



Lect 10: Dictionaries and Tolerant Retrieval

Dr. Subrat Kumar Nayak

Associate Professor

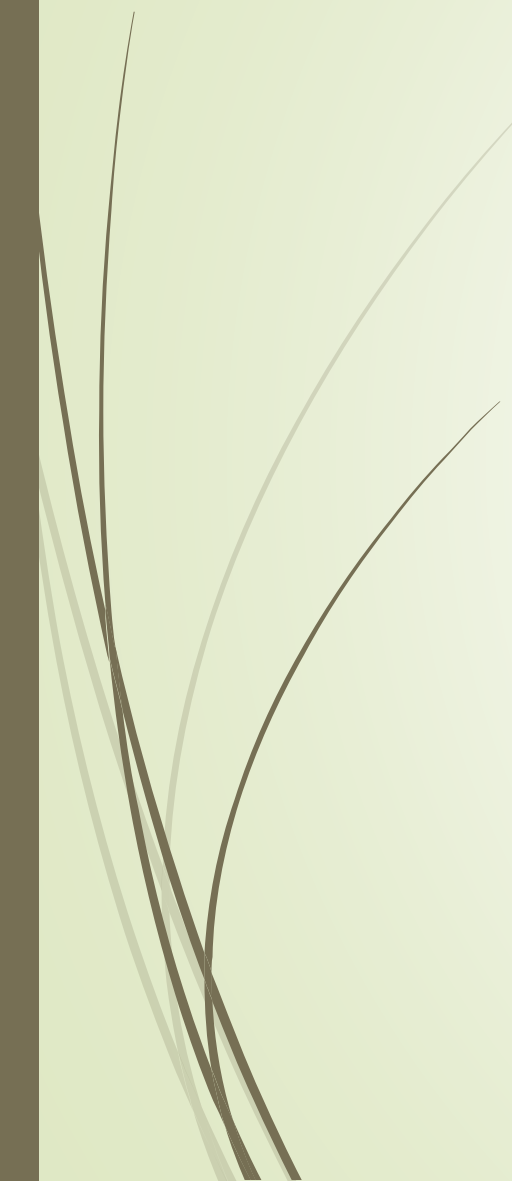
Dept. of CSE, ITER, SOADU

Hashes

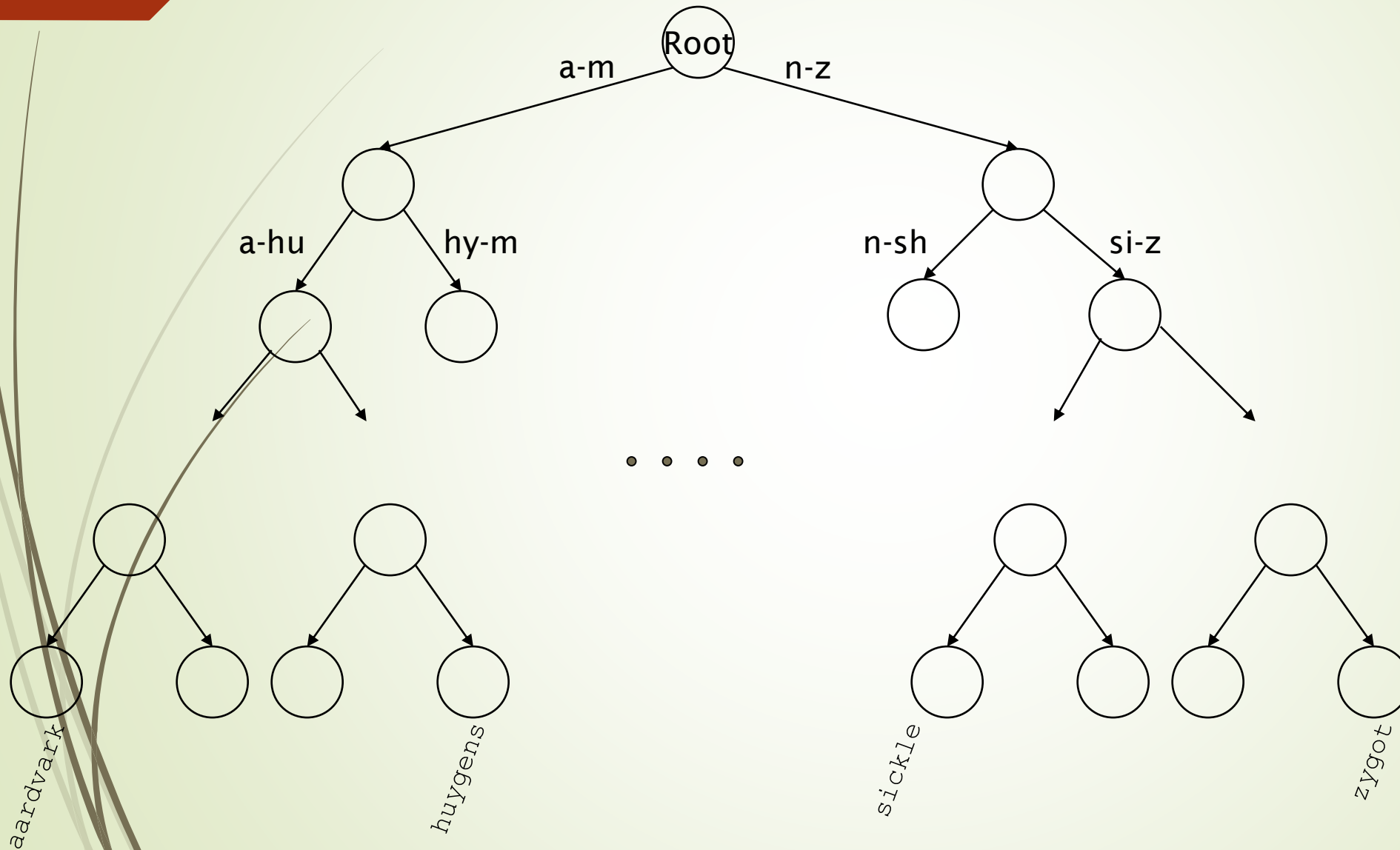
- Hash table: an array with a hash function
 - Input key; output integer: index in array.
 - Hash function: determine where to store / search key.
 - Hash function that minimizes chance of collisions
 - Use all info provided by key (among others).
- Each vocabulary term (key) is hashed into an integer.
- At query time: hash each query term, locate entry in array.
- Pros: Lookup in a hash is faster than lookup in a tree. (Lookup time is constant.)
- Cons:
 - No easy way to find minor variants (resume vs. resumé)
 - No prefix search (all terms starting with [automat](#))
 - Need to rehash everything periodically if vocabulary keeps growing
 - Hash function designed for current needs may not suffice in a few years' time



Trees

- Trees solve the prefix problem (find all terms starting with *automat*).
 - Simplest tree: binary tree
 - Search is slightly slower than in hashes: $O(\log M)$, where M is the size of the vocabulary.
 - $O(\log M)$ only holds for **balanced** trees.
 - Rebalancing binary trees is expensive.
 - **B-trees** mitigate the rebalancing problem.
 - B-tree definition: every internal node has a number of children in the interval $[a, b]$ where a, b are appropriate positive integers, e.g., $[2, 4]$.
- 

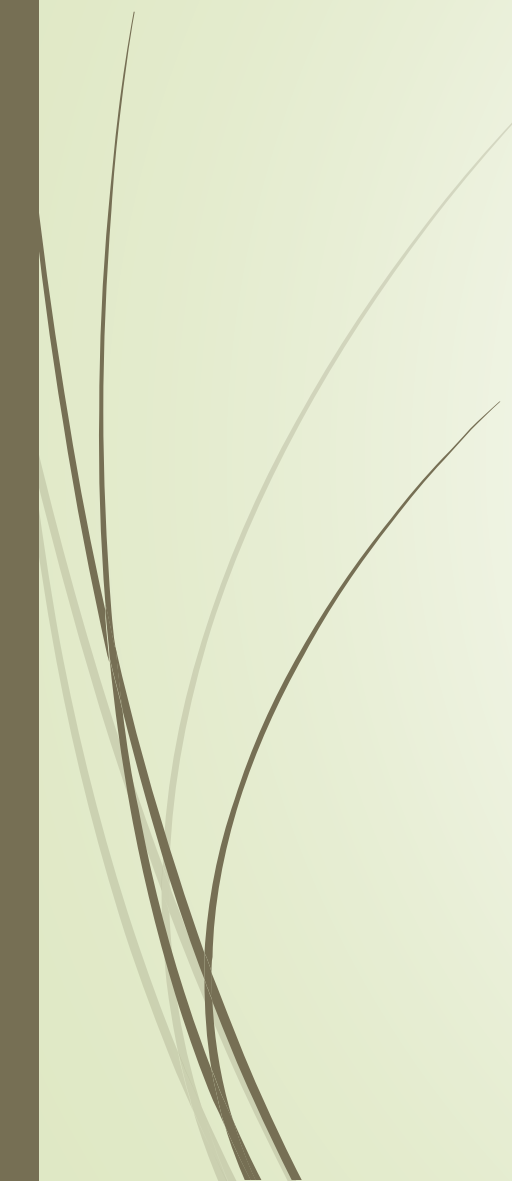
Binary tree



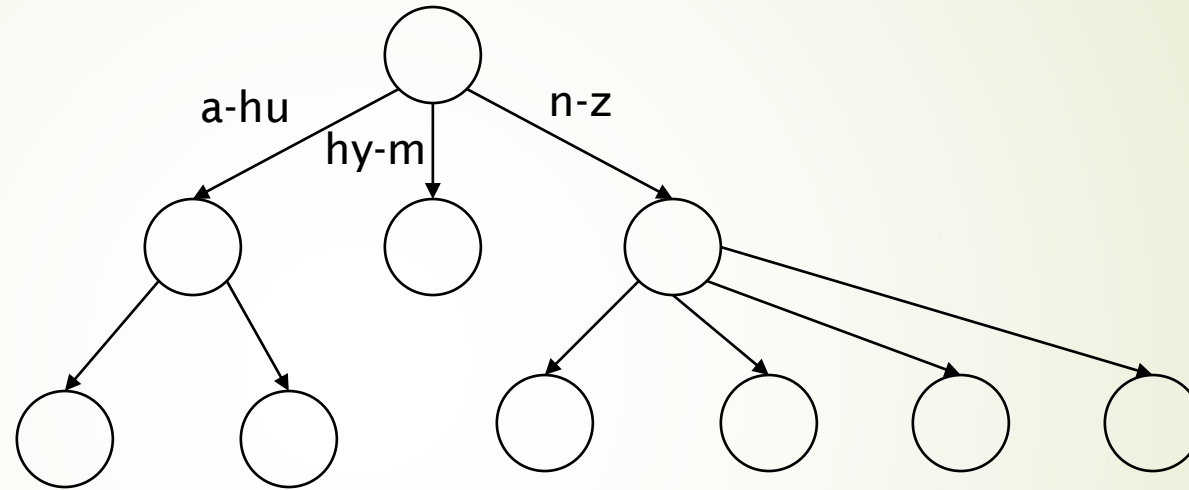
- partition vocabulary terms into two subtrees, those whose first letter is between **a** and **m**, and the rest (actual terms stored in the leaf).
- Anything that is on the **left** subtree is **smaller** than what's on the right.
- Trees solve the prefix problem (find all terms starting with **automat**).



Binary Tree...

- Cost of operations depends on height of tree.
 - Keep height minimum / keep binary tree balanced: for each node, heights of subtrees differ by no more than 1.
 - $O(\log M)$ search for balanced trees, where M is the size of the vocabulary.
 - Search is slightly slower than in hashes But: re-balancing binary trees is expensive (insertion and deletion of terms).
- 

B-tree



- Definition: Every internal node has a number of children in the interval $[a,b]$ where a, b are appropriate natural numbers, e.g., $[2,4]$.

Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings ... but we typically have one
- Pros:
 - Solves the prefix problem (terms starting with *hyp*)
- Cons:
 - Slower: $O(\log M)$ [and this requires *balanced* tree]
 - Rebalancing binary trees is expensive
 - But B-trees mitigate the rebalancing problem

Wild-card Queries

- mon^* : find all docs containing any term beginning with *mon*
- Easy with B-tree dictionary: retrieve all terms t in the range: $mon \leq t < moo$
- $*mon$: find all docs containing any term ending with *mon*
 - Maintain an additional tree for terms *backwards*
 - Then retrieve all terms t in the range: $nom \leq t < non$
- Result: A set of terms that are matches for wildcard query
- Then retrieve documents that contain any of these terms

Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro*cent*** ?

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query:

se*ate AND fil*er

This may result in the execution of many Boolean *AND* queries.

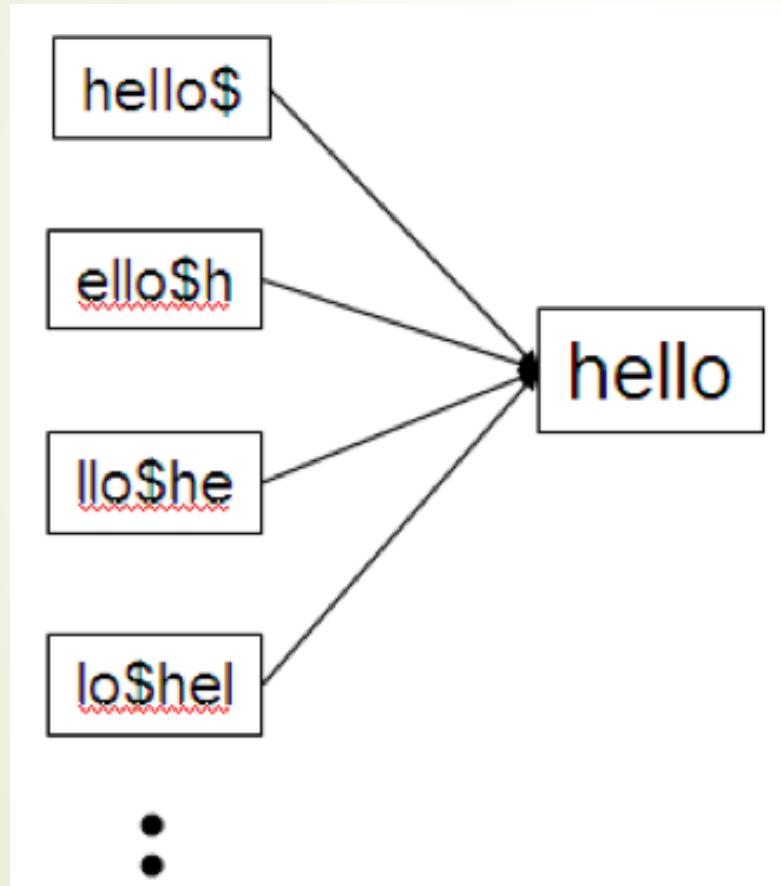
B-trees handle *'s at the end of a query term

- How can we handle *'s in the middle of query term?
 - **co*tion**
- We could look up **co*** AND ***tion** in a B-tree and intersect the two term sets
 - Expensive
- The solution: transform wild-card queries so that the *'s occur at the end
- This gives rise to the **Permuterm** Index.

Permuterm index

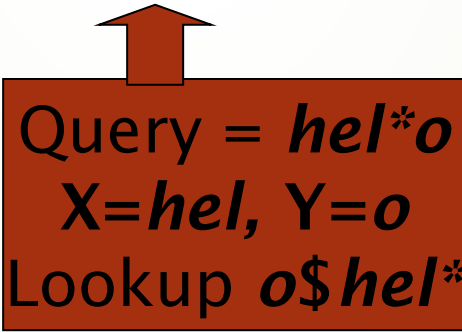
- For term HELLO: add *hello\$*, *ello\$h*, *llo\$he*, *lo\$hel*, and *o\$hell* to the B-tree where \$ is a special symbol
- These terms in the permuterm index are referred as **permuterm vocabulary**

Permuterm \rightarrow term mapping



Permuterm index

- ▶ For term **hello**, index under:
 - ▶ **hello\$, ello\$h, llo\$he, lo\$hel, o\$hell, \$hello**
where \$ is a special symbol.
- ▶ Queries:
 - ▶ **X** lookup on **X\$** **X*** lookup on **\$X***
 - ▶ ***X** lookup on **X\$*** ***X*** lookup on **X***
 - ▶ **X*Y** lookup on **Y\$X*** **X*Y*Z** ??? Exercise!



Query = **hel*o**
X=hel, Y=o
Lookup **o\$hel***

- ▶ Permuterm index would better be called a permuterm **tree**.
- ▶ But permuterm index is the more common name.

Bigram (k -gram) indexes

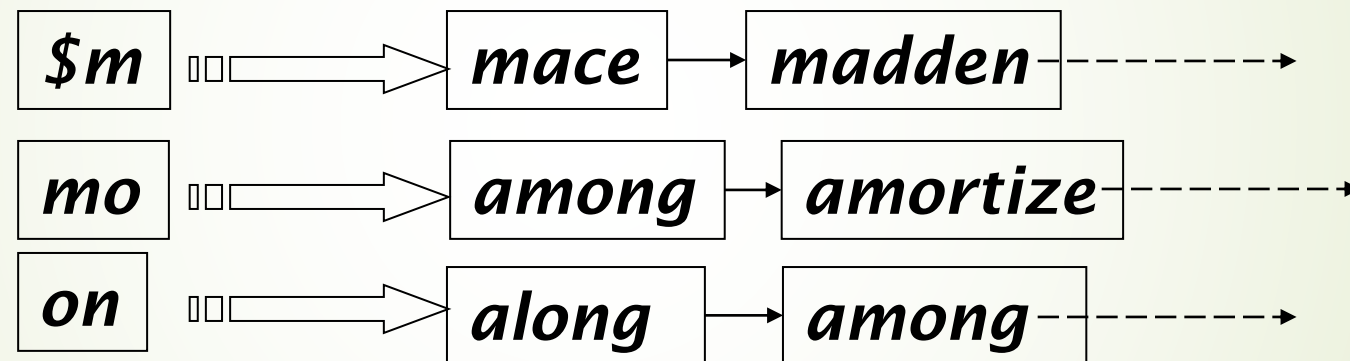
- More space-efficient than permuterm index
- Enumerate all k -grams (sequence of k chars) occurring in any term
- e.g., from text “***April is the cruelest month***” we get the 2-grams (bigrams)

\$a,ap,pr,ri,il,l\$, \$i,is,s\$, \$t,th,he,e\$, \$c,cr,ru,
ue,el,le,es,st,t\$, \$m,mo,on,nt,h\$

- \$ is a special word boundary symbol
- Maintain a second inverted index from bigrams to dictionary terms that match each bigram.

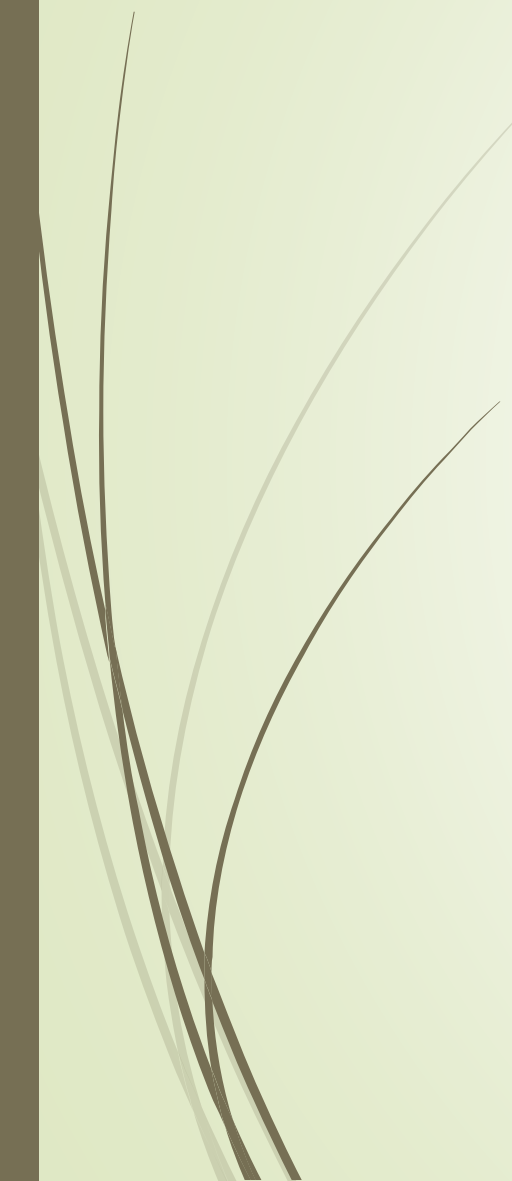
Bigram index example

- The k -gram index finds *terms* based on a query consisting of k -grams (here $k=2$).





k -gram (bigram, trigram, . . .) indexes

- Note that we now have two different types of inverted indexes
 - The term-document inverted index for finding documents based on a query consisting of terms
 - The k -gram index for finding terms based on a query consisting of k -grams
- 

Processing wild-cards

- ▶ Query **mon*** can now be run as
 - ▶ **\$m AND mo AND on** ←
- ▶ Gets terms that match AND version of our wildcard query.
- ▶ ... but also many “false positives” like MOON.
- ▶ Another example: red*
 - ▶ \$re AND red (3-gram index)
 - ▶ May lead to a match on terms such as **retired**, yet do not match original wildcard query red*
- ▶ We must postfilter these terms against query.
- ▶ Surviving terms are then looked up in the term-document inverted index.
- ▶ k-gram index vs. permuterm index
 - k-gram index is more fast and space efficient (compared to permuterm).
 - Permuterm index doesn't require postfiltering.

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions...)
 - `pyth*` AND `prog*`
- If you encourage “laziness” people will respond!

Search

Type your search terms, use ‘*’ if you need to.
E.g., `Alex*` will match Alexander.

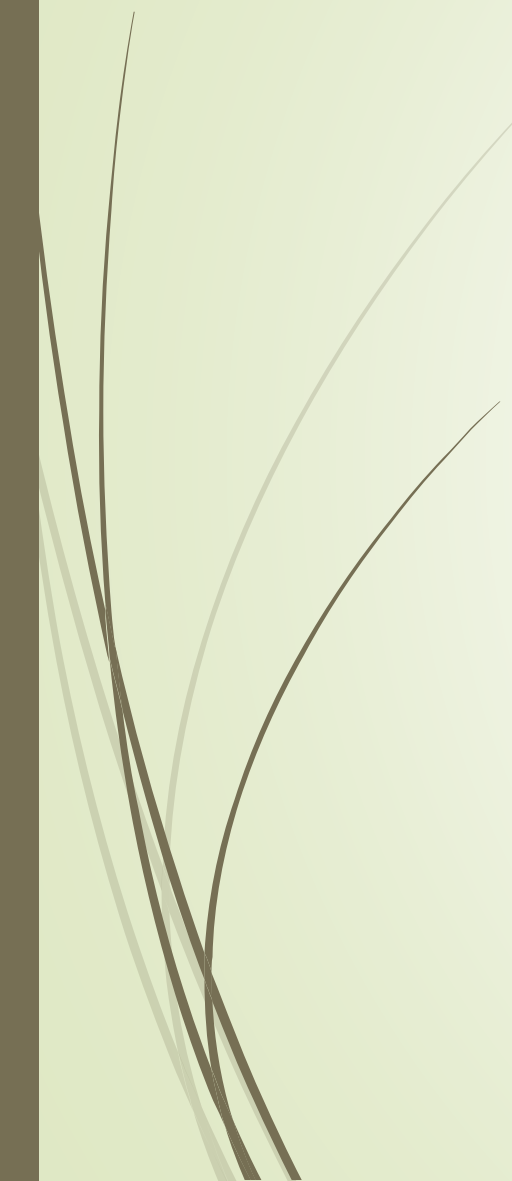
- Which web search engines allow wildcard queries?

Spelling correction

- Two principal uses
 - Correcting documents being indexed
 - Correcting user queries
- Two different methods for spelling correction
- **Isolated word** spelling correction
 - Check each word on its own for misspelling
 - Will not catch typos resulting in correctly spelled words, e.g.,
*an asteroid that fell **form** the sky*
- **Context-sensitive** spelling correction
 - Look at surrounding words
 - Can correct *form/from* error above



Correcting documents

- We're not interested in interactive spelling correction of documents (e.g., MS Word) in this class.
 - In IR, we use document correction primarily for OCR'ed documents. (OCR = optical character recognition)
 - Goal: the dictionary contains fewer misspellings
 - But often we don't change the documents and instead fix the query-document mapping
 - The general philosophy in IR is: don't change the documents.
- 

Query mis-spellings

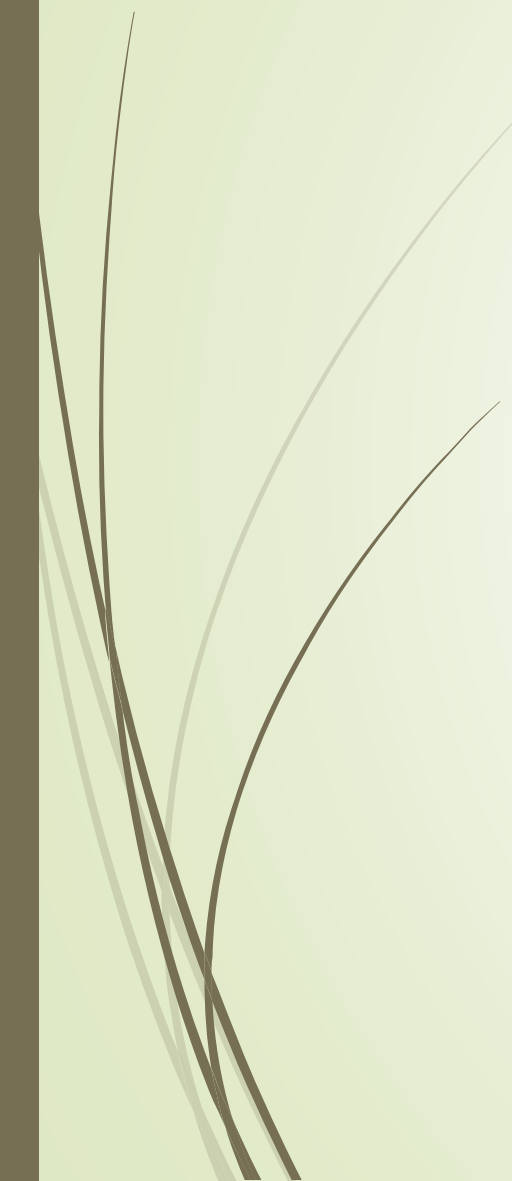
- ▶ Our principal focus here
 - ▶ E.g., the query ***Alanis Morisset***
- ▶ We can either
 - ▶ Retrieve documents indexed by the correct spelling, OR
 - ▶ Return several suggested alternative queries with the correct spelling
 - ▶ ***Did you mean ... ?***

Correcting queries

- First: isolated word spelling correction
 - Premise 1: There is a list of “correct words” from which the correct spellings come.
 - Premise 2: We have a way of computing the **distance** between a misspelled word and a correct word.
- Simple spelling correction algorithm: return the “correct” word that has the smallest distance to the misspelled word.
- Example: *informaton* → *information*
- For the list of correct words, we can use the vocabulary of all words that occur in our collection.




Isolated word correction

- Two basic choices for this
 - A standard lexicon such as
 - Webster's English Dictionary
 - An "industry-specific" lexicon – hand-maintained
 - The lexicon of the indexed corpus
 - E.g., all words on the web
 - All names, acronyms etc.
 - (Including the mis-spellings)
- 



Isolated word correction

- Given a lexicon and a character sequence Q , return the words in the lexicon closest to Q
 - What's "closest"?
 - We'll study several alternatives
 - Edit distance (Levenshtein distance)
 - Weighted edit distance
 - n -gram overlap
- 

Edit distance

- Given two strings S_1 and S_2 , the minimum number of operations to convert one to the other
- Operations are typically character-level
 - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from **dof** to **dog** is 1
 - From **cat** to **act** is 2 (Just 1 with transpose.)
 - from **cat** to **dog** is 3.
- Generally found by dynamic programming.
- Damerau-Levenshtein includes transposition as a fourth possible operation.
- Few more examples:

Levenshtein distance *dog-do*: 1

Levenshtein distance *cat-cart*: 1

Levenshtein distance *cat-cut*: 1

Damerau-Levenshtein distance *cat-act*: 1

Levenshtein distance: Algorithm

LevenshteinDistance(s_1, s_2)

1. **for** $i \leftarrow 0$ **to** $|s_1|$
 2. **do** $m[i, 0] = i$
 3. **for** $j \leftarrow 0$ **to** $|s_2|$
 4. **do** $m[0, j] = j$
 5. **for** $i \leftarrow 1$ **to** $|s_1|$
 6. **do for** $j \leftarrow 1$ **to** $|s_2|$
 7. **do** if ($s_1[i] = s_2[j]$)
 8. **then** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
 9. **else** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
 10. **return** $m[|s_1|, |s_2|]$
- ➡ Operations: insert (cost1), delete(cost1), replace(cost1), copy(cost 0)

Levenshtein distance: Algorithm

LevenshteinDistance(s_1, s_2)

1. **for** $i \leftarrow 0$ **to** $|s_1|$
 2. **do** $m[i, 0] = i$
 3. **for** $j \leftarrow 0$ **to** $|s_2|$
 4. **do** $m[0, j] = j$
 5. **for** $i \leftarrow 1$ **to** $|s_1|$
 6. **do for** $j \leftarrow 1$ **to** $|s_2|$
 7. **do** if ($s_1[i] = s_2[j]$)
 8. **then** $m[i, j] = \min\{m[i-1, j] + 1, \mathbf{m[i, j-1] + 1}, m[i-1, j-1]\}$
 9. **else** $m[i, j] = \min\{m[i-1, j] + 1, \mathbf{m[i, j-1] + 1}, m[i-1, j-1] + 1\}$
 10. **return** $m[|s_1|, |s_2|]$
- ➡ Operations: **insert (cost 1)**, delete(cost 1), replace(cost 1), copy(cost 0)

Levenshtein distance: Algorithm

LevenshteinDistance(s_1, s_2)

1. **for** $i \leftarrow 0$ **to** $|s_1|$
 2. **do** $m[i, 0] = i$
 3. **for** $j \leftarrow 0$ **to** $|s_2|$
 4. **do** $m[0, j] = j$
 5. **for** $i \leftarrow 1$ **to** $|s_1|$
 6. **do for** $j \leftarrow 1$ **to** $|s_2|$
 7. **do** if ($s_1[i] = s_2[j]$)
 8. **then** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
 9. **else** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
 10. **return** $m[|s_1|, |s_2|]$
- ➡ Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein distance: Algorithm

LevenshteinDistance(s_1, s_2)

1. **for** $i \leftarrow 0$ **to** $|s_1|$
 2. **do** $m[i, 0] = i$
 3. **for** $j \leftarrow 0$ **to** $|s_2|$
 4. **do** $m[0, j] = j$
 5. **for** $i \leftarrow 1$ **to** $|s_1|$
 6. **do for** $j \leftarrow 1$ **to** $|s_2|$
 7. **do** if ($s_1[i] = s_2[j]$)
 8. **then** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
 9. **else** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, \mathbf{m[i-1, j-1]+1}\}$
 10. **return** $m[|s_1|, |s_2|]$
- ➡ Operations: insert (cost1), delete(cost1), **replace(cost1)**, copy(cost0)

Levenshtein distance: Algorithm

LevenshteinDistance($s1, s2$)

1. **for** $i \leftarrow 0$ **to** $|s1|$
 2. **do** $m[i, 0] = i$
 3. **for** $j \leftarrow 0$ **to** $|s2|$
 4. **do** $m[0, j] = j$
 5. **for** $i \leftarrow 1$ **to** $|s1|$
 6. **do for** $j \leftarrow 1$ **to** $|s2|$
 7. **do** if ($s1[i] = s2[j]$)
 8. **then** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1]\}$
 9. **else** $m[i, j] = \min\{m[i-1, j] + 1, m[i, j-1] + 1, m[i-1, j-1] + 1\}$
 10. **return** $m[|s1|, |s2|]$
- ➡ Operations: insert (cost1), delete(cost1), replace(cost1), copy(cost0)

Levenshtein distance: Example

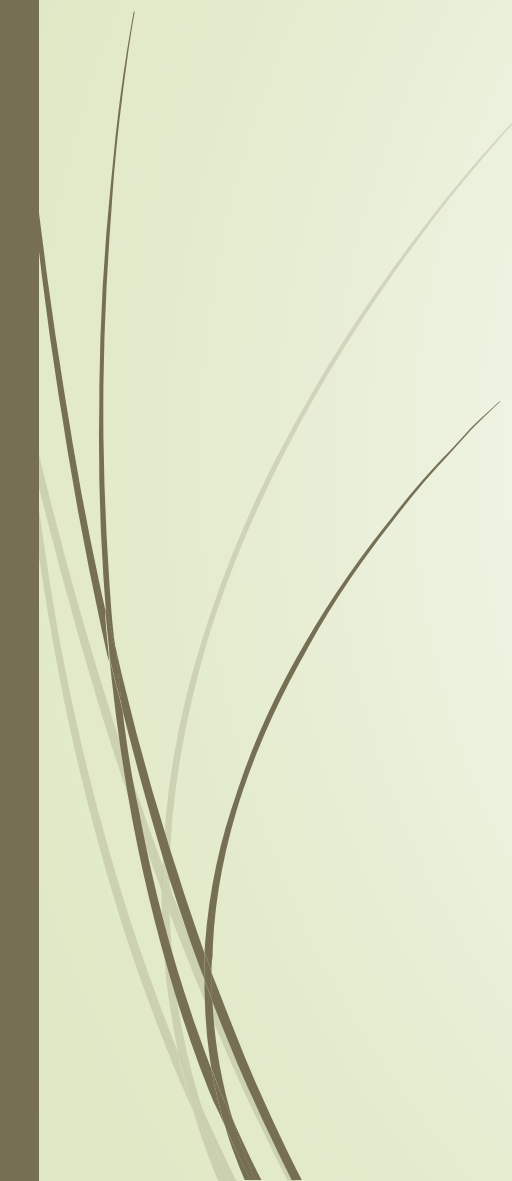
		f	a	s	t
	<div><div></div><div>0</div></div>	<div><div>1</div><div>1</div></div>	<div><div>2</div><div>2</div></div>	<div><div>3</div><div>3</div></div>	<div><div>4</div><div>4</div></div>
c	<div><div>1</div><div>1</div></div>	<div><div>1</div><div>2</div><div>2</div><div>1</div></div>	<div><div>2</div><div>3</div><div>2</div><div>2</div></div>	<div><div>3</div><div>4</div><div>3</div><div>3</div></div>	<div><div>4</div><div>5</div><div>4</div><div>4</div></div>
a	<div><div>2</div><div>2</div></div>	<div><div>2</div><div>2</div><div>3</div><div>2</div></div>	<div><div>1</div><div>3</div><div>3</div><div>1</div></div>	<div><div>3</div><div>4</div><div>2</div><div>2</div></div>	<div><div>4</div><div>5</div><div>3</div><div>3</div></div>
t	<div><div>3</div><div>3</div></div>	<div><div>3</div><div>3</div><div>4</div><div>3</div></div>	<div><div>3</div><div>2</div><div>4</div><div>2</div></div>	<div><div>2</div><div>3</div><div>3</div><div>2</div></div>	<div><div>2</div><div>4</div><div>3</div><div>2</div></div>
s	<div><div>4</div><div>4</div></div>	<div><div>4</div><div>4</div><div>5</div><div>4</div></div>	<div><div>4</div><div>3</div><div>5</div><div>3</div></div>	<div><div>2</div><div>3</div><div>4</div><div>2</div></div>	<div><div>3</div><div>3</div><div>3</div><div>3</div></div>

Each cell of Levenshtein matrix

cost of getting here from my upper left neighbor (copy or replace)	cost of getting here from my upper neighbor (delete)
cost of getting here from my left neighbor (insert)	the minimum of the three possible “movements”; the cheapest way of getting here



Dynamic programming

- Optimal substructure: The optimal solution to the problem contains within it **subsolutions**, i.e., optimal solutions to subproblems.
 - Overlapping subsolutions: The subsolutions overlap. These subsolutions are computed over and over again when computing the global optimal solution in a brute-force algorithm.
 - Subproblem in the case of edit distance: what is the edit distance of two prefixes
 - Overlapping subsolutions: We need most distances of prefixes 3 times – this corresponds to moving right, diagonally, down.
- 

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
 - Meant to capture OCR or keyboard errors
Example: **m** more likely to be mis-typed as **n** than as **q**
 - Therefore, replacing **m** by **n** is a smaller edit distance than by **q**
 - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights



Using edit distances

- Given query, first enumerate all character sequences within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with list of “correct” words
- Show terms you found to user as suggestions
- Alternatively,
 - We can look up all possible corrections in our inverted index and return all docs ... slow
 - We can run with a single most likely correction
- The alternatives disempower the user, but save a round of interaction with the user



Exercise

- 
- 1 Compute Levenshtein distance matrix for OSLO – SNOW
 - 2 What are the Levenshtein editing operations that transform *cat* into *catcat*?

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{\quad}{1} \frac{\quad}{1}$	$\frac{\quad}{2} \frac{\quad}{2}$	$\frac{\quad}{3} \frac{\quad}{3}$	$\frac{\quad}{4} \frac{\quad}{4}$
o	$\frac{\quad}{1} \frac{\quad}{1}$				
s	$\frac{\quad}{2} \frac{\quad}{2}$				
l	$\frac{\quad}{3} \frac{\quad}{3}$				
o	$\frac{\quad}{4} \frac{\quad}{4}$				

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{?}$			
s	$\frac{2}{2}$				
l	$\frac{3}{3}$				
o	$\frac{4}{4}$				

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{1}$			
s	$\frac{2}{2}$				
l	$\frac{3}{3}$				
o	$\frac{4}{4}$				

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{1}$	$\frac{2}{2}$ $\frac{3}{?}$		
s	$\frac{2}{2}$				
l	$\frac{3}{3}$				
o	$\frac{4}{4}$				

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{1}$	$\frac{2}{2}$ $\frac{3}{2}$		
s	$\frac{2}{2}$				
l	$\frac{3}{3}$				
o	$\frac{4}{4}$				

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{1}$	$\frac{2}{2}$ $\frac{3}{2}$	$\frac{2}{3}$ $\frac{4}{?}$	
s	$\frac{2}{2}$				
l	$\frac{3}{3}$				
o	$\frac{4}{4}$				

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{1}$	$\frac{2}{3}$ $\frac{2}{2}$	$\frac{2}{4}$ $\frac{3}{2}$	$\frac{4}{5}$ $\frac{3}{?}$
s	$\frac{2}{2}$				
l	$\frac{3}{3}$				
o	$\frac{4}{4}$				

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{1}$	$\frac{2}{2}$ $\frac{3}{2}$	$\frac{2}{3}$ $\frac{4}{2}$	$\frac{4}{3}$ $\frac{5}{3}$
s	$\frac{2}{2}$	$\frac{1}{3}$ $\frac{2}{1}$	$\frac{2}{2}$ $\frac{3}{2}$	$\frac{3}{3}$ $\frac{3}{3}$	$\frac{3}{4}$ $\frac{4}{3}$
l	$\frac{3}{3}$	$\frac{3}{4}$ $\frac{2}{2}$	$\frac{2}{3}$ $\frac{3}{2}$	$\frac{3}{3}$ $\frac{4}{3}$	$\frac{4}{4}$ $\frac{4}{4}$
o	$\frac{4}{4}$	$\frac{4}{5}$ $\frac{3}{3}$	$\frac{3}{4}$ $\frac{3}{3}$	$\frac{2}{4}$ $\frac{4}{2}$	$\frac{4}{3}$ $\frac{5}{?}$

		s	n	o	w
	$\frac{\quad}{0}$	$\frac{1}{1}$	$\frac{2}{2}$	$\frac{3}{3}$	$\frac{4}{4}$
o	$\frac{1}{1}$	$\frac{1}{2}$ $\frac{2}{1}$	$\frac{2}{3}$ $\frac{2}{2}$	$\frac{2}{4}$ $\frac{3}{2}$	$\frac{4}{5}$ $\frac{3}{3}$
s	$\frac{2}{2}$	$\frac{1}{3}$ $\frac{2}{1}$	$\frac{2}{3}$ $\frac{2}{2}$	$\frac{3}{3}$ $\frac{3}{3}$	$\frac{3}{4}$ $\frac{4}{3}$
l	$\frac{3}{3}$	$\frac{3}{4}$ $\frac{2}{2}$	$\frac{2}{3}$ $\frac{3}{2}$	$\frac{3}{4}$ $\frac{3}{3}$	$\frac{4}{4}$ $\frac{4}{4}$
o	$\frac{4}{4}$	$\frac{4}{5}$ $\frac{3}{3}$	$\frac{3}{4}$ $\frac{3}{3}$	$\frac{2}{4}$ $\frac{4}{2}$	$\frac{4}{5}$ $\frac{3}{3}$

		s		n		o		w	
		<div><div></div><div>0</div></div>	<div><div>1</div><div>1</div></div>	<div><div>2</div><div>2</div></div>	<div><div>3</div><div>3</div></div>	<div><div>4</div><div>4</div></div>			
o		<div><div>1</div><div>1</div></div>	<div><div>1</div><div>2</div><div>2</div><div>1</div></div>	<div><div>2</div><div>3</div><div>2</div><div>2</div></div>	<div><div>2</div><div>4</div><div>3</div><div>2</div></div>	<div><div>4</div><div>5</div><div>3</div><div>3</div></div>			
s		<div><div>2</div><div>2</div></div>	<div><div>1</div><div>2</div><div>3</div><div>1</div></div>	<div><div>2</div><div>3</div><div>2</div><div>2</div></div>	<div><div>3</div><div>3</div><div>3</div><div>3</div></div>	<div><div>3</div><div>4</div><div>4</div><div>3</div></div>			
l		<div><div>3</div><div>3</div></div>	<div><div>3</div><div>2</div><div>4</div><div>2</div></div>	<div><div>2</div><div>3</div><div>3</div><div>2</div></div>	<div><div>3</div><div>4</div><div>3</div><div>3</div></div>	<div><div>4</div><div>4</div><div>4</div><div>4</div></div>			
o		<div><div>4</div><div>4</div></div>	<div><div>4</div><div>3</div><div>5</div><div>3</div></div>	<div><div>3</div><div>3</div><div>4</div><div>3</div></div>	<div><div>2</div><div>4</div><div>4</div><div>2</div></div>	<div><div>4</div><div>5</div><div>3</div><div>3</div></div>			

How do I read out the editing operations that transform OSLO into SNOW?

		s	n	o	w
	<u>0</u>	<u>1 1</u>	<u>2 2</u>	<u>3 3</u>	<u>4 4</u>
o	<u>1 1</u>	<u>1 2</u> <u>2 1</u>	<u>2 3</u> <u>2 2</u>	<u>2 4</u> <u>3 2</u>	<u>4 5</u> <u>3 3</u>
s	<u>2 2</u>	<u>1 2</u> <u>3 1</u>	<u>2 3</u> <u>2 2</u>	<u>3 3</u> <u>3 3</u>	<u>3 4</u> <u>4 3</u>
l	<u>3 3</u>	<u>3 2</u> <u>4 2</u>	<u>2 3</u> <u>3 2</u>	<u>3 4</u> <u>3 3</u>	<u>4 4</u> <u>4 4</u>
o	<u>4 4</u>	<u>4 3</u> <u>5 3</u>	<u>3 3</u> <u>4 3</u>	<u>2 4</u> <u>4 2</u>	<u>4 5</u> <u>3 3</u>

cost	operation	input	output
1	insert	*	w

		s	n	o	w
	<u> </u> 0	<u> 1 </u> 1	<u> 2 </u> 2	<u> 3 </u> 3	<u> 4 </u> 4
o	<u> 1 </u> 1	<u> 1 2 </u> 2 1	<u> 2 3 </u> 2 2	<u> 2 4 </u> 3 2	<u> 4 5 </u> 3 3
s	<u> 2 </u> 2	<u> 1 2 </u> 3 1	<u> 2 3 </u> 2 2	<u> 3 3 </u> 3 3	<u> 3 4 </u> 4 3
l	<u> 3 </u> 3	<u> 3 2 </u> 4 2	<u> 2 3 </u> 3 2	<u> 3 4 </u> 3 3	<u> 4 4 </u> 4 4
o	<u> 4 </u> 4	<u> 4 3 </u> 5 3	<u> 3 3 </u> 4 3	<u> 2 4 </u> 4 2	<u> 4 5 </u> 3 3

cost	operation	input	output
0	(copy)	o	o
1	insert	*	w

		s	n	o	w
	$\begin{array}{ c } \hline 0 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 1 & 1 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 2 & 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 3 & 3 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 4 & 4 \\ \hline \end{array}$
o	$\begin{array}{ c } \hline 1 \\ \hline 1 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 1 & 2 \\ \hline 2 & 1 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 2 & 3 \\ \hline 2 & 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 2 & 4 \\ \hline 3 & 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 4 & 5 \\ \hline 3 & 3 \\ \hline \end{array}$
s	$\begin{array}{ c } \hline 2 \\ \hline 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 1 & 2 \\ \hline 3 & 1 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 2 & 3 \\ \hline 2 & 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 3 & 3 \\ \hline 3 & 3 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 3 & 4 \\ \hline 4 & 3 \\ \hline \end{array}$
l	$\begin{array}{ c } \hline 3 \\ \hline 3 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 3 & 2 \\ \hline 4 & 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 2 & 3 \\ \hline 3 & 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 3 & 4 \\ \hline 3 & 3 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 4 & 4 \\ \hline 4 & 4 \\ \hline \end{array}$
o	$\begin{array}{ c } \hline 4 \\ \hline 4 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 4 & 3 \\ \hline 5 & 3 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 3 & 3 \\ \hline 4 & 3 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 2 & 4 \\ \hline 4 & 2 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 4 & 5 \\ \hline 3 & 3 \\ \hline \end{array}$

cost	operation	input	output
0	(copy)	o	o
1	insert	*	w

		s		n		o		w	
	<u> </u> 0	<u> </u> 1	<u> </u> 1	<u> </u> 2	<u> </u> 2	<u> </u> 3	<u> </u> 3	<u> </u> 4	<u> </u> 4
o	<u> </u> 1 <u> </u> 1	<u> </u> 1 <u> </u> 2	<u> </u> 2 <u> </u> 1	<u> </u> 2 <u> </u> 2	<u> </u> 3 <u> </u> 2	<u> </u> 2 <u> </u> 3	<u> </u> 4 <u> </u> 2	<u> </u> 4 <u> </u> 3	<u> </u> 5 <u> </u> 3
s	<u> </u> 2 <u> </u> 2	<u> </u> 1 <u> </u> 3	<u> </u> 2 <u> </u> 1	<u> </u> 2 <u> </u> 2	<u> </u> 3 <u> </u> 2	<u> </u> 3 <u> </u> 3	<u> </u> 3 <u> </u> 3	<u> </u> 3 <u> </u> 4	<u> </u> 4 <u> </u> 3
l	<u> </u> 3 <u> </u> 3	<u> </u> 3 <u> </u> 4	<u> </u> 2 <u> </u> 2	<u> </u> 2 <u> </u> 3	<u> </u> 3 <u> </u> 2	<u> </u> 3 <u> </u> 3	<u> </u> 4 <u> </u> 3	<u> </u> 4 <u> </u> 4	<u> </u> 4 <u> </u> 4
o	<u> </u> 4 <u> </u> 4	<u> </u> 4 <u> </u> 5	<u> </u> 3 <u> </u> 3	<u> </u> 3 <u> </u> 4	<u> </u> 3 <u> </u> 3	<u> </u> 2 <u> </u> 4	<u> </u> 4 <u> </u> 2	<u> </u> 4 <u> </u> 3	<u> </u> 5 <u> </u> 3

cost	operation	input	output
1	replace	l	n
0	(copy)	o	o
1	insert	*	w

		s		n		o		w	
	<u>0</u>	1	1	2	2	3	3	4	4
o	<u>1</u> 1	1 2	2 1	2 2	3 2	2 3	4 2	4 3	5 3
s	<u>2</u> 2	<u>1</u> <u>2</u> 3 <u>1</u>		2 2	3 2	3 3	3 3	3 4	4 3
l	<u>3</u> 3	3 4	2 2	<u>2</u> <u>3</u> 3 <u>2</u>		3 3	4 3	4 4	4 4
o	<u>4</u> 4	4 5	3 3	3 4	3 3	<u>2</u> <u>4</u> 4 <u>2</u>		<u>4</u> <u>5</u> 3 <u>3</u>	

cost	operation	input	output
0	(copy)	s	s
1	replace	l	n
0	(copy)	o	o
1	insert	*	w

		s	n	o	w
	<u> </u> 0	<u> </u> 1 1	<u> </u> 2 2	<u> </u> 3 3	<u> </u> 4 4
o	<u> </u> 1 1	<u> </u> 1 2 2 1	<u> </u> 2 3 2 2	<u> </u> 2 4 3 2	<u> </u> 4 5 3 3
s	<u> </u> 2 2	<u> </u> 1 2 3 1	<u> </u> 2 3 2 2	<u> </u> 3 3 3 3	<u> </u> 3 4 4 3
l	<u> </u> 3 3	<u> </u> 3 2 4 2	<u> </u> 2 3 3 2	<u> </u> 3 4 3 3	<u> </u> 4 4 4 4
o	<u> </u> 4 4	<u> </u> 4 3 5 3	<u> </u> 3 3 4 3	<u> </u> 2 4 4 2	<u> </u> 4 5 3 3

cost	operation	input	output
1	delete	o	*
0	(copy)	s	s
1	replace	l	n
0	(copy)	o	o
1	insert	*	w

Edit distance to all dictionary terms?

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
 - Expensive and slow
 - Alternative?
- How do we cut the set of candidate dictionary terms?
- One possibility is to use n -gram overlap for this
- This can also be used by itself for spelling correction.

Example with trigrams

- Suppose the text is **november**
 - Trigrams are *nov, ove, vem, emb, mbe, ber.*
- The query is **december**
 - Trigrams are *dec, ece, cem, emb, mbe, ber.*
- So 3 trigrams overlap (of 6 in each term)
- How can we turn this into a normalized measure of overlap?

One option – Jaccard coefficient

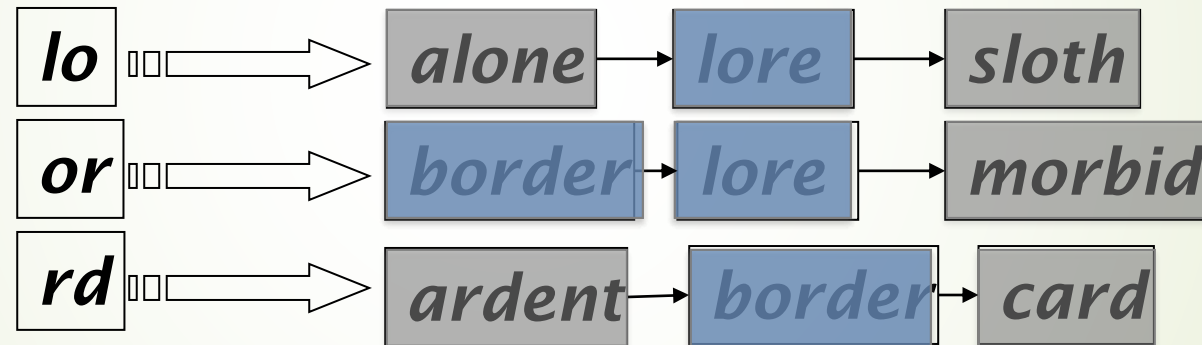
- A commonly-used measure of overlap
- Let X and Y be two sets; then the J.C. is

$$|X \cap Y| / |X \cup Y|$$

- Equals 1 when X and Y have the same elements and zero when they are disjoint
- X and Y don't have to be of the same size
- Always assigns a number between 0 and 1
 - Now threshold to decide if you have a match
 - E.g., if J.C. > 0.8, declare a match

Matching trigrams

- Consider the query **lord** – we wish to identify words matching 2 of its 3 bigrams (**lo**, **or**, **rd**)



Standard postings “merge” will enumerate ...

Adapt this to using Jaccard (or another) measure.

Context-sensitive spell correction

- Text: *I flew from Heathrow to Narita.*
- Consider the phrase query “*flew form Heathrow*”
- We’d like to respond

Did you mean “*flew from Heathrow*”?

because no docs matched the query phrase.

Context-sensitive correction

- Need surrounding context to catch this.
- First idea: retrieve dictionary terms close (in weighted edit distance) to each query term
- Now try all possible resulting phrases with one word “fixed” at a time
 - *flew from heathrow*
 - *fled form heathrow*
 - *flea form heathrow*
- **Hit-based spelling correction:** Suggest the alternative that has lots of hits.

Context-sensitive spell correction

Exercise:

- Suppose that for “**flew** form **Heathrow**” we have 7 alternatives for flew, 19 for form and 3 for heathrow.

How many “corrected” phrases will we enumerate in this scheme?

General issues in spell correction

- We enumerate multiple alternatives for “Did you mean?”
- Need to figure out which to present to the user
 - The alternative hitting most docs
 - Query log analysis

Soundex

- Soundex is the basis for finding **phonetic** alternatives.
- Example: *chebyshev* / *tchebyscheff*
- Algorithm:
 - Turn every token to be indexed into a 4-character reduced form
 - Do the same with query terms
 - Build and search an index on the reduced forms

Soundex algorithm

1. Retain the first letter of the term.
2. Change all occurrences of the following letters to '0' (zero): A, E, I, O, U, H, W, Y
3. Change letters to digits as follows:
 - B, F, P, V to 1
 - C, G, J, K, Q, S, X, Z to 2
 - D, T to 3
 - L to 4
 - M, N to 5
 - R to 6
4. Repeatedly remove one out of each pair of consecutive identical digits
5. Remove all zeros from the resulting string;
6. pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits

Example: Soundex of *HERMAN*

- Retain H
- *ERMAN* → *ORMON*
- *ORMON* → 06505
- 06505 → 06505
- 06505 → 655
- Return *H655*
- Note: *HERMANN* will generate the same code

How useful is Soundex?

- ▶ Not very – for information retrieval
- ▶ Ok for “high recall” tasks in other applications (e.g., Interpol)
- ▶ Zobel and Dart (1996) suggest better alternatives for phonetic matching in IR.

Exercise:

Beijing/ Peking