Introduction to

Information Retrieval

Dictionaries and tolerant retrieval

Recap of the previous lecture

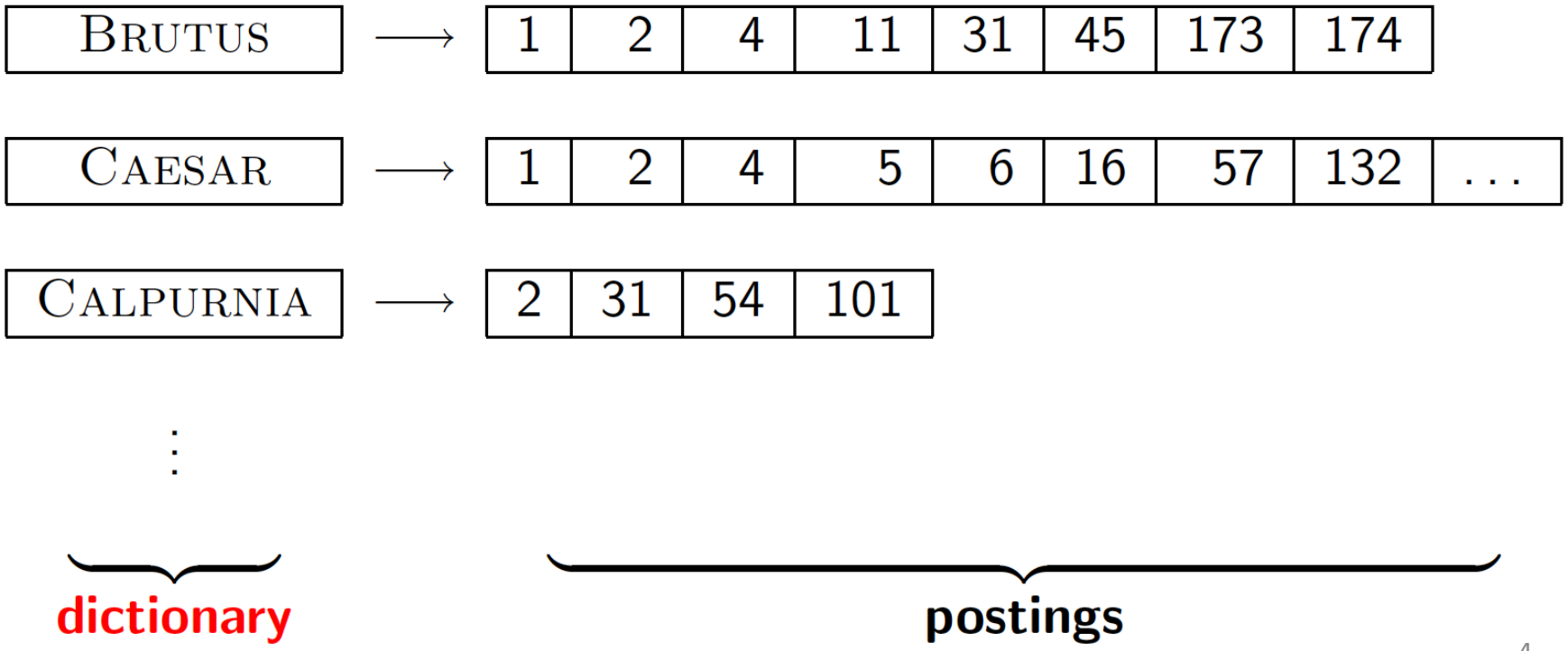
- The type/token distinction
  - Terms are normalized types put in the dictionary
- Tokenization problems:
  - Hyphens, apostrophes, compounds, CJK
- Term equivalence classing:
  - Numbers, case folding, stemming, lemmatization
- Skip pointers
  - Encoding a tree-like structure in a postings list
- Biword indexes for phrases
- Positional indexes for phrases/proximity queries

This lecture

- Dictionary data structures
- "Tolerant" (willingness to allow) retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?



A naïve dictionary

- An array of struct:

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
. . .	. . .	. . .
zulu	221	⟶

char[20]	int	Postings *
20 bytes	4/8 bytes	4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

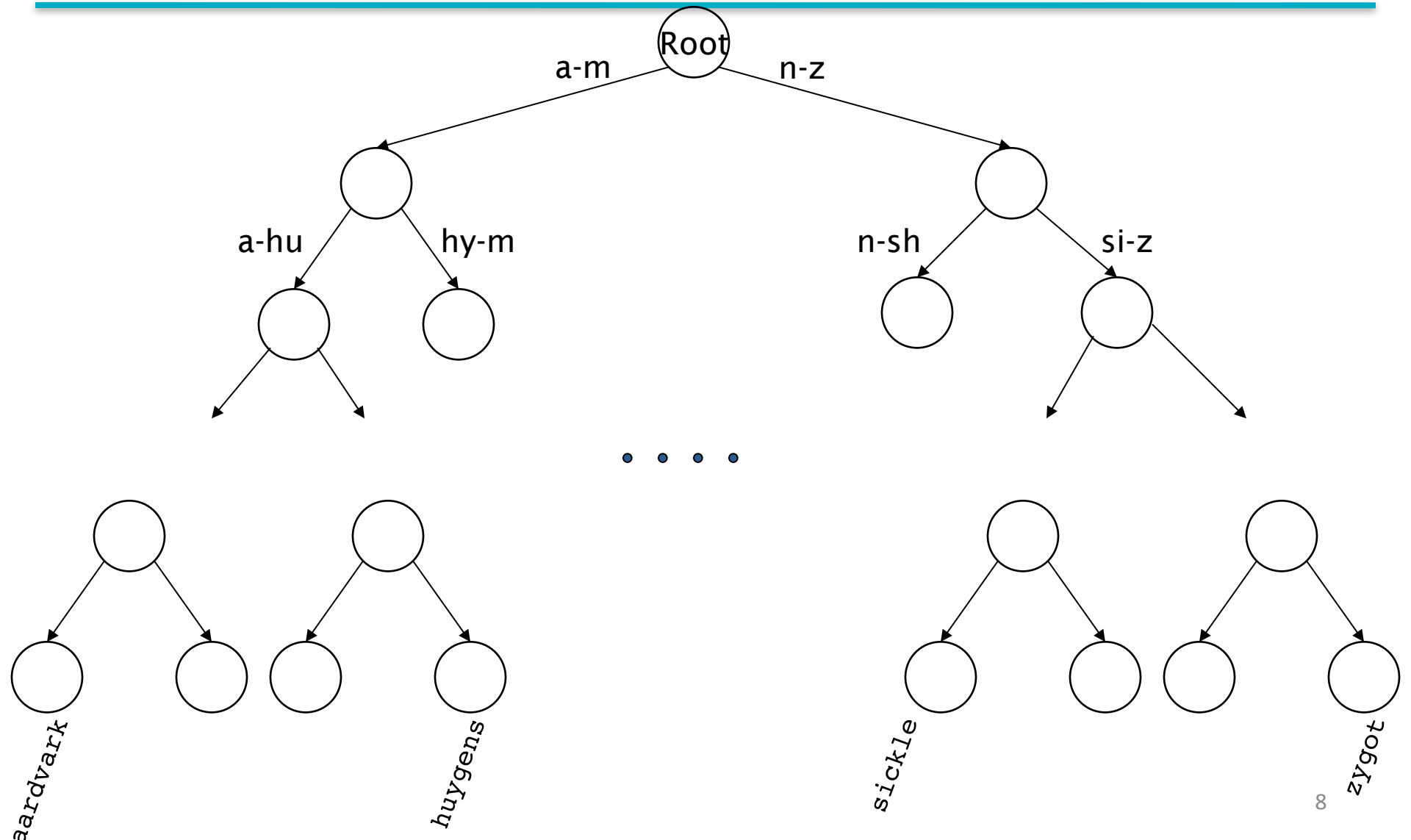
Dictionary data structures

- Two main choices:
  - Hashtables
  - Trees
- Some IR systems use hashtables, some trees

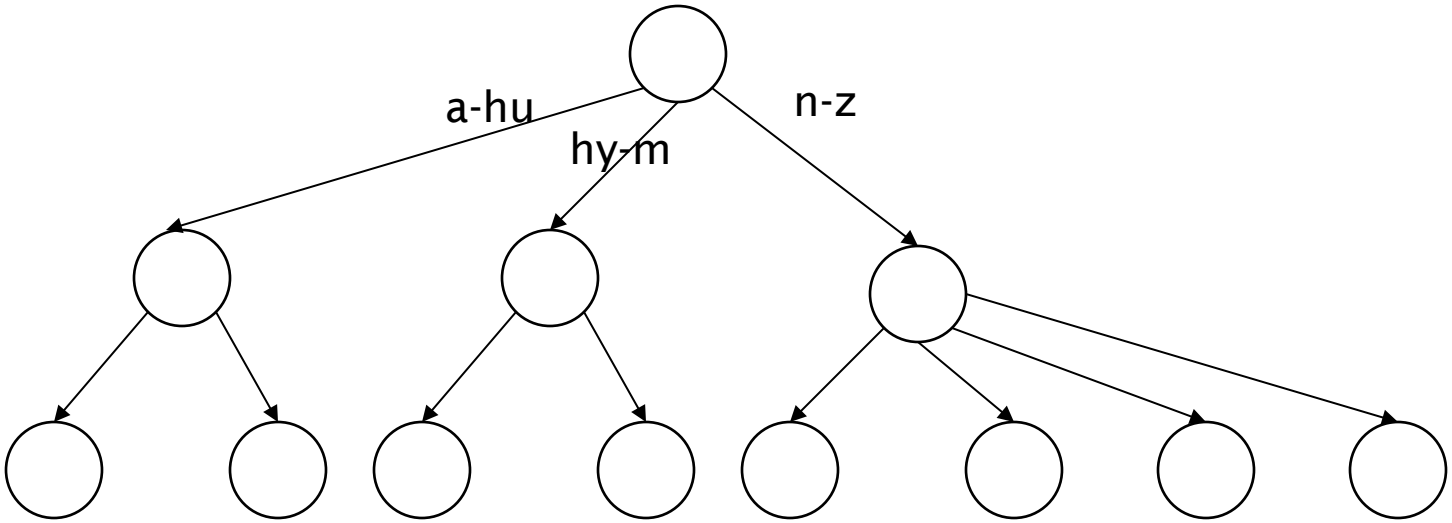
Hashtables

- Each vocabulary term is hashed to an integer (key)
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster due to hash function than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - Accented and non-accented versions of a word like *resume*
  - No prefix search like *automat*
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Tree: binary tree



Tree: B-tree



- Definition: Every internal nodel 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 … Chinese don't have ordering but now it is forced.
- Pros:
  - Solves the prefix problem
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate (less severe) the rebalancing problem

WILD-CARD QUERIES

Wild-card queries: *

- ***mon*:*** find all docs containing any word beginning with "mon".
  - Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo***
- ****mon:*** find words ending in "mon": harder
  - Maintain an additional B-tree for terms *backwards.*

  Can retrieve all words in range: ***nom ≤ w < non***.

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.

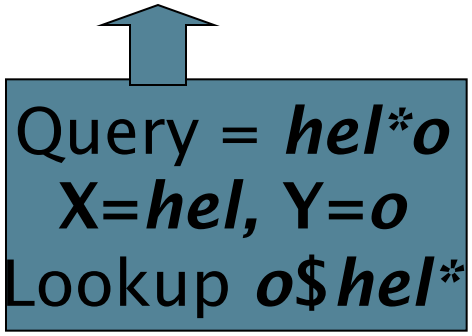
B-trees handle *'s at the end of a query term

- 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***, index under:
  - ***hello$, ello$h, llohe, lohel, 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!



Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- Lookup in standard inverted index to find the docs
- *Permuterm problem: ≈ quadruples lexicon size*

Empirical observation for English.

Bigram (*k*-gram) indexes

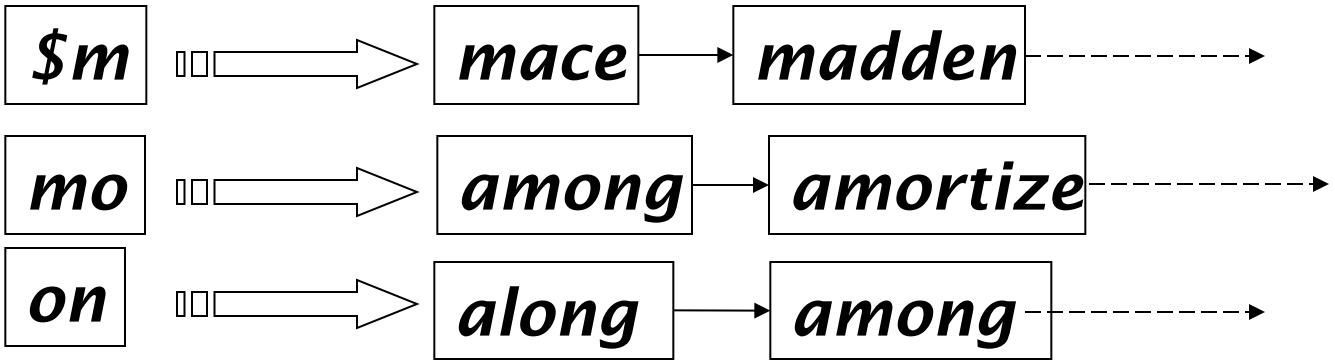
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.,* from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

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).

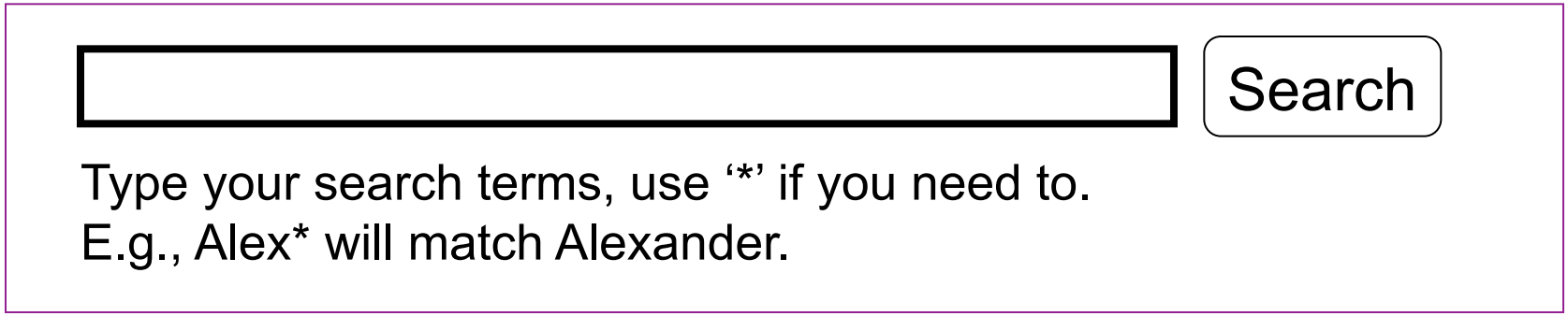


Processing wild-cards

- Query ***mon**** can now be run as
  - ***$m*** *AND* ***mo*** *AND* ***on***
- Gets terms that match AND version of our wildcard query.
- But we'd enumerate ***moon***.
- Must post-filter these terms against query.
- Surviving enumerated terms are then looked up in the term-document inverted index.
- Fast, space efficient (compared to permuterm).

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!



- Which web search engines allow wildcard queries?

Which web search engines allow wildcard queries?

- * supported by AltaVista, Inktomi (iWon), Northern Light, Yahoo
- ? supported by AOL Search, Inktomi (iWon)
- % supported by Northern Light none AllTheWeb, Direct Hit, Excite, Google, HotBot, LookSmart, Lycos, MSN

- For details: http://answers.google.com/answers/threadview/id/38353.html

Class Exercise

- **Exercise 3.2** Write down the entries in the permuterm index dictionary that are generated by the term "mama".

- **Solution:**

mama$
ama$m
ma$ma
a$mam
$mama

Class Exercise

- **Exercise 3.3**
  If you wanted to search for s*ng in a permuterm wildcard index, what key(s) would one do the lookup on?

- **Solution:** ng$s*

Homework #3

- **A. Exercise 3.4:** Refer to Figure 3.4; it is pointed out in the caption that the vocabulary terms in the postings are lexicographically ordered. Why is this ordering useful?

- **B. Exercise 3.5:** Consider again the query fi*mo*er from Section 3.2.1. What Boolean query on a bigram index would be generated for this query? Can you think of a term that matches the permuterm query in Section 3.2.1, but does not satisfy this Boolean query?

- **C. Exercise 3.6:** Give an example of a sentence that falsely matches the wildcard query mon*h if the search were to simply use a conjunction of bigrams.

SPELLING CORRECTION

Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve "right" answers
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from → form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew form Heathrow to Narita.***

Document correction

- Especially needed for OCR'ed documents
  - Correction algorithms are tuned for this: rn/m
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).
- But also: web pages and even printed material have typos
- Goal: the dictionary contains fewer misspellings
- But often we don't change the documents and instead fix the query-document mapping

Query mis-spellings

- Our principal focus here is on the query.
- We can either
  - Retrieve documents indexed by the correct spelling,
  - OR return several suggested alternative queries with the correct spelling. For example: *Did you mean … ?*

Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
  - A standard lexicon such as
    - Webster's English Dictionary
    - https://github.com/matthewreagan/WebstersEnglishDictionary
  - 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
- 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.

Edit distance

```
EDITDISTANCE(s1, s2)
 1  int m[i, j] = 0
 2  for i ← 1 to |s1|
 3  do m[i, 0] = i
 4  for j ← 1 to |s2|
 5  do m[0, j] = j
 6  for i ← 1 to |s1|
 7  do for j ← 1 to |s2|
 8       do m[i, j] = min{m[i − 1, j − 1] + if (s1[i] = s2[j]) then 0 else 1fi,
 9                        m[i − 1, j] + 1,
10                        m[i, j − 1] + 1}
11  return m[|s1|, |s2|]
```

Edit distance

N	9	8	9	10	11	12	11	10	9	**8**
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	**0**	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

- Further details can be seen in NLP-course portal on the instructor's page.

Edit distance

		s	n	o	w
	0	**1** **1**	**2** **2**	**3** **3**	**4** **4**
o	**1** / **1**	**1** 2 / 2 **1**	**2** 3 / **2** **2**	**2** 4 / 3 **2**	4 5 / **3** **3**
s	**2** / **2**	**1** 2 / 3 **1**	**2** 3 / **2** **2**	**3** **3** / **3** **3**	**3** 4 / 4 **3**
l	**3** / **3**	3 **2** / 4 **2**	**2** 3 / 3 **2**	**3** 4 / **3** **3**	**4** **4** / **4** **4**
o	**4** / **4**	4 **3** / 5 **3**	**3** **3** / 4 **3**	**2** 4 / 4 **2**	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

Weighted edit distance

- 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 (next slide)
- Modify dynamic programming to handle weights
- Can see a variant here: https://ideone.com/RblFK

Weighted edit distance

sub[X, Y] = Substitution of X (incorrect) for Y (correct)

X \ Y (correct)	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0	0	7	1	342	0	0	2	118	0	1	0	0	3	76	0	0	1	35	9	9	0	1	0	5	0
b	0	0	9	9	2	2	3	1	0	0	0	5	11	5	0	10	0	0	2	1	0	0	8	0	0	0
c	6	5	0	16	0	9	5	0	0	0	1	0	7	9	1	10	2	5	39	40	1	3	7	1	1	0
d	1	10	13	0	12	0	5	5	0	0	2	3	7	3	0	1	0	43	30	22	0	0	4	0	2	0
e	388	0	3	11	0	2	2	0	89	0	0	3	0	5	93	0	0	14	12	6	15	0	1	0	18	0
f	0	15	0	3	1	0	5	2	0	0	0	3	4	1	0	0	0	6	4	12	0	0	2	0	0	0
g	4	1	11	11	9	2	0	0	0	1	1	3	0	0	2	1	3	5	13	21	0	0	1	0	3	0
h	1	8	0	3	0	0	0	0	0	0	2	0	12	14	2	3	0	3	1	11	0	0	2	0	0	0
i	103	0	0	0	146	0	1	0	0	0	0	6	0	0	49	0	0	0	2	1	47	0	2	1	15	0
j	0	1	1	9	0	0	1	0	0	0	0	2	1	0	0	0	0	0	5	0	0	0	0	0	0	0
k	1	2	8	4	1	1	2	5	0	0	0	0	5	0	2	0	0	0	6	0	0	0	4	0	0	3
l	2	10	1	4	0	4	5	6	13	0	1	0	0	14	2	5	0	11	10	2	0	0	0	0	0	0
m	1	3	7	8	0	2	0	6	0	0	4	4	0	180	0	6	0	0	9	15	13	3	2	2	3	0
n	2	7	6	5	3	0	1	19	1	0	4	35	78	0	0	7	0	28	5	7	0	0	1	2	0	2
o	91	1	1	3	116	0	0	0	25	0	2	0	0	0	0	14	0	2	4	14	39	0	0	0	18	0
p	0	11	1	2	0	6	5	0	2	9	0	2	7	6	15	0	0	1	3	6	0	4	1	0	0	0
q	0	0	1	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	14	0	30	12	2	2	8	2	0	5	8	4	20	1	14	0	0	12	22	4	0	0	1	0	0
s	11	8	27	33	35	4	0	1	0	1	0	27	0	6	1	7	0	14	0	15	0	0	5	3	20	1
t	3	4	9	42	7	5	19	5	0	1	0	14	9	5	5	6	0	11	37	0	0	2	19	0	7	6
u	20	0	0	0	44	0	0	0	64	0	0	0	0	2	43	0	0	4	0	0	0	0	2	0	8	0
v	0	0	7	0	0	3	0	0	0	0	0	1	0	0	1	0	0	0	8	3	0	0	0	0	0	0
w	2	2	1	0	1	0	0	2	0	0	1	0	0	0	0	7	0	6	3	3	1	0	0	0	0	0
x	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0
y	0	0	2	0	15	0	1	7	15	0	0	0	2	0	6	1	0	7	36	8	5	0	0	1	0	0
z	0	0	0	7	0	0	0	0	0	0	0	7	5	0	0	0	0	2	21	3	0	0	0	0	3	0

Using edit distances

- Given query, first enumerate all character sequences within a preset (possibly weighted) edit distance with some threshold.
- Intersect this set with our list of "correct" words (lexicon)
- Then suggest terms in the intersection to the user.
- 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

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.

n-gram overlap

- Enumerate all the *n*-grams in the query string as well as in the lexicon
- Use the *n*-gram index (recall wild-card search) to retrieve all lexicon terms matching any of the query *n*-grams
- Threshold by number of matching *n*-grams (explain next)
  - Variants – weight by keyboard layout, etc.

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 (out 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

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

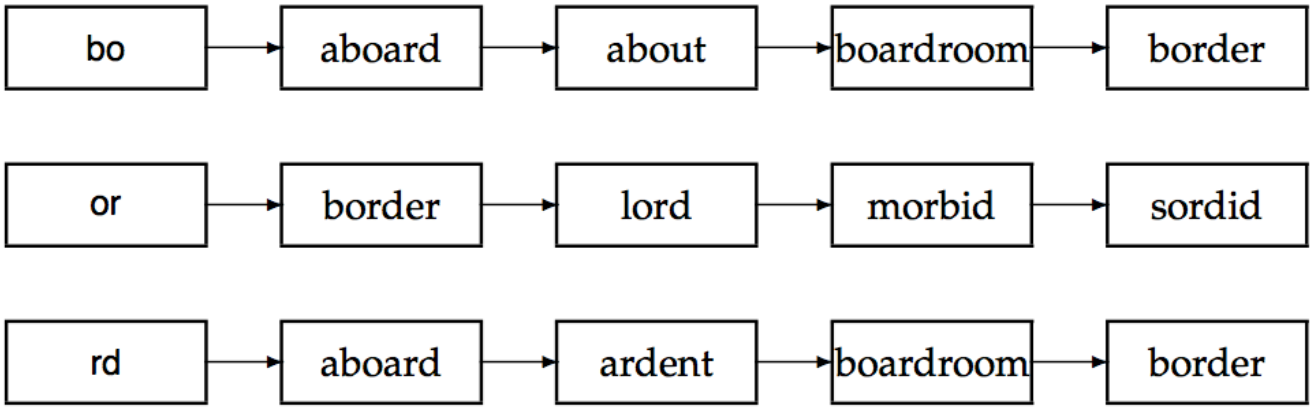
- Equals 1 when *X* and *Y* have the same elements and zero when they are disjoint

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}. \qquad 0 \leq J(A,B) \leq 1.$$

- Now threshold to decide if you have a match
  - E.g., if J.C. > 0.6, declare a match

Matching bigrams

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



- bo AND or: 1/5+3-1= 0.14
- bo AND rd: 3/5+3-3 = 0.6

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 ***"flew form Heathrow"***

- 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.

Another approach

- Break phrase query ***"flew form Heathrow"*** into a conjunction of biwords.
- Look for biwords that need only one term corrected.
- Enumerate only phrases containing "common" biwords.

- As an alternative to using the biword statistics in the collection, we may use the logs of queries issued by users; these could of course include queries with spelling errors.

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
- More generally, rank alternatives probabilistically

$$\text{argmax}_{corr}\ P(corr \mid query)$$

  - From Bayes rule, this is equivalent to

$$\text{argmax}_{corr}\ P(query \mid corr) * P(corr)$$

Noisy channel Language model

Homework #3

- **D. Exercise 3.8:** Compute the edit distance between *paris* and *alice*. Write down the 5 × 5 array of distances between all prefixes as computed by the algorithm in Figure 3.5.
- **E. Exercise 3.11:** Consider the four-term query catched in the rye and suppose that each of the query terms has five alternative terms suggested by isolated-term correction. How many possible corrected phrases must we consider if we do not trim the space of corrected phrases, but instead try all six variants for each of the terms?

SOUNDEX

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** (English) → ***tchebycheff*** (Welsh)
- Invented for the U.S. census … in 1918

Soundex – typical 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
  - (when the query calls for a soundex match)

- http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top

Soundex – typical algorithm

1. Retain the first letter of the word.
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 → 1
   - C, G, J, K, Q, S, X, Z → 2
   - D,T → 3
   - L → 4
   - M, N → 5
   - R → 6

Soundex continued

4. Remove all pairs 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 be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

Will ***hermann*** generate the same code?

Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, …)
- How useful is soundex?
  - Not very – for information retrieval
- Okay for "high recall" tasks (e.g., Interpol).

- Zobel and Dart (1996) show that other algorithms for phonetic matching perform much better in the context of IR

Articles to be read

- J. Zobel and P. Dart. Finding approximate matches in large lexicons. Software - practice and experience 25(3), March 1995.
- Ferragina, Paolo, and Rossano Venturini. 2007. Compressed permuterm indexes. In *Proc. SIGIR*. ACM Press.
- Toutanova, Kristina, and Robert C. Moore. 2002. Pronunciation modeling for improved spelling correction. In *Proc. ACL*, pp. 144–151.
- Cucerzan, Silviu, and Eric Brill. 2004. Spelling correction as an iterative process that exploits the collective knowledge of web users. In *Proc. Empirical Methods in Natural Language Processing*.

Homework #3

- **F:** Visit the following links for spell correction at http://norvig.com/spell-correct.html , Try to understand, code and configure the spell corrector. Finally submit the reports of your understanding along with the screen shots.

- **G:** Visit the following link for Soundex spell correction at http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top , Try to understand, code and configure the spell corrector. Finally submit the reports of your understanding along with the screen shots.

- **Note:** Prints are only accepted for programming assignments. Rest of the assignments should be hand written.