Burrows-Wheeler Transform

CS 4390/5390
Fall 2019

Rank and Select

Given a binary sequence *B*, define

$$rank_c(B, i) = \left| \left\{ i' \mid 1 \leq i' \leq i, B[i'] = c \right\} \right|$$

$$select_c(B, j) = arg \min_i \left\{ rank_c(B, i) = j \right\},$$

where $c \in \{0,1\}$

- the count of the number of *c*'s occurring before position *i* in *B*, and the j^{th} *c* in *B*
- note that $rank_0(B,i) = i - rank_1(B,i)$

Algorithms

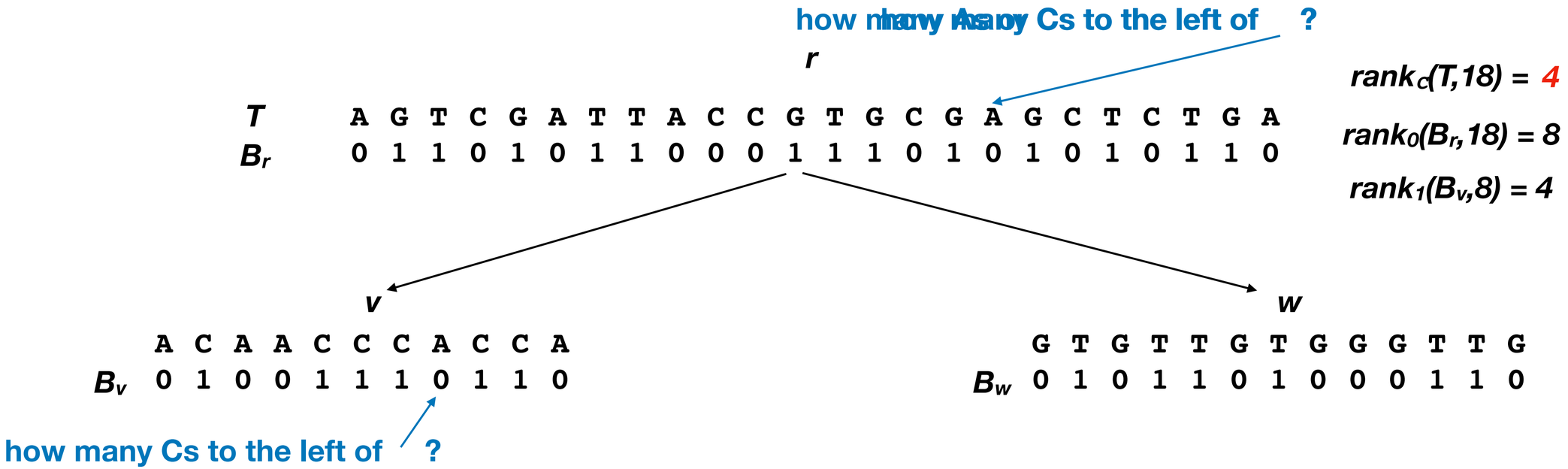
- *O(n log n)* space, *O(1)* time -- store all of the rank values in an array
- *O(n)* space, *O(n)* time -- compute rank manually for each value
- ***O(n)* space, *O(1)* time** -- store a subset of precomputed rank values (details omitted)

Wavelet Trees

Generalize *rank* and *select* to alphabet *Σ*

- can create σ binary strings and use independently, σn(1+o(1)) space
- more efficient model uses *n log σ(1+o(1))* bits, and O(*log σ)* time

Idea is to partition the alphabet and create a tree of bit vectors



Burrows-Wheeler Transform

Remember our old friend the suffix array?

T = `mississippi$`

SA_T	
12	`$`
11	`i$`
8	`ippi$`
5	`issippi$`
2	`ississippi$`
1	`mississippi$`
10	`pi$`
9	`ppi$`
7	`sippi$`
4	`sissippi$`
6	`ssippi$`
3	`ssissippi$`

Burrows-Wheeler Transform

Remember our old friend the suffix array?

T = `mississippi$`

SA_T	
12	`$mississippi`
11	`i$mississipp`
8	`ippi$mississ`
5	`issippi$miss`
2	`ississippi$m`
1	`mississippi$`
10	`pi$mississip`
9	`ppi$mississi`
7	`sippi$missis`
4	`sissippi$mis`
6	`ssippi$missi`
3	`ssissippi$mi`

Burrows-Wheeler Transform

Remember our old friend the suffix array?

T = `mississippi$`

SA_T	
12	`$mississippi`
11	`i$mississipp`
8	`ippi$mississ`
5	`issippi$miss`
2	`ississippi$m`
1	`mississippi$`
10	`pi$mississip`
9	`ppi$mississi`
7	`sippi$missis`
4	`sissippi$mis`
6	`ssippi$missi`
3	`ssissippi$mi`

Burrows-Wheeler Transform

Remember our old friend the suffix array?

T = `mississippi$`

SA_T		BWT_T
12	`$mississippi`	`i`
11	`i$mississip`	`p`
8	`ippi$mississ`	`s`
5	`issippi$miss`	`s`
2	`ississippi$m`	`m`
1	`mississippi$`	`$`
10	`pi$mississip`	`p`
9	`ppi$mississi`	`i`
7	`sippi$missis`	`s`
4	`sissippi$mis`	`s`
6	`ssippi$missi`	`i`
3	`ssissippi$mi`	`i`

$$BWT_T = \begin{cases} T\left[SA_T[i]-1\right] & \textbf{if } SA_T[i] > 1 \\ \textbf{\$} & \textbf{if } SA_T[i] = 1 \end{cases}$$

Burrows-Wheeler Transform

Claim given $BWT_T = L$, one can reconstruct T.

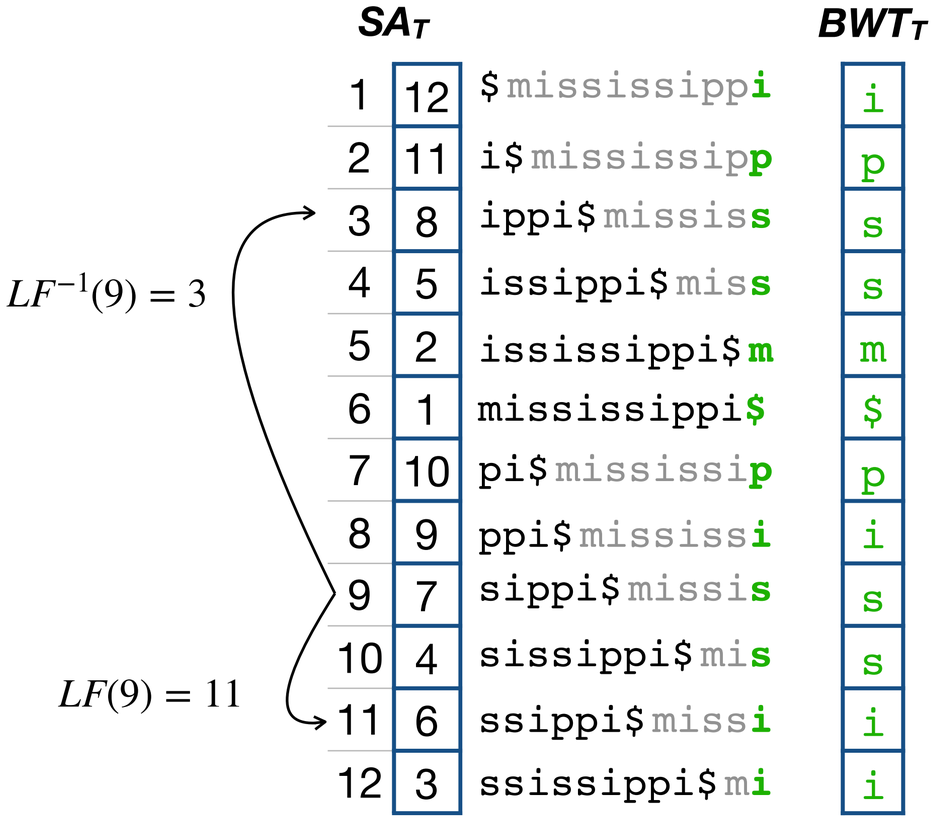
- let V and W be two suffixes of T such that $V < W$ (lexicographic order)
- assume both V and W are preceded by an a in T, then suffix $aV < aW$
- Separately, consider suffix $T[i..n]$ which coresponds to position p_i in SA_T
- if $L[p_i] = a$ is the k^{th} occurrence of a in L, then the suffix $T[i-1...n]$ is the k^{th} suffix in $\bar{a}$ (the region of SA of suffixes starting with a)

- define the last-to-first mapping:

$$LF(i) = j, \text{where } SA[j] = (SA[i] - 1)\,\%_1\, n$$

- similarly

$$LF^{-1}(j) = i, \text{where } SA[i] = (SA[j] + 1)\,\%_1\, n$$



$$i\,\%_1\, n = \begin{cases} n & \textbf{if } i = 0 \\ i & \textbf{if } i \in [1 \ldots n] \\ 1 & \textbf{if } i = n + 1 \end{cases}$$

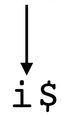
i	0	1	2	3	...	n-2	n-1	n	n+1
$i\%_1 n$	n	1	2	3	...	n-2	n-1	n	1

$$LF(i) = j, \text{where } SA[j] = (SA[i] - 1) \,\%_1\, n$$

	SA_T	BWT_T
1	12	i
2	11	p
3	8	s
4	5	s
5	2	m
6	1	$
7	10	p
8	9	i
9	7	s
10	4	s
11	6	i
12	3	i

$x_1 = 1$

$BWT_T[x_1]$



i $

$$LF(i) = j, \text{where } SA[j] = (SA[i] - 1) \,\%_1\, n$$

	SA_T	BWT_T
1	12	i
2	11	p
3	8	s
4	5	s
5	2	m
6	1	$
7	10	p
8	9	i
9	7	s
10	4	s
11	6	i
12	3	i

x_2 = LF(x_1) = LF(1) = 2

$BWT_T[x_2]$

↓

p i $

$$LF(i) = j, \text{where } SA[j] = (SA[i] - 1) \,\%_1\, n$$

	SA_T	BWT_T
1	12	i
2	11	p
3	8	s
4	5	s
5	2	m
6	1	$
7	10	p
8	9	i
9	7	s
10	4	s
11	6	i
12	3	i

$x_3 = LF(x_2) = LF(2) = 7$

$BWT_T[x_3]$

↓

ppi$

$$LF(i) = j, \text{where } SA[j] = (SA[i] - 1) \,\%_1\, n$$

	SA_T	BWT_T
1	12	i
2	11	p
3	8	s
4	5	s
5	2	m
6	1	$
7	10	p
8	9	i
9	7	s
10	4	s
11	6	i
12	3	i

$x_4 = LF(x_3) = LF(7) = 8$

$BWT_T[x_4]$

↓

ippi$

$$LF(i) = j, \text{where } SA[j] = (SA[i] - 1) \,\%_1\, n$$

	SA_T	BWT_T
1	12	i
2	11	p
3	8	s
4	5	s
5	2	m
6	1	$
7	10	p
8	9	i
9	7	s
10	4	s
11	6	i
12	3	i

$x_5 = LF(x_4) = LF(8) = 3$

$BWT_T[x_5]$

↓

sippi$

Compressibility

Suffix Arrays are not easily compressible, but BWTs are

Can we calculate *LF* without keeping *SA* around?

- yes! (big surprise 😒)
- keep an array *C[c]* that contains the number of occurrences of characters less than *c* in *T*
- then $LF(i) = C[L[i]] + rank_{L[i]}(L,i)$

start of the suffixes starting with *L[i]* in *SA*

which suffix *i* is in the range of things that start with *L[i]* in *SA*

BWT Index

A **BWT Index** for a sequence *T* is a data structure with:

- the $BWT_{T\$}$ encoded as a wavelet tree; and
- the integer array *C[0...σ],* where *C[c]* stores the number of occurances of the characters less than *c* in *T$*

With the BWT Index, you can:

- construct the Suffix Array
- recover *T* in *O(log n)* per character

Counting Occurrences

Input

- pattern, $P = p_1,p_2,p_3,...,p_m$
- count array, C
- $BWT_{T\$}$, L

Output

- number of occurrences of P in T

$i = m$

$(sp, ep) = (1,n)$

while $sp \leq ep$ **and** $i \geq 1$ **do**

  $c = p_j$

  $sp = C[c] + rank_c(L,sp-1)+1$

  $ep = C[c] + rank_c(L,ep)$

  $i = i - 1$

if $ep < sp$ **then**

  return 0

else

  return $ep - sp + 1$

Counting Occurrences

i = m
(sp, ep) = (1,n)
while *sp ≤ ep* **and** *i ≥ 1* **do**
    $c = p_j$
    $sp = C[c] + rank_c(L,sp-1)+1$
    $ep = C[c] + rank_c(L,ep)$
    i = i - 1
if *ep < sp* **then**
    return *0*
else
    return *ep - sp + 1*

$	**0**
A	**1**
C	**5**
G	**9**

P = AGC

p_i

$C[c] + rank_c(L,0) + 1$

$5 + 0 + 1$

$C[c] + rank_c(L,16)$

$5 + 4$

sp →	1	**C**	$
	2	**$**	AGAGCGAGAGCGCGC
	3	**G**	AGAGCGCGC$
	4	**G**	AGCGAGAGCGCGC$
	5	**G**	AGCGCGC$
	6	**G**	C$
	7	**G**	CGAGAGCGCGC$
	8	**G**	CGC$
	9	**G**	CGCGC$
	10	**C**	GAGAGCGCGC$
	11	**A**	GAGCGAGAGCGCGC$
	12	**A**	GAGCGCGC$
	13	**C**	GC$
	14	**A**	GCGAGAGCGCGC$
	15	**C**	GCGC$
ep →	16	**A**	GCGCGC$

Counting Occurrences

i = *m*
(sp, ep) = *(1,n)*
while *sp* ≤ *ep* **and** *i* ≥ *1* **do**
    c = p_j
    $sp = C[c] + rank_c(L,sp-1)+1$
    $ep = C[c] + rank_c(L,ep)$
    i = *i* - *1*
if *ep* < *sp* **then**
    return *0*
else
    return *ep* - *sp* + *1*

$	**0**
A	**1**
C	**5**
G	**9**

P = AGC

p_i

$C[G] + rank_G(L,5) + 1$

$9 + 3 + 1$

$C[G] + rank_G(L,9)$

$9 + 7$

1	**C**	$
2	**$**	AGAGCGAGAGCGCGC
3	**G**	AGAGCGCGC$
4	**G**	AGCGAGAGCGCGC$
5	**G**	AGCGCGC$
sp → 6	**G**	C$
7	**G**	CGAGAGCGCGC$
8	**G**	CGC$
ep → 9	**G**	CGCGC$
10	**C**	GAGAGCGCGC$
11	**A**	GAGCGAGAGCGCGC$
12	**A**	GAGCGCGC$
13	**C**	GC$
14	**A**	GCGAGAGCGCGC$
15	**C**	GCGC$
16	**A**	GCGCGC$

Counting Occurrences

i = *m*
(sp, ep) = *(1,n)*
while *sp* ≤ *ep* **and** *i* ≥ *1* **do**
    c = p_j
    $sp = C[c] + rank_c(L, sp-1)+1$
    $ep = C[c] + rank_c(L, ep)$
    i = *i* - *1*
if *ep* < *sp* **then**
    return *0*
else
    return *ep* - *sp* + *1*

P = AGC

p_i

$C[A] + rank_A(L,13) + 1$

$1 + 2 + 1$

$C[A] + rank_A(L,16)$

$1 + 4$

$	0
A	1
C	5
G	9

The BWT Index doesn't contain the SA, how do we recover the positions in T?

	1	C	$
	2	$	AGAGCGAGAGCGCGC
	3	G	AGAGCGCGC$
	4	G	AGCGAGAGCGCGC$
	5	G	AGCGCGC$
	6	G	C$
	7	G	CGAGAGCGCGC$
	8	G	CGC$
	9	G	CGCGC$
	10	C	GAGAGCGCGC$
	11	A	GAGCGAGAGCGCGC$
	12	A	GAGCGCGC$
sp →	13	C	GC$
	14	A	GCGAGAGCGCGC$
	15	C	GCGC$
ep →	16	A	GCGCGC$

Succinct Suffix Arrays

Recall that storing the whole suffix array would take $O(n \log n)$ space

Sample and store only the locations where $SA_{T\$}[i] = rk$ where $1 \leq k \leq n/r$

- call it *samples[1...n/r]*
- keep an additional bit vector of length n, where $B[i] = 1$ if $SA_{T\$}[i] = rk$

Recovering SA value at position i:

- if $B[i]=1$, return *samples[rank(B,i)]*
- otherwise set $j = i$, then $j = LF(j) = C[L[j]] + rank_{L[j]}(L,j)$ until $B[j]=1$ say d iterations
  - then $SA[i] = samples[rank(B,j)] + d$

Substrings

a **self-index** is a succinct representation of a string *T*

create a new sampling *pos2rank[1...n/r]* such that:

- *pos2rank[i] = j* if $SA_{T\$}[j] = ri$

to recover a substring *T[e...f]* :

- go to position *i = pos2rank[(f/r) + 1]*
- use *LF* backtracking to recover the string

Succinct Suffix Arrays

A **succinct suffix array** for a string *T* is a data structure containing:

- BWT of *T$, L* — *O(n log σ)* space
- count array *C* — *O(σ)* space
- *samples* array, and its indicator array *B* — *O(n/r log n)* and *O(n)* space
- *pos2rank* array — *O(n/r log n)* space

O(*n log σ + n/r log n*) total

and supports the following queries

- *count(P)* -- counts the occurrences of *P* in *T* in *O(m log σ)* time
- *locate(i)* -- returns $SA_{T\$}[i]$ in *O(r log σ)* time
- *substring(e,f)* -- returns *T[e...f]* in *O(((f-e)+r) log σ)* time

Bidirectional BWT

Given a string T, let $\underline{T}$ be the reverse string

Let $\boldsymbol{I}(W,T)$ returns the interval for suffixes prefixed by W in $BWT_{T\$}$

Bidirectional BWT

Given a string *T* a **bidirectional BWT index** is a data structure with the following operations:

- *isLeftMaximial(i,j)* -- 1 if $BWT_{T\$}[i...j]$ contains more than one value, 0 otherwise
- *isRightMaximial(i,j)* -- 1 if $BWT_{\underline{T}\$}[i...j]$ contains more than one value, 0 otherwise
- *enumerateLeft(i,j)* -- return the distinct values $BWT_{T\$}[i...j]$ in lexicographic order
- *enumerateRight(i,j)* -- return the distinct values $BWT_{\underline{T}\$}[i...j]$ in lexicographic order
- *extendLeft(c,**I**(W,T),**I**(<u>W</u>,<u>T</u>))* -- returns the pair (**I**(cW,T),**I**(<u>W</u>c,<u>T</u>))
- *extendRight(c,**I**(W,T),**I**(<u>W</u>,<u>T</u>))* -- returns the pair (**I**(Wc,T),**I**(c<u>W</u>,<u>T</u>))

Bidirectional BWT

Theorem Given a string T there is a representation of the bidirectional BWT of T that:

- $2n \log \sigma (1 + o(1))$ bits of space
- supports *isLeftMaximal*, *isRightMaximal*, *extendLeft*, & *extendRight* in $O(\log \sigma)$ time
- supports *enumerateLeft* & *enumerateRight* in $O(d \log (\sigma/d))$ time (where d is the size of the output)

Suffix Tree Traversal

Given bidirectional BWT *idx* of string *T* (interval *[1...n+1]* represents the root)

Output pairs *(v, |ℓ(v)|)* for all noes *v* in the suffix tree of *T* where *v* is the interval of *v* in the suffix array of *T$*

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

(2,4),(2,4),1
~~(1,8),(1,8),0~~

(2,4),(2,4)
(5,5),(5,5)
(6,6),(6,6)
(7,8),(7,8)

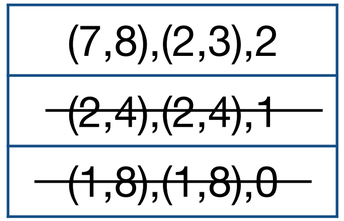
1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

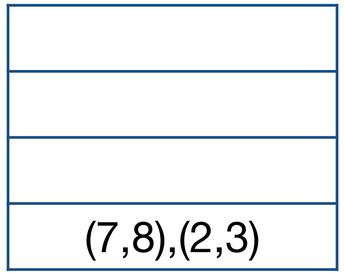
1	**A**	$
2	**G**	A$
3	**T**	ATCGA$
4	**$**	ATATGCA$
5	**A**	CGA$
6	**C**	GA$
7	**A**	TATGCA$
8	**A**	TCGA$

Output

[1,8], 0 " "

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```





1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

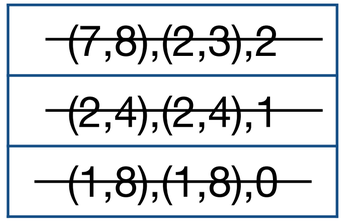
1	**A**	$
2	**G**	A$
3	**T**	ATCGA$
4	**$**	ATATGCA$
5	**A**	CGA$
6	**C**	GA$
7	**A**	TATGCA$
8	**A**	TCGA$

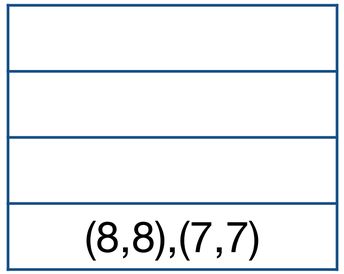
Output

[1,8], 0 " "

[2,4], 1 "A"

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```





1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

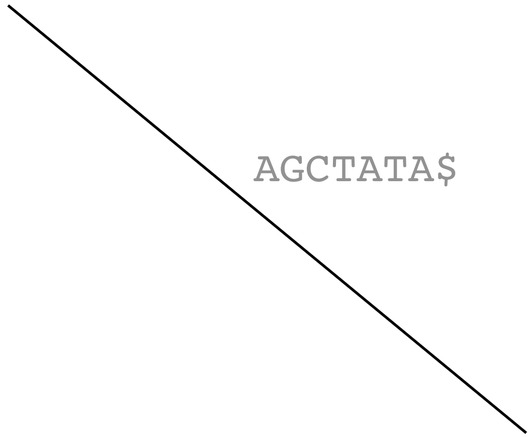
1	**A**	$
2	**G**	A$
3	**T**	ATCGA$
4	**$**	ATATGCA$
5	**A**	CGA$
6	**C**	GA$
7	**A**	TATGCA$
8	**A**	TCGA$

Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

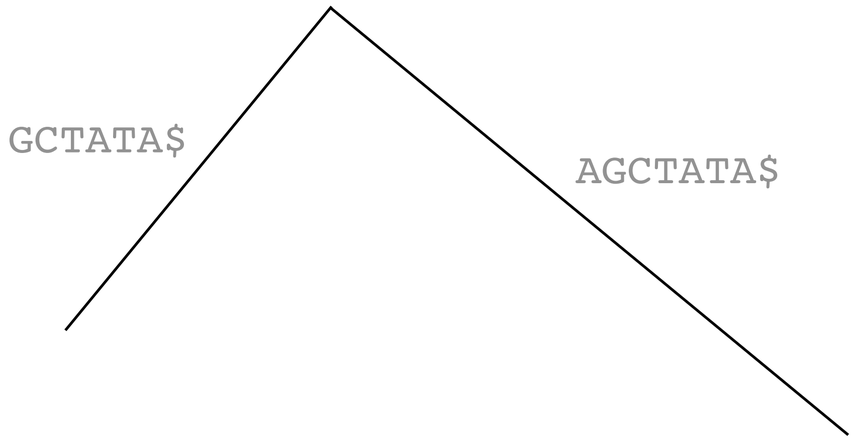


Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

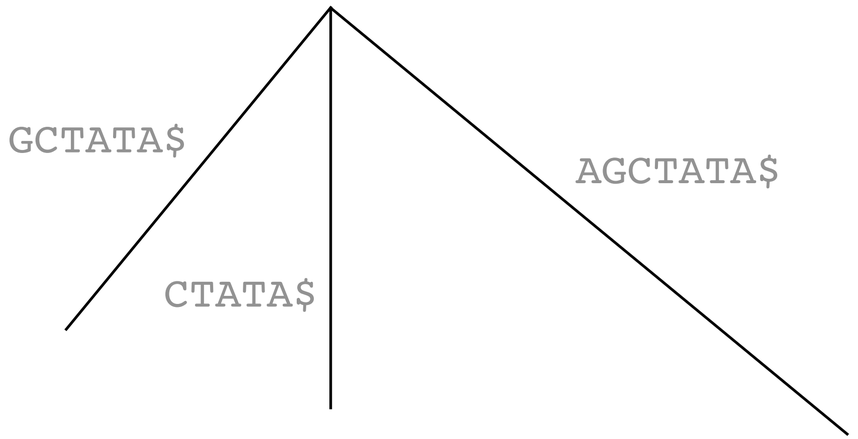


Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

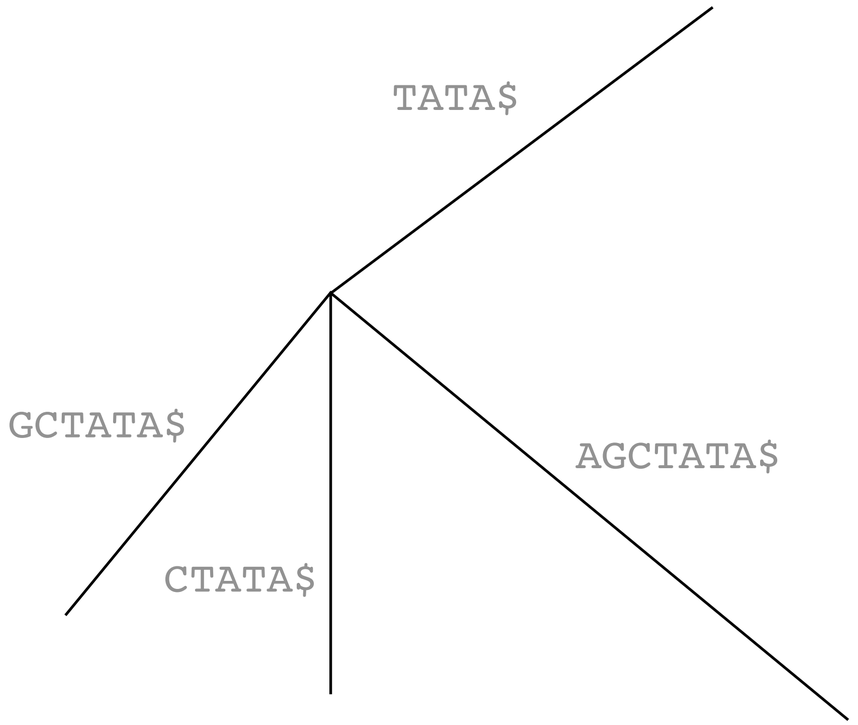
1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$



Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

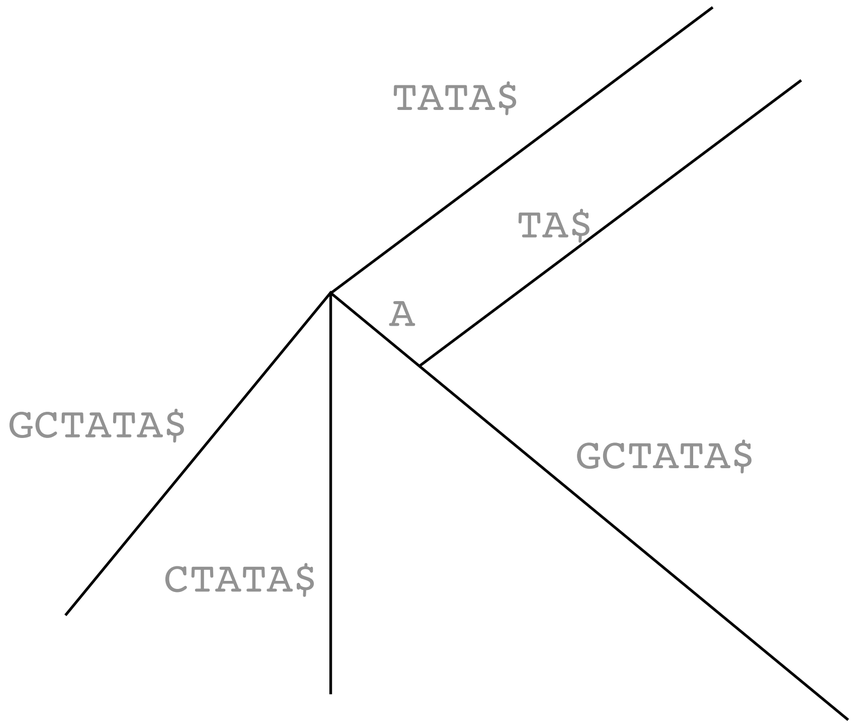


1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

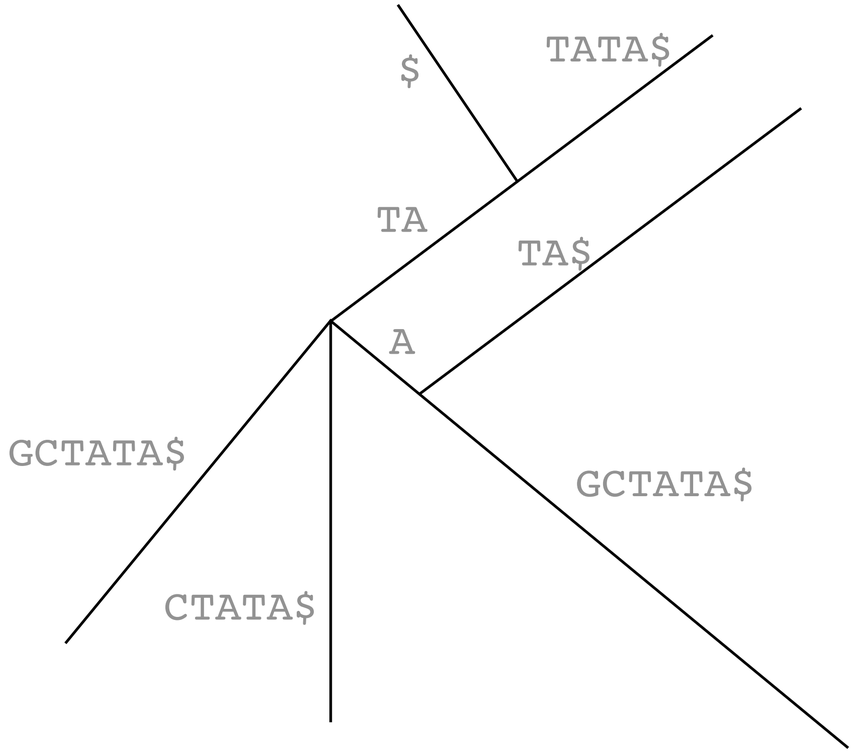


1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

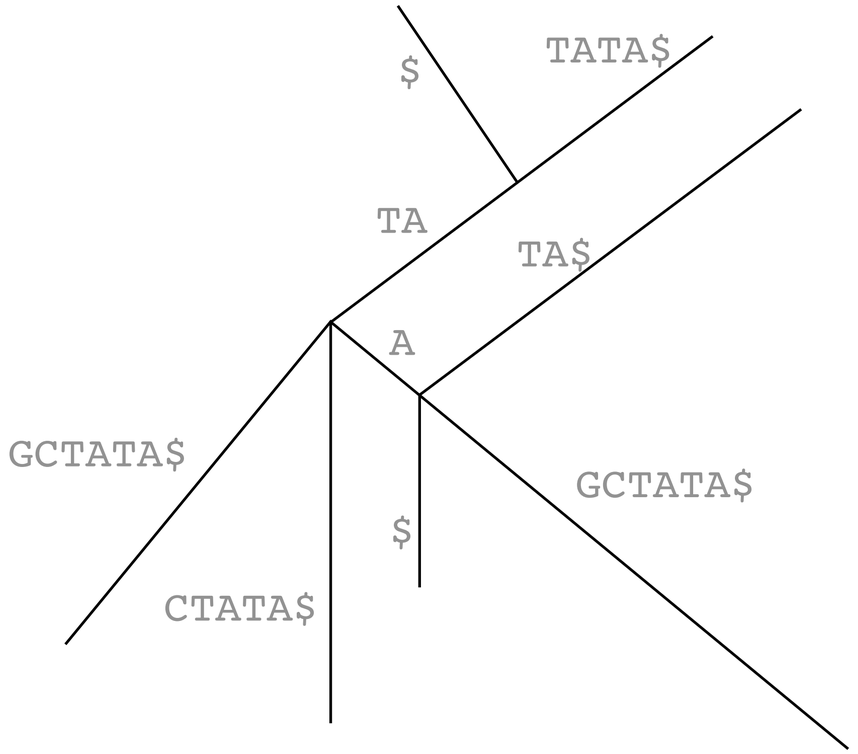


1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```

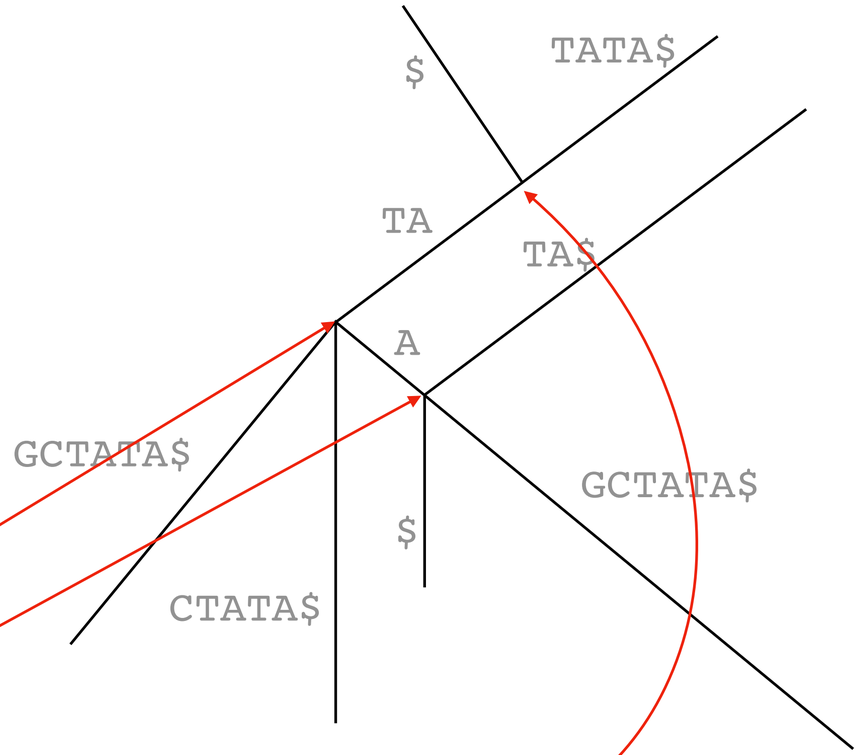


1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

```
S = empty stack
S.push(([1...n+1], [1...n+1], 0))
while S is not empty do
    ([i,j],[i',j'],d) = S.pop()
    output ([i,j],d)
    Σ' = idx.enumerateLeft(i,j)
    I = ∅
    for c ∈ Σ' do
        I = I ∪ {idx.exgendLeft(c,[i,j],[i',j'])}
    for ([i,j],[i',j']) ∈ I do
        if idx.isRightMaximal(i',j') then
            S.push(([i,j],[i',j'],d+1))
```



1	**A**	$
2	**T**	A$
3	**$**	AGCTATA$
4	**T**	ATA$
5	**G**	CTATA$
6	**A**	GCTATA$
7	**A**	TA$
8	**C**	TAT$

Output

[1,8], 0	**""**
[2,4], 1	**"A"**
[7,8], 2	**"TA"**

Summary

Suffix Arrays and perform all of the same functionality as Suffix Trees in a consistent size, *O(n log n)*

The BWT can recreate the *SA* and all of *T* in *O(n log σ)* space

in most cases $n >> \sigma$, so this savings is significant

Think human-genome
$n = 3 \times 10^9$
$\sigma = 4$