Secvența Kolakoski

Daniel Mitrache – Martie 2025

Cuprins

1 Introducere

Secvența Kolakoski este o succesiune autoreferențială formată exclusiv din valorile 1 și 2, având proprietatea distinctivă că fiecare element descrie lungimea unui bloc consecutiv de termeni identici. Definită inițial de William Kolakoski în 1965, această secvență, deși aparent aleatorie, respectă reguli bine determinate. Studiată în contextul matematicii recreative și al teoriei numerelor, secvența Kolakoski ridică întrebări interesante privind structura și distribuția elementelor sale, fiind un subiect de interes în analiza combinatorică și în studiul fractalilor.

Acest document explorează proprietățile secvenței, modalitățile de generare și oferă o comparație între diverse abordări de implementare în funcție de eficiența spațială și temporală.

2 Definiții și Proprietăți

2.1 Definiția Secvenței Kolakoski

Secvența Kolakoski este o succesiune infinită formată din valorile 1 și 2, cu următoarea proprietate fundamentală: fiecare termen indică lungimea blocului consecutiv de numere identice din secvență. De exemplu, secvența începe astfel:

$$1,\ 2,\ 2,\ 1,\ 1,\ 2,\ 1,\ 2,\ 2,\ 1,\ 2,\ 2,\ 1,\ 1,\ 2,\ \ldots$$

Aceasta poate fi explicată astfel:

- Primul element **1** generează blocul $\{1\}$.
- Următorul element **2** generează blocul $\{2, 2\}$.
- Elementul următor, tot **2**, generează blocul $\{1, 1\}$.
- Apoi, un **1** generează blocul $\{2\}$, și așa mai departe.

2.2 Proprietăți Fundamentale

2.2.1 Autoreferențialitate

Una dintre cele mai fascinante proprietăți ale secvenței Kolakoski este *autoreferențialitatea*. Dacă definim o secvență auxiliară care reprezintă lungimile blocurilor consecutive, observăm că această secvență auxiliară este identică cu secvența originală Kolakoski.

2.2.2 Distribuția Relativ Echilibrată

Deși la prima vedere secvența pare aleatorie, se crede (fără o demonstrație matematică riguroasă) că distribuția numerelor 1 și 2 este aproape echilibrată – aproximativ 50% din elemente fiind 1 și 50% 2. Această proprietate îi conferă secvenței un comportament similar unui proces aleator echilibrat.

2.2.3 Lipsa unei Formule Explicite

Un alt aspect interesant este absența unei formule directe care să permită calculul unui termen oarecare fără a genera secvența până la acea poziție. Această caracteristică face ca secvența Kolakoski să fie un exemplu intrigant de sistem determinist, dar cu comportament dificil de anticipat.

2.2.4 Relații cu Alte Secvențe și Structuri Matematice

Secvența Kolakoski este strâns legată de alte structuri matematice:

- **Secvențele Sturmiane** – întâlnite în geometria computațională și în teoria fractalilor.
- **Automatele celulare** – unde reguli simple de generare duc la structuri complexe.
- **Teoria numerelor** – datorită distribuției sale echilibrate.

2.3 Observații Computaționale

Experimentele realizate pe calculator indică faptul că, deși secvența pare să urmeze un tipar, nu există dovezi matematice definitive pentru toate proprietățile sale. Generarea a milioane de termeni sugerează o distribuție uniformă a numerelor 1 și 2, însă natura exactă a comportamentului asimptotic rămâne o problemă deschisă.

3 Generarea Secvenței Kolakoski

Există diverse abordări pentru generarea secvenței Kolakoski, variind în funcție de limbajul de programare și de strategia de implementare.
Observație: De fiecare dată când voi vorbii despre măsurarea timpului de execuție, nu voi lua in considerare operațiile de input/output, ci doar timpul efectiv de calcul al secvenței.

3.1 Algoritmi Iterativi vs. Algoritmi Recursivi

În informatică se pot utiliza două paradigme principale:

- **Algoritmi iterativi** – care folosesc bucle și sunt, de regulă, mai eficienți în gestionarea memoriei.
- **Algoritmi recursivi** – care pot oferi o implementare mai naturală pentru problemele autoreferențiale, dar implică un overhead suplimentar datorită apelurilor recursive.

Ambele abordări, aplicate la secvența Kolakoski, au o complexitate de ordin $O(n)$ atât din punct de vedere temporal, cât și spațial. Totuși, implementarea recursivă tinde să fie ușor mai lentă din cauza suprasarcinii stivei de apeluri.

3.2 Variante în Limbaje de Nivel Înalt (Python)

3.2.1 Varianta Iterativă în Python

Un exemplu de cod iterativ pentru calcularea primelor 100 de elemente:

```python
seq = [1, 2, 2]
pointer = 2
next_value = 2

while len(seq) < 100:
    # Alternam intre 1 si 2
    next_value = 3 - next_value
    # Determinam numarul de elemente de adaugat
    count = seq[pointer]
    # Adaugam elementele in secventa
    for i in range(count):
        seq.append(next_value)
    # Actualizam pointer-ul
    pointer += 1

print(seq)
```

Listing 1: Algoritm Iterativ în Python

3.2.2 Varianta Recursivă în Python

Și o abordare recursivă, bazată pe același principiu:

```python
def generate_kolakoski(seq, pointer, next_value, target_length):
    if len(seq) >= target_length:
        return seq[:target_length]
    next_val = 3 - next_value
    count = seq[pointer]
    seq.extend([next_val] * count)
    return generate_kolakoski(seq, pointer + 1, next_val, target_length)

seq = [1, 2, 2]
target_length = 100
sequence = generate_kolakoski(seq, 2, 2, target_length)
print(sequence)
```

Listing 2: Algoritm Recursiv în Python

Testele pe secvențe lungi (de exemplu, pentru 100.000.000 de elemente) au arătat că varianta iterativă se execută în aproximativ 21,5 secunde, comparativ cu circa 22,7 secunde pentru varianta recursivă.

3.3 Implementări în Limbaje Eficiente

3.3.1 Implementarea în C

În limbajul C, metoda de generare este similară, dar mult mai eficientă:

```c
#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;
    while (i < NRELEM) {
        // Calculam elementul de adaugat
        elem = 3 - elem;
        int j = seq[pointer];
        while (j > 0 && i < NRELEM) {
            seq[i++] = elem;
            j--;
        }
        pointer++;
    }

    for (i = 0; i < NRELEM; i++) {
        printf("%d ", seq[i]);
    }

    free(seq);
    return 0;
}
```

Listing 3: Algoritm în C

Pentru măsurarea timpului de execuție se poate utiliza biblioteca `time.h`:

```c
#include <time.h>
...
clock_t start = clock();
// Codul de generare a secventei
clock_t end = clock();
double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Timp de executie: %f\n", cpu_time_used);
```

Listing 4: Măsurarea timpului de execuție în C

Testele efectuate pentru generarea a 100.000.000 de elemente au evidențiat o medie de execuție de aproximativ 0,31 secunde – de circa 70 de ori mai rapidă decât varianta Python.

3.3.2 Implementarea în Assembly (x86, sintaxa AT&T)

În continuare voi prezenta o variantă scrisă 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. Astflel, obtinem in medie rezultatele:

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

Pentru a face o comparație relevantă cu timpul raportat de funcția `clock` din C, este necesar să combinăm timpii user și sys, obținând 0,386 secunde. Deși Assembly este un limbaj low-level, care teoretic ar trebui să fie mai rapid, compilatoarele C moderne efectuează optimizări atât de eficiente încât codul generat depășește adesea performanța codului Assembly scris manual.

3.3.3 Utilizarea unui Deque în C++

În continuare voi prezenta un algoritm pe care l-am scris cu scopul de elimina spațiul de care nu mai avem nevoie la începutul listei în care ținem minte elementele. Pentru acest scop vom folosi un `deque`, obiect deja implementat in STL in C++.

```
#include <iostream>
#include <deque>
using namespace std;

int main() {
    const int NMAX = 100;
    deque<int> dq;

    dq.push_back(1);
    dq.push_back(2);
    dq.push_back(2);

    int elem = 2, current = 3;

    while (current < NMAX) {
        elem = 3 - elem;
        // Adaugam elemente in functie de valoarea din pozitia a treia a deque-ului
        for (int i = 0; i < dq[2] && current < NMAX; i++) {
            dq.push_back(elem);
            current++;
        }
        // Eliminam elementul de la inceputul deque-ului
        cout << dq.front() << " ";
        dq.pop_front();
    }

    // Afisam restul elementelor din deque
    for (auto i : dq) {
        cout << i << " ";
    }

    return 0;
}
```

Listing 5: Algoritm cu Deque în C++

Această metodă nu salvează foarte mult spațiu, dar este cea mai naturală îmbunătățire. Vom încerca să reducem complexitatea spatială la *O(log(n))*.

4 Algoritmi cu Complexitate Spațială $O(\log n)$

În 2012, Johan Nilsson a propus o metodă inovatoare de generare a secvenței Kolakoski, bazată pe ideea că cifra necesară pentru a genera poziția k este, la rândul ei, derivată recursiv. Această abordare utilizează conceptul de „niveluri” (L_n), fiecare nivel conținând informația necesară pentru generarea secvenței fără a o stoca integral în memorie.

4.1 Algoritmul lui Nilsson

Inițializare

- Se definește nivelul inițial $L_0 = 22$.

Creșterea nivelului L_n

- Dacă L_{n+1} nu există, se setează $L_{n+1} = 22$.
- Dacă L_{n+1} conține două simboluri, eliminăm primul simbol din L_{n+1}, iar în L_n înlocuim simbolul curent cu numărul de simboluri specificat de primul element rămas din L_{n+1}.
- Simbolul scris în L_n este opus simbolului anterior din L_n.
- Dacă L_{n+1} conține un singur simbol, creștem L_{n+1} recursiv.
- După această creștere, noul simbol scris în L_n are o lungime egală cu prima valoare din L_{n+1} și este opus simbolului anterior din L_n.
- Spre deosebire de cazul cu două simboluri, aici nu eliminăm primul simbol din L_{n+1} când ne întoarcem din recursivitate.

Generarea secvenței K'

- Se execută în mod repetat creșterea nivelului L_0.
- Se menține o evidență a numărului de simboluri '1' și '2' întâlnite.
- Fiecare nivel parcurge întreaga secvență K', garantând astfel generarea corectă.

Explicație suplimentară

Algoritmul funcționează printr-o metodă de **auto-descriere recursivă**, unde fiecare nivel al structurii depinde de nivelurile inferioare. În loc să construim secvența K în memorie, folosim această abordare eficientă, care necesită doar $O(\log n)$ spațiu, ceea ce o face mult mai rapidă și scalabilă pentru valori mari de n.

4.2 Exemple de Algoritmi în C

4.2.1 Algoritmul Inspirat de Nilsson

```c
#include <stdio.h>
#define NMAX 100

enum kval { K22 = 0, K11 = 1, K2 = 2, K1 = 3 };

static void next(enum kval *v) {
```

```c
    if (*v == K22 || *v == K11) {
        *v += 2;
    } else {
        next(v + 1);
        *v = !(*v % 2) + 2 * (v[1] % 2);
    }
}

int main(void) {
    enum kval stack[256] = {0};
    printf("1 2 ");
    for (unsigned long long i = 2; i < NMAX; i++) {
        next(stack);
        printf("%d ", 2 - *stack % 2);
    }
    return 0;
}
```

Listing 6: Algoritm inspirat de Nilsson

4.2.2 Algoritm Iterativ folosind Operații pe Biți

```c
#include <stdio.h>
#define NMAX 100

int Kolakoski() {
    int x = -1, y = -1, k[2] = {2, 1}, f;
    for (unsigned long long i = 2; i < NMAX; i++) {
        printf("%d ", k[x & 1]);
        f = y & ~(y + 1);
        x ^= f;
        y = (y + 1) | (f & (x >> 1));
    }
    return 0;
}

int main() {
    Kolakoski();
    return 0;
}
```

Listing 7: Algoritm iterativ folosind operații pe biți

Aceste implementări demonstrează cum se poate genera secvența Kolakoski eficient, reducând complexitatea spațială și fiind potrivite pentru calcule la scară largă.

4.3 Calculul unui număr cât mai mare de termeni

După implementarea optimizărilor discutate anterior, vom încerca să generăm un număr cât mai mare de termeni ai secvenței Kolakoski, încercând să ne apropiem sau chiar să depășim numărul maxim documentat până în prezent. Conform articolului Wikipedia `https://en.wikipedia.org/wiki/Kolakoski_sequence`, în secțiunea „Research”, au fost generați până la 10^{13} termeni.

Am rulat programul pentru o perioadă îndelungată (peste două ore) și am obținut următoarele rezultate privind distribuția numerelor 1 și 2 în secvență:

Numărul de cifre	**Numărul de 1**	**Numărul de 2**
10^1	5	5
10^2	49	51
10^3	502	498
10^4	4996	5004
10^5	49972	50028
10^6	499986	500014
10^7	5000046	4999954
10^8	50000675	49999325
10^9	500001223	499998777
10^{10}	4999997671	5000002329
10^{11}	50000001587	49999998413
10^{12}	500000050701	499999949299
10^{13}	5000000008159	4999999991841
$2 * 10^{13}$	10000000073089	9999999926911
10^{14}	50000000316237	49999999683763

5 Bibliografie

- **chatgpt**
- **u/skeeto** pe Reddit
- **Arxiv:** `https://arxiv.org/abs/1110.4228`
- `https://11011110.github.io/blog/2016/10/14/kolakoski-sequence-via.html`
- `https://11011110.github.io/blog/2016/10/13/kolakoski-tree.html`
- `https://np.reddit.com/r/dailyprogrammer/comments/8df7sm/20180419_challenge_357_intermediate_kolakoski/`