Secventa Kolakoski

Daniel Mitrache

Martie 2025

1 Introducere

Secvența Kolakoski este o succesiune autoreferențială de numere formată doar din valorile 1 și 2, având proprietatea distinctivă că fiecare element descrie lungimea unui bloc consecutiv de termeni în secvență. Definită inițial de William Kolakoski în 1965, această secvență prezintă un comportament aparent aleatoriu, deși urmează reguli bine determinate. Studiată în matematică recreativă și teoria numerelor, secvența Kolakoski ridică întrebări interesante despre structura și distribuția sa, fiind subiect de cercetare în analiza combinatorică și teoria fractalilor. Acest document explorează proprietățile, generarea și aplicațiile secvenței Kolakoski.

2 Definiția și Proprietăți

2.1 Definiția Secvenței Kolakoski

Secvența Kolakoski este o succesiune infinită formată din valorile 1 și 2, cu proprietatea că descrie lungimea propriilor blocuri consecutive. Secvența începe astfel:

1, 2, 2, 1, 1, 2, 1, 2, 2, 1, 2, 2, 1, 1, 2 ...

Aceasta poate fi definită formal astfel: al n-lea termen al secvenței indică lungimea celui de-al n-lea bloc consecutiv de numere identice. Primele poziții generează următoarele grupuri:

- $1 \to \{1\}$
- $2 \to \{2, 2\}$
- $2 \to \{1, 1\}$
- $1 \to \{2\}$
- $1 \to \{1\}$
- și așa mai departe.

2.2 Proprietăți Fundamentale

2.2.1 Autoreferențialitate

Una dintre cele mai fascinante proprietăți ale secvenței Kolakoski este autoreferențialitatea sa: secvența este identică cu propria sa secvență de apariții. Mai exact, dacă definim o secvență auxiliară care reprezintă lungimea blocurilor consecutive, vom observa că aceasta este identică cu secvența Kolakoski originală.

2.2.2 Distribuția Relativ Echilibrată

Deși pare aleatorie, secvența Kolakoski are o distribuție aproape echilibrată a valorilor 1 și 2. Se crede (dar nu s-a demonstrat riguros) că aproximativ 50% dintre elementele secvenței sunt 1 și 50% sunt 2, ceea ce o face similară cu un proces aleatoriu echilibrat.

2.2.3 Lipsa unei Formule Explicite

Nu se cunoaște o formulă directă care să permită calculul termenului fără generarea secvenței până la acea poziție. Această proprietate face ca secvența Kolakoski să fie un exemplu interesant de sistem determinist cu comportament greu de anticipat.

2.2.4 Relații cu Alte Secvențe

Secvența Kolakoski este legată de alte structuri matematice, cum ar fi:

- **Secvențele Sturmiane**, care apar în geometrie computațională și teoria fractalilor.
- **Automatele celulare**, unde reguli simple de generare duc la structuri complexe.
- **Teoria numerelor**, prin distribuția sa echilibrată.

2.3 Observații Computaționale

Studiile pe calculator au arătat că, deși secvența pare să urmeze un tipar, nu există dovezi matematice clare pentru toate proprietățile sale. Generarea unor milioane de termeni sugerează o distribuție uniformă a valorilor 1 și 2, dar natura exactă a comportamentului asimptotic rămâne o problemă deschisă.

3 Generarea Secventei Kolakoski

3.1 Algoritmi iterativi vs. algoritmi recursivi

În informatică, problemele pot fi rezolvate folosind două abordări fundamentale: algoritmi iterativi și algoritmi recursivi. Alegerea între cele două depinde de natura problemei, eficiența execuției și constrângerile impuse de resursele sistemului.

Un algoritm iterativ utilizează structuri repetitive, cum ar fi buclele `for` sau `while`, pentru a repeta un set de instrucțiuni până la atingerea unei condiții de oprire. Pe de altă parte, un algoritm recursiv se bazează pe apeluri succesive ale propriei funcții, reducând problema inițială la subprobleme mai mici până la atingerea unui caz de bază.

Compararea celor două paradigme este esențială pentru înțelegerea compromisurilor dintre claritatea codului, consumul de memorie și complexitatea temporală. În acest capitol, vom analiza avantajele și dezavantajele fiecărei metode și vom explora exemple relevante pentru secvența Kolakoski.

3.2 Variante mai ineficiente in limbaj de nivel inalt

Voi incepe prin a prezenta o varianta de a calcula in mod iterativ primele 100 de numere din secventa Kolakoski in limbajul Python. Ne vom folosi de faptul că putem calcula secventa in urmatorul mod:

1. Pornim de la secventa $\{1, 2, 2\}$
2. Fie $k = 2$
3. Adaugam la final cifra diferita de ultima cifra de atatea ori cat ne arata cifra de pe pozitia k
4. Il incrementam pe k
5. Repetam pasii 3 si 4

```
seq = [1, 2, 2]
pointer = 2
next_value = 2
while len(seq) < 100:
    # Calculam care este urmatoarea valoare de adaugat
    next_value = 3 - next_value

    # Calculam cate elemente de tipul next_value trebuie
    #adaugate
    count = seq[pointer]

    # Adaugam elementele in secventa
    for i in range(count):
        seq.append(next_value)

    # Mutam pointerul la urmatorul element
    pointer = pointer + 1

# Afisam primele 100 de elemente din secventa
print(seq)
```

Acum o varianta recursiva, care functioneaza pe acelasi principiu.

```
def generate_kolakoski(seq, pointer, next_value, target_length):
    if len(seq) > target_length:
        return seq[:target_length]

    # Alternam intre 1 si 2
    next_val = 3 - next_value

    # Numarul de repetari este dat de elementul curent din secventa
    count = seq[pointer]

    # Extindem secventa cu next_val repetat de 'count' ori
    seq.extend([next_val] * count)

    # Recuram cu pointerul incrementat
    return generate_kolakoski(seq, pointer + 1, next_val, target_length)

seq = [1, 2, 2]

# Lungimea secventei
target_length = 100

sequence = generate_kolakoski(seq, 2, 2, target_length)
print(sequence)
```

Deși ambele variante au o complexitate temporală și spațială de *O(n)*, unde n este lungimea secvenței generate, varianta recursivă tinde să fie mai lentă și să ocupe mai mult spațiu în practică, datorită overhead-ului generat de stiva de apeluri. Pentru a evidenția diferența de timp de execuție dintre cele două abordări, am utilizat modulul `time` din Python, implementând următorul fragment de cod

```
import time

t0 = time.process_time()

# {bloc de cod}

t1 = time.process_time()

print("Timp:␣", t1 - t0)
```

Aplicând acest fragment de cod în ambele implementări, am generat secvențe de lungimi considerabile pentru a observa diferențele de performanță. Pentru testare, am ales lungimea de 100.000.000 (o sută de milioane) de elemente. Rezultatele obținute pe sistemul meu (i5-1135G7, 32Gb RAM) sunt:

- Varianta iterativa: 21.506s
- Varianta recursiva: 22.662s

3.3 Variante in limbaje eficiente

Voi prezenta un algoritm similar in limbajul C, limbaj care este mult mai rapid decat Python, mai ales pentru problema noastra.

```
#include <stdio.h>
#include <stdlib.h>
#define NRELEM 100

int main() {
    int *seq;
    if ( (seq = (int *) malloc(NRELEM * sizeof(int))) == NULL) {
        fprintf(stderr, "Eroare:␣Nu␣exista␣suficient␣spatiu\n");
        return 1;
    }
    seq[0] = 1;
    seq[1] = 2;
    seq[2] = 2;
    int pointer = 2, i = 3, elem = 2;
    // i va indica catre ultimul element din secventa + 1
    while (i < NRELEM) {
        // Calculam ce element adaugam
        elem = 3 - elem;

        // Calculam cate elemente adaugam
        int j = seq[pointer];

        // Adaugam elementele
        while (j > 0 && i < NRELEM) {
            seq[i ++] = elem;
            j --;
        }

        // Mutam pointerul
        pointer = pointer + 1;
    }
    // Afisam secventa
    for (i = 0; i < NRELEM; i ++) {
        printf("%d␣", seq[i]);
    }
    return 0;
}
```

Pentru a masura timpul de executie in C putem folosi biblioteca `time.h` astfel:

```
#include <time.h>

int main() {
    clock_t start , end;
    double cpu_time_used;
```

```
    start = clock();

    // {bucata de cod}

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Timp de executie: %f\n", cpu_time_used);
    return 0;
}
```

Dupa mai multe rulari cu acelasi numar ca mai intainte (100.000.000) am obtinut o medie de **0,31** secunde, de aproximativ 70 de ori mai rapid decat varianta iterativa scrisa in Python. In continuare voi prezenta o varianta scrisa in limbaj de asamblare (x86 Assembly, sintaxa AT&T). Codul va fi explicat prin comentarii.

```
.data
    NRMAX: .long 100
    seq: .space 400
    pointer: .long 2
    elem: .long 2

    formatPrintf: .asciz "%d "
.text
.global main
main:
    lea seq, %eax

    movl $1, (%eax)
    movl $2, 4(%eax)
    movl $2, 8(%eax)

    movl $3, %ecx
    generate_seq: ;//while (i < NRMAX)
        cmpl %ecx, NRMAX
        je exit_generate_seq

        ;//elem = 3 - elem
        movl $3, %edx
        subl elem, %edx
        movl %edx, elem

        ;//j = seq[pointer]
        movl pointer, %edx
        movl (%eax, %edx, 4), %ebx

        loop_add_elements: ;// while (j > 0 && i < NRMAX)
            cmpl $0, %ebx
            je exit_loop_add_elements
            cmpl %ecx, NRMAX
```

```
        je exit_loop_add_elements

        movl elem, %esi
        movl %esi, (%eax, %ecx, 4)
        incl %ecx ;//seq[i ++] = elem;
        decl %ebx ;//j --
        jmp loop_add_elements
    exit_loop_add_elements:
    incl pointer ;// pointer ++
    jmp generate_seq
exit_generate_seq:

lea seq, %eax
xorl %ecx, %ecx
print_loop:
    cmpl %ecx, NRMAX
    je exit_print_loop

    pushl %ecx
    pushl %eax
    movl (%eax, %ecx, 4), %ebx
    pushl %ebx
    pushl $formatPrintf
    call printf
    addl $8, %esp
    popl %eax
    popl %ecx

    incl %ecx
    jmp print_loop
exit_print_loop:

pushl $0
call fflush
addl $4, %esp

movl $1, %eax
xorl %ebx, %ebx
int $0x80
```

Pentru a testa timpul de executie pentru 100.000.000 putem folosi comanda `time` din bash. Astfel, obtinem in medie rezultatele:

- real: 0,4
- user: 0,194
- sys: 0,192

Pentru a compara cu timpul dat de `clock` din C, trebuie sa adunam user si sys, adica 0,194 + 0,192 = 0,386. Motivul pentru care timpul este mai mare

in Assembly chiar daca este un limbaj de nivel jos este acela ca compilatorul optimizeaza foarte mult codul scris in C, devenind mai rapid.
In continuare voi prezenta un algoritm pe care l-am scris cu scopul de elimina spatiul de care nu mai avem nevoie la inceputul listei in care tinem minte elementele. Pentru acest scop vom folosi un `deque`, obiect deja implementat in STL in C++.

```
#include <iostream>
#include <deque>

using namespace std;
deque <int> dq;

int main() {
    const int NMAX = 100;
    dq.push_back(1);
    dq.push_back(2);
    dq.push_back(2);
    int elem = 2, current = 3;
    while(current < NMAX) {
        elem = 3 - elem;
        for(int i = 0; i < dq[2] && current < NMAX; i++) {
            dq.push_back(elem);
            current++;
        }
        cout << dq.front() << " ";
        dq.pop_front();
    }
    for (auto i : dq) {
        cout << i << " ";
    }
    return 0;
}
```

Aceasta metoda nu salveaza foarte mult spatiu, dar este o imbunatatire. Vom incerca sa reducem complexitatea spatiala la *O(log(n))*.