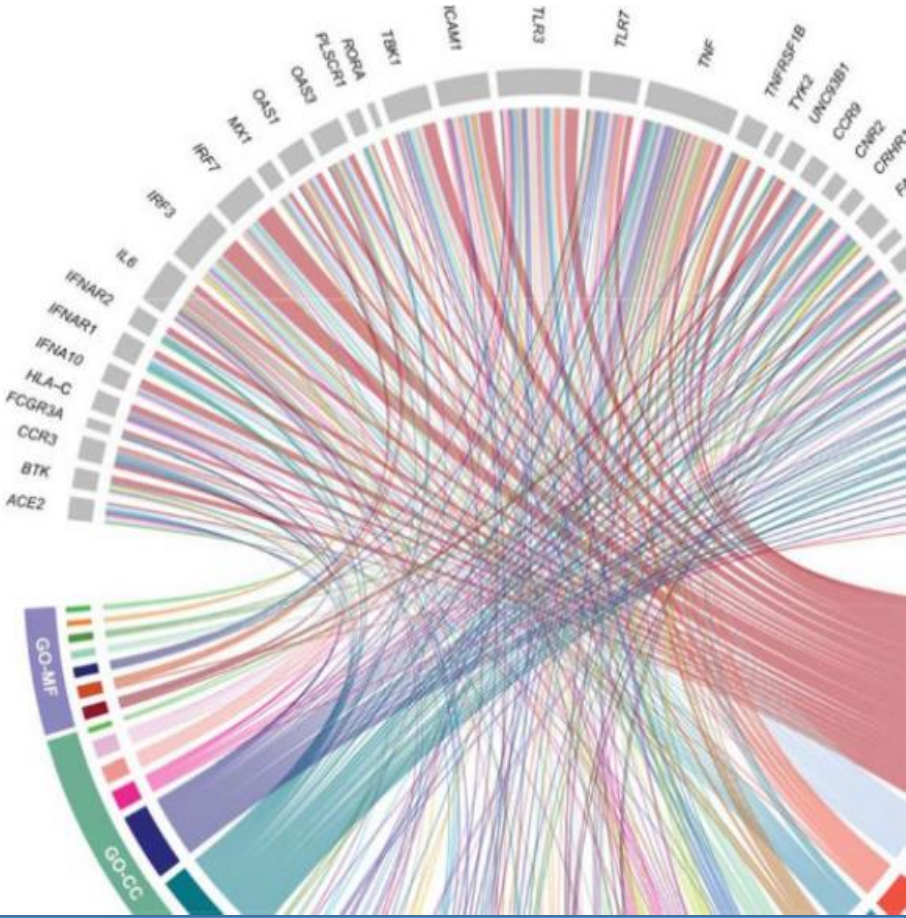
Tècniques i Eines Bioinformàtiques

Boyer-Moore Algorithm



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

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

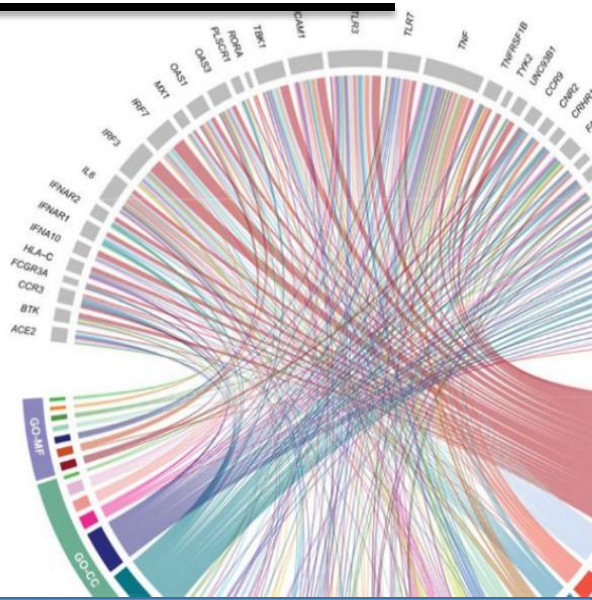
Acknowledgements

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 Boyer-Moore Algorithm



Can we improve on the naïve algorithm?

```
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
```

Boyer-Moore

Learn from character comparisons to skip pointless alignments

1. When we hit a mismatch, move *P* along until the mismatch becomes a match — "Bad character rule"
2. When we move *P* along, make sure characters that matched in the last alignment also match in the next alignment — "Good suffix rule"
3. Try alignments in one direction, but do character comparisons in *opposite* direction — "Longer skips"

P: `word`

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

`word`

Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

Boyer-Moore: Bad character rule

- Upon mismatch, skip alignments until:
  - (a) mismatch becomes a match
  - (b) P moves past mismatched character.
  - (c) If there was no mismatch, don't skip

Step 1:
T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C
Case (a)

Step 2:
T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C
Case (b)

Step 3:
T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C
Case (c)

Step 4:
T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C

Boyer-Moore: Bad character rule

Step 1:
T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C

Step 2:
T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C

Step 3:
T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C

Up to step 3, we skipped 8 alignments

5 characters in *T* were never looked at

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t* or (b) *P* moves past *t*

```
                       t
Step 1:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P: C T T A C T T A C

                       t
Step 2:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P:         C T T A C T T A C

Step 3:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P:                 C T T A C T T A C
```

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t* or (b) *P* moves past *t*

```
                        t
Step 1:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P: C T T A C T T A C
```

t occurs *in its entirety* to the left within *P*

```
                                t
Step 2:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P:         C T T A C T T A C
```

prefix of *P* matches a *suffix* of *t*

```
Step 3:  T: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
         P:                 C T T A C T T A C
```

Case (a) has two subcases according to whether *t* occurs *in its entirety* to the left within *P* (as in step 1), or a *prefix* of *P* matches a *suffix* of *t* (as in step 2)

Boyer-Moore: Putting it together

How to *combine* bad character and good suffix rules?

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

bad char says skip 2, good suffix says skip 7

Take the maximum! (7)

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*

Step 1:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G — bc: **6**, gs: 0 *bad character*

Step 2:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G — bc: 0, gs: **2** *good suffix*

Step 3:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G — bc: 2, gs: **7** *good suffix*

Step 4:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Comparisons Skipped (characters ignored)

11 characters of *T* we ignored

Step 1:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Step 2:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Step 3:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Step 4:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Skipped 15 alignments

Boyer-Moore: Good suffix rule

We learned the *weak* good suffix rule; there is also a *strong* good suffix rule

t

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

P: C T T A C T T A C

Weak: C T T A C T T A C

guaranteed mismatch!

Strong: C T T A C T T A C

Strong good suffix rule skips more than weak, at no additional penalty

Strong rule is needed for proof of Boyer-Moore's O(n + m) worst-case time. Gusfield discusses proof(s) in first several sections of ch. 3

Boyer-Moore implementation

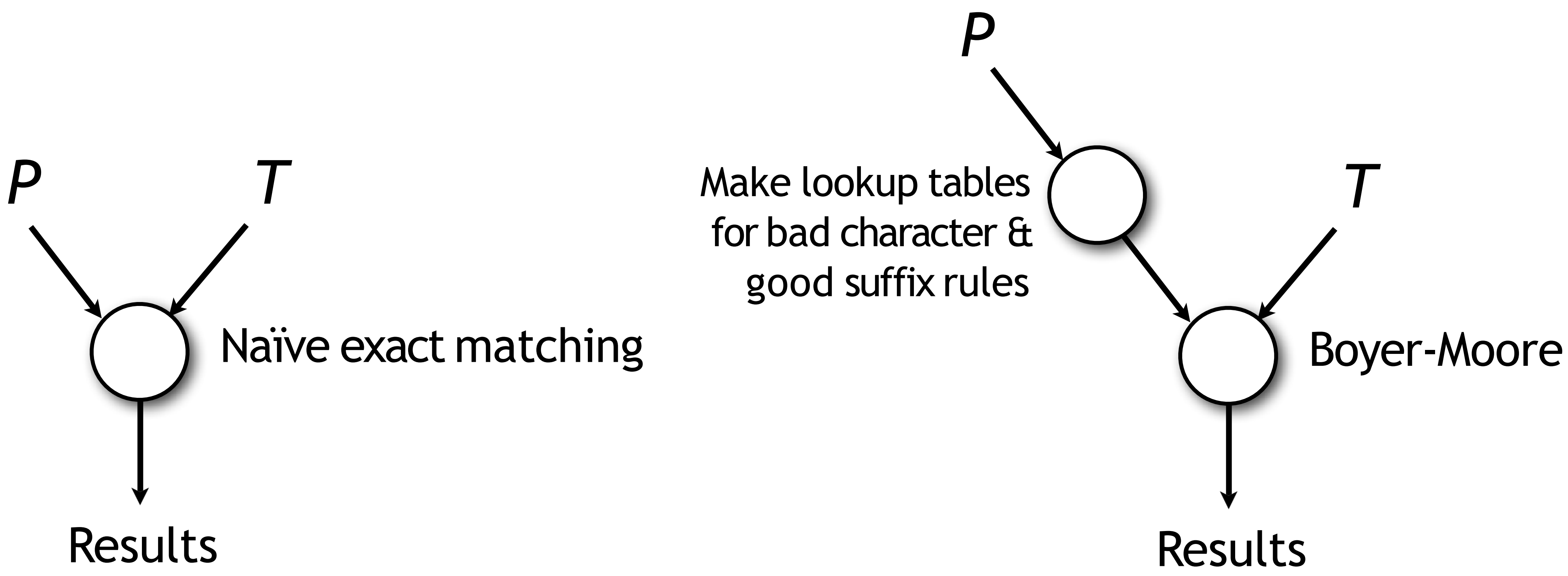
Boyer-Moore Algorithm

```
def boyer_moore(p,p_bm,t):
  # Do Boyer-Moore matching
  i = 0
  occurrences = []
  while i < len(t) - len(p) + 1: # Left to right
    shift = 1
    mismatched = False
    for j in range(len(p)-1,-1,-1):  #  Right  to  left
      if p[j] != t[i+j]:
        skip_bc = p_bm.bad_character_rule(j,t[i+j])
        skip_gs = p_bm.good_suffix_rule(j)
        shift = max(shift,skip_bc,skip_gs)
        mismatched = True
        break
    if not mismatched:
      occurrences.append(i)
      skip_gs = p_bm.match_skip()
      shift = max(shift,skip_gs)
    i += shift
    return  occurrences
```

Full Code:

http://j.mp/CG_BoyerMoore

Preprocessing. Naïve algorithm vs. Boyer-Moore



Boyer-Moore: Preprocessing

- Pre-calculate skips for all possible mismatch scenarios!
- This can be constructed efficiently. See Gusfield 2.2.2.
- As with bad character rule, good suffix rule skips can be precalculated efficiently. See Gusfield 2.2.4 and 2.2.5.
- For both tables, the calculations only consider P. No knowledge of T is required.

T: A A T C A A T A G C
P: T C G C

P

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

Boyer-Moore: Best case

What's the best case?

P: **bbbb**
T: **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**

bbbbbbbb bbbb bbbb bbbbbbbb
bbbbbbbb bbbb bbbb bbbb

Every alignment yields immediate mismatch and bad character rule skips *n* alignments

How many character comparisons? *floor(m / n)*

Boyer-Moore: Worst case

Boyer-Moore, with refinements in Gusfield, is $O(n + m)$ time

Given $n < m$, can simplify to $O(m)$

Is this better than naïve?

For naïve, worst-case # char comparisons is $n(m - n + 1)$

Boyer-Moore: $O(m)$, naïve: $O(nm)$

Reminder: $|P| = n$ $|T| = m$

Naïve vs Boyer-Moore

- As m and n grow, the number characters comparisons grows with...

  $|P| = n$

  $|T| = m$

	Naïve matching	Boyer-Moore
Worst case	$m \cdot n$	m
Best case	m	m / n

Performance comparison

Simple Python implementations of naïve and Boyer-Moore:

	Naïve matching		Boyer-Moore		
	# character comparisons	wall clock time	# character comparisons	wall clock time	
P: "tomorrow" **T:** Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s	17 matches $\lvert T \rvert$ = 5.59 M
P: 50 nt string from Alu repeat* **T:** Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s	336 matches $\lvert T \rvert$ = 249 M

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Preprocessing: Boyer-Moore

Preprocessing: trade one-time cost for reduced work overall via *reuse*

Boyer-Moore preprocesses P into lookup tables that are *reused*

reused for each alignment of P to T_1

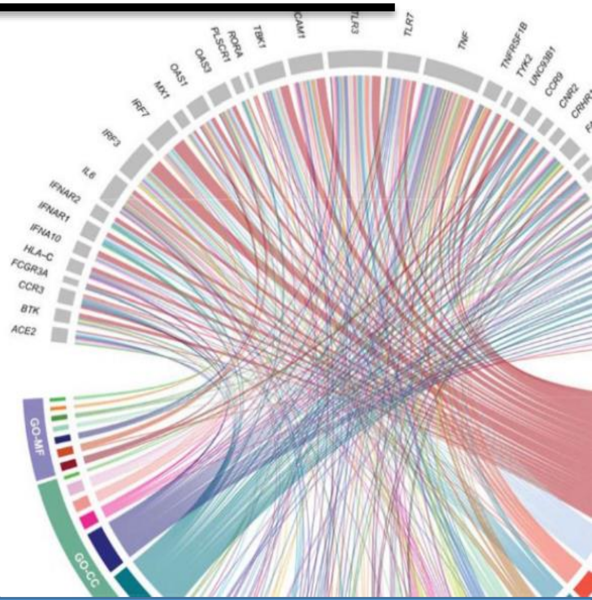
If you later give me T_2, I *reuse* the tables to match P to T_2

If you later give me T_3, I *reuse* the tables to match P to T_3

...

Cost of preprocessing is *amortized* over alignments & texts

3 Exercises (Hands-on)



Exercises

1. Implement the Bad Character Rule (Easy)

- Implement the bad character rule, one of the optimizations of Boyer-Moore. Create a function that precomputes the last occurrence of each character in P. Use this table to determine the shift when a mismatch occurs.

2. Combine Naïve Algorithm and Bad Character Rule (Intermediate)

- Find all occurrences of a short pattern (read) within a longer reference genome sequence. Modify the naïve string matching function to incorporate the bad character rule. Test it with a small reference genome.

3. Implement the Good Suffix Rule (Advanced)

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

4. Full Boyer-Moore Implementation (Expert)

- Implement the complete Boyer-Moore algorithm using both the bad character and good suffix rules. Combine the bad character and good suffix heuristics to achieve efficient string matching. Compare performance against the naïve algorithm.

Helpers

Download chr1.fa

```
> wget http://hgdownload.soe.ucsc.edu/goldenPath/hg38/chromosomes/chr1.fa.gz
```

Parsing and Reading Chr1.fa

```python
import argparse

def read_fasta(file_path):
  sequence = []
  with open(file_path, 'r') as file:
    for line in file:
      if not line.startswith('>'):
        sequence.append(line.strip())  # Remove newline and concatenate
  return ''.join(sequence)

if __name__ == "__main__":
  parser = argparse.ArgumentParser(
    description="Read a FASTA file and store the sequence in a string.")
  parser.add_argument("i", help="Path to the FASTA file")
  args = parser.parse_args()

  sequence = read_fasta(args.fasta_file)
```

Exercise 1: Implement the Bad Character Rule

1. Implement the Bad Character Rule (Easy)

- Implement the bad character rule, one of the optimizations of Boyer-Moore. Create a function that precomputes the last occurrence of each character in P. Use this table to determine the shift when a mismatch occurs.

Bad Character Rule

```python
import string

def bad_character_table(pattern):
    """
    Precompute the last occurrence table for each character in the pattern.
    """
    bad_char = {}
    for i in range(len(pattern)):
        bad_char[pattern[i]] = i  # Store the last occurrence of each character
    return bad_char

def bad_character_shift(pattern, mismatch_char, mismatch_index):
    """
    Determine the shift when a mismatch occurs.
    """
    bad_char = bad_character_table(pattern)
    last_occurrence = bad_char.get(mismatch_char, -1)  # Default to -1 if character not in pattern
    shift = max(1, mismatch_index - last_occurrence)  # Shift to align the next best possible match
    return shift
```

Exercise 1: Implement the Bad Character Rule

1. Implement the Bad Character Rule (Easy)

- Implement the bad character rule, one of the optimizations of Boyer-Moore. Create a function that precomputes the last occurrence of each character in P. Use this table to determine the shift when a mismatch occurs.

Bad Character Rule

```
# Example usage
pattern = "GCTTAC"
text = "AAGCTTGCCTTACGCTTAC"

# Generate the bad character table
bad_char_table = bad_character_table(pattern)
print("Bad Character Table:", bad_char_table)

# Simulate a mismatch at index 4 with character 'G'
mismatch_char = "G"
mismatch_index = 4
shift = bad_character_shift(pattern, mismatch_char, mismatch_index)

print(f"Mismatch at index {mismatch_index} with char '{mismatch_char}', shift by {shift} positions.")
```

Exercise 2: Combine Naïve Algorithm and Bad Character Rule

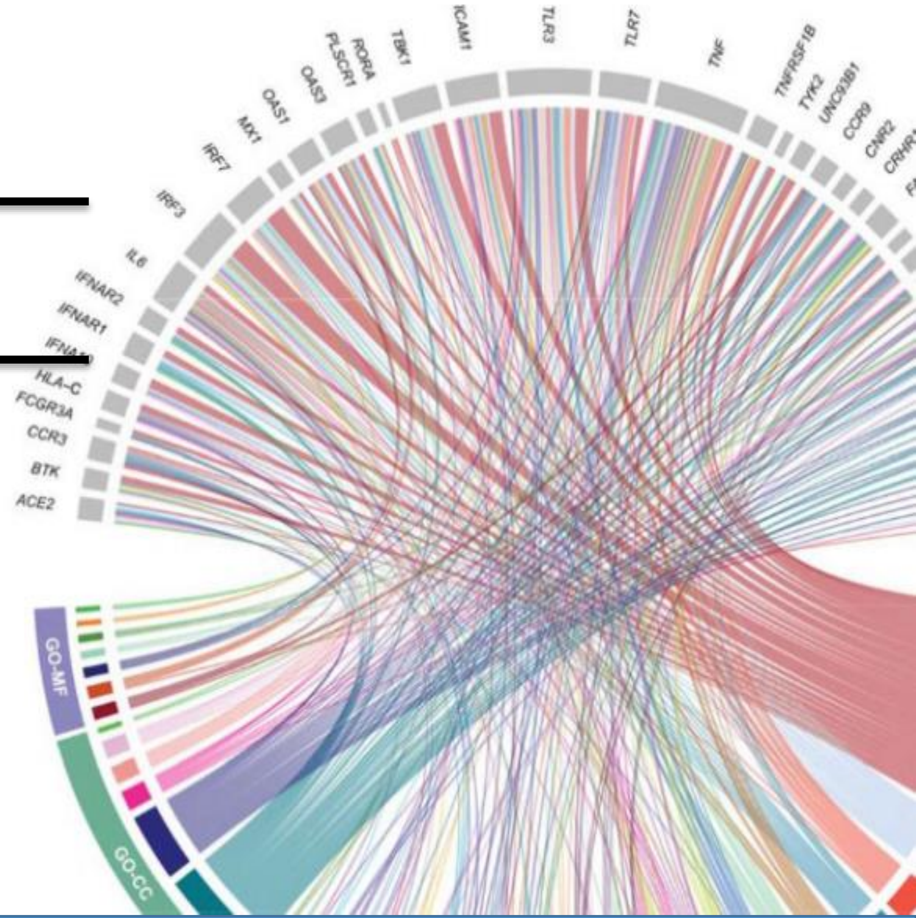
Naïve Algorithm and Bad Character Rule

```python
def boyer_moore_bad_character(text, pattern):
  """
  Boyer-Moore exact string matching using the Bad Character
  Rule.Finds all occurrences of `pattern` in `text`.
  """
  m, n = len(pattern), len(text)
  if m > n:
    return []  # Pattern is longer than text, no matches possible
  bad_char = bad_character_table(pattern) # Precompute bad character table
  occurrences = []
  i = 0  # Start index in text
  while i <= n - m:
    j = m - 1  # Start checking from the end of the pattern
    # Compare pattern with text from right to left
    while j >= 0 and pattern[j] == text[i + j]:
      j -= 1
    if j < 0:
      # Pattern found at index i
      occurrences.append(i)
      # Shift pattern
      i += (m - bad_char.get(text[i + m], -1)) if i + m < n else 1
    else:
      # Mismatch occurred, apply the Bad Character Rule
      shift = max(1, j - bad_char.get(text[i + j], -1))
      i += shift  # Move pattern to the right by the shift
  return occurrences
```

Testing

```python
# Example usage
text = "AGCTTAGCTAAGCTAGCTAGCTAGCTA"
pattern = "AGCTA"
matches = boyer_moore_bad_character(text, pattern)
print(f"Pattern found at positions: {matches}")
```

Questions



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