Algorithmes de calcul d'une calculatrice portable et leur réplication en Python

Elena LA SCALA

Type de travail : Recherche

Année d'édition : 2013

Table des matières

Introduction

Afin de faciliter les calculs, surtout ceux avec de gros nombres, les hommes ont inventé différents outils. L'histoire des aides au calcul est longue, mais afin de mieux comprendre comment nous sommes arrivés aux instruments actuellement disponibles, j'évoquerai certaines des inventions qui ont permis de progresser jusqu'au développement de la calculatrice disponible de nos jours.

Un à deux millénaires avant notre ère, le premier outil a été l'abaque. Ce terme désigne différents objets aidant au calcul ; il y avait par exemple l'abaque grec, chinois, indien, romain, …
Ces abaques n'étaient pas identiques et pouvaient avoir des usages différents ; l'abaque grec par exemple consistait en des pierres recouvertes de sable sur lesquelles on pouvait dessiner des symboles. Cela permettait d'écrire les calculs que l'on souhaitait effectuer, puis les effacer afin d'en faire de nouveaux.
L'abaque romain était formé d'une table séparée en plusieurs colonnes, chacune représentant une puissance de 10, sur laquelle on plaçait des galets. Il permettait d'effectuer des soustractions, des additions et, avec un peu plus de difficulté, également des multiplications.
Quant au boulier, construit avec du bois de bambou, ainsi qu'avec des pierres ou des perles qui glissent sur des bâtons, il s'agit également d'une sorte d'abaque qui était utilisé par différents peuples. Il servait à réaliser les opérations de base, c'est-à-dire l'addition, la soustraction, la multiplication et la division ; de plus, il était possible de trouver la racine n-ième d'un nombre entier.
De nos jours, les abaques sont encore utilisés, entre autres, par certains marchands en Asie ou en Afrique.

Pendant plusieurs millénaires il n'y eut pas énormément de progrès dans le domaine des instruments de calcul et ce n'est qu'en 1623, que l'universitaire allemand Wilhelm Schickard inventa l'horloge à calculer, la véritable première calculatrice. Cette machine pouvait effectuer l'addition, la soustraction, la multiplication et la division. Malheureusement, cette invention a été détruite lors d'un incendie et nous en connaissons l'existence uniquement grâce à une esquisse et à des lettres envoyées par Schickard à son ami astronome et mathématicien, Johannes Kepler.

En raison de la disparition inopinée de l'horloge à calculer, il fallut recommencer. Au milieu du XVII^{e} siècle le mathématicien, physicien et théologien français Blaise Pascal conçut la Pascaline, instrument qui permettait d'additionner et de soustraire au moyen du complément à 9, méthode dont je donnerai les détails plus loin dans le texte. La Pascaline pouvait également multiplier et diviser, grâce à la répétition d'additions (pour la multiplication) et de soustractions (pour la division).
Inspiré par l'instrument de Pascal, le philosophe et scientifique allemand Gottfried Wilhelm Leibniz inventa en 1671 un cylindre cannelé qui porte son nom, machine pouvant additionner, soustraire, multiplier et diviser.

Ces premières machines permirent le développement progressif de machines à calculer toujours plus performantes, jusqu'à arriver aux calculatrices d'aujourd'hui.
La première machine à calculer à être commercialisée fut l'arithmomètre de Thomas, qui doit son nom au français Charles Xavier Thomas de Colmar, et qui fut inventée au début du XIX^{e} siècle. Cette machine fut ensuite améliorée et rendue plus petite pour qu'elle soit plus maniable.

Cependant, les machines ne pouvaient toujours pas effectuer des multiplications de façon rapide et directe. Ce fut en 1888 que Léon Bollée, un inventeur français, créa la première machine capable de faire cela.
Ces premières machines étaient utilisées surtout pour le commerce et dans les banques.

L'utilisation de circuits électriques dans les calculatrices permit de faire beaucoup de progrès. Par exemple, l'ingénieur et mathématicien espagnol Leonardo Torres Quevedo conçut plusieurs machines à calculer, parmi lesquelles il y a l'arithmomètre électromécanique, en 1920, pouvant effectuer les quatre opérations de base de façon similaire à notre calculatrice moderne, c'est-à-dire que l'utilisateur ne devait que rentrer à l'aide d'un clavier de machine à écrire le premier nombre, sélectionner le symbole correspondant à l'opération qu'il souhaitait effectuer et le deuxième nombre pour obtenir le résultat, imprimé par la machine à écrire.

En 1948 l'ingénieur autrichien Curt Herzstark produisit au Lichtenstein un très petit calculateur mécanique, basé sur les machines de Leibniz et de Thomas, le Curta. Le Curta pouvait additionner, soustraire, multiplier et diviser, et, avec quelques artifices, calculer les racines carrées et réaliser d'autres opérations. Le Curta a été le meilleur calculateur portable jusqu'en 1970, moment de l'arrivée des calculateurs électroniques, grâce à l'invention du microprocesseur par Intel. Les calculateurs électroniques sont les premiers modèles de ceux encore utilisés de nos jours, avec des écrans montrant les chiffres grâces à des diodes luminescents LED à 7 éléments.

En 1972 Hewlett-Packard développa la HP35, la première calculatrice de poche scientifique, c'est-à-dire capable d'évaluer des fonctions trigonométriques et exponentielles. Le Texas Instruments TI30 fit son apparition en 1976 avec un prix de 25 dollars américains, rendant ces machines accessibles à tout le monde ; ce modèle est, par ailleurs, toujours produit de nos jours sous forme modernisée.

Une autre étape importante dans l'histoire de la calculatrice, fut le moment où on comprit que le système décimal n'était pas le plus simple pour faire des calculs et on commença donc à faire des calculateurs fonctionnant en système binaire, comme c'est toujours le cas aujourd'hui.
C'est George Robert Stibitz, américain du 20ème siècle considéré comme un des pères du premier ordinateur digital moderne, qui, en 1938, pensa à utiliser le système décimal-codé-binaire pour faire des calculs. Le décimal-codé-binaire prend chaque chiffre d'un nombre en écriture décimale et le transforme individuellement en binaire.

De nos jours, la calculatrice peut être, pour certaines personnes, un outil du quotidien ; cependant, bien peu de personnes savent quels sont les algorithmes responsables pour toutes les opérations que peut effectuer cette machine.
Je vais donc essayer de découvrir et d'expliquer certains de ces algorithmes en commençant par ceux de l'addition, de la soustraction, puis de la multiplication et de la division. Après ceux de l'exponentiation et de la racine n-ième, j'expliquerai l'algorithme de CORDIC qui permet de calculer sinus, cosinus et logarithme. Ensuite, j'implémenterai ces divers algorithmes sur l'ordinateur, au moyen du langage de programmation Python, afin de mieux comprendre comment ils fonctionnent.

Généralités

Tout d'abord, il faut préciser que, comme les ordinateurs, les calculatrices fonctionnent en système binaire. C'est un système qui fonctionne en base 2, contrairement à la base 10, qui est utilisée dans nos calculs arithmétiques habituels. Le système binaire fait usage de deux chiffres : 1 et 0, d'où son nom.

Dans le calculateur, chaque chiffre du code binaire est représenté par un transistor, éteint ou allumé. Les transistors utilisent de l'électricité pour être en état allumé « on », correspondant à 1, versus éteint « off » ou 0. Tout le fonctionnement de la calculatrice (et de l'ordinateur) est donc basé sur des combinaisons uniques de ces transistors.

Chaque position de ces chiffres correspond à une puissance de 2, la puissance la plus petite (0) se situant tout à droite. Par exemple, le nombre 1011_2 correspond à $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 8 + 0 + 2 + 1 = 11$ en base 10.

Par la suite, je vais analyser les algorithmes de calcul d'abord en système décimal, afin de pouvoir, pour certaines d'entre eux, ensuite mieux comprendre et expliquer comment ils fonctionnent en binaire.

Opérations de base

Addition

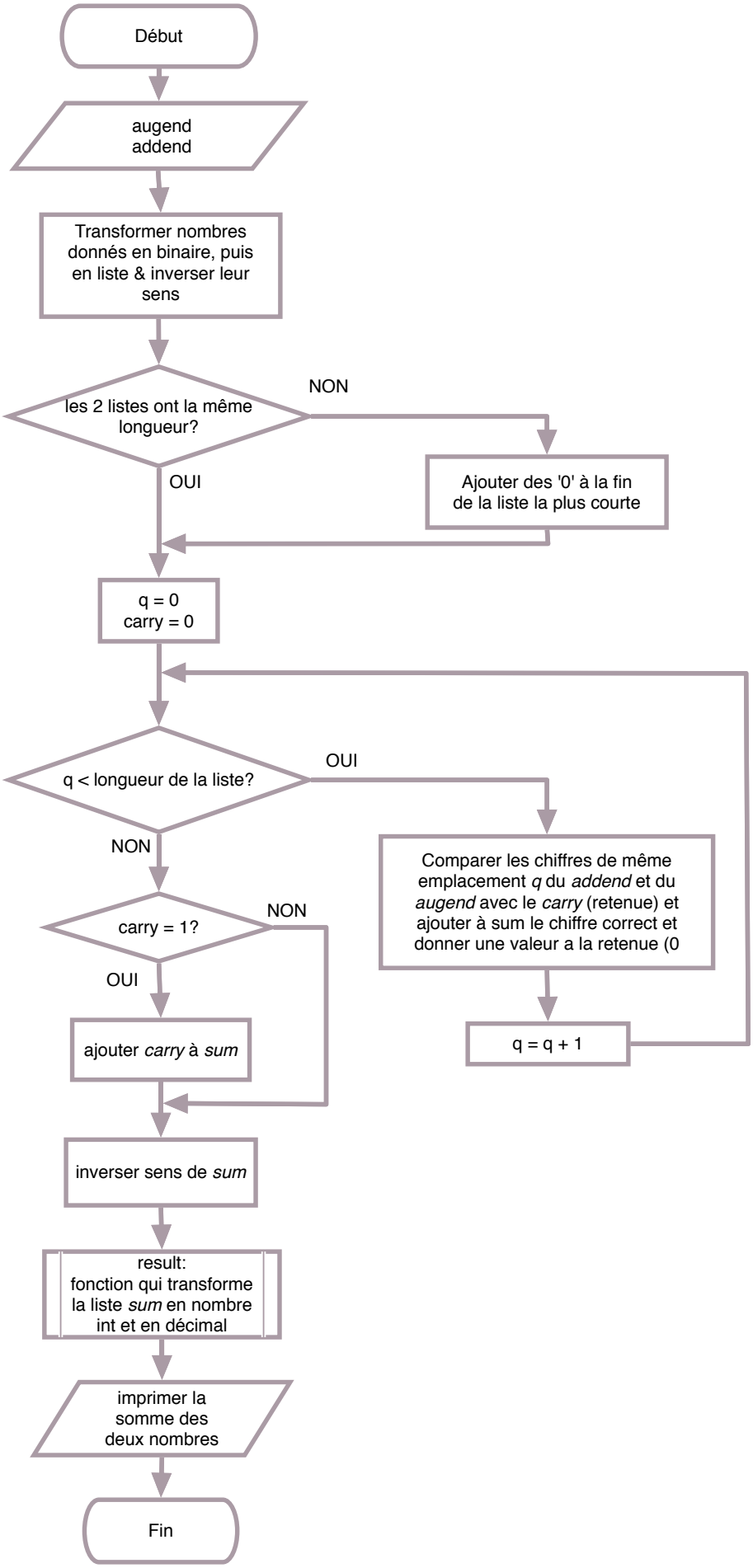
Dans les calculatrices, l'opération de base qui permet de calculer toutes les autres est l'addition.
La méthode d'addition utilisée dans une calculatrice est la même que celle apprise par les enfants à l'école primaire. On additionne donc entre eux les chiffres de même puissance de 10, que j'appellerai de même ordre de grandeur, en commençant par les unités, et si le résultat est plus grand que neuf on ajoute 1 (la retenue) à l'addition d'ordre de grandeur suivante. Si l'un des deux nombres est plus long que l'autre, ses chiffres d'ordre plus grand, c'est-à-dire qui sont situés plus à gauche par rapport au dernier chiffre de gauche du numéro plus court, vont être additionnés à des zéros.

```
    101
     236
+    905
-------
=   1141
```

En binaire la procédure est la même, sauf que l'on n'a pas besoin d'avoir recours à des additions, car on peut se servir de comparaisons. Les deux nombres sont alignés à droite sur deux lignes et on regarde les chiffres qui correspondent à la même puissance de 2 ainsi que l'éventuelle retenue : si tous les chiffres sont des 0, on marquera 0, si l'un des trois est égal à 1, le résultat sera 1, si deux valent 1, ce sera 0, mais on aura aussi une retenue de 1 et, finalement, si tous sont des 1, on marquera 1 et on aura une retenue de 1. En binaire, 18 + 11 serait donc calculé ainsi :

```
     1
   10010 (=18₁₀)
+   1011 (=11₁₀)
-------
= 11101 (=29₁₀)
```

Au moyen du langage de programmation Python, j'ai implémenté un algorithme permettant d'additionner deux nombres comme le ferait une calculatrice.
L'utilisateur doit tout d'abord entrer deux nombres, l'*augend* et l'*addend*, qui sont ensuite transformés en binaire. Les chiffres en binaire sont ensuite transformés en listes et finalement leur sens est inversé, ceci dans le but d'avoir les chiffres de plus petit ordre de grandeur au début des listes. De plus, les listes nous permettent de ne prendre qu'un chiffre du nombre à la fois, ce qui est nécessaire pour cette méthode.
Le programme commence par vérifier si les listes ont la même longueur ou pas : si tel n'est pas le cas, des 0 sont ajoutés à la fin de la liste la plus courte (padding).
Ensuite, on entre dans une boucle, où tous les chiffres de même emplacement sont comparés entre eux et avec l'éventuelle retenue pour savoir quel est le chiffre à insérer dans la liste qui contiendra la somme finale (*sum*).
Une fois sorti de cette boucle, le programme fait un test pour savoir si la dernière retenue a une valeur de 1 ou de 0. Dans le cas où elle vaut 1, il faut l'ajouter à *sum*.
Après cela, on fait appel à la fonction *result*, qui permet, après avoir inversé le sens de *sum* pour retrouver les chiffres de plus grand ordre de grandeur à gauche, de transformer le résultat qui se trouve dans une liste en *integer*.
A la fin de l'algorithme la somme est présentée à l'écran (imprimée).


Début
augend
addend
Transformer nombres donnés en binaire, puis en liste & inverser leur sens
les 2 listes ont la même longueur?
NON
OUI
Ajouter des '0' à la fin de la liste la plus courte
q = 0
carry = 0
q < longueur de la liste?
OUI
NON
Comparer les chiffres de même emplacement *q* du *addend* et du *augend* avec le *carry* (retenue) et ajouter à sum le chiffre correct et donner une valeur a la retenue (0
q = q + 1
carry = 1?
NON
OUI
ajouter *carry* à *sum*
inverser sens de *sum*
result:
fonction qui transforme la liste *sum* en nombre int et en décimal
imprimer la somme des deux nombres
Fin

Soustraction

Pour la soustraction de nombres entiers positifs avec résultat positif, c'est la méthode des compléments de base qui est utilisée. Cette méthode permet de soustraire deux nombres en ne faisant que des additions.
Dans le système décimal, c'est le complément à 10 qui est utilisé. Pour trouver le complément d'un nombre il faut calculer :

$$b^n - y$$

où *b* est la base, c'est-à-dire 10 et *n* est le nombre de chiffres dans *y*.
Par exemple, le complément à 10 de 345 est égal à $10^3 - 345 = 655$.
Nous pouvons remarquer que 345 peut également s'écrire 00345 et que, dans ce cas, son complément serait égal à $10^5 - 00345 = 99655$. Cela nous sera utile plus tard, lors du calcul de la soustraction.

Si on additionne *x* au complément à 10 de *y*, cela donne :

$$b^n - y + x$$

Si *y* est plus petit ou égal à *x*, nous pouvons affirmer que le résultat va être plus grand ou égal à b^n. Si nous enlevons le '1' initial du résultat de ce calcul, ce qui équivaut à soustraire b^n, nous trouvons donc $x - y$.

Si, par exemple, on souhaite soustraire 345 à 400, en additionnant le complément à 10 de 345 à 400 (655 + 400), nous trouvons 1055. En éliminant le '1' se situant au début du résultat, cela nous donne 55, ce qui est le bon résultat. Comme je l'ai expliqué plus haut, ce '1' correspond à b^n, dans ce cas 10^3.

Cependant, en utilisant cette méthode, il faut soustraire *y* à b^n pour le calcul du complément. C'est pourquoi, pour faciliter les calculs, on utilise plutôt le complément à 9, qui est le complément de base diminué et qui équivaut à :

$$b^n - 1 - y$$

(donc 654 pour le nombre 345 dans notre exemple) et qui nous évite une soustraction : en effet, pour trouver le complément à 9 de *y*, il suffit de chercher le complément pour chaque chiffre de *y* en les alignant et les mettant à la suite l'un de l'autre. Cela est possible, car avec un système en base 10, nous ne pouvons pas avoir de chiffres dépassant 9 et par conséquent le complément d'un chiffre est forcément compris entre 0 et 9. Ensuite, pour trouver le résultat correct, il suffit, à la fin du calcul, d'enlever le '1' se trouvant tout à gauche, autrement dit : soustraire b^n, et additionner 1 au résultat.
Si *x* a plus de chiffres que *y*, il faut ajouter des 0 à gauche de *y* jusqu'à ce qu'ils aient le même nombre de chiffres. Cela est possible, car comme nous avons vu plus haut, nous pouvons ajouter autant de 0 que nous le souhaitons à gauche d'un chiffre sans le changer, mais ceci entraine un changement de son complément. Nous pouvons donc dire que pour résoudre une soustraction nous allons utiliser la formule :

$$(b^n - 1 - y) + x - b^n + 1.$$

Si, par exemple, nous souhaitons soustraire 345 à 712 en utilisant cette méthode, il nous faudrait procéder comme suit

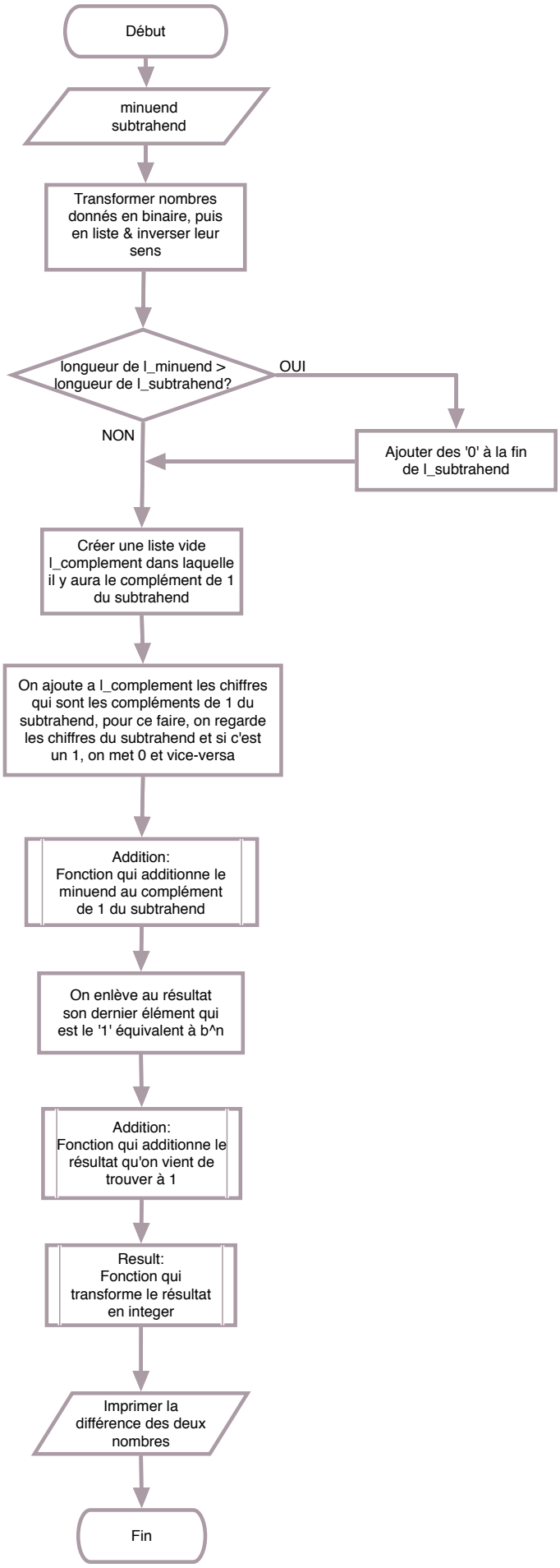
```
    1
     712
+    654 (complément à 9 de 345 ; b^n − 1 − y)
-------
=  ~~1~~366 (~~1~~ : -b^n)
+      1 (+1)
-------
=    367
```

Dans les calculatrices, ce sera le complément à 1 qui va être utilisé. Cela est très pratique et facile, parce que, pour le trouver, il nous suffit d'inverser chaque chiffre, c'est-à-dire remplacer les 1 par des 0 et vice-versa.

Par exemple le complément à 1 de 101_2 (=5) vaut 010_2 (=2). Nous pouvons vérifier cela au moyen de la formule du complément de base diminué d'un nombre *y* que nous avons prouvée plus haut : $b^n - 1 - y = 2^3 - 1 - 5 = 8 - 1 - 5 = 2$.
Pour calculer alors 12 – 5, nous ferions ainsi :

```
    1100 (=12_10)
+    010 (complément à 1 de 5_10)
-------
=  ~~1~~110
+      1
-------
=    111 (=7_10)
```

Dans l'algorithme que j'ai implémenté, l'utilisateur commence par entrer deux nombres, le *minuend* et le *subtrahend* qui sont ensuite transformés en binaire, puis en listes et, enfin, leur sens est inversé, ceci pour les mêmes raisons que pour l'addition.
Tout comme pour l'algorithme de l'addition, on regarde ensuite si les listes sont de même longueur. Si elles ne le sont pas, on ajoute des 0 à la fin de la liste la plus courte, c'est-à-dire *l_subtrahend*.
On crée ensuite une liste vide, *l_complement*, qui va contenir le complément à 1 du *subtrahend*.
Pour trouver le *l_complement*, le programme va examiner chaque chiffre de *subtrahend* et si c'est un 1 il va insérer un 0 dans la liste, et vice-versa.
Après, il suffit d'additionner le complément *l_complement* au *minuend* en faisant appel à la fonction *addition*. Cette fonction est la même que celle expliquée plus haut. Il faut ensuite enlever le '1' équivalent à b^n se trouvant à la fin de cette somme.
A la fin de l'algorithme, on additionne 1 à cette somme, étant donné que nous avons utilisé le complément de base diminué et on fait appel à la fonction *result*, pour pouvoir imprimer la différence des deux nombres en *integer* et pour mettre celle-ci dans le bon sens (c'est-à-dire les plus grands ordres de grandeur à gauche).


Début
minuend
subtrahend
Transformer nombres donnés en binaire, puis en liste & inverser leur sens
longueur de l_minuend > longueur de l_subtrahend?
OUI
NON
Ajouter des '0' à la fin de l_subtrahend
Créer une liste vide l_complement dans laquelle il y aura le complément de 1 du subtrahend
On ajoute a l_complement les chiffres qui sont les compléments de 1 du subtrahend, pour ce faire, on regarde les chiffres du subtrahend et si c'est un 1, on met 0 et vice-versa
Addition: Fonction qui additionne le minuend au complément de 1 du subtrahend
On enlève au résultat son dernier élément qui est le '1' équivalent à b^n
Addition: Fonction qui additionne le résultat qu'on vient de trouver à 1
Result: Fonction qui transforme le résultat en integer
Imprimer la différence des deux nombres
Fin

Multiplication

La méthode utilisée dans la calculatrice pour multiplier deux nombres entiers positifs est très semblable à celle enseignée aux enfants en école primaire, qui s'appelle la multiplication longue. On apprend qu'il faut multiplier le premier nombre par tous les chiffres du deuxième. Ces multiplications produisent des produits intermédiaires qui vont être additionnés l'un à l'autre pour donner le résultat final. Lors de chaque étape, il faut décaler l'emplacement de ces résultats vers la gauche, car on multiplie toujours par des ordres de grandeur de plus en plus grands.

Voici, par exemple, comment serait effectuée l'opération 123*456 :

```
    123
x   456
--------
    738
   6150
+ 49200
-------
= 56088
```

Cette méthode peut être prouvée grâce au concept de la distributivité. En effet :

$$123*456 = 123*(400+50+6) = 123*400 + 123*50 + 123*6.$$

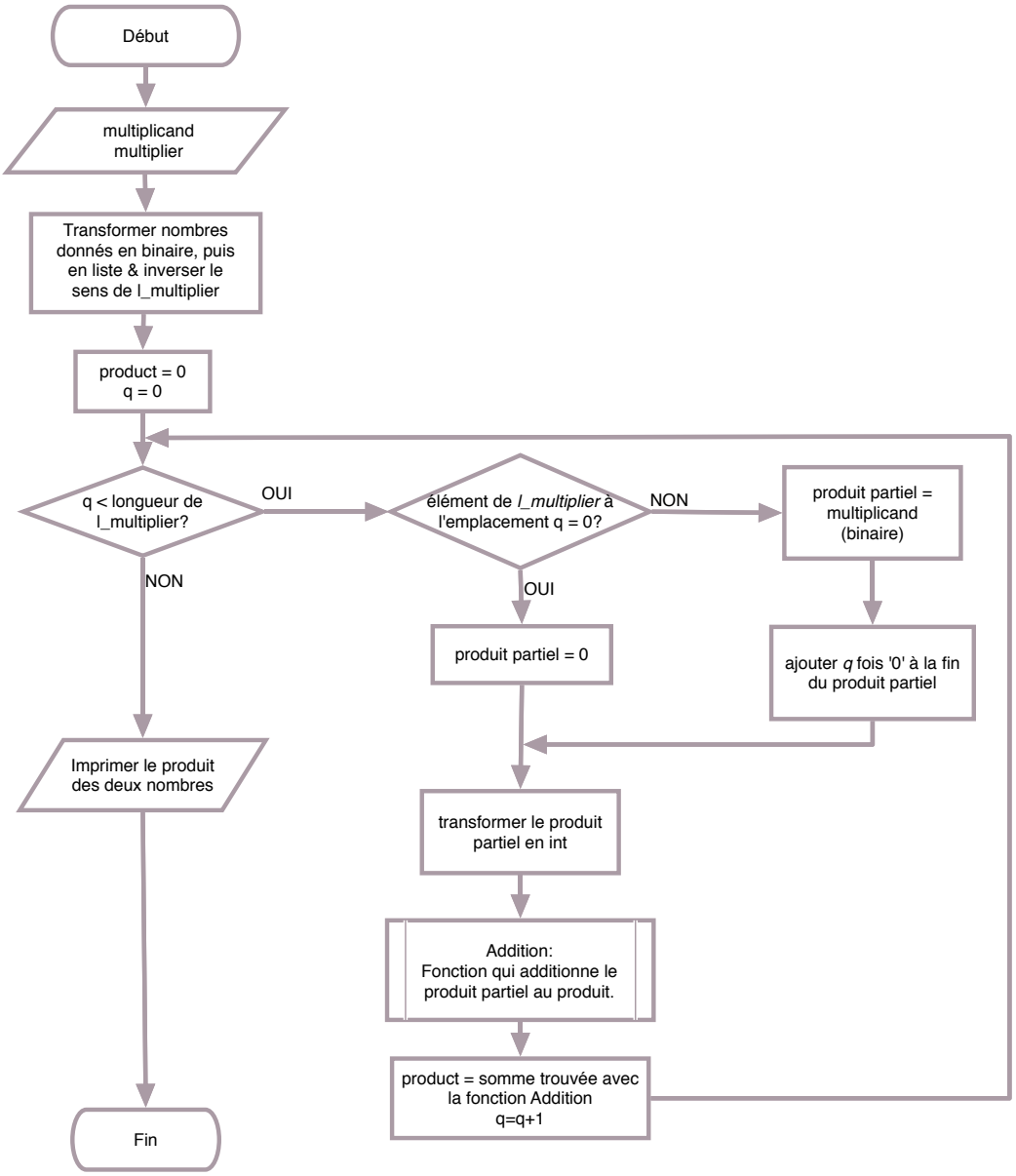
La multiplication binaire fonctionne de façon très similaire ; la seule différence est que l'on n'a pas besoin d'employer de multiplication, car on peut procéder par comparaisons. En effet, si un des deux chiffres, ou les deux, sont égaux à 0, le résultat sera 0, tandis que si les deux valent 1, il sera 1. De façon encore plus simple, nous pouvons dire que si le chiffre du deuxième nombre vaut 0, son produit intermédiaire vaudra également 0, tandis que, s'il est égal à 1, le produit intermédiaire aura la même valeur que le premier nombre et il suffit simplement de le décaler vers la gauche d'un nombre de places équivalent à la puissance correspondant à la position du deuxième chiffre. A la fin de l'algorithme, on additionne tous les produits partiels.

Voici l'exemple, de comment 5 et 7 seraient multipliés en binaire :

```
    101 (=5₁₀)
x   111 (=7₁₀)
-------
    101
   1010
+ 10100
-------
=100011 (=35₁₀)
```

Dans l'algorithme que j'ai implémenté, l'utilisateur doit saisir deux nombres, *multiplicand* et *multiplier*, qui sont ensuite transformés en binaire et mis dans des listes.
On inverse le sens de *l_multiplier*, qui est la liste formée par les chiffres du deuxième nombre, c'est-à-dire celui par lequel on multiplie le premier.
On entre ensuite dans une boucle qui multiplie les deux nombres au moyen de comparaisons et d'additions. Tout d'abord, on regarde si l'élément de *l_multiplier* à l'emplacement q est égal à 0 ou à 1. S'il vaut 0, le produit partiel sera égal à 0, sinon, il sera égal au *multiplicand* et on lui ajoute ensuite des 0 afin de le décaler du nombre de places nécessaires.

A chaque passage de la boucle on additionne le nouveau produit partiel au résultat précédent, ce qui nous permet, à la sortie de la boucle, c'est-à-dire après avoir passé tous les chiffres de *l_multiplicand*, d'imprimer le produit des deux nombres.



Division

La méthode de division de nombres entiers positifs en base 10 est appelée division longue et elle correspond à la méthode enseignée aux enfants : division entière avec reste. Le but de cette division est de procéder par des divisions plus simples et qui n'ont comme quotient qu'un seul chiffre. Pour ce faire, il faut travailler par tranches et chacune doit rester inférieure à x*b, où *b* est la base et *x* le diviseur. On commence les tranches avec les chiffres de plus grand ordre de grandeur du dividende : il faut diviser ce nombre par le diviseur et ensuite noter le résultat qui sera le chiffre de plus grand ordre de grandeur du quotient. Après cela, on doit aussi calculer le reste de cette division, qui se trouve en soustrayant le produit du quotient trouvé et du diviseur à la tranche. Il faut ensuite descendre le chiffre suivant du dividende et l'ajouter à la fin de ce reste, qui sera à nouveau divisé par le diviseur. Il faut continuer cette démarche jusqu'à ce que la tranche que nous trouvons soit plus petite que le diviseur.

Voici par exemple comment serait calculé 73/3 :

```
73|3_
6 |24
13
12
01
```

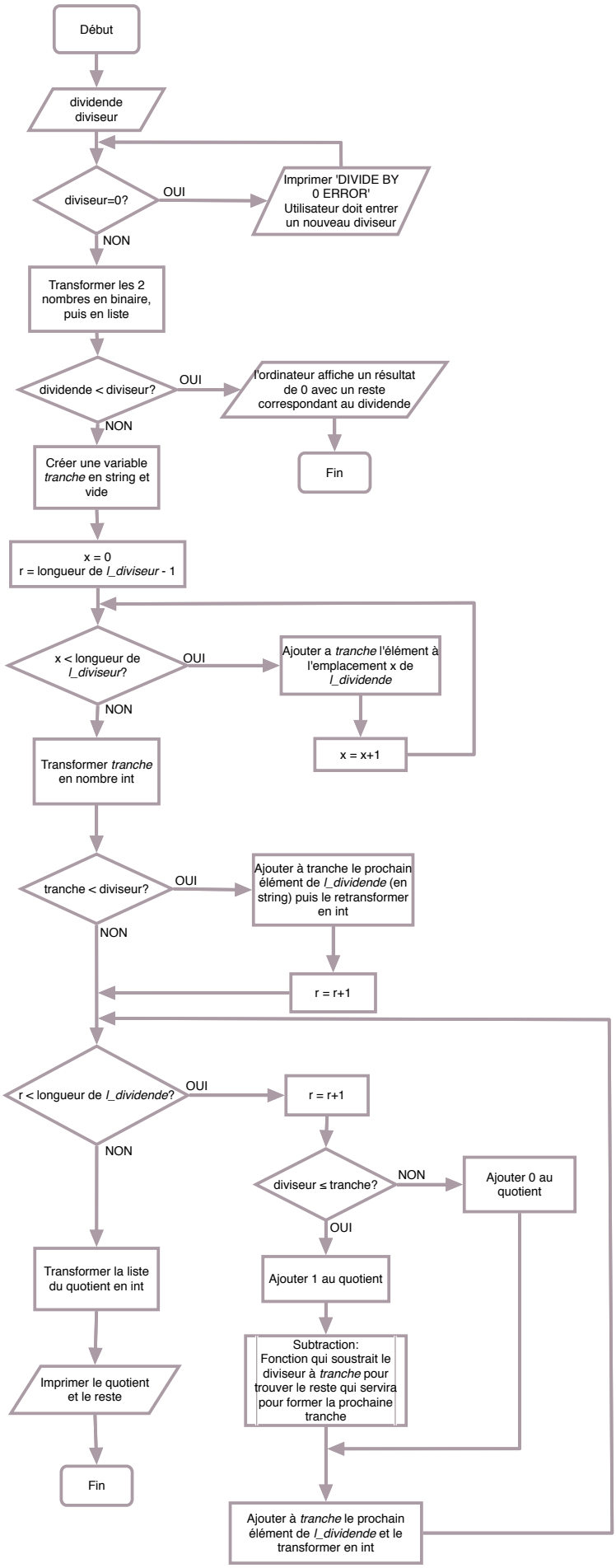
La méthode en binaire se fait plus facilement, car les chiffres du quotient ne peuvent être que 1 ou 0, cela signifie qu'il suffit de regarder si le diviseur est plus petit que le dividende ou pas et marquer ensuite le chiffre correspondant dans le quotient. Par exemple, 22 divisé par 2 serait résolu de cette façon :

```
10110|10__
10   |1011 (=11₁₀)
001
000
  11
  10
   10
   10
   00
```

Dans l'algorithme que j'ai écrit, l'utilisateur doit tout d'abord saisir les deux nombres qu'il souhaite diviser ; le premier est le *dividende* et le deuxième, celui par lequel on divise, s'appelle le *diviseur*.
On commence par vérifier que l'utilisateur n'ait pas rentré un diviseur égal à 0, car il n'est pas possible de diviser un nombre par 0. Si c'est le cas, le programme affichera une erreur et demandera de saisir un nouveau numéro.
Les deux nombres sont ensuite transformé en nombres binaires et sont mis dans des listes.
Si le dividende est plus petit que le diviseur, l'ordinateur affichera directement 0 comme résultat avec un reste équivalent au dividende.
Si ce n'est pas le cas, le programme effectuera la démarche expliquée ci-dessus.
Tout d'abord, afin de prendre la première tranche qui sera divisée par le diviseur, on prend les premiers éléments de la liste du dividende jusqu'à ce que la tranche ait la même longueur que le diviseur. Si la tranche est encore plus petite que le diviseur, on lui ajoute le prochain élément du dividende.

On entre ensuite dans une boucle, dans laquelle on reste jusqu'à ce que tous les chiffres du dividende soient utilisés. Dans cette boucle, l'ordinateur va regarder si le diviseur est plus petit que la tranche du dividende ou pas, si tel est le cas, le chiffre 1 est ajouté au quotient, et dans le cas contraire un 0 est ajouté.
Après cela, si la tranche est plus grande que le diviseur, il faut calculer la différence entre la tranche et le diviseur en faisant appel à la fonction *subtraction* que j'ai expliquée précédemment.
On ajoute, ensuite, au reste le prochain chiffre du dividende et on répète cette étape jusqu'à ce que la tranche que nous trouvons soit plus petite que le diviseur.

On transforme ensuite le quotient ainsi que le reste en *integer* et on imprime ces deux valeurs.


Début
dividende
diviseur
diviseur=0?
OUI
Imprimer 'DIVIDE BY 0 ERROR'
Utilisateur doit entrer un nouveau diviseur
NON
Transformer les 2 nombres en binaire, puis en liste
dividende < diviseur?
OUI
l'ordinateur affiche un résultat de 0 avec un reste correspondant au dividende
Fin
NON
Créer une variable tranche en string et vide
x = 0
r = longueur de l_diviseur - 1
x < longueur de l_diviseur?
OUI
Ajouter a tranche l'élément à l'emplacement x de l_dividende
x = x+1
NON
Transformer tranche en nombre int
tranche < diviseur?
OUI
Ajouter à tranche le prochain élément de l_dividende (en string) puis le retransformer en int
NON
r = r+1
r < longueur de l_dividende?
OUI
r = r+1
NON
diviseur ≤ tranche?
NON
Ajouter 0 au quotient
OUI
Ajouter 1 au quotient
Subtraction:
Fonction qui soustrait le diviseur à tranche pour trouver le reste qui servira pour former la prochaine tranche
Transformer la liste du quotient en int
Imprimer le quotient et le reste
Fin
Ajouter à tranche le prochain élément de l_dividende et le transformer en int

Opérations plus complexes

Afin d'avoir le temps de me pencher sur les algorithmes mathématiques plus complexes et de mieux les comprendre, je ne m'occuperai pas dans le détail des nombres négatifs et décimaux, ce qu'il aurait encore fallu faire pour les opérations précédentes.
Pour la même raison, les algorithmes que je vais expliquer par la suite seront implémentés uniquement en décimal et ne feront pas appel aux fonctions décrites dans les chapitres précédents.

Exponentiation

La méthode utilisée afin d'élever un nombre *x* à la puissance *n* s'appelle exponentiation rapide (*square and multiply* en anglais).
Pour une puissance n-ième (x^n), nous pouvons affirmer que si *n* est égal à 0, le résultat sera de 1 et, s'il est égal à 1, le résultat vaudra *x*. De plus, nous pouvons dire que si *n* est pair, x^n revient à faire :

$$(x^2)^{\frac{n}{2}},$$

puisque les 2 se simplifient et nous obtenons le même calcul que le précédent.
Pour *n* impair, il faut faire

$$x \cdot (x^2)^{\frac{n-1}{2}}$$

car, ainsi, la puissance sera un entier et le *x* qu'on enlève de la puissance est multiplié par ce résultat.
Au moyen d'une fonction récursive, c'est-à-dire qui s'appelle elle-même, nous pouvons donc éviter les puissances supérieures à 2. Cette fonction élèvera, pour une puissance paire, le carré de *x* (donc $x \cdot x$) à la puissance n/2 ou, pour une puissance impaire, multipliera *x* par le carré de *x* à la puissance (n-1)/2 jusqu'à ce que la puissance qu'on trouve soit égal à 1.
Si, par exemple, nous souhaitons effectuer le calcul suivant : 4^5, nous procéderons ainsi :

1. $x = 4$ et $n = 5$
2. 5 est impair, donc on calcule : $4^5 = 4 * (16)^{\frac{5-1}{2}} = 4 * 16^2$ et, pour calculer cette puissance, nous continuons avec la même méthode.
3. 2 est pair, donc on calcule : $16^2 = (16 * 16)^{2/2} = 256^1$
4. Puisque la puissance de 256 est de 1, nous gardons le résultat de 256 au point 3.
5. Il nous suffit ensuite de le multiplier par le 4 qui nous restait au point 2.
6. Nous trouvons donc le résultat, 256*4 = 1024

Dans l'algorithme que j'ai implémenté, l'utilisateur entre deux nombres ; la base, *b*, et l'exposant, *n*, qui sont ensuite transformés en nombres binaires.
Tout d'abord, on vérifie si l'exposant donné par l'utilisateur est plus petit que 0 ou pas. Si oui, on appelle à nouveau la fonction, mais cette fois, avec, comme base, l'inverse de la base entrée et comme exposant *–n*. Nous pouvons faire cela, car si on change le signe de la puissance, cela inverse la base.
Ensuite, il y a un test pour voir si *n* vaut 0, puisque dans ce cas il nous retourne la valeur 1 et un autre test pour voir s'il vaut 1, cas dans lequel la valeur de *b* sera retournée. Après cela, on vérifie si *n* est pair ou impair. Cela nous permet d'appeler la fonction récursivement avec les valeurs correctes et de faire les calculs nécessaires.
A la fin, on imprime le résultat.

Racine n-ième

L'algorithme qu'on utilise pour calculer la racine n-ième (x) d'un nombre A, positif, se base sur une suite définie par récurrence, c'est-à-dire qui utilise les termes précédents afin de calculer de nouveaux termes.
La première étape de cet algorithme est de choisir un x_0, proche de la valeur recherchée.
Il faut ensuite calculer

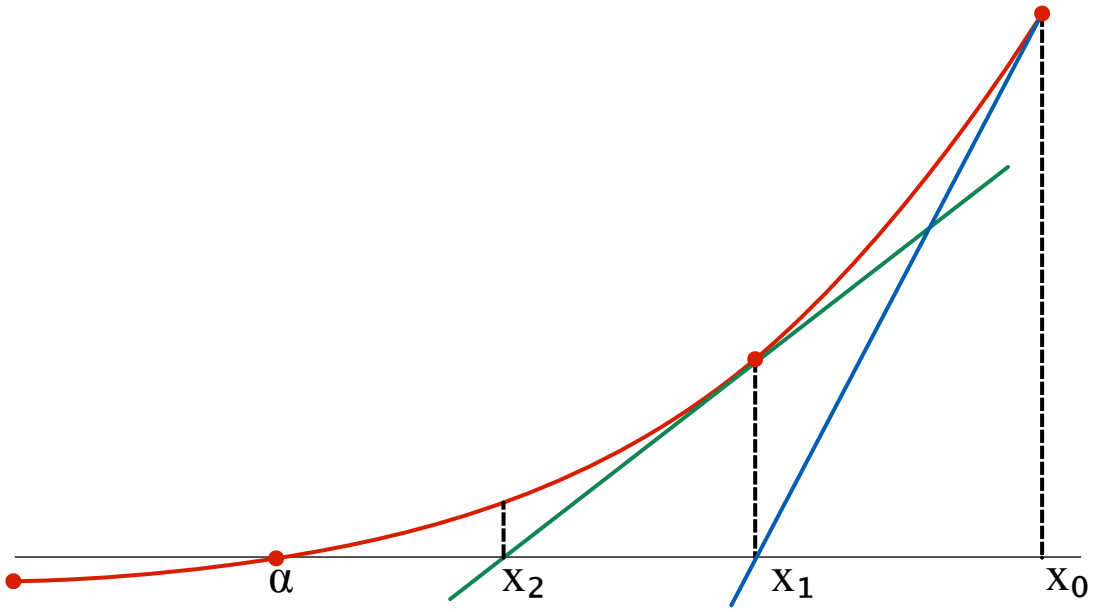
$$x_{k+1} = \frac{1}{n}\left[(n-1)\cdot x_k + \frac{A}{x_k^{n-1}}\right]$$

jusqu'à atteindre la précision souhaitée.

Afin d'expliquer comment cet algorithme peut fonctionner, je vais le mettre en lien avec la méthode de Newton, appelée également méthode de Newton-Raphson, dont il constitue un cas particulier.
Cette méthode a été découverte au XVII[e] siècle par l'anglais Isaac Newton, qui était astronome, mathématicien et physicien, et par un autre mathématicien anglais, Joseph Raphson. Elle permettait de trouver les zéros d'une fonction.
Avec cette méthode, il faut commencer de la même façon que l'algorithme énoncé précédemment et ensuite il faut calculer

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

jusqu'à ce que le résultat soit suffisamment précis.
Voici une illustration du fonctionnement de cette méthode :



(Adapté de http://fr.wikipedia.org/wiki/Fichier:Methode_newton.png)

Cette méthode consiste en une approximation de f par sa tangente en x_k, c'est-à-dire :

$$f(x) \approx f'(x_k)\cdot(x - x_k) + f(x_k)$$

Donc si on cherche les zéros de la fonction il faut résoudre

$$0 = f'(x_k)\cdot(x - x_k) + f(x_k) \rightarrow x = x_k - \frac{f(x_k)}{f'(x_k)}$$

où x correspond à l'approximation suivante : x_{k+1}. Afin que cette méthode puisse fonctionner, il ne faut pas que la dérivée s'annule.
Si nous prenons la fonction $f(x) = x^n - A$, nous pouvons affirmer que, en trouvant

le zéro de cette fonction, nous trouverons également la racine n-ième de *A*, car, pour qu'il y ait un zéro, *A* doit être égal à x^n.
Il suffit ensuite d'utiliser la méthode de Newton.
Tout d'abord, on doit calculer la dérivée de notre fonction, qui nous donne

$$f'(x) = nx^{n-1}.$$

Ensuite, nous remplaçons dans l'algorithme la fonction qui nous intéresse et nous trouvons :

$$x_{k+1} = x_k - \frac{x_k^n - A}{nx_k^{n-1}} = \frac{1}{n}\left[(n-1)\cdot x_k + \frac{A}{x_k^{n-1}}\right]$$

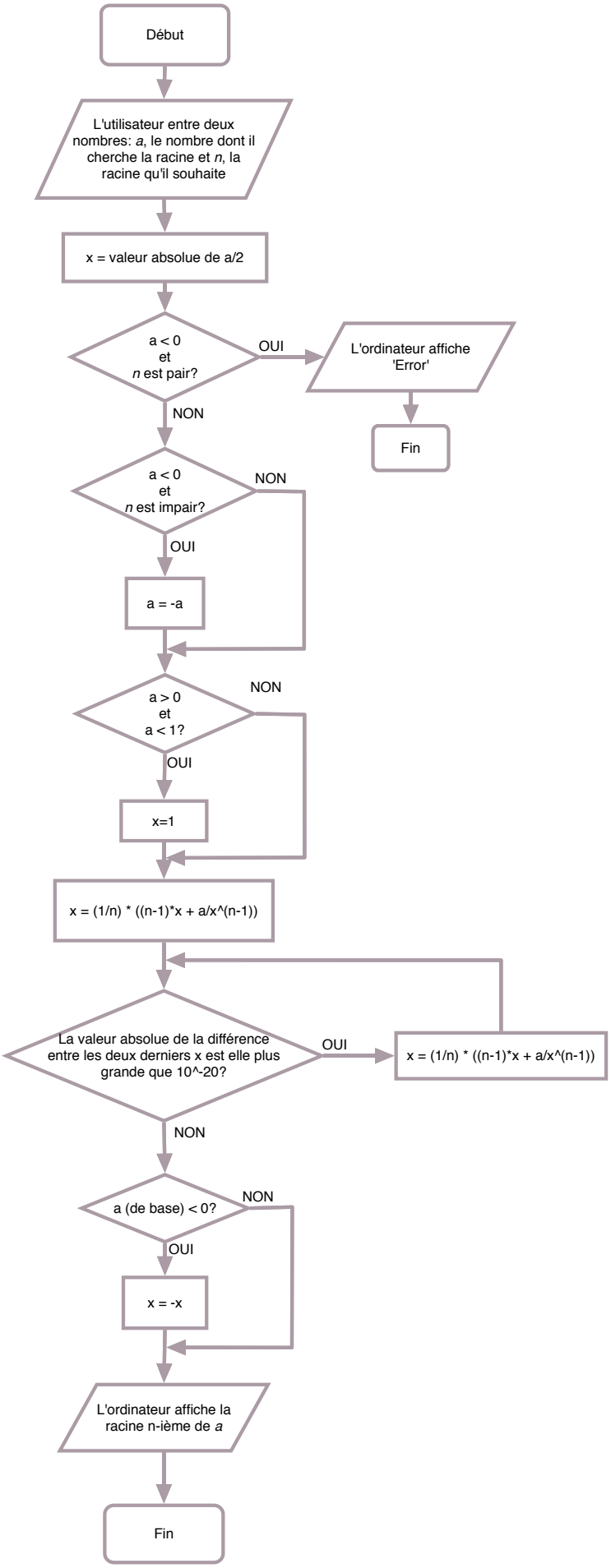
Pour la racine carrée, c'est un cas particulier de cet algorithme qui est utilisé et il s'appelle la méthode d'Héron, qui doit son nom à Héron d'Alexandrie, mathématicien grec du premier siècle après Jésus-Christ ; afin de trouver son algorithme, on doit remplacer *n* par 2 dans l'algorithme de la racine n-ième.

Si nous souhaitons trouver la racine n-ième d'un nombre négatif, il faut différencier deux situations ; si *n* est pair, il n'y a pas de solutions réelles.
Si, au contraire, *n* est impair, nous pouvons trouver la solution en calculant $-\sqrt[n]{-A}$.
En effet, *–A* nous donne un nombre positif et nous pouvons donc appliquer la méthode expliquée plus haut.

Dans mon algorithme, l'utilisateur doit tout d'abord entrer le nombre dont il cherche la racine (*a*), et ensuite le nombre *n* représentant la racine recherchée.
Si *a* est négatif et *n* pair, le programme affiche une erreur et se termine. Si, par contre, *a* est négatif, mais *n* est impair, on oppose la valeur de *a* pour pouvoir effectuer l'algorithme énoncé ci-dessus.
Si le nombre entré par l'utilisateur se trouve entre -1 et 0 ou entre 0 et 1 (pas compris), on attribue la valeur de 1 à la première valeur approximative du résultat.
Pour tout autre nombre, cette première valeur sera égale à la valeur absolue de la moitié de *a*.
Nous allons effectuer le calcul énoncé ci-dessus jusqu'à ce que la valeur absolue de la différence entre le résultat trouvé et celui précédent soit inférieure à 10^{-20}. Cela permet donc d'avoir un résultat suffisamment précis indépendamment des valeurs que l'utilisateur saisit, car, si on choisissait le nombre d'étapes à réaliser avant d'arrêter les calculs, le résultat trouvé pourrait, selon les valeurs choisies par l'utilisateur, avoir un écart important par rapport à la valeur exacte.
Avant d'afficher le résultat de la racine, on regarde si le nombre donné par l'utilisateur est négatif ou positif. S'il est négatif, le programme affiche l'opposé du résultat, tandis que s'il est positif il affiche directement le résultat trouvé.
Si, par exemple, nous souhaitons trouver la racine carrée de 5, l'algorithme va procéder comme suit ; il prendra tout d'abord 2,5 (la valeur absolue de la moitié de 5) comme valeur initiale pour x_0. Ensuite il va calculer :

$$x_1 = \frac{1}{2}\left[x_0 + \frac{5}{x_0^2}\right] = \frac{1}{2}\cdot\left(2{,}5 + \frac{5}{2{,}5}\right) = 2{,}25$$

Puisque $x_1 - x_0$ est plus grand que 10^{-20}, l'ordinateur calculera x_2, x_3 et ainsi de suite jusqu'à avoir atteint la précision voulue.


Début
L'utilisateur entre deux nombres: a, le nombre dont il cherche la racine et n, la racine qu'il souhaite
x = valeur absolue de a/2
a < 0 et n est pair?
OUI
L'ordinateur affiche 'Error'
Fin
NON
a < 0 et n est impair?
NON
OUI
a = -a
a > 0 et a < 1?
NON
OUI
x=1
x = (1/n) * ((n-1)*x + a/x^(n-1))
La valeur absolue de la différence entre les deux derniers x est elle plus grande que 10^-20?
OUI
x = (1/n) * ((n-1)*x + a/x^(n-1))
NON
a (de base) < 0?
NON
OUI
x = -x
L'ordinateur affiche la racine n-ième de a
Fin

CORDIC

CORDIC, qui est un acronyme venant de l'anglais *COordinate Rotation DIgital Computer*, est l'algorithme qui est utilisé dans les calculatrices afin de résoudre des fonctions trigonométriques et hyperboliques. Elle ne se sert que d'additions, de soustractions et de décalages de bit.

L'algorithme CORDIC, tel qu'il est utilisé aujourd'hui, fut inventé par Jack E. Volder, ingénieur américain, en 1959. Il fut développé par Convair afin de remplacer le système de navigation du Convair B-58 Hustler, qui était un bombardier étatsunien, par un ordinateur digital. Cela leur aurait permis d'acquérir plus de précision rapidement. En effet, le B-58 n'était pas capable, par exemple de calculer assez rapidement des fonctions trigonométriques.

De nos jours, grâce aux généralisations de John Stephen Walther qui travaillait à Hewlett-Packard, une multinationale américaine, cet algorithme qui fonctionne par approximations successives, peut également être utilisé pour trouver les racines n-ièmes d'un nombre, multiplier, diviser, calculer des logarithmes,…

Sinus et cosinus

Afin de trouver le sinus et le cosinus d'un angle il faudra effectuer des rotations vectorielles d'un angle de plus en plus petit prédéterminé pour pouvoir s'approcher de l'angle voulu.

Tout d'abord, il faut que l'angle $\alpha \in [-\pi; \pi]$ soit donné en radians et, afin de trouver le sinus et le cosinus de cet angle, il faut trouver les coordonnées *x* (pour le cosinus) et *y* (pour le sinus) sur le cercle trigonométrique qui lui correspondent.

On commence par prendre le vecteur $v_0 = \begin{pmatrix}1\\0\end{pmatrix}$ (angle $\alpha_0 = 0$) qui va subir une première rotation de $\frac{\pi}{4}$ dans le sens trigonométrique positif ($\alpha > 0$) ou négatif ($\alpha < 0$). Nous verrons par la suite, comment le sens de la rotation est déterminé.

A chaque itération *i*, le vecteur subira une rotation d'un angle de plus en plus petit, correspondant à :

$$\gamma_i = \tan^{-1}\left(\frac{1}{2^{i-1}}\right)$$

Nous pouvons remarquer, comme indiqué ci-dessus, que :

$$\gamma_1 = \tan^{-1}(1) = \frac{\pi}{4}$$

La rotation du vecteur est calculée au moyen d'une matrice de rotation :

$$R_i = \begin{pmatrix}\cos\gamma_i & -sin\gamma_i\\ sin\gamma_i & cos\gamma_i\end{pmatrix}$$

Cette matrice est celle qui est appliquée à un vecteur pour obtenir une rotation dans le sens trigonométrique d'angle γ dans un plan cartésien à deux dimensions.

Nous pouvons donc affirmer que le vecteur v_i se trouve en multipliant cette matrice par le vecteur précédent, v_{i-1}:

$$v_i = R_i v_{i-1}.$$

Nous savons :

$$\cos\alpha = \frac{1}{\sqrt{1+\tan^2\alpha}} \quad \text{et} \quad \sin\alpha = \frac{\tan\alpha}{\sqrt{1+\tan^2\alpha}}.$$

En remplaçant dans la matrice de rotation et en mettant $\frac{1}{\sqrt{1+\tan^2\gamma_i}}$ en évidence, nous trouvons :

$$R_i = \frac{1}{\sqrt{1+\tan^2\gamma_i}}\begin{pmatrix}1 & -\tan\gamma_i\\ \tan\gamma_i & 1\end{pmatrix}$$

Par conséquent :

$$v_i = \frac{1}{\sqrt{1+\tan^2\gamma_i}}\begin{pmatrix}1 & -\tan\gamma_i\\ \tan\gamma_i & 1\end{pmatrix}v_{i-1}$$

Ensuite, nous pouvons transformer l'équation du vecteur comme suit :

$$v_i = K_i\begin{pmatrix}1 & -2^{-i+1}\\ 2^{-i+1} & 1\end{pmatrix}v_{i-1}$$

En écrivant avec les coordonnées, nous trouvons cela :

$$\begin{pmatrix}x_i\\ y_i\end{pmatrix} = K_i\begin{pmatrix}x_{i-1}-2^{-i+1}y_{i-1}\\ 2^{-i+1}x_{i-1}+y_{i-1}\end{pmatrix}$$

où $K_i = \frac{1}{\sqrt{1+\tan^2\gamma_i}}$ et $\gamma_i = \tan^{-1}2^{-i+1}$.

Pour pouvoir modifier le sens de rotation, nous transformons légèrement :

$$v_i = K_i\begin{pmatrix}1 & -\sigma_i 2^{-i+1}\\ \sigma_i 2^{-i+1} & 1\end{pmatrix}v_{i-1}$$

Ce qui, avec les coordonnées, nous donne donc :

$$\begin{pmatrix}x_i\\ y_i\end{pmatrix} = K_i\begin{pmatrix}x_{i-1}-\sigma_i 2^{-i+1}y_{i-1}\\ \sigma_i 2^{-i+1}x_{i-1}+y_{i-1}\end{pmatrix}$$

où σ_i, qui peut avoir la valeur de +1 ou de -1, indique le sens de la rotation, c'est-à-dire sens trigonométrique positif ou négatif. Afin de déterminer quelle en sera la valeur, il faut soustraire l'angle qui vient d'être trouvé à la valeur de l'angle recherché. D'abord on calcule $\alpha_i = \alpha_{i-1} + \sigma_i\gamma_i$ afin de trouver le nouvel angle. Ensuite, on soustrait cette valeur à α. Si le résultat trouvé est positif, σ_{i+1} vaudra +1, et le sens sera donc trigonométrique positif, tandis que s'il est négatif, σ_{i+1} sera -1 et la rotation se fera dans le sens trigonométrique négatif.

Afin de simplifier les calculs, *K* peut être laissé de côté et calculé en dernier ainsi :

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1}\frac{1}{\sqrt{1+2^{-2i+2}}}$$

Si le nombre d'itérations à effectuer est décidé à l'avance, il est possible d'enregistrer la valeur de *K*, en nous permettant ainsi de nous épargner des calculs.

Les valeurs de γ_i, qui permettent de calculer les valeurs de α_i, sont également généralement stockées préalablement dans l'ordinateur ou dans la calculatrice afin de faciliter les calculs.

Procéder de cette façon, en faisant des rotations toujours plus petites dans un sens ou dans l'autre, nous permet de nous approcher de plus en plus de l'angle dont on cherche le sinus et le cosinus.

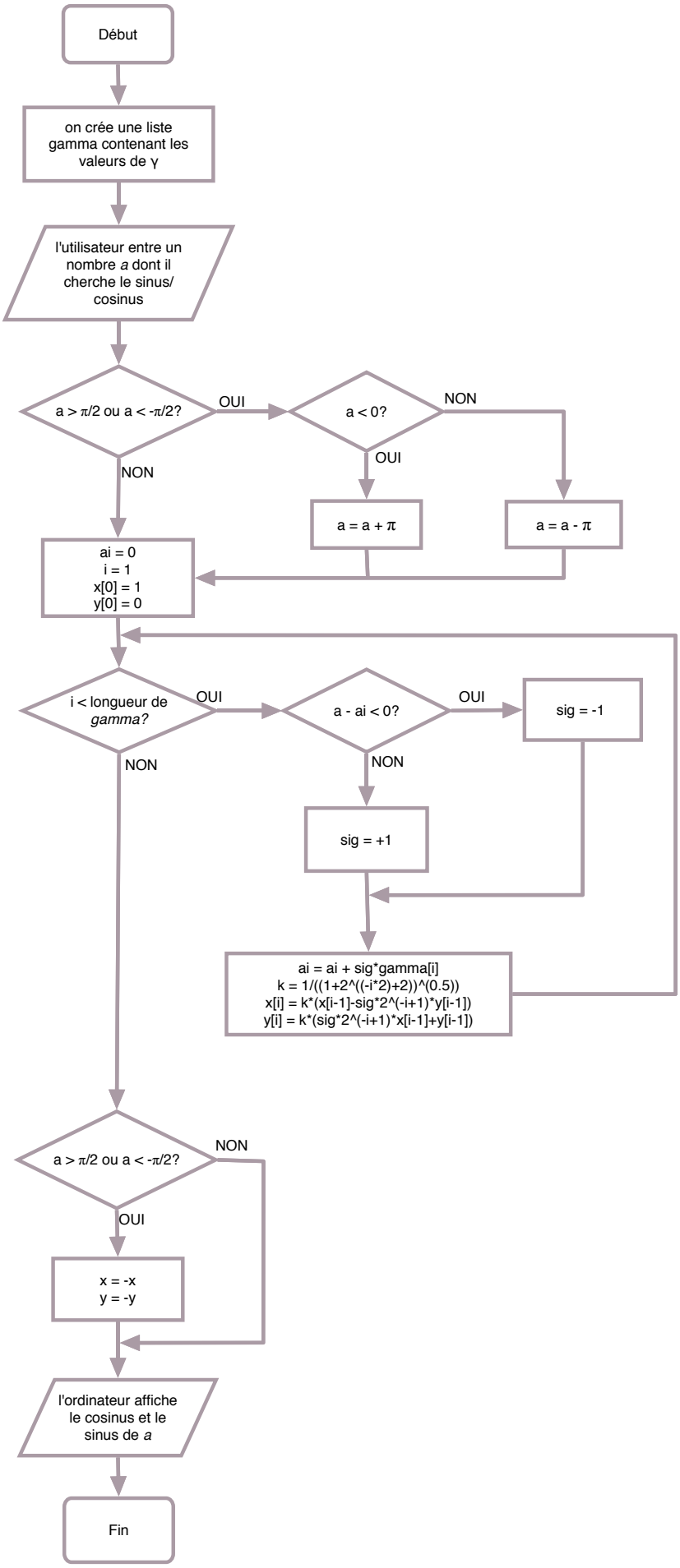
Quand nous avons fait assez d'itérations, la coordonnée *x* du vecteur v_i nous donne la valeur du cosinus, tandis que la valeur de la coordonnée *y* donne celle du sinus.

L'algorithme que j'ai expliqué ci-dessus fonctionne pour des nombres appartenant à l'intervalle $[-\sum\gamma_i\ ;\sum\gamma_i]$, ce qui, pour un $n > 3$, correspond aux premier et dernier quadrants du cercle trigonométrique, c'est-à-dire à l'intervalle $[-\frac{\pi}{2};\frac{\pi}{2}]$.

Si l'on souhaite calculer le sinus et le cosinus d'un autre angle, il faudra le ramener à un angle compris dans l'intervalle ; si notre angle α est plus petit que 0, on lui ajoutera π et sinon, on le lui soustraira. Pour finir, à la fin de l'algorithme il faudra inverser les signes des valeurs *x* et *y*, c'est-à-dire multiplier le vecteur final par -1.

Cela n'est, cependant, valable que pour les angles se trouvant dans l'intervalle $[-\pi\,;\pi]$. Si nous voulons trouver le sinus et le cosinus d'un autre angle, il faut le ramener dans cet intervalle. Cela pourrait se faire parce que le nouvel angle définit le même endroit sur le cercle trigonométrique, simplement à un tour différent, ce qui ne change en rien ni le sinus, ni le cosinus. Cependant, ici je me limite aux angles se trouvant dans l'intervalle $[-\pi\,;\pi]$.

Dans l'algorithme que j'ai implémenté, j'ai commencé par enregistrer les 40 premières valeurs de γ dans un tableau. Cela nous évite de devoir calculer des arctangentes. Le nombre d'étapes qu'effectuera le programme sera donc égal à 40.
L'utilisateur entre ensuite un angle *a* en radians, dont il cherche le sinus et cosinus.
Si l'angle ne se situe ni dans le premier quadrant, ni dans le quatrième, on lui ajoutera ou soustraira π afin de le ramener dans cet intervalle.
Afin de ne pas devoir programmer avec des vecteurs et des matrices, j'ai travaillé en coordonnées *x* et *y*. On affecte à *x* 1 comme première valeur, et 0 à *y*.
On entre ensuite dans une boucle définissant le nombre d'étapes que nous allons effectuer.
Dans celle-ci, on commence par donner une valeur de +1 ou de -1 à *sig* qui détermine le sens de la rotation du vecteur.
Après cela, on calcule *x* et *y* au moyen de la formule expliquée plus haut, ainsi que *ai* qui est l'angle auquel on se trouve à l'étape *i*.
A la fin de l'algorithme, si l'angle *a* qu'avait entré l'utilisateur n'appartenait pas aux premier ou dernier quadrant, on inverse le signe de *x* et de *y*, puis on affiche le résultat du cosinus (*x*) et du sinus (*y*).


Début
on crée une liste gamma contenant les valeurs de γ
l'utilisateur entre un nombre *a* dont il cherche le sinus/ cosinus
a > π/2 ou a < -π/2?
OUI
a < 0?
NON
OUI
NON
a = a + π
a = a - π
ai = 0
i = 1
x[0] = 1
y[0] = 0
i < longueur de *gamma?*
OUI
a - ai < 0?
OUI
sig = -1
NON
NON
sig = +1
ai = ai + sig*gamma[i]
k = 1/((1+2^((-i*2)+2))^(0.5))
x[i] = k*(x[i-1]-sig*2^(-i+1)*y[i-1])
y[i] = k*(sig*2^(-i+1)*x[i-1]+y[i-1])
a > π/2 ou a < -π/2?
NON
OUI
x = -x
y = -y
l'ordinateur affiche le cosinus et le sinus de *a*
Fin

Logarithme

En connaissant le logarithme naturel de 10, il est possible de trouver le logarithme de n'importe quel nombre en le ramenant à un nombre compris entre 1 et 10 : pour chaque $X \in \mathbb{R}_+^*$ il existe un entier $n \in \mathbb{Z}$ de sorte que :

$$1 \leq 10^n X < 10.$$

Au moyen des propriétés du logarithme, nous pouvons affirmer que :

$$\ln(10^n X) = n \ln(10) + \ln(X) \Rightarrow \ln(X) = \ln(10^n X) - n \ln(10)$$

Si $10^n X = 1$, le calcul est terminé, car $\ln(1) = 0$. Sinon, il faut calculer le logarithme de $10^n X$.

Dorénavant, je vais utiliser x pour $10^n X$ et exclure le cas $x = 1$, dont le logarithme vaut 0. Ce nombre décimal x se trouve donc dans l'intervalle]1 ; 10[.

L'algorithme que je vais expliquer ci-dessous se présente comme suit :

> « On se donne x compris entre 1 et 10.
> On affecte la valeur ln 10 à la variable y.
>
> Pour i allant de 0 jusqu'à 10 (ou jusqu'à N), faire :
> $1 + 10^{-i} \to z$
> Tant que $xz \leq 10$:
> $xz \to x$;
> $y - \ln(z) \to y$;
>
> Le résultat cherché est y. »

http://www.mlfmonde.fr/IMG/pdf/31_40_ams62.pdf

Tout comme pour l'algorithme du sinus et du cosinus, certaines valeurs sont préalablement stockées dans la mémoire de la calculatrice. Ces valeurs sont $\ln(10)$ et de $\ln(1 + 10^{-i})$. On choisit 10^{-i}, parce que, dans notre système décimal, cela consiste en des décalages de virgule. Dans le système binaire ce serait donc 2^{-i} qui serait utilisé.

Nous pouvons ensuite remarquer que l'algorithme est constitué de 2 boucles.

Dans la boucle intérieure (la deuxième), on soustrait $\ln(1 + 10^{-i})$ à y et on multiplie x par $1 + 10^{-i}$, jusqu'à ce qu'on dépasse 10. Afin de rendre l'explication plus lisible, je vais remplacer 10^{-i} par α, un réel positif plus grand que 0. A la première sortie de la boucle intérieure, on aura donc :

$$x_n = x(1 + \alpha)^n \text{ et } y_n = \ln(10) - n \ln(1 + \alpha)$$

où $n \in \mathbb{N}$ est tel que :

$$x(1 + \alpha)^n \leq 10 < x(1 + \alpha)^{n+1}$$

Ce qui est équivalent à un encadrement de x :

$$\frac{10}{(1 + \alpha)^{n+1}} < x \leq \frac{10}{(1 + \alpha)^n}$$

Puisque la fonction logarithmique est une fonction strictement croissante, nous obtenons :

$$\ln\left(\frac{10}{(1 + \alpha)^{n+1}}\right) < \ln(x) \leq \ln\left(\frac{10}{(1 + \alpha)^n}\right)$$

$$\Leftrightarrow$$

$$\ln(10) - (n + 1)\ln(1 + \alpha) < \ln(x) \leq \ln(10) - n \ln(1 + \alpha)$$

$$y_n - \ln(1 + \alpha) < \ln(x) \leq y_n$$

Cela signifie que la valeur ln(x) se trouve dans l'intervalle $]y_n - \ln(1 + \alpha) ; y_n]$.

Comme $y_n - \alpha < y_n - \ln(1+\alpha)$, nous avons également ln(x) ∈ $]y_n - \alpha; y_n]$. Cela nous permet de voir la précision du résultat, qui correspond donc à α, c'est-à-dire à 10^{-i}.
On repart ensuite dans la deuxième boucle avec un nouvel *x*.
Après *N* boucles, nous aurons donc:

$$x_N = x\prod_{i=1}^{N}(1+\alpha_i)^{n_i}$$

$$y_N = \ln(10) - \sum_{i=1}^{N} n_i \ln(1+\alpha_i)$$

Nous pouvons donc poser l'inégalité suivante :

$$x\prod_{i=1}^{N}(1+\alpha_i)^{n_i} \leq 10 < x\prod_{i=1}^{N}(1+\alpha_i)^{n_i}\cdot(1+\alpha_N)$$

Si nous la développons de façon analogue à celle utilisée précédemment, nous trouvons finalement :

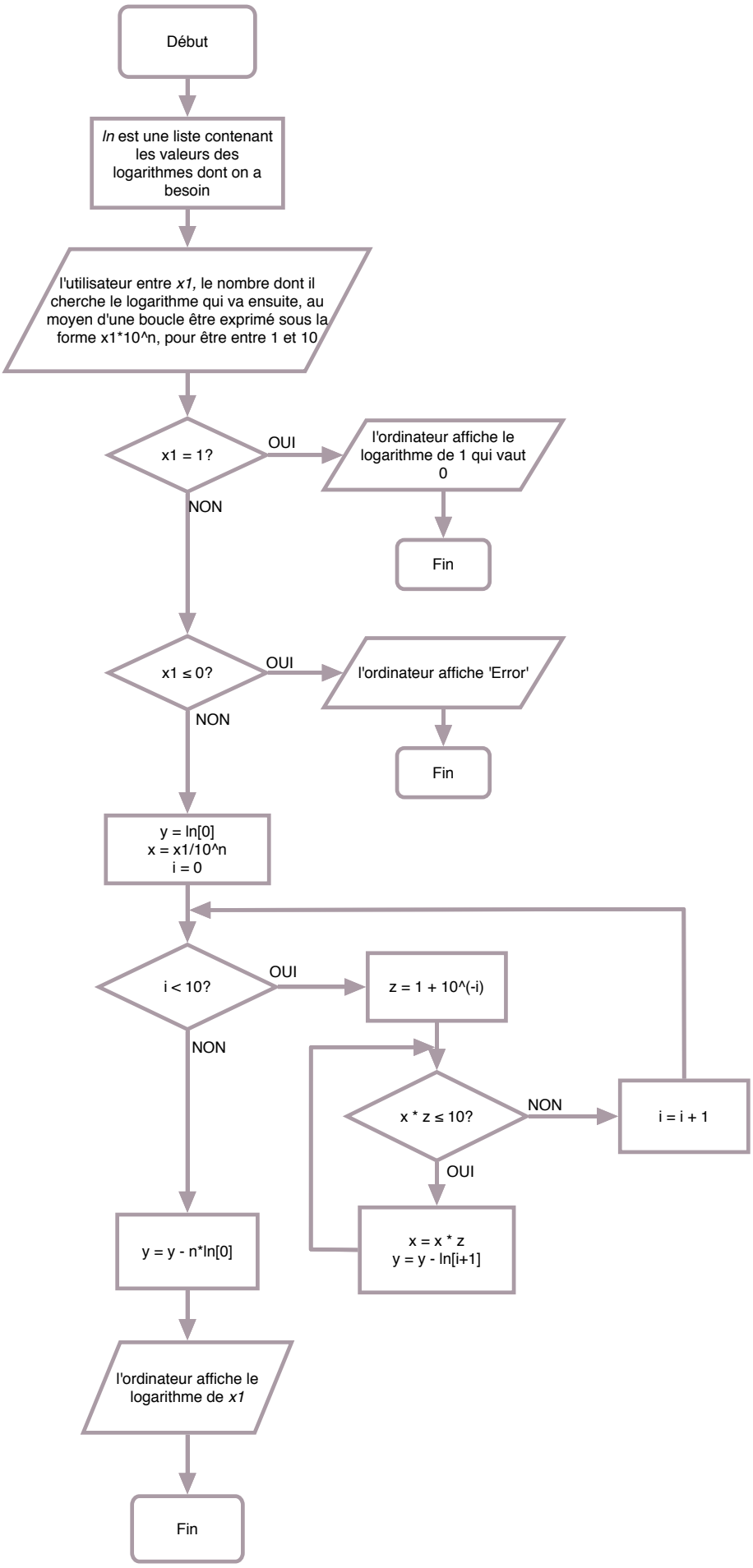
$$y_N - \ln(1+\alpha_N) < \ln(x) \leq y_n$$

Par conséquent, à la fin de l'algorithme, on aura :

$$\ln(x) \in]y_N - \alpha_N; y_N]$$

Cette première boucle permet de s'approcher de la valeur du logarithme par petits pas, avec des approximations de plus en plus proches. Si on se limitait à la deuxième boucle seulement, pour une précision élevée, il faudrait un nombre trop important d'étapes jusqu'à ce que *xz* dépasse 10. La première boucle sert donc à limiter le nombre de pas pour trouver la solution.

Dans l'algorithme que j'ai implémenté, j'ai d'abord inséré dans un tableau les logarithmes naturels des nombres dont nous avions besoin, c'est-à-dire les logarithmes de 10, 2 $(= 1 + 10^0)$, 1.1 $(= 1 + 10^{-1})$, 1.01 $(= 1 + 10^{-2})$,… jusqu'à celui de $1 + 10^{-10}$.
L'utilisateur entre ensuite un nombre *x1* dont il cherche le logarithme.
On fait tout d'abord un test pour voir si x1=1. Si c'est le cas, le logarithme vaut 0 et on arrête le programme.
On fait un autre test pour voir si x1 > 0, sinon l'ordinateur affiche une erreur et quitte le programme.
Au moyen d'une boucle, on détermine quelle sera la valeur de *x* ; on va transformer le nombre introduit par l'utilisateur en un nombre compris entre 1 et 10 en le mettant sous la forme $x1 \cdot 10^n$. C'est-à-dire qu'on multiplie le nombre entré par l'utilisateur par une puissance de dix le ramenant dans l'intervalle souhaité.
Après cela, on entre dans une boucle allant de 0 à 10 et qui va calculer l'algorithme de la manière énoncée ci-dessus.
A la sortie de la boucle, on aura le logarithme de *x* et, afin de trouver celui du nombre entré par l'utilisateur, il suffit de lui soustraire $n\ln(10)$ et d'afficher ensuite le résultat.


Début
ln est une liste contenant les valeurs des logarithmes dont on a besoin
l'utilisateur entre *x1*, le nombre dont il cherche le logarithme qui va ensuite, au moyen d'une boucle être exprimé sous la forme x1*10^n, pour être entre 1 et 10
x1 = 1?
OUI
l'ordinateur affiche le logarithme de 1 qui vaut 0
Fin
NON
x1 ≤ 0?
OUI
l'ordinateur affiche 'Error'
Fin
NON
y = ln[0]
x = x1/10^n
i = 0
i < 10?
OUI
z = 1 + 10^(-i)
NON
x * z ≤ 10?
NON
i = i + 1
OUI
x = x * z
y = y - ln[i+1]
y = y - n*ln[0]
l'ordinateur affiche le logarithme de *x1*
Fin

Conclusion et bilan personnel

Malgré le fait que les calculatrices portables soient des objets courants, leur fonctionnement est assez complexe et peu connu. Il y a peu d'ouvrages concernant les algorithmes utilisés dans les calculatrices de poche et les fabricants de ces appareils sont très réticents à donner des renseignements au sujet des algorithmes de calcul. En effet, j'espérais, initialement, pouvoir découvrir les algorithmes utilisés dans le modèle TI-30XS Multiview qui est la calculatrice que nous utilisons pour les cours, et j'ai donc écrit un message à Texas Instruments leur demandant s'il leur était possible de me donner des informations, ce qui n'a malheureusement pas été le cas.
C'est la raison pour laquelle j'ai dû utiliser des ressources bibliographiques ainsi que des sites internet donnant des informations génériques sur le mode de fonctionnement des calculatrices.
La recherche d'informations n'a pas été aisée et quelques algorithmes ont été initialement assez difficiles à comprendre ; toutefois j'ai eu beaucoup de plaisir et de satisfaction à découvrir chaque algorithme et à pouvoir ensuite le répliquer dans le langage de programmation Python.

On peut constater que certains algorithmes sont plus simples que d'autres, tels que ceux de l'addition, de la multiplication et de la division qui sont très semblables à la façon dont on apprend à les calculer à l'école primaire.
D'autres par contre, comme CORDIC ou celui nécessaire au calcul de la racine n-ième, nécessitent plus de connaissances mathématiques et plus de réflexion.
De plus, on a pu noter que l'utilisation du binaire permet de faciliter les calculs, puisqu'il ne se sert que de 2 chiffres et l'on peut donc procéder également par comparaison plutôt que par calcul.

Malencontreusement, en raison de contraintes de temps, je n'ai pas pu me pencher sur les opérations avec les nombres négatifs et les nombres décimaux.
Les derniers algorithmes de calcul, plus complexes, ont été reproduits en système décimal, afin de simplifier leur implémentation et pour avoir le temps de bien les comprendre. En outre, il manque également l'analyse de la convergence de certains algorithmes, tels que CORDIC et celui de la racine n-ième.
Pour finir, la liste des algorithmes utilisés dans une calculatrice n'est pas exhaustive et il y aurait donc encore beaucoup d'autres algorithmes à découvrir et à comprendre, tels que par exemple ceux liés à la probabilité,…

Bibliographie

Livres

1. MAXFIELD Clive et BROWN Alvin, *The Definitive Guide to How Computers Do Math: Featuring the Virtual DIY Calculator*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2005
2. PIGUET Christian et HÜGLI Heinz, *Du zéro à l'ordinateur : une brève histoire du calcul*, Presses polytechniques et universitaires romandes, Lausanne, 2004

Articles

1. BOPP Nicole, “Algorithme Cordic Pour Calculer Le Logarithme.” Activités Mathématiques et Scientifiques 62 (2007): 31-40

Sites Internet

1. Wikipedia:
   - http://en.wikipedia.org/wiki/Method_of_complements (soustraction)
   - http://en.wikipedia.org/wiki/Multiplication_algorithm (multiplication)
   - http://en.wikipedia.org/wiki/Division_algorithm (division)
   - http://en.wikipedia.org/wiki/Exponentiation_by_squaring (exponentiation)
   - http://fr.wikipedia.org/wiki/Algorithme_de_calcul_de_la_racine_n-ième (racine n-ième)
   - http://en.wikipedia.org/wiki/CORDIC (CORDIC sinus et cosinus)
2. http://www.physique.usherbrooke.ca/~afaribau/essai/essai.html (histoire de la calculatrice)
3. http://history-computer.com/MechanicalCalculators/Pioneers/Schickard.html (histoire de la calculatrice)

Annexes

Addition

```
augend=input('Tapez un nombre: ') #augend est le premier nombre auquel on va additionner le addend
addend=input('Tapez un deuxieme nombre: ')
addendbin=int(bin(addend)[2:]) #on transforme les deux nombres en binaire
augendbin=int(bin(augend)[2:])

def addition(augendbin,addendbin):
    l_augend=list(str(augendbin)) #chaque chiffre de augend a une case dans la liste l_augend
    l_addend=list(str(addendbin)) #chaque chiffre de addend a une case dans la liste l_addend
    l_augend=l_augend[::-1]
    l_addend=l_addend[::-1]#on inverse le sens des listes
    sum=[]
    q=0
    if len(l_augend)!=len(l_addend): #si les listes n'ont pas la meme longueur, cette boucle ajoute des 0 a la fin de la liste la plus courte
        o=int(max(len(l_augend),len(l_addend))-min(len(l_augend),len(l_addend)))
        s=0
        while s<o:
            s=s+1
            if len(l_augend)<len(l_addend):
                l_augend.append('0')
            else:
                l_addend.append('0')
    carry=0
    while q<len(l_addend): #avec des comparaisons, cette boucle va ajouter a sum les chiffres de la somme
        if int(l_addend[q])==0:
            if int(l_augend[q])==0 and carry==0:
                sum.append(0)
                carry=0
            if int(l_augend[q])==1 and carry==0:
                sum.append(1)
                carry=0
            if int(l_augend[q])==0 and carry==1:
                sum.append(1)
                carry=0
            if int(l_augend[q])==1 and carry==1:
                sum.append(0)
                carry=1
        else:
            if int(l_augend[q])==0 and carry==0:
                sum.append(1)
                carry=0
            if int(l_augend[q])==0 and carry==1:
                sum.append(0)
                carry=1
            if int(l_augend[q])==1 and carry==1:
                sum.append(1)
                carry=1
            if int(l_augend[q])==1 and carry==0:
                sum.append(0)
                carry=1
```

Addition

```
50          q=q+1
51      if carry==1: #si a la fin de toutes les comparaisons il reste un carry de 1, on
…   l'ajoute a la fin de la liste de la somme
52          sum.append(1)
53      result(sum)
54      return result
55
56  def result(sum):
57      sum=sum[::-1]
58      global result
59      result=''
60      for x in range(0,len(sum),1): #on transforme le resultat en int
61          result=result+str(sum[x])
62      result=int(result,2) #on transforme le resultat en nombre decimal
63      return result
64
65  print augend,'+',addend,'=',addition(augendbin,addendbin)
```

Soustraction

```
minuend=input('Tapez un nombre: ') #l'utilisateur doit entrer un minuend auquel on va soustraire le subtrahend
subtrahend=input('Tapez un deuxieme nombre: ')
minuendbin=int(bin(minuend)[2:])
subtrahendbin=int(bin(subtrahend)[2:]) #on transforme les nombres en binaire

def subtraction(minuendbin,subtrahendbin):
    l_minuend=list(str(minuendbin)) #on transforme les nombres en listes, dans lesquelles chaque chiffre a sa propre case
    l_subtrahend=list(str(subtrahendbin))
    l_minuend=l_minuend[::-1] #on inverse le sens des listes, donc les chiffres de plus petit ordre de grandeur sont au debut
    l_subtrahend=l_subtrahend[::-1]
    l_complement=[] #l_complement sera la liste contenant le complement de 1 du subtrahend
    p=0
    if len(l_minuend)>len(l_subtrahend): #si l_minuend est plus long que l_subtrahend, on ajoute des 0 a la fin de l_subtrahend jusqu'a ce qu'elles aient la meme longueur
        o=int(len(l_minuend)-len(l_subtrahend))
        r=0
        while r<o:
            r=r+1
            l_subtrahend.append('0')
    while p<len(l_subtrahend): #cette boucle permet de trouver le complement de 1 du subtrahend
            if int(l_subtrahend[p])==0:
                l_complement.append('1')
            if int(l_subtrahend[p])==1:
                l_complement.append('0')
            p=p+1
    addition(l_minuend,l_complement)
    un=[1]
    if len(sum)>len(un): #si l_minuend est plus long que l_subtrahend, on ajoute des 0 a la fin de l_subtrahend jusqu'a ce qu'elles aient la meme longueur
        o=int(len(sum)-len(un))
        t=0
        while t<o:
            t=t+1
            un.append('0')
    addition(sum,un)
    result(sum)
    return somme

def addition(x,y): #cette fonction additionne l_minuend et l_complement
    q=0
    global sum
    sum=[]
    carry=0
    while q<len(x): #avec des comparaisons, cette boucle va ajouter a sum les chiffres de la somme
        if int(x[q])==0:
            if int(y[q])==0 and carry==0:
                sum.append(0)
                carry=0
```

Soustraction

```
            if int(y[q])==1 and carry==0:
                sum.append(1)
                carry=0
            if int(y[q])==0 and carry==1:
                sum.append(1)
                carry=0
            if int(y[q])==1 and carry==1:
                sum.append(0)
                carry=1
        else:
            if int(y[q])==0 and carry==0:
                sum.append(1)
                carry=0
            if int(y[q])==0 and carry==1:
                sum.append(0)
                carry=1
            if int(y[q])==1 and carry==1:
                sum.append(1)
                carry=1
            if int(y[q])==1 and carry==0:
                sum.append(0)
                carry=1
        q=q+1
    if carry==1: #si a la fin de toutes les comparaisons il reste un carry de 1, on
… l'ajoute a la fin de la liste de la somme
        sum.append(1)
    return sum

def result(sum):
    sum=sum[::-1]
    global somme
    somme=''
    for x in range(1,len(sum),1):
        somme=somme+str(sum[x])
    somme=int(somme,2)
    return somme
print minuend,'-',subtrahend,'=',subtraction(minuendbin,subtrahendbin)

```

Multiplication

```
multiplicand=input('Tapez un nombre: ')
multiplier=input('Tapez un deuxieme nombre: ') #l'utilisateur doit entrer deux nombres pour les multiplier
multiplicandbin=int(bin(multiplicand)[2:])
multiplierbin=int(bin(multiplier)[2:]) #on transforme les deux nombres en binaire

def multiplication(multiplicandbin,multiplierbin):
    l_multiplicand=list(str(multiplicandbin))
    l_multiplier=list(str(multiplierbin)) #on transforme les deux nombres en listes
    l_multiplier=l_multiplier[::-1] #on inverse le sens des listes
    product=0
    for q in range(0,len(l_multiplier),1): #cette boucle multipliera les deux nombres
        partial_product=''
        if int(l_multiplier[q])==0: #si le chiffre du multiplier est egal a 0, le produit partiel sera aussi egal a 0
            partial_product=0
        else: #si le chiffre est egal a 1, le produit partiel sera egal au multiplicand
            partial_product=str(multiplicandbin)
            for n in range(0,q,1): #cette boucle permet de faire le decalage des produits partiels en leur ajoutant des 0
                partial_product=partial_product+'0'
        partial_product=int(partial_product)
        addition(product,partial_product) #on additionne le produit partiel au produit
        product=result
    product=int(str(product),2) #on transforme le produit en nombre decimal
    return product

def addition(augendbin,addendbin): #cette fonction permet d'additionner le produit et le produit partiel
    l_augend=list(str(augendbin)) #chaque chiffre de augend a une case dans la liste l_augend
    l_addend=list(str(addendbin)) #chaque chiffre de addend a une case dans la liste l_addend
    l_augend=l_augend[::-1]
    l_addend=l_addend[::-1]
    sum=[]
    q=0

    if len(l_augend)!=len(l_addend):
        o=int(max(len(l_augend),len(l_addend))-min(len(l_augend),len(l_addend)))
        s=0
        while s<o:
            s=s+1
            if len(l_augend)<len(l_addend):
                l_augend.append('0')
            else:
                l_addend.append('0')
    carry=0
    while q<len(l_addend): #avec des comparaisons, cette boucle va ajouter a sum les chiffres de la somme
        if int(l_addend[q])==0:
            if int(l_augend[q])==0 and carry==0:
                sum.append(0)
                carry=0
```

```
            if int(l_augend[q])==1 and carry==0:
                sum.append(1)
                carry=0
            if int(l_augend[q])==0 and carry==1:
                sum.append(1)
                carry=0
            if int(l_augend[q])==1 and carry==1:
                sum.append(0)
                carry=1
        else:
            if int(l_augend[q])==0 and carry==0:
                sum.append(1)
                carry=0
            if int(l_augend[q])==0 and carry==1:
                sum.append(0)
                carry=1
            if int(l_augend[q])==1 and carry==1:
                sum.append(1)
                carry=1
            if int(l_augend[q])==1 and carry==0:
                sum.append(0)
                carry=1
        q=q+1
    if carry==1: #si a la fin de toutes les comparaisons il reste un carry de 1, on
… l'ajoute a la fin de la liste de la somme
        sum.append(1)
    sum=sum[::-1]
    global result
    result=''
    for x in range(0,len(sum),1):
        result=result+str(sum[x])
    result=int(result)
    return result
print multiplicand,'*',multiplier,'=',multiplication(multiplicandbin,multiplierbin)
```

Division

```
dividende=input('Tapez un nombre: ')
diviseur=input('Tapez un deuxieme nombre: ') #l'utilisateur doit entrer les 2 nombres qu'il souhaite diviser
while diviseur==0: #cette boucle affiche une erreur et demande a l'utilisateur d'entrer un nouveau diviseur si celui ci etait egal a 0
    print 'DIVIDE BY 0 ERROR'
    diviseur=input('Tapez un deuxieme nombre: ')
dividendebin=int(bin(dividende)[2:])
diviseurbin=int(bin(diviseur)[2:]) #on transforme les deux nombres en binaire

def division(dividendebin,diviseurbin):
    if dividendebin<diviseurbin:
        print 0, 'reste: ',dividende
    else:
        l_dividende=list(str(dividendebin))
        l_diviseur=list(str(diviseurbin)) #on transforme les deux nombres en listes
        b=len(l_diviseur)
        global tranche
        tranche='' #ce sera la partie qui sera divisee par le diviseur
        l_quotient=[] #la liste dans laquelle on inserera les differents chiffres du quotient
        r=b-1
        for x in range(0,b,1): #cette boucle ajoute des chiffres du dividende a tranche, jusqu'a ce qu'il soit de la meme longueur que le diviseur
            tranche=str(tranche)+str(l_dividende[x])
        tranche=int(tranche)
        if tranche<diviseurbin: #si tranche est plus petite que le diviseur, on lui ajoute encore un chiffre
            tranche=str(tranche)+str(l_dividende[b])
            tranche=int(tranche)
            r=r+1
        while r<len(l_dividende): #cette boucle fait la division en regardant si le diviseur rentre dans la tranche ou pas et en ajoutant le chiffre correct au quotient
            r=r+1
            if diviseurbin<=tranche:
                l_quotient.append('1')
                subtraction(tranche,diviseurbin)
                tranche=somme
            else:
                l_quotient.append('0')
            try:
                tranche=str(tranche)+str(l_dividende[r])
                tranche=int(tranche)
            except:
                break
        quotient=''
        for x in range(0,len(l_quotient),1): #cette boucle transforme la quotient en nombre int
            quotient=quotient+str(l_quotient[x])
        quotient=int(quotient,2) #on transforme le quotient en chiffre decimal
        tranche=str(tranche)
        tranche=int(tranche,2)
        print quotient, 'reste: ',tranche

```

Division

```
48 def subtraction(minuendbin,subtrahendbin):
49     l_minuend=list(str(minuendbin)) #on transforme les nombres en listes, dans
   … lesquelles chaque chiffre a sa propre case
50     l_subtrahend=list(str(subtrahendbin))
51     l_minuend=l_minuend[::-1] #on inverse le sens des listes, donc les chiffres de
   … plus petit ordre de grandeur sont au debut
52     l_subtrahend=l_subtrahend[::-1]
53     l_complement=[] #l_complement sera la liste contenant le complement de 1 du
   … subtrahend
54     p=0
55     if len(l_minuend)>len(l_subtrahend): #si l_minuend est plus long que l_subtrahend,
   … on ajoute des 0 a la fin de l_subtrahend jusqu'a ce qu'elles aient la meme longueur
56         o=int(len(l_minuend)-len(l_subtrahend))
57         r=0
58         while r<o:
59             r=r+1
60             l_subtrahend.append('0')
61     while p<len(l_subtrahend): #cette boucle permet de trouver le complement de 1 du
   … subtrahend
62             if int(l_subtrahend[p])==0:
63                 l_complement.append('1')
64             if int(l_subtrahend[p])==1:
65                 l_complement.append('0')
66             p=p+1
67     addition(l_minuend,l_complement)
68     un=[1]
69     if len(sum)>len(un): #si l_minuend est plus long que l_subtrahend, on ajoute des 0
   … a la fin de l_subtrahend jusqu'a ce qu'elles aient la meme longueur
70         o=int(len(sum)-len(un))
71         t=0
72         while t<o:
73             t=t+1
74             un.append('0')
75     addition(sum,un)
76     result(sum)
77     return somme
78
79 def addition(x,y): #cette fonction additionne l_minuend et l_complement
80     q=0
81     global sum
82     sum=[]
83     carry=0
84     while q<len(x): #avec des comparaisons, cette boucle va ajouter a sum les chiffres
   … de la somme
85         if int(x[q])==0:
86             if int(y[q])==0 and carry==0:
87                 sum.append(0)
88                 carry=0
89             if int(y[q])==1 and carry==0:
90                 sum.append(1)
91                 carry=0
92             if int(y[q])==0 and carry==1:
93                 sum.append(1)
94                 carry=0
```

```
            if int(y[q])==1 and carry==1:
                sum.append(0)
                carry=1
        else:
            if int(y[q])==0 and carry==0:
                sum.append(1)
                carry=0
            if int(y[q])==0 and carry==1:
                sum.append(0)
                carry=1
            if int(y[q])==1 and carry==1:
                sum.append(1)
                carry=1
            if int(y[q])==1 and carry==0:
                sum.append(0)
                carry=1
        q=q+1
    if carry==1: #si a la fin de toutes les comparaisons il reste un carry de 1, on l'ajoute a la fin de la liste de la somme
        sum.append(1)
    return sum

def result(sum):
    sum=sum[::-1]
    global somme
    somme=''
    for x in range(1,len(sum),1):
        somme=somme+str(sum[x])
    somme=int(somme)
    return somme

print dividende,'/',diviseur,'=',division(dividendebin,diviseurbin)
```

Exponentiation

```
b=input('Tapez un nombre: ') #l'utilisateur entre la base
n=input('Tapez un deuxieme nombre: ') #il entre l'exposent
def exponentiation(b,n): #cette fonction recursive permet de calculer b^n
    if n<0:
        return exponentiation(1./b,-n)
    elif n==0:
        return 1
    elif n==1:
        return b
    elif n%2==0:
        return exponentiation(b*b,n/2)
    else:
        return b*exponentiation(b*b,(n-1)/2)
print b,'^',n,'=',exponentiation(b,n)
```

Racine n-ième

```
a=input('Tapez un nombre: ') #l'utilisateur entre le nombre dont il cherche la racine
n=input('Tapez la racine que vous souhaitez du nombre: ') #il met quelle racine il souhaite
b=a
x=abs(a/2.) #on prend une premiere variable pour x
if a<0 and n%2==0: #si a est plus petit que 0 et n impair, l'ordinateur affiche une erreur et quitte le programme
    print 'Error'
    exit()
elif a<0 and n%2!=0:
    a=-a
elif a>0 and a<1:
    x=1
def racine(a,n,x):
    x1=(1./n)*((n-1)*x+(a/(x**(n-1)))) #on attribue a x sa premiere valeur
    while abs(x-x1)>0.00000000000000000001: #tant que la difference des deux dernieres valeurs de x est plus grande que 10^-20 le programme effectue ce qui suit
        x=x1
        x1=(1./n)*((n-1)*x+(a/(x**(n-1)))) #on calcule la racine
    if b<0:
        x=-x
    return x

print 'La racine',n,'ieme de',b,'vaut:',racine(a,n,x)
```

Sinus et Cosinus

```
gamma=[0.0,0.78539816339745,0.46364760900081,0.24497866312686,0.12435499454676,0.06241880999596,
0.03123983343027,0.01562372862048,0.00781234106010,0.00390623013197,0.00195312251648,
0.00097656218956,0.00048828121119,0.00024414062015,0.00012207031189,0.00006103515617,
0.00003051757812,0.00001525878906,0.00000762939453,0.00000381469727,0.00000190734863,
0.00000095367432,0.00000047683716,0.00000023841858,0.00000011920929,0.00000005960464,
0.00000002980232,0.00000001490116,0.00000000745058,0.00000000372529,0.00000000186265,
0.00000000093132,0.00000000046566,0.00000000023283,0.00000000011641,0.00000000005821,
0.00000000002910,0.00000000001455,0.00000000000728,0.00000000000364,0.00000000000182 ]
#ce tableau contient les valeurs de gamma pour ne pas devoir calculer des arctangentes


a=input('Tapez un angle en radians: ') #l'utilisateur entre le nombre dont il cherche le sinus et le cosinus
b=a
if a<-1.570796326795 or a>1.570796326795: #si l'angle ne se trouve pas dans le premier ou dernier cadran du cercle trigonometrique, on lui ajoute ou soustrait pi pour l'y ramener
    if a<0:
        a=a+3.14159265359
    else:
        a=a-3.14159265359

x1=1
y1=0
ai=0
for i in range(1,len(gamma),1):
    if a-ai<0: #ce test permet d'attribuer une valeur a sig qui definira le sens de rotation
        sig=-1
    else:
        sig=1
    ai=ai+sig*gamma[i] #on calcule l'angle auquel on se trouve
    k=1./((1+2**((-i*2)+2))**(1./2))
    x=k*(x1-sig*2**(-i+1)*y1) #on calcule la valeur de x qui donnera ensuite le cosinus de a
    y=k*(sig*2**(-i+1)*x1+y1) #on calcule la valeur de y qui donnera ensuite le sinus de a
    x1=x
    y1=y
if a!=b:
    x1=-x1
    y1=-y1
print 'cos(',b,') =',x1
print 'sin(',b,') =',y1 #l'ordinateur affiche les valeurs de cosinus et du sinus
```

Logarithme

```
ln=[2.302585093,0.693147181,0.095310179,0.009950331,0.000999500,0.000099995,
0.00000999995,0.0000009999995,0.000000099999995,0.00000000999999995,0.000000001]
#table des logarithmes necessaires
x1=input('Tapez un nombre: ') #l'utilisateur doit entrer le nombre dont il cherche le
logarithme
x=str(x1)
if x1==1:
    print 'ln(1) = 0'
    exit()
elif x1<=0:
    print 'Error'
    exit()
elif x1<1:#les trois tests qui suivent permettent d'attribuer a n sa valeur
    n=1
    for g in range(2,len(x),1):
        if x[g]==0:
            n=n+1
        else:
            break
elif x1>=10:
    n=-1
    for g in range(0,len(x),1):
        if x[g]==',':
            break
        else:
            n=n-1
else:
    n=0
y=ln[0]
x=float(x)*(10**n)
for i in range(0,10,1): #cette boucle exterieure permet de s'approcher par petits pas a
la valeur du logarithme
    z=1+10**(-i)
    while x*z<=10: #cette boucle permet de calculer le logarithme
        x=x*z
        y=y-ln[i+1]
y=y-n*ln[0]
print 'ln(',x1,') =',y #l'ordinateur affiche la valeur du logarithme de x
```