Diviser pour régner

Crédits

1. « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.

Crédits

1. « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
2. Cours « diviser pour régner », Becirspahic

Crédits

1. « Option informatique MPSI, MP/MP* », Roger Mansuy, paru chez Vuibert.
2. Cours « diviser pour régner », Becirspahic
3. Wikipédia : Analyse de la compléxité des algorithmes.

Diviser pour régner

En informatique, ≪ diviser pour régner ≫ (en anglais divide and conquer) est un schéma de conception d'algorithmes qui suit trois étapes :

1. Diviser (Divide) : découper le problème initial en plusieurs sous-problèmes de même nature, mais de taille plus faible.

Diviser pour régner

En informatique, ≪ diviser pour régner ≫ (en anglais divide and conquer) est un schéma de conception d'algorithmes qui suit trois étapes :

1. Diviser (Divide) : découper le problème initial en plusieurs sous-problèmes de même nature, mais de taille plus faible.
2. Régner (Conquer) : résoudre chaque sous-problème, généralement de façon récursive.

Diviser pour régner

En informatique, ≪ diviser pour régner ≫ (en anglais divide and conquer) est un schéma de conception d'algorithmes qui suit trois étapes :

1. Diviser (Divide) : découper le problème initial en plusieurs sous-problèmes de même nature, mais de taille plus faible.
2. Régner (Conquer) : résoudre chaque sous-problème, généralement de façon récursive.
3. Combiner (Combine) : agréger les solutions partielles pour obtenir la solution du problème initial.

Exemples

1. Tri fusion (Merge Sort)

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.
   - Régner : trier récursivement chaque moitié.

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.
   - Régner : trier récursivement chaque moitié.
   - Combiner : fusionner les deux tableaux triés en un seul tableau trié.

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.
   - Régner : trier récursivement chaque moitié.
   - Combiner : fusionner les deux tableaux triés en un seul tableau trié.
2. Recherche binaire

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.
   - Régner : trier récursivement chaque moitié.
   - Combiner : fusionner les deux tableaux triés en un seul tableau trié.
2. Recherche binaire
   - Diviser : comparer la clé recherchée à l'élément central du tableau.

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.
   - Régner : trier récursivement chaque moitié.
   - Combiner : fusionner les deux tableaux triés en un seul tableau trié.
2. Recherche binaire
   - Diviser : comparer la clé recherchée à l'élément central du tableau.
   - Régner : éliminer la moitié où la clé ne peut se trouver et répéter sur l'autre moitié.

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.
   - Régner : trier récursivement chaque moitié.
   - Combiner : fusionner les deux tableaux triés en un seul tableau trié.
2. Recherche binaire
   - Diviser : comparer la clé recherchée à l'élément central du tableau.
   - Régner : éliminer la moitié où la clé ne peut se trouver et répéter sur l'autre moitié.
   - (Pas de combinaison, car la recherche aboutit directement.)

Exemples

1. Tri fusion (Merge Sort)
   - Diviser : couper le tableau en deux moitiés.
   - Régner : trier récursivement chaque moitié.
   - Combiner : fusionner les deux tableaux triés en un seul tableau trié.
2. Recherche binaire
   - Diviser : comparer la clé recherchée à l'élément central du tableau.
   - Régner : éliminer la moitié où la clé ne peut se trouver et répéter sur l'autre moitié.
   - (Pas de combinaison, car la recherche aboutit directement.)
3. Algorithme de Strassen pour la multiplication de matrices, Fast Fourier Transform (FFT), etc.

Avertissement

Dans tout ce qui suit :

- Par *liste trié*, on sous-entend implicitement « liste triée par ordre croissant ».

Avertissement

Dans tout ce qui suit :

- Par *liste trié*, on sous-entend implicitement ≪ liste triée par ordre croissant ≫.
- Par *permutation d'une liste* on sous-entend une nouvelle liste ayant la même taille que la première, et contenant tous les éléments de la première avec le même nombre d'occurences.

Avertissement

Dans tout ce qui suit :

- Par *liste trié*, on sous-entend implicitement ≪ liste triée par ordre croissant ≫.
- Par *permutation d'une liste* on sous-entend une nouvelle liste ayant la même taille que la première, et contenant tous les éléments de la première avec le même nombre d'occurences.
- Par *version triée d'une liste*, on sous entend une permutation de la liste mais triée par ordre croissant.

Historique

- Écrit vers 1945.

Historique

- Écrit vers 1945.
- Attribué au Hongro-Américain John Von Neumann.

Historique

- Écrit vers 1945.
- Attribué au Hongro-Américain John Von Neumann.
- Un des premiers algorithmes de tris proposé pour les ordinateurs.

Historique

- Écrit vers 1945.
- Attribué au Hongro-Américain John Von Neumann.
- Un des premiers algorithmes de tris proposé pour les ordinateurs.
- Utilise le principe *diviser pour régner* qui consiste à décomposer récursivement un gros problème en plus petits sous-problèmes.

Principe

On veut une version triée d'une liste `l`.

- Découper la liste en deux parties de tailles proches.

Principe

On veut une version triée d'une liste `l`.

- Découper la liste en deux parties de tailles proches.
- Trier ces deux parties.

Principe

On veut une version triée d'une liste `l`.

- Découper la liste en deux parties de tailles proches.
- Trier ces deux parties.
- Fusionner les deux listes triées obtenues.

Fusion (rappel)

```
let rec fusion l1 l2 = match l1, l2 with
  | [], l -> l
  | l, [] -> l
  | x1::t1, x2::t2 ->  if x1<=x2
    then x1 :: fusion t1 l2
    else x2 :: fusion l1 t2 ;;
```

- Terminaison : le variant de récursion est $|\ell_1| + |\ell_2|$: entier, strictement décroissant et positif.

Fusion (rappel)

```
let rec fusion l1 l2 = match l1, l2 with
  | [], l -> l
  | l, [] -> l
  | x1::t1, x2::t2 ->  if x1<=x2
    then x1 :: fusion t1 l2
    else x2 :: fusion l1 t2 ;;
```

- Terminaison : le variant de récursion est $|\ell_1| + |\ell_2|$: entier, strictement décroissant et positif.
- Correction : on veut montrer que pour deux listes triées par ordre croissant `l1,l2`, la liste obtenue par l'appel `fusion l1 l2` est une version triée de `l1@l2`.

Correction de la fusion

Préconditions : `l1` et `l2` sont des listes triées d'entiers.

- On note $n_1 = |\ell_1|$ et $n_2 = |\ell_2|$ les longueurs des listes opérandes.

Correction de la fusion

Préconditions : `l1` et `l2` sont des listes triées d'entiers.

- On note $n_1 = |\ell_1|$ et $n_2 = |\ell_2|$ les longueurs des listes opérandes.
- On montre par récurrence sur $n_1 + n_2 = n$ la propriété $P(n)$ « la liste résultat de `fusion l1 l2` est une version triée de `l1@l2` ».
  Remarque : sa taille est donc $n_1 + n_2$

Correction de la fusion

Préconditions : `l1` et `l2` sont des listes triées d'entiers.

- On note $n_1 = |\ell_1|$ et $n_2 = |\ell_2|$ les longueurs des listes opérandes.
- On montre par récurrence sur $n_1 + n_2 = n$ la propriété $P(n)$ « la liste résultat de `fusion l1 l2` est une version triée de `l1@l2` ».
  Remarque : sa taille est donc $n_1 + n_2$
- Cas de base. Lorsque $n_1 + n_2 = 0$, alors les deux listes sont vides. On retourne la liste vide qui est triée et contient les éléments de `l1,l2`.

Correction de la fusion

Hérédité

On suppose $P(n)$ vérifiée. On réalise l'appel `fusion l1 l2` avec $n_1 + n_2 = n + 1$.

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.

1. In Ze Pocket

Correction de la fusion

Hérédité

On suppose $P(n)$ vérifiée. On réalise l'appel `fusion l1 l2` avec $n_1 + n_2 = n + 1$.

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément x_1 de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).

1. In Ze Pocket

Correction de la fusion

Hérédité

On suppose $P(n)$ vérifiée. On réalise l'appel `fusion l1 l2` avec $n_1 + n_2 = n + 1$.

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément x_1 de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).
  - L'hypothèse de récurrence s'applique puisque $|t_1| + |l_2| = n_1 - 1 + n_2 = n$ et `t1,l2` sont triés

1. In Ze Pocket

Correction de la fusion

Hérédité

On suppose $P(n)$ vérifiée. On réalise l'appel `fusion l1 l2` avec $n_1 + n_2 = n + 1$.

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément x_1 de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).
  - L'hypothèse de récurrence s'applique puisque $|t_1| + |l_2| = n_1 - 1 + n_2 = n$ et `t1,l2` sont triés
  - Donc `fusion t1 l2` est une version triée de `t1@l2`.

1. In Ze Pocket

Correction de la fusion

Hérédité

On suppose $P(n)$ vérifiée. On réalise l'appel `fusion l1 l2` avec $n_1 + n_2 = n + 1$.

- Si l'une des deux listes est vide, on retourne celle qui ne l'est pas. L'invariant est vérifié trivialement.
- Si les deux listes sont non vides, supposons que l'appel interne soit `fusion t1 l2` (se produit si le premier élément x_1 de `l1` est inférieur au premier de `l2` ; l'autre cas est laissé au lecteur).
  - L'hypothèse de récurrence s'applique puisque $|t_1| + |l_2| = n_1 - 1 + n_2 = n$ et `t1,l2` sont triés
  - Donc `fusion t1 l2` est une version triée de `t1@l2`.
  - Comme `t1,l2` ne contiennent que des éléments plus grands que `x1`, `x1::fusion t1 l2` est triée et contient les bons éléments. IZP [1] ! !

1. In Ze Pocket

Fonction de séparation

La fonction suivante coupe en deux parties de longueurs presque égales une liste :

```
let rec separer l=
  match  l with
  | [] | [_] -> l, []
  | x::y::q -> let l1,l2 = separer q in x::l1,y::l2;;
```

Cette fonction termine car 1) la longueur de l'argument de `separer` diminue strictement à chaque appel récursif et 2) les cas de bases terminent.

Exercice

Produire une version en récurrence terminale.

Fonction de séparation

On montre que `separer l` renvoie 2 listes de tailles $\lceil \frac{\ell}{2} \rceil$ et $\lfloor \frac{\ell}{2} \rfloor$ (dans cet ordre) et que la concaténation des deux listes renvoyées est une permutation de la liste de départ :

- Cas de bases : C'est vrai pour les cas d'arrêts (vérification immédiate).

Fonction de séparation

On montre que `separer l` renvoie 2 listes de tailles $\lceil \frac{\ell}{2} \rceil$ et $\lfloor \frac{\ell}{2} \rfloor$ (dans cet ordre) et que la concaténation des deux listes renvoyées est une permutation de la liste de départ :

- Cas de bases : C'est vrai pour les cas d'arrêts (vérification immédiate).
- Hérédité : Si la propriété est vraie pour une liste ℓ de taille égale ou inférieure à $n > 0$ (récurrence forte), considérons une liste de taille $n+1$ et effectuons `separer l`.
  Par HR, l'appel interne (fait avec des listes de tailles $n-1$) renvoie deux listes de tailles $\lceil \frac{n-1}{2} \rceil$ et $\lfloor \frac{n-1}{2} \rfloor$ dont la concaténation est une permutation de `q`.
  La preuve en deux points (contenu et taille) est détaillée dans le transparent suivant.

Fonction de séparation

Hérédité

On admet HR pour $n > 0$. On effectue `separer l` pour $\ell = n + 1$.

- Contenu : En ajoutant les deux éléments x, y à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.

Fonction de séparation

Hérédité

On admet HR pour $n > 0$. On effectue `separer l` pour $\ell = n + 1$.

- Contenu : En ajoutant les deux éléments x, y à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.
- Taille :

Fonction de séparation

Hérédité

On admet HR pour $n > 0$. On effectue `separer l` pour $\ell = n + 1$.

- Contenu : En ajoutant les deux éléments x, y à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.
- Taille :
  - Si n est de la forme $2k + 1$, alors $\lceil \frac{n-1}{2} \rceil = \lceil k \rceil = k$ et $\lfloor \frac{n-1}{2} \rfloor = \lfloor k \rfloor = k$. Du fait de la concaténation, les listes retournées sont de tailles $k + 1 = \lceil \frac{n+1}{2} \rceil$ et $k + 1 = \lfloor \frac{n+1}{2} \rfloor$.

Fonction de séparation

Hérédité

On admet HR pour $n > 0$. On effectue `separer l` pour $\ell = n + 1$.

- Contenu : En ajoutant les deux éléments x, y à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.
- Taille :
  - Si n est de la forme $2k + 1$, alors $\lceil \frac{n-1}{2} \rceil = \lceil k \rceil = k$ et $\lfloor \frac{n-1}{2} \rfloor = \lfloor k \rfloor = k$. Du fait de la concaténation, les listes retournées sont de tailles $k + 1 = \lceil \frac{n+1}{2} \rceil$ et $k + 1 = \lfloor \frac{n+1}{2} \rfloor$.
  - Si n est de la forme $2k$, alors $\lceil \frac{n-1}{2} \rceil = \lceil k - \frac{1}{2} \rceil = k$ et $\lfloor \frac{n-1}{2} \rfloor = \lfloor k - \frac{1}{2} \rfloor = k - 1$. Du fait de la concaténation, les listes retournées sont de tailles $k + 1 = \lceil \frac{n+1}{2} \rceil$ et $k = \lfloor \frac{n+1}{2} \rfloor$.

Fonction de séparation

Hérédité

On admet HR pour $n > 0$. On effectue `separer l` pour $\ell = n + 1$.

- Contenu : En ajoutant les deux éléments x, y à chacune des deux listes résultats (dont la concaténation est une permutation de `q`), et en les concaténant, on obtient bien une permutation de `l`.
- Taille :
  - Si n est de la forme $2k + 1$, alors $\lceil \frac{n-1}{2} \rceil = \lceil k \rceil = k$ et $\lfloor \frac{n-1}{2} \rfloor = \lfloor k \rfloor = k$. Du fait de la concaténation, les listes retournées sont de tailles $k + 1 = \lceil \frac{n+1}{2} \rceil$ et $k + 1 = \lfloor \frac{n+1}{2} \rfloor$.
  - Si n est de la forme $2k$, alors $\lceil \frac{n-1}{2} \rceil = \lceil k - \frac{1}{2} \rceil = k$ et $\lfloor \frac{n-1}{2} \rfloor = \lfloor k - \frac{1}{2} \rfloor = k - 1$. Du fait de la concaténation, les listes retournées sont de tailles $k + 1 = \lceil \frac{n+1}{2} \rceil$ et $k = \lfloor \frac{n+1}{2} \rfloor$.
- Hérédité OK.

Le tri fusion

Exercice

Écrire la fonction `tri_fusion : 'a list -> 'a list` qui réalise le tri fusion. Etablir terminaison et correction.

Le tri fusion

Démonstration.

```
let rec tri_fusion l =  match l with
  | [] -> l
  | [_] -> l
  | _ -> let l1,l2 = separer l in
    let l1'= tri_fusion l1 and l2' = tri_fusion l2 in
    fusion l1' l2';;
```

Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.

Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille $\leq n$, considérons l'appel `tri_fusion l` pour `l` de taille $n+1$ avec $n > 0$.

Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille $\leq n$, considérons l'appel `tri_fusion l` pour `l` de taille $n+1$ avec $n > 0$.
- L'appel à `separer` termine et retourne deux listes de tailles $\lfloor \frac{n+1}{2} \rfloor$ et $\lceil \frac{n+1}{2} \rceil$. Ces deux listes sont passées chacune en argument de `tri_fusion`.

Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille $\leq n$, considérons l'appel `tri_fusion l` pour `l` de taille $n+1$ avec $n > 0$.
- L'appel à `separer` termine et retourne deux listes de tailles $\lfloor \frac{n+1}{2} \rfloor$ et $\lceil \frac{n+1}{2} \rceil$. Ces deux listes sont passées chacune en argument de `tri_fusion`.
- Les deux appels ci-dessus terminent par hypothèse de récurrence puisque, $n+1$ étant plus grand que 2, les deux parties entières de $\frac{n+1}{2}$ sont strictement plus petites que $n+1$.

Terminaison

- Dans les cas de bases, (liste vide ou singleton), la fonction termine de façon évidente.
- Si `tri_fusion l` termine pour tout `l` de taille $\leq n$, considérons l'appel `tri_fusion l` pour `l` de taille $n+1$ avec $n > 0$.
- L'appel à `separer` termine et retourne deux listes de tailles $\lfloor \frac{n+1}{2} \rfloor$ et $\lceil \frac{n+1}{2} \rceil$. Ces deux listes sont passées chacune en argument de `tri_fusion`.
- Les deux appels ci-dessus terminent par hypothèse de récurrence puisque, $n+1$ étant plus grand que 2, les deux parties entières de $\frac{n+1}{2}$ sont strictement plus petites que $n+1$.
- Enfin, la fusion de deux listes termine toujours donc `tri_fusion l` termine. Hérédité OK.

Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.

Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille $\leq n$, considérons l'appel `tri l` pour `l` de taille $n+1$ avec $n > 0$.

Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille $\leq n$, considérons l'appel `tri l` pour `l` de taille $n+1$ avec $n > 0$.
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque $n+1 > 1$.

Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille $\leq n$, considérons l'appel `tri l` pour `l` de taille $n+1$ avec $n > 0$.
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque $n+1 > 1$.
- Ces deux listes sont passées chacune en argument de `tri_fusion` et, par HR, on obtient une version triée de chacune. On les note `l1',l2'`.

Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille $\leq n$, considérons l'appel `tri l` pour `l` de taille $n+1$ avec $n > 0$.
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque $n+1 > 1$.
- Ces deux listes sont passées chacune en argument de `tri_fusion` et, par HR, on obtient une version triée de chacune. On les note `l1',l2'`.
- Enfin, `fusion` appliquée à ces 2 listes retourne une version triée de `l1'@l2'`.

Correction

- Dans les cas de bases, (liste vide ou singleton), la fonction retourne une version triée de `l` de façon évidente.
- Si `tri_fusion l` renvoie une version triée de `l` pour tout `l` de taille $\leq n$, considérons l'appel `tri l` pour `l` de taille $n+1$ avec $n > 0$.
- L'appel à `separer` termine et retourne 2 listes `l1,l2` qui, à elle deux, contiennent exactement les éléments de `l`. Elles sont chacune strictement plus courtes que `l` puisque $n+1 > 1$.
- Ces deux listes sont passées chacune en argument de `tri_fusion` et, par HR, on obtient une version triée de chacune. On les note `l1',l2'`.
- Enfin, `fusion` appliquée à ces 2 listes retourne une version triée de `l1'@l2'`.
- Cette liste est une version triée de `l1@l2` donc de `l`. IZP

Fusion (rappel)

```
let rec fusion l1 l2 = match l1, l2 with
  | [], l -> l
  | l, [] -> l
  | x1::t1, x2::t2 ->  if x1<=x2
    then x1 :: fusion t1 l2
    else x2 :: fusion l1 t2 ;;
```

- La complexité dépend de la façon dont sont réparties les données dans les listes. Meilleur cas en $O(\min(|\ell_1|, |\ell_2|))$: lorsque tous les éléments de la liste la plus courte sont plus petits que ceux de la plus longue.

Fusion (rappel)

```
let rec fusion l1 l2 = match l1, l2 with
  | [], l -> l
  | l, [] -> l
  | x1::t1, x2::t2 ->  if x1<=x2
    then x1 :: fusion t1 l2
    else x2 :: fusion l1 t2 ;;
```

- La complexité dépend de la façon dont sont réparties les données dans les listes. Meilleur cas en $O(\min(|\ell_1|, |\ell_2|))$: lorsque tous les éléments de la liste la plus courte sont plus petits que ceux de la plus longue.
- Pour éviter un raisonnement qui tiendrait compte de la répartition des données, on peut étudier la complexité d'une fonction manifestement plus coûteuse : `fusion_couteuse` du transparent suivant.

Tri fusion

Fusion coûteuse

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

Tri fusion

Fusion coûteuse

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

- La complexité dépend seulement de la somme des tailles des listes. En notant n, m les tailles de `l1,l2` on obtient la relation suivante $C(n+m) = C(n+m-1) + O(1)$ et $C(0) = 1$.

Tri fusion

Fusion coûteuse

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

- La complexité dépend seulement de la somme des tailles des listes. En notant n, m les tailles de `l1,l2` on obtient la relation suivante $C(n+m) = C(n+m-1) + O(1)$ et $C(0) = 1$.
- On étudie donc $C(n+m) = C(n+m-1) + 1$. Il s'agit d'une suite arithmétique. On a donc $C(n+m) = \Theta(n+m)$ (ordre de grandeur).

Tri fusion

Fusion coûteuse

- La fonction suivante réalise aussi la fusion. Elle est plus coûteuse que la précédente mais plus facile à étudier :

```
let rec fusion_couteuse l1 l2 = match (l1,l2) with
| ([],[]) -> []
| (a::r,[]) | ([],a::r) -> a::(fusion_couteuse [] r)
| (a::r,b::s) -> if a<=b
  then a::(fusion_couteuse r l2)
  else b::(fusion_couteuse l1 s);;
```

- La complexité dépend seulement de la somme des tailles des listes. En notant n, m les tailles de `l1,l2` on obtient la relation suivante $C(n+m) = C(n+m-1) + O(1)$ et $C(0) = 1$.
- On étudie donc $C(n+m) = C(n+m-1) + 1$. Il s'agit d'une suite arithmétique. On a donc $C(n+m) = \Theta(n+m)$ (ordre de grandeur).
- Donc la complexité de `fusion`, toujours meilleure, est aussi en $O(n+m)$ (majoration, ça nous suffit pour la suite).

Fonction de séparation

```
let rec separer l=
  match  l with
  | [] | [_] -> l, []
  | x::y::q -> let l1,l2 = separer q in x::l1,y::l2;;
```

Pour une liste de taille n.

- Récurrence de complexité : $C(n) = C(n-2) + O(1)$ simplifié en $C(n) = C(n-2) + 1$ avec $C(0) = C(1) = 1$.

Fonction de séparation

```
let rec separer l=
  match  l with
  | [] | [_] -> l, []
  | x::y::q -> let l1,l2 = separer q in x::l1,y::l2;;
```

Pour une liste de taille n.

- Récurrence de complexité : $C(n) = C(n-2) + O(1)$ simplifié en $C(n) = C(n-2) + 1$ avec $C(0) = C(1) = 1$.
- $C(n) \leq n + 1$ par récurrence immédiate. Complexité en $O(n)$.

Le tri fusion

Rappel

```
let rec tri_fusion l =  match l with
  | [] | [_] -> l
  | _ -> let l1,l2 = separer l in
    let l1'= tri_fusion l1 and l2' = tri_fusion l2 in
    fusion l1' l2';;
```

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.
- La complexité obéit à une relation diviser-pour-régner de la forme

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n) \text{ ; simplifié en } C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n$$

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.
- La complexité obéit à une relation diviser-pour-régner de la forme

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n) \text{ ; simplifié en } C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n$$

- Si $n = 2^k$ alors $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = 2^{k-1}$ (et $k = \log_2 n$) :

$$C_{2^k} = 2^k + 2C_{2^{k-1}} = 2 \cdot 2^k + 2^2 C_{2^{k-2}} = \cdots = k2^k + 2^k \cdot C_{2^{k-k}} = \Theta(n \log_2 n)$$

Tri fusion

Complexité du tri

- On note n la longueur de `l`. La division est en $\Theta(n)$. La fusion s'applique aux deux moitiés triées ; elle a ici un coût en $O(n)$ (et même en $\Theta(n)$ -en exo-). Bref : séparation + fusion en $\Theta(n)$.
- La complexité obéit à une relation diviser-pour-régner de la forme

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n) \text{ ; simplifié en } C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n$$

- Si $n = 2^k$ alors $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = 2^{k-1}$ (et $k = \log_2 n$) :

$$C_{2^k} = 2^k + 2C_{2^{k-1}} = 2 \cdot 2^k + 2^2 C_{2^{k-2}} = \cdots = k2^k + 2^k \cdot C_{2^{k-k}} = \Theta(n \log_2 n)$$

- En général, $2^{\lfloor \log_2 n \rfloor} \leq n < 2^{\lfloor \log_2 n \rfloor + 1}$. Comme la complexité est $\uparrow$:

$$C_n \leq C_{2^{\lfloor \log_2 n \rfloor + 1}} = O\left((\overbrace{\lfloor \log_2 n \rfloor + 1}^{O(\log_2 n)}) \underbrace{2^{\lfloor \log_2 n \rfloor + 1}}_{O(n)} \right) = O(n \log n)$$

Idem pour une minoration. Conclusion : $C_n = \Theta(n \log_2 n)$