Introduction à l'Algorithmique

N. Jacon

1 Définition et exemples

Un algorithme est une procédure de calcul qui prend en entier une valeur ou un ensemble de valeurs et qui donne en sortie une valeur ou un ensemble de valeurs. C'est donc une séquence d'étapes qui transforme une entrée en une sortie.

Par exemple, on peut considérer le problème du tri de nombres dans un ordre croissant. L'entrée est donc ici une suite de n nombres réels que l'on peut écrire sous forme de liste :

$$(a_1, \ldots, a_n)$$

La sortie est la suite de ces n nombres écrits dans l'ordre croissant c'est à dire la suite des n nombres de la liste réordonnés :

$$(a'_1, \ldots, a'_n)$$

de sorte que $a'_1 \leq q'_2 \leq \ldots \leq a'_n$. Ainsi, la liste $(23, 32, 8, 2, 90, 12, 98)$ en entrée devra donner en sortie $(2, 8, 12, 23, 32, 90, 98)$. Le but est donc de déterminer un moyen pour passer de cette entrée à cette sortie.

Un algorithme est dit correct si pour chaque entrée possible, il se termine en produisant la bonne sortie. Dans ce cours, les algorithmes seront décrits au moyen de programme écrits en *pseudo-code* (très proche du Pascal, C, Algol). Ceci diffère néanmoins d'un langage de programmation classique :

1. le pseudo-code permet la description, la plus imagée possible, d'une suite d'actions qui ne dépendent d'aucune machine;

2. Un programme écrit dans un langage est un texte très lourd, très pointilleux, destiné à un compilateur. Il est écrit dans un langage donné.

Prenons deux exemples : le premier est très simple. On désire calculer la puissance d'un nombre réel quelconque sachant que l'on sait multiplier deux nombres réels quelconques. La donnée de départ est donc un nombre réel a et un entier strictement positif n. On définit une variable x qui sera initialisée à 1 et donnera notre résultat a^n à la fin. On va ensuite multiplier x par a pendant n étapes.

Algorithme 1: Calcul de puissance d'un nombre réel

```
1 Nom : Puissance.
2 Données : a ∈ ℝ, n ∈ ℕ;
3 Variables : x ∈ ℝ;
4 x ← 1;
5 tant que n ≠ 0 faire
6 |   a × x;
7 |   n ← n − 1;
8 fin
9 Résultat :  x
```

Voici un deuxième exemple où on désire calculer les racines d'une équation du second degré.

Algorithme 2: Calcul des racines d'un polynôme du second degré

Nom : Secondegre.
Données : $a \in \mathbb{R}$, $b \in \mathbb{R}$, $c \in \mathbb{R}$;
Variables : Δ;
si $a = 0$ **alors**
  si $b = 0$ **alors**
    si $c = 0$ **alors**
      Résultat : $\mathbb{R}$
    sinon
      Résultat : $\emptyset$
    fin
  sinon
    Résultat : $\{-c/b\}$
  fin
fin
$\Delta \leftarrow b^2 - 4 \times a \times c$;
si $\Delta < 0$ **alors**
  Résultat : $\emptyset$;
fin
si $\Delta = 0$ **alors**
  Résultat : $\{-b/(2 \times a)\}$;
fin
si $\Delta = 0$ **alors**
  Résultat : $(\{-b - \sqrt{\Delta})/(2 \times a)), (-b + \sqrt{\Delta})/(2 \times a)\}$;
fin

Les principales instructions qui vont nous servir à construire nos algorithmes sont les suivantes :

- "Si X alors Y" c'est à dire, dans le cas où X est vérifié alors le programme exécute la proposition Y.
- "Tant que X faire Y" c'est à dire, tant que la proposition X est vérifiée alors le programme exécute la proposition Y.
- " Pour $i \in I$ faire Y" c'est à dire, le programme exécute la proposition Y pour tous les éléments dans I.

Un dernier exemple très lié aux problèmes de tri que nous rencontrerons par la suite : Etant donnée une liste finie de nombres nommée "Liste", on désire ici trouver le plus petit élément de celle-ci. L'algorithme de base est simple et suit exactement le procédé intuitif que nous adoptons quand nous désirons résoudre ce problème par nos propres moyens :

- On crée une variable "mini" qui donnera à la fin de la procédure le résultat voulu.
- On prend le premier nombre de la liste "Liste" : Liste[1]. On assigne à la variable "mini" ce premier nombre.
- Ensuite, on parcourt les nombres de le liste : Liste[2], Liste[3], etc. Dès que l'on rencontre un nombre plus petit que notre "mini", la variable "mini" devient alors ce nouveau nombre de la liste.
- Quand on arrive à la fin de la liste, on a terminé : notre minimum est la valeur "mini".

On suppose donc ici que l'on dispose d'une fonction Liste[k] qui nous renvoie le $k$ième élément de la liste. On suppose aussi que l'on a une fonction "longueur" (que l'on peut facilement programmer : exercice !) qui nous renvoie la longueur (c'est à dire le nombre d'éléments) de la liste. On vérifie que l'algorithme décrit ci-dessus est correct : on a bien testé tous les éléments de la liste et gardé en mémoire le plus petit de ceux-ci.

Enfin, on prendra bien soin de traiter le cas "pathologique" où la liste est vide (c'est à dire lorsque sa longueur est 0).

Algorithme 3: Recherche du minimum dans une liste.

```
1  Nom : Minimum.
2  Données :  Liste.
3  Variables :  mini, n ∈ ℕ, i ∈ ℕ;
4  n ← Longeur(Liste);
5  si n = 0 alors
6  |  Résultat :  ∅ ;
7  fin
8  mini ← Liste[1];
9  pour i ∈ [2, n] faire
10 |  si Liste[i] < mini alors
11 |  |  mini←Liste[i];
12 |  fin
13 fin
14 Résultat :  mini
```

2 Performance d'un algorithme

Pour résoudre un problème, il existe en général plusieurs algorithmes possibles. Ces algorithmes ont en principe différents coûts en termes de :

- temps d'exécution (c'est à dire en termes de nombre d'opérations effectuées)
- de taille mémoire (taille nécessaire pour stocker les différentes structures de données pour l'exécution).

Ces deux concepts sont appelés "complexité en temps et en espace". On s'intéressera ici principalement à la complexité en temps. La complexité permet donc de mesurer l'efficacité d'un algorithme et de le comparer avec d'autre algorithmes résolvant le même problème. Cette mesure représente le nombre d'étapes qui seront nécessaires pour résoudre le problème pour une entrée de taille donnée. Il y a deux types de complexité :

- Etude du cas le plus défavorable : on estime le nombre d'opérations réalisé dans l'algorithme dans le pire des cas.
- Etude du cas moyen : on estime le nombre moyen d'opérations nécessaires pour réaliser l'algorithme en fonction de toutes les entrées possibles.

La seconde complexité est beaucoup plus difficile à calculer que la première. Nous nous concentrerons ici à l'étude de la complexité dans le cas le plus défavorable. En général :

- le temps d'exécution d'une affectation ou d'un test est considéré comme constant (ainsi les tests du type "Si X alors Y" sont considérés comme constants)
- le temps d'exécution d'une séquence d'instructions est la somme des temps d'exécution qui la compose.

- le temps d'exécution d'une condition de type

  Si *Condition1* Alors *Processus 1* sinon *Processus 2*

  est égal au temps d'exécution de "Condition1" ajouté au maximum du temps d'exécution de "Processus 1" et de "Processus 2".

- Le temps de Condition d'une boucle de type "tant que ... faire ..." ou "Pour $i = \ldots$ faire ..." est égal à la somme des coûts du test plus le coût du corps de la boucle.

Voici un exemple d'algorithme qui calcule la somme des carrés d'une liste. La procédure donnée est classique. Nous faisons ensuite une analyse de sa complexité.

Algorithme 4: Calcul d'une somme de carrés des éléments d'un tableau.

```
1  Nom Somme2carres. Données : Liste.
2  Variables : n ∈ ℕ, i ∈ ℕ, somme;
3  n ← Longeur(Liste);
4  si n = 0 alors
5  |  Résultat :  ∅;
6  sinon
7  |  somme← 0;
8  |  pour i ∈ [1, n] faire
9  |  |  somme ← somme+L[i] × L[i] ;
10 |  fin
11 fin
12 Résultat : somme.
```

Calculons la complexité de cet algorithme : on a une première affectation au début le premier "Si" fournit une première instruction/affectation, le coût est constant, disons donc que l'on réalise 1 opération. Ensuite nous avons un premier test qui coûte une opŕation puis une affectation (à l'intérreur du "sinon"). Puis, on va avoir n calcul à faire avec 3 opérations à chaque fois : une somme, une multiplication et une affectation. On obtient $3n$ opérations. On obtient $3n + 3$ opérations. Pour simplifier, on dit que la complexité est en $O(n)$. Ceci signifie que le résultat trouvé divisé par n est infèrieur à une constante ce qui est le cas ici ... Par exemple, l'algorithme 3 est aussi en $O(n)$. L'algorithme 2 est en temps constant : on dit qu'il est en $O(1)$. Les principales complexités sont les suivantes. Elles sont ici classées en ordre croissant de coût :

- $O(1)$: le temps constant,
- $O(\log(n))$: complexité logarithmique,
- $O(n)$: complexité linéaire,
- $O(n\log(n))$,
- $O(n^2)$, $O(n^3)$, ..., $O(n^p)$: complexité polynômiale,
- $O(p^x)$ complexité exponentielle.

Bien sûr, on cherche en pratique à réaliser des algorithmes avec des complexités les moins couteuses possibles. Ceci est fondamental. Voici un exemple permettant de se rendre compte de l'importance de ceci.

Example 2.1. On dispose d'un ordinateur A qui exécute un algorithme I permettant de résoudre un problème donné. On peut prendre pour exemple un problème de tri, probème que l'on étudiera par la suite. Il s'agit dans ce cas de trier n nombres dans l'ordre croissant.

On suppose que cet algorithme demande $2n^2$ opérations de sorte que la complexité est en $O(n^2)$. Un ordinateur B exécute un algorithme 2 permettant de résoudre le même problème grâce à $5n\log(n)$ opérations. Cet algorithme est donc en $O(n\log(n))$.

On suppose enfin que la puissance de calcul de l'ordinateur A est de 10^9 opérations par seconde tandis que celle de l'ordinateur B est de 10^6 opérations par seconde. Supposons qu'on veuille trier $n = 1000$ nombres.

- L'ordinateur A demande donc $\dfrac{2.1000^2}{10^9} = 2.10^{-3}$ secondes.
- L'ordinateur B demande donc $\dfrac{5.1000\log(1000)}{10^6} \simeq 5.10^{-2}$ secondes.

mais si on prend maintenant $n = 10^6$,

- L'ordinateur A demande donc $\dfrac{2.((10)^6)^2}{10^9} = 2000$ secondes.
- L'ordinateur B demande donc $\dfrac{5.10^6\log(10^6)}{10^6} \simeq 100$ secondes.

Considérons le problème de calcul de la suite de Fibonacci. Initialement, cette suite était destiné à compter le nombre de lapins génération aprés génération. Le modèle initial était en effet : partant d'un couple, combien de couples de lapins obtiendrons nous après un nombre donné de mois sachant que chaque couple produit chaque mois un nouveau couple qui ne devient lui-même "productif" qu'après deux mois ? Cette suite est entièrement déterminée par la formule suivante :

$$\forall N \geq 2,\ F(N) = F(N-1) + F(N-2),\ F(0) = F(1) = 1$$

L'algorithme qui nous vient immédiatement à l'esprit pour résoudre ce problème est le suivant :

Algorithme 5: Calcul de la suite de Fibonnacci (1).

```
1 Nom Fibo1.
2 Données : N ∈ ℕ .
3 si N ≤ 1 alors
4 |  Résultat :  1;
5 sinon
6 |  Résultat : Fibo1(N − 1) + Fibo1(N − 2).
7 fin
```

Quelle est la complexité de cet algorithme ? à constante près le nombre d'opérations nécessaires est ici $f(N)$ avec

$$f(N) = f(N-1) + f(N-2).$$

Considérons la suite $g(n) = f(N)/f(N-1)$. Si cette suite converge vers x, alors x vérife $x = 1 + 1/x$ et on a $x = (1+\sqrt{5})/2$. On peut ensuite montrer que cette suite est bien convergente. On obtient alors que lorsque N tend vers l'infini, $f(N)/f(N-1)$ tend vers x ce qui implique que $f(N)$ est en $O(x^N)$ ce qui est une complexité exponentielle !

En fait, on voit qu'avec cet algorithme, on calcul beaucoup de chose plusieurs fois. Voici un algorithme beaucoup plus rapide où tout n'est calculé qu'une et une seule fois. L'idée est de définir une liste comportant tous les éléments de la suite de Fibonacci.

Algorithme 6: Calcul de la suite de Fibonacci (2).

Nom Fibo2.
Donnée : $N \in \mathbb{N}$.
Variables : L Liste, $i \in \mathbb{N}$
$L \leftarrow [1, 1]$
pour $i \in [2, N]$ **faire**
  $L[i+1] := L[i-1] + L[i]$
fin
Résultat : $L[N+1]$.

On vérifie facilement que cet algorithme n'est qu'en $O(N)$! par contre, la complexité en taille mémoire est énorme puisqu'on stocke tous les éléments de la précédents de la suite. On peut alors améliorer cet algorithme de ce point de vue car à chaque étape, seuls les deux derniers termes de la suite sont nécessaires.

Algorithme 7: Calcul de la suite de Fibonacci (3).

Nom Fibo3.
Donnée : $N \in \mathbb{N}$.
Variables : L Liste, $i \in \mathbb{N}$, $x_1 \in \mathbb{R}$, $x_2 \in \mathbb{R}$
si $N = 0$ **alors**
  Résultat : 1.
fin
$L \leftarrow [1, 1]$;
$i \leftarrow 2$;
tant que $i \leq N$ **faire**
  $x_1 \leftarrow L[1]$;
  $x_2 \leftarrow L[2]$;
  $L[1] \leftarrow x_2$;
  $L[2] \leftarrow x_1 + x_2$;
  $i \leftarrow i + 1$;
fin
Résultat : $L[2]$.

Vérifions la justesse de cet algorithme. Si $N = 0$ où 1, notre algorithme renvoie le nombre 1 ce qui est correct. Supposons maintenant qu'au pas $i \geq 2$, l'algorithme crée une liste de deux élément dont la première valeur est F_{i-1} et la seconde F_i (l'algorithme répond donc justement au problème en renvoyant $F_i = L[2]$). Au pas $i+1$, l'algorithme crée une nouvelle liste dont le premier élément est F_i et le second $F_{i-1} + F_i = F_{i+1}$. Le résultat est donc correct.

Une classe d'algorithme particulièrement efficace est maintenant donnée par la classe d'algorithme suivante.

3 L'approche "diviser pour mieux régner" : la dichotomie

La méthode "Diviser pour mieux régner" consiste à décomposer le problème initial en sous-problèmes de petite taille puis résoudre les problèmes décomposés un par un. Pour adopter une telle méthode, on doit

- Savoir résoudre les “petits” problèmes,
- Diviser le problème initial en ces sous-problèmes,
- Résoudre les sous-problèmes par induction,
- Combiner les solutions des sous-problèmes.

Ainsi le coût de l'algorithme correspond à la somme des coûts des sous-algorithmes auquel on ajoute le coût de la recombinaison.

Nous verrons des exemples de cette approche dans la prochaine section. Pour l'instant, concentrons nous sur une méthode particulière de cette classe d'algorithme. La dichotomie (“couper en deux” en grec) est une stratégie algorithmique de cette catégorie. C' est un processus itératif ou récursif où, à chaque étape, on découpe lespace de recherche en deux parties (non forcément égale) puis ayant déterminé dans laquelle se trouve la solution, on restreint l'espace à cette partie.

Cette approche est en fait très intuitive. Voici un exemple tiré de l'excellent site **www.siteduzero.com**. Le Sphinx choisit un nombre entre 1 et 100 et le garde secret. Le but du joueur est de trouver ce nombre avec le moins de tentatives possibles. On souhaite donc déterminer un algorithme avec un coût optimal pour résoudre ce problème. A chaque proposition fausse, le joueur reçoit une indication “c'est plus” (si le nombre recherché est plus grand) ou “c'est moins”.

Une solution naïve consiste à énumérer les nombres les uns après les autres, sans utiliser les indications. On commence par 1, puis on poursuit avec 2, etc. Dans le pire des cas, on risque donc de compter jusqu'à 100. La complexité est donc de $O(n)$ (dans le pire des cas)

On peut imaginer une solution qui semble plus rapide : on commence par proposer 50. Quelque soit la réponse du Sphinx, on peut éliminer 50 possibilités :

- si c'est plus que 50, la solution est entre 50 et 100 ;
- si c'est moins, la solution est entre 1 et 50.

Et ainsi de suite. à chaque étape, on réduit donc le nombre de possibilités par deux. Cet algorithme est donc beaucoup plus efficace que le précédent. En effet, dans le pire des cas, sept propositions sont nécessaires (on verra comment calculer ce nombre plus tard).

Example 3.1. On dispose d'une liste de n nombres triés du plus petit au plus grand. Le problème posé est de savoir si un nombre p est dans cette liste. Pour simplifier, on suppose que $n = 2^k$ pour un entier k (sinon les algorithmes peuvent facilement s'adapter en prenant la partie entière). Un algorithme naïf consiste à parcourir toute la liste du premier nombre au dernier.

Algorithme 8: Recherche d'un nombre dans une liste triée.

```
1  Nom :Recherche
2  Données : L liste,x nombre.
3  Variables : n ∈ ℕ, i ∈ ℕ
4  n ← Longeur(Liste).
5  si n = 0 alors
6  |  Résultat :  Non
7  fin
8  i ← 1;
9  tant que i ≠ N + 1 faire
10 |  si Liste[i] = x; alors
11 |  |  Résultat Oui ;
12 |  sinon
13 |  |  i ← i + 1;
14 |  fin
15 fin
16 Résultat : Non;
```

La complexité de cet algorithme est en $O(n)$. Mais on peut faire beaucoup mieux : comme la liste est triée, si on choisit de comparer notre nombre avec l'élément du milieu de la liste, on saura à coup sûr, où l'élément x se trouve (ou pas). Ceci permet de diviser par 2 à chaque étape le nombre de comparaisons :

Algorithme 9: Recherche d'un nombre dans une liste triée par dichotomie.

```
1  Nom :Rechercheopt
2  Données : L liste,x ∈ ℝ.
3  Variables : n ∈ ℕ
4  n ← Longeur(Liste).
5  si n = 0 alors
6  |  Résultat :  Non
7  fin
8  si n = 1 alors
9  |  si Liste[1] = x alors
10 |  |  Résultat : Oui
11 |  sinon
12 |  |  Résultat Non
13 |  fin
14 fin
15 si L[n/2] < x alors
16 |  Résultat : Rechercheopt ([Liste[n/2+1],..., Liste[n]],x)
17 sinon
18 |  Résultat : Rechercheopt ([Liste[1],..., Liste[n/2]],x)
19 fin
```

Ici, si on note $f(n)$ le nombre d'opérations à faire, on voit que l'on al'èquation suivante :

$$f(n) = f(n/2) + C$$

où C est une constante. Comme on a $n = 2^k$, on voit que

$$f(n) = k.C$$

avec $k = \ln(n)/\ln(2)$. Il suit que notre algorithme est de complexité logarithmique !

4 Les algorithmes de tri

Un algorithme de tri est un algorithme permettant de résoudre le problème d'organisation d'une collection d'élément dans un certain ordre donné. Nous nous bornerons ici au problème de classer une collection d'entiers d'une liste L dans un ordre croissant. Ceci peut paraître simple mais il s'avère qu'il existe beaucoup d'algorithmes plus ou moins efficaces en rapport avec ce problème. Ceci permet d'illustrer parfaitement le concept de complexité.

4.1 Tri par sélection

Le premier exemple de tri est le plus immédiat (avec le suivant). L'idée est la suivante :

- On trouve le plus petit élément de la liste
- On le met au début de la liste,
- on cherche le deuxième plus petit élément
- on let met en second position etc.

Pour décrire cet algorithme, on suppose que l'on dispose d'un algorithme Indicemini (L,i,n) qui donne l'indice du plus petit élément de la liste $[L[i], \ldots, L[n]]$ (il suffit d'adapter un peu notre algorithme 3 en $O(n)$). Cet algorithme est en $O(n)$.

Algorithme 10: Tri par sélection.

```
1 Nom : Triselection
2 Données : L liste.
3 Variables : n ∈ ℕ, i ∈ ℕ , mini ∈ ℝ
4 n ← Longeur(Liste).
5 pour i ∈ [1, n − 1] faire
6 |   mini ← indicemini(L, i, n);
7 |   L[i] ↔ L[mini];
8 fin
9 Résultat : L
```

Ici $\leftrightarrow$ signifie que l'on a simplement échangé les places des éléments en question dans la liste. Montrons maintenant que cet algorithme est correct: en fait, on va montrer qu'à chaque ètape i à l'intérieur de la boucle, les i premiers éléments de la liste sont triés et plus petits que les termes restants. Ceci se fait dans le même esprit qu'une preuve par récurrence :

- Initialisation : on montre que la propriété est vraie avant la première itération. Ceci est évident dans notre cas.

- Conservation : on montre que si cette propriété est vraie avant une itération, elle l'est après. Supposons que les i premiers éléments soient triés et plus petits que les suivants. On choisit ensuite le minimum de la liste $[L[i+1], \ldots, L[n]]$ sachant que ce minimum est plus grand que tous les éléments déjà triés. A la fin de l'iteration $i+1$, la liste $[L[1], \ldots, L[i+1]]$ est donc triée et ses éléments sont plus petits que les autres termes.
- terminaison : la boucle se termine lorsque $i = n-1$, la liste L est donc triée car les $n-1$ premiers termes le sont et sont plus petits que le dernier élément.

Calculons la complexité. On a une boucle $n-1$ opérations faisant appelle à une affectation en temps constant et une fonction qui demande $n-i$ opérations. On a donc un algorithme de complexité $O(f(n))$ où $f(n) = n+n-1+(n-2)+\ldots+1 = n(n+1)/2$. La complexité est donc de $O(n^2)$. Passons à un nouveau tri en rapport avec celui-ci.

4.2 Tri par insertion

L'idée est ici la suivante :

- On commence par ordonner les deux premiers éléments de la liste,
- on insère le troisème élément de telle façon que les trois premiers éléments soient ordonnés. Pour ceci, on procède de la façon suivante :
  - si le troisième élément est plus grand que le deuxième alors les 3 nombres sont déja bien classés,
  - sinon, on échange le 3eme élément et le second et on insère le second dans la liste constituée du premier élément.
- on procède de la même manière pour le quatrième élément, le cinquième etc.

Algorithme 11: Tri par insertion.

```
1  Nom : Triinsertion
2  Données : L liste.
3  Variables : n ∈ ℕ, i ∈ ℕ, x ∈ ℝ
4  n ← Longeur(Liste).
5  pour i ∈ [2, n] faire
6  |  x ← L[i];
7  |  j ← i;
8  |  tant que j > 1 et L[j − 1] > x faire
9  |  |  L[j] ← L[j − 1];
10 |  |  j ← j − 1;
11 |  fin
12 |  L[j] ← x;
13 fin
14 Résultat : L
```

Etudions la correction de cet algorithme. Là encore, on montre que à l'étape i, les $i-1$ premiers éléments de la liste sont triés dans le bon ordre.

- Initialisation : on montre que la propriété est vraie avant la première itération. Ceci est évident dans notre cas.

- Conservation : on montre que si cette propriété est vraie avant une itération, elle l'est après. Supposons que les $i-1$ premiers éléments soient triés. On prend ensuite l'élément $x := L[i]$ et on l'insère dans la liste consituée des $i-1$ premiers éléments qui sont donc déjà triés. On parcourt les éléments de cette liste et on échange x et $L[j-1]$ si $L[j-1] > x$ de sorte que x est inséré à une place tel que k tel que $L[k-1] < x < L[k+1]$. On obtient bien une liste de i éléments triés.
- terminaison : la boucle se termine lorsque $i = n$, la liste L est donc trié.

Quelle est la complexité (dans le cas le plus défavorable) ? Comme dans l'algorithme précédent, à constante près, le nombre d'opérations nécessaires est

$$2 + 3 + \ldots + n = n(n+1)/2 - 1.$$

La complexité est donc encore une fois en $O(n^2)$. Cependant, cet algorithme peut nécessiter beaucoup moins d'opérations que le premier selon la nature de la liste !

4.3 Tri par fusion

Le tri par fusion est un exemple parfait pour décrire la procédure "diviser pour mieux régner". Voici l'idée géneral que l'on décrit sur un exemple de façon à rendre la procédure plus claire. Imaginons que l'on doit faire un tri sur la liste suivante

$$L := [1, 8, 3, 0, 12, 4, 6, 9]$$

On commence par diviser notre tableau en 2 :

$$L_1 = [1, 8, 3, 0],\ L_2 := [12, 4, 6, 9]$$

par récurrence, on sait trier les listes L_1 et L_2 car leurs longueurs est plus petite que celle de L. En les triant, on obtient donc deux listes :

$$L_1' = [0, 1, 3, 8],\ L_2 = [4, 6, 9, 12]$$

Maintenant, on va fusionner nos deux tableaux de faon à avoir une liste trié. Pour ceci : on compare d'abord 0 et 4, le plus petit est 0 :

$$L = [0],\ L_1' = [1, 3, 8],\ L_2 = [4, 6, 9, 12]$$

Ensuite, on compare 1 et 4, le plus petit est 1 :

$$L = [0, 1],\ L_1' = [3, 8],\ L_2 = [4, 6, 9, 12]$$

puis encore :

$$L = [0, 1, 3],\ L_1' = [8],\ L_2 = [4, 6, 9, 12]$$

et on continue jusque'à ce que les deux liste L_1' et L_2' soient vides. On obtient

$$L = [0, 1, 3, 4, 6, 8, 9, 12]$$

Plus généralement l'idée est donc

- d'initialiser en disant qu'une liste de 1 nombre est déjà convenablement triée.
- Si notre liste comporte plus de 1 nombre, on divise notre liste en deux sous-listes que l'on trie par récurrence.
- on fusionne les deux listes obtenues

Ecrivons un algorithme permettant de fusionner deux listes :

Algorithme 12: Fusion de deux listes triées.

Nom : Fusion
Données : L_1 liste L_2 liste.
Variables : $n \in \mathbb{N}$, $\mathcal{L}$ liste
si $L_1 = []$ **alors**
  Résultat : L_2;
fin
si $L_2 = []$ **alors**
  Résultat : L_1;
fin
si $L_1[1] < L_1[2]$ **alors**
  Résultat : Concatenation($[L_1[1]]$, Fusion($[L_1[2], \ldots, L_1[$, Longueur$[L_1]]]$), L_2)
sinon
  Résultat : Concatenation($[L_2[1]]$, Fusion(L_1, $[L_2[2], \ldots, L_2[$Longueur$[L_2]]]$))
fin

Ici la fonction "Concatenation" concatène simplement deux listes données et c'est une fonction de complexité négligeable. Cet algorithme est en $O(n)$ et il permet effectivement de fusionner deux listes triées en une liste triée. Pour montrer ceci, on fait un récurrence sur la longueur de L_1.

- Si L_1 est une liste vide alors le programme retourne L_2 qui est déjà convenablement trié.
- Si L_1 est de longueur n, par récurrence, la procédure de fusion retourne une liste fusionnée triée. On voit que les deux cas considérés dans l'algorithme permet de retourner la liste voulue.
- L'algorithme se termine car forcément, la liste L_1 ou L_2 est vide à partir d'un instant.

Algorithme 13: Tri par fusion.

Nom : Trifusion
Données : L liste.
Variables : $n \in \mathbb{N}$, L_1 liste, L_2 liste, $\mathcal{L}$ liste
$n \leftarrow$ Longeur(L).
si $n = 1$ *ou* $n = 0$ **alors**
  Résultat L
fin
$L_1 \leftarrow$ Trifusion$([L[1], \ldots L[[n/2]]])$;
$L_2 \leftarrow$ Trifusion$([L[[n/2]+1], \ldots L[n]])$;
Résultat : Fusion(L_1, L_2)

Il est clair que cet algorithme retourne bien la liste triée voulue :

- Si L_1 est une liste vide ou de longueur 1, l'algorithme renvoie la liste elle-même qui est bien triée.
- sinon on fusionne deux listes déjà bien triées par récurrence. On obtient donc bien la liste voulue.
- L'algorithme se termine car nécessairement, les listes considérées deviennent de longueur 1.

Quelle est la complexité de cet algorithme ? Si on note $f(n)$ le nombre d'opérations effectués, à constante près, on a :

$$f(n) = 2f(n/2) + n$$

ce qui nous donne, pour simplifier, si on pose $n = 2^k$:

$$f(2^k) = (k+1)2^k$$

ce qui nous donne un algorithme de complexité en $O(n\ln(n))$.

4.4 Tri rapide

Le tri rapide est de la même famille que le tri par fusion, l'idée est ici de développer cette idée de fusion de façon à obtenir un algorithme permettant de résoudre le problème plus rapidement. Il aura en fait sensiblement la même complexité mais sera plus efficace en pratique et fait partie de la catégorie des algorithmes "diviser pour mieux régner".

La méthode consiste ici à prendre un élément de la liste L et de le placer à sa place définitive en permutant tous les éléments de telle sorte que tout ceux qui sont plus petits que cet élément sont à sa gauche, et tous les plus grands à sa droite. Concrètement, nous allons choisir un pivot, disons le dernier élément de la liste (mais on peut aussi se poser le problème du choix optimal d'un tel pivot !).

- on parcourt tous les éléments de la liste, on met tous les éléments plus petits que le pivot en début de liste en faisant des échanges, tous les éléments plus grand en fin de liste
- on insère le pivot à la bonne position
- on trie de la même manière les deux listes de part et d'autres du pivot.

Prenons un exemple avec la liste $[3, 1, 7, 9, 4, 2, 5]$. Notre premier pivot est le nombre 5. On parcourt ensuite les éléments de la liste, 3 et 1 sont tous les deux plus petits que 5, 7 ne l'est pas, 9 non plus, ensuite 4 est plus petit, il faut donc échanger 7 et 4 :

$$[3, 1, 4, 9, 7, 2, 5]$$

et on garde en mémoire que le pivot doit être placé en position 4. Ensuite, 2 est plus petit que 5, il faut donc échanger 2 et 9 :

$$[3, 1, 4, 2, 7, 9, 5]$$

et on garde en mémoire que le pivot doit être placé en position 5. On a fini de parcourir la liste et on peut donc insèrer 5 en 5ième position :

$$[3, 1, 4, 2, 5, 7, 9]$$

Ensuite, on trie les deux listes à droite et à gauche de 5 : $[3, 1, 4, 2]$ et $[7, 9]$.

Algorithme 14: Tri rapide

Nom : Trirapide
Données : L liste.
Variables : x nombre,$i \in \mathbb{N}$, $j \in \mathbb{N}$
$n \leftarrow$ Longeur(Liste).
si $n = 1$ *ou* $n = 0$ **alors**
  Résultat L
fin
$i \leftarrow 1$;
$j \leftarrow n$;
tant que $i \leq j$ **faire**
  tant que $L[i] < L[n]$ **faire**
    $i \leftarrow i + 1$
  fin
  tant que $L[j] > L[n]$ **faire**
    $j \leftarrow j - 1$
  fin
  si $i \leq j$ **alors**
    $L[i] \leftrightarrow L[j]$;
    $i \leftarrow i + 1$;
    $j \leftarrow j - 1$
  fin
fin
Résultat : Concatenation(Trirapide($[L[1], \ldots, L[j-1]), L[n]$, Trirapide($[L[j+1], \ldots, L[n-1]$))

L'algorithme est correct : à chaque étape,, pour chaque pivot, on le place à la $j$ième position dans la liste et on a les propreétés suivantes :

- il y a $j - 1$ nombre plus petit à gauche de ce nombre
- il y a $n - j - 1$ nombre plus grand à sa droite.

ceci doit bien être la position finale de notre élément ce qui permet de conclure. Notons aussi que l'initialisation à $n = 0$ et $n = 1$ est correct.

Concernant la complexité, on peut montrer que cet algorithme est en $O(n^2)$ dans le pire des cas. Cependant, en pratique, il se trouve qu'il est plus efficace que le tri par fusion !

References

[1] Cormen, T.; Leierson, C.; and Rivest, R. Introduction à l'algorithmique, Dunod.