser les paramètres, les variables locales ainsi que l'adresse de retour de chaque fonction en cours d'exécution.

- Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré, premier sorti.

# Remarque : pile d'exécution

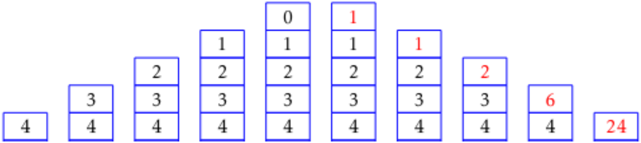
## Définition

La pile d'exécution d'un programme récursif est un emplacement mémoire qui est destiné à mémoriser les paramètres, les variables locales ainsi que l'adresse de retour de chaque fonction en cours d'exécution.

- Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré, premier sorti.
- Attention, l'exécution de la solution récursive peut provoquer des débordements de la mémoire.

# Exemple : Algorithme récursif pour la factorielle

La pile d'exécution de la fonction factorielle va évoluer de la façon suivante :



# Récursivités terminale et non terminale

Considérons maintenant les deux fonctions suivantes :

**_Fonction_ factorielle(n : Entier) : Entier**

*Début*
  Si (n≤1) Alors Retourner 1
  Sinon Retourner n*factorielle(n-1)
  FinSi
*Fin*

et

**_Fonction_ factorielleRT(n : Entier, acc : Entier) : Entier**

*Début*
  Si (n≤1) Alors Retourner acc
  Sinon Retourner factorielleRT(n-1, acc*n)
*Fin*

# Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.

# Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*

# Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*
  - Appel de la seconde : *factorielleRT(n,1)*

## Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*
  - Appel de la seconde : *factorielleRT(n,1)*
- On remarque que la dernière instruction de la fonction est

## Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*
  - Appel de la seconde : *factorielleRT(n,1)*
- On remarque que la dernière instruction de la fonction est
  - l'appel récursif dans la seconde,

# Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*
  - Appel de la seconde : *factorielleRT(n,1)*
- On remarque que la dernière instruction de la fonction est
  - l'appel récursif dans la seconde,
  - une autre instruction (une multiplication) dans la première.

# Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*
  - Appel de la seconde : *factorielleRT(n,1)*
- On remarque que la dernière instruction de la fonction est
  - l'appel récursif dans la seconde,
  - une autre instruction (une multiplication) dans la première.
- Nous avons donc :

# Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*
  - Appel de la seconde : *factorielleRT(n,1)*
- On remarque que la dernière instruction de la fonction est
  - l'appel récursif dans la seconde,
  - une autre instruction (une multiplication) dans la première.
- Nous avons donc :
  - une récursion terminale dans la seconde,

# Récursivités terminale et non terminale

- Les deux fonctions permettent de calculer la factorielle.
  - Appel de la première : *factorielle(n)*
  - Appel de la seconde : *factorielleRT(n,1)*
- On remarque que la dernière instruction de la fonction est
  - l'appel récursif dans la seconde,
  - une autre instruction (une multiplication) dans la première.
- Nous avons donc :
  - une récursion terminale dans la seconde,
  - une récursion non terminale dans la première.

# Récursivités terminale et non terminale

**Définition :**

La récursivité est dite terminale lorsque l'appel (récursif) est la dernière instruction de la fonction.

# Récursivités terminale et non terminale

**Définition :**

La récursivité est dite terminale lorsque l'appel (récursif) est la dernière instruction de la fonction. C'est-à-dire qu'un algorithme récursif simple est terminal lorsque l'appel récursif est le dernier calcul effectué pour obtenir le résultat. Il n'y a pas de "calcul en attente".
On utilise un accumulateur, passé en paramètre, pour calculer le résultat au fur et à mesure des appels récursifs. La valeur de retour du cas de base devient la valeur initiale de l'accumulateur et lors d"un appel récursif, le "calcul en attente" sert à calculer la valeur suivante de l'accumulateur.

# Récursivités terminale et non terminale

- Avantages : L'avantage est qu'il n'y a rien à mémoriser dans la pile. La fonction passe la main à la fonction appelée (elle même) sans avoir besoin de la reprendre. L'implémentation peut donc se faire sans ajouter un élément à la '**pile des appels**'. Le contexte de la fonction appelante n'a pas besoin d'être sauvegardé $\rightarrow$ réduction de l'espace mémoire utilisé. De plus les compilateurs récents savent optimiser ce type d'algorithme.

# Diviser pour régner : Principe

- L'approche Diviser pour régner consiste à :

# Diviser pour régner : Principe

- L'approche Diviser pour régner consiste à :
  1. **[Diviser]** : Séparer le problème à résoudre en sous-problèmes semblables au problème initial mais de taille plus petite.

# Diviser pour régner : Principe

- L'approche Diviser pour régner consiste à :
  1. **[Diviser]** : Séparer le problème à résoudre en sous-problèmes semblables au problème initial mais de taille plus petite.
  2. **[Régner]** : Résoudre récursivement les sous problèmes.

# Diviser pour régner : Principe

- L'approche Diviser pour régner consiste à :
  1. **[Diviser]** : Séparer le problème à résoudre en sous-problèmes semblables au problème initial mais de taille plus petite.
  2. **[Régner]** : Résoudre récursivement les sous problèmes.
  3. **[Combiner]** : Combiner les solutions des sous-problèmes pour produire la solution du problème initial.

# Exemple : Tri par fusion

- **Principe :** Pour trier un tableau de taille *n* :

# Exemple : Tri par fusion

- **Principe :** Pour trier un tableau de taille $n$ :
    1. Le diviser en deux sous tableaux de taille (presque) égale.

# Exemple : Tri par fusion

- **Principe :** Pour trier un tableau de taille $n$ :
  1. Le diviser en deux sous tableaux de taille (presque) égale.
  2. Trier récursivement les deux sous tableaux.

# Exemple : Tri par fusion

- **Principe :** Pour trier un tableau de taille $n$ :
  1. Le diviser en deux sous tableaux de taille (presque) égale.
  2. Trier récursivement les deux sous tableaux.
  3. **Fusionner** les deux sous tableaux triés.

# Tri par fusion-2

***Algorithme* Algo-Tri-Fusion**

*Début*

...

Tri-Fusion(A, 1, n)

*Fin*

# Tri par fusion-3

***Procedure* Tri-Fusion(E/S A, p, r)**

*Début*

Si (p $\leq$ r) Alors

$q = (p + r)/2$

Tri-Fusion(A, p, q)

Tri-Fusion(A, q+1, r)

Fusion(A, p, q, r)

*Fin*

# Tri par fusion-4

***Procedure* Fusion(E/S A, p, q, r)**

*variables*

....

*Début*

n1 := q-p+1
n2 := r-q
L := créerTableau(n1)
R := créerTableau(n2)
Pour i de 1 à n1 L[i] := A[p+i-1]
Pour j de 1 à n2 R[j] := A[q+j]
L[n1] := INF
R[n2] := INF

# Tri par fusion-5

**(* Suite de la procédure Fusion *)**

```
i := 1
j := 1
Pour k de p à r
 Si L[i] ≤ R[j]
  A[k] := L[i]
  i := i+1
 Sinon
  A[k] := R[j]
  j := j+1
Fin
```

# Analyse des algorithmes - Correction des algorithmes : Méthode générale

- Pour démontrer la correction d'un algorithme 'Diviser pour régner' on est souvent amené à combiner :

# Analyse des algorithmes - Correction des algorithmes : Méthode générale

- Pour démontrer la correction d'un algorithme 'Diviser pour régner' on est souvent amené à combiner :
  - une récurrence pour les parties récursives (Régner),

# Analyse des algorithmes - Correction des algorithmes : Méthode générale

- Pour démontrer la correction d'un algorithme 'Diviser pour régner' on est souvent amené à combiner :
  - une récurrence pour les parties récursives (Régner),
  - d'autres méthodes (notamment les invariants de boucle) pour les autres parties (Diviser, Combiner).

# Exemple : Tri par fusion

- la partie 'Diviser' se réduit au calcul de la position du milieu du tableau.

# Exemple : Tri par fusion

- la partie 'Diviser' se réduit au calcul de la position du milieu du tableau.
- les parties 'Régner' et 'Combiner' sont analysées de la manière suivante :

# Exemple : Tri par fusion

- la partie 'Diviser' se réduit au calcul de la position du milieu du tableau.
- les parties 'Régner' et 'Combiner' sont analysées de la manière suivante :
  - Cas de base : tableau réduit à un seul élément. Trié.

# Exemple : Tri par fusion

- la partie 'Diviser' se réduit au calcul de la position du milieu du tableau.
- les parties 'Régner' et 'Combiner' sont analysées de la manière suivante :
  - Cas de base : tableau réduit à un seul élément. Trié.
  - Supposons que pour $n \geq 2$ l'algorithme trie correctement les tableaux de taille $m \leq n/2$.

# Exemple : Tri par fusion

- la partie 'Diviser' se réduit au calcul de la position du milieu du tableau.
- les parties 'Régner' et 'Combiner' sont analysées de la manière suivante :
  - Cas de base : tableau réduit à un seul élément. Trié.
  - Supposons que pour $n \geq 2$ l'algorithme trie correctement les tableaux de taille $m \leq n/2$.
  - Il nous faut déduire que l'algorithme trie correctement un tableau de taille $n$.

# Exemple : Tri par fusion

- la partie 'Diviser' se réduit au calcul de la position du milieu du tableau.
- les parties 'Régner' et 'Combiner' sont analysées de la manière suivante :
  - Cas de base : tableau réduit à un seul élément. Trié.
  - Supposons que pour $n \geq 2$ l'algorithme trie correctement les tableaux de taille $m \leq n/2$.
  - Il nous faut déduire que l'algorithme trie correctement un tableau de taille $n$.
    - Cela revient à démontrer la correction de la procédure 'Fusion'.

# Exemple : Tri par fusion-2

- Nous allons démontrer la correction de la procédure 'Fusion' par l'invariant de boucle suivant :

# Exemple : Tri par fusion-2

- Nous allons démontrer la correction de la procédure 'Fusion' par l'invariant de boucle suivant :
  - Au début de l'itération $k$ le tableau $A[p, .., k-1]$ contient les $k-p$ plus petits éléments de $L[1..n1+1]$ et $R[1..n2+1]$ en ordre trié. En outre, $L[i]$ et $R[j]$ sont les plus petits éléments de leurs tableaux à ne pas avoir été copiés dans $A$.

# Exemple : Tri par fusion-3

- **Initialisation :** Avant la première itération nous avons $k = p$.Le sous-tableau $A[p..k-1]$ est donc vide : il contient les '0=$k-p$ éléments' les plus petits de $L$ et $R$. Les tableaux $L$ et $R$ étant triés et $i = j = 1$, $L[i]$ et $R[j]$ sont effectivement les plus petits éléments de leur tableaux à ne pas être copiés dans $A$.

## Exemple : Tri par fusion-4

- **Conservation** : Supposons que la propriété soit vraie jusqu'à l'itération $k$. À l'itération suivante $k+1$, le plus petit des deux éléments $L[i]$ et $R[j]$ est ajoutée en position $k+1$. Le sous-tableau $A[p..k]$ contient donc les $k+1$ plus petits éléments triés et $L[i]$ et $R[j+1]$ contiennent les plus petit élément de leurs tableaux à ne pas avoir été copiés dans $A$ (noter que seule l'une des deux valeurs de $i$ et $j$ a été incrémentée).

# Exemple : Tri par fusion-5

- **Terminaison :** À la fin $k = r + 1$, $i = n1 + 1$ et $j = n2 + 1$. L'invariant de boucle s'écrit comme suit :
  - Le tableau $A[p..r]$ contient les $r - p + 1$ éléments les plus petits de $L$ et $R$ triés. Seuls les éléments 'artificiels' (sentinelles) INF n'ont pas été copiés dans $A$. Ces sentinelles sont les valeurs actuelles de $L[i]$ et $R[j]$.

# Analyse des algorithmes - Complexité : Méthode générale

- Le nombre d'opérations d'un algorithme récursif est souvent donné par une **équation de récurrence**.

# Analyse des algorithmes - Complexité : Méthode générale

- Le nombre d'opérations d'un algorithme récursif est souvent donné par une **équation de récurrence**.
- Cette équation décrit le nombre d'opérations de l'algorithme pour une taille de problème $n$ (problème initial) en fonction des nombres d'opérations de l'algorithme appliqué au(x) sous-problème(s) ($taille < n$).

- Supposons que :

- Supposons que :
  - le problème de taille $n$ est divisé en $a$ sous-problèmes de taille $n/b$.

- Supposons que :
  - le problème de taille $n$ est divisé en $a$ sous-problèmes de taille $n/b$.
  - le coût des opérations qui précèdent l'appel récursif soit $C(n)$.

- Supposons que :
  - le problème de taille $n$ est divisé en $a$ sous-problèmes de taille $n/b$.
  - le coût des opérations qui précèdent l'appel récursif soit $C(n)$.
  - le coût des opérations qui suivent l'appel récursif soit $D(n)$.

- Supposons que :
  - le problème de taille $n$ est divisé en $a$ sous-problèmes de taille $n/b$.
  - le coût des opérations qui précèdent l'appel récursif soit $C(n)$.
  - le coût des opérations qui suivent l'appel récursif soit $D(n)$.
  - pour les petites valeurs de $n$ ($n \leq c$) le nombre d'opérations est constant et on le note $\Theta(1)$.

- Supposons que :
    - le problème de taille $n$ est divisé en $a$ sous-problèmes de taille $n/b$.
    - le coût des opérations qui précèdent l'appel récursif soit $C(n)$.
    - le coût des opérations qui suivent l'appel récursif soit $D(n)$.
    - pour les petites valeurs de $n$ ($n \leq c$) le nombre d'opérations est constant et on le note $\Theta(1)$.
- L'équation de récurrence est donc $T(n) = \begin{cases} \Theta(1) \ si \ n \leq c \\ aT(n/b) + D(n) + C(n) \end{cases}$

- Appliquons cette méthode au tri par fusion.

- Appliquons cette méthode au tri par fusion.
- Nous prenons $c = 2$

# Analyse des algorithmes - Complexité : Tri par fusion

- Appliquons cette méthode au tri par fusion.
- Nous prenons $c = 2$
- La partie qui précède l'appel se contente de calculer le milieu du tableau, donc $C(n) = \Theta(1)$.

# Analyse des algorithmes - Complexité : Tri par fusion

- Appliquons cette méthode au tri par fusion.
- Nous prenons $c = 2$
- La partie qui précède l'appel récursif se contente de calculer le milieu du tableau, donc $C(n) = \Theta(1)$.
- Chaque étape divise le problème (taille $p$) en deux sous-problèmes de taille $p/2$ ($a = 2$ et $b = 2$).

## Analyse des algorithmes - Complexité : Tri par fusion

- Appliquons cette méthode au tri par fusion.
- Nous prenons $c = 2$
- La partie qui précède l'appel récursif se contente de calculer le milieu du tableau, donc $C(n) = \Theta(1)$.
- Chaque étape divise le problème (taille $p$) en deux sous-problèmes de taille $p/2$ ($a = 2$ et $b = 2$).
- La partie qui suit l'appel récursif est la fusion. Sa complexité $D(n)$ est en $\Theta(n)$ (voir détails ci-dessous).

## Analyse des algorithmes - Complexité : Tri par fusion

- Appliquons cette méthode au tri par fusion.
- Nous prenons $c = 2$
- La partie qui précède l'appel récursif se contente de calculer le milieu du tableau, donc $C(n) = \Theta(1)$.
- Chaque étape divise le problème (taille $p$) en deux sous-problèmes de taille $p/2$ ($a = 2$ et $b = 2$).
- La partie qui suit l'appel récursif est la fusion. Sa complexité $D(n)$ est en $\Theta(n)$ (voir détails ci-dessous).
- L'équation de récurrence est donc $T(n) = \begin{cases} \Theta(1) \ si \ n \leq 1 \\ 2T(n/2) + \Theta(n) \ sinon \end{cases}$

# Analyse des algorithmes - Complexité : Tri par fusion

- Appliquons cette méthode au tri par fusion.
- Nous prenons $c = 2$
- La partie qui précède l'appel récursif se contente de calculer le milieu du tableau, donc $C(n) = \Theta(1)$.
- Chaque étape divise le problème (taille $p$) en deux sous-problèmes de taille $p/2$ ($a = 2$ et $b = 2$).
- La partie qui suit l'appel récursif est la fusion. Sa complexité $D(n)$ est en $\Theta(n)$ (voir détails ci-dessous).
- L'équation de récurrence est donc $T(n) = \begin{cases} \Theta(1) \ si \ n \leq 1 \\ 2T(n/2) + \Theta(n) \ sinon \end{cases}$
- Il est aisé d'en déduire que la complexité du tri par fusion est en $\Theta(nlogn)$.

- Complexité de la procédure Fusion :
  - Nous avons trois boucles qui comportent chacune $\Theta(r - p)$ affectations (et autant de tests pour la troisième).

# Conclusion

- Rappels sur la récursivité.
- Récursivités terminale et non terminale.
- Approche 'Diviser pour régner' pour la conception des algorithmes.
- Analyse des algorithmes utilisant cette approche : correction & complexité.