Tri rapide : terminaison, correction, complexités

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.
- On donne une version pour les listes. On peut en donner une version *en place* pour des tableaux.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.
- On donne une version pour les listes. On peut en donner une version *en place* pour des tableaux.
- Mis au point en 1960 par Tony Hoare, alors étudiant en visite à l'université d'État de Moscou.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.
- On donne une version pour les listes. On peut en donner une version *en place* pour des tableaux.
- Mis au point en 1960 par Tony Hoare, alors étudiant en visite à l'université d'État de Moscou.
- Possède une variante *Quick select* pour le calcul du k-ième élément d'une liste sans procéder au tri préalable (application : calcul de la médiane).

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;
  - et il n'y a pas de boucle ou d'appel à une fonction qui ne termine pas.

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;
  - et il n'y a pas de boucle ou d'appel à une fonction qui ne termine pas.
  - C'est tout bon !

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;
  - et il n'y a pas de boucle ou d'appel à une fonction qui ne termine pas.
  - C'est tout bon !
- Complexité temporelle : dans tous les cas de la forme $C_n = C_{n-1} + 1$ pour une liste de taille n. Linéaire.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.
  - Correction OK. Le cas $t > p$ est laissé au lecteur.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.
  - Correction OK. Le cas $t > p$ est laissé au lecteur.

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).
- L'appel à `partition` termine (déjà vu).

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).
- L'appel à `partition` termine (déjà vu).
- Seulement deux appels internes, tous deux effectués avec des listes de taille strictement inférieure à $|\ell|$ (puisque $|\ell_1| + |\ell_2| = |\ell| - 1$).

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).
- L'appel à `partition` termine (déjà vu).
- Seulement deux appels internes, tous deux effectués avec des listes de taille strictement inférieure à $|\ell|$ (puisque $|\ell_1| + |\ell_2| = |\ell| - 1$).
- Terminaison OK.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les élements plus grands que le pivot rangés dans l'ordre.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les élements plus grands que le pivot rangés dans l'ordre.
  - contient exactement tous les éléments de `q` (puisque la réunion de `l1` et `l2` les contient) plus l'élément manquant `t`.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les élements plus grands que le pivot rangés dans l'ordre.
  - contient exactement tous les éléments de `q` (puisque la réunion de `l1` et `l2` les contient) plus l'élément manquant `t`.
  - La concaténation est donc la version triée de `l`. Hérédité OK

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre (obtenu par correction de `partition` + HR), puis le pivot, puis les éléments strictement plus grands que le pivot rangés dans l'ordre (correction + HR).

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre (obtenu par correction de `partition` + HR), puis le pivot, puis les éléments strictement plus grands que le pivot rangés dans l'ordre (correction + HR).
  - contient exactement tous les éléments de `q` (puisque `l1 @ l2` les contient) plus l'élément manquant `t`.

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre (obtenu par correction de `partition` + HR), puis le pivot, puis les éléments strictement plus grands que le pivot rangés dans l'ordre (correction + HR).
  - contient exactement tous les éléments de `q` (puisque `l1 @ l2` les contient) plus l'élément manquant `t`.
  - La concaténation est donc la version triée de `l`. Hérédité OK.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.
- Alors T_n vérifie une relation de la forme $T_n = T_{n-1} + n - 1$ ($n-1$: nombre exact de comparaisons dans `partition`) et $T_0 = 0$ (aucune comparaison).
  Remarquons au passage que $T_1 = 0$ (aucune comparaison) et que $T_{1-1} + 1 - 1 = 0 = T_1$.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.
- Alors T_n vérifie une relation de la forme $T_n = T_{n-1} + n - 1$ ($n-1$: nombre exact de comparaisons dans `partition`) et $T_0 = 0$ (aucune comparaison).
  Remarquons au passage que $T_1 = 0$ (aucune comparaison) et que $T_{1-1} + 1 - 1 = 0 = T_1$.
- Complexité en somme des premiers entiers soit $O(n^2)$.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.
- Alors T_n vérifie une relation de la forme $T_n = T_{n-1} + n - 1$ ($n-1$: nombre exact de comparaisons dans `partition`) et $T_0 = 0$ (aucune comparaison).
  Remarquons au passage que $T_1 = 0$ (aucune comparaison) et que $T_{1-1} + 1 - 1 = 0 = T_1$.
- Complexité en somme des premiers entiers soit $O(n^2)$.
- On montre dans le transparent suivant que ce cas est bien le pire, c'est à dire que la complexité C_n en nombre de comparaisons est la pire si le pivot est à une extrémité et donc que $C_n = T_n$.

Tri rapide : complexité en nombre de comparaisons

Le cas où la liste est triée est le pire (1)

- HR(n) : pour tout $q \leq n$: 1) T_q est la pire complexité et 2)

$$\forall k \in \{1, \ldots, n\}, \; T_{k-1} + T_{n-k} + n - 1 \leq T_n \text{ (égalité si } k = 1)$$

Tri rapide : complexité en nombre de comparaisons

Le cas où la liste est triée est le pire (1)

- HR(n) : pour tout $q \leq n$: 1) T_q est la pire complexité et 2)

$$\forall k \in \{1, \ldots, n\}, \; T_{k-1} + T_{n-k} + n - 1 \leq T_n \text{ (égalité si } k = 1)$$

- Cas de base $n = 1$. $T_0 = 0 = T_1$, $k \in \{1\}$. Alors $T_{1-1} + T_{1-1} + 1 - 1 = 0 \leq T_1$. Et T_1 est la pire cpx. HR(1) OK.

Tri rapide : complexité en nombre de comparaisons

Le cas où la liste est triée est le pire (1)

- HR(n) : pour tout $q \leq n$: 1) T_q est la pire complexité et 2)

$$\forall k \in \{1, \ldots, n\}, \; T_{k-1} + T_{n-k} + n - 1 \leq T_n \text{ (égalité si } k = 1)$$

- Cas de base $n = 1$. $T_0 = 0 = T_1$, $k \in \{1\}$. Alors $T_{1-1} + T_{1-1} + 1 - 1 = 0 \leq T_1$. Et T_1 est la pire cpx. HR(1) OK.
- Si HR(n) est vérifiée. Soit $k \in \{1, \ldots, n+1\}$

$$T_{k-1} + T_{n+1-k} + n \underbrace{=}_{\text{def. de } T} T_{k-1} + T_{n-k} + (n-k) + n - 1 + 1 \underbrace{\leq}_{HR(n)}$$

$T_n + (n - k + 1) \leq T_n + n = T_{n+1}$: Point 2 OK.

Tri rapide : complexité en nombre de comparaisons

Le cas où la liste est triée est le pire (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C_{n+1}^{ℓ} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C_k^{\ell_1}$ (resp. $C_{n-k}^{\ell_2}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).

Tri rapide : complexité en nombre de comparaisons

Le cas où la liste est triée est le pire (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C^{ℓ}_{n+1} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C^{\ell_1}_{k}$ (resp. $C^{\ell_2}_{n-k}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).
- Si ℓ_i est vide, $C^{\ell}_{n+1} = C^{\ell_j}_{n} + n \leq T_n + n$ par HR.1 (puisque $|\ell_i| \leq n$) donc $C^{\ell}_{n+1} \leq T_{n+1}$.

Tri rapide : complexité en nombre de comparaisons

Le cas où la liste est triée est le pire (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C^{ℓ}_{n+1} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C^{\ell_1}_{k}$ (resp. $C^{\ell_2}_{n-k}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).
- Si ℓ_i est vide, $C^{\ell}_{n+1} = C^{\ell_j}_{n} + n \leq T_n + n$ par HR.1 donc $C^{\ell}_{n+1} \leq T_{n+1}$.
- Si $|\ell_1| \times |\ell_2| \neq 0$ alors $1 \leq k \leq n-1$ et :

$$\begin{aligned} C^{\ell}_{n+1} &= C^{\ell_1}_{k} + C^{\ell_2}_{n-k} + n \\ &\underbrace{\leq}_{\text{HR}(n)} T_k + T_{n-k} + n \underbrace{=}_{\substack{k'=k+1 \\ k' \in [\![2,n]\!]}} T_{k'-1} + T_{n+1-k'} + n \\ &\leq T_{n+1} \text{ (d'après la preuve du transparent précédent)} \end{aligned}$$

Tri rapide : complexité en nombre de comparaisons

Le cas où la liste est triée est le pire (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C^{ℓ}_{n+1} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C^{\ell_1}_{k}$ (resp. $C^{\ell_2}_{n-k}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).
- Si ℓ_i est vide, $C^{\ell}_{n+1} = C^{\ell_j}_{n} + n \leq T_n + n$ par HR.1 donc $C^{\ell}_{n+1} \leq T_{n+1}$.
- Si $|\ell_1| \times |\ell_2| \neq 0$ alors $1 \leq k \leq n-1$ et :

$$\begin{aligned} C^{\ell}_{n+1} &= C^{\ell_1}_{k} + C^{\ell_2}_{n-k} + n \\ &\underbrace{\leq}_{\text{HR}(n)} T_k + T_{n-k} + n \underbrace{=}_{\substack{k'=k+1 \\ k' \in [\![2,n]\!]}} T_{k'-1} + T_{n+1-k'} + n \\ &\leq T_{n+1} \text{ (d'après la preuve du transparent précédent)} \end{aligned}$$

- Ainsi, ℓ étant quelconque, T_{n+1} est la pire complexité possible. IZP !

Complexité moyenne en nombre de comparaisons

On suppose que les listes à trier n'ont pas de doublon et que toutes les tailles de première listes après partitionnement sont équiprobables.

- Soit P la variable aléatoire qui correspond à la taille de la première liste après appel à `partition` pour une liste de taille n. Valeurs prises dans $[\![0, n-1]\!]$.

Complexité moyenne en nombre de comparaisons

On suppose que les listes à trier n'ont pas de doublon et que toutes les tailles de première listes après partitionnement sont équiprobables.

- Soit P la variable aléatoire qui correspond à la taille de la première liste après appel à `partition` pour une liste de taille n. Valeurs prises dans $[\![0, n-1]\!]$.
- La *complexité moyenne* $C(n)$ pour une liste de taille n est définie récursivement comme l'espérance suivante :

$$C(n) = \mathbb{E}[\underbrace{n-1}_{\text{partition}} + \underbrace{C(P)}_{\text{tri liste 1}} + \underbrace{C(n-1-P)}_{\text{tri liste 2}}].$$

Complexité moyenne en nombre de comparaisons

On suppose que les listes à trier n'ont pas de doublon et que toutes les tailles de première listes après partitionnement sont équiprobables.

- Soit P la variable aléatoire qui correspond à la taille de la première liste après appel à `partition` pour une liste de taille n. Valeurs prises dans $[\![0, n-1]\!]$.
- La *complexité moyenne* $C(n)$ pour une liste de taille n est définie récursivement comme l'espérance suivante :

$$C(n) = \mathbb{E}[\underbrace{n-1}_{\text{partition}} + \underbrace{C(P)}_{\text{tri liste 1}} + \underbrace{C(n-1-P)}_{\text{tri liste 2}}].$$

- Par th. du transfert (cf. maths seconde année) :

$$C(n) = \sum_{k=0}^{n-1} \mathbb{P}(P=k)\,(n-1+C(k)+C(n-1-k))$$

Complexité moyenne en nombre de comparaisons

- Par équiprobabilité $\mathbb{P}(P = k) = \frac{1}{n}$

$$C(n) = \frac{1}{n} \sum_{k=0}^{n-1} (C(k) + C(n-k-1) + n - 1)$$

Complexité moyenne en nombre de comparaisons

- Par équiprobabilité $\mathbb{P}(P = k) = \frac{1}{n}$

$$C(n) = \frac{1}{n}\sum_{k=0}^{n-1}(C(k) + C(n-k-1) + n - 1)$$

- Donc

$$C(n) = n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}(C(k) + C(n-k-1))$$

On constate que chaque terme apparaît deux fois dans cette somme.

Complexité moyenne en nombre de comparaisons

- Par équiprobabilité $\mathbb{P}(P = k) = \frac{1}{n}$

$$C(n) = \frac{1}{n}\sum_{k=0}^{n-1}(C(k) + C(n-k-1) + n - 1)$$

- Donc

$$C(n) = n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}(C(k) + C(n-k-1))$$

On constate que chaque terme apparaît deux fois dans cette somme.

- Alors $C(n) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} C(k)$. Par suite

$nC(n) = 2\sum_{k=0}^{n-1} C(k) + n(n-1)$

Complexité moyenne en nombre de comparaisons

- Par équiprobabilité $\mathbb{P}(P = k) = \dfrac{1}{n}$

$$C(n) = \frac{1}{n}\sum_{k=0}^{n-1}(C(k) + C(n-k-1) + n - 1)$$

- Donc

$$C(n) = n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}(C(k) + C(n-k-1))$$

On constate que chaque terme apparaît deux fois dans cette somme.

- Alors $C(n) = n - 1 + \dfrac{2}{n}\sum_{k=0}^{n-1} C(k)$. Par suite
$nC(n) = 2\sum_{k=0}^{n-1} C(k) + n(n-1)$
- Il vient
$nC(n) - (n-1)C(n-1) = 2C(n-1) + n(n-1) - (n-1)(n-2)$
puis $nC(n) - (n+1)C(n-1) = 2(n-1)$

Complexité moyenne du tri rapide

- Il vient

$$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$
$$= \frac{4}{n+1} - \frac{2}{n} \text{ (décomp. en éléments simples)}$$

Complexité moyenne du tri rapide

- Il vient

$$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$
$$= \frac{4}{n+1} - \frac{2}{n} \text{ (décomp. en éléments simples)}$$

- Par télescopage, et puisque $C(0) = 0$

$$\frac{C(n)}{n+1} - \frac{C(0)}{1} = \frac{C(n)}{n+1} = 4\sum_{k=1}^{n} \frac{1}{k+1} - 2\sum_{k=1}^{n} \frac{1}{k}$$
$$= 4\sum_{k=2}^{n+1} \frac{1}{k} - 2\sum_{k=1}^{n} \frac{1}{k}$$
$$= \frac{4}{n+1} - 2 + 2\sum_{k=2}^{n} \frac{1}{k}$$
$$= \underbrace{\frac{4}{n+1}}_{O(1)} - 4 + 2 \times \underbrace{\sum_{k=1}^{n} \frac{1}{k}}_{\sim \ln(n)} = \Theta(\ln(n))$$

Donc $C(n) = \Theta((n+1)\ln n) = \Theta(n \log n)$

Complexité moyenne du tri rapide

- Plus finement, on sait que $\sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n)$ où γ est la constante d'Euler ($\gamma \simeq 0.577$, cf. cours de maths).

Complexité moyenne du tri rapide

- Plus finement, on sait que $\sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n)$ où γ est la constante d'Euler ($\gamma \simeq 0.577$, cf. cours de maths).
- Alors
$$\frac{C(n)}{n+1} = (2\ln n + 2\gamma + O(1/n)) + \frac{4}{n+1} - 4 = 2\ln n + (2\gamma - 4) + O(1/n).$$