Representations of strings

- Strings - linear sequences of symbols drawn from a character set, Σ
  - 1) Σ can be the set of alphanumeric characters, punctuations and spaces, and strings are text files
  - 2) Σ can be the quantized voltage levels from an A/D converter, and strings could be digital representation of voice waveforms
- Two basic problems in string representations:
  - 1) How to use the minimal number of bits to represent a string
  - 2) How to efficiently match strings

String encodings

- The bit sequence representing a symbol from a character set is called its encoding
  - 1) ASCII is a common encoding for alphanumeric strings
  - 2) For numerical strings, the encoding is just the integers
- General problem: Given a string w (called the text) over Σ, store it using as few bits as possible.
  - 1) Lossless techniques allow for exact reconstruction of string from encoding
  - 2) Lossy techniques allow for only aproximate reconstruction of string from its encoding - vector quantization

Huffman encoding

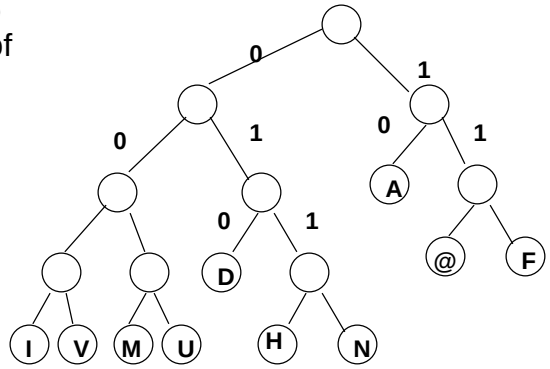
- Can use a straightforward encoding of symbols
  1) suppose size of Σ is n
  2) then represent each symbol in Σ using a b = roof[logn] bit string
  3) Then a string of length n will require bn bits to encode
- Suppose that we use bit sequences of different lengths to represent different symbols
  1) symbols that appear frequently will be represented using short sequences
  2) sequences that appear rarely will be encoded using long sequences

Huffman encoding

- Sequences have to be defined so that they cannot be confused!
- Bad example:
  1) E = 101
  2) T = 110
  3) Q = 101110
- How could we distinguish between the encoding of ET and Q?
- Solution: No bit sequence used to encode a symbol may be the prefix of the encoding of another symbol
  – so if the next k bits of the input matches the encoding for a given symbol, then that symbol must be the next symbol

Huffman encodings

- Can use binary tries to provide a description of such encodings
  - short paths leading to frequent characters
- Such tries are called **encoding trees**
  - 0 for left branches
  - 1 for right branches
- We will assume all encoding tries are full



Strings - 5

Huffman encodings

- Encoding a string, w, is easy - just concatenate the bit sequence that encodes each symbol.
  - I FIV = 0000001011100000001
- Decoding is also easy

  1) start at the root and move down the trie using the bits from w', the encoding of w, to determine path

  2) When a leaf is reached, output the character and return to the root.
- Big question - how should the tree be chosen to provide the best encoding?

Strings - 6

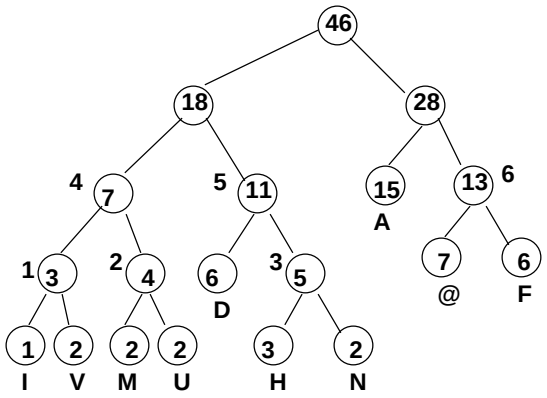
Huffman encodings

- Suppose we want to develop the best encoding for a fixed string w
  - w might be the string corresponding to the text of the Encyclopedia Britannica
- Let f_i be the number of times that symbol c_i occurs in w
  - f_i is related to the probability that a random symbol chosen from w will be c_i.
- We start by assigning one node to each symbol in Σ. Each of these nodes will be a leaf of the encoding tree, T.
  - each node has a field to store its weight. Initially, the weight of the node for c_i is f_i.

The magic of Huffman encodings

- Repeatedly perform the following step:
  - Pick two nodes, n_1 and n_2, having smallest weights (it doesn't matter how ties are broken)
  - Make a new node corresponding to the father of n_1 and n_2. The weight of this new node is the sum of the weights of n_1 and n_2.
  - Nodes n_1 and n_2 are marked as used, and will not participate in further mergings - so we reduce the number of active nodes at each step by 1.
  - Eventually only a single node is left - it is the root of the encoding tree.
- The tree constructed is called the Huffman encoding tree.

Constructing the Huffman tree



Optimality of the Huffman tree

- We can prove that no other encoding yields a smaller representation for w.
- For any tree, T, and any node n in T let:
  1) DT(n) = the depth of n in T
  2) L(T) be the set of leaves of T
  3) C(n) be the cost associated with each node in L(T)
- The weighted path length is defined as:

  $WPL(T) = \sum_{n \in L(T)} DT(n) * C(n)$

For our example tree WPL(T) = 1*4 + 2*4 +2*4 +6*3 +3*4 +2*4 +15*2 + 7*3 + 6*3 = 135

Optimality of Huffman coding

- If these weights are set to the frequencies of the symbols in w, then WPL(T) is the number of bits needed to encode w using the tree T.
- Lemma: Let T be any full binary tree with weights assigned to its leaves. Suppose n_1 and n_2 are any two leaves of T with smallest weight. Then we can construct a new tree, T', from T such that:

  1) the leaves of T' are the same as the leaves of T, except n_1 and n_2 are not leaves of T' and T' has a new leaf n_3.

  2) the weight of n_3 in T' is $C(n_3) = C(n_1) + C(n_2)$ and the weights of all other leaves in T' are the same as their weights in T.

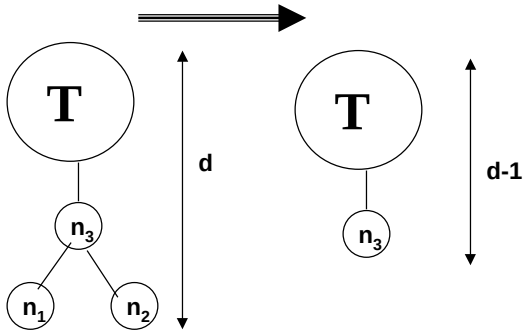
  3) WPL(T') Š WPL(T) - C(n3) with equality if n_1 and n_2 are siblings in T.

Strings - 11

Optimality of Huffman encoding

- First assume that n_1 and n_2 are siblings. What is the WPL of T'?
  - let d be the depths of n1 and n_2
  - then depth of n_3 must be d-1
  - deleting n_1 and n_2 reduces WPL(T) by $d \bullet (C(n_1) + C(n_2))$
  - adding n_3 increases WPL(T) by $(d-1) \bullet C(n_3)$

    $= (d-1) \bullet (C(n_1) + C(n_2))$
  - the difference is exactly - $C(n_3)$ as required

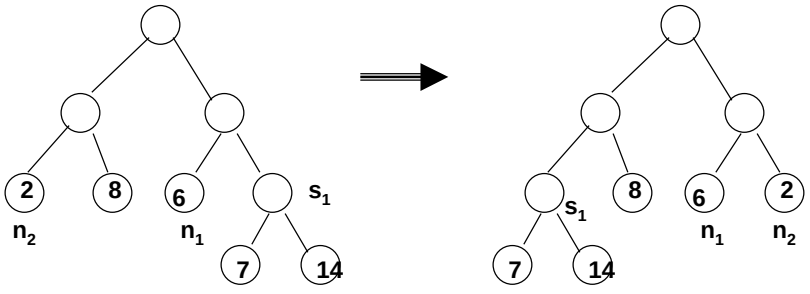


Strings - 12

Optimality of Huffman encoding

- Suppose that n_1 and n_2 are not siblings, but let s_1 be the sibling of n_1
- If the depth of n_1 and n_2 are the same, we just exchange n_2 and s_1
- This has no effect on WPL(T) because the leaves are still at the same depths
- But n_1 and n_2 are now siblings, so we are back to the previous transparancy.

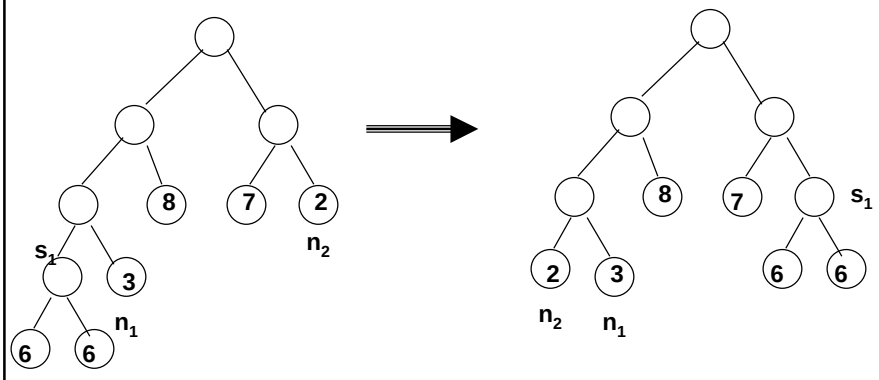
Optimality of Huffman encoding



Optimality of Huffman encodings

- Final case is when n_1 is deeper in the tree than n_2 (symmetric case handled the same)
  - Again exchange n_2 with s_1
- Moving n_2 down to the depth of n_1 increases WPL(T) by $C(n_2)\bullet[D(n_1)-D(n_2)]$
- But moving s_1 up in the tree means that ll of the leaves of s_1 move up by this same amount - $D(n_1)-D(n_2)$.
  - But each of these leaves has weight greater than the weight of n_2 - since n_2 was one of the two leaves in the original tree of minimal weight
- So, this operation either decreases WPL(T) or leaves it unchanged.
- But then we are back two transparancies, and we can merge n_1 and n_2 and satisfy the lemma.

Optimality of Huffman encodings



WPL(T) =2 • 6 • 4 + 3 • 3 + 8 • 2 + 7 • 2 + 2 • 2 = 91

WPL(T) = 7 • 2 + 2 • 6 • 3 + 8•2 + 3 • 3 + 2 • 3= 81

Huffman optimality theorem

- Theorem: Let N be a set of nodes and let each n ε N be assigned a weight C(n). Let T be the encoding tree constructed from the nodes in N by the Huffman algorithm. If X is any other encoding tree with the same leaves, then WPL(T) Š WPL(X)

Proof:

1) The proof is by induction on the number of leaves of T.

2) If T has two leaves, then the result is trivial because there is only one encoding tree.

Huffman optimality theorem

- 3) Otherwise let n_1 and n_2 be the first two members of N that are selected by the Huffman algorithm.
  - Apply the lemma to T and X producing T' and X'
  - n_1 and n_2 must be siblings in T, since T is constructed using the Huffman algorithm. Therefore WPL(T') = WPL(T) - C(n_1) - C(n_2)
  - The Lemma also guarantees that WPL(X') Š WPL(X) - C(n_1) - C(n_2)
  - But we also know that WPL(T') Š WPL(X') because they are trees with one node less than T and X (use induction hypothesis) and T' is the tree constructed from the leaves of T using the Huffman algorithm

W(T) = W(T') + c

W(X') +c Š W(X)

W(T') Š W(X')

W(T) = W(T') + c Š W(X') + c Š W(X)

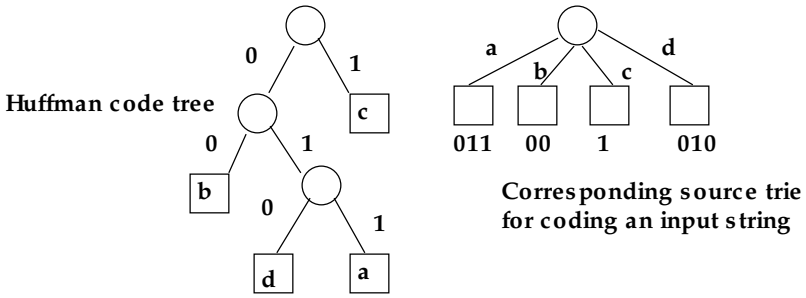
Extensions

- Biggest problem with Huffman algorithm is that all character frequencies must be known in advance
  - must read w twice - once to build T and second time to encode it
  - not always feasible to do this
- Static Huffman encoding - fix a single encoding tree, and use it for many texts
  - Many English texts may have similar symbol frequencies

Extensions

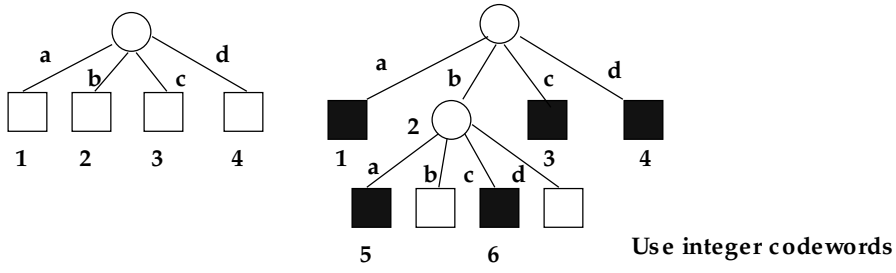
- Adaptive Huffman encoding
  - start with the empty encoding tree
  - after each character of w is processed, update T so that it is the Huffman tree for that part of w seen so far
  - can be done fast, and we don't have to transmit the encoding tree because it can be reconstructed at the destination
  - fast algorithms exist for updating the Huffman tree
- Ziv-Lempel-Welch (ZLV)encoding - capture sequences that occur in the text

ZLW encoding algorithm



- Decoding is traversing paths in code trees
- Encoding is traversing paths in source tries
- Why restrict ourselves to depth one source tries?
  - ZLW uses variable depth source tries

ZLW encoding algorithm



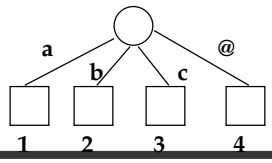
- String adbacbcd is partitioned into a | d | ba | c | bc | b | d
- Encoded text is 1453624
- Compare to 142132324 for the standard Huffman code tree

ZLW encoding

- Source trees do not satisfy the prefix property
- But we can use them as long as we employ a Greedy partitioning strategy
  - continue extending a root to frontier path as long as we can
  - that is why we treat ba as ba rather than a b followed by an a
- For ZLW both the source tokens and codewords are variable length
  - **tokenizing** - partitioning of the source text into tokens before encoding them
- In our examples the codes will be integers - but they can in turn be represented by variable length bit strings

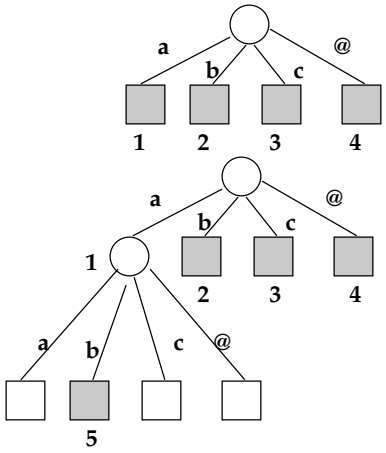
ZLW encoding

- With prefix codes we assume that encoder and decoder know the code **before** the source text is encoded
- ZLW - encoder and decoder know the source alphabet and its ordering as integers
  - Source trie is initialized by both encoder and decoder as a height one trie with one leaf for each alphabet symbol
- Example string - ababcbababaaaa@
  - source alphabet is {a,b,c,@}
- Trie will be grown by extending each **matched path** by one token

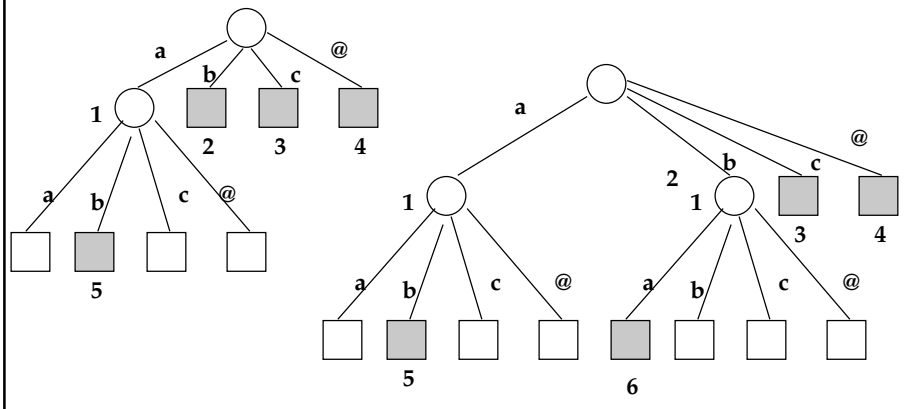


ZLW encoding - ababcbababaaaa@

- Applying the initial source trie to the input string, the first token found is an a.
  - transmit its codeword -1
  - add to the trie a **new token** (ab)corresponding to the current token (a) followed by the **next symbol** (b)
  - next integer codeword is always one greater than current largest integer codeword
- b is the next token
  - 2 is transmitted
  - ba is added to the source trie

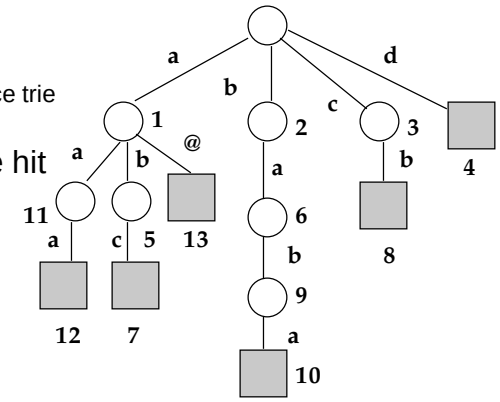


ZLW encoding - ababcbababaaaa@



ZLW encoding - ababcbababaaaa@

- Now, the next token is **ab**
  - codeword 5 is transmitted
  - token **abc** is added to the source trie with codeword 7
- Process continues until we hit the termination symbol
- Finally, the source text is tokenized as



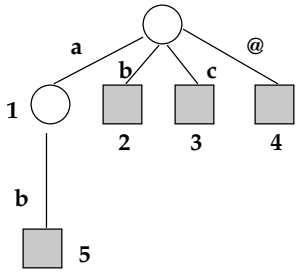
a | b | ab | c | ba | bab | a | aa | a | @
1 2 5 3 6 9 1 11 1 4

ZLW decoding algorithm

- Can the receiver build the source trie as codes are received and decode the text?
- Decoder begins with the same initial source trie defined over only the alphabet of individual symbols
- It then reconstructs the same sequence of source tries that the encoder produces.
- Assume that decoder has accomplished this for a nonempty prefix of the encoded text that ends with the code integer, i, and that the next integer received is j. Two cases:
  - j is already in the source trie
  - j is not already in the source trie

ZLW decoding algorithm

- Let the token corresponding to i be x, which by assumption **is** in the source trie
- Suppose j **is** in the source trie
  - Let its corresponding token be ay for some symbol a and string y.
  - Then the encoder would have processed **xay** and would have added the token xa to its source trie with the next integer as its codeword
  - So, decoder also adds xa to its source trie

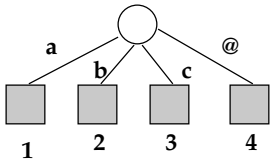


•source trie after encoding abab...
•decoder has seen 1 2 and has created same source trie
•encoder them emits 5
•decoder knows that ab is next matching token; this begins with a
•so, decoder adds ba to source trie - previous token followed by first symbol of new token

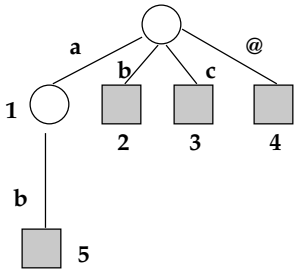
ZLW decoding algorithm

- But what if j is **not** in the source trie?
- This can only occur if j corresponds to the token dx'd where x = dx'.
  - j must begin and end with the same symbol as the previous token
- So, decoder adds the token dx'd to the source trie with integer codeword j.

Example of decoding



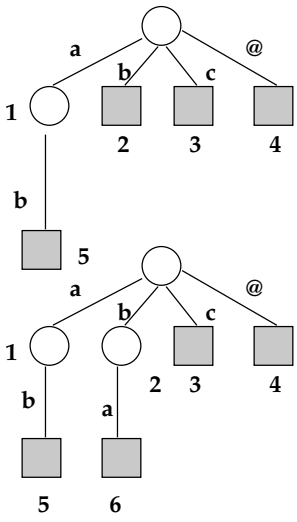
- Decoder's initial source trie
- Decoder receives "1" and emits "a" into output stream



- Decoder receives a "2" and emits "b"
- Decoder adds ab, previous token followed by first symbol of new token, to the source trie

Strings - 31

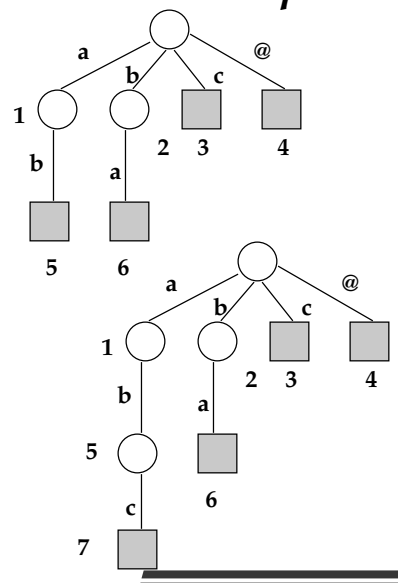
Example



- Decoder receives a "5"
- Decoder adds "ab" to the output stream
- Decoder adds "ba" to the source trie - last token (b) followed by first symbol of current token

Strings - 32

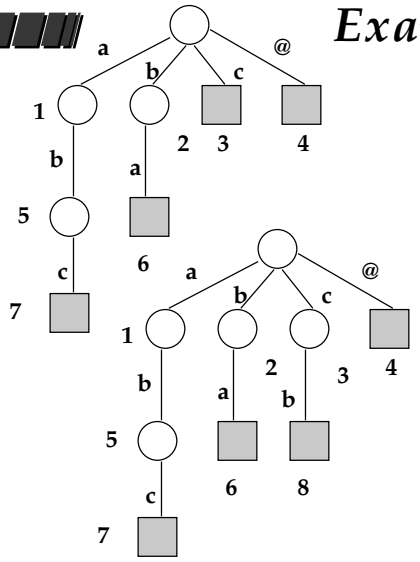
Example



- Decoder receives “3”
- Decoder emits “c” into the output stream
- Decoder adds abc to the source trie - previous token (ab) followed by first symbol (c) of current token

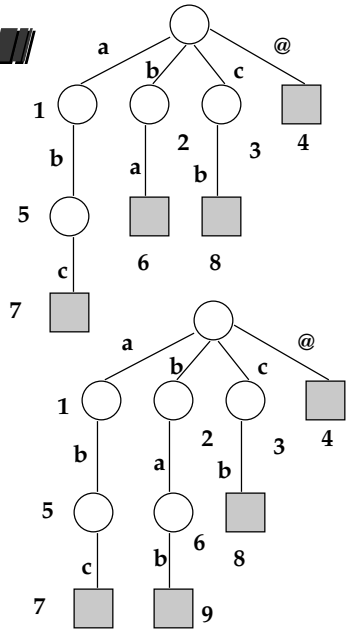
Strings - 33

Example



- Decoder receives “6”
- Decoder emits “ba” into output stream
- Decoder adds cb to the source trie - previous token (c) followed by first symbol (b) of current token.

Strings - 34



Example

- Decoder receives **"9"**!
- It is **not** in the source trie
- So, the encoder must have **just** put it there after transmitting **ba**
  - therefore, it certainly starts with ba
- Why must the next symbol in the string be b, yielding bab?
  - because if it weren't, then the next token could not take us down the path in the trie that begins with b - but that is where the encoder placed the new token!

Strings - 35

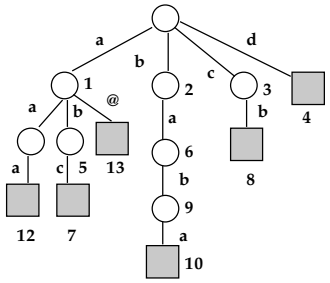
Implementation of the source trie at the decoder

- Don't need to use a tree structure to represent the trie at the decoder
- Sufficient to have a table indexed by codeword
- Each table entry could contain the token corresponding to that codeword.
- But then we would need to allocate large amounts of storage to each entry in the table, since the tokens can get long
- Instead, can store the last symbol and the codeword for the prefix of the token
  - this prefix must be in the table since we have a trie.

Strings - 36

Table representation of source trie

Index prefix	Last symbol	Index of
1	a	-
2	b	-
3	c	-
4	d	-
5	b	1
6	a	2
7	c	5
8	b	3
9	b	6
10	a	9



Strings - 37

String searching

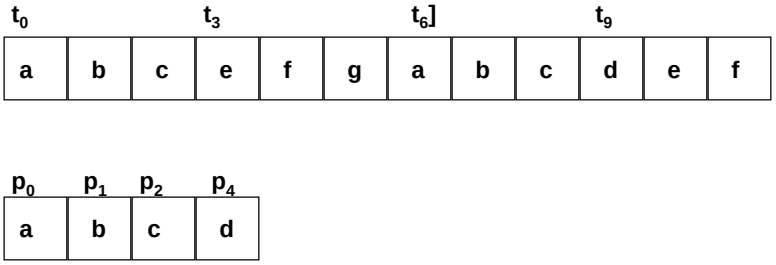
- General problem: Given two strings, p(pattern) and t(target) over the same alphabet, Σ, determine whether p is a substring of t.
- There is a very simple O(mn) algorithm to solve this problem:

```
function simplestringsearch(string p,t):integer
    for  k from 0 to length(t) - length(p) do
       i <-- 0
      while i < length (p) and p_i= t_{k+1}do
          i <-- i+1
       if i = length(p) then return k
    return -1
```

- We can do better - O(m + n)!

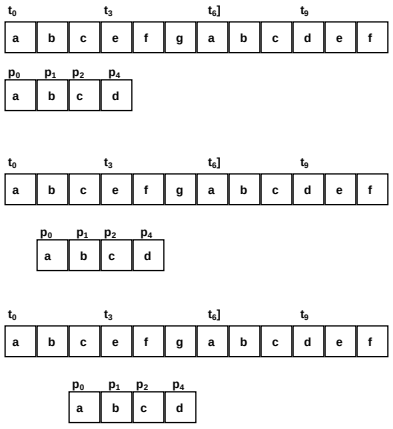
Strings - 38

String matching by box translation



Strings - 39

Example



Strings - 40

How bad is the naive algorithm?

- Actually, not bad at all!
- Let A be the symbol alphabet with c>1 symbols.
- Let p_i be the probability that symbol a_i occurs in the input string
- Assume that symbols in text are generated randomly using these probabilities
  - ignores digram frequencies, but good enough
- So, when we compare a random symbol from the text to a random symbol from the pattern, the probability that they are equal to some a_i is

$$p_{eq} = \sum_{i=1}^{c} p_i^2$$

Strings - 41

How good is bad?

- If the probability distribution is uniform ($p_i = 1/c$), then $p_{eq} = 1/c$
- Can now compute probability that two random strings P and T of the same length are equal.
  - If the strings are both of length m, then this probability is $1/c^m$
- Using this we can compute the **expected number** of symbol comparisons that the naive algorithm makes.
- Algorithm compares P_{i+1} to T_{i+1} only when $P_{1...i} = T_{1...i}$
  - This happens with probability $1/c^i$.

Strings - 42

How good is bad?

- So, the expected number of comparisons is

$$1+\sum_{i=2}^{m}\frac{1}{c^{i-1}}=\frac{c}{c-1}(1-\frac{1}{c^{m}})$$

- For large m this is approximately c/(c-1)
- For a P of length m and a T of length n the algorithm examines n-m+1 substrings of T of length m.
- Since these substrings are random the expected number of symbol comparisons is

$$\frac{c}{c-1}(1-\frac{1}{c^{m}})(n-m+1)$$

Strings - 43

Concrete examples

$$\frac{c}{c-1}(1-\frac{1}{c^{m}})(n-m+1)$$

- Binary alphabet
  - c = 2
  - approximately 2(n-m+1)
- Ascii
  - c is big, so c/(c-1) is approximated by 1
  - n-m+1 comparisons
- So, the naive algorithm is close to linear time, expected, anyway
- Nevertheless, the asymptotically faster algorithms are even faster in practice.

Strings - 44

Rabin-Karp algorithm

- Algorithm has worst case running time of O((n-m+1)m), but an average case running time of O(m+n)
- Algorithm uses some elementary ideas from number theory - the equivalence of two numbers modulo a third
- For simplicity assume that $\Sigma = \{0,1,2,..., 9\}$ so that each character is a decimal digit
  - more generally, if the size of Σ is d, then we will treat each character as a digit in radix d notation.
- We can think of a string of length k as a k-digit decimal number.

Rabin Karp algorithm

- Example - string of characters 354123 corresponds to the number 354,123
- Given a pattern P[1 .. m], let p be the decimal number corresponding to the m character pattern
- Given a text T[1 ..n], let t_s = T[s+1 .. s+m] for s = 0, 1, ... n-m denotes the decimal value of the m character substring starting at position s+1.
- Clearly, t_s = p if and only if P[1 ..m] = T[s+1 .. s+m]
  - So, if we could compute p in O(m) time and compute all of the t_s in O(n) time, then in O(n) time we can compare all of the shifts with p and find all matching locations.
- Of course, the numbers may get very large, but we'll solve that problem in 2-3 transparencies

P --> p

- p = P[m] + 10 x P[m-1] + 10^2 x P[m-2] + ...
  10^{m-1} x P[1]
- Computing this in a brute force way would require $O(n^2)$ operations to compute all of the powers of 10.
- But notice, that we can eliminate one multiplication by 10 in all terms (but the first), by factoring the 10 out:
- p = P[m] + 10 (P[m-1] + 10 x P[m-2] + ... +
  10^{m-2} x P[1])
- Applying this factoring rule repeatedly will lead to only a linear number, in m, of multiplications.

Rabin-Karp

- Computing p in O(m) time. We use what is called **Horner's rule**
- p = P[m] + 10 (P[m-1] +10 (P[m-2]) + ... + 10(P[2] + 10P[1])...)))
- Example: P = 3276; P[4] = 6, P[3] = 7, P[2] = 2, P[1] =3
- p = 6 + 10(7 +10(2 + 10(3))))
  = 6 + 10(7 + 10(32))
  = 6 + 10(327) = 6 + 3270 = 3276
- t_0 can be computed from T[1 .. m] using the same method

Rabin Karp

- Big question is then computing the remaining t_i in O(n-m) time - that is with just a constant number of operations for each remaining t_i.
- But, there is a simple relationship between t_{s+1} and t_s:

  $t_{s+1} = 10(t_s - 10^{(m-1)} T[s+1]) + T[s+m+1]$
- Example: Suppose T = 314152 and the length of the pattern is 5. Then t_0 = 31, 415. What is t_1?

  $t_1 = 10(31,415 - 10^4(3)) + 2$

  $= 10(1,415) + 2 = 14,152$

Rabin Karp

- We precompute the constant $10^{(m-1)}$ (which can be done in logm time) so each update of a t takes a constant number of operations.
- So, p and all of the t_i can be computed in O(m+n) time.

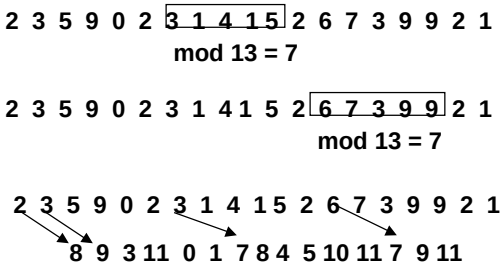
Rabin Karp

- So - what is the problem? Well, if P is a long pattern, or if the size of the alphabet is large, then the p and t_i may be **very large numbers**, and each arithmetic operation will not take constant time - the numbers might require many words of memory to represent
- Solution - compute the p and t's modulo a suitable modulus q.
  - Horner's rule and the simple updating rule will work using mod q arithmetic
  - q is chosen such that 10q just fits within one computer word. This allows all of the arithmetic operations to be performed using single precision arithmetic, while keeping q as large as possible
  - for an alphabet with d symbols, q is chosen so that dq fits within one word.

Strings - 51

What's the catch?

- The catch is that $p = t_s$ is no longer a sufficient condition for deciding that P matches a substring of T.



  - But these are the only two positions that our algorithm would have to check by a detailed comparison of P and T

Strings - 52

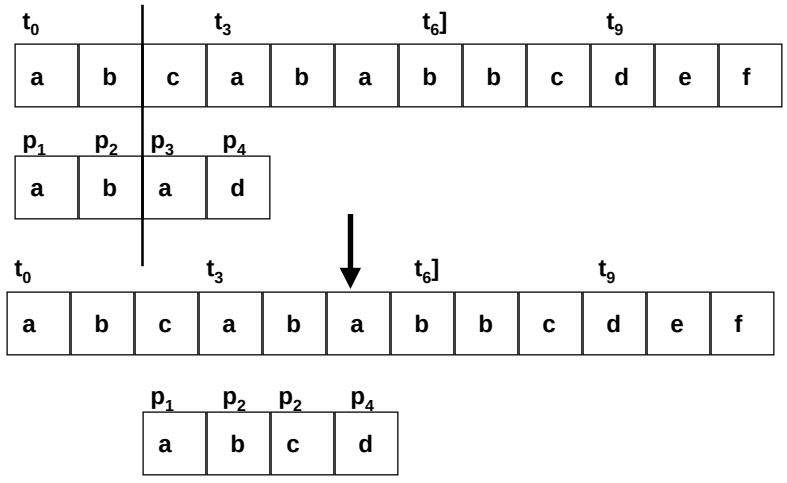
What is a worst case?

- Let P be a string of length m of all a's
- Let T be a string of length n of all a's
- Then all substrings of length m of T will have the same "key" as P, and all will have to be checked.
- But we know that if a substring of T matches P, to determine if the "next" substring matches we only have to perform one additional comparison.
- So, the Rabin Karp algorithm can do a lot of unnecessary work.
- In practice, though, it doesn't.

Knuth-Morris-Pratt

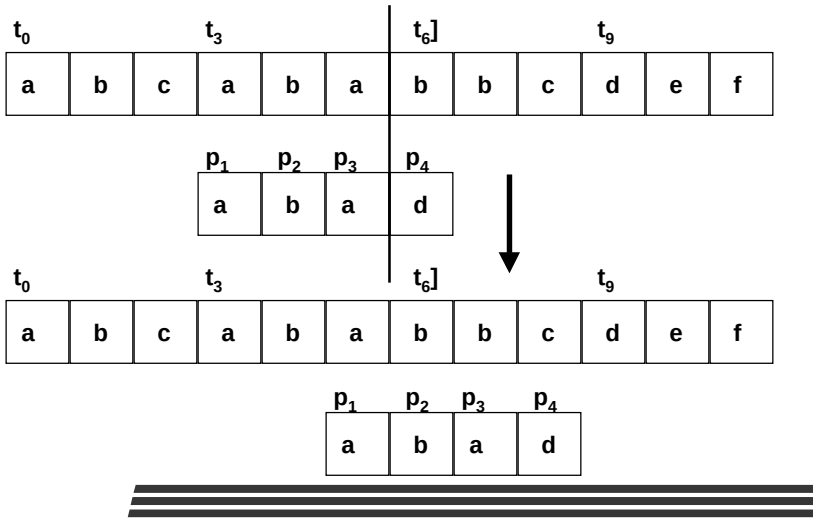
- What does the brute force algorithm do when a mismatch occurs when matching the i'th character of the pattern at the j'th position of the target?
  - it slides the pattern one position down the target string
  - but this ignores the fact that we KNOW that the last i-1 positions of the target match the first i-1 positions of the pattern
  - So, maybe we can move the pattern further along the target than just one position when the match fails

Examples



Strings - 55

Examples



Strings - 56

A concrete example

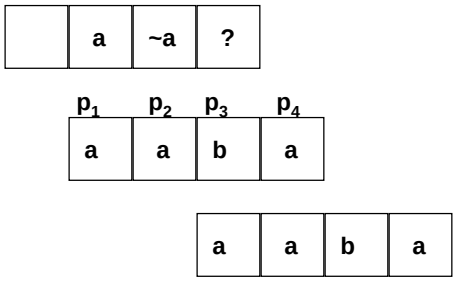
- Consider the string aaba
- Its pattern matching algorithms is:

```
i:=1
while T[i+4] <> @ do
  if T[i] = a
   then if T[i+1] = a
    then if T[i+2] = b
     then if T[i+3] = a
      then report (i);
i := i + 1
```

- Algorithm always increments i by 1 if a mismatch is found - it does not distinguish between the different mismatches

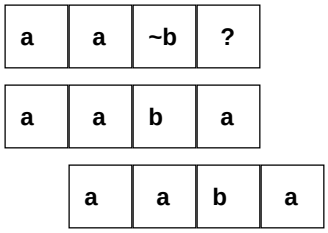
A concrete example - aaba

- But consider the case T[i+1] <> a
  - In this case we can increment i by 2 rather than 1
  - Incrementing i by 1 would (re)compare T[i+1] (old i) to P[1] = a
  - But we just discovered that T[i+1] <> P[2] = a = P[1]
  - So, we can move the pattern 2 positions rather than 1



Concrete example - aaba

- Suppose that T[i+2] <> b
  - T[i] = a and T[i+1] = a
  - T[i+2] **may** be a, so the pattern may occur at position i+1.
  - However, next comparison should be between P[2] and T[i+2], not between P[1] and T[i+1]
  - We **already** know that P[1] = P[2] and that T[i+1] = P[2].



General case

- There are two cases that might result from a mismatch
  - continue by comparing P[1] with the next text symbol - that is, slide the pattern beyond the current textual position (first concrete example).
  - continue by comparing the current textual symbol with some **earlier** pattern symbol; we do not slide the pattern beyond the current textual position (second concrete example)
- When a mismatch occurs at pattern position j, the algorithm continues matching at some position k in the pattern, with k < j - that is, it compares the current symbol in the text against the k'th symbol in the pattern

General case

- The value of k is defined uniquely for each mismatch position j
  - Next: {1, ..., m} --> {0,..., m-1}
- When a mismatch occurs at position j
  - if next(j)= 0, then we match P[1] against T[i+1], where T[i] is the character at which the match failed
  - if next(j)<> 0, then we match T[i] against P[next(j)]
  - on both cases, we move the pattern j-next(j) positions

KMP algorithm

- Assume that the Next function for P has already been computed

```
i:= 1; j:= 1;
while T[i+m-j] <> @ do
begin
  if T[i] = P[j]
   then if j = m
    then begin report (i-m+1) {i has been moved m positions into the target}
                  j := next(m+1); i := i+1 {Pattern might have repeated substrings}
   end
   else begin i:= i+1; j := j+1 end {Move forward one position}
  else j := next(j); {failed, used next function}
  if j = 0 then begin i := i + 1; j := 1 end
end
```

Next function

- Generally, how do we know how many positions we can move the pattern along the target after a failure?
- We do this using **suffix** functions
- Let $next_j$ be the number of positions we move the pattern when its first j-1 characters match the target, but the (j)'th character does not
- Then, $next_j$ has the property that it is the largest k **less than** j such that $p_1p_2 ...p_{k-1}$ is a **suffix** of $p_1p_2...p_{j-1}$
  - i.e., $p_1...p_{k-1} = p_{i-k+1}...p_{j-1}$ but $p_j \neq p_k$
  - if there is no such k, then $next_j = 0$

Next functions

- Given that P[1 ...k] = T[s+1, ..., s+k] what is the **least shift** s' > s such that

  P[1, ..., j] = T[s' +1, ..., s' +j] where s'+j = s+k
- s' is the first shift greater than s that is **not necessarily invalid** given our knowledge of T[s+1, ..., s+k]
  - at this new shift we do not have to compare the first j characters of P against T since we are guaranteed that they match.
- This information can be precomputed from P.
- Let P_k denote the k prefix of P
- Now, T[s' +1, ..., s' +j] is a **suffix** of P_k
- So, for each k, we are looking for the largest j such that P_j is a suffix of P_k. Largest suffix gives the least shift

Example - abababacba

k	P_k	P_j	next(k)
1	a	-	0
2	ab	-	0
3	aba	a	1
4	abab	ab	2
5	ababa	aba	3
6	ababab	abab	4
7	abababa	ababa	5
8	abababab	ababab	6
9	ababababc	-	0
10	abababacba	a	1

String matching with finite automata

- Recall:

A finite automaton, M, is a 5-tuple (Q, q_0, A, Σ, F) where:

- Q is a finite set of states
- q_0 in Q is the start state
- A, a subset of Q, is a set of accepting states
- Σ is a finite input alphabet
- F is a function from Q xΣ into Q called the transition function of M

Final state function

- A finite automaton, M, has an associated final state function, ϕ, from Σ^* to Q such that $\phi(w)$ is the state that M ends in after scanning the string w.
- So, M accepts w if and only if $\phi(w)$ is in A.
- ϕ is defined by the recursive definition:

  $\phi(\varepsilon) = q_0$

  $\phi(wa) = F(\phi(w), a)$ for w in Σ^* and a in Σ

String matching automata

- Corresponding to every pattern P, there is a string matching automaton
- This automaton can be constructed in a preprocessing step before it is compared against the text
- The automaton is defined using a function, σ, called the suffix function corresponding to P.
- σ is a mapping from Σ^* to {0, 1, ..., m} such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x!
- $\sigma(x) = \max \{k : P_k \heartsuit x\}$

Suffix functions

- The suffix function is well defined, because $p_0 = \varepsilon$ is a suffix of every string (so k can take on the value 0)
- Example

  P = ab

  σ(ccaca) = 1; a is the longest prefix of P that is a suffix of x

  σ(ccab) = 2; ab is the longest prefix of P that is a suffix of x
- For a pattern of length m, $\sigma(x) = m$ if and only if P ♥ x

String matching automaton

- The string matching automaton for a pattern P[1 .. m] is defined as follows:

  1) The state set Q is {0, 1, ..., m}

  2) q_0 is the start state and q_m is the accepting state

  3) The transition function, F, is defined by the following for any state q and any character a:

  $F(q,a) = \sigma(P_q a)$

String matching automaton

- What does this mean?

  1) If we are in state q, then the last q characters scanned from T match the first q characters of P - that is, q is the length of the longest suffix of the scanned part of T that matches a prefix of P.

  2) If q+1 characters of T had matched p, then we would be in state q+1. So, q must be the length of the longest suffix of T that is a prefix of P.

  3) If a, the next character of T, matches the q+1'st character of P, then we want to move to state q+1, indicating that the matching process is still successful.

  4) If a does not match P[q+1], then we want to return to the state that requires us to do the minimal amount of extra comparisons, but still guarantess finding any match. This state will be a number < q.

Examples

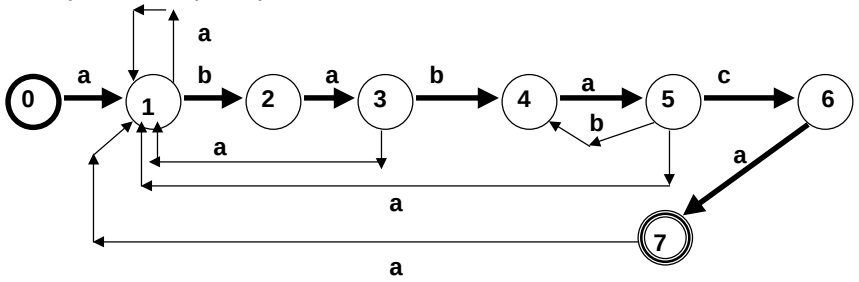
-) P = axaxyxa
  - T = abxaxcd**axaxa...**
  - So, we should match the next character of T against the 4'th character of P. This will be done by placing the automaton in state 3, because we **enter** a state, q, when we determine that the last q characters of T match the first q characters of P.
- P = aabbaabbc
  - T = ababaabb**aabba**
  - The automaton is in state 4, ready to read the last a
  - The a matches the 5'th character in P, so we go to state 5.

String matching automaton

- The following is always true:
  - After scanning the first i characters of T, $f(T_i) = q$ where $q = \sigma(T_i)$ is the length of the longest suffix of T_i that is a prefix of P.
- Suppose the next character T[i+1] is an a. Then the machine should transfer to state $\sigma(T_{i+1}) = \sigma(T_i a)$.
- But it is clear that $\sigma(T_i a) = \sigma(P_q a)$
  - This is because the suffix of length q of T is the prefix, P_q, of P.
  - So, computing the longest suffix of $T_i a$ that is a prefix of P is **equivalent** to computing the longest suffix of $P_q a$ that is a prefix of P.
  - Therefore, the machine's next state is determined completely by P and the next character from T.

Example automaton

- Some edges for failing matches are not shown (example - if you are in state 3 and you see a **c**, you go back to state 0; if you are in state 0 and see a **c**, you stay in state 0)
- Moving right is progress; moving left is price of failure - but you don't always fail completely.



P = a b a b a c a

i	1	2	3	4	5	6	7	8	9	10	11
T[i]	a	b	a	b	a	b	a	c	a	b	a
state f(Ti) 0	1	2	3	4	5	4	5	6	7	2	3

Computing the transition function

- Compute-transition-function (P, S)

```
m <-- length[P]
for q <-- 0 to m do
  for each character a in S do
   k <-- min(m+1, q+2)  {start with largest possible suffix}
     repeat
     k <-- k-1  { and decrease the length of the suffix}
     until P_k ♥ P_q a  {until it  is a prefix of P}
     F(q,a) <-- k
return F
```

- The running time of this algorithm is $O(m^3S)$

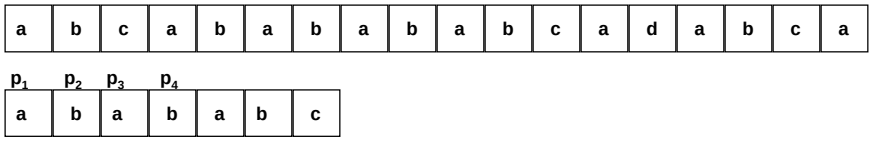
Strings - 75

You should be able to:

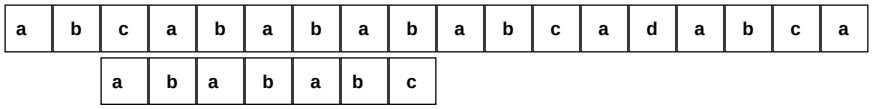
1) Compute the state transition function for a given pattern and alphabet S.
2) Draw the finite state automaton (leaving out all of the arrows that bring you back to the start state).
3) Simulate the activitity of the FSA on a given target string, up to the point where the first match is encountered, or a complete mismatch can be declared.

Strings - 76

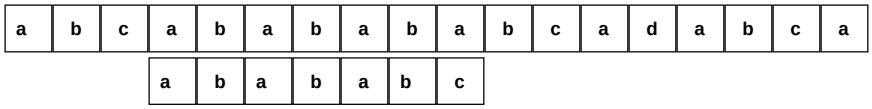
KMP next function



- next(3) = 0



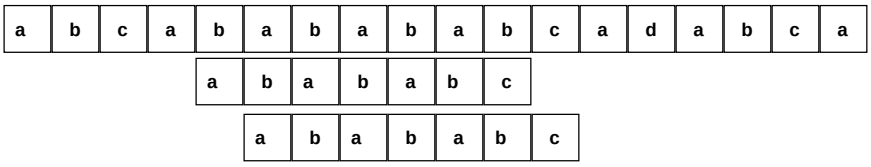
- next(1) = 0



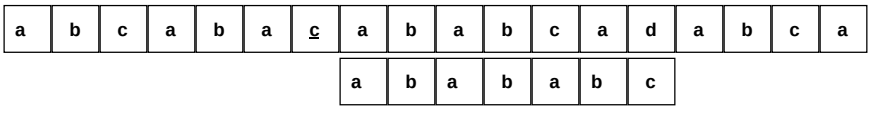
- next(7) = 5
- Pattern shifts by 2
- Suffix of T that matches prefix of P is of length 4 = 6-2

Strings - 77

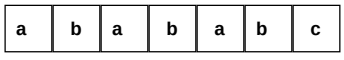
KMP next function



- next(8) = 0 - done

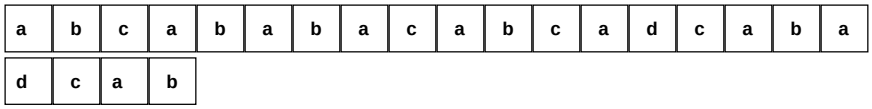


- next(5) = 3
- ab is the longest suffix of abab that is a prefix of abab

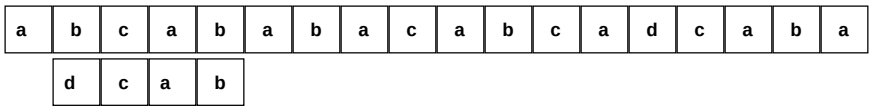


Strings - 78

Boyer Moore algorithm - scan backwards



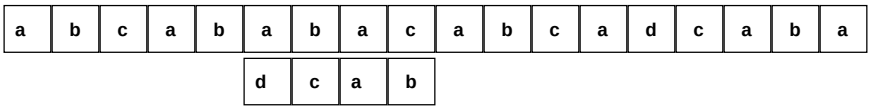
- Start at end of P
- Move towards front of P as long as characters match
- When you encounter a mismatch, slide the pattern to the right so that the rightmost instance in P_{m-1} of the character in T[i] is aligned with T[i].
- Movement function called symnext : Σ --> {0.,,, m-1,}
- Symnext(a) = 3



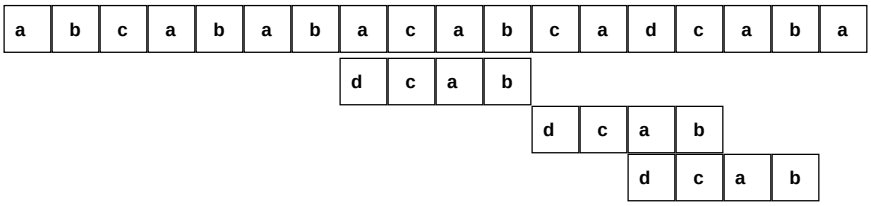
- symnext (b) = 0; special case - mover pattern completely past b

Strings - 79

Boyer-Moore algorithm



- symnext(c) = 2
- move P so that its rightmost c is under the c in T



Strings - 80

Boyer-Moore

- Fact: For a random pattern of length m, a random text of length n and an alphabet of size c > 1, where c <<n, the BM algorithm makes

  (n/m) + (n/2c)

  symbol comparisons in the expected case.
- This is **very** fast

Symnext example

- Suppose that P = abacdfeabdf

s	symnext(s)
a	8
b	9
c	4
d	10
e	7
f	6