Module EA4 – Éléments d'Algorithmique II

Dominique Poulalhon
dominique.poulalhon@liafa.univ-paris-diderot.fr

Université Paris Diderot
L2 Informatique
Année universitaire 2014-2015

Génération aléatoire de permutations

`RandomPermutation(n)`

construire une des $n!$ permutations de taille n avec proba uniforme

(*i.e.* : si on exécute tous les comportements (aléatoires) possibles, chaque permutation doit être obtenue le même nombre de fois)

Génération aléatoire de permutations

`RandomPermutation(n)`

construire une des $n!$ permutations de taille n avec proba uniforme

Lemme

toute permutation σ possède une unique décomposition en produit de transpositions $(a_1\ b_1)(a_2\ b_2)\ldots(a_\ell\ b_\ell)$ *avec la contrainte :*

$$\forall i \leqslant \ell,\quad a_i < b_i \quad \textit{et} \quad a_1 < a_2 < \cdots < a_\ell$$

Génération aléatoire de permutations

RandomPermutation(n)

construire une des $n!$ permutations de taille n avec proba uniforme

Lemme

toute permutation σ possède une unique décomposition en produit de transpositions $(a_1\ b_1)(a_2\ b_2)\dots(a_\ell\ b_\ell)$ avec la contrainte :

$$\forall i \leqslant \ell,\ \ a_i < b_i \quad \textit{et} \quad a_1 < a_2 < \cdots < a_\ell$$

Remarque : de manière équivalente, $\sigma = \tau_1 \dots \tau_n$ avec pour chaque i :

$$\tau_i = \mathrm{id} \ \text{ ou } \ \tau_i = (i\ j) \text{ avec } j > i$$

$\Longrightarrow$ le nombre de tels produits est donc exactement $n!$

Génération aléatoire de permutations

RandomPermutation(n)

construire une des $n!$ permutations de taille n avec proba uniforme

Lemme

toute permutation σ possède une unique décomposition en produit de transpositions $(a_1\ b_1)(a_2\ b_2)\dots(a_\ell\ b_\ell)$ avec la contrainte :

$$\forall i \leqslant \ell, \quad a_i < b_i \quad \textit{et} \quad a_1 < a_2 < \cdots < a_\ell$$

Démonstration par récurrence sur le cardinal du support de σ :

- si $\mathrm{Supp}(\sigma) = \emptyset$, $\sigma = \mathrm{id}$, $\tau_i = \mathrm{id}$ pour tout i
- sinon, nécessairement, $a_1 = \min \mathrm{Supp}(\sigma)$ et $b_1 = \sigma(a_1)$
  alors $(a_1\ b_1) \circ \sigma$ a au moins un point fixe de plus que σ,
  donc on peut lui appliquer l'hypothèse de récurrence

Génération aléatoire de permutations

RandomPermutation(n)

construire une des $n!$ permutations de taille n selon la loi de probabilité uniforme

```
from random import randint
# générateur uniforme d'entiers dans un intervalle
def randomPerm(n) :
  l = [ i+1 for i in range(n) ]
  for i in range(n-1) :
    r = randint(i, n-1) # id ou transposition (i r) ?
    if i != r : # multiplication à droite par (i r)
      l[i], l[r] = l[r], l[i]
  return l
```

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Lemme

le nombre de permutations de taille n *est* $n!$

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Lemme

le nombre de permutations de taille n est $n!$

Corollaire

un algorithme de tri doit avoir $n!$ comportements différents (au moins) sur les entrées de taille n

Borne inférieure pour la complexité des tris par comparaisons

Lemme

un algorithme de tri par comparaisons est correct si et seulement s'il trie correctement toutes les permutations

Lemme

le nombre de permutations de taille n *est* $n!$

Corollaire

un algorithme de tri doit avoir $n!$ *comportements différents (au moins) sur les entrées de taille* n

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ comparaisons dans le pire cas parmi les entrées de taille n

Question : c'est gros comment, $\log_2 n!$?

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Question : c'est gros comment, $\log_2 n!$?

Théorème

$$\log_2 n! \in \Theta(n \log n)$$

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

un algorithme de tri par comparaisons fait au moins $\log_2 n!$ *comparaisons dans le pire cas parmi les entrées de taille* n

Question : c'est gros comment, $\log_2 n!$?

Théorème

$$\log_2 n! \in \Theta(n \log n)$$

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en

$$\Omega(n \log n)$$

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Questions :

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Questions :

- existe-t-il des algorithmes de tri de complexité $\Theta(n \log n)$ en moyenne ? dans le pire cas ?

Borne inférieure pour la complexité des tris par comparaisons

Corollaire

la complexité dans le pire cas (et en moyenne) d'un algorithme de tri par comparaisons est en $\Omega(n \log n)$

Rappel : le tri par sélection est de complexité $\Theta(n^2)$ dans tous les cas, de même que le tri par insertion dans le pire cas

Questions :

- existe-t-il des algorithmes de tri de complexité $\Theta(n \log n)$ en moyenne ? dans le pire cas ?
- quid de la complexité en moyenne du tri par insertion ?

Tri par fusion

Une étape : la fusion de listes triées

2 **3** **6** **8**   **1** **4** **5** **7**

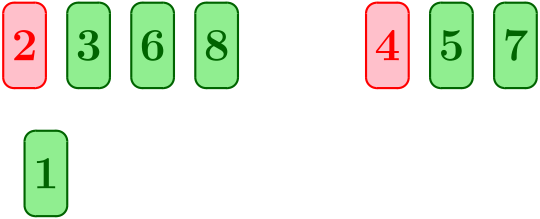
Tri par fusion

Une étape : la fusion de listes triées

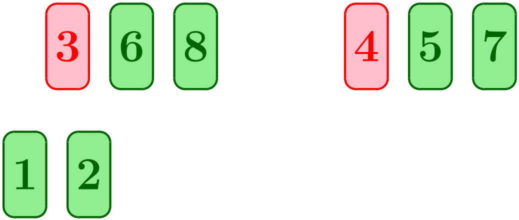
2 **3** **6** **8**   **1** **4** **5** **7**

Tri par fusion

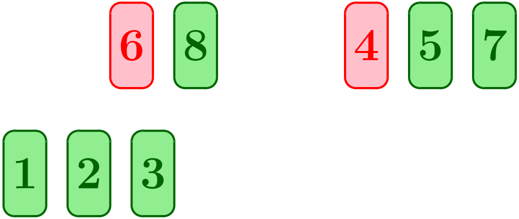
Une étape : la fusion de listes triées



Une étape : la fusion de listes triées



Une étape : la fusion de listes triées



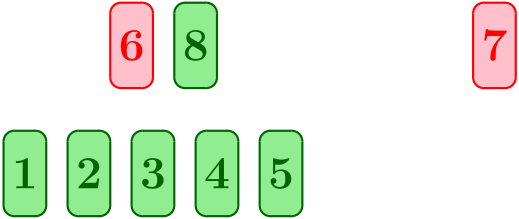
Tri par fusion

Une étape : la fusion de listes triées

6 **8** **5** **7**

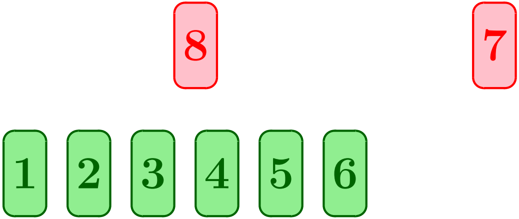
1 **2** **3** **4**

Une étape : la fusion de listes triées

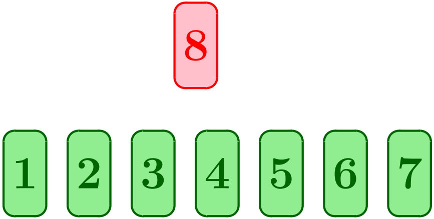


Tri par fusion

Une étape : la fusion de listes triées



Une étape : la fusion de listes triées



Tri par fusion

Une étape : la fusion de listes triées

1 **2** **3** **4** **5** **6** **7** **8**

Une étape : la fusion de listes triées

2	3	6	8		1	4	5	7

1	2	3	4	5	6	7	8

Une étape : la fusion de listes triées

```
def fusion(L1, L2) :
  if len(L1) == 0 : return L2
  elif len(L2) == 0 : return L1
  elif L1[0] < L2[0] :
    return [L1[0]] + fusion(L1[1:], L2)
  else :
    return [L2[0]] + fusion(L1, L2[1:])
```

Tri par fusion

Exemple :

3	**5**	**1**	**7**	**4**	**6**	**2**

Tri par fusion

Exemple :

3 **5** **1** **7**   **4** **6** **2**

Exemple :

3 **5**   **1** **7**   **4** **6** **2**

Tri par fusion

Exemple :

3 **5**   **1** **7**   **4** **6** **2**

Exemple :

3	**5**		**1**	**7**		**4**	**6**	**2**

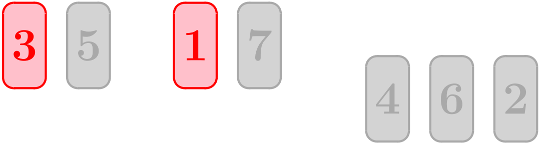
Tri par fusion

Exemple :

3 **5**   **1** **7**   **4** **6** **2**

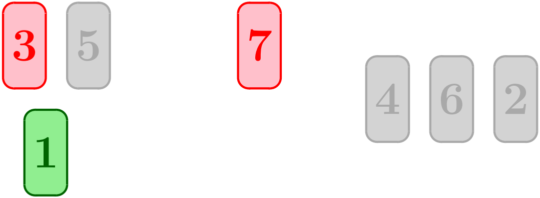
Tri par fusion

Exemple :

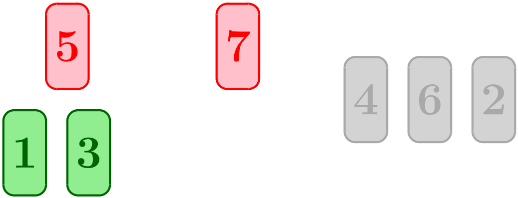


Tri par fusion

Exemple :

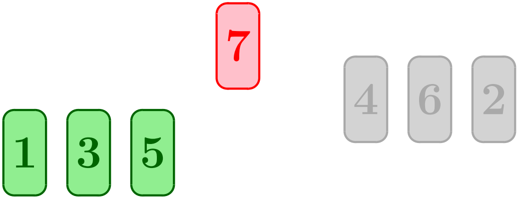


Exemple :



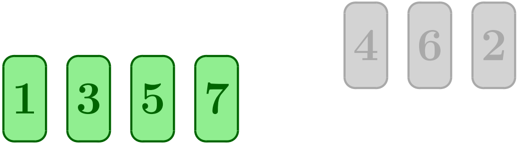
Tri par fusion

Exemple :



Tri par fusion

Exemple :



Exemple :

1	3	5	7		4	6	2

Tri par fusion

Exemple :

1 3 5 7 4 6 2

Exemple :

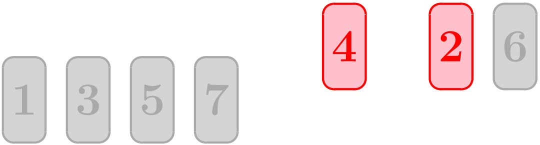
Tri par fusion

Exemple :

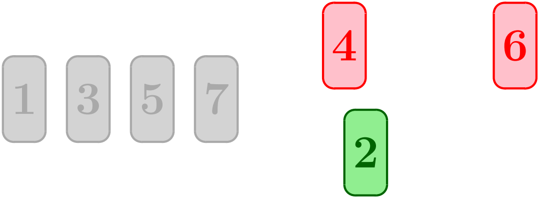
Exemple :

Tri par fusion

Exemple :

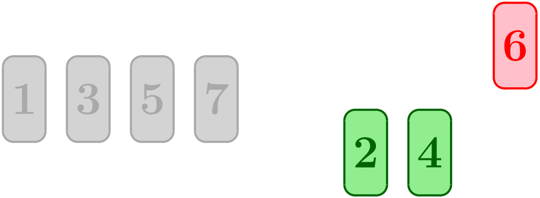


Exemple :



Tri par fusion

Exemple :



Exemple :

Exemple :

1 3 5 7   2 4 6

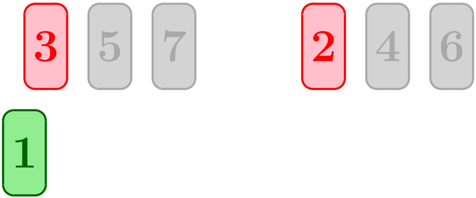
Tri par fusion

Exemple :

1 3 5 7   2 4 6

Tri par fusion

Exemple :



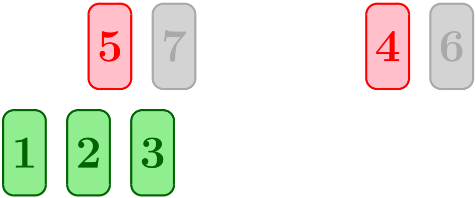
Exemple :

3 5 7   4 6

1 2

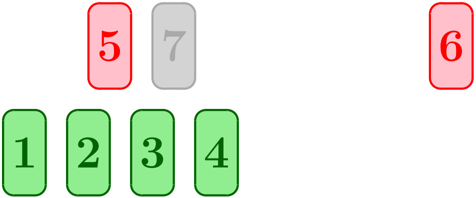
Tri par fusion

Exemple :

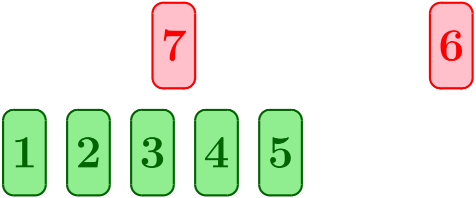


Tri par fusion

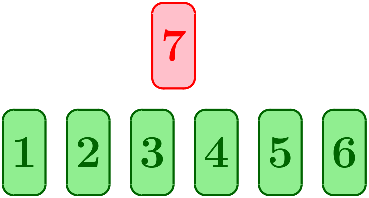
Exemple :



Exemple :



Exemple :



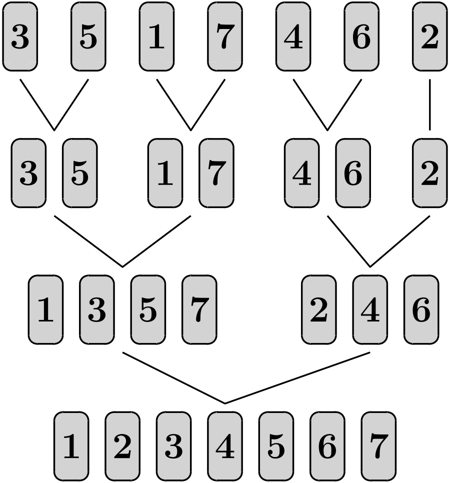
Exemple :

1 **2** **3** **4** **5** **6** **7**

Exemple :

1	**2**	**3**	**4**	**5**	**6**	**7**

Exemple :



```
def tri_fusion(T, debut, fin) :
  if fin - debut < 2 : return T[debut:fin]
  else :
    milieu = (debut + fin)//2
    gauche = tri_fusion(T, debut, milieu)
    droite = tri_fusion(T, milieu, fin)
    return fusion(gauche, droite)
```

Tri par fusion

- $\Theta(n \log n)$ comparaisons dans tous les cas,
- la constante cachée dans le Θ est importante,
- ne trie pas en place : complexité en espace $\in \Theta(n)$

Corollaire

Le tri fusion est un tri par comparaison asymptotiquement optimal.