Information Retrieval

WS 2013 / 2014

Lecture 3, Tuesday November 5th, 2013
(Efficient List Intersection)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

UNI FREIBURG

Overview of this lecture

- Organizational
  - Your experiences with Exercise Sheet 2 (Ranking)
  - Some general implementation advice
- List intersection
  - Time-measurement How-To
  - Non-algorithmic improvements
  - Algorithmic improvements: galloping-search intersect
  - Lower bound
  - **Exercise Sheet 3:** implement the galloping-search list intersection and compare to the linear-time one

Experiences with ES2 (ranking)

- Summary / excerpts last checked November 5, 15:00
  - Good exercise to understand ranking + how it works
  - Some Java folks with very long indexing times, reasons:

    Insufficient heap space ... increase with java -Xmx=3g ...

    Search doc id with ArrayList.contains ... oh oh, see Slide 5
  - In general, many programming issues, which have nothing to do with the topic of this course

    Nevertheless: important that you are learning this now!
  - Use formula formatting in PPT ... I don't like PPT formulas
  - Put the annotated slides online after the lecture ... I did !
  - Happy about extensive feedback from tutor

Your results for ES2 (ranking)

- Some interesting observations you made:
  - Time-consuming to find a good query

    But also quite instructive wrt how the ranking works
  - BM25 has a strong preference for shorter documents containing the query words

    This was not such a great advantage here, because all the documents in people.tsv were relatively short
  - In Wikipedia, document length is actually a sign of significance of an article = in our case, the person the article is about

    True! Popularity is another important ingredient of a ranking function, which we did not (yet) talk about

Using functions from a Library

- Yes, you can do that, **but**

  … **you should really know what you are doing!**

  – Example 1: using ArrayList.contains to check if an element is already in the array

  If done for each element added, takes quadratic time !

  For this application, it suffices to look at last element

  – Example 2: using std::set_intersection to implement the linear-time intersect

  Ok, if you convinced yourself that the method is doing the right thing with the right time complexity (cf. above)

  In any case, add a comment that you understood this !

Time measurement 1/3

- In **Java**
  - For **milli**second resolution

```
long start = System.currentTimeMillis();
// whatever code you want to time
long end = System.currentTimeMillis();
long millis = end – start;
```

  - For **micro**second resolution (or maybe better)

```
long start = System.nanoTime();
// whatever code you want to time
long end = System.nanoTime();
long micros = (end – start) / 1000;
```

Time measurement 2/3

- In **C++**
  - For **milli**second resolution

```
include <time.h>
clock_t start = clock();
// whatever code you want to time
clock_t end = clock();
size_t millis = 1000 * (end – start) / CLOCKS_PER_SEC;
```

  - For **micro**second resolution

```
include <sys/time.h>
struct timeval tv;
struct timezone tz;
gettimeofday(&tv, &tz);
size_t startMicros = 1000000L * tv.tv_sec + tv.tv_usec;
…
```

Time measurement 3/3

- Never rely on a single measurements
  - There can be significant variation, for example due to:

    Other jobs running on your machine + JIT compilation

    The Java garbage collector running unpredictably

    Data read from disk in the memory cache or not

    Data read from memory in the L1-cache or not

    Virtual memory addresses in the TBL cache or not
  - For ES3, **repeat** each run **10 times** and take the average

    Note: for small inputs, this can distort the actual truth, because of caching effects. But not a big issue for ES3.

Non-algorithmic improvements 1/3

- Type of array to store the inverted lists
  - **Java:** ArrayList is much worse than native [] array

    Note: elements of an ArrayList cannot be basic data types (e.g. int), but have to be objects (e.g. Integer)
  - **C++:** std::vector is as good as [] with option –O3

    Note: elements of an std::vector can be basic data types as well as objects, unlike in Java

Non-algorithmic improvements 2/3

- Branch prediction
  - This pertains to all conditional parts in your code, in particular, if – then – else parts
  - Modern processors do pipelining = speculative execution of future instructions before the current ones are done
  - For conditional parts they have to guess the outcome
  - So good to **minimize** amount of conditional parts

Non-algorithmic improvements 3/3

- Simple code in loops
  - Within a loop try to keep the number of variables small

    This will allow the compiler to use (fast) registers

    Don't worry about **constants** though, modern compilers figure out that they don't need a variable for those
  - In C++, when you call a function that does something very simple very often, then **inline** it

    Inline = put the code in the header file + precede by the keyword **inline** (the latter is not necessary for short code)

    The compiler will then avoid the function call and instead put a copy of the code of the fct. at each place you call it

Algorithmic improvements 1/4

UNI FREIBURG

- Binary search in the longer list
  - Call the smaller list A, and the longer list B
  - Search each element from A in B, using binary search

    k = #elements in A, n = #elements in B
  - This has time complexity $\Theta(k \cdot \log n)$

$\Theta(\log n)$ time per element of A

13 15 17

A

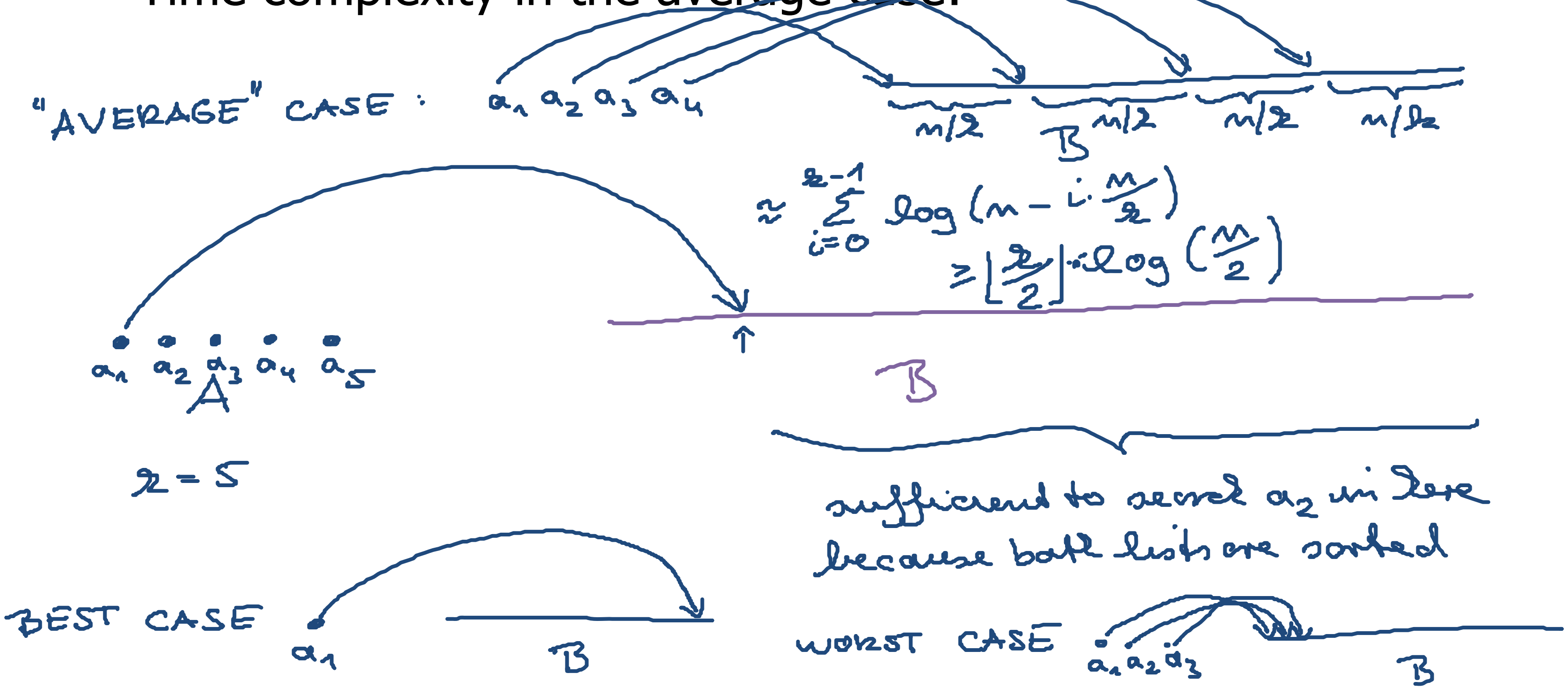
3 5 9 10 11 14 21 27 43

B

NOTE: for $k = n$, this is $\Theta(n \cdot \log n)$, which is worse than the $\Theta(n)$ of the linear-time intersect.

Algorithmic improvements 2/4

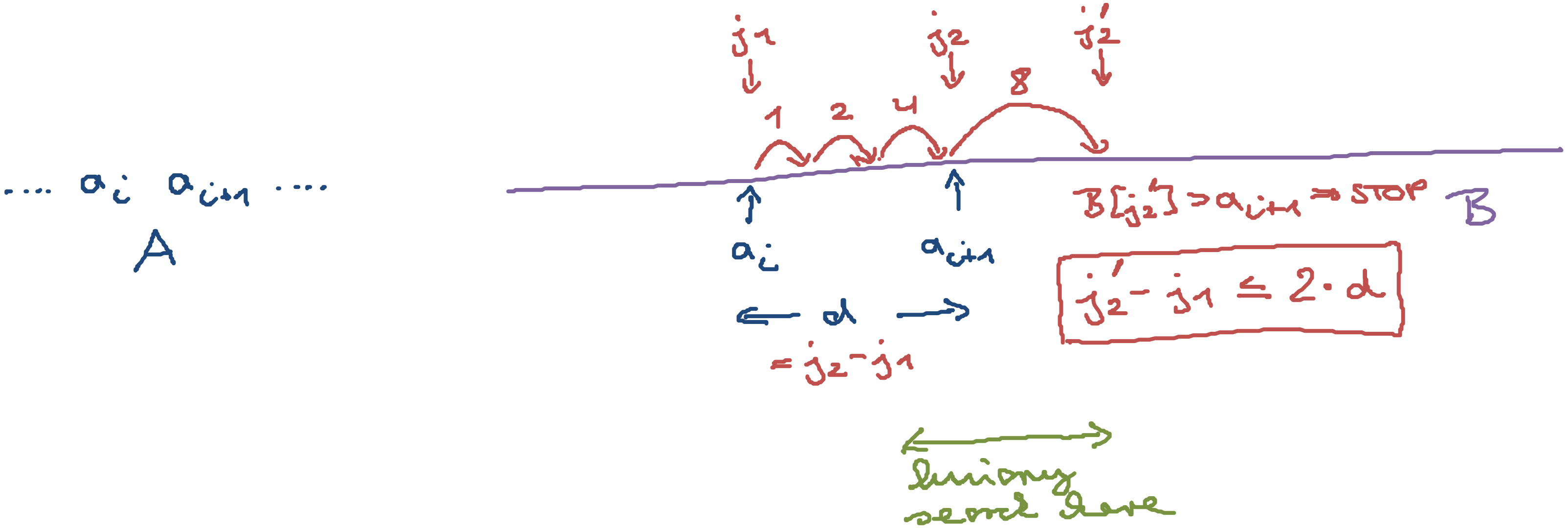
- Binary search only in the remaining part of B
  - Time complexity in the best case: $O(k + \log n)$
  - Time complexity in the worst case: $O(k \cdot \log n)$
  - Time complexity in the average case: $\Omega(k \cdot \log n)$



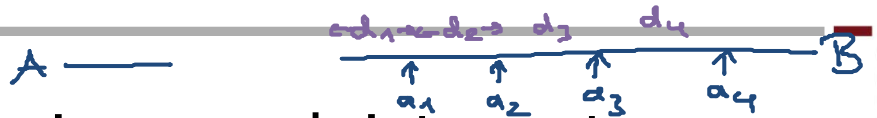
Algorithmic improvements 3/4

- "Galloping" search in B
  - **Goal:** when elements A[i] and A[i+1] are located at positions j_1 and j_2 in B, then, with $d := j_2 - j_1$ ("gap"):

    spend only time **Θ(log d)** to locate element A[i+1]
  - **Idea:** first do an exponential search, to get an upper bound on the range, then a binary search as before



Algorithmic improvements 4/4



- Time complexity of galloping-search intersect
  - Let $d_1, \ldots, d_k$ be the gaps between the locations of the k elements of A in B

    d_1 = from beginning to first location
  - Note that $\Sigma_i\, d_i \leq n$ = the number of elements in B
  - Then the time complexity is $O(\Sigma_i \log d_i)$
  - **Goal:** find a formula that is independent of the d_i
  - **Idea:** maximize $\Sigma_i \log d_i$ under the constraint $\Sigma_i\, d_i \leq n$
  - This is called optimization with side constraints or Lagrangian optimization

    shown by example on the next slide …

Lagrangian Optimization

- Maximize $\Sigma_i \log d_i$ under the constraint $\Sigma_i d_i \leq n$

Let's max $\Sigma_i \ln d_i$, note $\ln \sim \log$

$$\mathcal{L} := \sum_{i=1}^{k} \ln d_i + \lambda \cdot \left(n - \sum_{i=1}^{k} d_i\right)$$

$$\frac{\partial \mathcal{L}}{\partial d_i} = \frac{1}{d_i} - \lambda \overset{!}{=} 0 \implies \lambda = \frac{1}{d_i} \implies \text{all } d_i \text{ equal}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = n - \sum_{i=1}^{k} d_i = 0 \implies \sum_{i=1}^{k} d_i = n \implies d_i = n/k$$

So, we have a local extremum at $d_i = n/k$

$$\frac{\partial^2 \mathcal{L}}{\partial d_i^2} = -\frac{1}{d_i^2} < 0 \implies \text{MAX.}$$

$$\implies \sum_{i=1}^{k} \log d_i \leq k \cdot \log\left(\frac{n}{k}\right)$$

+ also check border cases: one $d_i = n$, all others 0

Skip pointers

- A heuristic approach
  - **Idea:** place skip pointers at "strategic" places in B, to potentially enable skipping large parts of B

    The heuristic part is to decide where these "strategic" places are + how much to skip … see references for details
  - **Advantage:** very simple to implement, in particular, simpler than galloping search

    For ES2, you will implement the galloping search though

Lower bound 1/2

- Let's summarize our upper bounds so far
  - As before, let k = the size of the smaller list, and let n = the size of the larger list
  - Linear-time intersection: $O(k + n)$
  - Galloping-search intersection: $O(k \cdot \log (n/k))$

Lower bound 2/2

3 5 7 (A) 105 107 . . . . . (B)

- Can we do even better than $k \cdot \log(n/k)$?
  - No, at least not for general inputs

    For particular inputs, you can finish in time $O(1)$ … why?
  - Recall the lower bound for comparison-based sorting:

    There are $n!$ possible outputs, we have to differentiate between all of them, and only two choices per step

    Hence #steps required $\geq \log_2(n!) = \Omega(n \cdot \log n)$
  - We can use a similar argument for intersection / union:

    There are n+k over k ways how the k elements from A can be placed within the n elements from B, …

    Hence #steps required $\geq \log_2(n/k)^k = k \cdot \log_2(n/k)$

References

- In the Raghavan/Manning/Schütze textbook

  Section 2.3: Faster intersection with skip pointers

- Relevant Papers

  A simple algorithm for merging two linearly ordered sets

  F.K. Hwang and S. Lin SICOMP 1(1):31–39, 1980

  A fast set intersection algorithm for sorted sequences

  R. Baeza-Yates CPM, LNCS 3109, 31–39, 2004

- Relevant Wikipedia articles

  http://en.wikipedia.org/wiki/Lagrange_multiplier