Algorithms and Data Structures

Ulf Leser

Once upon a Time ...

- IT company A develops software for insurance company B
  - Volume: ~4M Euros
- B not happy with delivered system; doesn't want to pay
- A and B call a referee to decide whether requirements were fulfilled or not
  - Volume: ~500K Euros
- Job of referee is to understand requirements (~60 pages) and specification (~300 pages), survey software and manuals, judge whether the contract was fulfilled or not

One Issue

This is hardly testable

- Requirement: „Allows for smooth operations in daily routine“

One Issue

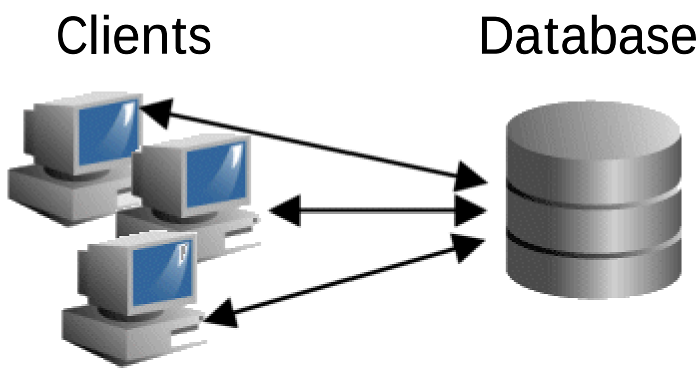
- Requirement: „Allows for smooth operations in daily routine“
- Claim from B
  - I search a specific contract
  - I select a region and a contract type
  - I get a list of all contracts sorted by name in a drop-down box
  - This sometimes takes minutes! A simple drop-down box! This is inacceptable performance for our call centre!

Discussion

- A: We tried and it worked fined
- B: OK, most of the times it works fine, but sometimes it is too slow
- A: We cannot reproduce the error; please be more specific in what you are doing before the problem occurs
- B: Come on, you cannot expect I log all my clicks and take notes on what is happening
- A: Then we conclude that there is no error
- B: Of course there is an error
- A: Please pay as there is no reproducible error
- ...

A Closer Look

- System has classical two-tier architecture



- Upon selecting a region and a contract, a query is constructed and send to the database
- Procedure for “query construction” is used a lot
  – All contracts in a region, ... running out this year, ... by first letter of customer, ... sum of all contract revenues per year, ...
  – “Meta” coding: very complex, hard to understand

Requirement

- Recall

- After retrieving the list of customers, it has to be sorted

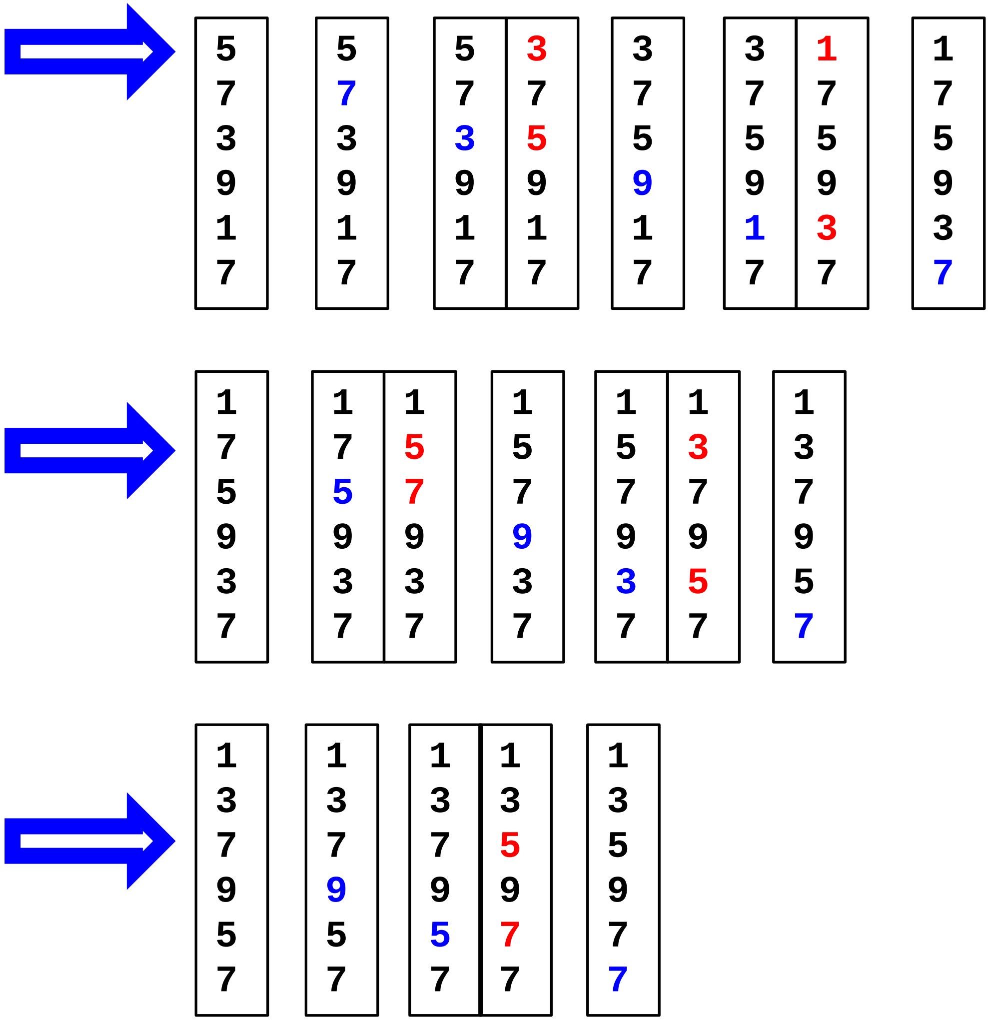
Code used for Sorting the List of Customer Names

```
S: array_of_names;
n := |S|;
for i = 1..n-1 do
  for j = i+1..n do
    if S[i]>S[j] then
      temp := S[j];
      S[i] := S[j];
      S[j] := tmp;
    end if;
  end for;
end for;
```

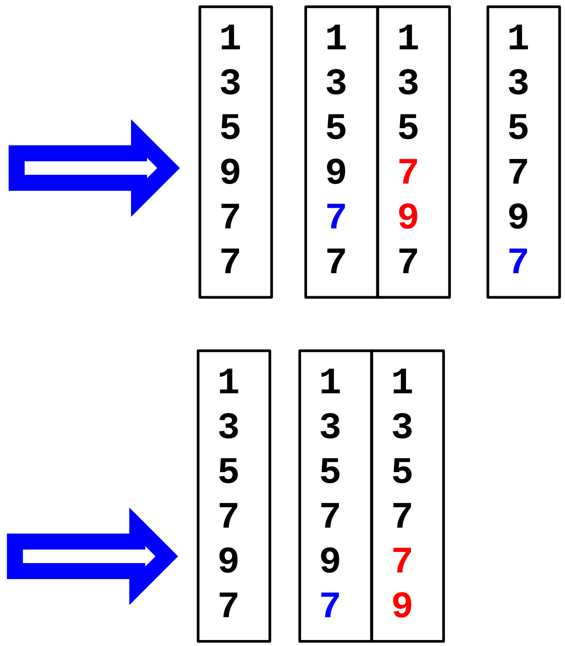
- S: array of Strings, |S|=n
- Sort S alphabetically
  - Take the first string and compare to all others
  - Swap whenever a later string is smaller
  - Repeat for 2nd, 3rd, ...
  - After 1st iteration of outer loop, S[1] contains the smallest string in S
  - After 2nd iteration of outer loop: S[2] contains the 2nd smallest
  - Etc.

Example

```
S: array_of_names;
n := |S|;
for i = 1..n-1 do
  for j = i+1..n do
    if S[i]>S[j] then
      temp := S[j];
      S[i] := S[j];
      S[j] := tmp;
    end if;
  end for;
end for;
```



Example continued



- Seems to work
- This algorithm is called “selection sort”
  - Select smallest element and move to front, select second-smallest and move to 2nd position, ...

Analysis

- How long will it take (depending on n)?
- Which parts of the program take time?

1. No time
2. Not sure ... maybe n additions
3. n-1 times one assignment
    4. n-i+1 times one assignment
        5. One comparison
            6. One assignment
            7. One assignment
            8. One assignment
        9. No time
    10. One increment (j+1); one test
11. One increment (i+1); one test

```
1.  S: array_of_names;
2.  n := |S|;
3.  for i = 1..n-1 do
4.    for j = i+1..n do
5.      if S[i]>S[j] then
6.        temp := S[j];
7.        S[i] := S[j];
8.        S[j] := tmp;
9.      end if;
10.   end for;
11. end for;
```

Slightly More Abstract

- Assume one assignment/test costs c, one addition d
- Which parts of the program take time?

1. 0
2. n*d+c
3. (n-1)*c
    4. n-i+1 (hmmm ...)
        5. c
            6. c
            7. c
            8. c
        9. 0
    10. c+d
11. c+d

```
1.  S: array_of_names;
2.  n := |S|;
3.  for i = 1..n-1 do
4.    for j = i+1..n do
5.      if S[i]>S[j] then
6.        temp := S[j];
7.        S[i] := S[j];
8.        S[j] := tmp
9.      end if;
10.   end for;
11. end for;
```

Slightly More Compact

- Assume one assignment/test costs c, one addition d
- Which parts of the program take time?
  - Let's be pessimistic: We always swap
    - How would the list have to look like in first place?
  - n*d+c
  - (n-1)*c* (
    - n-i+1* (
      - 4*c
    - c+d) +
  - c+d)

```
S: array_of_names;
n := |S|;
for i = 1..n-1 do
  for j = i+1..n do
    if S[i]>S[j] then
      temp := S[j];
      S[i] := S[j];
      S[j] := tmp;
    end if;
  end for;
end for;
```

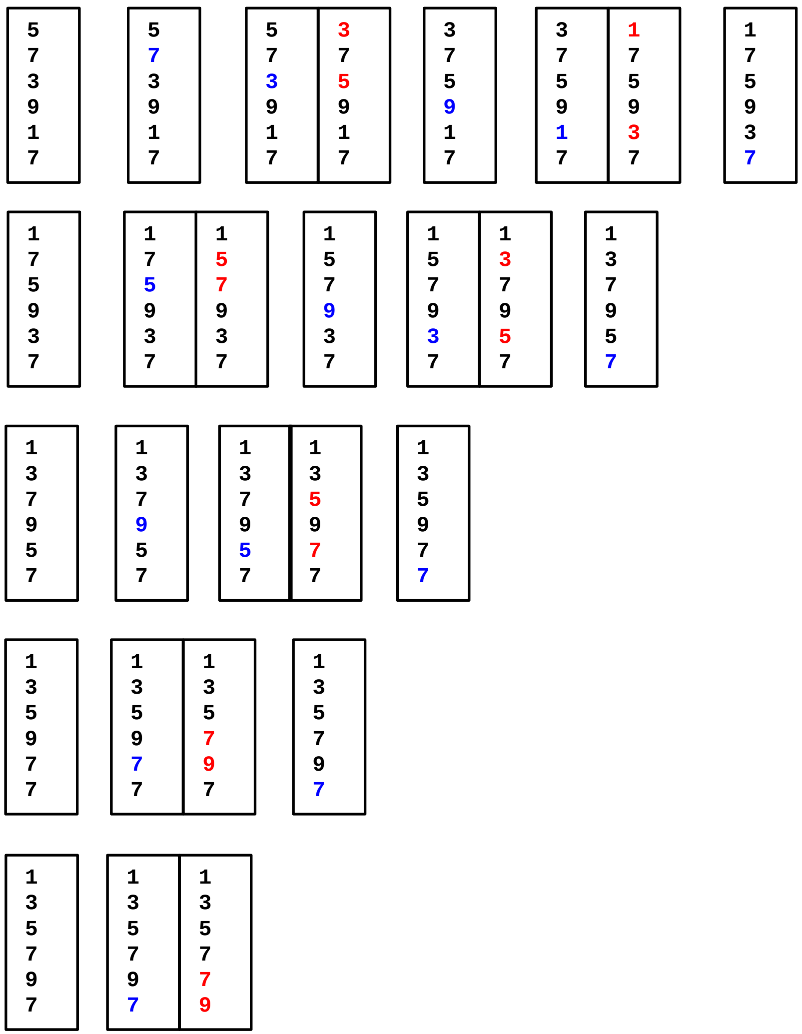
This is not yet clear

Even More Compact

- Assume one assignment/test costs c, one addition d
- Which parts of the program take time?
  - We have some cost outside the loop (out_loops)
  - And some cost inside the loop (in_loops)
  - How often do we need to perform in_loops?
  - $n*d+c+(n-1)*c*(n-i+1*(6*c+2*d))=$ out_loops+(n-1)*c*?*in_loops

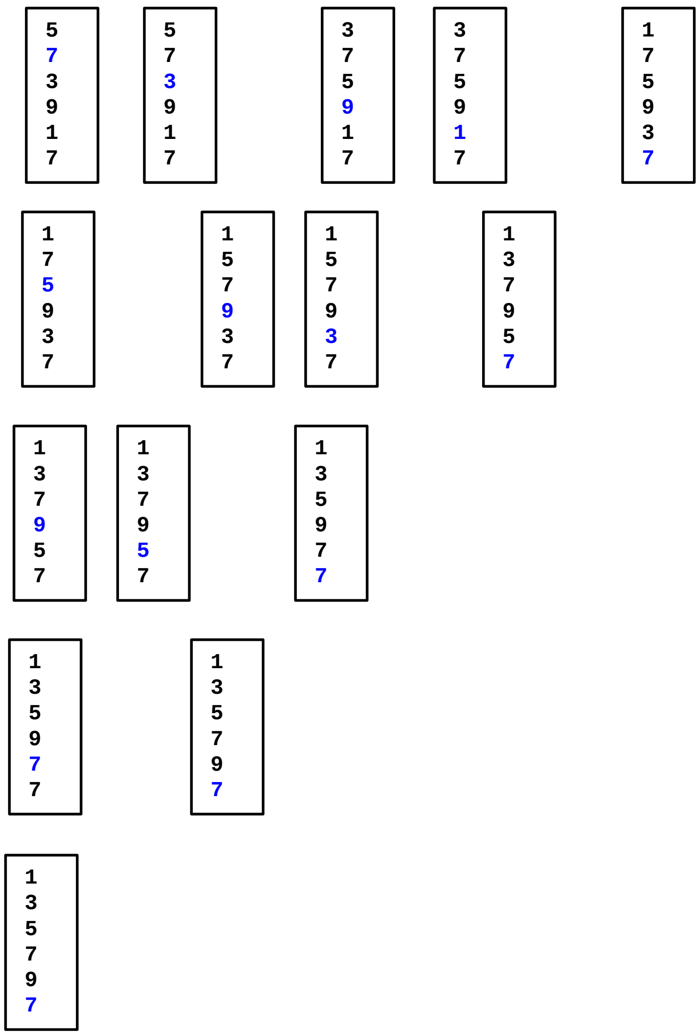
```
S: array_of_names;
n := |S|;
for i = 1..n-1 do
  for j = i+1..n do
    if S[i]>S[j] then
      temp := S[j];
      S[i] := S[j];
      S[j] := tmp;
    end if;
  end for;
end for;
```

Observations



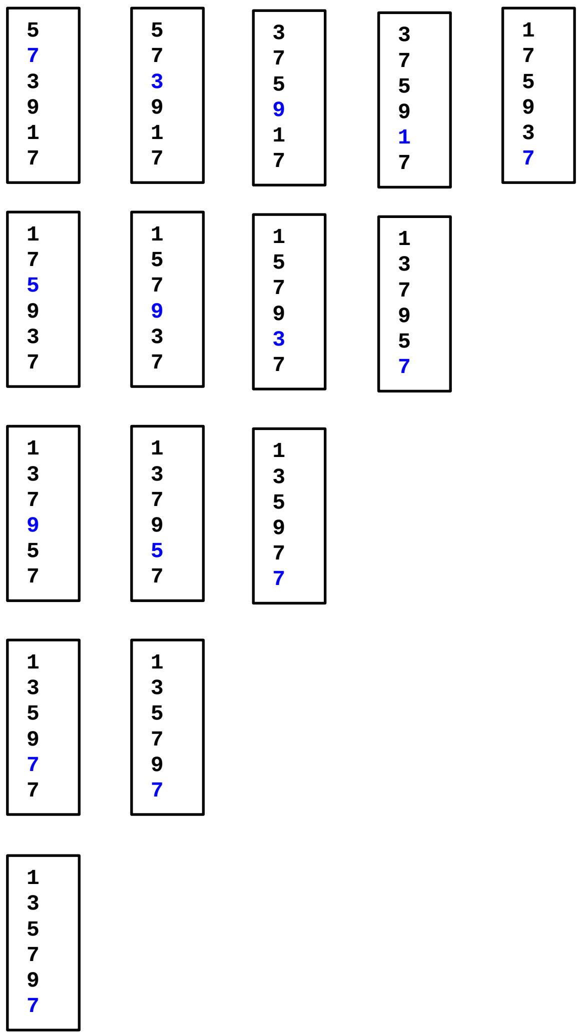
- The number of comparisons is independent from the number of swaps
  - We always compare, but we do not always swap

Observations



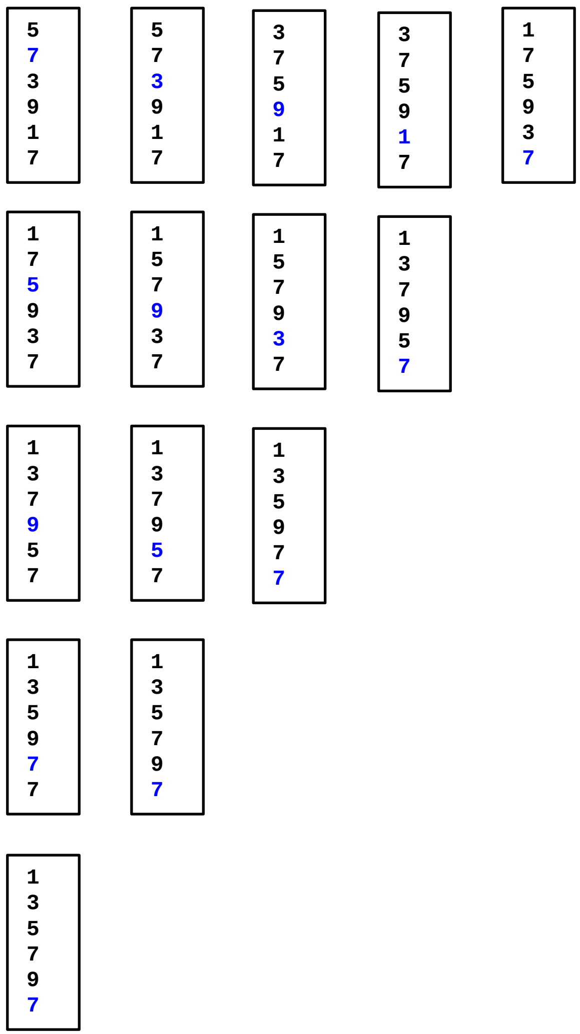
- The number of comparisons is independent from the number of swaps
  - We always compare, but we do not always swap
- How many comparisons do we perform in total?

Observations



- The number of comparisons is independent from the number of swaps
  - We always compare, but we do not always swap
- How many comparisons do we perform in total?

Observations



- The number of comparisons is independent from the number of swaps
- First string is compared to n-1 other strings
- Second is compared to n-2
- Third is compared to n-3
- ...
- n-1'th is compared to 1

Together

$$(n-1)+(n-2)+(n-3)+\ldots+1=\sum_{1}^{n-1} i=\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

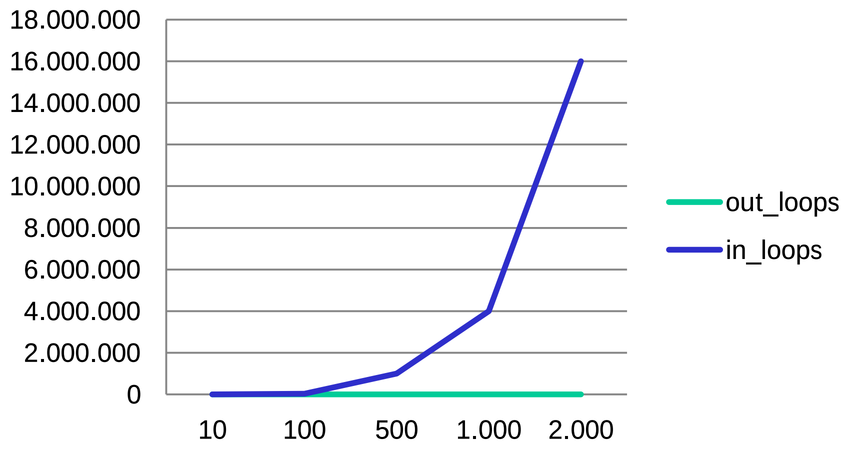
- This leads to the following total cost estimation

$$out\_loops+(n^2-n)*in\_loops/2$$

- Let's assume c=d=1

$$n+1+(n^2-n)*8/2$$

	out_loops	in_loops	total
10	**11**	**360**	**371**
100	**101**	**39.600**	**39.701**
500	**501**	**998.000**	**998.501**
1.000	**1.001**	**3.996.000**	**3.997.001**
2.000	**2.001**	**15.992.000**	**15.994.001**

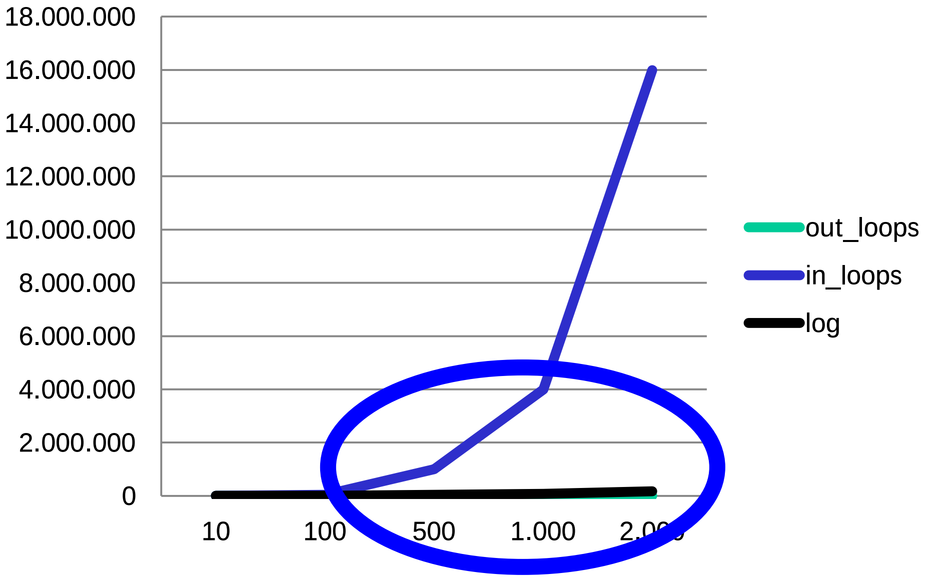


What Happened?

- Most combinations (region, contract type) select only a handful of contracts
- A few combinations select many contracts (2000-5000)
- Time it takes to fill the drop-down list is not proportional to the number of contracts (n), but proportional to $n^2/2$
  – Required time is quadratic in n
  – Assume one comparison takes 10 nanoseconds (0.000001 sec)
  – A handful of contracts (~10): ~400 operations => 0,0004 sec
  – Many contracts (~5000) => ~100M operations => 100 sec
  – Humans always expect linear time ...
- Question: Could they have done it better?

Of course

- An efficient sorting algorithm needs app. n*log(n)*x ops
  - Quick sort, merge sort, ... – see later
  - For comparability, let's assume x=8
  - Under certain reasonable assumptions, one cannot sort faster than with ~n*log(n) operations



"Almost" linear

So there is an End to Research in Sorting?

- We didn‘t consider how long it takes to compare 2 strings
  - We used c=d=1, but we need to compare strings char-by-char
  - Time of every comparison is proportional to the length of the shorter string
- We want algorithms that require less than 8 operations per inner loop
- We want algorithms that are fast even if we want to sort 1.000.000.000 strings
  - Which might not fit into main memory
- We made a pessimistic estimate – what is a realistic estimate (how often do we swap in the inner loop?)?
- ...

Terasort Benchmark

- 2009: 100 TB in 173 minutes
  - Amounts to 0.578 TB/min
  - 3452 nodes x (2 Quadcore, 8 GB memory)
  - Owen O'Malley and Arun Murthy, Yahoo Inc.
- 2010: 1,000,000,000,000 records in 10,318 seconds
  - Amounts to 0.582 TB/min
  - 47 nodes x (2 Quadcore, 24 GB memory), Nexus 5020 switch
  - Rasmussen, Mysore, Madhyastha, Conley, Porter, Vahdat, Pucher
- Other goals
  - PennySort: Amount of data sorted for a penny's worth of system time
  - JouleSort: Minimize amount of energy required during sorting

Content of this Lecture

- This lecture
- Algorithms and ...
- Data Structures
- Concluding Remarks

Algorithms and Data Structures

- Slides are English
- Vorlesung wird auf Deutsch gehalten
- Lecture: 4 SWS; exercises 2 SWS
- Contact
  - Ulf Leser,
  - Raum IV.103
  - Tel: 2093 – 3902
  - eMail: leser (..) informatik . hu...berlin . de

Schedule

- Lectures: Monday 11-13, Wednesday 11-13, EZ 0115
- Exercises: See Goya

Exercises

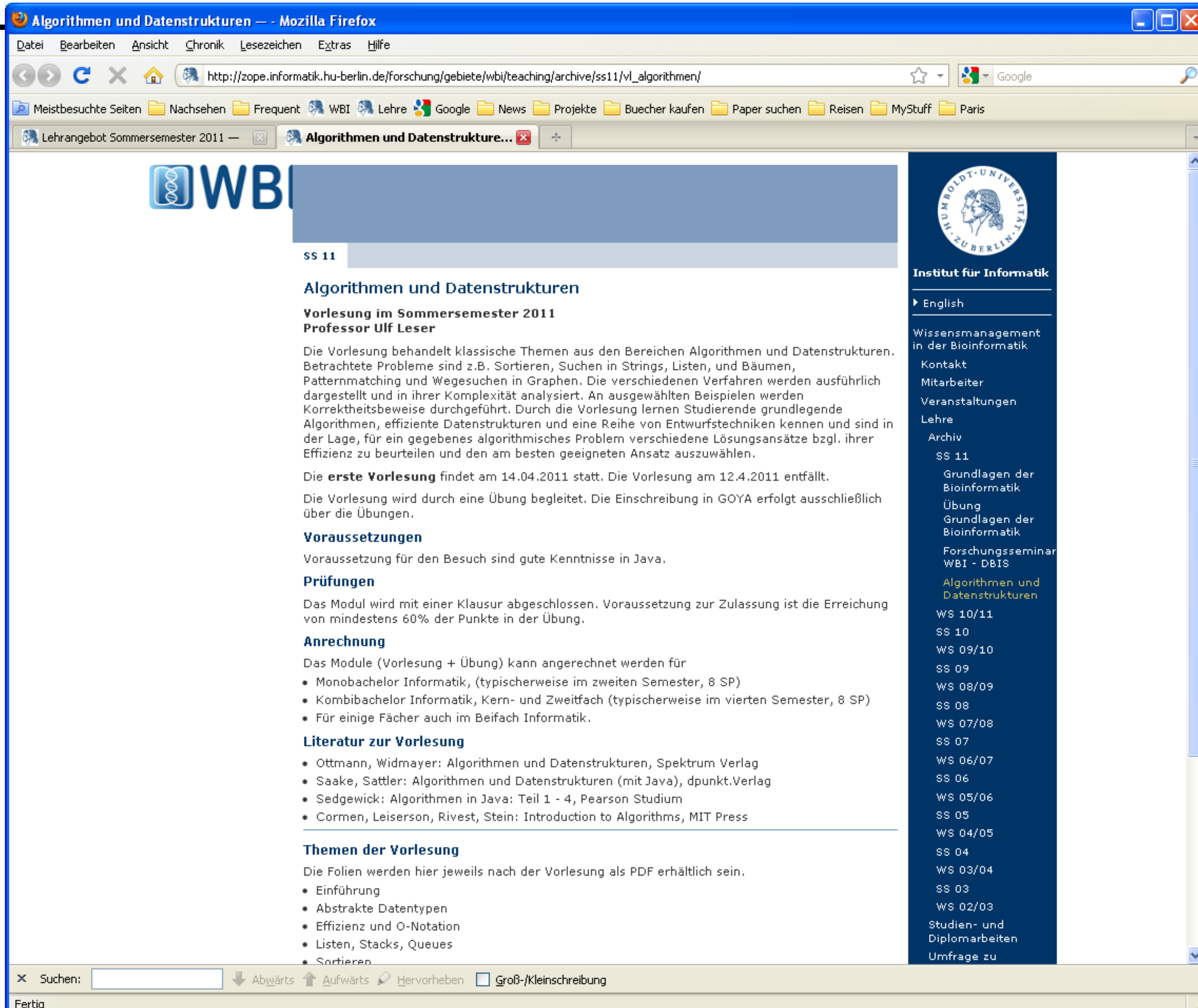
- You will build teams of two students
- There will be an assignment every two weeks
- You need to work on every assignment
- Each assignment gives 40 points max
- Only groups having ≥60% of the maximal number of points over the entire semester are admitted to the exam
- For every assignment, 2-3 students are selected at random (in each slot) and must present their solution
- Failing to do so more than two times implies exclusion from exercise
- It will be fun (mostly)

Literature

- Ottmann, Widmayer: Algorithmen und Datenstrukturen, Spektrum Verlag, 2002
  - 20 copies in library

- Other
  - Saake / Sattler: Algorithmen und Datenstrukturen (mit Java), dpunkt.Verlag, 2006
  - Sedgewick: Algorithmen in Java: Teil 1 - 4, Pearson Studium, 2003
    - 20 copies in library
  - Güting, Dieker: Datenstrukturen und Algorithmen, Teubner, 2004
  - Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, MIT Press, 2003
    - 10 copies in library

Web


Algorithmen und Datenstrukturen — - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://zope.informatik.hu-berlin.de/forschung/gebiete/wbi/teaching/archive/ss11/vl_algorithmen/
Google
Meistbesuchte Seiten Nachsehen Frequent WBI Lehre Google News Projekte Buecher kaufen Paper suchen Reisen MyStuff Paris
Lehrangebot Sommersemester 2011 —
Algorithmen und Datenstrukture...
WBI
SS 11
Algorithmen und Datenstrukturen
Vorlesung im Sommersemester 2011
Professor Ulf Leser
Die Vorlesung behandelt klassische Themen aus den Bereichen Algorithmen und Datenstrukturen. Betrachtete Probleme sind z.B. Sortieren, Suchen in Strings, Listen, und Bäumen, Patternmatching und Wegesuchen in Graphen. Die verschiedenen Verfahren werden ausführlich dargestellt und in ihrer Komplexität analysiert. An ausgewählten Beispielen werden Korrektheitsbeweise durchgeführt. Durch die Vorlesung lernen Studierende grundlegende Algorithmen, effiziente Datenstrukturen und eine Reihe von Entwurfstechniken kennen und sind in der Lage, für ein gegebenes algorithmisches Problem verschiedene Lösungsansätze bzgl. ihrer Effizienz zu beurteilen und den am besten geeigneten Ansatz auszuwählen.
Die erste Vorlesung findet am 14.04.2011 statt. Die Vorlesung am 12.4.2011 entfällt.
Die Vorlesung wird durch eine Übung begleitet. Die Einschreibung in GOYA erfolgt ausschließlich über die Übungen.
Voraussetzungen
Voraussetzung für den Besuch sind gute Kenntnisse in Java.
Prüfungen
Das Modul wird mit einer Klausur abgeschlossen. Voraussetzung zur Zulassung ist die Erreichung von mindestens 60% der Punkte in der Übung.
Anrechnung
Das Module (Vorlesung + Übung) kann angerechnet werden für
Monobachelor Informatik, (typischerweise im zweiten Semester, 8 SP)
Kombibachelor Informatik, Kern- und Zweitfach (typischerweise im vierten Semester, 8 SP)
Für einige Fächer auch im Beifach Informatik.
Literatur zur Vorlesung
Ottmann, Widmayer: Algorithmen und Datenstrukturen, Spektrum Verlag
Saake, Sattler: Algorithmen und Datenstrukturen (mit Java), dpunkt.Verlag
Sedgewick: Algorithmen in Java: Teil 1 - 4, Pearson Studium
Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, MIT Press
Themen der Vorlesung
Die Folien werden hier jeweils nach der Vorlesung als PDF erhältlich sein.
Einführung
Abstrakte Datentypen
Effizienz und O-Notation
Listen, Stacks, Queues
Sortieren
Institut für Informatik
English
Wissensmanagement in der Bioinformatik
Kontakt
Mitarbeiter
Veranstaltungen
Lehre
Archiv
SS 11
Grundlagen der Bioinformatik
Übung Grundlagen der Bioinformatik
Forschungsseminar WBI - DBIS
Algorithmen und Datenstrukturen
WS 10/11
SS 10
WS 09/10
SS 09
WS 08/09
SS 08
WS 07/08
SS 07
WS 06/07
SS 06
WS 05/06
SS 05
WS 04/05
SS 04
WS 03/04
SS 03
WS 02/03
Studien- und Diplomarbeiten
Umfrage zu
Suchen: Abwärts Aufwärts Hervorheben Groß-/Kleinschreibung
Fertig

Pseudo Code

- You need to program all exercises in Java
- I will use informal pseudo code
  - Much more concise than Java
  - Goal: You should understand what I mean
  - Syntax is not important; don’t try to execute programs from slides
- Translation into Java should be simple

Topics of the Course

- Abstract data types (~2)
- Machine models and complexity (~2)
- Lists (~3)
- Sorting (~5)
- Hashing (~3)
- Trees (~5)
- Graphs (~4)

Questions?

Questions

- Diplominformatiker?
- Bachelor?
- Semester?
- Kombibachelor?
- INFOMIT? Biophysics? Beifach?
- Who heard this course before?

Content of this Lecture

- This lecture
- Algorithms and ...
- Data Structures
- Concluding Remarks

What is an Algorithm?

- An algorithm is a recipe for doing something
  - Washing a car, sorting a set of strings, preparing a pancake, employing a student, ...
- The recipe is given in a (formal, clearly defined) language
- The recipe consists of atomic steps
  - Someone (the machine) must know what to do
- The recipe must be precise
  - After every step, it must be uniquely decidable what comes next
  - Does not imply that every run has the same sequence of steps
- The recipe must not be infinitely long

More Formal

- Definition (general)
  An algorithm is a precise and finite description of a process consisting of elementary steps.

- Definition (Computer Science)
  An algorithm is a precise and finite description of a process that is (a) given in a formal language and (b) consists of elementary and machine-executable steps.

- Usually we also want: “and (c) solves a given problem”
  – But algorithms can be wrong ...

Almost Synonyms

- Rezept
- Ausführungsvorschrift
- Prozessbeschreibung
- Verwaltungsanweisung
- Regelwerk
- Bedienungsanleitung
  - Well …
- …

History

- Word presumably dates back to “Muhammed ibn Musa abu Djafar alChoresmi”,
  - Published a book on calculating in the 8th century in Persia
  - See Wikipedia for details
- Given the general meaning of the term, there have been algorithms since ever
- One of the first in math: Euclidian algorithm for finding the greatest common divisor (gcd) of two integers
  - Assume $a,b \geq 0$; define $gcd(a,0)=a$

Euclidian Algorithm

Actually not really precise

- Recipe: Given two integers a, b. As long as neither a nor b is 0, take the smaller of both and subtract it from the greater. If this yields 0, return the other number
- Example: (28, 92)
  - (28, 64)
  - (28, 36)
  - (28, 8)
  - (20, 8)
  - (12, 8)
  - (4, 8)
  - (4, 4)
  - (4, 0)
- Will this always work?

```
a,b: integer;
if a=0 return b;
while b≠0
  if a>b
    a := a-b;
  else
    b := b-a;
  end if;
end while;
return a;
```

Proof that an Algorithm is Correct

```
func euclid(a,b: int)
  if a=0 return b;
  while b≠0
    if a>b
      a := a-b;
    else
      b := b-a;
    end if;
  end while;
  return a;
end func;
```

- Assume our function “euclid” returns x
- We write “b|a” if (a mod b)=0
  - We say: “b teilt a”
- Note: if c|a and c|b and a>b $\Rightarrow$ c|(a-b)
- Thus, if euclid(a,b)=x $\Rightarrow$ x|a and x|b
- But is it the greatest?
  - Assume some y with y|a and y|b
  - It follows that y|(a-b) (or y|(b-a))
  - It follows that y|((a-b)-b) (or y|((b-a)-b) ...)
  - ...
  - It follows that y|x
  - Thus, y≤x

Properties of Algorithms

- Definition
  We say an algorithm A is terminating, if A stops after a finite number of steps for every valid input.
- Definition
  We say an algorithm A is deterministic, if A always performs the same series of steps for the same input.

- We only study terminating and mostly deterministic algs
  - Operating systems are “algorithms” that do not terminate
  - Algs randomly deciding about next steps are not deterministic

Algorithms and Runtimes

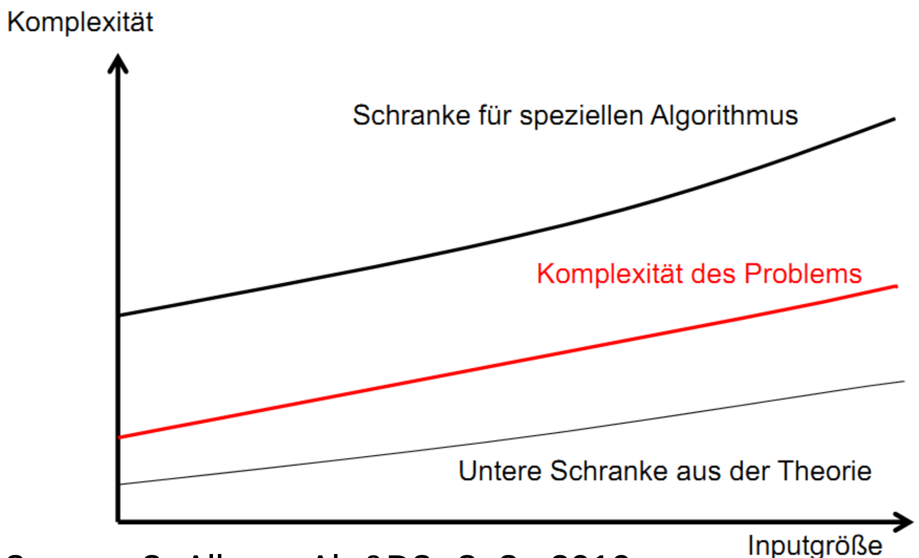
- Usually, one seeks efficient (read: fast) algorithms
- We will analyze the efficiency of an algorithm as a function of the size of its input; this is called its (time-)complexity
  - Selection-sort has time-complexity "O(n^2)"
- The runtime of an algorithm depends on many factors most of which we gracefully ignore
  - Clock rate, processor, programming language, representation of primitive data types, available main memory, cache lines, ...
- But: Complexity in some sense correlates with runtime
  - It should correlate well in most cases, but there may be exceptions (especially on small inputs)

Algorithms, Complexity and Problems

- An (correct) algorithm solves a given problem
- An algorithm has a certain complexity
  - Which is a statement about the time it will take to finish as a function on the size of its input
- Also problems have complexities
  - The complexity of a problem is a lower bound on the complexity of any algorithm that solves it
  - If an algorithm has the same complexity as the problem it solves, it is optimal – no algorithm can solve this problem faster
- Beware: Proving the complexity of a problem usually is much harder than proving the complexity of an algorithm
  - Needs to make a statement about any possible algorithms

Anything Goes

- There are problems for which we know their complexity, but no optimal algorithm is known
- Their are problems for which we do not know the complexity yet more and more efficient algorithms are discovered over time
- There are problems for which we only know lower thresholds on their complexity, but not the precise complexity
- There are problems of which we know that no algorithm exists
  - Undecidable problems
  - Example: “Halteproblem”
  - Implies that we cannot check in general if an algorithm is terminating



Source: S. Albers, Alg&DS; SoSe 2010

Properties of Algorithms

1. Efficiency – how long will it take?
   - Time complexity – changes in runtime with growing input
   - Worst-case, average-case, best-case
   - Alternative: Run on reference machine usi
     - Done a lot in practical algorithm engineeri
     - Not so much in this introductory course
2. Space consumption – how much memory will it need?
   - Space complexity
   - Worst-case, average-case, best-case
   - Can be decisive for large inputs
3. Correctness – does the algorithm solve the problem?

Often, one can trade space for time – look at both

In This Course

- We will only occasionally look at space complexity
- We will mostly focus on worst-case (time) complexity
  - Best-case is not very interesting
  - Average-case often is hard to determine
    - What is an „average string list“?
    - What is average number of twisted sorts in an arbitrary string list?
    - What is the average length of an arbitrary string?
    - May depend in the semantic of the input (person names, DNA sequences, job descriptions, book titles, language, ...)
- Keep in mind: Worst-case often is overly pessimistic

Content of this Lecture

- This lecture
- Algorithms and ...
- Data Structures
- Concluding Remarks

What is a Data Structure?

- Algorithms work on input data, generate intermediate data, and finally produce a result (data)
- A data structure is a way how data is represented inside the machine
  - In memory or on disc (see Database course)
- Data structures determine what algs may do at what cost
  - More precisely: ... what a specific step of an algorithm costs
- Complexity of algs is tightly bound to the ds they use
  - So tightly that one often subsumes both concepts under the term "algorithm"

Example: Selection Sort (again)

```
S: array_of_names;
n := |S|;
for i = 1..n-1 do
  for j = i+1..n do
    if S[i]>S[j] then
      tmp := S[j];
      S[i] := S[j];
      S[j] := tmp;
    end if;
  end for;
end for;
```

- We assumed that S is
  - a list of strings (abstract), represented
  - as an array (concrete data structure)
- Arrays allow us to access the i'th element with a cost that is independent of i (and |S|)
  - Constant cost, O(1)
- Let's use a linked list for storing S
  - Create a class C holding a string and a pointer to an object of C
  - Put first s∈S into first object and point to second object, put second s into second object and point to third object, ...
  - Keep a pointer p_0 to the first object

Selection Sort with Linked Lists

```
1.  i := p0;
2.  while i.next != null do
3.    j := i.next;
4.    while j.next != null do
5.      if i.val > j.val then
6.        tmp := i.val;
7.        i.val := j.val;
8.        j.val := tmp;
9.      end if;
10.     j = j.next;
11.   end while;
12.   i := i.next;
13. end while;
```

- How much do the algorithm's steps cost now?
  - Assume following a pointer costs c

  1. One assignment
  2. One comparison, n times
     3. One assignment, n-i+1 times
     4. One comparison
        5. ...
- Apparently no change in complexity

Example Continued

```
i := p0;
while i.next != null do
  j := i.next;
  while j.next != null do
    if i.val > j.val then
      tmp := i.val;
      i.val := j.val;
      j.val := tmp;
    end if;
    j = j.next;
  end while;
  i := i.next;
end while;
```

- No change in complexity, but
  - Previously, we accessed array elements, performed additions of integers and comparisons of strings, and assigned values to integers
  - Now, we assign pointers, follow pointers, compare strings and follow pointers again
- These differences are not reflected in our “cost model”, but may be big in practice

Content of this Lecture

- This lecture
- Algorithms and Data Structures
- Concluding Remarks

Why do you need this?

- You will learn things you will need a lot through all of your professional life
- Searching, sorting, hashing – cannot Java do this for us?
  - Java libraries contain efficient implementations for most of the (basic) problems we will discuss
  - But: Choose the right algorithm / data structure for your problem
    - TreeMap? HashMap? Set? Map? Array? ...
    - “Right” means: Most efficient (space and time) for the expected operations: Many inserts? Many searches? Biased searches? ...
- Few of you will design new algorithms, but all of you often will need to decide which algorithm to use when
- To prevent problems like the ones we have seen earlier