Algorithms and Data Structures

Sorting:
Simple Methods and a Lower Bound

Ulf Leser
This Course

- Introduction 2
- Abstract Data Types 1
- Complexity analysis 1
- Styles of algorithms 1
- Lists, stacks, queues 1
- **Sorting (lists)** 3
- Searching (in (sorted) lists) 4
- Hashing (to manage lists) 2
- Trees (to manage lists) 4
- Graphs (no lists!) 5
- Sum 6/24
Large-Scale Sorting

• Imagine you are the IT head of a telco-company
• You have 30.000.000 customers each performing ~100 telephone calls per months, each call creating 200 bytes
  - That’s 30M*100*12*200=7.200.000.000.000 bytes per year
  - Somewhere in the 200 bytes is information on revenue per call
  - Imagine the data is in one file, one line per call
• At the end of the year, management wants a list of all customers with **aggregated revenue per day** (for one year)
  - That’s ~30M*12*30 ~ 10.000.000.000 real numbers
• Problem: How can we compute these 10E9 numbers?
Approach 0a: Load into Memory and Scan

- This won’t work
- Data is too big to be loaded into main memory
Approach 0b: Load into a DBMS and use SQL

- This will work
- Not topic of our lecture

- [Will be slow – inserting is costly]
- [DBMS will use the same trick we present right now]
Approach 1: Scan and Keep Intermediate Results

- Eventually, we need 10E9 real numbers
- Scan the file from start to end
  - Build a table (how?) on every combination of customer and day
  - When reading a record, look-up combination in table and update
- That’s fast (if the table-look-up is fast)
- But we need ~64GB
- What if want the sum for each day over 10 years?
- This won’t scale
Approach 2: Partition Data, Multiple Reads

- Assume we can keep $30M \times 30 \sim 1E9$ numbers in memory
  - Solve the problem month-by-month
  - Read the call-file 12 times, each time computing aggregates for all customers and the days of one month
  - This will be slow

<table>
<thead>
<tr>
<th>1st read</th>
<th>2nd read</th>
<th>3rd read</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meier, 10.1.2010</td>
<td>Meier, 10.1.2010</td>
<td>Meier, 10.1.2010</td>
<td>...</td>
</tr>
<tr>
<td>Müller, 18.4.2010</td>
<td>Müller, 18.4.2010</td>
<td>Müller, 18.4.2010</td>
<td>...</td>
</tr>
<tr>
<td>Meier, 1.2.2010</td>
<td>Meier, 1.2.2010</td>
<td>Meier, 1.2.2010</td>
<td>...</td>
</tr>
<tr>
<td>Meier, 18.1.2010</td>
<td>Meier, 18.1.2010</td>
<td>Meier, 18.1.2010</td>
<td>...</td>
</tr>
<tr>
<td>Schmidt, 14.1.2010</td>
<td>Schmidt, 14.1.2010</td>
<td>Schmidt, 14.1.2010</td>
<td>...</td>
</tr>
<tr>
<td>Schmidt, 6.4.2010</td>
<td>Schmidt, 6.4.2010</td>
<td>Schmidt, 6.4.2010</td>
<td>...</td>
</tr>
<tr>
<td>Müller, 27.2.2010</td>
<td>Müller, 27.2.2010</td>
<td>Müller, 27.2.2010</td>
<td>...</td>
</tr>
<tr>
<td>Müller, 9.4.2010</td>
<td>Müller, 9.4.2010</td>
<td>Müller, 9.4.2010</td>
<td>...</td>
</tr>
<tr>
<td>Schmidt, 1.3.2010</td>
<td>Schmidt, 1.3.2010</td>
<td>Schmidt, 1.3.2010</td>
<td>...</td>
</tr>
<tr>
<td>Schmitt, 9.2.2010</td>
<td>Schmitt, 9.2.2010</td>
<td>Schmitt, 9.2.2010</td>
<td>...</td>
</tr>
<tr>
<td>Schmitt, 30.3.2010</td>
<td>Schmitt, 30.3.2010</td>
<td>Schmitt, 30.3.2010</td>
<td>...</td>
</tr>
<tr>
<td>Schmitt, 3.1.2010</td>
<td>Schmitt, 3.1.2010</td>
<td>Schmitt, 3.1.2010</td>
<td>...</td>
</tr>
</tbody>
</table>
Approach 3: Sorting

• Alternative?
  - Sort the file by customer and day
  - Read sorted file once and compute aggregates on the fly
  - Whenever a pair (day, customer) is finished (i.e., new values appear), sum can be written out and next day/customer starts
  - This will be very fast
  - Needs virtually no memory during counting

• But: Can we sort ~3 billion records using less than 12 reads?
Content of this Lecture

- Sorting
- Simple Methods
- Lower Bound
Sorting

• Assumptions
  - We have n values (integer, called keys) that should be sorted
  - Values are stored in an array $S$ (i.e., $O(1)$ access to i’th element)
  - Comparing two values costs $O(1)$
  - We usually count # of comparisons; sometimes also # of swaps
  - Values are not interpreted
    • We do not know what a “big” value is or how many percent of all values are smaller than a given value or …
    • All we can do is compare two values
  
• We seek a permutation $\pi$ of the indexes of $S$ such that
  $$\forall i, j \leq n \text{ with } \pi(i) < \pi(j) : S[\pi(i)] \leq S[\pi(j)]$$
Variations

• **External versus internal sorting**
  - Internal sorting: S fits into main memory
  - External sorting: There are too many records to fit in memory
  - We only look at internal sorting (see DB lecture)

• **In-place or with additional memory**
  - In-place sorting only requires a constant (independent of n) amount of additional memory (on top of S)
  - We will look at both

• **Pre-Sorting**
  - Some algorithms can take advantage of an existing (incomplete, erroneous) order in the data, some not
  - We will not exploit pre-sorting
Applications

- Sorting is a **ubiquitous** task in computer science
  - [OW93] claims that 25% of all computing time is spent in sorting

- Second example: Information Retrieval
  - Imagine you want to build g*****++
  - Fundamental operation: In a very large set of documents, find those that contain a given **set of keywords**
    - [Note: That’s not what a search engine does!]
  - Popular way of doing this: Build an **inverted index**
# Inverted Index

<table>
<thead>
<tr>
<th>ID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Baseball is played during summer months.</td>
</tr>
<tr>
<td>2</td>
<td>Summer is the time for picnics here.</td>
</tr>
<tr>
<td>3</td>
<td>Months later we found out why.</td>
</tr>
<tr>
<td>4</td>
<td>Why is summer so hot here?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Freq</th>
<th>Document ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseball</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>during</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>found</td>
<td>1</td>
<td>[3]</td>
</tr>
<tr>
<td>here</td>
<td>2</td>
<td>[2], [4]</td>
</tr>
<tr>
<td>hot</td>
<td>1</td>
<td>[4]</td>
</tr>
<tr>
<td>is</td>
<td>3</td>
<td>[1], [2], [4]</td>
</tr>
<tr>
<td>months</td>
<td>2</td>
<td>[1], [3]</td>
</tr>
<tr>
<td>summer</td>
<td>3</td>
<td>[1], [2], [4]</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
<td>[2]</td>
</tr>
<tr>
<td>why</td>
<td>2</td>
<td>[3], [4]</td>
</tr>
</tbody>
</table>

Source: [http://docs.lucidworks.com](http://docs.lucidworks.com)
Answering a IR-style Query

- A **query** is a set of keywords
- Finding the answer
  - For each keyword $k_i$ of the query, load list $d_i$ of docs containing $k_i$ from inverted index
  - Build intersection of all $d_i$
  - Docs in this list are your answer
- Imagine the query “the man eats a bread” on the Web
  - Doc-list for “the” and “a” will contain $>10$ billion documents
- How do we compute the **intersection of two sets** of 10 billion IDs?
Intersection of Two Sets

With non-sorted sets: $O(m\times n)$

With sorted sets: $O(n+m)$
Content of this Lecture

- Sorting
- **Simple Methods**
  - Selection sort
  - Insertion sort
  - Bubble sort
- Lower Bound
Recall: Selection Sort

- Analysis showed that selection sort is in $O(n^2)$
- It is easy to see that selection sort also is in $\Omega(n^2)$
- How often do we swap values?
  - That depends a lot on the pre-sorted’ness of the array
  - But actually we can do a bit better

```plaintext
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[j] := S[i];
      S[i] := tmp;
    end if;
  end for;
end for;
```
Selection Sort Improved

- How often do we swap values?
  - Once for every position
  - Thus: \( O(n) \) swaps
  - But more (cheaper) assignments

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

• Let’s assume you keep your cards sorted
• How to get this order?
  – Selection sort: Take up all cards at once and build **sorted prefixes** of increasing length
  – Insertion sort: Take up cards one by one and **sort every new card** into the sorted subset in your hand
  – Bubble sort: Take up all cards at once and **swap neighbors** until everything is fine
Insertion Sort

- After each loop of i, the prefix S[1..i] of S is sorted
- While-loop runs backwards from current position (to be inserted) until values get too small (smaller than S[j])
- Example: 5 4 8 1 6
- One problem is the required movement of many values until correct place is found
  - Could be implemented much better with a double-linked list

```
S: array_of_names;
n := |S|
for i = 2..n do
  j := i;
  key := S[j];
  while (S[j-1]>key) and (j>1) do
    S[j] := S[j-1];
    j := j-1;
  end while;
  S[j] := key;
end for;
```
Complexity (Worst Case)

- Comparisons
  - Outer loop: \( n \) times
  - Inner-loop: \( i \) times
  - Thus, \( O(n^2) \)

- How many swaps?
  - (We move and don’t swap, but both are in \( O(1) \))
  - In worst-case, every comparison incurs a swap
  - Thus: \( O(n^2) \)

- We got worse?
Complexity (Best Case)

- Assume the best case: \( S \) is already sorted
- Comparisons
  - Outer loop: \( n \) times
  - Inner-loop: 1 time
  - Thus, \( O(n) \)
- Swaps
  - None
- We might be better!
Bubble Sort

- Go through array again and again
- Compare all direct neighbors
- Swap if in wrong order
- Repeat until a loop finishes without a single swaps
- Analysis: About as good/bad as the others (so far)
  - Worst case $O(n^2)$ comparisons and $O(n^2)$ swaps
  - Best case $O(n)$ comparisons and 0 moves / swaps

Source: HKI, Köln
# Summary

<table>
<thead>
<tr>
<th></th>
<th>Comparisons worst case</th>
<th>Comparisons best case</th>
<th>Additional space</th>
<th>Moves worst/best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(n)^*$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2) / O(n)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2) / O(1)$</td>
</tr>
</tbody>
</table>

*: Key assignments
## Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparisons worst case</th>
<th>Comparisons best case</th>
<th>Additional space</th>
<th>Moves worst/best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(n)^*$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2) / O(n)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2) / O(1)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \cdot \log(n))$</td>
<td>$O(n \cdot \log(n))$</td>
<td>$O(n)$</td>
<td>$O(n \cdot \log(n))$</td>
</tr>
</tbody>
</table>
## Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparisons worst case</th>
<th>Comparisons best case</th>
<th>Additional space</th>
<th>Moves worst/best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$O(n)^*$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$ / $O(n)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n^2)$ / $O(1)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n*\log(n))$</td>
<td>$O(n*\log(n))$</td>
<td>$O(n)$</td>
<td>$O(n*\log(n))$</td>
</tr>
<tr>
<td>Magic Sort (?)</td>
<td>$O(n)$</td>
<td></td>
<td></td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Content of this Lecture

- Sorting
- Simple Methods
- Lower Bound
Lower Bound

- We found three algorithms with WC-complexity $O(n^2)$
- Maybe there is no better algorithm?
- There are some in $O(n \log(n))$
- Maybe there are even better algorithms?

- Is there a lower bound on the number of comparisons?
Lemma

To sort a list of $n$ distinct keys using only key comparisons, every algorithm needs $\Omega(n \cdot \log(n))$ comp’s in worst case.

Implications

- We cannot sort with less than $O(n \cdot \log(n))$ comparisons.
- Still, different algorithms with $O(n \cdot \log(n))$ may exhibit different real runtimes.
- We can be better, when other operations than comparisons are allowed – see radix sort.
Proof Structure

• We find the best way to find the right permutation $\pi$
• There are $n!$ different permutations
• Each could be the right one
  – And there is only one “right one”
• To find the right one, we may only compare two keys
• Every comparison we do splits the group of all permutations into two disjoint partitions
  – One with all permutations where the result of the test is TRUE
  – One with all permutations where the result of the test is FALSE
• How often do we need to compare at least such that every partition eventually has size 1
  – At least: In the best of all worlds
Some exemplary permutations (columns) of an arbitrary list $S$ with $|S|=9$
General Case

Decision Tree

All permutations of S where the value at position $i_1$ is before the value at position $j_1$

All permutations of S where the value at position $i_1$ is after the value at position $j_1$
Decision Tree

\[
S[i_1] < S[j_1]?
\]

\[
\begin{array}{cccc}
1 & 8 & 6 & 3 \\
5 & 3 & 7 & 1 \\
9 & 6 & 1 & 5 \\
4 & 4 & 3 & 6 \\
7 & 2 & 5 & 8 \\
2 & 7 & 4 & 9 \\
3 & 1 & 8 & 4 \\
6 & 5 & 9 & 1 \\
8 & 9 & 5 & 2 \\
\end{array}
\]

\[
S[i_2] < S[j_2]?
\]

\[
\begin{array}{cccc}
5 & 9 & 3 & 1 \\
8 & 3 & 6 & 7 \\
3 & 2 & 4 & 8 \\
1 & 6 & 8 & 3 \\
4 & 5 & 9 & 2 \\
9 & 8 & 2 & 9 \\
7 & 7 & 1 & 5 \\
1 & 4 & 7 & 4 \\
6 & 1 & 5 & 3 \\
\end{array}
\]

\[
S[i_6] < S[j_6]?
\]
Decision Tree

S[i_1] < S[j_1]?

S[i_2] < S[j_2]?

S[i_6] < S[j_6]?

Non-optimal choice of i_1, j_1
Full Decision Tree

S[i₁] < S[j₁]?

S[i₂] < S[j₂]?

S[i₆] < S[j₆]?

3 1 5 6 8 9 4 1 2

6 7 1 3 5 4 8 9 5

1 5 9 4 7 2 3 6 8

8 3 6 4 2 7 1 5 9
Optimal Sequence of Comparisons

• We have no clue about which concrete series of comparisons is optimal for a given list
• But: Here we are looking for a lower bound: We may always assume to take the best choice
• Best choice: Creating all 1-partitions with as few comparisons as possible
• Thus, we want to know the length of the longest path through the optimal (lowest) decision tree
  – Even in the best of all worlds we may need to make this number of comparisons to find the correct permutation
• The optimal tree is the one with the shortest longest path
Intuition

Good (not optimal)

Bad
Shortest Longest Path

• Definition
  *The height of a binary tree is the length of its longest path.*

• Lemma
  *A binary tree with k leaves has at least height log(k).*

• Proof
  - Every inner node has at most two children
  - To cover as many leaves as possible in the level above the leaves, we need ceil(k/2) nodes
  - In the second-last level, we need ceil(k/2/2) nodes
  - Etc.
  - After log(k) levels, only one node remains (root)
  - qed.
Putting it all together

- Our decision tree has $n!$ leaves.
- The height of a binary tree with $n!$ leaves is at least $\log(n!)$.
- Thus, the longest path in the optimal tree has at least $\log(n!)$ comparisons.
- Since $n! \geq (n/2)^{n/2}$: $\log(n!) \geq \log((n/2)^{n/2}) = n/2 \times \log(n/2)$.
- This gives the overall lower bound $\Omega(n \times \log(n))$.
- qed.

![Decision Tree Diagram]
Stop: Why not test in O(n)?

- This is the best case – not the best worst case
- In general, the solution will not be in this partition
- We need a strategy that is always fast, not “faster” in some cases
Exemplary Exam Questions

• Give best case and worst case instances for the following algorithms: insertion sort, bubble sort. Explain your examples
• Proof that bubble sort is in $O(n^2)$ and $\Omega(n^2)$ worst case (comparisons)
• Image a list $S$ consisting of $k$ sorted subarrays of arbitrary size (example for $k=4$: $<1,6,7,8,2,5,1,5,7,9,3,5>$). Find an algorithm for sorting $S$ which runs in $O(n*k)$