Machine, Algorithme et Programme : une introduction à la logique et à la complexité des algorithmes.

Philippe Langevin
langevin@univ-tln.fr

Author address:

LABORATOIRE GRIM, UNIVERSITÉ DE TOULON ET DU VAR

Algorithmique

1. Algorithme

La notion d'algorithme n'est pas une notion propre à la programmation moderne. Les premiers prototypes d'ordinateurs naissent après (ou pendant ?) la seconde guerre mondiale, de la nécessité de décrypter les messages secrets. Le plus célèbre et probablement le plus anciens des algorithmes est exposé dans le septième élément d'EUCLIDE, un texte du troisième siècle avant J.-C. apparemment bien loin des préoccupations cryptologique du XX-ième siècle. Entre ces deux dates, des machines à tricoter et à calculer plus ou moins célèbre : l'horloge à calculs de Wihlem SCHIKARD (1623), la Pascaline de PASCAL (1642), la machine de Gottfried Wilhem LEIBNITZ (1673), l'arithmomètre de Charles Xavier THOMAS (1822), la machine du mathématicien britannique Charles BABBAGE (1840), le « millionnaire » de Léon BOLLÉE (1888), la machine électrique du statisticien américain Hermann HOLLERITH (1889) à l'origine de la compagnie IBM, sans oublier le célèbre métier à tisser de Joseph Marie JACQUARD (1793) !

EXERCICE 1. *Considérez les égalités :*

$$1 = 1, \quad 4 = 3 + 1, \quad 9 = 5 + 3 + 1, \quad 16 = 7 + 5 + 3 + 1.$$

Déduisez un algorithme de calcul de racine carrée d'un nombre entier. Bravo ! Vous avez (re)découvert le principe des différences finies utilisé par THOMAS *dans son arithmomètre.*

L'informatique est une discipline récente, et encore en pleine mutation. L'étudiant sera peut-être déçu d'apprendre que l'auteur de ces notes n'a jamais (ou presque) suivi ni de cours d'informatique ni cours d'algorithmique ! Mais de fait, comme tous les enseignants-chercheurs de sa génération, ma connaissance dans le domaine s'est faite sur le tas…

Le mot même « algorithme » n'est pas sans mystère. Le lecteur a probablement été glaner quelques information sur un dictionnaire, encyclopédie ou la toile avant de s'inscrire à ce cours. L'édition de 1980 du Petit-Larousse illustré définissait les mots « algorithme » et « algorithmique » comme suit :

Algorithme n.m. (d'al-Kharezmi, médecin ar.). Math. Processus de calcul permettant d'arriver à un résultat final déterminé.

Algorithmique adj. Qui concerne l'algorithme.

Alors qu'une version d'une douzaine d'années moins ancienne nous apprend que l'algorithmique n'est pas qu'un adjectif mais bien une discipline[1] :

[1] Certains de mes confrères disent « algorithmie », pourquoi pas ?

Algorithme n.m. (d'al-Kharezmi, surnom d'un mathématicien arabe). Math. et Inform. Suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.

Algorithmique adj. De la nature de l'algorithme.◇ n.f. Sciences des algorithmes, utilisés notamment en informatique.

Une nouvelle définition vient et chasse l'autre. Le personnage al Kharesmi passe du statut de médecin à celui de mathématicien ce qui n'est que justice puisque ce dernier n'est autre que l'inventeur de l'algèbre ! Pour quelques savoir de plus, le lecteur peut consulter le premier tome [**?**] de l'oeuvre de D. KNUTH, ou encore l'encyclopédie des sciences de D'ALEMBERT et DIDEROT, une vielle référence signalée dans *histoire d'algorithmes* [**?**] ou encore, mais dans une bien moindre mesure, ma petite note sur le cryptosystème RSA [**?**].

EXERCICE 2. *Depuis Léonard de Pise, « algorithme » et « nombre » vont souvent de paire. Plus encore, l'étymologie du mot algorithme semble associer ces deux mots. Voyez vous comment ?*

La douleur des nombres suggérée par la solution de l'exercice précédent et/ou une note catastrophique dans un module de physique ont vite fait de fâcher les étudiants avec les logarithmes. Et du coup, au rythme du galop [2] , ils optent pour un anagramme sans trop savoir de quoi il en retourne...Welcome ! Profitons de l'occasion qui, après tout, n'est pas forcément un accident de parcours, pour préciser les choses.

Un problème est la donnée de deux ensembles et d'une application, le certificat. Le premier est composé des instances du problème, le second contient des solutions possibles. Le certificat associe à chaque couple (d, s) formé d'une instance et d'une solution, une valeur de vérité : vrai ou faux ; Lorsque le certificat de (d, s) est vrai, on dit que s est une solution de d. Trouver un algorithme pour résoudre un problème, c'est déterminer une démarche générale qui associe à chaque instance de ce problème une solution. Intuitivement, on conçoit qu'il est plus facile de certifier que de calculer une solution. Cette manière un petit peu pompeuse de voir les choses conduit à la théorie de la décidabilité et aux questions de complétude qui seront abordées en fin de deuxième cycle.

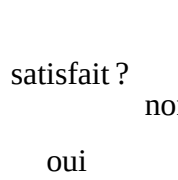
EXEMPLE 1.1. *Vérifier que* 3 *est une solution de l'équation*

$$X^3 - 15X^2 + 71X - 105,$$

est très facile. Mais, quelles sont les autres solutions ? Indication : diviser le polynôme par $X - 3$ *puis appliquer vos connaissances de l'équation du deuxième degré. Qu'est-ce qui vous a demandé le plus d'énergie ?*

[2] Juste un troisième anagramme : -).

Face à un problème, il existe plusieurs attitudes et toujours au moins deux : résoudre ou ne rien faire, je rigole ! Disons plutôt qu'il existe deux algorithmes universels, l'algorithme stochastique et l'algorithme le plus bête. Pour appliquer l'algorithme le plus bête, il faut « inventer » une relation d'ordre sur l'ensemble des solutions possibles. Chaque ordre donne lieu à un algorithme, on teste la plus petite valeur, si c'est la solution alors c'est gagné. Sinon, on teste la valeur qui suit et ainsi de suite jusqu'à obtenir satisfaction...



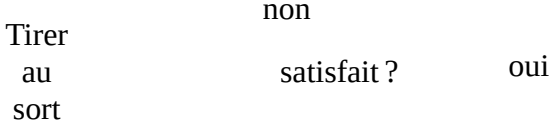
(A) Affecter la valeur minimale à s.
(B) Si s est solution aller en (D).
(C) Changer s en son successeur.
(D) Terminer.

Il faut avoir conscience que cet algorithme peut parfois être le plus performant des algorithmes pour résoudre un problème donné. Moralité, en matière d'algorithmique, il faut avoir l'intelligence de passer pour un idiot, une attitude à adopter avec modération.

EXERCICE 3. *Décrire des situations pour lesquelles l'algorithme le plus bête ne l'est pas obligatoirement.*

Notons bien qu'il est toujours possible de construire un ordre pour appliquer l'algorithme précédent. En effet, on peut décider au hasard du plus petit élément, du second, du troisième etc... L'algorithme stochastique qui, comme son nom ne l'indique pas forcément, est, quant à lui, fondé sur le hasard.

(A) Choisir s aléatoirement.
(B) Si s est une solution de d alors aller en (D).
(C) Aller en (A).
(D) Terminer.



EXERCICE 4. *L'algorithme stochastique est-il aussi simple qu'il en a l'air, où se cache la difficulté ?*

FIG. 1 – Un autre calcul de π...

EXERCICE 5. *Écrire un algorithme stochastique pour calculer la valeur de π. Indication : la probabilité pour qu'un point du plan (x, y) vérifiant $|x| \leq 1$ et $|y| \leq 1$ soit dans le disque unité est proportionnelle à la surface du disque.*

2. Principes

Outre ces deux approches que l'on peut qualifier d'universelles, comment doit on procéder pour inventer un algorithme ? Dans pas mal de situation, il existe une démarche « naïve » qui consiste à appliquer les définitions avec lesquelles le problème est formulé. Par exemple, le déterminant d'une matrice carrée (a_{ij}) de dimension n est donnée par la formule :

$$\det(a) = \sum_{\sigma} \epsilon(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)}$$

où la somme porte sur toutes les permutation de $\{1, 2, \dots, \}$ et $\epsilon(\sigma)$ désigne la signature de la permutation σ. La démarche naïve consiste à énumérer toutes les permutations σ pour appliquer la formule du déterminant. Elle conduit à un algorithme difficile à mettre en oeuvre, qui plus est, particulièrement inefficace.

EXERCICE 6. *L'algorithme naïf que je viens tout juste de critiquer ne manque pas d'intérêt. Convenons de représenter une permutation de l'ensemble $\{1, 2, \dots, n\}$ par un tableau de taille n. Écrire un algorithme* permut(n) *pour énumérer toutes les permutations.*

L'algorithme permut(n) passe par $n!$ étapes. Pour quelles valeurs de n, votre programme aura-t-il de bonnes chances de terminer en moins d'un jour sur une machine actuelle ?

EXERCICE 7. *Révisez les notions d'orbites, de cycles et de signatures pour écrire* sign(p)*, un algorithme de calcul de signature de permutation.*

Quelque soit la finalité et la démarche choisie, le discours de la méthode demeure une bonne source d'inspiration qu'il faut adapter au langage informatique. Une première analyse doit faire ressortir les éléments constitutifs

du problème à résoudre. Quelles sont les données ? Quelle est la question ? Qu'est-ce qu'une solution ?

L'élaboration de l'algorithme passe par le choix d'une approche algorithmique (glouton, dynamique, récursive. . .) qui décrit un enchaînement de sous problèmes locaux dont les résolutions successives ou intermédiaires conduisent de proche en proche à la solution globale.

Il convient ensuite de faire une prévision précise et détaillée des variables qui seront nécessaire à la mise en oeuvre de la stratégie algorithmique. Des choix judicieux pour les types de variables, les identificateurs de types, de variables et sous programmes vous souffleront peu à peu les détails de l'implantation algorithmique.

3. Variables

Disons le de suite, notre vision des algorithmes est proche de celle des programmes ce n'est pas nécessaire mais cela facilite les phases d'implantations [3] ou mises en oeuvre i.e. le passage de l'algorithme au programme. La contrepartie d'un langage rigoureux, est le manque de souplesse qui rend parfois pénible l'expression des idées simples et claires. Un algorithme est composé de deux parties bien distinctes. Une partie définition ou déclaration de données et une partie code ou instruction. La partie déclaration décrit la forme des variables utilisées dans la partie instructions. La partie instructions de l'algorithme décrit un mécanisme de construction d'états intermédiaires qui, au cours de l'exécution, correspondent aux différentes valeurs prises par les variables de l'algorithme. L'initialisation de l'algorithme, c'est-à-dire l'affectation primordiales des variables avant de lancer le code, détermine tout le fil d' exécution.

En général, on distingue des variables *statiques* et des variables *dynamiques*. Les variables statiques complètement déterminées avant l'exécution de l'algorithme ; elles sont logées dans la zone des *données principales* du programme. Les variables dynamiques sont crées au cours de l'exécution comme les paramètres des sous programmes logés dans la *zone de pile* et, les variables allouées dans la *zone de tas* par le précieux ramasseur de miettes.

EXERCICE 8. *La notion d'algorithme s'applique aux situations les plus diverses. Les garçons pourront essayer de retrouver les notions d'instructions et de variables d'une recette de cuisine. Les filles feront de même sur le thème comment changer une roue crevée.*

EXERCICE 9. *Que signifie créer une variable ?*

[3] Note that "implémentation" is not french.

```
int   x, *z;  x = 64218; z = &x;
printf("\nadresse de x %p ou %p, pas %p !", &x, z, *z);
printf("\nvaleur de x %d ou %x, mais pas %x !",x,*z,z);
```

FIG. 2 – Adresse et valeur.

Dans tous les cas, une variable est complètement caractérisée par : son adresse et son type, desquels on déduit ou calcule la valeur.

$$\text{valeur} \Longleftarrow \begin{cases} \text{adresse} \\ + \\ \text{type} \end{cases}$$

Pour une variable statique, l'adresse est constante au cours de l'exécution du programme. Le type de la variable dépend du lieu de consultation, et peut être modifié par une opération de transtypage. Le transtypage n'a pas le même effet sur une variable typée que sur une variable non typée. Le transtypage d'une variable de type flottant x en un entier z par

```
z = ( int ) x
```

relève davantage de la conversiuon que du transtypage à proprement parler.

EXERCICE 10. *Quelle est la norme de représentation des variables de type* `double` *? Utiliser un transtypage pour trouver exposant et mantisse d'une variable* `double`.

La fonction `(void *) malloc( size_t size )` renvoie une adresse sans type. Elle peut être utilisée pour allouer de l'espace à un pointeur vers un entier par un transtypage

```
p = ( int ) malloc (sizeof(p)*int)
```

EXERCICE 11. *L'écho du code (*FIG.

```
adresse de x 0xbffffb44 ou 0xbffffb44, pas 0xfada !
valeur de x 64218 ou fada, mais pas bffffb44 !
```

Pour les architectures de la famille des microprocesseurs INTEL 8086–80486, unités centrales des micro-ordinateurs P.C., une adresse est exprimée sur deux mots de 16 bits, appelés segment et offset. À tout couple de mots formé d'un segment s et d'un offset d, on associe une adresse sur 20 bits : $s \times 16 + d$. Cette particularité fût assez mal gérée pendant quelques années au niveau des systèmes, et des compilateurs : obstruction à la manipulation des tableaux de plus de 64Ko. La gestion des pointeurs dans un programme en TURBO PASCAL *était presque décourageante comparée au langage* C*...*

```
TYPE PTR=RECORD
          SEG,OFS:WORD
        END;
VAR  PB:^BYTE;
     PW:^INTEGER;
BEGIN
    ...
    INC( PTR(PB).OFS );
    ...
    INC( PTR(PW).OFS, 2);
    ...
END.
```

```
char *pb;
int  *pw;
int main (void)
  {
    ...
    pb++;
    ...
    pw++;
    ...
  }
```

La segmentation de la mémoire facilite la conception des systèmes multitaches, il est toujours possible d'associer à une adresse un entier mais il convient de faire une différence entre les deux. En particulier, une adresse représente toujours un nombre, le numéro d'une cellule mémoire, alors qu'un entier arbitraire a de bonne chance de « pointer » une zone interdite.

4. Mode d'adressage

Il existe plusieurs de types de variables : simple, structurée (enregistrement ou tableau) et pointeur, auxquels correspondent des modes d'adressages particuliers. Une variable de type simple est utilisée pour un adressage direct, une variable de type enregistrement pour un adressage par champ, une variable de type tableau pour un adressage indexé et une variable de type pointeur pour un adressage indirect. En `langage C`*, et par ordre de priorité croissant, les opérateurs : « . », « [] » et « * » permettent de spécifier le mode d'adressage.*

TAB. *1 – Les modes d'adressage*

adr.	*04*	*05*	*06*	*07*	*08*	*09*	*0A*	*0B*	*0C*	*0D*	*0E*	*0F*	*10*
mem.	*1*	*3*	*1*	*1*	*2*	*5*	*1*	*2*	*2*	*1*	*3*	*1*	*5*
mod.	*x*	**y*				*x[y]*							

EXERCICE 12. *Ci-dessous l'affichage résultant de l'exécution du programme (*FIG.

```
Tailles: float 4, VECTEUR 800
vec[5].im est en 0xbffff854
vec[0]    est en 0xbffff828
un float : 1.100000e+01
```

```
#include <stdio.h>
typedef  struct
         {  float re;
            float im; } COMPLEXE;

typedef  COMPLEXE VECTEUR[100];

int main(void)
{ VECTEUR vec;
  float   *ptr;
  int i;
printf("\nTailles : floats %d", sizeof(float) );
printf(", VECTEUR %d", sizeof(VECTEUR));
printf("\nvec[5].im est en  %p", &(vec[5].im));
printf("\nvec[0]    est en %p", &(vec[0]));
for( i = 0; i < 100; i++)
    { vec[i].re = 2*i;
      vec[i].im = 2*i+1; }
ptr = &(vec[5]);
ptr++;
printf("\n%e'',*ptr);
```

FIG. *3 – Exemple de code*

5. Dynamique

```
procedure
    Swap(var a, b:integer);
begin
    a:= a+b;
    b:= a-b;
    a:= a-b;
end.
```

On peut comparer l'exécution abstraite d'un algorithme avec une preuve logique dans laquelle les variables sont **doublement** *variables. Elles sont variables en tant qu'objets de la logique mathématique et, leur valeurs sont variables au cours de la démonstration. Cette dynamique est la source (intarissable) de la majorité des erreurs de conception d'un algorithme.*

EXERCICE 13. *L'effet de la procédure* SWAP *n'est pas immédiat.*

À toutes fins utiles, remarquons que pour échanger le contenu des variables a et b, il vaut tout de même mieux utiliser une variable « temporaire », et appliquer le schéma de la relation de Chasles i.e. jouer au domino, en langage C *:*

```
void echange( int *a, int *b)
     {  int tmp;
        tmp = *a;
          *a = *b;
          *b = tmp;
     }
```

EXERCICE 14. *Écrire un algorithme* inverse(p) *pour calculer l'inverse d'une permutation p représentée par un tableau. Pour le fun, trouver une solution qui fait le calcul sur place, sans mémoire ajoutée.*

6. Passeport

Avant de rentrer dans le vif du sujet, je vous propose un petit test. Il s'agit d'assimiler la philosophie d'algorithmique, au travers d'un exemple simplissime.

Déterminer la plus grande valeur d'un tableau.

Je vous laisse réfléchir, mais poser vous les bonnes questions. Une table de quelle dimension ? Quelle est la nature des éléments du tableau ? Quelle est la relation d'ordre ? Comment procéder ? Quelles variables dois je définir ? Comment les nommer ? Il s'agit d'écrire un algorithme PlusGrandeValeur(t) *qui prend en entrée un tableau de t composé de n éléments indexés par les entiers de 1 à n pour calculer et retourner la plus grande de ses valeurs.*

```
PlusGrandeValeur(t)
retourne un element;
valeur   t: tableau
                d'elements;
variable g: element;
         i, n: indice;
debut
n := taille(t);
g := t[1];
i := 2;
tant que ( i <= n )
  si ( t[i] > g )
     alors g := t[i];
  fsi
  i := i + 1;
fintq
retourner(g);
fin
```

Il suffit de parcourir toutes les cellules du tableau en maintenant à jour le meilleur score qui sera stocké dans une variable g initialisée à l'une des valeurs du tableau, $t[1]$ par exemple. Une variable supplémentaire i est nécessaire pour procéder au parcours, elle pilote une boucle en partant de $i = 2$, sans dépasser n la taille du tableau.

Vous obtenez une solution qui ressemble à la mienne ? Très bien, vous avez votre passeport pour la suite du cours. Attention, inutile d'aller plus loin sans une compréhension parfaite et totale des aspects syntaxique et logique sous-jacents.

Les plus en avance trouveront que l'algorithme « passeport » vole bien bas. Ils n'ont pas tort, pour les occuper, je leur propose de décoller par une analyse du point de vue temps de son calcul, c'est l'objet de l'exercice ci-dessous.

EXERCICE 15. *On veut étudier le temps de calcul de l'algorithme* PlusGrandeValeur(t) *sur l'ensemble des tableaux sans répétition. On suppose que les instructions s'exécutent à temps constant. On note $K(t)$ le nombre de succès du test $(t[i] > g)$.*

(a) Montrer que $0 \leq K(t) \leq n - 1$.
(b) Pour quels tableaux t, de taille n, a-t-on $K(t) = 0$?
(c) Pour quels tableaux t, de taille n, a-t-on $K(t) = n - 1$?
(d) Montrer qu'il existe trois constantes A, B, et C strictement positive tel que $T(t) = An + B + CK(t)$, où n est la taille de t.
(e) Soit $P(k, n)$ la probabilité d'avoir une instance t de taille n tel que $K(t) = k$. Montrer que

$$nP(k, n) = P(k - 1, n - 1) + (n - 1)P(k, n - 1).$$

EXERCICE 16. *Écrire une fonction pour calculer $P(k, n)$.*

7. Tracé de droite

Le problème du tracé de segment de droite et sa résolution par Breshenham illustre parfaitement l'écart qui peut exister entre une solution naïve et une solution efficace.

Dans sa formulation, le problème est élémentaire ; il s'agit de tracer un segment de droite sur une grille discrète (écran d'ordinateur) à partir d'une procédure plot(x,y) *capable d'allumer le pixel de coordonnées entières (x, y).*

Nous nous contenterons d'écrire un algorithme pour tracer le segment issu de

```
Segment(a,b: entier)
variable p : reel ;
         x,y : entier;
debut
p := b/a;
x := 0 ;
y := 0 ;
tant que ( x < a) faire
    y := arrondi(p*x);
    plot(x,y);
    x := x + 1;
fintq
fin
```

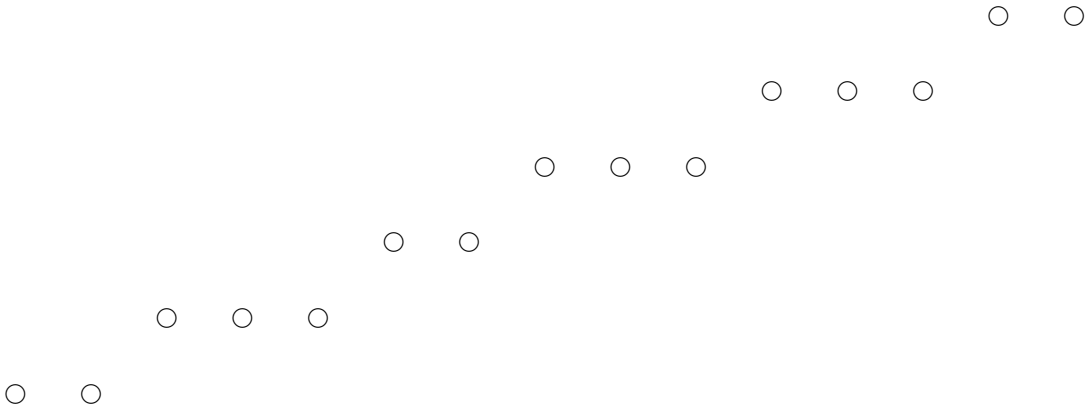


FIG. *4 – Segment de droite*

l'origine vers une extrémité (a, b)*, en faisant l'hypothèse supplémentaire* $0 \leq b \leq a$*. Sans réfléchir davantage, nous sommes en mesure de fournir une solution naïve, c'est l'algorithme* segment(a,b) *proposé ci-contre. Pour comprendre en quoi cet algorithme est peu efficace, il faut avoir en tête les temps de calculs des opérations élémentaires. Sur la plupart des machines, les temps d'exécutions des opérations arithmétiques sont très différents suivant le type manipulé : flottant ou entier. Par exemple, la table (*TAB.

TAB. *2 – Temps des instructions arithmétiques*

time	*seconds*	*ms/call*	*name*
56.54	*7.18*	*7180.00*	*mulrat*
27.64	*3.51*	*3510.00*	*addrat*
5.43	*0.69*	*690.00*	*vidrat*
5.04	*0.64*	*640.00*	*mulint*
2.99	*0.38*	*380.00*	*addint*
2.36	*0.30*	*300.00*	*vidint*

EXERCICE 17. *Les temps de calculs des instructions ont été obtenues avec le profileur* `gprof` *du compilateur public* `gcc`*. Retrouvez ces résultats sur votre machine*

```
#define NBITER 5000001

void vidint(void)
   { long cpt;
     long i, j;
     cpt = NBITER;
     while ( cpt-- )
         i=j;
   }

void addint(void)
   { long cpt;
     long i, j;
     cpt = NBITER;
     while ( cpt-- )
         i = j + k;
   }

void mulint(void)
   { long cpt;
     long i, j;
     cpt = NBITER;
     while ( cpt-- )
         i = j * k;
   }
```

```
void vidrat(void)
   { long cpt;
     double x, y, z;
     cpt = NBITER;
     while ( cpt-- )
         x = y;
    }

void addrat(void)
   { long cpt;
     double x, y, z;
     cpt = NBITER;
     while ( cpt-- )
         x = y + z;
   }

void mulrat(void)
   { long cpt;
     double x, y, z;
     cpt = NBITER;
     while ( cpt-- )
         x = y * z;
    }
```

FIG. *5 – Chronométrage des instructions*

favorite. Quelles sont les autres méthodes pour mesurer des temps machines ?

EXERCICE 18. *De la table (*TAB.

8. Bresenham

Les points à allumer lors d'un tracé de segment sont à coordonnées entières. Dans l'algorithme de Bresenham, on évite le monde rationnel qui coûte cher, pour calculer dans le domaine intègral.

Continuons, toujours en supposant que la pente satisfait à $0 < b/a < 1$. *Il faut procéder en* $n := a + 1$ *étapes. À chaque*

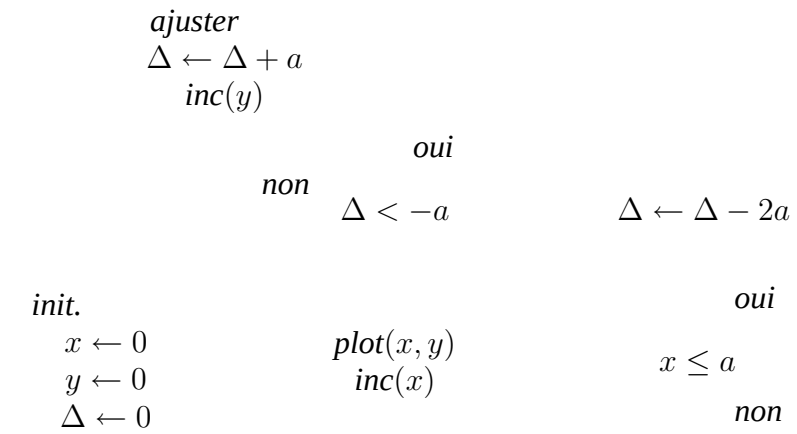


FIG. *6 – Algorithme de Bresenham*

étape, il s'agit de calculer le point de coordonnées (x, y) *à allumer, complètement déterminé par les inéquations :*

$$-\frac{1}{2} < y - px \leq +\frac{1}{2}.$$

Remarquons que le point (x, y) *est au-dessus ou au dessous de la droite suivant que le signe de* $\delta(x) := y - px$ *est positif ou négatif. Les hypothèses faites sur la pente font que si* (x, y) *désigne le point allumé à une certaine étape alors le point suivant sera d'ordonnée* y *ou* $y + 1$*, c'est le premier si* $\frac{-1}{2} < \delta(x) - p$ *avec* $\delta(x+1) = \delta(x) - p$*, et le second sinon avec* $\delta(x + 1) = \delta(x) - p + 1$*.*

Pour optimiser les calculs, on passe dans le monde entier en manipulant la quantité intégrale $\Delta(x) = 2a\delta(x)$*, c'est l'algorithme de Bresenham illustré par la figure (*FIG.

EXERCICE 19. *Traduire l'organigramme de la figure*

EXERCICE 20. *Écrire un algorithme* trace(A,B) *qui trace le segment* $[A, B]$ *sans aucune précondition sur* A *et* B*.*