Algorithme d'Euclide

- Algorithme d'Euclide : calcul du PGCD de deux entiers. Version itérative et version récursive.
- Théorème de Bézout : algorithme d'Euclide étendu. Version itérative et version récursive.

Algorithme d'Euclide

Le calcul du PGCD de deux entiers positifs a et b utilise l'algorithme d'Euclide, remarquablement général (il fonctionne aussi pour les polynômes) et efficace. Soit r le reste de la division euclidienne de a par b :

$$a = bq + r \; , r < b.$$

Tout diviseur commun de a et b divise aussi $r = a - bq$, et réciproquement tout diviseur commun de b et r divise aussi $a = bq + r$. Donc le calcul du PGCD de a et b se ramène à celui du PGCD de b et r ; et on peut recommencer sans craindre une boucle sans fin, car les restes successifs sont strictement décroissants. Le dernier reste non nul obtenu est le PGCD cherché.

Par exemple si a=96 et b=81, les calculs sont les suivants :

a				b		r
96	=	1	*	81	+	15
81	=	5	*	15	+	6
15	=	2	*	6	+	3
6	=	2	*	3	+	0

et le PGCD vaut 3. Voici le programme C :

```
long pgcd (long a, long b) {
  long r;

  while (1) {
    r = a % b;
    if (r == 0)
      return b;
    a = b;
    b = r;
  }
}
```

et une variante plus élégante :

```
long pgcd (long a, long b) {
  long r;

  while (b > 0) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

Commentaires de programmation

- La première version utilise une boucle inconditionnelle `while (1)` : l'expression booléenne entre parenthèses est toujours vraie. Bien entendu le corps de la boucle doit contenir un test de sortie ; l'instruction `return` termine l'exécution de la fonction, quelle que soit sa place.

- Attention à l'ordre des instructions `a = b; b = r;` attention à la valeur retournée dans la seconde version (b vaut 0 en sortie de boucle).
- Il est inutile de traiter à part le cas $a < b$: la première itération échange a et b dans ce cas.

Complexité

On prouve facilement par récurrence que le cas le pire (c'est à dire celui où le nombre d'itérations à exécuter est le plus grand), est celui où a et b sont des termes consécutifs de la suite de Fibonacci. Comparer par exemple la suite de 4 divisions pour a=96 et b=81 avec les 11 opérations nécessaires pour $a=F_{12}$=144 et $b=F_{11}$=89 :

$$
\begin{aligned}
144 &= 1 * 89 + 55\\
89 &= 1 * 55 + 34\\
55 &= 1 * 34 + 21\\
34 &= 1 * 21 + 13\\
21 &= 1 * 13 + 8\\
13 &= 1 * 8 + 5\\
8 &= 1 * 5 + 3\\
5 &= 1 * 3 + 2\\
3 &= 1 * 2 + 1\\
2 &= 1 * 1 + 1\\
1 &= 1 * 1 + 0
\end{aligned}
$$

Donc si a et b sont inférieurs au $n^{\text{ème}}$ nombre de Fibonacci F_n , l'algorithme d'Euclide effectue au plus n-1 itérations ; en particulier le calcul du pgcd de deux entiers représentables sur 32 chiffres binaires exige au pire 46 divisions (et beaucoup moins en général : on peut prouver, ou vérifier expérimentalement, que le nombre *moyen* de divisions vaut 18).

Comme F_n est une fonction exponentielle de n, la complexité au pire de l'algorithme d'Euclide est proportionnelle au *logarithme* de a (en supposant $a>b$), autrement dit proportionnelle au *nombre de chiffres* nécessaires pour écrire a (par exemple en base 10). L'algorithme d'Euclide est donc très efficace : si l'on dispose d'une bibliothèque d'opérations sur les entiers de taille quelconque (comme en *Maple* ; on parle alors de calculs en *multiprécision*) calculer le PGCD de nombres de 100 chiffres ne demande que quelques centaines d'opérations.

Version récursive

Une fonction peut, pendant son exécution, faire appel à une autre fonction, qui elle-même en appelle d'autres, etc. Le mécanisme qui garantit que ces appels en cascade fonctionnent correctement, autorise du même coup une fonction à s'appeler elle-même ; un tel appel, et la fonction ainsi définie, sont dits *récursifs*. Pour éviter les cercles vicieux, une fonction récursive doit toujours comporter un cas particulier où le résultat est calculé directement, c'est à dire sans appel récursif ; il faut aussi s'assurer que ce cas particulier finira toujours par se présenter.

```
long pgcd (long a, long b) {
  long r;

  r = a % b;
  if (r == 0)
    return b;
  else
    return pgcd (b, r);
}
```

La version récursive de l'algorithme d'Euclide est peut-être un peu plus facile à écrire que la version itérative. Les deux versions ont fondamentalement la même complexité, avec un petit avantage à la version itérative, car l'appel d'une fonction n'est pas gratuit.

Théorème de Bézout

Le théorème de Bézout affirme que le PGCD d de deux entiers a et b est une combinaison linéaire (à coefficients entiers) de a et b :

$$d = au + bv.$$

Une modification simple de l'algorithme d'Euclide (qu'on appelle alors algorithme d'Euclide *étendu*) permet de calculer ces coefficients u et v. Remarquons d'abord que l'algorithme d'Euclide calcule une suite définie par une récurrence à deux termes :

$$a_0 = a \ , a_1 = b$$

$$a_{n-1} = q_n \, a_n + a_{n+1}$$

autrement dit :

$$a_{n+1} = - \, q_n \, a_n + a_{n-1} \ (*)$$

donc en posant :

$$a_n = a \, u_n + b \, v_n$$

u_n et v_n vérifient la même récurrence (*) que a_n, avec les conditions initiales :

$$u_0 = 1 \ , v_0 = 0$$

$$u_1 = 0 \ , v_1 = 1$$

Exemple (suite) :

		u		v
96 =	96 *	1	+ 81 *	0
81 =	96 *	0	+ 81 *	1
15 =	96 - 81			
	= 96 *	(1 - 0)	+ 81 *	(0 - 1)
	= 96 *	1	+ 81 *	-1
6 =	81 - 5 * 15			
	= 96 *	(0 - 5 * 1)	+ 81 *	(1 - 5 * (-1))
	= 96 *	-5	+ 81 *	6
3 =	15 - 2 * 6			
	= 96 *	(1 - 2 * (-5))	+ 81 *	(-1 - 2 * 6)
	= 96 *	11	+ 81 *	-13

Voici le programme C :

```
long pgcd (long a, long b, long *u, long *v) {
  long q, r, s, t, tmp;
```

```
    *u = 1;
    *v = 0;
    s = 0;
    t = 1;
    while (b > 0) {
      q = a / b;
      r = a % b;
      a = b;
      b = r;
      tmp = s;
      s = *u - q * s;
      *u = tmp;
      tmp = t;
      t = *v - q * t;
      *v = tmp;
    }
    return a;
  }
```

Commentaires de programmation

- Comme la fonction *pgcd* doit calculer les valeurs des deux derniers paramètres, il faut transmettre les *adresses* de ceux-ci, d'où les notations `*u` et `*v`.

  Voici un exemple d'appel de cette fonction :

  ```
  d = pgcd (96, 81, &x, &y);
  ```
  qui aura pour effet d'attribuer les valeurs 3, 11 et -13 aux variables *d, x, y*.

- Après *n* itérations, on a :

  $$\texttt{*u} = u_n \text{ , } \texttt{s} = u_{n+1}$$
  $$\texttt{*v} = v_n \text{ , } \texttt{t} = v_{n+1}$$

  avec les définitions ci-dessus des suites récurrentes u_n et v_n.

Version récursive

On remarque que si l'on a :

$$a = bq + r$$
$$d = bs + rt$$

alors :

$$d = bs + (a\text{-}bq)t = at + b(s\text{-}qt)$$

d'où la fonction C :

```
  long pgcd (long a, long b, long *u, long *v) {
    long q, r, d, s, t;

    q = a / b;
    r = a % b;
    if (r == 0) {
      *u = 0;
      *v = 1;
      return b;
    }
    d = pgcd (b, r, &s, &t);
    *u = t;
    *v = s - q * t;
    return d;
  }
```

Commentaires de programmation

- Comme la première branche de l'alternative `if` se termine par `return`, le symbole `else` est facultatif ; pour illustrer ce point, `else` apparaît dans la version élémentaire de l'algorithme d'Euclide, et n'apparaît pas dans la version *étendue* ci-dessus.

- Noter à nouveau que dans cette fonction, les paramètres *u* et *v* sont des adresses, et que l'appel récursif

  ```
  d = pgcd (b, r, &s, &t);
  ```

  utilise comme arguments les adresses de *s* et *t*.

- Voir en annexe une trace détaillée d'une exécution de cette fonction.