CMSC423: Bioinformatic Algorithms, Databases and Tools
Lecture 6

Exact string matching

Suffix trees

Suffix arrays

String matching

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)

What string achieves worst case?

Worst case?

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAT

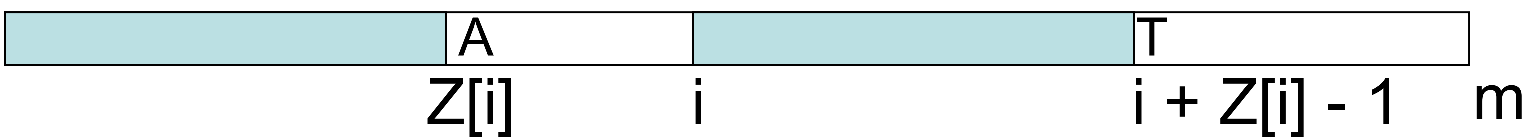
(m – n + 1) * n comparisons

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]

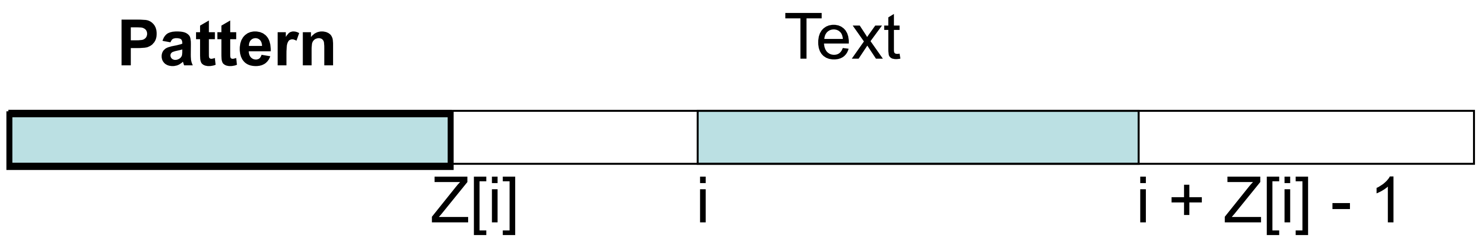


Example Z values

```
ACAGGTACAGTTCCCTCGACACCTACTACCTAAG
0010004010000000003010002002000110
```

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

example matching

```
CCTACT$ACAGGTACAGTTCCCTCGACACCTACTACCTAAG
0100100010000010000231010010610010041000 0
```

- What is the largest Z value possible?

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].

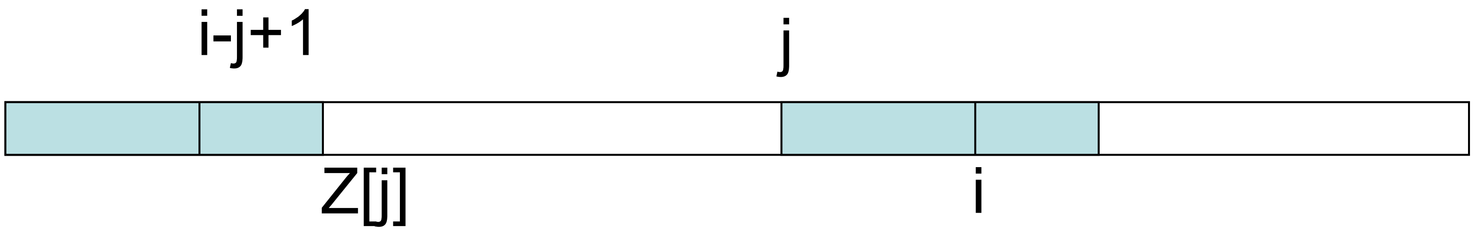
Trick to computing Z values in linear time:

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?



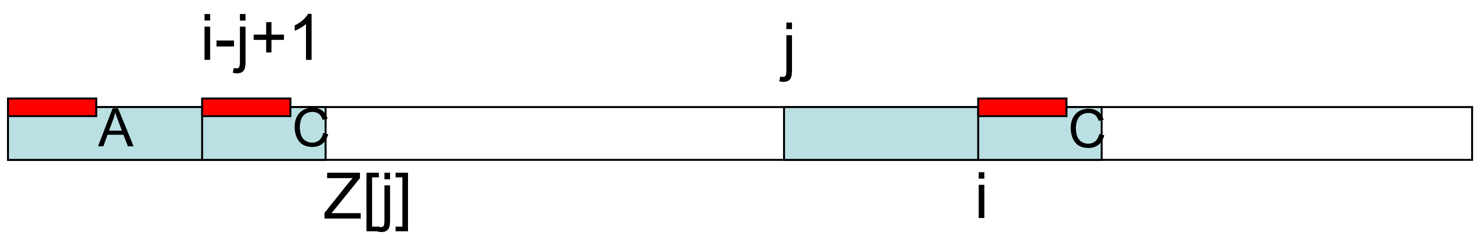
Assume there exists j < i, s.t. j + Z[j] – 1 > i
then Z[i – j + 1] provides information about Z[i]

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

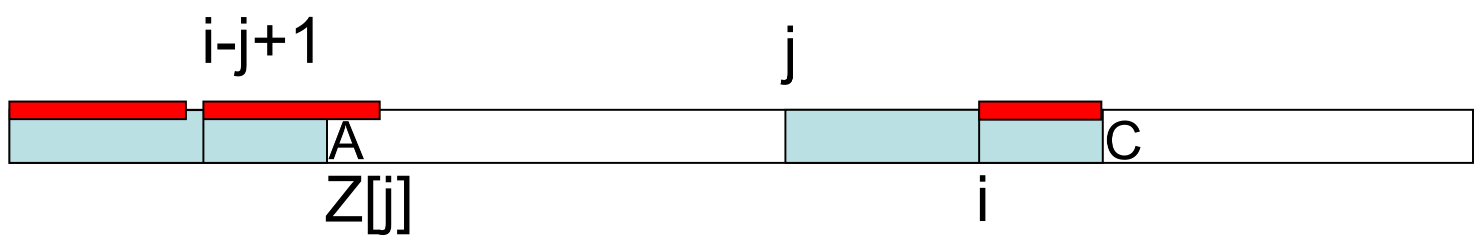
Three cases

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

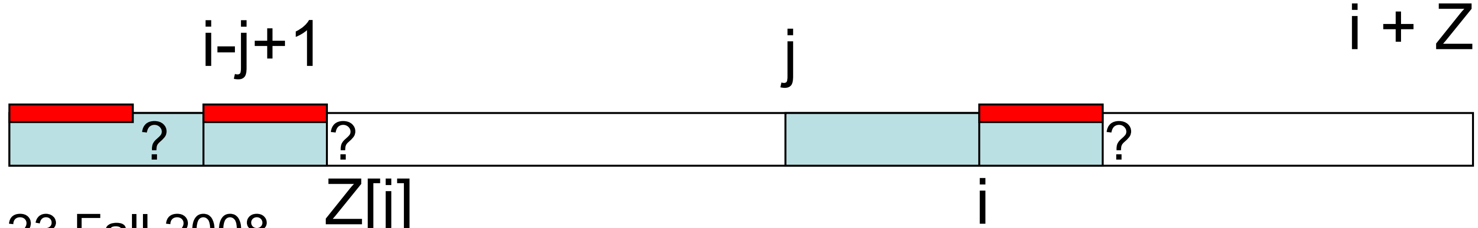
I. Z[i – j + 1] < Z[j] – i + j – 1 => Z[i] = Z[i – j + 1]



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]



Time complexity analysis

- Why do these tricks save us time?

1. Cases I and II take constant time per Z-value computed – total time spent in these cases is O(n)
2. Case III might involve 1 or more comparisons per Z-value

   however:

   - every successful comparison (match) shifts the rightmost character that has been visited
   - every unsuccessful comparison terminates the "round" and algorithm moves on to the next Z-value

   total time spent in III cannot be more than # of characters in the text

Overall running time is O(n)