The Fibonacci Numbers

COMS10017 - (Object-Oriented Programming and) Algorithms

Dr Christian Konrad

The Fibonacci Numbers

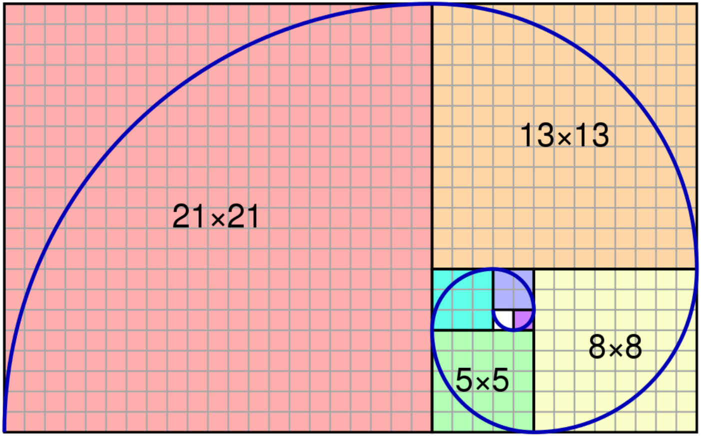
Fibonacci Numbers

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ for } n \geq 2 \,. \end{aligned}$$

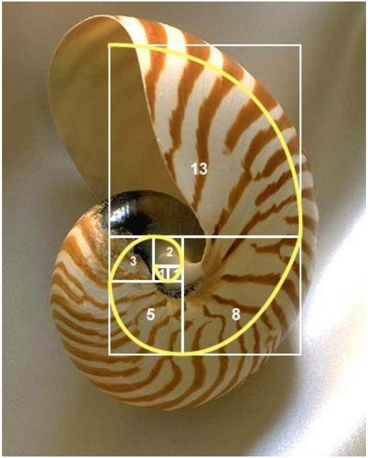
$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad \ldots$$

Why are they important?

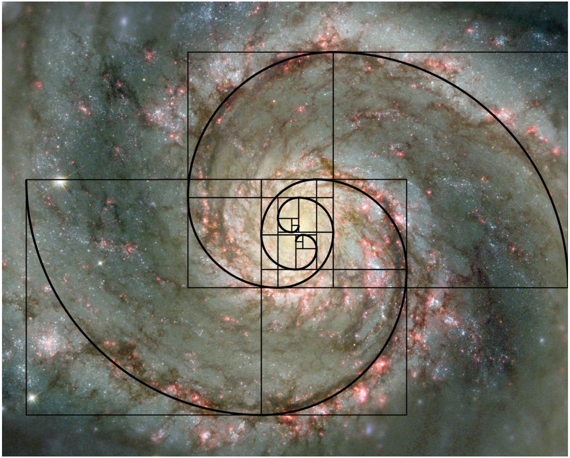
- Fibonacci heaps (data structure)
- Appear in analysis of algorithms (e.g. Euclid's algorithm)
- Appear everywhere in nature
- Interesting and instructive computational problem



source: wikipedia



source: realworldmathematics at wordpress



source: brian koberlein

Computing the Fibonacci Numbers

Naïve Algorithm

Require: Integer $n \geq 0$
  if $n \leq 1$ **then**
    return n
  else
    return $\textsc{Fib}(n-1) + \textsc{Fib}(n-2)$

$\textsc{Fib}(n)$

What is the runtime of this algorithm?

Runtime:

- Without recursive calls, runtime is $O(1)$
- Hence, runtime is $O($"number of recursive calls"$)$

Runtime Analysis

Define Recurrence:

$T(n)$: number of recursive calls to $\textsc{Fib}$ when called with parameter n

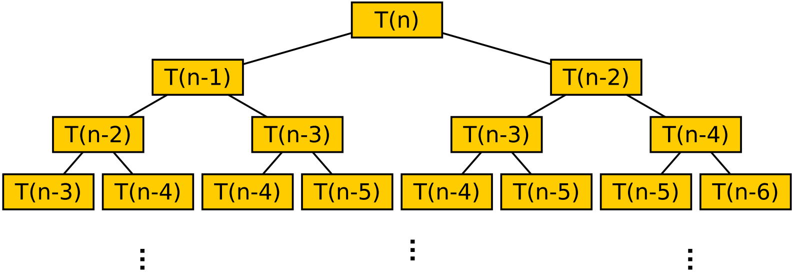
```
if n ≤ 1 then
   return n
else
   return FIB(n−1) + FIB(n−2)
```

$$T(0) = T(1) = 1$$
$$T(n) = 1 + T(n-1) + T(n-2)\ , \text{ for } n \geq 2 \ .$$

How to Solve this Recurrence?

- We will use the recursion tree technique to obtain a guess for an upper bound
- We will verify the guess with the substitution method

Recursion Tree for T



Observe:

- Each node contributes 1
- Hence, $T(n)$ equals number of nodes
- Number of levels of recursion tree: n
- Our guess: $T(n) \leq c^n$ (we believe $c \leq 2$)

Verification with the Substitution Method

Recall:

$$\begin{aligned} T(0) &= T(1) = 1 \\ T(n) &= 1 + T(n-1) + T(n-2) \text{ , for } n \geq 2 \quad . \end{aligned}$$

Our guess: $T(n) \leq c^n$

Substitute Guess into Recurrence:

$$T(n) = 1 + T(n-1) + T(n-2) \leq 1 + c^{n-1} + c^{n-2}$$

- It is required that $1 + c^{n-1} + c^{n-2} \leq c^n$
- The additive 1 prevents us from getting a similar form as c^n
- Try different guess: $T(n) \leq c^n - 1$

Verification with the Substitution Method (2)

New Guess: $T(n) \le c^n - 1$

$$\begin{aligned} T(n) &= 1 + T(n-1) + T(n-2) \\ &\le 1 + (c^{n-1} - 1) + (c^{n-2} - 1) = c^{n-1} + c^{n-2} - 1 \ . \end{aligned}$$

Select smallest possible c:

$$\begin{aligned} c^{n-1} + c^{n-2} &= c^n \\ 0 &= c^2 - c - 1 \\ c &= \frac{1+\sqrt{5}}{2} \approx 1.618033989 \ . \text{ Golden Ratio!} \end{aligned}$$

Base Case:

- $T(0) = T(1) = 1$
- $c^0 - 1 = 0$ and $c^1 - 1 \approx 0.61$ ✗

Verification with the Substitution Method (3)

Another New Guess: $T(n) \leq k \cdot c^n - 1$

$$\begin{aligned} T(n) &= 1 + T(n-1) + T(n-2) \\ &\leq 1 + (k \cdot c^{n-1} - 1) + (k \cdot c^{n-2} - 1) \\ &= k\left(c^{n-1} + c^{n-2}\right) - 1 \ . \end{aligned}$$

Select smallest possible c: $c = \frac{1+\sqrt{5}}{2}$ as before

Base Case:

- $T(0) = T(1) = 1$
- $k \cdot c^0 - 1 = k - 1$ and $k \cdot c^1 - 1 > k - 1$ ✓
- We can hence select $k = 2$!

We proved $T(n) \leq 2 \cdot (\frac{1+\sqrt{5}}{2})^n - 1$. Hence $T(n) \in O\left((\frac{1+\sqrt{5}}{2})^n\right)$.

Fibonacci Numbers: Closed-form Expression

Golden Ratio:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

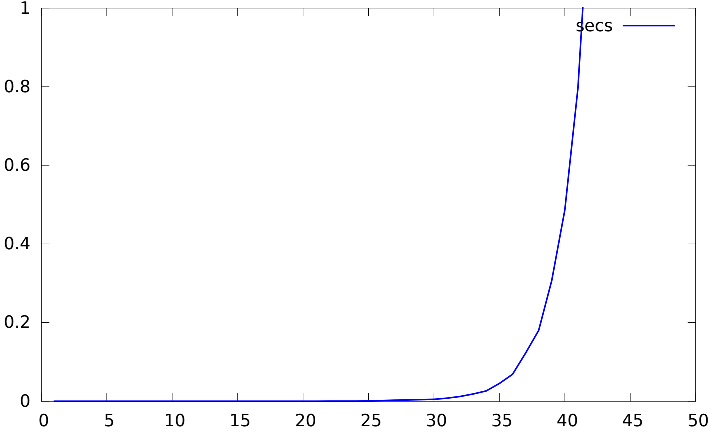
Closed-form Expression:

$$F_n \quad = \quad \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} \; .$$

Why not compute Fibonacci Numbers this way?

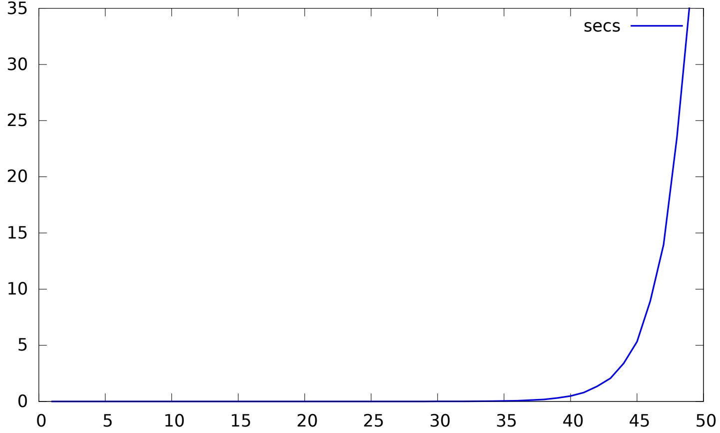
- Floating point operations, precision
- Large numbers involved
- Impractical

Experiments



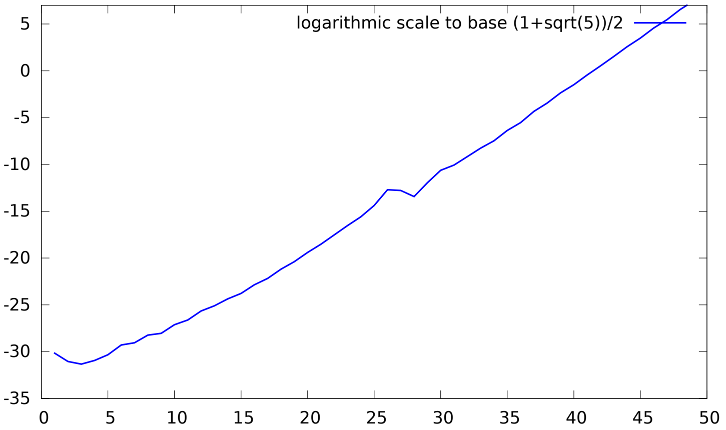
Exponential growth

Experiments



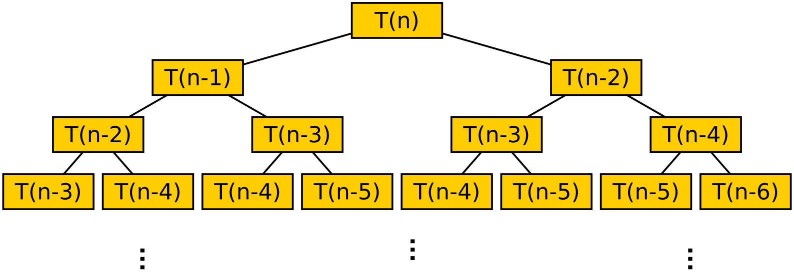
Exponential growth

Experiments



Logarithmic scale

Why is this Algorithm so slow?



Discussion:

- We compute solutions to subproblems many times ($T(i)$ is computed often, for most values of i)
- How can we avoid this?

Dynamic Programming!

Dynamic Programming Solution

Dynamic Programming (will be discussed in more detail later)

- Store solutions to subproblems in a table
- Compute table bottom up

Require: Integer $n \geq 0$
    if $n \leq 1$ **then**
        return n
    else
        $A \leftarrow$ array of size n
        $A[0] \leftarrow 1$, $A[1] \leftarrow 1$
        for $i \leftarrow 2 \dots n$ **do**
            $A[i] \leftarrow A[i-2] + A[i-1]$
        return $A[n]$

$\textsc{DynPrgFib}(n)$

Dicussion

Analysis:

- DynPrgFib() runs in time $O(n)$
- It uses space $\Theta(n)$ since it uses an array of size n

Can we reduce the space to $O(1)$?

Improvement:

- Observe that when $T(i)$ is computed, the values $T(1), T(2), \ldots, T(i-3)$ are no longer needed
- Only store the last two values of T

Improved Algorithm

Require: Integer $n \geq 0$
  if $n \leq 1$ **then**
    return n
  else
    $a \leftarrow 0$
    $b \leftarrow 1$
    for $i \leftarrow 2 \ldots n$ **do**
      $c \leftarrow a + b$
      $a \leftarrow b$
      $b \leftarrow c$
    return c

$\textsc{ImprovedDynPrgFib}(n)$

Correctness: via loop invariant!