Understanding Graphemes

This is a concise guide to bytes, Unicode code points ("chars"), and graphemes, with strong review notes and corrected examples. Use it for input limits, slicing, display, and search logic across languages.

1) Essentials: What to Count & When

• Bytes = storage/transmission (file sizes, memory, network).

• Characters (Unicode code points) = language building blocks (not always what users see).

• Graphemes = what users see as one character (use for UI, limits, cursor movement).

When to use each:

- Bytes: memory allocation, file I/O, network payload sizes.

- Code points: Unicode processing that cares about scalars (rare for UI).

- Graphemes: user input limits, cursor movement, selection, word wrapping, display width.

2) Strong Review of the Source (Applied Fixes)

• The “family emoji” example must be a ZWJ sequence to be 1 grapheme. Use “👨‍👨‍👧‍👧” (with zero-width joiners), not “👨👨👧👧”. The latter is 4 graphemes.

• JavaScript’s `text.length` counts UTF-16 code units, not Unicode code points; prefer `Intl.Segmenter` or iteration over code points (for modern engines).

• For search (e.g., "café" vs "cafe\u{0301}"), normalize first (NFC/NFKC) before comparing, or compare on grapheme-cluster boundaries.

• Never slice by raw byte offsets unless you checked a char boundary; in Rust, slicing at non-char boundary panics.

3) One Consistent Demo String (Use This)

Let: s = "Hello 👨‍👨‍👧‍👧 café"   (family emoji uses ZWJ)

Expected counts on typical systems:

• Bytes (UTF-8): platform-dependent for emoji sequences but > ASCII length; verify via code below.

• Chars (code points): larger than graphemes for ZWJ sequences.

• Graphemes: each user-visible character (family emoji counts as 1).

Rust (recommended)

```
use unicode_segmentation::UnicodeSegmentation;
fn main() {
   let s = "Hello 🧑\u{200D}🧑\u{200D}🧒\u{200D}🧒 café";
   println!("Bytes: {}", s.as_bytes().len());
   println!("Chars: {}", s.chars().count());
   println!("Graphemes: {}", s.graphemes(true).count());
   // Safe truncation to first 10 graphemes:
   let first10 = s.graphemes(true).take(10).collect::<String>();
   println!("First 10 graphemes safely: {}", first10);
   // Avoid: &s[0..N]; may panic if not at char boundary.
}
```

Python

```
from unicodedata import normalize
# pip install grapheme
import grapheme

s = "Hello 🧑\u200d🧑\u200d🧒\u200d🧒 café"
print("Bytes:", len(s.encode("utf-8")))
print("Chars:", len(s))
print("Graphemes:", grapheme.length(s))
# Safe limit by graphemes:
print("First 10:", grapheme.slice(s, 0, 10))
# Search equivalence (normalize both):
a = normalize("NFC", "café")
b = normalize("NFC", "cafe\u0301")
print("Equal after NFC:", a == b)
```

JavaScript

```
const s = "Hello 🧑\u200d🧑\u200d🧒\u200d🧒 café";
console.log("Bytes:", new Blob([s]).size);
console.log("UTF-16 code units (length):", s.length);
// Graphemes via Intl.Segmenter
const seg = new Intl.Segmenter(undefined, { granularity: "grapheme" });
const graphemes = [...seg.segment(s)];
console.log("Graphemes:", graphemes.length);
// Safe truncate to 10 graphemes:
console.log("First 10:", graphemes.slice(0,10).map(x => x.segment).join(""));
```

4) Do & Don’t

✅ DO: Count graphemes for user-visible limits (tweets, names, messages).

✅ DO: Normalize text (NFC/NFKC) before search/compare when accents can vary.

✅ DO: Wrap/cursor by grapheme boundaries in editors/UI.

❌ DON’T: Slice strings by bytes or assume `.length` equals user characters.

❌ DON’T: Assume 1 code point == 1 glyph; emoji, accents, and complex scripts break this.

5) Quick Recipes

- Safe truncate N graphemes: take N graphemes and collect back to a string.

- Count “real” characters: grapheme count, not bytes or code points.

- Search: normalize both sides, compare on grapheme boundaries where relevant.

- Test strings: ASCII, accented (café), combining (e +´), emoji, ZWJ family, non-Latin (e.g., Thai, Hindi).