Chapitre 1 : Algorithmes et Complexités.

1. Introduction

L'informatique et l'algorithmique nécessitent de s'adapter aux contraintes matérielles et aux étapes de développement.
Cela implique qu'on fait face à un manque de précision apparent, tout en étant contraint par une grande rigidité en ce qui concerne l'ordre dans lequel les opérations doivent se dérouler.
Il faut être en mesure de distinguer les contraintes liées à la programmation (connaissance du langage, du matériel, du système) de celles liées à la notion d'algorithme elle-même.

On peut ajouter à ces difficultés le fait que l'informatique est une discipline en évolution permanente, impliquant des changements réguliers concernant les couches matérielles et logicielles, mais aussi les langages de programmation.

Le but de l'algorithmique telle qu'elle sera enseignée ici est donc de se dégager des langages de programmation pour abstraire quelques grandes idées et paradigmes[1] de l'informatique pour les expliciter et les illustrer.

2. Un peu d'histoire

L'algorithmique et les algorithmes sont apparus bien avant les ordinateurs. Il s'agissait de tentatives de transmission de moyens efficaces pour obtenir des résultats en partant d'éléments donnés et en appliquant des directives simples, étape par étape.

Exemples d'algorithmes sans lien avec l'informatique : une recette de cuisine, un mode d'emploi d'assemblage de meubles,…

Chronologie approximative :

- 1800 av. JC : les Babyloniens formulent les règles de résolution précises pour la résolution d'équations
- En chinois ancien, *shu* (règle, procédé, stratagème) est un terme employé aussi bien en mathématiques qu'en arts martiaux
- 300 ap. JC : chez les grecs, mise en place de procédés de calcul (Euclide)
- 900 ap. JC : en Perse, on trouve l'origine du mot algorithme dans un ouvrage arithmétique rédigé par Abu Ja'Far Mohammed Ibn Mûsâ al-Khowârismi, qui utilise la numérotation arabe et les règles opératoires.
- XII° siècle : en Occident s'amorce le système de numération de position encore utilisé aujourd'hui grâce à la diffusion du manuel de al-Khowârismi.

La signification du terme algorithme ou « algorisme » s'élargit ensuite progressivement pour désigner tout procédé de calcul systématique, voire automatique (sans référence à la machine).

[1] Un paradigme est une représentation du monde, et donc par extension « une représentation largement acceptée dans un domaine particulier ».

Aujourd'hui, en informatique, on peut dire qu'un algorithme est « un mode d'emploi pour résoudre un problème ». Le problème étant alors défini de l'extérieur par ses données fournies (entrées) et un résultat à obtenir (sortie).

Ex : le problème de la somme des nombres 5 et 7 peut être décrit par les entrées « 5,7 » et la sortie « 12 ».

Définition : Un algorithme associé à un problème est un procédé automatique et effectif comportant une description finie :

- des entrées
- des sorties
- des tâches élémentaires à effectuer pour obtenir la sortie attendue.

C'est donc un procédé de calcul automatique, effectif, et qui s'arrête quelles que soient les entrées données.

Propriétés : un algorithme devrait respecter les propriétés suivantes…

- Correction : toutes les solutions données sont correctes
- Complétude : l'algorithme trouve toutes les solutions
- Terminaison : il s'arrête quelle que soient les entrées

Les opérations arithmétiques peuvent être explicitées par des algorithmes, mais c'est également le cas d'autres opérations plus complexes, comme le tri de mots par ordre alphabétique, …

Algorithme versus programme

Un algorithme est indépendant des machines sur lesquelles il est exécuté (à l'exception de la parallélisation massive, dont il n'est pas question ici) et du langage dans lequel il est codé.

Un algorithme n'est pas un programme : il doit être compréhensible par l'humain, tandis que le programme doit être compréhensible par la machine. A tout programme correspond un algorithme sous-jacent, mais l'inverse n'est pas forcément vraie.

A tout algorithme est associé le problème qu'il résout, mais à l'inverse, on ne peut pas forcément associer à tout problème un algorithme pour le résoudre. Les problèmes admettant un (des) algorithme(s) pour les résoudre sont dits calculables ou décidables ; il existe des problèmes non décidables (eg : problèmes de la diagonalisation de Cantor ou de l'arrêt de la machine de Turing).

3. Traiter un problème

Pour aborder les concepts de l'algorithmique, il est nécessaire de poser et résoudre les problèmes de la façon suivante :

- Identifier les données, la problématique, y associer des structures de données et des techniques de résolution
- Transcrire le problème *via* une spécification (description) indépendante des contraintes de langage, de programmation et d'exécution
- Analyser le coût indépendamment des conditions de programmation et d'exécution (on parle de complexité).

En principe, on ne programme qu'après avoir suivi ces étapes.

a. Spécification informelle d'un problème

Il s'agit d'associer au problème un court résumé en langage vernaculaire. La description précisera (sans expliciter) les entrées, les sorties et ce que doit faire l'algorithme.

Ex : si le problème consiste à calculer la somme d'une suite de nombres, une spécification informelle pourrait être.

```
/* entrée : une liste de nombres entiers – sortie : un entier égal à leur somme */
```

On remarque que « liste de nombres entiers » et « entier » précisent des structures de données et des types.
Dans l'exemple, la sortie précise suffisamment ce que doit faire l'algorithme. Dans certains cas, il faudra ajouter une courte description en plus de celle des entrées/sorties.

b. Description algorithmique du problème

Il s'agit de donner une description de l'algorithme qui soit indépendante de tout langage de programmation. On utilise pour cela un pseudo-langage algorithmique convenu, dont la syntaxe n'est pas très stricte.

Nous utiliserons notamment :

- l'affectation (notée ← pour éviter toute confusion avec l'égalité)
- la mise en séquence
- les tests
- les boucles
- les fonctions et procédures (méthodes) avec passages de paramètres et récursivité

Pour l'exemple donné ci-dessus (faire la somme d'une suite de nombres), la description de l'algorithme pourrait prendre plusieurs formes différentes :

Dans un style impératif, en parcourant la séquence de nombres	En utilisant une liste et un procédé récursif
Algorithme pour la somme de N nombres /* entrée : N nombres entiers*/ /*sortie : un entier S égal à leur somme */ donner à S la valeur du premier entier pour k allant de 2 à N ajouter le k-ième entier à S donner S en sortie	Fonction somme (L : liste) : entier /* entrée : une liste L de nombres entiers*/ /*sortie : un entier S égal à leur somme */ Si L est vide alors rendre 0 sinon rendre prem(L) + somme(fin(L))

Si maintenant le problème consistait à trouver l'élément maximum (le plus grand) dans une liste d'entiers, il faudrait parcourir la séquence en comparant chaque élément au candidat maximum et en gardant systématiquement le meilleur.

Soit, en style impératif :

```
Algorithme du Maximum
                                        /*Connaissant en entrée une liste L d'entiers non vide*/
                          /*cet algorithme calcule la valeur maximale de la liste, qui sera placée dans maxi*/
maxi ← 0
Tant que L non vide
        si prem(L) > maxi alors maxi ← prem(L)
        L ← fin(L)
rendre maxi en sortie
```

Et en style récursif :

```
Fonction Max(L : Liste) : entier
                                        /*Connaissant en entrée une liste L non vide*/
                                        /*cette fonction rend la valeur maximale*/
si (L est vide) alors rendre 0
sinon rendre Maximum( prem(L), Max(fin(L)))
```

On suppose ici que la fonction Maximum retourne le plus grand des deux entiers obtenus en entrée : quel est son algorithme ?

Une fois l'algorithme défini et décrit, il est important de calculer son coût, qu'on appelle la complexité.

c. Evaluation de la complexité d'un algorithme

La complexité d'un algorithme est une mesure d'efficacité en durée d'exécution (quelquefois en espace) estimée en fonction de la taille des données, indépendamment du mode de codage et d'exécution, donc du langage et de la machine.

La complexité peut aussi être donnée en « nombre d'opérations » : la durée moyenne d'une opération permet ensuite de se ramener à la durée d'exécution.

Il ne s'agit généralement pas d'une valeur exacte, mais d'une approximation prenant tout son intérêt pour les données de très grande taille. On négligera donc certains détails lors du calcul, pour

s'intéresser à une tendance générale. On verra dans la suite que les améliorations ponctuelles ou locales n'ont souvent qu'une influence mineure sur la performance globale finale dès l'instant que la taille des données devient conséquente.

La complexité d'un algorithme dépend de la taille des données et de l'algorithme lui-même.

On nomme N la taille de l'entrée (la taille des données en entrée) et on évalue le nombre d'opérations élémentaires effectuées par l'algorithme (donne une estimation du temps d'exécution en fonction du matériel).

Hypothèses fortes pour le calcul de complexité

- On calcule la complexité pour N (taille des données) très grand
- le programme est séquentiel (par opposition au calcul hautement parallèle), exécuté sur une seule machine générale (par opposition au calcul distribué)
- les opérations sont effectuées les unes après les autres en mémoire centrale (le coût des transferts de données est négligeable : attention, selon l'architecture, ce n'est pas toujours vrai, notamment pour le calcul distribué)
- nous utilisons nos connaissances sur les coûts des opérations (par exemple, l'addition ayant un coût plus faible que la multiplication, il peut arriver qu'on néglige les additions)

Etapes du calcul de complexité

Etape 1 : Déterminer les opérations fondamentales selon le problème traité.
Etape 2 : Compter les opérations fondamentales ou les borner.
Etape 3 : Evaluer l'ordre de grandeur de la complexité en fonction de la taille des données et du nombre d'opérations fondamentales.

On cherche, de manière générale, à évaluer les complexités au pire et au mieux (cas extrêmes) et en moyenne (il s'agit du coût pratique, le plus difficile à calculer en général).

Exemple de calcul de complexité dans le cas de la somme d'une série de N entiers :

```
Algorithme pour la somme de N nombres
                    /* entrée : N nombres entiers*/
                    /*sortie : un entier S égal à leur somme */
donner à S la valeur du premier entier
pour k allant de 2 à N
        ajouter le k-ième entier à S
donner S en sortie
```

Pour une entrée de taille N (autrement dit pour une liste de N entiers fournis en entrée), l'algorithme effectue N affectations et N-1 additions, ce qui fait 2N-1 opérations nécessaires (on obtiendra une estimation du temps d'exécution en fonction des caractéristiques du matériel).

d. Exemple : recherche du Min et du Max dans une série d'entiers. Calcul de complexité sur 3 versions.

On veut rechercher simultanément le maximum et le minimum d'une liste donnée. Pour ce faire, nous allons rédiger trois algorithmes plus ou moins efficaces et comparer leurs complexités.

Spécification informelle :

```
Recherche_Maximum_et_Minimum = classe{
        Champs
                L = liste d'entiers
        Méthodes
                Procédure Max_Min(var max, min : entier)
                Procédure Bis_Max_Min(var max, min : entier)
                Procédure Ter_Max_Min(var max, min : entier) }
```

- **Première possibilité : balayer sur une fenêtre simple.**

On parcourt la liste en comparant chaque élément au candidat maximum et au candidat minimum. On garde toujours les meilleurs.

```
Procédure Max_Min (var max, min : entier)
                /* Entrée : L une liste avec au moins deux éléments*/
                /*Sortie : deux valeurs max et min, la plus grande et la plus petite*/
Si prem(L) < second(L) alors                    /* initialisation des candidats*/
        max← second(L)
        min← prem(L)
sinon
        min← second(L)
        max← prem(L)
L← fin(fin(L))
tant que L non vide faire    /*parcours de la liste avec comparaison*/
        si prem(L) > max alors max← prem(L)
        si prem(L) < min alors min← prem(L)
        L← fin(L)           /*et mise à jour des candidats*/
```

Nombre de comparaisons : 1+2*(N-2) = 2*N-3
Nombre d'affectations : 2+1+2(N-2) = 2*N-1

- **<u>Seconde possibilité (amélioration) : éviter de comparer un élément successivement au max et au min.</u>**

On sait que min≤max, donc il faut ajouter un sinon bien placé…

```
Procédure Bis_Max_Min (var max, min : entier)
                    /* Entrée : L une liste avec au moins deux éléments*/
                    /*Sortie : deux valeurs max et min, la plus grande et la plus petite*/
Si prem(L) < second(L) alors                    /* initialisation des candidats*/
        max← second(L)
        min← prem(L)
sinon
        min← second(L)
        max← prem(L)
L← fin(fin(L))
tant que L non vide faire    /*parcours de la liste avec comparaison*/
        si prem(L) > max alors max← prem(L)
        sinon si prem(L) < min alors min← prem(L)
        L← fin(L)            /*et mise à jour des candidats*/
```

On obtient le même nombre d'affectations : seule une des deux affectations du si…alors…sinon se fait dans les deux cas
Le nombre de comparaisons obtenu varie :

- dans le meilleur cas, N-1, pour une liste déjà triée en ordre croissant, on ne passe pas dans le sinon
- dans le pire, 2*N-3, quand le maximum est au début

- **Troisième possibilité : balayer avec une fenêtre double**

On utilise une fenêtre de taille 2, on compare les deux éléments entre eux, puis le plus petit à *min* et le plus grand à *max*. On met les candidats à jour si nécessaire.

```
Procédure Ter_Max_Min (var max, min : entier)
                /* Entrée : une liste d'au moins deux éléments*/
                /*Sortie : deux valeurs max et min, la plus grande et la plus petite*/

Si prem(L) < second(L) alors                    /* initialisation des candidats*/
        max← second(L)
        min← prem(L)
sinon   min← second(L)
        max← prem(L)
L←  fin(fin(L))

Tant que ( fin(L) non vide et fin(fin(L)) )[2] non vide faire        /*parcours avec comparaison*/
        Si prem(L) < second(L) alors
                si prem(L)< min alors min←prem(L)
                si second(L) > max alors max←second(L)
        sinon
                si second(L) < min alors min← second(L)
                si prem(L) > max alors max← prem(L)
                                                        /*mise à jour des candidats*/
        L←fin(fin(L))
/*sortie du tant que */

/*rattrapage éventuel de la dernière case en taille impaire*/
Si fin(L) non vide alors
        si prem(L) < min alors min← prem(L)
        sinon
                si prem(L) > max alors max← prem(L)
/*en fin de procédure, les deux valeurs cherchées sont dans min et max*/
```

On tombe à environ (N-2)/2 passages dans la boucle, avec 3 comparaisons par passage, soit une complexité en nombre de comparaisons d'éléments d'environ 3*N/2. Gain : N/2.

En fait, dans les trois cas, la complexité est dite linéaire : c'est une fonction linéaire de la taille des données.

e. Calcul de complexité : détail des étapes

Par convention, lorsqu'on calcule la complexité d'un algorithme, on émet l'hypothèse que les données sont en très grand nombre (autrement dit que la taille des données est très grande), ce qui fait qu'on retiendra seulement l'équivalent à l'infini de la fonction de complexité.

La complexité **en temps** d'un algorithme est donnée par le nombre d'opérations fondamentales nécessaires à l'exécution de ce dernier en fonction de N, la taille des données. Le calcul se décompose en trois étapes.

[2] Evaluation paresseuse : si la première expression est fausse, pas d'évaluation de la seconde.

Etape 1 : déterminer les opérations fondamentales

Avant de compter les opérations fondamentales, il faut déterminer quelles sont les opérations de l'algorithme qui seront effectivement considérées comme fondamentales : elles peuvent être différentes en fonction du problème traité (par exemple, pour la recherche d'un élément en mémoire centrale, les comparaisons sont des opérations fondamentales, tandis que pour la recherche d'un élément en mémoire périphérique, le nombre d'accès mémoire sera important).

NB : Il est également possible de choisir plusieurs opérations comme étant fondamentales et de les pondérer.

Etape 2 : Comptage ou bornage des opérations fondamentales

On essaie de compter les opérations ou au moins d'en donner les bornes inférieures et supérieures, ou des valeurs moyennes. On utilise pour cela des règles simples en fonction des primitives de langage algorithmique considérées :

- Pour les séquences d'instructions, les coûts s'ajoutent
  $coût(I_1 \dots I_n) = coût(I_1) + \dots + coût(I_n)$
- Pour les boucles, les coûts s'ajoutent (attention à tant que… faire et répéter… jusqu'à, pour lesquelles le nombre de passages est compliqué à évaluer)
  *coût(*pour k allant de a à b faire I(k)*) = coût(I(a))+ coût(I(a+1))… +coût(I(b))*
  *coût(*tant que C(p) faire (I(p) ; p←p+1)*) = coût(C(p_1)) + coût(I(p_1))+ … coût(C(p_n)) + coût(I(p_n))*
- Pour les branchements conditionnels, on peut calculer deux évaluations en fonction du bloc d'instructions qui s'exécute :
  *coûtMax(*si C alors I_1 sinon I_2*) = coût(C) + Max(coût(I_1), coût(I_2))*
  *coûtMin (*si C alors I_1 sinon I_2*) = coût(C) + Min(coût(I_1), coût(I_2))*
- Pour les appels de méthodes
  - s'il n'y a pas de récursivité, on applique les règles ci-dessus pour calculer la complexité des sous-méthodes et méthodes.
  - En cas de récursivité, il faut utiliser des procédés mathématiques plus complexes, tels que la résolution d'équations de récurrence.

Etape 3 Calcul de complexité

On déduit du comptage des opérations élémentaires des fonctions de N (la taille des données) pouvant être de plusieurs natures. Notamment, complexité au pire, en moyenne, ou au mieux, selon les cas considérés. Dans tous les cas, on considère que N est très grand.

- ComplMin(A, données) = Min {coût(A, d) / d parcourant toutes les données}
- ComplMax(A, données) = Max {coût(A, d) / d parcourant toutes les données}
- ComplMoy(A, données) = Somme {proba(d)*coût(A, d) / d parcourant les données}

Où coût(A,d) est le coût (la complexité) de l'algorithme A en fonction de la configuration des données d et proba(d) est la probabilité que les données soient dans cette configuration.

On a toujours : ComplMin ≤ ComplMoy ≤ ComplMax.

Les complexités dans le pire et le meilleur cas permettent de borner la durée de traitement du problème, tandis que la complexité moyenne, plus difficile à calculer, donne une idée du coût en pratique. Bien souvent, on cherchera à borner la complexité plutôt qu'à la calculer précisément.

4. Complexité et ordre de grandeur

On s'intéresse à l'ordre de grandeur de la complexité plus qu'à sa valeur exacte, qui est souvent difficile à estimer avec précision.
Imaginons des algorithmes dont on aurait calculé les complexités en fonction de la taille N de l'entrée, fonctions qui seraient respectivement N-1, 2*N –3, N^2+N –5, N^3+ N.
Calculons les valeurs de ces fonctions pour quelques valeurs de N.

Taille de la donnée	N	10	100	1000	10 000
Complexité	N-1	9	99	999	9 999
Complexité	2*N-3	17	197	1997	19 997
Complexité	N^2 +N –5	105	10 095	1 000 995	100 009 995
Complexité	N^3+N	1010	1 000 100	1 000 000 100	1 000 000 010 000

On peut voir ici que la taille des données a beaucoup d'importance dans la valeur effective que prend la complexité. Si N est petit, les valeurs liées aux complexités sont différentes mais faibles. Si N est grand, la différence relative peut devenir très importante.
On voit par exemple que N^2+N –5 sera relativement proche de N^2 alors que l'écart entre N et N^2 sera clairement important.

On dit que les deux premiers algorithmes sont « en N », et le suivant « en N^2 » ; le dernier est « en N cube ».

Si la fonction complexité est un polynôme de la variable N, taille de l'entrée, et de degré p, elle sera dite « en N^p », et ce quel que soit le coefficient du terme de plus haut degré.

Conventions et règles pour l'expression de la complexité :

- Par convention (et hypothèse), lors du calcul de la complexité d'un algorithme, on se place dans le cas où les données sont en très grand nombre et on ne retient qu'une sorte d'équivalent à l'infini de la fonction complexité.
- Les fonctions f et g sont dites équivalentes en l'infini si f(n)/g(n) admet une limite égale à 1 quand n tend vers l'infini. On note f≈g.
- On écrit f=O(g) s'il existe une constante K indépendante de n telle que pour tout n grand, on ait f(n)<K*g(n)
- On écrit f=Θ(g) si f=O(g) et g=O(f)
- On écrit f=o(g) si le rapport f(n)/g(n) admet une limite égale à 0 quand n tend vers l'infini.

Quelques formules utiles aux calculs :

- $1 + 2 + \ldots + n = n*(n+1)/2$
- $1^\alpha + 2^\alpha + \ldots + n^\alpha = \Theta(n^{\alpha+1})$
- $1/1 + 1/2 + \ldots + 1/n \approx \log(n)$
- $2^0 + 2^1 + 2^2 + \ldots + 2^\pi = 2^{\pi+1} - 1$
- si $a \neq 1$, $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1) / (a - 1)$
- $\log(n!) \approx \Theta(n*\log(n))$
- $a_n x^n + \ldots + a_1 x^1 + a_0 x^0 = O(x^n) = \Theta(x^n)$
- $\forall n,\ a_n x^n = o(e^n)$

Nomenclature :

Un algorithme A est dit de complexité :

-constante,	si complexité(A) = $\Theta(1)$	
-logarithmique,	si complexité(A) = $\Theta(\log(n))$	« en log(n) »
-linéaire,	si complexité(A) = $\Theta(n)$	« en n »
-quadratique,	si complexité(A) = $\Theta(n^2)$	« en n^2 »
-cubique,	si complexité(A) = $\Theta(n^3)$	« en n^3 »
-polynomial,	si complexité(A) = $\Theta(n^\alpha)$ avec $\alpha >= 0$.	« en n^α »
-exponentiel,	si complexité(A) = $\Theta(e^n)$	« en e^n »)

S'il fallait encore vous convaincre que la taille des données est importante et qu'on voit nettement les différences de complexités dès qu'on en tient compte, voici quelques chiffres…

N	4	16	256	65 536	4 294 967 296	$1{,}8447 * 10^{19}$
log(N)	2	4	8	16	32	1024
N*log(N)	8	64	2048	1048576	$1{,}37* 10^{11}$	$1{,}89* 10^{22}$
N^2	16	256	65 536	4 294 967 296	$\sim 1{,}8447 * 10^{19}$	$3{,}4* 10^{38}$

Et ci-dessous, voici une estimation du temps d'exécution d'algorithmes selon leur complexité et la taille des données, pour un ordinateur pouvant effectuer 10^6 opérations/seconde.

Complexité taille	cste	Log(N)	N	N*log(N)	N^2	N^3	2^N
$N = 10^2$	1 µ s	6,6 µ s	0,1 m s	0,66 ms	10 ms	1 s	4.10^7 milliards d'années
$N = 10^3$	1 µ s	9,9µ s	1 m s	9,9 s	1 s	16,6 mn	Infini physique
$N = 10^4$	1 µ s	13,3µ s	10 m s	0,13 s	1,5 m	11,5 j	Infini physique
$N = 10^5$	1 µ s	16,6µ s	0,1 s	1,64 s	2,7 h	31,7 années	Infini physique
$N = 10^6$	1 µ s	19,9µ s	1 s	19,9 s	11,5 j	31,7 milliers d'années	Infini physique

Notion de faisabilité et de gain de complexité

On peut voir que gagner un facteur logarithmique, c'est-à-dire trouver pour un problème un algorithme en log(N) plutôt qu'un algorithme en N, peut se révéler très vite particulièrement économique.

On considère qu'un algorithme « en N » est faisable, ce qui est déjà plus délicat pour un algorithme en N^2 (carré ou quadratique), alors que « du N^3 » est souvent rédhibitoire.
Les problèmes réellement exponentiels sont « infaisables » dans le cas général mais posent de passionnantes questions de classification théorique et de restrictions à des cas abordables.

Considérations pratiques et complexités typiques

Les algorithmes de recherche d'un élément dans une liste de taille N sont en général de complexité « en N », ou en log(N) si la liste est triée.
Les algorithmes naïfs de tri d'une liste de taille N sont en général de complexité « en N^2 » mais il en existe en N*log(N).
Les algorithmes naïfs sur les graphes sont très souvent au moins « en N^3 ».
Une structuration en arbres permet dans certains cas de passer d'une complexité « en N » à une complexité « en log(N) ».

5. Diminuer la complexité d'un algorithme : un exemple

Dans cette partie, nous allons illustrer par un exemple une stratégie classique permettant de passer d'un algorithme linéaire (en N) à un algorithme logarithmique (en log(N)).
Le but de l'algorithme choisi est d'extraire les deux plus grands éléments d'une liste d'entiers.

Convention : la liste contient au moins deux éléments, et si le plus grand élément apparaît deux fois ou plus, on fournit comme résultat deux fois cet élément maximum.

a. Recherche des deux plus grands éléments : complexité en 2*N

```
Recherche_Premier_Second = classe {
        Champs
            L = Liste d'entiers
        Méthodes
            Procédure Premier_Second(var premier, second : entier)
            Procédure Tournoi(var premier, second : entier) }
```

- **Approche naïve**

On peut reprendre ce qui a été vu pour la recherche du max et du min dans une liste d'entiers et parcourir la liste en comparant l'élément courant à deux candidats (premier et second). On garde alors les meilleurs à chaque étape.

```
Algorithme_Premier_Second
                /* entrée : une liste L avec au moins deux éléments*/
                /*Sortie : deux entiers premier, second, qui sont les deux plus grands éléments de la liste */

si prem(L) < prem(fin(L)) alors                  /* initialisation des candidats*/
                premier ← prem(fin(L))
                second ← prem(L)
sinon           second ← prem(fin(L))
                premier ← prem(L)
L ← fin(fin(L))                                  /* on saute les deux premiers éléments traités*/
Tant que (L non vide) faire
        si prem(L) > premier alors
                second ← premier
                premier ← prem(L)
        sinon
                si prem(L) > second alors
                        second ← prem(L)         /*mise à jour des candidats*/
        L ← fin(L)                               /* on avance dans la liste*/
```

Nb : fin(L) donne accès aux éléments suivant le premier élément (*i.e.* à la liste privée de sa tête)

Complexité en nombre de comparaisons entre :
- N-1 dans le meilleur des cas (liste triée par ordre décroissant)
- 2*N -3 dans le pire des cas (liste triée par ordre croissant)

Pour une liste non triée, la complexité est à calculer en fonction des probabilités.

Cette solution n'est pas optimale, nous allons voir dans la suite comment en améliorer les performances.

b. Recherche des deux plus grands éléments : complexité en N+log2(N)

Une stratégie pour diminuer la complexité d'un algorithme consiste à diminuer la taille de la collection de données traitée à chaque « tour » de l'algorithme. C'est ce que l'on fait, par exemple, lors d'une recherche dichotomique.

Nous allons utiliser la stratégie du tournoi, une stratégie classique permettant d'illustrer le passage d'un algorithme linéaire à un algorithme logarithmique.

Principe de l'algorithme du tournoi : des joueurs s'affrontent en tournoi. Chacun porte un numéro différent des autres. Dans un combat à deux joueurs, celui qui porte le plus grand numéro gagne.

Déroulement (on suppose qu'il y a au moins 2 joueurs) :
Lors d'une première phase, tous jouent deux contre deux (sauf si le nombre de joueurs est impair, le dernier engagé « gagne » contre lui-même).
Les gagnants de la première phase s'affrontent de la même façon en une seconde phase.
Ainsi de suite jusqu'à ce qu'il ne reste qu'un candidat.
Le gagnant est le dernier en lice, et le second est celui qui n'a perdu qu'une fois, contre le gagnant.
Le problème revient donc à trouver le second.

L'algorithme se déroule donc en deux phases :

1. Construction de l'arbre du tournoi, dont la racine sera le gagnant.
2. Recherche du second en parcourant une branche de l'arbre du tournoi.

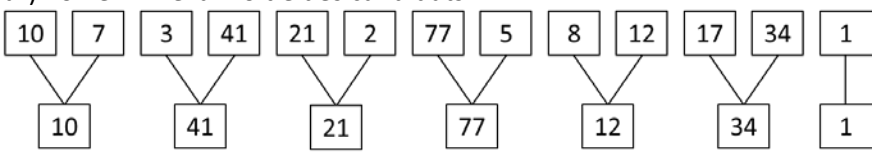
Tournoi, partie 1 : construction de l'arbre.

Le jeu ressemble à un tournoi sportif dans lequel les joueurs s'opposent deux à deux. Seuls les gagnants du tour i restent en lice pour le tour i+1.

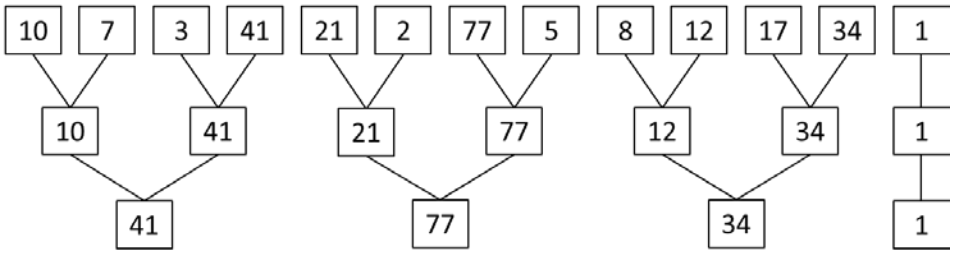
Prenons un exemple pour des joueurs portant les numéros 10,7, 3, 41, 21, 2, 77, 5, 8, 12, 17, 34, 1.

T0 : tout le monde en lice

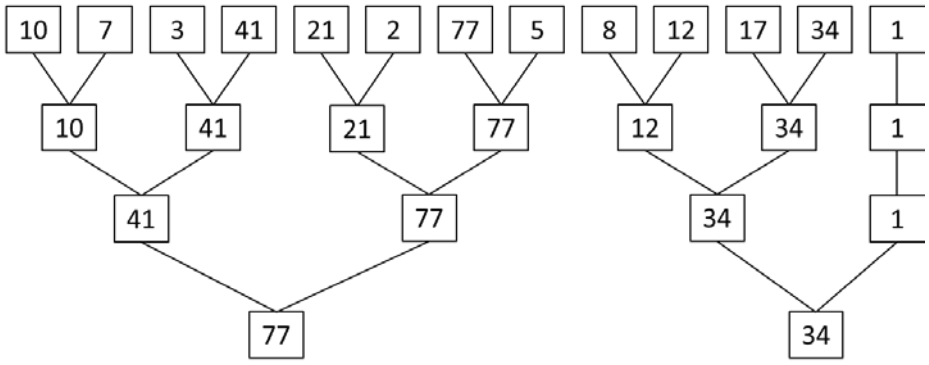
T1 (premier tour) : on élimine la moitié des candidats



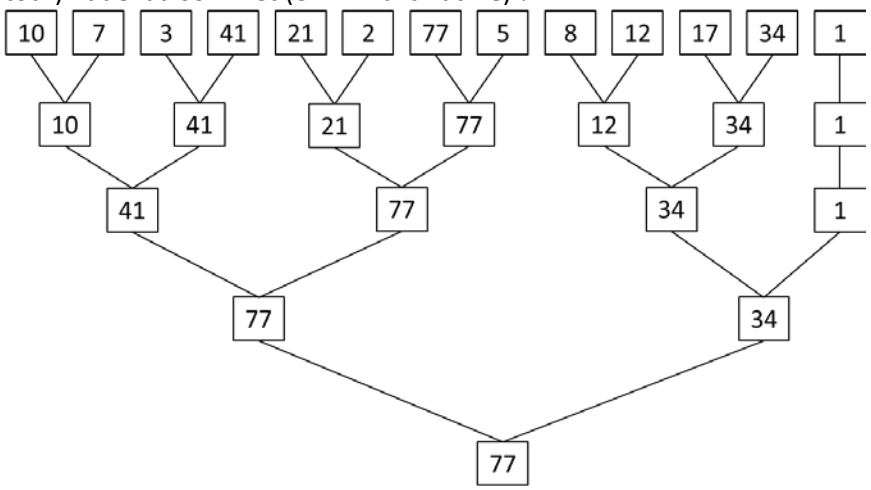
T2 (second tour) : on élimine encore la moitié des candidats (et le dossard 1 s'en tire très bien).



T3 (troisième tour) : encore et encore (et le dossard 1 va enfin devoir s'y coller)…



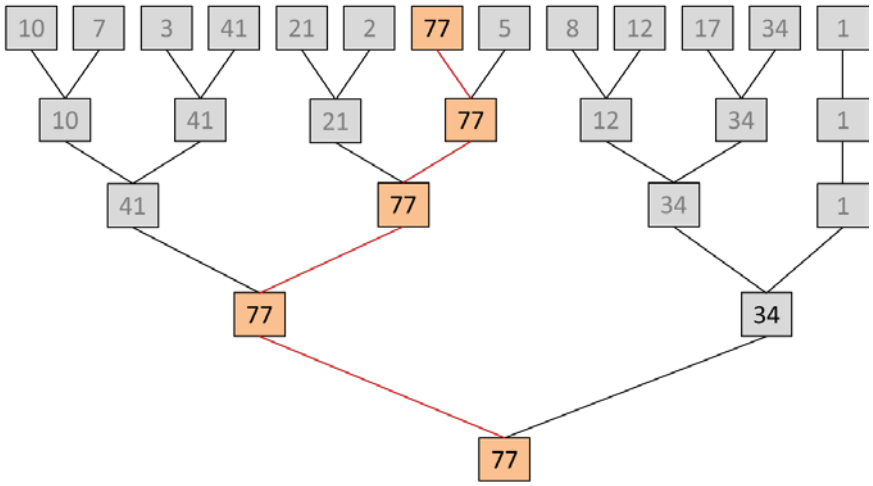
T4 (quatrième tour) : duel au sommet (enfin… à la racine) !



A la fin du tournoi, la racine de l'arbre contient le vainqueur.
Nombre de comparaisons pour obtenir le plus grand élément : N (autant que d'éléments dans la collection)

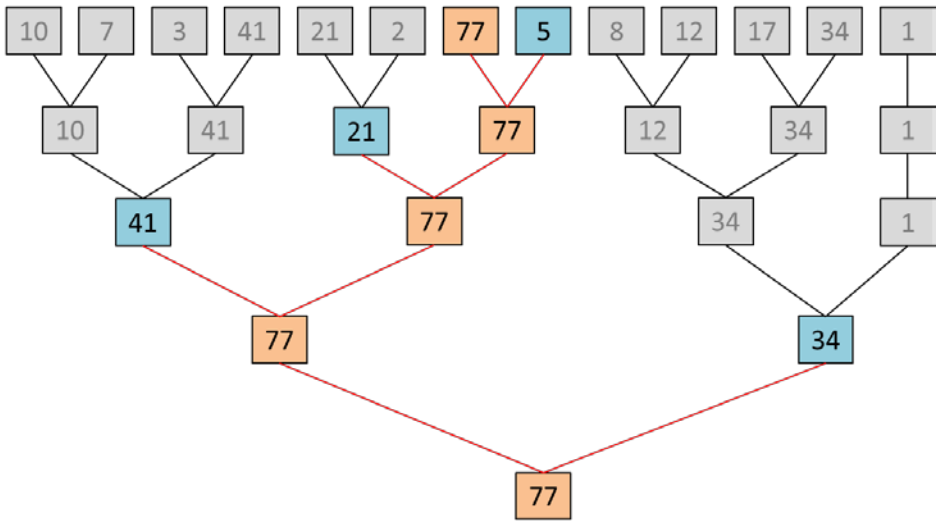
<u>Tournoi, partie 2 : recherche du second sur une branche de l'arbre.</u>

L'arbre qu'on obtient a une branche dont tous les nœuds valent 77.



Le second ne peut avoir perdu qu'une seule fois, contre le maximum… mais pas nécessairement au dernier tour.

On va donc rechercher le second parmi les éléments ayant affronté le maximum, c'est-à-dire le long de la branche qui porte le plus grand élément.



La longueur de cette branche est log(N) (cf. plus loin).
La recherche du second sur la branche est linéaire en la taille de la branche. La taille de la branche est en log(N), la recherche du 2^{nd} élément est donc linéaire en log(N).

On est donc passé d'une complexité en 2N à une complexité en N+log(N).

Complexité de la construction de l'arbre du tournoi.

Les étapes sont :

1. Partir d'une liste de joueurs qui formera les feuilles de l'arbre
2. Phase : remplacer une liste à N éléments par une liste à [N/2] éléments en remplaçant 2 par 2 les joueurs par un arbre de racine le gagnant, et de fils les deux joueurs.
3. Répéter « Phase » jusqu'à ce qu'il ne reste qu'un seul élément dans la liste

A la sortie, on a un arbre tournoi dont le vainqueur est la racine.

Calcul de la complexité

Si elle part d'une liste à 2^z éléments, une phase exécute 2^{z-1} fois le paquet d'opérations :

- 1 comparaison de deux numéros
- 1 construction de racine
- 1 construction de 2 pointeurs vers les fils

Si l'on choisit la comparaison comme opération principale, la complexité de cette phase est donc 2^{z-1}
Si la liste comporte $N=2^y$ éléments, la somme des complexités des phases successives peut s'écrire :
$Compl(N) = 2^{y-1} + 2^{y-2} + \dots + 2^0 = N-1$
Si la liste de départ comporte un nombre N d'éléments compris entre 2^{y-1} et 2^y éléments, on peut montrer par récurrence sur N que cette complexité reste égale à N-1.

Complexité de la recherche du gagnant et du second dans l'arbre tournoi.

Les étapes sont :

1. Le gagnant est le numéro porté par la racine de l'arbre.
2. Le second est le max des numéros des fils du gagnant (non gagnants) sur la branche de l'arbre tournoi portant le gagnant.

Il nous reste donc à montrer que la branche qu'on parcourt est de longueur $\log_2(N)$ car la recherche du maximum est linéaire en la taille des données.

Preuve que la hauteur de l'arbre est en $\log_2(N)$.

La recherche de l'élément sur la branche de l'arbre est similaire à celle vue précédemment, donc de complexité linéaire en la taille des données. Ce qui a changé ici, c'est justement la taille de la liste d'entiers sur laquelle on recherche : il s'agit non plus de toutes les données, mais de celles qui se trouvent sur une seule branche de l'arbre-tournoi.
On a donc une complexité linéaire en la hauteur de l'arbre-tournoi.

> Définition : la hauteur de l'arbre est la taille de sa plus grande branche.
> Pour l'arbre que nous avons pris en exemple, la hauteur est de 4.

Notons h(N) la hauteur de l'arbre à N feuilles construit lors d'un tournoi à N joueurs.
Pour les premiers calculs, on a :
h(2) = 1
h(3) = h(4) = 2
h(5) = h(6) = h(7) = h(8) = 3
h(9) = 4
par récurrence, on voit que $h(2^y) = y$
On a un arbre du type « arbre binaire équilibré complet » pour lequel la hauteur est $h(N)=\log_2(N)$.

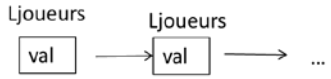
⇨ On a donc une complexité de la recherche du deuxième plus grand élément en $\log_2(N)$, ce qui nous amène à une complexité totale en $N+ \log_2(N)$ au lieu de 2*N.

- **Algorithme du tournoi pour la recherche des deux plus grands éléments d'une liste**

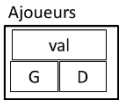
Les éléments seront initialement contenus dans une structure de données de type liste.

Structures de données

Ljoueurs : liste d'entiers avec un champ .val (le n° porté par le joueur).



Ajoueurs : Arbre binaire avec un champ .val pour le n°, deux champs .g et .d. Chaque nœud est soit une feuille, soit aura deux fils Ajoueur, le nœud lui-même portant le plus grand des deux numéros de ses fils.



LAjoueurs : liste d'arbres Ajoueurs.

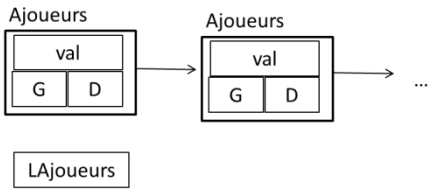


Schéma général de la construction de l'arbre-tournoi

- Partir d'une liste Ljoueurs et la transformer en une liste LAjoueurs formée des feuilles de l'arbre-tournoi.
- Phase 1 : remplacer la liste LAjoueurs à N éléments par une liste à [N/2] éléments en remplaçant deux par deux les joueurs par l'arbre composé du gagnant des deux nombres à la racine et de ses deux fils (les joueurs qui se sont affrontés).
- Répéter la phase 1 jusqu'à ce qu'il ne reste qu'un seul élément dans la liste : ce sera l'arbre-tournoi, dont la racine contiendra le vainqueur.

Transcription en langage algorithmique

```
Fonction faitA (x : entier) : Ajoueurs                    /*chaque joueur devient un arbre*/
        initialiser un nouvel arbre A, avec
        A.val ← x ; A.g ← vide ; A.d ← vide ;


Fonction construire (L : Ljoueurs) : LAjoueurs            /*la liste de joueurs se transforme en liste d'arbres*/
        si (L = vide) alors rendre Vide
        sinon rendre cons (faitA(L.val), construire(fin(L)))      /*cons(elem, L) ajoute l'élément elem à la liste L*/


Fonction gagnant(A, B : Ajoueur) : Ajoueur                /*construction de l'arbre gagnant de deux joueurs*/
                                                          /*A et B sont non vides*/
initialiser un nouvel arbre C, de type Ajoueur, avec
        C.val ← max(A.val, B.val)                         /* le gagnant dans val */
        C.g ← A                    /*le fils gauche et le fils droit du nouveau nœud */
        C.d ← B


Fonction phase (M : LAjoueurs) : LAjoueur                         /*une phase du tournoi*/
si (M = vide) alors
        rendre Vide
sinon                                                     /*la taille de M sera divisée par 2*/
        si (fin(M) = vide) alors rendre M
        sinon     rendre cons( gagnant(M.val, fin(M).val), phase(fin(fin(M))))


Fonction tournoi (M : Lajoueur) : Ajoueur                 /*fournit l'arbre du tournoi*/
                                                          /*M est non vide*/
si fin(M) = vide alors
        rendre M.val
sinon rendre tournoi(phase(M))
```

Schéma général de la recherche du gagnant et du second

Le gagnant est le numéro porté par la racine de l'arbre tournoi, soit A.val
Le second est le max des numéros des fils du gagnant (non gagnants) sur la branche de l'arbre tournoi qui porte le gagnant.

Transcription en langage algorithmique

```
Fonction premier (A : Ajoueur) : entier
        Renvoyer A.val

Fonction second (A : Ajoueur) : entier                    /*A est l'arbre du tournoi créé précédemment*/
si feuille(A) alors
        rendre 0                          /*convention : 0 est plus petit que tous les autres numéros*/
sinon
        si A.g.val = A.val alors
                rendre max(A.d.val, second(A.g))
        sinon rendre max(A.g.val, second(A.d))
```

On parcourt la branche de façon récursive.