CMSC423: Bioinformatic Algorithms, Databases and Tools
Lecture 6

Sequence alignment: exact alignment

Z algorithm, KMP, Boyer-Moore

Homework 2 questions?

Reading assignment

- Check out the first three chapters in the textbook

Sequence alignment: exact matching

```
ACAGGTACAGTTCCCTCGACACCTACTACCTAAG    Text
CCTACT                                Pattern
 CCTACT
  CCTACT
   CCTACT
```

```
for i = 0 .. len(Text) {
  for j = 0 .. len(Pattern) {
    if (Pattern[j] != Text[i]) go to next i
  }
  if we got there pattern matches at i in Text
}
```

Running time = O(len(Text) * len(Pattern)) = O(mn)

Worst case?

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAT
```

(m – n + 1) * n comparisons

Can we do better?

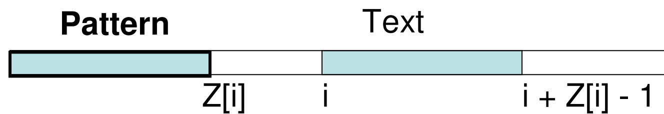
the Z algorithm (Gusfield)

For a string T, Z[i] is the length of the longest prefix of T[i..m] that matches a prefix of T. Z[i] = 0 if the prefixes don't match.

T[0 .. Z[i]] = T[i .. i+Z[i] -1]

Can the Z values help in matching?

Create string Pattern$Text where $ is not in the alphabet



If there exists i, s.t. Z[i] = length(Pattern)
Pattern occurs in the Text starting at i

Can Z values be computed in linear time?

AAAGGTACAGTTCCCTCGACACCTACTACCTAAG

Z[1]? compare T[1] with T[0], T[2] with T[1], etc. until mismatch

Z[1] = 2

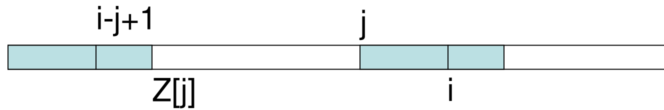
This simple process is still expensive:
T[2] is compared when computing both Z[1] and Z[2].

Trick to computing Z values in linear time:
each comparison must involve a character that was not compared before

Since there are only m characters in the string, the overall # of comparisons will be O(m).

Basic idea: 1-D dynamic programming

Can Z[i] be computed with the help of Z[j] for j < i?



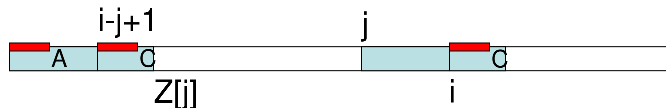
Assume there exists j < i, s.t. j + Z[j] – 1 > i
then Z[i – j + 1] provides information about Z[i]

If there is no such j, simply compare characters T[i..] to T[0..] since they have not been seen before.

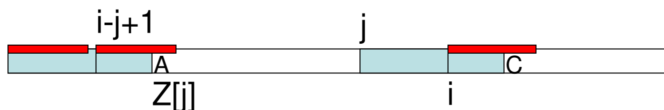
Three cases

Let j < i be the coordinate that maximizes j + Z[j] – 1
(intuitively, the Z[j] that extends the furthest)

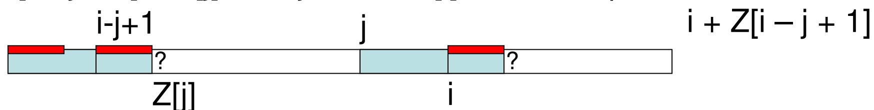
I. Z[i – j + 1] < Z[j] – i + j – 1 => Z[i] = Z[i – j + 1]



II. Z[i – j + 1] > Z[j] – i + j – 1 => Z[i] = Z[j] –i + j - 1

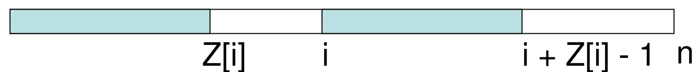


III. Z[i – j + 1] > Z[j] – i + j – 1 => Z[i] = ??, compare from i + Z[i – j + 1]



Z algorithm, not just for matching

- Lempel-Ziv compression (e.g. gzip)



if Z[i] = 0, just send/store the character T[i], otherwise, instead of sending T[i..i+Z[i] – 1] (Z[i] – 1 characters/bytes) simply send Z[i] (one number)

- Note: other exact matching algorithms used for data compression (e.g. Burrows-Wheeler transform relates to suffix arrays)

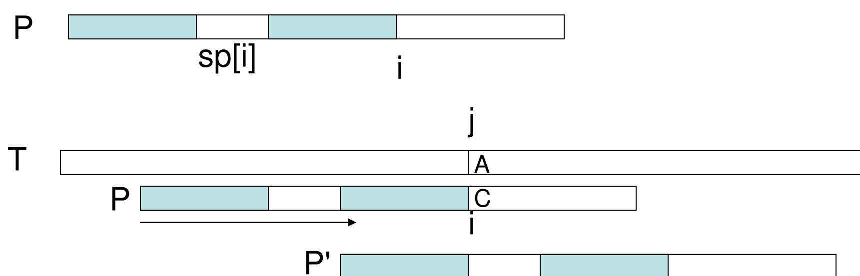
Knuth-Morris-Pratt algorithm

Given a Pattern and a Text, preprocess the Pattern to compute sp[i] = length of longest prefix of P that matches a suffix of P[0..i]

P

sp[i]

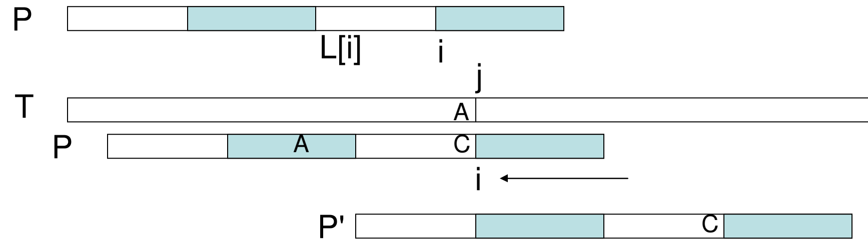
i



Compare P with T until finding a mis-match (at coordinate i + 1 in P and j + 1 in T). Shift P such that first sp[i] characters match T[j – sp[i] + 1 .. j]. Continue matching from T[i+1], P[sp[i]+1]

Boyer-Moore algorithm

Preprocess the pattern, computing, for every i, L[i] = largest coordinate < n, s.t. P[i..n] matches a suffix of P[1..L[i]] (inverted Z function)



Match the pattern backwards (starting at the right) until mismatch.
Shift the pattern such that P[L[i] – n + i + 1] matches at T[j]
Repeat.

Bad character rule: find character T[j – 1] in P and shift until it matches. Choose the longest shift (btwn. suffix & char. rules)