Computing the optimal BWT of very large string collections

Davide Cenzato[1], Veronica Guerrini[2], Zsuzsanna Lipták[3], Giovanna Rosone[2]

[1]Ca' Foscari University of Venice, Department of Environmental Sciences, Informatics and Statistics

[2]University of Pisa, Department of Computer Science

[3]University of Verona, Department of Computer Science

DCC 2023, March 22nd, 2023 - Snowbird, Utah, United States

The Burrows-Wheeler-Transform (BWT)

- **sorting** the rotations of the input text

text: BANANA

↓

rotations of the text

ABANAN
ANABAN
ANANAB
BANANA
NANABA
NABANA

→

BWT

NNBAAA

String collections availability

The focus has shifted from individual sequences to collections of strings

- strings collections are abundant in bioinformatics
- BWT plays a **central role** in processing such large and repetitive datasets
- pattern matching while keeping data compressed (**run-length encoding**)

The Burrows-Wheeler-Transform for string collections

- basis of several **compressed data structures** for strings
- originally defined for **single sequences**
- several tools in literature computing **different BWT variants**

text		BWT
ATATGTGAACGATCAGGA	→	GGACAGTTATGCATAAGA
string collection		
{ATATG,TGA,ACG,ATCA,GGA}	→	?

The effect on the r parameter

There are **different methods** to compute the BWT of string collections

- systematically analyzed the different methods [Cenzato and Lipták, CPM 2022]

These differences extend to the **number of runs** of the BWT ($\boldsymbol{r}$)

- $r(\mathrm{w})$ = number of equal-letter runs of BWT(w)
- depend on the **input order** of the sequences

The **most used** BWT variant is the "multidollar-BWT"

The multidollar BWT

$$\mathrm{mdolBWT}(\mathcal{M}) = \mathrm{BWT}(T_1\$_1 T_2\$_2 \ldots T_k\$_k)$$

ATATG $\$_1$TGA $\$_2$ACG $\$_3$ATCA $\$_4$GGA $\$_5$ →

$\$_1$ … ATATG	G
$\$_2$ … TGA	A
$\$_3$ … ACG	G
$\$_4$ … ATCA	A
$\$_5$ … GGA	A
A$\$_2$ … TG	G
A$\$_4$ … ATC	C
A$\$_5$ … GG	G
ACG$\$_3$ … $\$_2$	\$
ATATG$\$_1$ … $\$_5$	\$
ATCA$\$_4$ … $\$_3$	\$
ATG$\$_1$ … AT	T
CA$\$_4$ … AT	T
CG$\$_3$ … A	A
G$\$_1$ … ATAT	T
G$\$_3$ … AC	C
GA$\$_2$ … T	T
GA$\$_5$ … G	G
GGA$\$_5$ … $\$_4$	\$
TATG$\$_1$ … A	A
TCA$\$_4$ … A	A
TG$\$_1$ … ATA	A
TGA$\$_2$ … $\$_1$	\$

$\$_1$ … ATATG	G
$\$_2$ … TGA	A
$\$_3$ … ACG	G
$\$_4$ … ATCA	A
$\$_5$ … GGA	A

A$\$_2$ … TG	G
A$\$_4$ … ATC	C
A$\$_5$ … GG	G

- **traditionally** used for generating the suffix tree and suffix array of multiple strings
- append (implicitly) **different** dollars
- shared suffixes: **input order**

The effect on the r parameter

- different input orders generate **different outputs**

SAP-intervals

$r = 17$

$ATATG\,\$_1 TGA\,\$_2 ACG\,\$_3 ATCA\,\$_4 GGA\,\$_5$ → G A G A A | G C G | \$ \$ \$ T T A | T C | T G | \$ A A A \$

$r = 15$

$GGA\,\$_1 ATATG\,\$_2 ACG\,\$_3 TGA\,\$_4 ATCA\,\$_5$ → A G G A A | G G C | \$ \$ \$ T T A | T C | G T | \$ A A A \$

$r = 14$

$ATCA\,\$_1 GGA\,\$_2 TGA\,\$_3 ACG\,\$_4 ATATG\,\$_5$ → A A A G G | C G G | \$ \$ \$ T T A | C T | G T | \$ A A A \$

Colexicographic order (aka reverse lexicographic order RLO) [Cox et al., Bioinformatics, 2012]

- Is **colexicographic order** optimal?

Computing the Optimal BWT

Bentley et al. [ESA 2020] introduced a linear-time algorithm for computing the **optimal permutation** of the input collection, which yields the minimum number of runs of the resulting multidollarBWT.

- we refer to this transform as **optimal BWT (optBWT)**

Here, we give the **first tool** **("optimalBWT")** to compute the optBWT:

- use the BWT, the SAP-array, and the algorithm by Bentley et al.
- **reduce the number of runs** up to a 31 factor
- **negligible** time and space overhead

The SAP-array

The SAP-array is a **bitvector** storing the positions of the BWT blocks containing rotations starting with a shared suffix.

- the 0s define the starting points of the SAP-intervals
- the 1s define the extensions of the SAP-intervals

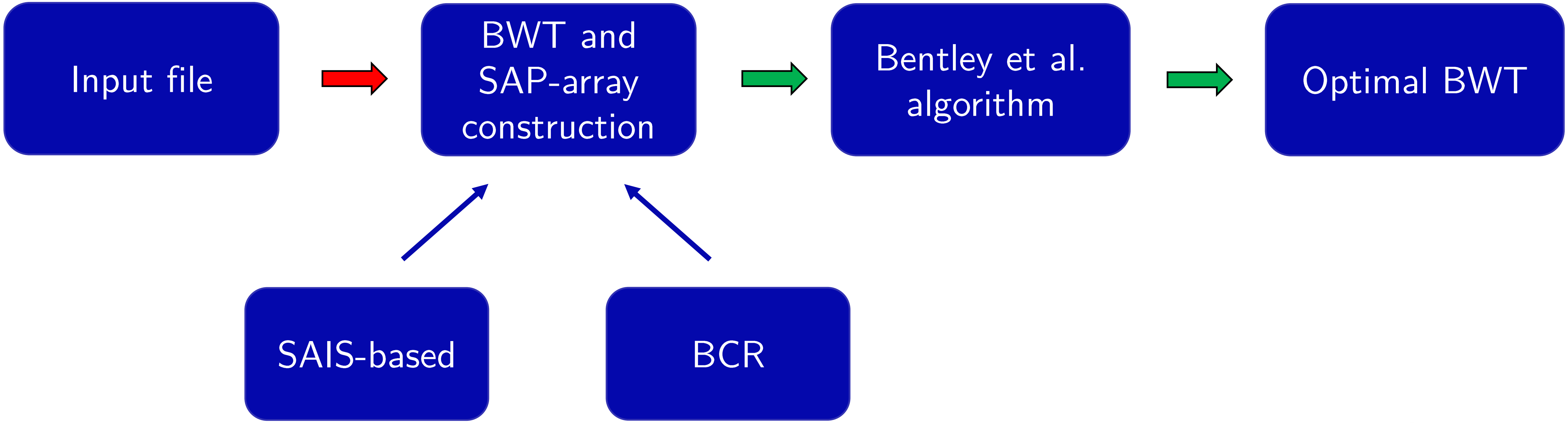
all rotations starting with \$ → 0 1 1 1 1 | 0 1 1 | 0 0 …

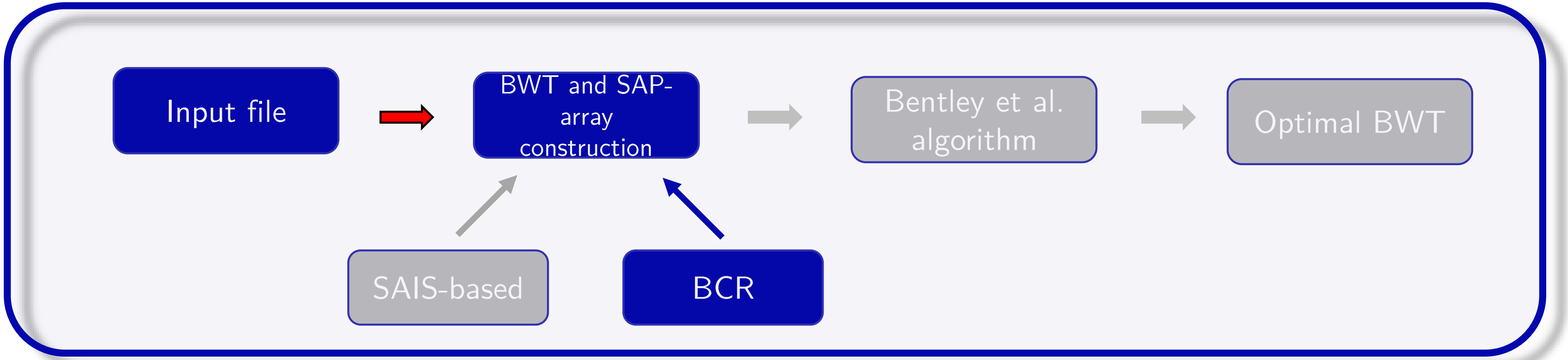
↑ all rotations starting with A\$

$\$_1$ … ATATG	G	0
$\$_2$ … TGA	A	1
$\$_3$ … ACG	G	1
$\$_4$ … ATCA	A	1
$\$_5$ … GGA	A	1
A$\$_2$ … TG	G	0
A$\$_4$ … ATC	C	1
A$\$_5$ … GG	G	1
ACG$\$_3$ … $\$_2$	\$	0
ATATG$\$_1$ … $\$_5$	\$	0
ATCA$\$_4$ … $\$_3$	\$	0
ATG$\$_1$ … AT	T	0
CA$\$_4$ … AT	T	0
CG$\$_3$ … A	A	0
G$\$_1$ … ATAT	T	0
G$\$_3$ … AC	C	1
GA$\$_2$ … T	T	0
GA$\$_5$ … G	G	1
GGA$\$_5$ … $\$_4$	\$	0
TATG$\$_1$ … A	A	0
TCA$\$_4$ … A	A	0
TG$\$_1$ … ATA	A	0
TGA$\$_2$ … $\$_1$	\$	0

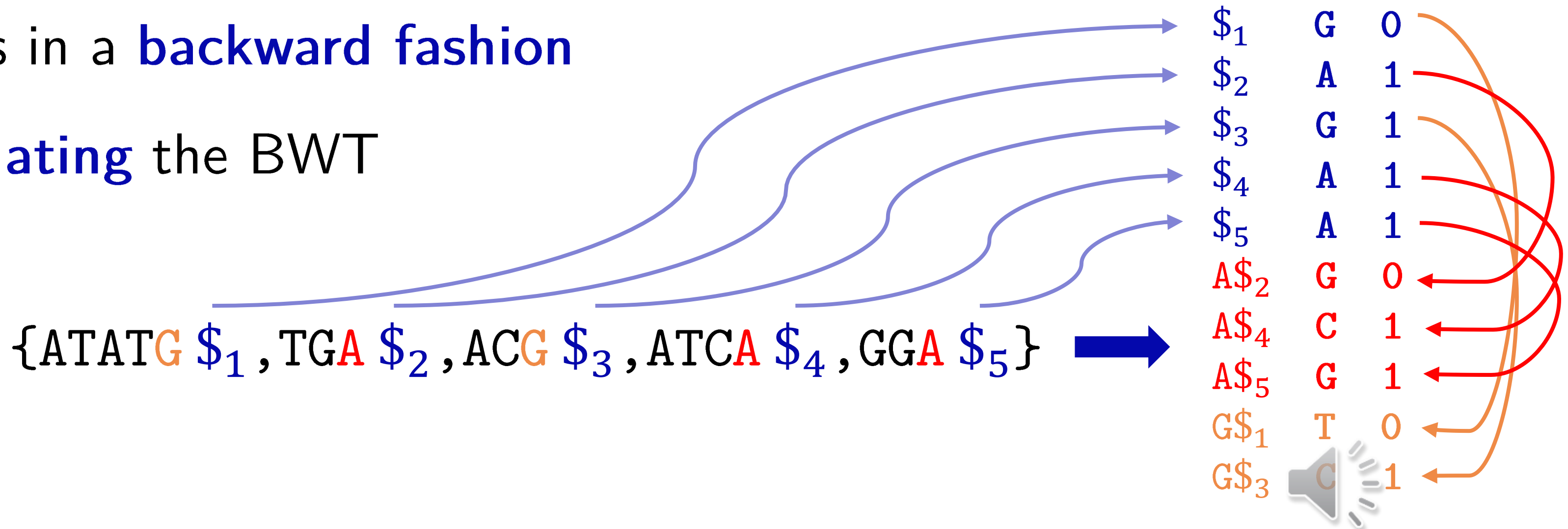
The Workflow

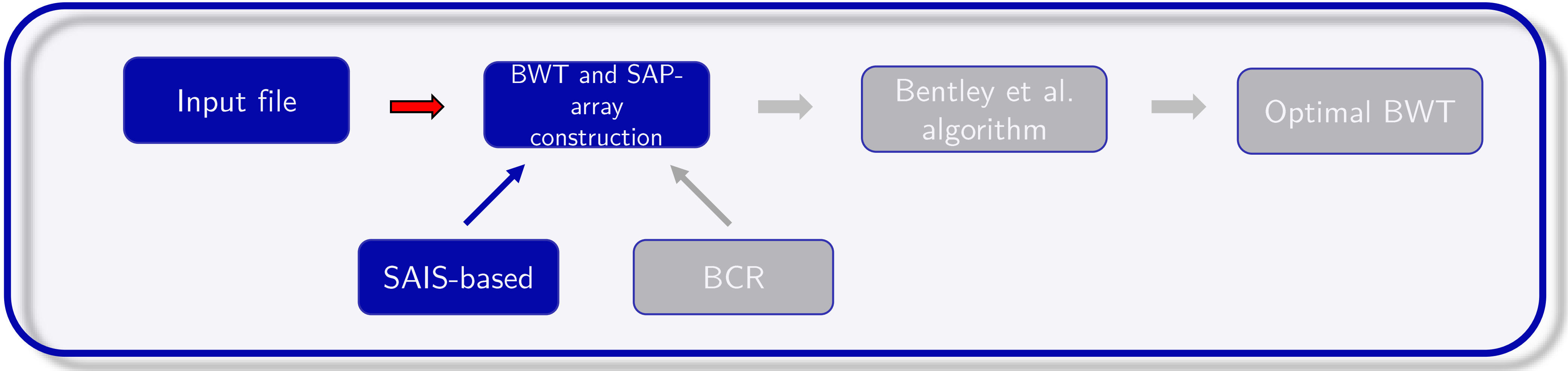
Our method is divided in two steps, first (→), we compute the BWT, and second (→), we compute the optimal permutation of the BWT characters.



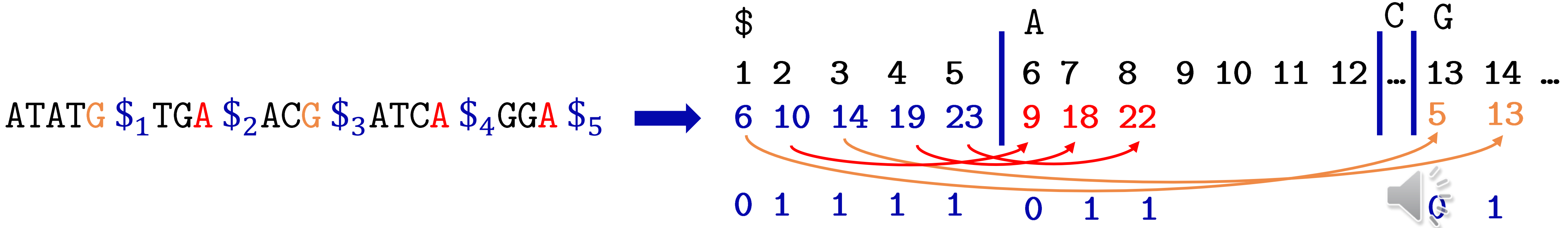


- BCR inserts BWT characters in a **backward fashion**
- update SAP-array **while updating** the BWT





- SAIS sort suffixes by **induced sorting**
- **update** SAP-array while inducing types



Bentley et al. reduced the problem of finding an optimal input permutation to the **Tuple Ordering** (TO) problem.

```
0 1 1 1 1 1 1 | 0 1 1 1 1 | 0 1 | 0 0 | 0 1 | 0 0 0 0 0 0 | 0 1 | 0 1 | 0 1 | 0 0 0
A A T A T A A | G A A C T | C T | C $ | G G | C A $ $ $ T | A C | A A | G G | $ $ $
```

(A,T) (A,C,G,T) (C,T)(C)($) (G) (C)(A)($)($)($)(T)(A,C) (A) (G) ($)($)($)

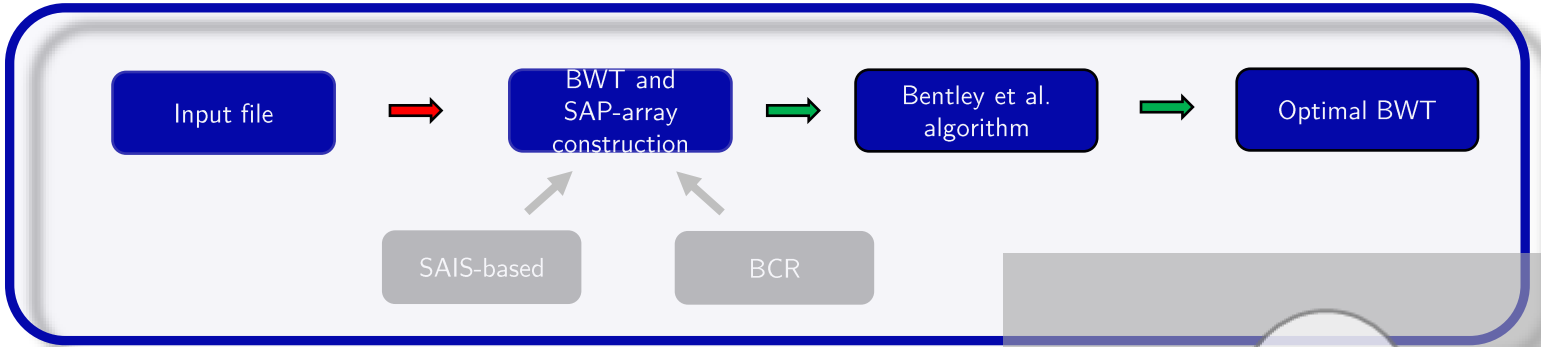
(A,T)(A,C,G,T)(C,T)(C)($)(G)(C)(A)($)($)($)(T)(A,C)(A)(G)($)($)($)

(T,A)(A,C,G,T)(T,C)(C)($)(G)(C)(A)($)($)($)(T)(C,A)(A)(G)($)($)($)

- we implemented this algorithm using a **stack** data structure containing the Parikh vectors of the SAP-intervals
- we keep inserting the Parikh vectors in the stack until we find either a fixed position or a tuple containing one character

(A,T)(A,C,G,T)(C,T)(C)($)(G)(C)(A)($)($)($)(T)(A,C)(A)(G)($)($)($)

$ A C G T
(0,5,0,0,2)

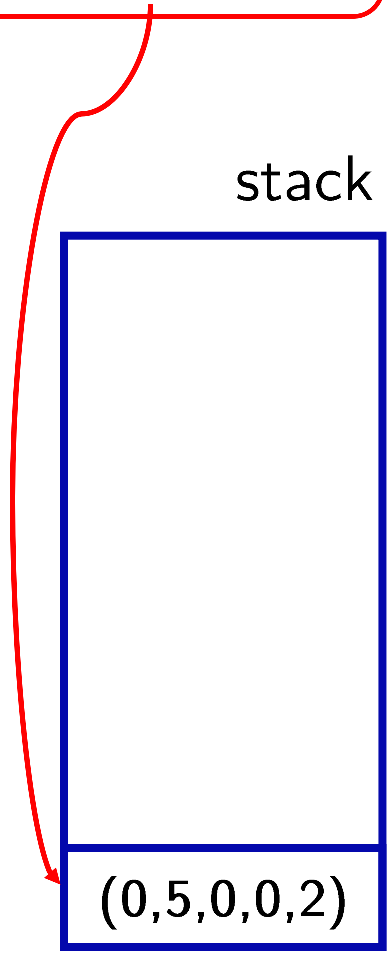


- we output the optimal BWT **permuting** different blocks separately

Computing the optimal BWT

0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0

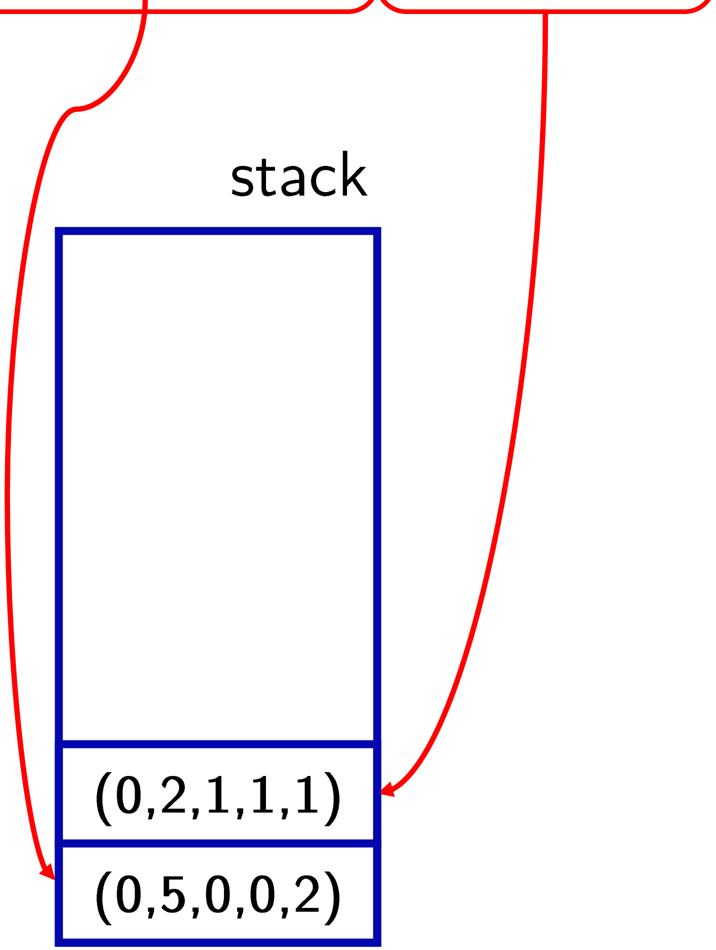
A A T A T A A G A A C T C T C $ G G C A $ $ $ T A C A A G G $ $ $



Algorithm 1 Procedure to process a Parikh vector P

```
if Stack is empty then
    if there is exactly one j such that P[j] > 0 then   // interval not interesting
        write P[j] copies of character j
    else
        if P[x] > 0 where x is the last character inserted in the BWT then
            write P[x] copies of the character x, P[x] ← 0
        end if
        Stack ← pushTop(P)                 // push a new Parikh vector on the stack
    end if
else
    T ← Stack.top()                              // first element of the stack
    if there are at least two j s.t. T[j] > 0 and P[j] > 0 then
        Stack ← pushTop(P)
    else
        write corresponding characters for each T in Stack   // see text for details
    end if
end if
```

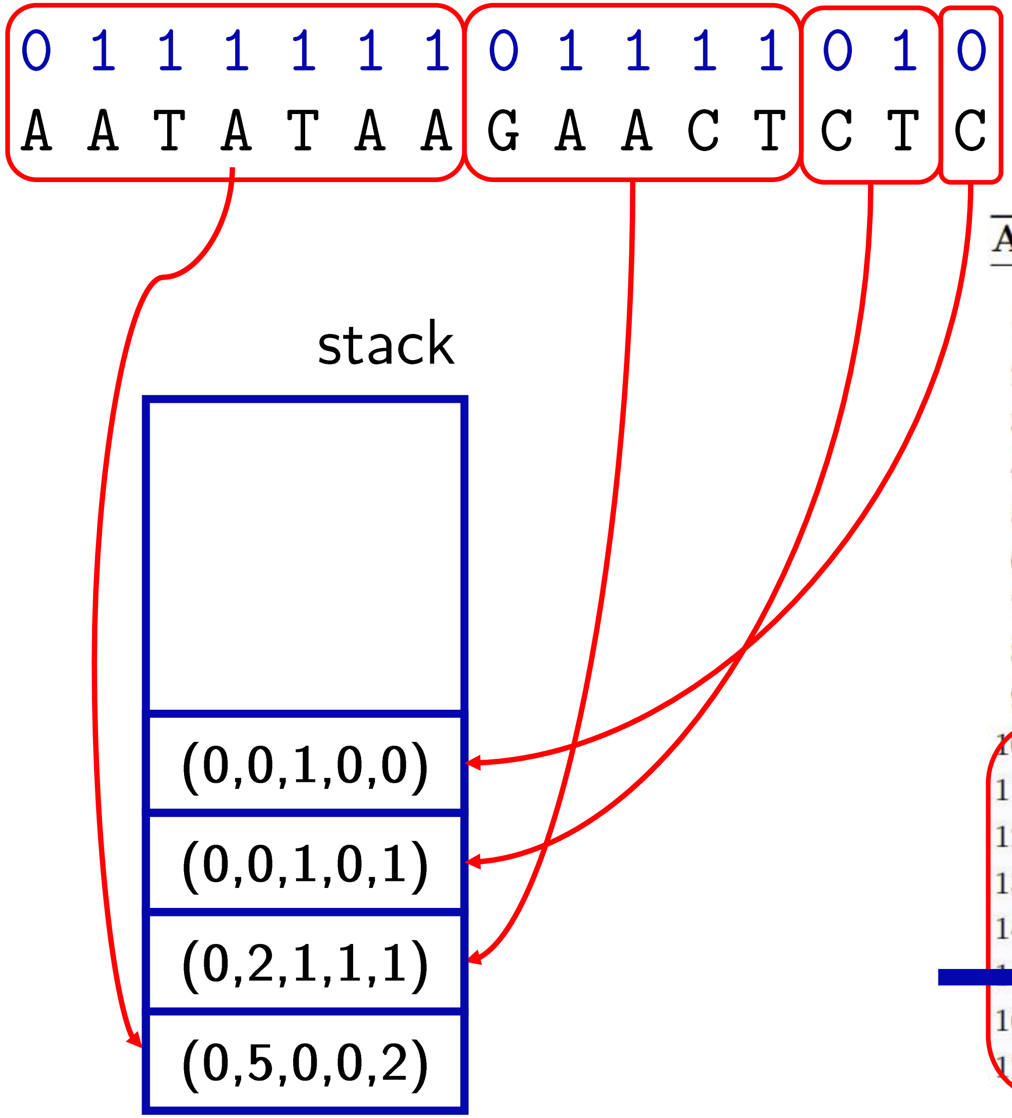
```
0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0
A A T A T A A G A A C T C T C $ G G C A $ $ $ T A C A A G G $ $ $
```



Algorithm 1 Procedure to process a Parikh vector P

```
if Stack is empty then
    if there is exactly one j such that P[j] > 0 then    // interval not interesting
        write P[j] copies of character j
    else
        if P[x] > 0 where x is the last character inserted in the BWT then
            write P[x] copies of the character x, P[x] ← 0
        end if
        Stack ← pushTop(P)                 // push a new Parikh vector on the stack
    end if
else
    T ← Stack.top()                        // first element of the stack
    if there are at least two j s.t. T[j] > 0 and P[j] > 0 then
        Stack ← pushTop(P)
    else
        write corresponding characters for each T in Stack   // see text for details
    end if
end if
```

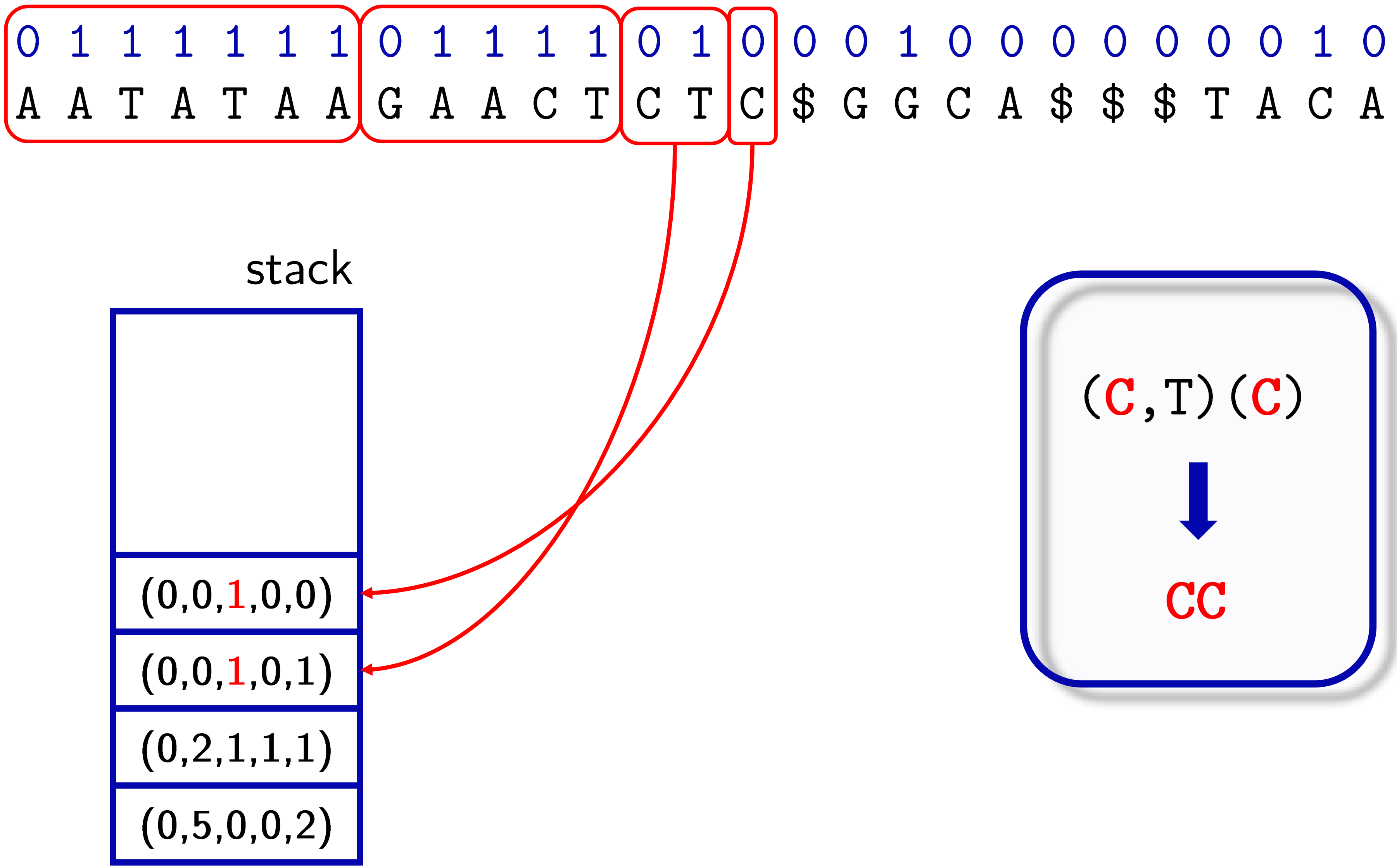
Computing the optimal BWT



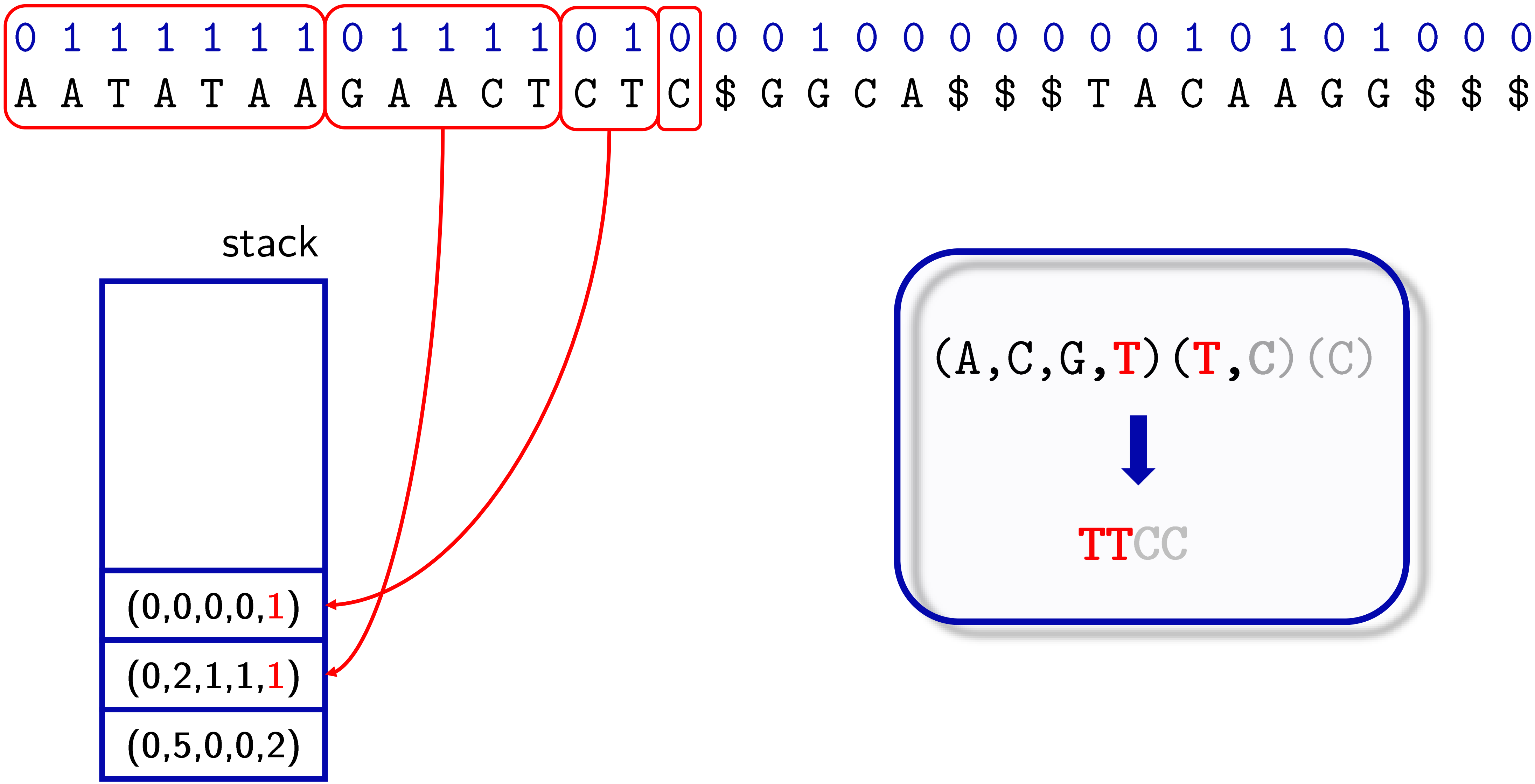
Algorithm 1 Procedure to process a Parikh vector P

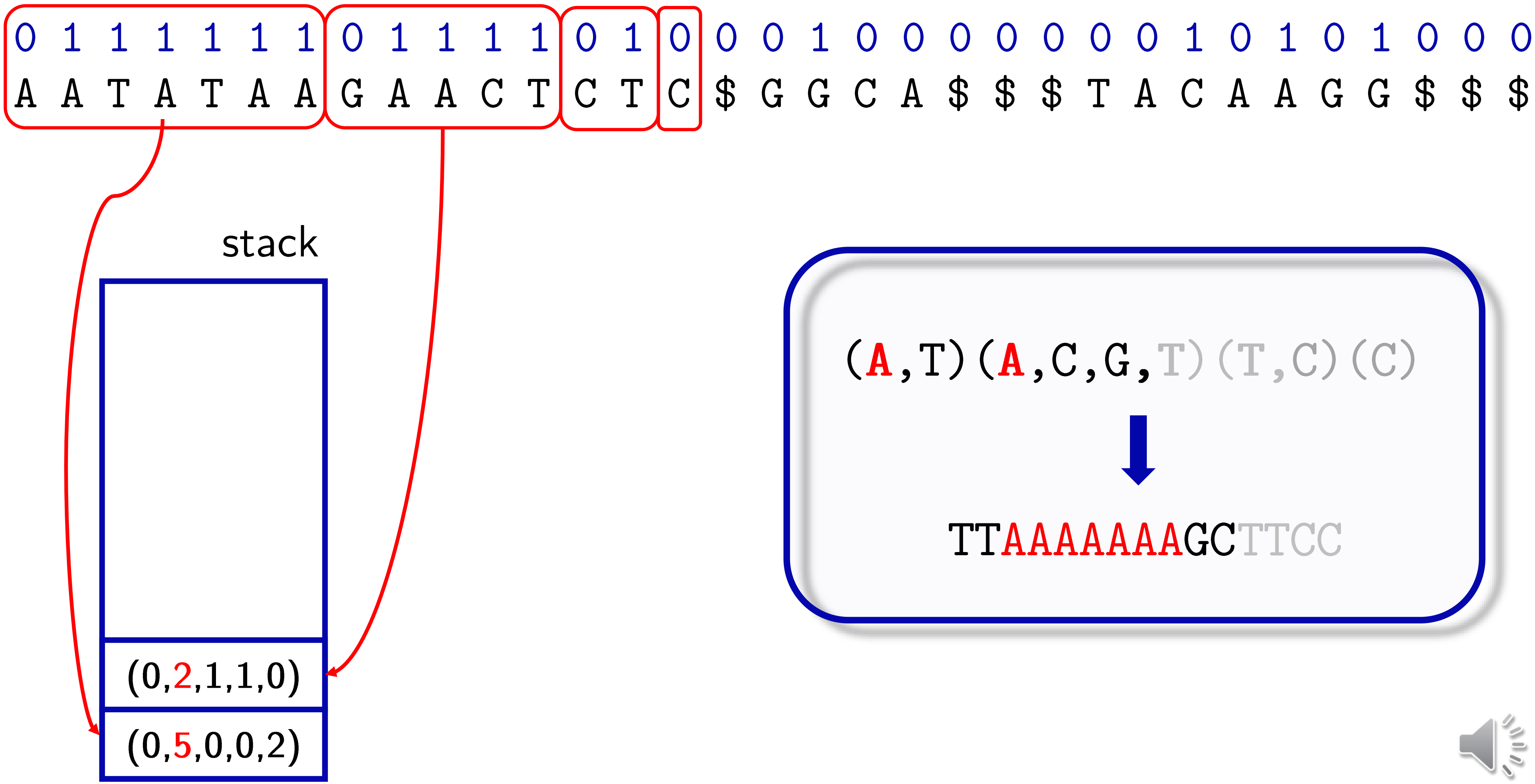
```
if Stack is empty then
    if there is exactly one j such that P[j] > 0 then    // interval not interesting
        write P[j] copies of character j
    else
        if P[x] > 0 where x is the last character inserted in the BWT then
            write P[x] copies of the character x, P[x] ← 0
        end if
        Stack ← pushTop(P)                // push a new Parikh vector on the stack
    end if
else
    T ← Stack.top()                       // first element of the stack
    if there are at least two j s.t. T[j] > 0 and P[j] > 0 then
        Stack ← pushTop(P)
    else
        write corresponding characters for each T in Stack   // see text for details
    end if
end if
```

Computing the optimal BWT

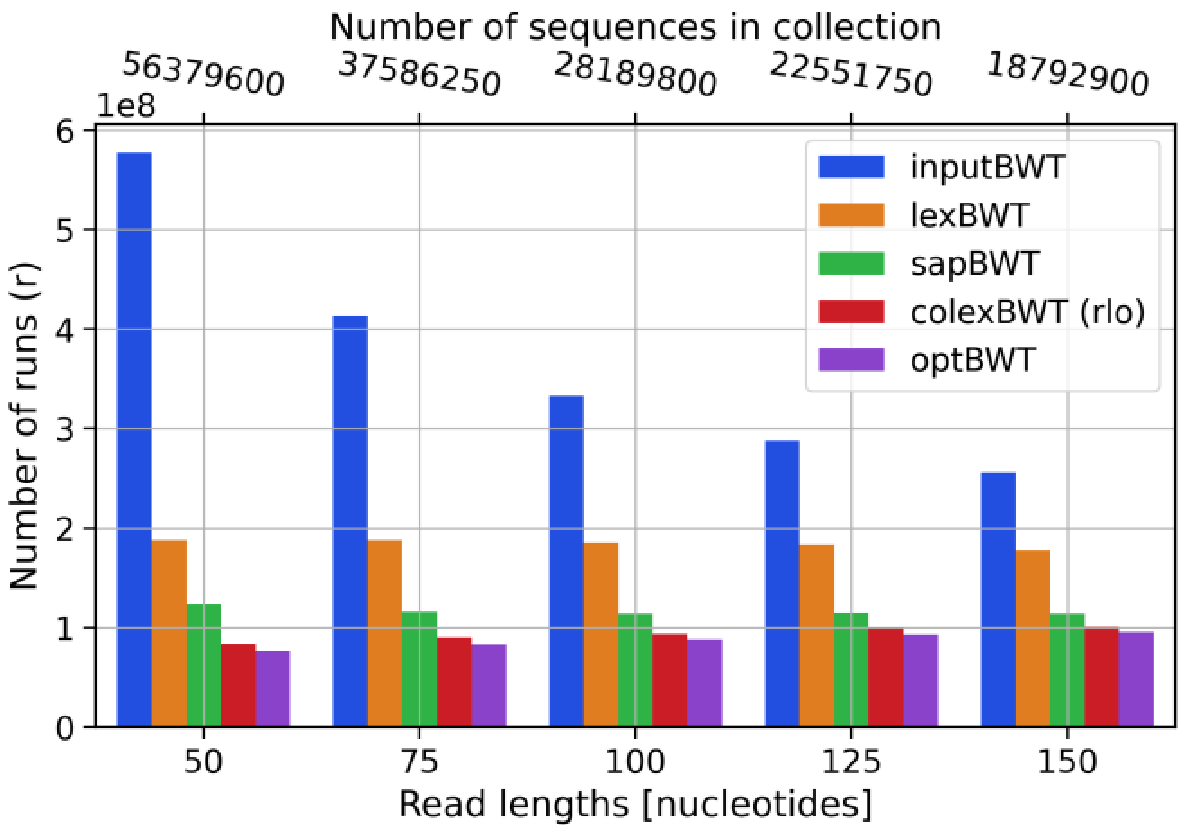


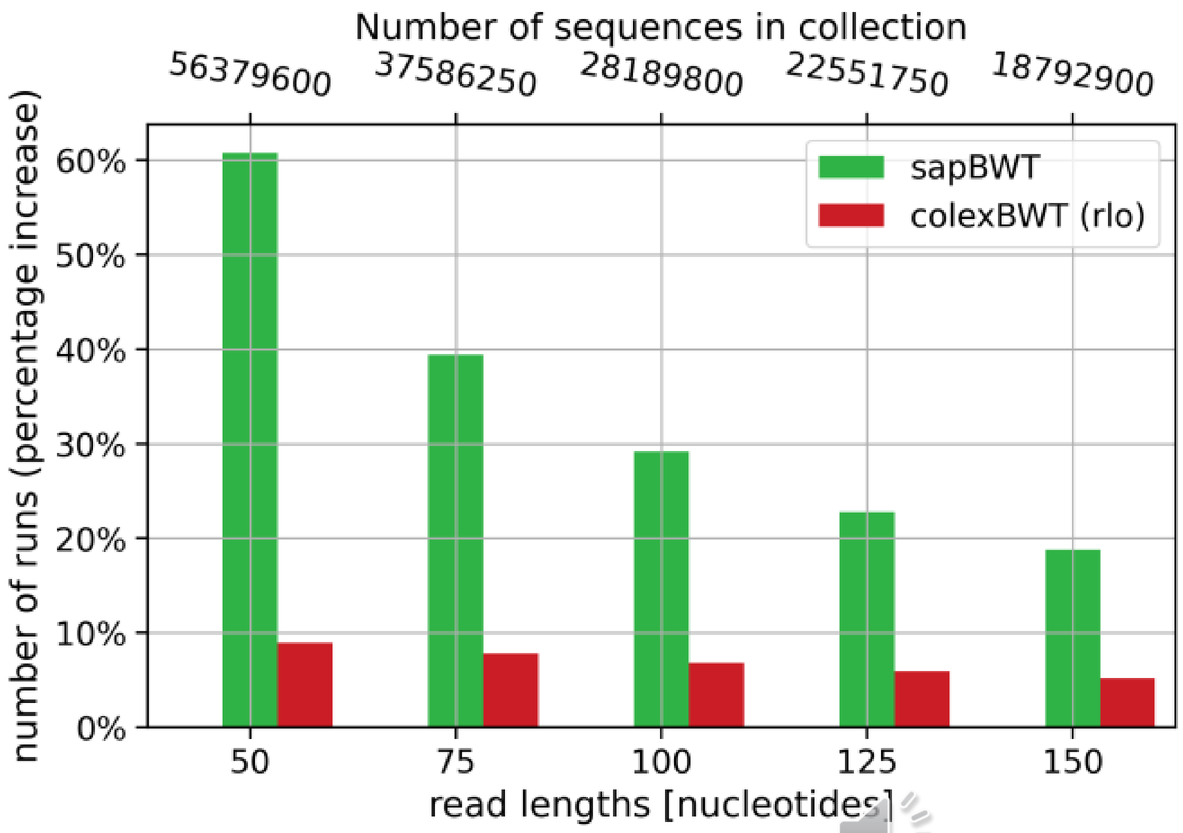
Computing the optimal BWT




0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0
A A T A T A A G A A C T C T C $ G G C A $ $ $ T A C A A G G $ $ $
stack
(0,2,1,1,0)
(0,5,0,0,2)
(A,T)(A,C,G,T)(T,C)(C)
TTAAAAAAAGCTTCC

- Pseudomonas **aeruginosa**





Real data

- we tested our method on 7 **real-life** datasets
- **different features**: read length, dataset size, n/r

dataset		description	length BWT n	len.	no. seq	r_{opt}	n/r_{opt}	n/r
1	ERR732065–70	*HIV-virus*	1,345,713,812	150	8,912,012	11,539,661	116.62	27.62
2	SRR12038540	*SARS-CoV-2 RBD*	1,690,229,250	50	33,141,750	14,864,523	113.71	8.08
3	ERR022075_1	*E. Coli str. K-12*	2,294,730,100	100	22,720,100	71,203,469	32.23	8.83
4	SRR059298	*Deformed wing virus*	2,455,299,082	72	33,634,234	48,376,632	50.75	9.83
5	SRR065389–90	*C. Elegans*	14,095,870,474	100	139,563,074	921,561,895	15.30	6.26
6	SRR2990914_1	*Sindibis virus*	15,957,722,119	36	431,289,787	105,250,120	151.62	4.81
7	ERR1019034	*H. Sapiens*	123,506,926,658	100	1,222,840,858	10,860,229,434	11.37	5.35

Real data

dataset		description	length BWT n	len.	no. seq	r_{opt}	n/r_{opt}	n/r
1	ERR732065–70	*HIV-virus*	1,345,713,812	150	8,912,012	11,539,661	116.62	27.62
2	SRR12038540	*SARS-CoV-2 RBD*	1,690,229,250	50	33,141,750	14,864,523	113.71	8.08
3	ERR022075_1	*E. Coli str. K-12*	2,294,730,100	100	22,720,100	71,203,469	32.23	8.83
4	SRR059298	*Deformed wing virus*	2,455,299,082	72	33,634,234	48,376,632	50.75	9.83
5	SRR065389–90	*C. Elegans*	14,095,870,474	100	139,563,074	921,561,895	15.30	6.26
6	SRR2990914_1	*Sindibis virus*	15,957,722,119	36	431,289,787	105,250,120	151.62	4.81
7	ERR1019034	*H. Sapiens*	123,506,926,658	100	1,222,840,858	10,860,229,434	11.37	5.35

data set	number of runs increase compared to optimal BWT (factor and perc.)				resource usage (optBWT)	
	inputBWT	colexBWT (rlo)	sapBWT	lexBWT	RAM (GB)	Time (hh:mm:ss)
1	**4.22** (322.26%)	1.03 (3.48%)	1.53 (53.06%)	1.30 (30.13%)	6.45 (1.02×)	7:18 (1.12×)
2	**14.07** (1306.95%)	1.15 (14.54%)	1.21 (20.75%)	3.52 (252.39%)	8.08 (1.03×)	6:32 (1.15×)
3	**3.65** (264.90%)	1.07 (6.52%)	1.30 (29.63%)	2.07 (107.01%)	11.15 (1.04×)	18:29 (1.26×)
4	**5.17** (416.52%)	1.04 (4.38%)	1.55 (55.33%)	1.55 (54.87%)	21.03 (1.02×)	22:08 (1.08×)
5	**2.44** (144.36%)	1.05 (5.05%)	1.16 (15.73%)	2.03 (103.35%)	4.31 (1.04×)	2:25:46 (1.28×)
6	**31.49** (3048.66%)	1.04 (4.30%)	1.79 (79.40%)	1.89 (89.17%)	8.86 (1.05×)	1:59:46 (1.39×)
7	**2.13** (112.56%)	1.04 (4.17%)	1.12 (11.89%)	1.96 (96.04%)	34.42 (1.03×)	26:24:18 (1.48×)

Conclusions

We presented the first tool for computing the BWT of a string collection that guarantees the **fewest** possible runs.

- number of runs reduction is **significant for all read lengths**
- on real-data, the optBWT can reduce r by up to a **31 factor**
- the space and time **overhead** to compute the optimal BWT is small
- can compute the optimal BWT of **very large** string collections
- implementation **available** on GitHub

Thank you for your attention

contact: davide.cenzato@unive.it

github: https://github.com/davidecenzato/optimalBWT