1 La notion d'algorithme

Recette de l'omelette
Voici comment préparer une délicieuse omelette

nécessaire
- 2 oeufs
- sel
- un peu de matière grasse
- un bol
- une poêle

étapes

1. casser les oeufs dans le bol
2. ajouter le sel
3. battre les oeufs
4. faire chauffer la matière grasse dans la poêle
5. verser le mélange des oeufs dans la poêle et faire cuire doucement jusqu'à la consistance souhaitée

Algorithme de construction à la règle et au compas de la bissectrice d'un angle.
rappel : la bissectrice est la droite divisant un angle en deux angles égaux.

données trois points A, O et B

résultat une droite D

spécification D est la bissectrice de l'angle $\widehat{AOB}$

1. tracer le cercle $\mathcal{C}$ de centre O passant par A.
2. soit B' le point d'intersection de $\mathcal{C}$ et de la demi-droite $[OB)$
3. tracer $\mathcal{C}_{\mathcal{A}}$ le cercle de centre A passant par O
4. tracer $\mathcal{C}_{\mathcal{B}}$ le cercle de centre B' passant par O
5. marquer I "l'autre" point d'intersection de $\mathcal{C}_{\mathcal{A}}$ et $\mathcal{C}_{\mathcal{B}}$
6. tracer la droite D passant par I et O. C'est la bissectrice de l'angle $\widehat{AOB}$.

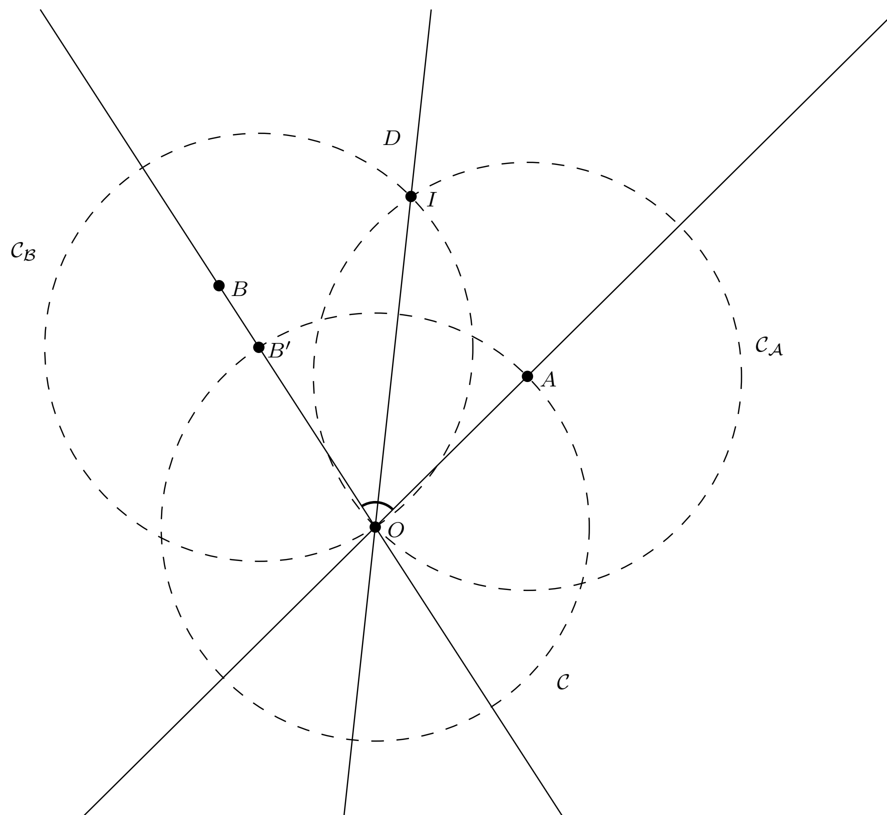


FIG. 1 – Construction de la bissectrice

Algèbre en langage courant

1. Trouver le nombre tel que quand il est porté au cube et que ce nombre est ajouté à ce cube, il est égal à cinq.
2. Un carré et dix de ses racines égalent trente-neuf.
3. Trouver deux nombres l'un étant le double de l'autre tels que quand le carré du plus grand est multiplié par le plus petit et que ce produit est ajouté aux deux nombres originaux, le résultat est égal à 40.

Une citation

« Computer science is no more about computers than astronomy is about telescopes. »

E. Dijkstra

Résumé du chapitre 1

– L'algorithmique est la discipline qui étudie les algorithmes : leur conception, leurs performances, les moyens de prouver qu'ils sont valides. C'est une discipline dont les fondements remontent à l'antiquité mais qui s'est considérablement développée tout au long de la seconde moitié du vingtième siècle avec l'avènement de l'ordinateur.
– Un algorithme agit sur des données et consiste en une séquence finie d'opérations qui permettent d'obtenir un résultat. La relation entre les données et le résultat (ce "à quoi sert" l'algorithme) est appelée sa spécification.
– L'exécution d'un algorithme consiste à effectuer scrupuleusement la séquence d'opérations. Cette exécution est mécanisable, c'est à dire qu'elle peut être réalisée par un serviteur stupide mais dévoué.
– S'il est en théorie possible d'utiliser le français pour décrire un algorithme, cela n'est raisonnable que pour des algorithmes très simples. Dès que les choses deviennent un peu compliquées on doit utiliser un langage spécialisé (algorithmique) qui permet de s'exprimer et de penser de manière précise et concise.

2 Bases du langage algorithmique

```
Algorithme EquationPremierDegre
(* Résoud une équation du premier degré, de type a x = b *)

var a:réel (* le coefficient a, donné par l'utilisateur *)
    b:réel (* le coefficient b, donné par l'utilisateur *)
    r:réel (* le résultat, solution de l'équation *)
début
   écrire("Donnez le coefficient a")
   lire(a)
   écrire("Donnez le coefficient b")
   lire(b)
   r ← b/a
   écrire("La solution est ", r)
fin
```

entier : sous-ensemble de l'ensemble des entiers relatifs. Valeurs littérales 1 2 –5 9999. On peut effectuer sur les entiers des opérations arithmétiques : – unaire (négation) +, –, ×, div (quotient entier), mod (reste de la division).

réel : type des valeurs numériques non forcément entières (valeurs approchées d'éléments de $\mathbb{R}$). Valeurs littérales : 3.1415927 –1.5 9999.0 (utilisation du point décimal). Les opérations arithmétiques classiques – unaire (négation), + – × / (division réelle) sont disponibles.

booléen (ou logique) : ensemble ne contenant que les 2 valeurs : vrai, faux. Les opérations disponibles sont : non, et, ou.

car : l'ensemble des caractères représentables pour un codage particulier (essentiellement les caractères alphabétiques, numériques, et de ponctuation). Les valeurs littérales sont notées entre apostrophes : 'a' '4' ',' ' ' (espace).

chaîne (de caractères) : une chaîne de caractères est une suite (finie) de caractères. Elle sont principalement utilisées dans les algorithmes pour communiquer avec l'utilisateur (lire ou écrire). Les valeurs littérales sont notées entre guillements : `"Bonjour !"`, `"Donnez le coefficient a"`, `"La solution est "`.

<table>
<tr><th>opérateur</th><th>type des opérandes</th><th>type du résultat</th><th>opération</th></tr>
<tr><td>–</td><td>1 entier ou réel</td><td>idem opérande</td><td>négation</td></tr>
<tr><td>+
–
×</td><td rowspan="2">2 entiers ou 2 réels
ou 1 entier et 1 réel</td><td>entier si les 2
opérandes le sont,
réel sinon</td><td>addition
soustraction
multiplication</td></tr>
<tr><td>/</td><td>réel</td><td>division réelle</td></tr>
<tr><td>div
mod</td><td>2 entiers</td><td>entier</td><td>quotient entier
reste de la division</td></tr>
</table>

TAB. 1 – Opérateurs arithmétiques

<table>
<tr><th>opérateur</th><th>type des opérandes</th><th>type du résultat</th><th>opération</th></tr>
<tr><td>$<$
$\leq$
$>$
$\geq$
$=$
$\neq$</td><td>2 nombres, 2
caractères ou 2
chaînes</td><td>booléen</td><td>inférieur
inférieur ou égal
supérieur
supérieur ou égal
égal
différent</td></tr>
</table>

TAB. 2 – Opérateurs relationnels

opérateur	type des opérandes	type du résultat	opération
non	1 booléen	booléen	négation logique
et	2 booléens		conjonction
ou			disjonction

TAB. 3 – Opérateurs logiques

Plus forte classe 1	non – unaire
classe 2	et × / div mod
classe 3	ou + –
Plus faible classe 4	$= \neq < \leq > \geq$

TAB. 4 – Priorité des opérateurs

Résumé du chapitre 2

- Les variables sont des “boites” dans lesquelles on peut stocker une valeur. Une variable doit être *déclarée*, sa valeur est initialement *indéterminée*, elle peut ensuite prendre toute valeur correspondant à son *type*.
- Chaque donnée manipulée par un algorithme est caractérisée par un *type*. Celui-ci définit l’ensemble des valeurs possibles et des opérations applicables sur cette donnée. On définit les types `entier`, `booléen`, `réel`, `car` (caractère) et `chaîne` (de caractères, c’est à dire un fragment de texte).
- Une expression représente une valeur. Cette valeur n’est pas forcément immédiatement apparente, elle est calculée par *évaluation* de l’expression. Une expression peut être une simple *valeur littérale*, une variable (c’est alors sa valeur qui est prise en compte) ou une combinaison d’expressions par des opérateurs du type.
- L’affectation est une instruction de la forme *var* ← *exp* qui procède à l’évaluation de l’expression *exp* puis copie cette valeur dans la variable *var*.
- Les instructions d’entrées/sorties permettent de communiquer avec l’utilisateur. `lire(var)` attend que l’utilisateur tape une valeur et l’affecte à la variable *var*, `écrire(exp)` évalue l’expression *exp* et affiche sa valeur.

3 Traitements conditionnels

```
Algorithme EquationPremierDegre
(* Résoud une équation du premier degré, de type a x = b *)

var a:réel (* le coefficient a, donné par l'utilisateur *)
    b:réel (* le coefficient b, donné par l'utilisateur *)
début
   écrire("Donnez le coefficient a")
   lire(a)
   écrire("Donnez le coefficient b")
   lire(b)
   si a ≠ 0 alors
        écrire("La solution est ", b/a)
   fin si
fin
```

```
   ...
   si a ≠ 0 alors
         écrire("La solution est ", b/a)
   sinon
         écrire("Je ne sais pas faire")
   fin si
```

```
   ...
   si a ≠ 0 alors
          écrire("la solution est ", b/a)
   sinon
          si b = 0 alors
               écrire("Tout x est solution")
          sinon
               écrire("Pas de solution")
          fin si
   fin si
   ...
```

```
   ...
   si a ≠ 0 alors
         écrire("la solution est ", b/a)
   fin si
   si (a = 0) et (b = 0) alors
         écrire("Tout x est solution")
   fin si
   si (a = 0) et (b ≠ 0) alors
         écrire("Pas de solution")
   fin si
   ...
```

```
...
selon jour faire
   1 : écrire("Lundi")
   2 : écrire("Mardi")
   3 : écrire("Mercredi")
   4 : écrire("Jeudi")
   5 : écrire("Vendredi")
   6 : écrire("Samedi")
   7 : écrire("Dimanche")
   autrement : écrire("Erreur")
fin selon
...
```

```
...
si jour=1 alors
     écrire("Lundi")
sinon
     si jour=2 alors
          écrire("Mardi")
     sinon
          si jour=3 alors
               écrire("Mercredi")
          sinon
               si jour=4 alors
                    écrire("Jeudi")
               sinon
                    si jour=5 alors
                         écrire("Vendredi")
                    sinon
                         si jour=6 alors
                              écrire("Samedi")
                         sinon
                              si jour=7 alors
                                   écrire("Dimanche")
                              sinon
                                   écrire("Erreur")
                              fin si
                         fin si
                    fin si
               fin si
          fin si
     fin si
fin si
...
```

Résumé du chapitre 3

- La construction conditionnelle permet de n'exécuter une séquence d'instructions que dans le cas où une condition, représentée par une expression booléenne, est vraie.
- La construction alternative permet de choisir entre deux séquences d'instructions en fonction de la valeur d'une expression booléenne.
- Ces constructions peuvent être composées pour gérer des cas complexes. Il est alors primordial de bien utiliser l'indentation pour faire apparaître la structure de l'algorithme. La complexité peut parfois se trouver dans les expressions booléennes elles-mêmes.
- La constuction de choix permet de choisir dans un ensemble de séquences d'instructions en fonction de la valeur d'une expression.

4 Constructions itératives

```
Algorithme Compteur
(* compte de 1 à n *)

var n:entier  (* entrée par l'utilisateur *)
    i:entier  (* utilisée pour compter    *)
début
    lire(n)
    i ← 1
    tant que i≤n faire
         écrire(i)
         i ← i + 1
    fin tant que
fin
```

```
Algorithme TabledeMultiplication
(* Affiche la table de multiplication de n *)

var n:entier   (* la donnée *)
    i:entier
début
    écrire("Table de multiplication de :")
    lire(n)

    i ← 1
    répéter
        écrire(i, " × ", n, " = ", i×n)
        i ← i + 1
    jusqu'à i>10
fin
```

```
Algorithme TabledeMultiplicationBis
(* Affiche la table de multiplication de n *)

var n:entier   (* la donnée *)
    i:entier
debut
    écrire("Table de multiplication de :")
    lire(n)

    pour i de 1 à 10 faire
        écrire(i, " × ", n, " = ", i×n)
    fin pour
fin
```

Résumé du chapitre 4

- La répétition des traitements est fondamentale en algorithmique. Elle peut correspondre à un dialogue ou à la réalisation progressive d'un calcul répétitif.
- Cette répétition peut être exprimée par des constructions itératives, aussi appelées *boucles*. Chaque exécution d'un traitement dans une boucle est appelée une *itération*.
- La construction `tant que` permet de répéter un traitement *tant qu*'une *condition de continuation* est vraie.
- La construction `répéter` permet de répéter un traitement *jusqu'à* ce qu'une *condition d'arrêt* devienne vraie.
- La construction `pour` est à utiliser quand on connait à l'avance le nombre d'itérations nécessaires.

5 Conception des algorithmes

Exemple détaillé : calcul de l'épargne

On cherche à calculer le nombre d'années nécessaires pour qu'une somme de 1000 € placée à un taux annuel de 8,25 % ait atteint le double de sa valeur. Sans chercher de solution directe, on peut répéter des multiplications par 1.0825 un certain nombre de fois... et prévoir que le calcul s'arrête dès que le montant est supérieur à 2000.

Un algorithme ne s'écrit pas directement ligne par ligne "de haut en bas". Il faut réfléchir au problème posé et décider point par point de la solution que l'on va mettre en œuvre. Une fois que la solution est claire, on peut rédiger proprement l'algorithme.

Voici la résolution proposée :

1. Nous allons avoir besoin d'une variable qui stockera le montant du compte après un certain nombre d'années, appelons la `montant`, elle aura le type `réel`. Il nous faut également une variable comptant le nombre d'années écoulées, puisque c'est ce que nous cherchons. Elle sera de type entier, appelons la `nb_années`.

   Nous savons donc que notre environnement comprendra les variables suivantes :

   `montant` type réel, montant du compte à un moment donné

   `nb_années` type entier, nombre d'années écoulées

   Peut-être y en aura-t-il d'autres, nous nous en rendrons compte plus tard le cas échéant.

2. Le principe de l'algorithme est de simuler le passage des années sur notre compte bancaire. Que se passe-t-il à chaque fois qu'une année s'est écoulée ?
   – le compte reçoit les intérêts : `montant ← montant × 1.0825`
   – évidemment, le nombre d'années écoulées augmente de 1 : `nb_années ← nb_années + 1`

   Nous venons de définir les traitements (modifications de l'environnement) qui vont être effectués de manière répétée.

3. Quel est l'état initial de l'environnement (c'est à dire, quelle est la valeur initiale de chacune des variables, avant que l'on applique les traitements répétitifs ?)
   – l'énoncé indique que le montant du compte est initialement égal à 1000 : `montant ← 1000`
   – avant de commencer, 0 années se sont écoulées : `nb_années ← 0`

4. Les traitements répétitifs ne doivent pas être appliqués indéfiniment, quand termine-t-on ?

   L'objectif est de savoir au bout de combien d'années le compte aura atteint la somme de 2000 €. On termine donc dès que `montant` est supérieur ou égal à 2000. Le résultat recherché, le nombre d'années écoulées, est alors dans la variable `nb_années`, dont on pourra afficher la valeur.

À ce stade, nous avons tous les éléments permettant de rédiger l'algorithme : l'environnement, sa valeur initiale, les traitements répétitifs à effectuer, la condition indiquant quand la répétition doit se terminer, l'emplacement du résultat.

Il nous reste à préciser comment répéter les traitements. Nous allons utiliser pour cela une construction `tant que`, rappelons que sa forme est :

```
tant que expression booléenne faire
        instructions
fin tant que
```

Les instructions sont les traitements que nous avons énoncés au point 2, reste à préciser la forme de l'expression booléenne. Les instructions sont exécutées *tant que* l'expression est vraie. Nous avons dit que nous devons terminer dès que `montant ≥ 2000`, c'est à dire que nous devons continuer tant que `montant < 2000`.

Il nous reste à écrire l'algorithme au propre, sans oublier sa spécification et en plaçant éventuellement des commentaires facilitant sa lecture. Il est également très important d'utiliser correctement l'indentation pour bien faire apparaître sa structure.

```
Algorithme Épargne
(* Calcule le nombre d'années nécessaire pour doubler son épargne *)

var montant:réel          (* le montant de l'épargne *)
    nb_années:entier      (* nb d'années écoulées    *)
début
    montant ← 1000
    nb_années ← 0
    tant que montant < 2000 faire
        montant ← montant × 1.0825
        nb_années ← nb_années + 1
    fin tant que
    écrire("Montant final: ", montant, " dans ", nb_années, " an(s)")
fin
```

Une fois l'algorithme écrit, on a intérêt à faire quelques vérifications, en particulier sur la terminaison des traitements répétitifs. Peut-on se convaincre que cette répétition ne risque pas de durer indéfiniment ?

Ici on voit que la répétition se termine quand `montant` est supérieur ou égal à 2000. Sa valeur initiale est 1000, et à chaque itération elle est multipliée par 1.0825 qui est supérieur à 1. Elle augmente donc strictement à chaque fois et atteindra ou dépassera forcément la valeur 2000 après un nombre fini d'itérations.

Exécution Essayons maintenant d'exécuter l'algorithme. Nous allons pour cela faire une trace des variables. Nous nous intéressons en particulier à leur valeur avant chaque itération, au moment où la condition de continuation est évaluée.

juste avant itération n^o	`montant`	`nb_années`
1	1000.00	0
2	1082.50	1
3	1171.81	2
4	1268.48	3
5	1373.13	4
6	1486.41	5
7	1609.04	6
8	1741.79	7
9	1885.49	8
10	2041.04	9

Les 2 instructions contenues dans la construction `tant que` sont exécutées 9 fois. Leur dernière exécution commence avec `montant`=1885.49 et se termine avec `montant`=2041.04. Au dernier test de la condition de continuation, `montant` vaut 2041.04 et `nb_années` vaut 9. La dixième itération n'a pas lieu, on passe à la suite et l'algorithme affiche le message :

```
Montant final: 2041.04 dans 9 an(s)
```

Résumé de la démarche Pour écrire un algorithme il faut :

– déterminer quelles vont être les données à manipuler,
– déterminer les traitements que ces données vont subir.

Les deux aspects sont complètement liés. Ici nous avons d'abord défini l'environnement (point 1), mais pour cela nous avions déjà en tête "en gros" les traitements que nous allions devoir effectuer. Parfois, en déterminant les traitements à effectuer nous nous rendrons compte que nous devons ajouter des variables à l'environnement, ou y apporter d'autres modifications.

Quand des traitements répétitifs sont en jeu (ce qui sera presque toujours le cas), il faut commencer par bien les identifier puis régler la question de la valeur initiale des variables (l'état initial de l'environnement), et de l'arrêt de la répétition.

```
Algorithme CalculetteEuro
(* calculette Francs Euros *)
var francs:réel (* une variable pour les montants successifs en francs *)

début
   écrire("Donner un montant en Francs (0 pour finir)")
   lire(francs)
   répéter
      écrire("La valeur en Euros est : ", francs / 6.55957)
      écrire("Donner un montant en Francs (0 pour finir)")
      lire(francs)
   jusqu'à francs = 0
   écrire("Au revoir")
fin
```

```
Algorithme CalculetteEuro
(* calculette Francs Euros *)
var francs:réel (* une variable pour les montants successifs en francs *)

début
   écrire("Donner un montant en Francs (0 pour finir)")
   lire(francs)
   tant que francs ≠ 0 faire
      écrire("La valeur en Euros est : ", francs / 6.55957)
      écrire("Donner un montant en Francs ")
      lire(francs)
   fin tant que
   écrire("Au revoir")
fin
```

```
Algorithme factorielle
(* calcul de la factorielle *)
var n:entier (* on veut calculer n! *)
    i:entier (* compteur de boucle *)
    f:entier (* pour le calcul *)
début
    écrire("factorielle de :")
    lire(n)
    si n ≤ 1 alors
            écrire("résultat ", 1)
    sinon
            f ← 1
            pour i de 2 à n faire
                 f ← f × i
            fin pour
            écrire("résultat ", f)
    fin si
fin
```

```
Algorithme factorielle_bis
(* calcul de la factorielle *)
var n:entier (* on veut calculer n! *)
    i:entier (* compteur de boucle *)
    f:entier (* pour le calcul *)
début
    écrire("factorielle de :")
    lire(n)
    f ← 1
    pour i de 2 à n faire
         f ← f × i
    fin pour
    écrire("résultat ", f)
fin
```

```
Algorithme dialogue_factorielle
(* calcule la factorielle des nombres entrés *)
(* jusqu'à ce qu'on entre un nb négatif      *)

var n : entier (* on calcule n! *)
    i : entier (* compteur de boucle *)
    f : entier (* pour le calcul *)
début
  écrire("factorielle de :")
  lire(n)
  tant que n ≥ 0 faire
       f ← 1
       pour i de 2 à n faire
            f ← f × i
       fin pour
       écrire("résultat ", f)
       écrire("factorielle de (<0 pour terminer) :")
       lire(n)
  fin tant que
  écrire("au revoir")
fin
```

```
Algorithme e
(* calcul approché de e *)
var som:réel     (* somme partielle *)
    i,j,n:entier (* compteurs, rang du dernier terme *)
    fi:entier    (* fact de i *)

début
   écrire("Rang du dernier terme ?")
   lire(n)
   som ← 0
   pour i de 0 à n faire
        fi ← 1
        pour j de 2 à i faire
             fi ← fi × j
        fin pour
        (* fi = fact de i *)
        som ← som + (1/fi)
   fin pour
   écrire("Approximation de e : ", som)
fin
```

```
Algorithme e
(* calcul approché de e *)
var som:réel   (* somme partielle *)
    i,n:entier (* compteur, rang du dernier terme *)
    fi:entier  (* fact de i *)

début
   écrire("Nombre de termes ?")
   lire(n)
   som ← 1
   fi ← 1
   pour i de 1 à n faire
        fi ← fi × i
        som ← som + (1/fi)
   fin pour
   écrire("Approximation de e: ", som)
fin
```

Résumé du chapitre 5

- Un algorithme ne s'écrit pas linéairement, mais se construit par raffinement à partir d'un plan et d'un brouillon.
- Les boucles sont la principale source de difficulté (et donc d'erreur et ... d'exercices !) dans un algorithme.
- Dans le cas où la boucle met en œuvre un dialogue répétitif, il faut identifier le traitement à répéter puis déterminer quand la répétition doit se terminer.
- Dans le cas où la boucle met en œuvre un calcul itératif, les instructions qui sont répétées vont modifier progressivement l'environnement à chaque itération. Pour construire correctement la boucle, il faut identifier la modification à répéter, l'état initial de l'environnement, et combien de fois ou jusqu'à quand on doit répéter le traitement (quel doit être l'état final de l'environnement).
- Après avoir écrit une boucle il faut s'astreindre à vérifier :
  - qu'elle termine bien dans tous les cas (en tout cas pour les données valides),
  - qu'elle ne termine ni trop tôt, ni trop tard,
  - sur un exemple, que le résultat est bien celui attendu.
- Il n'est pas rare de devoir imbriquer des boucles. Il est alors primordial de ne pas essayer de les écrire simultanément. Il faut au contraire diviser la difficulter en les concevant séparément.

6 Procédures et fonctions

```
Algorithme dessine_un_carré
(* dessine un carré de côté 50 dont le coin
   inférieur gauche est en position (10,10) *)
début
    déplacer_crayon(10, 10)
    tracer(50, 0)
    tracer(0, 50)
    tracer(-50, 0)
    tracer(0, -50)
fin
```

```
Algorithme dessine_4_carrés
(* dessine les 4 carrés ... *)
début
    déplacer_crayon(10, 10)
    tracer(50, 0)
    tracer(0, 50)
    tracer(-50, 0)
    tracer(0, -50)
    déplacer_crayon(100, 10)
    tracer...
    ...
    déplacer_crayon(10, 100)
    tracer...
    ...
    déplacer_crayon(100, 100)
    tracer...
    ...
fin
```

```
Algorithme dessine_4_carrés_bis
(* dessine les 4 carrés ... *)

   procédure dessiner_carré
   (* dessine un carré de côté 50 à la position courante *)
   début
       tracer(50, 0)
       tracer(0, 50)
       tracer(-50, 0)
       tracer(0, -50)
   fin

début
   déplacer_crayon(10, 10)
   dessiner_carré
   déplacer_crayon(100, 10)
   dessiner_carré
   déplacer_crayon(10, 100)
   dessiner_carré
   déplacer_crayon(100, 100)
   dessiner_carré
fin
```

```
procédure dessiner_carré(côté:entier)
(* dessine un carré de taille côté à la position courante *)
début
     tracer(côté, 0)
     tracer(0, côté)
     tracer(-côté, 0)
     tracer(0, -côté)
fin
```

```
Algorithme dessine_4_carrés_ter
(* dessine les 4 carrés ... *)

   procédure dessiner_carré(côté:entier; posx,posy:entier)
   (* dessine un carré de taille côté à la position (posx,posy) *)
   début
      déplacer_crayon(posx, posy)
      tracer(côté, 0)
      tracer(0, côté)
      tracer(-côté, 0)
      tracer(0, -côté)
   fin

début
   dessiner_carré(50, 10, 10)
   dessiner_carré(50, 100, 10)
   dessiner_carré(50, 10, 100)
   dessiner_carré(50, 100, 100)
fin
```

```
Algorithme test_paramètres
var x,y:entier

   procédure proc_test(a,b:entier)
   début
       écrire("a=", a, " b=", b)
       a ← a + b
       écrire("a=", a, " b=", b)
   fin

début
   x ← 4
   y ← 6
   écrire("x=", x, " y=", y)
   proc_test(x,y)
   écrire("x=", x, " y=", y)
fin
```

```
procédure échange(var a,b:entier)
(* échange les valeurs de a et b *)
var tmp:entier
début
    tmp ← a
    a ← b
    b ← tmp
fin
```

```
Algorithme factorielle
(* calcul de la factorielle *)
var n:entier (* on veut calculer n! *)
    i:entier (* compteur de boucle *)
    f:entier (* pour le calcul *)
début
    écrire("factorielle de :")
    lire(n)
    f ← 1
    pour i de 2 à n faire
         f ← f × i
    fin pour
    écrire("résultat ", f)
fin
```

```
procédure factorielle(n:entier; var fn:entier)
(* place n! dans fn *)
var i:entier
début
    fn ← 1
    pour i de 2 à n faire
         fn ← fn × i
    fin pour
fin
```

```
Algorithme dialogue_factorielle
(* calcule la factorielle des nombres entrés *)
(* jusqu'à ce qu'on entre un nb négatif      *)

var n : entier (* on calcule n! *)
    f : entier (* résultat *)

    procédure factorielle(n:entier; var fn:entier)
    (* place n! dans fn *)
    var i:entier
    début
       fn ← 1
       pour i de 2 à n faire
            fn ← fn × i
       fin pour
    fin

début
   écrire("factorielle de :")
   lire(n)
   tant que n ≥ 0 faire
       factorielle(n, f)
       écrire("résultat ", f)
       écrire("factorielle de (<0 pour terminer) :")
       lire(n)
   fin tant que
   écrire("au revoir")
fin
```

```
Algorithme dialogue_factorielle
(* calcule la factorielle des nombres entrés *)
(* jusqu'à ce qu'on entre un nb négatif      *)

var n : entier (* on calcule n! *)
    f : entier (* résultat *)

    fonction factorielle(n:entier):entier
    (* retourne valeur de n! *)
    var i:entier
        f:entier
    début
       f ← 1
       pour i de 2 à n faire
            f ← f × i
       fin pour
       retour(f)
    fin

début
   écrire("factorielle de (<0 pour terminer) :")
   lire(n)
   tant que n ≥ 0 faire
       f ← factorielle(n)
       écrire("résultat ", f)
       écrire("factorielle de (<0 pour terminer) :")
       lire(n)
   fin tant que
   écrire("au revoir")
fin
```

Résumé du chapitre 6

- Les procédures et fonctions permettent d'abstraire une séquence d'instructions en la désignant par un nom. Ce sont en quelque sorte des "sous algorithmes" que l'on peut définir puis utiliser à l'intérieur d'un algorithme, de façon à améliorer sa lisibilité.
- Une procédure s'utilise comme une instruction alors qu'une fonction produit une valeur et s'utilise donc comme une expression.
- Les procédures et fonctions peuvent avoir des *paramètres d'entrée* dans lesquels elles trouveront leurs données, et des *paramètres de sortie* dans lesquels elles fourniront des résultats.
- Une fonction n'aura généralement pas de paramètre de sortie car elle produit un résultat qui sera sa *valeur de retour*.
- Les paramètres déclarés dans la définition d'une procédure ou fonction sont appelés les *paramètres formels*. Les paramètres fournis dans chaque appel sont les *paramètres effectifs*.
- Il existe deux modes de passage des paramètres. Dans le *passage par valeur*, les paramètres effectifs sont des expressions dont les valeurs sont copiées dans les paramètres formels. Dans le *passage par variable* (ou *référence*) les paramètres effectifs sont des variables, dont les paramètres formels deviennent des alias. Les paramètres d'entrée seront généralement passés par valeur et ceux de sortie toujours par référence.
- Une procédure ou fonction peut définir des *variables locales* qui constitueront, avec les paramètres formels, son environnement personnel. Ces variables n'existent que pendant l'exécution de la procédure ou fonction.

7 Analyse descendante

Les procédures et fonctions sont des constructions puissantes permettant de structurer un algorithme de façon à faciliter d'une part sa lecture et sa compréhension, d'autre part sa conception, en évitant les erreurs. Elles sont en particulier très utiles pour mettre en œuvre une méthode d'*analyse descendante* dans la résolution d'un problème algorithmique. C'est ce que nous allons tenter d'illustrer dans ce chapitre.

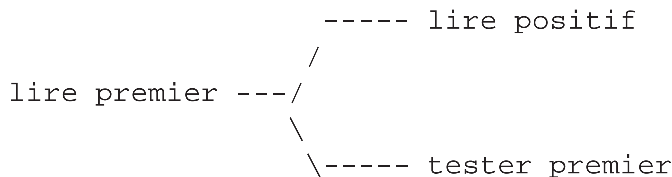
7.1 Un problème complexe

Je veux écrire un algorithme qui demande à l'utilisateur de proposer des nombres entiers positifs juqu'à ce qu'il en trouve un de premier. Rappelons qu'un nombre premier n'a que deux diviseurs, 1 et lui-même et que 1 n'est pas un nombre premier.

L'algorithme devra, selon le nombre proposé par l'utilisateur :
- redemander un nombre s'il n'est pas positif,
- donner une explication adaptée (préciser la liste des diviseurs) s'il n'est pas premier, et redemander,
- féliciter l'utilisateur si le nombre est premier.

Ce problème étant relativement complexe, il est peu raisonnable d'essayer de le résoudre "d'un bloc". Essayons plutôt de le décomposer en sous-problèmes moins complexes. Il y a dans ce problème plusieurs répétitions :
- la repétition de saisie d'un nombre jusqu'à ce que l'utilisateur en donne un positif,
- la répétition de l'affichage des diviseurs d'un nombre si le nombre proposé n'est pas premier,
- la répétition de demandes de nombres à l'utilisateur jusqu'à ce qu'il en propose un qui soit premier.

Les deux premières répétitions ne sont en fait que des traitements "internes" à la troisième. Ainsi le problème global, que l'on appelera "lire premier" comprend deux sous problèmes, "lire positif" et "tester premier". Cette décomposition peut être illustrée par l'arbre des sous problèmes suivant :



En laissant de côté provisoirement les deux sous problèmes, on peut alors écrire un squelette de l'algorithme :

```
Algorithme NombrePremier
(* Demande un nombre positif, jusqu'à ce que l'utilisateur donne un nombre premier *)
var nombre:entier   (* le nombre demandé *)
    premier:booléen (* vrai ssi nombre est premier *)

début
   écrire("Pour trouver un nombre premier ...")
   répéter
       lire un entier strictement positif et l'appeler nombre
       tester si nombre est premier, afficher éventuellement ses diviseurs,
       et enregistrer le resultat (vrai/faux) sous le nom premier
   jusqu'à premier
   écrire("Bravo, vous avez trouvé un nombre premier : ", nombre)
fin
```

À ce niveau de l'analyse, on a déjà choisi les variables nécessaires à l'expression du traitement global, qui est une répétition de deux traitements particuliers qui pourront être précisés par la suite. Ces sous-problèmes sont :

1. Lire un entier strictement positif et l'appeler `nombre`,
2. Tester si `nombre` est premier, afficher éventuellement ses diviseurs, et enregistrer le resultat sous le nom `premier`.

Ces deux sous-problèmes peuvent être résolus indépendamment l'un de l'autre, dès que l'on a choisi quelles informations sont utilisées et/ou produites par chacun des sous-problèmes.

Premier sous-problème Pour continuer l'analyse, on étudie le premier sous-problème. Il consiste en une répétition de saisies de nombres tant que l'utilisateur entre des nombres négatifs ou nuls. Ce traitement doit se terminer avec une valeur strictement positive pour la variable `nombre`. Nous avons déjà vu ce schéma de saisie d'une valeur devant respecter une condition donnée.

```
répéter
    écrire("Donner un entier strictement positif : ")
    lire(nombre)
jusqu'à nombre > 0
```

Si l'on veut afficher un message différent à partir de la deuxième invite (pour insister sur l'erreur commise), on utilisera une construction `tant que` :

```
écrire("Donner un entier strictement positif : ")
lire(nombre)
tant que nombre ≤ 0 faire
    écrire("Donner un entier STRICTEMENT POSITIF : ")
    lire(nombre)
fin tant que
```

Second sous-problème Le second sous-problème est plus complexe, car pour savoir si un nombre est premier il faut tester s'il n'a que 2 diviseurs, 1 et lui-même. Pour cela, on peut essayer de diviser le nombre par tous les entiers compris entre 2 et sa racine carrée[1]. Si on en trouve un, alors le nombre n'est pas premier. Il nous faudra donc calculer la racine du nombre, puis tester ses diviseurs. Nous posons comme sous-problèmes le calcul de la racine et le test de divisibilité d'un nombre par un autre. L'arbre des sous-problèmes raffiné est maintenant :

```
                  ----- lire positif
                 /
lire premier ---/
                \                          ---- calcul racine
                 \----- tester premier ---/
                                          \ ---- test divisibilité
```

En supposant que nous savons résoudre ces deux sous-problèmes, comment tester si un nombre est premier ?

1. si le nombre est égal à 1, alors il n'est pas premier.
2. sinon, il faut chercher tous les diviseurs du nombre entre 2 et sa racine, et les afficher s'ils existent. S'il en existe au moins un, le nombre n'est pas premier, sinon il l'est.

Comment tester tous les diviseurs ? Puisque nous voulons tous les tester, nous pouvons utiliser une boucle pour, avec la variable de boucle représentant le diviseur à tester. Le test de divisibilité nous dira si oui on non (vrai ou faux, c'est une valeur booléenne), le nombre est divisible par le diviseur courant. Au final, la variable `premier` devra avoir la valeur vrai si on n'a trouvé aucun diviseur et faux dans le cas contraire. Le fragment d'algorithme est le suivant :

```
   si nombre = 1 alors premier ← faux
   sinon
       calculer la racine de nombre et l'enregistrer sous le nom r
       premier ← vrai
       pour d de 2 à r faire
            tester si nombre est divisible par d, enregistrer sous le nom estdivpard
(a)         premier ← premier et non estdivpard
       fin pour
   fin si
```

L'idée est qu'en cherchant des diviseurs, on va essayer d'invalider l'hypothèse que le nombre est premier. Tant que l'on n'en a pas trouvé, on le considère comme premier. Si `estdivpard` est vrai au moins une fois, alors `premier` restera "bloqué" sur la valeur faux. Si on ne trouve aucun diviseur, `premier` gardera la valeur vrai.

On aurait pu écrire la ligne (a) d'une façon différente mais strictement équivalente :

[1] Si un nombre n a un diviseur x>$\sqrt{n}$, il en a forcément un autre y tel que x×y=n et donc y<$\sqrt{n}$.

```
si estdivpard alors
        premier ← faux
fin si
```

Une autre possibilité serait de compter le nombre de diviseurs et de fixer la valeur de `premier` après la terminaison de la boucle, selon que le nombre de diviseurs a été trouvé nul ou pas.

On aurait aussi pu utiliser une boucle `tant que` pour s'arrêter au premier diviseur, mais ici l'énoncé nous demande de tous les afficher.

Les sous-problèmes à traiter par la suite sont :
- calculer la racine de `nombre` et l'enregistrer sous le nom `r`
- tester si `nombre` est divisible par `d`, enregistrer le résultat sous le nom `estdivpard`

Premier sous-sous-problème Le premier peut être résolu par le fragment d'algorithme suivant, qui calcule la partie entière de la racine carrée d'un nombre entier strictement positif (le plus grand nombre entier dont le carré est inférieur ou égal à nombre) :

```
r ← 1
tant que (r+1)×(r+1) ≤ nombre faire
    r ← r + 1
fin tant que
```

Cet algorithme mérite qu'on s'y attarde quelques instants. Il "a l'air" de fonctionner. Pour s'en convaincre définitivement, il suffit en fait de remarquer qu'après la dernière itération :

1. (r+1)×(r+1) > nombre
2. r × r ≤ nombre (puisqu'on n'est pas sorti de la boucle à l'itération précédente)

Le premier point revient à dire que `r+1` est strictement supérieur à la racine carrée de `nombre`. Le deuxième que `r` est inférieur ou égal à la racine carré de `nombre`. Comme il n'existe pas d'entier entre `r` et `r+1`, `r` est la partie entière de la racine carrée de `nombre`.

Dans le raisonnement précédent, nous avons supposé qu'il y avait eu au moins deux tests, c'est à dire une itération. Voyons le cas où il n'y a eu qu'un test de la condition, i.e. 0 itérations. Cela se produit (2×2>`nombre`) pour `nombre` = 1, 2 ou 3. On finit alors avec `r` = 1, ce qui est correct.

Second sous-sous-problème Le dernier sous-problème consiste à tester si le reste de la division de `nombre` par `i` est nul, et si oui à afficher les deux diviseurs de `nombre` qui viennent d'être trouvés. Le résultat du test sera placé dans la variable `estdivpard` :

```
estdivpard ← (nombre mod d) = 0
si estdivpard alors
    écrire(nombre, " est divisible par ", d, " et par ", nombre div d)
fin si
```

À l'issue de cette analyse, il faut remettre en place les morceaux d'algorithmes pour obtenir une solution globale au problème complet. On obtient ainsi une imbrication correcte des différentes répétitions contenues dans cet algorithme.

7.2 Première solution

Pour obtenir cette solution, on a remis bout à bout tous les fragments d'algorithmes écrits précédemment, et reconstitué l'environnement complet (figure 2).

Bien que la démarche pour obtenir cet algorithme ait été rigoureuse, il est difficile, à posteriori, de le comprendre, si l'on n'a pas suivi toutes les étapes de la conception, car on ne "voit" plus les différentes parties de l'algorithme tel qu'il a été conçu. Il faudrait pouvoir faire apparaître cette décomposition dans l'algorithme final, ce que nous avons tenté de faire en utilisant des commentaires.

```
Algorithme NombrePremierVersion1
(* Demande un nombre positif, jusqu'à ce que l'utilisateur donne un nombre premier *)
var nombre:entier      (* le nombre demandé *)
    premier:booléen    (* vrai ssi nombre est premier *)
    estdivpard:booléen (* vrai ssi nombre est divisible par d *)
    d:entier           (* compteur de boucle pour le test des diviseurs *)
    r:entier           (* partie entière de la racine carrée de nombre *)

début
   écrire("Pour trouver un nombre premier ...")
   répéter
       écrire("Donner un entier strictement positif : ") (* début lire positif *)
       lire(nombre)
       tant que nombre ≤ 0 faire
            écrire("Donner un entier, qui doit être strictement positif : ")
            lire(nombre)
       fin tant que                                      (* fin lire positif *)

       si nombre = 1 alors                               (* début tester si premier *)
            premier ← faux
       sinon
            premier ← vrai

            r ← 1                                             (* début calcul racine *)
            tant que (r+1)×(r+1) ≤ nombre faire
                 r ← r + 1
            fin tant que                                      (* fin calcul racine *)

            pour d de 2 à r faire
                 estdivpard ← (nombre mod d) = 0              (* début test divisible *)
                 si estdivpard alors
                      écrire(nombre, " est divisible par ", d, " et par ", nombre div d)
                 fin si                                       (* fin test divisible *)

                 premier ← premier et non estdivpard
            fin pour
       fin si                                            (* fin tester si premier *)
   jusqu'à premier
   écrire("Bravo, vous avez trouvé un nombre premier : ", nombre)
fin
```

FIG. 2 – Version initiale

7.3 Utilisation de procédures et fonctions

Dans cette version (figure 3), nous avons créé une fonction pour chacun des sous problèmes. Cela permet de faire apparaître très clairement dans l'algorithme la décomposition que nous avons utilisée.

Les fonctions utilisées sont les suivantes :

- `lire_positif` : pas de paramètre, retourne un entier
- `teste_premier` : paramètre d'entrée, retourne un booléen
  - `racine` : paramètre d'entrée, retourne entier
  - `divisible_par_d` : 2 paramètres d'entrée, retourne booléen

On note l'imbrication, qui permet de créer des fonctions "locales" à une fonction.

De plus, l'environnement "privé" de chaque sous problème est défini localement à la procédure ou fonction correspondante. Cela apporte plus de clarté et évite tout conflit de nom de variable. Ainsi, la variable locale nommée `premier` dans `teste_premier` ne peut pas être confondue avec la variable globale de même nom.

Cette version est plus longue à cause des diverses déclarations et des lignes blanches destinées à aérer le code mais elle est bien plus claire.

De plus on pourrait facilement remplacer une des fonctions par une autre vérifiant la spécification. Par exemple le

calul de la racine carrée pourrait être effectué à l'aide d'un algorithme plus performant, sans rien changer au reste de l'algorithme, ce qui est un bel exemple de modularité.

7.4 Quelques mots sur l'abstraction

> « Le second [précept], de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre. »
>
> René Descartes, Discours de la méthode, seconde partie, 1637.

Nous venons de faire une "analyse descendante" pour résoudre un problème algorithmique (relativement) complexe. Le principe est de décomposer le problèmes en sous-problèmes plus simples. Cette méthode est l'application à l'algorithmique d'une méthode de gestion de la complexité des systèmes, applicable dans beaucoup de domaines.

On décompose un système en composants dont on abstrait le fonctionnement interne (on les considère comme des "boîtes noires"). Si les composants sont encore trop complexes, on les divise en sous-composants et ainsi de suite. Chaque composant n'a pas besoin de savoir comment fonctionnent ses propres composants. On procède hiérarchiquement en décomposant ensuite chacune des boites noires et ainsi de suite jusqu'à arriver à des composants "triviaux".

7.5 Usage

L'usage des procédures et fonctions permet d'améliorer la clarté des algorithmes et rend plus facile l'utilisation de l'analyse descendante pour résoudre un problème complexe.

On peut également constituer des *bibliothèques* de procédures et fonctions. Ce sont des collections d'algorithmes résolvant des problèmes "classiques", qui peuvent être utilisés dans la résolution d'un problème complexe. Par exemple, le calcul d'une racine carrée intervient probablement comme sous-problème d'un grand nombre de problèmes mathématiques.

Les procédures et fonctions facilitent également le partage du travail entre plusieurs personnes. Une fois que le problème est décomposé en plusieurs sous-problème, chaque sous-problème peut être résolu par une personne différente rédigeant une procédure. Il suffit de s'être accordé sur les spécifications et les paramètres et valeurs de retour.

Un autre intérêt est dans la maintenance de gros systèmes logiciels. Une petite partie du système peut être modifiée sans avoir d'impact sur l'ensemble du système. Dans notre algorithme de dessin des quatre carrés, si les commandes de dessin changent avec un nouveau périphérique de dessin, seule la procédure de dessin d'un carré doit être réécrite.

Enfin, nous avons mentionné comment on pourrait facilement remplacer dans notre exemple la fonction qui calcule la racine carrée.

Dans ce cours... Désormais on exprimera la plupart de nos algorithmes sous forme de fonction ou de procédure.

On n'utilisera donc généralement plus l'instruction `lire` pour obtenir les données, celles-ci seront des paramètres de la fonction. De même, on n'utilisera généralement plus l'instruction `écrire` pour afficher le résultat, celui-ci sera fourni comme valeur de retour de la fonction (comme nous l'avons déjà fait pour la factorielle).

7.6 Résumé

- L'analyse descendante consiste à décomposer un problème en sous-problèmes plus simples, qui seront eux même décomposés si nécessaire jusqu'à ce que l'on obtienne des sous-problèmes faciles à résoudre.
- Cette méthode est classique en algorithmique mais est d'une portée bien plus générale.
- Les procédures et fonctions facilitent grandement la mise en œuvre de cette méthode puisque la formulation finale de l'algorithme peut être calquée sur cette décomposition. Cela facilite à la fois la conception d'un nouvel algorithme et la compréhension d'un algorithme existant.

```
Algorithme NombrePremierVersion2
(* Demande un nombre positif, jusqu'à ce que l'utilisateur donne un nombre premier *)
var nombre:entier      (* le nombre demandé *)
    premier:booléen    (* vrai ssi nombre est premier *)

    fonction lire_positif:entier
    (* lit un entier jusqu'à ce qu'il soit positif *)
    var n:entier
    début
        écrire("Donner un entier strictement positif : ")
        lire(n)
        tant que n ≤ 0 faire
            écrire("Donner un entier, quoi doit être strictement positif :")
            lire(n)
        fin tant que
        retour(n)
    fin

    fonction racine(n:entier):entier
    (* calcule la partie entière de la racine de n (n>0) *)
    var p:entier
    début
        p ← 1
        tant que (p+1)×(p+1) ≤ n faire
             p ← p + 1
        fin tant que
        retour(p)
    fin

    fonction divisible_par_i(n,i:entier):booléen
    (* n divisible par i ? si oui affiche diviseurs *)
    début
        si (n mod i) = 0 alors
             écrire(n, " est divisible par ", i, " et par ", n div i)
             retour(vrai)
        sinon
             retour(faux)
        fin si
    fin

    fonction teste_premier(n:entier):booléen
    (* n est premier ? affiche diviseurs le cas échéant *)
    var i:entier
        premier:booléen
    début
        si n=1 alors retour(faux)
        sinon
            premier ← vrai
            pour i de 2 à racine(n) faire
                premier ← premier et non divisible_par_i(n,i)
            fin pour
            retour(premier)
        fin si
    fin

début
    écrire("Pour trouver un nombre premier ...")
    répéter
         nombre ← lire_positif
         premier ← teste_premier(nombre)
    jusqu'à premier

    écrire("Bravo, vous avez trouvé un nombre premier : ", nombre)
fin
```

FIG. 3 – Version avec fonctions

8 Types complexes : tableaux et articles

```
fonction moyenne(notes:tnotes):réel
(* calcule la moyenne des notes *)
var i:entier
    som:réel
début
    som ← 0
    pour i de 1 à nbélèves faire
         som ← som + notes[i]
    fin pour
    retour(som/nbélèves)
fin
```

```
fonction est_présent(notes:tnotes; x:réel):booléen
(* x présent dans notes ? *)
var i:entier
    trouvé:booléen
début
    trouvé ← faux
    pour i de 1 à nbélèves faire
         si notes[i] = x alors trouvé ← vrai fin si
    fin pour
    retour(trouvé)
fin
```

```
fonction est_présent(t:tnotes; x:réel):booléen
(* x présent dans t ? *)
var i:entier
    trouvé:booléen
début
    trouvé ← faux
    i ← 1
    tant que non trouvé et (i ≤ nbélèves) faire
         trouvé ← t[i] = x
         i ← i + 1
    fin tant que
    retour(trouvé)
fin
```

```
procédure affiche_par_colonne(notes:tnotes1A)
(* affiche toutes les notes pour chaque élève *)
var élève, matière:entier
début
    pour élève de 1 à nbélèves faire
         écrire("élève n° ", élève)
         pour matière de 1 à nbmat faire
              écrire(notes[matière, élève])
         fin pour
         écrire(findeligne)
    fin pour
fin
```

```
procédure affiche_par_ligne(notes:tnotes1A)
(* affiche toutes les notes pour chaque matière *)
var élève, matière:entier
début
    pour matière de 1 à nbmat faire
         écrire("matière n° ", matière)
         pour élève de 1 à nbélèves faire
              écrire(notes[matière, élève])
         fin pour
         écrire(findeligne)
    fin pour
fin
```

```
procédure trisélection(var t:tableau[1..n] d'entiers)
(* trie le tableau t *)
var i,min:entier
    q:entier
début
    pour i de 1 à n-1 faire
         recherche le plus petit élément de t[i..n]
         et place son indice dans min

         q ← t[min]
         t[min] ← t[i]
         t[i] ← q
    fin pour
fin
```

```
procédure trisélection(var t:tableau[1..n] d'entiers)
(* trie le tableau t *)
var i,j,min:entier
    q:entier
début
    pour i de 1 à n-1 faire
        min ← i
        pour j de i+1 à n faire
               si t[j]<t[min] alors min ← j fin si
        fin pour
        (* t[min] minimum de t[i..n] *)

        q ← t[min]
        t[min] ← t[i]
        t[i] ← q
    fin pour
fin
```

```
type élève = article
                 nom : chaîne
                 notes : tableau[1..nbmat] de réels
                 redoublant : booléen
             fin article

     complexe = article (* nombre complexe *)
                    re : réel  (* partie réelle *)
                    im : réel  (* partie imaginaire *)
                fin article
```

```
type point = article
                 x:entier
                 y:entier
             fin article

     fenêtre = article
                   pig:point  (* point inférieur gauche *)
                   psd:point  (* point supérieur droit  *)
               fin article

var f:fenêtre
    p1,p2:point
début
    p1.x ← 5
    p1.y ← 4
    p2 ← p1
    f.pig ← p1
    f.psd.x ← 10
    f.psd.y ← 23
...
```

Résumé du chapitre 8

- En plus des types de base disponibles dans le langage, on peut créer des *types complexes* dont les valeurs sont formées de plusieurs éléments juxtaposés.
- Un tableau est une séquence d'éléments ayant tous le même type, chacun étant identifié par un indice qui est sa position dans le tableau. Le nombre d'éléments est fixé à la déclaration du tableau.
- Un article est un agrégat d'éléments appelés *champs* chacun pouvant avoir un type différent. Chaque champ est identifié par son nom et accessible par la notion pointée `nomarticle.nomchamp`.
- Pour clarifier un algorithme, on peut déclarer des *constantes* qui auront une valeur fixée durant toute l'éxécution.
- Toujours pour clarifier et alléger les notations, les déclarations de type permettent de donner un nom à un type complexe.

9 Structures de données pile et file

```
fonction créer_pile_vide:pile
(* retourne une pile vide *)

fonction est_pile_vide(p:pile):booléen
(* p est vide ? *)

procédure empiler(var p:pile; e:élt)
(* ajoute l'élément e au sommet de p *)

fonction dépiler(var p:pile):élt
(* retire l'élément au sommet de la pile et retourne sa valeur *)
(* p ne doit pas être vide ! *)
```

```
  const max=200
  type  pile=article
                sommet:entier      (* indice de l'élément sommet *)
                tab:tableau[1..max] de élt
             fin article

  fonction créer_pile_vide:pile
  (* crée une pile vide *)
  var p:pile
  début
       p.sommet ← 0
       retour(p)
  fin

  fonction est_pile_vide(p:pile):booléen
  (* p est vide ? *)
  début
      retour(p.sommet = 0)
  fin

  procédure empiler(var p:pile; e:élt)
  (* empile e sur p *)
  début
      p.sommet ← p.sommet + 1
      p.tab[p.sommet] ← e
  fin

  fonction dépiler(var p:pile):élt
  (* retire l'élément au sommet de la pile et retourne sa valeur *)
  début
      p.sommet ← p.sommet - 1
      retour(p.tab[p.sommet+1])
  fin
```

```
fonction créer_file_vide:file
(* retourne une file vide *)

fonction est_file_vide(f:file):booléen
(* f est vide ? *)

procédure enfiler(var f:file; e:élt)
(* ajoute e en queue de f *)

fonction défiler(var f:file):élt
(* retire tête de f et renvoie sa valeur *)
```

Réalisation naïve

```
type file=article
             tab:tableau[1..max] de élt
             longueur:entier
          fin article

fonction créer_file_vide:file
(* retourne une file vide *)
var f:file
début
     f.longueur ← 0
     retour(f)
fin

fonction est_file_vide(f:file):booléen
(* f est vide ? *)
début
     retour(f.longueur = 0)
fin

procédure enfiler(var f:file; e:élt)
(* ajoute e en queue de f *)
début
     f.longueur ← f.longueur + 1
     f.tab[f.longueur] ← e
fin

fonction défiler(var f:file):élt
(* retire tête de f et renvoie sa valeur *)
var e:élt
    i:entier
début
     e ← f.tab[1]
     pour i de 2 à f.longueur faire
          f.tab[i-1] ← f.tab[i]
     fin pour
     f.longueur ← f.longueur - 1
     retour(e)
fin
```

Réalisation un peu moins naïve

```
type file = article
                tab:tableau[1..max] de élt
                début:entier     (* indice du premier élément *)
                fin:entier       (* indice du dernier élément *)
            fin article

fonction créer_file_vide:file
(* retourne une file vide *)
var f:file
début
    f.début ← 1
    f.fin ← 0
    retour(f)
fin
```

```
fonction est_file_vide(f:file):booléen
(* f est vide ? *)
début
    retour(f.début = f.fin+1)
fin

fonction défiler(var f:file):élt
(* retire tête de f et renvoie sa valeur *)
début
    f.début ← f.début + 1
    retour(f.tab[f.début-1])
fin

procédure enfiler(var f:file; e:élt)
(* ajoute e en queue de f *)
var lg:entier
début
    si f.fin=max alors
         (* on recale à gauche *)
         lg ← f.fin - f.début + 1
         pour i de 1 à lg faire
              f.tab[i] ← f.tab[i+f.début-1]
         fin pour
         f.début ← 1
         f.fin   ← lg
    fin si

    (* on ajoute l'élément *)
    f.fin ← f.fin + 1
    f.tab[f.fin] ← e
fin
```

```
procédure traitement(n:entier)
(* applique traitement à n *)
début
    écrire(n×n)      (* par exemple ... *)
fin

procédure démo_file
(* lit des entiers > 0 puis leur applique un traitement *)
var f:file
    n:entier

début
    f ← créer_file_vide
    lire(n)
    tant que n>0 faire
         enfiler(f,n)
         lire(n)
    fin tant que

    tant que non est_file_vide(f) faire
         traitement(défiler(f))
    fin tant que
fin
```

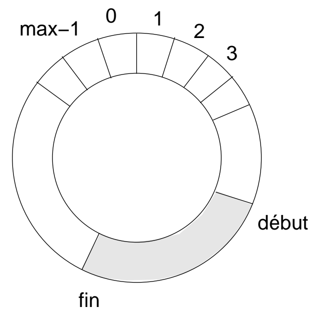


FIG. 4 – File en tableau circulaire

```
const max = 200
type file = article
               début, fin:entier
               tab:tableau[0..max-1] d'élt
            fin article
```

```
fonction créer_file_vide:file
(* retourne une file vide *)
var f:file
début
   f.début ← 1
   f.fin ← 0
   retour(f)
fin

fonction est_file_vide(f:file):booléen
(* f est vide ? *)
début
   retour(f.début = (f.fin+1) mod max)
fin

procédure enfiler(var f:file; e:élt)
(* ajoute e en fin de f *)
début
   f.fin ← (f.fin+1) mod max
   f.tab[f.fin] ← e
fin

fonction défiler(var f:file):élt
(* retire début de f et renvoie sa valeur *)
var e:élt
début
   e ← f.tab[f.début]
   f.début ← (f.début+1) mod max
   retour(e)
fin
```

```
  Algorithme test
  (* exemple pour montrer la pile des appels de procédures/fonctions *)

  var f:entier

     fonction fact(n:entier):entier
     (* calcule la factorielle de n *)
     var n:entier
         f:entier
1    début
...      ...
n    fin

     procédure dialogue
     (* calcule la factorielle des nombres entrés au clavier *)
     (* jusqu'à ce qu'un nombre négatif soit entré           *)
     var n:entier
         f:entier
1    début
2        lire(n)
3        tant que n>0 faire
4             f ← fact(n)
5             écrire(f)
6             lire(n)
7        fin tant que
8    fin

1 début
2     écrire("On commence...")
3     dialogue
4     f ← fact(12)
5     écrire(f)
6     écrire("On termine...")
7 fin
```

pile				test/4	test/4	test/4	test/4
instruction	test/1	test/2	test/3	dialogue/1	dialogue/2	dialogue/3	dialogue/4

pile	dialogue/5 test/4	dialogue/5 test/4	dialogue/5 test/4	test/4	test/4	test/4	test/4
instruction	fact/1	fact/...	fact/n	dialogue/5	dialogue/6	dialogue/7	dialogue/3

pile	test/4	... ...	test/4		test/5	test/5	test/5			
instruction	dialogue/...	...	dialogue/8	test/4	fact/1	fact/...	fact/n	test/5	test/6	test/7

Résumé du chapitre 9

- Une pile est une séquence d'éléments de même type, telle que l'on ne peut consulter, ajouter ou retirer un élément qu'à une extrémité, appelée le *sommet* de la pile.
- Une file est une séquence d'éléments de même type telle que l'on ne peut ajouter un élément qu'à une extrémité, appelée la queue, et retirer à l'autre, appelée la tête.
- Piles et files sont des *types abstraits*, définis par des *primitives* qui sont les opérations possibles. L'utilisateur (programmeur) n'a pas besoin de connaître la technique d'implémentation utilisée, mais simplement les primitives (et leur signification).
- Les piles jouent un rôle fondamental en informatique, et sont utilisées en particulier dans l'implémentation des langages de programmation.

10 La récursivité

```
fonction fact(n:entier):entier
(* calcule n! *)
début
    si n = 0 alors
        retour(1)
    sinon
        retour(n × fact(n-1))
    fin si
fin
```

```
fonction neterminepas(x:réel):réel
(* cette fonction ne termine pas ! *)
début
    si x=0 alors
         retour(1)
    sinon
         retour(neterminepas(x/2))
    fin si
fin
```

pile					
fact(0)					
fact(1)					n=1 →1× ?
fact(2)				n=2 →2× ?	n=2 →2× ?
fact(3)			n=3 →3× ?	n=3 →3× ?	n=3 →3× ?
fact(4)		n=4 →4× ?	n=4 →4× ?	n=4 →4× ?	n=4 →4× ?
	x= ?	x= ?	x= ?	x= ?	x= ?
étape	début exécution	appel de fact(4)	appel fact(3)	appel de fact(2)	appel de fact(1)

n=0 →1					
n=1 →1× ?	n=1 →1×1				
n=2 →2× ?	n=2 →2× ?	n=2 →2×1			
n=3 →3× ?	n=3 →3× ?	n=3 →3× ?	n=3 →3×2		
n=4 →4× ?	n=4 →4× ?	n=4 →4× ?	n=4 →4× ?	n=4 →4×6	
x= ?	x= ?	x= ?	x= ?	x= ?	x=24
appel de fact(0)	retour de fact(0)	retour de fact(1)	retour de fact(2)	retour de fact(3)	retour de fact(4)

```
fonction pgcd(a,b:entier):entier
(* calcule pgcd de a et b *)
var r:entier
début
    tant que b ≠ 0 faire
        r ← a mod b
        a ← b
        b ← r
    fin tant que
    retour(a)
 fin
```

```
fonction pgcd(a,b:entier):entier
(* calcule pgcd de a et b *)
début
    si b=0 alors
        retour(a)
    sinon
        retour(pgcd(b,a mod b))
    fin si
fin
```

pile		a=18 b=12 → ?	a=12 b=6 → ? a=18 b=12 → ?	a=6 b=0 → ? a=12 b=6 → ? a=18 b=12 → ?
	x= ?	x= ?	x= ?	x= ?
étape		appel de pgcd(18,12)	appel de pgcd(12,6)	appel de pgcd(6,0)

a=6 b=0 →6 a=12 b=0 → ? a=18 b=12 → ?	a=12 b=6 →6 a=18 b=12 → ?	a=18 b=12 →6	
x= ?	x= ?	x= ?	x=6
exécution de pgcd(6,0)	retour de pgcd(6,0)	retour de pgcd(12,6)	retour de pgcd(18,12)

```
procédure déplace(pieu1,pieu2:entier)
(* déplace le disque au sommet du pieu1 au sommet du pieu2 *)
début
    écrire(pieu1, " -> ", pieu2)
fin

procédure hanoi(n:entier; pa,pb,pc:entier)
(* résoud hanoi avec n disques de pa à pb *)
(* en utilisant pc                        *)
début
    si n>0 alors
        hanoi(n-1, pa, pc, pb)
        déplace(pa, pb)
        hanoi(n-1, pc, pb, pa)
    fin si
fin
```

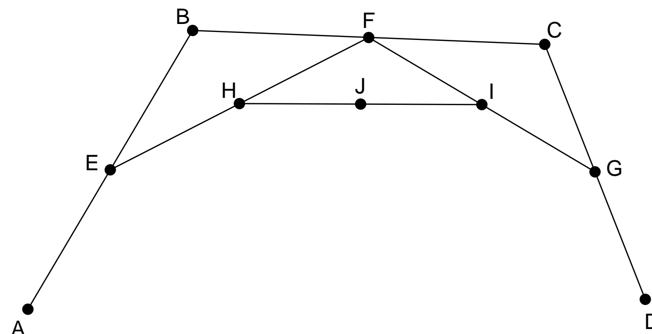
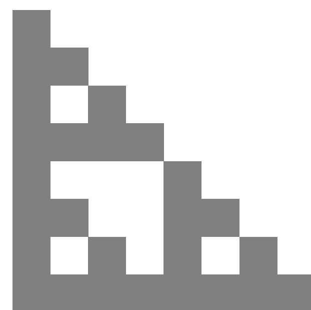


FIG. 5 – Construction d'une courbe de Bézier

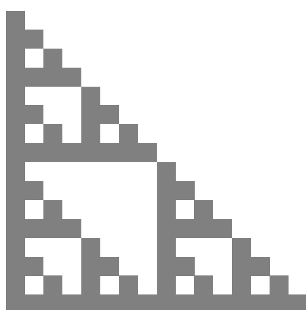
```
type point = ...

fonction milieu(a,b:point):point
(* calcule le milieu de a et b *)
...

procédure segment(a,b:point)
(* trace le segment [ab] *)
...

fonction petit(a,b,c,d:point):booléen
(* (abcd) est petit ? *)
...

procédure bézier(a,b,c,d:point)
(* trace la courbe de bézier de a,b,c,d *)
var e,f,g,h,i,j:point
début
    si petit(a,b,c,d) alors
          segment(a,d)
    sinon
          e ← milieu(a, b)
          f ← milieu(b, c)
          g ← milieu(c, d)

          h ← milieu(e, f)
          i ← milieu(f, g)
          j ← milieu(i, h)
          bézier(a,e,h,j)
          bézier(j,i,g,d)
    fin si
fin
```

(a) n=1

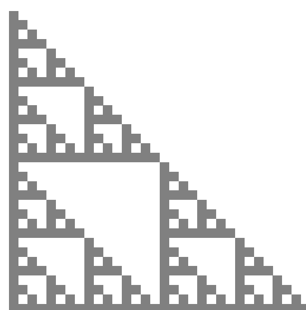
(b) n=2



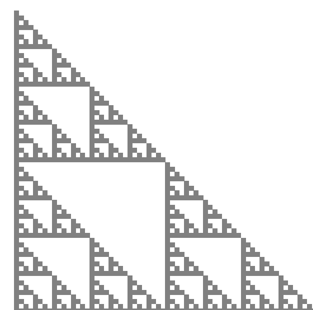
(c) n=3



(d) n=4



(e) n=5



(f) n=6

FIG. 6 – Le triangle de Sierpinski

Une figure fractale est composée de figures identiques à elle même mais à une échelle réduite. C'est un concept très lié à la récursivité. Un exemple de telle figure est le triangle de Sierpinski.
Le triangle de Sierpinski au niveau 0 est un simple carré plein, d'une taille donnée. Pour passer au niveau 1, on décompose le carré en quatre carrés de taille moitié et on "laisse de côté" le carré en haut à droite (figure 6(a)). Pour passer au niveau 2, on répète l'opération pour les trois autres carrés (figure 6(b)) et ainsi de suite.
De telles figures fractales sont très faciles à dessiner avec des algorithmes récursifs. La procédure ci-dessous dessine le triangle de Sierpinski à un niveau n donné.

```
procédure trace_carré(x,y,taille:réel)
(* trace un carré plein de côté taille avec   *)
(* le coin inférieur gauche en position (x,y) *)
...

procédure sierp(n:entier; x,y,taille:réel)
(* dessine le triangle de Sierpinski de niveau n *)
(* à la position et de taille données            *)
début
    si n=0 alors
             trace_carré(x, y, taille)
    sinon
             sierp(n-1, x,          y,          taille/2)
             sierp(n-1, x+taille/2, y,          taille/2)
             sierp(n-1, x,          y+taille/2, taille/2)
    fin si
fin
```

Résumé du chapitre 10

- Une procédure ou fonction est récursive si elle s'appelle elle même.
- Une définition récursive contient au moins un cas récursif et un cas de base où la résolution se fait sans appel récursif.
- Pour que la définition ne soit pas circulaire, il faut qu'il y ait au moins un cas de base et s'assurer que tout appel récursif se "rapproche" du ou d'un des cas de base, qui sera atteint après un nombre fini d'appels récursifs.
- L'exécution d'une procédure (ou fonction) récursive est basée sur l'utilisation d'une pile.
- La récursivité peut paraître déroutante, mais elle permet en fait d'exprimer très simplement la résolution de toute une catégorie de problèmes, si l'on adopte le bon mode de pensée.
- L'approche "diviser pour régner" consiste à diviser un problème en deux sous problèmes identiques mais n'opérant chacun que sur la moitié des données du problème initial.

11 Pointeurs

Résumé du chapitre 11

- Les variables que nous avons utilisées jusqu'à présent sont désignées chacune par un nom, qui permet d'y accéder.
- Tous les noms et donc le nombre de variables est fixé par le texte de l'algorithme.
- L'allocation dynamique permet de créer et détruire des variables en fonction des besoins, durant l'éxécution de l'algorithme.
- On peut déclarer des variables de type *pointeur sur un type t*, noté ↑`t`, qui seront destinées à contenir l'adresse d'une variable de type t.
- Si `p` est de type ↑*t*, `nouveau(p)` crée une variable de type *t*, accessible par l'intermédiaire de `p` : `p`↑.
- `libère(p)` détruit la variable pointée par `p` (`p`↑).

12 Listes chaînées

```
type cellule = article
                   val:élt
                   suiv:↑cellule
               fin article

     liste = ↑cellule
```

```
fonction tableau_vers_liste(t:tableau[1..n] de élt):liste
(* crée une liste contenant les éléments de t *)
var i:entier
    l,p:liste
début
    l ← nil
    pour i décroissant de n à 1 faire
        nouveau(p)
        p↑.val  ← t[i]
        p↑.suiv ← l
        l ← p
    fin pour
    retour(l)
fin
```

```
procédure parcours_dg(l:liste)
(* parcourt l de droite à gauche *)
var p:pile (* une pile de ↑cellule *)
début
    p ← créer_pile_vide
    tant que l ≠ nil faire
        empiler(p,l)
        l ← l↑.suiv
    fin tant que
    tant que non est_pile_vide(p) faire
        l ← dépiler(p)
        traiter(l↑.val)
    fin tant que
fin
```

```
fonction longueur(l:liste):entier
(* renvoie longueur de la liste *)
début
    si l = nil alors
        retour(0)
    sinon
        retour(1+longueur(l↑.suiv))
    fin si
fin
```

```
fonction longueur(l:liste):entier
(* longueur de l *)
var long:entier
début
    long ← 0
    tant que l ≠ nil faire
        long ← long + 1
        l ← l↑.suiv
    fin tant que
    retour(long)
fin
```

```
fonction ième_élément(l:liste; i:entier):liste
(* retourne pointeur sur le ième élément de la liste l, nil si inexistant *)
début
    si l = nil alors
        retour(nil)
    sinon
        si i = 1 alors
            retour(l)
        sinon
            retour(ième_élément(l↑.suiv, i-1))
        fin si
    fin si
fin
```

```
fonction ième_élément(l:liste; i:entier):liste
(* retourne pointeur sur le ième élément de la liste l, nil si inexistant *)
début
    tant que (i>1) et (l ≠ nil) faire
        i ← i - 1
        l ← l↑.suiv
    fin tant que
    retour(l)
fin
```

```
fonction accès_assoc(l:liste; v:élt):liste
(* retourne pointeur sur élément de l égal à v, nil si aucun *)
début
    si l = nil alors
        retour(nil)
    sinon si l↑.val = v alors
            retour(l)
        sinon
            retour(accès_assoc(l↑.suiv, v))
        fin si
    fin si
fin
```

```
fonction accès_assoc(l:liste; v:élt):liste
(* retourne pointeur sur élément de l égal à v, nil si aucun *)
var trouvé:booléen
début
    trouvé ← faux
    tant que (l ≠ nil) et non trouvé faire
        trouvé ← l↑.val = v
        si non trouvé alors       (* si trouvé, ne plus avancer !*)
            l ← l↑.suiv
        fin si
    fin tant que
    retour(l)
fin
```

```
procédure insère_tête(var l:liste; e:élt)
(* insère élt de valeur e en tête de l *)
var p:liste
début
    nouveau(p)
    p↑.val ← e
    p↑.suiv ← l
    l ← p
fin
```

```
procédure insère_n(l:liste; n:entier; e:élt)
(* insère en n>1 ième position de l la valeur e *)
var c, nc:↑cellule
début
    c ← ième_élément(l, n-1)
    (* c ≠ nil *)
    nouveau(nc)
    nc↑.val ← e
    nc↑.suiv ← c↑.suiv
    c↑.suiv ← nc
fin
```

Résumé du chapitre 12

- Une liste chaînée est une séquence d'éléments de même type, dont la taille peut varier dynamiquement et sans limite à priori.
- Chaque élément contient un pointeur vers l'élément suivant, ce qui constitue le chaînage de la liste.
- Il n'est possible d'accéder directement qu'au premier élément. Pour les éléments suivants il faut partir du premier et suivre le chaînage. Par contre il est possible, contrairement aux tableaux, de réorganiser la liste sans recopier tous les éléments.

13 Retour sur les piles et files

```
type cellule = article
                 val:élt
                 suiv:↑cellule
               fin article

     pile=↑cellule

fonction créer_pile_vide:pile
(* crée une pile vide *)
début
    retour(nil)
fin

fonction est_pile_vide(p:pile):booléen
(* p est vide ? *)
début
    retour(p=nil)
fin

procédure empiler(var p:pile; e:élt)
(* empile e sur p *)
var c:↑cellule
début
    nouveau(c)
    c↑.val ← e
    c↑.suiv ← p
    p ← c
fin

fonction dépiler(var p:pile):élt
(* retire l'élément au sommet de la pile et retourne sa valeur *)
var e:élt
    c:↑cellule
début
    e ← p↑.val
    c ← p
    p ← p↑.suiv
    libère(c)
    retour(e)
fin
```

```
type file = article
               début:↑cellule
               fin:↑cellule
            fin article

fonction créer_file_vide:file
(* retourne une file vide *)
var f:file
début
    f.début ← nil
    f.fin ← nil
    retour(f)
fin

fonction est_file_vide(f:file):booléen
(* f est vide ? *)
début
    retour(f.début = nil)   (* ou f.fin ... *)
fin

procédure enfiler(var f:file; e:élt)
(* ajoute e en queue de f *)
var c:↑cellule
début
    nouveau(c)
    c↑.val ← e
    c↑.suiv ← nil
    si f.début = nil alors  (* si la file était vide *)
         f.début ← c
    sinon
         f.fin↑.suiv ← c
    fin si
    f.fin ← c
fin

fonction défiler(var f:file):élt
(* retire tête de f et renvoie sa valeur *)
var c:↑cellule
    e:élt
début
    c ← f.début
    f.début ← c↑.suiv
    e ← c↑.val
    libère(c)
    retour(e)
fin
```

Résumé du chapitre 13

- Les listes chaînées peuvent servir à implémenter toutes sortes de structures de données. On a vu ici les piles et les files.
- Un algorithme ayant besoin d'une pile peut utiliser une réalisation basée sur un tableau ou une liste chaînée, sans aucune modification sur l'algorithme lui même. Le type *abstrait* est défini par ses *primitives* qui sont indépendantes de la réalisation.