Algorithms for NGS

Raluca Uricaru

Topics covered by this class :

1. Sequencing data and what we do with it
2. Read alignment as a pattern matching problem
3. Assembly problem, two solutions
    a. Overlap Graphs
    b. De Bruijn Graphs

Sequencing data and what we do with it

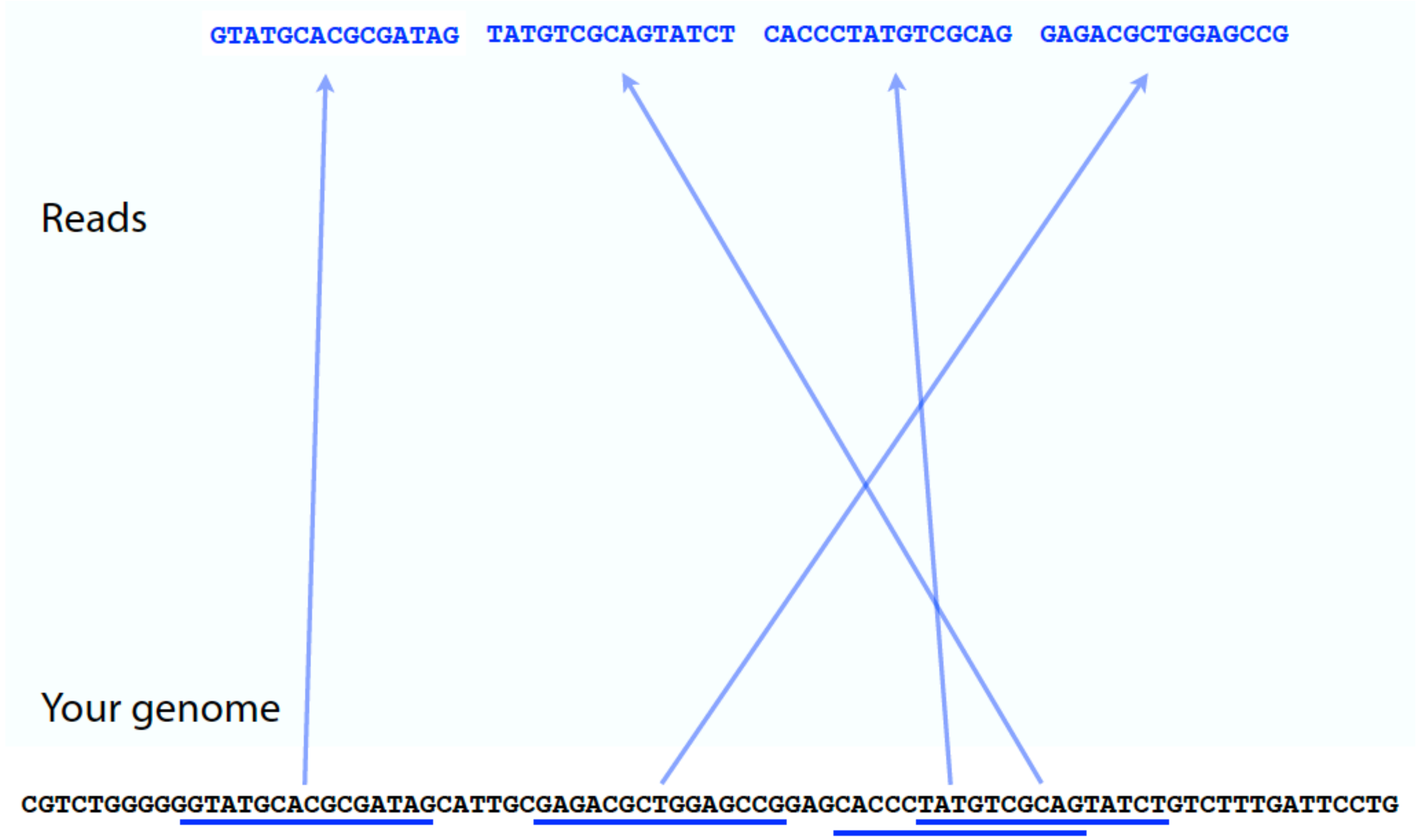
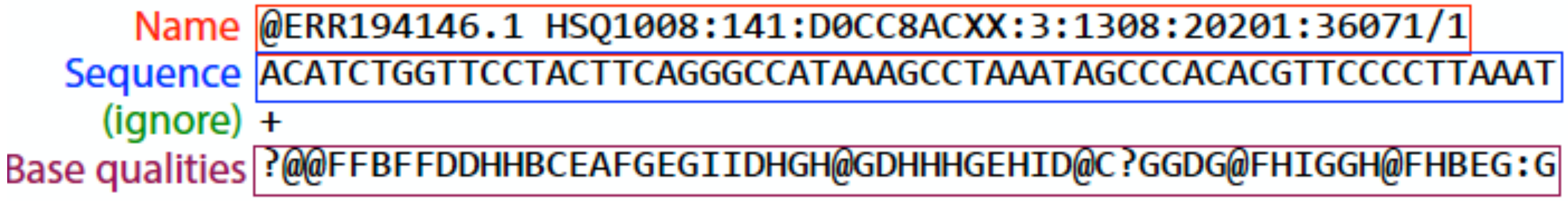
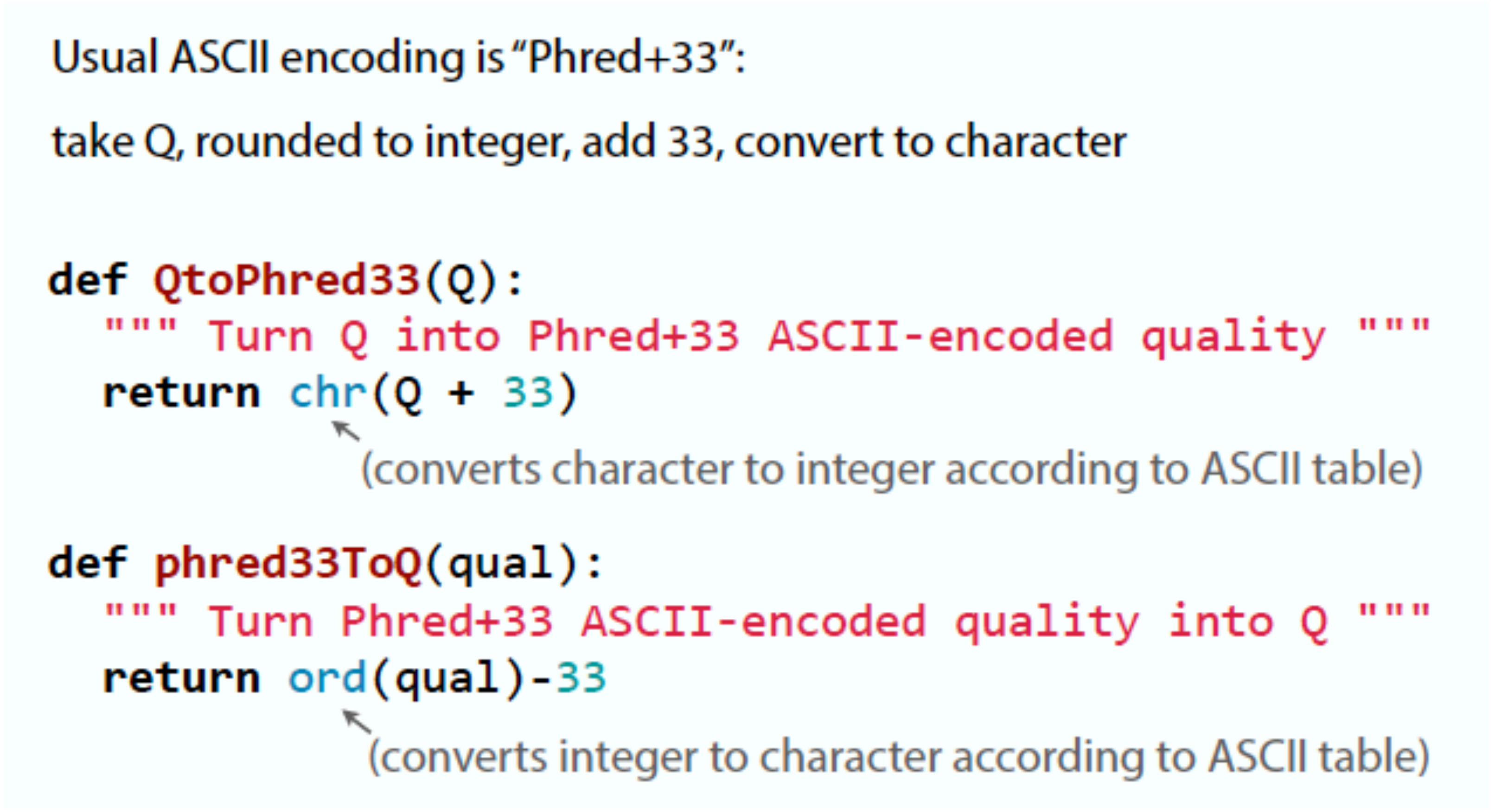

GTATGCACGCGATAG
TATGTCGCAGTATCT
CACCCTATGTCGCAG
GAGACGCTGGAGCCG
Reads
Your genome
CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTATGTCGCAGTATCTGTCTTTGATTCCTG

image extracted from [1]

Fastq format



```
Name            @ERR194146.1 HSQ1008:141:D0CC8ACXX:3:1308:20201:36071/1
Sequence        ACATCTGGTTCCTACTTCAGGGCCATAAAGCCTAAATAGCCCACACGTTCCCCTTAAAT
(ignore)        +
Base qualities  ?@@FFBFFDDHHBCEAFGEGIIDHGH@GDHHHGEHID@C?GGDG@FHIGGH@FHBEG:G
```



Usual ASCII encoding is "Phred+33":

take Q, rounded to integer, add 33, convert to character

```
def QtoPhred33(Q):
  """ Turn Q into Phred+33 ASCII-encoded quality """
  return chr(Q + 33)
```

(converts character to integer according to ASCII table)

```
def phred33ToQ(qual):
  """ Turn Phred+33 ASCII-encoded quality into Q """
  return ord(qual)-33
```

(converts integer to character according to ASCII table)

images extracted from [1]

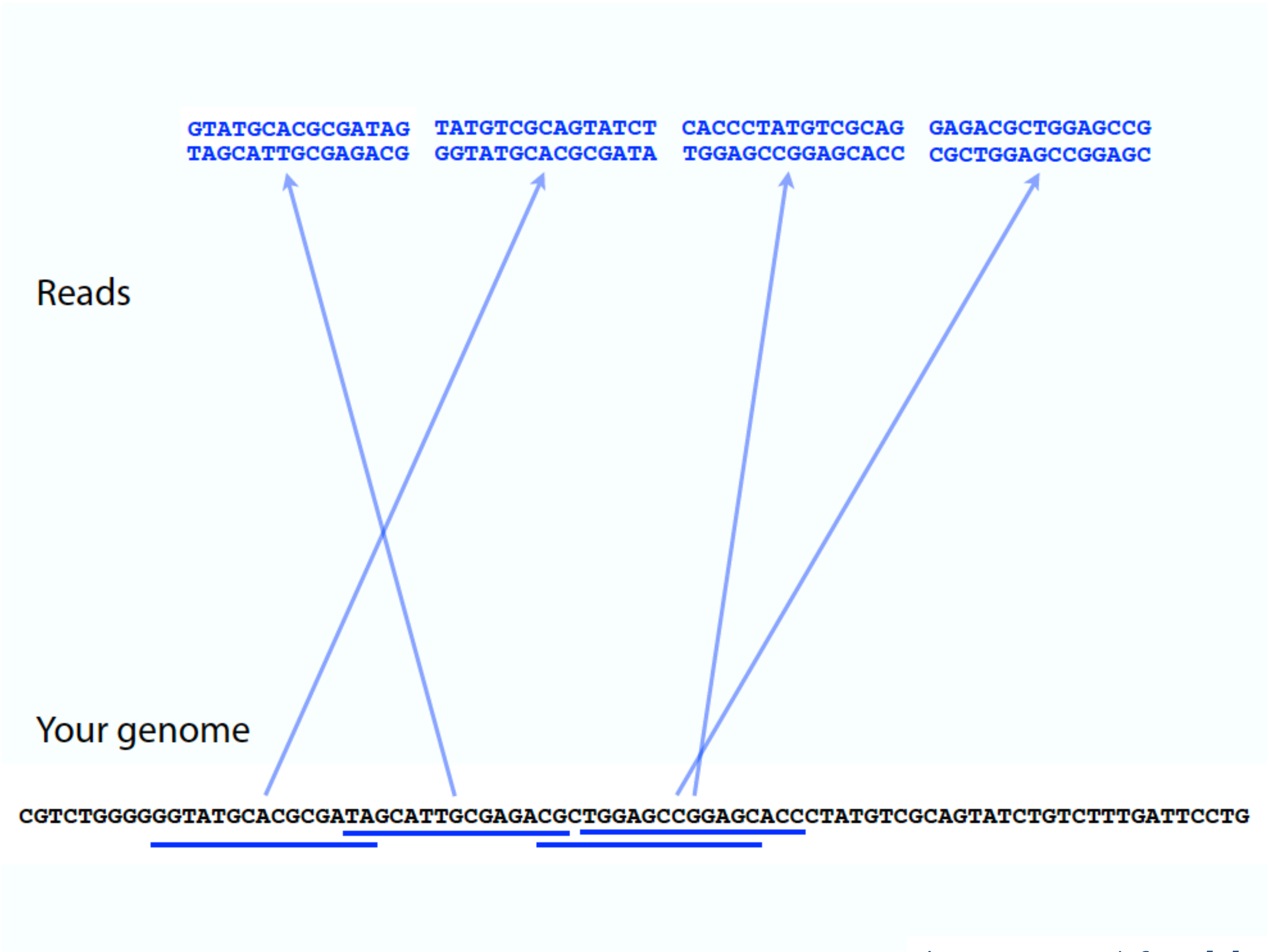


image extracted from [1]

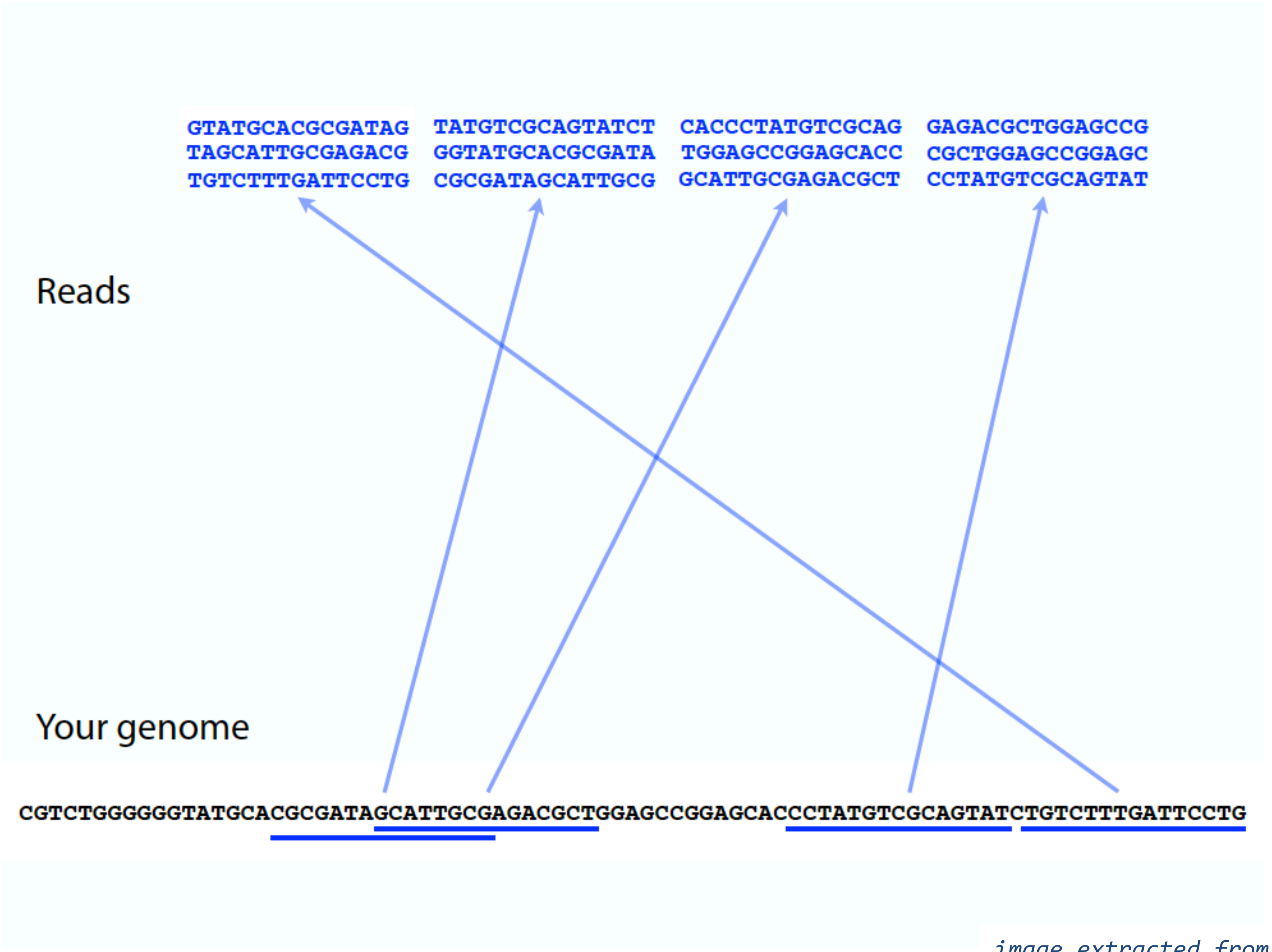


image extracted from [1]

Reads

GTATGCACGCGATAG	TATGTCGCAGTATCT	CACCCTATGTCGCAG	GAGACGCTGGAGCCG
TAGCATTGCGAGACG	GGTATGCACGCGATA	TGGAGCCGGAGCACC	CGCTGGAGCCGGAGC
TGTCTTTGATTCCTG	CGCGATAGCATTGCG	GCATTGCGAGACGCT	CCTATGTCGCAGTAT
GACGCTGGAGCCGGA	GCACCCTATGTCGCA	GTATCTGTCTTTGAT	CCTCATCCTATTATT
TATCGCACCTACGTT	CAATATTCGATCATG	GATCACAGGTCTATC	ACCCTATTAACCACT
CACGGGAGCTCTCCA	TGCATTTGGTATTTT	CGTCTGGGGGGTATG	CACGCGATAGCATTG
GTATGCACGCGATAG	ACCTACGTTCAATAT	TATTTATCGCACCTA	CCACTCACGGGAGCT
GCGAGACGCTGGAGC	CTATCACCCTATTAA	CTGTCTTTGATTCCT	ACTCACGGGAGCTCT
CCTACGTTCAATATT	GCACCTACGTTCAAT	GTCTGGGGGGTATGC	AGCCGGAGCACCCTA
GACGCTGGAGCCGGA	GCACCCTATGTCGCA	GTATCTGTCTTTGAT	CCTCATCCTATTATT
TATCGCACCTACGTT	CAATATTCGATCATG	GATCACAGGTCTATC	ACCCTATTAACCACT
CACGGGAGCTCTCCA	TGCATTTGGTATTTT	CGTCTGGGGGGTATG	CACGCGATAGCATTG

Your genome

CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTATGTCGCAGTATCTGTCTTTGATTCCTG

image extracted from [1]

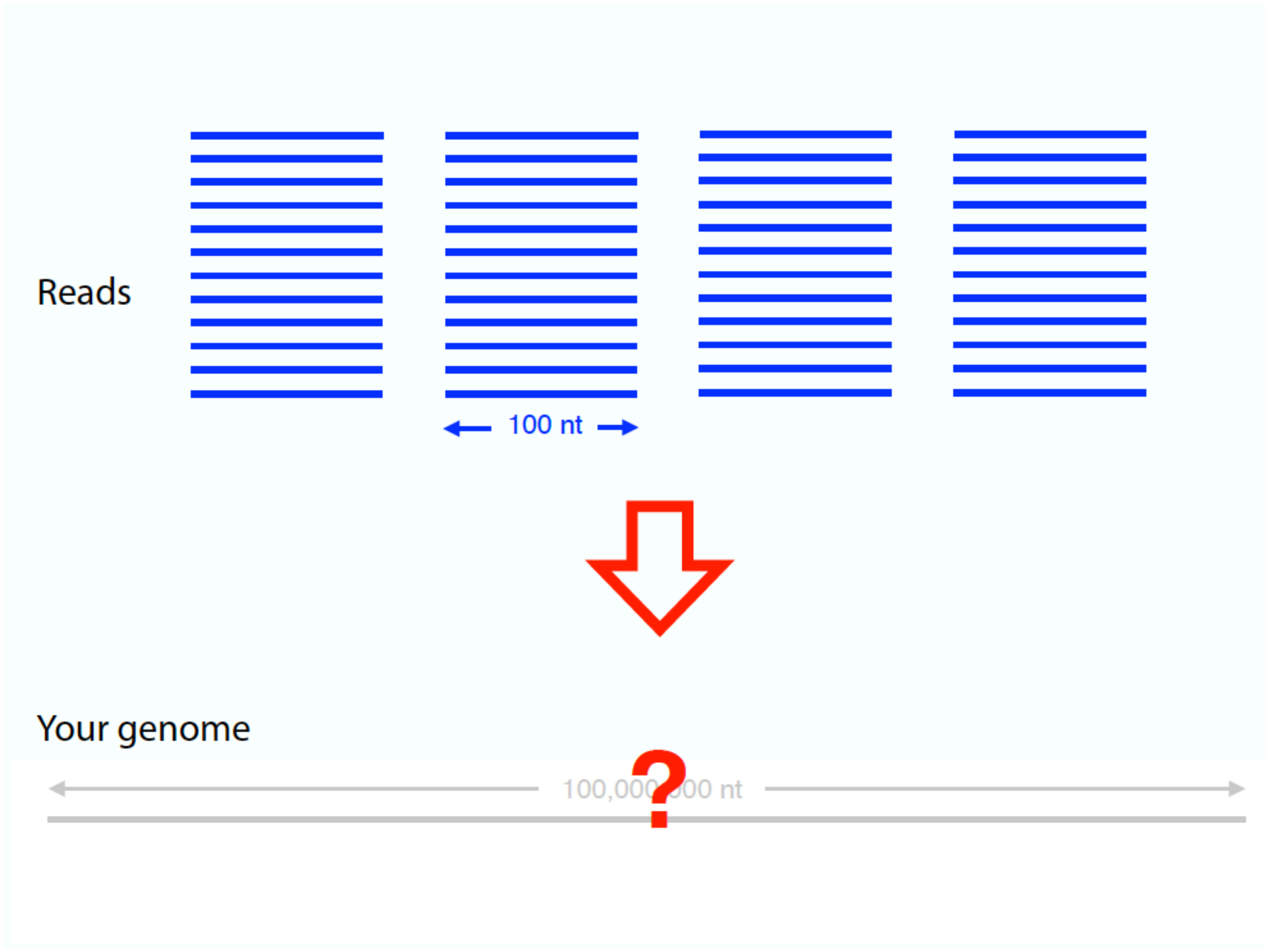


image extracted from [1]

Alignment

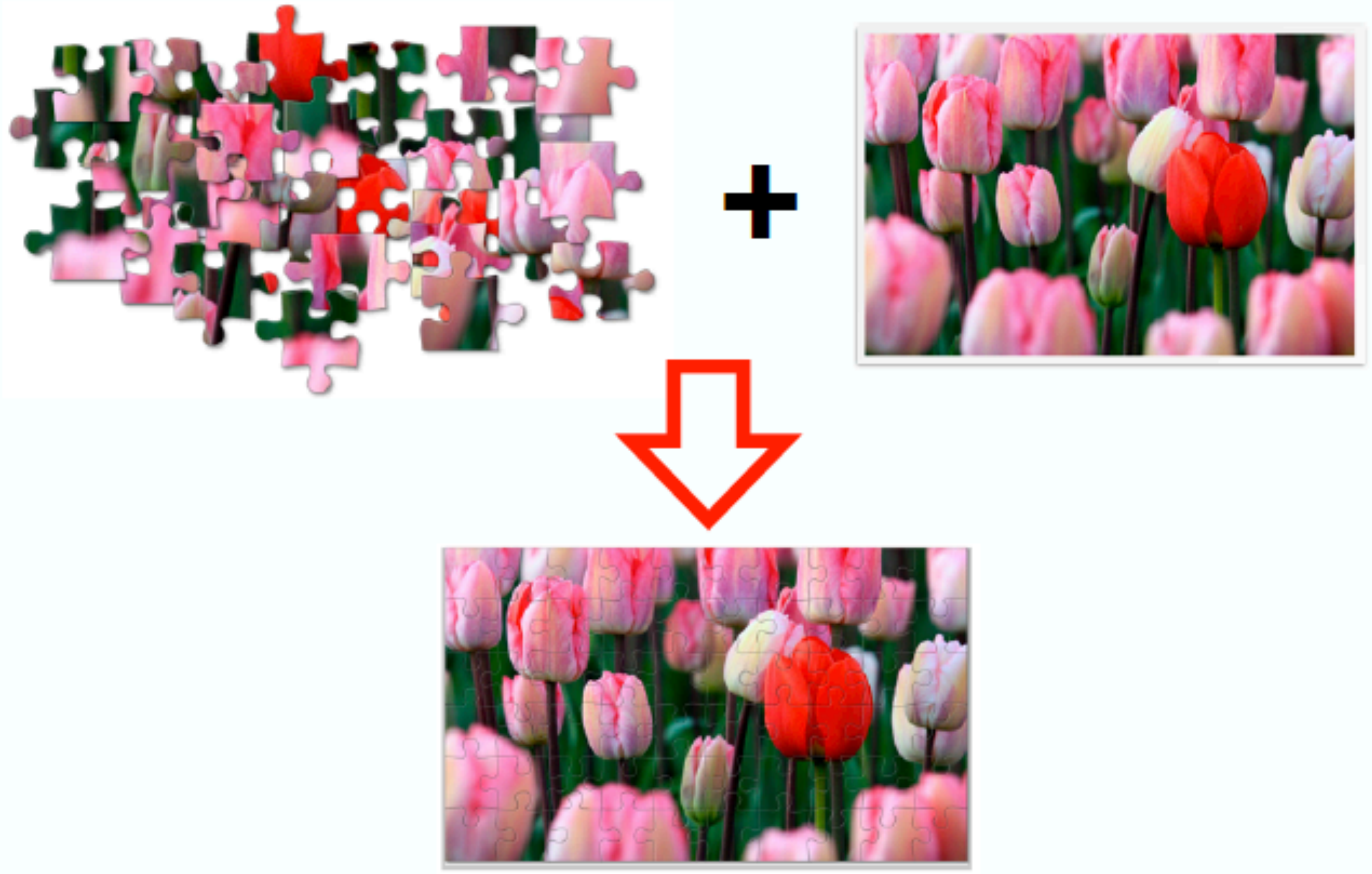


image extracted from [1]

Read alignment (mapping) applications :

- Genotyping : differences between you and me
- RNA-seq, ChIP-seq, Methyl-seq to measure significant *peaks* (enriched areas) corresponding to DNA fragments being representative of a biological process

Assembly

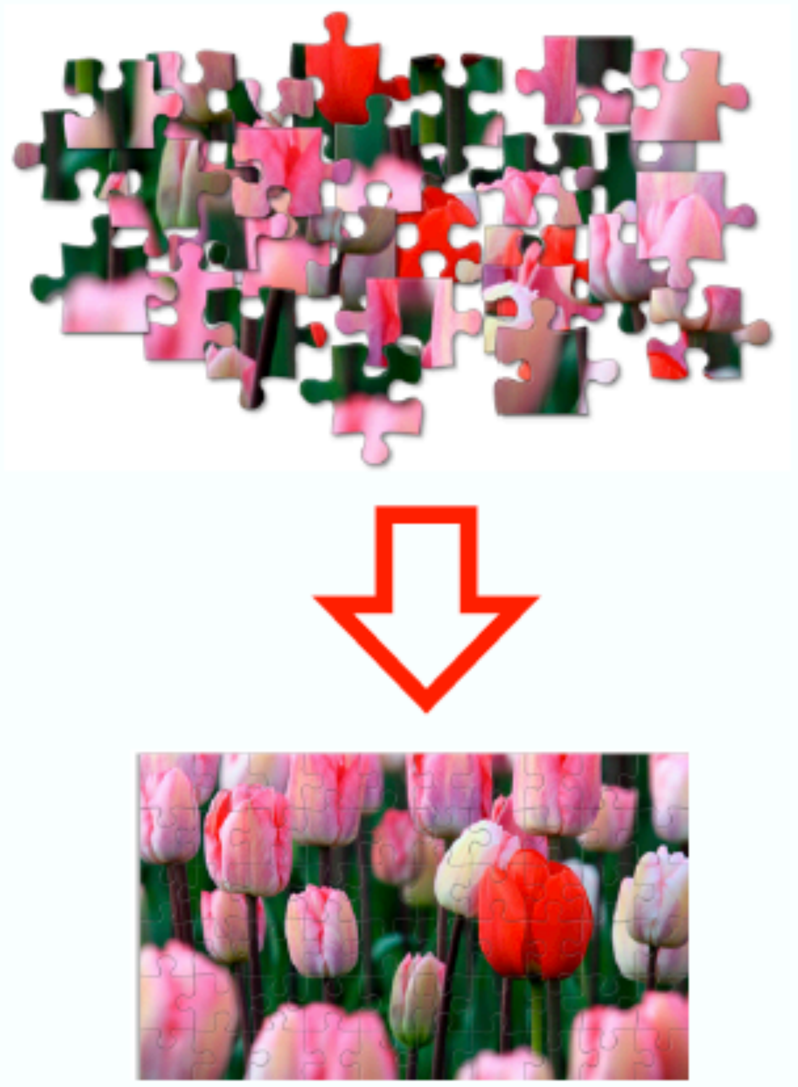


image extracted from [1]

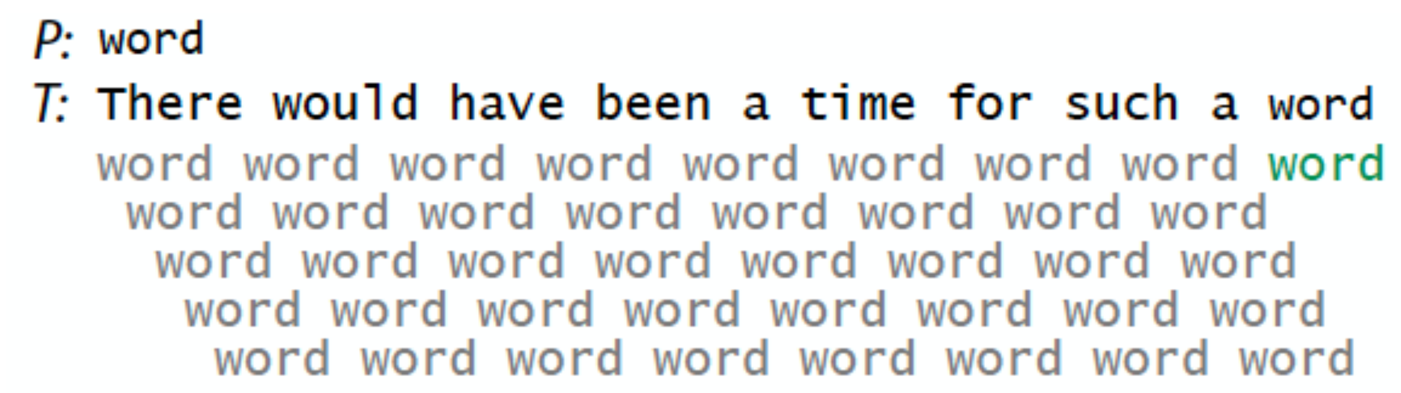
Read alignment problem

I. As an exact pattern matching problem

read R = pattern P

Genome G = text T

Where exactly does pattern P occur in the text T?



```
P: word
T: There would have been a time for such a word
   word word word word word word word word word
    word word word word word word word word
     word word word word word word word word
      word word word word word word word word
       word word word word word word word word
```

slide adapted from [1]

```
Naive matcher (T, P):

for i in range (0, |T| - |P| + 1):
    if T[i, i+|P|-1] == P:
        print(« occurrence at position i »)
```

Complexity issues w.r.t the naïve algorithm

given |P| and |T|, what are the:

- number of possible matches
- greatest number of character comparisons
- lowest number of character comparisons
- what happens when all characters in |P| are different ?

What about the previous example :

P = *word*

T = *There would have been a time for such a word*

Complexity issues w.r.t the naïve algorithm

given |P| and |T|, what are the:

- number of possible matches |T|-|P|+1
- greatest number of character comparisons |P|*(|T|-|P|+1)
- lowest number of character comparisons |T|-|P|+1
- what happens when all characters in |P| are different ?

What about the previous example :

P = *word*

T = *There would have been a time for such a word*

Conclusions :

- worst case O(|P|*(|T|-|P|+1))
- in special cases (all characters in the pattern are identical or 100% different): O(|T|)

- by gathering more information on the internal redundancy of P, we can learn how to make bigger shifts
  - > pattern pre-processing step

  - finite automata algorithm
  - Boyer Moore
  - Knuth Morris Pratt
  - shift-or algorithm (special class of algorithms)

Shift – OR algorithm

A new approach to text searching
Yates and Gonnet, 1998

- O(|T|) for searching small patterns
- O(|P|+|Σ|) for pre-processing, where Σ is the alphabet

- simple and quick, based on bit operations (thus suitable for hardware implementations)

- no buffering of the text

Shift – OR algorithm

- a table M with |P| masks corresponding to every character in the alphabet

```
P = ababc
M[a] = 11010 (a matches at positions 1 and 3)
M[b] = 10101 (b matches at positions 2 and 4)
M[c] = 01111 (c matches at position 5)
M[*] = 11111
```

- a state vector s, initialized with 1s
  11111 for this example (= no match)

Shift – OR algorithm

- state 10101 means that we are at position 4 in the pattern (and got 2 partial matches : one of size 2 and one of size 4)
- a match of P in T will generate a state starting with 0 (ex. 01111) meaning that a prefix of P of size 5 is matched (= P)
- at each step we go to a new state

  s = (previous s << 1) | M[current_char in T]

Shift – OR algorithm

P ababc

T abdabababc

initial s = 11111

M[T[1]] = 11010

s = (initial s << 1) | 11010 = 11110 | 11010 = 11110

Shift – OR algorithm

```
P   ababc
T   abdababab c
previous s = 11110

M[T[2]] = 10101
s = (prev s << 1) | 10101 = 11100 | 10101 = 11101
```

Shift – OR algorithm

P   ababc

T   abdababab c

previous s = 11101

M[T[3]] = 11111

s = (prev s << 1) | 11111= 11010 | 11111 = 11111

no match, re-start from the previous match !!!

here from the beginning of the pattern as there is no partial match (no prefix of P is a suffix of T)

Shift – OR algorithm

```
P   ababc
T   abdababab c
previous s = 11111

M[T[4]] = 11010
s = (prev s << 1) | 11010 = 11110 | 11010 = 11110
```

Shift – OR algorithm

```
P   ababc
T   abdababab c
previous s = 11110

M[T[5]] = 10101
s = (prev s << 1) | 10101 = 11100 | 10101 = 11101
```

Shift – OR algorithm

P   ababc

T   abdabababc

previous s = 11101

M[T[6]] = 11010

s = (prev s << 1) | 11010 = 11010 | 11010 = 11010

Shift – OR algorithm

```
P   ababc
T   abdababababc
previous s = 11010

M[T[7]] = 10101
s = (prev s << 1) | 10101 = 10100 | 10101 = 10101
```

Shift – OR algorithm

P ababc

T abdabAbabc

previous s = 10101

M[T[8]] = 11010

s = (prev s << 1) | 11010 = 01010 | 11010 = 11010

no match, re-start from the previous partial match !!!

begins at position 3 in the current part of the text

Shift – OR algorithm

P   ababc

T   abdabababc

previous s = 11010

M[T[9]] = 10101

s = (prev s << 1) | 10101 = 10100 | 10101 = 10101

Shift – OR algorithm

P   ababc

T   abdabababc

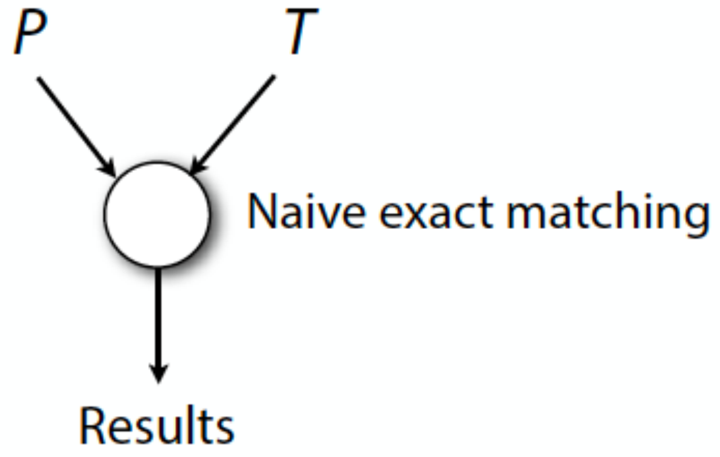
previous s = 10101

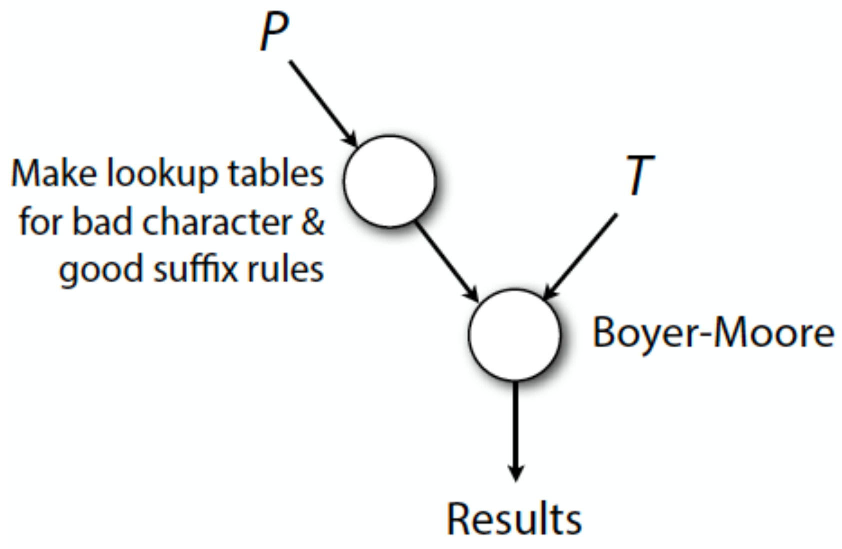
M[T[10]] = 01111

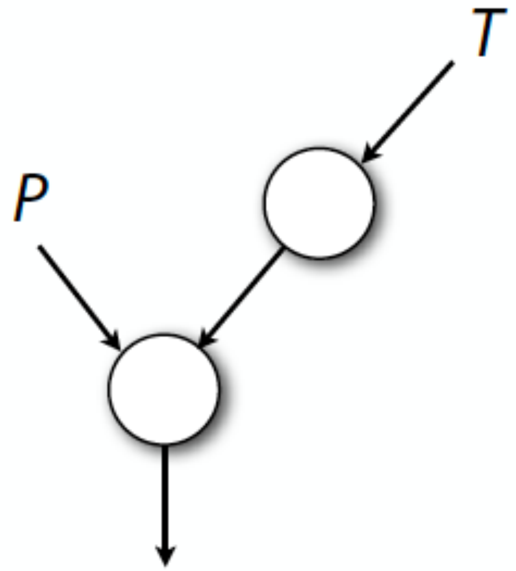
s = (prev s << 1) | 01111 = 01010 | 01111 = 01111

final state = match

Pre-processing





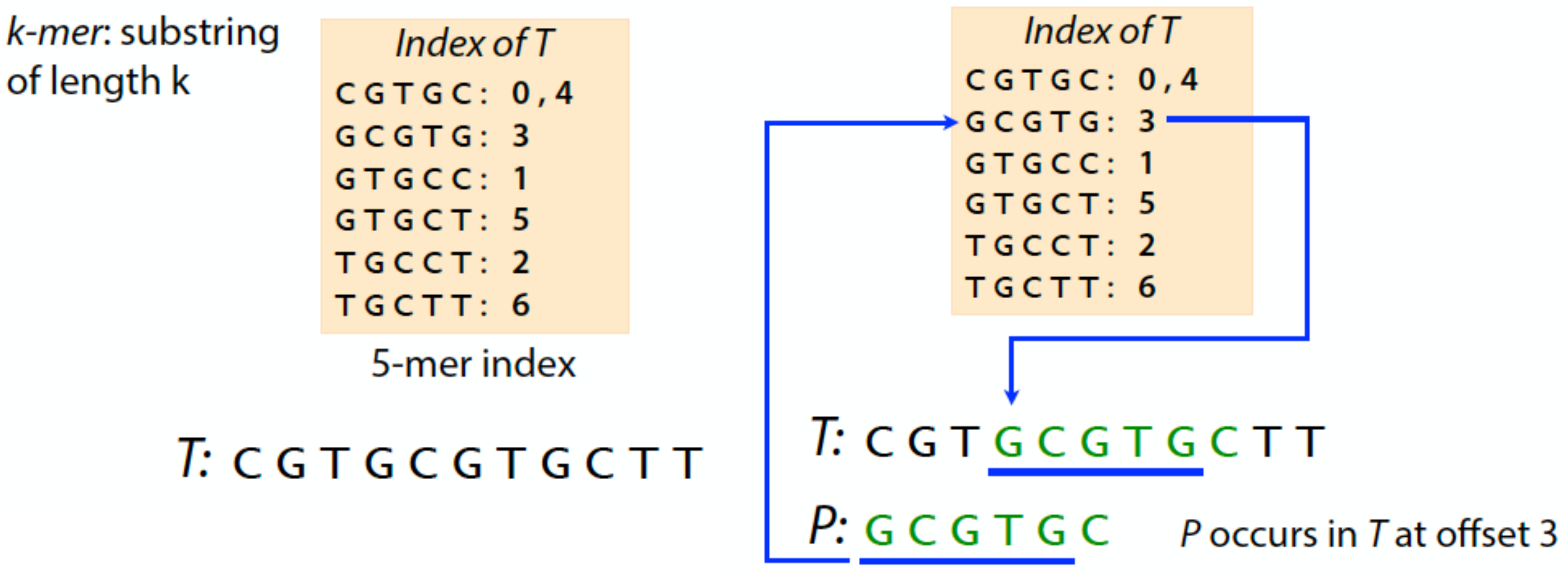


Algorithms that preprocess T are offline.

Otherwise, they are called online.

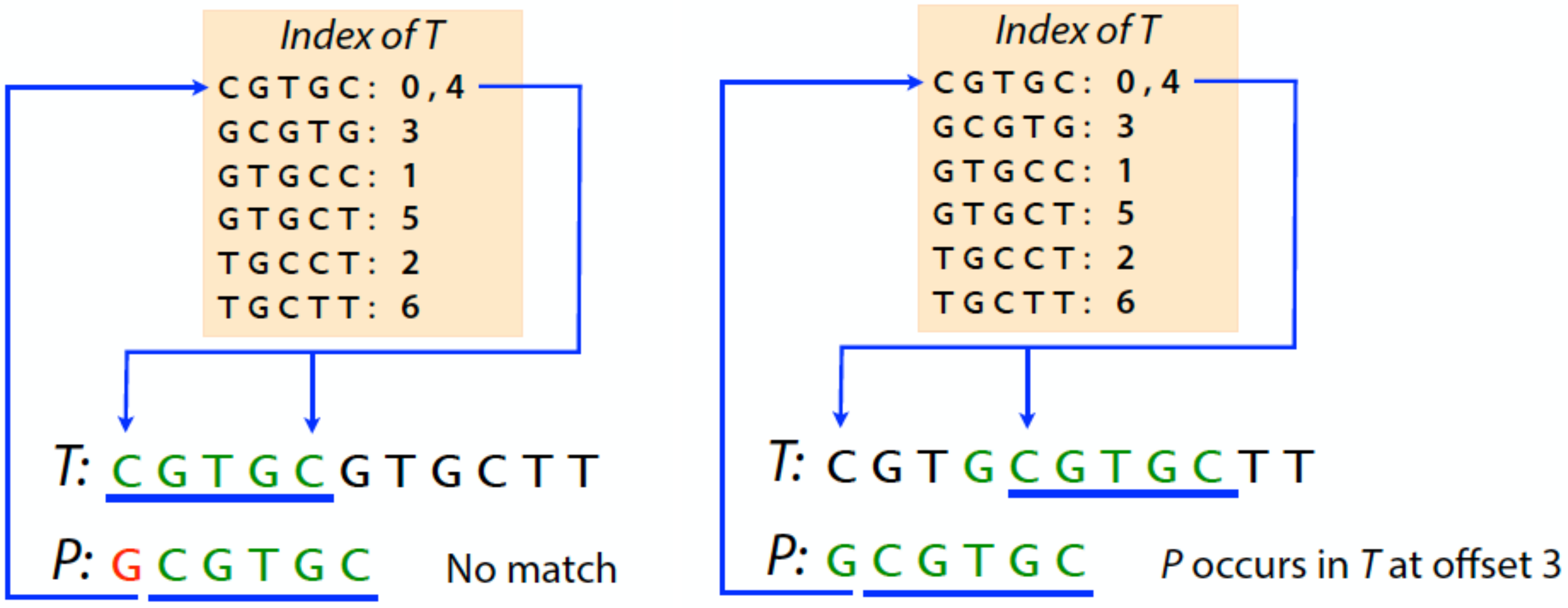
slide adapted from [1]

Offline exact matching: build a k-mer index of T



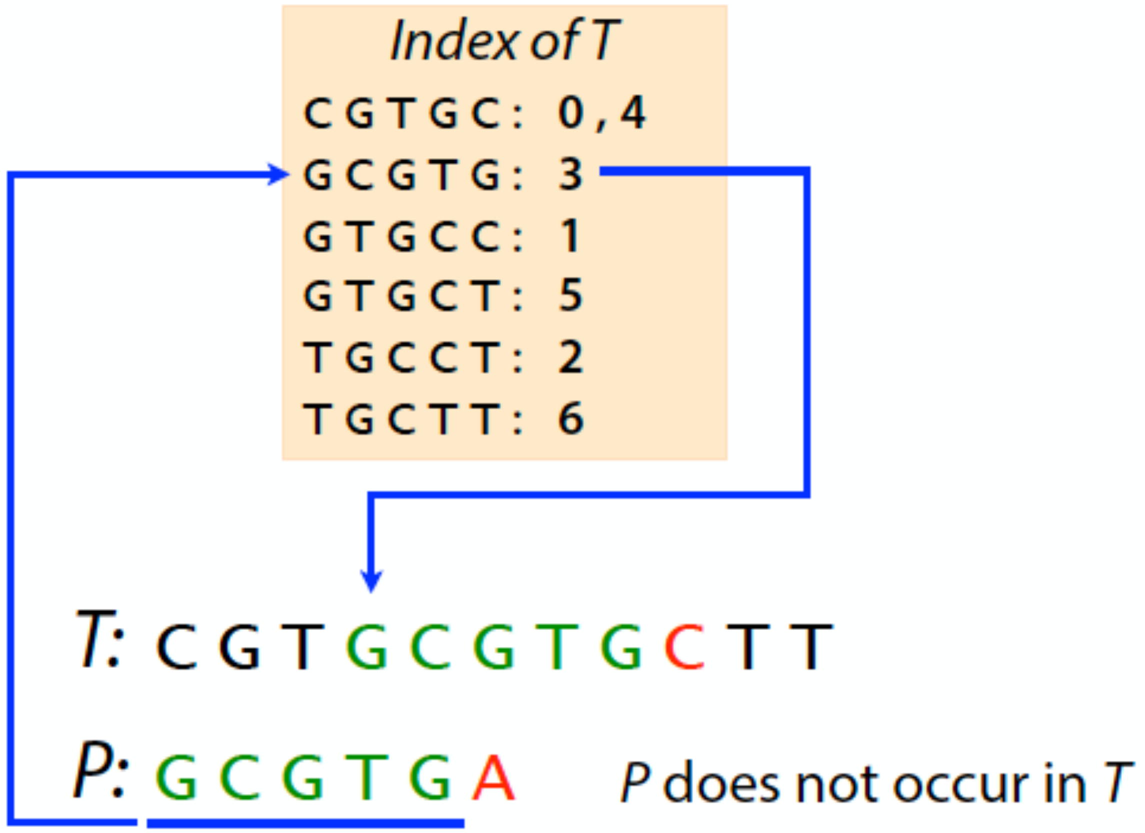
slide adapted from [1]

Still querying the index – several hits



slide adapted from [1]

Still querying the index – no hit



slide adapted from [1]

Multimaps to implement the index.

1. Ordered list

   – binary search O(log(|T|)

   – in python we can use *bisect* package with a sorted list
     bisect_left(a, x) : leftmost offset where x can be inserted
     into a while maintaining order

   – how ? show how this works on our example

   – what about filling the data structure ?

2. Hash table

   in python dictionaries are based on hash tables