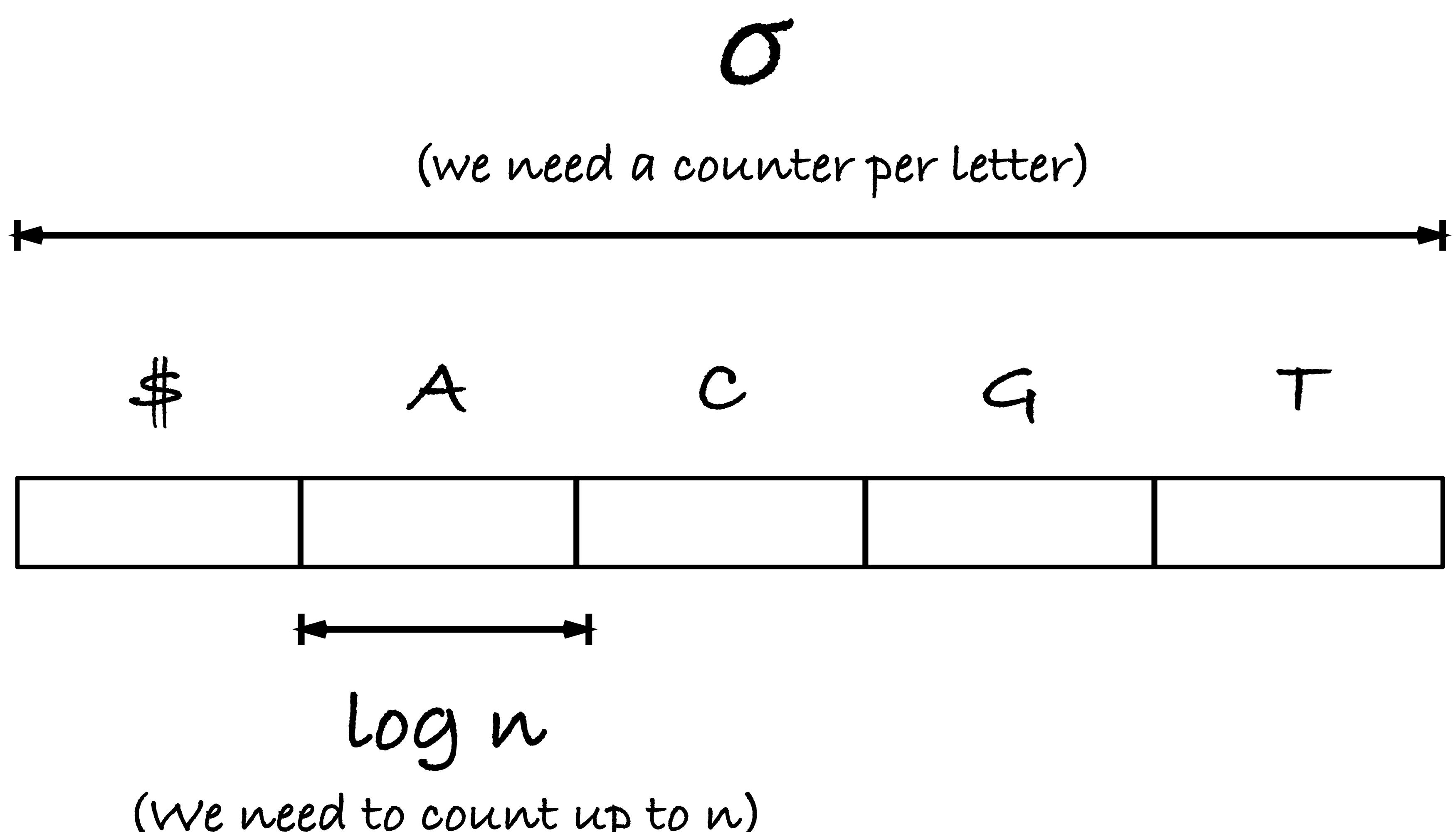
Saving space

Some tricks for saving space when readmapping, and a very useful data structure for strings

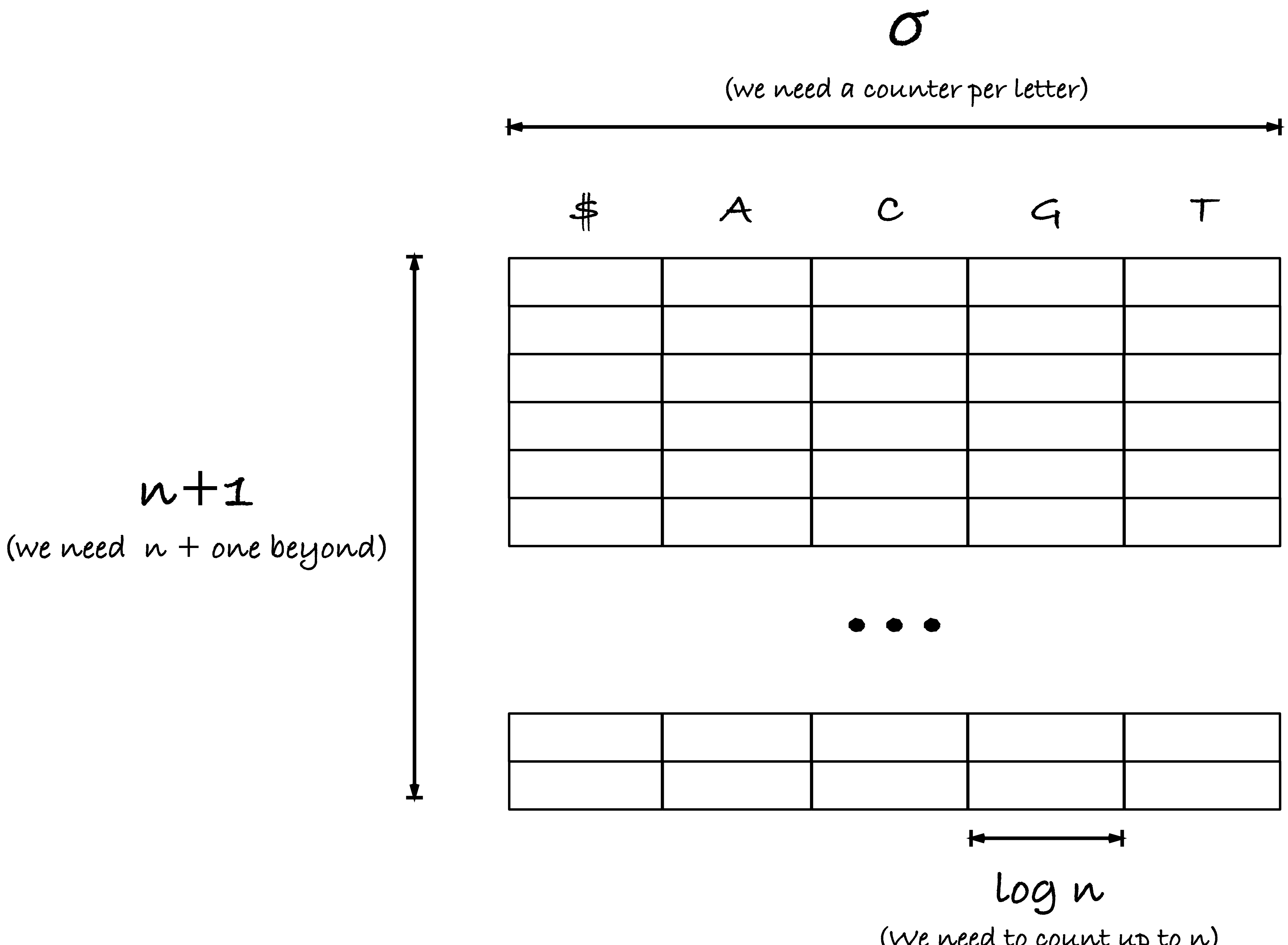
How much memory are we using?

Structure	**Usage**
C table	Rotations (select)
O table	Rotations (rank)
RO table	Computing D
p	Need to know the pattern
D table	Bounding branches
SA	Reporting matches

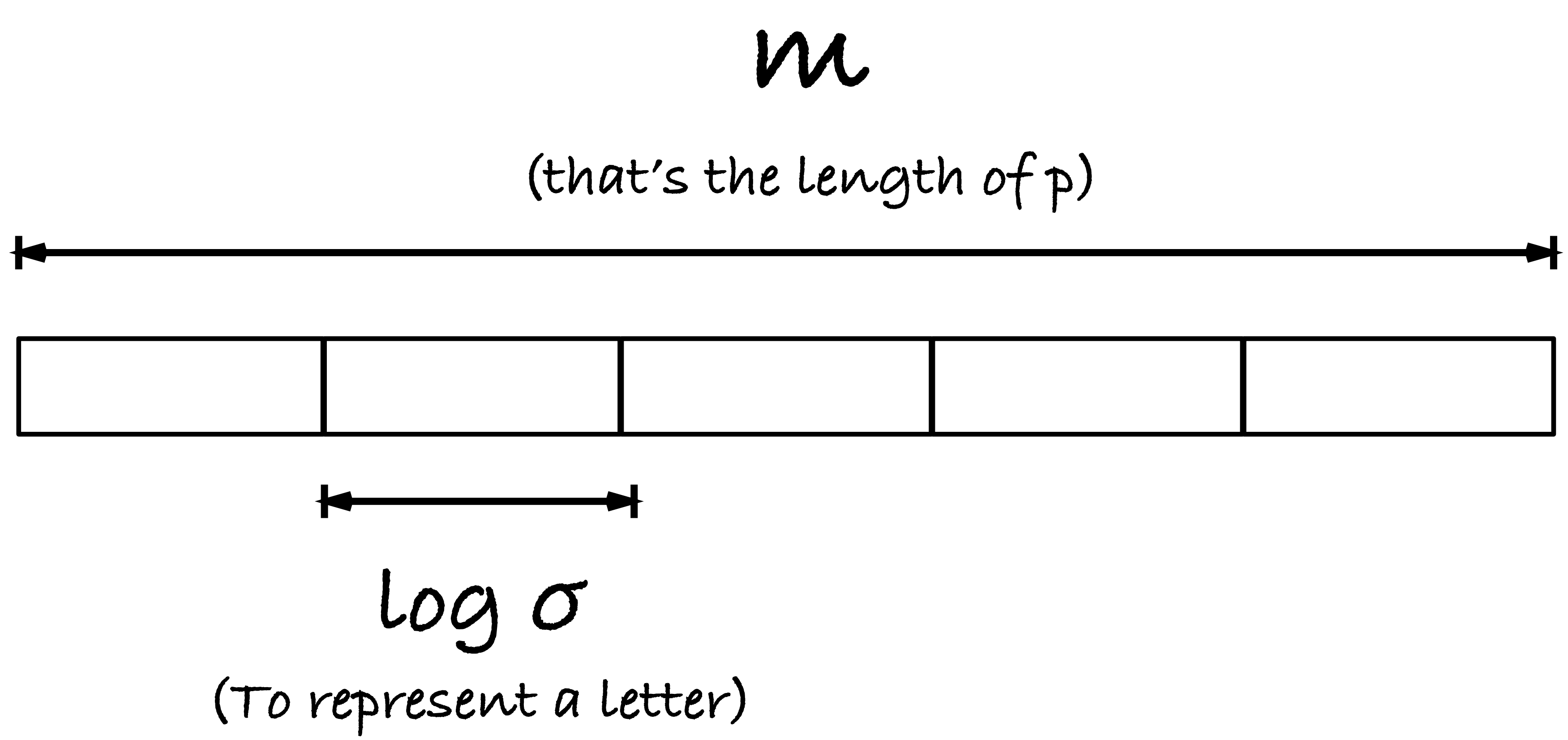
Space for C


σ
(we need a counter per letter)
$
A
C
G
T
log n
(We need to count up to n)

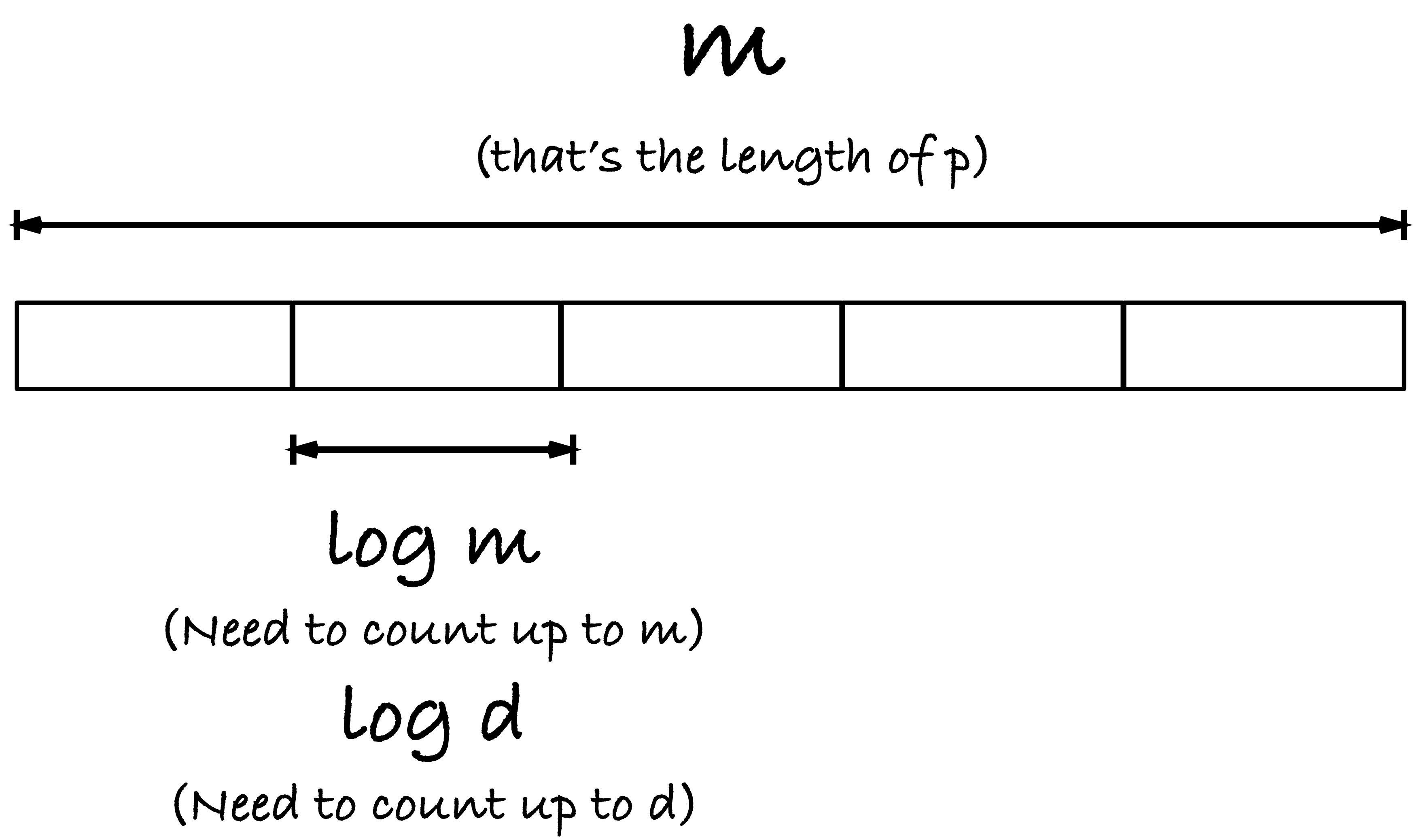
Space for O and RO



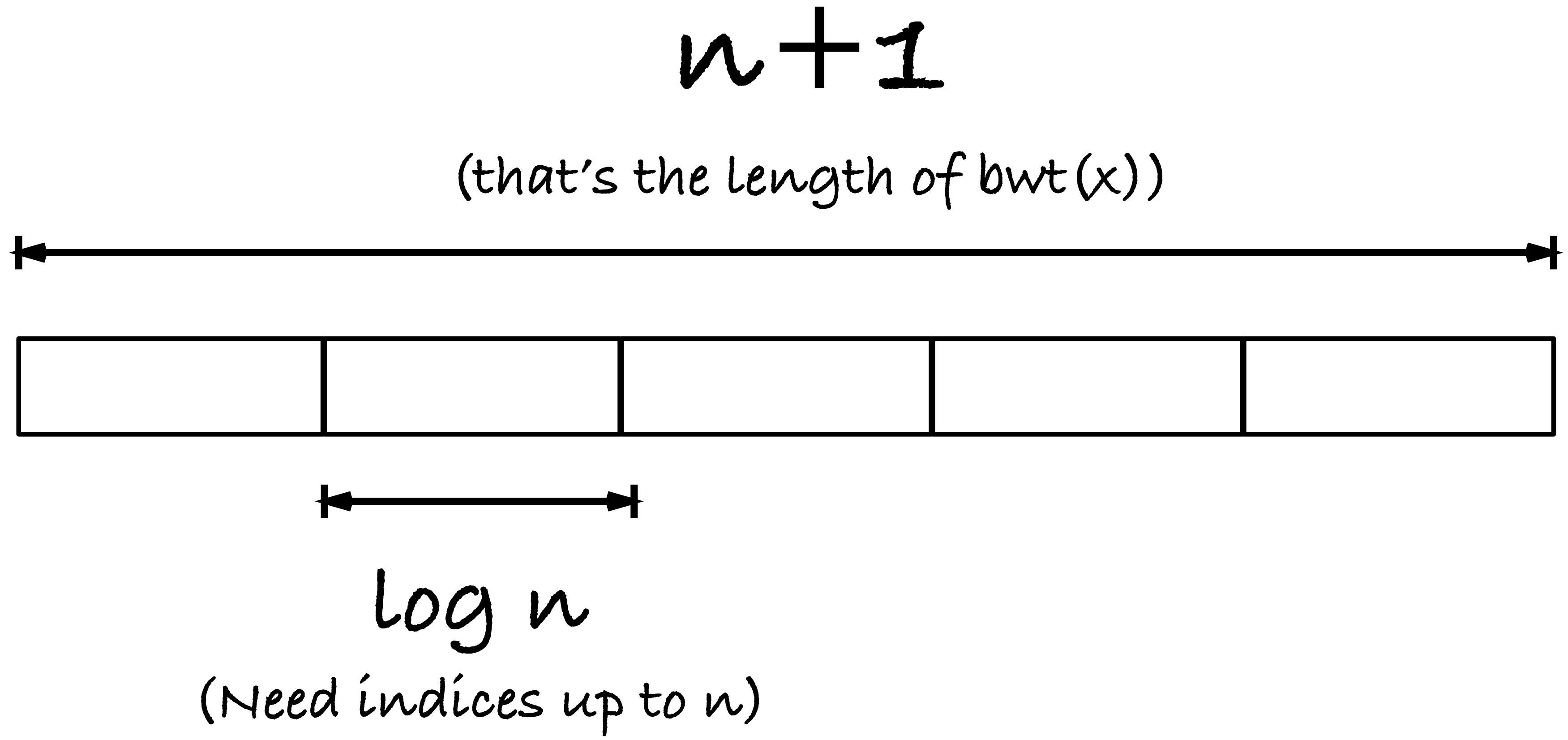
Space for p



Space for D


m
(that's the length of p)
log m
(Need to count up to m)
log d
(Need to count up to d)

Space for SA



How much memory are we using?

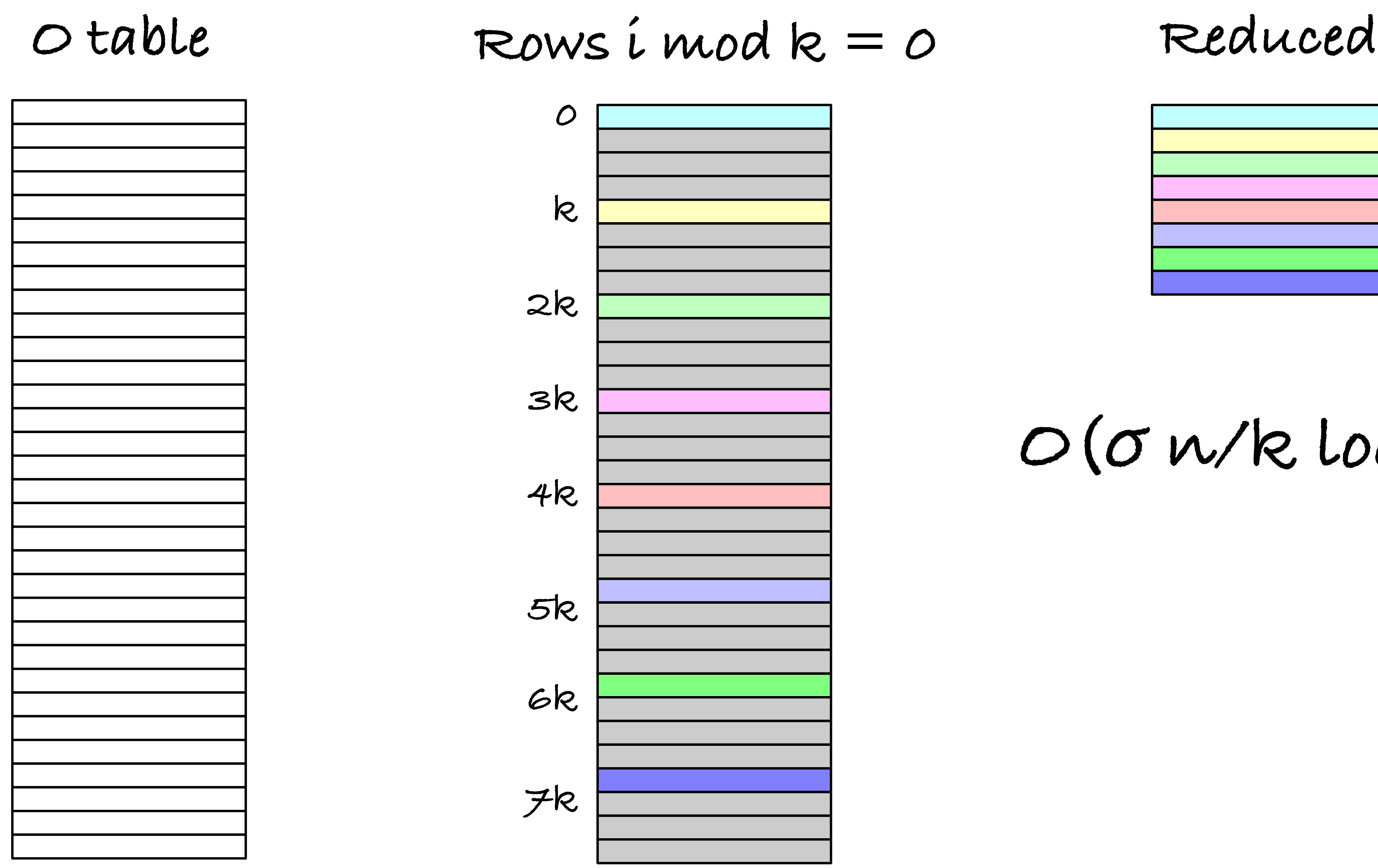
Structure	**Usage**	**Memory (bits)**
C table	Rotations	$O(\sigma \log n)$
O table	Rotations	$O(\sigma n \log n)$
RO table	Computing D	$O(\sigma n \log n)$
p	Need to know the pattern	$O(m \log \sigma)$
D table	Bounding branches	$O(m \log m)$ / $O(m \log d)$
SA	Reporting matches	$O(n \log n)$

How much memory are we using?

The troublesome ones are those that has a factor of n log n.
The parameters σ, m, and d are tiny, and log n
isn't large by itself. Factors of n log n, that is the problem.

		Memory (bits)
C table	Rotations	$O(\sigma \log n)$
O table	Rotations	**$O(\sigma n \log n)$**
RO table	Computing D	**$O(\sigma n \log n)$**
p	Need to know the pattern	$O(m \log \sigma)$
D table	Bounding branches	$O(m \log m)$ / $O(m \log d)$
SA	Reporting matches	**$O(n \log n)$**

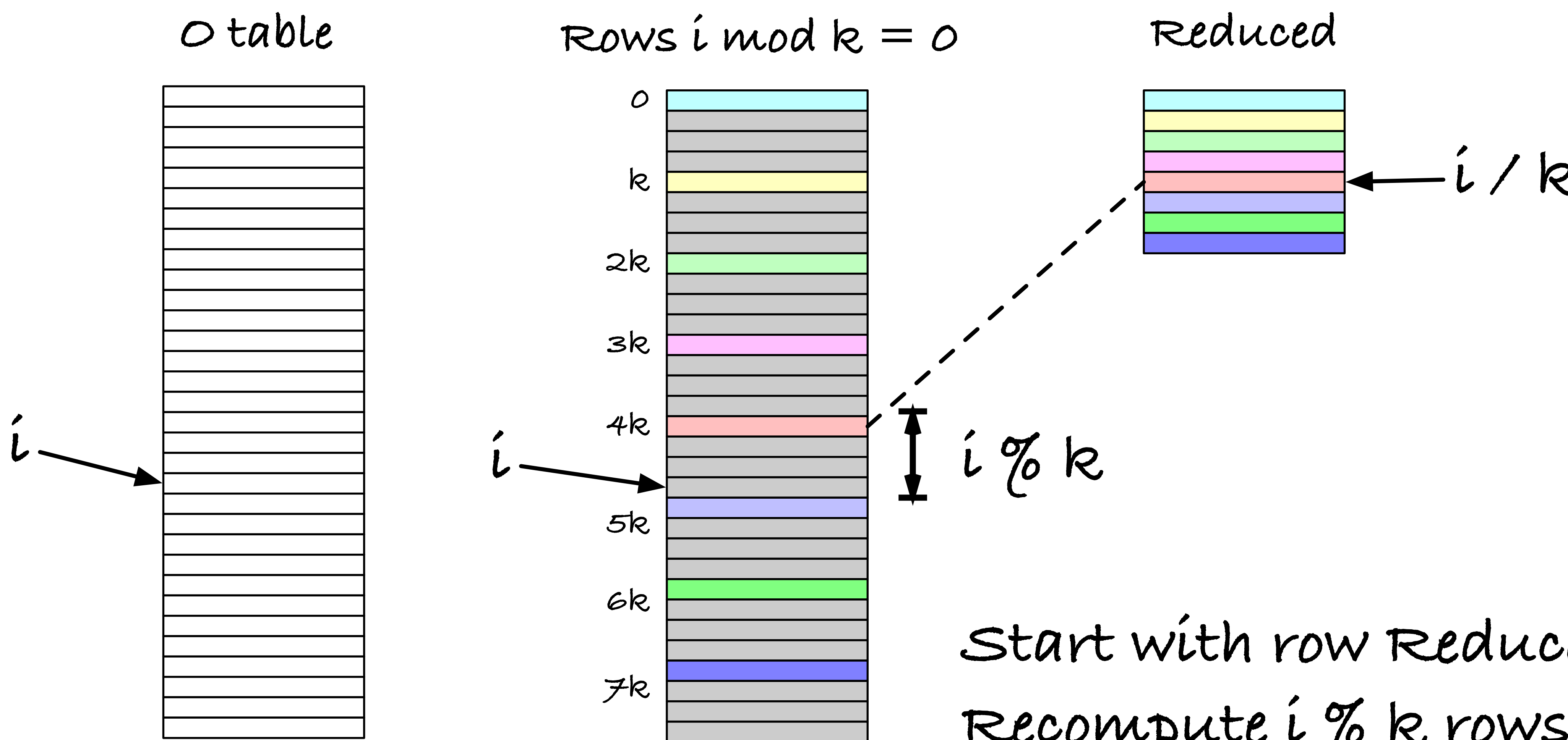
Reducing O (take 1)



$O(\sigma\, n/k \log n)$

$O(\sigma\, n \log n)$

Reducing O (take 1)



Start with row Reduced[i/k].

Recompute i % k rows.

Get your answer.

```python
class Reduced:
    """Assumes bwt is a numpy array.
    That enables the sum in __getitem__."""

    def __init__(self, bwt, σ, k):
        self.bwt = bwt
        self.k = k
        self.rows = np.empty((len(bwt)//k, σ), dtype='i')

        row = np.zeros(σ)
        for i, a in enumerate(bwt):
            if i % k == 0:  # only save every k’th row…
                self.rows[i//k] = row
            row[a] += 1

    def __getitem__(self, idx):
        i, a = idx
        count = self.rows[i//self.k, a]
        round_down = self.k * (i//self.k)
        count += sum(self.bwt[round_down:i] == a)
        return count
```

How much memory are we using?

Structure	Usage	Memory (bits)	Access cost
C table	Rotations	$O(\sigma \log n)$	
O table	Rotations	**$O(\sigma\, n/k \log n)$**	**$O(k\sigma)$**
bwt(x)	Recompute rows	**$O(n \log \sigma)$**	
RO table	Computing D	**$O(\sigma\, n/k \log n)$**	**$O(k\sigma)$**
bwt(rev(x))	Recompute rows	**$O(n \log \sigma)$**	
p	Need to know the pattern	$O(m \log \sigma)$	
D table	Bounding branches	$O(m \log m)$ / $O(m \log d)$	
SA	Reporting matches	**$O(n \log n)$**	

How much memory are we using?

We can do better for these tabes (later), but it is a general trick, so you should know it.

It is cache efficient (`incs` table and row will fit nicely). There are lookups in `bwt(x)` and `bwt(rev(x))`, but these are local.

ccess cost

og n)	
k log n)	O(kσ)
og σ)	
k log n)	O(kσ)
og σ)	
log σ)	
og m) / log d)	
og n)	

Hardwiring k, you can do with a single computed jump (for i%k) so you only have one branch.

```
switch (i % k) {
   case k - 1: row = row + …
   case k - 2: row = row + …
   …
   case 1: row = row + …
   case 0: break
}
```

No need for the loop-branching.

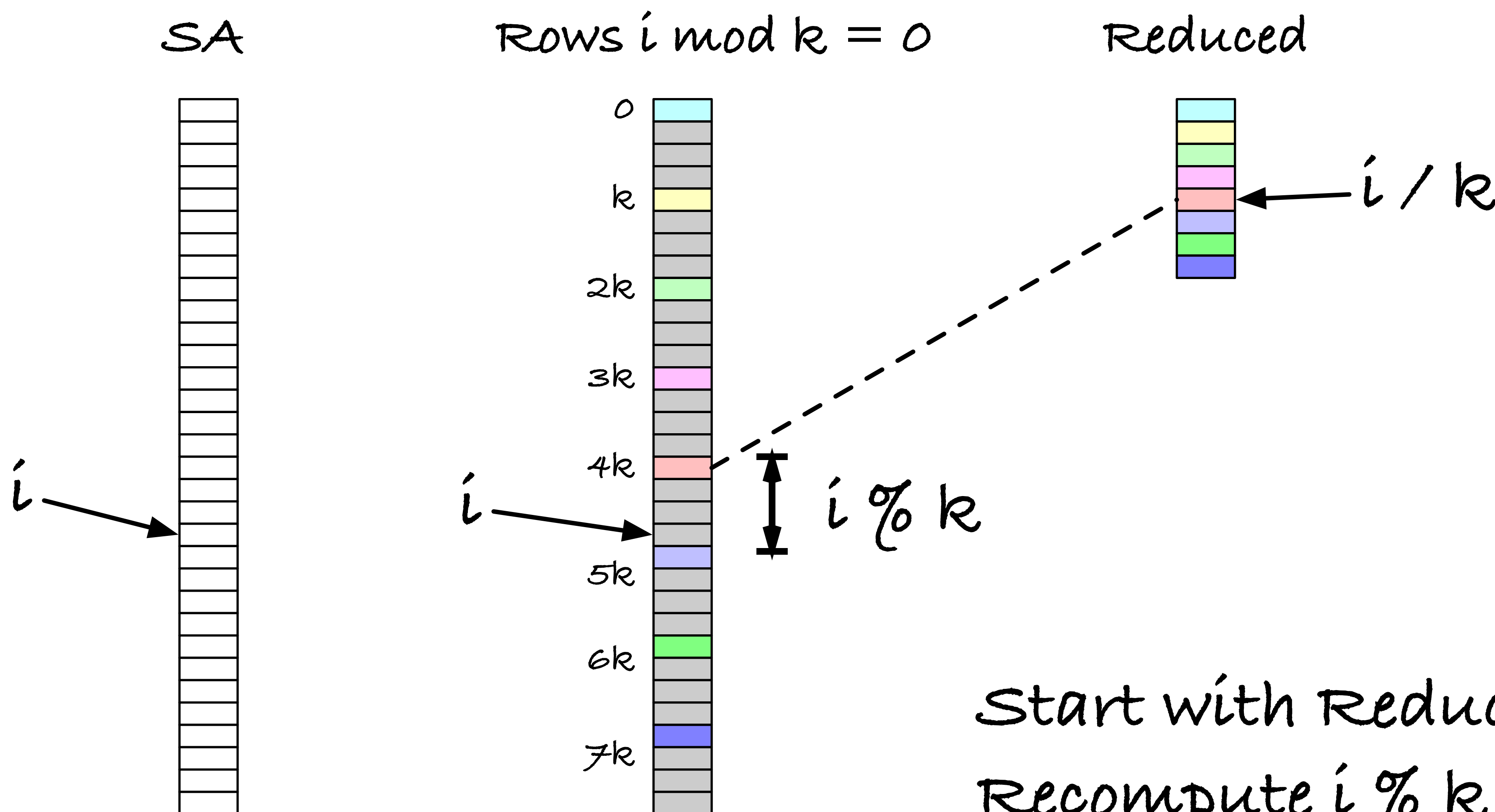
It can be often be implemented as a **very** efficient trick!

How much memory are we using?

Structure	Usage	Memory (bits)	Access cost
C table	Rotations	O(σ log n)	
O table	Rotations	**O(σ n/k log n)**	**O(kσ)**
bwt(x)	Recompute rows	**O(n log σ)**	
RO table	Computing D	**O(σ n/k log n)**	**O(kσ)**
bwt(rev(x))	Recompute rows	**O(n log σ)**	
p	Need to know the pattern	O(m log σ)	
D ta		O(m log m) /	
SA	Reporting matches	**O(n log n)**	

Can we use the same trick for the suffix array?

Reducing the suffix array

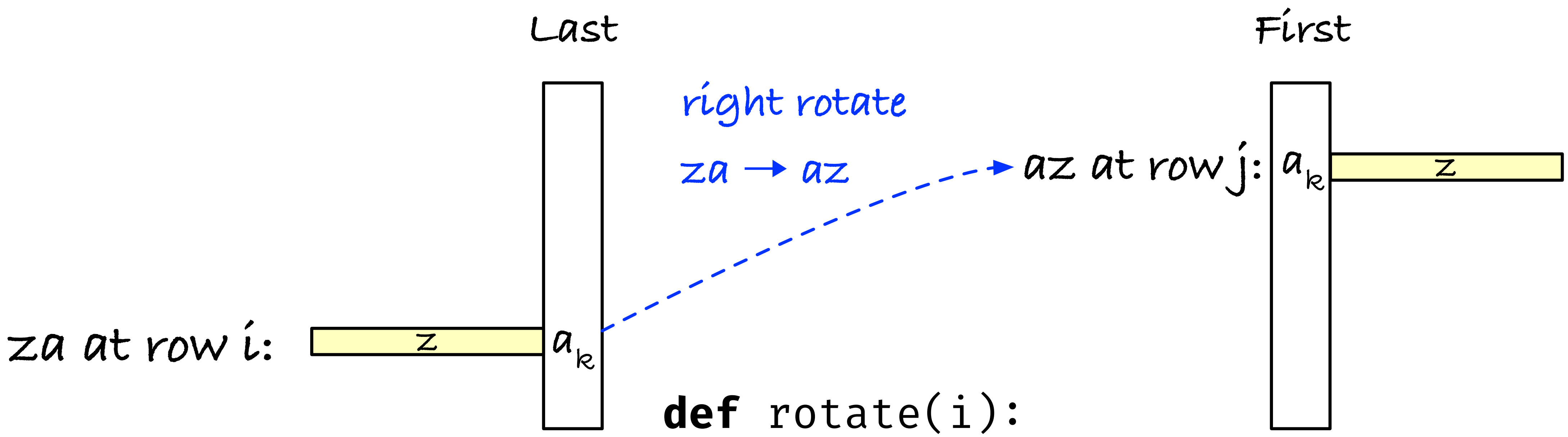


Start with Reduced[i/k].

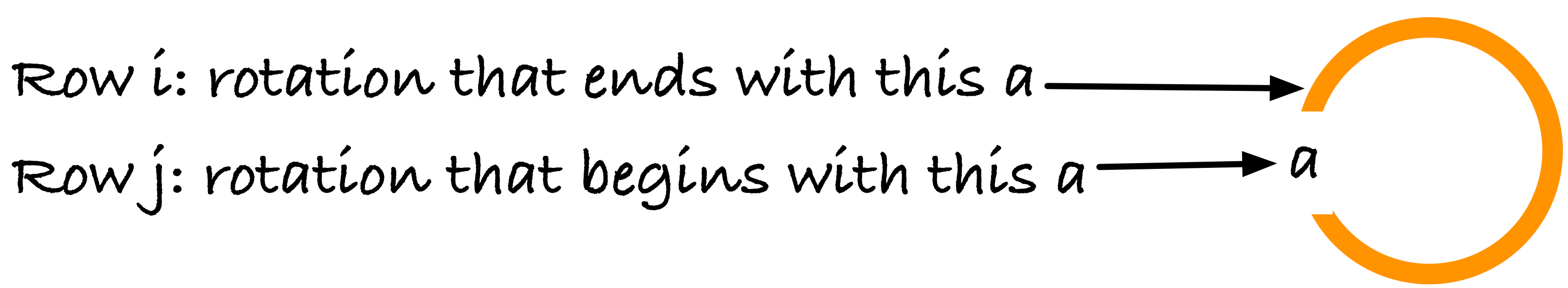
Recompute i % k entries.

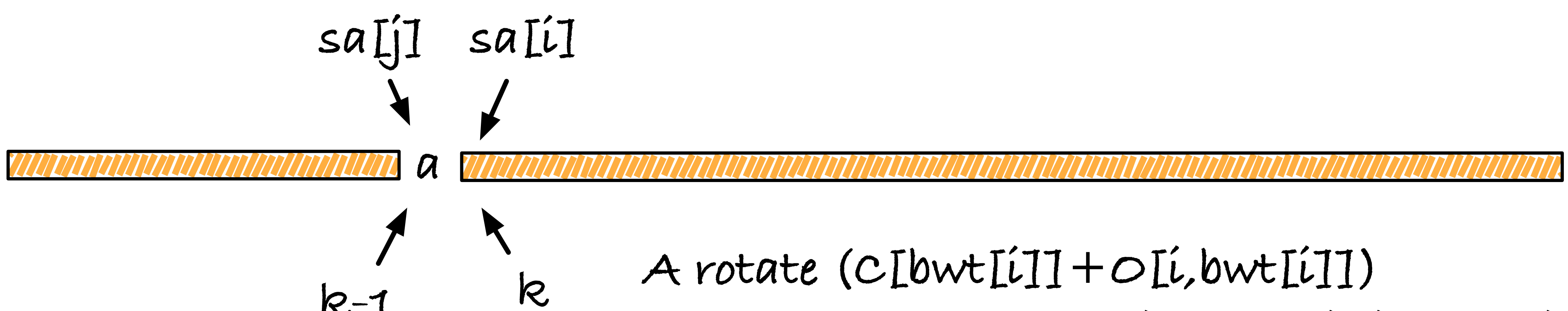
HOW DO YOU DO THAT????

Help from the bwt reversal….




```
def rotate(i):
    return C[bwtx[i]] + O[i, bwtx[i]]
```

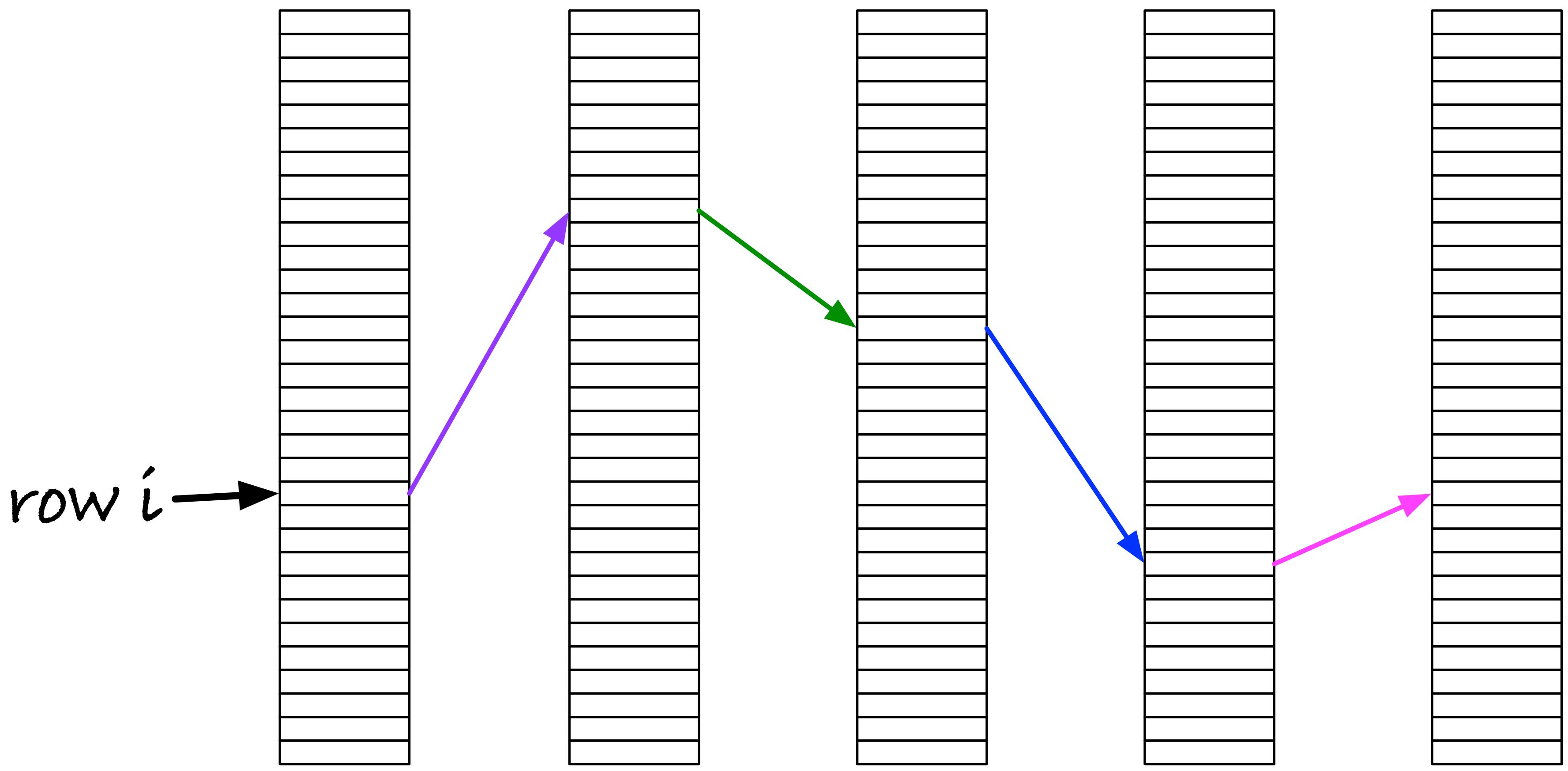




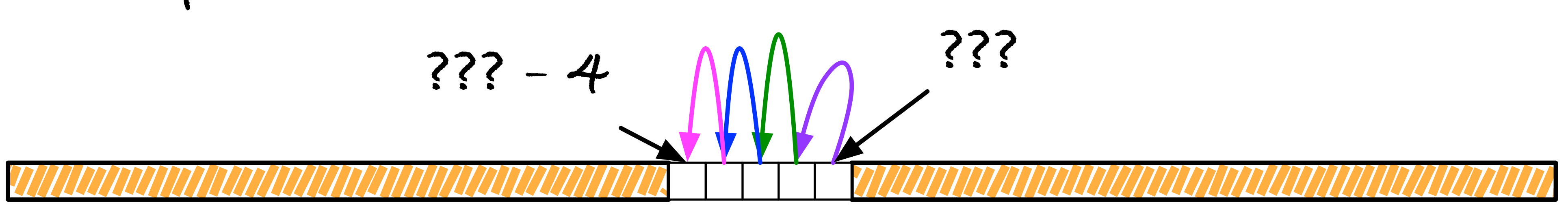
A rotate (C[bwt[i]]+O[i,bwt[i]])
moves us one step left in the original string.

Rotating left in x

Rotate in rows…

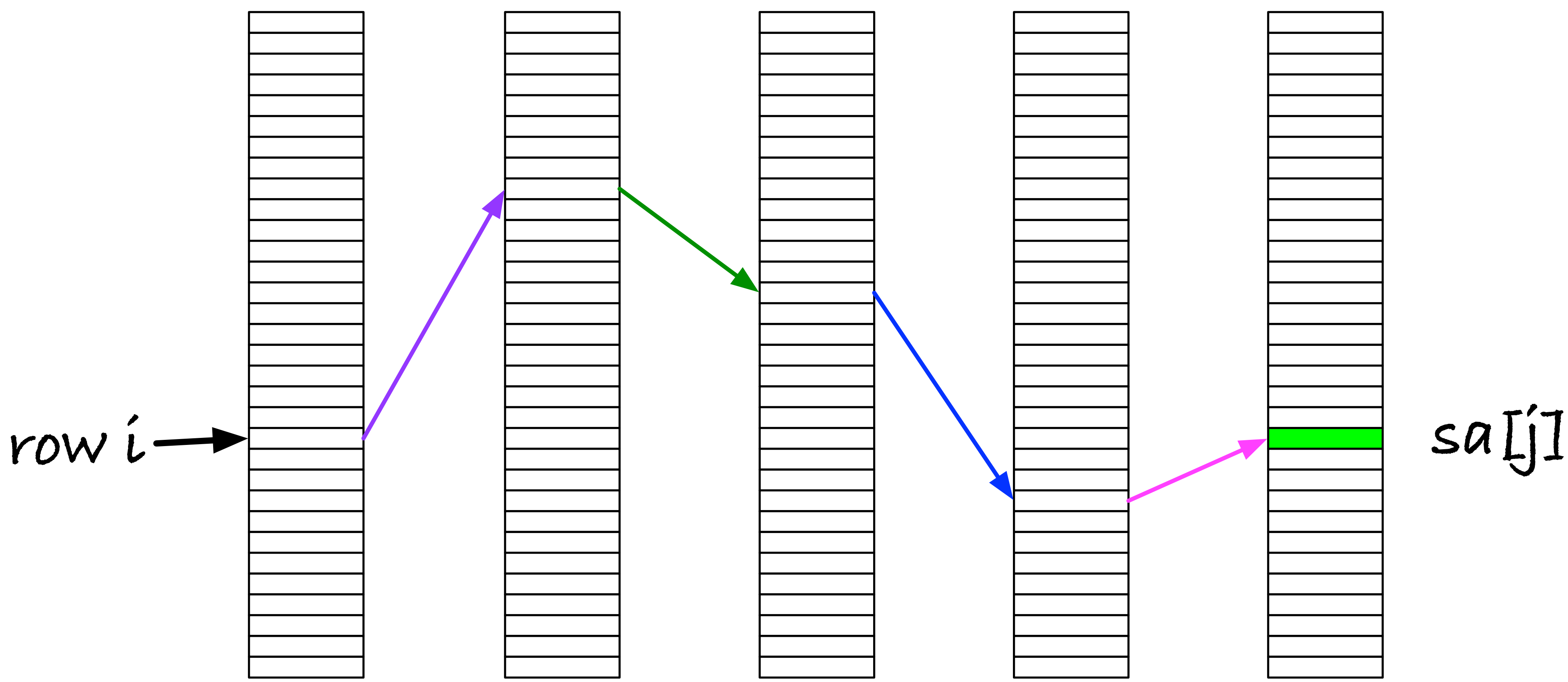


…move left in x.

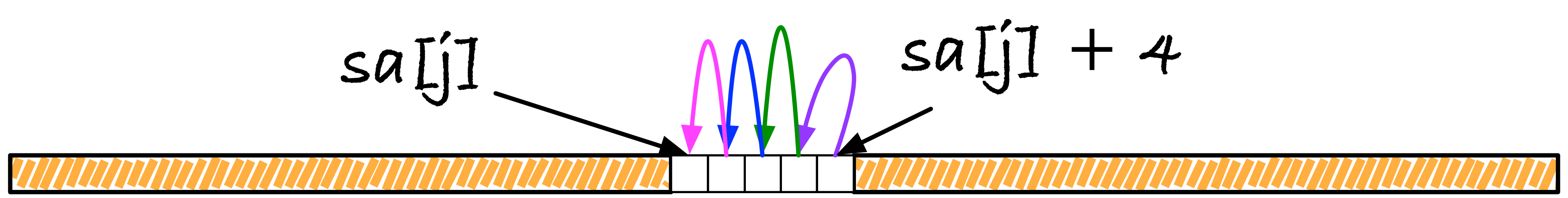


Rotating left in x

If we find a row with known sa[j] value...



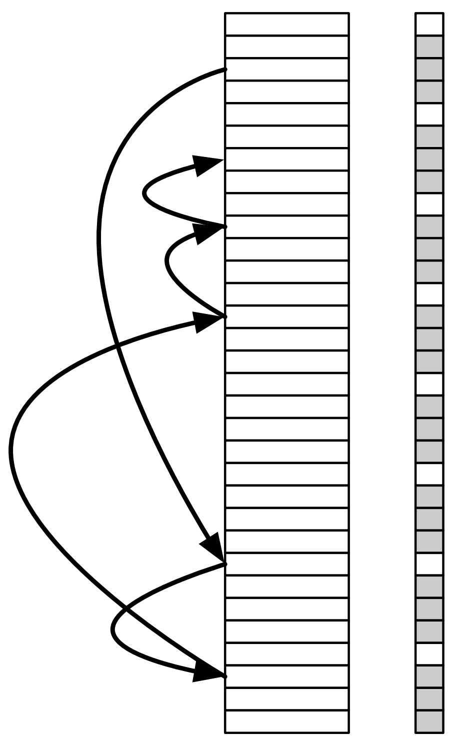
...we know the x index of the original row!



Reducing the suffix array

Idea 1:

Store the sa value for every k'th row…

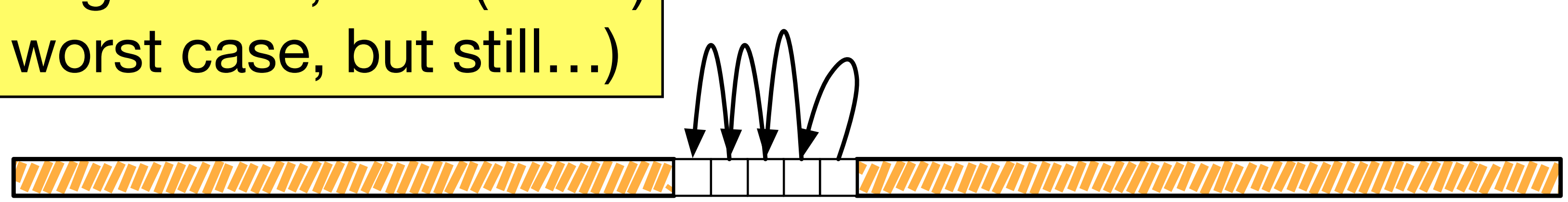


```
jumps = 0
while i % k ≠ 0:
    i = rotate(i)
    jumps += 1
idx = sa[i//k] + jumps
```

When we move left, we will eventually see one,
but how many jumps do we need?

There are n-n/k missing values, so O(n-n/k)
(admittedly unlikely worst case, but still…)

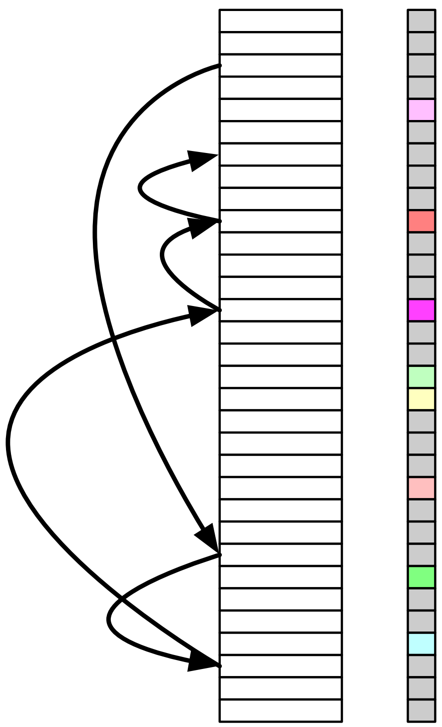
???

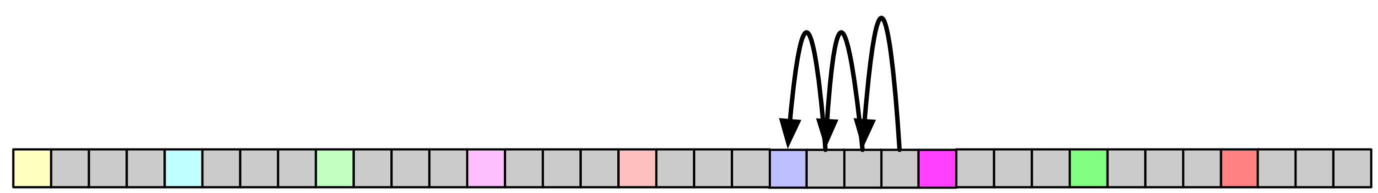


Reducing the suffix array

Idea 2:

Store the sa value for every k'th index in x





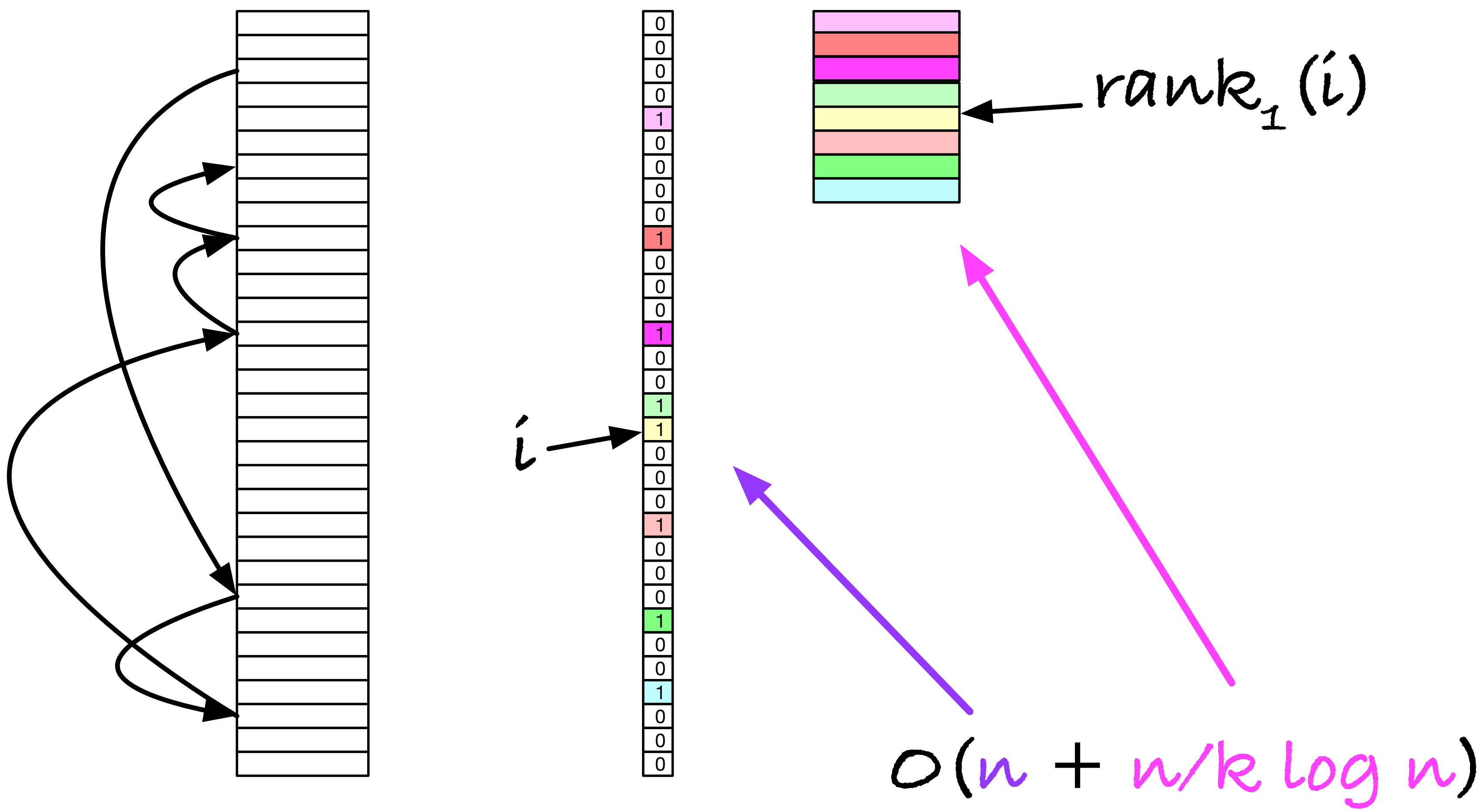

```
jumps = 0
while i % k ≠ 0:
    i = rotate(i)
    jumps += 1
idx = sa[i//k] + jumps
```

Dividing by k doesn't tell us where the defined values are anymore!

We cannot leave entries undefined without using the same memory as before!

Reducing the suffix array

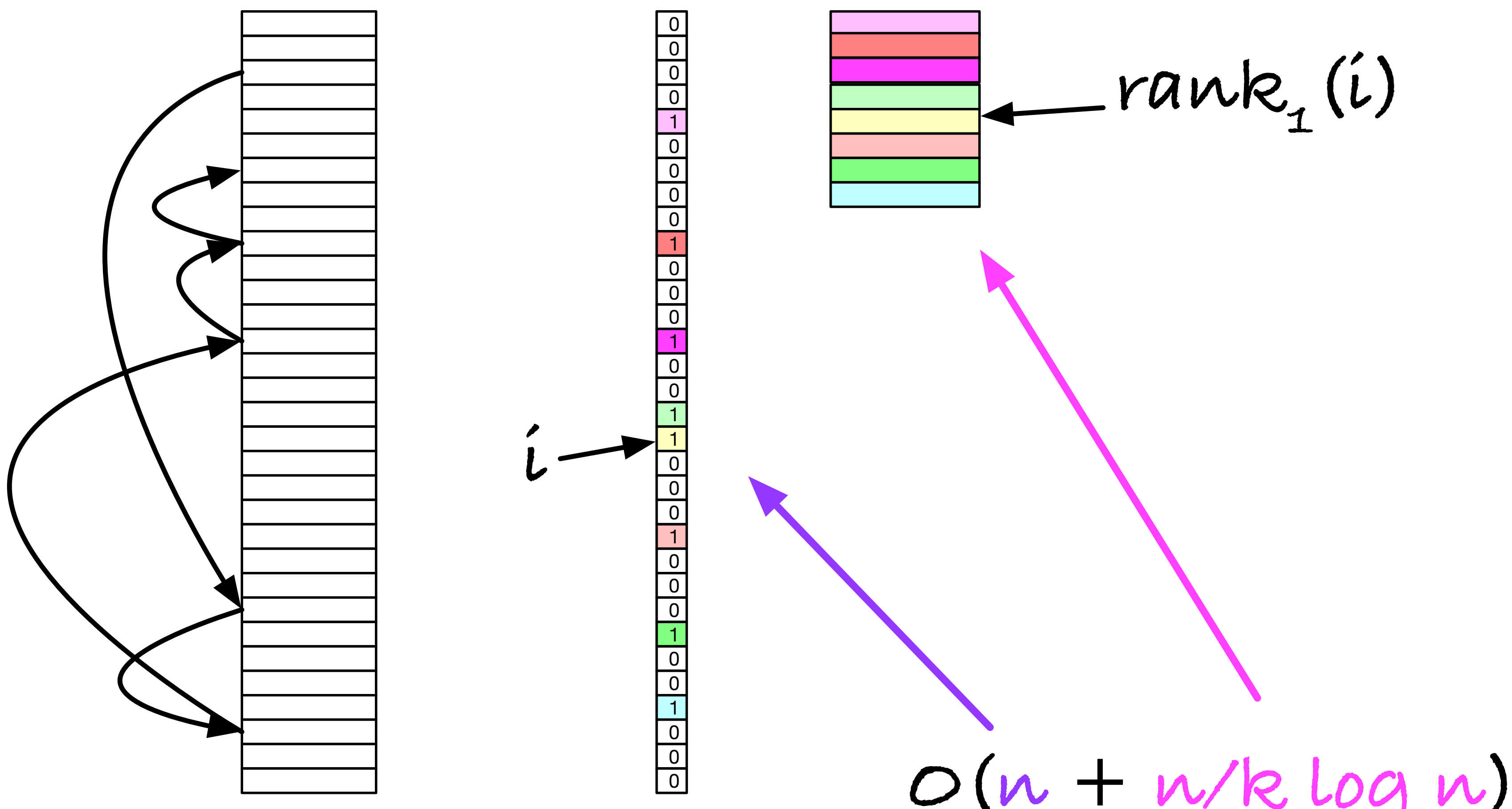
Bitvectors to the rescue!




```
jumps = 0
while not bv[i]:
    i = rotate(i)
    jumps += 1
idx = sa[bv.rank1(i)] + jumps
```

Reducing the suffix array

Bitvectors to the rescue!



Warning: Can you wrap around here, if you jump past the row that starts with $?

```
(sa[bv.rank1()] + jump) % n
```

```
x[0] is stored! We won't wrap!
```

```
jumps = 0
while not bv[i]:
    i = rotate(i)
    jumps += 1
idx = sa[bv.rank1(i)] + jumps
```

How much memory are we using?

Structure	**Usage**	**Memory (bits)**	**Access cost**
C table	Rotations	$O(\sigma \log n)$	
O table	Rotations	$O(\sigma\, n/k \log n)$	$O(k\sigma)$
bwt(x)	Recompute rows	$O(n \log \sigma)$	
RO table	Computing D	$O(\sigma\, n/k \log n)$	$O(k\sigma)$
bwt(rev(x))	Recompute rows	$O(n \log \sigma)$	
p	Need to know the pattern	$O(m \log \sigma)$	
D table	Bounding branches	$O(m \log m)$ / $O(m \log d)$	
SA	Reporting matches	$O(n + n/k' \log n)$	$O(k'\, k\, \sigma)$

How much memory are we using?

Now we replace these with something better.

Structure	Usage	Memory (bits)	Access cost
		…og n)	
O table	Rotations	**$O(\sigma\ n/k \log n)$**	**$O(k\sigma)$**
bwt(x)	Recompute rows	**$O(n \log \sigma)$**	
RO table	Computing D	**$O(\sigma\ n/k \log n)$**	**$O(k\sigma)$**
bwt(rev(x))	Recompute rows	**$O(n \log \sigma)$**	
p	Need to know the pattern	$O(m \log \sigma)$	
D table	Bounding branches	O(m lo… $O(m \log d)$	
SA	Reporting matches	**$O(n + n/k' \log n)$**	**$O(k'\ k\ \sigma)$**

And we reduce this to $O(k' \log \sigma)$.

Strings with rank

- We only have the O and RO tables for rank queries on `bwt(x)` and `bwt(rev(x))` respectively.
- If we had a text representation that gave us efficient `rank(a,i)` queries, then we wouldn't need the tables.
- Can we get that?

String representation

Normal encoding:

```
bwt(x) = A   A   C   G   T   A   C   A   $   G   A   C
         001 001 010 011 100 001 010 001 000 011 001 010
```

```
$: 000
A: 001
C: 010
G: 011
T: 100
```

$\sigma = 5$, $\log \sigma = 3$ bits

(in practise 8 bits)

Fixed sized characters, one after another, are easy and fast to index (usually, depends on how characters fit in word sizes, but usually).

But we don't know how to do `rank(a, i)` [or we wouldn't need the O/RO tables].

String representation

"One-hot" encoding:

bwt(x) = A A C G T A C A $ G A C

	A	A	C	G	T	A	C	A	$	G	A	C
$bv_\$$	0	0	0	0	0	0	0	0	1	0	0	0
bv_A	1	1	0	0	0	1	0	1	0	0	1	0
bv_C	0	0	1	0	0	0	1	0	0	0	0	1
bv_G	0	0	0	1	0	0	0	0	0	1	0	0
bv_T	0	0	0	0	1	0	0	0	0	0	0	0

$: 10000
A: 01000
C: 00100
G: 00010
T: 00001

$\sigma = 5$, 5 bits

$rank(a, i) = bv_a.rank_1(i)$

String rank is now bit vector rank!
Rank in $O(1)$

access(i) — locate bv with 1

You need to look at the bit vectors to work out which has the one.
Rank in $O(\sigma)$

A `rotate(i)` for the reduced SA is one `access(i)` and one `rank(a,i)`, so: $O(\sigma)$

How much memory are we using?

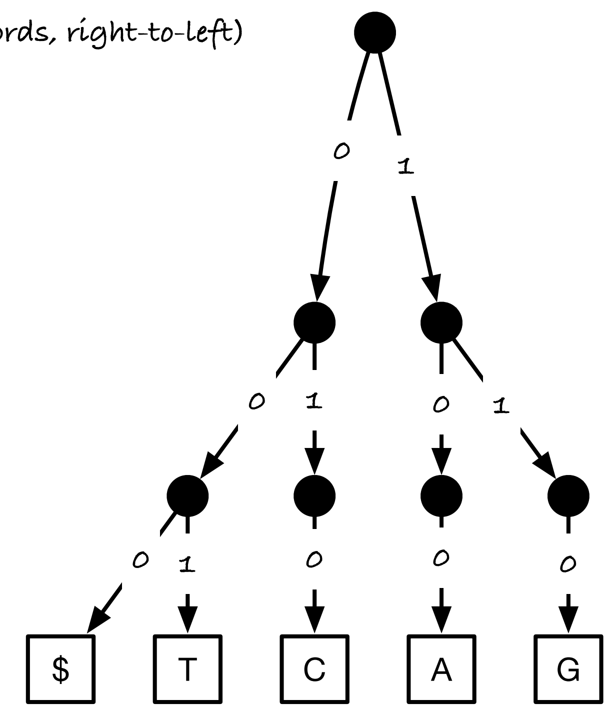
Structure	**Usage**	**Memory (bits)**	**Access cost**
C table	Rotations	O(σ log n)	
bwt(x)	Rank and rotations	**O(n σ)**	**O(σ) access** **O(1) rank**
bwt(rev(x))	Rank and rotations	**O(n σ)**	**O(σ) access** **O(1) rank**
p	Need to know the pattern	O(m log σ)	
D table	Bounding branches	O(m log m) / O(m log d)	
SA	Reporting matches	**O(n + n/k’ log n)**	**O(k’ σ)**

Alphabet encoding

"Normal" encoding

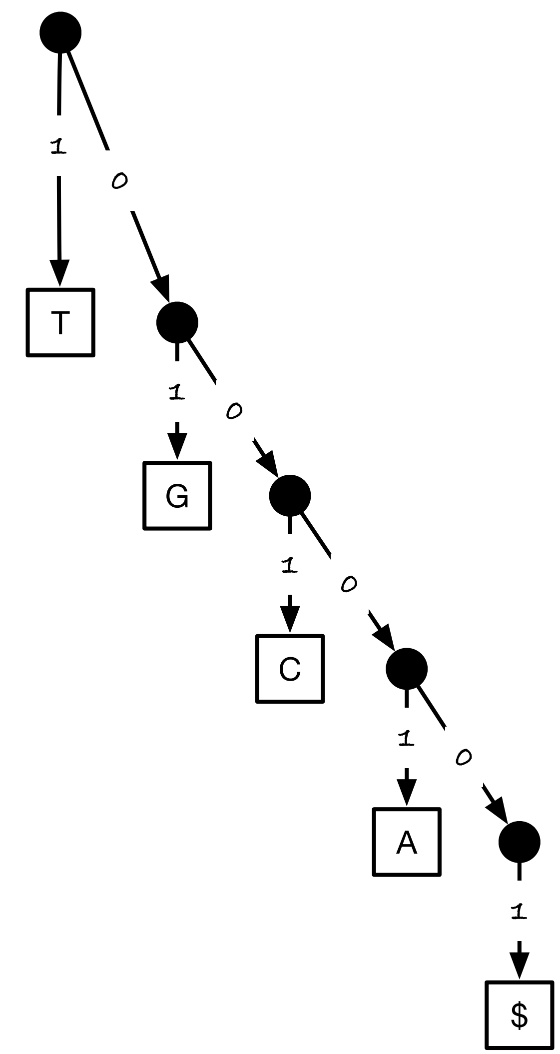
(read like binary words, right-to-left)

```
$: 000
A: 001
C: 010
G: 011
T: 100
```



"One-not" encoding

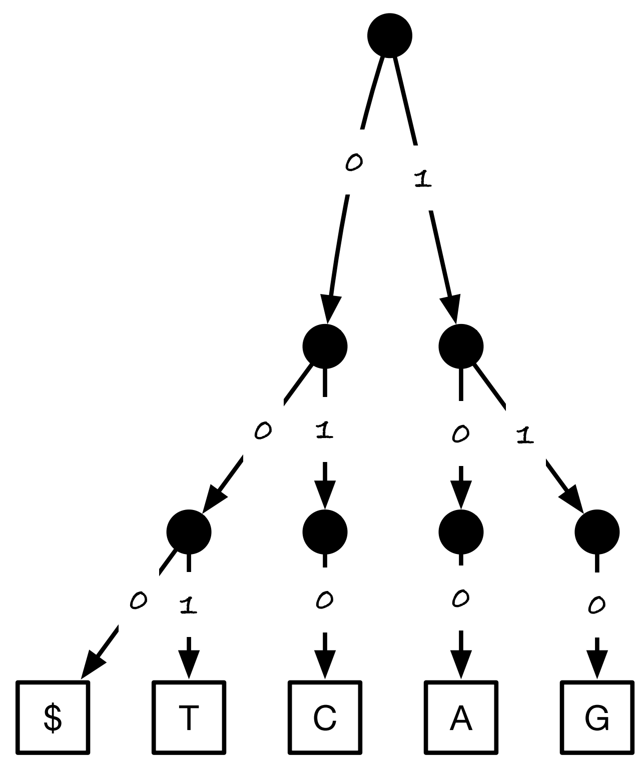
```
$: 10000
A: 01000
C: 00100
G: 00010
T: 00001
```



Alphabet encoding

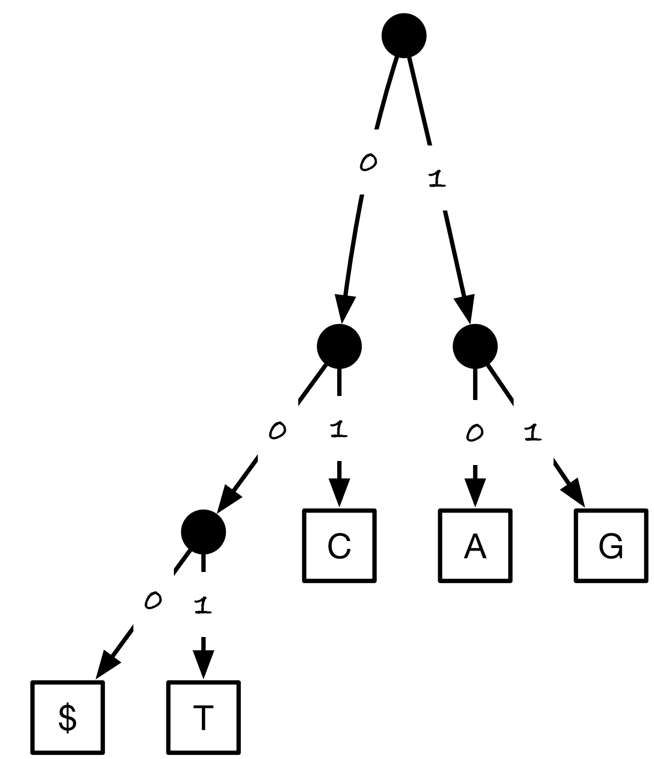
"Normal" encoding

```
$: 000
A: 001
C: 010
G: 011
T: 100
```



Variable number of bits

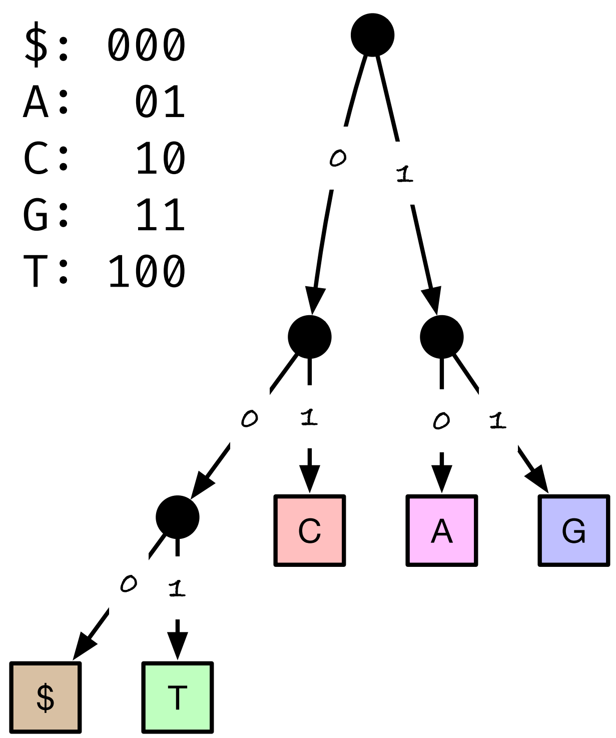
```
$: 000
A:  01
C:  10
G:  11
T: 100
```



Reduces number of bits, but difficult to index...

Alphabet encoding

$: 000
A: 01
C: 10
G: 11
T: 100



bwt(x) = A A C G T A C A $ G A C

01 01 10 11 100 01 10 01 000 11 01 10

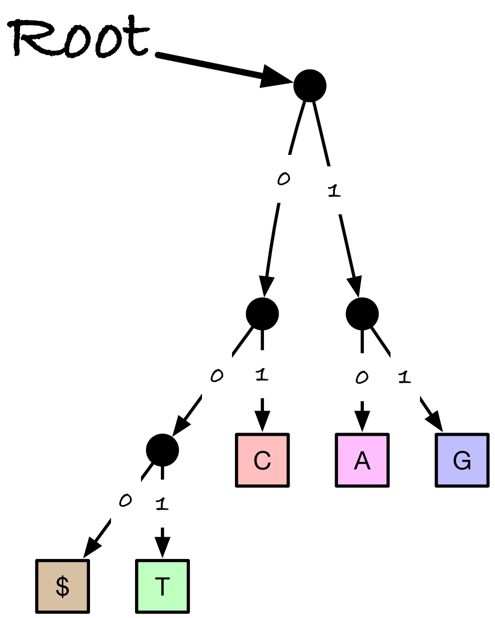
Hard to index (where does a letter start?)
Doesn't help us with rank.

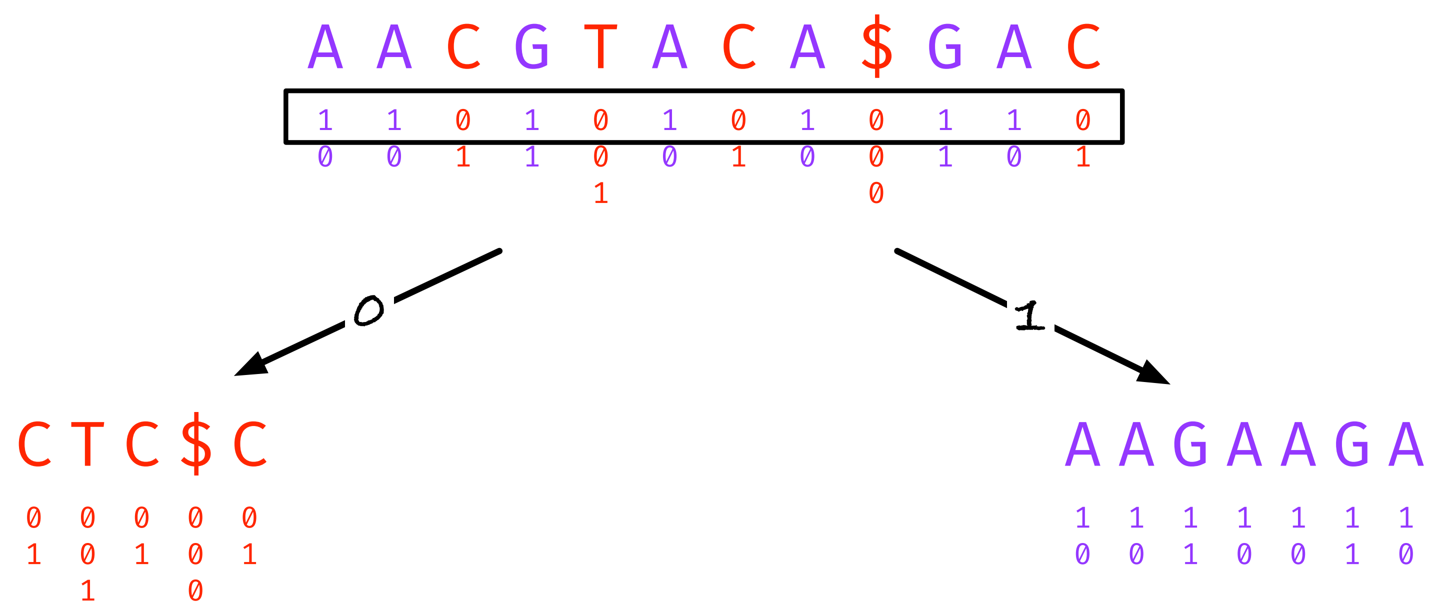
bwt(x) = A A C G T A C A $ G A C

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

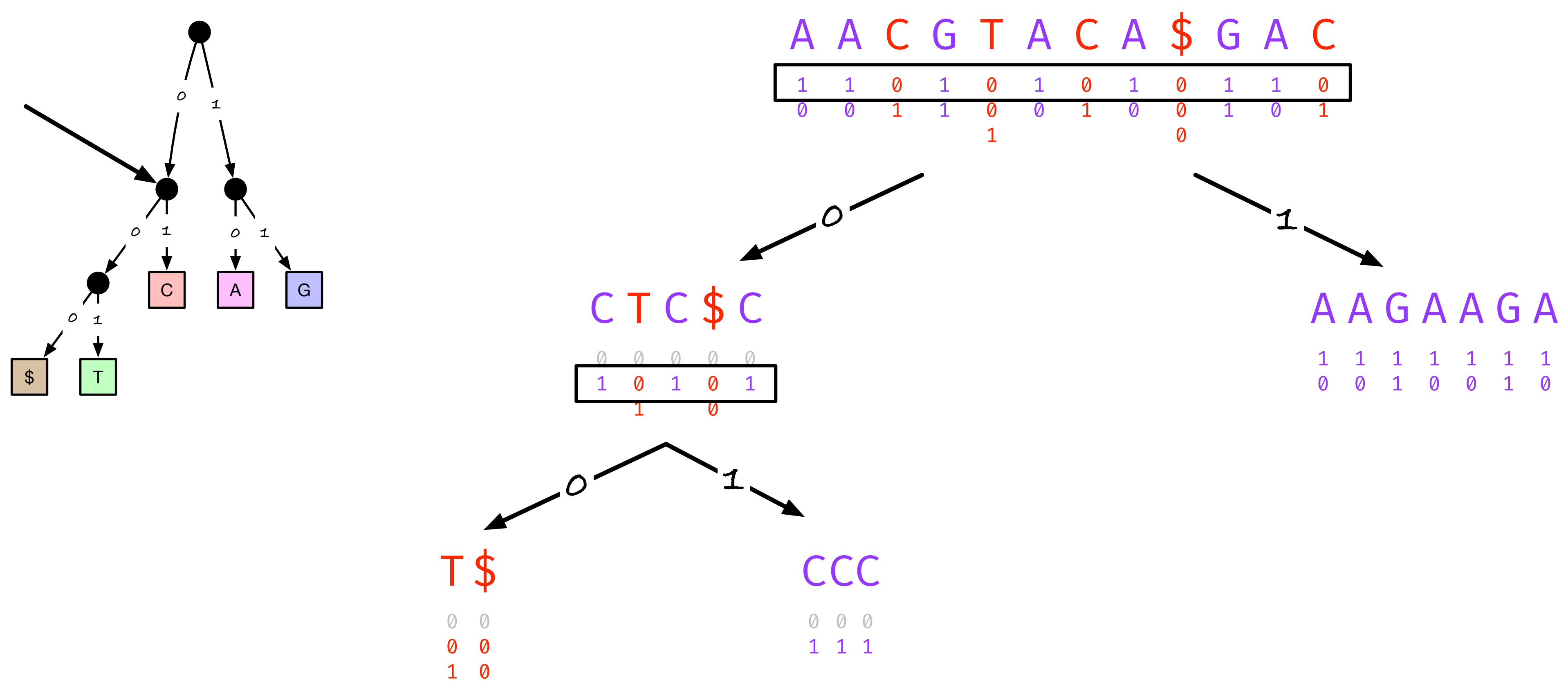
We don't know how to represent these jagged columns,
but if we could we could access(i) efficiently.
Does it help us with rank?

Wavelet trees

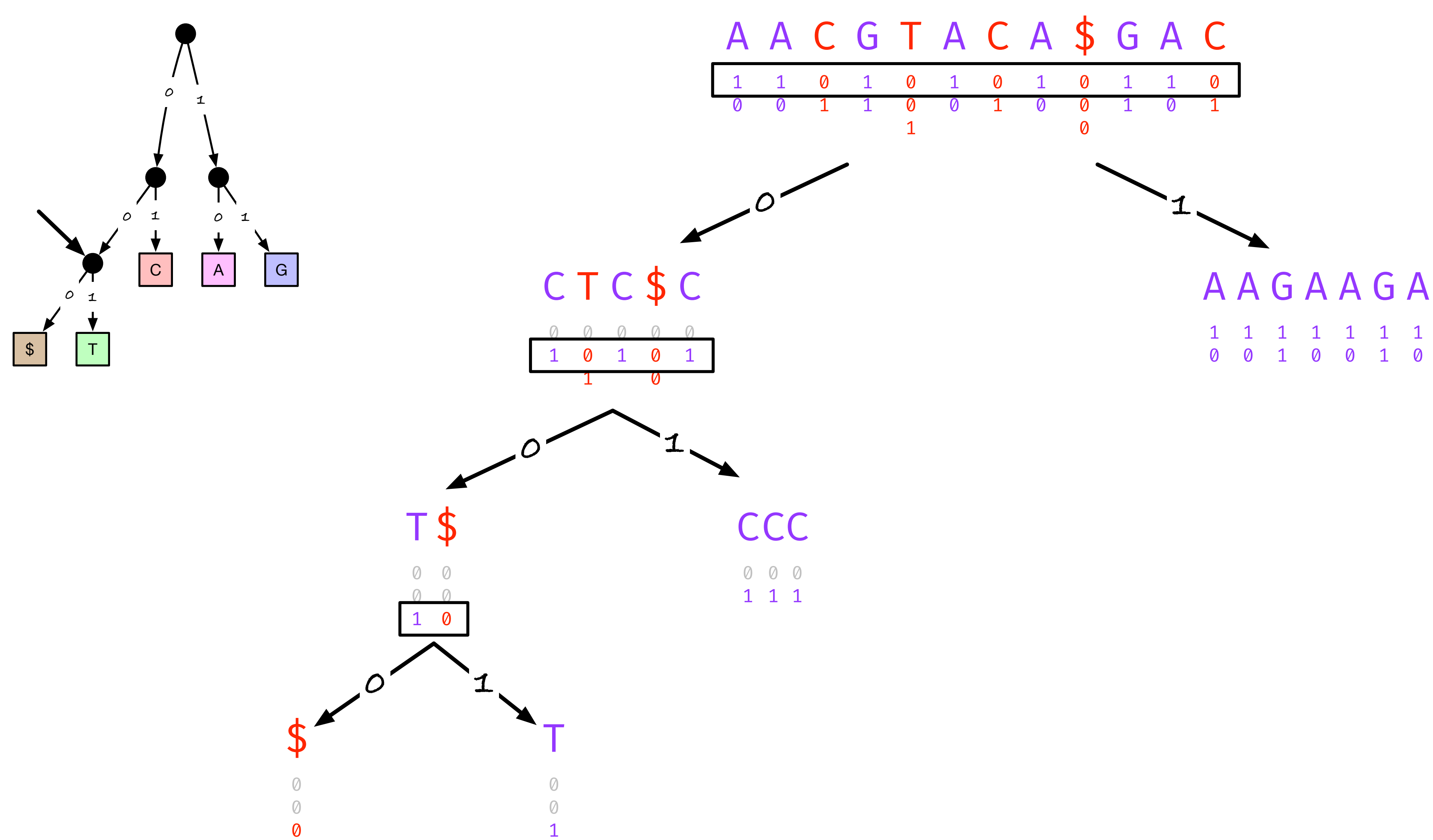




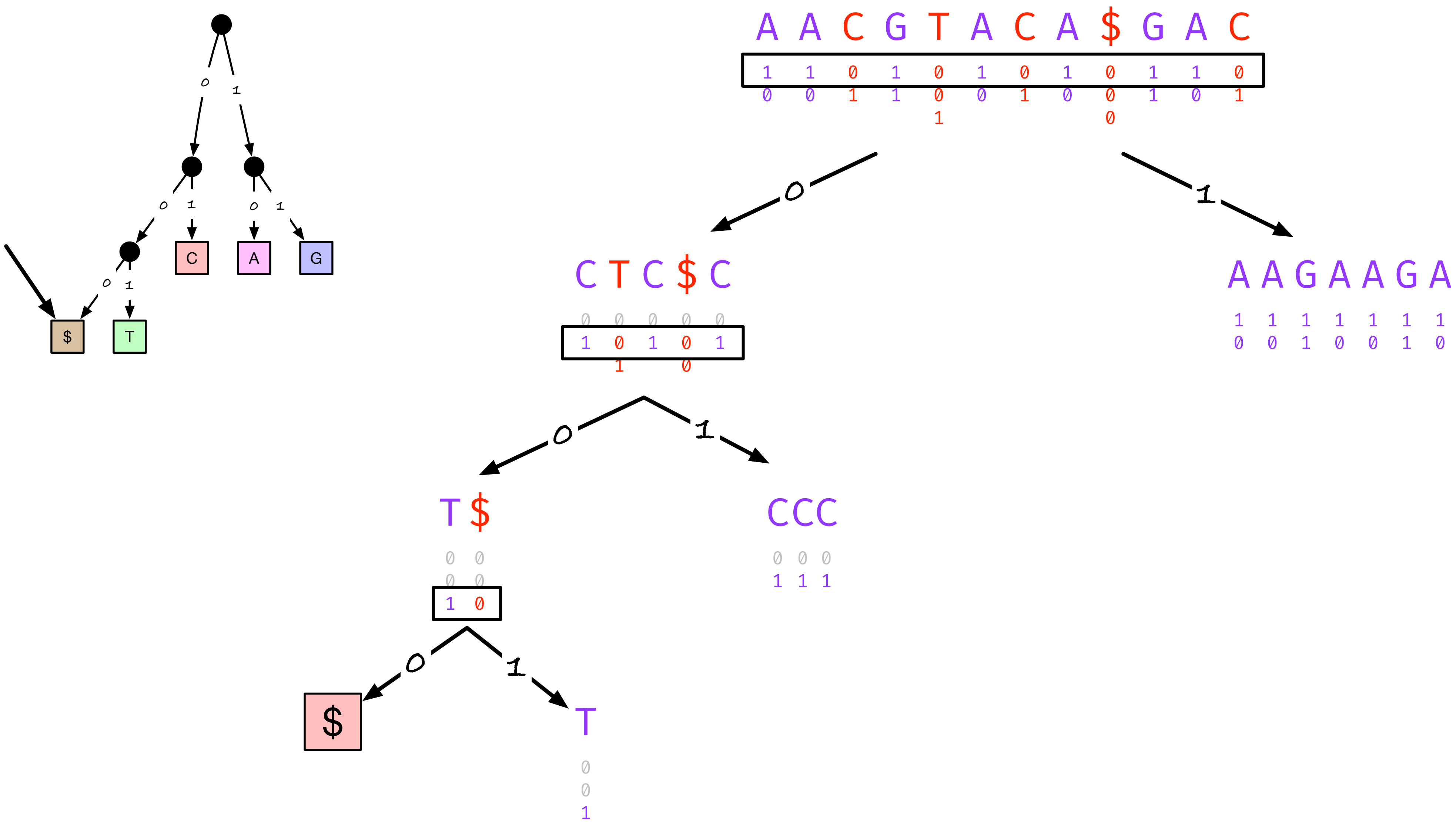
Wavelet trees



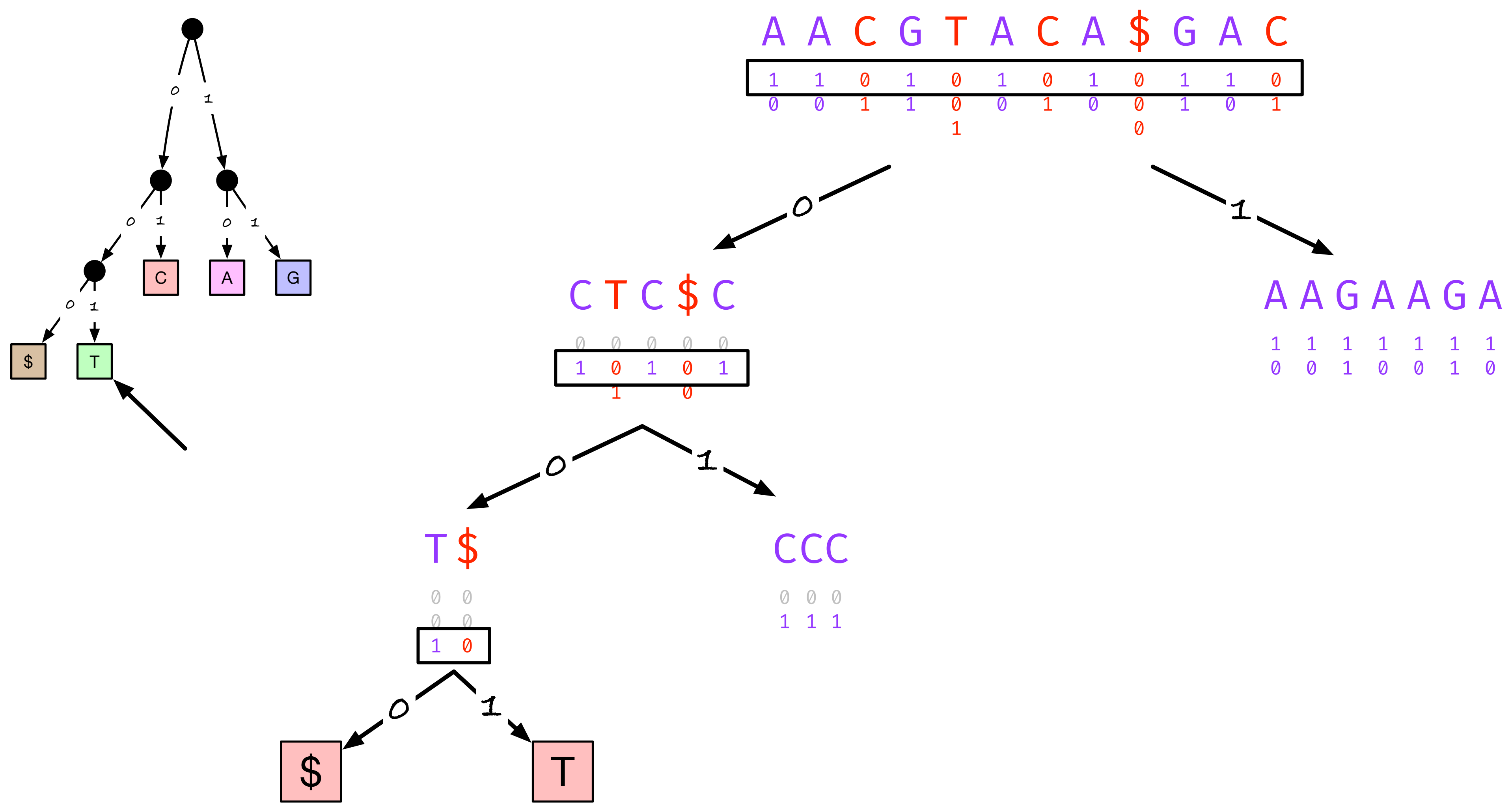
Wavelet trees



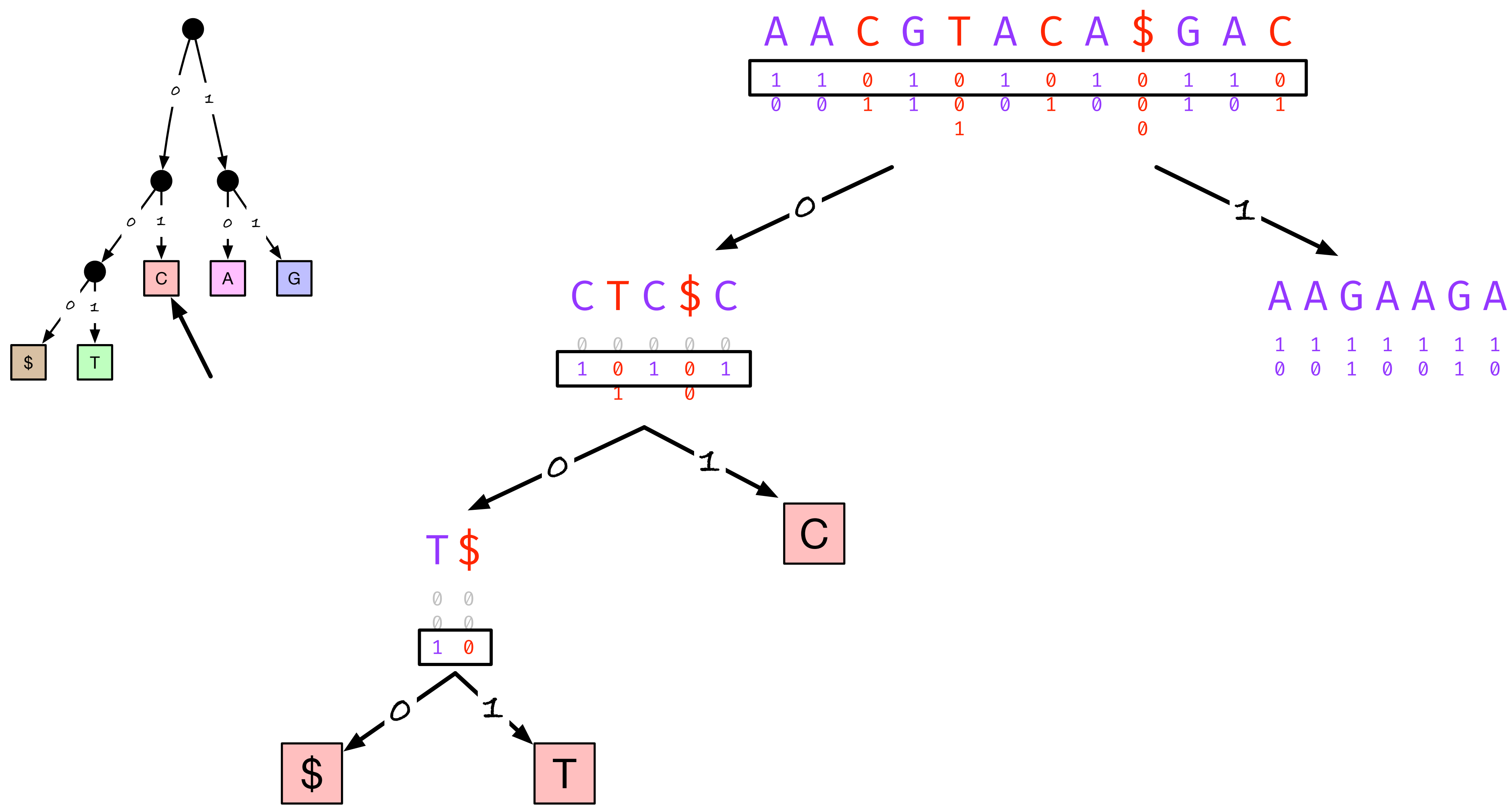
Wavelet trees



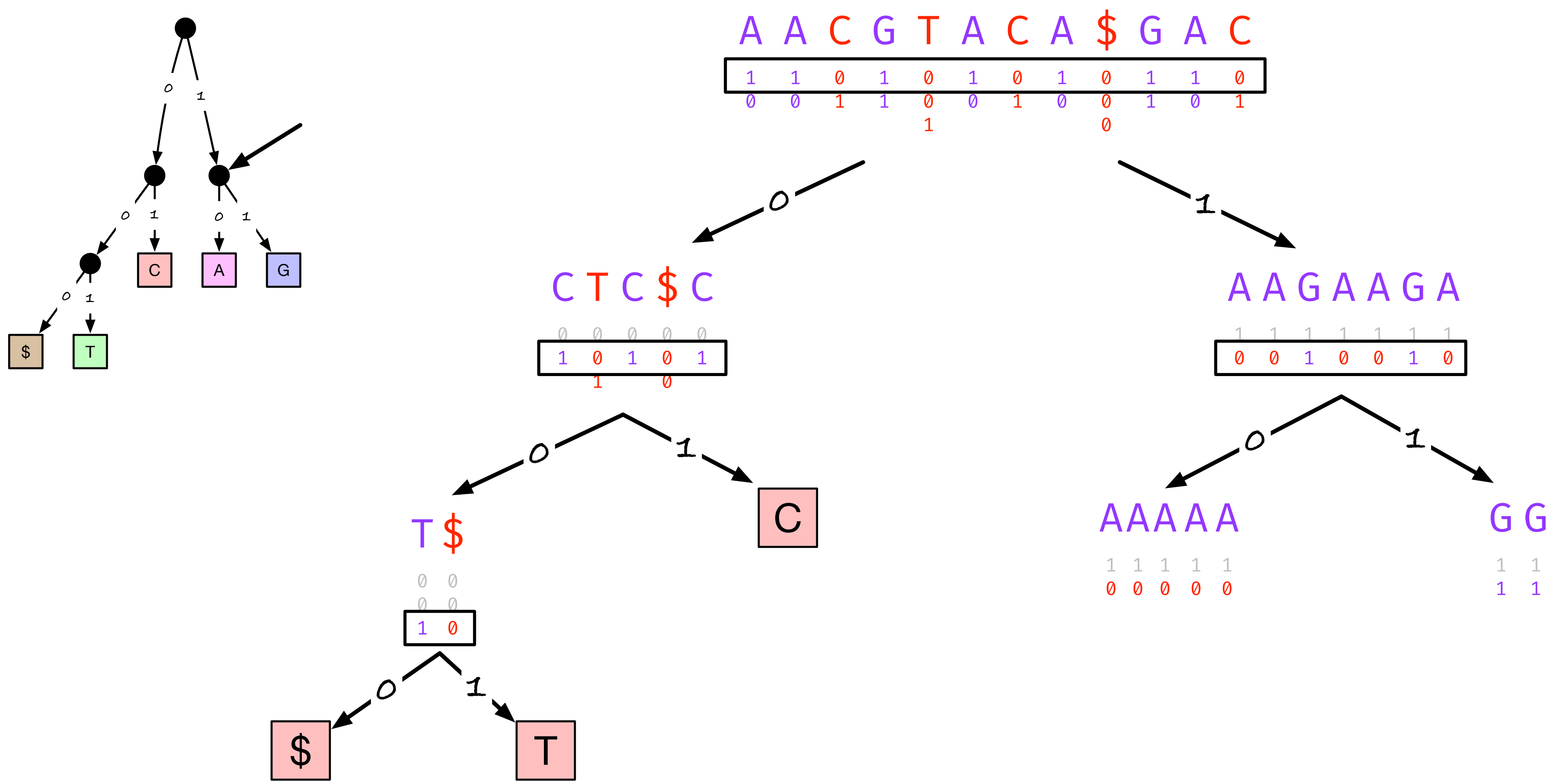
Wavelet trees



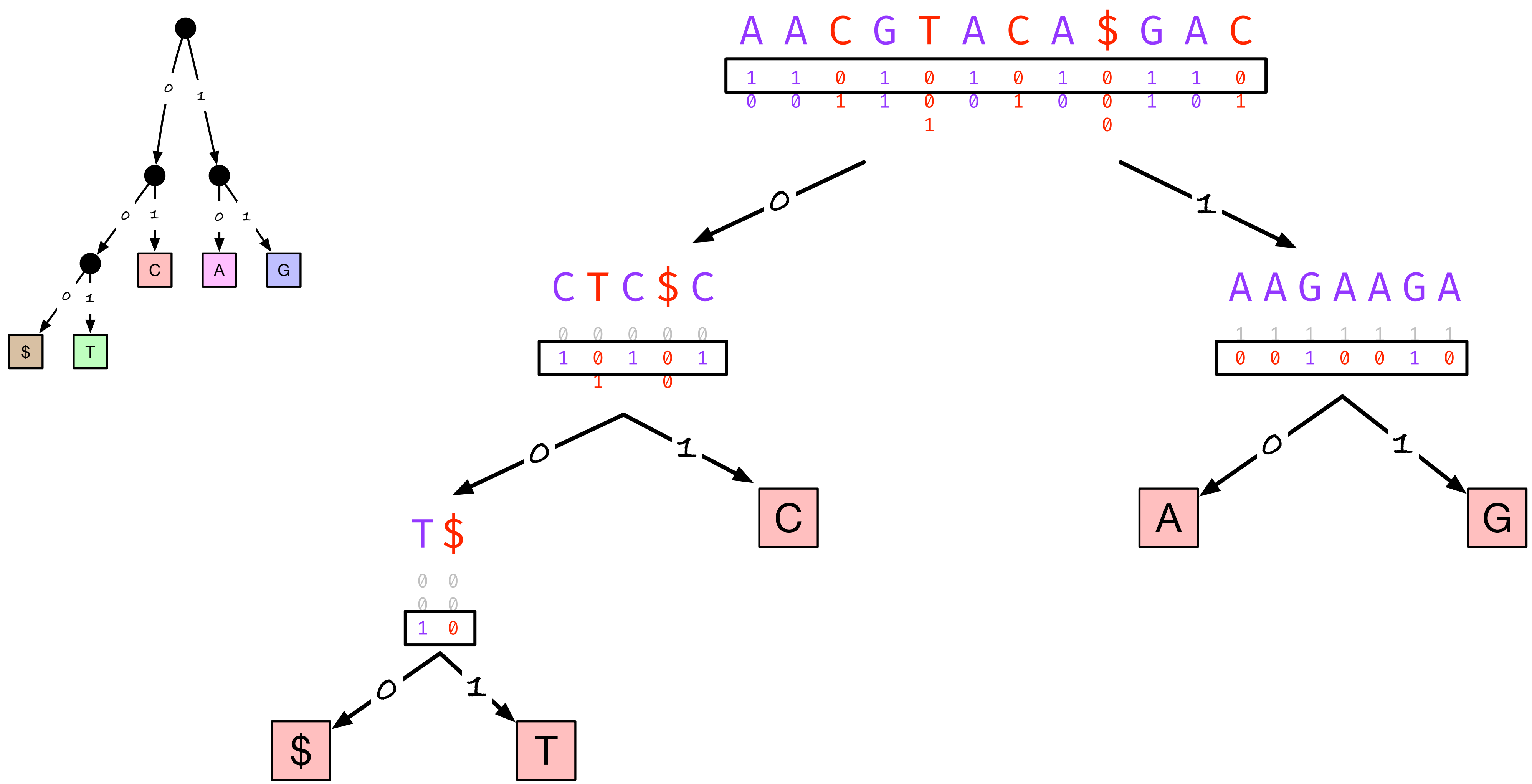
Wavelet trees



Wavelet trees

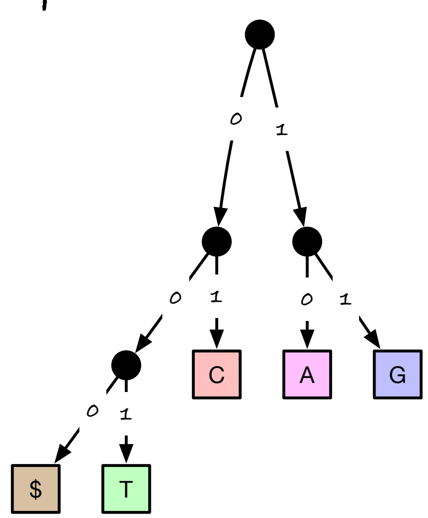


Wavelet trees



Wavelet trees

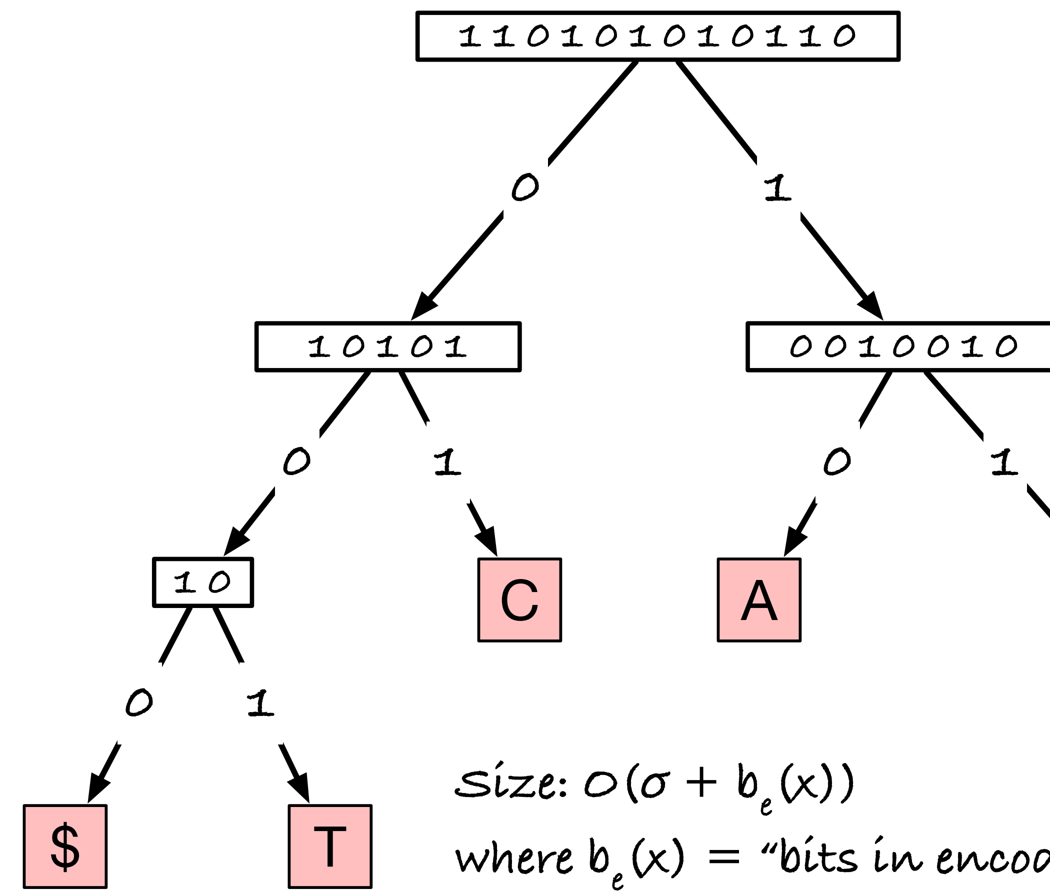
Alphabet:



String:

bwt(x) =	A	A	C	G	T	A	C	A	$	G	A	C
	1	1	0	1	0	1	0	1	0	1	1	0
	0	0	1	1	0	0	1	0	0	1	0	1
					1				0			

Wavelet tree:



Size: $O(\sigma + b_e(x))$

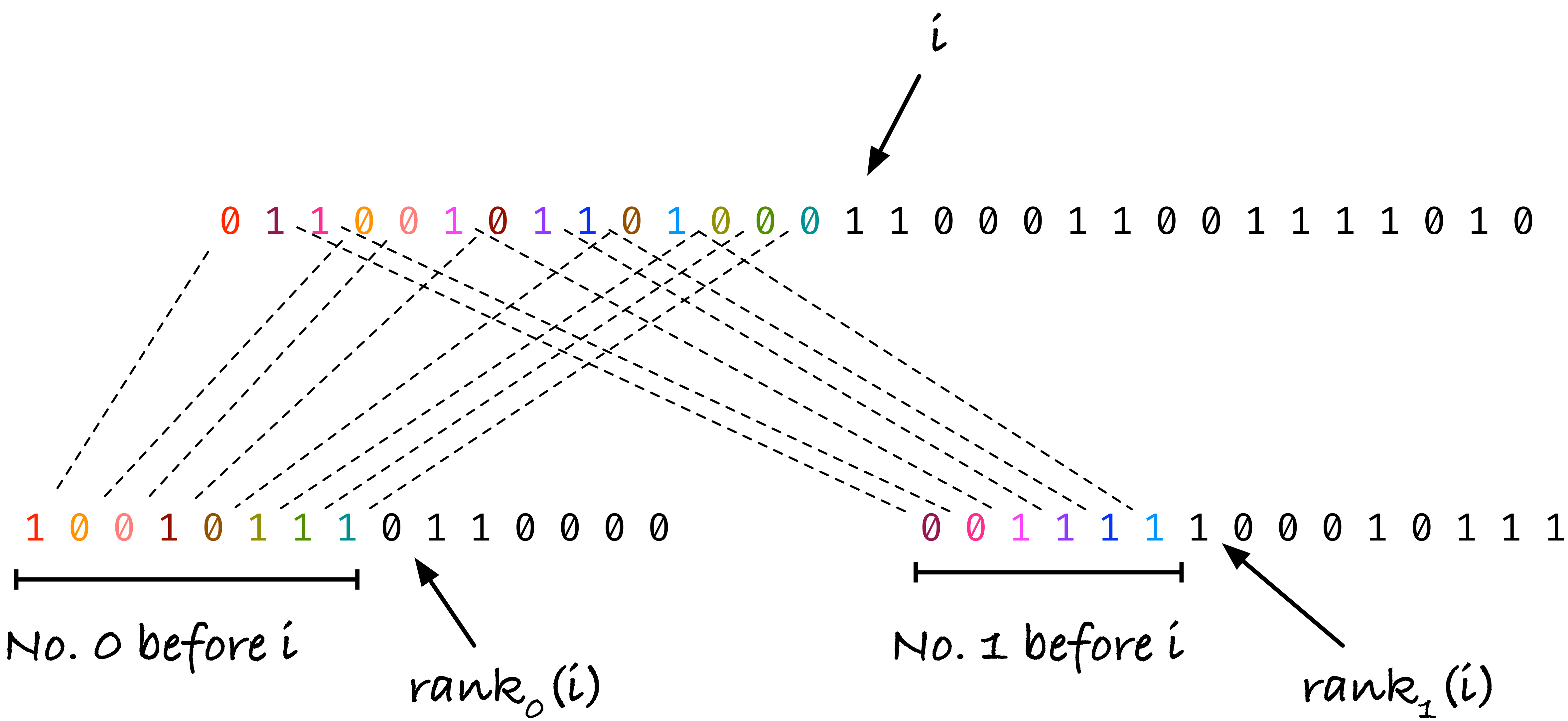
where $b_e(x)$ = "bits in encoding of x"

$n \leqq b_e(x) \leqq \sigma n$

(let's say $b_e(x) = n \log \sigma$)

If we split the strings recursively as shown, then we can construct wavelet trees in $O(\sigma + b_e(x))$

Wavelet trees



Wavelet trees

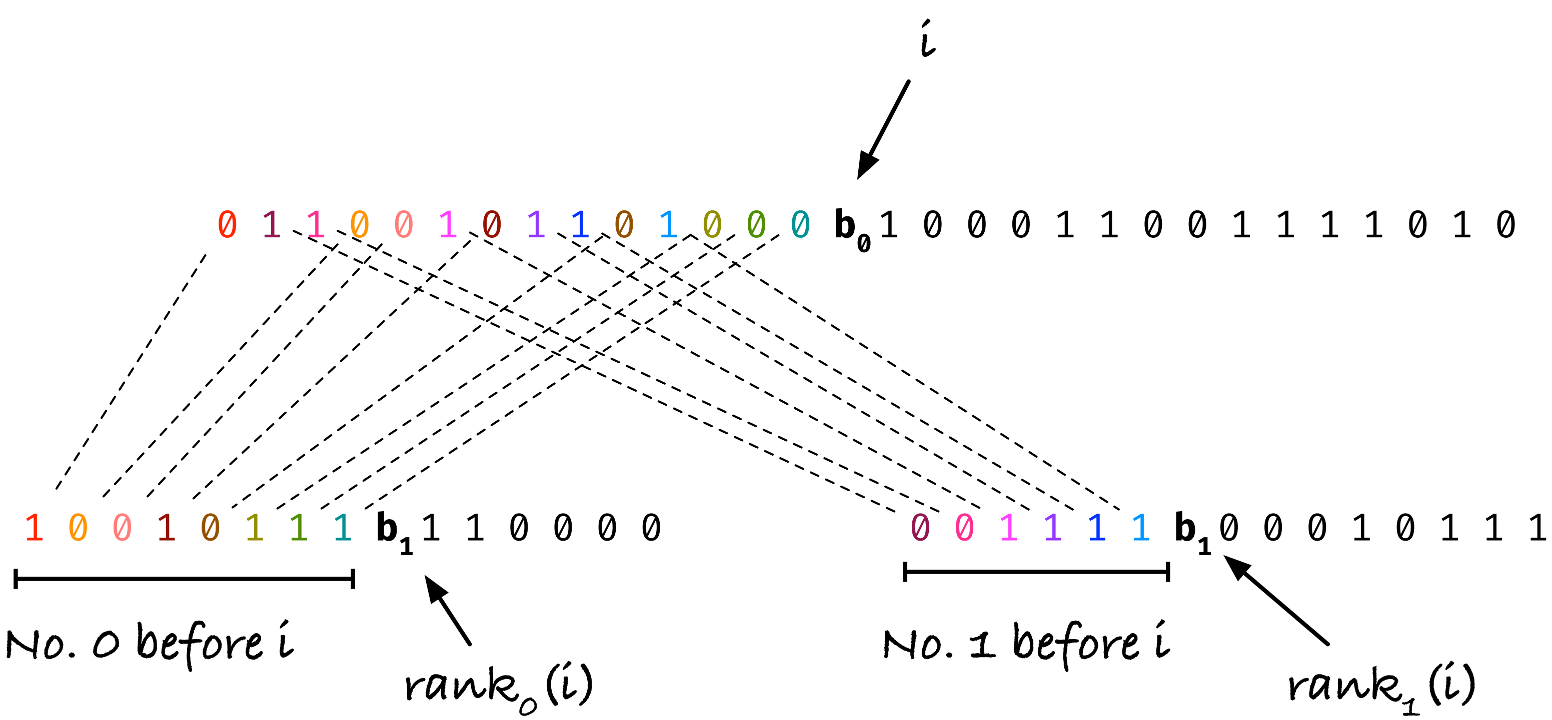
i

$x =$ 0 1 1 0 0 1 0 1 1 0 1 0 0 0 $\mathbf{b_0}$ 1 0 0 0 1 1 0 0 1 1 1 1 0 1 0
1 0 0 0 0 1 1 1 1 0 1 1 1 1 $\mathbf{b_1}$ 1 0 0 0 1 1 0 0 1 1 1 1 0 1 0

$\text{access}(v, i) = A \text{ if } v = \boxed{A}$

$\text{access}(v, i) = \text{access}(\text{left}, v.\text{rank}_0(i)) \text{ if } v[i] = 0$

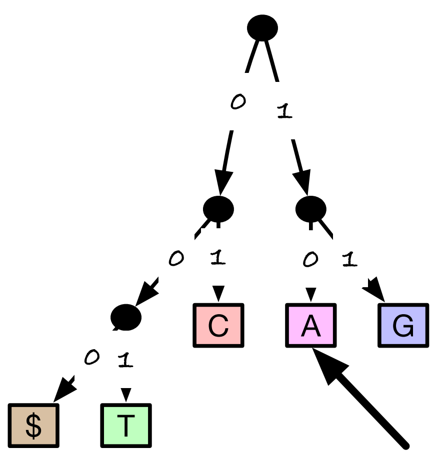
$\text{access}(v, i) = \text{access}(\text{right}, v.\text{rank}_1(i)) \text{ if } v[i] = 1$



Time: O("depth of leaf")

O(log σ) with sensible encoding

Wavelet trees

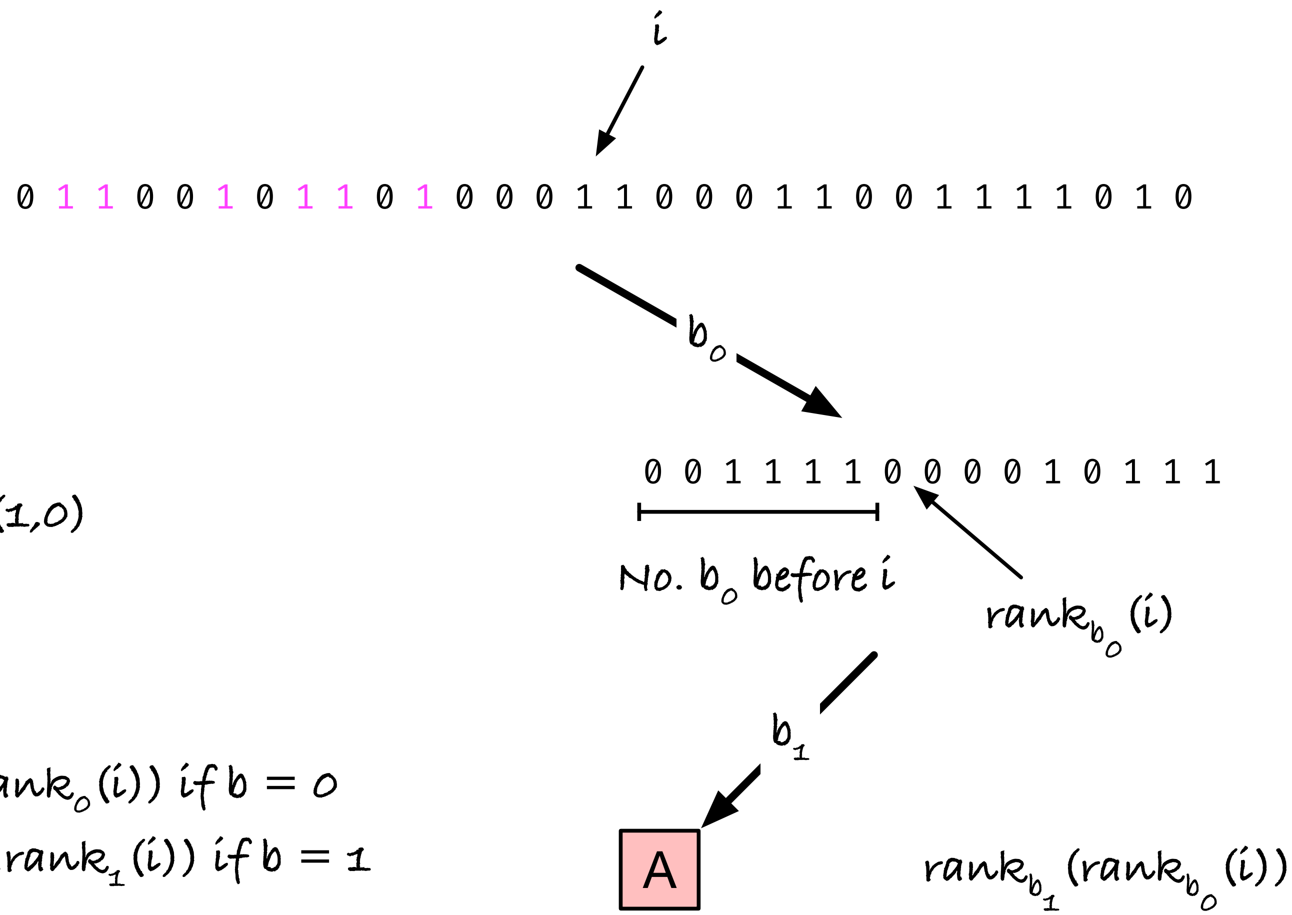


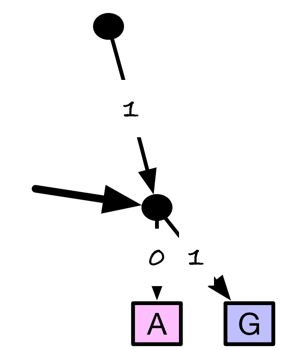
$enc(A) = (b_0, b_1)\ (1,0)$

$rank(v, b::bs, i) = i \text{ if } v = \boxed{A}$

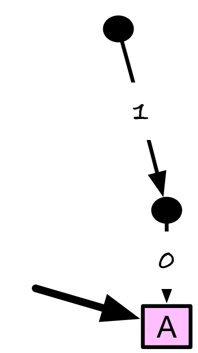
$rank(v, b::bs, i) = rank(left, bs, v.rank_0(i)) \text{ if } b = 0$

$rank(v, b::bs, i) = rank(right, bs, v.rank_1(i)) \text{ if } b = 1$





Number of As and Gs.



Time: O("depth of leaf")

O(log σ) with sensible encoding

```python
@dataclass
class WTNode:
    a:    bytes      | None = None
    bv:   BitVector | None = None
    zero: WTNode     | None = None
    one:  WTNode     | None = None

    @property
    def is_leaf(self) -> bool:
        return self.a is not None
```

```
def split_bits(x: str, bit_mask: int) → str:
    # Warning: excessive copying
    bits =  [a & bit_mask for a in x]           # get the bits for this level
    zeros = [a for a in x if not a & bit_mask]  # get the substring with zeros
    ones =  [a for a in x if a & bit_mask]      # get the substring with ones
    return bits, zeros, ones
```

Assumption: up to bit k, the letters we have are identical and match the path down the tree so far.

bit_mask = (1 << k) extracts the bit at position k; the next we should split on.

```
k = 2
bit_mask = 0b0100
shared = bit_mask - 1 = 0b0011

x = [ 0b0101, 0b0101, 0b1001, 0b1101, 0b0001, 0b0101 ]
  &   0b0011, 0b0011, 0b0011, 0b0011, 0b0011, 0b0011
  =   0b0001, 0b0001, 0b0001, 0b0001, 0b0001, 0b0001

x = [ 0b0101, 0b0101, 0b1001, 0b1101, 0b0001, 0b0101 ]
  &   0b0100, 0b0100, 0b0100, 0b0100, 0b0100, 0b0100
  =   0b0100, 0b0100, 0b0000, 0b0100, 0b0000, 0b0100
```

```python
def build_wt(x: str, bit_mask: int, prefix_mask: int) -> WTNode:
    bits, zeros, ones = split_bits(x, bit_mask)
    if len(zeros) == 0 or len(ones) == 0:
        # We have a leaf. The prefix_mask is the bit-pattern for the
        # characters here.
        return WTNode(a=prefix_mask)

    bv = BitVector(bits)  # (changed interface so init with list)
    bv.preprocess_rank()

    return WTNode(bv=bv,
                  zero=build_wt(zeros, bit_mask << 1, prefix_mask),
                  one=build_wt(ones, bit_mask << 1, prefix_mask | bit_mask))
```

```
bit_mask =    0b00010000 → 0b00100000
prefix_mask = 0b0000xxxx → 0b0000xxxx
```

```python
def build_wt(x: str, bit_mask: int, prefix_mask: int) -> WTNode:
    bits, zeros, ones = split_bits(x, bit_mask)
    if len(zeros) == 0 or len(ones) == 0:
        # We have a leaf. The prefix_mask is the bit-pattern for the
        # characters here.
        return WTNode(a=prefix_mask)

    bv = BitVector(bits)  # (changed interface so init with list)
    bv.preprocess_rank()

    return WTNode(bv=bv,
                  zero=build_wt(zeros, bit_mask << 1, prefix_mask),
                  one=build_wt(ones, bit_mask << 1, prefix_mask | bit_mask))
```

```
bit_mask =    0b00010000 -> 0b00100000
prefix_mask = 0b0000xxxx -> 0b0001xxxx
```

```python
def split_bits(x: str, bit_mask: int) -> str:
    # Warning: excessive copying
    bits =  [a & bit_mask for a in x]            # get the bits for this level
    zeros = [a for a in x if not a & bit_mask]   # get the substring with zeros
    ones =  [a for a in x if a & bit_mask]       # get the substring with ones
    return bits, zeros, ones
```

Warning: Assumption of sensible encoding here. We don't handle redundant bits (nor should we have to with a good encoding).

```python
def build_wt(x: str, bit_mask: int,
    bits, zeros, ones = split_bits(x, bit_mask)
    if len(zeros) == 0 or len(ones) == 0:
        # We have a leaf. The prefix_mask is the bit pattern for the
        # characters here.
        return WTNode(a=prefix_mask)

    bv = BitVector(bits)  # (changed interface so init with list)
    bv.preprocess_rank()

    return WTNode(bv=bv,
                  zero=build_wt(zeros, bit_mask << 1, prefix_mask),
                  one=build_wt(ones, bit_mask << 1, prefix_mask | bit_mask))
```

```
class WaveletTree:
    tree: WTNode

    def __init__(self, x):
        bit_mask    = 0b1  # pick first bit ...
        prefix_mask = 0b0  # empty prefix
        self.tree = build_wt(x, bit_mask, prefix_mask)
```

```python
def __getitem__(self, i):
    wt = self.tree
    while not wt.is_leaf:
        if wt.bv[i] == 0:
            wt, i = wt.zero, wt.bv.rank(0, i)
        else:
            wt, i = wt.one, wt.bv.rank(1, i)
    return wt.a

def rank(self, a, i):
    wt = self.tree
    while not wt.is_leaf:
        bit, a = a & 1, a >> 1
        if bit == 0:
            wt, i = wt.zero, wt.bv.rank(0, i)
        else:
            wt, i = wt.one, wt.bv.rank(1, i)
    return i
```

```go
type wtNode struct {
    a    byte
    bv   *bv.BitVector
    zero *wtNode
    one  *wtNode
}

func (n *wtNode) isLeaf() bool { return n.zero == nil }

type WaveletTree struct {
    root *wtNode
}
```

```go
// Warning: destroys x!
func NewWaveletTree(x []byte) *WaveletTree {
	buf := make([]byte, len(x)) // shared buffer for saving space…
	return &WaveletTree{root: buildWaveletTree(x, buf, 1, 0)}
}
```

```go
func buildWaveletTree(x, buf []byte, bit_mask, mask byte) *wtNode {
    noZeros, noOnes := countZeroOne(x, bit_mask)
    if noZeros == 0 || noOnes == 0 {
        return &wtNode{a: mask}
    }

    bv := bv.NewBitVector(noZeros + noOnes)
    // but the zeros first in buf and then the ones
    zeros, ones := buf[:noZeros], buf[noZeros:]
    splitOnBits(x, bv, zeros, ones, bit_mask)

    // the recursion swap buf/x so it can use x for the buffer
    // while it has the string in buf and vice versa.
    return &wtNode{
        bv:   bv,
        zero: buildWaveletTree(zeros, x[:noZeros],
                               bit_mask<<1, mask),
        one:  buildWaveletTree(ones, x[noZeros:],
                               bit_mask<<1, bit_mask|mask)}
}
```

```go
func (wt *WaveletTree) Access(i uint32) byte {
	n := wt.root
	for !n.isLeaf() {
		switch n.bv.Access(i) {
		case false:
			n, i = n.zero, n.bv.Rank0(i)
		case true:
			n, i = n.one, n.bv.Rank1(i)
		}
	}
	return wt.dec[n.a]
}

func (wt *WaveletTree) Rank(a byte, i uint32) uint32 {
	n := wt.root
	for ; !n.isLeaf(); a >>= 1 {
		switch a & 1 {
		case 0:
			n, i = n.zero, n.bv.Rank0(i)
		case 1:
			n, i = n.one, n.bv.Rank1(i)
		}
	}
	return i
}
```

Combined access and rank
(when recomputing the SA)

```
# O[i, bwt(x)[i]] - offset in C[a] + O[i,a] when rotating
def rotate_offset(self, i: int) -> int:
    return self.rank(self[i], i)
```

Two almost identical searches in the wavelet tree

Combined access and rank

```python
def rotate_offset(self, i: int) -> int:
    wt = self.tree
    while not wt.is_leaf:
        if wt.bv[i] == 0:
            wt, i = wt.zero, wt.bv.rank(0, i)
        else:
            wt, i = wt.one, wt.bv.rank(1, i)
    return i
```

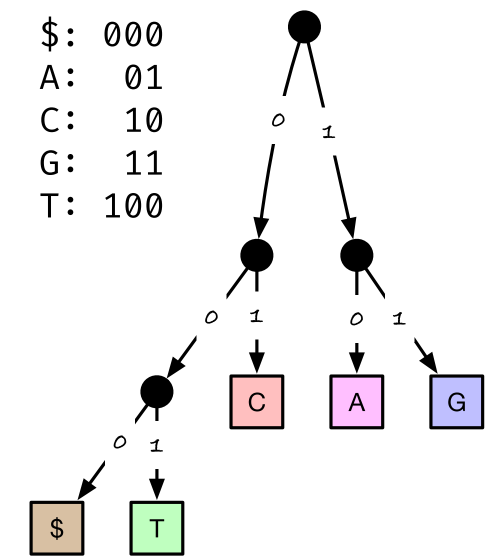
Search as in access:

Return as in rank

How much memory are we using?

Structure	**Usage**	**Memory (bits)**	**Access cost**
C table	Rotations	$O(\sigma \log n)$	
WT[bwt(x)]	Rank and rotations	**$O(\sigma + n \log \sigma)$**	**$O(\log \sigma)$**
WT[bwt(rev(x))]	Rank and rotations	**$O(\sigma + n \log \sigma)$**	**$O(\log \sigma)$**
p	Need to know the pattern	$O(m \log \sigma)$	
D table	Bounding branches	$O(m \log m)$ / $O(m \log d)$	
SA	Reporting matches	**$O(n + n/k' \log n)$**	**$O(k' \log \sigma)$**

Huffman encoding



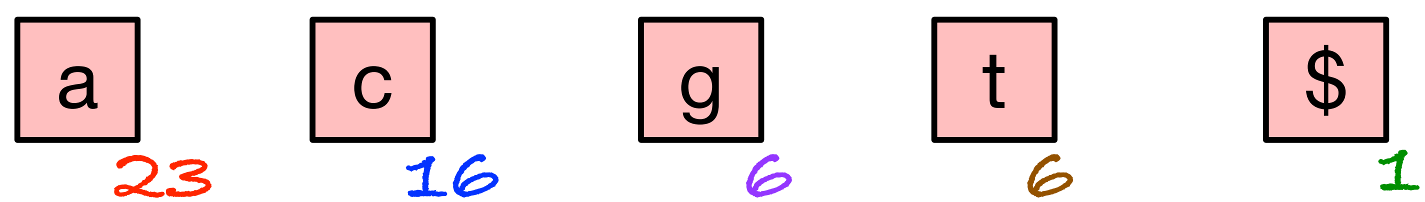
- How do we choose a character encoding?
- Can we get one that is optimal for our usage?
    - Minimal number of bits (while supporting operations)
    - Minimal average time on access/rank
    - **Huffman encoding** gives us that!

Huffman encoding

- Adapts the encoding to the string:
  - Frequent letters are encoded with fewer bits
  - Rare letters are encoded with more bits
  - Overall, we use fewer bits
- At this point, we don’t need to preserve the order of the words (we don’t need the order after we have the SA)

Huffman encoding

aaaccaggacattactttaccataaaccaacaaacaaaagccccaccaggg$

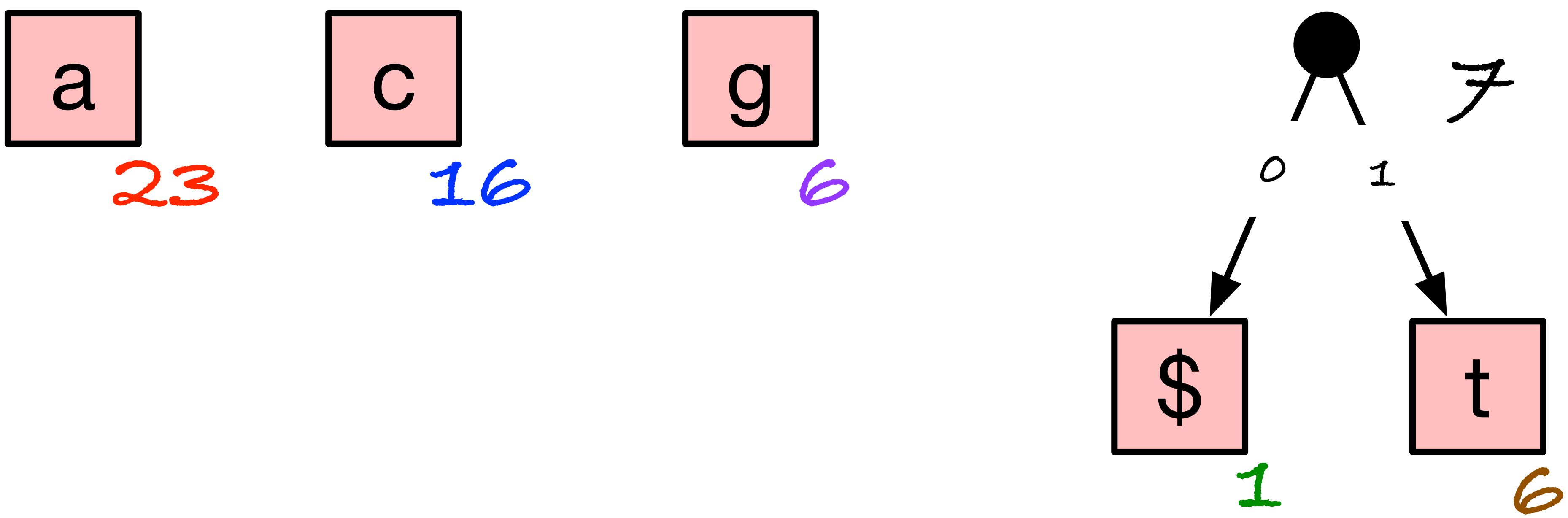


First: Count the characters and make leaves for each, weighted by the occurrence of each character.

Huffman encoding

Iteratively: Merge the nodes with the lowest counts to make new trees.

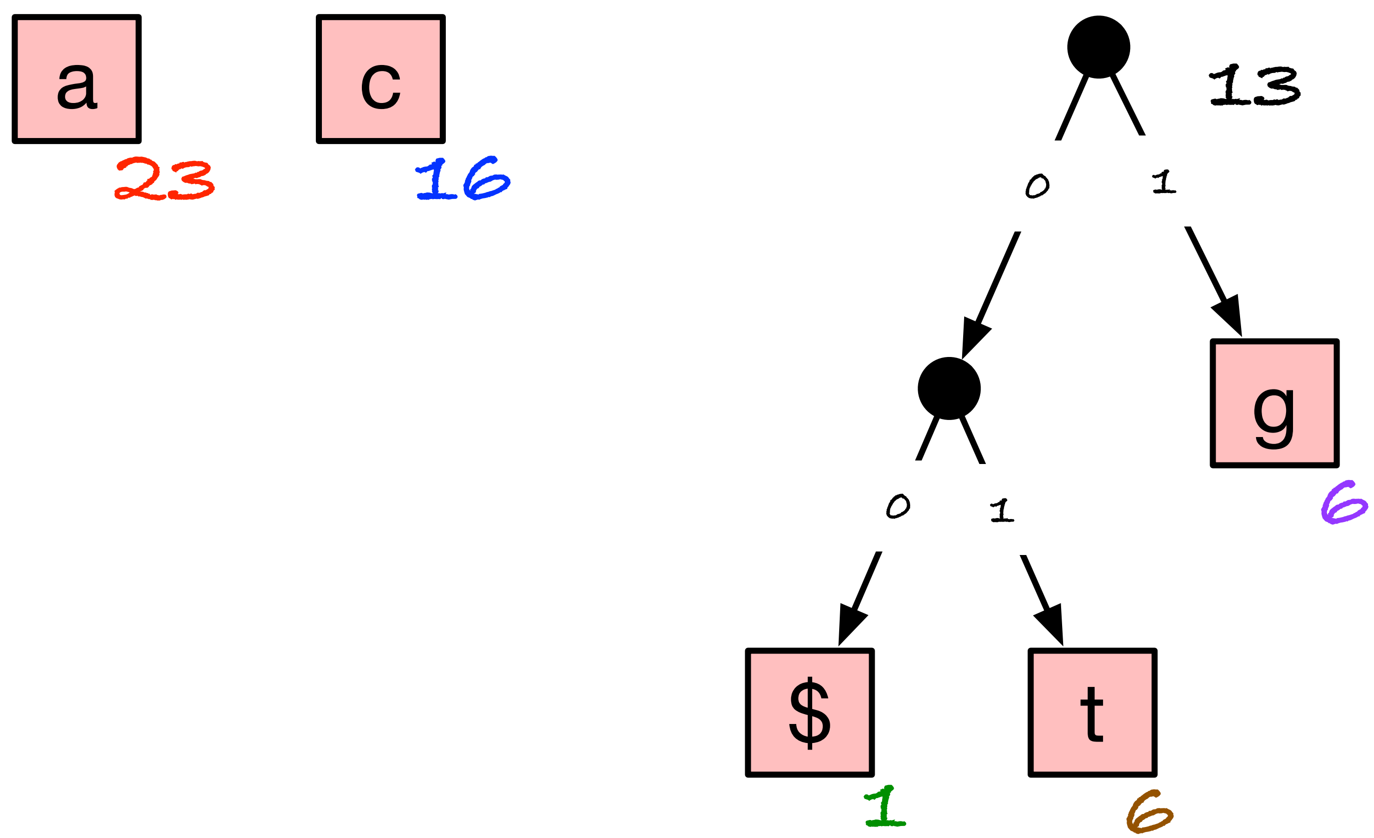
aaaccaggacattactttaccataaaccaacaaacaaaagccccaccaggg$



Huffman encoding

Iteratively: Merge the nodes with the lowest counts to make new trees.

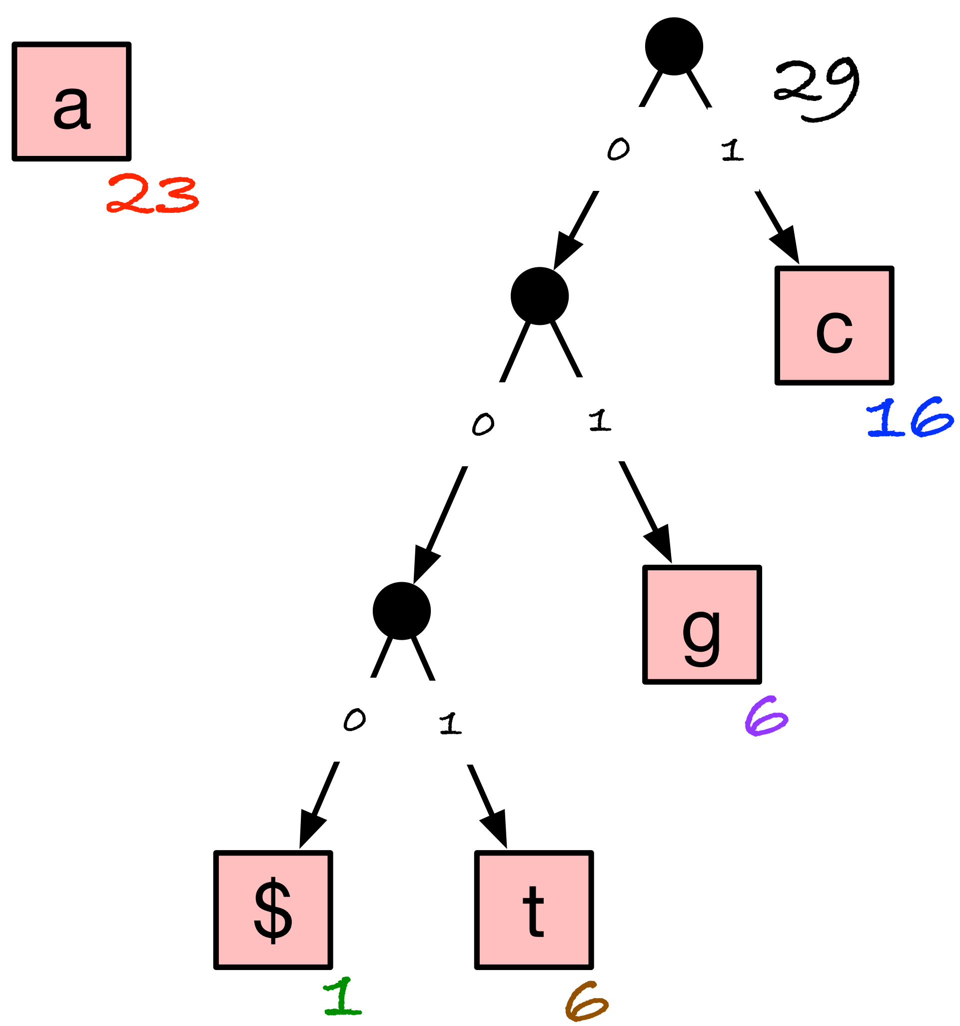
aaaccaggacattactttaccataaaccaacaaacaaaagccccaccaggg$



Huffman encoding

Iteratively: Merge the nodes with the lowest counts to make new trees.

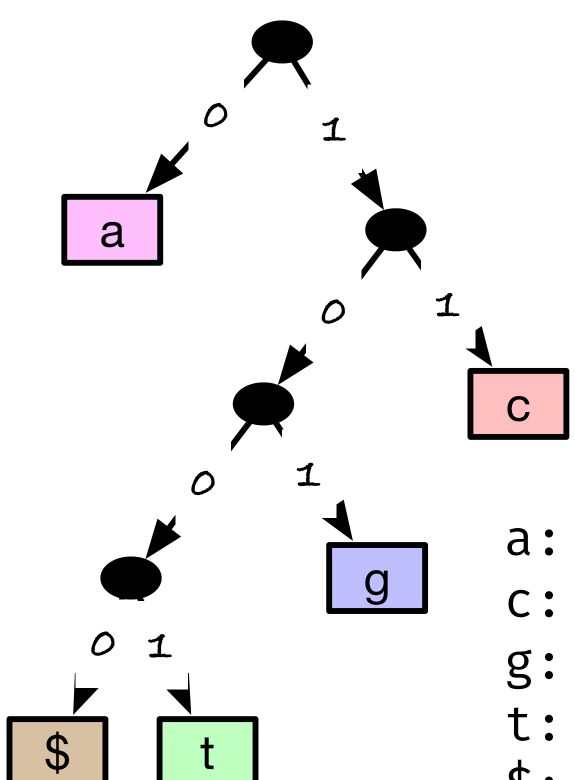
aaaccaggacattactttaccataaaccaacaaacaaaagccccaccaggg$



Huffman encoding

Done: when there is only a single tree. That tree is the encoding.

aaaccaggacattactttaccataaaccaacaaacaaaagccccaccaggg$




```
a:    0
c:   11
g:  101
t: 1001
$: 0001
```

```
Byte encoding of string is      416 bits.
With log sigma bits per char    156 bits.
Huffman encoding of string is   101 bits.
```

Construction time: $O(n + \sigma \log \sigma)$
$O(n)$ for counting characters, and
$O(\sigma \log \sigma)$ for building the tree
(using a heap for picking smallest trees).

```go
type huffmanNode struct {
    letter    byte
    size      int
    zero, one *huffmanNode
}

type huffmanHeap []*huffmanNode

// Stuff for a heap, because Go doesn't have generics yet
func (h huffmanHeap)  Len() int             { … }
func (h huffmanHeap)  Less(i, j int) bool { … }
func (h huffmanHeap)  Swap(i, j int)        { … }
func (h *huffmanHeap) Push(n interface{}) { … }
func (h *huffmanHeap) Pop() interface{}   { … }
```

```go
func letterCounts(x []byte) map[byte]int {
	counts := map[byte]int{}
	for _, a := range x {
		counts[a]++
	}
	return counts
}

func populateHuffmanHeap(x []byte) *huffmanHeap {
	counts := letterCounts(x)
	h := huffmanHeap(make([]*huffmanNode, len(counts)))

	i := 0
	for a, count := range letterCounts(x) {
		h[i] = &huffmanNode{letter: a, size: count}
		i++
	}

	heap.Init(&h)

	return &h
}
```

```go
func buildHuffmanTree(x []byte) *huffmanNode {
    hh := populateHuffmanHeap(x)
    for hh.Len() > 1 {
        a := heap.Pop(hh).(*huffmanNode)
        b := heap.Pop(hh).(*huffmanNode)
        heap.Push(hh, &huffmanNode{size: a.size + b.size,
                                   zero: a, one: b})
    }
    return heap.Pop(hh).(*huffmanNode)
}
```

```go
func buildHuffmanMap(hn *huffmanNode,
                     enc *map[byte]byte,
                     prefix, bit byte) {
    if hn.zero == nil {
        (*enc)[hn.letter] = prefix
    } else {
        buildHuffmanMap(hn.zero, enc, prefix,          bit+1)
        buildHuffmanMap(hn.one,  enc, (1<<bit)|prefix, bit+1)
    }
}

func HuffmanEncoding(x []byte) (enc map[byte]byte) {
    enc = map[byte]byte{}
    buildHuffmanMap(buildHuffmanTree(x), &enc, 0, 0)
    return enc
}
```

Huffman encoding

- Huffman encoding is optimal for compression (when each character is represented as its own series of bits)
- This makes the wavelet tree as small as possible
- If the patterns we search for follow the same character frequency (they should if they are reads from the same genome), then we also optimise the time we use for access and rank.

How much memory are we

The log σ could be better even than that.

However, for DNA strings, the bias in frequencies isn't large, and σ is only 5 (when we include $ in the alphabet), so we don't gain terribly much in practise.

			Access cost
C table	Rotations	O(σ log n)	
WT[bwt(x)]	Rank and rotations	**O(σ + n log σ)**	**O(log σ)**
WT[bwt(rev(x))]	Rank and rotations	**O(σ + n log σ)**	**O(log σ)**
p	Need to know the pattern	O(m log σ)	
D table	Bounding branches	O(m log m) / O(m log d)	
SA	Reporting matches	**O(n + n/k' log n)**	**O(k' log σ)**

Structure	Before (memory in bits)	Now (memory in bits)
C table	$O(\sigma \log n)$	C
O table	**$O(\sigma n \log n)$**	
WT[bwt(x)]		**$O(\sigma + n \log \sigma)$**
RO table	**$O(\sigma n \log n)$**	
WT[bwt(rev(x))]		**$O(\sigma + n \log \sigma)$**
p	$O(m \log \sigma)$	$O(m \log \sigma)$
D table	$O(m \log m)$ / $O(m \log d)$	$O(m \log m)$ / $O(m \log d)$
SA	**$O(n \log n)$**	**$O(n + n/k' \log n)$**

$\sigma \log n = 5 \times 32 = 128$
$\log \sigma = 3$
$3/128 \approx 2\%$

Running time	Before	Now (penalties)
Searching for (L,R) interval	**$O(m)$**	**$O(m \log \sigma)$**
Reporting matches	**$O(z)$**	**$O(z\, k' \log \sigma)$**

Structure	Before (memory in bits)	Now (memory in bits)
C table	$O(\sigma \log n)$	O
O table	**$O(\sigma n \log n)$**	
WT[bwt(x)]		**$O(\sigma + n \log \sigma)$**
RO table	**$O(\sigma n \log n)$**	
WT[bwt(rev(x))]		**$O(\sigma + n \log \sigma)$**
	$O(m \log \sigma)$	$O(m \log \sigma)$
	$O(m \log m)$ / $O(m \log d)$	$O(m \log m)$ / $O(m \log d)$
	$O(n \log n)$	**$O(n + n/k' \log n)$**

$\sigma \log n = 5 \times 32 = 128$

$\log \sigma = 3$

$3/128 \approx 2\%$

Can we get rid of $?

Then σ: $5 \Rightarrow 4$ and $\log \sigma$: $3 \Rightarrow 2$

That's another potential 50% (although it won't be with Huffman encoding)

Running time	Before	Now (penalties)
Searching for (L,R) interval	**$O(m)$**	**$O(m \log \sigma)$**
Reporting matches	**$O(z)$**	**$O(z k' \log \sigma)$**

Do we need $?

- The LM search algorithm needs the sentinel, or at least, the indices in the bwt(x) should take it into account.
- However, we never use `rank(i,$)` in the search, because $ isn't in the patterns.
- In the rotations for the reduced suffix array, we would use access and rank with $ if we look at the zeroth rotation, but we know the SA value there, and wouldn't rotate.

bwt(x) = ipssm$pissii

Wavelet tree represents this

We never rotate this row

	$	i	m	p	s
$mississippi	0	0	0	0	0
i$mississipp	0	1	0	0	0
ippi$mississ	0	1	0	1	0
issippi$miss	0	1	0	1	1
ississippi$m	0	1	0	1	2
mississippi$	0	1	1	1	2
pi$mississip	1	1	1	1	2
ppi$mississi	1	1	1	2	2
sippi$missis	1	2	1	2	2
sissippi$mis	1	2	1	2	3
ssippi$missi	1	2	1	2	4
ssissippi$mi	1	3	1	2	4
	1	4	1	2	4

O[$, i, m, p, s]

We never rank this column

Wavelet tree represents this

bwt(x) = ipssm$pissii

wt(ipssmpissii)

	O[i, m, p, s]
$mississippi	0, 0, 0, 0
i$mississipp	1, 0, 0, 0
ippi$mississ	1, 0, 1, 0
issippi$miss	1, 0, 1, 1
ississippi$m	1, 0, 1, 2
mississippi$	[missing index]
pi$mississip	1, 1, 1, 2
ppi$mississi	1, 1, 2, 2
sippi$missis	2, 1, 2, 2
sissippi$mis	2, 1, 2, 3
ssippi$missi	2, 1, 2, 4
ssissippi$mi	3, 1, 2, 4
	4, 1, 2, 4

Wavelet tree represents this

bwt(x) = ipssm$pissii   wt(ipssmpissii)

	O[i, m, p, s]
$mississippi	0, 0, 0, 0
i$mississipp	1, 0, 0, 0
ippi$mississ	1, 0, 1, 0
issippi$miss	1, 0, 1, 1
ississippi$m	1, 0, 1, 2
mississippi$	1, 1, 1, 2
pi$mississip	1, 1, 2, 2
ppi$mississi	2, 1, 2, 2
sippi$missis	2, 1, 2, 3
sissippi$mis	2, 1, 2, 4
ssippi$missi	3, 1, 2, 4
ssissippi$mi	4, 1, 2, 4

sentIndex = 5

map: i = i − (i ⩾ sentIndex)

That's all Folks!