Dictionary data structures for the Inverted Index

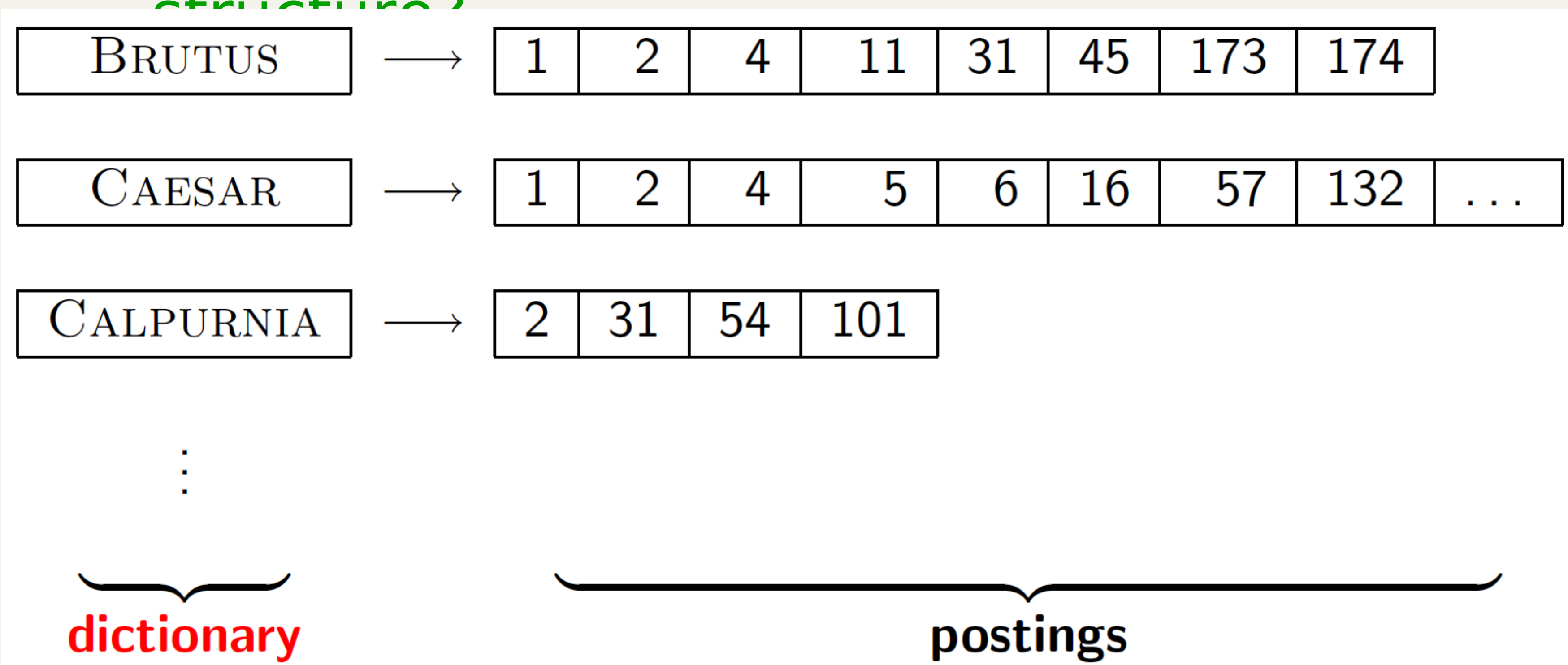
This lecture

- Dictionary data structures
  - Exact search
  - Prefix search

- “Tolerant” retrieval
  - Edit-distance queries
  - Wild-card queries
  - Spelling correction
  - Soundex

Basics

- The dictionary data structure stores the term vocabulary, but… in what data structure?



A naïve dictionary

- An array of struct:

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

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?

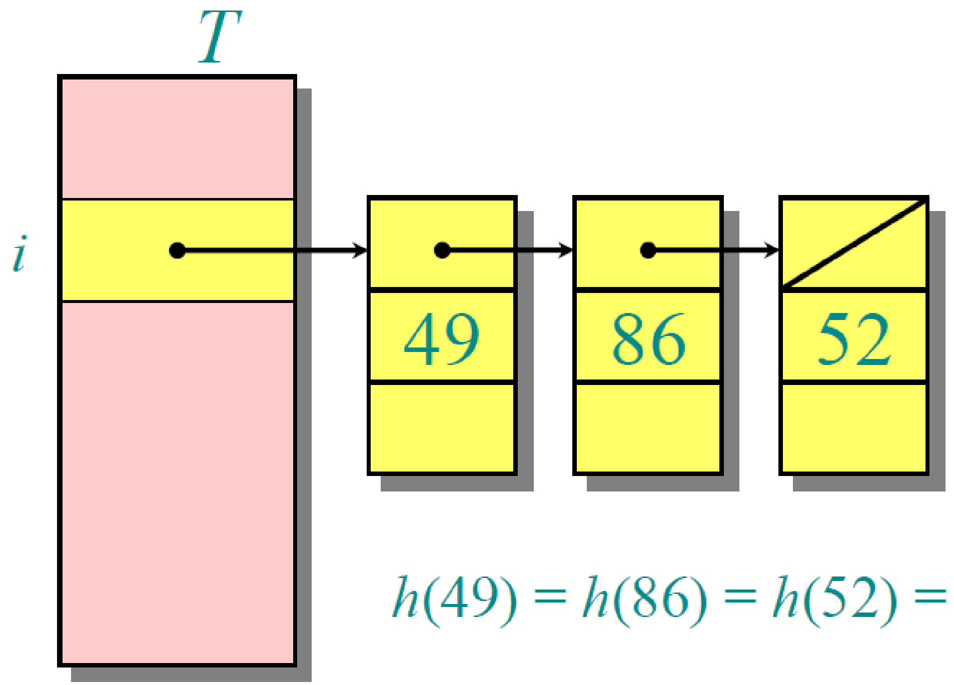
Dictionary data structures

- Two main choices:
  - Hash table
  - Tree
  - Trie

- Some IR systems use hashes, some trees/tries

Hashing with chaining

- Link records in the same slot into a list.



$$h(49) = h(86) = h(52) = i$$

The current version is **MurmurHash** (vers 3) yields a 32-bit or 128-bit hash value.

Prefix search

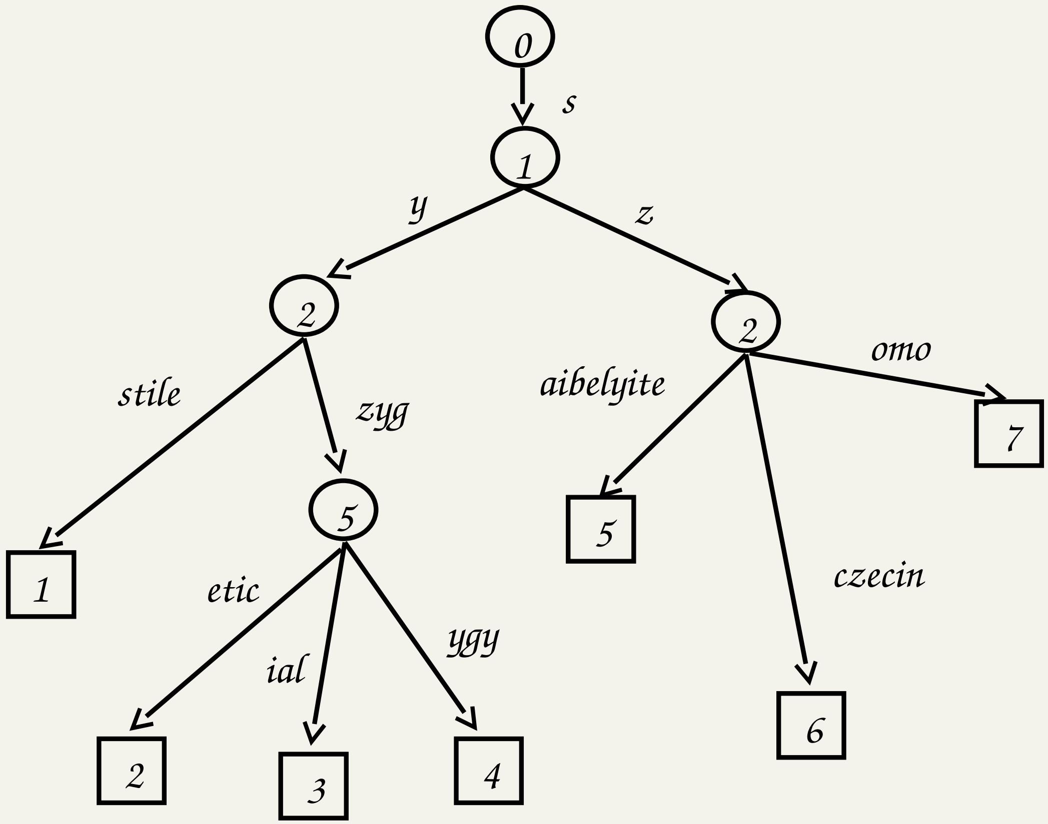
Prefix-string Search

Given a dictionary D of K strings, of total length N, store them in a way that we can efficiently support prefix searches for a pattern P over them.

Ex. Pattern P is **pa**

Dict = {abaco, box, paolo, patrizio, pippo, zoo}

Trie: speeding-up searches



Pro: O(p) search time = path scan

Cons: edge + node labels + tree structure

Tries

Do exist many variants and their implementations

- Pros:
  - Solves the prefix problem
- Cons:
  - Slower: O(p) time, many cache misses
  - From 10 to 60 (or, even more) bytes per node

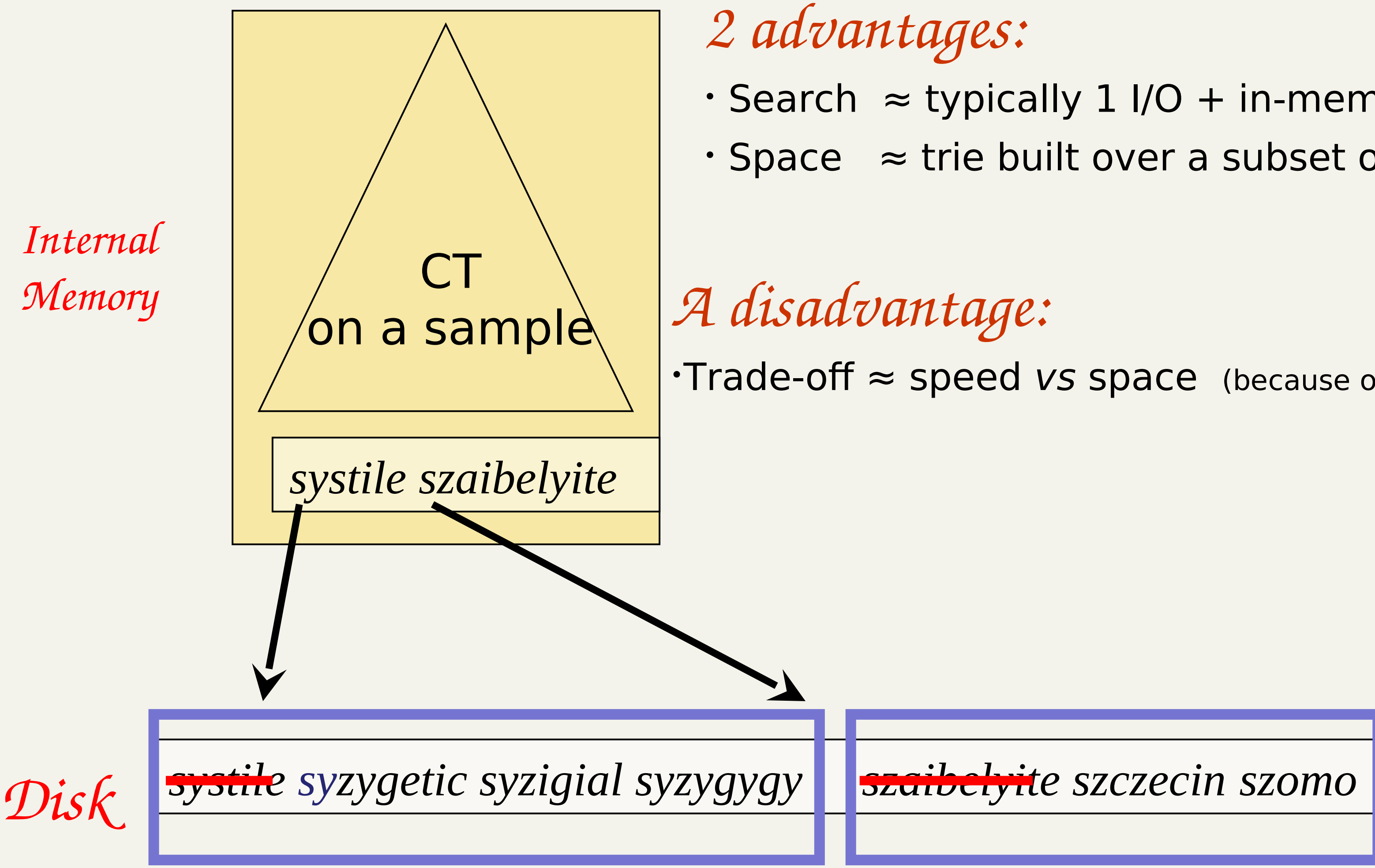
2-level indexing

2 advantages:

- Search ≈ typically 1 I/O + in-mem comp.
- Space ≈ trie built over a subset of strings

A disadvantage:

- Trade-off ≈ speed *vs* space (because of bucket size)



Front-coding: squeezing dict

systile syzygetic syzygial syzygy

2 5 5

http://checkmate.com/All_Natural/
http://checkmate.com/All_Natural/Applied.html
http://checkmate.com/All_Natural/Aroma.html
http://checkmate.com/All_Natural/Aroma1.html
http://checkmate.com/All_Natural/Aromatic_Art.html
http://checkmate.com/All_Natural/Ayate.html
http://checkmate.com/All_Natural/Ayer_Soap.html
http://checkmate.com/All_Natural/Ayurvedic_Soap.html
http://checkmate.com/All_Natural/Bath_Salt_Bulk.html
http://checkmate.com/All_Natural/Bath_Salts.html
http://checkmate.com/All/Essence_Oils.html
http://checkmate.com/All/Mineral_Bath_Crystals.html
http://checkmate.com/All/Mineral_Bath_Salt.html
http://checkmate.com/All/Mineral_Cream.html

http://checkmate.com/All/Natural/Washcloth.html
...

33÷45%

0 http://checkmate.com/All_Natural/
33 Applied.html
34 roma.html
38 1.html
38 tic_Art.html
34 yate.html
35 er_Soap.html
35 urvedic_Soap.html
33 Bath_Salt_Bulk.html
42 s.html
25 Essence_Oils.html
25 Mineral_Bath_Crystals.html
38 Salt.html
33 Cream.html

0 http://checkmate.com/All/Natural/Washcloth.html...

Gzip may be much better...

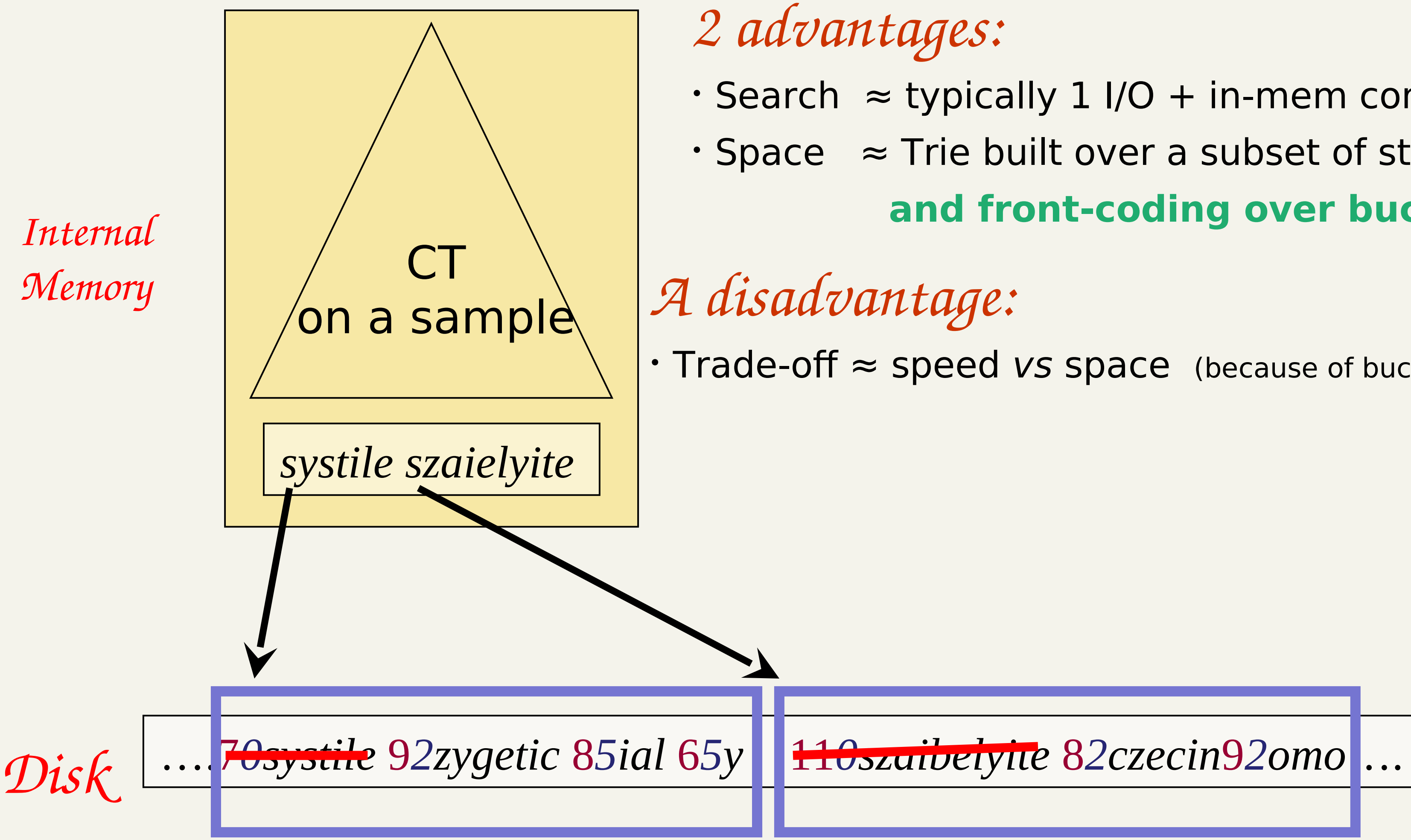
2-level indexing

2 advantages:

- Search ≈ typically 1 I/O + in-mem comp.
- Space ≈ Trie built over a subset of strings **and front-coding over buckets**

A disadvantage:

- Trade-off ≈ speed *vs* space (because of bucket size)



Spelling correction

Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting queries to retrieve “right” answers
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - But what about: ***from*** → ***form***
  - Context-sensitive is more effective
    - Look at surrounding words
    - e.g., ***I flew <u>form</u> Heathrow to Narita.***

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
    - 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)
    - Mining algorithms to derive the possible corrections

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 measures
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - *n*-gram overlap

Brute-force check of ED

Given query Q,

- 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

How the time complexity grows
with #errors allowed and string length ?

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 (possibly, 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.**

DynProg for Edit Distance

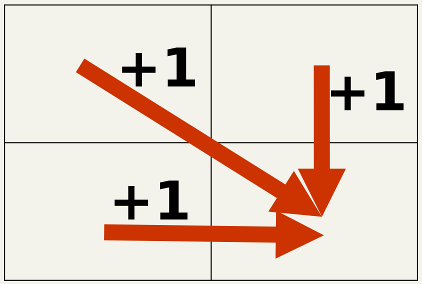
- Let $E(i,j)$ = edit distance to transform $S_1[1,i]$ in $S_2[1,j]$
- Example: **cat** *versus* **dea**
- Consider the sequence of ops: ins, del, subst, *match*
  - Model the edit distance as an alignment problem where **insertion** in S_2 correspond to a **–** in S_1 whereas **deletion** from S_1 correspond to a **–** in S_2.
  - If $S_1[i] = S_2[j]$ then last op is a **match**, and thus it is not counted
  - Otherwise the last op is: **subst**(S_1[i], S_2[j]) or **ins**(S_2[j]) or **del**(S_1[i])

$$E(i,0)=i,\ E(0,j)=j$$

$$E(i,j) = E(i-1, j-1) \qquad \text{if } S_1[i]= S_2[j]$$

$$\begin{aligned} E(i,j) = 1 + \min\{&E(i, j-1), \\ &E(i-1, j), \\ &E(i-1, j-1)\} \qquad \text{if } S_1[i] \neq S_2[j] \end{aligned}$$

Example



			S_2						
			0	1	2	3	4	5	6
				p	t	t	a	p	a
	0		0	1	2	3	4	5	6
	1	p	1	**0**	**1**	**2**	3	4	5
S_1	2	a	2	1	1	2	**2**	3	4
	3	t	3	2	1	1	2	**3**	4
	4	t	4	3	2	1	2	3	**4**

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture keyboard errors, e.g. ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a ***smaller cost*** than by ***q***

- Requires weighted matrix as input
- Modify DP to handle weights

One-error correction

The problem

- 1-error = insertion, deletion or substitution of **one single character**

  Farragina → Ferragina (substitution)

  Feragina → Ferragina (insertion)

  Ferrragina → Ferragina (deletion)

- A string of length L over A chars → #variants = L (A-1) + (L+1)A + L = A * (2L+1)
- You could have many candidates
- You can still deploy keyb… statistical information (w…

Do we need to make the enumeration?

A possible approach

- Create two dictionaries:
  - D1 = { strings }
  - D2 = { strings of D1 with *one deletion* }

D1 = {cat, cast, cst, dag}

D2 = {

- ag [→ dag], da [dag], dg [dag],
- ast [→cast], cas [→cast], cat [→cast], cst [→cast]
- cs [cst], ct [cst], st [cst]}
- at [cat], ca [cat], ct [cat]

}

Assume a fast string-check in a dictionary

An example

D1 = {cat, cast, cst, dag}

D2 = { ag [dag], ast [cast], at [cat], ca [cat], cas [cast], cat [cast], cs [cst], cst [cast], ct [cat; cst], da [dag], dg [dag], st [cst] }

- Query(«cat»):
  - Perfect match: Search(P) in D1, *1 query [→* cat*]*
  - P 1-char less: Search(P) in D2, *1 query [→* cat [cast]]
  - P 1-char more: Search(P $_{-1 \text{ char}}$) in D1, *p queries [search for {at, ct, ca} in D1 →* No match]
  - Substitution: Search(P $_{-1 \text{ char}}$) in D2, *p queries [e.g. {at,ct, ca} in D2 →* at [cat], ct [cat], ct [cst], ca [cat]]

A possible approach

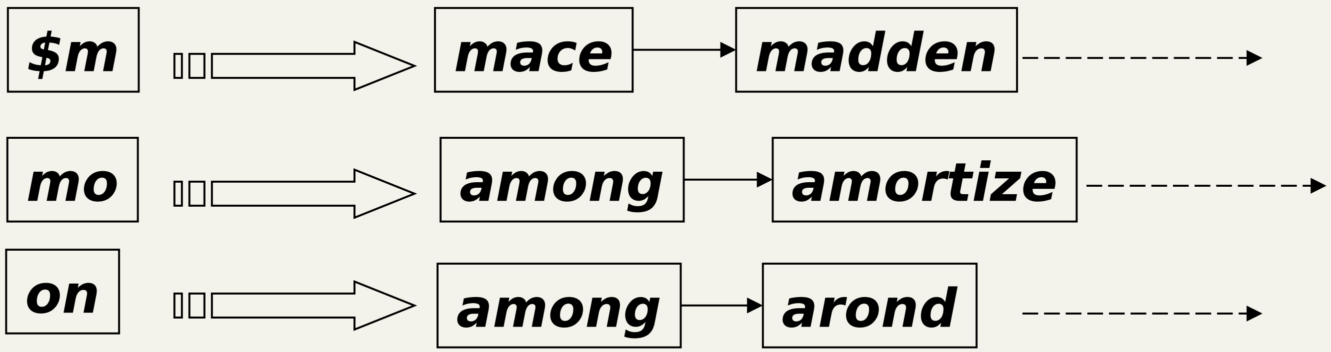
- Query(P):
  - Perfect match: Search for P in D1, *1 query*
  - P 1-char less: Search for P in D2 , *1 query*
  - P 1-char more: Search for P $_{-1\ \text{char}}$ in D1 , *p queries*
  - Substitution: Search for P $_{-1\ \text{char}}$ in D2 , *p queries*

- We need 2p + 2 hash computations for P
  - **Pro**: CPU efficient, no cache misses for computing P's hashes; but O(p) cache misses to search in D1 and D2
  - **Cons**: Large space because of the many strings in D2 which must be stored to search in the hash table of D2, unless we avoid collision (perfect hash)
  - **FALSE MATCHES:** **ctw** matches **cat** and **cst** as SUBST.

Overlap vs Edit distances

K-gram index *(useful for >1 errors)*

The *k*-gram index contains for every *k*-gram (here *k*=2) all terms including that k-gram.

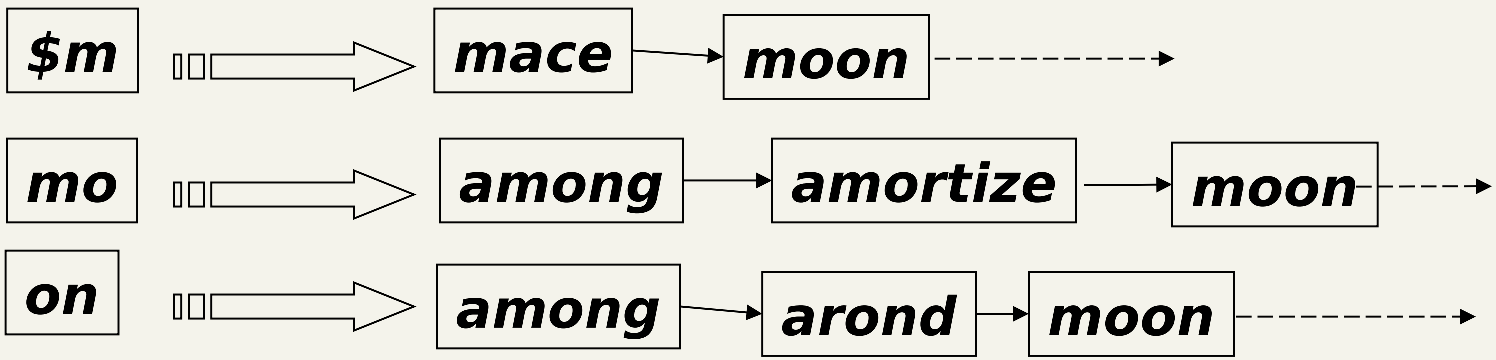


Append k-1 symbol $ at the front of each string, in order to generate a number L of

k-grams for a string of length L.

Overlap Distance

Enumerate all *2*-grams in Q = mon ($m, mo, on)



Use the *2*-gram index to rank all lexicon terms according to the **number of matching** *2*-grams (moon)

Overlap Distance *versus* Edit Distance

Select terms by **threshold on matching** *k*-grams

- If the term is **L** chars long (it consists of L k-grams)
- If **E** is the number of allowed errors (E*k of the k-grams of Q might be different from term's ones because of the E errors)
- So at least L – E*k of the k-grams in Q must match a dictionary term to be a candidate answer

If ED is required, post-filter results dyn. prog.

Necessary but not sufficient condition !!

T=$mom, Q=$omo, L-E*k=3-1*2=1, T∩Q = {mo,om} but EditDist = 2

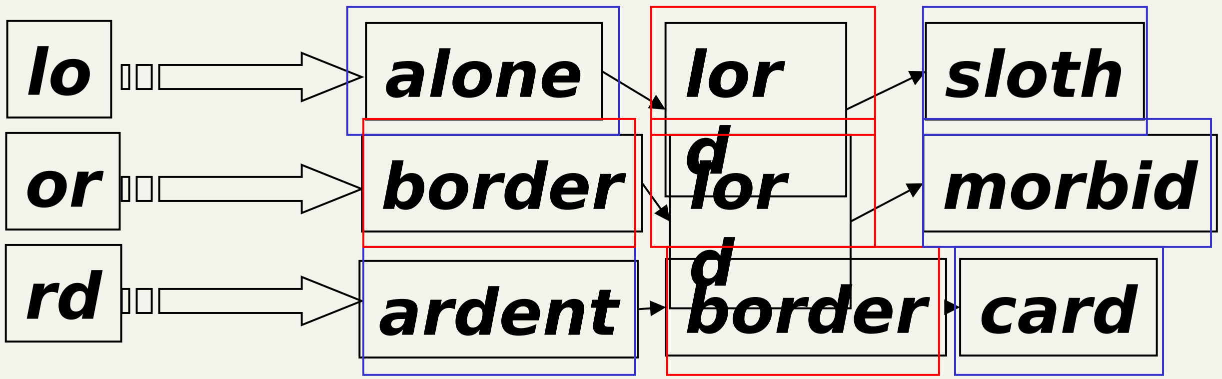
Example with trigrams (K=3)

- Suppose the term to compare is $$***november***
  - Trigrams are *$$n, $no, nov, ove, vem, emb, mbe, ber*
- The query **Q =** $$***december***
  - Trigrams are *$$d, $de, dec, ece, cem, emb, mbe, ber*
- |Q|=8, K=3, if E=1 → L – E*k = 8 – 1*3 = 5
  NO!
- |Q|=8, K=3, if E=2 → L – E*k = 8 – 2*3 = 2
  OK!

Post-filter is needed to check that the distance > 2

Fast searching for k-grams

- Assume we wish to match the following bi-grams (***lo, or, rd***) against a **2-gram index** built over a dictionary of terms.



Standard postings "merge" will enumerate terms with their cardinality

Then choose terms if #occ possibly filter out < L – k*E

Context-sensitive spell correction

Text: ***I flew <u>from</u> Heathrow to Narita.***

- Consider the phrase query ***"flew <u>form</u> Heathrow"***
- We'd like to respond

  Did you mean "***flew from Heathrow***"?

because no docs matched the query phrase.

General issues in spell correction

- We enumerate multiple alternatives and then need to figure out which to present to the user for “Did you mean?”
- Use heuristics
  - The alternative hitting most docs
  - Query log analysis + tweaking
    - For especially popular, topical queries
- Spell-correction is computationally expensive
  - Run only on queries that matched few docs

Other sophisticated queries

Wild-card queries: *

- ***mon*:*** find all docs containing words beginning with “mon”.
  - Use a Prefix-search data structure

- ***mon:*** find words ending in “mon”
  - Maintain a prefix-search data structure for *reverse* terms.

How can we solve the wild-card query ***pro*cent*** ?

What about * in the middle?

- ***co*tion***

  We could look up ***co**** AND ***tion** and intersect the two lists *(expensive)*

- ***se*ate*** *AND* ***fil*er***

  This may result in many Boolean *ANDs*.

- 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
  - P*Q → Q$P*
- Now use prefix-search data structure
- *Permuterm problem: ≈ 4x lexicon size*

Empirical observation for English.

Soundex

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** → ***tchebycheff***

- Invented for the U.S. census in 1918

Soundex – typical algorithm

- Turn every token to be indexed into a reduced form consisting of 4 chars
- Do the same with query terms
- Build and search an index on the reduced forms

Soundex – typical algorithm

1. Retain the first letter of the word.
   - ***Herman → H…***
2. Change all occurrences of the following letters to '0' (zero): 'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
   - ***Herman →*** H0rm0n
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

H0rm0n → H06505

Soundex contd

4. Remove all pairs of consecutive equal digits.

H06505 → H06505

5. Remove all zeros from the resulting string.

H06505 → H655

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., ***Hermann*** also becomes H655.

Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, …)
- Not very useful – for information retrieval
- Okay for “high recall” tasks (e.g., Interpol), though biased to names of certain nationalities
- Other algorithms for phonetic matching perform much better in the context of IR

Conclusion

- We have
  - Positional inverted index (with skip pointers)
  - Wild-card index
  - Spell-correction
  - Soundex

- We could solve queries such as

(SPELL(moriset) AND toron*to) OR *SOUNDEX* ***(chaikofski)***