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.

Présentation

- *tri fusion* (*merge sorte*) : algorithme de tri par comparaison stable (qui conserve l'ordre des éléments identiques).

Présentation

- *tri fusion* (*merge sorte*) : algorithme de tri par comparaison stable (qui conserve l'ordre des éléments identiques).
- Complexité temporelle pour une entrée de taille n, de l'ordre de $O(n \log_2 n)$. Asymptotiquement optimal.

Présentation

- *tri fusion* (*merge sorte*) : algorithme de tri par comparaison stable (qui conserve l'ordre des éléments identiques).
- Complexité temporelle pour une entrée de taille n, de l'ordre de $O(n \log_2 n)$. Asymptotiquement optimal.
- Principe : *diviser pour régner* (comme le quicksort).

Présentation

- *tri fusion* (*merge sorte*) : algorithme de tri par comparaison stable (qui conserve l'ordre des éléments identiques).
- Complexité temporelle pour une entrée de taille n, de l'ordre de $O(n \log_2 n)$. Asymptotiquement optimal.
- Principe : *diviser pour régner* (comme le quicksort).
- Opération principale : *la fusion*, *i.e.* réunir deux listes triées en une seule.

Présentation

- *tri fusion* (*merge sorte*) : algorithme de tri par comparaison stable (qui conserve l'ordre des éléments identiques).
- Complexité temporelle pour une entrée de taille n, de l'ordre de $O(n \log_2 n)$. Asymptotiquement optimal.
- Principe : *diviser pour régner* (comme le quicksort).
- Opération principale : *la fusion*, *i.e.* réunir deux listes triées en une seule.
- L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.

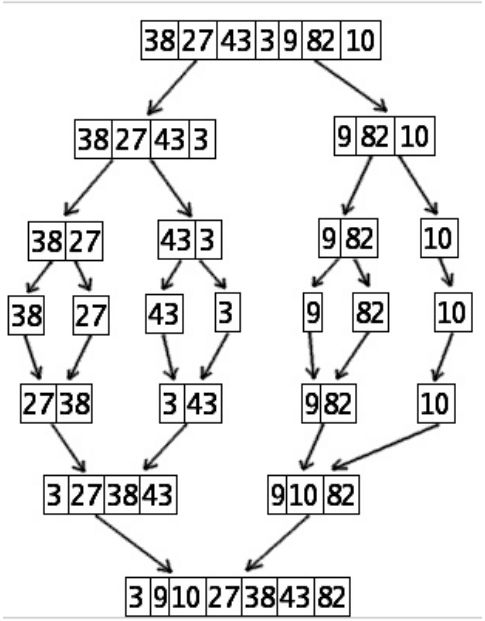
Présentation

- *tri fusion* (*merge sorte*) : algorithme de tri par comparaison stable (qui conserve l'ordre des éléments identiques).
- Complexité temporelle pour une entrée de taille n, de l'ordre de $O(n \log_2 n)$. Asymptotiquement optimal.
- Principe : *diviser pour régner* (comme le quicksort).
- Opération principale : *la fusion*, *i.e.* réunir deux listes triées en une seule.
- L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.
- Difficile à réaliser en place : on travaille avec des tableaux *auxiliaires*.

Principe

- On coupe en deux parties à peu près égales les données à trier ;

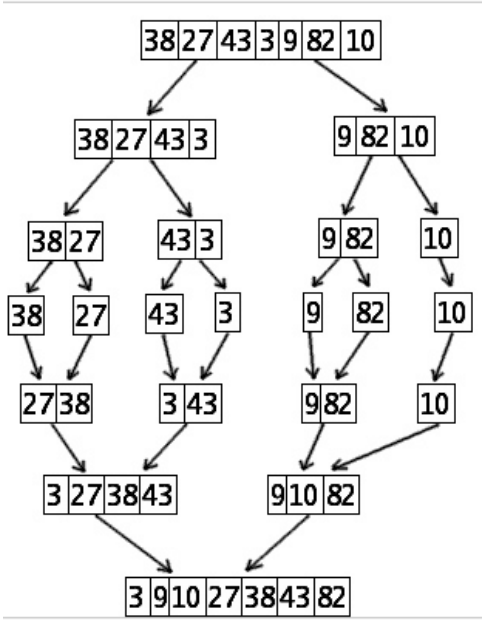
Figure – Diviser pour régner



Principe

- On coupe en deux parties à peu près égales les données à trier ;
- On trie les données de chaque partie (pour cela, on coupe chaque partie en deux et on trie chacune) ;

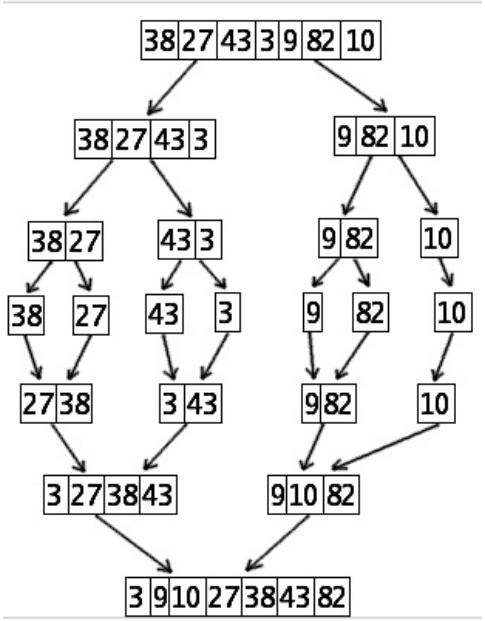
Figure – Diviser pour régner



Principe

- On coupe en deux parties à peu près égales les données à trier ;
- On trie les données de chaque partie (pour cela, on coupe chaque partie en deux et on trie chacune) ;
- On fusionne les deux parties.

Figure – Diviser pour régner



Fusion

- On veut fusionner | 1 | 2 | 5 | et | 3 | 4 |
  On sait que le premier élément de la liste fusionnée sera le premier élément d'une des deux listes d'entrée (soit 1, soit 3) car ce sont des listes triées.
  On compare donc 1 et 3 $\rightarrow$ 1 est plus petit.

Fusion

- On veut fusionner | 1 | 2 | 5 | et | 3 | 4 |
  On sait que le premier élément de la liste fusionnée sera le premier élément d'une des deux listes d'entrée (soit 1, soit 3) car ce sont des listes triées.
  On compare donc 1 et 3 $\rightarrow$ 1 est plus petit.
- On gère : | 2 | 5 | ; | 3 | 4 | et la liste résultat | 1 |
  On compare 2 et 3 $\rightarrow$ 2 est plus petit.

Fusion

- On veut fusionner | 1 | 2 | 5 | et | 3 | 4 |
  On sait que le premier élément de la liste fusionnée sera le premier élément d'une des deux listes d'entrée (soit 1, soit 3) car ce sont des listes triées.
  On compare donc 1 et 3 $\rightarrow$ 1 est plus petit.
- On gère : | 2 | 5 | ; | 3 | 4 | et la liste résultat | 1 |
  On compare 2 et 3 $\rightarrow$ 2 est plus petit.
- On gère | 5 | ; | 3 | 4 | et | 1 | 2 |
  On compare 5 et 3 $\rightarrow$ 3 est plus petit.

Fusion

- On veut fusionner | 1 | 2 | 5 | et | 3 | 4 |
  On sait que le premier élément de la liste fusionnée sera le premier élément d'une des deux listes d'entrée (soit 1, soit 3) car ce sont des listes triées.
  On compare donc 1 et 3 $\rightarrow$ 1 est plus petit.
- On gère : | 2 | 5 | ; | 3 | 4 | et la liste résultat | 1 |
  On compare 2 et 3 $\rightarrow$ 2 est plus petit.
- On gère | 5 | ; | 3 | 4 | et | 1 | 2 |
  On compare 5 et 3 $\rightarrow$ 3 est plus petit.
- On gère | 5 | ; | 4 | et | 1 | 2 | 3 |
  On compare 5 et 4 $\rightarrow$ 4 est plus petit.

Fusion

- On veut fusionner | 1 | 2 | 5 | et | 3 | 4 |
  On sait que le premier élément de la liste fusionnée sera le premier élément d'une des deux listes d'entrée (soit 1, soit 3) car ce sont des listes triées.
  On compare donc 1 et 3 $\rightarrow$ 1 est plus petit.
- On gère : | 2 | 5 | ; | 3 | 4 | et la liste résultat | 1 |
  On compare 2 et 3 $\rightarrow$ 2 est plus petit.
- On gère | 5 | ; | 3 | 4 | et | 1 | 2 |
  On compare 5 et 3 $\rightarrow$ 3 est plus petit.
- On gère | 5 | ; | 4 | et | 1 | 2 | 3 |
  On compare 5 et 4 $\rightarrow$ 4 est plus petit.
- On gère
  | 5 | ; | | et | 1 | 2 | 3 | 4 |
  Pas de comparaison. On ajoute le tableau non vide à la fin des résultats : | 1 | 2 | 3 | 4 | 5 |

Fusion

Code

D'abord la fonction qui fusionne deux listes A, B **triées par ordre croissant** :

```
1  def merge(A,B):
2      a,b,r=0,0,[]
3      while a < len(A) and b <len(B):
4          if  A[a]<=B[b]:
5              r.append(A[a])
6              a+=1
7          else:
8              r.append(B[b])
9              b+=1
10     return r + A[a:] + B[b:]
```

Convention

Comme il est d'usage dans les preuves de terminaison/correction pour les fonctions impératives, on indice les variables par le numéro de tour de boucle.

- La numérotation des tours de boucle commence à 1. Le tour de boucle 0 désigne ce qu'il se passe AVANT la boucle.

Convention

Comme il est d'usage dans les preuves de terminaison/correction pour les fonctions impératives, on indice les variables par le numéro de tour de boucle.

- La numérotation des tours de boucle commence à 1. Le tour de boucle 0 désigne ce qu'il se passe AVANT la boucle.
- La valeur a_k (resp b_k, r_k) désigne le contenu de `a` (resp. `r`, `b`) au tour k. Ainsi $a_0 = 0$, $b_0 = 0$ et $r_0 = \emptyset$. Après un tour, on a $r_1 = [\min(A, B)]$ (élément minimum de A et B).

Convention

Comme il est d'usage dans les preuves de terminaison/correction pour les fonctions impératives, on indice les variables par le numéro de tour de boucle.

- La numérotation des tours de boucle commence à 1. Le tour de boucle 0 désigne ce qu'il se passe AVANT la boucle.
- La valeur a_k (resp b_k, r_k) désigne le contenu de `a` (resp. `r`, `b`) au tour k. Ainsi $a_0 = 0$, $b_0 = 0$ et $r_0 = \emptyset$. Après un tour, on a $r_1 = [\min(A, B)]$ (élément minimum de A et B).
- Enfin, si x est du même type que les éléments des listes, on note abusivement $x > r_k$ pour désigner $x > \max_{i=0,\ldots,|r_k|-1}(r_k[i])$.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.
  - Elle est positive tant qu'il y a des entrées dans la boucle.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.
  - Elle est positive tant qu'il y a des entrées dans la boucle.
  - Considérons un passage k. S'il y a un passage $k + 1$ dans la boucle, $(a_k < a_{k+1}$ et $b_k = b_{k+1})$ ou $(a_k = a_{k+1}$ et $b_k < b_{k+1})$.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.
  - Elle est positive tant qu'il y a des entrées dans la boucle.
  - Considérons un passage k. S'il y a un passage $k + 1$ dans la boucle, $(a_k < a_{k+1}$ et $b_k = b_{k+1})$ ou $(a_k = a_{k+1}$ et $b_k < b_{k+1})$.
    - On a de toute façon $a_k + b_k < a_{k+1} + b_{k+1}$, donc

      $|A|+|B|-a_{k+1}-b_{k+1} < |A|+|B|-a_k-b_k$: décroissance stricte.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.
  - Elle est positive tant qu'il y a des entrées dans la boucle.
  - Considérons un passage k. S'il y a un passage $k + 1$ dans la boucle, ($a_k < a_{k+1}$ et $b_k = b_{k+1}$) ou ($a_k = a_{k+1}$ et $b_k < b_{k+1}$).
    - On a de toute façon $a_k + b_k < a_{k+1} + b_{k+1}$, donc

      $|A|+|B|-a_{k+1}-b_{k+1} < |A|+|B|-a_k-b_k$: décroissance stricte.

    - Il est clair que si $|A| + |B| - (a_k + b_k)$ est entier, alors $|A| + |B| - a_{k+1} - b_{k+1}$ aussi.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.
  - Elle est positive tant qu'il y a des entrées dans la boucle.
  - Considérons un passage k. S'il y a un passage $k + 1$ dans la boucle, ($a_k < a_{k+1}$ et $b_k = b_{k+1}$) ou ($a_k = a_{k+1}$ et $b_k < b_{k+1}$).
    - On a de toute façon $a_k + b_k < a_{k+1} + b_{k+1}$, donc $|A|+|B|-a_{k+1}-b_{k+1} < |A|+|B|-a_k-b_k$: décroissance stricte.
    - Il est clair que si $|A| + |B| - (a_k + b_k)$ est entier, alors $|A| + |B| - a_{k+1} - b_{k+1}$ aussi.
- Si $|A| + |B| - (a_k + b_k) < 0$ alors $a_k > |A|$ ou $b_k > |B|$: donc il y a sortie de boucle.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.
  - Elle est positive tant qu'il y a des entrées dans la boucle.
  - Considérons un passage k. S'il y a un passage $k + 1$ dans la boucle, ($a_k < a_{k+1}$ et $b_k = b_{k+1}$) ou ($a_k = a_{k+1}$ et $b_k < b_{k+1}$).
    - On a de toute façon $a_k + b_k < a_{k+1} + b_{k+1}$, donc $|A|+|B|-a_{k+1}-b_{k+1} < |A|+|B|-a_k-b_k$: décroissance stricte.
    - Il est clair que si $|A| + |B| - (a_k + b_k)$ est entier, alors $|A| + |B| - a_{k+1} - b_{k+1}$ aussi.
- Si $|A| + |B| - (a_k + b_k) < 0$ alors $a_k > |A|$ ou $b_k > |B|$: donc il y a sortie de boucle.
- On a donc montré que $|A| + |B| - (a_k + b_k)$ est une quantité positive décroissante et entière, d'où la terminaison.

Terminaison

- On constate que s'il y a un passage dans la boucle $|A| + |B| - (a + b) \geq 0$ car $|A| > a$ et $|B| > b$.
- Cette quantité $|A| + |B| - (a + b)$ est un variant de boucle.
  - Elle est positive tant qu'il y a des entrées dans la boucle.
  - Considérons un passage k. S'il y a un passage $k + 1$ dans la boucle, ($a_k < a_{k+1}$ et $b_k = b_{k+1}$) ou ($a_k = a_{k+1}$ et $b_k < b_{k+1}$).
    - On a de toute façon $a_k + b_k < a_{k+1} + b_{k+1}$, donc $|A|+|B|-a_{k+1}-b_{k+1} < |A|+|B|-a_k-b_k$: décroissance stricte.
    - Il est clair que si $|A| + |B| - (a_k + b_k)$ est entier, alors $|A| + |B| - a_{k+1} - b_{k+1}$ aussi.
- Si $|A| + |B| - (a_k + b_k) < 0$ alors $a_k > |A|$ ou $b_k > |B|$: donc il y a sortie de boucle.
- On a donc montré que $|A| + |B| - (a_k + b_k)$ est une quantité positive décroissante et entière, d'où la terminaison.
- Ce raisonnement nous donne par ailleurs que $A + B$ est un majorant du nombre de passage dans la boucle.

Correction de la fusion

On montre que :**À la fin de tout passage k dans la boucle, r_k est trié (croissant) et $r_k[-1]$ (s'il existe) est plus petit que tous les éléments de $A[a_k :]$ et $B[b_k :]$** (si ces tableaux sont non vides).

- Au départ, r_0 est vide donc trié.

Correction de la fusion

On montre que :**À la fin de tout passage k dans la boucle, r_k est trié (croissant) et $r_k[-1]$ (s'il existe) est plus petit que tous les éléments de $A[a_k :]$ et $B[b_k :]$** (si ces tableaux sont non vides).

- Au départ, r_0 est vide donc trié.
- Supposons qu'à la fin d'une étape k, il y a une nouvelle étape $k+1$. HR : r_k est triée par ordre croissant (1) et le dernier élément de r_k est plus petit que tous les éléments de $A[a_k :]$ et $B[b_k :]$ (2). Pour simplifier, supposons $A[a_k] \leq B[b_k]$.

Correction de la fusion

On montre que :**À la fin de tout passage k dans la boucle, r_k est trié (croissant) et $r_k[-1]$ (s'il existe) est plus petit que tous les éléments de $A[a_k :]$ et $B[b_k :]$** (si ces tableaux sont non vides).

- Au départ, r_0 est vide donc trié.
- Supposons qu'à la fin d'une étape k, il y a une nouvelle étape $k+1$. HR : r_k est triée par ordre croissant (1) et le dernier élément de r_k est plus petit que tous les éléments de $A[a_k :]$ et $B[b_k :]$ (2). Pour simplifier, supposons $A[a_k] \leq B[b_k]$.
  - On ajoute le premier (et donc plus petit) élément de $A[a_k :]$ à r_k pour former r_{k+1}. Notons x cet élément. Par HR.2 $x \geq r_k$ (comprendre : $x \geq$ à tous les éléments de r_k). Ainsi, r_{k+1} reste trié par ordre croissant : HR.1 OK.

Correction de la fusion

On montre que :**À la fin de tout passage k dans la boucle, r_k est trié (croissant) et $r_k[-1]$ (s'il existe) est plus petit que tous les éléments de $A[a_k :]$ et $B[b_k :]$** (si ces tableaux sont non vides).

- Au départ, r_0 est vide donc trié.
- Supposons qu'à la fin d'une étape k, il y a une nouvelle étape $k+1$. HR : r_k est triée par ordre croissant (1) et le dernier élément de r_k est plus petit que tous les éléments de $A[a_k :]$ et $B[b_k :]$ (2). Pour simplifier, supposons $A[a_k] \leq B[b_k]$.
  - On ajoute le premier (et donc plus petit) élément de $A[a_k :]$ à r_k pour former r_{k+1}. Notons x cet élément. Par HR.2 $x \geq r_k$ (comprendre : $x \geq$ à tous les éléments de r_k). Ainsi, r_{k+1} reste trié par ordre croissant : HR.1 OK.
  - Par hypothèse, A et B sont triés donc aussi $A[a_k :]$ et $B[b_k :]$. On a donc : $x = A[a_k] \leq A[a_k+1 :] = A[a_{k+1} :]$ et $x \leq B[b_k] \leq B[b_k :] = B[b_{k+1}]$.
    Ainsi HR.2 OK

3ème point de l'invariant

- Pour des raisons de place, on a omis une 3ème propriété invariante.

3ème point de l'invariant

- Pour des raisons de place, on a omis une 3ème propriété invariante.
- La liste r_k, à la fin du tour est k, est de taille $a_k + b_k$ et contient k tous les éléments de $A[: a_k]$ et $B[b : b_k]$.

3ème point de l'invariant

- Pour des raisons de place, on a omis une 3ème propriété invariante.
- La liste r_k, à la fin du tour est k, est de taille $a_k + b_k$ et contient k tous les éléments de $A[: a_k]$ et $B[b : b_k]$.
- On laisse au lecteur scrupuleux le soin de vérifier l'hérédité

Correction de la fusion (fin)

- A la fin du dernier passage dans la boucle, on obtient une liste r triée.

Correction de la fusion (fin)

- A la fin du dernier passage dans la boucle, on obtient une liste r triée.
- On y ajoute la fin d'un des deux tableaux A ou B. Cette fin de tableau est triée par ordre croissant car A, B le sont.

Correction de la fusion (fin)

- A la fin du dernier passage dans la boucle, on obtient une liste r triée.
- On y ajoute la fin d'un des deux tableaux A ou B. Cette fin de tableau est triée par ordre croissant car A, B le sont.
- Or, l'invariant précédent établit que ce qui reste de A (resp. de B) est constitué d'éléments plus grands que le dernier élément de r.

Correction de la fusion (fin)

- A la fin du dernier passage dans la boucle, on obtient une liste r triée.
- On y ajoute la fin d'un des deux tableaux A ou B. Cette fin de tableau est triée par ordre croissant car A, B le sont.
- Or, l'invariant précédent établit que ce qui reste de A (resp. de B) est constitué d'éléments plus grands que le dernier élément de r.
- Par conséquent, à la fin de l'algorithme, r est trié.

Fusion

Complexité

- Avant la boucle, on a 3 opérations.

1. En fait, c'est un coût amorti... mais chut !

Fusion

Complexité

- Avant la boucle, on a 3 opérations.
- S'il y a m passages dans la boucle ($m \leq |A| + |B|$), r contient m cases en sortie. La complexité du corps de boucle est en $O(1)$ (pas de boucle interne, pas d'appel récursif et `append` est supposé en $O(1)$ [1]). La complexité cumulée de tous les passages dans la boucle est donc en $O(|A| + |B|)$ (et même en $\Theta(|A| + |B|)$.

1. En fait, c'est un coût amorti... mais chut !

Fusion

Complexité

- Avant la boucle, on a 3 opérations.
- S'il y a m passages dans la boucle ($m \leq |A| + |B|$), r contient m cases en sortie. La complexité du corps de boucle est en $O(1)$ (pas de boucle interne, pas d'appel récursif et `append` est supposé en $O(1)$ [1]). La complexité cumulée de tous les passages dans la boucle est donc en $O(|A| + |B|)$ (et même en $\Theta(|A| + |B|)$.
- On ajoute ensuite à r : $|A| + |B| - m$ cases, pour un coût max en $O(|A| + |B|)$ (en fait la concaténation telle qu'elle est écrite nous coûte $\Theta(|A| + |B|)$).

1. En fait, c'est un coût amorti... mais chut !

Fusion

Complexité

- Avant la boucle, on a 3 opérations.
- S'il y a m passages dans la boucle ($m \leq |A| + |B|$), r contient m cases en sortie. La complexité du corps de boucle est en $O(1)$ (pas de boucle interne, pas d'appel récursif et `append` est supposé en $O(1)$ [1]). La complexité cumulée de tous les passages dans la boucle est donc en $O(|A| + |B|)$ (et même en $\Theta(|A| + |B|)$.
- On ajoute ensuite à r : $|A| + |B| - m$ cases, pour un coût max en $O(|A| + |B|)$ (en fait la concaténation telle qu'elle est écrite nous coûte $\Theta(|A| + |B|)$).
- Au total, la complexité de `merge` est dominée par un $O(|A| + |B|)$.
  En fait elle est en $\Theta(|A| + |B|)$.

1. En fait, c'est un coût amorti... mais chut !

Tri fusion

Code et terminaison

```
1 def mergesort(L):
2     if len(L)<=1:
3         return L
4     n = len(L)
5     return merge(mergesort(L[:n//2]),\
6                  mergesort(L[n//2:]))
```

- Il y a un nombre fini de cas de base (juste 1). Et l'appel à `mergesort` termine dans ce cas (on retourne L).

Tri fusion

Code et terminaison

```
1 def mergesort(L):
2     if len(L)<=1:
3         return L
4     n = len(L)
5     return merge(mergesort(L[:n//2]),\
6                  mergesort(L[n//2:]))
```

- Il y a un nombre fini de cas de base (juste 1). Et l'appel à `mergesort` termine dans ce cas (on retourne L).
- Les appels internes sont en nombre fini (juste 2), et les listes passées en argument de `mergesort` sont strictement plus courte que la liste initiale. De plus l'appel à `merge` termine.

Tri fusion

Code et terminaison

```
1 def mergesort(L):
2     if len(L)<=1:
3         return L
4     n = len(L)
5     return merge(mergesort(L[:n//2]),\
6                  mergesort(L[n//2:]))
```

- Il y a un nombre fini de cas de base (juste 1). Et l'appel à `mergesort` termine dans ce cas (on retourne L).
- Les appels internes sont en nombre fini (juste 2), et les listes passées en argument de `mergesort` sont strictement plus courte que la liste initiale. De plus l'appel à `merge` termine.
- Terminaison prouvée.

Tri fusion

Correction

Récurrence On montre par récurrence sur $n = |L|$ que `mergesort` retourne la liste triée par ordre croissant des éléments de L.

Tri fusion

Correction

Récurrence On montre par récurrence sur $n = |L|$ que `mergesort` retourne la liste triée par ordre croissant des éléments de L.

Cas de base Si $|L| \leq 1$, on retourne L (qui est triée).

Tri fusion

Hérédité

Soit $n \geq 1$. **HR :** `mergesort(T)` **renvoie une copie triée ↑ de la liste** `T` **pour toute liste** `T` **de taille** $k \leq n$.

Soit L de longueur $n+1$. Alors `L[:n//2]` et `L[n//2:]` sont de tailles inférieures à n.

- `L[:n//2]` est de taille $< n$ donc, par HR, `mergesort(L[:n//2])` est la liste triée ↑ des éléments de `L[:n//2]`.

Tri fusion

Hérédité

Soit $n \geq 1$. **HR : `mergesort(T)` renvoie une copie triée $\uparrow$ de la liste `T` pour toute liste `T` de taille $k \leq n$.**

Soit L de longueur $n+1$. Alors `L[:n//2]` et `L[n//2:]` sont de tailles inférieures à n.

- `L[:n//2]` est de taille $< n$ donc, par HR, `mergesort(L[:n//2])` est la liste triée $\uparrow$ des éléments de `L[:n//2]`.
- De même, `mergesort(L[n//2:])` est la liste triée $\uparrow$ contenant les mêmes éléments que `L[n//2:]`.

Tri fusion

Hérédité

Soit $n \geq 1$. **HR : `mergesort(T)` renvoie une copie triée $\uparrow$ de la liste `T` pour toute liste `T` de taille $k \leq n$.**

Soit L de longueur $n+1$. Alors `L[:n//2]` et `L[n//2:]` sont de tailles inférieures à n.

- `L[:n//2]` est de taille $< n$ donc, par HR, `mergesort(L[:n//2])` est la liste triée $\uparrow$ des éléments de `L[:n//2]`.
- De même, `mergesort(L[n//2:])` est la liste triée $\uparrow$ contenant les mêmes éléments que `L[n//2:]`.
- A elles deux, les deux listes triées renvoyées contiennent tous les éléments de L et aucun autre.

Tri fusion

Hérédité

Soit $n \geq 1$. **HR :** `mergesort(T)` **renvoie une copie triée ↑ de la liste** `T` **pour toute liste** `T` **de taille** $k \leq n$.

Soit L de longueur $n+1$. Alors `L[:n//2]` et `L[n//2:]` sont de tailles inférieures à n.

- `L[:n//2]` est de taille $< n$ donc, par HR, `mergesort(L[:n//2])` est la liste triée ↑ des éléments de `L[:n//2]`.
- De même, `mergesort(L[n//2:])` est la liste triée ↑ contenant les mêmes éléments que `L[n//2:]`.
- A elles deux, les deux listes triées renvoyées contiennent tous les éléments de L et aucun autre.
- Appliquons `merge` à ces deux listes. On obtient une liste triée qui contient tous leurs éléments et aucun autre. Alors les éléments de ce tableau sont exactement ceux de L. Hérédité OK.

Tri fusion

Complexité

On pose $n = |L|$.

- Si $n > 1$, il y a la séparation en deux sous-listes qui se fait en $\Theta(n)$. On effectue deux appels internes sur des listes de taille égales à 1 près. Enfin, on réalise une opération de fusion de complexité $\Theta(n)$ (longueur totale des deux sous-tableaux).

On pose $n = |L|$.

- Si $n > 1$, il y a la séparation en deux sous-listes qui se fait en $\Theta(n)$. On effectue deux appels internes sur des listes de taille égales à 1 près. Enfin, on réalise une opération de fusion de complexité $\Theta(n)$ (longueur totale des deux sous-tableaux).
- Si U_n est la complexité de l'appel `mergesort(L)`, on a

$$U_n = U_{\lceil \frac{n}{2} \rceil} + U_{\lfloor \frac{n}{2} \rfloor} + \Theta(n)$$

Ce que l'on simplifie en se donnant a, b tels que

$$U_{\lceil \frac{n}{2} \rceil} + U_{\lfloor \frac{n}{2} \rfloor} + an \leq U_n \leq U_{\lceil \frac{n}{2} \rceil} + U_{\lfloor \frac{n}{2} \rfloor} + bn$$

Tri fusion

Complexité

On pose $n = |L|$.

- On suppose $n = 2^k$. Alors $\frac{n}{2} = \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$.

$$an + 2U_{\frac{n}{2}} \leq U_n \leq bn + 2U_{\frac{n}{2}}$$

Tri fusion

Complexité

On pose $n = |L|$.

- On suppose $n = 2^k$. Alors $\frac{n}{2} = \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$.

$$an + 2U_{\frac{n}{2}} \leq U_n \leq bn + 2U_{\frac{n}{2}}$$

- On a

$$U_{2^k} \leq b2^k + 2U_{2^{k-1}} \leq 2b2^k + 2^2 U_{2^{k-2}} = kb2^k + 2^k U_{2^{k-k}}$$

Tri fusion

Complexité

On pose $n = |L|$.

- On suppose $n = 2^k$. Alors $\frac{n}{2} = \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$.

$$an + 2U_{\frac{n}{2}} \leq U_n \leq bn + 2U_{\frac{n}{2}}$$

- On a

$$U_{2^k} \leq b2^k + 2U_{2^{k-1}} \leq 2b2^k + 2^2 U_{2^{k-2}} = kb2^k + 2^k U_{2^{k-k}}$$

- Alors $U_n = U_{2^k} \leq bn \log_2 n + nU_1$ où U_1 ne dépend pas de n. On obtient donc que $U_n = O(n \log_2 n)$.

Tri fusion

Complexité

On pose $n = |L|$.

- On suppose $n = 2^k$. Alors $\frac{n}{2} = \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$.

$$an + 2U_{\frac{n}{2}} \leq U_n \leq bn + 2U_{\frac{n}{2}}$$

- On a

$$U_{2^k} \leq b2^k + 2U_{2^{k-1}} \leq 2b2^k + 2^2 U_{2^{k-2}} = kb2^k + 2^k U_{2^{k-k}}$$

- Alors $U_n = U_{2^k} \leq bn \log_2 n + nU_1$ où U_1 ne dépend pas de n. On obtient donc que $U_n = O(n \log_2 n)$.
- De même $U_n = \Omega(n \log_2 n)$. Et donc $U_n = \Theta(n \log_2 n)$.

Complexité

Encadrement

On admet que $(U_n)_n$ est croissante.

- Pour $p \in \mathbb{N}$, existent λ, μ tels que $\lambda p 2^p \leq U_{2^p} \leq \mu p 2^p$

On admet que $(U_n)_n$ est croissante.

- Pour $p \in \mathbb{N}$, existent λ, μ tels que $\lambda p 2^p \leq U_{2^p} \leq \mu p 2^p$
- Soit $n \in \mathbb{N}$. On a $n = 2^{\log_2 n}$. Alors $2^{\lfloor \log_2 n \rfloor} \leq n \leq 2^{\lceil \log_2 n \rceil}$.

On admet que $(U_n)_n$ est croissante.

- Pour $p \in \mathbb{N}$, existent λ, μ tels que $\lambda p 2^p \leq U_{2^p} \leq \mu p 2^p$
- Soit $n \in \mathbb{N}$. On a $n = 2^{\log_2 n}$. Alors $2^{\lfloor \log_2 n \rfloor} \leq n \leq 2^{\lceil \log_2 n \rceil}$.
- Par croissance de U_n :

$$\lambda \lfloor \log_2 n \rfloor 2^{\lfloor \log_2 n \rfloor} \leq U_n \leq \mu \lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil}$$

On admet que $(U_n)_n$ est croissante.

- Pour $p \in \mathbb{N}$, existent λ, μ tels que $\lambda p 2^p \leq U_{2^p} \leq \mu p 2^p$
- Soit $n \in \mathbb{N}$. On a $n = 2^{\log_2 n}$. Alors $2^{\lfloor \log_2 n \rfloor} \leq n \leq 2^{\lceil \log_2 n \rceil}$.
- Par croissance de U_n :

$$\lambda \lfloor \log_2 n \rfloor 2^{\lfloor \log_2 n \rfloor} \leq U_n \leq \mu \lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil}$$

- Comme $\log_2 n = \Theta(\lfloor \log_2 n \rfloor)$ et $\lceil \log_2 n \rceil = \Theta(\log_2 n)$, on a l'existence de a', b' tels que

$$a' n \log_2 n \leq U_n \leq b' n \log_2 n$$

Ainsi, $U_n = \Theta(n \log n)$.

Conclusion

- Le calcul précédent est valable quelle que soit la liste (triée ou non au départ).

Conclusion

- Le calcul précédent est valable quelle que soit la liste (triée ou non au départ).
- Donc, pour le tri fusion, les complexités temporelles au pire et au mieux sont en $O(n \log n)$.

Conclusion

- Le calcul précédent est valable quelle que soit la liste (triée ou non au départ).
- Donc, pour le tri fusion, les complexités temporelles au pire et au mieux sont en $O(n \log n)$.
- C'est donc aussi le cas en moyenne.

Conclusion

- Le calcul précédent est valable quelle que soit la liste (triée ou non au départ).
- Donc, pour le tri fusion, les complexités temporelles au pire et au mieux sont en $O(n \log n)$.
- C'est donc aussi le cas en moyenne.
- On peut montrer qu'aucun algorithme ne fait mieux que $O(n \log n)$ au pire des cas. Il vient que le tri fusion est optimal de ce point de vue.