1 Overview

First question: why a course on string algorithm? String algorithm is a classic computer science algorithmic subject. Manipulating text efficiently has been a long standing question. One of the most frequently encountered problems, for example, is finding whether a pattern occurs in text. Many well-known computer scientists have worked on string algorithms. We will hear of their names frequently in (early part of) this course. Many interesting ideas and problems are quite intuitive and easy to understand but elegant algorithms.

Second question: what will this course cover? In the first part of this lecture, I will cover classic string algorithms (e.g. KMP, suffix tree and suffix array, and Burrows-Wheeler transform). In the second part, I will mainly focus on applications (in the broad sense) of string algorithms in bioinformatics, especially in analyzing high-throughput sequencing data. So why do I want to focus on sequence data analysis while there are many algorithmic approaches in bioinformatics? Sequence data analysis is one of the main applications of string algorithm in bioinformatics. Efficient algorithmic processing of sequence data is especially useful in the context of “big data”. Another challenge is the inherent noise in the sequence data, which poses significant difficulty for algorithm development.

2 String algorithms: various formulations

First, we need to know the problem formulation before talking about algorithms. One of the central problems in this course is the string matching problem. Given a text string \( T \) with length \( m \) and a pattern string \( P \) of length \( n \), we want to find whether \( P \) occurs in \( T \) and if so, report all occurrences of \( P \). This problem sounds simple, but there are subtle details that need to be known.

1. Do we allow inexact matches? Inexact matches allow \( P \) to be somewhat modified in their “copies” in \( T \). If inexact matching is allowed, the string matching problem becomes much more difficult. At present, the gold-standard algorithms for inexact string matching are those based on dynamic programming. If you have taken an algorithm course or bioinformatics course before, you may have learned how to do sequence alignment or edit distance between two strings. The most efficient algorithms run roughly in \( O(mn) \) time. If this sounds unfamiliar, it is time to study such simple dynamic programming algorithms. See the lecture notes posted on the class web page.

2. Now suppose we focus on exact matching only. Now situation improves significantly. While the exact string matching problem can be solved in \( O(mn) \) time, there exists many clever linear time algorithms that run in \( O(m+n) \) time. Almost all such algorithms perform some sort of preprocessing. That is, we first collect some information about \( P \) or \( T \), and then such preprocessing will help to perform string matching. Here, we can preprocess \( P \), which are done by many classic algorithms such as KMP and Boyer-Moore. Alternatively, one may preprocess \( T \), which
may make more sense if $T$ is more or less fixed (like a dictionary). Suffix tree and suffix array belong to such category.

3. Other issues: what is the alphabet of $P$ and $T$? how large can $P$ and $T$ be? Is $P$ usually much shorter than $T$? Is memory usage an issue? Do we allow some chance for wrong answer (e.g. with certain probability, the algorithm may report the wrong or uncertain answer). These will affect the design of string matching algorithms.

Note: string matching is an important problem but has been very thoroughly studied. There are many related but different problems in string processing, especially from bioinformatics (e.g. sequence data analysis). We will cover various such problems in the second half of the course.

## 3 First taste of string matching: Z preprocessing

Recall Text $T$ is length $m$ array of characters drawn from finite alphabet. We now show how to perform indexing (preprocessing) of $T$, which allows fast string matching. Here, indexing means that given $T$, we want to create some data structure to allow efficient repetitive searches. Here, we explain a simple preprocessing called Z preprocessing, which can lead to a very simple exact string matching algorithm that runs in linear time $O(n + m)$.

Now our first algorithmic topic: preprocessing with Z-scores. **Definition:** for each position $i$ (where $1 \leq i \leq n$) in a string $S$ of length $n$, $Z_i$ is the length of the longest substring of $S$ that starts at $i$ and matches a prefix of $S$. Example: suppose $S = aabcaabzaaz$. Then $Z_5 = 3$, $Z_6 = 1$, $Z_9 = 2$ and so on. The Z value defines a Z-box for each position $i$, which is the matching substring start at position $i$ and matches the prefix. Note $Z_i$ can be equal to 0 if no such prefix exists (e.g. $Z_3 = 0$).

How do we compute $Z$ efficiently? We start with $i = 2$. $Z_2$ is computed by direct comparison. Then suppose we have computed all $Z_j$ for all $j < i$. We now show how to compute $Z_i$. **Definition:** for each $i$, $l_i$ is the right-most endpoint of Z-box that begin at or before $i$. And we let $r_i$ be the starting point of that Z-box ending at $r_i$. Note that $r_i$ and $l_i$ can be maintained properly easily during the algorithm. Make sure you understand this definition. In the above example, $r_2 = 2$ and $l_2 = 2$. $r_3 = 2$ and $l_3 = 2$. $r_5 = 7$ and $l_5 = 5$. Note for $r_2$ and $l_2$ can be easily setup in computing $Z_2$.

Now suppose we are computing $Z_i$. Note that we know all $Z_j$ when $j \leq i - 1$ and also $r = r_{i-1}$ and $l = l_{i-1}$. There are several cases.

1. Case 1: $i > r$. That is, all previously matched Z-box do not cover $i$. Then we simply perform direct comparison starting from $i$ (with the prefix of $S$). We can do this because there is no danger for repetitively comparing: new chars are compared here.

2. Case 2: $i \leq r$. Note the whole $[l, r]$ Z-box will match the prefix of $P$ and this Z-box overlaps with $i$. So let $i'$ be the corresponding position in this prefix. There are two sub-cases. (2a) if $Z_{i'} < r - l + 1$, then we immediately conclude $Z_i = Z_{i'}$. You should draw a picture here to understand what it means. (2b) if $Z_{i'} \geq r - l + 1$, to know $Z_i$, we can start comparing from $S[r + 1]$ with $S[r - l + 1]$. Suppose this comparison stops at position $q$ (either reaching the end or there is a mismatch). For simplicity, we assume this iteration ends with a mismatch. Then $Z_i = q - i$. We also set $r_i = q - 1$ and $l_i = i$.

**Time analysis:** comparisons are either mismatches or matches. Note that the number of mismatches is at most $n$ since a mismatch ends an iteration and there are at most $n$ of them. Each match increases the $r_i$ by 1 (or we never compare with old ones without increasing $r_i$). Note $r_i \leq n$. So total $2n$ comparisons are performed.

It is possible to do pattern matching with Z score. Do you see how Z preprocessing can help to determine whether $P$ appears in $T$? But there are better approaches (some runs in sub-linear time usually) and so Z only serves as a theoretical tool: simple to see it is linear and useful in different context.