```
# -*- coding: utf-8 -*-
"""AGC_128_Standart_notepad_v.3.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1U2EvlrXkoK9gtJF6Ci8fF4HqoIh3QC1V

```
AGC_128_Standart_notepad_v.1 — Adaptive Genetic Code 128
Official README (First Recorded Chat Edition)

Authors
- **Aleksandar Kitipov**
  Emails: aeksandar.kitipov@gmail.com / aeksandar.kitipov@outlook.com
- **Copilot**
  Co-author, technical collaborator, documentation support

  - **Gemini 2.5 Flash**
  Co-author, technical collaborator, documentation support

Overview
AGC_128_v.1 is a lightweight, fully reversible, DNA-inspired text encoding system.
It converts any ASCII text into a stable A/T/G/C genetic sequence and can decode it back **1:1 without loss**.

The entire encoder/decoder is approximately **15 KB**, requires **no external libraries**, and runs instantly even on older 32-bit machines.
This README is based on the very first conversation where AGC-128 was conceived, tested, and formalized.

What the Program Does
AGC_128_v.1 performs a complete reversible transformation:

```
Text → ASCII → Binary → Genetic Bits → A/T/G/C DNA Sequence
```

and back:

```
DNA Sequence → Genetic Bits → Binary → ASCII → Text
```

The system preserves:
- letters
- numbers
- punctuation
- whitespace

- ASCII extended symbols
- structured blocks
- FASTA-formatted sequences

If you encode text and decode it again, the output will match the original **exactly**, character-for-character.

Key Features

1. Fully Reversible Encoding
Every ASCII character becomes a 4-gene sequence.
Decoding restores the exact original text with zero corruption.

2. Self-Checking Genetic Structure
AGC-128 uses three internal biological-style integrity rules:

Sum-2 Rule
Each 2-bit gene has a total bit-sum of 2.
Any bit flip breaks the rule and becomes detectable.

No-Triple Rule
The sequence can never contain `111` or `000`.
If such a pattern appears, the data is invalid.

Deterministic-Next-Bit Rule
- After `11` → the next bit must be `0`
- After `00` → the next bit must be `1`

This allows partial reconstruction of missing or damaged data.

Together, these rules make AGC-128 extremely stable and self-verifying.

Genetic Alphabet
AGC-128 uses four genetic symbols mapped from 2-bit pairs:

```
11 → G
00 → C
10 → A
01 → T
```

Every ASCII character (8 bits) becomes four genetic symbols.

FASTA Compatibility
The DNA output can be saved as a `.fasta` file and later decoded back into text.

This makes AGC-128 suitable for:

- digital archiving
- DNA-like storage experiments
- long-term data preservation
- bioinformatics-style workflows

Why It Works So Well
AGC-128 is powerful because the **structure itself** enforces stability.
No heavy algorithms, no compression, no GPU, no dependencies.

It is inspired by biological DNA:
- small alphabet
- simple rules
- strong internal consistency
- natural error detection
- predictable rhythm

This allows the entire system to remain tiny (≈15 KB) yet extremely robust.

Example

Input:
```
Hello!
```

Encoded DNA:
```
T C G A  T C G A  T C G G  T C G G  T C A A  C C A G
```

Decoded Back:
```
Hello!
```

Perfect 1:1 recovery.

Project Status
- **AGC_128_v.1** — stable core
- **AGC_128_v.2 (planned)** — Unicode, Cyrillic, binary files, metadata, extended genome logic

---

Notes
This README represents the **first official documentation** of AGC-128, created directly from the original chat where the concept was born, tested, and refined.

```
## 🧬 **AGC-128 = Adaptive Genetic Code — 128-bit ASCII bridge**
import tkinter as tk
from tkinter import filedialog, simpledialog, messagebox

# ========================
# GLOBAL STATE
# ========================
current_encoded_nucleotide_sequence = []

# ========================
# AGC-128 CORE TABLES
# ========================

# 00 -> C, 01 -> T, 10 -> A, 11 -> G
nuc_to_int = {
    'C': 0,
    'T': 1,
    'A': 2,
    'G': 3
}
int_to_nuc = {v: k for k, v in nuc_to_int.items()}

# ========================
# ENCODING: TEXT → NUCLEOTIDES
# ========================

def string_to_nucleotide_sequence(text):
    '''
    Всеки символ -> ASCII (8 бита) -> 4 двойки бита -> 4 нуклеотида.
    '''
    seq = []
    for ch in text:
        ascii_val = ord(ch)
        # Extract 2-bit chunks
        b1 = (ascii_val >> 6) & 0b11 # Most significant 2 bits
        b2 = (ascii_val >> 4) & 0b11
        b3 = (ascii_val >> 2) & 0b11
        b4 = ascii_val & 0b11       # Least significant 2 bits
        seq.extend([int_to_nuc[b1], int_to_nuc[b2], int_to_nuc[b3],
int_to_nuc[b4]])
    return seq

# ========================
# CHECKSUM (2-NUC) - FIXED
# ========================

def calculate_genetic_checksum(nucleotide_sequence):
    '''
    Calculates a genetic checksum for a given nucleotide sequence.
    The checksum is based on the sum of 2-bit integer representations
    of nucleotides, modulo 16, encoded as two nucleotides.
    This uses the previously working logic (total_sum % 16).
    '''
```

```
    total_sum = 0
    for nuc in nucleotide_sequence:
        total_sum += nuc_to_int.get(nuc, 0) # Use .get with default 0 for safety

    checksum_value = total_sum % 16 # Checksum is a value between 0 and 15 (4-bit
value)

    # Convert checksum value to 4-bit binary string (e.g., 0 -> "0000", 15 ->
"1111")
    checksum_binary = f"{checksum_value:04b}"

    # Convert 4-bit binary string to two nucleotides using int_to_nuc
    checksum_nuc1_int = int(checksum_binary[0:2], 2) # Convert "00" to 0, "01" to
1, etc.
    checksum_nuc2_int = int(checksum_binary[2:4], 2)

    checksum_nuc1 = int_to_nuc[checksum_nuc1_int]
    checksum_nuc2 = int_to_nuc[checksum_nuc2_int]

    return [checksum_nuc1, checksum_nuc2]

def add_genetic_checksum(seq):
    '''
    Appends the calculated genetic checksum to a copy of the original nucleotide
sequence.
    '''
    checksum = calculate_genetic_checksum(seq)
    sequence_with_checksum = list(seq) # Create a copy
    sequence_with_checksum.extend(checksum)
    return sequence_with_checksum

def verify_genetic_checksum(seq):
    '''
    Verifies the genetic checksum of a sequence.
    Assumes the last two nucleotides are the checksum.
    '''
    if len(seq) < 2:
        return False
    data = seq[:-2] # The original data part
    checksum = seq[-2:] # The provided checksum part
    expected = calculate_genetic_checksum(data)
    return checksum == expected

# ========================
# DECODING: NUCLEOTIDES → TEXT
# ========================

def decode_nucleotide_sequence_to_string(nucleotide_sequence):
    '''
    4 нуклеотида -> 4x2 бита -> 8-битов ASCII.
    '''
    decoded_chars = []
    for i in range(0, len(nucleotide_sequence), 4):
        chunk = nucleotide_sequence[i:i+4]
```

```
        if len(chunk) != 4:
            # Warning already handled in GUI if length mismatch
            break

        # Convert each nucleotide to its 2-bit integer representation
        b1 = nuc_to_int[chunk[0]]
        b2 = nuc_to_int[chunk[1]]
        b3 = nuc_to_int[chunk[2]]
        b4 = nuc_to_int[chunk[3]]

        # Combine the four 2-bit integers to form a single 8-bit integer
        ascii_val = (b1 << 6) | (b2 << 4) | (b3 << 2) | b4
        decoded_chars.append(chr(ascii_val))
    return "".join(decoded_chars)

# ========================
# FASTA
# ========================

def generate_fasta_string(seq, header, line_width=60):
    out_lines = [f">{header}"]
    for i in range(0, len(seq), line_width):
        out_lines.append("".join(seq[i:i+line_width]))
    return "\n".join(out_lines) + "\n"

# ========================
# DUMMY VISUALIZATION (placeholder) - IMPROVED MESSAGE
# ========================

def visualize_nucleotide_sequence(seq, title="AGC-128 Sequence",
checksum_length=0, error_index=-1):
    '''
    Плейсхолдър – няма графика, само показва информация.
    '''
    info_message = f"Title: {title}\n"
    info_message += f"Sequence Length: {len(seq)} nucleotides\n"
    if checksum_length > 0:
        info_message += f"Checksum Length: {checksum_length} nucleotides\n"
        info_message += f"Checksum Nucleotides: {'
'.join(seq[-checksum_length:])}\n"
    if error_index != -1:
        info_message += f"Highlighted Error at index: {error_index} (nucleotide:
{seq[error_index]})\n"
    info_message += "\n(Visualization functionality is a placeholder in this
Colab environment. "\
                    "Run locally for full matplotlib visualization.)"

    messagebox.showinfo(
        "Visualize Sequence (Placeholder)",
        info_message
    )

# ========================
# GUI
```

```
# ========================

def setup_gui():
    global current_encoded_nucleotide_sequence

    root = tk.Tk()
    root.title("AGC-128 Notepad")

    text_widget = tk.Text(root, wrap='word')
    text_widget.pack(expand=True, fill='both')

    menubar = tk.Menu(root)
    root.config(menu=menubar)

    # ---------- FILE ----------
    file_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="File", menu=file_menu)

    def open_file():
        global current_encoded_nucleotide_sequence
        file_path = filedialog.askopenfilename(
            filetypes=[("Text files", "*.txt"), ("All files", "*.* затем")]
        )
        if file_path:
            with open(file_path, 'r', encoding='utf-8') as file:
                content = file.read()
            text_widget.delete("1.0", tk.END)
            text_widget.insert(tk.END, content)
            current_encoded_nucleotide_sequence.clear()

    def save_file():
        file_path = filedialog.asksaveasfilename(
            defaultextension=".txt",
            filetypes=[("Text files", "*.txt"), ("All files", "*.* затем")]
        )
        if file_path:
            content = text_widget.get("1.0", tk.END)
            with open(file_path, 'w', encoding='utf-8') as file:
                file.write(content)

    file_menu.add_command(label="Open", command=open_file)
    file_menu.add_command(label="Save", command=save_file)
    file_menu.add_separator()
    file_menu.add_command(label="Exit", command=root.quit)

    # ---------- ENCODE ----------
    encode_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Encode", menu=encode_menu)

    def encode_to_fasta_action():
        global current_encoded_nucleotide_sequence

        input_text = text_widget.get("1.0", tk.END).strip()
        if not input_text:
```

```
            messagebox.showwarning("No Input", "Please enter text to encode in
the editor.")
            return

        fasta_id = simpledialog.askstring("FASTA Identifier", "Enter FASTA header
ID:")
        if not fasta_id:
            messagebox.showwarning("Missing ID", "FASTA identifier cannot be
empty.")
            return

        add_checksum = messagebox.askyesno("Checksum Option", "Do you want to add
a genetic checksum?")

        try:
            nucleotide_sequence_temp = string_to_nucleotide_sequence(input_text)
            if add_checksum:
                processed_sequence =
add_genetic_checksum(nucleotide_sequence_temp)
            else:
                processed_sequence = nucleotide_sequence_temp

            current_encoded_nucleotide_sequence[:] = processed_sequence

            fasta_output = generate_fasta_string(
                processed_sequence,
                fasta_id,
                line_width=60
            )

            save_path = filedialog.asksaveasfilename(
                defaultextension=".fasta",
                filetypes=[("FASTA files", "*.fasta"), ("All files", "*.*
затем")],
                title="Save Encoded FASTA As"
            )
            if save_path:
                with open(save_path, 'w', encoding='utf-8') as f:
                    f.write(fasta_output)
                messagebox.showinfo("Success", f"FASTA encoded and saved to
{save_path}")
            else:
                messagebox.showinfo("Cancelled", "FASTA save operation
cancelled.")
        except Exception as e:
            messagebox.showerror("Encoding Error", f"An error occurred during
encoding: {e}")

    encode_menu.add_command(label="Encode to AGC-128 FASTA",
command=encode_to_fasta_action)

    # ---------- DECODE ----------
    decode_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Decode", menu=decode_menu)
```

```
    def load_and_decode_fasta_action():
        global current_encoded_nucleotide_sequence

        file_path = filedialog.askopenfilename(
            filetypes=[("FASTA files", "*.fasta"), ("All files", "*.* затем")]
        )
        if not file_path:
            messagebox.showinfo("Cancelled", "FASTA load operation cancelled.")
            return

        try:
            with open(file_path, 'r', encoding='utf-8') as file:
                content = file.read()

            lines = content.splitlines()
            if not lines or not lines[0].startswith('>'):
                messagebox.showwarning(
                    "Invalid FASTA",
                    "Selected file does not appear to be a valid FASTA format
(missing header)."
                )
                return

            # Extract sequence, ignore header(s), keep only A/T/G/C
            seq_raw = "".join(line.strip() for line in lines[1:] if not
line.startswith(">"))
            valid = {'A', 'T', 'G', 'C'}
            extracted_nucs_list = [c for c in seq_raw if c in valid]

            if not extracted_nucs_list:
                messagebox.showwarning("Empty Sequence", "No nucleotide sequence
found in the FASTA file.")
                return

            current_encoded_nucleotide_sequence[:] = extracted_nucs_list

            sequence_to_decode = extracted_nucs_list
            checksum_info = ""

            # Check for checksum based on length: if length % 4 == 2, it
indicates a 2-nucleotide checksum
            if len(extracted_nucs_list) >= 2 and len(extracted_nucs_list) % 4 ==
2:
                ask_checksum = messagebox.askyesno(
                    "Checksum Detected?",
                    "The sequence length suggests a 2-nucleotide checksum.\n"
                    "Do you want to verify and remove it before decoding?"
                )
                if ask_checksum:
                    is_valid_checksum =
verify_genetic_checksum(extracted_nucs_list)
                    checksum_info = f"\nChecksum valid: {is_valid_checksum}"
                    if is_valid_checksum:
```

```
                        messagebox.showinfo("Checksum Status", f"Checksum is
valid!{checksum_info}")
                    else:
                        messagebox.showwarning(
                            "Checksum Status",
                            f"Checksum is INVALID! Data may be
corrupted.{checksum_info}"
                        )
                    sequence_to_decode = extracted_nucs_list[:-2] # Remove
checksum for decoding

            elif len(extracted_nucs_list) % 4 != 0:
                messagebox.showwarning(
                    "Sequence Length Mismatch",
                    "The nucleotide sequence length is not a multiple of 4, nor
does it suggest a 2-nucleotide checksum.\n"
                    "Decoding might result in an incomplete last character."
                )

            decoded_text =
decode_nucleotide_sequence_to_string(sequence_to_decode)

            text_widget.delete("1.0", tk.END)
            text_widget.insert(tk.END, decoded_text)
            messagebox.showinfo("Decoding Success", f"FASTA file successfully
loaded and decoded!{checksum_info}")

        except Exception as e:
            messagebox.showerror("Decoding Error", f"An error occurred during
FASTA loading or decoding: {e}")

    decode_menu.add_command(label="Load and Decode AGC-128 FASTA",
command=load_and_decode_fasta_action)

    # ---------- TOOLS ----------
    tools_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Tools", menu=tools_menu)

    def verify_checksum_action():
        global current_encoded_nucleotide_sequence
        if not current_encoded_nucleotide_sequence:
            messagebox.showwarning("No Sequence", "No encoded nucleotide sequence
is currently loaded or generated.")
            return

        if len(current_encoded_nucleotide_sequence) >= 2 and
len(current_encoded_nucleotide_sequence) % 4 == 2:
            is_valid =
verify_genetic_checksum(current_encoded_nucleotide_sequence)
            messagebox.showinfo("Checksum Verification", f"Checksum valid:
{is_valid}")
        else:
            messagebox.showwarning(
                "No Checksum Detected",
```

```
                "The current sequence length does not suggest a 2-nucleotide
checksum.\n"\
                "Checksum verification requires the sequence to be 'data + 2
checksum nucleotides'."
            )

    def visualize_action():
        global current_encoded_nucleotide_sequence
        if not current_encoded_nucleotide_sequence:
            messagebox.showwarning(
                "No Sequence",
                "No encoded nucleotide sequence is currently loaded or generated
to visualize."
            )
            return

        checksum_len = 0
        if len(current_encoded_nucleotide_sequence) >= 2 and
len(current_encoded_nucleotide_sequence) % 4 == 2:
            checksum_len = 2

        try:
            visualize_nucleotide_sequence(
                current_encoded_nucleotide_sequence,
                "Current AGC-128 Sequence",
                checksum_length=checksum_len
            )
        except Exception as e:
            messagebox.showerror("Visualization Error", f"An error occurred
during visualization: {e}")

    tools_menu.add_command(label="Verify Checksum",
command=verify_checksum_action)
    tools_menu.add_command(label="Visualize Sequence", command=visualize_action)

    root.mainloop()

# ========================
# MAIN
# ========================

if __name__ == "__main__":
    try:
        setup_gui()
    except tk.TclError as e:
        print(f"Error: {e}")
        print("Tkinter GUI cannot be displayed in this environment (e.g., Google
Colab).")
        print("Run this script locally on your computer with a graphical
interface.")

# AGC-128 v.2 — Adaptive Genetic Code 128 Unicode Edition

## Official README (Unicode Edition)
```

---

1. Overview
AGC-128 v2 extends the lightweight, fully reversible, DNA-inspired text encoding system to support the full Unicode character set. Building upon the core principles of AGC-128 v1 (ASCII), version 2 introduces a variable-length encoding scheme based on UTF-8 bytes, ensuring 1:1 lossless transformation for any character, from basic ASCII to complex emojis.

Like its predecessor, AGC-128 v2 requires no external libraries (for core encoding/decoding) and aims for efficiency and robustness, translating `Unicode Text` into `Genetic Sequences` and back.

---

2. What the Program Does (v2)
AGC-128 v2 performs a complete reversible transformation:

```
Unicode Text → UTF-8 Bytes → Length Gene + Genetic Bytes → A/T/G/C DNA Sequence
```

and back:

```
DNA Sequence → Genetic Bytes + Length Gene → UTF-8 Bytes → Unicode Text
```

This system preserves:
- Any Unicode character (including ASCII, Cyrillic, CJK, Emojis, Symbols)
- Letters, numbers, punctuation, whitespace
- Structured blocks and FASTA-formatted sequences

If you encode Unicode text and decode it again, the output will match the original **exactly**, character-for-character.

---

3. Key Features

3.1. Full Unicode Support
Leverages UTF-8 encoding to support all characters in the Unicode standard, from 1-byte ASCII to 4-byte emojis.

3.2. Variable-Length Genetic Encoding
Each Unicode character is encoded with a `Length Prefix Gene` followed by the appropriate number of 4-nucleotide byte representations.

3.3. Full Reversibility
Every Unicode character is reversibly transformed. Decoding restores the exact original text with zero corruption.

3.4. Self-Checking Genetic Structure (Inherited from v1)

AGC-128 maintains its three biological-style integrity rules:
- **Sum-2 Rule**: Each 2-bit gene has a total bit-sum of 2.
- **No-Triple Rule**: No `111` or `000` patterns allowed.
- **Deterministic-Next-Bit Rule**: Predictable bit sequences (`11` -> `0`, `00` -> `1`).

3.5. FASTA Compatibility
The DNA output can be saved as a `.fasta` file, making it suitable for digital archiving, DNA-like storage experiments, and bioinformatics-style workflows.

---

4. Genetic Alphabet (from v1)
AGC-128 uses four genetic symbols mapped from 2-bit pairs:

```
11 → G
00 → C
10 → A
01 → T
```

---

5. AGC-128 v2 Core Principles

5.1. UTF-8 as Foundation
Unicode characters are first converted to their UTF-8 byte representation. This handles the variable length nature of Unicode efficiently (1 to 4 bytes per character).

5.2. Length Prefix Gene
Each encoded Unicode character begins with a special single-nucleotide `Length Gene` that indicates how many UTF-8 bytes follow. This allows the decoder to know exactly how many subsequent nucleotides to read for the character's data.

UTF-8 Length	Number of Bytes	2-bit Marker	Length Gene
1 byte	ASCII	00	C
2 bytes	Cyrillic	01	T
3 bytes	Multi-byte	10	A
4 bytes	Emojis	11	G

5.3. Byte Encoding (from v1)
Each individual UTF-8 byte (0-255) is then encoded into four 2-bit nucleotide genes, exactly as in AGC-128 v1.

Thus, a Unicode character's genetic sequence is: `[Length Gene] + [4 genes per byte]`.

- **1-byte UTF-8 (ASCII)** → `C` + 4 genes = 5 nucleotides
- **2-bytes UTF-8 (e.g., Cyrillic)** → `T` + 8 genes = 9 nucleotides
- **3-bytes UTF-8 (e.g., Chinese)** → `A` + 12 genes = 13 nucleotides
- **4-bytes UTF-8 (e.g., Emojis)** → `G` + 16 genes = 17 nucleotides

5.4. Decoding Algorithm
1. Read the first nucleotide: this is the `Length Gene`.
2. Determine the number of UTF-8 bytes (`N`) from the `Length Gene` using `REV_LENGTH_MAP`.
3. Read the next `4 * N` nucleotides: these are the data genes.
4. Convert these data genes back into `N` bytes.
5. Decode the `N` bytes into the original Unicode character using UTF-8.
6. Repeat until the entire sequence is processed.

5.5. Compatibility with AGC-128 v1
ASCII characters (1-byte UTF-8) encoded with AGC-128 v2 will have a `Length Gene` of `C`. This design ensures:
- v1 sequences can be recognized (though v2 encoding is slightly longer for ASCII due to the length gene).
- A v2 decoder can correctly read v1 encoded ASCII sequences (assuming the v1 sequence is prefixed with `C` or the decoder intelligently handles it).
- A v1 decoder will not be able to read v2 encoded sequences (expected).

5.6. Genetic Checksum (from v1)
An optional 2-nucleotide genetic checksum can be appended to the entire sequence to verify data integrity. It works identically to v1 (sum of 2-bit values modulo 16).

---

6. Examples

6.1. Encoding the Cyrillic character `Ж`
- UTF-8 encoding of `Ж` is `D0 96` (2 bytes).
- `Length Gene` for 2 bytes is `T`.
- `D0` (11010000) → `G T C C`
- `96` (10010110) → `A T T G`
- **Resulting sequence**: `T G T C C A T T G` (1 Length Gene + 8 data genes = 9 nucleotides)

6.2. Encoding the emoji `☺`
- UTF-8 encoding of `☺` is `F0 9F 99 82` (4 bytes).
- `Length Gene` for 4 bytes is `G`.
- Followed by 16 data genes (4 genes per byte).
- **Resulting sequence**: `G` + 16 data genes (17 nucleotides total)

6.3. Encoding the text `Hello, свят!😊`
```
Hello, свят!😊
```

- `H` (ASCII) → `C` + 4 genes
- `e` (ASCII) → `C` + 4 genes
- `л` (Cyrillic) → `T` + 8 genes
- `т` (Cyrillic) → `T` + 8 genes
- `😊` (Emoji) → `G` + 16 genes

---

7. Code Structure (Pseudocode)

`encode_string_to_unicode_tagc_sequence(text)`:
```python
encoded_sequence = []
for char in text:
    utf8_bytes = char.encode('utf-8')
    length_gene = LENGTH_MAP[len(utf8_bytes)]
    encoded_sequence.append(length_gene)
    for byte_val in utf8_bytes:
        encoded_sequence.extend(byte_to_tagc_v2(byte_val))
return encoded_sequence
```

`decode_unicode_tagc_sequence_to_string(tagc_sequence)`:
```python
decoded_chars = []
index = 0
while index < len(tagc_sequence):
    length_gene = tagc_sequence[index]
    num_bytes = REV_LENGTH_MAP[length_gene]
    char_chunk_length = 1 + (num_bytes * 4)
    char_tagc_chunk = tagc_sequence[index : index + char_chunk_length]

    # Reconstruct bytes from data_nucleotides
    byte_array = bytearray()
    data_nucleotides = char_tagc_chunk[1:]
    for i in range(0, len(data_nucleotides), 4):
        byte_array.append(tagc_to_byte_v2(data_nucleotides[i:i+4]))

    decoded_chars.append(byte_array.decode('utf-8'))
    index += char_chunk_length
return "".join(decoded_chars)
```

---

8. Next Steps
- **Comprehensive Documentation**: Detailed specification of AGC-128 v2.
- **Official Naming**: Finalize the standard's name (e.g., AGC-UTF, AGC-256).
- **Python Package**: Create a distributable Python package.
- **GitHub Repository**: Establish a dedicated GitHub page for development and community contributions.
- **Online Converter**: Develop an online tool for easy encoding/decoding demonstration.

```
import tkinter as tk
from tkinter import filedialog, simpledialog, messagebox

# ========================
# GLOBAL STATE
# ========================
current_encoded_nucleotide_sequence = []
```

```
# ========================
# AGC-128 CORE TABLES
# ========================

# 00 -> C, 01 -> T, 10 -> A, 11 -> G
nuc_to_int = {
    'C': 0,
    'T': 1,
    'A': 2,
    'G': 3
}
int_to_nuc = {v: k for k, v in nuc_to_int.items()}

# For V2 Unicode
LENGTH_MAP = {
    1: 'C',  # 1 byte UTF-8 (ASCII)
    2: 'T',  # 2 bytes UTF-8 (e.g., Cyrillic)
    3: 'A',  # 3 bytes UTF-8 (other multi-byte)
    4: 'G'   # 4 bytes UTF-8 (emojis)
}
REV_LENGTH_MAP = {v: k for k, v in LENGTH_MAP.items()}

# Map 2-bit strings to nucleotides for V2 byte-level encoding
bit_to_nuc = {
    '00': 'C',
    '01': 'T',
    '10': 'A',
    '11': 'G'
}

# ========================
# ENCODING: TEXT → NUCLEOTIDES
# ========================

# V1 ASCII Encoding
def string_to_nucleotide_sequence_v1(text):
    '''
    Всеки символ -> ASCII (8 бита) -> 4 двойки бита -> 4 нуклеотида.
    '''
    seq = []
    for ch in text:
        ascii_val = ord(ch)
        # Extract 2-bit chunks
        b1 = (ascii_val >> 6) & 0b11  # Most significant 2 bits
        b2 = (ascii_val >> 4) & 0b11
        b3 = (ascii_val >> 2) & 0b11
        b4 = ascii_val & 0b11        # Least significant 2 bits
        seq.extend([
            int_to_nuc[b1],
            int_to_nuc[b2],
            int_to_nuc[b3],
            int_to_nuc[b4]
        ])
```

```
    return seq

# V2 Unicode Helper Functions (byte-level)
def byte_to_tagc_v2(byte):
    '''
    Converts a single byte (0-255) into its corresponding 4 TAGC nucleotides.
    '''
    bits = f"{byte:08b}"
    tagc_nucleotides = []
    for i in range(0, 8, 2):
        two_bit_chunk = bits[i:i+2]
        tagc_nucleotides.append(bit_to_nuc[two_bit_chunk])
    return tagc_nucleotides

# V2 Unicode Encoding
def encode_unicode_char_to_tagc(unicode_char):
    '''
    Converts a single Unicode character into a TAGC nucleotide sequence,
    prefixed with a Length Gene.
    '''
    utf8_bytes = unicode_char.encode('utf-8')
    num_bytes = len(utf8_bytes)
    encoded_sequence = []

    if num_bytes not in LENGTH_MAP:
        raise ValueError(f"Unsupported UTF-8 byte length: {num_bytes} for
character '{unicode_char}'")

    length_gene = LENGTH_MAP[num_bytes]
    encoded_sequence.append(length_gene)

    for byte_val in utf8_bytes:
        tagc_nucleotides = byte_to_tagc_v2(byte_val)
        encoded_sequence.extend(tagc_nucleotides)

    return encoded_sequence

def encode_string_to_unicode_tagc_sequence(input_string):
    '''
    Encodes an entire string into a Unicode TAGC nucleotide sequence.
    '''
    full_tagc_sequence = []
    for char in input_string:
        char_tagc = encode_unicode_char_to_tagc(char)
        full_tagc_sequence.extend(char_tagc)
    return full_tagc_sequence

# ========================
# CHECKSUM (2-NUC) - FIXED
# ========================

def calculate_genetic_checksum(nucleotide_sequence):
    '''
    Calculates a genetic checksum for a given nucleotide sequence.
```

```
    The checksum is based on the sum of 2-bit integer representations
    of nucleotides, modulo 16, encoded as two nucleotides.
    '''
    total_sum = 0
    for nuc in nucleotide_sequence:
        total_sum += nuc_to_int.get(nuc, 0)  # Use .get with default 0 for safety

    checksum_value = total_sum % 16  # Checksum is a value between 0 and 15
(4-bit value)

    # Convert checksum value to 4-bit binary string (e.g., 0 -> "0000", 15 ->
"1111")
    checksum_binary = f"{checksum_value:04b}"

    # Convert 4-bit binary string to two nucleotides using int_to_nuc
    checksum_nuc1_int = int(checksum_binary[0:2], 2)
    checksum_nuc2_int = int(checksum_binary[2:4], 2)

    checksum_nuc1 = int_to_nuc[checksum_nuc1_int]
    checksum_nuc2 = int_to_nuc[checksum_nuc2_int]

    return [checksum_nuc1, checksum_nuc2]

def add_genetic_checksum(seq):
    '''
    Appends the calculated genetic checksum to a copy of the original nucleotide
sequence.
    '''
    checksum = calculate_genetic_checksum(seq)
    sequence_with_checksum = list(seq)  # Create a copy
    sequence_with_checksum.extend(checksum)
    return sequence_with_checksum

def verify_genetic_checksum(seq):
    '''
    Verifies the genetic checksum of a sequence.
    Assumes the last two nucleotides are the checksum.
    '''
    if len(seq) < 2:
        return False
    data = seq[:-2]        # The original data part
    checksum = seq[-2:]    # The provided checksum part
    expected = calculate_genetic_checksum(data)
    return checksum == expected

# ========================
# DECODING: NUCLEOTIDES → TEXT
# ========================

# V1 ASCII Decoding
def decode_nucleotide_sequence_to_string_v1(nucleotide_sequence):
    '''
    4 нуклеотида -> 4x2 бита -> 8-битов ASCII.
    '''
```

```
    decoded_chars = []
    for i in range(0, len(nucleotide_sequence), 4):
        chunk = nucleotide_sequence[i:i+4]
        if len(chunk) != 4:
            # Warning already handled in GUI if length mismatch
            break

        # Convert each nucleotide to its 2-bit integer representation
        b1 = nuc_to_int[chunk[0]]
        b2 = nuc_to_int[chunk[1]]
        b3 = nuc_to_int[chunk[2]]
        b4 = nuc_to_int[chunk[3]]

        # Combine the four 2-bit integers to form a single 8-bit integer
        ascii_val = (b1 << 6) | (b2 << 4) | (b3 << 2) | b4
        decoded_chars.append(chr(ascii_val))
    return "".join(decoded_chars)

# V2 Unicode Helper Functions (byte-level)
def tagc_to_byte_v2(nucleotides):
    '''
    Converts 4 TAGC nucleotides back into a single byte.
    '''
    if len(nucleotides) != 4:
        raise ValueError("Input must be a list of exactly 4 nucleotides.")

    binary_string = ""
    for nuc in nucleotides:
        int_value = nuc_to_int[nuc]
        binary_string += f"{int_value:02b}"

    byte_value = int(binary_string, 2)
    return byte_value

# V2 Unicode Decoding
def decode_tagc_to_unicode_char(tagc_sequence_chunk):
    '''
    Decodes a chunk of TAGC nucleotides representing a single encoded Unicode
character
    back into the original Unicode character.
    '''
    if not tagc_sequence_chunk:
        raise ValueError("Input tagc_sequence_chunk cannot be empty.")

    length_gene = tagc_sequence_chunk[0]

    if length_gene not in REV_LENGTH_MAP:
        raise ValueError(f"Invalid Length Gene '{length_gene}' found.")
    num_bytes = REV_LENGTH_MAP[length_gene]

    expected_length = 1 + (num_bytes * 4)

    if len(tagc_sequence_chunk) != expected_length:
        raise ValueError(
```

```
            f"Mismatch in TAGC sequence chunk length. Expected {expected_length}
nucleotides "
            f"but got {len(tagc_sequence_chunk)}. (Length Gene: {length_gene},
num_bytes: {num_bytes}) "
            f"Full chunk: {tagc_sequence_chunk}"
        )

    data_nucleotides = tagc_sequence_chunk[1:]
    byte_array = bytearray()

    for i in range(0, len(data_nucleotides), 4):
        nuc_chunk = data_nucleotides[i:i+4]
        decoded_byte = tagc_to_byte_v2(nuc_chunk)
        byte_array.append(decoded_byte)

    decoded_char = byte_array.decode('utf-8')
    return decoded_char

def decode_unicode_tagc_sequence_to_string(tagc_sequence):
    '''
    Decodes an entire Unicode TAGC nucleotide sequence back into a string.
    '''
    decoded_chars = []
    current_index = 0

    while current_index < len(tagc_sequence):
        length_gene = tagc_sequence[current_index]

        if length_gene not in REV_LENGTH_MAP:
            raise ValueError(f"Invalid Length Gene '{length_gene}' at index
{current_index}.")
        num_bytes = REV_LENGTH_MAP[length_gene]

        char_chunk_length = 1 + (num_bytes * 4)

        char_tagc_chunk = tagc_sequence[current_index:current_index +
char_chunk_length]

        if len(char_tagc_chunk) != char_chunk_length:
            raise ValueError(
                f"Incomplete TAGC sequence at index {current_index}. "
                f"Expected {char_chunk_length} nucleotides, but found
{len(char_tagc_chunk)}."
            )

        decoded_char = decode_tagc_to_unicode_char(char_tagc_chunk)
        decoded_chars.append(decoded_char)

        current_index += char_chunk_length

    return "".join(decoded_chars)

# ========================
# FASTA
```

```
# ========================

def generate_fasta_string(seq, header, line_width=60):
    out_lines = [f">{header}"]
    for i in range(0, len(seq), line_width):
        out_lines.append("".join(seq[i:i+line_width]))
    return "\n".join(out_lines) + "\n"

# ========================
# DUMMY VISUALIZATION (placeholder)
# ========================

def visualize_nucleotide_sequence(seq, title="AGC-128 Sequence",
checksum_length=0, error_index=-1):
    '''
    Плейсхолдър – няма графика, само показва информация.
    '''
    info_message = f"Title: {title}\n"
    info_message += f"Sequence Length: {len(seq)} nucleotides\n"
    if checksum_length > 0:
        info_message += f"Checksum Length: {checksum_length} nucleotides\n"
        info_message += f"Checksum Nucleotides: {'
'.join(seq[-checksum_length:])}\n"
    if error_index != -1:
        info_message += f"Highlighted Error at index: {error_index} (nucleotide:
{seq[error_index]})\n"
    info_message += (
        "\n(Visualization functionality is a placeholder in this environment. "
        "Run locally for full matplotlib visualization.)"
    )

    messagebox.showinfo(
        "Visualize Sequence (Placeholder)",
        info_message
    )

# ========================
# GUI
# ========================

def setup_gui():
    global current_encoded_nucleotide_sequence

    root = tk.Tk()
    root.title("AGC-128 Notepad")

    # Frame for encoding version selection
    version_frame = tk.Frame(root)
    version_frame.pack(pady=5, anchor='w')

    tk.Label(version_frame, text="Encoding/Decoding Version:").pack(side=tk.LEFT)
    version_var = tk.StringVar(value="v1_ascii")  # Default to v1 (ASCII)

    v1_radio = tk.Radiobutton(version_frame, text="v1 (ASCII)",
```

```
variable=version_var, value="v1_ascii")
    v1_radio.pack(side=tk.LEFT, padx=5)

    v2_radio = tk.Radiobutton(version_frame, text="v2 (Unicode)",
variable=version_var, value="v2_unicode")
    v2_radio.pack(side=tk.LEFT, padx=5)

    text_widget = tk.Text(root, wrap='word')
    text_widget.pack(expand=True, fill='both')

    menubar = tk.Menu(root)
    root.config(menu=menubar)

    # ---------- FILE ----------
    file_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="File", menu=file_menu)

    def open_file():
        global current_encoded_nucleotide_sequence
        file_path = filedialog.askopenfilename(
            filetypes=[("Text files", "*.txt"), ("All files", "*.* затем")]
        )
        if file_path:
            with open(file_path, 'r', encoding='utf-8') as file:
                content = file.read()
            text_widget.delete("1.0", tk.END)
            text_widget.insert(tk.END, content)
            current_encoded_nucleotide_sequence.clear()

    def save_file():
        file_path = filedialog.asksaveasfilename(
            defaultextension=".txt",
            filetypes=[("Text files", "*.txt"), ("All files", "*.* затем")]
        )
        if file_path:
            content = text_widget.get("1.0", tk.END)
            with open(file_path, 'w', encoding='utf-8') as file:
                file.write(content)

    file_menu.add_command(label="Open", command=open_file)
    file_menu.add_command(label="Save", command=save_file)
    file_menu.add_separator()
    file_menu.add_command(label="Exit", command=root.quit)

    # ---------- ENCODE ----------
    encode_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Encode", menu=encode_menu)

    def encode_to_fasta_action():
        global current_encoded_nucleotide_sequence

        input_text = text_widget.get("1.0", tk.END).strip()
        if not input_text:
            messagebox.showwarning("No Input", "Please enter text to encode in
```

```
the editor.")
            return

        fasta_id = simpledialog.askstring("FASTA Identifier", "Enter FASTA header
ID:")
        if not fasta_id:
            messagebox.showwarning("Missing ID", "FASTA identifier cannot be
empty.")
            return

        add_checksum = messagebox.askyesno("Checksum Option", "Do you want to add
a genetic checksum?")

        try:
            selected_version = version_var.get()
            if selected_version == "v1_ascii":
                nucleotide_sequence_temp =
string_to_nucleotide_sequence_v1(input_text)
            else:  # v2_unicode
                nucleotide_sequence_temp =
encode_string_to_unicode_tagc_sequence(input_text)

            if add_checksum:
                processed_sequence =
add_genetic_checksum(nucleotide_sequence_temp)
            else:
                processed_sequence = nucleotide_sequence_temp

            current_encoded_nucleotide_sequence[:] = processed_sequence

            fasta_output = generate_fasta_string(
                processed_sequence,
                fasta_id,
                line_width=60
            )

            save_path = filedialog.asksaveasfilename(
                defaultextension=".fasta",
                filetypes=[("FASTA files", "*.fasta"), ("All files", "*.*
затем")],
                title="Save Encoded FASTA As"
            )
            if save_path:
                with open(save_path, 'w', encoding='utf-8') as f:
                    f.write(fasta_output)
                messagebox.showinfo("Success", f"FASTA encoded and saved to
{save_path}")
            else:
                messagebox.showinfo("Cancelled", "FASTA save operation
cancelled.")
        except Exception as e:
            messagebox.showerror("Encoding Error", f"An error occurred during
encoding: {e}")
```

```
    encode_menu.add_command(label="Encode to AGC-128 FASTA",
command=encode_to_fasta_action)

    # ---------- DECODE ----------
    decode_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Decode", menu=decode_menu)

    def load_and_decode_fasta_action():
        global current_encoded_nucleotide_sequence

        file_path = filedialog.askopenfilename(
            filetypes=[("FASTA files", "*.fasta"), ("All files", "*.* затем")]
        )
        if not file_path:
            messagebox.showinfo("Cancelled", "FASTA load operation cancelled.")
            return

        try:
            with open(file_path, 'r', encoding='utf-8') as file:
                content = file.read()

            lines = content.splitlines()
            if not lines or not lines[0].startswith('>'):
                messagebox.showwarning(
                    "Invalid FASTA",
                    "Selected file does not appear to be a valid FASTA format
(missing header)."
                )
                return

            # Extract sequence, ignore header(s), keep only A/T/G/C
            seq_raw = "".join(line.strip() for line in lines[1:] if not
line.startswith(">"))
            valid = {'A', 'T', 'G', 'C'}
            extracted_nucs_list = [c for c in seq_raw if c in valid]

            if not extracted_nucs_list:
                messagebox.showwarning("Empty Sequence", "No nucleotide sequence
found in the FASTA file.")
                return

            current_encoded_nucleotide_sequence[:] = extracted_nucs_list

            sequence_to_decode = list(extracted_nucs_list) # Use a copy to allow
modification
            checksum_info = ""

            # --- MODIFIED CHECKSUM HANDLING ---
            ask_if_checksum_present = messagebox.askyesno(
                "Checksum Query",
                "Is a 2-nucleotide genetic checksum expected at the end of this
sequence?"
            )
```

```
            if ask_if_checksum_present:
                if len(extracted_nucs_list) < 2:
                    messagebox.showwarning("Checksum Error", "Sequence is too
short to contain a 2-nucleotide checksum.")
                else:
                    is_valid_checksum =
verify_genetic_checksum(extracted_nucs_list)
                    checksum_info = f"\nChecksum valid: {is_valid_checksum}"
                    if is_valid_checksum:
                        messagebox.showinfo("Checksum Status", f"Checksum is
valid!{checksum_info}")
                        sequence_to_decode = extracted_nucs_list[:-2] # Remove
checksum for decoding
                    else:
                        messagebox.showwarning(
                            "Checksum Status",
                            f"Checksum is INVALID! Data may be
corrupted.{checksum_info}\n"
                            "The checksum will NOT be removed before decoding as
it's invalid."
                        )
                        # If checksum is invalid, we don't automatically remove
it.
                        # The user might want to inspect the corrupted checksum
itself.
                        # The sequence_to_decode remains the full
extracted_nucs_list.
            # --- END MODIFIED CHECKSUM HANDLING ---

            # Determine the selected version for decoding
            selected_version = version_var.get()

            # Perform pre-decoding length check if no checksum was removed and
it's V1.
            # V2 has variable length chunks, so len % 4 is not a strong indicator
for end truncation.
            if not ask_if_checksum_present and selected_version == "v1_ascii" and
len(sequence_to_decode) % 4 != 0:
                 messagebox.showwarning(
                    "Sequence Length Mismatch (V1)",
                    "The V1 ASCII nucleotide sequence length is not a multiple of
4.\n"
                    "Decoding might result in an incomplete last character."
                )

            if selected_version == "v1_ascii":
                decoded_text =
decode_nucleotide_sequence_to_string_v1(sequence_to_decode)
            else: # v2_unicode
                decoded_text =
decode_unicode_tagc_sequence_to_string(sequence_to_decode)

            text_widget.delete("1.0", tk.END)
            text_widget.insert(tk.END, decoded_text)
```

```
            messagebox.showinfo("Decoding Success", f"FASTA file successfully
loaded and decoded!{checksum_info}")

        except ValueError as ve: # Catch specific ValueError from decoding
functions
            messagebox.showerror("Decoding Error (Data Integrity)", f"A data
integrity error occurred during decoding: {ve}\nThis might indicate a corrupted
sequence or incorrect encoding version/checksum assumption.")
        except Exception as e:
            messagebox.showerror("Decoding Error", f"An unexpected error occurred
during FASTA loading or decoding: {e}")

    decode_menu.add_command(label="Load and Decode AGC-128 FASTA",
command=load_and_decode_fasta_action)

    # ---------- TOOLS ----------
    tools_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Tools", menu=tools_menu)

    def verify_checksum_action():
        global current_encoded_nucleotide_sequence
        if not current_encoded_nucleotide_sequence:
            messagebox.showwarning("No Sequence", "No encoded nucleotide sequence
is currently loaded or generated.")
            return

        # --- MODIFIED CHECKSUM HANDLING IN VERIFY ACTION ---
        ask_if_checksum_present = messagebox.askyesno(
            "Checksum Query",
            "Is a 2-nucleotide genetic checksum expected at the end of the
current sequence?"
        )

        if ask_if_checksum_present:
            if len(current_encoded_nucleotide_sequence) < 2:
                messagebox.showwarning("Checksum Error", "The current sequence is
too short to contain a 2-nucleotide checksum.")
                return

            is_valid =
verify_genetic_checksum(current_encoded_nucleotide_sequence)
            messagebox.showinfo("Checksum Verification", f"Checksum valid:
{is_valid}")
        else:
            messagebox.showinfo("Checksum Information", "No checksum verification
performed as none was expected.")
        # --- END MODIFIED CHECKSUM HANDLING ---

    def visualize_action():
        global current_encoded_nucleotide_sequence
        if not current_encoded_nucleotide_sequence:
            messagebox.showwarning(
                "No Sequence",
                "No encoded nucleotide sequence is currently loaded or generated
```

```
to visualize."
            )
            return

        checksum_len = 0
        sequence_for_viz = list(current_encoded_nucleotide_sequence) # Make a
copy

        # --- MODIFIED CHECKSUM HANDLING IN VISUALIZE ACTION ---
        ask_if_checksum_present = messagebox.askyesno(
            "Checksum Query",
            "Is a 2-nucleotide genetic checksum expected at the end of the
current sequence for visualization?"
        )

        if ask_if_checksum_present:
            if len(current_encoded_nucleotide_sequence) < 2:
                messagebox.showwarning("Checksum Error", "Sequence is too short
to contain a 2-nucleotide checksum for visualization.")
            else:
                is_valid_checksum =
verify_genetic_checksum(current_encoded_nucleotide_sequence)
                if is_valid_checksum:
                    checksum_len = 2 # Indicate to visualization to highlight
last 2 nucs
                    messagebox.showinfo("Checksum Status", "Checksum is valid and
will be highlighted.")
                else:
                    messagebox.showwarning("Checksum Status", "Checksum is
INVALID. Will still highlight, but data may be corrupted.")
                    checksum_len = 2 # Still highlight, even if invalid
        # --- END MODIFIED CHECKSUM HANDLING ---

        try:
            visualize_nucleotide_sequence(
                sequence_for_viz, # Pass the original sequence, checksum_len will
handle highlighting
                "Current AGC-128 Sequence",
                checksum_length=checksum_len
            )
        except Exception as e:
            messagebox.showerror("Visualization Error", f"An error occurred
during visualization: {e}")

    tools_menu.add_command(label="Verify Checksum",
command=verify_checksum_action)
    tools_menu.add_command(label="Visualize Sequence", command=visualize_action)

    root.mainloop()

# ========================
# MAIN
# ========================
```

```
if __name__ == "__main__":
    try:
        setup_gui()
    except tk.TclError as e:
        print(f"Error: {e}")
        print("Tkinter GUI cannot be displayed in this environment (e.g., Google
Colab).")
        print("Run this script locally on your computer with a graphical
interface.")
"""

#AGC_128_Standart_notepad_v.3
import tkinter as tk
from tkinter import filedialog, simpledialog, messagebox
import sys

# ==========================
# GLOBAL STATE
# ==========================
current_encoded_nucleotide_sequence = []

# ==========================
# AGC-128 CORE TABLES
# ==========================

# 00 -> C, 01 -> T, 10 -> A, 11 -> G
nuc_to_int = {
    'C': 0,
    'T': 1,
    'A': 2,
    'G': 3
}
int_to_nuc = {v: k for k, v in nuc_to_int.items()}

# For V2 Unicode
LENGTH_MAP = {
    1: 'C',  # 1 byte UTF-8 (ASCII)
    2: 'T',  # 2 bytes UTF-8 (e.g., Cyrillic)
    3: 'A',  # 3 bytes UTF-8 (other multi-byte)
    4: 'G'   # 4 bytes UTF-8 (emojis)
}
REV_LENGTH_MAP = {v: k for k, v in LENGTH_MAP.items()}

# Map 2-bit strings to nucleotides for V2 byte-level encoding
bit_to_nuc = {
    '00': 'C',
    '01': 'T',
    '10': 'A',
    '11': 'G'
}

# ==========================
# ENCODING: TEXT → NUCLEOTIDES
# ==========================
```

```
# V1 ASCII Encoding
def string_to_nucleotide_sequence_v1(text):
    """
    Всеки символ -> ASCII (8 бита) -> 4 двойки бита -> 4 нуклеотида.
    """
    seq = []
    for ch in text:
        ascii_val = ord(ch)
        # Extract 2-bit chunks
        b1 = (ascii_val >> 6) & 0b11  # Most significant 2 bits
        b2 = (ascii_val >> 4) & 0b11
        b3 = (ascii_val >> 2) & 0b11
        b4 = ascii_val & 0b11        # Least significant 2 bits
        seq.extend([
            int_to_nuc[b1],
            int_to_nuc[b2],
            int_to_nuc[b3],
            int_to_nuc[b4]
        ])
    return seq

# V2 Unicode Helper Functions (byte-level)
def byte_to_tagc_v2(byte):
    """
    Converts a single byte (0-255) into its corresponding 4 TAGC nucleotides.
    """
    bits = f"{byte:08b}"
    tagc_nucleotides = []
    for i in range(0, 8, 2):
        two_bit_chunk = bits[i:i+2]
        tagc_nucleotides.append(bit_to_nuc[two_bit_chunk])
    return tagc_nucleotides

# V2 Unicode Encoding
def encode_unicode_char_to_tagc(unicode_char):
    """
    Converts a single Unicode character into a TAGC nucleotide sequence,
    prefixed with a Length Gene.
    """
    utf8_bytes = unicode_char.encode('utf-8')
    num_bytes = len(utf8_bytes)
    encoded_sequence = []

    if num_bytes not in LENGTH_MAP:
        raise ValueError(f"Unsupported UTF-8 byte length: {num_bytes} for
character '{unicode_char}'")

    length_gene = LENGTH_MAP[num_bytes]
    encoded_sequence.append(length_gene)

    for byte_val in utf8_bytes:
        tagc_nucleotides = byte_to_tagc_v2(byte_val)
        encoded_sequence.extend(tagc_nucleotides)
```

```
    return encoded_sequence

def encode_string_to_unicode_tagc_sequence(input_string):
    """
    Encodes an entire string into a Unicode TAGC nucleotide sequence.
    """
    full_tagc_sequence = []
    for char in input_string:
        char_tagc = encode_unicode_char_to_tagc(char)
        full_tagc_sequence.extend(char_tagc)
    return full_tagc_sequence

# ========================
# CHECKSUM (2-NUC) - FIXED
# ========================

def calculate_genetic_checksum(nucleotide_sequence):
    """
    Calculates a genetic checksum for a given nucleotide sequence.
    The checksum is based on the sum of 2-bit integer representations
    of nucleotides, modulo 16, encoded as two nucleotides.
    """
    total_sum = 0
    for nuc in nucleotide_sequence:
        total_sum += nuc_to_int.get(nuc, 0)  # Use .get with default 0 for safety

    checksum_value = total_sum % 16  # Checksum is a value between 0 and 15
(4-bit value)

    # Convert checksum value to 4-bit binary string (e.g., 0 -> "0000", 15 ->
"1111")
    checksum_binary = f"{checksum_value:04b}"

    # Convert 4-bit binary string to two nucleotides using int_to_nuc
    checksum_nuc1_int = int(checksum_binary[0:2], 2)
    checksum_nuc2_int = int(checksum_binary[2:4], 2)

    checksum_nuc1 = int_to_nuc[checksum_nuc1_int]
    checksum_nuc2 = int_to_nuc[checksum_nuc2_int]

    return [checksum_nuc1, checksum_nuc2]

def add_genetic_checksum(seq):
    """
    Appends the calculated genetic checksum to a copy of the original nucleotide
sequence.
    """
    checksum = calculate_genetic_checksum(seq)
    sequence_with_checksum = list(seq)  # Create a copy
    sequence_with_checksum.extend(checksum)
    return sequence_with_checksum

def verify_genetic_checksum(seq):
```

```
    """
    Verifies the genetic checksum of a sequence.
    Assumes the last two nucleotides are the checksum.
    """
    if len(seq) < 2:
        return False
    data = seq[:-2]        # The original data part
    checksum = seq[-2:]    # The provided checksum part
    expected = calculate_genetic_checksum(data)
    return checksum == expected

# ========================
# DECODING: NUCLEOTIDES → TEXT
# ========================

# V1 ASCII Decoding
def decode_nucleotide_sequence_to_string_v1(nucleotide_sequence):
    """
    4 нуклеотида -> 4x2 бита -> 8-битов ASCII.
    """
    decoded_chars = []
    for i in range(0, len(nucleotide_sequence), 4):
        chunk = nucleotide_sequence[i:i+4]
        if len(chunk) != 4:
            # Warning already handled in GUI if length mismatch
            break

        # Convert each nucleotide to its 2-bit integer representation
        b1 = nuc_to_int[chunk[0]]
        b2 = nuc_to_int[chunk[1]]
        b3 = nuc_to_int[chunk[2]]
        b4 = nuc_to_int[chunk[3]]

        # Combine the four 2-bit integers to form a single 8-bit integer
        ascii_val = (b1 << 6) | (b2 << 4) | (b3 << 2) | b4
        decoded_chars.append(chr(ascii_val))
    return "".join(decoded_chars)

# V2 Unicode Helper Functions (byte-level)
def tagc_to_byte_v2(nucleotides):
    """
    Converts 4 TAGC nucleotides back into a single byte.
    """
    if len(nucleotides) != 4:
        raise ValueError("Input must be a list of exactly 4 nucleotides.")

    binary_string = ""
    for nuc in nucleotides:
        int_value = nuc_to_int[nuc]
        binary_string += f"{int_value:02b}"

    byte_value = int(binary_string, 2)
    return byte_value
```

```
# V2 Unicode Decoding
def decode_tagc_to_unicode_char(tagc_sequence_chunk):
    """
    Decodes a chunk of TAGC nucleotides representing a single encoded Unicode
character
    back into the original Unicode character.
    """
    if not tagc_sequence_chunk:
        raise ValueError("Input tagc_sequence_chunk cannot be empty.")

    length_gene = tagc_sequence_chunk[0]

    if length_gene not in REV_LENGTH_MAP:
        raise ValueError(f"Invalid Length Gene '{length_gene}' found.")
    num_bytes = REV_LENGTH_MAP[length_gene]

    expected_length = 1 + (num_bytes * 4)

    if len(tagc_sequence_chunk) != expected_length:
        raise ValueError(
            f"Mismatch in TAGC sequence chunk length. Expected {expected_length}
nucleotides "
            f"but got {len(tagc_sequence_chunk)}. (Length Gene: {length_gene},
num_bytes: {num_bytes}) "
            f"Full chunk: {tagc_sequence_chunk}"
        )

    data_nucleotides = tagc_sequence_chunk[1:]
    byte_array = bytearray()

    for i in range(0, len(data_nucleotides), 4):
        nuc_chunk = data_nucleotides[i:i+4]
        decoded_byte = tagc_to_byte_v2(nuc_chunk)
        byte_array.append(decoded_byte)

    decoded_char = byte_array.decode('utf-8')
    return decoded_char

def decode_unicode_tagc_sequence_to_string(tagc_sequence):
    """
    Decodes an entire Unicode TAGC nucleotide sequence back into a string.
    """
    decoded_chars = []
    current_index = 0

    while current_index < len(tagc_sequence):
        length_gene = tagc_sequence[current_index]

        if length_gene not in REV_LENGTH_MAP:
            raise ValueError(f"Invalid Length Gene '{length_gene}' at index
{current_index}.")
        num_bytes = REV_LENGTH_MAP[length_gene]

        char_chunk_length = 1 + (num_bytes * 4)
```

```
        char_tagc_chunk = tagc_sequence[current_index:current_index +
char_chunk_length]

        if len(char_tagc_chunk) != char_chunk_length:
            raise ValueError(
                f"Incomplete TAGC sequence at index {current_index}. "
                f"Expected {char_chunk_length} nucleotides, but found
{len(char_tagc_chunk)}."
            )

        decoded_char = decode_tagc_to_unicode_char(char_tagc_chunk)
        decoded_chars.append(decoded_char)

        current_index += char_chunk_length

    return "".join(decoded_chars)

# ========================
# FASTA
# ========================

def generate_fasta_string(seq, header, line_width=60):
    out_lines = [f">{header}"]
    for i in range(0, len(seq), line_width):
        out_lines.append("".join(seq[i:i+line_width]))
    return "\n".join(out_lines) + "\n"

# ========================
# DUMMY VISUALIZATION (placeholder)
# ========================

def visualize_nucleotide_sequence(seq, title="AGC-128 Sequence",
checksum_length=0, error_index=-1):
    """
    Плейсхолдър – няма графика, само показва информация.
    """
    info_message = f"Title: {title}\n"
    info_message += f"Sequence Length: {len(seq)} nucleotides\n"
    if checksum_length > 0:
        info_message += f"Checksum Length: {checksum_length} nucleotides\n"
        info_message += f"Checksum Nucleotides: {'
'.join(seq[-checksum_length:])}\n"
    if error_index != -1:
        info_message += f"Highlighted Error at index: {error_index} (nucleotide:
{seq[error_index]})\n"
    info_message += (
        "\n(Visualization functionality is a placeholder in this environment. "
        "Run locally for full matplotlib visualization.)"
    )

    messagebox.showinfo(
        "Visualize Sequence (Placeholder)",
        info_message
```

```
    )

# ========================
# GUI
# ========================

def setup_gui():
    global current_encoded_nucleotide_sequence

    root = tk.Tk()
    root.title("AGC-128 Notepad")
    root.geometry("1050x600") # Set initial window size

    # Frame for encoding version selection
    version_frame = tk.Frame(root)
    version_frame.pack(pady=5, anchor='w')

    tk.Label(version_frame, text="Encoding/Decoding Version:").pack(side=tk.LEFT)
    version_var = tk.StringVar(value="v1_ascii")  # Default to v1 (ASCII)

    v1_radio = tk.Radiobutton(version_frame, text="v1 (ASCII)",
variable=version_var, value="v1_ascii")
    v1_radio.pack(side=tk.LEFT, padx=5)

    v2_radio = tk.Radiobutton(version_frame, text="v2 (Unicode)",
variable=version_var, value="v2_unicode")
    v2_radio.pack(side=tk.LEFT, padx=5)

    # Configure text_widget with undo/redo history and scrollbar
    text_frame = tk.Frame(root) # New frame for text widget and scrollbar
    text_frame.pack(expand=True, fill='both')

    text_widget = tk.Text(text_frame, wrap='word', undo=True,
autoseparators=True)
    text_widget.pack(side=tk.LEFT, expand=True, fill='both')

    # Add a scrollbar
    scrollbar = tk.Scrollbar(text_frame, command=text_widget.yview)
    scrollbar.pack(side=tk.RIGHT, fill='y')
    text_widget.config(yscrollcommand=scrollbar.set)

    menubar = tk.Menu(root)
    root.config(menu=menubar)

    # ---------- FILE ----------
    file_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="File", menu=file_menu)

    def new_file():
        text_widget.delete("1.0", tk.END)
        current_encoded_nucleotide_sequence.clear()
        messagebox.showinfo("New File", "New file created. Editor cleared.")

    def open_file():
```

```
    global current_encoded_nucleotide_sequence
    file_path = filedialog.askopenfilename(
        filetypes=[("Text files", "*.txt"), ("All files", "*.* затем")]
    )
    if file_path:
        with open(file_path, 'r', encoding='utf-8') as file:
            content = file.read()
        text_widget.delete("1.0", tk.END)
        text_widget.insert(tk.END, content)
        current_encoded_nucleotide_sequence.clear()

def save_file():
    file_path = filedialog.asksaveasfilename(
        defaultextension=".txt",
        filetypes=[("Text files", "*.txt"), ("All files", "*.* затем")]
    )
    if file_path:
        content = text_widget.get("1.0", tk.END)
        with open(file_path, 'w', encoding='utf-8') as file:
            file.write(content)

def save_file_as():
    file_path = filedialog.asksaveasfilename(
        defaultextension=".txt",
        filetypes=[("Text files", "*.txt"), ("All files", "*.* затем")]
    )
    if file_path:
        content = text_widget.get("1.0", tk.END)
        with open(file_path, 'w', encoding='utf-8') as file:
            file.write(content)

file_menu.add_command(label="New", command=new_file)
file_menu.add_command(label="Open", command=open_file)
file_menu.add_command(label="Save", command=save_file)
file_menu.add_command(label="Save As...", command=save_file_as)
file_menu.add_separator()
file_menu.add_command(label="Exit", command=root.quit)

# ---------- EDIT MENU (NEW) ----------
edit_menu = tk.Menu(menubar, tearoff=0)
menubar.add_cascade(label="Edit", menu=edit_menu)

def undo_action():
    try:
        text_widget.edit_undo()
    except tk.TclError:
        pass # Cannot undo

def redo_action():
    try:
        text_widget.edit_redo()
    except tk.TclError:
        pass # Cannot redo
```

```
    def cut_action():
        text_widget.event_generate('<<Cut>>')

    def copy_action():
        text_widget.event_generate('<<Copy>>')

    def paste_action():
        text_widget.event_generate('<<Paste>>')

    def delete_action():
        try:
            text_widget.delete(tk.SEL_FIRST, tk.SEL_LAST)
        except tk.TclError: # No text selected
            pass

    def select_all_action():
        text_widget.tag_add(tk.SEL, '1.0', tk.END)
        text_widget.mark_set(tk.INSERT, '1.0')
        text_widget.see(tk.INSERT) # Scroll to the beginning

    edit_menu.add_command(label="Undo", command=undo_action)
    edit_menu.add_command(label="Redo", command=redo_action)
    edit_menu.add_separator()
    edit_menu.add_command(label="Cut", command=cut_action)
    edit_menu.add_command(label="Copy", command=copy_action)
    edit_menu.add_command(label="Paste", command=paste_action)
    edit_menu.add_command(label="Delete", command=delete_action)
    edit_menu.add_separator()
    edit_menu.add_command(label="Select All", command=select_all_action)

    # ---------- CONTEXT MENU (NEW) ----------
    def show_context_menu(event):
        context_menu = tk.Menu(text_widget, tearoff=0)
        context_menu.add_command(label="Cut", command=cut_action)
        context_menu.add_command(label="Copy", command=copy_action)
        context_menu.add_command(label="Paste", command=paste_action)
        context_menu.add_separator()
        context_menu.add_command(label="Select All", command=select_all_action)
        context_menu.add_command(label="Clear", command=lambda:
text_widget.delete('1.0', tk.END))
        try:
            context_menu.tk_popup(event.x_root, event.y_root)
        finally:
            context_menu.grab_release()

    text_widget.bind("<Button-3>", show_context_menu)

    # ---------- ENCODE ----------
    encode_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Encode", menu=encode_menu)

    def encode_to_fasta_action():
        global current_encoded_nucleotide_sequence
```

```
        input_text = text_widget.get("1.0", tk.END).strip()
        if not input_text:
            messagebox.showwarning("No Input", "Please enter text to encode in
the editor.")
            return

        fasta_id = simpledialog.askstring("FASTA Identifier", "Enter FASTA header
ID:")
        if not fasta_id:
            messagebox.showwarning("Missing ID", "FASTA identifier cannot be
empty.")
            return

        add_checksum = messagebox.askyesno("Checksum Option", "Do you want to add
a genetic checksum?")

        try:
            selected_version = version_var.get()
            if selected_version == "v1_ascii":
                nucleotide_sequence_temp =
string_to_nucleotide_sequence_v1(input_text)
            else:  # v2_unicode
                nucleotide_sequence_temp =
encode_string_to_unicode_tagc_sequence(input_text)

            if add_checksum:
                processed_sequence =
add_genetic_checksum(nucleotide_sequence_temp)
            else:
                processed_sequence = nucleotide_sequence_temp

            current_encoded_nucleotide_sequence[:] = processed_sequence

            fasta_output = generate_fasta_string(
                processed_sequence,
                fasta_id,
                line_width=60
            )

            save_path = filedialog.asksaveasfilename(
                defaultextension=".fasta",
                filetypes=[("FASTA files", "*.fasta"), ("All files", "*.*
затем")],
                title="Save Encoded FASTA As"
            )
            if save_path:
                with open(save_path, 'w', encoding='utf-8') as f:
                    f.write(fasta_output)
                messagebox.showinfo("Success", f"FASTA encoded and saved to
{save_path}")
            else:
                messagebox.showinfo("Cancelled", "FASTA save operation
cancelled.")
        except Exception as e:
```

```
            messagebox.showerror("Encoding Error", f"An error occurred during
encoding: {e}")

    encode_menu.add_command(label="Encode to AGC-128 FASTA",
command=encode_to_fasta_action)

    # ---------- DECODE ----------
    decode_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Decode", menu=decode_menu)

    def load_and_decode_fasta_action():
        global current_encoded_nucleotide_sequence

        file_path = filedialog.askopenfilename(
            filetypes=[("FASTA files", "*.fasta"), ("All files", "*.* затем")]
        )
        if not file_path:
            messagebox.showinfo("Cancelled", "FASTA load operation cancelled.")
            return

        try:
            with open(file_path, 'r', encoding='utf-8') as file:
                content = file.read()

            lines = content.splitlines()
            if not lines or not lines[0].startswith('>'):
                messagebox.showwarning(
                    "Invalid FASTA",
                    "Selected file does not appear to be a valid FASTA format
(missing header)."
                )
                return

            # Extract sequence, ignore header(s), keep only A/T/G/C
            seq_raw = "".join(line.strip() for line in lines[1:] if not
line.startswith(">"))
            valid = {'A', 'T', 'G', 'C'}
            extracted_nucs_list = [c for c in seq_raw if c in valid]

            if not extracted_nucs_list:
                messagebox.showwarning("Empty Sequence", "No nucleotide sequence
found in the FASTA file.")
                return

            current_encoded_nucleotide_sequence[:] = extracted_nucs_list

            sequence_to_decode = list(extracted_nucs_list) # Use a copy to allow
modification
            checksum_info = ""

            # --- MODIFIED CHECKSUM HANDLING ---
            ask_if_checksum_present = messagebox.askyesno(
                "Checksum Query",
                "Is a 2-nucleotide genetic checksum expected at the end of this
```

```
sequence?"
            )

            if ask_if_checksum_present:
                if len(extracted_nucs_list) < 2:
                    messagebox.showwarning("Checksum Error", "Sequence is too
short to contain a 2-nucleotide checksum.")
                else:
                    is_valid_checksum =
verify_genetic_checksum(extracted_nucs_list)
                    checksum_info = f"\nChecksum valid: {is_valid_checksum}"
                    if is_valid_checksum:
                        messagebox.showinfo("Checksum Status", f"Checksum is
valid!{checksum_info}")
                        sequence_to_decode = extracted_nucs_list[:-2] # Remove
checksum for decoding
                    else:
                        messagebox.showwarning(
                            "Checksum Status",
                            f"Checksum is INVALID! Data may be
corrupted.{checksum_info}\n"
                            "The checksum will NOT be removed before decoding as
it's invalid."
                        )
                        # If checksum is invalid, we don't automatically remove
it.
                        # The user might want to inspect the corrupted checksum
itself.
                        # The sequence_to_decode remains the full
extracted_nucs_list.
            # --- END MODIFIED CHECKSUM HANDLING ---

            # Determine the selected version for decoding
            selected_version = version_var.get()

            # Perform pre-decoding length check if no checksum was removed and
it's V1.
            # V2 has variable length chunks, so len % 4 is not a strong indicator
for end truncation.
            if not ask_if_checksum_present and selected_version == "v1_ascii" and
len(sequence_to_decode) % 4 != 0:
                 messagebox.showwarning(
                    "Sequence Length Mismatch (V1)",
                    "The V1 ASCII nucleotide sequence length is not a multiple of
4.\n"
                    "Decoding might result in an incomplete last character."
                )

            if selected_version == "v1_ascii":
                decoded_text =
decode_nucleotide_sequence_to_string_v1(sequence_to_decode)
            else: # v2_unicode
                decoded_text =
decode_unicode_tagc_sequence_to_string(sequence_to_decode)
```

```
            text_widget.delete("1.0", tk.END)
            text_widget.insert(tk.END, decoded_text)
            messagebox.showinfo("Decoding Success", f"FASTA file successfully
loaded and decoded!{checksum_info}")

        except ValueError as ve: # Catch specific ValueError from decoding
functions
            messagebox.showerror("Decoding Error (Data Integrity)", f"A data
integrity error occurred during decoding: {ve}\nThis might indicate a corrupted
sequence or incorrect encoding version/checksum assumption.")
        except Exception as e:
            messagebox.showerror("Decoding Error", f"An unexpected error occurred
during FASTA loading or decoding: {e}")

    decode_menu.add_command(label="Load and Decode AGC-128 FASTA",
command=load_and_decode_fasta_action)

    # ---------- TOOLS ----------
    tools_menu = tk.Menu(menubar, tearoff=0)
    menubar.add_cascade(label="Tools", menu=tools_menu)

    def verify_checksum_action():
        global current_encoded_nucleotide_sequence
        if not current_encoded_nucleotide_sequence:
            messagebox.showwarning("No Sequence", "No encoded nucleotide sequence
is currently loaded or generated.")
            return

        # --- MODIFIED CHECKSUM HANDLING IN VERIFY ACTION ---
        ask_if_checksum_present = messagebox.askyesno(
            "Checksum Query",
            "Is a 2-nucleotide genetic checksum expected at the end of the
current sequence?"
        )

        if ask_if_checksum_present:
            if len(current_encoded_nucleotide_sequence) < 2:
                messagebox.showwarning("Checksum Error", "The current sequence is
too short to contain a 2-nucleotide checksum.")
                return

            is_valid =
verify_genetic_checksum(current_encoded_nucleotide_sequence)
            messagebox.showinfo("Checksum Verification", f"Checksum valid:
{is_valid}")
        else:
            messagebox.showinfo("Checksum Information", "No checksum verification
performed as none was expected.")
        # --- END MODIFIED CHECKSUM HANDLING ---

    def visualize_action():
        global current_encoded_nucleotide_sequence
        if not current_encoded_nucleotide_sequence:
```

```
            messagebox.showwarning(
                "No Sequence",
                "No encoded nucleotide sequence is currently loaded or generated
to visualize."
            )
            return

        checksum_len = 0
        sequence_for_viz = list(current_encoded_nucleotide_sequence) # Make a
copy

        # --- MODIFIED CHECKSUM HANDLING IN VISUALIZE ACTION ---
        ask_if_checksum_present = messagebox.askyesno(
            "Checksum Query",
            "Is a 2-nucleotide genetic checksum expected at the end of the
current sequence for visualization?"
        )

        if ask_if_checksum_present:
            if len(current_encoded_nucleotide_sequence) < 2:
                messagebox.showwarning("Checksum Error", "Sequence is too short
to contain a 2-nucleotide checksum for visualization.")
            else:
                is_valid_checksum =
verify_genetic_checksum(current_encoded_nucleotide_sequence)
                if is_valid_checksum:
                    checksum_len = 2 # Indicate to visualization to highlight
last 2 nucs
                    messagebox.showinfo("Checksum Status", "Checksum is valid and
will be highlighted.")
                else:
                    messagebox.showwarning("Checksum Status", "Checksum is
INVALID. Will still highlight, but data may be corrupted.")
                    checksum_len = 2 # Still highlight, even if invalid
        # --- END MODIFIED CHECKSUM HANDLING ---

        try:
            visualize_nucleotide_sequence(
                sequence_for_viz, # Pass the original sequence, checksum_len will
handle highlighting
                "Current AGC-128 Sequence",
                checksum_length=checksum_len
            )
        except Exception as e:
            messagebox.showerror("Visualization Error", f"An error occurred
during visualization: {e}")

    tools_menu.add_command(label="Verify Checksum",
command=verify_checksum_action)
    tools_menu.add_command(label="Visualize Sequence", command=visualize_action)

    root.mainloop()

# ========================
```

```
# MAIN
# ========================

if __name__ == "__main__":
    # Check if running in Google Colab (or similar non-GUI environment)
    if 'google.colab' in sys.modules:
        print("Running in Google Colab environment. Tkinter GUI cannot be
displayed.\n")
        print("Here's an example of how to use the core encoding/decoding
functions directly:\n")

        sample_text = "Здравейте, свят!😊 123"
        print(f"Original Text (V2 Unicode): {sample_text}")

        # V2 Unicode Encoding Example
        try:
            encoded_v2 = encode_string_to_unicode_tagc_sequence(sample_text)
            print(f"Encoded (V2 Unicode): {''.join(encoded_v2[:60])}{'...' if
len(encoded_v2) > 60 else ''} (Total: {len(encoded_v2)} nucleotides)")

            # Add and verify checksum
            encoded_v2_with_checksum = add_genetic_checksum(encoded_v2)
            print(f"Encoded with Checksum (V2 Unicode):
{''.join(encoded_v2_with_checksum[:60])}{'...' if len(encoded_v2_with_checksum) >
60 else ''} (Total: {len(encoded_v2_with_checksum)} nucleotides)")
            print(f"Checksum for V2 is valid:
{verify_genetic_checksum(encoded_v2_with_checksum)}")

            decoded_v2 = decode_unicode_tagc_sequence_to_string(encoded_v2)
            print(f"Decoded (V2 Unicode): {decoded_v2}")
            print(f"V2 Encoding/Decoding successful: {sample_text ==
decoded_v2}")

        except Exception as e:
            print(f"Error during V2 Unicode example: {e}")

        # V1 ASCII Encoding Example (for comparison, only works for ASCII
characters)
        print("\n---\n")
        ascii_sample_text = "Hello, Colab!"
        print(f"Original ASCII Text (V1 ASCII): {ascii_sample_text}")
        try:
            encoded_v1 = string_to_nucleotide_sequence_v1(ascii_sample_text)
            print(f"Encoded (V1 ASCII): {''.join(encoded_v1)}")

            encoded_v1_with_checksum = add_genetic_checksum(encoded_v1)
            print(f"Encoded with Checksum (V1 ASCII):
{''.join(encoded_v1_with_checksum)}")
            print(f"Checksum for V1 is valid:
{verify_genetic_checksum(encoded_v1_with_checksum)}")

            decoded_v1 = decode_nucleotide_sequence_to_string_v1(encoded_v1)
            print(f"Decoded (V1 ASCII): {decoded_v1}")
            print(f"V1 Encoding/Decoding successful: {ascii_sample_text ==
```

```
decoded_v1}")
        except Exception as e:
            print(f"Error during V1 ASCII example: {e}")

    else:
        try:
            setup_gui()
        except tk.TclError as e:
            print(f"Error: {e}")
            print("Tkinter GUI cannot be displayed in this environment (e.g.,
Google Colab). Not a local environment.")
            print("Run this script locally on your computer with a graphical
interface.")

"""# AGC_128_Standart_notepad_v.3 — Adaptive Genetic Code 128 (Unified Edition)

## Official README (Unified Edition)

---

## Authors
- **Aleksandar Kitipov**
  Emails: aeksandar.kitipov@gmail.com / aeksandar.kitipov@outlook.com
- **Copilot**
  Co-author, technical collaborator, documentation support
  - **Gemini 2.5 Flash**
  Co-author, technical collaborator, documentation support

---

## 1. Overview
**AGC_128_Standart_notepad_v.3** is the unified and enhanced version of the
Adaptive Genetic Code 128 text encoding system. It combines the original **v.1
(ASCII)** capabilities with the **v.2 (Unicode)** extension, offering a robust,
fully reversible, DNA-inspired encoding solution for both basic ASCII text and
the entire Unicode character set.

This version features an interactive Graphical User Interface (GUI) built with
`tkinter`, allowing users to seamlessly switch between v1 and v2
encoding/decoding, open/save text files, encode to/decode from FASTA files, and
manage genetic checksums. Like its predecessors, it requires **no external
libraries** for its core encoding/decoding logic, maintaining a tiny footprint
while ensuring high data integrity through its self-checking genetic structure.

---

## 2. What the Program Does

AGC_128_Standart_notepad_v.3 provides complete reversible transformations:

### v.1 (ASCII) Transformation:
```
Text → ASCII (8 bits) → 4 (2-bit genes) → A/T/G/C DNA Sequence
```
```

And back:
```
DNA Sequence → 4 (2-bit genes) → 8-bit ASCII → Text
```

v.2 (Unicode) Transformation:
```
Unicode Text → UTF-8 Bytes → Length Gene + Genetic Bytes → A/T/G/C DNA Sequence
```
And back:
```
DNA Sequence → Genetic Bytes + Length Gene → UTF-8 Bytes → Unicode Text
```

This system precisely preserves:
- **v.1:** Letters, numbers, punctuation, whitespace, and ASCII extended symbols.
- **v.2:** Any Unicode character (including ASCII, Cyrillic, CJK, Emojis, Symbols), preserving structured blocks and FASTA-formatted sequences.

If you encode text and decode it again using the correct version, the output will match the original exactly, character-for-character, byte-for-byte.

---

3. Key Features

3.1. Unified Encoding/Decoding
Supports both `v1 (ASCII)` for lightweight, fixed-length encoding of standard ASCII characters and `v2 (Unicode)` for comprehensive variable-length encoding of all Unicode characters via UTF-8.

3.2. Intuitive Graphical User Interface (GUI)
Built with `tkinter` for ease of use, featuring:
- **Version Selection:** Radio buttons to choose between `v1 (ASCII)` and `v2 (Unicode)` encoding/decoding modes.
- **File Operations:** Open, Save, Save As, and New file functionalities.
- **Edit Menu:** Standard text editor actions like Undo, Redo, Cut, Copy, Paste, Delete, and Select All.
- **Context Menu:** Right-click menu for quick editing actions.
- **FASTA Management:** Encode current text to FASTA and Load/Decode FASTA files.
- **Checksum Integration:** Options to add, verify, and consider genetic checksums during encoding/decoding.
- **Visualization Placeholder:** Provides basic sequence information.

3.3. Full Reversibility
Every character, whether ASCII or Unicode, is transformed reversibly, ensuring zero data loss upon decoding.

3.4. Self-Checking Genetic Structure (Inherited from v1 & v2)
AGC-128 maintains its three core biological-style integrity rules:
- **Sum-2 Rule**: Each 2-bit gene has a total bit-sum of 2. Any bit flip breaks the rule and becomes detectable.
- **No-Triple Rule**: The sequence can never contain `111` or `000`. If such a pattern appears, the data is invalid.

- **Deterministic-Next-Bit Rule**: Predictable bit sequences (`11` → `0`, `00` → `1`). This allows partial reconstruction of missing or damaged data.

3.5. FASTA Compatibility
The DNA output can be saved as a `.fasta` file, making it suitable for digital archiving, DNA-like storage experiments, and bioinformatics-style workflows.

---

4. Genetic Alphabet
AGC-128 uses four genetic symbols mapped from 2-bit pairs:

```
11 → G
00 → C
10 → A
01 → T
```

---

5. AGC-128 v2 (Unicode) Core Principles in Detail

5.1. UTF-8 as Foundation
Unicode characters are first converted to their UTF-8 byte representation (1 to 4 bytes).

5.2. Length Prefix Gene
Each encoded Unicode character begins with a single-nucleotide `Length Gene` that indicates the number of UTF-8 bytes that follow for that character:

UTF-8 Length	Number of Bytes	2-bit Marker	Length Gene
1 byte	ASCII	00	C
2 bytes	Cyrillic	01	T
3 bytes	Multi-byte	10	A
4 bytes	Emojis	11	G

5.3. Byte Encoding
Each individual UTF-8 byte (0-255) is encoded into four 2-bit nucleotide genes, consistent with AGC-128 v1's 8-bit to 4-nucleotide conversion.

Thus, a Unicode character's genetic sequence is: `[Length Gene] + [4 genes per byte]`.
- **1-byte UTF-8 (ASCII)** → `C` + 4 genes = 5 nucleotides
- **2-bytes UTF-8 (e.g., Cyrillic)** → `T` + 8 genes = 9 nucleotides
- **3-bytes UTF-8 (e.g., Chinese)** → `A` + 12 genes = 13 nucleotides
- **4-bytes UTF-8 (e.g., Emojis)** → `G` + 16 genes = 17 nucleotides

---

6. Genetic Checksum
An optional 2-nucleotide genetic checksum can be appended to the entire sequence to verify data integrity. It calculates the sum of all 2-bit nucleotide values,

modulo 16, and encodes this 4-bit result into two nucleotides. The GUI provides explicit options to add and verify this checksum, ensuring flexibility and data validation.

---

7. Usage (GUI)

To use the GUI:
1. **Run the script locally** (as `tkinter` requires a graphical environment).
2. **Select Encoding/Decoding Version:** Use the radio buttons (`v1 (ASCII)` or `v2 (Unicode)`).
3. **Type or Load Text:** Enter text directly or use `File > Open`.
4. **Encode to FASTA:** Go to `Encode > Encode to AGC-128 FASTA`. You'll be prompted for a FASTA header and whether to add a checksum.
5. **Load and Decode FASTA:** Go to `Decode > Load and Decode AGC-128 FASTA`. The system will prompt if a checksum is expected and verify it if present.
6. **Tools:** `Verify Checksum` (for currently loaded sequence) and `Visualize Sequence` (placeholder info).

---

8. Command-Line Usage (Colab Environment)

When running in environments without a GUI (like Google Colab), the script automatically executes example encoding/decoding functions for both v1 and v2, demonstrating core functionality.

Example Output in Colab:
```
Running in Google Colab environment. Tkinter GUI cannot be displayed.

Here's an example of how to use the core encoding/decoding functions directly:

Original Text (V2 Unicode): Здравейте, свят!😊 123
Encoded (V2 Unicode): TGTCCA TTGTCCAGTC TGTC TACCCTGTCCAGCCC
TTGTCCAGCATGTCCAGTTTGTC ... (Total: 73 nucleotides)
Encoded with Checksum (V2 Unicode): TGTCCA TTGTCCAGTC TGTC TACCCTGTCCAGCCC
TTGTCCAGCATGTCCAGTTTGTC ... (Total: 75 nucleotides)
Checksum for V2 is valid: True
Decoded (V2 Unicode): Здравейте, свят!😊 123
V2 Encoding/Decoding successful: True

---

Original ASCII Text (V1 ASCII): Hello, Colab!
Encoded (V1 ASCII): TCACTATTTAGCTAGCAGGCA CCTCC TGGTAGCTACTACACACT
Encoded with Checksum (V1 ASCII): TCACTATTTAGCTAGCAGGCA CCTCC
TGGTAGCTACTACACACTCG
Checksum for V1 is valid: True
Decoded (V1 ASCII): Hello, Colab!
V1 Encoding/Decoding successful: True
```

---

9. Project Status
- **AGC_128_Standart_notepad_v.3** — Stable unified core with GUI.

---

10. Notes
This README represents the comprehensive documentation for the unified AGC-128 Notepad, incorporating all features and improvements developed through collaborative discussions and testing.
"""