HUMBOLDT-UNIVERSITÄT ZU BERLIN

Algorithms and Data Structures

Sorting:
Simple Methods and a Lower Bound

Marius Kloft

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!)
- The End

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
  - Imagine the data is in one file, one line per call
- At the end of the year, management wants list of all customers with aggregated revenue per day
  - That's ~30M*12*30 ~ 10.000.000.000 real numbers
- Problem: How can we compute these 10.000.000.000 numbers?

Approach 0: Load once and scan in memory

- This won‘t work
- Data is too big to load into main memory

Approach 0b: Load into a database and formulate a SQL Query

- This will work
- Not topic of our lecture

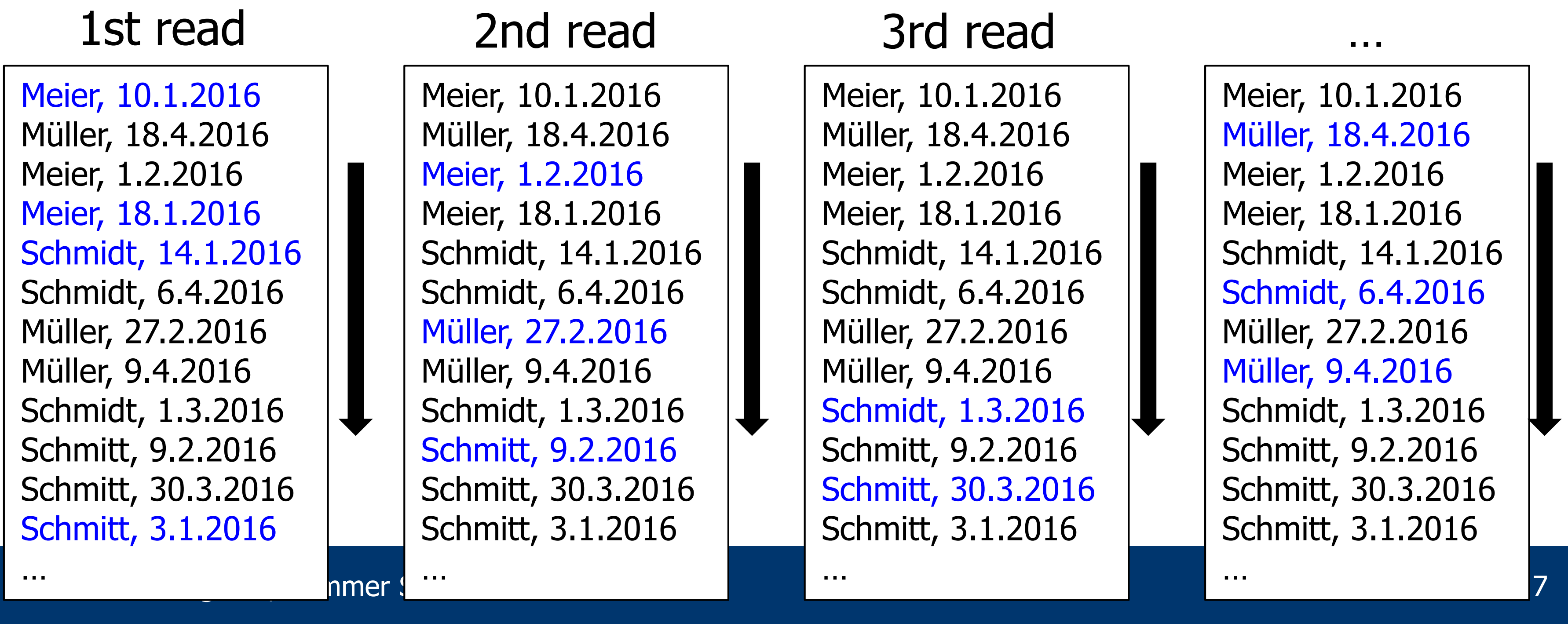
- [Will be slow – inserting is costly]

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 ~80GB
- What if we want values for each day over 10 years?
- This won‘t scale

Approach 2: Multiple Reads

- Assume we can keep 30M*30 ~ 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



1st read

Meier, 10.1.2016
Müller, 18.4.2016
Meier, 1.2.2016
Meier, 18.1.2016
Schmidt, 14.1.2016
Schmidt, 6.4.2016
Müller, 27.2.2016
Müller, 9.4.2016
Schmidt, 1.3.2016
Schmitt, 9.2.2016
Schmitt, 30.3.2016
Schmitt, 3.1.2016
…

2nd read

Meier, 10.1.2016
Müller, 18.4.2016
Meier, 1.2.2016
Meier, 18.1.2016
Schmidt, 14.1.2016
Schmidt, 6.4.2016
Müller, 27.2.2016
Müller, 9.4.2016
Schmidt, 1.3.2016
Schmitt, 9.2.2016
Schmitt, 30.3.2016
Schmitt, 3.1.2016
…

3rd read

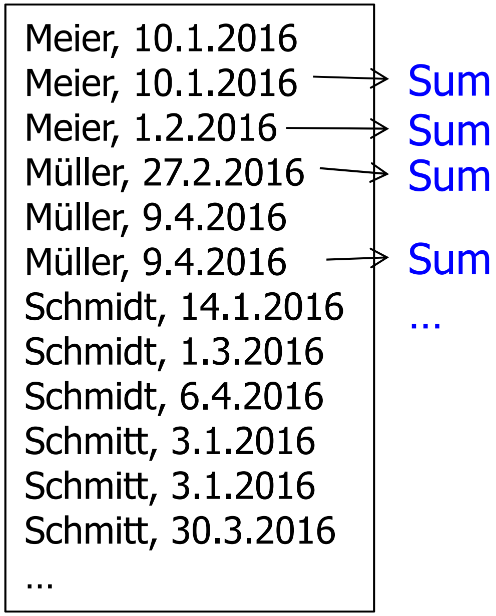
Meier, 10.1.2016
Müller, 18.4.2016
Meier, 1.2.2016
Meier, 18.1.2016
Schmidt, 14.1.2016
Schmidt, 6.4.2016
Müller, 27.2.2016
Müller, 9.4.2016
Schmidt, 1.3.2016
Schmitt, 9.2.2016
Schmitt, 30.3.2016
Schmitt, 3.1.2016
…

…

Meier, 10.1.2016
Müller, 18.4.2016
Meier, 1.2.2016
Meier, 18.1.2016
Schmidt, 14.1.2016
Schmidt, 6.4.2016
Müller, 27.2.2016
Müller, 9.4.2016
Schmidt, 1.3.2016
Schmitt, 9.2.2016
Schmitt, 30.3.2016
Schmitt, 3.1.2016
…

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 ID 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 the call file using less than 12 reads?



Meier, 10.1.2016
Meier, 10.1.2016 → Sum
Meier, 1.2.2016 → Sum
Müller, 27.2.2016 → Sum
Müller, 9.4.2016
Müller, 9.4.2016 → Sum
Schmidt, 14.1.2016
…
Schmidt, 1.3.2016
Schmidt, 6.4.2016
Schmitt, 3.1.2016
Schmitt, 3.1.2016
Schmitt, 30.3.2016
…

Content of this Lecture

- Sorting
- Simple Methods
- Lower Bound

Sorting

- Assumptions
  - We have n values (integer) 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 probably smaller than a given value
  - All we can do is compare two values
- We seek a permutation π 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 into memory
  - We only look at internal sorting
- 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 times is spent on sorting
- Second example: Information Retrieval
  - Imagine you want to build Goo***++
  - 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

ID	Text
1	Baseball is played during summer months.
2	Summer is the time for picnics here.
3	Months later we found out why.
4	Why is summer so hot here?

Term	Freq	Document ids
baseball	1	[1]
during	1	[1]
found	1	[3]
here	2	[2], [4]
hot	1	[4]
is	3	[1], [2], [4]
months	2	[1], [3]
summer	3	[1], [2], [4]
the	1	[2]
why	2	[3], [4]

Source: http://docs.lucidworks.com

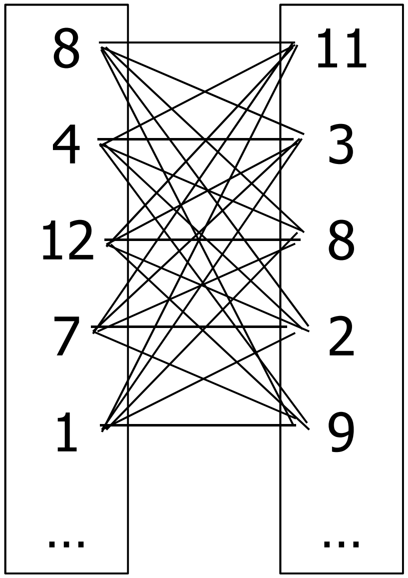
Answering a IR-style Query

- 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

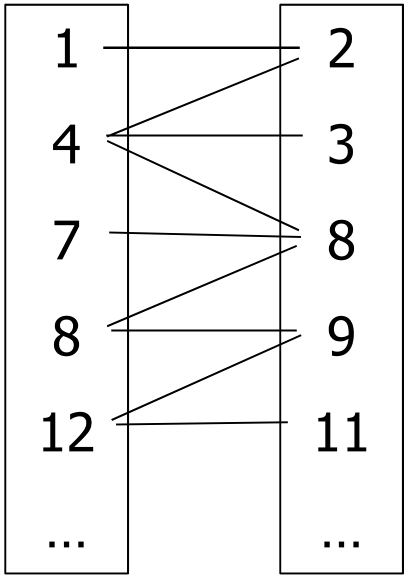
With non-sorted sets:

O(m*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

```
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;
```

- 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

Selection Sort Improved

```
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;
  tmp := S[i];
  S[i] := S[min_pos];
  S[min_pos] := tmp;
end 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 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

```
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;
```

- 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

Complexity (Worst Case)

```
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;
```

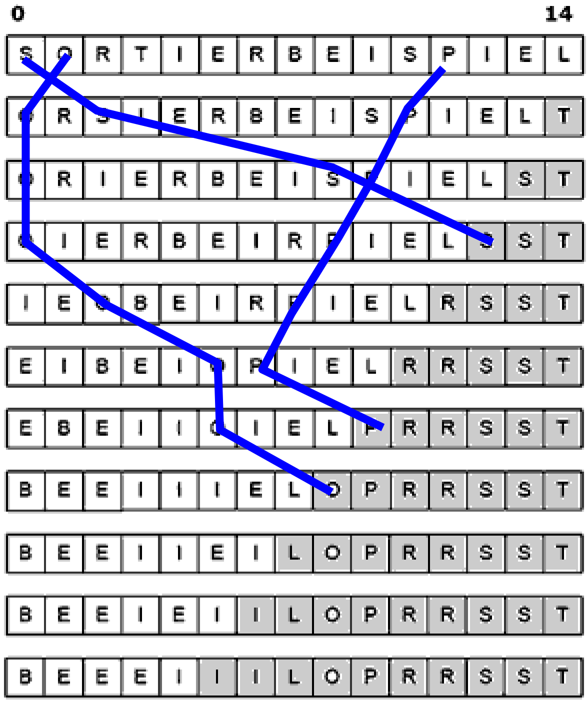
- Comparisons (worst-case)
  - Outer loop: n times
  - Inner-loop: n-i times
  - Thus, $O(n^2)$
- How many swaps?
  - (We move and don't swap, but both are in O(1))
  - In worst-case, every comparison incurs a swap
  - Thus: $O(n^2)$

Complexity (Best Case)

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

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

Bubble Sort



0														14
S	O	R	T	I	E	R	B	E	I	S	P	I	E	L
O	R	S	I	E	R	B	E	I	S	P	I	E	L	T
O	R	I	E	R	B	E	I	S	P	I	E	L	S	T
O	I	E	R	B	E	I	R	P	I	E	L	S	S	T
I	E	O	B	E	I	R	P	I	E	L	R	S	S	T
E	I	B	E	I	O	P	I	E	L	R	R	S	S	T
E	B	E	I	I	O	I	E	L	P	R	R	S	S	T
B	E	E	I	I	I	E	L	O	P	R	R	S	S	T
B	E	E	I	I	E	I	L	O	P	R	R	S	S	T
B	E	E	I	E	I	I	L	O	P	R	R	S	S	T
B	E	E	E	I	I	I	L	O	P	R	R	S	S	T

Source: HKI, Köln

- 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

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?
- Maybe the problem is $\Omega(n^2)$?

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

Lemma

- Lemma
  To sort a list of n distinct values using only comparisons, every algorithm needs $\Omega(n*\log(n))$ comp's in worst case
- Proof structure
  - We argue about all possible ways to find the right permutation π
  - Observe that there are n! different permutations
  - Each could be the right one (and there is only one "right 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?

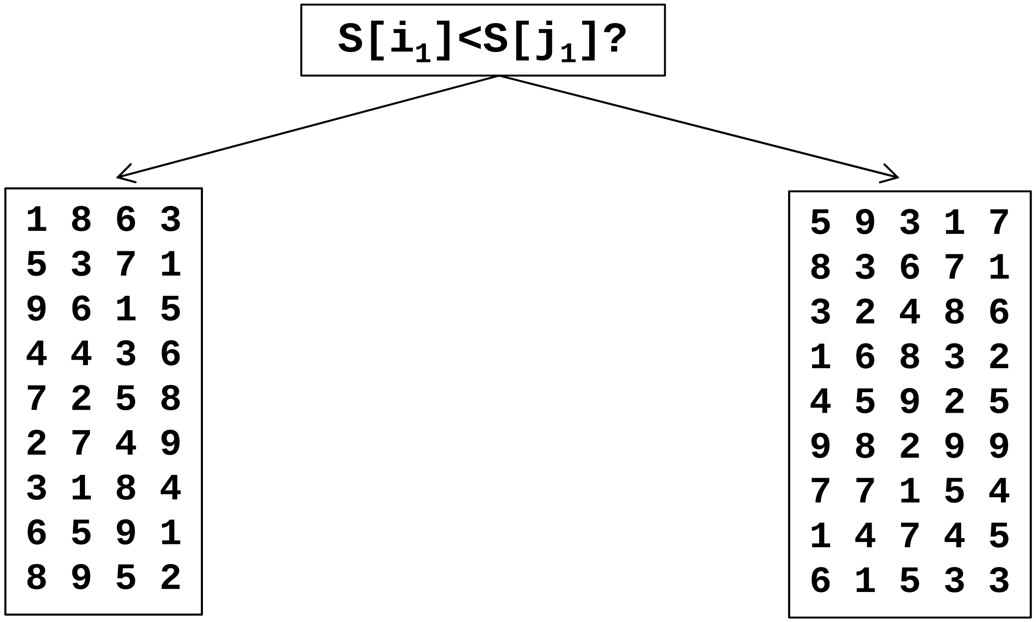
Decision Tree

$S[i_1]<S[j_1]?$

π_1	π_2	π_3	…					
1	8	6	3	5	9	3	1	7
5	3	7	1	8	3	6	7	1
9	6	1	5	3	2	4	8	6
4	4	3	6	1	6	8	3	2
7	2	5	8	4	5	9	2	5
2	7	4	9	9	8	2	9	9
3	1	8	4	7	7	1	5	4
6	5	9	1	1	4	7	4	5
8	9	5	2	6	1	5	3	3

Some exemplary
permutations of an arbitrary
list L with |L|=9

Decision Tree



$S[i_1]<S[j_1]$?

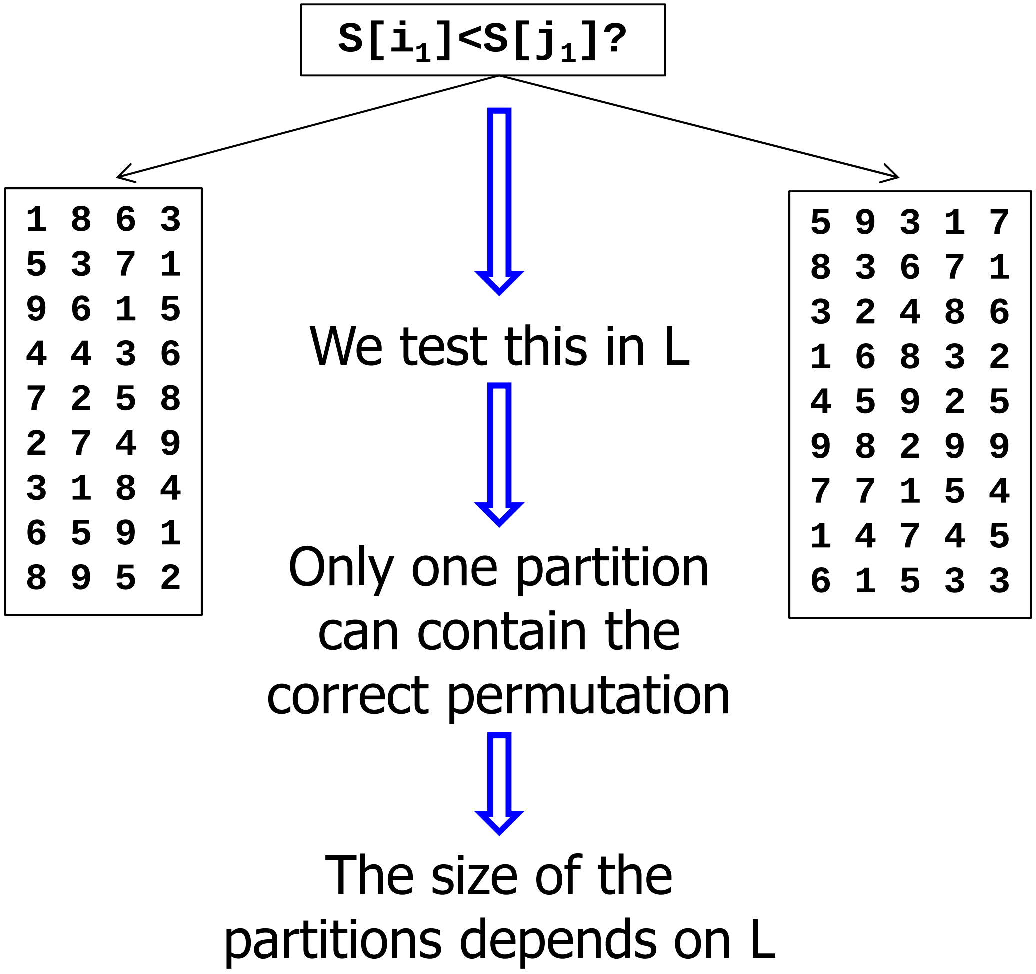
1	8	6	3
5	3	7	1
9	6	1	5
4	4	3	6
7	2	5	8
2	7	4	9
3	1	8	4
6	5	9	1
8	9	5	2

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

All permutations of L where the value at position i_1 stays before the value at position j_1

All permutations of L where the value at position i_1 stays after the value at position j_1

Decision Tree



$S[i_1]<S[j_1]$?

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

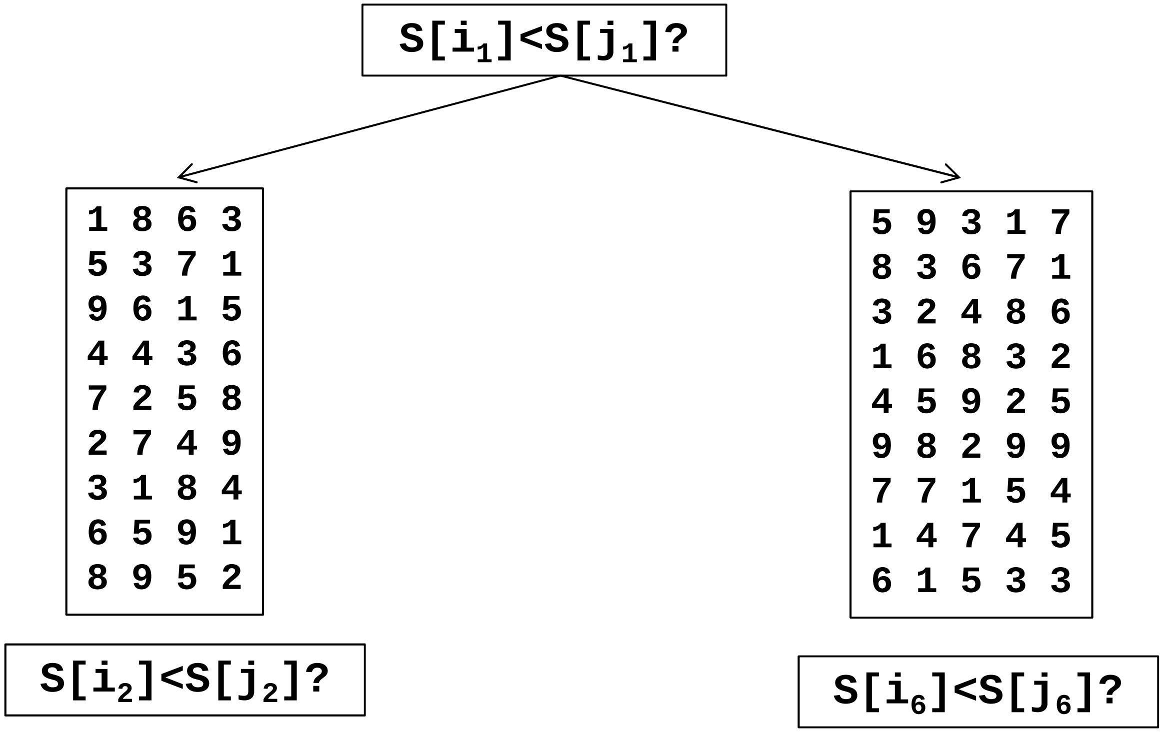
5	9	3	1	7
8	3	6	7	1
3	2	4	8	6
1	6	8	3	2
4	5	9	2	5
9	8	2	9	9
7	7	1	5	4
1	4	7	4	5
6	1	5	3	3

We test this in L

Only one partition can contain the correct permutation

The size of the partitions depends on L

Decision Tree



$S[i_1]<S[j_1]$?

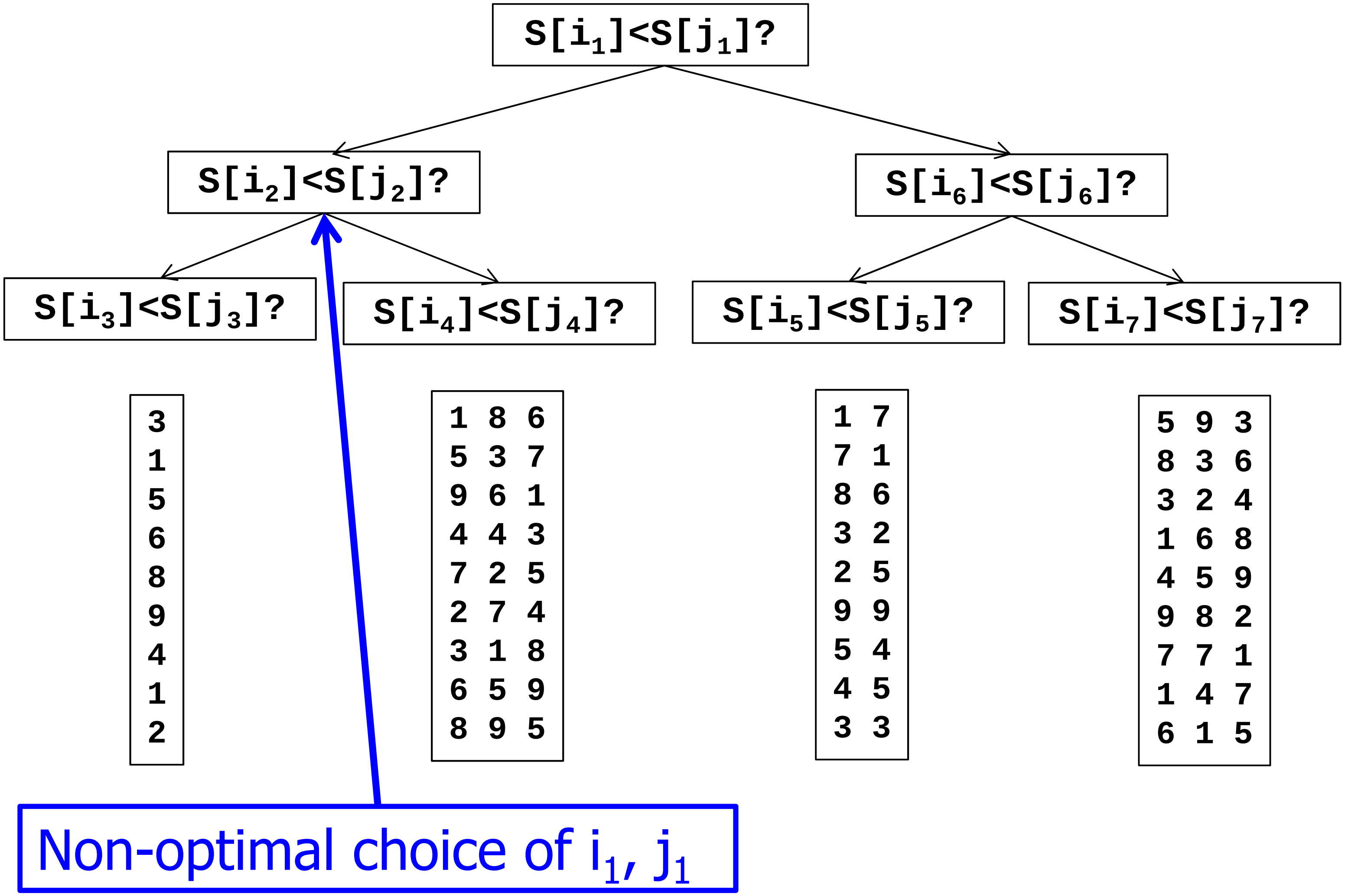
1 8 6 3
5 3 7 1
9 6 1 5
4 4 3 6
7 2 5 8
2 7 4 9
3 1 8 4
6 5 9 1
8 9 5 2

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

$S[i_2]<S[j_2]$?

$S[i_6]<S[j_6]$?

Decision Tree



3
1
5
6
8
9
4
1
2

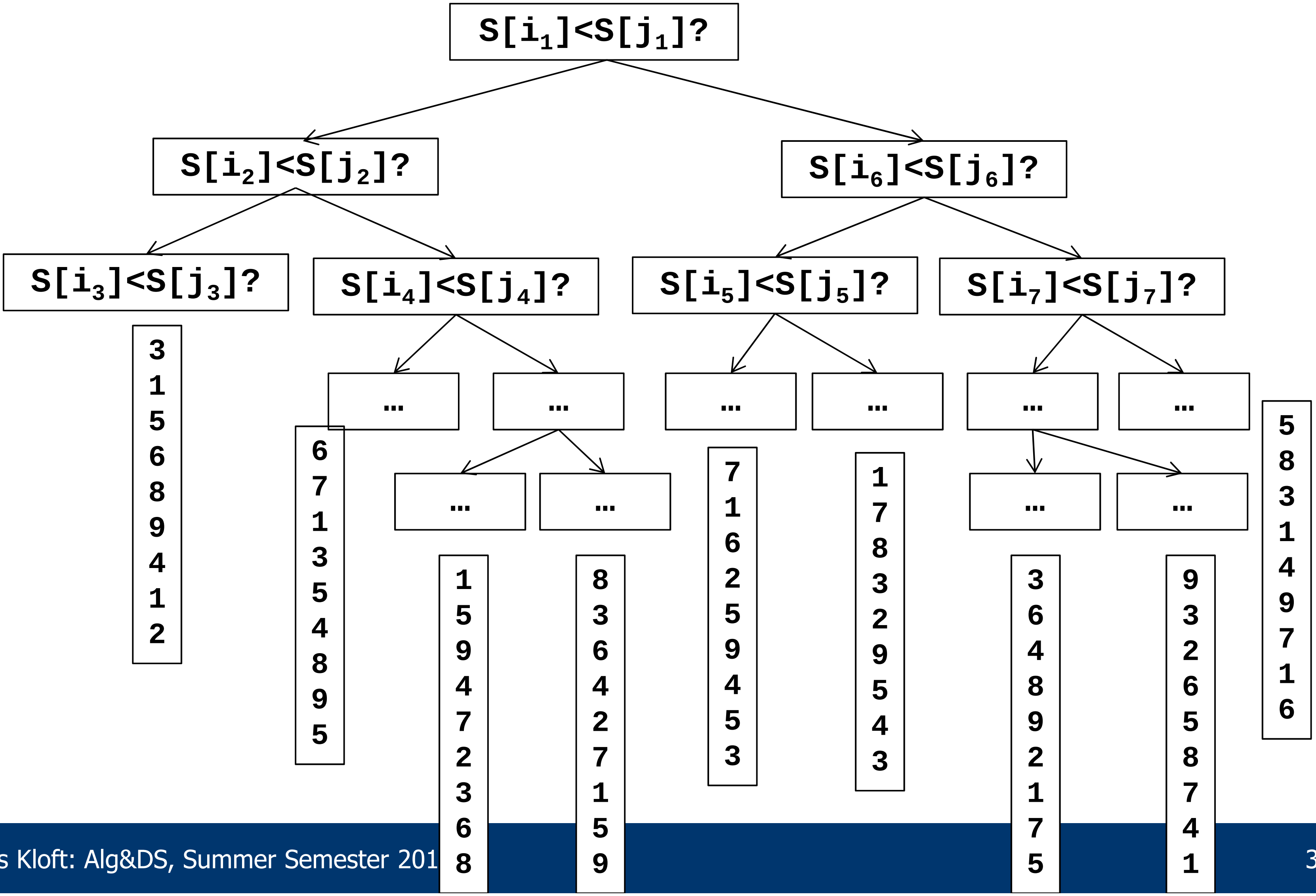
1	8	6
5	3	7
9	6	1
4	4	3
7	2	5
2	7	4
3	1	8
6	5	9
8	9	5

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

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

Non-optimal choice of i_1, j_1

Full Decision Tree

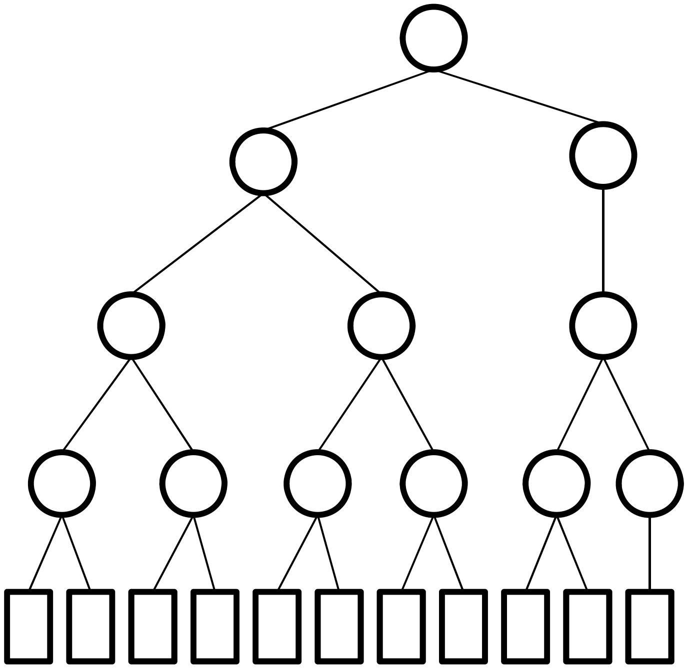


Optimal Sequence of Comparisons

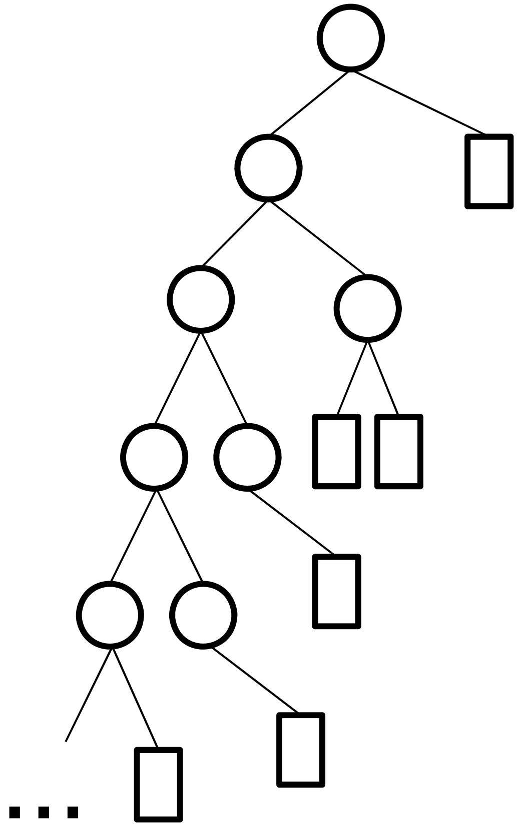
- We have no clue about which concrete series of comparisons is optimal for a given list
- But: Here we are looking for a lower bound
  - We may always assume to take the best choice
- Best choice: Creating 1-partitions with as few comparisons as possible
- Thus, we want to know the length of the longest 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
- The optimal tree is the one with the shortest longest path

Intuition

Good



Bad

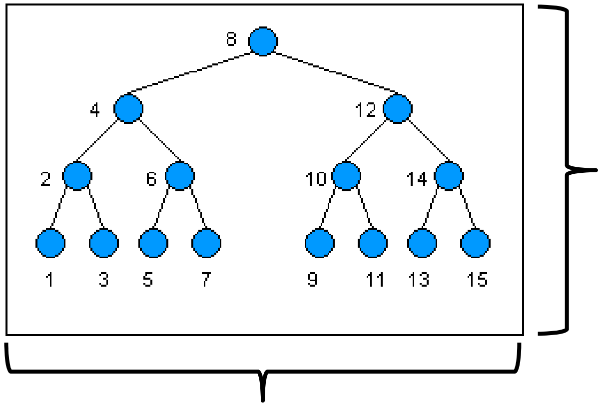


Shortest Longest Path

- Definition
  The height of a binary tree is the length of its longest path.
- 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 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 (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 $n! \geq (n/2)^{n/2}$: $\log(n!) \geq \log((n/2)^{n/2}) = n/2 * \log(n/2)$
- This gives the overall lower bound $\Omega(n*\log(n))$
- qed.



log(n!) in O(n*log(n))

n! leaves

Summary

	Comparisons worst case	Comparisons best case	Additional space	Moves worst/best
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(1)$	$O(n^2)$ / $O(n)$
Bubble Sort	$O(n^2)$	$O(n)$	$O(1)$	$O(n^2)$ / $O(1)$
Merge Sort	$O(n*\log(n))$	$O(n*\log(n))$	$O(n)$	$O(n*\log(n))$