Lecture 5

Analysis of algorithms

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
September 16th, 2024

IDA, Linköping University

5.1

Content

Contents

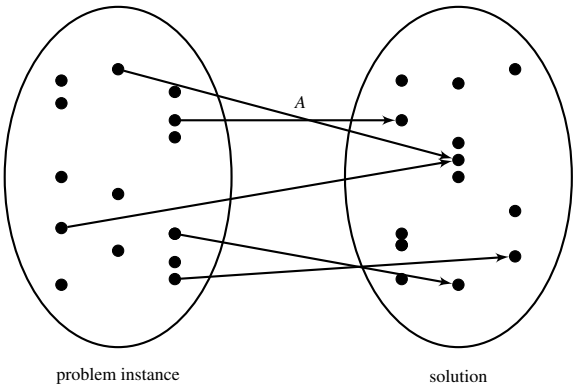
5.2

1 Introduction

Definition 1 (Algorithm)**.** An *algorithm* is a finite description of how to solve a problem step-by-step.

An algorithm usually takes *input data* describing a *problem instance* and produces *output data* which describes a *solution*.

An algorithm can be seen as a function A : PROBLEM INSTANCE $\rightarrow$ SOLUTION.



5.3

Algorithm

Correctness
The output is in agreement with a specification.

Analysis of algorithm
time and memory consumption, scalability, efficiency, worst case, best case, average case, amortised analysis

Algorithmic paradigms
common problem solving strategies, eg divide and conquer, dynamic programming, greedy algorithms, ...

Pseudo code
algorithms are usually described in pseudo code, a mixture of natural language and programming constructs

5.4

Example

Implementation of ADTn Lexicon with ordered data structure: array/table.

function LOOKUPTABLE(**table** $T[1,\ldots,n]$,**key** k)
  for i **from 1 to** n **do**
    if $T[i] = k$ **then return** true
    if $T[i] > k$ **then return** false
  return false

5.5

An other implementation

ADTn Lexikon using an ordered linked list.

function LISTSEARCH(**pointer** *List*,**key** k)
  $P \leftarrow List$
  while $P \neq$ null **and** KEY$(P) \leq k$ **DO**
    IF KEY$(P) = k$ **THEN return** true
    $P \leftarrow$ NEXT(P)
  return false

5.6

Bubble sort

function BUBBLESORT(**table** $A[1,\ldots,n]$)
  repeat
    $swapped \leftarrow false$
    for i **from 2 to** n **do**
      if $A[i-1] > A[i]$ **then**
        $swap(A[i-1], A[i])$
        $swapped \leftarrow true$
  until *not swapped*

5.7

Discussion

An ADT tells *what* can be done: a set of operations on data.

To describe *how* this is done:

- we choose a data structure (a memory representation)
- we construct algorithm that operates on the ADT

The same ADT:

- can be implemented with many different data structures
- can be used with many different algorithm

The choice of algorithms depends on the selected data structure. What is the *most effective* solution. What is *effectiveness*?

5.8

1.1 Analysis of algorithms

Analysis of algorithms

What is analysis?

- Correctness (not in this course)
- Termination (not in this course)
- Efficiency, resources, complexity

Time complexity — how long it takes an algorithm in the worst case?

- as a function of what?
- what is a time step?

Memory complexity — how much memory is required in the worst case?

- as a function of what?
- how is it measured?
- remember that code and function calls also takes memory

5.9

How can you compare different effectiveness

- **Study execution time (or memory consumption) in function of the size of input data.**
- When can we say that two algorithms have "similar effectiveness"?
- When can we say that an algorithm is better than an other?

Comparison between some elementary functions

n	$\log_2 n$	n	$n\log_2 n$	n^2	2^n
2	1	2	2	4	4
16	4	16	64	256	$6.5 \cdot 10^4$
64	6	64	384	4096	$1.84 \cdot 10^{19}$

$1.84 \cdot 10^{19}$ nanoseconds $= 212963$ days $= 583.5$ years

5.10

Simplify calculations

> *" It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. "* — Alan Turing

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

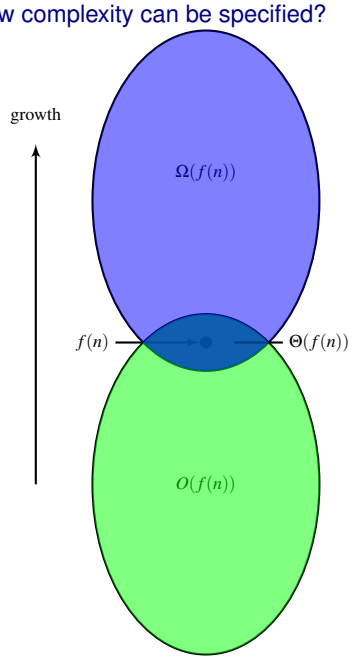
[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

5.11

How complexity can be specified?



- How does the complexity grow with the size n of input data?
- Asymptotic complexity — what happens when n grows to infinity?
- Much easier if we ignore constant factors

- $O(f(n))$ – grows at most as fast as $f(n)$
- $\Omega(f(n))$ – growth at least as fast as $f(n)$
- $\Theta(f(n))$ – grows as fast as $f(n)$

5.12

Ordo-notation

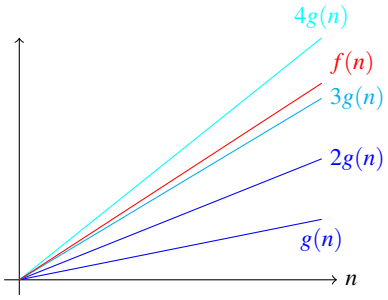
f, g: grow from $\mathbb{N}$ to $\mathbb{R}^+$

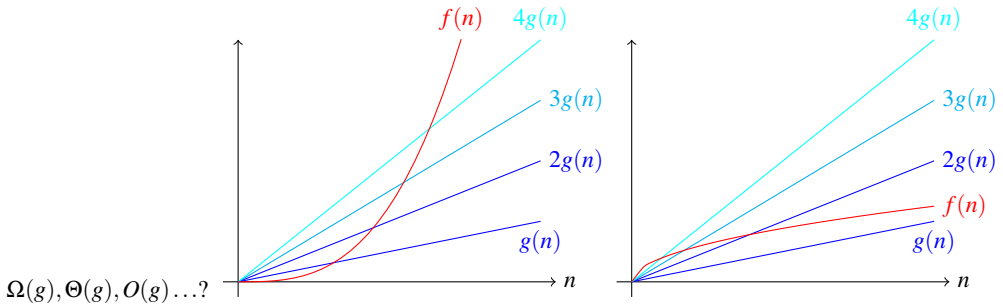
- $f \in O(g)$ if and only if it exists $c > 0, n_0 > 0$ such as $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ Intuition: ignoring the constant factor, f does not grow faster than g
- $f \in \Omega(g)$ if and only if it exists $c > 0, n_0 > 0$ such as $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$ Intuition: ignoring the constant factor, f grows at least as fast as g
- $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$ Intuition: ignoring the constant factor, f and g have similar growth

Note: Ω is the opposite of O, i.e. $f \in \Omega(g)$ if and only if $g \in O(f)$.

5.13

Comparison of growth





5.14

"Rules" for ordo-notation

- If $f(n)$ is a polynomial of degree d it means $f(n) \in O(n^d)$, i.e.
  - Ignore the term of lower degree
  - Ignore the constant terms
- Use the smallest class of equivalence – say $2n \in O(n)$ instead of $2n \in O(n^2)$
- Use the minimal representation – say $3n+5 \in O(n)$ instead of $3n+5 \in O(3n)$

5.15

One way to compare growth

f, g: growth functions from $\mathbb{N}$ to $\mathbb{R}^+$

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- $f \in O(g)$ if $L = 0$
- $f \in \Omega(g)$ if $L = \infty$
- $f \in \Theta(g)$ if $0 < L < \infty$

5.16

1.2 Upper and lower limits

Analysis of a problem

Call it a problem complexity!

Upper limit:

- Given an algorithm that solve he problem.
- The algorithm complexity is a *upper limit* for the problem complexity.

Lower limit:

- Often difficult to give.
- The properties of the problem can be used.
  - if it needs to check all data inputs $\Rightarrow \Omega(n)$
  - if it needs to provide all output
  - decision tree — many different cases must be considered

5.17

1.3 A calculation model

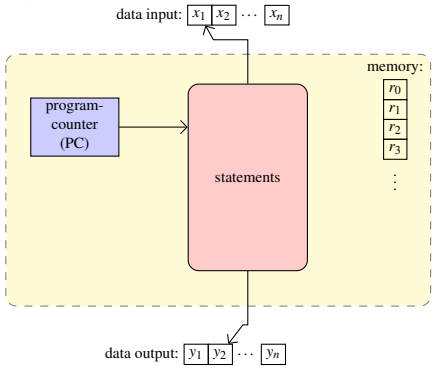
A grand father

John von Neumann (1903 – 1957)

John von Neumann has made important contributions to pure mathematics, quantum physics, game theory, economic sciences and computer science. He is a computer science pioneer with contributions to the development of logical design and the theory of cellular machines. He developed the computer architecture that bears his name and which is the basis for nearly all commercially available PCs.

5.18

RAM (Random Access Machine)



The program consists of statements performed sequentially (not parallel). Each statement can only read and influence a constant number of memory locations. Only one symbol can be read/written at a time. Each statement takes constant time.

Because of the robustness of the O-notation, we can ignore constants.

5.19

Cost dimensions

Unit cost

- each operation takes one unit of time
- each variable takes up one memory unit

calculation: RAM

Bit cost

- each bit operation take one unit of time
- each bit takes up one memory unit

calculation: RAM with limited word length, Turing machine

Example: Addition of two n-bit integer

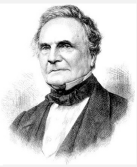
- time in $O(1)$ with unit cost
- time in $O(n)$ with bit cost

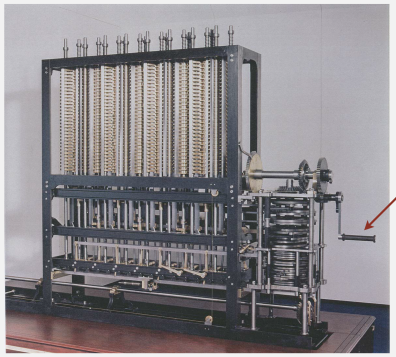
5.20

2 Execution

Execution

> *" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? "* — *Charles Babbage (1864)*





How many turns are needed?

Analytic Engine

5.21

Interesting operators

A programmer who must provide an efficient solution

The student may play one of those roles one day

A customer who wants an efficient solution to a problem

A theorist who wants to understand

5.22

Reasons for analysing algorithms

- Predicting performance
- Comparing algorithms
- Provide guarantees
- Understand theoretical basis

Primarily practical reason
Avoid bugs that affect performance

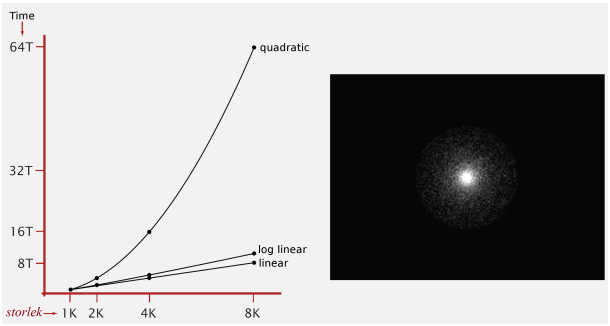
The client gets poor performance because the programmer does not understand the performance characteristics of its program.

5.23

A pair of algorithm success

Simulation of N bodies

- Simulation of the gravitational force between N bodies
- Brute force: N^2 calculations steps
- Barnes-Hut-algorithm: $N \log N$ calculations steps, *allows new research*

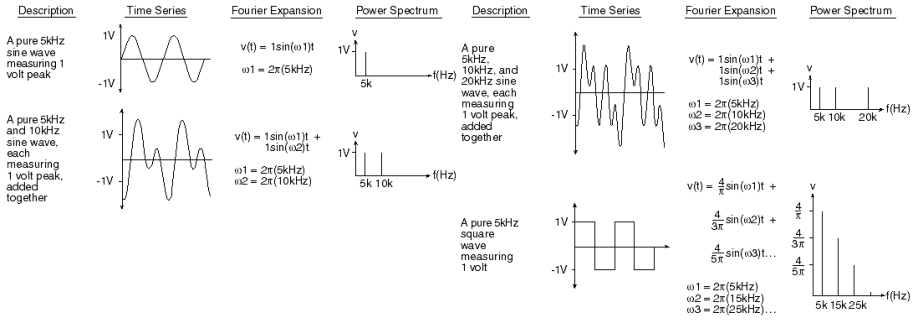


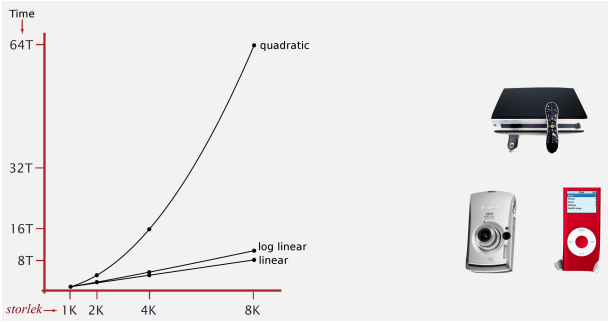
5.24

A pair of algorithm success

Discrete Fourier transform

- Break down the waveform consisting of N samples into periodic components
- Used for: DVD, JPEG, MRI, astrophysics, ...
- Brute force: N^2 calculation steps
- FFT-algorithm: $N \log N$ calculation steps, *allows new technologies*

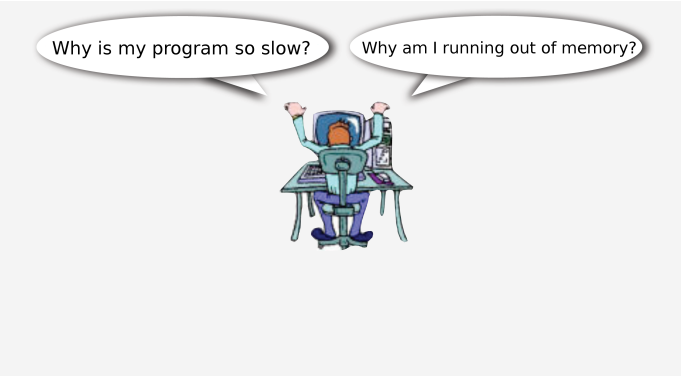




5.25

Challenges

Question: Will my program manipulate large amount of data (big data)?



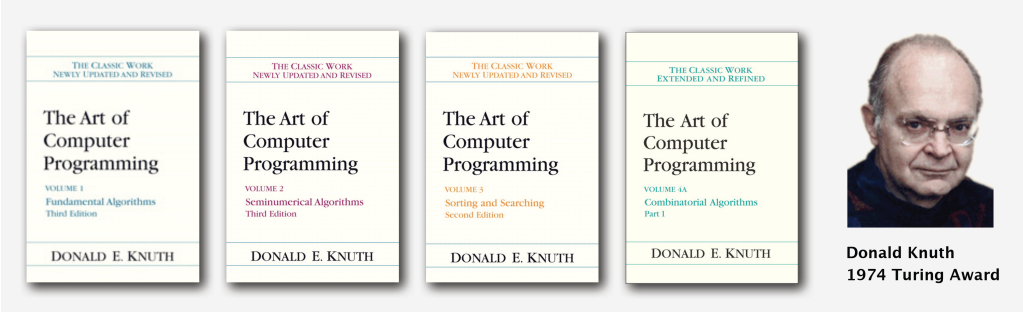
Insight: [Knuth 1970-years] used *scientific methodology* to Understand performance

5.26

Mathematical models for execution

Total execution time: sum of costs $\times$ frequency of all operations

- Needs to analyse the program to find the set of operations
- Cost depends on the machine and compiler
- Frequency depends on the input data



In theory we have access to good mathematical models

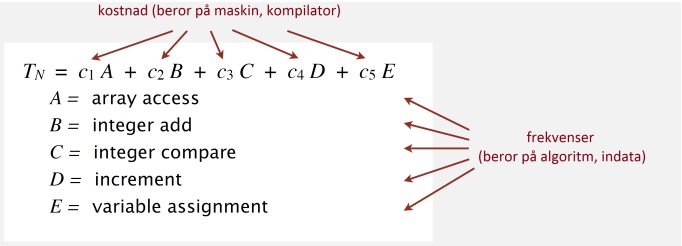
5.27

Mathematical models for execution

In theory we have access to good mathematical models

In practice

- The formulas can be complicated
- Advanced mathematics may be required
- Accurate models are best left to scientists



Conclusion: we use approximate methods for this course: $T(N) \in \mathcal{O}(f(N))$

5.28

Various techniques

The question is what is the growth rate of . . .

- memory usage
- execution time

Situation analysis:

- worst case, best case, expected case
- amortised: a sequence of call to the algorithm

Technique:

- algebraically (count iterations)
- estimate recursion relation (recursive algorithm)
- analyse probabilistic (figure the behaviour of the average error)

5.29

Analysis of algorithms — how to do it?

An algorithm should (in theory) work with arbitrary size.

Describe resource consumption (time/memory) as a *not decreasing function* of *input size*.

Focus on the behaviour for the *worst case*!

Ignore the *constant factor*

- analysis should be machine independent;
- more powerful CPU $\Rightarrow$ speedup with a constant factor.

Study *scalability/asymptotic behaviour* for large problem size: ignore lower terms and focus on the dominant term.

5.30

2.1 Iterative algorithm

Execution of iterative algorithm

- Elementary operations: limited by a constant time
  - assign a value to a variable
  - call a function/method/procedure
  - performing arithmetic operation
  - compare two numbers
  - index an array
  - follow an object reference
  - return from a function/method/procedure
- Sequence of operations: sum of components
- Loop (for. . . och while. . .): Time (in the worst case) of the condition plus the body times the number of iterations N (in the worst case): $t_{while} = N \cdot (t_{cond} + t_{body})$
- Conditions (if. . . then. . . else): Time (in the worst case) for evaluation the condition plus the maximum time (in the worst case) of the two statements $t_{if} = t_{cond} + \max(t_{then}, t_{else})$

5.31

Analysis of algorithms — how to do in practice?

```
1: function FIND(A[1,...,n],t)
2:     for i ← 1 to n do
3:         if A[i] == t then
4:             return true
5:     return false
```

- What is the worst case instance like?
- How much time is spent in the worst case?
- What is " time complexity " of this function?

5.32

Example: Two loops

```
1: function FIND(A[1,...,n],B[1,...,n],t)
2:     for i ← 1 to n do
3:         if A[i] == t then
4:             return true
5:     for i ← 1 to n do
6:         if B[i] == t then
7:             return true
8:     return false
```

- What is the worst case instance like?
- How much time is spent in the worst case?
- What is " time complexity " of this function?

5.33

Example: Two nested loops

```
1: function COMMON(A[1,...,n],B[1,...,n])
2:     for i ← 1 to n do
3:         for j ← 1 to n do
4:             if A[i] == B[j] then
5:                 return true
6:     return false
```

- What is the worst case instance like?
- How much time is spent in the worst case?
- What is " time complexity " of this function?

5.34

Example: Two other nested loops

```
1: function DUPLICATE(A[1,...,n])
2:     for i ← 1 to n do
3:         for j ← i+1 to n do
4:             if A[i] == A[j] then
5:                 return true
6:     return false
```

- What is the worst case instance like?
- How much time is spent in the worst case?
- What is " time complexity " of this function?

5.35

Example: Binary Search

- Trivial to implement?
  - First algorithm published in 1946; first bug free version on 1962.
  - Java-bug in `Arrays.binarysearch()` discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
  }
  return -1;
}
```

Worst time is: $c_1 + \text{maxit} \cdot c_2 + c_3$, where maxit is the maximum number of iterations in the while loop.

If $n = 2^p$, then $maxit = log_2(n)$ and worst time is in $O(log_2(n)) = O(log(n))$. More generally, $2^{\lfloor log_2(n) \rfloor} \leq n \leq 2^{\lceil log_2(n) \rceil}$ and $log_2(n) - 1 \leq maxit \leq log_2(n) + 1$ hence worst case still in $O(log(n))$

Significant difference between $O(log(N))$ and $O(N)$ when N is large: for any practical problem it is crucial that we avoid $O(N)$ searches. Suppose your array contains 2 billion ($2^{30} \leq 2 \cdot 10^9 \leq 2^{31}$) values. Binary search would require at most 32 comparisons!

5.36

Conclusion

- A method to represent complexity of algorithms
- Useful for comparison between two algorithms
- Usually worst case, sometime the most common case is the best case
- Only an indication, does not tell the full picture
- $O(1) \in O(log(n)) \in O(n) \in O(n \cdot log(n)) \in O(n^2 \cdot log(n)) \in O(2^n)$

5.37