Week 11: String Algorithms

Strings

Strings

2/67

A *string* is a sequence of characters.

An *alphabet* Σ is the set of possible characters in strings.

Examples of strings:

- C program
- HTML document
- DNA sequence
- Digitised image

Examples of alphabets:

- ASCII
- Unicode
- {0,1}
- {A,C,G,T}

... Strings

3/67

Notation:

- *length(P)* … #characters in *P*
- λ … *empty* string   ($length(\lambda) = 0$)
- Σ^m … set of all strings of length *m* over alphabet Σ
- Σ^* … set of all strings over alphabet Σ

$\nu\omega$ denotes the *concatenation* of strings ν and ω

Note: length(νω) = length(ν)+length(ω) $\lambda\omega = \omega = \omega\lambda$

... Strings

4/67

Notation:

- *substring* of *P* … any string *Q* such that $P = \nu Q\omega$, for some $\nu,\omega \in \Sigma^*$
- *prefix* of *P* … any string *Q* such that $P = Q\omega$, for some $\omega \in \Sigma^*$
- *suffix* of *P* … any string *Q* such that $P = \omega Q$, for some $\omega \in \Sigma^*$

Exercise #1: Strings

5/67

The string `a/a` of length 3 over the ASCII alphabet has

- how many prefixes?
- how many suffixes?
- how many substrings?

- 4 prefixes: `""` `"a"` `"a/"` `"a/a"`
- 4 suffixes: `"a/a"` `"/a"` `"a"` `""`

- 6 substrings: `""` `"a"` `"/"` `"a/"` `"/a"` `"a/a"`

Note:
`""` means the same as λ (= empty string)

... Strings

ASCII (American Standard Code for Information Interchange)

- Specifies mapping of 128 characters to integers 0..127
- The characters encoded include:
  - upper and lower case English letters: A-Z and a-z
  - digits: 0-9
  - common punctuation symbols
  - special non-printing characters: e.g. *newline* and *space*

Ascii	Char	Ascii	Char	Ascii	Char	Ascii	Char
0	Null	32	Space	64	@	96	`
1	Start of heading	33	!	65	A	97	a
2	Start of text	34	"	66	B	98	b
3	End of text	35	#	67	C	99	c
4	End of transmit	36	$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledge	38	&	70	F	102	f
7	Audible bell	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Horizontal tab	41	)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift in	46	.	78	N	110	n
15	Shift out	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Neg. acknowledge	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End trans. block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitution	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	File separator	60	<	92	\	124	|
29	Group separator	61	=	93	]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	Forward del.

... Strings

Reminder:

In C a string is an array of `char`s containing ASCII codes

- these arrays have an extra element containing a 0
- the extra 0 can also be written `'\0'` (*null character* or *null-terminator*)
- convenient because don't have to track the length of the string

Because strings are so common, C provides convenient syntax:

```
char str[] = "hello";  // same as char str[] = {'h','e','l','l','o','\0'};
```

Note: `str[]` will have 6 elements

... Strings

C provides a number of string manipulation functions via `#include <string.h>`, e.g.

```
strlen()   // length of string
strncpy()  // copy one string to another
```

```
strncat()  // concatenate two strings
strstr()   // find substring inside string
```

Example:

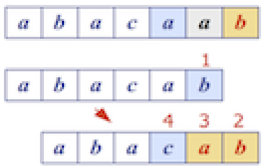
```
char *strncat(char *dest, char *src, int n)
```

- appends string `src` to the end of `dest` overwriting the `'\0'` at the end of `dest` and adds terminating `'\0'`
- returns start of string `dest`
- will never add more than `n` characters
  (If `src` is less than `n` characters long, the remainder of `dest` is filled with `'\0'` characters. Otherwise, `dest` is not null-terminated.)

Pattern Matching

Pattern Matching

Example (pattern checked *backwards*):



- *Text* … `abacaab`
- *Pattern* … `abacab`

... Pattern Matching

Given two strings T (text) and P (pattern),
the *pattern matching problem* consists of finding a substring of T equal to P

Applications:

- Text editors
- Search engines
- Biological research

... Pattern Matching

Naive pattern matching algorithm

- checks for each possible shift of P relative to T
  - until a match is found, or
  - all placements of the pattern have been tried

```
NaiveMatching(T,P):
|  Input  text T of length n, pattern P of length m
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  for all i=0..n-m do
|  |  j=0                         // check from left to right
|  |  while j<m ∧ T[i+j]=P[j] do  // test ith shift of pattern
|  |  |  j=j+1
|  |  |  if j=m then
|  |  |     return i             // entire pattern checked
|  |  |  end if
```

```
|   |  end while
|  end for
|  return -1                          // no match found
```

Analysis of Naive Pattern Matching

Naive pattern matching runs in O(n·m)

Examples of worst case (forward checking):

- *T* = `aaa…ah`
- *P* = `aaah`
- may occur in DNA sequences
- unlikely in English text

Exercise #2: Naive Matching

Suppose all characters in *P* are different.

Can you accelerate `NaiveMatching` to run in *O(n)* on an *n*-character text *T*?

When a mismatch occurs between *P[j]* and *T[i+j]*, shift the pattern all the way to align *P[0]* with *T[i+j]*

⇒ each character in *T* checked at most twice

Example:

```
abcdabcdeabcc   abcdabcdeabcc
abcdexxxxxxxx   xxxxabcde
```

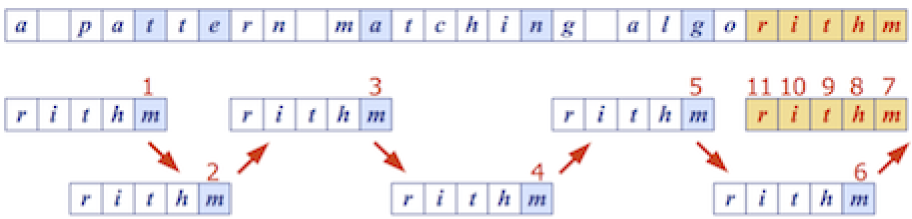
Boyer-Moore Algorithm

The *Boyer-Moore* pattern matching algorithm is based on two heuristics:

- *Looking-glass heuristic*: Compare *P* with subsequence of *T* moving *backwards*
- *Character-jump heuristic*: When a mismatch occurs at *T[i]*=`c`
  - if *P* contains `c` ⇒ shift *P* so as to align the last occurrence of `c` in *P* with *T[i]*
  - otherwise ⇒ shift *P* so as to align *P[0]* with *T[i+1]*   (a.k.a. "big jump")

... Boyer-Moore Algorithm

Example:



... Boyer-Moore Algorithm

Boyer-Moore algorithm preprocesses pattern *P* and alphabet Σ to build

- *last-occurrence function L*
  - *L* maps Σ to integers such that *L(c)* is defined as
    - the largest index *i* such that *P[i]=c*, or
    - -1 if no such index exists

Example: Σ = {a,b,c,d}, *P* = acab

c	a	b	c	d
L(c)	2	3	1	-1

- *L* can be represented by an array indexed by the numeric codes of the characters
- *L* can be computed in *O(m+s)* time (*m* … length of pattern, *s* … size of Σ)

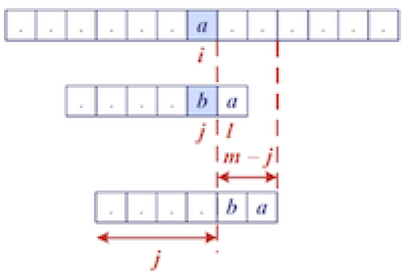
... Boyer-Moore Algorithm


```
BoyerMooreMatch(T,P,Σ):
|  Input  text T of length n, pattern P of length m, alphabet Σ
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  L=lastOccurenceFunction(P,Σ)
|  i=m-1, j=m-1                  // start at end of pattern
|  repeat
|  |  if T[i]=P[j] then
|  |  |  if j=0 then
|  |  |     return i             // match found at i
|  |  |  else
|  |  |     i=i-1, j=j-1
|  |  |  end if
|  |  else                       // character-jump
|  |  |  i=i+m-min(j,1+L[T[i]])
|  |  |  j=m-1
|  |  end if
|  until i≥n
|  return -1                     // no match
```

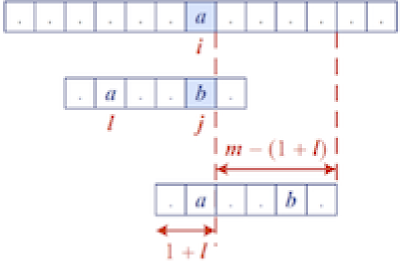
- Biggest jump (m characters ahead) occurs when *L[T[i]]* = -1

... Boyer-Moore Algorithm

Case 1: $j \le 1+L[c]$



Case 2: $1+L[c] < j$

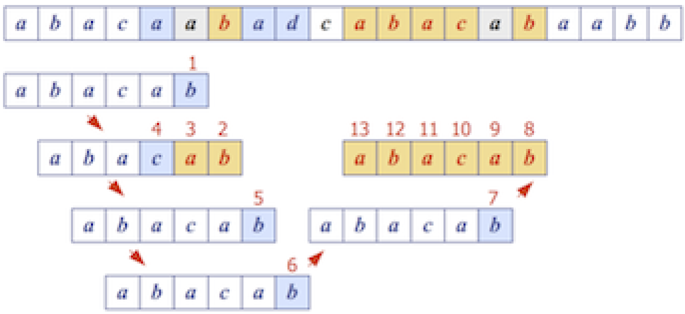


Exercise #3: Boyer-Moore algorithm

For the alphabet Σ = {`a`,`b`,`c`,`d`}

1. compute last-occurrence function L for pattern P = **`abacab`**
2. trace Boyer-More on P and text T = `abacaabadcabacabaabb`
   - how many comparisons are needed?

c	a	b	c	d
L(c)	4	5	3	-1



13 comparisons in total

... Boyer-Moore Algorithm

Analysis of Boyer-Moore algorithm:

- Runs in *O(nm+s)* time
  - *m* … length of pattern   *n* … length of text   *s* … size of alphabet
- Example of worst case:
  - T = `aaa … a`
  - P = `baaa`
- Worst case may occur in images and DNA sequences but unlikely in English texts
  ⇒ Boyer-Moore significantly faster than naive matching on English text

Knuth-Morris-Pratt Algorithm

The *Knuth-Morris-Pratt* algorithm …

- compares the pattern to the text *left-to-right*
- but shifts the pattern more intelligently than the naive algorithm

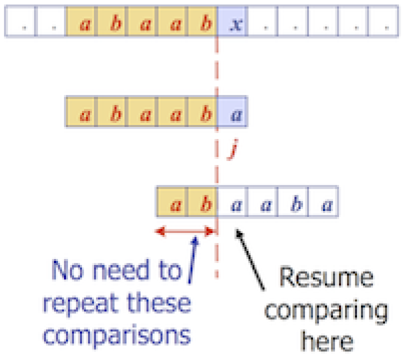
Reminder:

- Q is a *prefix* of P … $P = Q\omega$, for some $\omega \in \Sigma^*$
- Q is a *suffix* of P … $P = \omega Q$, for some $\omega \in \Sigma^*$

... Knuth-Morris-Pratt Algorithm

When a mismatch occurs …

- what is the most we can shift the pattern to avoid redundant comparisons?
- Answer: the largest *prefix* of *P[0..j]* that is a *suffix* of *P[1..j]*



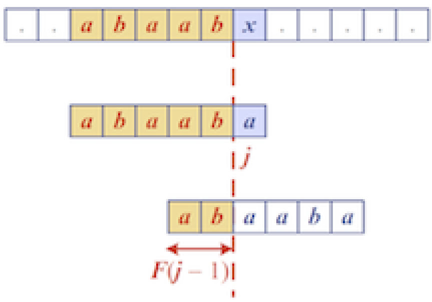
... Knuth-Morris-Pratt Algorithm

KMP preprocesses the pattern to find matches of its prefixes with itself

- *Failure function F(j)* defined as
  - the size of the *largest prefix* of *P[0..j]* that is also a *suffix* of *P[1..j]*
- if mismatch occurs at P_j ⇒ advance *j* to *F(j-1)*

Example: *P* = `abaaba`

j	0	1	2	3	4	5
P_j	a	b	a	a	b	a
F(j)	0	0	1	1	2	3



... Knuth-Morris-Pratt Algorithm


```
KMPMatch(T,P):
|  Input  text T of length n, pattern P of length m
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  F=failureFunction(P)
|  i=0, j=0                    // start from left
|  while i<n do
```

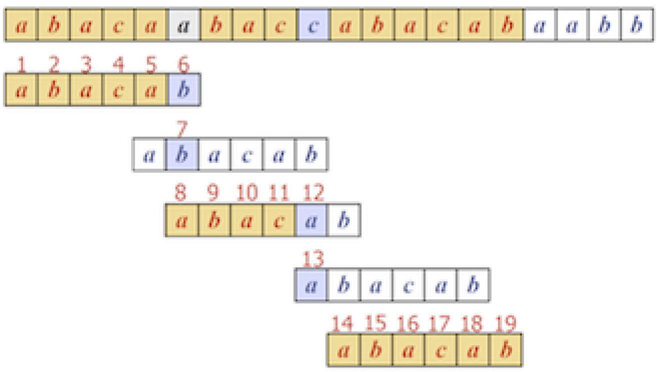
```
|   |  if T[i]=P[j] then
|   |  |  if j=m-1 then
|   |  |     return i-j          // match found at i-j
|   |  |  else
|   |  |     i=i+1, j=j+1
|   |  |  end if
|   |  else                      // mismatch at P[j]
|   |  |  if j>0 then
|   |  |     j=F[j-1]            // resume comparing P at F[j-1]
|   |  |  else
|   |  |     i=i+1
|   |  |  end if
|   |  end if
|   end while
|   return -1                    // no match
```

Exercise #4: KMP-Algorithm

1. compute failure function F for pattern P = **abacab**
2. trace Knuth-Morris-Pratt on P and text T = abacaabaccabacabaabb
   - how many comparisons are needed?

---

j	0	1	2	3	4	5
P_j	a	b	a	c	a	b
$F(j)$	0	0	1	0	1	2



19 comparisons in total

... Knuth-Morris-Pratt Algorithm

Construction of the failure function is similar to the KMP algorithm itself:

```
failureFunction(P):
|  Input  pattern P of length m
|  Output failure function for P
|
|  F[0]=0
|  i=1, j=0
|  while i<m do
|  |  if P[i]=P[j] then    // we have matched j+1 characters
|  |     F[i]=j+1
```

```
|   |       i=i+1, j=j+1
|   |    else if j>0 then     // use failure function to shift P
|   |       j=F[j-1]
|   |    else
|   |       F[i]=0            // no match
|   |       i=i+1
|   |    end if
|   end while
|   return F
```

... Knuth-Morris-Pratt Algorithm

Analysis of failure function computation:

- At each iteration of the while-loop, either
  - *i* increases by one, or
  - the "shift amount" *i-j* increases by at least one (observe that F(j-1)<j)
- Hence, there are no more than $2 \cdot m$ iterations of the while-loop

⇒ failure function can be computed in *O(m)* time

... Knuth-Morris-Pratt Algorithm

Analysis of Knuth-Morris-Pratt algorithm:

- Failure function can be computed in *O(m)* time
- At each iteration of the while-loop, either
  - *i* increases by one, or
  - the "shift amount" *i-j* increases by at least one (observe that F(j-1)<j)
- Hence, there are no more than $2 \cdot n$ iterations of the while-loop

⇒ KMP's algorithm runs in *optimal time O(m+n)*

Boyer-Moore vs KMP

Boyer-Moore algorithm

- decides how far to jump ahead based on the mismatched character in the text
- works best on large alphabets and natural language texts (e.g. English)

Knuth-Morris-Pratt algorithm

- uses information embodied in the pattern to determine where the next match could begin
- works best on small alphabets (e.g. `A,C,G,T`)

For the keen: The article "Average running time of the Boyer-Moore-Horspool algorithm" shows that the time is inversely proportional to size of alphabet

---

Word Matching With Tries

Preprocessing Strings

Preprocessing the *pattern* speeds up pattern matching queries

- After preprocessing *P*, KMP algorithm performs pattern matching in time proportional to the text length

If the text is large, immutable and searched for often (e.g., works by Shakespeare)

- we can preprocess the *text* instead of the pattern

... Preprocessing Strings

A *trie* ...

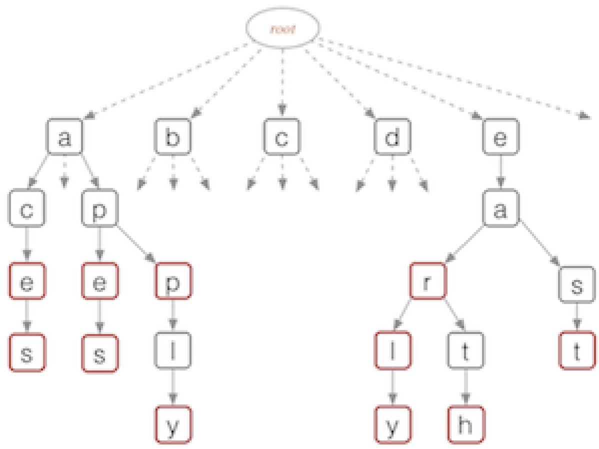
- is a compact data structure for representing a set of strings
  - e.g. all the words in a text, a dictionary etc.
- supports pattern matching queries in time proportional to the pattern size

Note: Trie comes from *retrieval*, but is pronounced like "try" to distinguish it from "tree"

Tries

Reminder (COMP9021):

Tries are trees organised using parts of keys (rather than whole keys)



... Tries

Each node in a trie ...

- contains one part of a key (typically one character)
- may have up to 26 children
- may be tagged as a "finishing" node
- but even "finishing" nodes may have children

Depth d of trie = length of longest key value

Cost of searching *O(d)* (independent of *n*)

... Tries

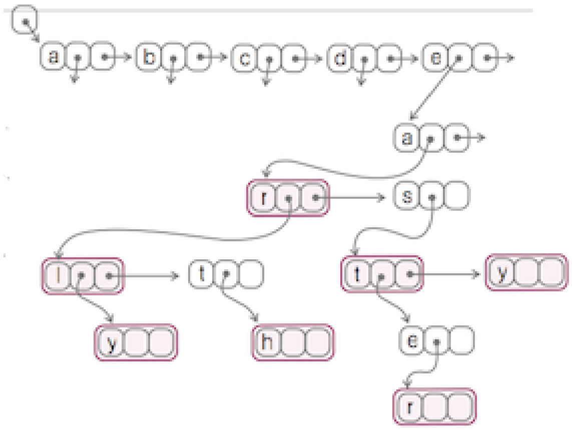
Possible trie representation:

```
#define ALPHABET_SIZE 26

typedef struct Node *Trie;

typedef struct Node {
   bool finish;      // last char in key?
   Item data;        // no Item if !finish
```

```
    Trie child[ALPHABET_SIZE];
} Node;

typedef char *Key;
```

... Tries

Note: Can also use BST-like nodes for more space-efficient implementation of tries

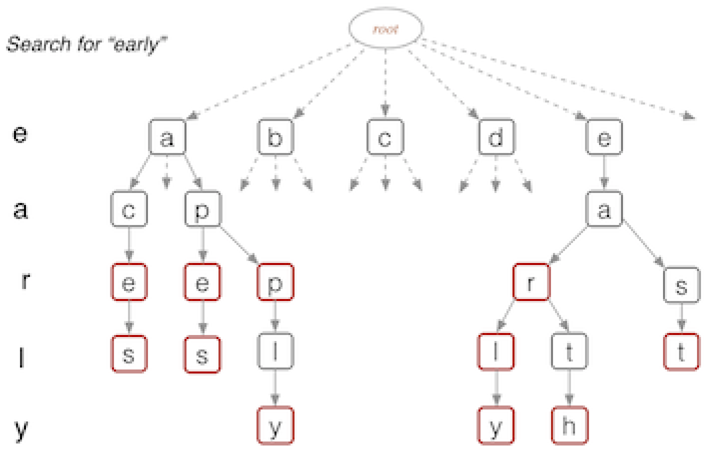


Trie Operations

Basic operations on tries:

1. search for a key
2. insert a key

Trie Operations



... Trie Operations

Traversing a path, using char-by-char from Key:

```
find(trie,key):
|  Input  trie, key
|  Output pointer to element in trie if key found
|         NULL otherwise
|
|  node=trie
|  for each char in key do
```

```
|  |  if node.child[char] exists then
|  |     node=node.child[char]  // move down one level
|  |  else
|  |     return NULL
|  |  end if
|  end for
|  if node.finish then           // "finishing" node reached?
|     return node
|  else
|     return NULL
|  end if
```

... Trie Operations

Insertion into Trie:

```
insert(trie,item,key):
|  Input  trie, item with key of length m
|  Output trie with item inserted
|
|  if trie is empty then
|     t=new trie node
|  end if
|  if m=0 then
|     t.finish=true, t.data=item
|  else
|     t.child[key[0]]=insert(t.child[key[0]],item,key[1..m-1])
|  end if
|  return t
```

... Trie Operations

Analysis of standard tries:

- $O(n)$ space
- insertion and search in time $O(d \cdot m)$
    - n … total size of text (e.g. sum of lengths of all strings in a given dictionary)
    - m … size of the string parameter of the operation (the "key")
    - d … size of the underlying alphabet (e.g. 26)

Word Matching With Tries

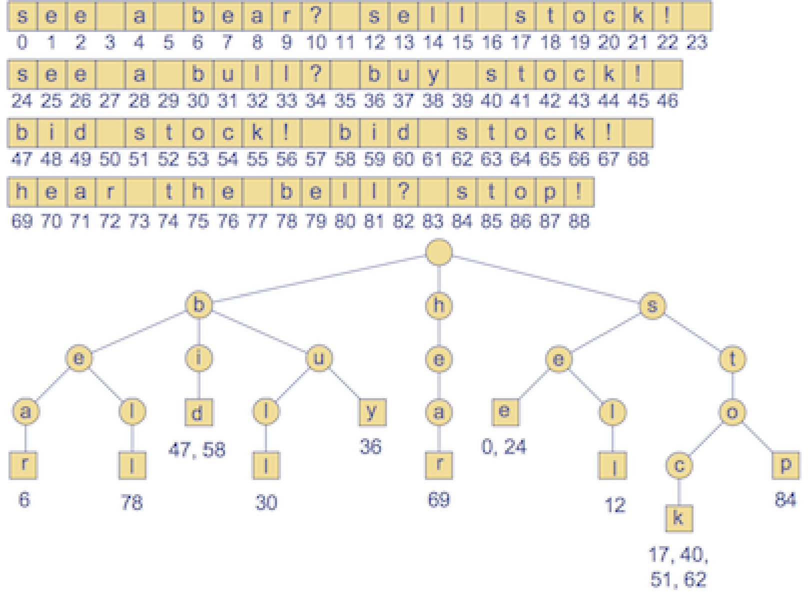
Word Matching with Tries

Preprocessing the text:

1. Insert all searchable words of a text into a trie
2. Each leaf stores the occurrence(s) of the associated word in the text

... Word Matching with Tries

Example text and corresponding trie of searchable words:

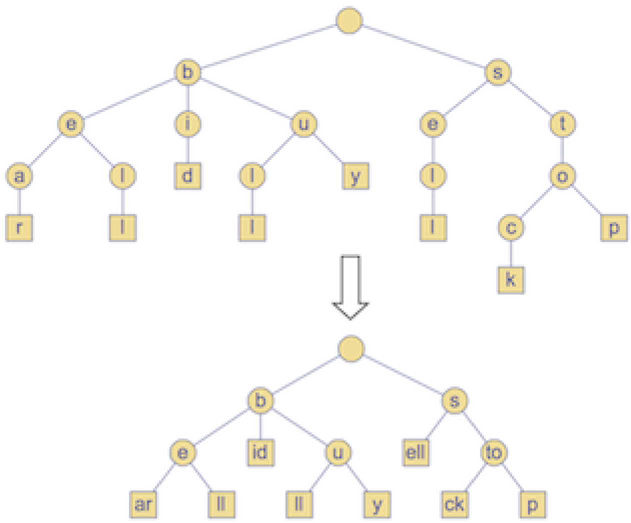


Compressed Tries

Compressed tries …

- have internal nodes of degree ≥ 2
- are obtained from standard tries by compressing "redundant" chains of nodes

Example:

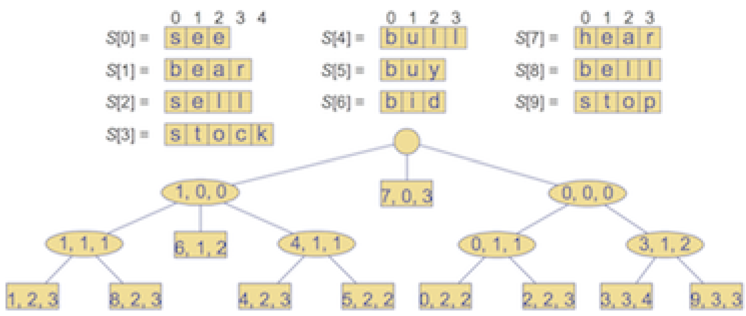


... Compressed Tries

Possible compact representation of a compressed trie to encode an array *S* of strings:

- nodes store *ranges of indices* instead of substrings
  - use triple *(i,j,k)* to represente substring *S[i][j..k]*
- requires *O(s)* space (*s* = #strings in array *S*)

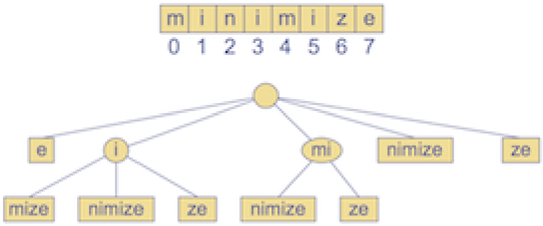
Example:



Pattern Matching With Suffix Tries

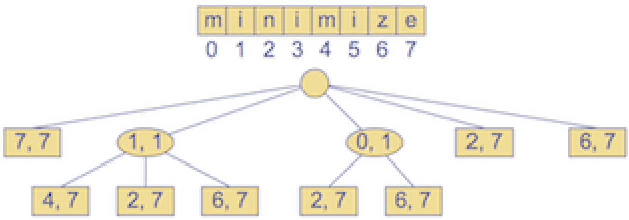
The *suffix trie* of a text *T* is the compressed trie of all the suffixes of *T*

Example:



... Pattern Matching With Suffix Tries

Compact representation:



... Pattern Matching With Suffix Tries

Input:

- compact suffix trie for text *T*
- pattern *P*

Goal:

- find starting index of a substring of *T* equal to *P*

... Pattern Matching With Suffix Tries


```
suffixTrieMatch(trie,P):
|  Input  compact suffix trie for text T, pattern P of length m
|  Output starting index of a substring of T equal to P
|         -1 if no such substring exists
|
|  j=0, v=root of trie
|  repeat
|  |  // we have matched j+1 characters
|  |  if ∃w∈children(v) such that P[j]=T[start(w)] then
|  |  |  i=start(w)              // start(w) is the start index of w
|  |  |  x=end(w)-i+1            // end(w) is the end index of w
|  |  |  if m≤x then    // length of suffix ≤ length of the node label?
|  |  |     if P[j..j+m-1]=T[i..i+m-1] then
|  |  |        return i-j       // match at i-j
|  |  |     else
|  |  |        return -1        // no match
|  |  |  else if P[j..j+x-1]=T[i..i+x-1] then
|  |  |     j=j+x, m=m-x        // update suffix start index and length
|  |  |     v=w                 // move down one level
|  |  |  else return -1         // no match
|  |  |  end if
|  |  else
|  |     return -1
```

```
|   |  end if
|  until v is leaf node
|  return -1                  // no match
```

... Pattern Matching With Suffix Tries

Analysis of pattern matching using suffix tries:

Suffix trie for a text of size *n* …

- can be constructed in *O(n)* time
- uses *O(n)* space
- supports pattern matching queries in *O(s·m)* time
  - *m* … length of the pattern
  - *s* … size of the alphabet

Text Compression

Text Compression

Problem: Efficiently encode a given string *X* by a smaller string *Y*

Applications:

- Save memory and/or bandwidth

Huffman's algorithm

- computes frequency *f(c)* for each character *c*
- encodes high-frequency characters with short code
- no code word is a prefix of another code word
- uses optimal *encoding tree* to determine the code words

... Text Compression

Code … mapping of each character to a binary code word

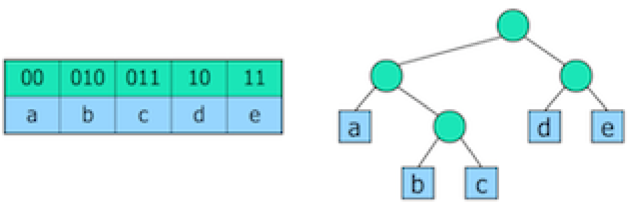
Prefix code … binary code such that no code word is prefix of another code word

Encoding tree …

- represents a prefix code
- each leaf stores a character
- code word given by the path from the root to the leaf   (0 for left child, 1 for right child)

... Text Compression

Example:



... Text Compression

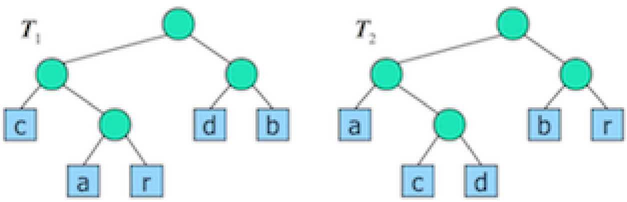
Text compression problem

Given a text T, find a prefix code that yields the shortest encoding of T

- short codewords for frequent characters
- long code words for rare characters

... Text Compression

Example: T = `abracadabra`



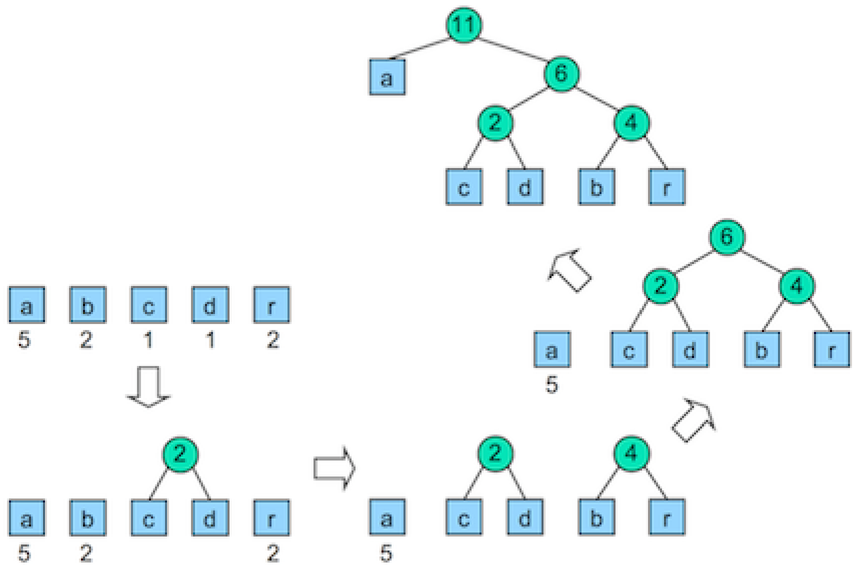
T_1 requires 29 bits to encode text T,
T_2 requires 24 bits

... Text Compression

Huffman's algorithm

- computes frequency *f(c)* for each character
- successively combines pairs of lowest-frequency characters to build encoding tree "bottom-up"

Example: `abracadabra`



Huffman Code

Huffman's algorithm using priority queue:

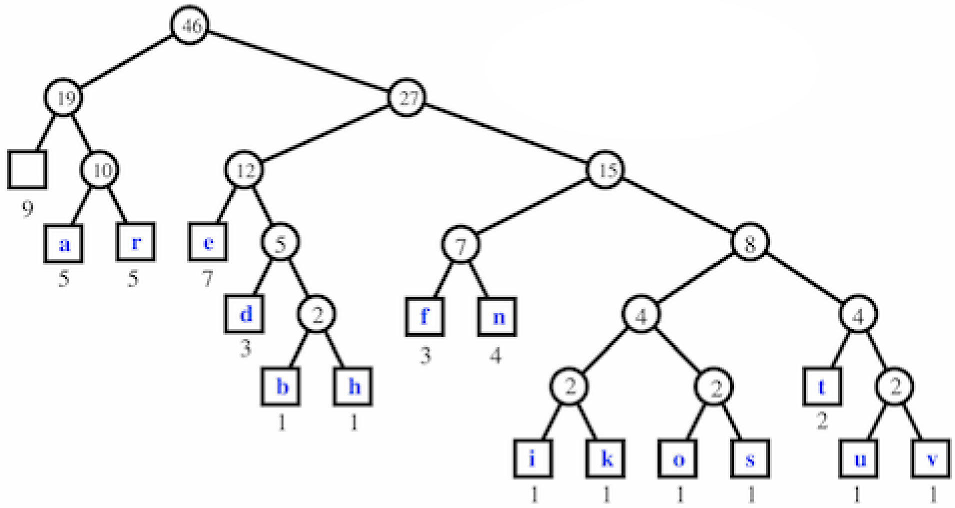
```
HuffmanCode(T):
|  Input  string T of size n
|  Output optimal encoding tree for T
|
|  compute frequency array
|  Q=new priority queue
```

```
   for all characters c do
      T=new single-node tree storing c
      join(Q,T) with frequency(c) as key
   end for
   while |Q|≥2 do
      f1=Q.minKey(), T1=leave(Q)
      f2=Q.minKey(), T2=leave(Q)
      T=new tree node with subtrees T1 and T2
      join(Q,T) with f1+f2 as key
   end while
   return leave(Q)
```

... Huffman Code

Larger example: a fast runner need never be afraid of the dark

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



... Huffman Code

Analysis of Huffman's algorithm:

- $O(n+d{\cdot}log\ d)$ time
  - n … length of the input text T
  - d … number of distinct characters in T

Summary

- Alphabets and words
- Pattern matching
  - Boyer-Moore, Knuth-Morris-Pratt
- Tries
- Text compression
  - Huffman code

- Suggested reading:
  - Tries … Sedgewick, Ch.15.2