Lecture 15: The Fibonacci Numbers

COMS10007 - Algorithms

Dr. Christian Konrad

25.03.2019

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 1 1 2 3 5 8 13 21 34 55 89 ...

Why are they important?

- Fibonacci heaps (data structure)
- Pseudo-random generators
- Appear in analysis of algorithms (e.g. Euclid's algorithm)
- Interesting computational problem

Computing the Fibonacci Numbers

Naïve Algorithm

```
Require: Integer n ≥ 0
  if n ≤ 1 then
    return n
  else
    return FIB(n − 1) + 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:

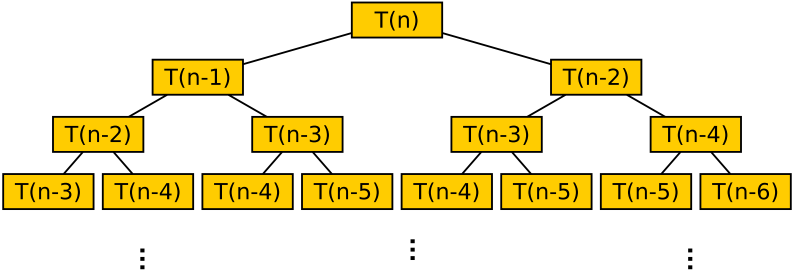
$T(n)$: number of recursive calls to $\textsc{Fib}$ when called with parameter n

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

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

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



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: $T(n) \leq c^n - 1$

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

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

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

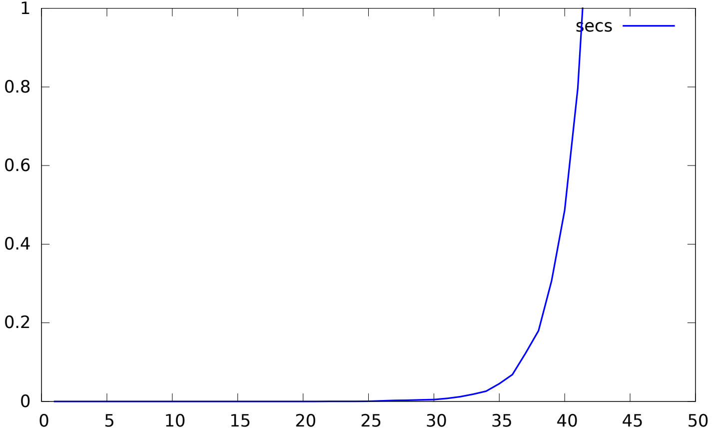
Select smallest possible c: $c = \frac{1+\sqrt{5}}{2}$ as before

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 $k = 2$!

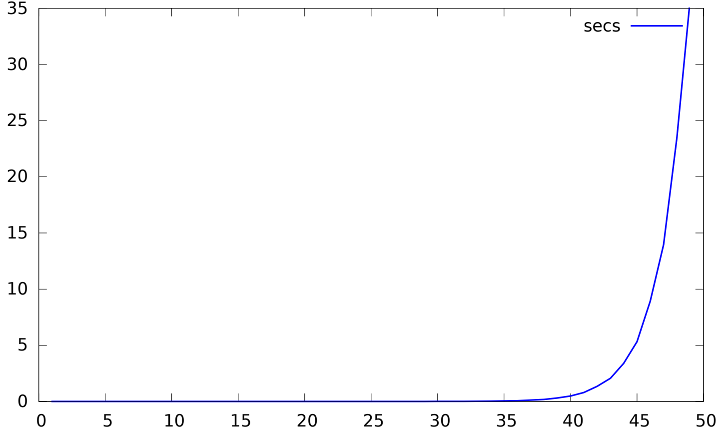
We proved $T(n) \leq 2 \cdot (\frac{1+\sqrt{5}}{2})^n - 1$. Hence $T(n) \in O\left((\frac{1+\sqrt{5}}{2})^n\right)$.

Experiments



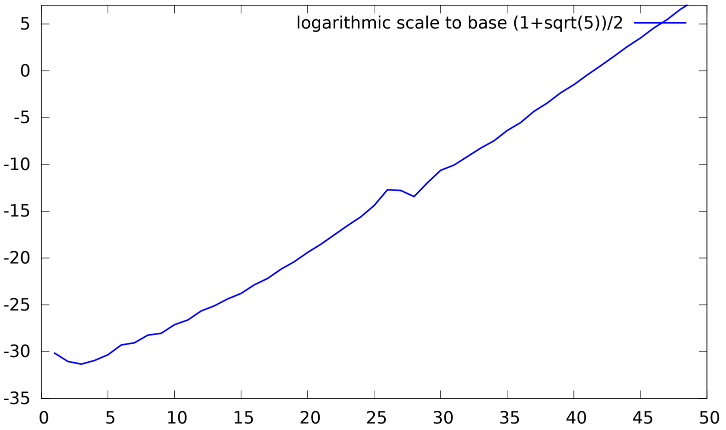
Exponential growth

Experiments



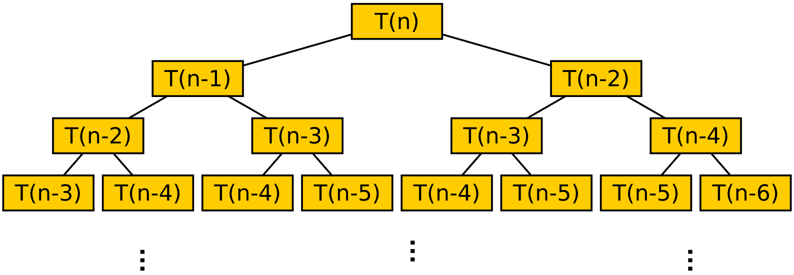
Exponential growth

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 ≥ 0
  if n ≤ 1 then
    return n
  else
    A ← array of size n
    A[0] ← 1, A[1] ← 1
    for i ← 2 ... n do
      A[i] ← 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 ≥ 0
  if n ≤ 1 then
    return n
  else
    a ← 0
    b ← 1
    for i ← 2 ... n do
      c ← a + b
      a ← b
      b ← c
    return c
```

ImprovedDynPrgFib(n)

Correctness: via loop invariant! (on exercise sheet 3)