

Cours d'Algorithmique

Site: Espadon
Cours: ALGORITHMIQUE ET PROGRAMMATION NIVEAU 2
Livre: Cours d'Algorithmique
Imprimé par: Thomas Stegen
Date: mercredi 23 mars 2016, 16:20

Table des matières

1 Introduction

2 Algos de base

3 Les listes

4 Les listes mono-dimension (suite)

5 Recherches dans une liste

6 Complexité

7 Les tris simples

1 Introduction

De quoi s'agit-il ?

Définitions

Il n'est pas aisé de donner une bonne définition de ce qu'est un algorithme ou l'algorithmique mais voici deux petits essais :

- **Algorithme** : c'est une procédure de résolution de problème énoncée sous la forme d'une série d'opérations à effectuer. L'**implémentation** de l'algorithme consiste en l'écriture de ces opérations dans un langage de programmation ;
- **Algorithmique** : néologisme désignant la science des algorithmes; en effet **algorithmique** est absent des dictionnaires où seul l'adjectif algorithmique est défini pour ce qui utilise ou se réfère aux algorithmes, par exemple une méthode algorithmique.

La rédaction et la vérification d'un algorithme sont **des étapes fondamentales indispensables** à l'écriture d'un programme correct et sont quasi **indépendantes du langage de programmation** utilisé pour l'implémentation.

En avant la musique !

Si l'on compare la programmation informatique à la musique, que les mélomanes me pardonnent, l'algorithme serait alors la partition musicale et le langage de programmation l'instrument. Cette métaphore est intéressante à plus d'un titre, en effet :

1. Au risque de vexer les interprètes, il est bien plus difficile d'écrire de la musique (écrire un algorithme) que de la jouer (programmer un algorithme) ;
2. Une même partition (algorithme) peut être jouée sur différents instruments (langage de programmation) ;
3. On peut savoir jouer d'un instrument (programmer) sans être capable de composer un morceau (analyser le problème et écrire l'algorithme) ;
4. Une partition (algorithme) peut être plus ou moins adaptée à un instrument (langage de programmation) ;

Démarche à suivre

Les étapes fondamentales de l'écriture d'un algorithme sont les suivantes :

1. Avoir parfaitement compris le sujet
2. Être capable de faire ce que l'on vous demande avec un papier et un crayon
3. Isoler les étapes fondamentales
4. Écrire l'algorithme en pseudo-code
5. Faire tourner l'algorithme écrit *à la main*
6. Coder l'algorithme dans le langage de programmation souhaité

Deux petits exemples

Exemple 1 : nombres divisibles par 3

On souhaite afficher les nombres divisibles par 3 inférieurs à 100

Le problème est simple à condition toutefois de savoir ce qu'est un nombre divisible par un autre (ou un multiple) et de préciser si l'on souhaite les nombres strictement inférieurs à 100 ou inférieurs ou égal à 100. Autres questions : veut-on les seuls nombres positifs ? uniquement les entiers naturels ?

La question aurait donc tout intérêt à être précisée de la manière suivante :

On souhaite afficher les nombres **entiers** divisibles par 3 et **strictement** inférieurs à 100

Savons nous résoudre le problème à la main ? OUI.

3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 75 78 81 84 87 90 93 96 99

Première solution : "on ajoute 3"

1. On commence par l'entier 3
2. on affiche l'entier
3. à l'entier que l'on vient d'afficher, on ajoute 3
4. Répétition des 2 opérations précédentes jusqu'à ce que l'entier trouvé soit supérieur ou égal à 100.

Seconde solution : "on récite notre table de 3"

1. On commence par l'entier "1 fois 3"
2. on affiche l'entier
3. on passe à "2 fois 3" puis "3 fois 3", "4 fois 3", "5 fois 3", ...
4. on continue la récitation de notre table jusque "33 fois 3" car 33 est le quotient de la division entière de 100 par 3.

Troisième solution : "on teste tous les entiers entre 1 et 99 pour savoir si ils sont divisibles par 3"

1. On commence par l'entier 1
2. si l'entier est divisible par 3 on l'affiche
3. on passe à l'entier suivant
4. Répétition des 2 opérations précédentes jusqu'à 99

Il est remarquable de noter que cette dernière solution est très souvent proposée "spontanément" pour résoudre ce problème alors qu'il ne viendrait à l'idée de personne de procéder de cette manière "à la main". En effet, cette méthode nécessite un très grand nombre de divisions entières qui peuvent très simplement être évitées en adoptant l'une des deux premières solutions.

Nous reviendrons plus tard sur l'écriture de ces algorithmes en pseudo-code, nous retiendrons pour le moment qu'**il existe presque toujours plusieurs algorithmes pour résoudre un problème donné.**

Exemple 2 : petit jeu de devinette

Pierre et Teva veulent passer le temps en jouant à un petit jeu de devinette. Pierre pense (ou écrit) un nombre compris entre 0 et 100 et Teva doit trouver le nombre en un maximum de 5 essais. A chaque essai, la réponse est "gagné", "plus petit" ou "plus grand".

Le problème est simple et semble ne pas poser de problème de compréhension majeur. Toutefois, en regardant de plus près on pourrait se demander si les valeurs 0 et 100 sont des valeurs possibles. Après avoir répondu à cette question, il est facile de jouer à ce jeu avec un papier et un crayon.

Décomposons les opérations élémentaires nécessaires et écrivons l'algorithme "en langage de tous les jours" (le pseudo-code sera abordé dans la section suivante).

1. Pierre pense à un nombre
2. Teva propose une valeur
3. si la valeur est la bonne Pierre répond "gagné" et le jeu est terminé
si la réponse n'est pas la bonne Pierre répond "plus petit" ou "plus grand"
4. Répétition des 2 opérations précédentes jusqu'à ce que Teva ait gagné ou que le nombre maximum d'essais ait été atteint.

Pseudo-code

Dans la totalité de ce cours, nous utiliserons un pseudo-code qui va nous permettre d'écrire l'ensemble de nos algorithmes. Cet ensemble est principalement constitué des familles d'instructions suivantes :

- commentaires
- variables et affectations
- boucles
- tests
- entrées/sorties
- valeur de retour
- "descriptions verbales"

Ce chapitre du cours ne fait que poser les bases du pseudo-code ou pseudo-langage, les sections suivantes illustrerons abondamment ce pseudo-code sur de nombreux exemples.

Définitions

Commentaires

A tout seigneur, tout honneur. Les commentaires sont trop souvent ignorés des analystes et programmeurs débutants alors qu'ils sont **indispensables** à la rédaction claire de tout algorithme ou programme. Sans tomber dans l'excès consistant à commenter la moindre affectation, il est indispensable de prendre cette excellente habitude consistant à commenter un minimum.

Rien de très formel pour les commentaires dans ce cours d'algorithmique, nous utiliserons la notation classique du C et d'autres langage consistant à mettre les commentaires entre `/*` et `*/`

Variables et affectations

Variables

Une variable est un identificateur quelconque de type chaîne de caractère. Il est fortement conseillé de donner des noms explicites à vos variables. Par exemple, il est bien plus "parlant" d'appeler une variable "somme" ou "a_deviner" ou encore "resultat" plutôt que a, b, i ou k. La relecture et la compréhension de vos algorithmes et programmes par une autre personne en sera grandement facilité. Seule exception à cette règle de bonne programmation : les variables simples de boucle peuvent porter le doux nom de "i", "j" ou "k" même si il serait plus adapté de leur donner des noms plus explicites.

Vous éviterez également de former des noms de variables avec des accents, des blancs, des indices, des caractères autres que alpha-numérique et du signe `_` (souligné).

Affectations

Commentaires

Variables et affectations

```
variable <== valeur
```

```
adeviner <== 25
resultat <== resultat +1
som <== som - i
```

Une variable est un identificateur quelconque de type chaîne de caractère. Il est fortement conseillé de donner des noms explicites à vos variables.

Il s'agit de mettre une valeur "dans le ventre" d'une variable. Cette affectation (sorte de flèche vers la gauche) est la base de la programmation et donc de l'algorithmique.

Boucles

Il s'agit de la structure de programmation la plus employée, il existe trois grands types de boucles que nous allons passer en revue dans cette section.

Boucle "Répéter pour ..."

Boucle Répéter pour ...

```
Répéter pour var allant de début à fin <par
pas de inc>
  Bloc d'instructions
fin_répéter
```

```
Répéter pour i allant de 1 à 100
  Bloc d'instructions
fin_répéter

Répéter pour j allant de 100 à 1 par pas
de -1
  Bloc d'instructions
fin_répéter
```

Attention ! <par pas de inc> est une partie optionnelle, en son absence l'incrément est de 1 !

Cette boucle est généralement utilisée lorsque l'on connaît, avant le début de la boucle, le nombre d'itérations.

Boucle "répéter ... jusqu'à"

Boucle Répéter ... jusqu'à

```
<initialisation de variables>
Répéter
  Bloc d'instructions
  <incrément éventuelle>
jusqu'à (test d'arrêt)
```

```
i <== 1
Répéter
  Bloc d'instructions
  i <== i + 1
jusqu'à (i > 100)
```

```

j <== 100
Répéter
    Bloc d'instructions
    j <== j - 1
jusque (j <= 0)

Répéter
    Bloc d'instructions
    Lire (réponse)
jusque (réponse différente de "Oui")

```

Attention ! <initialisation de variables> et <incrémentement éventuelle> sont des parties optionnelles uniquement requises pour "imiter" une boucle "Répéter pour ..." !

Boucle "Répéter ... tant que"

Boucle Répéter ... tant_que

<pre> <initialisation de variables> Répéter Bloc d'instructions <incrémentement éventuelle> tant_que (test de continuation) </pre>	<pre> i <== 1 Répéter Bloc d'instructions i <== i + 1 tant_que (i <= 100) j <== 100 Répéter Bloc d'instructions j <== j - 1 tant_que (j > 0) Répéter Bloc d'instructions Lire (réponse) tant_que (réponse est "Oui") </pre>
--	---

Attention ! <initialisation de variables> et <incrémentement éventuelle> sont des parties optionnelles uniquement requises pour "imiter" une boucle "Répéter pour ..." !

Remarque : Certains langages ne possèdent pas les deux derniers types de boucles mais seulement l'une des deux boucles. Par exemple, la boucle Répéter ... jusqu'à n'existe pas en langage C.

Boucle "Tant que ... répéter"

Boucle Tant_que répéter ...

<initialisation de variables>

Tant_que (test de continuation) répéter

Bloc d'instructions

<incrémenter éventuelle>

fin_tant_que

i <= 1

tant_que (i <= 100) répéter

Bloc d'instructions

i <= i + 1

fin_tant_que

Lire (réponse)

Tant_que (réponse est "oui") répéter

Bloc d'instructions

Lire (réponse)

fin_tant_que

Attention ! <initialisation de variables> et <incrémenter éventuelle> sont des parties optionnelles uniquement requises pour "imiter" une boucle "Répéter pour ..." !

La différence fondamentale entre cette dernière boucle et les deux précédentes est que cette dernière **peut ne jamais être exécutée**. En effet, le test se faisant avant le passage dans le corps de la boucle, si ce test vaut FAUX, la boucle n'est jamais exécutée !

Tests

Le test de base est le "si ... alors ... sinon" qui sera remplacé par le test "cas où ..." lorsque les possibilités de choix sont multiples et le choix ne porte que sur la valeur d'une seule variable.

Test "si ... alors ... sinon"

Test si ... alors ... sinon ...

si (test)

alors

Bloc d'instructions

<sinon

Bloc d'instructions

>

fin_si

si (a est divisible par 3)

alors

Bloc d'instructions

si (a < b)

alors

Bloc d'instructions

sinon

Bloc d'instructions

fin_si

Attention ! <sinon ...> est une partie optionnelle !

Test "cas où ... "

Test cas où ...

<pre>cas où (variable vaut) valeur 1 : Bloc d'instructions 1 valeur 2: Bloc d'instructions 2 valeur 3: Bloc d'instructions 3 ... valeur n: Bloc d'instructions n sinon Bloc d'instructions sinon fin_cas_où</pre>	<pre>cas où (reponse vaut) 1 : Bloc d'instructions 1 2 : Bloc d'instructions 2 3 : Bloc d'instructions 3 sinon Bloc d'instructions sinon fin_cas_où</pre>
--	---

Attention ! <sinon ...> est une partie optionnelle mais attention à ne pas faire de bêtise avec cette instruction !

Entrées/sorties

Il faut se placer du côté du programme (ou de la machine) pour juger si une action est une entrée ou une sortie. Une entrée sera en général une lecture au clavier (ou dans un fichier) et une sortie sera une écriture à l'écran (ou dans un fichier).

Ecrire(...) Lire(...)

<pre>Lire (variable) Ecrire (variable) Ecrire ("chaîne de caractères")</pre>	<pre>Lire (reponse) Ecrire (resultat) Ecrire ("Bonjour entrer une valeur SVP : ")</pre>
--	---

Valeur de retour

Il est possible de faire rendre ou retourner une valeur par un algorithme. Il s'agit de faire en sorte que votre algorithme/programme retourne une valeur exploitable par un autre algorithme/programme. L'instruction Rendre(...) **stoppe l'exécution des boucles**.

Attention ! Cette instruction est très différente de l'affichage qui ne fait qu'un "effet de bord" au niveau de l'écran (ou d'un fichier).

Rendre(...)

"Descriptions verbales"

Il s'agit ici de tout ce qui est en langage naturel et qui sert à décrire ce que fait l'algorithme. Lors de l'implémentation, il faudra traduire ces expressions en instructions exploitables par le programme.

Ce sont des expressions du type : "i est divisible par 10", "le reste de la division entière de truc par machin", ...

Booléens et expressions à valeur booléenne

Définitions

Bool est le nom d'un Mathématicien Anglais (1815 - 1864) connu, par les informaticiens, pour ses travaux sur l'Algèbre dite de Boole.

Les tests sont généralement une source non négligeable d'erreurs dans la rédaction des algorithmes, nous allons donc essayer de préciser un peu ce que nous entendons tests, booléens et expressions à valeur booléenne.

Au sens strict du terme les booléens sont au nombre de deux : VRAI (TRUE) et FAUX (FALSE) auxquels il convient d'ajouter un certain nombre d'opérateurs :

- Opérateurs de conjonction : ET (AND), OU (OR), NON (NOT), OU_EXCLUSIF (XOR), ...
- Opérateurs de comparaisons : <, >, <=, >=, =, <> (différent)

La conjonction de l'ensemble de ces opérateurs permet de construire une algèbre au sens Mathématique et des expressions à valeur Booléenne au sens informatique. Par exemple : $(a < b) \text{ ET } (a > c)$, $((a = b) \text{ ET } (c = d)) \text{ OU } (c = e)$, $\text{NON}((a = b) \text{ ET } (c = d))$, $((a = b) \text{ OU } \text{NON}(c < d))$, ...

Attention ! Il est fortement conseillé de ne pas hésiter à mettre différents niveaux de parenthèses dans vos expressions ! Il est en effet peu probable que vous maîtrisiez parfaitement les règles de priorité des opérateurs booléens. Par exemple $(a = b) \text{ ET } (c = a) \text{ OU } (c = d)$ si $a=5$, $b=6$, $c=5$, $d=5$? Doit-on faire le OU avant le ET ou le contraire ? Une expression bien parenthésée telle que $((a = b) \text{ ET } (c = a)) \text{ OU } (c = d)$ ou $(a = b) \text{ ET } ((c = a) \text{ OU } (c = d))$ ne posent quant à elles aucun problème. Si vous voulez la solution laissez donc la souris sur l'ampoule.

Tables de vérité

Tables de vérité des opérateurs ET, OU et NON

Table de vérité du ET			Table de vérité du OU			Table de vérité du NON		
ET	VRAI	FAUX	OU	VRAI	FAUX	NON	VRAI	FAUX
VRAI	VRAI	FAUX	VRAI	VRAI	VRAI		FAUX	VRAI
FAUX	FAUX	FAUX	FAUX	VRAI	FAUX			

Indentation de vos algorithmes et programmes

L'indentation est l'art et la manière de rajouter des espaces (ou des tabulations) en début de ligne d'un algorithme ou d'un programme afin de former des blocs parfaitement identifiables.

Vous aurez probablement remarqué que l'ensemble de mon code dans les algorithmes ci-dessus est parfaitement indenté et cela rajoute énormément à la lisibilité de vos algorithmes. C'est donc une habitude à prendre dès maintenant et à garder au moment de l'implémentation dans n'importe quel langage.

Comparez la lisibilité de l'algorithme écrit ci-dessous avec ou sans indentation !!!

Importance de l'indentation

Répéter

```
nbEssai <== nbEssai + 1
essai <== Lire(devine)
si essai = devine
alors Ecrire ("vous avez gagné")
sinon
    si (essai > devine)
    alors Ecrire ("le nombre est plus petit")
    sinon Ecrire ("le nombre est plus
grand")
fin_si
fin_si
jusque (essai = devine) OU (nbEssai >=3)
```

Répéter nbEssai <== nbEssai + 1

```
essai <== Lire(devine) si essai = devine
alors Ecrire ("vous avez gagné") sinon si
(essai > devine) alors Ecrire ("le nombre est
plus petit")
sinon Ecrire ("le nombre est plus grand")
fin_si fin_si jusque (essai = devine) OU
(nbEssai >=3)
```

2 Algos de base

Algorithmique : premiers algorithmes

Quelques exemples

Cette section présente quelques algorithmes simples permettant de se familiariser avec le raisonnement et la rédaction des algorithmes en pseudo-code.

Il est vivement conseillé :

1. de se poser les questions : l'énoncé est-il bien compris ? Suis-je capable de résoudre le problème à la main ? les étapes fondamentales sont-elles isolées ?
2. Si vous avez répondu OUI aux questions ci-dessus alors vous pouvez vous lancer dans l'écriture de l'algorithme en pseudo-code.

Suite d'entiers : calculer un terme particulier

Soit la liste définie par la relation de récurrence ci-dessous, on souhaite calculer et rendre le terme de rang 25 (U₂₅):

$$U_n = 3 U_{n-1} - 2$$
$$U_0 = 2$$

Calculer un terme particulier

```
res <== 2
Répéter pour i allant de 1 à 25
    res <== 3 * res - 2
fin_de_répéter
Rendre (res)
```

Suite d'entiers : calculer un terme répondant à un critère

Soit la liste définie par la relation de récurrence ci-dessous, on souhaite calculer et rendre le premier terme strictement supérieur à 100 :

$$U_n = 3 U_{n-1} - 2$$
$$U_0 = 2$$

Calculer un terme répondant à un critère

```
res <== 2
Répéter
    res <== 3 * res - 2
```

tant_que (res <= 100)
Rendre (res)

Suite d'entiers : calculer et afficher des termes

Soit la liste définie par la relation de récurrence ci-dessous, on souhaite calculer et afficher les 20 premiers termes :

$$U_n = 2 U_{n-2} + U_{n-1}$$
$$U_0 = 1, U_1 = 3$$

Cette définition de suite est doublement récurrente (le terme U_n est défini à partir de U_{n-2} et U_{n-1} et non à partir du seul U_{n-1} !

Calculer et afficher les termes de la suite

Avec boucle Répéter pour	Avec boucle Répéter ... tant_que
<pre>/* les deux termes de départ */ AvDernier <== 1 Dernier <== 3 Ecrire(AvDernier) Ecrire(Dernier) Répéter pour i allant de 1 à (20-2) /* Calcul et affiche le terme suivant */ Nouveau <== 2*AvDernier + Dernier Ecrire (Nouveau) /* Permutation des variables */ AvDernier <== Dernier Dernier <== Nouveau fin_de_répéter</pre>	<pre>/* les deux termes de départ */ AvDernier <== 1 Dernier <== 3 Ecrire(AvDernier) Ecrire(Dernier) i <== 1 Répéter /* Calcul et affiche le terme suivant */ Nouveau <== 2*AvDernier + Dernier Ecrire (Nouveau) /* Permutation des variables */ AvDernier <== Dernier Dernier <== Nouveau i <== i + 1 tant_que (i <= 20 - 2)</pre>

Calculer une approximation de la constante Pi

Chacun sait que la constante Pi est le rapport entre le périmètre et le rayon d'un cercle et vaut 3,14159...

Une méthode surprenante, au moins au premier abord, de calcul de la variable Pi consiste à tirer au hasard un certain nombre de points dans un carré donné et à compter combien sont inclus dans le cercle inscrit. Le rapport nombre de points inscrits/nombre total de points tend vers $\pi/4$.

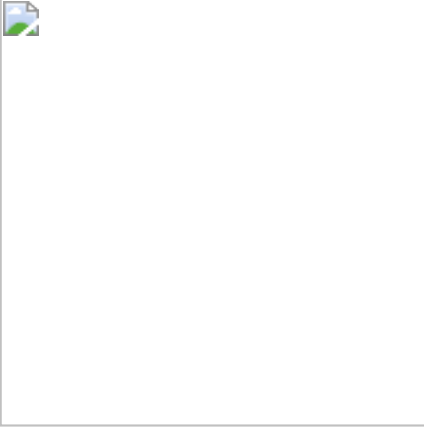
Indications utiles :

- Cette méthode est, bien entendu, valable quel que soit le carré considéré mais un choix judicieux de carré amène à une simplification relativement importante,
- plus le nombre de points est grand, meilleure est l'approximation,
- une bonne question à se poser avant de se lancer dans la programmation est de savoir quel critère

mathématique simple permet de déterminer si un point donné est inclus dans le cercle inscrit (faites vous un petit schéma).

Calculer, par la méthode du cercle inscrit décrite ci-dessus, une approximation de la constante Pi en effectuant autant d'itérations que demandé par l'utilisateur.

Approximation de la constante Pi

Indication "Mathématique"	Algorithme
	<pre>/* Les points sont tirés dans un carré de coté 1 dont le coin inférieur gauche est en (0,0) */ Lire (Nb_points) Nb_pts_inscrits <== 0 Répéter pour i allant de 0 à Nb_points-1 x <== réel au hasard entre 0 et 1 y <== réel au hasard entre 0 et 1 si (x-0.5)*(x-0.5) + (y-0.5)*(y-0.5) <= 0.25 alors Nb_pts_inscrits <== Nb_pts_inscrits + 1 fin_de_répéter Pi <== 4*(Nb_pts_inscrits / Nb_points) Ecrire ("Pi = ", Pi)</pre>

Faire tourner un algorithme "à la main"

Lors de l'écriture d'un algorithme ou d'un programme se pose toujours la question : "mon algorithme est-il correct ? Fait-il le travail demandé ? Comporte-t-il un "bug" ?"

Il n'existe malheureusement pas de méthode simple pour répondre à ces questions si ce n'est de se mettre à la place de la machine qui fera tourner l'algorithme et d'exécuter cet algorithme. Attention, c'est un exercice difficile car il faut avant tout vous abaisser au niveau d'une machine c'est à dire mettre en sommeil les milliards de neurones de votre cerveau et ne plus tourner que sur deux ou trois neurones maximum !! seul moyen, à ma connaissance, d'imiter un morceau de silicium.

En pratique, il s'agit d'exécuter **ligne par ligne** l'algorithme sur des exemples simples (mais pas trop) sans **rien inventer**, sans **rien omettre** et en **notant scrupuleusement la valeur de chacune des variables utilisée**.

Attention ! les "cas aux limites" sont souvent intéressant à tester. Par exemple si la borne maximale est elle-même un multiple de 3 dans l'exemple ci-dessous.

Exemple 1

Reprenons l'un de nos algorithmes d'affichage des multiples de 3 et exécutons le.

Affichage des multiples de 3 inférieurs à 100

```
i <== 1
Répéter
```

```

Ecrire (3 * i)
i <== i + 1
tant_que (3 * i < 100)

```

Tourner "à la main"

Variable i	Affichage écran
1	3
2	6
3	9
4	12
5	15
6	18
...	
32	96
33	99
34	

Noter qu'en sortie de boucle i vaut 34

Exemple 2

Reprenons notre algorithme du jeu de la devinette et exécutons le.

Petit jeu de devinette

```

/* Petit jeu de devinette */

devine <== nombre tiré au hasard entre 0 et 100
nbEssai <== 0
Répéter
    nbEssai <== nbEssai + 1
    Lire(essai)
    si essai = devine
    alors Ecrire ("Gagné")
    sinon si (essai > devine)
        alors Ecrire ("Plus petit")
        sinon Ecrire ("Plus grand")
tant_que (essai <> devine) ET (nbEssai <3) /* Solution avec répéter ... tantque */
/* Si perdu alors affichage message*/
si essai <> devine

```

alors Ecrire ("Perdu !!!")

Avec fin perdante

Variable <i>devine</i>	Variable <i>nbEssai</i>	Variable <i>essai</i>	Affichage écran
58	0		
	1	72	Plus petit
	2	45	Plus grand
	3	60	Plus petit Perdu !!!

Avec fin gagnante

Variable <i>devine</i>	Variable <i>nbEssai</i>	Variable <i>essai</i>	Affichage écran
37	0		
	1	83	Plus petit
	2	40	Plus petit
	3	37	Gagné

Nombre d'opérations élémentaires

Dans une section ultérieure, nous reviendrons plus en détails sur la notion de nombre d'opérations nécessaires à un algorithme pour s'exécuter. Néanmoins, il est souhaitable de se demander d'où et déjà et pour chacun de vos algorithmes combien d'opérations (multiplications, divisions, additions, affectations, tests, entrées/sorties, ...) sont nécessaires à son exécution. C'est ce nombre d'opérations qui nous permettra de dire qu'un algorithme est *meilleur* qu'un autre.

Ce calcul du nombre d'opérations se fait généralement en se plaçant dans **le pire cas possible**.

Exemple 1

Reprenons par exemple notre désormais habituel exemple des nombres divisibles par 3 avec les 3 méthodes possibles.

Affichage des multiples de 3 inférieurs à 100

Solution 3 : "on teste tous les

Solution 1 : "on ajoute 3 à chaque étape"	Solution 2: "on récite notre table de 3"	entiers entre 1 et 99 pour savoir si ils sont divisibles par 3"
Répéter pour i allant de 3 à 100 par pas de 3 Ecrire (i) fin_répéter	i <== 1 Répéter Ecrire (3 * i) i <== i + 1 tant_que (3 * i < 100)	Répéter pour i allant de 1 à 100 si (reste de i/3 = 0) alors Ecrire (i) fin_répéter
<ul style="list-style-type: none"> • 33 additions • 33 écritures 	<ul style="list-style-type: none"> • 33 additions • 2*33 multiplications • 33 écritures 	<ul style="list-style-type: none"> • 99 divisions entières • 99 tests • 33 écritures
La boucle va de 3 à 99 par pas de 3 donc s'exécute 33 fois donc 33 écritures et 33 additions.	La boucle va de 1 à 33 donc s'exécute 33 fois. Donc 33 additions et 33 écritures mais 2 multiplications à chaque passage de boucle, donc 66 multiplications !	La boucle va de 1 à 99 par pas de 1 donc s'exécute 99 fois. Il y a donc 99 tests, 99 divisions entières et 33 écritures. On peut même ajouter 33 additions "cachées" dans la boucle.

Il est clair au vu de tableau ci-dessus que la troisième solutions sera, de loin, la plus coûteuse en nombre d'opérations et donc en temps de calcul. Quant à savoir laquelle des deux premières solutions est la meilleure la discussion est tout aussi simple : **la solution 1 est la meilleure.**

La solution 2 pourrait facilement être améliorée en utilisant une petite variable dans la boucle permettant d'éviter une multiplication en la remplaçant par une affectation.

En l'occurrence, il semble que la solution la plus simple à programmer soit aussi la plus "économique". Méfiez vous de cette apparence, il n'en est pas toujours ainsi, nous le verrons bientôt !

Exemple 2

Reprenons notre petit jeu de la devinette et comptons les opérations nécessaires dans le pire cas possible c'est à dire lorsque le jeu se termine par une partie perdante après *NbEssai* essais infructueux.

Petit jeux de devinette

```

/* Petit jeu de devinette */
devine <== nombre tiré au hasard entre 0 et 100
nbEssai <== 0
Répéter
  nbEssai <== nbEssai + 1
  Lire(essai)
  si essai = devine
  alors Ecrire ("Gagné")
  sinon si (essai > devine)
    alors Ecrire ("Plus petit")
    sinon Ecrire ("Plus grand")
  tant_que (essai <> devine) ET (nbEssai <3) /* Solution avec répéter ... tantque */

```

```
/* Si perdu alors affichage message*/
```

```
Si essai <> devine
```

```
alors Ecrire ("Perdu !!!")
```

- *NbEssai* additions
- *NbEssai* + 1 écritures
- *NbEssai* lectures
- $4 * NbEssai + 1$ tests

La boucle s'exécute *NbEssai* fois, donc *NbEssai* additions , *NbEssai* lectures, *NbEssai* écritures et $2 * NbEssai$ tests (les 2 si). Sans oublier que chaque passage de boucle entraîne deux tests pour le `tant_que`. Un dernier test et une dernière écriture complète l'algorithme après la fin de la boucle.

3 Les listes

Listes mono-dimension : les bases

Définition

Une liste mono-dimensionnelle que nous appellerons très vite "liste" ou "tableau" est **un ensemble d'éléments homogènes regroupés et accessibles par un indice**.

La notion fondamentale ici est **homogène**, en effet un tableau (ou liste) ne peut comporter que des entiers ou que des flottants ou que des booléens ou que des enregistrements d'un certain type. En aucun cas, il n'est possible de mélanger le type des données contenues dans un tableau. Nous verrons plus tard qu'il existe d'autres structures (enregistrements ou autres) permettant de mélanger des données de différents types.

Accès aux éléments

On accède aux éléments du tableau en utilisant la notation `[]` et en précisant entre les crochets l'indice de l'élément dans le tableau. Par exemple, l'élément d'indice 8 du tableau T sera référencé par **T[8]** et utilisé comme une variable dans toutes nos instructions de pseudo-code.

Les indices sont obligatoirement des entiers. Les indices de nos tableaux commencent à 0.

Exemples :

- Lire(T[4]) pour lire la valeur du **cinquième** élément du tableau T,
- Ecrire(Tableau[0]) pour écrire la valeur du premier élément du tableau Tableau,
- $T[i] \leq T[i] + 1$ pour incrémenter la valeur de l'élément d'indice i du tableau T,
- **si** (Tab[i] >= Tab[i+1]) **alors** ... pour tester si l'élément d'indice i est supérieur à l'élément d'indice i+1

Attention ! Ne pas confondre indice et valeur d'un tableau ! Si T vaut [12, 58, 4, 6, 9] alors la valeur de l'élément d'indice 3 (T[3]) est 6 !

Taille d'un tableau

La taille d'un tableau est toujours une donnée prépondérante quelque soit le traitement à effectuer sur une liste. Dans l'ensemble de nos algorithmes nous postulons que **la taille de la liste est connue en entrée de l'algorithme**.

Si la taille est considérée comme connue en entrée, nous prendrons toujours grand soin de la mettre à jour si celle-ci est modifiée. Par exemple l'ajout (resp. la suppression) d'un élément dans une liste *T1* de taille *taille1* devra systématiquement entraîner l'incrément (resp. le décrétement) de la variable *taille1*.

Attention ! Un tableau de *taille* éléments aura des indices allant de 0 à (*taille*-1) !

Parcours simple des éléments d'un tableau

Il s'agit de parcourir tous les éléments d'un tableau afin de leur faire subir un traitement particulier (saisie, affichage, sommation, recopie sélective, ...). Le principe est simple, on utilise une variable (disons *i* pour ne pas être trop original) et utiliser cette variable comme indice de parcours du tableau.

Parcours simple d'un tableau

```
Répéter pour i allant de 0 à taille - 1
    Traitement
fin_répéter
```

Saisie des éléments

Il est ici question de saisir un à un tous les éléments du tableau. Vous noterez que nous lisons la taille avant toute chose et que nous prenons soin de faire rendre cette taille afin que celle-ci soit exploitable par n'importe quel autre sous-programme intervenant après cette saisie.

Saisie des éléments d'un tableau

```
Lire(taille)
Répéter pour i allant de 0 à taille - 1
    Lire(T[i])
fin_répéter
Rendre(taille)
```

Impression des éléments

C'est l'opération "inverse" de la précédente, on souhaite afficher à l'écran les éléments de la liste.

Impression des éléments d'un tableau

```
Répéter pour i allant de 0 à taille - 1
    Ecrire(T[i])
fin_répéter
```

Affichage de la somme et de la moyenne des éléments d'une liste

Il s'agit là encore d'un parcours simple de tableau avec utilisation d'une variable de type compteur qui est initialisée avant traitement et utilisée pour calculer la somme.

Affichage de la somme et de la moyenne

```
Somme <== 0
Répéter pour i allant de 0 à taille - 1
    Somme <== Somme + T[i]
fin_répéter
Ecrire("La somme est :", Somme)
Ecrire("La moyenne est :", Somme/taille)
```

Attention ! Cet algorithme AFFICHE la somme est la moyenne mais ne les REND PAS. C'est donc un sous-programme qui ne pourra pas être utilisé pour des traitements ultérieurs. Par exemple, il serait impossible d'utiliser ce sous-programme pour comparer la somme de deux listes distinctes.

Calcul et rendu de la somme

```
Somme <== 0
Répéter pour i allant de 0 à taille - 1
    Somme <== Somme + T[i]
fin_répéter
Rendre(Somme)
```

Cet algorithme peut LUI être utilisé comme sous-programme pour toutes sortes de traitements. Si l'on désire afficher la somme, il suffirait d'appeler le sous-programme si-dessus et d'afficher sa valeur de retour.

Attention ! Un algorithme (ou un sous-programme) ne peut rendre qu'une **unique** valeur.

Inversion d'une liste

La plupart des traitements effectués sur une liste peuvent l'être de deux manières différentes :

- **en copie** : la liste d'origine n'est pas modifiée par l'algorithme, c'est une nouvelle liste qui est construite
- **en place** : aucune autre liste n'est utilisée, c'est la liste d'origine qui est modifiée

Nous souhaitons inverser le sens d'une liste. La liste 5, 7, 4, 2, 10, 21 deviendra 21, 10, 2, 4, 7, 5

Inversion en copie

Inversion en copie d'une liste

```
/* T1 est de dimension taille1 */
j <== taille1 - 1
Répéter pour i allant de 0 à taille1 - 1
    T2[j] <== T1[i]
    j <== j - 1
fin_répéter
taille2 <== taille1
```

Inversion en place

Inversion en place d'une liste

```
/* T est de dimension taille */  
Répéter pour i allant de 0 à (taille - 1)/2  
    tmp <== T[i]  
    T[i] <== T[taille - 1 - i]  
    T[taille - 1 - i] <== tmp  
fin_répéter
```

Attention ! La permutation en place de deux éléments d'une liste nécessite obligatoirement l'utilisation d'une variable intermédiaire (*tmp* ici) au risque d'écraser tout simplement une moitié de la liste

Insertion d'un élément dans une liste

Insertion en copie

Soit une liste *T1* de dimension *taille1*, on souhaite insérer en copie (on construit une nouvelle liste) un entier *Nombre* en position *place*. Si *T1* est la liste : 5, 7, 4, 2, 10, 21 et le nombre à insérer est 150 en position 3 (indice 3) alors la liste résultat sera : 5, 7, 4, 150, 2, 10, 21

Insertion en copie d'un élément dans une liste

```
/* T1 est de dimension taille1 */  
/* Recopie de la première partie de la liste */  
Répéter pour i allant de 0 à place - 1  
    T2[i] <== T1[i]  
fin_répéter  
  
/* Insertion de l'élément */  
T2[place] <== Nombre  
  
/* Recopie de la fin de la liste */  
Répéter pour i allant de place+1 à taille1  
    T2[i] <== T1[i - 1]  
fin_répéter  
taille2 <== taille1 + 1
```

Insertion en place

Le problème est identique au précédant mais cette fois l'insertion se fait en place.

Insertion en place d'un élément dans une liste

```
/* T est de dimension taille */
```

```
/* On crée un "espace" à la bonne position en décalant la fin de la liste */
```

```
Répéter pour i allant de taille - 1 à place par pas de -1
```

```
    T[i + 1] <== T[i]
```

```
fin_répéter
```

```
/* Insertion de l'élément */
```

```
T[place] <== Nombre
```

```
taille <== taille + 1
```

Attention ! Le décalage vers la droite des éléments de la liste doit impérativement se faire en partant de la fin au risque d'écraser toute la fin de la liste si l'on fait la boucle de la gauche vers la droite.

Suppression d'un élément dans une liste

Suppression en copie

Soit une liste *T1* de dimension *taille1*, on souhaite supprimer en copie l'élément situé en position *place*. Si *T1* est la liste : 5, 7, 4, 2, 10, 21 et l'indice de suppression est 3 alors la liste résultat sera : 5, 7, 4, 10, 21

Suppression en copie d'un élément dans une liste

```
/* T1 est de dimension taille1 */
```

```
/* Recopie de la première partie de la liste */
```

```
Répéter pour i allant de 0 à place -1
```

```
    T2[i] <== T1[i]
```

```
fin_répéter
```

```
/* Recopie de la fin de la liste en décalant de manière à écraser le nombre souhaité */
```

```
Répéter pour i allant de place+1 à taille1 - 1
```

```
    T2[i - 1] <== T1[i]
```

```
fin_répéter
```

```
taille2 <== taille1 - 1
```

Suppression en place

Le problème est identique au précédant mais cette fois la suppression se fait en place.

Suppression en place d'un élément dans une liste

```
/* T est de dimension taille */
```

```
/* On décale la fin de la liste */
```

```
Répéter pour i allant de place à taille - 2
```

```
    T[i] <== T[i + 1]
```

```
fin_répéter
```

taille <== taille - 1

Attention ! La fin de boucle est en taille - 2 afin de ne pas sortir du tableau

Nombre d'occurrences

Occurrence d'un entier donné

Le but de cet algorithme est de compter combien de fois un entier donné apparaît dans une liste. Par exemple si la liste T est : 12, 5, 5, 14, 12, 25, 5, 17, 9 alors l'entier 5 apparaît 3 fois, l'entier 12 apparaît 2 fois, le 9 une seule fois et le 45 apparaît 0 fois.

Cet algorithme est utilisé relativement fréquemment lors de différents types de traitements (jeux de cartes ou de dés, statistiques divers, ...).

Nombre d'occurrence d'un entier dans une liste

```
/* T est de dimension taille */
/* Nombre est le nombre cherché */
compteur <== 0
Répéter pour i allant de 0 à taille - 1
    si (Nombre = T[i])
        alors compteur <== compteur + 1
fin_répéter
Rendre(compteur)
```

Une rapide analyse de cet algorithme nous permet de dire que la boucle est exécutée *taille* fois, nous aurons donc *taille* tests. N'oublions pas que l'algorithme ci-dessus peut être utilisé quel que soit le type d'information contenue dans le tableau T. Si T est un tableau d'entiers alors chaque test est un test entre deux entiers mais si T est un tableau de chaînes de caractères alors nous aurons à tester si une chaîne est identique à une autre (problème beaucoup plus complexe déjà !). Poussons encore le raisonnement, si T contient des informations encore plus complexes (des enregistrements comportant chacun un grand nombre d'informations par exemple) alors les comparaisons seront encore plus difficiles et donc plus lentes.

Liste d'occurrences

L'algorithme précédant peut être utilisé pour répondre à la question suivante : Construire une liste *Tocc* des occurrences de tous les entiers contenus dans une liste *T*. En indice 0 de la liste *Tocc* nous aurons alors le nombre de d'occurrences de l'entier 0, en indice 1 nous aurons le nombre d'occurrences de l'entier 1, ..., en indice *i* nous aurons le nombre d'occurrences de l'entier *i*.

Attention ! il est nécessaire de connaître les bornes des entiers contenus dans T. En effet, si les entiers de T sont compris entre 0 et 100 (bornes incluses) alors le tableau *Tocc* comportera 101 éléments.

Exemple :

T : 12, 5, 5, 14, 12, 25, 5, 17, 9 (les éléments de T sont dans l'intervall [0, 25])

Tocc : 0, 0, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 2, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1

Cet algorithme sera par exemple utilisé pour la programmation d'un jeu de dés nommée YATZEE ou YAMS. Le joueur dispose de 5 dés, un dé peut prendre les valeurs 1 à 6. Après le jet des dés, il importe de détecter les paires, brelans, carrés, fulls, quinte, ... Un tableau d'occurrences sera alors construit et la simple présence d'un 3 dans le tableau indiquera la présence d'un brelan, la présence d'un 3 et d'un 2 indiquera un full, ...

Pour écrire cet algorithme, nous considérerons que nous disposons de l'algorithme précédant en tant que sous programme se nommant Nb_occ et prenant 3 paramètres : une liste, la taille de la liste, le nombre à chercher. Un appel au sous-programme se fera donc de la manière suivante : Nb_occ(T, 12, 5) si l'on cherche le nombre d'occurrences de l'entier 5 dans la liste T de taille 12.

Construction d'une liste d'occurrences

```
/* T est de dimension taille et contient des nombres compris entre 0 et max */  
Répéter pour i allant de 0 à max  
    Tocc[i] <== Nb_occ(T, taille, i)  
fin_répéter  
tailleTocc <== max + 1
```

Comptons le nombre d'opérations exécutées dans cet algorithme. La boucle ci-dessus est exécutée ($max+1$) fois, nous avons ($max+1$) appels au sous-programme Nb_occ. Chaque appel à Nb_occ nécessitant *taille* tests (voir ci-dessus), le nombre total de tests de cet algorithme est donc de $(max+1)*taille$ tests. Ce nombre de tests peut rapidement être considérable et les tests eux-mêmes peuvent être fort complexes.

Dans le cas de tableaux d'entiers, il existe un algorithme pouvant s'exécuter **sans aucun test**, je vous renvoie à la rubrique exercices.

Recherche du maximum

La recherche de l'élément maximum (ou minimum) dans une liste est un "grand classique". La méthode consiste à parcourir le tableau du début à la fin en comparant systématiquement l'élément courant au maximum actuel.

Recherche du maximum

```
/* T est de dimension taille */  
max <== T[0]  
Répéter pour i allant de 1 à (taille - 1)  
    si (T[i] > max)  
        alors max <== T[i]  
fin_répéter  
Rendre(max)
```

Attention ! vous noterez que la boucle démarre à 1 et non 0, il est en effet inutile de comparer l'élément 0 avec lui-même.

Recherche simple ou séquentielle d'un élément dans une liste

L'un des traitements les plus courants sur les listes est la recherche d'un élément précis dans la liste. Il s'agit de parcourir la liste en cherchant un élément donné et stopper le parcours de la liste dès que l'on trouve l'élément en question (la première occurrence de l'élément), la valeur de retour est alors l'indice de l'élément dans le tableau. Si l'élément ne figure pas dans la liste l'algorithme rend -1. Cette recherche se nomme "recherche simple" ou "recherche séquentielle".

Si la liste n'est pas triée (ordre croissant par exemple), la recherche simple est alors le seul algorithme possible de recherche. Nous verrons plus tard qu'il existe un algorithme de recherche beaucoup plus efficace dans le cas d'une liste triée.

Recherche simple ou séquentielle

méthode privilégiée	autre méthode
<pre>/* T est de dimension taille */ /* Nombre est l'élément cherché */ i <== 0 tant_que (i < taille) ET (T[i] <> Nombre) répéter i <== i + 1 fin_tant_que si (i >= taille) alors Rendre(-1) sinon Rendre(i)</pre>	<pre>/* T est de dimension taille */ /* Nombre est l'élément cherché */ i <== -1 Répéter i <== i + 1 tant_que (i < taille-1) ET (T[i] <> Nombre) si (i >= taille) alors Rendre(-1) sinon Rendre(i)</pre>

Attention ! je vous demande d'éviter, dans la mesure du possible (et c'est presque toujours possible), de forcer les sorties de boucles. Une boucle **Répéter** pour i allant de 0 à (taille - 1) comportant une instruction Rendre(...) est ce que j'appelle une sortie de boucle "forcée".

Attention ! Après la boucle, il est plus "prudent" de tester si (i >= taille) plutôt que (T[i] = Nombre) car si Nombre n'appartient pas à la liste, le i "pointe" en dehors de la liste et il vaut mieux éviter de tester cette valeur que nous ne connaissons pas !

4 Les listes mono-dimension (suite)

Algorithmes sur UNE liste mono-dimension

Détection de la position du premier entier vérifiant un critère

Il s'agit de trouver la position du premier élément d'une liste vérifiant un critère donné (élément pair, divisible par 3, égal à une valeur, ...). Le critère peut être absolument quelconque, il suffit de disposer d'un prédicat (sous programme retournant Vrai ou Faux). Il n'y a en fait rien à ajouter puisque l'algorithme est exactement le même que celui de la recherche simple. Par exemple, la recherche du premier élément divisible par 3 dans une liste se programme de la manière suivante :

Recherche du premier élément vérifiant un critère

```
/* T est de dimension taille */  
i <= -1  
Répéter  
    i <= i + 1  
tant_que (i < taille) ET (T[i] NON divisible par 3)  
si (T[i] divisible par 3)  
alors Rendre(i)  
sinon Rendre(-1)
```

Suppression en place du dernier entier pair

La suppression en place d'un élément dans une liste a déjà été traitée. Il s'agit ici de combiner cette suppression en place avec la recherche d'un élément qui, dans le cas qui nous intéresse, est le DERNIER élément vérifiant le critère (être pair). Rien de bien compliqué, il suffit de partir de la fin de la liste et de combiner les deux algorithmes de recherche simple et de suppression en place.

Suppression en place du dernier élément pair d'une liste

```
/* Supprime le dernier éléments pair de T (de dimension taille) dans T elle même */  
/* Recherche du dernier entier pair */  
i <= taille - 1  
Tant_que (i >= 0) ET (T[i] est non pair) répéter  
    i <= i - 1  
fin_tant_que  
si i < -1 /* la liste comporte bien au moins un élément pair */  
alors /* suppression de l'élément en question */  
    Répéter pour j allant de i à taille - 2  
        T[j] <= T[j+1]  
    fin_répéter  
taille <= taille - 1
```

Comptage des éléments multiples

Il s'agit de compter combien d'éléments de la liste apparaissent plus d'une fois dans celle-ci. Un exemple valant mieux qu'un long discours :

résultat = 0 pour T : 3 7 11 14 18 22 car aucun entier n'apparaît plus d'une fois

résultat = 1 pour T : 18 27 14 23 27 3 27 5 car seul 27 apparaît plus d'une fois

résultat = 3 pour T : 10 15 10 15 10 22 4 4 25 78 car 10, 15 et 4 apparaissent plus d'une fois

Cet algorithme peut sembler facile au premier abord mais sera faux si l'on n'y prend garde. En effet, la première idée consiste à parcourir la liste de gauche à droite et, pour chaque valeur, parcourir le reste de la liste (à droite de la position courante) pour voir si l'élément courant apparaît une ou plusieurs autres fois. En gros, il s'agit d'une double boucle pouvant s'écrire de la manière suivante :

Comptage du nombre d'entiers apparaissant plus d'une fois - version TROP simple

```
/* T de dimension taille */
resultat <== 0
Répéter pour i allant de 0 à taille -1
    j <== i+1
    /* Recherche de la valeur T[i] dans la partie droite de la liste T */
    Tant_que (T[i] <> T[j]) ET (j < taille-1)
        j <== j + 1
    fin_tant_que

    si T[i] = T[j]
        alors resultat <== resultat + 1
    fin_répéter
Rendre (resultat)
```

Attention ! Cette méthode fonctionne tant qu'aucun élément n'apparaît plus de 2 fois mais rend un résultat erroné sinon. Faisons tourner l'algorithme à la main sur le troisième exemple ci-dessus (T : 10 15 10 15 10 22 4 4 25 78) dans lequel la valeur 10 apparaît 3 fois, le résultat sera 4 alors que 3 chiffres seulement (10, 15 et 4) apparaissent plusieurs fois !!! Explications : le premier 10 (T[0]) sera compté comme multiple car comparé au second 10 (T[2]) mais ensuite l'algorithme va continuer et le second 10 (T[2]) sera utilisé, à son tour, comme valeur référence et comptabilisé à cause du troisième 10 (T[4]).

L'une des solutions possible pour remédier à ce problème et de modifier la liste T dès que l'on a trouvé un élément en plusieurs exemplaires en remplaçant les occurrences suivantes dans la liste par une valeur "bidon" qui pourra être une valeur négative par exemple dans le cas d'une liste d'entiers positifs.

Comptage du nombre d'entiers apparaissant plus d'une fois - version correcte mais PEU ELEGANTE

```
/* T de dimension taille ne contient que des entiers positifs ou nuls */
/* ATTENTION T est modifiée !! */
```

```

resultat <== 0
Répéter pour i allant de 0 à taille -1
  si (T[i] <> -1) /* on ne compte pas l'élément "bidon" */
  alors
    j <== i+1
    /* Recherche de la valeur T[i] dans la partie droite de la liste T */
    Tant_que (T[i] <> T[j]) ET ( j < taille )
      j <== j + 1
    fin_tantque

    si T[i] = T[j]
    alors resultat <== resultat + 1
    /* Remplacement de T[i] par un élément "bidon" */
    /* (ici -1 car T est une liste d'entiers positifs) */

      Répéter pour k allant de j à taille - 1
        si (T[k] = T[i])
        alors T[k] <== -1
      fin_répéter
    fin_répéter
  Rendre (resultat)

```

Il existe d'autres méthodes probablement un peu plus élégantes mais nécessitant la construction d'une ou plusieurs listes intermédiaires de travail. Nous reviendrons sur ces algorithmes un peu plus tard dans les exercices d'application.

Il est toutefois possible de modifier l'algorithme ci-dessus afin que, lors de chaque détection d'un "doublon", il supprime toutes les occurrences de ce "doublon" trouvées à sa droite dans la liste.

Comptage du nombre d'entiers apparaissant plus d'une fois - version CORRECTE

```

/* T de dimension taille */
resultat <== 0
Répéter pour i allant de 0 à taille -1
  j <== i+1
  /* Recherche de la valeur T[i] dans la partie droite de la liste T */
  Tant_que (T[i] <> T[j]) ET ( j < taille )
    j <== j + 1
  fin_tant_que

  si T[i] = T[j]
  alors resultat <== resultat + 1
  /* suppression de tous les T[i] dans la partie droite de la liste T */
  k <== j

```

```

    Tant_que ( k < taille ) répéter
        si T[ k ] = T[ i ]
            alors Suppression_element(T, k)
            /* on avance le k uniquement si aucune suppression n'a eu lieu */
            sinon k <== k + 1
        fin_tant_que
    fin_répéter
    Rendre (resultat)

```

Algorithmes portant sur DEUX listes mono-dimension

Union en copie

Rien de bien difficile, il s'agit de construire une nouvelle liste représentant la réunion des deux listes de départ, c'est à dire tous les éléments de la première liste ET tous les éléments de la seconde.

Union en copie de deux listes

```

/* T1 et T2 de dimensions respectives taille1 et taille2 */
Répéter pour i allant de 0 à ( taille1 - 1 )
    Tunion[i] <== T1[i]
fin_répéter
Répéter pour i allant de 0 à ( taille2 - 1 )
    Tunion[taille1 + i] <== T2[i]
fin_répéter
taille_Tunion <== taille1 + taille2

```

Détection de la première différence

Il s'agit d'un algorithme simple permettant de connaître l'indice de la première différence entre deux listes éventuellement de tailles différentes. Rien de très difficile.

Détection de la première différence entre deux listes

```

/* T1 et T2 de dimensions respectives taille1 et taille2 */
/* Rend l'indice de la première différence s'il y a une différence, -1 si aucune différence */
i <== -1
Répéter
    i <== i + 1
tant_que ( i < taille1 ) ET ( i < taille2 ) ET ( T1[i] = T2[i] )
si ( T1[i] <> T2[i] )
    alors Rendre(i)
sinon /* pas de différence */

```

```

    si (taille1 = taille2)
        alors Rendre(-1)

```

Intersection en copie

Considérons deux listes T1 et T2 de dimensions respectives *taille1* et *taille2*, on souhaite construire une liste Tinter contenant les éléments communs à T1 et T2. Dans un premier temps, nous ne nous soucierons pas d'éventuels éléments apparaissant plusieurs fois dans l'une ou l'autre des deux listes.

Intersection en copie de deux listes : algo TROP simple

```
/* T1 et T2 de dimensions respectives taille1 et taille2 */
k <== 0 /* indice servant à remplir Tinter */
Répéter pour i allant de 0 à (taille1 - 1)
  Répéter pour j allant de 0 à (taille2 - 1)
    si (T1[i] = T2[j]) /* on a trouvé un élément commun */
      alors Tinter[k] <== T1[i]
        k <== k + 1
  fin_répéter
fin_répéter
tailleTinter <== k
```

Cet algorithme a deux inconvénients importants :

1. Pour chaque valeur de T1, il parcourt TOUT le tableau T2 (même si l'élément cherché est en première position dans T2 !)
2. Toute valeur de T1 apparaissant plus d'une fois dans T2 sera copiée plusieurs fois dans Tinter

Pour résoudre ces deux problèmes, il convient de remplacer la seconde boucle "répéter pour " par une boucle "répéter ... tant_que".

Intersection en copie de deux listes

```
/* T1 et T2 de dimensions respectives taille1 et taille2 */
k <== 0 /* indice servant à remplir Tinter */
Répéter pour i allant de 0 à (taille1 - 1)
  j <== -1
  Répéter
    j <== j + 1
  tant_que (j < taille2) ET (T2[j] <> T1[i])
  si (j <> taille2) /* on a trouvé un élément commun */
    alors Tinter[k] <== T1[i]
      k <== k + 1
  fin_répéter
tailleTinter <== k
```

Si nous regardons de près l'algorithme ci-dessus, il apparaît clairement que la boucle imbriquée (celle utilisant la variable j) n'est autre qu'une recherche simple. Nous pouvons donc ré-écrire cet algorithme en faisant directement appel à un sous programme dont la signature est *Rech_simple(Nombre, T, taille)* pour chercher *Nombre* dans le tableau T de dimension *taille*.

Intersection en copie de deux listes avec appel de sous-programme

```
/* T1 et T2 de dimensions respectives taille1 et taille2 */
j <== 0
Répéter pour i allant de 0 à (taille1 - 1)
    si Rech_simple(T1[i], T2, taille2) <> -1
        alors Tinter[j] <== T1[i]
            j <== j + 1
fin_répéter
tailleTinter <== j
```

Attention ! En cas d'élément apparaissant plus d'une fois dans l'une des listes, celui-ci risque d'apparaître également plus d'une fois dans la liste intersection !!

Attention ! Le résultat risque d'être différent si l'on inverse l'ordre des listes !!

Vous trouverez dans la partie exercices un algorithme résolvant ce problème.

Exemple

T1 : 12 6 8 9 12 7 T2 : 7 12 9 4 2	Tinter : 12 9 12 7
T2 : 7 12 9 4 2 T1 : 12 6 8 9 12 7	Tinter : 7 12 9

Différence en copie

Peu de choses à dire, il s'agit ici d'obtenir une liste contenant les éléments de la première liste n'apparaissant PAS dans la seconde. L'ordre est ici important et c'est normal. L'algorithme est rigoureusement identique au précédent si ce n'est que le résultat de la recherche simple doit être -1 pour que l'élément soit intégré à la liste différence.

5 Recherches dans une liste

Recherche simple : rappel

Je vous invite à vous reporter à la section du cours traitant de la recherche simple et je ne fait que répéter ici que cette méthode de recherche est la seule possible, ou presque, dans le cas d'une liste non ordonnée. L'algorithme est simplement ré-écrit ci-dessous par soucis de commodité.

Recherche simple ou séquentielle

méthode privilégiée	autre méthode
<pre>/* T est de dimension taille */ /* Nombre est l'élément cherché */ i <== 0 tant_que (i < taille) ET (T[i] <> Nombre) répéter i <== i + 1 fin_tant_que si (T[i] = Nombre) alors Rendre(i) sinon Rendre(-1)</pre>	<pre>/* T est de dimension taille */ /* Nombre est l'élément cherché */ i <== -1 Répéter i <== i + 1 tant_que (i < taille-1) ET (T[i] <> Nombre) si (T[i] = Nombre) alors Rendre(i) sinon Rendre(-1)</pre>

Recherche au hasard !!

La recherche simple est "presque" la seule possible dans le cas d'une liste non ordonnée car on peut toujours imaginer une méthode dite "randomisée" consistant à tirer au hasard des nombres dans la liste jusqu'à trouver le nombre cherché ou à avoir tiré assez de nombre pour que la théorie des probabilités nous dise que le nombre a "très peu de chances de figurer dans la liste".

Cet algorithme peut se programmer et même être utilisé dans certains cas bien précis vérifiant généralement les propriétés suivantes :

1. très grandes listes
2. recherche d'éléments ayant peu de chance de se trouver dans la liste
3. erreurs de l'algorithme "acceptables"

Nous ne décrivons pas plus avant cet algorithme.

Recherche dichotomique dans une liste TRIÉE

Je ne répéterais jamais assez que ce qui suit concerne **uniquement les listes triées** donc des éléments pour lesquels il est possible de donner ce que les mathématiciens nomment une relation d'ordre, c'est à dire que l'on est capable de dire qu'un élément est "plus petit" ou "plus grand" qu'un autre.

Exemples : l'ordre croissant pour les entiers ou les flottants, l'ordre lexicographique pour les chaînes de caractères.

Dichotomie : définition et exemples

En algorithmique, la dichotomie (du grec "couper en deux") est un processus itératif de recherche où à chaque étape l'espace de recherche est restreint à l'une des deux parties.

On suppose bien sûr qu'il existe un test relativement simple permettant à chaque étape de déterminer l'une des deux parties dans laquelle se trouve la solution. Pour optimiser le nombre d'itérations nécessaires, on s'arrangera pour choisir, à chaque étape, deux parties sensiblement de la même "taille" (pour un concept de "taille" approprié au problème). Nous verrons plus tard que, dans ce cas, le nombre total d'itérations nécessaires à la complétion de l'algorithme est alors logarithmique en la taille totale du problème initial.

L'algorithme s'applique typiquement à la recherche d'un élément dans un ensemble fini ordonné organisé en séquence ou liste. A chaque étape, on coupera l'espace de recherche (la liste ou une partie de la liste) en deux parties de même taille (à un élément près) de part et d'autre de l'élément médian.

La dichotomie peut être vue comme une variante simplifiée de la stratégie plus générale diviser pour régner (en anglais, "divide and conquer").

Exemple 1 : petit jeu de cours de récréation

Prenons un exemple simple et ludique pour illustrer le mécanisme de recherche par dichotomie : Pierre propose à Paul le jeu suivant : "choisis en secret un nombre compris entre 0 et 100; je vais essayer de le deviner le plus rapidement possible, mais tu ne dois répondre à mes questions que par oui ou par non".

Paul choisit 65 et attend les questions de Pierre:
est-ce que le nombre est plus grand que 50? ($(0 + 100) / 2$)
oui
est-ce que le nombre est plus grand que 75? ($(50 + 100) / 2$)
non
est-ce que le nombre est plus grand que 63? ($(50 + 75 + 1) / 2$)
oui
Pierre réitère ses questions jusqu'à trouver 65. Par cette méthode itérative, Pierre est sûr de trouver beaucoup plus rapidement le nombre qu'en posant des questions du type "est-ce que le nombre est égal à 30?".

Exemple 2 : Mathématiques

Cette méthode est très efficace pour la recherche des zéros approchés d'une fonction à condition que la fonction soit approchée par une fonction linéaire au voisinage du zéro cherché:

Soit $f(x)$ une fonction telle que:
 $f(a) < 0$
 $f(b) > 0$
 f est strictement croissante entre les points a et b ($a < b$)

Alors une dichotomie permet de trouver rapidement la valeur y telle que $f(y) = 0$.

- * partir du couple de valeurs (a, b) ;
- * évaluer la fonction en $(a+b)/2$;
- * si $f((a+b)/2) < 0$, remplacer a par $(a+b)/2$, sinon remplacer b par $(a+b)/2$;
- * recommencer à partir du nouveau couple de valeurs jusqu'à ce que la différence entre les deux valeurs soit inférieure à la précision voulue.

Recherche dichotomique : les algorithmes

Nous supposons que la liste considérée est triée en ordre croissant et donnerons deux algorithmes légèrement différents :

- le premier rendra la place de l'élément cherché si cet élément appartient à la liste et -1 si il n'appartient pas
- le second rendra la place d'insertion de l'élément cherché en supposant que l'on souhaite insérer l'élément à la bonne place dans la liste triée

Dans les deux cas, la méthode consiste à utiliser deux variables, ici nommées *debut* et *fin*, permettant de "cadrer" la recherche dans une partie donnée de la liste. Ces deux indices permettent de trouver l'indice de l'élément médian dans la partie de liste considérée. Cet élément médian est comparé à l'élément recherché : si on a trouvé, l'algorithme est interrompu sinon, on recommence sur la partie à droite de l'élément médian si l'élément cherché est plus grand que l'élément médian ou dans la partie à gauche de l'élément médian si l'élément cherché est plus petit. L'algorithme stoppe lorsque l'on a trouvé l'élément ou lorsque les indices *debut* et *fin* se "téléscoquent", donc que la zone de recherche est vide.

Place de l'élément

Recherche dichotomique : place de l'élément

```
/* Retourne la place de l'élément x dans la liste TRIEE tab de longueur taille */
/* Retourne -1 si x n'est pas dans la liste*/
debut <== 0
fin <== taille - 1
place <== -1
Répéter
    med <== (debut + fin)/2
    si tab[med] = x
        alors place <== med
    sinon
        si tab[med] > x
            alors fin <== med - 1
        sinon debut <== med + 1
jusque (debut>fin) OU (place <> -1) /* ou alors */ tant_que (debut <= fin) ET (place = -1)
Rendre (place)
```

Faisons tourner cet algorithme "à la main" sur un exemple simple (l'élément d'indice debut sera en rouge, l'élément d'indice fin en bleu et l'élément médian en **gras**) :

Cherchons l'élément 8 :

				démarrage				
5	8	12	17	21	28	32	44	50
				8 < 21				
5	8	12	17	21	28	32	44	50
				8 = 8				

sortie et retour avec place = 1

Cherchons l'élément 63 :

démarrage									
5	8	12	17	21	28	32	44	50	_

car 63 > 21									
5	8	12	17	21	28	32	44	50	–
car 63 > 32									
5	8	12	17	21	28	32	44	50	–
car 63 > 44									
5	8	12	17	21	28	32	44	50	–
car 63 > 50									
5	8	12	17	21	28	32	44	50	–
debut et fin sont inversés donc sortie et retour avec place = -1									

Place d'insertion de l'élément

Dans le cas où seule la place d'insertion nous intéresse, l'algorithme est un peu plus simple puisqu'il est inutile de comparer l'élément cherché à l'élément médian, on fait simplement tourner l'algorithme jusqu'au "télécopage" des indices *debut* et *fin*. Il faut noter que si l'élément que l'on souhaite insérer est déjà dans la liste, le présent algorithme fonctionne tout de même parfaitement.

Recherche dichotomique : place d'insertion de l'élément

```

/* Retourne la place d'insertion de l'élément x dans la liste TRIEE tab de longueur taille */
debut <== 0
fin <== taille - 1
Répéter
    med <== (debut + fin)/2
    si tab[med] > x
        alors fin <== med - 1
        sinon debut <== med + 1
tant_que (debut<=fin)
Rendre (debut)

```

Faisons tourner cet algorithme "à la main" sur un exemple simple (l'élément d'indice debut sera en rouge, l'élément d'indice fin en bleu et l'élément médian en **gras**) :

Cherchons la place d'insertion de l'élément 6 :

démarrage								
05	08	12	17	21	28	32	44	50
6 < 21								
05	08	12	17	21	28	32	44	50
6 < 8								

05	08	12	17	21	28	32	44	50
6 > 5								
05	08	12	17	21	28	32	44	50
debut et fin sont inversés donc sortie et retour avec place=debut=1								

Recherche indexée

L'algorithme de recherche dichotomique dans une liste triée est très efficace (nous verrons cela dans un prochain chapitre sur la complexité) mais peut, dans certains cas, être encore amélioré.

Prenons l'exemple de la recherche d'un nom dans un annuaire téléphonique personnel (le petit répertoire papier ou même l'annuaire de votre téléphone portable) ou d'un mot dans un dictionnaire. Bien entendu, l'annuaire et le dictionnaire sont des listes TRIÉES dans l'ordre lexicographique croissant (ordre de l'alphabet), on peut donc faire une recherche dichotomique dans ces documents. On ouvre l'annuaire ou le dictionnaire au milieu et si le mot cherché est "plus grand" (au sens lexicographique) on recoupe en deux la partie "droite" sinon on recoupe en deux la partie "gauche". Cette méthode va nous conduire assez rapidement à trouver la page dans laquelle se trouve le numéro de téléphone convoité ou le mot recherché.

Question : procédez-vous comme cela ? Il y a fort à parier que non ! En effet, dans un calepin vous disposez d'onglets avec une lettre, vous sautez donc directement à la bonne page puis vous cherchez dans cette page. Même chose avec votre cher téléphone portable, vous sautez immédiatement à la lettre recherchée puis vous passez les numéros en revue ou, si votre téléphone le permet, vous tapez la seconde lettre ce qui vous rapproche encore un peu du nom recherché. Dans un cas comme dans l'autre, vous venez d'effectuer une **recherche indexée**. Dans cette notion d'indexation deux concepts sont importants :

1. la fréquence de l'indexation : on peut poser un index (une marque) à chaque lettre de l'alphabet ou toute les trois lettres par exemple
2. le niveau de l'indexation : index sur la première lettre d'un mot puis, à l'intérieur de cette première lettre, une autre indexation sur la seconde lettre puis sur la troisième, etc ...

Attention ! Cette méthode de recherche peut être très efficace mais un important pré-traitement et un effort constant pour les mises à jour. Il faut, bien entendu, que la liste soit triée dès le départ mais aussi que cette liste soit indexée. Encore plus délicat : il faut que la liste reste triée et l'index reste à jour lors de tout ajout ou suppression dans la liste. **L'algorithme de recherche dans une liste indexée est donc utilisé en priorité pour des listes rarement mises à jour mais qui sont l'objet de recherches fréquentes.**

Indexation d'une liste d'entiers

Après toutes ces belles paroles, il serait dommage de ne pas écrire deux petits algorithmes : l'un pour l'indexation et l'autre pour la recherche dans une liste indexée.

Restons dans notre problématique des listes d'entiers et cherchons à indexer notre liste selon une fréquence donnée : un index à chaque changement de dizaine par exemple.

Exemple : La liste triée d'entiers

4 ; 12 ; 34 ; 44 ; 46 ; 52 ; 55 ; 59 ; 74 ; 88 ; 125 ; 148

se verrait dotée, pour une fréquence de 10, d'une liste d'index

0 ; 1 ; -1 ; 2 ; 3 ; 5 ; -1 ; 8 ; 9 ; -1 ; -1 ; -1 ; 10 ; -1 ; 11

signifiant que les unités commencent à l'indice 0, les dizaines à l'indice 1, qu'il n'y a pas de

vingtaines, les trentaines commencent à l'indice 2, ..., les cinquantaines commencent à l'indice 5, ...

Indexation d'une liste d'entiers

```
/* Indexation (dans la liste index de dimension taille_index) de la liste triée tab de dimension taille
*/
/* la fréquence d'index est freq */

/* La taille de l'index est fonction de l'entier maximum dans tab */
taille_index <== tab[taille-1] / freq + 1

/* Initialisation du tableau des index à -1 */
Répéter pour i allant de 0 à taille_index -1
    index[i] <== -1
fin_répéter

/* Passage de l'index */
Répéter pour i allant de 0 à taille - 1
    quotient <== tab[i] / freq
    /* si non encore indexé */
    Si index[quotient] = -1
        alors index[quotient] = i
    fin_répéter
```

Algorithme de recherche indexée

Recherche de la place d'insertion dans une liste d'entiers indexée

```
/* Recherche de la place d'insertion de x dans une liste triée tab de dimension taille */
/* tab possède une liste d'index index de dimension taille_index (fréquence d'index de freq) */

/* Si x est plus grand que tous */
Si x >= tab[taille-1]
    alors Rendre(taille)
sinon
    /* Endroit du début de la recherche */
    quotient <== x / freq
    i <== quotient
    tant_que (index[i] = -1) ET (i >= 0)
        i <== i - 1
    fin_tant_que
    Si i = -1 /* uniquement des -1 en début de l'index */
        alors debut <== 0
```

```

sinon debut <== index[i]

/* Endroit de fin de la recherche*/
i <== quotient
  tant_que (index[i] = -1)
    i <== i + 1
  fin_tant_que
fin <== index[i]
/* Recherche de x entre debut et fin */
/* recherche SIMPLE dans ce cas mais on pourrait faire une DICHOTOMIE car tab est triée */
i <== debut
  tant_que (tab[i] < x) ET (i <= fin)
    i <== i + 1
  fin_tant_que
Rendre(i)

```

Bien entendu, nous avons soigneusement évité de parler du tri de la liste avant recherche dichotomique mais c'est l'objet d'un autre chapitre de ce cours intitulé : Algorithmique : les tris simples.

6 Complexité

Introduction

La théorie de la complexité algorithmique est indispensable à une bonne utilisation des algorithmes. Elle répond à la question : entre différents algorithmes réalisant une même tâche, lequel est le plus rapide ?

Dans les années 60 et 70, alors que l'activité de découverte des algorithmes fondamentaux battait son plein, on savait mal mesurer leur efficacité. On se contentait de dire : "ce nouvel algorithme (de tri) se déroule en 5 minutes 12 centièmes avec un tableau de 500 entiers choisis au hasard en entrée, sur un ordinateur MARK XX de la société YY. Avec un tableau de 1000 éléments, cela prend 7 minutes 72 centièmes. Le langage de programmation PL/I avec les optimisations standard ont été utilisées." (c'est un exemple totalement imaginaire).

Une telle démarche rendait quasiment impossible la comparaison des algorithmes entre eux. La mesure publiée était dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation/compilateur utilisé, etc... Une approche indépendante des facteurs matériels était nécessaire pour évaluer l'efficacité des algorithmes.

Un algorithme reçoit toujours une Donnée et produit un Résultat. Il est censé résoudre un Problème. Si l'on élimine de l'analyse de la complexité la question de la vitesse d'exécution de la machine et la qualité du code produit par le compilateur, il ne reste comme paramètre significatif que "la taille des données sur lesquelles il s'exécute".

Définition

On exprime le temps d'exécution en **nombre d'opérations élémentaires**. Ce nombre d'opérations est **exprimé en fonction de la taille des données en entrée**. Le temps d'exécution d'une opération élémentaire ne dépend pas de la taille de la donnée elle-même (par exemple, l'addition de deux entiers codés sur 32 bits prend un même temps, quels que soient les entiers considérés). Exemples d'opérations élémentaires : accès à une cellule mémoire, comparaison de valeurs, opérations arithmétiques (sur valeurs à codage de taille fixe), opérations sur des pointeurs, entrée_sortie, ...

Pour la taille de la donnée, on se limite à un ou plusieurs entiers liés au nombre d'éléments de la donnée. Par exemple, le nombre d'éléments dans une liste pour un algorithme de tri ou de recherche, le nombre de sommets et d'arcs dans un graphe orienté pour un algorithme de parcours du graphe.

On évalue le nombre d'opérations élémentaires en fonction de la taille de la donnée : si 'n' est la taille, on calcule une fonction $t(n)$. Le nombre d'opérations élémentaires peut varier substantiellement pour deux données de même taille. On retiendra deux critères :

1. analyse au sens du **plus mauvais cas** ou **cas le pire** : $t(n)$ est le temps d'exécution du plus mauvais cas et le maximum sur toutes les données de taille n. Par exemple, le tri par insertion simple avec des entiers présents en ordre décroissants. Autre exemple : si l'on recherche séquentiellement un élément dans une liste de taille **n**, il est possible que cet élément se trouve en première position, sa recherche nécessite alors une seule comparaison mais la complexité sera calculée dans le cas le plus défavorable possible c'est à dire le cas où l'élément n'appartient pas à la liste. Il faut alors parcourir toute la liste et donc effectuer **n** comparaisons.
2. analyse au sens **de la moyenne** : $t_m(n)$ est l'espérance sur l'ensemble des temps d'exécution muni d'une distribution des probabilités des temps d'exécution. L'analyse mathématique de la complexité moyenne est souvent très délicate. De plus, la signification de la distribution des probabilité par rapport à l'exécution réelle (sur un problème réel) est à considérer. On a donc introduit des 'notations asymptotiques'.

- idée 1 : évaluer l'algorithme sur des données de grande taille. Par exemple, lorsque n est 'grand', $3*n^3 + 2*n^2$ devient indiscernable de $3*n^3$
- idée 2 : on élimine les constantes multiplicatrices, car deux ordinateurs de puissances différentes diffèrent en temps d'exécution par une constante multiplicatrice. A partir de là, $3*n^3$ devient n^3 . L'algorithme est dit **en $O(n^3)$** .

L'idée de base est donc qu'un algorithme en $O(n^a)$ est "meilleur" qu'un algorithme en $O(n^b)$ si $a < b$.

Les limites de cette théorie :

- le coefficient multiplicateur est oublié : est-ce qu'en pratique $100*n^2$ est "meilleur" que $5*n^3$?
- le "cas le pire" peut, en pratique, n'apparaître que très rarement
- l'occupation mémoire, les problèmes d'entrées/sorties sont souvent occultés
- dans les algorithmes numériques, la précision et la stabilité sont primordiaux

Point fort : c'est une aide incomparable pour le développement d'algorithmes efficaces.

En résumé :

Complexité : définition

La complexité est le **nombre d'opérations élémentaires** nécessaires à un algorithme pour s'exécuter. Ce nombre d'opérations est **exprimé en fonction de la taille des données en entrée**. Ces opérations élémentaires sont généralement : additions/soustractions, multiplications/divisions, affectations, comparaisons, affichage, ... La complexité est exprimée en **$O(\text{taille_entrée})$** . La notation $O(n)$ cache toujours une constante multiplicative.

Familles de complexité

Il existe quatre grandes familles de complexité d'algorithmes :

- Les algorithmes linéaires dont la complexité est de l'ordre de **$O(n)$** ;
- les algorithmes sub-linéaires dont la complexité est inférieure à n , typiquement **$O(\log(n))$** (logarithme à base 2) ;
- les algorithmes sur-linéaires dont la complexité est supérieure à n , typiquement **$O(n^2)$** ou **$O(n \log(n))$** ;
- les algorithmes exponentiels dont la complexité est supérieure à tout polynôme en n , typiquement **$O(2^n)$** , ces derniers algorithmes sont inexploitable dès que la taille dépasse quelques unités.

Calculs de complexité

Il est souvent aisé de calculer la complexité d'un algorithme même si il est quelque fois indispensable de recourir à des calculs de sommes d'entiers ou d'autres calculs analytiques plus ou moins simples.

Il convient dans tous les cas de parfaitement compter le nombre de fois ou une boucle est exécutée. En effet, ce sont généralement les boucles (ou les appels récursifs en cas de programmation récursive) qui font la majorité des opérations. Voyons quelques exemples simples.

Complexité du calcul de la somme des éléments d'une liste

Le calcul de la somme des éléments d'un tableau de dimension **taille** a été étudié dans un précédent cours. Ce calcul se fait à l'aide d'une boucle Répéter pour i allant de 0 à ($\text{taille} - 1$), boucle qui s'exécute donc (taille) fois. Dans cette boucle une seule addition est effectuée à chaque itération, cela fait donc (taille) additions donc avec une complexité en **$O(\text{taille})$ additions**

Complexité de la recherche séquentielle dans une liste quelconque

La recherche simple ou séquentielle d'un élément donné dans une liste de dimension **taille** a été étudiée dans un précédent cours sur les algorithmes de recherche et fait appel à la boucle suivante :

Répéter

$i \leq i + 1$

tant_que ($i < \text{taille}$) **ET** ($T[i] \neq \text{Nombre}$)

Dans ce cas, le cas "le pire" est clairement le cas où le nombre n'appartient pas à la liste puisqu'il faudra alors parcourir la liste en entier avant de sortir sans avoir trouvé, la boucle est alors exécutée (**taille**) fois. A chaque passage dans la boucle sont exécutés :

- une addition ($i \leq i + 1$)
- un test ($i < \text{taille}$)
- et une comparaison ($T[i] \neq \text{Nombre}$) d'un élément de la liste avec l'élément cherché

Le travail le plus important dans ce cas est bien la comparaison puisque je rappelle que cet algorithme peut être utilisé sur une liste contenant n'importe quel type d'élément. Le nombre de comparaisons effectuées dans le "cas le pire" est donc de (**taille**) comparaisons donc une complexité en **O(taille) comparaisons**.

Complexité de la recherche dichotomique dans une liste TRIÉE

Comme précédemment, la recherche dichotomique d'un élément donné dans une liste TRIÉE de dimension **taille** a été étudiée dans le cours portant sur les algorithmes de recherche et fait appel à la boucle suivante :

Répéter

$\text{med} \leq (\text{debut} + \text{fin})/2$

Si $\text{tab}[\text{med}] = x$

alors $\text{place} \leq \text{med}$

sinon

Si $\text{tab}[\text{med}] > x$

alors $\text{fin} \leq \text{med} - 1$

sinon $\text{debut} \leq \text{med} + 1$

jusque ($\text{debut} > \text{fin}$) **ou** ($\text{place} \neq -1$)

Le cas le pire est, comme précédemment, le cas où l'élément n'appartient pas à la liste. A chaque passage dans cette boucle deux comparaisons d'éléments de la liste sont effectuées : (**Si** $\text{tab}[\text{med}] = x$) et (**Si** $\text{tab}[\text{med}] > x$).

Compter le nombre de fois où cette boucle s'exécute est un peu plus délicat. Il est certain que la dimension de la liste est divisée par 2 à chaque passage dans la boucle. Partant de là les mathématiciens peuvent calculer, qu'en moyenne, cette boucle va s'exécuter $\log_2(\text{taille})$ fois. La notation $\log_2(\dots)$ cache le logarithme à base 2 qui est défini par $\log_2(n) = \ln(n)/\ln(2)$ où \ln est, cette fois, le logarithme Népérien classique. Bref, retenons donc que le nombre de comparaisons est de $2 * \log_2(\text{taille})$ comparaisons donc une complexité en **O(log2(taille)) comparaisons**.

Comparaison des algorithmes de recherche séquentielle et dichotomique

Ces complexités sont bien jolies mais que représentent-elles au juste. Le petit tableau ci-dessous reprend quelques tailles de tableau et indique, pour chacune le nombre de comparaisons effectuées ainsi qu'une estimation du temps de calcul en faisant l'hypothèse qu'une comparaison se fait en 1/1000 de seconde (ce qui est déjà pas mal rapide !!). Place aux chiffres :

Comparaison des algorithmes de recherche séquentielle et dichotomique

Taille	Séquentielle liste quelconque $O(taille)$		Dichotomique liste TRIEE $O(\log_2(taille))$	
	Compar.	Temps	Compar.	Temps
10	10	0.01 s	4	0.004 s
100	100	0.1 s	7	0.007 s
1 000	1 000	1 s	10	0.010 s
10 000	10 000	10 s	14	0.014 s
100 000	100 000	1 mn 40 s	17	0.017 s
1 000 000	1 000 000	16 mn 40 s	20	0.020 s

Remarque : Vous pouvez remarquer que le calcul exact de complexité de la recherche dichotomique fait apparaître une constante multiplicative de 2 mais même en multipliant la dernière colonne par 2, 4 ou même 10 les différences sont éloquentes !!

7 Les tris simples

Introduction

En informatique, le tri ("sort" en anglais) est une opération consistant à réorganiser une collection d'objets selon un ordre déterminé. Les objets à trier font donc partie d'un ensemble muni d'une relation d'ordre (ordre total pour les Mathématiciens).

La méthode de tri est généralisable indépendamment de la relation d'ordre, la seule opération nécessaire étant de pouvoir comparer tout couple d'objets de la collection à trier.

Les principales caractéristiques d'un tri sont :

- la complexité algorithmique dans le pire des cas : le nombre d'opérations effectuées dans le pire des cas pour trier un ensemble de n éléments. On compte en général le nombre de comparaisons
- la complexité en moyenne : le nombre d'opérations effectuées en moyenne pour trier un ensemble de n éléments
- son caractère en place ou non : hormis la mémoire nécessaire pour stocker les éléments à trier, l'algorithme nécessite-t-il une quantité de mémoire supplémentaire dépendant du nombre d'éléments à trier ?
- son caractère stable ou non : lorsque deux éléments sont égaux pour la relation d'ordre, l'algorithme de tri conserve-t-il l'ordre dans lequel ces deux éléments se trouvaient avant son exécution.

On peut citer quelques algorithmes de tris parmi les plus connus :

- tri bulle
- tri sélection
- tri insertion simple ou dichotomique
- tri fusion (non abordé dans ce cours)
- tri par tas ou "heap sort" en anglais (non abordé dans ce cours)
- tri rapide ou "quick sort" en anglais (abordé un peu plus tard dans ce cours)

Les trois derniers types de tris sont optimaux, en ce sens qu'ils leur faut en moyenne de l'ordre de $n \cdot \log(n)$, le log étant le log à base 2, comparaisons pour trier un ensemble de n éléments. On sait prouver que, pour des tris basés sur la comparaison, on ne peut pas faire mieux en moyenne.

Pour certains types de données (entiers et chaînes de caractères de taille bornée), il existe cependant des algorithmes plus efficaces au niveau du temps d'exécution, comme le tri comptage ou le tri radix. Ces algorithmes n'utilisent pas la comparaison entre éléments (la borne $n \cdot \log(n)$ ne s'applique donc pas pour eux) mais nécessitent des hypothèses sur les objets à trier. Par exemple, le tri comptage et le tri radix s'appliquent à des entiers que l'on sait appartenir à l'ensemble $[1, m]$ avec comme hypothèse supplémentaire pour le tri radix que m soit de la forme 2^k .

Tri bulle (tri en place)

Le tri à bulle est une méthode de tri beaucoup critiquée à cause de sa lenteur d'exécution. En effet et pour un tri en ordre croissant, la méthode consiste à faire "monter", par permutations successives (comme une bulle), le plus grand élément du tableau en fin de liste en comparant les éléments successifs. C'est-à-dire qu'on va comparer le 1er et le 2e élément du tableau et échanger s'ils sont désordonnés l'un par rapport à l'autre. On recommence cette opération jusqu'à la fin du tableau. Ensuite, il ne reste plus qu'à renouveler cela jusqu'à l'avant-dernière place (le dernier étant forcément le plus grand et ainsi de suite jusqu'à obtenir une liste complètement triée.

Il est bien entendu possible, toujours pour un tri en ordre croissant, de partir de la fin et de faire "descendre" par permutations successives, le plus petit élément vers le début de liste.

Ce tri est basé sur une simple observation du comportement du tri à bulles : en effet, quand on fait un passage pour trier le maximum du tableau, on a tendance à déplacer les éléments les plus petits du tableau vers le début de celui-ci. Donc l'astuce consiste à alterner les sens de passage de notre bulle. On obtient un tri plus rapide que la bulle mais toujours médiocre en terme de temps d'exécution.

Nous allons gérer deux indices, l'un **i** qui nous permet de savoir à partir de quelle position la liste est déjà triée, l'autre **j** permettant de gérer les permutations d'un élément et de son voisin.

[illegible]

	2	7	3	12	5	8	3	15	
	i	j							
Premier passage terminé, le plus petit est bien placé, on recommence ...									
	2	7	3	12	5	8	3	15	
		i						j	
... fin du second passage ...									
	2	3	7	3	12	5	8	15	
		i	j						
Second passage terminé, les deux plus petits sont bien placés, on recommence ...									
	2	3	7	3	12	5	8	15	
			i					j	
... fin du troisième passage ...									
	2	3	3	7	5	12	8	15	
			i	j					
Troisième passage terminé, les trois plus petits sont bien placés, on recommence ...									
	2	3	3	7	5	12	8	15	
				i				j	
... fin du quatrième passage ...									
	2	3	3	5	7	8	12	15	
				i	j				
Quatrième passage terminé, les quatre plus petits sont bien placés, on recommence ...									
	2	3	3	5	7	8	12	15	
					i			j	
... fin du cinquième passage ...									
	2	3	3	5	7	8	12	15	
					i	j			
Cinquième passage terminé, les cinq plus petits sont bien placés, on recommence ...									

	2	3	3	5	7	8	12	15	
						i		j	
... fin du sixième passage ...									
	2	3	3	5	7	8	12	15	
						i	j		
Sixième passage terminé, les six plus petits sont bien placés, on recommence ...									
	2	3	3	5	7	8	12	15	
							i	j	
... fin du septième passage, ouf c'est terminé !									

Voilà un passage certes un peu fastidieux mais, je l'espère, instructif. Et oui, on s'en rend pas toujours compte mais cet algorithme est facile à écrire mais fort lent à l'exécution.

Algorithme "standard" du tri bulle

Après la longue exécution "manuelle" précédente, on doit être capable d'écrire simplement l'algorithme. Donnez-vous la peine de l'écrire sur un papier et pas seulement de le lire ci-dessous !

Tri bulle : algorithme "standard"

$O(\text{taille}^2)$ comparaisons

```

/* Tri en place de la liste T de longueur taille */
Répéter pour i allant de 0 à taille - 2
  Répéter pour j allant de taille - 1 à i + 1 par pas de -1
    si  $T[j] < T[j - 1]$ 
      /* permutation des éléments  $T[j]$  et  $T[j - 1]$  */
      alors
         $\text{tmp} \leftarrow T[j]$ 
         $T[j] \leftarrow T[j - 1]$ 
         $T[j - 1] \leftarrow \text{tmp}$ 
  fin_répéter
fin_répéter

```

Attention ! Attention remarquez bien l'utilisation d'une variable temporaire *tmp* afin de procéder à la permutation de deux valeurs dans une liste, c'est indispensable sous peine d'écraser l'une des deux valeurs à permuter !!

Attention ! La première boucle (sur la variable i) ne va que jusque $\text{taille} - 2$

Cette variante de l'algorithme ne compare pas deux éléments consécutifs $T[j]$ et $T[j - 1]$ mais $T[j]$ et $T[i]$ c'est à dire l'élément considéré, à tout moment de l'algorithme, comme le plus grand ($T[i]$) avec l'élément courant ($T[j]$) et les permute le cas échéant. Ecrivez-le !

$O(\text{taille}^2)$ comparaisons

Attention ! On peut faire encore mieux en remarquant que : si, lors d'un "passage" de l'algorithme, aucune permutation n'a été faite, c'est que le reste de la liste est déjà trié, on peut donc stopper l'algorithme ! Pourquoi ne pas essayer de programmer ce tri bulle "intelligent".

Exécutons ce second algorithme sur l'exemple précédent.

démarrage									
	7	3	12	5	8	2	15	3	
	i							j	
Compare 7 (T[i]) et 3 (T[j]) ==> permutation car 3 < 7									
	3	3	12	5	8	2	15	7	
	i						j		
Compare 3 et 15 ==> Rien à faire									
	3	3	12	5	8	2	15	7	
	i					j			
Compare 3 et 2 ==> Permutation car 2 < 3									

	2	3	12	5	8	3	15	7	
	i				j				
Compare 2 et 8 ==> Rien à faire									
	2	3	12	5	8	3	15	7	
	i			j					
Compare 2 et 5 ==> Rien à faire									
	2	3	12	5	8	3	15	7	
	i		j						
Compare 2 et 12 ==> Rien à faire									
	2	3	12	5	8	3	15	7	
	i	j							
Compare 2 et 3 ==> Rien à faire									
Premier passage terminé, le plus petit est bien placé, on recommence ...									
	2	3	12	5	8	3	15	7	
		i						j	
... etc ...									

Complexité du tri bulle

Analysons la structure des deux algorithmes ci-dessus qui sont très similaires dans leurs structures. La première boucle (portant sur **i**) s'exécute **taille** fois, la seconde boucle s'exécute elle **taille - 1** fois au premier passage ($i=0$), **taille - 2** fois au second passage ($i=1$), **taille - 3** fois au troisième passage ($i=2$), ainsi de suite... Le corps de l'algorithme effectue une comparaison à chaque étape. Au total nous aurons donc **taille - 1 + taille - 2 + taille - 3 + ... + 1 Comparaisons**. Les "plus mathématiciens d'entre vous" auront reconnu ici la somme des $taille-1$ premiers entiers qui vaut, comme chacun sait, **$taille * (taille + 1) / 2 = (1/2) * (taille^2 + taille)$ Comparaisons**. Les "moins mathématiciens d'entre vous" sont priés d'accepter ce résultat. Si l'on se souvient de la définition de la complexité avec une notation **O(...)** cachant une constante multiplicative, la complexité finale de cet algorithme est donc de **O(taille^2)**.

Le tri insertion (tri en copie ou en place)

Le tri par insertion est le tri le plus efficace sur des listes de petite taille. C'est pourquoi il est quelques fois utilisé par d'autres méthodes comme le Quick Sort.

Le principe de ce tri est très simple: c'est le tri que toute personne normale utilise quand elle a des dossiers (ou n'importe quoi d'autre) à classer. On prend un dossier et on le met à sa place parmi les dossiers déjà triés. Puis on recommence avec le dossier suivant.

Il faut noter que le tableau Ttrie reste trié à tout moment.

[illegible]

[illegible]

Trie	2	3	5	7	8	12	15		
			pos						
Recherche de la place d'insertion de 3 (pos)									
Trie	2	3	3	5	7	8	12	15	
"décalage" et insertion de 3									
Terminé Ttrie est bien triée et T n'a pas été modifiée									

Tri insertion simple en copie

Algorithme du tri insertion simple en copie

Ayant bien compris le principe de ce tri sur notre exemple, il ne reste plus qu'à écrire l'algorithme en pseudo-code, à vos crayons !

Tri insertion simple en copie

$O(\text{taille}^2)$ comparaisons

```

/* Tri en copie de la liste T de longueur taille */
/* avec construction de la liste Trie de longueur taille_Ttrie */
Ttrie[0] <== T[0]
taille_Ttrie <== 1
Répéter pour i allant de 1 à taille - 1
    /* recherche simple de la place d'insertion de l'élément d'indice i de T dans Ttrie*/
    place <== recherche_simple(T[i], Ttrie, taille_Ttrie)
    /* décalage des éléments de Ttrie depuis la fin jusque "place" */
    Répéter pour j allant de taille_Ttrie à place+1 par pas de -1
        Ttrie[j] <== Ttrie[j-1]
    fin_répéter
    /* insertion de l'élément et incrément de la taille */
    Ttrie[place] <== T[i]
    taille_Ttrie <== taille_Ttrie + 1
fin_répéter

```

Attention ! Dans cet algorithme, l'instruction *recherche_simple(T[i], Ttrie, taille_Ttrie)* fait appel à l'algorithme de Recherche simple de la place d'insertion de l'élément *T[i]* dans la liste *Ttrie* de longueur *taille_Ttrie* que nous avons vu dans une précédente feuille d'exercices.

Attention ! Regardez bien la boucle de "décalage" *Répéter pour j allant de taille_Ttrie à place+1*, il est impératif de commencer le décalage à partir de la fin et en "descendant" sous peine d'écraser tous les éléments se trouvant à la droite de l'élément d'indice *place* !!

Complexité du tri insertion simple

L'analyse de l'algorithme ci-dessus est relativement aisée à effectuer pour ce qui concerne le nombre de comparaisons. La première boucle (celle portant sur la variable *i*) est exécutée (*taille* - 1) fois, dans cette boucle la seule chose qui engendre des comparaisons est l'appel du sous-programme *recherche_simple*. Nous savons que cette recherche simple se fait avec une complexité de l'ordre de $O(\text{taille})$ comparaisons (voir section complexité). Cette recherche simple s'effectuant (*taille* - 1) fois, la complexité finale de l'algorithme est donc de **$O(\text{taille}^2)$ comparaisons**. Je rappelle que les calculs de complexité se font toujours pour "le cas le pire" et que la notation $O(\dots)$ cache toujours des constantes additives et multiplicatives. C'est l'ordre de grandeur qui importe avant tout !!!

Tri insertion dichotomique en copie

Algorithme du tri insertion dichotomique en copie

Le tri par insertion simple peut aisément être grandement amélioré. Pour cela, il suffit de remarquer que la liste *Ttrie* est toujours une liste triée et ce à n'importe quel moment lors de l'exécution de l'algorithme.

Partant de cette constatation, il semble évident que la recherche de la place d'insertion de l'élément dans la liste triée *Ttrie* pourra avantageusement se faire par une recherche dichotomique et non une simple recherche simple.

L'algorithme est donc strictement identique si ce n'est qu'il fait appel, cette fois, à la recherche dichotomique.

Tri insertion dichotomique en copie

$O(\text{taille} \cdot \log(\text{taille}))$ comparaisons

```
/* Tri en copie de la liste T de longueur taille */
/* avec construction de la liste Ttrie de longueur taille_Ttrie */
Ttrie[0] <== T[0]
taille_Ttrie <== 1
Répéter pour i allant de 1 à taille - 1
    /* recherche dichotomique de la place d'insertion de l'élément T[i] dans Ttrie*/
    place <== recherche_dichotomique(T[i], Ttrie, taille_Ttrie)
    /* décalage des éléments de Ttrie depuis la fin jusque "place" */
    Répéter pour j allant de taille_Ttrie à place+1 par pas de -1
        Ttrie[j] <== Ttrie[j-1]
    fin_répéter
    /* insertion de l'élément et incrément de la taille */
    Ttrie[place] <== T[i]
    taille_Ttrie <== taille_Ttrie + 1
fin_répéter
```

Complexité du tri insertion dichotomique

L'analyse de cet algorithme est strictement identique à l'analyse du précédent les deux algorithmes étant identiques en tout point si ce n'est dans la méthode de recherche de la place d'insertion de l'élément considéré. La recherche dichotomique se faisant en $O(\log(\text{taille}))$, la complexité finale du

T	7	3	12	5	8	15	3		
Trie	2								
Recherche de l'indice du minimum de T									
T	7	3	12	5	8	15	3		
		mini							
Insertion du minimum dans Ttrie et suppression dans T									
T	7	12	5	8	15	3			
Trie	2	3							
Recherche de l'indice du minimum de T									
T	7	12	5	8	15	3			
						mini			
Insertion du minimum dans Ttrie et suppression dans T									
T	7	12	5	8	15				
Trie	2	3	3						
Recherche de l'indice du minimum de T									
T	7	12	5	8	15				
			mini						
Insertion du minimum dans Ttrie et suppression dans T									
T	7	12	8	15					
Trie	2	3	3	5					
Recherche de l'indice du minimum de T									
T	7	12	8	15					
	mini								
Insertion du minimum dans Ttrie et suppression dans T									
T	12	8	15						
Trie	2	3	3	5	7				
Recherche de l'indice du minimum de T									

T	12	8	15						
		mini							
Insertion du minimum dans Ttrie et suppression dans T									
T	12	15							
Trie	2	3	3	5	7	8			
Recherche de l'indice du minimum de T									
T	12	15							
	mini								
Insertion du minimum dans Ttrie et suppression dans T									
T	15								
Trie	2	3	3	5	7	8	12		
Recherche de l'indice du minimum de T									
T	15								
	mini								
Insertion du minimum dans Ttrie et suppression dans T									
T									
Trie	2	3	3	5	7	8	12	15	
Terminé Ttrie est bien triée et T est vide !!!									

Algorithme du tri sélection en copie

Vous avez la méthode ainsi qu'un exemple d'exécution, il doit donc être aisé d'écrire cet algorithme sans regarder la solution ci-dessous !

Tri sélection en copie

$O(\text{taille}^2)$ comparaisons

```

/* Tri en copie de la liste T de longueur taille */
/* avec construction de la liste Trie de longueur taille_Ttrie */
taille_Ttrie <== 0
Répéter
/* Recherche de l'indice du minimum */
    ind_mini <== 0

```



```

Répéter pour i allant de 1 à taille - 1
    si T[i] < T[ind_mini]
    alors ind_mini <== i
fin_répéter
/* insertion de l'élément et incrément de la taille */
Ttrie[taille_Ttrie] <== T[ind_mini]
taille_Ttrie <== taille_Ttrie + 1
/* suppression de l'élément d'indice mini */
Répéter pour i allant de ind_mini à taille - 2
    T[i] <== T[i+1]
fin_répéter
taille <== taille - 1
tant que (taille > 0)

```

Tri sélection en copie avec utilisation de sous-programmes

$O(\text{taille}^2)$ comparaisons

```

/* Tri en copie de la liste T de longueur taille */
/* avec construction de la liste Ttrie de longueur taille_Ttrie */
taille_Ttrie <== 0
Répéter
/* Recherche de l'indice du minimum */
ind_mini <== Rech_indice_minimum(T, taille)
/* insertion de l'élément et incrément de la taille */
Ttrie[taille_Ttrie] <== T[ind_mini]
taille_Ttrie <== taille_Ttrie + 1
/* suppression de l'élément d'indice mini */
taille <== Suppr_place(T, taille, ind_mini)
tant que (taille > 0)

```

Attention ! Dans cet algorithme le tableau initial T est modifié !

Complexité du tri sélection

Il est plus efficace que le tri par insertion, pour trier des données où l'insertion ne se fait pas en temps constant ($O(1)$) et aussi efficace sinon. Mais, contrairement au tri par insertion, le nombre de comparaisons ne varie pas selon l'état d'ordre des données. (ie. même si les données sont presque triées le nombre de comparaison sera toujours du même ordre). Sa complexité finale est donc de $O(\text{taille}^2)$ comparaisons.

Tri sélection en place

L'algorithme ci-dessus présente, à l'évidence deux petits handicaps :

- après chaque insertion dans la liste resultat, il est nécessaire de supprimer l'élément inséré de la liste de départ;
- la liste de départ est plus que modifiée puisqu'elle est finalement complètement vidée.

Quitte à modifier la liste d'origine, autant faire ce tri en place ! Je vous invite donc à écrire cet algorithme de tri insertion en place et à vous reporter à la partie exercices pour la correction.