Algorithms and Data Structures

Ulf Leser

Ettikette

- Recording is not allowed
- Enrollment in exercises is still going on
- We are heavily overbooked
- If we reach 300 – everybody without an assignment to an exercise group please leave
- Put question in the chat
  - Let’s see how many there will be
- The chat is surveyed – misbehavior will have consequences

Who am I

- Ulf Leser

- 1995 Diploma in Computer Science, TU München
- 1996-1997 Database developer at MPI-Molecular Genetics
- 1997-2000 Dissertation in Database Integration, TU Berlin
- 2000-2003 Developer and project manager at PSI AG
- 2002- Prof. Knowledge Management in Bioinformatics

- I do answer emails

Wissensmanagement in der Bioinformatik

- Our topics in research
  - Biomedical data management
  - Text Mining
  - Scientific Data Analysis
- Our topics in teaching
  - Bsc: Grundlagen der Bioinformatik (5 SP)
  - Bsc: Information Retrieval (5 SP)
  - Msc: Algorithmische Bioinformatik (10 SP)
  - Msc: Data Warehousing und Data Mining (10 SP)
  - Msc: Informationsintegration (10 SP)
  - Msc: Maschinelle Sprachverarbeitung (5 SP)
  - Msc: Implementierung von Datenbanken (10 SP)

SHK Stelle

- FONDA: Foundations of Workflows for Large-Scale Scientific Data Analysis
- Tasks
  - Pflege und Entwicklung von Softwarebibliotheken
  - Pflege und Aufbau von verteilten Software-Infrastrukturen
  - Entwicklung von Datenanalyse-Workflows
  - Mitarbeit in interdisziplinären Projekten in der Genomforschumg, den Materialwissenschaften, oder der Satellitenbildaufklärung
- Requirements
  - Erfahrung im Programmieren, insb. mit modernen Scriptsprachen
  - Gute Kenntnisse im Software Engineering
  - Interesse an der interdisziplinärer Forschung
- Interested: https://www.informatik.hu-berlin.de/wbi

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

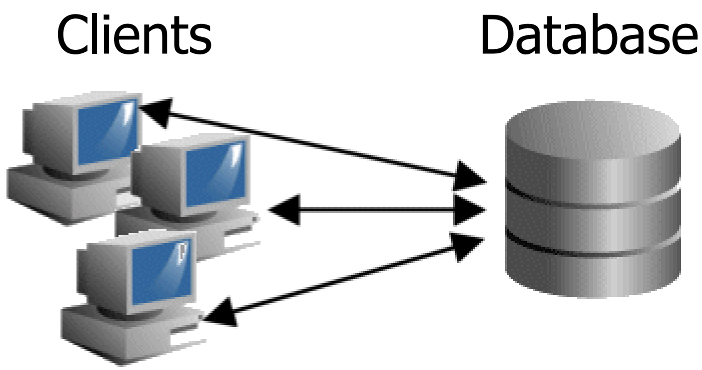
- Requirement: „Allows for smooth operations in daily routine“
- Claim from B
  - I search a specific contract
  - I select a region and a contract type
  - I get a list of all contracts sorted by name in a drop-down box
  - This sometimes takes minutes! A simple drop-down box! This performance is inacceptable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

Discussion

- A: We tested and it worked fined
- B: Yes - 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 in real-life operations
- 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

Query Construction

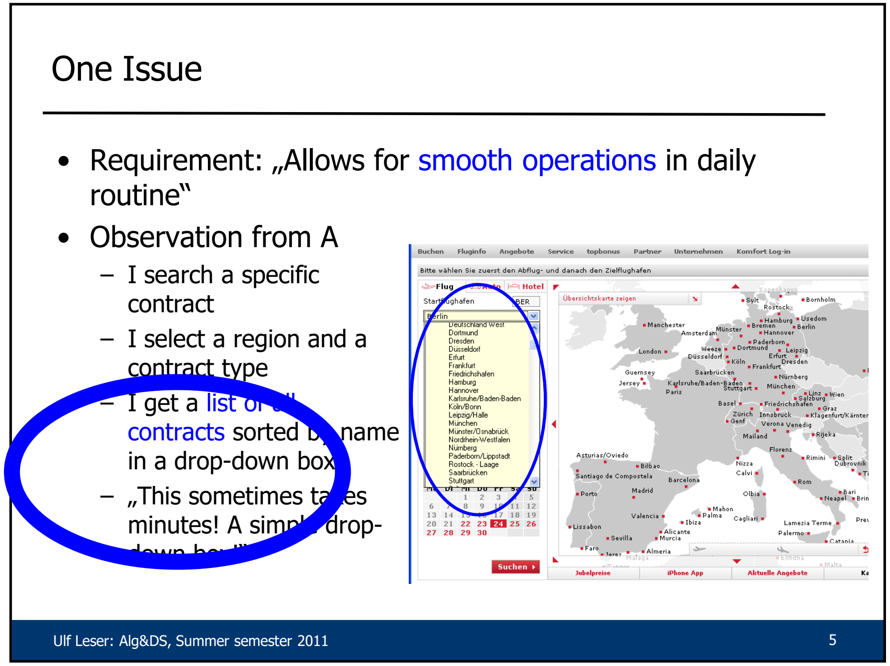
```
SELECT CU.name, CO.type, CO.start, CO.end, CO.volume, …
FROM customer CU, contracts CO, c_c CC, region R, …
WHERE   CU.ID=CC.CU_ID AND
        CO.ID=CC.CO_ID AND
        CU.regionID = R.ID AND
        …
        CU.ID=4711 AND CO.type=„Hausrat"
```

Query Construction

```
SELECT CU.name, CU.street, CU.status, CU.contact, …
FROM customer CU, contracts CO, c_c CC, region R, …
WHERE   CU.ID=CC.CU_ID AND
        CO.ID=CC.CO_ID AND
        CU.regionID = R.ID AND
        …
        R=„Berlin“ AND CO.type=„Leben“
```

Requirement

- Recall



- After retrieving the list of customers, it has to be sorted
- Adding a SQL "order by" deemed too complicated
- But– sorting is easy!

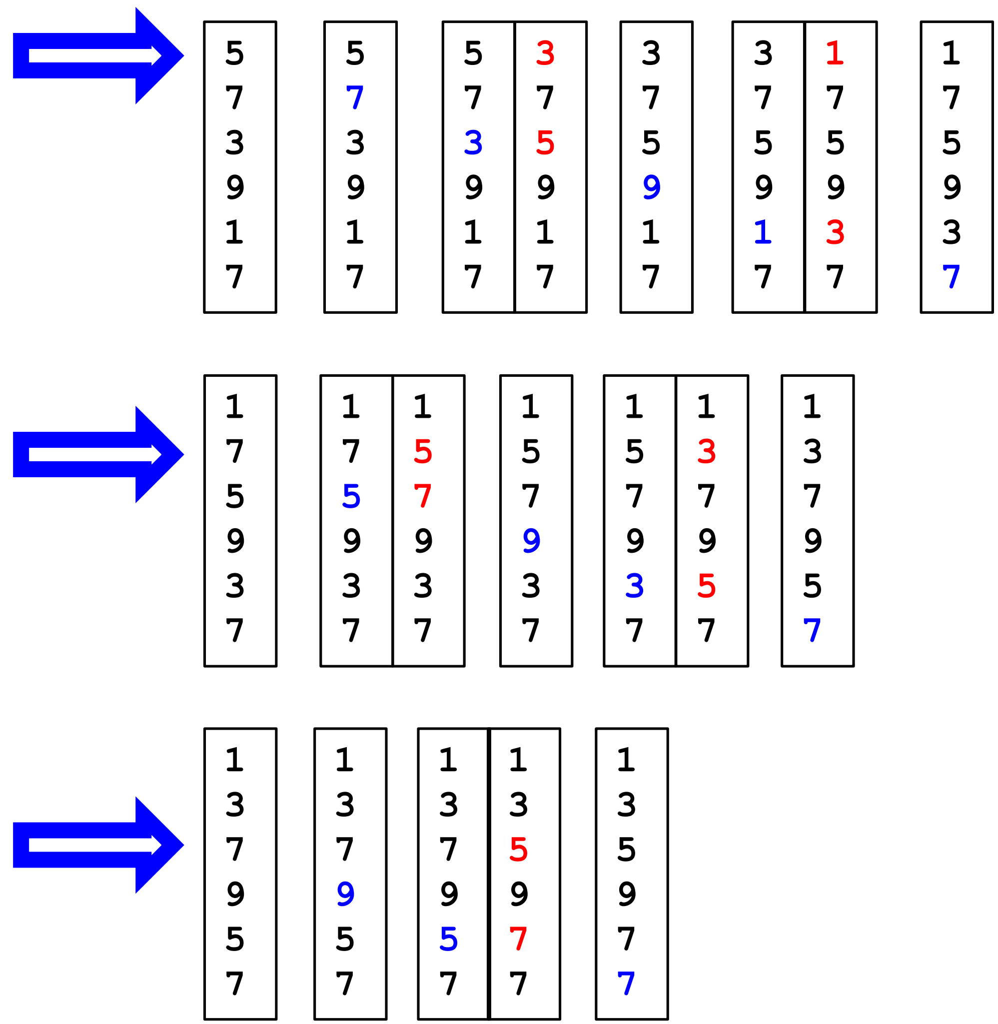
Code used for Sorting the List of Customer Names

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

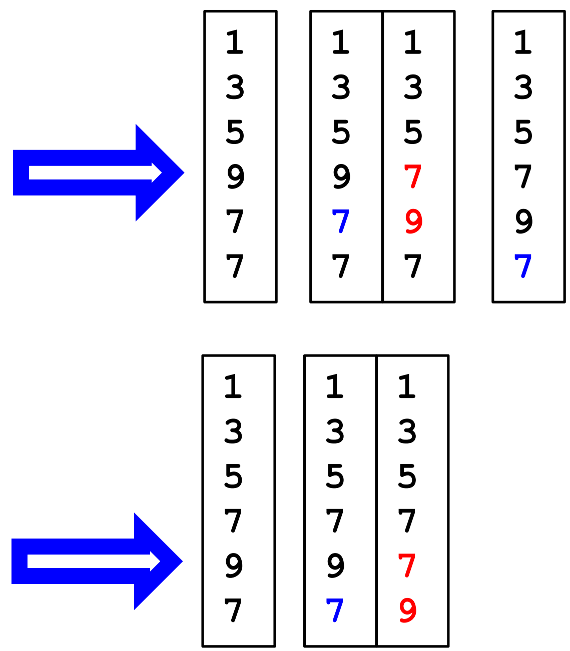
- S: array of Strings, |S|=n
- Sort S alphabetically
  - Take the first string and compare to all others
  - Swap whenever a later string is alphabetically smaller
  - Repeat for 2nd, 3rd, … string
  - After 1st iteration of outer loop: S[1] contains smallest string from S
  - After 2nd iteration of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

Analysis

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

1. Probably very little, constant time
2. Probably very little, constant time
3. n-1 assignments
    4. n-i assignments
        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.        tmp := S[i];
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. c
2. c
3. (n-1)
   4. (n-i) (hmmm …)
      5. c
         6. c (hmmm …)
         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.        tmp := S[i];
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?
  - 2*c
  - (n-1)* (
    - n-i* (
      - 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
      tmp := S[i];
      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 loops (out_loop)
  - And some cost inside the loops (in_loop)
  - How often do we need to perform in_loop?
  - Total:
    $c+(n-1)*(c+d)+(n-1)*((n-i)*\ldots)=$
    $out\_loop+(n-1)*(c+d)+(n-1)*?*in\_loop$

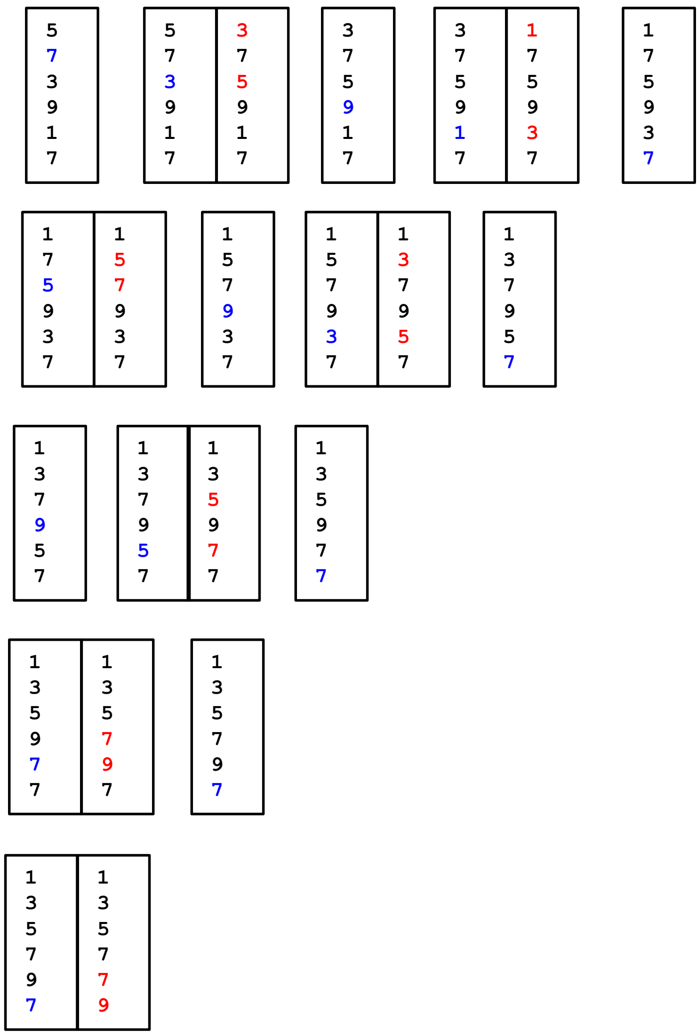
Outer FOR-LOOP

Inner FOR-LOOP

out_loop (lines 1–2), in_loop (lines 5–9):

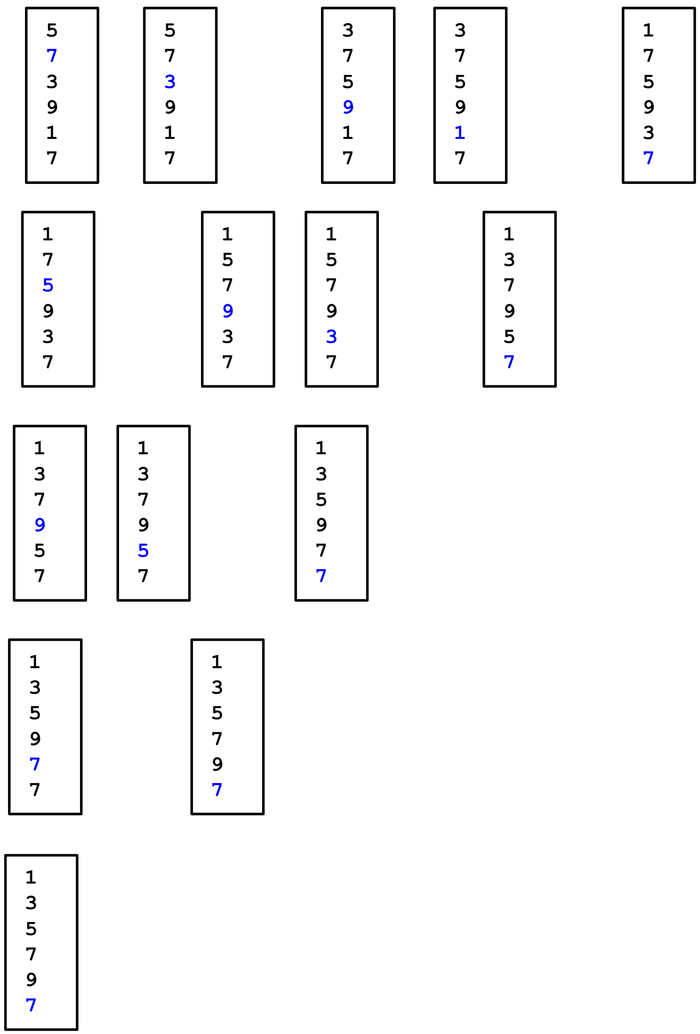
```
S: array_of_names;
n := |S|;
for i = 1..n-1 do
  for j = i+1..n do
    if S[i]>S[j] then
      tmp := S[i];
      S[i] := S[j];
      S[j] := tmp;
    end if;
  end for;
end for;
```

Observations



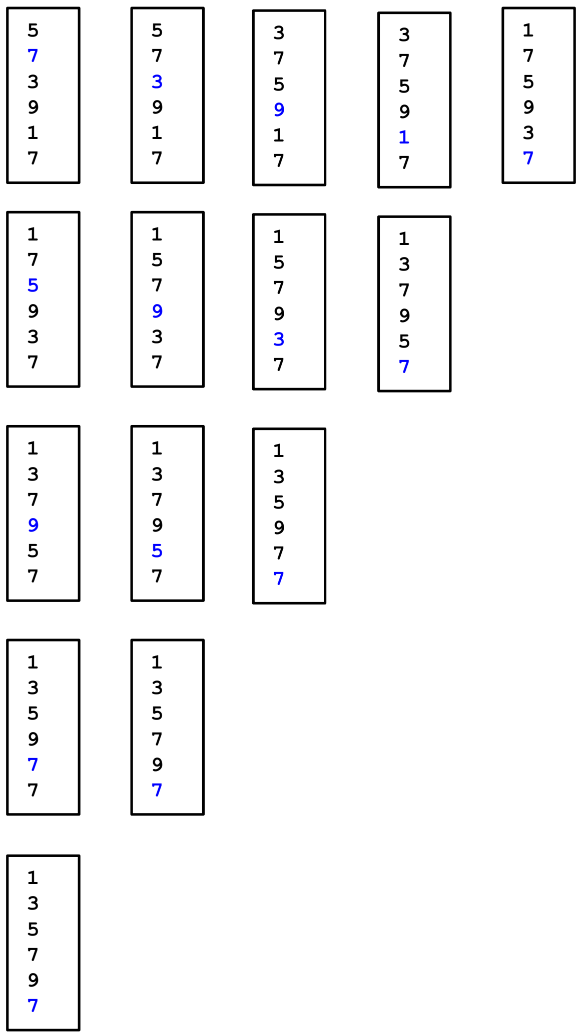
- The number of comparisons is independent of the number of swaps
  - We always compare, but we do not always swap

Observations



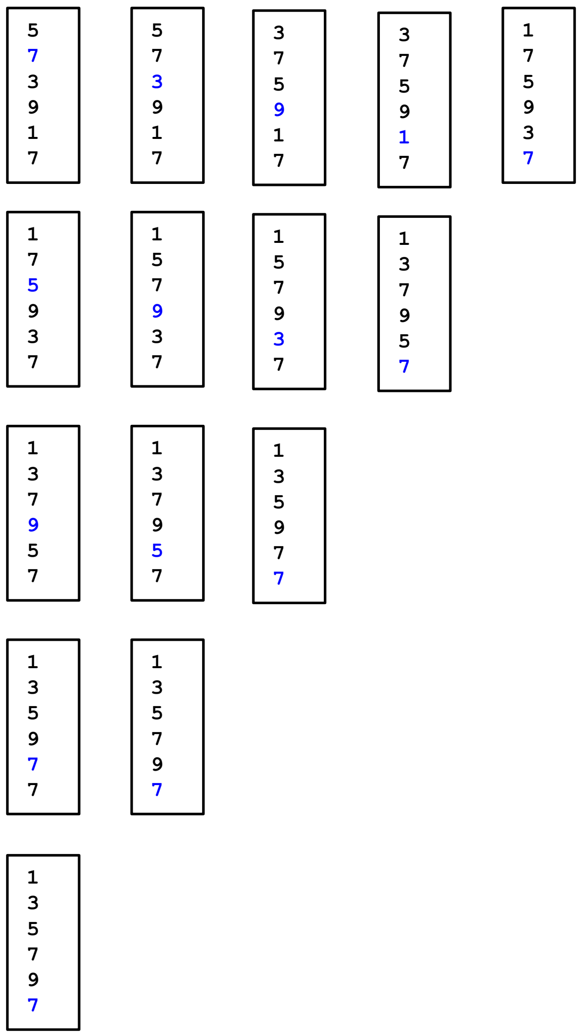
- The number of comparisons is independent of the number of swaps
  - We always compare, but we do not always swap
- How many comparisons do we perform in total?

Observations



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

Observations



- First string is compared to n-1 other strings
  - First row
- Second is compared to n-2
  - Second row
- Third is compared to n-3
- …
- n-1'th is compared to 1

Together

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

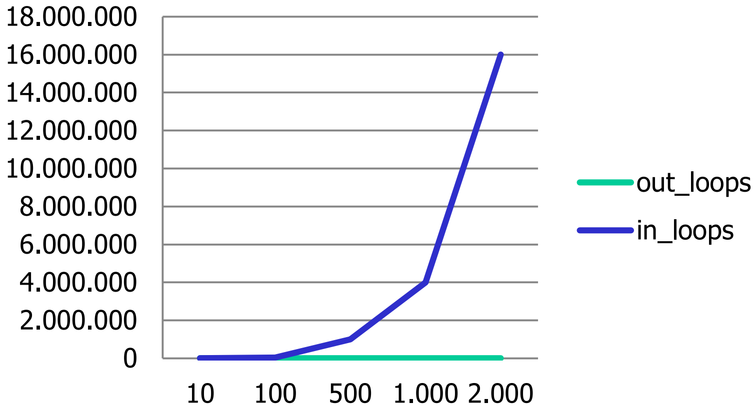
- This leads to the following estimation for the total cost

  out_loop+3(n-1)*(c+d)+(n^2-n)*in_loop/2

- Let's assume c=d=1

  2+3(n-1)+(n^2-n)*6/2=3n+1+3(n^2-2)

	out_loop	in_loop	total
10	31	294	325
100	301	29.994	30.295
500	1.501	749.994	751.495
1.000	3.001	2.999.994	3.002.995
2.000	6.001	11.999.994	12.005.995
5.000	15.001	74.999.994	75.014.995

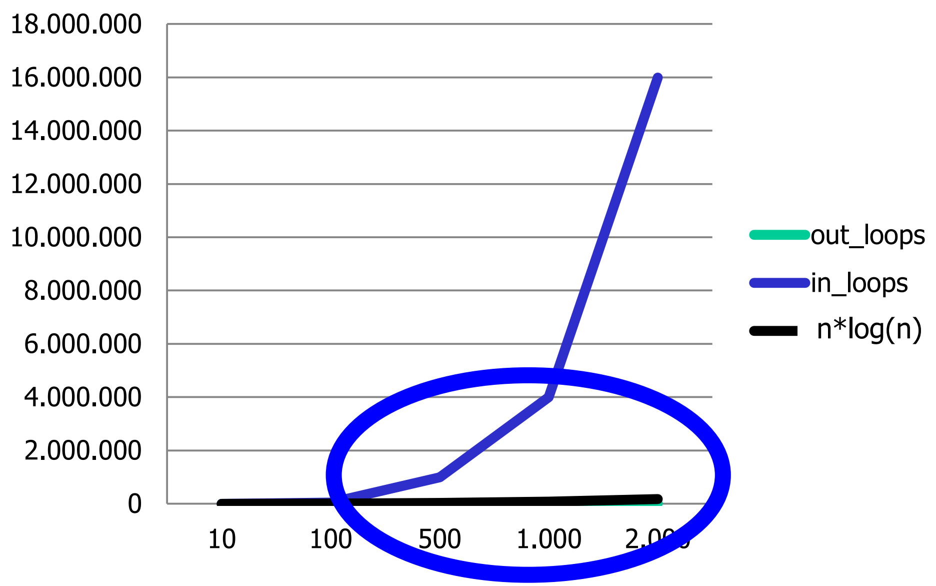


What Happened?

- Most combinations (region, contract type) select only a handful of contracts
- A few combinations select many contracts (>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 operation takes 100 nanoseconds (0.000 000 1 sec)
  - A handful of contracts (~10): ~300 operations => 0,000 03 sec
  - Many contracts (~5000) => ~75M operations => 7,5 sec
  - Humans tend to always expect linear relationships …
- Question: Could they have done it better?

Of course

- Efficient sorting algorithms need ~n*log(n)*x operations
  - Quick sort, merge sort, … see later
  - For comparability, let's assume x=6
  - We will proof that sorting in less operations in impossible
    - In some sense



"log-linear",
"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 requiring less operations per inner loop (smaller x)
- We want algorithms that are fast even if we want to sort 1.000.000.000 strings
  - Which do 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

More recent results

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Ja	Ja	Nein
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

	Daytona	Indy
Gray	2016, 44.8 TB/min **Tencent Sort** 100 TB in 134 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 60.7 TB/min **Tencent Sort** 100 TB in 98.8 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub

Only throughput?

- PennySort: Amount of data sorted for a penny's worth of system time
- CloudSort: Cost (Euro) for sorting a data on a public cloud
- 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.401
  - Tel: 2093 – 3902
  - eMail: leser (..) informatik . hu…berlin . de

Lecture: Schedule and Modus

- Lectures
  - Monday 11-13, Wednesday 11-13
  - Held synchronous over Zoom (no recording)
  - Slides are available shortly after lecture on web page
  - Pre-recorded lectures available from SoSe 2020
    - Thanks to Henning Meyerhenke!
  - Zoom chat will be monitored, do not misbehave
  - Questions always possible (zoom chat)

Exercises

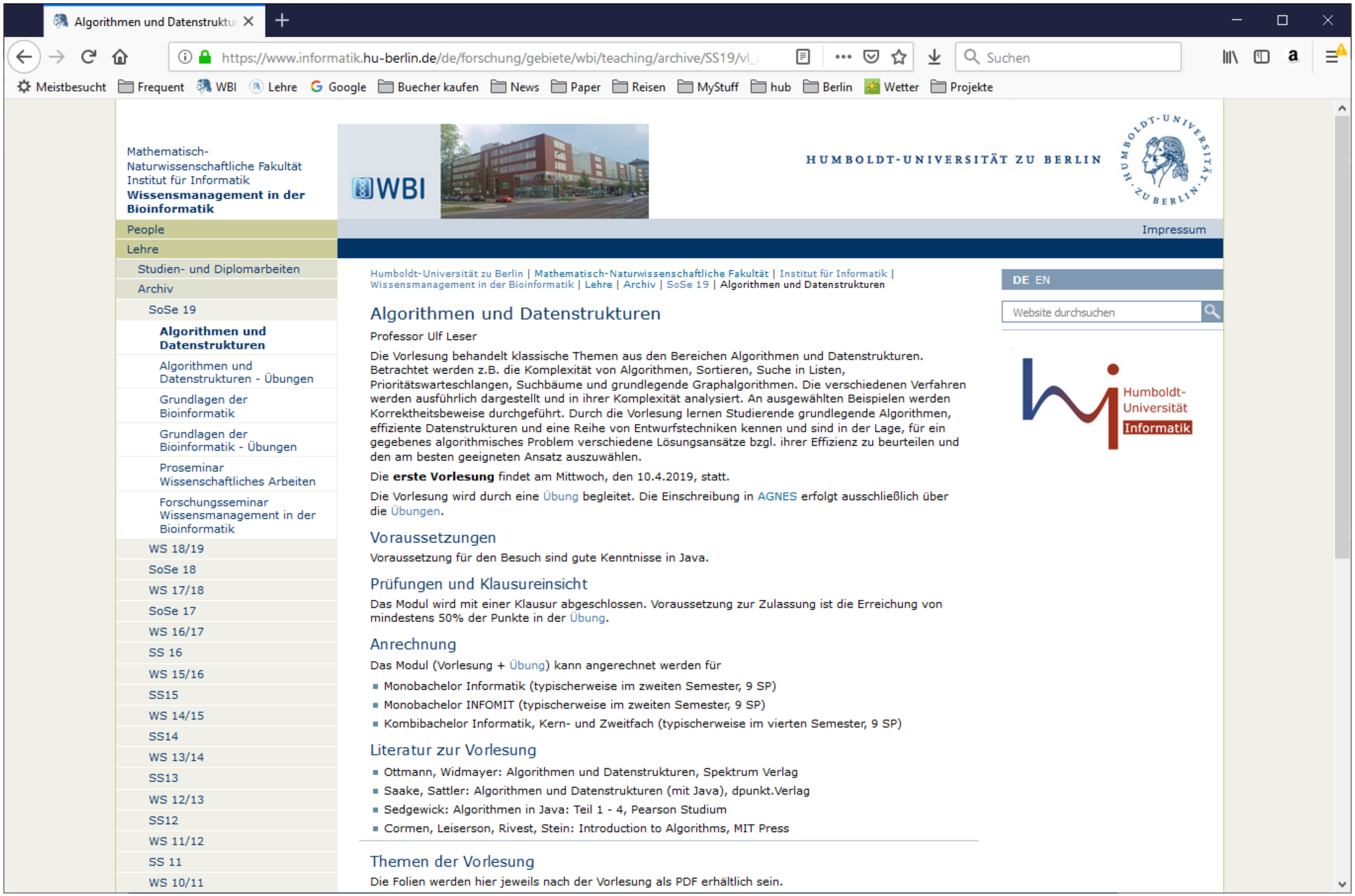
- Several slots: See webpage / AGNES / Moodle
  - Two slots are English
- Held synchronous over Zoom (no recording)
- Start this week, but first assignment is next week
- You will build teams of two students
- There will be an assignment about every two weeks
- You need to work on every assignment
- Each assignment gives 50 points max
- Only groups having >50% of the maximal number of points over the entire semester are admitted to the exam
- Moodle key: Dijkstra_2021!

Questions?

Literature

- Ottmann, Widmayer: Algorithmen und Datenstrukturen, Spektrum Verlag, 2002-2012
  - 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 Datenstruktu
https://www.informatik.hu-berlin.de/de/forschung/gebiete/wbi/teaching/archive/SS19/vl_
Suchen
Meistbesucht
Frequent
WBI
Lehre
Google
Buecher kaufen
News
Paper
Reisen
MyStuff
hub
Berlin
Wetter
Projekte
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik
Wissensmanagement in der Bioinformatik
WBI
HUMBOLDT-UNIVERSITÄT ZU BERLIN
People
Lehre
Studien- und Diplomarbeiten
Archiv
SoSe 19
Algorithmen und Datenstrukturen
Algorithmen und Datenstrukturen - Übungen
Grundlagen der Bioinformatik
Grundlagen der Bioinformatik - Übungen
Proseminar Wissenschaftliches Arbeiten
Forschungsseminar Wissensmanagement in der Bioinformatik
WS 18/19
SoSe 18
WS 17/18
SoSe 17
WS 16/17
SS 16
WS 15/16
SS15
WS 14/15
SS14
WS 13/14
SS13
WS 12/13
SS12
WS 11/12
SS 11
WS 10/11
Impressum
Humboldt-Universität zu Berlin | Mathematisch-Naturwissenschaftliche Fakultät | Institut für Informatik | Wissensmanagement in der Bioinformatik | Lehre | Archiv | SoSe 19 | Algorithmen und Datenstrukturen
Algorithmen und Datenstrukturen
Professor Ulf Leser
Die Vorlesung behandelt klassische Themen aus den Bereichen Algorithmen und Datenstrukturen. Betrachtet werden z.B. die Komplexität von Algorithmen, Sortieren, Suche in Listen, Prioritätswarteschlangen, Suchbäume und grundlegende Graphalgorithmen. 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 Mittwoch, den 10.4.2019, statt.
Die Vorlesung wird durch eine Übung begleitet. Die Einschreibung in AGNES erfolgt ausschließlich über die Übungen.
Voraussetzungen
Voraussetzung für den Besuch sind gute Kenntnisse in Java.
Prüfungen und Klausureinsicht
Das Modul wird mit einer Klausur abgeschlossen. Voraussetzung zur Zulassung ist die Erreichung von mindestens 50% der Punkte in der Übung.
Anrechnung
Das Modul (Vorlesung + Übung) kann angerechnet werden für
Monobachelor Informatik (typischerweise im zweiten Semester, 9 SP)
Monobachelor INFOMIT (typischerweise im zweiten Semester, 9 SP)
Kombibachelor Informatik, Kern- und Zweitfach (typischerweise im vierten Semester, 9 SP)
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.
DE EN
Website durchsuchen
Humboldt-Universität
Informatik

Pseudo Code

- You need to program 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

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

April

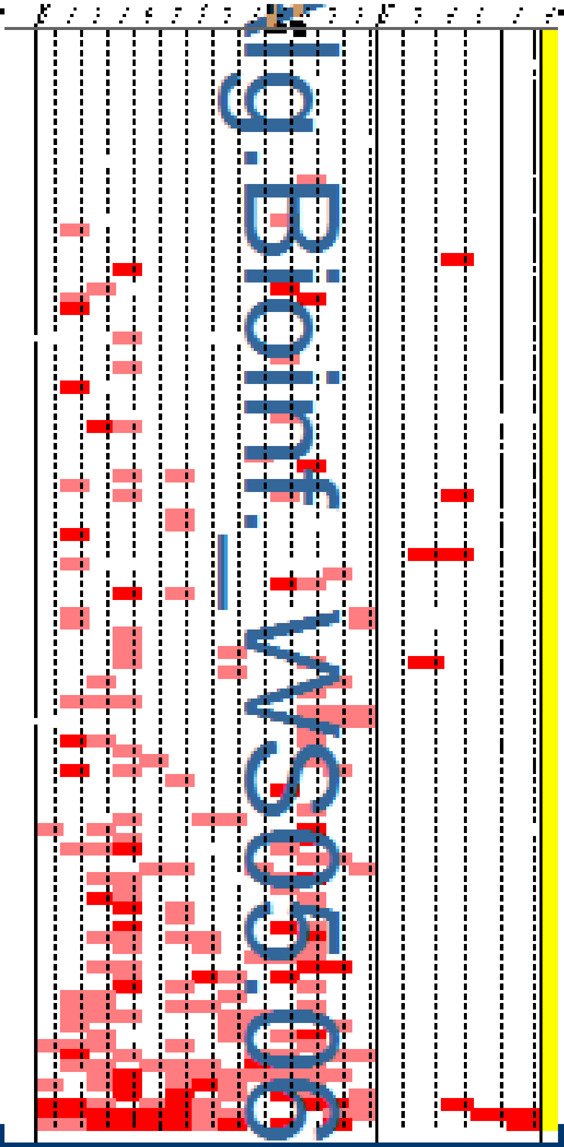
Mai

June

July

113 Evaluation Forms

- Very good scores
- Materials could (always) be better
- Discerning BA, KB, INFOMIT impossible
- Many liked it a lot, a few strongly disliked it



Freitexthinweise

Gut gefallen	Nicht gefallen	Zu wenig	Zu viel	Sonstiges
• 21 Beispiele (Praxis) • 15 Stil • 15 Sehr gut erklärt • 5 Gute Struktur • Möglichkeit für Fragen • Abstimmung VL – UE • 3 Engagement für Verständnis • 12 Alg der Woche • 11 Hochschulpolitik • 3 Tempo • 2 Zweiwöchige Übung • 2 Folien • 2 Englische Folien • Übung • Themenvielfalt • 3 Einleitende Wdhs • Verbindungen zu anderen Themen • 2 Pünktlichkeit • Wenig Vertretung • Sehr nützliche Inhalte • 2 Es wurde diskutiert • Schnelle Korrekturen der Folien	• 4 Zu langsam • 11 Englische Folien • Struktur manchmal unklar • Manche Themen zu kurz • **3 Husten und räuspern** • **Hinweis auf „nur Grundlagen"** • Terminkollision • Mathematische Wüsten • **Grüner Laserpointer** • Langsamer sprechen • Zu viel Text • Amortisierte Analyse raus • **2 Folien kein Script** • **Uni-Politik zu reißerisch und einseitig** • **3 Mikro-Einstellung** • **VL-Zeit nicht voll ausgenutzt** • Manchmal einschläfernd	• **4 Formaler machen** • Englisch vortragen • 7 Alg der Woche • 2 Programmierung • 4 Beweise • Hochschulpolitik • Lambda-Notation zu schnell • Interaktion und Tafel • Zusatzliteratur • **Motivierende Erklärungen** • 2 Beispiele • Mehr Tafel benutzen	• **11 Hochschulpolitik** • **4 Bioinformatik** • Verschiedene Fak beim Verfolgen der VL (?) • Zu viel * in UE • **Zu wenig echtes Interesse an Bildung** • 2 Übungen • Sehr zeitaufwändig • AlgdWoche weglassen • **2 Fehler in Folien** • Sehr lange Beispiele • Komplexitätsanalysen	• Mikro leiser • Mehr Praxis • Alg der Woche erfordern zu viel Vorwissen • Licht für Tafel • **Schwierige Themen einfacher darstellen** • 3 Folien verbessern (überladen) • **Team der Übungen super** • **Quiz in letzten 10m** • **Schlechte Luft** • **Folien nicht doppelt zeigen** • **Gesellschaftlich relevante Dinge besprechen, nicht nur Uni-Politik** • Mehr Ersatzbatterien • Variablen in Pseudo-Code bei Wdh unklar • **2 Niemand schläft ein** • Pseudo-Code besser erklären • Mehr Zeit bei komplexen Themen • Mute-Knopf benutzen • Lieber wöchentliche Übungen • Folien vorab online stellen

Highlights

- Danke für MERGESORT, half beim Sortieren von Blumentöpfen in der Gärtnerei meiner Oma
- Prof. Leser ist vertrauenswürdig. Wenn er sagt, dass etwas stimmt, glaube ich es auch ohne Beweis. Beweise weglassen und Zeit sinnvoller nutzen

Zusammenfassung

- Hochschulpolitik: 12 gut, 11 schlecht
- Alg der Woche: 19 gut, 1 schlecht
- Englische Folien: 2 gut, 11 schlecht
- Tempo: 3 gut, 4 zu langsam, 6 zu schnell
- Formale Beweise: 8 bitte formaler, 7 bitte weniger formal

Questions?

Questions – ZOOM QUIZ

- Monobachelor?
- Kombibachelor?
- INFOMIT?
- IMP?
- Semester
- 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 at each step
- The recipe must be precise
  - After every step, it is unambiguously decidable what to do next
  - Does not imply that every run has the same sequence of steps
    - There can be randomized steps; there is input
- 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
  - "To hunt a mammoth, you should …"
- One of the first mathematical ones: Euclidian algorithm for finding the greatest common divisor of two integers a, b
  - Assume $a,b \geq 0$; define $gcd(a,0)=a=gcd(0,a)$

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) (a_0, b_0)
  - (28, 64) (a_1, b_1)
  - (28, 36) (a_2, b_2)
  - (28, 8) …
  - (20, 8)
  - (12, 8)
  - (4, 8)
  - (4, 4)
  - (4, 0)

```
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;
```

- Will this always work?

Proof (sketch) 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)
- We prove the claim in two steps
  - We show that x is a common divisor
  - We prove that no greater common divisor can exist

Proof (sketch) 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;
```

- 1st step: We prove that x is a common divisor of a and b
  - Assume we required k loops
  - k'th step: $b_k=0$ and $x=a_k \neq 0 \Rightarrow x|a_k, x|b_k$
  - k-1: It must hold: $a_{k-1}=b_{k-1} \Rightarrow x|a_{k-1}, x|b_{k-1}$
  - k-2: Either $a_{k-2}=2x$ or $b_{k-2}=2x \Rightarrow x|a_{k-2}, x|b_{k-2}$
  - k-3: Either $(a_{k-3},b_{k-3})=(3x,x)$ or $(a_{k-3},b_{k-3})=(2x,2x)$ or … $\Rightarrow x|a_{k-3}, x|b_{k-3}$
  - …

Proof (sketch) 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;
```

- 2^{nd} step: We prove that no common divisor greater than x can exist
  - Assume any 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
  An algorithm is called terminating if it stops after a finite number of steps for every finite input
  - We so-far required that the algorithm (specification) is finite; here we require that the execution is finite
- Definition
  An algorithm is called deterministic if it always performs the same series of steps given the same input

- We only study terminating and mostly only deterministic algs
  - Operating systems are “algorithms” that do not terminate
  - Algs which at some point randomly decide about the next step are not deterministic (nondeterministic)

Algorithms and Runtimes

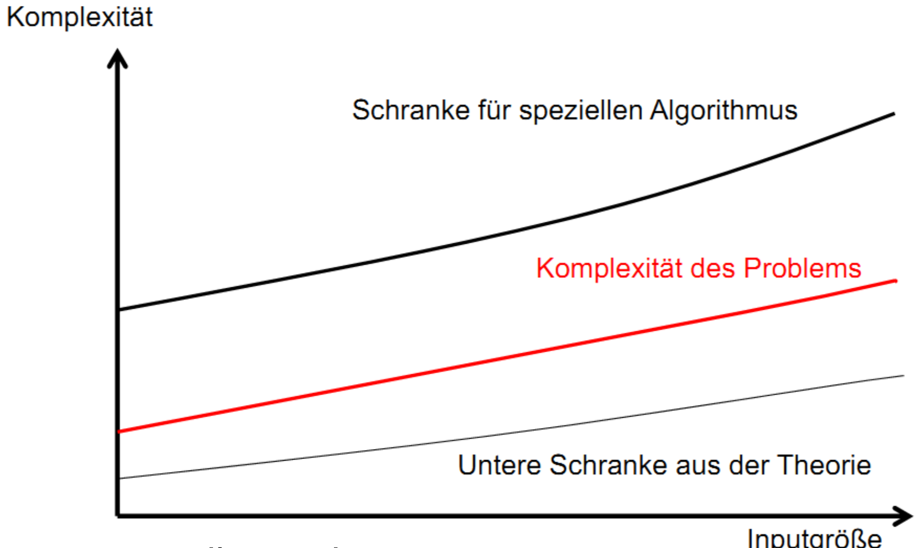
- Usually, one seeks efficient (read for now: fast) algorithms
- We will analyze the efficiency of an algorithm as a function of the size of its input; this is called its (time-)complexity
  - Selection-sort has time-complexity "$O(n^2)$"
- The real runtime of an algorithm on a real machine depends on many additional factors 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
  - Precise definition follows

Algorithms, Complexity and Problems

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

Relationships

- There are problems for which we know their complexity, but no optimal algorithm is known
- There 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 bounds 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. Time consumption – how many operations will it need?
   - Time complexity
   - Worst-case, average-case, best-case
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

Formal Analysis versus Empirical Analysis

- In this lecture, we usually perform a complexity analysis of the algorithms we study
  - Goal: Derive a simple formula which helps to compare the general runtime behavior of different algorithms
  - Should correlate with the true runtime on any machine
    - In some yet-to-be-defined sense
  - However, this doesn't help to decide which of 10 sorting algorithms with complexity O(n*log(n)) are actually the fastest for your setting
    - Machine, nature and amount of data to be sorted, …
- Alternative: Implement carefully and run on reference machine using reference data set
  - Done a lot in practical algorithm engineering
  - Not so much in this introductory course

In this Module

- 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

Small quiz

- Which of the following statements is correct (0-5)
  - Recipes or process descriptions are very similar to algorithms
  - An operating system is an algorithm
  - A deterministic algorithm always performs the same sequence of operations, irrespective of the input
  - It is impossible to improve space and time of an algorithm at the same time
  - The average case complexity of an algorithm is never worse than its worst-case complexity

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 result data
- A data structure is the 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 algorithms is tightly bound to the data structures 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[i];
      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.  if i.next = null
3.    return;
4.  repeat
5.    j := i.next;
6.    repeat
7.      if i.val > j.val then
8.        tmp := i.val;
9.        i.val := j.val;
10.       j.val := tmp;
11.     end if;
12.     j = j.next;
13.   until j.next = null;
14.   i := i.next;
15. until i.next.next = null;
```

- How much do the algorithm's steps cost now?
  - Assume following/comparing a pointer costs c'
    - 1: One assignment
    - 2: One comparison
    - 5: One assignment, n-1 times
    - 7: One comparison, … times
    - …
- Apparently no change in complexity
  - Why? Only sequential access

Example Continued

```
i := p0;
if i.next = null
  return;
repeat
  j := i.next;
  repeat
    if i.val > j.val then
      tmp := i.val;
      i.val := j.val;
      j.val := tmp;
    end if;
    j = j.next;
  until j.next = null;
  i := i.next;
until i.next.next = null;
```

- 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 have a big impact in practice
  - In this case especially regarding space

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

Exemplary Questions

- Give a definition of the concept “algorithm”
- What different types of complexity exist?
- Given the following algorithm …, analyze its worst-case time complexity
- The following algorithm … uses a double-linked list as basic set data structure. Replace this with an array
- When do we say an algorithm is optimal for a given problem?
- How does the complexity of an algorithm depend on (a) the data structures it uses and (b) the complexity of the problem it solves?