Zestaw 1

1. Napisać program, który czyta podane jako zewnętrzne argumenty liczby naturalne, a następnie każdą rozkłada na czynniki pierwsze (co polega na zapisaniu dowolnej liczby naturalnej za pomocą iloczynu liczb pierwszych). Wymagany jest format wyjściowy w postaci a_1^k_1*a_2^k_2*…*a_3, jeśli k_i==1 to opuszczamy wykładnik potęgi. Przykładowo, jeśli wywołamy:

```
zadanie1.py  4407  13041599400
```

to powinno się wypisać (proszę tak to sformatować, sprawdzany będzie rozkład czyli prawa strona):

```
4407 = 3*13*113
13041599400 = 2^3*3^4*5^2*805037
```

Do wczytania zewnętrznych argumentów proponuję na początek coś bardzo podobnego do tego, co jest w języku C++, czyli użycie listy argumentów (bez używania `getopt` czy `argparse`):

```
import sys # importujemy modul

argv = sys.argv[1:] # argv to lista, a 1: robi selekcje bez pierwszego argumentu – nazwy programu

for i in range(1,len(argv)): # za pomoca generatora
          print(int(sys.argv[i])) # wpisujemy, rzutowanie z typu str na int przyda sie potem
```

Opis i program w C++ https://www.algorytm.edu.pl/algorytmy-maturalne/rozklad-na-czynniki.html
Pomocny może też być kalkulator rozkładu na czynniki pierwsze https://www.liczebnik.pl/czynniki-pierwsze.php

Wymagania formalne Użyć plik `ZADANIE1/zadanie1.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Nie zmieniać nazw funkcji. Dane wejściowe w postaci liczb naturalnych, oddzielonych spacją (spacjami). Testowane będzie działanie funkcji `rozkład_na_czynniki(n)`, która powinna zwracać uformowany łańcuch znakowy w postaci jak poniżej „wyjście”.

wejście: `805037` **wyjście:** `805037`
wejście: `13041599400` **wyjście:** `2^3*3^4*5^2*805037`

2. Napisać program, który generuje na bieżąco ciąg znaków z alfabetu A–Z, a–z. Każdy znak powtarza się losową liczbę razy 1–10, po czym pojawia się kolejny wylosowany i proces toczy się dalej, aż do wygenerowania 1000 znaków. Program ma w jednej linii drukować „przepływ” ostatnich 80 wylosowanych znaków: na początku linia wypełnia się od pustego do WIDTH = 80, a następnie przy każdym nowym znaku wszystko przesuwa się w lewo. Po prawej stronie tej animowanej linii ma być wyświetlana, aktualizowana na bieżąco, wartość kompresji RLE w formacie xx%.

Kompresja RLE (Run-Length Encoding) zamienia kolejne identyczne znaki jako „znak + liczba powtórzeń”. Długość zapisu pojedynczego znaku to 1, a jeśli znak występuje N≥2 razy z rzędu, długość zapisu to 1 + liczba_cyfr(N) (np. a3 → 2, a10 → 3). Zatem jeden znak `a` zamieniamy na `a`, ciąg `aa` na `a2`, ciąg `aaaaa` na `a5` itd.

Przykład pojedynczej, 80-znakowej linii i wyliczonego współczynnika (tu 55%):

```
aaabbcccdddddeeeeeffffffgggghhhhiiiijjjjkkkllllmmmnnnooopppqqqrrrrsssstttttuuuuvvvv  55%
```

Aby uzyskać efekt przesuwania w jednej linii, należy użyć funkcję `print` z parametrami `end="\r"` i `flush=True`. Znak `\r` (powrót karetki) przenosi kursor na początek bieżącej linii, dzięki czemu nowy wydruk nadpisuje poprzedni. Dla czytelności można dopełnić linię spacjami (np. `ljust(WIDTH)`) i dodać na końcu stały sufiks z `xx%`. Po zakończeniu działania programu należy wywołać `print()` bez argumentów, aby przejść do nowej linii. Poniżej, kilka pomocniczych informacji i objaśnień.

Moduł string

W Pythonie moduł string zawiera gotowe zestawy znaków, dzięki czemu nie trzeba pisać alfabetu ręcznie:

```
string.ascii_letters   # 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
string.ascii_lowercase # tylko małe litery
string.ascii_uppercase # tylko wielkie litery
string.digits          # '0123456789'
string.punctuation     # znaki interpunkcyjne
```

W zadaniu warto użyć `string.ascii_letters`. Na przykład: `ALPHABET = string.ascii_letters`

Losowanie

Z modułu `random` korzystamy, aby wybrać losową literę (z zakresu A-Za-z), oraz żeby wylosować liczbę całkowitą z zakresu 1-10 jako liczbę powtórzeń danego znaku, obydwa losowania w funkcji znaki(). Przykładowy kod:

```
import random   # na początku pliku

ch = random.choice(ALPHABET)     # losowy znak z alfabetu
repeat = random.randint(1, 10)   # losowa liczba powtórzeń 1–10
```

Jeśli chcemy, by program losował za każdym razem to samo (np. do testów), można ustawić ziarno:

```
random.seed(123)
```

Generator i yield

Funkcja `znaki()` ma działać jako generator, czyli funkcja, która zwraca kolejne wartości w miarę potrzeby, a nie wszystkie naraz. W generatorze używa się słowa `yield` zamiast return. Po każdym `yield` funkcja „zapamiętuje" swój stan i przy następnym wywołaniu wznawia działanie od tego miejsca. Przykład prostego generatora:

```
def znaki_demo():
    while True:
        yield 'A'
        yield 'B'
```

Każde wywołanie `next()` lub kolejna iteracja for zwraca następny znak: `'A'`, potem `'B'`, potem znowu `'A'` itd. Przykład kodu:

```
it = znaki_demo()
print(next(it))      # pierwszy znak 'A'
print(next(it))      # drugi znak 'B'
print(next(it))      # trzeci znak 'A'

# albo pętla for – pobieramy np. 10 kolejnych znaków:
for i, ch in zip(range(10), znaki_demo()):
          print(ch, end="")
print()  # nowa linia po zakończeniu
```

Warto skomentować powyższy przykład - funkcja `zip()` łączy kilka sekwencji w pary elementów - w tym przypadku `range(10)` (czyli liczby `0-9`) i `znaki()` (generator znaków). Dzięki temu pętla wykona się dokładnie 10 razy, nawet jeśli generator `znaki_demo()` działa nieskończenie długo.

W naszym zadaniu znaki() ma losować znak z alfabetu i powtarzać go określoną liczbę razy, np. jeśli wylosowana wartość liczby powtórzonego znaku to zmienna repeat, to koniec funkcji może wyglądać następująco:

```
for _ in range(repeat):   # znak _ używa się do nazwania zmiennej, której wartość nas nie interesuje
  yield ch
```

deque (dwukierunkowa kolejka) z collections

`deque` to struktura umożliwiająca szybkie dopisywanie i usuwanie elementów z obu końców. W naszym zadaniu idealnie nadaje się do utrzymania okna ostatnich `WIDTH` znaków. Ustawienie parametru `maxlen=WIDTH` sprawia, że po przekroczeniu rozmiaru najstarszy element „wypada" automatycznie z lewej, co dokładnie daje efekt „przesuwania się" linii. Przykład:

```
from collections import deque

buf = deque(maxlen=5)   # okno o stałej szerokości
for ch in "ABCDEFG":
    buf.append(ch)      # dokładamy nowy znak
    print("".join(buf)) # pokaz okna
```

Wydruk:

```
A
AB
ABC
ABCD
ABCDE
BCDEF   # po dołożeniu 'F' wypadło 'A'
CDEFG   # po dołożeniu 'G' wypadło 'B'
```

W efekcie `deque(maxlen=WIDTH)` daje nam bezobsługowe, wydajne „przesuwne okno" do animowanego podglądu ostatnich znaków.

Wymagania formalne Użyć plik `ZADANIE2/zadanie2.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. W zadanie2.py mają pozostać i zostać zaimplementowane funkcje `znaki()` oraz `dlugosc(count)`, a także uzupełniona logika w `main()` zgodnie z opisem. Nie zmieniać nazw funkcji i interfejsu; wolno modyfikować jedynie wartości `WIDTH`, `MAXLEN`, `DELAY_SEC`, `ALPHABET`. Program posiada testy, dość wyrainowane, bo również symulujące podmianę losowanych wartości (poprzez „fikstur ę" `monkeypatch` z pakietu `pytest`). Testy służą wyłącznie sprawdzeniu poprawności funkcji `znaki()` i `dlugosc()`, choć można podejrzeć ich działanie, żeby zobaczyć wyliczanie współczynnika kompresji dla całego ciągu znaków.

3. Napisz program, który wyświetla w terminalu duży zegar w formacie HH : MM : SS (24h) zbudowany z matryc znaków `█`. Zegar ma odświeżać się dynamicznie. Ekran powinien być czytelnie przerysowywany przy każdej zmianie. Przykładowy wygląd fragmentu terminala:

Aby odczytać bieżący czas, skorzystaj z `datetime.now()` i jego pól: `now.hour`, `now.minute`, `now.second`. Przykład: `now = datetime.now()` daje aktualny moment, a potem `now.hour` to liczba z zakresu `0-23`. Do zdefiniowania „matryc" poszczególnych cyfr użyto listy oraz typu słownikowego (dict), który jest parą klucz → wartość. W tym zadaniu kluczami są znaki cyfr `'0'..'9'`, a wartościami listy 8-elementowe. Każdy element listy to wiersz o szerokości 8 znaków, gdzie `█` oznacza piksel włączony, a spacja - tło. W podobny sposób zdefiniowany jest dwukropek COLON jako 8 wierszy po 4 znaki. Czyszczenie ekranu w terminalu zależy od systemu. Na Windows użyj polecenia `cls`, a na Linux/macOS użyj polecenia `clear`. W Pythonie wywołasz je przez `os.system('cls')` albo `os.system('clear')` po sprawdzeniu `os.name`.

Wymagania formalne Użyć plik `ZADANIE3/zadanie3.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Program będzie uruchomiony i oceniony wizualnie, program nie posiada testów.

4. Napisać program, który pobiera bieżące dane pogodowe z internetu dla piętnastu największych miast Polski i wypisuje w terminalu aktualne temperatury w stopniach Celsjusza. Miasta, które należy uwzględnić: Warszawa, Kraków, Łódź, Wrocław, Poznań, Gdańsk, Szczecin, Bydgoszcz, Lublin, Białystok, Katowice, Gdynia, Częstochowa, Radom, Toruń. Program powinien wyświetlić zestawienie „Miasto : temperatura °C” w oddzielnych liniach, a na samym dole dodać podsumowanie z informacją, w którym mieście jest obecnie najniższa temperatura (wartość i nazwa miasta) oraz w którym najwyższa (wartość i nazwa miasta).

   W tym zadaniu będziemy pobierać dane z internetu za pomocą biblioteki `requests` i funkcji `get(url)`. Biblioteka `requests` to proste narzędzie Pythona służące do wykonywania zapytań HTTP; instalujemy ją poleceniem `pip install requests`, a w kodzie importujemy przez `import requests`. Do pobrania danych użyjemy serwis `wttr.in`, który udostępnia darmowe dane pogodowe w formacie JSON. Adres zapytania ma postać: https://wttr.in/NazwaMiasta?format=j1. Otwórzmy stronę dla Kraków, aby zobaczyć jakie dane są pobierane (poniżej tylko początek):


```
{
    "current_condition": [
        {
            "FeelsLikeC": "0",
            "FeelsLikeF": "33",
            "cloudcover": "0",
            "humidity": "87",
            "lang_pl": [
                {
                    "value": "Bezchmurnie"
                }
            ],
            "localObsDateTime": "2025-10-20 12:13 AM",
            "observation_time": "10:13 PM",
            "precipInches": "0.0",
            "precipMM": "0.0",
            "pressure": "1020",
            "pressureInches": "30",
            "temp_C": "2",
            "temp_F": "36",
            "uvIndex": "0",
            "visibility": "10",
            "visibilityMiles": "6",
            "weatherCode": "113",
```


   Funkcja `requests.get(url)` wykonuje zapytanie do wskazanego adresu i zwraca odpowiedź serwera. Najwygodniej od razu przekształcić ją na struktury Pythona poleceniem `data = requests.get(url).json()`. Zmienna data jest słownikiem (dict) zawierającym m.in. klucz "current_condition" z listą bieżących parametrów; pierwszy element tej listy to kolejny słownik opisujący aktualne warunki. Temperatura w stopniach Celsjusza jest pod kluczem "`temp_C`" jako łańcuch znaków. Przykładowy fragment kodu ilustrujący pobranie temperatury dla jednego miasta wygląda tak:

```
import requests
url = "https://wttr.in/Kraków?format=j1"
data = requests.get(url).json()
cur = data["current_condition"][0]
temp_c = cur["temp_C"]          # np. "2"
print("Kraków :", temp_c, "°C")
```

   W analogiczny sposób należy zbudować pętlę po zadanych miastach, zebrać ich temperatury, wypisać je w kolejnych wierszach, a następnie wyznaczyć i wypisać minimum oraz maksimum wraz z nazwami miast. W zadaniu należy zrobić to poprzez funkcję `skaner_temperatur()`, która pobiera dane dla wszystkich wskazanych miast i zwraca listę krotek (miasto, temperatura_int) w tej samej kolejności, co w zadanej liście miast; na podstawie zwróconych danych program ma wykonać wypisanie tabeli oraz podsumowania z najniższą i najwyższą temperaturą.

Wymagania formalne Użyć plik `ZADANIE4/zadanie4.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Nie zmieniać nazwy funkcji `skaner_temperatur()`. Zadaniu towrzyszy zestaw testów, które sprawdzą obecność funkcji o tej nazwie, poprawność zwracanego typu i długości listy wyników, zgodność formatu elementów listy, a także poprawność działania na danych podstawionych testowo bez rzeczywistych wywołań sieciowych.

5. Napisać prosty czat z lokalnym modelem LLM, który uruchamia dialog w terminalu: użytkownik wpisuje pytanie, program wysyła je do lokalnie działającego modelu przez interfejs HTTP i wypisuje odpowiedź. Pusty wiersz kończy rozmowę. Program ma korzystać z dwóch funkcji: `pobierz_odpowiedz(prompt)`, która zwraca tekst wygenerowany przez model dla zadanego promptu, oraz `uruchom_czat()`, która obsługuje pętlę wejścia/wyjścia w terminalu. Wersja startowa zawiera już szkielety obu funkcji — należy uzupełnić ich treść.

   Do uruchomienia lokalnego modelu użyjemy aplikację Ollama. Po instalacji należy pobrać mały model i sprawdzić, czy serwer działa. Instalacja odbywa się ze strony ollama.com - po zainstalowaniu (https://ollama.com/download) uruchom Ollama; na systemach desktopowych serwis startuje w tle i nasłuchuje pod adresem http://localhost:11434. Pobranie modelu wykonuje się w terminalu poleceniem:

   ```
   ollama pull llama3.2:1b
   ```

   Sprawdzenie działania:

   ```
   ollama run llama3.2:1b
   ```

   Wpisz krótkie pytanie i upewnij się, że otrzymujesz odpowiedź. Zakończ, np. Ctrl+C.

   Zapytanie do modelu (`payload`) to obiekt JSON zawierający co najmniej nazwę modelu i treść promptu. W tym zadaniu użyjemy pola „model” (np. "`llama3.2:1b`"), pola „`prompt`” (łańcuch z pytaniem użytkownika) oraz wyłączamy strumieniowanie przez „`stream`”: False. Przykładowy kształt danych wysyłanych do serwisu:

   ```
   {"model": "llama3.2:1b", "prompt": "Twoje pytanie tutaj", "stream": False}
   ```

   Wysyłka odbywa się metodą POST na adres http://localhost:11434/api/generate przy użyciu biblioteki `requests`. Odpowiedź serwera ma postać JSON, w którym interesujący nas tekst modelu znajduje się w polu „`response`”. Minimalna ilustracja pojedynczego wywołania (do wglądu, nie jest to gotowa implementacja wymaganych funkcji):

   ```
   import requests
   URL = "http://localhost:11434/api/generate"
   payload = {"model": "llama3.2:1b", "prompt": "Napisz jedno zdanie.", "stream": False}
   r = requests.post(URL, json=payload)
   txt = r.json()["response"]
   print(txt)
   ```

   W analogiczny sposób należy zbudować funkcję `pobierz_odpowiedz(prompt)`, która odsyła wyłącznie tekst odpowiedzi, oraz funkcję `uruchom_czat()`, która w pętli pobiera wpisy użytkownika i wypisuje odpowiedzi modelu aż do pustego wiersza.

   Wymagania formalne Użyć plik `ZADANIE5/zadanie5.py` w repozytorium GitHub Classroom do uzupełnienia swoim kodem. Należy zdefiniować funkcję `pobierz_odpowiedz(prompt)`, która zwraca łańcuch znaków z odpowiedzią modelu oraz funkcję `uruchom_czat()`, która uruchamia prostą pętlę dialogu w terminalu i kończy działanie po wprowadzeniu pustej linii. Nie zmieniać nazw tych funkcji. Załączone testy sprawdzą obecność obu funkcji, oraz że `pobierz_odpowiedz(prompt)` zwraca wynik typu str wyodrębniony z pola „response” odpowiedzi JSON, a `uruchom_czat()` poprawnie obsługuje pojedynczą iterację rozmowy i kończy się po pustym wierszu; w testach wywołania sieciowe są podstawione lokalnie (`mock`), rzeczywisty serwer Ollama nie jest wymagany.