# CMSC423: Bioinformatic Algorithms, Databases and Tools

Exact string matching:

KMP – analysis and computation of sp values

- Recap: The KMP algorithm uses suffix-prefix (sp) values to decide how far to shift the pattern along the text

- Here we analyze why the algorithm works, and describe an algorithm to compute sp values efficiently.

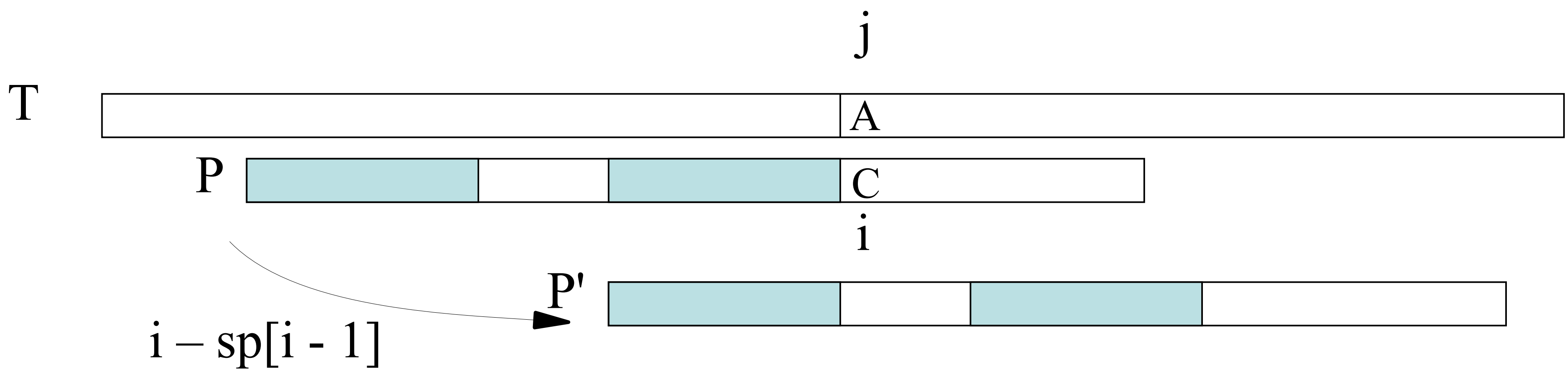
# KMP – speed

- How many character comparisons are made during the execution?

- If a character in the text matches a character in the pattern, do we have to look at it again?

- How many times can a character in the text fail to match the pattern?

# Run time analysis




Observation 1: after each shift, the prefix of the pattern matches the text. Only need to check whether T[j] matches.

Corollary: Once a character in the text matches the pattern, we no longer need to look at it.

Observation 2: We can get “stuck” mismatching T[j] over multiple rounds

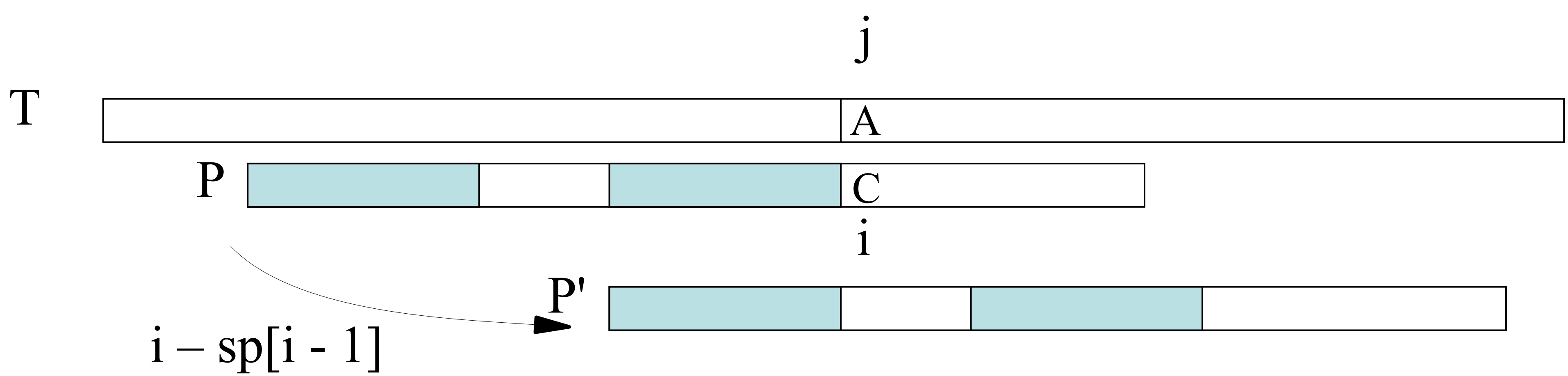
.

BUT: each time we do, the pattern shifts by at least one character.

# Runtime – putting it all together

- # of times a character in text matches the pattern: O(n) – length of text
- # of times a character in text mismatches the pattern: O(n) – after each mismatch the pattern advances to a new location

- Hence: Runtime(KMP) = O(n) + O(n) = O(n)

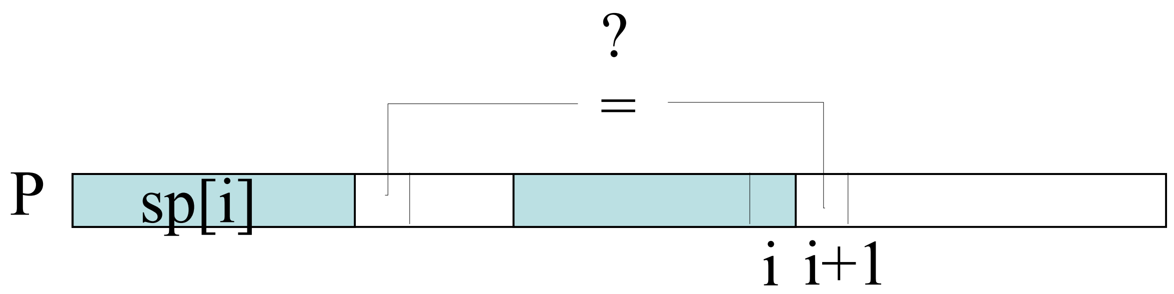


# KMP – computing sp values

- Can sp values be computed efficiently?

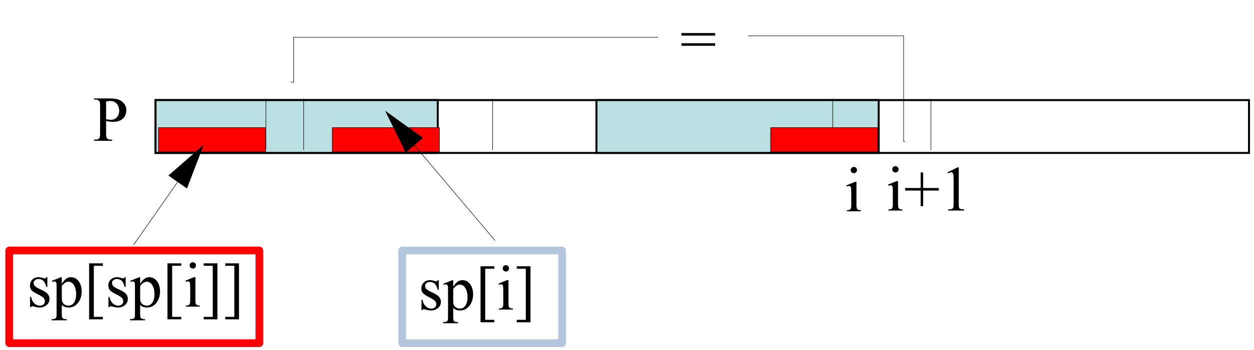

- Stop and think: Can you use Z values of P to compute the sp values?
- Stop and think: Can you use a similar algorithmic strategy (induction) as for computing the Z values?
- Stop and think: what is the relationship between sp[i] and sp[i + 1]?

# Computing sp values



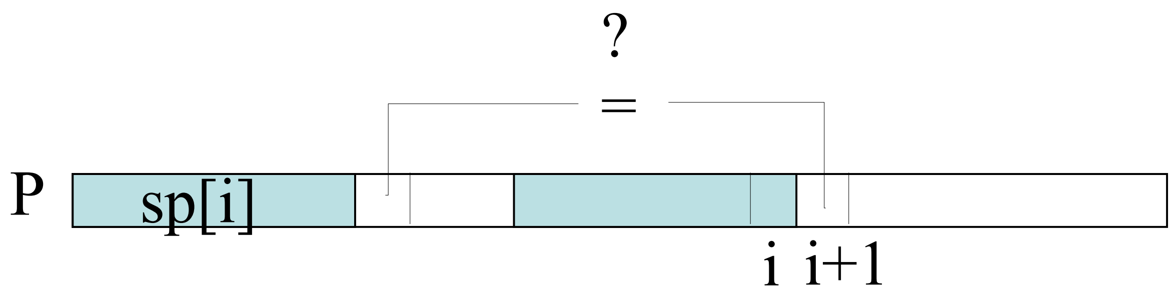

sp[i + 1] = sp[i] + 1, if and only if P[i+1] == P[sp[i]+1]

what if P[i+1] != P[sp[i]+1] ?




simply check P[i + 1] == P[sp[sp[i]]+1]
if yes, then sp[i + 1] = P[sp[sp[i]]] + 1
else
    repeat with sp[sp[sp[i]]] ...

# Computing sp values – runtime?

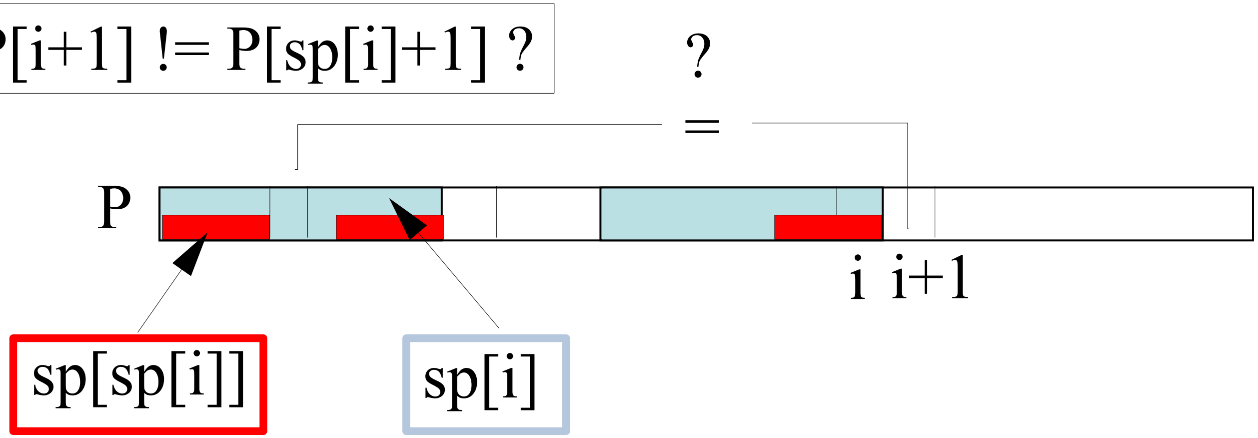



I. sp[i + 1] = sp[i] + 1, if and only if P[i+1] == P[sp[i]+1]

This case has one operation per sp value (linear)

what if P[i+1] != P[sp[i]+1] ?

II.

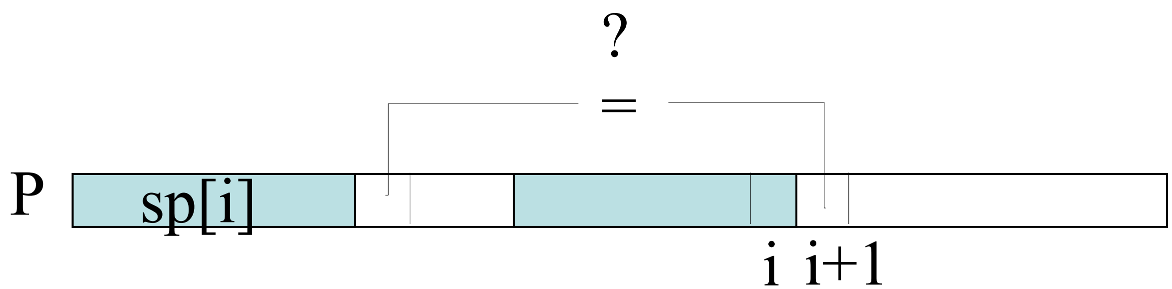



simply check P[i + 1] == P[sp[sp[i]]+1]
if yes, then sp[i + 1] = P[sp[sp[i]]] + 1
else
    repeat with sp[sp[sp[i]]] ...

This case may have an arbitrary number of operations.

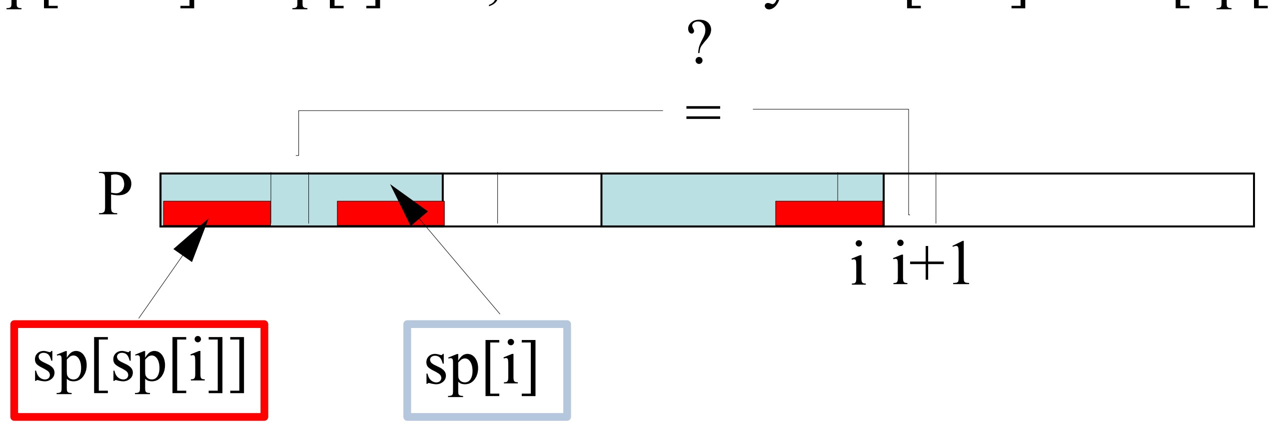
Worst case quadratic?

# A bank analogy




I. sp[i + 1] = sp[i] + 1, if and only if P[i+1] == P[sp[i]+1]

II.




simply check P[i + 1] == P[sp[sp[i]]+1]
if yes, then sp[i + 1] = P[sp[sp[i]]] + 1
else
    repeat with sp[sp[sp[i]]] ...

each iteration is a comparison
but...sp value becomes lower too

sp grows slowly – by 1 every time case I occurs

# The bank analogy

- sp grows by at most 1 per round – hence max(sp) <= len(P)
- in round i, # of comparisons <= sp[i]
- then it takes a while to regain “potential” in sp
- hence – runtime = O(len(p)) = O(m)

- In bank terms, if you are paid 1$/day, you cannot spend more than $7/week

# The End (or is it?)

- More exact matching in Chapter 9

- In preparation, Stop and Think!

  Can you find a linear time algorithm to find the longest match between a prefix of a pattern and the text? The whole pattern match (this module) is a special case

  Can you find a linear time algorithm that finds the longest match (not restricted to the beginning of the pattern) between the pattern and the text?