universität wien

Fakultät für Informatik

Algorithms and data structures VU

4 hours / 6 ECTS points

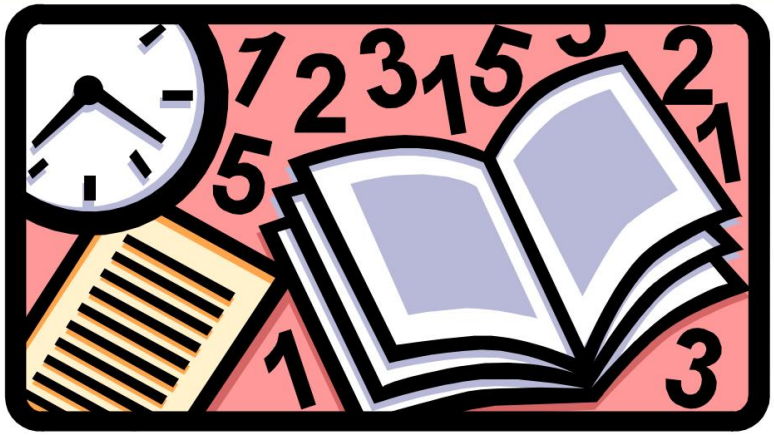
Lecture part

Ass.-Prof. Dr. Kathrin Hanauer Research
Group Theory and Application of Algorithms

Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta
Dipl.-Ing. Helmut Wanek
Workflow Systems and Technology Research Group

SS 2023

Content overview

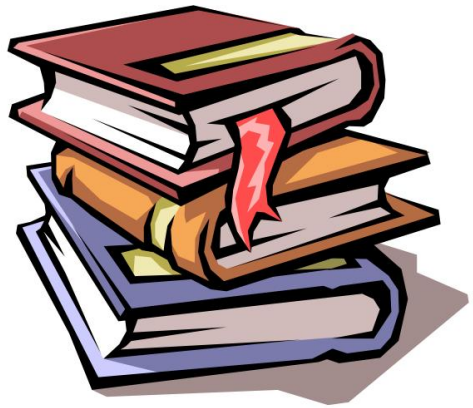


1. Algorithms
   paradigms, analysis
2. Data structures
   Motivation, overview
3. Vector
   hashing, sorting
4. Lists
   Linear memory structures, stack, queue
5. Trees
   Search structures
6. Graph
   traversal and optimization algorithms

literature

R. Sedgewick, *Algorithms in C++* (Parts 1-4), Addison Wesley, 3rd revised edition, 2002

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms,* published by MIT Press, 2009



thanksgiving

My special goes to collaboration and reviewing the slides
Thanks to Helmut Wanek and Martin Polaschek

My further thanks go to numerous students from recent years who provided the basis for some of the dynamic examples of the VO as part of their exercises.

Chapter 1
Algorithms

1.1 Motivation

Algorithms

Procedural regulations, instruction sequences, process modeling, described solutions

Two goals

1. Find algorithms for problems!

Find and “construct” solutions 2.

Find “good” algorithms!

“better” algorithms

faster, more complete, more correct, ...

“more powerful” data structures, more

compact, more efficient, more flexible, ...

In general: savings in computing time and/or storage space

Find algorithms for problems!

Task: “Sum of the integers up to n”

Straight-forward solution: “Summing up the individual values between 1 and n”

$$sum \quad \ddot{y} \quad \ddot{\ddot{y}}_{i\,1\,\ddot{y}}^{n} \; i$$

Realization (C/C++ program) **int**

```
sum(int n) {
    int i, summe = 0; for(i=1;
    i<=n; i++) summe += i;
        return summe;

}
```

Compare with other programming approaches, e.g. while instead of for loop! ÿ alternative programming style

Alternative implementation (1)

Two alternatives

1. Alternative programming style

   Maintain problem-solving approach, but implement it in programming term
   revise

Example:

Loop shape (see above)

Recursion instead of iteration

```
int sum(int n) { if(n <=
    0) return 0; if(n == 1) return
    1; else return n+sum(n-1);


}
```

universität wien

2. Alternative solution

Choosing a different way to solve the problem, e.g

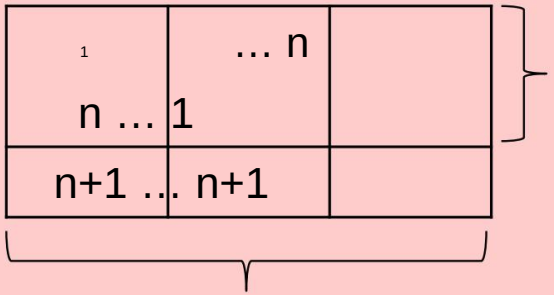
Gaussian molecular formula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

implementation

```
int sum(int n) {
    return (n*(n+1))/2;
}
```

Derivation of Gauss' molecular formula



= 2 * Sum from 1 to n

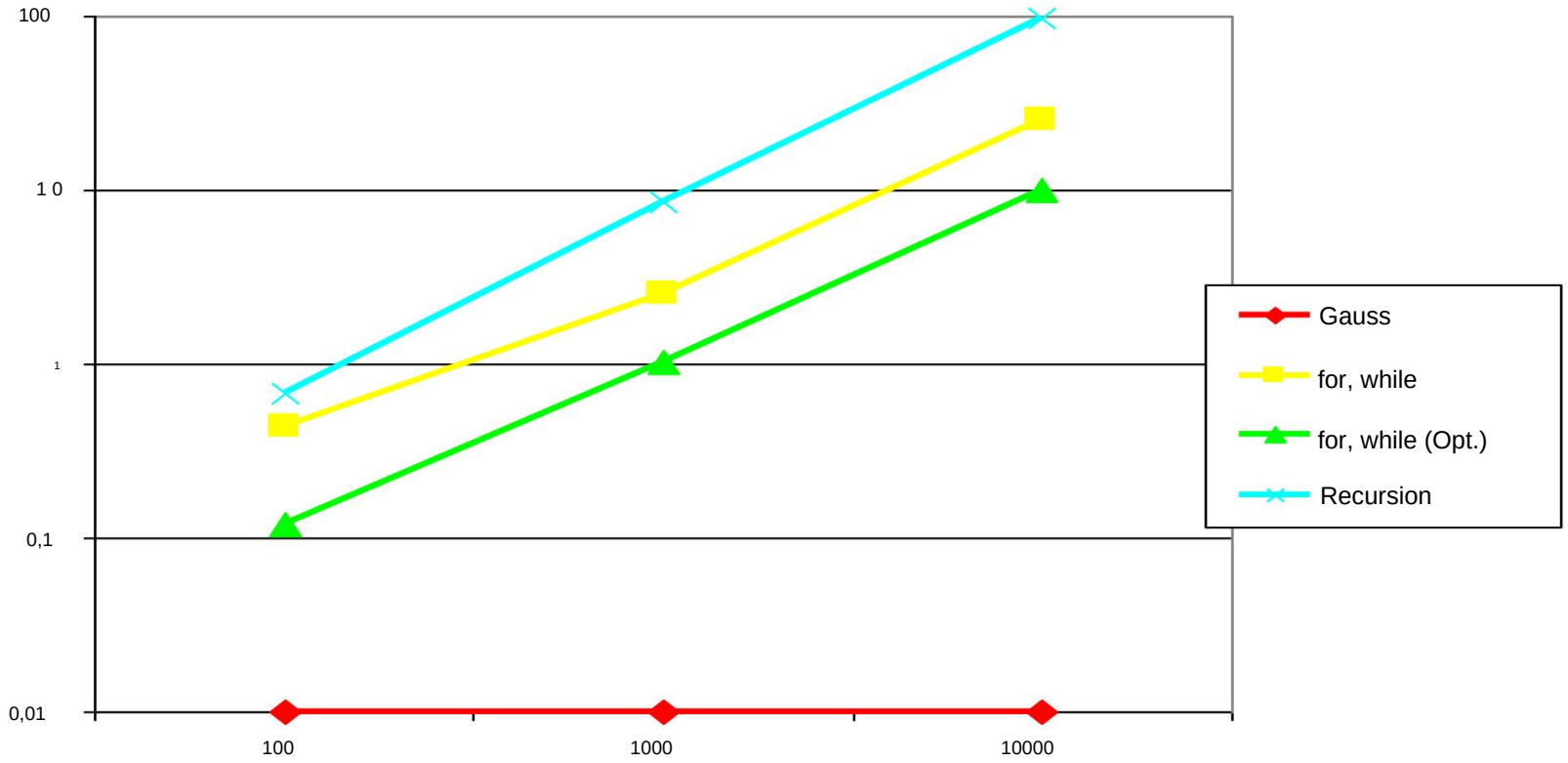
From this it follows that Sum formula

n * (n+1) / 2

Find “good” algorithms!

Comparison of running times

Sum calculation, 100000 repetitions, CPU: 200 MHz Pentium



What is good? Or maybe better?

- Problem: Comparing running times only leads to selective quality determination
  - Runtime, storage space consumption
  - Depending on
    - Computer
    - operating system
    - Compiler, ...
- Goal: Methodology for general quality comparison between Algorithms
  - Independent of external influences

1.2 Algorithm paradigms

General techniques for solving large classes of Problems

Greedy algorithms

"gluttonous, greedy" approach, choosing the local optimum

Divide-and-conquer algorithms

step-by-step decomposition of the problem of size n into smaller ones

Partial problems

Dynamic programming

Dynamic, successive construction of the solution from already calculated ones

Partial solutions

1.2.1 Greedy (1)

In each step of a *greedy* algorithm, the *option* that is *immediately* (locally) *optimal* is chosen (smallest or largest) *value* with respect to the *objective function* . The global view of the end goal is neglected.

Advantage:

Efficient problem solving, often very quickly, can in many cases find a relatively good solution

Disadvantage:

Often does not find an optimal solution

universität wien

Example: coin changing machine

"Change the amount of 18.- into as small a number as possible
Coins in size 10, 5 and 1"

Greedy approach:

choose largest coin
less than amount
spend the coin
subtract hers
Value of the amount
repeat until difference
equals 0

dh:

18.- - **10.-** = 8.- ÿ
8.- - **5.-** = 3.- ÿ
3.- - **1.-** = 2.- ÿ
2.- - **1.-** = 1.- ÿ
1.- - **1.-** = 0 ÿ

The solution to this problem is not just "good" but even optimal!

Greedy (3)

BUT

Problem with small change in the problem statement:

5.- ÿ 6.-

Coin values 10.-, ***6.-*** 1.-

greedy approach delivers

18.- - **10.-** = 8.- ÿ

8.- - **6.-** = 2.- ÿ

2.- - **1.-** = 1.- ÿ

1.- - **1.-** = 0 ÿ

ÿ 4 coins

But optimal would be 3 x 6.-

18.- - **6.-** = 12.- ÿ

12.- - **6.-** = 6.- ÿ

6.- - **6.-** = 0 ÿ

ÿ 3 coins

1.2.2 Divide-and-conquer (1)

In *divide-and- conquer* algorithms

Starting from a general abstraction , the problem is iteratively refined until solutions for simplified sub-problems have been found, from which an overall solution can be constructed.

This approach is often referred to as “stepwise refinement” or “top-down approach”.

Divide-and-conquer (2)

Different approaches

Problem size division

Decomposition of a problem of size n into a finite number of
Subproblems smaller than n

Step division

Splitting a task into a sequence (sequence) of individual ones
Subtasks

Case division

Identification of special cases for a general problem from a certain level of abstraction

...

Problem size division

By decomposition reducing the problem size, ie

P(n) ÿ k*P(m),
where k, m < n

Binary search

Find a number x in the (ascending) sorted sequence z1 ,z2 ,..,zn (generally: zl ,zl+1,..,zr with l=1, r=n) and determine its

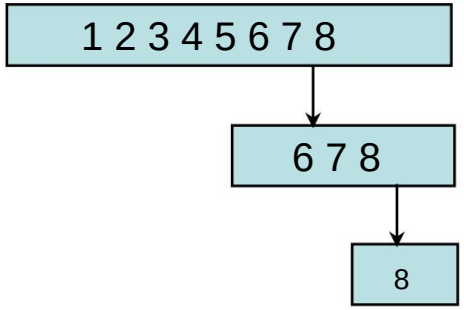
position i Decomposition: k =

1 and m ÿ n/2 Trivial: find middle index

m = (l+r)÷2 If zm=x result m,

otherwise If x<zm find x in the range zl ,z2 ,..,zm-1 else search x in the range zm+1,zm+2,..,zr

Find number 8



Step division

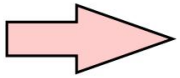
Idea: street sweeper philosophy:

"Breath - broom stroke - step"

Example

Salary increase

Designate employees
Find unique identification
Search the database
Determine current salary
Change to new salary
Store information
Notes change process



Store information

Delete old record
Insert new record

Case division

Approach: Identification of case differences in Problem data area

Example

Calculating the solutions to a quadratic equation

$$az^2 + bz + c = 0$$

$$z_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \quad \text{from 1.2}$$

if $q > 0$, 2 real roots

$q = 0$, real double solution $q < 0$, pair of complex roots

1.2.3 Dynamic Programming (1)

Problem

Often a division of the original problem into a 'small' number of Partial problems are not possible, but leads to an exponential algorithm.

But we know that there is only a polynomial number of subproblems.

Idea *Dynamic Programming*

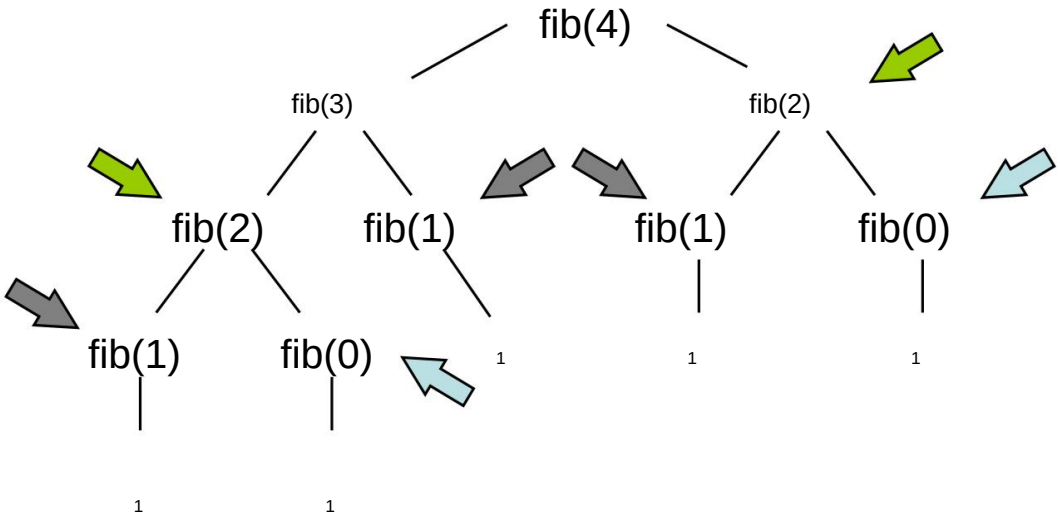
not start from problem size n and split up to size 1,
RATHER

Start with solution for problem size 1, combination of the calculated partial solutions until a solution for problem size n was reached.

Dynamic Programming (2)

Example: Calculating the Fibonacci numbers

```
int fib(int n) {
    if(n <= 1) return 1;
    return fib(n-1) + fib(n-2)
}
```

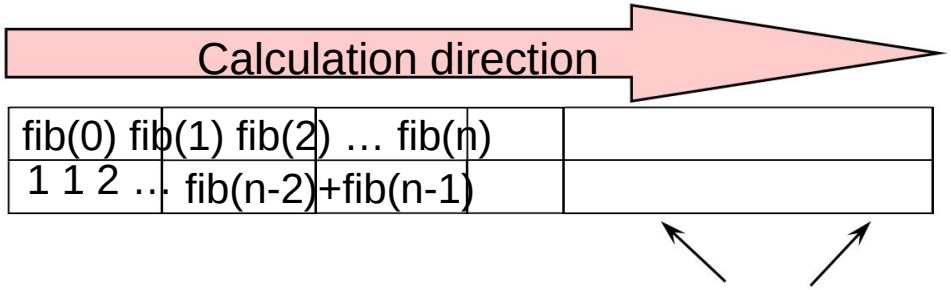


fib(2) calculated twice
fib(1) calculated 3x
fib(0) calculated 2x

Problem: Repeated solution of a sub-problem

solution

Start with calculation for **fib(0),** create a table of all calculated values and construction of the new values from the calculated table entries.




```
int fib(int n) {
        int F[MAXSIZE];
        F[0] = 1; F[1] = 1;
        for(i = 2; i <= n; i++) {
            F[i] = F[i-1] + F[i-2];
        }
        return F[n];
    }
```

Note: Improvement in runtime BUT additional storage space required

1.3 Analysis and evaluation of algorithms

The aim is to objectively evaluate algorithms

criteria

effectiveness

Is the problem solvable, can the approach be implemented in a program?

correctness

Does the algorithm do what it's supposed to do?

Termination

Does the algorithm stop, does it have a finite execution time?

Complexity

How fast is the algorithm ÿ Runtime complexity?

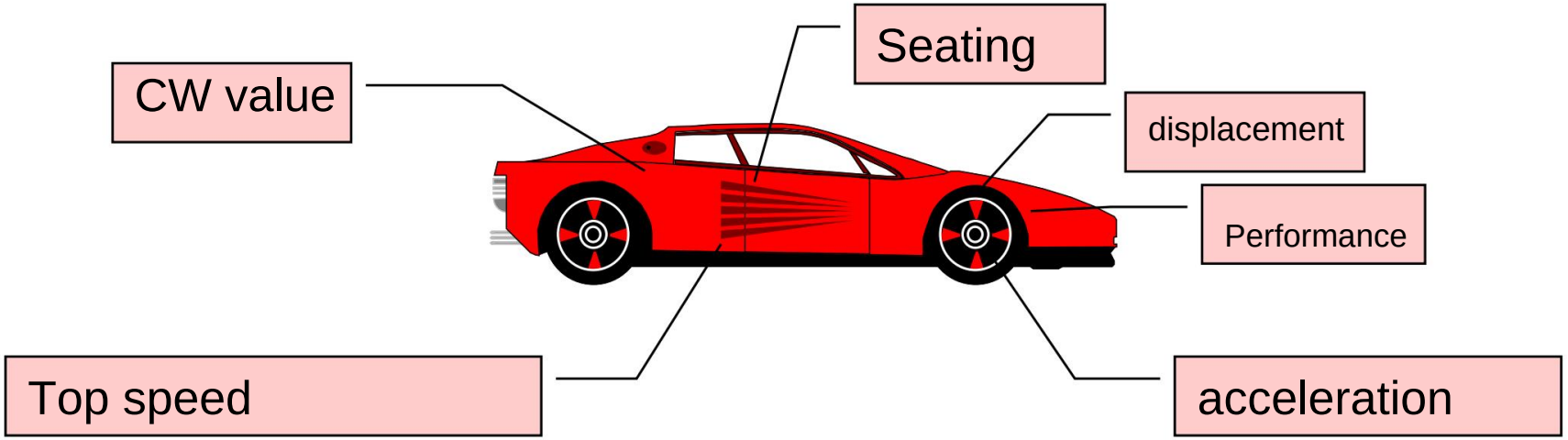
How structured is the algorithm ÿ structural complexity?

How much storage space does the algorithm need ÿ Space complexity?

Analogy: Car

Specific criteria

Selection

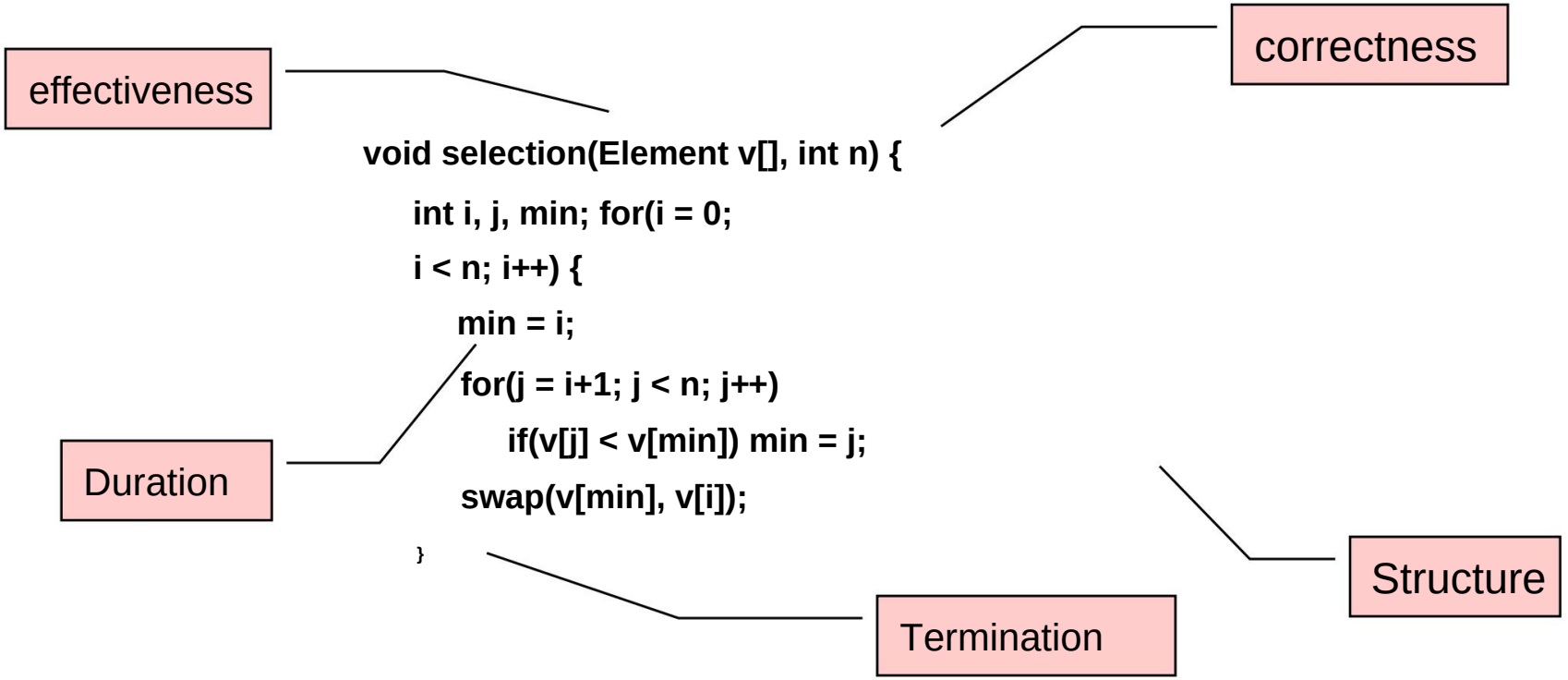


Statistical key figures

Assessment option, classification option sports cars, trucks, family sedans, ...

universität wien

Example sorting program



Classification

fast, correct, maintainable, problem-covering, finally, …

1.3.1 Effectiveness

Principle

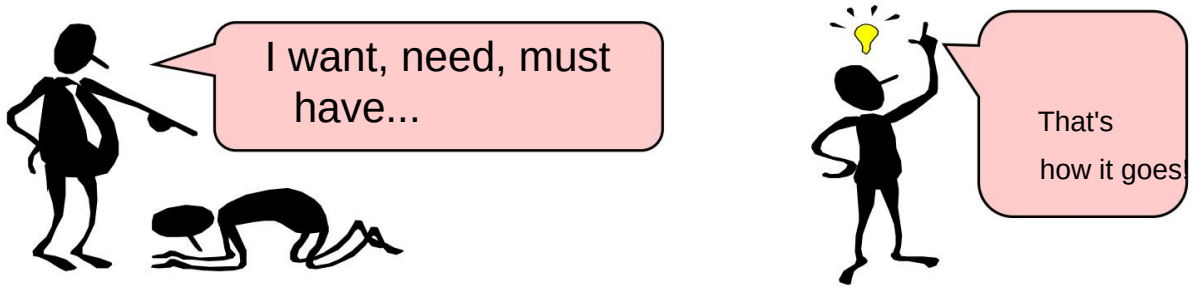
Algorithm can be formulated as an executable computer program.
effective ÿ it does work

Problem

formulation

Transformation of the problem description into an executable one
algorithm

Question: Is there even an executable solution to this problem?
Problem?



Does the algorithm produce the desired result?

Two procedures possible

- To test
- Verification

To test

- Complete testing is usually not possible
- Statistical approach is usually followed, e.g. path coverage (structural Complexity)
- At best “falsification” is achievable
  - Only errors can be found, but correctness cannot be found be proven

Verification

Mathematically oriented verification techniques allow
- Correctness of program pieces depending on
- To prove conditions on the input data (premises).

forces the developer to revisit design decisions and helps find
- logical errors

Algorithm needs to be understood

the larger a program system, the more difficult it is
- Verification (hardly important in practice)

Proof approach depends on the problem

Complete induction

Example: Gaussian molecular formula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Ind. start for
n = 1 ÿ 1*2/2 = 1 ÿ

Ind. prerequisite
(1+2+…+n-1) = (n-1)*n/2

Ind.
conclusion (1+2+…+n-1)+n = (n-1)*n/2 +n = = ((n-1)*n +2n)/2 = (n2 -n+2n)/2 = = n*(n+1)/2 ÿ

Program verification

different approaches

McCarthy, Naur, Floyd, Hoare, Knuth, Dijkstra, etc.

“Backward proof”:

Structured approach, based on *predicate transformation*

Checking whether the *weakest preconditions (wp)* on the input data guarantee that the program terminates in a state that meets the desired program goal.

This program goal, which describes the task of the program, is called *the correctness condition, KB*

For this purpose, the program may only consist of simple assignments, comparisons, while-form loops and sequences of statements. All other constructs must be translated.

Verification - example

Example

```
int sum(int n) {
    int s = 0;
    int i = 1;
    while(i <= n) {
        s = s + i;
        i = i + 1;
    }
    return s;
}
```

Backward proof

Correctness condition KB s $= \ddot{y}_{i=1}^{n} i$

(I) *Loop invariant SI*

SI: $s_{(} = \ddot{y}_{j=1}^{i-1}$ *fe-(1)g* $\ddot{y}\ddot{y}+$

(1) *S I Bed KB*

$s = (\ddot{y})_{j=1}^{i-1}(1)(\,)($ blood $\ddot{y}\,\ddot{y} + \ddot{y}\,\ddot{y}\,\ddot{y} =$ s $\ddot{y})_{j=1}^{i-1}($ feel $\ddot{y} = + \ddot{y} = \ddot{y}$ $1)$ $s \ddot{y}_{j=1}^{n}$ *j KB*

(2) (*SI Bed wp loop block SI*

$(\,|\,(\neg$ *wp s s i wp i* $i = +$ $1|\,(s = \ddot{y}_{j=1}^{i-1})\,(1))\,)\,(\,|\,(\,j\,i\,n\,\ddot{y}\,\ddot{y} + \ddot{y} = + wp\,s\,s\,i\,s = \ddot{y}_{j=1}^{i} j\,)\,\ddot{y}$

$= (i \quad 1\,1)\,\prime\,(\,$ and $\ddot{y} + \ddot{y} + \ddot{y}_{j=1}^{i}\,\ddot{y}\,) = (\,)\,($ hearing $\ddot{y}\,\ddot{y}$ $s \ddot{y}_{j=1}^{i-1}$ *jin)* $\ddot{y}\ddot{y}\ddot{y}\ddot{y}$ *SI Bed*

(II) *Rest of the program*

$s\,(\,0\,(|\,0\,(\,|\,)\,)$ *wp s wp* $\ddot{y}$ *and SI wp* $s = \ddot{y}_{j=1}^{11} j\,)\,(\ddot{y}\,\ddot{y}\,1\,1))\,\ddot{y}\ n$

$\ddot{y} = (0 \quad \ddot{y}_{j=1}^{0} j\,)\,(\ddot{y}\,\ddot{y}\,\ddot{y}\,\ddot{y}\ n \qquad n \quad 0$

1.3.3 Termination (1)

If the algorithm stops, that is, it has a finite
Execution time?

If not clear, the following technique can sometimes be used:

Find the determining quantity or property of the
Algorithm that satisfies the following 3 characteristics:

- A 1-1 mapping of this size to the integers can be constructed become.
- This quantity is positive.
- The size increases continuously during the execution of the algorithm from (decremented).

Termination (2)

Idea:

At the start of the algorithm, the size found has a predetermined positive starting value that continuously decreases. Since the size can never become negative, it follows that the algorithm must terminate before the size is less than 0.

Example:

Size n

Termination before n < 0

```
int sum(int n) {
    if(n <= 0) return 0;
    if(n == 1) return 1;
    else
    return n+sum(n-1);
}
```

Decrement

1.3.4 Runtime complexity

The *runtime complexity* provides information about this

Runtime behavior of algorithms depending on the

Problem size

Goal

Compare algorithms

Approach

Measuring the execution time of each instruction

Determine how often each statement is executed during program execution

becomes

Calculate total

Problem

Execution times depend on machine or system architecture, translation quality of the compiler, etc.

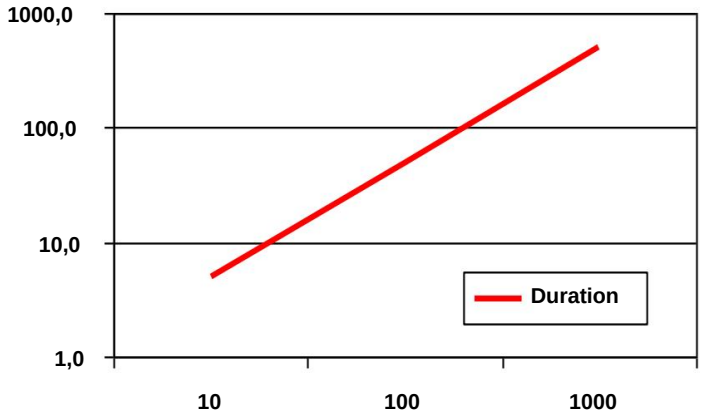
Sum example: simple approach

```
int sum(int n) int s
    = 0; int i = 1;

    while(i <= n) { s = s +
        i; i = i + 1;}

    return s;
}
```

Time in msec

T1 ÿ 0.1
T2 ÿ 0.1
T3 ÿ 0.3
T4 ÿ 0.1
T5 ÿ 0.1
T6 ÿ 0.1

Measured times in program



Calculation of the term

fsum(n) =T1+T2+n*(T3+T4+T5)+T3+T6

fsum(10) = 0.1+0.1+10*(0.3+0.1+0.1)+0.3+0.1 = 5.6

fsum(100) = 0.1+0.1+100*(0.3+0.1+0.1)+0.3+0.1 = 50.6

fsum(1000) = 0.1+0.1+1000*(0.3+0.1+0.1)+0.3+0.1 = 500.6

fsum(n) = 0.6 + n*(0.5)

Problems of the simple approach

Corresponds to a “try it out” approach Only

possible in certain

areas Specific problem size, data

distribution Special cases

problematic

System-dependent Hardware, processor, ...

Operating systems, compilers, libraries, ...

Load-

dependent question difficult to answer as to whether

“gradual” or “fundamental” improvement can be achieved

Order notation

Basic principle

The exact values of the execution times are uninteresting!

Using the *order notation* , one would like to make statements (see goal) that algorithm A is roughly the same speed as algorithm B.

By describing the running times of A and B using functions f(n) and g(n), this question is reduced to a comparison of these functions.

One considers the *asymptotic growth* of the Execution times of the algorithms as the problem size n increases.

that is, when the problem size approaches infinity

Big - O / W / ÿ - Notation

Big-O notation: A function f(n) is said to be of order O(g(n)) if two constants c0>0 and n0 exist such that f(n) ÿ c0g(n) for all n > n0 .

provides an upper bound for the growth rate of functions fÿO(g)

if f grows at most as fast as g.

zB: 17n2 ÿ O(n2), 17n2 ÿ O(2n)

Big-IN Notation: A function f(n) is said to be of order W(g(n)) if there exist two constants c0>0 and n0 such that f(n) ÿ c0g(n) for all n > n0 .

provides a lower bound for the growth rate of functions fÿO(g) if

f grows at least as fast as g.

zB: 17n2 ÿ W(n2), 2n ÿ W(n ^{2}), n37 ÿ W(n2)

ÿ *-Notation:* The runtime behavior is ÿ(g(n)) if the following applies: f(n)ÿO(g(n)) and f(n)ÿW(g(n)) (exactly describes the runtime behavior)

More notations

Little-o notation: A function f(n) is called of order o(g(n)), if for every c>0 there exists an n0 such that f(n) ÿ cg(n) for all n > n0 .

f is asymptotically negligible compared to g.

Little ÿ notation: A function f(n) is said to be of order ÿ(g(n)), if for every c>0 there exists an n0 such that f(n) ÿ cg(n) for all n > n0 .

f dominates g asymptotically

ÿ *-Notation:* f(n)g(n), if lim ÿ f(n)/g(n) = 1
ÿÿ

f and g are asymptotically equal.

Remarks

Instead of ÿ, the use of = is (unfortunately) also widespread, e.g.: f(x)=O(g(x)). Avoid confusion!

The notations presented are often referred to as Landau notation, Bachmann-Landau notation, or also as Landau symbols, Bachmann-Landau symbols.

In mathematics the definitions are generally formulated somewhat more strictly. The absolute values of the functions f(x) and g(x) are considered. This is not necessary in our context because the functions used for runtime analysis always only return positive values.

There is also another definition for the symbol in mathematics (especially in number theory) that is incompatible with the one used here.

Sum example

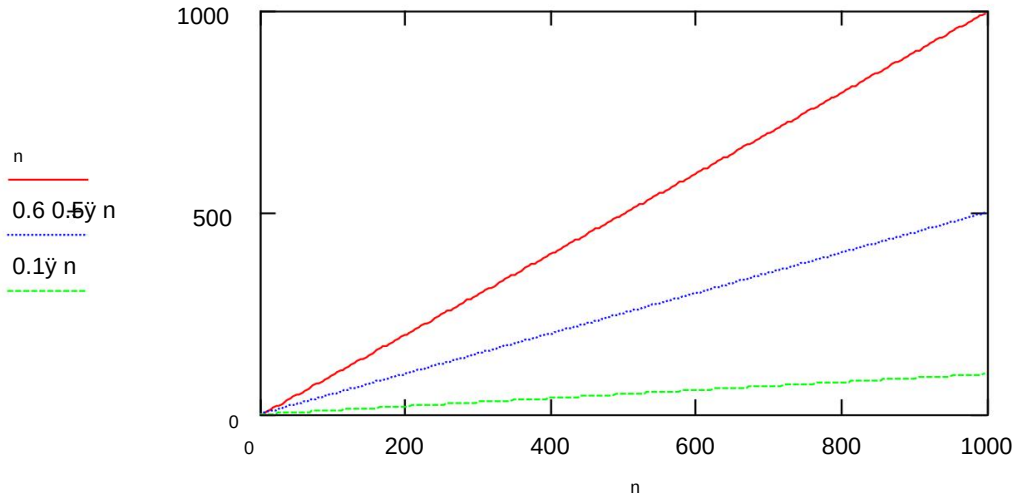
Note: fsum(n) =T1+T2+n*(T3+T4+T5)+T3+T6 Message: T1=0.1,
T2=0.1, T3=0.3, T4=0.1, T5=0.1, T6=0.1

gives fsum(n) =0.6+n*(0.5)

g(n)=n ÿ Lower limit: 0.1*g(n) Upper limit: h(n)=1*g(n)

real values, in which Implementation measured

one of many possible Boundaries



From this it follows regarding the order of the
algorithm fsum(n) ÿ O(n) and further fsum(n) ÿ W(n), ie fsum(n) ÿ ÿ(n).

The statement that can be derived from this for our example is that the running time of the algorithm is directly proportional to the problem size n

Runtime comparison (1)

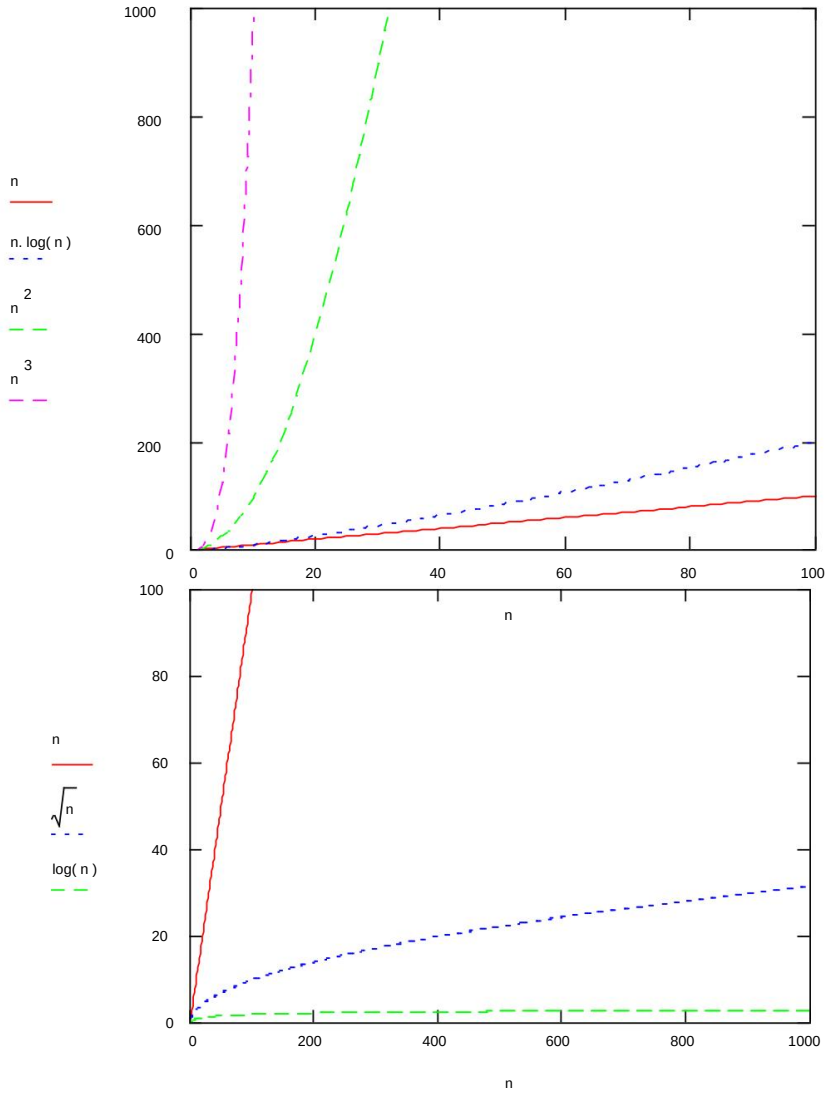
Running time comparison

assumption given

problem size and various

Runtime behavior

Order	Duration
log n	1.2 x 10-5 sec
ÿn	3.2 x 10-4 sec
n	0.1 sec
n log n	1.2 sec
n ÿn	6.5 sec
n^2	2.8 h
n^3	31.7 a
2 n	over 1 century

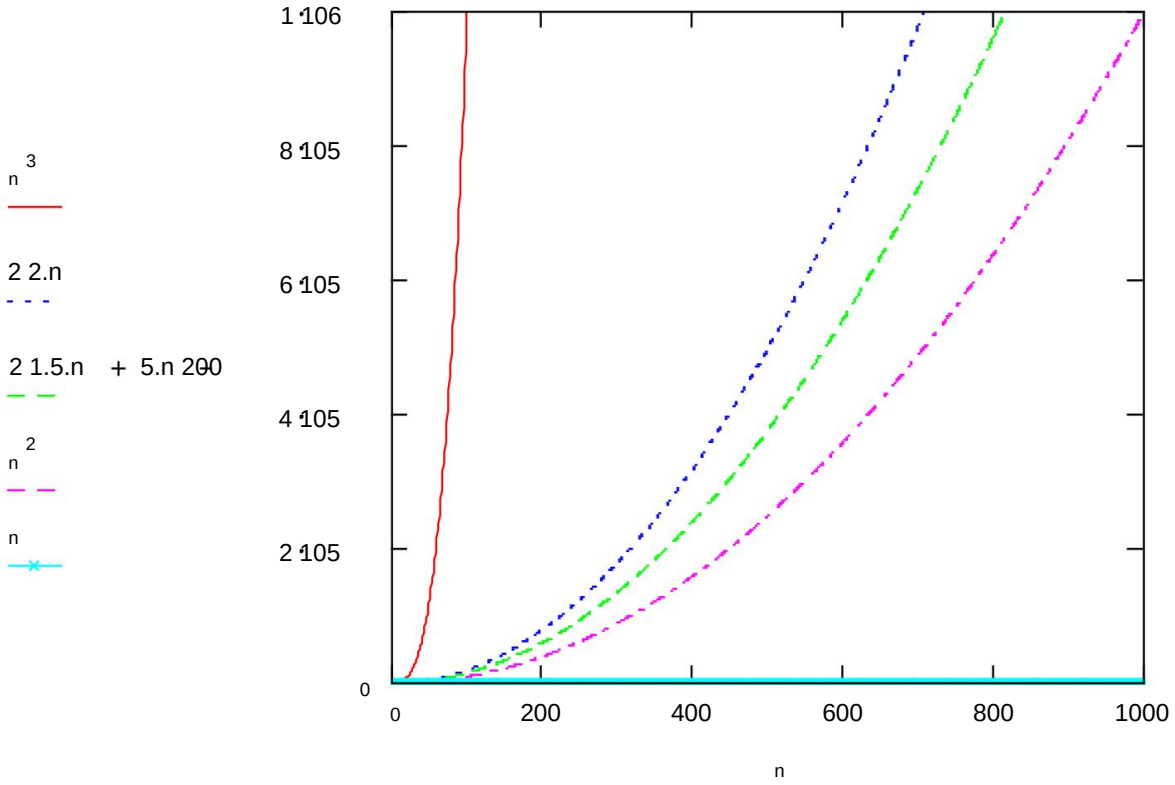


Runtime comparison (2)

Description of the runtime behavior

through upper O(n) and lower bound W(n)

e.g.: T(n) = 1.5n2 + 5n - 200 ÿ O(n2), since n2 ÿ T(n) ÿ 2n2



Principles of mathematical analysis

Definition of the problem size n

Finding a suitable problem parameter that describes the problem size. This characterizes the load (= f(n))

worst-case versus *average-case*

worst-case: behavior in the worst case, maximum expected Expense

average-case: ÿÿbehavior, expected value of the effort

Runtime analysis via O(n) and W(n)

Ignore constant factors

Note only highest powers

often difficult to determine

Practical runtime analysis

Ignore constant factors Constant values are reduced to a factor of 1, ie

T(n) = 13 * n ÿ O(n)

T(n) = log2 (n) ÿ O(log(n)), da logx (n) = loga (n) / loga (x)

Note only highest potency

Ignore all other powers except the highest

T(n) = n2 - n + 1 ÿ O(n2)

Therefore in combination:

T(n) = 13*n 2 + 47*n – 11*log2 (n) + 50000 ÿ O(n2)

Example: Runtime analysis 'Bubblesort'

```
void bubble(Element v[], int n) {
    int i, j; for(i
    = n-1; i >= 1; i--)
        for(j = 1; j <= i; j++) if(v[j-1] > v[j])
            swap(v[j-1], v[j]);

}
```

Problem size n

T1

T2

T3

T4

T5

Assumption

T1 = ... T5 = = 1

Informal approach:

Find equation

$T(n) = T1 + (n-1)*(T2 + (n-1)*(T3+T4 +T5))$

$T(n) = 1+(n-1)*(1+(n-1)*(1+1+1)) = 3n2 -5n+3$

Ignore constant factors

$T(n) = n2 -n+1$

Note highest potency

$T(n) = O(n2) = W(n2) = ÿ(n^2)$

worst-case =

best-case =

average case

Recurrences

Simple recursion

```
int sum(int n) { if(n <=
    0) return 0; if(n == 1) return
    1; return n + sum(n-1);
}
```

T1

T2

T3+T(n-1)

Recurrence equation approach:

T(n) = T1+T2+T3+T(n-1)

T(n) = T(n-1) + 1 T(0)=T(1)=1

T(n) = T(n-2) + 1 + 1

…

T(n) = 1 + … + 1 + 1

n

T(n) = O(n)

Some important recurrences

T(n) = ***T(n-1)+n*** ÿ ÿ
T(n) = n*(n+1)/2 = ***O(n2)***

T(n) = ***T(n/ 2)+1*** ÿT(n) = log2 n = ***O(log n)***

T(n) = ***T(n/ 2)+n*** ÿ T(n) = 2*n = ***O(n)***

T(n) = ***2*T(n/ 2)+n*** ÿ ÿ
T(n) = n*log2 n = ***O(n*log n)***

T(n) = ***2*T(n/ 2)+1*** ÿ T(n) = 2*n-1 = ***O(n)***

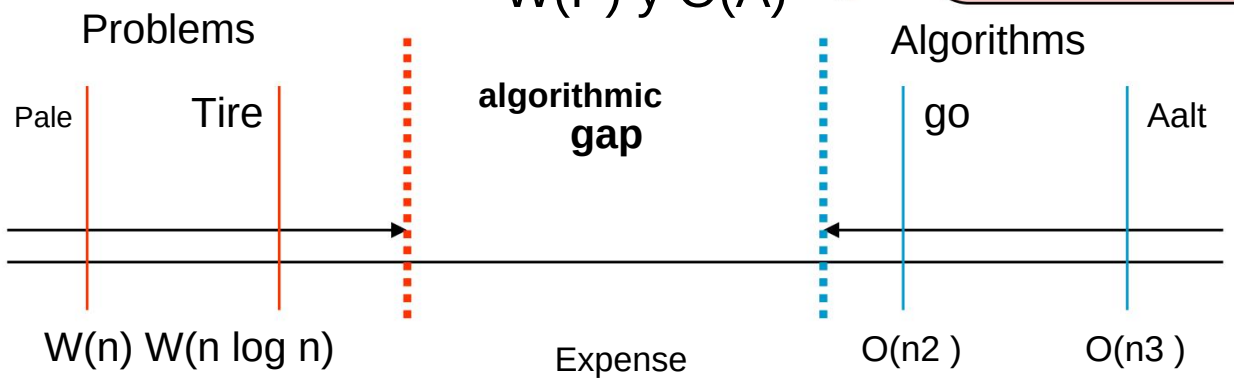
Algorithmic gap

An *algorithmic gap* for a problem exists if the effort required by the known problem-solving algorithms A is greater than the derivable effort required to solve the problem P.

O(A) is used to describe the algorithm

W(P) is used to describe the problem

Goal: algorithmic to close the gap, i.e. W(P) = O(A)



Closed gap

Search in sorted sequence, T(A) = O(log(n)) = W(log(n))

Sort, T(A) = O(n*log(n)) = W(n*log(n))

Open gap

Graph isomorphism, T(Anaiv) from O(|V|!)

1.3.5 Runtime analysis of recursive programs

The runtime analysis of recursive programs is usually non-trivial

The runtime behavior of a recursive program can be described by a recurrence

Example: merge sort

$$T(n) = \begin{cases} \ddot{y}(1) \text{ if } n = 1 \\ 2T(n/2) + \ddot{y}(n) \text{ if } n > 1 \end{cases}$$

where the following solution was given

$$T(n) = \ddot{y}(n \log n)$$

Solutions

There are different approaches to solving the problem, such as:

Substitutionsmethode

- Guessing an asymptotic limit and proving that limit by Induction

Iterationsmethode

- Converting the recurrence into a sum and applying Techniques for calculating limit values of sums

Mastermethode

- Provides bounds for recurrences of the form
  - T(n) = aT(n/b) + f(n), where a ÿ 1, b > 1 and f(n) is a given function

Master Theorem

The *Master Theorem* provides a “recipe” for determining the runtime behavior

Simplified form (general version Cormen et al.)

Let a ÿ 1, b > 1 and c ÿ 0 be constants

If T(n) is defined by aT(n/b) + ÿ(nc), where
n/b is either ÿn/bÿ or ÿn/bÿ,

Then T(n) has the following asymptotic limit Case 1: if c < logba (ie bc <
a) then T(n) = ÿ(nlogba)

Case 2: if c = logba (ie bc = a) then T(n) = ÿ(nc log n)

Case 3: if c > logba (ie b c > a) then T(n) = ÿ(n c)

Example

Binary searching

```
int bs (int x, int z[], int l, int r) {
    if (l > r) // threshold, no element present
        return -1; // 0 verboten, da gültiger Indexwert! else { int m = (l + r) / 2; //

    gefunden! if (x == z[m]) return
        z[m]) return bs(x, z,          m; else if (x <
            l, m-1); else //
        x > z[m] return bs(x, z,
            m+1, r);

    }
}
```

Case 1: if c < logba then T(n) = ÿ(n logba)
Case 2: if c = logba then T(n) = ÿ(n c log n)
Case 3: if c > logba then T(n) = ÿ(n c)

Running time: T(n) = T(n/2) + 1 = T(n/2) + ÿ(1) a = 1, b = 2, c = 0

Case 2, since c = logba ÿ 0 = log21, gives T(n) = ÿ(log n)

Further examples

Case 1: if c < logba then T(n) = ÿ(n logba)
Case 2: if c = logba then T(n) = ÿ(nc log n)
Case 3: if c > logba then T(n) = ÿ(nc)

Example: merge sort

T(n) = 2T(n/2) + n = 2T(n/2) + ÿ(n) a = 2, b = 2, c = 1

Case 2, since c = logba ÿ 1 = log22, gives T(n) = ÿ(n log n)

Example: T(n) = 4T(n/2) + n = 4T(n/2) + ÿ(n)

a = 4, b = 2, c = 1

Case 1, since c < logba ÿ 1 < log24, gives T(n) = ÿ(nlog24) = ÿ(n2)

Example: T(n) = T(n/2) + n = T(n/2) + ÿ(n) a = 1, b = 2, c = 1

Case 3, since c > logba ÿ 1 > log21, gives T(n) = ÿ(n1) = ÿ(n)

1.3.6 Structural complexity

Evaluative statement about the structural structure of the program and the program parts among themselves, ie Evaluation of programming style

interne Attribute

Generally assumed that programming style with the expected

Software maintenance costs are correlated.

Statements about the quality of a software product (external Attribute)

Susceptibility to errors

Maintenance effort

Cost

Assumption:
internal attributes are correlated with external attributes!

These

complex program ÿ high costs clear

program ÿ lower costs

„When you can measure what you are speaking about and express it in numbers you know something about it, but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.“

Lord Kelvin

Software metrics

The goal is to find metrics (measures) that determine the structure, structure,

Have the style of a *module* (program piece) evaluated and compared .

Requirements

validity

Actually measures what it claims to measure

simplicity

Results are easy to understand and interpret

sensitivity

Reacts adequately to different manifestations

robustness

Does not react to characteristics that are uninteresting in the context

Measuring structural complexity

The focus of the measurement is the software module. Definition difficult - can be a function, method, class, etc. The *intra-modular complexity* describes the complexity of a individual SW module

- *Complexity module* (internal)
  - LOC, Line-of-codes (simpel)
  - NCSS, non commenting source statements
  - McCabe (cyclomatic complexity), …
- *Cohesion* (external)
  - Henry-Kafura metric (information flow), …

Inter-modular complexity describes complexity between modules - *coupling*

- Fenton and Melton,...

Connection coupling and cohesion

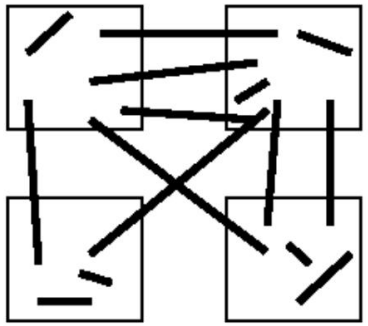
coupling

Measures complexity of relationships between modules

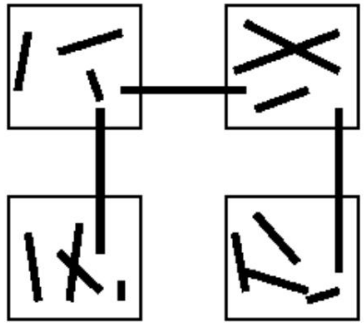
cohesion

Measures information flow to and from the outside depending on the Module size

Mostly relationship between coupling and cohesion



Strong clutch
Weak cohesion



Weak clutch
Strong cohesion

(usually the goal of good software Development)

1.3.6.1 Intra-modular complexity – McCabe's metric

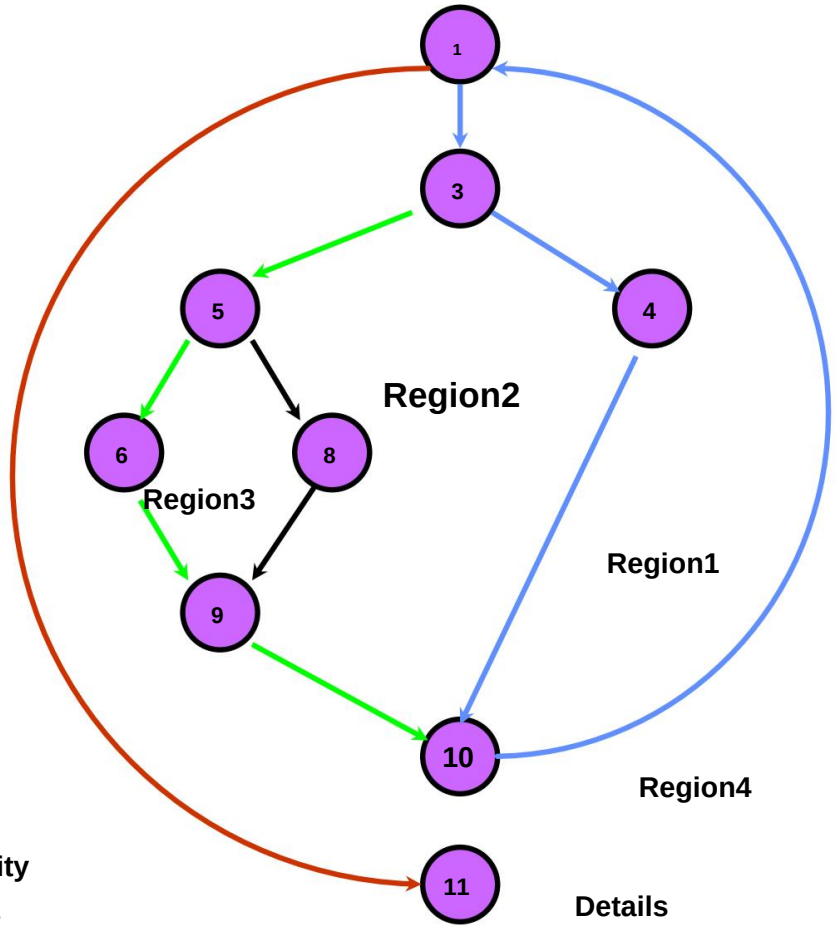
McCabe's metric is a measure for assessing the module complexity

- Based on *cyclomatic complexity* V = the number of independent paths in a program graph
- With an independent path, at least one 'new' edge is taken in the program flowchart.
- Experience values for cyclomatic complexity

V(G)	assessment in normal cases
< 5	simply
5-10	normal
> 10	complex, should be restructured
> 20	difficult to understand, probably incorrect

Program example

```
0: void foo (int a) {
1:          while (a < limit) {
2:              doit1();
3:              if(check1(a))
4:                  doit2();
5:              else { if(check2(a))
6:                      doit3();
7:                  else
8:                      doit4();
9:              } // than else
10:         } // end while
11: } // end foo
```



Base quantity

Path 1: 1.11

Path 2: 1, 3,4,10,1,11

Path 3: 1,3,5,6,9,10,1,11

Path 4: 1,3,5,8,9,10,1,11

Details

Sides = 11

Node = 9

Condition node = 3

Cyclomatic complexity

Multiple calculation options

1. The number of regions in the program graph G
2. V(G) = E - N + 2 (E = number of edges, N = number of nodes)
3. V(G) = P + 1 (P = number of binary condition nodes)

Cyclomatic complexity of the example 1. Regions = 4 2. V(G) = 11 - 9 + 2 = 4 3. V(G) = 3 + 1 = 4 In our example the magic number is 4, i.e. “simple” program (Metric McCabe < 5)

1.3.6.2 Intra-modular complexity – Henry-Kafura Metric

Measure for determining cohesion

- Describes the functional strength of the module; to what degree the
  - Module components perform the same task

The *Henry-Kafura Metric* (Sallie Henry and Dennis Kafura)

- is based on the relationship between module complexity and
- Connection complexity between modules

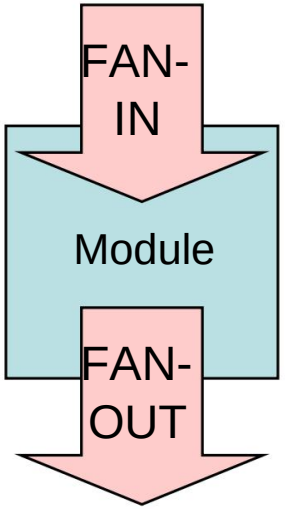
Measure of module complexity

- PLACE
- NCSS
- McCabe

Connection complexity

Quantification of reading and modifying access module to the environment

Counts the data flows between modules



FAN-INm: “Number of modules that use m”
more precisely: number of data flows that terminate in module m + number of data structures from which module m reads data

FAN-OUTm: “Number of modules that m uses” more precisely: number of data flows that originate from module m + number of data structures that module m changes

Henry-Kafura Formula

$Cim * (FAN\text{-}INm * FAN\text{-}OUTm)^2$

Cim … module complexity (e.g. LOC, McCabe, …)

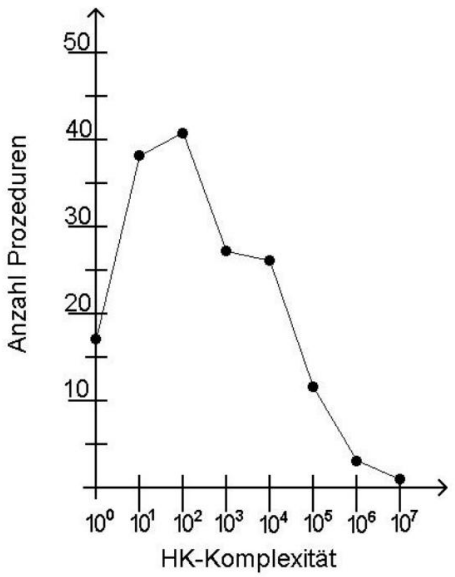
Application

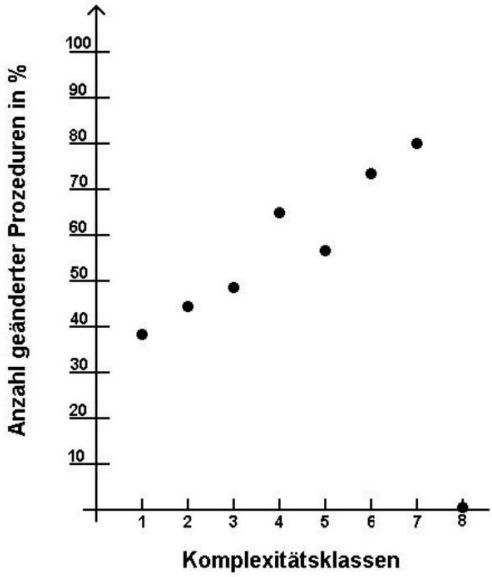
Applied to Unix code by Henry and Kafura

- Assumption: connection between complexity and
  - Frequency of changes (error correction) of a procedure
- 165 procedures investigated, patch information from newsgroups
- Acceptance confirmed:
  - Take changes with you
  - Complexity class too

Weaknesses of HK:

- Depends on the data flow
  - one FAN = 0, total 0
    even at high
  - Module complexity
- Reusability is “punished” by a high FAN
  - value





1.3.6.3 Inter-modular complexity – Fenton and Melton metric

Fenton and Melton metric is a measure for determining coupling, i.e. the independence between modules

Global coupling of a program is derived from the coupling values between all possible pairs of modules

Coupling types

Binary relations defined on pairs of modules x, y

Sorted according to the degree of “undesirability”.

0. No coupling: no communication between x and y
1. Data coupling: Communication via parameters (data)
2. Stamp coupling: accept the same record type as a parameter
3. Control coupling: Communication via parameters (control)
4. Common coupling: Access to the same global data structure
5. Content coupling: x directly accesses the internal structure of y (changes data, instructions)

Coupling calculation

Fenton dimension for the coupling between 2 modules

c(x,y) = i + n/ (n+1)

i is the worst coupling type between module x and yn

is the number of “couplings” of type i

Measure C(S) for the global coupling of a system S consisting of n modules D1 , ..., Dn *C(S)*

= median of the set of coupling values of all module pairs

What do we take with us?

Algorithmenparadigmen

- Greedy, Divide and Conquer, Dynamic Programming

Analysis and evaluation of algorithms

- Effectiveness, correctness, termination

Runtime complexity

- Order notation, algorithmic gap, master theorem

Structural complexity

- McCabe, Henry-Kafura, Fenton