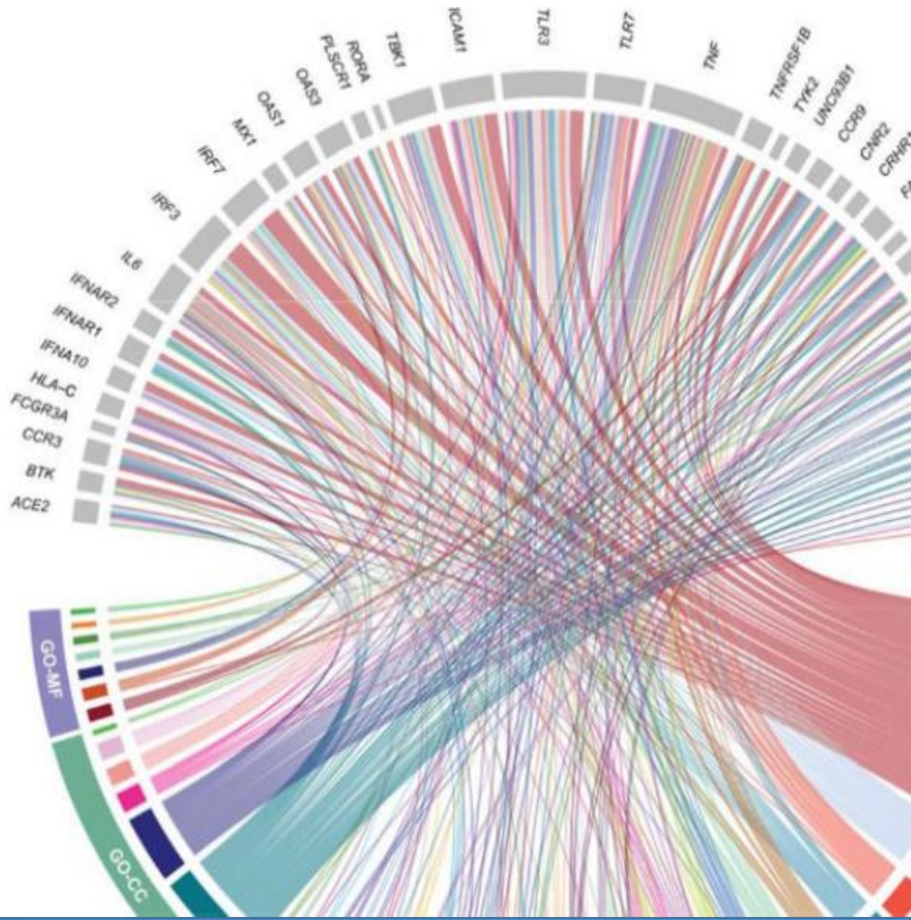
Tècniques i Eines Bioinformàtiques

Exact Matching Problem

Santiago Marco-Sola (santiago.marco@upc.edu)

Màster en Enginyeria Informàtica, UPC
Departament of Computer Science
Facultat d'Informàtica de Barcelona (FIB), UPC



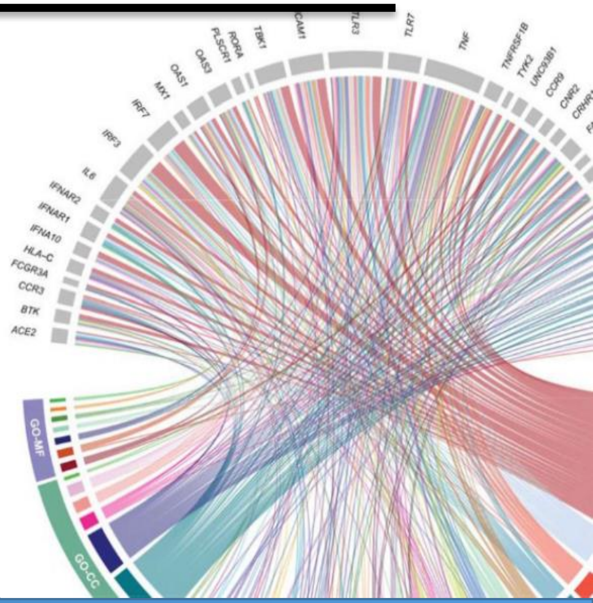
Acknowledgements

Many pictures and materials are taken from **Ben Langmead's course**.

Course heavily inspired in:

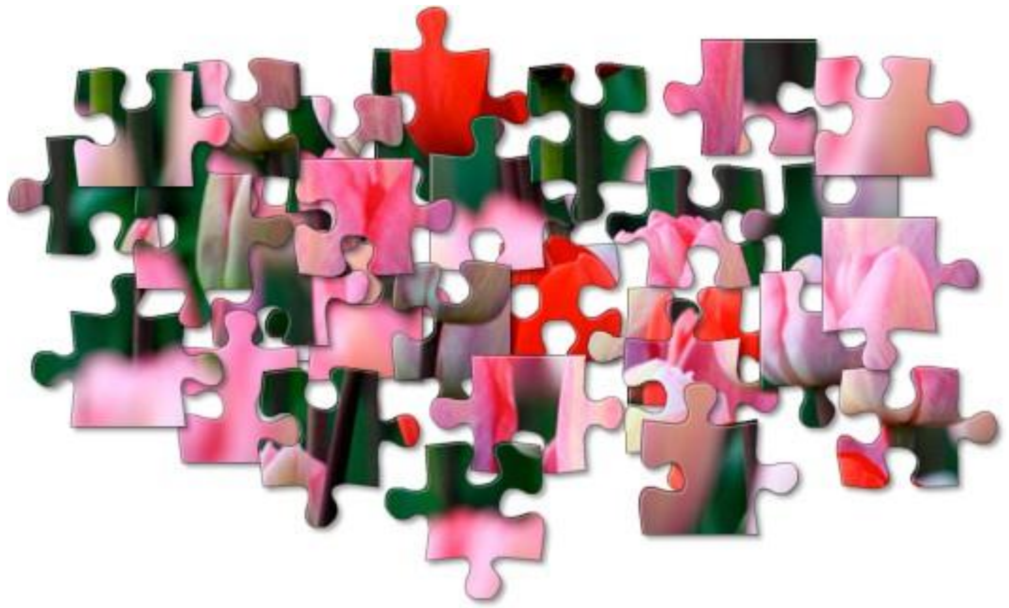
- **Genome-Scale Algorithm Design**. Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, Alexandru I. Tomescu. Cambridge University Press.
- **Algorithms on Strings, Trees, and Sequences**. Dan Gusfield. Cambridge University Press.
- **An Introduction to Bioinformatics Algorithms.** Neil C. Jones, Pavel A. Pevzner. MIT Press.

1 Strings


ACE2
BTK
CCR3
FCGR3A
HLA-C
IFNA10
IFNAR1
IFNAR2
IL6
IRF3
IRF7
MX1
OAS1
TLR7
TNF
GO-MF
GO-CC

Sequencing Reads are Strings

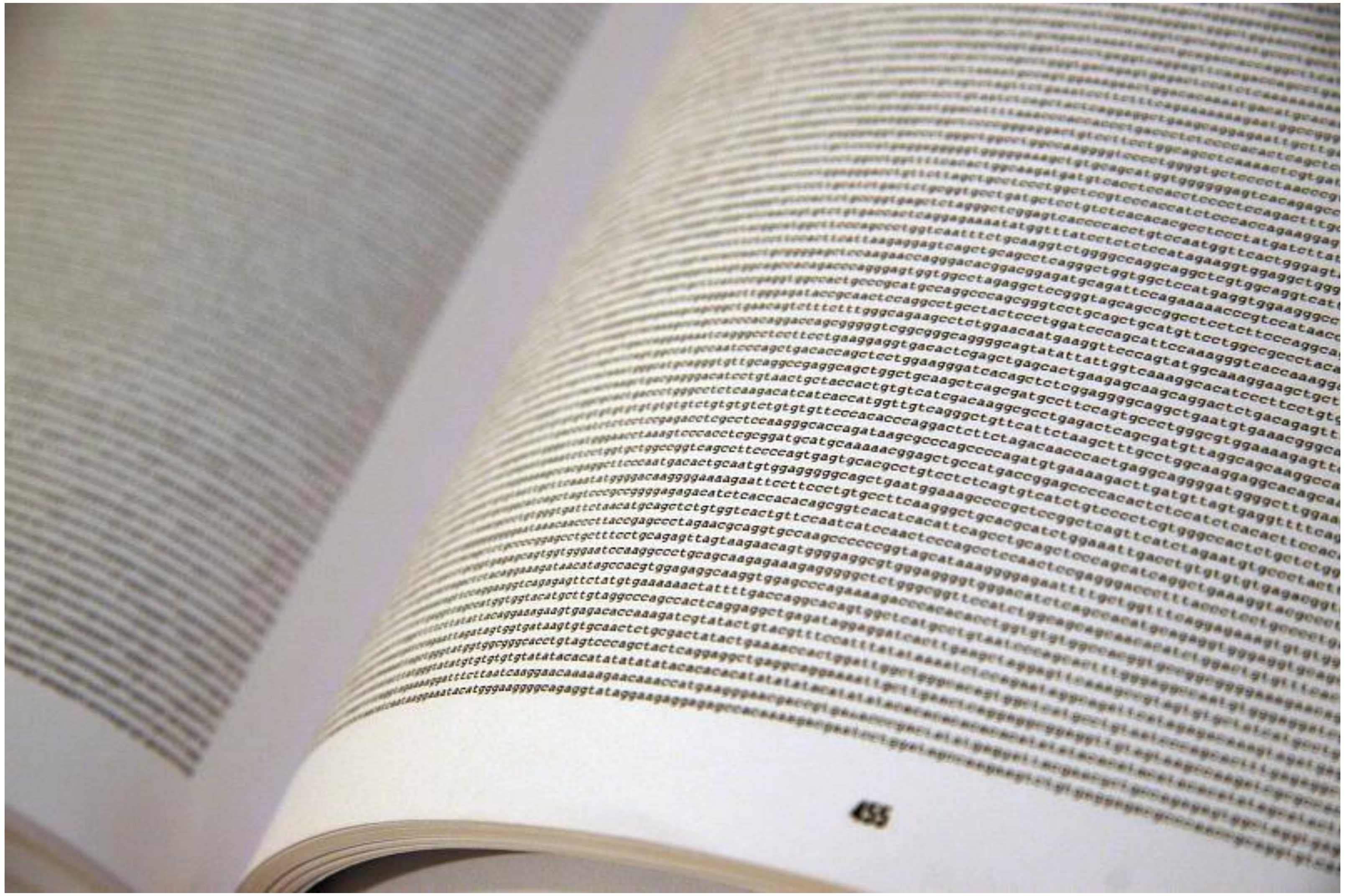
```
GTATGCACGCGATAG   TATGTCGCAGTATCT   CACCCTATGTCGCAG
TAGCATTGCGAGACG   GGTATGCACGCGATA   TGGAGCCGGAGCACC
TGTCTTTGATTCCTG   CGCGATAGCATTGCG   GCATTGCGAGACGCT
GACGCTGGAGCCGGA   GCACCCTATGTCGCA   GTATCTGTCTTTGAT
TATCGCACCTACGTT   CAATATTCGATCATG   GATCACAGGTCTATC
CACGGGAGCTCTCCA   TGCATTTGGTATTTT   CGTCTGGGGGGTATG
GTATGCACGCGATAG   ACCTACGTTCAATAT   TATTTATCGCACCTA
GCGAGACGCTGGAGC   CTATCACCCTATTAA   CTGTCTTTGATTCCT
CCTACGTTCAATATT   GCACCTACGTTCAAT   GTCTGGGGGGTATGC
GACGCTGGAGCCGGA   GCACCCTATGTCGCA   GTATCTGTCTTTGAT
TATCGCACCTACGTT   CAATATTCGATCATG   GATCACAGGTCTATC
CACGGGAGCTCTCCA   TGCATTTGGTATTTT   CGTCTGGGGGGTATG
```



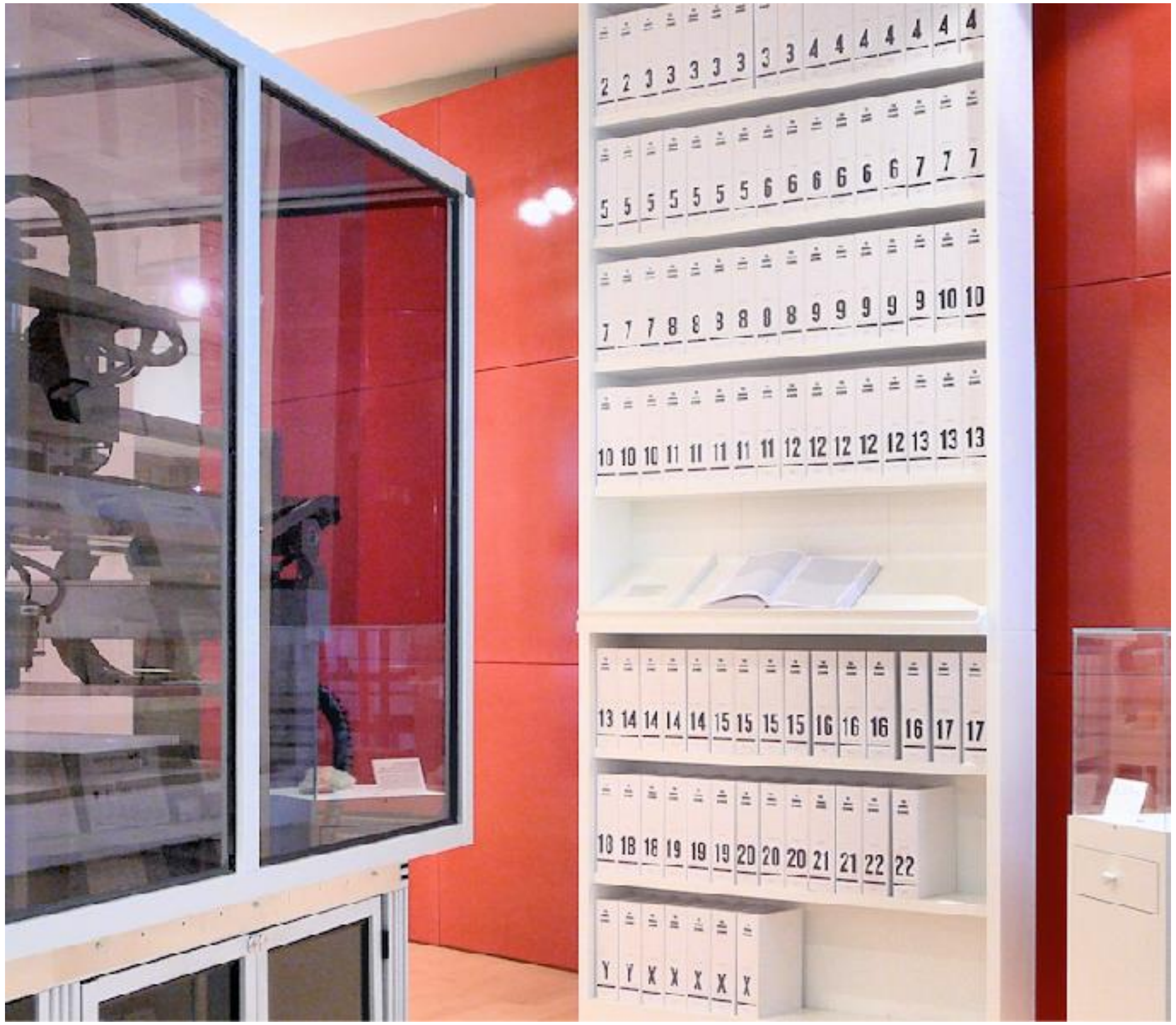
Sequencing reads are *strings;* sequences of characters

The strings are the only hints we get about *where the reads came from from* with respect to the longer DNA molecules...

... like pictures on puzzle pieces



What if I told you to find all the places where the string GATACCA occurs in here?



What if I told you to find all the places where the string GATACCA occurs in here?

Strings

Read

CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTAGGCCTTC

x billions

Reference

x million

We're going to *need* the right algorithms...

Strings Algorithms and Data Structures are Well Studied

Many kinds of data are string-like: books, web pages, files on your hard drive, medical records, chess games, ...

Algorithms for one kind of string are often applicable to others:

- Regular expression matching can find files on your filesystem (grep), or bad network packets (snort)
- Indexes for books and web pages (inverted indexing) can be used to index DNA sequences
- Methods for understanding speech (HMMs) can be used to understand handwriting or identify genes in genomes

Strings Come from Somewhere

Processes that give rise to real-world strings are complicated. It helps to understand them.

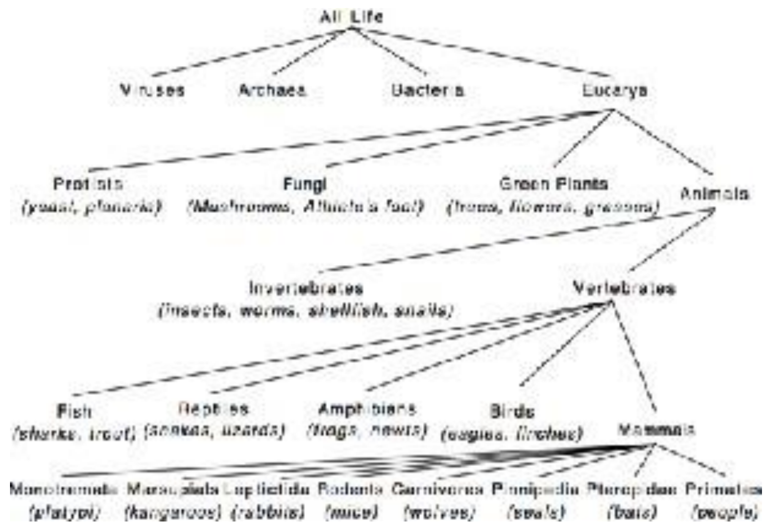
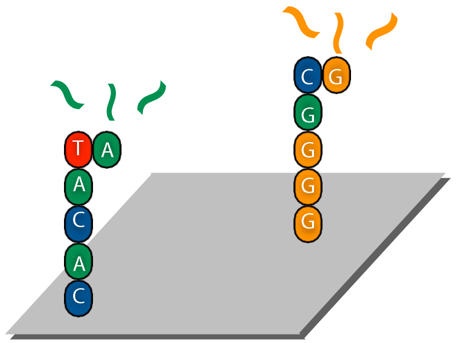


Figure from: Hunter, Lawrence. "Molecular biology for computer scientists." *Artificial intelligence and molecular biology* (1993): 1-46.

1. Evolution: Mutation
Recombination
(Retro)transposition

2. Lab procedures: PCR
Cell line passages



3. Sequencing: Fragmentation bias
Miscalled bases

Strings have Structure

One way to model a string-generating process is with coin flips:

{ = A, = C, = G, = T }

But such strings lack internal patterns (“structure”) exhibited by real strings

> 40% of the human genome is covered by *transposable elements*, which copy-and-paste themselves across the genome and mutate

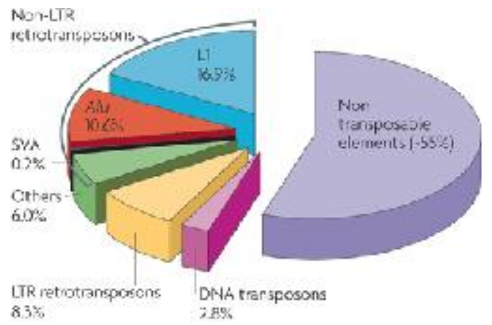


Image from: Cordaux R, Batzer MA. The impact of retrotransposons on human genome evolution. Nat Rev Genet. 2009 Oct;10(10):691-703

Slipped strand mis-pairing during DNA replication results in expansion or retraction of simple (*tandem*) repeats

••• ATATATATATATAT •••

↕

••• ATATATATATATATATAT •••

String Definitions

String S is a finite sequence of characters

Characters are drawn from alphabet Σ Usually, Σ = { A, C, G, T }

| S | = number of characters in *S*

```
>>> s = 'ACGT'
>>> len(s)
4
```

ε is "empty string" | ε | = 0

```
>>> len('')
0
```

Positions within a string *S* are referred to with *offsets*

```
>>> s = 'ACGT'
>>> s[0]
'A'
>>> s[2]
'G'
```

Usually assume alphabet Σ is finite, with O(1) elements

Nucleic acid alphabet: { A, C, G, T }

Amino acid alphabet: { A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V }

Occasionally we'll consider what happens as |Σ| grows

String Definitions

Concatenation of *S* and *T* , *ST* = characters of *S* followed by characters of *T*

```
>>> s = 'AACC'
>>> t = 'GGTT'
>>> s + t
'AACCGGTT'
```

Substring of *S* is a string occurring inside *S*

```
>>> s = 'AACCGGTT'
>>> s[2:6]
'CCGG' # substring of seq
```

S is a *substring* of *T* if there exist (possibly empty) strings *u* and *v* such that *T* = *uSv*

String Definitions

Prefix of *S* is a substring starting at the beginning of *S*

```
>>> s = 'AACCGGTT'
>>> s[0:6]
'AACCGG' # prefix
>>> s[:6] # same as above
'AACCGG'
```

S is a *prefix* of *T* if there exists a string *u* such that $T = Su$

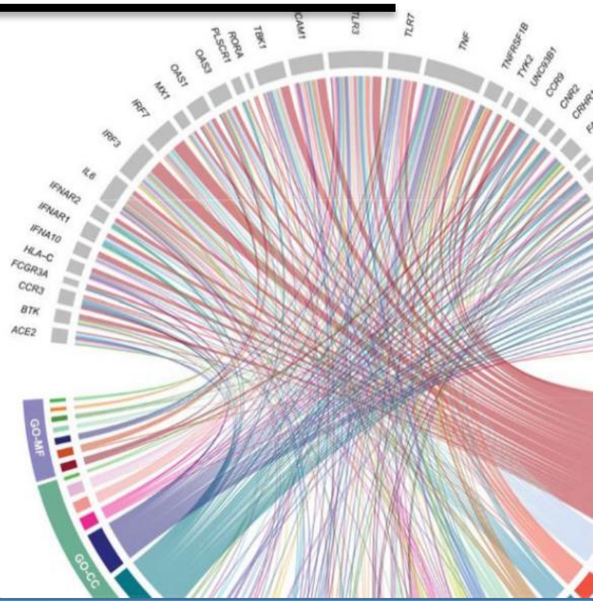
String Definitions

Suffix is substring ending at end of *S*

```
>>> s = 'AACCGGTT'
>>> s[4:8]
'GGTT' # suffix
>>> s[4:] # like s[4:len(s)]
'GGTT'
>>> s[-4:] # like s[len(s)-4:len(s)]
'GGTT'
```

S is a *suffix* of *T* if there exists a string *u* such that $T = uS$

2 Exact Matching



Exact Matching Problem

Find places where *pattern P* occurs as a substring of *text T*.
Each such place is an *occurrence* or *match*.

Let $n = |P|$, and let $m = |T|$ Assume $n \leq m$

Alignment: a way of putting *P's* characters opposite *T's*.
May or may not correspond to an match.

```
P:  word
T:  There would have been a time for such a word
            Alignment 1: word         Alignment 2: word
```

Exact Matching Problem

What's a simple algorithm for exact matching?

```
P: word
T: There would have been a time for such a word
   word word word word word word word word word   <- One occurrence
    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
```

Try all possible alignments. For each, check if it matches.
This is the *naïve algorithm*.

Naïve Exact Matching Algorithm

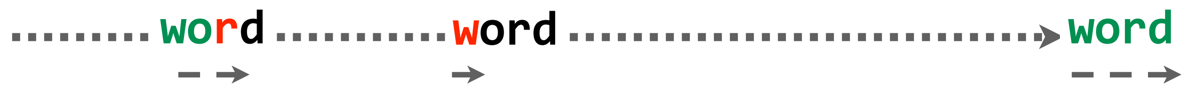
Naïve Exact Matching Algorithm

```
def naive(p,t):
  occurrences = []
  for i in range(len(t)-len(p)+1): # Loop over positions (of T)
    match = True
    for j in range(len(p)): # Loop over characters (of P)
      if t[i+j] != p[j]:
        match = False # Mismatch found! (reject alignment!)
        break
    if match:
      occurrences.append(i) # All chars matched (accept alignment!)
  return occurrences
```

Even more naïve: remove **break**

P: **word**

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

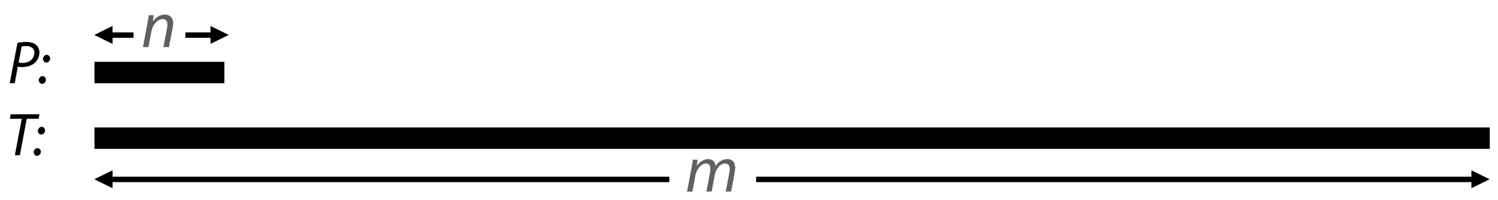


Naïve Exact Matching Algorithm

$n = |P| \qquad m = |T|$

How many alignments are possible?

$$m - n + 1$$



Naïve Exact Matching Algorithm

$n = |P| \qquad m = |T|$

Greatest # character comparisons possible?

$$n(m - n + 1)$$

```
P: aaaa
T: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
    aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
     aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
      aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
       aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
```

Naïve Exact Matching Algorithm

$n = |P| \qquad m = |T|$

Least # character comparisons possible?

m - n + 1

P: **abbb**

T: **bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb**

```
abbb  abbb  abbb  abbb  abbb  abbb  abbb  abbb  abbb
 abbb  abbb  abbb  abbb  abbb  abbb  abbb  abbb
  abbb  abbb  abbb  abbb  abbb  abbb  abbb  abbb
   abbb  abbb  abbb  abbb  abbb  abbb  abbb  abbb
    abbb  abbb  abbb  abbb  abbb  abbb  abbb  abbb
```

Naïve Exact Matching Algorithm

How many character comparisons in this example?

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

Hint: there are 41 possible alignments

Naïve Exact Matching Algorithm

How many character comparisons in this example?

P: `word`

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

```
 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
```

40 mismatches + 6 matches = 46 character comparisons

Closer to the minimum (41) than the maximum (164)

Exact Matching. Are there Better Algorithms?

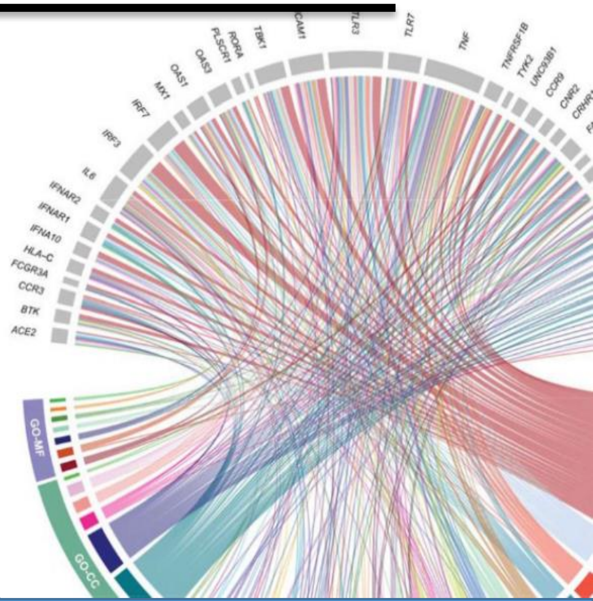
```
P: word
T: There would have been a time for such a word
           word
```

u doesn't occur in *P*, so skip next two alignments

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

<u>We'll take such ideas further when we discuss Boyer-Moore</u>

3 Exercises (Hands-on)



Exercises

1. GC Content Calculation.

- Given a DNA sequence, compute the GC content of a given DNA sequence. Write a function to calculate the percentage of 'G' and 'C' bases in a DNA string. Test it with sample sequences.

2. Reverse Complement of a DNA Sequence

- Generate the reverse complement of a given DNA sequence. Implement a function that replaces each base with its complement ('A' <-> 'T', 'C' <-> 'G') and reverses the string. Test with different sequences.

3. Naïve Exact String Matching

- Find all occurrences of a short pattern (read) within a longer reference genome sequence. Implement a simple exact matching algorithm. Test it with a small reference genome.

4. k-mer Counting

- Count the frequency of all k-mers (substrings of length k) in a given DNA sequence. Implement a function to extract all k-mers of a given length from a DNA sequence. Count the occurrences of each k-mer. Test it with different values of k.

Exercises

1. GC Content Calculation.

- Given a DNA sequence, compute the GC content of a given DNA sequence. Write a function to calculate the percentage of 'G' and 'C' bases in a DNA string. Test it with sample sequences.

GC Content Calculation

```python
def gc_content(seq):
    gc_count = sum(1 for base in seq if base in "GC")
    return (gc_count / len(seq)) * 100 if len(seq) > 0 else 0

# Example usage
sequence = "AGCTATAG"
print(f"GC content: {gc_content(sequence):.2f}%")
```

Exercises

2. Reverse Complement of a DNA Sequence

- Generate the reverse complement of a given DNA sequence. Implement a function that replaces each base with its complement ('A' <-> 'T', 'C' <-> 'G') and reverses the string. Test with different sequences.

Reverse Complement of a DNA Sequence

```python
def reverse_complement(seq):
    complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return "".join(complement[base] for base in reversed(seq))

# Example usage
sequence = "AGCTATAG"
print(f"Reverse complement: {reverse_complement(sequence)}")
```

Exercises

3. Naïve Exact String Matching

- Find all occurrences of a short pattern (read) within a longer reference genome sequence. Implement a simple exact matching algorithm. Test it with a small reference genome.

Naïve Exact String Matching

```
def naive_exact_match(pattern, text):
    matches = []
    for i in range(len(text) - len(pattern) + 1):
        if text[i:i+len(pattern)] == pattern:
            matches.append(i)
    return matches

# Example usage
genome = "AGCTTAGCTAAGCTAGCTAGCTAGCTA"
pattern = "AGCTA"
print(f"Pattern found at positions: {naive_exact_match(pattern, genome)}")
```

Exercises

4. k-mer Counting

- Count the frequency of all k-mers (substrings of length k) in a given DNA sequence. Implement a function to extract all k-mers of a given length from a DNA sequence. Count the occurrences of each k-mer. Test it with different values of k.

K-mer Counting

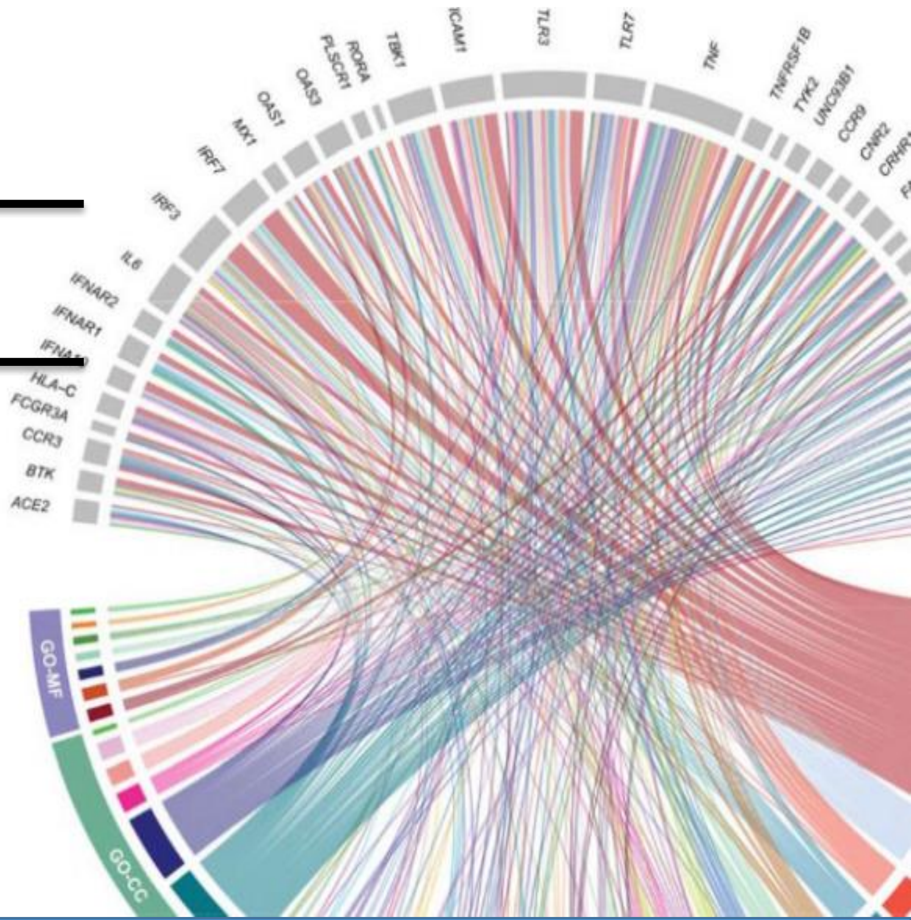
```python
from collections import defaultdict

def kmer_count(seq, k):
    counts = defaultdict(int)
    for i in range(len(seq) - k + 1):
        kmer = seq[i:i+k]
        counts[kmer] += 1
    return counts

# Example usage
sequence = "AGCTTAGCTAAGCTAGCTAGCTAGCTA"
k = 3
kmer_counts = kmer_count(sequence, k)

# Print k-mer frequencies
for kmer, count in kmer_counts.items():
    print(f"{kmer}: {count}")
```

Questions


ACE2
BTK
CCR3
FCGR3A
HLA-C
IFNAR1
IFNAR2
IL6
IRF3
IRF7
MX1
OAS1
OAS3
PLSCR1
RORA
TBK1
ICAM1
TLR3
TLR7
TNF
TNFRSF1B
TYK2
UNC93B1
CCR9
CNR2
GO-MF
GO-CC