WS 2007/2008

Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

Chapter 0 Administrative Issues

Chapter 0 Administrative Issues

- Lectures:
  - 2 Hours/Week
  - Times/Place: Tuesday, 11:30-13:00, room MI 00.08.038
- Tutorials:
  - 2 Hours/Week

    http://wwwmayr.informatik.tu-muenchen.de/lehre/2007WS/fa-cse/tutorial.html
  - Proposal for Times/Plase : Wednesday, 11:00-12:30, room MI 03.11.018
- Office Hours:
  - By Appointment (chibisov@in.tum.de)
- Written examination

Chapter 0 Administrative Issues

- Lectures:
  - 2 Hours/Week
  - Times/Place: Tuesday, 11:30-13:00, room MI 00.08.038
- Tutorials:
  - 2 Hours/Week

    http://wwwmayr.informatik.tu-muenchen.de/lehre/2007WS/fa-cse/tutorial.html
  - Proposal for Times/Plase : Wednesday, 11:00-12:30, room MI 03.11.018
- Office Hours:
  - By Appointment (chibisov@in.tum.de)
- Written examination

Chapter 0 Administrative Issues

- Lectures:
  - 2 Hours/Week
  - Times/Place: Tuesday, 11:30-13:00, room MI 00.08.038
- Tutorials:
  - 2 Hours/Week

    http://wwwmayr.informatik.tu-muenchen.de/lehre/2007WS/fa-cse/tutorial.html
  - Proposal for Times/Plase : Wednesday, 11:00-12:30, room MI 03.11.018
- Office Hours:
  - By Appointment (chibisov@in.tum.de)
- Written examination

Chapter 0 Administrative Issues

- Lectures:
  - 2 Hours/Week
  - Times/Place: Tuesday, 11:30-13:00, room MI 00.08.038
- Tutorials:
  - 2 Hours/Week

    http://wwwmayr.informatik.tu-muenchen.de/lehre/2007WS/fa-cse/tutorial.html
  - Proposal for Times/Plase : Wednesday, 11:00-12:30, room MI 03.11.018
- Office Hours:
  - By Appointment (chibisov@in.tum.de)
- Written examination

1. Goals of this Course

In this course you should learn to formalize and model algorithmic problems in such a way that they become accessible to techniques based on such things as graphs strings, algebraic objects, etc. We will be studying standard approaches to problems formulated within these models. Each algorithm will be derived and analyzed in terms of their time and space complexity. At the end of this course you should understand the underlying algorithmic methodologies and, ideally, be able to adapt the algorithms shown here to problems related, but not identical, to those shown in this class.

- Introduction to algorithmics terminolgy
- Formalization of algorithmic problems
- Fundamental methodologies of algorithm design
- Algorithms for standard problems
- Basic methods of algorithm analysis
- Primitive and higher data structures
- Tutorials covering all this

- Introduction to algorithmics terminolgy
- Formalization of algorithmic problems
- Fundamental methodologies of algorithm design
- Algorithms for standard problems
- Basic methods of algorithm analysis
- Primitive and higher data structures
- Tutorials covering all this

- Introduction to algorithmics terminolgy
- Formalization of algorithmic problems
- Fundamental methodologies of algorithm design
- Algorithms for standard problems
- Basic methods of algorithm analysis
- Primitive and higher data structures
- Tutorials covering all this

- Introduction to algorithmics terminolgy
- Formalization of algorithmic problems
- Fundamental methodologies of algorithm design
- Algorithms for standard problems
- Basic methods of algorithm analysis
- Primitive and higher data structures
- Tutorials covering all this

- Introduction to algorithmics terminolgy
- Formalization of algorithmic problems
- Fundamental methodologies of algorithm design
- Algorithms for standard problems
- Basic methods of algorithm analysis
- Primitive and higher data structures
- Tutorials covering all this

- Introduction to algorithmics terminolgy
- Formalization of algorithmic problems
- Fundamental methodologies of algorithm design
- Algorithms for standard problems
- Basic methods of algorithm analysis
- Primitive and higher data structures
- Tutorials covering all this

- Introduction to algorithmics terminolgy
- Formalization of algorithmic problems
- Fundamental methodologies of algorithm design
- Algorithms for standard problems
- Basic methods of algorithm analysis
- Primitive and higher data structures
- Tutorials covering all this

2. Contents

2. Contents

- Introduction, Basics, Notation
- Deriving Algorithms by Induction
- Sorting and Searching
- Data Structures
- Algorithms on Graphs
- Algorithms on Texts
- Algebraic and Numerical Problems

2. Contents

- Introduction, Basics, Notation
- Deriving Algorithms by Induction
- Sorting and Searching
- Data Structures
- Algorithms on Graphs
- Algorithms on Texts
- Algebraic and Numerical Problems

2. Contents

- Introduction, Basics, Notation
- Deriving Algorithms by Induction
- Sorting and Searching
- Data Structures
- Algorithms on Graphs
- Algorithms on Texts
- Algebraic and Numerical Problems

2. Contents

- Introduction, Basics, Notation
- Deriving Algorithms by Induction
- Sorting and Searching
- Data Structures
- Algorithms on Graphs
- Algorithms on Texts
- Algebraic and Numerical Problems

2. Contents

- Introduction, Basics, Notation
- Deriving Algorithms by Induction
- Sorting and Searching
- Data Structures
- Algorithms on Graphs
- Algorithms on Texts
- Algebraic and Numerical Problems

2. Contents

- Introduction, Basics, Notation
- Deriving Algorithms by Induction
- Sorting and Searching
- Data Structures
- Algorithms on Graphs
- Algorithms on Texts
- Algebraic and Numerical Problems

2. Contents

- Introduction, Basics, Notation
- Deriving Algorithms by Induction
- Sorting and Searching
- Data Structures
- Algorithms on Graphs
- Algorithms on Texts
- Algebraic and Numerical Problems

3. Literature

- Robert Sedgewick
  Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching (3rd Edition).
  Addison-Wesley Publishing Company, Reading (MA), 1990
- T.H. Cormen, C.E. Leiserson, and R.L. Rivest.
  Introduction to Algorithms.
  MIT Press, McGraw-Hill Book Company, 1990
- Donald E. Knuth.
  Art of Computer Programming, Volume 1: Fundamental Algorithms.
  Addison-Wesley Publishing Company, Reading (MA), 1973

Chapter I Motivation

1. Introduction

Definition 1

An Algorithm is an unambiguously secified method for obtaining some desired output, given some input. Here we consider algorithms satisfying the following special properties:

Chapter I Motivation

1. Introduction

Definition 1

An Algorithm is an unambiguously secified method for obtaining some desired output, given some input. Here we consider algorithms satisfying the following special properties:

- sequential: At any point during the execution, exactly one operation is carried out.
- deterministic: At any point during the execution, the subsequent step is uniquely defined.
- statically finite: The description of the algorithm requires only a finite amount of space.
- dynamically finite: At any point during the execution, only a finite amount of storage is occupied.
- termination: The execution is guaranteed to end after a finite number of steps.

Chapter I Motivation

1. Introduction

Definition 1

An Algorithm is an unambiguously secified method for obtaining some desired output, given some input. Here we consider algorithms satisfying the following special properties:

- sequential: At any point during the execution, exactly one operation is carried out.
- deterministic: At any point during the execution, the subsequent step is uniquely defined.
- statically finite: The description of the algorithm requires only a finite amount of space.
- dynamically finite: At any point during the execution, only a finite amount of storage is occupied.
- termination: The execution is guaranteed to end after a finite number of steps.

Chapter I Motivation

1. Introduction

Definition 1

An Algorithm is an unambiguously secified method for obtaining some desired output, given some input. Here we consider algorithms satisfying the following special properties:

- sequential: At any point during the execution, exactly one operation is carried out.
- deterministic: At any point during the execution, the subsequent step is uniquely defined.
- statically finite: The description of the algorithm requires only a finite amount of space.
- dynamically finite: At any point during the execution, only a finite amount of storage is occupied.
- termination: The execution is guaranteed to end after a finite number of steps.

Chapter I Motivation

1. Introduction

Definition 1

An Algorithm is an unambiguously secified method for obtaining some desired output, given some input. Here we consider algorithms satisfying the following special properties:

- sequential: At any point during the execution, exactly one operation is carried out.
- deterministic: At any point during the execution, the subsequent step is uniquely defined.
- statically finite: The description of the algorithm requires only a finite amount of space.
- dynamically finite: At any point during the execution, only a finite amount of storage is occupied.
- termination: The execution is guaranteed to end after a finite number of steps.

Chapter I Motivation

1. Introduction

Definition 1

An Algorithm is an unambiguously secified method for obtaining some desired output, given some input. Here we consider algorithms satisfying the following special properties:

- sequential: At any point during the execution, exactly one operation is carried out.
- deterministic: At any point during the execution, the subsequent step is uniquely defined.
- statically finite: The description of the algorithm requires only a finite amount of space.
- dynamically finite: At any point during the execution, only a finite amount of storage is occupied.
- termination: The execution is guaranteed to end after a finite number of steps.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

Example 2

Examples of real-life problems requiring elaborate, efficient algorithms:

- Data organization and efficient lookup in a web search engine
- Data management in a large customer database
- Weather forecast by simulation of fluid flows
- Assembly of the human genome from hundreds of thousands of sequenced fragments
- Computation of a VLSI layout
- Compression of an audio or video file
- Encryption and decrytion of secret information for transmission over the internet
- etc.

2. Algorithms and Efficiency

The efficiency of an algorithm is mostly measured in terms of the running time and the usage of (storage) space during its execution. Both are typically specified as functions of the input size (in bits). Mostly, the running time is specified as the *number of operations* executed (e.g. additions, comparisons).

Example 3

Suppose that a given machine takes 1μs per operation. Let us consider different algorithms of varying time complexity for the same problem. We show the *runnning time* $T(n)$ (in seconds, hours, etc.) for different *input sizes* n and for different algorithms whose time complexities are

Example 3

Suppose that a given machine takes 1μs per operation. Let us consider different algorithms of varying time complexity for the same problem. We show the *runnning time* $T(n)$ (in seconds, hours, etc.) for different *input sizes* n and for different algorithms whose time complexities are

- $t(n) = 1000n$ (A1)
- $t(n) = 1000n \log n$ (A2)
- $t(n) = n^2$ (A3)
- $t(n) = 10n^3$ (A4)
- $t(n) = 2^n$ (A5).

Example 3

Suppose that a given machine takes 1μs per operation. Let us consider different algorithms of varying time complexity for the same problem. We show the *runnning time* $T(n)$ (in seconds, hours, etc.) for different *input sizes* n and for different algorithms whose time complexities are

- $t(n) = 1000n$ (A1)
- $t(n) = 1000n \log n$ (A2)
- $t(n) = n^2$ (A3)
- $t(n) = 10n^3$ (A4)
- $t(n) = 2^n$ (A5).

Example 3

Suppose that a given machine takes 1μs per operation. Let us consider different algorithms of varying time complexity for the same problem. We show the *runnning time* $T(n)$ (in seconds, hours, etc.) for different *input sizes* n and for different algorithms whose time complexities are

- $t(n) = 1000n$ (A1)
- $t(n) = 1000n \log n$ (A2)
- $t(n) = n^2$ (A3)
- $t(n) = 10n^3$ (A4)
- $t(n) = 2^n$ (A5).

Example 3

Suppose that a given machine takes 1μs per operation. Let us consider different algorithms of varying time complexity for the same problem. We show the *runnning time* $T(n)$ (in seconds, hours, etc.) for different *input sizes* n and for different algorithms whose time complexities are

- $t(n) = 1000n$ (A1)
- $t(n) = 1000n \log n$ (A2)
- $t(n) = n^2$ (A3)
- $t(n) = 10n^3$ (A4)
- $t(n) = 2^n$ (A5).

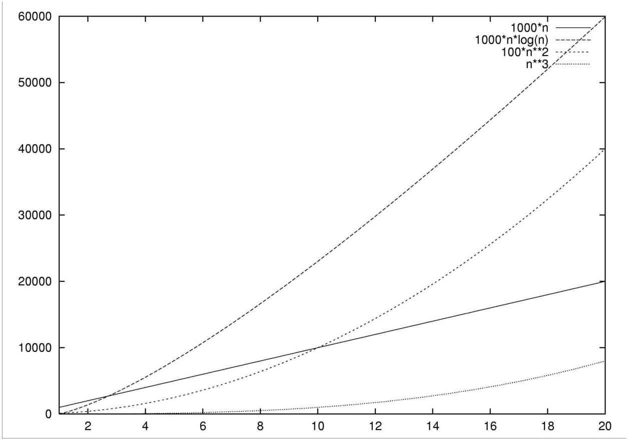
Example 3

Suppose that a given machine takes 1μs per operation. Let us consider different algorithms of varying time complexity for the same problem. We show the *runnning time* $T(n)$ (in seconds, hours, etc.) for different *input sizes* n and for different algorithms whose time complexities are

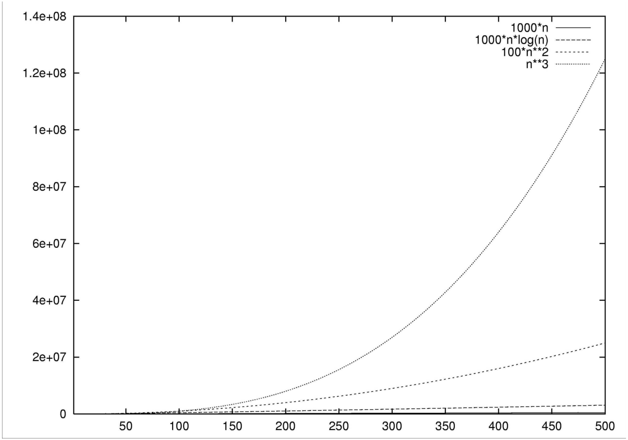
- $t(n) = 1000n$ (A1)
- $t(n) = 1000n \log n$ (A2)
- $t(n) = n^2$ (A3)
- $t(n) = 10n^3$ (A4)
- $t(n) = 2^n$ (A5).

	20	50	100	200	500	1000	10000
A1	0.02s	0.05s	0.1s	0.2s	0.5s	1s	10s
A2	0.09s	0.3s	0.6s	1.5s	4.5s	10s	2min
A3	0.04s	0.25s	1s	4s	25s	2min	2.8h
A4	0.02s	1s	10s	1min	21min	2.7h	116d
A5	1s	35yrs	3×10^4cent.				

Here is a plot of the running times (in μs) as a function of the input size. Algorithms more efficient for some n coste more for other n.



But in general these examples show us that, for sufficiently large input sizes, the complexity of an algorithm determines whether or not a given algorithm is usable in practice:



But in general these examples show us that, for sufficiently large input sizes, the complexity of an algorithm determines whether or not a given algorithm is usable in practice.

Some argue as follows:
" There is no need for efficient algorithms — If some computation is too slow, I'll buy a faster machine."

Well, all that results from a faster machine is a different *constant factor* in the running time. This, however, is typically dwarfed by the effect of slightly increasing n if the time complexity is high. Let us examine this phenomenon:

But in general these examples show us that, for sufficiently large input sizes, the complexity of an algorithm determines whether or not a given algorithm is usable in practice.

Some argue as follows:
" There is no need for efficient algorithms — If some computation is too slow, I'll buy a faster machine."

Well, all that results from a faster machine is a different *constant factor* in the running time. This, however, is typically dwarfed by the effect of slightly increasing n if the time complexity is high. Let us examine this phenomenon:

But in general these examples show us that, for sufficiently large input sizes, the complexity of an algorithm determines whether or not a given algorithm is usable in practice.

Some argue as follows:
" There is no need for efficient algorithms — If some computation is too slow, I'll buy a faster machine."

Well, all that results from a faster machine is a different *constant factor* in the running time. This, however, is typically dwarfed by the effect of slightly increasing n if the time complexity is high. Let us examine this phenomenon:

3. Maximum Feasible Input Size

Suppose we are using a machine that executes f operations per second (in the example: $f = 10^6$). The algorithm requires $t(n)$ operations on inputs of size n (, where $t(n)$ strictly grows in n). The measured running time then is

$$T(n) = \frac{t(n)}{f} \text{ [in seconds]}$$

If we need the computation to be finished within s seconds, the input size is limited to

$$n \leq t^{-1}(s \cdot f).$$

3. Maximum Feasible Input Size

Suppose we are using a machine that executes f operations per second (in the example: $f = 10^6$). The algorithm requires $t(n)$ operations on inputs of size n (, where $t(n)$ strictly grows in n). The measured running time then is

$$T(n) = \frac{t(n)}{f} \text{ [in seconds]}$$

If we need the computation to be finished within s seconds, the input size is limited to

$$n \leq t^{-1}(s \cdot f).$$

3. Maximum Feasible Input Size

Suppose we are using a machine that executes f operations per second (in the example: $f = 10^6$). The algorithm requires $t(n)$ operations on inputs of size n (, where $t(n)$ strictly grows in n). The measured running time then is

$$T(n) = \frac{t(n)}{f} \text{ [in seconds]}$$

If we need the computation to be finished within s seconds, the input size is limited to

$$n \leq t^{-1}(s \cdot f).$$

3. Maximum Feasible Input Size

Suppose we are using a machine that executes f operations per second (in the example: $f = 10^6$). The algorithm requires $t(n)$ operations on inputs of size n (, where $t(n)$ strictly grows in n). The measured running time then is

$$T(n) = \frac{t(n)}{f} \text{ [in seconds]}$$

If we need the computation to be finished within s seconds, the input size is limited to

$$n \leq t^{-1}(s \cdot f).$$

Example 4

Suppose the algorithm's time complexity is $t(n) = n^2$, and suppose the maximum permissible running time is s. Increasing the machine's speed f by a factor of 2 (1000) allows us to increase the input size n by only a factor of 1.414 (31.62).

Example 5

Suppose the algorithm's time complexity is $t(n) = 2^n$, and suppose the maximum permissible running time is s. Increasing the machine's speed f by a factor of 2 (1000) allows us to increase the input size n only by the additive constant 1 $(\lfloor \log 1000 \rfloor = 9)$.

Exercise 1

Suppose the algorithm's time complexity is $t(n) = 2^{log(n)}$, and suppose the maximum permissible running time is s. What increasing of the input size n would be caused by increasing of the machine's speed f by a factor of 2 (1000) ?

Example 4

Suppose the algorithm's time complexity is $t(n) = n^2$, and suppose the maximum permissible running time is s. Increasing the machine's speed f by a factor of 2 (1000) allows us to increase the input size n by only a factor of 1.414 (31.62).

Example 5

Suppose the algorithm's time complexity is $t(n) = 2^n$, and suppose the maximum permissible running time is s. Increasing the machine's speed f by a factor of 2 (1000) allows us to increase the input size n only by the additive constant 1 $(\lfloor \log 1000 \rfloor = 9)$.

Exercise 1

Suppose the algorithm's time complexity is $t(n) = 2^{log(n)}$, and suppose the maximum permissible running time is s. What increasing of the input size n would be caused by increasing of the machine's speed f by a factor of 2 (1000) ?

Chapter II Introduction, Basics and Notation

1. Introductory Example: The Fibonacci Numbers

Problem: How fast does a population of rabbits grow?

Suppose:

Chapter II Introduction, Basics and Notation

1. Introductory Example: The Fibonacci Numbers

Problem: How fast does a population of rabbits grow?

Suppose:

- At the beginning of the first month, 1 pair of rabbits exists
- After being born, a rabbit begins breeding at the age of 1 month
- Each pair of rabbits produces one new pair (1 female, 1 male) per month
- Rabbits live infinitely long
- We disregard the genetic effects of inbreeding

Chapter II Introduction, Basics and Notation

1. Introductory Example: The Fibonacci Numbers

Problem: How fast does a population of rabbits grow?

Suppose:

- At the beginning of the first month, 1 pair of rabbits exists
- After being born, a rabbit begins breeding at the age of 1 month
- Each pair of rabbits produces one new pair (1 female, 1 male) per month
- Rabbits live infinitely long
- We disregard the genetic effects of inbreeding

Chapter II Introduction, Basics and Notation

1. Introductory Example: The Fibonacci Numbers

Problem: How fast does a population of rabbits grow?

Suppose:

- At the beginning of the first month, 1 pair of rabbits exists
- After being born, a rabbit begins breeding at the age of 1 month
- Each pair of rabbits produces one new pair (1 female, 1 male) per month
- Rabbits live infinitely long
- We disregard the genetic effects of inbreeding

Chapter II Introduction, Basics and Notation

1. Introductory Example: The Fibonacci Numbers

Problem: How fast does a population of rabbits grow?

Suppose:

- At the beginning of the first month, 1 pair of rabbits exists
- After being born, a rabbit begins breeding at the age of 1 month
- Each pair of rabbits produces one new pair (1 female, 1 male) per month
- Rabbits live infinitely long
- We disregard the genetic effects of inbreeding

Chapter II Introduction, Basics and Notation

1. Introductory Example: The Fibonacci Numbers

Problem: How fast does a population of rabbits grow?

Suppose:

- At the beginning of the first month, 1 pair of rabbits exists
- After being born, a rabbit begins breeding at the age of 1 month
- Each pair of rabbits produces one new pair (1 female, 1 male) per month
- Rabbits live infinitely long
- We disregard the genetic effects of inbreeding

We can see from the above specification that:

- In the n-th month there exist the rabbits that already existed in the $(n-1)$-th month, and
- those who existed in the $(n-2)$-th month were old enough to breed. Hence the latter have produced offspring.

So the number f_n of rabbits existing in the n-th month can be described by the following recurrence relation:

$$\begin{array}{rcl} f_1 & = & 1 \\ f_2 & = & 1 \\ f_n & = & f_{n-1} + f_{n-2} \text{ for } n \geq 3 \end{array}$$

Definition 6

For $n \geq 1$, the numbers f_n defined above are known as Fibonacci Numbers.

1.1 1st Algorithm for Computing Fibonacci Numbers

This algorithm is a straightforward implementation of the above (where we denote $f(n) := f_n$):

Algorithm:

unsigned f(unsigned n){
  if $(n \leq 2)$ then return 1
  else return $f(n-1) + f(n-2)$
  fi
}

The recurrence relation leads to a simple recursive algorithm: a function that repeatedly calls itself. Let us take a look at the complexity of this algorithm.

1.1 1st Algorithm for Computing Fibonacci Numbers

This algorithm is a straightforward implementation of the above (where we denote $f(n) := f_n$):

Algorithm:

unsigned f(unsigned n){
  if ($n \leq 2$) then return 1
  else return $f(n-1) + f(n-2)$
  fi
}

The recurrence relation leads to a simple recursive algorithm: a function that repeatedly calls itself. Let us take a look at the complexity of this algorithm.

1.1 1st Algorithm for Computing Fibonacci Numbers

This algorithm is a straightforward implementation of the above (where we denote $f(n) := f_n$):

Algorithm:

unsigned f(unsigned n){
  if ($n \leq 2$) then return 1
  else return $f(n-1) + f(n-2)$
  fi
}

The recurrence relation leads to a simple recursive algorithm: a function that repeatedly calls itself. Let us take a look at the complexity of this algorithm.

We measure the complexity in terms of the number $t_{\mathsf{rek}}(n)$ of arithmetic operations to be performed as a function of the value of n (rather than the size $\lceil \log n \rceil$ of input n). Given an algorithm like this, the complexity can easily be written down in a recursive way, symetrically to the algorithm itself. It holds that

The problem with this formulation is that we need to obtain an explicit representation of $t_{\mathsf{rek}}(n)$ in a separate step.

We measure the complexity in terms of the number $t_{\mathsf{rek}}(n)$ of arithmetic operations to be performed as a function of the value of n (rather than the size $\lceil \log n \rceil$ of input n). Given an algorithm like this, the complexity can easily be written down in a recursive way, symetrically to the algorithm itself. It holds that

- $t_{\mathsf{rek}}(1) = 0$
- $t_{\mathsf{rek}}(2) = 0$
- $t_{\mathsf{rek}}(n) = 3 + t_{\mathsf{rek}}(n-1) + t_{\mathsf{rek}}(n-2)$ for $n \geq 3$.

The problem with this formulation is that we need to obtain an explicit representation of $t_{\mathsf{rek}}(n)$ in a separate step.

We measure the complexity in terms of the number $t_{\mathsf{rek}}(n)$ of arithmetic operations to be performed as a function of the value of n (rather than the size $\lceil \log n \rceil$ of input n). Given an algorithm like this, the complexity can easily be written down in a recursive way, symetrically to the algorithm itself. It holds that

- $t_{\mathsf{rek}}(1) = 0$
- $t_{\mathsf{rek}}(2) = 0$
- $t_{\mathsf{rek}}(n) = 3 + t_{\mathsf{rek}}(n-1) + t_{\mathsf{rek}}(n-2)$ for $n \geq 3$.

The problem with this formulation is that we need to obtain an explicit representation of $t_{\mathsf{rek}}(n)$ in a separate step.

We measure the complexity in terms of the number $t_{\text{rek}}(n)$ of arithmetic operations to be performed as a function of the value of n (rather than the size $\lceil \log n \rceil$ of input n). Given an algorithm like this, the complexity can easily be written down in a recursive way, symetrically to the algorithm itself. It holds that

- $t_{\text{rek}}(1) = 0$
- $t_{\text{rek}}(2) = 0$
- $t_{\text{rek}}(n) = 3 + t_{\text{rek}}(n-1) + t_{\text{rek}}(n-2)$ for $n \geq 3$.

The problem with this formulation is that we need to obtain an explicit representation of $t_{\text{rek}}(n)$ in a separate step.

Lemma 1

It holds that $t_{rek}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

Assume the statement holds for $n-2$, $n-1$:

Using

- $f_n = f_{n-1} + f_{n-2}$

we obtain:

Lemma 1

It holds that $t_{rek}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n - 2$, $n - 1$:

Using

- $f_n = f_{n-1} + f_{n-2}$

we obtain:

Lemma 1

It holds that $t_{\textit{rek}}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n-2$, $n-1$:

Using

- $f_n = f_{n-1} + f_{n-2}$

we obtain:

Lemma 1

It holds that $t_{rek}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n-2$, $n-1$:

Using

- $f_n = f_{n-1} + f_{n-2}$

we obtain:

Lemma 1

It holds that $t_{\textit{rek}}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n - 2$, $n - 1$:

- $t_{\text{rek}}(n-2) = 3 \cdot f_{n-2} - 3$,
- $t_{\text{rek}}(n-1) = 3 \cdot f_{n-1} - 3$.

Using

- $f_n = f_{n-1} + f_{n-2}$

we obtain:

Lemma 1

It holds that $t_{rek}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n-2$, $n-1$:

- $t_{\text{rek}}(n-2) = 3 \cdot f_{n-2} - 3$,
- $t_{\text{rek}}(n-1) = 3 \cdot f_{n-1} - 3$.

Using

- $f_n = f_{n-1} + f_{n-2}$

we obtain:

Lemma 1

It holds that $t_{\textit{rek}}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n - 2$, $n - 1$:

- $t_{\text{rek}}(n-2) = 3 \cdot f_{n-2} - 3$,
- $t_{\text{rek}}(n-1) = 3 \cdot f_{n-1} - 3$.

Using

- $f_n = f_{n-1} + f_{n-2}$

we obtain:

Lemma 1

It holds that $t_{\textit{rek}}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n-2$, $n-1$:

- $t_{\text{rek}}(n-2) = 3 \cdot f_{n-2} - 3$,
- $t_{\text{rek}}(n-1) = 3 \cdot f_{n-1} - 3$.

Using

- $t_{\text{rek}}(n) = 3 + t_{\text{rek}}(n-1) + t_{\text{rek}}(n-2)$ for $n \geq 3$,
- $f_n = f_{n-1} + f_{n-2}$

we obtain:

- $t_{\text{rek}}(n) = \underbrace{3 + \overbrace{3 \cdot f_{n-2} - 3 + 3 \cdot f_{n-1}} - 3} = 3 \cdot f_n - 3$

Lemma 1

It holds that $t_{\textit{rek}}(n) = 3 \cdot f_n - 3$ *for* $n \geq 1$.

Proof.

The base case:

- $t_{\text{rek}}(1) = 0$,
- $t_{\text{rek}}(2) = 0$.

Assume the statement holds for $n-2$, $n-1$:

- $t_{\text{rek}}(n-2) = 3 \cdot f_{n-2} - 3$,
- $t_{\text{rek}}(n-1) = 3 \cdot f_{n-1} - 3$.

Using

- $t_{\text{rek}}(n) = 3 + t_{\text{rek}}(n-1) + t_{\text{rek}}(n-2)$ for $n \geq 3$,
- $f_n = f_{n-1} + f_{n-2}$

we obtain:

- $t_{\text{rek}}(n) = \underbrace{3 + \overbrace{3 \cdot f_{n-2} - 3 + 3 \cdot f_{n-1}} - 3} = 3 \cdot f_n - 3$