Search Engines

WS 2009 / 2010

Lecture 5, Thursday November 19th, 2009
(Efficient List Intersection)

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

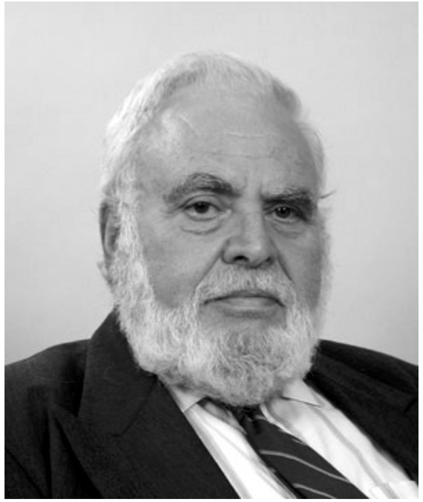
Overview of today's Lecture

- List intersection
  - you already know that it is at the core of what every search engine does

    (no fast search without fast list intersection)
  - revisit standard (linear-time) algorithm
  - prove a lower bound (how fast can we get)
  - algorithm that achieves a matching upper bound
- But before that
  - do missing proof for Golomb encoding from the last lecture
  - talk a bit about the exercises (again)

Golomb encoding (again)

- By Solomon Golomb, 1966
  - comes with a parameter M (modulus)
  - write positive integer x as $q \cdot M + r$
  - where $q = x \text{ div } M$ and $r = x \bmod M$
  - the codeword for x is then the concatenation of
    - the quotient q written in unary with 0s
    - a single 1 (as a delimiter)
    - the remainder r written in binary
  - examples



Solomon Golomb
*1932 Maryland

Golomb Encoding — Analysis

- Show that Golomb encoding is optimal
  - for gap-encoding inverted lists
  - assuming the doc ids in a list of size m are a random subset of size m of all doc ids $1..n$

[the proof]

On the exercises

- Amount
  - Should be less work this time
  - Do you prefer theoretical or practical or a mix?
- About the aspect of well-specifiedness
  - I am aware that the exercises are often not fully specified
  - this requires two things from your side
    - apply your common sense
    - in the case of doubt ask (intelligently)
  - these are two super-important skills to learn
    - real-life (research) problems are always ill-specifed (and actually much worse than in the exercises!)
    - common sense + communication are a must

List Intersection — Standard Algorithm

- For two lists A, B of sizes n and m:

[pseudocode of standard algorithm]

Improving the Standard Algorithm

- Two "engineering" problems with the previous code

1. there is an if-statement for each iteration of the loop

   why is that a problem?

   - modern processors do pipelining = execute future instruction while current instruction is not yet finished
   - in the case of an if, the processor tries to predict which part gets executed (so-called branch prediction)
   - if the prediction fails, the speculative execution of the future instructions has to be rolled back

2. the if-condition is also quite complex

   - costly to evaluate that in each iteration

List Intersection — Improved Version

- This code is (or can be) significantly faster:

[pseudocode of improved algorithm]

List Intersection — Lower Bound

- Can we do better than order n + m ?
  - if we want to compute the *union* of the two lists, we obviously can not
    - we have to output n + m elements in any case
  - for intersection we obviously can for special cases
    - for example: largest element of one list is smaller than smallest element of other list
    - then we can tell after one comparison that the intersection is empty
  - how about the general case
- Problem from now on
  - "locate" element from one list in the other list

List Intersection — Improvement 1

- Let's first try to improve on the standard algorithm
  - what if one list is much smaller than the other list?
    - length of smaller list is k
    - length of larger list is m
  - then we can binary search each element of the smaller list in the larger list

[an illustration of this]

  - complexity is $\sim k \cdot \log_2 m$
  - this is obviously better than $k + m$ when $k << m$

List Intersection — Improvement 2

- Simple observation:
  - if the previous element has been located at position i, the next binary search need only look at positions $\geq i$

[an illustration of this]

- Does this help us?
  - in the best case: [short calculation]
  - in the worst case: [short calculation]
  - in the "average" case: [short calculation]

List Intersection — Lower Bound

- Recall the lower bound for sorting n integers
  - there are $n!$ possible outputs
  - the algorithm has to distinguish between all of them
  - each comparison distinguishes between two cases
  - hence we need at least $\log_2 n!$ comparisons
  - by Stirling's formula $(n/e)^n \leq n! \leq n^n$
  - hence $\log_2 n! \sim n \cdot \log n$
  - hence every comparison-based sorting algorithm has a running time of $\Omega(n \cdot \log n)$
  - Note: not true for non-comparison based algorithms:

[explain by example 0-1 sequence]

List Intersection — Lower Bound

- Let's try the same for merging two lists A and B
  - that is, locate each element from A in B
  - again let k and m = number of elements in A and B resp.
  - same argument: how many ways are there to locate the k elements from A in the m elements from B
  - observe: each such way corresponds to a tuple $(i_1, \ldots, i_k)$ where $0 \leq i_1 \leq \ldots \leq i_k \leq m$

    (i_j is simply the location of the j-th element of A in B, location 0 means before the first element, location $i > 0$ means after the i-th element)
  - how many such tuples are there?

List Intersection — Lower Bound

- There is a similar quantity which is easy to count
  - the number of tuples $(i_1,\ldots, i_k)$ where $1 \leq i_1 < \ldots < i_k \leq n$
  - this is just the number of size-k subsets of $\{1,\ldots,n\}$
  - and the number of those is n over k $= n! / (k! \cdot (n-k)!)$
  - which by Stirling's formula is approximately $(e \cdot n/k)^k$

[relate the two kinds of quantities + prove the lower bound]

List Intersection — Lower Bound

List Intersection — Matching Upper Bound

- Idea:
  - after previous element from A has been located in B
  - start search from that location
  - but try to search not much further than next location

[an illustration of this]

  - trick: first *exponentional* search, then binary search
  - if the difference from the previous to the next location is d, this can be done in time O(d)

Analysis of this algorithm

- Terminology
  - 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 $\Sigma_i\ d_i \leq m$ = number of elements in B
  - then the time complexity of the algorithm is $O(\Sigma_i \log d_i)$

[derive upper bound in terms of k and m]