

Week 04b: String Algorithms, Approximation

Strings

Strings

2/86

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

3/86

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

$v\omega$ denotes the *concatenation* of strings v and ω

Note: $length(v\omega) = length(v) + length(\omega)$ $\lambda\omega = \omega = \omega\lambda$

... Strings

4/86

Notation:

- **substring** of P ... any string Q such that $P = vQ\omega$, for some $v, \omega \in \Sigma^*$
 - **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

5/86

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

7/86

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

8/86

Reminder:

In C a string is an array of `chars` 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

11/86

Example (pattern checked *backwards*):



- Text ... abacaab
- Pattern ... abacab

... Pattern Matching

12/86

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

13/86

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 and 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

14/86

Naive pattern matching runs in $O(n \cdot m)$

Examples of worst case (forward checking):

- $T = \text{aaa...ah}$
- $P = \text{aaah}$
- may occur in DNA sequences
- unlikely in English text

Exercise #2: Naive Matching

15/86

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
abcdexxxxxxxxx  xxxxxabcde

```

Boyer-Moore Algorithm

17/86

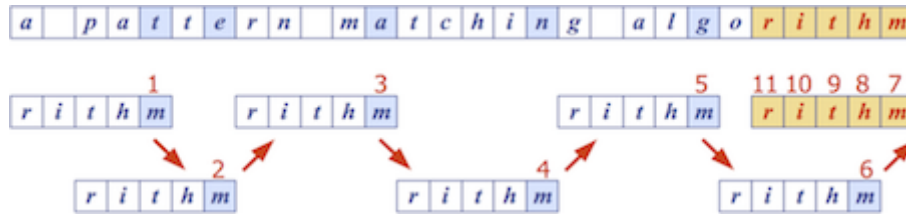
heuristics 启发

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: $\Sigma = \{a, b, c, d\}$, $P = acab$

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

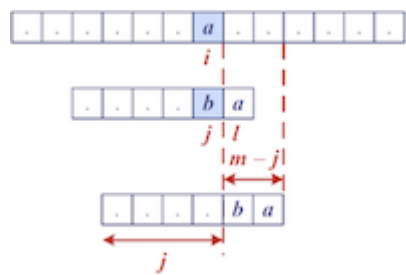
```

L=lastOccurenceFunction( $P, \Sigma$ )
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
    i=i+m- $\min(j, 1+L[T[i]])$  // character-jump
    j=m-1
  end if
until i>n
return -1               // no match
  
```

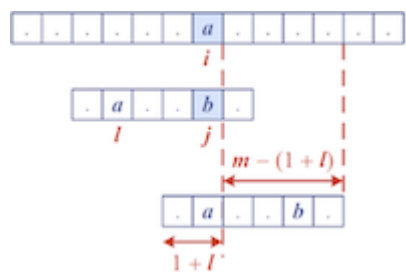
- **Biggest jump (m characters ahead) occurs when $L[T[i]] = -1$**

... Boyer-Moore Algorithm

Case 1: $j \leq l + L[c]$



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

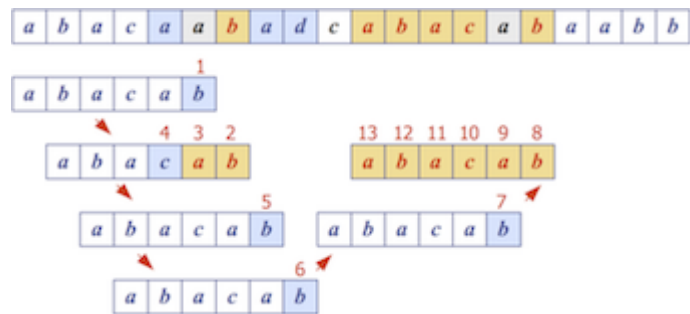


Exercise #3: Boyer-Moore algorithm

For the alphabet $\Sigma = \{a, b, c, d\}$

- 1. compute last-occurrence function L for pattern $P = \text{abacab}$
- 2. trace Boyer-More on P and text $T = \text{abacaabadcabacabaabb}$
 - 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:
 - $T = \text{aaa} \dots \text{a}$
 - $P = \text{baaa}$
- Worst case may occur in images and DNA sequences but unlikely in English texts
 \Rightarrow Boyer-Moore significantly faster than naive matching on English text

Knuth-Morris-Pratt Algorithm

25/86

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:

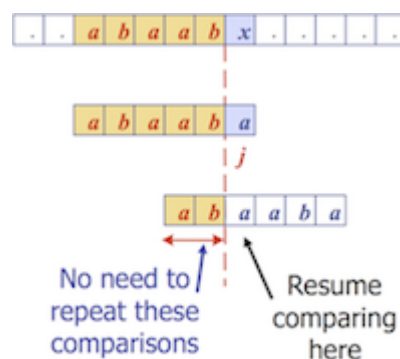
- Q is a *prefix* of P ... $P = Q\omega$, for some $\omega \in \Sigma^*$
- Q is a *suffix* of P ... $P = \omega Q$, for some $\omega \in \Sigma^*$

... Knuth-Morris-Pratt Algorithm

26/86

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

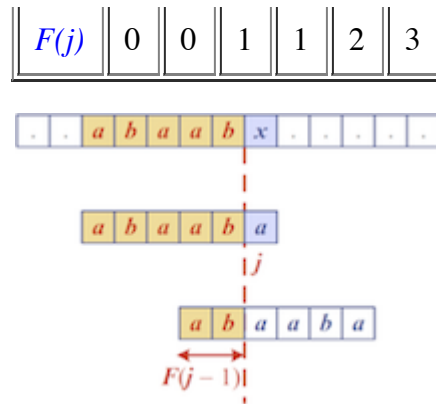
27/86

KMP preprocesses the pattern $P[0..m-1]$ 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 $P[1..j]$ for each position $j=0..m-1$
- if mismatch occurs at $P_j \Rightarrow$ advance j to $F(j-1)$

Example: $P = \text{abaaba}$

j	0	1	2	3	4	5
P_j	a	b	a	a	b	a



... Knuth-Morris-Pratt Algorithm

28/86

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)
j=0                                // number of characters matched
i=0                                // scan the text from left to right
while i<n do
    if T[i]=P[j] then
        i=i+1, j=j+1
        if j=m then                // all of P matched?
            return i-j              // match found at i-j
        end if
    end if
    else                            // next character does not match
        j=F[j]
    end while
    i=i+1
end while
return -1                          // no match

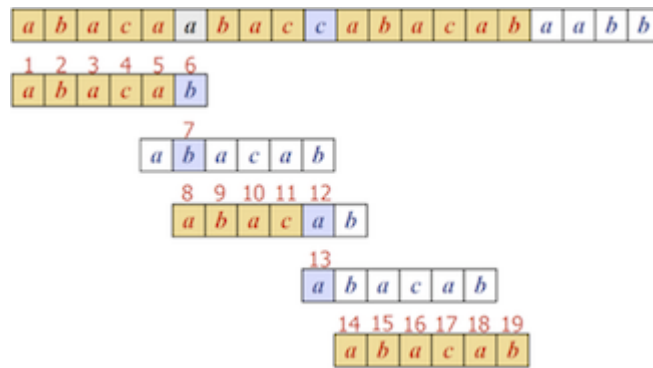
```

Exercise #4: KMP-Algorithm

29/86

1. compute failure function F for pattern $P = \text{abacab}$
2. trace Knuth-Morris-Pratt on P and text $T = \text{abacaabaccabacabaabb}$
 - 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

31/86

Construction of the failure function matches pattern against 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

32/86

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

33/86

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

34/86

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

36/86

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

37/86

A *trie* ...

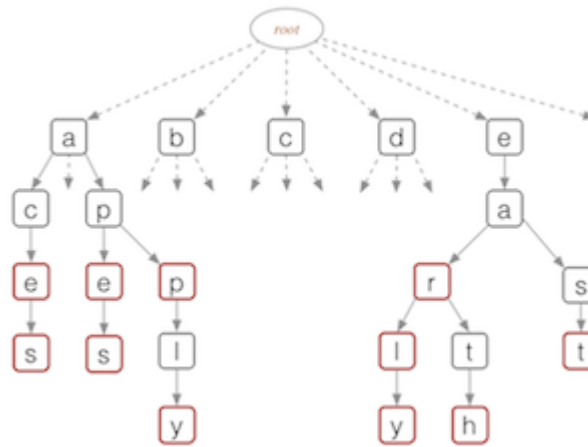
- is a compact data structure for representing a set of strings
 - e.g. all the words in a text, a dictionary etc.
- supports pattern matching queries in time proportional to the pattern size

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

Tries 字典树

38/86

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



... Tries

39/86

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

40/86

Possible trie representation:

```
#define ALPHABET_SIZE 26

typedef struct Node *Trie;

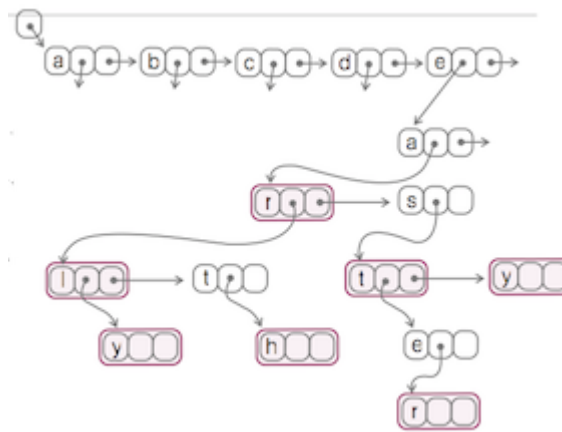
typedef struct Node {
    bool finish;        // last char in key?
    Item data;          // no Item if !finish
    Trie child[ALPHABET_SIZE];
} Node;

typedef char *Key;
```

... Tries

41/86

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



Trie Operations

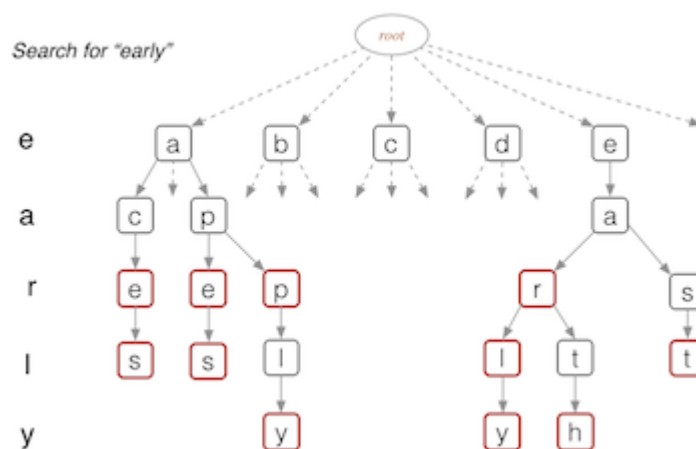
42/86

Basic operations on tries:

1. search for a key
2. insert a key

Trie Operations

43/86



... Trie Operations

44/86

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

45/86

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

46/86

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

48/86

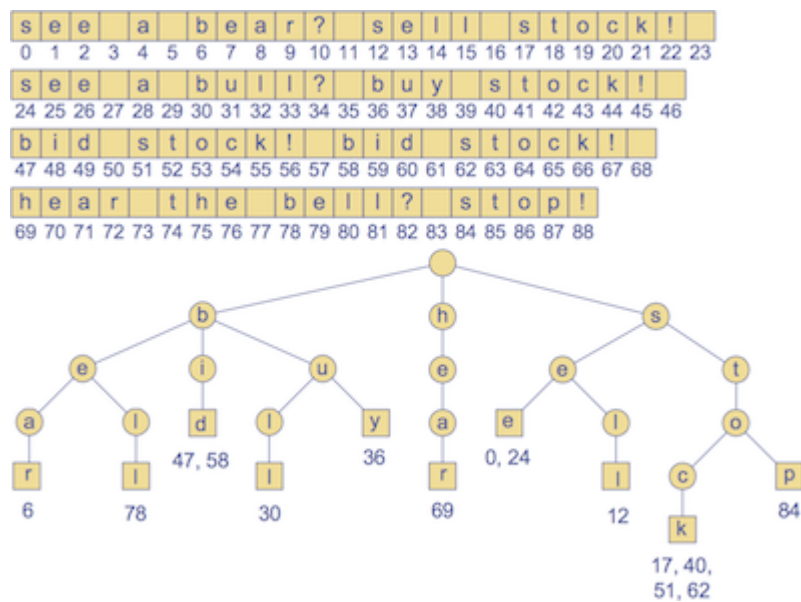
Preprocessing the text:

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

49/86

Example text and corresponding trie of searchable words:



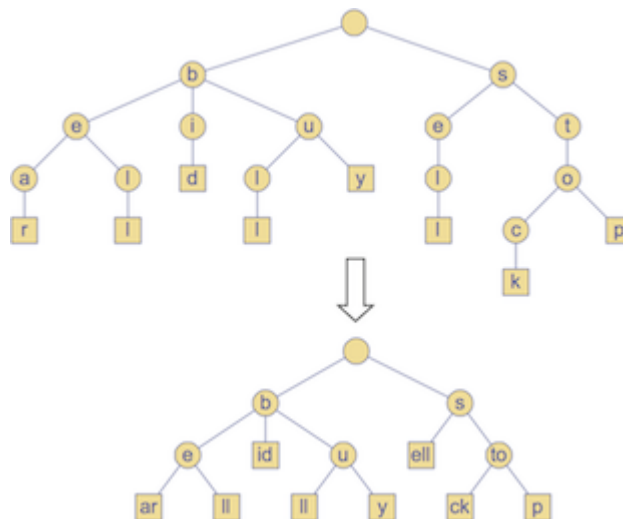
Compressed Tries

50/86

Compressed tries ...

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

Example:



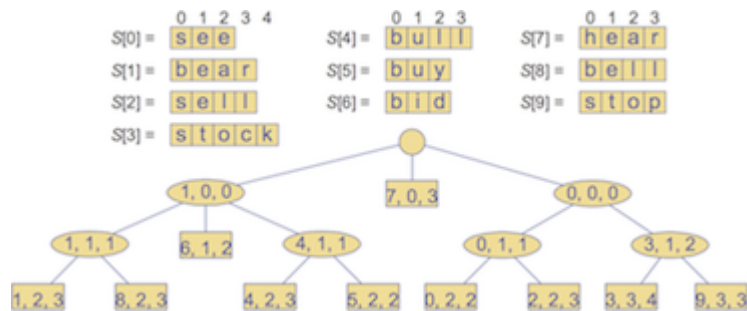
... Compressed Tries

51/86

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 represent substring $S[i][j..k]$
- requires $O(s)$ space ($s = \#$ strings in array S)

Example:

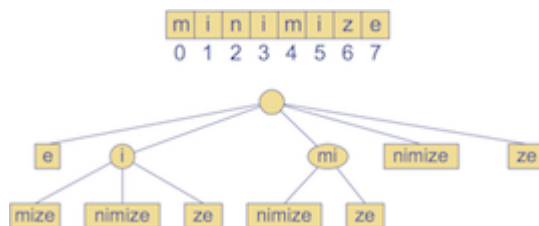


Pattern Matching With Suffix Tries

52/86

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

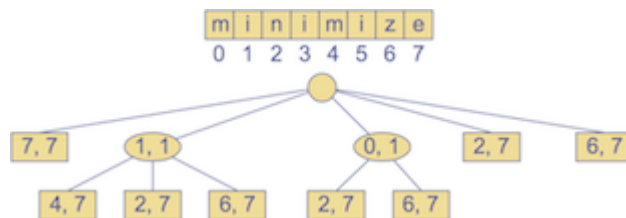
Example:



... Pattern Matching With Suffix Tries

53/86

Compact representation:



... Pattern Matching With Suffix Tries

54/86

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

55/86

`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  $\exists w \in \text{children}(v)$  such that  $P[j]=T[\text{start}(w)]$  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  $m \leq x$  then      // length of suffix  $\leq$  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

56/86

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 $O(s \cdot m)$ time
 - m ... length of the pattern
 - s ... size of the alphabet

Text Compression

Text Compression

58/86

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

59/86

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

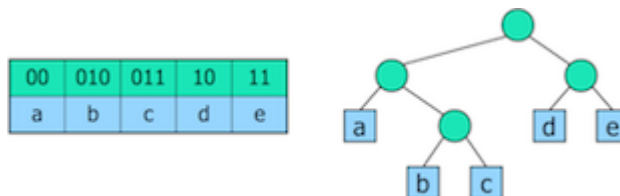
Encoding tree ...

- 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

60/86

Example:



... Text Compression

61/86

Text compression problem

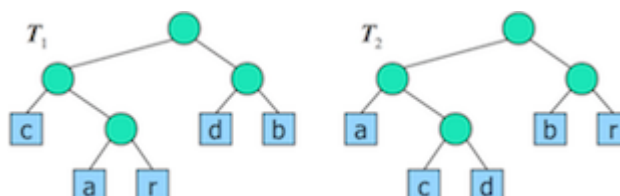
Given a text T , find a prefix code that yields the shortest encoding of T

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

... Text Compression

62/86

Example: $T = \text{abracadabra}$



T_1 requires 29 bits to encode text T ,

T_2 requires 24 bits

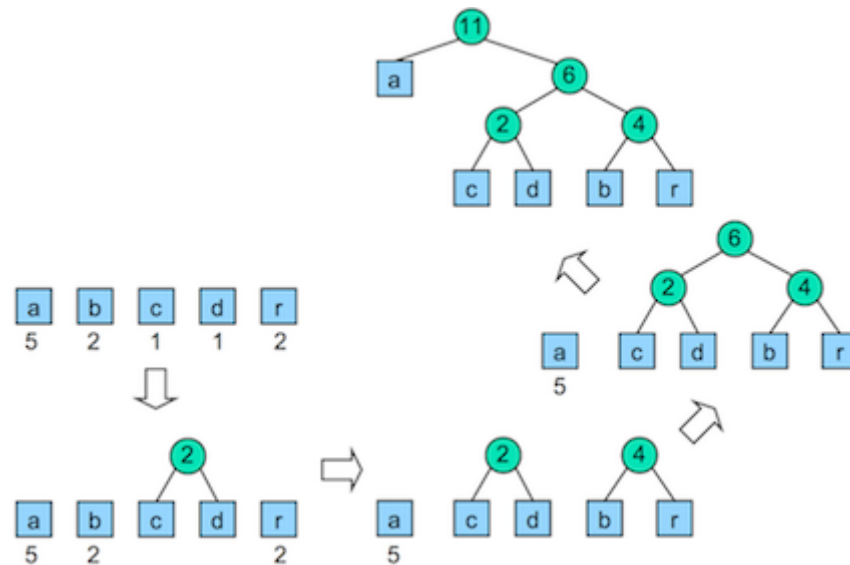
... Text Compression

63/86

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

64/86

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

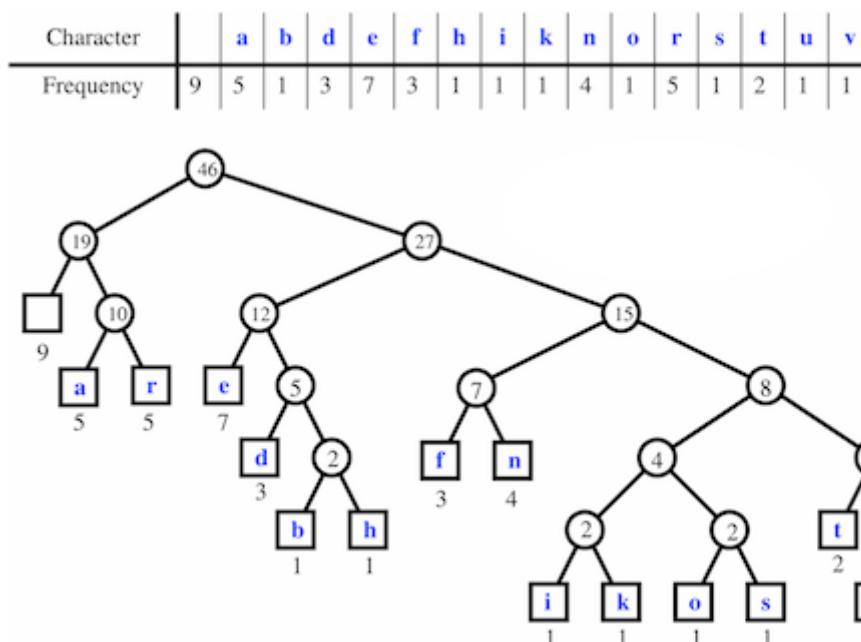
end while

return leave(Q)

... Huffman Code

65/86

Larger example: [a fast runner need never be afraid of the dark](#)



... Huffman Code

66/86

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

Approximation

Approximation for Numerical Problems

68/86

Approximation is often used to solve numerical problems by

- solving a simpler, but much more easily solved, problem
- where this new problem gives an approximate solution
- and refine the method until it is "accurate enough"

Examples:

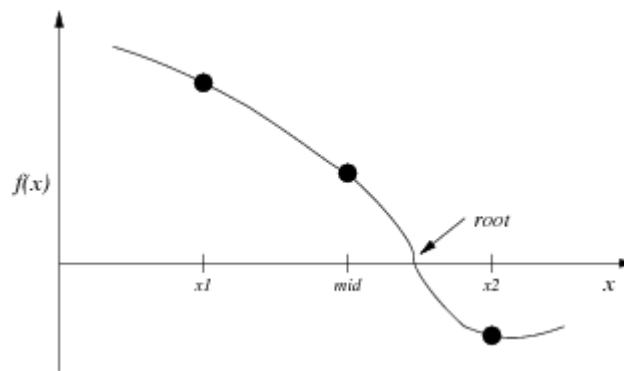
- roots of a function f
- length of a curve determined by a function f
- ... and many more

... Approximation for Numerical Problems

69/86

Example: Finding Roots

Find where a function crosses the x-axis:



Generate and test: move x_1 and x_2 together until "close enough"

... Approximation for Numerical Problems

70/86

A simple approximation algorithm for finding a root in a given interval:

```

bisection(f, x1, x2):
|   Input  function f, interval [x1, x2]
|   Output x ∈ [x1, x2] with f(x) ≈ 0
|
|   repeat
|   |   mid = (x1 + x2) / 2
|   |   if f(x1) * f(mid) < 0 then
|   |   |   x2 = mid          // root to the left of mid
|   |   else
|   |   |   x1 = mid          // root to the right of mid
|   |   end if
|   until f(mid) = 0 or x2 - x1 < ε    // ε: accuracy
|   end while
|   return mid
  
```

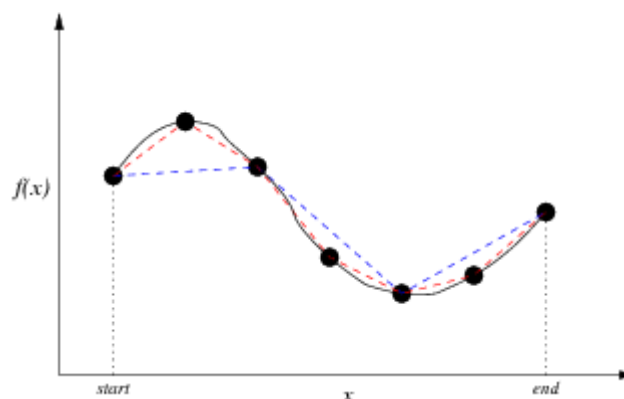
bisection guaranteed to converge to a root if f continuous on $[x_1, x_2]$ and $f(x_1)$ and $f(x_2)$ have opposite signs

... Approximation for Numerical Problems

71/86

Example: Length of a Curve

Estimate length: approximate curve as sequence of straight lines.



... Approximation for Numerical Problems

72/86

```

curveLength(f,start,end):
|   Input   function f, start and end point
|   Output curve length between f(start) and f(end)
|
|   length=0,  $\delta=(\text{end}-\text{start})/\text{StepSize}$ 
|   for each  $x \in [\text{start}+\delta, \text{start}+2\delta, \dots, \text{end}]$  do
|       length = length +  $\text{sqrt}(\delta^2 + (f(x)-f(x-\delta))^2)$ 
|   end for
|   return length

```

Sidetrack: Function Pointers

73/86

Function pointers ...

- are references to memory address of a function
- are pointer values and can be assigned/passed

Function pointer variables/parameters are declared as:

```

typeOfReturnValue (*fname)(typeOfArguments)

```

... Sidetrack: Function Pointers

74/86

Example:

```

// define a function of type double → double
double myfun(double x) {
    return sqrt(1-x*x);
}

double curveLength(double start, double end, double (*f)(double)) {
    ...
    deltaY = f(x) - f(x-delta);
    length += sqrt(delta*delta + deltaY*deltaY);
    ...
}

printf("%.10f\n", curveLength(-1, 1, myfun));

```

Approximation for Numerical Problems

75/86

Trade-offs in curve length approximation algorithm:

- large step size ...
 - less steps, less computation (faster), lower accuracy
- small step size ...
 - more steps, more computation (slower), higher accuracy

However, too many steps may lead to higher rounding error.

Each f has an optimal step size ...

- but this is difficult to determine in advance

... Approximation for Numerical Problems

Example: `length = curveLength(0, π , sin);`

Convergence when using more and more steps

```
steps =      0, length = 0.000000
steps =     10, length = 3.815283
steps =    100, length = 3.820149
steps =   1000, length = 3.820197
steps =  10000, length = 3.819753
steps = 100000, length = 3.820198
steps =1000000, length = 3.820198
```

Actual answer is 3.820197789...

Approximation for NP-hard Problems

77/86

Approximation is often used for NP-hard problems ...

- computing a near-optimal solution
- in polynomial time

Examples:

- vertex cover of a graph
- subset-sum problem

Vertex Cover

78/86

Reminder: Graph $G = (V, E)$

- set of vertices V
- set of edges E

Vertex cover C of G ...

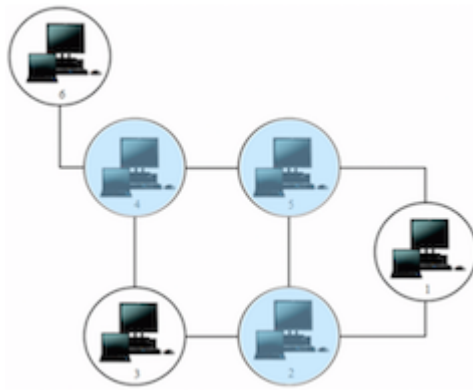
- $C \subseteq V$
- for all edges $(u, v) \in E$ either $v \in C$ or $u \in C$ (or both)

\Rightarrow All edges of the graph are "covered" by vertices in C

... Vertex Cover

79/86

Example (6 nodes, 7 edges, 3-vertex cover):



Applications:

- Computer Network Security
 - compute minimal set of routers to cover all connections
- Biochemistry

... Vertex Cover

80/86

size of vertex cover $C \dots |C|$ (number of elements in C)

optimal vertex cover ... a vertex cover of minimum size

Theorem.

Determining whether a graph has a vertex cover of a given size k is an NP-complete problem.

... Vertex Cover

81/86

An approximation algorithm for vertex cover:

```

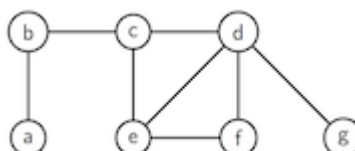
approxVertexCover(G):
  Input undirected graph  $G=(V,E)$ 
  Output vertex cover of  $G$ 

   $C = \emptyset$ 
   $unusedE = E$ 
  while  $unusedE \neq \emptyset$ 
  |   choose any  $(v,w) \in unusedE$ 
  |    $C = C \cup \{v,w\}$ 
  |    $unusedE = unusedE \setminus \{\text{all edges incident on } v \text{ or } w\}$ 
  end while
  return  $C$ 
  
```

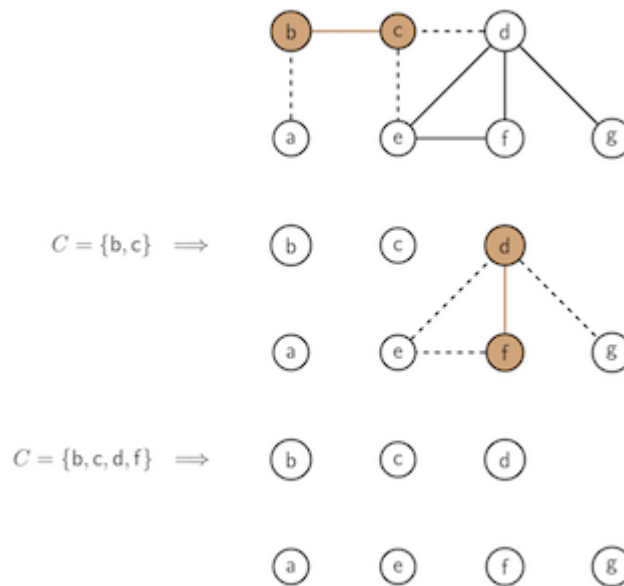
Exercise #5: Vertex Cover

82/86

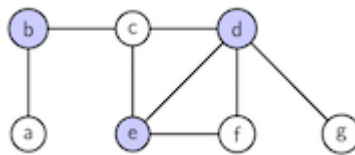
Show how the approximation algorithm produces a vertex cover on:



Possible result:



What would be an optimal vertex cover?



... Vertex Cover

85/86

Theorem.

The approximation algorithm returns a vertex cover *at most twice the size* of an optimal cover.

Proof. Any (optimal) cover must include at least one endpoint of each chosen edge.

Cost analysis ...

- repeatedly select an edge from E
 - add endpoints to C
 - delete all edges in E covered by endpoints

Time complexity: $O(V+E)$ (adjacency list representation)

Summary

86/86

- Alphabets and words
- Pattern matching
 - Boyer-Moore, Knuth-Morris-Pratt
- Tries
- Text compression
 - Huffman code
- Approximation
 - factor-2 approximation for vertex cover

- Suggested reading:
 - tries ... Sedgewick, Ch. 15.2
 - approximation ... Moffat, Ch. 9.4
-

Produced: 6 Jan 2020