Rekursive Folgen

Manchmal sind Folgen in der Mathematik rekursiv definiert, d.h. man erfährt nur, wie man aus alten Folgengliedern neue berechnen kann:

Fibonacci-Folge und ähnliche rekursive Folgen:

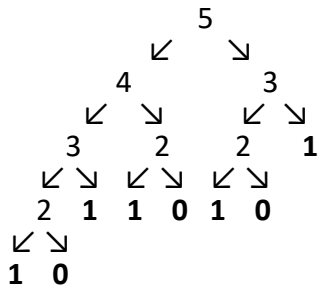
Die klassische Fibonacci-Folge ist festgelegt durch `F(0) = 0, F(1) = 1,`
`F(n) = F(n-1) + F(n-2)` für alle `n >= 2`

d.h. aus 2 Folgengliedern ergibt sich das nächste als deren Summe. Die Rekursionstiefe ist hier 2, weil man 2 alte Werte braucht, um einen neuen zu berechnen.

Am schnellsten programmiert man solche Folgen als rekursive Funktionen, was in C/C++ erlaubt ist (allerdings muss diese Variante nicht unbedingt die effizienteste sein!):

```
long long F(int n, long long a = 0, long long b = 1)
                                        // berechne Folgenglied mit Index n
{
     if (n >= 2)                        // der Hauptfall, n = 2, 3, 4, …
          return F(n-1)+F(n-2);         // die Rekursion
     return n ? b : a;                  // die Startwerte bei n = 0, n = 1
}
```

Trotz Optimierung braucht z.B. der Aufruf `F(50)` extrem lang. Der Grund dafür liegt in der Rekursionstiefe 2, d.h. der Aufwand steigt wie 2^n: In der folgenden Tabelle habe ich den „Aufruf-Graphen“ für `F(5)` gezeichnet. Die Pfeile bedeuten, dass hier durch die Rekursion weitere Aufrufe von `F()` gemacht werden, die fett gedruckten Argumente erzeugen keinen weiteren Aufruf, da es sich um die Anfangswerte der Rekursion handelt. Insgesamt erfolgen „nur“ 1+2+4+6+2 = 15 Funktionsaufrufe (statt 16)



`F(50)` benötigt also ungefähr 2^{50}= **`1125899906842624`** Funktionsaufrufe!

Obwohl die rekursive Funktion am schnellsten zu programmieren ist, erzeugt sie nicht immer das schnellste Programm (vor allem bei Rekursionstiefen > 1).

Was kann man tun, um eine zu langsame Rekursion zu beschleunigen?

1) Mathematische Verfahren:
   a) explizite Darstellung der Funktionswerte (das geht natürlich nicht immer!)
   b) Reduktion der Rekursionstiefe (das geht auch nicht immer)
2) Programmiertechnische Vereinfachungen:
   a) Umschreiben der Funktion in eine Schleife (das geht auch nicht immer).
   b) Man gibt der Funktion ein Gedächtnis, sodass sie schon berechnete Ergebnisse nicht jedes Mal neu evaluiert.

1a) Die Fibonacci-Folge hat eine explizite Darstellung, in der „Goldene Schnitt" vorkommt. Mit `g = (1+sqrt(5))/2` (das ist der goldene Schnitt) und `h = (1-sqrt(5))/2` kann man `F(n)` berechnen als

$$F(n) = (g^n - h^n)/\mathrm{sqrt}(5)$$

Die exakte Berechnung (nicht mittels Gleitkommarechnung wegen der Rundungsfehler) ist jedoch keineswegs trivial. Am besten definiert man Zahlen vom Typ `a + b*sqrt(5)` mit rationalen Koeffizienten als neuer Datentyp, z.B.

```
struct sqrt5 {
     Bruch a, b;     // entspricht der Zahl  a + b*sqrt(5)
};
```

Jetzt kann man die Grundrechnungsoperationen für diesen Datentyp analog zu den komplexen Zahlen definieren. Damit lässt sich die Berechnung von F(n) exakt durchführen, solange man im Bruch keinen Überlauf hat.

1b) Diese Methode ist immer dann möglich, wenn die Rekursionsformel nur von den alten Werten und nicht explizit vom Index `n` abhängt:

```
F(n) = F(n-1) + F(n-2)              // das geht
F(n) = F(n-1)  + n * F(n-2)         // das geht nicht
```

Der Trick besteht darin, die Anfangswerte in die Funktion aufzunehmen. Um z.B. das **9**. Glied `F(8)` der Fibonacci-Folge zu berechnen, schreibt man die Folge an:

Anfangswerte `0, 1:` `0, 1, 1, 2, 3, 5, 8, 13, 21 = F(8)`

Anfangswerte **`1, 1:`** **`1, 1,`** `2, 3, 5, 8, 13, 21 = F(8)`

Man sieht, dass `F(8)` das **8**. Glied einer Fibonacci-Folge mit Anfangswerten 1, 1 ist. Man definiert eine neue Fibonacci-Funktion `F(n, a, b)` mit Anfangswerten `a,b`:

Anfangswerte `a, b:` `a,` **`b, a+b,`** `a+2b, 2a+3b, …`

Anfangswert **`b, a+b:`** **`b, a+b,`** `a+2b, 2a+3b, …`

Man sieht: `F(n, a, b) = F(n-1, b, a+b)`
Diese neue rekursive Funktion `F` hat eine einstufige Rekursion und erzeugt damit keinen exponentiellen Aufwand mehr!

```
int F(int n, long long a = 0, long long b = 1)
                                        // berechne Folgenglied mit Index n
{                                       // und Anfangswerten  a, b
     if (n >= 2)                        // der Hauptfall, n = 2, 3, 4, …
          return F(n-1, b, a+b);        // die Rekursion
     return n ? b : a;                  // die Startwerte bei n = 0, n = 1
}
```

2b) Das Umschreiben in eine Schleife funktioniert bei der Fibonacci-Folge gut, da man ausgehend von den Anfangswerten jeden Index leicht erreichen kann. Man muss zur Berechnung von `F(n)` alle Folgenglieder bis zum Index n berechnen, allerdings geht das viel schneller als die zweistufige Rekursion:

```
long long F(int n, long long a = 0, long long b = 1)
{    if (n < 2)
          n = ? b : a;

     std::array kreis{a, b, b};   // Index modulo 3 geht im Kreis
     for (int k = 1; k < n; ++k)
          kreis[(k+1)%3] = kreis[(k-1)%3] + kreis[k%3];
     return kreis[n%3];
}
```

2b) Wir brauchen dazu einen Datentyp, der berechnete Folgenglieder speichert und somit das Gedächtnis der Funktion darstellt. Da man sehr schnell nachsehen muss, ob ein Folgenglied gespeichert ist, verwendet man dazu meistens einen assoziativen Container `c` (in Python ein dictionary, in C++ ein `std::map`). Diese Container speichern Paare der Form *`(key, value)`*. Man kann in beiden Sprachen schnell abfragen, ob ein `key` schon gespeichert ist und welchen Wert `value` er hat. In unserem Fall wollen wir zu jedem Index `n` den Wert `F(n)` speichern , d.h. der Gedächtnistyp ist
`std::map<int, long long>`.

```
long long F_hidden(int n, std::map<int, long long>& memo)
{    if (auto it = memo.find(n); it != memo.end())
               // found key = n
          return it->second;        // return value

                    // Rekursion, wenn Wert nicht in memo ist
     long long f = F_hidden(n-1, memo) + F_hidden(n-2, memo);
```

```
    memo[n] = f;    // wir merken uns den berechneten Wert
    return f;       // und geben ihn zurück
}
```

Die Funktion `F_hidden()` macht die ganze Arbeit, allerdings ist sie nicht für einen direkten Aufruf gedacht, da man hier schon das Gedächtnis übergeben muss. Deshalb schreiben wir noch die „echte“ Funktion `F(n, a, b)` (Fibonaccifolge mit Anfangswerten), die das Gedächtnis erzeugt und damit die versteckte Funktion aufruft. In anderen Sprachen (Python, Javascript) kommt man ohne die versteckte Funktion aus, in C++ leider nicht!

```
long long F(int n, long long a, long long b)
{   std::map<int, long long> memo{{0, a}, {1, b}};
    return F_hidden(n, memo);
}
```

Bei dieser Konstruktion werden bei jedem Aufruf von `F()` alle Folgenwerte neu berechnet. Man kann das Gedächtnis speichern und braucht daher nicht jedes Mal bereits bekannte Werte neu berechnen. Dazu kreiert man am besten einen neuen Typ, etwa `Fibonacci`:

```
struct Fibonacci {
    mutable std::map<int, long long> memo;  // Gedächtnis

    explicit Fibonacci(long long a = 0, long long b = 1)
    : memo{{0, a}, {1, b}}   {}             // Konstruktor

    // das folgende entspricht  F_hidden()
    long long operator()(int n) const       // Aufruf-Operator
    {   if (auto it = memo.find(n); it != memo.end())
            return it->second;
        long long f = operator()(n-1) + operator()(n-2);
        memo[n] = f;
        return f;
    }
};
```

im Programm:

```
Fibonacci F01;              // klassische Folge mit Startwerten  0,1
cout << F01(10);            // Element 11 ausgeben, verwende F01 als Funktion

Fibonacci F12{1LL, 2LL}; // Folge mit Startwerten  1, 2
cout << F12(10);            // Element 11 ausgeben
```