Information Retrieval

WS 2015 / 2016

Lecture 3, Tuesday November 3rd, 2015
(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 ES 2 — Ranking
  - Using built-in functions — Beware !!
- Contents
  - List Intersection — Recap, Time Measurement
  - Non-algorithmic improvements — Arrays, Branching, Sentinels
  - Algorithmic improvements — Galloping Search, Skip Pointers

Exercise Sheet 3: implement list intersection and make it as fast as possible on a small benchmark we have prepared

Experiences with ES2 1/4

- Summary / excerpts
  - Time-intensive for many, mainly due to debugging

    This should become (much) better with experience
  - Parameter tuning: waiting long for each index build

    Some problems with built-in functions … see slide 5
  - Master solutions for ES1 would have been nice

    Always available in the course SVN under /solutions
  - Feedback from the tutors very much appreciated
  - In Python, 4-space indent + 80-char limit is annoying

    I agree, but it's absolutely standard in Python world

Experiences with ES2 2/4

- Results
  - Standard BM25 parameters gave sub-optimal results

    Smaller **b** worked better (less penalty for longer docs)

    Smaller **k** worked better (less boost for larger tf)
  - Boosting popular documents helped a bit
  - Boosting matches in title helped a bit
  - Boost matches of most or all query words helped a bit
  - Best results: P@3 ≈ 60%, P@R ≈ 40%, MAP ≈ 40%

    The last two results are typical: it's extremely hard to get most or even all relevant documents at the top

    For better P@3, sth like synonyms or query logs needed

Experiences with ES2 3/4

- Built-in functions / library functions
  - Using them is OK, if and only if:

    You are aware of the complexity of the function

    You are aware of the complexity of the code using them

    That complexity is ok for the task at hand

    Not doing this is one of the most common reasons
    for performance leaks in software

Experiences with ES2 4/4

- Built-in functions / library functions
  - Example 1: merging two lists by concatenating them and then sorting the concatenated list

    Takes time n · log n, versus linear time for "zipper" alg
  - Example 2: use "in" or "find" to locate an element in a list, and doing this n times

    Each call to "in" or "find" uses linear time, which gives quadratic time overall → terrible running time
  - Example 3: use std::set_intersection to implement a linear-time intersect

    Ok, provided that you convinced yourself that this works only on sorted lists and runs in linear time

List Intersection 1/4

- Motivation (recap)
  - In Lectures 1 & 2 we have merged the inverted lists

    So that we also have a chance to find relevant docs that do not contain all of the query words
  - For efficiency reasons, many search engines only return results which contain all the query words

    Apache's Lucene, the most widely used open-source search engine, supports intersect (AND) and merge (OR)

    In most applications, intersect is used by default
  - Today we will focus on efficiency and therefore on list intersection

List Intersection 2/4

- Time measurement
  - There can be significant variation, for example due to:

    Other jobs running on your machine

    The Java garbage collector running unpredictably

    Data is partly in disk cache / L1-cache / TBL cache
  - Therefore, always repeat your time measurements, and take the average over all these

    For ES3, **repeat 10 times** for each measurement

    Note: repetition itself can also distort the truth because of caching effects ... but not an issue for us today

List Intersection 3/4

- Time measurement in **Java**
  - For **milli**second resolution

```
long time1 = System.currentTimeMillis();
// whatever code you want to time
long time2 = System.currentTimeMillis();
long millis = time2 – time1;
```

  - For **micro**second resolution

```
long time1 = System.nanoTime();
// whatever code you want to time
long time2 = System.nanoTime();
long micros = (time2 – time1) / 1000;
```

List Intersection 4/4

- Time measurement in **C++**
  - For **milli**second resolution (C-Style) #include <time.h>

```
clock_t time1 = clock();
// whatever code you want to time
clock_t time2 = clock();
size_t millis = 1000 * (time2 – time1) / CLOCKS_PER_SEC;
```

  - For **micro**second resolution (C++11) #include <chrono>

```
auto time1 = high_resolution_clock::now();
// whatever code you want to time
auto time2 = high_resolution_clock::now();
size_t micros = duration_cast<microseconds>(…).count();
```

Non-algorithmic improvements 1/3

- Native arrays
  - **Java:** ArrayList much worse than native [] array

    Elements of an ArrayList cannot be basic data types (e.g. int), but have to be objects (e.g. Integer)

    This causes inefficient byte code / machine code
  - **C++:** std::vector is as good as [] with option –O3

    Elements of an std::vector can be basic data types as well as objects

    Due to C++'s templating mechanism, machine code for std::vector<int> is almost the same as for int[]

Non-algorithmic improvements 2/3

- Predictable branches
  - Branches = 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

- Sentinels
  - Special elements to avoid testing for index out of bound

    Less code + further reduction in number of branches
  - For list intersection: id ∞ at the end of both lists

    For Java, take: Integer.MAX_VALUE

    For C++, take: std::numeric_limits<int>::max()

Algorithmic improvements 1/8

- Preliminaries
  - We have to two lists, which we want to intersect
  - Let **A** be the smaller list, with **k** elements
  - Let **B** be the longer list, with **n** elements

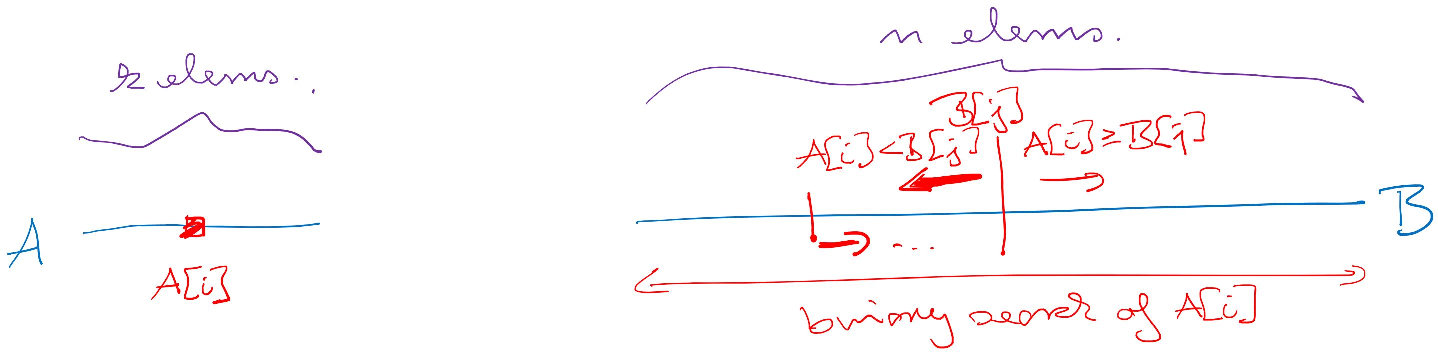
    List intersection is commutative, so we can always assume that the first list is A, and the second is B

Algorithmic improvements 2/8

Don't forget: A and B are SORTED

- Binary search in the longer list
  - Search each element from A in B, using binary search
  - This has time complexity $\Theta(k \cdot \log n)$

  Good for small k … but for $k = \Theta(n)$ this is $\Theta(n \cdot \log n)$, and hence slower than the "zipper"-style linear intersect



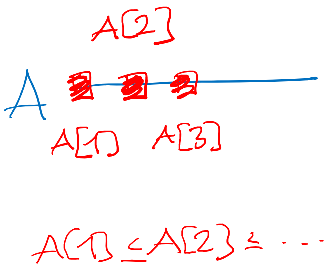
Algorithmic improvements 3/8

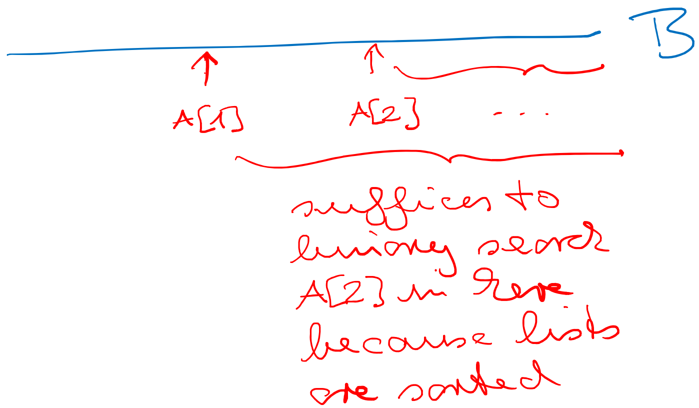
- Binary search in remainder of longer list
  - Time complexity in the best case $\Theta(k + \log n)$

    First element from A at the end of list B
  - Time complexity in the worst case $\Theta(k \cdot \log n)$

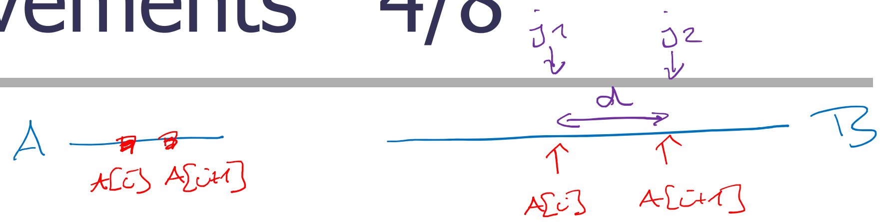
    All elements of A at the beginning of list B
  - Time complexity in the "typical" case $\Theta(k \cdot \log n)$

    Elements of A "evenly distributed" over list B





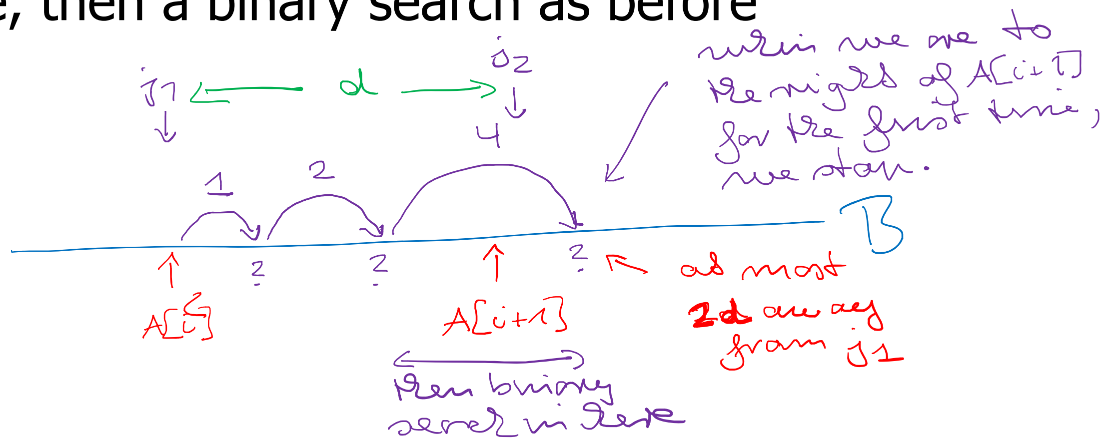
Algorithmic improvements 4/8



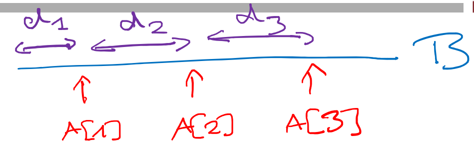
- Galloping search
  - **Goal:** when elements A[i] and A[i+1] are located at positions j_1 and j_2 in B, then, with $d := j_2 - j_1$ ("gap"):

    spend only time **$\Theta(\log d)$** 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 5/8



- Galloping search, time complexity
  - Let $j_1, \ldots, j_k$ the positions of the elements of A in B
  - Let $d_i = j_i - j_{i-1}$ for $i > 1$ and $d_i = 1$ (the "gaps")

    Note that $\Sigma_i\, d_i \leq n$ = the number of elements in B
  - Then the time complexity is $O(\Sigma_i \log d_i)$

    Not a nice formula, so let's find the maximum value, independent of the particular $d_1, \ldots, d_k$
  - Lemma: $\Sigma_i \log d_i$ is maximized when all $d_i = n / k$
  - Galloping search therefore takes time $\mathbf{O(k \cdot \log (1 + n/k))}$

    this is always O(n) !

Algorithmic improvements 6/8

- Proof of Lemma ... max $\Sigma_i \log d_i$ under constraint $\Sigma_i d_i \leq n$
  - This is an instance of **Lagrangian optimization**:
    1. Write constraint as equation: $\mathbf{\Sigma_i d_i - n' = 0}$ ... $n' < n$
    2. Define $\mathbf{L(d_1, \ldots, d_k, \lambda)} = \Sigma_i \log d_i + \lambda \cdot (\Sigma_i d_i - n')$
    3. Set partial derivatives = 0 to find all local optima and check the objective function at the borders

Algorithmic improvements 7/8

- Comparison-based lower bound
  - 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

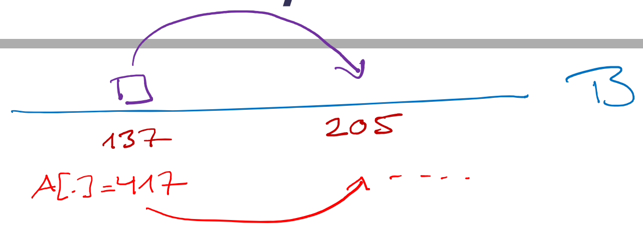
    Hence #steps required $\geq \log_2 (n!) = \mathbf{\Omega(n \cdot \log n)}$
  - We can use a similar argument for intersection / union:

    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 $\geq \log_2 (n/k)^k = \mathbf{k \cdot \log_2 (n/k)}$

  Galloping search is hence asymptotically optimal

Algorithmic improvements 8/8



- Skip Pointers
  - **Idea:** potentially skip large parts of longer list B
  - Skip pointer = special element in list B that points to an element B[j] further to the right

    When intersecting, follow pointer if current A[i] ≥ B[j]

    Placement of skip pointers is heuristic … for ES3 you can investigate good placements experimentally
  - Advantage: **very simple** to implement

    In particular, simpler than galloping search and thus often more effective in practice, even if not "optimal"

References

- Textbook

  Section 2.3: Faster intersection with skip pointers

- Literature

  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

- Wikipedia

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