Fundamental Algorithms

Chapter 1: Introduction

Michael Bader

Winter 2011/12

Part I

Overview

Organizational Stuff

- 2 SWS / 3 credits
- Master CSE $\rightarrow$ compulsory
  Master BiomedComp $\rightarrow$ elective
- Lecture only
- But practice necessary (as usual)
  - Offer of tutorial sheets
  - Maybe review of one exercise at beginning of next lecture
- Slides, tutorial sheets and announcements on website

Contents

- Introduction of “fundamental” algorithms and their analysis
- Aim: get common basis for other lectures

Topics

- Fundamentals (Analysis, Complexity Measures)
- Basic discipline: sorting
- (Selecting)
- Searching (hashing, search trees, . . .)
- Arithmetic problems (e.g. parallel matrix and vector operations)
- Graph problems

Part II

Algorithms

What is an Algorithm? – Some Definitions

Definition (found on numerous websites)

An algorithm is a set of rules that specify the order and kind of arithmetic operations that are used on a specified set of data.

Definition (Wikipedia)

An algorithm is an effective method for solving a problem using a finite sequence of instructions.

Definition (Donald Knuth)

An algorithm is a finite, definite, effective procedure, with some output.

Definition (Britannica.com)

Systematic procedure that produces – in a finite number of steps – the answer to a question or the solution of a problem.

Example Algorithm: Chocolate Chip Cookies

Ingredients

- 1 cup butter, softened
- 1 cup white sugar
- 1 cup packed brown sugar
- 2 eggs
- 2 teaspoons vanilla extract
- 3 cups all-purpose flour
- 1 teaspoon baking soda
- 2 teaspoons hot water
- 1/2 teaspoon salt
- 2 cups semisweet chocolate chips
- 1 cup chopped walnuts

Directions

1. Preheat oven to 350 degrees F (175 degrees C).
2. Cream together the butter, white sugar, and brown sugar until smooth. Beat in the eggs one at a time, then stir in the vanilla. Dissolve baking soda in hot water. Add to batter along with salt. Stir in flour, chocolate chips, and nuts. Drop by large spoonfuls onto ungreased pans.
3. Bake for about 10 minutes in the preheated oven, or until edges are nicely browned.

Essential properties of an algorithm

- an algorithm is **finite**
  (w.r.t.: set of instructions, use of resources, time of computation)
- instructions are **precise** and **computable**
- instructions have a specified logical order, however, we can discriminate between
  - **deterministic** algorithms
    (every step has a well-defined successor)
  - **non-deterministic** algorithms
    (randomized algorithms, but also parallel algorithms!)
- produce a **result**

Basic Questions About Algorithms

For each algorithm, we should answer the following basic questions:

- does it terminate?
- is it correct?
- is the result of the algorithm determined?
- how much resources will it use in terms of
    - memory? (and memory bandwidth?)
    - operations?
    - run-time?
    - . . . ?

Example: Fibonacci Numbers

Definition

The sequence f_j, $j \in \mathbb{N}$, of the Fibonacci numbers is defined recursively as:

$$\begin{aligned} f_0 &:= 1 \\ f_1 &:= 1 \\ f_j &:= f_{j-1} + f_{j-2} \qquad \text{for} \quad j \geq 2 \end{aligned}$$

Origin: simple model of a rabbit population

- starts with one pair of rabbits (male and female)
- every month, each pair of rabbits gives birth to a new pair
- but: new-born rabbits need one month to become mature

(compare lecture in Scientific Computing)

A Recursive Algorithm for the Fibonacci Numbers

```
Fibo(n:Integer) : Integer {
    if n=0 then return 1;
    if n=1 then return 1;
    if n>1 then return Fibo(n-1) + Fibo(n-2);
}
```

→ How many arithmetic operations does it take to compute f_j?

Definition

$T_{\mathrm{Fibo}}(n)$ shall be the number of arithmetic operations (here: additions) that the algorithm Fibo will perform with n as input parameter.

Number of Additions by Fibo

We observe that:

- $T_{\text{Fibo}}(0) = T_{\text{Fibo}}(1) = 0$
  (both cases do not require any additions)

If the parameter n is larger than 1, then we have to:

- perform all additions of calling Fibo(n−1) and Fibo(n−2)
- and add the two results
- thus:

$$T_{\text{Fibo}}(n) = T_{\text{Fibo}}(n-1) + T_{\text{Fibo}}(n-2) + 1$$

  No $\rightarrow$ better:

$$T_{\text{Fibo}}(n) = T_{\text{Fibo}}(n-1) + T_{\text{Fibo}}(n-2) + 3$$

- because: we forgot to compute $n-1$ and $n-2$

We obtain a so-called **recurrence equation**

Number of Additions by Fibo (2)

Solving the recurrence: (in this example)

- first observation: recurrence looks a lot like Fibonacci recurrence, itself
- draw a table of n vs. additions
  $\rightarrow$ assumption: $T_{\text{Fibo}}(n) = 3f_n - 3$
- Proof: by induction over n

Estimate of the number of operations:

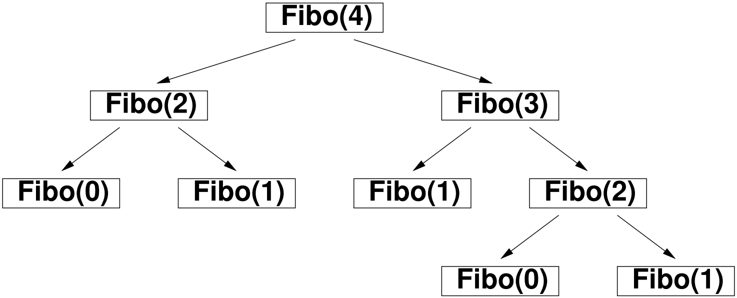
- algebraic formulation of the Fibonacci numbers:

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

- **exponential** growth of number of operations
- example: $T_{\text{Fibo}}(100) \approx 10^{21}$
  (requires more than 30,000 years, if we process one addition per nanosecond)

Why is Fibo so Slow?

Examine recursive calls:



→ Obviously, lots of numbers f_j are computed multiple times!

An Iterative Algorithm for the Fibonacci Numbers

```
FibIt(n : Integer) : Integer {
  if n < 2 then return 1;
  else {
    last2 := 1;
    last1 := 1:
    for i from 2 to n do {
      f := last2 + last1;
      last2 := last1;
      last1 := f;
    }
    return f;
  }
}
```

Idea:

- keep the last two values f_{i-2} and f_{i-1} in last2 and last1

Is This Correct?

Only loop critical

- Basic idea: use so-called **loop invariant** to prove properties about loop
- Statement of conditions that are valid for each loop execution
- Here, e.g.
  - before the loop body is executed:
    last1 and last2 contain f_{i-1} and f_{i-2}, respectively

For loop invariants, we need to prove:

Initialization: It is true prior to first execution of loop (body)

Maintenance: If it is true before iteration of loop, it remains true before next iteration

Termination: When loop terminates, invariant gives us useful property, helping to prove correctness

(Note: compare scheme of proof by induction)

Correctness

Invariant

$$\{\text{last1} = f_{i-1}; \text{last2} = f_{i-2}\}$$

Initialization

Before first iteration of loop, we have

- $i = 2$
- $\text{last1} = 1 = f_1$
- $\text{last2} = 1 = f_0$

Correctness (2)

Maintenance: Proof of invariant:
Consider function body

$$\{\text{last1} = f_{i-1}; \text{last2} = f_{i-2}\}$$

```
f := last2 + last1;
```

$$\{\text{last1} = f_{i-1}; \text{last2} = f_{i-2}; f = f_i\}$$

```
last2 := last1;
```

$$\{\text{last1} = f_{i-1}; \text{last2} = f_{i-1}; f = f_i\}$$

```
last1 := f;
```

$$\{\text{last1} = f_i; \text{last2} = f_{i-1}; f = f_i\}$$

At end of (before beginning of next) loop iteration, we have implicitly

```
i := i + 1;
```

$$\{\text{last1} = f_{i-1}; \text{last2} = f_{i-2}\}$$

thus, invariant still holds at next loop entry

Correctness (3)

Termination

- At loop termination, i exceeds n; thus $i = n + 1$
  (Note: think in while-loops where increment is done explicitly)
- If loop invariant holds, then last1 and last2 contain $f_{i-1} = f_n$ and $f_{i-2} = f_{n-1}$, respectively
- f equals last1, hence f_n

q.e.d.

Does FibIt Require Fewer Operations?

We observe that:

- $T_{\text{FibIt}}(1) = T_{\text{Fibo}}(1) = 0$
  (no additions, if input parameter n < 2)
- If $n \geq 2$:
  - the for loop will be executed $n - 1$ times
  - in the loop body, there is always exactly one addition per loop iteration

Therefore:

$$T_{\text{FibIt}}(n) = \begin{cases} 0 & \text{for} \quad n \leq 1 \\ n-1 & \text{for} \quad n \geq 2 \end{cases}$$

$\rightarrow$ the operation count of FibIt increases **linearly** with n.

Question: will f_{10^9} be computed in 1 second?

Part IV

Asymptotic Behaviour of Functions

Asymptotic Behaviour of Functions

Definition (Asymptotic upper bound)

g is called an asymptotic upper bound of f, or $f \in O(g)$, if

$$\exists c > 0 \, \exists n_0 \, \forall n \geq n_0 \colon \, f(n) \leq c \cdot g(n)$$

Definition (Asymptotic lower bound)

g is called an asymptotic lower bound of f, or $f \in \Omega(g)$, if

$$\exists c > 0 \, \exists n_0 \, \forall n \geq n_0 \colon \, f(n) \geq c \cdot g(n)$$

Definition (Asymptotically tight bound)

g is called an asymptotically tight bound of f, or $f \in \Theta(g)$, if

$$f \in O(g) \qquad \text{and} \qquad f \in \Omega(g)$$

Asymptotic Behaviour of Functions (2)

Definition (Asymptotically smaller)

f is called asymptotically smaller than g, or $f \in o(g)$, if

$$\forall c > 0 \, \exists n_0 \, \forall n \geq n_0 : \, f(n) \leq c \cdot g(n) \quad \Leftrightarrow \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Definition (Asymptotically larger)

f is called asymptotically larger than g, or $f \in \omega(g)$, if

$$\forall c > 0 \, \exists n_0 \, \forall n \geq n_0 : \, f(n) \geq c \cdot g(n) \quad \Leftrightarrow \quad \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Properties of the Asymptotics Relations

O, Ω, Θ, o, and ω define relations:

- all of the relations are **transitive**, e.g.:

$$f \in O(g) \quad \text{and} \quad g \in O(h) \quad \Rightarrow \quad f \in O(h)$$

- O, Ω, and Θ are **reflexive**:

$$f \in O(f) \qquad f \in \Omega(f) \qquad f \in \Theta(f)$$

- only Θ is **symmetric**:

$$f \in \Theta(g) \quad \Leftrightarrow \quad g \in \Theta(f)$$

- and there is a **transpose symmetry**:

$$f \in O(g) \quad \Leftrightarrow \quad g \in \Omega(f)$$
$$f \in o(g) \quad \Leftrightarrow \quad g \in \omega(f)$$

Example: Asymptotics of the Fibonacci Numbers

"Famous" inequality

$$2^{\lfloor \frac{n}{2} \rfloor} \leq f_n \leq 2^n$$

$f_n \in O(2^n)$ (with $c = 1$, proof by induction):

- (Base case) for $n = 0$: $f_0 = 1 \leq 2^0 = 1$
- (Base case) for $n = 1$: $f_1 = 1 \leq 2^1 = 2$
- (Inductive case) from $n - 1$ and $n - 2$ to n ($n \geq 2$):

$$f_n = f_{n-1} + f_{n-2} \leq 2^{n-1} + 2^{n-2} = 3 \cdot 2^{n-2} \leq 2^n$$

$f_n \in \Omega(2^{n/2})$ (proof by induction over $k = n/2$ – only for even n):

- (Base case) for $k = 0 \Rightarrow n = 0$: $f_0 = 1 \geq 2^0 = 1$
- (Ind. case) induction step: from $n = 2k - 2$ to $n = 2k$ ($n \geq 2$):

$$f_{2k} = f_{2k-1} + f_{2k-2} \geq 2f_{2k-2} = 2f_{2(k-1)} \geq 2 \cdot 2^{k-1} = 2^k$$