Größter gemeinsamer Teiler ggT und der Euklidische Algorithmus

Der Euklidische Algorithmus:

Es seien ***a >= b >= 0*** ganze Zahlen:
Es gilt: *b == 0:* *ggt(a, b) = a*
b != 0: *ggt(a, b) = ggt(b, a mod b)* (Rekursion)

die ggT-Funktion kann man genauso programmieren. Sie ruft sich dann selbst auf: Rekursion

```
int ggt_recursiv(int a, int b)      // ggT von a >=  b >= 0
{    if (b == 0)
          return a;
     else
          return ggt_recursiv(b, a % b);
}
```

Noch kürzer kann man die Funktion schreiben, wenn man `b == 0` durch das äquivalente `!b` und den `if else`-Block durch den Fragezeichen-Operator `?:` ersetzt. Die Syntax ist

`Bedingung ? Wert_if_true : Wert_if_false`

Damit lassen sich kurze `if else`-Blöcke oft vermeiden:

```
int ggt_recursiv(int a,int b) // ggT von a >=  b > 0
{
     return !b ? a : ggt_recursiv(b, a % b);
}
```

Bei dieser Rekursion erzeugt ein Aufruf von `ggt_recursiv()` nur einen weiteren Aufruf, sodass hier kein exponentielles Wachstum der Aufrufe entsteht. Trotzdem kann man auch hier die Abarbeitung in eine Schleife verlagern und dadurch Ressourcen einsparen:

```
int ggt_loop(int a, int b)     // ggT von a >=  b >= 0
{
   for (int r; b; a = b, b = r)
        r = a % b;
   return a;
}
```

Wenn es möglich ist, Algorithmen nichtrekursiv zu formulieren, wird man meist mit Zeit- oder Speicherplatzersparnis belohnt. Aber nicht alle rekursiven Funktionen lassen sich in eine nichtrekursive Form bringen!

Der Aufruf beider Varianten funktioniert nur, wenn `a >= b >= 0` gilt. In beiden Varianten muss man noch eine zusätzliche Funktion `int ggt(int a, int b)` schreiben, die vorher dafür sorgt, dass die Bedingungen an `a` und `b` (`a >= b >= 0`) zutreffen oder hergestellt werden, und die abschließend eine der obigen `ggt_???()` aufruft:

```
int ggt(int a, int b)
{
     if (a < 0)
          a = -a;              // jetzt ist sicher a >= 0
     if (b < 0)
          b = -b;              // jetzt ist sicher b >= 0
     if (a < b)                // vertausche a und b
          std::swap(a, b);     // jetzt gilt a >= b >= 0
     // jetzt gilt sicher  a >= b >= 0
     return ggt_recursiv(a,b);      // oder ggt_loop(a,b);
}
```

Wenn man die Schleifen-Variante zur Berechnung verwendet, kann man beide Funktionen leicht zu einer zusammenfassen:

```
int ggt(int a, int b)
{
     if (a < 0)
          a = -a;    // jetzt ist sicher a >= 0
     if (b < 0)
          b = -b;    // jetzt ist sicher b >= 0
     if (a < b)      // vertausche a und b
          std::swap(a, b);  // jetzt gilt a >= b >= 0
     for (int r; b; a = b, b = r)
          r = a % b;
     return a;
}
```

Den Euklidischen Algorithmus kann man auch auf alle anderen Ganzzahltypen anwenden. Deshalb wird er meist als Template realisiert.

Der **erweiterte Euklidische Algorithmus**:

Dieser berechnet nicht nur den ggt von *a, b* sondern auch noch 2 Faktoren *fa, fb mit*

*ggt = a * fa + b * fb*

Dieser Algorithmus wird stets rekursiv programmiert. Man braucht ihn vor allem in der Zahlentheorie und in deren wichtigster Anwendung: **Kryptographie**

Da hier 3 Werte zurückgegeben werden müssen aber nur ein Rückgabe-Objekt zulässig ist, muss man sich zwischen 2 Möglichkeiten entscheiden:

1) Man gibt einen Wert (den ggt) als Ergebnis zurück, die beiden Faktoren werden per Referenz-Parameter „zurückgegeben“ („Output-Parameter“ sind aber nicht gerne gesehen). Die Signatur wäre dann:
```
int  ggt_extended(int a, int b, int& fa, int& fb);
```
2) Man gibt ein `struct` zurück (oder etwas vergleichbares), in dem alle 3 Werte enthalten sind. Dies ist sicher die elegantere Lösung, z.B.:
```
struct ggt_tripel { int ggt, fa, fb; };
ggt_tripel  ggt_extended(int a, int b);
```

Wenn man diesen Algorithmus als Template schreiben will, muss man beachten, dass die Faktoren fa und fb auf jeden Fall `signed` sein müssen, auch wenn a, b `unsigned` sind.