**Generowanie podzbiorów z wykorzystaniem kodu Gray’a**

Aby w bardzo prosty sposób wygenerować wszystkie podzbiory pewnego zbioru U, można wykorzystać najważniejszą właściwość kodu Gray’a, a mianowicie, że kolejne jego wyrazy różnią się tylko na jednej pozycji. Przyjmujemy, że 1 na danej w słowie oznacza występowanie elementu w wynikowym zbiorze, natomiast 0 brak.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int\* tab; // globalna dynamiczna tablica z elementami (liczby naturalne)

unsigned int gray(unsigned int num) { // zamiana liczby z NKB na kod Graya

return (num >> 1) ^ num;

}

char litera(int n) { // wypisywanie liter zamiast liczb

return n + 'A'; // 0-A 1-B ... , nie działa ponad Z, ale wystarczy

}

void print (unsigned int n) {

int it = 0;

if (n == 0) printf("(#)"); // pusty zbiór to 0 (nie pusta linia) szczególny przypadek

else while (n) { // standardowe przesuwanie bitowe i testowanie najmłodszego bitu

if ((n & 0x1) == 1) printf("%c", litera(tab[it]));

n >>= 1;

it++;

}

puts("");

}

int main(int argc, char\*\* argv) {

if (argc != 2) { // Argumenty wywołania

printf("Użycie: %s <ilość elementów>\n", argv[0]);

return 1;

}

int i = atoi(argv[1]); // tego ponoć nie ma pod Windowsem, pod linuksowym libc jest

tab = malloc(i\*sizeof(unsigned int));

long int t;

for (t = 0; t < i; ++t) tab[t] = t; // wartość elementu to indeks

for (t = 0; t < (long int)pow(2, i); ++t) print(gray(t));

free(tab);

return 0;

}

Przykładowe wyjście programu dla n = 5

(#)

A

AB

B

BC

ABC

AC

C

CD

ACD

ABCD

BCD

BD

ABD

AD

D

DE

ADE

ABDE

BDE

BCDE

ABCDE

ACDE

CDE

CE

ACE

ABCE

BCE

BE

ABE

AE

E

Przeprowadziliśmy badanie czasu wykonywania programu, bez wypisywania zbiorów na ekran (duże obciążenie), do tego zadania wykorzystaliśmy polecenie time powłoki bash.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **n [-]** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| **t [s]** | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,001 | 0,002 | 0,004 |
| **n [-]** | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |  |
| **t [s]** | 0,008 | 0,017 | 0,035 | 0,069 | 0,142 | 0,295 | 0,604 | 1,232 | 2,533 | 5,141 | 10,668 | 23,676 | 45,542 |  |

I sporządziliśmy wykres:

Jak widać, złożoność algorytmu jest wykładnicza. Zgadza się to z teorią, ponieważ dla każdego n-elementowego zbioru istnieje 2^n jego podzbiorów, z których każdy wymaga w przybliżeniu stałej ilości czasu do generacji.