УДК 519.1:004.42

DOI 10.31651/2076-5886-2024-1-34-45

PACS 02.10.Ox

КУРИЛО Олександр Миколайович
директор, учитель інформатики Золотоніської загальноосвітньої школи I–III ступенів № 5

НІЦАК Владислав Костянтинович
учень Золотоніської загальноосвітньої школи I–III ступенів № 5

ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ ГЕНЕРАЦІЇ КОМБІНАТОРНИХ ОБ'ЄКТІВ

У статті розглянуто основні комбінаторні об'єкти – перестановки, розміщення та комбінації, а також їхні варіанти з повторенням. Наведено математичне обґрунтування відповідних формул підрахунку кількості об'єктів кожного типу. Для кожного з розглянутих видів комбінаторних об'єктів описано та реалізовано алгоритми генерації усіх можливих варіантів. Реалізацію виконано мовою програмування Python з використанням рекурсивного та ітераційного підходів. Особливу увагу приділено порівнянню обох підходів з огляду на їхню ефективність та зручність реалізації. Розроблено програмний продукт із графічним інтерфейсом, що дозволяє наочно демонструвати результати роботи алгоритмів і може застосовуватись як навчальний засіб у курсах математики та інформатики.

Ключові слова: *комбінаторика, перестановки, розміщення, комбінації, рекурсивний алгоритм, Python, генератор.*

Вступ

Комбінаторика є одним із фундаментальних розділів дискретної математики, що займається підрахунком кількості об'єктів, які задовольняють певному набору властивостей, а також побудовою цих об'єктів. Термін «комбінаторика» з'явився ще у 1666 році в роботі Лейбніца, однак у сучасному значенні цей розділ математики інтенсивно розвинувся лише у XX столітті у зв'язку з потребами криптографії, теорії кодування та теорії алгоритмів [4, 6].

Комбінаторні об'єкти – перестановки, розміщення та комбінації – є базовими поняттями, що широко застосовуються у теорії ймовірностей, математичній статистиці, оптимізації та криптографії. Водночас задачі генерації (тобто побудови і виведення) усіх можливих комбінаторних конфігурацій належать до класу NP-задач, де кількість об'єктів зростає надзвичайно швидко зі збільшенням вхідних параметрів. Дослідження ефективних алгоритмів генерації таких об'єктів залишається актуальною задачею, особливо з огляду на потреби навчання та ілюстрації математичних понять.

Незважаючи на те що теоретичний апарат комбінаторики є добре розробленим, практична реалізація відповідних алгоритмів з акцентом на порівнянні рекурсивного та ітераційного підходів, а також на застосуванні сучасних засобів мови програмування Python (зокрема, генераторів та ключового слова yield), залишається методично цінною темою для досліджень прикладного рівня.

Метою статті є систематизація математичних основ основних комбінаторних об'єктів, опис та порівняння алгоритмів їх генерації, реалізованих мовою Python з використанням рекурсивного та ітераційного підходів, а також розробка програмного продукту з графічним інтерфейсом для наочної демонстрації результатів.

Виклад основного матеріалу

1. Огляд задач перебору та класи складності P і NP

Задачі в програмуванні поділяють на класи залежно від часової складності їхнього розв'язання. Задачі класу **P** розв'язуються за поліноміальний час відносно розміру вхідних даних – тобто існує алгоритм зі складністю O(n^k) для деякого фіксованого k. До таких задач належать, зокрема, множення матриць зі складністю $O(n^3)$, сортування масивів та знаходження ейлерового циклу у графі [1, 7].

Задачі класу **NP** (недетерміновано поліноміальні) – це такі, розв'язок яких можна перевірити за поліноміальний час, проте ефективного алгоритму побудови розв'язку для них не знайдено. Прикладами є задача комівояжера та задача визначення існування цілочисельного розв'язку системи лінійних нерівностей [5]. Задачі генерації комбінаторних об'єктів фактично є NP-задачами: кількість об'єктів зростає факторіально або степенево, що робить повний перебір прийнятним лише для невеликих вхідних розмірів.

Важливим відкритим питанням теоретичної інформатики залишається рівність або нерівність класів P і NP. У разі P = NP для всіх NP-задач існував би ефективний алгоритм класу P, що докорінно змінило б підходи до розв'язання задач оптимізації та криптографії. Сьогодні переважає думка, що P ≠ NP, проте формального доведення досі немає [1].

2. Комбінаторні об'єкти: математичні основи

2.1 Правила суми та добутку

В основі комбінаторики лежать два фундаментальні принципи.

Правило суми: якщо об'єкт типу A_1 можна обрати n_1 способами, а об'єкт типу A_2 – n_2 іншими способами (і ці вибори взаємно виключають один одного), то вибір одного з них здійснюється $n_1 + n_2$ способами [6, 8].

Правило добутку: якщо об'єкт A_1 обирається n_1 способами, і після кожного такого вибору об'єкт A_2 може бути обраний n_2 способами, то послідовний вибір обох об'єктів здійснюється $n_1 \cdot n_2$ способами [6, 8].

Ці два правила є підґрунтям для виведення формул підрахунку перестановок, розміщень і комбінацій.

2.2 Перестановки без повторення

Перестановкою з n елементів називається будь-яке впорядковане розташування усіх n елементів. Кількість перестановок розраховується за формулою:

$$P_n = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1 = n!. \qquad (1)$$

Приклад: кількість способів впорядкувати три літери *A*, *B*, *C* дорівнює 3! = 6 (рис. 1):

ABC, *ACB*, *BAC*, *BCA*, *CAB*, *CBA*

2.3 Розміщення без повторення

Розміщенням з n елементів по m називається будь-яка впорядкована підмножина з m елементів n-елементної множини, де $m \le n$. Кількість розміщень:

$$A_n^m = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-m+1) = \frac{n!}{(n-m)!}. \qquad (2)$$

Відмінність від перестановок полягає в тому, що кількість позицій для розміщення менша за загальну кількість елементів (рис. 2).

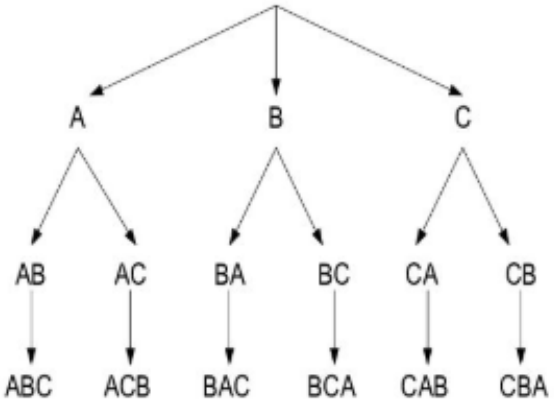

Рис.1. Утворення усіх перестановок 3-х літер: *A*, *B*, *C*

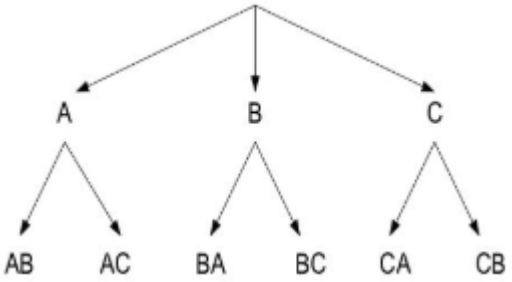

Рис. 2. Утворення усіх розміщень з 3-х літер, *A*, *B*, *C*, по 2

2.4 Комбінації без повторення

Комбінацією з n елементів по m називається підмножина з m елементів, де порядок розташування елементів не важливий. Кількість комбінацій:

$$C_n^m = \frac{A_n^m}{P_m} = \frac{n!}{m!(n-m)!}. \tag{3}$$

Зазначимо, що C_n^m збігається з біноміальним коефіцієнтом у розкладі біному Ньютона:

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}. \tag{4}$$

Для обчислення біноміальних коефіцієнтів зручно використовувати трикутник Паскаля (рис. 3), де кожний елемент є сумою двох сусідніх елементів рядка вище:

$$C_n^m = C_{n-1}^{m-1} + C_{n-1}^m. \tag{5}$$

2.5 Перестановки з повторенням

Якщо серед n елементів є групи однакових, де кожен елемент i-го типу повторюється k_1 разів ($k_1 + k_2 + ... + k_l = n$), кількість різних перестановок:

$$\overline{P}_n(k_1,\ldots,k_l)=\frac{n!}{k_1!\cdot\ldots\cdot k_l!}. \qquad (6)$$

$$\begin{array}{ccccccccccc}
 & & & & C_0^0 & & & & & & \\
 & & & & C_1^0 & & C_1^1 & & & & \\
 & & & C_2^0 & & C_2^1 & & C_2^2 & & & \\
 & & C_3^0 & & C_3^1 & & C_3^2 & & C_3^3 & & \\
 & C_4^0 & & C_4^1 & & C_4^2 & & C_4^3 & & C_4^4 & \\
. & & . & & . & & . & & . & & . \\
C_n^0 & & C_n^1 & & C_n^2 & \ldots & C_n^{n-2} & & C_n^{n-1} & & C_n^n
\end{array}$$

Рис. 3. Обчислення біноміальних коефіцієнтів за допомогою трикутника Паскаля

Приклад: кількість різних слів зі слова «математика» (10 літер: «м» – 2, «а» – 3, «т» – 2, «е» – 1, «и» – 1, «к» – 1) дорівнює $\frac{10!}{2!\cdot 3!\cdot 2!\cdot 1!\cdot 1!\cdot 1!}$ = 151 200.

2.6 Розміщення з повторенням

Якщо на кожну з k позицій незалежно може бути поставлений будь-який із n елементів (елементи повторюватись можуть), загальна кількість варіантів:

$$\overline{A}_n^k = n^k. \qquad (7)$$

Приклад: кількість кодів чотиризначного цифрового замка – 10^4 = 10000.

2.7 Комбінації з повторенням

Якщо з n типів елементів обирається k елементів із можливим повторенням і порядок не важливий, кількість таких комбінацій:

$$\overline{C}_n^k = C_{k+n-1}^k = \frac{(k+n-1)!}{(n-1)!\cdot k!}. \qquad (8)$$

Для інтуїтивного розуміння цієї формули використовують графічний метод «шарів і перегородок» (рис. 4-5): будь-яке розміщення k однакових шарів у n ящиках однозначно кодується послідовністю з k нулів та (n–1) одиниць – перегородок між ящиками [8].

Рис. 4. Початкові дані до прикладу

Рис. 5. Перегородки задають три ящики, що містять відповідно 4, 1 і 2 об’єкти

2.8 Узагальнена схема методів

Для зручності вибору формули у конкретній задачі можна використати схему-класифікатор, де основними критеріями є: (1) чи важливий порядок елементів у наборі, та (2) чи допускається повторення елементів.

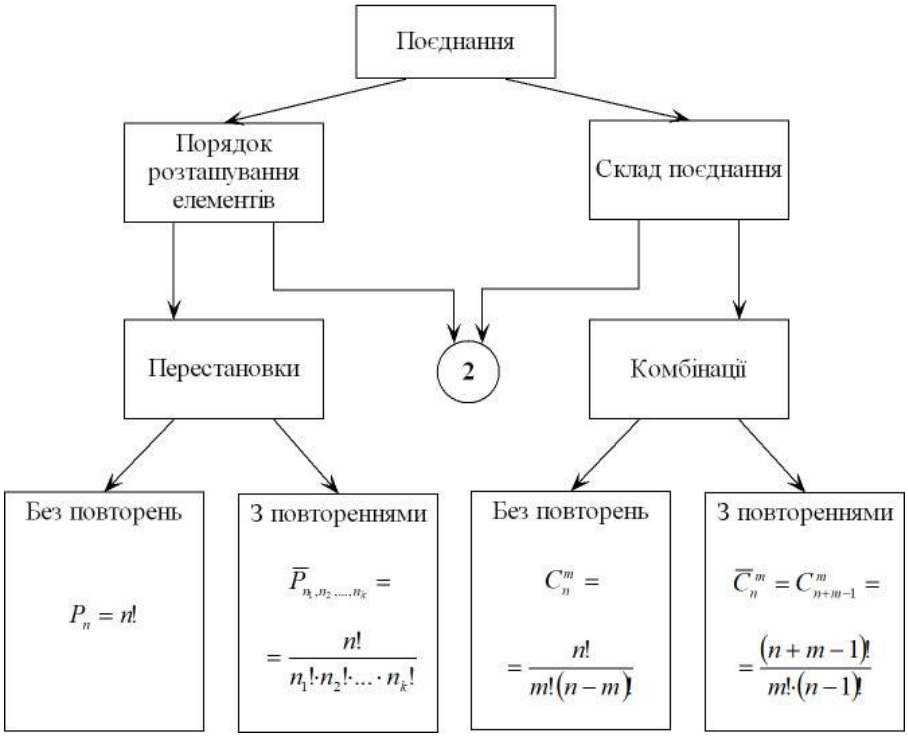

Рис. 6. Формули розрахунку кількості перестановок та комбінацій

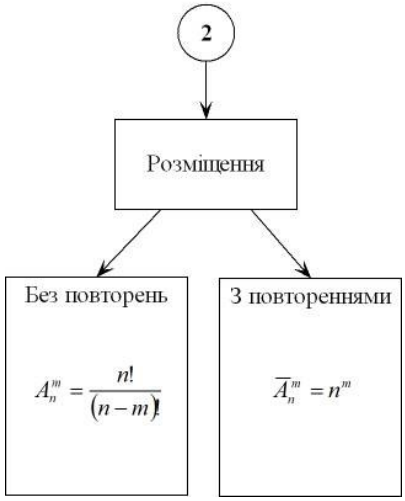

Рис. 7. Формули розрахунку кількості розміщень (продовження рис. 6)

Таблиця 1 узагальнює відповідні формули.

Таблиця 1

Формули підрахунку кількості комбінаторних об'єктів

Об'єкт	Без повторень	З повторенням
Перестановки	$P_n = n!$	$\overline{P} = \frac{n!}{k_1!\cdot\ldots\cdot k_l!}$
Розміщення	$A_n^m = \frac{n!}{(n-m)!}$	$\overline{A}_n^k = n^k$
Комбінації	$C_n^m = \frac{n!}{m!(n-m)!}$	$\overline{C}_n^k = \frac{(k+n-1)!}{(n-1)!\cdot k!}$

3. Алгоритми генерації комбінаторних об'єктів

Для практичної реалізації алгоритмів обрано мову програмування Python. Вибір зумовлений такими характеристиками мови: чіткий і виразний синтаксис, розвинена стандартна бібліотека, підтримка різних парадигм програмування (зокрема, функційної та об'єктно-орієнтованої), крос-платформеність [3, 13].

3.1 Рекурсивний та ітераційний підходи

Ітераційний підхід ґрунтується на циклічному виконанні певної послідовності дій. Аналіз складності таких алгоритмів зводиться до підрахунку ресурсоємності циклічних конструкцій.

Рекурсивний підхід – це такий спосіб організації обчислень, при якому функція звертається до самої себе для розв'язання задачі меншого розміру. Звернення до себе у ході виконання програми називають *прямим ходом рекурсії*, а послідовне повернення з «найглибшого» рекурсивного виклику – *зворотнім ходом* [1].

Рекурсивні алгоритми є природними для задач, де розв'язання зводиться до підзадач того ж типу. Однак вони, як правило, споживають більше оперативної пам'яті через накопичення стеку викликів і можуть бути перетворені на ітераційні.

Класичним прикладом рекурсії є обчислення факторіалу:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

3.2 Генератори та ключове слово yield у Python

Для ефективної генерації великих наборів комбінаторних об'єктів у Python доцільно використовувати *генератори* – спеціальні функції, що повертають значення по одному, не завантажуючи усі результати у пам'ять одночасно.

Ключове слово `yield` відрізняється від `return` тим, що після повернення значення виконання функції не завершується – при наступному зверненні воно продовжується з місця, де було зупинено. Це дозволяє реалізовувати потокову генерацію об'єктів без значних витрат пам'яті – суттєва перевага у задачах комбінаторики, де кількість об'єктів може бути дуже великою [13].

Для порівняння: звичайний список зберігає усі значення в оперативній пам'яті, тоді як генератор повертає їх по одному у міру потреби:

```
# список – зберігає усі значення в пам'яті
list_result = [x for x in range(10)]
# генератор – обчислює по одному
gen_result = (x for x in range(10))
```

3.3 Алгоритм генерації перестановок без повторення

Рекурсивний алгоритм генерує перестановки, послідовно додаючи до поточного префіксу `pref` черговий невикористаний елемент:

```
def recursion_permutations(N, M=None, pref=[]):
    M = N if M is None else M
    if M == 0:
        print(*pref, end=" ", sep="")
        return
    for number in range(1, N + 1):
```

```
        if number in pref:
            continue
        pref.append(number)
        recursion_permutations(N, M - 1, pref)
        pref.pop()
```

Логіка алгоритму: обираємо один елемент із доступних *N*, ставимо його на перше місце і рекурсивно генеруємо перестановки решти елементів. Параметр *M* вказує, скільки позицій залишилось заповнити. При *M* = *N* алгоритм генерує усі перестановки; при *M* < *N* – усі розміщення з *N* по *M* (рис. 8).

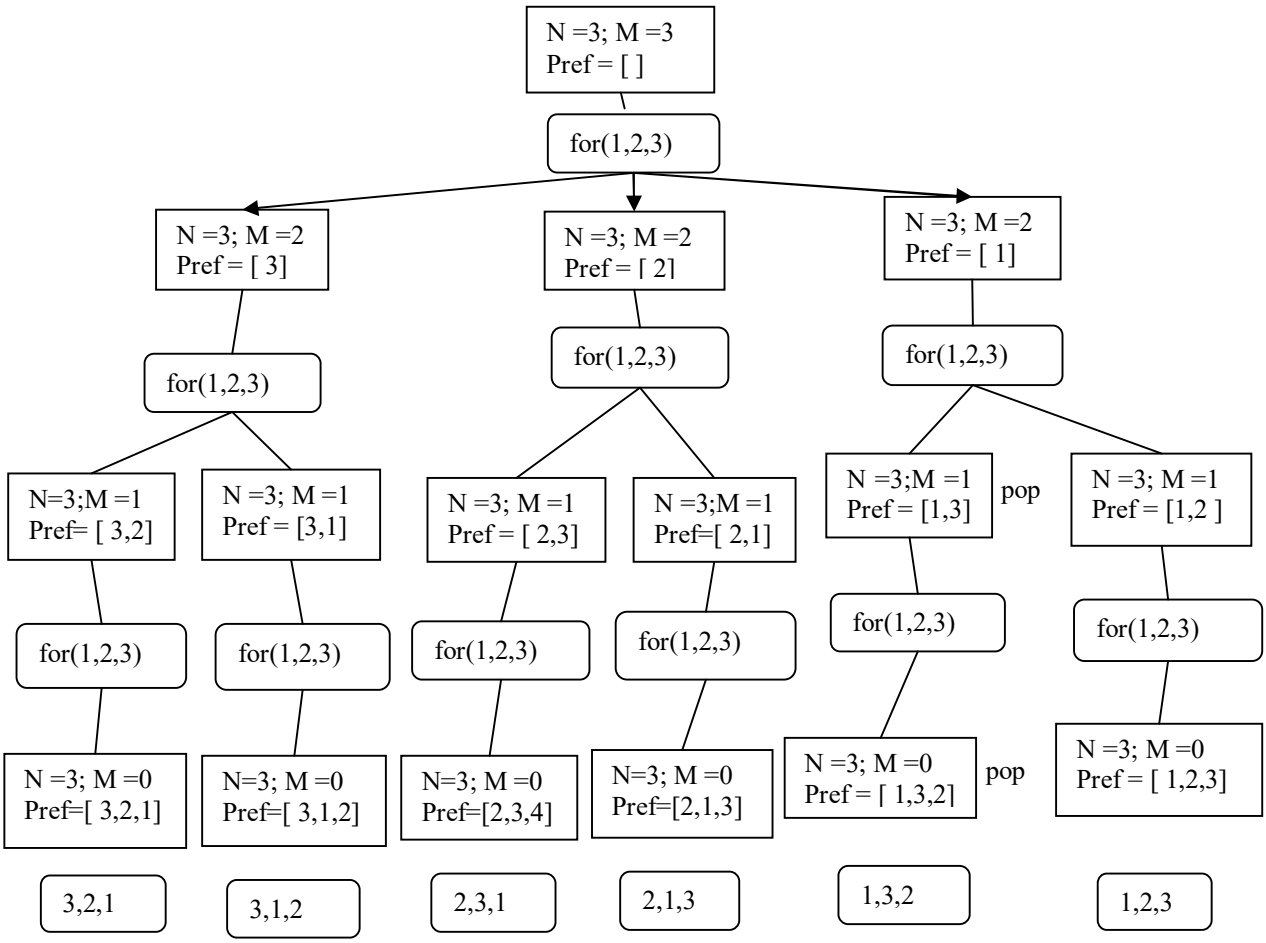

Рис. 8. Схема роботи алгоритму генерації перестановок для вхідних даних *N*=3

3.4 Алгоритм генерації розміщень без повторення

Оскільки розміщення з *N* по *M* відрізняються від перестановок лише тим, що кількість позицій *M* менша за *N*, достатньо передати попередньому алгоритму відповідне значення *M*. Таким чином, один рекурсивний алгоритм охоплює обидва випадки.

3.5 Алгоритм генерації комбінацій без повторення

Для генерації комбінацій реалізовано ітераційний підхід із використанням масиву індексів. Алгоритм починає з першої комбінації (елементи з індексами 0, 1, …, *r*-1) та послідовно переходить до наступної, збільшуючи індекси справа наліво:

```
def iter_combinations(iterable, r):
```

```
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i + 1, r):
            indices[j] = indices[j - 1] + 1
        yield tuple(pool[i] for i in indices)
```

Конструкція `for ... else` у Python є нестандартною: гілка `else` виконується лише тоді, коли цикл завершився без виклику `break`, тобто коли всі індекси досягли своїх максимальних значень – це сигнал, що усі комбінації вже згенеровано [13].

Також реалізовано рекурсивний варіант алгоритму з використанням `yield`. Незважаючи на те що він демонструє рекурсивний підхід, він менш ефективний через зайві «холості» рекурсивні виклики і наведений передусім для ілюстрації принципу.

3.6 Алгоритми генерації об'єктів із повторенням

Перестановки з повторенням. Оскільки алгоритм перестановок обробляє кожен елемент як унікальний, для отримання перестановок з повторенням достатньо передати список із повторюваними елементами та видалити дублікати з результату. У Python це реалізується за допомогою множин:

```
def permutation_with_repeat(lst):
    if len(lst) == 0:
        return []
    if len(lst) == 1:
        return [lst]
    result = []
    for i in range(len(lst)):
        m = lst[i]
        rem = lst[:i] + lst[i + 1:]
        for p in permutation_with_repeat(rem):
            result.append([m] + p)
    return result
# Видалення дублікатів
res = set([str(x) for x in permutation_with_repeat(lst)])
```

Розміщення з повторенням. Реалізуються ітераційним алгоритмом, що будує усі декартові добутки вхідного набору на себе *k* разів:

```
def placements_with_repeat(*args, repeat=1):
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x + [y] for x in result for y in pool]
    for prod in result:
```

```
        yield tuple(prod)
```

Комбінації з повторенням. Алгоритм структурно схожий на ітераційний алгоритм комбінацій, але умова зупинки та спосіб оновлення індексів змінені: індекс може повторюватись (кожен наступний індекс ≥ поточному), що відповідає можливості повторного вибору елементів [13].

4. Результати та аналіз

4.1 Порівняння рекурсивного та ітераційного підходів

Обидва підходи дають коректні результати, проте мають різні характеристики (табл. 2).

Таблиця 2

Порівняння рекурсивного та ітераційного підходів

Критерій	Рекурсивний підхід	Ітераційний підхід
Зрозумілість коду	Висока (природна декомпозиція)	Середня (потрібно відстежувати індекси)
Витрати пам'яті	Вищі (стек викликів)	Нижчі
Швидкодія	Нижча (накладні витрати викликів)	Вища
Придатність до великих N	Обмежена глибиною стеку	Краща
Підтримка yield	Так (через рекурсивний генератор)	Так (природна)

Для педагогічних цілей рекурсивний підхід є кращим, оскільки код наочно відображає математичне визначення об'єкта. Для практичних задач із великими вхідними даними перевагу слід надавати ітераційному підходу або генераторам.

4.2 Порівняння кількості об'єктів різних типів

Для ілюстрації швидкості зростання кількості комбінаторних об'єктів наведемо значення для деяких параметрів.

З таблиці 3 видно, що кількість об'єктів зростає надзвичайно швидко. Це підтверджує практичну обмеженість алгоритмів повного перебору і пояснює належність задач генерації до класу NP.

Таблиця 3

Кількість комбінаторних об'єктів залежно від параметрів

N	M	Перестановки P_n	Розміщення A_n^m	Комбінації C_n^m
3	2	6	6	3
4	2	24	12	6
5	3	120	60	10
6	3	720	120	20
7	4	5040	840	35
10	5	3628800	30240	252

4.3 Програмний продукт

Реалізований програмний продукт має графічний інтерфейс, побудований із використанням бібліотеки PyQt5 (обгортка PySide2) [2]. Інтерфейс поділено на чотири вертикальних секції (рис. 9):

1. Формули – відображення відповідної математичної формули для кожного типу об'єктів;
2. Кнопки – запуск обчислень для відповідного об'єкта;
3. Параметри – поля для введення n та k (або списку кратностей для перестановок із повторенням);
4. Результат – поле виведення усіх варіантів об'єкта та їхньої кількості.

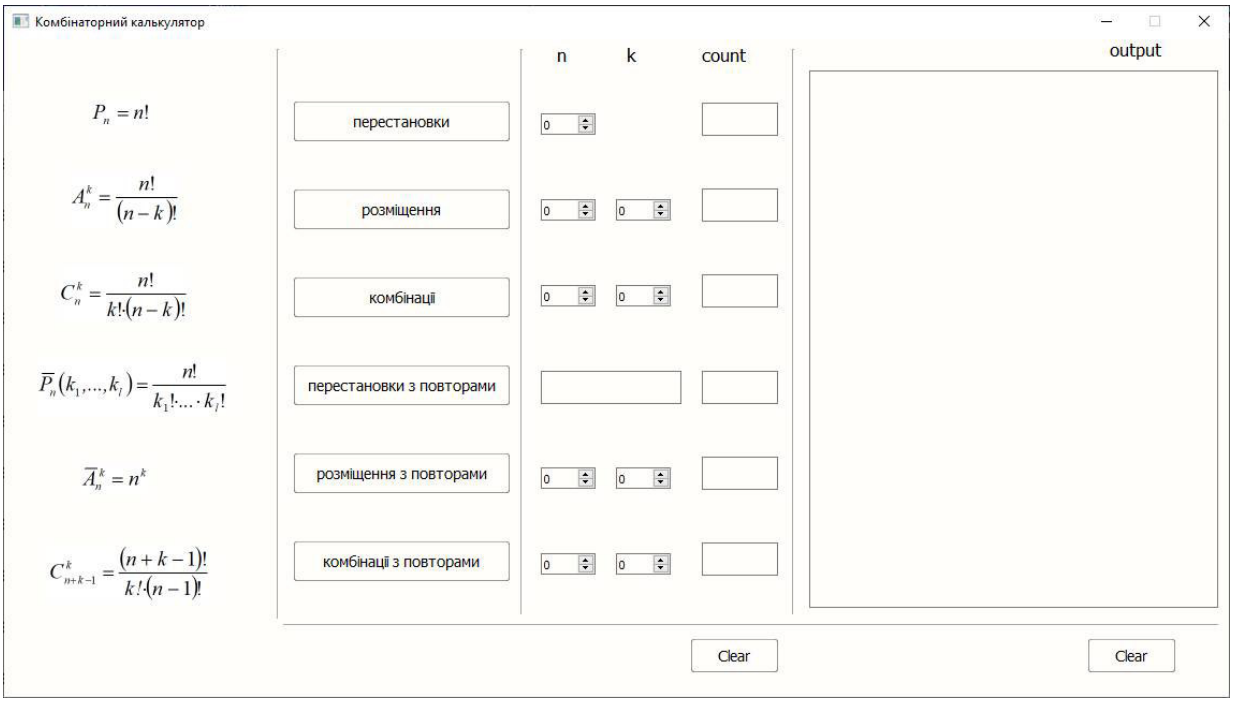

Рис. 9. Інтерфейс демонстраційної програми

Програма підтримує всі шість розглянутих типів об'єктів. Нижче наведено приклади результатів роботи.

4.4 Практичне значення

Практичне значення роботи полягає в тому, що розроблений програмний продукт може використовуватись як навчальний засіб на уроках математики та інформатики для ілюстрації понять комбінаторики, а також як калькулятор для швидкого отримання усіх варіантів комбінаторного об'єкта при невеликих значеннях параметрів.

Висновки

У статті розглянуто основні типи комбінаторних об'єктів: перестановки, розміщення та комбінації, а також їхні варіанти з повторенням. Для кожного типу наведено математичну формулу підрахунку кількості об'єктів та алгоритм їх генерації.

Реалізовано рекурсивні та ітераційні алгоритми мовою Python. Рекурсивний підхід є природним і зрозумілим, проте менш ефективним із погляду використання пам'яті при великих вхідних параметрах. Ітераційні алгоритми з генераторами є кращим вибором для практичних застосувань. Використання `yield` у Python є ефективним засобом потокової генерації великих наборів даних без надмірного навантаження на пам'ять.

Підтверджено, що задачі генерації комбінаторних об'єктів за своєю природою є NP-задачами: кількість об'єктів зростає факторіально або степенево і стає практично нездоланною вже при відносно невеликих значеннях параметрів (наприклад, 10! ≈

3.6 млн). Це пояснює, чому при розв'язанні прикладних задач оптимізації, що зводяться до перебору комбінаторних конфігурацій, застосовують евристичні та метаевристичні методи.

Розроблено програмний продукт із графічним інтерфейсом на базі бібліотеки PyQt5, який наочно демонструє роботу реалізованих алгоритмів і може застосовуватись у навчальному процесі.

Список використаної літератури

1. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms. – 4th ed. – Cambridge, MA: The MIT Press, 2022. – 1312 p.
2. PyQt5 5.15.10 [Електронний ресурс]. – Режим доступу: https://pypi.org/project/PyQt5/
3. Python Programming Language [Електронний ресурс]. – Режим доступу: https://www.python.org/
4. Real Python Tutorials [Електронний ресурс]. – Режим доступу: https://realpython.com/
5. Schmidt W.M. Diophantine Approximations and Diophantine Equations. – Berlin: Springer-Verlag, 1991. – 228 p.
6. Вигоднер І.В., Білоусова Т.П., Ляхович Т.П. Теорія ймовірностей та математична статистика: навчальний посібник. – Херсон: Гельветика, 2019. – 336 с.
7. Гаврилків В.М. Формальні мови та алгоритмічні моделі. – Івано-Франківськ: Голіней, 2023. – 180 с.
8. Зайцев Є. Теорія ймовірностей і математична статистика. – Київ: Алерта, 2013. – 440 с.
9. Костарчук В. М. Теорія графів та її застосування / В. М. Костарчук. – Київ : Вища школа, 1986. – 224 с.
10. Мартинюк О.М., Попіна С.Ю. Елементи комбінаторики й класичне означення ймовірності. – Тернопіль, 2003. – 40 с.
11. Diestel R. Graph Theory / R. Diestel. – 5th ed. – Hamburg : Springer, 2017. – 428 p.
12. Підручник з Python [Електронний ресурс]. – Режим доступу: https://docs.python.org/uk/3/tutorial/index.html

References

1. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2022) Introduction to Algorithms (4th ed.). Cambridge, MA: The MIT Press.
2. PyQt5 5.15.10. Available at: https://pypi.org/project/PyQt5/
3. Python Programming Language. Available at: https://www.python.org/
4. Real Python Tutorials. Available at: https://realpython.com/
5. Schmidt W.M. (1991) Diophantine Approximations and Diophantine Equations. Berlin: Springer-Verlag.
6. Vyhondner I.V., Bilousova T.P., Liakhovych T.P. (2019) Probability Theory and Mathematical Statistics: a textbook. Kherson: Helvetyka. [in Ukrainian]
7. Havrylkiv V.M. (2023) Formal Languages and Algorithmic Models. Ivano-Frankivsk: Holinei. [in Ukrainian]
8. Zaitsev Ye. (2013) Probability Theory and Mathematical Statistics. Kyiv: Alerta. [in Ukrainian]
9. Kostarchuk V. M. (1986) Graph Theory and Its Applications / V. M. Kostarchuk. – Kyiv : Vyshcha Shkola. [in Ukrainian]
10. Martyniuk O.M., Popina S.Yu. (2003) Elements of Combinatorics and the Classical Definition of Probability. Ternopil. [in Ukrainian]
11. Diestel R. (2017) Graph Theory. Hamburg : Springer.
12. Python Tutorial (Ukrainian). Available at: https://docs.python.org/uk/3/tutorial/index.html

KURYLO Oleksandr Mykolaiovych,
head, teacher of Computer Science at Zolotonosha General Education School of Levels I-III No. 5

NITSAK Vladyslav Kostiantynovych,
student of Zolotonosha General Education School of Levels I-III No. 5

RESEARCH AND SOFTWARE IMPLEMENTATION OF COMBINATORIAL OBJECT GENERATION ALGORITHMS

Summary. Introduction. *Combinatorics is one of the fundamental branches of discrete mathematics concerned with counting and constructing objects that satisfy given properties. Combinatorial objects – permutations, placements, and combinations – are widely used in probability theory, statistics, cryptography, and algorithm design. The generation of all possible configurations of such objects belongs to the class of NP problems, where the number of configurations grows factorially or exponentially with input size.*

Purpose. *The aim of this work is to systematize the mathematical foundations of the main combinatorial objects, describe and compare algorithms for their generation implemented in Python using recursive and iterative approaches, and develop a software product with a graphical interface for visual demonstration.*

Results. *The paper presents mathematical formulas for counting all six types of combinatorial objects (permutations, placements, and combinations, with and without repetition). Recursive and iterative generation algorithms are described and implemented in Python. The use of generators and the yield keyword is shown to provide an efficient approach to streaming generation of large sets of combinatorial objects. The recursive approach is more readable and natural but less memory-efficient; iterative algorithms with generators are preferable for practical use. A graphical application built with PyQt5 demonstrates the algorithms' results interactively.*

Conclusion. *The generation of combinatorial objects is inherently NP in nature: the number of objects grows unmanageably fast even for moderate input values. Both recursive and iterative algorithms yield correct results; the choice between them depends on the specific requirements of readability, memory efficiency, and the size of input data. The developed software product can be used as an educational tool in mathematics and computer science classes.*

Keywords: *combinatorics, permutations, placements, combinations, recursive algorithm, Python, generator.*

Одержано редакцією 12.11.2024 р.
Прийнято до публікації 11.12.2024 р.

УДК 378:517

DOI 10.31651/2076-5886-2024-1-45-56

PACS 01.40.Fk, 02.30.Hq

БОСОВСЬКИЙ Микола Васильович,
кандидат педагогічних наук, доцент,
доцент кафедри математики та методики навчання математики, Черкаський національний університет імені Богдана Хмельницького
e-mail: bosovskyy@gmail.com
ORCID: 0000-0003-1187-5550

СЕРДЮК Зоя Олексіївна,
кандидат педагогічних наук, доцент,
завідувач кафедри математики та методики навчання математики, Черкаський національний університет імені Богдана Хмельницького
e-mail: serdyuk_z@ukr.net
ORCID: 0000-0002-9376-4346

ТРЕТЯК Микола Васильович,
кандидат педагогічних наук, доцент,
доцент кафедри математики та методики навчання математики, Черкаський