Table des matières

Algorithme

Algorithme[1]

n. m., XIII^{e}s., voir § III.

I. (a) **Un algorithme est une succession de manœuvres à accomplir toujours dans le même ordre et de la même façon, manœuvres qui sont en nombre fini et s'appliquent à un nombre fini de données.**

Par exemple, on connaît depuis l'enfance des algorithmes de calcul, ceux qui permettent de trouver ce que vaut la somme, le produit, la différence ou le quotient de deux nombres quand ils s'écrivent avec plus d'un chiffre. Il ne s'agit pas d'autre chose quand on entend dire « je ne sais plus faire une division », par exemple celle de 793 par 32 ; en fait, c'est l'algorithme de calcul du quotient qu'on a plus en mémoire, c'est-à-dire le « j'ai deux chiffres au diviseur, j'en prends deux au dividende, je dis en 79 combien de fois 32 ou en 7 combien de fois 3, il y va deux fois, etc »

(b) Les livres donnent volontiers comme exemples d'algorithmes ceux qui consistent à trouver le nouveau prix d'un objet s'il y a au moment de l'achat une baisse ou une hausse de, mettons, 10 % ; si le prix initial est x, la hausse ou la baisse est les 10/100 de x, soit $0,1x$; le nouveau prix est donc :

- en cas de baisse : $x - 0,1x = 0,9x$,
- en cas de hausse : $x + 0,1x = 1,1x$.

(c) Un algorithme est généralement répétitif ; c'est le cas si on place une somme d'argent à la banque à un taux de 4 % ; au bout d'une année, elle devient les 104/100 de ce qu'elle était, au bout de deux années les 104/100 des 104/100 de ce qu'elle était, etc.

En désignant cette somme par S, on a donc, au bout de n années, une somme :

$$S_n = \underbrace{(104/100)(104/100)\dots(104/100)}_{n\text{fois}} S = 1,04^n S$$

II. On désigne par algorithme un procédé automatique que l'on peut confier à un ordinateur et qu'il répétera autant de fois qu'il le faudra pour arriver au résultat. Imaginons qu'on l'ait programmé pour la suite d'opérations suivantes : à partir d'un entier naturel n quelconque, si n est pair le diviser par 2 ; s'il est impair, prendre son triple et ajouter 1, c'est-à-dire fabriquer $3n+1$; dans chaque cas, recommencer avec le nouveau nombre obtenu. Essayons avec quelques nombres :

Nombre de départ												
16	8	4	2	1								
17	52	26	13	40	20	10	5	16	8	4	2	1
18	9	28	14	7	22	11	34	17	...			
19	58	29	88	44	22	11	34	17	...			

Il est clair qu'en arrivant à 1, le processus va 'se boucler' sur lui-même, puisqu'on aura la suite 1, 4, 2, 1, 4, 2, 1, etc. On voit que c'est ce qui se produit pour 16 et 17 ; or, pour 18 et 19, on retombe sur 17, donc on arrivera aussi à 1.

La ∗**conjecture** qui s'établit à partir de plusieurs tentatives qui, toutes, amènent à 1 est que, quel que soit le nombre de départ et le nombre d'étapes, cet algorithme produira toujours 1 ; mais elle n'est, malgré son apparente simplicité, toujours pas démontrée aujourd'hui. Ce 'beau' problème s'appelle "le problème de Collatz", du nom du professeur de Hambourg qui l'a lancé [G_{17}].

III. *Algorithme* s'est d'abord dit *algorisme*, du bas latin *algorismus*, déformation d'après le mot grec *arithmos*, "nombre", du nom propre Al-∗Khwarizmi (a).

1. *Dictionnaire de mathématiques élémentaires* Stella Baruk, Seuil, p. 77

Sept épatant[2]

Le véritable problème fut posé quand le père Mathieu revint de la foire, poussant devant lui les vingt-huit moutons acquis le matin même. Jusqu'alors, les opérations s'étaient déroulées sans aucune difficulté. Mais il fallait maintenant répartir ces vingt-huit bêtes dans les sept bergeries que comportait la ferme, et ça, croyez-en le père Mathieu, ce n'était pas une mince affaire.

Il appela Toine, son fils aîné :

– Toine, lui dit-il, tu vas me prendre ces vingt-hui bêtes et me les installer dans nos sept bergeries. T'en mettras le même nombre dans chacune.

– Et ça en fait combien donc dans chaque ? Questionna le Toine.

– Décidément, Toine, t'es pas bien futé. Apprends que, pour faire un partage, on pose une division. Tiens prends une feuille de papier, je vas te montrer.

Et le père Mathieu expliqua au Toine les subtilités de l'opérations :

28 | 7	– Vingt-huit divisé par sept : En 8 combien de fois 7 ? Il y va une fois. Une fois sept fait 7 ;
21 | 13	ôté de 8 il reste 1. J'abaisse le 2. En 21 combien de fois 7 ? Il y va 3 fois. 3 fois 7 font 21 ; ôté
0 |	de 21, il reste 0. Tu mettras donc 13 moutons dans chaque bergeries.

– Bien, père, fit le Toine, convaincu par la science.

Il partit incontinent, pour procéder à la répartition. Une heure plus tard, Mathieu le vit revenir tout piteux :

– J'y arrive pas, père. Il doit y avoir une erreur.

– Écoute-moi bien, lui dit son père. Y a pas d'erreur possible. D'ailleurs pour te le prouver, on va procéder autrement. Je t'ai dit 13 moutons dans chaque bergeries. Si on multiplie 13 par 7, on doit retrouver les 28 têtes. Allons-y :

13	Treize multiplié par sept : 7 fois 3 font 21 ; et 7 fois 1 fait 7. Tu vois que 21 et 7, ça fait bien
× 7	28.
21	D'ailleurs, pour être plus sûr, on va faire la preuve par neuf :
7	3 et 1 font 4. Je pose 4 en haut et j'écris 7 en dessous. 7 fois 4 font 28. 8 et 2 font 10. J'écris
28	1 à gauche. Maintenant le résultat : 8 et 2 font 10. J'écris 1 à droite. Tu vois bien que c'est
4	juste. Allez, va-t-en me mettre treize bêtes dans chaque bergerie.
1 × 1	(Ici, normalement, Mathieu aurait dû s'inquiéter, puisque 7 fois 13, comme 7 fois 4 font
7	également 28. Mais s'il fallait encore s'attacher à tant de menus détails, on n'avancerait jamais. On continua donc).

C'est un Toine effondré qui revint une heure plus tard.

– J'y arrive toujours pas. Y a sûrement quelque chose qui ne va pas dans les comptes.

– Y a surtout qu't'es pas bien malin, fils, dit le père Mathieu. La division, la multiplication, c'est trop fort pour toi. L'addition, ça doit aller mieux.

13	J'écris 13, sept fois de suite, et j'additionne : 3 et 3, 6 ; et 3, 9 ; et 3, 12 ; et 3, 15 ; et 3, 18 ; et 3, 21 ;
13	et 1, 22 ; et 1, 23 ; 24 ; 25 ; 26 ; 27 ; 28 .
13	Es-tu convaincu, cette fois ? Allez, va.
13	Et le Toine repartit encore une fois, loger les maudites bêtes.
13	Et fin de soirée, il revint triomphant.
13	– Ça y est, père, tous les moutons sont rentrés
28	– Comment que t'as fait ?

– Je les ai fait rentrer un par un en faisant le tour des bergeries. Et pour être tout à fait sûr, quand ils ont été placés, moi aussi, j'ai fait mes comptes : j'ai compté les pattes ; j'ai trouvé 16 pattes dans chaque bergeries.

– Attends voir, dit le Père Mathieu. Faut pas s'emballer. Étant donné qu'un mouton a 4 pattes, si je divise 16 par 4, je saurai combien tu as mis de bêtes dans chacune.

16 | 4	Et la nouvelle division fut posée : seize divisé par quatre : en 6 combien de fois 4 ? Il y va une
12 | 13	fois. Une fois 4 fait 4 ; ôté de 6, il reste 2. J'abaisse mon 1. En 12 combien de fois 4 ? Il y va
0 |	3 fois. 3 fois 4 font 12 ; ôté de 12, il reste 0.

2. A. Thuizat, in Le petit Archimède N^o 3, Association pour le développement de la culture scientifique.

Les variables

2.1 Les scalaires

Il ne faut pas confondre l'*identificateur* d'une variable et la *valeur* de la variable. La valeur d'une variable est conservée dans une zone mémoire.

Si SOMME vaut `10`, la notation algorithmique

```
SOMME = SOMME + 5
```

affecte[3] à la zone SOMME la *valeur précédente* de SOMME + 5 (soit `15`) .

Autre exemple :

```
SOMME = SOMME + SOMME
```

fait passer la zone valeur SOMME de `10` à `20`.

Les termes de *left value* et *right value* désignent respectivement la zone mémoire et la valeur qu'elle recelle.

Un ordinateur manipule des *représentations* de valeurs, qui sont des configurations de bits, d'octets ou de mots de la mémoire. Comme les représentations physiques varient selon les objets, on est conduit à spécifier leurs types.

Type de variables élémentaires

1. Les **entiers** sont en nombre fini dans l'ordinateur,
   donc certaines opérations peuvent creer un débordement, *ie* fournir un résultat hors des limites de l'intervalle. Le langage C ne teste jamais le résultat.
2. L'ensemble des **réels** est discontinu,
   la norme IEEE 754 décrit les nombres à virgule flottante.
3. L'ensemble des **caractères** n'est pas le même selon le pays,
   l'ensemble des caractères est un sous-ensemble des entiers.

2.2 Tableaux

Il est fréquent de manipuler des données de même type formant une collection que l'on nomme *tableau*. Chaque valeur est désignée par le *nom* du tableau et d'un ou plusieurs *indice(s)* ; ce nom peut être manipulé comme celui d'une variable. Un tableau à une seule dimension est souvent appelé *vecteur*.

2.3 Les variables composites

Il est parfois nécessaire de manipuler des ensembles de données formant un tout ; ces ensembles sont nommés agrégats, enregistrements ou structures[4].

2.4 Pointeurs

Toutes ces variables occupent un emplacement de la mémoire et le système pour les retrouver en conserve l'adresse ou *pointeur*.

3. cf. page 8
4. [**??**]

L'affectation[5]

Affecter, affecté, e

affecter : V. ; (**e**$_1$) XIV^{e}S., du latin *affectare*, "feindre" ;
(**e**$_2$) 1551, du latin *affectare*, au sens de "disposer, attribuer", comme dans *affectation* ;
(**e**$_3$) XVIII^{e}s., de *affectare*, au sens de "toucher, atteindre" ; *affecté, e :* adj. et part. passé de *affecter*.

I. (a) D'après (**e**$_1$), "feindre" : *une gentillesse affectée.*

(b) D'après (**e**$_2$), "attribuer" : *de nouveaux crédits vont être affectés à ce poste.*

II. (a) *Affecter* (**e**$_1$) a le sens qu'il a dans la langue courante quand on dit : *la courbe affecte la forme d'un huit.*

(b) *Affecté de...,* au sens de "pourvu de...", semble n'être utilisé qu'en mathématiques et ne figure pas dans les dictionnaires de langue courante, dans lesquels on rencontre surtout *désaffecté*, du verbe *désaffecter* au sens de "ne plus affecter à" : *désaffecter les sommes destinées à construire un stade* ; d'où, par extension, *désaffecté*, qui ne remplit plus l'emploi qui lui étatit destiné : *une église désaffectée.*

D'utilisation récente, *affecté de...* veut dire "auquel est attribué..." ou "auquel on attribue" ; par exemple, pour désigner les termes successifs d'une suite, on *affecte* une même lettre d'*indices indiquant le rang du terme de la suite :

$$u_0, u_1, u_2, \ldots, u_p, \ldots, u_{n-1}, u_n, \ldots$$

(c) D'après (**e**$_3$), "toucher, atteindre" : *affecté(e)* a le sens qu'il a dans la langue courante quand on dit "la valeur de vérité de l'équation n'est pas *affectée* si on multiplie ses deux membres par 3", ou encore "une quantité n'est pas *affectée* par un changement d'unité".

III. En informatique l'*affectation* associe une variable (un identifiant) avec une valeur (résultat de l'évaluation d'une expression).

`Variable = expression`[6]

cette opération n'est pas symétrique !

Lorsque l'on fait une affectation, on poursuit un objectif : se rapprocher d'un résultat escompté ; la nouvelle valeur de la variable modifie l'état du programme, c'est à dire les relations entre les variables. Ex :

	c'est à dire :
`/*` $x > n$ `*/`	**si** $x > n$
$x = x + 1$	**et si** l'on exécute l'affectation $x = x + 1$,
`/*` $x > n + 1$ `*/`	**alors** il s'ensuit $x > n + 1$.

La deuxième assertion représente l'état du programme après l'exécution de $x = x + 1$.
La première assertion est conservée à ceci près qu'il faut substituer à x sa nouvelle valeur et rétablir l'inégalité.

Un autre exemple :

	En effet :
`/*` $0 < x < 1$ `*/`	`si` $x > y + 1$,
$x = \frac{1}{x} + y$	`alors` x `valant` $\frac{1}{x} + y$
`/*` $x > y + 1$`*/`	$\Rightarrow \frac{1}{x} + y > y + 1$
	$0 < x < 1 \Rightarrow \frac{1}{x} > 1 \Rightarrow \frac{1}{x} + y > y + 1$

Notez qu'on raisonne à *reculons* (de l'affirmation finale vers l'affirmation initiale), ce qui correspond au cas naturel où l'on voudra atteindre un certain état du programme, état pour lequel une certaine affirmation est vérifiée, et où l'on construira les hypothèses initiales et les instructions successives en fonction de cette conclusion.

5. *Dictionnaire de mathématiques élémentaires* Stella Baruk, Seuil, p. 50
6. Faire très attention à différencier = (l'affectation) et == (le test d'égalité).

L'affectation - Un exemple (dé)trompeur.

Considérons l'algorithme suivant :

`Saisir`(x) ; `Saisir`(y) ;
$x = x + y$;
$y = x - y$;
$x = x - y$;
`Afficher`(x) ; `Afficher`(y) ;

Par nature, une instruction d'**affectation** réalise une transformation : elle change l'**état** d'une variable. Pour expliquer l'effet de la séquence

$$x = x + y\,; \qquad y = x - y\,; \qquad x = x - y\,;$$

il faut décrire la suite engendrée par les 3 instructions. Nous noterons $/*\ \ldots\ */$ une description d'état, représentée par une relation entre les variables du programmes et des constantes. Soit donc a et b les valeurs initiales des variables x et y

$$/* \, x == a \text{ et } y == b \, */ \quad x = x + y\,;$$

Si l'on exécute l'instruction $x = x + y$ seul x est modifié

$$/* \, x == a \text{ et } y == b \, */ \quad x = x + y\,; \qquad /* \, x == a + b \text{ et } y == b \, */$$

L'instruction suivante modifie y

$$/* \, x == a + b \text{ et } y == b \, */ \quad y = x - y\,; \qquad /* \, x == a + b \text{ et } y == (a + b) - b == a \, */$$

La dernière instruction modifie x

$$/* \, x == a + b \text{ et } y == a \, */ \quad x = x - y\,; \qquad /* \, x == (a + b) - a == b \text{ et } y == a \, */$$

Ainsi donc les valeurs finales de x et y sont la permutation des valeurs initiales.

On appelle "assertion" l'affirmation d'une relation vraie entre les variables du programme en un point donné. Dire comment une instruction modifie l'assertion qui la précède (pré-assertion) pour donner celle qui la suit (post-assertion), c'est définir la *sémantique* de cette instruction.

Réécrivons les 3 instructions avec leurs assertions

$$\begin{array}{llll} /* \, x == a & \text{et} & y == b \, */ & x = x + y\,; \\ /* \, x == a + b & \text{et} & y == b \, */ & y = x - y\,; \\ /* \, x == a + b & \text{et} & y == a \, */ & x = x - y\,; \\ /* \, x == b & \text{et} & y == a \, */ & \end{array}$$

Supposons que les nombres a et b soient des réels [7], et que b soit très petit devant a. les calculs étant faits avec un nombre constant de chiffres significatifs, à la précision des calculs b est négligeable devant a et l'addition de b à a ne modifie pas a

$$\begin{array}{ll} /* \, x == a \text{ et } y == b \, */ & x = x + y\,; \\ /* \, x == a \text{ et } y == b \, */ & y = x - y\,; \end{array}$$

De même, retrancher b de a ne change pas a

$$\begin{array}{ll} /* \, x == a \text{ et } y == a \, */ & x = x - y\,; \\ /* \, x == 0 \text{ et } y == a \, */ & \end{array}$$

L'échange des valeurs ne s'est pas fait. Il y a 0 en x et a en y. Ainsi le mécanisme des assertions peut être un mécanisme très fin décrivant même la façon dont les calculs sont exécutés dans l'ordinateur. Il permet une interprétation très précise de l'effet d'une séquence d'instructions. C'est lui qui nous permettra de donner un sens à un programme.

7. Cf page **??**, l'ensemble des entiers est fini.

Complexité

Le temps d'exécution et la place mémoire requise caractérisent la complexité[8] d'un programme. Sous UNIX, on mesure le temps d'excécution d'un programme au moyen de la commande `time`. Il tombe sous le sens qu'un programme mettra d'autant plus de temps à s'achever qu'il aura de données à traiter. Pour un même résultat, deux algorithmes équivalents ne verront sans doute pas leur temps d'exécution croître dans les mêmes proportions en fonction des données à traiter. Exemple : Calculs du plus grand diviseur de n.

1ère méthode	**2ème méthode**

`i = n - 1`
while `(0 != (n % i))`
`  /* pgd ≤ i const. de boucle */`
`  i = i - 1`
`pgd = i`
	`i = 2`
	while `(i <` $\sqrt{n}$ `et 0 != (n % i))`
	`  /* ppd > i constante de boucle */`
	`  i = i + 1`
	`/* pgcd =` SI `(0 ==` n `modulo` i`)` ALORS $\frac{n}{i}$ SINON `1 */`
	`pgd = (0 == (n % i))? n/i: 1`

– **Complexité en temps**

La complexité d'un algorithme se mesure essentiellement en calculant le nombre d'opérations élémentaires pour traiter une donnée de taille n. Les opérations élémentaires considérées sont

- le nombre de comparaisons (algorithmes de recherche)
- le nombre d'affectations (algorithmes de tris)
- Le nombre d'opérations ($+$, $*$) réalisées par l'algorithme (calculs sur les matrices ou les polynômes).

Le coût d'un algorithme A pour une donnée D est le nombre d'opérations élémentaires nécessaires au traitement de la donnée D et est noté $0_A(D)$

– **Complexité dans le pire des cas**,
exemple : recherche d'un nombre dans un tableau, alors qu'il n'y est pas :

$$\mathrm{Max}_{(n)} = \max\{O_A(d_1), d_i \in D_i\}$$

– **Complexité dans le meilleur des cas**,
ex : rechercher d'un nombre dans un tableau, alors qi'il est en première position :

$$\mathrm{Min}_{(n)} = \min\{O_A(d_1), d_i \in D_i\}$$

– **Complexité en moyenne**

$$\mathrm{Moy}_A(n) = \sum_{d \subset D_n} p(d) O_A(d)$$

où $p(d)$ est la probabilité d'avoir en entrée la donnée d parmi toutes les données de taille n.

Si toutes les données sont équiprobables, alors on a,

$$\mathrm{Moy}_A = \frac{1}{|D_i|} \sum_{d \in D_n} O_A(d)$$

– **Espace mémoire**

Il est quelquefois nécessaire d'étudier la complexité en mémoire lorsque l'algorithme requiert de la mémoire supplémentaire[9] (tableau auxiliaire de même taille que le tableau donné en entrée par exemple).

8. La *simplicité* d'un algorithme n'est donc pas le contraire de la complexité.
9. Cf page 12, un exemple de choix d'algorithme selon la place.

Exemple de calcul de complexité d'un algorithme : apparition d'un nombre dans un tableau.

Soit T un tableau de taille N contenant des nombres entiers de 1 à k. Soit a un entier entre 1 et k.

La fonction suivante renvoie 1 lorsque l'un des éléments du tableau est égal à a, et 0 sinon.

```
int Trouve (int T[], int n, int a)
{
  int i = 0 ;
  while (i < n)
    if (T[i] == a )
      return i ;
  /* i == n ; le tableau est parcouru */
  return 0 ;
}
```

Cas le pire : N (le tableau ne contient pas a)
Cas le meilleur : 1 (le premier élém. du tableau est a)
Complexité moyenne : Si les nombres entiers de 1 à k apparaissent de manière équiprobable, on peut montrer que le cout moyen de l'algorithme est $N = k(1-(1-1/k))$.

De fait les cas où l'on peut explicitement calculer la complexité en moyenne sont rares. Cette étude est un domaine à part entière de l'algorithmique

Analyse asymptotique

- Analyse du temps d'un calcul d'un programme
  - Valeur approchée Le temps de calcul d'un programme dépend trop de la vitesse de l'ordinateur et du compilateur utilisés. On peut donc calculer les performances d'un algorithme à facteur multiplicatif constant. Des programmes de complexités n, $2n$ ou $3n$ sont quasiment équivalents.
  - Règle des 90/10 : 90% du temps de calcul d'un programme est réalisé dans 10% du code. Inutile donc d'essayer de perdre trop de temps à optimiser les 90% qui ne prennent que 10% du temps. Autant se consacrer à ce qui est le plus pénalisant.
  - Exemple : comparaisons d'un algorithme A de complexité $100n$ et d'un algorithme B de complexité $2n^2$ (cf Aho-Ullman)
- Notations θ et O :
  - Définitions. On dit que $f = \theta(g)$ lorsqu'il existe deux constantes c_1 et c_2 positives (f et g sont égalements supposées à valeurs positives) telles que, pour n assez grand

    $$c_1.g(n) < f(n) < c_2.g(n)$$

    Remarquons que cette relation est réflexive.
- On dit que $f = O(g)$ lorsqu'il existe une constante c positive telles que, pour n assez grand

  $$f(n) < c.g(n)$$

  c'est dire que f est bornée par g à un facteur multiplicatif près.

  Cette relation n'est pas réflexive.
- Propriétés.

  Un polynôme est de l'ordre de son degré. On distingues les fonctions linéaires (en $O(n)$), les fonctions quadratiques (en $O(n^2)$) et les fonctions cubiques (en $O(n^3)$).

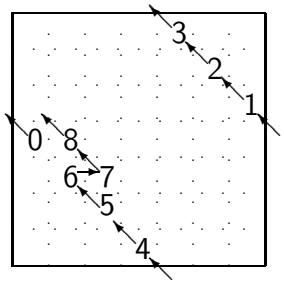
  Les fonctions d'orde exponentiell sont les fonctions en $O(a^n)$ ou $a > 1$.

  Les fonctions d'ordre logarithmique sont les fonctions en $O(\log(n))$ (Remarquons que peu importe la base du logarithme).
- Une classe intéressante d'algorithme est en $n\log(n)$. Comparaison de $n\log(n)$ et de n^2. [Image]
- Comparaison des asymptotiques classiques.
  - Rappellons que $log(n)^i << n^k << a^n (f << g$lorsque limite$(g/f) = 0)$.
  - Fractions rationnelles
  - Factorielle : formule de Stirling [Image]
  - Nombres de Fibonacci
- Calcul de complexité dans les structures de contrôle.
  - Les intructions élémentaires (affectations, comparaisons) sont et temps constant, soit en O(1).
  - Tests : O(if A then B else C fi) = O(A)+max(O(B),O(C))
  - Boucles O(for i from 1 to n do Ai od) = somme(O(Ai)) Lorsque O(Ai) est constant à O(A), on a O(for i from 1 to n do A od) = nO(A)
  - Cas particuliers : boucles imbriquées
    - O(for i from 1 to n do for j from 1 to n do A od
    - Le fait que la borne sup de la boucle intérieure soit i plutot que n ne change rien : O(for i from 1 to n do for j from 1 to i do A od od) = (1+2+...+n). O(A).

Carrés magiques[10]

REMPLIR un carré magique : c'est disposer les nombres 0 à p dans un carré de $n \times n$ cases, de telle sorte que la somme des nombres dans chaque ligne, chaque colonne et sur les deux diagonales soit la même. Dès que le carré est supérieur à 5×5 cases, il devient nécessaire de disposer d'une méthode. Nous allons considérer un exemple.

Remplissons une grille de 7×7.



Nous plaçons tout d'abord 0 dans la case du milieu du bord gauche. En suivant la petite flèche nous sortirions du carré. Il faut donc placer 1 dans la case correspondante sur le bord opposé, et ainsi de suite jusqu'à 6. Il n'est plus possible de continuer puisque la case suivante est occupée. Il suffit de placer 7 à droite de 6 puis de reprendre la progression.
Essayez de terminer seul ; puis, vous pourrez vérifier que vous avez réalisé un *carré magique* puisque la somme des nombres placés dans chaque ligne, chaque colonne et chaque diagonale est constante (168).

L'inconvénient majeure de cette solution c'est l'espace mémoire occupé puisqu'il faut se doter d'un tableau de $n \times n$ cases. Un autre algorithme consiste à produire la valeur de chaque case dans l'ordre de lecture, de gauche à droite et de bas en haut.

Supposons, pour simplifier, un *carré magique* de 5×5 déjà réalisé et étudions sa composition :

14	15	21	2	8
7	13	19	20	1
0	6	12	18	24
23	4	5	11	17
16	22	3	9	10

$2n+4$	$3n+0$	$4n+1$	$0n+2$	$n+3$
$n+2$	$2n+3$	$3n+4$	$4n+0$	$0n+1$
$0n+0$	$n+1$	$2n+2$	$3n+3$	$4n+4$
$4n+3$	$0n+4$	$n+0$	$2n+1$	$3n+2$
$3n+1$	$4n+2$	$0n+3$	$n+4$	$2n+0$

En exprimant le contenu de chaque case par rapport au côté du carré n la progression d'une case à la suivante est simple : les coefficients progressent régulièrement de 0 à $n-1$.

$X_{(ij)+1} = (a_{ij} + 1 \times n) + b_{ij} + 1$ ou a et $b \in \{0, 1, 2 \ldots n-1\}$

Lorsqu'une ligne est remplie, on passe à la première case de la ligne suivante en retranchant 1 au contenue de la dernière case de la ligne qui vient de s'achever. Enfin le premier terme du *carré magique* est obtenue par : $x_0 = \frac{n-1}{2} \times n + n - 1$

D'où l'algorithme :

```
SAISIR le cote du carre (n)
b = n - 1
a = (n - 1) / 2
i = j = 1

POUR CHAQUE ligne
    POUR CHAQUE case
        IMPRIMER (a * n) + b
                                .../...
```

```
        SI ce n'est pas la derniere case
        ALORS a = a + 1 ; a = a modulo n
              b = b + 1 ; b = b modulo n
        FIN DE SI
    FIN DE POUR

    b = b - 1

FIN DE POUR CHAQUE
```

10. *in PPC* Dhénin Jean-Jacques

Schémas linéaires

5.1 Déroulement linéaire

La forme la plus simple de l'algorithme (cf p. 5) est le déroulement linéaire (enchaînement).

- Le déroulement linéaire ne comporte aucune prise de décision. Les lignes de programme qui s'y trouvent seront toujours exécutées dans le même ordre.
- Le déroulement linéaire est le mode implicite d'exécution d'un programme. Sans instruction qui suspend ou modifie le déroulement linéaire, l'ordinateur *passe* toujours à l'instruction suivante après l'exécution de l'instruction courante.
- La caractéristique d'un algorithme linéaire est de n'utiliser que l'enchaînement séquentiel d'actions dont la plus simple est l'affectation [11]).

5.2 Propriétés de l'enchaînement

Une succession d'instructions poursuit un objectif : se rapprocher d'un résultat prévu ; les relations entre les variables sont modifiées. Bien entendu, l'état final est le résultat de la succéssion des états intermédiaires.

Exemple :

Supposons les nombres x et y, avec
/* $0 < x < 1$ */

Alors, si l'on exécute
$x = \frac{1}{x} + y$;
$x = x + 1$;

l'on aura /* $x > y + 2$ */

En effet, nous avons vu (cf p. 8) que :
/* $0 < x < 1$ */
$x = \frac{1}{x} + y$
/* $x > y + 1$ */

et que
/* $x > n$ */
$x = x + 1$
/* $x > n + 1$ */

11. (cf page 8

Schémas conditionnels

L'algorithme linéaire[12] correspond à un schéma très simple : les actions s'enchaînent dans un ordre figé. La réalité est plus souvent construite suivant un schéma conditionnel. La mise en œuvre d'algorithmes conditionnels permet de supprimer le déterminisme lié aux algorithmes linéaires. En programmant la prise de décision nous donnerons à l'ordinateur la capacité de *raisonner*, c'est-à-dire de suivre une démarche logique (exemple : jouer aux échecs) donnant au profane l'impression que l'ordinateur est capable de *penser*.

La puissance de calcul de votre ordinateur est mise en œuvre lorsqu'il évalue les expressions contenues dans les lignes de programmes. Le pouvoir de décision est utilisé pour déterminer l'ordre d'exécution des lignes.

Pour bien saisir le concept de prise de décision d'un ordinateur, il faut savoir ce qu'est le compteur programme. Le compteur programme est la partie du système interne de l'ordinateur capable d'indiquer à l'ordinateur la prochaine ligne à exécuter. À moins d'une indication contraire, le compteur programme s'incrémente à la fin de chaque ligne afin d'indiquer la prochaine ligne du programme.

Sélection
La sélection ou exécution conditionnelle constitue le cœur du pouvoir de décision d'un ordinateur. Comme ce nom l'indique, en fonction des résultats d'un test ou d'une condition, une partie de programme est exécutée ou non. Ceci est la fonction fondamentale qui nous fait croire que la machine est capable de raisonner. En effet, dans le cas de l'exécution conditionnelle, le programme prend en compte et reflète totalement le raisonnement du programmeur.

Prenons comme exemple un laboratoire de chimie. Un ordinateur n'y sera pas d'une grande utilité si sa fonction se limite à ouvrir une valve lorsqu'un technicien appuie sur le bouton START . Dans ce cas, le technicien fera aussi bien de l'ouvrir lui-même. Toutefois, l'ordinateur exécutera une tâche beaucoup plus utile s'il ouvre la valve lorsqu'on appuie sur START et la ferme lorsqu'on atteint la valeur donnée du pH. Il peut surpasser le technicien par exemple dans l'utilisation de valves télécommandées et des dispositifs de mesure électroniques du pH. Dans cet exemple, le côté utile de l'ordinateur demeure sa capacité de décider de la fermeture de la valve. En fait, c'est le programmeur qui spécifie les critères de décision. Ces critères sont ensuite communiqués à l'ordinateur grâce aux structures d'exécution conditionnelle du programme. En conséquence, L'ordinateur est capable d'interpréter la décision du programmeur à une vitesse et à une précision plus grandes que celles d'un être humain.

Il existe diverses applications d'instructions d'exécution conditionnelle

1. Choix conditionnel d'un segment parmi deux[13],
2. Exécution conditionnelle d'un segment (Schéma conditionnel simplifié).
3. Choix conditionnel d'un segment parmi plusieurs[14].

12. Cf. page 13
13. Cf. page 15
14. Cf. page 17

Le schéma alternatif

Grâce à l'instruction **if** ... **else**, vous pouvez exécuter ou sauter un segment de programme. Cette instruction contient une expression numérique qui peut être vraie ou fausse. Si elle est vraie (différente de zéro), le segment conditionnel est exécuté. Si elle est fausse (égale à zéro), le segment conditionnel est ignoré.

Le segment conditionnel peut être soit une seule instruction, soit une partie de programme comprenant un nombre quelconque d'instructions.

if (condition)
{ action(s) 1 **}**
else
{ action(s) 2 **}**

La condition ou *prédicat* est *une fonction propositionnelle* dont le résultat est booléen c'est-à-dire a 2 valeurs : vrai, faux.
Il existe un shéma conditionnel simplifié qui omet le deuxième terme de l'alternative. Exemple Valeur absolue d'un nombre donné.

Les actions internes 1 et 2 du schéma conditionnel peuvent être elles-mêmes conditionnelles. On obtient alors des structures conditionnelles imbriquées [15].

Enfin il existe en langage C une abréviation pour ce schéma : max = a > b **?** a **:** b ;

Propriétés de l'alternative

La succession d'instructions poursuit un objectif : se rapprocher d'un résultat prévu. Avec la structure de contrôle **if** ... **else**, il y a 2 chemins possible pour se rapprocher du même objectif, il y a 2 possibilités de modifier les relations entre les variables.

Supposons

- les relations P et C entre les variables, et qu'une action A conduise à une nouvelle relation Q,
- les relations P et $\overline{C}$ entre les variables, et qu'une action B conduise à une nouvelle relation Q.

La propiété de la structure de contrôle **if** ... **else**, s'écrit :

```
/* P */ if (C) A ; else B ; /* Q */
```

Exemple :

pour montrer que
/* $x < y$ */
if ($x < -2$)
{ $y = x^2$
$x = -x$
}
else $y = x + y + 3$
/* $y > x + 1$ */

il suffit, d'après les propriétés de l'alternative, de montrer séparément deux choses :

1. /* $x < y$ et $x < -2$ */ $y = x^2$; $x = -x$ /* $y > x + 1$ */
2. /* $x < y$ et $x \geq -2$ */ $y = x + y + 3$ /* $y > x + 1$ */

Pour montrer la première partie, il faut appliquer les règles de l'affectation et de l'enchaînement. Raisonnant de *droite à gauche*, nous sommes ramenés à montrer (application des règles de l'affectation à $x = -x$ et $P =$ `/*` $y > x + 1$ `*/`) que :

`/*` $x < y$ et $x < -2$ `*/` $y = x^2$ `/*` $y > -x + 1$ `*/`

ce qui est vrai [16] puisque $x < -2 \Rightarrow x^2 > -x + 1$

Pour montrer la deuxième partie, il suffit de montrer que (substitution de $x + y + 3$ pour y dans `/*` $y > x + 1$ `*/`) :

`/*` $x < y$ et $x \geq -2$ `*/` $\Rightarrow$ `/*` $x + y + 3 > x + 1$ `*/`

ce qui est vrai puisque :

`/*` $x < y$ et $x \geq -2$ `*/` $\Rightarrow$ `/*` $y > -2$ `*/`

15. Cf. page 17
16. Pour éviter le calcul du discriminant voici une démonstration simple : $x < -2 \rightarrow 0 > x + 2$
$x^2 > -2x$ par addition : $x^2 > -2x + x + 2$ et donc $x^2 > -x + 2 > -x + 1$

Raymond Queneau a proposé, sous le titre *Un conte à votre façon*, le texte suivant (extrait de : *L'Oulipo*, Coll. Idées, Gallimard 1972).

Un conte à votre façon

Ce texte soumis à la 83e réunion de travail de l'Ouvroir de Littérature Potentielle, s'inspire de la présentation des instructions destinées aux ordinateurs ou bien encore de l'enseignement programmé. C'est une structure analogue à la littérature "en arbre" proposée par F. Lionnais à la 79e réunion.

1. Désirez-vous connaître l'histoire des trois alertes petits pois ?
   si oui, passez à 4,
   si non, passez à 2.
2. Préférez-vous celle des trois minces grands échalas ?
   si oui, passez à 16,
   si non, passez à 3.
3. Préférez-vous celles des trois moyens médiocres arbustes ?
   si oui, passez à 17,
   si non, passez à 21.
4. Il y avait une fois trois petits pois vêtus de vert qui dormaient gentiment dans leur cosse. Leur visage bien ronds respirait par les trous de leurs narines et l'on entendait leur ronflement doux et harmonieux.
   si vous préférez une autre description, passez en 9
   si celle-ci vous convient, passez à 5.
5. Ils ne rêvaient pas. Ces petits êtres en effet ne rêvent jamais.
   si vous préférez qu'ils rêvent, passez à 6,
   sinon, passez à 7.
6. Ils rêvaient. Ces êtres en effet rêvent toujours et leurs nuits sécrètent des songes charmants.
   si vous désirez connaître ces songes, passez à 11,
   si vous n'y tenez pas, vous passez à 7.
7. Leurs pieds mignons trempaient dans de chaudes chaussettes et ils portaient au lit des gants de velours noirs.
   si vous préférez des gants d'une autre couleur, passez en 7,
   si cette couleur vous convient, passez en 10
8. Ils portaient au lit des gants de velours bleu.
   si vous préférez des gants d'une autre couleur, passez en 8,
   si cette couleur vous convient, passez en 10.
9. Il y avait une fois trois petits pois qui roulaient leur bosse sur les grands chemins. Le soir venu, fatigués et las, ils s'endormirent très rapidement.
   si vous désirez connaître la suite, passez à 5
   sinon, passez à 21.
10. Tous les trois faisaient le même rêve, ils s'aimaient en effet tendrement et, en bons fiers trumeaux, songeaient toujours semblablement.
    si vous désirez connaître leur rêve, passez à 11,
    si non, passez à 12.
11. Ils rêvaient qu'ils allaient chercher leur soupe à la cantine populaire et qu'en ouvrant leur gamelle ils découvraient que c'était de la soupe d'ers. D'horreur, il s'éveillent.
    si vous voulez savoir pourquoi il s'éveillent d'horreur, consultez le Larousse au mot "ers" et n'en parlons plus.
    si vous jugez inutile d'approfondir la question, passez à 12.
12. Opopoï! s'écrient-ils en ouvrant les yeux. Opopoï! Quel songe avons-nous enfanté là ! Mauvais présage, dit le premier. Oui-da, dit le second, c'est bien vrai, me voilà triste. Ne vous troublez pas ainsi, dit le troisième qui était le plus futé, il ne s'agit pas de s'émouvoir, mais de comprendre, bref, je m'en vais vous analyser ça.
    si vous désirez connaître tout de suite l'interprétation de ce songe, passez à 15,
    si vous souhaitez au contraire connaître les réactions des deux autres, passez à 13.
13. Tu nous la bailles belle, dit le premier. Depuis quand sais-tu analyser les songes ? Oui, depuis quand ? Ajouta le second.
    si vous désirez aussi savoir depuis quand, passez à 14,
    si non, passez à 14 tout de même, car vous ne le saurez pas plus.
14. Depuis quand ? s'écria le troisième. Est-ce que je sais moi ! Le fait est que je pratique la chose. Vous allez voir !
    si vous voulez aussi voir, passez à 15,
    si non, passez également à 15, car vous ne verrez rien.
15. Eh bien ! Voyons, dirent ses frères. Votre ironie ne me plaît pas, répliqua l'autre, et vous ne saurez rien. D'ailleurs, au cours de cette conversation d'un ton assez vif, votre sentiment d'horreur ne s'est-il pas estompé ? effacé même ? Alors à quoi bon remuer le bourbier de votre inconscient de papilionacées ? Allons plutôt nous laver à la fontaine et saluer ce gai matin dans l'hygiène et la sainte euphorie ! Aussitôt dit, aussitôt fait : les voilà qui se glissent hors de leur cosse, se laissent doucement rouler sur le sol et puis au petit trot gagnent joyeusement le théâtre de leurs ablutions.
    si vous désirez savoir ce qui se passe sur le théâtre de leurs ablutions, passez à 16,
    si vous ne le désirez pas, vous passez à 21.
16. Trois grands échalas les regardaient faire.
    si les trois grands échalas vous déplaisent passez à 21,
    s'ils vous conviennent passez à 18.
17. Trois moyens médiocres arbustes les regardaient faire.
    si les trois moyens médiocres arbustes vous déplaisent passez à 21,
    s'ils vous conviennent passez à 18.
18. Se voyant ainsi zyeutés, les trois alertes petits pois qui étaient fort pudiques s'ensauvèrent.
    si vous désirez savoir ce qu'ils firent ensuite, passez à 19,
    si vous ne le désirez pas passez à 21.
19. Ils coururent bien fort pour regagner leur cosse et, refermant celle-ci derrière eux, s'y endormirent de nouveau.
    si vous désirez connaître la suite, passez à 20,
    si vous ne le désirez pas passez à 21.
20. Il n'y a pas de suite, le conte est terminé.
21. Dans ce cas, le conte est également terminé.

Raymond Queneaud.

Schémas conditionnels emboîtés et ventilation

Le schéma conditionnel permet de régler l'alternative ou choix d'un ensemble d'actions *parmi* 2 ensembles possibles.

Lorsque l'on doit réaliser le choix d'un ensemble d'actions *parmi* n ($n > 2$) on peut :

– soit utiliser les structures conditionnelles imbriquées mais la lisibilité de la définition algorithmique devient rapidement malaisée dès que n devient important :

```
if    ( condition1 )
      { Action(s) 1 }
else
      { if   ( condition2 )
             { Action(s) 2 }
      else   { Action(s) 4 }
      }
```

– soit utiliser la ventilation qui généralise de manière beaucoup plus satisfaisante le choix de : 1 *parmi* n.

```
switch   ( expression )
         {
         constante1 :     action(s) 1 ;
         constante2 :     action(s) 2 ;
         constante3 :     action(s) 3 ;
         default :        action(s) 4 ;
         }
```

Problème

Définition du problème

Écrire un programme qui fournit le nombre de jours écoulés depuis le premier janvier jusqu'au début du mois. Cet algorithme sera utilisé dans le calendier perpétuel **??**.

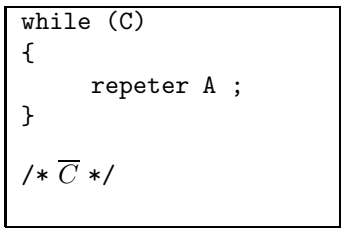
Grille d'analyse

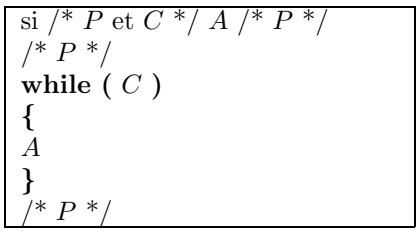
Variables	Actions	ordre
ecoule		

	scanf("%d", &mois) ;
	switch (mois)
	{
	case 1 : nombre = 0 ; break ;
	case 2 : nombre = 31 ; break ;
	case 3 : nombre = 59 ; break ;
	case 4 : nombre = 90 ; break ;
	case 5 : nombre = 120 ; break ;
	case 6 : nombre = 151 ; break ;
	case 7 : nombre = 181 ; break ;
	case 8 : nombre = 212 ; break ;
	case 9 : nombre = 243 ; break ;
	case 10 : nombre = 273 ; break ;
	case 11 : nombre = 304 ; break ;
	case 12 : nombre = 334 ; break ;
	}
	printf("%d\n", nombre) ;	

Le schéma de la répétition

Propriétés de la répétition





À la "sortie" d'une boucle `tant que`, la condition de boucle est toujours *fausse*. Ceci correspond à l'utilisation naturelle de la boucle **while** : on veut assurer, en un certain point du programme, la validité d'une affirmation C. On connaît une action A (qui peut évidemment être complexe) dont on espère qu'elle *rapproche* l'état initial d'un état où C est vraie. On va donc `répéter` A `tant que` C `est vraie`.

Cette propriété est extrêmement importante. Elle exprime le fait que si P est *invariant* pour l'action A, c'est-à-dire si l'exécution de A laisse P vraie lorsque P était vraie initialement, alors P est aussi un invariant pour la boucle **while** (C) { A }.

La notion d'invariant de boucle joue un rôle décisif dans la construction de programmes par des méthodes systématiques. En fait, on peut considérer qu'une boucle `tant que` est entièrement définie par sa condition d'arrêt **et** un invariant. Aussi souvent que possible, nous indiquerons en même temps qu'une boucle un invariant significatif associé.

Un invariant est souvent de la forme : "telle variable a telle valeur". À titre d'exemple, soit le programme ci-dessous, qui localise dans une liste l'élément x :

```
Localiser(x, L)
{
   vrai = 1 ; faux = -1 ; trouve = faux ;
   p = Premier(L) ;
   /* TANT QUE p n'a pas parcouru toute la liste */
   while (p != Fin(L))
   {
     if ( Identique ((Acceder(p, L), x)))
       trouve = vrai ; break ;
     else p = Suivant(p, L) ;
   }
   /* Si trouve == vrai alors renvoie p sinon renvoie faux */
   return  (trouve == vrai)? p:  faux ;
}
```

`/*` la variable "`trouve`" vaut `vrai` si et seulement si $L[p] = x$, et pour tout i compris entre `Premier(L)` et `Acceder(p-1, L)` $\neq$ `x */`

Cet invariant étant vrai initialement (puisque `trouve = faux`, il reste vrai au sortir de la boucle. Joint à la négation de la condition de bouclage, c'est-à-dire `/* p = Fin(L) ou trouve */`, il donne comme assertion finale :

```
   /* trouve == faux et aucun élément de L ne vaut x,
ou bien trouve == vrai et x = L(p) */
```

ce qui est bien le but recherché.

Corriger un programme faux

Dans le numéro de janvier-février 1979 de l'*ordinateur individuel*, est paru ce programme de calcul de `x` puissance `n`, que nous étudierons réécrit en langage C :

```
01 void main()
02 {
03     int x, n, a, i = 0, t ;
04     scanf("%d", &x) ; scanf ("%d, &n) ;
05    a = x ;
06    do
07    {
08        a = a * x ;
09        i = i + 1 ;
10        t = n - 1 ;
11    } while (i < t) ;
12    printf("%d\n", a) ;
13 }
```

La ligne (8) opère par multiplications répétées par x. Il est facile de voir qu'il est faux pour des valeurs simples :

$x = 2$ et $n = 1$

(5) a prend la valeur de x, donc $a = 2$.
(3), $i = 0$, (8), a est multiplié par x, $a = 2 \times 2 = 4$
(9), i est augmenté de 1 $i = 1$
(11), $i = 1$ n'est pas inférieur à t ($t = 0$), on passe donc en (12) et on affiche le résultat **4**, évidemment faux.

Examinons la boucle (6-11) : on arrive la $1^{ère}$ fois en (6) en venant de (5) avec /* $a = x$ et $i = 0$ */

Essayons l'assertion $a = x^{i+1}$ $\left\{ \begin{array}{lll} (9) & /*\ a = x^{x+1}\ */ & a = a \times x \quad /*\ a = x^{x+2}\ */ \\ (10) & /*\ a = x^{x+2}\ */ & i = i + 1 \quad /*\ a = x^{x+1}\ */ \end{array} \right.$

Remarquons que l'assertion $a = x^{x+1}$ est rétablie, avec une valeur de i plus grande. Mais la donnée introduite est i . Il faut donc situer i par rapport à n.

`(11)` /* $a = x^{i+1}$ */ **si** $i < t$ **alors** $a = x^{i+1}$ `et` $i < n - 1$ **boucler**
`(12)` /* $a = x^{i+1}$ `et` $i >= n - 1$ */

Si, en (11), i devient égal à $n - 1$ alors $a = x^n$ et le programme est correct. Or par l'initialisation $a = x$ et $i = 0$

Il faut donc avoir $0 < n - 1$ ou $n > 1$. Nous constatons que le programme calcule $a = x^n$ **si** et **si seulement** $n > 1$.

Il n'est pas très facile de corriger le programme. Il paraît probable que l'auteur a mal choisi son test d'arrêt, et qu'il a cherché à le corriger en introduisant la variable t calculée ligne (10) dont la valeur est constante. Son calcul dans la boucle n'est pas normal.

Il est plus important de considérer les situations que les actions. Pour **rédiger** le programme, il faut d'abord proposer une situation générale.

Supposons que l'on ait fait une partie du travail, et calculé $a = x^i$ pour $i \leq n$.

`(7bis) /* on a calculé` $a = x^i$ `et` $i \leq n$ `*/`

Si $i = n$, c'est fini ; si non, il faut se rapprocher de la solution en faisant croître i `(8-9)` $a = a \times x$; $i = i + 1$; **boucler en**`(6)`

Il reste a voir le démarrage, c'est à dire trouver des valeurs de a et i telles que l'assertion soit vérifiée pour tout n positif ou nul. Il faut prendre $i = 0$ et donc $a = x^0 = 1$. D'ou le nouveau programme correct, cette fois.

```
01 void main()
02 {
03     int x, n, a, i = 0, t ;
04     scanf("%d", &x) ; scanf ("%d, &n) ;
05    a = 1 ;
06    while (i < n)
07    {
07bis     /* a == x^i et i < n */
08        a = a * x ;
09        i = i + 1 ;
10
11    }
11bis /* a == x^i et i == n */
12    printf("%d\n", a) ;
13 }
```

La grille d'analyse

La grille d'analyse intervient lors de l'étape de l'écriture de l'algorithme avec un métalangage. Cette grille permet d'organiser l'expression de l'algorithme.

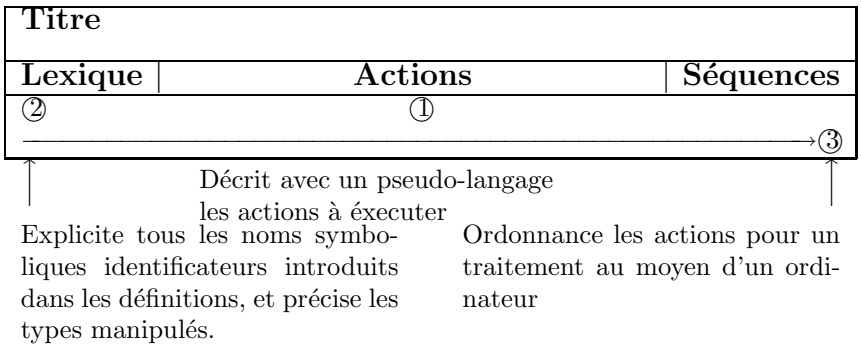


FIG. 10.1 – *La grille d'analyse*

La grille d'analyse est un tableau de 3 colonnes (fig. 10.1) réalisant le schéma dans lequel la conception de l'algorithme s'organise et se développe.

Dans la colonne centrale, **on commence par la dernière action**, ce qui fait apparaitre une variable au moins, que l'on cherche à **expliciter**. Cette variable en **présuppose** d'autres que l'on explicite, et ainsi de suite jusqu'à ce que toutes les variables soint expliquées. Dans la colonne *Lexique*, on précise pour chaque variable son domaine de définition. On termine en fixant dans la colonne de droite l'ordre d'exécution pour le programme.

Ordre ① ⟷ ② puis à la fin ③

Exemple : Écrire le programme donnant le poids idéal d'une personne.

Le POIDS en Kg dépend de la TAILLE en cm et d'une constante :
`POIDS = TAILLE − VALEUR`
où `TAILLE > 150 cm`

VALEUR est fonction du sexe

`VALEUR = 10×FACTEUR`

`FACTEUR =`	`9`	`SI SEXE = "masculin"`
	`10`	`SI SEXE = "féminin"`

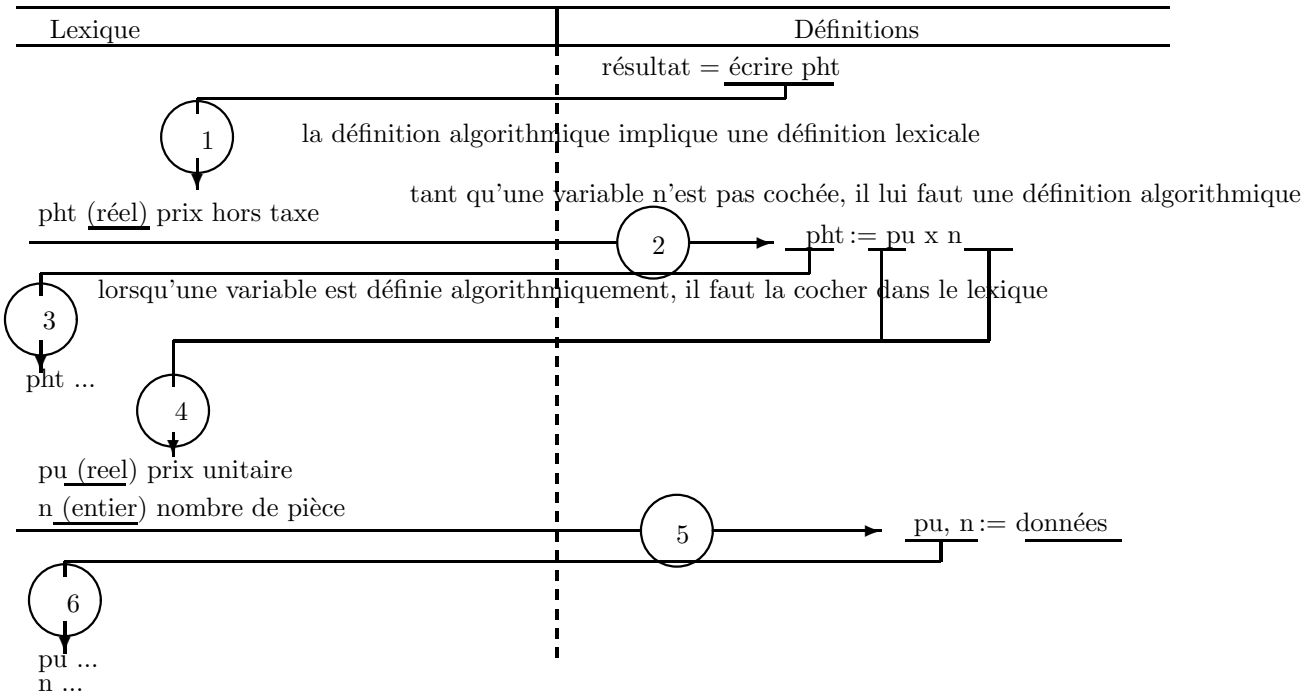
Poids-idéal		
Lexique	**Définitions**	**Séquence**
NOM (chaîne) : nom de la personne (2)	Résultat = écrire NOM, POIDS (1) NOM = donnée ("nom") (4)	7 1
POIDS (réel) : poids en Kg (3)	POIDS = TAILLE − VALEUR (5)	6
TAILLE (entier) : taille en cm (6)	TAILLE = donnée ('taille en cm > 150') (8) VALEUR = 10 × FACTEUR (9)	3 5
VALEUR (entier) : valeur à soustraire de la taille (7)	(11) SI SEXE = "masculin" ALORS FACTEUR = 9 SINON FACTEUR = 10 FINSI	4
FACTEUR (réel) : coefficient de correction fonction du sexe (10)		
SEXE (chaîne) : sexe de la personne (12)	(13) SEXE = donnée ("sexe? répondre par masculin ou féminin")	2

Les numéros entre parenthèses sont indiqués optionnellement pour montrer l'ordre de remplissage.

Problème : Facture de pièces identiques

pièces_id		
Lexique	Définitions	Séquence
PHT (réel) prix hors taxe des pièces	résultat = écrire "prix hors taxe=" PHT à la ligne "prix toute taxe =" PTT	7
PTT (réel) prix toute taxe des pièces	PHT = PU × N	4
	PTT = PHT + TAXE	6
PU (réel) prix unitaire ht	PU = donnée ("prix unitaire")	3
N (entier) nombre de pièces	N = donnée ("nombre de pièces")	2
TAXE (réel) TVA	TAXE = TAUX × PHT	5
TAUX (réel) taux de TVA	TAUX = 0.186	1

Contrôle de l'algorithme à l'aide du lexique
À chaque définition algorithmique de l'identificateur d'une donnée ou d'un résultat intermédiaire, on *coche* celui-ci dans le lexique(fig. ci-dessous). Ce lexique montre donc à *chaque instant* ce qui reste à définir.



Tous les identificateurs sont cochés (donc définis)
L'algorithme est terminé

Le pgcd d'Euclide

Euclide (300 AJC) a donné un algorithme pour trouver le PGCDde 2 **nombres entiers positifs** :

```
PGCD (a, b)
    si (b == 0) renvoie (a) ;
    sinon       renvoie (PGCD(b, a mod b)) ;
```

ou son équivalent sous forme d'une boucle

```
1 PGCD (a, b)
2     tant que b != 0
3         t = a
4         a = b
5         b = t mod b
6     fin de tant que
7 renvoie a
```

11.1 Preuve mathématique

1. Si $d = \text{PGCD}(a, b)$ alors d divise a et b
2. $(a \bmod b) = a - q.b$ (q = reste de la division de a par b)
   $\longrightarrow (a \bmod b)$ est une combinaison linéaire de a et b
   comme d divise b et a, et que $(a \bmod b)$ est une combinaison linéaire de a et b
   alors d divise $(a \bmod b)$
   cela permet de dire que d divise $\text{PGCD}(b, a \bmod b)$

   Cela implique que :

   d qui est $\text{PGCD}(a, b)$ divise $\text{PGCD}(b, a \bmod b)$
   donc, $\text{PGCD}(a, b)$ divise $\text{PGCD}(b, a \bmod b)$

   idem pour $\text{PGCD}(b, a \bmod b)$ divise $\text{PGCD}(a, b)$

   donc : $\text{PGCD}(a, b) = \text{PGCD}(b, a \bmod b)$

11.2 Preuve de l'algorithme

11.2.1 Trace de l'algorithme pour 30 et 21

Lignes	1	3	4-5	3	4-5	3	4-5	
a	30	30	21	21	9	9	**3**	⟵ PGCD
b	21	21	9	9	3	3	0	⟵ conditon d'arrêt
t	–	30	30	21	21	9	9	

b ne peut jamais devenir négatif et il est toujours diminué de la valeur $(a \bmod b)$ qui est $< b$.

Modulo est une fonction qui
- ramène toujours une valeur inférieure d'au moins 1 par rapport à l'opérande droit.
  Ex : $3 \bmod 2 = 1$, $1 < 2$,
- possède aussi comme caractéristique de donner 0 si b vaut 1, car $\forall x, x/1 = x$, donc $(x \bmod 1) = 0$,
- renvoie a, si $a < b$.

Donc, b diminue de un moins 1 à chaque appel, et à la fin il doit valoir 0, donc la condition de sortie de la fonction arrivera **toujours**, quelles que soient les valeurs a et b données en entrées entières positives. Cette première validation est extrêmement importante, il est fondamental qu'un programme récursif, un boucle, ait une condition d'arrêt.

Le schéma fonctionnel (I)

Dans la méthode de construction des algorithmes[17], nous partons de l'objectif final, le plus souvent une valeur à calculer ; ce faisant, nous faisons apparaître une variable pour laquelle il faut expliciter le mode de calcul. Ce calcul **présuppose** lui-même le calcul d'autres variables. On comprend intuitivement que la valeur finale est calculée en *fonction* des variables dont elle dépend. Chaque calcul peut faire l'objet d'un algorithme et on peut utiliser, dans une définition d'un algorithme, une valeur résultant de l'exécution d'un autre algorithme.

Soit par exemple à fabriquer une série d'appareils dont chaque exemplaire sera équipé d'un quartz et de deux diviseurs de fréquence correspondant à la demande de chaque client. À la commande, le client précisera les 2 fréquences dont il a besoin. Nous allons écrire le programme qui détermine la valeur du quartz et les valeurs des deux diviseurs de fréquence, sachant que les toutes ces valeurs sont des nombres entiers.

FREQ : /* Calcul d'une fréquence et de deux diviseurs */		
Lexique	Actions	ordre
Fq : Fréquence du quartz	AFFICHER(Fq, $div1$, $div2$) ;	5
F_1 et F_2 : Fréquences client	$Fq =$ PPCM(F_1, F_2) ;	2
$div1$: diviseur pour F_1	$div1 = \frac{Fq}{F_1}$;	3
	$div2 = \frac{Fq}{F_2}$;	4
$div2$: diviseur pour F_2	SAISIR(F_1, F_2) ;	1

La valeur Fq sera renvoyée par le calcul du PPCM de F_1etF_2.

PPCM(a, b) /* Calcul du Plus Petit Commun Multiple */		
	PPCM $= \frac{a \times b}{\text{PGCD}(a,b)}$;	

La valeur du PPCM sera renvoyée par le calcul du PGCD de F_1etF_2.

PGCD(x, y) /* Calcul du Plus Grand Commun Diviseur */		
m : variable temporaire	**while** ($x! = 0$) **{** $m = x$; $x = y$; $y = m$ modulo y ; **}** **return** y ;	

On voit donc que l'on est conduit à écrire 2 nouvelles fonctions (qui ne sont pas natives) le PPCM et le PGCD pour réaliser ce programmme.

Relation de précèdence On définit une relation de précèdence sur les variables d'un algorithme. Nous dirons que la variable x **précède** la variable y **si** x a au moins **une occurrence dans la définition** de y. À cause de cela, la valeur de y dépend de la valeur de x, et le calcul de y n'est possible que si x a déjà été calculé. La relation "x précède y" peut donc être lue comme "le calcul de x doit précéder le calcul de y".

Cette relation est un ordre partiel sur les variables de l'algorithme.

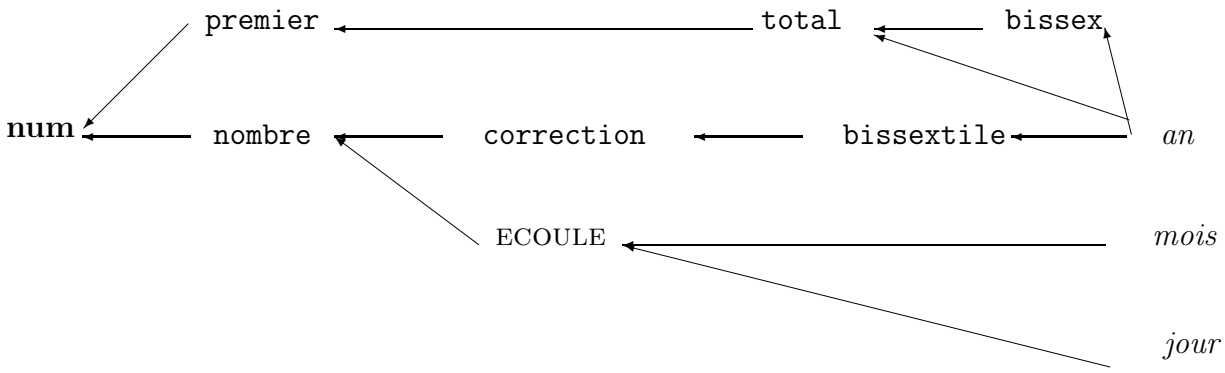
17. Cf. page 20

Un calendrier perpétuel

Définition du problème : Une date étant donnée par son quantième (`jour`), son mois (`mois`), son millésime (`an`) (sur 2 chiffres dans l'intervalle [1901–2099]), dire quel jour de la semaine (`num` ∈ [0-6], 0 ≡ dimanche) lui correspond.

Pour cela, on va chercher le décalage entre le 1^{er} janvier de l'année `an` et la date considérée. On se donne une table du nombre de jours écoulés depuis le début de l'année jusqu'au début du mois `mois`. Quant au jour du 1^{er} janvier de l'année `an` , c'est le jour de Noël de l'armée `an`-1.

Calper(jour, mois, an)	
`num ∈ [0-6]` `nombre ∈ [1-365]`	**printf** (”%d\n”, num) ; num = (nombre + premier - 1) **modulo** 7 ; nombre = jour + ECOULE(mois) + correction ; correction = (bissextile **et** mois > 2) ? 1 : 0 ; bissextile = (0 == (an **modulo** 4)) ? vrai : faux ; premier = NOEL(an - 1) ;
NOEL(an)	
int total ; **int bissex ;**	bissex = an **modulo** 4 ; total = an + bissex ; **return** (total + 2) **modulo** 7 ;
ECOULE(mois)	
	switch (mois) **{ case** 1 **: return** 0 **; case** 2 **: return** 31 **; case** 3 **: return** 59 **;** **case** 4 **: return** 90 **; case** 5 **: return** 120 **; case** 6 **: return** 151 **;** **case** 7 **: return** 181 **; case** 8 **: return** 212 **; case** 9 **: return** 243 **;** **case** 10 **: return** 273 **; case** 11 **: return** 304 **; case** 12 **: return** 334 **; }**



On vérifie sur le graphe de *dépendance des variables* de la figure que l'objectif (`num`) présuppose le calcul des variables dont il dépend, et que chacune d'elles est explicitées, soit par un calcul simple, soit par l'appel d'une fonction, jusqu'à remonter aux variables fournies à l'algorithme.

La programmation fonctionnelle (II)

15.1 La récursivité

Reprenons le calcul de x^n. On suppose que l'on a fait une partie du travail, disons on a calculé x^i. C'est fini si $i = n$. Sinon, en multipliant le résultat déjà calculé par x, on obtient x^{i+1} Par changement de i en $i+1$, on se ramène à l'hypothèse de départ. Pour démarrer, on peut prendre $i = 1$ et $x^i = x$.

La récurrence joue donc de la façon suivante:

- si j'ai pu calculer x^i, je peux calculer x^{x+1}
- or je peux calculer x^1.

Sans rien changer à cette forme de raisonnement, nous pouvons construire un algorithme "récursif".

```
  expr(x, n)
{
    if (n == 1) return x ;
    else return x * expr(x, n-1) ;
}
```

Pour écrire la définition récursive de $\texttt{exp}(x, n)$, nous avons utilisé la propriété bien connue

$$x^n = x \times x^{n-1}$$

15.2 Construction de définitions récursives

Reprenons le calcul de x^n

$$\texttt{exp}(x, n) = \underbrace{x \times x \times x \times ... \times x}_{(nx \text{ dans cette formule})}$$

Encadrons les $n-1$ x de droite

$$\texttt{exp}(x, n) = x \times \boxed{x \times x \times ... \times x}$$

($n-1$ x dans la boîte)

On retrouve la relation connue

$$\texttt{exp}(x, n) = x \times \texttt{exp}(x, n-1)$$

Comme cas singulier, on peut prendre $\texttt{exp}(x, 1) = x$ ou $\texttt{exp}(x, 0) = 1$ (qui a l'avantage d'étendre la définition au cas $n = 0$)

```
exp(x, n)
{
    if (n == 0)
        return 1 ;

    return x * expr(x, n-1) ;
}
```

On peut grouper autrement les x dans $\texttt{exp}(x, n)$. Supposons n pair. On peut partager les x en deux paquets égaux :

$$\texttt{exp}(x, n) = \boxed{x \times x \times \ldots \times x} \times \boxed{x \times x \times \ldots \times x}$$

avec $\frac{n}{2}$ x dans chaque paquet.
Dans ce cas

$$\texttt{exp}(x, n) = \texttt{exp}(x, \tfrac{n}{2})^2$$

Si maintenant n est impair, nous pouvons faire de même, en mettant à part le premier x

$$\texttt{exp}(x, n) = x \times \boxed{x \times x \times \ldots \times x} \times \boxed{x \times x \times \ldots \times x}$$

On obtient ainsi une deuxième définition :

```
exp2(x, n)
{
    if (n == 0)
        return 1 ;

    y = expr2(x, n/2) ;
    if (pair(n))
        return y * y ;
    return y * y * x ;
}
```

Nous pouvons faire, quand n est pair, les groupements d'une autre façon, en enfermant les paires de x dans des boîtes.

$$\texttt{exp}(x, n) = \boxed{x \times x} \times \boxed{x \times x} \times \boxed{x \times x} \ldots \times \boxed{x \times x}$$

Il y a $\frac{n}{2}$ boîtes ayant chacune deux x

```
exp3(x, n)
{
    if (!n) return 0 ;
    if (pair(n)
        return exp3(x*x, n/2) ;
    return x * exp3(x*x, n/2) ;
}
```

Les listes

16.1 Généralités

Une liste est un objet informatique comprenant les informations (données) ainsi que les opérations nécessaires au traitement de ces données.

Une liste doit pouvoir évoluer : on doit être capable de supprimer ou d'ajouter des données.

Pour accéder à une donnée appartenant à une liste, il convient de disposer de la liste mais aussi de la place de cette donnée dans la liste.

Il existe une liste vide ; si une liste n'est pas vide, on peut la vider ; on peut accéder au $i$ème élément, en connaître le contenu ; on peut connaître la longueur de la liste (le nombre d'élément qu'elle contient) ; on peut supprimer un élément de la liste, y insérer un nouvel élément et enfin accéder au sucesseur d'un élément dont on connaît la place.

D'autre part, on peut vouloir fabriquer une liste à partir de deux autres.

Il existe plusieurs type de listes.

1. La Pile (LIFO) **2. La File. (FIFO)**

16.2 Mise en œuvre des listes dans un tableau

La figure 16.2 montre un tableau `ESPACE` contenant deux listes $L = a, b, c$ et $M = d, e$. Remarquez que toutes les cellules du tableau n'appartenant à aucune des deux listes sont chaînées en une autre liste appelée *disponible*. Cette liste sert à trouver un espace libre pour insérer un nouvel élément, ou à récupérer les espaces libérés par la suppression d'éléments précédemment dans la liste en vue d'une utilisation ultérieure.

Pour insérer un élément x dans une liste L, on utilise la première cellule libre dans la liste disponible et on la place à la bonne position dans la liste L. L'élément x est ensuite placé dans le champ élément de cette cellule. Pour supprimer un élément x de L, on retire la cellule contenant x de la liste et on la remet en tête de la liste disponible.

En d'autres termes, C est inséré entre q et l'élément sur lequel pointait q. Par exemple, si l'on supprime b de la liste L sur la figure 16.2 C se trouve à la ligne 8 d'`ESPACE`, p est égal à `ESPACE[5].suivant` et q est égal à disponible. Les curseurs avant (tracé continu) et après (tracé en pointillés) le groupe l'actions précédent sont représentés sur la figure 16.2. La fonction générique correspondante `transfert` est présentée à la figure 16.2 le transfert est opéré si C existe et la valeur retournée est alors `vrai`. Sinon, le transfert ne peut être fait et la fonction retourne `faux`.

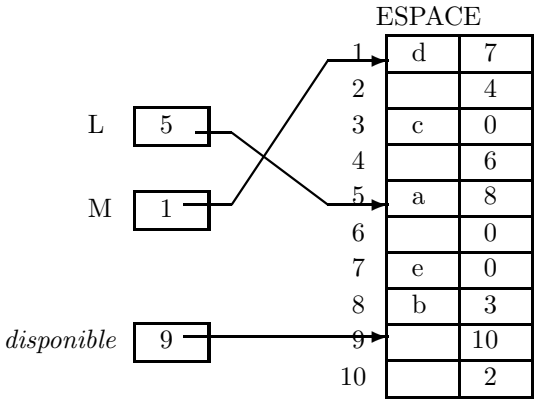


FIG. 16.2 – *Implantation d'une liste chaînée par faux-pointeur*

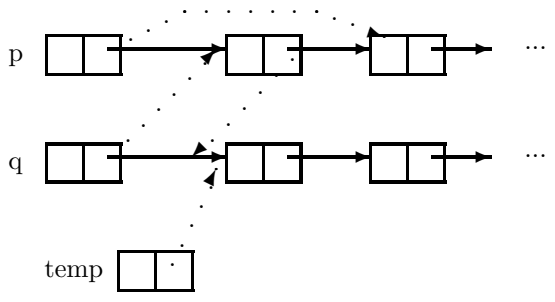


FIG. 16.3 – *Transfert d'une cellule C d'une liste à l'autre.*

Les piles et les files

17.1 Exemple de pile : éditeur de ligne

Les éditeurs de textes permettent toujours l'utilisation d'un caractère spécifique pour l'effacement de caractère (par exemple le caractère correspondant à la touche "retour arrière"). L'action d'un tel caractère est de détruire le dernier caractère non détruit. Par exemple, si '`#`' est le caractère d'effacement, la chaîne `abc###e` est considérée finalement comme la chaîne `ae`. Le premier caractère '`#`' détruit `c`, le second `d` et le troisième `b`.

Les éditeurs de texte disposent aussi d'un caractère de destruction voué à l'annulation de tous les caractères se trouvant précédemment sur la ligne courante. Pour les besoins de cet exemple, nous supposerons que '`@`' est ce caractère de destruction.

Un éditeur de texte peut traiter une ligne de caractère à l'aide d'une pile, en lisant un caractère à la fois. Si le caractère lu n'est ni le caractère d'effacement ni celui de destruction, il est placé au somment de la pile. S'il s'agit du caractère d'effacement, un dépilement est effectué, et s'il s'agit du caractère de destruction, la pile est complètement vidée.

17.2 Les files

Peut-être plus encore que les piles, les files d'attente (ou simplement files) font partie de notre vie courante ; du moins en sommes-nous plus conscients, puisqu'il nous arrive quotidiennement de faire la queue devant un guichet.

Une file est un TDA formé d'un nombre variable, éventuellement nul, de données, sur lequel on peut effectuer les opérations suivantes :

- ajout d'une nouvelle donnée ;
- test déterminant si la file est vide ou non ;
- consultation de la première données ajoutée et non supprimée depuis (donc la plus ancienne) s'il y en a une ;
- suppression de la donnée la plus ancienne.

Cette conception s'accorde bien avec la conception intuitive que l'on a d'une file d'attente.

Les files d'attente ont une grande importance en informatique ; elles s'appliquent à deux types de problèmes :

- la simulation de files réelles ; les techniques modernes de communication (terminaux reliés à un ou plusieurs centres de communication). Il est devenu courant d'écrire des programmes qui étudient les comportements de réseaux.
- la résolution de problèmes purement informatiques, en particulier dans le domaine des systèmes d'exploitation.

17.2.1 Analyse fonctionnelle :

La file est constituée

- d'une suite d'éléments ordonnés $a_1, a_2, \ldots, a_n$, désignée ici par F, éventuellement vide.
- des primitives suivantes :

  creer(F) Cette fonction crée une file de nom F et retourne la valeur `vrai` si l'opération a pu s'effectuer sans problème, sinon elle retourne `faux`.

  filevide(F) Cette fonction teste la vacuité de la file F et retourne `vrai` si la file est vide `faux` sinon.

  enfiler(x, F) Cette fonction ajoute un élément en queue de file, renvoie `faux` s'il l'élément n'a pas pu être introduit, `vrai` sinon.

  defiler(F) Cette primitive retire l'élément de tête de la file.

  premier(F) Cette fonction renvoie la valeur de l'élément en tête de file, si la pile n'est pas vide, sinon retourne un code d'erreur.

17.2.2 Description logique et fonctionnement

La file ne peut avoir qu'une taille maximale égale à la dimension du tableau, et peut être schématisée selon la figure 17.4

Notez que le fait de choisir les indices `TETE` et `QUEUE` comme nous l'avons fait, avec `TETE` désignant la position passée de la tête de la file, n'influe pas sur le problème. Cette convention facilite uniquement l'écriture des fonctions `vide` et `enfiler`.

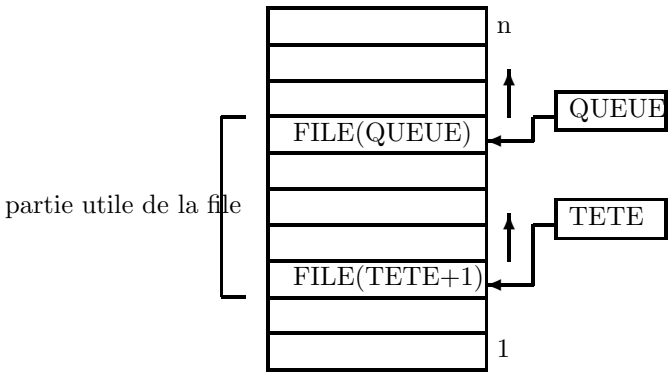


FIG. 17.4 – *Concretisation d'une file dans un tableau.*

Un problème se pose : même si la taille de la file reste constamment en dessous du maximum permis n, la file "monte" inexorablement, puisque les défilages s'effectuent par le bas et les enfilages par le haut. Si l'on n'y prend garde, la file débordera du tableau au bout de n opérations `enfiler`.

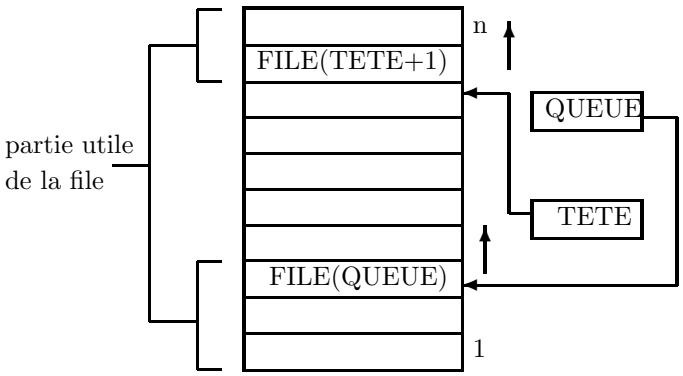


FIG. 17.5 – *Fonctionnement d'une file circulaire dans un tableau.*

Plusieurs solutions sont possibles :

1. à chaque défilage, récupérer l'espace libéré en bas du tableau en "redescendant" toute la file d'un cran. C'est la solution simple à mettre en oeuvre, mais coûteuse sur les grandes files.

2. laisser la file monter tant qu'il reste de la place pour effectuer les enfilages ; quand il n'y a plus de place et que l'on veut opérer un enfilage, on récupère d'un seul coup la place libérée par les défilages en "redescendant" la file de toute la hauteur possible. Cette solution est plus économique que la précédente, mais exige encore des transferts d'informations inutiles.

3. quand la queue atteint le haut du tableau, effectuer les enfilages suivants à partir du bas du tableau, qui prend l'aspect de la figure 17.5

L'intérêt de cette méthode est qu'elle ne nécessite aucun décalage. On parle d'une représentation par file circulaire, on peut représenter chacune des deux figures précédentes par un anneau.

La programmation de cette solution présente un piège : le test déterminant si la file est vide s'écrit maintenant `TETE = QUEUE` si la file à n éléments. Pour pouvoir distinguer entre ces deux cas (file vide et file pleine), on imposera à la file la capacité maximale $n-1$ (et non n). Dans ces conditions `|TETE-QUEUE|`$\geq$`1` si la file n'est pas vide.

Table de hachage

18.1 Objectif

On veut vérifier que $x \in \mathcal{E}$. Par exemple une vérification orthographique recherche si le mot (x) appartient au dictionnaire $\mathcal{E}$. La complexité [18] en temps de la recherche dans un tableau ou dans une liste devient prohibitive avec la taille de l'ensemble.

18.2 Algorithme

On choisit donc une fonction, appelée fonction de hachage, qui associe à chaque élément une valeur numérique. On peut dès lors, accéder à la position dans le tableau qui est indexé par cette valeur numéque.

18.3 Exemple

Soit une fonction qui associe à chaque lettre son rang dans l'alphabet,

lettre	a	b	c	...	r	s	...	z
rang	1	2	3	...	18	19	...	26

et une fonction $\mathcal{H}$ de hachage qui associe à chaque mot une classe d'équivalence modulo 6 de la somme des rangs de chacune des lettres qui le composent.

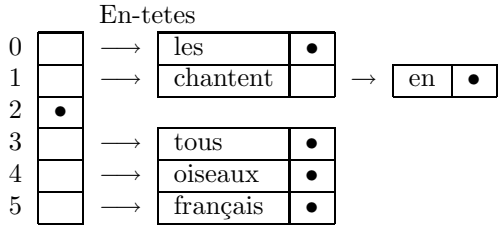
Pour l'ensemble des mots :

tous les oiseaux chantent en francais.

on obtient :

mots	tous	les	oiseaux	chantent	en	francais
somme	75	36	94	85	19	71
$\mathcal{H}$	3	0	4	1	1	5

que l'on conserve en mémoire :



18.4 Implémentation des opérations du dictionnaire par une table de hachage

1. calculer le bon paquet pour le mot x_0 à rechercher : $\mathcal{H}(x_0)$,
2. utiliser le tableau de pointeurs d'*En-tetes* pour trouver la liste des éléments pour le paquet numéroté $\mathcal{H}(x_0)$,
3. réaliser l'opération sur cette liste, comme si la liste repésentait l'ensemble complet.

18. [p.10]

Table des figures

Index