Algorytm generujący podziały zbioru

Sprawozdanie – Łukasz Konieczny | LK4 | LAB2

1. Wstęp teoretyczny

Podczas zajęć poznaliśmy algorytmy generujące podziały zbioru na „k” bloków.

„K” -blokowy podział zbioru (inaczej nazywany partycją), to dowolne ułożenie elementów zbioru w „k” blokach (czyli podzbiorach zbioru).

Na przykład, jeśli chcemy podzielić zbiór {1, 2, 3, 4, 5} na 3 bloki, możemy to zrobić w następujący sposób:

{1, 4}, {2, 3}, {5} – gdzie każdy z tych podzbiorów to jeden blok.

W podziałach nie ma znaczenia ilość elementów w jednym bloku, dopóki każdy z „k” bloków zawiera przynajmniej jeden.

Podział {1, 4}, {2, 3, 5}, {} ma tylko 2 bloki, więc jest to NIEPRAWIDŁOWY 3-blokowy podział, ale już prawidłowy 2-blokowy.

Do określenia na ile sposobów możemy podzielić zbiór służy liczba Bella, dana wzorem:

$$B_n = \Sigma_{k=0}^{n} \{\frac{n}{k}\}$$

Gdzie $\{\frac{n}{k}\}$ to liczba Stirlinga II rodzaju, która definiuje liczbę „k”-blokowych podziałów „n” elementowego zbioru

Algorytmy które implementowaliśmy, tworzą podziały w rosnącym porządku leksykograficznym. Oznacza to, że liczby (do których przejdę w trakcie omawiania algorytmów) uporządkowane są w następujący sposób:

Dla liczb

1321

2468

1234
Patrzymy na pierwsze cyfry: drugą liczbę od razu uznajemy za największą, trafi na koniec wyniku. Liczby pierwsza i trzecia obie zaczynają się od 1, więc musimy porównać ich kolejne cyfry.

Patrzymy na cyfry nr. 2. Widzimy, że cyfra 3 jest większa od 2, więc ostateczne ułożenie liczb to:
1234
1321
2468

Jak więc widać, porządek leksykograficzny polega na ustawieniu elementów zbioru, w tym wypadku liczb, zależnie od tego, w jakiej relacji są ich kolejne elementy. Ponieważ chcemy ustawić liczby rosnąco, sprawdzamy, która z cyfr na danym indeksie w liczbie jest mniejsza i zamieniamy je (liczby) ze sobą zgodnie z wynikiem.

2. Omówienie algorytmów

Pierwszym algorytmem, który mieliśmy zaimplementować, jest algorytm który generuje WSZYSTKIE możliwe podziały zbioru „n”-elementowego. Jako parametr przyjmował on liczbę „n”, określającą wielkość zbioru. Oto jego zapis w pseudokodzie:

inicjalizacja:
V0 For [i] From [0] To [n], execute the following steps:
  V0.1 Set [$a_i \leftarrow 1$].
  V0.2 Set [$b_i \leftarrow 1$].
powtarzaj:
  V1 Set [$c \leftarrow n$].
  V2 While [$a_c = n$ Or $a_c > b_c$] Do [$c \leftarrow c - 1$].
  V3 If [c = 1] Report the end of enumeration.
  V4 Set [$a_c \leftarrow a_c + 1$].
  V5 For [i] From [c + 1] To [n], execute the following steps:
    V5.1 Set [$a_i \leftarrow 1$].
    V5.2 Set [$b_i \leftarrow \max(a_{i-1}, b_{i-1})$].
  V6 wyprowadź a

UWAGA: Na zdjęciu widzimy zmodyfikowany algorytm, który numeruje grupy od 1, zamiast od 0! Moja implementacja również zawiera tą zmianę.

Przykładowym wynikiem działania algorytmu dla „n” = 4, jest:

1 1 1 1
1 1 1 2
1 1 2 1
1 1 2 2
1 1 2 3
1 2 1 1
1 2 1 2
1 2 1 3
1 2 2 1
1 2 2 2
1 2 2 3
1 2 3 1
1 2 3 2
1 2 3 3
1 2 3 4

Jednak co oznaczają te liczby? Każda cyfra w wyniku, odpowiada numerowi bloku, do którego element znajdujący się na indeksie odpowiadającym indeksowi tej cyfry, został przyporządkowany.

Pokażmy to na przykładzie:

Dla zbioru { 1, 2, 3, 4, 5, 6 }, 3 blokowy podział, reprezentowany ciągiem
1 2 2 3 1 2

Oznacza następujące ułożenie elementów w blokach:

{ 1, 5 }, { 2, 3, 6 }, { 4 }

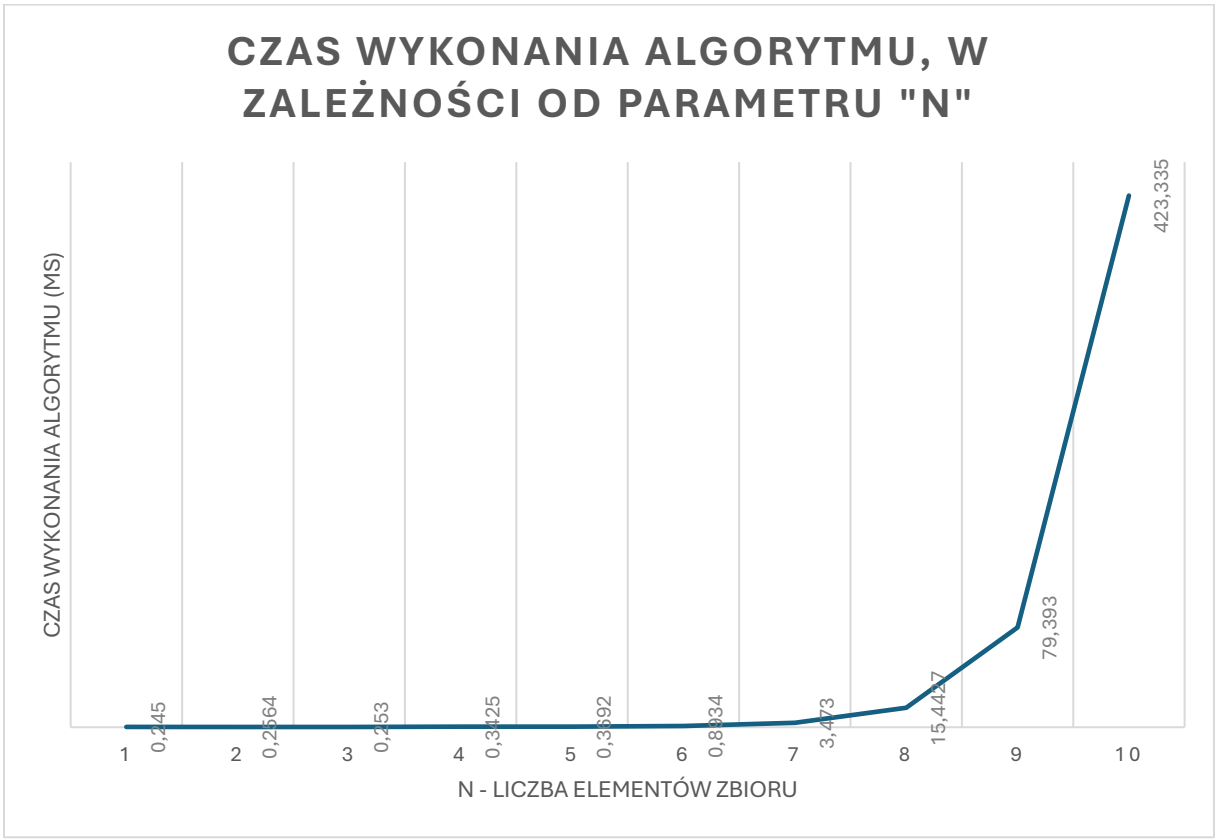
Gr 1. Gr.2 Gr.3

Algorytm, według opisu w dokumencie „Lexicographic Enumeration of Set Partitions", charakteryzuje się złożonością liniową O(n)

Poniżej przedstawiam wykres, obrazujący czas wywołania dla rosnących wartości parametru „n":

(Co ciekawe, dla „n" = 1 i prośby o wygenerowanie wszystkich możliwych podziałów, program wpadał w nieskończoną pętlę. Udało mi się to naprawić, zmieniając warunek V3 na

„if (c <= n)"...)



Zdjęcie uruchomionego algorytmu na moim komputerze:

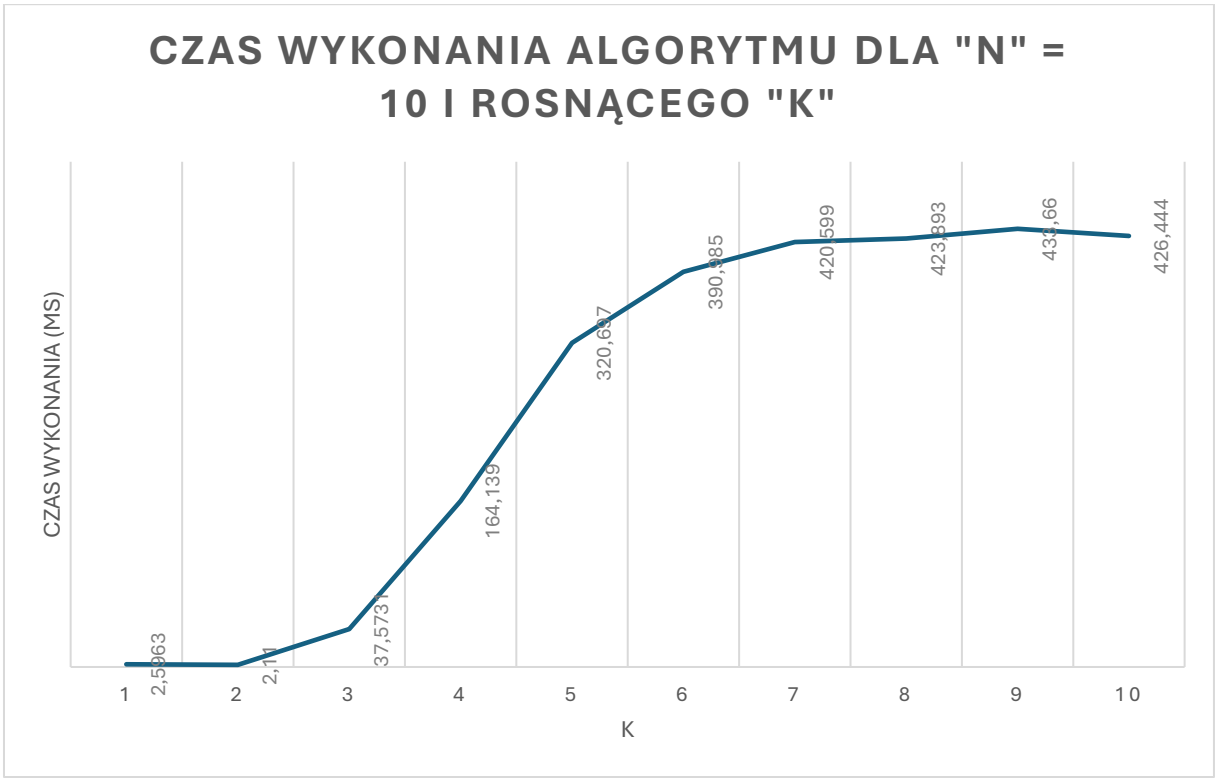
```
Łukasz Konieczny LK4
LAB2 - generowanie podzialow zbioru "n" elementowego
-----------------------------------------
Podaj n: 3
Czy chcesz wygenerowac WSZYSTKIE podzialy? Jesli tak, wpisz 1, jesli nie, wpisz 0: 1
Wynik zapisano do pliku wynik.txt
Czas wykonania: 0.2567 ms
```

wynik.txt

1	1	1	1
2	1	1	2
3	1	2	1
4	1	2	2
5	1	2	3
6			

Kolejnym algorytmem do implementacji, był wariant algorytmu pierwszego, który generował ***co najmniej k-blokowe podziały***.
Aby osiągnąć to działanie, wystarczyło zmienić w warunku V2 porównanie z
„a[c] == n” na „a[n] == k”
Zmiana ta gwarantuje, że największą możliwą ilością utworzonych bloków będzie „k”, jednocześnie zachowując porządek leksykograficzny.

Podobnie jak algorytm 1, cechuje go złożoność liniowa w zależności od „n”, jednak im większe „k” dla danego „n” podamy, tym jego szybkość się zwiększy. Gdy podamy „k” równe „n”, złożoność algorytmu będzie identyczna jak algorytmu pierwszego.
Poniżej przedstawiam wykres obrazujący czas wywołania dla stałego n = „10” i rosnących wartości „k”:



Jak widzimy, im bliżej naszemu „k” do „n”, tym czas wykonania algorytmu względem poprzedniej wartości „k” różni się coraz mniej, można więc powiedzieć, że algorytm „przyspiesza” dla k bliższego „n”.

Zdjęcie uruchomionego na moim komputerze algorytmu:

```
Łukasz Konieczny LK4
LAB2 - generowanie podzialow zbioru "n" elementowego
-------------------------------------------
Podaj n: 5
Czy chcesz wygenerowac WSZYSTKIE podzialy? Jesli tak, wpisz 1, jesli nie, wpisz 0: 0
Podaj k: 3
Czy chcesz wygenerowac dokladnie k-blokowe podzialy? Jesli tak, wpisz 1, jesli nie, wpisz 0: 0
Wynik zapisano do pliku wynik.txt
Czas wykonania: 0.3333 ms
```

wynik.txt

```
 1  1 1 1 1 1
 2  1 1 1 1 2
 3  1 1 1 2 1
 4  1 1 1 2 2
 5  1 1 1 2 3
 6  1 1 2 1 1
 7  1 1 2 1 2
 8  1 1 2 1 3
 9  1 1 2 2 1
10  1 1 2 2 2
11  1 1 2 2 3
12  1 1 2 3 1
13  1 1 2 3 2
14  1 1 2 3 3
15  1 2 1 1 1
16  1 2 1 1 2
17  1 2 1 1 3
18  1 2 1 2 1
19  1 2 1 2 2
20  1 2 1 2 3
21  1 2 1 3 1
22  1 2 1 3 2
23  1 2 1 3 3
24  1 2 2 1 1
25  1 2 2 1 2
26  1 2 2 1 3
27  1 2 2 2 1
28  1 2 2 2 2
29  1 2 2 2 3
30  1 2 2 3 1
31  1 2 2 3 2
32  1 2 2 3 3
33  1 2 3 1 1
34  1 2 3 1 2
35  1 2 3 1 3
36  1 2 3 2 1
37  1 2 3 2 2
38  1 2 3 2 3
39  1 2 3 3 1
40  1 2 3 3 2
41  1 2 3 3 3
```

Ostatni wariant algorytmu miał za zadanie generować ***dokładnie k-blokowe podziały***.

Jego zapis w pseudokodzie jest następujący:

Y1 Execute Algorithm W.

Y2 If $[\max(a_n, b_n) \neq k\ ]$:

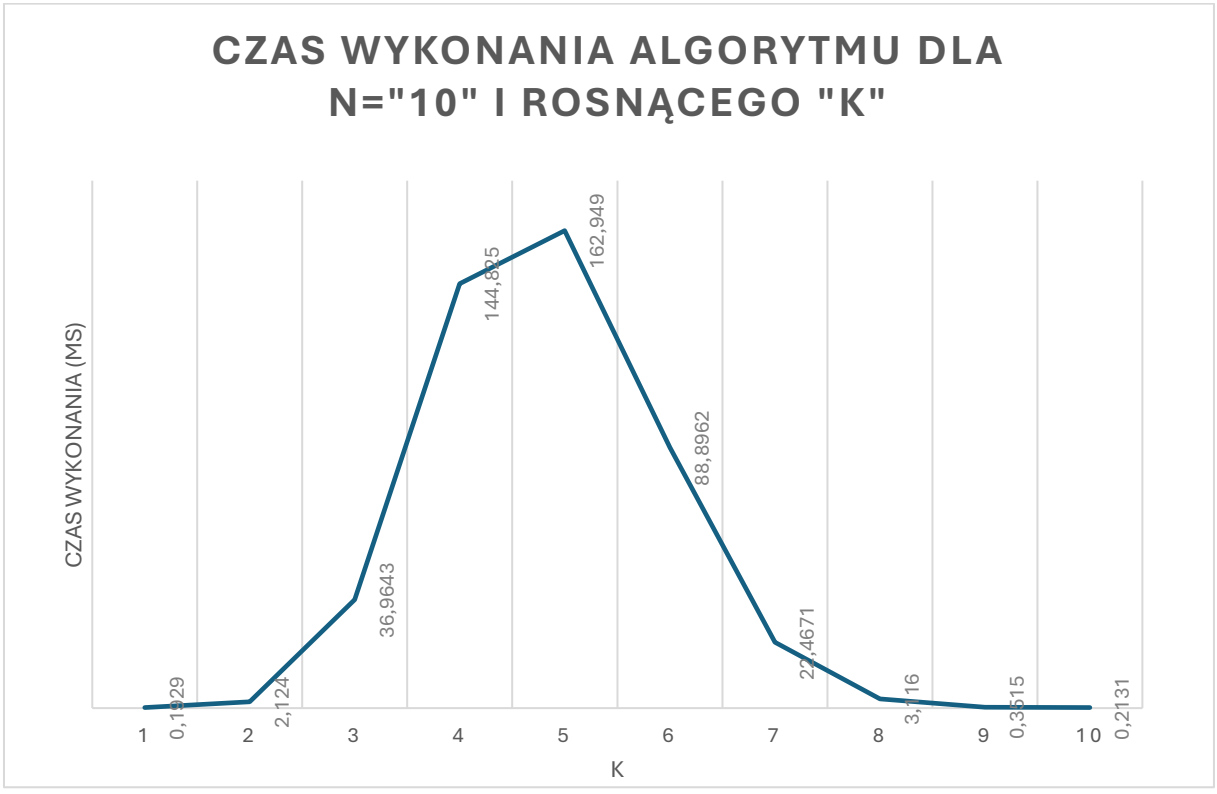
Y2.1 For $[(i, k_0)]$ From $[(n, k)]$ Down To $[(1, 1)]$ As Long As $[k_0 > b_i]$.

Y2.1.1 Set $[a_i \leftarrow k_0]$.

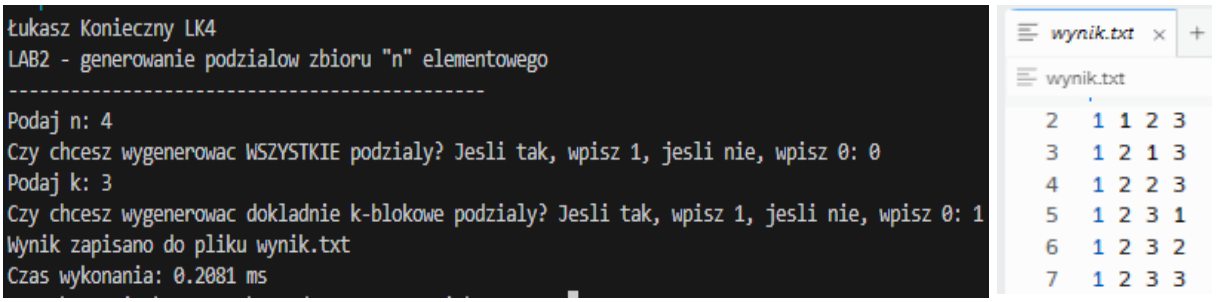
Y2.1.2 Set $[b_i \leftarrow k_0 - 1]$.

Jest to ponownie lekko zmodyfikowana wersja oryginalnego algorytmu, która bierze poprawkę na numerowanie zbioru od 1 a nie od 0.
Algorytm W, to w naszym wypadku drugi algorytm, który generuje ***co najwyżej k-blokowe podziały***.

Algorytm posiada podobną złożoność jak algorytm 2, ponieważ operacje w dodanym przez niego warunku wywołają się tylko wtedy, gdy podział wygenerowany przez algorytm 2 nie składa się z „k” bloków.



Zdjęcie uruchomionego na moim komputerze algorytmu:



3. Szczegóły implementacji

Algorytm można w łatwy sposób zaimplementować w języku C++ za pomocą jednej funkcji, przyjmującej trzy argumenty:

Liczbę n, liczbę k, oraz wartość bool określającą, czy użytkownik chce wygenerować dokładnie „k” elementowe podziały.

Można stworzyć dodatkową funkcję pomocniczą przyjmującą tylko liczbę n, i wywołującą główną funkcję z argumentami (n, n, false), aby łatwiej uruchamiać algorytm pierwszy.

Implementujemy od razu algorytm drugi, jeśli będziemy chcieli generować wszystkie podziały to za liczbę k podamy liczbę n i uzyskamy identyczny rezultat jak przy uruchomieniu algorytmu pierwszego, jednocześnie zwiększając funkcjonalność naszej funkcji.

Aby funkcja obsługiwała także wariant trzeci, tuż po zakończeniu wykonywania instrukcji z algorytmu drugiego (ale przed zamknięciem pętli algorytmu!) dodajemy instrukcję warunkową z algorytmu trzeciego, która wykona się tylko wtedy, gdy oprócz spełnienia jej warunków, zmienna bool odpowiadająca za generację DOKŁADNIE k-blokowych podziałów będzie ustawiona na „true”.

Dzięki takiemu zabiegowi, otrzymujemy jedną funkcję wypełniającą zadania wszystkich trzech warunków algorytmu, unikając niepotrzebnego powtarzania tego samego kodu (gdybyśmy każdy algorytm implementowali jako osobną funkcję).

4. Podsumowanie

Podczas zajęć poznaliśmy i zaimplementowaliśmy algorytmy, które pozwalają podzielić zbiór dowolnej wielkości na ustaloną przez nas ilość bloków, maksymalnie, lub dokładnie. Algorytmy te cechuje złożoność liniowa, co sprawia że są bardzo wydajne w wykonywaniu zadanych im operacji. Jednocześnie zauważyliśmy niewielkie różnice pomiędzy nimi, co pozwoliło nam zaimplementować tylko jedną funkcję, działającą dla wszystkich trzech wariantów zadania.

5. Bibliografia

- Materiały z wykładów nt. obiektów kombinatorycznych
- W. Lipski, "Kombinatoryka dla programistów", WNT, Warszawa, 1989
- Giorgos Stamatelatos, Pavlos S. Efraimidis, Lexicographic Enumeration of Set Partitions, arXiv - CS - Discrete Mathematics, 2021