BOX

Pattern matching

Léo Ackermann

Pattern matching

Intuition. Looking for a given pattern within a given string.

Definition (Pattern matching variants).

harder than

Input: a string S and a pattern P

Output (Membership): whether P appears in S as a substring

Output (Count): the number of occurrences of P in S

Output (LocateAll): the starting positions of P in S, that is the i's such that $S_{[i..i+|P|)} = P$

Naive approach.

Algorithm 1: Naive algorithm for LocateAll

```
occ ← []
for k ∈ [0..|S| − |P|) do
    if Occurs(P, S, k) then
        return occ = occ + [k]
return occ

function Occurs(P, S, k):
    for i ∈ [0..|P|) do
        if P[i] ≠ S[k+i] then
            return False
    return True
```

➡ $\mathcal{O}(|S| \cdot |P|)$ time

Pattern matching as a routine task

Motivation. Pattern matching is rarely an isolated task

[A] Preprocessing pattern, for a $((P, S_i))_{i \in \mathbb{N}}$ instance list

Application. Epidemic surveillance

1. Sequence sick persons' genomes
2. Locate the virus (fixed pattern) within the latter
3. Look whether/how the virus evolved

[B] Preprocessing genome, for a $((P_i, S))_{i \in \mathbb{N}}$ instance list

Application. Genetic condition diagnosis

1. Sequence a human genome
2. Search for fragments within preprocessed reference genomes (whose conditions are known)
3. Aggregate and diagnose

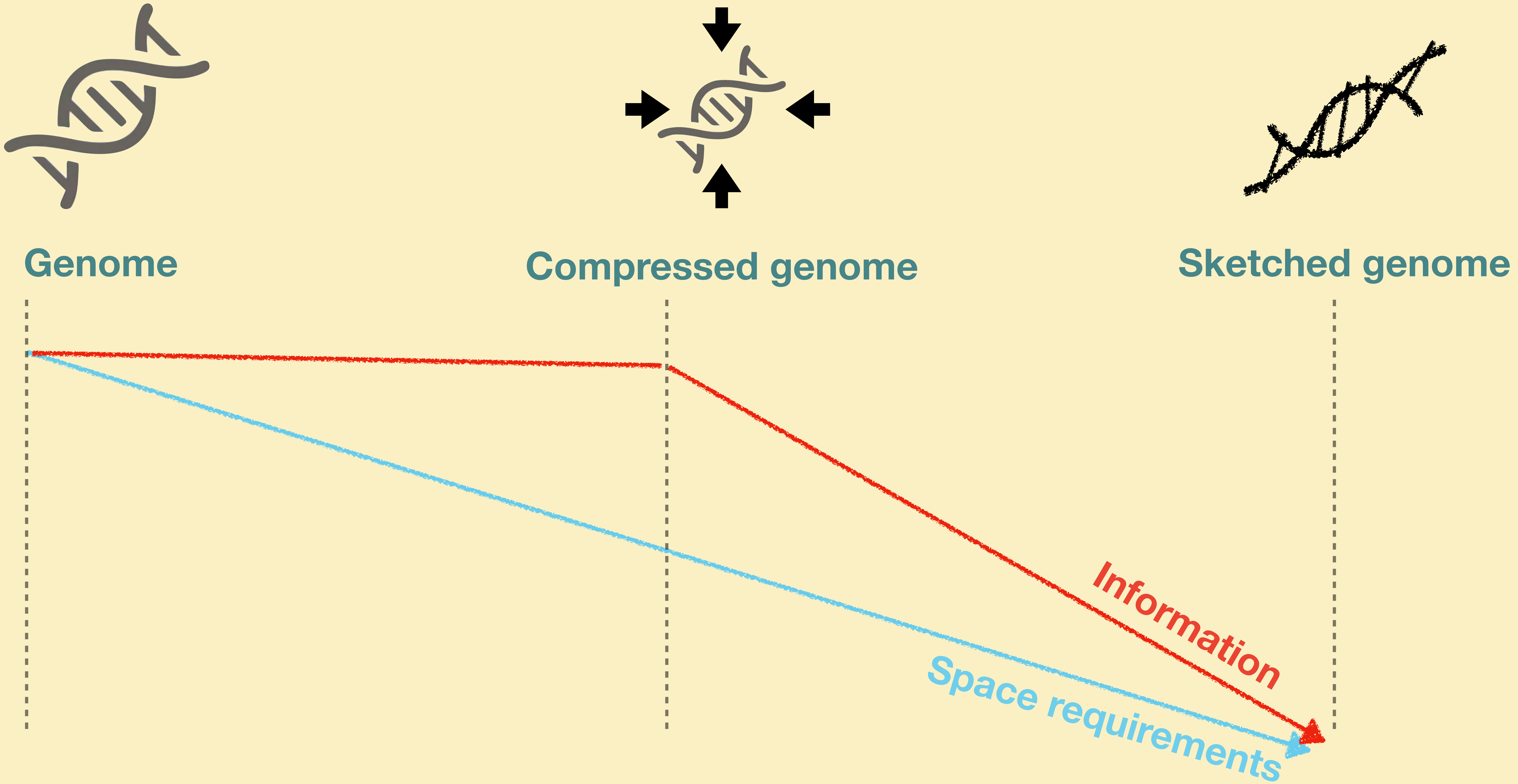
Application. In-depth study of a genome

1. Search for specific sequences, tandem repeats, palindromes, ...

Today's program

Outline

Various genome representations.



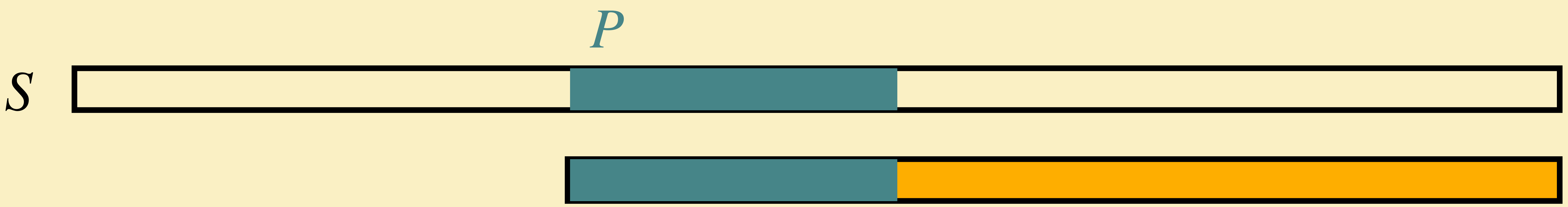
Query (membership/count/locate) complexity around $\mathcal{O}(P + |output|)$

Part PS-A

Preprocessing string - Searching with suffixes

Main observation

Key idea. A pattern P is a substring of a string S iff P is the prefix of a suffix of S.




```
function OCCURS(P, S, k):
    for i ∈ [0..|P|) do
        if P[i] ≠ S[k+i] then
            return False
    return True
```

➡ Prefix matching (easier) in the suffixes of P, simultaneously

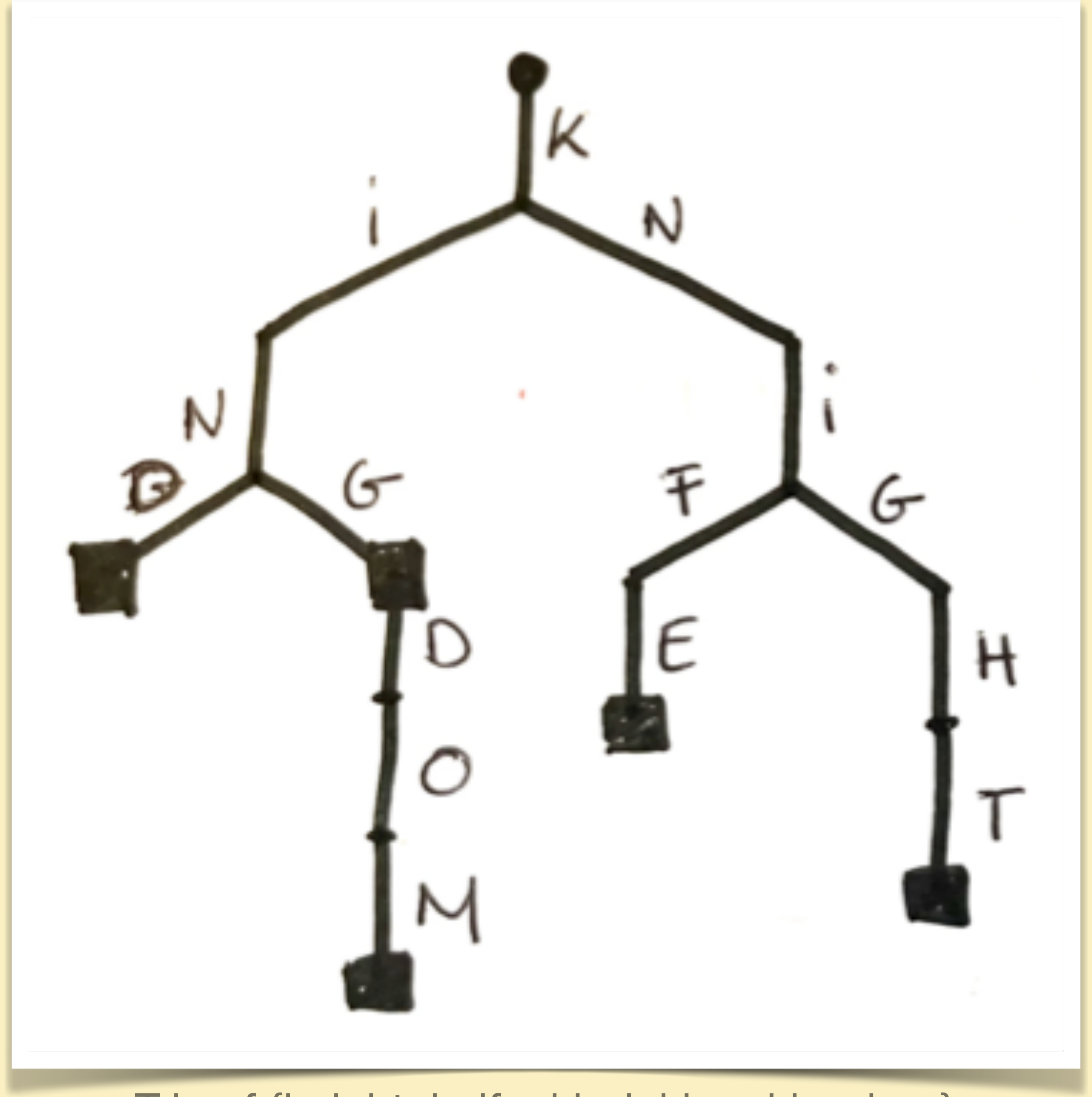
Suffix tries

Tries

Definition (Trie).

A trie that represent a set of strings $\mathcal{S}$ is a tree of at most $|\mathcal{S}|$ leaves and $|\mathcal{S}|$ marked nodes whose edges are labeled by letters of Σ and such that:

- (membership) $S \in \mathcal{S}$ iff S spells a path from the root to a marked node
- (compaction 1) Outgoing edges of a given nodes are decorated by different letters
- (compaction 2) Every leaf is marked



Trie of {knight, knife, kind, king, kingdom}

Membership (algo). Try to unroll the word from the root of the tree.

$\Longrightarrow \mathcal{O}(|S|)$ time

Build (algo).

Algorithm 4: Construction of the trie of $\mathcal{S}$

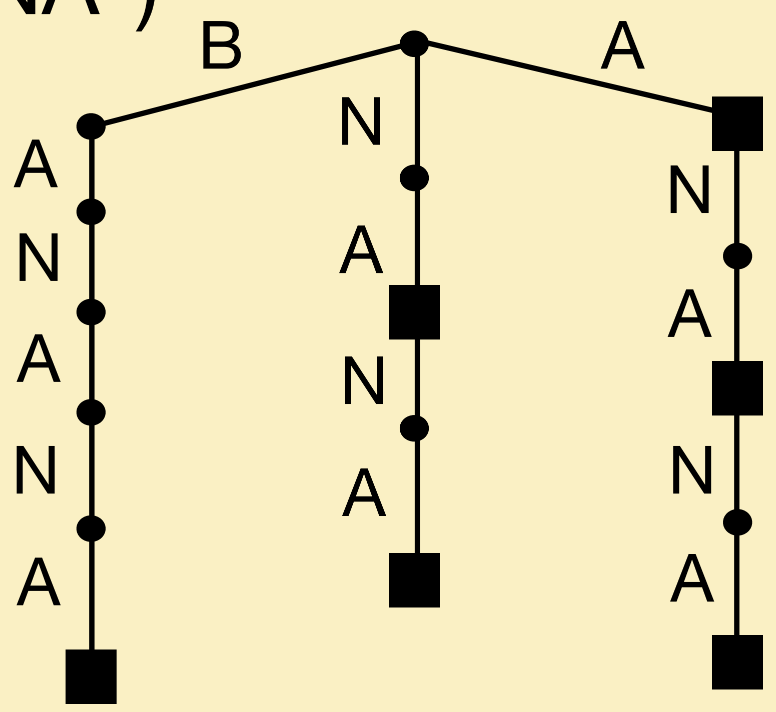
```
T ← NEWUNMARKEDROOT
for S ∈ 𝒮 do
    i_letter ← 0, node ← root(T)
    for i ∈ [0..|S|) do
        if child(node, S[i]) doesn't exists then
            CREATEUNMARKEDCHILD(node, S[i])
        node ← child(node, S[i])
    MARK(node)
return T
```

$\Longrightarrow \mathcal{O}(\sum_{S \in \mathcal{S}} |S|)$ space and time

Suffix tries

Naive idea. The suffix trie of S is the trie of its suffixes.

PROBLEM. eg. ST("BANANA")

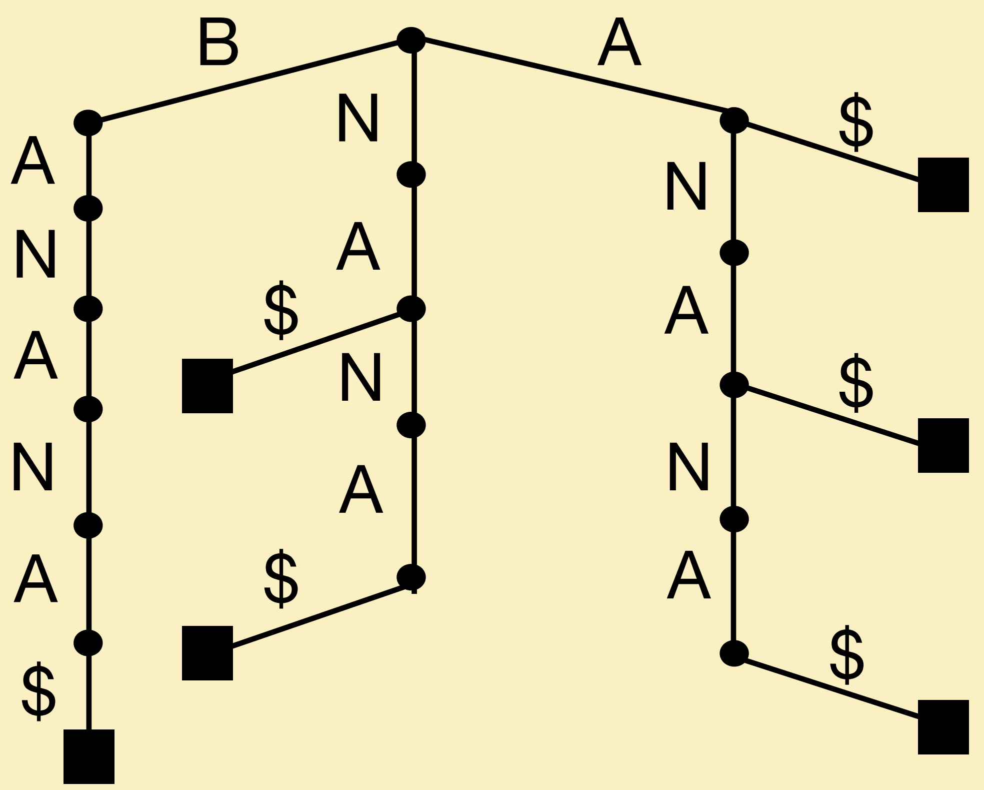


Not all suffixes are created equal!

Definition (Suffix trie).

> The suffix tree of S is the trie of the suffixes of $S\$$, where $\$$ is a fresh symbol that doesn't appear in S, called the termination symbol.

eg. ST("BANANA")



$\Longrightarrow \mathcal{O}(|S|^2)$ space

$\Longrightarrow \mathcal{O}(|S|^2)$ time to build

Pattern matching in suffix tries (Membership)

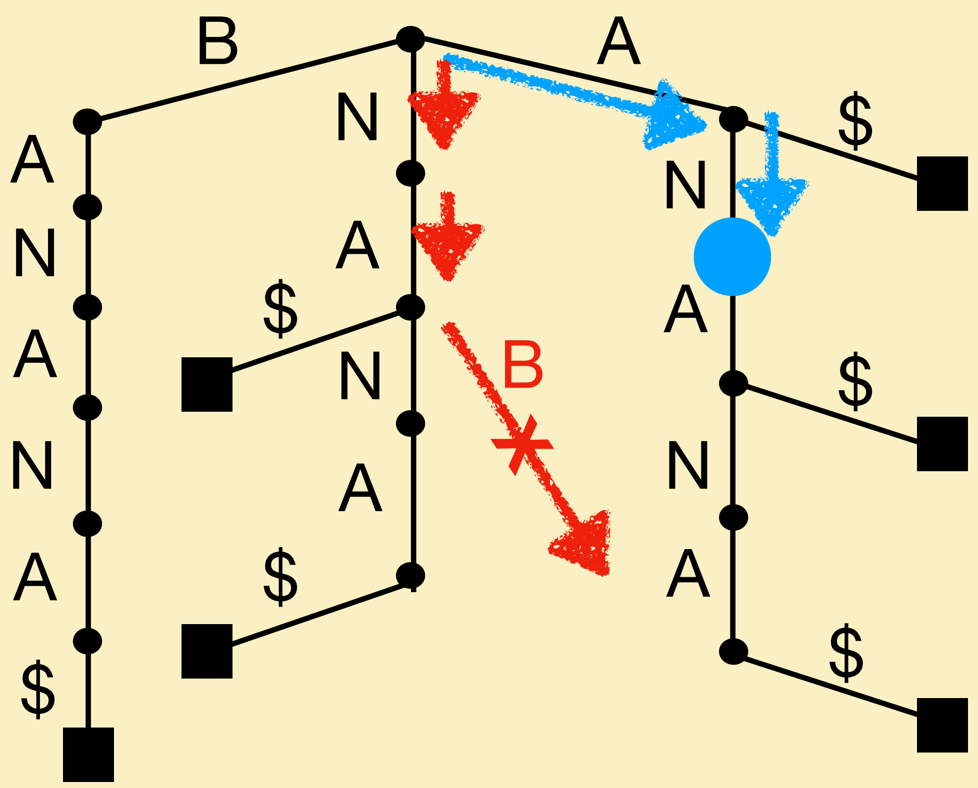
Algorithm 5: Membership on suffix trie

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

```
node ← root(ST_S)
for i ∈ [0..|P|) do
    if child(node, P[i]) doesn't exists then
        return False
    node ← child(node, P[i])
return True
```

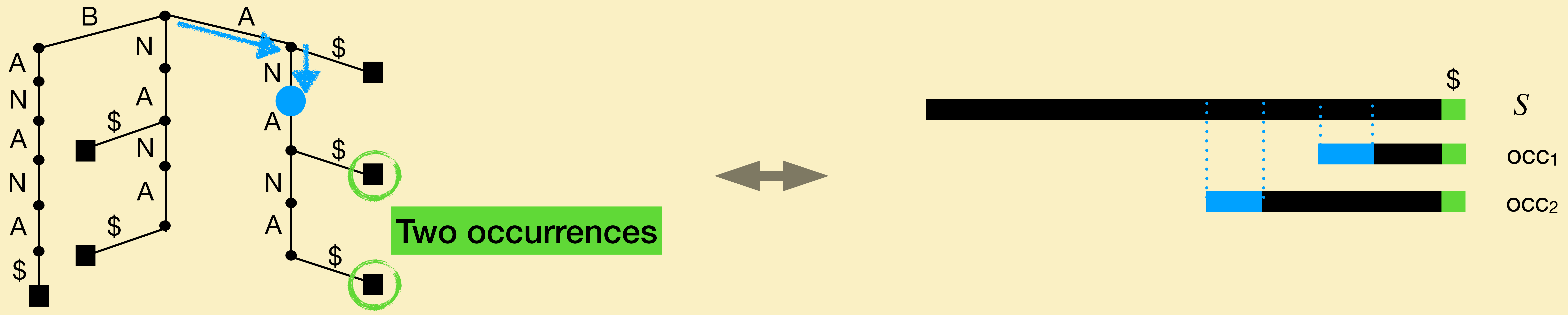
$\Longrightarrow \mathcal{O}(|P| \cdot \mathscr{C}(\exists?\text{child}))$ time

Eg. Is "AN" a substring of "BANANA"? what about "NAB"?



Pattern matching in suffix tries (Count)

Eg. How many times "AN" appears as a substring of "BANANA"?



Algo.

Algorithm 6: Count on suffix trie

Input: The suffix trie ST_S of S, a pattern P

Output: The number of occurences of P in S

- $node \leftarrow \mathrm{root}(ST_S)$
- **for** $i \in [0..|P|)$ **do**
  - **if** $\mathrm{child}(node, P_{[i]})$ doesn't exists **then**
    - **return** False
  - $node \leftarrow \mathrm{child}(node, P_{[i]})$
- **return** CountCoveredLeaves$(node)$

$\Longrightarrow \mathcal{O}((|P| + |S|^2) \cdot \mathscr{C}(\text{child}))$ **time**

Improvement.

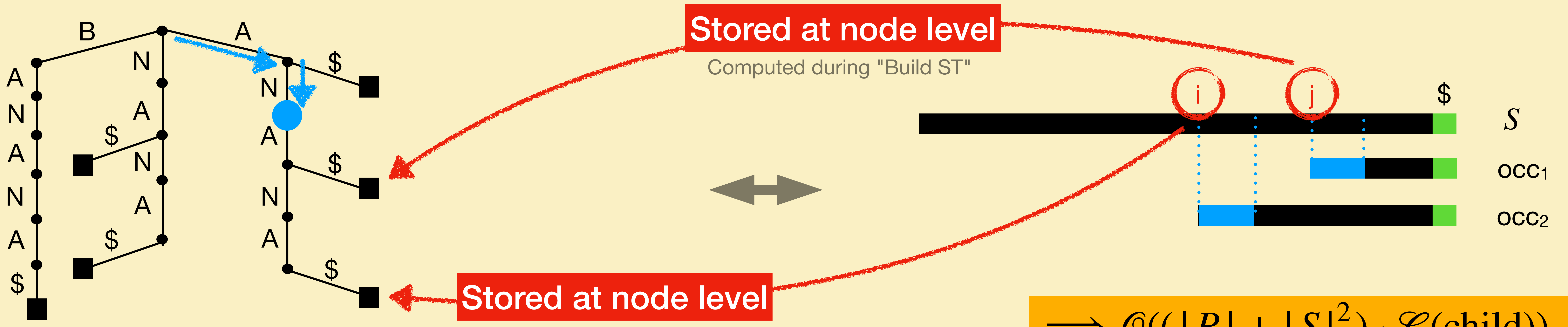
$\Longrightarrow \mathcal{O}((|P|) \cdot \mathscr{C}(\text{child}))$ **time**

Preprocess "countCoveredLeaves" in $\mathcal{O}(|S|^2 \cdot \mathscr{C}(\text{child}))$ time

Store the results at the level of tree nodes

Pattern matching in suffix tries (LocateAll)

Eg. Where are the starting positions of the "AN" pattern in "BANANA"?



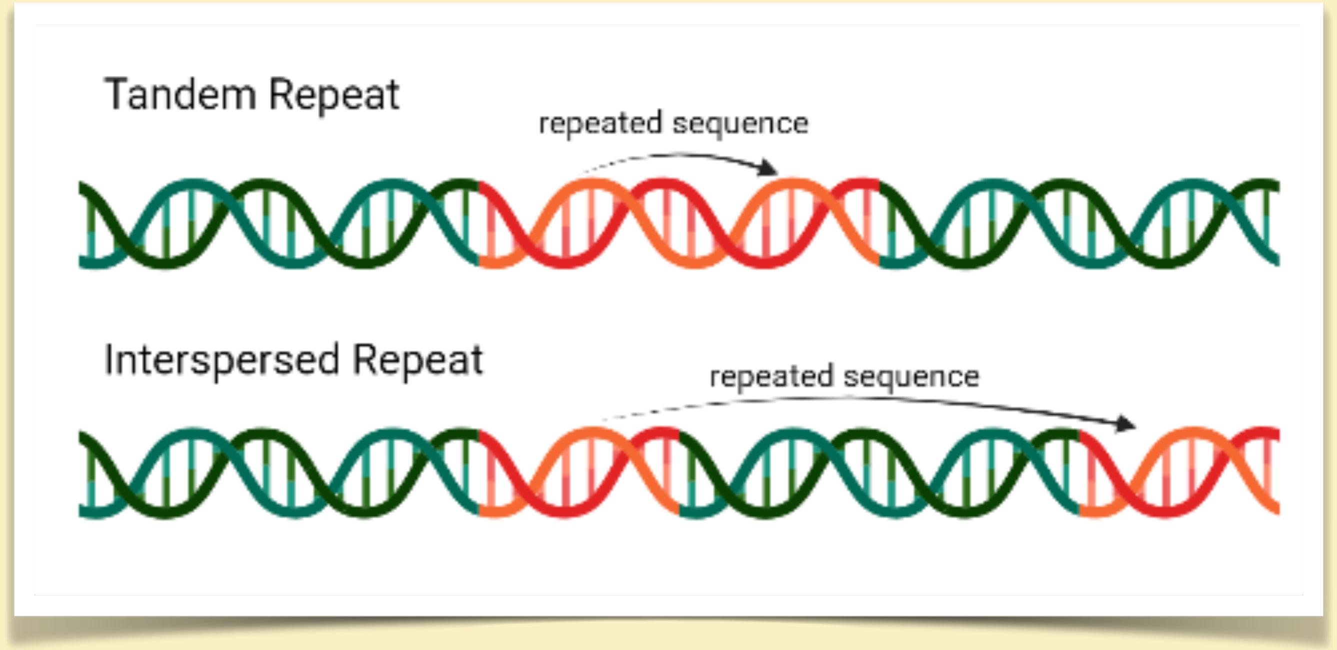
$\Longrightarrow \mathcal{O}((|P| + |S|^2) \cdot \mathscr{C}(\text{child}))$ time

Other method (same time complexity). Starting position = end position - depthNode

Beyond pattern matching

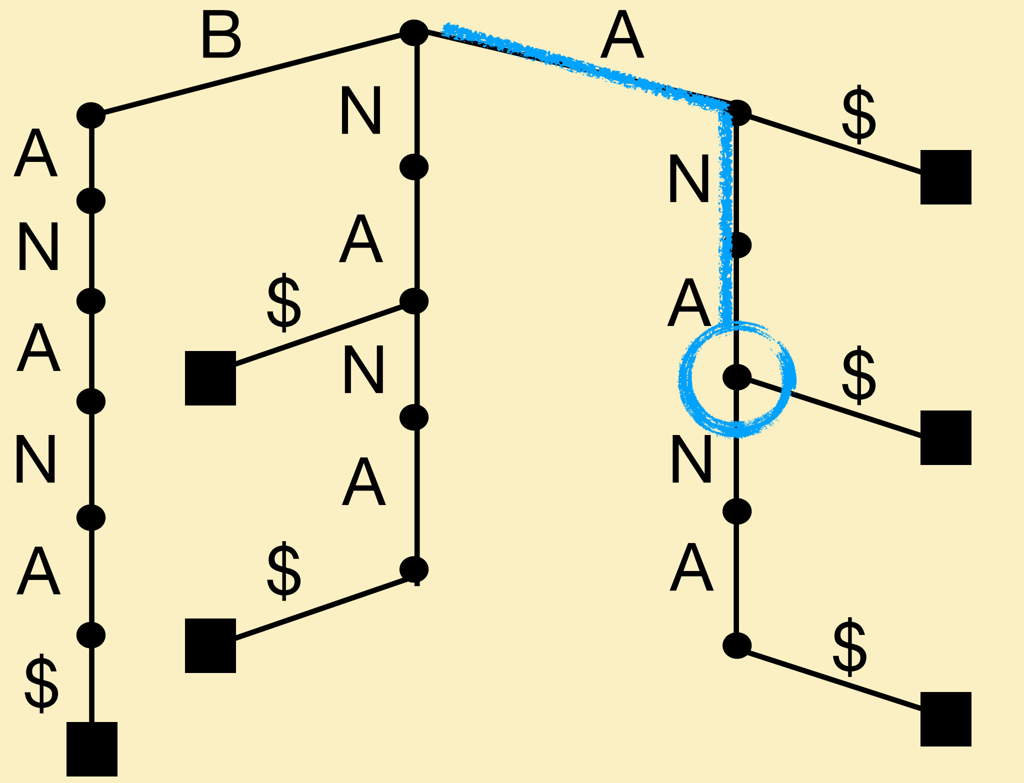
A few examples.

Longest repeating factor

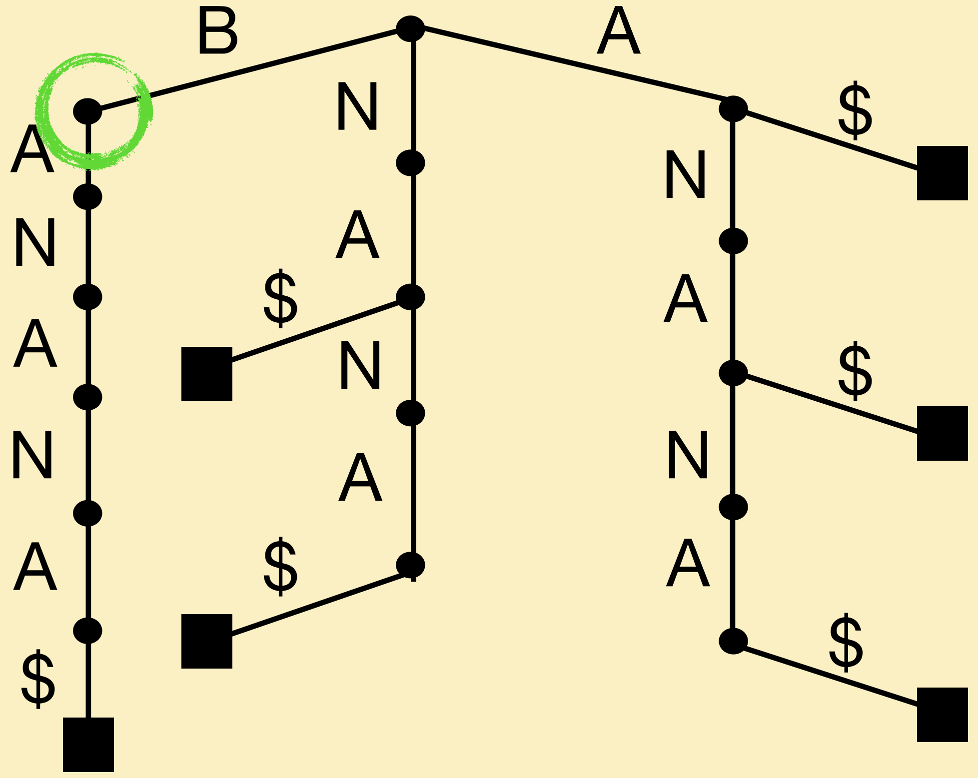


➔ Hotspots for recombination

➔ Transposable elements



Deepest node that covers multiple leaves



Highest node that covers a single leaf

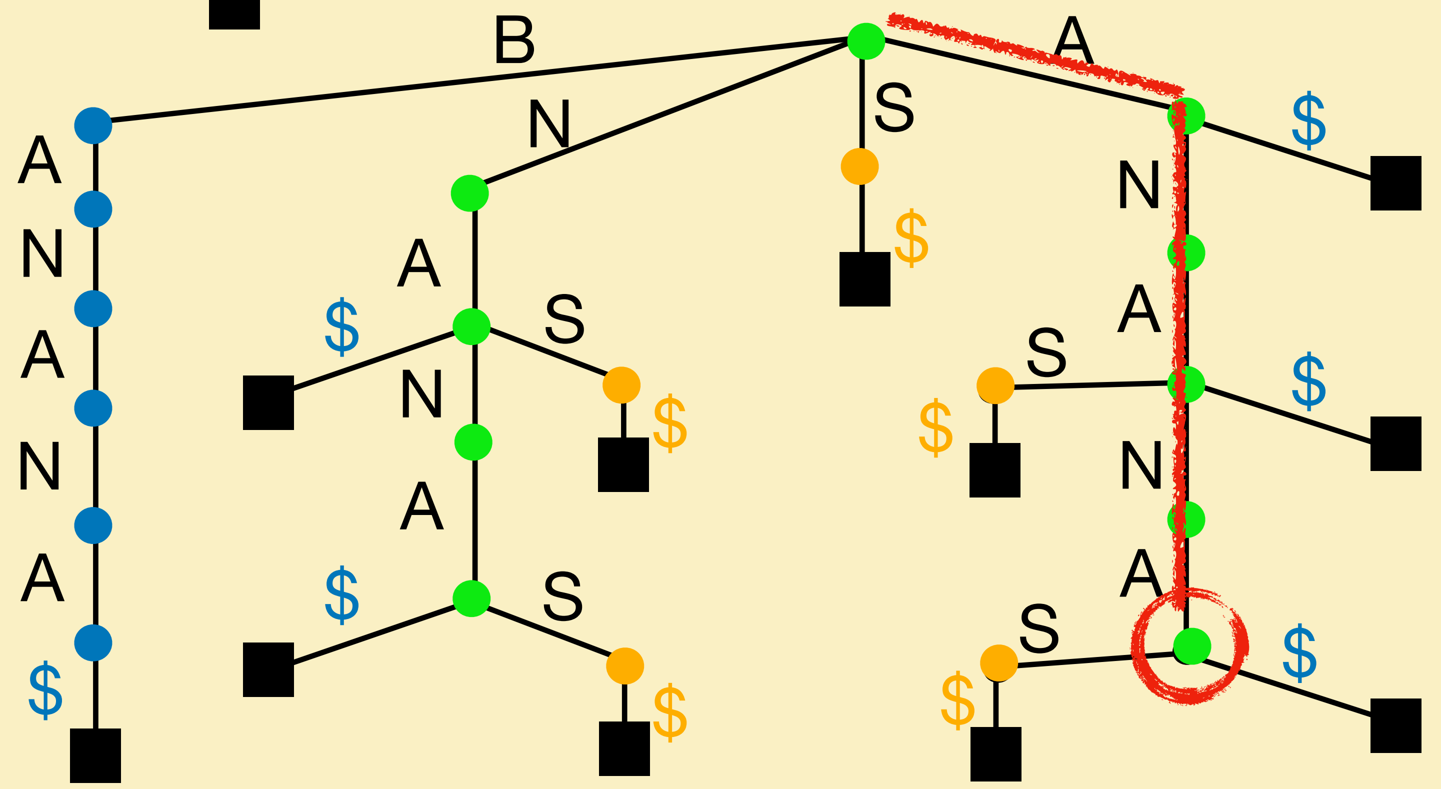
Shortest substring occurring only once

➔ Find good PCR primers

Longest common subsequence

➔ Local alignment

eg. ST("BANANA") ∪ ST("ANANAS")



Deepest "green" node

Limits

Children storage.

- **Array**: test existence in $\mathcal{O}(1)$ time, takes $\mathcal{O}(|\Sigma|)$ space
- **List**: test existence in $\mathcal{O}(|children|)$ time, takes $\mathcal{O}(|children|)$ space
- **Dictionary**: test existence in $\mathcal{O}(1)$ time, takes $\mathcal{O}(|children|)$ space, but no ordering on children
- **Skip list**: test existence in $\mathcal{O}(\log|children|)$ time, takes $\mathcal{O}(|children|)$ space

➡ In practice, varies depending on the depth of the node

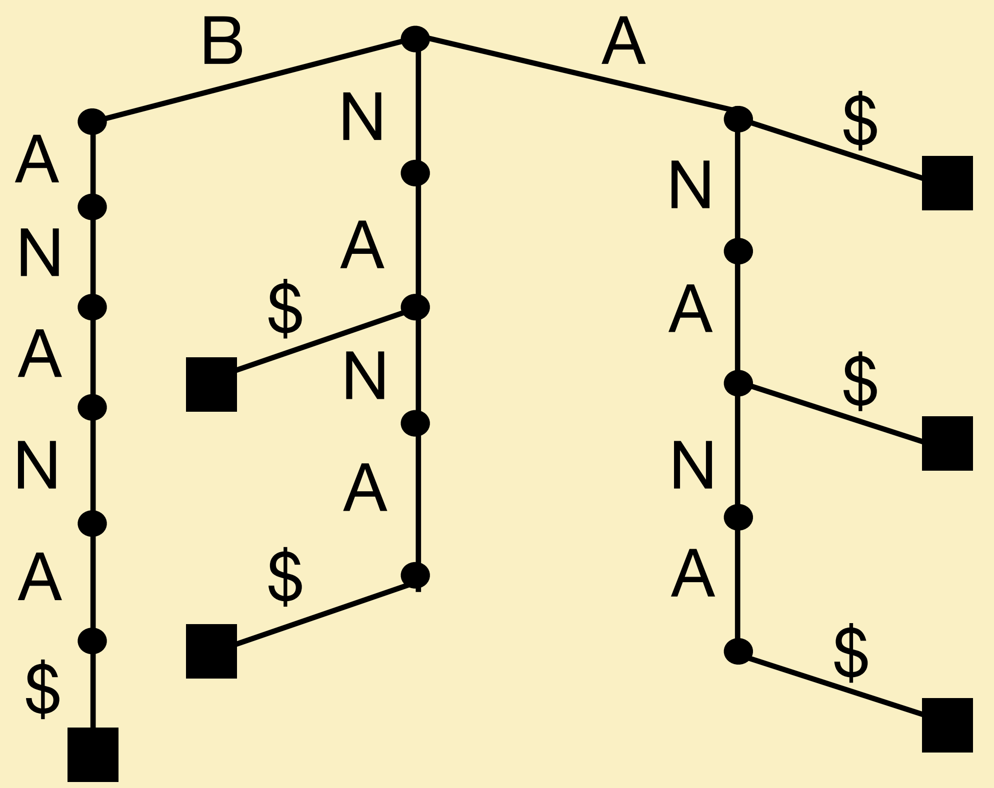
Size. It takes $\mathcal{O}(|S|^2)$ space, hence so does the DFS steps...

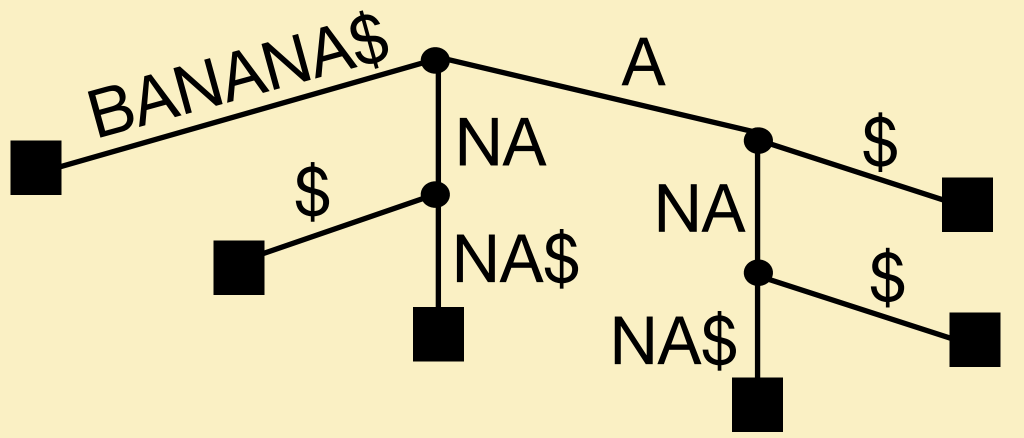
➡ A 10M bp genomes would require ~300 TB to store (nodes + labels + links)

Suffix trees

Compacting the suffix tree

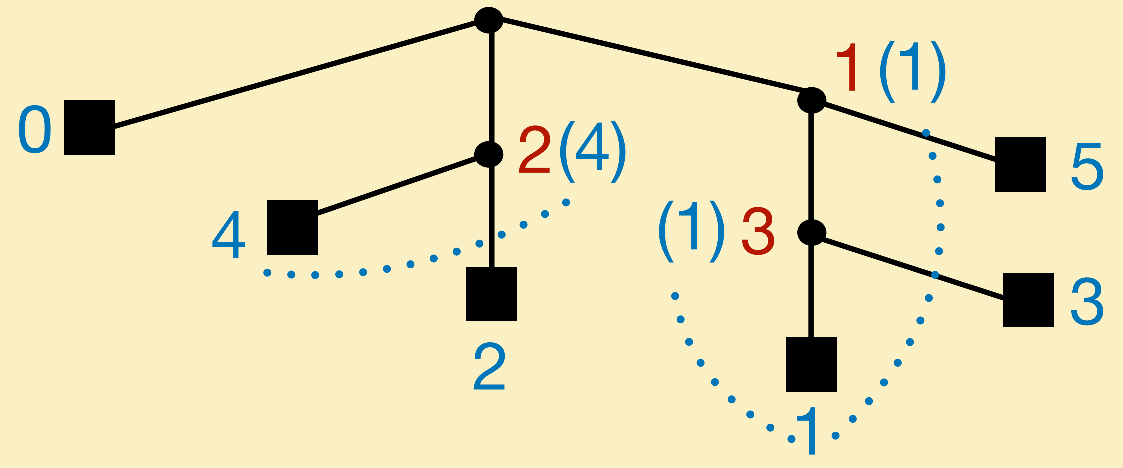
Merging non-branching paths.





0 1 2 3 4 5 6
B A N A N A $

starting position of the suffix
(starting position of a covered suffix)
depth in the suffix trie



At most $2|S|-1$ nodes (all internals are branching)

Construction from Suffix Trie in $\mathcal{O}(|S^2|)$ with a DFS

Problem. Storing labels is still $\mathcal{O}(|S^2|)$

Solution. Recompute them on the fly

$$\mathscr{L}(u \to v) = S[i\,..)_{[depth(u)..depth(v))}$$

with i the starting position of any covered leaves of v

Construction from Suffix Trie in $\mathcal{O}(|S^2|)$ with a DFS

Definition. The suffix tree of a string S is the compacted version of the suffix trie of S.

$\implies \mathcal{O}(|S| \cdot \mathscr{C}(\text{children}))$ **space**

Typically, $|\Sigma|$

Crucial observation. The DFS is now $\mathcal{O}(|S|)$ and, even better, $\mathcal{O}(\#leaves)$

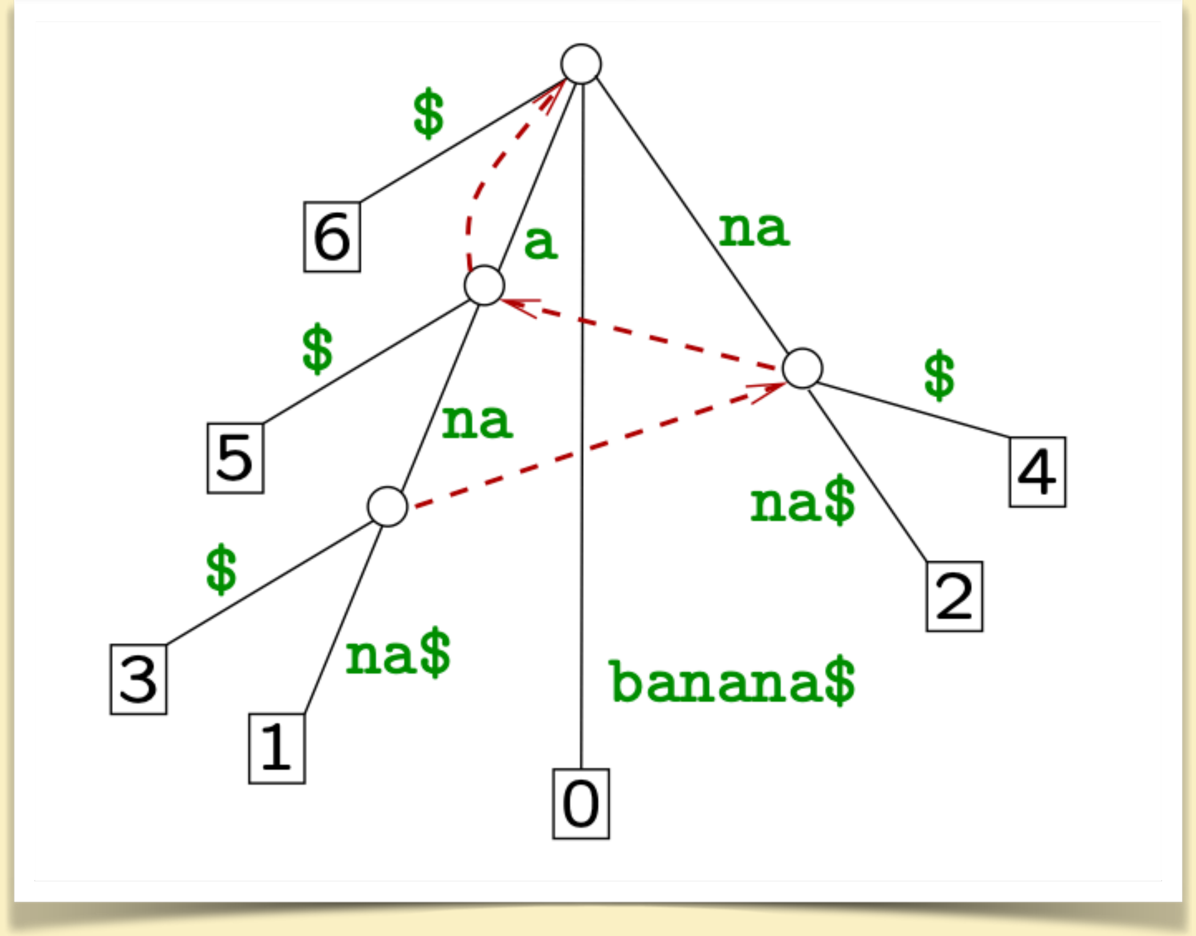
output size!

A linear time construction (flavors of)

Leverage the fact that we are inserting suffixes within the compacted trie

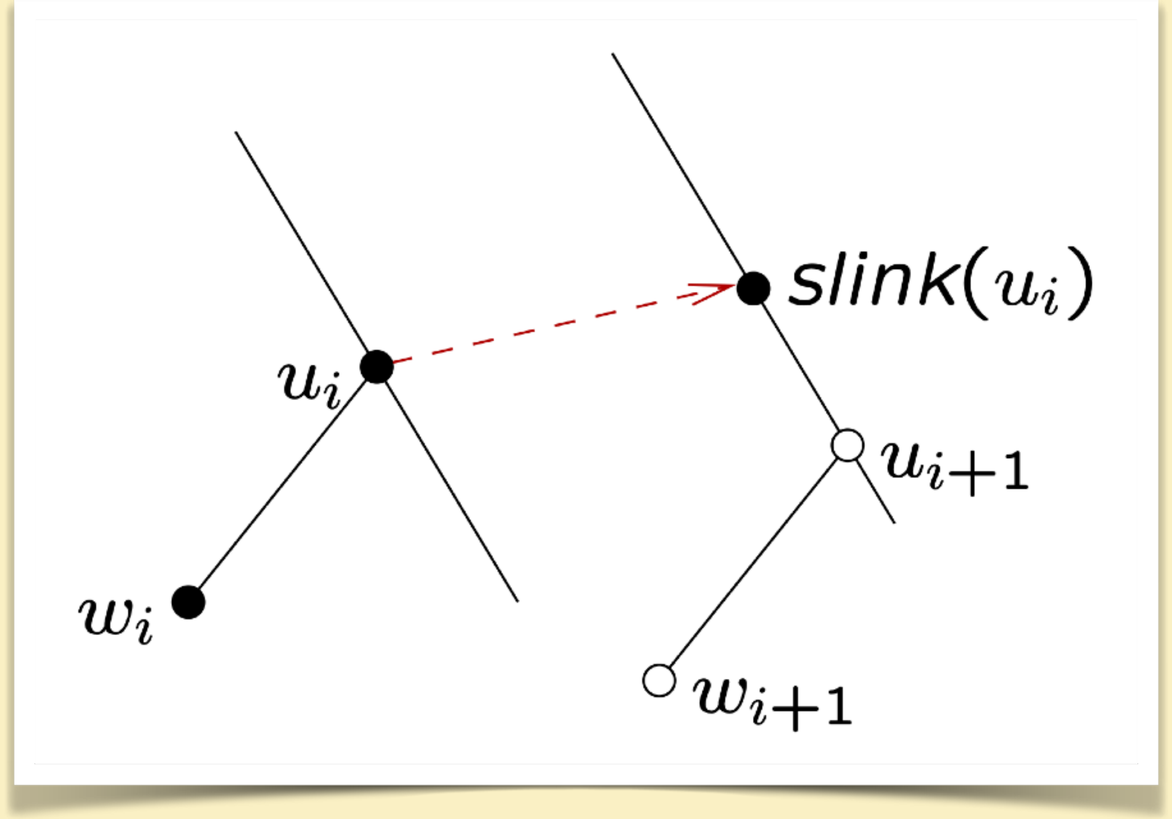
Definition (suffix link). The suffix link $u \to v$ is such that S_v is the longest proper suffix of S_u.

$S_u = S[i\,..\,j] \implies S_v = [i+1..j]$



Proposition. If the leaf that correspond to the suffix $S[i\,..\,]$ is attached to an internal node u_i, then $\mathrm{slink}(u_i)$ is a prefix of S[i+1..]. Hence, the insertion can start from this point (instead of starting from the root).

Rq. If suffixes are taken in order, the suffix link always links to an existing point in the tree.



Consequence. By showing that slink can be computed while inserting prefixes, and by performing a rigorous complexity analysis, one can show that the suffix tree **can be computed in linear time.**

Pattern matching in the suffix tree (Membership)

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

$node \leftarrow \text{root}(ST_S)$
for $i \in [0..|P|)$ **do**
    if child$(node, P_{[i]})$ doesn't exists **then**
        return False
    $node \leftarrow \text{child}(node, P_{[i]})$
return True

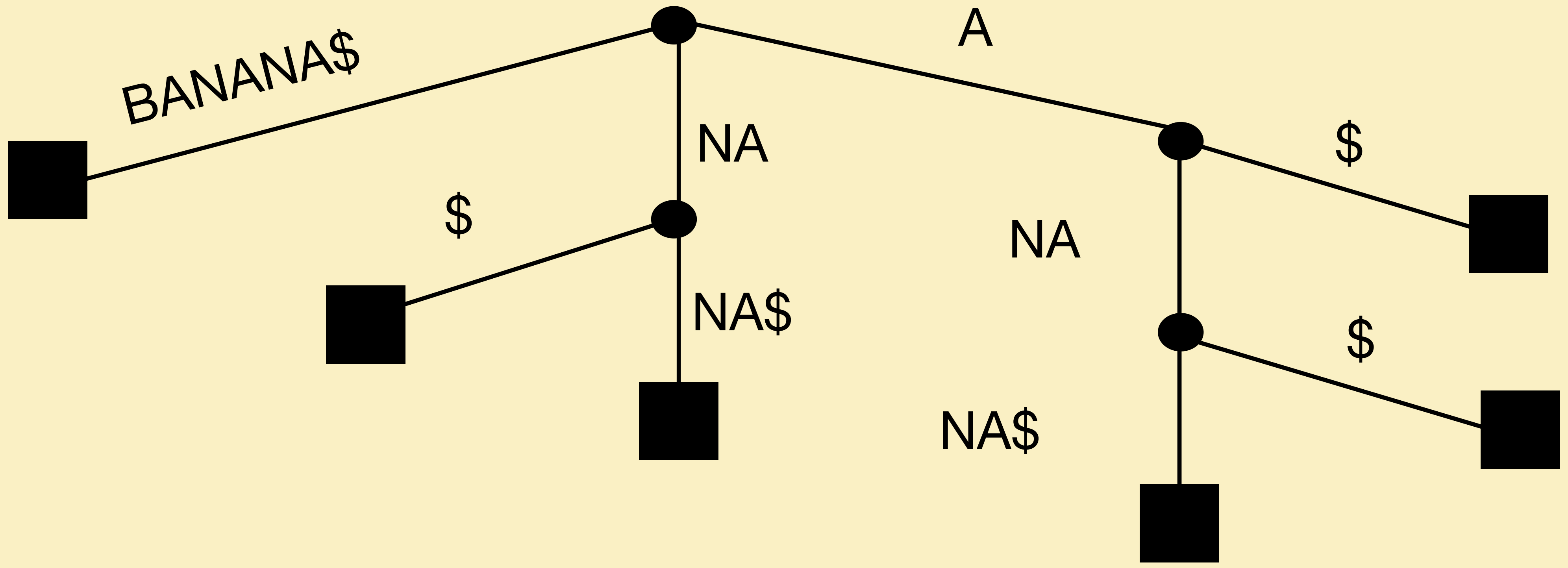
≡

Algorithm 7: Membership on suffix tree

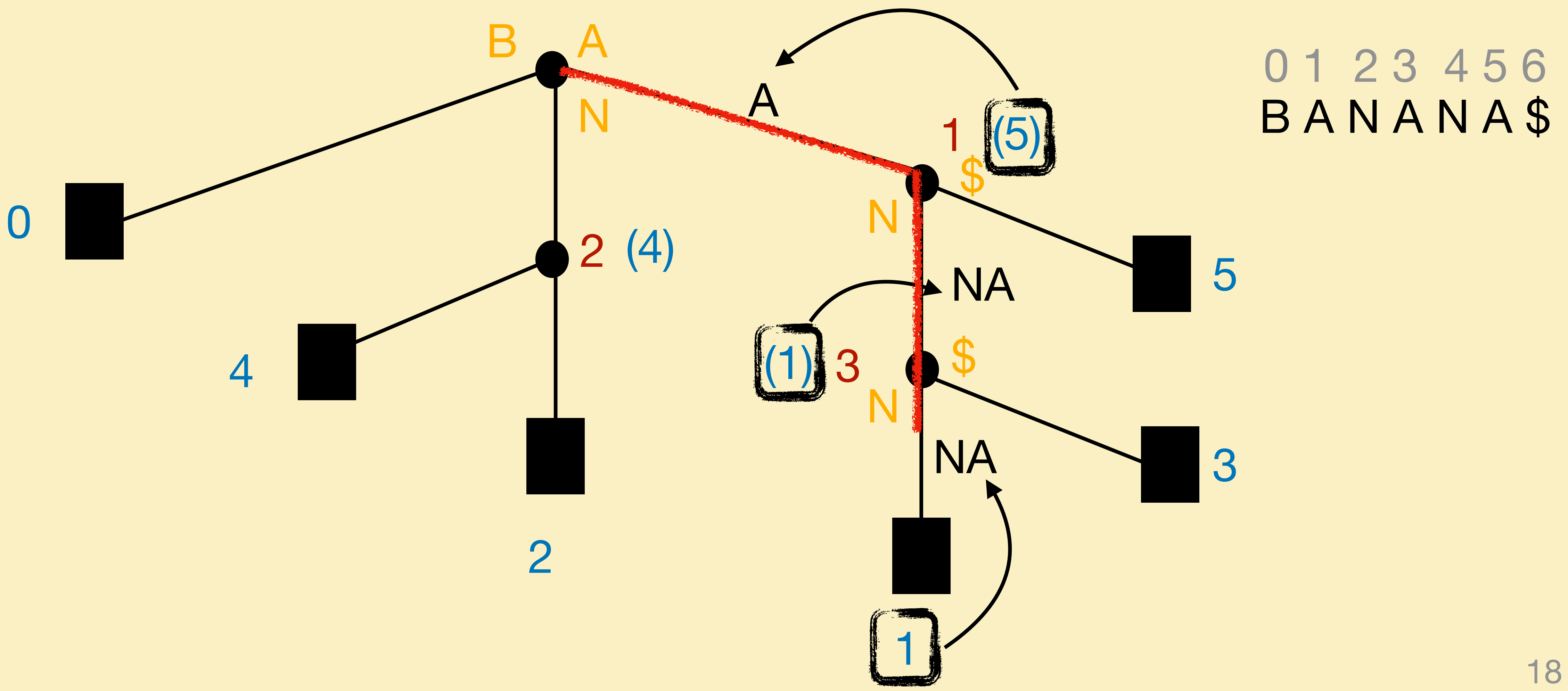
Input: The suffix tree ST_S of S, a pattern P
Output: Whether P is a substring of S, or not

$u \leftarrow \text{root}(ST_S)$
for $i \in [0..|P|)$ **do**
    ▷ *If we are at a branching node, branch as dictated by the prefix (if possible)* ◁
    ▷ *Otherwise, check the characters that are on the branch* ◁
    if $i \geq \text{depth}(node)$ **then**
        if child$(u, P_{[i]})$ doesn't exists **then**
            return False
        $u \leftarrow \text{child}(u, P_{[i]})$
    else
        if $S_{[\text{someCoveredLeaf}(u)+i]} \neq P_{[i]}$ **then**
            return False
return True

Eg. Searching for "ANAN"



≡



Limits

Size.

- Depends on the alphabet... $(\mathcal{O}(|S| \cdot \mathcal{C}(\text{children})))$
- Quite big in reality (many information stored at the level of node)
- So big the linear time construction is not practical: jumping to unrelated part of the tree is not "fact constant-time", because of cache locality

Suffix arrays

Suffix array

Definition (Suffix array).

The suffix array of a string S is the array of the $|S|+1$ starting positions of the lexicographically sorted suffixes of $S\$$, where the termination symbol is smaller than all other letters.

S = BANANA$
0 1 2 3 4 5 6

➔

$ (6)
A$ (5)
ANA$ (3)
ANANA$ (1)
BANANA$ (0)
NA$ (4)
NANA$ (2)

➔ SA_S = [6, 5, 3, 1, 0, 4, 2]

$\Longrightarrow \mathcal{O}(|S|)$ space (no $|\Sigma|$ dependancies)

Construction.

[1] Sorting suffixes. Naively, this takes $\mathcal{O}(|S|^2 \log |S|)$ time.

[2] From the suffix tree. Retrieve the leaves with an (ordered) DFS, in overall (non-efficient) $\mathcal{O}(|S|)$ time.

[3] Direct fast construction. Cf. literature (linear time).

Pattern matching with the suffix array

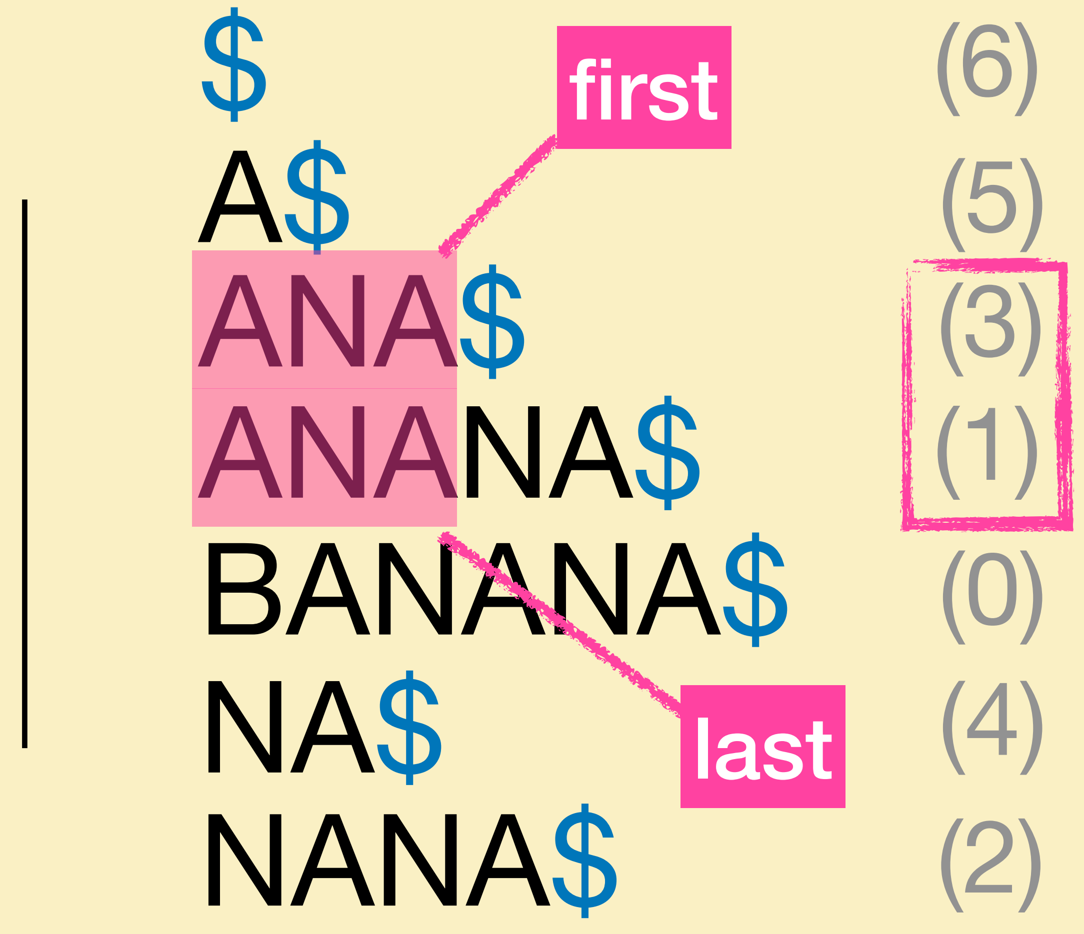
Key idea. If a pattern P is a substring of a string S, its occurrences are contiguous in SA_S.

Algorithm 9: Pattern matching with the suffix array

Input: The suffix array SA_S of a string S, a pattern P

```
first ← BinarySearch(P, SA_S, First)
return True iff first ≠ ⊥                          ▷ Membership
last ← BinarySearch(P, SA_S, Last)
return last − first + 1 if the makes sense, 0 otherwise   ▷ Count
return SA[first..last] if makes sense, 0 otherwise        ▷ Locate
```

$\Longrightarrow \mathcal{O}(|P|\log|S| + |out|)$ time



What do you think of this bound?

Very pessimistic! Way better in practice :)

Eg. On random strings, the expected comparison time (for one step of the search) is $\mathcal{O}(1)$

LCP-fastened comparison

Let i_{min} and i_{max} be the indexes manipulated during the binary search.

Observation 1. The characters comparison between the pattern P and the mid-word $\mathsf{SA}_S[\ \lfloor (i_{min} + i_{max})/2 \rfloor\ ]$ will correspond to equalities on the first $|\operatorname{lcp}(\mathsf{SA}_S[i_{min}], \mathsf{SA}_S[i_{max}])|$ characters.

Observation 2. It holds that

$$\operatorname{lcp}(\mathsf{SA}_S[i_{min}], \mathsf{SA}_S[i_{max}]) = \min \begin{cases} \operatorname{lcp}(\mathsf{SA}_S[i_{min}], \mathsf{SA}_S[i_{mid}]) \\ \operatorname{lcp}(\mathsf{SA}_S[i_{mid}], \mathsf{SA}_S[i_{max}]) \end{cases}$$

Can be computed jointly with word comparison

Algorithm 10: Comparing strings with lcp length

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

```
i ← ℓ + 1
while i ≤ min{|S|, |S'|} do
    if S[i] ≠ S'[i] then
        return (">", i − 1) or ("<", i − 1)
    i ← i + 1
return (">", i − 1), ("<", i − 1), or ("=", i − 1)        ▷ Comparing |S| and |S'|
```

Note. There exists an algorithm that runs in better $\mathcal{O}(|P| + \log|S|)$ time in the worst-case, but it is less practical.

Enhancing the suffix array

Idea. The suffix array is just the leaf order of the suffix tree. To make is useful beyond simple pattern matching, we need other arrays to completely capture the tree topology.

Definition (LCP array).

This is the array defined by $\mathrm{LCP}_S[i] = |\mathrm{lcp}(S_{[\mathrm{SA}_S[i]..)}, S_{[\mathrm{SA}_S[i+1]..)})|$

➡ **Allows for queries such as longest repeated subsequence**

Construction.

[1] Naive. This takes $\mathcal{O}(|S|^2)$ time.

[2] From the suffix tree. Retrieve it with an (ordered) DFS, in overall (non-efficient) $\mathcal{O}(|S|)$ time.

[3] Direct fast construction. Cf. literature (linear time).

And beyond.

Replacing suffix trees with enhanced suffix arrays

Mohamed Ibrahim Abouelhoda [a], Stefan Kurtz [b], Enno Ohlebusch [a,*]

In this article, we will overcome this obstacle. We will show how *every* algorithm that uses a suffix tree as data structure can systematically be replaced with an algorithm that uses an enhanced suffix array and solves the same problem in the same time complexity.

Part PS-B

Preprocessing string - Searching compressed sequences

Motivation

A far-fetched remarkable instance. Pattern matching over homopolymers:

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
```

$\Longrightarrow \mathcal{O}(|P|)$ time and constant space

Intuition. Possible because S can be described with only $\mathcal{O}(1)$ characters!

S is highly **compressible**

Entropy-based compression is not enough.

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

while

$$\mathcal{K}(S^n) \leq$$

Kolmogorov complexity. Size of the smallest program that generates the argument

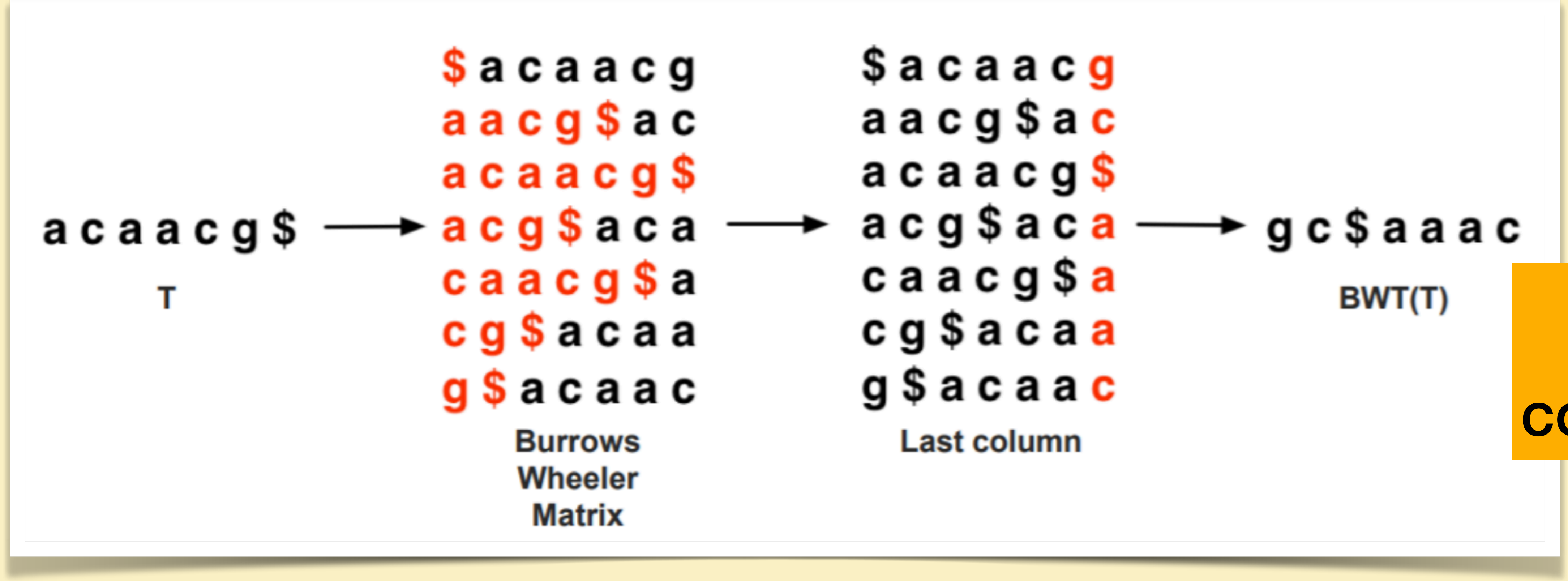
Repetitive strings call for dedicated compression methods

The Burrows-Wheeler transform

The Burrows-Wheeler transform

Definition (Burrows-Wheeler transform).

The BWT of a string is the string corresponding to the last column of the matrix whose rows are the $|S|+1$ rotations of $S\$$ sorted lexicographically, where the termination symbol $\$$ is smaller than any other letter



$\Longrightarrow \mathcal{O}(|S|^2 \log(|S|))$ time construction using $\mathcal{O}(|S|^2)$ space

Room for improvement...

Idea. The order of the sorted rotations of S is the order of the sorted suffixes of S.

Definition (Burrows-Wheeler transform, bis).

The BWT of a string S is the array of length $|S|+1$ given by:

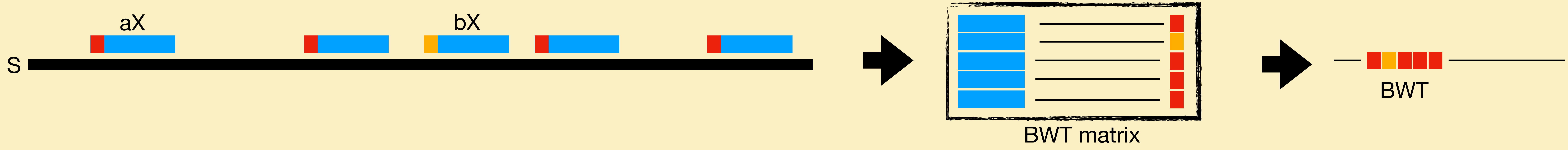
$$\mathrm{BWT}_S[i] = \begin{cases} S[\mathrm{SA}_S[i]-1] & \text{if } \mathrm{SA}_S[i] > 0 \\ \$ & \text{otherwise} \end{cases}$$

$\Longrightarrow \mathcal{O}(|S|)$ space and time construction

Compression capabilities of the BWT

Idea. If a pattern aX is repeated in S, the BWT region that correspond to the X chunk will contains many a's.

$\longrightarrow$ only a few "character changes" in the region



Two compression techniques (that can be combined).

Run-length encoding

AAAAAAAATTTTTTAAAACCCCCCCCCCGGGGGG $\longrightarrow$ A8.T6.A4.C10.G6

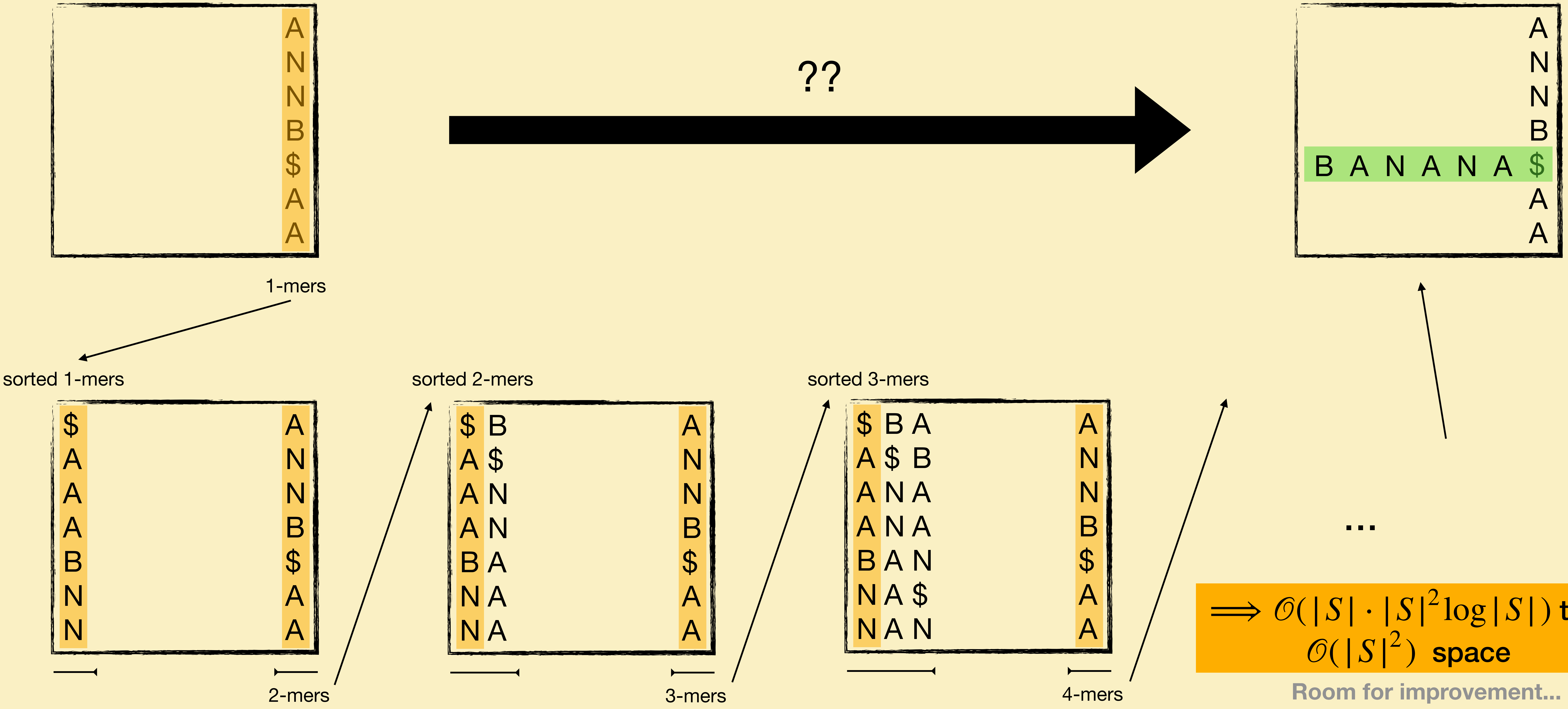
Move-to-front Encode repeated letters with smaller integers

INEFFI**CIE**N**CIE**S: 8.13.4.5.5.8.2.8.4.13.2.8.4.18 $\longrightarrow$ 8.13.6.7.0.3.6.1.3.4.**3.3.3.**18

*Letters are a list: when a letter is seen, bring is to the front

Inverting the BWT

Let S be such that $\text{BWT}(S) = ANNB\$AA$.

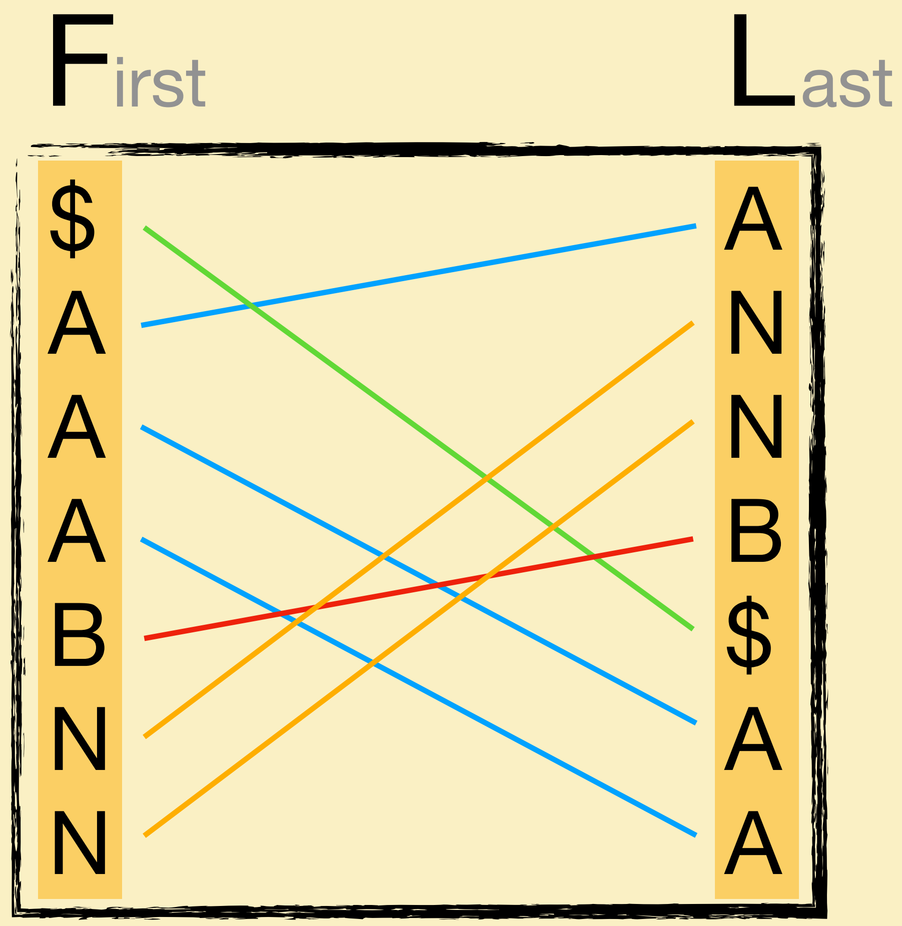


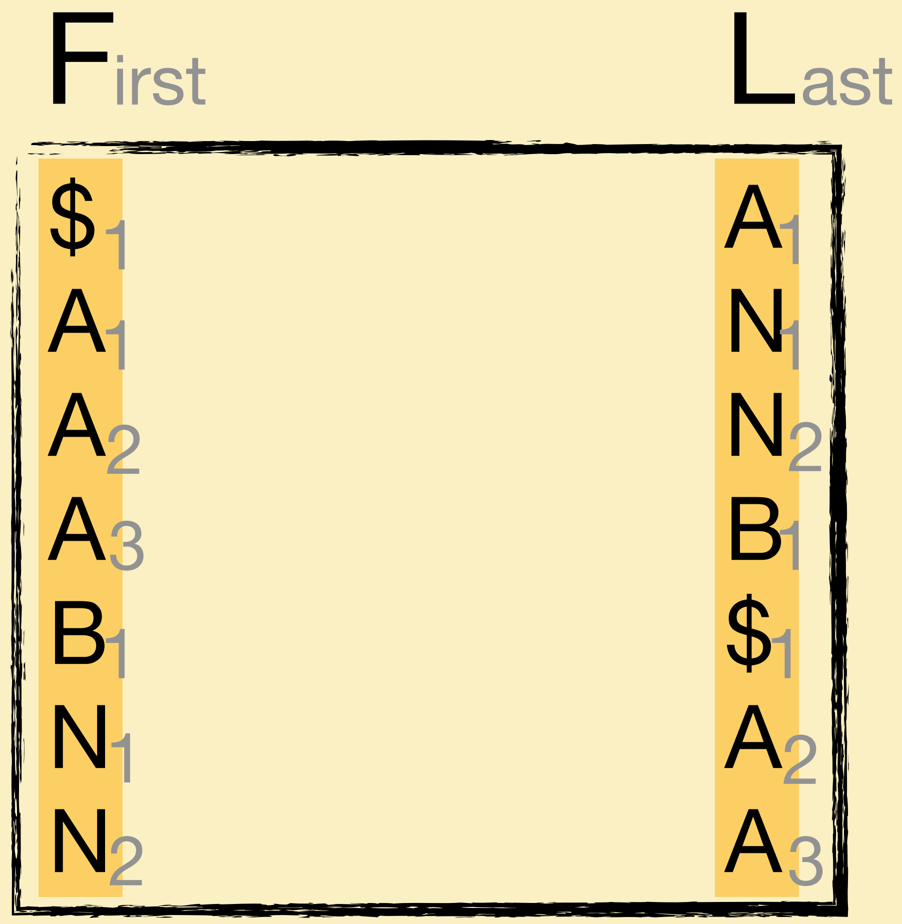
$\Longrightarrow \mathcal{O}(|S| \cdot |S|^2 \log |S|)$ time
$\mathcal{O}(|S|^2)$ space

Room for improvement...

The LF-mapping

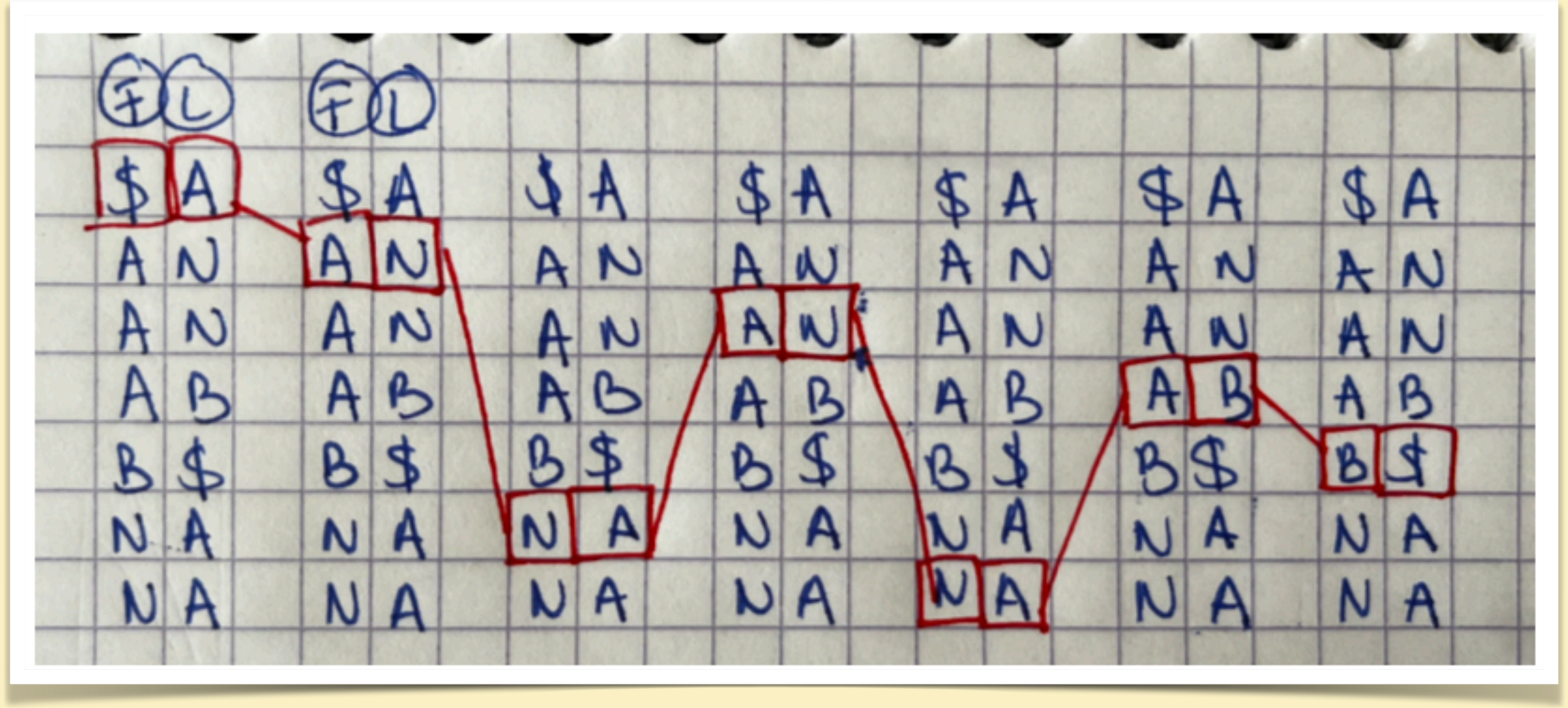
Lemma. For every character a, the i-th occurence of a in L and the i-th occurence of a in F correspond to the same character in S.





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 preceeded with a that preserves the relative order of these suffixes. ∎

Inverting the BWT, faster



$\Longrightarrow$ seems $\mathcal{O}(|S|)$ space!

Algo (informal).

[1] Let S = ''. Each time you see a new character in F, prepend it to S.
[2] Put yourself at F's termination symbol
[3] Repeat |S| times:
  [3.a] Go to the corresponding cell in L
  [3.b] Use the LF mapping to locate the current symbol within F
[4] Return S[:-1]

How to perform 3.b efficiently ?

The FM-index

Rank array. The letter at position i in L is the $R[i]$-th of its kind:

$$R[i] = \#\{j \mid j \in [0..i], \text{BWT}(S)_{[j]} = \text{BWT}(S)_{[i]}\}$$

Cumulative count map. $C[x]$ is the amount of letters of $\text{BWT}(S)$ that are strictly smaller than x:

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

C	F	L	R
0	$	A	1
1	A	N	1
1	A	N	2
1	A	B	1
3	B	$	1
4	N	A	2
4	N	A	3

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 $\text{BWT}_S = L$

$S \leftarrow$ "$", $i_{row} \leftarrow 0$
loop $|S|$ times
    $prec \leftarrow \text{BWT}_S[i_{row}]$
    $S \leftarrow prec + S$
    $i_{row} \leftarrow$ LFMapping(i_{row})
return S

function LFMapping(i)
    $\leftarrow C[\text{BWT}_S[i]] + R[i] - 1$ ▷ *Ranks starts at* 1 *while indexes at* 0 *(hence* -1

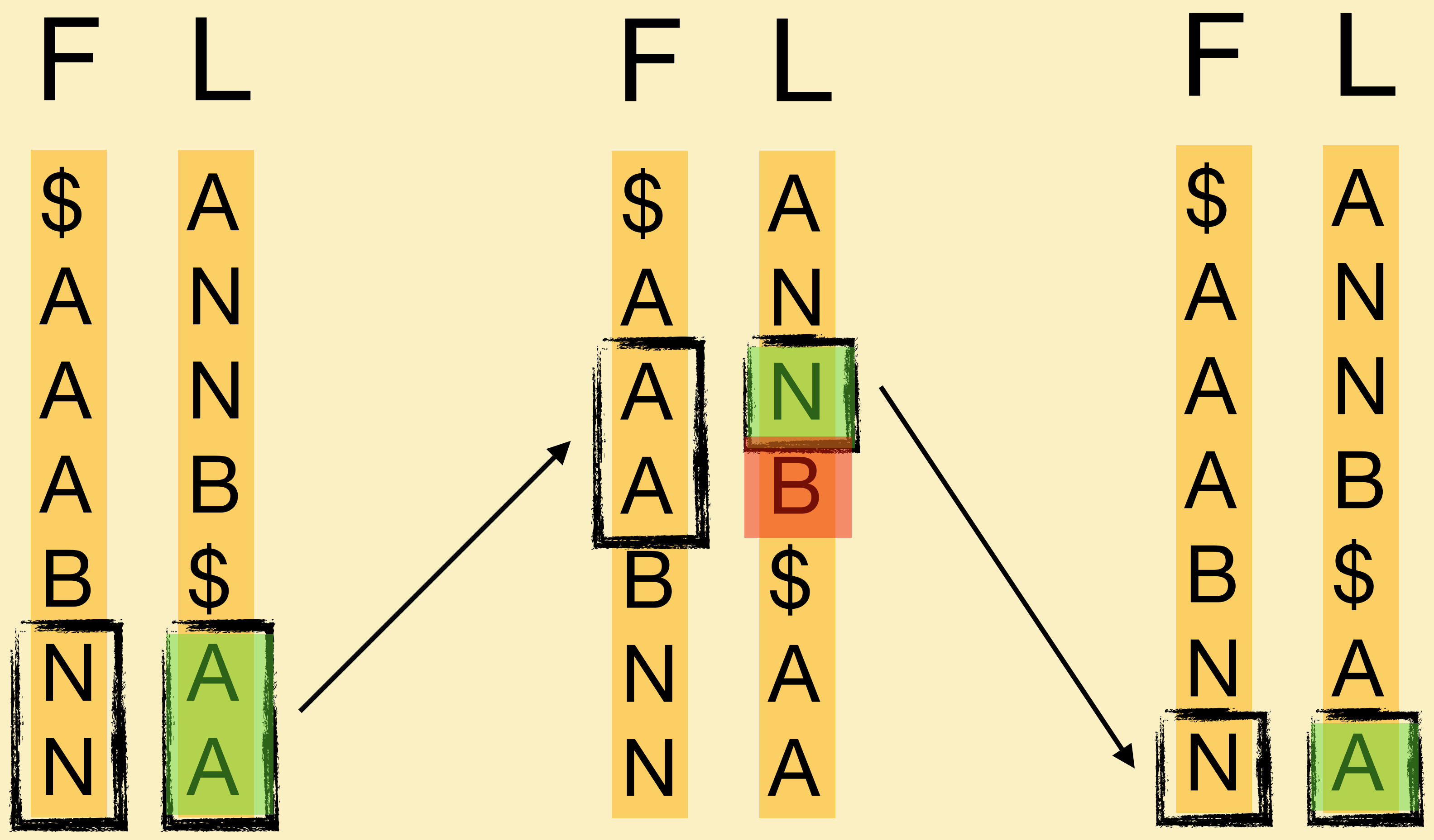
$\Longrightarrow \mathcal{O}(|S|)$ space and time!

Pattern matching from the BWT

Pattern matching with the FM-index

We just did it (somehow). Inverting the BWT is recovering the longest suffix of $S\$$ that ends with $\$$.

Is "ANAN" present in "BANANA"? We look for matching interval, going right-to-left,



Yes, the pattern is present

It appears once

Its location can be recovered with SA

Formally,

Algorithm 14: Pattern matching on the BW transform

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

$(i_{min}, i_{max}) \leftarrow (C[P_{[-1]}], C[\text{nextSmallestLetter}(P_{[-1]})] - 1)$
if $(i_{min}, i_{max}) = \perp$ **then**
  return False (resp. 0) ▷ *Membership (resp. Count)*

for $k \in [0..|P| - 1)$ in reverse order **do**
  $oldRange \leftarrow [i_{min}..i_{max}]$
  $i_{min} \leftarrow$ FirstOccWithin($P_{[k]}$, $L[oldRange]$)
  $i_{max} \leftarrow$ LastOccWithin($P_{[k]}$, $L[oldRange]$)
  if $(i_{min}, i_{max}) = \perp$ **then**
    return False (resp. 0) ▷ *Membership (resp. Count)*
  $(i_{min}, i_{max}) \leftarrow$ (LFMapping(i_{min}), LFMapping(i_{min}))
return True (resp. $i_{max} - i_{min} + 1$) ▷ *Membership (resp. Count)*

$\implies \mathcal{O}(|P| \cdot |S|)$ time

Trading space for time

Rank arrayS. The rank array is split by letters:

$$R_x[i] = \#\{j \mid j \in [0..i], \text{BWT}(S)_{[j]} = x\}$$

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.

$(i_{min}, i_{max}) \leftarrow (C[P_{[-1]}], C[\text{nextSmallestLetter}(P_{[-1]})] - 1)$
if $(i_{min}, i_{max}) = \perp$ **then**
  return `False` (resp. 0) ▷ *Membership (resp. Count)*

for $k \in [0..|P| - 1)$ in reverse order **do**
  $i_{min} \leftarrow R_{P_{[k]}}[i_{min} - 1] + 1$
  $i_{max} \leftarrow R_{P_{[k]}}[i_{max}]$
  if $i_{min} > i_{max}$ **then**
    return `False` (resp. 0) ▷ *Membership (resp. Count)*
  $(i_{min}, i_{max}) \leftarrow (\text{LFMAPPING}(i_{min}), \text{LFMAPPING}(i_{min}))$
return `True` (resp. $i_{max} - i_{min} + 1$) ▷ *Membership (resp. Count)*

$\implies \mathcal{O}(|P| + |out|)$ time

Real-life FM-index

I. **LF** needs the **rank of characters** from the BWT
- Storing the rank of each character: too heavy (|B| bytes)
- **Solution:** *rank subsampling* - store checkpoints every N lines
  - — need the rank of all {A, C , G, T} characters
  - — the real rank is made by walking up or down the lines reaching a checkpoint
  - — at most $\lfloor \frac{N}{2} \rfloor$ walks
  - — *Bowtie: N = 448*

II. **SA:** too heavy (|B| bytes)
- **Solution:** *SA subsampling* – (*every 32 positions in Bowtie*)
  - — for each line k in [i, j]:
    - if line k marked: position = SA[k]
    - else reversed walk until a marked position is found

FM index Bowtie memory balance

- **BWT:** $\frac{|B|}{4}$ bytes (2 bits per character, no need to have $ in some case)
- **rank:** $\frac{|B|}{448} \cdot 4$ bytes (4 letter's rank, following lexical order)
- **SA:** $\frac{|B|}{32}$ bytes

Finally: ~0.29 · $|B|$ bytes = 1.16 times the single storage of B (with 2 bits per character)

0.87 GB for the 3 billion characters human genome

*Pierre Peterlongo's slides

Homeworks

Burrows-Wheeler transform

[1] Implement a working RLE (+MTF?) Burrows-Wheeler transform.

[2] Try to compress a random string, and your favorite genome.

[3] Implement a FM-index class, able to answer membership/count/locate queries.

Pour ce TP, nous utiliserons une implémentation du tableau des suffixes disponible ici : http://bioinformatique.rennes.inria.fr/tools_karkkainen_sanders.py adapté de http://code.google.com/p/pysuffix/.

Voici un exemple d'utilisation :

```
from tools_karkkainen_sanders import simple_kark_sort
s = 'GGCGGCACCGC$'
sa = simple_kark_sort(s)
print('i\tSA\tf\tSuffixes')
for i in range(len(s)): print(f'{i}\t{sa[i]}\t{s[sa[i]]}\t{s[sa[i]:]}')
```

Produira la sortie suivante :

```
i   SA  f   Suffixes
0   11  $   $
1   6   A   ACCGC$
2   10  C   C$
3   5   C   CACCGC$
4   7   C   CCGC$
5   8   C   CGC$
6   2   C   CGGCACCGC$
7   9   G   GC$
...
```

BOX

Pattern matching - again

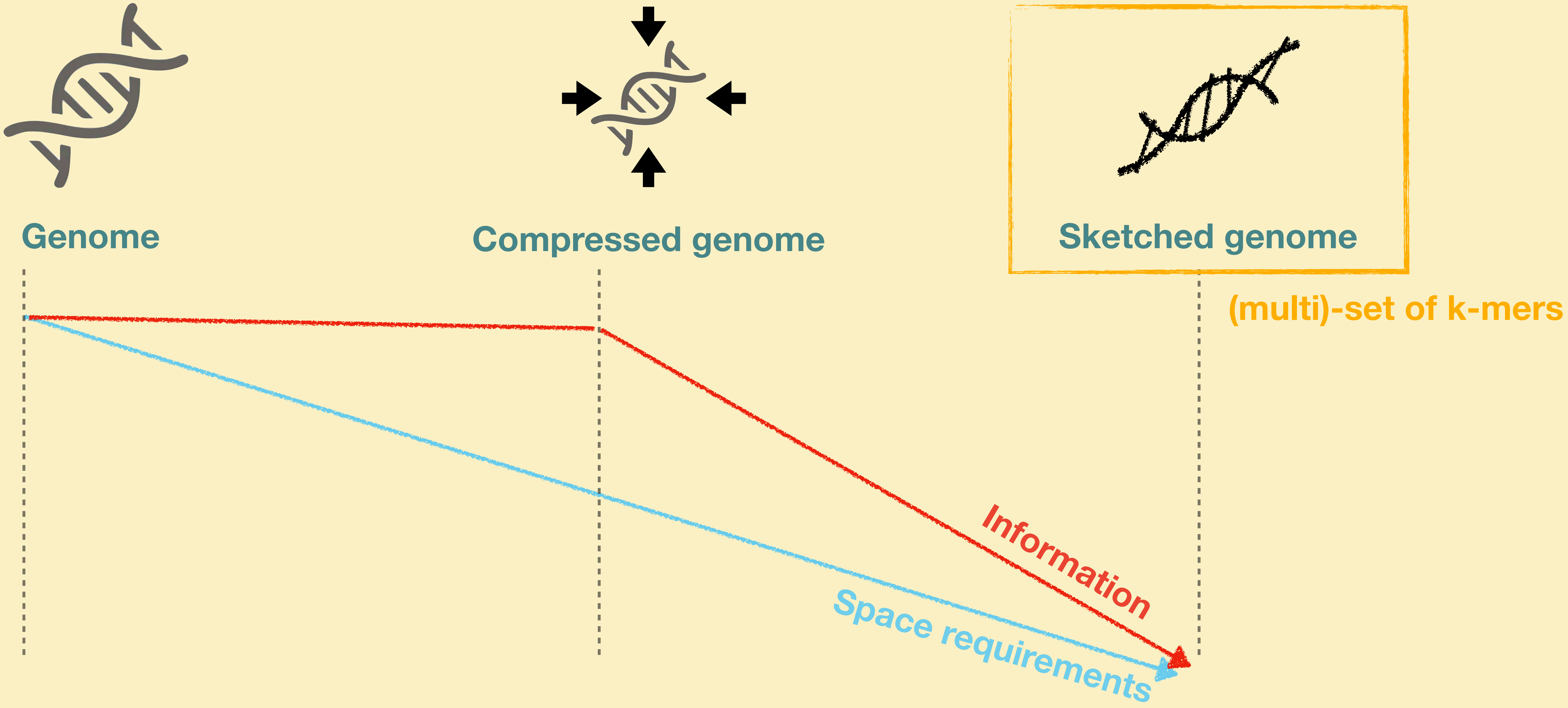
Léo Ackermann

Part PS-C

Preprocessing string - Searching sketched sequences

Outline

Various genome representations.



Query (membership/count/locate) complexity around $\mathcal{O}(P + |output|)$

Call for sketches

Modern laptop storage. Around 1 Terabyte

Modern genome storage demand.

- Bacterial genome: ~1MB
- Human genome: ~1GB
- BLAST nr/nt: ~1TB
- Tara Oceans DNA/RNA: ~60TB
- NCBI SRA (reads): ~32PB, and still x2 every two years

Today: efficient representation of k-mer sets for pattern matching

But many other bioinfo-relevant sketches exist!

Inverted index

Inverted index

Definition (Inverted index).

A k-mer index of a string S is a data structure that represents occurrence lists O_K for each k-mer present in S: $i \in O_k \Leftrightarrow S_{[i..i+k)} = K$.

$S =$ ATTCGATTCCGAT

ATT	$\longrightarrow$ [0, 5]
CCG	$\longrightarrow$ [8]
CGA	$\longrightarrow$ [3, 9]
GAT	$\longrightarrow$ [4, 10]
TCC	$\longrightarrow$ [7]
TCG	$\longrightarrow$ [2]
TTC	$\longrightarrow$ [1, 6]

k-mer index of S

How to query general queries?

If $|q| < k$. Easy, just find first/last match as in SA.

If $|q| > k$. Combinatorial explosion or seed-extend

query — set of k-mers

Query. Takes $\mathcal{O}(k \cdot \log|\mathcal{K}| + |output|)$ time

Space. Takes $2k$ bits per k-mer, and $|S|$ bytes for positions

Can we do better?

Hash functions

Intuition. There are way less than 4^{31} distinct 31-mers in a typical genomic string S. Storing them explicitly using $2k$ bits per k-mer is ineffective.

Would be nice to have $f : \mathcal{K} \subsetneq [0..4^{31}] \to [0..\mathcal{O}(|\mathcal{K}|)]$.

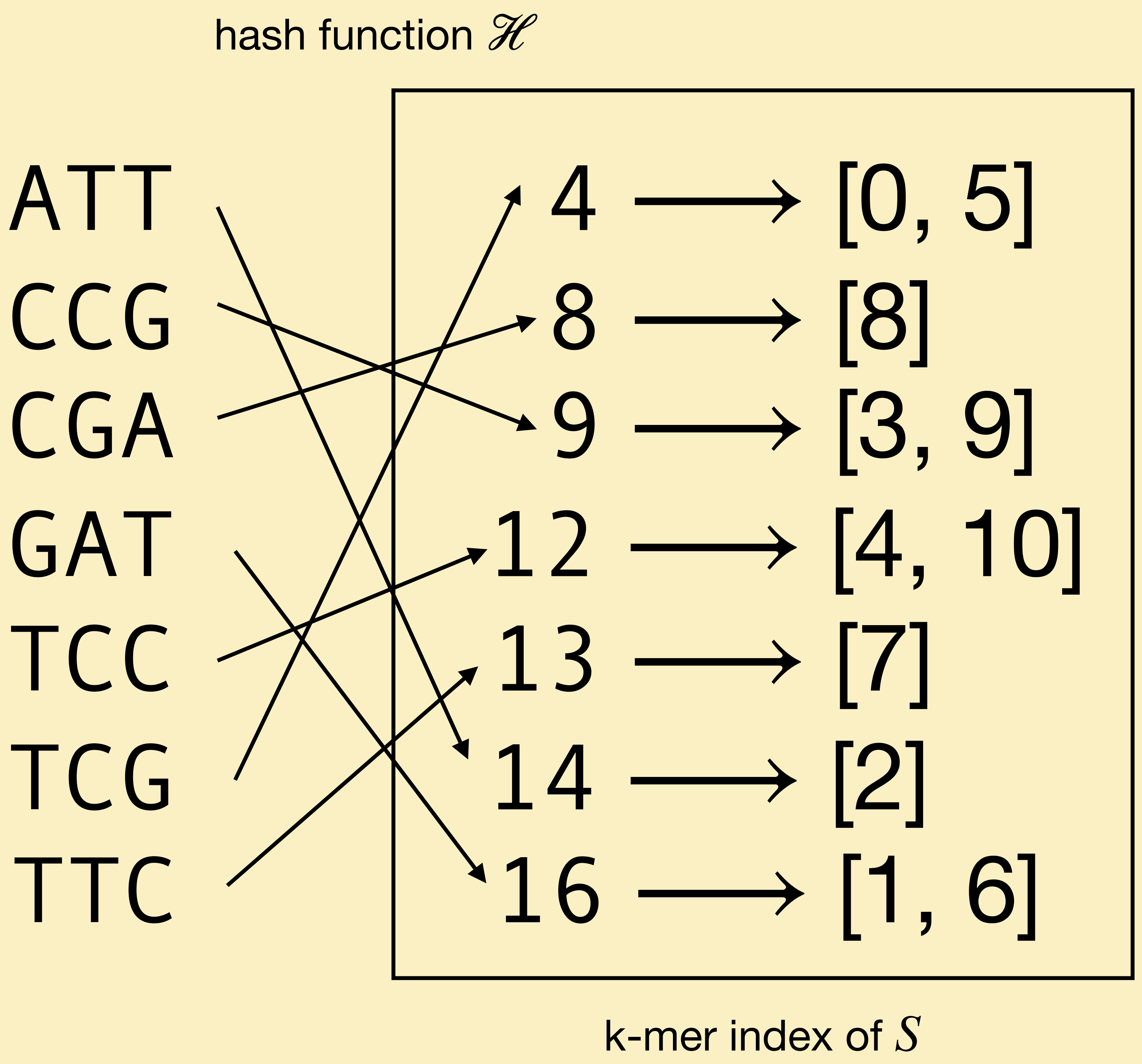
For f to be a good hash function.

- Efficiently computable (and storable)
- Bijective (or so: cases where $x \neq y$ but $f(x) = f(y)$ are called collisions)

Naturally, tradeoff between efficiency and size of range for a given probability of collision!

Inverted index - hash based

$S = \text{ATTCGATTCCGAT}$



Algo. Query q of length k

1. Let $h = \mathscr{H}(q)$
2. Look for h in the k-mer index

Note: the hash function breaks the contiguity we relied on for smaller queries

Filters

Filters

Definition (Filter).

A filter **approximately** represents a set. It must support entry insertions and queries. Additionally, it might also support entry deletion, or filters union/intersection operations.

False positive are allowed

as they typically just waste some work

Eg. filtering low-abundant k-mers

False negative are (typically) not allowed

Today.

- Bloom filters
- Cuckoo filters
- Quotient filters

Bloom filter // Definition

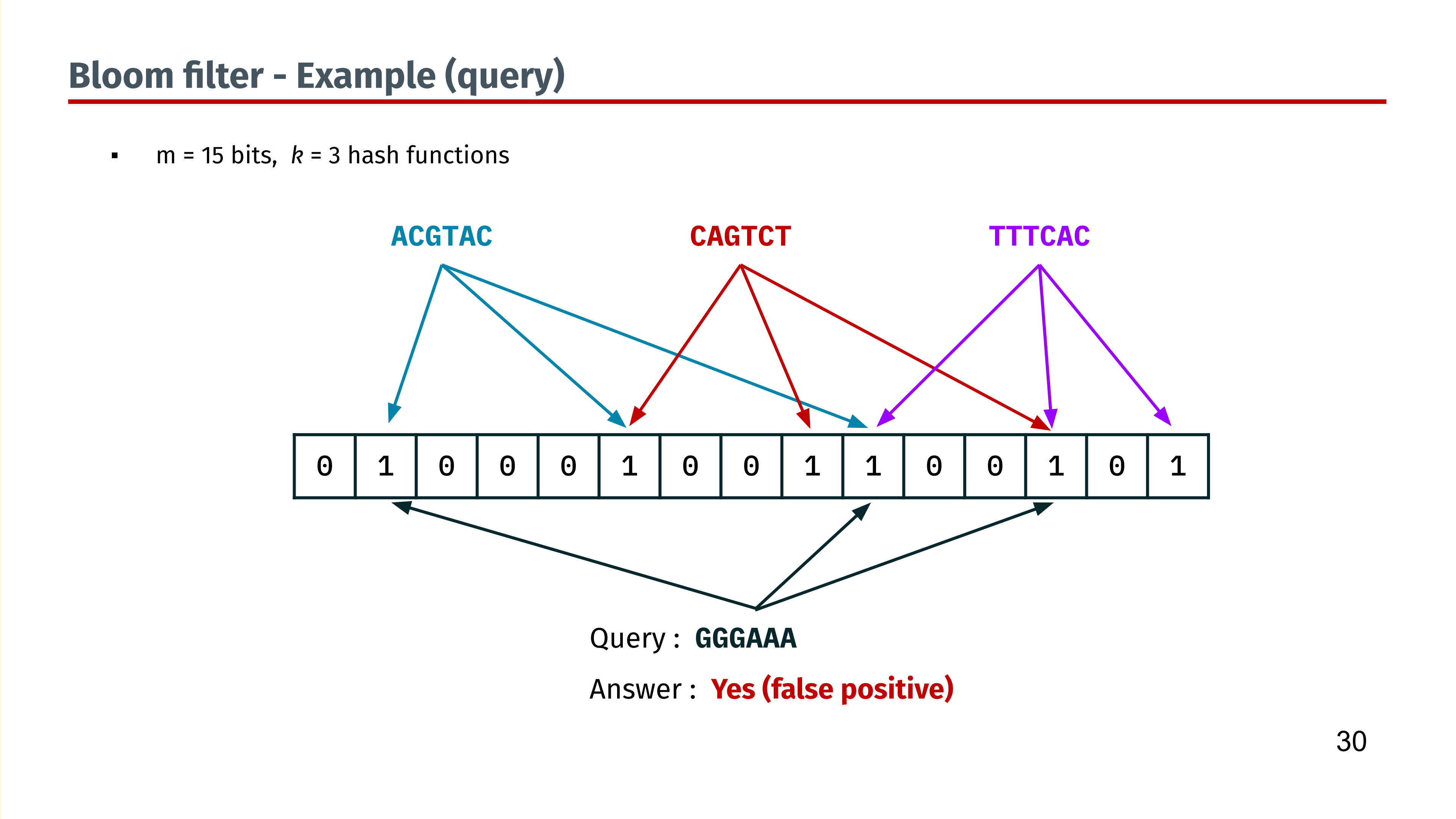
Definition (Bloom Filter).

The Bloom filter only supports probabilistic (only FP can happen) membership queries. It represents a set of n elements using a bit array B of size m, and k distinct hash functions.

Insert(B, e). For all $i \in [1..k]$, let $B[h_i(e)] = 1$

Query(B, e). Check that it holds for all $i \in [1..k]$ that $B[h_i(e)] = 1$

Example (m=15, k=3).



Bloom filter // Choosing parameters

Probability of FP. Let assume that hash functions are random and independent.

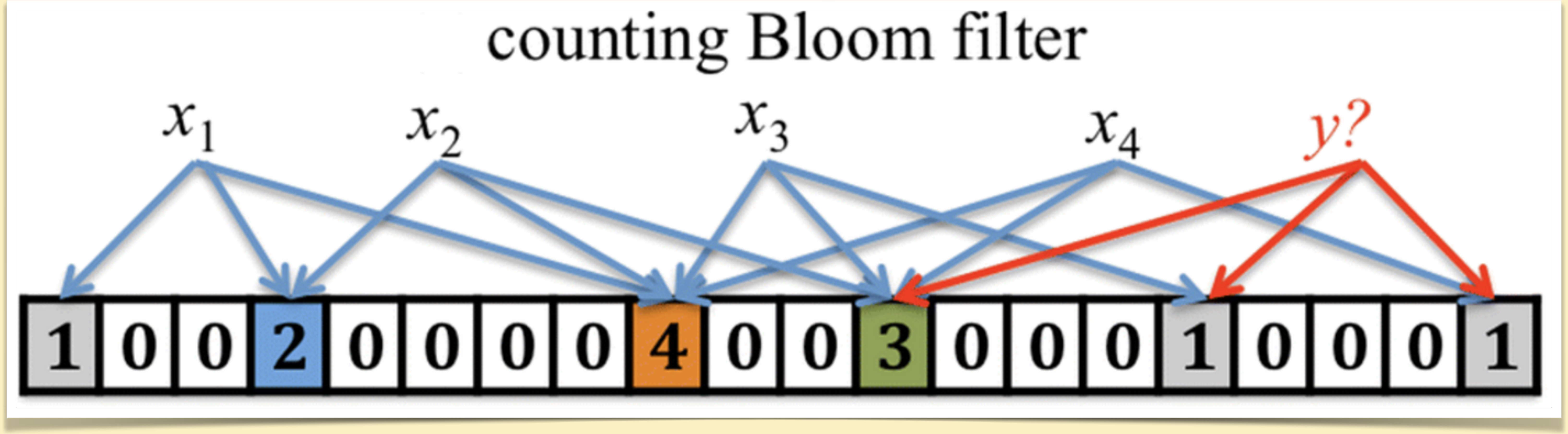
- $P(B[i]$ is not set to 1 during insertion$) = (1 - 1/m)^k = ((1 - 1/m)^m)^{k/m} \underset{m \to \infty}{\approx} e^{-k/m}$
- $P(B[i]$ is still 0 after n insertions$) \underset{m \to \infty}{\approx} (e^{-k/m})^n$
- $P(B[i]$ is 1 after n insertions$) \underset{m \to \infty}{\approx} 1 - e^{-kn/m}$
- $P(e$ is $FP) = P(\forall i, B[\mathscr{H}_i(e)] = 1) \underset{m \to \infty}{\approx} (1 - e^{-kn/m})^k$

Choice of parameters. For fixed values of n and ε (FP probability), one can derive the optimal values for the scheme:

$$m = -\frac{n \ln \varepsilon}{(\ln 2)^2} \quad \text{and} \quad k = m/n \cdot \ln 2$$

Bloom filter // Counting variant

Idea. Store x bits integers instead of bits in B. Increment counters when inserting.



Two flavors.

Insert_1(B, e). For all $i \in [1..k]$, let $B[h_i(e)] = (B[h_i(e)] + 1)\,\%\,x$

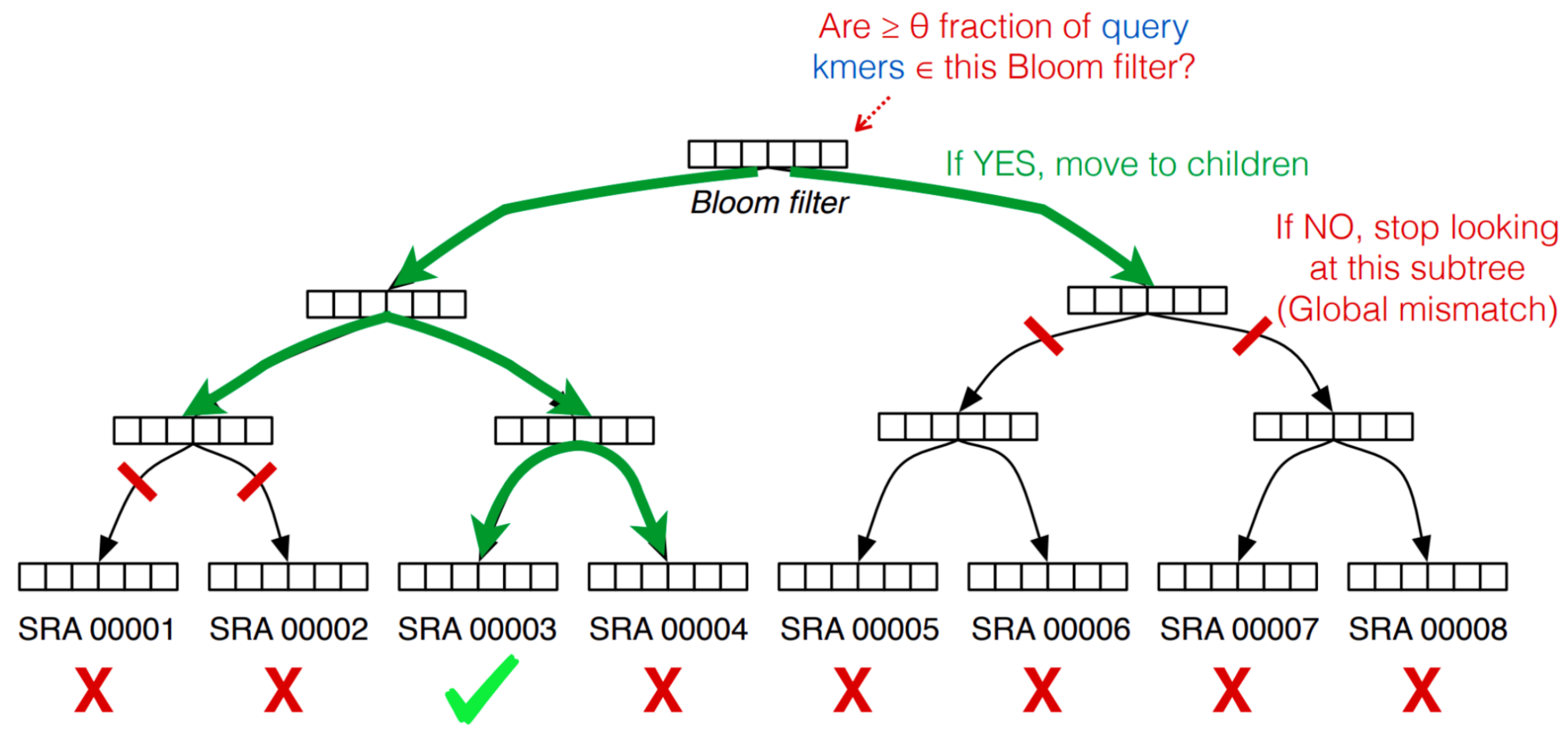
Insert_2(B, e). For all $i \in [1..k]$ that minimize $B[h_i(e)]$, let $B[h_i(e)] = (B[h_i(e)] + 1)\,\%\,x$

Query(B, e). Return $\min_i B[h_i(e)]$

[exo] Compare these variants

Bloom filter // Hierarchical variant (union)

Idea. Propagate Bloom filters bottom-up to quickly identify documents of interest



Credit: B. Solomon

Quotient filter // Motivation

Idea. Bloom filters uses multiple hash functions to prevent collisions, but...

Lemma (birthday paradox). The expected number of samples to take from [0..n] before observing a collision is $\sqrt{\pi/2 \cdot n} = \mathcal{O}(\sqrt{n})$.

⬇

=> the array is far from being full when this problematic arises

can we exploit empty spaces to prevent collisions?

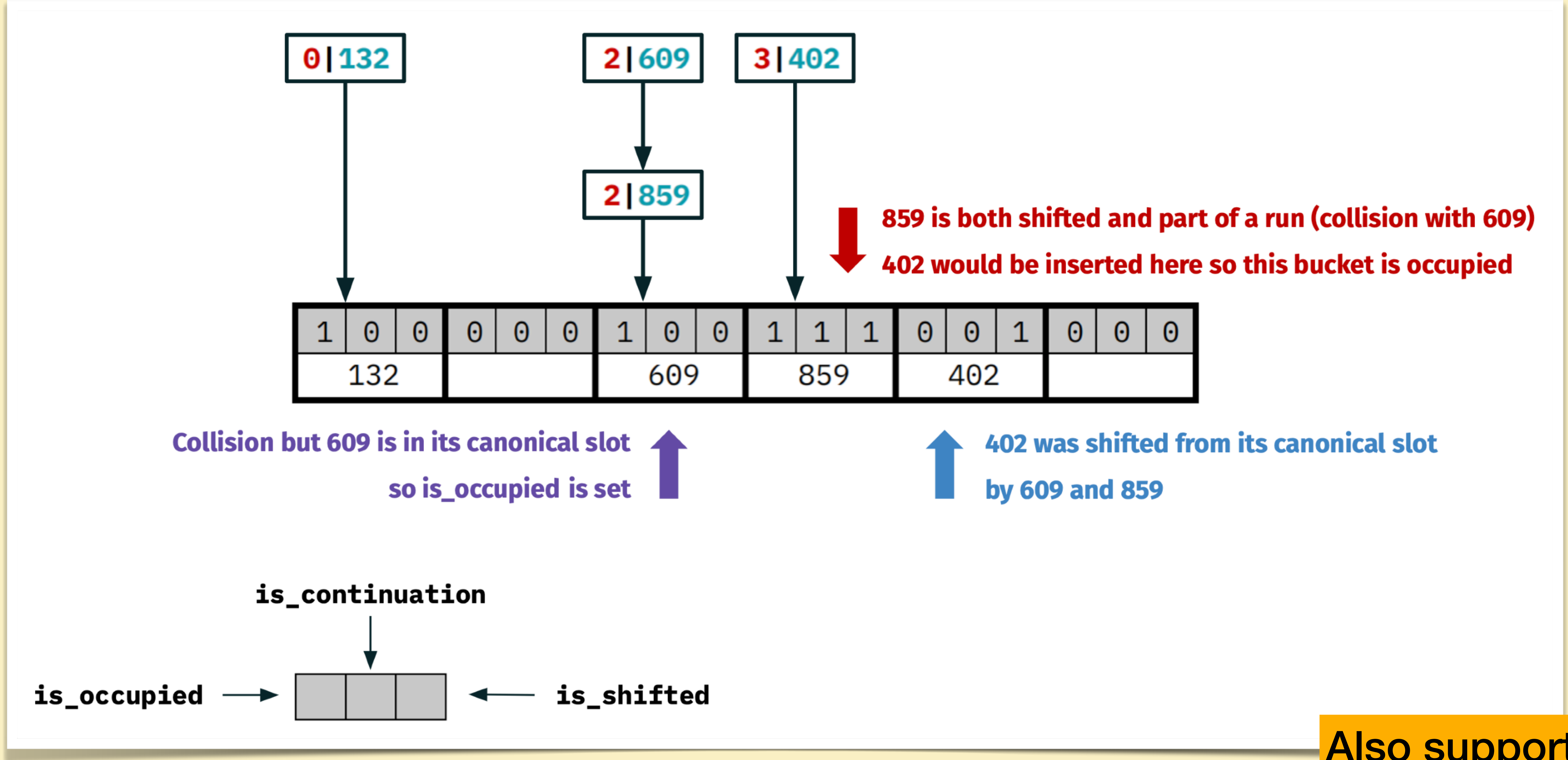
Quotient filter // Definition

H = 0101110100011000110000111 0011101

quotient = where to insert in Q

remainder = what to insert in Q

Idea. Use neighboring cells instead of a mv hash function



Also support deletions!

Cuckoo filter // Definition

Definition (Cuckoo filter).

The Cuckoo filter supports probabilistic (only FP can happen) membership queries. It represents a set of n elements using an array C of size m, each cell made of f bits. It requires three hash functions: h and mv ranging in [0..m), and fgp ranging in [0..f).

~quotient ~remainder $f \in \Omega(\log n)$

Insert(B, e).

1. Try to put fgp(e) within $C[h(e)]$
2. If the cell wasn't empty, put $fgp(e)$ in $C[h(e) \oplus mv(fgp(e))]$.
3. If a value y was there, move it to $C[h(e) + mv(fgp(e)) + mv(y)]$, and so on.

B is almost filled when this procedure fails => space gain

Query(B, e). Check whether fgp(e) indeed lives within $C[h(e)]$ or $C[h(e) \oplus mv(fgp(e))]$

You can delete such an entry!

Going further: https://www.cs.cmu.edu/~binfan/papers/conext14_cuckoofilter.pdf

Homeworks

Bloom filter

[1] Implement a working Bloom filter.

To implement a Bloom Filter you will need to compute a certain number of hash functions. In order to do that, you can use the Python library `mmh3` (you can install the package using conda) The next few lines illustrate a usage example:

```
import mmh3

nb_hashes = 7
size_max = 100000000
item = "ACGGACGACGACT"
for seed in range(nb_hashes):
    key = mmh3.hash(item, seed, signed=False) % size_max
    print(f"Seed {seed} is {key}")
```

[2] Implement the counting variant(s). Recompute kmer histograms from last sessions using this new data structure.