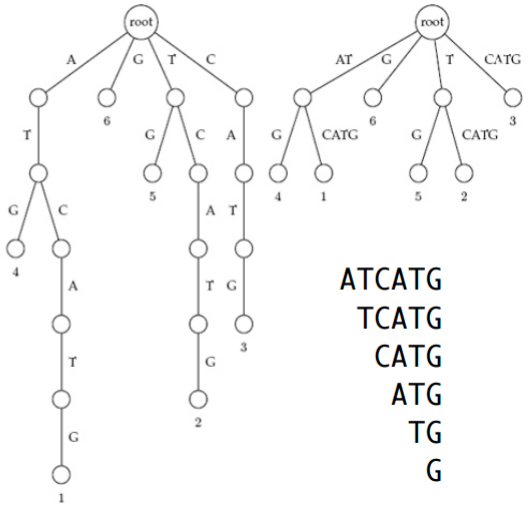
Lecture 17: Suffix Arrays and Burrows Wheeler Transforms

Not in Book

Recall Suffix Trees

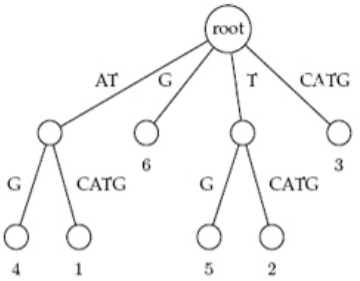
- A **suffix trie** is a keyword trie of all suffixes (left figure)
- A **suffix tree** compresses the trie by combining nodes with out degree 1
  - Edges represent a substring of text
  - All internal nodes have at least 3 edges
  - All leaf nodes are labeled with an index



ATCATG
TCATG
CATG
ATG
TG
G

Suffix Trees

- Nodes (fixed alphabet) - pointer to edge for each letter
  - DNA - 4 letters
  - ~16 bytes per node
- Edges are typically stored as (offset, length) pairs
  - "AT" = (0, 2)
  - "G" = (5, 1)
  - "CATG" = (2, 4)
  - ~8 bytes per edge




```
ATCATG
012345
```

3

Suffix Tree Summary

- Input: text of length *m*
- Computation
  - O(*m*) to compute a suffix tree
  - Does not require building the suffix trie first
- Memory
  - O(*m*) - nodes are stored as offsets and lengths
  - Huge hidden constant, best implementations requires about 20**m* bytes
  - 3 GB human genome = 60 GB RAM

4

Suffix Arrays

- Related to suffix trees, but reduced memory
  - Keep string on disk O(m)
  - Keep array of pointers indicating sorted order of suffixes O(m*log(m)) bits
  - In practice the log(m) is typically 4 bytes
- Computation
  - Can also be done in O(m) time
  - Uses O(m) memory; approximately 12 GB for the human genome

```
13 $
5  abananas$
3  amabananas$
1  anamabananas$
7  ananas$
9  anas$
11 as$
6  bananas$
4  mabananas$
2  namabananas$
8  nanas$
10 nas$
0  panamabananas$
12 s$
```

The suffix array (and suffixes) for the string "panamabananas$"

5

Searching Suffix Arrays

- Binary search
  - Pattern length d
  - O(log(m)) comparisons
  - Worst case is O(d) operations per comparison
  - O(d*log(m)) per pattern

```
13 $
5  abananas$
3  amabananas$
1  anamabananas$
7  ananas$
9  anas$
11 as$
6  bananas$
4  mabananas$
2  namabananas$
8  nanas$
10 nas$
0  panamabananas$
12 s$
```

6

Searching Suffix Arrays

```
def searchFirst(sa, t, p):
  l = 0
  h = len(t)
  while l < h:
    m = (l+h)/2
    if t[sa[m]:] < p:
      l = m+1
    else:
      h = m
  return l
```

searchFirst(sa, t, 'am')
(0, 14) -> (0, 7) -> (0, 3) -> (2, 3) -> (2, 2)
return 2

i	sa[i]	t[sa[i]]
0	13	$
1	5	abananas$
2	3	amabananas$
3	1	anamabananas$
4	7	ananas$
5	9	anas$
6	11	as$
7	6	bananas$
8	4	mabananas$
9	2	namabananas$
10	8	nanas$
11	10	nas$
12	0	panamabananas$
13	12	s$

7

Searching Suffix Arrays

```
def searchLast(sa, t, p):
  l = 0
  h = len(t)
  while l < h:
    m = (l+h)/2
    if t[sa[m]:sa[m]+len(p)] <= p:
      l = m+1
    else:
      h = m
  return l-1
```

8

Other Data Structures

- There is another trick for finding patterns in a text string, it comes from a rather odd remapping of the original text called a "Burrows-Wheeler Transform" or BWT.
- BWTs have a long history. They were invented back in the 1980s as a technique for improving lossless compression. BWTs have recently been rediscovered and used for DNA sequence alignments. Most notably by the Bowtie and BWA programs for sequence alignments.

9

String Rotation

Before describing the BWT, we need to define the notion of Rotating a string. The idea is simple, a rotation of *i* moves the prefix_i, to the string's end making it a suffix.

Rotate("tarheel$", 3) → "heel$tar"
Rotate("tarheel$", 7) → "$tarheel"
Rotate("tarheel$", 1) → "arheel$t"

10

BWT Algorithm

BWT (string text)

table$_i$ = Rotate(text, i) for i = 0..len(text)-1

sort table alphabetically

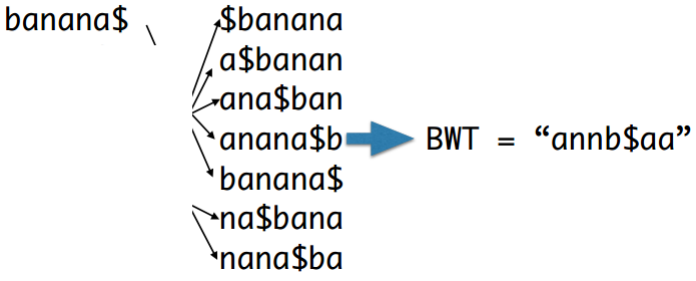
return (last column of the table)

```
tarheel$        $tarheel
arheel$t        arheel$t
rheel$ta        eel$tarh
heel$tar        el$tarhe
eel$tarh        heel$tar
el$tarhe        l$tarhee
l$tarhee        rheel$ta
$tarheel        tarheel$
```

BWT("tarheels$") = "ltherea$"

BWT Example

BWT('banana$')



BWT in Python

- This one of the simpler algorithms that we've seen

```
def BWT(s):
    # create a table, with rows of all possible rotations of s
    rotation = [s[i:] + s[:i] for i in xrange(len(s))]
    # sort rows alphabetically
    rotation.sort()
    # return (last column of the table)
    return "".join([r[-1] for r in rotation])
```

- Input string of length *m*, output a messed up string of length *m*

13

Inverse of BWT

A property of a transform is that there is no information loss and they are invertible.

inverseBWT(string *s*)
  add *s* as the first column of a table strings
  repeat length(s)-1 times:
    sort rows of the table alphabetically
    add *s* as the first column of the table
  return (row that ends with the 'EOF' character)

```
l   l$   l$t   l$ta   l$tar   l$tarh   l$tarhe   l$tarhee
t   ta   tar   tarh   tarhe   tarhee   tarheel   tarheel$
h   he   hee   heel   heel$   heel$t   heel$ta   heel$tar
e   ee   eel   eel$   eel$t   eel$ta   eel$tar   eel$tarh
r   rh   rhe   rhee   rheel   rheel$   rheel$t   rheel$ta
e   el   el$   el$t   el$ta   el$tar   el$tarh   el$tarhe
a   ar   arh   arhe   arhee   arheel   arheel$   arheel$t
$   $t   $ta   $tar   $tarh   $tarhe   $tarhee   $tarheel
```

14

Inverse BWT in Python

- A slightly more complicated routine

```
def inverseBWT(s):
    # initialize table from s
    table = [c for c in s]
    # repeat length(s) - 1 times
    for j in xrange(len(s)-1):
        # sort rows of the table alphabetically
        table.sort()
        # insert s as the first column
        table = [s[i]+table[i] for i in xrange(len(s))]
    # return (row that ends with the 'EOS' character)
    return table[[r[-1] for r in table].index('$')]
```

15

How to use a BWT?

- A BWT is a "*last-first*" mapping meaning the i^{th} occurrence of a character in the first column corresponds to the i^{th} occurrence in the last.
- Also, recall the first column is sorted
- BWT("mississippi$") → "ipssm$pissii"
- Compute from BWT(s) a sorted dictionary of the number of occurrences of each letter

  N = { '$':1, 'i':4, 'm':1, 'p':2, 's':4 }
- Using N it is a simple matter to find indices of the first occurrence of a character on the "left" sorted side

  I = { '$':0, 'i':1, 'm':5, 'p':6, 's':8 }
- We also use N to compute the "right-hand" offsets or C-index

C-index

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

16

Searching for a Pattern

- Find “sis” in “mississippi”
- Search for patterns take place in reverse order (last character to first)
- Use the I index to find the range of entries starting with the last character

I = { ‘$’:0, ‘i’:1, ‘m’:5, ‘p’:6, ‘s’:8 }

```
$mississippi
i$mississipp
ippi$mississ
issippi$miss
ississippi$m
mississippi$
pi$mississip
ppi$mississi
sippi$missis
sissippi$mis
ssippi$missi
ssissippi$mi
```

17

Searching for a Pattern

- Find “sis” in “mississippi”
- Of these, how many BWT entries match the second-to-last character? If none string does not appear
- Use the C-index to find all offsets of occurrences of these second to last characters, which will be contiguous

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

18

Searching for a Pattern

- Find “sis” in “mississippi”
- Combine offsets with I index entry to narrow search range
- Add the C-index offsets to the I-index of the second-to-last character to find new search range

I = { ‘$’:0, ‘i’:1, ‘m’:5, ‘p’:6, ‘s’:8 }

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

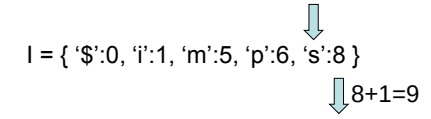

Searching for a Pattern

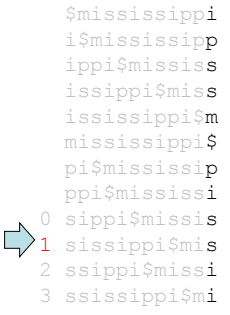
- Find “sis” in “mississippi”
- Find BWT entries that match the previous next-to-next-to-last character, ‘s’
- Use the C index to find the offsets of these second to last characters
- Now we know that the string appears in the text, but not where

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

Searching for a Pattern

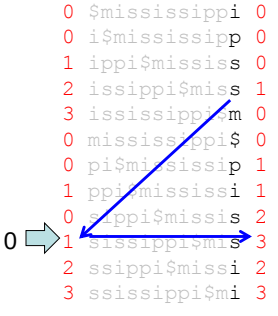
- Find “sis” in “mississippi”
- We can find the pattern’s offset on the left side by combining the C index with the I index value for the first character
- Now, if we had a Suffix array we could use it to find the offset into the original text





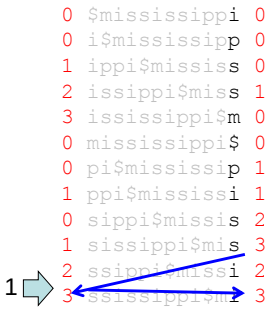
Searching for a Pattern

- Find “sis” in “mississippi”
- Actually, *there is an implicit suffix array* in our BWT
- We can use the last first-last property and the C index to thread back through the array to find the start position



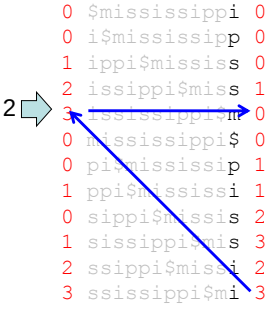
Searching for a Pattern

- Find “sis” in “mississippi”
- Actually, there is an implicit suffix array in our BWT
- We can use the last first-last property and the C index to thread back through the array to find the start position



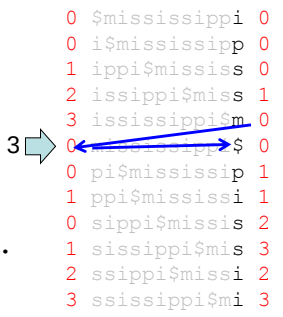
Searching for a Pattern

- Find “sis” in “mississippi”
- Actually, there is an implicit suffix array in our BWT
- We can use the last first-last property and the C index to thread back through the array to find the start position



Searching for a Pattern

- Find “sis” in “mississippi”
- Actually, there is an implicit suffix array in our BWT
- We can use the last first-last property and the C index to thread back through the array to find the start position
  - We’re done. The text offset is 3.



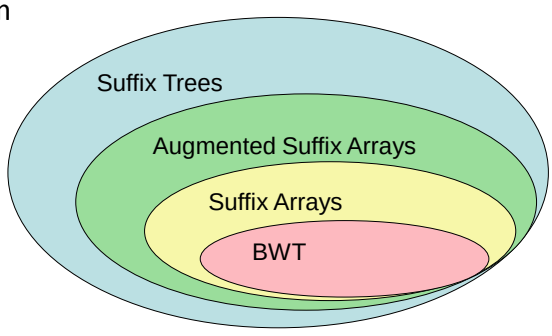
BWT Compression

- Uncompressed
  - Same as input text size
  - O(*N*) bytes
- Compression
  - Tendency to form long runs
  - Run-length encoding (RLE)
- Can be stored as: ACG$C3AGC

Index (N)	Suffix Array	BWT
0	$ACACGGACA	A
1	A$ACACGGAC	C
2	ACA$ACACGG	G
3	ACACGGACA$	$
4	ACGGACA$AC	C
5	CA$ACACGGA	A
6	CACGGACA$A	A
7	CGGACA$ACA	A
8	GACA$ACACG	G
9	GGACA$ACAC	C

Summary

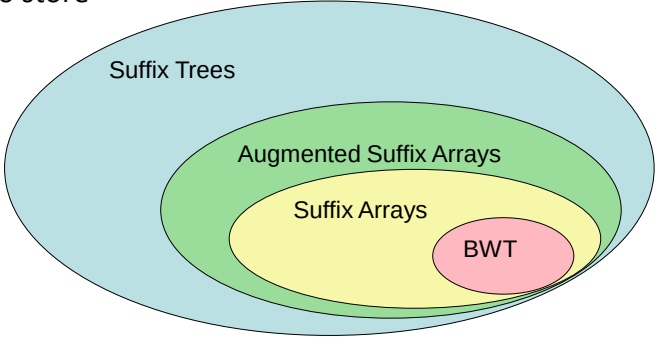
- Query Power (Big is good)
  - BWTs support the fewest query types of these data structs
  - Suffix Trees perform a variety of queries in O(*m*)



27

Summary

- Memory Footprint (Small is good)
  - BWTs compress very well on real data
  - Difficult to store the full suffix tree for an entire genome



28

Comparison

Let m = len(Genome)
Let d = len (longestPattern)
Let x = # of Patterns

Method	Storage Cost	Single Pattern Search Time	Multiple Pattern Search Time
Brute Force	O(*m*)	O(*dm*)	O(*xdm*)
Keyword Tries	O(*xd*)	O(*dm*)	O(*dm*)
Suffix Trees	O(*m*) [20*m* bytes]	O(*d*)	O(*xd*)
Suffix Arrays	O(*m**log(*m*)) [4*m* bytes]	O(*d**log(*m*))	O(*xd**log(*m*))
BWT	**O(*m*) [often *m* bits]**	**O(*d*)**	**O(*xd*)**

29

Tools using BWT in Exact Pattern Matching

- Alignment
  - Bowtie (2009) and BWA (2009)
  - Build a BWT of the reference genome (~2-3 GB)
  - Align:
    - Given a 100 base pair read
    - Cut into smaller seed pieces (i. e. four 25-mers)
    - Exact search for the pieces separately - very fast using BWT
    - Use local alignment (dynamic program) to extend initial seed alignments and account for errors
  - Bowtie2 (2011) and Tophat2 (2013) are still very prominent and fast aligners

30

FM-Index

- Ferragina & Manzini, 2005
- Enables fast exact searches
- **LF-mapping property** - Takes advantage of "last-first" relationship between BWT and suffix array
  - See colors on right
  - First "**A**" in BWT corresponds to first suffix starting with "**A**"
  - First "**C**" in BWT corresponds to first suffix starting with "**C**"
  - …
  - Second "**A**" in BWT corresponds to second suffix starting with "**A**"
  - …

Index (N)	Suffix Array	BWT
0	$ACACGGACA	A
1	A$ACACGGAC	C
2	ACA$ACACGG	G
3	ACACGGACA$	$
4	ACGGACA$AC	C
5	CA$ACACGGA	A
6	CACGGACA$A	A
7	CGGACA$ACA	A
8	GACA$ACACG	G
9	GGACA$ACAC	C

31

FM-Index

- ***A*** - alphabet size
- ***N*** - text length
- ***F*** - ***FM-index***
  - *F*[*i*][*c*] stores the number of times symbol *c* occurs before index *i*
  - O(*NA*) memory
  - Generated in a linear pass over the BWT
- ***O*** - Offset Array
  - *O*[*c*] stores the index of the first suffix starting with symbol *c*
  - Derived from the final entry in *F*
    `O[c] = sum(F[-1][0:c])`
  - O(*A*) memory

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	—	**1**	**4**	**3**	**2**
Offset (O)	—	—	**0**	**1**	**5**	**8**

32

Find Predecessor Suffix

- Given an index *i* in the BWT, find the index in the BWT of the suffix preceding the suffix represented by *i*
  - suffix 0 is preceded by suffix 1
  - suffix 1 is preceded by suffix 5
  - suffix 5 is preceded by suffix 2
- The predecessor suffix of index *i*:

```
c = BWT[i]
predec = O[c]+F[i][c]
```

- Predecessor of index 1
  c = BWT[1] = 'C'
  predec = O['C']+F[1]['C'] = 5+0 = 5
- Predecessor of index 8
  c = BWT[8] = 'G'
  predec = O['G']+F[8]['G'] = 8+1 = 9
- Time to find predecessor: **O(1)**

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

33

Suffix Recovery

- Suffix recovery:
  - Start at an index *i*
  - Repeatedly find the predecessor
  - Stop when back at original index
- Original string recovery:
  - Start at 0
  - Repeatedly find predecessor until back at 0
  - **O(*N*)** time to get original string back

```
def recoverSuffix(BWT, F, O, i):
  ret = []
  c = BWT[i]
  predec = O[c]+F[i][c]
  ret.append(c)
  while predec != i:
    c = BWT[predec]
    predec = O[c]+F[predec][c]
    ret.append(c)
  return "".join(ret[::-1])
```

34

Find k-mer

- ***k*-mer**: a pattern of length *k*
- All searches occur in reverse order
  - Start with full BWT range (0, N)
  - Restrict by one symbol at a time
- Find *k*-mer "ACA"
  - Initialize to full range ("")
    low, high = 0, 10
  - Find occurrences of "A"
    low = O['A']+F[low]['A'] = 1+0 = 1
    high = O['A']+F[high]['A'] = 1+4 = 5
  - Find occurrences of "CA"
    low = O['C']+F[low]['C'] = 5+0 = 5
    high = O['C']+F[high]['C'] = 5+2 = 7
  - Find occurrences of "ACA"
    low = O['A']+F[low]['A'] = 1+1 = 2
    high = O['A']+F[high]['A'] = 1+3 = 4

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGACA	A	0	0	0	0
1	A$ACACGGAC	C	0	1	0	0
2	ACA$ACACGG	G	0	1	1	0
3	ACACGGACA$	$	0	1	1	1
4	ACGGACA$AC	C	1	1	1	1
5	CA$ACACGGA	A	1	1	2	1
6	CACGGACA$A	A	1	2	2	1
7	CGGACA$ACA	A	1	3	2	1
8	GACA$ACACG	G	1	4	2	1
9	GGACA$ACAC	C	1	4	2	2
10	—	—	**1**	**4**	**3**	**2**
Offset (O)	—	—	**0**	**1**	**5**	**8**

35

Find k-mer

- *p* - pattern
- *F* - FM-index
- Time complexity - O(*k*)
  - Requires O(*k*) lookups
  - Search time only dependent on length of *k*-mer
  - Does not depend on BWT (data) size!!!

```
def find(p, F, O):
  lo = 0
  hi = len(F)
  for l in reversed(p):
    lo = O[l] + F[lo][l]
    hi = O[l] + F[hi][l]
  return lo, hi
```

36

find("AGG", F, 0)

```
def find(p, F, O):
  lo = 0
  hi = len(F)
  for l in reversed(p):
    lo = O[l] + F[lo][l]
    hi = O[l] + F[hi][l]
  return lo, hi
```

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGACA	A	0	0	0	0
1	A$ACACGGAC	C	0	1	0	0
2	ACA$ACACGG	G	0	1	1	0
3	ACACGGACA$	$	0	1	1	1
4	ACGGACA$AC	C	1	1	1	1
5	CA$ACACGGA	A	1	1	2	1
6	CACGGACA$A	A	1	2	2	1
7	CGGACA$ACA	A	1	3	2	1
8	GACA$ACACG	G	1	4	2	1
9	GGACA$ACAC	C	1	4	2	2
10	—	—	**1**	**4**	**3**	**2**
Offset (O)	—	—	**0**	**1**	**5**	**8**

37

Practical Adaptations

- Compressed BWT is small
- FM-index is not, O(*A***N*) for alphabet of size *A* and a BWT of length *N*
- Trade-off, space v. time:
  - Use a sampled FM-index
  - *B* - bin size
  - Uses O(*A***N*/*B*) values
  - Requires O(*B*) time per lookup (for a fixed size *B*, this is just a larger constant time lookup)
  - typically use 1024 in practice

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGACA	A	**0**	**0**	**0**	**0**
1	A$ACACGGAC	C	0	1	0	0
2	ACA$ACACGG	G	0	1	1	0
3	ACACGGACA$	$	0	1	1	1
4	ACGGACA$AC	C	1	1	1	1
5	CA$ACACGGA	A	**1**	**1**	**2**	**1**
6	CACGGACA$A	A	1	2	2	1
7	CGGACA$ACA	A	1	3	2	1
8	GACA$ACACG	G	1	4	2	1
9	GGACA$ACAC	C	1	4	2	2
10	—	—	**1**	**4**	**3**	**2**
Offset (O)	—	—	**0**	**1**	**5**	**8**

38