# BIF CM1
# Exact pattern matching
# without index

Téo lemane, Claire Lemaitre, Catherine Belleannée

10 september 2020

# Structure

# Pattern Matching

## PM - What

- Are you here?
- Where are you?

## PM - Why

- CTRL-F
- Search engines (50 billion web pages)
- DB requests
- Music (youtube, shazam)
- Biology (Zettabytes (ZB, $10^{21}$) from 2020)

# Sub problems

## Distinct PM

- **Exact** : Search *ACTG* in *ACGCTAACGGACGCA*
- **Approximate**: Search *ACTG* in *ACGCTAACGGACGCA* with at most $x$ substitutions, insertions and deletions

## PM - two approaches

- **On the fly**: reference non indexed
- **Indexed**: reference pre-treated

# Definitions (I)

- An alphabet $\Sigma$ is a finite non-empty set of symbols. We denote $|\Sigma|$ the alphabet cardinality.
- Examples:
  - Unary alphabet: contains only one symbol
  - Binary alaphabet: two symbols
  - In our case $\Sigma = \{A, C, G, T\}$, $|\Sigma| = 4$
- A string over an alphabet $\Sigma$ is a finite ordered sequence of symbols from $\Sigma$.
- A set of all strings over a given alphabet $\Sigma$ is denoted $\Sigma^*$. For example $AACCC \in \{A, C\}^*$ or $\{A, C, G, T\}^*$.
- A set of all strings of size $n$ over a given alphabet $\Sigma$ is denoted $\Sigma^n$.
- A length of a string $\mathbf{S}$ is denoted $|\mathbf{S}|$.
- The empty string is denoted $\epsilon$ and can be over any alphabet. $|\epsilon| = 0$

# Definitions (II)

- For any string $S$:
  - $S[i]$ denotes the $i^{th}$ symbol.
  - $S[i,j]$, with $i, j \in \mathbb{N} \cup \{0\}$, is the contiguous substring of $S$ that starts at position $i$ and ends at position $j$.
  - $S[i,j]$ is the empty string if $i > j$.
  - $S[0,i]$ and $S[i,|S|-1]$ are respectively a **prefix** and a **suffix** of string $S$.

# Definitions (III)

- Let $S = aagcgccgaa$ and $a, c, g \in \Sigma$.
- Let **p** and **s** a prefix and a suffix of $S$.
- If $|\mathbf{p}| < |S|$, then **p** is a **proper prefix** of $S$.
- If $|\mathbf{s}| < |S|$, then **s** is a **proper suffix** of $S$.
- *aagc* is a proper prefix of $S$. *aagcgccgaa* is a prefix of $S$ but not a proper prefix.

- A **border** of $S$ is a substring which is both a proper prefix and a proper suffix of $S$.
- **The** border of $S$ is the longest border of $S$. It's denoted $Border(S)$.
- *a* is a border of $S$, *aa* is **the** border of $S$.

# Exact pattern matching problem

## Problem

Let $T$ a string. $T \in \Sigma^*$.
Let $P$ a searched pattern. $P \in \Sigma^*$.
Let $|P| < |T|$.
Let $O$ the set of the starting positions of $P$ in $T$.
Find $O = \{i_1, ..., i_n\}$ such as $\forall i \in O, T[i, (i + |P| - 1)] = P$.

## Solutions

- Naive algorithm.
- Rabin-Karp algorithm.
- (Knuth-Morris-Pratt algorithm).
- (Boyer-Moore algorithm).

# Naive algorithm

```
Data: P ∈ Σ*, T ∈ Σ*
m = length(P);
n = length(T);
list match = emptyList;
for i = 0; i < n − m + 1; i = i + 1 do
    j = 0;
    while j < m do
        if T[i + j] ≠ P[j] then
            break
        end
        j = j + 1;
    end
    if j == m then
        print i;
        match.insert(i);
    end
end
return match
```

# Naive algorithm complexity

```
Data: P ∈ Σ*, T ∈ Σ*
m = length(P);
n = length(T);
list match = emptyList;
for i = 0; i < n − m + 1; i = i + 1 do
    j = 0;
    while j < m do
        if T[i + j] ≠ P[j] then
            break
        end
        j = j + 1;
    end
    if j == m then
        print i;
        match.insert(i);
    end
end
return match
```

- To perform all the necessary comparisons we need to visit $n - m$ positions of $T$ (the **for** loop).
- For each position, we compare $T[i + j]$ and $P[j]$ (the **while** loop). This corresponds to $m$ operations ($|P|$).
- Time complexity : $O(m(n - m))$ that can be simplified such as $O(mn)$.
- Space complexity : $O(1)$.

# Rabin-Karp algorithm

- Introduced by Richard M. Karp and Michael O. Rabin :
  R. M. Karp and M. O. Rabin, ”Efficient randomized pattern-matching algorithms,” in IBM Journal of Research and Development, vol. 31, no. 2, pp. 249-260, March 1987, doi: 10.1147/rd.312.0249.

- We no longer compare each symbol of a sub-string to pattern symbols but sub-string hashes to a pattern hash.
- Avoids a large number of symbol comparisons.

# Rabin-Karp algorithm

```
Data: P ∈ Σ*, T ∈ Σ*
m = length(P);
n = length(T);
list match = emptyList;
hash_pattern = hash(P);
for i = 0; i < n − m + 1; i = i + 1 do
    hash_text = hash(T[i, i + m − 1]);
    if hash_pattern == hash_text then
        j = 0;
        while j < m do
            if T[i + j] ≠ P[j] then
                break;
            end
            j = j + 1;
        end
        if j == m then
            print i;
            match.insert(i);
        end
    end
end
return match
```

# Rabin-Karp complexity

```
Data: P ∈ Σ*, T ∈ Σ*
1  m = length(P);
2  n = length(T);
3  list match = emptyList;
4  hash_pattern = hash(P);
5  for i = 0; i < n − m + 1; i = i + 1 do
6      hash_text = hash(T[i, i + m − 1]);
7      if hash_pattern == hash_text then
8          j = 0;
9          while j < m do
10             if T[i + j] ≠ P[j] then
11                 break;
12             end
13             j = j + 1;
14         end
15         if j == m then
16             print i;
17             match.insert(i);
18         end
19     end
20 end
21 return match
```

- The hashes comparison is $O(1)$ but is executed for each position of $T$ so its impact is $O(n)$ (line 7).
- Lines 4, 6, 10 are $O(m)$. But line 4 is executed once and 10 is executed if the hashes match: rarely with a good hash function.

# Rolling hash

A rolling hash function allows to compute, for a text $T$, hash($T[i, i+m-1]$) from hash($T[i-1, i+m-2]$).

Example:
Let $\Sigma = \{A, C, G, T\}$
Let $h(A) = 0$, $h(C) = 1$, $h(G) = 2$, $h(T) = 3$
Let $T = ACGTAT$
Let $hash(T[i, i+m-1]) = \sum_{n=i}^{n \leq i+m-1} h(T[n])$

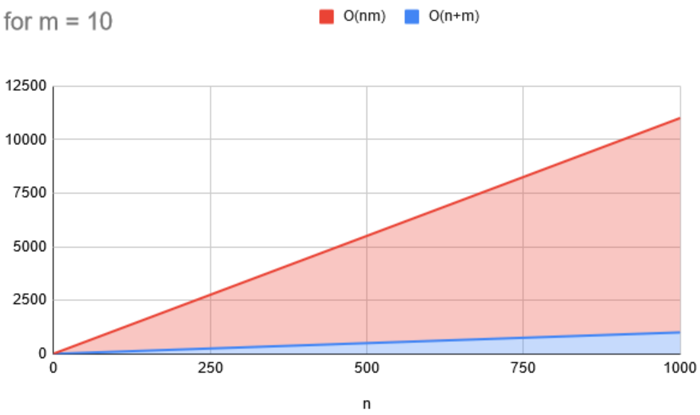
For $|P| = m = 3$:

- $hash(T[0,2]) = h(A) + h(C) + h(G)$
- $hash(T[1,3]) = h(C) + h(G) + h(A) \Leftrightarrow hash(T[1,3]) = hash(T[0,2]) - h(T[0]) + h(T[3])$

$hash(T[i+1, i+m-1]) = hash(T[i, i+m-2]) - h(T[i]) + h(T[i+m-1])$

# Rabin-Karp and Rolling hash

With a rolling hash function, Rabin-Karp algorithm is average $O(n+m)$, worst $O(nm)$.



# Rabin fingerprint

## Compute fingerprint

For a string $S$ and a prime $p$:

$$hash(S) = (S[0] \times p^0) + (S[1] \times p^1) + ... + (S[|S|-1] \times p^{|S|-1})$$
$$hash(S[i, i+m-1]) = (S[i] \times p^0) + (S[i+1] \times p^1) + ... + (S[i+m-1] \times p^{m-1})$$

## Example using ASCII values as hash values

- Let $S = ACGCCGCGG$, $m = 3$, $p = 11$.
- $h(x) = ascii(x)$
- $H0 = hash(S[0, 0+3-1]) = (h(A) \times 11^0) + (h(C) \times 11^1) + (h(G) \times 11^2)$
- $H0 = hash(S[0, 0+3-1]) = (65 \times 11^0) + (67 \times 11^1) + (71 \times 11^2)$
- $H1 = hash(S[1, 1+3-1]) = (67 \times 11^0) + (71 \times 11^1) + (67 \times 11^2)$

  $\Leftrightarrow H1 = (65 \times 11^0) + \frac{(67 \times 11^1)}{11} + \frac{(71 \times 11^2)}{11} - 65 + (67 \times 11^2)$

# Rabin fingerprint

## Example using ASCII values as hash values

- Let $S = ACGCCGCGG$, $m = 3$, $p = 11$.
- $h(x) = ascii(x)$
- $H0 = hash(S[0, 0+3-1]) = (h(A) \times 11^0) + (h(C) \times 11^1) + (h(G) \times 11^2)$
- $H0 = hash(S[0, 0+3-1]) = (65 \times 11^0) + (67 \times 11^1) + (71 \times 11^2)$
- $H1 = hash(S[1, 1+3-1]) = (67 \times 11^0) + (71 \times 11^1) + (67 \times 11^2)$

  $\Leftrightarrow H1 = (65 \times 11^0) + \frac{(67 \times 11^1)}{11} + \frac{(71 \times 11^2)}{11} - 65 + (67 \times 11^2)$

$$hash(S[(i+1), (i+m)]) = hash(S[i, i+m-1]) - h(S[i]) + h(S[i+m]) \times p^{m-1}$$

# Rabin-Karp and Rabin fingerprint

```
Data: P ∈ Σ*, T ∈ Σ*, p ∈ 𝒫
m = length(P);
n = length(T);
list match = emptyList;
hash_pattern = hash(P, p);
hash_text = hash(T[0, 0 + m − 1], p);
for i = 0; i < n − m + 1; i = i + 1 do
    if hash_pattern == hash_text then
        j = 0;
        while j < m do
            if T[i + j] ≠ P[j] then
                break;
            end
            j = j + 1;
        end
        if j == m then
            print i;
            match.insert(i);
        end
    end
    hash_text = next_hash(hash_text, i, T[i, i + m − 1], p);
end
return match
```

# Rabin fingerprint functions

**Function** `hash`(*string, prime*) **:**
    $h = 0$
    **for** $m = 0;\ m < length(string);\ m = m + 1$ **do**
        $h = h + (value(string[m]) \times prime^{m})$
    **return** $h$

**Function** `next_hash`(*previous, offset, string, prime*) **:**
    $h = previous - string[offset]$
    $h = h / prime$
    $m = length(string)$
    $h = h + (string[offset + m] \times prime^{m-1})$
    **return** *hash*

# Rabin-Karp for multiple pattern

For a single exact pattern matching, other algorithms which pre-process the pattern are in practice more efficient than Rabin-Karp, such as Boyer-Moore or Knuth-Morris-Pratt. But for multiple exact pattern matching, Rabin-Karp can be a very good choice.

# Rabin-Karp for multiple pattern

**Data:** $P = \{P_0, P_1, ..., P_n\}, T \in \Sigma^*, p \in \mathcal{P}$

```
m = length(P);
n = length(T);
list match = emptyList;
set hash_patterns = emptySet;
foreach pattern ∈ P do
    hash_patterns.insert(hash(pattern));
end
hash_text = hash(T[0, 0 + m − 1], p);
for i = 0; i < n − m + 1; i = i + 1 do
    if hash_text ∈ hash_patterns and T[i, i + m − 1] ∈ P then
        print i;
        match.insert(i);
    end
    hash_text = next_hash(hash_text, i, T[i, i + m − 1], p)
end
return match
```