Partie 7
Techniques Rusées

« Informatique : alliance d'une science inexacte et d'une activité humaine faillible »
Luc Fayard

Une fois n'est pas coutume, ce chapitre n'a pas pour but de présenter un nouveau type de données, un nouveau jeu d'instructions, ou que sais-je encore.

Son propos est de détailler quelques techniques de programmation qui possèdent deux grands points communs :

- Leur connaissance est parfaitement indispensable
- Elles sont un rien finaudes

Et que valent quelques kilos d'aspirine, comparé à l'ineffable bonheur procuré par la compréhension suprême des arcanes de l'algorithmique ? Hein ?

1. Tri d'un tableau : le tri par insertion

Première de ces ruses de sioux, et par ailleurs tarte à la crème absolue du programmeur, donc : le tri de tableau.

Combien de fois au cours d'une carrière (brillante) de développeur a-t-on besoin de ranger des valeurs dans un ordre donné ? C'est inimaginable. Aussi, plutôt qu'avoir à réinventer à chaque fois la roue, le fusil à tirer dans les coins, le fil à couper le roquefort et la poudre à maquiller, vaut-il mieux avoir assimilé une ou deux techniques solidement éprouvées, même si elles paraissent un peu ardues au départ.

Il existe plusieurs stratégies possibles pour trier les éléments d'un tableau ; nous en verrons deux : le tri par insertion, et le tri à bulles. Champagne !

Commençons par le tri par insertion.

Admettons que le but de la manœuvre soit de trier un tableau de 12 éléments dans l'ordre croissant. La technique du tri par sélection est la suivante : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis en met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier. Par exemple, si l'on part de :

45	122	12	3	21	78	64	53	89	28	84	46

On commence par rechercher, parmi les 12 valeurs, quel est le plus petit élément , et où il se trouve. On l'identifie en quatrième position (c'est le nombre 3), et on l'échange alors avec le premier élément (le nombre 45). Le tableau devient ainsi :

3	122	12	**45**	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément, mais cette fois, seulement à partir du deuxième (puisque le premier est maintenant correct, on n'y touche plus). On le trouve en troisième position (c'est le nombre 12). On échange donc le deuxième avec le troisième :

3	**12**	**122**	45	21	78	64	53	89	28	84	46

On recommence à chercher le plus petit élément à partir du troisième (puisque les deux premiers sont maintenant bien placés), et on le place correctement, en l'échangeant, ce qui donnera in fine :

3	12	**21**	45	**122**	78	64	53	89	28	84	46

Et cetera, et cetera, jusqu'à l'avant dernier.

En bon français nous pourrions décrire le processus de la manière suivante :

- Boucle principale : prenons comme point de départ le premier élément, puis le second, etc., jusqu'à l'avant dernier.
- Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel et le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

Cela s'écrit :

```
Pour i ← 0 à 10  boucle principale : le point de départ se décale à chaque tour
     posmini ← i   on considère provisoirement que t(i) est le plus petit élément
     Pour j ← i + 1 à 11   on examine tous les éléments suivants
        Si t(j) < t(posmini) Alors
              posmini ← j
        Finsi
     j suivant
A cet endroit, on sait maintenant où est le plus petit élément. Il ne reste plus qu'à effectuer la
permutation.
     temp ← t(posmini)
     t(posmini) ← t(i)
     t(i) ← temp
i suivant
```

2. Un exemple de flag : la recherche dans un tableau

Nous allons maintenant nous intéresser au maniement habile d'une variable booléenne : la technique dite du « **flag** ».

Le flag, en anglais, est un petit drapeau, qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas. Et, aussitôt que cet événement a lieu, le petit drapeau se lève (la variable booléenne change de valeur). Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Tout ceci peut vous sembler un peu fumeux, mais cela devrait s'éclairer à l'aide d'un exemple extrêmement fréquent : la recherche de l'occurrence d'une valeur dans un tableau. On en profitera au passage pour corriger une erreur particulièrement fréquente chez le programmeur débutant.

Soit un tableau comportant, disons, 20 valeurs. L'on doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.

La première étape, évidente, consiste à écrire les instructions de lecture / écriture, et la boucle – car il y en a manifestement une – de parcours du tableau :

```
Variable Tab(20) en Numérique
Variable N en Numérique
…
      Ecrire "Entrez la valeur à rechercher"
      Lire N
      Pour i ← 0 à 19
            ???
      i suivant
```

Il nous reste à combler les points d'interrogation de la boucle Pour. Evidemment, il va falloir comparer N à chaque élément du tableau : si les deux valeurs sont égales, alors bingo, N fait partie du tableau. Cela va se traduire, bien entendu, par un Si … Alors … Sinon. Et voilà le programmeur raisonnant hâtivement qui se vautre en écrivant :

```
Variable Tab(20) en Numérique
Variable N en Numérique
…
      Ecrire "Entrez la valeur à rechercher"
      Lire N
      Pour i ← 0 à 19
            Si N = Tab(i) Alors
                  Ecrire "N fait partie du tableau"
            Sinon
                  Ecrire "N ne fait pas partie du tableau"
            FinSi
      i suivant
```

Et patatras, cet algorithme est une véritable catastrophe.

Il suffit d'ailleurs de le faire tourner mentalement pour s'en rendre compte. De deux choses l'une : ou bien la valeur N figure dans le tableau, ou bien elle n'y figure pas. Mais dans tous les cas, **l'algorithme ne doit produire qu'une seule réponse, quel que soit le nombre d'éléments que compte le tableau**. Or, l'algorithme ci-dessus envoie à l'écran autant de messages qu'il y a de valeurs dans le tableau, en l'occurrence pas moins de 20 !

Il y a donc une erreur manifeste de conception : **l'écriture du message ne peut se trouver à l'intérieur de la boucle : elle doit figurer à l'extérieur**. On sait si la valeur était dans le tableau ou non **uniquement lorsque le balayage du tableau est entièrement accompli.**

Nous réécrivons donc cet algorithme en plaçant le test après la boucle. Faute de mieux, on se contentera de faire dépendre pour le moment la réponse d'une variable booléenne que nous appellerons Trouvé.

```
Variable Tab(20) en Numérique
Variable N en Numérique
…
      Ecrire "Entrez la valeur à rechercher"
      Lire N
      Pour i ← 0 à 19
            ???
      i suivant
      Si Trouvé Alors
            Ecrire "N fait partie du tableau"
      Sinon
            Ecrire "N ne fait pas partie du tableau"
      FinSi
```

Il ne nous reste plus qu'à gérer la variable Trouvé. Ceci se fait en deux étapes.

- Un test figurant dans la boucle, indiquant lorsque la variable Trouvé doit devenir vraie (à savoir, lorsque la valeur N est rencontrée dans le tableau). Et attention : le test est asymétrique. Il ne comporte pas de "sinon". On reviendra là dessus dans un instant.
- Last, but not least, l'affectation par défaut de la variable Trouvé, dont la valeur de départ doit être évidemment Faux.

Au total, l'algorithme complet – et juste ! – donne :

```
Variable Tab(20) en Numérique
Variable N en Numérique
…
      Ecrire "Entrez la valeur à rechercher"
      Lire N
      Trouvé ← Faux
      Pour i ← 0 à 19
            Si N = Tab(i) Alors
                  Trouvé ← Vrai
            FinSi
      i suivant
      Si Trouvé Alors
            Ecrire "N fait partie du tableau"
      Sinon
            Ecrire "N ne fait pas partie du tableau"
      FinSi
```

Méditons un peu sur cette affaire.

La difficulté est de comprendre que dans une recherche, le problème ne se formule pas de la même manière selon qu'on le prend par un bout ou par un autre. On peut résumer l'affaire ainsi : **il suffit que N soit égal à une seule valeur de Tab pour qu'elle fasse partie du**

tableau. En revanche, il faut qu'elle soit différente de toutes les valeurs de Tab pour qu'elle n'en fasse pas partie.

Voilà la raison qui nous oblige à passer par une variable booléenne**, un « drapeau » qui peut se lever, mais ne jamais se rabaisser.** Et cette technique de flag (que nous pourrions élégamment surnommer « gestion asymétrique de variable booléenne ») doit être mise en œuvre chaque fois que l'on se trouve devant pareille situation.

Autrement dit, connaître ce type de raisonnement est indispensable, et savoir le reproduire à bon escient ne l'est pas moins.

3. Tri de tableau + flag = tri à bulles

Et maintenant, nous en arrivons à la formule magique : tri de tableau + flag = tri à bulles.

L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel **tout élément est plus petit que celui qui le suit**. Cette constatation percutante semble digne de M. de Lapalisse, un ancien voisin à moi. Mais elle est plus profonde – et plus utile - qu'elle n'en a l'air.

En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands « remontent » ainsi peu à peu vers les dernières places, ce qui explique la charmante dénomination de « tri à bulle ». Comme quoi l'algorithmique n'exclut pas un minimum syndical de sens poétique.

En quoi le tri à bulles implique-t-il l'utilisation d'un flag ? Eh bien, par ce qu'on ne sait jamais par avance combien de remontées de bulles on doit effectuer. En fait, tout ce qu'on peut dire, c'est qu'on devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés. Ceci est typiquement un cas de question « asymétrique » : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié.

Nous baptiserons le flag Yapermute, car cette variable booléenne va nous indiquer si nous venons ou non de procéder à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est signe que le tableau est trié, et donc qu'on peut arrêter la machine à bulles). La boucle principale sera alors :

```
TantQue Yapermute
    …
FinTantQue
```

Que va-t-on faire à l'intérieur de la boucle ? Prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, et procéder à un échange si nécessaire. C'est parti :

```
TantQue Yapermute
    Pour i ← 0 à 10
        Si t(i) > t(i+1) alors
            temp ← t(i)
            t(i) ← t(i+1)
            t(i+1) ← temp
        Finsi
```

```
    i suivant
FinTantQue
```

Mais il ne faut pas oublier un détail capital : la gestion de notre flag. L'idée, c'est que cette variable va nous signaler le fait qu'il y a eu au moins une permutation effectuée. Il faut donc :

- lui attribuer la valeur Vrai dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
- la remettre à Faux à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés),
- dernier point, il ne faut pas oublier de lancer la boucle principale, et pour cela de donner la valeur Vrai au flag au tout départ de l'algorithme.

La solution complète donne donc :

```
Yapermut ← Vrai
TantQue Yapermut
    Yapermut ← Faux
    Pour i ← 0 à 10
        Si t(i) > t(i+1) alors
            temp ← t(i)
            t(i) ← t(i+1)
            t(i+1) ← temp
            Yapermut ← Vrai
        Finsi
    i suivant
FinTantQue
```

Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie de l'arsenal du programmeur bien armé.

4. La recherche dichotomique

Je ne sais pas si on progresse vraiment en algorithmique, mais en tout cas, qu'est-ce qu'on apprend comme vocabulaire !

Blague dans le coin, nous allons terminer ce chapitre migraineux par une technique célèbre de recherche, qui révèle toute son utilité lorsque le nombre d'éléments est très élevé. Par exemple, imaginons que nous ayons un programme qui doive vérifier si un mot existe dans le dictionnaire. Nous pouvons supposer que le dictionnaire a été préalablement entré dans un tableau (à raison d'un mot par emplacement). Ceci peut nous mener à, disons à la louche, 40 000 mots.

Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier. Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et

même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20 000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, un être humain qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots sont triés.

Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve pile poil au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans la première moitié du dico. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié.

A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne en termes de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.

Au départ, on cherche le mot parmi 40 000.
Après le test n°1, on ne le cherche plus que parmi 20 000.
Après le test n°2, on ne le cherche plus que parmi 10 000.
Après le test n°3, on ne le cherche plus que parmi 5 000.
Etc.
Après le test n°15, on ne le cherche plus que parmi 2.
Après le test n°16, on ne le cherche plus que parmi 1.
Et là, on sait que le mot n'existe pas. Moralité : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment ! Il n'y a pas photo sur l'écart de performances entre la technique barbare et la technique futée. Attention, toutefois, même si c'est évident, je le répète avec force : la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.

Eh bien maintenant que je vous ai expliqué comment faire, vous n'avez plus qu'à traduire !

Au risque de me répéter, la compréhension et la maîtrise du principe du flag font partie de l’arsenal du programmeur bien armé.

Exercice 7.1

Exercice 7.2

Exercice 7.3

Exercice 7.4

Exercice 7.5