# Pomyśl komputacyjnie zanim zaczniesz programować

**Artykuł jest komentarzem do tekstu „O sztuce programowania" (Domena 2/2023) Janusza Zalewskiego[1], w którym wyjaśnia i uzasadnia, w jakim sensie programowanie jest i może być sztuką, opierając się na opiniach czołowych informatyków lat 70. XX w. Wcześniej, gdy informatyka dopiero stawała się samodzielną dyscypliną, Alan Perlis[2] uzasadniał, że informatykę należy uważać za część edukacji ogólnej i że każdy powinien nauczyć się programować (ang. *computer science should be considered part of a liberal education, and that everyone should learn to program*)[3]. Twierdził, że programowanie jest podstawową umiejętnością intelektualną, podobnie jak matematyka, a komputery „będą uczestniczyć w niemal każdym intelektualnym przedsięwzięciu".**

**Maciej M. Sysło**

matematyk i informatyk, wykładowca w Warszawskiej Wyższej Szkole Informatyki. W połowie lat 60. XX w. przyglądał się pierwszym zajęciom z programowania w dwóch LO we Wrocławiu, a w latach 80. włączył się w przygotowanie uczniów, nauczycieli i szkół na ekspansję komputerów w edukacji. Za swój największy sukces uznaje utrzymanie, wbrew tendencjom w kraju i za granicą, wydzielonych zajęć z informatyki w szkołach, a ostatnio – objęcie nauczaniem informatyki i programowania wszystkich uczniów na wszystkich poziomach edukacyjnych. Teraz upowszechnia myślenie komputacyjne jako uzupełnienie tradycyjnych kompetencji 3R – czytania, pisania i rachowania. Więcej na http://mmsyslo.pl oraz https://pl.wikipedia.org/wiki/Maciej_Marek_Sysło.

Do tego wizjonerskiego – wtedy – przekonania o roli programowania i informatyki nie trzeba już specjalnie przekonywać. Powszechne stało się nawoływanie do nauki programowania wszystkich uczniów, jako umiejętności przydatnej w ich dalszym kształceniu i rozwoju zawodowym oraz życiu osobistym. Wykreowano w tym celu środowisko **myślenia komputacyjnego**, wspomagające proces dochodzenia do takich umiejętności, które faktycznie jest poszerzeniem znanego podejścia do rozwiązywania problemów z wykorzystaniem myślenia algorytmicznego, obecnego w podstawie programowej szkolnej informatyki w naszym kraju od połowy lat 90. XX w. Znaczną część mojej wypowiedzi ilustruję przykładami programów, które w pewnym sensie są piękne jako dzieła sztuki programowania.

1 J. Zalewski, O sztuce programowania, *Domena* 2/2023, 66–68.

2 Alan Perlis (1922–1990). Uznawany za ojca informatyki (ang. *computer science*) jako samodzielnej dyscypliny akademickiej. Twórca systemów i języków programowania, m.in. języka Algol 60, który odegrał przełomową rolę w rozumieniu programowania nie tylko jako praktycznej umiejętności pracy z komputerem, lecz również jako obiektu badań. Uznawał języki programowania i algorytmy za centralne pojęcia informatyki. W latach 1962–1964 był prezesem amerykańskiego towarzystwa informatycznego ACM i to za jego kadencji ACM opublikowało pierwsze zalecenia dla studiów informatycznych na poziomie licencjackim. W uznaniu zasług, w 1966 roku Alan Perlis otrzymał jako pierwszy Nagrodę Turinga.

2 Jest to konkluzja z wystąpienia Alana Perlisa na konferencji z okazji 100-lecia MIT, zamieszczona w M. Guzdial, E. Soloway, Computer Science Is More Important Than Calculus: The Challenge of Living Up to Our Potential, *inroads* 2/35, 2003, 5–8, Pełny tekst wystąpienia: A.J. Perlis, The Computer in the University, w: M. Greenberger (ed.), *Computers and the World of the Future*, MIT Press, 1962, 180–217.

## Sztuka programowania

Podobnie jak pionier informatyki Donald Knuth, większość zawodowych informatyków, bliskich akademii, uzasadnia, że programowanie jest sztuką (przyjmując również argumenty innych, jak: Dijkstry, Bentleya czy Kernighana). Sam Knuth w swoim wielotomowym dziele[4] akcentuje przede wszystkim podejście matematyczne, a w rezultacie – algorytmiczne do stosowania komputerów. Wystarczy przytoczyć tytuły kolejnych tomów: 1. *Fundamental Algorithms*, 2. *Seminumerical Algorithms*, 3. *Sorting and Searching*, 4. *Combinatorial Algorithms*, 5. *Syntactical Algorithms*. Dla zainteresowanych programowaniem ma radę (Tom 1, przedmowa, str. xii): „Czytelnik, który jest zainteresowany przede wszystkim programowaniem, a nie związaną z nim matematyką, może przestać czytać większość rozdziałów, gdy tylko matematyka stanie się wyraźnie trudna." (ang. *A reader who is interested primarily in programming rather than in the associated mathematics may stop reading most sections as soon as the mathematics becomes recognizably difficult*). Dla osób zainteresowanych programowaniem dopiero na stronie 120 pierwszego tomu zdefiniował język MIX dla nieistniejącej maszyny o tej samej nazwie, będącej jednocześnie maszyną binarną i dziesiętną. Można mieć wątpliwości, na ile MIX był i jest wykorzystywany przez czytelników jego dzieła – osobom zainteresowanym matematyką i algorytmiką programy w tym języku niewiele wnoszą, z kolei zainteresowani głównie programowaniem – do tych programów nie przedrą się przez gąszcz dość zaawansowanej matematyki.

Reasumując, dzieło Knutha oferuje czytelnikom – zarówno informatykom, jak i programistom – głównie sztukę programowania w rozumieniu sztuki algorytmizowania. Tego wątku związanego z programowaniem Janusz Zalewski dotyka w ostatnim punkcie swojej wypowiedzi.

## Myślenie komputacyjne

Pojęcie **myślenia komputacyjnego** (ang. *computational thinking*) zrobiło w ostatniej dekadzie zawrotną karierę. Pojawia się niemal w każdym dokumencie, publikacji i prezentacji dotyczących zwłaszcza edukacji informatycznej i jej powiązań z innymi edukacjami. Można odnieść wrażenie, że to inna nazwa tego, co wcześniej występowało pod nazwą „myślenie algorytmiczne", jednak jest to „nowa jakość" w kształceniu, przede wszystkim informatycznym, które kładzie fundamenty pod zrozumienie i stosowanie tego sposobu myślenia z jednoczesnym otwarciem szerokiego pola do jego stosowania we wszystkich innych przedmiotach szkolnych i dziedzinach wiedzy oraz aktywności.

Niektórzy w propozycji myślenia komputacyjnego, adresowanego do wszystkich, widzą analogię z ideą z początków lat 60., gdy Alan Perlis twierdził, że każdy powinien nauczyć się programować w ramach edukacji powszechnej, widząc w programowaniu sposób na zrozumienie „teorii obliczeń", co z kolei spowoduje zrozumienie szerokiej gamy zagadnień z różnych dziedzin w terminach obliczeń.

**Dzisiaj jest już truizmem stwierdzenie, że automatyzacja (komputeryzacja) procesów zmienia postrzeganie – zarówno przez profesjonalistów, jak i zwykłych ludzi – swojej pracy i roli w swoich społecznościach i społeczeństwie. Komputery mają wpływ na wszystkie dziedziny życia i będzie się on tylko powiększał, należy więc wszystkich przygotować do korzystania z tego narzędzia i wyposażyć w towarzyszące procesy myślowe, uczulając jednocześnie na aspekty społeczne tych rewolucyjnych zmian, powodowanych przez rozwój technologii. Głównym celem integracji myślenia komputacyjnego w edukacji, obok przygotowania uczniów do zawodów jutra z naciskiem na rozwiązywanie problemów, jest przede wszystkim kształtowanie sposobów myślenia z tym związanych.**

Już w 1980 r., a później w 1996, o myśleniu komputacyjnym pisał Seymour Papert, wyprzedzając swoimi ideami konstruktywistycznymi możliwości technologii[5]. Jeannette Wing ożywiając myślenie komputacyjne w 2006 r.[6], określiła tym terminem „użyteczne postawy i umiejętności, jakie każdy, nie tylko informatyk, powinien starać się wykształcić i stosować" (ang. *a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use*). Chociaż myślenie komputacyjne uznała wtedy za skrót od „myśleć jak informatyk"[7] (ang. *thinking like a computer scientist*), pod wpływem szerokiej dyskusji na temat definicji i znaczenia tego terminu przyjęła później, że:

4 D.E. Knuth, *Sztuka programowania*, Tomy 1–3 , WNT, Warszawa 2002.

5 M.M. Sysło, Ewolucja urządzeń i przyrządów do obliczeń na drodze do komputerów w edukacji, *Meritum* 4(67) 2022, 16–24.

6 J.M. Wing, Computational thinking, *Comm. ACM* 3/49, 2006, 35–37.

7 Peter Denning (2009), informatyk, starał się ostudzić zapędy informatyków: „Wy, informatycy, jesteście nienasyceni! Najpierw chcecie, abyśmy uczestniczyli w waszych kursach dotyczących alfabetyzacji i biegłości [komputerowej]. A teraz chcecie, abyśmy myśleli jak wy".

> *Myślenie komputacyjne to procesy myślowe angażowane w formułowanie problemu i przedstawianie jego rozwiązań w taki sposób, aby komputer*[8] *– a najczęściej człowiek z maszyną – mógł skutecznie wykonać.*

Jest to obecnie najpowszechniej przyjmowane określenie myślenia komputacyjnego jako aktywności umysłowych towarzyszących rozwiązywaniu problemów i rozumieniu przy tym ludzkich zachowań poprzez odwoływanie się do podstawowych pojęć informatyki. Wing podkreśliła, że myślenie komputacyjne jest związane z tym, jak ludzie myślą, jak ludzie rozwiązują problemy, a nie ma na celu skłonienia ludzi, by myśleli tak, jak (działają) komputery. To ludzie nadają dopiero moc komputerom i wykorzystują je w rozwiązywaniu problemów na swój sposób. To umiejętność człowieka myślenia z komputerem jako narzędziem.

Wing uznała też, że myślenie komputacyjne stanowi naturalne poszerzenie[9] kompetencji określanych jako 3R (**R**eading, w**R**iting, a**R**ithmethic) o jednoczesne rozwijanie umiejętności stosowania metod pochodzących z informatyki i analitycznego myślenia przy rozwiązywaniu problemów pochodzących z różnych dziedzin. Dodała przy tym swoją wizję: tak jak prasa drukarska przyczyniła się do rozprzestrzeniania się tych 3R, tak komputery przyczynią się do rozpowszechniania się myślenia komputacyjnego.

Podana definicja myślenia komputacyjnego jest bardzo ogólna, zwłaszcza dla nauczycieli, którzy chcieliby ją przełożyć na praktyczne działania w klasie, a także dla uczniów, którzy mieliby nabywać i posługiwać się tym sposobem myślenia podczas rozwiązywania problemów. Doceniając znaczenie myślenia komputacyjnego dla uczniów i dla nauczycieli, głównie jako zestawu praktyk, które wymagają zarówno wiedzy, jak i umiejętności, należy odpowiednio przełożyć, rozwinąć i skomentować, czym jest, by zostało ono uwzględnione w podstawach i programach nauczania, a także mogło być odpowiednio oceniane.

Z podanej definicji myślenia komputacyjnego wynika, że są to procesy myślowe prowadzące do realizacji (implementacji) rozwiązania za pomocą komputera lub w inny sposób. Musimy jednak być świadomi, że zanim posłużymy się komputerem, sam problem należy najpierw zrozumieć i dobrze określić oraz znaleźć sposób jego rozwiązania. Temu właśnie służy myślenie komputacyjne, co jest odzwierciedlone w kolejnych etapach jego operacyjnej definicji, którą podajemy tutaj w wersji występującej w naszej podstawie programowej informatyki już w 1997 r. pod nieco inną nazwą. Ten fakt jest potwierdzeniem opinii Denninga[10], że: „Myślenie komputacyjne ma długą historię w informatyce. Znane w latach 50. i 60. jako «myślenie algorytmiczne», oznacza mentalną orientację na formułowanie problemów jako konwersję pewnego wejścia na wyjście i szukanie algorytmów do przeprowadzenia tej konwersji" (ang. *Computational thinking has a long history within computer science. Known in the 1950s and 1960s as „algorithmic thinking", it means a mental orientation to formulating problems as conversions of some input to an output and looking for algorithms to perform the conversions*)[11].

**Uczeń:**

- definiuje sytuację problemową w postaci jej specyfikacji określającej dane [*abstrakcja*], oczekiwane wyniki i związki między danymi i wynikami [*logiczne myślenie*];
- projektuje plan rozwiązania problemu – wyodrębnia podproblemy [*dekompozycja*] i wskazuje powiązania między nimi;
- wybiera sposób rozwiązania problemu:
  - tworzy algorytm [*myślenie algorytmiczne*],
  - wykorzystuje istniejące oprogramowanie lub programuje metodę rozwiązywania w wybranym języku programowania [*abstrakcja, implementacja, programowanie*];
- analizuje poprawność algorytmu i jego implementacji [*debugowanie*], testuje program [*testowanie*];
- ocenia poprawność rozwiązania [*logiczne myślenie*] i jego efektywność oraz złożoność [*ewaluacja*];
- rozwiązuje złożone projekty w zespole [*współpraca*];
- wybiera i rozwiązuje problemy z różnych przedmiotów szkolnych [*zastosowania informatyki w innych przedmiotach, uogólnianie*].

8 Na podstawie klasycznych źródeł, np. (*Webster's New World Dictionary*, 1969): *computer* to: 1. *a person who computes*; 2. *a device used for computing*.

9 Dopisywanie kolejnych umiejętności do 3R należy czynić z umiarem. Na przełomie wieków XX/XXI 3R uzupełniano o TI, czyli zastosowania informatyki, głównie posługiwanie się gotowymi aplikacjami – przeżywaliśmy to w Polsce, chociaż informatyka nigdy nie zniknęła z podstaw programowych. Obecnie pojawiają się propozycje dopisania do 3R kolejnego **R** dla uznania znaczenia algo**R**ytmiki, a także **P** od **P**rogramowania, w obu przypadkach byłoby to znaczącym zawężeniem pełnego znaczenia myślenia komputacyjnego.

10 P. J. Denning, Beyond Computational Thinking, *Comm. ACM* 6/52, 2009, 28-30

11 M.M. Sysło, Informatyka z komputerem w tle (*unplugged revisited*), *W cyfrowej szkole*, 3/2023

W powyższej wersji określenia, czym jest myślenie algorytmiczne, zostały wstawione (pisane kursywą) niektóre sposoby rozumowania, których zespół uznaje się za inną definicję myślenia komputacyjnego[12]. Myślenie komputacyjne jako myślenie algorytmiczne ma więc długą tradycję w kształceniu informatycznym w Polsce. W następnych latach te założenia programowe powtarzano z niewielkimi przeformułowaniami, skierowano je również do szkół podstawowych i gimnazjów. Zostały uwzględnione w obowiązującej obecnie podstawie programowej.

Myślenie komputacyjne, zwłaszcza w wersji operacyjnej definicji, wpisuje się w opinię Władysława M. Turskiego, cytowaną przez Janusza Zalewskiego: „przede wszystkim musimy uczyć myślenia problemowego… aby nasi programiści myśleli w kategoriach problemów, które rozwiązują, a nie w kategoriach technik programowania, które stosują."

## Przykłady pięknych programów

Pokusiłem się o osobisty wybór kilku programów. Ich piękno krótko uzasadniam, podaję jednak same kody, bez ich wyprowadzania, bez matematyki i algorytmiki, bardzo prostej, która za nimi stoi, to na ogół są znane fakty.

### Schemat Hornera

Schemat Hornera to metoda obliczania wartości wielomianu o znanych współczynnikach (w programie są one umieszczone na liście a). Podał go William Horner w 1819 r., wcześniej znał go Isaac Newton i oczywiście Chińczycy. Sam program jest piękny! Łatwo zauważyć, że oblicza wartość wielomianu stopnia $n$ za pomocą $n$ mnożeń i $n$ dodawań. Allan Borodin udowodnił w 1971 r., że nie da się szybciej obliczać wartości wielomianu, czyli za pomocą mniejszej liczby działań – i to jest ekstra piękno tego algorytmu i programu. Dodatkowo, to jego piękno podkreśla ogrom zastosowań w obliczeniach numerycznych, bowiem wartości każdej funkcji są obliczane jako wartości odpowiednich wielomianów.

```
# Schemat Hornera
def Horner(a,x):
  y=a[0]
  for i in range(1,len(a)):
    y=y*x+a[i]
  return(y)
```

### Trójkąty

Program obok rozwiązuje zadanie z I Olimpiady Informatycznej[13] w uproszczonej wersji: dany jest ciąg liczb naturalnych większych od zera będących długościami boków trójkąta i należy zbadać, czy z każdych trzech odcinków z tego zbioru można zbudować trójkąt. Program może nie jest piękny, ale jest wynikiem bardzo ładnego rozumowania.

Każdy uczeń zna tzw. warunek trójkąta, którego spełnienie gwarantuje, że z trzech odcinków można zbudować trójkąt: suma długości każdych dwóch odcinków powinna być większa niż długość trzeciego odcinka. W czym więc tkwi problem, by zaprogramować rozwiązanie tego zadania? Po pierwsze, jeśli $n$ jest liczbą danych, to sprawdzenie warunku trójkąta dla wszystkich trójek zajmie czas rzędu $n^3$, a rozwiązania zadań olimpijskich są oceniane również pod względem efektywności działania. Po drugie, liczba danych może być tak duża, że nie zmieszczą się w pamięci, do której miały dostęp programy w języku Pascal (było to w latach 1993/1994). Miejsce więc na błysk programisty! Otóż, z trzech odcinków można zbudować trójkąt wtedy i tylko wtedy, gdy suma długości dwóch najkrótszych odcinków jest większa niż długość najdłuższego odcinka[14]. Dla rozwiązania powyższego zadania wystarczy więc znaleźć w ciągu liczby: najmniejszą, drugą najmniejszą i największą – wystarczy w tym celu jednokrotnie przejrzeć ciąg danych – a następnie sprawdzić, czy suma dwóch pierwszych jest większa od trzeciej, co też robi nasz program. Złożoność tego algorytmu/programu jest więc liniowa względem $n$ i to jest piękne w tym rozwiązaniu tego zadania.

```
# Boki trójkątów
def Trójkąty():
  min1,min2,mxa=10e6,10e6,-1
  filepath="dane.txt"
  f=open(filepath,"r")
  for line in f:
    a=int(line)
    if a < min2:
      if a < min1:
        min2,min1=min1,a
      else:
        min2=a
    else:
      if mxa < a:
        mxa=a
  if min1+min2 > mxa:
    print('Tak')
  else:
    print('Nie')
```

12 J.M. Wing, *Computational Thinking Benefits Society*, 2014, http://socialissues.cs.toronto.edu/index.html%3Fp=279.html

13 Autorem zadania jest Piotr Chrząstowski-Wachtel. M.M. Sysło (red.), *I Olimpiada Informatyczna 1993/1994*, Warszawa, Wrocław 1994, 36-42.

14 Szkoda, że na lekcji matematyki (geometrii) w szkole nie podaje się takiej wersji warunku trójkąta.

Czasem używam tego zadania, by zademonstrować głównie matematykom różnicę między myśleniem i rozwiązaniem matematycznym (rzędu $n^3$) a informatycznym (liniowe). Niestety, ku mojemu zdziwieniu, nie powala to ich z nóg.[15]

## Liczby Fibonacciego

Generowanie liczb Fibonacciego to jedno ze stałych zadań, obok obliczania silni, którymi zarzucani są uczniowie na lekcjach informatyki. W obu przypadkach celem jest zachęcenie ich do posłużenia się rekurencją, mimo że jedne i drugie liczby nie mają jakichś specjalnych zastosowań w tym, co uczniowie robią w szkole. Utworzenie programów rekurencyjnych nie nastręcza uczniom specjalnych kłopotów – wystarczy zapisać odpowiednie wzory w notacji języka programowania.

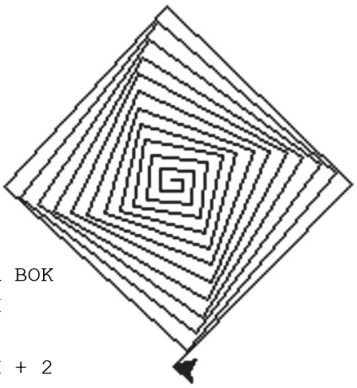
Chociaż rekurencja – jak zatytułowałem jeden z rozdziałów w swojej książce[16] – to metoda „jak zrzucić robotę na komputer", pozostaje jednak pytanie, czy uczniowie dociekają i poznają, jak komputer wykonuje programy rekurencyjne? Czy dostrzegają w realizacji rekurencji właściwy dla niej „model kopiowania", czy raczej posługują się błędnym „modelem pętli", uznając rekurencję za inny sposób realizacji iteracji?

```
# Liczby Fibonacciego rekurencyjnie
def FibRek(k):
  if k <= 2:
    return 1
  else:
    return FibRek(k-1)+FibRek(k-2)
```

Janusz Zalewski cytuje w swoim artykule Briana Kernighana, według którego […] *rekurencja to strzał w dziesiątkę. Ta fundamentalna technika programistyczna prawie zawsze prowadzi do powstania mniejszego, przejrzystego i bardziej eleganckiego kodu niż jego odpowiednik napisany przy zastosowaniu pętli.* Jednak znajomość realizacji rekurencji w komputerze jest niezbędna, by wiedzieć, jak rozrzutne mogą być takie obliczenia. Ilustruje to obok ślad wywołań funkcji rekurencyjnej przy obliczaniu wartości FibRek(6). Widać wielokrotne obliczanie tych samych wartości $F_3$ i $F_4$.

Rekurencja jest jednak bardzo ważnym sposobem rozumowania i podejściem do projektowania obliczeń, a w ogólności – ekspresji, która może mieć postać graficzną, dźwiękową, multimedialną. Dostrzegł to już w 1980 r. Seymour Papert w swoich *Burzach mózgów*[17] pisząc (str. 94): *Ze wszystkich pomysłów, jakie przedstawiłem dzieciom, rekurencja wyróżnia się jako idea, która potrafi wywołać szczególnie entuzjastyczne reakcje. Myślę, że częściowo dlatego, że idea wiecznego poruszania się pobudza fantazję każdego dziecka, a częściowo ponieważ sama rekurencja ma swoje korzenie w kulturze ludowej*. To wieczne poruszanie się odnosi się do rysowania nieskończonych spiral za pomocą programów z rekurencją ogonową (*tail*), która faktycznie jest wykonywana jak iteracja.

Ślad wywołań przy obliczaniu $F_6$;
Wywołaj: $F_6$
  Wywołaj: $F_5$
    Wywolaj: $F_4$
      Wywołaj: $F_3$
        Pobierz: $F_2$=1
        Pobierz: $F_1$=1
      Zwróć: $F_3=F_2+F_1=1+1=2$
      Pobierz: $F_2$=1
    Zwróć: $F_4=F_3+F_2=2+1=3$
    Wywołaj: $F_3$
      Pobierz: $F_2$=1
      Pobierz: $F_1$=1
    Zwróć: $F_3=F_2+F_1=1+1=2$
  Zwróć: $F_5=F_4+F_3=3+2=5$
  Wywołaj: $F_4$
    Wywołaj: $F_3$
      Pobierz: $F_2$=1
      Pobierz: $F_1$=1
    Zwróć: $F_3=F_2+F_1=1+1=2$
      Pobierz: $F_2$=1
  Zwróć: $F_4=F_3+F_2=2+1=3$
Zwróć: $F_6=F_5+F_4=5+3=8$



```
OTO SPIRALA BOK
NAPRZÓD BOK
PRAWO 91
SPIRALA BOK + 2
```

[15] Wytrwałym czytelnikom polecam inne zadanie o trójkątach z II Olimpiady Informatyczne, którego autorem był również Piotr Chrząstowski-Wachtel. Napisz program, który dla skończonego ciągu dodatnich liczb całkowitych, nie większych niż miliard, reprezentujących długości odcinków, znajduje trzy liczby, że z odpowiadających im odcinków można zbudować trójkąt. *Wskazówka*. Program może mieć stałą złożoność, nie zależną od długości ciągu. W rozwiązaniu przydatne mogą się okazać liczby Fibonacciego.

[16] Patrz rozdz. 5 w książce M.M. Sysło, *Piramidy, szyszki i inne konstrukcje algorytmiczne*, Helion 2015.

[17] S. Papert, *Burze mózgów. Dzieci i komputery*, WN PWN 1996 (oryginał Bacic Books, 1980).

Powyższy program w Logo[18] ilustruje, jak prosto dzieci mogą zapisać w postaci rekurencji rysowanie spirali. Papert proponował przy tym, by najpierw fizycznie przechodziły one po spirali – program jest właśnie zapisem takiego przechodzenia: idź do przodu z krokiem początkowym, skręć trochę bardziej niż w prawo, i kontynuuj z coraz dłuższym krokiem. To podejście, wiążące się z kinestetyczną (ruchową) aktywnością uczniów, wyzwalania myśli i tworzenia wzorców myślenia, bazujące na relacjach ze światem fizycznym, Papert nazwał **syntonicznością ciała**.

Tyle na plus rekurencji, która faktycznie jest świetnym narzędziem do rysowania przepięknych spiral, a zwłaszcza fraktali. Jednak już w takim przypadku, jak liczby Fibonacciego, rekurencyjna realizacja wzoru na te liczby prowadzi do obliczeń w czasie rosnącym wykładniczo – co nie jest już piękne – a wynika z dwukrotnego odwołania po prawej stronie wzoru.

Na szczęście możemy ten wzór zrealizować iteracyjnie, również w postaci krótkiego, przejrzystego i eleganckiego kodu (Kernighan), który zapewnia obliczenia w czasie liniowym względem numeru generowanej liczby.

```
# Liczby Fibonacciego iteracyjnie
def FibIter(k):
  Fk,Fk1=1,1
  for i in range(k-2):
    Fk,Fk1=Fk+Fk1,Fk
  return Fk
```

Czy to już pełnia szczęścia programisty-artysty? Niezupełnie. Czas teraz na odwołanie się do „myślenia logarytmicznego", które można uznać za efekt podejścia dziel i zwyciężaj, popularnie zwanego metodą przez połowienie[19].

W przypadku wyszukiwania elementów w ciągu uporządkowanym dzielimy ciąg na pół i w każdym kroku ciąg maleje o połowę. W rezultacie, liczba kroków w tym wyszukiwaniu w ciągu o $n$ elementach, to ile razy należy podzielić $n$ przez 2, aby otrzymać 1, a to jest inna definicja logarytmu przy podstawie 2 z $n$ (a dokładniej powała z takiej liczby).

W przypadku liczb Fibonacciego wystarczy posłużyć się wzorem rekurencyjnym, w którym są odwołania do tych liczb, ale o indeksach o połowę mniejszych. Mamy takie wzory:

$$F_{2n-1} = F_{n-1}^2 + F_n^2$$
$$F_{2n} = 2F_{n-1}F_n + F_n^2$$
$$F_0 = 0 \text{ i } F_1 = 1$$

Ponieważ ponownie, po prawej stronie są po 2–3 odwołania do liczb Fibonacciego, realizujemy obliczenia w sposób iteracyjny.

```
# Liczby Fibonacciego iteracyjnie Log
def FibLog(k):
  Fib1,Fib2,FibEven,FibOdd=1,0,1,0
  while k > 0:
    if k % 2 != 0:
      Pom=Fib2*FibEven;
      Fib2,Fib1=Fib1*FibEven+Fib2*FibOdd+Pom,Fib1*FibOdd+Pom;
    if k > 1:
      Pom=FibEven*FibEven;
      FibEven,FibOdd=2*FibEven*FibOdd+Pom,FibOdd*FibOdd+Pom
    k=k // 2
  return(Fib2)
```

Program już nie jest ani krótki, ani przejrzysty, ani elegancki, ale jego piękno tkwi w tym, że działa w czasie logarytmicznym względem numeru generowanej liczby i działa zdecydowanie najszybciej, o czym można się przekonać, wykonując na komputerze porównanie z innymi algorytmami dla liczb Fibonacciego[20].

### Algorytm Euklidesa

Na zakończenie czas na klasykę sprzed blisko 2500 lat, czyli algorytm Euklidesa. By znaleźć NWD ($n$,$m$) – Największy Wspólny Dzielnik dwóch liczb $n$ i $m$, $n \geq m$ – odejmujemy mniejszą $m$ od większej $n$, tak długo jak się to da, czyli dzielimy $n$ przez $m$, i powtarzamy ten krok z $m$ i resztą z tego dzielenia, aż mniejsza z liczb stanie się zerem. To dokładnie tak, jak opisał Euklides w swoich wiekopomnym dziele *Elementy*. Oto program z prostą rekurencyjną realizacją tego algorytmu.

```
# Algorytm Euklidesa
def EuklidRek(n,m):
  if m > n:
    return EuklidRek(m,n)
  else:
    if m == 0:
      return n
    else:
      return EuklidRek(m,n % m)
```

18 To niewielka modyfikacja programu z *Burz mózgów* Paperta.

19 M.M. Sysło, A.B. Kwiatkowska, *Myśl logarytmicznie!*, Delta 12/2014, 10-13.

20 Patrz rozdz. 6 w książce M.M. Sysło, *Piramidy, szyszki i inne konstrukcje algorytmiczne*, Helion 2015.

Czy nie jest piękny jako tekst? Uzasadnijmy to głębiej w podobny sposób jak we wcześniejszych przypadkach – tym, co się pod nim kryje. W tabelce:

| *n* | *m* | *reszta* |
|---|---|---|
| **34** | **21** | 13 |
| 21 | 13 | 8 |
| 13 | 8 | 5 |
| 8 | 5 | 3 |
| 5 | 3 | 2 |
| 3 | 2 | **1** |
| 2 | 1 | 0 |

umieszczono wyniki działania tego algorytmu dla dwóch liczb $n = 34$ i $m = 21$. Zwróćmy uwagę na liczby w trzeciej kolumnie – to kolejne reszty z dzielenia. Można zauważyć, że w każdej parze co drugich liczb ta poniżej jest mniejsza od połowy tej powyżej! To kapitalna obserwacja, którą nie tak trudno jest udowodnić np. geometrycznie, proponujemy spróbować. A zatem algorytm Euklidesa jest metodą połowienia w co drugim kroku. Great! Stąd wynika, że złożoność tego algorytmu jest logarytmiczna. Nie przykuło to jednak uwagi Euklidesa, wielka szkoda. Miał szansę stać się przy okazji odkrywcą logarytmu, który wprowadził do matematyki dopiero John Napier w 1614 r. To jest przepiękny fakt w algorytmice – algorytm Euklidesa, który chyba najczęściej był i jest kojarzony z pojęciem algorytm, wynaleziony ponad 2000 lat temu, poddany dzisiejszej analizie okazuje się być algorytmem optymalnym (z dokładnością do stałej proporcjonalności) w sensie złożoności obliczeniowej!

A liczby 34 i 21 użyte w ilustracji tego algorytmu powyżej to dwie kolejne liczby Fibonacciego. I ciekawostka, równie piękna: algorytm Euklidesa wykonuje najwięcej iteracji właśnie dla dwóch kolejnych liczb Fibonacciego. Zbieg okoliczności? Niezupełnie, w kolejnych krokach tego algorytmu dla takich liczb ilorazy są możliwie najmniejsze i wynoszą 1, a ciąg generowanych reszt to ciąg Fibonacciego.

## Uwagi końcowe

Powyższe przykłady ilustrują opinię Janusza Zalewskiego, że „sztukę programowania należy rozumieć jako sztukę rozwiązywania problemów, czyli tworzenia algorytmów", ewentualnie dodatkowo także w postaci ich komputerowej realizacji. Ale sztuką i pięknem nie jest sam program, a ukryta pod nim droga rozumowania – myślenia komputacyjnego – prowadząca do rozwiązania zapisanego w programie.

Na koniec jeszcze komentarz do książek o programowaniu, których tytuły i zawartość sprowokowały Janusza Zalewskiego. Powstaje ich bardzo wiele. To efekt rozwoju istniejących i nowych języków i środowisk programowania, jak również polityki edukacyjnej i rządowej, by każdy, w tym zwłaszcza uczeń, posiadł umiejętność programowania. Strategia myślenia komputacyjnego może gwarantować, że rzeczywiście umiejętności programowania znajdą swoje właściwe miejsce w procesie znajdowania rozwiązań problemów. Kadry edukacji informatycznej postępują tak od dawna, czego przykładem może być jeden z pierwszych podręczników do nauki programowania[21] zespołu z Uniwersytetu Wrocławskiego. We Wstępie autorzy napisali: *Myślą przewodnią książki pozostaje splot dwu zagadnień: prezentacji języka programowania i programowania jako przedmiotu działalności intelektualnej pozostającego w związku, ale i poza wszelkimi konkretnymi językami*." Stąd też tytuł tej książki, „Programowanie i Logo" – a nie „Programowanie w Logo".

[21] E. Gurbiel, H. Krupicka, Z. Płoski, *Programowanie i Logo*, Wydawnictwo Uniwersytetu Wrocławskiego, Wrocław 1987.