RICE SOCIAL SCIENCES
Department of Linguistics

# encoding & tokenization

## LING 430 Computational Linguistics Fall 2026
## lecture 4b
## Sep 17 2026

**Siyu Liang**
siyu.liang@rice.edu

# last time

- regular expressions

# agenda

- encoding & Unicode
- tokenization
- byte-pair encoding

# today's notebook

## class04b.ipynb

- course website → week 4 → activity
- **File → Save a copy in Drive**

# encoding

# encoding

- a computer stores numbers
  - just 0s and 1s
- every character is converted to numbers
- an **encoding** is the set of rules about which number means which character

# encoding

## bit & byte

- at the bottom a computer stores **bits**, each 0 or 1
- a **byte** is **eight bits**, the unit files are measured in
  - eight bits give $2^8 = 256$ different values
- with one byte per character, an alphabet can have at most 256 characters
- e.g. *H* is 72, which is 01001000: 64 + 8

```
ord("H")
```
→ 72

```
bin(72)
```
→ '0b1001000'

# encoding

## ASCII

- American Standard Code for Information Interchange (ASCII) was the first major encoding convention
  - one byte per character
  - 128 codes
- English letters, digits and punctuation
- the first 32 are control characters, many of which obsolete now
  - but some are useful
  - `\t` is 9, `\n` is 10

**ASCII TABLE**

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [END OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

# encoding

## ASCII

# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [END OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

# encoding

- `ord()` takes a **character** and returns its **code point**
- `chr()` does the reverse

`ord("a"), ord("A")`

→ (97, 65)

`chr(97)`

→ 'a'

# encoding

- sorting with `sorted()` uses the code point

  `sorted(["banana", "Apple", "cherry", "zebra", "Zoo", "啊", "😮", "Éclair"])`
  → ['Apple', 'Zoo', 'banana', 'cherry', 'zebra', 'Éclair', '啊', '😮']
  `"Z" < "a"`
  → True

- what about non-Latin alphabet characters?

# encoding

- non-Latin characters do have code points, so what is the pattern?

```
ord("É"), ord("啊"), ord("😮")
```

→ (201, 21834, 128558)

```
chr(601), chr(643)
```

→ ('ə', 'ʃ')

# encoding

- there have been temporary patches to ASCII to include other scripts
- each one reused the same 128 ASCII values
- some older systems relies on this, but breaks with multi-script texts

| standard | the upper 128 codes hold |
|---|---|
| ISO 8859-1 (Latin-1) | Western European letters: é ñ ü ß |
| ISO 8859-7 | the Greek alphabet |
| ISO 8859-8 | the Hebrew alphabet |
| JIS X 0208 | Japanese, in pairs of bytes |

# encoding

## Unicode

- the Unicode standard is the convention now
- goal: one **code point** for **every character** in **every writing system**
- written `U+` and a **hexadecimal** number
  - *a* is U+0061, which is 97
- room for 1,114,112 code points, up to U+10FFFF
  - some constraints govern the number
- the first 128 are same as ASCII for compatibility

# encoding

## Unicode

- the newest version, Unicode 18.0, was released yesterday! 🥳
- see the news article to find out how it matters for emoji users like □
- warning: depending on your Python version, you might not be able to directly process certain emojis
  - e.g. □ was added in Unicode 17 which is still not supported in Python 3.14

# encoding

## Unicode

- you can look up a Unicode in a database such as this
- and in Python

```
import unicodedata
unicodedata.name("é")
```

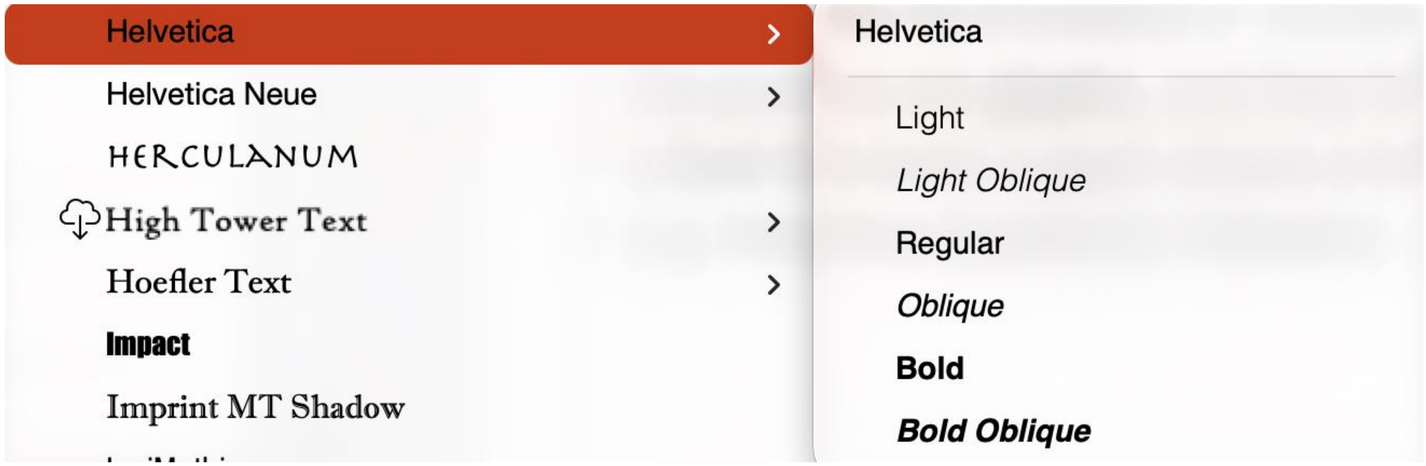
→ 'LATIN SMALL LETTER E WITH ACUTE'

| code point | character | name |
|---|---|---|
| U+0061 | a | LATIN SMALL LETTER A |
| U+00E9 | é | LATIN SMALL LETTER E WITH ACUTE |
| U+00FC | ü | LATIN SMALL LETTER U WITH DIAERESIS |
| U+8FDB | 进 | CJK UNIFIED IDEOGRAPH-8FDB |
| U+1F600 | 😀 | GRINNING FACE |

# encoding

## typeface & font

- in a sense, all texts you see on a screen are pictures
- the “a” letter as a (Times), `a` (Courier), **a (bold)** or *a (italic)* are all U+0061
- the pictures are **glyphs**, and they live in **typefaces**
- a **font** that lacks a glyph shows a box □
- e.g. Helvetica (typeface)
  - Light (font)
  - ***Bold Oblique (font)***



# encoding

- `len()` actually counts based on code points
- this is how things like skin tone is handled too: 👍 has a len of two, a hand and a skin tone

```
len("🦆")
```

→ 1

```
len("👩‍👧")
```

→ 5

```
for ch in "👩‍👧":
    print(ord(ch), unicodedata.name(ch))
```

→ 128104 MAN

→ 8205 ZERO WIDTH JOINER

→ 128105 WOMAN

→ 8205 ZERO WIDTH JOINER

→ 128103 GIRL

# encoding

- as we mentioned: Unicode allows up to 1,114,112 code points
- which needs 21 bits ($2^{21}$ = 2,097,152)
- but computers also store zeros, so you would end up with a lot of zeros
- and your files will be much larger than it should be!
- so, how about we use fewer bytes for the more “common” characters in languages?
- … said two English speakers (Ken Thompson & Rob Pike, Bell Labs)

# encoding

## Unicode Transformation Format

- **UTF-8** is the most common encoding, as in almost the entire web
- key feature: **variable-length**, common characters one byte, others up to four
- every ASCII file is already a valid UTF-8 file
- UTF-16 (2-4 bytes per char); UTF-32 (4-byte per char)

| code points | bytes | used for |
|---|---|---|
| U+0000 to U+007F | 1 | ASCII |
| U+0080 to U+07FF | 2 | most European and Middle Eastern scripts |
| U+0800 to U+FFFF | 3 | Chinese, Japanese, Korean, Ethiopic, etc. |
| U+10000 and above | 4 | rarer CJK characters, emoji |

# encoding

## .encode()

- `.encode()` turns a string into bytes
- `b''` marks a bytes value; `\xc3` is a byte in hex

`"café".encode("utf-8")`

→ b'caf\xc3\xa9'

`len("café"), len("café".encode("utf-8"))`

→ (4, 5)

`"中".encode("utf-8")`

→ b'\xe4\xb8\xad'

# encoding

## .decode()

- `.decode()` turns bytes back into a string using a specified encoding
- **mojibake** 文字化け
  - decoding errors you have seen on players in old cars, subtitles, etc.

```
b'caf\xc3\xa9'.decode("latin-1")
```

→ 'cafÃ©'

# encoding

## in Python

- a file is a sequence of bytes
- `open()` decodes on the way in and encodes on the way out
  - there is no such thing as a text file without an encoding!
- most data is UTF-8, Colab assumes UTF-8, so most of the time `open("")` needs no argument
  - but you'll see `open(f, encoding="utf-8")`
- when a file is not UTF-8, say so: `open(f, encoding="latin-1")`

# activity 1

- work in groups, each of you takes a word in a language you know, ideally one with accents, another script, or emoji
- for each character print `ord()` and `unicodedata.name()`
- report three numbers: characters you see, code points, and bytes (`len(s.encode())`)
- then swap: explain to your partner the one code point in your string that surprised you

# tokenization

# tokenization

- written language data are basically strings, which are usually very long
- **tokenization**: the process to segment text into **tokens**, the first step of all language processing task
- why is it hard?
  - punctuation as separate tokens, but kept inside *m.p.h.*, *Ph.D.*, *AT&T*, *cap'n*
  - prices, dates, URLs, hashtags and email addresses kept whole or standardized: *$45.55*, *01/02/06*, *#grwm*
  - numbers: *555,500.50* in the US; *555 500,50* in many European countries
  - clitics expanded: *what're* → *what are*; French *j'ai*, *l'homme*
  - multiword expressions could be one token: *New York*, *rock 'n' roll*

# tokenization

## a first tokenizer

- a regex pattern is already a tokenizer
- both cut *lawyer's* and *wasn't* at the apostrophe, which is the wrong analysis; a real tokenizer is this idea plus a list of exceptions

```
s = "The lawyer's group wasn't amused, was it?"
re.findall(r"\w+", s)
```

→ ['The', 'lawyer', 's', 'group', 'wasn', 't', 'amused', 'was', 'it']

```
re.findall(r"\w+|[^\w\s]", s)
```

→ ['The', 'lawyer', "'", 's', 'group', 'wasn', "'", 't', 'amused', ',', 'was', 'it', '?']

# tokenization

## a regex tokenizer

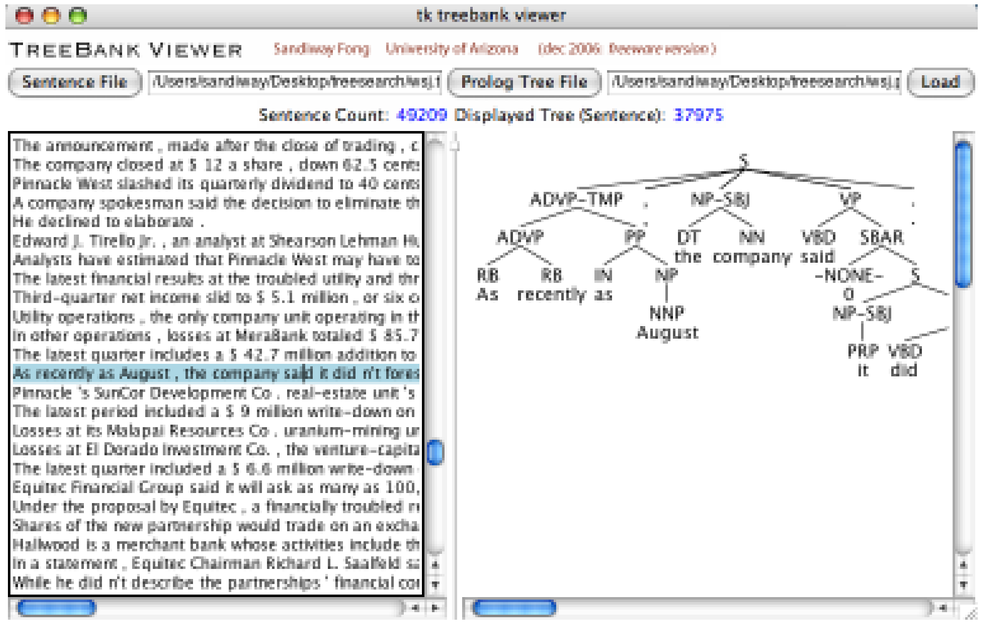
- you can define more complex tokenizer with regex using nltk.regexp_tokenize

```
text = "That U.S.A. poster-print costs $12.40..."
pattern = r"""(?x)
    (?:[A-Z]\.)+     # abbreviations, e.g. U.S.A.
  | \w+(?:-\w+)*     # words with optional internal hyphens
  | ...
"""
nltk.regexp_tokenize(text, pattern)
```

→ ['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']

# Treebank

- words have **part of speech (POS)**
  - noun, verb, adjective, etc.
- POS tagging was considered a fundamental problem in natural language processing
  - people have spent a lot of time building the data to train and test computers on it
- **Penn Treebank** is the most famous one
  - available in many languages, supposedly universal
- we'll talk more about it in week 9



# tokenization

## the Penn Treebank standard

- the tokenization standard behind the Penn Treebank & many other corpora
- clitics split off: *doesn't* becomes *does* and *n't*; if *not* is a word, *n't* is one too
- hyphenated words kept together
- all punctuation separated, including the *$*

```
input:   "The San Francisco-based restaurant," they said,
"doesn't charge $10".
```

```
output:  " The San Francisco-based restaurant , " they said , "
does n't charge $ 10 " .
```

# tokenization

## word_tokenize()

- nltk's tokenizer implements the Treebank standard

```
from nltk import word_tokenize
s = "The lawyer's group wasn't amused, was it?"
s.split()
```

→ ['The', "lawyer's", 'group', "wasn't", 'amused,', 'was', 'it?']

```
word_tokenize(s)
```

→ ['The', 'lawyer', "'s", 'group', 'was', "n't", 'amused', ',', 'was', 'it', '?']

# tokenization

## word_tokenize()

- a period ends a sentence, marks an abbreviation, or both at once (*Inc.* at the end of a sentence)
- *Dr.* and *p.m.* were kept whole; *al.* was not: the abbreviation list did not know it

```
word_tokenize("Dr. Niedzielski et al. arrived at 5 p.m. today.")
```

→ ['Dr.', 'Niedzielski', 'et', 'al', '.', 'arrived', 'at', '5', 'p.m.', 'today', '.']

# tokenization

## tokenizers compared

| tokenizer | wasn’t | $2.44 | it? |
|---|---|---|---|
| `.split()` | wasn’t | $2.44 | it? |
| `re.findall(r"\w+|[^\w\s]")` | wasn · ’ · t | $ · 2 · . · 44 | it · ? |
| `wordpunct_tokenize` | wasn · ’ · t | $ · 2 · . · 44 | it · ? |
| `word_tokenize` | was · n’t | $ · 2.44 | it · ? |

# activity 2

- work in pairs to write sentences that to test the limit of the tokenizers
- print several tokenizers output side by side
- does output make sense to you?
- suggestions:
  - abbreviations, URLs, email addresses, emoji, hyphens, other scripts
- discussion: how would you do differently?

# byte-pair encoding

# BPE

- there are several candidates for tokens

| unit | for | not for |
|---|---|---|
| words | carry meaning | no orthographic words in many languages<br>the vocabulary never stops growing |
| morphemes | the right size | hard to define<br>hard to segment automatically<br>different in every language |
| characters | easy to define, few of them | too small to mean anything on their own<br>sequences get very long |

# BPE

- to build computational models, a system learns from a **training** corpus and is used on a **test** corpus it has not seen
  - e.g. training has *low*, *new*, *newer*
  - the test text has *lower*
- with words as the unit, *lower* is simply unknown
- but if we have pieces smaller than words, *lower* could be *low* + *er*, so both already have been seen

# BPE

- **subword tokenization**: let the data decide whatever character sequences recur often to become tokens
- the tokens that come out could be word-sized, morpheme-sized, across morpheme boundaries, or single characters
- two algorithms in wide use
  - **byte-pair encoding (BPE)** (Sennrich et al. 2016)
  - unigram language modeling

# BPE

## how to build a BPE vocabulary

- start with a **vocabulary of single characters**
- count **every pair of adjacent symbols** in the corpus
- **merge the most frequent pair** into one new symbol, and add it to the vocabulary
- repeat *k* times
- A B D C A B E C A B  # most frequent pair: A B
- AB D C AB E C AB  # then: C AB

```
AB D CAB E CAB
```

# BPE

## the corpus

- in practice merges stay inside words, so the corpus is split at spaces first and each word keeps its leading space as a symbol, ␣
- input: `set new new renew reset renew`
- output: `s e t · ␣new · ␣new · ␣re new · ␣re s e t · ␣re new`

| merge | pair | count | vocabulary |
|---|---|---|---|
| 0 | | | ␣ e n r s t w |
| 1 | (n, e) → ne | 4 | ␣ e n r s t w **ne** |
| 2 | (ne, w) → new | 4 | ␣ e n r s t w ne **new** |
| 3 | (␣, r) → ␣r | 3 | ␣ e n r s t w ne new **␣r** |
| 4 | (␣r, e) → ␣re | 3 | ␣ e n r s t w ne new ␣r **␣re** |

# BPE

- real systems run BPE over UTF-8 **bytes** rather than characters
- only 256 byte values exist, so there is no such thing as an unknown token
- usually tens of thousands of merges to get vocabularies of 50,000 to 200,000 tokens
- GPT-4o uses about 200,000
  - try it out on Tiktokenizer

# activity

## Tiktokenizer

- work in a group, open tiktokenizer.vercel.app, and choose GPT-4o
- pick one or more directions to explore
  - a language one of you knows: tokens per word on three sentences, against the same sentences in English
  - numbers, dates, URLs and hashtags: how are they chunked?
  - a sentence that switches language halfway: what happens at the switch?
  - one word in six forms, *walk walks walked walking walker unwalkable*: where do the splits fall?

# next time

- solutions will be posted on Canvas after each homework is due
  - Canvas → File → reference solutions
  - hw1 solution is posted
- Tuesday
  - computational morphology
- **HW 2 due tonight, 11:59 pm**