String Matching...

String Matching Problems and Bioinformatics

Exact String Matching. Given a string $S = s_1s_2 \ldots s_n$ of charachters in some alphabet $\Sigma = \{a_1 \ldots a_K\}$ and a *pattern* or *query* string $P = p_1 \ldots p_m$, $m < n$ in the same alphabet, find all occurrences of P in S.

"find all occurrences" = report all positions $i_1, \ldots, i_k$ such that $s_{i_j} \ldots s_{i_j+m} = P$.

Approximate String Matching.[4] Given a string $S = s_1s_2 \ldots s_n$ of charachters in some alphabet $\Sigma = \{a_1 \ldots a_K\}$, *pattern* or *query* string $P = p_1 \ldots p_m$, $m < n$ in the same alphabet, ***a maximum error allowed*** $k \in \mathbb{R}$, and a *distance function* $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$, find all such positions i in S, that for some j,

$$d(s_i \ldots s_j, P) \leq k.$$

String matching in bioinformatics. Certain known nucleotide and/or amino acid sequences have properties known to biologists. E.g.,`ATG` is a string which must be present at the beginning of every protein (gene) a DNA sequence.

A *primer* is a *conserved DNA sequence* used in the Polymerase Chain Reaction (PCR) to identify the location of the DNA sequence that will be amplified (amplification starts at the location immediately following the 3' end primer, known as the forward primer). Finding if a DNA sequence contains a specific (candidate) primer is therefore paramount to the ability to run correct PCR.

A *conserved DNA sequence* is a sequence of nucleotides in DNA, which is found in the DNA of multiple species and/or multiple strains (for bacteria/prokaryotes), or, in general, in all/almost all sequences in a specific collection. Some sequences are conserved precisely. However, a lot of sequences are conserved with some modifications. Finding such modified strings is an important process for mapping DNA of a new organism, based on the known DNA of a related organism.

For example, we know that genetic code is redundant. Consider the following sequence of nucleotides:

GCTACTATTTTTCAT

When read in forward frame 0, this sequence encodes the following sequence of amino acids:

ATIFH

Because of redundancy of the genetic code, the following sequences of nucleotides will produce the same sequence of amino acids (lower case letters show differences with the original string):

```
GCT ACT ATT TTT CAT
GCc ACc ATT TTa CAc
GCc ACc ATc TTg CAT
GCT ACT ATc TTa CAc
GCg ACc ATa TTc CAc

 A   T   I   F   H
```

Therefore, it is often useful to search for approximate string matches in DNA sequences.

Exact String Matching.

There is a wide range of exact string matching algorithms. We briefly outline the following:

1. Naïve string matching.
2. Rabin-Karp algorithm.
3. Finite-automata-based string matching.
4. Knuth-Morris-Pratt (KMP) algorithm.
5. Boyer-Moore algorithm.
6. Gusfield's Z algorithm.

Algorithm Evaluation. We are interested in the following properties of each algorithm:

- Worst-case algorithmic complexity. The traditional measure of algorithm efficiency.
- Expected algorithmic complexity. Behavior of some algorithms is *much better* than the worst case on *most* inputs. (E.g., Rabin-Karp algorithm has the same worst-case complexity as the naïve method), but, in practice, runs much faster on most inputs of interest).
- Space complexity. We prefer our algorithms, even the fast ones to occupy space that is linear in the problem size.

Problem size. The "official" input to all string matching algorithms is a pair of strings S and P. Their lenghts, n and m, respecitvely, supply the problem size for us.

We note that $m < n$. In some cases, especially in bioinformatics applications $m << n$: our DNA sequences may be millions of nucleotides long, while the pattern P we are looking for (e.g., for primer identification) may be on the order of tens of nucleotides. In such cases, we may consider the size of input to be n, rather than $n + m$.

In our studies we fix the alphabet Σ. In bioinformatics, it makes sense, since the two known alphabets, nucleotide and amino acid ones, have 4 and 20 characters in them repsectively. Occasionally, $\Sigma = K$ may be considered part of the size of input. (in our studies, it won't).

Naïve Algorithm

The naïve algorithm takes as input strings S and P and checks if $s_i \dots s_{i+m} = P$ for each $i = 1, \dots n - m + 1$ directly. The pseudocode for this algorithm looks as follows. Here and in further algorithms we adopt notation $S[x, y]$ to represent the substring $s_x \dots s_y$ of S.

```
ALGORITHM NaiveStringMatch(S,P,n,m)
begin
  for k= 1 to n − m + 1   do
    if P = S[k, k + m − 1]   then
      output(k);
 end for
end
```

Analysis. The **if** statement in the algorithm takes (up to) m atomic comparisons of the form $P[i]? = S[k+i-1]$ to perform. The **for** loop repeats $n-m+1$ times. Therefore the running time of this algorithm is $O((n - m + 1) \cdot m) = O(nm - m^2 + m) = O(nm)$. Note, that is $m = \frac{n}{2}$, then $O(nm) = O(n^2)$, so the in the worst-case, the naïve algorithm is quadratic in the length of the input string S.

Rabin-Karp Algorithm

Idea. The key ideas behind this algorithm are:

- Comparison of two numbers is up to m times faster than comparison of two strings of length m.
- Represent the string P and substrings of S of length m as numbers in base K (size of the alphabet) numeric system. Compare the numbers to check if the strings match.
- If numeric representations of strings start being too large to allow for a single comparison, compare numeric representations of the two strings `mod` q for some relatively large number q.

  The latter will require a comparison of the actual strings in case their numeric values co-incide, but in practice there should be very few of such comparisons that would fail.

Numeric representation of strings. Let $\Sigma = \{a_1, \ldots, a_K\}$. We associate with each character a_i a digit $i-1$ in base K numeric system.

Example. The nucleotide alphabet is $\Sigma = \{A, C, G, T\}$. We can map individual characters in this alphabet to digits in base 4 system as follows:

Character	Digit
`A`	0
`C`	1
`G`	2
`T`	3

Consider a string $P = p_1 \ldots p_m$ in alphabet Σ. Without loss of generality we use p_i to refer both to the character in Σ and to the digit in the base K alphabet, associated with it. The numeric p value represented by P can be computed using the following expression:

$$p = p_m + K(p_{m-1} + K(p_{m-2} + \ldots + K(p_2 + Kp_1)) \ldots).$$

Given a string $S = s_1 \ldots s_n$, where $n > m$, the numeric value of $S[1, m]$, t_0 can be computed in the same way:

$$t_0 = s_m + K(s_{m-1} + K(s_{m-2} + \ldots + K(s_2 + Ks_1)) \ldots).$$

More importantly, given t_{i-1}, the numeric value t_i representing substring $S[i+1, i+m]$ can be computed incrementally as follows:

$$t_i = K(t_{i-1} - K^{m-1}s_i) + s_{i+m}.$$

We can precompute K^{m-1} as this number is a constant.

Example. Consider for following string $S=$ `"ATTCCGT"`. Suppose we are looking for four-letter substrings. There are $|S| - 4 + 1 = 7 - 4 + 1 = 4$ four-letter substrings in S: $S[1,4], S[2,5], S[3,6]$ and $S[4,7]$. We compute t_0 for $S[1,4]$ using the mapping of nucleotide symbols to digits in base 4 numeric system as follows:

$S[1,4] =$ `"ATTC"`; $t_0 = 1 + 4(3 + 4(3 + 4 \cdot 0)) = 1 + 4(3 + 4 \cdot 3) = 1 + 4 \cdot 15 = 61.$

Further, we compute t_1, t_2 and t_3 as follows:

$S[2,5] =$ `"TTCC"`; $t_1 = \; 4(t_0 - 4^3 \cdot 0) + 1 = 4(61 - 0) + 1 = 244 + 1 = 245.$
$S[3,6] =$ `"TCCG"`; $t_2 = \; 4(t_1 - 4^3 \cdot 3) + 2 = 4(245 - 64 \cdot 3) + 2 = 4(245 - 192) + 2 = 4 \cdot 53 + 2 = 214.$
$S[4,7] =$ `"CCGT"`; $t_3 = \; 4(t_2 - 4^3 \cdot 3) + 3 = 4(214 - 192) + 3 = 4 \cdot 22 + 3 = 91.$

Algorithm in a nutshell. Given P and S, we compute the numeric values p for P and t_0 for $S[1,m]$. We compare p to t_0, and then, using the incremental recompuation method, compute t_1, t_2, etc. and compare them to p. Any time $p = t_i$ for some i, we report $i+1$ as the position of a match.

Complication. This version of Rabin-Karp algorithm works as desired, when m is small enough, that p and t_i can fit the machine word. E.g.,

$$4^{50} = 1,267,650,600,228,229,401,496,703,205,376 \sim 10^{31}.$$

This means that strings P up to length 51 can be found in larger strings on processors with 32-bit architectures. For longer strings, the algorithm will have to be modified.

Modification. Pick a number q, and perform computations of p and t_i values $modq$. If $p \neq t_i(modq)$, then $S[i+1, i+m] \neq P$. However, $p = t_i$ **does not immediately guarantee** $S[i+1, i+m] = P$. We run the full comparison of P and $S[i+1, i+m]$. The pseudocode for the modified algorithm is as follows. The algorithm takes as input, in addition to strings P and S and their respective lenghts m and n, two more numbers: K, the size of the alphabet and q.

```
ALGORITHM RabinKarp(S, P, n, m, K, q)
begin
  h ← K^(m−1) mod q;
  p ← 0;
  t_0 ← 0;
  for i= 1 to m   do     //compute initial numeric values
    p ← (K · p + P[i]) mod q;
    t_0 ← (K · t_0 + S[i]) mod q;
 end for
  for j= 0 to n − m   do     //search for substring P in S
    if  p = t_j  then      //if numeric values match,
      if  P = S[j + 1, j + m]   then      need to compare substrings
        output  j + 1;
      end if
    end if
    if  j < n − m  then      //move onto the next substring in S
      t_(j+1) = ((K(t_j − S[j + 1] · h) + S[j + m + 1])  mod q;
 end for
end
```

Analysis. The **if** $P = S[j+1, j+m]$ comparison costs $O(m)$ atomic comparisons. In the worst case (e.g., when $S = A^n$ and $P = A^m$), this line will be executed on each iteration of the **for** loop. There are $n - m + 1$ iterations, and therefore, the *worst-case complexity* of this algorthm is $O((n - m + 1) \cdot m) = O(nm)$. (We also note that the first loop has running time $O(m)$, and therefore does not contribute significantly to the overall complexity of the algorithm).

However, unlike the naïve algorithm, Rabin-Karp algorithm improves its running time significantly when there are relatively few, comparing to n matches between p and t_0. E.g., in many cases when P is NOT a substring of S, the running time of the algorithm will be $O(m + n)$ (the first loop takes $O(m)$, the second – $O(n)$).

If S has v valid shifts (i.e., v locations in which P occurs), then we can estimate the running time of the Rabin-Karp algorithm as $O(m)+O(n(v+n/q))$, where n/q is our estimate of the probability of a "misfire" (i.e., $p = t_i$, when $P \neq S[i+i, i+m]$).

Choosing q. The best value of q is a prime number that is just a bit smaller than $2^(B-1)$ where B is the number of bits in the machine word on a given processor architecture. For 16-bit words (16-bit INTs), take $q = 32,749$. For 32-bit words, you can use $q = 2,147,483,629$ or $q = 2,147,483,647 = 2^{31} - 1$.

Finite-automata-based String Matching

Idea. A string $P = p_1 \dots p_m$ is a finite string, and therefore, is a regular expression in Σ^*. A regular expression matching all strings in $\Sigma^* = \{a_1, \dots, a_K\}$ that have P as a substring is

$$(a_1|a_2|\dots|a_K)^* p_1 p_2 \dots p_m (a_1|a_2|\dots|a_K)^*.$$

Therefore, we can use finite automata to check if an input string S has P as a substring.

Algorithm structure. The finite-automata-based string matching consists of two parts:

1. **Preprocessing.** Given a string P, a finite state machine (finite automaton) M_P for checking for the occurrence of P is constructed.
2. **Matching.** Given a string S and M_P, M_P is run on S, and it outputs the index $i - |P|$ for each index i of S, at which M_P reaches a final state.

We notice that these two stages are completely decoupled. M_P can be constructed, given P ahead of time. Also, if multiple strings need to be tested for P being a substring (a situation very common in bioinformatics), the **Preprocessing** step shall only be executed once.

Finite automata. A ***finite automaton*** M is a tuple

$$M = \langle Q, q_0, A, \Sigma, \delta \rangle,$$

where

- $Q = \{q_1, \dots, q_l\}$ is called a set of *states*.
- $q_0 \in Q$, is called *the start state*.
- $A \subseteq Q$ is the set of *accepting* or *final states*.
- $\Sigma = \{a_1, \dots, a_K\}$ is a finite *input alphabet*.
- $\delta : \Sigma \times Q \to Q$ is the *transition function*.

M accepts string $S = s_1 \dots s_n$ iff $\delta(s_n, \delta(s_{n-1}, \delta(\dots \delta(s_1, q_0) \dots) \in A$.

Finite Automata for substring matching. We want to design a finite automaton M_P for recognizing a given string $P = p_1 \dots p_m$ as a substring in the input stream. M_P is constructed as follows: $M_P = \langle Q, q_0, A, \Sigma, \delta \rangle$, where

- $Q = \{0, 1, \dots m\}$.
- $q_0 = 0$.

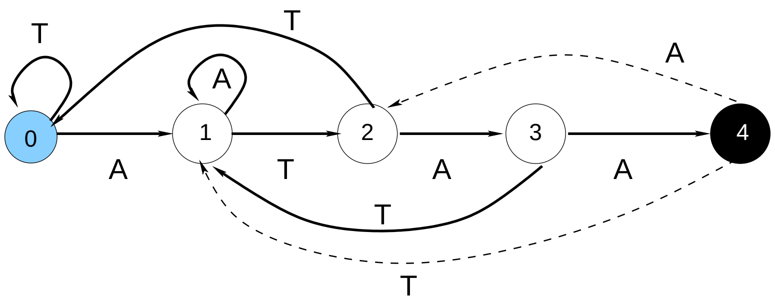

Figure 1: A finite automaton for matching against the string $P =$`ATAA`.

- $A = \{m\}$.
- $\Sigma = \{a_1, \ldots, a_K\}$.
- *delta* is constructed as described below.

Defining the transition function. Clearly, we want $\delta(p_i, i-1) = p_i$ for $i = 1, \ldots m$.

However, we cannot just say $\delta(a, i-1) = 0$ if $a \neq p_i$. We may have to move to a different state, because we may still be observing a valid non-empty prefix of P.

Example. Consider a string $P =$ `ATAA` and $S =$`ATATATAA`. We note that $p_1 = s_1 = A$, $p_2 = s_2 = T$ and $p_3 = s_3 = A$. However, $p_4 = A$, while $s_4 = T$, so $S[1,4] \neq P$:

```
S:   A T A T A T A A
P:   A T A a
```

However, at this point, $s_3 s_4 = AT$ form a valid non-empty prefix of P, so rather than trying to reapply the entire string P to $S[5,8]$, we must to check if $S[3,8]$ contains P, or, knowing that $s_3 s_4 = p_1 p_2$ — if $S[5,8]$ contains $p_3 p_4$ as its prefix:

```
S:   A T A T A T A A
P:   A T A a
P:       A T A a
```

Similar situation occurs on this step as well, and again $s_5 s_6 = p_1 p_2$, so we need to check if $S[7,8]$ has $p_3 p_4$ as its prefix:

```
S:   A T A T A T A A
P:   A T A a
P:       A T A a
P:           A T A A
```

The finite state machine M_P (in the $\{A, T\}$ alphabet for simplicity) is shown in Figure 1.

Constructing the transition function. A ***suffix function*** $\sigma : \Sigma^* \longrightarrow Q$ is defined as follows:

> $\sigma(x) = i$, where $p_1 \dots p_i$ is the **longest suffix** of x, i.e., if $x = x_1 \dots x_n$, then $\sigma(x) = i$ means that $x_{n-i} \dots x_n = p_1 \dots p_i$.

For a string $x \in \Sigma^*$, $\sigma(x) = m$ iff P is the suffix of x. We can now define the transition function δ for our automaton:

$$\delta(a, i) = \sigma(p_1 \dots p_i a).$$

Computing the transition function. The following algorithm computes δ directly according to the definition of the σ function.

```
ALGORITHM TransitionFunction(P, Σ, m)
begin
  for q= 0 to m   do     // iterate over states of the DFA
    foreach a ∈ Σ do     // iterate over input characters
      k ← min(m+1, q+2);     // determine max length of prefix of P
      repeat    // which prefix of P is a suffix of P[1,q]a?
        k ← k − 1;    // decrease the length of prefix of P
      until P[1,k]  is a suffix of  P[1,q]a;
      δ[q,a] ← k;
    end for
  end for
  return δ;
end
```

String matching. Given a computed transition function δ of the DFA M_P constructed for $P = p_1 \dots p_m$, and an input string S, we can check if S contains P (and report all appropriate shifts) using the following straightforward algorithm:

```
ALGORITHM DFAMatch(S, n δ, m)
begin
  q ← 0;    // Start in initial state
    for i = 1 to  n  do     // for each input character
      q ← δ(q, S[i]);    // transition to a new state
      if q = m  then     // accepting state reached?
        output i − m;    // start position of P in S is m positions to the right of i
      end if
  end for
end
```

Analysis. Algorithm TransitionFunction runs in $O(m^3 \cdot K)$ time. The outer loops repeat $m \cdot K$ times, the **repeat** loop runs for at most $m + 1$ times, and suffix check is an $O(m)$ operation.

Algorithm DFAMatch runs in $\Theta(n)$ time, so therefore, the total running time of a single DFA-based match is $O(n + m^3 \cdot K)$.

There is a faster way to compute δ which runs in $O(Km)$ time, which makes the overall complexity of DFA matching $O(n + Km)$. For small/fixed K (for example, for our nucleotide alphabet), this reduces to $O(n + m)$.

Knuth-Morris-Pratt Algorithm

Outline. The DFA string matching requires a precomputation of a full transition function. However, this function essentially contains "too much information". We can precompute in time $O(m)$ the essential information needed, but keep the actual matching process to $\Theta(n)$.

Example. Let P =`ATAT`. Consider a string S = `ATTAATAT`. Consider the initial attempt to align P against the first four characters of S (we show mismatches in lower case):

```
S:  A T T A A T A T
P:  A T a t
```

At this point, we know that $s_2 = T$ and $s_3 = T$. But $p_1 = A$, so *there is no need* to check if $S[2,5] = P$, as we know $s_2 \neq p_1$. Similarly, there is no need to check $S[3,6] = P$, because $s_3 \neq p_1$. The key is that we've already acquired enough information to draw the conclusion that the next check should start at position 4:

```
S:  A T T A A T A T
P:  A T a t
P:        A t a t
```

We learned that $s_5 = A = p_1$, so next check has to come against $S[5,8]$:

```
S:  A T T A A T A T
P:  A T a t
P:        A t a t
P:          A T A T
```

The Knuth-Morris-Pratt algorithm[3] finds a way to formalize these shortcuts and savings.

Prefix function. Given a string $P = p_1 \ldots p_m$, a **prefix function** $\pi : \{1, \ldots, m\} \longrightarrow \{0, \ldots, m-1\}$ is defined as follows:

$$\pi[q] = \max(k|k < q \text{ and } P[1,k] \text{ is a suffix of } P[1,q]).$$

Essentially, we shift P against itself and determine, when its prefixes appear as suffixes in the shift.

Example. Let $P = AATAAT$. We construct π as follows. For $q = 1$, $P[1,1] = A$, and the only prefix of P of length less than 1 is ϵ, so $\pi[1] = 0$.

For $q = 2$, $P[1,2] = AA$, the largest prefix of P of length of 1 or less, which is a suffix of $P[1,2]$ is $P[1,1] = A$:

```
 P[1,2]:  A A
 P     :    A | A T A A T
```

Therefore, $\pi[2] = 1$.

We continue shifting P against itself (vertical bar indicates the largest substring to try):

```
P[1,3]:  A A T
P     :        . A A | T A A T

P[1,4]:  A A T A
P     :          A A T | A A T

P[1,5]:  A A T A A
P     :          A A T A | A T

P[1,6]:  A A T A A T
P     :          A A T A A | T
```

This implies the following prefix function:

q:	1	2	3	4	5	6
π[q]	0	1	0	1	2	3
$P[1, \pi[q]]$	ϵ	A	ϵ	A	AA	AAT

Computing prefix function. The following algorithm computes the prefix function:

ALGORITHM PrefixFunction(P, m)
begin
  declare `int` $\pi[1, .., m]$;
  $\pi[1] \leftarrow 0$;
  $k \leftarrow 0$;
  for $q = 2$ **to** m **do**
    while $k > 0$ and $P[k+1] \neq P[q]$ **do**
      $k \leftarrow \pi[k]$;
    end while
    if $P[k+1] = P[q]$ **then**
      $k \leftarrow k + 1$;
    end if
    $\pi[q] \leftarrow k$;
  end for
  return π;
end

Knuth-Morris-Pratt Algorithm. The prefix function allows for the string matching algorithm to "know" how far it is possible to slide our pattern string P along the input string S.

```
ALGORITHM KMP-Match(S, P,n, m)
begin
  π[1,..,m] ← PrefixFunction(P,m);
  q ← 0;     //number of characters matched
  for i = 1 to  n  do
    while q > 0  and  P[q] ≠ S[i]  do
      q ← π[q];
    end while
    if  P[q+1] = S[i]  then
      q ← q+1;
    end if
    if  q = m  then
      output m − i;
      q ← π[q];
    end if
  end for
  return π;
end
```

Analysis. KMP-Match runs in $O(n)$ plus the time it takes to construct π. PrefixFunction runs in $O(m)$, and the total running time of KMP-Match is $O(n+m)$.

Boyer-Moore Algorithm

Idea. The Boyer-Moore algorithm[1] is well designed for the following types of problems:

- Large pattern string P.
- Large alphabet Σ.

While the first situation is not uncommon in bioinformatics, the traditional alphabet of nucleotides $\Sigma = \{A, T, C, G\}$ is small, and Boyer-Moore algorithm might not provide the best benefit. The amino acid alphabet has 20 characters in it, for this alphabet, Boyer-Moore is a good choice.

The algorithm is based on three key ideas.

Idea 1: Right-to-left scan. All previous algorithms compared P to a substring of S in a left-to-right fashion, first comparing p_1 to s_i, then p_2 to s_{i+1} and so on.

Boyer and Moore observed, that sometimes, one can get more information from a right-to-left comparison. E.g., consider a pattern string $P =$`AATAT` and a string $S =$`AATAGTGATCG`. When P aligned against $S[1,5]$, we get:

```
S:  A A T A G T G A T C G
P:  A A T A t
```

If we attempt a left-to-right match, we will successfully match $P[1,4]$ and $S[1,4]$ before discovering $p_5 \neq s_5$. We might need to consider matching P against $S[2,6]$ and against $S[4,8]$.

A *right-to-left match* with some extra knowledge about P can allow us to shift P much further. In fact, $s_5 = G$ and G is **not found** in P. Therefore, no string that includes G can match P. Thus, we can avoid comparing P to $S[2,6]$, $S[3,7]$, $S[4,8]$ and $S[5,9]$. Our next comparison will be:

```
S:  A A T A G T G A T C G
P:            A A T A t
```

By comparing s_{10} to p_5, we immediately reject $S[6,10] = P$. More importantly, $s_{10} = C$ and C is also **not** in P, so no substring including s_{10} can match P.

We have just discovered that S does not contain P using only two comparisons: p_5 vs. s_5 and p_5 vs. s_{10}, and scanning P to determine which characters are present in it.

Idea 2: Bad character rule. Example above illustrates how Boyer-Moore algorithm proceeds when $p_m \neq s_{i+m}$ when comparing P to $S[i, i+m]$. In a more general case, s_{i+m} may be present in P at one or more positons. We compute the requisite shift using a special helper function R.

Let $R : \Sigma \rightarrow \{1, \ldots m\}$ be defined as follows: $R(x)$ is the position of the **rightmost** occurrence of x in P. $R(x) = 0$ if x does not occur in P.

Example. Let $\Sigma = \{A, T, C, G\}$ and let $P =$`AATATTGAT`. Then, R is defined as follows:

$R(A) = 8$; $R(T) = 9$
$R(C) = 0$; $R(G) = 7$

$R(x)$ can be precomputed in linear time ($O(m + K)$) using the following straightforward algorithm:

```
ALGORITHM ComputeR(P,m,Σ,K)
begin
  for i = 1 to K do
    R[a_i] = 0;
  end for
  for i = 1 to m do
    R[P[i]] = i;
 end for
 return R;
end
```

R values are used in the **bad character shift rule**.

Let P be aligned against $S[i, i+m]$ and let, for some $j < m$, $p_{j+1} \ldots p_m = s_{i+j+1} \ldots s_{i+m}$, and $p_j \neq s_{i+j}$. Then, P can be shifted by $\max(1, j - R(s_{i+j}))$ places to the right, i.e., we can shift P in a way, that would align s_{i+j} with the rightmost appearance of the same character in P.

Example. Consider the situation below.

```
S:   A T G A C T G C A T A A T G
P:         A G C t C A
```

$p_4 = T$ is mismatched against $s_7 = G$, while $p_5 = s_8$ and $p_6 = s_9$. $R(T) = 2$, $j = 4$, and $\max(1, j - R(T)) = 2$, i.e, the next shift that needs to be considered, is the one, that aligns $p_2 = G$ with s_7:

```
 S:   A T G A C T G C A T A A T G
 P:         A G C t C A
 P:             A G C T C A
```

On the other hand, if the rightmost occurence of the "bad" character overshoots the current position, we shift only by one position, as illustrated below:

```
S: A T T G A T T A A T A G
P:       T c T T A
```

Here, $p_2 \neq s_5$, but $p_3 = s_6$, $p_4 = s_7$ and $p_5 = s_8$. $s_5 = A$, but in P, $R(A) = 5$, i.e., A appears as the last character in P, and it is currently to the right of s_5. We would like to align s_5 with the closest A in P (if any), but *we do not know where any other As are.* Because of this, we prefer a conservative shift by one position to the right:

```
S: A T T G A T T A A T A G
P:       T c T T A
P:         T C T T A
```

Idea 3. Good suffix rule. Let P be matched against $S[i, i + m]$, let, for some $j < m$, $p_{j+1} \ldots p_m = s_{i+j+1} \ldots s_{i+m}$, but $p_j \neq s_{i+j}$.

The substring $P' = p_{j+1} \ldots p_m$ is called *the good suffix*, because we were able to successfully match it against a substring in S.

*If there is another occurrence of the sequence of P' in P, **to the left of the suffix occurrence of** P', we can shift P to align that occurrence of P' with $s_{i+j+1} \ldots s_{i+m}$.*

Example. Consider $P =$`ATACGTACTTAC` and consider the following alignment:

```
S:      T G T A G A T A A G T A C T T A C T A C G T C G A
P:        G T A C T T A C t T A C
                  # * * * ^
```

P is matched against $S[2, 13]$ and, the good suffix of P is $p_{10}p_{11}p_{12} = TAC$. $p_9 = T$ is mismatched with $s_10 = G$. We notice that TAC occurs in P in a substring $p_6p_7p_8$. We can now shift P in a way that preserves the good suffix alignment against S:

```
S:      T G T A G A T A A G T A C T T A C T A C G T C G A
P:        G T A C T T A C T T A C
P:                G T A C T T A C T T A C
                          # * * *
```

A **strong good suffix rule** can go even further, and search for an occurrence of the good suffix, prefaced with a *different character*. In the example above $p_5 = p_9 = T$ is the character that prefaces the rightmost occurence of the good suffix, and the next occurrence to the left. But we already know that T

positioned against s_10 is a mismatch, so there is no need to attempt aligning $p_6p_7p_8$ against $s_{11}s_{12}s_{13}$. Instead, we can look further left to see if another occurrent of TAC exists, prefaced with anything other than a T. In our example, $p_2p_3p_4 = TAC$ and $p_1 = G$, so we can, indeed, attempt to align $p_2p_3p_4$ against $s_{11}s_{12}s_{13}$.

```
S:     T G T A G A T A A G T A C T T A C T T A C G T C G A
P:       G T A C T T A C T T A C
P:                         G T A C T T A C T T A C
                           # * * *
```

The good suffix rule heuristic can be precomputed using the following algorithm.

```
ALGORITHM ComputeGoodSuffixShift(P,m)
begin
  π ← PrefixFunction(P, m);
  P′ = reverse(P);
  π′ ← PrefixFunction(P′, m);
  for j = 0 to m do
    GS[j] ← m − π[m];
  end for
  for l = 1 to m do
    j ← m − π′[l];
    if GS[j] > l − π′[l] then
      GS[j] ← l − π′[l]
    end if
 end for
 return GS;
end
```

Boyer-Moore Algorithm. Boyer-Moore Algorithm on each step tries to align P and S left-to-right. When a mismatch occurs, *bad character rule* and *good suffix rule* provide suggestions for the next shift. The higher of the two is chosen.

```
ALGORITHM Boyer-Moore(S, n,P,m,Σ, K)
begin
  R ← ComputeR(P, m);
  GS ← ComputeGoodSuffixShift(P, m);
  s ← 0;
  while s ≤ n − m do
    j ← m;     //right-to-left scan
    while j > 0 and P[j] = S[s + j] do j ← j − 1;
    if j = 0 then
      output s;
      s ← s + GS[0];
    else
      s ← max(GS[j], j − R[S[s + j]]);
 end while
end
```

Gusfield's Z Algorithm

Dan Gusfield proposed a very elegant exact string matching algorithm [2].

Z function. Given a string $S = s_1 \dots s_i \dots s_n$, $Z_i(S) = k$ is the *length of the longest prefix* $s_1 \dots s_k$ that matches a a substring $s_i \dots s_{i+k}$.

Gusfield has shown that Z_i scores can be computed in $O(n)$ time.

Exact string matching using Z_i scores. Let $P = p_1 \dots p_m$ be a pattern string and $S = s_1 \dots s_n$ be the query string. Consider a string $Q = P\$S$ where "\$" is a symbol not found in either P or S.

We find all valid shifts of P in S by computing $Z_i(Q)$ for all $i = 1, \dots n+m+1$.

We note the following:

- Because "\$" is not found neither P nor S, $Z_i(Q) \leq m$ for $i > m+1$, i.e, no substring in S can have a prefix from Q that goes beyond P.
- Any time $Z_i(Q) = m$ for $i > m+1$, we are documenting an occurrence of P in S starting with position $i - m - 1$.

The algorithm then, computes $Z_i(Q)$ scores for $i = 1 \dots m+n+1$ and reports all positions i (as $i - m - 1$) such that $Z_i(Q) = m$.

References

[1] Robert S. Boyer, J. Strother Moore (1977), A Fast String Searching Algorithm, *Communications of the ACM*, Vol. 20, No. 10, pp. 762-772, October 1977.

[2] Dan Gusfield (1997), *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*, Cambridge University Press, 1997.

[3] Donald E. Knuth, James H. Morris, Vaughan R. Pratt, (1977), Fast Pattern Matching in Strings, *SIAM Jorunal on Computing*, Volume 6, pp. 323-350, 1977.

[4] Gonzalo Navarro, (2001), A Guided Tour to Approximate String Matching, *ACM Computing Surveys*, Vol. 33, No. 1, pp. 31–88, March, 2001.