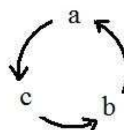
Quelques algorithmes et programmes classiques (en langage C)

1. Les incontournables

1.1. Echange

On se donne par exemple $a = 5$ et $b = 3$. On veut échanger les contenus des cases a et b pour obtenir $a = 3$ et $b = 5$. Comment faire ? Comme si l'on avait deux casseroles, la première a avec de l'eau, la deuxième b avec du lait. Pour faire l'échange des contenus on a besoin d'une troisième casserole c. On transvase l'eau de la première dans la troisième, puis le lait de la deuxième dans la première, puis l'eau de la troisième dans la deuxième, dans un mouvement tournant. D'où le programme :

a=5 ; b=3 ; *afficher le contenu de a et b*
c=a ; a=b ; b=c ;
afficher le contenu de a et b



1.2. Recherche du minimum dans une liste

On se donne N nombres disposés dans un tableau $a[N]$ et l'on prend une variable *mini* qui à la fin contiendra le nombre minimal du tableau. Au départ, on met un grand nombre dans *mini* (par exemple 65000 si l'on travaille sur des entiers et que *mini* est déclarée comme entier). Puis on parcourt le tableau nombre après nombre. A chaque fois, si *mini* contient une valeur supérieure au nombre du tableau, on met ce nombre dans *mini*. Ainsi, à l'étape k (indice de la case du tableau), *mini* contient le plus petit des nombres de $a[0]$ à $a[k]$. A la fin du parcours, *mini* contient le minimum de la liste.

Programme :

mini=32000 ;
for(k=0 ; k<N ; k++) if (a[k]<mini) mini=a[k] ;
afficher mini

Exemple :

a[8]	6	7	5	4	8	3	2	9
mini	6	6	5	4	4	3	2	2

d'où *mini* = 2

Remarque : On pourrait aussi bien prendre comme conditions initiales *mini*=$a[0]$ et parcourir le tableau à partir de la case 1. Mais dans certains problèmes, la recherche du minimum n'est qu'une chose parmi d'autres et l'on a besoin de parcourir le tableau entièrement, à partir de la case 0.

1.3. Suite de nombres

Une suite numérique est une succession de nombres indexés par les entiers naturels, soit u_0, u_1, u_2, u_3, etc. , comme par exemple $u_n = 2^n$ (avec n entier naturel) qui est la suite des puissances successives

de 2. Mais une suite est le plus souvent définie par une relation de récurrence permettant d'avoir un terme à partir de certains de ceux qui précèdent, et de conditions initiales.

Par exemple la suite de Fibonacci obéit à la relation de récurrence $u_{n+2} = u_{n+1} + u_{n-2}$ chaque terme étant la somme des deux qui le précèdent, avec comme conditions initiales $u_0 = 0$ et $u_1 = 1$. A partir de ces deux valeurs connues, on peut calculer u_2, puis u_3, et un terme quelconque u_n de proche en proche. On trouve ainsi la succession : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, … Comment programmer une telle suite ?

Comme un terme dépend des deux qui le précèdent, on a besoin de trois cases mémoires *u*, *v*, *w,* le nouveau terme *w* étant tel que $w = v + u$ somme des deux précédents, étant entendu qu'à chaque nouvelle étape *u* et *v* sont décalés d'un cran.

Si l'on veut avoir u_N, *N* étant un nombre entier donné, on fait :

```
u=0 ; v=1 ; /* il s'agit de u0 et u1 */
for(i=2 ; i<=N ; i++)   /* la première valeur de w est u2  */
  { w=v+u ;
    u=v ; v=w ;     /* pour préparer l'étape suivante, dans u on met le v actuel et
                       dans v on met w  */
  }
afficher w (ou v)
```

1.4. PPMC de deux nombres entiers positifs *a* et *b*

Les multiples positifs de *a* sont *a*, 2*a*, 3*a*, 4*a*, … (on passe de l'un au suivant en ajoutant *a*), et ceux de *b* sont *b*, 2*b*, 3*b*, 4*b*, …. Le plus petit multiple commun de *a* et *b* est le premier nombre commun aux deux successions des multiples de *a* et de *b*. Pour cela on prend les multiples de *a* un par un et à chaque fois on teste si c'est aussi un multiple de *b* (on regarde s'il est divisible par *b*). Quand cela arrive pour la première fois, on a le ppmc. D'où le programme :

```
On se donne a et b ;
multiple= a ;
while (multiple%b !=0) multiple+=a ;
afficher multiple ; /*c'est le ppmc */
```

1.5. Histogramme de fréquences

Un dé compte 6 numéros de 1 à 6. On le lance *N* fois. Au terme de ces *N* lancers, on veut savoir combien de fois est sorti le 1, combien de fois le 2, etc., c'est-à-dire la fréquence de sortie de chacun des numéros.

Pour cela, on définit un tableau *nbdefois*[10], mis à 0 au départ et qui à la fin contiendra les résultats demandés. Puis on tire un numéro au hasard entre 1 et 6 et cela de façon répétée. A chaque fois, on augmente *nbdefois*[*numero*] de 1.

```
mettre nbdefois[7] à 0  /* on utilisera seulement les cases de 1 à 6, et pas la case 0  */
for(i=0 ;i<N ;i++) { numero=random(6)+1 ; nbdefois[numero]++ ;}
```

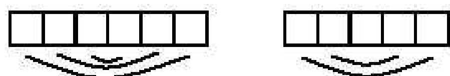
Remarque : la fonction *random*(*n*) ramène un nombre au hasard entre 0 et *n*-1. Il s'agit d'un générateur de nombres pseudo-aléatoires, dans la mesure où ces nombres apparaissent certes dans le désordre, mais sont obtenus par une formule de récurrence (du style de celle de la suite de Fibonacci). Cela a une conséquence : si l'on relance plusieurs fois le même programme, c'est toujours la même liste de nombres aléatoires qui apparaît. Pour éviter cela, on rajoute au début du programme la

fonction *randomize*() qui se charge de changer les conditions initiales du générateur de nombres aléatoires, tout en gardant la même récurrence.[1]

Il ne reste plus qu'à présenter ces résultats sous forme d'un diagramme à barres (histogramme).

1.6. Mettre à l'envers un tableau

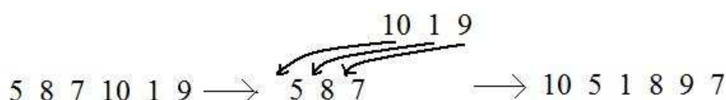
A partir d'un tableau *a*[*N*], on prend une variable gauche *g* qui part de 0 et avance pas à pas, ainsi qu'une variable droite *d* qui part de la dernière case *N*-1 et qui recule aussi pas à pas. A chaque pas pour *g* et pour *d* on procède à l'échange des contenus des cases correspondantes. On continue ainsi tant que $g<d$. Que *N* soit pair ou impair, comme les deux dessins suivants l'indiquent, à la fin toutes les données sont échangées. Le contenu du tableau a été mis à l'envers.



```
remplir a[N]
g=0 ; d=N-1 ;
while (g<d) { aux=a[g] ; a[g]= a[d] ; a[d]=aux ; g++ ; d-- ;}
afficher a[N] ;  /* il est bien à l'envers */
```

1.7. Mélange de cartes en peigne

On dispose d'un paquet de *N* cartes, avec *N* pair. Pour les mélanger, on coupe le paquet en deux parts égales. Puis on insère le deuxième paquet dans le premier en alternant les cartes de l'un et celles de l'autre, de façon que la première carte du nouveau paquet soit la première du deuxième paquet. La deuxième carte du nouveau paquet est la première du premier paquet, etc., comme dans l'exemple suivant avec *N* = 6 :



Le paquet initial des cartes est placé dans un tableau, la première carte étant mise en position 1 (et non 0). Cela est voulu : au cours du mélange, la carte en position 1 va en position 2, celle en position 2 va en position 4, etc. Traitons ce problème sous forme d'exercice.

1) Programmer cette opération de mélange.

On se donne *N* pair et on remplit le tableau *a*[] déclaré *int a*[*N*+1], la case 0 étant sans intérêt. Pour le mélange, on commence par couper en deux, ce qui revient à mettre le deuxième paquet –la deuxième moitié de *a*[], dans un nouveau tableau *b*[*N*/2] à partir de la case 0, le premier paquet restant dans la première moitié de *a*[] .

```
k=0 ;
for(i=N/2+1 ; i<=N ; i++) b[k++]=a[i] ;
```

A noter que nous avons pris une variable courante *k* qui suit le mouvement de *i*, mais à partir de 0.

---

[1] Si vous n'avez pas la fonction *randomize*(), prenez l'autre fonction équivalente *srand*(*time*(*NULL*)). Si vous avez seulement une fonction *rand*() qui ramène un nombre au hasard entre 0 et *RAND_MAX*, fabriquez une fonction *rand01*() qui ramène un flottant entre 0 et 1 (1 non compris), ce qui revient à faire (*float*)*rand*() / ((*float*)*RAND_MAX*+1.), puis fabriquer la fonction *random*(*n*) en faisant *rand01*()**n*. L'autre méthode, consistant à faire *rand*()%*n* pour avoir *random*(*n*) ne peut être utilisée que pour de faibles valeurs de *n*.

Avant d'insérer le deuxième paquet en alternance dans le premier, il convient de faire des trous intermédiaires en décalant les cartes du premier paquet, sachant que les positions de ces cartes doublent. En plus le parcours doit être fait de droite à gauche, et non de gauche à droite, sinon le déplacement d'une carte vers sa droite pourrait écraser une carte qui n'a pas encore été déplacée.

```
for(i=N/2 ; i>=1 ; i--) a[2*i]=a[i] ;
```

Enfin on insère le paquet *b*[] dans les trous (en position 1, 3, …) que nous venons de créer :

```
k=0 ;
for(i=1 ; i<=N ;i+=2)  a[i]= b[k++] ;
```

2) Répéter cette opération de mélange jusqu'à ce que l'on retrouve pour la première fois l'ordre initial des cartes. Ce nombre de répétitions s'appelle la période T du mélange.

Remarquons que pour *N* donné le choix du paquet initial n'a pas d'importance. Au lieu de prendre 5 8 7 10 1 9, on peut aussi bien prendre 1 2 3 4 5 6. Cela ne change pas la période du mélange. Par exemple, on convertit 5 en 1, puis on répète le mélange jusqu'à ce que 1 revienne en première position, suite à quoi on le reconvertit en 5 qui est aussi en première position. Avec les cartes mises ainsi de 1 à *N*, il est facile de savoir si l'on retrouve l'ordre initial : on regarde si l'on a partout *a*[*i*]==*i* . En fait on parcourt le tableau *a*[] et si l'on trouve une valeur de *i* pour laquelle *a*[*i*] !=*i* , on sait que ce n'est pas fini, et on recommence l'opération de mélange. Pour cela on utilise une variable *fini* (appelé booléen ou drapeau (*flag*)) qui prend soit la valeur *non* (ou 0), soit la valeur *oui*, ou 1.

```
for(i=1 ;i<=N ;i++) a[i]=0 ;
T=0 ;
do {  fini=OUI ; T++ ;
        placer ici le programme précédent de mélange
        for(i=1 ;i<=N ;i++) if (a[i] !=i)  { fini=NON ; break ;}
     }
while (fini==NON) ;
afficher T ;
```

1.8. Conversion d'un nombre décimal en binaire

Prenons le nombre 13. Il s'écrit aussi $13 = 1\times 10 + 3$, en séparant le chiffre des dizaines et celui des unités. Plus généralement un nombre écrit en décimal est une somme de puissances de 10 avec des coefficients multiplicatifs qui sont des chiffres compris entre 0 et 9. On veut maintenant l'écrire en binaire, comme somme de puissances de 2, avec des coefficients qui sont 0 ou 1. Ainsi $13 = 1\times 2^3 + 1 \times 2^2 + 1\times 2^0$ devient ▼1101 en binaire (la flèche descendante indique que l'écriture se fait suivant les puissances décroissantes de 2, des poids forts aux poids faibles). Comment passer de 13 à 1101 ? On commence par diviser 13 par 2, ce qui donne comme quotient 6 et comme reste 1. Ce reste est le dernier chiffre de l'écriture en binaire. Puis on recommence avec 6 :

```
13 | 2
 1 | 6 | 2
     0 | 3 | 2
         1 | 1 | 2
             1 | 0
```

On s'arrête lorsque le quotient est 0. La lecture des restes successifs donne 1011, il s'agit de l'écriture en binaire mais suivant les puissances croissantes de 2, ce que nous écrirons ▲1011. Chaque division est de la forme q | 2 / r | q avec le nouveau quotient *q* obtenu à partir de l'ancien quotient *q*,

sauf au départ, quand on part du nombre 13. Pour des besoins d'unification on posera en conditions initiales $q = 13$. D'où le programme :

```
nombre= 13 ; q = nombre ; k=0 ;
do { r [k++] = q%2 ; q=q/2 ; }
while (q != 0) ;
```

afficher le tableau r[] *dont la zone occupée va de 0 à k-1, dans le sens croissant ou décroissant selon les besoins.*

Une variante de ce problème consiste à écrire tous les nombres ayant K chiffres en binaire. Maintenant, pour $K = 5$, 13 va s'écrire ▼01101 ou ▲10110 avec un zéro supplémentaire du côté des grandes puissances de 2. Il suffit pour cela de faire K divisions par 2. Les nombres concernés vont de 0 à $2^K - 1$ en décimal. Il n'y a plus qu'à faire les K divisions successives pour chacun de ces nombres.

```
On se donne K
for (nombre = 0 ; nombre < pow(2,K) ; nombre++)
  { q=nombre ;  for(0=1 ;i<K ;i++)  { r[i]=q%2 ; q=q/2 ;}
    for(i=0 ; i<K ; i++) printf(« %d »,r[i]) ;  /*affichage selon les puissances croissantes ▲ */
    printf(« \n ») ;  /* passage à la ligne */
  }
```

A noter la fonction puissance (*power*) : *pow*(2,*K*) qui donne 2^K. Pour qu'elle soit reconnue, il faudra ajouter au début du programme *#include <math.h>*. Cette fonction *power* travaille normalement sur des nombres flottants, il faudra parfois se méfier si on l'utilise avec des entiers.

1.9. Tri par sélection-échange

C'est le tri le plus simple à programmer. Au départ on dispose d'une liste de N nombres placées dans un tableau $a[N]$. On commence par comparer le premier élément et le deuxième. S'ils ne sont pas dans le bon ordre, on les échange. Puis on compare le premier élément avec le troisième, et en cas de besoin on les échange. On continue ainsi jusqu'à la comparaison du premier élément avec le dernier, avec échange éventuel. Au terme de ce premier parcours, le plus petit élément se trouve dans la première case, dans sa position définitive. Puis on recommence à partir du deuxième élément en le comparant avec ceux qui le suivent…. Voici un exemple :

Pour N=5, on prend $a[N]$= { 3 5 7 2 4} ;

parcours (i=0)	parcours (i=1)	parcours (i=2)	parcours (i=3)
3 5 7 2 4	2 5 7 3 4	2 3 7 5 4	2 3 4 7 5
2 5 7 3 4	2 3 7 5 4	2 3 5 7 4	2 3 4 5 7
		2 3 4 7 5	

Remarquons que l'on a des étapes de $i = 0$ à $i = N - 2$, où c'est $a[i]$ que l'on compare aux suivants, cette case contenant l'élément le plus petit à la fin de l'étape, et qu'à chaque fois, on parcourt la partie à droite de la case i du tableau, avec une variable j allant de $i + 1$ à la dernière case $N - 1$. La comparaison se fait entre $a[i]$ et $a[j]$, et si le premier est supérieur au second, on les échange. Pourquoi cela s'appelle-t-il un tri par sélection-échange ? *Echange*, cela est clair, et *sélection* car à chaque grande étape on a sélectionné le minimum d'une partie de la liste. Le programme s'en déduit :

```
for (i=0 ; i<N-1 ; i++)     for(j=i+1; j<N ; j++)
if (a[i]>a[j]  { aux=a[i] ; a[i]=a[j] ; a[j]= aux ;}
```

2. Exercices

2.1. *Dégager le minimum d'une liste de nombres, et le nombre de fois où il apparaît*

```
mini=32000 ;
for(k=0 ;k<N ;k++)
if (a[i]<mini) {mini=a[i] ; nbdefois=1 ;}
else if (a[i]==mini)  nbdefois++ ;
```

2.2. *Parties à deux éléments d'un ensemble*

Considérons un ensemble de nombres placés dans un tableau *a*[*N*]. On veut obtenir toutes les façons de prendre deux éléments parmi ces *N* objets sans tenir compte de l'ordre dans lequel on les prend, ou encore toutes les parties (les paquets) à deux éléments de cet ensemble.

Il suffit de constater que les crochets de jonction utilisés lors du tri par sélection-échange donnent justement toutes ses parties. D'où le programme :

```
for(i=0 ; i<N-1 ; i++) for(j=i+1; j<N ; j++)  afficher { a[i], a[j]}
```

Comme il y a *N*-1 crochets lors de la première étape, puis *N*-2, etc., le nombre de parties à deux éléments est $(N\text{-}1)+(N\text{-}2)+ \ldots +2+1= N(N\text{-}1)/2$.

Mais si l'on voulait tous les couples, c'est-à-dire les groupes de deux éléments en tenant compte de l'ordre (le couple (1 2) étant différent de (2 1)) , il faudrait faire :

```
for(i=0 ; i<N ; i++)  for(j=0 ; j<N ; j++)  if (i !=j)  afficher (a[i], a[j])
```

2.3. *Mettre les négatifs avant les positifs*

On a un tableau *a*[*N*] rempli de *N* nombres positifs ou négatifs. On veut un programme qui met les nombres négatifs avant les nombres positifs

- En utilisant un nouveau tableau

Au départ on prend un nouveau tableau *b*[*N*] vide (toutes les cases sont à 0). Puis on parcourt le tableau *a*[]. Si on tombe sur un nombre négatif, on le met à gauche dans *b*[] (d'abord dans la case 0 puis dans la case 1, etc.) et si l'on a un nombre positif, on le met à droite, d'abord dans la dernière case, puis dans l'avant-dernière. Ce qui nous conduit à utiliser deux variables gauche *g* et droite *d* :

```
g=0 ; d=N-1 ;
for(i=0 ; i<N ; i++) if (a[i]<0) b[g++]=a[i] ; else b[d--]=a[i] ;
```

Signalons que *b*[*g*++] est un raccourci pour *b*[*g*]=*a*[*i*] suivi de *g*++.

- En restant dans le même tableau

On prend deux variables *g* et *d* qui vont parcourir le tableau *a*[], l'une de gauche à droite et l'autre de droite à gauche, tant que *gauche* est à gauche ($g<d$). Si on tombe sur un nombre négatif dans *a*[*g*] on n'y touche pas, on avance *g* d'un cran et on continue. Sinon, si on a un nombre positif dans *a*[*d*], on le laisse et on recule *d* de 1, et cela se fait autant que possible. Sinon, on est tombé sur un nombre *a*[*g*] positif ou nul et sur un nombre *a*[*d*] négatif, alors on les échange, on avance *g* de 1 et on recule *d* de 1.

```
g=0;d=N-1;
while (g<d)
```

```
if (a[g]<0) g++;
else if (a[d]>=0) d--;
else if (a[g]>=0 && a[d]<0) { aux= a[g]; a[g]=a[d]; a[d]=aux; g++; d--;}
```

2.4. *Suite de Morse-Thue*

On fabrique un mot à base de deux lettres 0 et 1 (on pourrait aussi bien prendre *a* et *b*). Au départ le mot est réduit à 0. Puis il grossit étape par étape, car on applique à chaque fois les règles de substitution suivantes lors de la lecture du mot lettre par lettre :
0 est remplacé par 01, et 1 est remplacé par 10. On obtient ainsi :

Etape 0 : 0
Etape 1 : 01
Etape 2 : 0110
Etape 3 : 01101001
…

Il s'agit de programmer la fabrication de ce mot jusqu'à une étape *N* que l'on se donne.

Pour cela, on déclare un tableau *a*[] de grande longueur, par exemple *a*[20000] où le mot en construction va être placé à chaque étape. Mais celui-ci n'occupe qu'une partie du tableau. Il convient alors de gérer la longueur *L* occupée par ce mot dans le tableau. Comme une lettre est remplacée par deux lettres d'une étape à la suivante, la longueur *L* du mot double à chaque fois, et comme elle est égale à $1=2^0$ au départ à l'étape 0, elle vaut 2^k à l'étape *k*. D'où le programme :

```
a[0]=0 ; L=1 ; /* conditions initiales, étape 0 */
for(etape=1 ; etape<=N ; etape++)
  { for(i=L-1 ; i>=0 ; i- -)  a[2*i]=a[i]) ;
     L= 2*L ;
    for(i=0 ; i<L ; i+=2) a[i+1]= (a[i]+1)%2 ;
  }
afficher le tableau a[] sur une longueur L
```

3. Encore quelques programmes

3.1. Compression et décompression d'un fichier image

Ici les couleurs sont représentées par des nombres, et une image en 2 dimensions, lue ligne après ligne, est stockée dans un tableau linéaire de longueur *N*, avec un numéro de couleur pour chacun des *N* pixels d'écran. Un tel fichier a pour particularité d'avoir des successions de plages de même couleur. La compression consiste à remplacer une plage d'une même couleur par le nombre correspondant à cette couleur, suivi de la longueur de la plage (le nombre de fois où la même couleur est répétée). Par exemple, le fichier image 12222111112 sera remplacé par 11241521, ce qui signifie que la couleur 1 est répétée une fois, puis la couleur 2 répétée quatre fois, puis la couleur 1 répétée cinq fois, et enfin la couleur 2 une fois. La compression est d'autant meilleure que les plages sont longues.

Programme de la compression

On se donne un tableau *a*[] contenant *N* numéros correspondant à des couleurs, mais on va le déclarer sur une longueur *N*+1 en ajoutant en plus à la fin une case butoir, par exemple *a*[*N*]=-1 qui n'est pas un numéro de couleur. La lecture du tableau se fait par plages, la frontière entre deux plages étant obtenue quand *a*[*i*] est différent de *a*[*i*+1]. Par exemple pour le tableau suivant, on a ce découpage par plages, avec des crochets de fin de plage :

1 2 2 2 2 1 1 1 1 1 2 -1

C’est pour avoir aussi à la fin un crochet que l’on a rajouté la case butoir -1. On va remplir le tableau compressé *b*[] par groupe de deux valeurs : le numéro de couleur et le nombre de fois où il apparaît dans une plage.

```
nbdefois=0; k=0;
for(i=0;i<N;i++)    if (a[i] != a[i+1])
                       { nbdefois++; b[k]=a[i]; b[k+1]=nbdefois;    nbdefoisf=0; k+=2;}
                    else  nbdefois++;
longueurb=k;
for(i=0;i<longueurb;i++) printf("%d ",b[i]);  /* affichage du tableau b[] compressé */
```

Décompression

On part maintenant du tableau *b*[*longueurb*] compressé avec pour objectif de le décomprimer, et de retrouver l’image initiale. La décompression est plus simple que la compression, car maintenant il suffit de lire le tableau *b*[] par groupes de deux éléments successifs, le premier donnant la couleur et le deuxième le nombre de fois.

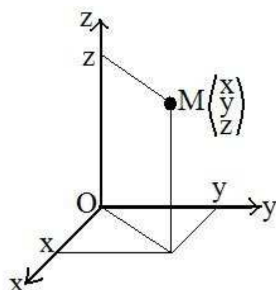
```
k=0 ;
for(j=0; j<longueurb; j+=2)
   {  couleur=b[j]; nbdefois=b[j+1] ;
      for(i=0;i<nbdefois;i++)    a[k++]=couleur;
   }
longueura=k;
for(i=0;i<longueura;i++)printf("%d ",a[i]);
```

3.2. Recherche d’un élément dans une liste déjà triée

On dispose d’une liste triée de N nombres, pour nous un tableau $a[N]$. On veut savoir si un certain nombre d’y trouve. Si oui, on ramène la position de ce nombre dans la liste. Sinon, on ramène « non ». Pour cela, on profite du fait que la liste est déjà triée, en faisant une recherche par dichotomie, c’est-à-dire en coupant en deux, de façon à savoir dans quelle moitié se trouve éventuellement le nombre. A chaque coupe, la zone de recherche est diminuée de moitié, d’où la rapidité du résultat. Pour avoir le milieu d’une zone située entre les indices g et d de la liste, on fait la demi somme : $m = (g+d)/2$, mais cette division entière ne donne pas forcément exactement le milieu. Aussi pour être sûr d’avoir la bonne réponse, on distingue trois cas : soit le nombre cherché est au milieu m, et on s’arrête, soit il est peut-être à gauche entre g et m-1, soit il est peut-être à droite entre m+1 et d. Dans le cas extrême, on arrive à $g == d$ et le nombre n’est pas présent.

```
g=0 ; d=N-1 ; flag=0 ;   /* flag est un « drapeau » prenant la valeur 0 ou 1  */
while (g<=d)
  {  m=(g+d)/2 ;
     if (a[m] == nombre)
        { afficher : « l’élément nombre a été trouvé en position m » ;  flag=1 ; break ;}
     else if (nombre < a[m])  d = m-1 ;
     else  g = m+1 ;
  }
if (flag==0) afficher : « le nombre n’est pas présent dans la liste »
```

3.3. Dessin d'une surface en trois dimensions

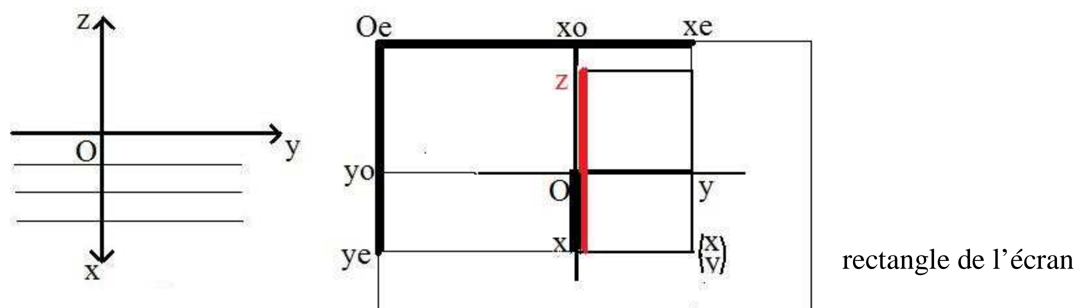


Plaçons-nous dans un repère en trois dimensions, chaque point de l'espace possède trois coordonnées x,y,z, comme sur le dessin, où z est l'altitude du point et x,y les coordonnées du point projeté sur le sol. Remarquons que ce dessin représente l'espace en trois dimensions lorsqu'on le plaque sur un plan, comme l'écran de l'ordinateur.

Dans ces conditions, si l'on se donne $z = f(x, y)$ où z est une fonction (donnée) de x et y, on obtient des points qui forment une surface, et $z = f(x, y)$ est l'équation d'une surface en trois dimensions.

Nous allons prendre comme équation $z = exp(-x^2 -y^2)$, La surface correspondante est une sorte de bosse. En effet, on sait que la courbe (plane) d'équation $y = exp(-x^2)$ est la courbe en cloche de Gauss, et lorsqu'on remplace $-x^2$ par $-x^2 -y^2$, cela revient à faire tourner la courbe autour de l'axe vertical (Oz) d'où la surface.

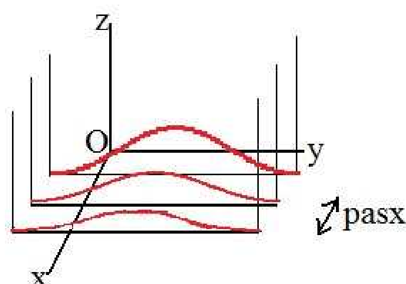
Maintenant simplifions les choses. Au lieu de dessiner sur l'écran de l'ordinateur le repère du dessin précédent, on va le dessiner ainsi, avec les axes des x et des z à l'opposé l'un de l'autre, ce qui correspond à une certaine façon de regarder le repère :



Cela a un premier avantage : les coordonnées xe, ye du point sur l'écran correspondant à M (x,y,z) sont faciles à calculer, en s'aidant du dessin ci-dessus à droite :

$xe = xo + zoomy*y$ et
$ye = yo + zoomx*x - zoomx*z$

où xo et yo sont les coordonnées de l'origine O sur l'écran. On a ajouté un zoom horizontal (*zoomy*) et un zoom vertical (*zoomx*) car on doit passer de x, y, z qui sont de l'ordre de quelques unités et en flottants, à xe et ye qui sont des entiers positifs qui peuvent valoir des centaines (rappelons que l'origine Oe des pixels d'écran est en haut à gauche de l'écran).

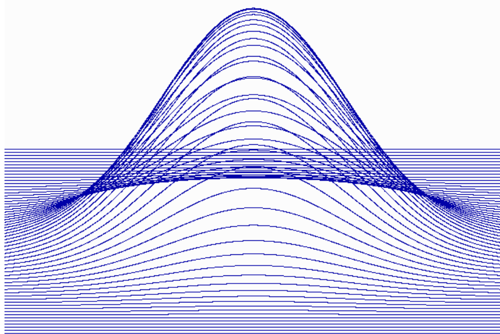


Puis on va découper la surface suivant des plans verticaux parallèles au plan yOz, et régulièrement espacés par une distance *pasx*, ce qui va donner dans chaque plan une courbe. Ces courbes vont être dessinées sur l'écran suivant les valeurs décroissantes de x, pour nous de l'avant vers l'arrière.

Cela se fait par le programme suivant :

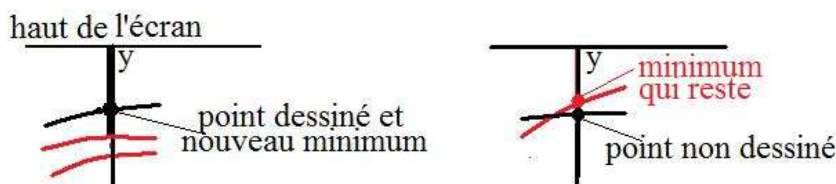
```
for(x=3. ;x>= -3. ; x-=pasx) for(y=-2. ; y<=2. ; y+=pasy)
  { z=exp(-x*x-y*y) ;  xe=xo+zoomy*y ;  ye=yo+zoomx*x – zoomx*z ;
```

```
    afficher le point (xe,ye) ;
  }
```



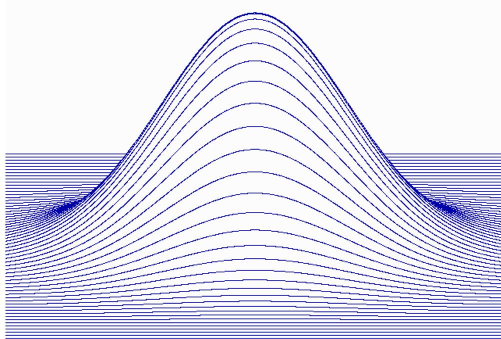
Dans le programme précédent, chaque courbe est dessinée point par point, ce qui impose que *pasy* soit très petit pour qu'on ait une impression de continuité. Mais le résultat obtenu n'est pas bon, car certaines des parties de courbes situées à l'arrière devraient être cachées par celles qui sont devant.

Pour corriger ce défaut, on doit gérer un problème de minimum (à partir du haut de l'écran), sur chaque colonne verticale, c'est-à-dire pour chaque valeur de *ye*. Rappelons que l'on dessine les courbes de l'avant vers l'arrière, pour les valeurs décroissantes de *x*). Si la courbe qu'on est en train de tracer est au-dessous de l'une des précédentes pour certaines valeurs de *y*, on ne dessine pas les points correspondants. Par contre si elle est au-dessus des précédentes, c'est elle qui donne les nouvelles valeurs de la variable *minimum* associée à chaque valeur de *y*.

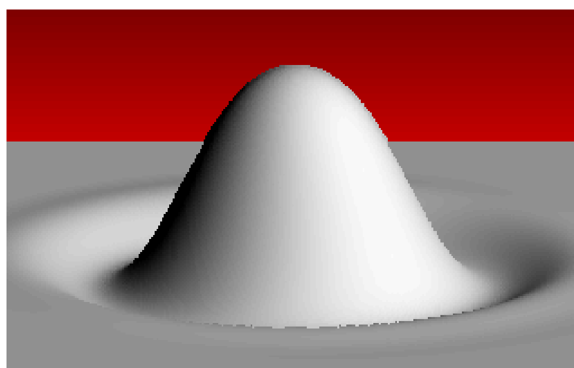


On est ainsi amené à gérer un problème de minimum pour chaque valeur de *y*. On utilise pour cela un tableau *minimum*[*compteur*], où la variable *compteur* démarre à 0 puis augmente de 1 chaque fois que *y* augmente (on ne fait pas *minimum*[*y*] car *y* démarre à une valeur négative et n'est pas non plus un entier).

```
for(compteur=0 ; compteur<10000 ; compteur++) minimum[compteur]=1000 ;
for(x=3. ;x>= -3. ; x-=pasx)
{ compteur=0 ;
  for(y=-2. ; y<=2. ; y+=pasy)
    { z=exp(-x*x-y*y) ;  xe=xo+zoomy*y ;  ye=yo+zoomx*x – zoomx*z ;
      if (yecran<minimum[compteur]
         {afficher le point (xe,ye) ;  minimum[compteur]=yecran ; }
       compteur++ ;
    }
  }
```



Dessin rectifié



Légère modification de la surface et rajout d'ombres (voir cours *graphisme et géométrie*)