903 — Exemples d'algoritmes de tri. Complexité.

Motivations : Les tris sont d'un usage constant en informatique, souvent en tant que sous-routines de programmes. Il faut donc un algorithme efficace pour traiter cette tâche.

Pb : On considère un tableau T de longueur n d'éléments comparables. On cherche à réorganiser le tableau par ordre croissant. Un algorithme de tri doit donc satisfaire :
- la sortie est triée
- la sortie est une permutation de l'entrée

I Tris naïfs

Tri par insertion

C'est une méthode usuelle pour trier un jeu de cartes.
À l'instant j on considère que le tableau $T[1..j-1]$ est trié et l'on insère $T[j]$ dedans.

```
TRI-INSERTION(T) :
pour j=2 à n faire
  | x ← A[j],   i ← j-1
  | tant que i>0 et A[i] > x faire
  |     A[i+1] ← A[i] ,  i ← i-1
  | A[i+1] ← x
```

Terminaison : claire

Correction : au j^e passage $T[1..j-1]$ est $T_0[1..j-1]$ trié et $T[j..n] = T_0[j..n]$.
(T_0 : tableau initial)

Complexité au pire :
Ici et dans toute la suite on évalue la complexité en comptant le nombre de comparaisons entre éléments du tableau.

* Ce tri est <u>en place</u> : on ne stocke qu'un nombre borné d'éléments en dehors de T.
* Ce tri est <u>stable</u> : si deux éléments du tableau ont la même <u>clé</u>, l'algo ne change pas leur ordre relatif.

D'autres exemples de tris naïfs :
tri par sélection, tri bulles, ...

II Tris en $\Theta(n \log n)$

1/ Tri Fusion

C'est un algorithme qui utilise le paradigme "diviser pour régner".
On trie d'abord $T[1..\frac{n}{2}]$ et $T[\frac{n}{2}+1..n]$ puis on les combine.

```
TRI-FUSION(T, p, n):
  si p < n alors
    q ← ⌊(p+n)/2⌋
    TRI-FUSION(T, p, q)
    TRI-FUSION(T, q+1, n)
    FUSION(T, p, q, n)
```

```
FUSION(T, p, q, n):
  copier T[p..q] dans G
  copier T[q+1..n] dans D
  ajouter +∞ à la fin de G et D
  i ← 1, j ← 1
  pour k = p à n faire
    si G[i] ≤ D[j] alors
      T[k] ← G[i]
      i ← i+1
    sinon
      T[k] ← D[j]
      j ← j+1
```

Terminaison et correction: découlent de celles de FUSION (démontré avec un invariant de boucle).

Ce tri n'est pas en place (c'est son défaut majeur). Toutefois, il est stable.

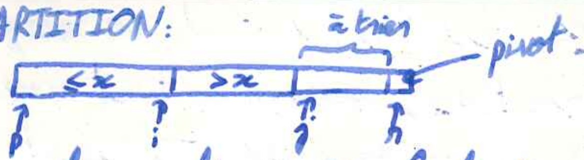
2/ Tri Rapide

Une autre utilisation de diviser pour régner:

```
TRI-RAPIDE(T, p, n):
  si p < n alors
    q ← PARTITION(T, p, n)
    TRI-RAPIDE(T, p, q-1)
    TRI-RAPIDE(T, q+1, n)
```

```
PARTITION(T, p, n):
  x ← A[n],  i ← p-1
  pour j = p à n-1 faire
    si A[j] ≤ x alors
      i ← i+1
      A[i] ↔ A[j]
  A[i+1] ↔ A[n]
  retourner i+1
```

Effet de PARTITION:



Terminaison et correction: découlent de celles de PARTITION.

Ce tri est en place et n'est pas stable.

Rq: cette version est stable...

Complexité: Au pire, le tri fusion est en $O(n \log n)$ et le tri rapide en $O(n^2)$. En moyenne, ils sont en $O(n \log n)$.

dev 1

3/ Tri par tas

Ce tri est plus complexe : il utilise la structure de tas. Mais il combine les avantages des deux tris précédents : il est en $O(n \log n)$ au pire et il est en place.

dev 2

III Est-ce optimal?

1/ Sans hypothèses supplémentaires : oui

Thm: Un tri par comparaisons utilise au moins $\Omega(n \log n)$ comparaisons en moyenne.

Preuve: avec les arbres de décision

|| Froidevaux

Rq : Il faut distinguer la théorie et la pratique : le tri rapide est souvent plus performant que le tri fusion ou le tri par tas, et pour trier moins de 30 éléments on utilise souvent un tri par insertion.

2/ Avec plus d'hypothèses : oui

Principe : plus on a d'hypothèses sur l'entrée plus on peut donner un algorithme performant.
Ex : si on sait que l'entrée est triée on a un algorithme en $O(1)$.

* tri par dénombrement

On suppose que les éléments du tableau peuvent prendre k valeurs.
On parcourt une première fois le tableau pour compter combien de fois chaque valeur apparaît dans T.
Puis on reparcourt T et on copie les éléments dans T' (une copie de T) directement au bon endroit.

Ce tri n'est pas en place, il est stable, il s'exécute en $O(n+k)$.

* tri par base

On suppose que les entrées sont des entiers donnés en base b avec au plus d chiffres.
On trie d'abord selon le chiffre de poids le plus faible, puis selon celui d'avant, ... (d passes) en utilisant un algorithme de tri stable.
Par exemple en utilisant le tri par dénombrement on obtient un algo en $O(d(n+b))$.

* tri par paquets

Cette fois on suppose que les entrées sont réparties uniformément dans $[0,1]$ et on subdivise régulièrement $[0,1]$ en n intervalles ("paquets"). ($\leftarrow n = |T|$)
Pour chaque paquet on crée une liste vide.
Puis on parcourt T en insérant chaque élément dans la liste du paquet correspondant (le $\lfloor xn \rfloor$-ième).
Ensuite on trie toutes les listes avec un tri par insertion (elles sont très courtes !).
Enfin on concatène les listes dans l'ordre.
On peut montrer que cet algorithme est en $\Theta(n)$ en moyenne.

Réf = Cormen !
↳ surtout tri fusion et rapide, pb sélection et tout III.2

Analyse comparée du Tri-Fusion et du Tri-Rapide

Ces deux tris utilisent "diviser pour régner" mais de deux façons différentes. Il est donc intéressant de comparer leur complexité.

Complexité du tri fusion

- TRI-FUSION(T, p, r) :

```
si p<r alors
  | q ← ⌊(p+r)/2⌋
  | TRI-FUSION(T, p, q)
  | TRI-FUSION(T, q+1, r)
  | FUSION(T, p, q, r).
```

Sur un tableau de taille n FUSION (cf. plan) fait toujours exactement n comparaisons. Donc on a :

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n \quad , \quad T(0) = T(1) = 0$$

- Pour simplifier la résolution de cette récurrence on admet que $T(n)$ est croissante et on calcule quand n est une puissance de 2.

$$T(2^{k+1}) = 2\,T(2^k) + 2^{k+1}$$

d'où $\dfrac{T(2^{k+1})}{2^{k+1}} = \dfrac{T(2^k)}{2^k} + 1$

donc par rec : $\dfrac{T(2^k)}{2^k} = k$ d'où $T(n) = n \log n$ $(n = 2^k)$.

Donc dans tous les cas $\underline{T(n) = \Theta(n \log n)}$ (par croissance).

<u>Rq</u> : Il existe un théorème général qui donne la solution à ce genre de récurrences.

Complexité du Tri Rapide

```
TRI-RAPIDE(T, p, r) :
  si p<r alors
   | q ← PARTITION(T, p, r)
   | TRI-RAPIDE(T, p, q-1)
   | TRI-RAPIDE(T, q+1, r)
```

PARTITION fait exactement $n-1$ comparaisons (cf. plan).

Cas le plus défavorable

↳ Quand à chaque exécution de PARTITION un des deux tableaux obtenus est vide (ie. pivot = min ou max).

Alors $C_{pire}(n+1) = n-1 + C_{pire}(n)$

d'où $C_{pire}(n) = \Theta(n^2)$.

Mais les cas pathologiques sont peu nombreux... d'où le cas moyen en $O(n \log n)$!

Cas moyen

Quand le pivot est en $j^{ème}$ position les deux sous-problèmes à traiter sont de taille $j-1$ et $n-j$.

Par hypothèse (cas moyen) toutes les permutations de l'entrée sont equiprobables donc le pivot a pour probabilité $1/n$ d'être en $j^{ème}$ position après l'exécution de partition.

On a donc :

$$C_{moy}(n) = n-1 + \frac{1}{n}\sum_{j=1}^{n}\left(C(j-1) + C(n-j)\right)$$

$$= n-1 + \frac{2}{n}\sum_{j=1}^{n-1} C(j) \qquad \text{car } C(0) = C(1) = 0$$

On fait un peu de calcul :

$$n\left(C(n) - n + 1\right) = 2\sum_{j=1}^{n-1} C(j)$$

$$(n-1)\left(C(n-1) - n + 2\right) = 2\sum_{j=1}^{n-2} C(j)$$

On soustrait et on obtient :

$$n\,C(n) - (n+1)\,C(n-1) = 2(n-1)$$

d'où $$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$

d'où $$\frac{C(n)}{n+1} \leq \frac{C(n-1)}{n} + \frac{2}{n+1}$$

Donc par rec : $C(n) \leq (n+1) \sum_{k=1}^{n} \frac{2}{k+1}$

$$= 2(n+1)(H_{n+1} - 1)$$

$$\underset{n\to\infty}{\sim} 2n \ln n$$

$$\sim 1{,}38\, n \log n$$

d'où $C_{moy}(n) = \Theta(n \log n)$.

Bilan

- Le tri fusion fait moins de comparaisons que le tri rapide ($n \log n$ vs $1{,}38\, n \log n$ en moy). Mais il est cependant plus lent, notamment car il n'est pas en place : c'est long de recopier tous les tableaux (ce développement aussi d'ailleurs). C'est donc pour cela que l'on utilise le tri rapide en pratique !

DVPT

Tri par tas max

Référence : Froidevaux, Gaudel, Soria p343

• Définition :

Un tas max est un arbre binaire A vérifiant :
- A est parfait
- Tout nœud de A a une étiquette supérieure à celles de ses fils

On va utiliser deux opérations sur ces tas, l'ajout d'un élément à un tas et la suppression du maximum

Lemme préliminaire : la hauteur h d'un arbre parfait de taille n vérifie : $\log_2(n+1)-1 \leq h < \log_2(n+1)$

Preuve : Un arbre parfait de hauteur h est un arbre complet de hauteur $h-1$ auquel on a ajouté des feuilles de profondeur h, placées de gauche à droite. En particulier :

$$\underbrace{2^h - 1}_{\text{nombre de nœuds d'un arbre complet de hauteur } h-1} < n \leq 2^{h+1} - 1$$

On en déduit $h < \log_2(n+1) \leq h+1$

d'où $\log_2(n+1) - 1 \leq h < \log_2(n+1)$

• Représentation et idée du tri

On représente notre liste d'éléments à trier par un tableau de taille n $t[1..n]$.

Ce tableau servira ensuite en cours d'algorithme à maintenir notre structure de tas, avec les conventions suivantes :

- $t[1]$ est la racine
- si pour $i > 1$ $t[i]$ est un élément du tas, alors $t\left[E\left(\frac{i}{2}\right)\right]$ est son père
- $t[2i]$ et $t[2i+1]$ sont, s'ils existent, les fils de $t[i]$
- Si p le nombre de noeuds du tas est pair, alors $t\left[\frac{p}{2}\right]$ a uniquement fils $t[p]$
- Si $i > E\left(\frac{p}{2}\right)$, $t[i]$ est une feuille

Le principe du tri va être de construire un tas à partir des éléments à trier, puis d'extraire le maximum des tas successifs pour obtenir la liste triée.

Algorithmes du tri par tas :

- Ajout d'un élément à un tas
  - Etat initial : $t[1..n]$ un tableau et $p < n$ tq $t[1..p]$ représente un tas et $t[p+1]$ l'élément à ajouter

```
Ajouter(t, p)
  i ← p+1
  tant que i > 1 et t[i] > t[E(i/2)] faire
    t[i] ↔ t[E(i/2)]
    i ← E(i/2)
  fin tant
fin
```

- Etat final : $t[1..p+1]$ représente un tas

Terminaison : pour $i > 0$ $E\left(\frac{i}{2}\right) < i$.

Correction : On ajoute le nouvel element sur une feuille et on le remonte le long de la branche si besoin est, ie si il est superieur à son père dans ce cas, il est aussi superieur à son eventuel frère, et la structure de tas est vérifiée localement après l'échange

Complexité : Dans le pire cas, on compare le nouvel element à tous les éléments d'une branche, ce qui nous donne $O(\log_2 p) = O(\log_2 n)$ comparaisons, d'après le lemme. De même pour les échanges

• Suppression du maximum du tas

• Etat initial : $t[1..n]$ un tableau et $p \leq n$ tq $t[1..p]$ represente un tas

```
Supprimer_max(t, p)
  t[1] <-> t[p]
  i <- 1
  tant que i ≤ E((p-1)/2) faire
      si 2i = p-1 ou t[2i] > t[2i+1] faire
          j <- 2i
      sinon
          j <- 2i+1
      finsi
      si t[i] < t[j]
          t[i] <-> t[j]
          i <- j
      sinon
          exit
      finsi
  fin tantque
```

• **Etat final** : $t[1,\dots,p-1]$ est un tas ne contenant pas le/l'ancien maximum qui lui se trouve en $t[p]$.

• **Terminaison** : $\forall i \geq 1$, $2i > i$, et on finit par dépasser $E\left(\frac{p-1}{2}\right)$.

• **Correction** : On place l'élément $t[p]$ à la racine puis, on l'échange avec le plus grand de ses fils (lequel est alors supérieur à son frère) tant qu'il y a des fils, pour conserver la structure de tas.

• **Complexité** : Dans le pire cas, on compare le dernier élément ramené à la racine avec tous les éléments d'une branche, ce qui nous donne $O(\log p) = O(\log n)$ comparaisons.

Algorithme final :

• **Etat initial** : t un tableau à n éléments

```
Tri_par_tas(t) :
  pour i de 1 à n-1 faire
     Ajouter(t, i)
  fin pour
  pour p de n à 2 faire
     Supprimer_max(t, p)
  fin pour
```

• **Etat final** : t est trié.

- **Terminaison** : triviale

- **Correction** :
  - A la fin de la première boucle $t[1..n]$ est un tas
  - A la i^e itération de la seconde boucle, le maximum de $t[1..n-i]$ est placé en $t[n-i+1]$

  Le tableau est donc trié

- **Complexité** :

  Chaque boucle fait intervenir une fonction de complexité en $O(\log_2 n)$.

  La complexité dans le cas le pire est donc en $O(n \log_2 n)$

Conclusion :

Le tri par tas est donc un tri stable de complexité en $O(n \log_2 n)$ dans le cas pire et en moyenne, (car le théorème d'optimalité des tris comparatifs interdit une meilleure complexité moyenne)