

*Please do not circulate.
These notes are still under construction.*

Pattern matching

Lecture notes of the ALG course

Part I

Introduction

Pattern matching is the process of finding occurrences of a specific pattern into a larger text. When you hit `ctrl-F` on a webpage to quickly find a word of interest, pattern matching algorithms are running under the hood. When you use a web search engine, asking for webpage that contain most of the keywords you queried, pattern matching algorithms are working for you. When you try to find similarities between two documents to detect plagiarism, these algorithms are still there. And their applications do not limit to prose **Ex. 1**. For example, by defining a “letter” to be a couple note-duration, then one could search for the pattern,

$$(C,1)(D,1)(F,1)(D,1)(A,3)(A,3)(G,6)$$

within a labelled database of melodies to identify the music that is currently playing.

In this course, we are interested in pattern matching problems in the context of sequence bioinformatics. Hence, we will work with the letters A, T, C and G that correspond to the four DNA nucleotides.

The aim of this unit is to introduce you to stringology, the research domain of text algorithm, and its concepts, by presenting you a few classical algorithms with a focus on the idea rather than the actual super-efficient implementation. We will cover ideas and algorithms that allows for pattern matching over strings of millions of characters. Hopefully, you will enjoy their beauty as much as we are!

■ Fifty shades of pattern matching

Now that we understand informally the problem, let us formalize it. As we saw, the string S and the pattern P that will be given to our pattern matching algorithms are made of letters, that may be a little different from what we know. Hence, we need to define the *alphabet* of letters, which is simply a set of symbol. For such input, we can define many variants depending

on what is expected as an answer. In any case, the concept of *occurrence* is central. We say that P occurs in S if it is a substring of S , that is if there is some $k \in [0..|S|)$ such that

$$S = S_{[0..k)}PS_{[k+1..|S|)},$$

in which case k is the (starting) *position* of an occurrence of P in S . When clear, we confound the occurrence and its starting position.

The things we want to learn about the occurrences of P in S define different flavours of pattern matching. In this course, we will mostly focus on the following problems,

Definition 1 (Pattern matching problems, informal). *Given a pattern P and a text S over an ordered alphabet Σ , with $|P| \leq |S|$, we consider the following problems:*

- **Membership.** *Return whether P occurs in S , or not*
- **Count.** *Return the number of occurrence of P in S .*
- **LocateAll.** *Return (if any) the positions of the occurrences of P in S .*

Note that outputs are of very different nature: Membership returns a boolean, Count an integer, and LocateAll a list of integers. We also highlight that LocateAll is the hardest, in the sense of complexity theory, that the others. Indeed, if you find an algorithm for this task, it immediately gives you an algorithm for the two others with the same complexity [Ex. 2](#). Similarly, Count is harder than Membership [Ex. 3](#).

■ Naive algorithms for pattern matching

As always in algorithmic, we start by establishing a very simple algorithm that solves the problem. Indeed, it may be the case that this very first idea is sufficient. Our idea is as follows: we will scan the text from left to right checking, at each position k , whether the substring of $|S|$ of length $|P|$ starting at index k is equal to P . The occurrence checking verifies the characters from left to right, reporting a mismatch as soon as two characters mismatch. What we do when an occurrence is found depends on the problem: in Membership, we early return True; in Count, we increment a counter; in LocateAll, we append the occurrence to a list.

Here is the algorithm for LocateAll, the two others being left as exercise [Ex. 4](#).

Algorithm 1: Naive algorithm for LocateAll

```

1:  $occ \leftarrow []$ 
2: for  $k \in [0..|S| - |P|)$  do
3:   if OCCURS( $P, S, k$ ) then
4:     return  $occ = occ + [k]$ 
5: return  $occ$ 

6: function OCCURS( $P, S, k$ ):
7:   for  $i \in [0..|P|)$  do
8:     if  $P[i] \neq S[k+i]$  then
9:       return False
10:  return True

```

Algorithms allow us to reason on the behavior of a method while being agnostic to programming languages. The intermediate format between prose and source code allows to hide some implementation

detail for easing analysis. Nevertheless, to trully and deeply master an algorithm, the best way is still to program it yourself [Ex. 5](#).

Time complexity Let us analyse Algorithm 1. The function OCCURS performs, in the worst-case, $|P|$ comparisons (l. 7-8). It is called $(|S| - |P|)$ times (l. 2-3). After each call, a constant-time operation is run. Overall, the worst-case complexity is thus

$$\mathcal{O}(|S| - |P|) \cdot \mathcal{O}(|P|) \cdot \mathcal{O}(1) = \mathcal{O}(|S| \cdot |P|).$$

We can be slightly more precise. Consider running the algorithm on $P = aa \cdots a$ and $S = aa \cdots a$, and observe that the running time of this particular instance reaches $\mathcal{O}(|S| \cdot |P|)$. This means that our worst-case complexity bound is tight, and we can write that the time complexity is $\Theta(|S| \cdot |P|)$ instead.

We highlight that this is only a worst-case bound: for some instance, the running time can be considerably faster [Ex. 6](#).

■ Outline

In Part II, we introduce the classical notations and concepts used through the course. We then explore various efficient pattern matching algorithm depending on whether the preprocess applies on the pattern (Part III), or the string (Part IV).

■ Chapter exercises

- [Ex. 1](#) Can you think of other pattern matching application beyond prose? For each of them, write a pattern one could search for.
- [Ex. 2](#) Give an algorithm for Membership and Count that use LocateAll as a subroutine call
- [Ex. 3](#) Give an algorithm for Membership that uses Count as a subroutine call
- [Ex. 4](#) Write explicitly the naive algorithm for Membership and Count.
- [Ex. 5](#) Implement those simple algorithms in the language of your choice
- [Ex. 6](#) Propose an instance of the LocateAll problem that runs in time $\mathcal{O}(|S|)$.



Part II

Preliminaries

(1) ■ Maths symbols

A set is an unordered collection of objects, called elements. For a set S , the sentence “ x is an element of S ” is expressed notationally as $x \in S$.

A **proposition** is a sentence that is either true or false. For P and Q two propositions

- conjunction. The proposition “ P and Q ” is true if and only if both P and Q are true.
- disjunction. The proposition “ P or Q ” is true if and only if at least one of P or Q is true.
- conditional. The proposition “ $P \implies Q$ ” is true if and only if P is false or both P and Q are true. It reads “ P implies Q ”, which is close to the (perhaps) more intuitive “if P , then Q ”. anything can happen.
- biconditional. The proposition “ $P \iff Q$ ” is true if and only if P and Q have the same truth value. It reads “ P if and only if Q ”.
- universal quantifier. The proposition “ $\forall x \in S, P(x)$ ” is true if and only if $P(x)$ is true for every x that lives in the set S . It can be read “For all x in S , it holds that $P(x)$ ”.
- existential quantifier. The proposition “ $\exists x \in S, P(x)$ ” is true if and only if $P(x)$ is true for at least one of the x that lives in the set S . It can be read “there exist x in S such that $P(x)$ ”.
- unique existential quantifier. The proposition “ $\exists! x \in S, P(x)$ ” is true if and only if $P(x)$ is true for exactly one of the x that lives in the set S . It can be read “there exist a unique x in S such that $P(x)$ ”.

Rq. Don't mix maths and text: in particular, don't use \exists as a quickterm for “it exists”. First-order logic statements, but allow for “and” and “or” as handier than “ \wedge ” and “ \vee ”.

The summation symbol allows for a compact notation for the addition of a sequence of numbers. It is defined as

$$\sum_{i=k}^{\ell} a_i = a_k + a_{k+1} + \cdots + a_{\ell-1} + a_{\ell},$$

that is the addition of a_i 's terms, for i varying from k to ℓ . We extend the notation to finite sets with

$$\sum_{x \in S} f(x)$$

being the sum of $f(x)$'s over all elements x in the set S .

(2) ■ Stringology stuff (alphabet, prefix)

An (totally) **ordered alphabet** is a couple (Σ, \preceq) . The first element is the alphabet Σ , a nonempty finite set of symbols, called **letters**. The second element \preceq is an order over Σ^* such that for any distinct letters $x, y \in \Sigma$, it either holds $x \preceq y$ or $y \preceq x$. For two letters $x, y \in \Sigma$, we write $x \prec y$ when both $x \preceq y$ and $x \neq y$ hold.

A **string** (or **word**) over a given alphabet is a finite sequence of letters. For a string s , we denote by $s_{[i]}$ its $(i + 1)$ -th letter, and by $|s|$ its length, which is number of letter it is made of. Note that the first letter is the letter indexed by 0, while the last letter is the letter indexed by $|s| - 1$. The indexes are meant to be read modulo $|s| - 1$, so we allow ourselves to use $s_{[-1]}$ to denote the last character of s .

The empty string, with no letter, is written ε .

The **lexicographic order** \preceq_{lex} over words of (Σ, \preceq) is defined by:

1. $\forall w \in \Sigma^*, w \neq \varepsilon \implies \varepsilon \prec_{\text{lex}} w$,
2. $\forall w, w' \in \Sigma^+, (w \prec_{\text{lex}} w') \iff (w_{[0]} \prec w'_{[0]}) \text{ or } (w_{[0]} = w'_{[0]} \text{ and } w_{[1..]} \prec_{\text{lex}} w'_{[1..]})$.

A **substring** of s is a contiguous subsequence of s . For $n < m$ two integers, we write $s_{[n..m]}$ for the word $s_{[n]}s_{[n+1]} \cdots s_{[m]}$, and we write $s_{[n..m]}$ for the word $s_{[n..m-1]}$. When n is omitted we refer to a **prefix** of s , as in $s_{[..m]} = s_{[0]}s_{[1]} \cdots s_{[m]}$; and when m is omitted we refer to a **suffix** of s , as in $s_{[n..]} = s_{[n]}s_{[n+1]} \cdots s_{[l-1]}$.

(3) ■ Landau notation

In computer science, the **complexity** of an algorithm generally refers to the amount of computational resources needed by this algorithm to complete its task, notably the time it takes and the space it requires. This complexity is often expressed as a function of the input size.

We write

$$f = \mathcal{O}(g)$$

for

$$\exists c, x_0 \in \mathbb{R}, \forall x \geq x_0, |f(x)| \leq c \cdot |g(x)|.$$

*This is a binary relation that is reflexive ($\forall x \in \Sigma, x \preceq x$), antisymmetric ($\forall x, y \in \Sigma, (x \prec y \text{ and } y \prec x) \implies x = y$) and transitive ($\forall x, y, z \in \Sigma, (x \preceq y \text{ and } y \preceq z \implies x \preceq z)$).

Note that seeing $\mathcal{O}(g)$ as a set and write $f \in \mathcal{O}(g)$ would be more precise as the equality can only be read one-way, but this abuse of notation is common.

We now list a few consequences of the definitions, that are useful to analyse the asymptotical complexity of algorithms. **Ex. 7** **Ex. 8**

Proposition 2 (Big-O manipulations). *Let $f_1 = \mathcal{O}(g_1)$, $g_1 = \mathcal{O}(h_1)$, $f_2 = \mathcal{O}(g_2)$ and $c \in \mathbb{R}$. The following assertions hold:*

1. $f_1 f_2 = \mathcal{O}(g_1 g_2)$
2. $f_1 + f_2 = \mathcal{O}(\max\{|g_1|, |g_2|\})$
3. $c \cdot f_1 = \mathcal{O}(g_1)$
4. $f_1 = \mathcal{O}(h_1)$

When both $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$, we write $f = \Theta(g)$. This better characterizes the complexity of an algorithm: while utterly non-precise, saying that a linear algorithm runs in $\mathcal{O}(n!)$ time is true.

■ Chapter exercises

Ex. 7 Demonstrate these propositions

Ex. 8 For each of the following Python codes, estimate the running time as a function of n given that $\text{op}()$ runs in constant time, and $\text{op}(k)$ runs in time proportional to k^2 . You may want to have a look to triangle and pyramide numbers.



```
# Program 1
for i in range(n):
    op()

# Program 2
for i in range(n):
    for j in range(n):
        op()

# Program 3
for i in range(n):
    for j in range(i):
        op()

# Program 4
for i in range(n):
    for j in range(i, n):
        op()

# Program 5
for i in range(n):
    op(i)
```


Part III

Preprocessing strings

(1) ■ Kmer indexing

This is arguably the most natural method one can think of: an inverted file. Given a text in natural language, the idea is to build a dictionary of the words used in the text, together with their locations within it. Searching for a word then amounts to consult this dictionary, by running a binary search. In order to apply this idea in bioinformatics, we need a notion of word. As there is no such notion in biology, we create it artificially: we will index the k -mer of the string, that is all its substring of length k .

Definition 3 (k -mer index). Let $k \in \mathbb{N}$. A k -mer index of a string S is a data structure that represents occurrence lists O_W for each k -mer K present in S ,

$$i \in O_K \iff S_{[i..i+k)} = K.$$

We define the size of the index to be the number of distinct k -mers it hosts.

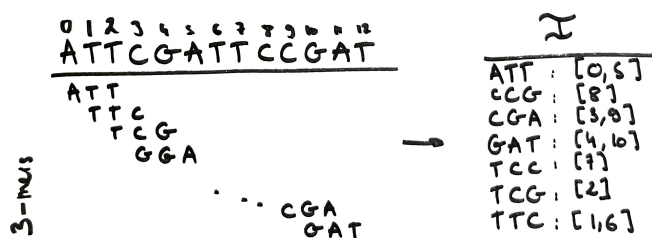


Figure 1: The 3-mer index \mathcal{I} of the string ATTCGATTCCGAT

Indexing k -mers Here is an algorithm for construction the k -mer index. We choose to use an array of list structure for the index, as it allows to run a binary search for finding keys of entries, and then easily append an occurrence in constant time. The drawback of this design is that inserting a new key takes $\mathcal{O}(|I|)$ time once the location of insertion has been found.

Algorithm 2: Building the k -mer index

Input: A string S , an integer k

Output: The k -mer index of S , an array of lists denoted I

```

1:  $I = []$ 
2: for  $i \in [0..|S| - k]$  do
3:    $kmer \leftarrow S_{[i..i+k]}$ 
4:    $(I, j_{kmer}) \leftarrow \text{FINDKEYORCREATEIT}(I, kmer)$ 
5:   Append  $i$  to  $I[j_{kmer}]$ 
6: return  $I$ 

7: function  $\text{FINDKEYORCREATEIT}(I, kmer)$ :
8:    $(j, found) \leftarrow \text{BINARYSEARCHINDEX}(I, kmer)$ 
9:   if  $found$  then  $\triangleright$  A  $kmer$  entry was found
10:    return  $(I, out)$ 
11:  else  $\triangleright$  No  $kmer$  entry was found,  $j$  is where  $kmer$  should be inserted
12:     $I' \leftarrow I_{[0..j]} + [kmer :: []] + I_{[j..]}$ 
13:    return  $(I', j)$ 

```

The index requires $\mathcal{O}(|S| + |I|)$ space to store all the occurrences. The building time is dominated in each loop by the call of `FINDKEYORCREATEIT`. The complexity of the later is $\mathcal{O}(\log |I|)$ if *found* is true, because of the binary search, and $\mathcal{O}(|I|)$ otherwise because of the creation of a new index I' (l. 12). Overall, this gives a $\mathcal{O}(|S| \cdot |I|)$ complexity, which is $\mathcal{O}(|S|^2)$. In fact, this is even $\Theta(|S|^2)$ **Ex. 1**.

Searching patterns Clearly, searching for a pattern of length k is immediately done using binary search in $\Theta(\log |I|)$. This extends to patterns of length k' , k because as I is sorted by its keys, the occurrences of P are the ones of all the k -mers prefixed by P . This gives naive running time of $\mathcal{O}(\log |I| \cdot |\Sigma|^{k-k'} + \#occs)$ for `LocateAll`. But this can be improved. Indeed, all the k -mers prefixed by P appear contiguously in I . This allows one to search for the first and the last such k -mers using binary searches, and then report all the occurrences within the interval. This idea is formalized in Algorithm 3, that runs in $\Theta(\log |I| + \#occs)$.

Algorithm 3: LocateAll using k -mer index

Input: The k -mer index of a string S , a pattern P of length bounded by k

Output: The list of occurrences of P in S

```

1:  $L = []$ 
2:  $first \leftarrow \text{BINARYSEARCHFIRST}(I, P)$ 
3:  $last \leftarrow \text{BINARYSEARCHLAST}(I, P)$ 
4: for  $i \in [first..last]$  do
5:    $L \leftarrow L + I[i]$ 
6: return  $L$ 

```

The limitation of k -mer indexing is that it doesn't allow for searching patterns of arbitrary size at reasonable cost **Ex. 2**.

Another approach Our definition of the k -mer index imposes that if a k -mer is not present in S , then it is neither a key in I . But one could have reasoned differently: all possible k -mers are entries of I , but only some of them carry non-empty lists.

This greatly ease the building algorithm, as there is no need to create new keys in the index. Moreover, it also fasten both the building and searching algorithms because the binary search is no longer needed: one could directly derive j_{kmer} from $kmer$ in $\mathcal{O}(k)$ time **Ex. 3**. Nevertheless, this comes at some cost. The downside of this approach is its space requirement of $\mathcal{O}(|\Sigma|^k + |S|)$ **Ex. 4**.

(2) ■ Suffix tries, and their compaction

Searching for a pattern within a string appears to be difficult because we don't know where to start. This is why the naive algorithms are so inefficient: they test all candidate starting position. While not practical to search for long patterns, the k -mer index from last section had an important design choice: it directly reasons on the pattern. Given one pattern, one locates the relevant rows within the index and report occurrences.

Here is a nice observation:

💡 **Key idea.** P is a substring of S iff P is the prefix of a suffix of S .

Indeed, it rephrases the complex problem of checking substring to the more simpler problem of checking prefixes **Ex. 5**. So, the pattern matching problem boils down to finding a nice data structure that is space efficient and well suited for prefix matching, and build it efficiently for the set of suffixes of S . This is where tries, and their refinement as compact tries, enter the realm.

(2.1) ◆ Tries

A trie is a labeled tree that represent a set of strings \mathcal{S} , made of at most n leaves and n marked nodes such that: **(1)** the edges are labeled by letters of Σ , **(2)** the outgoing edges of any given node are labeled by distinct letters, **(3)** every leaf is marked, **(4)** a string S belongs to \mathcal{S} if and only if S labels a path that goes from the root to a marked node.

Introduction being done, let be a bit more formal. Constructing a trie of a list of strings is done by successively inserting each string within the trie. For adding a new string, one follows the existing trie as long as possible: either we end up at an existing node and we mark it, or we get stuck at some node and we branch to add the remaining part of the string, marking the new added leaf. This gives the following algorithm, that runs in time $\Theta(\sum_{S \in \mathcal{S}} |S|)$ **Ex. 6**, and produces a tree made of $\Theta(\sum_{S \in \mathcal{S}} |S|)$ nodes **Ex. 7**.

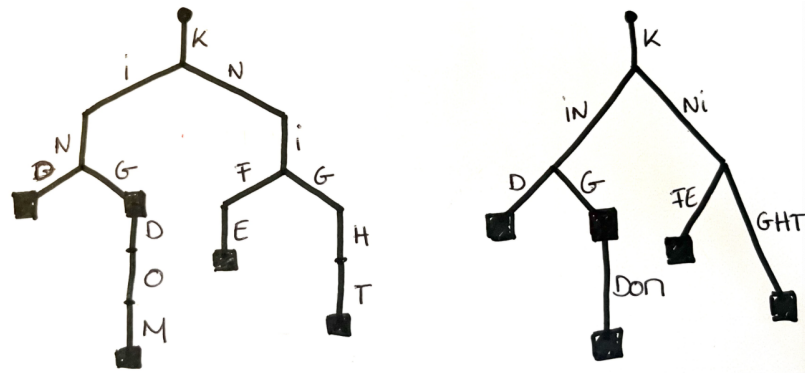


Figure 2: On the left, the trie of $S = \{\text{kind, king, kingdom, knife, knight}\}$. On the right, the compacted tree of S .

Algorithm 4: Construction of the trie of S

```

1:  $T \leftarrow \text{NEWUNMARKEDROOT}$ 
2: for  $S \in \mathcal{S}$  do
3:    $i_{\text{letter}} \leftarrow 0, \text{node} \leftarrow \text{root}(T)$ 
4:   for  $i \in [0..|S|)$  do
5:     if  $\text{child}(\text{node}, S_{[i]})$  doesn't exist then
6:        $\text{CREATEUNMARKEDCHILD}(\text{node}, S_{[i]})$ 
7:        $\text{node} \leftarrow \text{child}(\text{node}, S_{[i]})$ 
8:      $\text{MARK}(\text{node})$ 
9: return  $T$ 

```

Compacted trie The compacted trie, also known a Patricia trie, greatly reduce the number of node needed to store equivalent information. Starting from a trie T , the compacted trie T_C is obtained by merging edges of non-branching non-marked nodes, resulting a tree whose edges are labeled by words instead of letters. This decreases the upperbound on the number of nodes down to $2|\mathcal{S}| - 1$. This is because there are $|\mathcal{S}|$ marked nodes in T (one per string), and that non-marked nodes are now all branching by construction, so there are at most $|\mathcal{S}| - 1$ of them. As the merging phase can be performed alongs a depth-first search, the construction of a compacted trie requires $\mathcal{O}(\sum_{S \in \mathcal{S}} |S|)$ time.

(2.2) **◆ Suffix trie**

Leveraging the initial discussion of the section, one would probably define the suffix trie of a string to be the trie of its suffixes. The downside of this immediate approaches is that not all suffixes are created equal: some are leaves, while others are internal nodes.

The small twist is to append a character $\$ \notin \Sigma$ to the string S and to focus on $\text{Suffs}(S\$)$. Indeed, as no suffix of $S\$$ are prefix of another suffix of $S\$$ (Ex. B), this ensures that all suffixes appears in the trie as leaves. We thus define

Definition 4 (Suffix trie). *The suffix trie of a string S over an alphabet Σ , denoted ST_S is the trie of $\text{Suffs}(S\$)$, where $\$ \notin \Sigma$ is a fresh symbol called the termination symbol. It verifies the property*

$$s \in \text{Suffs}(S\$) \iff s \text{ spells a root-to-leaf path in } ST_S.$$

Additionally, its leaves are decorated with the starting position of their corresponding suffix in S .

that can be computed in time $\mathcal{O}(\sum_{s \in \text{Suffs}(S\$)} |s|) = \mathcal{O}(|S|^2)$, based on preceeding section. There are as many letters to store as nodes, and each branching node has almost $|\Sigma|$ branches. Overall, the suffix trie of S requires $\mathcal{O}(|S|^2 + |S|^2 + |S||\Sigma|) = \mathcal{O}(|S|^2)$ space. This is prohibitive in practice, even for moderate size genomes (eg. bacteria) [Ex. 9](#).

Here is the suffix trie of “banana”. A good exercise [Ex. 10](#) would be to build the suffix trie of “abracadabra” on your own.

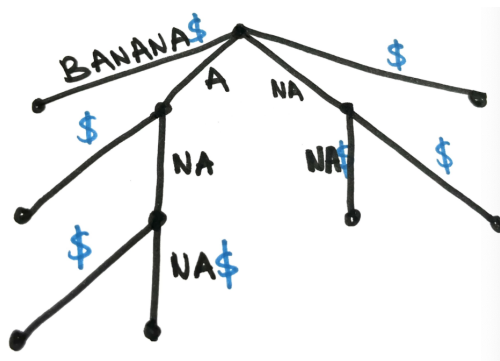


Figure 3: The suffix trie of BANANA, in its compacted form

(2.3) **◆ Suffix tree**

To limit the space requirement, we naturally introduce suffix trees as follows.

Definition 5 (Suffix tree). *The suffix tree of a string S over an alphabet Σ , denoted ST_S is the compact trie of $\text{Suffs}(S\$)$, where $\$ \notin \Sigma$ is a fresh symbol called the termination symbol. All its internal nodes are branching [Ex. 11](#), and it verifies the property*

$$s \in \text{Suffs}(S\$) \iff s \text{ spells a root-to-leaf path in } ST_S.$$

Additionally, its leaves are decorated with the starting position of their corresponding suffix in S .

that can be computed in $\mathcal{O}(\sum_{s \in \text{Suffs}(S\$)} |s|) = \mathcal{O}(|S|^2)$, based on preceeding section.

Space requirements We already mentioned that ST_S is made of $\Theta(|S|)$ nodes, each of them having up to $|\Sigma|$ children. So storing the topology of ST_S requires $\Theta(|S| \cdot |\Sigma|)$ space.

However, we didn’t discuss yet the cost of storing edge labels. Currently, as each root-to-leaf path spells a suffix of $S\$$, this costs amounts to the prohibitive $\mathcal{O}(|S|^2)$. In order to lower it, one could remark that the label of an edge (u, v) in ST_S is

$$\ell((u, v)) = (S_{[i..]})_{[\text{depth}(u).. \text{depth}(v)]},$$

where i is the starting occurrence of any (eg. the left-most) root-to-leaf path going through v , and thus through the edge (u, v) . That is, if for any node we were able to recover **(1)** such a i , **(2)** the depth of the node, we could forget the label and recompute it on-the-fly. Nicely, the trie computation and its compression can be adapted to compute those integers and store them at the level of nodes without increasing the asymptotical complexity **Ex. 12**. This drastically reduces the space requirements of edge labels down to $\mathcal{O}(|S|)$.[†] Overall, the suffix tree necessitates $\mathcal{O}(|S| \cdot |\Sigma| + |S|) = \mathcal{O}(|S| \cdot |\Sigma|)$ space to be stored.

Here is the suffix tree of “banana”, a good exercise **Ex. 13** is to build the one of “abracadabra” on your own.

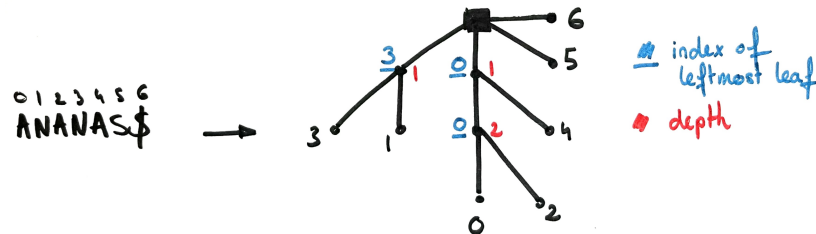


Figure 4: The space efficient representation of the suffix tree of ANANAS

(2.4) **◆ Linear time construction of the suffix tree**

This section is rather advanced, and under construction.

[Léo: TODO later, McCreight’s construction]

(2.5) **◆ Querying the suffix trie**

(2.5.1) **• Pattern matching**

For S a string whose suffix tree is ST_S , and P a pattern, let consider again our favorite problems Membership, Count and LocateAll.

Membership As motivated earlier, testing whether P belongs to S amounts to testing whether P is the prefix of a suffix of S . For simplicity, first imagine we are give the trie of the suffixes of $S\$$, that we call suffix trie. Given a pattern, navigate the tree by following the letters of the patterns: if you succeed, then P is a prefix of a suffix of S and thus belongs to S ; if you get stuck, P is not a substring of S . We formalize this in the following algorithms, that clearly runs in $\Theta(|P|)$ time.

[†]The rigorous reader probably noted that this comes at the cost of longer access time to labels. In reality this is not a problem for our pattern matching applications, because we will recover the label as we progress in the pattern to search, one letter at a time; and that accessing a specific letter of a label is still constant time.

Algorithm 5: Membership on suffix trie

Input: The suffix trie ST_S of S , a pattern P
Output: Whether P is a substring of S , or not

```

1:  $node \leftarrow \text{root}(ST_S)$ 
2: for  $i \in [0..|P|)$  do
3:   if  $\text{child}(node, P_{[i]})$  doesn't exist then
4:     return False
5:    $node \leftarrow \text{child}(node, P_{[i]})$ 
6: return True

```

Count Counting the number of occurrence of a pattern starts in a very similar way: letter by letter, we explore the trie by following the pattern to search. Naturally, if we get stuck this means that there are no occurrences of P in S ; but what if we succeed? In this case, our pattern is a prefix of all of the leaves covered by the node with ended at. But the later ones exactly correspond to suffixes of S : so each of the covered leaves correspond to a starting position of P in S . The algorithm follows.

Algorithm 6: Count on suffix trie

Input: The suffix trie ST_S of S , a pattern P
Output: The number of occurrences of P in S

```

1:  $node \leftarrow \text{root}(ST_S)$ 
2: for  $i \in [0..|P|)$  do
3:   if  $\text{child}(node, P_{[i]})$  doesn't exist then
4:     return False
5:    $node \leftarrow \text{child}(node, P_{[i]})$ 
6: return COUNTCOVEREDLEAVES( $node$ )

```

The function COUNTCOVEREDLEAVES is implemented using a depth-first search [Ex. 14](#) and as such take $\mathcal{O}(|ST_S|) = \mathcal{O}(|S|^2)$ time to run, dominating the overall running time. By slightly adapting the construction of the suffix trie, one can get this overall complexity back to $\Theta(|P|)$ time [Ex. 15](#).

Locate Locating all occurrences can be done with by replacing COUNTCOVEREDLEAVES by RETRIEVECOVEREDLEAVES whose implementation are closely related: the only change is the information collected during the depth-first search. Unfortunately, no preprocessing can fasten the retrieval of covered leaves without degrading the space complexity: the algorithm for LocateAll hence runs in time $\mathcal{O}(|S|^2)$. However, if one is only interested into locating one and any of the occurrences of P in S , then $\mathcal{O}(|P|)$ running time is reachable [Ex. 16](#).

(2.5.2) • Other applications

While primarily designed to answer exact matching queries, it turns out that suffix tries are much more versatile. Here, we skim a few examples.

Longest repeating factor A repeating factor (RF) is a substring R that appears more than once in S . Reasoning on the suffix trie of S , we immediately see that the node that correspond to any repeating factor R must cover at least two leaves, that link to its multiple occurrences.

Now, observe that a longest RF necessarily correspond to a branching node: if it wasn't the case, taking one more step in the tree would give a longer RF, contradicting its length-maximality. Finally, the length of the factor being given by the depth of its corresponding node in the trie, we found our approach to solve the problem: longest repeating factors correspond to the deepest branching nodes in ST_S . It only remains to perform a depth-first search in time $\mathcal{O}(|S|)$ to find them.

Shortest substring occurring only once Conversely, the shortest substring that occurs only once in S correspond in ST_S to the highest node that covers a unique leaf. Similarly, a breath-first search find them in time $\mathcal{O}(|S|)$, while being practically faster in practice than depth-first search because of early returns.

Longest common prefix The longest common prefix of two suffixes $S_{[i..)}$ and $S_{[j..)}$ of S is $S_{[i..l]}$, where l is the largest integer such that: $\forall k \leq l, S_{[i+k]} = S_{[j+k]}$. Within the tree, the longest common prefix of two suffixes corresponds to the shared part of their respective root-to-leaf paths, that is the root-to-node path ending at the lowest common ancestors of the corresponding leaves. While being outside the scope of this course, it is possible to preprocess a tree so that lowest common ancestor queries can be answered in constant time, with impacting (asymptotically) the space required to store the tree. It follows that one can found the longest common prefix of any two suffixes of S in constant time.

(2.6) ◆ Querying the suffix tree

While its compact representation makes it less intuitive to manipulate, the suffix tree behaves very similarly to its trie equivalent: the algorithms on tries directly translate to their realm.

Besides saving space, it also improve their complexity. Recall the algorithm for `LocateAll`: it starts by recognizing the pattern, and then perform a DFS to recover the covered leaves. Here is the twist: as the suffix tree is only made of $\mathcal{O}(|S|)$ nodes (while they are $\mathcal{O}(|S|^2)$ of them in the suffix trie), its DFS runs in time linear in $|S|$ (instead of quadratic in the context of tries).

We now detail the algorithm for `Membership` on suffix trees, and let as an exercise [Ex. 17](#) the translation of algorithms from subsections [\[Léo: X and Y\]](#) to the suffix tree realm.

As said, the general behavior of the algorithm is the same. The only difference is that we need to recover the “labels of the edges” on the fly, as none is stored in the tree explicitly. To this end, we observed that the label of an edge is contained in any suffix whose root-to-leaf path go through this edge, or equivalently through the downmost node of this edge. Specifically, the label of (u, v) is $S_{[\text{SOMECOVEREDLEAF}(v)..][\text{depth}(u)..\text{depth}(v)]}$.

Let i be such that $P_{[i]}$ corresponded to a non-branching node in the suffix trie. It now correspond to the some point (strictly) within an edge $(\text{parent}(u), u)$ in the suffix tree. As a consequence, the next character to compare $P_{[i+1]}$ with is also contained within the $(\text{parent}(u), u)$ segment: one hence test whether $P_{[i+1]}$ equals $S_{[\text{someCoveredLeaf}(u)..][i]}$ or not. Now, if i is such that $P_{[i]}$ is branching, checking whether $P_{[i+1]}$ is first done as in the trie setting, and then the edge $(\text{parent}(u), u)$ we're in is updated.

This gives rise to the next algorithm, that also run in $\Theta(|P|)$ time.

Algorithm 7: Membership on suffix tree

Input: The suffix tree ST_S of S , a pattern P
Output: Whether P is a substring of S , or not

```

1:  $u \leftarrow \text{root}(ST_S)$ 
2: for  $i \in [0..|P|)$  do
3:      $\triangleright$  If we are at a branching node, branch as dictated by the prefix (if possible)  $\triangleleft$ 
4:      $\triangleright$  Otherwise, check the characters that are on the branch  $\triangleleft$ 
5:     if  $i \geq \text{depth}(\text{node})$  then
6:         if  $\text{child}(u, P_{[i]})$  doesn't exist then
7:             return False
8:          $u \leftarrow \text{child}(u, P_{[i]})$ 
9:     else
10:        if  $S_{[\text{someCoveredLeaf}(u)+i]} \neq P_{[i]}$  then
11:            return False
12: return True
    
```

(2.7) **◆ Generalizing suffix trees to sets of strings**

So far, a suffix tree were attached to a specific string S . It can be extended naturally to set of string.

Definition 6 (Generalized suffix tree). *The generalized suffix tree of a set of strings $\mathcal{S} = \{S_1, \dots, S_n\}$ over an alphabet Σ , denoted $ST_{\mathcal{S}}$ is the compact trie of $\bigcup_{0 < k \leq |\mathcal{S}|} \text{Suffs}(S_k \$_k)$, where all $\$_k$ are distinct terminasion symbols that doesn't belong to Σ . It verifies the property*

$$\forall k, s \in \text{Suffs}(S_k \$_k) \iff s \text{ spells a root-to-leaf path ended by } \$_k \text{ in } ST_{\mathcal{S}}.$$

Additionally, its leaves are decorated with the starting position of their corresponding suffix in S .

Its construction can be done in time $\mathcal{O}(\sum_{S \in \mathcal{S}} |S|)$

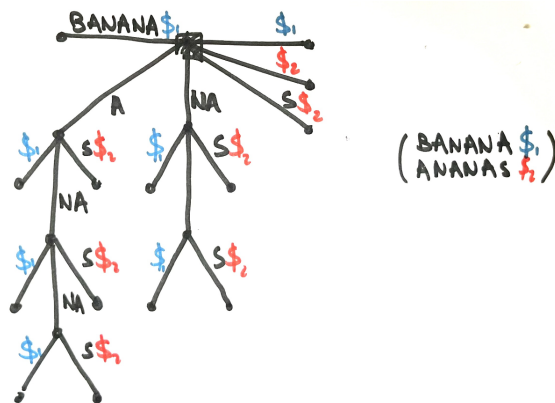


Figure 5: The generalized suffix tree of {BANANA, ANANAS}

(2.8) ◆ Discussion

One limitation of the suffix tree is their dependence, in size, to the alphabet size. Indeed, when analysing the algorithm, we (implicitly) assumed that checking the existence of a child for some letter was a constant time operation: this suggests that all node stores a $|\Sigma|$ long array, where each cell possibly contain a child. While this seems reasonable at the top levels of the tree, this may cause some loss in space at the lower levels, as the degree of node will likely decrease. A tradeoff between space and time can be found by changing those arrays to $\# \text{childs}(\text{node})$, at the cost of having a $\log |\# \text{childs}(\text{nodes})|$ cost for advancing through the node.

Another limitation is to be found in the linear time construction. Indeed, in practice the tree won't fit in the cache: the real cost of following suffix-links is way higher than other $O(1)$ operations.

Also, while taking space $\mathcal{O}(|\Sigma| \cdot |S|)$ it is clear that it demands way more space than storing S , because of the number of times we add linear space informations (depth, someCoveredleaf, starting index in leaf, ...).

(3) ■ Suffix array

In this section, Σ is ordered and $\$ \notin \Sigma$, the termination symbol, is considered smaller than all letters. We introduce a new data structure of interest, whose capabilities are comparable to the suffix tree while requiring much less space in practice.

Definition 7 (Suffix array). *The suffix array of a string S , denoted SA_S , is the array of the s starting positions of the lexicographically sorted suffixes of $S\$$, that is*

$$\forall i \in [0..|S\$| - 1), (S\$)_{[SA[i]..)} \preceq_{lex} (S\$)_{[SA[i+1]..)}.$$



Figure 6: The suffix array of BANANA

Assuming that $\log |S|$, the number of bit needed to store the starting index $i \in [0..|S|)$ of a suffix, is smaller than a word size, the suffix array can be store using $|S|$ computer words, without any dependence of the size of the alphabet.

While there exists direct construction of the suffix array that are space optimal, in the sense that the space needed by the algorithm is the space of the suffix array plus some constant extra space, we focus here on a simpler construction that highlight the connection between suffix trees and suffix arrays.

Algorithm 8: Construction of the suffix array**Input:** The suffix tree ST_S of S **Output:** The suffix array SA_S of S .

```

1:  $node \leftarrow \text{root}(ST_S)$ 
2:  $SA_S \leftarrow \text{NEWARRAYOFSIZE}(|S| + 1), i \leftarrow 0$ 
3: while  $i < |S| + 1$  do
4:    $node \leftarrow \text{NEXTSTEPDFS}(node)$   $\triangleright$  This step never fails Ex. 19
5:   if  $node$  is a leaf then
6:      $SL \leftarrow SL + [\text{GETSUFFIXSTART}(node)]$ 

```

The running time of this algorithm crucially depends on the complexity of `NEXTSTEPDFS`, which is itself linked to the implementation choices of the suffix tree for storing children. On the one hand, if children of u are stored within sorted array of length $\# \text{Children}(u)$, then this step is always constant time. On the other hand, if children of u are stored within array of length $|\Sigma|$, this step take $\mathcal{O}(|\Sigma|)$ in the worst case. But with a preprocess of the tree in time $\mathcal{O}(|S| \cdot |\Sigma|)$, one can transform such an instance into one of the first kind. The overall running time thus ranges from $\mathcal{O}(|S|)$ to $\mathcal{O}(|S| \cdot |K|)$, depending on the design choices.

(3.1) ◆ Searching the array

The most important characteristic of the suffix-array, that we will leverage for search, is a consequence of the suffixes being sorted.

💡 Key idea. If P occurs in S , then its occurrences are contiguous in SA_S .

Let demonstrate how to use this idea for solving our Membership, Count and Locate problems. As before, the steps consisting in “locating the patterns” will be the same for all three, but the post-processing of this information will slightly differ. Informally, we will track the range $[i_{top}..i_{bottom}]$ of the suffix array where the pattern has a chance to happen. As the suffix array is sorted, we will iteratively tighten this range using dichotomic searches.

Algorithm 9: Pattern matching with the suffix array**Input:** The suffix array SA_S of a string S , a pattern P

```

1:  $first \leftarrow \text{BINARYSEARCH}(P, SA_S, \text{First})$ 
2: return True iff none  $first \neq \perp$  ▷ Membership
3:  $last \leftarrow \text{BINARYSEARCH}(P, SA_S, \text{Last})$ 
4: return  $last - first + 1$  if the makes sense, 0 otherwise ▷ Count
5: return  $SA_{[first..last]}$  if the makes sense, 0 otherwise ▷ Locate

6: function BINARYSEARCHSA( $P, SA_S, mode$ )
7:    $(i_{min}, i_{max}) \leftarrow (0, |S| - 1)$ 
8:   while  $i_{min} \neq i_{max}$  do
9:      $i_{mid} \leftarrow \lfloor (i_{min} + i_{max}) / 2 \rfloor$  ▷ If mode = First
10:     $i_{mid} \leftarrow \lceil (i_{min} + i_{max}) / 2 \rceil$  ▷ If mode = Last
11:    if  $S[SA_S[i_{mid}..]] < P$  then
12:       $i_{min} \leftarrow i_{mid} + 1$ 
13:    else if  $S[SA_S[i_{mid}..]] > P$  then
14:       $i_{max} \leftarrow i_{mid} - 1$ 
15:    else if  $S[SA_S[i_{mid}..]] = P$  then
16:       $i_{max} \leftarrow i_{mid}$  ▷ If mode = First
17:       $i_{min} \leftarrow i_{mid}$  ▷ If mode = Last
18:    if  $S[SA_S[i_{min}..]] = P$  then
19:      return  $i_{min}$ 
20:    else
21:      return  $\perp$ 

```

This algorithm clearly runs in time $\Theta(|P| \log |S|)$ for Membership and Count, as each comparison in the dichotomic search take $\mathcal{O}(|P|)$ in the worst case. This time turns to $\Theta(|P| \log |S| + \#occ_S)$ for Locate. Note however that this bound is quite pessimistic: morally, if only a very few long prefixes of P appears in T , then the expected time for a comparison is no longer $|P|$ but rather constant! Hence, for example if P is a random string that occurs exactly once in T , one can show [Ex. 20](#) that the expected running time is $\mathcal{O}(|P| + \log |T|)$.

(3.1.1) • Speeding up comparisons

Longest common prefixes Here is a small change in the algorithm that, while letting untouched the asymptotical complexity, drastically improves its performances in practice. The idea is to get rid of unnecessary character comparison while comparing P to some $S[SA_S[i_{mid}..]]$. For that, observe that if the longest common prefix of $SA_S[i_{min}]$ and $SA_S[i_{max}]$ — confounding index with the suffix they correspond to, for readability — is lcp , then lcp is both a prefix of P and of $SA_S[i_{mid}]$: the first $|lcp|$ comparison will always succeed and can hence safely be skipped. Moreover, as

$$\text{lcp}(SA_S[i_{min}], SA_S[i_{max}]) = \min \left\{ \begin{array}{l} \text{lcp}(SA_S[i_{min}], SA_S[i_{mid}]) \\ \text{lcp}(SA_S[i_{mid}], SA_S[i_{max}]) \end{array} \right\},$$

one can just adapt the comparison step so that it also returns the length lcp of the two compared elements as follows.

Algorithm 10: Comparing strings with lcp length**Input:** Two strings S and S' , a lower bound ℓ on the length of $\text{lcp}(S, S')$ **Output:** An order of S and S' , together with $|\text{lcp}(S, S')|$

```

1:  $i \leftarrow \ell + 1$ 
2: while  $i \leq \min\{|S|, |S'|\}$  do
3:   if  $S_{[i]} \neq S'_{[i]}$  then
4:     return (" $>$ ",  $i - 1$ ) or (" $<$ ",  $i - 1$ )
5:    $i \leftarrow i + 1$ 
6: return (" $>$ ",  $i - 1$ ), (" $<$ ",  $i - 1$ ), or (" $=$ ",  $i - 1$ )    ▷ Comparing  $|S|$  and  $|S'|$ 

```

Asymptotically faster comparisons Despite its practical capabilities, the former speed-up doesn't improve the asymptotical complexity. This is because $\text{lcp}(\text{SA}_S[\min], \text{SA}_S[\max])$ can be way smaller [Ex. 21](#) than:

$$\max\{\text{lcp}(\text{SA}_S[\min], \text{SA}_S[\text{mid}]), \text{lcp}(\text{SA}_S[\text{mid}], \text{SA}_S[\max])\}.$$

As a consequence, characters of P can be compared many times. There exists a more involved comparison algorithm (that won't be described here) that ensures that, each iteration, there is at most one compared character that was compared before. This ensures that at most $\mathcal{O}(n)$ character comparisons are performed during the binary search. The resulting time complexity is $\mathcal{O}(|P| + \log |T|)$ in the worst case.

(3.1.2) • Enhancing the suffix array

In this simplest version, the suffix array may look deceptive because it doesn't support the wide variety of queries handled by the suffix tree. This is not a surprise, as the suffix array only contains the leaf order of the suffix tree: a lot of information is lost on the way. Fortunately, it can be augmented with a few extra arrays to make it as powerful as the suffix trees.

LCP array In order to capture part of the topology of the tree, and motivated by the centrality of lowest common ancestors in the suffix tree applications, one can desire to store the longest common prefixes.

Definition 8 (LCP array). *The longest common prefix (LCP) array stores the lengths of the longest common prefixes between all pairs of consecutive suffixes. Namely,*

$$\text{LCP}_S[i] = |\text{lcp}(S_{[\text{SA}_S[i]..]}, S_{[\text{SA}_S[i+1]..]})|.$$

It can be constructed naively in time $\mathcal{O}(|S|^2)$ or derived from the suffix tree in time $\mathcal{O}(|S|)$ [Ex. 22](#). There is also a direct linear time construction, known as Kasai's algorithm, we won't develop here. It enables the search of longest repeated substring, shortest unique substring and longest common substrings.

A compact representation of the suffix tree By adding two more arrays, one can make a data structure fully equivalent to the suffix tree, in the sense that any problem solvable in the suffix tree is also on this new datastructure with the same time complexity. Intuitively, these extra arrays end to capture the topology of the suffix tree.

More precisely, a node n in the suffix tree is associated to the interval of occurrences of the word spelt by the root-to- n path, and the remaining arrays encode the child and suffix links using this association.

(4) ■ Searching over compressed space

This section is hidden until next lecture.

In order to further reduce the space requirement of our scheme, one could try work on a space-efficient representation of S , rather than S itself. This may sound abstract, so let consider a motivational example. Let S to be the string made of a million of 'A'. The previous sections tell us that one can build a $\mathcal{O}(|S|)$ suffix array/tree to answer pattern matching question efficiently. But here is a much more compact algorithm that answer those question directly:

Algorithm 11: Pattern matching on homopolymers

Input: A string S made of a single (repeated) letter, a pattern P

```

1: letter ← S[0]
2: if P is only made of letter then
3:     return True                                     ▷ for Membership
4:     return |S| - |P| + 1                             ▷ for Count
5:     return [0..|S| - |P| + 1)                       ▷ for LocateAll

```

Of course, this is very specific to this kind of string. But the characteristic that a string can take less characters to be described than to be written extensively is common to many others, and relates to the domain of text compression.

(4.1) ◆ Compressing repetitive strings

The intuitive definition we gave of compressibility can be formalized with respect to the notion of Kolmogorov complexity.

Definition 9 (Kolmogorov complexity). *The Kolmogorov complexity of a string S , denoted $\mathcal{K}(S)$, is the size of smallest program that produces S .*

While this value cannot be computed, it helps us drive design of efficient compression scheme for the repetitive strings we observe in practice in biology. It allows for example to set aside compression scheme based on character entropy, that encode the most seen letter using less bits in order to reduce the overall cost. Indeed, for a string S , we expect

$$\# \text{ bits}(\text{entrComp}(S^n)) \approx n \cdot \# \text{ bits}(\text{entrComp}(S)),$$

while

$$\mathcal{K}(S^n) \leq \mathcal{K}(S) + \mathcal{O}(\log n),$$

as one can build a program that, looping, prints n times in a row the string S to produce S^n .

Different family of compression methods that leverage the repetitions within string have arisen: lempel-ziv methods, bidirectional macro schemes, grammar compressors, etc. In this course we focus on the Burrows-Wheeler transform, a (reversible) permutation of the string that greatly enhance its compression, together with an index built on top of it, the FM-index, to answer pattern matching queries.

(4.2) **◆ The Burrow-Wheeler transform**

Let start with the transform, defined as follows.

Definition 10 (Burrows-Wheeler transform). *The Burrow-Wheeler transform of a string S , denoted BWT_S , is the string corresponding to last column of the matrix whose rows are the $|S| + 1$ rotations of $S\$$ sorted lexicographically, where $\$ \in \Sigma$ is a termination symbol smaller than the others.*

This definition immediately gives an algorithm for constructing the Burrows-Wheeler transform in time $\mathcal{O}(|S|^2 \log(|S|))$ and space $\mathcal{O}(|S|^2)$ **Ex. 23**. It also allows to see that the Burrows-Wheeler is a permutation of S .

But this naive bound can greatly enhanced by making the following observation.

Key idea. The order of the lexicographically sorted rotations of $|S|$ is the order of its lexicographically sorted suffixes.

The BWT is thus highly related to the suffix array, and can also be defined relatively.

Definition 11 (Burrows-Wheeler transform, alternative definition). *The Burrow-Wheeler transform of a string S , denoted BWT_S , is the string BWT_S of length $(|S| + 1)$ defined by*

$$(BWT_S)_{[i]} = \begin{cases} S_{[SA_S[i]-1]} & \text{if } SA_S[i] > 0 \\ \$ & \text{otherwise} \end{cases}.$$

This definition gives a linear time algorithm for the computation of the Burrow-Wheeler transform.

(4.2.1) **• Reversibility**

Recall that the purpose of the Burrows-Wheeler transform of a string aims to describes it with an as-small-as-possible description. It is thus of utmost importance that one can recover the original string from its transform.

Reconstructing the BW matrix This approach consists in retrieving the BW matrix from the transform, and then return the first row deprived from its first character, which is $\$$ by construction.

Given BWT_S , which correspond to the last column of the BW matrix, one can easily reconstruct the first column. Indeed, we know that the rows of the matrix are lexicographically sorted so the first column correspond to the lexicographically sorted letters. Easy. Let's iterate this reasoning. The first two columns of the BW matrix correspond of the 2-mers of $S\$$ seen as a circular word, sorted lexicographically. So, if we are able to recover these 2-mers from the first and last column, we get one step further. Here is how: as a row r represents rotations of $S\$$, we get that $r[0]$ is preceded by $r[-1]$ is the circular world $S\$$, ie. that $r[-1]r[0]$ is a 2-mers of the circular world $S\$$. We then continue with 3-mers, 4-mers, until the final step that consists in sorting the $(|S| + 1)$ -mers of $S\$$. Formally, this gives the following algorithm.

Algorithm 12: Inverting the Burrow-Wheeler transform (naive)

Input: The BW transform BWT_S of some string S

Output: The string S

```

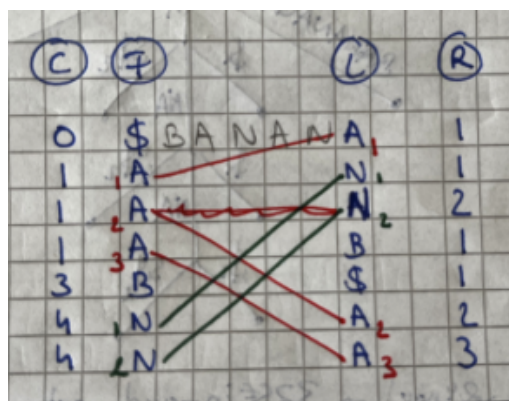
1:  $bwm \leftarrow \text{EMPTYMATRIX}$ 
2: loop  $(|S| + 1)$  times
3:    $bwm \leftarrow [bwm \mid \text{asColumn}(BWT_S)]$ 
4:    $\text{LEXSORTROWS}(bwm)$ 
5: return  $bwm[0][1..]$ 
    
```

The former algorithm perform $(|S| + 1)$ sorts of rows. As comparing two rows requires $\mathcal{O}(|S|)$ time, the overall running time is $\mathcal{O}(|S|^3 \cdot \log |T|)$. It also requires $\mathcal{O}(|S|^2)$ space to store the matrix

By leveraging the LF mapping While the former algorithm is great to get familiar with the BW transform, its time and space complexities are prohibitive. We propose here another construction that relies and the following central property of the BW transform, known as the LF-mapping — standing for LastFirst-mapping.

Lemma 12 (LF-mapping). *For every character $a \in \Sigma$, the i -th occurrence of a in L and the i -th occurrence of a in F correspond to the same character in T .*

Proof. For $a \in \Sigma$, let $aX \prec_{\text{lex}} aY$ be two suffixes of T . Clearly, ordering of the suffixes holds if and only if $X \prec_{\text{lex}} Y$. So, there is a bijection from the suffixes that start with a and those that are preceded with a that preserves the relative order of these suffixes. ■



This property allows to reconstruct S directly from BWT_S , by navigating between the first (F) and last (L) column of the matrix, preventing the need of huge sorts and matrix reconstruction. Starting by $\$$, we will iteratively find the preceding character in $S\$$, hence reconstructing S . The two ingredients that operate are **(1)** the row mapping, that allows to find the preceding character of any letter occurrence in F by reading the character of L within the same row, and **(2)** the LF-mapping property, that allows to go from a character of L to the same in F . In order to make the LF-mapping step efficient, we store two auxiliary structures that are trivially computable in linear time:

- R is the rank array, and is such that the letter seen at position i in BWT_S is the $R[i]$ -th of its kind to appear in BWT_S (going left-to-right). Formally,

$$R[i] = |\{j \mid j \in [0..i], (BWT_S)_{[j]} = (BWT_S)_{[i]}\}|;$$

- C is the cumulative count map, such that for $x \in \Sigma \cup \{\$\}$ the number $C[x]$ is the amount of letters of BWT_S that are strictly lexicographically smaller than x . Formally,

$$C[x] = |\{j \mid j \in [0..|S| + 1), (BWT_S)_{[j]} \prec_{\text{lex}} x\}|.$$

Equivalently, this is the index of the first occurrence of the character x in F .

This gives the following algorithm for the BW transform inversion, that no longer relies on the BW transform alone, but on the FM-index, that augments the BW transform with the two aforementioned arrays.

Algorithm 13: Inverting the Burrow-Wheeler transform (fast)

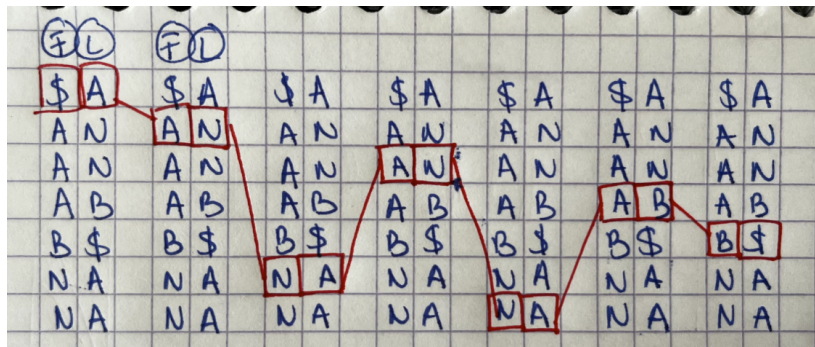
Input: The FM-index (BWT_S, R, C) of a string S .

Output: The string S such that $BWT_S = L$

```

1:  $S \leftarrow "\$"$ ,  $i_{row} \leftarrow 0$ 
2: loop  $|S|$  times
3:    $prec \leftarrow BWT_S[i_{row}]$ 
4:    $S \leftarrow prec + S$ 
5:    $i_{row} \leftarrow \text{LFMAPPING}(i_{row})$ 
6: return  $S$ 

7: function  $\text{LFMAPPING}(i)$ 
8:    $\leftarrow C[BWT_S[i]] + R[i] - 1$   $\triangleright$  Ranks starts at 1 while indexes at 0 (hence -1
    
```



(4.3) **◆ Pattern matching with the FM-index**

Perhaps surprisingly, we just saw the core mechanism to search for a pattern using the FM-index. After all, inverting the BW transform was exactly the task of recovering the unique $(|S| + 1)$ long substring of $S\$$ that ends with $\$$. We generalize the idea.

Let $P = p_0 \dots p_{|P|-1}$ be the pattern to search. We will update iteratively the index i_{min} and i_{max} such that at the k -th iteration the letters of $F_{[i_{min}..i_{max}]}$ that **(1)** are equal to $p_{|P|-k}$, and **(2)** are followed by $p_{|P|-k+1} \dots p_{|P|-1}$ in $S\$$. Therefore, we start by locating the occurrences of $p_{|P|}$ in F using the cumulative count array: they form the range

$$[i_{min}..i_{max}] = [C[p_{|P|}]..C[\text{nextSmallestLetter}(p_{|P|})]].$$

Then, we look at the corresponding range within L and shorten it by making its extremities to be the first and last occurrences of $p_{|P|-1}$ in L . This range is then mapped back to F using the LF-mapping, and now correspond to a range of contiguous $p_{|P|-1}$ in F . The process is then

iterated. At any point, if the range becomes empty this means that P doesn't match anywhere in S . At the end, all the people in the range are matches. Formally,

Algorithm 14: Pattern matching on the BW transform

Input: The FM-index (BWT_S, R, C) of a string S , a pattern P .

```

1:  $(i_{min}, i_{max}) \leftarrow (C[P_{[-1]}], C[\text{nextSmallestLetter}(P_{[-1]})] - 1)$ 
2: if  $(i_{min}, i_{max}) = \perp$  then
3:   return False (resp. 0) ▷ Membership (resp. Count)
4:
5: for  $k \in [0..|P| - 1)$  in reverse order do
6:    $oldRange \leftarrow [i_{min}..i_{max}]$ 
7:    $i_{min} \leftarrow \text{FIRSTOCCWITHIN}(P_{[k]}, L[oldRange])$ 
8:    $i_{max} \leftarrow \text{LASTOCCWITHIN}(P_{[k]}, L[oldRange])$ 
9:   if  $(i_{min}, i_{max}) = \perp$  then
10:    return False (resp. 0) ▷ Membership (resp. Count)
11:     $(i_{min}, i_{max}) \leftarrow (\text{LFMAPPING}(i_{min}), \text{LFMAPPING}(i_{min}))$ 
12: return True (resp.  $i_{max} - i_{min} + 1$ ) ▷ Membership (resp. Count)

```

The pattern search is done in time $\mathcal{O}(|P| \cdot \mathcal{C}(\text{FIRSTOCCWITHIN}))$, which is $\mathcal{O}(|P| \cdot |S|)$.

This complexity can be enhanced by refining the rank array, and having one such array per letter. Namely, we define $|\Sigma|$ such arrays by

$$\forall x \in \Sigma, R_x[i] = |\{j \mid j \in [0..i], (\text{BWT}_S)_{[j]} = x\}|.$$

This allows to get rid of the `FIRSTOCCWITHIN` [Ex. 24](#), giving the corresponding algorithm.

Algorithm 15: (better) Pattern matching on the BW transform

Input: The FM-index $(\text{BWT}_S, \{R_x\}_{x \in \Sigma}, C)$ of a string S , a pattern P .

```

1:  $(i_{min}, i_{max}) \leftarrow (C[P_{[-1]}], C[\text{nextSmallestLetter}(P_{[-1]})] - 1)$ 
2: if  $(i_{min}, i_{max}) = \perp$  then
3:   return False (resp. 0) ▷ Membership (resp. Count)
4:
5: for  $k \in [0..|P| - 1)$  in reverse order do
6:   ▷ LF-mapping, but shrinks the window to matched character ◁
7:    $i_{min} \leftarrow C[P_{[k]}] + R_{P_{[k]}}[i_{min} - 1] + 1 - 1$ 
8:    $i_{max} \leftarrow C[P_{[k]}] + R_{P_{[k]}}[i_{max}] - 1$ 
9:   if  $i_{min} > i_{max}$  then
10:    return False (resp. 0) ▷ Membership (resp. Count)
11: return True (resp.  $i_{max} - i_{min} + 1$ ) ▷ Membership (resp. Count)

```

In order to locate occurrences, one can simply add a suffix array [Ex. 25](#). The complexity of reporting the occurrences is then $\mathcal{O}(\#occs)$.

(4.3.1) • Compression property of the BWT

- The occurrences of the suffixes of T that starts with X will, by definition of SA, appear contiguously in the suffix array. We call this section the X -interval. \Rightarrow many aX in T implies many in X -interval, which gives good chance of runs. T has many repeated

substrings => many U-intervals mostly same character - $L = \text{bwt}(T)$ has few runs => runlength encoding (RLE) is good

An example: the U-interval for $U = \text{he} + \text{emptyspace}$ in an English text -> many the's, some he, she, The => long runs of 't', cutted by some h, s, T.

RLE.

MoveToFront.

(4.3.2) • **Subsampling**

- Subsampling rank, subsampling SA (+ resolving holes). cf s68 Pierre.

(5) ■ **Searching over sketched space**

(5.1) ◆ **Bloom filters**

Quotient filters

■ **Chapter exercises**

- Ex. 9 Find an instance (S, k) for which the construction of the index takes $O(|S|^2)$.
- Ex. 10 By decomposing a pattern P into pieces of length at most k , devise an algorithm that search for long patterns based on k -mer indexes. Estimate its complexity, and conclude.
- Ex. 11 Give a mathematical formula or an algorithm that computes j_{kmer} given $kmer$.
- Ex. 12 Derive the $\mathcal{O}(|\Sigma|^k + |S|)$ space requirement of the alternative approach for k -mer indexing
- Ex. 13 Propose an algorithm that tests whether P is a prefix of some string S . What is its complexity?
- Ex. 14 Prove the (tight) worst-case time complexity of Algorithm 4.
- Ex. 15 Prove the (tight) upperbound on the number of leaves in the trie produced by Algorithm 4.
- Ex. 16 Show that, remarkably, the compact trie of $\text{Suffs}(S\$)$ only admit branching internal nodes.

- Ex. 17** What would be the space needed by the suffix trie of the human genome? Of an E. Coli genome?
- Ex. 18** Build the suffix trie of “abracadabra”.
- Ex. 19** Show that, remarkably, the compact trie of $\text{Suffs}(S\$)$ only admit branching internal nodes.
- Ex. 20** Make these slight changes explicit, and argue they have no impact on the complexity.
- Ex. 22** Propose an algorithm for `COUNTCOVEREDLEAVES`.
- Ex. 23** Propose an algorithm for construction the suffix trie that still run in $\mathcal{O}(|S|^2)$ time but allow for a $\Theta(|P|)$ time algorithm for `Count`.
- Ex. 24** Adapt the construction of the suffix trie so that `LocateOne` can be solved in $\mathcal{O}(|P|)$ time.
- Ex. 25** Adapt the algorithms from subsection X and Y, so that they run directly on suffix trees.
- Ex. 27** Shows that we never call `NEXTSTEPDFS` when the DFS is finished.
- Ex. 28** (*hard*) Let P and S be two random strings such that P appears exactly once in S . Show that the expected time for comparing P to some $S_{[\text{SA}_S[i_{mid}]\dots]}$ is constant. Conclude on the expected running time complexity.
- Ex. 29** Propose an instance for the binary search algorithm where ℓ is never updated until the very last iteration.
- Ex. 30** Propose an algorithm that build the LCP array of S using the suffix tree of S
- Ex. 31** Derive these complexities after expliciting the algorithm derived from the definion.
- Ex. 32** Prove step correctness of steps 6/7
- Ex. 33** Propose an algorithm for `LocateAll` that relies on the FM-index, augmented with a suffix array.

