Algorithms and Data Structures

Sorting:
Simple Methods and a Lower Bound

Ulf Leser
This Course

- Introduction
- Abstract Data Types
- Complexity analysis
- Styles of algorithms – Part I
- Lists, stacks, queues
- Sorting (lists)
- Searching (in (sorted) lists)
- Hashing (to manage lists)
- Trees (to manage lists)
- Graphs (no lists!)
- Styles of algorithms – Part II
This Course

- Introduction 2
- Abstract Data Types 1
- Complexity analysis 1
- Styles of algorithms – Part I 1
- Lists, stacks, queues 3
- Sorting (lists) 2
- Searching (in (sorted) lists) 3
- Hashing (to manage lists) 2
- Trees (to manage lists) 3
- Graphs (no lists!) 4
- Styles of algorithms – Part II 2
- Sum 8/25
Large-Scale Sorting Problem

• 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
  – Imagine the data is in a file, one line per call containing all data
• At the end of the year, management wants list of all customers with aggregated revenue per day
  – That’s ~10,000,000,000 real numbers
• Problem: How can we compute these 10,000,000,000 numbers?
Approach: Multiple Reads

- Assume we can keep 1,000,000,000 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
  - That will be slow
Approach: 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

• But: Can we sort 3 billion records with 8GB memory using less than 12 reads?
Side Notice

- Management won’t look at all of the data – **Top-100** customer/days is enough
  - Let’s concentrate on our best customers
- Second problem: Compute the **Top-100 of 10 billion** sums
- Approach 1
  - Sort file
  - Take first 100
- Is there a **better way**?
  - Yes – go once through aggregated values and always keep top-100 in memory
Content of this Lecture

• Sorting
• Simple Methods
• Lower Bound
• Merge Sort
Sorting

• Assumptions
  – We have n values that should be sorted
  – Values are stored in an array S (i.e., O(1) access to n’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 probably smaller than a given value
  – All we can do is comparing two values

• We seek a permutation $\pi$ of the indexes of S such that
  $\forall i,j \leq n$ with $\pi(i) < \pi(j) : S[\pi(i)] \leq S[\pi(j)]$
Variations of the Problem

• **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

• **In-place or with additional memory dependent on n**
  – In-place sorting requires no more memory than S (except a constant number of variables)
  – 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 times is spent in sorting
• Second example: Information Retrieval
  – Imagine you want to build google++
  – Fundamental operation: In a very large set of documents, find those that contain a given **set of keywords**
  – Popular way of doing this: Build an **inverted index**
Inverted Index

- Doc 1: **Certified software** within your **reach** is **fading** away
- Doc 2: The **witch** took a hasty **sniff**.
- Doc 3: ...

<table>
<thead>
<tr>
<th>Word</th>
<th>Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>certified</td>
<td>1, 4, 56, ...</td>
</tr>
<tr>
<td>document</td>
<td>4, 6, 7, ...</td>
</tr>
<tr>
<td>fading</td>
<td>1, 2, 43, ...</td>
</tr>
<tr>
<td>hallow</td>
<td>2, 12, 34, ...</td>
</tr>
<tr>
<td>legal</td>
<td>4, 5, 17, ...</td>
</tr>
<tr>
<td>Potter</td>
<td>4, 42, 121, ...</td>
</tr>
<tr>
<td>reach</td>
<td>1, 3, 4, ...</td>
</tr>
<tr>
<td>sniff</td>
<td>2, 5, 59, ...</td>
</tr>
<tr>
<td>software</td>
<td>1, 12, 13, ...</td>
</tr>
<tr>
<td>witch</td>
<td>2, 3, 90, ...</td>
</tr>
<tr>
<td>wizard</td>
<td>3, 102, 105, ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
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 ~100 billion documents
- How do we compute the **intersection of two sets** of 100 billion IDs?
Intersection of Two Sets

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

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

• Sorting
  • Simple Methods
    – Selection sort
    – Insertion sort
    – Bubble sort
  • Lower Bound
  • Merge Sort
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-sortedness of the array
  – But actually we can do a bit better

```
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

\[
S: \text{array_of_names};
\]
\[
n := |S|
\]
\[
\text{for } i = 1..n-1 \text{ do}
\]
\[
\quad \text{min_pos} := i;
\]
\[
\quad \text{for } j = i+1..n \text{ do}
\]
\[
\quad \text{if } S[\text{min_pos}] > S[j] \text{ then}
\]
\[
\quad \quad \text{min_pos} := j;
\]
\[
\quad \end \text{if};
\]
\[
\quad \end \text{for};
\]
\[
\text{tmp} := S[i];
\]
\[
S[i] := S[\text{min_pos}];
\]
\[
S[\text{min_pos}] := \text{tmp};
\]
\[
\end \text{for};
\]

- How often do we swap values?
  - Once for every position
  - Thus: \( O(n) \)
Analogy

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

- After each iteration of 1, the prefix $A[1..i]$ of $A$ 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 within the area.
  - Could be implemented much better with a double-linked list.

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

- Comparisons (worst-case)
  - Outer loop: n times
  - Inner-loop: n-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 move
  - Thus: $O(n^2)$

- We got worse?

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

- Assume the best case
  - Array is already sorted

- Comparisons
  - Outer loop: n times
  - Inner-loop: 1 time
  - Thus, $O(n)$

- Moves
  - None
  - (But this_key is assigned $O(n)$ times)

- We might be better!

```plaintext
S: array_of_names;
n := |S|
for i = 2..n do
  j := i;
  this_key := S[j];
  while (S[j-1] > this_key) and (j > 1) do
    S[j] := S[j-1];
    j := j-1;
  end while;
  S[j] := this_key;
end for;
```
Bubble Sort

- Go through array again and again
- Compare all direct neighbors
- Swap if in wrong order
- Repeat until first loop without swaps
- Intuitive algorithm
- 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
Content of this Lecture

- Sorting
- Simple Methods
- Lower Bound
- Merge Sort
Lower Bound

• We found three algorithms with WC-complexity $O(n^2)$
• Maybe there is no better algorithm?
• Maybe the problem is $\Omega(n^2)$?

• Let’s see if we can find a lower bound on the number of comparisons
Lemma

To sort a list of $n$ values, every algorithm using only value comparisons will need at least $\Omega(n \log(n))$ comparisons.

Proof structure
- We argue about all possible ways to find the right permutation $\pi$.
- Observe that there are $n!$ different permutations.
- Each of these could be the right one (and there is only one).
- To decide which, we are only allowed to compare two values.
- Every comparison splits the group of all permutations into two disjoint partitions.
- How often do we need to compare such that every partition has size 1 – in the best of all worlds?
Decision Tree

\[ S[i1] < S[j1] ? \]

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

S[i1]<S[j1]?

S[i2]<S[j2]?

S[i6]<S[j6]?

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

5 9 3 1 7
8 3 6 7 1
3 2 4 8 6
1 6 8 3 2
4 5 9 2 5
9 8 2 9 9
7 7 1 5 4
1 4 7 4 5
6 1 5 3 3
Decision Tree

Non-optimal choice of question
Full Decision Tree

$S[i1] < S[j1]?$

$S[i2] < S[j2]?$

$S[i3] < S[j3]?$  

$S[i4] < S[j4]?$

$S[i5] < S[j5]?$

$S[i6] < S[j6]?$

$S[i7] < S[j7]?$
Optimal Set of Comparison

• We have no clue about which concrete series of comparisons is optimal for a given set, but
  – This doesn’t matter, as we are looking for a lower bound
  – We may always assume to take the best choice
• Best choice means: Creating only 1-partitions with as few comparisons as possible
• Thus, we want to know the length of the longest path through the optimal decision tree
  – Even in the best of all worlds we may need to make this number of comparisons to find the correct permutation
• Other way round: The optimal tree is the one with the shortest longest path
Intuition

Good

Bad
Shortest Longest Path

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

- Theorem
  *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 level, we need ceil(k/2/2) nodes, etc.
  - After log(k) levels, only one node remains (root)
  - qed.

- Note that this tree is balanced and complete
Putting it all together

- Our decision tree has $n!$ leaves (all permutations)
- 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]
Content of this Lecture

- Sorting
- Simple Methods
- Lower Bound
- Merge Sort
Central Idea for Improvement

- The methods we analyzed so-far did not optimally exploit transitivity of the "greater-or-equal" relationship
- If $x \leq y$ and $y \leq z$, then $x \leq z$
- If we compared $x$ and $y$ and $y$ and $z$, there often is no need any more to compare $x$ and $z$
- The clue to lower complexities in sorting are ways to exploit such information
Merge Sort

- Given the lower bound, we hope that we can do better
  - Not necessarily: The lower bound does not imply per-se that there is (and that we know) an algorithm which runs in this complexity
- Good news: There are various sort algorithms with $O(n \times \log(n))$ comparisons
- (Probably) Simplest one: Merge Sort
  - Divide-and-conquer algorithm
  - Break array in two partitions of equal size
  - Sort each partition recursively, if it has more than 1 elements
  - Merge sorted partitions
- Merge Sort is not in-place: Requires $O(n)$ additional space
Illustration

Source: WikiPedia
Illustration

Divide - Partition

Conquer - Merge

- Here we exploit transitivity
- We save comparisons during merge because both sub-lists are sorted

Source: WikiPedia
function void mergesort(A array; l,r integer) {
    if (l<r) then
        m := (r-l) div 2;
        mergesort(A, l, m);
        mergesort(A, m+1, r);
        #merges two sorted lists:
        merge(A, l, m, r);
    else
        # Nothing to do, 1-element list
    end if;
}
Merging Two Sorted Lists

- We briefly looked at this problem before: Intersection of two sorted doc-lists in Information Retrieval

- Idea
  - Move one pointer through each list
  - Whatever element is smaller, copy to a new list and increment this pointer
    - “New list” requires additional space
  - Repeat until one list is exhausted
  - Copy rest of other list to new list
Example

\[\begin{array}{c|c|c}
1 & 1 & \rightarrow \\
4 & 2 & \rightarrow \\
7 & 3 & \rightarrow \\
8 & 4 & \rightarrow \\
12 & 7 & \rightarrow \\
\ldots & \ldots & \rightarrow \\
\end{array}\]

\[\begin{array}{c|c|c}
1 & 2 & \rightarrow \\
4 & 2 & \rightarrow \\
7 & 2 & \rightarrow \\
8 & 2 & \rightarrow \\
12 & 2 & \rightarrow \\
\ldots & \ldots & \rightarrow \\
\end{array}\]

\[\begin{array}{c|c|c}
1 & 2 & \rightarrow \\
4 & 2 & \rightarrow \\
7 & 2 & \rightarrow \\
8 & 2 & \rightarrow \\
12 & 2 & \rightarrow \\
\ldots & \ldots & \rightarrow \\
\end{array}\]

\[\begin{array}{c|c|c}
1 & 2 & \rightarrow \\
4 & 2 & \rightarrow \\
7 & 2 & \rightarrow \\
8 & 2 & \rightarrow \\
12 & 2 & \rightarrow \\
\ldots & \ldots & \rightarrow \\
\end{array}\]
function void merge(A array; 
1,m,r integer) {

B: array;
i := l;  # Start 1st list
j := m+1;  # Start 2nd list
k := 1;  # Target list

while (i<=m) and (j<=r) do
   if A[i]<=A[j] then
      B[k] := A[i];  # From 1st list
      i := i+1;
   else
      B[k] := A[j];  # From 2nd list
      j := j+1;
   end if;
   k := k+1;  # Next target
if i>m then  # What remained?
   copy A[j..r] to B[k..k+r-j+1];
else
   copy A[i..m] to B[k..k+m-i+1];
end if;

# Back to place
copy B[1..k] to A[l..r]
end if;
}
Complexity Analysis

• Theorem
  \textit{MergeSort requires }\Omega(n*\log(n)) \textit{ and } O(n*\log(n)) \textit{ comparisons}

• Proof
  – Merging two sorted lists of size n requires \(O(n)\) comps
    • After every comp, 1 element is moved; there are only 2*n elements
  – Merge Sort calls Merge Sort \textit{twice with half} of the array
    • Let \(T(n)\) be the number of comparisons
    • Thus: \(T(n) = T(n/2) + T(n/2) + O(n) \sim 2*T(n/2) + n\)
  – This is \(O(n*\log(n))\)
    • See recursive solution of max subarray – same formula
Remarks

• MergeSort is **worst-case optimal**: Even in the worst of all cases, it does not need more than (in the order of) the minimal number of comparisons
  – Given our lower bound for sorting

• But not the end; **disadvantages**:
  – O(n) additional space
  – Requires \( \Omega(n \cdot \log(n)) \) moves
    • Sorted sub-arrays get copied to new array in any case
    • See Ottmann/Widmayer for proof

• Both disadvantages make MergeSort rather unattractive in practical applications (we can do better)

• Basis for most sort algorithms on **external memory**
## Summary

<table>
<thead>
<tr>
<th>Sort 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>
</tbody>
</table>
What the Future will bring

<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>
<tr>
<td>Merge Sort</td>
<td>$O(n^2 \log (n))$</td>
<td>$O(n^2 \log (n))$</td>
<td>$O(n)$</td>
<td>$O(n^2 \log (n))$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>