# Search Engines

Gertjan van Noord

September 16, 2024

# Last week

- Choice of data structure for posting lists
  - hash (Python set, Python dict): fast, not compact
  - ordered lists: fast, compact
  - unorderd lists: very slow, compact
- Normalization (case, diacritics, stemming, decompounding, . . . )
- Phrase Queries
- Posting List with positions
- Suffix Array

# This week: Tolerant Retrieval

- Wildcard queries
- Spell correction
- Alternative indexes
- Finding the most similar terms

# Wildcard queries: *

- `mon*`: find documents with words that start with “mon”.
- `*mon`: find documents with words that end with “mon”. Harder.
- `mo*n`: find documents with words that start with “mo” and end with “n”. Even harder.
- `m*o*n`: Yet harder.

# Wildcard queries

- Step 1: find all terms that fall within the wildcard definition
- Step 2: find all documents containing any of these terms (business as usual)

# Wildcard queries

- Step 1: find all terms that fall within the wildcard definition
  - B-trees
  - Permuterm index
  - K-gram index
- Step 2: find all documents containing any of these terms

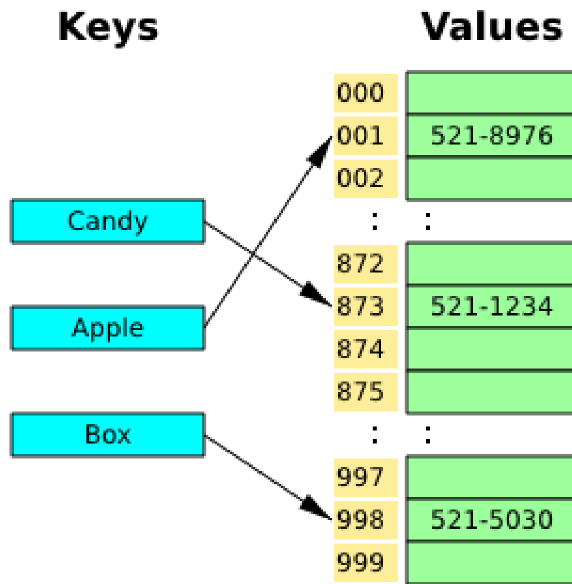
# Data structures for tolerant retrieval

- Hash. Very efficient lookup and construction, but a hash cannot be used to find terms that are "close" to the key.

- Python dictionaries and Python sets are implemented by hashes.

- Binary tree, B-tree, Tries.
  - fairly efficient search
  - words with same suffix are close together
  - compact

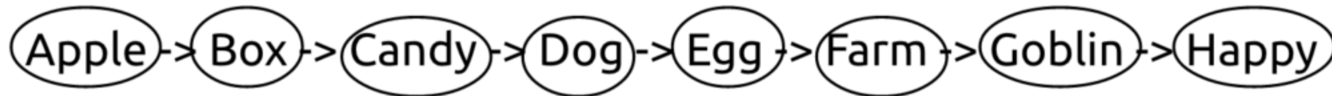
# Hash

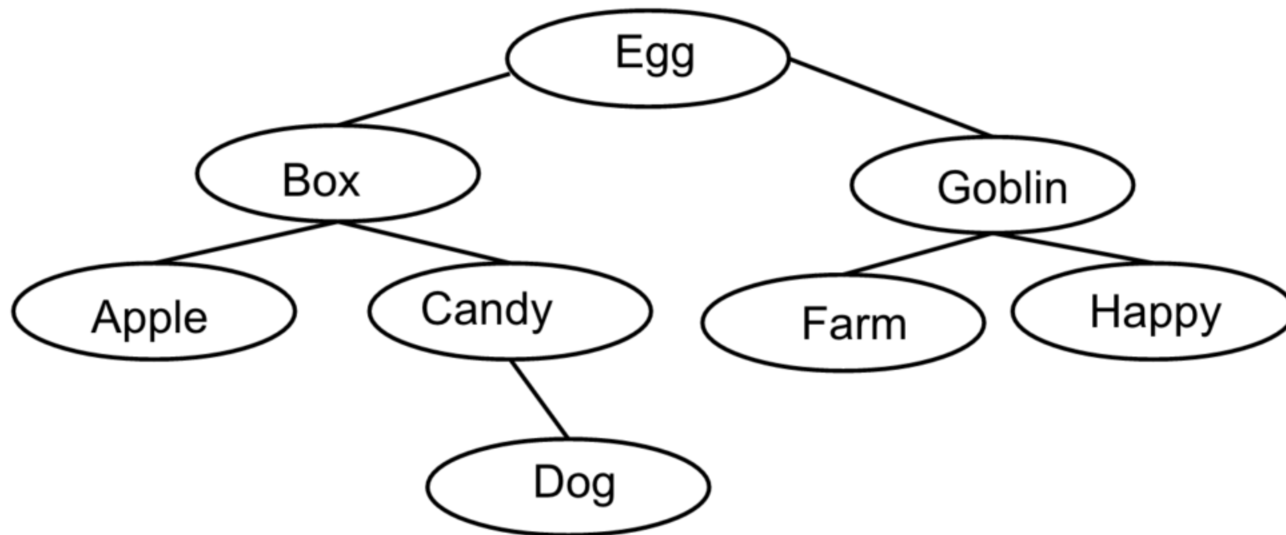


# Binary tree

## Binary Tree

The Dictionary Problem

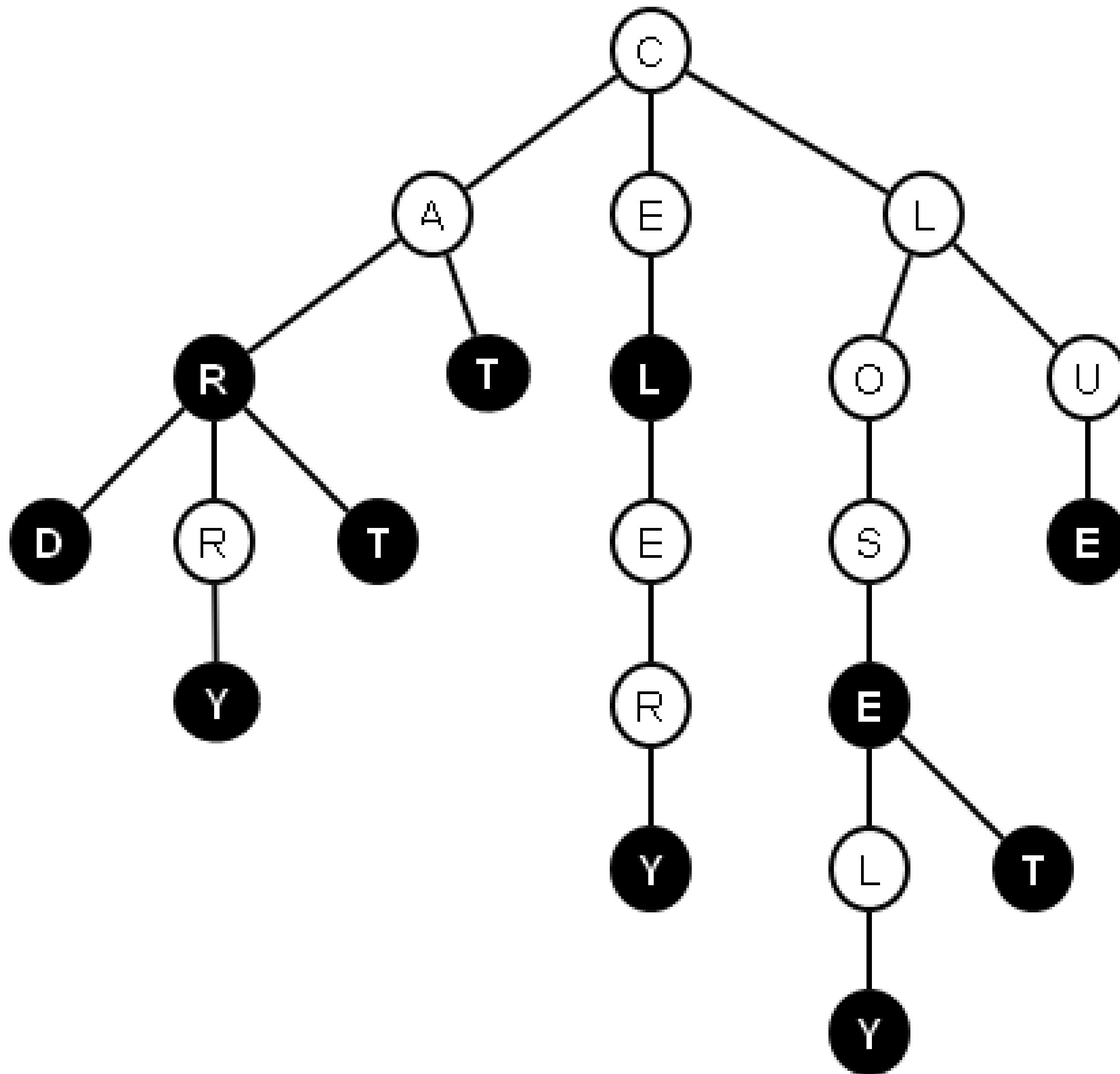


# B-tree

Extension of binary tree in which the tree remains *balanced*

# Trie



# Wildcard queries: *

- `mon*`: Easy with B-tree, easy with trie.
- `*mon`: Maintain additional B-tree or trie for all words in reverse
- `mo*n`: Intersect `mo*` and `*n`. Use reverse tree for `*n`.
- `m*o*n`: ??

# Wildcard queries: *

- `mon*`: Easy with B-tree, easy with trie.
- `*mon`: Maintain additional B-tree or trie for all words in reverse
- `mo*n`: Intersect `mo*` and `*n`. Use reverse tree for `*n`.
- `m*o*n`: ?? Use *permuterm* index or *K-gram* index

# K-gram index

K-gram: group of K consecutive items. Here: characters.

For example, if K=3, the K-gram index has keys of three consecutive characters. The key points to all terms which contain that sequence of three characters.

Index for dictionary lookup, not for document retrieval.

In a k-gram index, a key points to all relevant search terms.

## Split words in K-grams, K=3

```
kitchen
  ||
  ||
  \/
$kitchen$
  ||
  ||
  \/

 $ki
 kit
 itc
 tch
 che
 hen
 en$
```

# K-gram index, K=3

In a k-gram index, a key points to all relevant terms.

```
$ki  ==> {kift kinkiten kitchen kitten}
en$  ==> {een kinkiten kitchen kitten zen}
che  ==> {bitches cher kitchen witches}
ink  ==> {dinky kinkiten kinky link}
itt  ==> {bitter kitten litter}
kit  ==> {bokito kinkiten kitchen kitten nikita}
```

# K-gram index, K=3

In a k-gram index, a key points to all relevant terms.

```
$ki   ==> {kift kinkiten kitchen kitten}
en$   ==> {een kinkiten kitchen kitten zen}
che   ==> {bitches cher kitchen witches}
ink   ==> {dinky kinkiten kinky link}
itt   ==> {bitter kitten litter}
kit   ==> {bokito kinkiten kitchen kitten nikita}
```

Traditionally, the terms are *sorted* (why?) (alternative?)

# K-gram index for wildcard queries

Initial Query: `kit*en`

Mapped to: `$kit*en$`

Search in K-gram index: `$ki` AND `kit` AND `en$`

# K-gram index for wildcard queries

Initial Query: `kit*en`

Mapped to: $`kit*en`$

Search in K-gram index: $`ki` AND `kit` AND `en`$

Result: kinkiten kitchen kitten

Postprocessing required: kinkiten

# K-gram index for wildcard queries

Initial Query: `kit*en`

Mapped to: `$kit*en$`

Search in K-gram index: `$ki` AND `kit` AND `en$`

Result: kinkiten kitchen kitten

Postprocessing required: kinkiten

The remaining terms are used in OR query:

`kitchen` OR `kitten`

# Query processing

- What to do for this query: `se*ate` AND `fil*er`
- Expand `se*ate` to OR-query, e.g., `selfhate` OR `seagate`
- Expand `fil*er` to OR-query, e.g., `filter` OR `filler`
- Combine into
  ((`selfhate` OR `seagate`) AND (`filter` OR `filler`))

# Wildcard queries

- Map wild card expression to all possible terms
- Special data-structures to make that mapping efficient
- From then on, business as usual

# Spell Correction for Query Terms

# Spell Correction for Query Terms

If a query term is not present in the term index (or if it is very rare) . . .

- Find “similar” terms
- Calculate similarity to the query term
- Use most similar one(s)
- Use most frequent one(s)

# Spell Correction for Query Terms

Find most similar ones:

1. select a set of candidates (quick and dirty)
2. further reduce that set by a slower but more precise method

# Spell Correction for Query Terms

- Find most similar ones

  1. select a set of candidates (quick and dirty): **K-gram index, Jaccard**
  2. further reduce that set by a slower but more precise method: **Levenshtein**

- Use most similar one(s)

- Use most frequent one(s)

# Spell Correction for Query Terms

- Find “similar” terms: **K-gram index**

  For instance: a term $t_1$ is similar to $t_2$ if one of the 3-grams of $t_1$ and $t_2$ are identical.

  For unknown term $t_1$, collect all of the terms in the 3-gram index of all 3-grams.

  Lots of candidates, only use “good” ones?

# Spell Correction for Query Terms

Query: `brook`

`$bro` ⟶ `broek, brok, brommen, brons`

`roo` ⟶ `roomijs,vuurrood,brood,rook`

`ook` ⟶ `wierookstaafjes,stookolie,brood,rook`

`ok$` ⟶ `werknemersblok,varkenshok,rook`

# Spell Correction for Query Terms

Query: `brook`

`$bro` ⟶ `broek, brok, brommen, brons`

`roo` ⟶ `roomijs,vuurrood,brood,rook`

`ook` ⟶ `wierookstaafjes,stookolie,brood,rook`

`ok$` ⟶ `werknemersblok,varkenshok,rook`

From all those, only select the ones that are close to the original term

# Spell Correction for Query Terms

- Calculate similarity to the query term **Jaccard**

  Jaccard coefficient:

  $$\frac{\mid A \cap B \mid}{\mid A \cup B \mid}$$

  A: character trigrams in term $t_1$

  B: character trigrams in term $t_2$

# Jaccard

$$\frac{| A \cap B |}{| A \cup B |}$$

A: brook: $\$$br,bro,roo,ook,ok$\$$

B: rook: $\$$ro,roo,ook,ok$\$$

# Jaccard

$$\frac{\mid A \cap B \mid}{\mid A \cup B \mid}$$

A: brook: $\$$br,bro,roo,ook,ok$\$$

B: rook: $\$$ro,roo,ook,ok$\$$

A $\cap$ B: roo,ook,ok$\$$

A $\cup$ B: $\$$br,bro,roo,ook,ok$\$$,$\$$ro

# Jaccard

$$\frac{| A \cap B |}{| A \cup B |}$$

A: brook: $br,bro,roo,ook,ok$

B: rook: $ro,roo,ook,ok$

A $\cap$ B: roo,ook,ok$

A $\cup$ B: $br,bro,roo,ook,ok$,$ro

Jaccard: $3/6 = 0.5$

(in this context, for Jaccard, multisets are better than sets)

# More precise

Minimum Edit Distance

Levenshtein Distance

# Levenshtein Distance

Distance between A and B:

Minimum number of insertions, deletions or substitutions to map A to B

# Levenshtein Distance

Distance between A and B:

Minimum number of insertions, deletions or substitutions to map A to B

A: brook

B: rook

# Levenshtein Distance

Distance between A and B:

Minimum number of insertions, deletions or substitutions to map A to B

A: brook

B: rook

Distance: 1

# Levenshtein distance

| | |
|---|---|
| `bakker` | `brak` |
| `otter` | `boter` |
| `bloed` | `bode` |
| `ondersteboven` | `binnenstebuiten` |

# Efficient Algorithm

- try out all possibilities?
- No. First compute Levenshtein distance for all prefixes
- *Dynamic programming*

Suppose we need to compute distance for:

`ondersteboven,binnenstebuiten`

and we are given that:

dist(`onderstebove,binnenstebuite`) = x
dist(`onderstebove,binnenstebuiten`) = y
dist(`ondersteboven,binnenstebuite`) = z

min(x, y+1, z+1)

# Efficient Algorithm

x and y are sequences of symbols

a and b are symbols

Suppose we need to compute distance for

dist(xa,yb)

and we have

dist(x,y)
dist(xa,y)
dist(x,yb)

# Efficient Algorithm

? dist(xa,yb)

and we have

dist(x,y)
dist(xa,y)
dist(x,yb)

# Efficient Algorithm

cost(a,b): 0 if a==b; 1 otherwise

There are three ways to construct xa,yb:

dist(x,y) + cost(a,b) (substitution)
dist(xa,y) + 1 (insdel)
dist(x,yb) + 1 (insdel)

# Efficient Algorithm

cost(a,b): 0 if a==b; 1 otherwise

There are three ways to construct xa,yb:

dist(x,y) + cost(a,b) (substitution)
dist(xa,y) + 1 (insdel)
dist(x,yb) + 1 (insdel)

Take the minimum

# Efficient Algorithm: matrix

| | # | b | l | o | e | d |
|---|---|---|---|---|---|---|
| # | 0 | | | | | |
| b | | | | | | |
| o | | | | | | |
| d | | | | | | |
| e | | | | | | |

# Efficient Algorithm: matrix

| | # | b | l | o | e | d |
|---|---|---|---|---|---|---|
| # | 0 | 1 | | | | |
| b | | | | | | |
| o | | | | | | |
| d | | | | | | |
| e | | | | | | |

# Efficient Algorithm: matrix

| | # | b | l | o | e | d |
|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 |
| b | | | | | | |
| o | | | | | | |
| d | | | | | | |
| e | | | | | | |

# Efficient Algorithm: matrix

| | # | b | l | o | e | d |
|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 |
| b | 1 | | | | | |
| o | | | | | | |
| d | | | | | | |
| e | | | | | | |

# Efficient Algorithm: matrix

| | # | b | l | o | e | d |
|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 |
| b | 1 | 0 | 1 | 2 | 3 | 4 |
| o | 2 | 1 | 1 | 1 | 2 | 3 |
| d | 3 | 2 | 2 | 2 | 2 | 2 |
| e | 4 | 3 | 3 | 3 | 2 | 3 |

# Efficient algorithm: matrix

- Each cell in the matrix represents the distance between the corresponding prefixes
- The final result, therefore, can be found in . . .

# Efficient algorithm: matrix

- Each cell in the matrix represents the distance between the corresponding prefixes
- The final result, therefore, can be found in the rightmost and lowest cell
- Other cost functions are possible too
- E.g., substitutions for characters that are pronounced similarly could be given lower cost
- Sometimes, other basic edit operations can be considered (e.g. transposition `ab` - `ba`, `ij` - `y`

# Efficient?

- If a particular approach is efficient depends on the problem that you try to solve (some problems are harder than others)

- In addition, it depends on the typical instances of the problem

# Efficient?

- Intersection of lists: quadratic is inefficient
- Levenshtein distance: quadratic is efficient
  - there is no better algorithm
  - the typical input size is very small (if applied to words, not if applied to DNA sequences)
- . . .

# Suffix Array

# Suffix array

- Very simple, but interesting data structure useful for several string problems.
- Used in:
  - text search
  - plagiarism detection
  - data compression
  - computational biology
  - large language models

# Suffix array

- Searching very long phrase queries
- What is the longest repetition in a given text
- What are very long identical passages in two given texts
- What is most frequent 12-gram (sequence of 12 words) in a given text

# Suffix array

Wikipedia: A suffix array is a sorted array of all suffixes of a string.

# Suffix array

String: informatie

Suffixes:

```
informatie
nformatie
formatie
ormatie
rmatie
matie
atie
tie
ie
e
```

Ordered:

```
atie
e
formatie
ie
informatie
matie
nformatie
ormatie
rmatie
tie
```

# Suffix Array

Represent each suffix $c_i \ldots c_n$ by its starting position $i$ Suffix:

0
1
2
3
4
5
6
7
8
9

Ordered:

6
9
2
8
0
5
1
3
4
7

# Suffix Array

Suppose we create a suffix array for `small.txt` (ignoring the keys), and then print the first characters of each suffix in the order of the suffix array:

```
aajee aajoo " .\nIn het dagelijks leven is hij vri
aajoo " .\nIn het dagelijks leven is hij vrijgeves
aak ( " Acer campestre " ) is een plant uit de ze
aak .\nDe Spaanse aak ( " Acer campestre " ) is ee
aak herfstkleuren .\nDe soort kan eenhuizig of twe
aak wordt tot 10 m hoog .\nDe plant wordt vaak als
aal ’ dan wel met ’ paling ’ worden aangeduid .\nE
aal ( " Anguilla anguilla " ) , is een straalvinn
aal en de roofzuchtige murene , alle soorten zijn
aalmoes of smeergeld functioneert .\nDe term is al
aalmoezenier , maar bleef doorstuderen .\nVlad Dra
aalmoezenier en koormeester van de Londense St. P
aalscholver ( " Phalacrocorax auritus " ) ( Doubl
aalscholver ( " Phalacrocorax carbo " ) , ook wel
aalscholver .\nDe Amerikaanse aalscholver ( " Phal
aalscholver behoort tot de familie van de aalscho
```

aalscholver uit de familie van fuutkoeten .\nUiter
aalscholvers ( Phalacrocoracidae ) , waarvan 36 s
aalscholvers .\nHet was het eerste gebied dat de v
aalscholvers en een aantal sterns .\nEtheenoxide .
aaltjes " behoren tot de nematoden .\nNematologie
aambeeld ( of aanbeeld ) is een gereedschap .\nHet
aambeeld .\nEen aambeeld ( of aanbeeld ) is een ge
aambeeld .\nSoms is de gehele troposfeer onstabiel
aambeeld er tien dagen over zou doen om van het o
aambeeld zou tien dagen nodig hebben om vanaf Our
aan\nFrans .\nHet Frans ( " français " ) behoort t
aan\nIJshockey valt onder de " balsporten " maar d
aan\nInterferentie van elektronen door Claus Jöns
aan\nMineraal .\nEen mineraal is een stof die in ho
aan\nNog anders geformuleerd , een basis van vecto
aan " ) en het verbreken van een verbinding een l

# Suffix Array

- Searching very long phrase queries:

  do a binary search in the suffix array to find the first suffix which starts with the query

# Suffix Array

- What is the longest repetition in a given text:

  one pass through the suffix array and keep track of the longest identical prefix seen so far

# Suffix Array

- What are very long identical passages in two given texts:

  do a kind of merge sort of the two suffix arrays of both texts and report suffixes with long identical prefixes

# Suffix Array

- What is most frequent 12-gram (sequence of 12 words) in a given text:

  one pass through the suffix array and keep track of the most frequent 12-gram prefix seen so far

# Suffix Array

- A suffix array for a given text can be constructed
  - in linear time
  - without much additional memory
- the naive method is slow, but is a one-liner in Python

```
def construct_sufarr(corpus):
    return sorted(range(len(corpus)), key=lambda el: corpus[el:])
```

- there are Python libraries with efficient implementation

# pydivsufsort

- We use `pydivsufsort`, a Python library
- `github.com/louisabraham/pydivsufsort`
- Installation (for me . . . ):

```
python3 -m pip install pydivsufsort
```

# pydivsufsort

```
>>> from pydivsufsort import divsufsort
>>> mystr = "abracadabra"
>>> sufarr = divsufsort(mystr)
>>> sufarr
array([10,  7,  0,  3,  5,  8,  1,  4,  6,  9,  2], dtype=int32)
```

# pydivsufsort

- this only supports ASCII
- we can use bytes instead - but now it gets a bit messy
- my `sufarr.py` does some of the messy stuff for you

```
def construct_corpus(fd):
    """ read in corpus from file descriptor fd """
    corpus = fd.read()
    return corpus.encode('utf-8')
```

# sufarr.py

```
def construct_sufarr(corpus):
    """ Return suffix array for <corpus> """
    print("Constructing suffix array...", file=sys.stderr)
    sufarr = divsufsort(corpus)
    print("Constructing suffix array done", file=sys.stderr)
    return sufarr
```

# sufarr.py

```
def print_sufarr(sufarr, corpus, suflen=None):
    """ Print first <suflen> chars for each suffix of sufarr """
    for ix in sufarr:
        print_suffix(ix, corpus, suflen)

def print_suffix(ix, corpus, suflen=None):
    """ Print first suflen characters of suffix nr ix for corpus """
    str = get_suffix(ix, corpus, suflen)
    if str:
        print(str)
```

# sufarr.py

```
def get_suffix(ix, corpus, suflen=None):
    """ Print first <suflen> characters of corpus[ix:] """
    # 1. return None if first byte is not valid
    # 2. throw away last byte if not valid
    if suflen:
        suffix = corpus[ix:ix+suflen]
    else:
        suffix = corpus[ix:]
    suf_str = suffix.decode(’utf-8’, errors="replace")
    if ord(suf_str[0]) == 65533:  # check if valid start
        return None
    suf_str = suf_str.rstrip(chr(65533))  # remove invalid last
    return suf_str.replace("\n", "\\n")
```

# sufarr.py

```
def main():
    """ Construct suffix array for input and print each suffix. """
    """ For readability, each newline is printed as \n """
    parser = argparse.ArgumentParser(description='Print suffixes ordered')
    parser.add_argument('-w', '--width', type=int, default=50)
    args = parser.parse_args()
    corpus = construct_corpus(sys.stdin)
    sufarr = construct_sufarr(corpus)
    print_sufarr(sufarr, corpus, args.width)
```

# sufarr.py

```
$ head -n 1000 small.txt | cut -f 2 | python3 sufarr.py -w 20
...
 a-krol , alfa , adr
 aan .\nHet atoomnumm
 aan 150 miljoen kil
 aan Archimedes word
 aan Botswana .\nHet
 aan Europa en Afrik
 aan Frans-Guyana ,
 aan Iran en in het
 aan Jacob Gerritsz
 aan Pakistan , in h
 aan Turkmenistan ,
...
```

(the Unix command `cut` is used to remove the keys in `small.txt`)

# Assignment: A. print long repeated sequences

Your task is to create a program `repetitions.py` which will take a (potentially large) text, and then print all identical sub-sequences of that text of a particular length. The length is given as a parameter to the program. Using the `small.txt` file, my program produces the following output:

```
$ cut -f 2 small.txt | python3 repetitions.py 50

Constructing suffix array...
Constructing suffix array done

\n! colspan= " 4 " bgcolor= " bisque " | Faustina d
\n" Little Sammy Sneeze " ( 1904-06 ) door Winsor M
\nAfhankelijk van het product , de productiemethode
\nAlfastraling ( bovenste ) wordt al tegengehouden
\nAlleen onafhankelijke landen , " landen die deels
```

## Assignment: B. print most frequent long repeated sequence

For the B part, you are asked to create the file `most_freq_repetitions.py`.

This program is a variation of the program you created in part A. This variation keeps track which repeated sequence occurs most frequently. That sequence and its frequency is printed. Here is how my program does this:

```
$ cut -f2 small.txt |python3 most_freq_repetitions.py 80
Constructing suffix array...
Constructing suffix array done
76    ste dag in een schrikkeljaar ) in de gregoriaanse kalender .\nHierna
```

# Assignment C. Not obligatory. Not for grade. Just to impress me.

Implement the program `query_sufarr.py` which takes one argument: the name of a (potentially large) text file. After creating a suffix array for that file, the program reads queries from standard input. For each query (a string) it will print the first 50 characters of the suffix that starts with the same string as the query.

```
$ python3 query_sufarr.py <(cut -f2 ../../small.txt)
Constructing suffix array...
Constructing suffix array done
query> Dat is een
Dat is een andere benadering dan een compiler , di
Dat is een derde van al het afgedankt textiel .\nDe
Dat is een in reuzel gebakken kleine aardappel .\nH
Dat is een organisatie waarin alle schakels uit de
Dat is een zeer uitgestrekt gebied , waar de sprek
query> Maar dat wil
Maar dat wil niet zeggen dat mantra's oude begripp
query>
```

```
python3 query_sufarr.py <(cut -f2 ../../nlwiki2011.txt)
Constructing suffix array...
Constructing suffix array done
query> Hij heeft toen
Hij heeft toen de hoogste score ooit behaald in ee
Hij heeft toen in Haarlem de eerste voetbalclub op
Hij heeft toen ook de nummer-1-hit " Check On It "
Hij heeft toen vermoedelijk de nederzetting gestic
query>
```

Constructing the suffix array (252Mb) takes about 30 secs.

## Assignment D. Not obligatory. Not for grade. Just to show off.

Construct a program `longest_repetition.py` which prints the longest repeated substring of a given (large) text file.

My solution is slow. The following takes 35 secs:

```
$ cut -f2  ../../small.txt | python3 longest_repetition.py
434
 hiervan zo onder de indruk dat hij besloot dit voorbeeld te volgen .\nHij
 gaf zijn bezit aan de armen en ging prediken .\nHet evangelie liet hij
 vertalen in het Occitaans , de streektaal van Zuid-Frankrijk , waardoor
 het ook voor de gewone mensen goed te begrijpen was .\nPetrus en zijn
 volgelingen trokken rond om hun kennis van de bijbel te verspreiden .\n
Zij brachten daarbij kritiek uit op de levenswijze van de rijke
 geestelijken .\n
```

Maybe you can do better.