Chapitre I

Algorithmes de tris

1 Généralités

1.1 Différentes approches pratiques en algorithmique

En algorithmique, on retrouve souvent (mais pas toujours) les stratégies suivantes pour résoudre les problèmes :

- **pré-traitement / post-traitement** : parfois, certains algorithmes comportent une ou deux phases identifiées comme des pré-traitements (à faire avant l'algorithme principal), ou post-traitement (à faire après l'algorithme principal), pour simplifier l'écriture de l'algorithme général.
- **décomposition top-down / bottom-up** : (décomposition descendante, décomposition remontante) les décompositions top-down consistent à essayer de décomposer le problème en sous-problèmes à résoudre successivement, la décomposition allant jusqu'à des problèmes triviaux faciles à résoudre. La démarche bottom-up est la démarche inverse, elle consiste à partir d'algorithmes simples, ne résolvant qu'une étape du problème, pour essayer de les composer pour obtenir un algorithme global.
- **algorithme glouton** : il s'agit d'une méthode top-down. Lorsqu'on cherche un algorithme d'optimisation, on peut tenter une méthode simple. On décompose le problème en sous-problèmes pour lesquels on cherche une solution locale optimale (sans tenir compte des autres données du problèmes global). Puis on rassemble ces solutions pour construire une solution globale. Cette méthode fonctionne bien pour certains problèmes et peut souvent être améliorer dans un second temps.
- **diviser pour régner** : une technique usuelle consiste à diviser les données d'un problème en sous-ensembles de tailles plus petites, jusqu'à obtenir des données que l'algorithme pourra traiter au cas par cas. Une seconde étape dans ces algorithmes consiste à « fusionner » les résultats partiels pour obtenir une solution globale. Ces algorithmes sont souvent associés à la récursivité. Cette méthode est particulièrement intéressante pour chercher une alternative à un algorithme non linéaire. Si la taille des entrées est n et que l'on réussit à diviser à chaque étape la taille des données par 2, on obtient des données de taille 1 en $\log_2 n$ étapes. Comme les solutions sur les données de taille 1 sont a priori en temps constant, la complexité finale dépend surtout de l'algorithme de découpage du problème et de celui de fusion des sous-solutions.
- **recherche exhaustive (ou combinatoire ou par force brute)** : pour trouver une solution à un problème, on peut essayer de passer en revue toutes les possibilités. C'est une méthode à laquelle on peut recourir quand on ne sait rien ou presque des solutions. L'efficacité de tels algorithmes est au mieux très mauvaise, souvent catastrophique au point que les méthodes par force brute sont souvent inutilisables car il faudrait des siècles voir des milliards d'années pour terminer le calcul. Cependant, il reste des cas où il n'y a pas d'autres solutions. Exemple élémentaire : chercher un chemin dans un labyrinthe.
- **programmation dynamique** : c'est une méthode bottom-up. Lorsqu'une méthode top-down conduit à traiter plusieurs fois les mêmes sous-problèmes, il vaut mieux utiliser une méthode bottom-up. En programmation dynamique, on stocke les solutions optimales pour tous les sous-problèmes et on en

déduit des solutions optimales pour les problèmes de niveau supérieur. Progressivement, on construit la solution pour les entrées de départ.

Parmi ces méthodes, lesquelles semblent possibles pour un algorithme de tri ?

1.2 Complexité algorithmique

Dans cette partie du cours, on s'intéressera surtout à la **complexité temporelle** (dans le pire des cas, en moyenne ou dans le meilleur des cas), c.a.d. au temps d'exécution d'un algorithme.

En pratique, on décompose l'algorithme en opérations élémentaires qui s'exécutent en temps constant, puis on les dénombre en fonction de n (la taille de l'entrée).
Cependant certaines opérations à temps constant exigent beaucoup plus de temps que d'autres et cela dépend de la machine réelle qui exécutera l'algorithme. Pour s'affranchir de cette difficulté, on étudie la complexité asymptotique, c.a.d. lorsque la taille des entrées devient infiniment grande. De ce fait, on ne retient que le terme prépondérant de la complexité. Le temps relatif de chaque opération élémentaire à temps constant n'a plus d'importance, seul compte leur nombre.

On peut même se limiter à compter les d'opérations à temps constant du type le plus fréquent, par exemple les comparaisons ou les affectations ou même un bloc d'instructions à temps constant.

Finalement, on exprime la complexité asymptotique comme un 'O' (lire "grand O") d'une fonction usuelle de n. On peut préciser si cette borne est optimale (est-elle atteinte pour certaines entrées) ou non.

On classe alors les algorithmes selon le "grand O" obtenu.

Notation	Type de complexité
$O(1)$	complexité constante (indépendante de la taille de la donnée)
$O(\log(n))$	complexité logarithmique
$O(n)$	complexité linéaire
$O(n\log(n))$	complexité quasi-linéaire
$O(n^2)$	complexité quadratique
$O(n^3)$	complexité cubique
$O(n^p)$	complexité polynomiale
$O(n^p\log(n))$	complexité quasi-polynomiale
$O(2^n)$	complexité exponentielle
$O(n!)$	complexité factorielle

1.3 Classification des algorithmes de tri

On classe les algorithmes de tri selon les critères suivant :

- **tri par comparaisons** versus autres méthodes (par exemple comptage).
- **tri en place** versus tri non en place.
- **tri stable** versus instable. Un algorithme stable préserve l'ordre des éléments possédant une clef de tri identique. Si l'algorithme est instable, on peut récupérer la stabilité en ajoutant l'information sur la position de départ à chaque élément. En ajoutant à la relation d'ordre un test sur la position en cas d'égalité sur la clef de tri, le tableau trié selon ce nouvel ordre préservera bien la stabilité.
- **Complexité en temps** : on compare les temps d'exécution. Pour les tris par comparaison, on ne sait pas faire mieux qu'une complexité quasi-linéaire et cela se démontre. D'autres algorithmes ont une complexité quadratique. C'est en fait le pire que puisse faire un algorithme bien écrit puisqu'en n^2 comparaisons, on peut comparer chaque élément à tous les autres, donc on est sûr de pouvoir les classer. On peut faire mieux pour des tris qui ne sont pas par comparaison. Si on suppose que la plage des données est connue et pas trop grande, on peut trier par comptage. La complexité asymptotique est alors linéaire.

- **Complexité en ressource mémoire (complexité spatiale)**. Ce critère est souvent lié à la question des tris en place. Dès qu'un tris ne sera pas en place, cela signifie qu'il faut une deuxième structure de taille n ou parfois $n/2$. Il est également lié à la structure itérative ou récursive de l'algorithme. Les algorithmes récursifs sont très gourmands en mémoire car chaque appel récursif nécessite le stockage de tout le contexte de l'instance de la fonction qui fait l'appel (paramètres et variables).

2 Algorithmes quadratiques de tri par comparaisons

NB : dans tous les algorithmes de tri qui suivent, on considère un tableau T de longueur n indexé de 0 à n-1. `T[i..j]` désigne le sous-tableau délimité par les indices i et j.

2.1 Tri par sélection

Méthode du tri par sélection par ordre croissant

Pour i_debut variant de 0 à n-2, on recherche le minimum du sous-tableau `T[i_debut..n-1]` et on l'échange avec l'élément en position i_debut.

Une écriture possible de l'algorithme est :

Algorithme 1.

```
tri_selection(tableau T)
    pour i_debut de 0 à n-2 faire
        i_min=i_debut
        pour i de i_debut+1 à n-1 faire
            si T[i]<T[i_min] alors
                i_min=i
        si i_min != i_debut alors
            echanger T[i_min] et T[i_debut]
```

Cet algorithme trie en place mais n'est pas stable.

L'opération la plus fréquente est la comparaison. La recherche du minimum sur le sous-tableau `T[i_debut..n-1]` nécessite $n - i_{debut} - 1$ comparaisons.

On effectue donc au total $\sum_{i_{debut}=0}^{n-2} n - i_{debut} - 1 = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$ comparaisons.

La complexité temporelle est en $O(n^2)$ et est indépendante de l'ordre initial, il n'y pas de pire ou de meilleur des cas.

2.2 Tri à bulles

Méthode du tri à bulles par ordre croissant

On parcourt le tableau et si deux éléments consécutifs ne sont pas dans l'ordre, ils sont échangés. On recommence jusqu'à exécuter un parcours du tableau sans échange.

On peut remarquer aussitôt qu'après le premier parcours du tableau, le plus grand élément est forcément en fin de tableau, à sa position définitive. En effet, aussitôt que le plus grand élément est rencontré durant le parcours, il n'est pas dans l'ordre par rapport à tous les éléments suivants, donc il est échangé à chaque fois jusqu'à la fin du tableau.

De même après le second parcours, le second plus grand élément est en avant dernière position et ainsi de suite.

On peut retirer la dernière cellule de chaque nouveau parcours.

Une écriture possible de l'algorithme est :

Algorithme 2.

```
tri_bulle(tableau T)
    est_trié=false
    i_fin = n-1
    tant que i_fin > 0 et NON(est_trié) faire
        est_trié=true
        pour i de 0 à i_fin-1 faire
            si T[i]>T[i+1] alors
                echanger T[i] et T[i+1]
                est_trié=false
        décrémenter i_fin
```

Cet algorithme trie en place et est stable.

L'opération la plus fréquente est la comparaison. Le pire des cas survient si le tableau est trié initialement en ordre décroissant car la condition `T[i]>T[i+1]` est systématiquement vraie. La complexité est alors $O(n^2)$.

En moyenne, la complexité est aussi $O(n^2)$. En effet, on peut montrer que le nombre d'échanges est en moyenne égal à $n(n-1)/4$.

Le meilleur des cas survient si le tableau est trié initialement en ordre croissant. Dans ce cas, la complexité est linéaire puisqu'un seul parcours est effectué.

Il existe une variante du tri à bulles nommée tri à peigne dont la complexité théorique est difficile à analyser mais qui est en pratique aussi rapide que des algorithmes de tri quasi-linéaires.

2.3 Tri par insertion

Méthode du tri par insertion par ordre croissant

Pour chaque position i de 1 à $n-1$, on sait que les éléments précédents la position i sont dans l'ordre croissant et on insère l'élément $T[i]$ parmi eux selon l'ordre croissant.

Une écriture possible de l'algorithme est :

Algorithme 3.

```
    tri_insertion(tableau T)
        pour i de 1 à n-1 faire
            tmp = t[i]
            j = i
            tant que j > 0 et T[j-1] > tmp
                T[j] = T[j-1]
                décrémenter j
            T[j] = tmp
```

C'est un tri stable et en place.

Le pire des cas survient lorsque le tableau est trié à l'envers. La boucle interne se répète alors jusqu'à $j = 0$, c.a.d. i fois et elle s'exécute à temps constant. Son nombre total d'exécution est $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.

La complexité dans le pire des cas est donc en $O(n^2)$.

Si les éléments sont distincts et que la distribution est uniforme, la complexité en moyenne de l'algorithme est de l'ordre de $O(n^2)$.

Le meilleur des cas survient lorsque le tableau est déjà trié. La boucle interne ne s'exécute jamais et la complexité est donc linéaire.

Bien que le tri par sélection, le tri à bulles et le tri par insertion sont tous les trois quadratiques, leurs efficacités ne sont pas équivalentes dans la pratique (cf. TP). Le tri par insertion est un bon algorithme de tri pour les entrées de petites tailles.

Le tri Shell est une variante du tri par insertion. Le calcul de complexité du tri Shell pose problème mais son efficacité dans la pratique est proche des algorithmes quasi-linéaires.

3 Algorithmes quasi-linéaires de tri par comparaisons

NB : dans tous les algorithmes de tri qui suivent, on considère un tableau T de longueur n indexé de 0 à n-1. `T[i..j]` désigne le sous-tableau délimité par les indices i et j.

3.1 Tri fusion

C'est un **algorithme top-down de type diviser pour régner**.

Méthode du tri fusion pour l'ordre croissant

- Si le tableau n'a qu'un élément, il est déjà trié.
- Sinon, séparer le tableau en deux parties à peu près égales.
- Trier récursivement les deux parties avec l'algorithme du tri fusion.
- Fusionner les deux tableaux triés en un seul tableau trié.

Une écriture possible de l'algorithme de tri est :

Algorithme 4.

```
tri_fusion(tableau T)
    n= longueur(T)
    si n<=1 alors
              renvoyer T
    m = longueur(T)//2
    Tg=tri_fusion(T[0..m-1])
    Td=tri_fusion(T[m..n-1])
    return fusionner(Tg, Td)
```

Méthode de la fusion de deux tableau trié Tg et Td dans un tableau T

Pour k de 0 à n-1, on copie en `T[k]` le plus petit de `Tg[g]` et `Td[d]`.
Quand on a épuisé Tg ou Td, on recopie systématiquement les éléments de l'autre tableau.

Une écriture possible de l'algorithme de fusion est :

Algorithme 5.

```
fusionner(Tableau Tg, Tableau Td)
    créer un tableau T de longueur n=longueur(Tg)+longueur(Td)
    g = 0
    d = 0
    pour k de 0 à n-1 faire
        # si g est out of range, on copie Td[d] dans T
        si g >= longueur(Tg)
            T[k] = Td[d]
            incrementer d
        # sinon si d est out of range, on copie Tg[g] dans T
        sinon si d >= longueur(Td)
            T[k] = Tg[g]
            incrementer g
        # sinon g et d sont valides, on copie dans T le plus petit de Tg[g] et Td[d]
        sinon
            si Tg[g] <= Td[d]
                T[k] = Tg[g]
                incrémenter g
            sinon
                T[k] = Td[d]
                incrémenter d
    renvoyer T
```

Cet algorithme est récursif et se termine effectivement car la longueur des tableaux diminuent strictement à chaque appel.

Il ne trie pas en place à cause de la fusion. Il est stable si on privilégie dans la fusion les éléments de Tg à ceux de Td.

La complexité de la fusion est linéaire en fonction de la somme des tailles des deux entrées. Par le Master Theorem ou par la méthode de l'arbre des appels récursifs, on montre que la complexité asymptotique est $O(n \log n)$.

Cependant, la complexité pratique est assez mauvaise pour les entrées de petites tailles. Il vaut mieux utiliser un algorithme hybride qui applique un tri par insertion ou une variante dès que la taille du tableau est suffisamment faible.

3.2 Tri rapide (Quick Sort)

C'est un **algorithme top-down de type diviser pour régner**.

Méthode du tri rapide

- La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. C'est le partitionnement.
- Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à des sous-tableaux de taille 1.

Une écriture possible de l'algorithme est :

Algorithme 6.

```
tri_rapide(tableau T, entier indice_debut, entier indice_fin)
        si indice_debut < indice_fin alors
            indice_pivot = choix_pivot(T, indice_debut, indice_fin)
            indice_pivot = partitionner(T, indice_debut, indice_fin, indice_pivot)
            tri_rapide(T, indice_debut, indice_pivot-1)
            tri_rapide(T, indice_pivot+1, indice_fin)
```

Le choix le plus simple pour le pivot est indice_pivot=indice_debut. L'inconvénient de ce choix est que si le tableau est déjà trié, ce choix de pivot conduit systématiquement au plus mauvais partitionnement. Un choix plus efficace consiste à tirer aléatoirement le pivot entre indice_debut et indice_fin.

Conception du partitionnement

Résultat attendu du partitionnement

On peut définir le partionnement comme une permutation p des éléments de T dont le résultat est :

- $\forall i < new\_indice\_pivot, T[i] < pivot$
- $\forall i > new\_indice\_pivot, T[i] \geqslant pivot$

Où new_indice_pivot est la nouvelle position du pivot.
Noter que l'on peut positionner à gauche ou à droite les éléments égaux au pivot.

On utilise cette propriété pour imaginer un invariant de boucle portant sur des indices $j \leqslant i$,

- $\forall k \in [indice\_debut, j[, \quad T[k] < pivot$
- $\forall k \in [j, i[, \quad T[k] \geqslant pivot$

Invariant de boucle du partitionnement

Méthode pour le partitionnement

- Échanger le pivot avec le dernier élément.
- Initialiser i et j à 0 ce qui assure que l'invariant est vrai au départ.
- Faire une boucle sur i variant de 0 à indice_fin-1.
- A chaque itération, on veut que l'invariant de boucle vrai pour i devienne vrai pour $i+1$. Pour obtenir ce résultat, il suffit de faire, si `T[i]<pivot` : échanger `T[i]` et `T[j]` puis incrémenter j de 1.
- Une fois la boucle terminée, échanger le pivot en dernière position avec `T[j]`

L'algorithme de partitionnement peut s'écrire (pour diminuer le nombre d'échange et d'accès au tableau, on a utilisé une variable pivot :

Algorithme 7.

```
partitionner(tableau T, entier indice_debut, entier indice_fin, entier indice_pivot)
    pivot =T[indice_pivot]
    T[indice_pivot]  = T[indice_fin]
    # j est la position où on pourra déplacer le prochain element < pivot
    j = indice_debut
    pour i de indice_debut à indice_fin - 1
        si T[i] < pivot alors
            échanger T[i] et T[j]
            incrémenter j
    # placer le pivot avec l'élément T[j]
    T[indice_fin]=T[j]
    T[j]=pivot
    renvoyer j      # j est la nouvelle position du pivot
```

Dans cet algorithme, le partitionnement est fait en temps linéaire, en place.

Par contre ce partitionnement est non stable donc le tri rapide non stable.

Pour la complexité, le cas le plus favorable est un partitionnement systématique en deux sous-tableaux de longueur égale. Comme le partitionnement est linéaire, on se retrouve dans le cas du tri fusion et la complexité est quasi-linéaire.

Le pire des cas un partitionnement systématique avec un sous-tableau de taille 1. La complexité est alors quadratique. Ce cas se rencontre pour un tableau trié si on prend comme pivot le premier ou le dernier élément. Avec un choix aléatoire du pivot, on le retrouve encore si les éléments du tableau sont tous égaux.

On peut montrer que la complexité moyenne est quasi-linéaire si le pivot est choisi aléatoirement.

4 Algorithme linéaire de tri

4.1 Tri par dénombrement

Si le domaine des valeurs est fini, il est possible pour chacune de ces valeurs de compter ses occurrences dans le tableau T en un temps linéaire de la taille du tableau.

Supposons que le domaine des valeurs est borné par v_min et v_max, et que les indices des tableaux commencent à 0, voici un algorithme de comptage :

Algorithme 8.

```
compter(tableau T, entier v_min, entier v_max)
    créer un tableau C d'entiers de taille (v_max-v_min+1)
    # C est un tableau de compteurs, le compteur pour la valeur v est en C[v-v_min]
    pour i de 0 à longueur(T)-1 faire
        # on incrémente de 1 le compteur correspondant à la valeur T[i]
        C[T[i]-v_min] +=1
    retourner C
```

Supposons également que l'on trie des données avec plusieurs champs comprenant un champ de tri compris entre v_min et v_max.

Une idée est de parcourir le tableau T en utilisant le tableau de comptage pour calculer la position où écrire chaque donnée `T[i]` dans le tableau trié. On peut choisir cette position égale au nombre de valeurs inférieures ou égales à `T[i]` qu'il reste à écrire.

Méthode :

- Modifier C pour que pour tout i, `C[T[i]-v_min]` soit le nombre de valeurs de T inférieures où égales à `T[i]`
- Créer un nouveau tableau R de taille identique à celle de T.
- Parcourir T de droite à gauche, écrire dans R l'élément `T[i]` à la position `C[T[i]-v_min]-1` (-1 car les indices de T commencent à 0) puis décrémenter `C[T[i]-v_min]` de 1.

Notre invariant de boucle est : au début de l'itération i, pour tout v $\in [v\_min, v\_max]$, `C[v-v_min]-1` est la position dans R où écrire le prochain élément de T dont le champ de tri vaut v.

Un algorithme possible de tri par dénombrement est :

Algorithme 9.

```
tri_denombrement(tableau T, entier v_min, entier v_max)
    C=compter(T, v_min, v_max)
    pour i de 1 à longueur de C-1 faire
        C[i]= C[i]+C[i-1]
    pour i de longueur(T)-1 à 0 pas -1 faire
        # copier T[i] dans R à l'indice C[T[i]-v_min]-1
        R[C[T[i]-v_min]-1]=T[i]
        C[T[i]-v_min] -=1
    retourner R
```

La fonction `compter` est linéaire en fonction de la taille n de T.

La ligne 4 de la fonction `tri_denombrement` est à temps constant et s'exécute m fois en notant m la longueur du tableau de comptage c.a.d. le cardinal du domaine des valeurs.

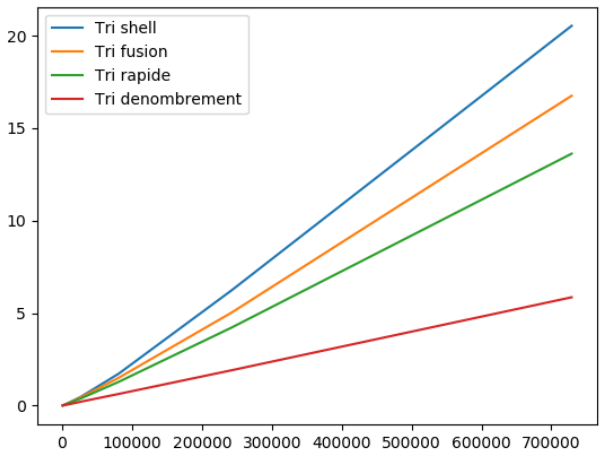
Les lignes 7 et 8 sont à temps constant et s'exécutent n fois.

La fonction `tri_denombrement` est donc un $O(n+m)$. Si le domaine des valeurs est aussi $O(n)$ alors la complexité est simplifiable en un $O(n)$.

Techniquement, cet algorithme peut s'adapter à des données non entières dès qu'on a une correspondance bijective du domaine des valeurs vers l'ensemble des indices du tableau de comptage. Pour conserver la même complexité, cette correspondance doit se faire à temps constant. Ici on faisait une simple translation $v \mapsto v - v\_min$.

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

Application : tri par base.



Comparaison des temps d'exécution d'algorithmes de tris sur des tableaux d'entiers