Computational Biology

Exact Matching

Department of Mathematics
Stockholm University

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Notation

Let $S = x_1 \dots x_n$ be a string:

- $|S| = n$ length of S
- $S(i) = x_i$ character at position i
- $S[1..j] = x_1 \dots x_j$ prefix of S ending at position j
- $S[j..n] = x_j \dots x_n$ suffix of S starting at position j

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Notation

Let $S = x_1 \dots x_n$ be a string:

- $|S| = n$ length of S
- $S(i) = x_i$ character at position i
- $S[1..j] = x_1 \dots x_j$ prefix of S ending at position j
- $S[j..n] = x_j \dots x_n$ suffix of S starting at position j

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Notation

Let $S = x_1 \dots x_n$ be a string:

- $|S| = n$ length of S
- $S(i) = x_i$ character at position i
- $S[1..j] = x_1 \dots x_j$ prefix of S ending at position j
- $S[j..n] = x_j \dots x_n$ suffix of S starting at position j

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Notation

Let $S = x_1 \dots x_n$ be a string:

- $|S| = n$ length of S
- $S(i) = x_i$ character at position i
- $S[1..j] = x_1 \dots x_j$ prefix of S ending at position j
- $S[j..n] = x_j \dots x_n$ suffix of S starting at position j

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Notation

Let $S = x_1 \dots x_n$ be a string:

- $|S| = n$ length of S
- $S(i) = x_i$ character at position i
- $S[1..j] = x_1 \dots x_j$ prefix of S ending at position j
- $S[j..n] = x_j \dots x_n$ suffix of S starting at position j

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Notation

Let $S = x_1 \dots x_n$ be a string:

- $|S| = n$ length of S
- $S(i) = x_i$ character at position i
- $S[1..j] = x_1 \dots x_j$ prefix of S ending at position j
- $S[j..n] = x_j \dots x_n$ suffix of S starting at position j

If $P(i) = T(k)$ then they match at position i and k, resp. (else mismatch)

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

Exact matching has applications in

- Bioinformatics:
  Here, T a biological sequence database, e.g. of DNA sequences
- word processing
- internet search
- `fgrep` on UNIX
- search for plagiarism
- ...

Given: pattern P, text T (P, T are strings)

Aim: Find occurences of P in T

Example: $P = \texttt{ATG}$ occurs in $T = \texttt{AATGCATGCA}$ at position 2 and 6

We now have a closer look to the following methods:

- Naive Method
- Z-algorithm (pre-process P)
- Suffixtrees (pre-process T)

Naive Method

```
1: function NAIVE(P, T)
2:     occurences = ∅
3:     for i = 1, ..., |T| − |P| + 1 do
4:         for j = 1, ..., |P| do
5:             match = true
6:             if P(j) ≠ T(i + j − 1) then
7:                 match = false break
8:         if match then add i to occurences
9:     return occurences
```

Example and runtime : board

Naive Method

```
1: function NAIVE(P, T)
2:     occurences = ∅
3:     for i = 1, ..., |T| − |P| + 1 do
4:         for j = 1, ..., |P| do
5:             match = true
6:             if P(j) ≠ T(i + j − 1) then
7:                 match = false break
8:         if match then add i to occurences
9:     return occurences
```

Example and runtime : board

- before looking at text T examine internal structure of P
- preprocessing should be in $O(|P|)$ time
- several different algorithms use same fundamental preprocessing: Z-algorithm
- will later use preprocessing results to search P in T in $O(|T|)$.

We now continue on the board (slides 5 to 16 below summarize the content)

Z-algorithm

- before looking at text T examine internal structure of P
- preprocessing should be in $O(|P|)$ time
- several different algorithms use same fundamental preprocessing: Z-algorithm
- will later use preprocessing results to search P in T in $O(|T|)$.

We now continue on the board (slides 5 to 16 below summarize the content)

- before looking at text T examine internal structure of P
- preprocessing should be in $O(|P|)$ time
- several different algorithms use same fundamental preprocessing: Z-algorithm
- will later use preprocessing results to search P in T in $O(|T|)$.

We now continue on the board (slides 5 to 16 below summarize the content)

- before looking at text T examine internal structure of P
- preprocessing should be in $O(|P|)$ time
- several different algorithms use same fundamental preprocessing: Z-algorithm
- will later use preprocessing results to search P in T in $O(|T|)$.

We now continue on the board (slides 5 to 16 below summarize the content)

Preprocessing

We will preprocess a string S. S will later often play the role of P.

Definition 1 (Z values)

Let $i > 1$ be a position in string S.
$Z_i = Z_i(S)$ is the length of the longest substring of S that starts at i and matches a prefix of S.

Example 2

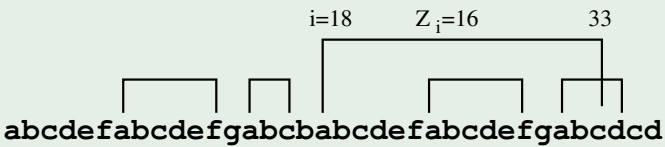
$S =$ `eiderdeiderlei`
$Z_7 = 5$
$Z_2 = 0$
$Z_{13} = 2$

Z-Box

Definition 3 (Z-Box)

Let $i > 1$ such that $Z_i > 0$. Then the Z-box at i is the interval starting at i and ending at $i + Z_i - 1$.

Example 4

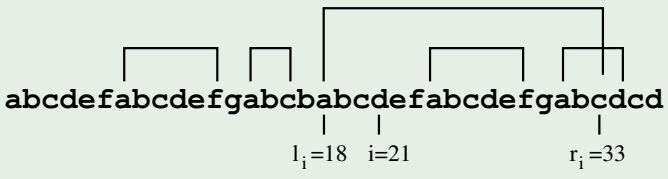


Z-Box

r_i, ℓ_i

Let $i > 1$. In the following $[\ell_i, r_i]$ will denote a Z-box that contains position i. If no Z-box contains position i, then $r_i < i$ (e.g. $r_i = 0$).

Example 5



Z Algorithm

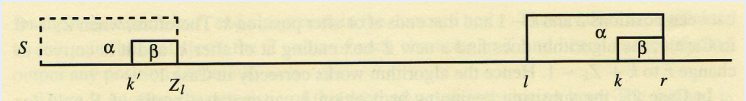
Z Algorithm

- preprocessing of S: compute all Z_i values
- direct approach takes time $O(|S|^2)$
- want to do it in $O(|S|)$
- will compute Z_k for increasing k
- Idea: When computing Z_k, r_k, ℓ_k we will reuse
  - $r = r_{k-1}, \ell = \ell_{k-1}$ and
  - previously computed values of $Z_2, \ldots, Z_{k-1}$.

  in order to save comparisons.

Z Algorithm: Idea

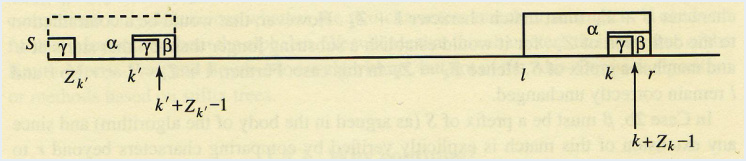
Reuse previously computed $Z_{k'}$



If $r \geq k$ then the string $\beta = S[k..r]$ occurs also at the end of the prefix of S of length Z_ℓ, starting at $k' = k - \ell + 1$ in S. We can use the value of $Z_{k'}$ to save comparison operations when determining Z_k.

Z Algorithm: Idea

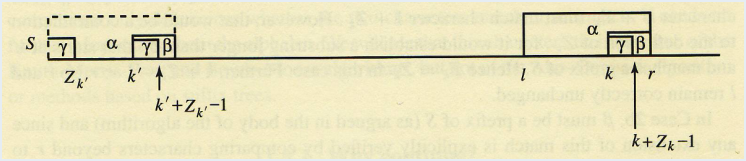
Case A



If $Z_{k'} < |\beta| = r - k + 1$, then the string $\gamma = S[1..Z_{k'}]$ starts also at k' and at k.
We get $Z_k = Z_{k'}$ without any further comparisons.

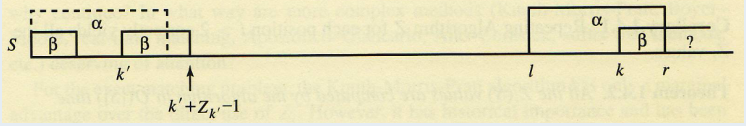
Z Algorithm: Idea

Case A



If $Z_{k'} < |\beta| = r - k + 1$, then the string $\gamma = S[1..Z_{k'}]$ starts also at k' and at k. We get $Z_k = Z_{k'}$ without any further comparisons.

Case B



If $Z_{k'} \geq |\beta|$, then Z_k is at least $|\beta|$, too. We make further comparisons starting at $r + 1$, but don't have to compare the characters in $[k..r]$.

Z Algorithm

1: $r \leftarrow \ell \leftarrow 0$
2: **for** $k = 2$ to $|S|$ **do**
3: // Case 1:
4: **if** $r < k$ **then**
5: Compute Z_k explicitly by comparing the characters starting at k to the characters starting at 1 until a mismatch is found.
6: Set Z_k to the length of the match.
7: **if** $Z_k > 0$ **then**
8: Set $r \leftarrow k + Z_k - 1$, $\ell \leftarrow k$.
9: // Case 2:
10: **else**
11: $k' \leftarrow k - \ell + 1$
12: // Case 2a:
13: **if** $Z_{k'} < r - k + 1$ **then**
14: $Z_k \leftarrow Z_{k'}$
15: // Case 2b:
16: **else**
17: $\ell \leftarrow k$
18: compare the characters starting at $r + 1$ with the characters starting at $r - k + 2$ of S until a mismatch occurs
19: let $q > r$ be the position of the first mismatch or $q = |S| + 1$ if no mismatch occurs
20: $Z_k \leftarrow q - k$, $r \leftarrow q - 1$

Proof of Correctness

Theorem 6 (Correctness of Z Algorithm)

The Z algorithm computes all values $Z_2, \ldots, Z_{|S|}$ correctly.

Proof.

(chalk board)

□

Running Time of Z Algorithm

Theorem 7 (Running Time of Z Algorithm)

The Z algorithm computes all the Z_i values in $O(|S|)$ time.

Proof.

(chalk board)

□

A Linear-Time Exact Matching Algorithm

Simple Linear-Time Exact Matching Algorithm

Require: character \$ not occuring in P or T

1: $n \leftarrow |P|$, $m \leftarrow |T|$
2: $S \leftarrow P\$T$
3: apply Z algorithm to S
4: **for** $i = n + 2$ to $m + 2$ **do**
5: **if** $Z_i = n$ **then**
6: output occurence of P starting at position $i - n - 1$ in T

A Linear-Time Exact Matching Algorithm

Theorem 8 (Correctness of Simple Exact Matching Algorithm)

Above algorithm reports all exact occurences of P in T.

A Linear-Time Exact Matching Algorithm

Theorem 8 (Correctness of Simple Exact Matching Algorithm)

Above algorithm reports all exact occurences of P in T.

Proof.

If $Z_i = n$ for any i in the range $n + 2 \leq i \leq m + 2$ then, by definition of Z_i, we have $S[1..n] = S[i..i + n - 1]$, and therefore, by definition of S, $P = T[i - n - 1..i - 2]$. Therefore, P occurs as substring in T at position $i - n - 1$.
Conversely, if P occurs starting at position j in T, then P occurs starting at position $j + n + 1$ in S and $i := j + n + 1$ is in the range $i = n + 2$ to $m + 2$. Therefore, $Z_i \geq n$. As \$ does not occur in T, we have $Z_i = n$ and position j is reported as occurence of P in T. □

A Linear-Time Exact Matching Algorithm

Theorem 9

Above algorithm runs in time $O(m)$ with $|T| = m$.

A Linear-Time Exact Matching Algorithm

Theorem 9

Above algorithm runs in time $O(m)$ with $|T| = m$.

Proof.

As shown previously, the Z algorithm takes time $O(|S|) = O(m + n) = O(m)$, since $n \leq m$. □

A Linear-Time Exact Matching Algorithm

Theorem 9

Above algorithm runs in time $O(m)$ with $|T| = m$.

Proof.

As shown previously, the Z algorithm takes time $O(|S|) = O(m + n) = O(m)$, since $n \leq m$. □

Space

Above algorithm can be implemented requiring $O(n)$ space in addition to storing P and T: We simply store the Z_i values only for $i \leq n$. Since \$ is not in T, we always have $Z_i \leq n$ and therefore k' from the Z algorithm is always less than or equal to n. We never need to recurse to a Z_i value for $i > n$.

There is another important algorithm for pattern matching: **Boyer-Moore Algorithm**
[skipped due to time-limitations; cf any standart bioinformatic book]

SuffixTrees

- data structure build from a string

SuffixTrees

- data structure build from a string
- will assume that the alphabet size is a constant

SuffixTrees

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$, $|T| = n, |P| = m$

SuffixTrees

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$, $|T| = n, |P| = m$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T

  This is reasonable as text T is often static and does not change (human genom, collected work of Shakespeare)

SuffixTrees

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$, $|T| = n, |P| = m$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T

  This is reasonable as text T is often static and does not change (human genom, collected work of Shakespeare)
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching

SuffixTrees

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$, $|T| = n, |P| = m$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T

  This is reasonable as text T is often static and does not change (human genom, collected work of Shakespeare)
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text

SuffixTrees

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$, $|T| = n, |P| = m$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T

  This is reasonable as text T is often static and does not change (human genom, collected work of Shakespeare)
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems

SuffixTrees

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$, $|T| = n, |P| = m$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T

  This is reasonable as text T is often static and does not change (human genom, collected work of Shakespeare)
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems

Now: Board