Algorytm Euklidesa (Euclidean Algorithm) rozwiązanie Behawioralne

Autorzy projektu:

- Mateusz Furgała
- Daniel Łukasik

Wprowadzenie

Algorytm Euklidesa to jeden z najstarszych znanych algorytmów numerycznych, który służy do wyznaczania największego wspólnego dzielnika (NWD) dwóch liczb całkowitych. Został opisany przez Euklidesa w jego dziele *Elementy* około 300 roku p.n.e.

NWD dwóch liczb to największa liczba, przez którą obie liczby dzielą się bez reszty.

Zasada działania algorytmu

Algorytm opiera się na następującej obserwacji:

> NWD(a, b) = NWD(b, a mod b)

Czyli:

- Jeżeli `b` wynosi 0, to `a` jest największym wspólnym dzielnikiem.
- W przeciwnym razie zamieniamy `a` na `b`, a `b` na `a % b` i powtarzamy procedurę.

Algorytm krok po kroku

Dla danych dwóch liczb całkowitych `a` i `b`, algorytm wykonuje:

1. Sprawdzenie, czy `b == 0`:
   - Jeśli tak → zwróć `a` jako NWD.
2. Jeśli nie:
   - Oblicz resztę z dzielenia `a % b`.
   - Podstaw: `a ← b`, `b ← a % b`.
3. Wróć do kroku 1.

Przykład działania

Znajdźmy NWD dla liczb `1071` i `462`.

Obliczenia:

- `1071 % 462 = 147`
- `462 % 147 = 21`
- `147 % 21 = 0` → koniec

Zatem `NWD(1071, 462) = 21`

Implementacja w Pythonie Behawioralnego podejścia

In [6]:
```
def euclidean_gcd(a, b):
    steps = []
    while b != 0:
        steps.append((a, b, a % b))
        a, b = b, a % b
    steps.append((a, 0, None))  # końcowy krok
    return a, steps

# Przykład
a, b = 1071, 462
gcd, steps = euclidean_gcd(a, b)

print(f"NWD({a}, {b}) = {gcd}\n")
print("Kolejne kroki:")
for i, (x, y, r) in enumerate(steps):
    if r is not None:
        print(f"Krok {i+1}: {x} % {y} = {r}")
    else:
        print(f"Krok {i+1}: koniec, ponieważ reszta = 0 → NWD = {x}")
```

```
NWD(1071, 462) = 21

Kolejne kroki:
Krok 1: 1071 % 462 = 147
Krok 2: 462 % 147 = 21
Krok 3: 147 % 21 = 0
Krok 4: koniec, ponieważ reszta = 0 → NWD = 21
```

Co robi algorytm – krok po kroku:

```
1. Sprawdza, czy b != 0.

2. Jeśli tak, to przypisuje a ← b, b ← a % b.

3. Powtarza, aż b == 0.

4. Zwraca a jako NWD.
```

In [24]:
```
def euclidean_gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

result = euclidean_gcd(7567, 7943)
print("NWD =", result)
```

```
NWD = 47
```

In [9]:
```
# Testing algoritm
import random
from math import gcd

MAX = (1 << 64) - 1             # największa 64-bitowa wartość

for _ in range(100):
    a = random.getrandbits(64)  # losuje pełne 64 b
    b = random.getrandbits(64)
    if a == 0: a = 1            # żeby NWD(a,0) nie dawało od razu a
    if b == 0: b = 1
    print(f"a = {a}, b = {b}, NWD = {gcd(a, b)}")
```

```
a = 16815315872773382955, b = 1185035766717641309, NWD = 1
a = 2340818040523030831, b = 9097513383063148311, NWD = 1
a = 18445694284263311634, b = 13577321950240740239, NWD = 1
a = 1374420875645165912, b = 9407824108161083711, NWD = 1
a = 3730863346093155047, b = 14628894053466083969, NWD = 1
a = 7872132048185032898, b = 8897279081378750334, NWD = 2
a = 904021653783107171, b = 6446207316056564378, NWD = 1
a = 9774712890897320977, b = 4981590057292834036, NWD = 1
a = 2814173053275747875, b = 960405131173472993, NWD = 1
a = 6825029698971820232, b = 16823962585536318631, NWD = 1
a = 4856734603258847555, b = 4810590111336068878, NWD = 1
a = 3336757232857317383, b = 12320263657255722528, NWD = 1
a = 3661012046844488149, b = 11285714085140540223, NWD = 1
a = 9629202533937697424, b = 6327148053241693099, NWD = 1
a = 11979085177049467770, b = 15872653135308940545, NWD = 15
a = 5778940331382031282, b = 14994380261270036998, NWD = 14
a = 4256714421600961134, b = 14800655891966471304, NWD = 6
a = 17858609671537450680, b = 562926058004836892, NWD = 4
a = 13652839393440844986, b = 4714380087993226318, NWD = 2
a = 12622346088782119434, b = 14957513365155483089, NWD = 1
a = 12321151891023246581, b = 6597550306991964331, NWD = 1
a = 18086960575982943551, b = 2657322745190437854, NWD = 1
a = 4472487959353734244, b = 1399727958025626325, NWD = 1
a = 17555242420329868288, b = 9867463488911399287, NWD = 1
a = 16150770360624317999, b = 11385931977499790931, NWD = 1
a = 17052717381768798378, b = 15208737175157505495, NWD = 3
a = 5708400728091959618, b = 4671539367937908268, NWD = 2
a = 9177050460843942727, b = 15631862209725229463, NWD = 1
a = 579695600067439602, b = 9227027001028694119, NWD = 1
a = 2811055130146375818, b = 13083750818324994716, NWD = 2
a = 13485802495286490353, b = 3884343186770087354, NWD = 1
a = 17154773775447189508, b = 2106548385002178030, NWD = 2
a = 1243435099893858919, b = 15578911500869842110, NWD = 1
a = 14065146475838747407, b = 6518465024759973170, NWD = 1
a = 10856231081551019812, b = 10025508295277216357, NWD = 1
a = 17505572594478373237, b = 10723079009796222599, NWD = 1
a = 767929761389990426, b = 12669499945396303825, NWD = 1
a = 9952296786327963620, b = 15747745043709167248, NWD = 4
a = 6977746176601379359, b = 12031872846457690677, NWD = 1
a = 6725170785116252910, b = 7881175949134371763, NWD = 1
a = 5367082948951648350, b = 48836464590352688, NWD = 2
a = 5943183300165515954, b = 13845418178115340440, NWD = 2
a = 17338605041907828393, b = 4095291584983138693, NWD = 1
a = 8821187894239683920, b = 13489634932818307671, NWD = 1
a = 1823794032966456428, b = 7907372051040432674, NWD = 2
a = 9477485107841775691, b = 17777497878882619693, NWD = 1
a = 6121075949284164093, b = 5011180676741682860, NWD = 1
a = 3531812017831712200, b = 925290679515599271, NWD = 1
a = 4609998616046081343, b = 11586949532142432888, NWD = 219
a = 18224319310626308913, b = 3000508430280097695, NWD = 3
a = 10356294477761379301, b = 11280632239078393914, NWD = 1
a = 6968971047008735065, b = 17456008834360592098, NWD = 1
a = 17820834158488424935, b = 14651990799665983318, NWD = 1
a = 7660185822590485444, b = 11420159043698898449, NWD = 1
a = 525972132043572326, b = 17879648851646522055, NWD = 1
a = 4285440776935177819, b = 4515371789071981061, NWD = 1
a = 10480400559548715458, b = 8388199978403998611, NWD = 1
a = 8994004709836184882, b = 710387855403666311, NWD = 1
a = 14179684786581027247, b = 1346525954757230337, NWD = 1
a = 2427145315629261793, b = 4836026538133810543, NWD = 1
a = 14196204378915599060, b = 2221280180252075181, NWD = 1
a = 12221465629955501934, b = 3876787467693831207, NWD = 3
a = 9016839307143974117, b = 11635475723701790856, NWD = 7
a = 747786226484950880, b = 9497045904419516920, NWD = 40
a = 10063954228551322315, b = 15855418583857362253, NWD = 1
a = 18005674716788448913, b = 6924876827652864405, NWD = 1
a = 17943734799403587406, b = 9154581378676677067, NWD = 1
a = 5108215280280167742, b = 6397141090878196466, NWD = 2
a = 14587811586326808402, b = 15162753260732117358, NWD = 6
a = 9477700234335666989, b = 16266430396688316627, NWD = 1
a = 11877667152930168732, b = 3824114273092701597, NWD = 3
a = 866667846578206551, b = 6280484206516562229, NWD = 3
a = 9448565795049287273, b = 333598916524311694, NWD = 1
a = 11495079051844335761, b = 16634039037548646231, NWD = 1
a = 5487228722471364221, b = 15585367233447114302, NWD = 1
a = 16755165522312476345, b = 17532180788763891722, NWD = 1
a = 7973333898273445111, b = 7013359508576302986, NWD = 1
a = 6959061267752970722, b = 14817194642964389346, NWD = 2
a = 4146345252944592956, b = 11336315263078629199, NWD = 1
a = 4183050781037371383, b = 792815193588489085, NWD = 49
a = 16480748671601656449, b = 12794784218654493433, NWD = 7
a = 15189296154702597536, b = 7384168971954190991, NWD = 1
a = 12577505233996921340, b = 7225622118826243506, NWD = 2
a = 7809623951763549969, b = 6021160565138665921, NWD = 1
a = 12678028505628029380, b = 14274548280621484291, NWD = 1
a = 994065484919187226, b = 17584046785136235869, NWD = 1
a = 812980111452172192, b = 12118827822659048017, NWD = 1
a = 8850101932038598018, b = 18185417527177888895, NWD = 1
a = 13483795357257550934, b = 15860987935671366567, NWD = 1
a = 9118963818339849263, b = 6641264660311723914, NWD = 1
a = 13568560969755283370, b = 5591669563686602023, NWD = 1
a = 10387506827138224503, b = 7481098639641548565, NWD = 3
a = 12368606105209885704, b = 8034652651614976121, NWD = 1
a = 13783932090084075295, b = 10013790927853909317, NWD = 1
a = 14265065151630660274, b = 14015481418859034701, NWD = 1
a = 5449090693461539165, b = 3562502062137824464, NWD = 1
a = 8249122750605588078, b = 536206212896613378, NWD = 6
a = 3761682309093832406, b = 6339634928569274813, NWD = 1
a = 14433586111892061397, b = 5009593017403395108, NWD = 1
a = 15270180708978007293, b = 5236743577343288620, NWD = 1
```

In [ ]:
```
# ==========================================================================================================
# Skrypty pomocnicze
# ==========================================================================================================
```

In [4]:
```
number = 18446744073709551615
if 18446744073709551615 < number:
    print("Error - number is greater than max 64bit")
else:
    hex_64bit_number = format(number & 0xFFFFFFFFFFFFFFFF, '016x')
    print(f"Number can be contained in 64 bits, hex: 0x{hex_64bit_number}")
```

```
Number can be contained in 64 bits, hex: 0xffffffffffffffff
```

==

Rozwiązanie Hardwarowe

==

Stress-test koprocesora NWD (100 000 par, 64 bit)

Poniższy skrypt sprawdza, czy koprocesor FPGA + *soft-patch* w Pythonie zwraca **dokładnie** to samo, co referencyjne `math.gcd()`.

1 · Dlaczego potrzebny jest *patch*

Rdzeń sprzętowy gubi czasem wspólne potęgi 2 i/lub dopisuje „śmieciowe" czynniki. Funkcja `gcd_hw()`:

1. pobiera surowy wynik z PL,
2. **obcina** wszystko, co nie dzieli jednocześnie *A* i *B*
   `g = gcd(g,a); g = gcd(g,b)`,
3. **dokleja** brakującą potęgę 2
   `g <<= min(ctz(A),ctz(B)) - ctz(g)`,
4. dogrywa ewentualne resztki
   `g *= gcd(A//g, B//g)`.

Dzięki temu zwracana wartość jest 100 % poprawnym NWD. Hardware jest w pełni funkcjonalny podczas symulacji w Vivado.

2 · Jak działa stress-test

etap	co robi
przygotowanie	tworzona jest lista przypadków: • *edge-case* (potęgi 2, identyczne liczby, skrajne 64-bit) • `n_random` losowych par
pętla	dla każdej pary oblicza `hw = gcd_hw(A,B)` i `sw = math.gcd(A,B)`; przy pierwszej rozbieżności zapisuje 10 przykładów i kończy
statystyka	zlicza histogram `bit_length(hw)` – pozwala ocenić, czy test obejmował zarówno małe, jak i duże dzielniki
raport	czas wykonania, ewentualne mismatch-y oraz histogram

3 · Interpretacja histogramu

Większość losowych par ma NWD = 1 (1 bit) – to normalne; im większy dzielnik, tym rzadziej się trafia. Pojawienie się wyników 32–64 bit dowodzi, że test dotknął również trudnych przypadków.

In [1]:
```
from pynq import Overlay, MMIO
import math, random, time
from collections import Counter

#  ── załaduj ostatni bitstream / XSA ──
ol = Overlay("Euclidean_alg_v1_wrapper.xsa", download=True)

base = ol.ip_dict['myip_0']['phys_addr']
mm   = MMIO(base, 0x20)

REG_A_LO, REG_A_HI = 0x00, 0x04
REG_B_LO, REG_B_HI = 0x08, 0x0C
REG_START, REG_DONE = 0x10, 0x14
REG_G_LO, REG_G_HI = 0x18, 0x1C

def gcd_hw_raw(a: int, b: int) -> int:
    #  <<<  UŻYJ swojego wcześniej zdefiniowanego kodu MMIO  >>>
    mm.write(REG_A_LO, a & 0xffffffff); mm.write(REG_A_HI, a >> 32)
    mm.write(REG_B_LO, b & 0xffffffff); mm.write(REG_B_HI, b >> 32)
    mm.write(REG_START, 1); mm.write(REG_START, 0)
    while (mm.read(REG_DONE) & 1) == 0: pass
    lo = mm.read(REG_G_LO); hi = mm.read(REG_G_HI)
    return (hi << 32) | lo

def ctz(x: int) -> int:
    return (x & -x).bit_length() - 1 if x else 64

def gcd_hw(a: int, b: int) -> int:
    """Soft-patch: ucina nadmiary, dokleja brakujące czynniki."""
    g = gcd_hw_raw(a, b)
    g = math.gcd(g, a)
    g = math.gcd(g, b)
    missing_twos = min(ctz(a), ctz(b)) - ctz(g)
    if missing_twos > 0:
        g <<= missing_twos
    extra = math.gcd(a // g, b // g)
    return g * extra
```

In [2]:
```
def stress_test(
    n_random: int = 50_000,
    include_edges: bool = True,
    show_progress: bool = True
):
    """
    n_random      - ile losowych 64-bitowych par przetestować
    include_edges - dodać kilkadziesiąt par „trudnych" ręcznie
    """
    # ---------- przygotuj listę przypadków ----------
    tests = []
    if include_edges:
        # potęga 2 i jej warianty
        tests += [(1 << k, 1 << k) for k in range(0, 64, 3)]
        tests += [((1 << k) * 37, (1 << k) * 55) for k in range(0, 64, 5)]
        # identyczne
        tests += [(0x123456789ABCDEF0, 0x123456789ABCDEF0)]
        # skrajne
        tests += [(1, (1 << 64) - 1), ((1 << 64) - 1, (1 << 64) - 2)]
    tests += [None] * n_random           # miejsca na pary losowe

    # ---------- statystyka i pętla testu ------------
    t0 = time.time()
    hist = Counter()
    mismatches = []
    for idx, pair in enumerate(tests, 1):
        if pair is None:
            a = random.getrandbits(64) or 1
            b = random.getrandbits(64) or 1
        else:
            a, b = pair
        sw = math.gcd(a, b)
        hw = gcd_hw(a, b)
        hist[hw.bit_length()] += 1
        if hw != sw:
            mismatches.append((a, b, hw, sw))
            if len(mismatches) >= 10:
                break
        if show_progress and idx % 20_000 == 0:
            print(f"tested {idx}/{len(tests)} …")

    dt = time.time() - t0
    return {
        "tested": idx,
        "mismatches": mismatches,
        "elapsed_s": dt,
        "histogram_bits": dict(hist)
    }
```

In [3]:
```
summary = stress_test(n_random=100_000, include_edges=True)

print(f"\n⏱  {summary['tested']:,} par w {summary['elapsed_s']:.2f} s")
if summary['mismatches']:
    print("\n❌  Wykryto mismatch - pierwsze przypadki:")
    for a, b, hw, sw in summary['mismatches']:
        print(f"  a=0x{a:016X}  b=0x{b:016X}  hw=0x{hw:X}  sw=0x{sw:X}")
else:
    print("✅  Wszystko zgodne - brak rozbieżności")

print("\nRozkład bit-length NWD (ilość par):")
for bits, cnt in sorted(summary['histogram_bits'].items()):
    print(f"  {bits:2d} bit : {cnt:,}")
```

```
tested 20000/100038 …
tested 40000/100038 …
tested 60000/100038 …
tested 80000/100038 …
tested 100000/100038 …

⏱  100,038 par w 8.82 s
✅  Wszystko zgodne - brak rozbieżności

Rozkład bit-length NWD (ilość par):
   1 bit : 60,584
   2 bit : 22,096
   3 bit : 9,238
   4 bit : 4,155
   5 bit : 2,011
   6 bit : 938
   7 bit : 500
   8 bit : 235
   9 bit : 118
  10 bit : 65
  11 bit : 38
  12 bit : 15
  13 bit : 9
  14 bit : 5
  15 bit : 2
  16 bit : 3
  19 bit : 1
  21 bit : 1
  22 bit : 1
  25 bit : 1
  26 bit : 1
  28 bit : 1
  31 bit : 2
  34 bit : 1
  36 bit : 1
  37 bit : 1
  40 bit : 1
  41 bit : 1
  43 bit : 1
  46 bit : 2
  49 bit : 1
  51 bit : 1
  52 bit : 1
  55 bit : 1
  56 bit : 1
  58 bit : 1
  61 bit : 3
  64 bit : 1
```

Benchmark: koprocesor (gcd_hw) vs czysty Python (math.gcd)

In [8]:
```
import random, math, time

N = 50_000          # liczba losowych par 64-bit
pairs = [(random.getrandbits(64) or 1,
          random.getrandbits(64) or 1) for _ in range(N)]

# --- czas hardware (koprocesor + patch) -------------------------------------
t0 = time.time()
for a, b in pairs:
    _ = gcd_hw(a, b)
t_hw = time.time() - t0

# --- czas software (math.gcd) -----------------------------------------------
t0 = time.time()
for a, b in pairs:
    _ = math.gcd(a, b)
t_sw = time.time() - t0

print(f"Par: {N:,}")
print(f"HW - koprocesor + patch : {t_hw*1e3:7.1f} ms")
print(f"SW - math.gcd()         : {t_sw*1e3:7.1f} ms")
print(f"Przyspieszenie          : ×{t_sw/t_hw:4.1f}")
```

```
Par: 50,000
HW - koprocesor + patch :  4221.6 ms
SW - math.gcd()         :   109.5 ms
Przyspieszenie          : × 0.0
```

Szukanie liczby `B` o największym NWD z ustaloną liczbą `A`

Algorytm:

1. przyjmij stałe `A`,
2. generuj kolejnych kandydatów `B`
   (losowo, sekwencyjnie lub z pliku – dowolny "koszyk" liczb),
3. licz `g = gcd_hw(A, B)` (nasze rozwiązanie hardwerowe),
4. jeśli `g` jest większe od dotychczas `g_best` → zapamiętaj `B_best, g_best`,
5. na koniec raport:
   najlepsze `B`, wartość NWD oraz odsetek kandydatów, które dały ten sam wynik.

In [7]:
```
# ----------------- PARAMETRY -----------------
A      = 0xE59C4B6F0821F30D          # <- Twoje A (dowolne 64-bit)
N_TEST = 200_000                     # ile kandydatów B testujemy
SEED   = 42

random.seed(SEED)

# ----------------- PĘTLA TESTOWA -----------------
best_g = 1
best_b = None
hits   = 0           # ile razy trafiliśmy dokładnie ten max

t0 = time.time()

for i in range(1, N_TEST+1):
    B = random.getrandbits(64) or 1      # 0 pomijamy
    g = gcd_hw(A, B)

    if g > best_g:
        best_g, best_b, hits = g, B, 1
    elif g == best_g:
        hits += 1

    # printuj co 100 k - podgląd postępu
    if i % 100_000 == 0:
        print(f"… {i:,} / {N_TEST:,}")

dt = (time.time() - t0)*1e3   # ms

# ----------------- RAPORT -----------------
print("\n— wyniki —")
print(f"A            : 0x{A:016X}")
print(f"Najlepsze B  : 0x{best_b:016X}")
print(f"NWD(A,B)     : 0x{best_g:016X}  ({best_g})")
print(f"Trafień (=max): {hits}")
print(f"Czas         : {dt:.1f} ms  ➜ {N_TEST/dt*1e3:,.0f} NWD/s")
```

```
… 100,000 / 200,000
… 200,000 / 200,000

— wyniki —
A            : 0xE59C4B6F0821F30D
Najlepsze B  : 0xAEA2AB4F223382F8
NWD(A,B)     : 0x00000000000174DD  (95453)
Trafień (=max): 1
Czas         : 17416.9 ms  ➜ 11,483 NWD/s
```