Pragmatic Programming

Session 7 - Tries, Suffix Arrays, and Strings

Max Nilsson

December 7th

Hashing

- There are many ways to hash a string `s`. The following formula will hash a string in $\mathcal{O}(n)$ such that the hash of each contiguous substring can be retrieved in $\mathcal{O}(1)$.
- The idea is to to let P be some large prime, e.g. `1e9+9`, and B be some generator in the field $\mathbb{Z}_P$., e.g. 9973.
- Then we can compute prefix-sums in some array: `ll hash[s.size()+1]`, using the following formula

$$\texttt{hash[n+1]} \equiv \sum_{i=0}^{n} B^{n-i}\texttt{s}[i] \mod P.$$

- A hash of the substring `s[a:b]` (inclusive range) then equals

$$\sum_{i=a}^{b} B^{b-i}\texttt{s}[i] \equiv \texttt{hash[b+1]} - \texttt{hash[a]}\,B^{b-a+1} \mod P.$$

- The probability of two strings `s, t` having the same hashing over all possible choices of the base B is approximately `max(s.size(), t.size())`$/P$. This follows from a the non-trivial Schwarz-Zippel lemma.

Trie (Prefix Tree)

- A trie (coming from the word re*trie*val) is a tree which stores all prefixes of some collection of strings over an alphabet Σ.
- Every node in the trie contains a flag `bool end_of_word;` and $|\Sigma|$ pointers to other (child) nodes (initialized as `null`).
- We can insert a string of size n

  ```
  void insert(string);
  ```

  by traversing the trie and creating new children when necessary, in $\mathcal{O}(n)$. We can similarly search for a string in our trie in $\mathcal{O}(n)$.
- The applications range from string completion algorithms to the `burst_sort` algorithm, which is the fastest string sorting algorithm (making efficient use of the cache).
- The major downside of the trie is the memory size, which equals $\mathcal{O}(n|\Sigma|) = \mathcal{O}(n)$. There exists remedies for this issue, such as the **Bitwise trie** and the **Radix trie**, which reduces the memory usage but increases the insertion/searching to $\mathcal{O}(n \log |\Sigma|)$.

Suffix Array

- This is an array `sa` which stores indices `i` of a string `s` which represent suffixes `s[i:n]`. The important part is that these indices are stored such that the suffixes are *sorted*.
- The naive implementation (of simply sorting all suffixes) is $\mathcal{O}(n^2 \log n)$. But can we do better?
- Yes of course, otherwise I would not be asking. We utilize the structure of suffixes, i.e. we do not treat them as normal strings but that they share a lot of characters. This can be done with $\mathcal{O}(n)$ time and memory complexity! (Farach (1997))
- For the easiest implementation, see[1].
- From a suffix array, one can in $\mathcal{O}(n \log n)$ construct a Longest Common Prefix (LCP) array. The LCP array contains the maximum length prefix match between two consecutive suffixes, after they are sorted in a suffix array.
- See coding example!

[1] Ge Nong, Sen Zhang and Wai Hong Chan. "Linear Suffix Array Construction by Almost Pure Induced-Sorting". In: *2009 Data Compression Conference*. 2009, pp. 193–202. DOI: 10.1109/DCC.2009.42.

Suffix Tree

- It combines the idea of a suffix array (stored indices of sorted suffixes) in a trie.
- The construction can be made again with Farach's algorithm in $\mathcal{O}(n)$ time and memory.
- It allows us to solve the following problems
  - Find if a string $\mathtt{t}$ appears as a substring of $\mathtt{s}$, where $\mathtt{t}, \mathtt{s}$ have sizes m, n respectively, in $\mathcal{O}(m)$. With a suffix array this can be done, using a binary search, in $\mathcal{O}(m \log n)$.
  - Find the longest common substring of $\mathtt{t}$ and $\mathtt{s}$ in $\mathcal{O}(n+m)$ time complexity (used for regex matching).
- A downside is the memory usage, which is again quite high.
- If you have multiple strings $\{\mathtt{t}_i\}_{i=1}^{k}$ that you want to match in $\mathtt{s}$ an efficient linear algorithm is the Aho-Corasick automaton which is the basis of the orignal `fgrep` Unix command.

This Week

- Skim read chapter 14 in Sannemo[2] and chapter 14 in Laaksonen[3].
- Solve half of this weeks 9 problems (truncated).

[2] Johan Sannemo. "Principles of Algorithmic Problem Solving". In: *Draft version* (2018).

[3] Antti Laaksonen. *Guide to competitive programming*. Springer, 2020.