

Organisation du cours

- 9 blocs, soit 9 vendredis.
 - ▶ Le matin, amphi, de **10h30 à 12h00**.
 - ▶ L'après midi, TP.
 - Gr1 -2, Salle info 34, J.C. Filliâtre - S. Lengrand
- Page du cours
www.enseignement.polytechnique.fr/informatique/INF421
et vos questions à Olivier.Bournez@polytechnique.fr et/ou
aux enseignants de l'équipe.
- Évaluation.
 - ▶ TP noté, le sixième, **11** décembre.
 - ▶ Contrôle à la fin,
 - ▶ Note de module : $\max(CC, \frac{3}{4}CC + \frac{1}{4}f(TP))$, avec f proche de l'identité.

Organisation du cours

- 9 blocs, soit 9 vendredis.
 - ▶ Le matin, amphi, de **10h30 à 12h00**.
 - ▶ L'après midi, TP.
 - Gr1 -2, Salle info 34, J.C. Filliâtre - S. Lengrand
- Page du cours
www.enseignement.polytechnique.fr/informatique/INF421
et vos questions à Olivier.Bournez@polytechnique.fr et/ou
aux enseignants de l'équipe.
- Évaluation.
 - ▶ TP noté, le sixième, **11** décembre.
 - ▶ Contrôle à la fin,
 - ▶ Note de module : $\max(CC, \frac{3}{4}CC + \frac{1}{4}f(TP))$, avec f proche de l'identité.
- Possibilité de tutorat: les lundis de 16h30 à 18h30, Salle info 36,
avec E. Nicolas.

Static/ non-Static

- Retour sur une question:

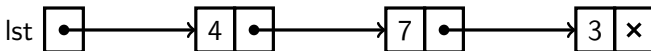
```
class MonInt {  
    int val;  
    static int abs(int x) {  
        if (x<0) return -x; else return x;}  
    int abs() {  
        if (val<0) return -val; else return val;}  
}
```

```
MonInt x = new MonInt();  
x.val = 2;  
System.out.println(MonInt.abs(-3)+"/"+x.abs());
```

Rappel: Liste

```
class Liste {  
    int contenu;  
    Liste suivant;  
    Liste (int x, Liste a) {  
        contenu = x;  
        suivant = a;  
    }  
}
```

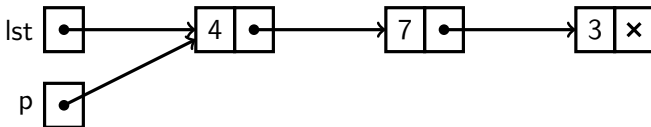
```
Liste lst = new Liste(4, new Liste (7, new Liste (3, null)));
```



Parcourir une liste

```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
// traiter p.contenu
}
```

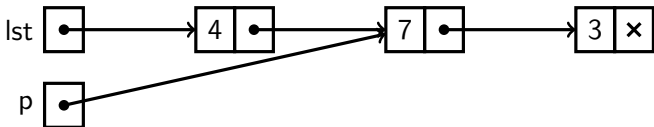
```
Liste p = ...;
while (p ≠ null) {
// traiter p.contenu
p = p.suivant;
}
```



Parcourir une liste

```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
  // traiter p.contenu
}
```

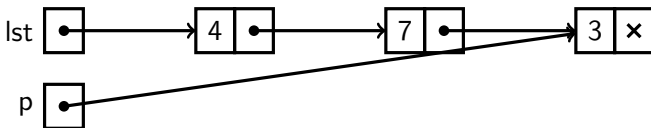
```
Liste p = ...;
while (p ≠ null) {
  // traiter p.contenu
  p = p.suivant;
}
```



Parcourir une liste

```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
  // traiter p.contenu
}
```

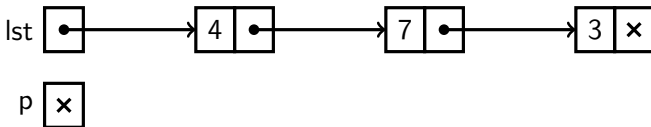
```
Liste p = ...;
while (p ≠ null) {
  // traiter p.contenu
  p = p.suivant;
}
```



Parcourir une liste

```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
// traiter p.contenu
}
```

```
Liste p = ...;
while (p ≠ null) {
// traiter p.contenu
p = p.suivant;
}
```



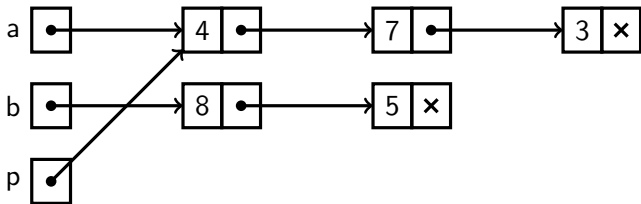
Concaténer deux listes (version itérative, mutable)

- Objectif:

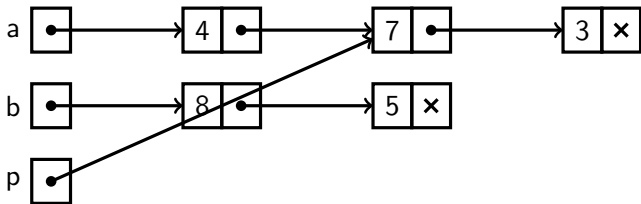
- ▶ ajouter les éléments de la liste b à la fin de ceux de la liste a.

```
static Liste nappend2(Liste a, Liste b) {  
    if (a==null)  
        return b;  
    Liste p;  
    for (p=a; p.suivant !=null; p=p.suivant)  
        {}  
    p.suivant = b;  
    return a;  
}
```

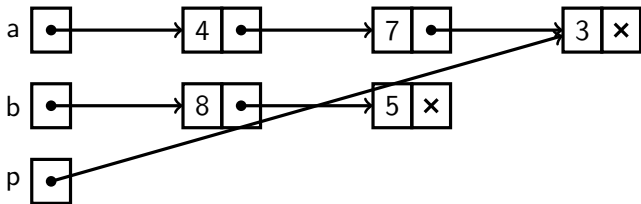
Concaténer deux listes (version itérative, mutable)



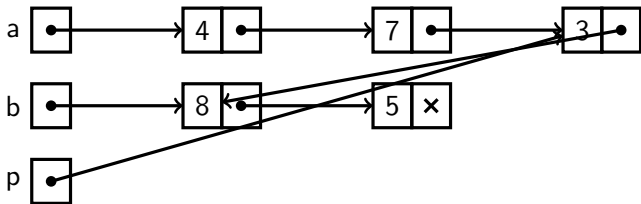
Concaténer deux listes (version itérative, mutable)



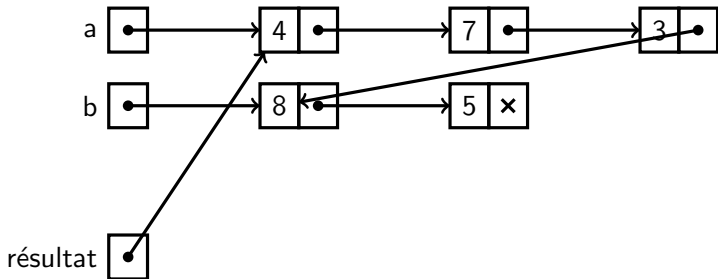
Concaténer deux listes (version itérative, mutable)



Concaténer deux listes (version itérative, mutable)



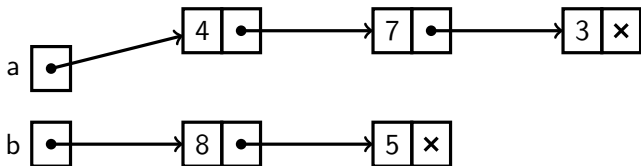
Concaténer deux listes (version itérative, mutable)



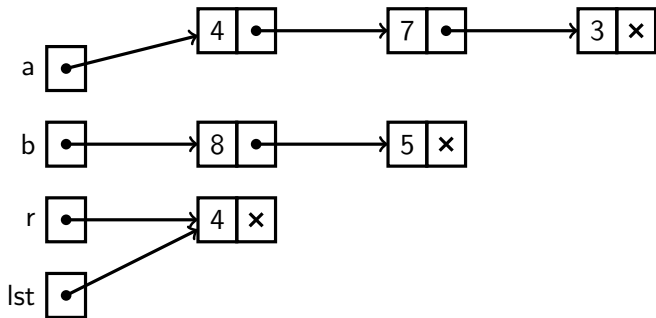
Concaténer deux listes (version itérative, persistante)

```
static Liste append2(Liste a, Liste b) {  
    if (a== null) return b;  
    Liste r = new Liste(a.contenu, null) ;  
    Liste lst = r;  
    for (a= a.suivant; a ≠ null; a = a.suivant) {  
        lst.suivant = new Liste(a.contenu, null);  
        lst = lst.suivant;  
    }  
    lst.suivant = b;  
    return r;  
}
```

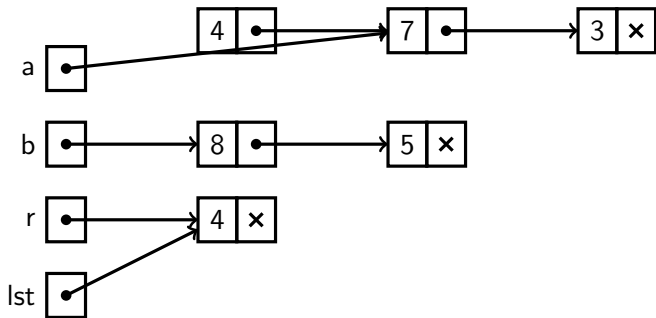
Concaténer deux listes (version itérative, persistante)



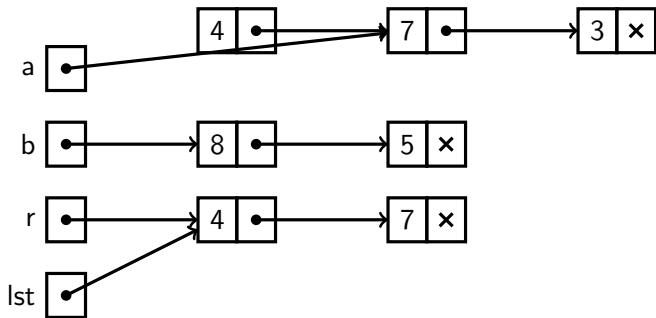
Concaténer deux listes (version itérative, persistante)



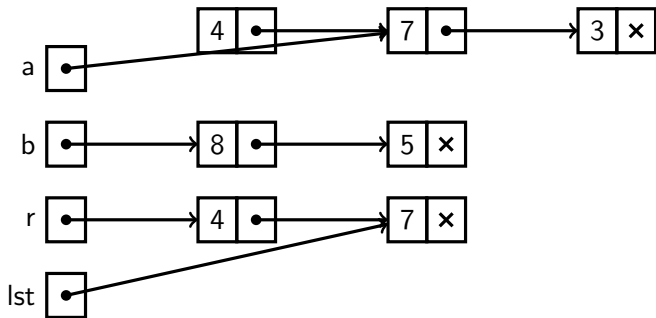
Concaténer deux listes (version itérative, persistante)



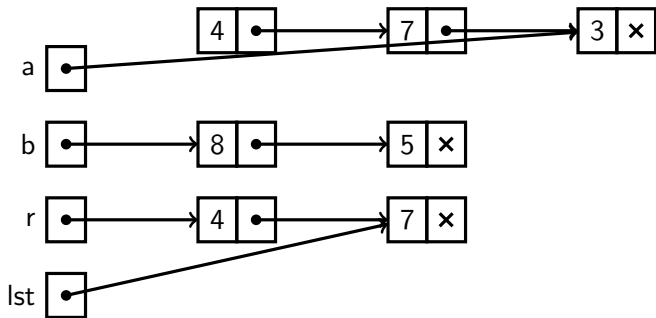
Concaténer deux listes (version itérative, persistante)



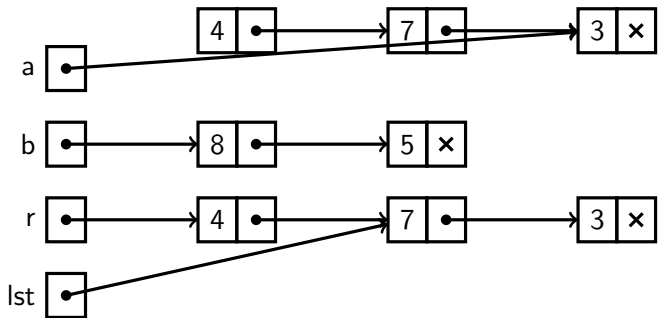
Concaténer deux listes (version itérative, persistante)



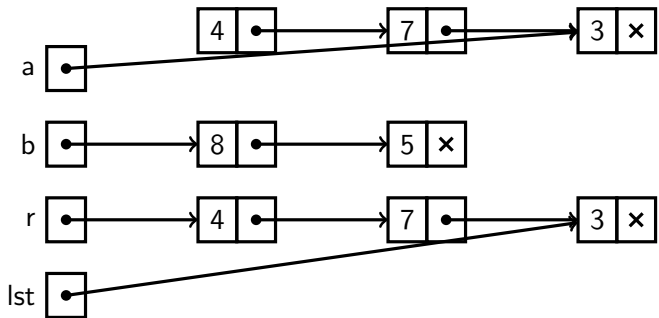
Concaténer deux listes (version itérative, persistante)



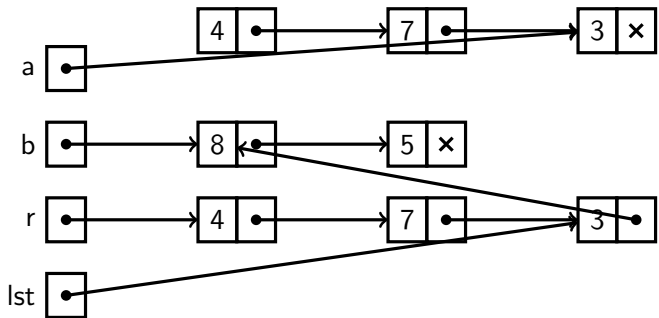
Concaténer deux listes (version itérative, persistante)



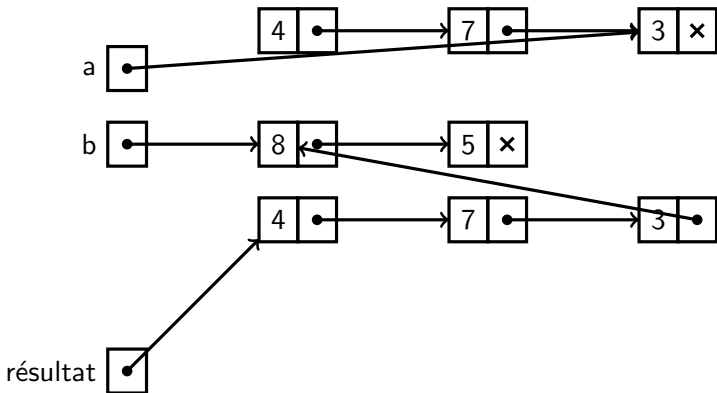
Concaténer deux listes (version itérative, persistante)



Concaténer deux listes (version itérative, persistante)



Concaténer deux listes (version itérative, persistante)



Concaténer deux listes (version récursive, persistante)

- Une liste d'entiers est solution de l'équation

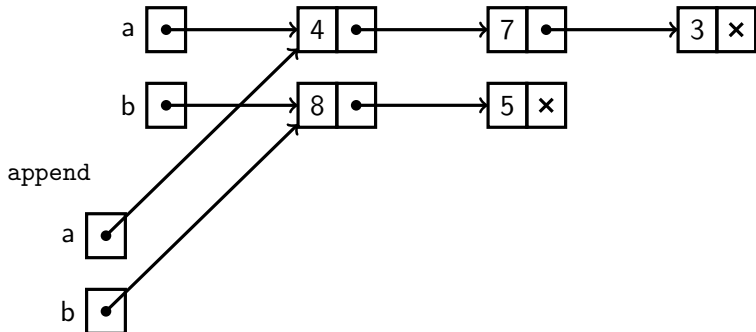
$$L = \text{null} \uplus (\text{int} \times L)$$

- Equations récursives, en notant \emptyset la liste vide, et en notant (x, L) une liste non-vide.

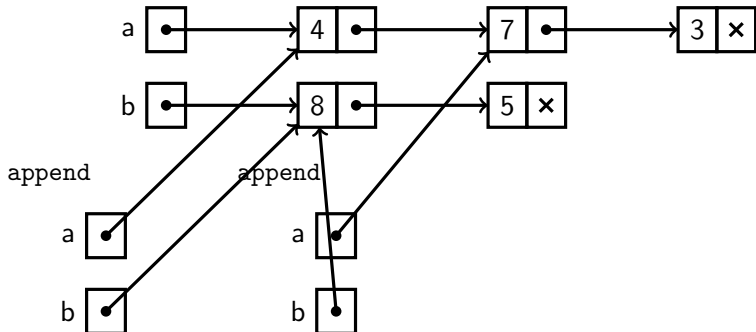
$$\begin{aligned} \text{append}(\emptyset, b) &= b \\ \text{append}((x, L), b) &= (x, \text{append}(L, b)) \end{aligned}$$

```
static Liste append(Liste a, Liste b) {  
    if (a== null)  
        return b;  
    else  
        return new Liste(a.contenu, append(a.suivant,b));  
}
```

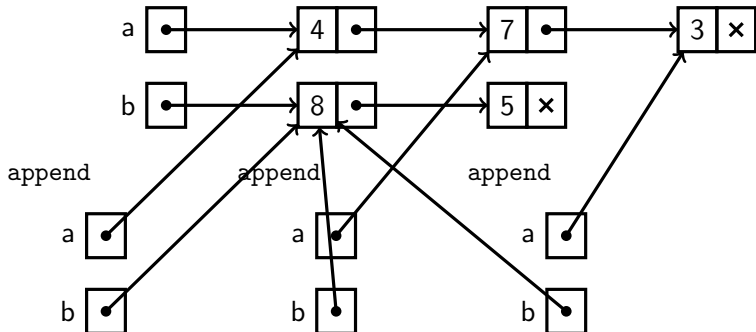
Concaténer deux listes (version récursive, persistante)



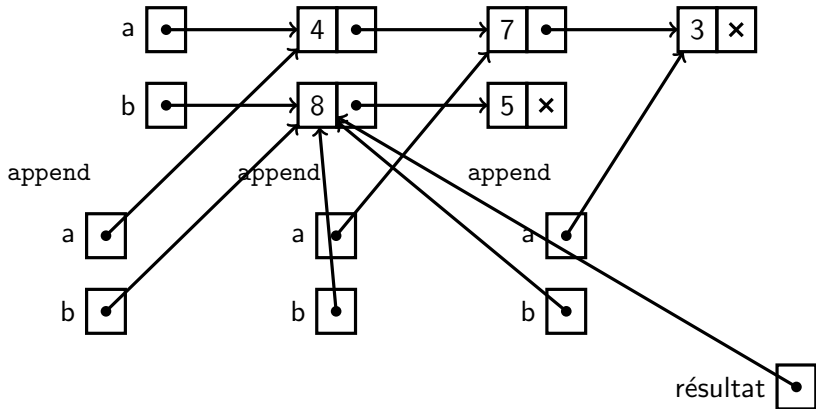
Concaténer deux listes (version récursive, persistante)



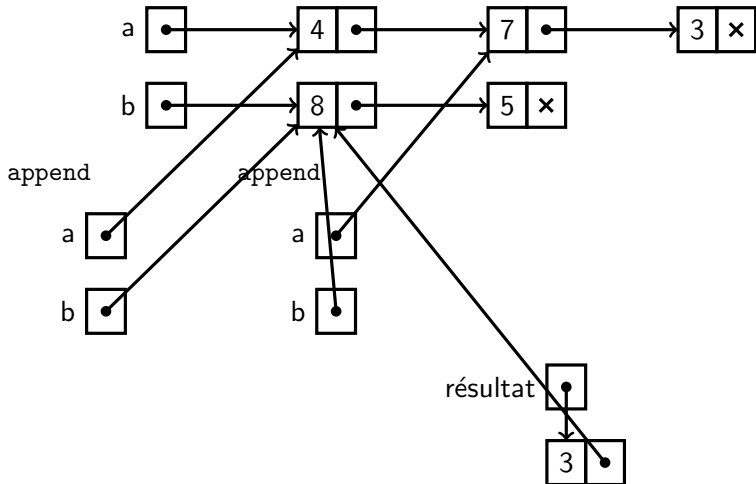
Concaténer deux listes (version récursive, persistante)



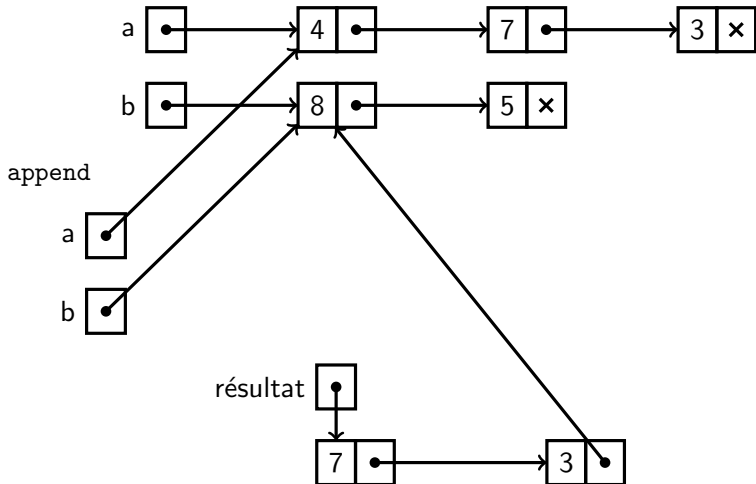
Concaténer deux listes (version récursive, persistante)



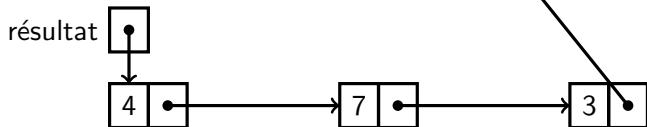
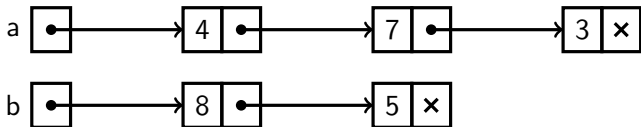
Concaténer deux listes (version récursive, persistante)



Concaténer deux listes (version récursive, persistante)



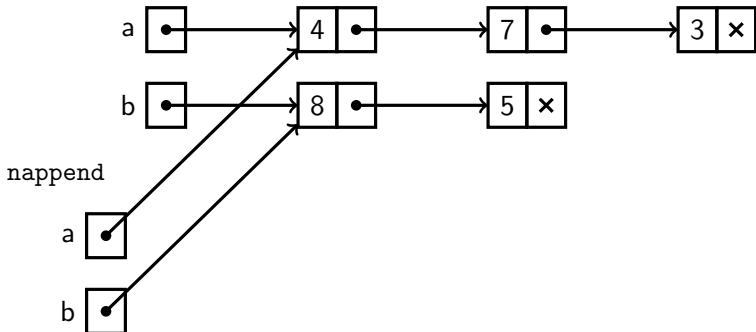
Concaténer deux listes (version récursive, persistante)



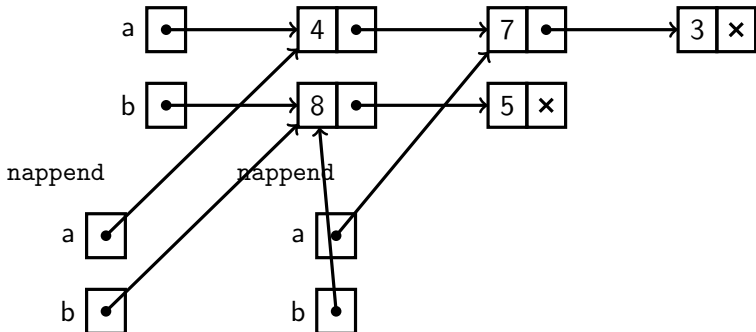
Concaténer deux listes (version récursive, mutable)

```
static Liste nappend(Liste a,Liste b) {  
    if (a==null)  
        return b;  
    else {  
        a.suivant = nappend(a.suivant,b);  
        return a;  
    }  
}}
```

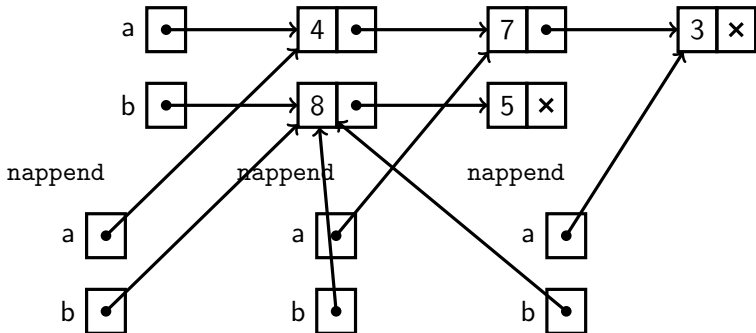
Concaténer deux listes (version récursive, mutable)



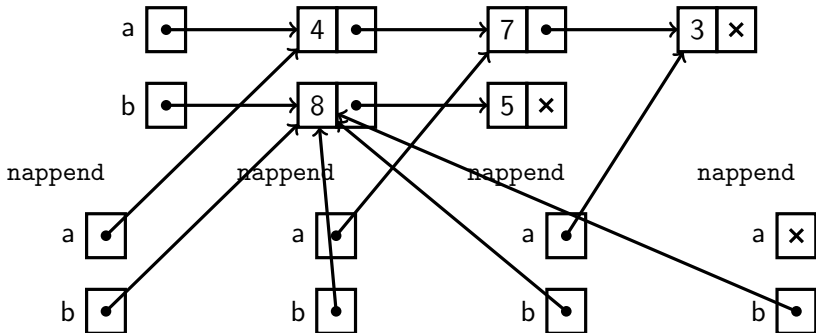
Concaténer deux listes (version récursive, mutable)



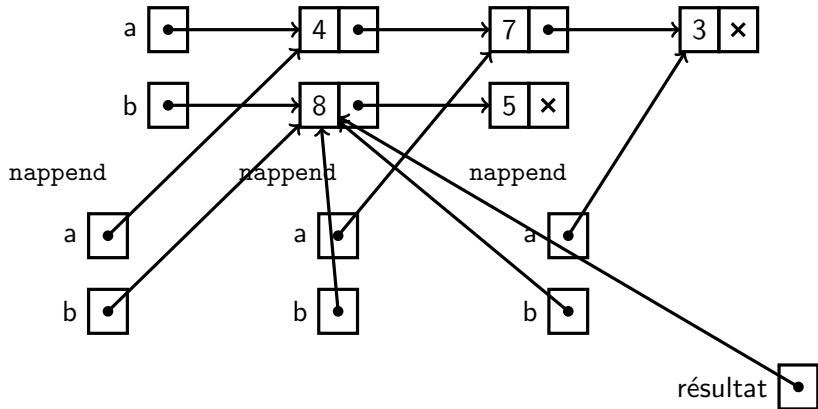
Concaténer deux listes (version récursive, mutable)



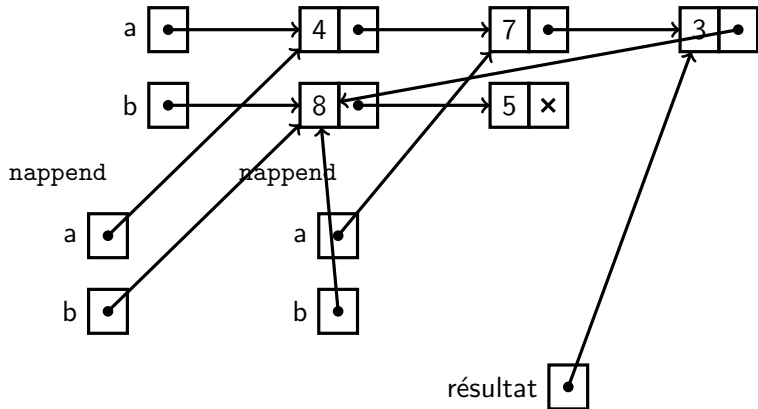
Concaténer deux listes (version récursive, mutable)



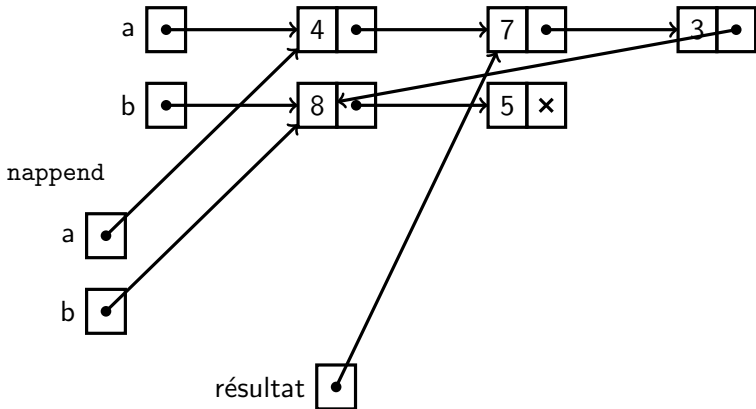
Concaténer deux listes (version récursive, mutable)



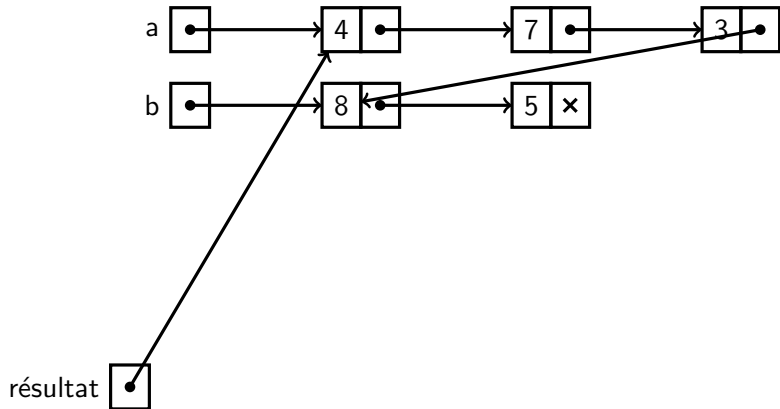
Concaténer deux listes (version récursive, mutable)



Concaténer deux listes (version récursive, mutable)



Concaténer deux listes (version récursive, mutable)



Cours 2: Programmer avec des listes. Complexité. Tris.

Olivier Bournez

bournez@lix.polytechnique.fr
LIX, Ecole Polytechnique

Aujourd'hui

Programmer avec des listes

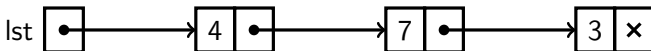
Complexité

Trier

Rappel: Liste

```
class Liste {  
    int contenu;  
    Liste suivant;  
    Liste (int x, Liste a) {  
        contenu = x;  
        suivant = a;  
    }  
}
```

```
Liste lst = new Liste(4, new Liste (7, new Liste (3, null)));
```



Calculer le maximum

- Le problème *MAX*.
 - ▶ On se donne une liste d'entiers naturels e_1, e_2, \dots, e_n .
 - ▶ On veut calculer $\max\{e_1, e_2, \dots, e_n\}$.

Calculer le maximum

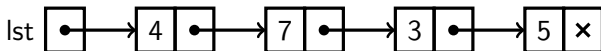
- Le problème *MAX*.

- ▶ On se donne une liste d'entiers naturels e_1, e_2, \dots, e_n .
- ▶ On veut calculer $\max\{e_1, e_2, \dots, e_n\}$.

-1 si la liste est vide.

Calculer le maximum (itératif)

```
static int max(Liste a) {  
    if (a == null) return -1;  
    int max = a.contenu;  
    for (a = a.suivant; a != null; a = a.suivant) {  
        if (a.contenu > max)  
            max = a.contenu;}  
    return max;}  
}
```



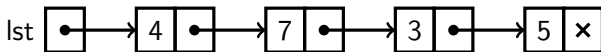
Nombre de comparaisons $C(n)$:

Nombre d'affectations $B(n)$:

n = longueur de la liste.

Calculer le maximum (itératif)

```
static int max(Liste a) {  
    if (a== null) return -1;  
    int max = a.contenu;  
    for (a = a.suivant; a ≠ null; a = a.suivant) {  
        if (a.contenu > max)  
            max = a.contenu;}  
    return max;}  
}
```



Nombre de comparaisons $C(n)$:

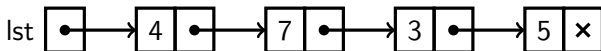
Nombre d'affectations $B(n)$:

n = longueur de la liste.

comparaison = comparaison entre entiers.

Calculer le maximum (itératif)

```
static int max(Liste a) {  
    if (a == null) return -1;  
    int max = a.contenu;  
    for (a = a.suivant; a != null; a = a.suivant) {  
        if (a.contenu > max)  
            max = a.contenu;}  
    return max;}  
}
```



Nombre de comparaisons $C(n)$: $C(n) = n - 1$.

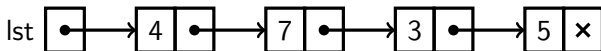
Nombre d'affectations $B(n)$:

n = longueur de la liste.

comparaison = comparaison entre entiers.

Calculer le maximum (itératif)

```
static int max(Liste a) {  
    if (a == null) return -1;  
    int max = a.contenu;  
    for (a = a.suivant; a != null; a = a.suivant) {  
        if (a.contenu > max)  
            max = a.contenu;}  
    return max;}  
}
```



Nombre de comparaisons $C(n)$: $C(n) = n - 1$.

Nombre d'affectations $B(n)$: $n + 1 \leq B(n) \leq 2n$.

n = longueur de la liste.

comparaison = comparaison entre entiers.

Calculer le maximum (récursif)

- Une liste d'entiers est solution de l'équation

$$L = \text{null} \uplus (\text{int} \times L)$$

- Equations récursives, en notant \emptyset la liste vide, et en notant (x, L) une liste non-vide.

$$\begin{aligned} \text{max}_{\text{liste}}(\emptyset) &= -1 \\ \text{max}_{\text{liste}}((x, L)) &= \max(x, \text{max}_{\text{liste}}(L)) \end{aligned}$$

Calculer le max (récursif)

```
static int max(Liste a) {  
    if (a == null) return -1;  
    return max(a.contenu, max(a.suivant));  
}  
  
static int max(int a, int b) {  
    if (a>b) return a;  
    return b;  
}
```

Remarque de JAVA: on utilise une surcharge de méthode.

Nombre de comparaisons $C(n)$:

$$\begin{aligned} & \text{Solution de } C(0) = 0, \\ C(n) &= 1 + C(n-1), \text{ pour } n > 0. \end{aligned}$$

Calculer le max (récursif)

```
static int max(Liste a) {  
    if (a == null) return -1;  
    return max(a.contenu, max(a.suivant));  
}  
  
static int max(int a, int b) {  
    if (a>b) return a;  
    return b;  
}
```

Remarque de JAVA: on utilise une surcharge de méthode.

Nombre de comparaisons $C(n)$: $C(n) = n$.

Solution de $C(0) = 0$,
 $C(n) = 1 + C(n - 1)$, pour $n > 0$.

Calculer le max (récurusif inefficace)

Equations récursives:

$$\begin{aligned} \text{max}_{\text{liste}}(\emptyset) &= -1 \\ \text{max}_{\text{liste}}((x, L)) &= \begin{cases} x & \text{si } x > \text{max}_{\text{liste}}(L) \\ \text{max}_{\text{liste}}(L) & \text{sinon.} \end{cases} \end{aligned}$$

```
static int max(Liste a) {  
    if (a == null) return -1;  
    if (a.contenu > max(a.suivant))  
        return a.contenu;  
    else  
        return max(a.suivant);  
}
```

Nombre de comparaisons: $n \leq C(n) \leq 2^n - 1$.

Meilleur cas $C(n) = 1 + C(n - 1)$, pour $n > 0$, $C(0) = 0$

Pire cas $C(n) = 1 + 2 * C(n - 1)$, pour $n > 0$, $C(0) = 0$.

Quel est le meilleur programme?

- Styles

- ▶ Récursif / Itératif

- ▶ Persistant / Non persistant

- Structures de données:

- ▶ Tableaux / Listes / Piles, Files, Arbres, Graphes, ...

Quel est le meilleur programme?

■ Styles

- ▶ Récursif / Itératif

Souvent:

- Itératif plus efficace (en Java).
 - Récursif plus simple et élégant.
- ▶ Persistant / Non persistant

■ Structures de données:

- ▶ Tableaux / Listes / Piles, Files, Arbres, Graphes, ...

Quel est le meilleur programme?

■ Styles

- ▶ Récursif / Itératif

Souvent:

- Itératif plus efficace (en Java).
 - **Récursif plus simple et élégant.**
- ▶ Persistant / Non persistant

■ Structures de données:

- ▶ Tableaux / Listes / Piles, Files, Arbres, Graphes, ...

Quel est le meilleur programme?

■ Styles

- ▶ Récursif / Itératif

Souvent:

- Itératif plus efficace (en Java).
- **Récursif plus simple et élégant.**

- ▶ Persistant / Non persistant

Souvent:

- Persistant moins problématique.

■ Structures de données:

- ▶ Tableaux / Listes / Piles, Files, Arbres, Graphes, ...

Quel est le meilleur programme?

■ Styles

- ▶ Récursif / Itératif

Souvent:

- Itératif plus efficace (en Java).
- **Récursif plus simple et élégant.**

- ▶ Persistant / Non persistant

Souvent:

- Persistant moins problématique.

■ Structures de données:

- ▶ Tableaux / Listes / Piles, Files, Arbres, Graphes, ...

Souvent:

- Fonction du contexte

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Complexité d'un problème, d'un algorithme

- On fixe une mesure élémentaire:

- ▶ nombre de comparaisons,
- ▶ nombre d'affectations,
- ▶ mémoire utilisée,
- ▶ temps utilisé,
- ▶ ...

- Cette mesure associe à un algorithme \mathcal{A} et une entrée d une valeur

$$\text{Complexite}(\mathcal{A}, d).$$

- On cherche souvent à évaluer cette complexité en fonction d'un paramètre naturel $n = \text{taille}(d)$, représentatif des entrées.

- Exemple:

- ▶ \mathcal{A} : l'algorithme itératif pour *MAX* précédent.
- ▶ d : une liste donnée en entrée.
- ▶ $n = \text{taille}(d)$, le nombre d'éléments dans la liste.
- ▶ $\text{Complexite}(\mathcal{A}, d)$, le nombre de comparaisons de \mathcal{A} sur d .

- On peut alors parler de la complexité (au pire cas)

- ▶ d'un algorithme (on fait varier seulement les entrées)

$$Complexite_{\mathcal{A}}(n) = \max_{d/taille(d)=n} Complexite(\mathcal{A}, d).$$

- ▶ d'un problème (on fait varier l'algorithme, et les entrées)

$$Complexite(n) = \inf_{\mathcal{A} \text{ correct}} \max_{d/taille(d)=n} Complexite(\mathcal{A}, d)$$

- On arrive parfois à parler de la complexité exacte.
- On doit souvent se limiter à une étude asymptotique, ou à des bornes asymptotiques.

Exemple d'étude exacte

- Etude du problème *MAX* en nombre de comparaisons:
 - ▶ Le problème *MAX* admet une solution avec $n - 1$ comparaisons.

```
static int max(Liste a) {  
    if (a== null) return -1;  
    int max = a.contenu;  
    for (a = a.suivant; a ≠ null; a = a.suivant) {  
        if (a.contenu > max)  
            max = a.contenu;}  
    return max;}  
}
```

Exemple d'étude exacte

- Etude du problème *MAX* en nombre de comparaisons:
 - ▶ Le problème *MAX* admet une solution avec $n - 1$ comparaisons.

```
static int max(Liste a) {  
    if (a == null) return -1;  
    int max = a.contenu;  
    for (a = a.suivant; a != null; a = a.suivant) {  
        if (a.contenu > max)  
            max = a.contenu;}  
    return max;}  
}
```

- ▶ Cet algorithme est optimal en nombre de comparaisons.

Preuve

- Proposition: dans tout algorithme, tout élément autre que le maximum doit être comparé au moins une fois avec un élément qui lui est plus grand.
- Preuve:
 - ▶ soit i_0 le rang du maximum M retourné par l'algorithme sur une liste $L = e_1.e_2.\dots.e_n$.
 - ▶ Par l'absurde: soit $j_0 \neq i_0$ tel que e_{j_0} n'est pas comparé avec un élément plus grand que lui.
 - ▶ e_{j_0} n'a donc pas été comparé avec e_{i_0} le maximum
 - ▶ Considérer la liste $L' = e_1.e_2.\dots.e_{j_0-1}.M + 1.e_{j_0+1}.\dots.e_n$ obtenue à partir de L en remplaçant l'élément d'indice j_0 par $M+1$.
 - ▶ L'algorithme effectue exactement les mêmes comparaisons sur L et L' , sans comparer $L'[j_0]$ avec $L'[i_0]$ et donc retournera $L'[i_0]$, ce qui est incorrect.
- Conséquence: il faut au moins $n - 1$ comparaisons pour résoudre *MAX*.

Asymptotiques

- Pour un processeur capable d'effectuer un million d'instructions élémentaires par seconde.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 m	10^{25} a
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 m	36 a	∞
$n = 10^2$	< 1 s	< 1 s	< 1 s	1s	12.9 a	10^{17} a	∞
$n = 10^3$	< 1 s	< 1 s	1s	18 m	∞	∞	∞
$n = 10^4$	< 1 s	< 1 s	2 m	12 h	∞	∞	∞
$n = 10^5$	< 1 s	2 s	3 h	32 a	∞	∞	∞
$n = 10^6$	1s	20s	12 j	31710 a	∞	∞	∞

- Notations: ∞ = le temps dépasse 10^{25} années, s= seconde, m= minute, h = heure, a = an.

Notation de Landau (Pire Cas)

■ Notation de Landau:

- ▶ $f(n) = O(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que

$$\forall n \geq n_0, f(n) \leq Bg(n)$$

Ce qui signifie que f ne croît pas plus vite que g .

■ Exemple:

- ▶ Le problème *MAX* admet une solution itérative en $O(n)$ affectations.
- ▶ L'algorithme itératif précédent fonctionne en $O(n)$ affectations.

■ Un algorithme

- ▶ en temps $O(1)$ effectue un nombre constant d'opérations.
- ▶ en temps $O(n)$ s'appelle un algorithme linéaire.
- ▶ en temps $O(n^k)$ s'appelle un algorithme polynomial.

Notation de Landau

■ Notation de Landau (suite)

- ▶ $f(n) = \Omega(g(n))$ si et seulement si il existe deux constantes positives n_0 et B telles que

$$\forall n \geq n_0, f(n) \geq Bg(n)$$

- ▶ $f(n) = \Theta(g(n))$ si et seulement si il existe trois constantes positives n_0 , B et C telles que

$$\forall n \geq n_0, Bg(n) \leq f(n) \leq Cg(n)$$

■ Exemple:

- ▶ Le problème *MAX* nécessite $\Theta(n)$ comparaisons.
- ▶ (à venir) Trier nécessite $\Omega(n \log n)$ comparaisons.

Complexité d'un algorithme en moyenne

- Si on fixe une distribution π sur les entrées d , il est parfois possible d'évaluer la complexité en moyenne d'un algorithme:

$$\text{Complexité-Moyenne}_{\mathcal{A}}(n) = \sum_{d/\text{taille}(d)=n} \pi(d) \text{Complexité}(\mathcal{A}, d)$$

- Exemple: pour l'algorithme \mathcal{A} suivant.

```
static boolean Dans(int x, Liste a) {  
    for (; a != null; a=a.suivant) {  
        if (a.contenu == x)  
            return true;  
    }  
    return false;  
}
```

Nombre de comparaisons (entre entiers): k , où k est le rang de l'élément lorsqu'il est dans la liste, n sinon.

- Si on suppose que les entrées sont les listes permutations de $\{1, 2, \dots, n\}$, et qu'elles sont équiprobables,

$$\begin{aligned} \text{Complexite} - \text{Moyenne}_{\mathcal{A}}(n) &= \text{Esperance}_{d/\text{taille}(d)=n}[k] \\ &= \sum_{i=1}^n i \times P(k = i) \end{aligned}$$

Puisque les permutations sont équiprobables,
 $\text{Proba}(k = i) = 1/n$.

$$\begin{aligned} \text{Complexite} - \text{Moyenne}_{\mathcal{A}}(n) &= \sum_{i=1}^n i \frac{1}{n} \\ &= \frac{n(n+1)}{2n} \\ &= \frac{n+1}{2} \end{aligned}$$

- (poly) D'autres exemples plus compliqués.
- (joli exercice) Pour \mathcal{A} algorithme itératif précédent pour MAX , $\text{Complexite} - \text{Moyenne}_{\mathcal{A}}(n)$ en affectations entre variables entières est en $\Theta(\log n)$ (de l'ordre de $H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}$).

Aujourd'hui

Programmer avec des listes

Complexité

Trier

Pourquoi trier?

- Trier est une opération naturelle.
- Travailler sur des données triées est parfois plus efficace.
 - ▶ Exemple:
 - Rechercher un élément dans un tableau à n éléments: $O(n)$ (utiliser le principe de la fonction *Dans* précédente sur les listes).
 - Rechercher un élément dans un tableau trié à n éléments: $O(\log n)$ (par dichotomie).

Pourquoi trier?

- Trier est une opération naturelle.
- Travailler sur des données triées est parfois plus efficace.
 - ▶ Exemple:
 - Rechercher un élément dans un tableau à n éléments: $O(n)$ (utiliser le principe de la fonction *Dans* précédente sur les listes).
 - Rechercher un élément dans un tableau trié à n éléments: $O(\log n)$ (par dichotomie).
 - Trier devient intéressant dès que le nombre de recherches est en $\Omega(\text{Complexite}(\text{tri})/n)$.

Recherche par dichotomie

```
static boolean trouve(int[] T, int v, int min, int max){
    if(min ≥ max) // vide
        return false;
    int mid = (min + max) / 2;
    if (T[mid] == v) return true;
    else if (T[mid] > v) return trouve(T, v, min, mid);
    else return trouve(T, v, mid + 1, max);
}
```

Nombre de comparaisons

- $C(1) = 2$,
- $C(n) \leq 2 + C(n/2)$, pour n pair,
- $C(n) \leq 2 \log n + 2$, pour n puissance de 2.

$$C(n) = O(\log n),$$

dans le cas général.

Trier par insertion

- Méthode souvent utilisée pour trier un jeu de cartes.
- Etape préliminaire: savoir insérer un élément x dans une liste triée $e_1.e_2.\dots.e_n$, avec $e_i \leq e_{i+1}$.
 - ▶ Equations récurrentes:

$$\begin{aligned} \text{Insere}(x, \emptyset) &= (x, \emptyset) \\ \text{Insere}(x, (a, L)) &= (x, (a, L)) && \text{si } x \leq a \\ \text{Insere}(x, (a, L)) &= (a, \text{Insere}(x, L)) && \text{sinon} \end{aligned}$$

- ▶ Correction: par induction, si on suppose (a, L) triée, le résultat est bien trié.

```
static Liste Insere(int x, Liste a) {
    if (a==null) return new Liste(x,null);
    if (x <= a.contenu) return new Liste(x,a);
    return new Liste(a.contenu,Insere(x,a.suivant));
}
```

Tri par insertion

```
static Liste Trie(Liste p) {  
    Liste r = null;  
    for (; p != null; p = p.suivant)  
        r = Insere(p.contenu,r);  
    return r;}  
}
```

- Complexité $I(n)$ de Insere en comparaisons : $I(n) = O(n)$.
- Complexité $C(n)$ de Trie en comparaisons:
 $C(n) \leq nI(n) = O(n^2)$.
- Le pire cas est atteint lorsqu'on trie une liste déjà triée, et mène à $\Omega(n^2)$ comparaisons. La complexité du tri par insertion est donc bien en $\Theta(n^2)$.

Tri par insertion en moyenne

- Le meilleur cas est atteint lorsqu'on trie une liste dans l'ordre décroissant, et mène à $O(n)$ comparaisons.
- $O(n) \leq C(n) \leq O(n^2)$.
- En moyenne? Si on se restreint au tri des listes d'entiers distincts deux à deux, et en supposant les permutations équiprobables,

$$\begin{aligned} C(n) &= \frac{1}{4}n^2 + \frac{3}{4}n - \ln n + O(1). \\ &= \Theta(n^2) \end{aligned}$$

(Supplément) Preuve

Principe:

- Coût moyen de Insere avec déjà $k - 1$ éléments:

$$I(k) = \frac{1}{k}(1 + 2 + \dots + k - 1 + k - 1) = \frac{1}{k}\left(\frac{k(k+1)}{2} - 1\right)$$

(position de l'élément inséré équiprobable)

- Coût moyen de Trie:

$$C(n) = 0 + \frac{1}{2}\left(\frac{2(2+1)}{2} - 1\right) + \dots + \frac{1}{n}\left(\frac{n(n+1)}{2} - 1\right)$$

(k premiers éléments répartis uniformément)

Tri par fusion

- Fusion:
 - ▶ Construire une liste triée qui contienne l'union des éléments de deux listes triées.
- Exemple: $Fusion(1.4.5.7, 2.4.9) = 1.2.4.4.5.7.9$
- Equations récursives de $Fusion(X, Y)$:

$$Fusion(X, \emptyset) = X$$

$$Fusion(\emptyset, Y) = Y$$

$$Fusion((a, L), (b, M)) = (a, Fusion(L, (b, M))) \quad \text{si } a \leq b$$

$$Fusion((a, L), (b, M)) = (b, Fusion((a, L), M)) \quad \text{sinon}$$

- Correction: par induction, si on suppose (a, L) et (b, M) triées, le résultat est correct.

Fusion

```
static Liste Fusion(Liste a, Liste b) {  
    if (a == null)  
        return b;  
    if (b == null)  
        return a;  
    if (a.contenu < b.contenu)  
        return new Liste(a.contenu, Fusion(a.suivant, b));  
    else  
        return new Liste(b.contenu, Fusion(a, b.suivant));  
}
```

- Complexité en nombre de comparaisons:
 $O(\text{longueur}(a) + \text{longueur}(b))$.

Tri fusion

- Une liste de 0 ou 1 élément est triée.
- Toute autre liste L
 - ▶ peut se découper en deux sous-listes L_1 et L_2 de même taille (à 1 près).
 - ▶ Les sous-listes L_1 et L_2 sont triées récursivement,
 - ▶ puis fusionnées.

$$TriFusion(L) = Fusion(TriFusion(L_1), TriFusion(L_2)).$$

Paradigme “Diviser pour régner”

- Complexité $C(n)$ en nombre de comparaisons en $O(\text{Fusion}) + 2C(n/2)$, soit

$$C(n) = O(n + 2C(n/2)),$$

ce qui mène à

$$C(n) = O(n \log n).$$

Paradigme “Diviser pour régner”

- Complexité $C(n)$ en nombre de comparaisons en $O(\text{Fusion}) + 2C(n/2)$, soit

$$C(n) = O(n + 2C(n/2)),$$

ce qui mène à

$$C(n) = O(n \log n).$$

- Note:

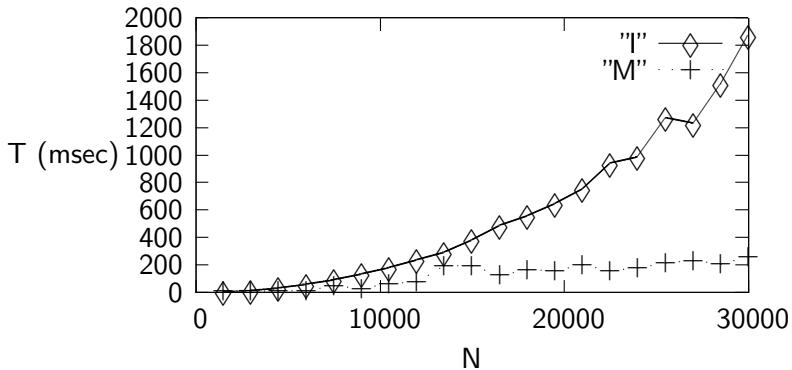
- ▶ comment résoudre une telle récurrence: poser $n = 2^p$ et faire le changement de variable $C'(p) = C(2^p)$.
- ▶ $C'(p) = O(2^p + 2C'(p-1))$,
- ▶ donc $C'(p) = O(2^p p)$.

Tri fusion

```
static Liste MergeSort(Liste l) {  
    Liste l1 = null, l2 = null;  
    boolean even = true ;  
    for ( ; l  $\neq$  null ; l = l.suivant ) {  
        if (even) {  
            l1 = new Liste (l.contenu, l1) ;  
        } else {  
            l2 = new Liste (l.contenu, l2) ;  
        }  
        even = !even ;  
    }  
    if (l2==null) return l1;  
    return Fusion(MergeSort(l1),MergeSort(l2));  
}
```

- Note: L1 et L2 sont en fait ici dans l'ordre inverse des éléments pairs et impairs, par commodité.
- Note: le tri fusion fait *toujours* de l'ordre de $n \log n$ comparaisons, et pas seulement au pire cas.

Insertion vs. fusion



I est le tri par insertion.

M est le tri par fusion.

Complexité du problème du tri

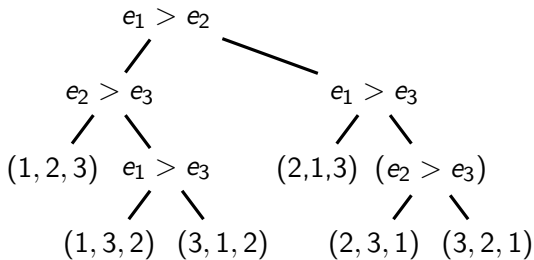
- Tout algorithme de tri
 - ▶ qui fonctionne par comparaisons,
 - ▶ qui n'a pas d'autres informations sur les données,

effectue $\Omega(n \log n)$ comparaisons dans le pire des cas.

- Autrement dit, la complexité du problème du tri (avec ces hypothèses) est en $\Theta(n \log n)$.
- Le tri par fusion est donc parmi les tris optimaux en $O(n \log n)$.

Preuve de la borne inférieure $\Omega(n \log n)$: 1/2

- A un algorithme on peut associer un arbre de décision:



- ▶ En chaque noeud qui n'est pas une feuille: une comparaison effectuée par l'algorithme.
 - La racine est la première comparaison.
 - Fils gauche: récursivement ce qui se passe alors pour un résultat positif.
 - Fils droit: récursivement ce qui se passe alors pour un résultat négatif.
- ▶ En chaque feuille, le résultat produit.

Preuve de la borne inférieure $\Omega(n \log n)$: 2/2

- Un arbre binaire de hauteur h a au plus 2^h feuilles.
 - ▶ par récurrence.
- Les feuilles doivent contenir au moins les $n!$ permutations, et donc $h \geq \log(n!) = \Omega(n \log n)$ par la formule de Stirling.