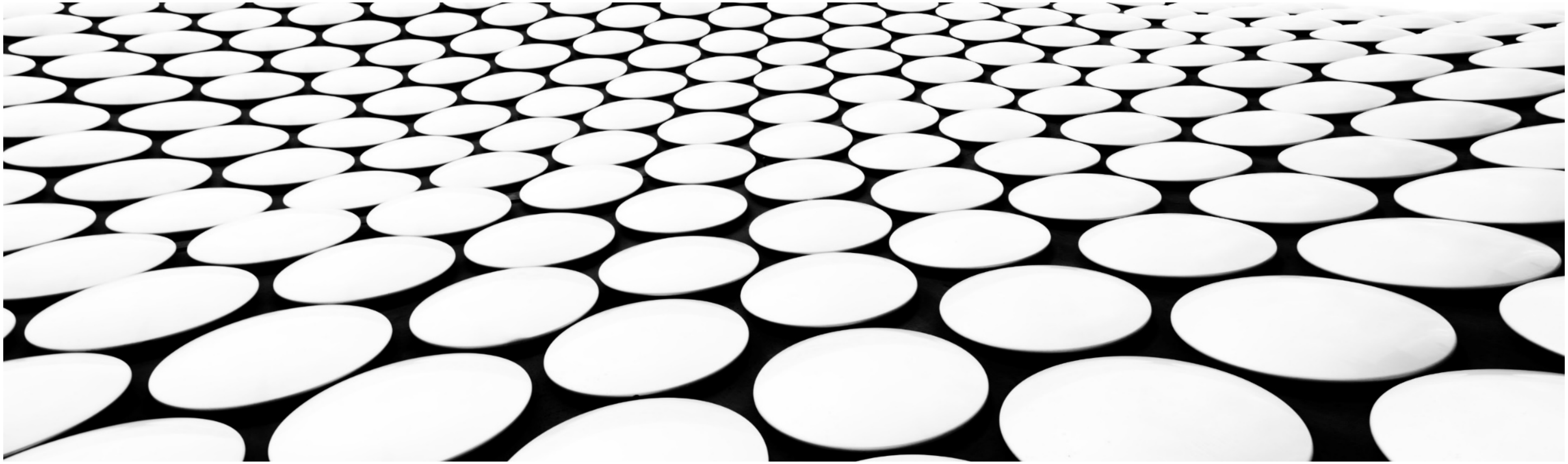
BIOINFORMATICS(BIOCOMPUTING)
(5)
Indexing

DR. IBRAHIM ZAGHLOUL



Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!

For bad character rule and *P* = TCGC:

P

Σ		T	C	G	C
	A				
	C				
	G				
	T				

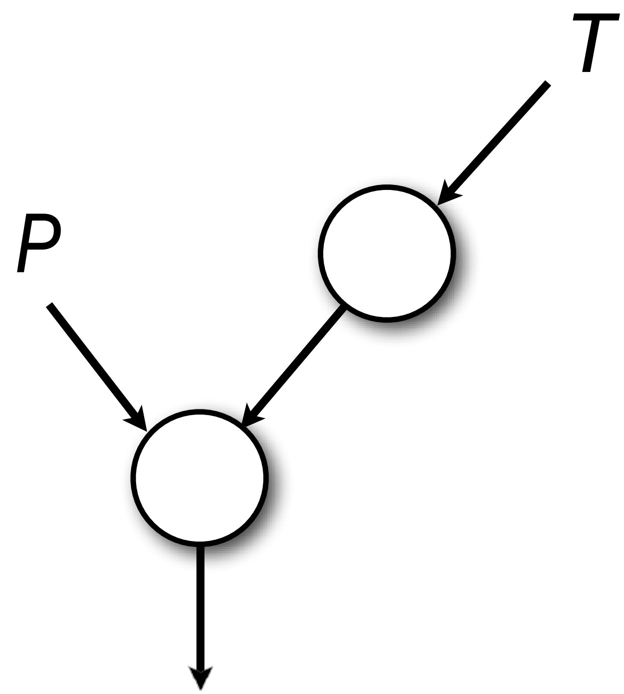
P

Σ		T	C	G	C
	A	0	1	2	3
	C	0	-	0	-
	G	0	1	-	0
	T	-	0	1	2

Indexing

Preprocessing

Algorithm that preprocesses T is *offline*.
Otherwise, algorithm is *online*.

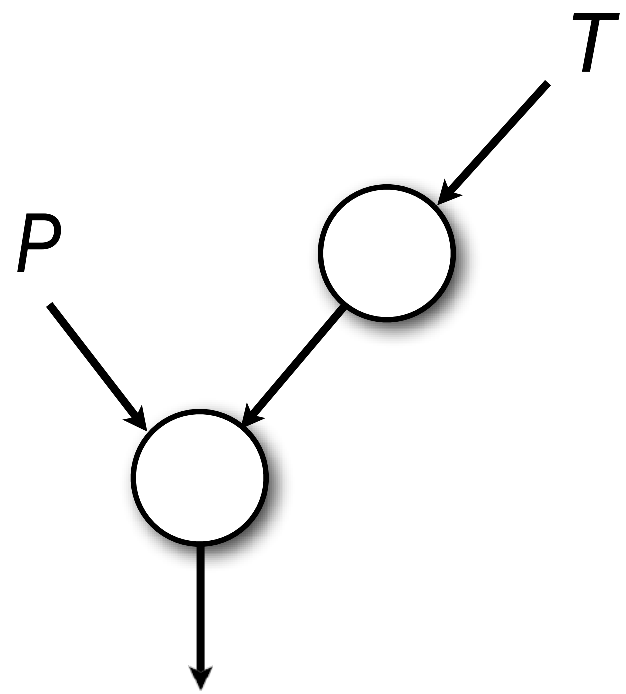


Online or offline?

- Naïve algorithm
- Boyer-Moore
- Web search engine
- Read alignment

Preprocessing

Algorithm that preprocesses *T* is *offline*.
Otherwise, algorithm is *online*.



Online or offline?

- Naïve algorithm
- Boyer-Moore
- Web search engine
- Read alignment

Index

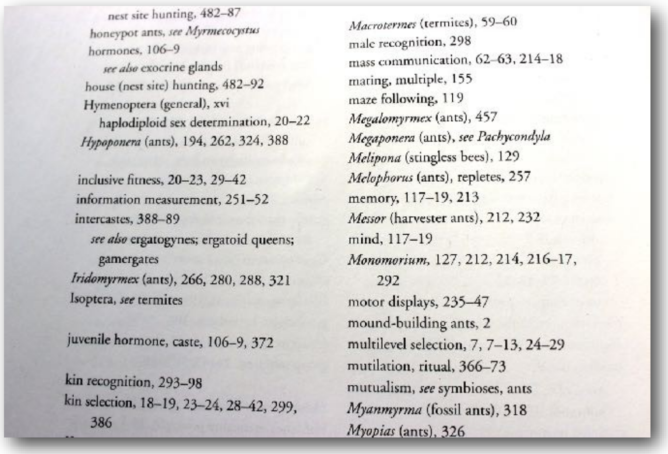
Key terms ordered alphabetically, with associated page #s

Index

Grocery store items grouped into aisles (Not Ordered)

Index

Indexes use *ordering* and *grouping* to make it easy to jump to relevant portions of the data



Indexing DNA

k-mer: substring of length k

Index of T

Substrings of length 5

T: C G T G C G T G C T T

Indexing DNA

Index of T

CGTGC: 0

Substrings of length 5

T: CGTGCGTGCTT

Indexing DNA

Index of T	
C G T G C :	**0**
G T G C G :	**1**

Substrings of length 5

T: C G T G C G T G C T T

Indexing DNA

Index of T

```
CGTGC:  0
GTGCG:  1
TGCGT:  2
```

T: C G T G C G T G C T T

Indexing DNA

Index of T

```
CGTGC:   0
GCGTG:   3
GTGCC:   1
TGCCT:   2
```

T: C G T G C G T G C T T

Indexing DNA

Index of T

```
CGTGC:    0,4
GCGTG:    3
GTGCC:    1
TGCCT:    2
```

T: C G T G C G T G C T T

Indexing DNA

Index of T

CGTGC:	0,4
GCGTG:	3
GTGCC:	1
GTGCT:	5
TGCCT:	2

T: C G T G C G T G C T T

Indexing DNA

Index of T

CGTGC:	0,4
GCGTG:	3
GTGCC:	1
GTGCT:	5
TGCCT:	2
TGCTT:	6

T: C G T G C G T G C T T

Indexing DNA

Index of T

```
CGTGC:    0,4
GCGTG:    3
GTGCC:    1
GTGCT:    5
TGCCT:    2
TGCTT:    6
```

5-mer index

T: **C G T G C G T G C T T**

Querying the index

Index of T

```
CGTGC:   0,4
GCGTG:   3
GTGCC:   1
GTGCT:   5
TGCCT:   2
TGCTT:   6
```

T: **C G T G C G T G C T T**

P: **G C G T G C**

Querying the index

Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

?

T: C G T G C G T G C T T

P: G C G T G C

Querying the index

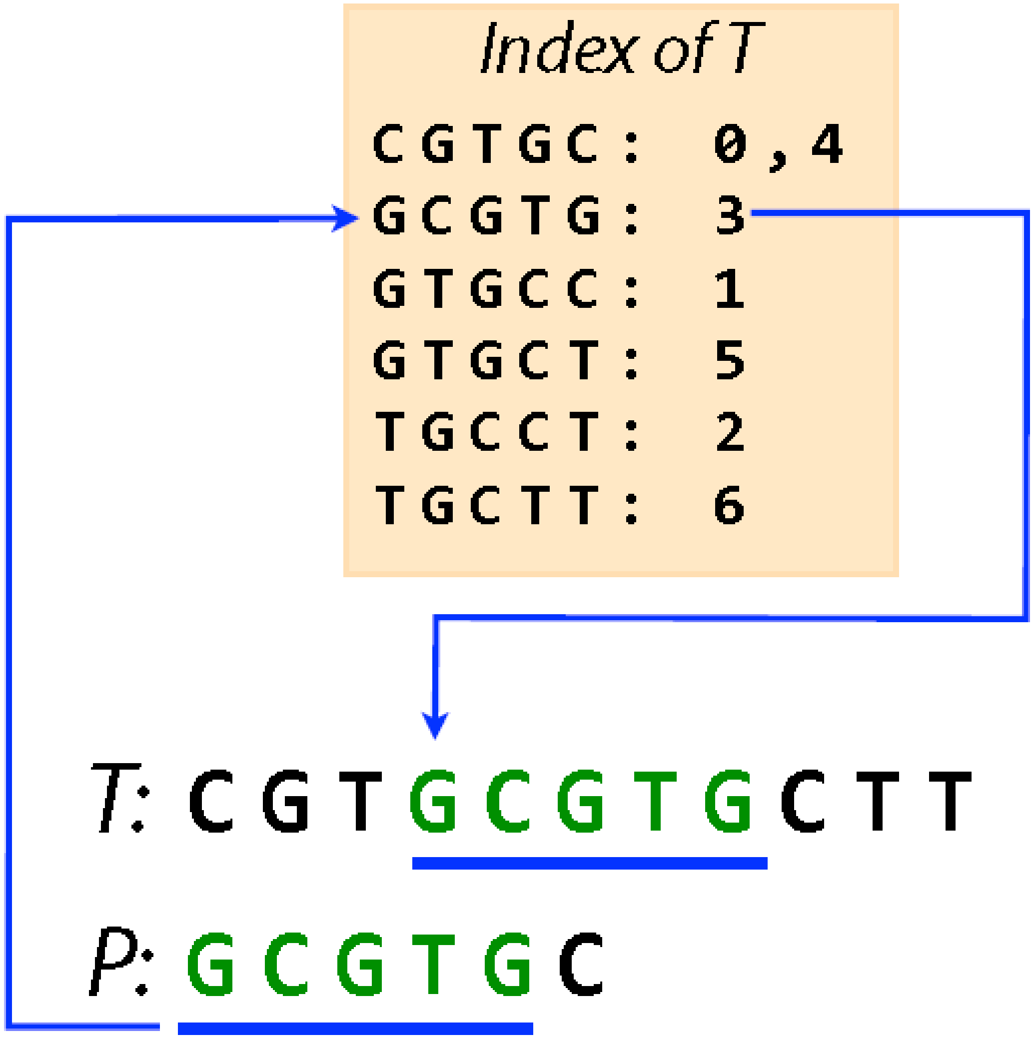
Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

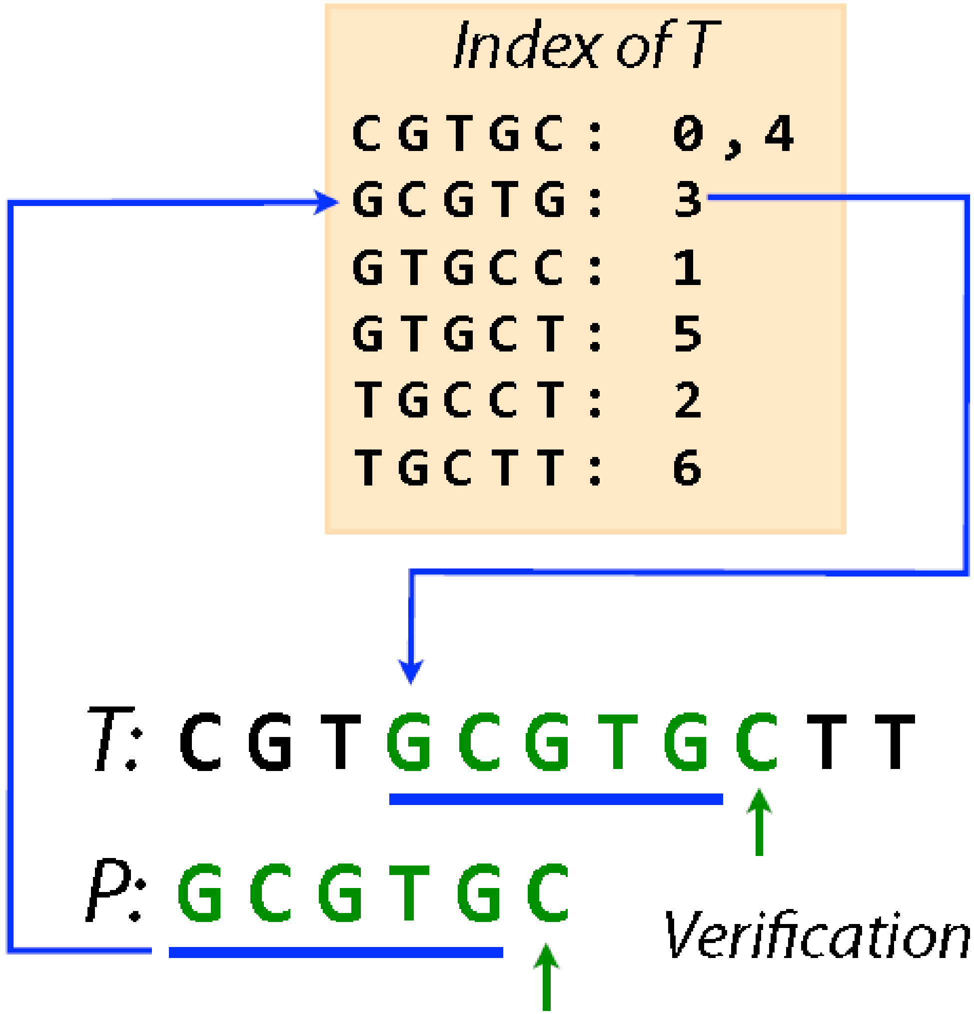
T: C G T G C G T G C T T

P: G C G T G C

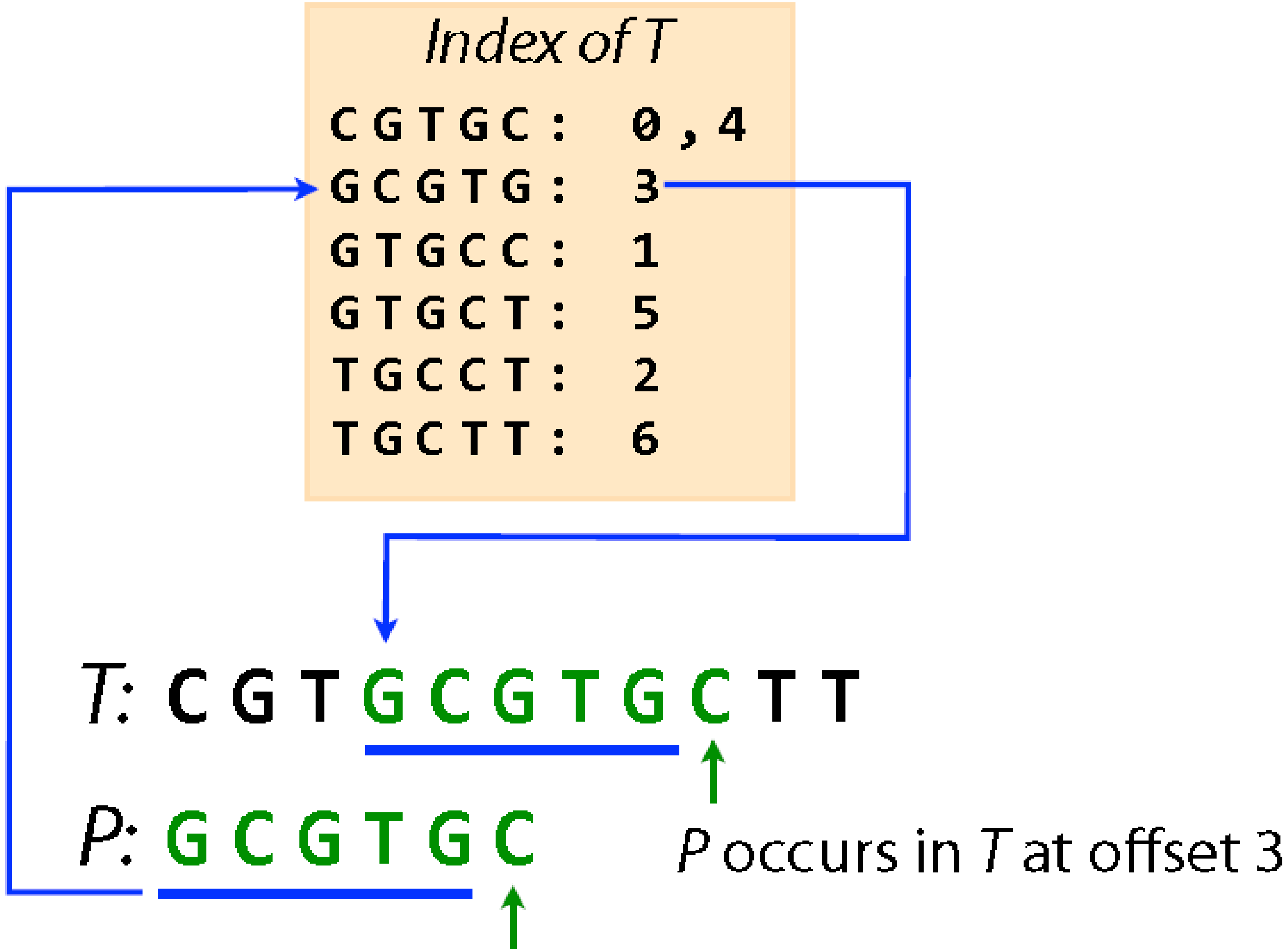
Querying the index



Querying the index



Querying the index



Querying the index

Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

?

T: C G T G C G T G C T T

P: G C G T G C

Querying the index

Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

T: C G T G C G T G C T T

P: G C G T G C

Querying the index

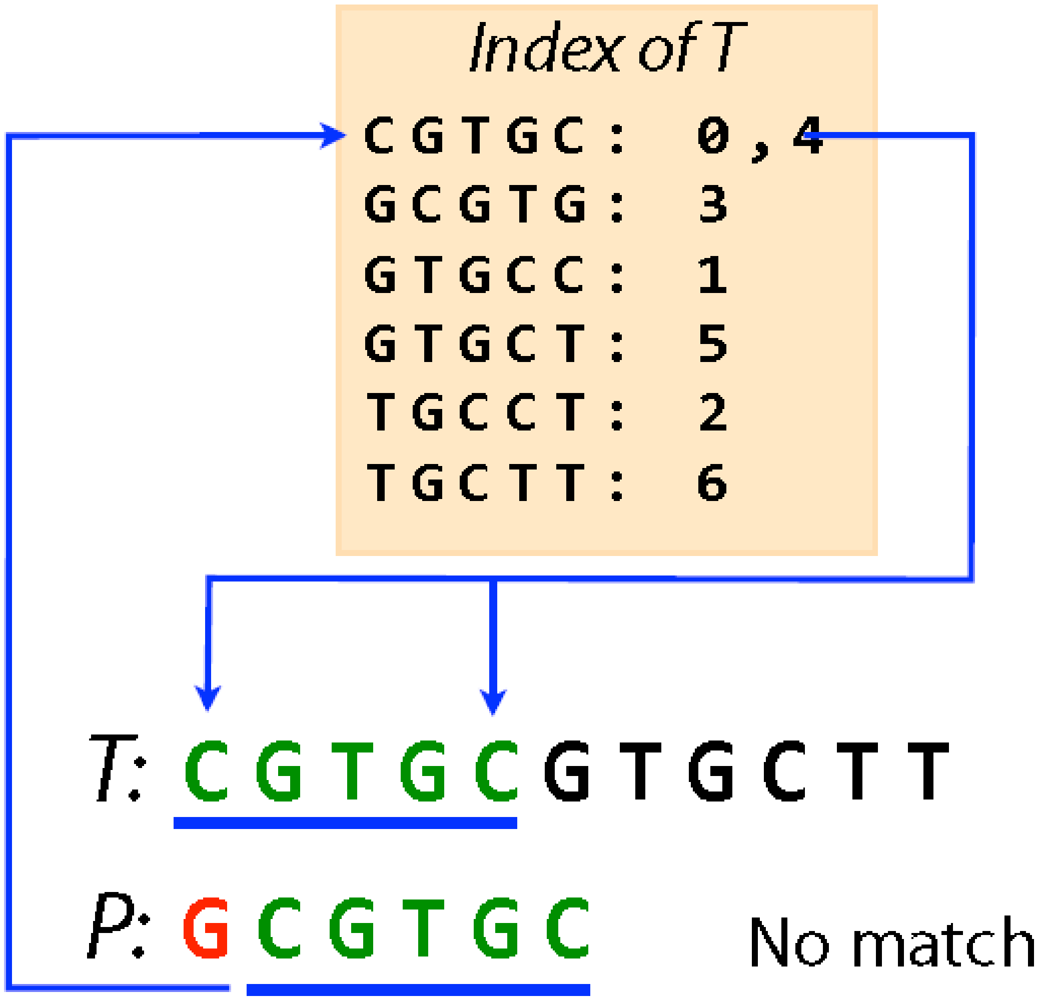
Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

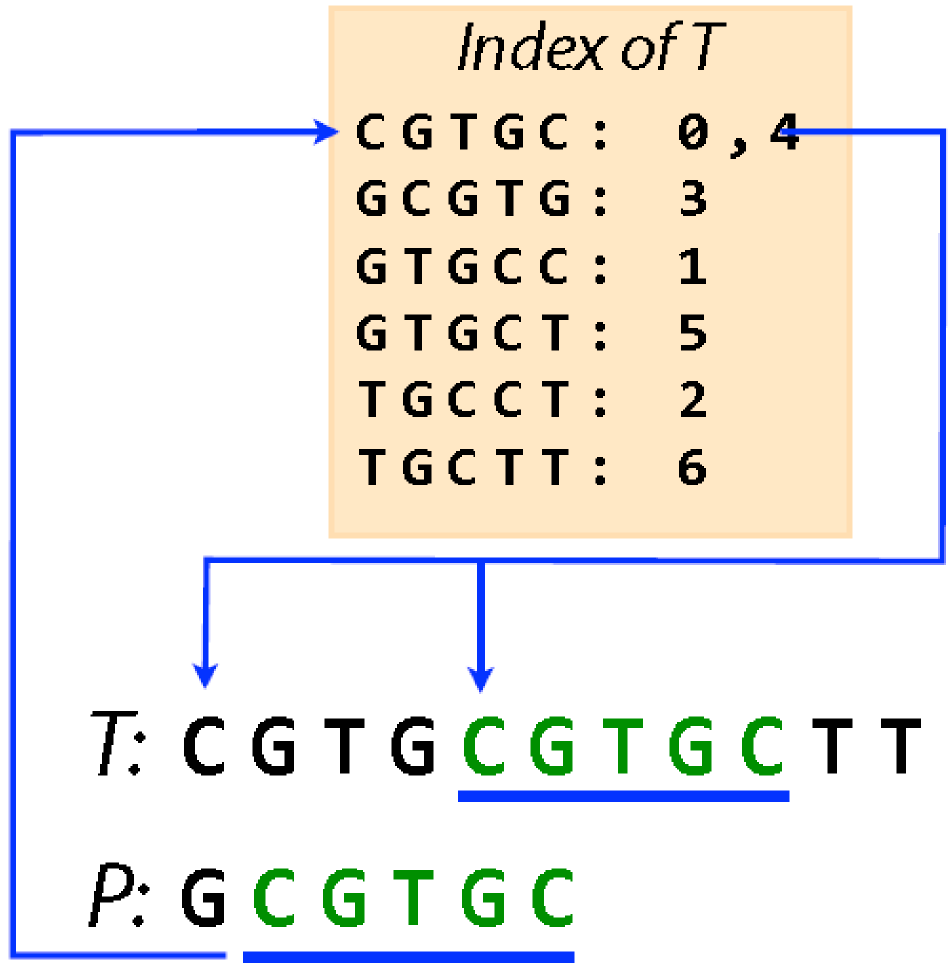
T: C G T G C G T G C T T

P: G C G T G C

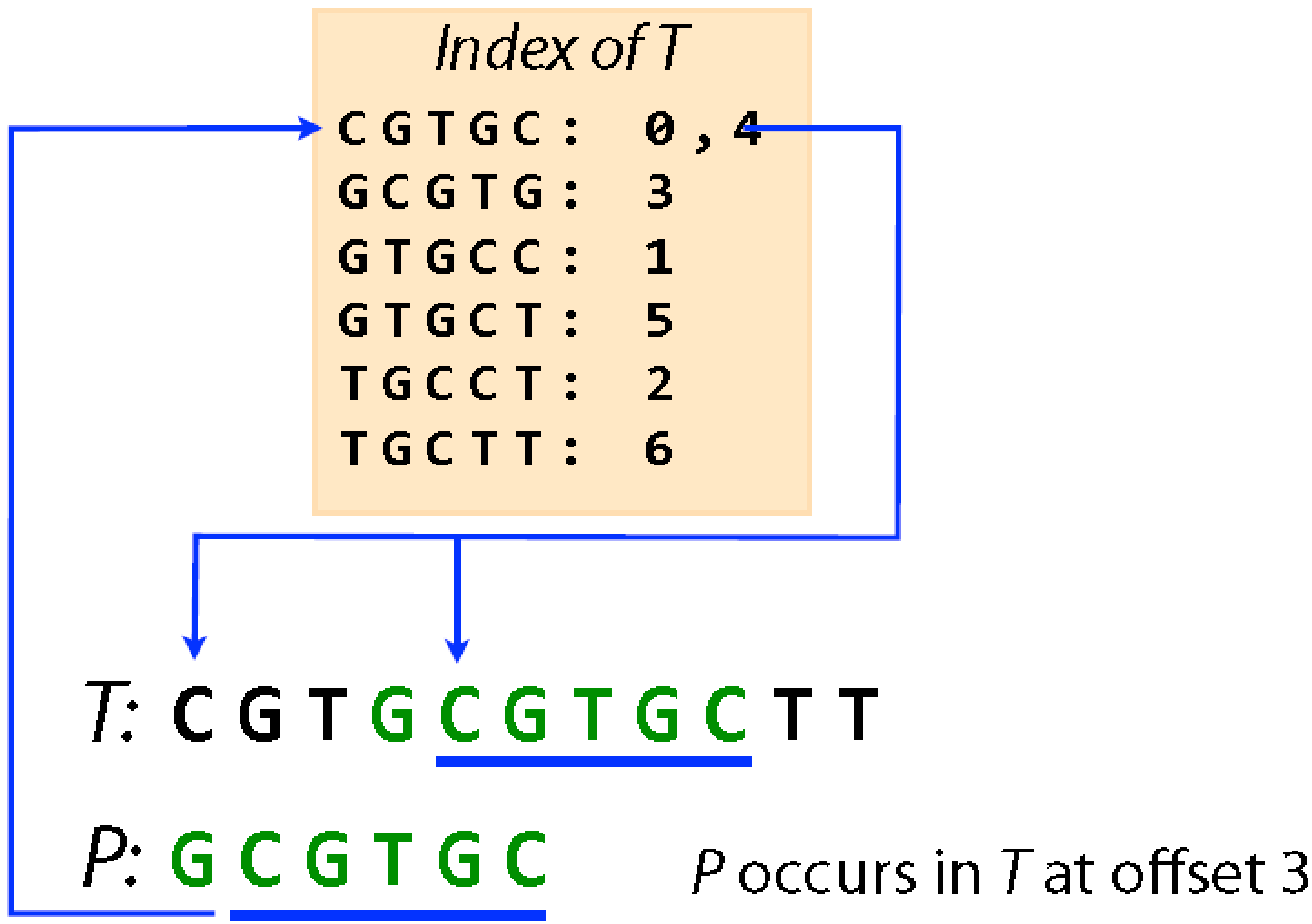
Querying the index



Querying the index



Querying the index



Querying the index

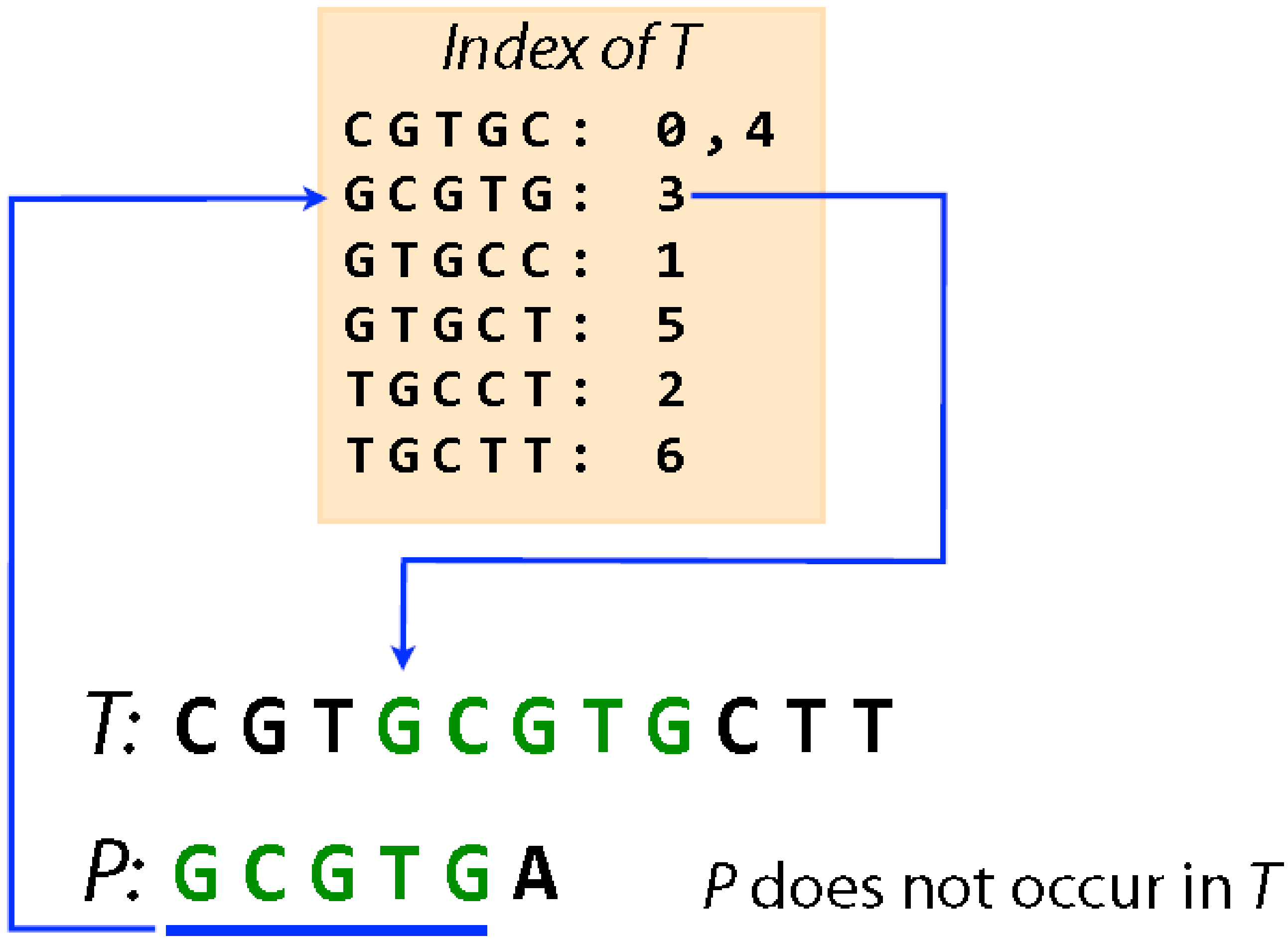
Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

T: C G T G C G T G C T T

P: G C G T G A

Querying the index



Querying the index

Index of T

```
CGTGC: 0,4
GCGTG: 3
GTGCC: 1
GTGCT: 5
TGCCT: 2
TGCTT: 6
```

T: C G T G C G T G C T T

P: G C G T A C

Querying the index

Index of T

CGTGC:	0,4
GCGTG:	3
GTGCC:	1
GTGCT:	5
TGCCT:	2
TGCTT:	6

T: C G T G C G T G C T T

P: G C G T A C

P does not occur in *T*

Querying the index

Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

1 index *hit*

T: C G T G C G T G C T T

P: G C G T G C

Querying the index

Index of T

```
CGTGC:  0,4
GCGTG:  3
GTGCC:  1
GTGCT:  5
TGCCT:  2
TGCTT:  6
```

2 index *hits*

T: C G T G C G T G C T T

P: G C G T G C

Data structures

Index of T

```
CGTGC:   0,4
GCGTG:   3
GTGCC:   1
GTGCT:   5
TGCCT:   2
TGCTT:   6
```

Abstractly, index is a *multimap* associating keys (k-mers) with one or more values (offsets)

What data structures allow us to represent and query a multimap?

Data structures

First idea: add key-value pairs to an array & sort the array

T: **G T G C G T G T G G G G G**

Data structures

3-mer

G T G	0

T: G T G C G T G T G G G G G

Data structures

GTG	0
TGC	1

T: G T G C G T G T G G G G G

Data structures

GTG	0
TGC	1
GCG	2

T: G T G C G T G T G G G G G

Data structures

GTG	0
TGC	1
GCG	2
CGT	3
GTG	4
TGT	5
GTG	6
TGG	7
GGG	8
GGG	9
GGG	10

T: **G T G C G T G T G G G G G**

Data structures

Alphabetical by k-mer

CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

T: **G T G C G T G T G G G G G**

Binary search

C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

T: G T G C G T G G G G G

P: G C G T G G

Binary search

CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

GCG < GTG

1st bisection

T: GTGCGTGGGGG

P: GCGTGG

Binary search

G C G < G G G

C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

2^{nd} bisection

T: G T G C G T G G G G G

P: G C G T G G

Binary search

GCG = GCG

CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

T: G T G C G T G G G G G

P: G C G T G G

Binary search

C G T	3
G C G	2
G G G	8
G G G	9
G G G	10
G T G	0
G T G	4
G T G	6
T G C	1
T G G	7
T G T	5

T: G T G C G T G G G G G

P: G C G T G G

Binary search

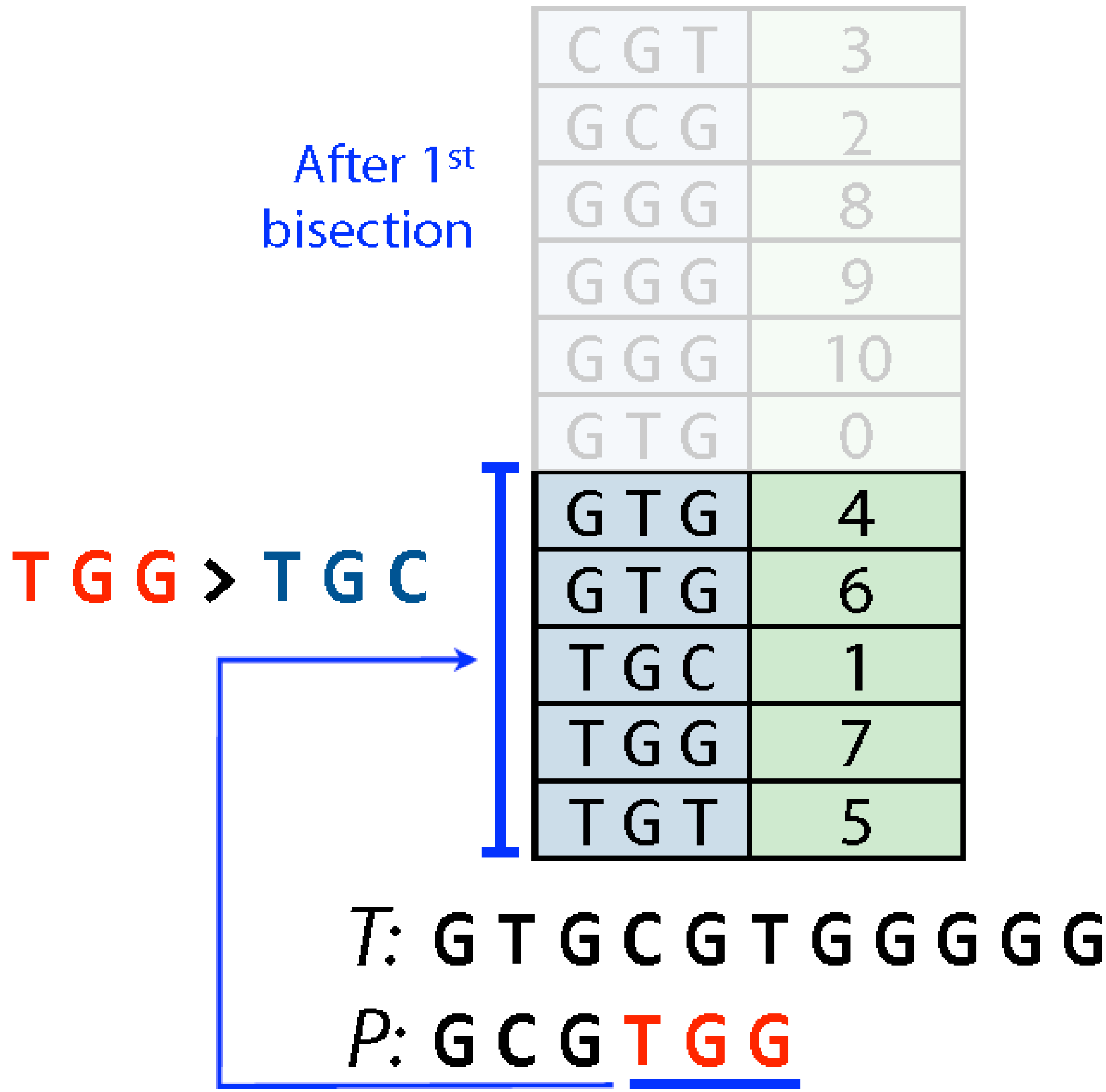
TGG > GTG

CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

T: GTGCGTGGGGG

P: GCGTGG

Binary search



Binary search

After 2nd bisection

TGG = TGG

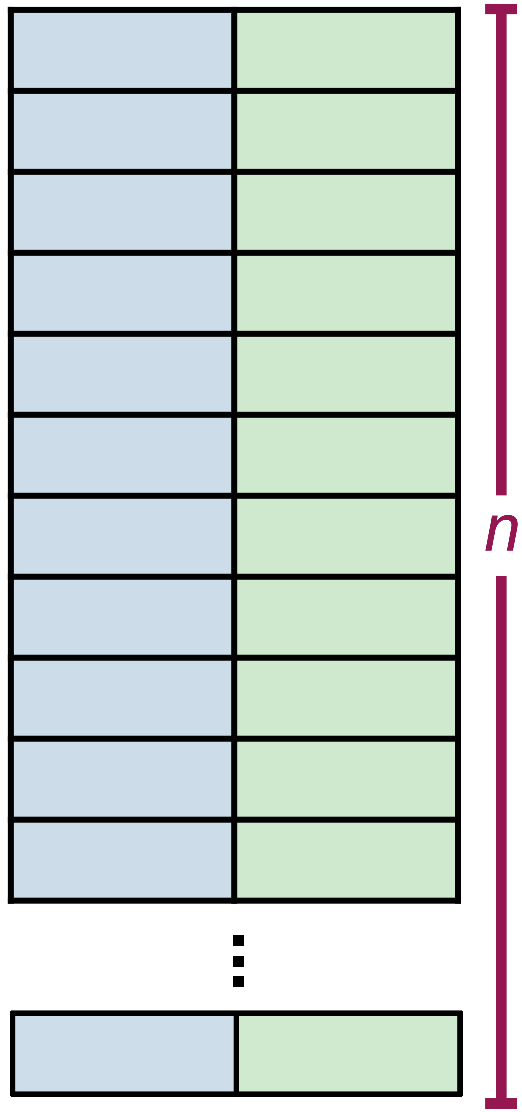
CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

T: G T G C G T G G G G G

P: G C G T G G

Binary search

About how many bisections in the worst case, as a function of *n*?



~ $\log_2(n)$ bisections

```
bisect_left(index, 'GTG')
```

CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

T: **G T G C G T G T G G G G G,**

P: **G C <u>G T G</u> G**

Suffix arrays

- The *suffix array* of a given *string* of length n (including a *sentinel* $) is an integer array containing the *suffix IDs* of the lexicographically sorted suffixes of the *original string.* ($ is considered the smallest character)
- A *suffix ID* is the start index of this suffix inside the *original string*.
- The *suffix array slower but more compact* than the *suffix tree*.
- Consider the suffixes of the string `ACGACTACGATAAC$` of length $n = 15$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	C	G	A	C	T	A	C	G	A	T	A	A	C	$

Suffix ID	Suffix string
0	ACGACTACGATAAC$
1	CGACTACGATAAC$
2	GACTACGATAAC$
3	ACTACGATAAC$
4	CTACGATAAC$
5	TACGATAAC$
6	ACGATAAC$
7	CGATAAC$
8	GATAAC$
9	ATAAC$
10	TAAC$
11	AAC$
12	AC$
13	C$
14	$

Suffix ID	Suffix string
14	$
11	AAC$
12	AC$
0	ACGACTACGATAAC$
6	ACGATAAC$
3	ACTACGATAAC$
9	ATAAC$
13	C$
1	CGACTACGATAAC$
7	CGATAAC$
4	CTACGATAAC$
2	GACTACGATAAC$
8	GATAAC$
10	TAAC$
5	TACGATAAC$

Therefore, the *suffix array* of the string ACGACTACGATAAC$ is:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Suffix array	14	11	12	0	6	3	9	13	1	7	4	2	8	10	5

ACGACTACGATAAC$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	C	G	A	C	T	A	C	G	A	T	A	A	C	$

The third column of *suffix* strings in the shown table is not part of the *suffix array* and is shown for illustration only since it can be deduced easily from the *suffix array* and the *original string*.

Index	Suffix array	Corresponding suffix
0	14	$
1	11	AAC$
2	12	AC$
3	0	ACGACTACGATAAC$
4	6	ACGATAAC$
5	3	ACTACGATAAC$
6	9	ATAAC$
7	13	C$
8	1	CGACTACGATAAC$
9	7	CGATAAC$
10	4	CTACGATAAC$
11	2	GACTACGATAAC$
12	8	GATAAC$
13	10	TAAC$
14	5	TACGATAAC$

- Here we trace the *binary search* for the substring `CGA` using the above *suffix array* only.
- We start with an unexplored interval [0, 15] representing:

[*first index, last index* + 1].

- Middle index is $\lfloor(0 + 15)/2\rfloor = 7$. `CGA` > `C$`. Interval shrinks to [8, 15].
- Middle index is $\lfloor(8 + 15)/2\rfloor = 11$. `CGA` < `GACTACGATAAC$`. Interval shrinks to [8, 11].
- Middle index is $\lfloor(8 + 11)/2\rfloor = 9$. `CGA` < `CGATAAC$`. Interval shrinks to [8, 9].
- Middle index is $\lfloor(8 + 9)/2\rfloor = 8$. `CGA` < `CGACTACGATAAC$`. Interval shrinks to [8, 8].
- Then, we test if `CGA` is prefix of suffixes at indexes ≥ 8 in *suffix array*:
- `CGA` is prefix of `CGACTACGATAAC$` at index 8. Report occurrence at index 1 in *original string*.
- `CGA` is prefix of `CGATAAC$` at index 9. Report occurrence at index 7 in *original string*.
- `CGA` is not prefix of `CTACGATAAC$` at index 10. Stop.

Index	Suffix array	Corresponding suffix
0	14	$
1	11	AAC$
2	12	AC$
3	0	ACGACTACGATAAC$
4	6	ACGATAAC$
5	3	ACTACGATAAC$
6	9	ATAAC$
7	13	C$
8	1	CGACTACGATAAC$
9	7	CGATAAC$
10	4	CTACGATAAC$
11	2	GACTACGATAAC$
12	8	GATAAC$
13	10	TAAC$
14	5	TACGATAAC$

Suffix array construction

- Consider constructing *suffix array* of string ACGACTACGATAAC$ using *prefix doubling*.
- The initial step is to sort all *suffixes* by their first character only, simply by assigning to each *suffix* the order of its first character in the alphabet. (Remember that $ is the smallest character)

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0

From the above table, we recognize that the smallest *suffix* $ gets the smallest integer 0. The immediately larger *suffixes* are those starting with A. They all got the next smallest integer 1, because they are equal if we look at their first character only which is A.

Suffix array construction

- The general rule is that, in iteration *i*, all *suffixes* are sorted according to their first 2^i characters only.
- That is, we assume that the length of each suffix is only 2^i.
- All *suffixes* starting with the same prefix of size 2^i are considered equal and assigned the same integer.
- Thus, the second iteration $i = 1$ assigns the same integer to all *suffixes* starting with the same $2^1 = 2$ characters:

Suffix ID	Suffix string
0	ACGACTACGATAAC$
1	CGACTACGATAAC$
2	GACTACGATAAC$
3	ACTACGATAAC$
4	CTACGATAAC$
5	TACGATAAC$
6	ACGATAAC$
7	CGATAAC$
8	GATAAC$
9	ATAAC$
10	TAAC$
11	AAC$
12	AC$
13	C$
14	$

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1 AC	2	3	AC 1	2	4	AC 1	2	3	AT 1	4	1 AA	AC 1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0

Suffix array construction

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0

- From the above table, we recognize that the smallest *suffix* $ gets the smallest integer **0**.
- The immediately larger *suffixes* are those starting with AA. It is exactly one *suffix* and it got the next smallest integer **1**.
- The immediately larger *suffixes* are those starting with AC. They all got the next smallest integer **2**, because they are equal if we look at their first two characters only which are AC.

Suffix array construction

- The next iteration $i = 2$ of the algorithm is going to sort *suffixes* according to their first $2^i = 2^2 = 4$ characters.
- Instead of comparing two *suffixes* by performing a string comparison of their first 4 characters, we will perform a more efficient *suffix* comparison using the results of the previous iteration $i = 1$.
- For example, to compare the first 4 characters of the two *suffixes* at indexes 4 (CTAC) and 7 (CGAT), look at their relative order according to their first 2 characters, appearing in last row in the above table to be 6 and 5, indicating that *suffix* 4 is larger than *suffix* 7. The relation remains the same.
- Another example for the other case, to compare the first 4 characters of the two *suffixes* at indexes 2 (GACT) and 8 (GATA). Their orders according to their first 2 characters, appearing in last row in the above table are 7 and 7, indicating that *suffix* 2 is equal to *suffix* 8 with respect to the first 2 characters (GA).

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0

- Compare the first 4 characters of the two *suffixes* at indexes 2 (GACT) and 8 (GATA). Their orders according to their first 2 characters, appearing in last row in the above table are 7 and 7, indicating that *suffix* 2 is equal to *suffix* 8 with respect to the first 2 characters (GA).
- Since they are equal, we consider the two *suffixes* shifted by 2 from the original *suffixes* indexes, which are *suffixes* at indexes 2 + 2 = 4 (CT) and 8 + 2 = 10 (TA).
- Their orders according to their first 2 characters, appearing in last row in the above table are 6 and 8, indicating that *suffix* 4 is smaller than *suffix* 10 with respect to their first 2 characters (GA), which implies the same relation between the original two *suffixes* 2 (GACT) and 8 (GATA) with respect to their first 4 characters.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iteration	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0

Suffix array construction

- Here we explain how to obtain all *suffix* orders from of iteration 2 from iteration 1.
- There is exactly one *suffix* with order 0 which is *suffix* 14, its order remains the same. Also, only *suffix* 11 has order 1 and remains the same.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$												1			0

- There are 4 *suffixes* with order 2 which are 0, 3, 6, 12.
- We look at shifted-by-2 *suffixes* 2, 5, 8, 14 their orders are 7, 8, 7, 0 to conclude that the smallest *suffix* is **12** so we assign to it order of **2** (because last assigned order was 1).
- Then, next smallest *suffixes* are **0** and **6** with the same order of **3**, meaning that they are still equal with respect to their first 4 characters.
- Then *suffix* **3** takes order of **4** (because last assigned order was 3).

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$	**3**			**4**			**3**					1	**2**		0

- Only *suffix* **9** has order **3** in iteration **1**. It is assigned order **5** in iteration **2** (because last assigned order was **4**).
- Only *suffix* **13** has order **4**. It is assigned order **6**.
- There are **2** *suffixes* with order **5** which are **1, 7**. We look at shifted-by-2 *suffixes* **3, 9** their orders are **2, 3** to conclude that the smaller *suffix* is **1** so we assign to it order of **7**, then *suffix* **7** takes order of **8**.
- Only *suffix* **4** has order **6**. It is assigned order **9**.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$	3	7		4	9		3	8		5		1	2	6	0

- There are 2 *suffixes* with order 7 which are 2, 8. We look at shifted-by-2 *suffixes* 4, 10 their orders are 6, 8 to conclude that the smaller *suffix* is **2** so we assign to it order of **10**, then *suffix* **8** takes order of **11**.
- There are 2 *suffixes* with order 8 which are 5, 10. We look at shifted-by-2 *suffixes* 7, 12 their orders are 5, 1 to conclude that the smaller *suffix* is **10** so we assign to it order of **12**, then *suffix* **5** takes order of **13**.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$	3	7	10	4	9	13	3	8	11	5	12	1	2	6	0

- Note that actually there should not be any two equal *suffixes*, so the algorithm terminates only if there are no two *suffixes* with the same order.

- To move to the next iteration 3, the only two suffixes with same order are 0, 6.
- We look at shifted-by-4 *suffixes* 4, 10 their orders in iteration 2 are 9, 12 to conclude that the smaller *suffix* is **0** so we assign to it a smaller order than suffix **6** as follows:

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$	3	7	10	4	9	13	3	8	11	5	12	1	2	6	0
3	$2^3 = 8$	**3**	**8**	**11**	**5**	**10**	**14**	**4**	**9**	**12**	**6**	**13**	1	**2**	**7**	0

- The algorithm terminates because all suffixes have different orders as they should.
- Since we terminated at iteration 3 we conclude that no two *suffixes* share the same prefix of $2^3 = 8$ characters.

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Iter	Sorted prefix len	A	C	G	A	C	T	A	C	G	A	T	A	A	C	$
0	$2^0 = 1$	1	2	3	1	2	4	1	2	3	1	4	1	1	2	0
1	$2^1 = 2$	2	5	7	2	6	8	2	5	7	3	8	1	2	4	0
2	$2^2 = 4$	3	7	10	4	9	13	3	8	11	5	12	1	2	6	0
3	$2^3 = 8$	3	8	11	5	10	14	4	9	12	6	13	1	2	7	0

The resulting array is not the *suffix array*, but it is the *inverse* of the *suffix array*.

The resulting array tells the order given a *suffix ID* (example: suffix 12 has the order 2).

The *suffix array* tells the *suffix ID* given an order (example: the suffix of order 2 is 12).

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Suffix array	14	11	12	0	6	3	9	13	1	7	4	2	8	10	5

Index	Suffix array	Corresponding suffix
0	14	$
1	11	AAC$
2	12	AC$
3	0	ACGACTACGATAAC$
4	6	ACGATAAC$
5	3	ACTACGATAAC$
6	9	ATAAC$
7	13	C$
8	1	CGACTACGATAAC$
9	7	CGATAAC$
10	4	CTACGATAAC$
11	2	GACTACGATAAC$
12	8	GATAAC$
13	10	TAAC$
14	5	TACGATAAC$