# Algorithms for matching

*Computational Linguistics,*
*Summer Semester 2010*

*Pierre Lison*

*(based on slides from Geert-Jan M. Kruijff)*

⟨plison,gj@dfki.de⟩

# Today's lecture

- Objective:
  - Efficient algorithms for finding matches of *patterns* (strings) in *texts*.
  - The focus is on *exact* matching
    - But we'll also quickly review inexact matching in the last part of the lecture
  - We deal with chars/Strings, but this generalizes to words/Strings
- Why efficient methods for pattern matching?
  - Applications of pattern matching in *search* (web search for IR, IE, Q/A), *tagging* (named entity recognition), *shallow processing* (parsing)
  - Efficiency pays off when dealing with large amounts of data!
  - Furthermore: preliminaries for finite-state automata, dynamic programming/memoization techniques in parsing

# What will you learn?



1. The naive method for exact string matching
    - Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons
2. Methods for fundamental preprocessing of a pattern
    - Pre-process the pattern to make smarter shifts (i.e. longer ones) when a mismatch is found
3. The Booyer-Moore algorithm
    - Smart shifts in sublinear $O(|P|+|T|)$ time (B-M) thanks two complementary rules: the bad character rule and the good suffix rule
4. Inexact matching
    - The *edit distance* algorithm

**Reference:** Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. CUP, 1997: Chapters 1 & 2

# Outline

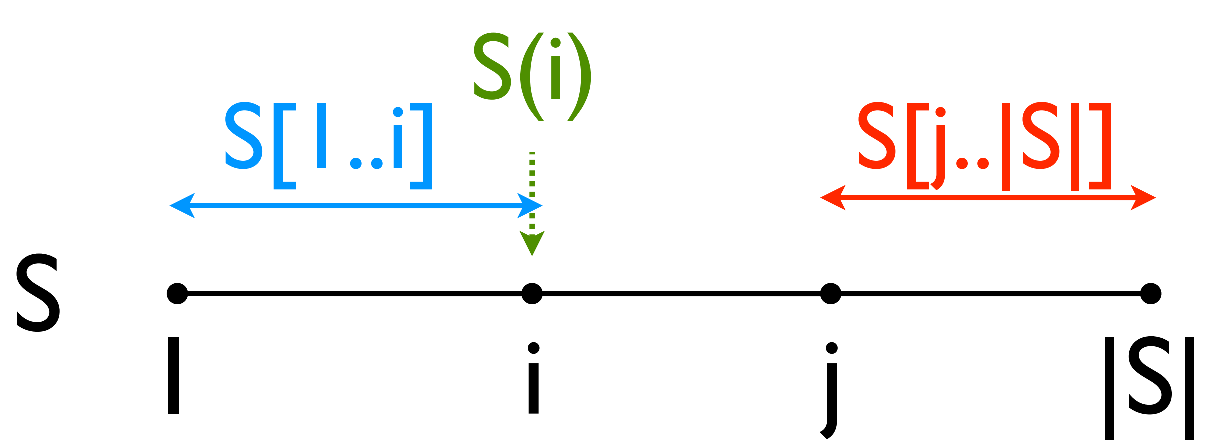


1. **The naive method for exact string matching**
   - **Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons**
2. Methods for fundamental preprocessing of a pattern
   - Pre-process the pattern to make smarter shifts (i.e. longer ones) when a mismatch is found
3. The Booyer-Moore algorithm
   - Smart shifts in sublinear $O(|P|+|T|)$ time (B-M), thanks to 2 complementary rules: the bad character rule and the good suffix rule
4. Inexact matching
   - The *edit distance* algorithm

# Preliminaries



- A **string** S is an ordered list of characters, written contiguously from left to right. For any string *S*, S[i..j] is the (contiguous) **substring** of S that starts at position i and ends at position j.
- The substring S[1..i] is the **prefix** of S that ends at position i, and the substring S[j..|S|] is the **suffix** of S starting at position i, with |S| the length of S.
- For any string S, S(i) denotes the **character** at position i in S.



# The naive method for matching

- Given
  - a **pattern** P, and a text **T** in which we are looking for matches of **P**
  - **Pointers:** *p* to position in P; *t* to position in T; *s* to start of matching P in T
- Algorithm

[Start: p=1, t=1,s=1]

1. Align the left of P with the left of T: set position in P, p=1; set position in T, t=1
2. Set the current left-alignment position in T to s=1

[Loop]

3. Compare the character at P(p) with the character at T(t)
4. **If** P(p) == T(t):

   If p < |P| then set p=p+1 and set t=t+1; else report match, and set p=1, s=s+1, t=s;

   **Else** p=1 and s=s+1, t=s

# The naive method for matching



1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

| p=1 | p=2 | p=3 | p=4 | p=5 | p=6 | p=7 | p=7 | ⟹ | p=1 |
|---|---|---|---|---|---|---|---|---|---|
| t=2 | t=3 | t=4 | t=5 | t=6 | t=7 | t=8 | t=8 | | t=3 |
| s=2 | s=2 | s=2 | s=2 | s=2 | s=2 | s=2 | s=2 | | s=3 |

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✗

A B X Y A B X Z

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

[Start: p=1, t=1,s=1]

1. Align the left of P with the left of T:
2. p=1; t=1; s=1

[Loop]

2. Compare P(p) with T(t)
3. If P(p) == T(t):
   - If p < |P|:
     - p=p+1 and t=t+1;
   - Else: report match, and p=1, s=s+1, t=s;
   
   Else: p=1 and s=s+1, t=s

In total,
we had to make
**20**
comparisons

# The naive method for matching



- Observations
  - The worst-case number of comparisons is $O(|P| \cdot |T|)$
  - This is not so useful in real-life applications!
  - E.g. |P|=30 and |T|=200K: 6M comparisons; with 1ms per comparison this would mean 6000s, or 100 minutes, i.e. 1:40h. If we manage to get linear complexity $O(|P|+|T|)$ we are down to 3.33min!
- Ideas for speeding up the naive method
  - Try to shift further when a mismatch occurs, but never so far as to miss an occurrence of P in T

# Speeding up thru smarter shifting



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T= | X | A | B | X | Y | A | B | X | Y | A | B | X | Z |
| P= | A | B | X | Y | A | B | X | Z | | | | | |
| | ✘ | | | | | | | | | | | | |
| | | A | B | X | Y | A | B | X | Z | | | | |
| | | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ | | | | |
| | | | | | | A | B | X | Y | A | B | X | Z |
| | | | | | | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |

The next occurrence of P(1)="A" in T is not before position 5 in T, so shift to position 6!

In total, we had to make **17** comparisons

# Speeding up thru smarter shifting

1 2 3 4 5 6 7 8 9 10 11 12 13

T= X A B X Y A B X Y A B X Z

P= A B X Y A B X Z

✗

A B X Y A B X Z

✓✓✓✓✓✓✓✗

Assume: we know prefix P[1..3]="A B X" starts at T(6). P[1..3]=T[6..8]; align at T(6) but start matching P(4) against T(9)

A B X Y A B X Z

✓✓✓✓✓

In total, we had to make **14** comparisons

# Outline

1. The naive method for exact string matching
   - Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons
2. **Methods for fundamental preprocessing of a pattern**
   - **Pre-process the pattern to make smarter shifts (i.e. longer ones) when a mismatch is found**
3. The Booyer-Moore algorithm
   - Smart shifts in sublinear $O(|P|+|T|)$ time (B-M), thanks to 2 complementary rules: the bad character rule and the good suffix rule
4. Inexact matching
   - The *edit distance* algorithm

# Smarter shifting thru preprocessing

- Before searching, preprocess P (or T, or P+T)
- Fundamental preprocessing of a string S
  - At S(i), i > 1 compute length of longest prefix of S[i..|S|] that is a prefix of S
  - Let $Z_i(S)$ be that length at i

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| S = | A | A | B | C | A | A | B | X | A | A | Z |

$Z_5(S)=3$: (A A B C...A A B X)

$Z_6(S)=1$: (A A ...A B)

$Z_7(S)=Z_8(S)=0$

$Z_9(S)=2$: (A A B ...A A Z)

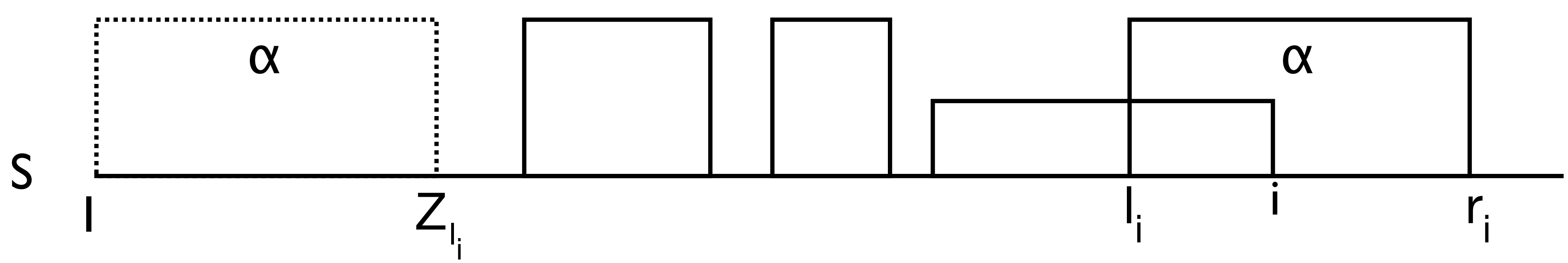
# Smarter shifting thru preprocessing



- Given a string S=P$T
  - The dollar sign $ is not in the languages for P or T
  - $|P|=n$, $|T|=m$, $n \leq m$, so $S=n+m+1$
- Compute $Z_i(S)$ for $2 < i < n+m+1$
  - Because “$” is not in the language for P, $Z_i(S) \leq n$ for every $i > 1$
  - $Z_i(S)=n$ for $i > n+1$ identifies an occurrence of P starting at $i-(n+1)$ in T
  - Also: If P occurs in T starting at position j, then it must be that $Z_{(n+1)+j}(S)=n$
- If $Z_i(S)$ is computable in linear time, then we have linear time matching
  - Matching = search ⇒ matching = preprocessing + search

# Computing $Z_i(S)$ in linear time

- The task: Compute $Z_i(S)$ in linear time, i.e. $O(|S|)$
- The notion of a **Z-box**
  - For every $i > 1$ with $Z_i(S) > 0$, define a Z-box to be the substring from $i$ until $i+Z_i(S)-1$, i.e. $S[i...i+Z_i(S)-1]$
  - For every $i > 1$, $r_i$ is the right-most endpoint of the Z-boxes that begin at or before $i$;
  - i.e, $r_i$ is the largest value of $j+Z_j(S)-1$ for all $1 < j \leq i$ such that $Z_i(S) > 0$



# Sketch of the algorithm

- We need to compute $Z_i(S)$, $r_i$ and $l_i$ for every i > 2
- In any iteration i, we only need $r_j$ and $l_j$ for j=i-1; i.e just r, l
- If we discover a new Z-box at i, set r to the end of that Z-box, which is the right-most position of any Z-box discovered so far
- Step 0 (initialisation)

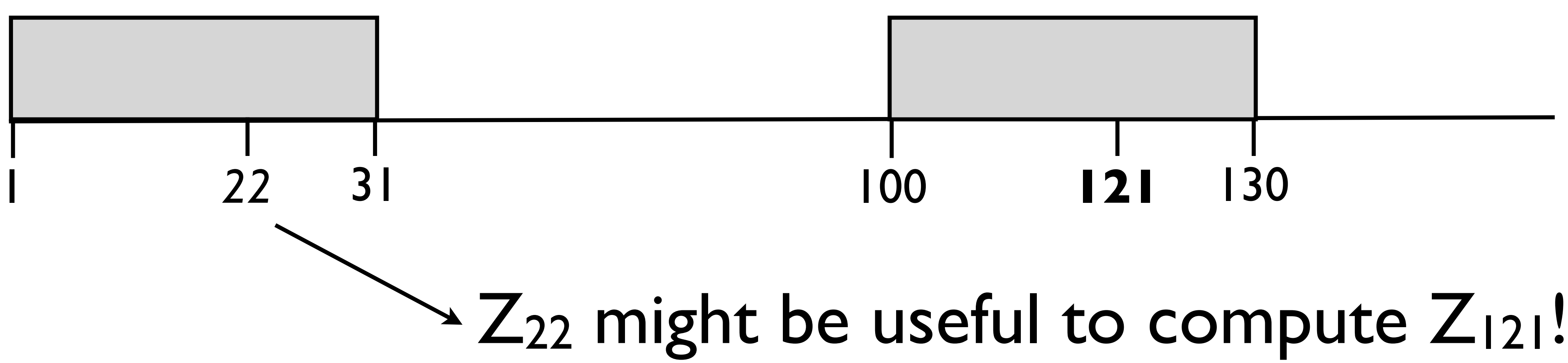
  Find $Z_2(S)$ by comparing left to right S[2..|S|] and S[1..|S|] until a mismatch is found; $Z_2(S)$ is the length of that string. If $Z_2(S) > 0$ then set $r=r_2$ to $Z_2(S)+1$ and $l=l_2$, else r=l=0

- *Induction hypothesis*: we have correct $Z_i(S)$ for i up to k-1>1, r, l
  - Next, compute $Z_i(S)$ from the already computed Z values

# Compute $Z_i(S)$ from $Z_j(S)$, $2 < j < i$



- Simplest case: inclusion
- E.g. for k=121, we have $Z_2(S) \ldots Z_{120}(S)$, and $r_{120}=130$, $l_{120}=100$
  - Thus: a substring of length 31 starting at position 100, matching S[1..31]
  - And: the substring of length 10 starting at 121 must match S[22..31], so $Z_{22}$ could help!
  - For example, if $Z_{22}$ is 3, then $Z_{121}$ must also be 3



# Compute $Z_i(S)$ from $Z_j(S)$, $2 < j < i$

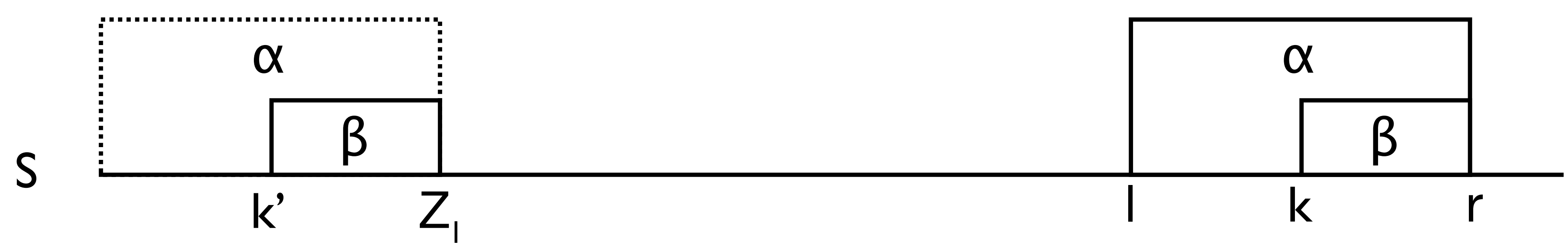


- Given $Z_i(S)$ for all $1 < i \leq k-1$, and the current values of $Z_k(S)$, r, and l; compute the updated r and l
- Step 1:
  - if $k > r$, then find $Z_k(S)$ by comparing the characters starting at k to the characters starting at position 1 in S, until a mismatch is found. The length of the match is $Z_k(S)$. If $Z_k(S) > 0$, set $r=k+Z_k(S)-1$, and $l=k$.

# Compute $Z_i(S)$ from $Z_j(S)$, $2 < j < i$



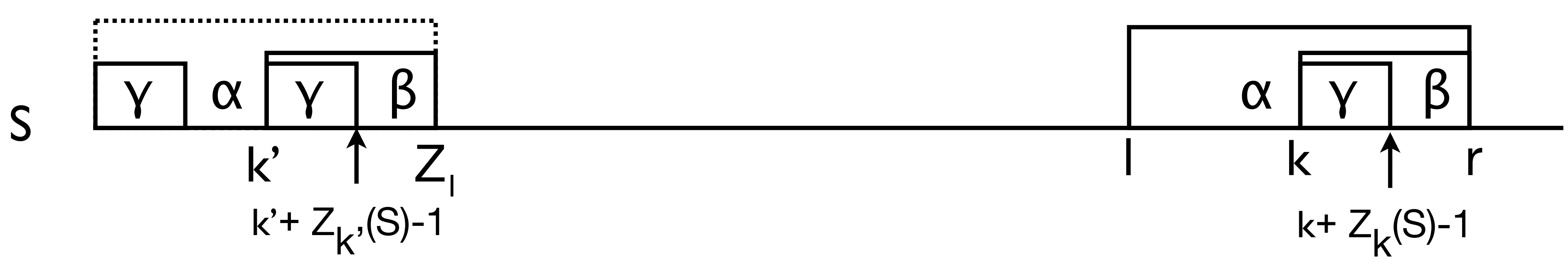
- Step 2
  - If $k \leq r$, then position k is contained in a Z-box, and hence S(k) is contained in a substring S[l..r] (call it α) such that l > 1 and α matches a prefix of S.
  - Therefore, character S(k) also appears in position k'=k-l+1 of S.
  - By the same reasoning, the substring S[k..r] (call it β) must match substring S[k'..$Z_l$(S)]. (*Remember the example with $Z_{22}(S)$, r=121!*)
  - Hence, the substring at position k must match a prefix of S of length at least the *minimum* of $Z_{k'}(S)$ and |β| (which is r-k+1).



# Two cases given the minimum



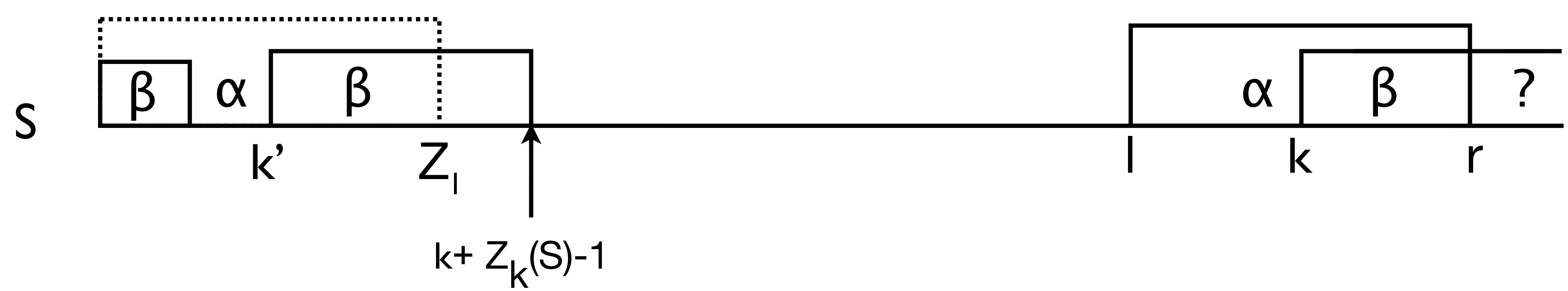
- Case 1: If $Z_{k'}(S) < |\beta|$
  - then position k is a Z-box (call it γ) contained within a larger Z-box
  - set $Z_k(S)=Z_{k'}(S)$ and leave r and l as they are



# Two cases given the minimum



- Case 2: If $Z_{k'}(S) \geq |\beta|$
  - then the entire substring S[k..r] must be a prefix of S and $Z_k(S) \geq |\beta| = r-k+1$
  - However, $Z_k(S)$ may be strictly larger, so compare characters starting at r +1 of S to the characters starting at $|\beta|+1$ of S until a mismatch occurs (*Remember the second smart improvement over the naive method!*)
  - Say the mismatch is at $q \geq r+1$. Then $Z_k(S)=q-k$, $r=q-1$, and $l=k$



# Linear time computation



- "The algorithm computes all the $Z_i(S)$ values in $O(|S|)$ time"

  The time is proportional to the number of iterations, |S|, plus the number of character comparisons. Each comparison is either a match or a mismatch. Each iteration that performs any character comparisons at all ends the first time it finds a mismatch; hence there are at most |S| mismatches during the entire algorithm. To bound the number of mismatches, note first that $r_k \geq r_{k-1}$ for every iteration k. Now, let k be an iteration where $q > 0$ matches occur. Then $r_k$ is set to $r_k+q$ at least. Finally, $r_k \leq |S|$ so the total number of matches that can occur during any execution of the algorithm is at most |S|.

- "Computing $Z_i(S)$ on S=P$T finds matches of P in T in $O(|T|)$"

# Outline



1. The naive method for exact string matching
   - Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons
2. Methods for fundamental preprocessing of a pattern
   - Pre-process the pattern to make smarter shifts (i.e. longer ones) when a mismatch is found
3. **The Booyer-Moore algorithm**
   - **Smart shifts in sublinear $O(|P|+|T|)$ time (B-M), thanks to 2 complementary rules: the bad character rule and the good suffix rule**
4. Inexact matching
   - The *edit distance* algorithm

# The Boyer-Moore algorithm



- Like the naive method
  - Align P with T, check whether characters in P and T match
  - After the check is complete, P is shifted rightwards relative to T
- Smarter shifting
  - For an alignment, check whether P occurs in T scanning right-to-left in P
  - The **bad character** shift: shift right beyond the bad character
  - The **good suffix** shift: shift right using the match of the good suffix of P

# Boyer-Moore: right-to-left scan

- For any alignment of P against T, check P **right-to-left**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T = | X | P | B | C | T | B | X | A | B | P | Q | X | C | T | B | P | Q |
| P = | | | T | P | A | B | X | A | B | | | | | | | | |
| | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | | | |
| | | | | | ✘ | ✔ | ✔ | ✔ | ✔ | | | | | | | | |

- For example,
  - P(7)=T(9) ... but P(3) ≠ T(5)
  - Upon a mismatch, shift P right relative to T
- The linear nature of the algorithm is in the shifts
  - Scanning right-to-left still yields an algorithm running in O(nm) time

# B-M: The bad character shift



- The basic idea
  - Suppose the rightmost character in P is y, aligned to x in T with x≠y
  - If x is in P, then we can shift P so that the rightmost x is below x in T
  - If x is not in P, then we can shift P completely beyond the x in T
- Possibly *sublinear* matching: not all characters in T may need to be compared
- Very efficient for natural language text, esp. English

# B-M: The bad character shift



- Store the right-most position of each character

  For each character x in the alphabet, let R(x) be the rightmost position of x in P. R(x) is defined to be 0 if x is not in P.

- The **bad character shift rule** makes use of R

  Suppose for an alignment of P against T, the rightmost n-i characters of P match against T, but the character at P(i) is a mismatch with the character T(k). Now, we can shift P right by *max*[1,i-R(T(k))] places; i.e. if the right-most occurrence in P of the character T(k) is in position j < i (possibly with j=0), then shift P so that the character j of P is below character k of T. Else, shift P by 1.

```
        1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
T =     X P B C T B X A B P  Q  X  C  T  B  P  Q
P =         T P A B X A B                          R("T")=1
            1 2 3 4 5 6 7
                ✘ ✔ ✔ ✔ ✔
                T P A B X A B
```

# B-M: the good suffix rule

- The basic idea:
  - Given the character T(k) against which P mismatches,
  - Take the good suffix t of P, i.e. the part that matched against T
  - Look in P for the right-most copy t’ of t, such that the character k’ to the immediate left of t’ differs from T(k); *else the shift would yield the same mismatch!*
  - Then, shift P to the right such that t’ is below the matching t in T.

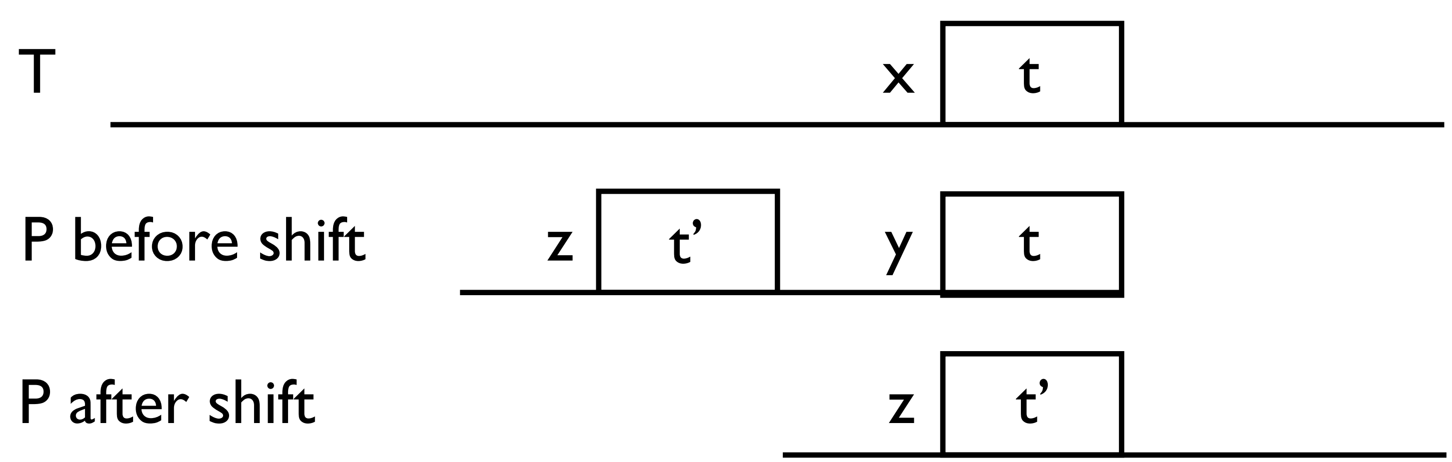
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T = | P | R | S | T | A | B | S | T | U | B | A | B | V | Q | X | R | S | T |
| P = | | Q | C | A | B | D | A | B | D | A | B | | | | | | | |
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | | | | | |
| | | | | | | | | | | ✘ | ✔ | ✔ | | | | | | |
| | | | | | | | | | Q | C | A | B | D | A | B | D | A | B |

# B-M: the good suffix rule



Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and the character to the left of t' in P differs from the character to the left of t in P. Shift P to the right so that the substring t' in P is below substring t in T. If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T.

If no such shift is possible, then shift P by n places to the right. If an occurrence of P is found, then shift P by the least amount so that a proper prefix of the shifted P matches a suffix of the occurrence of P in T. If no such shift is possible, then shift P by n places, past t in T.



# Preprocessing



- We need some preprocessing for the good suffix rule
  - We need to compute the positions of copies of suffixes of P
  - whereby a copy differs from the suffix in its immediate left character
- Definition

For each i, L(i) is the largest position less than n such that string P[i..n] matches a suffix of P[1..L(i)]. Let L(i) be zero if there is no position satisfying the conditions. For each i, L'(i) is the largest position less than n such that string P[i..n] matches a suffix of P[1..L'(i)] and such that the character preceding the suffix is not equal to P(i-1). Let L'(i) be 0 if there is no position satisfying the conditions.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| P = | C | A | B | D | A | B | D | A | B | L(8)=6 L'(8)=3 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |

# Preprocessing



- Computing L'(i)
  - For string P, $N_j(P)$ is the length of the longest suffix of the substring P[1...j] that is also a suffix of the full string P.

P = C $\boxed{\text{A B D A B}}$ D A B

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| C | A | B | D | A | B | D | A | B |

$N_3(P)=2$ $N_6(P)=5$

- We can compute $N_i(S)$ from $Z_i(S)$
  - Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and is a *prefix* of S
  - $N_i(S)$ is the reverse of Z: if $P^r$ is the reverse of P, then $N_j(P)=Z_{n-j+1}(P^r)$
  - Hence we can obtain the values for N using the linear algorithm for Z

# Preprocessing: from N to L'

- Z-based Boyer-Moore for obtaining $L'(i)$ from $N_i(P)$

```
for i := 1 to n do L'(i) := 0
for j := 1 to n-1 do
    begin
    i := n - N_j(P) + 1
    L'(i) := j
    end
```

- Intuition
  - We have computed the lengths of the longest suffixes as $N_j(P)$
  - Cycle over P right-to-left, looking at where the longest suffixes start
  - Assign to $L'(i)$ the largest index j such that $N_j(P) = |P[i..n]| = (n-i+1)$
  - Those $L'(i)$ for which there is no such index have been initialized to 0.

# The final preprocessing detail ...

- Let l'(i) denote the longest suffix of P[i..n] that is also a prefix of P, if one exists. If none exists, let l'(i) be zero.
- Once more, all the preprocessing and rules:
  - Bad character rule: given a mismatch on x in T, shift P right to align with an x in P (if any)
    - Compute R(x), the right-most occurrence of x in P
  - Good suffix rule: shift P right to a copy of the matching suffix but with a different character to its immediate left
    - Use $Z_j(P)$ to compute $N_j(P)$, the length of the longest suffix of P[1..j] that is a suffix of P
    - Use $N_j(P)$ to compute L'(i), the largest position less than n s.t. P[i..n] matches a suffix of P[1..L'(i)]
    - Compute l'(i), to deal with the case when we have L'(i) = 0 or when an occurrence of P is found

# The Boyer-Moore algorithm

## [Preprocessing stage]

Given the pattern P

Compute L'(i) and l'(i) for each position i of P

and compute R(x) for each character $x \in \Sigma$

## [Search stage]

```
k := n
while k ≤ m do
    i := n
    h := k
    while i > 0 and P(i) = T(h) do
        i := i-1
        h := h-1
    if i = 0 then
        report an occurrence of P in T ending at position k
        k := k+n - l'(2)
    else
        shift P (increase k) by the maximum amount determined by the bad character rule
        and the good suffix rule
```

# Outline



1. The naive method for exact string matching
   - Method for finding matches of a pattern P in a text T using $O(|P| \cdot |T|)$ comparisons
2. Methods for fundamental preprocessing of a pattern
   - Pre-process the pattern to make smarter shifts (i.e. longer ones) when a mismatch is found
3. The Booyer-Moore algorithm
   - Smart shifts in sublinear $O(|P|+|T|)$ time (B-M), thanks to 2 complementary rules: the bad character rule and the good suffix rule
4. **Inexact matching**
   - **The *edit distance* algorithm**

# Inexact matching



- So far: *exact* matching problem
  - Inexact matching: approximation of pattern in text
  - From substring to subsequence matching
- The **edit distance** between two strings
  - Transformation: insertion, deletion, substitution of material

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| R | I | M | D | M | D | M | M | I |
| v | | i | n | t | n | e | r | |
| w | r | i | | t | | e | r | s |

  - A string over the alphabet I, D, R, M, that describes a transformation of one string to another is called an *edit transcript* of the two strings

# Edit distance



- Edit distance

  The **edit distance** between two strings is defined as the *minimum* number of edit operations - insert, delete, substitute - needed to transform the first string into the second. (Matches are not counted.)

- The edit distance problem

  The edit distance problem is to compute the edit distance between two given strings, along with an optimal edit transcript that describes the transformation.

# Dynamic programming



- For strings S1 and S2, D(i,j) is the edit distance between S1[1..i] and S2[1..j]. Let n=|S1| and m=|S2|.
- Dynamic programming:
  - Recurrence relation: recursive relationship between i and j in D(i,j)
  - Tabular computation: memoization technique for computing D(i,j)
  - Traceback: computing the optimal edit transcript from the table

# Recurrence relation



- Recursive relationship
  - Relate value of D(i,j) for i and j positive, and values of D with index pairs smaller than i, j.
  - Base conditions: D(i,0) = i and D(0,j) = j
- Recurrence relation for D(i,j) for i,j > 0
  - D(i,j) = min[D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+t(i,j)]
  - where t(i,j) is 1 if S1(i) ≠ S2(j) and 0 if S1(i)=S2(j)
- Complexity issue
  - The number of recursive calls grows exponentially with n and m
  - But, there are only (n+1) * (m+1) combinations of i and j, hence only (n+1) * (m+1) *distinct* recursive calls

# Bottom-up tabular computation



- (n+1) * (m+1) table
- Base: compute D(i,j) for the smallest possible values of i and j
- Induction: compute D(i,j) for increasing values of i and j, one row at the time

| D(i,j) | | | w | r | i | t | e | r | s |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| v | 1 | 1 | 1 | 2 | 3 | * | | | |
| i | 2 | 2 | | | | | | | |
| n | 3 | 3 | | | | | | | |
| t | 4 | 4 | | | | | | | |
| n | 5 | 5 | | | | | | | |
| e | 6 | 6 | | | | | | | |
| r | 7 | 7 | | | | | | | |

D(1,1) = min[D(0,1)+1, D(1,0)+1, D(0,0)+t(1,1)]
= min[2,2,0+1] = 1

D(1,2) = min[D(0,2)+1, D(1,1)+1, D(1,1)+t(1,2)]
= min[3,2,1+1] = 2

D(1,3) = min[D(0,3)+1, D(1,2)+1, D(0,2)+t(1,3)]
= min[4,3,2+1] = 3

Base: D(i,0) = i, D(0,j) = j

Step: D(i,j) = min[D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+t(i,j)], t(i,j) is 1 if S1(i) ≠ S2(j) and 0 if S1(i)=S2(j)

# Traceback



- Pointer-based approach:
  - When computing (i,j), set a pointer to the cell yielding the minimum
  - If (i,j) = D(i,j-1)+1 set a pointer from (i,j) to (i,j-1): ←
  - If (i,j) = D(i-1,j)+1 set a pointer from (i,j) to (i-1,j): ↑
  - If (i,j) = D(i-1,j-1)+t(i,j) set a pointer from (i,j) to (i-1,j-1): ↖
  - There may be several pointers if several predecessors yield the same minimum value
- To retrieve the optimal edit transcripts
  - Trace back the path(s) from (n,m) to (0,0)
  - A horizontal edge (←) represents an *insertion*
  - A vertical edge (↑) represents a *deletion*
  - A diagonal edge (↖) represents a *match* if S1(i)=S2(j), and a *substitution* if S1(i)≠S2(j)

# Time-complexity



- Filling the table costs *O(nm)* time
  - To fill one cell takes a constant number of cell examinations, arithmetic operations, and comparisons
  - The table consists of n by m cells, hence O(nm) time
- Retrieving the optimal path(s) costs *O(n+m)* time

# Conclusions



- Exact matching problem
  - Naive method compares character by character, single shift of P against T
  - Optimization through smarter shifting; base information for smarter shifting is provided by Z-boxes, computable in linear time
  - Boyer-Moore algorithm can run in *sublinear* time; thanks to two complementary rules: the bad character rule, and the good suffix rule
- Inexact matching problem
  - Looking for subsequences rather than substrings
  - Dynamic programming approach to establishing edit distance between two strings, specified as an edit transcript