Suffix arrays

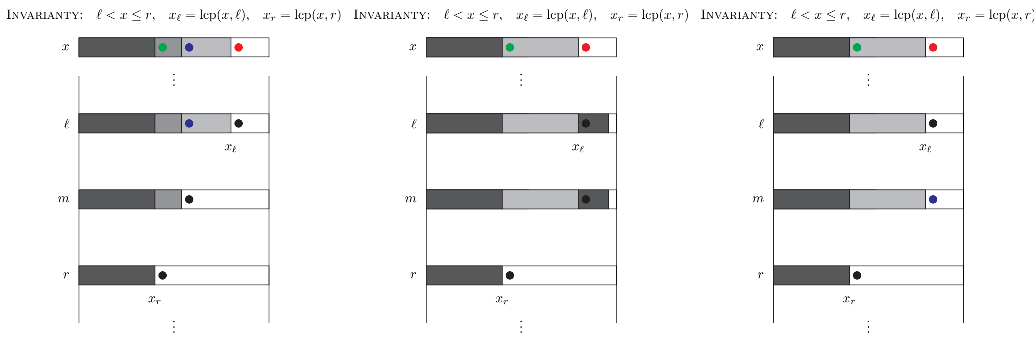
©kuko 26.11.2023

1 Introduction

- suffix array (SA) is simply an array of suffixes sorted lexicographically
- motivation: suffix trees (ST) take up an awful lot of memory – even if we're very very careful, 10-20B/character (just a single pointer takes 8 bytes!)
- suffix arrays need 1 int/character; if we have text up to 4 billion characters, we can use a 32-bit int, which is 4B/character + the text itself
- take for example the human genome, which is a string of about 3 billion characters from the alphabet `A`, `C`, `G`, `T`
- the string itself therefore takes about 3GB (if we use 1B/character), or 750MB if we use packed representation with 2bits/character
- the suffix tree will occupy 30–60GB or more and the suffix array about 12GB (+the string itself +0.75GB)
- and that's just the memory of the resulting structure, where we don't count the memory used temporarily during construction
- when processing larger inputs, we will be limited by the RAM size; if the data structure doesn't fit into RAM, we will get a lot of page swapping, which means disk accesses, which are much much slower than RAM accesses

2 Search

- binary search: $O(m \log n)$ (worse than ST – $O(m)$)
- suffixes starting with P form one continuous section in SA
- can be improved to $O(m + \log n)$ at the cost of more memory:
- let $\mathrm{lcp}(i, j)$ be the longest common prefix of the i-th and j-th suffix in order
- idea 1: if the upper and lower estimates have lcp > 0, we can skip these characters (still $m \log n$ in worst case)
- let's x be the searched text, let suffixes ℓ, r be the lower and upper limits, respectively
- invariant: $\ell < x \le r$, $x_\ell = \mathrm{lcp}(x, \ell)$, $x_r = \mathrm{lcp}(x, r)$
- i.e. x_ℓ (and x_r) is the number of symbols from the beginning, where x and ℓ (x and r) match (see the gray sections in the figure); $\ell[x_\ell] < x[x_\ell]$ (red character) and $x[x_r] < r[x_r]$ (green character)
- WLOG let $x_\ell > x_r$, let's look at the middle suffix m; what is $p = \mathrm{lcp}(\ell, m)$?
  - a) if $p < x_\ell$ (fig. left), it means that the common prefix $\mathrm{lcp}(\ell, m)$ is shorter than the common prefix $\mathrm{lcp}(x, \ell)$; at the same time $\ell < m$, i.e. $\ell[p] < \ell[m]$; but $\ell[p] = x[p]$ because their lcp is longer (the blue character in Fig. left is the same in x and ℓ and smaller than the black character in m); this implies $x < m$ and in constant time, we deduced that we need to continue searching in the first half
  - b) on the other hand, if $p > x_\ell$ (middle fig.), then ℓ and m have more characters in common than x_ℓ and, specifically, the x_ℓ-th character, in which ℓ and x differ; it follows that $x > m$ and we should search in the second half (time $O(1)$ again)
  - c) only if $p = x_\ell$ (fig. right), we cannot decide right away – in this case, we start comparing characters (from position p) and decide accordingly; in any case, $\max(x_\ell, x_r)$ will increase
- if $x_\ell \le x_r$, we proceed symmetrically (we compare $p = \mathrm{lcp}(m, r)$)
- note that every time we start comparing characters in case (c), $\max(x_\ell, x_r)$ increases and $\max(x_\ell, x_r) \le m$ so there will be at most m such comparisons in total; the other cases take $O(1)$, so the whole search takes $O(m + \log n)$



3 LCP

4 Construction

- qsort – $O(n^2 \log n)$ (meh)
- radix sort – $O(n^2)$ (meh)
- Manber–Myers: suffix of a suffix is also a suffix!
  - if we have an array sorted by the first K symbols, we can easily sort it by the first $2K$ symbols
  - when comparing $S[i \dots]$ vs. $S[j \dots]$, we already know the result of comparison $S[i \dots i+K]$ vs. $S[j \dots j+K]$; if that's a tie, look at the relative order of suffixes $S[i+K \dots]$ vs. $S[j+K \dots]$
  - we will have $\log n$ phases; in the k-th stage we sort all suffixes according to the first 2^k symbols
  - let $\text{rank}[i] = j$ if the suffix s_i is the j-th in alphabetical order (according to the first 2^k symbols; if two suffixes have the same first 2^k symbols, the ranks will be same)
  - phase: just sort the triples $(\text{rank}[i], \text{rank}[i + 2^k], i)$
- even better? yes, there is a linear construction:

- there are actually multiple $O(n)$ algs; this is by Kärkkäinen & Sanders '03:
- divided all suffixes into those at positions indivisible vs. positions divisible by 3
- recursively sort the positions $\equiv 1, 2 \pmod 3$ (and for every such suffix, calculate the position in the sorted array)
- when we have it, we sort the positions divisible by 3 by ünrollingöne symbol and looking at the relative order of the suffixes not divisible by 3 – we sort the pairs (first symbol, position in the 1 shorter suffix in the already sorted suffix array) by radix sort
- now we have two sorted arrays (suffixes at positions divisible and indivisible by 3) – we merge them with the classic merge algorithm, the comparison is in $O(1)$:
  - if we compare suffixes at positions 1 vs. $0 \pmod 3$, we ünrollöne symbol and get positions 2 vs. 1 (both positions are indivisible by 3 now so we know their relative order in $O(1)$)
  - if we compare suffixes at positions 2 vs. $0 \pmod 3$, we ünroll"two symbols and get positions 1 vs. $2 \pmod 3$ (again both are indivisible by 3 now)
- resulting complexity: $T(n) = T(\frac{2}{3}n) + O(n)$ where $T(\frac{2}{3}n)$ is the time of the recursive call and $O(n)$ is the sorting of positions divisible by 3 and merging; this recursion has solution $O(n)$
- note that we still have a problem: when using recursion, we can only do a recursive call for the same problem – but "calculate SA but just for selected positionsïs not the same problem as "calculate SA"
- trick: we take the original string from the 1st symbol and consider each triplet of characters as 1 symbol, then the suffixes of this string correspond to the suffixes at pos. $\equiv 1 \pmod 3$ in the original string

- after that, we concatenate the original string starting from the 2nd symbol and again, each triplet will be 1 symbol – suffixes of this string correspond to the suffixes at pos. $\equiv 2 \pmod 3$
- thus, we get a new string of length $2/3n$ and its suffixes correspond to suffixes at positions $\equiv 1, 2 \pmod 3$ in the original string
- new problem: huge alphabet; wtf you mean by "let's 3 chars now be 1 char- then the alphabet of size σ will become size σ^3 and it will grow exponentially
- second trick: a string of length n can contain at most n different symbols, i.e. we can just radix-sort and renumber the characters – this way, we keep the alphabet size $\leq n$