Fibonacci Zahlen

Empirische Untersuchung der Aufrufe

Idee

Um einen Überblick über die Rekursive Fibonacci Funktion zu erhalten könnte eine Untersuchung der Knotenpunkte Aufschluss über die Anzahl der Knoten für ein beliebiges n geben.

Laut Skript ist die Fibonacci Folge wie folgt Rekursiv darzustellen

$f(n)=(f(n-1)+f(n-2))$
$f(0)=1$
$f(1)=1$

Eine Möglichkeit zur Implementierung in C

```
int fib(int n)
{

	if(n==0 || n==1)
	{
		return n;
	}

	return fib(n-1)+fib(n-2);

}
```

Es müssen nun die Aufrufe gezählt werden, dass man eine Übersicht über die Anzahl der Knotenpunkte erhält. Denn der Aufwand der Funktion ist Anzahl Knoten * Rechenzeit pro Knoten (c).

Programm

```
#include <stdio.h>
#include <math.h>

int fib_count=0; /* counts the function calls */

/**
*	trial implementation of long long pow function
*	long long not being displayed...
*/
long long pow_long(int i,int j)
{
	int k;
	long long res=i;
	for(k=1;k<j;k++)
	{
		res=res*i;
	}
	return res;
}
```

```
/**
*       Calculates the Fibonacci Number...
*/
int fib(int n)
{

        fib_count++;

        if(n==0 || n==1)
        {
                return 1;
        }

        return fib(n-1)+fib(n-2);

}

int main (void)
{
        int res=0;
        int i,max=30;
        long long kmax=0;
        for(i=0;i<=max;i++)
        {
                kmax=(long long)((long long)pow_long(2,i)-1);
                fib_count=0;
                res=fib(i);
                printf("Elemente %d\t Aufrufe %d\t Ergebnis %d\t Max_Knoten %ld\t
                        n",i,fib_count,res,kmax);
                //printf("%d;%d;%ld;\n",i,fib_count,kmax);
        }
        return;
}
```

Es wird die Funktion mit den Zahlen 0 – 30 Aufgerufen und die Ergebnisse werden ausgegeben.

Problem: Ab n=32 überschreitet (2^n)-1 den Integer/Long Wertebereich. Auch die Implementierung einer Pow Funktion auf Long Long Basis brachte noch keine Lösung.

Ergebnis

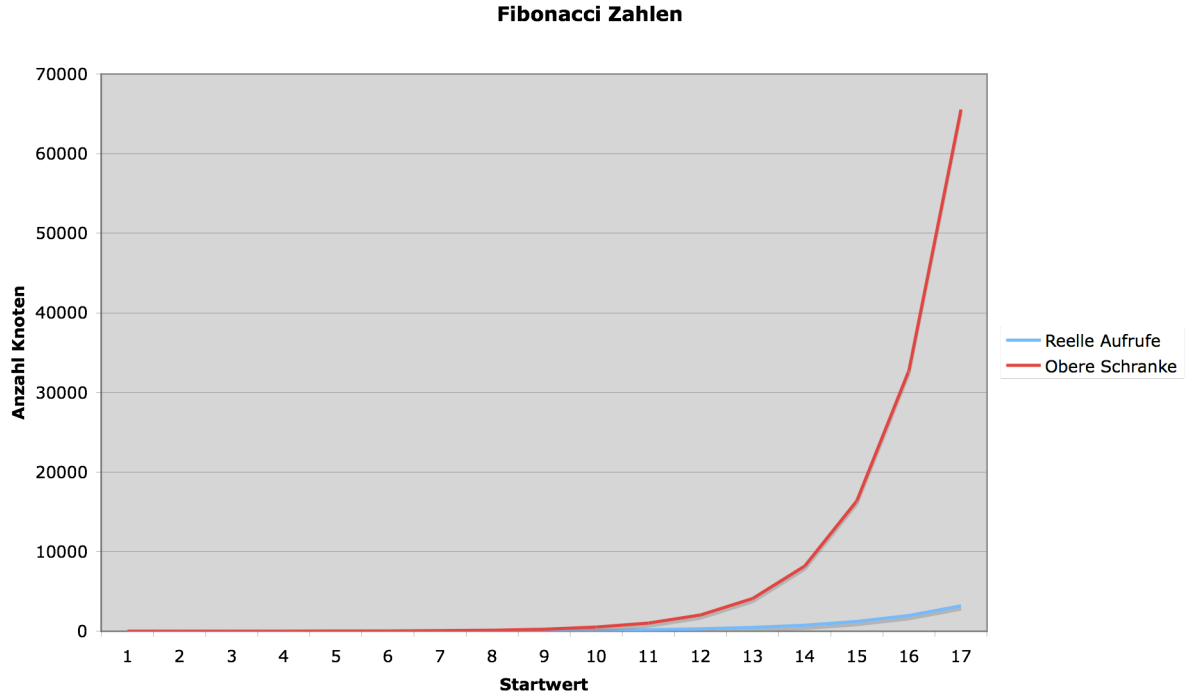
Das Ausführen des Programms bringt folgendes Ergebnis.

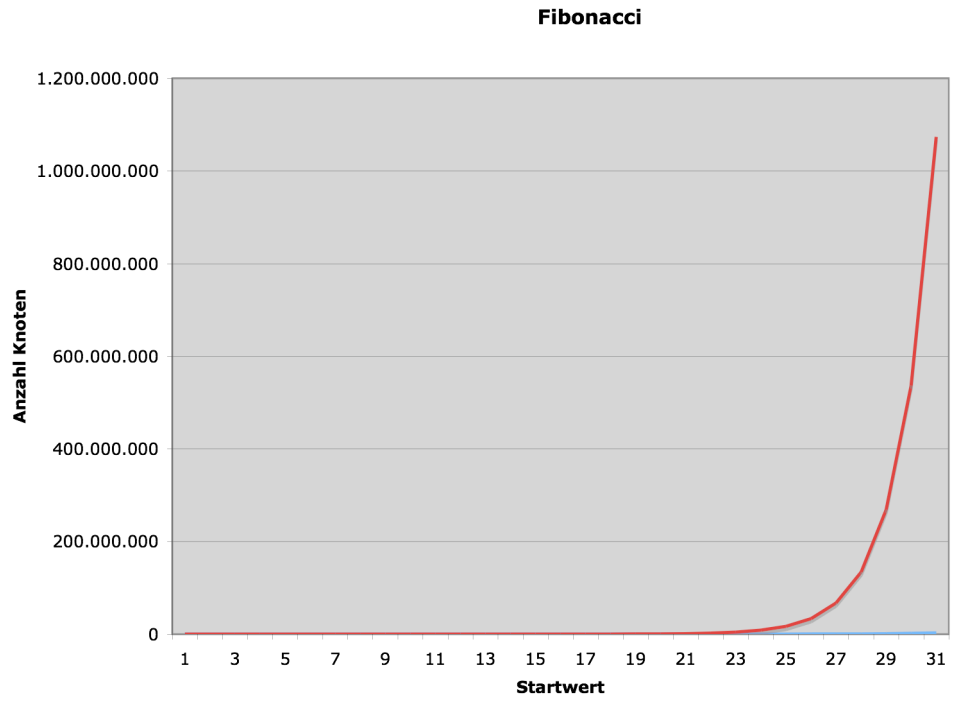
n	Reell	O
0	1	1
1	1	1
2	3	3
3	5	7
4	9	15
5	15	31
6	25	63
7	41	127
8	67	255
9	109	511
10	177	1.023
11	287	2.047
12	465	4.095
13	753	8.191
14	1.219	16.383
15	1.973	32.767
16	3.193	65.535
17	5.167	131.071
18	8.361	262.143
19	13.529	524.287
20	21.891	1.048.575
21	35.421	2.097.151
22	57.313	4.194.303
23	92.735	8.388.607
24	150.049	16.777.215
25	242.785	33.554.431
26	392.835	67.108.863
27	635.621	134.217.727
28	1.028.457	268.435.455
29	1.664.079	536.870.911
30	2.692.537	1.073.741.823

→ Ab n=5 ist die obere Schranke <= n/2

Graphische Auswertung

Zur Veranschaulichung des Wachstums erstmal bis n=17.





Berechnung des Wachstums

1. Versuch – Lineares Gleichungssystem

Anhand der oben aufgestellten Tabelle wurde ein LGS aufgestellt. Allerdings lies sich dafür keine eindeutige Lösung berechnen. Dies bedeutet, dass die Funktion nicht kubisch wächst.

2. Versuch – Betrachtung des exponentiellen Wachstums

Als nächstes wurde das exponentielle Wachstum betrachtet. Dazu wurden einige Werte zufällig verglichen.

25/15=1,666666667
5167/3193=1,61822737
150049/92735=1,618040653
635621/392835=1,618035562
2692537/1664079=1,618034

Es zeigt sich also, dass mit steigenden n-Werten bei der Division sich ein Wert von ~1,618 ergibt.

Dies wurde auch von dem Mathematiker Johannes Kepler festgestellt. Dieses Ergebnis bedeutet, dass sich der Quotient für große n-Werte dem goldenen Schnitt annähert.

Der goldene Schnitt berechnet sich anhand folgender Formel

$$y=\frac{\left(\sqrt[2]{5}+1\right)}{2}$$

Nun wollen wir aber noch eine exakte Formel berechnen.

$$y=a\cdot\frac{\left(\sqrt[2]{5}+1\right)^{n}}{2}$$

$$a=\frac{(resultn)}{\frac{\left(\sqrt[2]{5}+1\right)^{n}}{2}}$$

Für n=30 ist a=1,44722
Für n=20 ist a= 1,447147

Also ist a ~ 1,447 und dies entspricht ungefähr

$$a=\frac{1}{\frac{2}{\sqrt{5}}}+1$$

Also berechnet sich die Anzahl der Knoten für große n-Werte näherungsweise wie folgt:

$$y=\left(\frac{1}{\frac{2}{\sqrt{5}}}+1\right)\cdot\frac{\left(\frac{2}{\sqrt{5}}+1\right)^{n}}{2}$$

Kontrolle der Formel

Nun ist es interessant, die Abweichung der Formel für möglichst große n Werte zu bestimmen.

Dazu habe ich das C Programm angepasst an die errechnete Formel, das berechnet nun die reellen Aufrufe, die errechnete Näherung und die Ursprungsformel (2^n)-1.

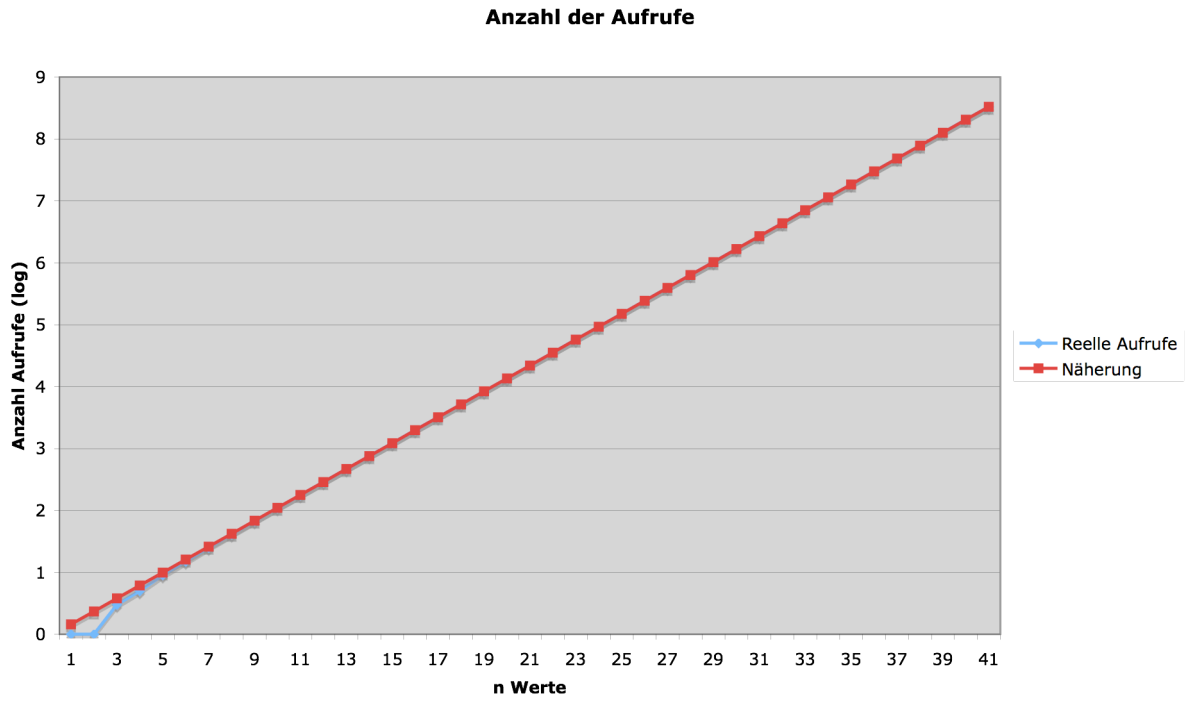
Dafür wurden folgende 2 Funktionen implementiert

```
/**
*
*       Approached function calls
*
*/
float function_approach(int i)
{
        float res;
        res=((1/sqrt(5))+1)*(pow((((sqrt(5)+1))/2),i));
        return res;
}

/**
*
*       Difference between real Calls and Approched ones
*
*/
double approach_difference(int nodes,float approach)
{
        return (double)1-(double)(approach/nodes);
}
```

Allerdings erwies sich C als zu ungenau, daher wurde für die Berechnung der Abweichung Excel verwendet.

n	Reell	Näherung	Abweichung (%)
0	1	1,45	-4,47E-01
1	1	2,34	-1,34E+00
2	3	3,79	-2,63E-01
3	5	6,13	-2,26E-01
4	9	9,92	-1,02E-01
5	15	16,05	-7,00E-02
6	25	25,97	-3,88E-02
7	41	42,02	-2,49E-02
8	67	67,99	-1,47E-02
9	109	110,01	-9,24E-03
10	177	178,00	-5,62E-03
11	287	288,00	-3,49E-03
12	465	466,00	-2,15E-03
13	753	754,00	-1,33E-03
14	1.219	1.220,00	-8,20E-04
15	1.973	1.974,00	-5,07E-04
16	3.193	3.194,00	-3,13E-04
17	5.167	5.168,00	-1,94E-04
18	8.361	8.362,00	-1,20E-04
19	13.529	13.530,00	-7,39E-05
20	21.891	21.892,00	-4,57E-05
21	35.421	35.422,00	-2,82E-05
22	57.313	57.314,00	-1,74E-05
23	92.735	92.736,00	-1,08E-05
24	150.049	150.050,00	-6,66E-06
25	242.785	242.786,00	-4,12E-06
26	392.835	392.836,00	-2,55E-06
27	635.621	635.622,00	-1,57E-06
28	1.028.457	1.028.458,00	-9,72E-07
29	1.664.079	1.664.080,00	-6,01E-07
30	2.692.537	2.692.538,00	-3,71E-07



Dies zeigt dass die Näherung für die Anzahl der Aufrufe sich der Anzahl der Reellen Aufrufe annähert. Dies muss aber noch für sehr hohe n-Werte bewiesen werden.

Auswertung

Das Ergebnis zeigt, dass für die Berechnung sehr viele Rechenschritte nötig sind. Beim Berechnen der Fibonacci Folge der Zahl 30 sind 2.692.537 Funktionsaufrufe zu bearbeiten.

Auch wenn die Obergrenze weit unterschritten wird, benötigt die Berechnung aufgrund der Komplexität von n^2 sehr viel Rechenzeit.

Daher liegt eine Optimierung des Algorithmus nahe.

Optimierung

```
int opt_fib(int n)
{
        int fib_arr[n+1];
        int i;
        fib_arr[0]=1;
        fib_arr[1]=1;
        for(i=2;i<=n;i++)
        {
                fib_arr[i]=fib_arr[i-1]+fib_arr[i-2];

        }
        return fib_arr[n];
}
```

So kann die Fibonacci Zahl um ein vielfaches schneller berechnet werden, da einige Rechenschritte gespart werden.

Dies kann auch wieder in einer rekursiven Funktion geschrieben werden, mit n+1 Aufrufen

```
int opt_fib_rec(int i,int n,int numbers[])
{
        if(i<=n)
        {
                numbers[i]=numbers[i-1]+numbers[i-2];
                opt_fib_rec(i+1,n,numbers);
        } else {
                return;
        }
}

int opt_fib(int n)
{
        int fib_arr[n+1];
        fib_arr[0]=1;
        fib_arr[1]=1;
        opt_fib_rec(2,n,fib_arr);
        return fib_arr[n];
}
```

→ Die Funktion ist optimiert signifikant schneller, da die optimierte Funktion eine Komplexität von n hat. Das heißt die Rechenzeit der Funktion wächst linear.